

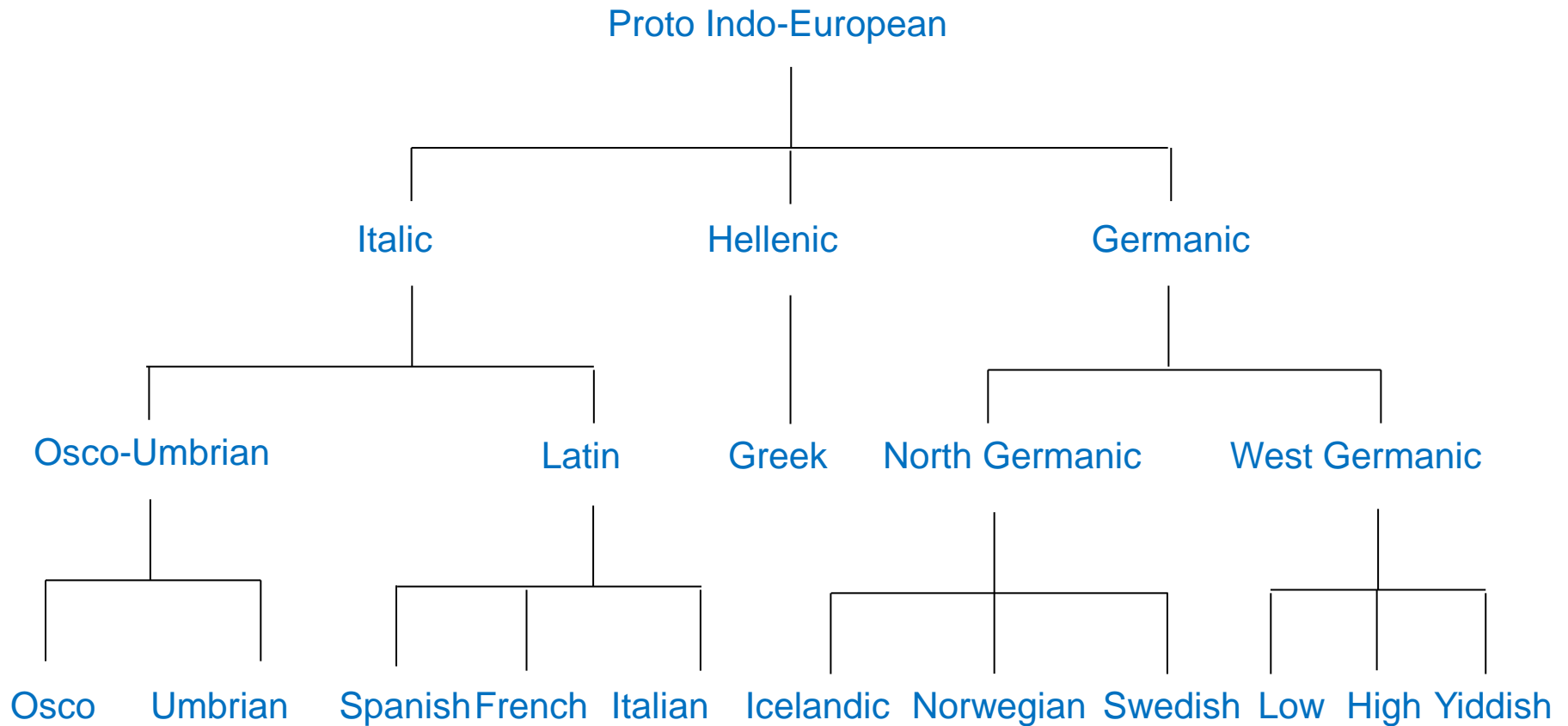
Trees

(chapter 5)

- Tree definition
- Tree representation
- Binary trees
 - ADT and representation
 - Traversal and other operations
 - Binary trees with threads
- Priority queues and heaps
- Binary search trees
- Selection trees
- Forests
- Representing sets with trees

A Tree Example

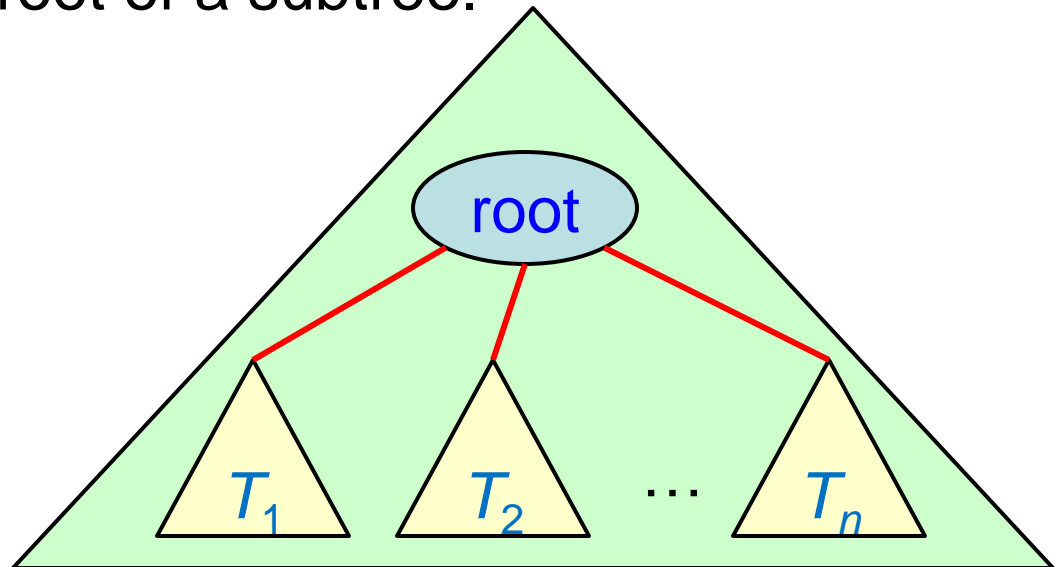
A "tree" is used to represent the hierarchical organization of a set of items.



A Recursive Definition of Trees

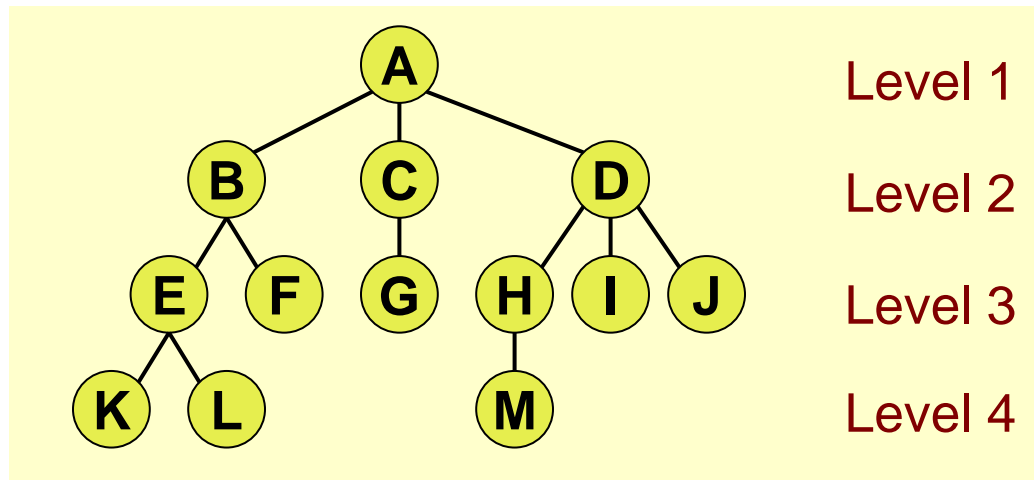
A **tree** T is a finite set of nodes:

- A particular node is the **root** of the tree.
- The other nodes are partitioned into disjoint sets T_1, T_2, \dots, T_n :
 - Each of these disjoint sets is itself a tree, and is called a **subtree** of the root of T .
 - Each node is the root of a subtree.



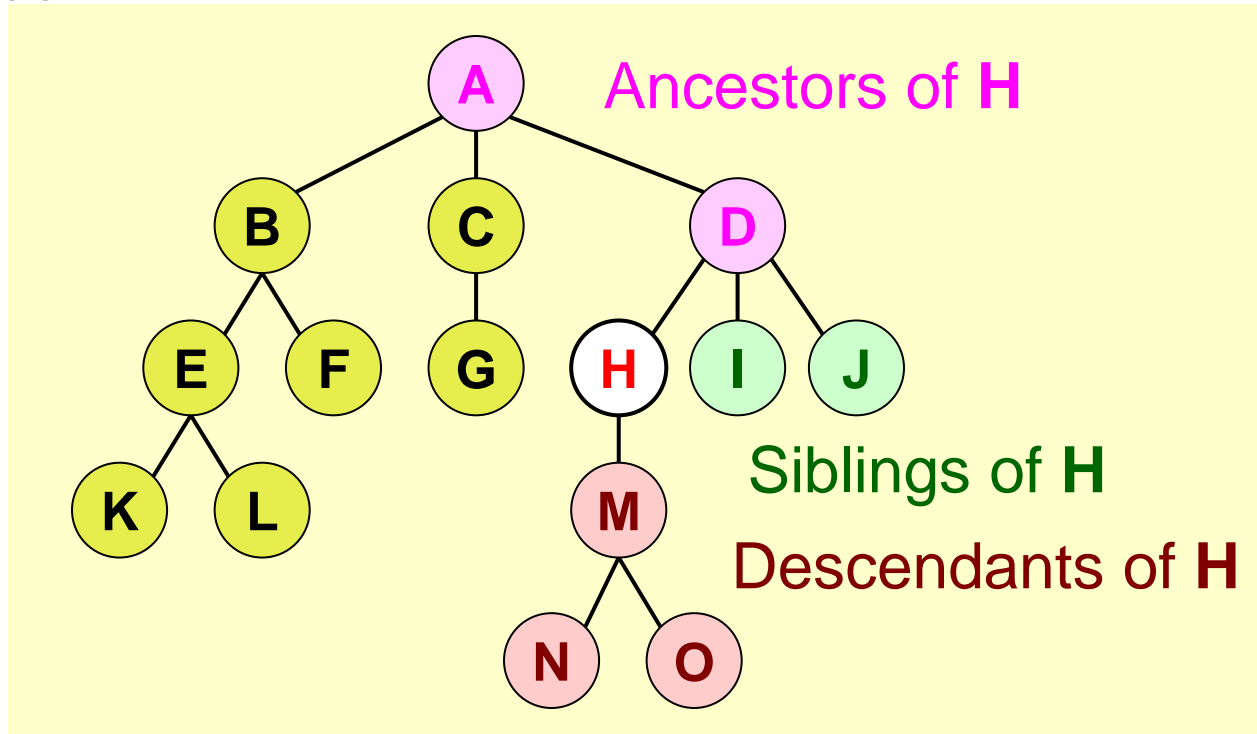
Terminologies and Properties

- Node **degree**: Number of subtrees of a node.
- Tree **degree**: Maximum degree of its nodes.
- **Leaf** nodes: Nodes with no subtrees (degree is zero).
- If a node r has subtrees T_1, T_2, \dots, T_n :
 - r is the **parent** of the roots of T_1, T_2, \dots, T_n .
 - The roots of T_1, T_2, \dots, T_n are **children** of r .
- Node **level**: (level of parent + 1); level of root = 1
- Tree **depth/height**: Maximum level of any node in the tree.



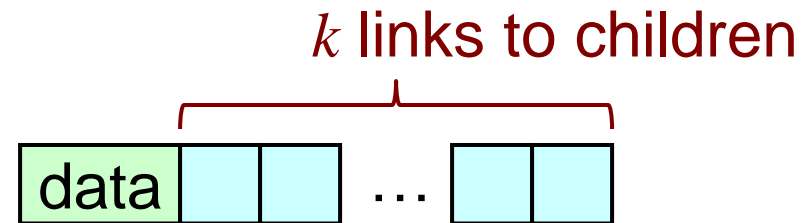
Terminologies and Properties

- Nodes with the same parent are **siblings** to one another.
- The **ancestors** of a node are all the nodes on the path from the root to the node.
- The **descendants** of a node are all the nodes in all its subtrees.



Representation of Tree Nodes

- A tree node may have links to many other tree nodes. (This is different from lists, where each node is linked to only one or two other nodes.)
- Example: One link for each possible child, assuming that the maximum degree is k .

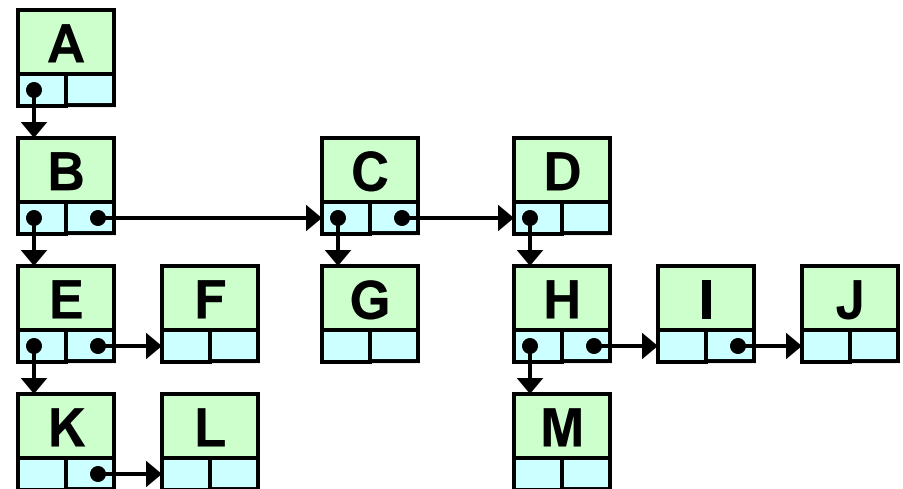
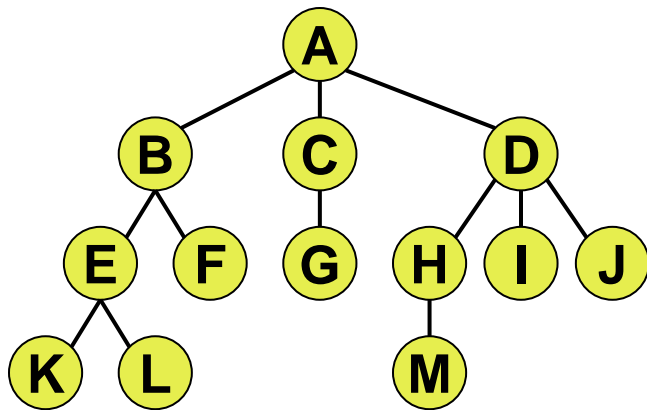


- Low flexibility / reusability.
- Waste of space. (Hint: compare the numbers of nodes and of the reserved spaces of links in the whole tree.)

Q: Can you think of a modification that can solve these problems?

Representation of Tree Nodes

- Let all the children of a tree node form a linked list.
- **Left-child-right-sibling** representation: A tree node has two links:
 - Link *left* points to the first node in its list of children;
 - Link *right* points to the next node in the list it belongs to.

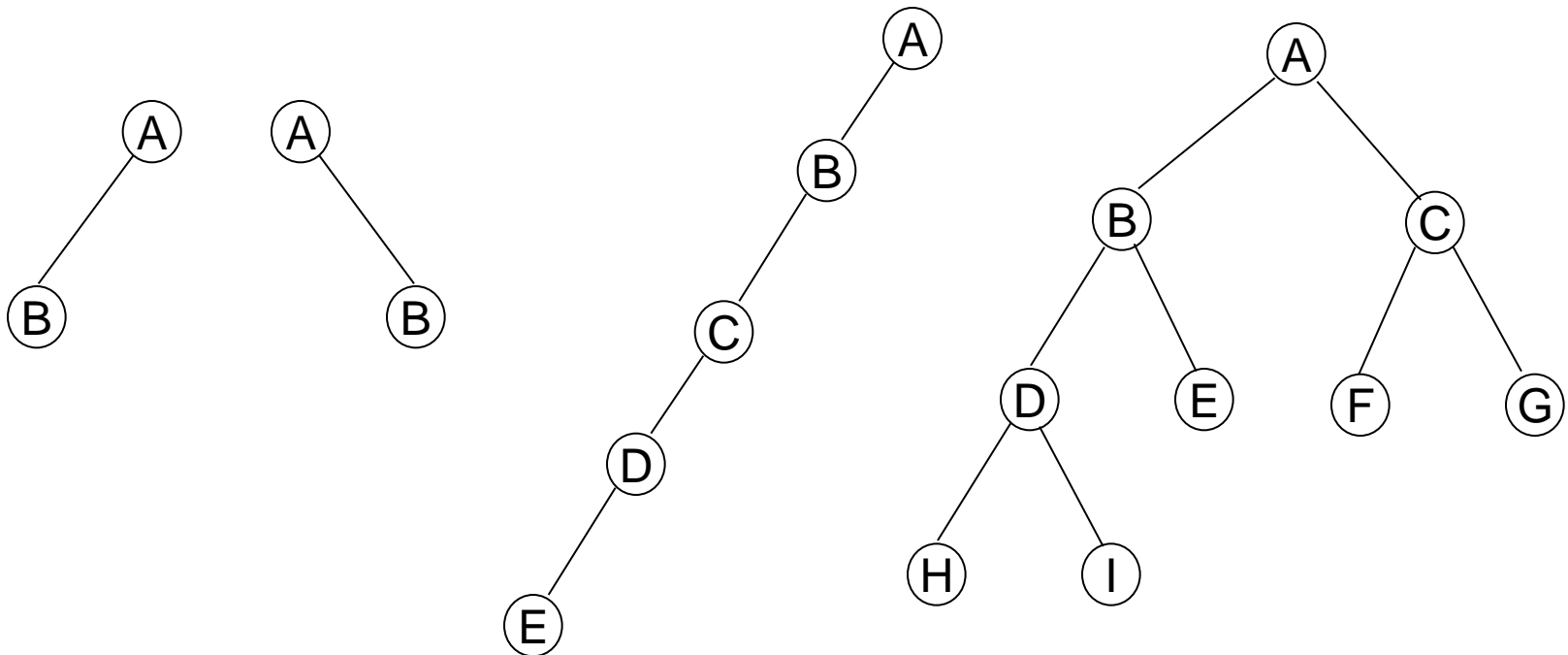


NULL links not shown

Q: What are the disadvantages of this representation?

Binary Trees

- A binary tree is a degree-2 tree (at most two children per node).
- The standard binary tree node uses the *left-child-right-child* representation (both links are to the children).
- Example binary trees:

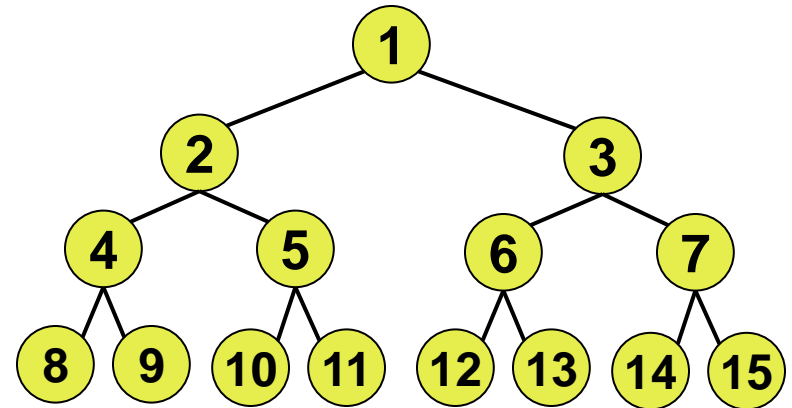


Binary Tree Properties

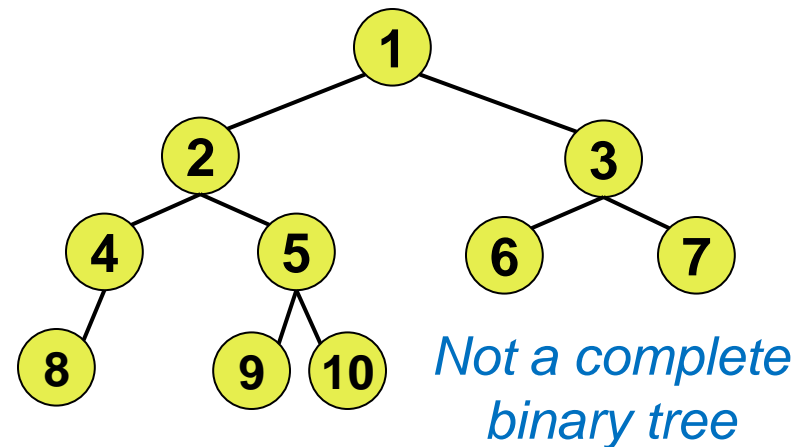
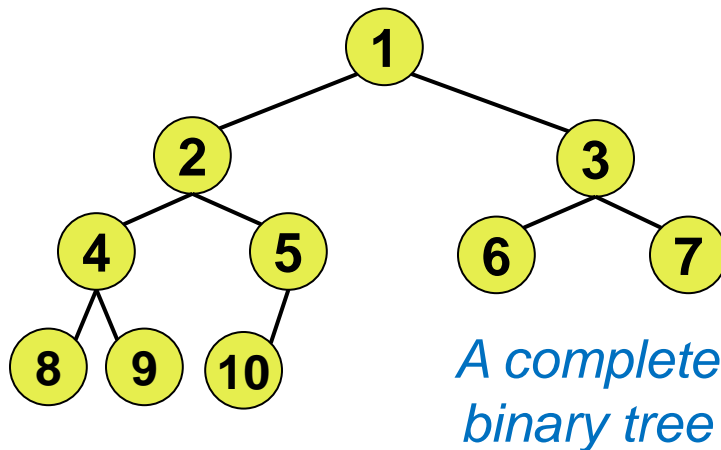
- Maximum number of nodes at level d : 2^{d-1} .
- Maximum number of nodes for a depth- k binary tree: $2^k - 1$.
 - A depth- k binary tree with $2^k - 1$ nodes is called a **full binary tree**.
- For any nonempty binary tree, if n_0 is the number of leaf nodes and n_2 the number of degree-2 nodes, then $n_0 = n_2 + 1$.
 - Proof: (consider the number of branches/links)

Complete Binary Trees

This is a depth-4 full binary tree, with the nodes numbered in a level-wise order (top-down and then left-to-right):

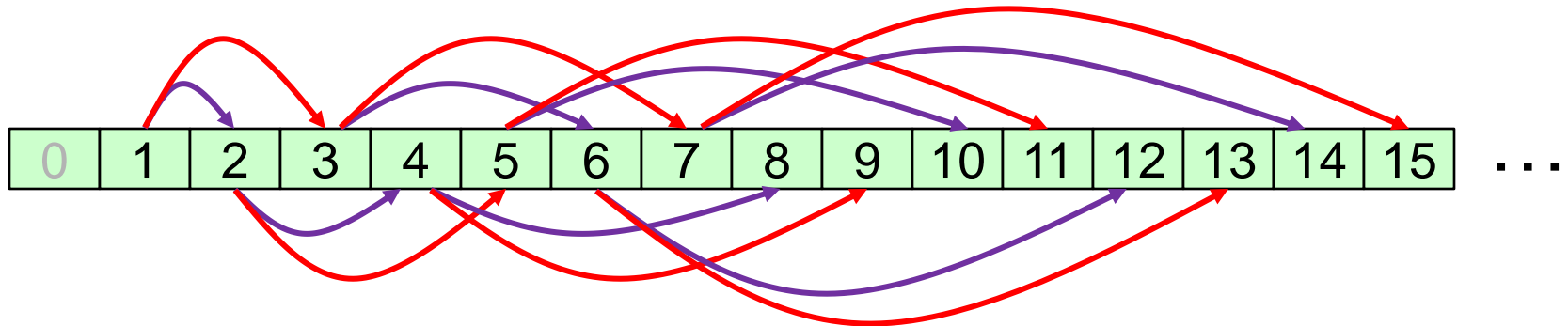
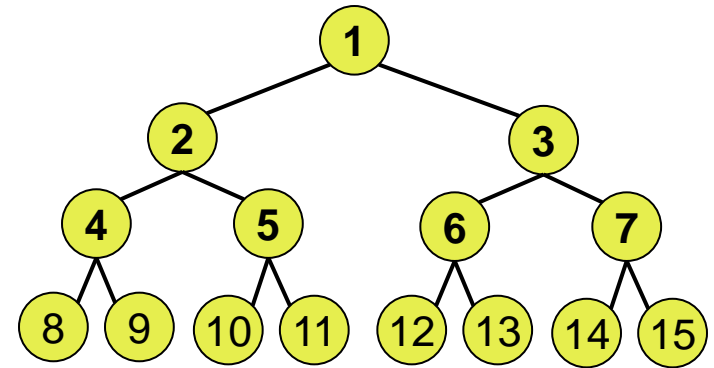


Complete binary tree: When its nodes are numbered in the same way, they are at the same respective positions in the tree as the corresponding nodes in a full binary tree.



Array Representation of Binary Trees

The way of numbering the nodes in a full binary tree inspires this representation:



Array Representation of Binary Trees

Properties:

- Advantage: Easy to find the parent and children for a given node:
 - If $i == 1$, node i is the root. Otherwise its parent is at $\lfloor i/2 \rfloor$.
 - The left child of node i is at $2i$.
 - The right child of node i is at $2i+1$.
- Disadvantages (very similar to those of using arrays to represent a list of items):
 - Waste of space, unless the tree is almost complete.
 - Array size? (What if nodes will be inserted and deleted dynamically?)

Linked Representation of Binary Trees

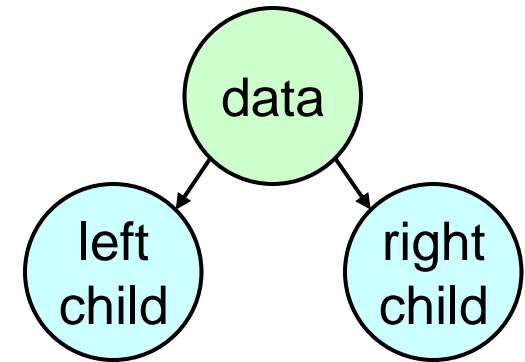
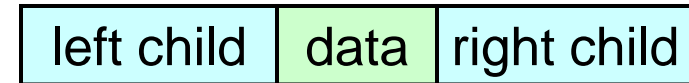
Binary Tree Class Template:

```
template<class T> class Tree;

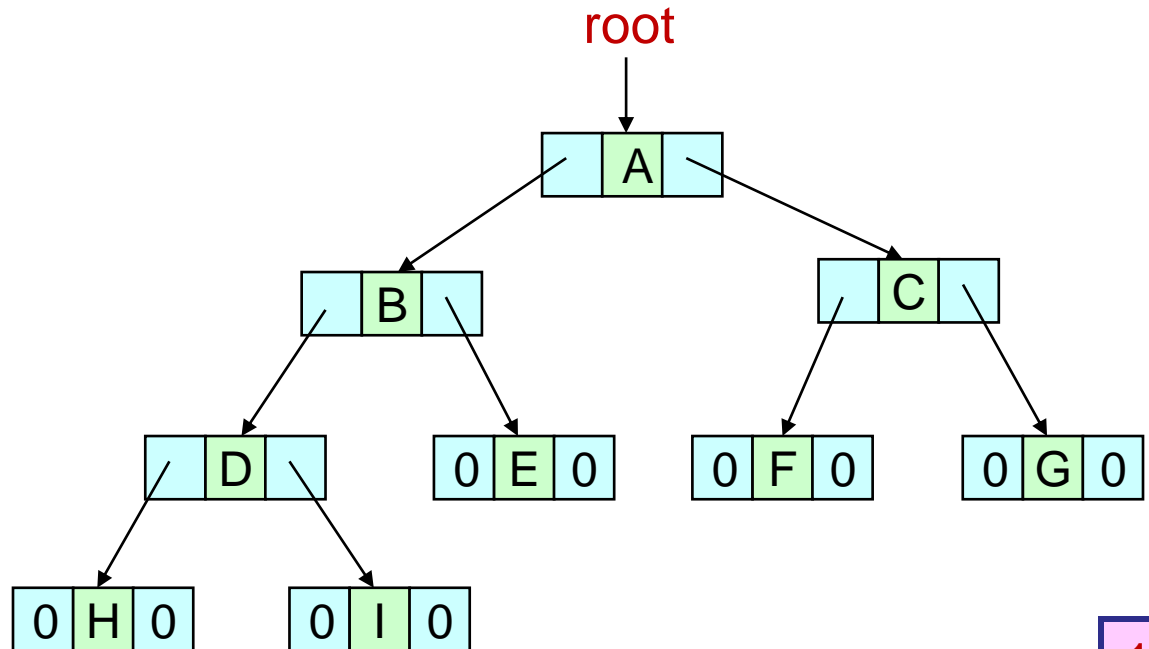
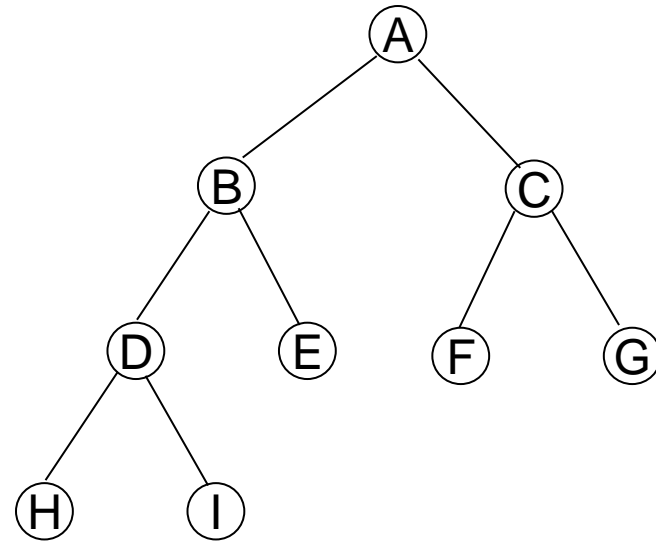
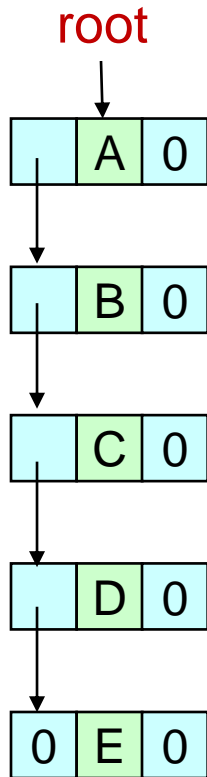
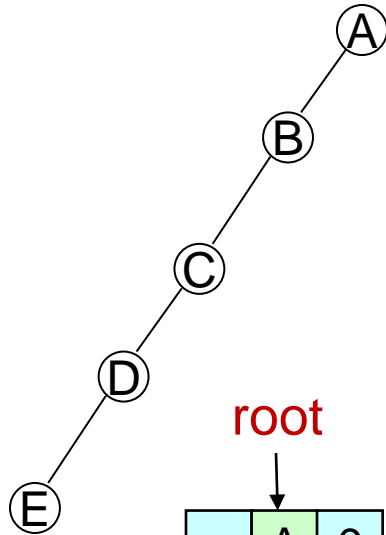
template<class T> class TreeNode {
friend class Tree<T>;
private:
    T data;
    TreeNode<T> *leftChild;
    TreeNode<T> *rightChild;
};

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

How we draw a node:



Linked Representation of Binary Trees

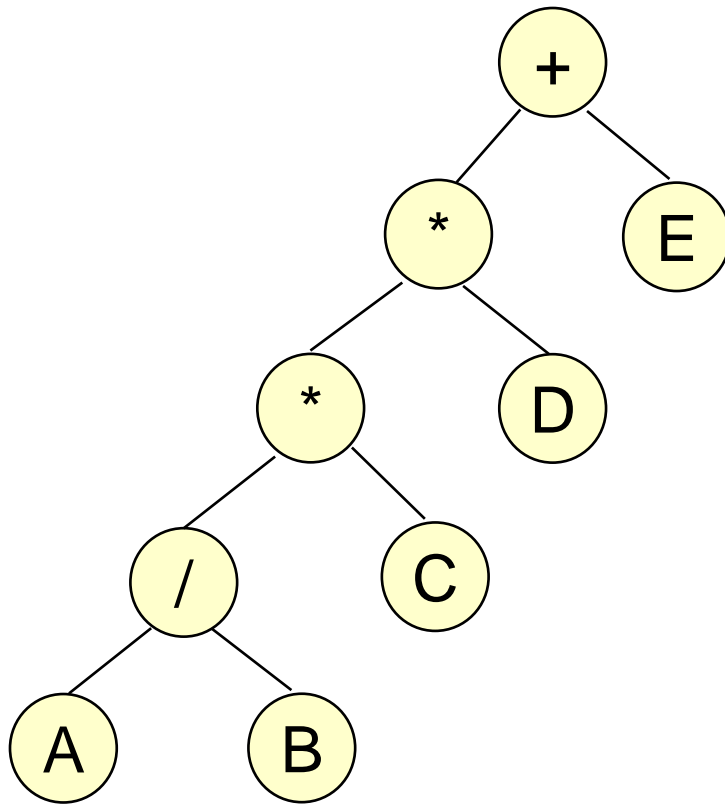


Binary Tree Traversal

- Traversal: Move along the tree and "visit" each node exactly once. (You can be at a node without visiting it.)
- Three possible moves when at a node: V (visiting the node), L (moving to the left child), and R (moving to the right child).
- ➔ Six possible combinations of traversal:
 - LVR, LRV, VLR, VRL, RVL, RLV
- Adopt the convention that L occurs before R ➔ only 3 traversals remain:
 - LVR (inorder), LRV (postorder), VLR (preorder)

Arithmetic Expression as a BT

Idea: An operator node has its operands as children.



inorder traversal (LVR)

$A / B * C * D + E$

preorder traversal (VLR)

$+ * * / A B C D E$

postorder traversal (LRV)

$A B / C * D * E +$

level-order traversal

$+ * E * D / C A B$

Binary Tree Traversal: Inorder

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

void Tree::inorder(TreeNode *CurrentNode)
// Workhorse traverses the subtree rooted at CurrentNode
// Declared as a private member function of Tree.
{
    if (CurrentNode) {
        inorder(CurrentNode -> LeftChild) ;
        cout << CurrentNode -> data; // visit the node
        inorder(CurrentNode-> RightChild) ;
    }
}
```

Binary Tree Traversal: Postorder

```
void Tree::postorder()  
// Driver calls workhorse for traversal of the entire tree.  
// Declared as a public member function of Tree.  
{  
    postorder(root) ;  
}  
  
void Tree::postorder(TreeNode *CurrentNode)  
// Workhorse traverses the subtree rooted at CurrentNode  
// Declared as a private member function of Tree.  
{  
    if (CurrentNode) {  
        postorder(CurrentNode -> LeftChild) ;  
        postorder(CurrentNode-> RightChild) ;  
        cout << CurrentNode -> data; // visit the node  
    }  
}
```

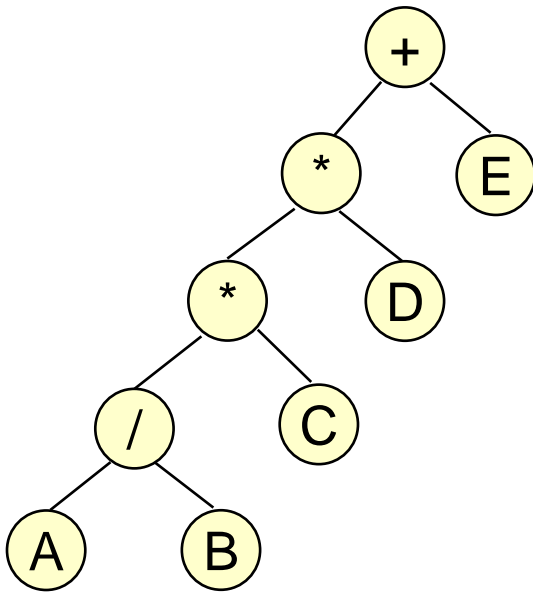
Binary Tree Traversal: Preorder

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

void Tree::preorder(TreeNode *CurrentNode)
// Workhorse traverses the subtree rooted at CurrentNode
// Declared as a private member function of Tree.
{
    if(CurrentNode) {
        cout << CurrentNode -> data; // visit the node
        preorder(CurrentNode -> LeftChild) ;
        preorder(CurrentNode-> RightChild) ;
    }
}
```

Binary Tree Traversal: Inorder

Tracing the operations of inorder traversal on the arithmetic expression example:



inorder traversal (LVR)

$A / B * C * D + E$

call to inorder	currentNode	action
1	+	
2	*	
3	*	
4	/	
5	A	
6	NULL	
5	A	cout
7	NULL	
4	/	cout
8	B	
9	NULL	
8	B	cout
10	NULL	
3	*	cout
11	C	
12	NULL	
11	C	cout
13	NULL	
2	*	cout
14	D	
15	NULL	
14	D	cout
16	NULL	
1	+	cout
17	E	
18	NULL	
17	E	cout
19	NULL	

Non-recursive Traversal (Inorder)

```
void Tree::NonrecInorder()  
// nonrecursive inorder traversal using a stack  
{  
    Stack<TreeNode*> s;          // declare and initialize the stack  
    TreeNode *CurrentNode = root;  
    while (1) {  
        while (CurrentNode) {    // move to left child  
            s.Push(CurrentNode); // add to stack  
            CurrentNode = CurrentNode->LeftChild;  
        }  
        if (!s.IsEmpty()) {      // stack is not empty  
            CurrentNode = s.Top();  
            s.Pop();  
            cout << CurrentNode->data << endl;  
            CurrentNode = CurrentNode->RightChild;  
        }  
        else break;  
    }  
}
```

Non-recursive Traversal

- The inorder version works by
 - Put the current node on the stack
 - If the current node has a left child
 - ◆ Go to the left child
 - Otherwise
 - ◆ Take the top node out of the stack as the current node
 - ◆ "Visit" the current node
 - ◆ Go to the right child if it exists
- Now consider how you can modify this procedure to do non-recursive preorder and postorder traversal.
- Also, you can think about the similarity and difference between this procedure and the maze problem.

Level-Order Traversal

- Use a queue instead of a stack.
- Nodes of lower levels are always visited before nodes of higher levels.
- The left child is visited before the right child.

```
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.Push(CurrentNode->LeftChild);
        if (CurrentNode->RightChild)
            q.Push(CurrentNode->RightChild);
        CurrentNode = *q.Front();
        q.Pop();
    }
}
```

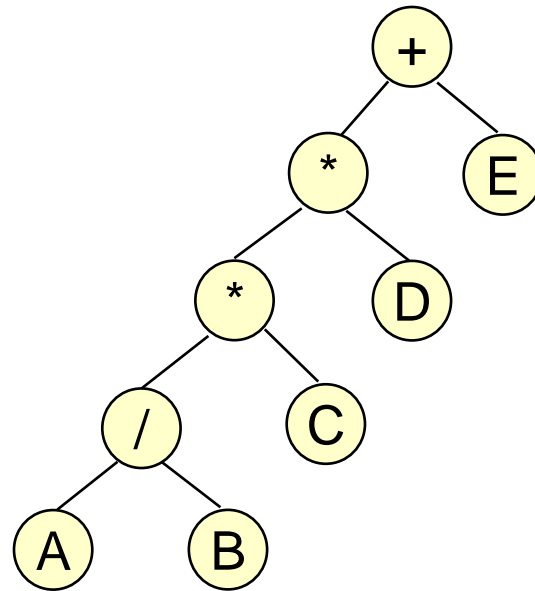
Copy a Binary Tree

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

TreeNode<T>* Tree::copy(TreeNode<T> *origNode) // workhorse
// returns a pointer to an exact copy of the binary
// tree rooted at origNode.
{
    if (origNode) {
        TreeNode<T> *temp = new TreeNode<T>;
        temp->data = origNode->data;
        temp->leftChild = copy(origNode->leftChild);
        temp->rightChild = copy(origNode->rightChild);
        return temp;
    }
    else return 0;
}
```


Example: Expression Evaluation

- In chapter 3, we studied the evaluation of postfix arithmetic expressions.
- Equivalently, we can evaluate an "expression tree" using postorder traversal.



$AB / C * D * E +$

- Here we will apply this method to the satisfiability problem.

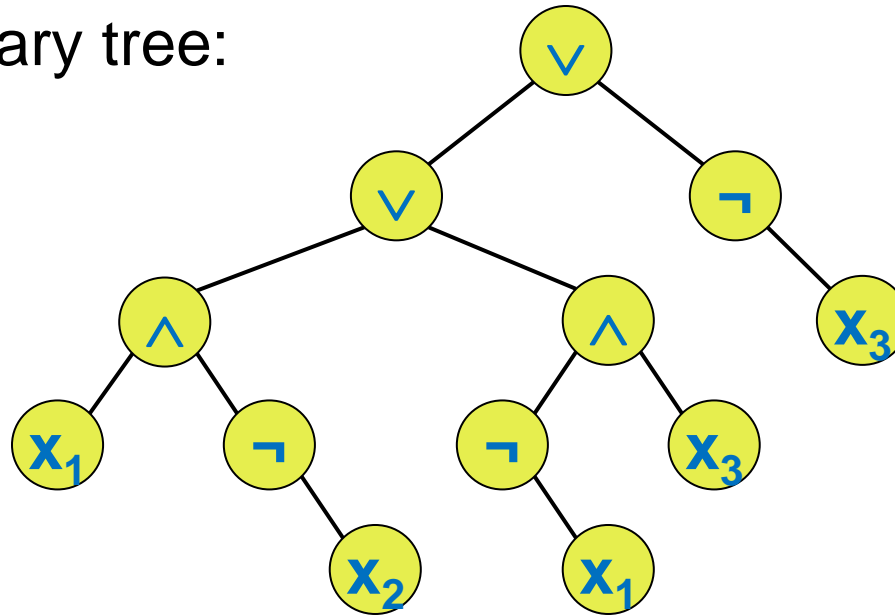
Propositional Calculus Expression

- 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 ($\neg > \wedge > \vee$).
- Example: $x \wedge (\neg y \vee z)$
- **Satisfiability problem**: Is there existing a set of assignments to the variables such that the expression is true?

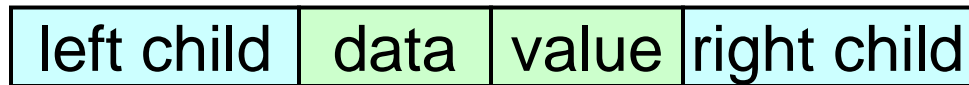
Propositional Calculus Expression

Here the expression $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$

is shown as a binary tree:



Tree Node for Expression Evaluation



- data: operators (non-leaf nodes) or variables (leaf nodes)
- value: evaluated value (true or false)

Determining Satisfiability

Pseudo-code for determining the satisfiability of an expression ('formula' is the binary tree representing the expression, and 'rootvalue' is its function returning the value at its root.)

```
For each of the  $2^n$  truth value combinations for the n variables
{
    Set the variables' values by their values in the combination;
    Evaluate the formula by traversing the tree in postorder;
    if (formula.rootvalue()) // value at the tree's root is true
    { cout << combination; return; }
}
cout << "expression not satisfiable";
```

Expression Evaluaion

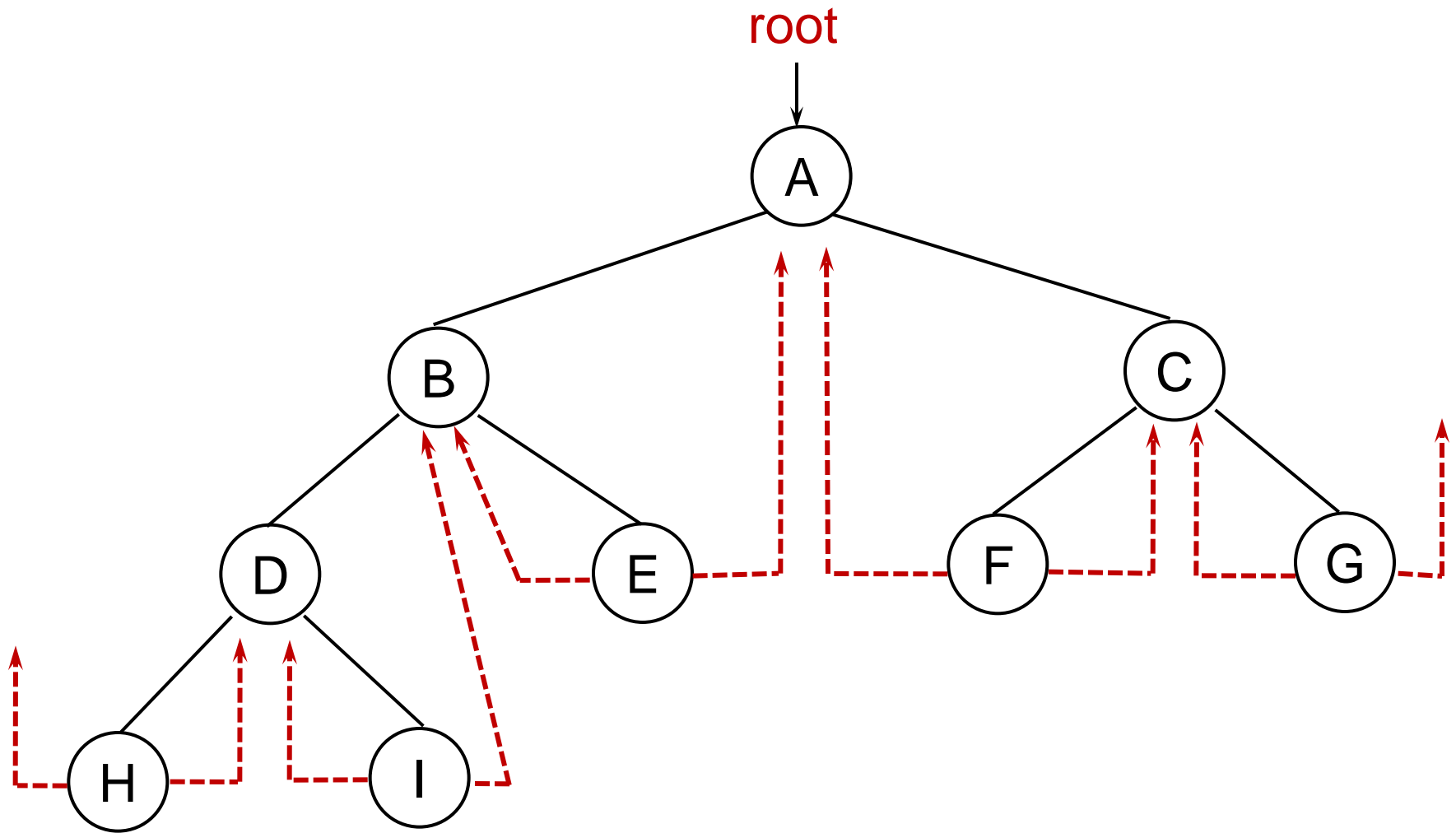
```
void SatTree::PostOrderEval (SatNode *s) // workhorse
{
    if (s) {
        PostOrderEval (s->leftChild);
        PostOrderEval (s->rightChild);
        switch (s->data) {
            case LogicalNot:
                s->value = !s->rightChild->value;
                break;
            case LogicalAnd:
                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

- Q: How many unused links (null **leftChild** and **rightChild** pointers) are there in a binary tree with n nodes?
- **Threads**: Using these pointers to speed up inorder traversal.
 - If a node has no left child, its **leftChild** pointer points to the previous node in inorder traversal.
 - If a node has no right child, its **rightChild** pointer points to the next node in inorder traversal.
 - Add two flags (**leftThread** and **rightThread**) to each node to indicate whether its pointers are threads.

leftThread	leftChild	data	rightChild	rightThread
------------	-----------	------	------------	-------------

Threaded Binary Trees: Example

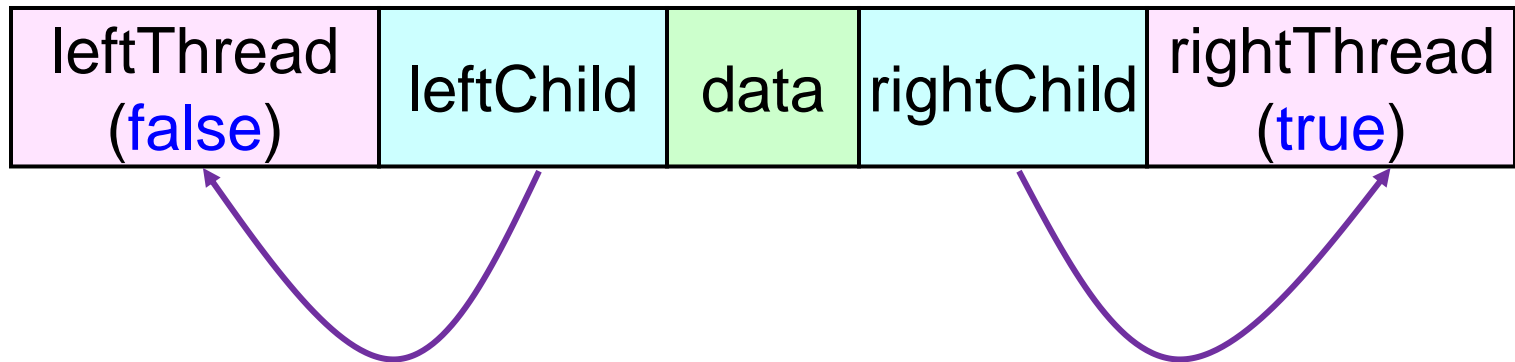


Q: Give the inorder traversal of this tree.

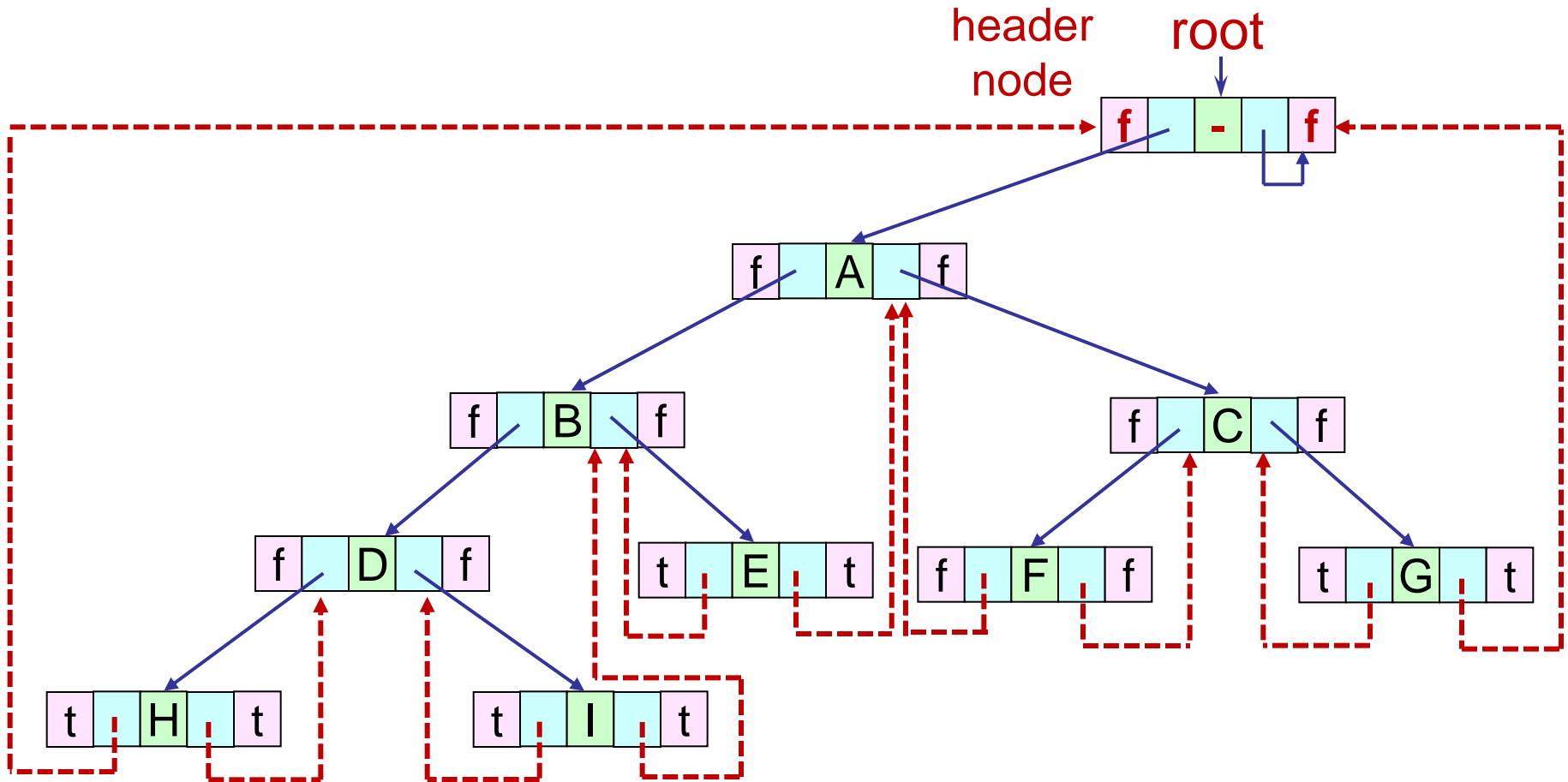
Threaded Binary Trees with Head Nodes

- The previous example has two dangling threads.
- Use a header node (and set the original root as the left child of the header node).

An empty threaded binary tree with a header node:



Threaded Binary Trees with Head Nodes



Inorder Traversal with Threads

- Previously, non-recursive inorder traversal requires a stack.
 - Purpose: To remember the path for back-tracing.
- With threads, the stack becomes unnecessary because we do not need to back-trace.
 - The right subtree is non-empty: The inorder successor (the next node in inorder traversal) is the leftmost node in the right subtree.
 - The right subtree is empty: The inorder successor is just the node pointed to by the right thread.

Inorder Traversal with Threads

```
template<class T>
ThreadedNode<T>*
ThreadedTree<T>::InorderSuccessor (ThreadedNode<T> *x)
{
    ThreadedNode<T> *temp = x->rightChild;
    if (!x->rightThread) { // x has a right subtree
        while (!temp->leftThread)
            temp = temp->leftChild;
    }
    return (temp == root) ? NULL : temp;
}
```

■ Additional notes:

- **x->rightChild** is the root → **x** is the rightmost node in the tree; **NULL** returned.
- **x** is the header node → The function returns the leftmost node of the tree; this is how we initialize the inorder traversal of the whole tree.

Priority Queues

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

Priority Queue ADT

```
template<class T> class MaxPQ
{
public:
    ~MaxPQ();
    bool IsEmpty() const;
    T& Top() const;
    void Push(const T&);
    void Pop();
};
```

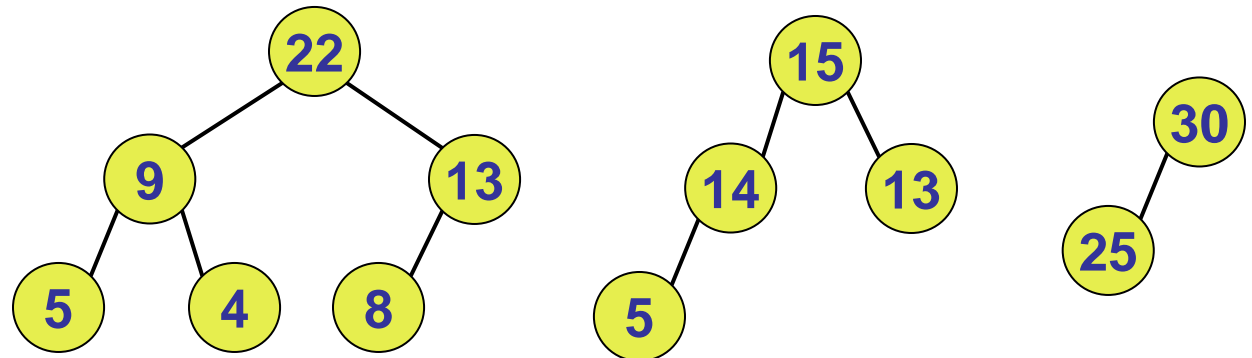
Representations of Priority Queues

Representation	Complexity (Insertion)	Complexity (Deletion)
Unsorted linked list	$O(1)$	$O(n)$
Sorted linked list	$O(n)$	$O(1)$
Max/Min heap	$O(\log n)$	$O(\log n)$

Max/Min Heaps

- Assume that each node has a key value.
- **Max tree**: y is a child of $x \rightarrow \text{key}(x) \geq \text{key}(y)$.
 - \rightarrow The root has the largest key.
- **Max heap**: a complete binary tree that is a max tree.
- **Min tree**: y is a child of $x \rightarrow \text{key}(x) \leq \text{key}(y)$.
 - \rightarrow The root has the smallest key.
- **Min heap**: a complete binary tree that is a min tree.

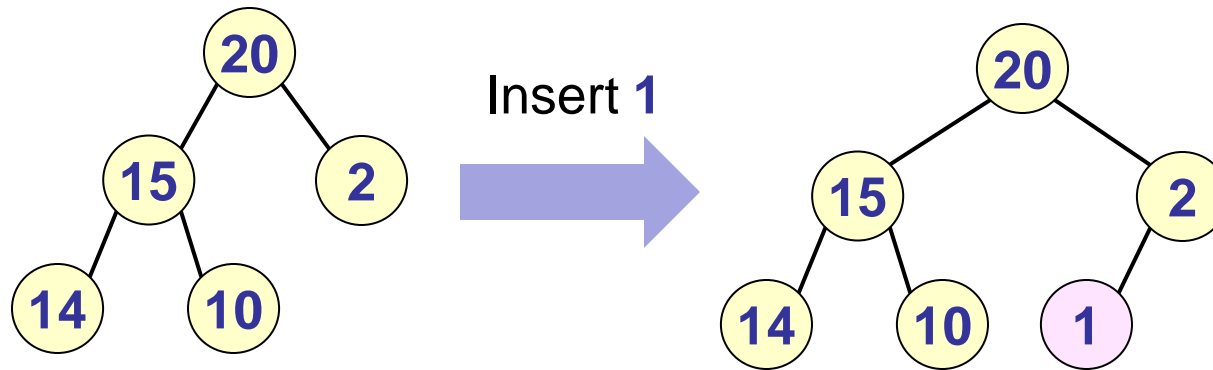
Example
Max heaps:



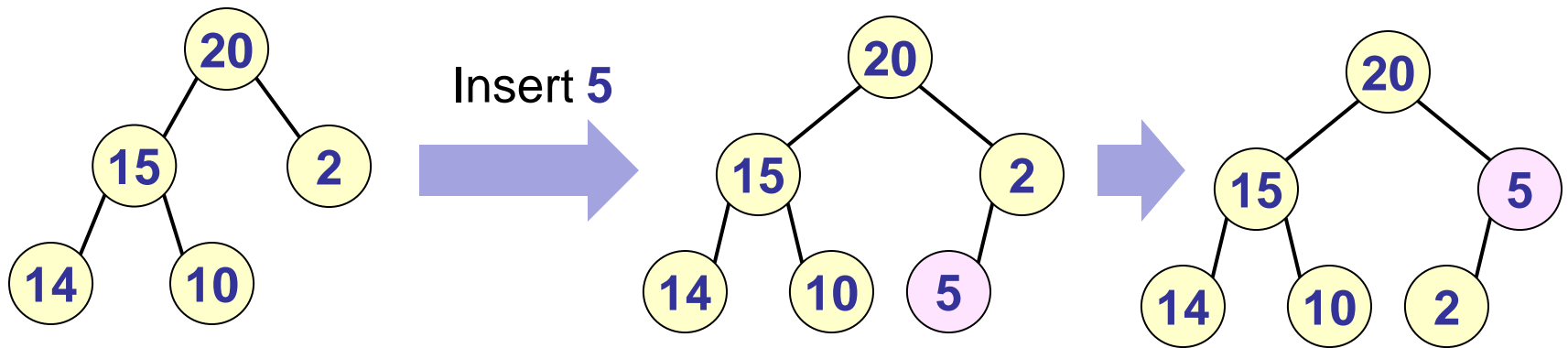
Max Heaps: Insertion

- Insert the new node at the next location of a complete binary tree.
- If necessary, swap the inserted node with its parent to ensure the "max heap" property. Repeat this step until the whole tree is a max heap.
- Time complexity: $O(\log n)$, $n = \text{\#nodes in heap}$
- The array representation of binary trees is used here to simplify the task of finding the parent of a node.

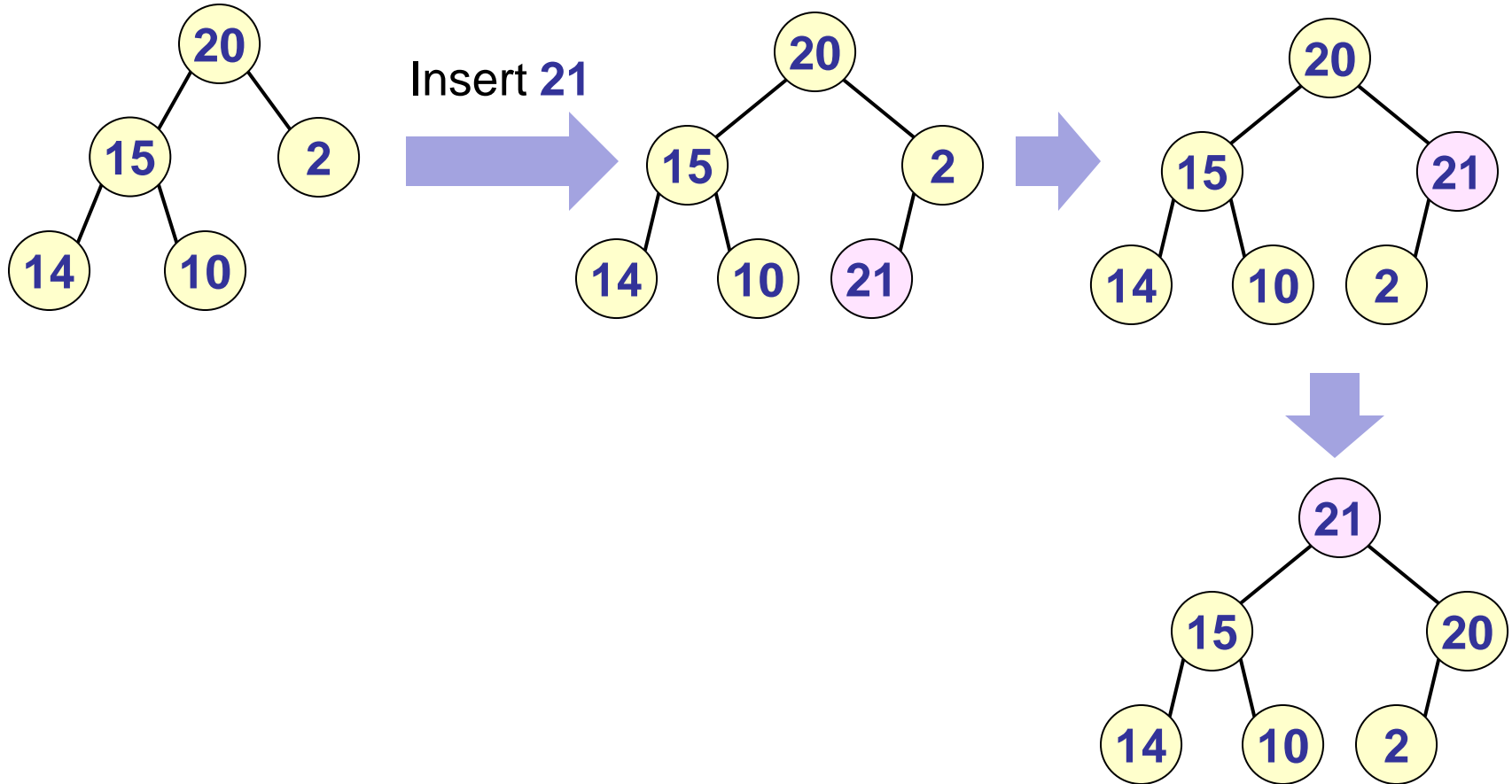
Max Heaps: Insertion



Max Heaps: Insertion



Max Heaps: Insertion



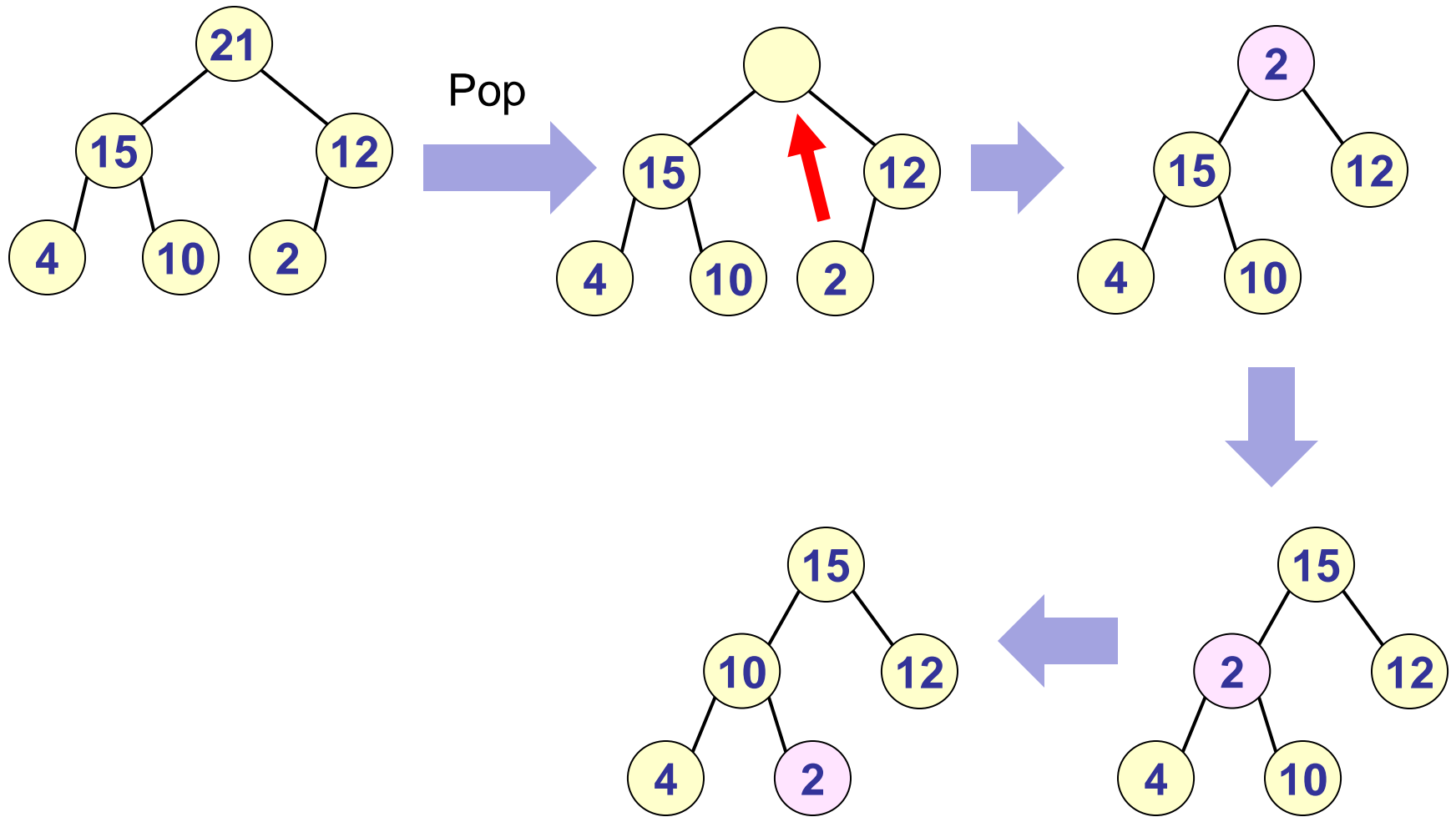
Max Heaps: Insertion

```
template<class T>
void MaxHeap<T>::Push(const T &x)
// array "heap" holds the items
// a field 'key' in type T holds the key value
{
    if (n == MaxHeapSize) { // n: #items in heap
        heap_full(); // exception handler
    }
    n++;
    int i = n; // current position of the inserted item
    while (i > 1 && x.key > heap[i/2].key) {
        heap[i] = heap[i/2]; // move parent down
        i /= 2;
    }
    heap[i] = x;
}
```

Max Heaps: Deletion

- The root node is to be deleted, thus creating a vacant space.
- The last node (in the order of level-order traversal) is moved to the root. This keeps the tree a complete binary tree. Let's call this node **x**.
- If necessary, swap **x** with one of its children to maintain the "max heap" property. Repeat this step until the key of **x** is no less than the keys of its children.
- Time complexity: $O(\log n)$, $n = \text{\#nodes in heap}$

Max Heaps: Deletion



Max Heaps: Deletion

```
template<class T>
void MaxHeap<T>::Pop()
{ // n: #items in heap
    if (n == 0) heap_empty(); // exception handler
    T e = heap[n]; // last item in heap
    n--;
    if (n > 0) {
        int i = 1; // position to move the last item to
        int j = 2; // child of item i
        while (j <= n) {
            // set j to the child of i with the larger key
            if (j < n && heap[j].key < heap[j+1].key) j++;
            // exit the loop if "max heap" condition is satisfied
            if (heap[j].key <= e.key) break;
            heap[i] = heap[j]; // move item j up one level
            i = j;  j = i * 2; // move i down on level
        }
        heap[i] = e;
    }
}
```


Key-Element Pairs

- Purpose: Key values are used to identify particular items in a collection of objects.
 - Example: student ID → student
 - Example: room number → room
- A collection of <key, element> pairs is called a dictionary.
- Keys are often distinct (e.g., student IDs).
- Sometimes multiple elements can have the same key (e.g., when using "year" as the key to "historical events").

Key-Element Pairs

- Important operations of a collection of key-element pairs:
 - Insert an element
 - Delete a particular element (by key or by rank)
 - Search for a particular element (by key or by rank)
- Apparently, we need a data structure that have its elements organized by keys.

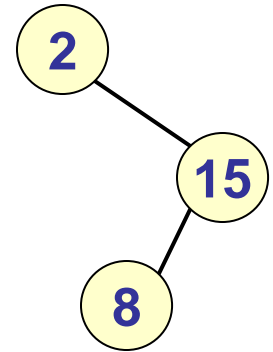
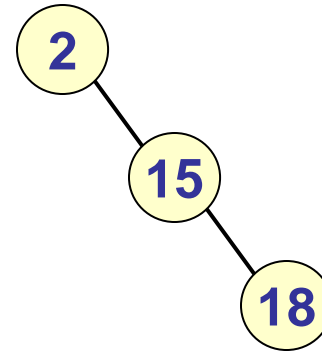
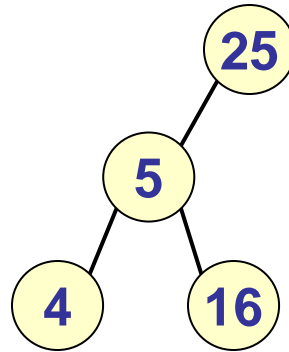
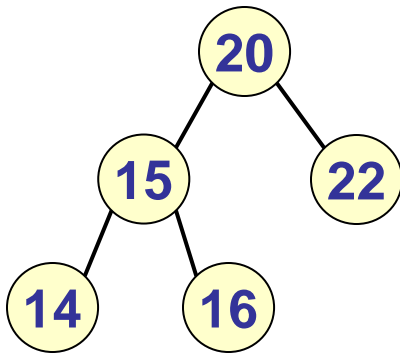
Q: How well does a sorted linked list work here?
Consider its time complexities.

Binary Search Trees (BSTs)

- Each tree node has a key.
- We assume here that the keys are distinct.
- For any node x :
 - For every node y in the left subtree of x , $\text{key}(y) < \text{key}(x)$.
 - For every node y in the right subtree of x , $\text{key}(y) > \text{key}(x)$.

Binary Search Trees (BSTs)

Some example BSTs



Search in BSTs

Similar to the binary search algorithm in chapter 1, we have iterative and recursive versions here.

Iterative BST search:

```
template <class K, class E>
BSTNode<K,E>* BST<K,E>::Search(const K& k)
{
    BSTNode<K,E> *t = root;
    while (t) {
        if (k == t->key) return t; // found
        t = (k > t->key) ? t->rightChild : t->leftChild;
    }
    return 0;
}
```

Search in BSTs

Recursive BST search:

```
template <class K, class E>
BSTNode<K,E>* BST<K,E>::Search(const K& k)
{ // driver
    return Search(root, k);
}

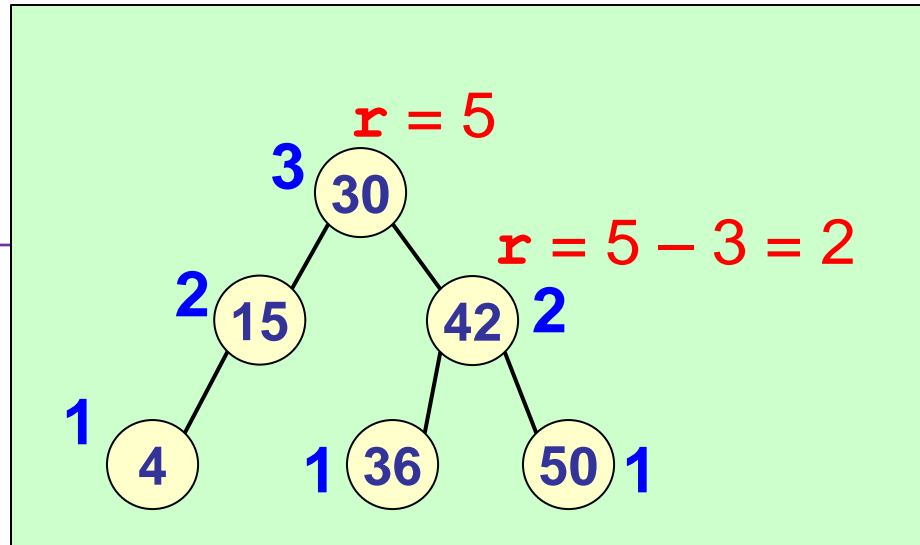
template<class K, class E>
BSTNode<K,E>* BST<K,E>::Search(BSTNode<K,E>* t,
                                const K& k)
{ // workhorse
    if (!t) return 0;
    if (k < t->key) return Search(t->leftChild, k);
    if (k > t->key) return Search(t->rightChild, k);
    return t; // k == t->key
}
```

Search a BST by Rank

- The **rank** of a node is its position during inorder traversal.
 - The inorder traversal of a BST generates a sorted list of all its nodes.
- We can search a BST by rank efficiently by adding an additional field **leftSize**.
 - **leftSize** = (#nodes in the left subtree) + 1.

Search a BST by Rank

```
template <class K, class E>
BSTNode<K,E>* BST<K,E>::SearchRank(int r)
{
    BSTNode<K,E>* t = root;
    while (t) {
        if (r == t->leftSize) return t;
        if (r < t->leftSize) t = t->leftChild;
        else {
            r -= t->leftSize;
            t = t->rightChild;
        }
    }
    return 0;
}
```



Insertion into a BST

- To ensure distinct keys, a search by key is executed.
 - If the key is already in the tree, just update its associated element. No new node is added to the tree.
 - If the key is not found in the tree and p is the last node visited during the search, add the new node as a child of p .
 - Update the *leftSize* field if applicable. (How do you do this?)

Insertion into a BST

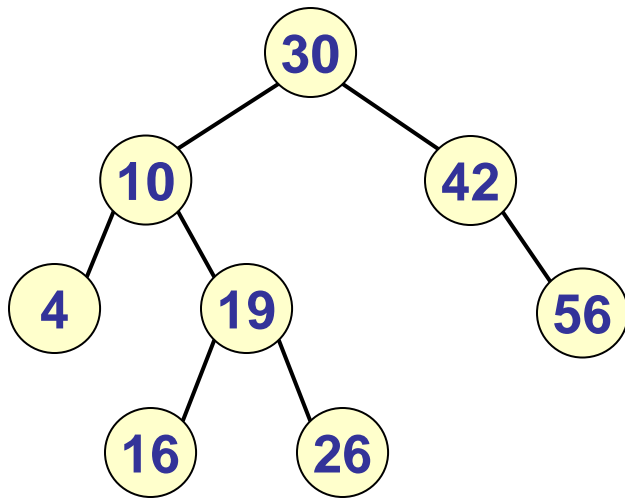
Modification of the BST search to do insertion:

```
template <class K, class E>
void BST<K,E>::Insert(const K& k, const E& e)
{
    BSTNode<K,E> *t = root, *p = 0;
    while (t) {
        if (k == t->key) { // duplicate key
            t->element = e; return; // just update element
        }
        p = t; // remember the parent of next t
        t = (k > t->key) ? t->rightChild : t->leftChild;
    }
    // insert a node
    t = new BSTNode<K,E>(k,e);
    if (root) // tree not empty
        if (k < p->key) p->leftChild = t;
        else p->rightChild = t;
    else root = t;
}
```

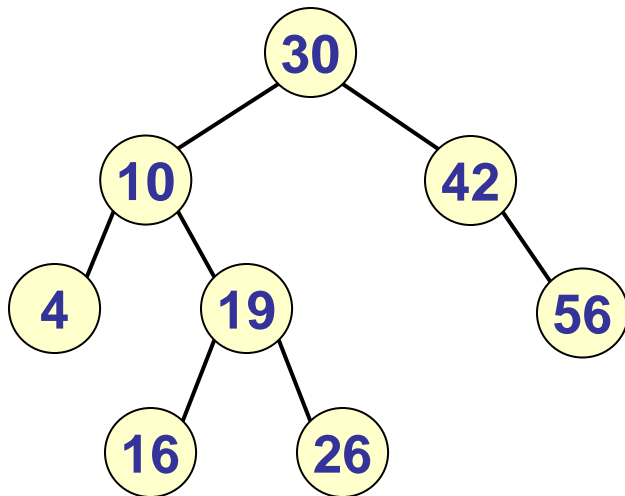
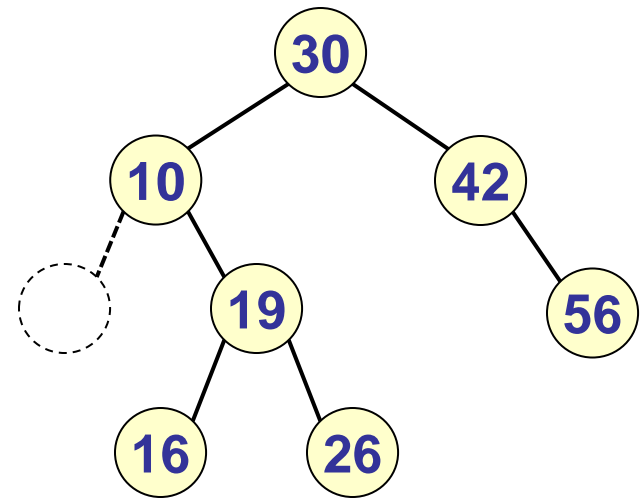
Deletion from a BST

- Need to first search for the node to delete by key.
- Let's assume that node x is to be deleted:
 - x is a leaf node \rightarrow just delete it
 - x has only one child $y \rightarrow y$ takes the place of x
 - x has two children:
 - ◆ Choose node z , which is either the leftmost node in its right subtree or the rightmost node in its left subtree.
 - ◆ Replace x with z .
 - ◆ Replace z with its only child (if any).

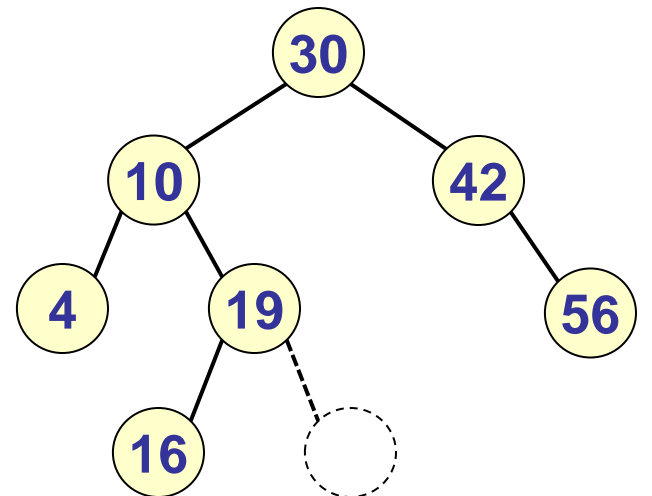
Deletion from a BST



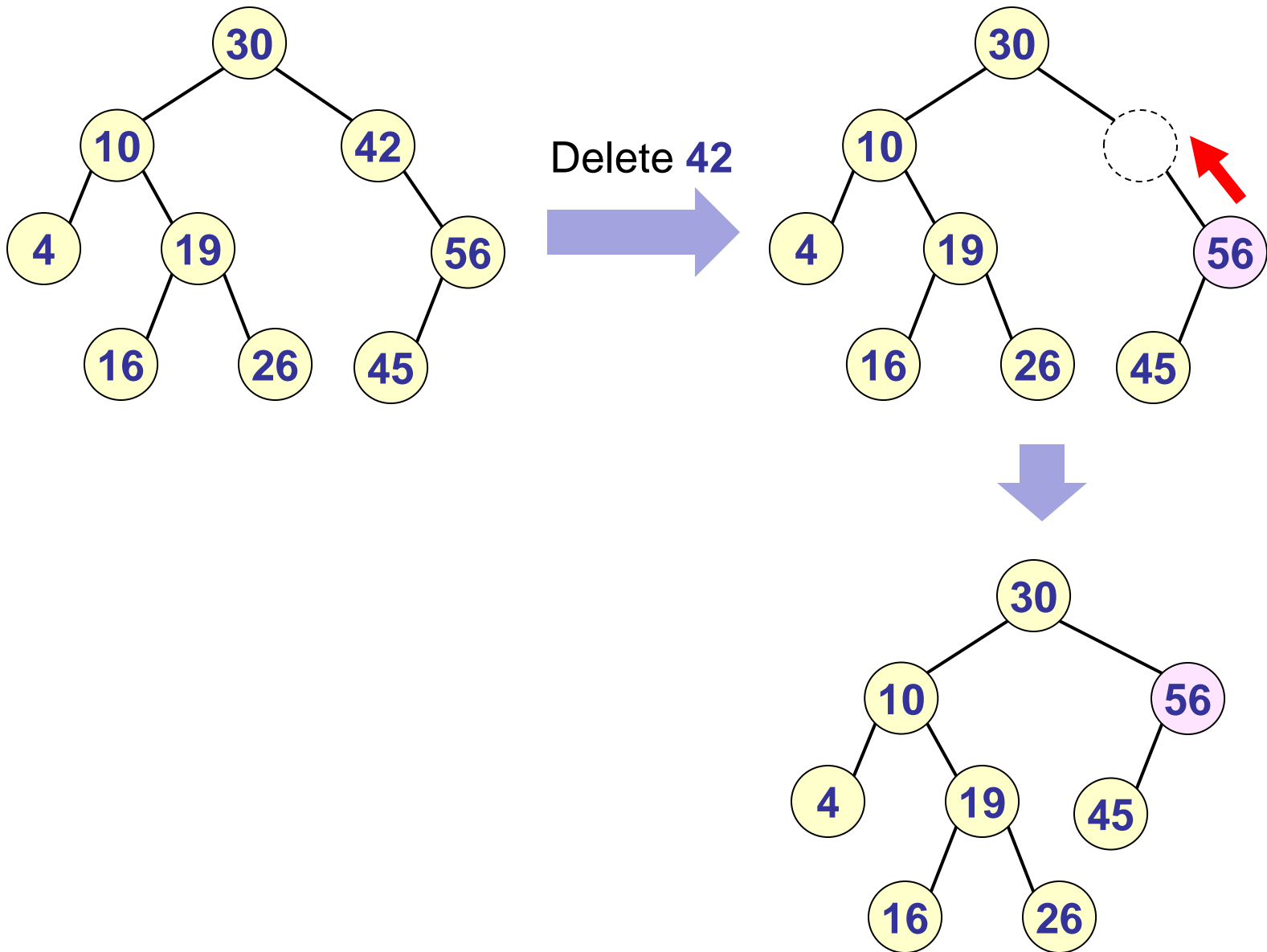
Delete 4



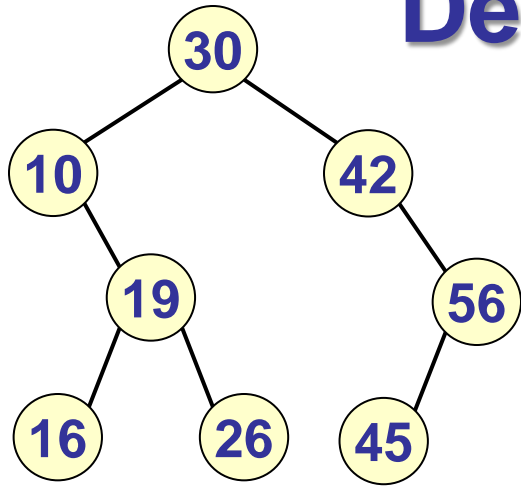
Delete 26



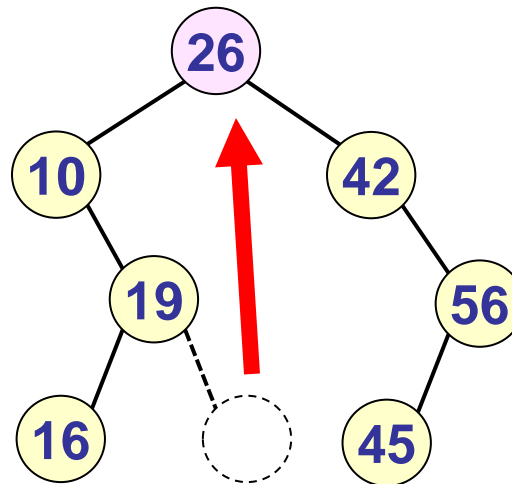
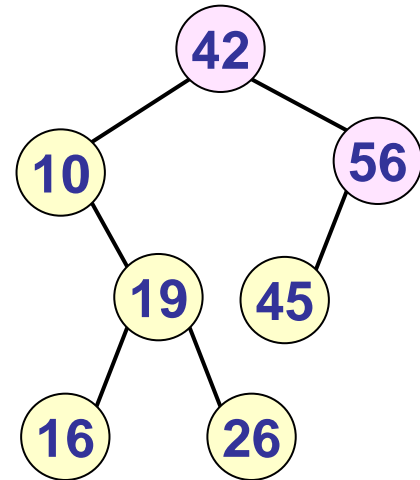
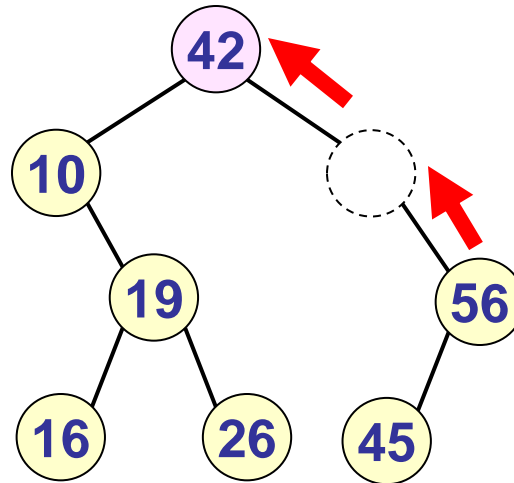
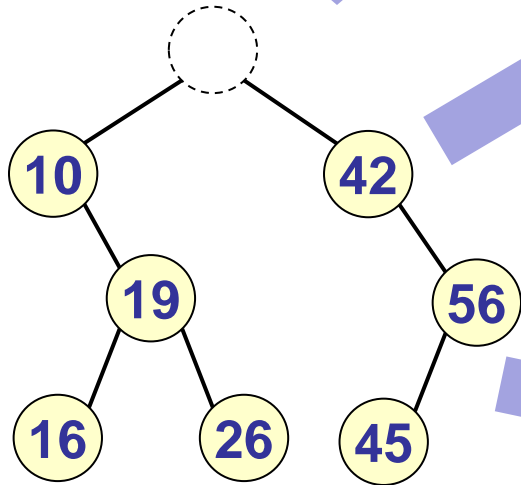
Deletion from a BST



Deletion from a BST



Delete 30

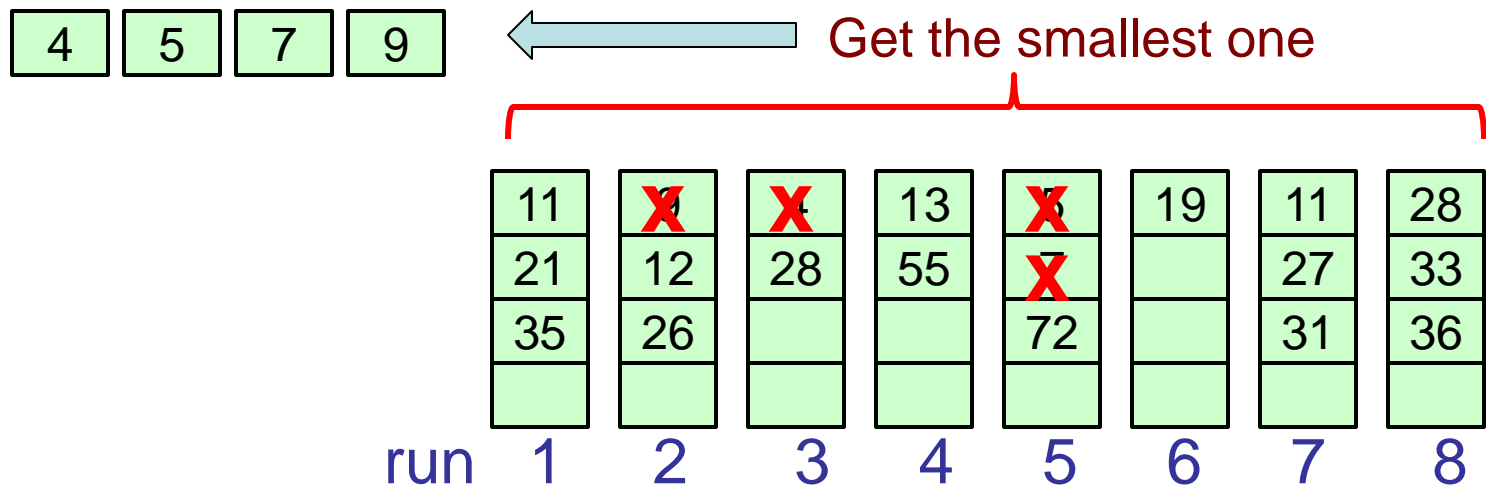


Notes on BSTs

- Time complexity (#nodes = n , tree height = h):
 - Search:
 - Insertion:
 - Deletion:
- Worst-case and average complexities:
 - Insert 1, 2, ..., n into a BST in this order
 - Insert 1, 2, ..., n into a BST in random order
- To make the worst-case complexity $O(\log n)$, we need balanced BSTs (covered in chapter 10).
- STL class templates:
 - ***map*** (actually a red-black tree; chp. 10): distinct keys
 - ***multimap***: map with non-distinct keys allowed

Selection Trees

- Assume that we have k lists (called *runs* in this section), with the items (called *records* in this section) in a list sorted in the small-to-large order according to their key values.
- Task: In each iteration, extract the record with the smallest key among all the records in all the lists.



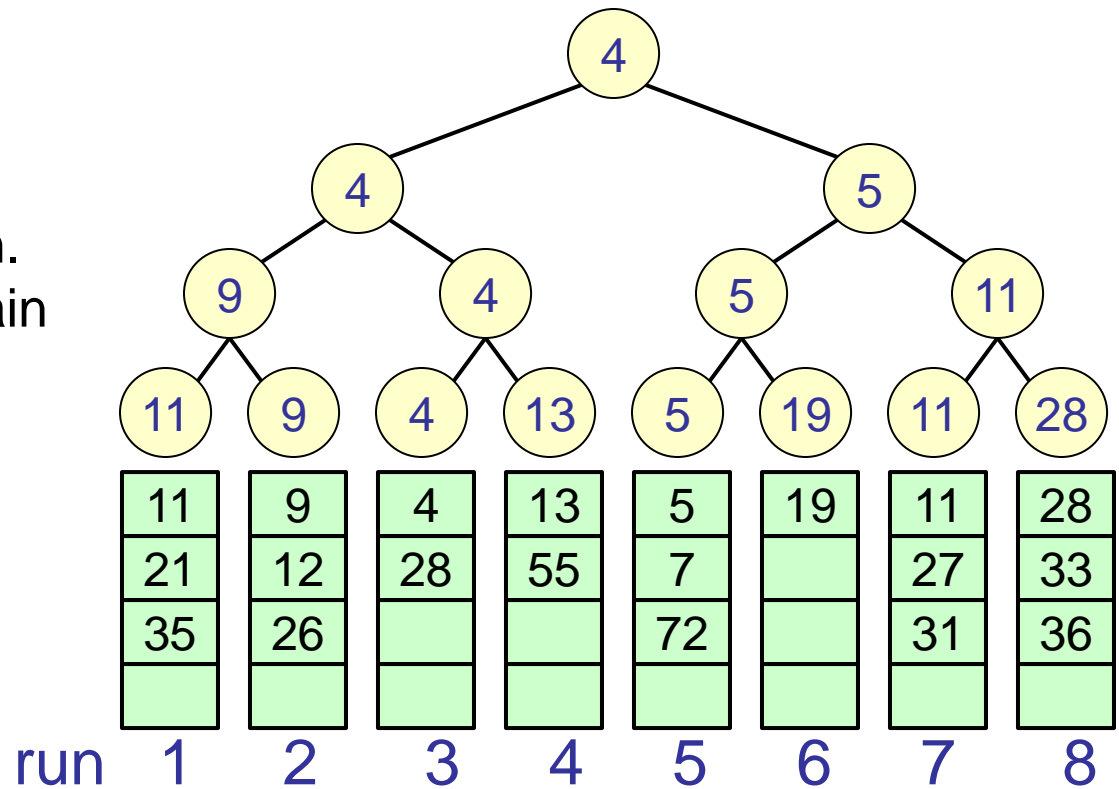
Q: Consider a possible method and its complexity.

Winner Trees

A type of selection tree where each node remembers the "winner" (the one to be selected) of its two children.

Let us initialize the tree with the first record of each run.

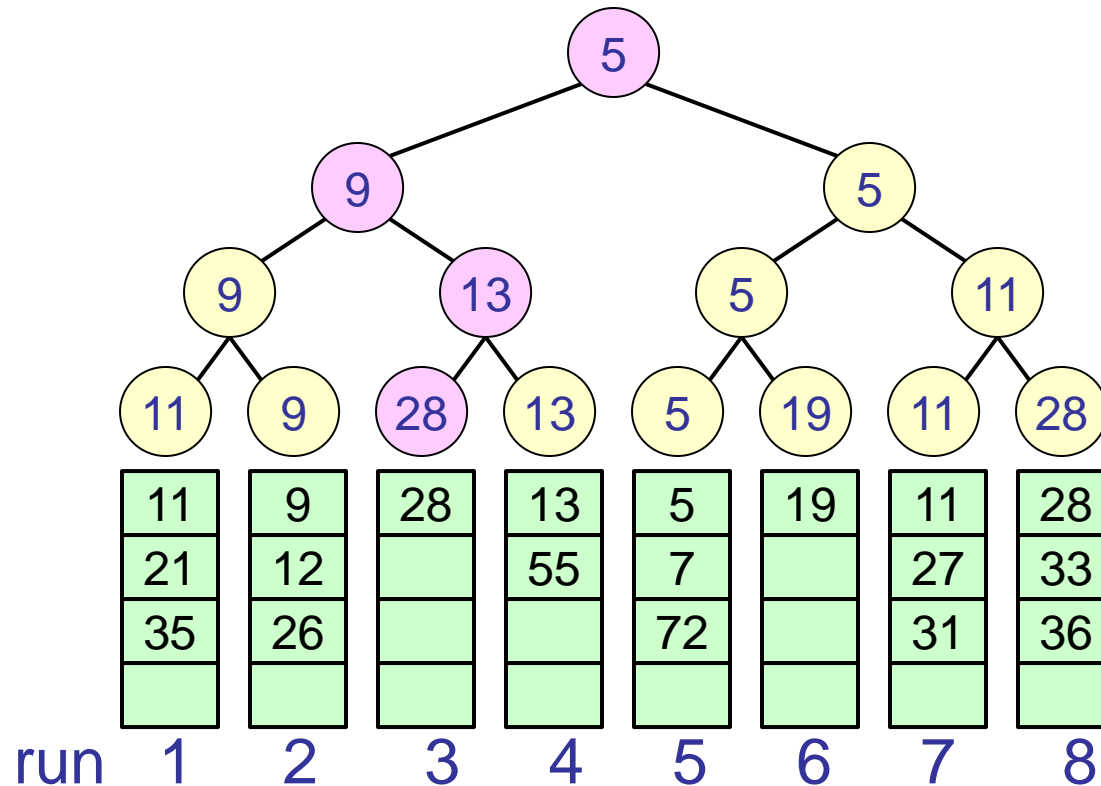
Note: Key values of winners are shown here for illustration. The tree nodes actually contain pointers to the items.



Winner Trees

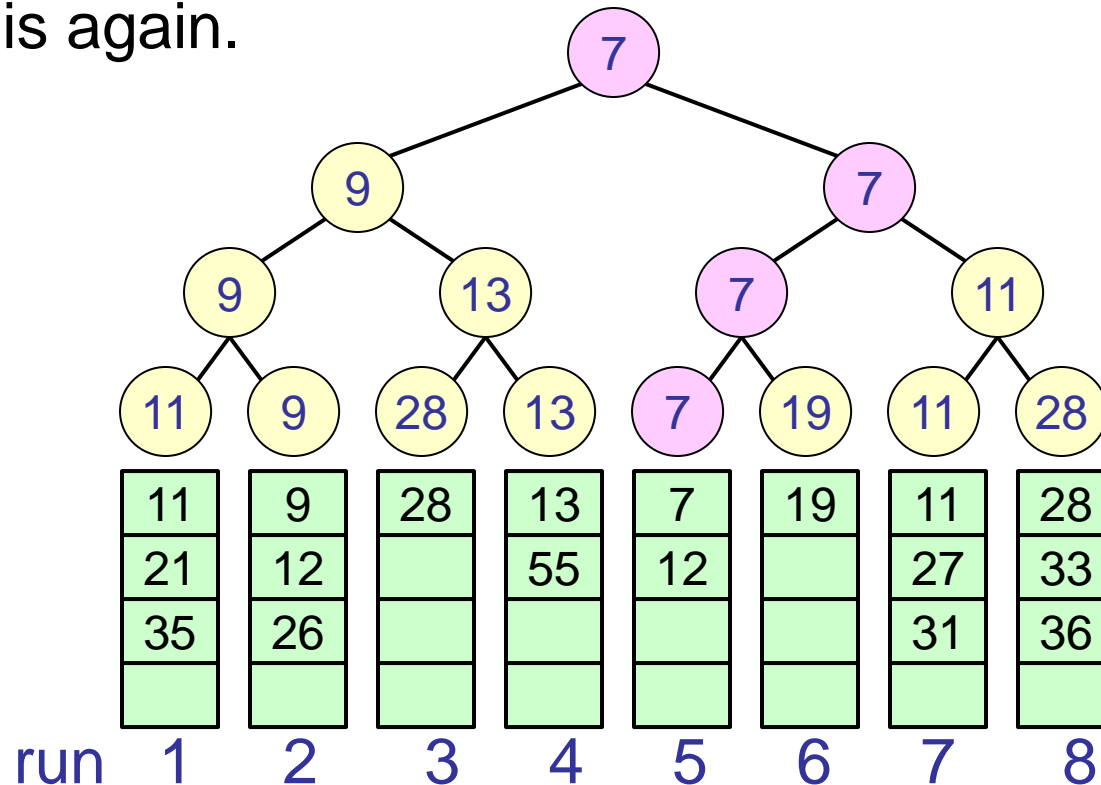
Now, what happens when we extract the overall winner (the record pointed to by the root)?

We need to determine the new winners of all the nodes along the path from the leaf node (which corresponds to the run containing the extracted record) to the root.



Winner Trees

Let's do this again.

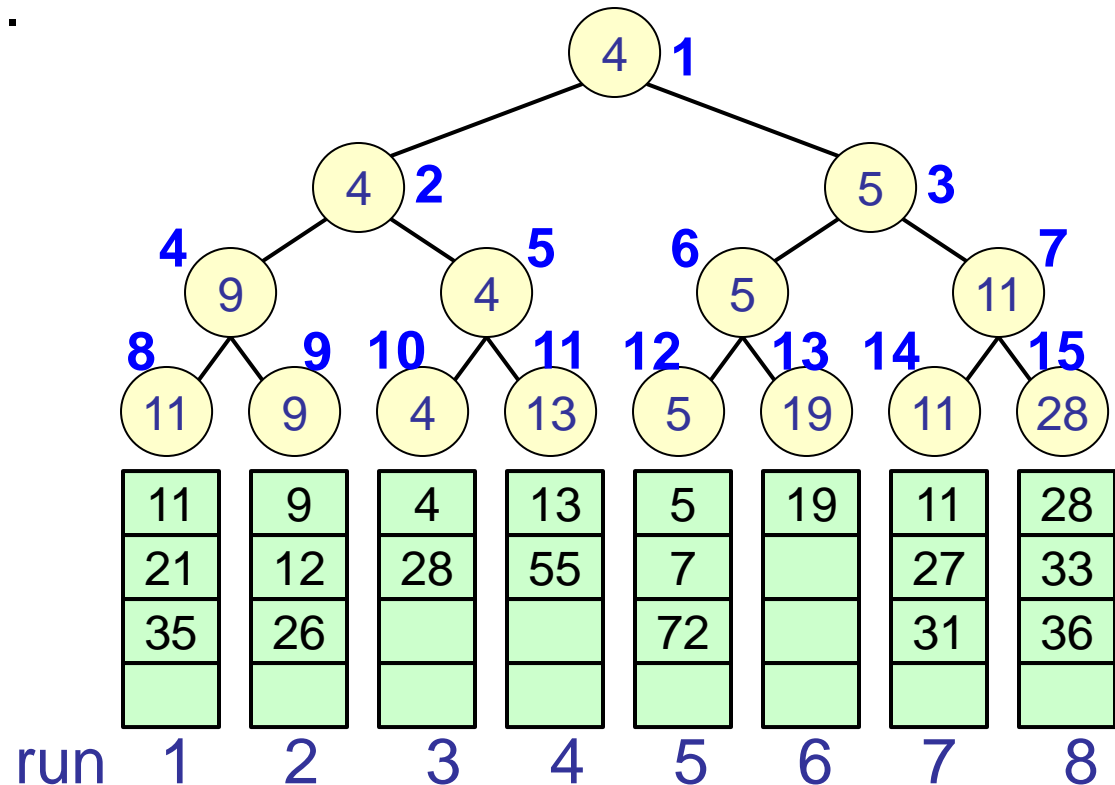


Q: What's the complexity of extracting the winner?

Q: What's the complexity of merging the runs (creating an overall sorted list), for k runs containing a total of n records?

Winner Trees

- Complexity: $O(n \log k)$.
- Winner trees are full binary trees → Use the array representation.



Q: How do we handle the case when k is not a power of 2?

Loser Trees

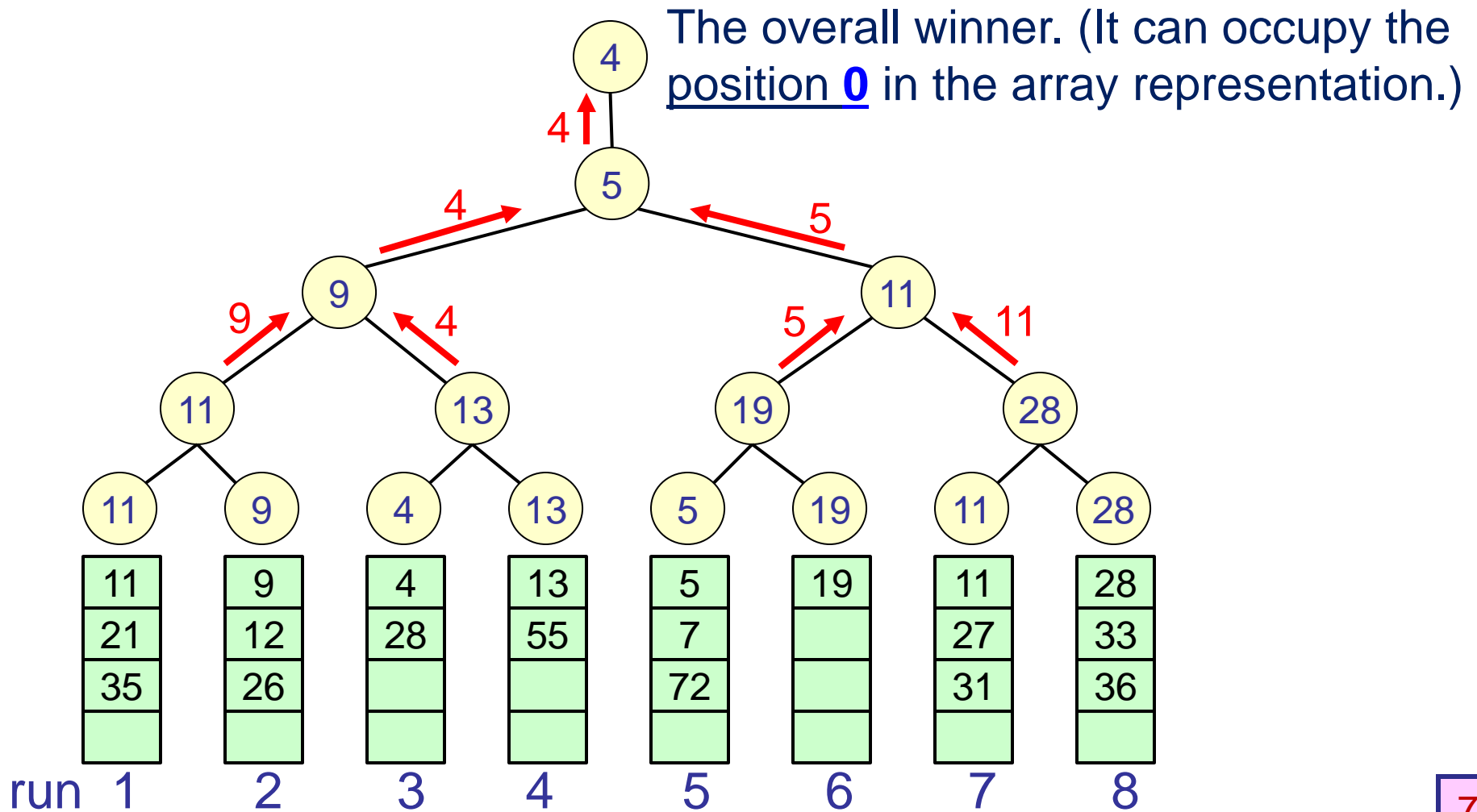
A type of selection tree where each node remembers the "loser" of its two children.

- ➔ The overall winner does not have to occupy the whole path from the root to a leaf node.
- ➔ When updating the node (after an extraction), we do not need to check the siblings of the nodes along the path. (Instead, the new key value is just compared with the losers' key values already in the path.)

Loser Trees

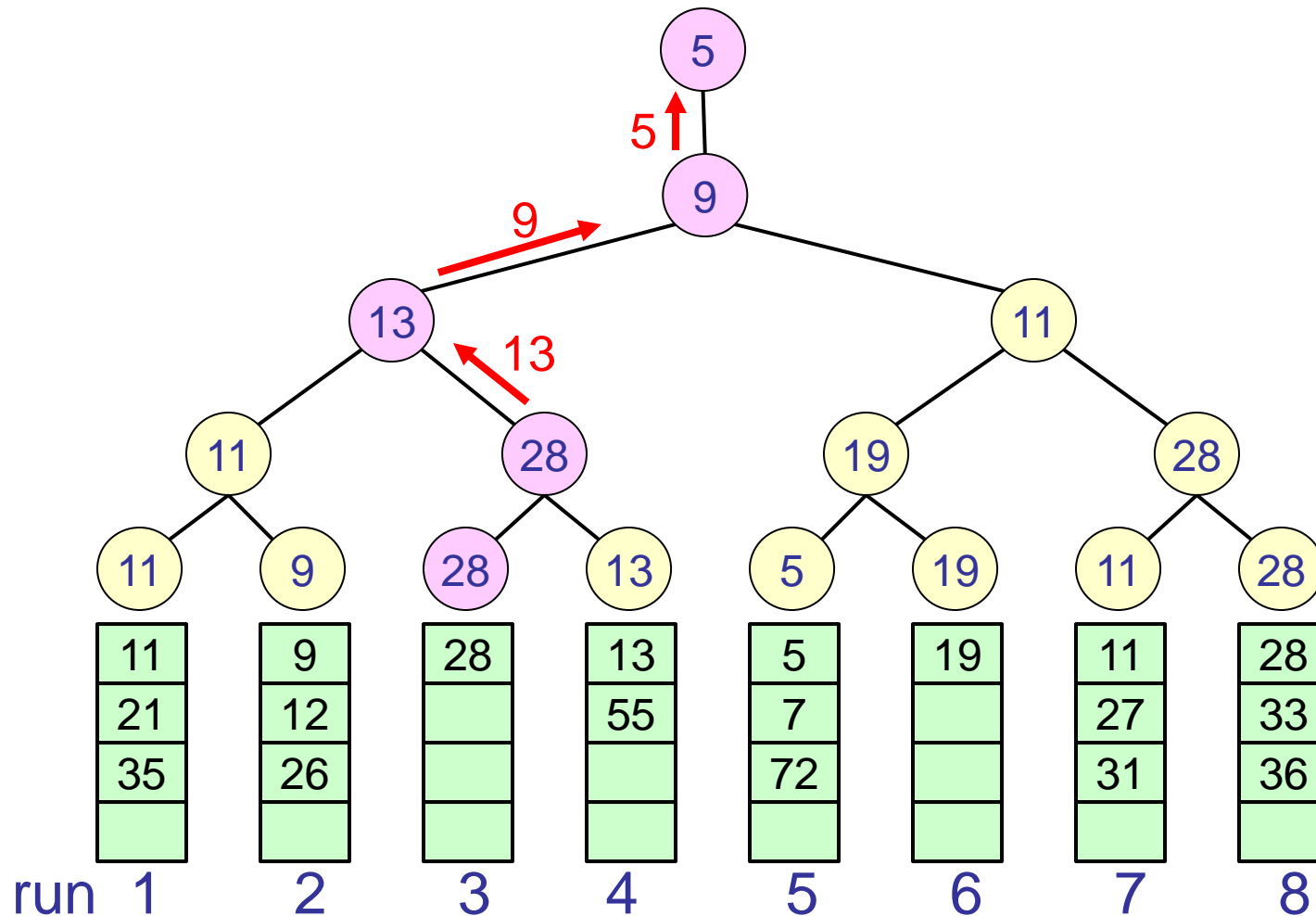
Rule for building and updating a loser tree:

At each node, the winner advances and the loser stays.



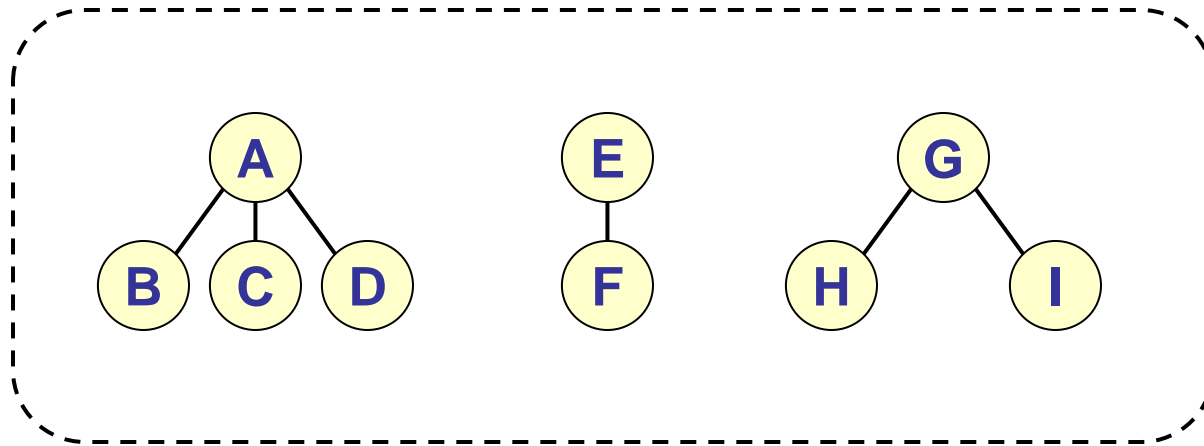
Loser Trees

After extracting the overall winner:

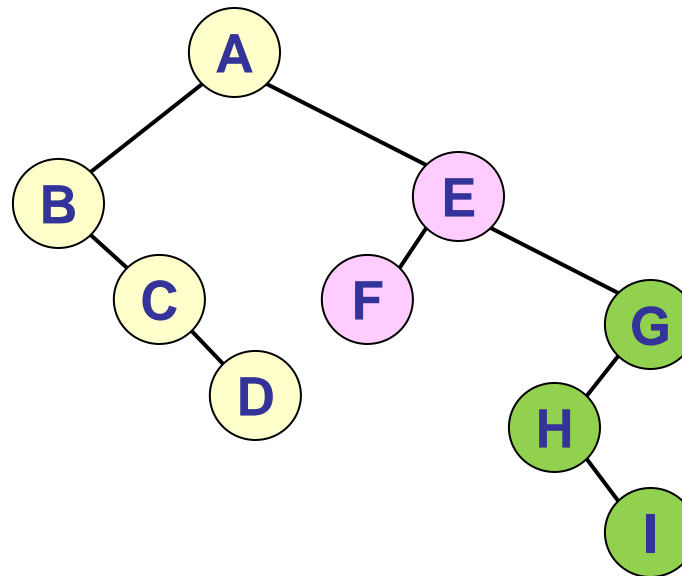
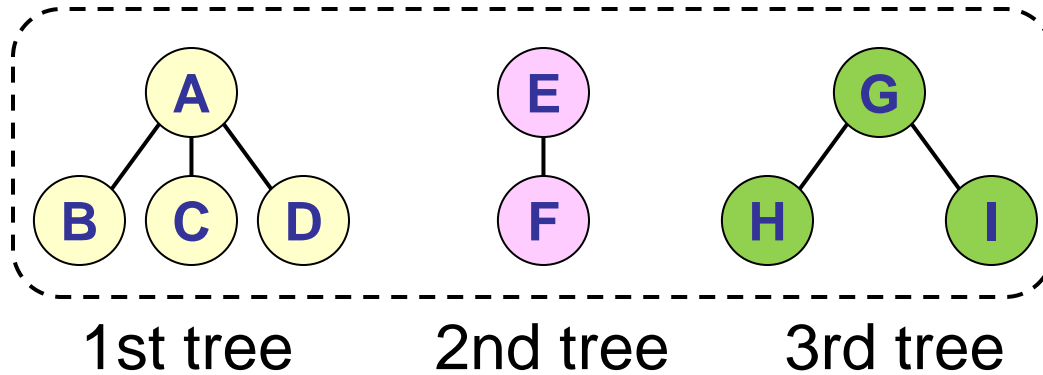


Forests

A forest is a collection of disjoint trees. An example:



Representing a Forest as a Binary Tree



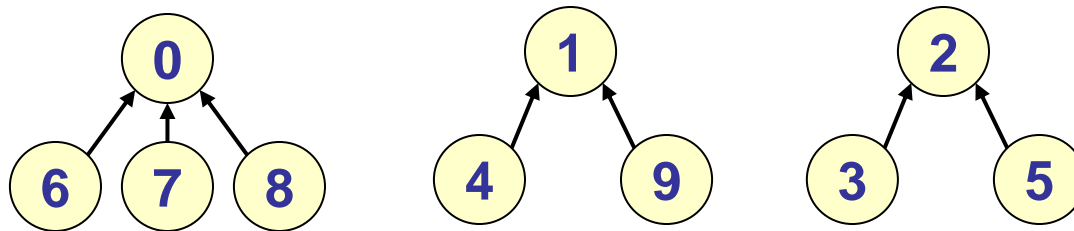
Representing a Forest as a Binary Tree

Rules (recursive) for converting a set of trees T_1, T_2, \dots, T_n into its binary tree representation $B(T_1, T_2, \dots, T_n)$:

- The root of the binary tree is $\text{root}(T_1)$.
- The left subtree:
 - Treat the subtrees of T_1 (named $T_{11}, T_{12}, \dots, T_{1m}$) as a forest.
 - The left subtree is $B(T_{11}, T_{12}, \dots, T_{1m})$.
- The right subtree
 - Treat the other trees (T_2, \dots, T_n) as a forest.
 - The right subtree is $B(T_2, \dots, T_n)$.

Representing Sets with Trees

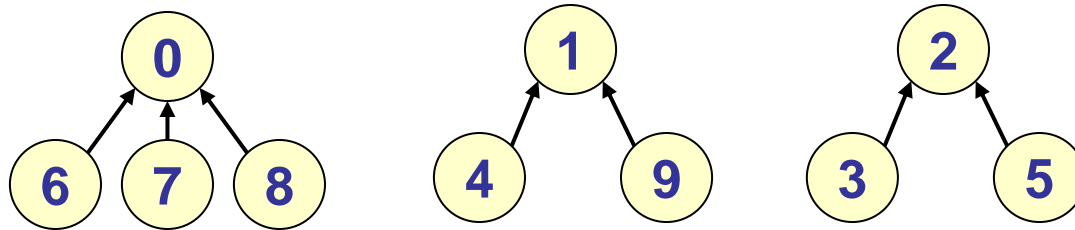
- Represent each set as a tree, with each node representing an element in the set.
- Each set is accessed through the root of its tree.
- Represent a collection of disjoint sets as a forest.
- Links between tree nodes are from child to parent. The purpose is to speed up the operation of finding the set containing a given element.
- Example: $S_1 = \{0, 6, 7, 8\}$, $S_2 = \{1, 4, 9\}$, $S_3 = \{2, 3, 5\}$



Representing Sets with Trees

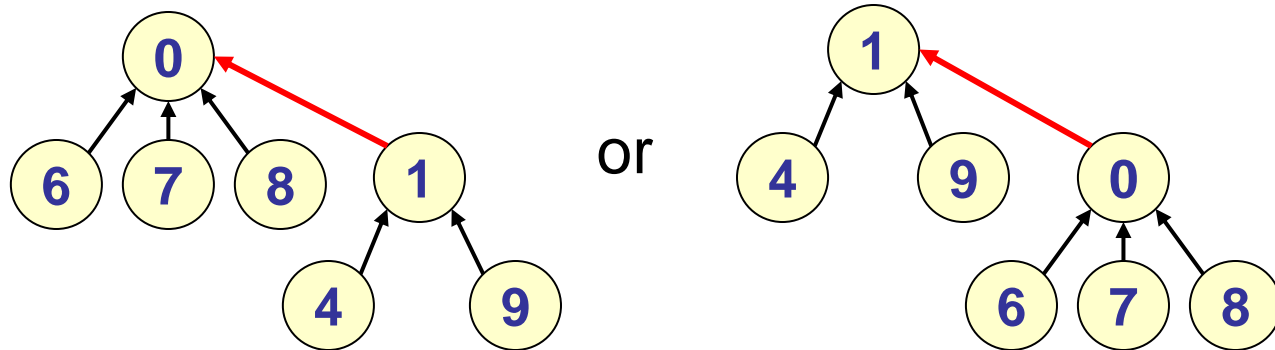
- Assume that all the elements in the sets are numbered 0, 1, 2,
- We can use an array to represent the **parent** of each node; a negative value indicates that the node is the root of its set.

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



Union of Disjoint Sets

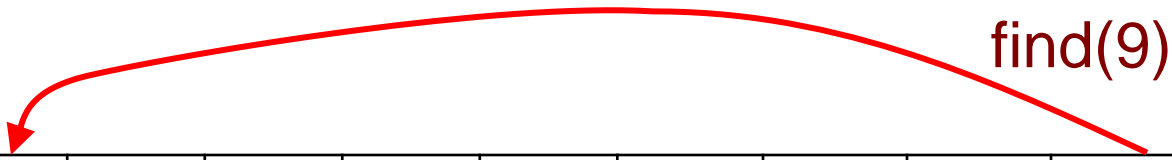
- Idea: Make the root of a tree a child of the other tree.
- Example:



Find the Set Containing a Given Element

- Goal: For a given element, return the root of the set containing that element.

```
int Sets::SimpleFind(int i)
{
    while (parent[i] >= 0)
        i = parent[i];
    return i;
}
```



find(9)

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

Problem with the Union Operation

- Let $\text{union}(i,j)$ be the union operation of the two sets with roots i and j .
- Assume that $\text{union}(i,j)$ always makes i a child of j . (The other way around is ok, too.)
- Consider this case: Start with elements $0, 1, \dots, n-1$ all being the root of its own one-element set, and apply $\text{union}(0,1)$, $\text{union}(1,2)$, \dots , $\text{union}(n-2,n-1)$, resulting in a single set:

Problem with the Union Operation

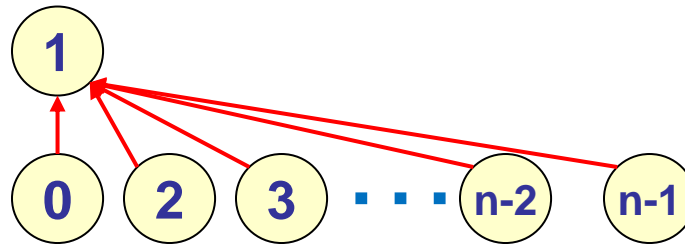


This is a **degenerate tree**
(a tree that looks like a
list).

Q: What's the complexity of find operations
for this tree?

Weighting Rule for Union Operations

- Keep track the number of elements in each set.
 - (Method in textbook): Store it (as a negative number) in the *parent* field of the root.
- **Weighting rule:** The union operation always makes the root of the "smaller" set a child of the root of the "larger" set.



Q: What's the complexity of find operations for this tree?

Property of Weighted Union

- Starting with a collection of one-element sets, a set with m elements, formed by a series of unions, has a height of no more than $\lfloor \log_2 m \rfloor + 1$.
- Worst-case example: Consider the sequence of operations:
union(0,1), union(2,3), union(4,5), union(6,7),
union(0,2), union(4,6), union(0,4)

Property of Weighted Union

0

1

2

3

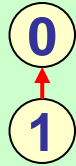
4

5

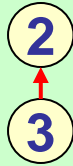
6

7

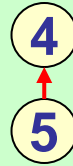
union(0,1)



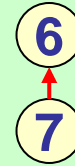
union(2,3)



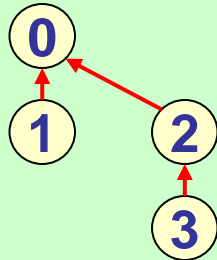
union(4,5)



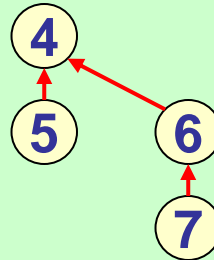
union(6,7)



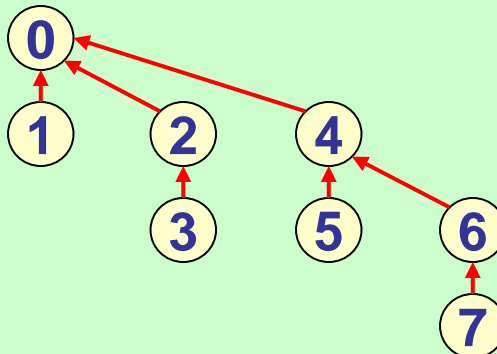
union(0,2)



union(4,6)



union(0,4)

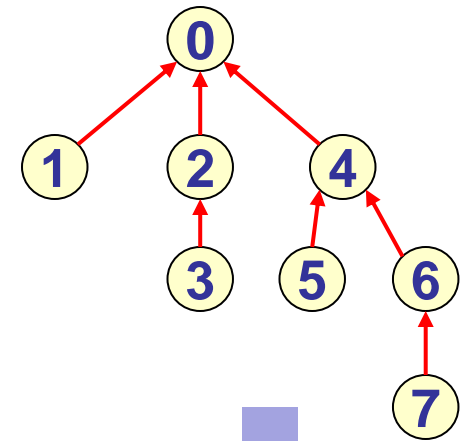


Collapsing Rule for Find Operations

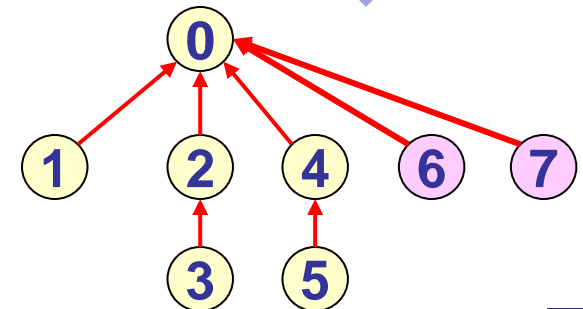
- The "find" operation is fastest when the node to be found is a child of the root.
- **Collapsing rule:** After each "find" operation, if the node is not a child of the root, make it a child of the root.
 - In addition, back-trace from that node to the root, and make every node on the path a child of the root.

Collapsing Rule for Find Operations

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



Find 7

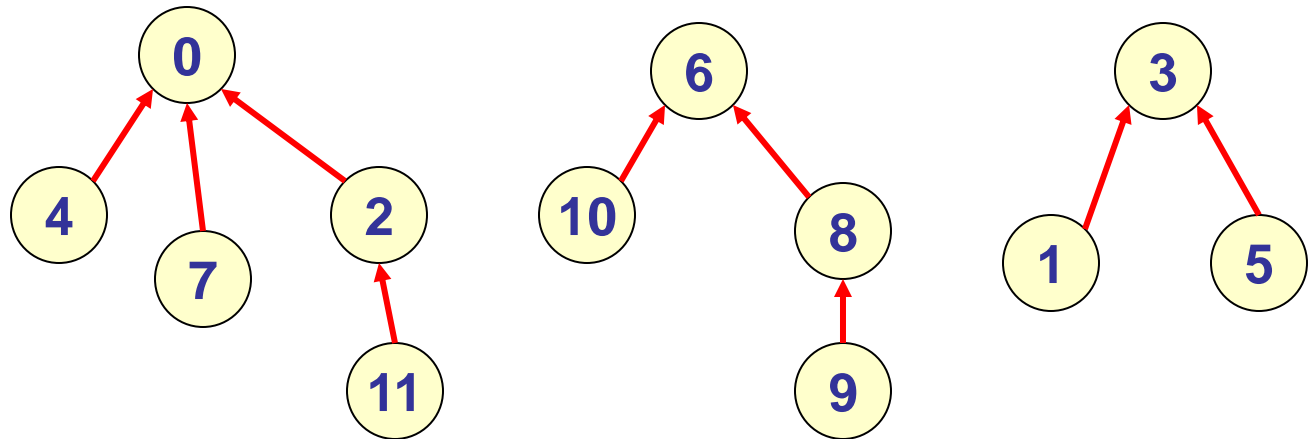


Application to Equivalent Classes

- Idea: Represent each equivalent class as a set.
- Algorithm (n items and m equivalent pairs):
 - Initialize n one-element sets.
 - For each equivalent pair $i \equiv j$:
 - ◆ **Find** the roots of i and j . (Collapsing rule applied here.)
 - ◆ If the roots are different, replace the two sets with their **union**. (Weighting rule applied here.)

Application to Equivalent Classes

- Example set: $\{0, 1, 2, \dots, 10, 11\}$
- Known equivalent pairs:
 - $0 \equiv 4, 3 \equiv 1, 6 \equiv 10, 8 \equiv 9, 7 \equiv 4, 6 \equiv 8, 3 \equiv 5, 2 \equiv 11, 11 \equiv 0$



Extra Reading Assignments

- From the textbook: Sections 5.7.5 and 5.11.1-3.