CSIEB0100 Data Structures

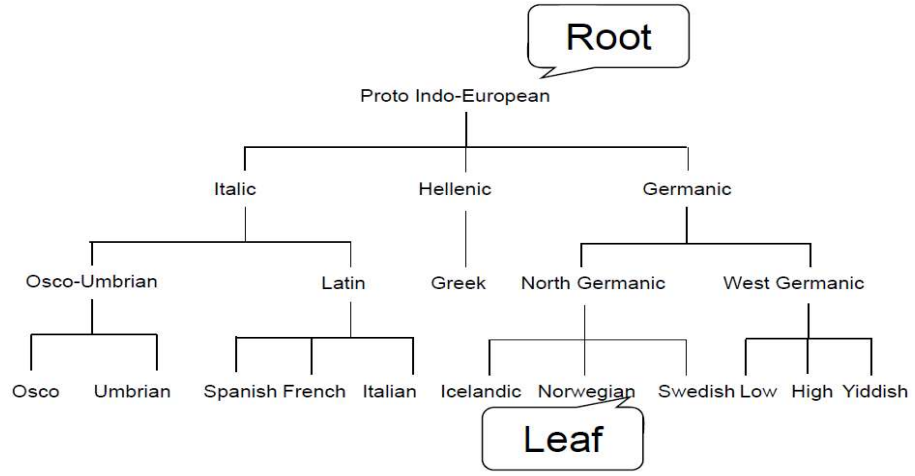
Lecture 06 Trees

Shiow-yang Wu 吳秀陽
Department of Computer Science and Information Engineering
National Dong Hwa University

Lecture material is partly home-grown, partly taken from slides came with the textbook originally prepared by Professor Jiun-Long Huang of NCTU.

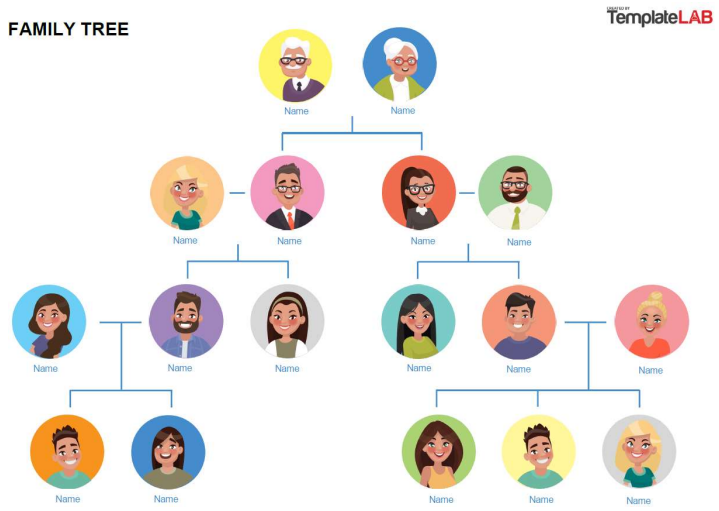
Language Tree

- The Proto-Indo-European language(原始印歐語系) tree



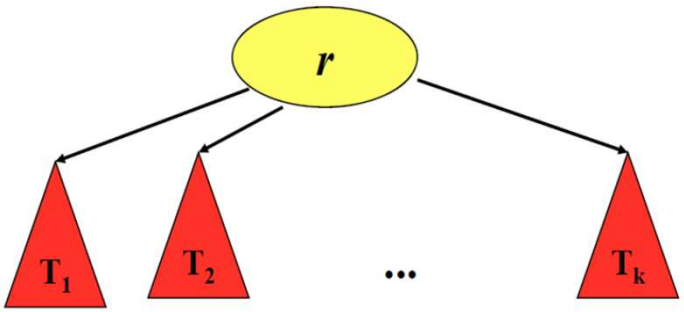
Family Tree

- We are all familiar with family tree.



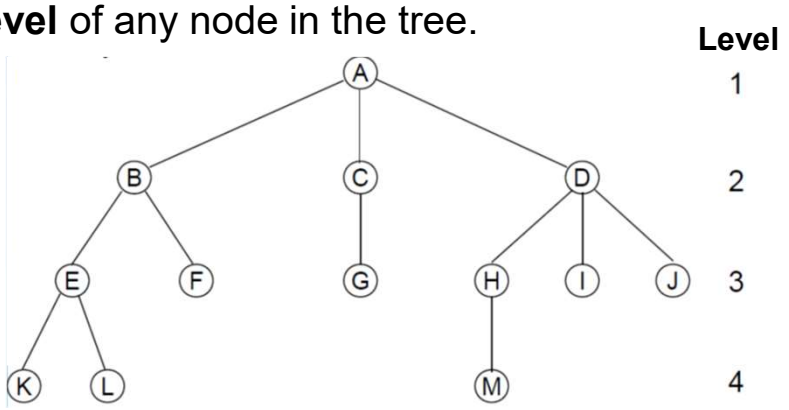
Trees - Definition

- Tree: a finite set of one or more nodes such that
  - a distinguished node ***r*** (root)
  - zero or more nonempty (sub)trees T1, T2, …, Tk
  - each of whose roots are connected by a directed edge from r



A Sample Tree

- Assume the root is at level 1, then the level of a node is the level of the node's parent plus one.
- The height or the depth of a tree is the **maximum level** of any node in the tree.

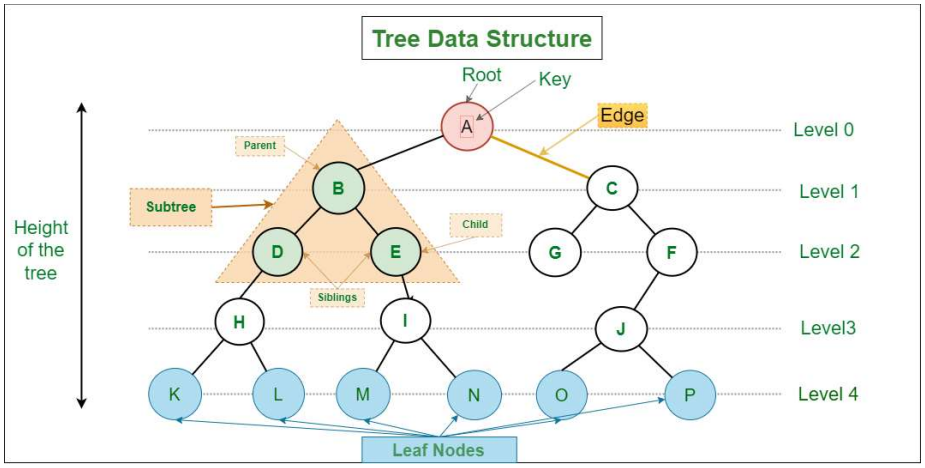


Terminologies

- The degree of a node is the **# subtrees** of the node.
  - The degree of A is 3; the degree of C is 1.
- The node with degree 0 is a leaf or terminal node.
- A node that has subtrees is the *parent* of the roots of the subtrees.
- The roots of these subtrees are the *children* of the node.
- Children of the same parent are *siblings*.
- The ancestors of a node are all the nodes along the **path** from the **root** to **the node**.
- The descendants of a node are all nodes along the **path** from the **leaf** node to **the node**.

Tree Data Structure

- In some definition, the level starts with 0.



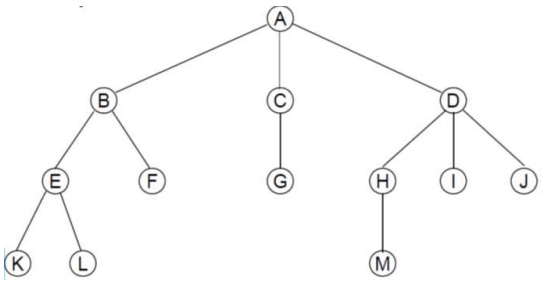
More Terminologies

- A path in a tree is a sequence of (0 or more) connected nodes.
- The length of a path is the # nodes in the path.
- The height of a tree is the length of the longest path from the root to a leaf.
- The depth of a node is the length of the path from the root to that node.
- The diameter of a tree is the length of the longest path between any tow nodes. (They must be two leaf nodes. Why?)

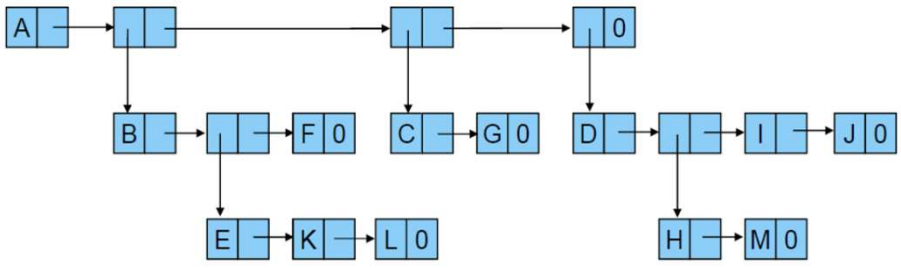
Representation of Trees

- List Representation
  - The root comes first, followed by a list of sub-trees
  - T=(root (T_1, T_2,…,T_n))

(A(B(E(K,L),F),C(G),D(H(M),I,J)))



List Representation of Trees



Node Structure

- Possible node structure of a tree of degree k

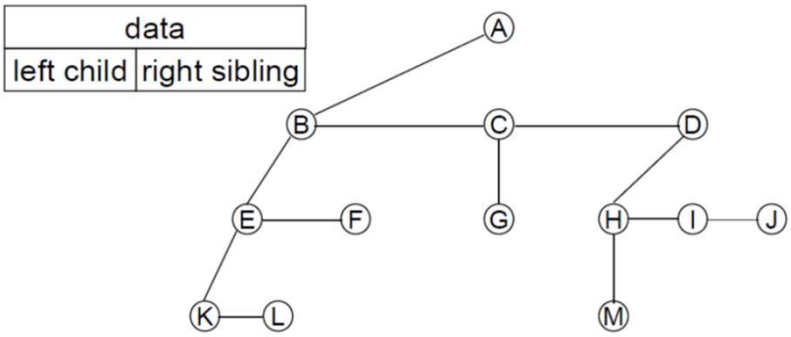
Data	Child 1	Child 2	Child 3	Child 4	...	Child k

- **Lemma 5.1**: If T is a *k*-ary tree (i.e., a tree of degree *k*) with *n* nodes, each having a fixed size as above, then *n(k-1) + 1* of the *nk* child fields are 0, $n \geq 1$. (why?)

$\Rightarrow$ Very wasteful of memory space!

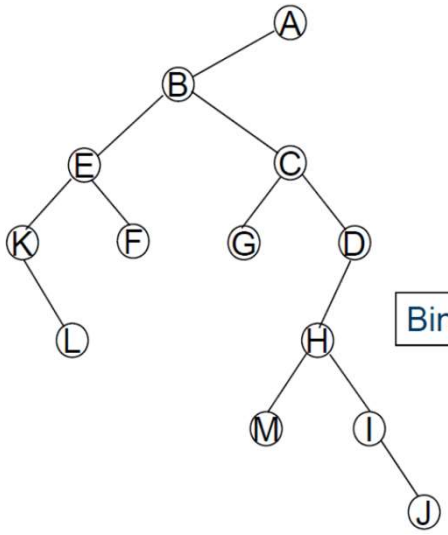
Representation of Trees

- Left Child-Right Sibling Representation
  - Each node has two links (or pointers).
  - Each node only has one leftmost child and one closest right sibling.



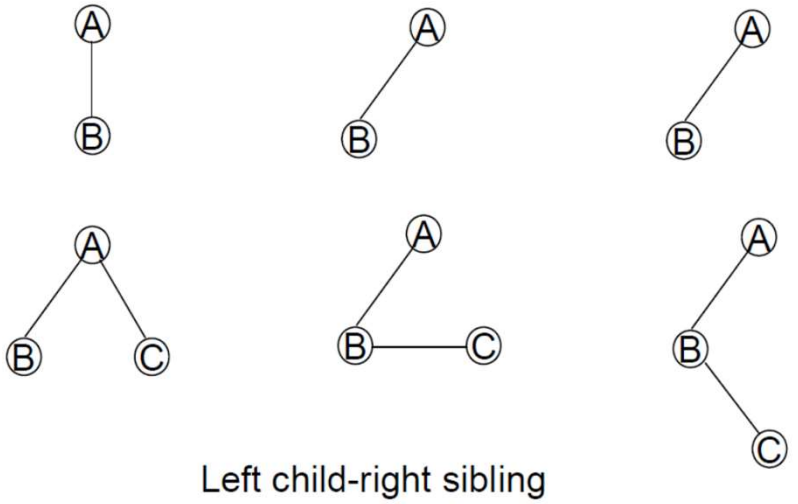
Representation of Trees

- Rotate the right sibling pointers in a left child right sibling tree by 45 degrees to get the left child-right child (or degree two) tree.



Binary Tree!

More Examples of Tree Representations



Left child-right sibling

Binary tree

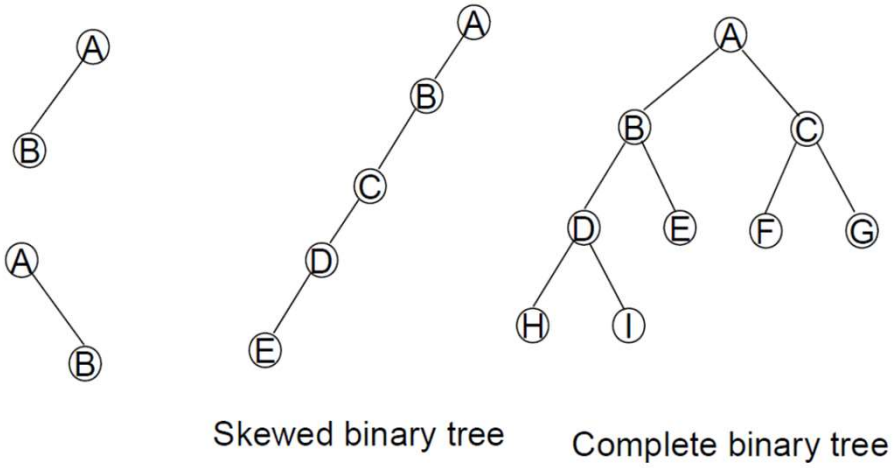
Binary Trees

- **Definition**:
  - A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.
- There is no tree with zero nodes. But there is an empty binary tree. (Quiz: How to modify the tree definition to allow empty tree?)
- Binary tree distinguishes between the order of the children while in a tree we do not.

Binart Tree vs Tree

	Binary tree	Tree
degree	≤ 2	Not limited
order of the subtrees	✓	×
allow zero nodes	✓	×

Binary Tree Examples



Properties of Binary Trees

- **Lemma 5.2** [Maximum number of nodes]
  - The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.
  - The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.
- **Lemma 5.3** [Relation between number of leaf nodes and nodes of degree 2]: For any non-empty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2 + 1$.
- **Definition**: A full binary tree of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$.

Nodes in Binary Trees

- The maximum number of nodes on level i of a binary tree is 2^{i-1}, i>=1.
- The maximum number of nodes in a binary tree of depth k is 2^k-1, k>=1.

Prove by induction

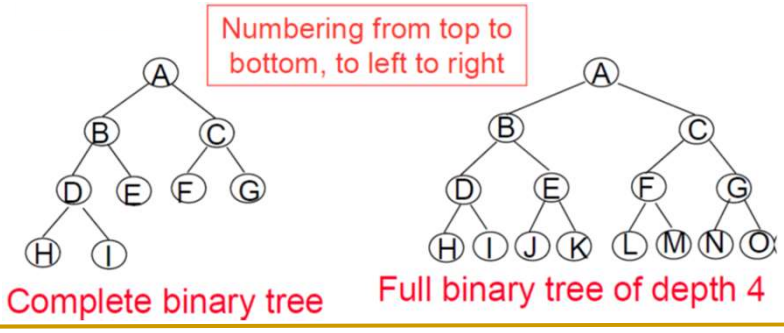
$$\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

Leaf Nodes & Nodes of Degree 2

- 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 and B denote the total number of nodes & branches in T.
  - Let n_0, n_1, n_2 represent the nodes with no children, single child, and two children respectively.
  - $n= n_0 + n_1 + n_2$, $B+1=n$, $B= n_1 +2n_2$ ==> $n_1 +2n_2 +1= n$,
  - $n_1 +2n_2 +1= n_0 + n_1 + n_2$ ==> $n_0 = n_2 +1$

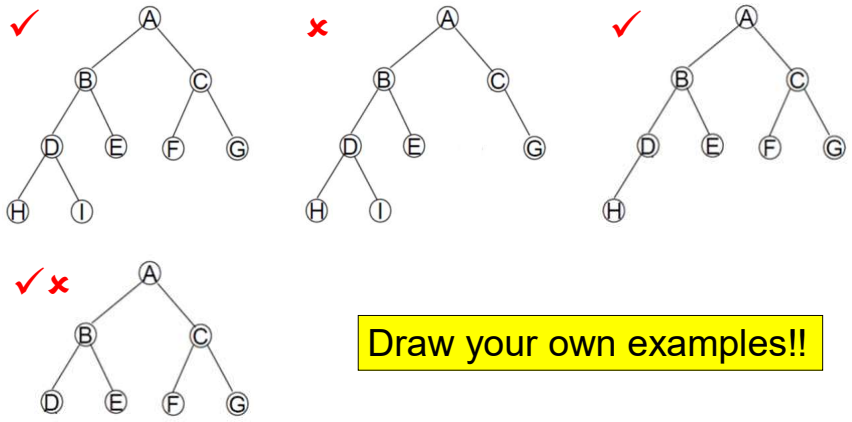
Full BT vs Complete BT

- A full binary tree of depth *k* is a BT of depth k having 2^k-1 nodes, $k>=0$
- A BT with *n* nodes and depth *k* is complete *iff* its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of depth *k*.



Examples of Complete BTs

- Which of the following BTs are complete BTs of depth 4?

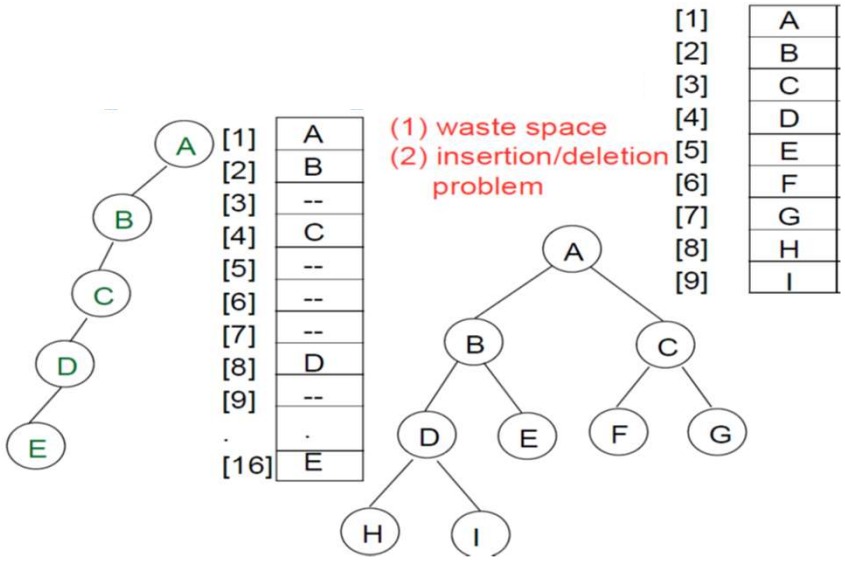


Draw your own examples!!

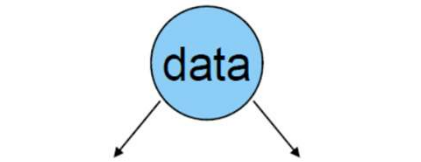
Array Representation of BT

- **Lemma 5.4**: If a complete binary tree with *n* nodes is represented sequentially, then for any node with index *i*, $1 \le i \le n$, we have:
  - parent(*i*) is at $\lfloor i / 2 \rfloor$ if $i \neq 1$. If $i = 1$, *i* is at the root and has no parent.
  - left_child(*i*) is at $2i$ if $2i \le n$. If 2i > n, then i has no left child.
  - right_child(*i*) is at $2i + 1$ if $2i + 1 \le n$. If $2i + 1 > n$, then *i* has no right child.
- Position zero of the array is not used.

Sequential Representation



Node Representation

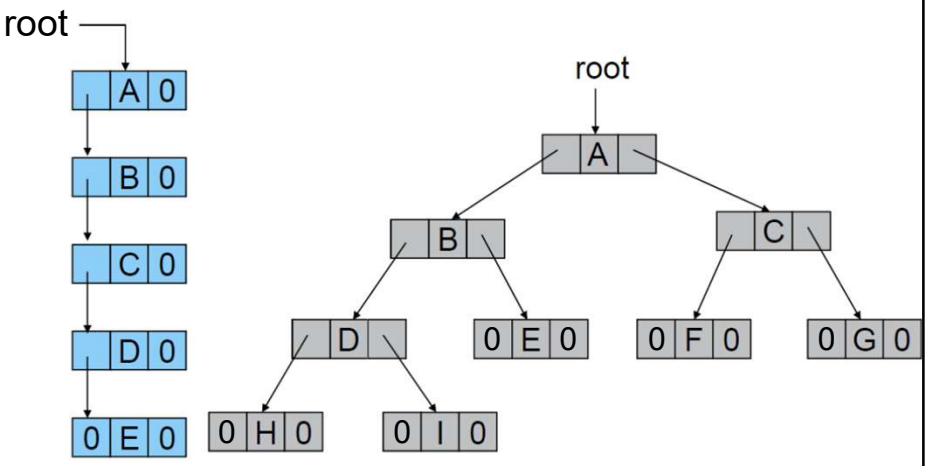


Linked Representation

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

class Tree {
  public:
  // Tree operations
  ...
  private:
    TreeNode *root;
};
```

Linked Representation for BT



For the tree in Figure 5.10(a)

For the tree in Figure 5.10(b)

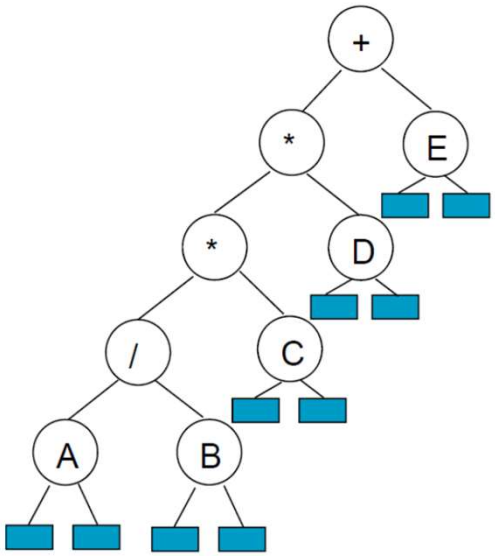
Compare Two BT Representation

	Array representation	Linked representation
Determination the locations of the parent, left child and right child	Easy	Difficult
Space overhead	Much	Little
Insertion and deletion	Difficult	Easy

Binary Tree Traversal

- Let L, V, and R stand for moving left, visiting the node, and moving right.
- There are six possible combinations of traversal
  - LVR, LRV, VLR, VRL, RVL, RLV
- Adopt the convention that we traverse left before right, only 3 traversals remain
  - L**V**R, LR**V**, **V**LR
  - inorder, postorder, preorder

Arithmetic Expression using BT



inorder traversal
A / B * C * D + E

preorder traversal
+ * * / A B C D E

postorder traversal
A B / C * D * E +

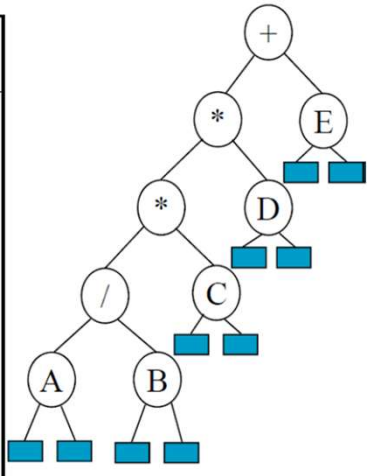
level order traversal
+ * E * D / C A B

Program 5.1

```
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 Operation of Inorder Traversal

Call of inorder	Value in root	Action	Call of inorder	Value in root	Action
1	+		11	C	
2	*		12	NULL	
3	*		11	C	cout
4	/		13	NULL	
5	A		2	*	cout
6	NULL		14	D	
5	A	cout	15	NULL	
7	NULL		14	D	cout
4	/	cout	16	NULL	
8	B		1	+	cout
9	NULL		17	E	
8	B	cout	18	NULL	
10	NULL		17	E	cout
3	*	cout	19	NULL	



Program 5.2

```
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 function of Tree.
{
  if(CurrentNode){
    cout << CurrentNode->data;
    preorder(CurrentNode->LeftChild);
    preorder(CurrentNode->RightChild);
  }
}
```


Program 5.3

```
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()
// nonrecursive inorder traversal using a stack
{
  Stack<TreeNode *> s; // declare and initialize stack
  TreeNode *CurrentNode = root;
  while (1) {
    while (CurrentNode) { // move down LeftChild fields
      s.Add(CurrentNode); // add to stack
      CurrentNode = CurrentNode->LeftChild;
    }
    if (!s.IsEmpty()) { // stack is not empty
      CurrentNode = *s.Delete(CurrentNode);
      cout << CurrentNode->data << endl;
      CurrentNode = CurrentNode->RightChild;
    }
    else break;
  }
}
```


Level Order Traversal

- All previous mentioned schemes use stacks
- Level-order traversal uses a queue
- Level-order scheme visit the root first, then the root's left child, followed by the root's right child
- All the nodes at a level are visited before moving down to another level

Level Order Traversal of BT

```
void Tree::LevelOrder()
// Traverse the binary tree in level order
{
  Queue<TreeNode *> q;
  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();
  }
}
```

+*E*D/CAB

Other BT Functions

- With the inorder, postorder, or preorder mechanisms, we can implement all needed binary tree functions. e.g.,
  - Copying Binary Trees
  - Testing Equality
    - Two binary trees are equal if their topologies are the same and the information in corresponding nodes is identical.

Program 5.9

```
//Copy constructor
Tree::Tree(const Tree& s) //driver
{
  root = copy(s.root);
}
TreeNode* Tree::copy(TreeNode *orignode)
//Workhorse
//This function returns a pointer to an exact copy of the binary
//tree rooted at orignode.
{
  if (orignode) {
    TreeNode *temp = new TreeNode;
    temp->data = orignode->data;
    temp->LeftChild = copy(orignode->LeftChild);
    temp->RightChild = copy(orignode->RightChild);
    return temp;
  }
  else return 0;
}
```


Program 5.10

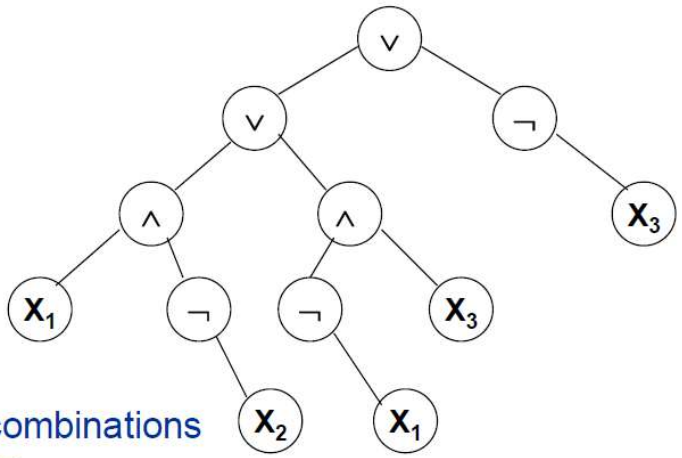
```
//Driver-assumed to be a friend of class Tree.
int operator==(const Tree& s, const Tree& t)
{
  return equal(s.root, t.root);
}
//Workhorse-assumed to be a friend of TreeNode.
int equal(TreeNode *a, TreeNode *b)
//This function returns 0 if the subtrees at a and b are not
//equivalent. Otherwise, it will return 1.
{
  if((!a)&&(!b)) return 1; //both a and b are 0
  if(a && b //both a and b are non-0
       && (a->data == b->data) //data is the same
       && equal(a->LeftChild, b->LeftChild) //L subtrees eql
       && equal(a->RightChild, b->RightChild)) //R subtrees eql
    return 1;
  return 0;
}
```

Propositional Calculus Expression

- A variable is an expression.
- If x and y are expressions, then ¬x, x∧y, x∨y are expressions.
- Parentheses can be used to alter the normal order of evaluation (¬ > ∧ > ∨).
- Example: x1 ∨ (x2 ∧ ¬x3)
- Satisfiability problem: Is there an assignment of variables to make an expression true?

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$

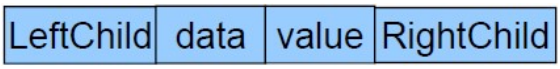
(t,t,t)
(t,t,f)
(t,f,t)
(t,f,f)
(f,t,t)
(f,t,f)
(f,f,t)
(f,f,f)



2^n possible combinations for n variables

Perform Formula Evaluation

- To evaluate an expression, we can traverse its tree in postorder.
- To perform evaluation, can add a value member so that each node has four fields
  - LeftChild
  - data
  - value
  - RightChild



Satisfiability Algorithm (V1)

For all 2^n possible truth value combinations for the n variables

```
{
  generate the next combination;
  replace the variables by their values;
  evaluate the formula by traversing the tree it points to
  in postorder;
  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 LogicalNot:
        s->value = !s->RightChild->value;
        break;
      case LogicalAnd:
        s->value = s->LeftChild->value && s->RightChild->value;
        break;
      case LogicalOr:
        s->value = s->LeftChild->value || s->RightChild->value;
        break;
      case LogicalTrue: s->value = TRUE; break;
      case LogicalFalse: s->value = FALSE;
    }
  }
}
```

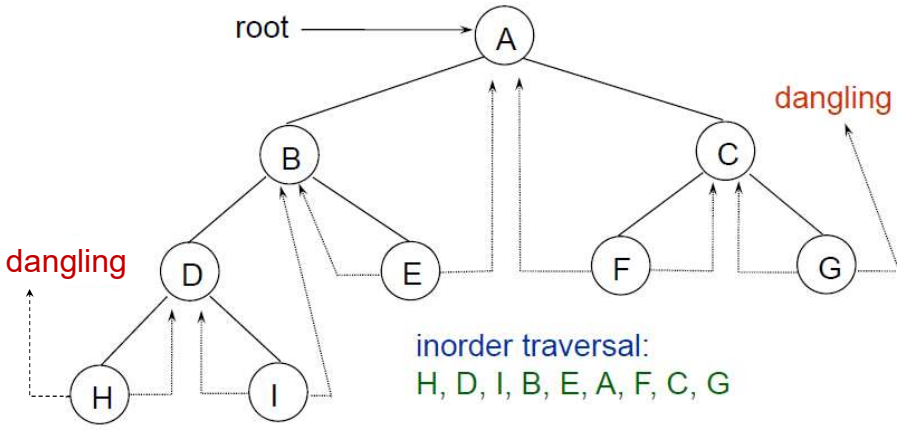

Threaded Binary Trees

- Too many null pointers in current representation of binary trees
  - n: number of nodes
  - number of non-null links: n-1
  - total links: 2n
  - null links: 2n-(n-1)=n+1
- Replace these null pointers with some useful "threads".

Threaded Binary Trees

- If ptr->left_child is null,
  - replace it with a pointer to the node that would be visited *before ptr in an inorder traversal* (inorder predecessor)
- If ptr->right_child is null,
  - replace it with a pointer to the node that would be visited *after ptr in an inorder traversal* (inorder successor)

A Threaded Binary Tree



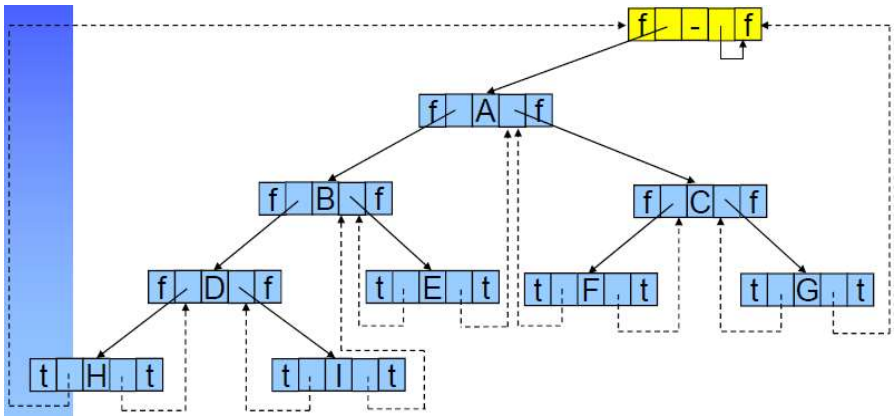
Threads

- To distinguish between normal pointers and threads, two boolean fields, LeftThread and RightThread, are added to the record in memory representation.
  - t->LeftThread = TRUE
    => t->LeftChild is a **thread**
  - t->LeftThread = FALSE
    => t->LeftChild is a **pointer to the left child.**

Threads

- To avoid dangling threads, a head node is used in representing a binary tree.
- The original tree becomes the left subtree of the head node.
- Empty Binary Tree

Threaded Tree of Fig 5.20



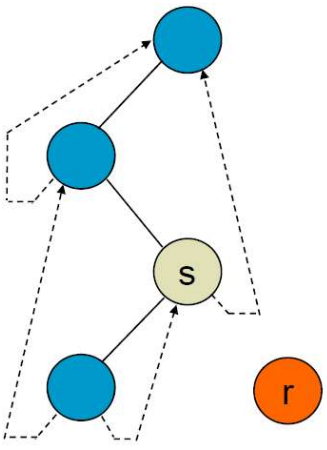

```
program 5.14
char* ThreadedInorderIterator::Next()
// Find the inorder successor of CurrentNode in a threaded
// binary tree
{
  ThreadedNode *temp = CurrentNode->RightChild;
  if(!CurrentNode->RightThread)
    while(!temp->LeftThread)
      temp = temp->LeftChild;
  CurrentNode = temp;
  if(CurrentNode == t.root) return 0;
  else return &CurrentNode->data;
}
program 5.15
void ThreadedInorderIterator::Inorder()
{
  for(char *ch = Next(); ch ; ch = Next())
    cout << *ch << endl;
}
```

Inorder travesal can be performaed without stack

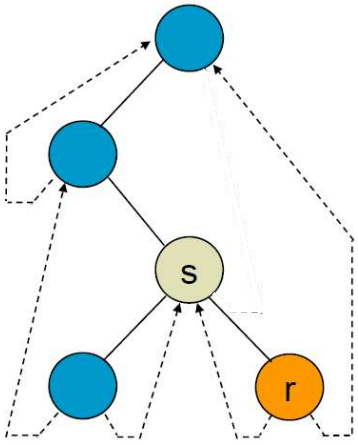
Inserting a Node to a TBT

- Inserting a node **r** as the right child of a node **s.**
  - If **s** has an empty right subtree, then the insertion is simple and diagram in Figure 5.23(a).
  - If the right subtree of **s** is not empty, then this right subtree is made the right subtree of **r** after insertion. When this is done, **r** becomes the inorder predecessor of a node that has a LeftThread==TRUE field, and consequently there is a thread which has to be updated to point to **r**. The node containing this thread was previously the inorder successor of **s**. Figure 5.23(b) illustrates the insertion for this case.

Inserting a Node to a TBT

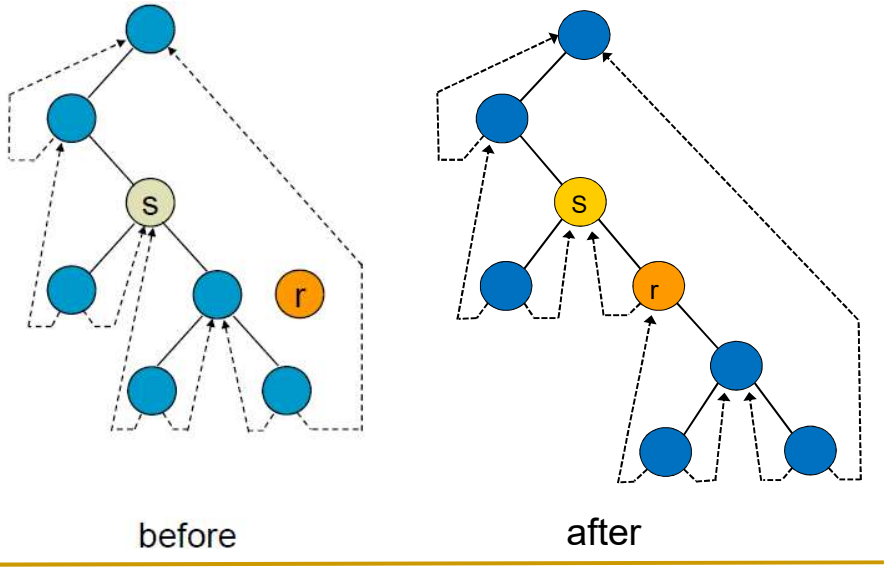


before



after

Inserting a Node to a TBT




```
void ThreadedTree::InsertRight(ThreadNode *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; // RightChild is a node
  if (!r->RightThread) {
    // gets the inorder successor of r
    ThreadedNode *temp = InorderSucc(r);
    temp->LeftChild = r;
  }
}
```

Priority Queues

- In a priority queue, the element to be deleted is the one with **highest** (or **lowest**) priority.
- An element with arbitrary priority can be inserted into the queue according to its priority.
- A data structure supports the above two operations is called **max** (**min**) priority queue.

Applications of Priority Queue

- **machine service**
  - amount of time (min heap)
  - amount of payment (max heap)

Data Structures for Priority Queues

- Unordered linked list
- Unordered array
- Sorted linked list
- Sorted array
- Heap

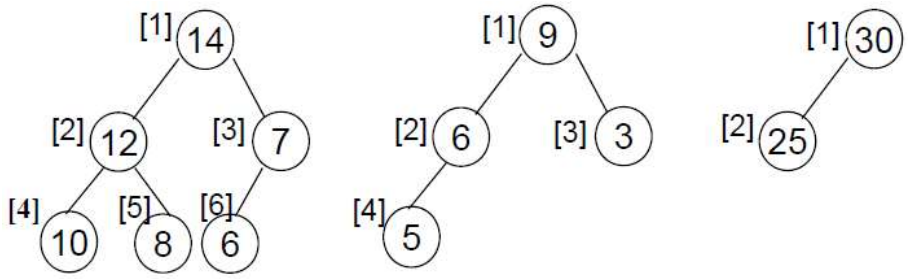
Priority Queue Representation

Representation	Insertion	Deletion
Unordered array	$\Theta(1)$	$\Theta(n)$
Unordered linked list	$\Theta(1)$	$\Theta(n)$
Sorted array	$O(n)$	$\Theta(1)$
Sorted list	$O(n)$	$\Theta(1)$
Max heap	$O(\log_2 n)$	$O(\log_2 n)$

Max (Min) Heap

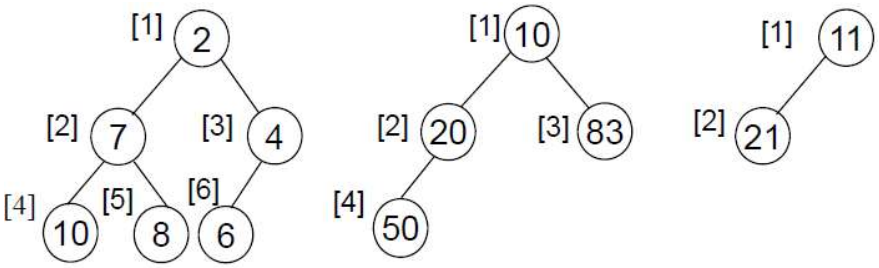
- Heaps are frequently used to implement priority queues. The complexity is **O(log n).**
- Definition:
  - A max (min) tree is a tree in which the key value in each node is no smaller (larger) than the key values in its children (if any).
  - A **max heap** is a **complete binary tree** that is also a max tree.
  - A **min heap** is a **complete binary tree** that is also a min tree.

Max Heap Examples



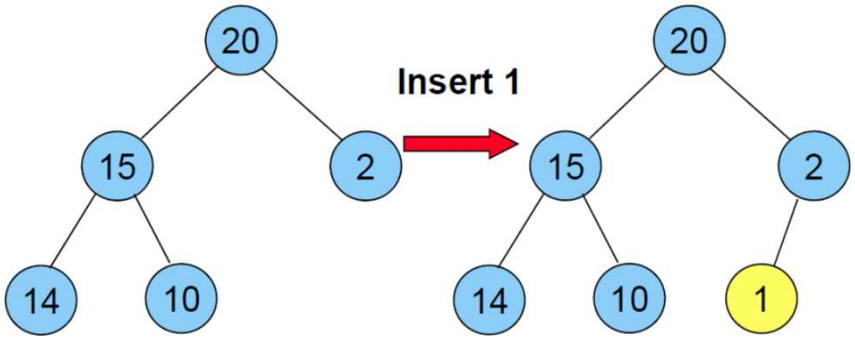
- Property: The root of max heap contains the largest.
- Can you provide more **examples** and **nonexamples**?

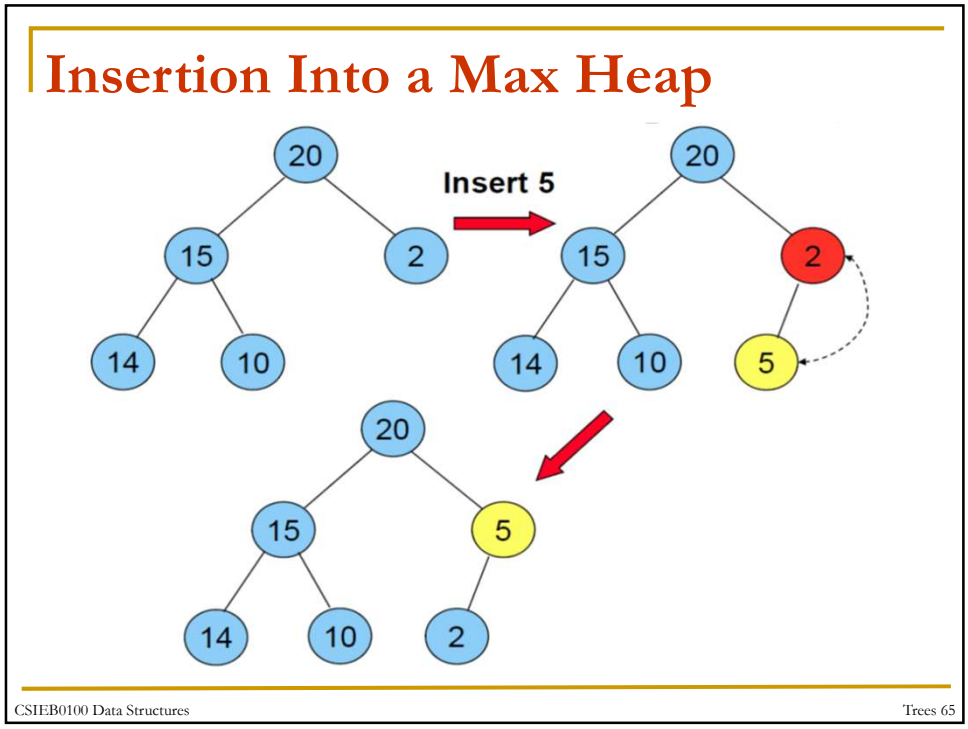
Min Heap Examples

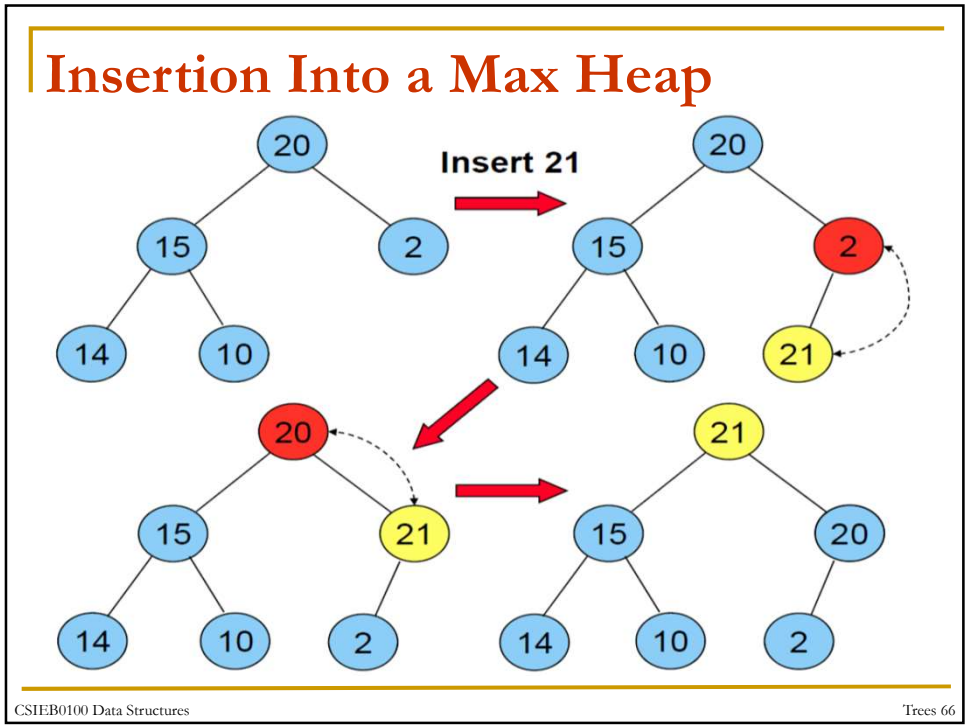


- Property: The root of min heap contains the smallest.
- Can you provide more **examples** and **nonexamples**?

Insertion Into a Max Heap




Insertion Into a Max Heap
Insert 5
20
15
2
14
10
20
15
2
14
10
5
20
15
5
14
10
2
CSIEB0100 Data Structures
Trees 65

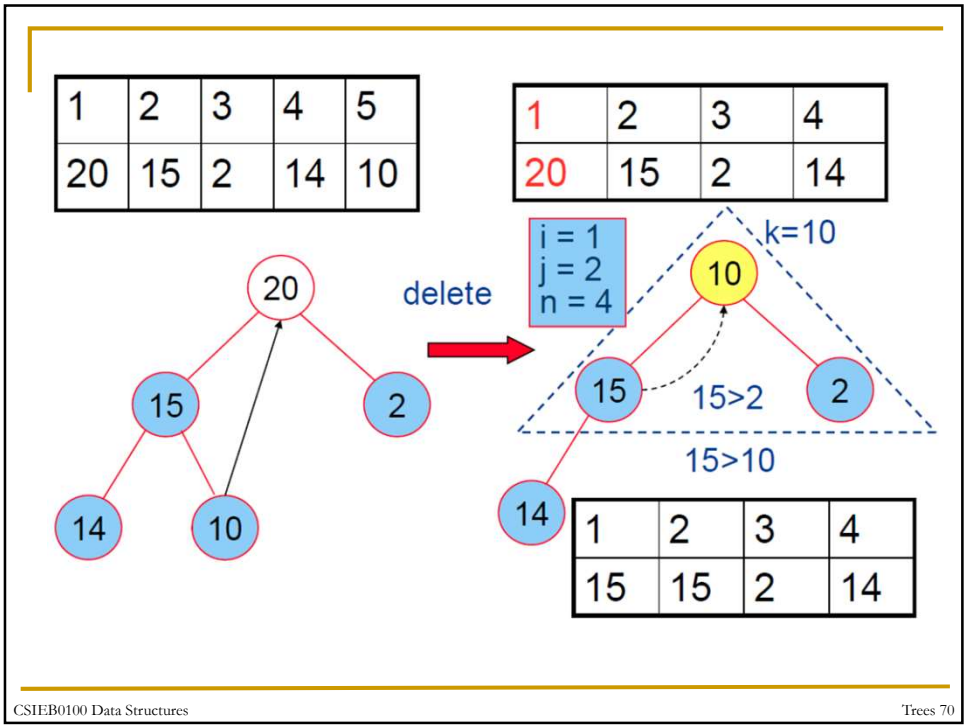

Insertion Into a Max Heap
Insert 21
20
15
2
14
10
20
15
2
14
10
21
20
15
21
14
10
2
21
15
20
14
10
2
CSIEB0100 Data Structures
Trees 66

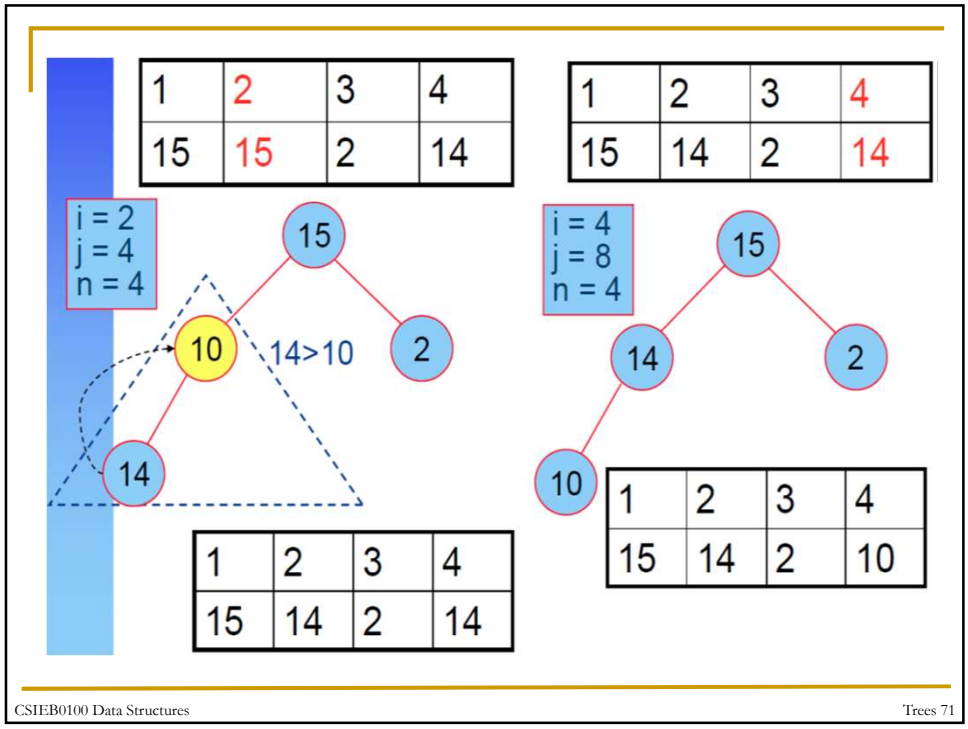
```
template <class Type>
void MaxHeap<Type>::Insert(const Element <Type> &x)
// insert x into the max heap
{
  if(n == MaxSize) {HeapFull(); return;}
  n++;
  int i;
  for(i = n; 1; ){
    if(i == 1) break; // at root
    if(x.key <= heap[i/2].key) break;
    // move from parent to i
    heap[i] = heap[i/2];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


Deletion from a Max Heap

```
template <class Type>
Element<Type>*
MaxHeap<Type>::DeleteMax(Element<Type>& x)
// Delete from the max heap
{
  if (!n) { HeapEmpty(); return 0; }
  x = heap[1];
  Element<Type> k = heap[n];
  n--;
  int i, j;
  for (i=1, j=2; j <= n;) {
    // i is the tentative location of k
    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;
    }
    heap[i] = k;
    return &x;
  }
```

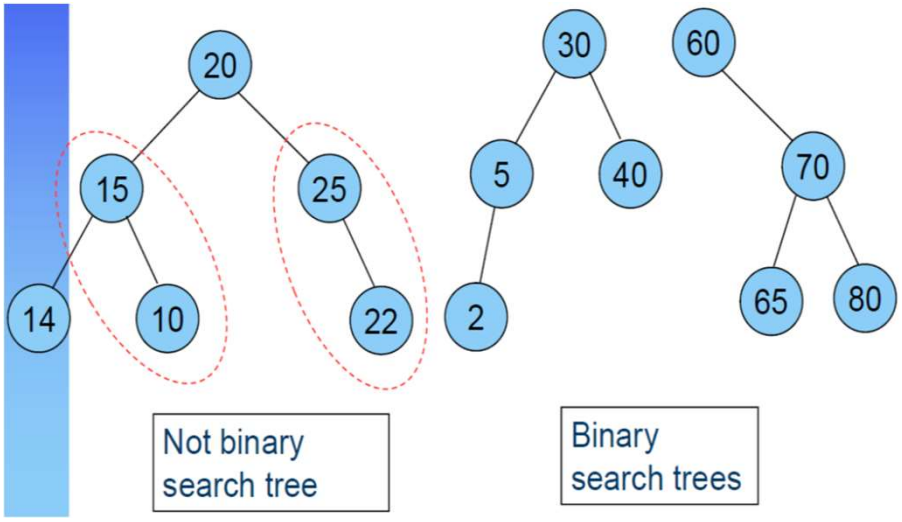





Binary Search Tree (BST)

- Heap
  - a min (max) element is deleted. $O(\log_2 n)$
  - deletion of an arbitrary element $O(n)$
  - search for an arbitrary element $O(n)$
- Binary search tree
  - Every element has a unique key.
  - The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
  - The left and right subtrees are also binary search trees.

Binary Trees



Searching a Binary Search Tree

- If the root is null, then this is an empty tree. No search is needed.
- If the root is not null, compare the *x* with the key of root.
  - If *x* is equal to the key of the root, then it's done.
  - If *x* is less than the key of the root, then no elements in the right subtree have key value *x*. We only need to search the left subtree.
  - If *x* is larger than the key of the root, only the right subtree is to be searched.

```
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,
const 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);
}  //recursive version
```

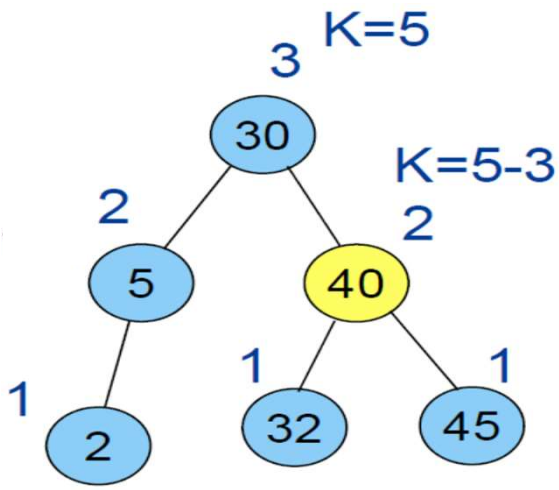
```
template <class Type>
BstNode<Type>* BST<Type>::IterSearch(const
Element<Type>& x)
/* Search the binary search tree for an element
with key x */
{
  for(BstNode<Type> *t = root; t; )
  {
    if(x.key == t->data.key) return t;
    if(x.key < t->data.key) t = t->LeftChild;
    else t = t->RightChild;
  }
  return 0;
}  //Iterative version
```


Search Binary Search Tree by Rank

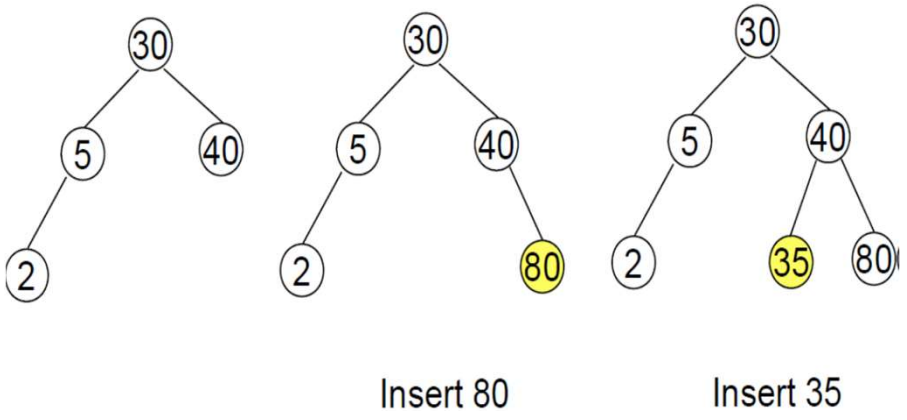
- Search by rank: eg. Find the kth smallest
- To search a binary search tree by the ranks of the elements in the tree, we need additional field *LeftSize*.
- LeftSize is **the number of the elements** in the **left subtree** of a node **plus one**.
- It is obvious that a binary search tree of height h can be searched by key as well as by rank in *O(h)* time.
  - What is the range of *h*?

```
template <class Type>
BstNode <Type>* BST<Type>::Search(int k)
// Search the BST for the kth smallest element
{
  BstNode<Type> *t = root;
  while(t)
  {
    if (k == t->LeftSize) return t;
    if (k < t->LeftSize) t = t->LeftChild;
    else {
      k -= t->LeftSize;
      t = t->RightChild;
    }
  }
  return 0;
}
```


Example of Searching by Rank



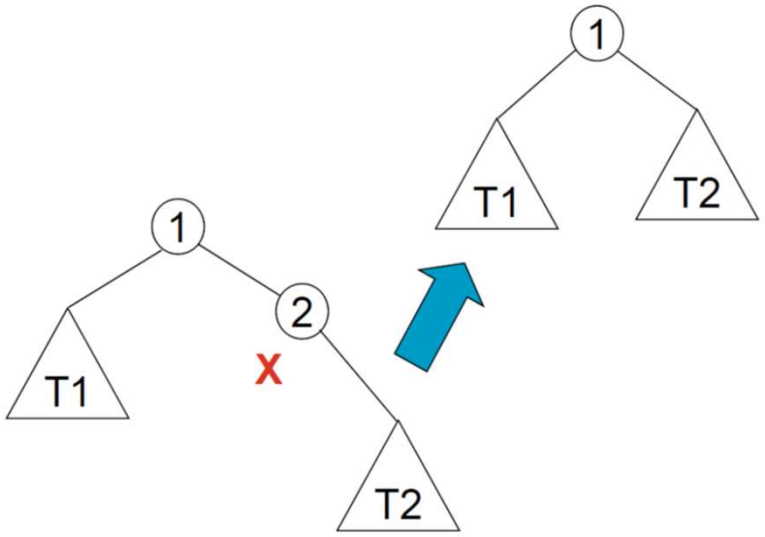
Inserting a Node into a BST



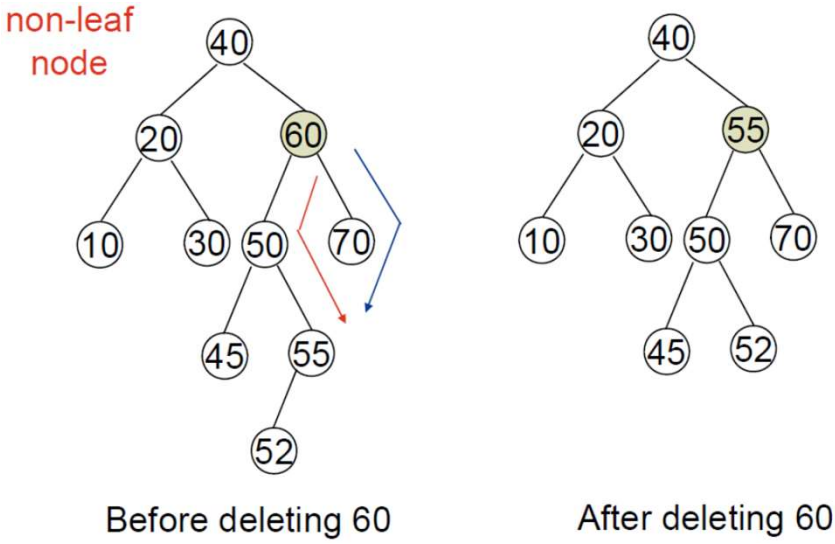
Deletion from a BST

- Deleting a node from a BST is more complicated.
- Two stages:
  - Search the node to remove.
  - If found, delete the node.
- Three cases to consider when deleting a node:
  - When the node has no children. (trivial, exercise)
  - When the node has one child. (simple)
  - When the node has two children. (two alternatives)

Deletion from a BST – One Child



Deletion from a BST – Two Children

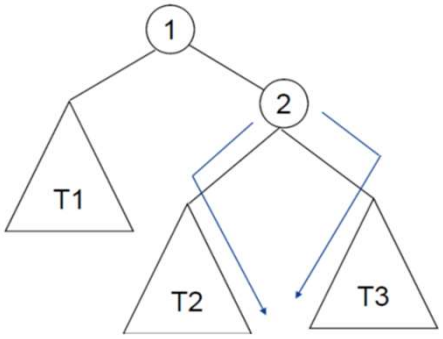


Deletion from a BST – Two Children

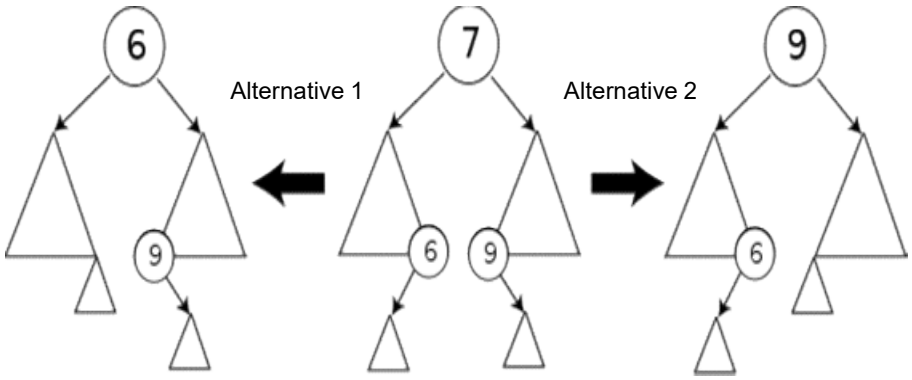
- Two alternatives. In both cases, we keep the node but replace its value.
- **Alternative 1**: Replace it with the **largest value** (say, of node N) in its **left subtree**. Then delete N.
- **Alternative 2**: Replace it with the **smallest value** (say, of node N) in its **right subtree**. Then delete N.
- We will discuss Alternative 1 and leave Alternative 2 for you as an exercise.

Deletion from a BST – Two Children

- The largest value in the left subtree must be the right most node of the left subtree. (Why?)
- The smallest value in the right subtree must be the left most node of the right subtree. (Why?)



Deletion from a BST – Two Children



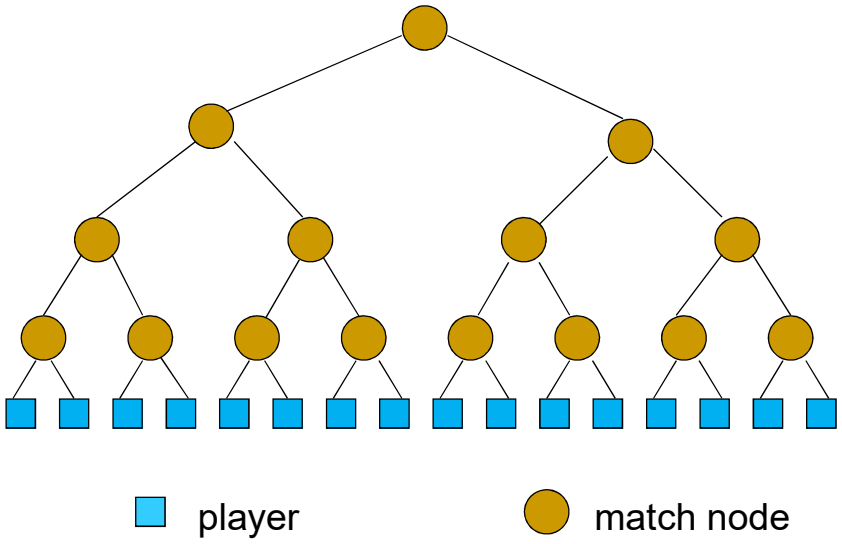
Selection Trees

- Winner tree
- Loser tree

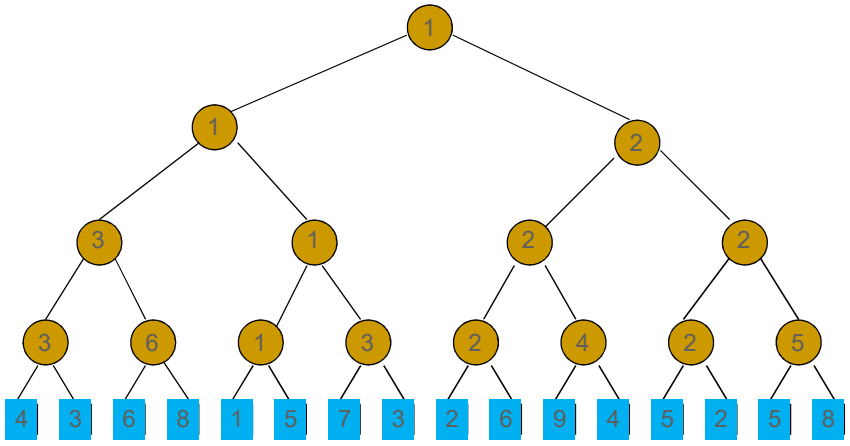
Winner Trees

- Complete binary tree with k external nodes and k - 1 internal nodes.
- External nodes represent tournament players.
- Each internal node represents a match played between its two children; the winner of the match is stored at the internal node.
- Root has overall winner.

Winner Tree For 16 Players

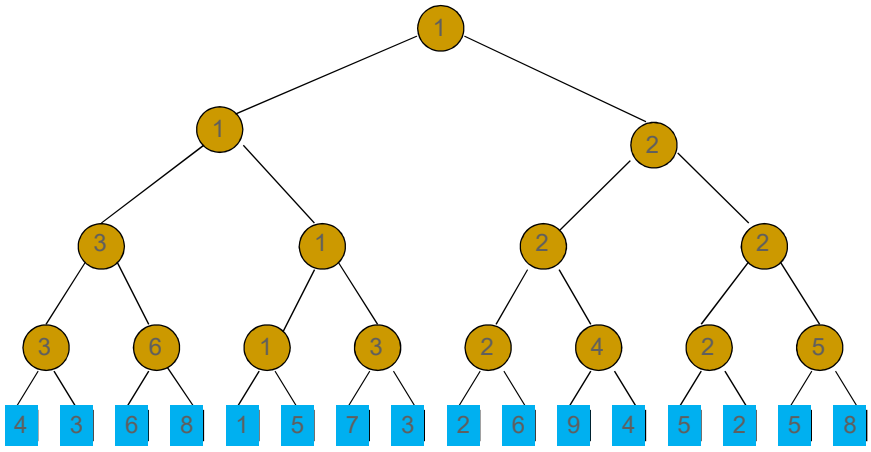


Winner Tree For 16 Players



Smaller element wins => min winner tree.

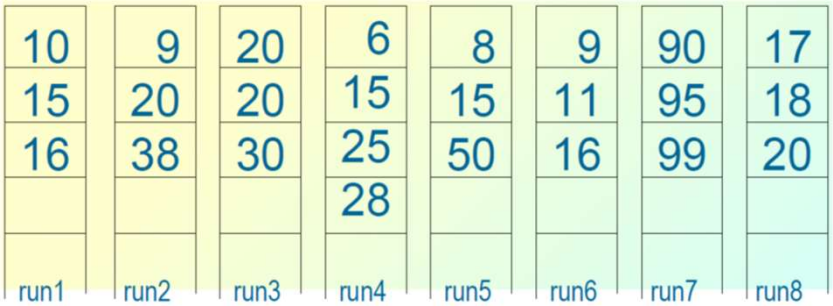
Winner Tree For 16 Players



height is $\log_2 k$ (excludes player level)

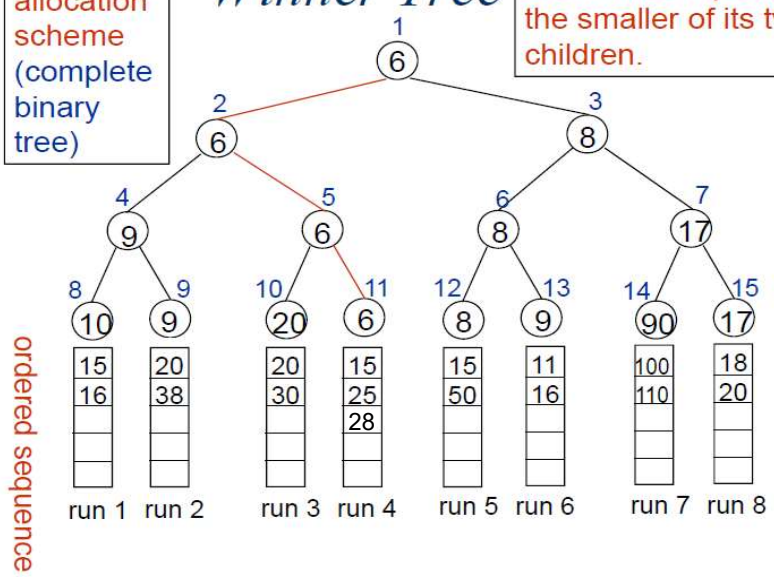
Merging Sequences with Winner Tree

- Given k ordered sequences (non-decreasing) called runs
- Want to merge these sequences into one sequence (of n numbers)

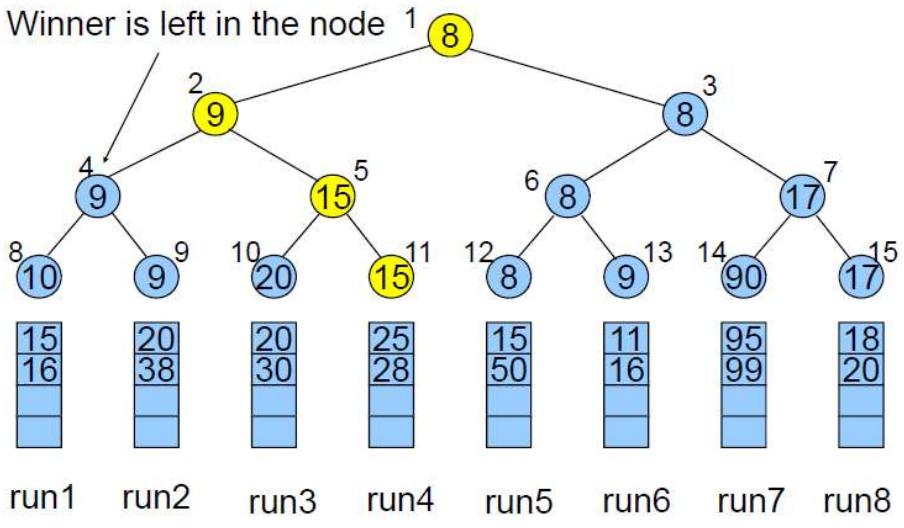


sequential allocation scheme (complete binary tree)

Each node represents the smaller of its two children.



Winner Tree for k = 8



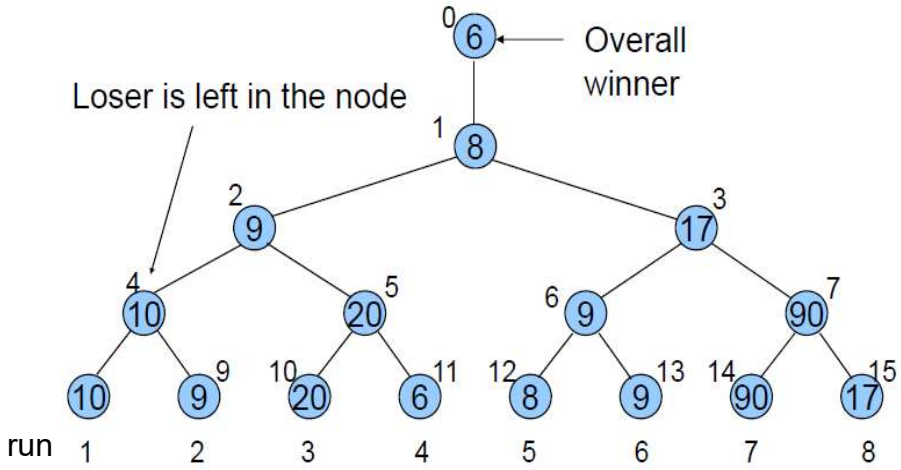
Analysis

- K: # of runs
- n: # of records
- setup time: O(K) (K-1)
- restructure time: $O(\log_2 K)$ $\lceil \log_2(K+1) \rceil$
- merge time: $O(n\log_2 K)$
- **slight** modification: tree of loser
  - consider the parent node only (vs. sibling nodes)

Loser Tree

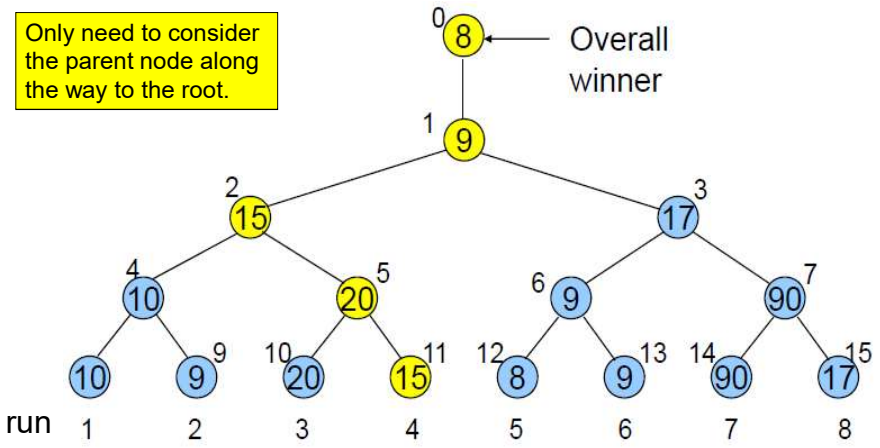
- Each match node stores the match loser rather than the match winner.
- Use the winner for matching at the next level.
- The winner at the root node is placed at an "extra" overall winner slot.

Loser Tree Merge



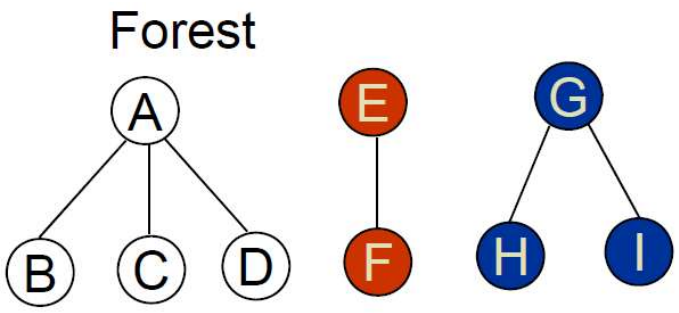
Loser Tree Merge

Only need to consider the parent node along the way to the root.



Forest

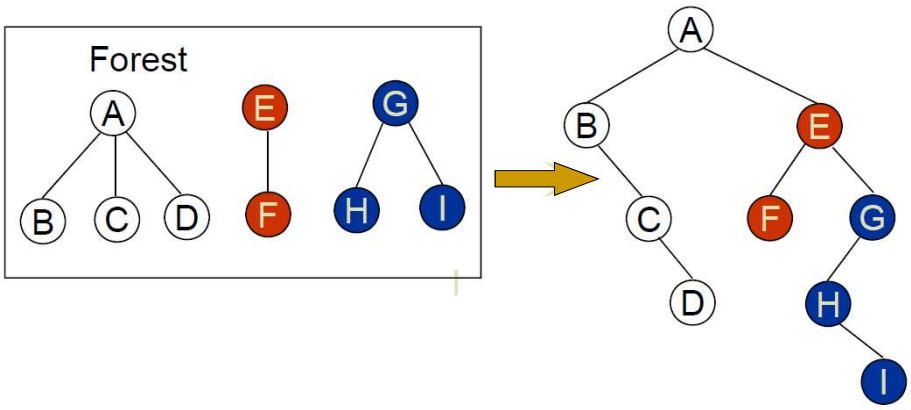
- A forest is a set of *n>=0 disjoint trees*.



Transform a Forest into a Binary Tree

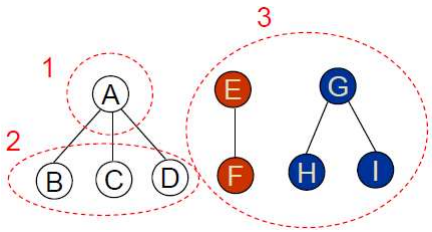
- *T1, T2, …, Tn:* a forest of trees
- *B(T1, T2, …, Tn):* a binary tree corresponding to this forest
- Algorithm
  - empty, if *n=0*
  - has root equal to *root(T1);* has left subtree equal to *B(T11,T12,…,T1m);* where *T11,T12,…,T1m* are subtrees of *root(T1);* and has right subtree equal to *B(T2,T3,…,Tn)*.

Transform a Forest into a Binary Tree



Forest Traversals

- Preorder
  - If F is empty, then return
  - Visit the root of the first tree of F
  - Traverse the subtrees of the first tree in forest preorder
  - Traverse the remaining trees of F in forest preorder

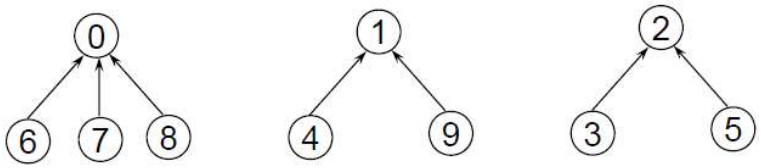


Forest Traversals

- Inorder
  - If F is empty, then return
  - Traverse the subtrees of the first tree in forest inorder
  - Visit the root of the first tree
  - Traverse the remaining trees of F in forest inorder
- Postorder
  - If F is empty, then return
  - Traverse the subtrees of the first tree in forest postorder
  - Traverse the remaining trees of F in forest postorder
  - Visit the root of the first tree

Set Representation

- S_1={0, 6, 7, 8}, S_2={1, 4, 9}, S_3={2, 3, 5}

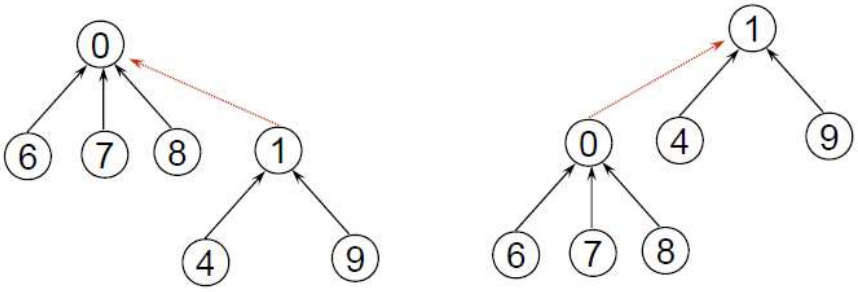


$S_i \cap S_j = \phi$

- Two operations considered here
  - *Disjoint set union* $S_1 \cup S_2$ = {0,6,7,8,1,4,9}
  - *Find(i):* Find the set containing the element i.
    - $3 \in S_3$, $8 \in S_1$

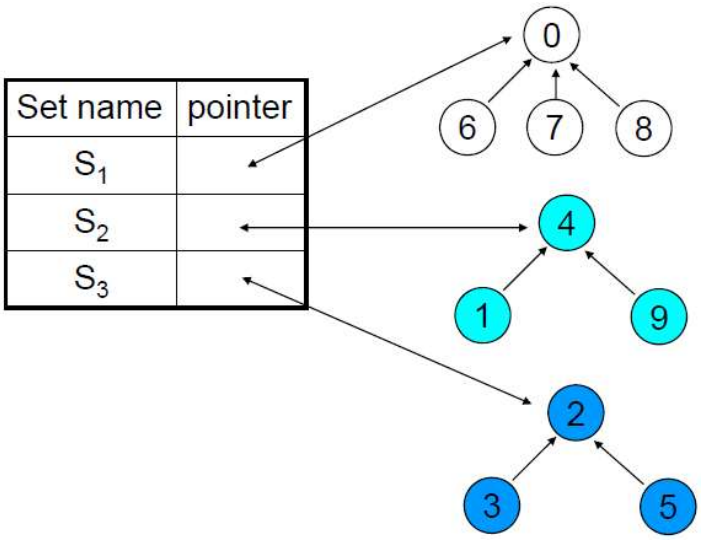
Disjoint Set Union

Make one of trees a subtree of the other



Possible representation for S_1 *union* S_2

Data Representation of S_1, S_2 and S_3



Array Representation of S_1, S_2, S_3

- We could use an array for the set name. Or the set name can be an element at the root.
- Assume set elements are numbered 0 through *n-1.*

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

root

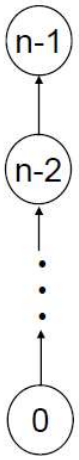
SimpleFind

```
SimpleFind(int i)
// Find the root of the tree containing
// element i
{
  while(parent[i]>=0)
    i=parent[i];
  return;
}
```

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

Degenerate Tree

union(0,1),
union(1,2),
.
.
.
union(n-2,n-1)
find(0),
find(1),
.
.
.
find(n-1)



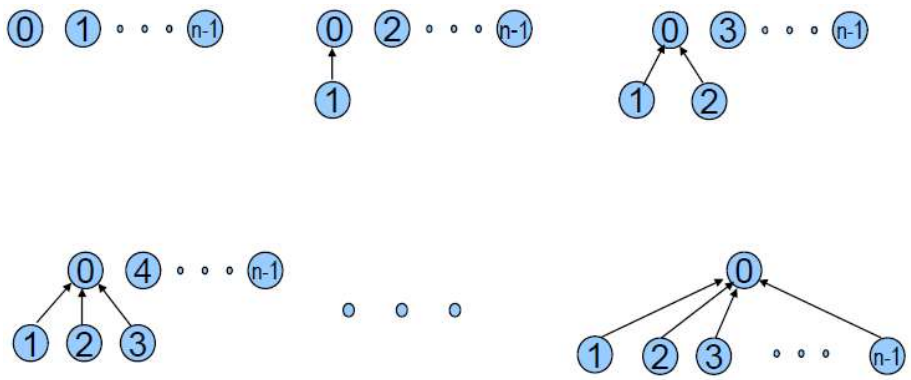
degenerate tree

- n-1 union operations
  - O(n)
- One union operation
  - O(1)
- n find operations
  - $O(n^2)$ $\sum_{i=2}^{n} i$
- One find operations
  - O(n)

Weighting Rule

- Weighting rule for *union(i, j)*
  - 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*.
- Use the weighting rule on the union operation to avoid the creation of degenerate trees.

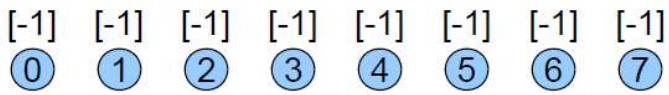
Trees Obtained Using The Weighting Rule



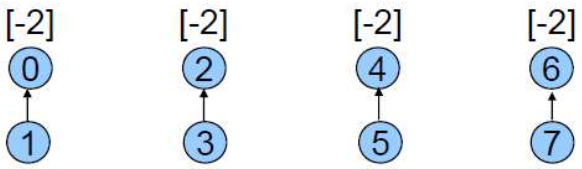
Weighted Union

- **Lemma 5.5**: Assume that we start with a forest of trees, each having one node. Let *T* be a tree with *m* nodes created as a result of a sequence of unions each performed using function WeightedUnion. The height of *T* is no greater than $\lfloor \log_2 m \rfloor + 1$
- For the processing of an intermixed sequence of *u–1* unions and f find operations, the time complexity is $O(u + f \times \log u)$.
  - No tree has more than *u* nodes in it.
  - We need O(n) additional time to initialize the n-tree forest

Can still have unbalance trees

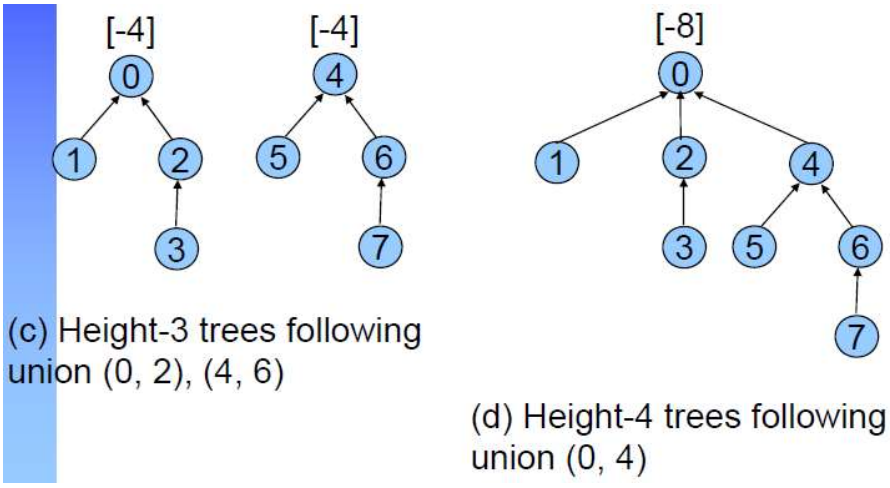


(a) Initial height trees



(b) Height-2 trees following union (0, 1), (2, 3), (4, 5), and (6, 7)

Can still have unbalance trees



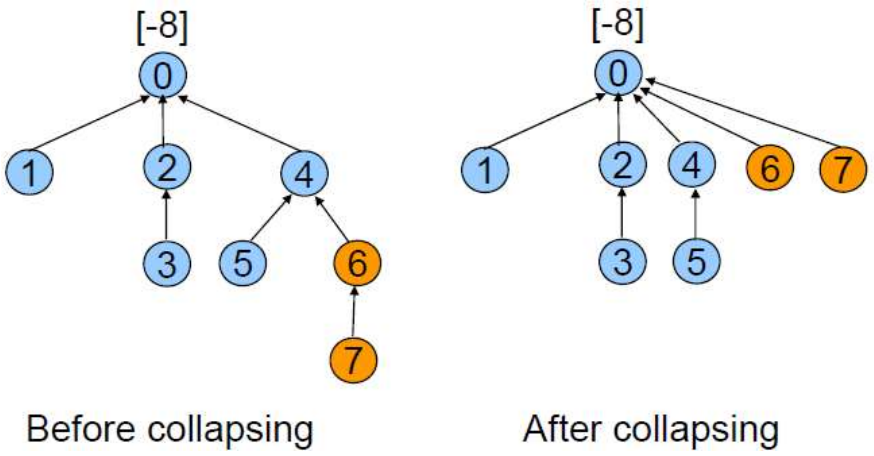
(c) Height-3 trees following union (0, 2), (4, 6)

(d) Height-4 trees following union (0, 4)

Collapsing Rule

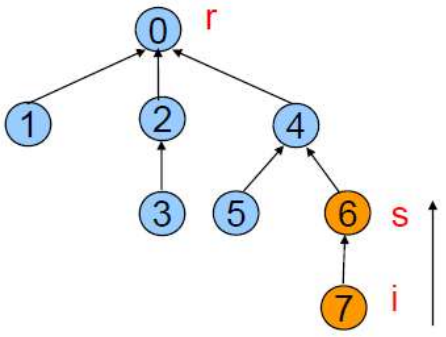
- Collapsing rule:
  - If *j* is a node on the path from *i* to its root and *parent[j]≠root(i),* then set *parent[j] to root(i).*
- The first run of find operation will collapse the tree. Therefore, each following find operation of the same element only goes up one link to find the root.

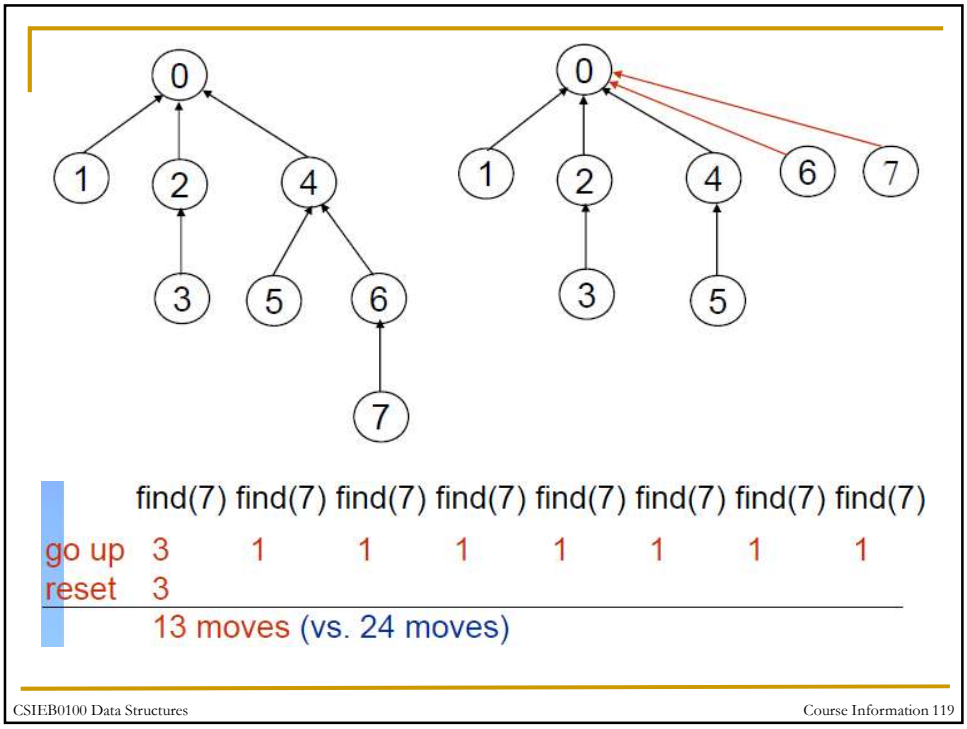
CollapsingFind



CollapsingFind

```
CollapsingFind(int i)
{
  // find root
  for(int r=i; parent[r]>=0; r=parent[r]);
  //collapse
  while(i!=r)
  {
    int s=parent[i];
    parent[i]=r;
    i=s;
  }
}
```





Applications

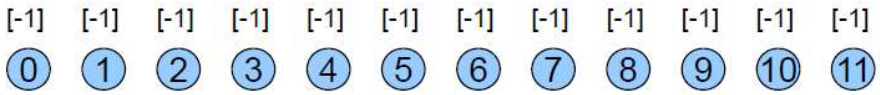
- Find equivalence class i ≡ j
- Find S_i and S_j such that i ∈ S_i and j ∈ S_j (two finds)
  - $S_i = S_j$ do nothing
  - $S_i \neq S_j$ union(S_i , S_j)
- Example:

  0 ≡ 4, 3 ≡ 1, 6 ≡ 10, 8 ≡ 9, 7 ≡ 4, 6 ≡ 8,

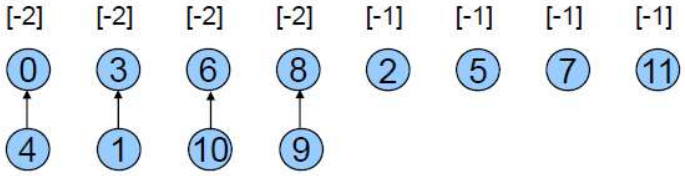
  3 ≡ 5, 2 ≡ 11, 11 ≡ 0

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

Example 5.5

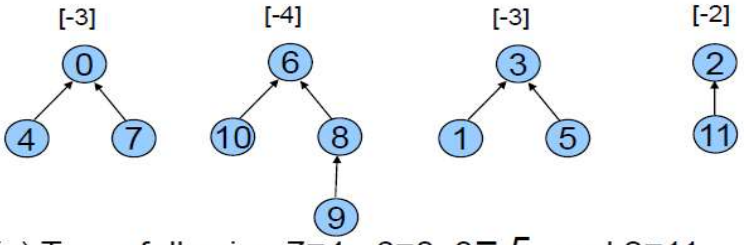


(a) Initial trees

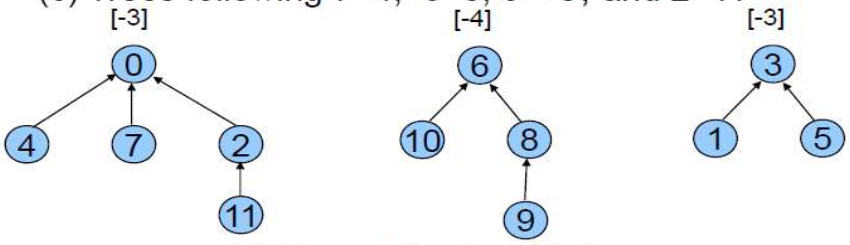


(b) Height-2 trees following 0≡4, 3≡1, 6≡10, and 8≡9

Example 5.5



(c) Trees following 7≡4, 6≡8, 3≡ 5, and 2≡11

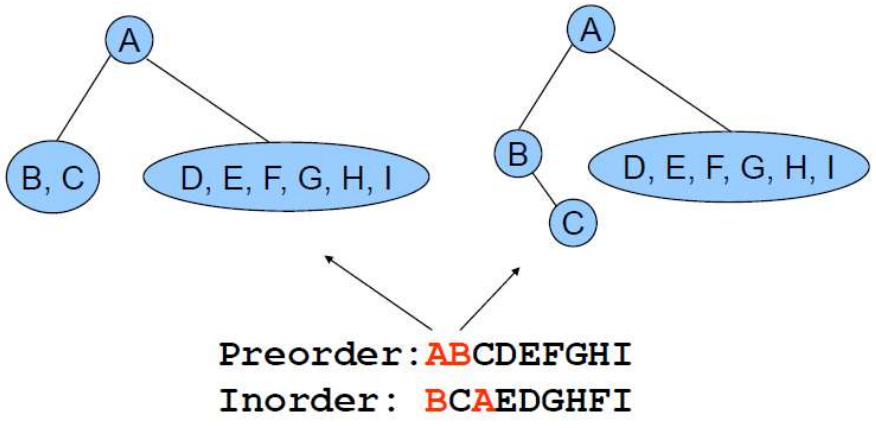


(d) Trees following 11≡0

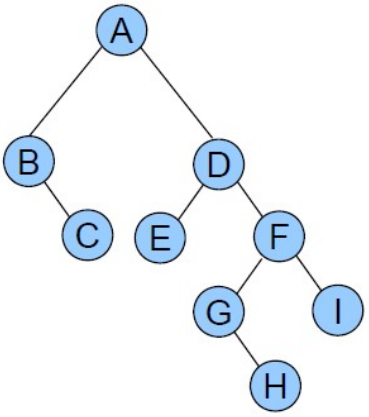
Uniqueness of a Binary Tree

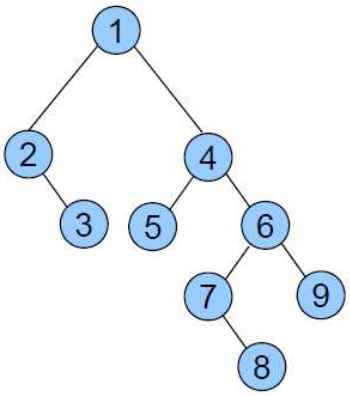
- Suppose that we have the preorder sequence ABCDEFGHI and the inorder sequence BCAEDGHFI of the same binary tree.
- Does such a pair of sequence uniquely define a binary tree?
  - Yes.
  - How to prove it?

Constructing a Binary Tree From Its Preoder and Inorder Sequences



Constructing a Binary Tree From Its Preoder and Inorder Sequences



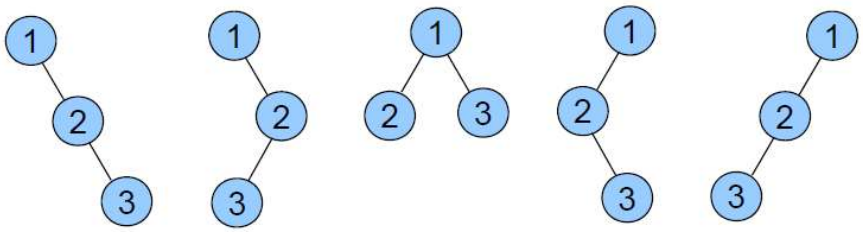


Preorder: 1, 2, 3, 4, 5, 6, 7, 8, 9

Inorder: 2, 3, 1, 5, 4, 7, 8, 6, 9

Distinct Binary Trees

preorder: (1, 2, 3)



(1, 2, 3) (1, 3, 2) (2, 1, 3) (2, 3, 1) (3, 2, 1)

inorder

Dynamic (Self-Adjusting) Tree Structures

- BST is simple and efficient on insert/delete/search but can be unbalanced with degraded performance.
- Dynamic (self-adjusting) tree structures can automatically adjust their structures to improve future operations.
  - AVL tree: a binary tree in which every node is balanced and has height O(log n).
  - Red-Black tree: balanced binary tree with height $h \leq 2 \times \log(n+1)$.
  - Splay tree: no guarantee of balance but good amortized performance (any sequence of m operations take at most $O(m \log n)$ time on an n-node splay tree.)