國立清華大學 電機工程學系
EE2410 Data Structure

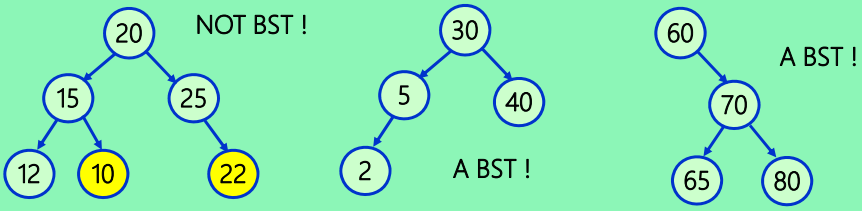
Chapter 5
Trees (Part II)

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))



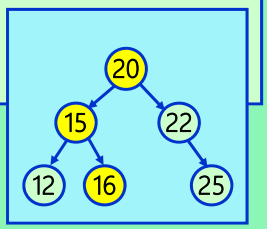
Why Binary Search Tree ?

- **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*

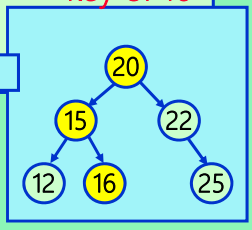


Iterative Search Of a BST

```
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

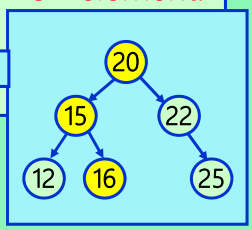
```
template <class Type>
BstNode<Type>*  BST<Type>::Search(int k)
// Search the binary search tree for the k-th smallest element
{
  BstNode<Type>  *t = root;
  while( t )
  {
     if ( k==t→LeftSize )  return t;
     if ( k < t→LeftSize )  t = t→LeftChild;
     else {
          k = k – t→LeftSize;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

     }
  }
  return 0;
}
```

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

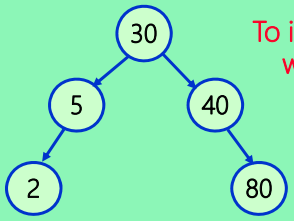
Complexity: O(h), where h is the height

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

Finding 3rd element:



Example: Insertion to a BST

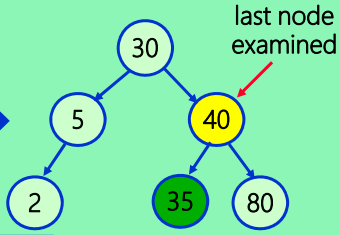


To insert a node with key 35

A search is first carried out

unsuccessful !

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



last node examined

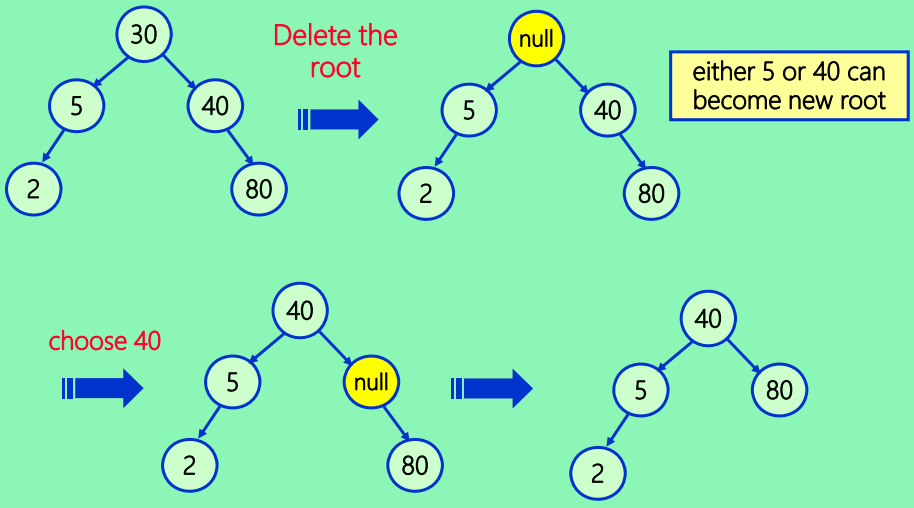
If a node has a LeftSize field, then it has to be updated

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;
}
```

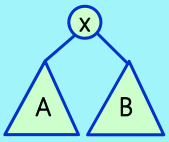
Complexity: O(h)
where h is the height of BST

Example: Deletion From a BST

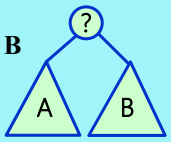


Joining and Splitting Binary Trees

- **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

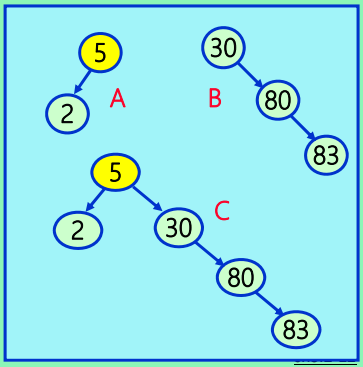
Example: Joining BST's

- **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

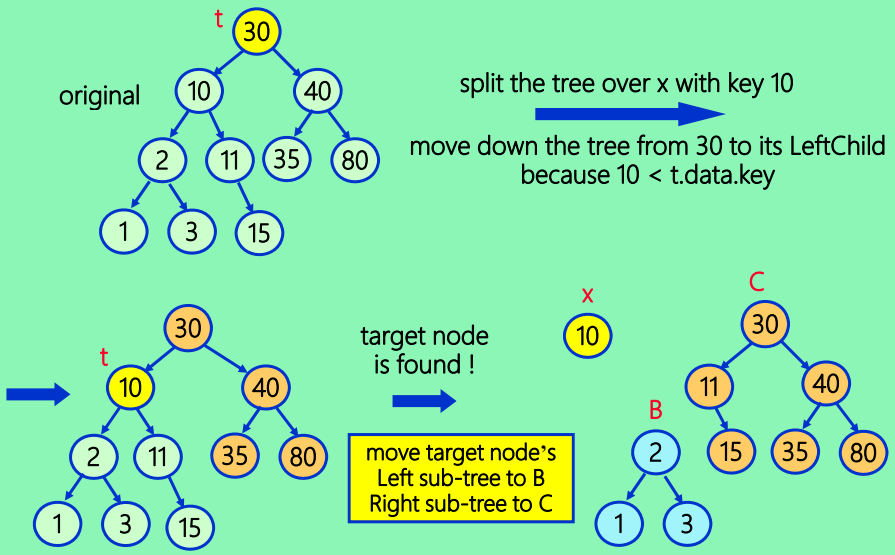
    Let the resulting tree as A'
  - Step 2: perform three-way join operation

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

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

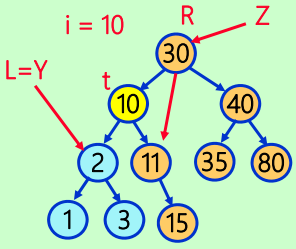


Example: Splitting a BST



Splitting A BST

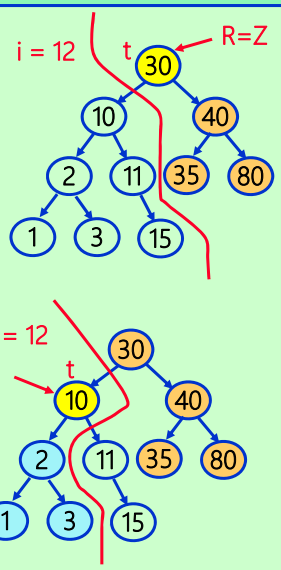
```
template <class Type>
Element<Type>* BST<Type>::Split(Type i, BST<Type>& B,
                                Element<Type>& x, BST<Type>& C)
// Split the binary search tree with respect to key i
{
   if ( ! root ) {  B.root = C.root = 0; return 0; } // empty tree
   // create head nodes for B and 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

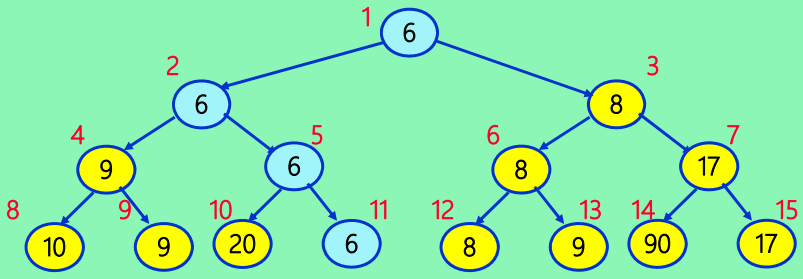
- Binary Search Trees
- **Selection Trees**
- Forests
- Set Representation
- Counting Binary Tree

Problem of Multi-Way Merging

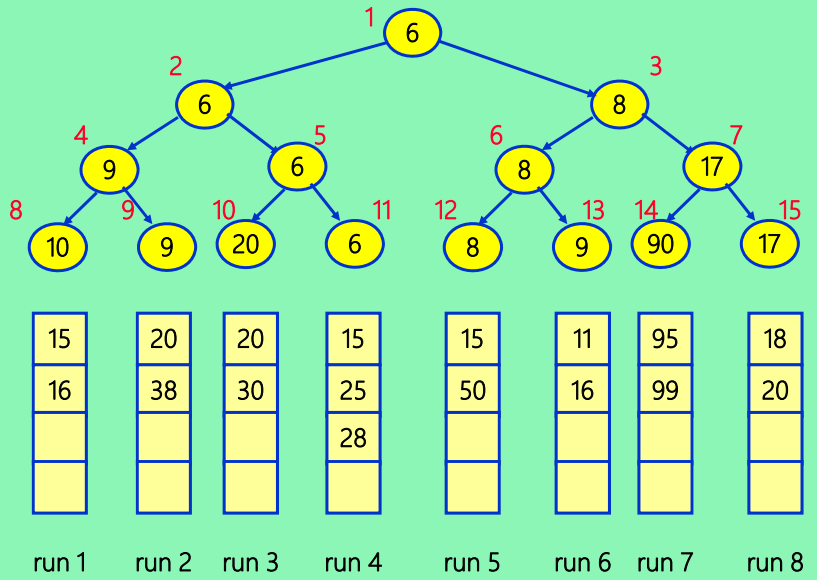
- **Problem**
  - Merge k ordered sequences, called runs, into a single ordered sequence
- **Example**
  - R1 = {15, 16}, R2 = {20, 38}, and R3 = {1, 17}
  - The merged output = {1, 15, 16, 17, 20, 38}
- **A naïve algorithm**
  - Find the smallest element by examining the first element of each run
  - 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

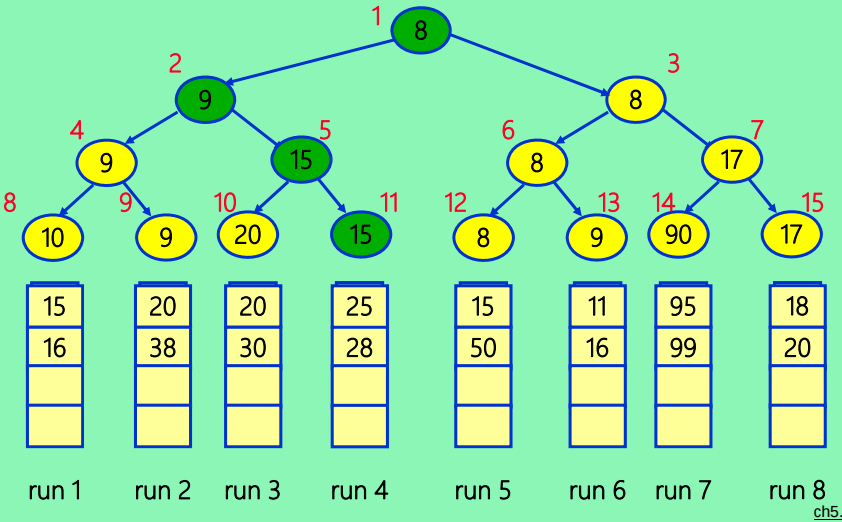


Winner Tree for k = 8



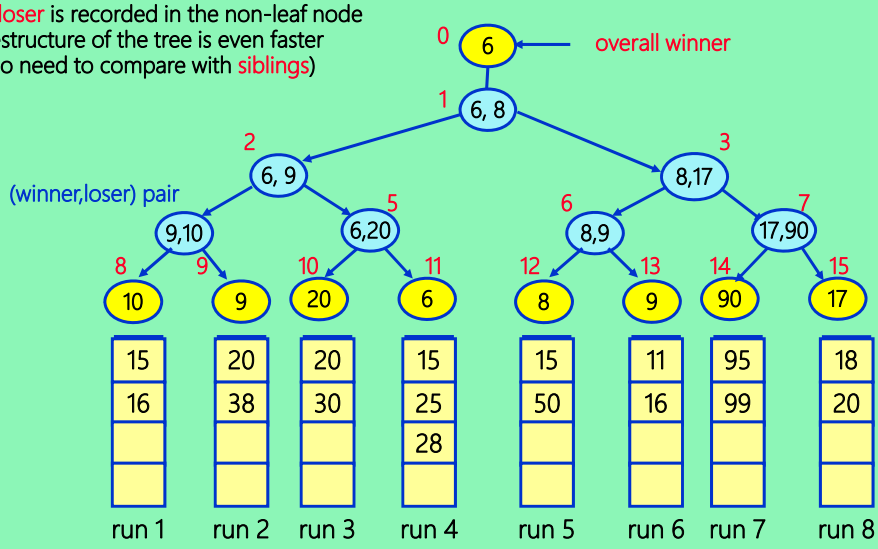
Winner Tree After One Record Is Output

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)

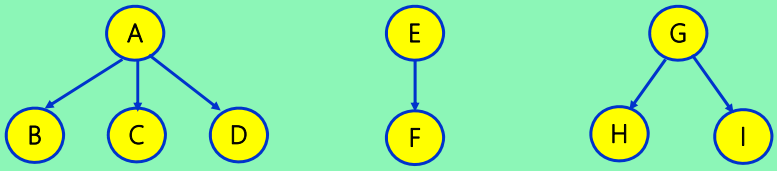


Outline

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

Forrest

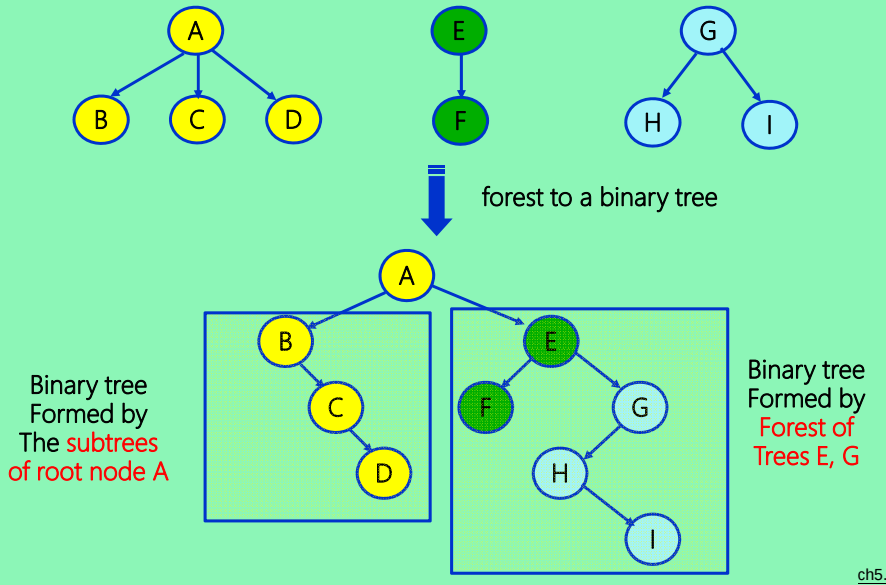
- **Definition**
  - **A Forest is a set of n ≧0 disjoint trees**
- **Operations**
  - **Transforming a forest into a binary tree**
  - **Forest traversal**
- **Examples**



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 Traversal

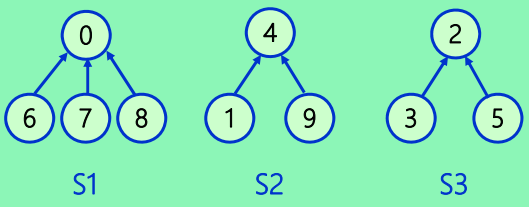
- **Forest Preorder Traversal**
  - **If F is empty then return**
  - **Visit the root of the first tree of F**
  - **Traverse the sub-trees of the first tree in forest preorder**
  - **Traverse the remaining trees of F in forest preorder**

Outline

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

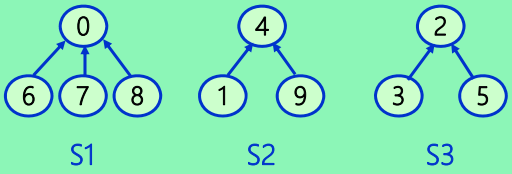
Set Representation

- 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}

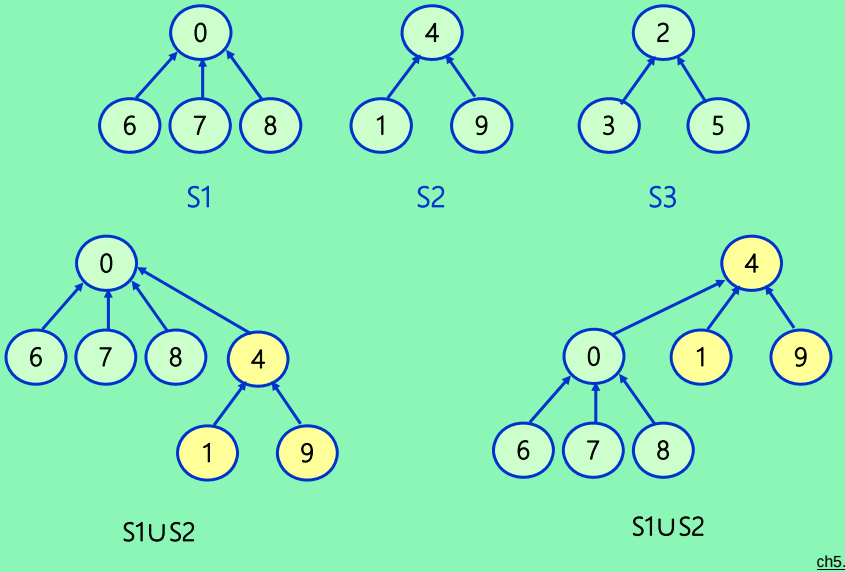


Set Operations

- **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 ∪ 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

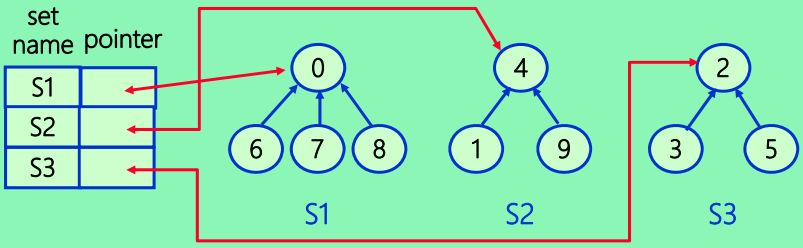


Union Operation



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

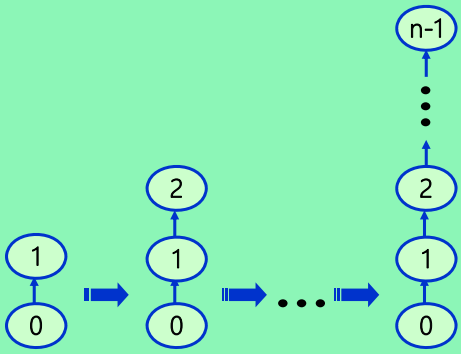
```
class Sets {
public:
      // Set operations follow
private:
      int   *parent;
      int   n;  // number of set elements
};
Sets:Sets(int sz = HeapSize)
{
      n = sz;  parent = new int[sz];
      for ( int i=0; i<n; i++)  parent[i] = -1;
}
void  Sets::SimpleUnion(int i, int j)
// replace the 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**

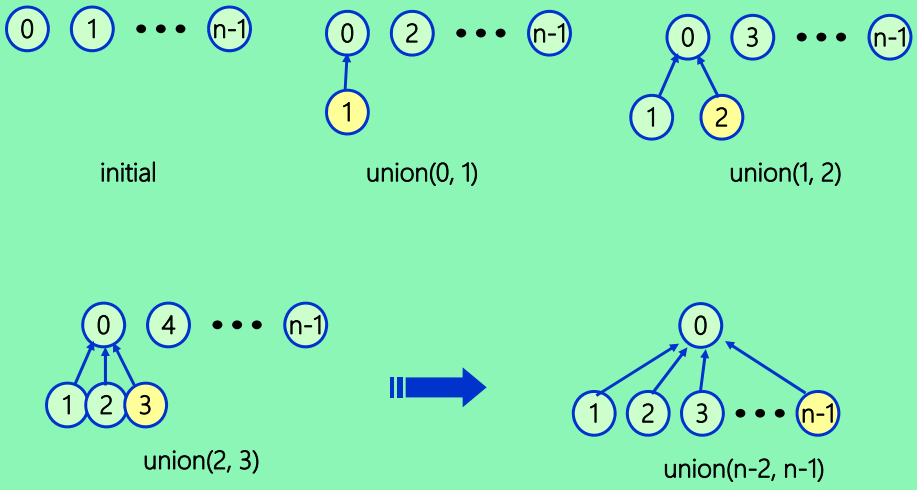


find(0) = O(n)
find(1) = O(n-1)
find(2) = O(n-2)
...
find(n-1) = 1
---
total = $O(n^2)$

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

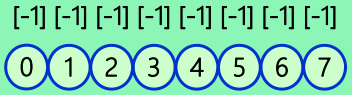


Union Algorithm With Weighting Rule

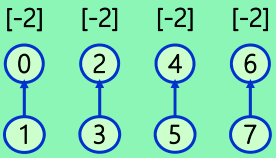
```
void  Sets::WeightedUnion(int i, int j)
// Union sets with roots i and j, i ≠j, using the weighted rule
// parent[i] = - count[i] and parent[j] = -count[j]
{
      int   temp = parent[i] + parent[j];
      if (parent[i] > parent[j]) {       // i has fewer nodes
            parent[i] = j;
       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

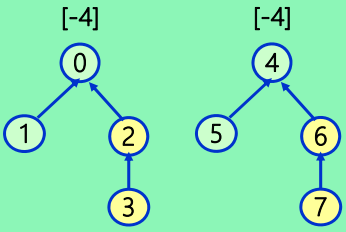
Trees Achieving Worst-Case Bound



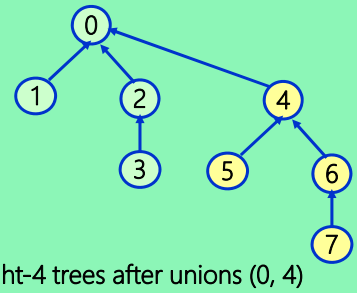
initial height-1 trees



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)

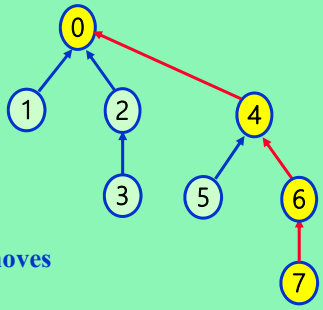
Collapsing Rule for Union(i, j)

- **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**

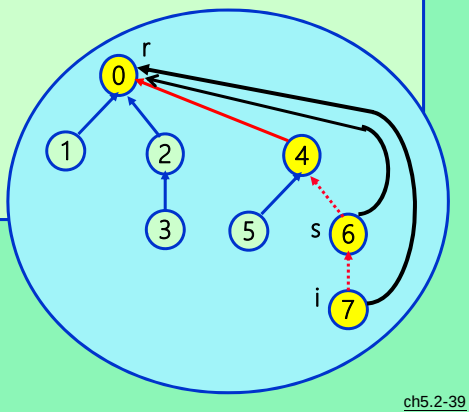
    ➔ **3 + 3 + 1 + 1+ 1+ 1+ 1+ 1+ 1 = 13 moves**



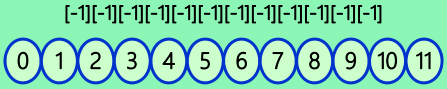
1st find(7): including traversal & collapsing

Find Algorithm With Collapsing Rule

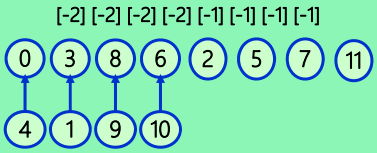
```
int  Sets::CollapsingFind(int i)
// Find the root of the tree containing element i
// Use the collapsing rule to collapse all nodes from i to the root
{
      for ( int r = i; parent[r] >= 0; r = parent[r]);  // find root
      while ( r != i ) {
            int  s = parent[i];
            parent[i]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



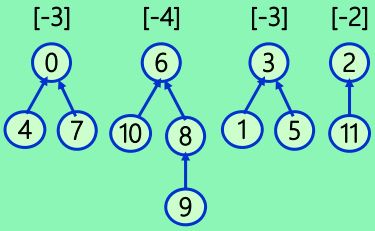
Application To Equivalence Classes



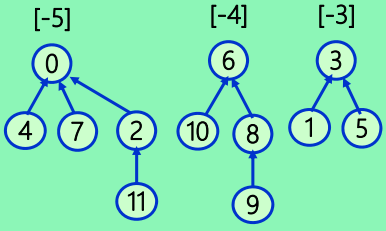
initial height-1 trees



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

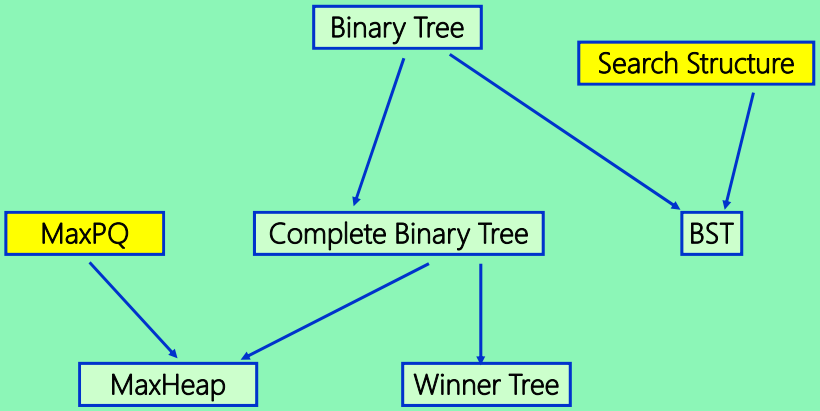


Trees after processing
7≡4, 6≡8, 3≡5, 2≡11



Trees after processing
11≡0

Inheritance Graph Among Trees



Outline

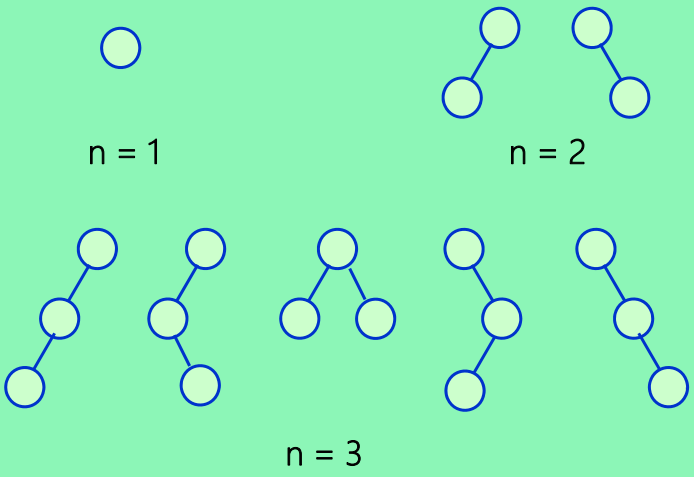
- **Binary Search Trees**
- **Selection Trees**
- **Forests**
- **Set Representation**
- **Counting Binary Tree**

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 ?**

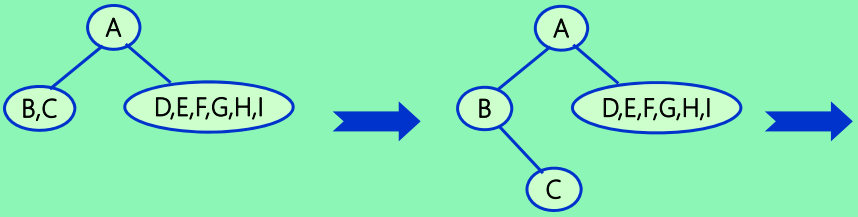
➔ **All the three problems have the same answer !**

Example: Distinct Binary Trees



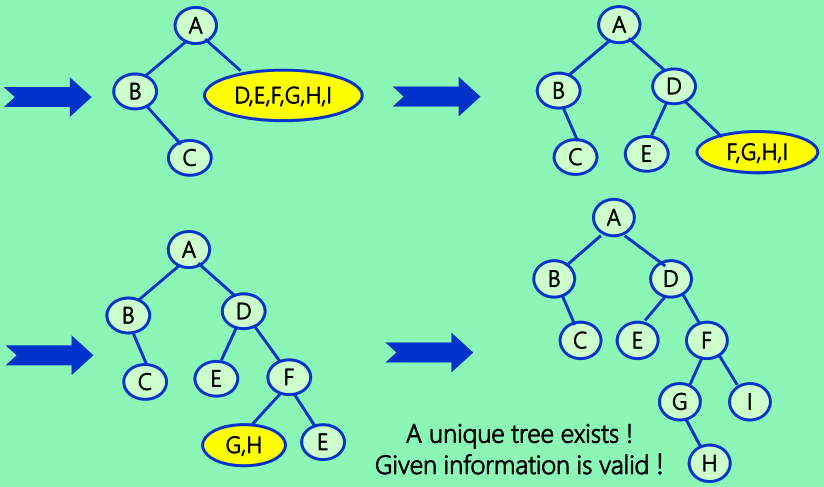
Distinct Binary Trees According To A PreOrder / InOrder Sequence Pair

- **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

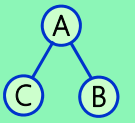


A unique tree exists !
Given information is valid !

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

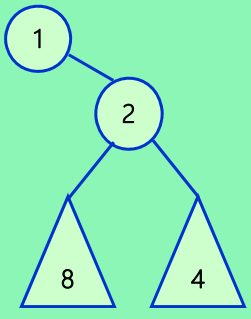


violate the preorder

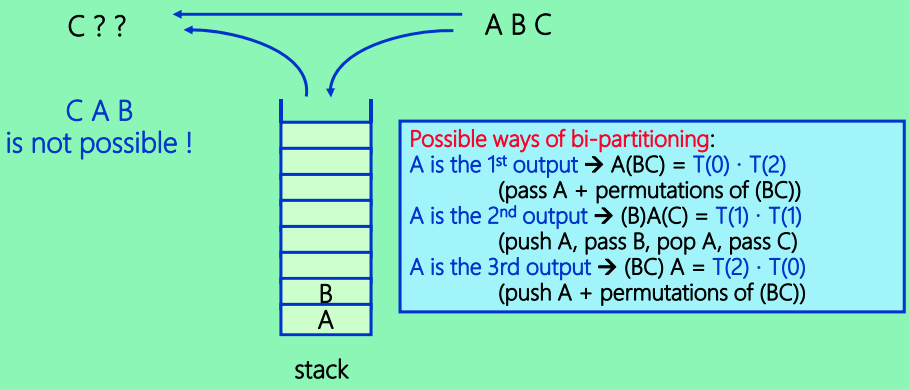
Example of Invalid Order Pair

- **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

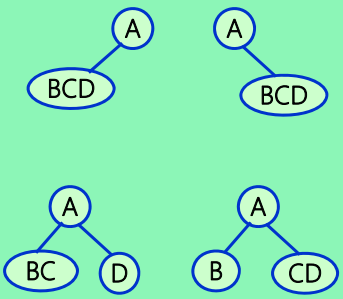


(A B C), (A C B), (B A C), (B C A), (C B A)
➔ all valid permutations and valid in-order sequence

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:

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

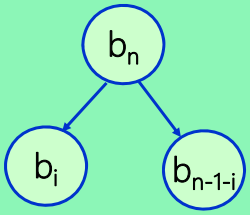
$$B_n = \sum_{i=0}^{n-1} B_i \, B_{n-i-1} \quad n \geq 2$$

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



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

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