國立清華大學 電機工程學系
EE2410 Data Structure

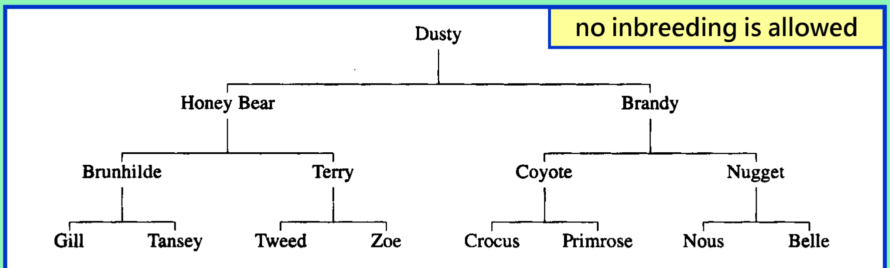
Chapter 5
Trees (Part I)

Outline

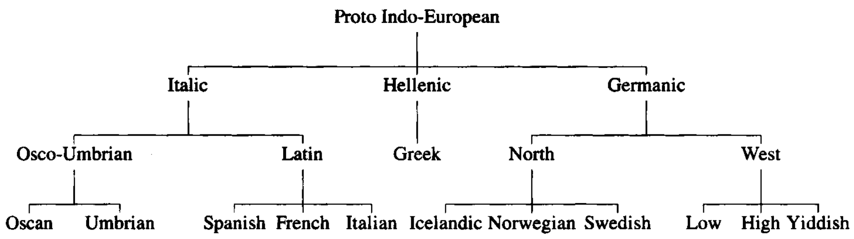
- **Introduction**
- **Binary Trees**
- **Binary Tree Traversal**
- **Additional Binary Tree Operations**
- **Threaded Binary Trees**
- **Heaps**

Genealogical Charts

no inbreeding is allowed



(a) Pedigree

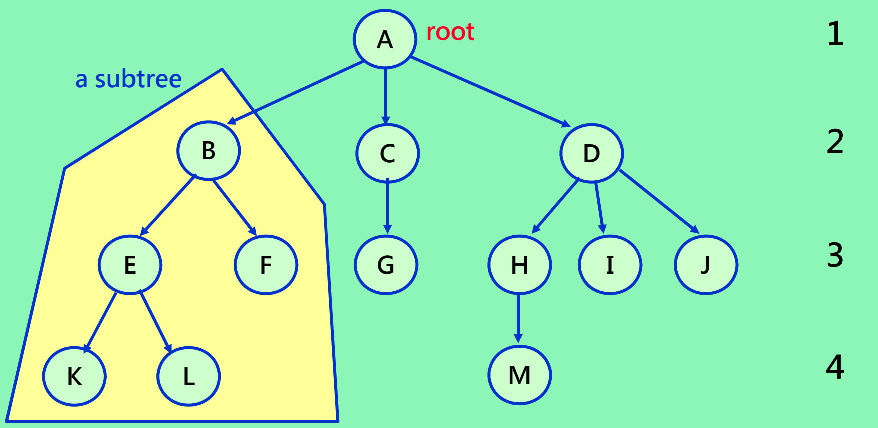


(b) Lineal

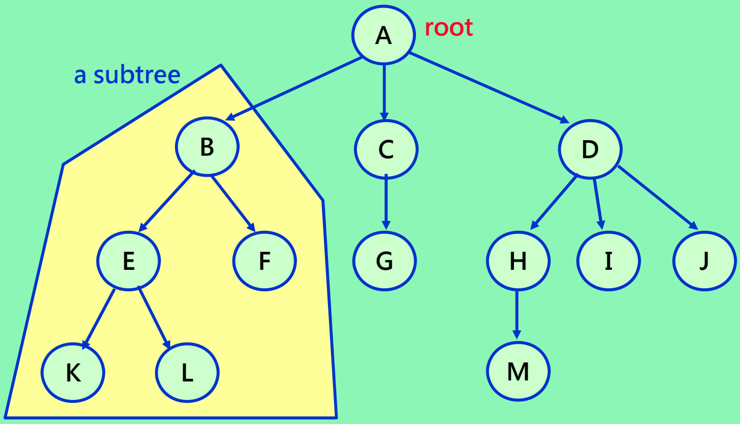
A Simple Tree

Every Node
→ can have many children nodes
→ but can only have one parent node
Degree → The maximum no. of children nodes

LEVEL



Direct Representation of a Tree



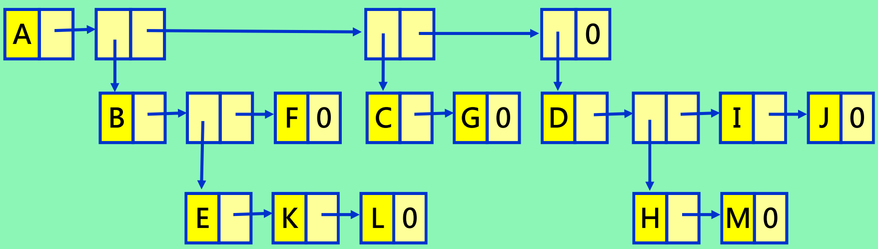
Possible node structure for a tree of degree k

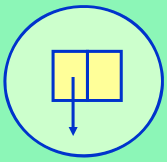
Data	Child1	Child2	...	Child k

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)))**





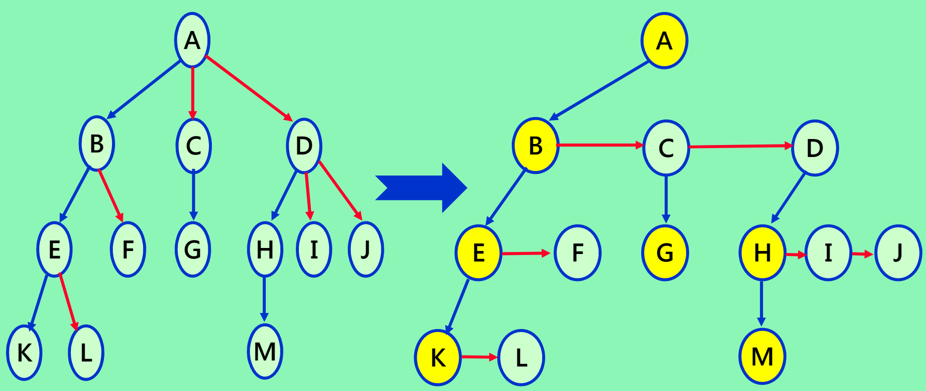
This type of node indicates a sub-tree

Lemma 5.1

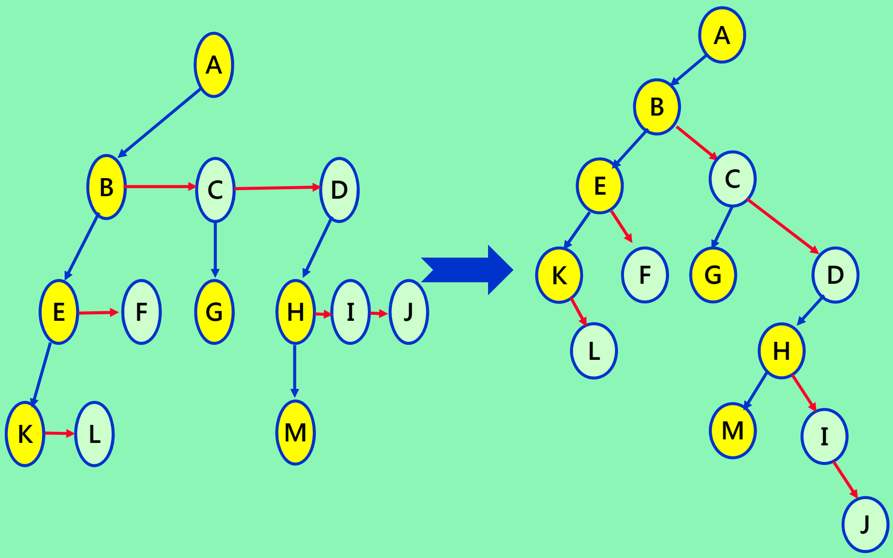
- **Lemma**
  - If T is a k-ary tree (I.e., a tree of degree k) with n nodes, each having a fixed size as shown in previous slide, then $n(k\text{-}1) + 1$ of the nk child fields are 0, $n \geqq 1$
- **Proof:**
  - The number of non-0 child fields in an n-node tree is exactly n-1
  - The total number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

Left Child-Right Sibling 嫡長子-庶子 Representation

Data	
left child	right sibling

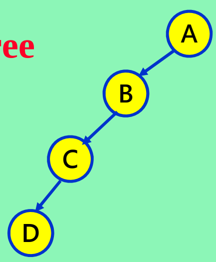


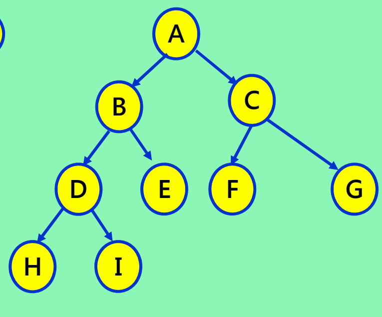
Degree-Two Tree



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}**

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

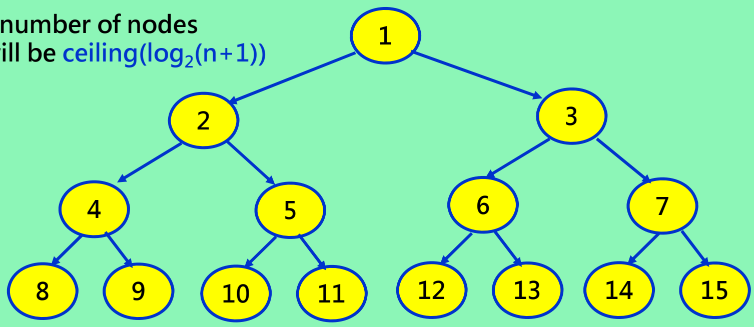
Leaf Nodes vs. Degree-2 Nodes

- **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

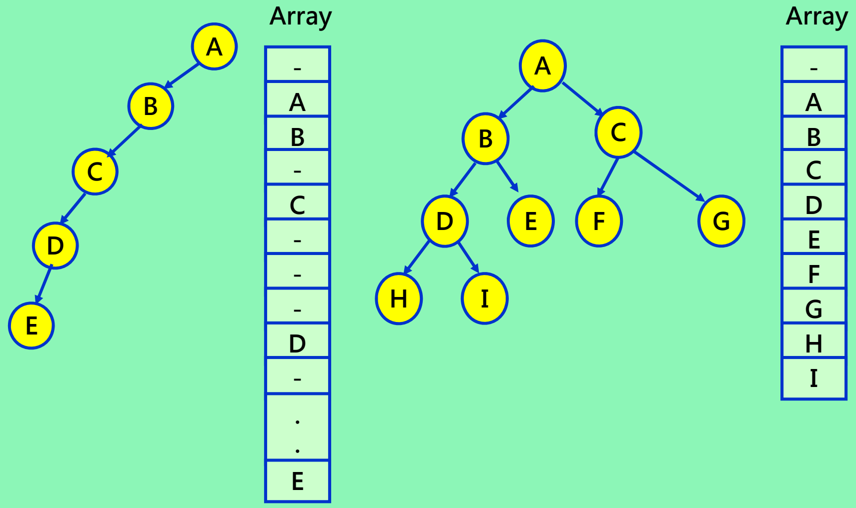
If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \leq i \leq n$, we have

(1) parent(i) is at $\lfloor i/2 \rfloor$ if $i \neq 1$

(2) LeftChild(i) is at 2i if $2i \leq n$. If $2i > n$, then i has no left child.

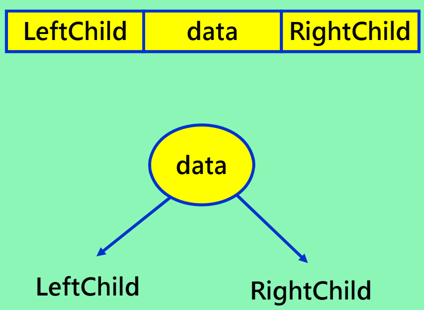
(3) RightChild(i) is at 2i+1 if $2i+1 \leq n$. If $2i+1 > n$, then i has no right child

Array Representation of A Tree

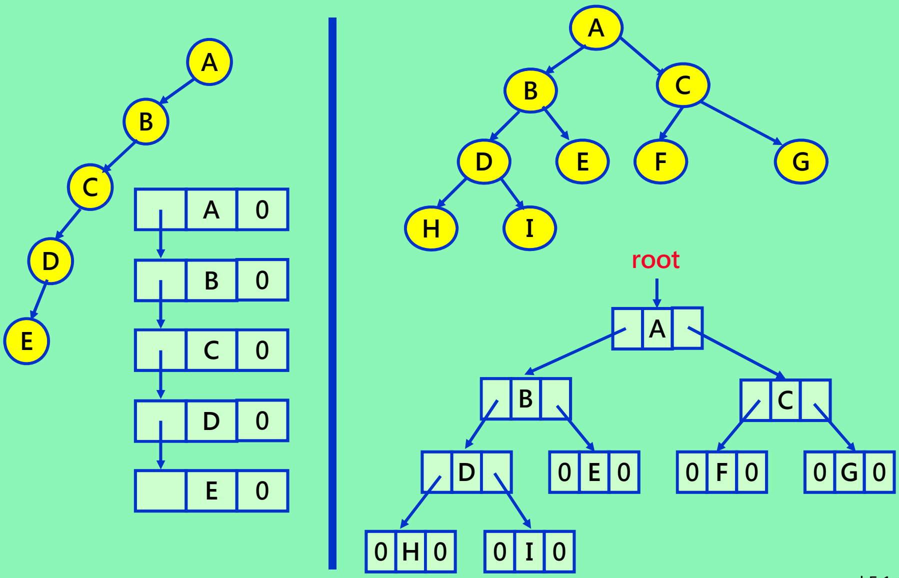


Linked Representation

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

class  Tree {
public:
   // Tree operations

private:
   TreeNode  *root;
};
```



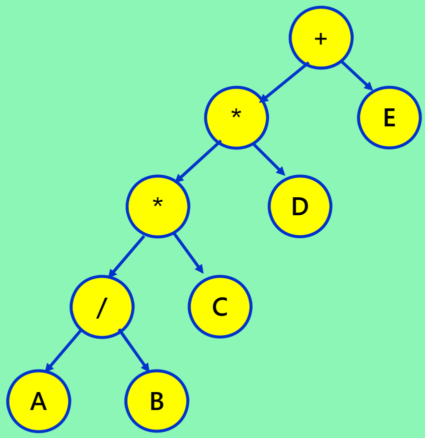
List Representation of A Tree



Outline

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

Binary Tree With Arithmetic Expression



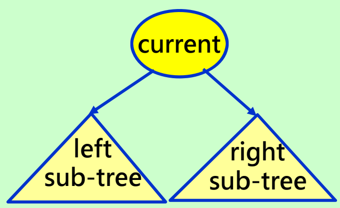
Inorder Traversal
A/B * C * D + E

Postorder Traversal
A B / C * D * E +

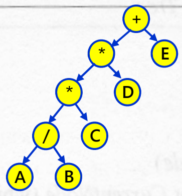
Preorder Traversal
+**/ A B C D E

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);
   }
}
```



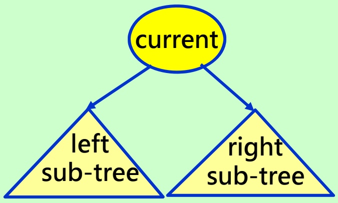
Trace of Inorder Traversal



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

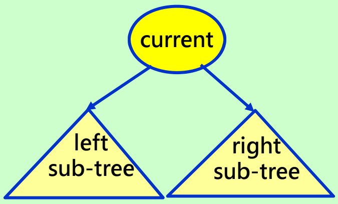
Preorder Traversal of a Binary Tree

```
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);
   }
}
```

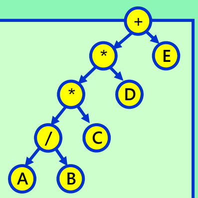


Postorder Traversal of a B-Tree

```
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);
   }
}
```



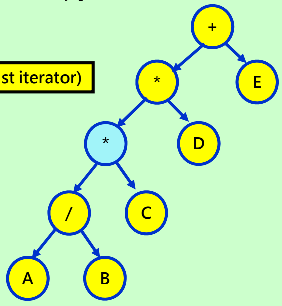
Iterative Inorder Traversal




```
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;
   }
}
```

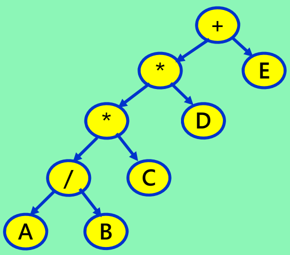
Inorder Traversal Using Iterator

```
class InorderIterator {
public:
   char *Next();
   InorderIterator(Tree tree): t (tree) { CurrentNode = t.root; }
private:
   const Tree& t;
   Stack<TreeNode*> s;   New data member (not needed in a list iterator)
   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
}
```



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)

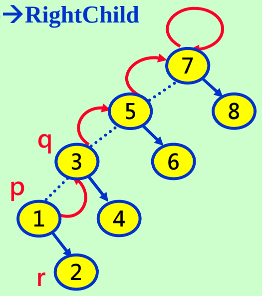
Level-order traversal:
+ * E * D / C A B

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

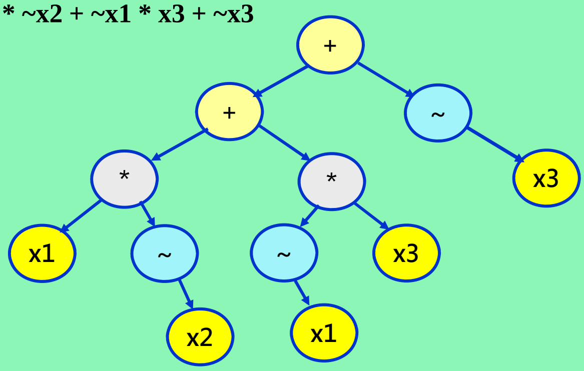
  (2) if x and y are expressions, then x*y, x+y, and ~x are expressions

  (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" ;
```

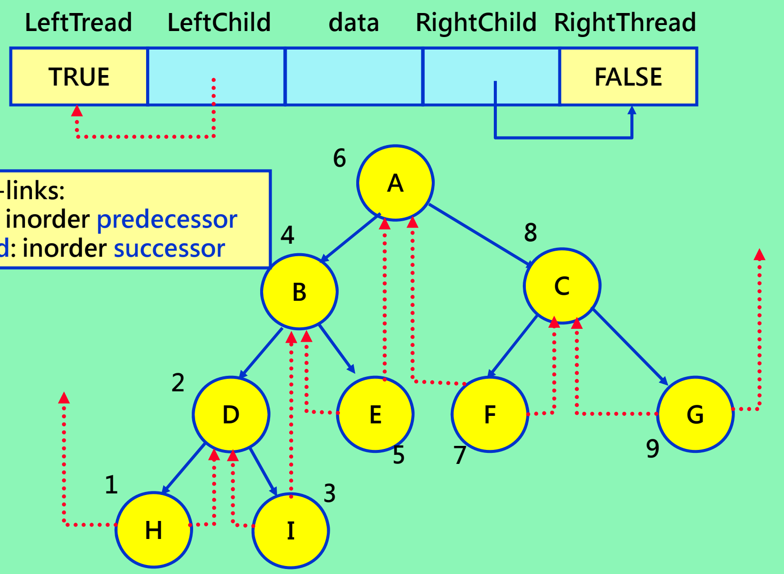
Evaluating A Formula

```
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

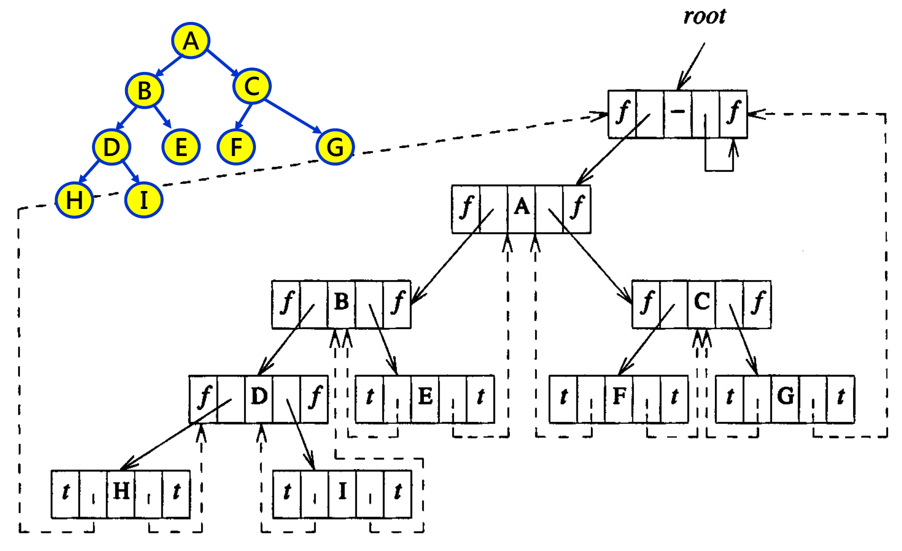


To utilize 0-links:
left thread: inorder predecessor
right thread: inorder successor

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;
}
```

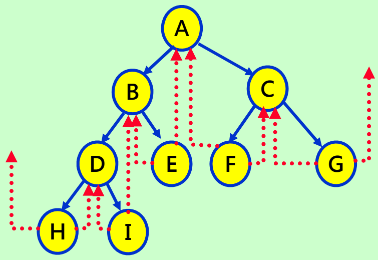
Memory Representation of Threaded Tree



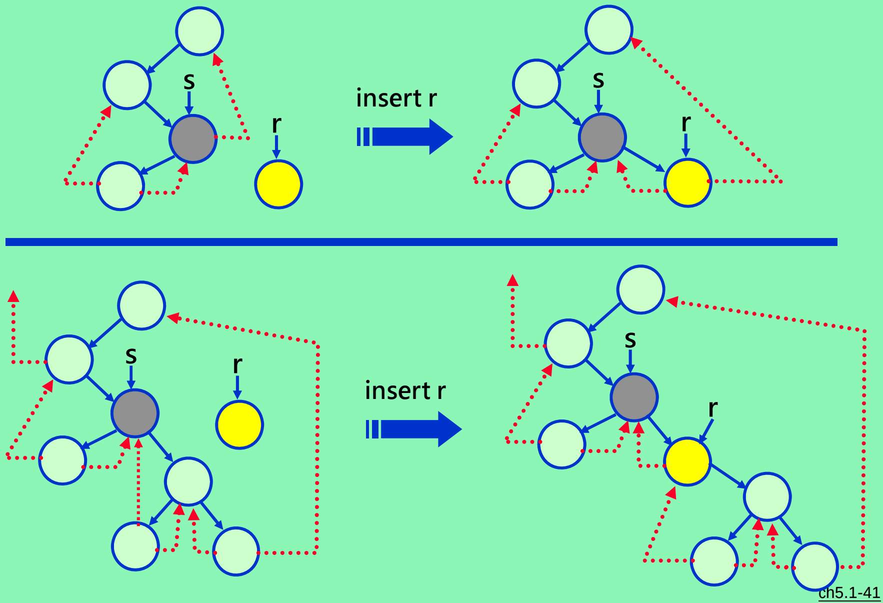
f = FALSE; *t* = TRUE

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;
   }
}
```

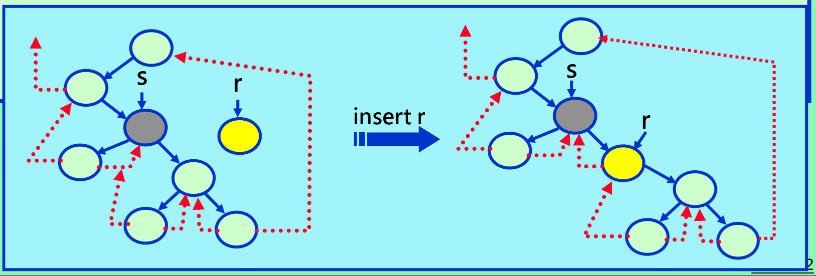


Inserting A Node In Threaded Tree



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**

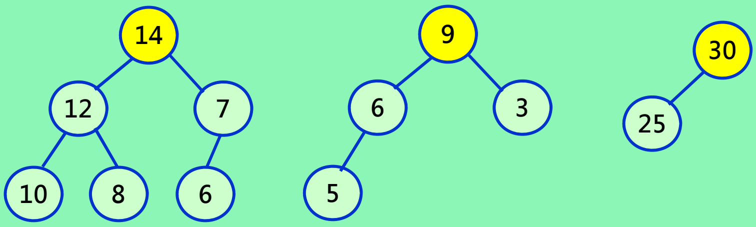
Priority Queue

- **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

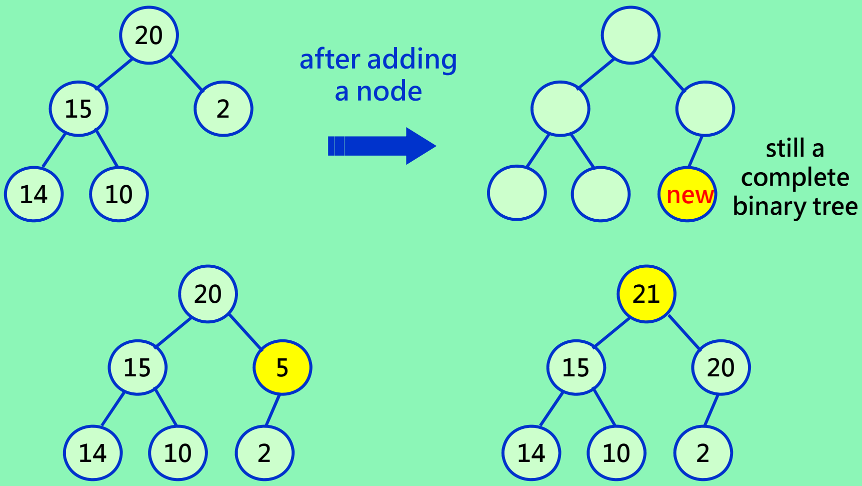
- **Heap**
  - is frequently used to implement a priority queue
- **Definition**
  - A max(min) tree is a tree in which the key value in each node is no smaller (larger) than the key values in its children (if any)
  - A max heap is a complete binary tree that is also a max tree
  - A min heap is a complete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

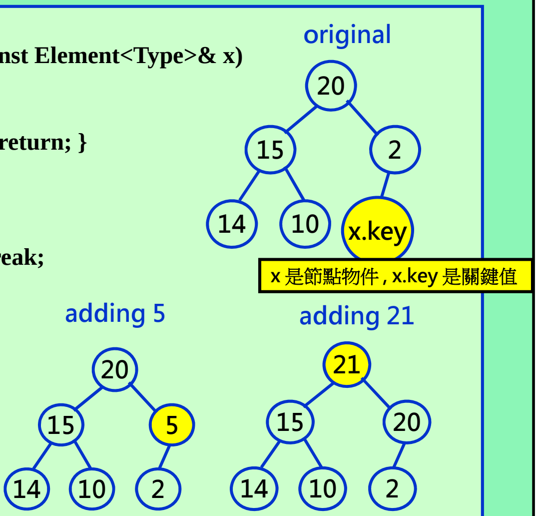
Insertion To a Max Heap



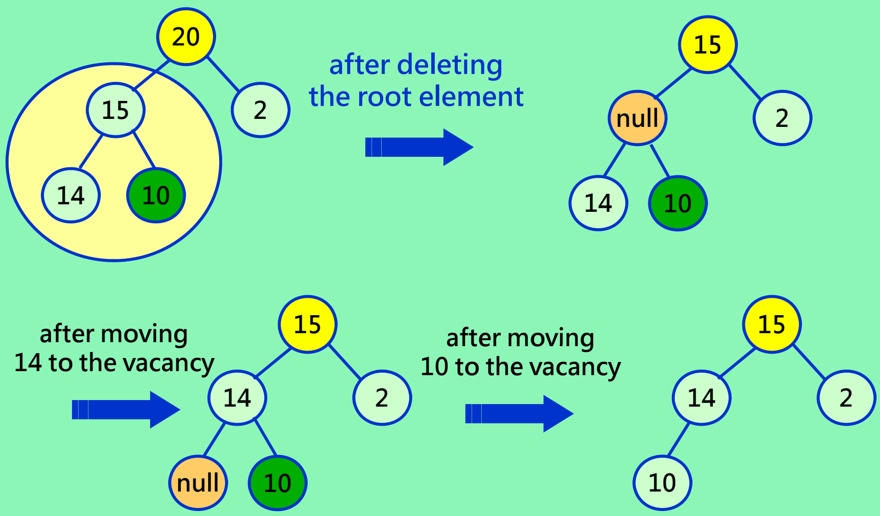
Final tree after inserting 5 Final tree after inserting 21

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
}
```

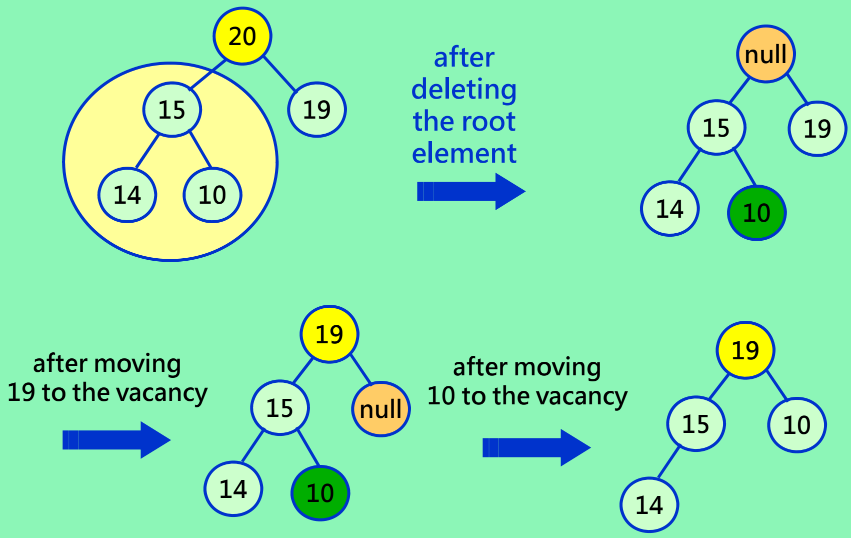


Deletion From a Max Heap



Final tree → still complete binary tree

Deletion From a Max Heap

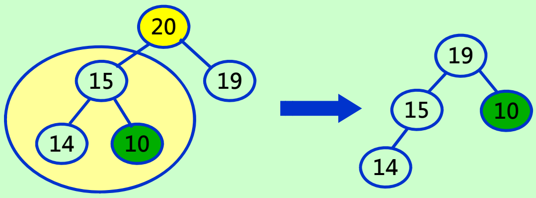


Deletion From a Max Heap

```
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