國立清華大學
電機工程學系
一〇二學年度第二學期

EE2410
『資料結構』講義

教師: 黃錫瑜
syhuang@ee.nthu.edu.tw
Feb. ~ June, 2014

國立清華大學 電機工程學系
EE2410 Data Structure

Chapter 1
Basic Concepts

Outline

- **Overview**
  - **System Life Cycle**
- **Object-Oriented Software Design**
- **Data Abstraction and Encapsulation**
- **Basics of C++**
- **Algorithm Specification**
- **Performance Analysis and Measurement**

Why We Need Data Structure?

- Elementary programming course
  - Emphasizes syntax of a language
  - Solves small problems
  - Requires simple construct like array or while
- This course
  - Provides techniques to solve large-scale problem
  - Data abstraction, encapsulation, algorithm specification, performance analysis, and measurement are important
  - We discuss not just "data structure" but also "Algorithm"
  - Problem solving techniques are applied to not just software development, but also hardware or system-level design.

System Life Cycle

- There are five major phases
  - Requirements
  - Analysis
  - Design
  - Coding
  - Verification

Requirements

- **All Large Programming Projects**
  - **Begin with a set of specifications**
  - **Input**
  - **Output**
  - **Frequently the initial specifications are vague, need rigorous description.**

Question: 討論以下程式的可能輸入方式及方法 **(e.g.,**鍵盤、檔案、或網路 **?)**
(1) 最大組合數計算**: finding C(n, m)**
(2) 子串搜尋 **(e.g.,** 從網頁搜尋某個字串**)**
(3) 運算式 **(e.g., e = 24 + 8 * 5)** 之求值 → 可能需要語意解析程式 **(Parser)**

Analysis

- **The problem**
  - **Is broken down into manageable pieces**
- **Two are two major approaches**
  - **Top-down partitioning**
  - **Bottom-up integration**

Divide and conquer (D&C)
is an important algorithm design paradigm based on multi-branched recursion.
A divide and conquer algorithm works by recursively
breaking down a problem into two or more sub-problems of the same (or related) type,
until these become simple enough to be solved directly.
The solutions to the sub-problems are then combined
to give a solution to the original problem

Design

- The programmer designs
  - Abstract data object
  - Operations on those objects
- Example
  - Problem: scheduling system of a university
  - Typical data objects
    - Students, courses, professors
  - Typical operations
    - Inserting, removing, searching
- So far, the programming
  - Is language independent

Language Independent

- For example
  - The student data object includes
  - (1) Name
  - (2) Social security number
  - (3) permission number
  - (4) major
  - (5) phone number
  - But we haven’t decided what is used to implement the list of the students yet

Refinement and Coding

- **First,**
  - Choose representations for data objects
  - Important, since data object may determine the efficiency of the program
- **Then,**
  - Write algorithm for each operation

Verification

- **This phase is to prove correctness**
  - (1) Testing the problem with a variety of input data
  - (2) Debugging until no error exists
- **Good test data example**
  - A program with a *switch* statement
  - Test data should be chosen so each *case* branch is checked
- **Debugging practices**
  - Spaghetti code would be a nightmare when debugging
  - Test each unit then whole system
  - Documentation is useful

Outline

- **Overview**
  - **System Life Cycle**
- **Object-Oriented Software Design**
- **Data Abstraction and Encapsulation**
- **Basics of C++**
- **Algorithm Specification**
- **Performance Analysis and Measurement**

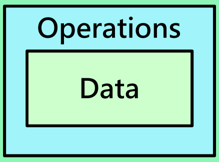
Object-Oriented Programming (OOP)

- **A fundamental change**
  - **From the structured programming design method**
  - **Divide-and-Conquer is still the principle**
  - **But how a project should be decomposed is different**
- **Traditional Programming**
  - **Views software as *process*, decomposed into functional modules**
- **OOP**
  - **Views software as a set of well-defined *objects***
  - **These objects interacts with each other to form a software system**

(1) 程式 = 運算流程 (Control Flow or Subroutines) +資料結構
(2) 早期的結構化程式以運算流程之設計為主，資料結構設計不易重覆使用
(3) 物件導向式語言：
希望寫程式像堆積木一般，而一塊塊的積木是一些容易重覆使用的物件 (Object)
物件 (Object) = 資料結構 (Data) + 一些運算副程式 (Operations)

Definitions of an Object

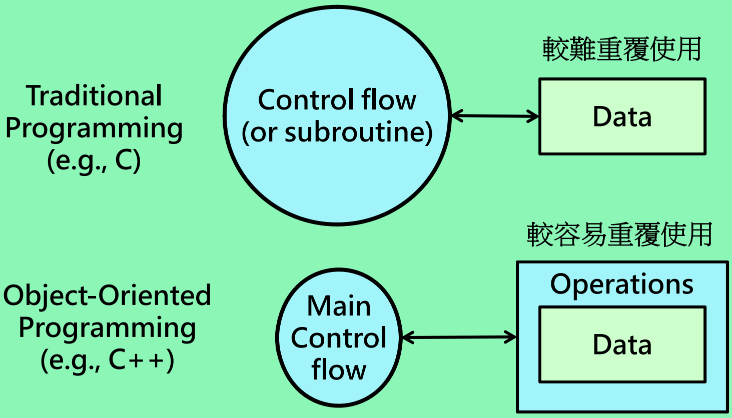
- **An object**
  - **Is an entity that performs computations and has a local state, a combination of**
  - **(1) data**
  - **(2) procedural elements (or called operations)**



- **Object-oriented programming**
  - **Is a method of implementation in which**
  - **(1) objects are the main building blocks**
  - **(2) each object is an instance of some type (or class)**
  - **(3) Classes are related to each other by inheritance relationships**

資料 (Object) 與資料型態 (Class) 的差別
Example: int *i, j, k*;

OOP – Control Flow vs. Data Structure



Ex: A class of “array” on which you can perform “insert”, “retrieve”, “delete” and “sort”. The elements of the array could be integer, floating point, etc.
➔ “Sorting” is originally part of the control flow, now it is part of the data structure

Object-Oriented Language

- **Three requirements**
  - **(1) It supports objects**
  - **(2) It requires objects to belong to a class**
  - **(3) It supports inheritance**

Comment: True OO programs → use inheritance

Inheritance 的語法範例: *class Stack: public Bag*
➔**“Bag”** 是一個已經定義好的物件類別 , **“Stack”** 是一個衍生出來的物件類別 ,
➔ **“Stack”** 可以繼承 **“Bag”** 裡已經有定義的 **operations and data**

Higher-Level Languages

- **First Generation**
  - **FORTRAN, noted as its ability to evaluate mathematical expressions**
- **Second Generation**
  - **Pascal and C, emphasize on effectively expressing algorithms**
- **Third Generation**
  - **Modula, introduces abstract data types**
- **Fourth Generation**
  - **Object-Oriented Languages, e.g., Smalltalk, Object C, and C++, emphasize inheritance**

Outline

- **Overview**
  - **System Life Cycle**
- **Object-Oriented Software Design**
- **Data Abstraction and Encapsulation**
- **Basics of C++**
- **Algorithm Specification**
- **Performance Analysis and Measurement**

Data Abstraction & Encapsulation

- **Consider a DVD Player**
  - **(1) The manual tells what the player is supposed to do, instead of how it does it, this is called data abstraction** (資料抽象介面 →介面運算的輸出入格式應與內部演算法無關，方便日後演算法的更新)
  - **(2) The internal representation is hidden from the users, this is called encapsulation** (資料包裹性 → 存取不能隨意，必須透過介面的 **Operations**)

Definitions

- **Data Abstraction**
  - Is the **separation** between the **specification** of a data object and its **implementation**
- **Data Encapsulation**
  - Or information hiding is the **concealing of the implementation details** of a data object from the outside world

Fundamental Data Types of C++

- **Basic Types**
  - char, int, float, double
- **Modification keywords**
  - short, long, signed, unsigned
- **Grouping of Basic Types**
  - array, struct, and class
- **User-Defined Type**
  -

```
typedef  struct _int_pair {
            int first_num;
            int second_num;
 } int_pair;
```

使用者自訂複雜資料結構的兩大關鍵字:
struct , *class*
但是 *struct* 所定義的結構不能有資料包裹性
i.e., *class* 才能定義真正的 OOP資料結構

Example Data Type *int*

- **Objects**
  - {0, +1, -1, +2, -2, …, MAXINT, MININT}
- **Operations**
  - + - * /
- **Abstract Data Type (ADT)**
  - A data type organized in a way that the specification is separated from the implementation

---

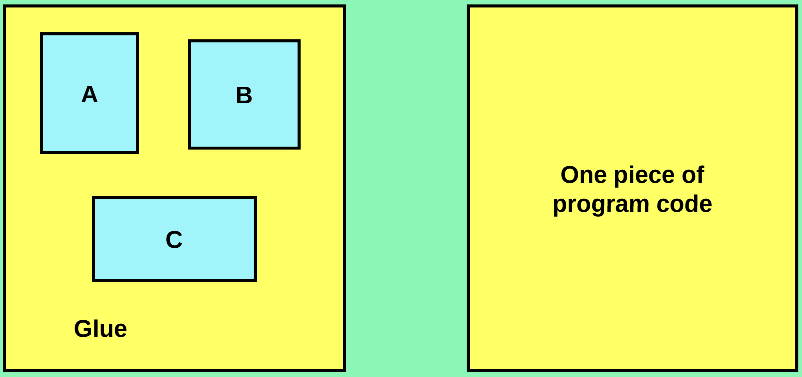
Abstract Data Type *NaturalNumber*

```
ADT NaturalNumber is
  objects: An ordered subrange of the integers starting at zero and
           ending at the maximum integer (MAXINT) on the computer
  functions:
      for all x, y ∈ NaturalNumber; TRUE, FALSE ∈ Boolean
      and where +, -, <, ==, and = are the usual integer operations

      Zero(): NaturalNumber          :=     0
      IsZero(): Boolean              :=     if(x==0) IsZero = true
                                            else IsZero = false
      Add(x, y): NaturalNumber       :=     if(x+y<=MAXINT) Add = x+y
                                            else Add = MAXINT
      Equal(x, y): Boolean           :=     if(x==y)  Equal = true
                                            else Equal = false
      Successor(x): NaturalNumber :=        if(x==MAXINT) Successor = x
                                            else Successor = x + 1
      Subtract(x,y): NaturalNumber :=       if(x<y) Subtract = 0
                                            else Subtract = x – y
end NaturalNumber
```

The Advantages of ADT (I)

- **(1) Development Style**

A, B, C are three abstract data types



A

B

C

Glue

One piece of program code

Good

Not so good !

The Advantages of ADT (II)

- **Testing and Debugging**
  - Programming styles with ADT is easier to debug
  - For example, if every ADT has been tested okay, then only the glue is checked if bugs found during integration
- **Reusability**
  - Data abstraction gives rise to reusability
- **Easier-to-modify**
  - Changes of a data type is localized, I.e., the rest of the program needs not be changed accordingly.

Problem of Not Using Data Encapsulation

- **Consider a program**
  - That directly accesses internal implementation of the data type
- **Suppose a change**
  - Is made to the data type (即直接性資料存取！)
- **Modification is laborious**
  - Exhaustive search for instances that access the modified data type and then made appropriate changes – A nightmare !
  - (這是直接性資料存取的重大缺點)

Overhead of ADT

- **Program is slower**
  - Direct data access versus subroutine invocation
  - This is the main reason that C is still in widespread use
- **Coding is more tedious (但這通常是值得的)**
  - A lot of simple data-access member functions need to be created

Outline

- **Overview**
  - System Life Cycle
- **Object-Oriented Software Design**
- **Data Abstraction and Encapsulation**
- **Basics of C++**
- **Algorithm Specification**
- **Performance Analysis and Measurement**

Multiple File Program

- **Development Cycle**
  - **Each source C++ file is compiled individually, producing an object file**
    - `% g++ –c –I./include file.c –o file.o`   % 是作業系統的提示符號
  - **All object files are linked together, along with other binary library, producing a binary executable file**
    - Assume that there is a library file called ./lib/libmymlib.a
    - Linking → `% g++ file1.o file2.o –L./lib –lmylib –o prog`
  - **Execute the program**
    - `% prog <with command line arguments>`
    - Example: `% prog –A –e 10`

Pre-Processor Directive

- **The Header Files (for 具有許多原始檔案的程式)**
  - Are mostly included at the beginning of **each source file**
  - Inclusion of a header file for multiple times creates a **compilation errors**
  - The following **pre-processor directives** can be used to avoid the above errors

```
#ifndef FILENAME_H
#define  FILENAME_H
// insert contents of the header file here
.
.
.
#endif
```

Utility *Make*

- **Purpose of *Make***
  - To **manage the compilation and linking** of a large software consisting of multiple files
  - Avoid typing compilation commands repeatedly
- **Procedure of Using *Make***
  - **Step1**: create a file called "**Makefile**"
  - **Step2**: type in "**make**" each time when any source file or header file is modified. In response to this command, **only the files modified will be recompiled**, while the others are left intact.

Example of Makefile

```
#---------- define macro names ----------
CC=g++ –g
SOURCE = file1.c file2.c
HEADER = project.h
OBJ = $(SOURCE: .c=.o)

#---------- perform linking when any source file or object file is changed -----
linking: $(SOURCE) $(HEADER) $(OBJ)
<tab>$(CC) $(OBJ) –o ./prog
```

Scope of Variables

- **A Variable is only visible within its scope**
- **Four Types of Scopes**
  - **(1) Global Scope: Variables that are available throughout the entire program**
  - **(2) File Scope: declarations not in a function definition or in a class definition**
  - **(3) Local Scope: Labels used within the function definition**
    (4) Class Scope: Declarations associated with a class definition

Example C++ Program

```
#include <iostream.h>

char course_name[100] = “data structure”;     A file-scope variable

main()
{
  int a = 84;   a is a local variable
  printf(“Welcome to %s\n”, course_name);
  printf(“n is %d, n+1 is %d\n”, a, add_one(a));
}

Int add_one(int b)   b is an input argument
{
   int  c;   c is a local variable
   printf(“A subroutine for %s\n”, course_name);
   c = b + 1;
   return(c);
}
```

Global Variables

- **Problem**
  - A global variable defined in file1.C, and to be also used in file2.C
  - → Use ***extern*** to declare the variable in file2.C

```
#ifdef MAIN  /* macro MAIN is defined in file1.C */
   int  global_variable;
#else
   extern int global_variable /* declare extern in all other files */
#endif
```

Example C++ Program – Global Variable

Source File 1

```
#include <iostream.h>

char course_name[100] = “data structure”;

main()
{
  int a = 84;
  printf(“Welcome to %s\n”, course_name);
  printf(“n is %d, n+1 is %d\n”, a, add_one(a));
}
```

Source File 2

```
#include <iostream.h>

extern char course_name[100] = “data structure”;

Int add_one(int b)
{
   printf(“A subroutine for %s\n”, course_name);
   return(b+1);
}
```

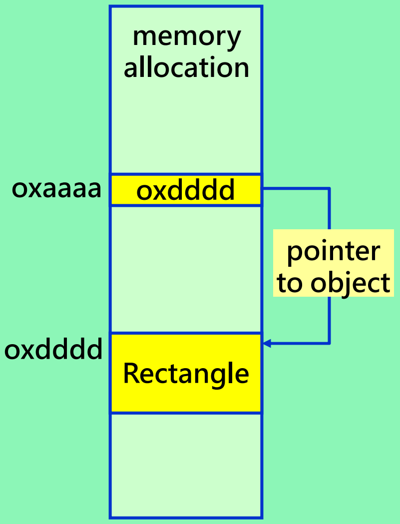
C++ Statement and Operators

- **Dynamic Memory Management**
  - ***“new”*** **and** ***“delete”***
- **Input/Output**
  - **Uses shift left (<<) and shift right (>>) operators**
- **Operator Overloading**
  - **An operator could have multiple functions, depending on the types of operands that it is being applied to**

Data Declaration in C++

- **(1) Constant Value**
- **(2) Variables**
- **(3) Constant Variable**
  - **const int MAX = 500;**
- **(4) Enumeration types**
  - **enum *Boolean* {FALSE, TRUE};**
- **(5) Pointers**
  - **Hold memory addresses of objects**
  - **int *i* = 25;**
  - **int **np;*** — ***np* is a pointer to an integer, where * is like “taking content”**
  - ***np* = *&i;*** — ***np* points to the location of *i*, where & is like “taking address”**

Object vs. Pointer



```
Rectangle r, s;
Rectangle *t = &s;
```

symbol table

name	type	address
s	Rectangle	oxdddd
t	pointer	oxaaaa

Data Declaration in C++ (con't)

- **(6) Reference types**
  - A unique feature of C++, (which is not available in C)
  - Is a mechanism to provide an alternative name for an object
  - Example

```
int i=5;
int& j=i;
i=7;
printf("i=%d, j=%d", i, j); → both i and j are 7;
```

Outputs in C++

```
#include <iostream.h>

main()
{
  int n=50; float f=20.3;
  cout << "n:" << n << endl;
  cout << "f:" << f << endl;
}

(結果)
n: 50
f: 20.3
```

Inputs in C++

```
#include <iostream.h>

main()
{
  int a, b;
  cin >> a >> b;
}

(結果)
input1:
5 10 <enter>
→ will set a=5; b=10;
```

File IO in C++

```
#include <iostream.h>
#include <fstream.h>

main()
{
  ofstream outFile(“my.out”, ios::out);
  if(!outFile) {
    cerr << “cannot open my.out” << endl; // standard error device
    return;
  }
  int n=50; float f=20.3;
  outFile << “n: “ << n << endl;
  outFile << “f: “ << f << endl;
}
```

Functions in C++

- Two kinds of functions
  - (1) Regular functions (非附屬於物件內的副程式)
  - (2) Member functions associated with a class
- A function consists of
  - Name
  - A list of arguments, also called input signature
  - A return type (output)
  - The body

```
int max(int a, int b)
{
   if(a>b) return a;
   else     return b;
}
```

Parameter Passing in C++

- (1) Pass by value (傳值呼叫)
  - Default mechanism
  - When an object is passed by value → it is copied into the function's local storage
  - could be slow when data to be passed is large !
- (2) Pass by reference (傳地址呼叫)
  - Done by appending an & to its type specifier
  - E.g., `int max(int& a, int& b);`
  - When an object is passed by reference → only the address of its location is copied into the function's local store
  - faster but less secure !

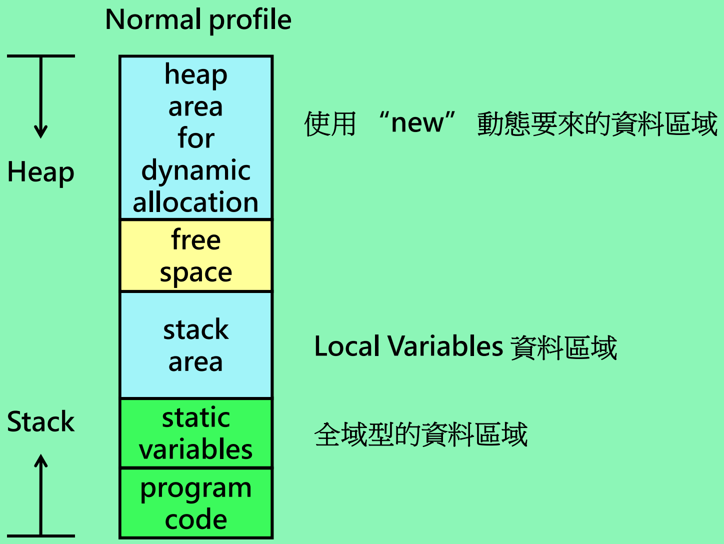
Call By Pointer Example

```
main()
{
  int i, j;
  cout << "Input 2 numbers:" << endl;
  cin >> i >>j;
  if( i > j )
            swap(&i, &j);
            cout << "The smaller number is " << i << endl;
            cout << "The larger is " << j << endl;
};

void swap(int *ptr_x, int *ptr_y)  // call by pointer
{
   int temp;
   temp = *ptr_x;
   *ptr_x = *ptr_y;
   *ptr_y = temp;
}
```

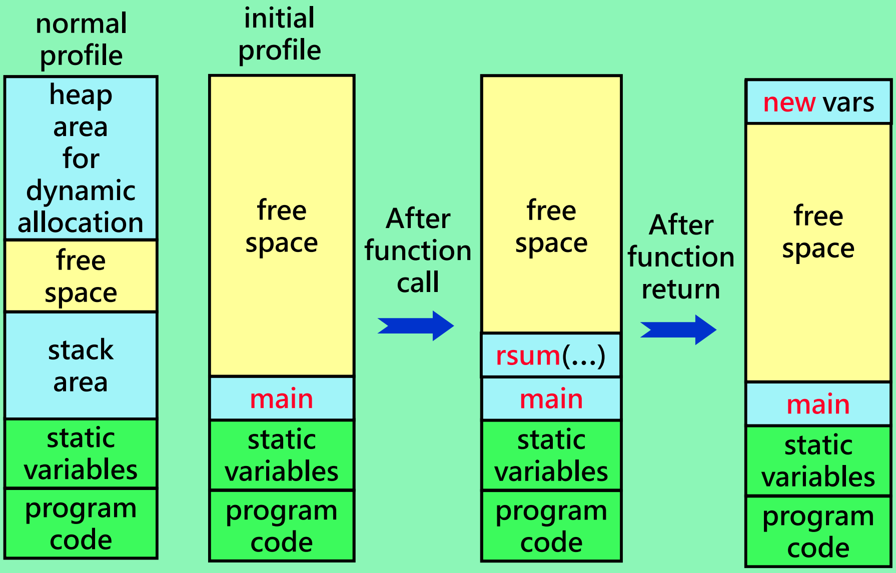
Call By Reference Example

```
main()
{
  int i, j;
  cout << "Input 2 numbers:" << endl;
  cin >> i >>j;
  if( i > j )
            swap(i, j);
            cout << "The smaller number is " << i << endl;
            cout << "The larger is " << j << endl;
};

void swap(int &x, int &y)  // call by reference
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

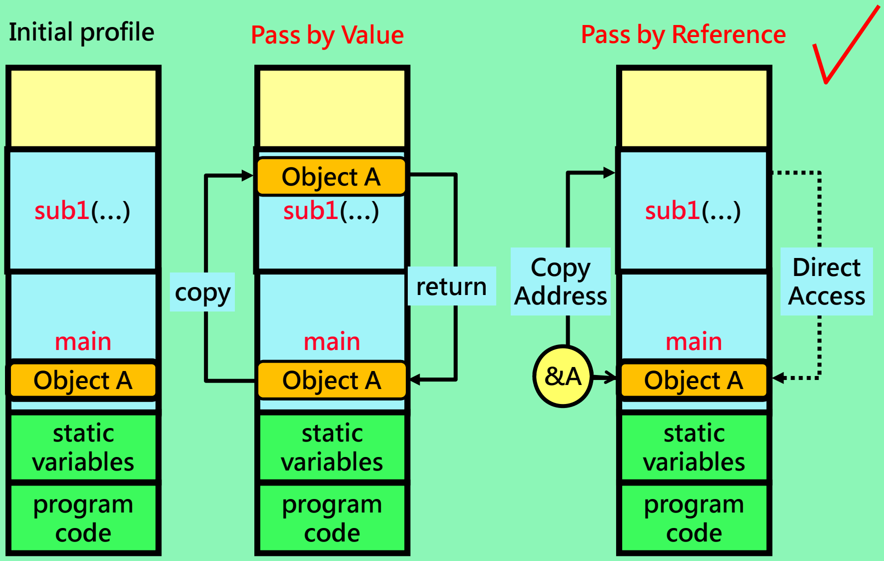
Memory Allocation



Memory Allocation – Subroutine Invocation



Pass-by-Value vs. Pass-by-Reference



Initial profile

Pass by Value

Pass by Reference

✓

sub1(...)

main

Object A

static variables

program code

copy

Object A

sub1(...)

return

main

Object A

static variables

program code

Copy Address

&A

sub1(...)

Direct Access

main

Object A

static variables

program code

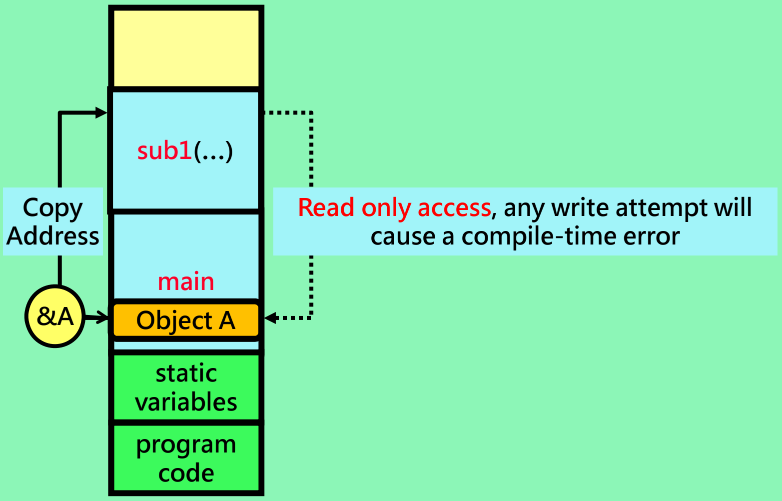
Pass by Const References

- **A Best Method**
  - **pass by "const T& *a*", T is the type of the argument *a***
  - **Faster than pass-by-value if a large chunk of arguments to be passed**
  - **Better protection of the actual arguments to be passed**
  - **Any attempt to modify a *const* argument in the function body will result in a compile-time error**

Improper manipulations of the input arguments
→ could lead to nasty bugs

Illustration:
Pass by Const References

Pass by Constant Reference



sub1(…)

main

Object A

static variables

program code

Copy Address

&A

Read only access, any write attempt will cause a compile-time error

---

One Exception

- **Array**
  - **Does not pass by value**
  - **I.e., it is not copied to the function’s local store**
  - **Only the pointer of the first element is passed**
  - **Function is not aware of the size of the array**
  - **Often the size of an array is also passed as another argument**

例子: A subroutine that sorts an array of *n* integer elements
Subroutine 結構如下:
float *sorting*(float **a*, const int *n*) {
// where *a* is the array name
…..
}

Function Name Overloading

Function over-loading: there can be more than one functions with the same name as long as they have different signatures

```
Int max(int, int);
Int max(int, int, int);
Int max(int*, int);
Int max(float, int);
Int max(int, float);
```

InLine Function

```
Inline int sum(int a, int b)
{
  return (a+b);
}
```

Inline function can eliminate the use of certain preprocessor directives such as #define, which is traditionally used for macro substitution

→ Excessive use of pre-processors make it harder to use debugger or profiler

Dynamic Memory Allocation

- **New**
  - **This operator creates an object of the desired type and return a pointer to the data type that follows it.**
  - **It returns 0 if not being able to create it**
- **Delete**
  - **Free the data allocated by “new” operator**

```
int *ip = new int;
If(ip==0) cerr << “Memory not allocated” << endl
.
.
delete ip;
```

Creating An Array

```
int *jp=new int[10];
if(jp==0) cerr << “Memory not allocated” << endl
.
.
delete [ ] jp;

/* The operator [ ] is used to inform the compiler that
the object being created or deleted is an array
```

Outline

- **Overview**
  - **System Life Cycle**
- **Object-Oriented Software Design**
- **Data Abstraction and Encapsulation**
- **Basics of C++**
- ➡ **Algorithm Specification**
- **Performance Analysis and Measurement**

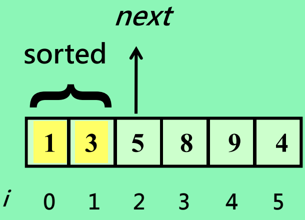
Definition

- **Algorithm**
  - **Is a finite set of instructions that, if followed, accomplishes a particular task.**
- **Criteria**
  - **Input**
  - **Output**
  - **Definite: each instruction is clear and unambiguous**
  - **Finiteness: for all cases, the algorithms terminate after a finite number of steps**
  - **Effectiveness: each instruction must be basic enough**

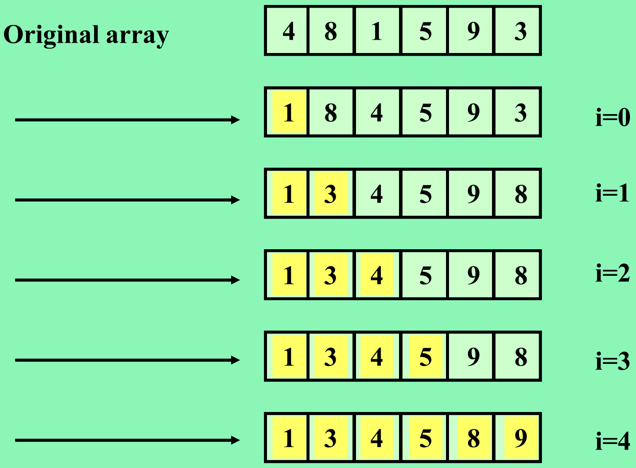
Example: Selection Sort

- **Problem**
  - To sort a collection of n ≧1 integers
- **A Solution**
  - From those integers that are currently unsorted, find the smallest and place it next in the sorted list
- **Selection Sort Algorithm**

```
for(int i=0; i<n; i++) {
  // Fixing the i-th smallest element
  examine a[i] to a[n-1] and suppose the smallest
  integer is at a[j]; // a[j] is the i-th smallest element
  interchange a[i] and a[j];
}
```



Example of Selection Sort



Selection Sort Algorithm

```
1.  void sort (int *a, const int n)
2.  // sort the n integers a[0] to a[n-1] into non-decreasing order
3.  {
4.     for(int i=0; i<n; i++){
5.          // find the smallest integer from a[i] to a[n-1];
6.          int smallest_index = i;
7.          for(int k=i+1; k<n; k++) {
8.                if (a[k] < a[smallest_index]) smallest_index = k;
9.          }
10.         // interchange
11.         int temp=a[i]; a[i]=a[smallest_index];
12.               a[smallest_index ]=temp;
13.    }
14. }
```

The upper limit index of the "for loop" in line 4 can be changed to *n*-1 without damaging the correctness of the algorithm

---

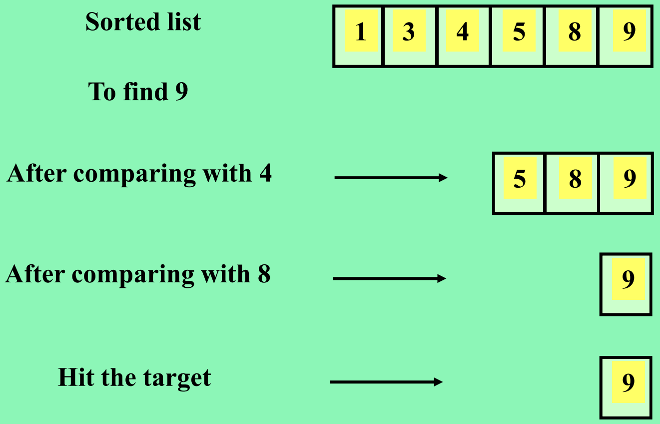
Binary Search

- **Problem**
  - Assume that we have $n \geqq 1$ distinct integers that are already sorted in array a[0],…a[n-1]
  - Determine if an integer x is present, if so, return its index

A sub-routine *compare*

```
char compare(int x, int y)
{
   if (x>y) return '>';
   else if (x<y) return '<';
   else  return '=';
} // end of compare
```

Example of Binary Search



Sorted list 1 3 4 5 8 9

To find 9

After comparing with 4 → 5 8 9

After comparing with 8 → 9

Hit the target → 9

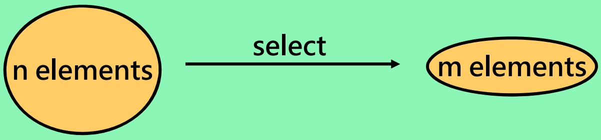
C++ Code for Binary Search

```
binary_search (int *a, const int x, const int n)
// search for the sorted array a[0],…,a[n-1] for x
{
  for(int left=0, int right=n-1; left <= right;)
  {
    middle = (left+right) /2;
    switch(compare(x, a[middle]){
      case '>': left = middle+1; break;
      case '<': right = middle-1; break;
      case '=': return middle;
    } // end of switch
  } // end of for
  return –1;
} // end of binary search
```

left middle right

Recursive Algorithms

- **Recursion**
  - Is similar to the method of induction which is often used to prove mathematical statements
  - (1) A basis is needed
  - (2) A terminating condition is needed
- **Applications**
  - Recursion is particularly suitable for problem recursively defined
  - E.g., Factorial n!
  - E.g., Binomial coefficient C(n,m) = C(n-1,m) + C(n-1, m-1);

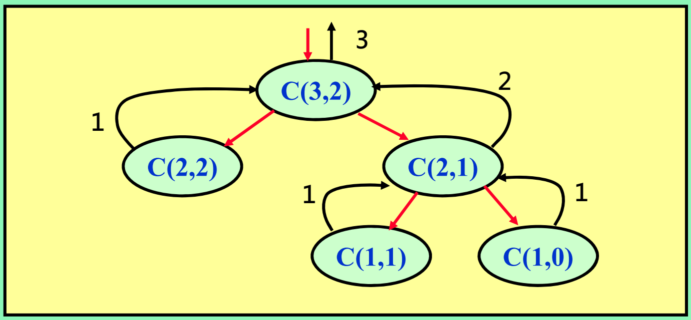


$$\frac{n!}{(n-m)!\ m!}$$

Recursive Binomial Coefficient

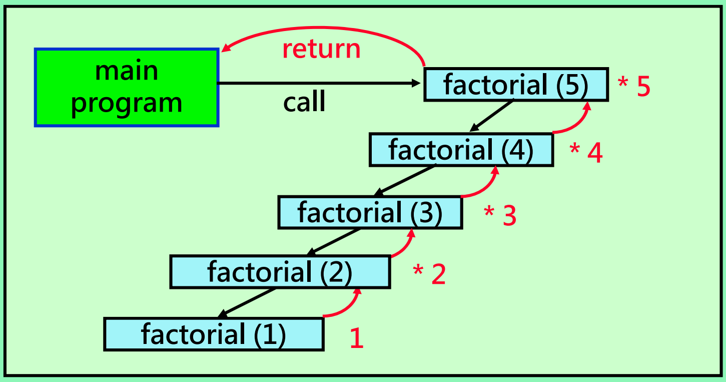
- C(n, m) = C(n-1, m) + C(n-1, m-1);

```
int binomial(int n, int m)
{
   if( n < m ) exit(-1);
   if(n==m) return(1);    if(m==0) return(1);
   return( binomial(n-1, m) + binomial(n-1, m-1) ) ;
}
```



Recursive Factorial

```
factorial (int n)
{
    if(n==1)   return (1);
    else         return( factorial(n-1) * n);
}
```



Recursive Binary Search

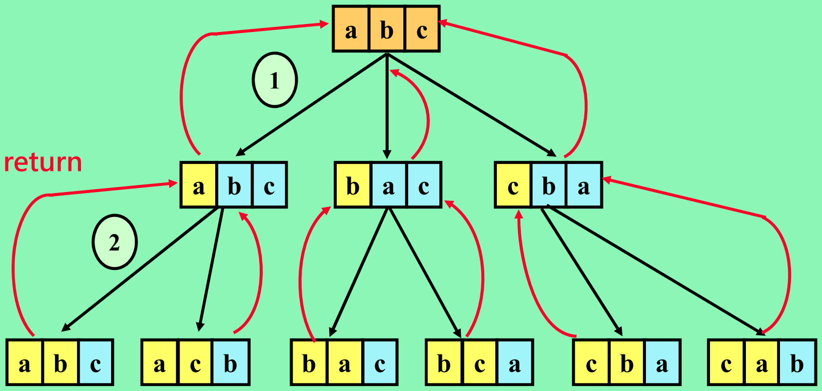
```
Recursive_BS(int *a, const int x, const int left,
                                        const int right)
// search for the sorted array a[left],…,a[right] for x
{
  if(left <= right) {
     int middle = (left+right) /2;
     switch(compare(x, a[middle]){
        case '>':
            return(Recursive_BS(a, x, middle+1, right));
        case '<':
            return(Recursive_BS(a, x, left, middle-1));
        case '=': return middle;
     }
  }
  return –1;
}
```

Permutation

- **Example**
  - A set of symbols {a, b, c}
  - All possible number of permutations is n!
  - {(a, b, c), (a, c, b), (b, a, c), (b, c, a), (c, a, b), (c, b, a)}
- **Recursive Permutation of {a, b, c, d}**
  - {a, permutation of (b, c, d)}
  - {b, permutation of (a, c, d)}
  - {c, permutation of (a, b, d)}
  - {d, permutation of (a, b, c)}

Demo of Recursive Permutation

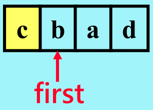
Original configuration



Maximum recursion depth = 2

Recursive Permutation Generator

```
void perm (char *a, const int first, const int n)
// generate all permutations of a[first],…,a[n-1]
// first - the first element in the undecided region
{
  if(first==n-1) {  // terminating condition
    for(int i=0; i<n; i++) cout << a[i] << “ “;
    cout << endl;
  }
  else {
     for (i=first; i<n; i++) {
        char temp=a[first]; a[first]=a[i]; a[i]=temp;
        perm(a, first+1, n);
        temp=a[first]; a[first]=a[i]; a[i]=temp;
        // return to original configuration
     }
  }
}
```



Program 1.11

Outline

- Overview
  - System Life Cycle
- Object-Oriented Software Design
- Data Abstraction and Encapsulation
- Basics of C++
- Algorithm Specification
- **Performance Analysis and Measurement**

Criteria of Judging a Program

1. Is it functioning?
2. Speed (i.e., CPU time)
3. Space (i.e., memory requirement)
4. Documentation
5. Readability

Complexity

- Space Complexity
  - The amount of memory a program needs to run to complete
- Time Complexity
  - The amount of computer time a program needs to run to complete
- Performance Analysis
  - To estimate a program's run time
- Performance Measurement
  - To actually measure a program's run time

Space Requirement

- **Fixed Part**
  - Instruction space, space for variables and constants
- **Variable Part**
  - Depends on instance characteristics, and the recursion stack space
  - This part is more important
- **Space requirement of a program P**
  - $S(P) = c + S_P$ (instance characteristics)

  c is a constant and S_P is a function of the problem size

可以簡單的把 instance characteristic 想成 problem size 即可

Example: Space Complexity

```
float abc(float a, float b, float c) {
      return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

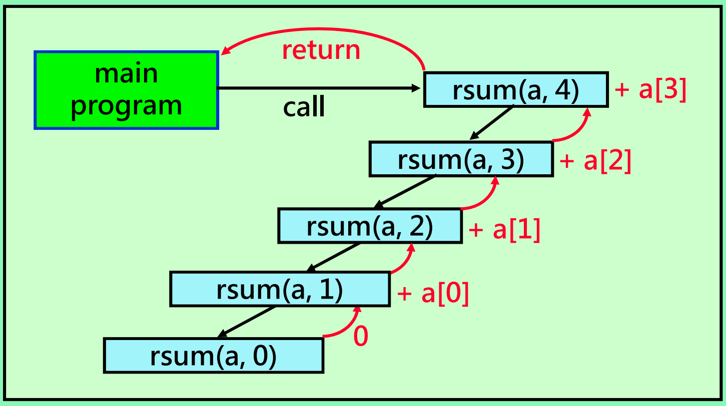
S_P(instance characteristics) = 0;
That is, space is independent of the instance characteristics

```
float sum(float *a, const int n) {
      float s=0;
      for(int i=0; i<n; i++){
            s += a[i];
      }
}
```

S_P(instance characteristics) = n;

Recursive Summation

```
float rsum(float *a, const int n) {
      if(n<=0)  return 0;
      else        return(rsum(a, n-1) + a[n-1]);
}
```



Example: Space Complexity

```
float rsum(float *a, const int n) {
      if(n<=0)  return 0;
      else  return(rsum(a, n-1) + a[n-1]);
}
```

(1) Instance characteristics = *n*

(2) Each call to *rsum* requires at least 4 words

space for *a, n,* the return value, and the return address

(3) The depth of recursion is *n*+1

(4) The recursion stack space is 4(*n*+1)

(5) For *n* = 1000 → stack space is 4004

Time Complexity

- **Total Time = Compile Time + Run Time**
- **Run Time is of more concern**
  - **t_P(instance characteristics)**
- **A program step**
  - **syntactically or semantically meaningful segment of a program**
- **For example**
  - **`return(a+b+b*c+(a+b-c)/(a+b)+4.0;)` can be regarded as a step**

    → because it is independent of (instance characteristics)

Step Counting (僅供參考)

- **(1) Comments: 0**
- **(2) Declarative statements: 0**
  - **int, long, short, char, float, double, const, enum, signed, unsigned, static, extern**
  - **class, struct, union, template**
  - **private, public, protected, friend**
  - **void, virtual**
- **(3) Expression and Assignments: 1**
- **(4) Iteration Statements (for, while, do): <iteration-count>**
- **(5) Switch statements:**
- **(6) If-else statements:**
- **(7) Function invocation: 1**
- **(8) Memory management statements: 1**
- **(9) Jump statements (break, return): 1**

```
for( <init-stmt>; <expr1>; <expr2>)
while <expr> do
do … while <expr>
switch <expr>{
      case cond1: <statement1>
      …
}
```

Example: Step-Counting

```
float sum(float *a, const int n)
{
   float s=0;
   count++; // count is global
   for(int i=0; i<n; i++){
         count++; // for for
         s += a[i];
         count++; // for assignment
   }
   count++; // for last time of for
   count++; // for return
   return s;
}
```

```
void sum(float *a, const int n)
{
    for(int i=0; i<n; i++){
          count += 2;
    }
    count += 3;
}
```

Example: Step Counting For Recursive Program

```
float rsum(float *a, const int n)
{
   count++; // for if conditional
   if(n <= 0){
         count++; // for return
         return 0;
   }
   else{
       count++; // for return
       return(rsum(a, n-1) + a[n-1]);
   }
}
```

recurrence relation for n>0

$$
\begin{aligned}
t_{rsum}(n) &= 2 + t_{rsum}(n-1) \\
&= 2 + 2 + t_{rsum}(n-2) \\
&= 2 * 2 + t_{rsum}(n-2) \\
&= \ldots \\
&= 2n + t_{rsum}(0) \\
&= 2n + 2
\end{aligned}
$$

solved by repeated substitution

Example: Matrix Addition

matrix addition with counting

```
void add (matrix a, matrix b, matrix c, int m, int n)
{
   for (int i = 0; i < m ; i++)
   {
      count++; // for for i
      for (int j = 0; j < n; j++)
      {
         count++; // for for j
         c [i ][j ] = a [i ][j ] + b [i ][j ];
         count++; // for assignment
      }
      count++; // for last time of for j
   }
   count++; // for last time of for i
}
```

Simplified version

```
line  void add (matrix a, matrix b, matrix c, int m, int n)
 1  {
 2     for (int i = 0; i < m; i++)
 3     {
 4        for (int j = 0; j < n ; j++)
 5           count += 2;
 6        count += 2;
 7     }
 8     count++;
 9  }
```

Tabular Method for Iterative SUM

```
1.   float sum(float *a, const int n) {
2.        float s=0;
3.        for(int i=0; i<n; i++){
4.             s += a[i];
5.        } return s;
6.   }
```

line	s/e	frequency	total steps
1	0	1	0
2	1	1	1
3	1	$n+1$	$n+1$
4	1	n	n
5	1	1	1
6	0	1	0
Total number of steps			$2n + 3$

s/e: steps per execution

Tabular Method for Recursive SUM

```
1.  float rsum(float *a, const int n) {
2.      if(n<=0)  return 0;
3.      else      return(rsum(a, n-1) + a[n-1]);
4.  }
```

line	s/e	frequency		total steps	
		$n=0$	$n>0$	$n=0$	$n>0$
1	0	1	1	0	0
2(a)	1	1	1	1	1
2(b)	1	1	0	1	0
3	$1+t_{rsum}(n-1)$	0	1	0	$1+t_{rsum}(n-1)$
4	0	1	1	0	0
	Total number of steps			2	$2+t_{rsum}(n-1)$

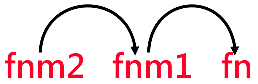
Tabular Method for Matrix Addition

```
line void add (matrix a, matrix b, matrix c, int m, int n)
 1 {
 2 for (int i = 0; i < m; i++)
 3     for (int j = 0 ; j < n ; j++)
 4         c [i ][j ] = a [i ][j ] + b [i ][j ];
 5 }
```

line	s/e	frequency	total steps
1	0	1	0
2	1	$m+1$	$m+1$
3	1	$m(n+1)$	$mn+m$
4	1	mn	mn
5	0	1	0
	Total number of steps		$2mn+2m+1$

Step Counting of Fibonacci Numbers

```
void fibonacci (int n)
// compute the Fibonacci number Fn
{
    if (n <= 1)  cout << n << endl; //F0 = 0 and F1 = 1
    else { // compute Fn
        int fn; int fnm2 = 0; int  fnm1 = 1;
        for (int i = 2; i <= n; i++)
        {
            fn = fnm1 + fnm2 ;
            fnm2 = fnm1 ;
            fnm1 = fn ;
        } // end of for
        cout << fn << endl;
    } // end of else
} // end of fibonacci
```



$F_0=0$ and $F_1=1$

$F_n = F_{n-1} + F_{n-2}$ for $n \geqq 2$

0 1 1 2 3 5 8 13 21 34 55 ...

Program of Fibonacci Sequence Generator

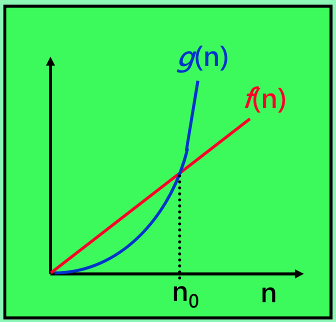
Summary of CPU Time Estimation

- **CPU Time**
  - **Is a function of "instance characteristics"**
  - **Varies as the magnitudes of the inputs increase**
- **In BinarySearch**
  - **The step count is dependent on the array and '*x*' to be searched**
  - **Best case, average case, and the worst case are different.**

所以我們應該要瞭解的是不同條件下的趨勢，而不只是一個值

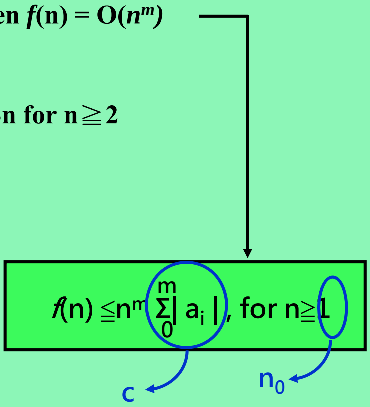
Asymptotic & Big-O Notation

- **Asymptotic Complexity**
  - Concerns about how space or time complexities grow as the size of the problem's inputs grows
- **Big-O Definition**
  - $f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \leqq cg(n)$ for all n, $n \geqq n_0$
  - That is, $g(n)$ is an upper bound of $f(n)$
- **Examples**
  - $O(1)$: constant time computing
  - $O(n)$: linear
  - $O(n^2)$: quadratic
  - $O(n^3)$: cubic
  - $O(2^n)$: exponential
  - $O(\log n)$: logarithmic, $O(n \log n)$



Complexity of Polynomial

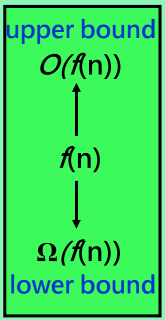
- **Theorem 1.2**
  - If $f(n) = a_m n^m + \ldots + a_1 n + a_0$, then $f(n) = O(n^m)$
- **Examples**
  - $3n+2 = O(n)$ → because $3n+2 \leqq 4n$ for $n \geqq 2$
  - $6 \cdot 2^n + n^2 = O(2^n)$
  - $3n+2 \neq O(1)$
  - $10n^2+4n+2 \neq O(n)$



$f(n) \leq n^m \underbrace{\sum_{0}^{m} |a_i|}_{c}$, for $n \geq \underbrace{1}_{n_0}$

Omega Definition

- **Omega**
  - $f(n) = \Omega(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \geqq cg(n)$ for all n, $n \geqq n_0$
  - That is, $g(n)$ is a lower bound of $f(n)$
  - There could be multiple lower bounds, but it is often that we choose the tight one
- **Theorem 1.3**
  - If $f(n) = a_m n^m + \ldots + a_1 n + a_0$, $a_m > 0$, then $f(n) = \Omega(n^m)$
- **Examples**
  - $3n+2 = \Omega(n)$ → because $3n+2 \geqq 3n$ for $n \geqq 2$
  - $6 \cdot 2^n + n^2 = \Omega(n^2)$
  - $10n^2+4n+2 = \Omega(n)$



Theta Definition

- **Theta**
  - $F(n) = \Theta(g(n))$ iff there exist positive constants c_1 and c_2 and n_0 such that $c_1 g(n) \leqq f(n) \leqq c_2 g(n)$ for all n, $n \geqq n_0$
  - That is, $g(n)$ is both a lower bound and upper bound of $f(n)$
- **Theorem 1.4**
  - If $f(n) = a_m n^m + \ldots + a_1 n + a_0$, $a_m > 0$, then $f(n) = \Theta(n^m)$
- **Examples**
  - $3n+2 = \Theta(n)$
  - $6 \cdot 2^n + n^2 = \Theta(2^n)$
  - $3n+2 \neq \Theta(1)$
  - $10n^2+4n+2 \neq \Theta(n)$

Common Recurrence Relation (I)

Reducing problem size by 1 after a constant time

$T(n) = T(n-1) + 1$

$= T(n-2) + 1 + 1$

$= \ldots$

$= T(1) + (n-1)$

$= O(n)$

E.g., iterative summation

Reducing problem size by 1 after a linear time

$T(n) = T(n-1) + n$

$= T(n-2) + n + (n-1)$

$= \ldots$

$= T(1) + O(n^2)$

$= O(n^2)$

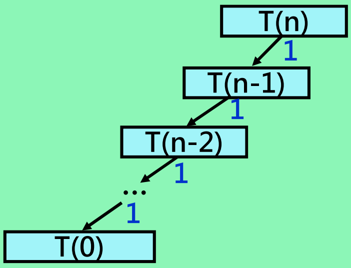
E.g., selection sort algorithm

Common Recurrence Relation (I)

Reducing problem size by 1 after a constant time

$T(n) = T(n-1) + 1$

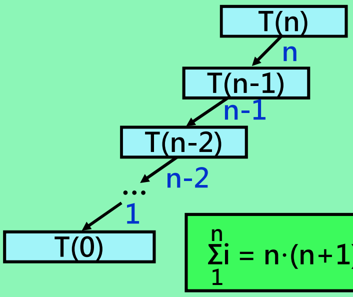
$= O(n)$



Reducing problem size by 1 after a linear time

$T(n) = T(n-1) + n$

$= O(n^2)$



$\sum_{1}^{n} i = n\cdot(n+1)/2 = O(n^2)$

Common Recurrence Relation (II)

Reducing problem size by half after a constant time

$T(n) = T(n/2) + 1$

$= T(n/4) + 1 + 1$

$= \ldots$

$= T(1) + k$

$= O(\log n)$

Assume $n = 2^k$

E.g., Binary search

Reducing problem size by half after a linear time

$T(n) = T(n/2) + n$

$= T(n/4) + n + (n/2)$

$= \ldots$

$= T(1) + (2^k+2^{k-1}+2)$

$= O(n)$

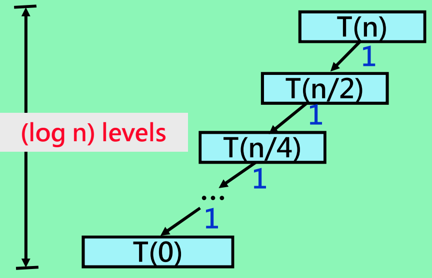
$\sum_{1}^{k} 2^i = (2^{k+1}-1)$

Common Recurrence Relation (II)

Reducing problem size by half after a constant time

$T(n) = T(n/2) + 1$

$= O(\log n)$

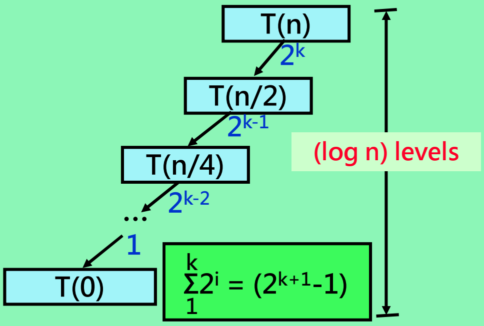


Reducing problem size by half after a linear time

$T(n) = T(n/2) + n$

$= O(n)$

Let $n = 2^k$



$\sum_{1}^{k} 2^i = (2^{k+1}-1)$

Common Recurrence Relation (III)

Split into two equal sub-problems after a constant time

$$
\begin{aligned}
T(n) &= 2T(n/2) + 1 \\
&= 4T(n/4) + (1 + 2) \\
&= \ldots \\
&= nT(1) + (2+2^{k-1}+2^{k}) \\
&= O(n)
\end{aligned}
$$

Split into two equal sub-problems after a linear time

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 4T(n/4) + n + 2(n/2) \\
&= \ldots \\
&= nT(1) + \overbrace{(n+n+\ldots+n)}^{\boxed{\text{k terms}}} \\
&= O(n \cdot \log n)
\end{aligned}
$$

Assume n = 2^k

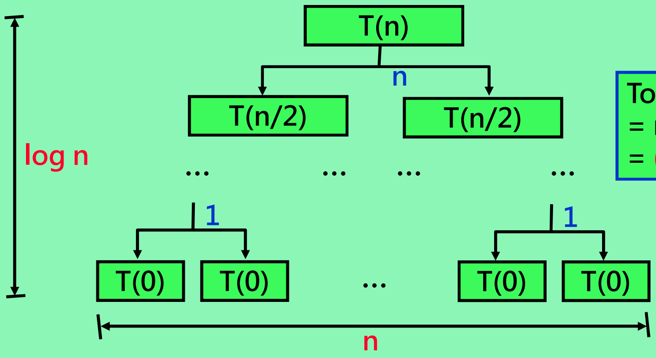
Common Recurrence Relation (III)

Split into two equal sub-problem after a constant time

$$
\begin{aligned}
T(n) &= 2T(n/2) + 1 \\
&= O(n)
\end{aligned}
$$

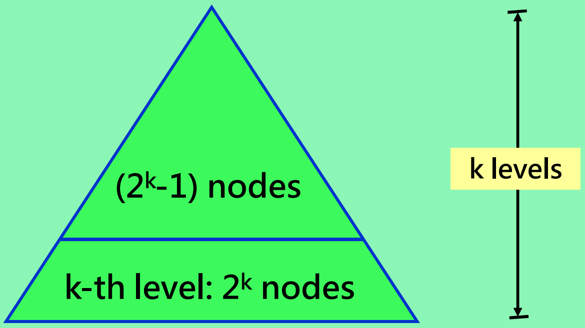
Split into two equal sub-problems after a linear time

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= O(n \cdot \log n)
\end{aligned}
$$



Total number of steps
= n · (#levels)
= (n · log n)

Property of Binary Tree



Total number of nodes in the sub-tree
$= 1 + 2 + 2^2 + \ldots + 2^{k-1}$
$= (2^k-1) / (2-1)$
→ has one node smaller than the last level

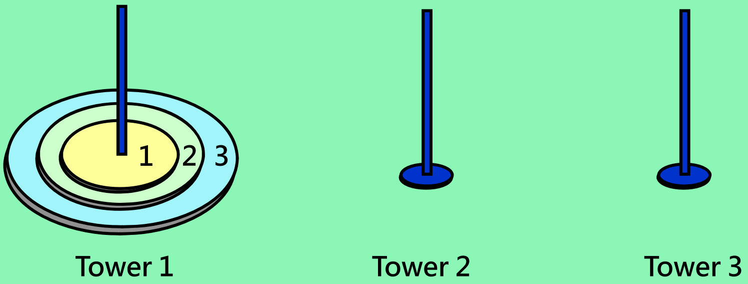
Comparison of Recurrence Relation

- $T(n) = T(n-1) + 1 = O(n)$
- $T(n) = T(n-1) + n = O(n^2)$
- $T(n) = T(n/2) + 1 = O(\log n)$
- $T(n) = T(n/2) + n = O(n)$
- $T(n) = 2T(n/2) + 1 = O(n)$
- $T(n) = 2T(n/2) + n = O(n \cdot \log n)$

這裡每一個公式代表了一種
重要解決問題的思考模式!

Exercise: What is the complexity of a recurrence relation $T(n) = 2T(n-1) + 1$?

Hanoi Towers



Goal: Move the three disks from Tower 1 to Tower 3

Rules:

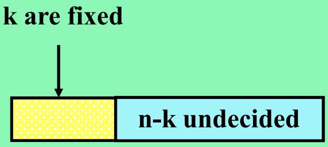
(1) One disk can be moved at a time

(2) No disk can be placed on top of a disk with a smaller diameter

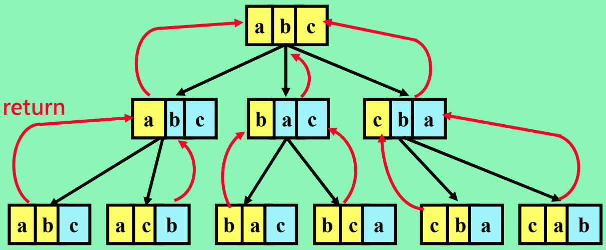
Complexity: $T(n) = 2T(n-1) + 1 \rightarrow T(n) = O(2^n)$

Asymptotic Complexity Of Permutation Generator

$$
\begin{aligned}
T(0, n) &= (n) \cdot T(1,n) \\
&= (n)(n-1) \cdot T(2,n) \\
&= \ldots \\
&= (n)(n-1)2 \cdot T(n-1,n) \\
&= (n)(n-1)2 \cdot O(n) \\
&= O(n! \cdot n)
\end{aligned}
$$

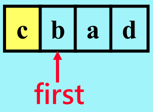


n is the total number elements
k is the number of positions fixed



Recursive Permutation Generator

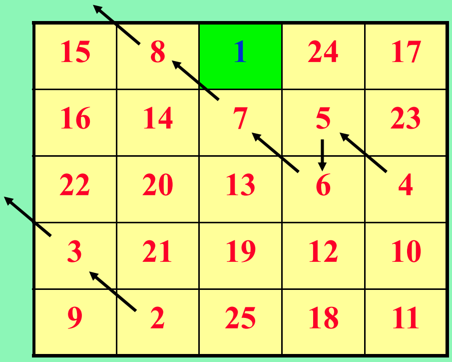
```
void perm (char *a, const int first, const int n)
// generate all permutations of a[first],…,a[n-1]
// first - the first element in the undecided region
{
   if(first==n-1) {  // terminating condition
      for(int i=0; i<n; i++) cout << a[i] << " ";
      cout << endl;
   }
   else {
        for (i=first; i<n; i++) {
             char temp=a[first]; a[first]=a[i]; a[i]=temp;
             perm(a, first+1, n);
             temp=a[first]; a[first]=a[i]; a[i]=temp;
             // return to original configuration
        }
   }
}
```



Program 1.11

Magic Square

- **A magic square**
  - Is an $n \times n$ matrix of the integers 1 to n^2 such that the sum of every row, column, and diagonal is the same



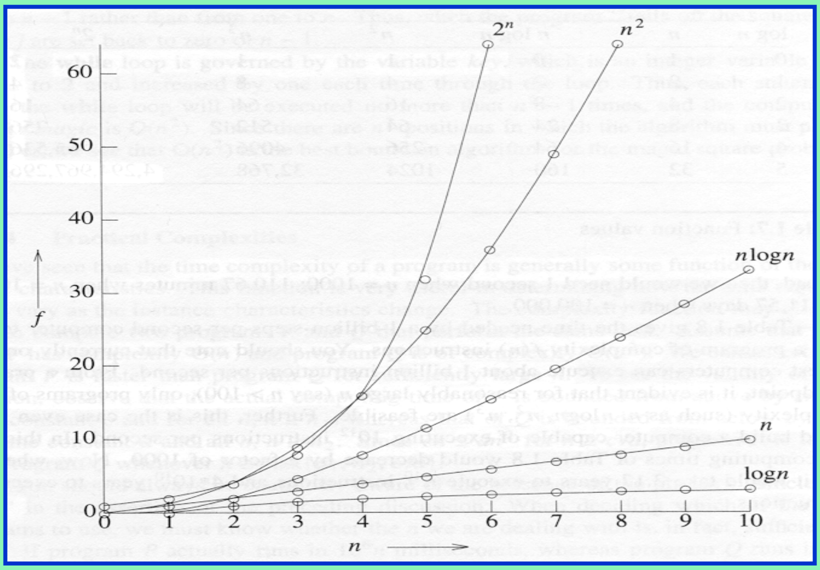
15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Time complexity of magic square = O(n^2)

Practical Complexity

log n	n	n·log n	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Comparison of Different Complexities



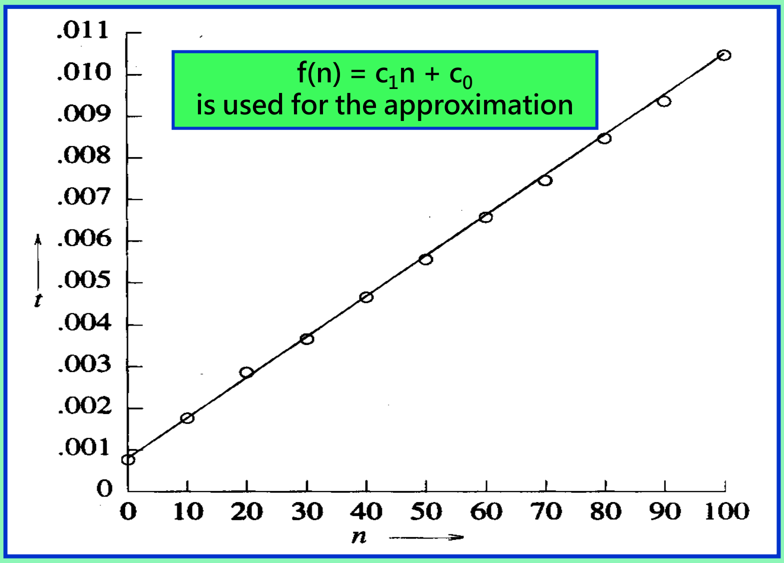
Performance Measurement

- **Performance measurement**
  - **is concerned about the actual time and space requirements of a program**
  - **related to compiler and computer**
- **Asymptotic analysis**
  - **only tells us the behavior for "sufficiently large" values of *n***
- **Actual time**
  - **may not lie exactly on the predicted curve because of the effects of low-order terms that are discarded in the asymptotic analysis**

Precision of Measurement Clock

- **Time() command**
  - **has a clock precision of only 1/100 second**
- **To time a short event**
  - **it is necessary to repeat it several times and divide the total time by the number of repetitions**
- **Random data**
  - **is also commonly used as inputs for average time measurement**
- **Measurement**
  - **could be for (1) comparison, or for (2) prediction**
  - **Least-square approximation could be used if the asymptotic complexity is known, e.g., $(a_0 + a_1 n + a_2 n \, logn)$ is used to approximate a program with O($n \, logn$) complexity**

Approximation



$f(n) = c_1 n + c_0$
is used for the approximation

Ex1: Measure The CPU Time

```
type_define struct _time_buffer {
      long utime;  long stime;  long cutime;  long cstime;
} time_buffer;

main(){
      time_buffer  T;
      float             start, stop, cpu_time;

      /*------ (1) record the start time --------*/
      times( & T );   start = (float) T.utime; /* a tick is 0.01 second */
      /*------ (2) perform operations to be measured ------*/
      target_function();
      /*------ (3) record the stop time -----*/
      times(&T);   stop = (float) T.utime;
      /*------ (4) measure the elapsed time ------*/
      cpu_time = (stop – start)/100.0;
}
```

Ex2: Measure The CPU Time

```
#include <stdlib.h>
#include <iostream.h>
#include <time.h>       //使用clock()函數 (測試程式目前執行時間)
                        //函數原型  clock_t clock(void)
int main(void)
{
     clock_t start,stop;       int n;

     cout<<endl<<"輸入一整數(10-20之間才不用等太久)"<<endl;
     cin>>n;

     start = clock();       //紀錄開始計算時間
     for(int i=0;i<1000000*n*n;i++) { long s;    s=s+i; }
     stop = clock();        //紀錄計算結束時間
     double usetime;
     usetime=((double)stop-(double)start)/CLK_TCK;  //除以CLK_TCK=1000把單位換成秒
     cout << endl << "usetime=“ << usetime << "sec.“ << endl;
     return 0;
}
```

Standard Template Library (STL)

The C++ STL (Standard Template Library) is a generic collection of class templates and algorithms that allow programmers to easily implement standard data structures like queues, lists and stacks.

(Informative web sites about programming in C++ using STL)

C++ reference 網頁: http://www.cppreference.com/wiki/start

STL 網頁: http://www.cppreference.com/wiki/stl/start

The End of
Chapter 1: Basic Concepts !

Next Topic: Arrays

國立清華大學 電機工程學系
EE2410 Data Structure

Chapter 2 Arrays

Outline

- ➡ **Abstract Data Types and Class**
- **The Array as an Abstract Data Type**
- **The Polynomial**
- **Sparse Matrix**
- **The String**

A Class

- **Four components of a class**
  - **A class name**
  - **Data members**
  - **Member functions**
  - **Levels of program access**
    - **Public:** data or functions can be accessed from anywhere
    - **Protected:** accessed from within its class, from its sub-class, or from a friend class
    - **Private:** accessed from within its class or by a friend class

---

Definition of a Class for Rectangle

```
#ifndef RECTANGLE_H
#define RECTANGLE_H
// In the header file Rectangle.h
class Rectangle {
public:       //  the following members are public
   // The next four members are member functions
   Rectangle();      // constructor
   ˜Rectangle();     // destructor
   int GetHeight();  // returns the height of the rectangle
   int GetWidth();   // returns the width of the rectangle
private:          // the following members are private
   // the following members are data members
   int x1, y1, h, w;
   // (x1, y1) are the coordinates of the bottom left corner of the rectangle
   // w is the width of the rectangle; h is the height of the rectangle
};
#endif
```

Special Class Operations

- **Constructor**
  - **Is a member function which initializes data members of an object**
  - **If provided, it is automatically executed when an object of that class is created**
  - **If not provided, data members are not properly initialized**
- **Destructor**
  - **is member function which deletes data members immediately before the object disappears**
  - **Invoked automatically when a class object goes out of scope or explicitly deleted**

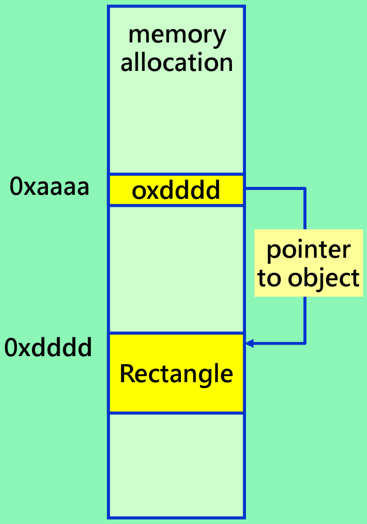
Implementation of Operations on Rectangle

```
// In the source file Rectangle.C
#include “Rectangle.h”

// The prefix “Rectangle::” identifies GetHeight() and GetWidth()
// as member functions belonging to class Rectangle. It is required
// because the member functions are implemented outside their
// class definition

int  Rectangle::GetHeight() { return h; }
int  Rectangle::GetWidth() { return w; }
```

Example: Object Usage

```
// In a source file main.C
#include <iostream.h>
#include "Rectangle.h"

main() {
   Rectangle r, s;      // r and s are objects of class Rectangle
   Rectangle *t = &s;   // t is a pointer to class object s
    .
    .
   // use · to access members of class objects.
   // use → to access members of class objects through pointers.
   if (r.GetHeight () * r.GetWidth () > t→GetHeight () * t→GetWidth ())
      cout << " r ";
   else cout << " s ";
   cout << "has the greater area" << endl;
}
```

Object vs. Pointer



Rectangle r, s;
Rectangle *t = &s;

symbol table

name	type	address
s	Rectangle	0xdddd
t	pointer	0xaaaa

Example: Constructor

```
Rectangle::Rectangle(int x, int y, int height, int width)
{
  x1 = x;   y1 = y;
  h = height;   w = width;
}
→ a constructor must be public, and has no return type
```

Example: initializes Rectangle objects:

Rectangle r(1, 3, 6, 6);
Rectangle *s = new Rectangle(0, 0, 3, 4)

Example: Illegal declaration

Once a constructor is defined, proper input arguments for initialization must be provided
→ otherwise, it is a compile-time error

Efficient Yet Sophisticated Constructor

```
Rectangle::Rectangle(int x=0, int y=0, int height=0, int
width=0): x1(x), y1 (y), h (height), w (width)
{}
```

The data members are initialized by using a member initialization list
(i.e., colon followed by a list of data members and the arguments to
which they are to be initialized in parentheses)
→ Directly initializes the data members in a single step

Operator Overloading

- **Overload operators for user-defined data types**
  - **Is allowed in C++**
  - **Takes the form of a class member function or an ordinary function**

```
int Rectangle::operator==(const Rectangle& s)
{
    if(this == &s) return(1);  // check if two objects are the same
    if( (x1 == s.x1) && (y1 == s.y1) && (h == s.h) (w == s.w) {
                return(1);
    }
            else  return(0);
}
```

“*this*” is the pointer to the data object upon which the operator is performed

Example: Overload Operator==

```
#include <iostream.h>
class complex{
    public:
        complex(int re, int im){ real = re; imaginary = im; }
        int         get_real(){             return(real); }
        int         get_imaginary(){   return(imaginary); }
       int         operator == (complex x){
                        if(real == x.get_real() && imaginary == x.get_imaginary())
                              return(1);
                        else  return(0);
                   }
    private:
        int   real;     int  imaginary;
};

main(){
        int         i;
        complex  a(1, 2), b(3, 4);
       cout << (a == b);
}
```

Example: Overload Operator<<

```
ostream& operator<<(ostream& os, Rectangle& r)
{
    os << “Position is: “ << r.x1 << “ “;
    os << r.y1 << endl;
    os << “Height is: “ << r.h << endl;
    os << “Width is: “ << r.w << endl;
    return os;
}
```

Operator<< accesses private data members of class Rectangle
→ therefore, it must be made a friend of Rectangle.

Note that: friend is an exception of data encapsulation, should be avoided in most cases. But sometimes it is necessary as in this case.

Example: cout << r;
→ Position is: 1 3
→ Height is: 6
→ Width is: 6

Example: Overload Operator<<

```
#include "iostream.h"
class complex{
public:
      complex(int re, int im){ real = re; imaginary = im; }
      int         get_real(){           return(real); }
      int         get_imaginary(){  return(imaginary); }
      friend     ostream& operator<<(ostream&, complex);
private:
      int   real;     int  imaginary;
};
ostream& operator<<(ostream& os, complex x){
      os << x.get_real() << endl;
      os << x.get_imaginary() <<endl ;
     return(os);
}
main(){
      int          i;
      complex  a(1, 2), b(3, 4);
     cout << a << b;   // will be illegal if the return type of << is not ostream&
}
```

declare operator<< as a friend operator

Miscellaneous Topics

- Struct
  - Is identical to a class, except that the default level of access is public.
  - In a class, the default is private
- Union
  - A struct that reserves storage for the largest of its data members
  - Useful for applications where only one of many possible data items need to be stored at any time
- Static class data member
  - May be thought of as a global variable for its class
  - A definition of the data member outside the class definition is required

---

Example of Union

```
struct pair {
     int   n1; // 32 bits
     float  n2; // 32 bits
}
```

A *pair* requires 2x32 bits = 64 bits

```
struct exclusive_pair {
     union {
          int   n1;  // 32 bits
          float  n2; // 32 bits
     };
}
```

An *exclusive_pair* contains only
an integer or a floating point number
→ requires only 32 bits

Example of Using Union

```
#include <iostream.h>
typedef  struct  _exclusive_pair {
     union {
          int     n1;
          int     n2;
     }
} ex_pair;
```

```
main()
{
    ex_pair p;
    p.n1 = 1;   p.n2 = 10;
    cout << p.n1 << “  “ << p.n2 ;
}
```

→(Result): 10 10

C++ ADT for Natural Numbers

```
class NaturalNumber {
// An ordered subrange of the integers starting at zero and ending at
// the maximum integer (MAXINT) on the computer
public:
    NaturalNumber Zero( );
    // returns 0

    Boolean IsZero( );
    // if *this is 0, return TRUE; otherwise, return FALSE

    NaturalNumber Add(NaturalNumber y);
    // return the smaller of *this + y and MAXINT;

    Boolean Equal(NaturalNumber y);
    // return TRUE if *this == y; otherwise return FALSE

    NaturalNumber Successor( );
    // if *this is MAXINT return MAXINT; otherwise return *this + 1

    NaturalNumber Subtract(NaturalNumber y);
    // if *this < y, return 0; otherwise return *this – y
};
```

Outline

- Abstract Data Types and Class
- The Array as an Abstract Data Type
- The Polynomial
- Sparse Matrices
- The String

Traditional Array

- Array
  - Is often viewed as a consecutive set of memory locations
  - Is a set of ordered pair, i.e., <index, value>
  - Provides two standard operations
    - Store a value to a given index
    - Retrieve a value corresponding to a given index
  - Store and Retrieve are performed in constant time
- A more robust array is needed
  - To avoid out-of-bound access

ADT GeneralArray

```
class GeneralArray {
// objects: A set of pairs <index, value> where for each value of index in IndexSet there
// is a value of type float . IndexSet is a finite ordered set of one or more
// dimensions, for example, {0, ··· , n – 1} for one dimension,
// {(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)} for two dimensions, etc.
public:
  GeneralArray(int j, RangeList list,float initValue = defaultValue);
  // The constructor GeneralArray creates a j dimensional array of floats; the range
  // of the kth dimension is given by the kth element of list. For each index i in the
  // index set, insert <i, initValue> into the array.

  float Retrieve(index i);
  // if (i is in the index set of the array) return the float associated with i
  // in the array; else signal an error.

  void Store(index i, float x);
  // if (i is in the index set of the array) delete any pair of the form <i, y> present
  // in the array and insert the new pair <i, x>; else signal an error.
}; // end of GeneralArray
```

Multi-dimensional Array

- **An n-dimensional array**
  - **Is usually implemented as a one-dimensional array**
  - **A mapping mechanism is needed**
  - **Assumption of $A[u_1][u_2]\ldots[u_n]$**
    - **First index range: $[p_1 .. q_1]$**
    - **Second index range: $[p_2 .. q_2]$**
    - **…**
    - **n-th index range: $[p_n .. q_n]$**
  - **The total number of elements in this n-dimensional array is**

$$\boxed{\prod_{i=1}^{n}(q_i - p_i + 1)}$$

Row Major

- **Row Major Order**
  - **Is also called lexicographic order**
- **Example**
  - **A[4..5] [2..4] [1..2] [3..4]**
  - **Total number of elements:2*3*2*2 = 24**
  - **Element order**

    A[4][2][1][3], A[4][2][1][4], A[4][2][2][3], A[4][2][2][4],

    A[4][3][1][3], A[4][3][1][4], A[4][3][2][3], A[4][3][2][4],

    A[4][4][1][3], A[4][4][1][4], A[4][4][2][3], A[4][4][2][4],

    …

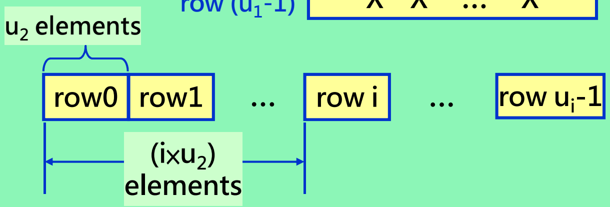
Demonstrations

Sequential representation of $A[u_1]$

Array element:	A[0]	A[1]	A[2]	…	A[i]	…	$A[u_1-1]$
Address:	α	α +1	α+2		α+i-1		α+i

Sequential representation of $A[u_1]\,[u_2]$

$$\begin{array}{r|cccc|} & \text{col0} & \text{col1} & & \text{col }(u_2-1) \\ \hline \text{row 0} & X & X & \dots & X \\ \text{row 1} & X & X & \dots & X \\ \text{row 2} & X & X & \dots & X \\ & & & \cdot\;\cdot\;\cdot & \\ \text{row }(u_1-1) & X & X & \dots & X \\ \hline \end{array}$$



Address Mapping Formula

- Assume α is the address of A[0][0][0]
- Address for $A[i][0][0] = \alpha + (i \times u_2\, u_3)$
- Address for $A[i][j][0] = \alpha + (i \times u_2\, u_3) + (j \times u_3)$
- Address for $A[i][j][k] = \alpha + (i \times u_2\, u_3) + (j \times u_3) + k$

$A[i_1]\,[i_2], \ldots, [i_n]$

(Row major order)

Mapped to

$$\alpha + \sum_{j=1}^{n} i_j\, a_j$$

$$\text{where } a_j = \prod_{k=j+1}^{n} u_k \quad 1 \leqq j \leqq n$$

$$a_n = 1$$

Outline

- Abstract Data Types and Class
- The Array as an Abstract Data Type
- The Polynomial
- Sparse Matrices
- The String

ADT of Polynomial

class *Polynomial* **{**
//**objects**: $p(x) = a_0 x^{e_0} + \cdots + a_n x^{e_n}$; a set of ordered pairs of $<e_i, a_i>$,
// where $a_i \in$ *Coefficient* and $e_i \in$ *Exponent*
// We assume that *Exponent* consists of integers ≥ 0
public:

Polynomial()**;**
//return the polynomial $p(x) = 0$

int operator!()**;**
//**if *this** is the zero polynomial, return 1; **else** return 0;

Coefficient Coef(*Exponent e*)**;**
//return the coefficient of *e* in ***this**

Exponent LeadExp()**;**
//return the largest exponent in ***this**

Polynomial Add(*Polynomial poly*)**;**
// return the sum of the polynomials ***this** and *poly*

Polynomial Mult(*Polynomial poly*)**;**
// return the product of the polynomials ***this** and *poly*

float *Eval*(**float** *f*) **;**
// Evaluate the polynomial ***this** at *f* and return the result.
}; // end of *Polynomial*

Polynomial Representation (1)

- **Fixed-size array representation**

```
private:
  int  degree; // degree ≦MaxDegree
  float coef[MaxDegree+1];
```

- **Example**
  - Assume that *a* is a polynomial class object and *n* ≦ MaxDegree
  - E.g., $a_n x^n + \ldots + a_1 x^1 + a_0$
  - Then, $a.degree = n$ and $a.coef[i] = a_{n-i}$, $0 \leqq i \leqq n$
- **Disadvantage of using static array**
  - Wasteful in its use of computer memory

Polynomial Representation (2)

- **Array with dynamic size**

```
private:
  int  degree;
  float *coef;
```

- **Constructor**

```
Polynomial::Polynomial(int d)
{
    degree = d;
    coef = new float [degree + 1]
}
```

- **Advantage**
  - **The size of array can be dynamically decided, leading to a more efficient memory usage**

Polynomial Representation (3)

- **For sparse polynomial**
  - **E.g., $x^{100} + 1$ → as <coef, exp> list → {<1, 100>, <1, 0>}**
- **Shared single array is used**
  - **All polynomials, (if there are many in the program), will be put together as a shared single array**

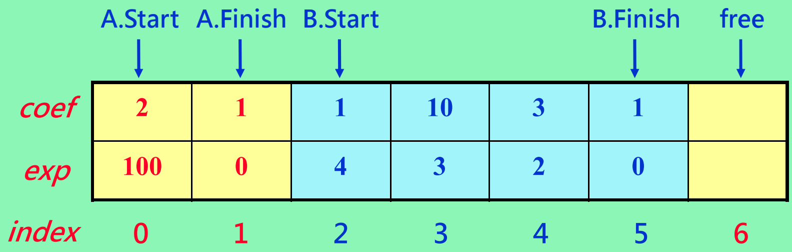
```
class Polynomial; // forward declaration

class term {
   friend Polynomial;
   private:
       float  coef; // coefficient
       int exp; // exponent
};
```

```
// private data members of Polynomial

private:
   static term termArray[MaxTerms];
   static int free;
   int  Start, Finish;
```

```
// static data outside class declaration
term  Polynomial::termArray[MaxTerms];
int Polynomial::free = 0;
           // next free location in termArray
```

Example: Array for Two Polynomials

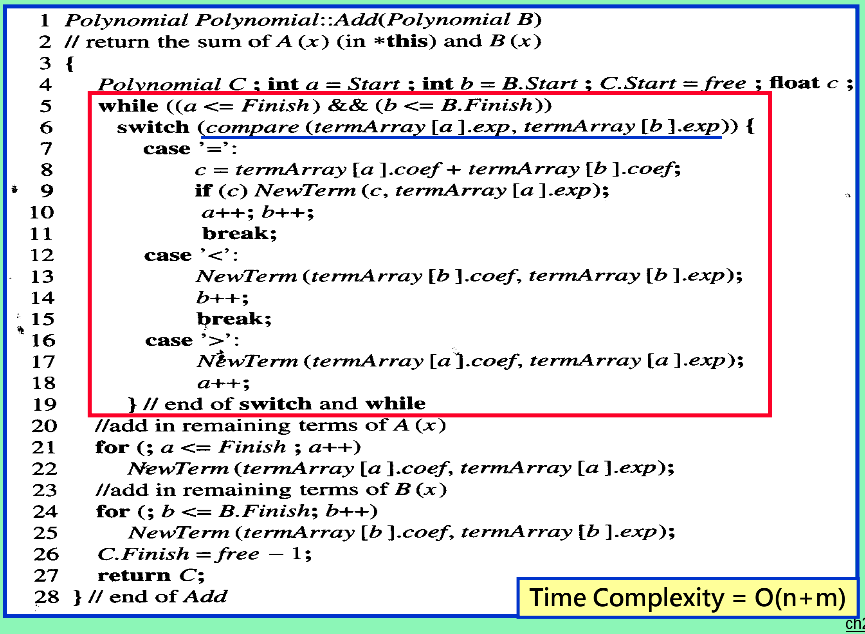
- $A(x) = 2x^{100} + 1$
  - A.Start = 0, A.Finish = 1
- $B(x) = x^4 + 10x^3 + 3x^2 + 1$
  - B.Start = 2, B.Finish = 5



	A.Start	A.Finish	B.Start			B.Finish	free
coef	2	1	1	10	3	1	
exp	100	0	4	3	2	0	
index	0	1	2	3	4	5	6

A zero polynomial *Z*(*x*)=0 → *Z.Finish* = *Z.Start*-1

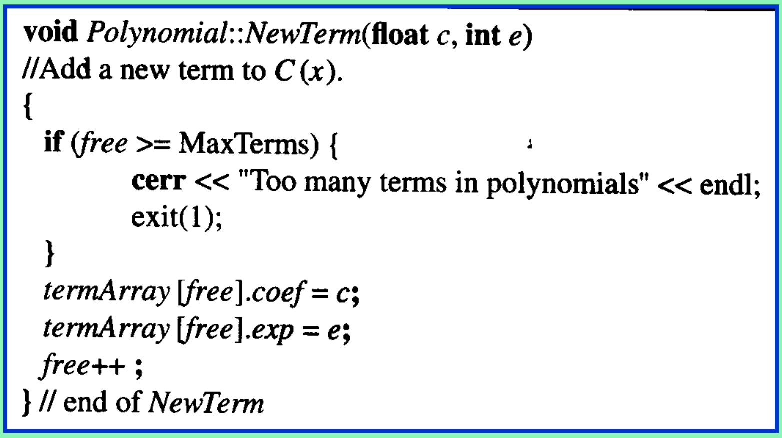
Polynomial Addition



```
Polynomial Polynomial::Add(Polynomial B)
// return the sum of A (x) (in *this) and B (x)
{
   Polynomial C ; int a = Start ; int b = B.Start ; C.Start = free ; float c ;
   while ((a <= Finish) && (b <= B.Finish))
     switch (compare (termArray [a ].exp, termArray [b ].exp)) {
        case '=':
             c = termArray [a ].coef + termArray [b ].coef;
             if (c) NewTerm (c, termArray [a ].exp);
              a++; b++;
              break;
        case '<':
             NewTerm (termArray [b ].coef, termArray [b ].exp);
             b++;
             break;
        case '>':
             NewTerm (termArray [a ].coef, termArray [a ].exp);
             a++;
      } // end of switch and while
   //add in remaining terms of A (x)
   for (; a <= Finish ; a++)
      NewTerm (termArray [a ].coef, termArray [a ].exp);
   //add in remaining terms of B (x)
   for (; b <= B.Finish; b++)
      NewTerm (termArray [b ].coef, termArray [b ].exp);
   C.Finish = free − 1;
   return C;
} // end of Add
```

Time Complexity = O(n+m)

Adding A New Polynomial Term



```
void Polynomial::NewTerm(float c, int e)
//Add a new term to C(x).
{
  if (free >= MaxTerms) {
          cerr << "Too many terms in polynomials" << endl;
          exit(1);
  }
  termArray [free].coef = c;
  termArray [free].exp = e;
  free++ ;
} // end of NewTerm
```

Disadvantages of Representing Polynomials by Arrays

- **What could happen when the array is used up?**
  - **Recycle certain polynomials no longer needed**
  - **A sophisticated compaction routine is required, involving a lot of data movement**
- **Another Choice**
  - **Use a single array of terms for each polynomial**
  - **Each array is created by using "new"**
  - **This will requires us to know the size of the polynomial prior to its creation**
    - **A potential trouble to run the addition algorithm twice**

Outline

- Abstract Data Types and Class
- The Array as an Abstract Data Type
- The Polynomial
- Sparse Matrices
- The String

How To Store Matrix ?

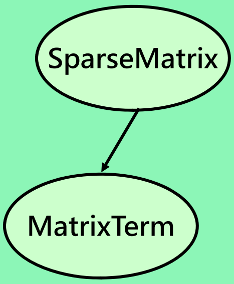
- Two dimensional array
  - int A[m][n]
  - may require huge memory space, e.g., A[5000][5000]
- Sparse matrix
  - Is a matrix in which most elements are zero
- Use two-dimensional array for sparse matrix is a big waste of memory space

Example of Sparse Matrix

SparseMatrix	Col 0	Col 1	Col 2	Col 3	Col 4	Col 5
Row 0	15	0	0	22	0	-15
Row 1	0	11	3	0	0	0
Row 2	0	0	0	-6	0	0
Row 3	0	0	0	0	0	0
Row 4	91	0	0	0	0	0
Row 5	0	0	28	0	0	0

Sparse Matrix Representation

```
class SparseMatrix; // forward declaration

class MatrixTerm {
friend class SparseMatrix;
private:
   int  row, col, value;
};

class SparseMatrix {
public: // member functions …
private:
   int  Rows, Cols, Terms;
   MatrixTerm smArray[MaxTerms];
}
```



sparseMatrix is modeled as a linear array

Example of Sparse Matrix and Its Transpose

The sparse matrix is ordered by rows, and within rows by columns

Original	Row	Col	value
smArray[0]	0	0	15
[1]	0	3	22
[2]	0	5	15
[3]	1	1	11
[4]	1	2	3
[5]	2	3	-6
[6]	4	0	91
[7]	5	2	28

Transposed	Row	Col	value
smArray[0]	0	0	15
[1]	0	4	91
[2]	1	1	11
[3]	2	1	3
[4]	2	5	28
[5]	3	0	22
[6]	3	2	-6
[7]	5	0	15

(0, 3, 22) → (3, 0, 22) after being transposed

Transposing A Matrix

```
SparseMatrix SparseMatrix::Transpose()
// return the transpose of a (*this)
{
  SparseMatrix b;
  b.Rows = Cols;    b.Cols = Rows;   b.Terms = Terms;
  if(Terms > 0) {  // nonzero matrix
    int CurrentB = 0;
    for(int c=0; c < Cols; c++){
        for(int i=0; i<Terms; i++){ // find elements in column c
            if(smArray[i].col == c) {
               b.smArray[CurrentB].row = c;
               b.smArray[CurrentB].col = smArray[i].row;
               b.smArray[CurrentB].value = smArray[i].value;
              CurrentB++; }
        }
     }
  }
  return(b);
}
```

Time Complexity = O(columns x Terms)

Demo of Transposing a Matrix (1)

Step 1: Scan Matrix A to construct the RowSize array of A^T

(記錄 A^T 每一列有多少 “非零元素”)

Matrix A (row,col,value) → RowSize array of A^T

	Matrix A	0	1	2	3
Row 0	0 0 1	1	0	0	0
	0 2 2	1	0	1	0
	0 3 3	1	0	1	1
Row 1	1 2 1	1	0	2	1
	1 3 2	1	0	2	2
Row 2	2 2 1	1	0	3	2
Row 3	4 1 4	1	1	3	2

Final result

Demo of Transposing a Matrix (2)

Step 2: Scan RowSize array to construct the RowStart array of A^T

RowSize of A^T

0	1	2	3
1	1	3	2

RowStart[0] = 0;
RowStart[i] = RowStart[i-1] + RowSize[i-1]

RowStart of A^T

0	1	2	3
0	1	2	5

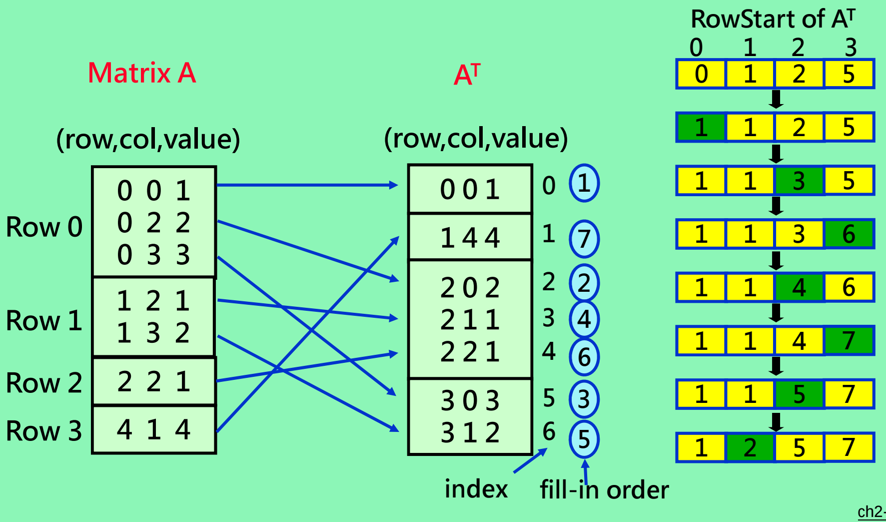
(A^T 每一列的第一個 “非零元素” 之位址)

A^T

0	row 0
1	row 1
2 3 4	row 2
5 6	row 3

Demo of Transposing a Matrix (3)

Step 3: Move each element of Matrix A to Matrix A^T



Transposing A Matrix Faster

```
SparseMatrix SparseMatrix::FastTranspose()
{
  int *RowSize = new int[Cols];
  int *RowStart = new int[Cols];
  SparseMatrix b;
  b.Rows = Cols;   b.Cols = Rows;   b.Terms = Terms;
  if(Terms > 0) {  // nonzero matrix

    // compute no. of elements in each row of B
    for(int i=0; i<Cols; i++){ RowSize[i]=0; }
    for(i=0; i<Terms; i++){ RowSize[smArray[i].col]++; }

    // compute the starting position of each row of B
    RowStart[0]=0;
    for(i=1; i<Cols; i++){
          RowStart[i] = RowStart[i-1]+RowSize[i-1];
    }
    TO BE CONTINUED …
```

Transposing A Matrix Faster (con’t)

```
SparseMatrix SparseMatrix::FastTranspose()
{
  if (Terms > 0) {
    … (in previous page)
    for(i=0; i<Terms; i++){ // move from a to b
        int j = RowStart[smArray[i].col];
        b.smArray[j].row = smArray[i].col;
        b.smArray[j].col = smArray[i].row;
        b.smArray[j].value = smArray[i].value;
        RowStart[smArray[i].col]++;
     }
  }
  delete [ ] RowSize;    delete [ ] RowStart;    return(b);
}
```

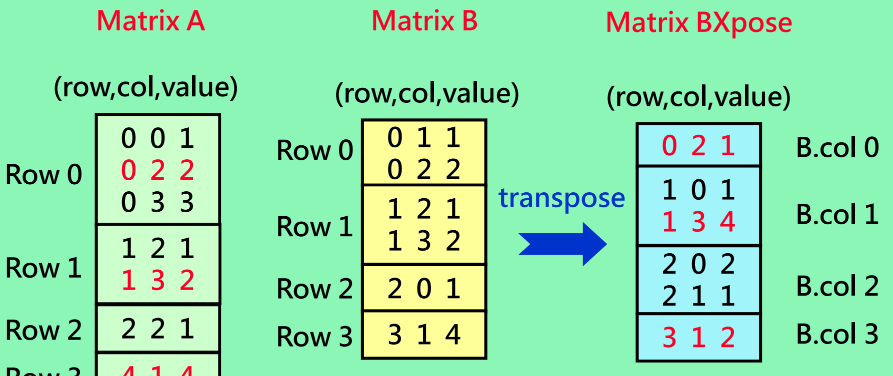
Time Complexity = O(Terms + Columns) ~ O(Terms)

Principles In Fast Transpose

- **Look Before You Leap**
- **A Stitch Beforehand Saves Nine**
- **Quick pre-computation of certain information pays back often**

Matrix Multiplication

$$C_{ij} = (A \times B)_{ij} = \Sigma_k A_{ik} \times B_{kj}$$



Matrix A

(row,col,value)

Row 0	0 0 1 0 2 2 0 3 3
Row 1	1 2 1 1 3 2
Row 2	2 2 1
Row 3	4 1 4

Matrix B

(row,col,value)

Row 0	0 1 1 0 2 2
Row 1	1 2 1 1 3 2
Row 2	2 0 1
Row 3	3 1 4

transpose

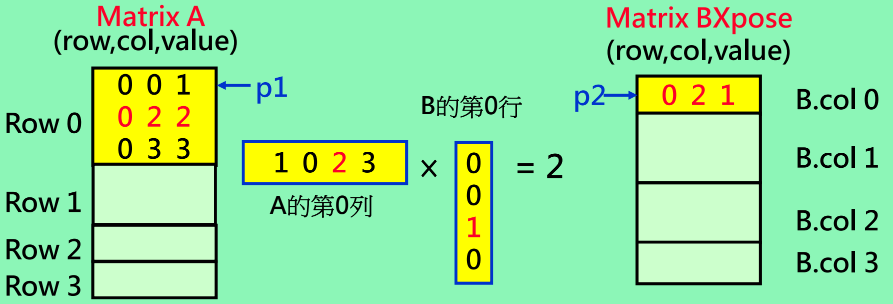
Matrix BXpose

(row,col,value)

0 2 1	B.col 0
1 0 1 1 3 4	B.col 1
2 0 2 2 1 1	B.col 2
3 1 2	B.col 3

Trick: Take the transpose of B before multiplication

Computing C_{00}



Matrix A

(row,col,value)

Row 0	0 0 1 0 2 2 0 3 3
Row 1	
Row 2	
Row 3	

p1

1 0 2 3

A的第0列

× B的第0行

0
0
1
0

= 2

Matrix BXpose

(row,col,value)

p2

0 2 1	B.col 0
	B.col 1
	B.col 2
	B.col 3

```
/*--------- for computing one c_ij ---------*/
   result = 0
   switch( compare(A[p1].col, Bxpose[p2].col) ){
      case ‘<‘: p1++; break;
      case ‘=‘: result += A[p1].value * Bxpose[p2].value;
                p1++; p2++; break;
      case ‘>’: p2++; break;
   }
```

Big-O of Matrix Multiplication

- **Notation**
  - $A_{n \times n} \times B_{n \times n}$
  - A_i: the no. of non-zero elements in i-th row of A
  - B_j: the no. of non-zero elements in j-th column of B
- **Upper Bound Complexity**

$$= O\left(\sum_{i=0}^{n} \sum_{j=0}^{n} (A_i + B_j) \right) = O\left(\sum_{i=0}^{n} (n \cdot A_i + \text{Terms\_of\_B}) \right)$$

$$= O(n \cdot \text{Terms\_of\_A} + n \cdot \text{Terms\_of\_B})$$

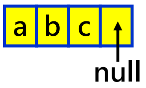
$$= O(n \cdot \max(\text{Terms\_of\_A}, \text{Terms\_of\_B}))$$

Outline

- **Abstract Data Types and Class**
- **The Array as an Abstract Data Type**
- **The Polynomial**
- **Sparse Matrices**
- **The String**
  - **String Pattern Matching**

Abstract Data Type *String*

string “abc” as an array



```
class String
{
// objects: A finite ordered set of zero or more characters.
public:
   String(char *init, int m);
   //Constructor that initializes *this to string init of length m
   int operator==(String t);
   // if (the string  represented by *this equals t) return 1 (TRUE)
   // else return 0 (FALSE);
   int operator!();
   // if *this is empty then return 1 (TRUE); else return 0 (FALSE);
   int Length();
   // return the number of characters in *this
   String Concat(String t);
   // return a string whose elements are those of *this followed by those of t.
   String Substr(int i, int j);
   // return a string containing j characters of *this at positions i, i+1, ..., i+j−1
   // if these are valid positions of *this; otherwise, return the empty string.
   int Find(String pat);
   // return an index i such that pat matches the substring of *this that begins at position i.
   // Return −1 if pat is either empty or not a substring of *this
};
```

→concatenation

String Pattern Matching

- **Problem**
  - **Finding if a pattern is contained in another string**
- **Example**
  - **Pattern: “data structure”**
  - **String: “The course identifier of data structure is EE2410”**

    ↑ 26
  - **Result = 26**
  - **If Pattern does not exist in the string, return –1**

Simple String Pattern Matching

Pattern: data structure (14 characters)
String: data encapsulation is an important concept in data structure

A sliding widow

- Comparing the sub-string within the window with the pattern
  → A mismatch at the sixth character

- Slide the window forward by one step

Pattern: data structure (14 characters)
String: data encapsulation is an important concept in data structure

- A mismatch again !
- Continue to slide the window forward …

Simple String Matching – con't

Pattern: data structure (14 characters)
47
String: data encapsulation is an important concept in data structure

A sliding widow

(1) A match !
(2) Return the starting index of the sliding window = 47

(Time Complexity)
The worst-case complexity of this algorithm = $O(m \cdot n)$
where m is the length of the pattern
and n is the length of the string
(The reason)
Window matching is performed for n times and each of them may take m steps to complete in the worst case

Algorithm – Simple Matching

```
int String::Find(String pat)
// i is set to –1 if pat does not occur in s (*this);
// otherwise i is set to point to the first position in *this, where pat begins.
{
   char *p = pat.str, *s = str ;
   int i = 0 ;  // i is starting point
   if (*p && *s)
     while (i <= Length() – pat.Length())
       if (*p++ == *s++) { //characters match, get next char in pat and s
         if (!*p) return i ; // match found
       }
       else { // no match
         i++ ; s = str + i ; p = pat.str ;
       }
   return –1 ; //pat is empty or does not occur in s
} // end of Find
```

i is the starting index of the window being compared

end of "pattern" reached

move window forward by 1

Optimal Pattern Matching
Knuth-Morris-Pratt Algorithm

- **The problem of the simple algorithm**
  - **The sliding window moves forward one step at a time**
- **Idea**
  - **Can we move the window several steps forward when a partial match occurs in a window matching ?**

Pattern: data structure (14 characters)
String: data encapsulation is an important concept in data structure

Failed at 6th characters, a partial match

Can the window leap forward by 6 ?

Pattern: data structure (14 characters)
String: data encapsulation is an important concept in data structure

What Happens When Window Matching Fails Partially ?

String: data encapsulation is …

Pattern: data structure

Failing point

The following information if pre-computed may enable the leap:

d	a	t	a		Sub-pattern

d | a | t | a | | → mismatch

d a | t | a | | → mismatch

d a t | a | | → mismatch

d a t a | | → mismatch

Failure Function

- **Definition**

f(j) 是 partial match 後 pattern 的新比較點
i.e., string 指標不變 , pattern 移至 f(j) 這個位置

  – If $p=p_0p_1\ldots p_{n-1}$ is a pattern, then its failure function, f, is defined as

  $f(j)$ =

  (1) largest k<j such that $(p_{j-k}p_{j-k+1}\ldots p_{j-1})=(p_0p_1\ldots p_{k-1})$ if such a k≧0 exists

  (2) otherwise $f(j) = 0$;

k 有點 max. self-matching length 的意思

- **Example:**

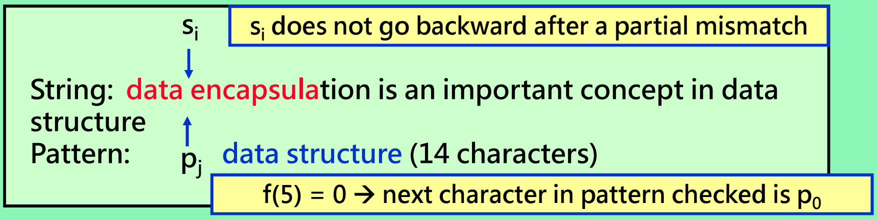
Length-2 prefix matches
length-2 subpattern p[0:1]

Target pattern	a	b	c	a	b	c	a	c	a	b
failing index j	0	1	2	3	4	5	6	7	8	9
f(j)	0	0	0	0	1	2	3	4	0	1

Usage of Failure Function

Failure function decides the leap size after a partial matching

Rule of pattern matching:
If a partial match is found such that
$(s_{i-j} \ldots s_{i-1}) = (p_0 p_1 \ldots p_{j-1})$ and $s_i \neq p_j$
then matching may be resumed by comparing
(1) s_{i+1} and p_0 if $j=0$;
(2) s_i and $p_{f(j)}$ if $j \neq 0$



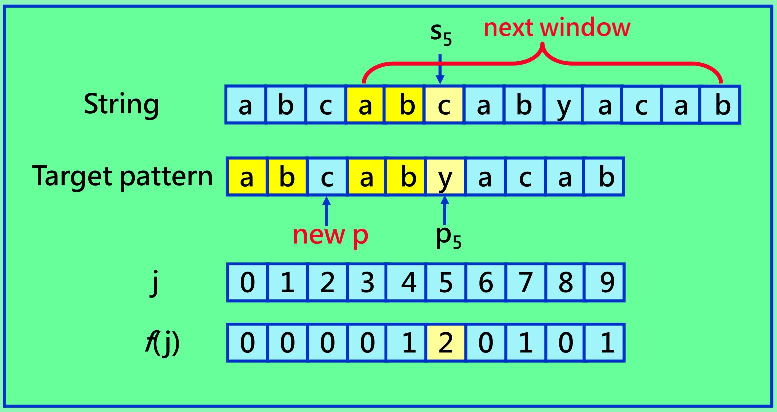
s_i does not go backward after a partial mismatch

String: data encapsulation is an important concept in data structure

Pattern: data structure (14 characters)

f(5) = 0 → next character in pattern checked is p_0

Example of Using Failure Function

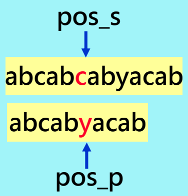


String	a	b	c	a	b	c	a	b	y	a	c	a	b
Target pattern	a	b	c	a	b	y	a	c	a	b			
j	0	1	2	3	4	5	6	7	8	9			
f(j)	0	0	0	0	1	2	0	1	0	1			

Next step: comparing s_5 with p_2
Now, the window of String being compared starts from s_3

FastFind Algorithm

```
int FastFind(String s, String pattern)
{
  // Determine if "pattern" is a substring of s
  int pos_p =0; pos_s=0;
  int length_p = pattern.Length();
  int length_s = s.Length();
  while (pos_p < length_p) && pos_s < length_s) {
     if(pattern.str[pos_p] == s.str[pos_s]) { // matched a character
         pos_p++; pos_s++;
     }
     else { // no match
          if(pos_p == 0) pos_s++;
          else pos_p = pattern.failure[pos_p];
     }
  }
  if(pos_p < length_p) return(-1);
  else return(pos_s – length_p);
}
```



String 是一個 class
(1) str() 取出其字串
(2) Length() 取其長度

Time = O(length_s)

取sliding window 起點當作 matching point

Computing the Failure Function

If the failure function can be computed in O(Length_of_Pattern)
→Then the FastFind has an optimal complexity of
O(Length_of_pattern + Length_of_String)

Target pattern	a	b	c	a	b	a	b	c	a	b	c	c
j	0	1	2	3	4	5	6	7	8	9	10	11
f(j)	0	0	0	0	1	2	1	2	3	4	5	?

Rule:

$f(0) = f(1) = 0$

$f(j) = f^m(j-1)+1$, where m is the least positive integer for which $pattern(f^m(j-1)) = pattern(j-1)$

where $f^m(j) = f(f^{m-1}(j))$

$f(j) = 0$ if no such m exists above

Example: 求 f(11) → f(10)=5 , 但是 pattern(5) ≠pattern(10) 所以失敗
→ 試下一個 self-match length k = $f^2(10)$ = ff(10) = f(5) = 2
→ f(5)=2 and pattern(2)=pattern(10)
Therefore, f(11) = 2 + 1 = 3

Algorithm – Failure Function

Note: 這是一個 Class *String* 的 member function

```
void String::compute_failure_function()
{
  failure[0] = 0;  failure[1] = 0;
  for(int j=1; j<Length(); j++){  // find failure function for each element
      int k = failure[j-1];
      while(1) {
          if (str[k] == str[j-1]) {   // found a match !
                failure[j] = k+1;
                break;
          }
          else if(k != 0) {   // apply the rule recursively here
                 k = failure[k];
          }
          else{ failure[j] = 0;  break; }
      }
  }
}
```

Using Vector in STL

C++ reference 網頁: http://www.cppreference.com/wiki/start

STL 網頁: http://www.cppreference.com/wiki/stl/start

STL Vector 網頁: http://www.cppreference.com/wiki/stl/vector/start

Example: create an array of string

```
vector<string> words;  // words is an array of string
string str; // str is a string
while( cin >> str ) words.push_back(str);
vector<string>::iterator iter;
for( iter = words.begin(); iter != words.end(); iter++ ) {
        cout << *iter << endl;
}
```

The End of
Chapter 2: Arrays !

Next Topic:

Stacks and Queues

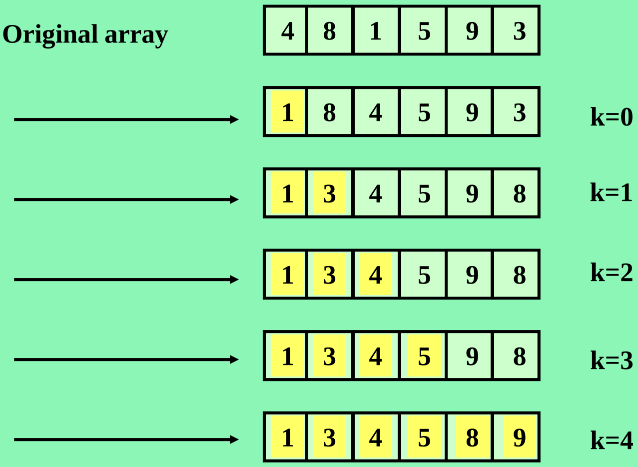
國立清華大學 電機工程學系
EE2410 Data Structure

Chapter 3
Stacks and Queues

Outline

- Templates in C++
- The Stack Abstract Data Type
- The Queue Abstract Data Type
- SubTyping and Inheritance in C++
- A Mazing Problem
- Evaluation of Expressions

Example of Selection Sort



Original array	4	8	1	5	9	3	
→	1	8	4	5	9	3	**k=0**
→	1	3	4	5	9	8	**k=1**
→	1	3	4	5	9	8	**k=2**
→	1	3	4	5	9	8	**k=3**
→	1	3	4	5	8	9	**k=4**

Selection Sort Using Template

```
template <class KeyType>
void  sort (KeyType *a, int n)
// sort the n KeyType a[0] to a[n-1] into nondecreasing order
{
   for(int k=0; k < n; k++){
        int smallest = k;
        // find the smallest KeyType in a[k] to a[n-1]
        for (int j = k+1; j < n; j++){
              if(a[j] < a[smallest]) smallest = j;
        }
        KeyType temp = a[k]; a[k] = a[smallest]; a[smallest] = temp;
   }
}
main(){
  float  real_array[100];
  int     int_array[250];
  … // assume that the arrays have been initialized
  sort(real_array, 100);
  sort(int_array, 250);
  …
}
```

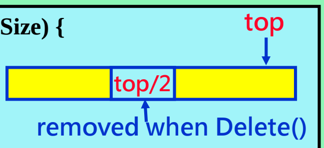
Concept of Template

- **A Template**
  - **also called parameterized types**
  - **It may be viewed as a variable which can be instantiated to any data type when a function or class is used**
- **Copy Constructor**
  - **is a special constructor which is invoked when an object is initialized with another object**
  - **e.g., *KeyType* temp=*a*[k];**
  - **might be overloaded if the parameterized type is not a pre-defined data type in C++**

Container Class

```
class Bag
{
public:
    Bag (int MaxSize = DefaultSize); // constructor
    ~Bag();                    // destructor
    void Add(int);         // insert an integer into Bag
    void Delete(int &); // delete an integer from Bag
    Boolean IsFull();      // return TRUE if the bag is full; FALSE otherwise
    Boolean IsEmpty(); // return TRUE if the Bag is empty; FALSE otherwise
private:
    void Full();      // action when bag is full
    void Empty(); // action when the bag is empty

    int *array;
    int  MaxSize; // size of array
    int  top;         // highest position in array that contains an element
};
```

Implementation of Bag



```
Bag::Bag(int MaxBagSize): MaxSize (MaxBagSize) {
   array = new int[MaxSize]; top = -1;
}
Bag::~Bag() { delete [] array; }
inline Boolean Bag::IsFull(){
       if(top == MaxSize – 1) return TRUE; else return FALSE;
}
inline Boolean Bag::IsEmpty(){
       if(top == -1) return TRUE; else return FALSE;
}
inline void Bag::Full(){ cout << “Bag is full” << endl; }
inline void Bag::Empty(){ cout << “Bag is empty” << endl; }
void  Bag::Add(int x) {  if(IsFull()) Full; else array[++top] = x;  }
int  *Bag::Delete(int &x){
        if( IsEmpty()) { Empty(); return(0); }
        int deletePos = top/2; x = array[deletePos];
        for(int k=deletePos; k<top; k++){
               array[k] = array[k+1]; // compact upper half of array
        top--; return(&x);
}
```

Parameterized Container Class

```
template <class Type>
class Bag
{
public:
   Bag (int MaxSize = DefaultSize); // constructor
   ~Bag();                  // destructor
   void Add(const Type&);   // insert an element into Bag
   Type *Delete(Type&);     // delete middle element from Bag
   Boolean IsFull();    // return TRUE if the bag is full; FALSE otherwise
   Boolean IsEmpty(); // return TRUE if the Bag is empty; FALSE otherwise
private:
   void Full();     // action when bag is full
   void Empty(); // action when the bag is empty

   Type *array;
   int  MaxSize; // size of array
   int  top;        // highest position in array that contains an element
};
main(){
  Bag<int> a;    Bag<Rectangle> r;    …
}
```

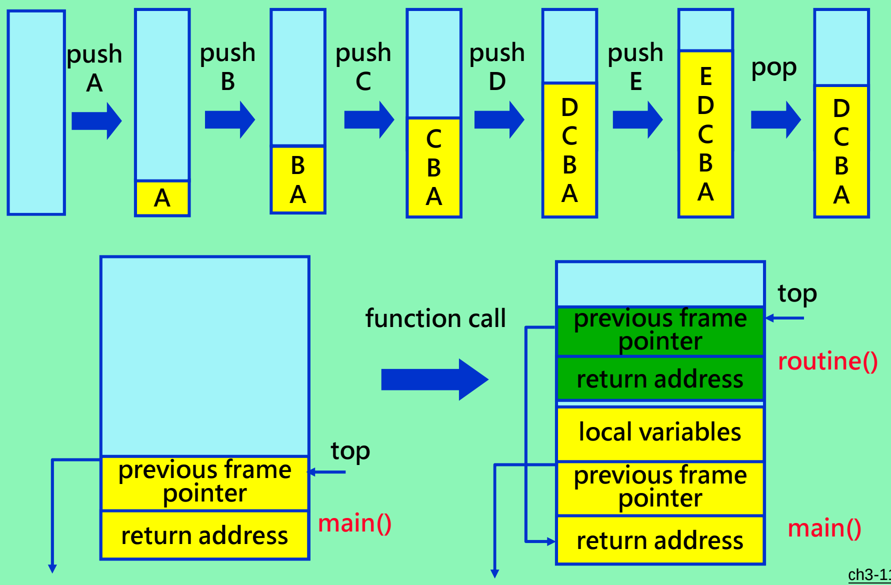
Outline

- **Templates in C++**
- **The Stack Abstract Data Type**
- **The Queue Abstract Data Type**
- **SubTyping and Inheritance in C++**
- **A Mazing Problem**
- **Evaluation of Expressions**

Stack

- **Ordered List**
  - **Suppose A = $a_0, a_1, \ldots, a_{n-1}$, where $n \geqq 0$**
  - **a_i is called an atom, or an element**
- **Stack: Last-In First-Out (LIFO)**
  - **is a special case of ordered list**
  - **the additions and deletions are made at one end called the *top***
  - **E.g., Given a stack S = $(a_0, a_1, \ldots, a_{n-1})$**
  - **a_0 is the bottom element**
  - **a_{n-1} is the top element**

Example: Stack Operation



ADT of Stack

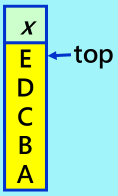
```
template <class KeyType>
class Stack
{
// objects: A finite ordered list of zero or more elements
public:
    Stack (int MaxStackSize = DefaultSize);
    Boolean IsFull();      // return TRUE if Stack is full; FALSE otherwise
    Boolean IsEmpty(); // return TRUE if Stack is empty; FALSE otherwise
    void Add(const KeyType&);
            // if IsFull(), return 0;
            // else insert an element to the top of the Stack
    KeyType *Delete(KeyType&);
            // if IsEmpty(), then return 0;
            // else remove and return a pointer to the top element
private:
    int  top;
    KeyType *stack;
    int  MaxSize;
};
```

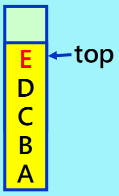
Member Functions of Stack

```
template <class KeyType>
Stack<KeyType>::Stack (int MaxStackSize) : MaxSize (MaxStackSize)
{
        stack = new KeyType[MaxSize];
        top = -1;
}

template <class KeyType>
inline Boolean Stack<KeyType>::IsFull()
{  if(top == MaxSize – 1) return TRUE;   else return FALSE;   }

template <class KeyType>
inline Boolean Stack<KeyType>::IsEmpty()
{    if (top == -1) return TRUE; else return FALSE;    }

```

Push & Pop of Stack

```
template <class KeyType>
void  Stack<KeyType>::Add(const KeyType& x)  // the push operation
{
        if( IsFull() ) StackFull();
        else {
                stack[++top] = x;
        }
}

template <class KeyType>
KeyType* Stack<KeyType>::Delete(KeyType& x) // the pop operation
{
        if (IsEmpty() ) StackEmpty();
        else {
                x = stack[top--];   // assigned the item being deleted to x
                return( &x );
        }
}
```





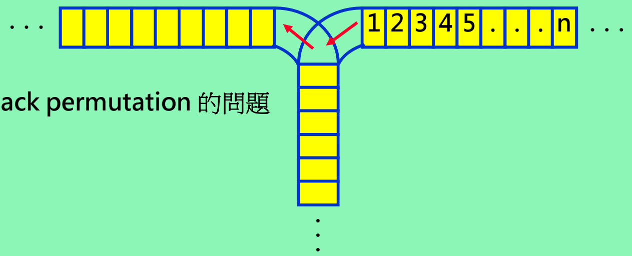
Railroad Switching System

- **Switching Rule**
  - **Initial**: train 1, 2, …, *n* in the top right track segment
  - **Movement**:

    (1) from top-right to the vertical segment one at a time

    (2) from the vertical to the top-left segment one at a time

    (3) The vertical segment operates like a stack
- **Question: What output permutations are not possible?**



3 1 2 …
possible ?

這是一個所謂 stack permutation 的問題

Outline

- **Templates in C++**
- **The Stack Abstract Data Type**
- **The Queue Abstract Data Type**
- **SubTyping and Inheritance in C++**
- **A Mazing Problem**
- **Evaluation of Expressions**

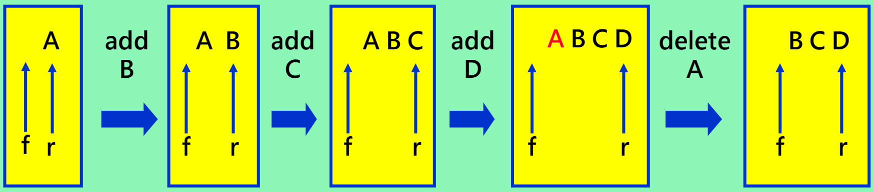
Queue

- **A Queue**
  - **is an ordered list in which all insertions take place at one end and all deletions take place at the opposite end**
  - **is also called First-In First-Out (FIFO)**
- **Example**
  - **Given a queue Q = (a_0, a_1,…, a_{n-1})**
  - **a_0 is the front element, a_{n-1} is the rear element**
  - **a_i is behind a_{i-1} for $1 \leqq i \leqq n$**

Data and Operations of Queue

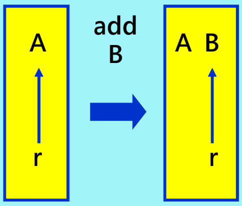
```
template <class KeyType>
class  Queue {
public: …
private:
    int  front;             // front: index of the first element to be retrieved
    int  rear;              // rear: index of the last element
    KeyType  *queue;  // KeyType array
    int  MaxSize;
};
```

f = r = -1 initially



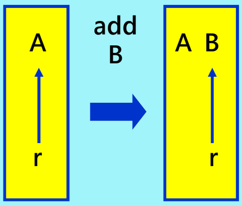
Member Functions of Queue

```
template <class KeyType>
Queue<KeyType>::Queue (int MaxQueueSize) : MaxSize (MaxQueueSize)
{
     queue = new KeyType[MaxSize];
     front = rear = -1;
}

template <class KeyType>
inline Boolean Queue<KeyType>::IsFull()
{
    if( rear == MaxSize –1 ) return TRUE;
    else  return FALSE;
}

template <class KeyType>
inline Boolean Queue<KeyType>::IsEmpty()
{
    if( front == rear ) return TRUE;
    else  return FALSE;
}
```



Add and Delete for Queue

```
template <class KeyType>
void Queue<KeyType>::Add (const KeyType& x)
// add x to the queue
{
     if( IsFull() ) QueueFull();
     else  queue[++rear] = x;
}

template <class KeyType>
KeyType *Queue<KeyType>::Delete(KeyType& x)
// remove front element from the queue
{
    if( IsEmpty() ) { QueueEmpty(); return(); }
    x = queue[++front];
    return x;
}
```



Example: Job Scheduling

- **In Operating System**
  - **jobs are processed in the order they enter the system if no priority is set on jobs**

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]	Q[5]	comments
-1	-1							queue is empty
-1	0	J1						Job 1 joins Q
-1	1	J1	J2					Job 2 joins Q
-1	2	J1	J2	J3				Job 3 joins Q
0	2		J2	J3				Job 1 leaves Q
0	3		J2	J3	J4			Job 4 joins Q
1	3			J3	J4			Job 2 leaves Q

Worst-Case Scenario

front	rear	Q[0]	Q[1]	Q[2]	…	Q[n-1]	Next Operation
-1	n-1	J1	J2	J3	…	J_n	initial state
0	n-1		J2	J3	…	J_n	delete J1
-1	n-1	J2	J3	J4	…	J_{n+1}	add J_{n+1} (J2 to Jn are moved)
0	n-1		J3	J4	…	J_{n+1}	delete J2
-1	n-1	J3	J4	J5	…	J_{n+2}	add J_{n+2}

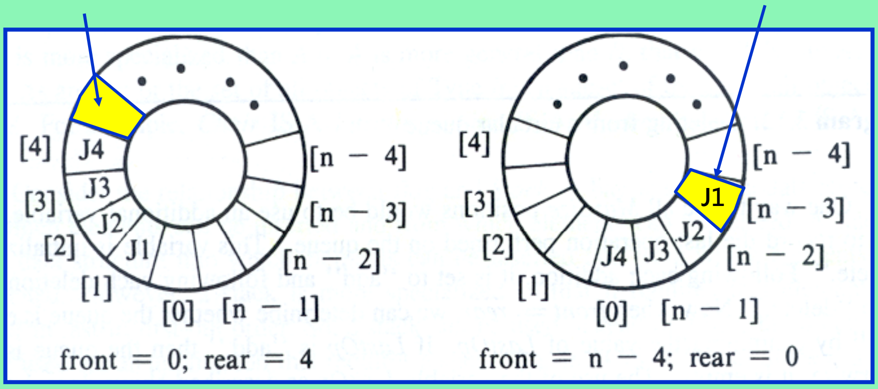
In the above job scheduling,
it takes n-1 steps to add a new job

Circular Queue

Queue size: n
Jobs: J1, J2, J3, J4

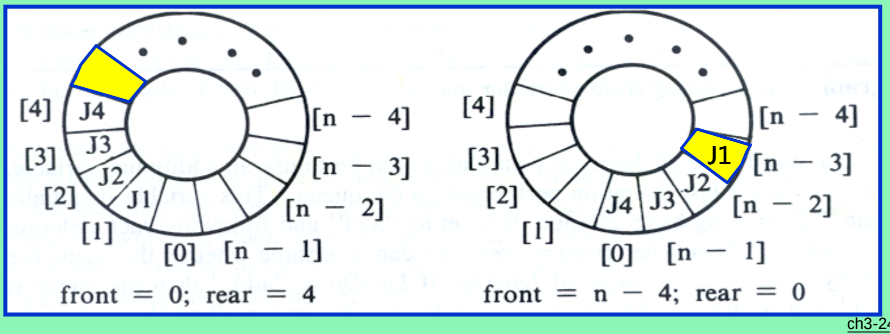
next to add

next to retrieve



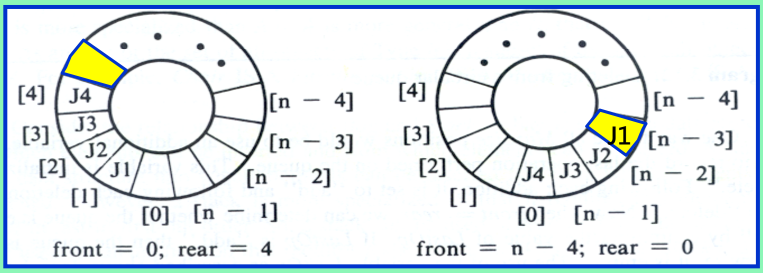
Add An Element to Circular Q

```
template <class KeyType>
void Queue<KeyType>::Add (const KeyType& x)
{
   int  new_rear=(rear+1)% MaxSize;
   if( front == new_rear)  QueueFull();
   else  queue[rear == new_rear] = x;
}
```



Delete An Element From Circular Q

```
template <class KeyType>
KeyType *Queue<KeyType>::Delete (KeyType& x)
// remove front element from queue
{
   if ( front == rear ) {  QueueEmpty(); return(0); }
   front = (front + 1) % MaxSize;
   x = queue[front];   return (&x);
}
```



DeQue

- **Definition**
  - A **double-ended queue (Deque)** is a linear list in which additions and deletions may be made at either end
- **Exercises**
  - Design a data representation that maps a deque into a one-dimensional array
  - Write algorithms to add and delete elements from either end of the queue

Outline

- Templates in C++
- The Stack Abstract Data Type
- The Queue Abstract Data Type
- **SubTyping and Inheritance in C++**
- A Mazing Problem
- Evaluation of Expressions

SubType and Inheritance

- **IS-A Relation**
  - Chair IS-A furniture, Lion IS-A Mammal
  - Rectangle IS-A Polygon
  - Stack IS-A Bag
- **Inheritance**
  - Is used to express IS-A relationship between ADTs.
  - derived class IS-A base class
  - C++ provides a mechanism called public inheritance
    - Ex: class Stack: public Bag
  - The inherited members have the same level of access in the derived class as in the base class

Bag and Stack

```
class Bag
{
public:
    Bag (int MaxSize = DefaultSize); // constructor
    ~Bag(); // destructor
    virtual void Add(int);
    virtual int *Delete(int&);
    virtual Boolean IsFull();
    virtual Boolean IsEmpty();
protected:
    virtual void Full();
    virtual void Empty();
    int  *array;   int MaxSize;   int top;  // data members
};
```

(1) Interface of Bag will be reused in Stack
(2) The implementation of virtual functions may be re-defined in the derived class

```
class Stack : public Bag
{
public:
    Stack (int MaxSize = DefaultSize);
    ~Stack();
    int  *Delete(int&); // delete the element from stack
};
```

constructor, destructor cannot be reused

Example of Derived Class

```
Stack::Stack(int MaxStackSize) : Bag(MaxStackSize) { }
// Constructor for Stack calls constructor for Bag

Stack::~Stack(): ~Bag() { }
// Destructor for Bag is automatically called when Stack is destroyed.
// This ensures that array is deleted

int *Stack::Delete(int& x)
{
    if ( IsEmpty() ) { Empty(); return(0); }
    x = array[ top-- ];
   return(&x);
}
```

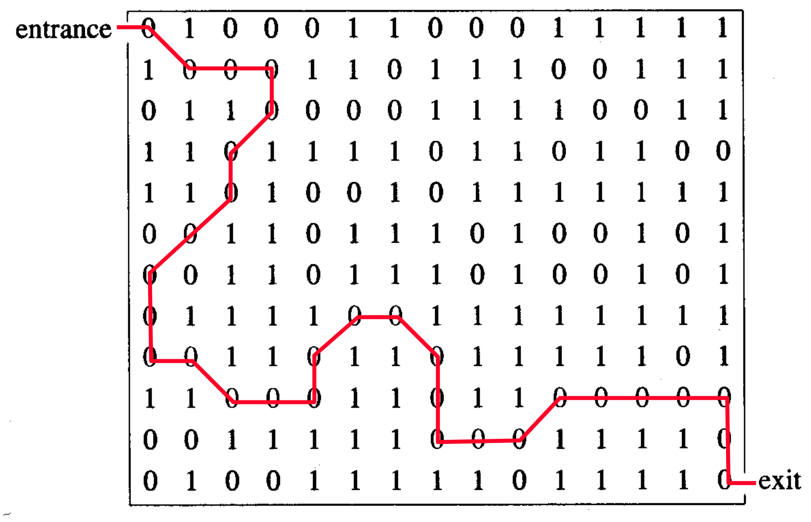
Example: Stack s(3), int *x*

s.Add(1); s.Add(2); s.Add(3);
// Stack::Add is not defined, so use Bag::Add instead
s.Delete(*x*)
// uses Stack::Delete, which calls Bag::IsEmpty and Bag::Empty
because these have not been redefined in Stack

Outline

- Templates in C++
- The Stack Abstract Data Type
- The Queue Abstract Data Type
- SubTyping and Inheritance in C++
- **A Mazing Problem**
- Evaluation of Expressions

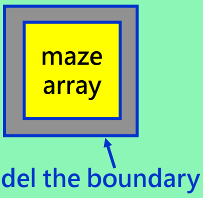
An Example Maze



0	1	0	0	0	1	1	0	0	0	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	0	0	1	1	1
0	1	1	0	0	0	0	1	1	1	1	0	0	1	1
1	1	0	1	1	1	1	0	1	1	0	1	1	0	0
1	1	0	1	0	0	1	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	1	1	1	1	0	0	1	1	1	1	1	1	1	1
0	0	1	1	0	1	1	0	1	1	1	1	1	0	1
1	1	0	0	0	1	1	0	1	1	0	0	0	0	0
0	0	1	1	1	1	1	0	0	0	1	1	1	1	0
0	1	0	0	1	1	1	1	1	0	1	1	1	1	0

Data Structure

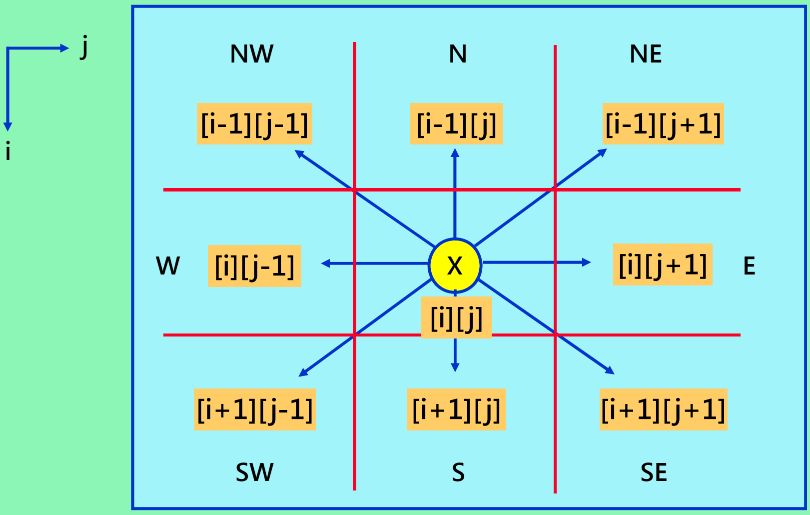
- **Maze**
  - Is represented as a two-dimensional array maze[i][j]
  - maze[i][j]=0: location that can be passed through
  - maze[i][j]=1: blocked location
  - Entrance: maze[0][0]
  - Exit: maze[m][p]
- **To model border condition**
  - The array is declared as maze[m+2][p+2]
  - I.e., the original maze array is surrounded by a border of ones



Strategy of Searching

- **As a rat walks through the maze**
  - (1) He picks a valid move from the current position
    - E.g., starting from north and looking clockwise
  - (2) Put the selected move into a stack
    - So that he can return from a dead path
  - (3) He learns not to make the same mistake twice
    - Avoid getting into a cell visited before
    - A 2-dimensional array, mark[m+2][p+2] is used
    - The mark array records the cells visited before

Allowable Moves



j

i

NW N NE

[i-1][j-1] [i-1][j] [i-1][j+1]

W [i][j-1] X [i][j+1] E

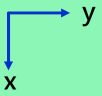
[i][j]

[i+1][j-1] [i+1][j] [i+1][j+1]

SW S SE

Coordinates of the Next Move

The coordinates of the next move is computed by the following data structure



```
struct offsets
{
   int  x, y;
};
enum directions {
   N, NE, E, SE, S, SW, W, NW
};
offsets move[8];
```

→ The SW of (i, j) will be (g, h) where
g = i + move[SW].x;
h = j + move[SW].y

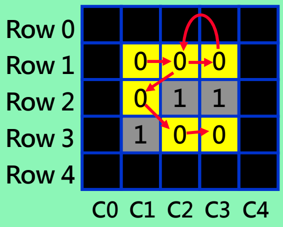
q	move[q].x	move[q].y
N	-1	0
NE	-1	1
E	0	1
SE	1	1
S	1	0
SW	1	-1
W	0	-1
NW	-1	-1

First Pass at Finding A Path Through a Maze

```
initialize stack to the maze entrance coordinates and direction east;
while (stack is not empty)
{
  (i,j,dir) = coordinates and direction deleted from top of stack ;
  while (there are more moves)
  {
    (g,h) = coordinates of next move ;
    if (( g == m ) && ( h == p )) success ;
    if ( (!maze [g ][h])  // legal move
            && (!mark [g ][h])  // haven't been here before
    {
            mark [g ][h] = 1 ;
            dir = next direction to try ;
            add (i,j,dir) to top of stack;
            i = g;  j = h;  dir = north;
    }
  }
}
cout << "no path found" << endl ;
```

An item of the stack:
struct items {
  int *x*, *y*, *dir*;
}

Example: A Mazing Problem



Current Position	Next Legal Move	Stack operation
(1, 1)	(1, 2, E)	Push (1, 1, E)
(1, 2)	(1, 3, E)	Push (1, 2, E)
(1, 3)	No legal move	Pop to backtrack
(1, 2)	(2, 1, SW)	Push (1, 2, SW)
(2, 1)	(3, 2, SE)	Push (2, 1, SE)
(3, 2)	(3, 3, E) success!	Pop out the entire stack

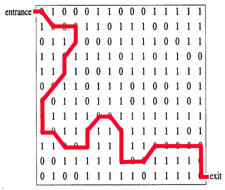
stack

top → (3, 2, E)
(2, 1, SE)
(1, 2, SW)
(1, 1, E)

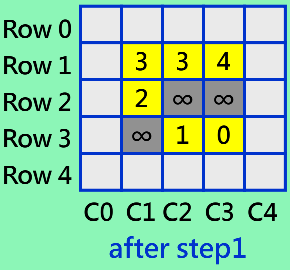
stack right before success

Complete path:
(3, 3) ← (3, 2, E) ← (2, 1, SE) ← (1, 2, SW) ← (1, 1, E)

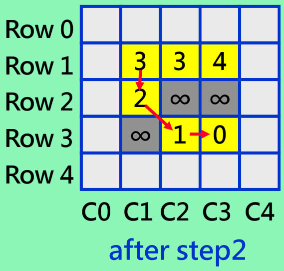
```
void path(int m, int p)
// Output a path (if any) in the maze; maze[0][i] = maze [m +1][i] =
// maze [j][0] = maze [j ][p +1] = 1, 0 ≤ i ≤ p +1, 0 ≤ j ≤ m+1.
{
    // start at (1,1)
    mark[1][1] = 1 ;
    Stack<items> stack(m*p) ;
    items temp ;
    temp.x = 1 ; temp.y = 1 ; temp.dir = E ;
    stack.Add(temp) ;

    while (!stack.IsEmpty()) // stack not empty
    {
        temp = *stack.Delete (temp) ; // unstack
        int i = temp.x ; int j = temp.y ; int d = temp.dir ;
        while (d < 8) // move forward
        {
            int g = i + move[d].a ; int h = j + move[d].b ;
            if ((g == m) && (h == p)) { // reached exit
                // output path
                cout << stack ;
                cout << i << " " << j << endl ; // last two squares on the path
                cout << m << " " << p << endl ;
                return;
            }
            if ((!maze[g][h]) && (!mark[g][h])) { // new position
                mark[g][h] = 1 ;
                temp.x = i ;  temp.y = j ; temp.dir = d+1 ;
                stack.Add(temp) ; // stack it
                i = g ; j = h ; d = N ; // move to (g,h)
            }
            else d++ ; // try next direction
        }
    }
    cout << "no path in maze " << endl ;
}
```



Backward Search



after step1



after step2

Backward_search_algorithm
{
Step 1:
compute the **distance** to the destination of each node by **wave-front propagation**
Step 2:
find a **shortest path** from the source to the destination node by **picking up the nodes with a shortest distance**
}

Outline

- **Templates in C++**
- **The Stack Abstract Data Type**
- **The Queue Abstract Data Type**
- **SubTyping and Inheritance in C++**
- **A Mazing Problem**
- ➡ **Evaluation of Expressions**

Types of Expression

- **Arithmetic Expression**
  - **For example: X = A/B – (C + D * E – A * C)**
  - **The evaluation of this expression is critical in enabling high-level programming**
  - **An expression consists of**
    - **(1) Operands: A, B, C, D, E**
    - **(2) Operator: plus, minus, times, and divide**
    - **(3) Delimiter: like parenthesis "(", ")"**
- **Boolean Expression**
  - **The result in TRUE or FALSE**
  - **Use relational and logical operators**
    - **<, <=, ==, >, >=, &&, ||, !**

Priority of Operator

- **Priority**
  - **The order of operations to be carried out in an expression**
  - **Different order leads to different results**

X = A/B – C + D*E – A*C
  = ((4/2) –2) + (3*3) – (4*2)
  = 0 + 9 – 8
  = 1

X = A / B – C + D*E – A*C
  = (4 / (2 – 2 + 3) * (3 – 4) * 2)
  = (4/3) * (-1) * 2
  = -2.6666

Times and divide
have higher priorities
→ Default

Plus and Minus
have higher priorities
→ Not default

Priority of Operations in C++

Evaluation of operators of the same priority will proceed from left to right
E.g., A/B*C → (A/B) * C

priority	operator
1	Unary minus, !
2	*, /, %
3	+, –
4	<, <=, >, >=
5	==, !=
6	&&
7	||

Postfix Notation

- **Compiler**
  - Translates an expression into a sequence of machine codes
  - It first re-writes the expression into a form called postfix notation
- **Infix notation**
  - The operators come in-between operands
- **Postfix notation**
  - The operators appear after its operands
- **Example**
  - Infix: A / B – C + D * E – A * C
  - Postfix: A B / C – D E * + A C * –

Evaluation of Postfix Notation

- **Scanning the notation from left to right**
- **Store temporary result in T_i, $i \geqq 1$**

Original expression: AB/C-DE*+AC*-	
Operation	**Postfix**
T_1=A/B	T_1C-DE*+AC*-
T_2=T_1-C	T_2DE*+AC*-
T_3=D*E	T_2T_3+AC*-
T_4=T_2+T_3	T_4AC*-
T_5=A*C	T_4T_5-
T_6=T_4-T_5	T_6

The Virtues of Postfix Notation

- **Evaluation is easier on postfix notation**
  - **The need for parenthesis is eliminated**
  - **The priority of operations is no longer relevant**
  - **Evaluation of each operator**
    - **(1) Pop correct number of operands from the stack**
    - **(2) Perform the operation**
    - **(3) Push the results onto the stack**

Evaluation Algorithm

```
 void evaluation(expression e)
// Evaluate the postfix expression e. It is assumed that the last
// token ( a token is either an operator, operand, or ‘#’) in e is ‘#’
// A function NextToken is used to get the next token from e.
{
    Stack<token> stack; // initialize the stack
    for( token x = NextToken(e); x != ‘#’; x = NextToken(e) ){
          if( x is an operand )  stack.Add(x)  // add to stack
          else { // operator
                 pop-up correct number of operands for operator x;
                perform the operation x and store the result onto
                the stack;
          }
    }
}
```

Conceptual Infix to Postfix

- **Algorithm**
  - **(1) Fully parenthesize the expression**
  - **(2) Move all operators so that they replace their corresponding right parentheses**
  - **(3) Delete all parentheses**

Example: A / B – C + D * E – A * C
→ ((((A / B) – C) + (D * E)) – (A * C))

→ A B / C – D E * + A C * –

Ex1: From Infix to Postfix

- **Translate A+B*C to ABC*+**

Next token	Stack	Output
None	Empty	None
A	Empty	A
+	+	A
B	+	AB
*	+ *	AB
The operator * has a higher priority than +, so it is placed on top of +		
C	+ *	ABC

(1) All operands go directly to the output
(2) When the input tokens are exhausted
→ All operators in the stack will be taken off

Ex2: From Infix to Postfix

- **Translate A*(B+C)/D to ABC+*D/**

Next token	Stack	Output
None	Empty	None
A	Empty	A
*	*	A
(	*(	A
B	*(	AB
+	*(+	AB
C	*(+	ABC
)	*	ABC+
/	/	ABC+*
D	/	ABC+*D
‘#’ (ending character)	Empty	ABC+*D/

Priority-Based Stack Operation

- **Left Parenthesis**
  - **Behaves as an operator with high priority when it is not in the stack → incoming priority (icp) = 0**
  - **Behaves as one with low priority when it is in the stack → in-stack priority (isp) = 8**
  - **Only the matching right parenthesis can get an in-stack left parenthesis unstacked**
- **Summary**
  - **Operators are taken out of the stack as long as their in-stack priority is numerically smaller than or equal to the in-coming priority of the new operator**
  - **Assuming that the icp(‘#’) = 8 (lowest)**

Algorithm of Infix to Postfix

```
void postfix (expression e)
// Output the postfix form of the infix expression e. NextToken
// and stack are as in function eval (Program 3.18). It is assumed that
// the last token in e is '#.' Also, '#' is used at the bottom of the stack
{
   Stack<token> stack; // initialize stack
   token y ;
   stack.Add('#');
   for (token x = NextToken(e) ; x != '#' ; x = NextToken(e))
   {
      if (x is an operand)  cout << x ;
      else if (x == ')')  // unstack until '('
         for (y = *stack.Delete (y); y != '('; y = *stack.Delete (y)) cout << y ;
      else { // x is an operator
         for (y = *stack.Delete (y); isp (y) <= icp (x); y = *stack.Delete (y)) cout << y ;
         stack.Add(y);   // restack the last y that was unstacked
         stack.Add(x);
      }
   }

   // end of expression; empty stack
   while (!stack.IsEmpty()) cout << *stack.Delete(y) ;
} // end of postfix
```

higher value
means
lower priority

Using Stack in STL

C++ reference 網頁: http://www.cppreference.com/wiki/start

STL 網頁: http://www.cppreference.com/wiki/stl/start

STL Stack 網頁: http://www.cppreference.com/wiki/stl/stack/start

Example: **the following code uses empty() as the stopping condition on a while loop to clear a stack and display its contents in reverse order:**

```
stack<int> s;
for( int i = 0; i < 5; i++ )  { s.push(i); }
while( !s.empty() )  { cout << s.top() << endl; s.pop(); }
```

The End of Chapter 3

Next Topic:

Linked Lists

國立清華大學 電機工程學系
EE2410 Data Structure

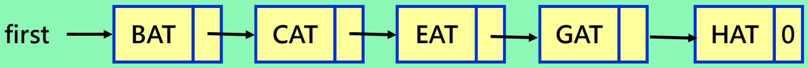
Chapter 4
Linked List (Part I)

Outline

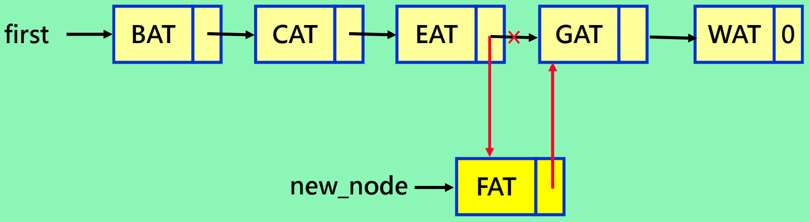
- **Singly Linked Lists**
- **A Reusable Linked List Class**
- **Circular Lists**
- **Linked Stacks and Queues**
- **Polynomials**

Why Linked List?

(Initial Linked List)



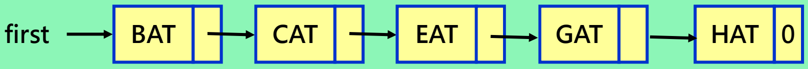
(Quick Insertion of a new node)



inserted !

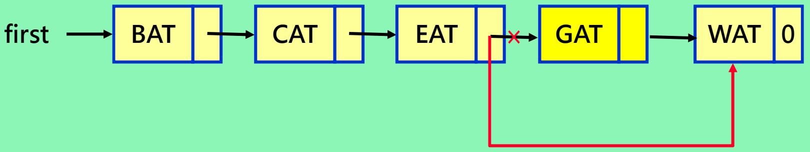
Deletion of A Node

(Initial Linked List)



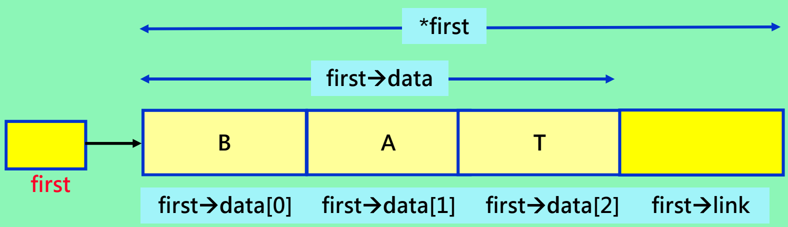
(Quick deletion of a new node)

deleted !



Ex: Access The Data Of a Node

```
class ThreeLetterNode {
private:
      char  data[3];
      ThreeLetterNode *link;
};
main(){
   …
   ThreeLetterNode *first;  /* The data references are shown below
   …
}
```



➡ Illegal access because data[] and link are private data !

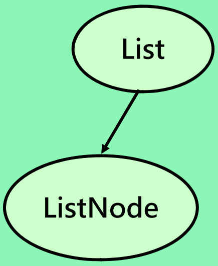
Dilemma of Node Data

- **Declaring the node data as public**
  - **will allow one to access the data through pointer**
  - **but the data encapsulation principle is violated**
- **Declaring the node data as private**
  - **will requires another member functions to access the data**
  - **E.g., Get_link(), Set_link(), Get_data(), Set_data()**
  - **The access is less efficient**

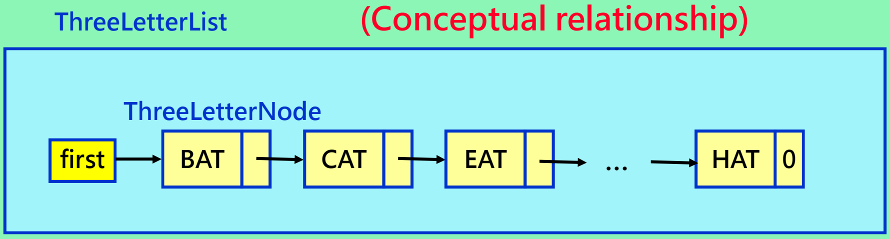
Composite Class

- **Has-A Relationship**
  - **We say that a data object of Type A HAS-A data object of Type B if A conceptually contains B or B is part of A**

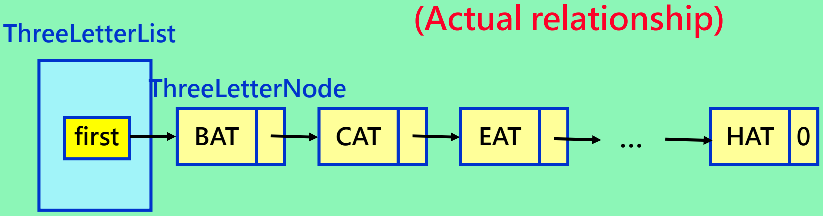
```
class ThreeLetterList; // forward declaration
class ThreeLetterNode{
friend class ThreeLetterList;
private:
       char  data[3];
       ThreeLetterNode  *link;
};
class ThreeLetterList {
public:
       // List Manipulation operations
…
private:
      ThreeLetterNode  *first;
};
```



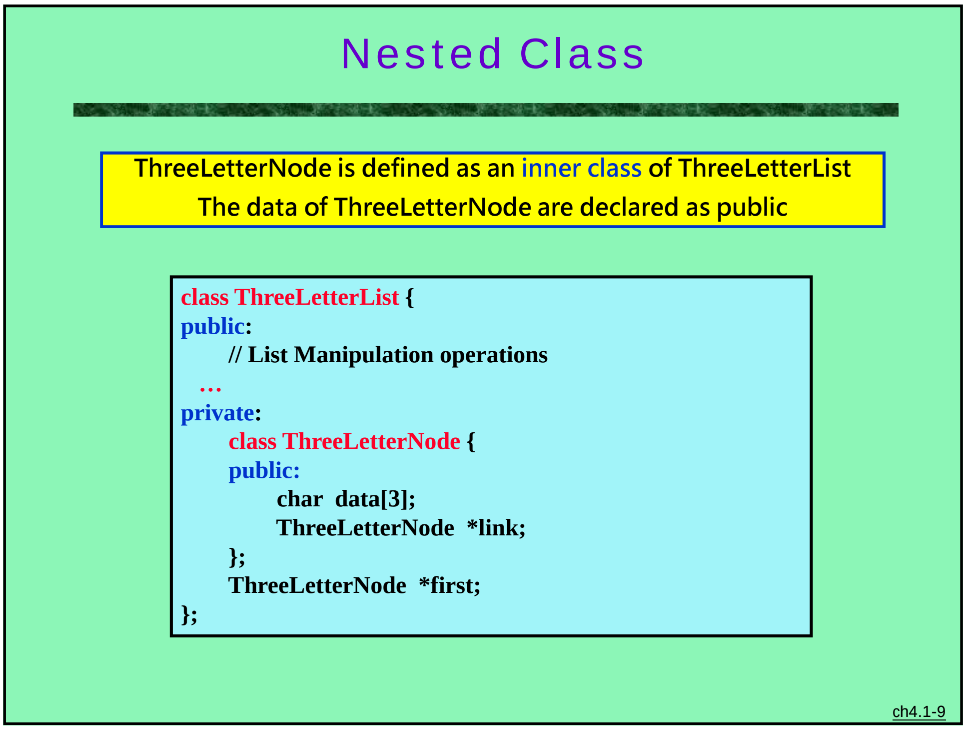
Relationship of List and Node


ThreeLetterList

(Conceptual relationship)


ThreeLetterList

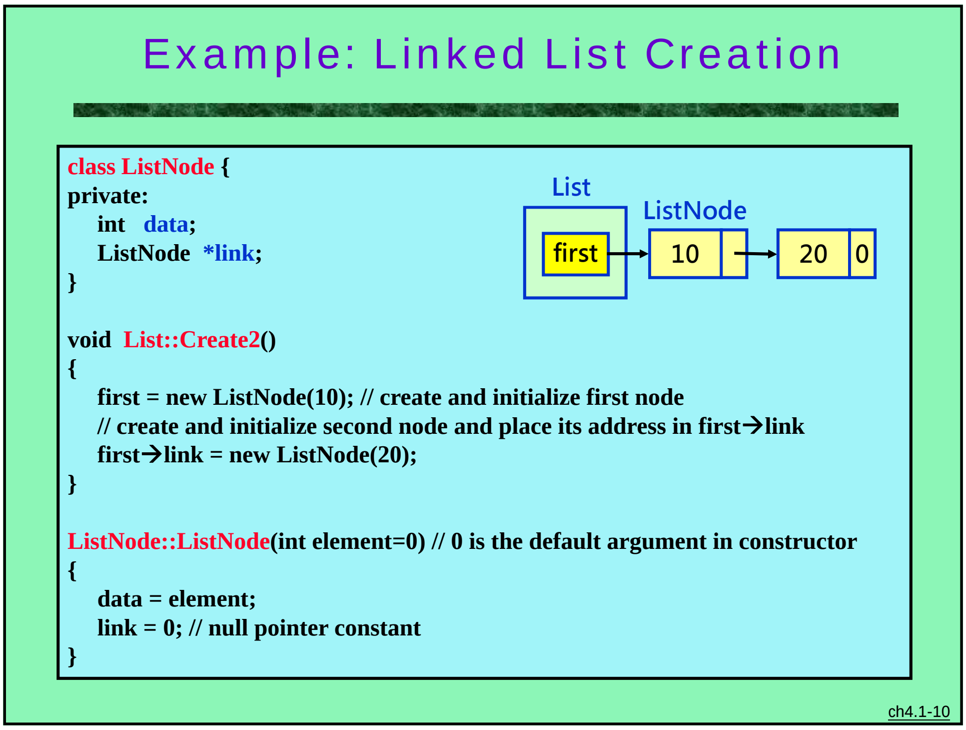
(Actual relationship)



Nested Class

ThreeLetterNode is defined as an inner class of ThreeLetterList
The data of ThreeLetterNode are declared as public

```
class ThreeLetterList {
public:
        // List Manipulation operations
   …
private:
        class ThreeLetterNode {
        public:
                char  data[3];
                ThreeLetterNode  *link;
        };
        ThreeLetterNode  *first;
};
```

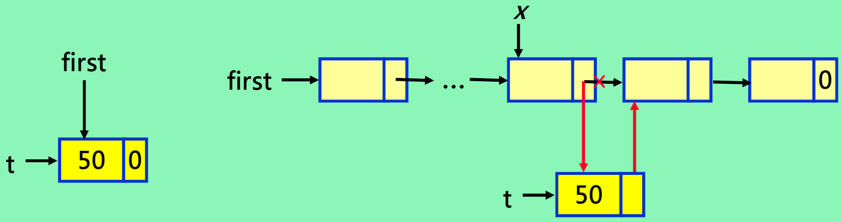


Example: Linked List Creation

```
class ListNode {
private:
    int   data;
    ListNode  *link;
}

void  List::Create2()
{
    first = new ListNode(10); // create and initialize first node
    // create and initialize second node and place its address in first→link
    first→link = new ListNode(20);
}

ListNode::ListNode(int element=0) // 0 is the default argument in constructor
{
    data = element;
    link = 0; // null pointer constant
}
```

Example: Linked List Insertion

```
void  List::Insert50(ListNode *x)
{
    ListNode  *t = new ListNode (50); // create and initialize new node
    if( ! first ) // insert into empty list
    {
        first = t;
        return; // exit List::Insert50
    }
    // insert after x
    t → link = x → link;
    x → link = t;
}
```



Outline

- Singly Linked Lists
- A Reusable Linked List Class
- Circular Lists
- Linked Stacks and Queues
- Polynomials

Template Definition of Linked List

```
template <class Type>  class List; // forward declaration

template <class Type>
class  ListNode {
friend class  List<Type>;
private:
      Type  data;
      ListNode  *link;
};

template  <class Type>
class   List {
public:
      List() { first = 0; };  // constructor initializing first to 0
      // list manipulation operations
      …
private:
     ListNode<Type>  *first;
};
```

A linked list of integers declared as:
    List<int> intlist;

Direct Traversal of a List

pseudo-code

```
1.      // initialize a container C
2.      int  x = -MAXINT;
3.      for each item in Container C
4.      {
5.           current = current item of C;
6.           x = max(current, x);  // body
7.      }
8.      return (x);  // post-processing step
```

Direct Traversal

(Revision of statements 3 to 7 for integer container)

```
for ( ListNode<int> *ptr = first;     ptr !=0;     ptr = ptr → link )
{
    current = ptr → data;
    x = max(current, x);
}
```

in a member of List<Type>

Drawbacks of Direct Traversal of a Container Class

- In a template class like List<Type>
  - Operations should be independent of the type, while direct traversal depends on the type of elements
  - For example, it does not make sense to compute the sum of a Rectangle container
- A new function requires traversal
  - of a container class needs the support of a new class member function
  - This is difficult because the class provider and class users might be different in a programming team
- Even if class user is allowed to add a new member function
  - He or she would need to know how the container is implemented

---

Linked List Iterator

- An Iterator
  - is an object that is used to traverse all the elements of a container class C
  - useful in operations like
    - (1) print all integers in C
    - (2) Obtain the maximum, minimum, mean, or median of all integers in C
    - (3) Obtain the sum, product, or sum of squares of all integers in C
    - (4) Obtain all integers in C that satisfy some property P (e.g., integers that are positive, or the square of an integer, etc.)
    - (5) Obtain the integer *x* from C such that, for some function *f*, *f*(*x*) is the maximum

Linked List With Iterator (I)

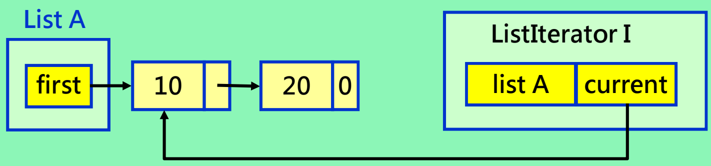
```
enum  Boolean  { FALSE, TRUE };
template <class Type> class List;
template <class Type> class ListIterator;

template <class Type> class ListNode{
friend class List<Type>;
friend class ListIterator<Type>;
private:
      Type  data;
      ListNode  *link;
};
template  <class Type> class List {
friend class ListIterator<Type>;
public:
      List() { first = 0; };  // constructor initializing first to 0
      // list manipulation operations  …
private:
     ListNode<Type>  *first;
};
TO BE CONTINUED
```

兩個朋友

一個朋友

Linked List With Iterator (II)

```
template <class Type> class ListIterator {
public:
     ListIterator( const List<Type>& l):  list(l), current (l.first) {};
     Boolean  NotNull();
     Boolean NextNotNull();
     Type  *First();
     Type  *Next();
private:
     const  List<Type>& list;       // refers to an existing list
     ListNode <Type>* current;  // points to a node in list
};
```

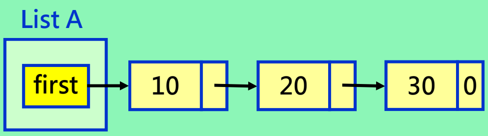


List Iterator Functions

```
template <class Type> // check that the current element in List is non-null
Boolean ListIterator<Type>::NotNull() {
   if ( current ) return TRUE;   else  return FALSE;
}
template <class Type> // check that the next element in List is non-null
Boolean ListIterator<Type>::NextNotNull() {
   if ( current && current → link)  return TRUE;   else  return FALSE;
}
template <class Type> // return a pointer to the first element of List
Type *ListIterator<Type>::First() {
   if ( list.first )  return (&list.first→data);   else  return 0;
}
template <class Type> // return a pointer to the next element of List
Type *ListIterator<Type>::Next()
   if ( current ) {
       current = current → link;
       if ( current )  return (&current→data);
  }
  else  return 0;
}
```

Example: Usage of Iterator

```
int  sum( const List<int>& input_list)
{
   ListIterator<int>  l (input_list); // l is associated with list input_list
   if( ! l.NotNull())  return 0;  // return 0 if the list is empty

   int  ret_value = *l.First();       // get the first element's pointer
   while ( l.NextNotNull() ) {      // iteratively sum up every element's value
            ret_value += *l.Next(); // get it, add it to the current total
   }
   return (ret_value);
}
```



A Stylish Foreach Macro

- **Assume List, ListNode, and ListIterator**
  - **have been defined as above**
- **Variables**
  - **list: an object of List**
  - **data_ptr: an object of ListNode's data pointer**
  - **gen: an object of ListIterator for list**

```
#define  List_foreach(gen, node)   \
         for ( node = gen.First();    \
               gen.NotNull();          \
               node = gen.Next();     \
         )
```

Macro Definition

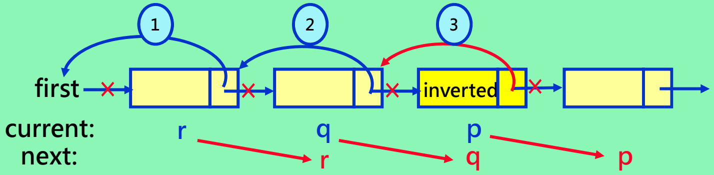
```
sum = 0;
ListIterator  gen(list);
List_foreach(gen, node){
    sum += *node;
}
```

Usage of List_foreach

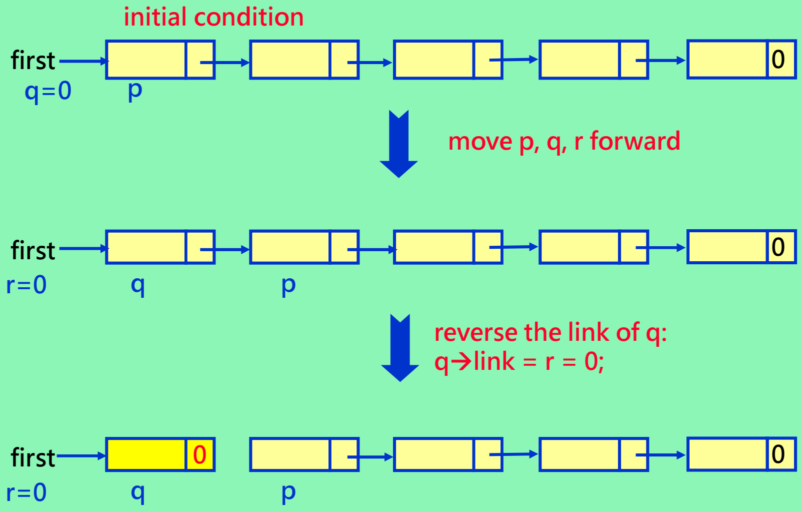
Inverting a Linked List (or Chain)

```
template <class Type> void List<Type>::Invert()
// A chain x is inverted so that if x = (a1, a2, …, a_n)
// then after execution, x = (a_n, a_n-1, …, a_1)
{
   ListNode<Type>  *p = first;  *q = 0; // q trails p
   while(p) {
       ListNode<Type>  *r = q;   q = p; // r trails q
       p = p → link;  // remember the next node of the node being inverted q
       q → link = r;  // link q to the preceding node r
   }
   first = q;
}
```

- **Inverting**
  - **can be done "in place" in linear time O(n)**



Process of Inverting a List



initial condition

first

q=0

p

0

move p, q, r forward

first

r=0

q

p

0

reverse the link of q:
q→link = r = 0;

first

0

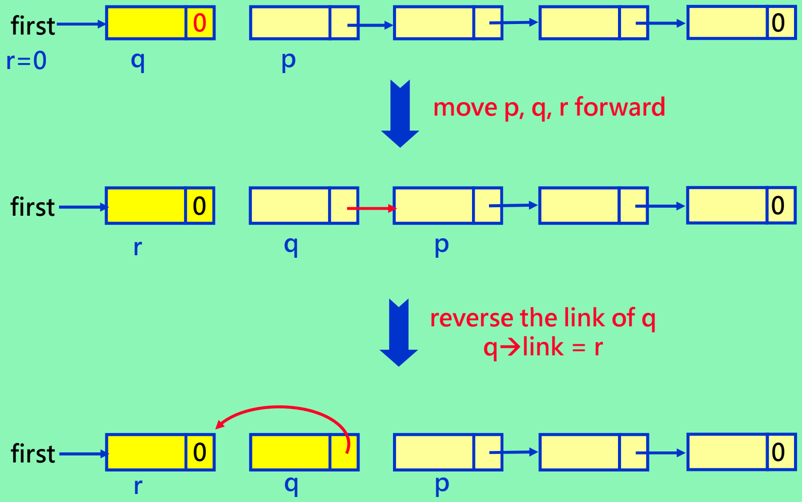
r=0

q

p

0

Process of Inverting a List



first

0

r=0

q

p

0

move p, q, r forward

first

0

r

q

p

0

reverse the link of q
q→link = r

first

0

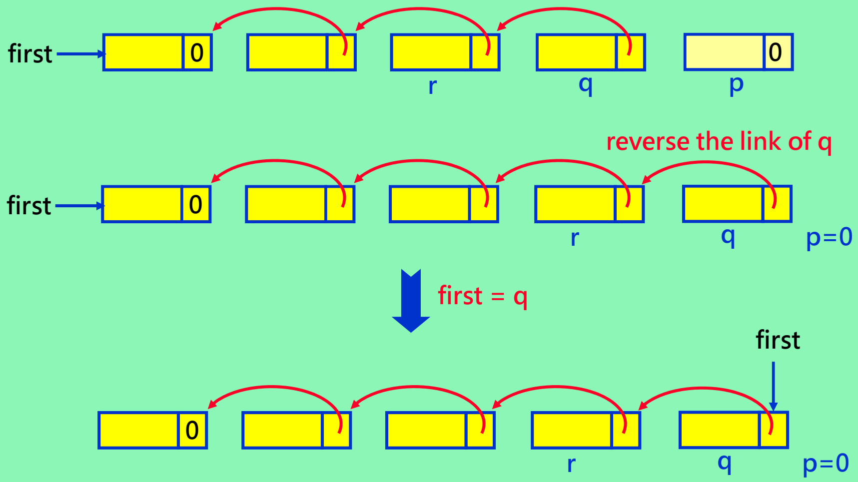
r

q

p

0

Process of Inverting a List



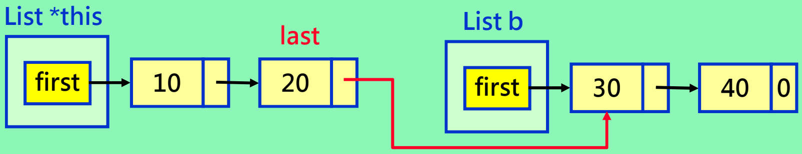
Final reversed linked list

Concatenating Two Chains

- **The complexity is also linear**

```
template <class Type>
void  List<Type>::Concatenate ( List<Type> b)
// this = (a1, a2, …, an) and b = (b1, b2, …, bm) m, n ≧ 0
// produces the new chain z = (a1, a2, …, an, b1, b2, …, bm) in this
{
  if ( ! first )  {  first = b.first;  return; }
  if ( b.first ) {  // finding the last node of *this
       for (ListNode<Type> *p = first;  p→link;   p = p → link);  // no body
       p → link = b.first;
  }
}
```

false when p is the last in *this



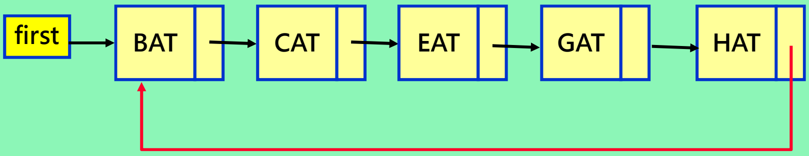
Outline

- Singly Linked Lists
- A Reusable Linked List Class
- Circular Lists
- Linked Stacks and Queues
- Polynomials

Basics of Circular List

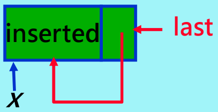
- Major Features
  - The link field of the last element points to the first element
  - Check if last element:

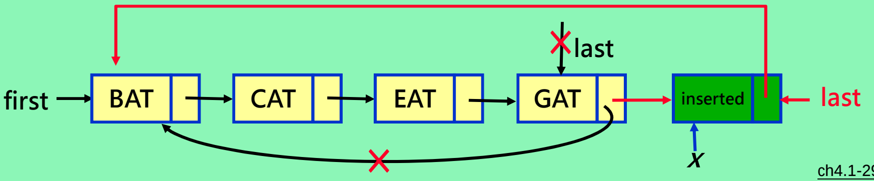
    (current → link == first) instead of (current → link = 0)



Insertion-To-Rear Function

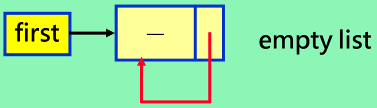
```
template <class Type>
void  CircularList::Insert_to_Rear( ListNode<Type> *x)
// insert the node pointed at by x at the rear of the circular
// list *this, where last points to the last node in the list
{
   if ( ! last )  {   // empty list
        last = x;  x → link = x;
   }
   else {
        x → link = last → link;
        last → link = x;
        last = x;
   }
}
```

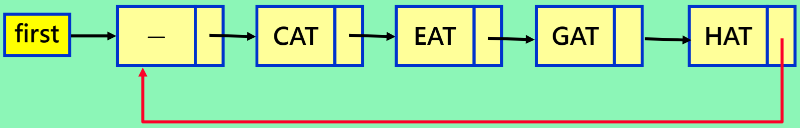




Dummy Head Node

- **In some applications**
  - **using simple circular list structure cause problems as the empty list has to be handled as a special case**
- **To avoid such as special cases**
  - **a dummy head node is introduced**

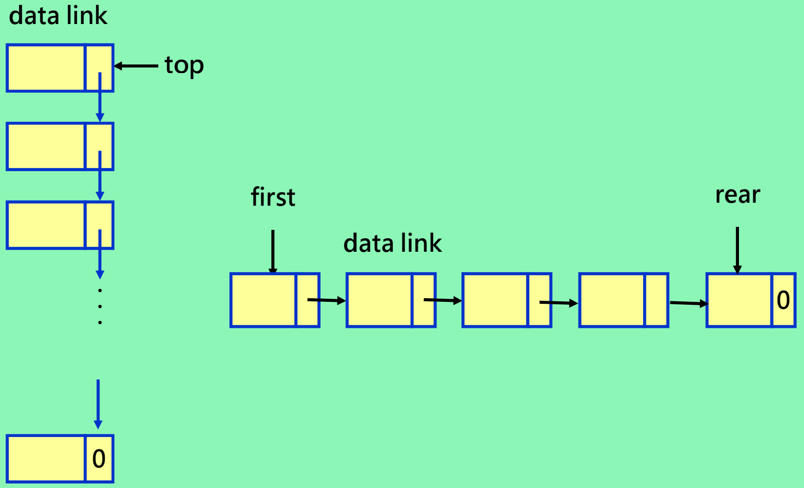




Outline

- Singly Linked Lists
- A Reusable Linked List Class
- Circular Lists
- Linked Stacks and Queues
- Polynomials

Linked Stack and Queue



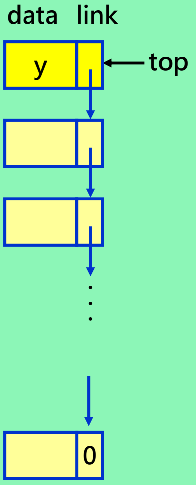
Stack Class Definition

```
class  Stack;  // forward declaration

class StackNode {
friend  class  Stack;
private:
    int   data;
    StackNode  *link;
    StackNode ( int d = 0, StackNode *l = 0) : data (d), link (l) {} ; // constructor
};

class Stack {
public:
   Stack() { top = 0; } ; // constructor
   void  Add(const int);
   int* Delete (int&);
private:
   StackNode  *top;
   void  StackEmpty();
};
```

Stack Add and Delete

```
void  Stack::Add ( const int y) {
    top = new StackNode (y, top);
}                               old top
```

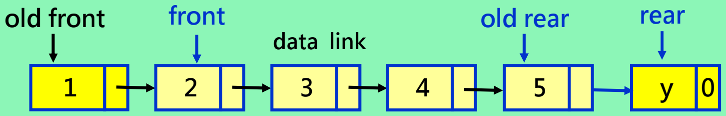
```
int  *Stack::Delete ( int& retvalue)
// Delete top node from stack and return a pointer to its data
{
    if (top == 0) {  StackEmpty(); return 0; }
     // return null pointer constant
    StackNode  *delnode = top;
    retvalue = top → data; // get data field of top node
    top = top→link;           // update the top pointer
    delete  delnode;           // free the node
    return   &retvalue;      // return pointer to data
}
```



Queue Add and Delete

```
void  Queue::Add ( const int y)
{
    if ( front == 0 )  front = rear = new  QueueNode(y, 0); // empty queue
    else   rear = rear → link = new  QueueNode(y, 0);
    // attach node and update rear
}
```

```
int  *Queue::Delete ( int& retvalue)
// Delete the first node in queue and return a pointer to its data
{
    if (front == 0) {  QueueEmpty(); return 0; } // return null pointer constant
    QueueNode  *x = front;
    retvalue = front → data; // get data
    front = x → link;           // update front node
    delete  x;                  // free the node
    return   &retvalue;  // return pointer to data
}
```



Outline

- **Singly Linked Lists**
- **A Reusable Linked List Class**
- **Circular Lists**
- **Linked Stacks and Queues**
- **Polynomials**

Is-Implemented-By Relationship

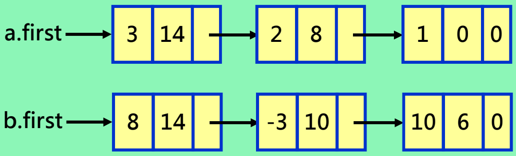
- **Definition**
  - A data object of Type A **IS-IMPLEMENTED-IN-TERMS-OF** a data object of Type B if the Type B object is central to the implementation of Type A object.
  - This relationship is usually expressed by declaring the **Type B object as a data member** of the Type A object
- **Next: Polynomial implemented by linked list**

Polynomial Class

```
struct  Term
// all members of Term are public by default
{
   int   coef; // coefficient
   int   exp;  // exponent
   void  Init ( int c, int e ) {  coef = c;  exp = e; } ;
};
class Polynomial
{
friend  Polynomial  operator+( const Polynomial&, const Polynomial&);
private:
     List<Term> poly;
};
```

$a = 3x^{14} + 2x^{8} + 1$

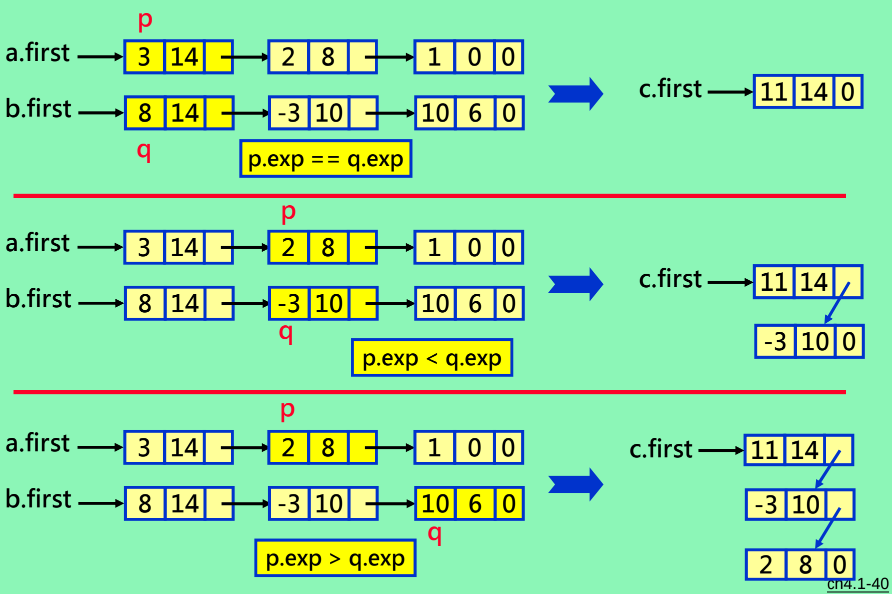
$b = 8x^{14} - 3x^{10} + 10x^{6}$



```
Polynomial operator+(const Polynomial& a , const Polynomial& b) {
// Polynomials a and b are added and the sum returned
    Term *p, *q, temp ;
    ListIterator <Element > Aiter (a.poly) ; ListIterator <Element > Biter (b.poly) ;
    Polynomial c ;
    p = Aiter. First () ; q = Biter. First () ; // get first node in a and b
    while (Aiter . NotNull () && Biter . NotNull ()) { // current node is not null
        switch (compare(p →exp,q →exp)) {
            case '=':
                int x = p →coef + q →coef ; temp . Init (x,q →exp) ;
                if (x) c . poly . Attach (temp) ;
                p = Aiter.Next () ; q = Biter.Next () ;  // advance to next term
                break;
            case '<':
                temp.Init(q →coef, q →exp) ; c.poly.Attach (temp) ;
                q = Biter.Next () ; // next term of b
                break;
            case '>':
                temp . Init(p →coef, p →exp) ; c.poly.Attach (temp) ;
                p = Aiter . Next () ; // next term of a
        }
    }
    while (Aiter.NotNull ()) {  // copy rest of a
        temp.Init(p →coef, p →exp) ; c.poly.Attach (temp) ;
        p = Aiter.Next () ;
    }
    while (Biter.NotNull ()) {  // copy rest of b
        temp.Init(q →coef, q →exp) ; c.poly.Attach (temp) ;
        q=Biter.Next () ;
    }
    return c ;
}
```

adding
two polynomials

Generating The First Three Terms



Analysis of Operator+

- Computing Time
  - (1) coefficient additions
  - (2) exponent comparisons
  - (3) addition/deletions to available space
  - (4) creation of new nodes
- Assume that
  - polynomial *a* has *m* terms, while *b* has *n* terms
- Coefficient additions: [0, min{m, n}] times
  - Lower-bound: when none of the exponents are equal
  - Upper-bound: when the exponents of one polynomial are a subset of the exponents of the other polynomial
- Overall Complexity: O(m+n)

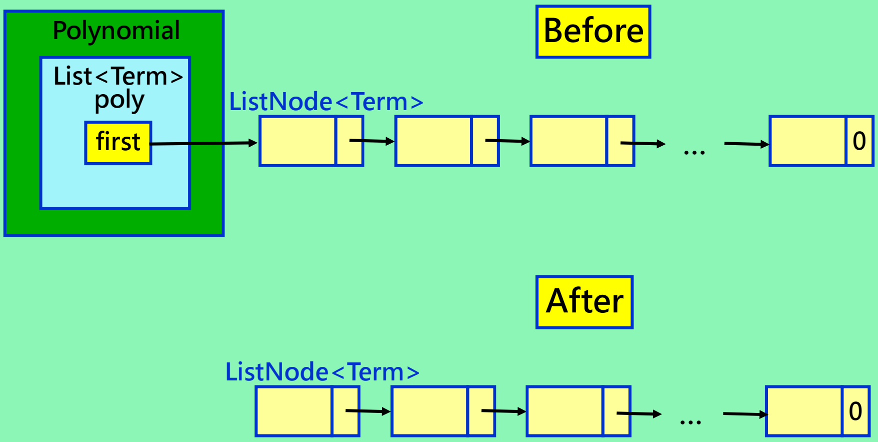
Example: Polynomial Computation

- d(x) = a(x) * b(x) + c(x)

```
void  func()
{
    Polynomial  a, b, c, d, t;
    cin >> a; // read and create polynomial
    cin >> b;
    cin >> c;
    t = a * b;
    d = t + c;
    cout << d;
}
```

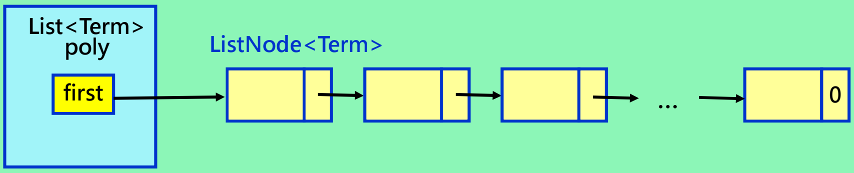
Problem:
When the function terminates, the memory occupied by the polynomials a, b, c, d, t may not be freed automatically
→ because ListNode<Term> objects are not physically contained in List<Term> objects.

Polynomial Object Before And After It Goes Out Of Scope



Erase A Polynomial

```
template <class Type>
List<Type>::~List()
// Free all nodes in the chain
{
    ListNode<Type> *next;
    for (; first; first = next) {
          next = first → link
          delete first;
    }
}
```



Space Management Of ListNodes

- **By incorporating Circular List**
  - **Freeing all the nodes in a list is more efficient**
- **Reuse strategy**
  - **Deletion: nodes that have been "deleted" are actually maintained in a pool → available (av) space**
  - **Request for a new node:**
    - **(1) If available space is not empty, recycle one of them**
    - **(2) If available space is empty → by "new" command**

Getting and Returning A Node

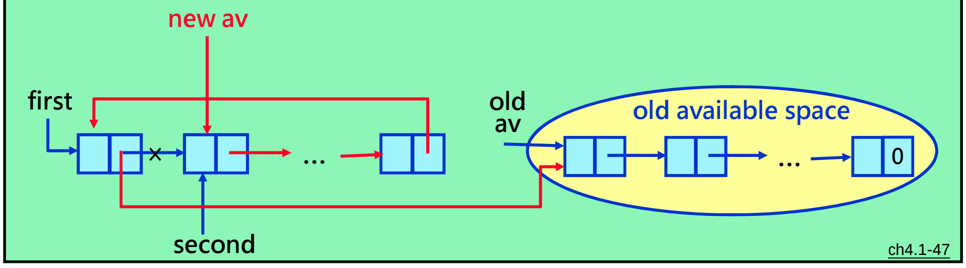
```
template <class Type>
ListNode<Type>* CircularList::GetNode()
// Provide a node for use
{
    ListNode<Type>* x;
    if ( ! av )  x = new ListNode<Type>;   // request for a new one
   else  {  x = av;   av = av → link; }        // recycle one from AV-pool
   return x;
}
```

```
template <class Type>
void  CircularList<Type>::RetNode(ListNode<Type>* x)
// Free the node pointed by x
{
    x → link = av;
   av = x;
}
```

Erasing A Circular List

```
template <class Type>
void CircularList<Type>::~CircularList()
// Erase the entire circular list pointed by first
{
   if ( first ) {
        ListNode*  second = first → link; // second node
        first → link = av;    // first node linked to av
        av = second;   // second node of list becomes front of av list
        first = 0;
   }
}
```

A Circular List can be erased in a fixed amount of time
→ Independent of the number of nodes in the list



```
Polynomial  operator+ (const Polynomial& a, const Polynomial& b)
{
    Term  *p, *q, temp;
   CircularListIterator<Term>  Aiter (a.poly);
   CircularListIterator<Term>  Biter (b.poly);
   Polynomial  c;  // assume the constructor creates a head node with exp = -1
   p = Aiter.first();   q = Biter.first();
   while(1) {
      switch ( compare ( p→exp, q→exp ) )  {
         case ‘=‘:
              if ( p → exp == -1 ) return c;
             else {
                   int  sum = p → coef + q → coef;
                   if (sum) {  temp.Init (sum, q → exp);  c.poly.Attach ( temp);  }
                   p = Aiter.Next();   q = Biter.Next();
              }
              break;
         case ‘<‘:
                  temp.Init(q→coef, q→exp); c.poly.Attach(temp); q=Biter.Next(); break;
         case ‘>’:
                  temp.Init(p→coef, p→exp); c.poly.Attach(temp); p=Aiter.Next(); break;
      }  // end of switch and while
   }
}
```

Adding circularly represented polynomials

Using List Iterator in STL

C++ reference 網頁: http://www.cppreference.com/wiki/start

STL 網頁: http://www.cppreference.com/wiki/stl/start

STL List 網頁: http://www.cppreference.com/wiki/stl/list/start

```
// Create a list of characters
list<char> my_list;
for( int i = 0; i < 10; i++ ) {
  my_list.push_front( i + 'a' );
}
// Display the list
list<char>::iterator it;
for( it = my_list.begin(); it != my_list.end(); ++it) {
    cout << *it;
}
```

The End of Part I

Next Topic:

Linked Lists Part II

國立清華大學 電機工程學系
EE2410 Data Structure

Chapter 4
Linked List (Part II)

Outline

- **Equivalence Class**
- **Sparse Matrices**
- **Doubly Linked Lists**
- **Generalized Lists**
- **Virtual Functions and Dynamic Binding**

Equivalence Relation

- **A relation is a set of pair (x, y)**
  - where x and y are elements in a set, say S
- **Three properties of an equivalence relation**
  - **Reflexive**: $x \equiv x$
  - **Symmetric**: If $x \equiv y$, then $y \equiv x$
  - **Transitive**: If $x \equiv y$ and $y \equiv z$ then $x \equiv z$
- **Definition**
  - A relation over a set S, is said to be an equivalence relation over S iff it is symmetric, reflexive, and transitive over S.
  - E.g., "equal to" (=) relationship is an equivalence relation

---

Finding Equivalence Classes

- **Input**

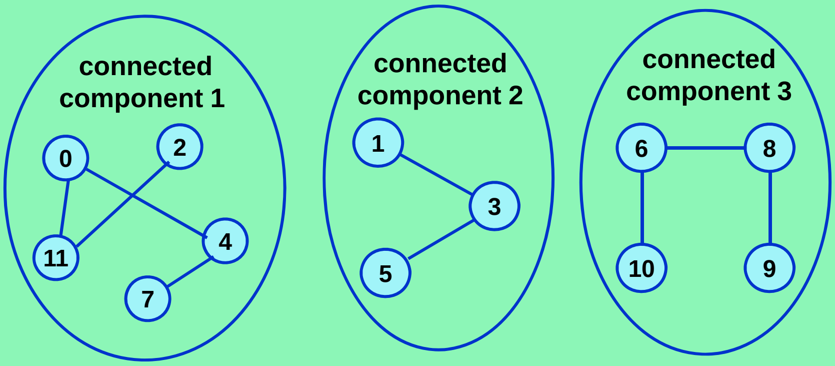
  $0 \equiv 4$, $3 \equiv 1$, $6 \equiv 10$, $8 \equiv 9$, $7 \equiv 4$, $6 \equiv 8$, $3 \equiv 5$, $2 \equiv 11$, $11 \equiv 0$
- **Output**
  - Three equivalence classes by applying the three properties:

    { 0,2,4,7,11}; { 1,3,5 }; { 6,8,9,10 }
- **Basic Algorithm**
  - (phase 1): equivalence pairs (i, j) are read in and stored
  - (phase 2): find each equivalence class by transitive rule

Relation Graph

- **Input**

0≡4, 3≡1, 6≡10, 8≡9, 7≡4, 6≡8, 3≡5, 2≡11, 11≡0

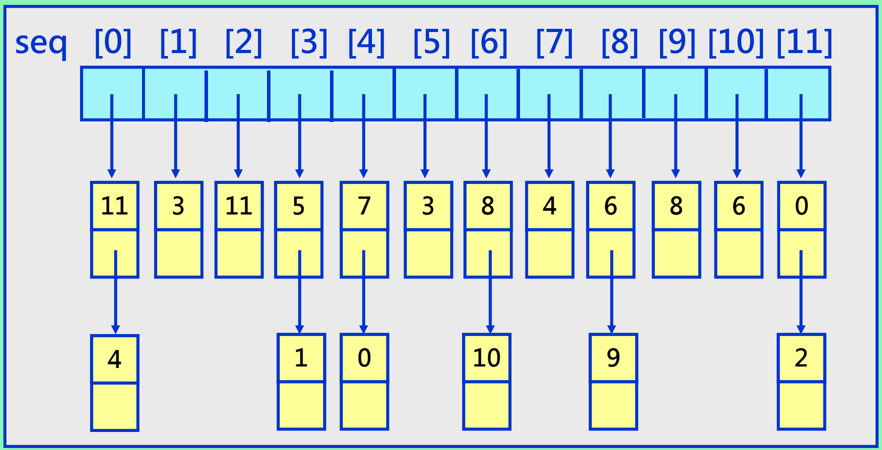


A Graph consists of vertices and edges

Data Structure For Storing the Equivalence Pairs

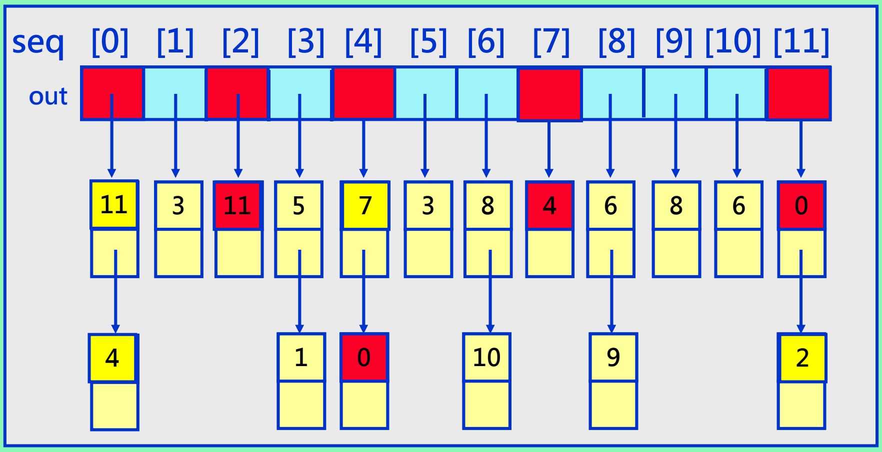
Equivalence pairs:

0≡4, 3≡1, 6≡10, 8≡9, 7≡4, 6≡8, 3≡5, 2≡11, 11≡0



Finding Equivalence Class Containing Node 0

Processing order: [0] → [11] → [4] → [2] → [7]
equivalence class containing 0 : { 11, 4, 2, 7}

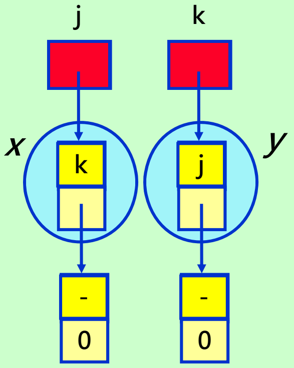


Overall Equivalence Class Computation

```
void equivalence()
{
   read  n;  // read in number of objects
  initialize seq to 0 and out to FASLE;
  while (more pairs) // input pairs
  {
       read the next pair (i, j);
       put j on the seq[i] list;
       put i on the seq[j] list;
   }
   /*------- print out equivalence classes -----*/
   for (i=0; i<n; i++){
       for( out[i] == FALSE ) {
            out[i] = TRUE;
            output the equivalence class that contains object i;
       }
   }
};
```

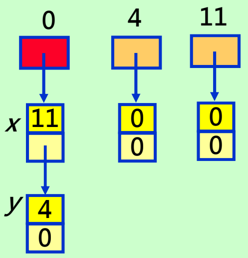
Data Structure in C++

```
enum  Boolean { FALSE, TRUE };

class ListNode {
friend void equivalence ();
private:
    int           data;
    ListNode  *link;
    ListNode(int);  // private constructor
};

typedef  ListNode *ListNodePtr;
// so we can create an array of pointers using new

ListNode::ListNode(int d)  // constructor
{
    data = d;
    link = 0;
}
```



```
void  equivalence ()
// Input the equivalence pairs and output the equivalence classes
{
  ifstream  inFile("equiv.in", ios::in); // "equiv.in" is the input file
  if ( ! inFile ) {
     cerr << "Cannot open input file" << endl;
     return;
  }
  int   i, j, n;
  inFile >> n;  // read number of objects
  // initialize seq and out
  ListNodePtr  *seq = new ListNodePtr[n];
  Boolean  *out = new Boolean[n];
  for (k=0; k<n; k++){
        seq[k] = 0;
        out[k] = FALSE;
  }
  // Phase 1: input equivalence classes
  inFile >> j >> k ;
  while ( inFile.good() ) {  // check end of file
        ListNode  *x = new ListNode(k);  x→link = seq[j]; seq[j] = x; // add k to seq[j];
        ListNode  *y = new ListNode(j);  y→link = seq[k]; seq[k] = y; // add j to seq[k];
        inFile >> j >> k;
   }
```

```
void  equivalence () // Phase 2: output equivalence classes
  …. (previous page)
for (k=0; k<n; k++){
        if (out[k] == FALSE) {   // needs to be output
             cout << endl << “A new class: “ << k;   out[k] = TRUE;
             ListNode *x = seq[k];  ListNode *top = 0;  // init stack
             while (1) {  // find rest of class
                 while (x) {  // process the list
                      j = x→data;
                    if ( out[j] == FALSE ) {
                         cout << “, “ << j;      out[j] = TRUE;
                         ListNode  *y = x→link;
                         x→link = top;  top = x;    x = y;
                    } else  x = x→link;  // skip current node x
                 }
                 if ( ! top )  break;
                 else { x = seq[top→data];   top = top→link;  // unstack }
             } // end of while(1)
        }  // end of if (out[k] == FALSE)
}
for ( k=0; k<n; k++)
     while(seq[k]) { ListNode *delnode = seq[k];  seq[k] = delnode→link; delete delnode; }
     delete [] seq;  delete [] out;
}
```



m equivalence pairs
n nodes
→ O(m+n) algorithm

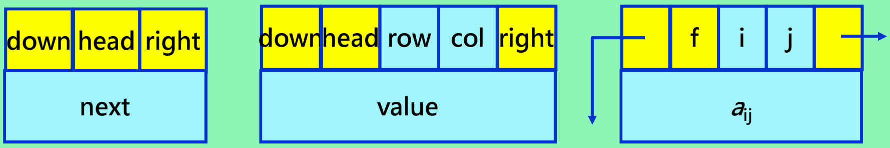
Outline

- Equivalence Class
- Sparse Matrices
- Doubly Linked Lists
- Generalized Lists
- Virtual Functions and Dynamic Binding

Sparse Matrix As A Two-Dimensional Linked List

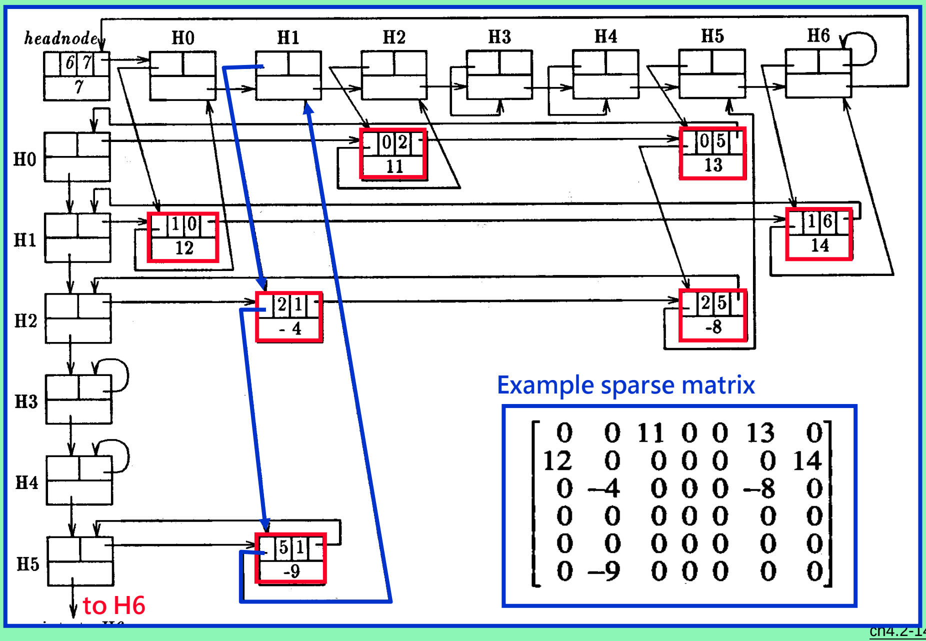
$$\begin{bmatrix} 0 & 0 & 11 & 0 & 0 & 13 & 0 \\ 12 & 0 & 0 & 0 & 0 & 0 & 14 \\ 0 & -4 & 0 & 0 & 0 & -8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -9 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

➡ 7 nonzero terms



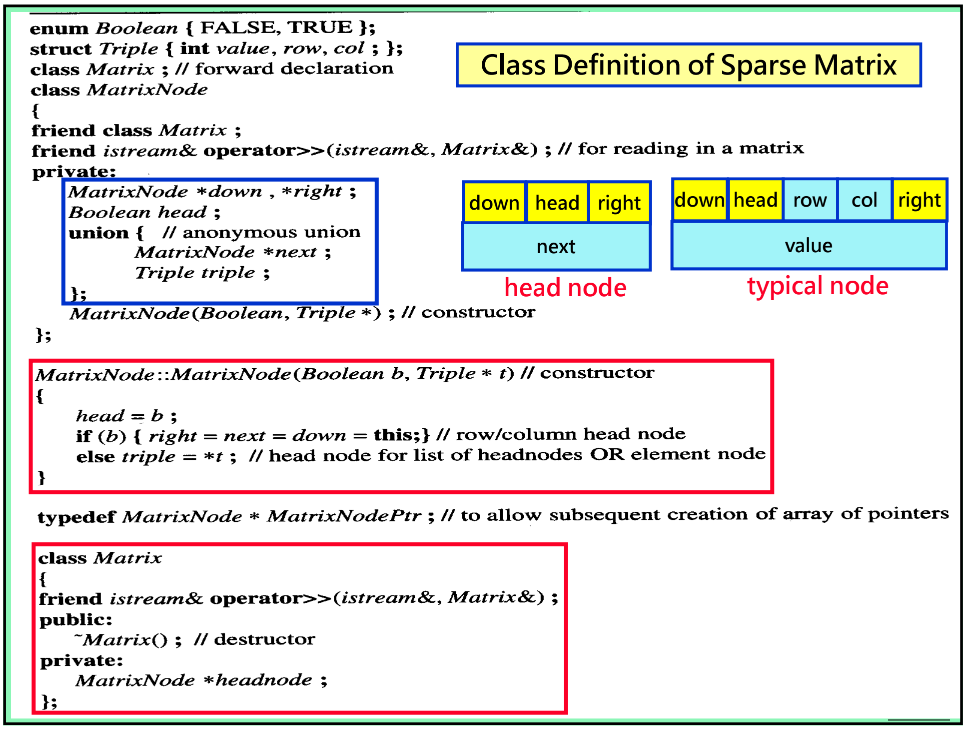
head node typical node setup for a_{ij}

Example: Sparse Matrix



Example sparse matrix

$$\begin{bmatrix} 0 & 0 & 11 & 0 & 0 & 13 & 0 \\ 12 & 0 & 0 & 0 & 0 & 0 & 14 \\ 0 & -4 & 0 & 0 & 0 & -8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -9 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$



Class Definition of Sparse Matrix

```
enum Boolean { FALSE, TRUE };
struct Triple { int value, row, col ; };
class Matrix ; // forward declaration
class MatrixNode
{
friend class Matrix ;
friend istream& operator>>(istream&, Matrix&) ; // for reading in a matrix
private:
    MatrixNode *down , *right ;
    Boolean head ;
    union {   // anonymous union
            MatrixNode *next ;
            Triple triple ;
    };
    MatrixNode(Boolean, Triple *) ; // constructor
};
```

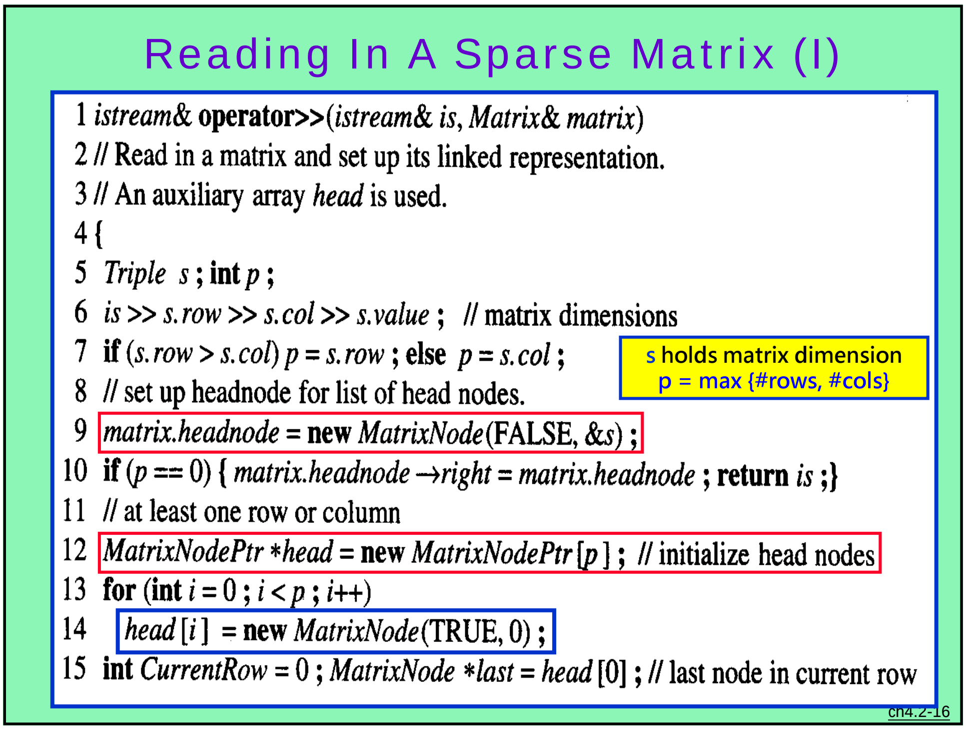
down	head	right
next		

head node

down	head	row	col	right
value				

typical node

```
MatrixNode::MatrixNode(Boolean b, Triple * t) // constructor
{
    head = b ;
    if (b) { right = next = down = this;} // row/column head node
    else triple = *t ;  // head node for list of headnodes OR element node
}

typedef MatrixNode * MatrixNodePtr ; // to allow subsequent creation of array of pointers

class Matrix
{
friend istream& operator>>(istream&, Matrix&) ;
public:
    ~Matrix() ;  // destructor
private:
    MatrixNode *headnode ;
};
```



Reading In A Sparse Matrix (I)

```
istream& operator>>(istream& is, Matrix& matrix)
// Read in a matrix and set up its linked representation.
// An auxiliary array head is used.
{
 Triple  s ; int p ;
 is >> s.row >> s.col >> s.value ;  // matrix dimensions
 if (s.row > s.col) p = s.row ; else  p = s.col ;
 // set up headnode for list of head nodes.
 matrix.headnode = new MatrixNode(FALSE, &s) ;
 if (p == 0) { matrix.headnode →right = matrix.headnode ; return is ;}
 // at least one row or column
 MatrixNodePtr *head = new MatrixNodePtr [p] ;  // initialize head nodes
 for (int i = 0 ; i < p ; i++)
   head [i]  = new MatrixNode(TRUE, 0) ;
 int CurrentRow = 0 ; MatrixNode *last = head [0] ; // last node in current row
```

s holds matrix dimension
p = max {#rows, #cols}

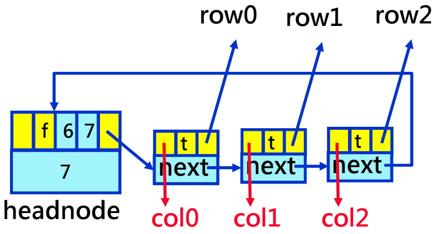
Reading In A Sparse Matrix (II)

```
16  for (i = 0 ; i < s.value ; i++)  // input triples
17  {
18     Triple t ;
19     is >> t.row >> t.col >> t.value ;
20     if (t.row > CurrentRow) {  // close current row
21        last →right = head [CurrentRow ] ;
22        CurrentRow = t.row ;
23        last = head [CurrentRow ] ;
24     } // end of if
25     last = last →right = new MatrixNode(FALSE, &t) ;  // link new node into row list
26     head [t.col]→next = head [t.col]→next →down = last ; // link into column list
27  } // end of for
28  last →right = head [CurrentRow ] ; // close last row
29  for (i = 0 ; i < s.col ; i++) head [i ]→next →down = head [i ] ; // close all column lists
30  // link the head nodes together
31  for (i = 0 ; i < p − 1 ; i++) head [i ]→next = head [i +1] ;
32  head [p −1]→next = matrix.headnode ;
33  matrix.headnode →right =  head [0] ;
34  delete [ ] head;
35  return is ;
36 }
```

Trick: head[i]→next is used initially to keep track of the last node in column i. But eventually, the head nodes are linked together through *next* (in line 30).

Erasing a Sparse Matrix

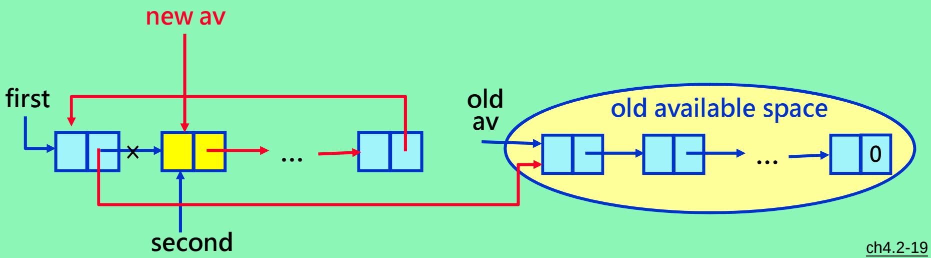
```
Matrix::~Matrix ()
// Return all nodes to the av list. This list is a chain linked via the right
// field. av is a global variable of type MatrixNode * and points to its first node.
{
   if (!headnode) return; // no nodes to dispose
   MatrixNode *x = headnode →right , *y ;
   headnode →right = av ; av = headnode ; // return headnode
   while (x != headnode) { // erase by rows
         y = x →right ;
         x →right = av ;
         av = y ;
         x = x →next ; // next row
   }
   headnode = 0 ;
}
```



Erasing A Circular List

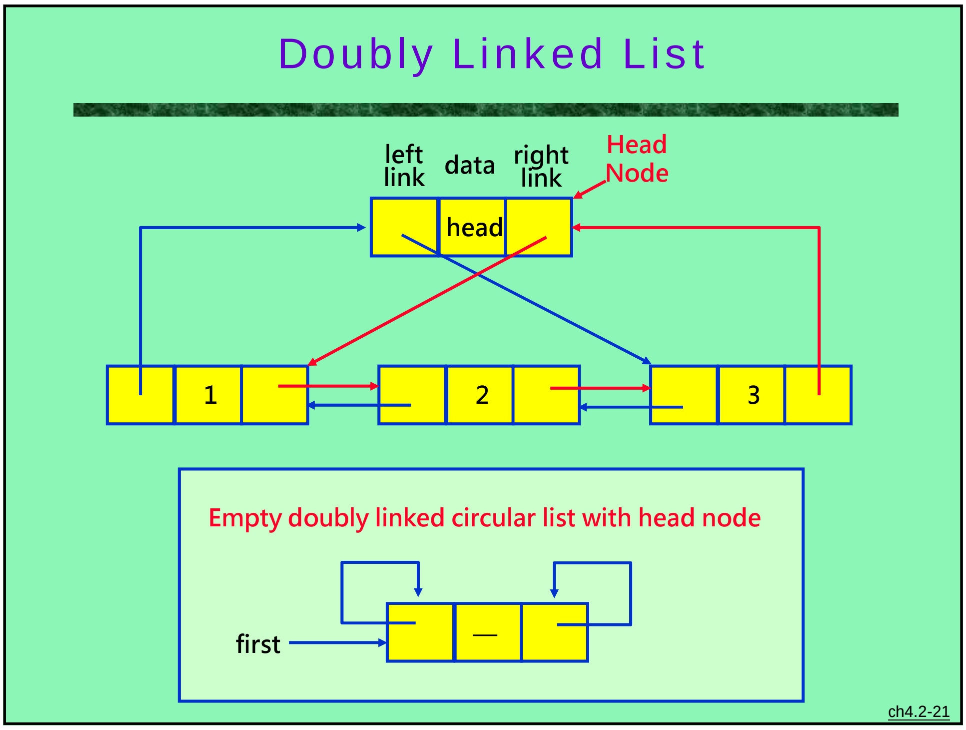
```
template <class Type>
void CircularList<Type>::~CircularList()
// Erase the entire circular list pointed by first
{
   if ( first ) {
        ListNode*  second = first → link; // second node
        first → link = av;    // first node linked to av
        av = second;   // second node of list becomes front of av list
        first = 0;
   }
}
```

A Circular List can be erased in a fixed amount of time
→ Independent of the number of nodes in the list



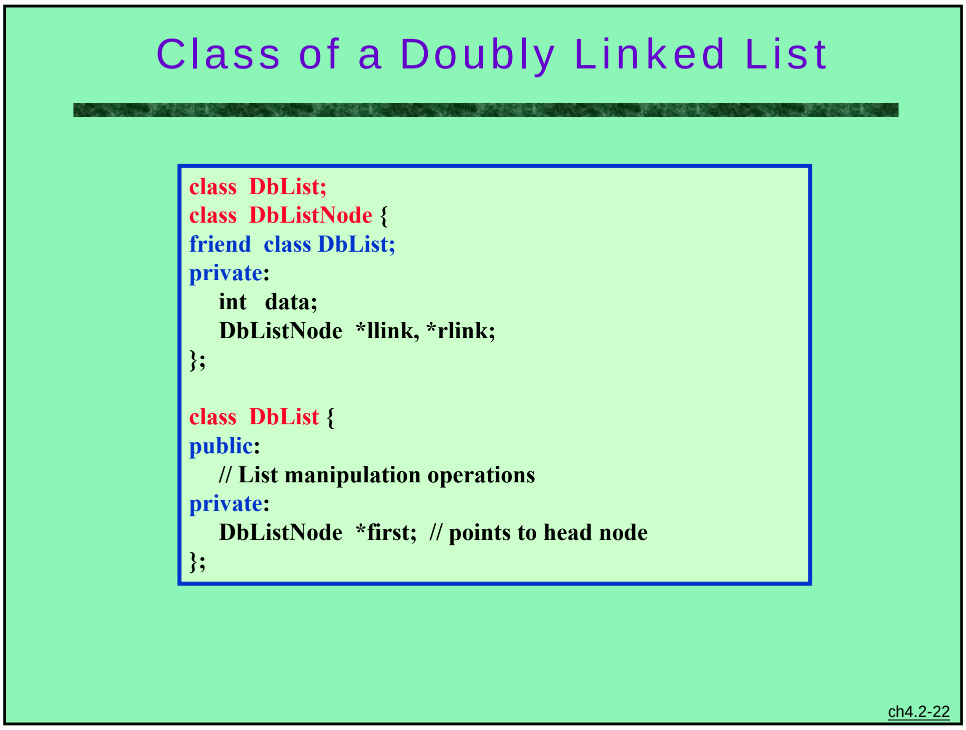
Outline

- Equivalence Class
- Sparse Matrices
- **Doubly Linked Lists**
- Generalized Lists
- Virtual Functions and Dynamic Binding



Doubly Linked List

Empty doubly linked circular list with head node

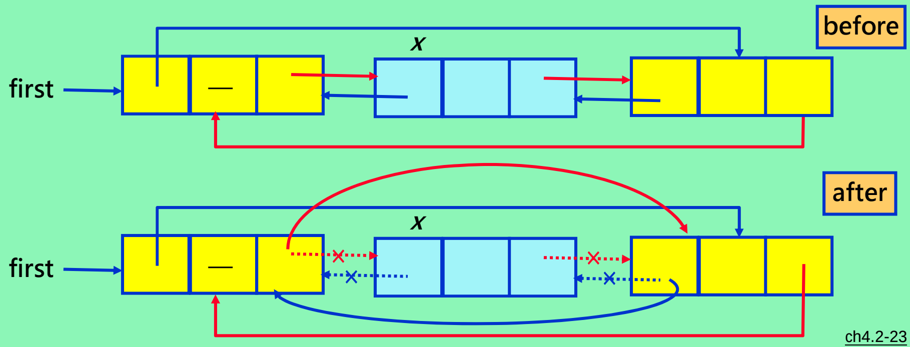


Class of a Doubly Linked List

```
class  DbList;
class  DbListNode {
friend  class DbList;
private:
    int   data;
    DbListNode  *llink, *rlink;
};

class  DbList {
public:
    // List manipulation operations
private:
    DbListNode  *first;  // points to head node
};
```

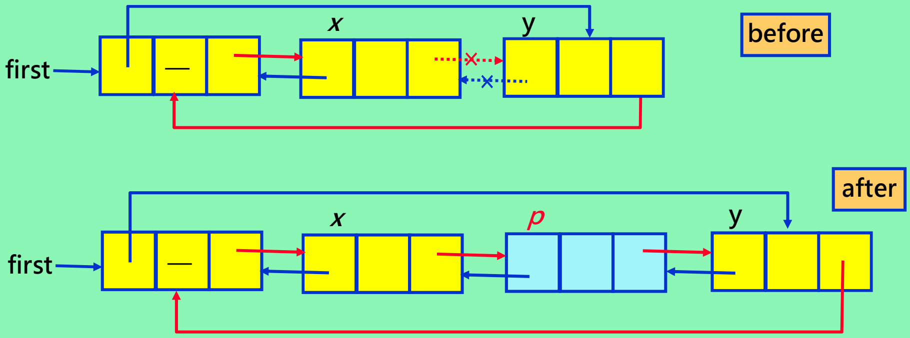
Deletion From a Doubly Linked List

```
void  DbList::Delete ( DbListNode *x)
{
   if (x == first)  cerr << “Deletion of head node not permitted” << endl;
   else {
         x→llink→rlink = x→rlink;
         x→rlink→llink = x→llink;
         delete x;
    }
}
```



Insertion To a Doubly Linked List

```
void  DbList::Insert ( DbListNode *p, DbListNode *x)
// Insert node p to the right of node x
{
    p→llink = x;  p→rlink = x→rlink;
    x→rlink→llink = p;  x→rlink = p;
}
```



Outline

- **Equivalence Class**
- **Sparse Matrices**
- **Doubly Linked Lists**
- **Generalized Lists**
- **Virtual Functions and Dynamic Binding**

Generalized Lists

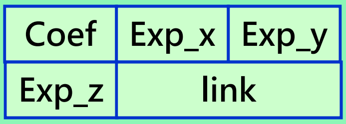
- **Definition**
  - **A generalized list, A, is a finite sequence of $n \geqq 0$ elements, $(\alpha_0, \ldots, \alpha_{n-1})$ where α_i is either an atom or a list.**
- **Head**
  - **α_0 is called the head of A**
- **Tail**
  - **$(\alpha_1, \ldots, \alpha_{n-1})$ is called the tail of A**
- **This is a recursive definition**
  - **E.g., C=(*a*, C)=(*a*, (*a*, (*a*, …))), A=(*a*, (*b*, *c*)), B=(A, A, ())**
  - **A compact way of describing a large and varied structure**

Polynomial With Multiple Variables

- **Example**

  $P(x, y, z) = x^{10}y^3z^2 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^4y^4z + 6x^3y^4z + 2yz$

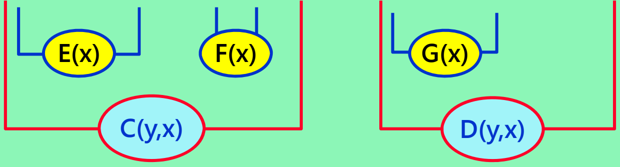
- **Sequential Representation**
  - Use a structure with four fields to represent a single array element
    - Coef, Exp_x, Exp_y, Exp_z



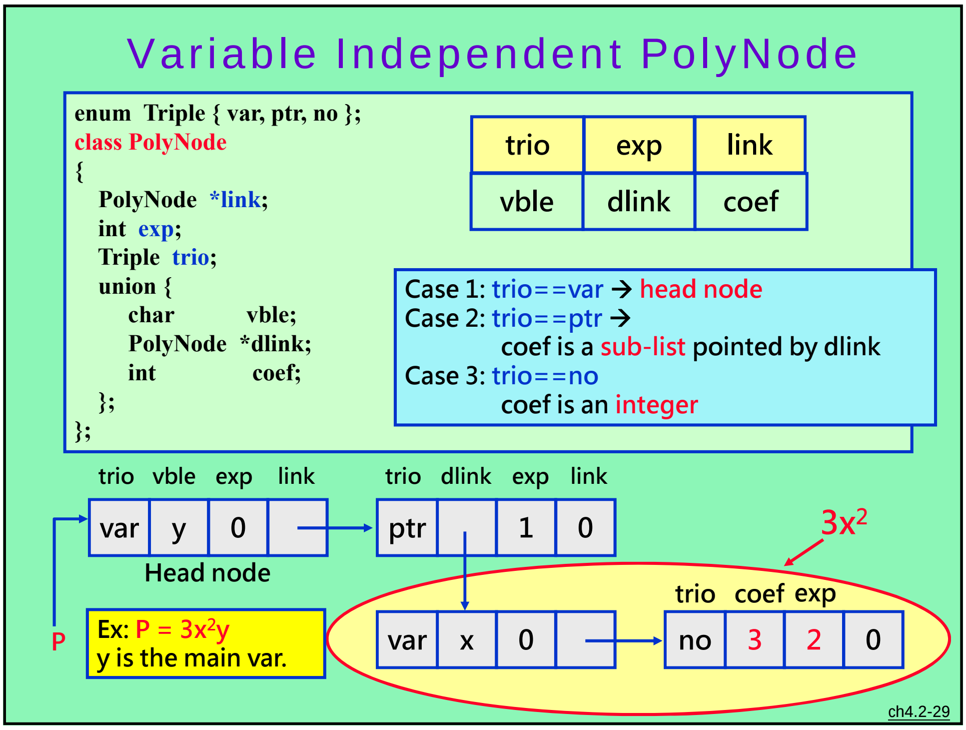
A polynomial term

Factored Form of Polynomial

- **Variable order**
  - { z, y, x }
  - z is the main variable, y is the second, x is the third
- **Factored Expression**
  - $((x^{10} + 2x^8) y^3 + 3x^8 y^2) z^2 + ((x^4 + 6x^3) y^4 + 2y) z$



$$\boxed{P(x, y, z) = x^{10}y^3z^2 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^4y^4z + 6x^3y^4z + 2yz}$$



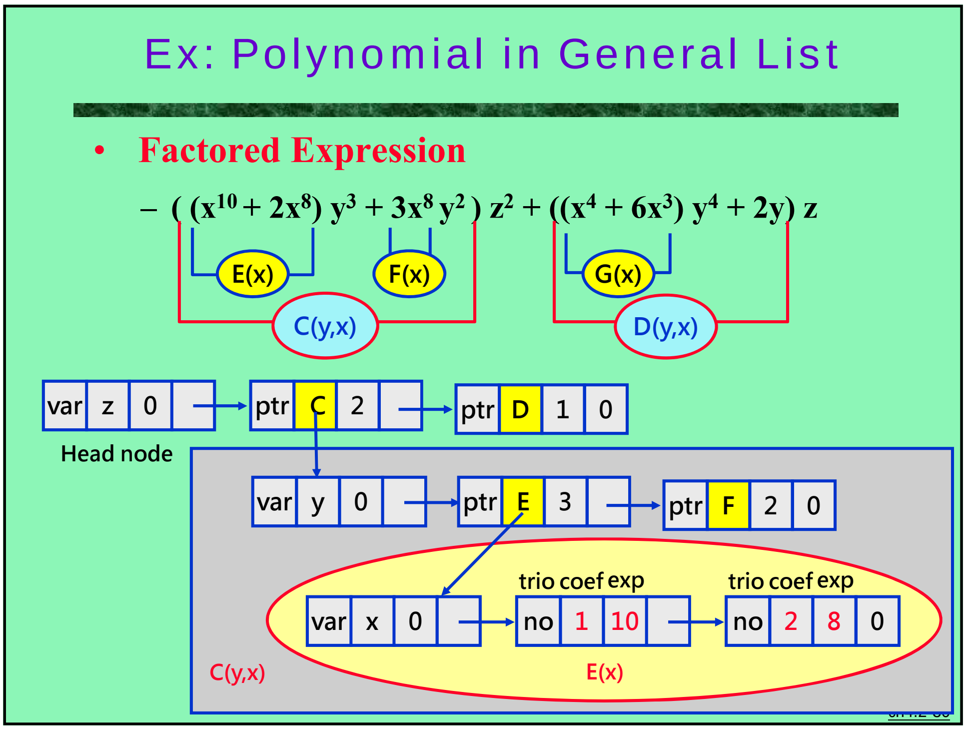
Variable Independent PolyNode

```
enum  Triple { var, ptr, no };
class PolyNode
{
    PolyNode  *link;
    int  exp;
    Triple  trio;
    union {
        char           vble;
        PolyNode  *dlink;
        int             coef;
    };
};
```

trio	exp	link
vble	dlink	coef

Case 1: trio==var → head node
Case 2: trio==ptr →
coef is a sub-list pointed by dlink
Case 3: trio==no
coef is an integer

Ex: P = $3x^2y$
y is the main var.



Ex: Polynomial in General List

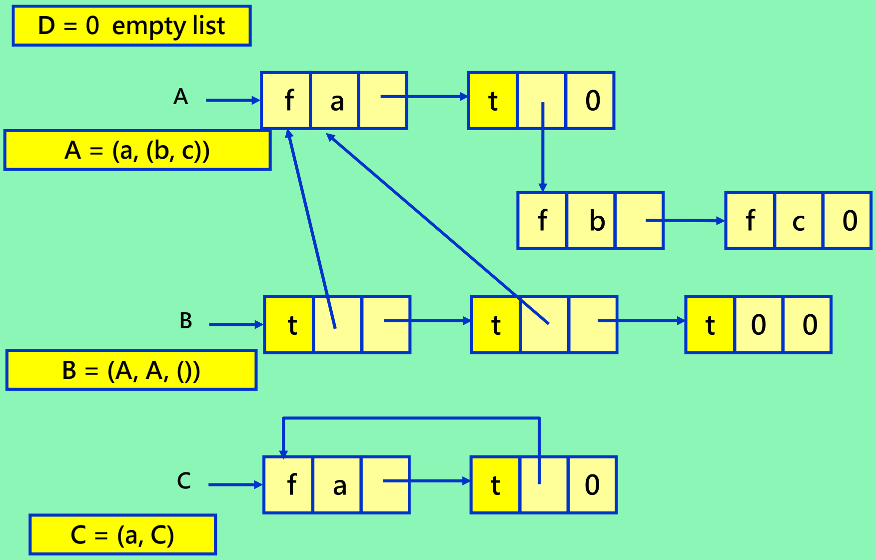
- **Factored Expression**
  - $((x^{10}+2x^8)\,y^3+3x^8\,y^2)\,z^2+((x^4+6x^3)\,y^4+2y)\,z$

General List Class

```
enum Boolean { FALSE, TRUE};
class GenList; // forward declaration
class GenListNode {
friend class GenList;
private:
    GenListNode  *link;
     Boolean tag;   // for indication of an atom or a list
     union {
         char  data;
         GenListNode  *dlink;
     };
};
class GenList {
public:
   // List manipulation operations
private:
        GenListNode  *first;
};
```

tag = FALSE/TRUE	data / dlink	link

Example: General Lists



D = 0 empty list

A = (a, (b, c))

B = (A, A, ())

C = (a, C)

Recursive Algorithms

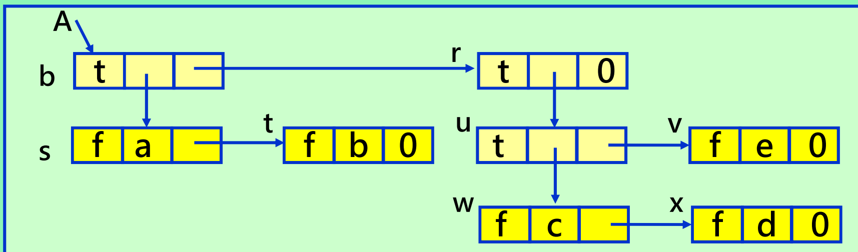
- **For recursively defined data object**
  - **It is often easy to describe algorithms that work on these objects recursively**
- **Two components in a recursive operation**
  - **(1) workhorse: the recursive function itself**
    - **Often declared as a private function**
  - **(2) driver: the function that invokes the recursive function at the top level**
    - **Declared as a public function**

Copying A General List

```
// Driver
void  GenList::Copy(const GenList& l)
{
    first = Copy(l.first);
}

// Workhorse
GenListNode *GenList::Copy(GenListNode *p)
// Copy the recursive list with no shared sublists pointed at by p
{
   GenListNode  *q = 0;
   if(p) {
      q = new GenListNode;  // q is the copied node
      q→tag = p→tag;
      if ( ! p→tag )  q→data = p→data;     // p is an atom node
      else   q→dlink = Copy ( p→dlink );  // p is a list pointer
      q→link = Copy( p→ link);
   }
   return q;
}
```

Proof: by induction
Complexity: O(m), or 3m steps
Recursion depth: m

Example: General List Copy

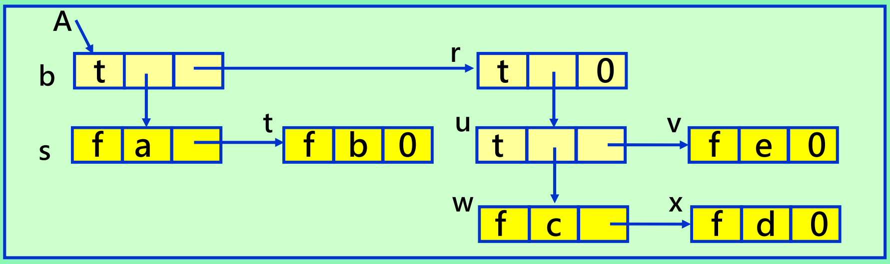


level of recursion	value of *p*	continuing level	*p*	continuing level	*p*
1	b	2	r	3	u
2	s	3	u	4	v
3	t	4	w	5	0
4	0	5	x	4	v
3	t	6	0	3	u
2	s	5	x	2	r
1	b	4	w	3	0
				2	r
				1	b

List Equality

```
// Driver – assumed to be a friend of GenList
int  operator==(const GenList& l, const GenList& m)
// l and m are non-recursive lists
// The function returns 1 if the two lists are identical and 0, otherwise
{
    return ( equal (l.first, m.first));
}
// Workhorse – assumed to be a friend of GenListNode
Int  equal ( GenListNode *s, GenListNode *t)
{
    int   x;
    if ( !s && !t ) return 1;      // both lists are null
    if ( s && t && (s→tag == t→tag) )
    {
         if ( ! (s→tag) )  // atom node
            if ( s→data == t→data )  x=1;  else x=0;
        else   x=equal (s→dlink, t→dlink);  // recursive call when list node
        if(x)   return ( equal ( s→link, t→link) ); else return 0;
     }
    return 0;  // only one list is null
}
```

Example: Depth of General List



A has two sub-lists: b→dlink and r→dlink
Depth of list pointed by b→dlink: 1
Depth of list pointed by r→dlink: 2
→ Depth(A) = max(Depth(b), Depth(r)) + 1 = 3

List Depth Computation

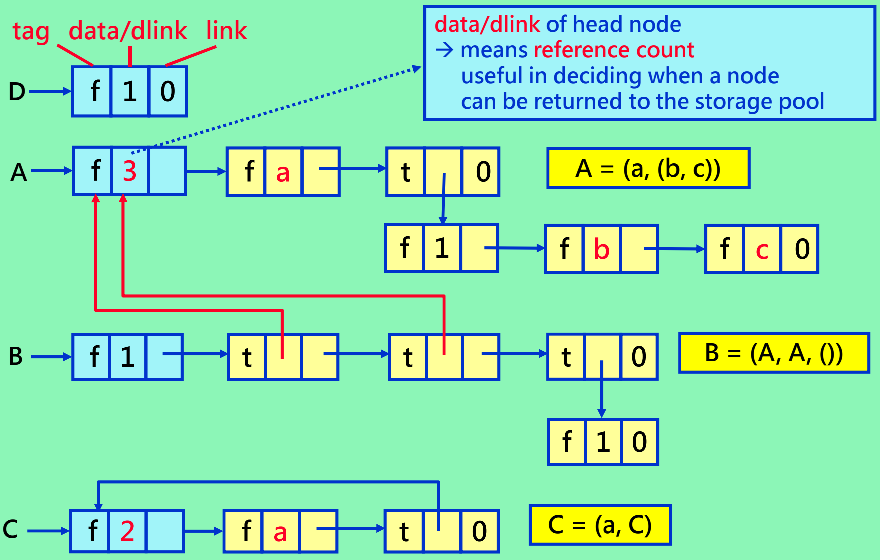
```
// Driver
int  GenList::depth()
// Compute the depth of a non-recursive list
{
    return (depth (first) );
}
// Workhorse
int  GenList::depth(GenListNode *s)
{
   if (!s) return 0;
   GenListNode *p = s; int  m=0;
   while (p) {
        if (p→tag) {   // sublist node
             int  n = depth (p→dlink);
             if (m < n)  m = n;
        }
        p = p→link;  // move forward
    }
    return  m+1;
}
```

depth(s) =

1 if s is an atom

$1 + \max\{ \text{depth}(x_1), \ldots, \text{depth}(x_n) \}$ if s is the list $(x_1, \ldots, x_n)$, $n \geq 1$

General Lists With Sharing



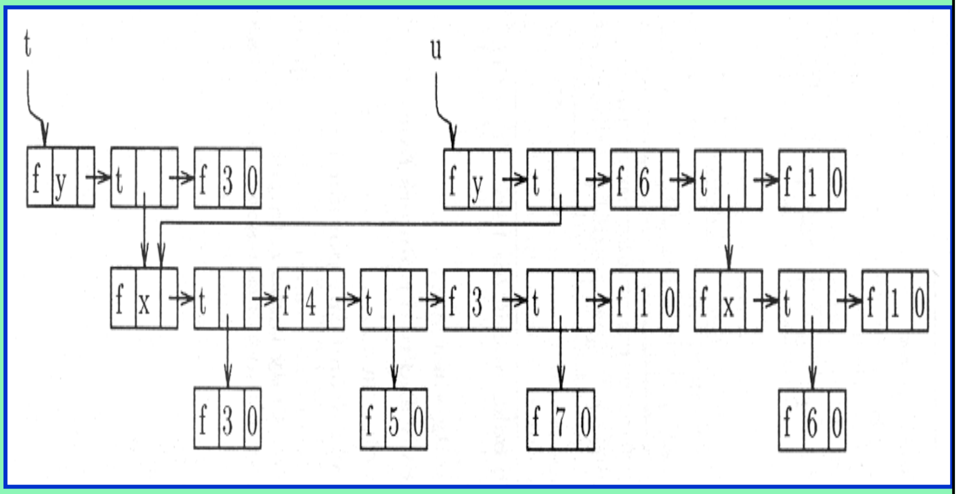
data/dlink of head node
→ means reference count
useful in deciding when a node
can be returned to the storage pool

A = (a, (b, c))

B = (A, A, ())

C = (a, C)

Polynomials With Sharing



$t = (3x^4 + 5x^3 + 7x) y^3$

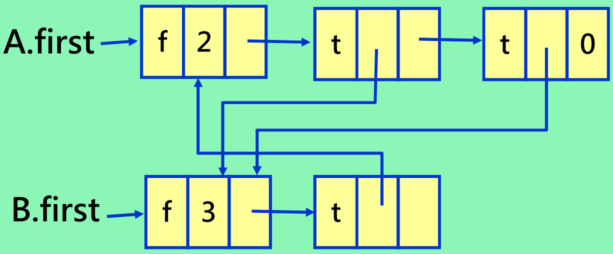
$u = (3x^4 + 5x^3 + 7x) y^6 + (6x)y$

Erasing A List Recursively

```
// Driver
int  GenList::~GenList()
// Each head node has a reference count. We assume first ≠0
{
    Delete ( first );     first = 0;
}
// Workhorse
int  GenList::Delete ( GenListNode *x)
{
   x → ref--;  // decrement the reference count of head node
   if ( ! x → ref )
   {
      GenListNode  *y = x;  // y traverses top-level of x
       while ( y → link )  {
             y = y → link;
             if ( y → tag == 1 )    Delete ( y → dlink ); }
             y → link = av;    // attach top-level nodes to av list
            av = y;
       }
   }
}
```

Indirect Recursion Case

- **For recursive list such as C = (a, C)**
  - **The reference count will never be 1**
  - **So, they cannot be recycled**
- **Indirect recursive lists cannot be recycled either**



A = (B, B)
B = (A)

Outline

- Equivalence Class
- Sparse Matrices
- Doubly Linked Lists
- Generalized Lists
- **Virtual Functions and Dynamic Binding**

Dynamic Binding

- **Public Inheritance**
  - Rectangle **IS-A** Polygon
  - Rectangle has all **attributes** of Polygon
  - Pointer to **a derived** class is implicitly converted to a pointer to its **bass class**
- **For example**
  - Rectangle r; // instance of derived class
  - Polygon *s = &r; // assign rectangle to polygon
- **Member function types**
  - Virtual functions
  - Non-virtual functions
  - Pure virtual functions
    - The responsibility of the implementation is passed on to the derived class

Example: Inheritance

```
class Polygon
{
public:
    int  GetId();  // non-virtual member function
    virtual  Boolean  Concave();
    virtual  int  Perimeter() = 0;  // pure virtual function
protected:
    int   id;
};

class Rectangle : public Polygon  // Rectangle publicly inherits from Polygon
{
public:
    Boolean  Concave();  // redefined in Rectangle
    int  Perimeter();        // defined in Rectangle
    // GetId() and id are inherited from Polygon
    // They, respectively, become public and protected members of Rectangle
private:
    // additional data members required to specialize Rectangle
    int  x1, y1, h, w;
};
```

Example: Inheritance (con’t)

```
// GetId() must never be redefined in a derived class
int  Polygon::GetId(){ return id; }

// Default implementation of Concave() in Polygon. A polygon is concave
// if it is possible to construct a line joining two points in the polygon
// that does not entirely lie within the polygon
Boolean  Polygon::Concave() {  return TRUE; }

// Rectangle must define Perimeter() because it is a pure virtual function
int  Rectangle::Perimeter() {  return 2*(h+w); }

// The default implementation of Concave() does not apply to rectangles
// So, it has to be redefined
Boolean Rectangle::Concave(){ return FALSE; }
```

The End of Linked Lists

Next Topic:

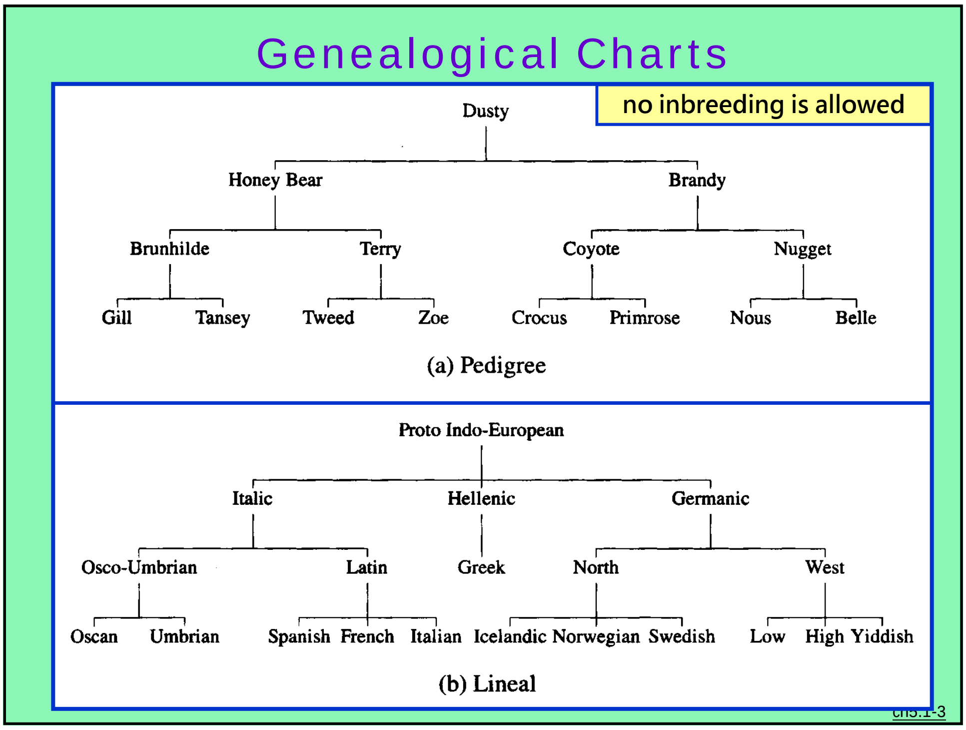
Trees

國立清華大學 電機工程學系
EE2410 Data Structure

Chapter 5
Trees (Part I)

Outline

- **Introduction**
- **Binary Trees**
- **Binary Tree Traversal**
- **Additional Binary Tree Operations**
- **Threaded Binary Trees**
- **Heaps**

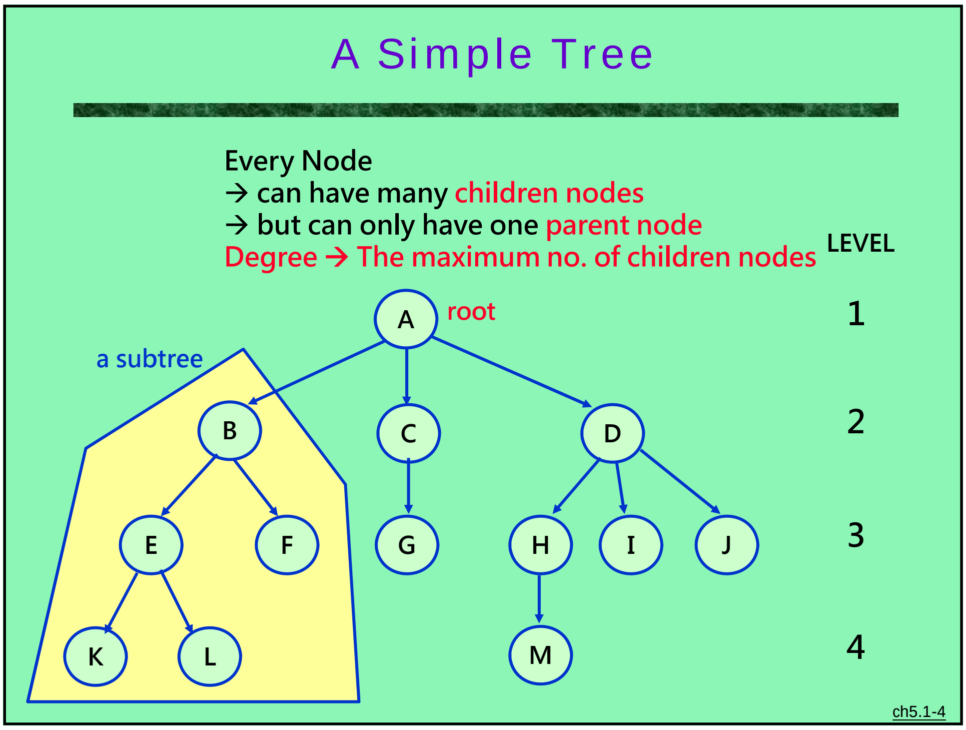


Genealogical Charts

no inbreeding is allowed

(a) Pedigree

(b) Lineal



A Simple Tree

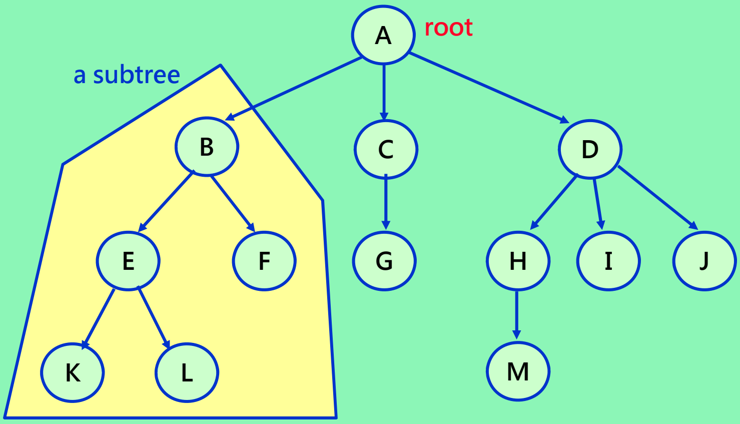
Every Node

→ can have many children nodes

→ but can only have one parent node

Degree → The maximum no. of children nodes

Direct Representation of a Tree



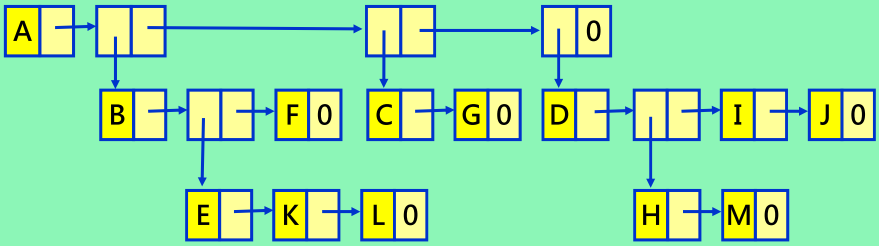
Possible node structure for a tree of degree k

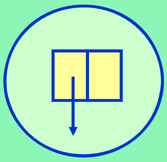
Data	Child1	Child2	…	Child k

Wasteful !

List Representation of A Tree

- **Representing a Tree as a List**
  - **(A (B (E(K,L), F), C(G), D(H(M), I, J)))**





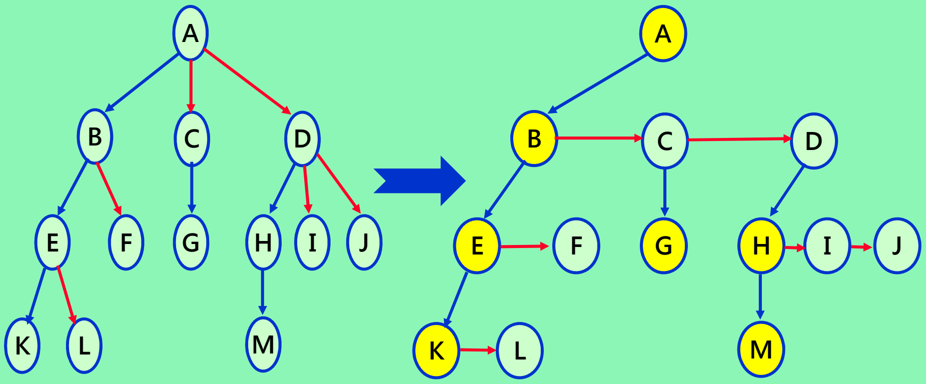
This type of node indicates a sub-tree

Lemma 5.1

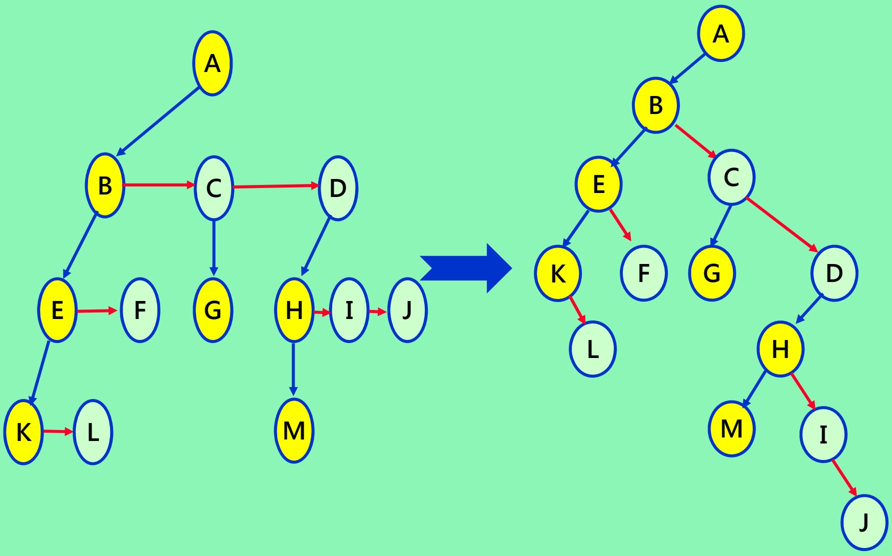
- **Lemma**
  - If T is a k-ary tree (I.e., a tree of degree k) with n nodes, each having a fixed size as shown in previous slide, then $n(k-1) + 1$ of the nk child fields are 0, $n \geqq 1$
- **Proof:**
  - The number of non-0 child fields in an n-node tree is exactly n-1
  - The total number of child fields in a k-ary tree with n nodes is nk
  - Hence, the number of 0 fields is $nk-(n-1) = n(k-1)+1$

Left Child-Right Sibling 嫡長子-庶子 Representation

Data	
left child	right sibling

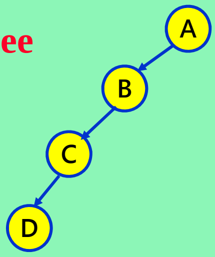


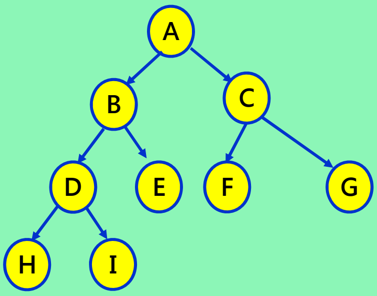
Degree-Two Tree



Binary Tree

- **Definition**
  - **A binary tree is a finite set of nodes that either is empty or consists of a root and two disjoint binary trees called left subtree and right subtree**
- **Skewed Tree**
- **Complete Tree**





ADT of BinaryTree

```
template <class KeyType>
class BinaryTree
{
// objects: A finite set of nodes either empty or consisting of a root node,
// left BinaryTree and right BinaryTree
public:
   BinaryTree();  // creates an empty binary tree

   Boolean IsEmpty();
   // if the binary tree is empty, return TRUE (1); else return FALSE (0)
   BinaryTree( BinaryTree bt1, Element<KeyType> item, BinaryTree bt2);
   // creates a binary tree whose left subtree is bt1, whose right subtree is bt2
   // and whose root node contains item
   BinaryTree Lchild();
   // if IsEmpty(), return error; else return the left subtree of *this
   Element<KeyType> Data();
   // if IsEmpty(), return error; else return the data in the root node of *this
   BinaryTree Rchild();
  // if IsEmpty(), return error; else return the right subtree of *this
};
```

Maximum Number of Nodes

- **Lemma 5.2**
  - **(1) The maximum no. of nodes on level *i* of a binary tree is 2^{i-1}, $i \geqq 1$**
  - **(2) The maximum number of nodes in a binary tree of depth *k* is 2^k-1, $k \geqq 1$**
- **Proof by induction**
  - **Induction base: root is the only node on level i=0**
  - **Induction hypothesis: max. no. of nodes on level i-1 is 2^{i-2}**
  - **Induction step: max. no. of nodes on level i is 2^{i-1}**

$$\sum_{i=1}^{k} \text{(maximum no. of nodes on level i)} = \sum_{i=0}^{k-1} 2^i = 2^k-1$$

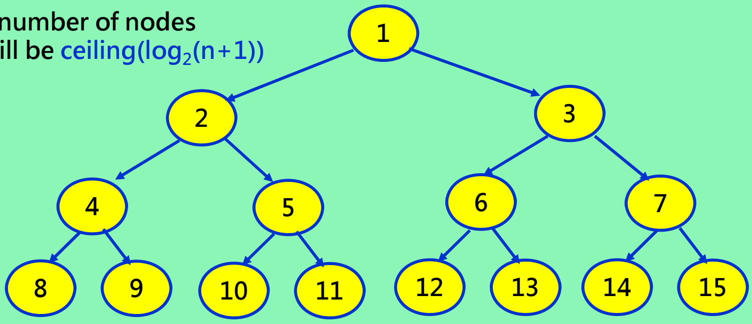
Leaf Nodes vs. Degree-2 Nodes

- **Lemma 5.3**
  - **For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2**

  → **then $n_0 = n_2 + 1$**
- **Proof**
  - **Let n be the total number of nodes**
  - **Let n_1 be the number of nodes of degree 1**
  - **We have $n = n_0 + n_1 + n_2$**
  - **If B is the number of branches, $n = B + 1$ and $B = n_1 + 2n_2$**
  - **Hence we obtain $n = B + 1 = n_1 + 2n_2 + 1$**
  - **Finally, we can reach $n_0 = n_2 + 1$**

Full Binary Tree With Sequential Node Number

Let n be the number of nodes
The depth will be ceiling($\log_2(n+1)$)

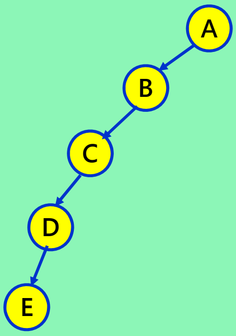


Lemma 5.4

If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \leqq i \leqq n$, we have

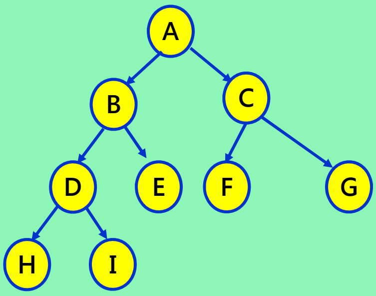
(1) parent(i) is at $\lfloor i/2 \rfloor$ if $i \neq 1$
(2) LeftChild(i) is at 2i if $2i \leqq n$. If $2i > n$, then i has no left child.
(3) RightChild(i) is at 2i+1 if $2i+1 \leqq n$. If $2i+1 > n$, then i has no right child

Array Representation of A Tree



Array

-
A
B
-
C
-
-
-
D
-
. .
E



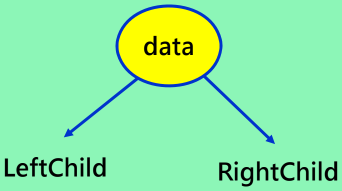
Array

-
A
B
C
D
E
F
G
H
I

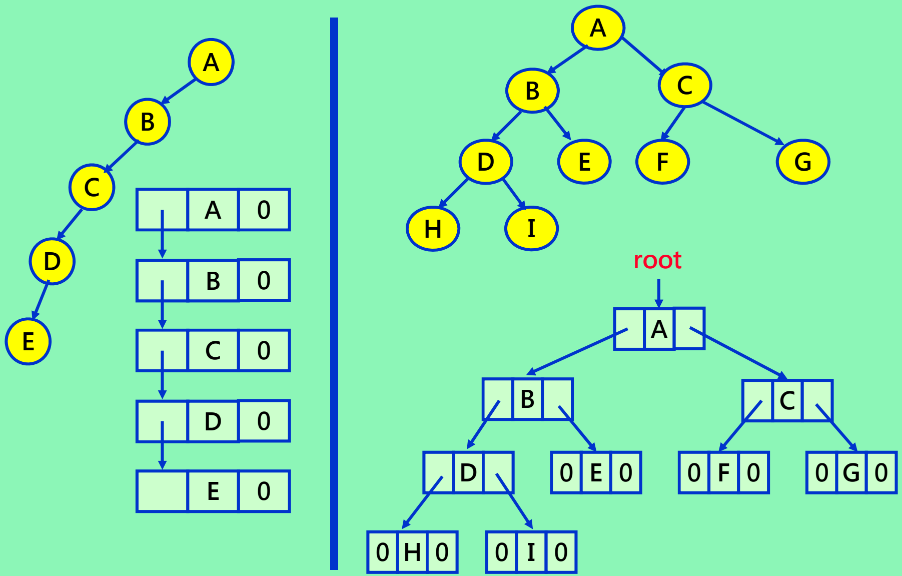
Linked Representation

```
class  Tree;  // forward declaration
class TreeNode {
friend class Tree;
private:
   TreeNode   *LeftChild;
   char  data;
   TreeNode   *RightChild;
};

class  Tree {
public:
   // Tree operations

private:
   TreeNode  *root;
};
```

LeftChild	data	RightChild



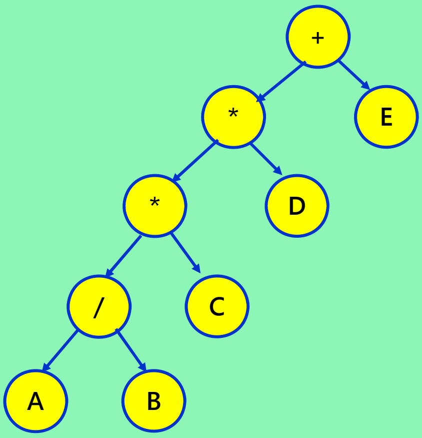
List Representation of A Tree



Outline

- **Introduction**
- **Binary Trees**
- **Binary Tree Traversal**
- **Additional Binary Tree Operations**
- **Threaded Binary Trees**
- **Heaps**

Binary Tree With Arithmetic Expression



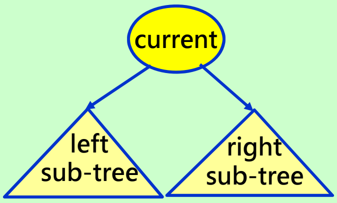
Inorder Traversal
A/B * C * D + E

Postorder Traversal
A B / C * D * E +

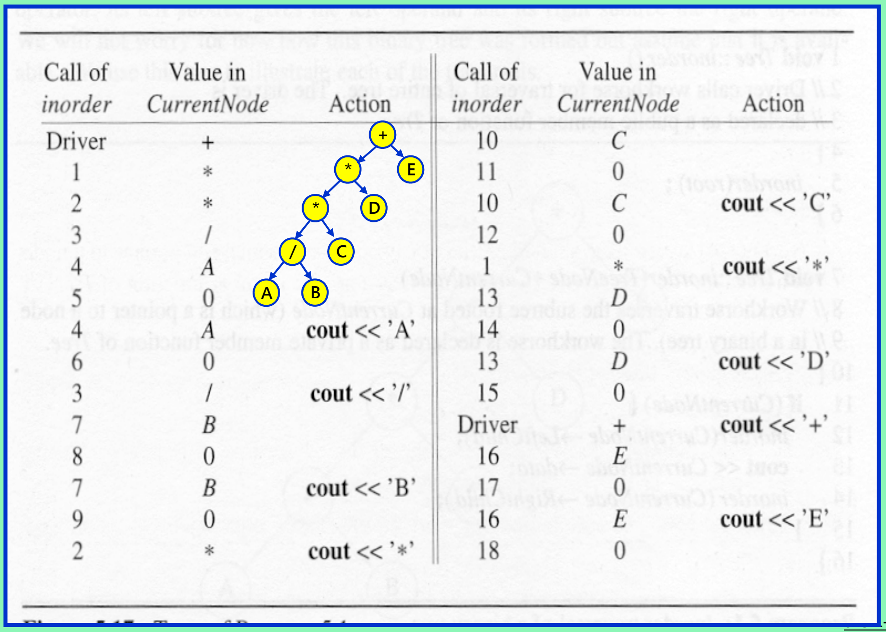
Preorder Traversal
+**/ A B C D E

Inorder Traversal of a Binary Tree

```
void  Tree::inorder()
// Driver calls workhorse for traversal of entire tree. The driver is declared
// as a public member function of Tree
{
   inorder(root);
}

void  Tree::inorder(TreeNode *CurrentNode)
// workhorse traverses the subtree rooted at CurrentNode (which is a pointer to
// a node in a binary tree). The workhorse is declared as a private member
// function of Tree
{
   if (CurrentNode) {
         inorder( CurrentNode → LeftChild);
         cout << CurrentNode → data);
         inorder ( CurrentNode → RightChild);
   }
}
```



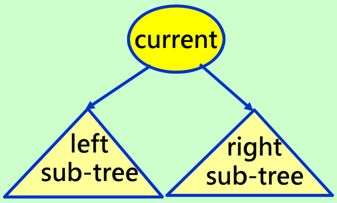
Trace of Inorder Traversal



Call of *inorder*	Value in *CurrentNode*	Action
Driver	+	
1	*	
2	*	
3	/	
4	*A*	
5	0	
4	*A*	**cout << 'A'**
6	0	
3	/	**cout << '/'**
7	*B*	
8	0	
7	*B*	**cout << 'B'**
9	0	
2	*	**cout << '*'**
10	*C*	
11	0	
10	*C*	**cout << 'C'**
12	0	
1	*	**cout << '*'**
13	*D*	
14	0	
13	*D*	**cout << 'D'**
15	0	
Driver	+	**cout << '+'**
16	*E*	
17	0	
16	*E*	**cout << 'E'**
18	0	

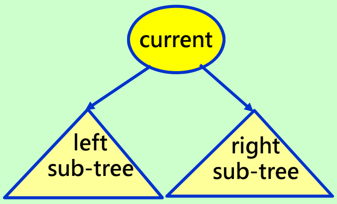
Preorder Traversal of a Binary Tree

```
void  Tree::preorder()
// Driver calls workhorse for traversal of entire tree. The driver is declared
// as a public member function of Tree
{
   preorder(root);
}

void  Tree::preorder(TreeNode *CurrentNode)
// workhorse traverses the subtree rooted at CurrentNode (which is a pointer to
// a node in a binary tree). The workhorse is declared as a private member
// function of Tree
{
   if (CurrentNode) {
         cout << CurrentNode → data);
         preorder( CurrentNode → LeftChild);
         preorder ( CurrentNode → RightChild);
   }
}
```

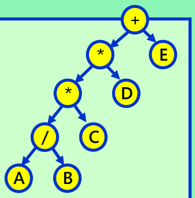


Postorder Traversal of a B-Tree

```
void  Tree::postorder()
// Driver calls workhorse for traversal of entire tree. The driver is declared
// as a public member function of Tree
{
   postorder(root);
}

void  Tree::postorder(TreeNode *CurrentNode)
// workhorse traverses the subtree rooted at CurrentNode (which is a pointer to
// a node in a binary tree). The workhorse is declared as a private member
// function of Tree
{
   if (CurrentNode) {
         postorder( CurrentNode → LeftChild);
         postorder ( CurrentNode → RightChild);
         cout << CurrentNode → data);
   }
}
```



Iterative Inorder Traversal

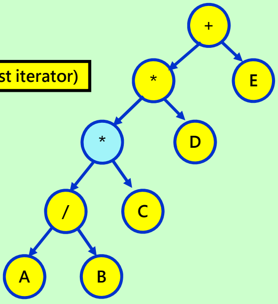


```
void  Tree::NonRecInorder()
// non-recursive inorder traversal using a stack
{
   Stack<TreeNode*> s;  // declare and initialize stack
   TreeNode  *CurrentNode = root;
   while(1) {
        while (CurrentNode) {          // move down the LeftChild fields all the way
                s.Add(CurrentNode);  // add to stack
                CurrentNode = CurrentNode→LeftChild;
        }
        if ( ! s.IsEmpty() ) {    // stack is not empty
            CurrentNode = *s.Delete ( CurrentNode );   // pop from stack
            cout << CurrentNode→data << endl;
            CurrentNode = CurrentNode → RightChild; // to explore Right SubTree
       }
       else  break;
   }
}
```

Inorder Traversal Using Iterator

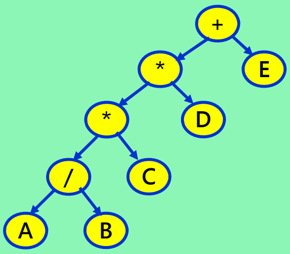
```
class  InorderIterator {
public:
   char  *Next();
   InorderIterator(Tree tree): t (tree) { CurrentNode = t.root; }
private:
   const  Tree& t;
   Stack<TreeNode*>  s;
   TreeNode  *CurrentNode;
};
char  *InorderIterator::Next()
{
   while (CurrentNode) {
          s.Add(CurrentNode);
          CurrentNode = CurrentNode → LeftChild;
   }
   if ( ! s.IsEmpty() ) {
          CurrentNode = *s.Delete( CurrentNode );
          char& temp = CurrentNode→data;
          CurrentNode = CurrentNode→RightChild;  // update CurrentNode
          return  &temp;
   }
   else  return(0);  // tree has been traversed, no more elements
}
```

New data member (not needed in a list iterator)



Level-Order Traversal

```
void  Tree::LevelOrder()
// Traverse the binary tree in level order
{
    Queue<TreeNode*>  q;    // A queue is needed here
    TreeNode  *CurrentNode  = root;
    while ( CurrentNode ) {
          cout << CurrentNode → data << endl;
          if ( CurrentNode → LeftChild ) q.Add(CurrentNode→LeftChild);
          if (CurrentNode→RightChild) q.Add(CurrentNode→RightChild);
          CurrentNode = *q.Delete(CurrentNode);
    }
}
```



Breadth-First Traversal (BFS)

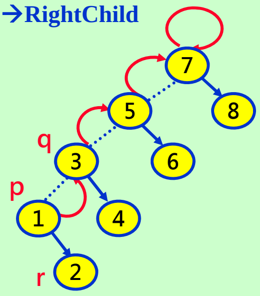
Level-order traversal:
+ * E * D / C A B

Three Ways of Tree Traversal Without Stacks

- **Add a parent field to each node**
  - **Doubly linked tree**
- **Use LeftChild and RightChild**
  - **To maintain the paths back to the root**
  - **shown in the next slide**
- **Threaded Binary Tree**
  - **To be introduced later**

Traversal Without A Stack

```
void  Tree::NoStackInorder()
// Inorder traversal of binary tree using a fixed amount of additional storage
{
   if ( ! root ) return;  // empty binary tree
   TreeNode *top = 0,  *LastRight = 0, *p, *q, *r, *r1;
   p = q = root;
   while (1) {
       while (1) {
           if ( ( ! p→LeftChild) && ( ! p→RightChild ) ) {   // leaf node
                cout << p → data;  break;
           }
           if ( ( ! p→LeftChid ) {  // visit p and move to p→RightChild
                 cout << p →data;
                 r = p → RightChild;  p →RightChild = q;
                 q = p;  p = r;
           }
           else {   // move to p→LeftChild
                 r = p → LeftChild;  p→LeftChild = q;
                 q = p;  p = r;
           }
      } // end of inner while
      TO BE CONTINUED …
```



```
while (1) {   …  previous page
    // p is a leaf node, move upward to a node whose right subtree not explored yet
    TreeNode  *av = p;
    while (1) {
        if (p==root)  return;
        if ( ! q→LeftChild) {  // q is linked via RightChild
            r = q→RightChild;   q→RightChild = p;   p = q;  q = r; }
        else if ( ! q→RightChild ) {  // q is linked via LeftChild
            r = q→LeftChild;  q→LeftChild = p;  p = q;  q = r;  cout << p→data;
        else {  // check if p is a RightChild of q
            if ( q == LastRight ) {
                r = top;  LastRight = r→LeftChild; top = r→RightChild;  // unstack
                r→LeftChild = r→RightChild = 0;
                r = q→RightChild;  q→RightChild = p;   p = q;  q = r;
            }
            else {   // p is LeftChild of q
                cout << q→data;  // visit q
                av→LeftChild = LastRight; av→RightChild = top;
                top = av;  LastRight = q;
                r = q→LeftChild;  q→LeftChild = p;  // restore link to p
                r1 = q→RightChild;  q→RightChild = r;   p = r1;   break;  }
        }
    }
}
```

Outline

- Introduction
- Binary Trees
- Binary Tree Traversal
- Additional Binary Tree Operations
- Threaded Binary Trees
- Heaps

Copying Binary Trees

```
// Copy constructor
Tree::Tree( const Tree& s) // driver
{
    root = copy(s.root);
}

TreeNode  *Tree::copy(TreeNode *orignode)  // Workhorse
// The function returns a pointer to an exact copy of the binary tree rooted
// at orginode
{
  if ( originode )  {
       ThreeNode  *temp = new TreeNode;
       temp→data = orignode → data;
       temp→LeftChild = copy(orignode→LeftChild);
       temp→RightChild = copy(orignode→RightChild);
       return  temp;
  }
  else   return 0;
}
```

Propositional Calculus

- **Boolean Formula**
  - is often constructed by
    - a set of variables { x_1, x_2, x_3, …}
    - operators such as * (AND) + (OR), and ~ (NOT)
  - holds the value of true or false
- **Construction Rules of Propositional Calculus**

  (1) A variable is an expression

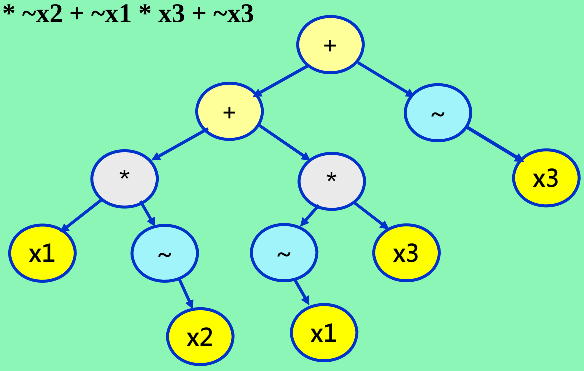
  (2) if x and y are expressions, then x*y, x+y, and ~x are expressions

  (3) Parentheses can be used to alter the normal order of evaluation

  Example: P = x1 + (x2 * ~ x3)
- **Evaluation**
  - when x1=false, x2=true, x3=false, then P = true

Satisfiability Problem

- **Satisfiability Problem**
  - **for formulas of propositional calculus asks if there is an assignment of values to the variables that causes the value of the expression to be true**
- **Example**
  - **P = x1 * ~x2 + ~x1 * x3 + ~x3**



First Version of Satisfiability Problem

```
enum Boolean { FALSE, TRUE };
enum TypesOfData { NOT, AND, OR,
        TRUE, FALSE }
class  SatTree; // forward declaration
class  SatNode {
friend  class  SatTree;
private:
     SatNode  *LeftChild;
     TypesOfData  data;
     Boolean  value;
     SatNode  *RightChild;
}
```

```
class  SatTree {
public:
     void  PostOrderEval();
     void  rootvalue() {
           cout << root→value;
     };
private:
     SatNode  *root;
     void   PostOrderEval ( SatNode *);
};
```

```
for all 2^n possible value combinations for the n variables
{
    generate the next combination;
    replace the variable by their values;
    evaluate the formula by traversing the tree by PostOrderEval();
    if ( formula.rootvalue()) { cout << combination; return; }
}
cout << “no satisfiable combination” ;
```

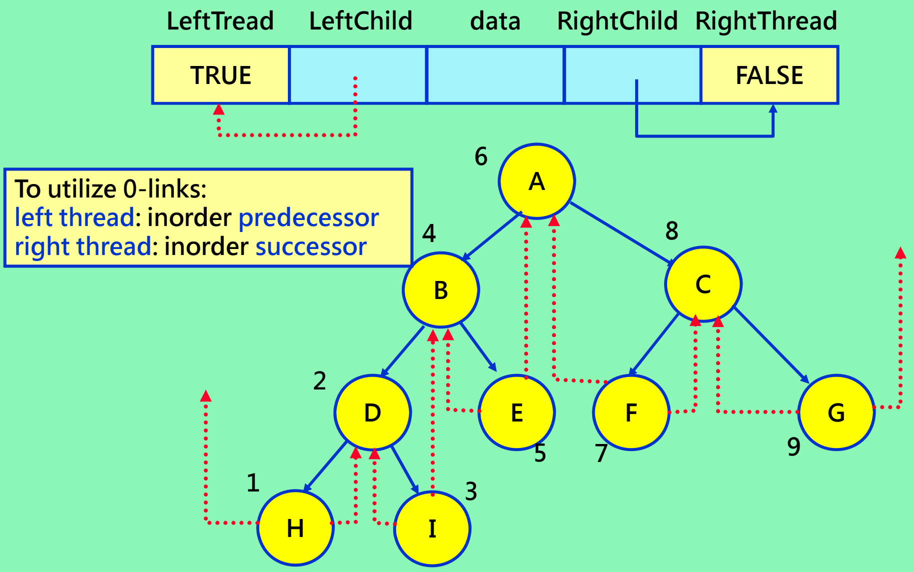
Evaluating A Formula

```
void  SatTree::PostOrderEval()  // Driver
{
    PostOrderEval ( root );
}

void  SatTree::PostOrderEval(SatNode *s)  // Workhorse
{
    if (s) {
        PostOrderEval ( s→LeftChild );
        PostOrderEval ( s→RightChild );
        switch ( s→data) {
            case  NOT:  s→value = ! s→RightChild→value;  break;
            case  AND:  s→value = s→LeftChild→value && s→RightChild→value;
                        break;
            case  OR:   s→value = s→LeftChild→value || s→RightChild→value;
                        break;
            case TRUE:  s→value = TRUE;  break;
            case FALSE: s→value = FALSE; break;
        }
     }
}
```

Outline

- **Introduction**
- **Binary Trees**
- **Binary Tree Traversal**
- **Additional Binary Tree Operations**
- **Threaded Binary Trees**
- **Heaps**

Threaded Tree

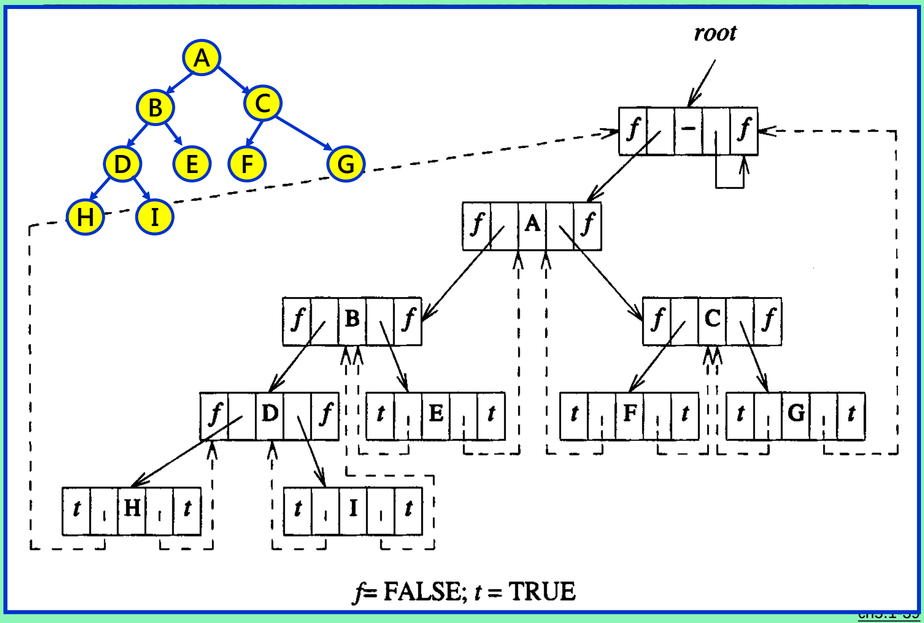


To utilize 0-links:
left thread: inorder predecessor
right thread: inorder successor

Class for Threaded Binary Tree

```
class  ThreadedNode {
friend  class  ThreadedTree;
friend  class  ThreadedInorderIterator;
private:
    Boolean  LeftThread;
    ThreadedNode  *LeftChild;
    char   data;
    ThreadedNode  *RightChild;
    Boolean RightThread;
};
class  ThreadedTree {
friend class  ThreadedInorderIterator;
public:  //  Tree manipulation operations follow
private:
   ThreadedNode  *root;
};
class  ThreadedInorderIterator {
public:
   char *next();
   ThreadedInorderIterator(ThreadedTree tree): t (tree) { CurrentNode = t.root; };
private:
   ThreadedTree  t;
   ThreadedNode  *CurrentNode;
}
```

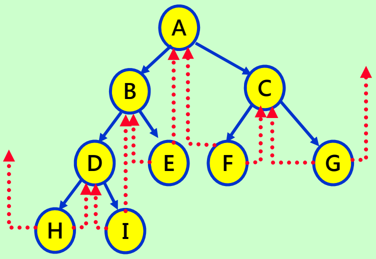
Memory Representation of Threaded Tree



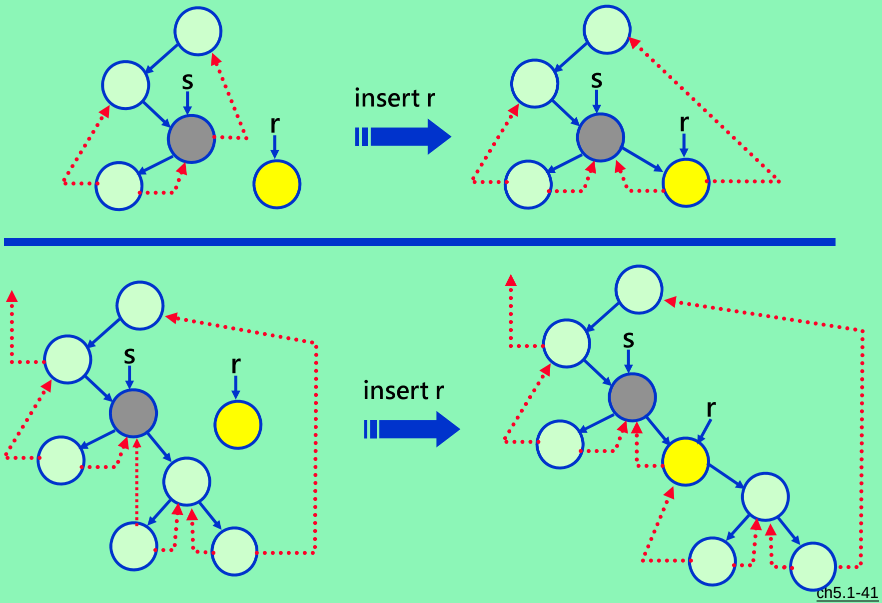
f= FALSE; *t* = TRUE

Finding the Inorder Successor

```
char  *ThreadedInorderIterator::Next()
//  Find the inorder successor of CurrentNode in a threaded binary tree
{
   ThreadedNode  *temp = CurrentNode→RightChild;  // rightchild or successor
   if ( ! CurrentNode→RightThread )
        while ( ! temp→LeftThread )  temp = temp→LeftChild;
   CurrentNode = temp;
   if ( CurrentNode == t.root )  return 0;    // last node has been reached
   else   return (&CurrentNode→data);
}

void  ThreadedInorderIterator::Inorder()
{
   for (char *ch = Next(); ch;  ch = Next() ) {
        cout << *ch << endl;
   }
}
```

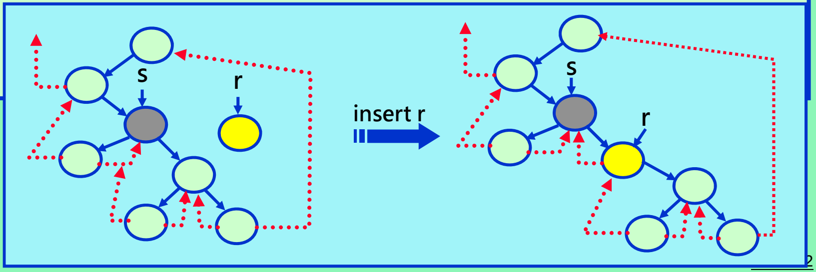


Inserting A Node In Threaded Tree



Inserting Operation

```
void  ThreadedTree::InsertRight( ThreadedNode *s, ThreadedNode *r)
// Insert r as the right child of s
{
   r→RightChild = s→RightChild;
   r→RightThread = s→RightThread;
   r→LeftChild = s;
   r→LeftThread = TRUE;  // LeftChild is a thread
   s→RightChild = r;  // Attach r to s
   s→RightThread = FALSE;
   if ( ! r→RightThread) {
        ThreadedNode *temp = InorderSucc(r) ; // returns the inorder successor of r
        temp→LeftChild = r;
   }
}
```



Outline

- Introduction
- Binary Trees
- Binary Tree Traversal
- Additional Binary Tree Operations
- Threaded Binary Trees
- ➡ **Heaps**

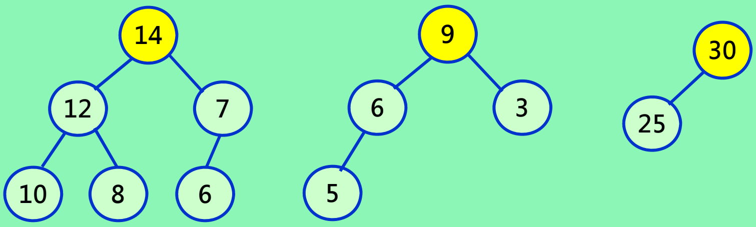
Priority Queue

- **Priority Queue**
  - Elements **deleted** is the one with the **highest priority**
  - An element with arbitrary priority may be inserted
  - This is called **max priority queue**
  - **min priority queue** is defined similarly

```
template<class Type>
class  MaxPQ {
public:
    virtual  void   insert(const  Element<Type>&) = 0;
    virtual  Element<Type>* DeleteMax( Element<Type>&) = 0;
};
```

Max Heap

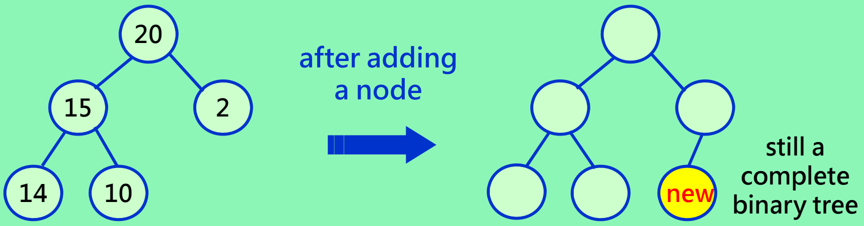
- **Heap**
  - is frequently used to implement a priority queue
- **Definition**
  - A max(min) tree is a tree in which the key value in each node is no smaller (larger) than the key values in its children (if any)
  - A max heap is a complete binary tree that is also a max tree
  - A min heap is a complete binary tree that is also a min tree

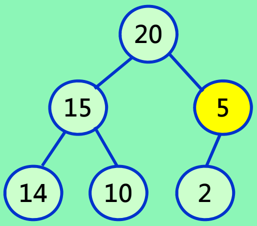


Class Definition of Max Heap

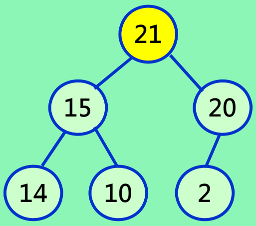
```
template <class KeyType>
class  MaxHeap : public MaxPQ<KeyType>
{
// objects: A complete binary tree of n > 0 elements organized in a way that
// the value in each node is at least as large as those in its children
public:
   MaxHeap( int  sz = DefaultSize);
   // Create an empty heap that can hold a maximum of sz elements
   Boolean  IsFull();
   // If the number of elements in the heap is equal to the maximum size of the
   // heap, return TRUE(1); otherwise, return FALSE(0)
   void  Insert(Element<KeyType> item);
   // If IsFull(), then error, else insert item into the heap
   Boolean  IsEmpty()
   // If number of elements in heap is 0, return TRUE(1); else return FALSE(0)
   Element<KeyType>* Delete(KeyType& x);
private:
   Element<Type>  *heap;
   int  n;    // current size of max heap
   int  MaxSize;  // Maximum allowable size of heap
}
```

Insertion To a Max Heap





Final tree after inserting 5

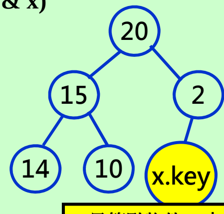


Final tree after inserting 21

Insertion To a Max Heap

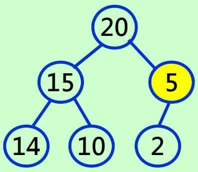
```
template <class Type>
void  MaxHeap<Type>::Insert( const Element<Type>& x)
// insert x into the max heap
{
   if (n==MaxSize) {  HeapFull();  return; }
   n++;
   for(int i=n; 1;) {
        if (i==1)  break;  // at root
        if (x.key <= heap[i/2].key) break;
        // move from parent to i
        heap[i] = heap[i/2];
        i = i /2;
   }
   heap[i] = x;   // write in the data
}
```

original

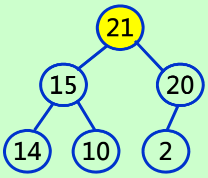


x 是節點物件 , x.key 是關鍵值

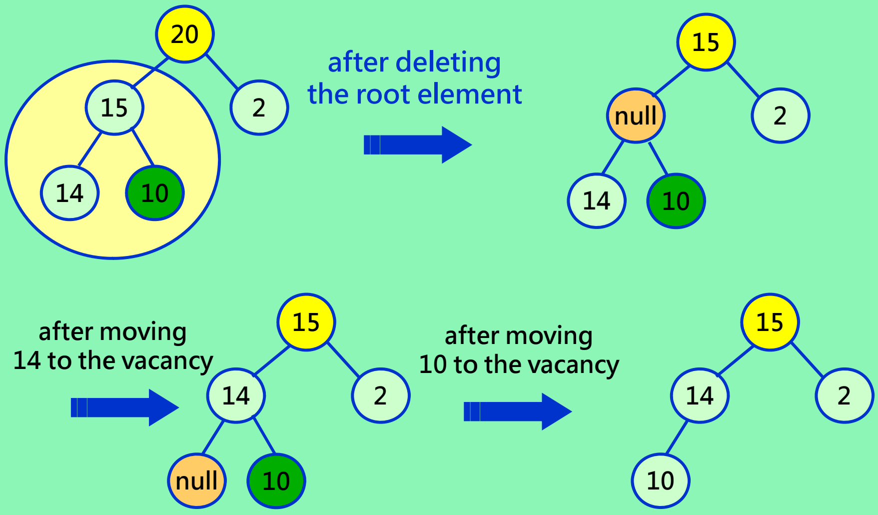
adding 5



adding 21



Deletion From a Max Heap

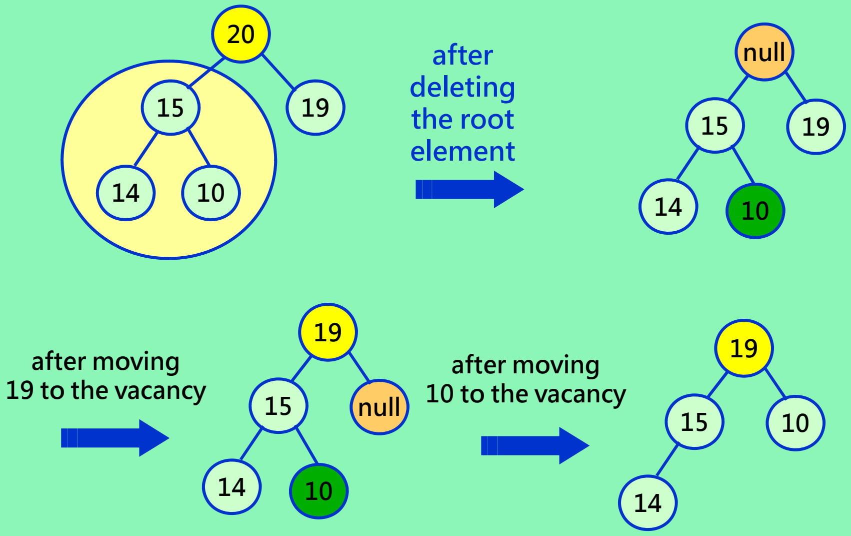

after deleting the root element

after moving 14 to the vacancy

after moving 10 to the vacancy

Final tree → still complete binary tree

Deletion From a Max Heap


after deleting the root element

after moving 19 to the vacancy

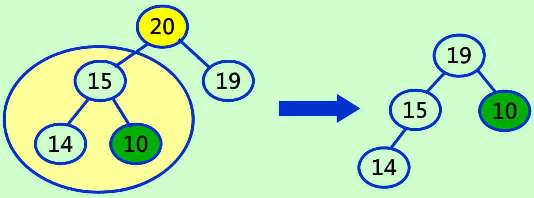
after moving 10 to the vacancy

Deletion From a Max Heap

```
template <class Type>
Element<Type>* MaxHeap<Type>::DeleteMax(Element<Type>& x)
// Delete from the max heap
{
   if (n==0) { HeapEmpty(); return 0; }
   x = heap[1]; Element<Type> k = heap[n]; n--;
   for ( int  i=1; j=2; j<=n;)
   {
      if (j<n)  if (heap[j].key < heap[j+1].key)  j++;
      // j points to the larger child
      if (k.key >= heap[j].key)  break;
      heap[i] = heap[j];   // move child up
      i = j;  j *= 2;  // move i and j down
   }
   heap[i] = k;
   return &x;
}
```

i: vacant position
j: larger child

height of a heap = $\lceil \log_2 (n+1) \rceil$
complexity = O(log n)



The End of Trees Part I

Next Topic:

Trees Part II

國立清華大學 電機工程學系
EE2410 Data Structure

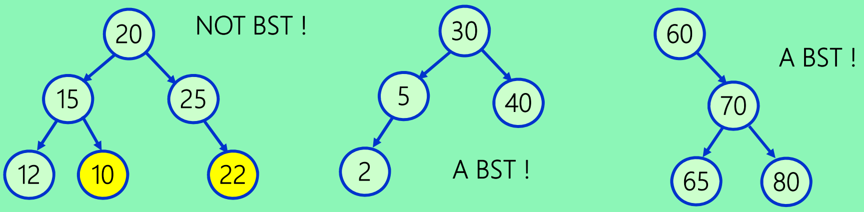
Chapter 5
Trees (Part II)

Outline

- **Binary Search Trees**
- **Selection Trees**
- **Forests**
- **Set Representation**
- **Counting Binary Tree**

Binary Search Tree (BST)

- **Definition**
  - A binary search tree is a binary tree
  - The left & right sub-tress are also binary search tree
  - If it is not empty, then it satisfies the following
    - Every element has a unique key
    - For every node N, Key(N) > Key(Left Subtree(N))
    - For every node N, Key(N) < Key(Right Subtree(N))

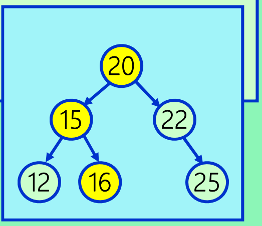


Why Binary Search Tree ?

- **Heap**
  - is well suited for priority queue
  - but is bad when deleting an arbitrary element is needed → O(n)
- **Binary Search Tree**
  - has a better performance when the functions to be performed are search, insert, and delete
  - operations can be done by key or by rank
  - Examples:
    - find an element with key *x*
    - delete an 5-th element of the tree

Recursive Search Of a BST

```
template <class Type> // Driver
BstNode<Type>*  BST<Type>::Search(const Element<Type>& x)
// Search the binary search tree (*this) for an element with key x
// If such an element is found, return a pointer to the node that contains it
{
   return  Search(root, x);
}

template <class Type> // Workhorse
BstNode<Type>* BST<Type>::Search(BstNode<Type>*b, Element<Type>& x) {
   if ( ! b ) return 0;
   if ( x.key == b→data.key ) return b;
   if ( x.key < b→data.key )
             return Search(b→LeftChild, x);
   return Search(b→RightChild, x);
}
```



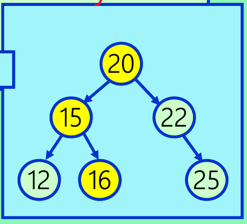
Each node has tree fields: LeftChild, data, RightChild
data is of class Element<Type> having a field *key*

Iterative Search Of a BST

```
template <class Type>
BstNode<Type>*  BST<Type>::IterSearch(const Element<Type>& x)
//  Search the binary search tree for an element with key x
{
  BstNode<Type>  *found, *t = root;
  while (1) {
     if ( t == 0)  { found = 0;  break; }  // the key being searched is not existent
     else {
           if ( x.key == t→data.key )  { found = t;  break; }
           else if ( x.key > t→data.key )  t = t→RightChild;
           else t = t→LeftChild;
     }
  }
  return (found);
}
```

Complexity: O(h), where h is the height

Finding Element with the key of 16



iteration	1	2	3
t→data.key	root (20)	15	16
found	-	-	16

Search A BST By Rank

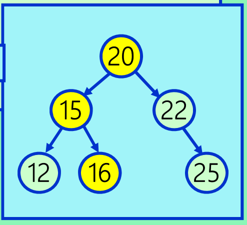
```
template <class Type>
BstNode<Type>*  BST<Type>::Search(int k)
// Search the binary search tree for the k-th smallest element
{
  BstNode<Type>  *t = root;
  while( t )
  {
     if ( k==t→LeftSize )  return t;
     if ( k < t→LeftSize )  t = t→LeftChild;
     else {
          k = k – t→LeftSize;
          t = t →RightChild;

     }
  }
  return 0;
}
```

Each node has an additional field: LeftSize
→ the number of elements in its left subtree + 1

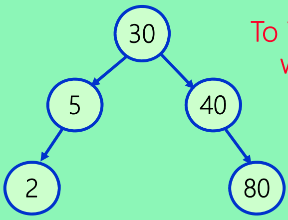
Complexity: O(h), where h is the height

iteration	1	2	3
t→LeftSize	(20) 4	(15) 2	(16) 1
k	3	3	1

Finding 3rd element:



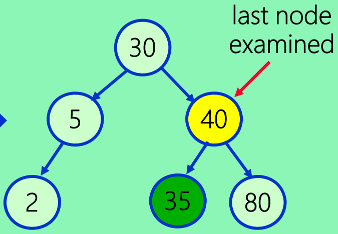
Example: Insertion to a BST



To insert a node with key 35

→ A search is first carried out → unsuccessful !

→ The new node is inserted as a child of the last node examined →



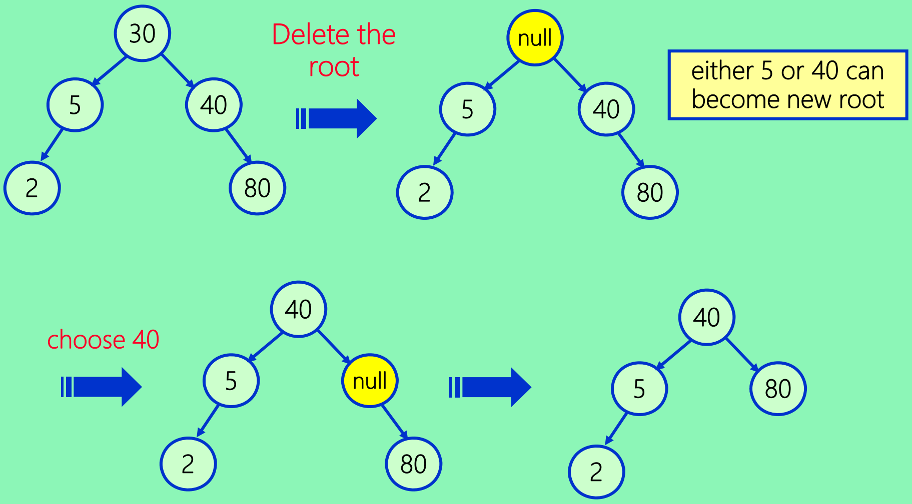
If a node has a LeftSize field, then it has to be updated

Insertion to A BST

```
template <class Type>
BstNode<Type>*  BST<Type>::Insert(const Element<Type>& x)
// Inserting x into the binary search tree
{
   // Search for x.key, q is parent of p
   BstNode<Type>  *p = root;  BstNode<Type>  *q = 0;
   while(p) {
        q = p;
        if ( x.key == p→data.key )  return FALSE;  // x.key is already in tree
        if ( x.key < p→data.key ) p = p →LeftChild;
        else  p = p→RightChild;
   }
   // Perform insertion
   p = new BstNode<Type>;
   p→LeftChild = p→RightChild = 0;  p→data = x;
   if ( ! root )  root = p;
   else  if  ( x.key < q→data.key )  q→LeftChild = p;
   else  q→RightChild = p;
   return  TRUE;
}
```

Complexity: O(h)
where h is the height of BST

Example: Deletion From a BST



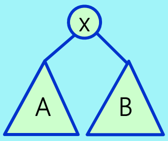
Delete the root

either 5 or 40 can become new root

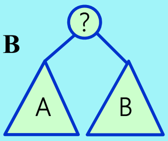
choose 40

Joining and Splitting Binary Trees

- **C.ThreeWayJoin(A, x, B)**
  - Assume that
    - each element in A has a smaller key than x
    - each element in B has a larger key than x
  - The operation creates a binary search tree C
    - consisting of every elements in A and B, and element x



- **C.TwoWayJoin(A, B)**
  - Assume that all keys of A are smaller than all keys of B
  - The operation creates a binary search tree C
    - consisting of all elements in A and B



- **A.Split(i, B, x, C)**
  - To split A into B and C
  - B: all elements with a key smaller than i
  - C: all elements with a key larger than i
  - x: if an element has a key equal to i, then it is copied into x

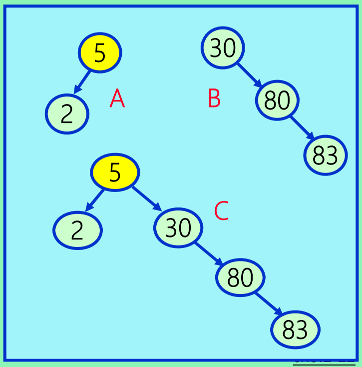
Example: Joining BST's

- **Three-way join operation C.ThreeWayJoin(A, x, B)**
  - Time is O(1)
  - The height of the new is max{ height(A), height(B) } + 1
- **Two-way join operation C.TwoWayJoin(A, B)**
  - Step 1: delete from A the record x with the largest key

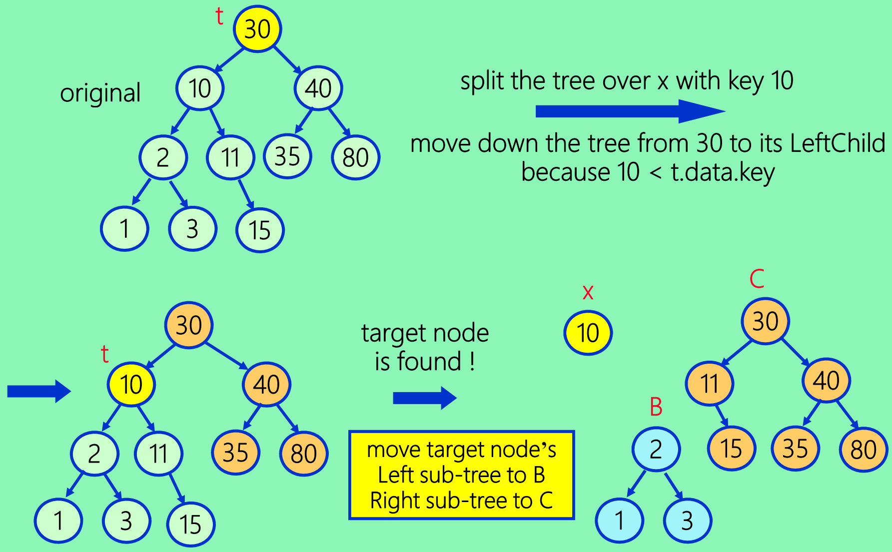
    Let the resulting tree as A'
  - Step 2: perform three-way join operation

    C.ThreeWayJoin(A', x, B)
  - Time is O(height(A))
  - The height of the new tree is

    max{ height(A'), height(B) } + 1



Example: Splitting a BST



split the tree over x with key 10

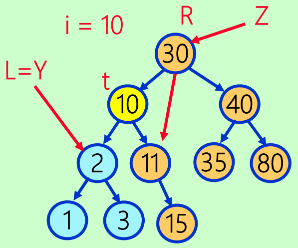
move down the tree from 30 to its LeftChild because 10 < t.data.key

target node is found !

move target node's
Left sub-tree to B
Right sub-tree to C

Splitting A BST

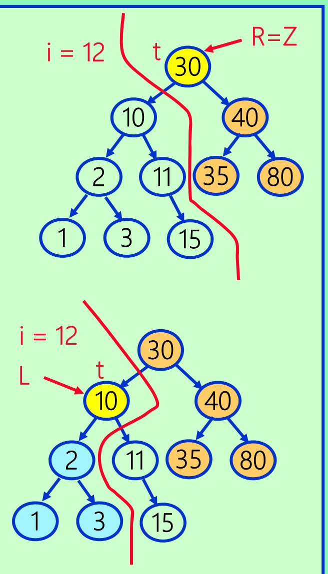
```
template <class Type>
Element<Type>* BST<Type>::Split(Type i, BST<Type>& B,
                                          Element<Type>& x, BST<Type>& C)
// Split the binary search tree with respect to key i
{
   if ( ! root ) {  B.root = C.root = 0; return 0; } // empty tree
   // create head nodes for B and C
   BstNode<Type>  *Y = new BstNode<Type>;  BstNode<Type> *L = Y;
   BstNode<Type>  *Z = new BstNode<Type>;  BstNode<Type> *R = Z;
   BstNode<Type>  *t = root;
   while ( t ) {
        if ( i == t→data.key ) {  // split at t
             L → RightChild = t→LeftChild;
             R →LeftChild = t→RightChild;
             x = t→data;
             B.root = Y→RightChild;  delete Y;
             C.root = Z→LeftChild;  delete Z;
             return &x;
        }
        TO BE CONTINUED ...
```



L and R are the frontiers of B and C

Splitting A BST

```
        else if ( i < t→data.key ) {
                R→LeftChild = t;
                R = t;  t = t→LeftChild;
        }
        else {   //  i > t→data.key
                L→RightChild = t;
                L = t;  t = t→RightChild;
        }
    }
    // Set 0 pointers and delete head nodes
    L→RightChild = R→LeftChild = 0;
    B.root = Y→RightChild;  delete Y;
    C.root = Z→LeftChild; delete Z;
    return 0;
}
```



Outline

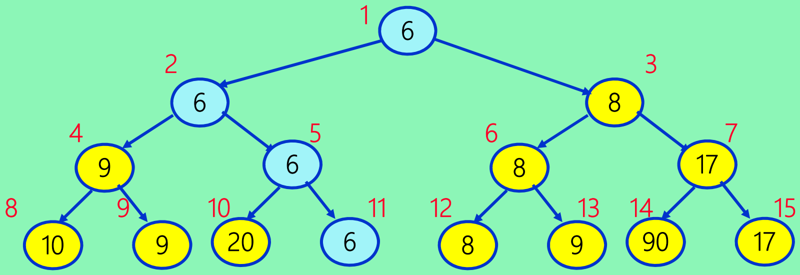
- **Binary Search Trees**
- **Selection Trees**
- **Forests**
- **Set Representation**
- **Counting Binary Tree**

Problem of Multi-Way Merging

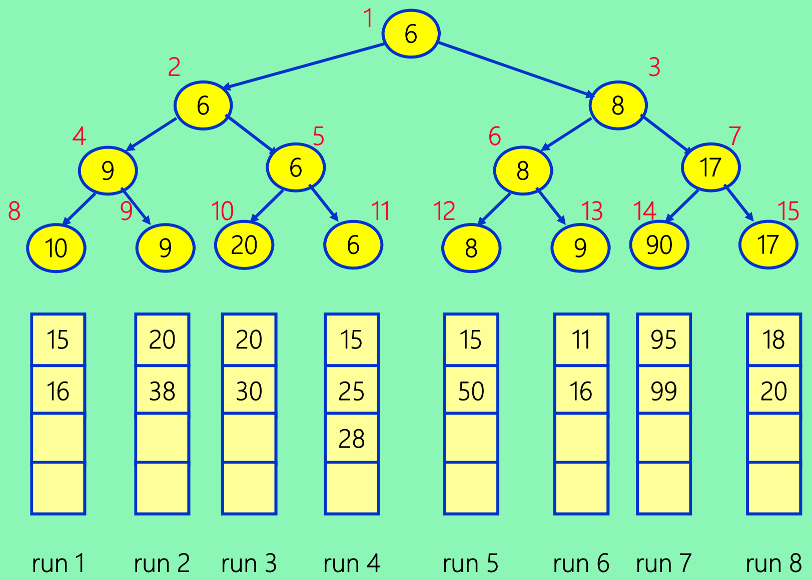
- **Problem**
  - Merge k ordered sequences, called runs, into a single ordered sequence
- **Example**
  - R1 = {15, 16}, R2 = {20, 38}, and R3 = {1, 17}
  - The merged output = {1, 15, 16, 17, 20, 38}
- **A naïve algorithm**
  - Find the smallest element by examining the first element of each run
  - Complexity is O(n · k), where n is the total number of elements in the k runs
  - But we can actually accomplish this in O(n · log k)

Winner Tree

- **Definition**
  - A winner tree is a complete binary tree in which each node represents the smallest of its two children
- **The construction of a winner tree**
  - Is like the playing of a tournament in which the winner is the record with the smaller key

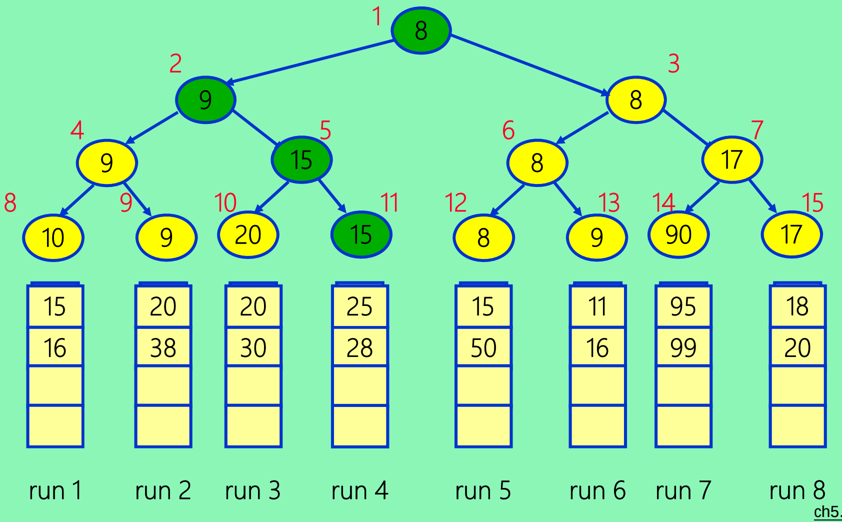


Winner Tree for k = 8



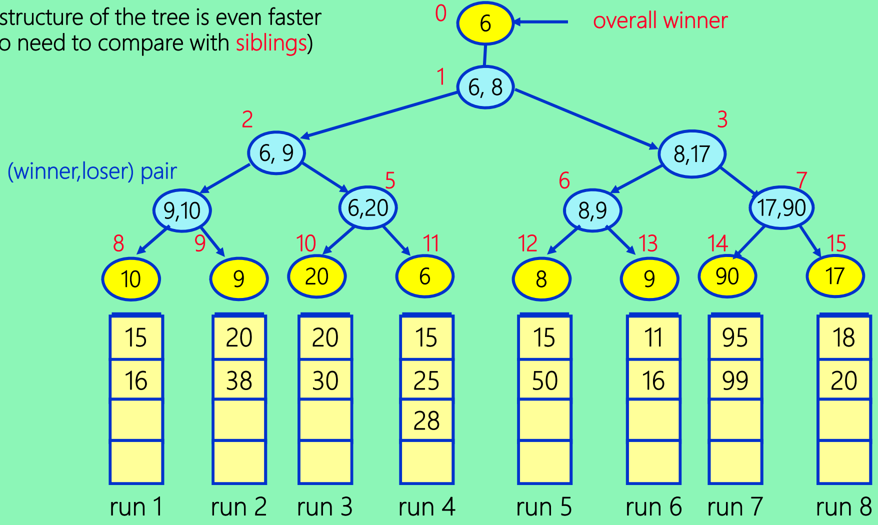
Winner Tree After One Record Is Output

After deleting the smallest:
The tournament is replayed only along the path from node 11 to the root



Loser Tree

A loser is recorded in the non-leaf node
Restructure of the tree is even faster
(No need to compare with siblings)

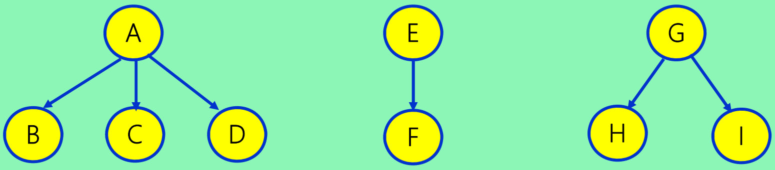


Outline

- **Binary Search Trees**
- **Selection Trees**
- **Forests**
- **Set Representation**
- **Counting Binary Tree**

Forrest

- **Definition**
  - A Forest is a set of n ≧0 disjoint trees
- **Operations**
  - Transforming a forest into a binary tree
  - Forest traversal
- **Examples**



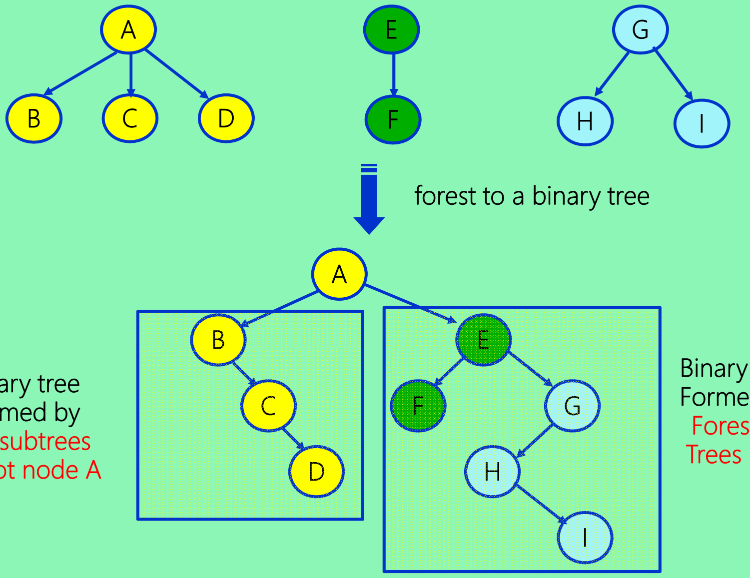
Transforming A Forest Into a BT

- **Definition**
  - **If T_1, T_2, ... , T_n is a forest of trees, then the binary tree corresponding to this forest, denoted by B(T_1,...,T_n)**

    (1) is empty if n = 0;

    (2) has root equal to root (T_1);
    - **Left-subtree: B(T_{11},...,T_{1m}), where T_{11},...,T_{1m} are the sub-trees of root(T_1)**
    - **Right-subtree: B(T_2,...,T_n).**

Binary Tree Representation
嫡長子-庶子 Concept



forest to a binary tree

Binary tree
Formed by
The subtrees
of root node A

Binary tree
Formed by
Forest of
Trees E, G

Forest Traversal

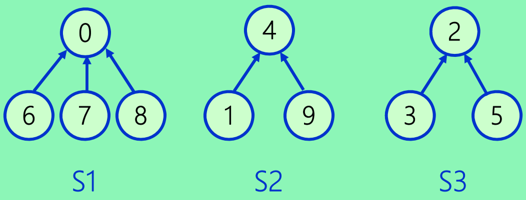
- **Forest Preorder Traversal**
  - **If F is empty then return**
  - **Visit the root of the first tree of F**
  - **Traverse the sub-trees of the first tree in forest preorder**
  - **Traverse the remaining trees of F in forest preorder**

Outline

- Binary Search Trees
- Selection Trees
- Forests
- Set Representation
- Counting Binary Tree

Set Representation

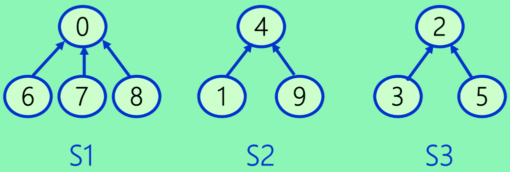
- Use of trees to represent sets
- Elements: {0, 1, 2, 3, ..., n-1}
  - indices into a symbol table where actual names are stored
- Assume that the sets being represented are mutually disjoint
  - For example
    - S1 = {0, 6, 7, 8}
    - S2 = {1, 4, 9}
    - S3 = {2, 3, 5}



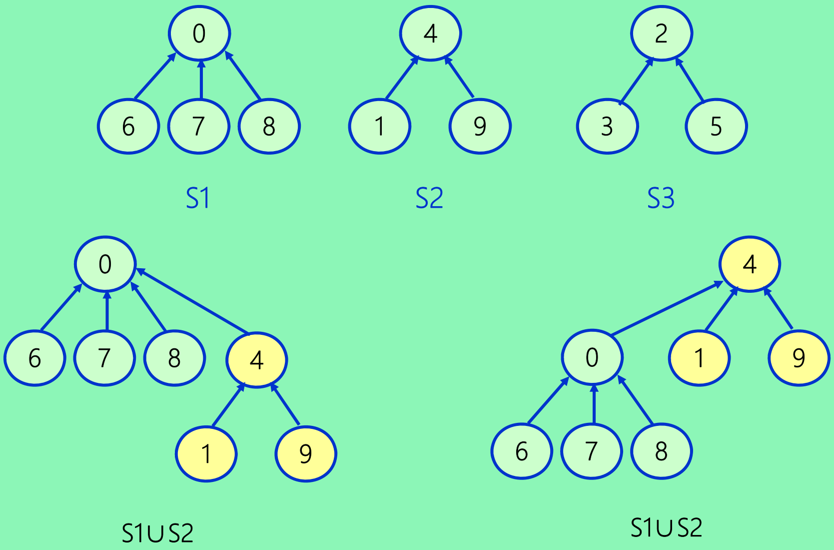
Set Operations

- **Disjoint set union**
  - **If S_i and S_j are two disjoint sets, then**

    $S_i \cup S_j$ = {all elements x such that x is in S_i or S_j}
  - **For example $S1 \cup S2$ = {0, 6, 7, 8, 1, 4, 9}**
- **Find(i)**
  - **Find the set containing i.**
  - **For examples**
    - **3 is in set S_3**
    - **8 is in set S_1**

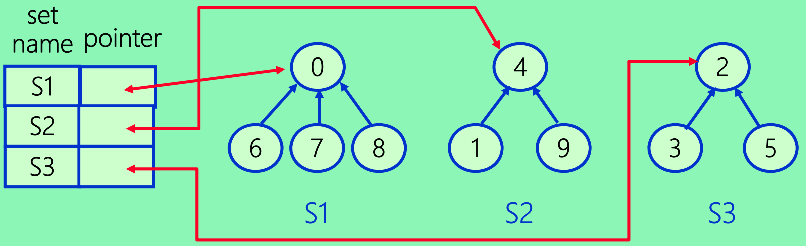


Union Operation



Data Structure

- **Symbol Table and Sets**
  - **A table entry is (set name, pointer)**



- **Array indicating a node's parent**

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

Class and Operations for Sets

```
class Sets {
public:
      // Set operations follow
private:
      int   *parent;
      int   n;  // number of set elements
};
Sets:Sets(int sz = HeapSize)
{
      n = sz;  parent = new int[sz];
      for ( int i=0; i<n; i++)  parent[i] = -1;
}
void  Sets::SimpleUnion(int i, int j)
// replace the disjoint sets with roots i and j, i != j with their union
{
      parent[i] = j;
}
int   Sets::SimpleFind(int i)
{
      while( parent[i] >=0) i = parent[i];            return i;
}
```

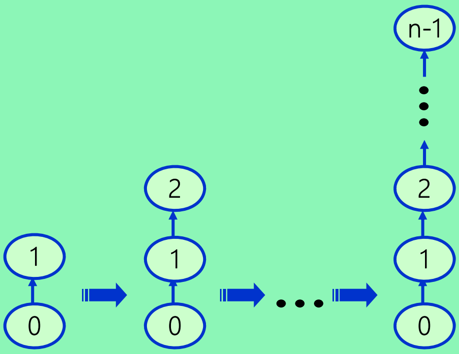
Worst-Case Scenario

- **Consider the following union-find operations**

  union(0, 1), union(1, 2), union(2, 3), union(3, 4), ..., union(n-2, n-1),

  find(0), find(1), ..., find(n-1)

- **The results**

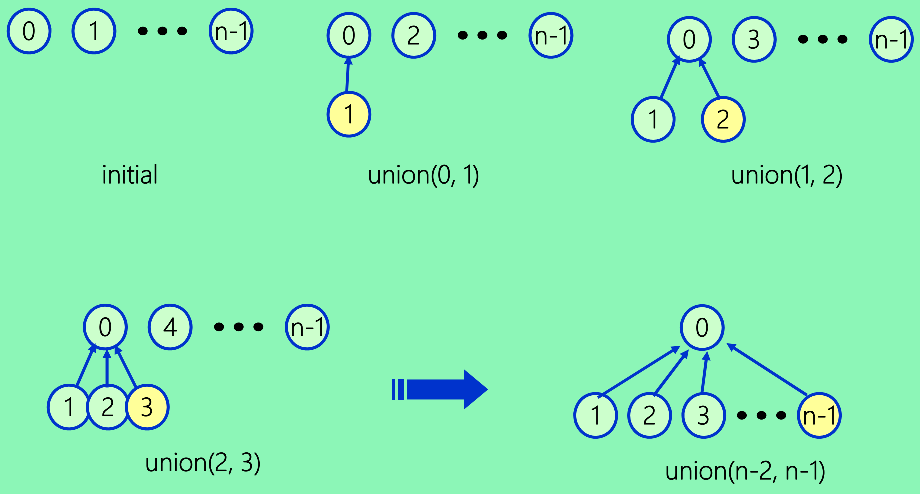


find(0) = O(n)
find(1) = O(n-1)
find(2) = O(n-2)
...
find(n-1) = 1
----
total = $O(n^2)$

Weighted Rule for Union(i, j)

- **Definition**
  - **If the number of nodes in the tree with root i is less than the number in the tree with root j, then make j the parent of i; otherwise make i the parent of j.**
- **Data Structure**
  - **A count field in the root node is used to indicate the total number of elements in the tree**

Trees Obtained Using The Weighted Rule



initial

union(0, 1)

union(1, 2)

union(2, 3)

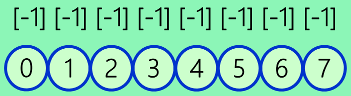
union(n-2, n-1)

Union Algorithm With Weighting Rule

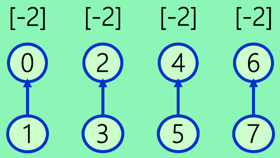
```
void  Sets::WeightedUnion(int i, int j)
// Union sets with roots i and j, i ≠j, using the weighted rule
// parent[i] = - count[i] and parent[j] = -count[j]
{
    int  temp = parent[i] + parent[j];
    if (parent[i] > parent[j]) {      // i has fewer nodes
        parent[i] = j;
        parent[j] = temp;
    }
    else{               // j has fewer nodes or i and j have the same number
    of nodes
        parent[j] = i;
        parent[i] = temp;
    }
}
```

It is provable that:
The height of the tree created by a sequence of unions over one-node trees is no greater than floor($\log_2$ n) + 1

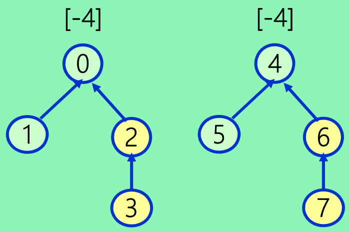
Trees Achieving Worst-Case Bound



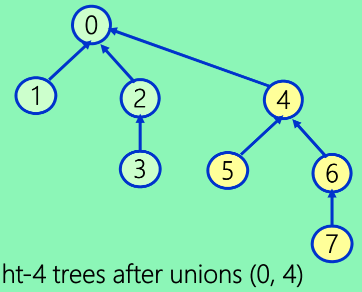
initial height-1 trees



Height-2 trees after unions
(0, 1), (2, 3), (4, 5), (6, 7)



Height-3 trees after unions
(0, 2), (4, 6)



Height-4 trees after unions (0, 4)

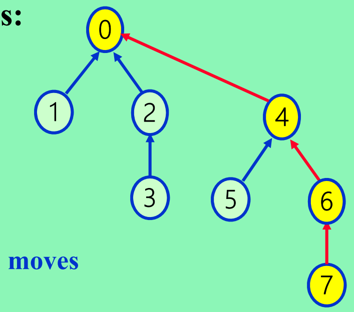
Collapsing Rule for Union(i, j)

- **Definition**
  - **If j is a node on the path from i to its root and parent[i] ≠root(i), then set parent[j] to root(i)**
- **Example**
  - **processing eight find operations:**

    find(7), find(7), ..., find(7)
  - **SimpleFind**

    ➔ **3*8 = 24 moves**
  - **CollapsingFind**

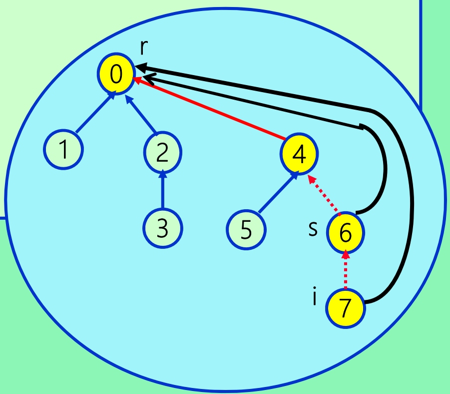
    ➔ **$\underbrace{3 + 3} + 1 + 1+ 1+ 1+ 1+ 1+ 1 = 13$ moves**



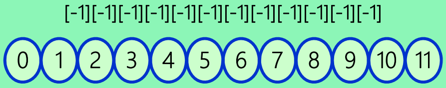
1st find(7): including traversal & collapsing

Find Algorithm With Collapsing Rule

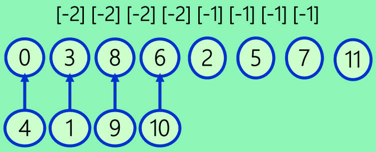
```
int  Sets::CollapsingFind(int i)
// Find the root of the tree containing element i
// Use the collapsing rule to collapse all nodes from i to the root
{
     for ( int r = i; parent[r] >= 0; r = parent[r]);  // find root
     while ( r != i ) {
          int  s = parent[i];
          parent[i] = r;
          i = s;
     }
     return r;
}
```



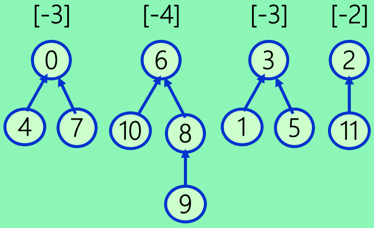
Application To Equivalence Classes



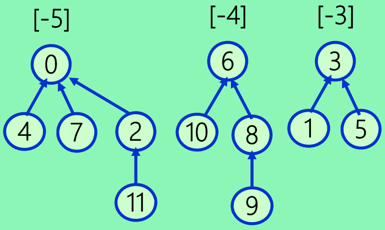
initial height-1 trees



Height-2 trees after processing
0≡4, 3≡1, 8≡9, 6≡10

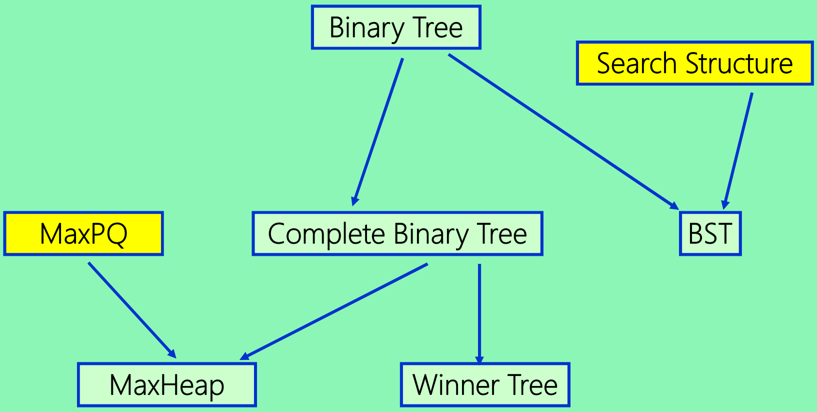


Trees after processing
7≡4, 6≡8, 3≡5, 2≡11



Trees after processing
11≡0

Inheritance Graph Among Trees



Binary Tree

Search Structure

MaxPQ

Complete Binary Tree

BST

MaxHeap

Winner Tree

Outline

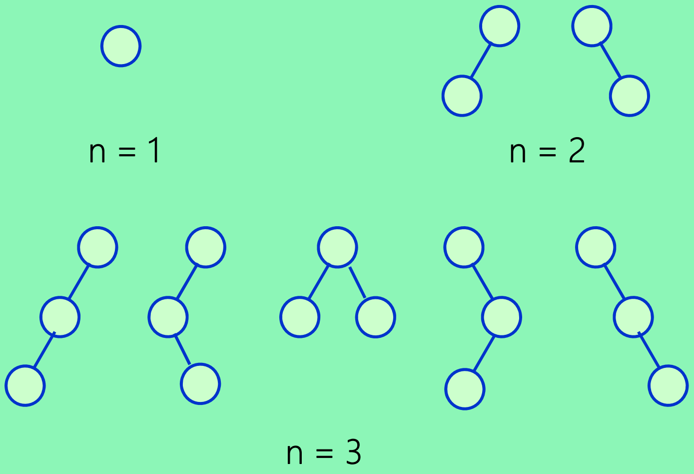
- **Binary Search Trees**
- **Selection Trees**
- **Forests**
- **Set Representation**
- **Counting Binary Tree**

Counting Binary Trees

- **Problem 1**
  - **What is the total possible number of distinct binary trees having n nodes?**
- **Problem 2**
  - **What is the number of permutations of the numbers from 1 through n obtainable by a stack ?**
- **Problem 3**
  - **What is the number of distinct ways of multiplying n+1 matrices ?**

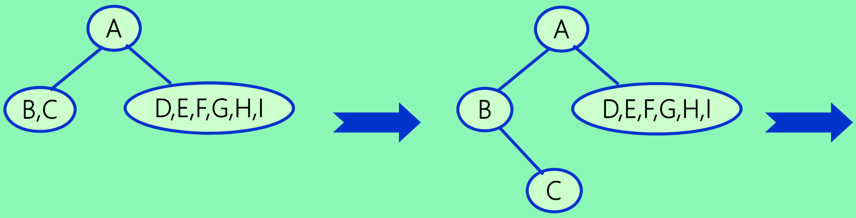
➔ All the three problems have the same answer !

Example: Distinct Binary Trees



Distinct Binary Trees According To A PreOrder / InOrder Sequence Pair

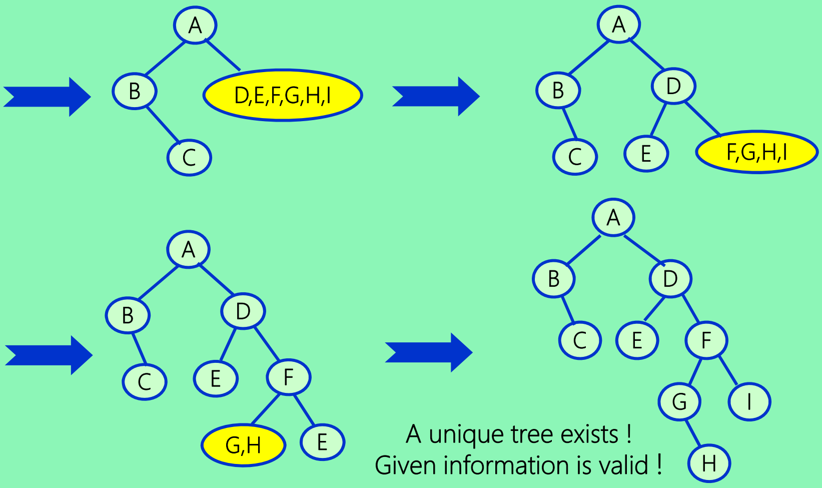
- **Given**
  - **the pre-order sequence A B C D E F G H I**
  - **the in-order sequence B C A E D G H F I**
- **Question**
  - **Does the given information defines a unique tree ?**
- **Construction Process**



Tree Construction Using Pre-order and In-order Information

Pre-order: **A B C D E F G H I**

In-order: B C A E D G H F I

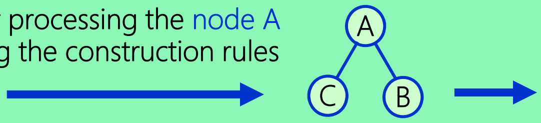


A unique tree exists !
Given information is valid !

What Orders Are Valid ?

- **Given**
  - **the pre-order sequence A B C**
  - **the in-order sequence C A B**
- **Question**
  - **Is the given order corresponds to any tree ?**
- **Answer**
  - **No, there is no binary tree according to this order pair**

after processing the node A
using the construction rules

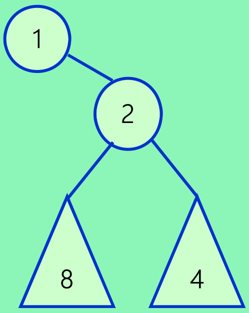


violate the preorder

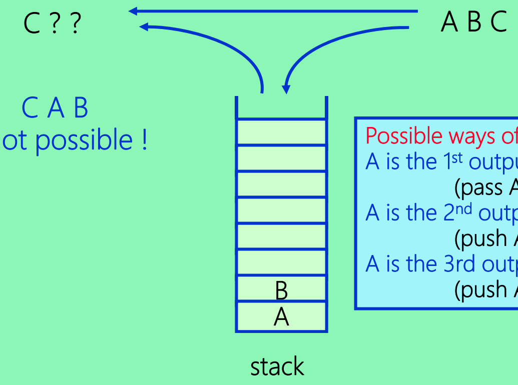
Example of Invalid Order Pair

- **The following is not valid**
  - **the pre-order sequence: 1 2 3 4 5 6 7 8 9 10**
  - **the in-order sequence: 1 8 . . 2 . . 4 . .**

When partitioning the tree based on node 2
→ node 8 is in the left sub-tree,
→ while node 4 in the right sub-tree
→ will violate the preorder 8 < 4



Stack Permutations



C A B
is not possible !

Possible ways of bi-partitioning:
A is the 1st output → A(BC) = T(0) · T(2)
(pass A + permutations of (BC))
A is the 2nd output → (B)A(C) = T(1) · T(1)
(push A, pass B, pop A, pass C)
A is the 3rd output → (BC) A = T(2) · T(0)
(push A + permutations of (BC))

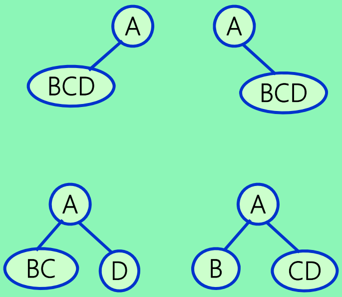
(A B C), (A C B), (B A C), (B C A), (C B A)
→ all valid permutations and valid in-order sequence

Total # of distinct trees = Total # of permutations through a stack

---

Possible Bi-Partitioning's For n=4

- **Preorder: A B C D**
- **Possible bi-partitioning's:**
  - **A(BCD) → b(0)b(3) = 5**
  - **(B)A(CD) → b(1)b(2) = 2**
  - **(BC)A(D) → b(2)b(1) = 2**
  - **(BCD)A → b(3)b(0) = 5**



b(4) = b(0)b(3) + b(1)b(2) + b(2)b(1) + b(3)b(0) = 14

$$b_n = \sum_{i=0}^{n-1} b_i \, b_{n-i-1} \quad n \geqq 1, \text{ and } b_0=1$$

Matrix Multiplication

- **Problem**
  - compute n matrices $M_1 * M_2 * M_3 * \cdots * M_n$
- **Matrix multiplication is associative**
- **For n=3, there are two ways**

  $M_1 * (M_2 * M_3)$ and $(M_1 * M_2) * M_3$
- **For n=4, there are five ways**

  $M_1 * ((M_2 * M_3) * M_4)$ $M_1 * (M_2 * (M_3 * M_4))$

  $(M_1 * M_2) * (M_3 * M_4)$

  $((M_1 * M_2) * M_3) * M_4$ $(M_1 * (M_2 * M_3)) * M_4$
- **Let B_n represents ways of multiplying n+1 matrices**
  - Then, $B_0=1, B_1=1,$

Bi-partitioning:

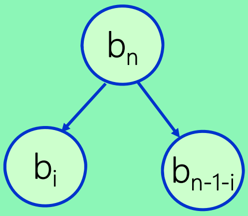
$$b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1} \quad n \geqq 1, \text{ and } b_0=1$$

$$B_n = \sum_{i=0}^{n-1} B_i B_{n-i-1} \quad n \geqq 2$$

Matrix multiplication

Distinct BT v.s. Matrix Multiplication

- **The no. of ways of doing matrix multiplication**
  - **can be characterized by recursive bi-partitioning as seen earlier**
- **Similarly, the number of distinct binary trees**
  - **can be characterized by recursive bi-partitioning, too**



$$b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1} \quad n \geqq 1, \text{ and } b_0=1$$

b_i 代表連乘 (i+1) 個矩陣的總共做法

Solving Bi-Partitioning

- To obtain the number of distinct binary trees
  - We have to solve the following recurrence relation

    $b_n = \sum_{i=0}^{n-1} b_i\, b_{n-i-1} \quad n \geqq 1, \text{ and } b_0=1$
- Let $B(x) = b_0+b_1x^1+b_2x^2+b_3x^3 + \ldots$
- Then $xB^2(x) = x\,[b_0+b_1x^1+b_2x^2+\ldots][b_0+b_1x^1+b_2x^2+\ldots]$

  $= B(x)-1$ using the recurrence relation
- Solve the quadratic equation $b_n = O(4^n/n^{1.5})$

The End of Trees Part II

Next Topic:

Graph

國立清華大學 電機工程學系
EE2410 Data Structure

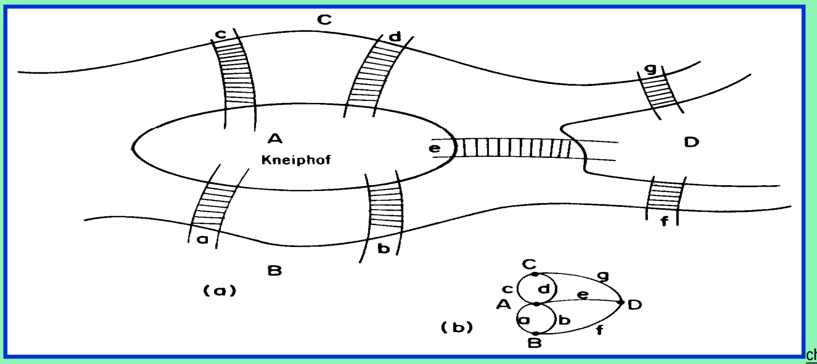
Chapter 6
Graph (Part I)

Outline

- **The Graph Abstract Data Type**
  - **Introduction**
  - **Definitions**
  - **Graph Representations**
- **Elementary Graph Operations**
- **Minimum Cost Spanning Trees**

River Pregel in Konigsberg

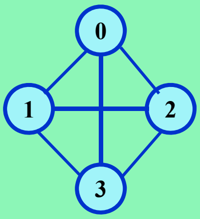
- **Problem**
  - **Is there a cyclic walk that traverses every bridge only once (1736)**
- **For an Euler's path to exist**
  - **The degree of each vertex is even**
  - **The degree is the number of edges incident to a vertex**



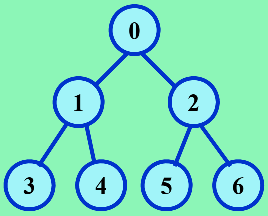
Definition and Notations of Graph

- **Definition**
  - **A graph, G, consists of two sets, V and E**
  - **V is a finite nonempty set of vertices → V(G)**
  - **E is a set of pairs of vertices, called edges → E(G)**
- **Terminology**
  - **Undirected graph: edges are not directed**
  - **Directed graph (Digraph): edges are directed**
    - **directed pair <u, v>, u is the tail and v is the head**

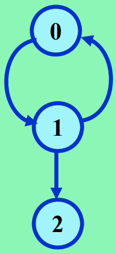
Sample Graphs



G1



G2



G3

V(G1) = { 0, 1, 2, 3 }; E(G1) = { (0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3) }

V(G2) = { 0, 1, 2, 3, 4, 5, 6 }; E(G2) = { (0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6) }

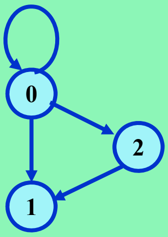
V(G3) = { 0, 1, 2 }; E(G3) = { <0, 1>, <1, 0>, <1, 2> }

Question: What are the maximum number of edges in a graph with n nodes?

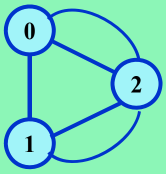
→ n · (n-1)/2 for undirected graph and n · (n-1) for digraph

Restrictions on Graph

- **No self loops**
  - **A self loop (or self edge) is an edge from a vertex v back to itself**
  - **That is, (v, v) and <v, v> are not legal**
- **No multiple occurrences of the same edge**



Graph with self edge



Multigraph

Terminology

- Complete graph
  - An n-vertex, undirected graph with exactly n(n-1)/2 edges is said to be complete
  - An n-vertex, directed graph with exactly n(n-1) edges is said to be complete
- Adjacent nodes
  - If (u, v) is an edge in E(G), then u and v are adjacent, and edge (u, v) is incident on vertices u and v
  - If <u, v> is a directed edge, then u is adjacent to v, and v is adjacent from u
- A subgraph of G
  - is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$

Path in A Graph

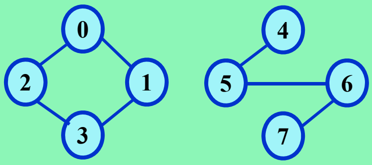
- A path from vertex u to vertex v
  - is a sequence of vertices $u, i_1, i_2, ..., i_k, v$ such that $(u, i_1), (i_1, i_2), ..., (i_k, v)$ are all edges in E(G)
  - A path (0, 1), (1, 3), (3, 2) is also written as 0, 1, 3, 2
- A simple path
  - is a path in which all vertices except possibly the first and last are distinct
- The length of a path
  - is the number of edges on a path
- A cycle
  - is a simple path in which the first and last vertices are the same

Connected Component

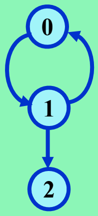
- **Connected vertices**
  - In a undirected graph, two vertices u and v are said to be connected iff there is a path in G from u to v
- **Connected graph**
  - An undirected graph is said to be connected iff for every pair of distinct vertices u and v in V(G) there is a path from u to v in G
- **A connected component**
  - is a maximal connected subgraph
- **A tree is a connected acyclic graph**

Strongly Connected Component

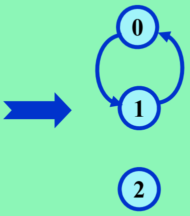
- **Strongly connected graph**
  - A digraph G is said to be strongly connected iff for every pair of distinct vertices u and v in V(G), there is a directed path from u to v and also from v to u
- **Strongly connected component (SCC)**
  - A SCC is a maximal subgraph that is strongly connected



A graph with two connected components



G3



Two SCC's of G3

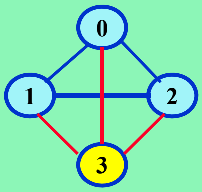
Abstract Data Type Graph

```
class  Graph
{
//  objects: A nonempty set of vertices and a set of undirected edges
//  where each edge is a pair of vertices
public:
     Graph ();  // Create an empty graph

     void   InsertVertex(Vertex  v);  //  Insert v into graph; v has no incident edges

     void   InsertEdge(Vertex u, Vertex v);  // Insert edge (u, v) into graph

     void   DeleteVertex(Vertex v);   //  Delete v and all edges incident to it

     void   DeleteEdge(Vertex u, Vertex v);  //  Delete edge (u, v) from the graph

     Boolean  IsEmpty ();
             //  if graph has no vertices return TRUE(1); else return FALSE(0);

     List<Vertex>  Adjacent( Vertex v);
             //  return a list of all vertices that are adjacent to v
}
```

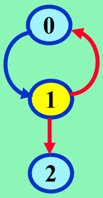
Graph Representations

- **Adjacency matrices**
- **Adjacency lists**
- **Adjacency multi-lists**

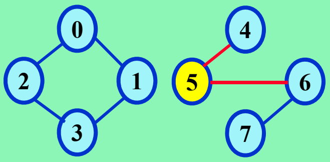
Adjacency Matrices



G1



G3



$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 2 & 1 & 1 & 0 & 1 \\ 3 & 1 & 1 & 1 & 0 \end{array}$$

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 \end{array}$$

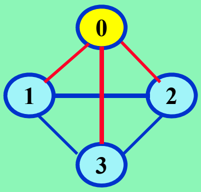
$$\begin{array}{c|cccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 3 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 7 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

Questions: How many edges? Is G connected ?

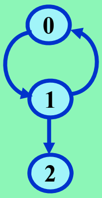
→ requires O(n^2)

Adjacency Lists

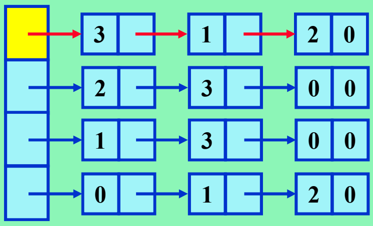
G1



G3



HeadNodes **data link**



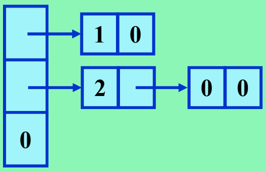
[0] → 3 → 1 → 2 0

[1] → 2 → 3 → 0 0

[2] → 1 → 3 → 0 0

[3] → 0 → 1 → 2 0

HeadNodes



[0] → 1 0

[1] → 2 → 0 0

[2] 0

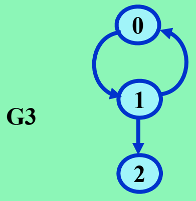
Graph Using Adjacency Lists

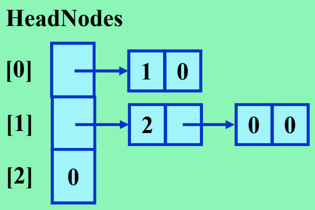
```
class  Graph
{
private:
       List<int>  *HeadNodes;
       int    n;
public:
       Graph( const  int  vertices = 0) : n ( vertices )
       {  HeadNodes = new List<int>[n]; } ;
};
```

Complexities of simple operations:

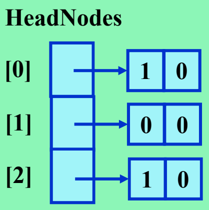
1. **Determine the total number of edges of a graph: O(n+e)**
2. **Determine the out-degree of a node: O(out-degree of the node)**
3. **Determine the in-degree of a node: needs inverse adjacency lists**

Inverse Adjacency Lists



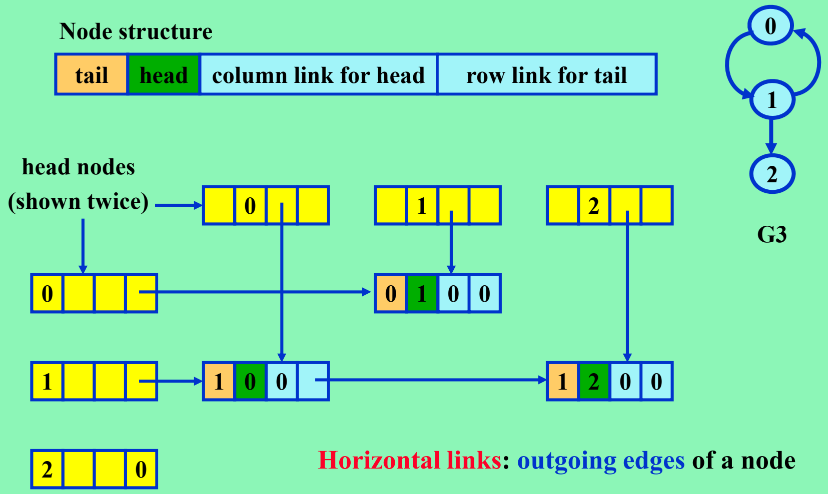


adjacency lists



inverse adjacency lists

Orthogonal List Representation



Horizontal links: **outgoing edges** of a node

Vertical links: **incoming edges** of a node

Adjacency Multi-Lists

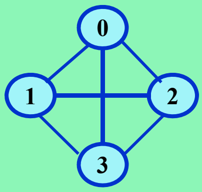
- **Motivations**
  - An **edge** (u,v) in adjacency lists is represented by **two entries**, one in list for u, the other in list for v
  - During graph traversal, we need to mark **an edge as visited** → need a better representation
- **Adjacency Multi-Lists**
  - There is **one node for each edge**
  - A node may be **shared** among several lists

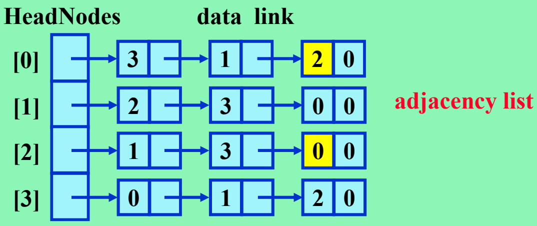
Node structure

m	vertex 1	vertex 2	list 1	list 2

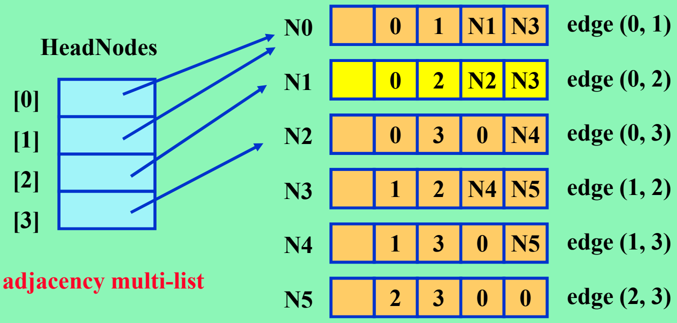
mark bit indicating whether or not an edge has been examined

Example: Adjacency Multi-Lists





[0]	3		1		2	0
[1]	2		3		0	0
[2]	1		3		0	0
[3]	0		1		2	0



N0		0	1	N1	N3	edge (0, 1)
N1		0	2	N2	N3	edge (0, 2)
N2		0	3	0	N4	edge (0, 3)
N3		1	2	N4	N5	edge (1, 2)
N4		1	3	0	N5	edge (1, 3)
N5		2	3	0	0	edge (2, 3)

ADT of Adjacency Multi-Lists

```
enum  Boolean  { FALSE, TRUE }
class   Graph;
class  GraphEdge {
friend  Graph;
private:
    Boolean  m;
    int   vertex1, vertex2;
    GraphEdge  *path1, *path2;
};
```

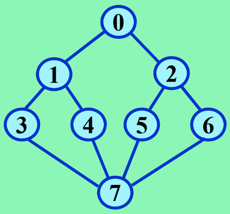
```
typedef  GraphEdge  *EdgePtr;
class Graph {
private:
    EdgePtr *HeadNodes;
    int  n;
public:
    Graph(const int);
};
```

```
Graph::Graph(int vertices=0) : n (vertices)
{
//  Set up the array of head nodes
    HeadNodes = new EdgePtr[n];
    for(i=0; i<n; i++)  HeadNodes[i] = 0;
}
```

Outline

- **The Graph Abstract Data Type**
- **Elementary Graph Operations**
  - **Depth First Search**
  - **Breadth First Search**
  - **Connected Components**
  - **Spanning Trees**
  - **Bi-connected Components**
- **Minimum Cost Spanning Trees**

Depth First Search

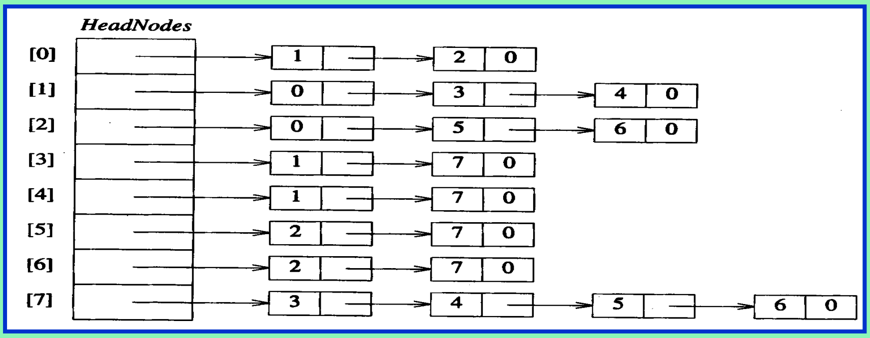


Depth First Search (DFS) orders: (for example)

0, 1, 3, 7, 4, 5, 2, 6

0, 1, 4, 7, 3, 5, 2, 6

etc.

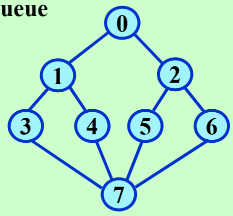


Depth First Search Algorithm

```
void  Graph::DFS()   //  Driver
{
   visited = new  Boolean[n];
   for ( int  i=0; i<n; i++ )  visited[i] = FALSE;

   DFS(0);  // start search at vertex 0

   delete[ ]  visited;
}

void  Graph::DFS(const int v) // Workhorse
// visit all previously unvisited vertices that are reachable from vertex v
{
   visited[v] = TRUE;
   for ( each vertex w adjacent to v )
            if ( ! visited[w] )  DFS(w);
   }
}
```

Breadth First Search

```
void  Graph::BFS( int  v )
// A breadth first search of the graph is carried out beginning at vertex v
// visited[i] is set to TRUE when v is visited. The algorithm uses a queue
{
  visited = new Boolean[n];
  for ( int i=0; i<n; i++)  visited[i] = FALSE;
  visited[v] = TRUE;
  Queue<int>  q;
  q.Insert(v);  //  add vertex v to the queue
  while ( ! q.IsEmpty() )  {
           v = *q.Delete(v);  //  remove vertex v from the queue
           for ( all vertices w adjacent to v ) {
                   if ( ! visited[w] ) {
                         q.Insert(w);
                         visited[w] = TRUE;
                   }
           }
   } // end of while loop
   delete [] visited;
}
```



BFS order: 0,1,2,3,4,5,6,7

Connected Components

- **For an undirected graph**
  - **The connected components can be computed by either DFS or BFS search**
  - **All nodes visited during a traversal along with their edges form a connected components**

```
void  Graph::Components()
// Determine the connected components of the graph
{
   visited = new Boolean[n];
   for ( int  i=0; i<n; i++)  visited[i] = FALSE;
   for ( i=0; i<n; i++) {
         if ( ! visited[i] ) {    // pick one node that is not visited yet
              DFS(i);  // Find a component
              OutputNewComponent();
         }
    delete [] visited;
}
```

Complexity = O(n+e)

for adjacency lists

Spanning Trees

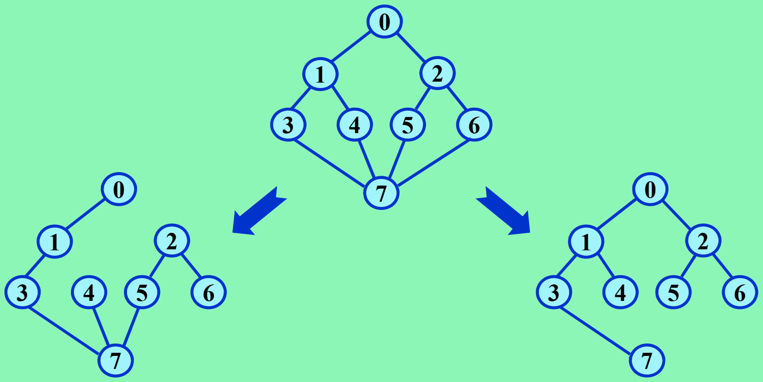
- **Definition**

  Any tree is a spanning tree of G if

  (1) The tree consists solely of edges in G

  (2) The tree includes all vertices in G
- **For a connected graph G**
  - **Depth-first or breadth-first search partitions the edges into two sets, T and N**
  - **T is the set of tree edges**
  - **N is the set of non-tree edges**
- **The tree edges of a traversal**
  - **and every vertex forms a spanning tree**

Examples: Spanning Trees



Depth-first spanning tree

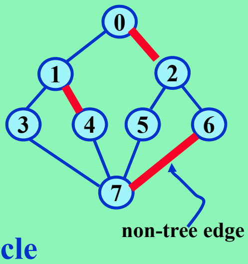
(starting from vertex 0)

Breadth-first spanning tree

(starting from vertex 0)

Creation of Circuit Equations

- **In a spanning tree of a connected graph**
  - **Each non-tree edge added to the tree forms a cycle**
  - **Each cycle is unique**
- **Application to circuit analysis**
  - **Represent a circuit as a graph**
  - **Find a spanning tree**
  - **Each non-tree edge corresponds to a cycle**
  - **Generate a current equation using Kirchhoff's law**
  - **A set of independent current equations are obtained**

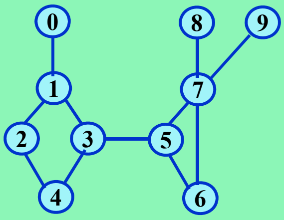


Minimal Connected Subgraph

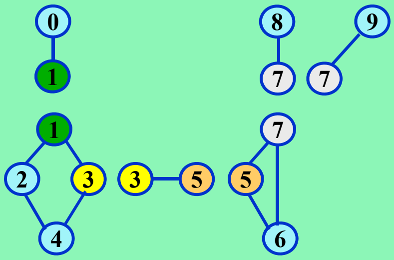
- **Property**
  - A spanning tree is a minimal sub-graph G' of G such that V(G') = V(G), and G' is connected
- **Reasons**
  - Any connected graph with n vertices must have n-1 edges
  - All connected graphs with n-1 edges are trees
  - Therefore, a spanning tree is a minimal sub-graph
- **Application to communication**
  - Vertices represent cities, while edges represents communication links
  - The minimum number of links connecting n cities is n-1
  - The cost of each link is different, represented as weight
  - Finding minimum-cost spanning tree is desired !

Articulation Point

- **Definition of Articulation Point**
  - A vertex v of G is an articulation point iff the deletion of v, together with the deletion of all edges incident to v, leaves behind a graph that has at least two connected components
- **Definition of Bi-connected Graph**
  - A bi-connected graph is a connected graph that has no articulation points



A connected graph



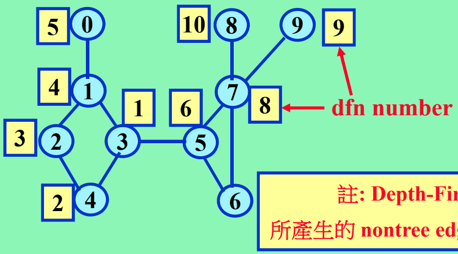
6 bi-connected components

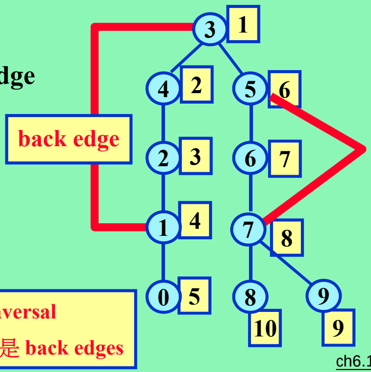
Bi-Connected Components

- **Definition**
  - A biconnected component of a connected graph G is a maximal biconnected subgraph H of G
  - By maximal, it means that G contains no other subgraph that is both biconnected and properly contains H
- **Properties**
  - A biconnected graph has just one biconnected component – the whole graph
  - Two biconnected components can have at most one vertex in common
  - No edge can be in two biconnected components
  - Hence, biconnected components of G partition the edges of G

Back Edge and Cross Edge

- **Depth first number (dfn)**
  - The order of a node visited during a depth first search
- **Back edge**
  - A nontree edge (u, v) is a back edge iff u is an ancestor of v or v is an ancestor of u
- **Cross edge**
  - A nontree edge that is not a back edge

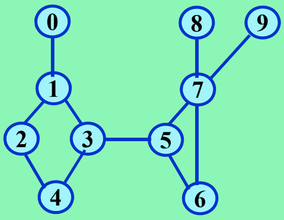


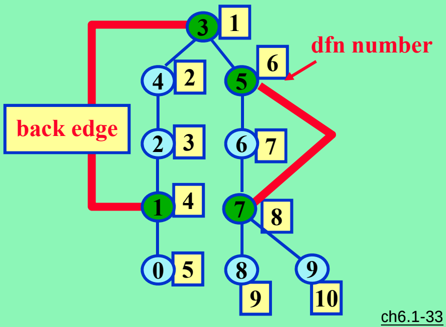


註: Depth-First Traversal
所產生的 nontree edges 都是 back edges

Where Are The Articulation Points ?

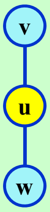
- **Root is an articulation point**
  - iff it has at least two children
- **Back path is a path starting from a vertex *u***
  - reaches an ancestor of u through u's descendants and single back edge
- **A Non-root vertex u is an articulation point iff**
  - (1) u has at least one child
  - (2) u has NO such child w that there exist a back path starting from w



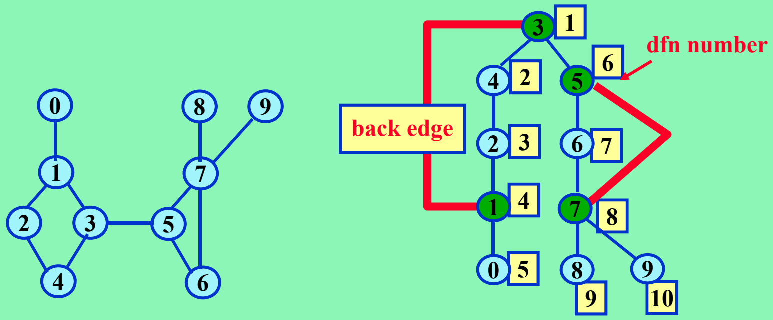


```
void  Graph::DfnLow(const int x)  //  begin DFS at vertex x
{
   num = 1;
   dfn = new int[n];
   low = new int[n];
   for ( int i=0; i<n; i++) {  dfn[i] = low[i] = 0; }
   DfnLow(x, -1);  // start at vertex x
   delete [ ]  dfn;
   delete [ ]  low;
}
void  Graph::DfnLow ( const int u, const int v)
//  Compute dfn and low while performing a depth first search beginning
//  at vertex u. vertex v is the parent (if any) of u in the resulting spanning tree
{
    dfn[u] = low[u] = num++;
    for ( each vertex w adjacent from u )
          if ( dfn[w]==0) {  w is an unvisited vertex
                DfnLow(w, u);
                low[u] = min2( low[u], low[w]);
          }
          else if ( w != v)  low[u] = min2( low[u], dfn[w] );  // back edge
     }
}
```

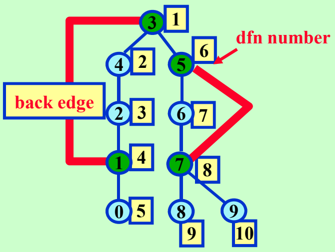
low(u) is the lowest depth first number reachable by back path starting from u

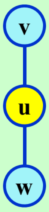


Example: Values of *dfn* and *low*



Vertex ID	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
dfn	5	4	3	1	2	6	7	8	9	10
low	5	1	1	1	1	6	6	6	9	10





```
void  Graph::Biconnected()
{
   num = 1;  dfn = new int[n];  low = new int[n];
   for ( int  i=0; i<n; i++) {  dfn[i] = low[i] = 0; }
   Biconnected(0, -1);  // start at vertex 0
   delete [] dfn;  delete [] low;
}
void  Graph::Biconnected ( const int u, const int v)
{
   dfn[u] = low[u] = num++;
   for ( each vertex w adjacent from u )
         if ( (w !=v) && (dfn[w] < dfn[u]) )  add (u, w) to stack S;
         if ( dfn[w] == 0) {  // w is an unvisited vertex
             Biconnected(w, u);    low[u] = min2( low[u], low[w] );
             if ( low[w] >= dfn[u])  {  // u an articulation point found
                 cout << “New biconnected components: “ << endl;
                 do {
                      delete an edge from the stack S;
                      let this edge be (x, y);    cout << x << “,” << y << endl;
                 } while ( (x,y) and (u,w) are not the same edge)
             }
         }
         else if ( w != v)  low[u] = min2( low[u], dfn[w] ) ;  // back edge
}
```

Outline

- **The Graph Abstract Data Type**
- **Elementary Graph Operations**
- **Minimum Cost Spanning Trees**
  - **Kruskal's Algorithm**
  - **Prim's Algorithm**
  - **Sollin's Algorithm**

Minimum-Cost Spanning Tree

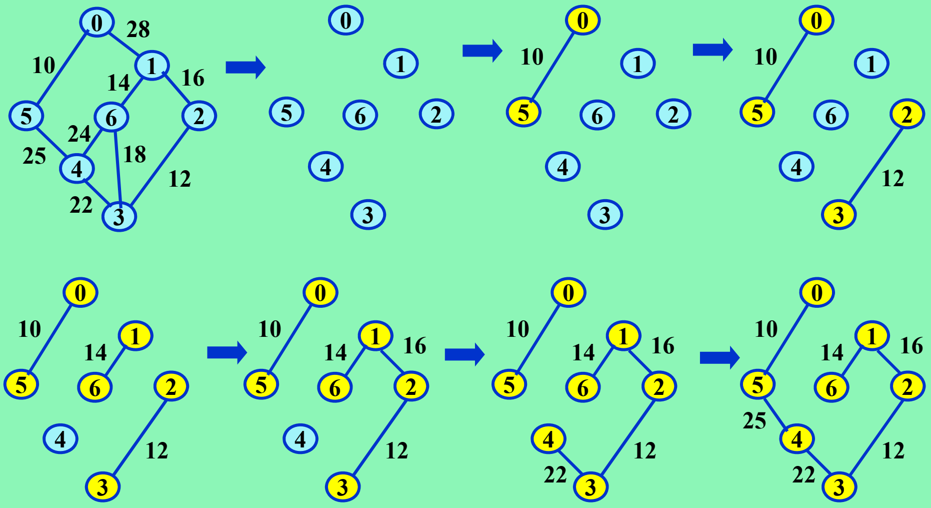
- **Cost of a spanning tree**
  - **is the sum of the costs (weights) of the edges in the spanning tree**
- **A minimum-cost spanning tree**
  - **is a spanning tree of least cost**
- **Greedy method**
  - **The solution is constructed in stages**
  - **At each stage, the best decision (using some criterion) is picked**
  - **No decision, once made, can be reversed**
- **Selection criterion in forming a min-cost spanning tree**

(1) Use only edges within the graph

(2) Use exactly n-1 edges

(3) Should not use edges that produce a cycle

Example of Forming A Min-Cost Spanning Tree – Kruskal's Algorithm



Total weight = 99

Kruskal's Algorithm

```
T = ∅;   T is the set of collected edges in the spanning tree
while ( ( T contains less than n-1 edges )  && ( E is not empty ) ) {
          choose an edge (v, w)  from E of lowest cost;
          delete (v, w) from E;
          if ( (v, w) does not create a cycle in T )  add (v, w) to T;
          else  discard (v, w);
}
if ( T contains fewer than n-1 edges )  {
      cout << "No spanning tree" << endl;
}
```

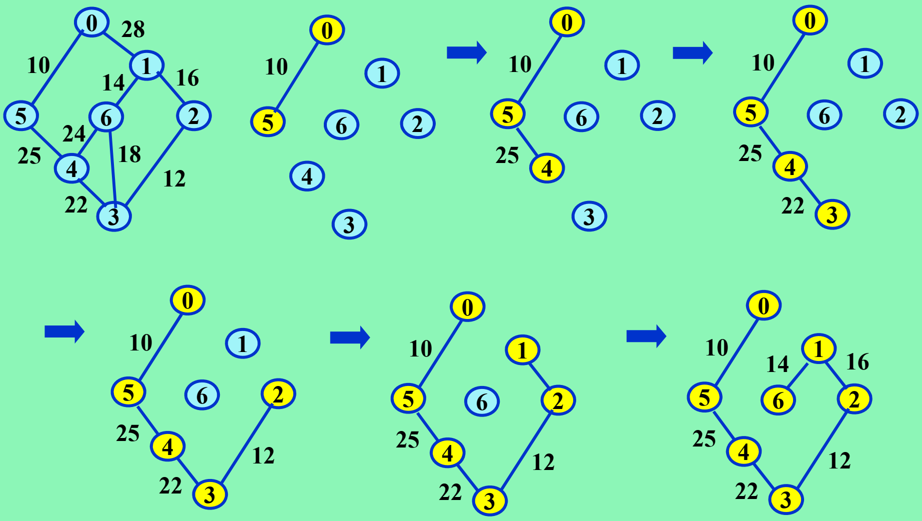
O(e·log e) if min heap is used, and set is used for cycle checking

It can be proved that: Kruskal's algorithm is optimal

(1) If there is a spanning tree, → Kruskal will find it

(2) If there is a min-cost spanning tree U, then there exists a cost-preserving transformation that maps U to the one Kruskal finds

Example of Prim's Algorithm



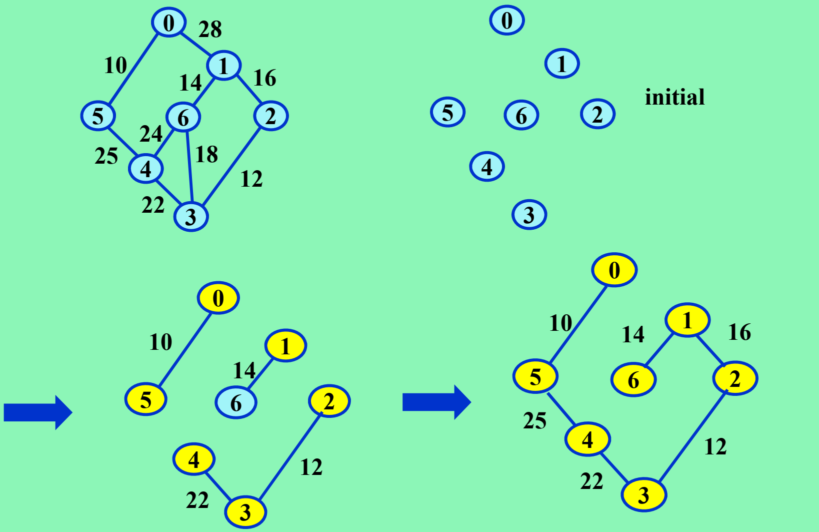
Total weight = 99

Prim's Algorithm

Notations:
(1) *TV* is the set of collected vertices in the spanning tree
(2) *T* is the set of collected edges in the spanning tree

```
// Assume that G has at least one vertex
TV = {0};  // Start with vertex 0 and no edges
for ( T= ∅; T contains fewer than n-1 edges;  add (u,v) to T )
{
    Let (u,v) be a least-cost edge such that u ∈ TV and v∉TV;
    if ( there is no such edge )  break;
    add v to TV;
}
if ( T contains fewer than n-1 edges)  {
      cout << "No spanning tree" << endl;
}
```

Stages in Sollin's Algorithm



Total weight = 99

國立清華大學 電機工程學系
EE2410 Data Structure

Chapter 6
Graph (Part II)

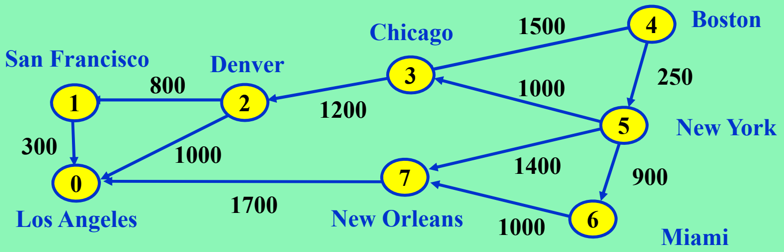
Outline

- ➡ **Shortest Path and Transitive Closure**
  - **Single Source / All Destinations**
  - **All-Pairs Shortest Paths**
  - **Transitive Closure**
- **Activity Network**
  - **Activity on Vertex (AOV) Networks**
  - **Activity on Edge (AOE) Networks**

Shortest Path Problem

- **Application**
  - Graph can be used to represent the **highway structure**
  - **Vertices** represent cities
  - **Edges** represent sections of highway
  - **Edge weight** is the distance of an edge
- **Questions**
  - (1) Is there a path from city A to city B?
  - (2) If there is more than one path from A to B, which is the shortest path?

Example: A Weighted Digraph

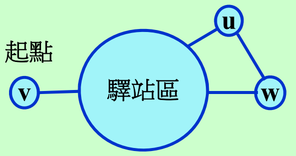


Length-adjacency matrix

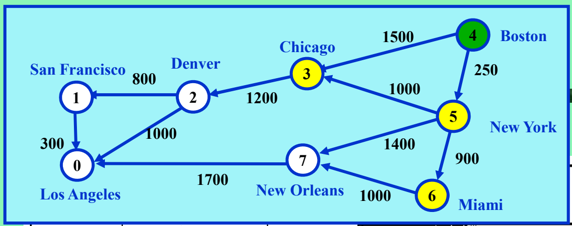
$$
\begin{array}{c|cccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
0 & 0 & & & & & & & \\
1 & 300 & 0 & & & & & & \\
2 & 1000 & 800 & 0 & & & & & \\
3 & & & 1200 & 0 & & & & \\
4 & & & & 1500 & 0 & 250 & & \\
5 & & & & 1000 & & 0 & 900 & 1400 \\
6 & & & & & & & 0 & 1000 \\
7 & 1700 & & & & & & & 0
\end{array}
$$

Edsger Dijkstra’s Algorithm

```
void  Graph::ShortestPath( const int n, const int v)
// dist[j], 0≦j<n, is set to the length of the shortest path from vertex v to vertex j
// in a graph G with n vertex and edge lengths given by length[i][j]
{
   for ( int  i=0; i<n; i++ ) {  s[i] = FALSE;  dist[i] = length[v][i]; }  // initialize
   s[v] = TRUE;
   dist[v] = 0;

    for ( i=0; i<n-2; i++) {
          int  u = choose(n);  // routine ‘choose’ returns a value u such that
                               // dist[u] = minimum dist[w], where s[w] = FALSE
          s[u] = TRUE;
          for ( int  w=0; w<n; w++) {
                 if ( ! s[w] )
                    if ( dist[u] + length[u][w] < dist[w] )
                         dist[w] = dist[u] + length[u][w];
          }
   }
}
```



驛站區由近而遠 , 從 0 個擴大成 n-1 個



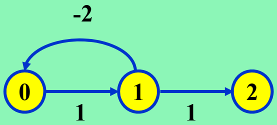
Example

Source is Boston

			Distance							
Iteration	*S*	Vertex	LA	SF	DEN	CHI	BOST	NY	MIA	NO
	en-route set	selected	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{4}	5	+∞	+∞	+∞	1250	0	250	1150	1650
2	{4,5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
3	{4,5,6}	3	+∞	+∞	2450	1250	0	250	1150	1650
4	{4,5,6,3}	7	3350	+∞	2450	1250	0	250	1150	1650
5	{4,5,6,3,7}	2	3350	3250	2450	1250	0	250	1150	1650
6	{4,5,6,3,7,2}	1	3350	3250	2450	1250	0	250	1150	1650
	{4,5,6,3,7,2,1}									

Digraph With Negative Edges

- When negative edge lengths are permitted
  - The digraph should have no cycle of negative length
  - This is to ensure that the shortest path consists of a finite number of edges
- Example
  - The shortest path from vertex 0 to 2 is -∞
    (0, 1, 0, 1, 0, 1, ..., 0, 1, 2)
  - Because there is a cycle (1, 0, 1) of length -1



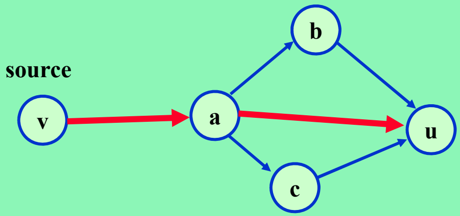
Possible Search Space

- Let $dist^k[u]$
  - be the length of a shortest path from the source vertex v to vertex u that contains at most *k* edges
- Then, $dist^1[u] = length[v][u], 0 \leqq u < n$
- Under the no-negative-cycle constraint
  - We can limit our search to shortest paths with at most n-1 edges
  - Hence, $dist^{n-1}[u]$ is the length of an unrestricted shortest path from v to u

Recurrence Relation

- **Given $dist^m[u]$ for every $0 \leqq u < n$**
- **How to compute $dist^{m+1}[u]$?**
- **Recurrence Relation**

$$dist^{m+1}[u] = \min\{\, dist^m[u],\ \min_i \{dist^m[i] + length[i][u]\}\, \}$$



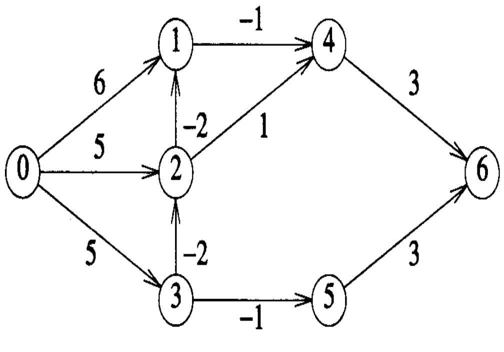
Assume that S-path^2[v][u] = (v, a, u)

What are the possible S-path^3[v][u] = { (v,a,u), (v,a,b,u), (v,a,c,u)}

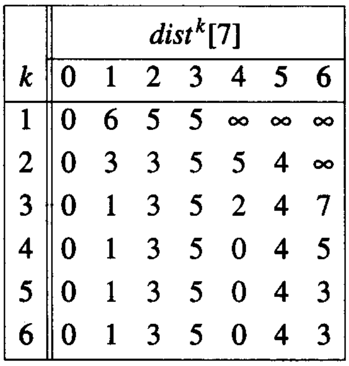
Example of Shortest Paths With Negative Edge Lengths

Source vertex: 0

Each shortest path consists of at most 6 edges



(a) A directed graph



k	$dist^k[7]$						
	0	1	2	3	4	5	6
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$

Bellman and Ford Algorithm
For computing Shortest Paths

```
void Graph::BellmanFord(const int n, const int v)
// Single source all destination shortest paths with negative edge lengths
{
  for (int i = 0; i < n; i++) dist [i ] = length [v ][i ]; // initialize dist
  for (int k = 2; k <= n –1; k++)
    for (each u such that u != v and u has at least one incoming edge)
      for (each <i, u> in the graph)
        if (dist [u ] > dist [i ] + length [i ][u ]) dist [u ] = dist [i ] + length [i ][u ];
}
```

Complexity:

$O(n^3)$ when adjacency matrix is used

$O(n \cdot e)$ when adjacency lists are used

Outline

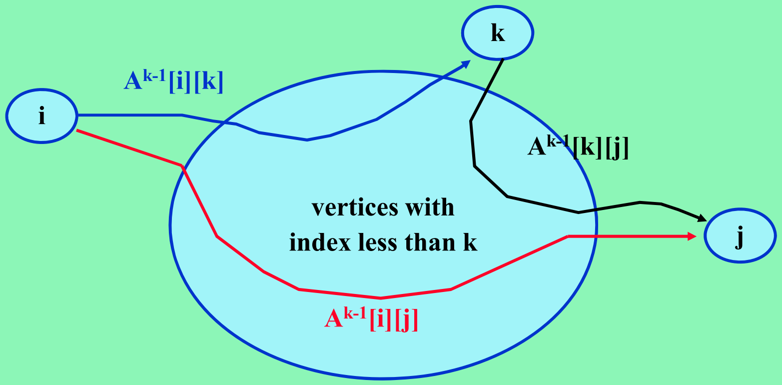
- **Shortest Path and Transitive Closure**
  - **Single Source / All Destinations**
  - **All-Pairs Shortest Paths**
  - **Transitive Closure**
- **Activity Network**
  - **Activity on Vertex (AOV) Networks**
  - **Activity on Edge (AOE) Networks**

Basics of All-Pairs Shortest-Paths

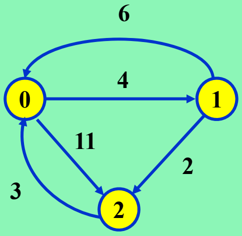
- Assume that
  - The digraph has n vertices with index of {0, ..., n-1}
- Let $A^k[i][j]$
  - be the length of the shortest path from i to j going through no intermediate vertex of index greater than k
- $A^{n-1}[i][j]$
  - will be the length of the shortest i-to-j path in G
- The basic idea in all-pair algorithm
  - is to successively generate the matrices $A^{-1}, A^0, \ldots, A^{n-1}$

Recurrence Relation

$A^k[i][j] = \min\{ A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j] \}, \; k \geqq 0$
where $A^{-1}[i][j] = \text{length}[i][j]$



Example for All-Pairs Shortest-Paths Problem



A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

A^{-1}

A^{0}	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

A^{0}

A^{1}	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

A^{1}

A^{2}	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

A^{2}

All-Pairs Shortest Paths

$A^k[i][j]$ = min{ $A^{k-1}[i][j]$, $A^{k-1}[i][k] + A^{k-1}[k][j]$ }, $k \geqq 0$
where $A^{-1}[i][j]$ = length[i][j]

Complexity:O(n^3)

```
void Graph::AllLengths(const int n)
// length [n ][n ] is the adjacency matrix of a graph with n vertices.
// a [i ][j ] is the length of the shortest path between i and j
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            a [i ][j ] = length [i ][j ];   // copy length into a
    for (int k = 0; k < n; k++)   // for a path with highest vertex index k
        for (i = 0; i < n; i++)   // for all possible pairs of vertices
            for (int j = 0; j < n; j++)
                if ((a [i ][k ] + a [k ][j ]) < a [i ][j ]) a [i ][j ] = a [i ][k ] + a [k ][j ];
}
```

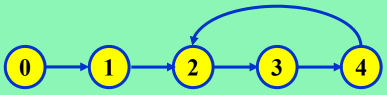
Outline

- **Shortest Path and Transitive Closure**
  - **Single Source / All Destinations**
  - **All-Pairs Shortest Paths**
  - **Transitive Closure**
- **Activity Network**
  - **Activity on Vertex (AOV) Networks**
  - **Activity on Edge (AOE) Networks**

Transitive Closure

- **Problem**
  - **Given a digraph with unweighted edges**
  - **We want to determine if there is a path from i to j for all values of i and j**
- **Transitive closure matrix of graph G, denoted as A^+**
  - **$A^+[i][j] = 1$ if there is a path of length >0 from i to j;**
  - **Otherwise, $A^+[i][j] = 0$;**
- **Reflexive transitive closure matrix, denoted as A^***
  - **$A^*[i][j] = 1$ if there is a path of length $\geqq 0$ from i to j;**
  - **Otherwise, $A^*[i][j] = 0$;**

Example of Transitive Closure Matrix



$$\begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 \\ 4 & 0 & 0 & 1 & 0 & 0 \end{array}$$

adjacency matrix

$$\begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 2 & 0 & 0 & 1 & 1 & 1 \\ 3 & 0 & 0 & 1 & 1 & 1 \\ 4 & 0 & 0 & 1 & 1 & 1 \end{array}$$

A^+

A[i][i]=1 when a cycle containing i exists

$$\begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 2 & 0 & 0 & 1 & 1 & 1 \\ 3 & 0 & 0 & 1 & 1 & 1 \\ 4 & 0 & 0 & 1 & 1 & 1 \end{array}$$

A^*

A[i][i]=1 is always 1

Finding Transitive Closure

- **For Directed Graph**
  - The transitive closure can be computed using the all-pairs shortest-path algorithm
  - But the recurrence relation is modified as follows:

    a[i][j] = a[i][j] || (a[i][k] && a[k][j]);
  - The final matrix obtained is A^+
- **For Undirected Graph**
  - Transitive closure can be found more easily through the identification of connected components, $O(n^2)$
  - For a vertex pair (i, j), $A^+[i][j]=1$ if vertices i and j are in the same connected component
  - $A^+[i][i] = 1$ iff the component containing i has at least two vertices

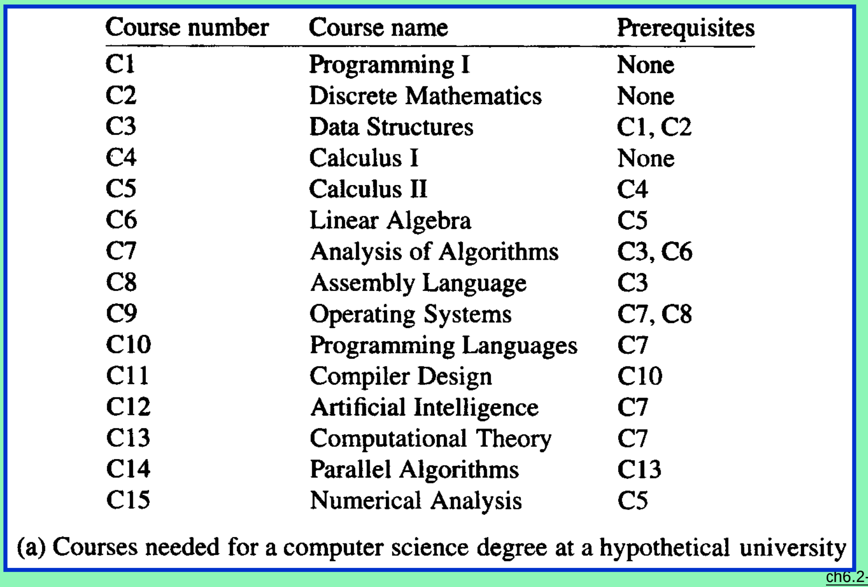
Outline

- Shortest Path and Transitive Closure
- Activity Network
  - Activity on Vertex (AOV) Networks
  - Activity on Edge (AOE) Networks

Activities-On-Vertex (AOV) Networks

- Activity
  - A project can be subdivided into several subprojects called activities
- Definition of AOV network
  - Activity-on-vertex network is a directed graph G
  - The vertices represent tasks or activities
  - The edges represent precedence relations between tasks

Activities For Completing a Degree

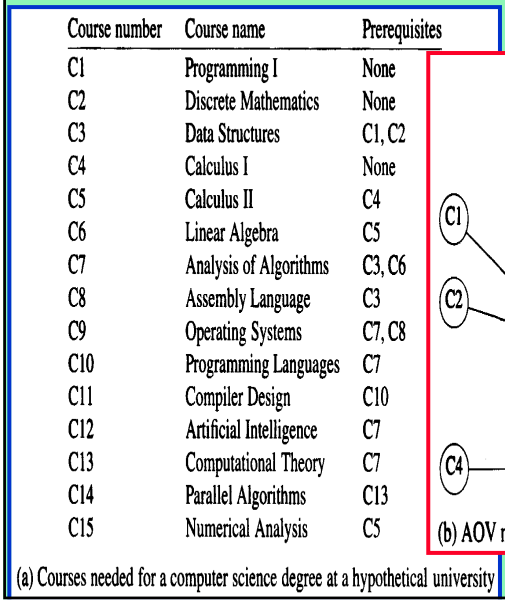


Course number	Course name	Prerequisites
C1	Programming I	None
C2	Discrete Mathematics	None
C3	Data Structures	C1, C2
C4	Calculus I	None
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Language	C3
C9	Operating Systems	C7, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithms	C13
C15	Numerical Analysis	C5

(a) Courses needed for a computer science degree at a hypothetical university

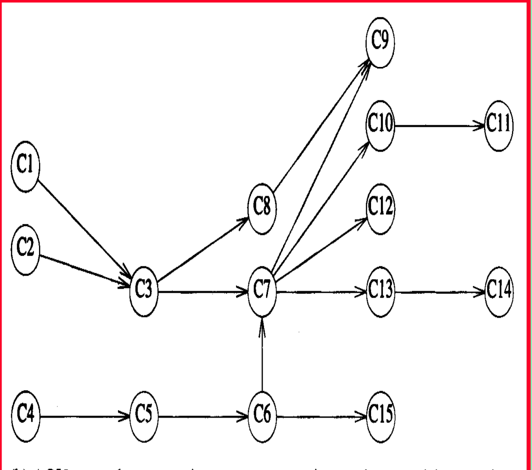
---

Example: AOV Network



Course number	Course name	Prerequisites
C1	Programming I	None
C2	Discrete Mathematics	None
C3	Data Structures	C1, C2
C4	Calculus I	None
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Language	C3
C9	Operating Systems	C7, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithms	C13
C15	Numerical Analysis	C5

(a) Courses needed for a computer science degree at a hypothetical university



(b) AOV network representing courses as vertices and prerequisites as edges

Terminology

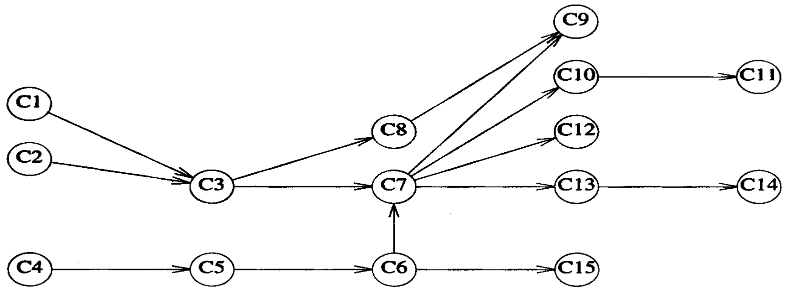
- **Predecessor**
  - Vertex i is a predecessor of vertex j and j is a successor of i iff there is a directed path from vertex i to vertex j
  - i is an immediate predecessor of j iff <i, j> is an edge
- **Transitive relation**
  - A relation · is transitive iff it is the case that for all triples i, j, k, i · j and j · k ➔ i · k
- **Partial order**
  - A precedence relation that is both transitive and irreflexive is a partial order

---

Topological Order

- **Topological order**
  - A topological order is a linear order of the vertices
  - For any two vertices i and j, if i is a predecessor of j in the network, then i precedes j in the linear ordering

Topological order: C1 C2 C4 C5 C3 C6 C8 C7 C10 C13 C12 C14 C15 C11 C9
C4 C5 C2 C1 C6 C3 C8 C15 C7 C9 C10 C11 C12 C13 C14

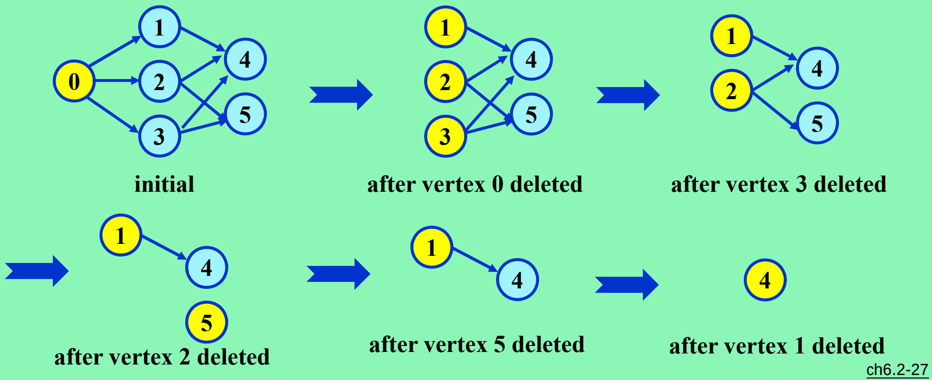


(b) AOV network representing courses as vertices and prerequisites as edges

A Topological Sorting Algorithm

```
Input the AOV network. Let n be the number of vertices
for ( int  i=0; i<n; i++){  //  output the vertices
{
    if ( every vertex has a predecessor )  return;  // network has a cycle
    pick a vertex v that has no predecessors;
    cout << v;
    delete v and all edges leading out of v from the network;
}
```

Complexity: O(e+n)



initial

after vertex 0 deleted

after vertex 3 deleted

after vertex 2 deleted

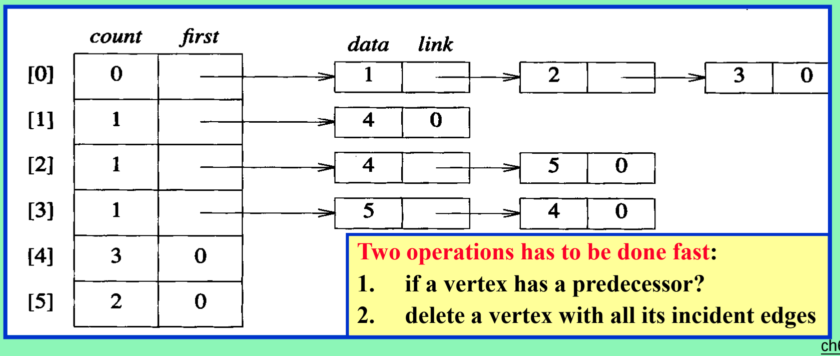
after vertex 5 deleted

after vertex 1 deleted

Internal Representations Used By Topological Sorting

```
class  Graph {
private:
   List<int>  *HeadNodes;
   int  *count;  // keep in-degree
   int   n;
```

```
public:
   Graph( const int  vertices=0) : n (vertices) {
   HeadNodes = new List<int>[n];
   count = new int[n];
   };
   void  TopologicalOrder();
};
```



Two operations has to be done fast:

1. if a vertex has a predecessor?
2. delete a vertex with all its incident edges

Topological Sorting Algorithm

```
void Graph::TopologicalOrder ()
// The n vertices of a network are listed in topological order
{
   int top = −1;
   for (int i = 0; i < n; i++) // create a linked stack of vertices with
      if (count [i ] == 0) { count [i ] = top; top = i;} // no predecessors

   for (i = 0; i < n; i++)
      if (top == −1) { cout << " network has a cycle" << endl; return;}
      else {
        int j = top; top = count [top ]; // unstack a vertex
        cout << j << endl;
        ListIterator<int> li (HeadNodes [j ]);
        if (! li.NotNull ()) continue;
        int k = *li.First ();
        while (1) { // decrease the count of the successor vertices of j
           count [k ]−−;
           if (count [k ] == 0) { count [k ] = top; top = k;} // add vertex k to stack
           if (li.NextNotNull ()) k = *li.Next (); // k is a successor of j
           else break;
        }
      } // end of else
}
```

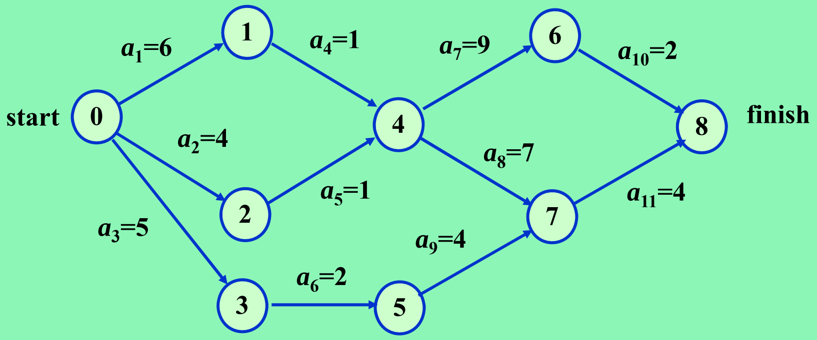
Complexity is: O(e+n)

Outline

- **Shortest Path and Transitive Closure**
- **Activity Network**
  - **Activity on Vertex (AOV) Networks**
  - **Activity on Edge (AOE) Networks**

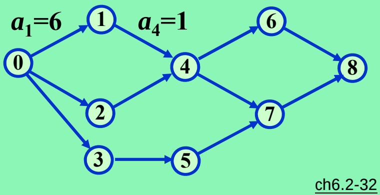
Activity-On-Edge (AOE) Network

- **AOE Network**
  - Is a **weighted directed graph** in which
  - The **vertices** represent **events**
  - The **edges** represent **activities or tasks**



Terminology

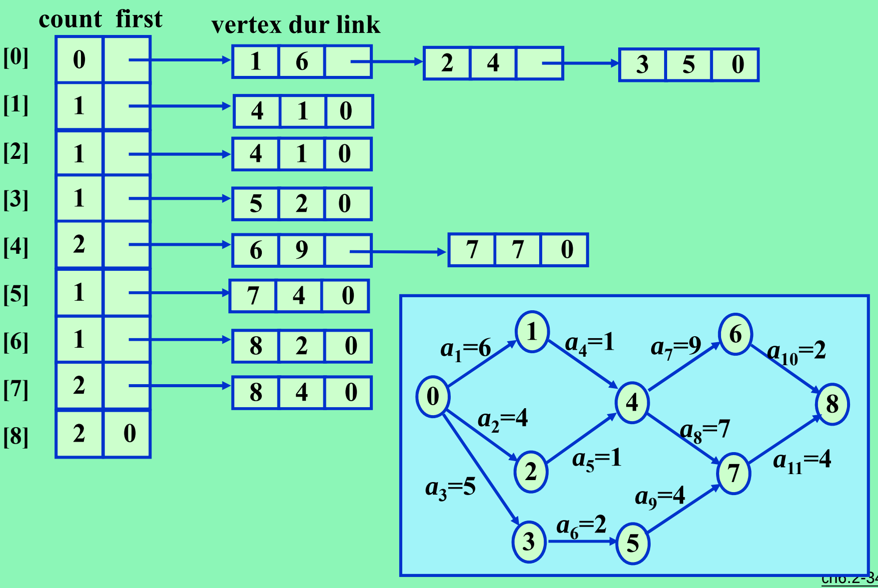
- **Critical Path**
  - Is the **longest path** from **start** vertex to **finish** vertex
  - determines the **minimum amount of time** to finish the project
- **Earliest Time of an activity a_i, denoted as $e(a_i)$**
  - is the **length of the longest path from start to the source vertex of a_i**
- **Latest Time of an activity a_i, denoted as $l(a_i)$**
  - indicates latest time an activity may start **without increasing the project duration**
- **Critical activity**
  - is an activity for which $e(a_i) = l(a_i)$



Critical Path Analysis

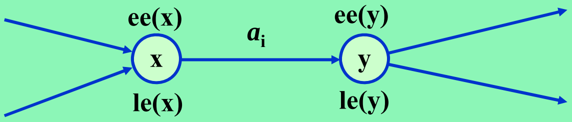
- **Purpose of critical path analysis**
  - **is to identify the critical activities so that resources may be concentrated on these activities in an attempt to reduce project finish time**
  - **That is, it is useful to identify project bottlenecks**
- **Finding all critical paths**
  - **(1) compute every activity's $e(a_i)$ and $l(a_i)$**
  - **(2) identify critical activities, i.e., a_i for which $e(a_i)=l(a_i)$**
  - **(3) remove all non-critical activities**
  - **(4) Generate all paths from start to finish**

Example: Data Representation



Overall Procedure of Critical Path Analysis

- **Procedure**
  - **Step 1: compute the earliest event time for each vertex i, denoted as ee(i)**
  - **Step 2: compute the latest event time for each vertex i, denoted as le(i)**
  - **Step 3: compute the earliest time for an activity a_i , <x, y>, using formula: $e(a_i) = ee(x)$**
  - **Step 4: compute the latest time for an activity a_i , <x, y>, using formula $l(a_i) = le(y) –$ duration of activity a_i**



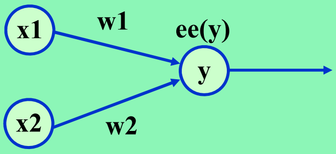
Calculation of Earliest Activity Times

- **The earliest event time of each vertex**
  - **can be computed by a forward stage**
- **Step 1: Sort vertices in the topological order**
- **Step 2: Evaluate earliest event time of each vertex by**

$$ee(y) = \max_{x_i \in P(y)} \{ ee(x_i) + \text{duration of } <x_i, y> \}$$

P(y) is the set of y's immediate predecessors

- **Step 3: $e(a_i) = ee($ source vertex of $a_i)$**



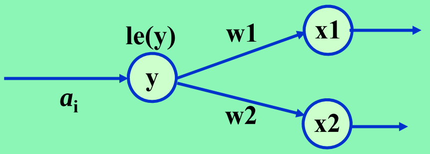
Calculation of Latest Activity Times

- **The latest event time of each vertex**
  - **can be computed by a backward stage**
- **Step 1: Sort vertices in reverse topological order**
- **Step 2: Evaluate latest event time of each vertex by**

$$le(y) = \min_{x_i \in S(y)} \{ le(x_i) - \text{duration of } \langle y, x_i \rangle \}$$

S(y) is the set of y's immediate successors

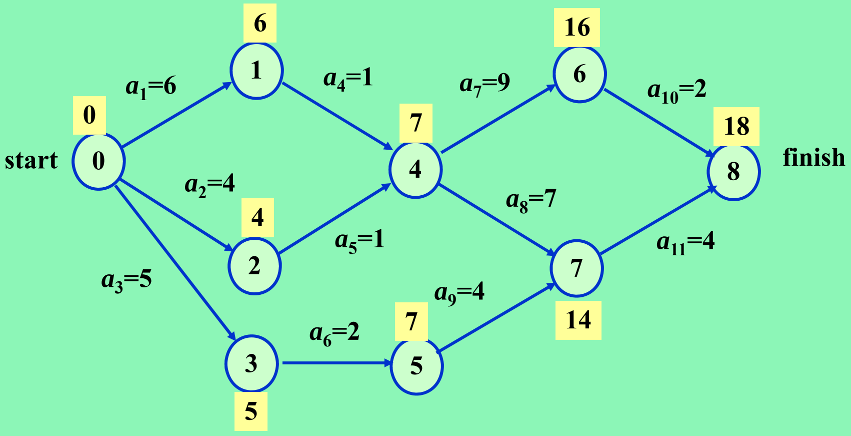
- **Step 3:** $l(a_i) = le(\text{destination vertex of } a_i) -$ **duration of activity** a_i



Example of Computing Earliest Event Times

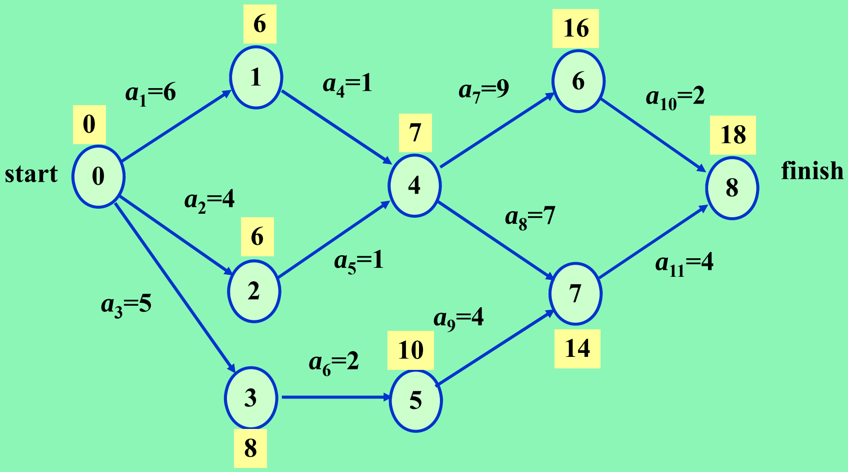
topological order: 0, 1, 2, 3, 4, 5, 6, 7, 8

$e(a_i) = ee(\text{source vertex of } a_i)$



Example of Computing Latest Event Times

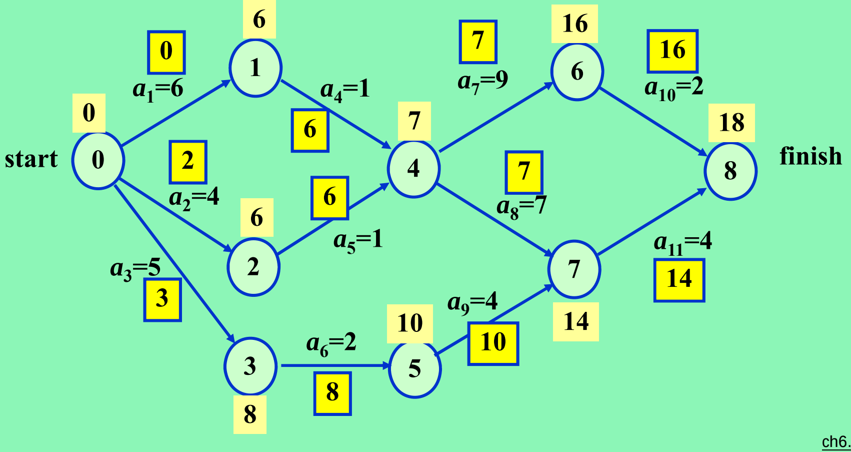
reverse topological order: 8, 7, 6, 5, 4, 3, 2, 1, 0



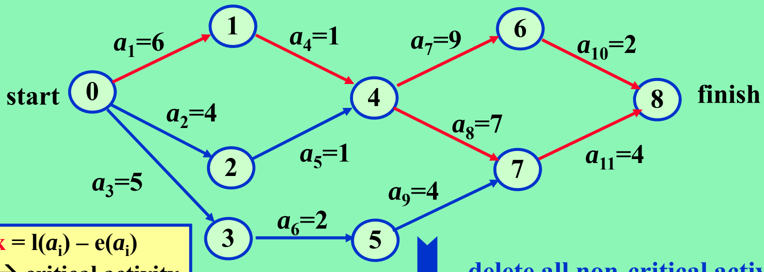
Example of Computing Latest Activity Times

reverse topological order: 8, 7, 6, 5, 4, 3, 2, 1, 0

l(a_i) = le(destination vertex of a_i) – duration of activity a_i

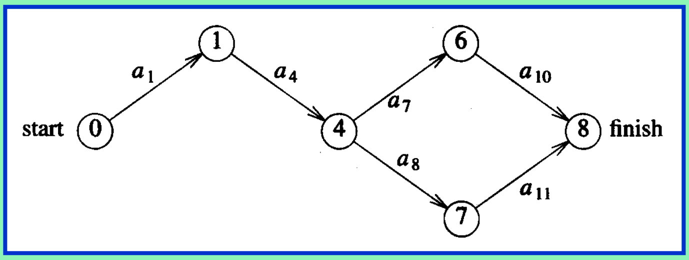


Graph of Critical Paths



slack = l(a_i) – e(a_i)
slack=0 → critical activity

delete all non-critical activities



國立清華大學 電機工程學系
EE2410 Data Structure

Chapter 7 Sorting

Outline

- Insertion Sort
- Quick Sort
- How Fast Can We Sort ?
- Merge Sort
- Heap Sort
- Sorting On Several Keys
- List and Table Sorts

Basics

- **Terminology**
  - **List → a collection of records having one or more fields**
  - **Key → the field used to distinguish among the records**

關鍵值

- **Example: telephone directory**
  - **a record has three fields: name, address, phone number**
  - **key is usually a person's name**
  - **key could also be the phone number**

```
class  Element
{
public
    int   getKey() const { return key; }
    void  setKey(int k) { key = k; }
private:
    int key;  // other fields not shown here
};
```

每一筆資料以一個或
多個關鍵值做為索引標準
其他資料省略未列出

Sequential Search

```
int  SeqSearch (Element  *f, const int n, const int k)
// Search a list f with key values f[1].key, ..., f[n].key
// Return i such that f[i].key == k. If there is no such record, return 0
{
   int   i = n;  // set pointer to the last element initially
   f[0].setKey(k); // set the key of the 0th-element to k
   while ( f[i].getKey() != k )  i--;  // sequential search from n to 1
   return i;
}
```

Time complexity is O(n)

Trick: introduce a dummy record 0 with f[0].key = k

→ simplifies the body in while loop

→ End-of-list test is avoided

→ can achieve significant speedup when n is large

Sorting Problem

- **Given**
  - A list of records (R_1, R_2,..., R_n)
  - Each record, R_i, has key value K_i.
  - We assume an ordering relation (<) on the keys

    (1) for any two key values x and y, x=y, or x<y, or x>y

    (2) the relation is transitive (i.e., x<y, y<z, then x<z)
- **Definition of a stable sorting method**
  - The sorting problem is to find a permutation σ, such that $K_{\sigma(i)} \leqq K_{\sigma(i+1)}$ $1 \leqq i \leqq n-1$
  - If i<j and K_i==K_j in the input list, then R_i precedes R_j in the sorted list

---

Example: Internal Revenue Service

- **Two lists of tax forms that come in at random**
  - One from the employee with m forms → F1
  - One from the employers with n forms → F2
  - Keys are the social security numbers
- **Problems**
  - To make sure that the two lists are consistent
- **Complexity comparison**
  - direct search: O(mn)
  - sort each list and then compare: $O(t_{sort}(n) + t_{sort}(m) + m + n)$

    where $t_{sort}(n)$ is the time to sort n records → $O(n \cdot \log n)$

    Therefore, the total time is $O(\max\{ n \cdot \log n, m \cdot \log m \})$

Sorting Methods Classification

- **Internal Sorting**
  - Methods to be used when the list to be sorted is small enough so that the entire sorting can be carried out in main memory
  - inserting sort, quick sort, merge sort, heap sort, and radix sort
- **External Sorting**
  - Methods to be used on larger lists

Insertion Sort

Initial list of records: R_0 R_1 . . . R_i ($K_1 \leqq K_2 \leqq ... \leqq K_i$)

```
int  insert(const  Element e, Element *list, int i)
// Insert element e with key e.key into the ordered sequence list
// list[0], ..., list[i], such that the resulting sequence is also ordered.
// Assume that e.key ≧list[0].key
// The array list must have space allocated for at least i+2 elements
{
   while ( e.getKey( ) < list[i].getKey( ) ) {
         list[i+1] = list[i];    i--;
   }
   list[i+1] = e;
}
```

The R_0 is an artificial record with key K_o=MININT (i.e., all keys are $\geqq K_0$)

```
int  InsertionSort( Element *list, const int n)
// Sort list in nondecreasing order of key
{
   list[0].setKey(MININT);
   for ( int  j=2; j<=n; j++){
         insert( list[j], list, j-1);
}
```

Example of Inserting Sort

Example:

	j	[1]	[2]	[3]	[4]	[5]
	-	5	4	3	2	1
list after insertion of 4	2	4	5	3	2	1
list after insertion of 3	3	3	4	5	2	1
list after insertion of 2	4	2	3	4	5	1
list after insertion of 1	5	1	2	3	4	5

- **Complexity**
  - $O(1+2+3+ \cdots +n-1) = O(n^2)$
- **Relative disorder in the input list**
  - A record R_i is left out of order (LOO) iff $R_i < \max_{1 \leqq j \leqq i} \{R_j\}$
  - Let k be the number of LOO records, the computing time is $O(k \cdot n)$

Variations

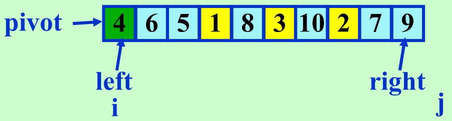
- **Binary Search Sort**
  - The number of comparisons made is reduced
  - But the complexity remains unchanged, because the number of records moved is not changed
- **List Insertion Sort**
  - The elements of the list are represented as a linked list rather than an array
  - No record movement
  - However, the search is still sequential

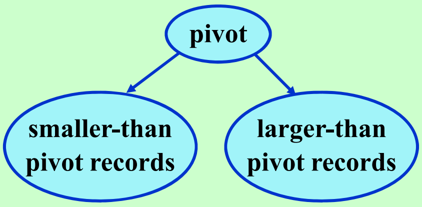
Outline

- **Insertion Sort**
- **Quick Sort**
- **How Fast Can We Sort ?**
- **Merge Sort**
- **Heap Sort**
- **Sorting On Several Keys**
- **List and Table Sort**

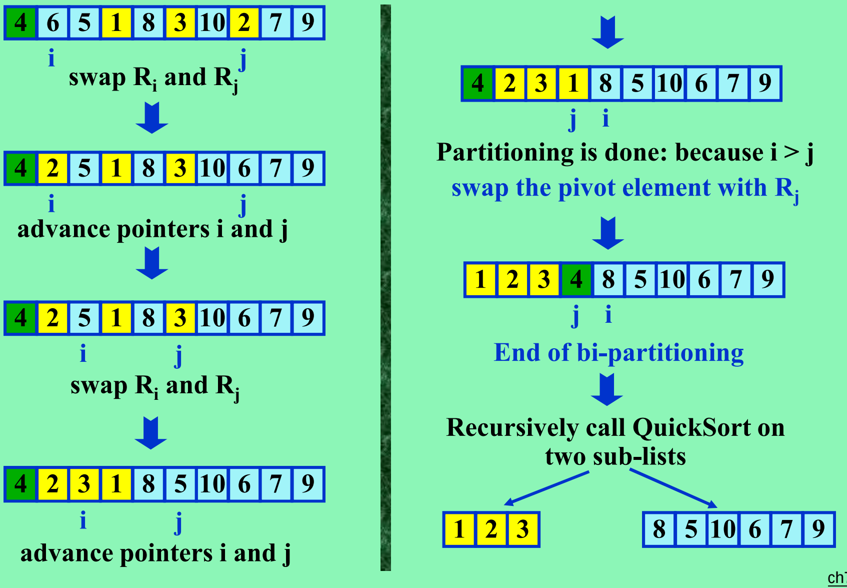
Quick Sort

```
void  QuickSort( Element *list, const int left, const int right)
// Sort records list[left], · · ·, list[right] into nondecreasing order on field key
// Key pivot = list[left].key
{
   if ( left < right ) {
        int   i = left,
              j = right + 1,
              pivot = list[left].getKey();
        do {
              do i++; while ( list[i].getKey() < pivot );  // forward search for larger records
              do j--; while ( list[j].getKey() > pivot); // backward search for smaller records
              if ( i < j )  InterChange( list, i, j );
        } while ( i < j );
        InterChange(list, left, j);

        QuickSort( list, left, j-1);
        QuickSort( list, j+1, right);
   }
}
```





Example Of Quick Sort



4 6 5 1 8 3 10 2 7 9

i j

swap R_i and R_j

4 2 5 1 8 3 10 6 7 9

i j

advance pointers i and j

4 2 5 1 8 3 10 6 7 9

i j

swap R_i and R_j

4 2 3 1 8 5 10 6 7 9

i j

advance pointers i and j

4 2 3 1 8 5 10 6 7 9

j i

Partitioning is done: because i > j

swap the pivot element with R_j

1 2 3 4 8 5 10 6 7 9

j i

End of bi-partitioning

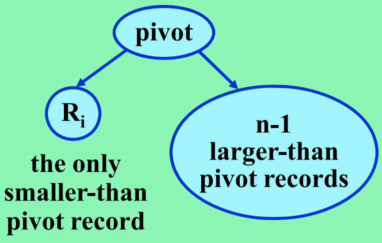
Recursively call QuickSort on two sub-lists

1 2 3

8 5 10 6 7 9

How Fast is Quick Sort ?

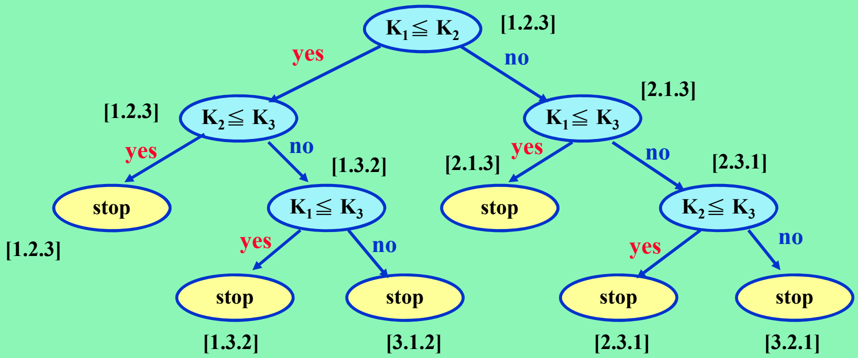
- Average case:
  - $T(n) = cn + 2T(n/2)$
  - $T(n) = O(n \cdot \text{long } n)$
- Worst case
  - $T(n) = cn + T(n-1)$
  - $T(n) = O(n^2)$
- Variation
  - A better pivot selection: pivot= median $\{K_l, K_{(l+r)/2}, K_r\}$
- It has been observed that
  - quick sort is the fastest sorting algorithm on average



worst-case of quick sort

How Fast Can We Sort?

- **The Sorting Algorithm**
  - **The best possible time is O(n · log n)**
  - **That is, the worst-case computing time Ω(n · log n)**



The height of this decision tree:

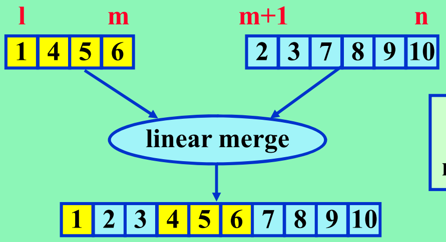
$$\log_2(n!) + 1 \geqq \log_2(\,(n/2)^{(n/2)}\,) = \Omega(\,n \cdot \log n)$$

Outline

- **Insertion Sort**
- **Quick Sort**
- **How Fast Can We Sort ?**
- ➡ **Merge Sort**
- **Heap Sort**
- **Sorting On Several Keys**
- **List and Table Sorts**

Merging Of Two Sorted Lists

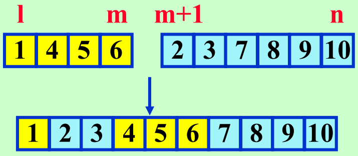
- **Given two sorted lists**
  - List 1: (initList$_l$, · · · , initList$_m$)
  - List 2: (initList$_{m+1}$, · · · , initList$_n$)
- **Output the merged list of the two lists**
  - merged list: (mergedList$_l$, · · · , mergedList$_n$)



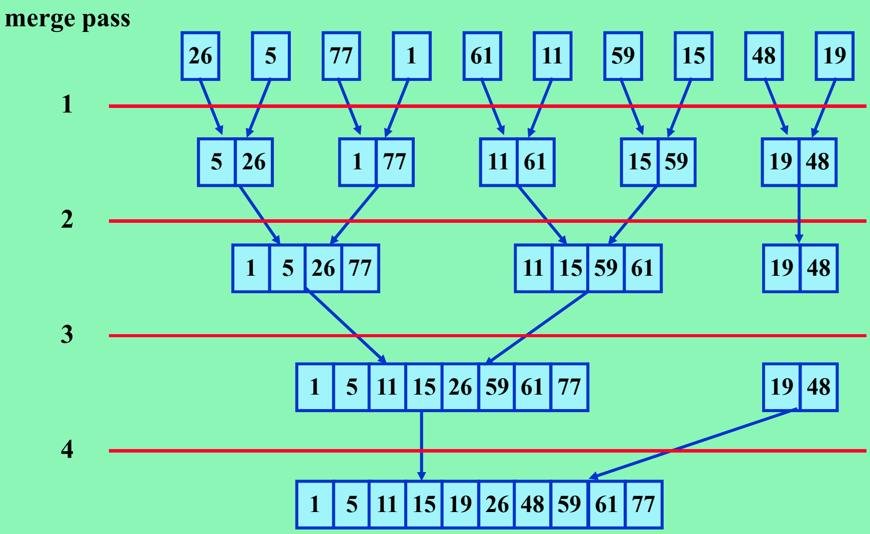
Time = O(n)
Space = O(n)
n is the total record number

Algorithm of Merging Two Lists

```
void  merge (Element *initList, Element *mergedList,
                    const int l, const int m, const int n)
{
   for ( int i1 = l, iResult = l, i2 = m+1; // i1, i2, and iResult are positions
          i1 <= m && i2 <=n; // both input lists not yet exhausted
          iResult++) {
         if ( initList[i1].getKey() <= initList[i2].getKey() ) {
               mergedList[iResult] = initList[i1];
               i1++;
         }
         else {
               mergedList[iResult] = initList[i2];
               i2++;
         }
   }
   if (i1 > m)   for ( t=i2; t<=n; t++) { mergedList[iResult+t-i2] = initList[t]; }
   else            for ( t=i1; t<=m; t++) { mergedList[iResult+t-i1] = initList[t]; }
}
```

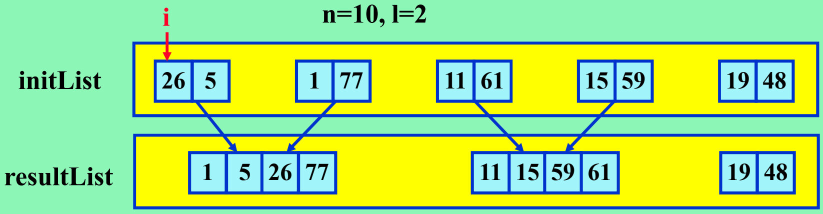


Iterative Merge Tree



MergePass Algorithm

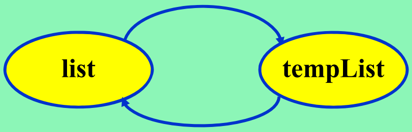
```
void  MergePass(Element *initList, Element *resultList, const int n, const int l)
// One pass of merge sort. Adjacent pairs of sublists of length l are merged
// from list initList to list resultList. n is the number of records in initList
{
   for (int  i=1; // i is the first position in the first of the two sublists being merged
          i <= n-2*l + 1;  // Are enough elements left to form two sublists of length l?
          i = i + 2*l ) {
              merge ( initList, resultList, i, i+l-1, i+2*l-1);
   }
   // merge remaining list of length < 2*l
   if ( ( i+l-1) < n)  merge ( initList, resultList, i, i+l-1, n);  // two sublists
   else  for ( int t=i; t<=n; t++)  resultList[t] = initList[t];  // one sublist
}
```



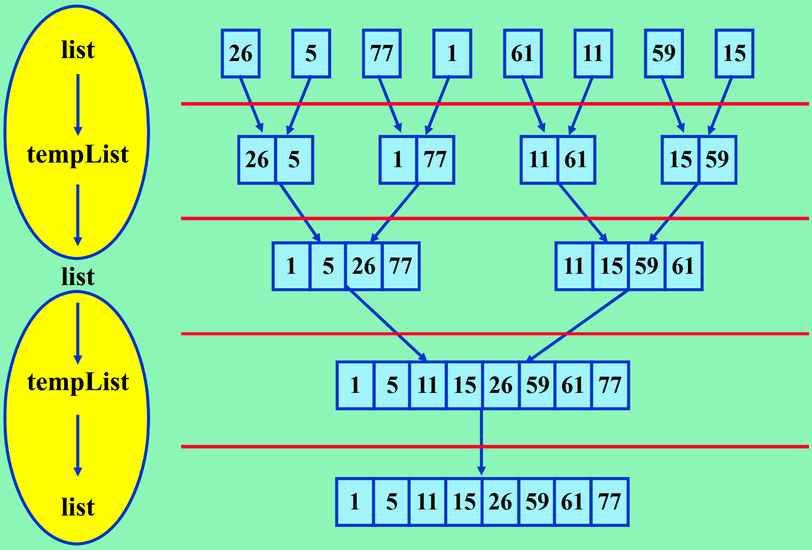
Merge Sort Algorithm

```
void  MergeSort( Element *list, const int n)
// Sort a list into nondecreasing order of the keys list[1].key, ..., list[n].key
{
   Element  *tempList = new Element[n+1];
   //  l is the length of the sublist currently being merged
   for ( int  l = 1; l < n;  l = l *2 )
   {
        MergePass ( list, tempList, n, l);
        l = l * 2;
        MergePass ( tempList, list, n, l);  // interchange role of list and tempList
   }
   delete  [ ]  tempList;
}
```

list and tempList hold the partially sorted list alternatively



Two Alternating Lists

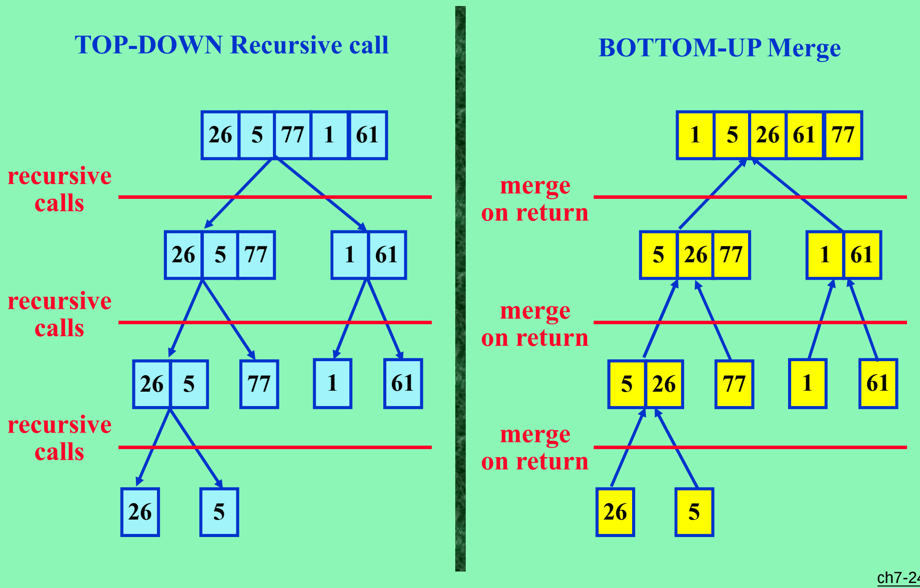


Recursive Merge Sort Algorithm

```
class  Element
{
private:
   int  key;  Field other;  int  link;
public:  Element() { link = 0; }
};
```

```
int  rMergeSort ( Element *list, const int left, const int right )
// List = ( list[left], ..., list[right] ) is to be sorted on the field key
// link is a field in each record that is initially 0
// rMergeSort returns the index of the first element in the sorted chain
// list[0] is a record for intermediate results used only in ListMerge
{
   if ( left >= right) return left;
   int  mid  = (left + right) /2;
   return ListMerge( list,
                         rMergeSort(list, left, mid), // sort left half
                         rMergeSort(list, mid+1, right); // sort right half
}
```

Execution of Recursive Merge Sort

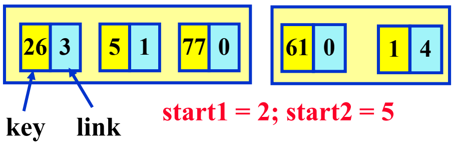


List Merge Algorithm

```
int ListMerge(Element *list, const int start1, const int start2)
// The sorted linked lists whose first elements are indexed by start1 and start2,
// respectively, are merged to obtain the sorted linked list. The index of the first element of the
// sorted list is returned. Integer links are used.
{
  int iResult = 0;
  for (int i1 = start1, i2 = start2; i1 && i2; )
    if (list [i1] . key <= list [i2] . key) {
      list [iResult ] . link = i1;
      iResult = i1; i1 = list [i1] . link;
    }
    else {
      list [iResult ] . link = i2;
      iResult = i2; i2 = list [i2] . link;
    }

  // move remainder
  if (i1 == 0) list [iResult ] . link = i2;
  else list [iResult ] . link = i1;
  return list [0] . link;
}
```

list[0] is the header element

Array index:
1 2 3 4 5



start1 = 2; start2 = 5

Outline

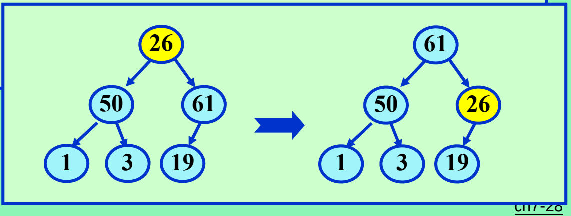
- **Insertion Sort**
- **Quick Sort**
- **How Fast Can We Sort ?**
- **Merge Sort**
- **Heap Sort**
- **Sorting On Several Keys**
- **List and Table Sorts**

Basics of Heap Sort

- **Procedure**
  - **Step 1: build the given list as a max heap**
  - **Step 2: extract one record at a time from the heap**
- **Time Complexity**
  - **worst case: O(n · log n)**
  - **average case: O(n · log n)**
- **Space Complexity**
  - **only a fixed amount of additional storage is needed: O(1)**
- **Heap Sort is not stable**

Adjust Routine

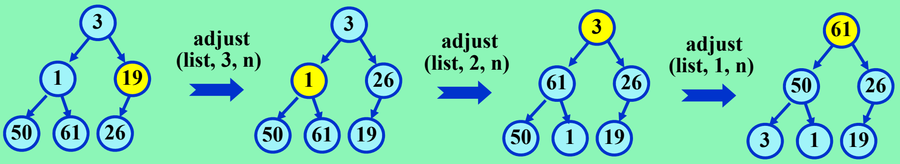
```
void  adjust ( Element *tree, const  int  root, const  int n)
// Adjust the binary tree with root to satisfy the heap property. The left and right
// subtrees of root already satisfy the heap property.
// No node has index greater than n
{
   Element  e = tree[root];
   int  k = e.getKey();
   for ( int j=2*root; j<=n; j*=2)
   {  // first find max of left and right child
        if ( j < n )
             if ( tree[j].getKey() < tree[j+1].getKey() )  j++;  // select larger child to j
        // compare max child with k. If k is max, then done
        if ( k >= tree[j].getKey() )  break;
        tree[j/2] = tree[j];  // move jth record up the tree
    }
    tree[j/2] = e;
}
```



Heap Sort

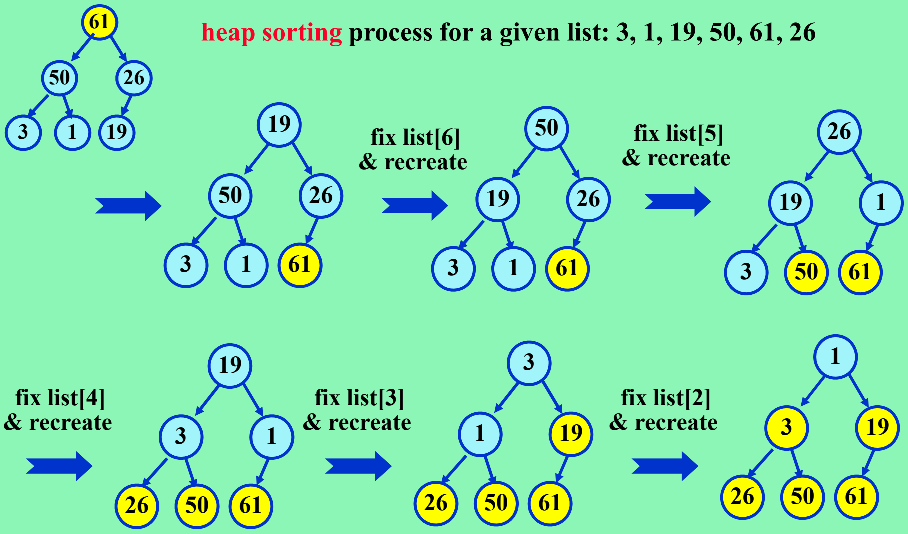
```
void  HeapSort ( Element *list, const int n)
//  The list = (list[1], ..., list[n]) is sorted into nondecreasing order of the field key
{
   for ( int  i = n/2 ;  i>=1; i--) // convert list into a heap
         adjust ( list, i, n);
   for ( i = n-1;  i>=1;  i--) // sort list
   {
        Element t = list[i+1]; // interchange list_1 and list_{i+1}
        list[i+1] = list[1];
        list[1] = t;
        adjust(list, 1, i);  // recreate the heap
   }
}
```

heap construction process for a given list: 3, 1, 19, 50, 61, 26



Step 2 of Heap Sorting

heap sorting process for a given list: 3, 1, 19, 50, 61, 26



Outline

- Insertion Sort
- Quick Sort
- How Fast Can We Sort ?
- Merge Sort
- Heap Sort
- ➡ Sorting On Several Keys
- List and Table Sorts

Basics of Sorting On Several Keys

- **Terminology**
  - **Keys: $K^1, K^2, \ldots, K^r$**
  - **K^1 is the most significant key**
  - **K^r is the least significant key**
- **Comparison of multiple key**
  - **The r-tuple $(x^1, x^2, \ldots, x^r)$ is less than or equal to the r-tuple $(y^1, y^2, \ldots, y^r)$ iff either one of the following two conditions is satisfied**

    (1) $x^i = y^i$ for $1 \leqq i \leqq r$,

    (2) $x^i = y^i$ for $1 \leqq i < \alpha$, and $x^\alpha < y^\alpha$ for some $1 \leqq \alpha \leqq r$

E.g., $(1, 2, \underset{\uparrow}{3}) < (1, 2, \underset{\uparrow}{5}) \rightarrow \alpha = 3$

Most Significant Digit First Sorting

- **Example**
  - Sorting a deck of cards
  - The first key K^1: suit (spade, heart, diamond, club)
  - The second key K^2: face value (2, 3, ..., J, Q, K, A)
- **Most Significant Digit (MSD) First Sorting**
  - Sort the cards into 4 piles using K^1, one for each suit
  - Sort each of the 4 piles using K^2
  - Cascade the sorted 4 piles with the order of (spade, heart, diamond, club)

Least Significant Key Sorting

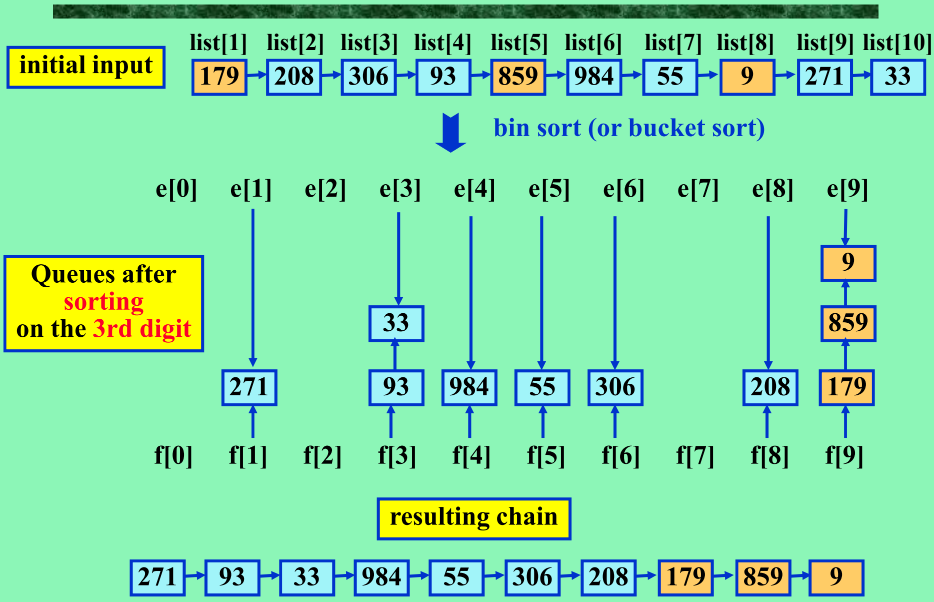
- **Example**
  - Sorting a deck of cards
  - The first key K^1: suit (spade, heart, diamond, club)
  - The second key K^2: face value (2, 3, ..., J, Q, K, A)
- **Least Significant Digit (LSD) First Sorting**
  - Sort the cards into 13 piles using K^2
  - Then, cascade the 13 piles into a big pile with the order of 2, 3, 4, ..., J, Q, K, A
  - Finally, sort the big pile using a stable sorting algorithm

LSD Radix Sort

```
void RadixSort(Element *list, const int d, const int n)
// Records list = (list [1],  · · · , list [n ]) are sorted on the keys key [0],  · · · , key [d−1].
// The range of each key is 0 ≤ key [i ] < radix.  radix is a constant.
// Sorting within a key is done using a bin sort.
{
   int e [radix ], f [radix ]; // queue pointers
   for (int i = 1; i <= n; i++) list [i ] . link = i + 1; // link into a chain starting at current
   list [n ] . link = 0; int current = 1;
   for (i = d − 1; i >= 0; i −−) // sort on key key [i ]
   {
      for (int j = 0; j < radix; j ++) f [j ] = 0; // initialize bins to empty queues
      for ( ; current; current = list [current ] . link) { // put records into queues
         int k = list [current ] . key [i ];
         if (f [k ] == 0) f [k ] = current;
         else list [e [k ]] . link = current;
         e [k ] = current;
      }
      for (j = 0; f [j ] == 0; j ++); // find first nonempty queue
      current = f [j ]; int last = e [j ];
      for (int k = j + 1; k < radix; k ++) // concatenate remaining queues
         if (f [k ]) {
            list [last ] . link = f [k ];
            last = e [k ];
         }
      list [last ] . link = 0;
   } // end of for (i = d − 1; i >= 0; i −−)
}
```

Complexity = O(d · n)

Radix Sort Example (I)



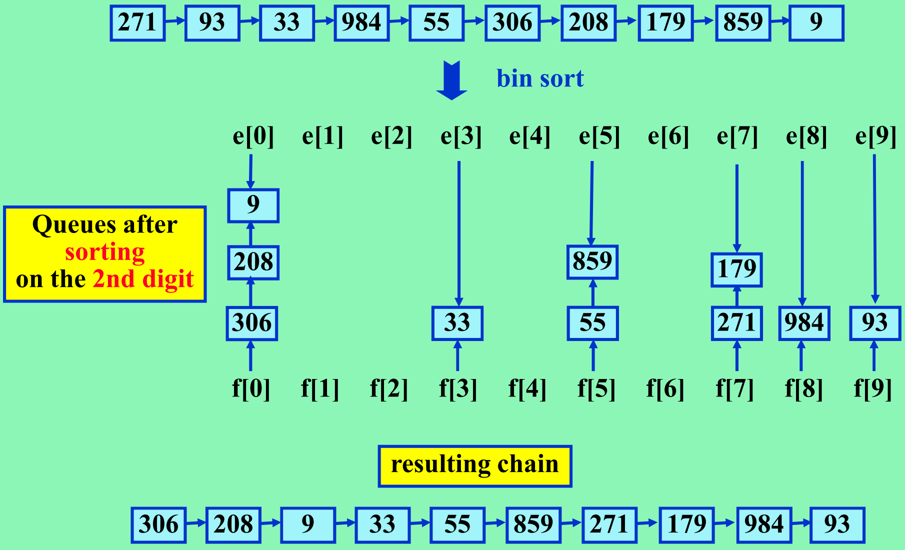
initial input

bin sort (or bucket sort)

Queues after sorting on the 3rd digit

resulting chain

Radix Sort Example (II)



271 → 93 → 33 → 984 → 55 → 306 → 208 → 179 → 859 → 9

bin sort

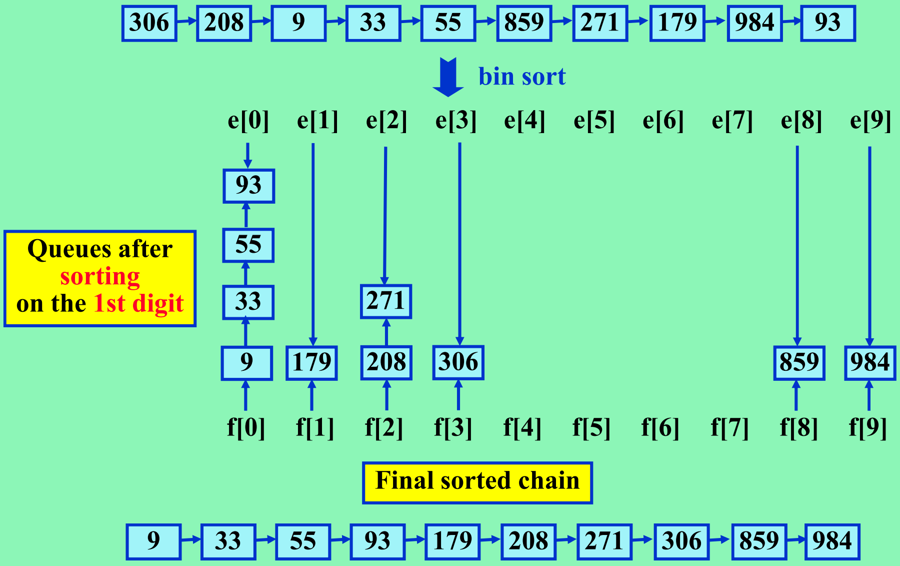
Queues after sorting on the 2nd digit

e[0]	e[1]	e[2]	e[3]	e[4]	e[5]	e[6]	e[7]	e[8]	e[9]
9									
208					859		179		
306			33		55		271	984	93
f[0]	f[1]	f[2]	f[3]	f[4]	f[5]	f[6]	f[7]	f[8]	f[9]

resulting chain

306 → 208 → 9 → 33 → 55 → 859 → 271 → 179 → 984 → 93

Radix Sort Example (III)



306 → 208 → 9 → 33 → 55 → 859 → 271 → 179 → 984 → 93

bin sort

Queues after sorting on the 1st digit

e[0]	e[1]	e[2]	e[3]	e[4]	e[5]	e[6]	e[7]	e[8]	e[9]
93									
55									
33		271							
9	179	208	306					859	984
f[0]	f[1]	f[2]	f[3]	f[4]	f[5]	f[6]	f[7]	f[8]	f[9]

Final sorted chain

9 → 33 → 55 → 93 → 179 → 208 → 271 → 306 → 859 → 984

Performance Consideration

- **Excessive Data Movement**
  - tends to slow down the sorting process
- **Avoid data movement as much as possible**
  - Modifications can be made to insertion sort or merge sort
- **In-place post-sorting rearrangement**
  - may be needed to convert a sorted list to an array

名次與陣列註標一致

sorted list (head → R4)

陣列註標	i	R1	R2	R3	R4
關鍵值	key	26	5	77	1
串列鏈結	link	3	1	0	2

sorted array

i	R1	R2	R3	R4
key	1	5	26	77
link	2	1	3	0

Example: In-Place Rearrangement

rank	6	2	10	1	9	3	8	4	7	5
i	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
key	26	5	77	1	61	11	59	15	48	19
link	9	6	0	2	3	8	5	10	7	1
linkb	10	4	5	0	7	2	9	6	1	8

first

fix the position of R4

to be changed

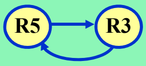
rank	6	2	10	1	9	3	8	4	7	5
i	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
key	1	5	77	26	61	11	59	15	48	19
link	2	6	0	9	3	8	5	10	7	4
linkb	0	4	5	10	7	2	9	6	4	8

first

{ R2, R3, ..., R10 } still forms a sorted list

Example: In-Place Rearrangement

rank	6	2	10	1	9	3	8	4	7	5
i	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
key	1	5	77	26	61	11	59	15	48	19
link	2	6	0	9	3	8	5	10	7	4
linkb	0	4	5	10	7	2	9	6	4	8
						first				

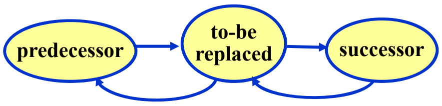


fix the position of R6

rank	6	2	10	1	9	3	8	4	7	5
i	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
key	1	5	11	26	61	77	59	15	48	19
link	2	6	8	9	6	0	5	10	7	4
linkb	0	4	2	10	7	5	9	6	4	8
								first		

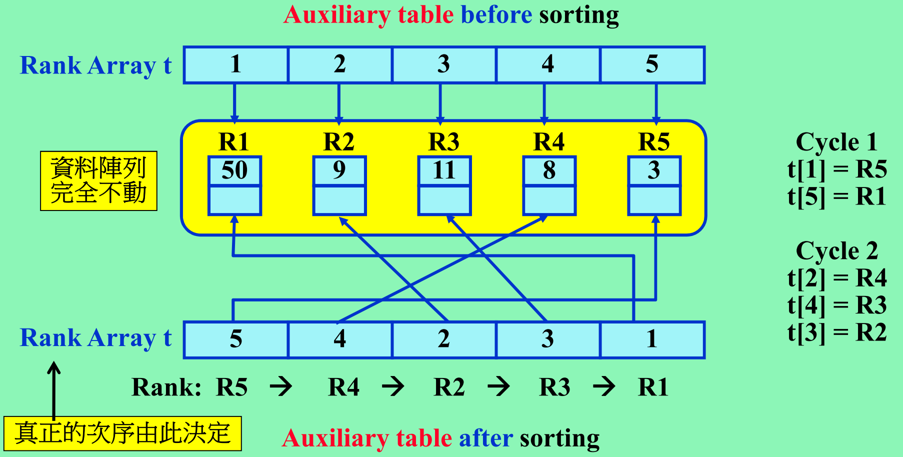
List2Array Algorithm

```
void list1(Element *list, const int n, int first)
// Rearrange the sorted chain first so that the records list[1], ···, list[n]
// are in sorted order. Each record has an additional link field linkb.
{
    int prev = 0;
    for (int current = first; current; current = list[current].link)
    // convert chain into a doubly linked list
    {
        list[current].linkb = prev;
        prev = current;
    }

    for (int i = 1; i < n ; i ++) // move list_first to position i while
    {                             // maintaining the list
        if (first != i) {
            if (list[i].link) list[list[i].link].linkb = first;
            list[list[i].linkb].link = first;
            Element a = list[first]; list[first] = list[i]; list[i] = a;
        }
        first = list[i].link;
    }
}
```



List-Based Array for Quick-Sort - To avoid data movement

We can use the following rank array for providing *array_based list* for Quick-Sort or Heap Sort ➔ The i-th record is R[t[i]].



Auxiliary table before sorting

Rank Array t

1	2	3	4	5

R1	R2	R3	R4	R5
50	9	11	8	3

資料陣列
完全不動

Rank Array t

5	4	2	3	1

Rank: R5 → R4 → R2 → R3 → R1

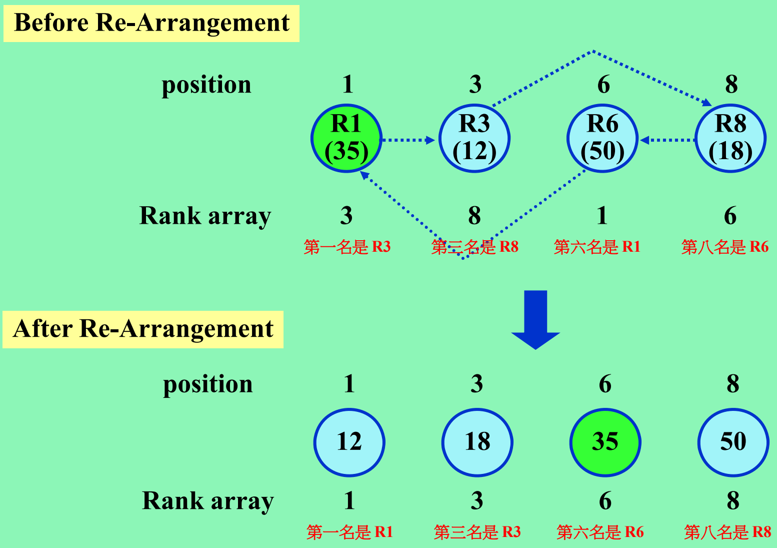
真正的次序由此決定

Auxiliary table after sorting

Cycle 1
t[1] = R5
t[5] = R1

Cycle 2
t[2] = R4
t[4] = R3
t[3] = R2

Re-Arrangement Within One Cycle



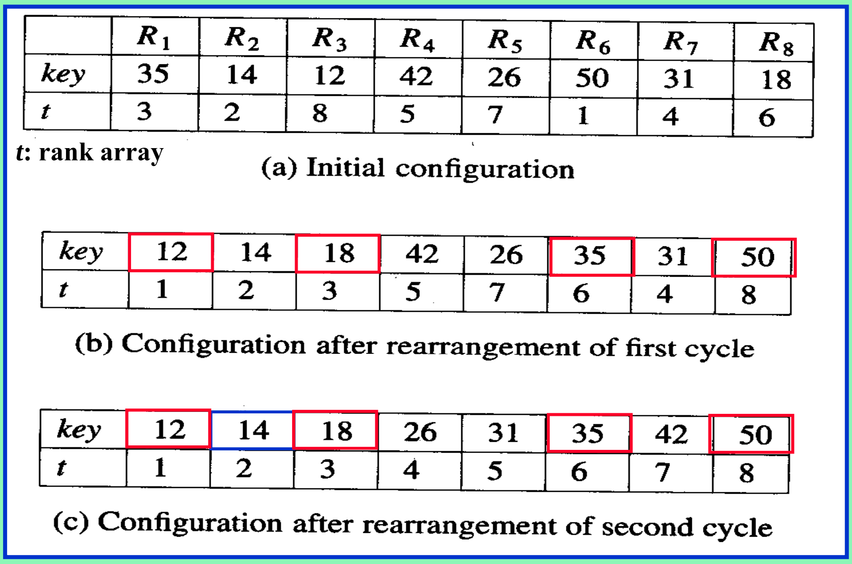
Before Re-Arrangement

position	1	3	6	8
	R1 (35)	R3 (12)	R6 (50)	R8 (18)
Rank array	3	8	1	6
	第一名是 R3	第三名是 R8	第六名是 R1	第八名是 R6

After Re-Arrangement

position	1	3	6	8
	12	18	35	50
Rank array	1	3	6	8
	第一名是 R1	第三名是 R3	第六名是 R6	第八名是 R8

Table Sort Example



	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
key	35	14	12	42	26	50	31	18
t	3	2	8	5	7	1	4	6

t: rank array

(a) Initial configuration

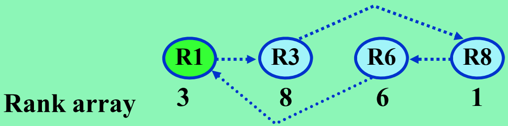
key	12	14	18	42	26	35	31	50
t	1	2	3	5	7	6	4	8

(b) Configuration after rearrangement of first cycle

key	12	14	18	26	31	35	42	50
t	1	2	3	4	5	6	7	8

(c) Configuration after rearrangement of second cycle

Table Sort



```
 void  table_sort ( Element  *list, const  int n,  int  *t )
// Rearrange list[1], ..., list[n] to correspond to the sequence
// list[ t[1]],  ...,  list[ t[n] ],  n≧1
{
   for ( int  i=1; i<n; i++) {
        if ( t[i] != i ) {  // There is a non-trivial cycle starting at i
              Element  p = list[i];  int j = i;  // remember first record p=list[i]
              do { // Move record list[k] to position j
                   int  k = t[j]; list[j] = list[k];  t[j] = j; j = k;
              } while ( t[j] != i );
              list[j] = p;   // Move record p to the last position in the cycle
              t[j] = j;
        } // end of if statement
   }
}
```

國立清華大學 電機工程學系
九十八學年度 第二學期
EE2410 Data Structure

Chapter 8 Hashing

清華大學電機系 黃錫瑜

Outline

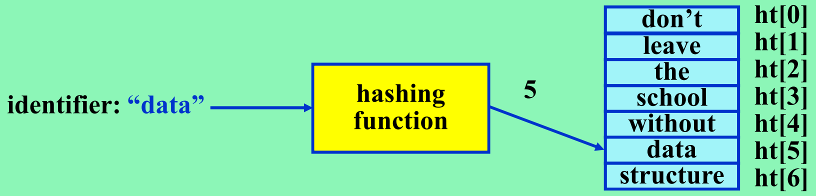
- The Symbol Table ADT
- Static Hashing
  - Hash Table
  - Hashing Function
  - Overflow Handling

Symbol Table

- **Symbol Table**
  - Can be viewed as a a set of name-attribute pairs
  - A form of dictionary
  - Applications include spelling checker, thesaurus, loaders, compilers
- **Common operations on a symbol table**
  - search a particular name in the table
  - retrieve the attributes of that name
  - modify the attributes of that name
  - insert a new name and its attributes
  - delete a name and its attributes

How To Implement Symbol Table?

- **Binary Search Tree**
  - allows efficient search, insert, and delete operation in O(h), where h is the height of the tree
  - Worst case O(n), where n is the total number of identifiers
  - Can be improved to O(log n) → Chapter 10
- **Hash Table**
  - A fixed-size linear array, ht
  - For an identifier, x,
  - The address of x is determined by a hashing function, h(x)



identifier: "data"

hashing function

5

don't	ht[0]
leave	ht[1]
the	ht[2]
school	ht[3]
without	ht[4]
data	ht[5]
structure	ht[6]

Terminology

- **Bucket and Slot**
  - There are 8 buckets and two slots per bucket in the hash table shown below
- **Identifier density**
  - is the ratio n/T, where
  - n is the number of identifiers in the table
  - T is the total number of possible identifiers
- **Loading factor**
  - is $\alpha = n/(sb)$, where
  - b is the number of buckets, s is the number of slots per bucket
- **Synonyms**
  - Two identifiers, I_1 and I_2 are synonyms if $h(I_1) = h(I_2)$

0	A	A2
1		
2		
3		
4	D	
5		
6		
7	GA	G

A hash table

Collision and Overflow

- **Collision**
  - When two nonidentical identifiers are hashed into the same bucket
- **Overflow**
  - when a new identifier is mapped or hashed by h into a full bucket

hashing function h(x) = 1st character of identifier x

Collision exists at location 0 and 7
Overflow will occur when AA is hashed into the table !

0	A	A2
1		
2		
3		
4	D	
5		
6		
7	GA	G

A hash table

Efficiency of Hash Table

- **Search or insertion time of a hash table**
  - **(1) compute the hash function**
  - **(2) search a bucket**
  - **The above times are independent of n**

0	A	A2
1		
2		
3	D	
4		
5		
6		
7	GA	G

A hash table

- **Collision is inevitable**
  - **Taking the first character is not a good hashing because of too much collision**
    - **many variables in a program begins with the same character**
  - **An overflow mechanism is necessary**

Uniform Hash Function

- **Basic desired properties of hash function**
  - **Easy to compute**
  - **The number of collisions is minimized**
- **A good hash function**
  - **should also depend on every character of an input identifier**
- **Uniform hash function**
  - **Let x be an identifier chosen at random**
  - **Then, the probability that h(x) = i is 1/b for every bucket i**
  - **That is, the hash function does not result in a biased use of the hash table for random inputs**

Hash Function (I): Division

hash function h_D (x) = x % M
where % is the modulo operator
That is, the remainder is used as the hash address

6 bits per character

0	0	0	0	0	0	A	1

right-justified zero-filled

6 bits per character

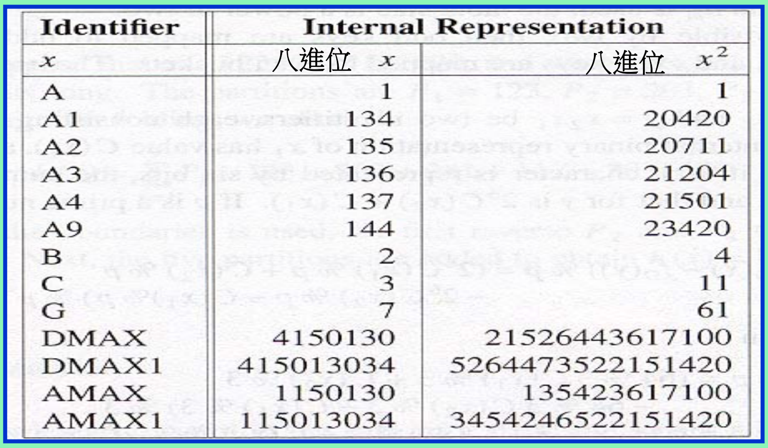
A	1	0	0	0	0	0	0

left-justified

- **Hash address**
  - in the range from 0 through (M-1) → implies that table size is M
- **M should not be a power of 2**
  - otherwise, $h_D(x)$ may depend only on the least significant bits of x
  - E.g., $M=2^3$, then A1→encoded to $2^6(10) + 1$ → $h_D(A1) = 1$
    XY1 → encoded to $2^{12}(33) + 2^6(34) + 1$ → $h_D(A1) = 1$
  - M is usually a prime number

Hash Function (II): Mid-Square

mid-square function h_m (x) = use appropriate # bits from (x^2)



Identifier	Internal Representation	
x	八進位 x	八進位 x^2
A	1	1
A1	134	20420
A2	135	20711
A3	136	21204
A4	137	21501
A9	144	23420
B	2	4
C	3	11
G	7	61
DMAX	4150130	21526443617100
DMAX1	415013034	5264473522151420
AMAX	1150130	135423617100
AMAX1	115013034	3454246522151420

The coding of identifier x is right-justified, zero-filled, and has six bits per character
Table size will be a power of 2

Hash Function(III): Folding

- **Example: x = 12320324111220**
- **Step1: Partition the identifier into several parts**
  - P_1=123, P_2=203, P_3=241, P_4=112, P_5=20
- **Step 2: Add up each part as a hash address**
  - (1) Shift folding

$$h(x) = \sum_{i=1}^{5} P_i = 123 + 203 + 241 + 112 + 20 = 699$$

  - (2) Folding at the boundaries

$$h(x) = 123 + 302 + 241 + 211 + 20 = 897$$

reversed

Hash Function (IV): Digit Analysis

- **Application**
  - when all the identifiers are known in advance
- **Procedure**
  - Step 1: Each identifier is interpreted as a number using radix r
  - Step 2: Analyze the distribution of each digit
  - Step 3: Drop biased digits
    - The digits with the most skewed distributions are deleted one by one until the remaining digits is small enough to give an address
- **Example**
  - Given three identifiers in radix-9 form: 891, 792, 793
  - Digit distribution: 1st {8, 7, 7}, 2nd {9, 9, 9}, 3rd {1, 2, 3}
  - The most skewed digits: the 2nd

Outline

- **The Symbol Table ADT**
- **Static Hashing**
  - **Hash Table**
  - **Hashing Function**
  - ➡ **Overflow Handling**

Overflow handling

- **Problem**
  - **When a new identifier is hashed into a full bucket, then we need to find another open bucket**
- **Methods**
  - **linear probing (or linear open addressing)**
    - **find the closest bucket that is not full**
  - **chaining**
    - **implement each bucket as a linked list**

Symbol Table Class Definition

```
struct  identifier {
    char  *id;
    int     n;
};

// Assume that operators == and != are defined for identifier
int  operator==(identifier&, identifier&);
int  operator!=(identifier&, identifier&);

class  SymbolTable {
public:
      SymbolTable( int  size = defaultsize ) {
            buckets = size;
            ht = new identifier[buckets];  // linear array as the table
      }
private:
      int  buckets;
      identifier  *ht;
};
```

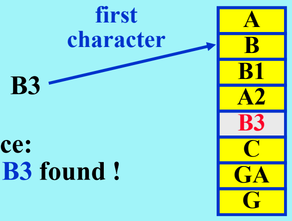
Linear Open Addressing

- **Procedure of searching an identifier x**
  - **Step 1: compute h(x)**
  - **Step 2: examine identifiers at positions ht[h(x)], ht[h(x)+1], ..., ht[(h(x)+j] in this order until one of the following happens:**

    (a) ht[h(x)+j] = x; in this case x is found

    (b) ht[h(x)+j] is null; x is not in the table

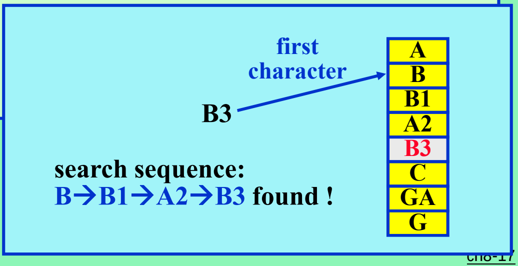
    (c) We return to the starting position h(x); the table is full and x is not in the table



search sequence:
B→B1→A2→B3 found !

Linear Search

```
int  SymbolTable::LinearSearch(
            const  identifier& x, int (*hashfunc) (identifier))
// Search the hash table ht ( each bucket has exactly one slot) for x using
// linear probing.
// Return j such that if x is already in the table, then ht[j] = x
// If x is not in the table, return –1
// The hash function "hashfunc" is passed as an argument to LinearSearch
{
   int  i = hashfunc(x);
   for ( int  j=i; ht[j].id && ht[j] !=x; ) {
          j = (j+1) % buckets;  // treat the table as circular
         if ( j==i ) return –1;  // back to start point
    }
    if ( ht[j] == x)  return j;
    else  return –1;
}
```



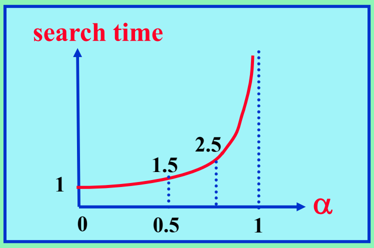
search sequence:
B→B1→A2→B3 found !

Problem of Linear Open Addressing

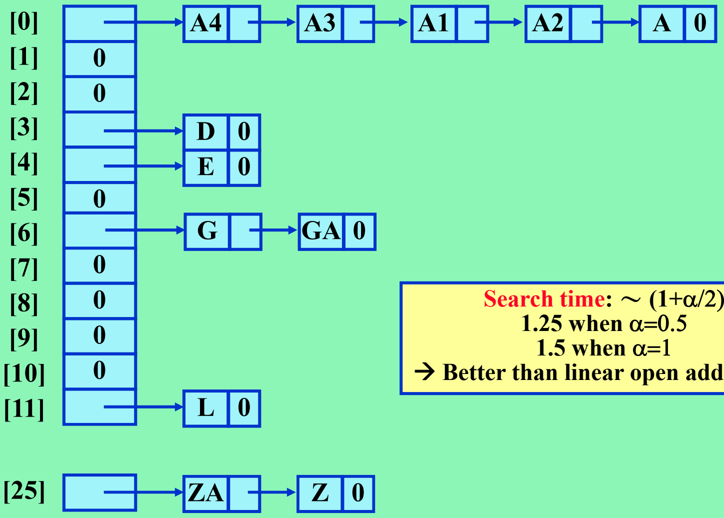
- Identifiers tend to cluster together
  - Increase the search time
  - Could be worse than the search tree structure
- An analysis shows that
  - It takes $(2-\alpha)/(2-2\alpha)$ to look up an identifier
  - Where α is the loading density
- Quadratic probing
  - improve the clustering problem
  - check sequence:

    $(h(x)+i^2)\%b$ and $(h(x)-i^2)\%b$

    i=1, 2, ...



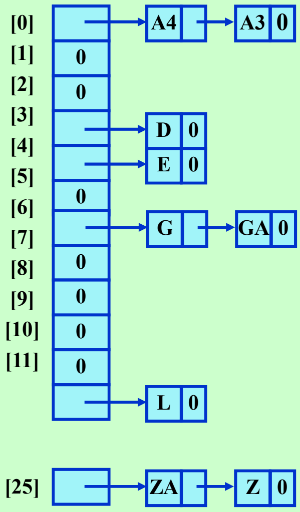
Chaining



Search time: ~ (1+α/2)
1.25 when α=0.5
1.5 when α=1
→ Better than linear open addressing

Class Definitions For Chain Search

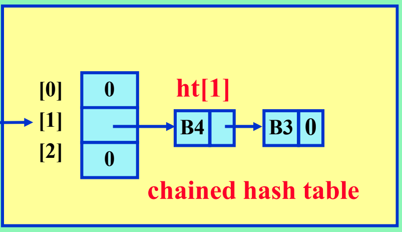
```
class  ListNode {
friend  SymbolTable;
private:
     identifier  ident;
     ListNode  *link;
};

typedef  ListNode*  ListPtr;

class  SymbolTable {
public:
   SymbolTable (  int  size = defaultsize ) {
         buckets = size;
         ht = new  ListPtr[buckets];
   };
private:
   int  buckets;   // table size
   ListPtr *ht;    // hash table
};
```



Chain Search

```
identifier*  SymbolTable::ChainSearch(const identifier& x, int (*hashfunc)
                                                       (identifier) )
// Search the chained hash table ht for x. On termination, return a pointer
// to the identifier in the hashtable. If the identifier does not exist, return 0
{
   int  j = hashfunc(x); // compute headnode address
   // search the chain starting at ht[j]
   for ( ListPtr  node = ht[j]; node; node = node→link )
         if ( node→ident == x)  return &(node→ident);
   return 0;
}
```

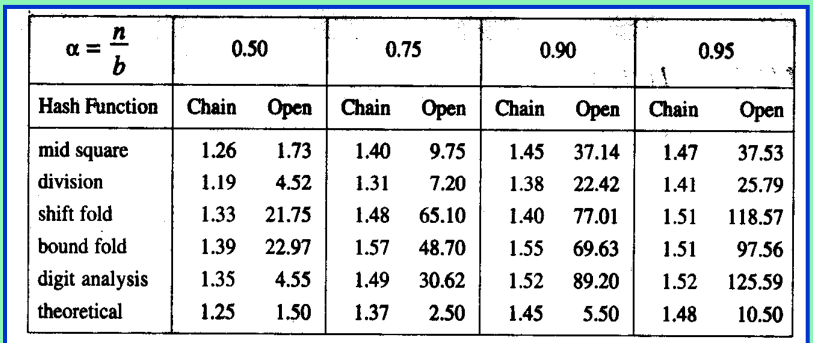
j = hashfunc(“B3”) = 1



chained hash table

A Comparison

- **Hash Function**
  - **division** is generally superior to the other types
- **Collision handling**
  - **Chaining** outperforms **linear opening addressing**



$\alpha = \frac{n}{b}$	0.50		0.75		0.90		0.95	
Hash Function	Chain	Open	Chain	Open	Chain	Open	Chain	Open
mid square	1.26	1.73	1.40	9.75	1.45	37.14	1.47	37.53
division	1.19	4.52	1.31	7.20	1.38	22.42	1.41	25.79
shift fold	1.33	21.75	1.48	65.10	1.40	77.01	1.51	118.57
bound fold	1.39	22.97	1.57	48.70	1.55	69.63	1.51	97.56
digit analysis	1.35	4.55	1.49	30.62	1.52	89.20	1.52	125.59
theoretical	1.25	1.50	1.37	2.50	1.45	5.50	1.48	10.50

(Adapted from V. Lum, P. Yuen, and M. Dodd, *CACM*, 14:4, 1971)

What We Have Learned?

- **Array, Stack, Queue, Linked List**
- **Tree, Graph, Sorting, Hash Table**
- 老鼠走迷宮…
- 尤拉的散步…
- 撲克牌的排序…

Programming is not just coding.
It is all about problem solving !
Good luck on your Journey
As a problem solver !