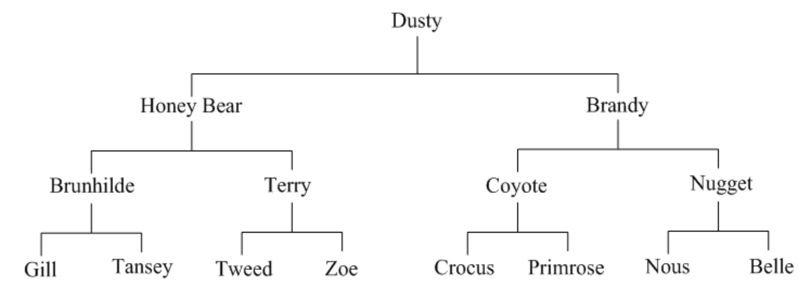
Chapter 5: Trees

Hsien-Chou Liao
Depart. of Comp. Sci. and Info. Eng.
Chaoyang University of Technology

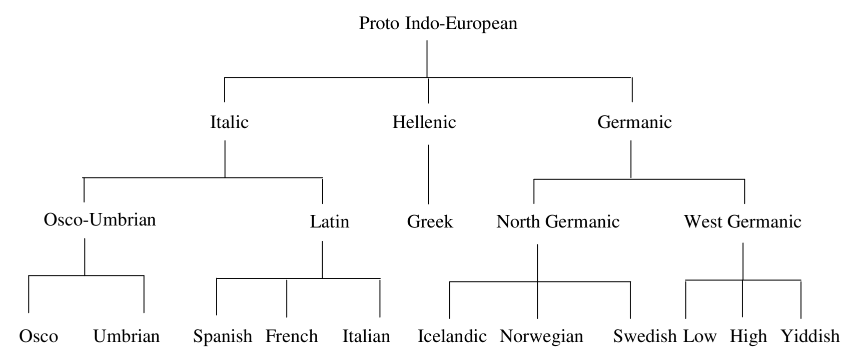
Trees

- Definition: A tree is a finite set of one or more nodes such that:
  - There is a specially designated node called the root.
  - The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots, T_n$, where each of these sets is a tree. We call $T_1, \ldots, T_n$ the subtrees of the root.

Pedigree Genealogical Chart



Lineal Genealogical Chart



Tree Terminology

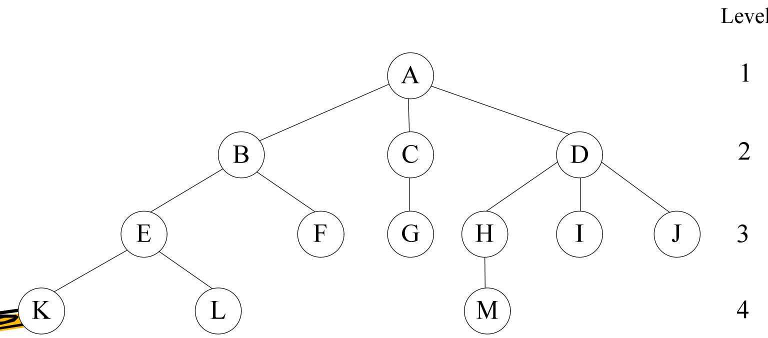
- Normally we draw a tree with the root at the top.
- The degree of a node is the number of subtrees of the node.
- The degree of a tree is the maximum degree of the nodes in the tree.
- A node with degree zero is a leaf or terminal node.
- A node that has subtrees is the parent of the roots of the subtrees, and the roots of the subtrees are the children of the node.
- Children of the same parents are called siblings.

Tree Terminology (Cont.)

- 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 the nodes that are in its subtrees.
- 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.

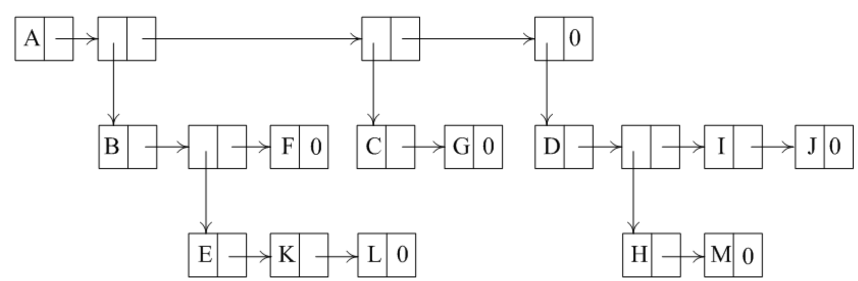
A Sample Tree

- The degree of A is?
- The degree of C is?
- The leaf nodes are?



List Representation of Trees

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



tag field not shown

Possible Node Structure for a Tree of Degree *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 in Figure 5.4, then $n(k-1) + 1$ of the *nk* child fields are 0, $n \geq 1$.

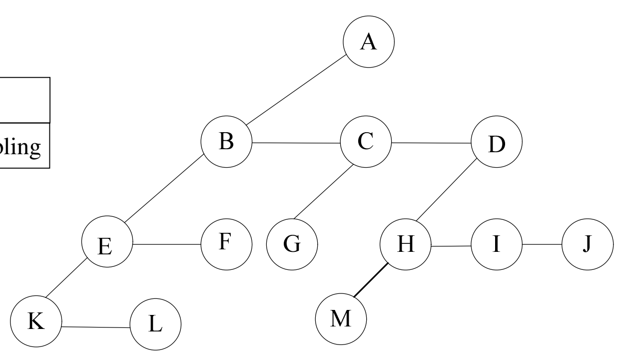
Data	Child 1	Child 2	Child 3	Child 4	…	Child *k*

Q: What is the problem of such structure?

Representation of Trees

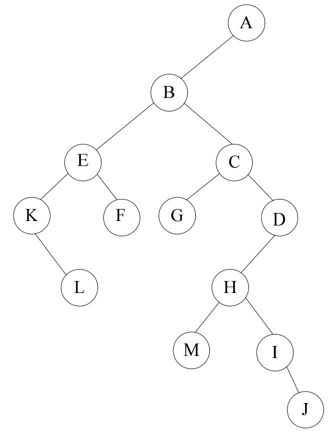
- Left Child-Right Sibling Representation
  - Each node has two links (or pointers).
  - Each node only has one leftmost child and one closest sibling.

data	
left child	right sibling

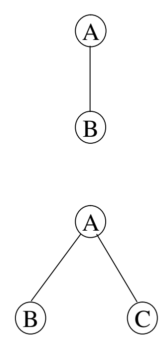


Representation of Degree-Two Tree

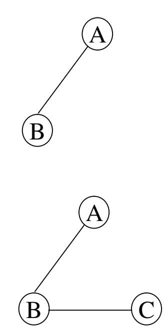
- Left child-right child tree representation.
- It is also known as *binary tree*.



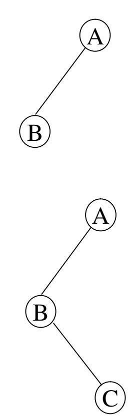
Tree Representations



tree



Left child-right sibling tree

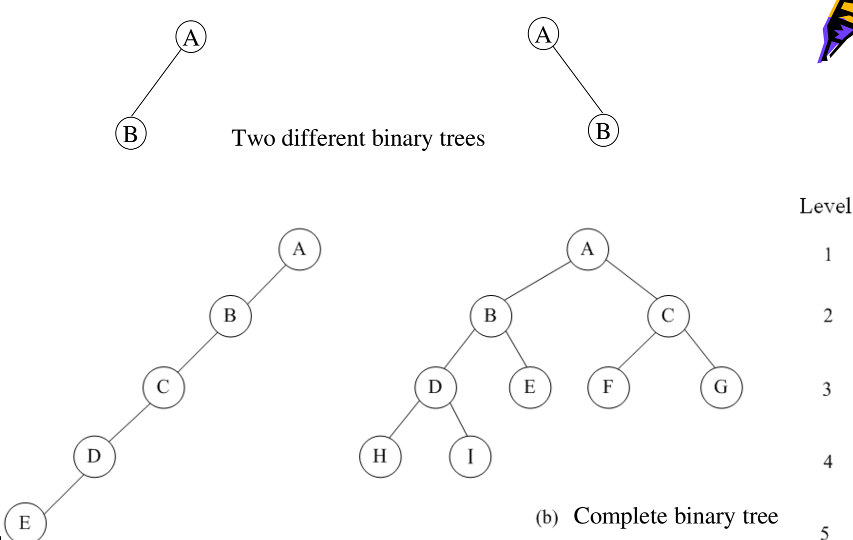


Binary tree

5.2 Binary Tree

- 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.
- The distinctions between a binary tree and a tree:
  - There is no tree with zero nodes. But there is an empty binary tree.
  - Binary tree distinguishes between the order of the children while in a tree we do not.

Figure 5.10: Binary Tree Examples



(a) Skewed binary tree

(b) Complete binary tree

The Properties of Binary Trees

- **Lemma 5.2** [Maximum number of nodes]
  1) The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.
  2) 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 $n_0 = n_2 + 1$.

  $n=n_0+n_1+n_2$ $\quad$ $n=B+1$ $\quad$ $B=n_1+n_2*2$

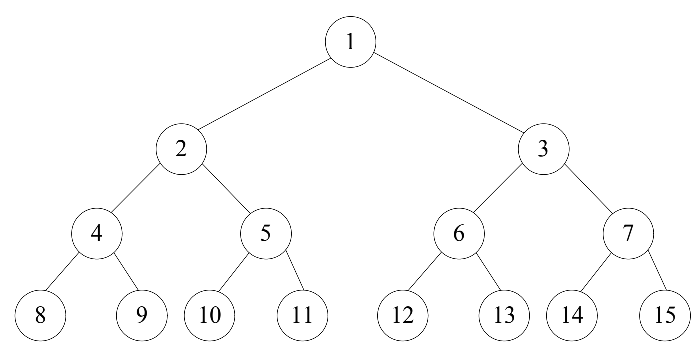
  $n_1+n_2*2+1 = n_0+n_1+n_2$

  (B: the number of branches)

  ($n1$: the number of node of degree one)
- **Definition**: A **full binary tree** of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$.

Binary Tree Definition

- **Definition**: A binary tree 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.



Full binary tree of depth 4

Array Representation of A Binary Tree

- 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 $i \ne 1$. 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.

Proof of Lemma 5.4 (2)

Assume that for all j, $1 \le j \le i$, left_child(j) is at $2j$.

Then two nodes immediately preceding left_child($i + 1$) are the right and left children of i. The left child is at $2i$. Hence, the left child of $i + 1$ is at $2i + 2 = 2(i + 1)$ unless $2(i + 1) > n$, in which case $i + 1$ has no left child.

Array Representation of Binary Trees

	tree
[0]	—
[1]	A
[2]	B
[3]	—
[4]	C
[5]	—
[6]	—
[7]	—
[8]	D
[9]	—
.	.
.	.
.	.
[16]	E

(a) Tree of Figure 5.10(a)

tree
—
A
B
C
D
E
F
G
H
I

(b) Tree of Figure 5.10(b)

Linked Representation

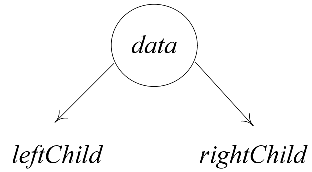
```
template <class T> class Tree; //forward declaration

template <class T>
class TreeNode {
friend class Tree<T>;
private:
    T data;
    TreeNode<T> *leftChild;
    TreeNode<T> *rightChild;
};

template <class T>
class Tree {
public:
  // Tree operations
  .
private:
    TreeNode<T> *root;
};
```

Node Representation

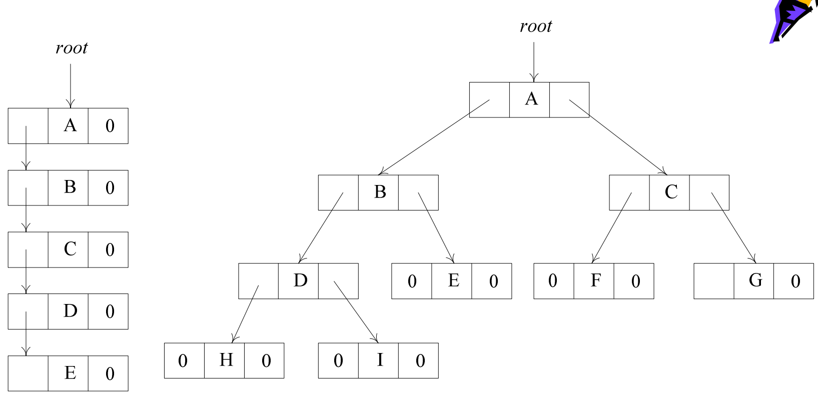
leftChild	*data*	*rightChild*



Q: How to determine the parent of a node?

a field, *parent*, may be included in the class *TreeNode*

Linked Representation for the Binary Trees

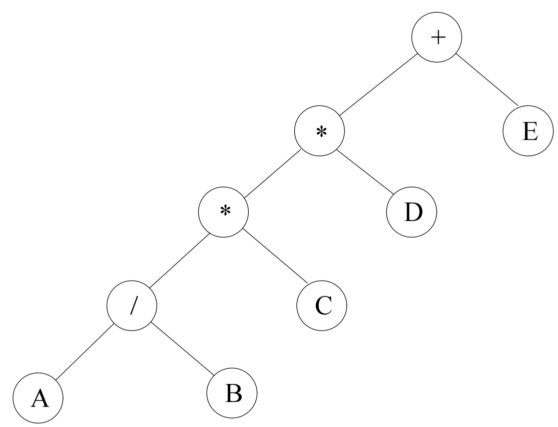


(a) (b)

5.3 Binary Tree Traversal and Tree Iterators

- When visiting each node of a tree exactly once, this produces a linear order for the node of a tree.
- There are 3 traversals if we adopt the convention that we traverse left before right: LVR (inorder), LRV (postorder), and VLR (preorder).
  - L: moving left
  - V: visiting the node
  - R: moving right
- When implementing the traversal, a recursion is perfect for the task.

Binary Tree With Arithmetic Expression



A/B*C*D+E +**/ABCDE AB/C*D*E+

Tree Traversal

- Inorder Traversal: A/B*C*D+E
  => Infix form (program 5.1)
- Preorder Traversal: +**/ABCDE
  => Prefix form (program 5.2)
- Postorder Traversal: AB/C*D*E+
  => Postfix form (program 5.3)

Nonrecursive Inorder Traversal

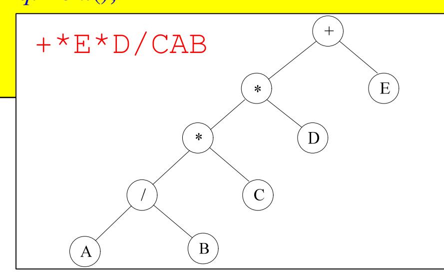
```
template < class T>
void Tree <T>::NonrecInorder()
{// Nonrecursive inorder traversal using a stack
    Stack < TreeNode < T > * > s;  // declare and initialize stack
    TreeNode < T > *currentNode = root;
    while(1) {
        while (currentNode) { // move down leftChild fields
            s.Push(currentNode); //add to stack
            currentNode = currentNode→leftChild;
        }
        if (s.IsEmpty()) return;
        currentNode = s.Top();
        s.Pop(); // delete from stack
        Visit(currentNode);
        currentNode = currentNode→rightChild;
    }
}
```


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 node at a level are visited before moving down to another level.

Level-Order Traversal of A Binary Tree

```
template <class T>
void Tree <T>::LevelOrder()
{// Traverse the binary tree in level order.
    Queue < TreeNode <T>*> q;
    TreeNode<T> *currentNode = root;
    while (currentNode) {
        Visit(currentNode);
        if (currentNode→leftChild) q.Push(currentNode→leftChild);
        if (currentNode→rightChild) q.Push(currentNode→rightChild);
        if (q.IsEmpty()) return;
        currentNode = q.Front();
        q.Pop();
    }
}
```



Traversal Without A Stack

Two methods:

1. Use of *parent* field to each node.
2. Use of two bits per node to represents binary trees as **threaded binary trees**.
   – It will be studied in Section 5.5.

Additional Binary Tree Operations

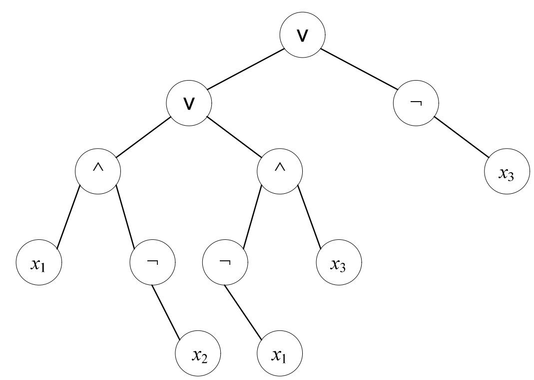
- Using the traversal of a binary tree, we can easily write other routines for binary tree. E.g.,
  – Copying Binary Trees (program 5.9)
  – Testing Equality
    - Two binary trees are equal if their topologies are the same and the information in corresponding nodes is identical.

The Satisfiability Problem

- Consider a set of expressions defined by the following rules:
  – A variable is an expression
  – If x and y are expressions then $x \wedge y, x \vee y, and \neg x$ are expressions
  – Parentheses can be used to alter the normal order of evaluation, which is **not** before **and** before **or**.
- A satisfiability problem: if there is an assignment of values to the variables that causes the value of the expression to be true.

Propositional Formula in a Binary Tree

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



For n variables, there are 2^n combinations of *true* and *false*.

Therefore, the algorithm take $O(g2^n)$, or exponential time.

g: the time to substitute values for variables and evaluate the expression.

Perform Formula Evaluation

- To evaluate an expression, we can traverse its tree in postorder. Why?
- To perform evaluation, each node has four fields:
  - *leftChild*
  - *rightChild*
  - *data.first*: enum Operator {Not, And, Or, True, False}
    - **Non-leaf node**: is set to one of the operators {Not, And, Or}
    - **Leaf node**: is set either *True* or *False* depending on the current truth assignment.
  - *data.second*: to store the evaluation result of subtree

First Version of Satisfiability Algorithm

for each of the 2^n possible truth value combinations for the *n* variables
{
replace the variables by their values in the current truth value combination;
evaluate the fomula by traversing the tree it points to in postorder;
if (*fomula.Data*().*second*()) **{ cout** << current combination ; **return** ; **}**
}
cout << "no satisfiable combination";

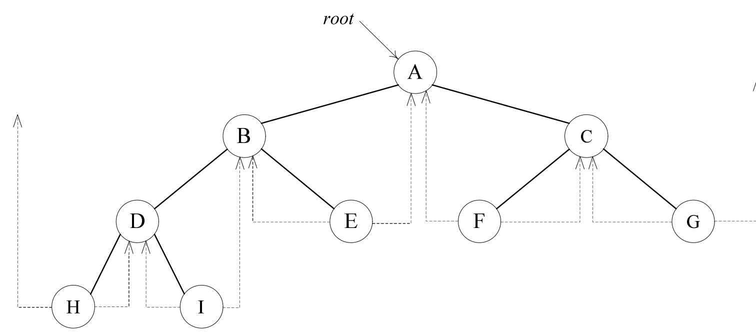
```
// visit the node pointed at by p
switch (p→data.first) {
    case Not: p→data.second = !p→rightChild→data.second; break;
    case And: p→data.second =
                p→leftChild→data.second && p→rightChild→data.second;
                break;
    case Or: p→data.second =
                p→leftChild→data.second || p→rightchild→data.second;
                break;
    case True: p→data.second = true; break;
    case False: p→data.second = false;
}
```

(Visiting a node in an expression tree)

5.5 Threaded Binary Tree

- For the linked representation, there are more 0-links than actual pointers.
  - There are n+1 0-links and 2n total links.
- Thread: a pointer to other nodes in the tree for replacing the 0-link.
- Threads are constructed using the following rules:
  - A 0 *rightChild* field at node *p* is replaced by a pointer to the node that would be visited after *p* when traversing the tree in **inorder**. That is, it is replaced by the inorder successor of *p*.
  - A 0 *leftChild* link at node *p* is replaced by a pointer to the node that immediately precedes node *p* in inorder (i.e., it is replaced by the inorder predecessor of *p*).

Threaded Tree Corresponding to Figure 5.10(b)



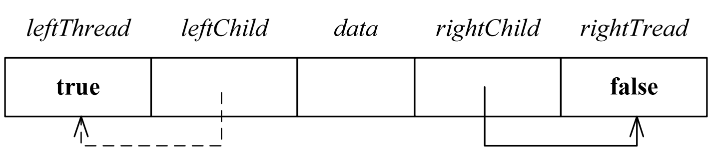
Inorder sequence: H, D, I, B, E, A, F, C, G

Threads

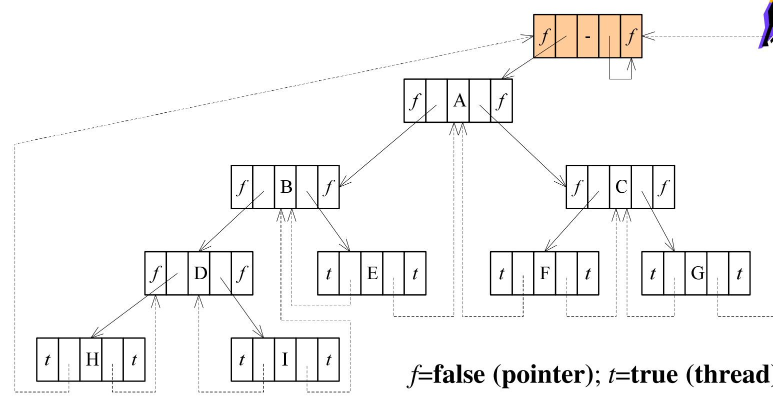
- To distinguish between normal pointers and threads, two boolean fields, *leftThread* and *rightThread*, are added to the record in memory representation.
  - `t->leftThread = TRUE`
    - => *t->leftChild* is a **thread**
  - `t->leftThread = FALSE`
    - => *t->leftChild* is a **pointer** to the left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

Threads (Cont.)

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



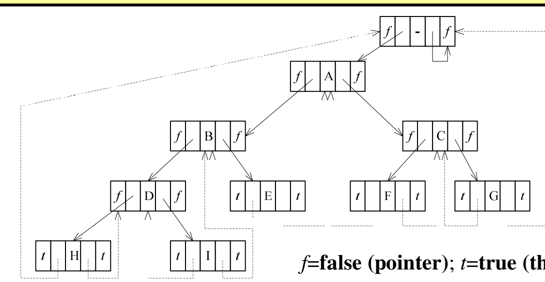
Memory Representation of Threaded Tree



f=**false (pointer)**; *t*=**true (thread)**

Inorder Traversal of a Threaded Binary Tree

```
T* ThreadedInorderIterator::Next ()
{// Return the inorder successor of currentNode in a threaded binary tree
    ThreadedNode <T> *temp = currentNode → rightChild;
    if (!currentNode → rightThread)
        while (!temp → leftThread) temp = temp → leftChild;
    currentNode = temp;
    if (currentNode = = root) return 0;
    else return &currentNode → data;
}
```

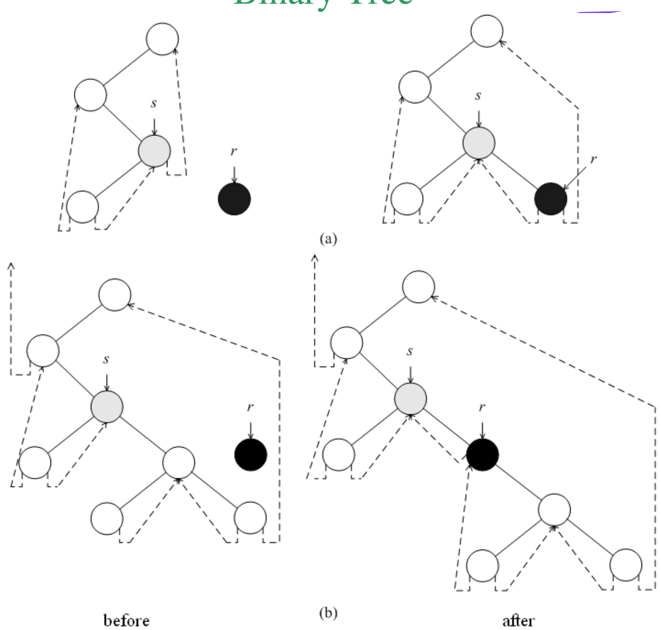


f=**false (pointer)**; *t*=**true (thread)**

Inserting a Node to a Threaded Binary Tree

- 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, the 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 an 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.

Insertion of *r* as a Right Child of *s* in a Threaded Binary Tree



Program 5.16: Inserting *r* as the Right Child of *s*

```
template <class T>
void ThreadedTree <T>::InsertRight (ThreadedNode <T> *s,
                                    ThreadedNode <T> *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;
    s → rightThread = false;
    if (! r → rightThread) {
            ThreadedNode <T> *temp = InorderSucc (r);
                                        // return the inorder successor of r
            temp → leftChild = r;
    }
}
```


5.6 Heap

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.

Examples of Priority Queues

- Suppose a machine that serves multiple users.
  - Each user pays a fixed amount per use. However, the time needed by each user is different.
  - In order to maximize the returns from this machine, the user with the smallest time requirement is selected.
  - Hence, a min priority queue is required.
- If each user needs the same amount of time but willing to pay different amounts for the service.
  - This requires a max priority queue.

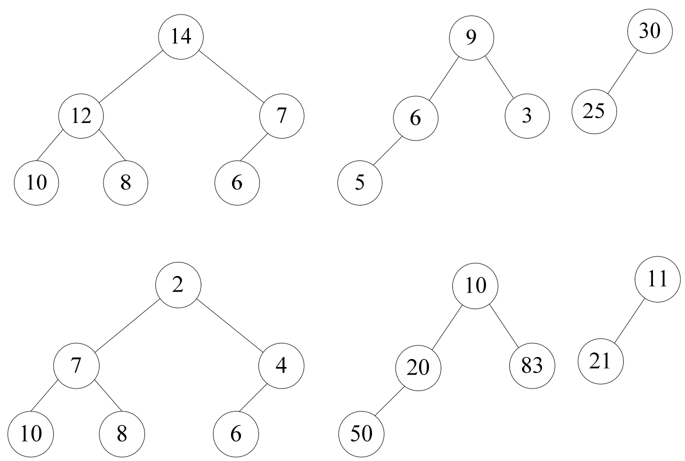
Priority Queue Representation

- Unorder Linear List: the simplest way to represent a priority queue.
  - *n*: the number of elements in the priority queue.
  - *push*: O(1)
  - *pop*: O(*n*)
    - Find the element with max priority and then delete it.

Max (Min) Heap

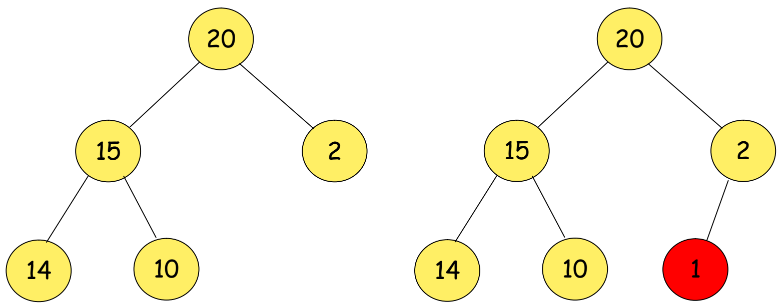
- Heaps are frequently used to implement priority queues. The complexity is O(log n).
- Definition:
  - A max (min) tree is a tree in which the key value in each node is no smaller (larger) than the key values in its children (if any).
  - A max heap is a complete binary tree that is also a max tree.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

Max (Min) Heap Examples

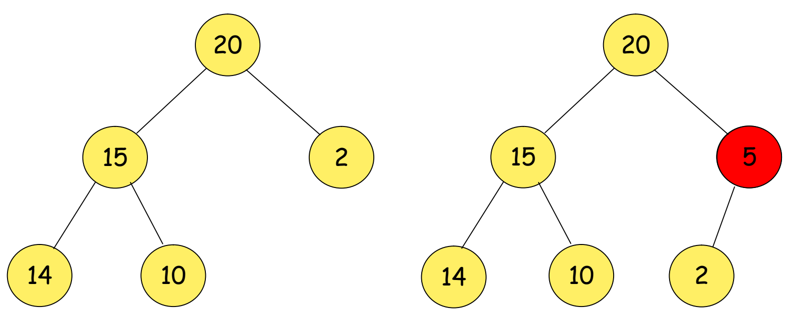


Insertion Into A Max Heap (1)

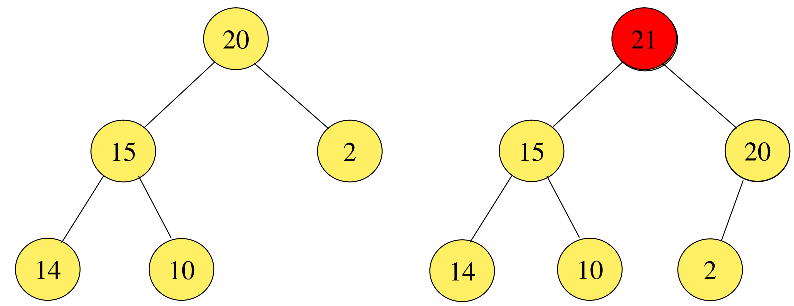
- A bubbling up process is used:
  - Begins at the new node of the tree and moves toward the root.



Insertion Into A Max Heap (2)



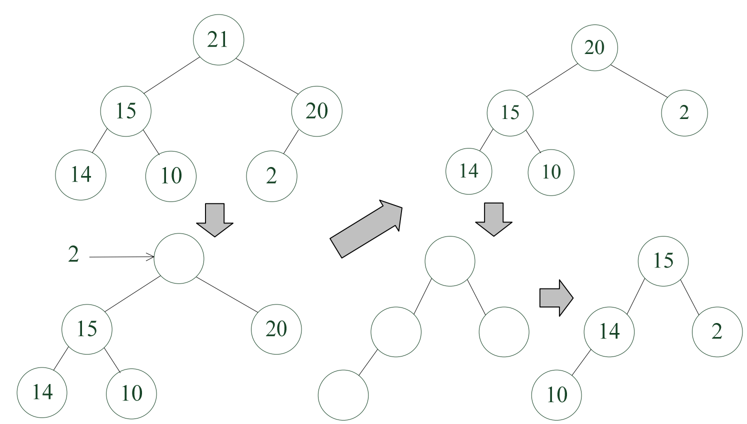
Insertion Into A Max Heap (3)



Program 5.16

Deletion From a Max Heap

- A trickle down strategy is used.



(a) (b) (c)

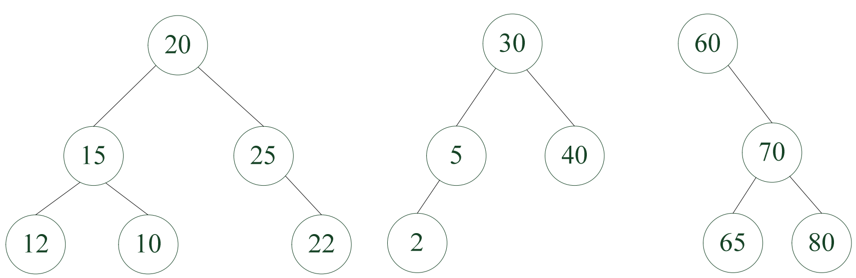
Delete the element 21 Delete the element 20

5.7 Binary Search Tree

- Binary search tree provide a better performance for search, insertion, and deletion.
- Definition: A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:
  - Every element has a key and no two elements have the same key (i.e., the keys are distinct)
  - The keys (if any) in the left subtree are smaller than the key in the root.
  - The keys (if any) in the right subtree are larger than the key in the root.
  - The left and right subtrees are also binary search trees.

Binary Trees

- Which one is not a binary search tree?



(a) (b) (c)

Searching a Binary Search Tree

- If the root is 0, then this is an empty tree. No search is needed.
- If the root is not 0, compare the *k* with the key of root.
  - If *k* is less than the key of the root, then no elements in the right subtree can have key value *k*. We only need to search the left tree.
  - If *k* larger than the key of the root, only the right subtree is to be searched.
  - If *k* equals to the key of the root, then the search terminates successfully.

Search Binary Search Tree by Rank

- Rank: the position of a node in inorder
  - The first node visited in inorder has rank 1.
- If we wish to search by rank, each node should have an additional field *leftSize*.
- *leftSize*= 1 + the number of elements in the left subtree of a node.
- 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.

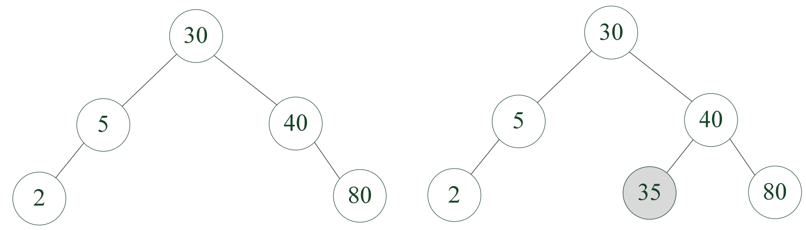
Searching a Binary Search Tree by Rank

```
template <class K, class E> // search by rank
pair<K, E>* BST<K, E> :: RankGet(int r)
{ // Search the binary search tree for the rth smallest pair
  TreeNode < pair<K, E> > *currentNode = root;
  while (currentNode) {
     if (r < currentNode →leftSize) currentNode = currentNode →leftChild;
     else if (r > currentNode →leftSize)
     {
        r −= currentNode →leftSize;
        currentNode = currentNode →rightChild;
     }
     else return & currentNode →data;
  }
  return 0;
}
```


Insertion into a Binary Search Tree

- Before insertion is performed, a search must be done to make sure that the value to be inserted is not already in the tree.
- If the search fails, then we know the value is not in the tree. So it can be inserted into the tree.
- It takes O(*h*) to insert a node to a binary search tree.

Inserting Into A Binary Search Tree



(a) Insert a element with key 80

(b) Insert a element with key 35

Insertion into a Binary Search Tree

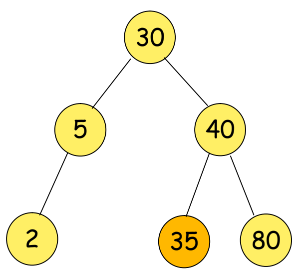
```
template <class K, class E>
void BST<K, E > :: Insert(const pair<K, E >& thePair)
{ // Insert thePair into the binary search tree
  // search thePair.first , pp is the parent of p
  TreeNode < pair<K, E> > *p = root, *pp = 0;
  while (p) {
     pp = p;
     if (thePair.first < p →data.first) p = p →leftChild;
     else if (thePair.first > p →data.first) p = p →rightChild;
     else // duplicate, update associated element
       { p →data.second = thepair.second; return;}
  }
  // perform insertion
  p = new TreeNode< pair<K, E> > (thePair);
  if (root) // tree not empty
     if (thePair.first < pp →data.first) pp→leftChild = p;
     else pp→rightChild = p
  else root = p;
}
```

Deletion from a Binary Search Tree

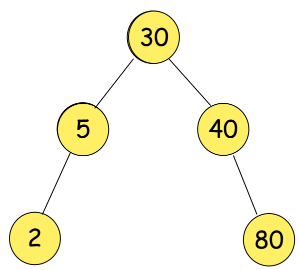
- Delete a leaf node
  - A leaf node which is a right child of its parent
  - A leaf node which is a left child of its parent
- Delete a non-leaf node
  - A node that has one child
  - A node that has two children
    - Replaced by the largest element in its left subtree, or
    - Replaced by the smallest element in its right subtree
- Again, the delete function has complexity of $O(h)$

Deleting from a Binary Search Tree

Delete a leaf node

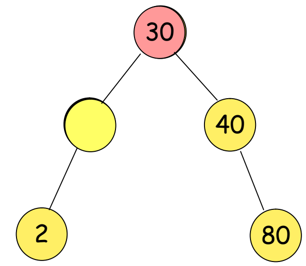


Delete a node with one child



Deleting from a Binary Search Tree

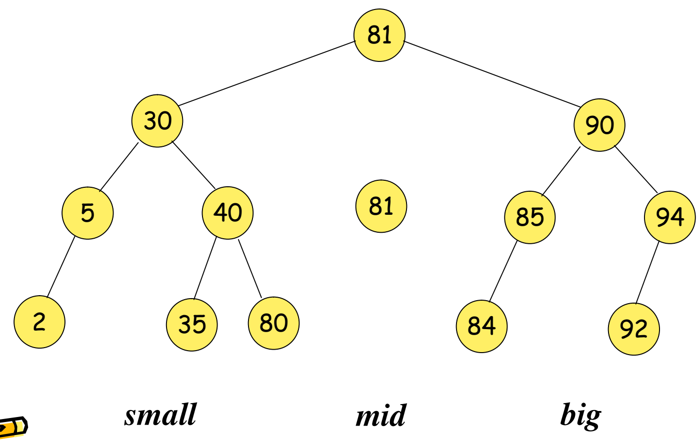
Delete a node with two children



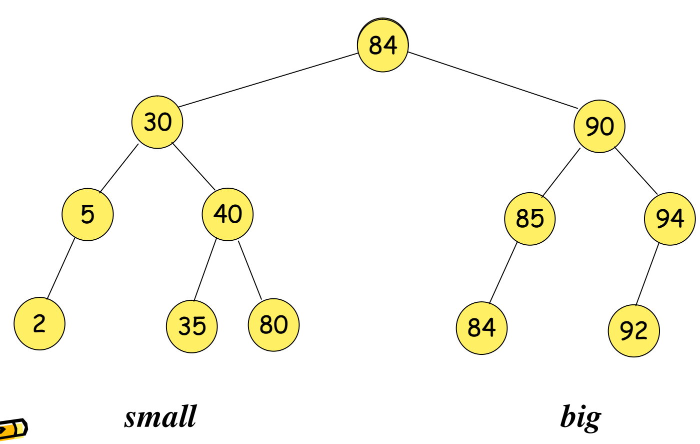
Joining and Splitting Binary Trees

- *ThreeWayJoin(small, mid, big)*: Creates a binary search tree consisting the BST *small*, *big*, and the pair *mid*.
- *TwoWayJoin(small, big)*: Joins two BST *small* and *big* to obtain a single BST.
- *Split(k, small, mid, big)*: BST is split into three parts:
  - *small*: a BST that contains all pairs that have key less than *k*
  - *mid*: the pair contains the key *k*
  - *big*: a BST that contains all pairs that have key larger than *k*.

ThreeWayJoin(small, mid, big)

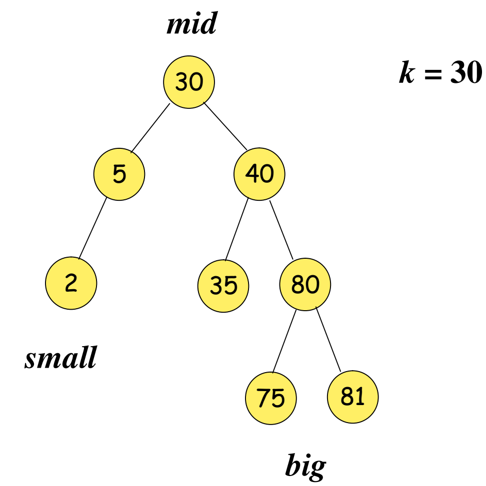


TwoWayJoin(small, big)



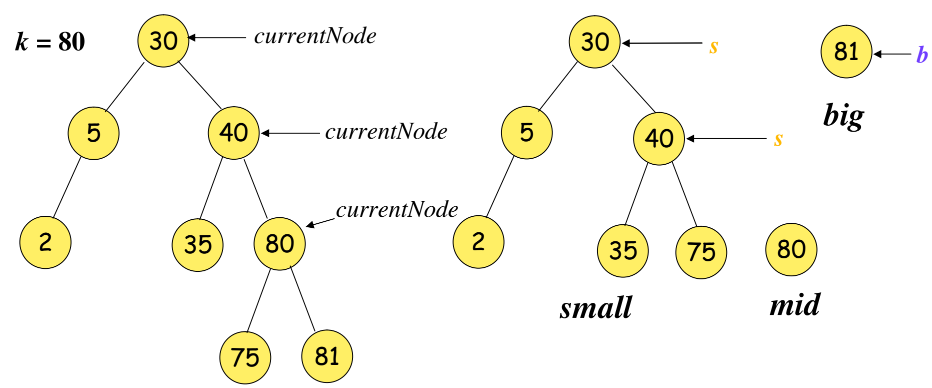
Split(k, small, mid, big)

Splitting at root



Split(k, small, mid, big)

Splitting at arbitrary node



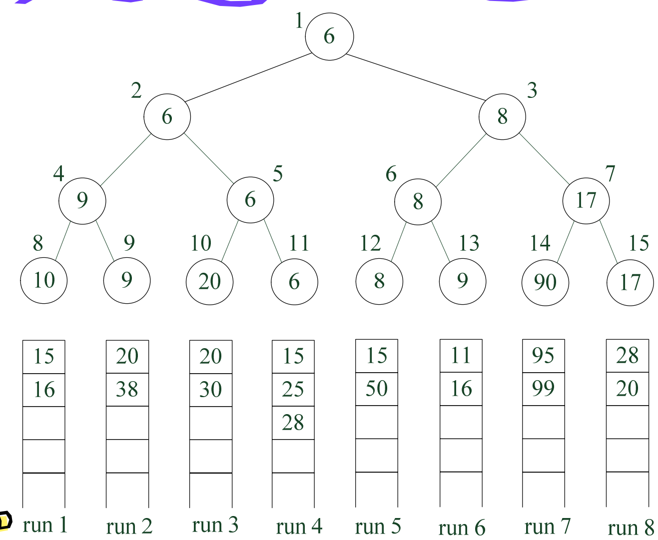
5.8 Selection Trees

- How to merge *k* ordered sequences, called *runs* (assume in non-decreasing order) into a single sequence?
  - the most intuitive way is probably to perform *k* – 1 comparison each time to select the smallest one among the first number of each of the *k* ordered sequences. This goes on until all numbers in every sequences are visited.
- Is there a better way?
  - Selection tree is the answer.

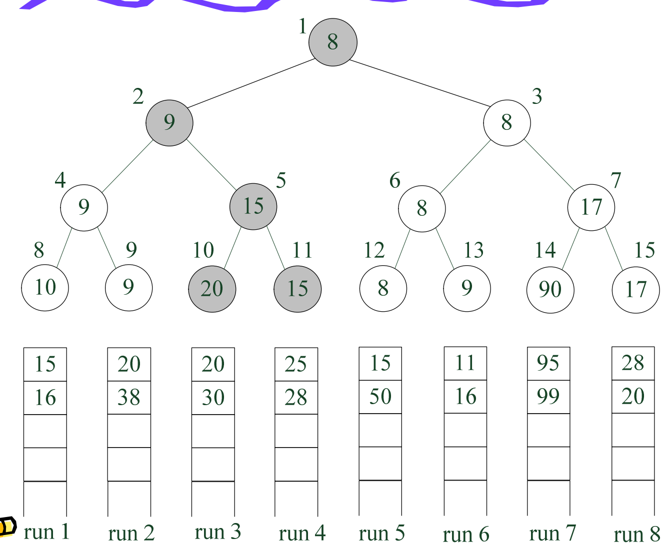
Winner Tree

- There are two kinds of selection trees:
  - Winner trees and loser trees
- A winner tree is a complete binary tree in which each node represents the smaller of its two children. Thus the root represents the smallest node in the tree.
- Each leaf node represents the first record in the corresponding run.
- Each non-leaf node in the tree represents the winner of its right and left subtrees.

Winner Tree for k = 8



The Reconstruction of Winner Tree



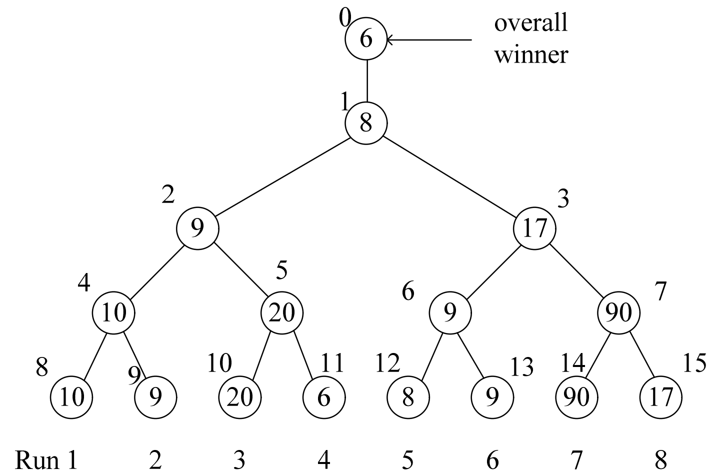
Analysis of Winner Tree

- The number of levels in the tree is $\lceil \log_2(k+1) \rceil$
  - The time to restructure the winner tree is $O(\log_2 k)$.
- Since the tree has to be restructured each time a number is output, the time to merge all n records is $O(n \log_2 k)$.
- The time required to setup the selection tree for the first time is $O(k)$.
- Total time needed to merge the k runs is $O(n \log_2 k)$.

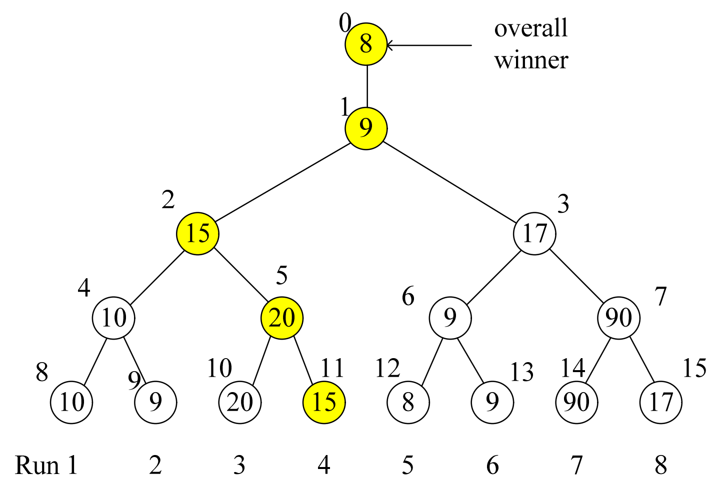
Loser Tree

- A selection tree in which each nonleaf node retains a pointer to the loser is called a loser tree.
- Again, each leaf node represents the first record of each run.
- An additional node, node 0, has been added to represent the overall winner of the tournament.

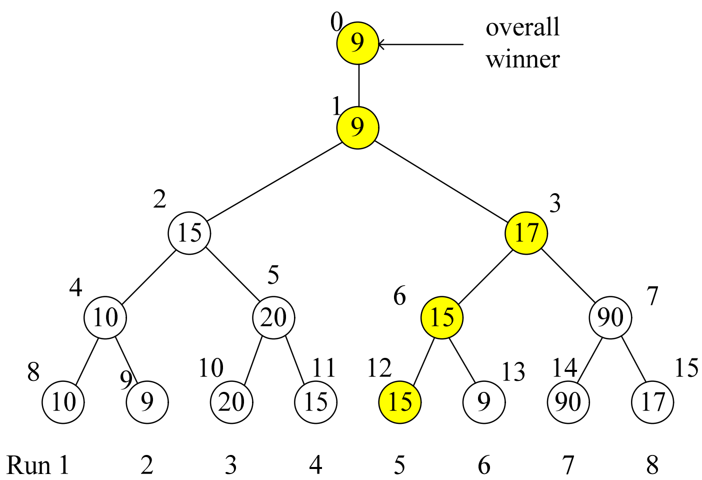
Loser Tree



Loser Tree

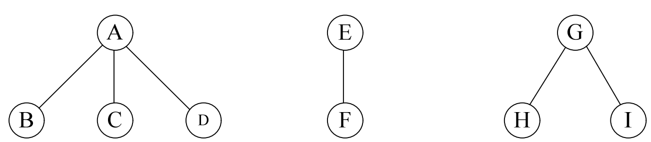


Loser Tree



5.9 Forests

- Definition: A forest is a set of $n \geq 0$ disjoint trees.
- When we remove a root from a tree, we'll get a forest. E.g., Removing the root of a binary tree will get a forest of two trees.

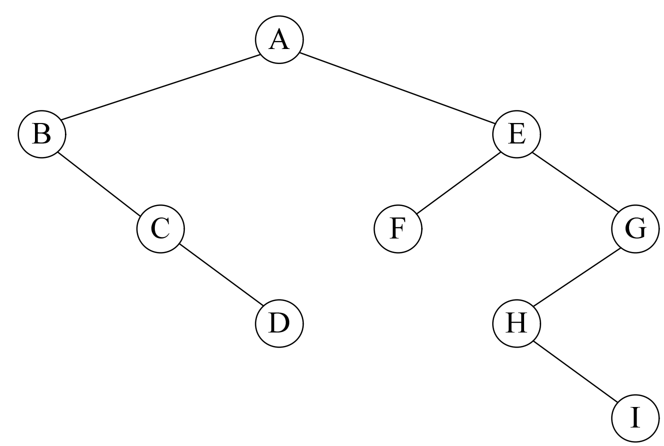


Three-tree forest

Transforming a Forest into a Binary Tree

- Definition: If $T_1, \ldots, T_n$ is a forest of trees, then the binary tree corresponding to this forest, denoted by $B(T_1, \ldots, T_n)$,
  - is empty if $n = 0$
  - has root equal to root (T_1); has left subtree equal to $B(T_{11}, T_{12}, \ldots, T_{1m})$, where $T_{11}, T_{12}, \ldots, T_{1m}$ are the subtrees of root (T_1); and has right subtree $B(T_2, \ldots, T_n)$.

Binary tree of forest

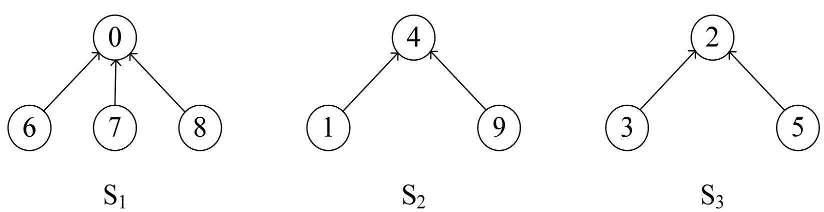


5.10 Representation of Disjoint Sets

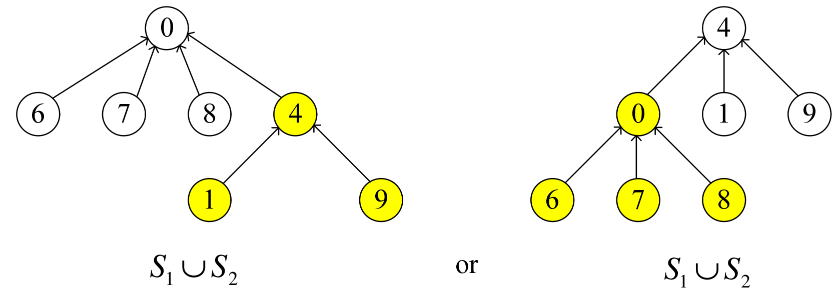
- Trees can be used to represent sets.
- Disjoint set union: If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j$ = {all elements x such that x is in S_i or S_j}.
- *Find*(i): Find the set containing element i.

Possible Tree Representation of Sets

- For example:
  - 3 in set S_3
  - 8 in set S_1



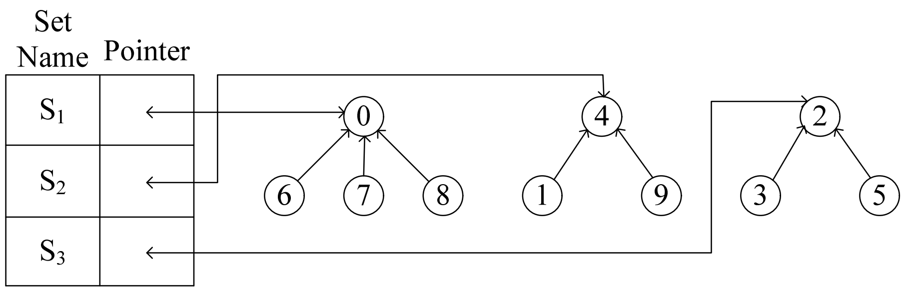
Possible Representations of $S_1 \cup S_2$



Unions and Find Operations

- Union: To obtain the union of two sets, just set the parent field of one of the roots to the other root.
- Find: To figure out which set an element is belonged to, just follow its parent link to the root and then follow the pointer in the root to the set name.

Data Representation for S_1, S_2, S_3



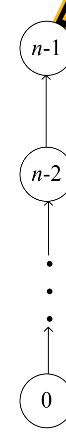
Array Representation of S_1, S_2, S_3

- We ignore the actual set names and just identify sets by the roots of the trees.
- Assume set elements are numbered 0 through $n-1$. The array representation of S_1, S_2, S_3 is shown below:

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

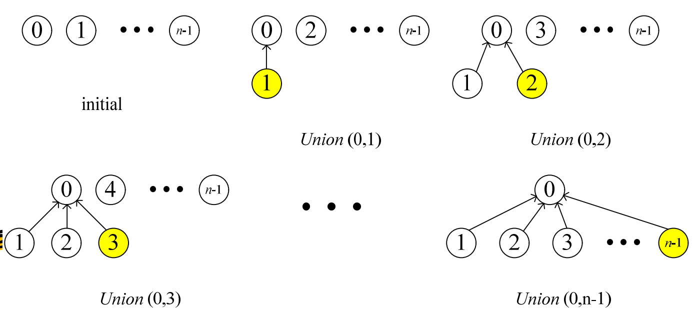
Analysis of *SimpleUnion* and *SimpleFind*

- For a set of *n* elements each in a set of its own, let us process the following operations:
  union(0, 1), union(1, 2), …, union(*n*-2, *n*-1)
  find(0), find (1), …, find(*n*-1)
- The result of the union function is a degenerate tree.
- The *n*-1 unions can be processed in time O(*n*)
- The time required to process a find for an element *i* is O(*i*). The total time of the *n* finds is $O\left(\sum_{i=1}^{n} i\right) = O(n^2)$
- The complexity can be improved by using weighting rule for union.



Weighting Rule

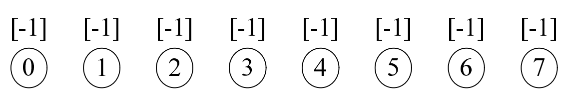
- Definition [Weighting rule for union(*i*, *j*)]:
  - If the number of nodes in the tree with root *i* is less than the number in the tree with root *j*, then make *j* the parent of *i*;
  - Otherwise make *i* the parent of *j*.



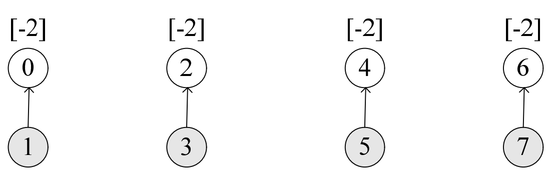
Weighted Union

- 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 $O(u + f \log u)$, as no tree has more than u nodes in it.

Trees Achieving Worst-Case Bound

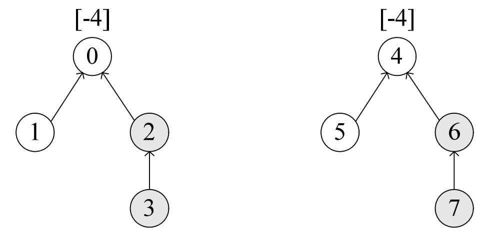


(a) Initial height-1 trees

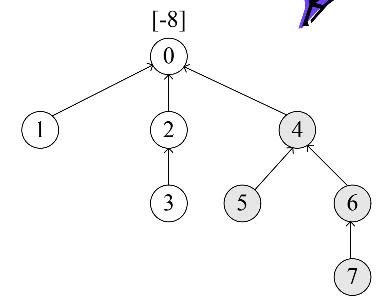


(b) Height-2 trees follows *Union* (0,1), (2,3), (4,5), and (6,7)

Trees Achieving Worst-Case Bound (Cont.)



(c) Height-3 trees following *Union* (0,2) and (4,6)

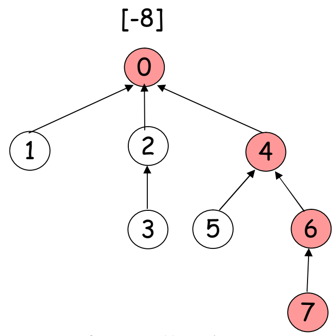


(d) Height-4 tree follows *Union* (0,4)

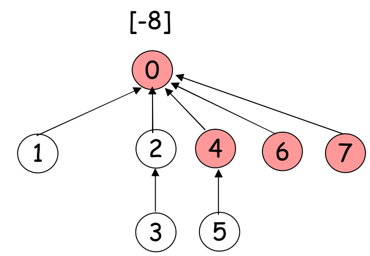
Collapsing Rule

- Definition [Collapsing rule]: 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*).
- The first run of find operation will collapse the tree. Therefore, all following find operation of the same element only goes up one link to find the root.

Collapsing Find



Before collapsing
the path from 7 to 0



After collapsing
the path from 7 to 0

Analysis of *WeightedUnion* and *CollapsingFind*

- The use of collapsing rule roughly double the time for an individual find. However, it reduces the worst-case time over a sequence of finds.
- Lemma 5.6 [Tarjan and Van Leeuwen]: Assume that we start with a forest of trees, each having one node. Let $T(f, u)$ be the maximum time required to process any intermixed sequence of f finds and u unions. Assume that $u \geq n/2$. Then

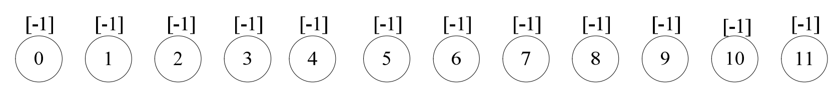
  $k_1(n + f\alpha(f + n, n)) \leq T(f, u) \leq k_2(n + f\alpha(f + n, n))$

  for some positive constants k_1 and k_2.

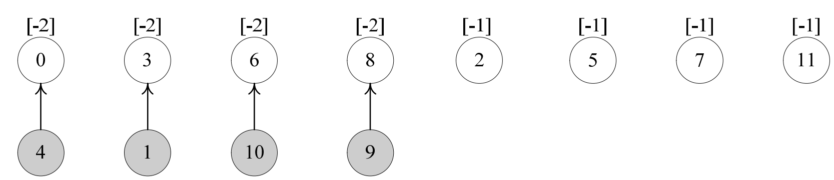
Application to Equivalence Class

- The aforementioned techniques can be applied to the equivalence class problem.
- Assume initially all n polygons are in an equivalence class of their own: $parent[i] = -1, 0 \leq i < n$.
  - Firstly, we must determine the sets that contains i and j.
  - If the two are in different set, then the two sets are to be replaced by their union.
  - If the two are in the same set, then nothing need to be done since they are already in the equivalence class.
  - So we need to perform two finds and at most one union.
- If we have n polygons and m equivalence pairs, we need
  - $O(n)$ to set up the initial n-tree forest.
  - $2m$ finds
  - at most $\min\{n-1, m\}$ unions.
- If *weightedUnion* and *CollapsingFind* are used, the time complexity is $O(n + m\ (2m, \min\{n-1, m\}))$.
  - This seems to slightly worse than section 4.7 ($O(m+n)$). But this scheme demands less space.

Trees for Example 5.5

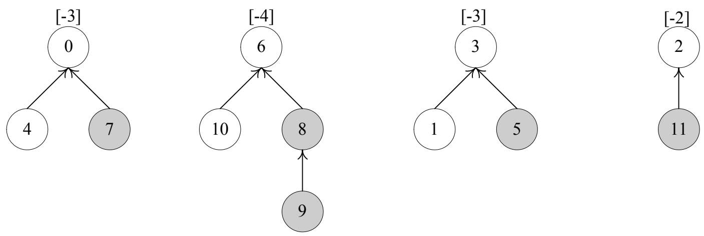


(a) Initial trees

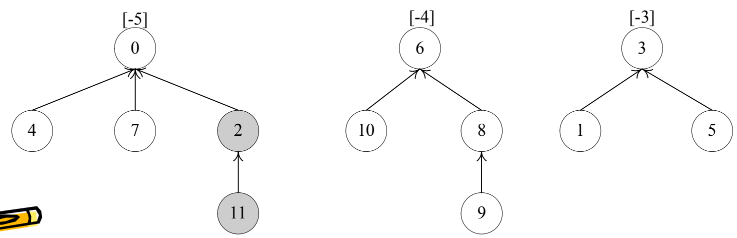


(b) Height-2 trees following $0 \equiv 4$, $3 \equiv 1$, $6 \equiv 10$, and $8 \equiv 9$

Trees for Example 5.5 (Cont.)



(c) Trees following $7 \equiv 4$, $6 \equiv 8$, $3 \equiv 5$, and $2 \equiv 11$



(d) Trees following $11 \equiv 0$

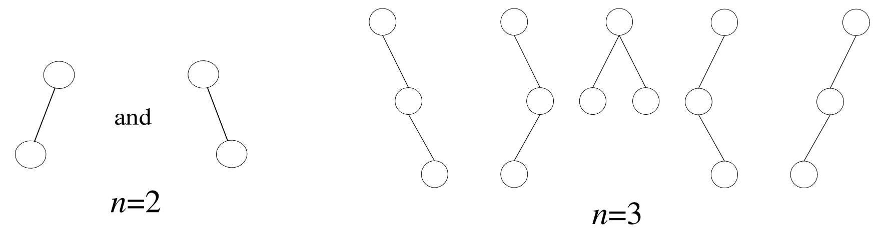
5.11 Counting Binary Tree

- Consider three problems:

1. The number of distinct binary trees having *n* nodes.
2. The number of distinct permutations of the numbers from 1 through *n* obtainable by a stack.
3. The number of distinct ways of multiplying $n + 1$ matrices.

Distinct Binary Trees

- If n=0 or n=1, there is only one binary tree.
- If n=2, then there are two distinct trees.
- If n=3, there are five such trees.
- How many distinct trees are there with n nodes?

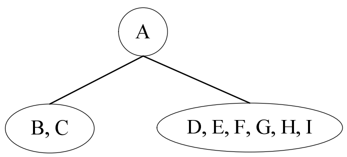


Stack Permutations

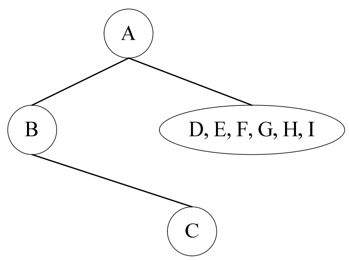
- In section 5.3 we introduced preorder, inorder, and postorder traversal of a binary tree.
- Suppose we have the preorder sequence *A B C D E F G H I* and the inorder sequence *B C A E D G H F I* of the same binary tree. Does such a pair of sequences uniquely define a binary tree?

Constructing A Binary Tree From Its Inorder and Preorder Sequences

- *A* must be the root by preorder traversal (*VLR*).
- According inorder traversal (*LVR*), *B C* are in the left subtree and the remaining nodes are in the right tree.
- *B* is the next root by preorder traversal.
- No node precedes *B* in the inorder sequence, *B* has an empty left subtree, which means *C* is in its right subtree.



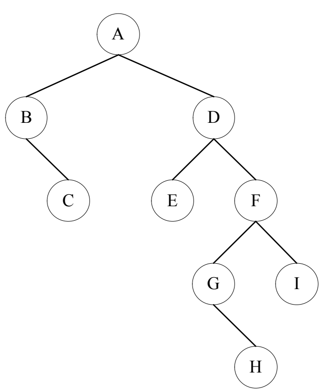
(a)



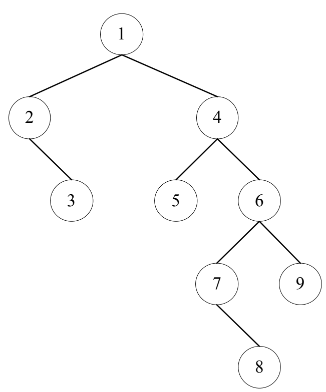
(b)

Constructing A Binary Tree From Its Inorder and Preorder Sequences (Cont.)

- Every binary tree has a unique pair of preorder/ inorder sequences.



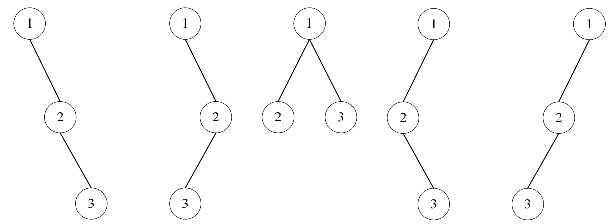
(a)



(b)

Distinct Binary Trees

- If the preorder permutation is 1, 2, …, *n*, the number of distinct binary trees is equal to the number of distinct inorder permutations.
- For example, the preorder permutation 1, 2, 3, the possible inorder permutation obtained by a stack are: (1, 2, 3)(1, 3, 2)(2, 1, 3)(2, 3, 1)(3, 2, 1)



Matrix Multiplication

- Computing the product of *n* matrices are related to the distinct binary tree problem.

$M_1 * M_2 * \ldots * M_n$

$n = 3$ $(M_1 * M_2) * M_3$

$M_1 * (M_2 * M_3)$

$n = 4$ $((M_1 * M_2) * M_3) * M_4$

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

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

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

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

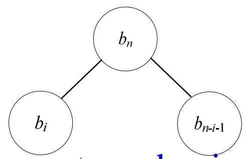
- Let b_n be the number of different ways to compute the product of *n* matrices. $b_1=1$, $b_2 = 1$, $b_3 = 2$.

$$b_n = \sum_{i=1}^{n-1} b_i b_{n-i}, \quad n > 1$$

Distinct Binary Trees (Cont.)

- Let b_n be the number of distinct binary trees with n nodes.
- b_n can be formed in the following way: a tree and two subtrees with b_i and b_{n-i-1}.

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



- Therefore, the number of distinct binary trees having n nodes, the number of distinct permutations of the numbers from 1 through n obtainable by a stack, and the number of distinct ways of multiplying $\boldsymbol{n+1}$ matrices are all equal.

Thanks for your attention!

Q & A