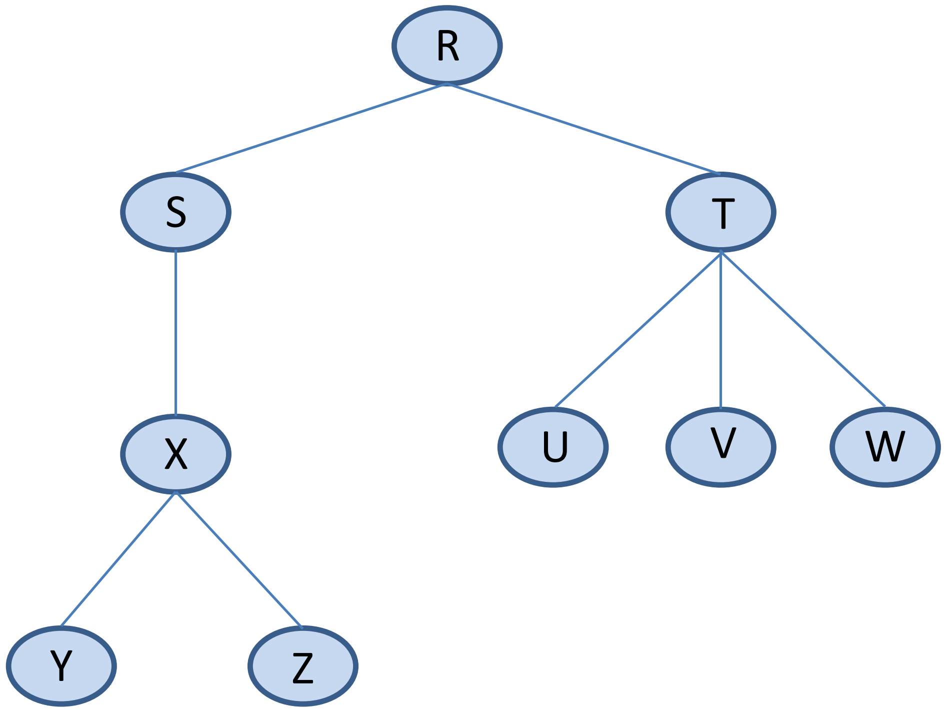
Trees

Trees

- **Trees (δένδρα)** are one of the most important data structures in Computer Science.

- Examples of trees:
  - Directory structure
  - Search trees (for stored information associated with search keys e.g., in relational databases)
  - Parse trees (in compilers)
  - Search trees (for problem solving in Artificial Intelligence)
  - Game trees (in Artificial Intelligence)
  - Decision trees (in Artificial Intelligence)
  - Heaps (for implementing priority queues)

Example



Formal Definition of Tree

- A **(rooted) tree** T is a set of nodes storing elements in a **parent-child** relationship with the following properties:
  - If T is nonempty, it has a special node, called the **root** of T, that has no parent.
  - Each node v of T different from the root has a unique **parent** node w; every node with parent w is a **child** of w.
- The **empty tree** is one which has no nodes.

Terminology

- An **edge** of tree T is a pair of nodes (u, v) such that u is a parent of v, or vice versa.
- A sequence of nodes that are connected by edges is called a **path**. The **length** of a path is the number of its edges.
- If we travel downwards along the edges that start at a node e.g., R, we arrive at R’s two **children** S and T.
- Two nodes that are children of the same parent are called **siblings**.
- The **descendants** of a node consist of the nodes that can be reached by travelling downwards along any path starting at the node.
- If we travel upwards from node e.g., S, we find the node R which is the **parent** of S.
- The **ancestors** of a node consist of the nodes that can be reached by travelling upwards along paths towards the root.
- If a node has no children, it is called a **leaf** or **external node**.
- If a node has children, then it is called an **internal node**.
- The root is an internal or external node depending on whether it has children.

Terminology (cont'd)

- The nodes of a tree can be arranged in **levels**.
- The root is at **level 0**. The children of the root are at **level 1**, their children are at **level 2** and so on.
- We often say **depth** instead of **level**.
- In a tree, there is **exactly one path** from the root *R* to each descendant of *R*.
- The **length** of the path from the root to a node is equal to the **level** or **depth** of the node.
- The largest depth of any node in a tree is called the **height** of the tree.
- We can use **spatial terminology** to refer to parts of a tree. For example, left child or right child or middle child.

Terminology (cont'd)

- The **subtree** of T **rooted** at a node v is the tree consisting of all descendants of v including v itself.
- A tree is **ordered** if there is a linear ordering defined for the children of each node; that is, we can identify children of a node as being the first, the second and so on.
- Ordered trees are usually drawn with siblings arranged from left to right, corresponding to their linear relationship.

Proposition

- Let T be a tree with n nodes, and let c_p denote the number of children of a node p of T. Then $\sum_p c_p = n - 1$.
- Proof?

Proof

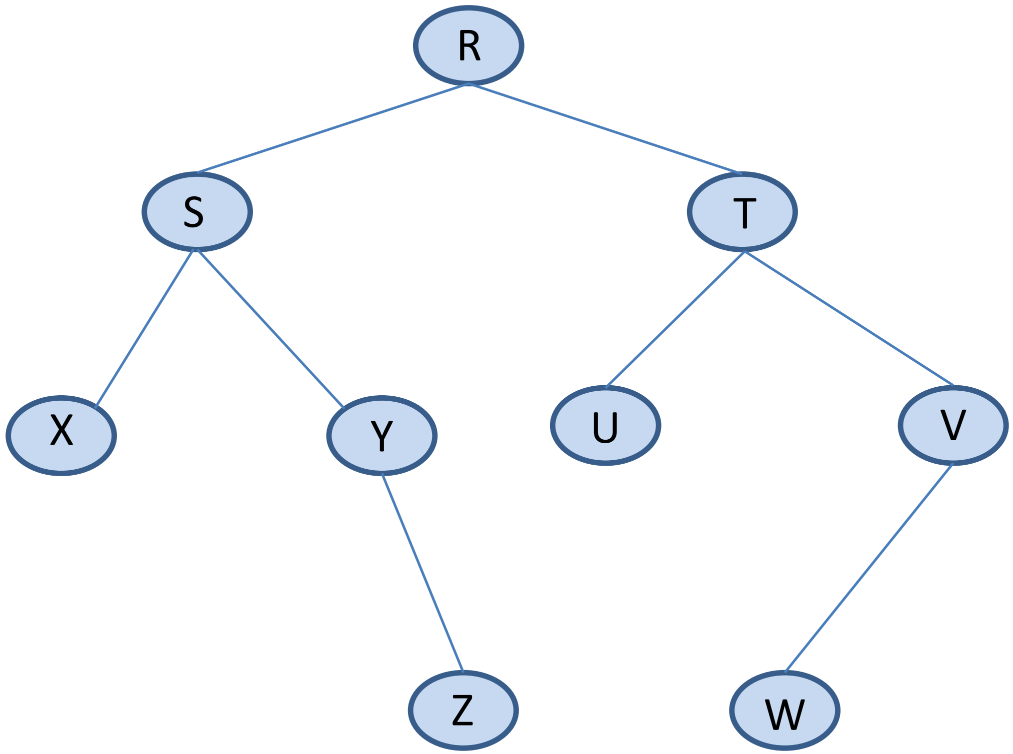
- Each node of T with the exception of the root is a child of another node, and thus contributes one unit to the above sum.

Binary Trees

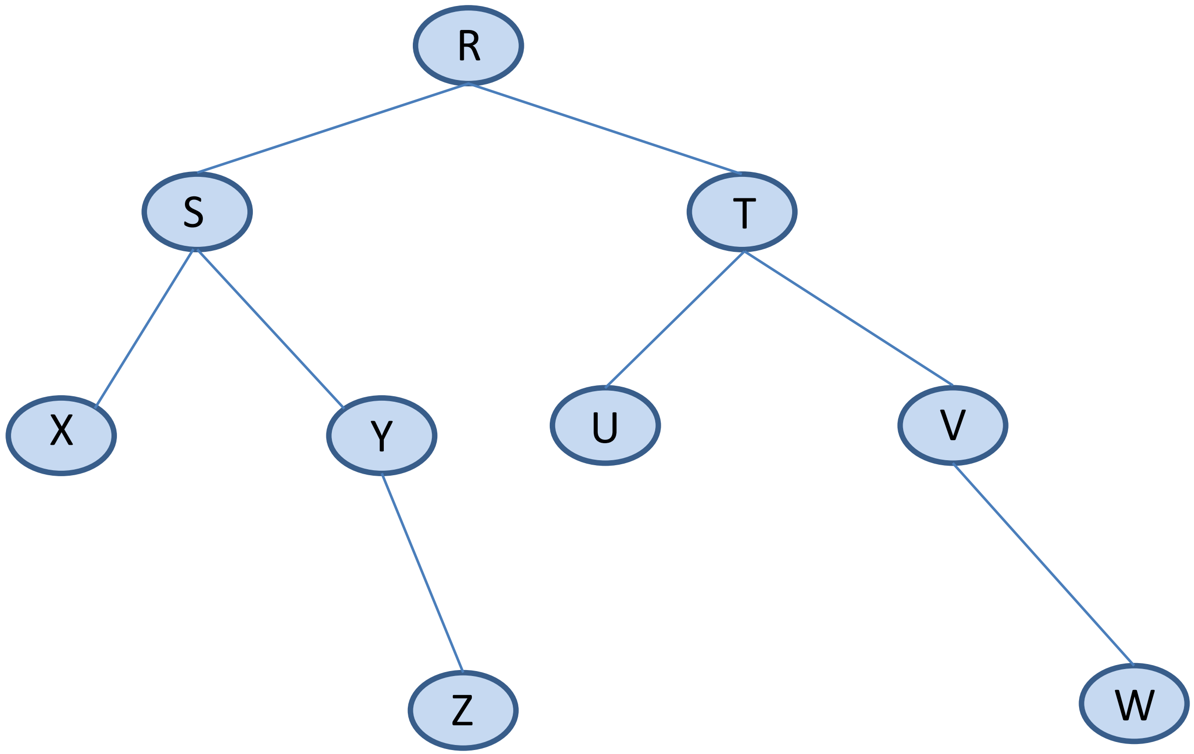
- A **binary tree (δυαδικό δένδρο)** is a tree in which:
  - Each node has at most two children and stores an item.
  - Each child node is labelled as being either a **left child** or a **right child**.
  - A left child precedes a right child in the ordering of children of a node.

- **Recursive definition**: A **binary tree** is either the empty tree or consists of:
  - A node r, called the **root** of T and storing an item.
  - A binary tree, called the **left subtree** of T .
  - A binary tree, called the **right subtree** of T .

Example: a Binary Tree



Example: a Different Binary Tree



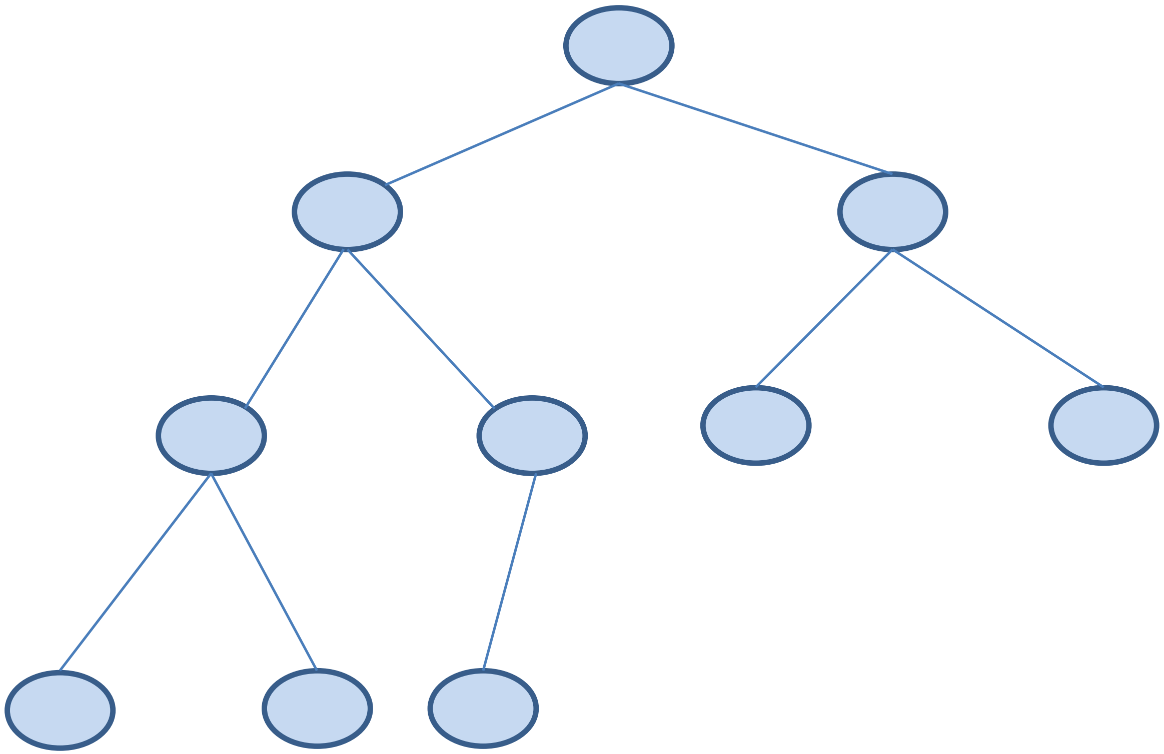
Whether a child is left or right matters. Now node W is the right child of node V.

Complete Binary Trees

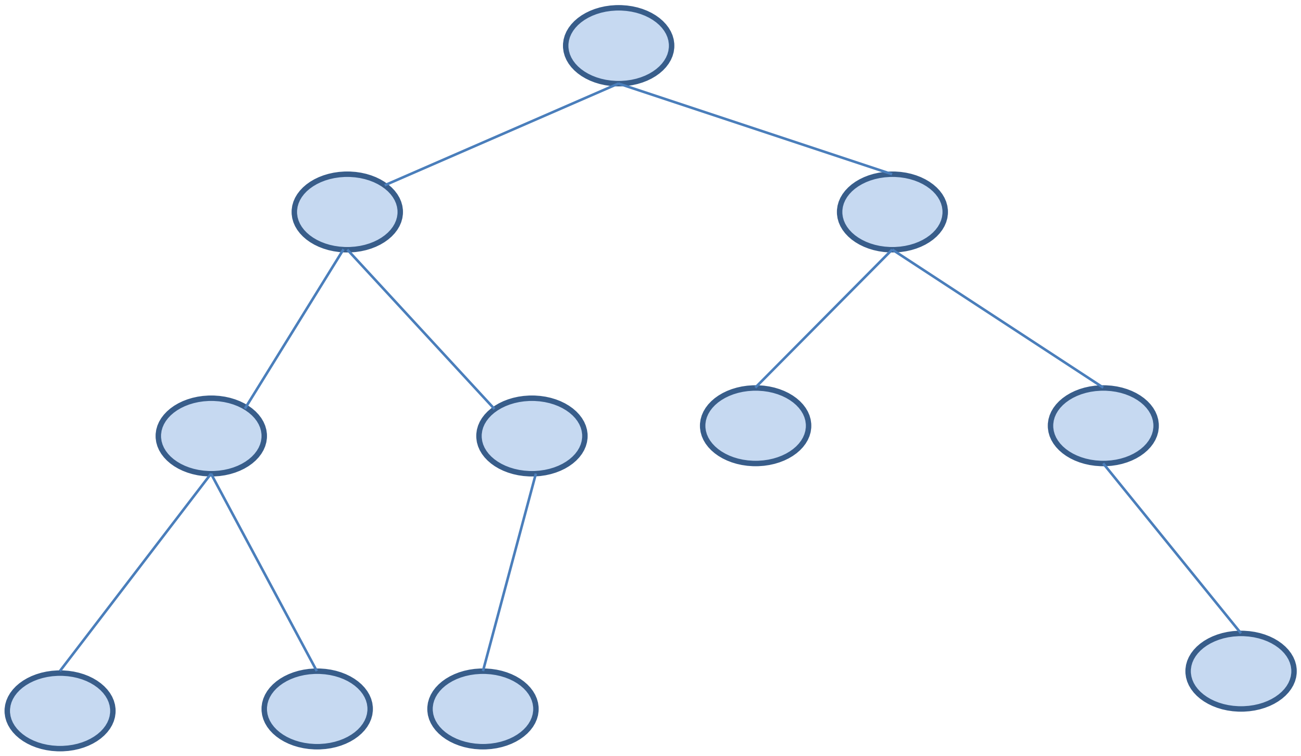
- A binary tree with height h is a **complete binary tree (πλήρες δυαδικό δένδρο)** if levels $0,1,2,\ldots,h-1$ have the maximum number of nodes possible (namely, level i has 2^i nodes, for $0 \leq i \leq h-1$) and the nodes at level h fill this level from left to right.

- In other words, in a complete binary tree, leaves are on either a single level or on two adjacent levels such that the leaves on the bottommost level are placed as far left as possible. Additionally, all levels except possibly the bottommost one are completely filled with nodes.

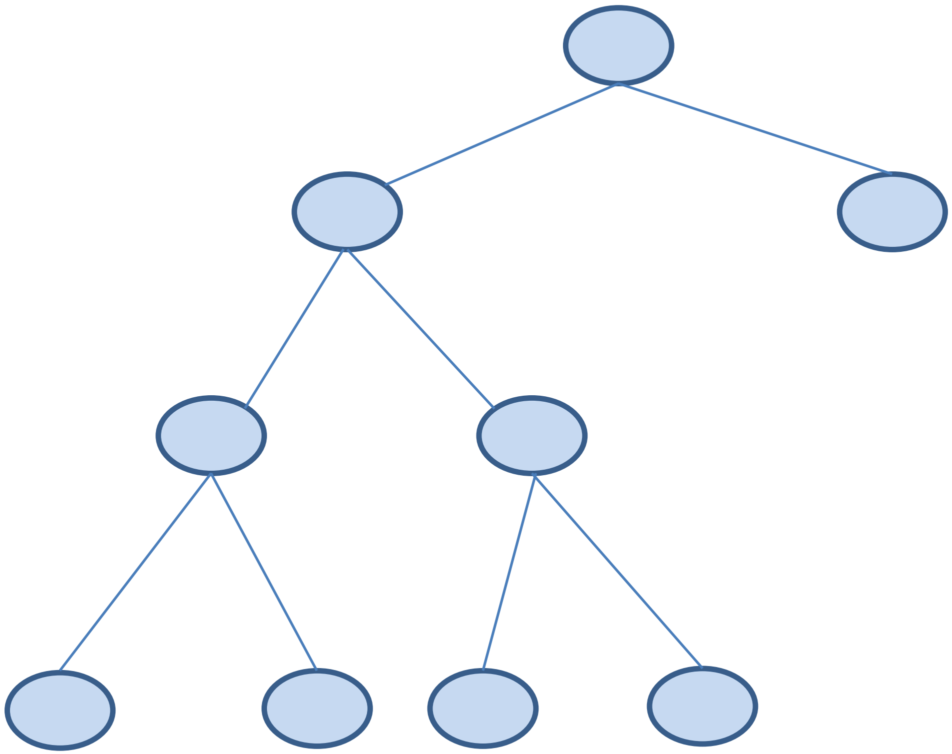
Example: complete binary tree



Example: not complete binary tree



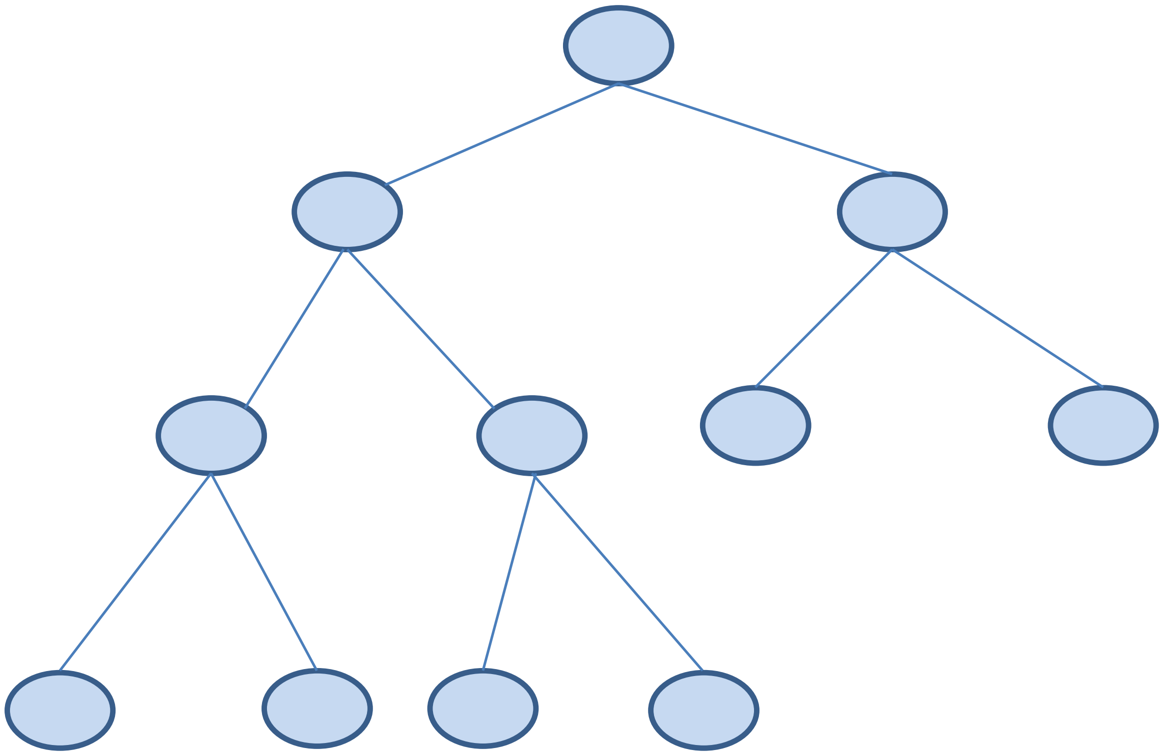
Example: not complete binary tree



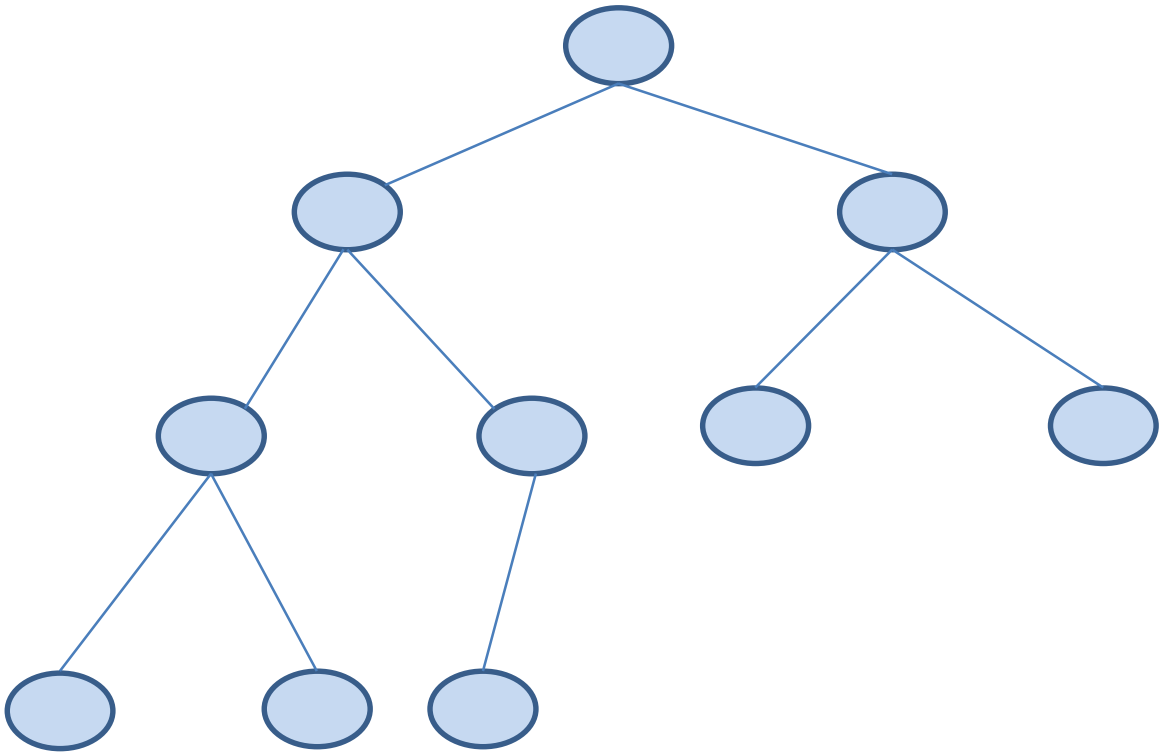
Proper Binary Trees

- A binary tree is called **proper** or **full (γνήσιο)** if each node has either zero or two children.
- A binary tree that is not proper is called **improper**.
- In a proper binary tree, each internal node has exactly two children.

Example: proper binary tree



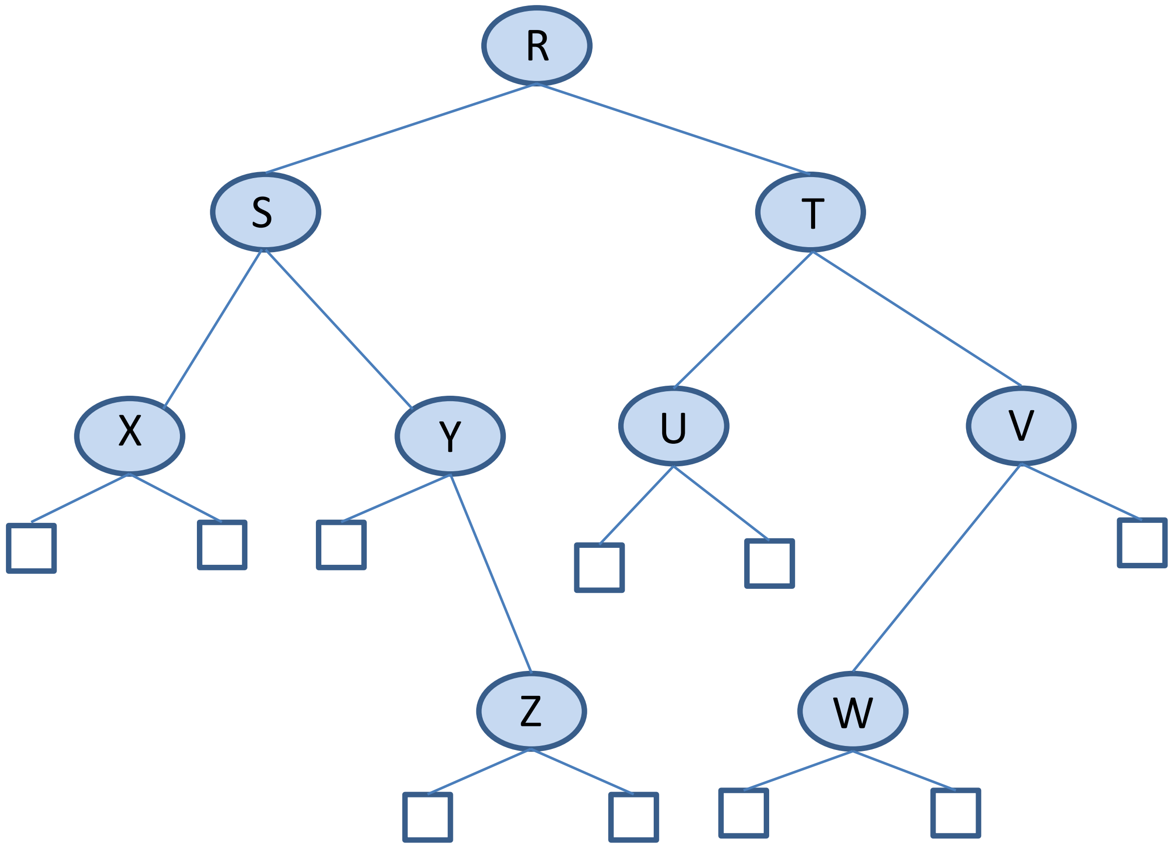
Example: improper binary tree



Extended Binary Trees

- Often, we view a binary tree as a **non-empty proper binary tree**. In this case, we draw each internal node of a binary tree as having exactly two children. Each external node is special: it has no children and it is represented by a **square symbol.**
- In the literature, the term **extended** (επεκταμένο) binary tree is also used for this case.
- This view of binary trees simplifies the programming of search and update functions (in a linked representation of the tree external nodes are represented by null links), and also the statement of relevant theoretical results.
- This view of binary trees also clarifies cases where a node has one child regarding whether it is a left or a right one.

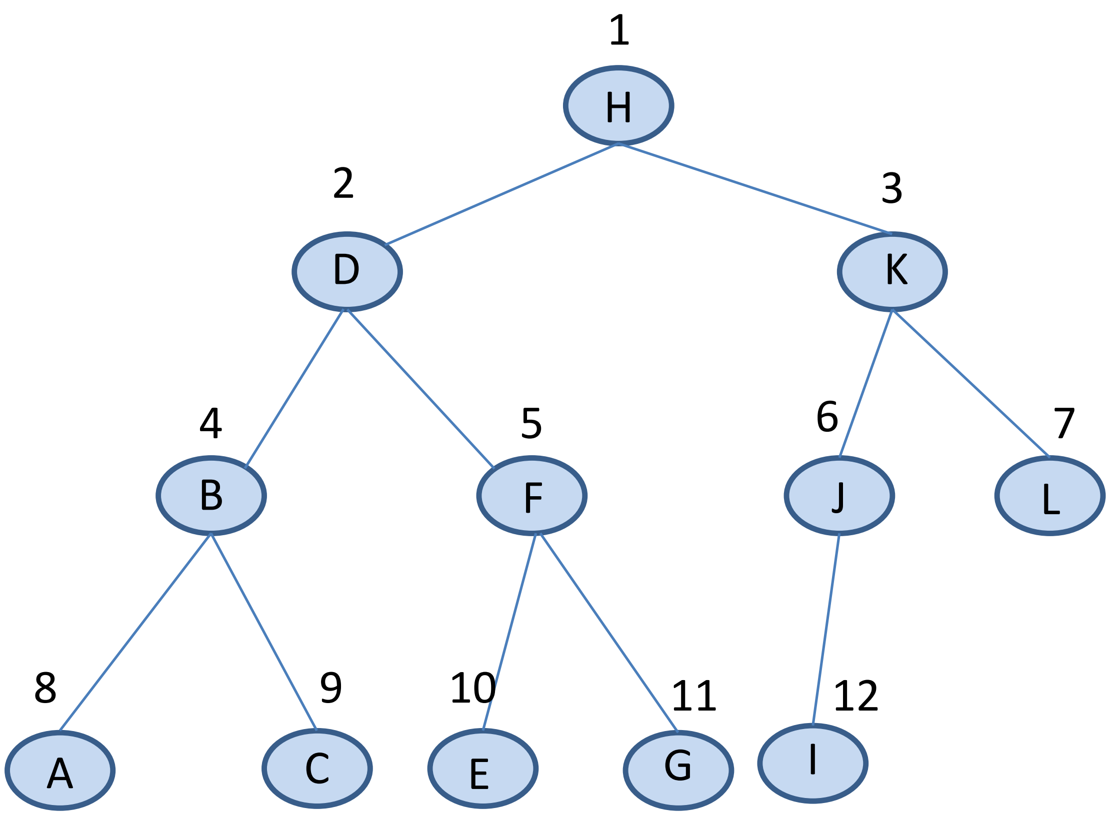
Example



Level Order of Nodes in a Tree

- If we number the nodes of a tree level-by-level and, in each level, going from left to right, we have the **level order (διάταξη επιπέδου)** of the nodes of the tree.

Example



Nodes of a Complete Binary Tree

- **Question:** How many nodes does a complete binary tree have at each level?

Nodes of a Complete Binary Tree

- **Answer**: At most
  - $2^0 = 1$ at level 0.
  - $2^1 = 2$ at level 1.
  - $2^2 = 4$ at level 2.
  - …
  - 2^k at level k.

Properties of Binary Trees

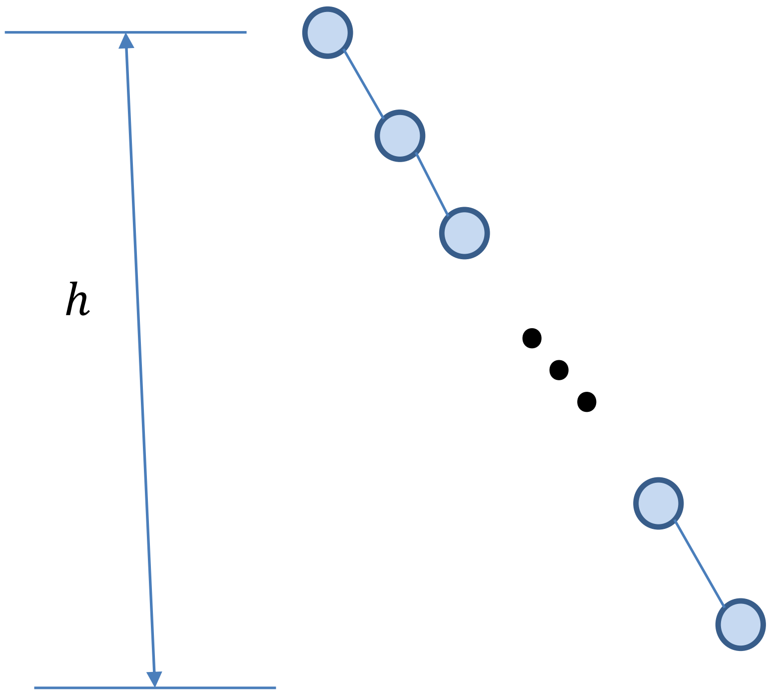
- Let T be a non-empty binary tree, and let n, n_I, n_E and h denote the number of nodes, number of internal nodes, number of external nodes, and height of the tree, respectively. Then T has the following properties:
  1. $h + 1 \leq n \leq 2^{h+1} - 1$
  2. $1 \leq n_E \leq 2^h$
  3. $h \leq n_I \leq 2^h - 1$
  4. $\log(n + 1) - 1 \leq h \leq n - 1$

Proof

- Let us prove (2) first. The lower bound is easy to see since the simplest non-empty binary tree has a single node, the root. The upper bound is reached when we have each node at each level of the tree having exactly two children.

Proof (cont'd)

- Let us now prove (3). The case that gives us the lower bound is a tree like the following where the internal nodes are h in number:



Proof (cont'd)

- The tree that gives us the upper bound is when we have each node at each level of the tree having exactly two children. In this case, the number of internal nodes is:

$$1 + 2 + 2^2 + \cdots + 2^{h-1} = \sum_{i=0}^{h-1} 2^i = \frac{2^h - 1}{2 - 1}$$

$$= 2^h - 1$$

Proof (cont'd)

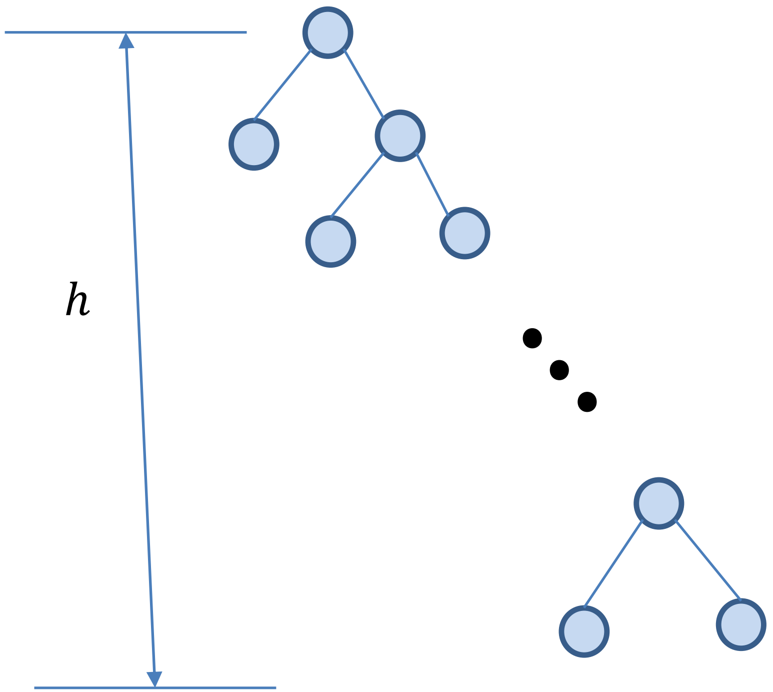
- To prove (1) simply add up the inequalities of (2) and (3).
- To prove (4), rewrite (1) and then take logarithms of each term.

Properties of Binary Trees (cont'd)

- Also, if T is proper, then it has the following properties:
  1. $2h + 1 \le n \le 2^{h+1} - 1$
  2. $h + 1 \le n_E \le 2^h$
  3. $h \le n_I \le 2^h - 1$
  4. $\log(n + 1) - 1 \le h \le \frac{n-1}{2}$
  5. $n_E = n_I + 1$

Proof

- The lower bounds of (2) and (3) can be seen from the following tree which has h internal nodes and $h + 1$ external nodes:



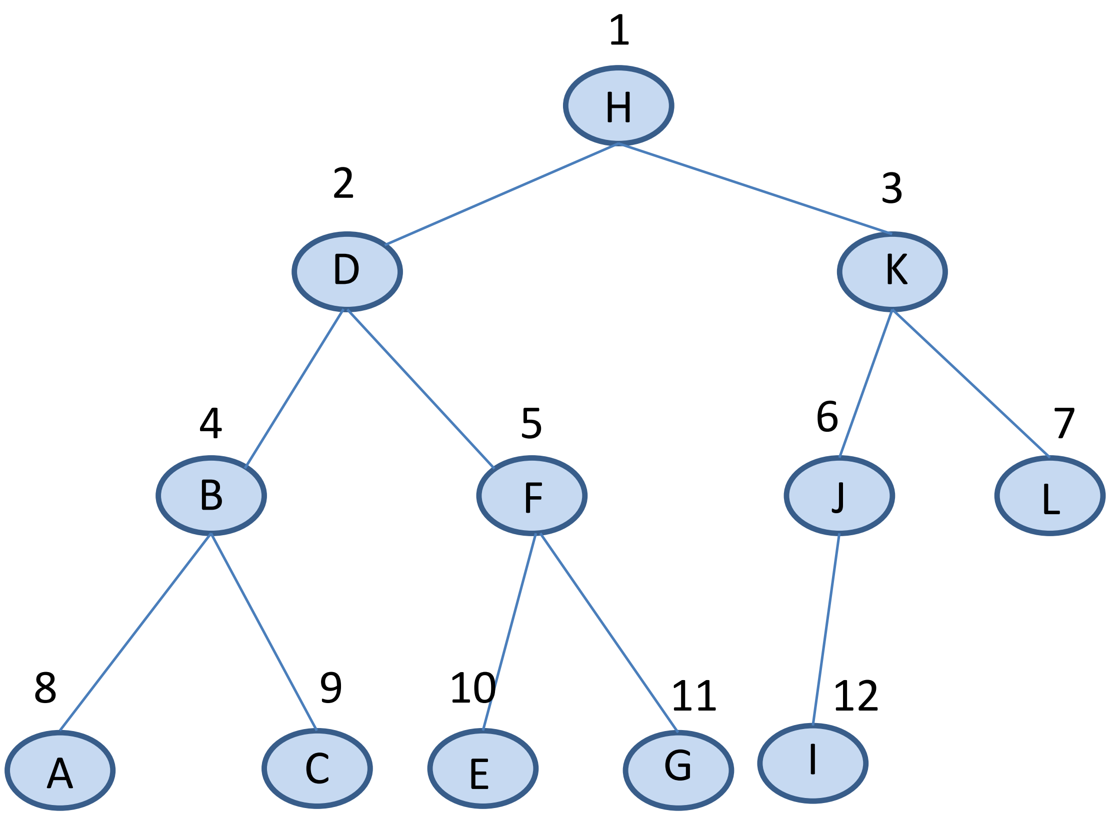
Proof (cont’d)

- The upper bounds for (2) and (3) can be seen from the previous proposition since the trees used in the proofs there are proper.
- (1) and (4) can then be proved as in the previous proposition.

Proof (cont’d)

- We can prove (5) using **induction**. For the base case, consider a tree consisting of a single root node. In this case we have 1 external node and 0 internal nodes so the relationship holds.
- If, on the other hand, we have a tree with two or more nodes, then the root has two subtrees. Since these subtrees are smaller than the original tree, we may assume they satisfy the relationship. Thus each subtree has one more external node than internal nodes. Between the two of these subtrees, there are two more external nodes than internal nodes. But the root is an internal node. So in total we have one more external node than internal nodes.

How Do We Represent a Binary Tree?



A Sequential Binary Tree Representation

- If a complete binary tree has *n* nodes then its **contiguous sequential representation** is an array *A[0:n]* as follows (for the previous example *n=12*). Note that the array stores the information on the tree nodes using **level order**.

A:		H	D	K	B	F	J	L	A	C	E	G	I
		1	2	3	4	5	6	7	8	9	10	11	12

How to Find Nodes

To Find:	**Use:**	**Provided:**
The left child of $A[i]$	$A[2i]$	$2i \leq n$
The right child of $A[i]$	$A[2i+1]$	$2i+1 \ \leq n$
The parent of $A[i]$	$A[i/2]$	$i > 1$
The root	$A[1]$	*A* is nonempty
Whether $A[i]$ is a leaf	*True*	$2i > n$

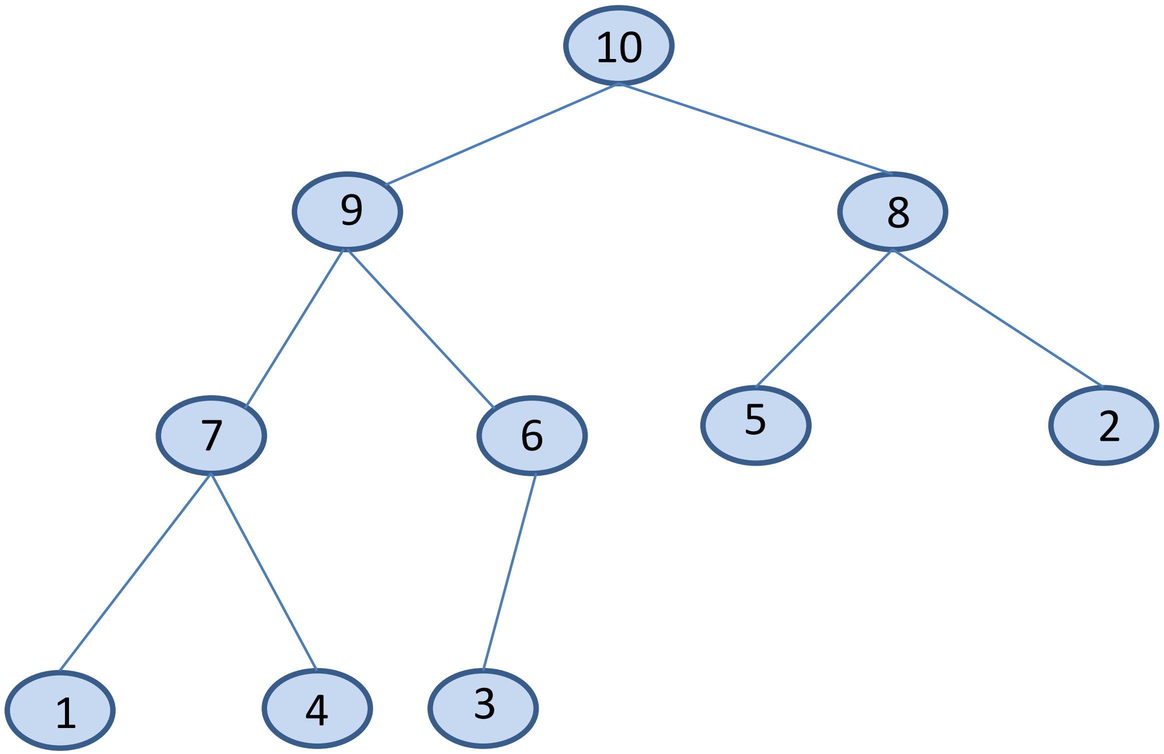
Sequential Representation (cont'd)

- The sequential representation can also be used in the case that a binary tree is **not complete.**
- In this case there will be **empty cells** in the respective array so such a representation can be wasteful.

Heaps

- A **heap** (σωρός) is a complete binary tree with values stored in its nodes such that no child has a value bigger than the value of its parent (i.e., the value of the parent of each node is greater than or equal the value of the node itself).
- Some authors call this a **max-heap** (σωρός μεγίστων).
- We can also define a **min-heap** (σωρός ελαχίστων) when the relationship between the value of a parent and the value of its child is "less than or equal".

Example



Heaps and Priority Queues

- A heap provides a representation for a priority queue.
- **Reminder:** A **priority queue** is an ADT having the property that items are removed in the order of highest-to-lowest priority regardless of the order in which they were inserted.
- If a heap is used to represent a priority queue, it is easy to find **the item of highest priority**, since it sits at the root of the tree.
- If we remove the value at the root, we have to **restore the heap property**.

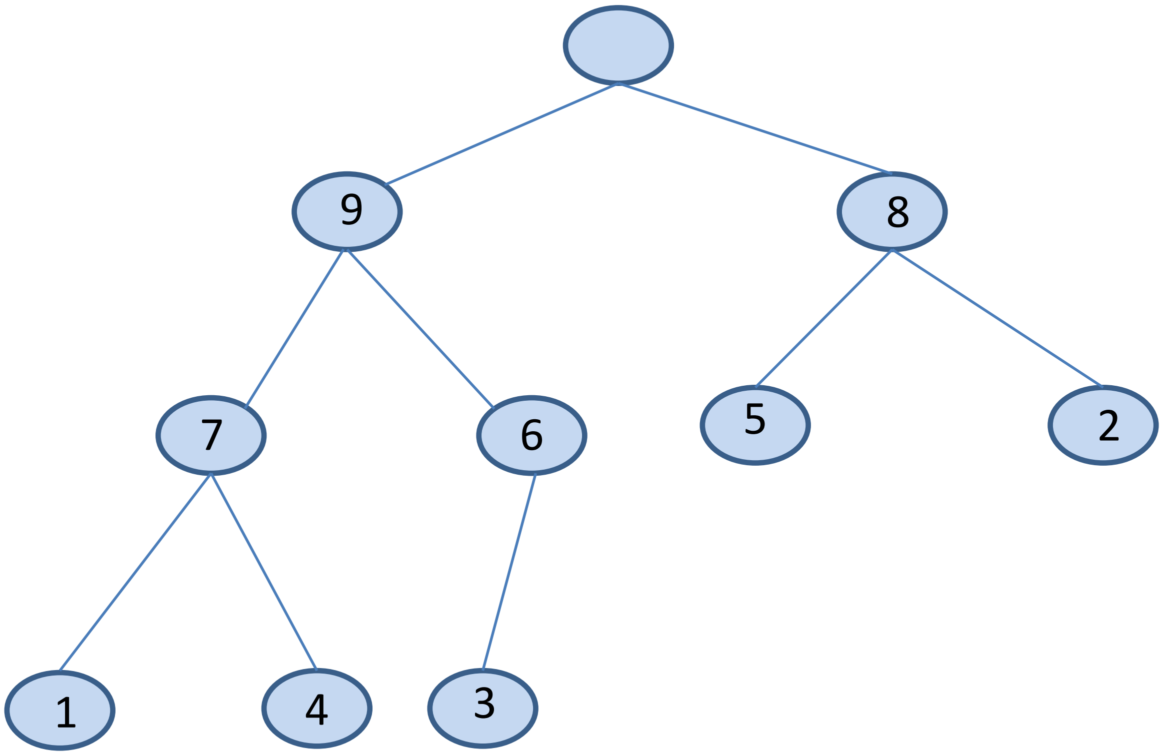
heapify-down (auxiliary function)

- Input: a node **n**
- Preconditions:
  - the subtrees of both children of **n** are heaps
- Postcondition:
  - the subtree of **n** is a heap
- Algorithm:
  - If **n** is >= its children (or is a leaf), stop
  - Otherwise **exchange n** with its **largest child**
  - Continue the process with the new position of **n**

remove-max

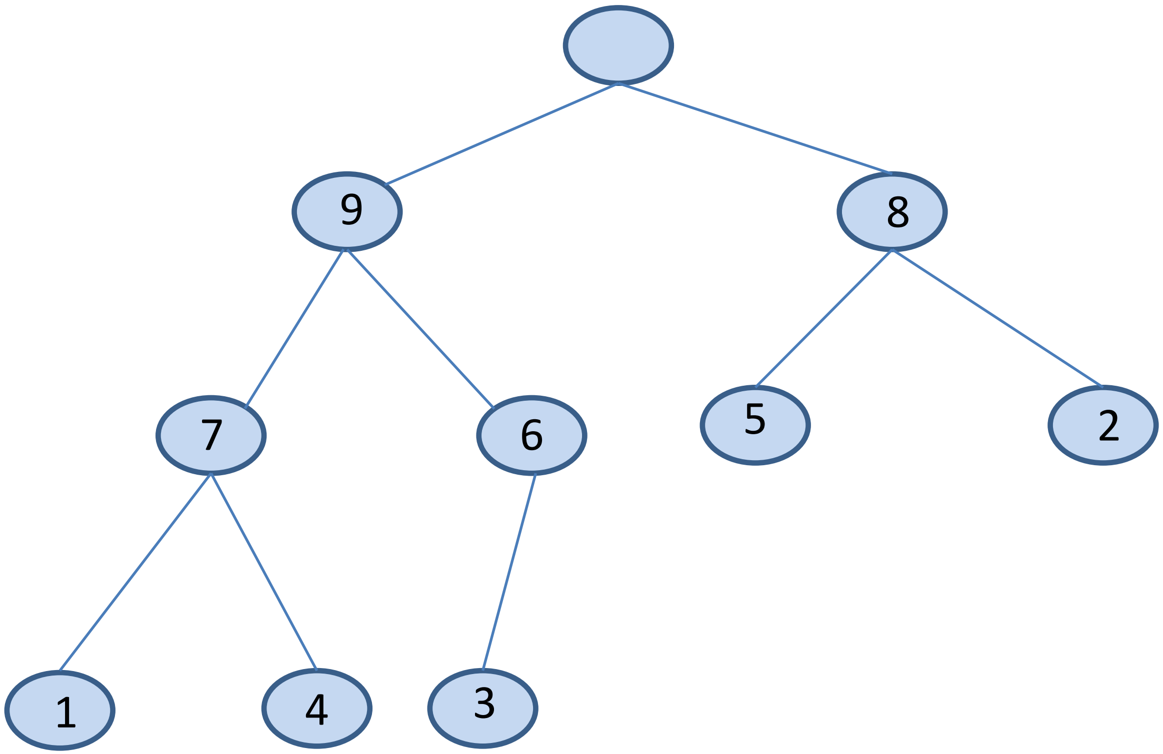
- Removes the max element (the root)
- Restores the heap property
- Algorithm:
  - Remember the **root** element
  - Remove the **last node**, set it at the root
    - The tree is **not a heap** anymore
    - **BUT:** the subtrees of root's children are heaps! so...
  - Call **heapify-down(root)**
  - Return the old root

remove-max, example



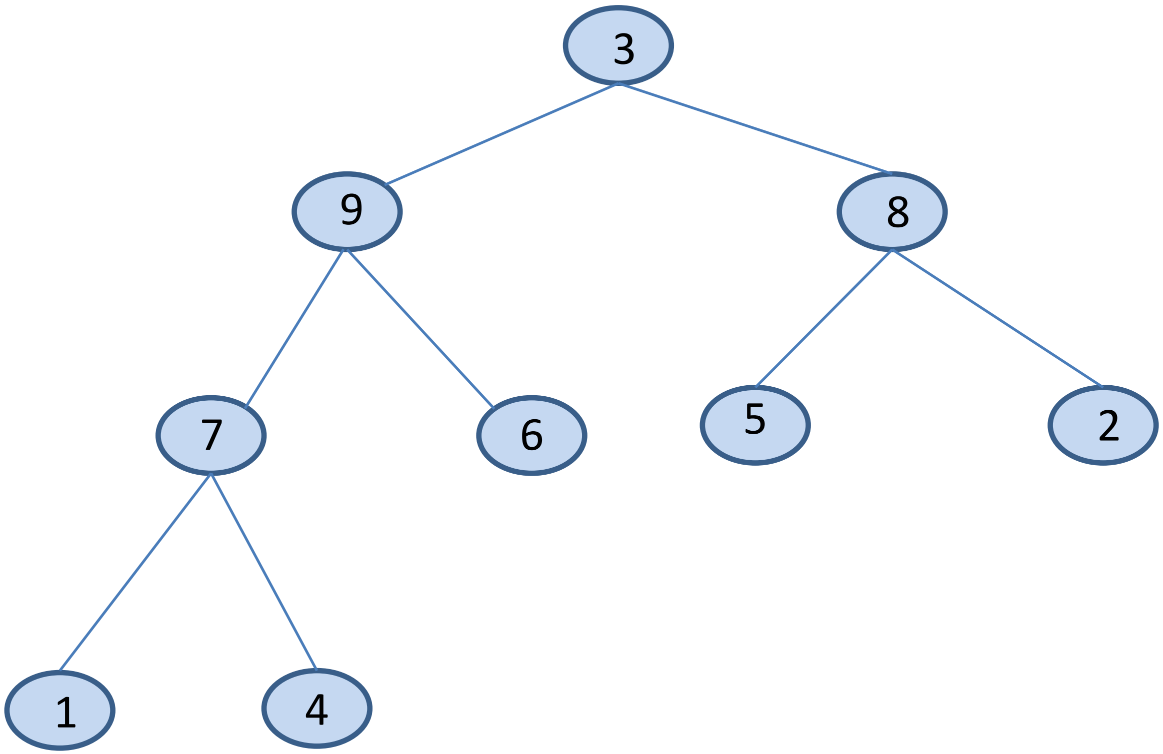
The root has been removed

remove-max, example



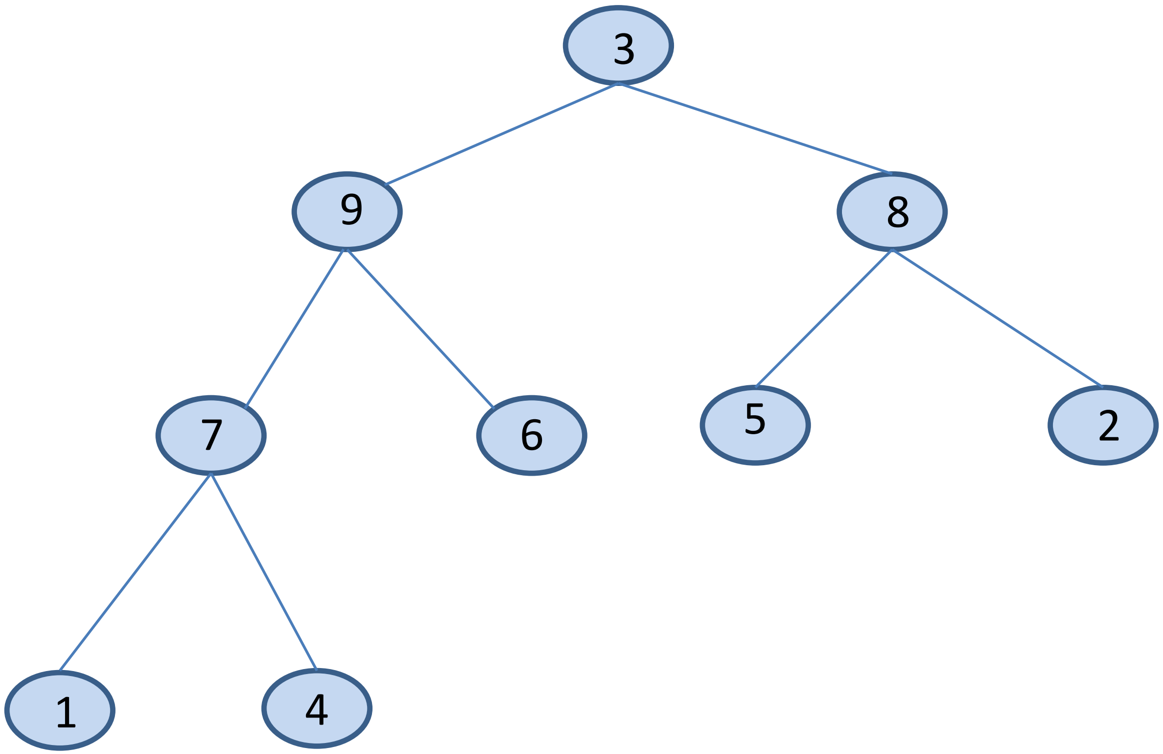
Delete the node with value 3 and insert this value into the root.

remove-max, example



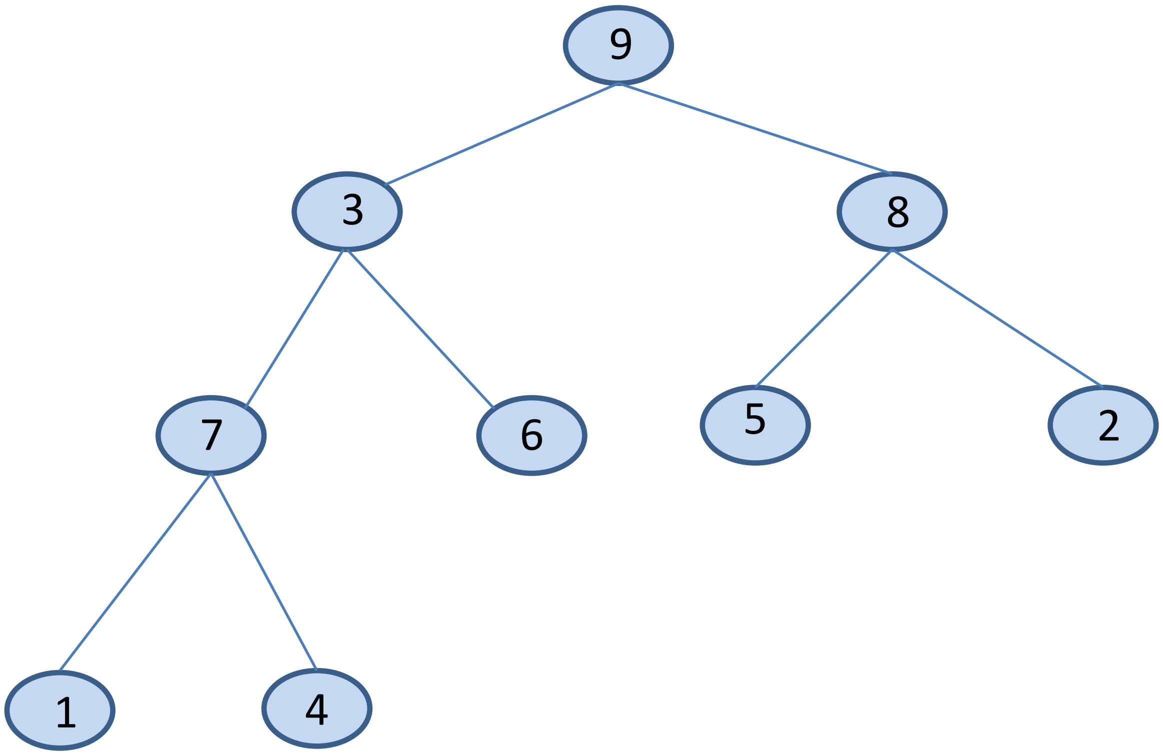
This not a heap. Call heapify-down(root)

heapify-down, example



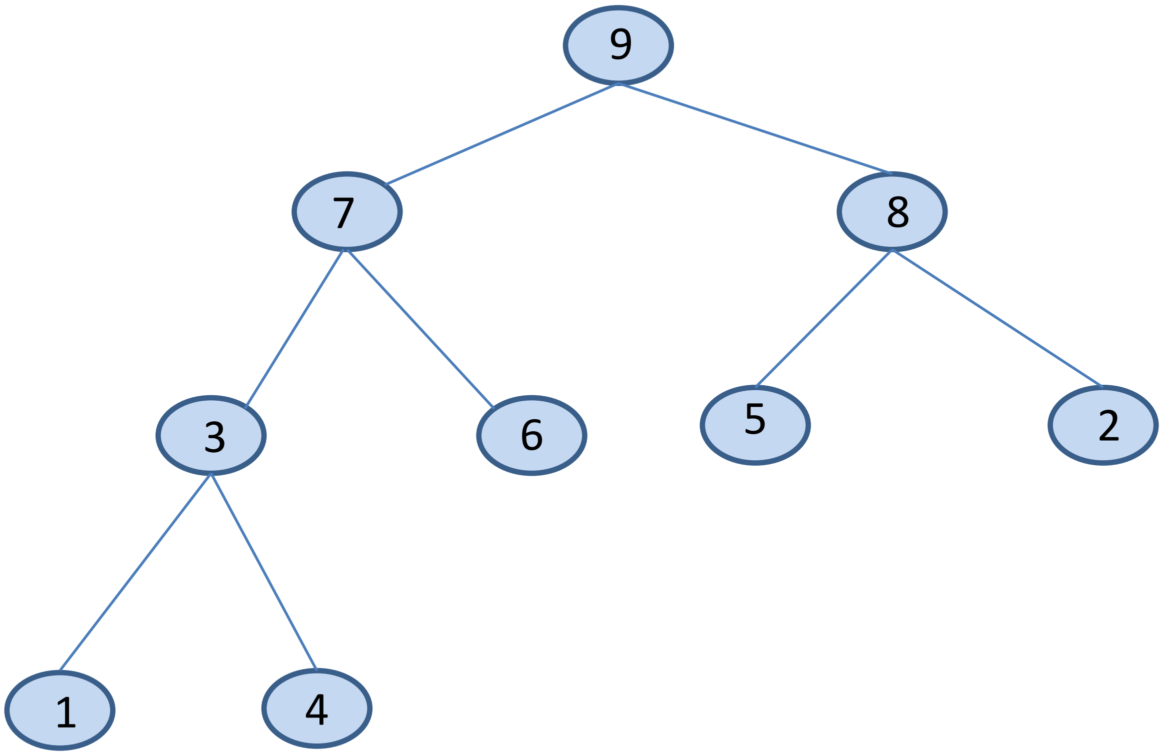
We are at the root, interchange 3 with 9.

heapify-down, example



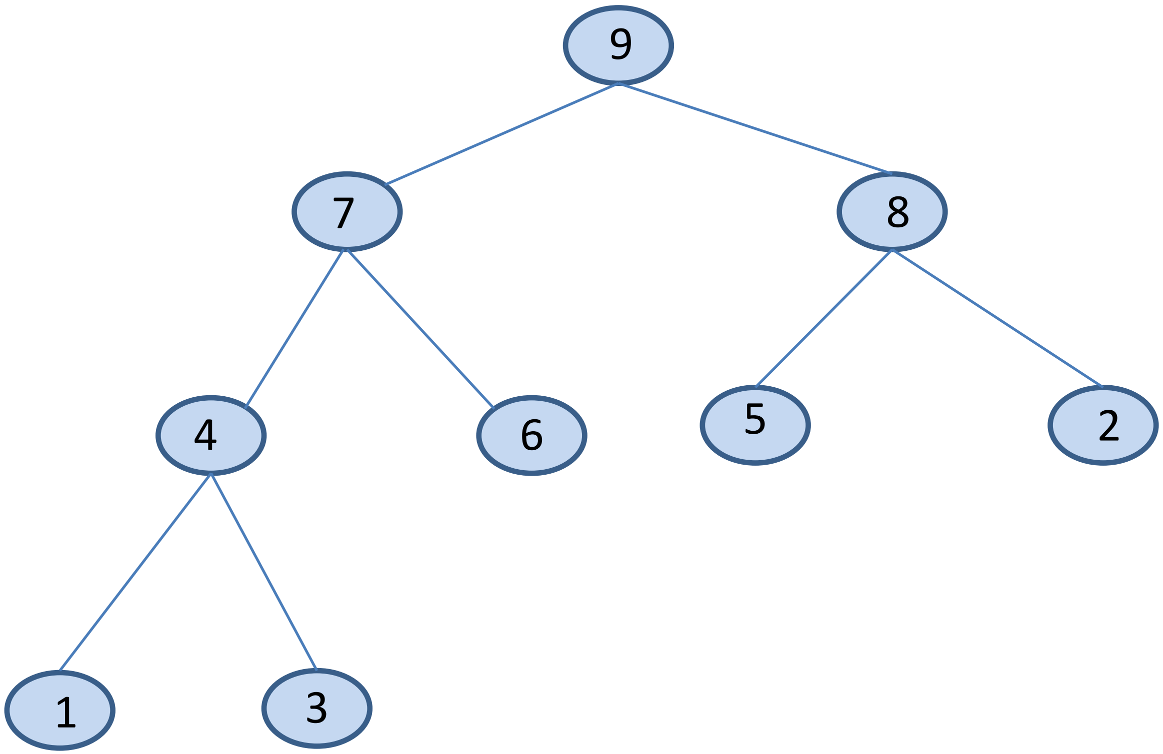
Continue with the new position of 3, Interchange 3 with 7.

heapify-down, example



Continue with the new position of 3, interchange 3 with 4.

heapify-down, example



The above tree has the heap property restored.

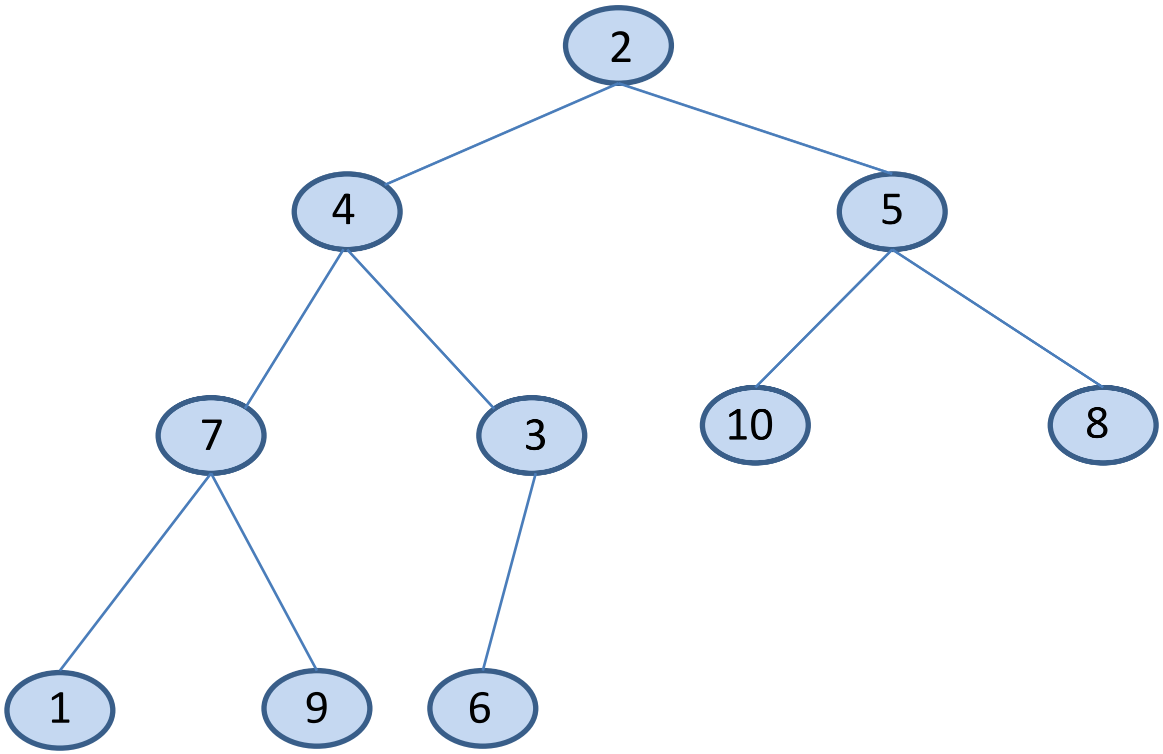
Heapifying a Complete Binary Tree

- Input: arbitrary binary tree
- Postcondition: the tree is a heap

- Idea: for each internal node n
  - first restore the heap property of its **children**
  - apply **heapify-down(n)**

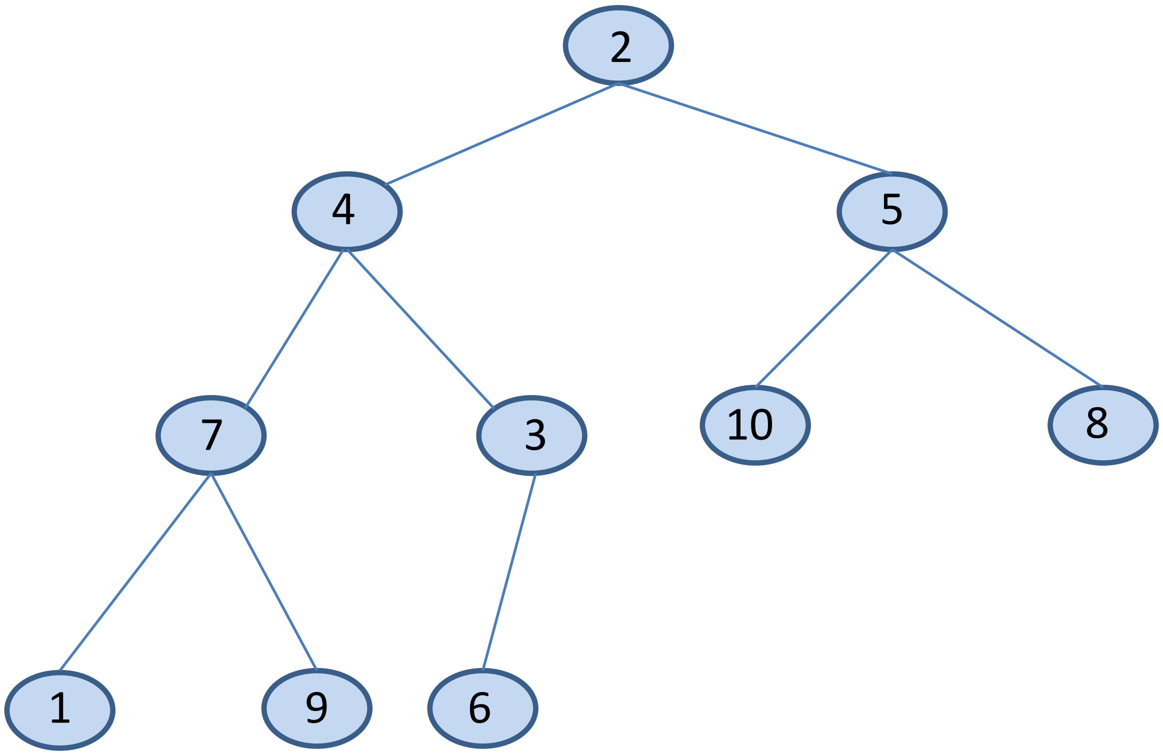
- Simple iterative algorithm:
  - For each internal node **n** in **reverse level order**
    - apply **heapify-down(n)**

Example



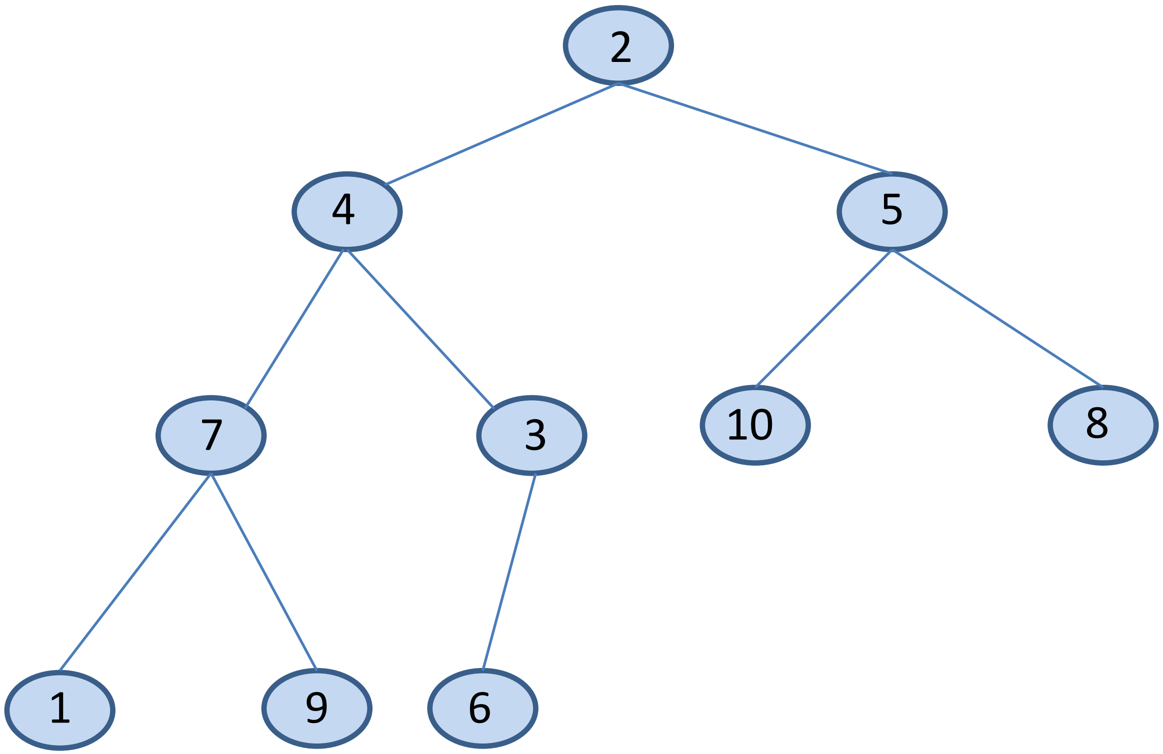
Let us heapify the above tree.

Heapifying the Tree



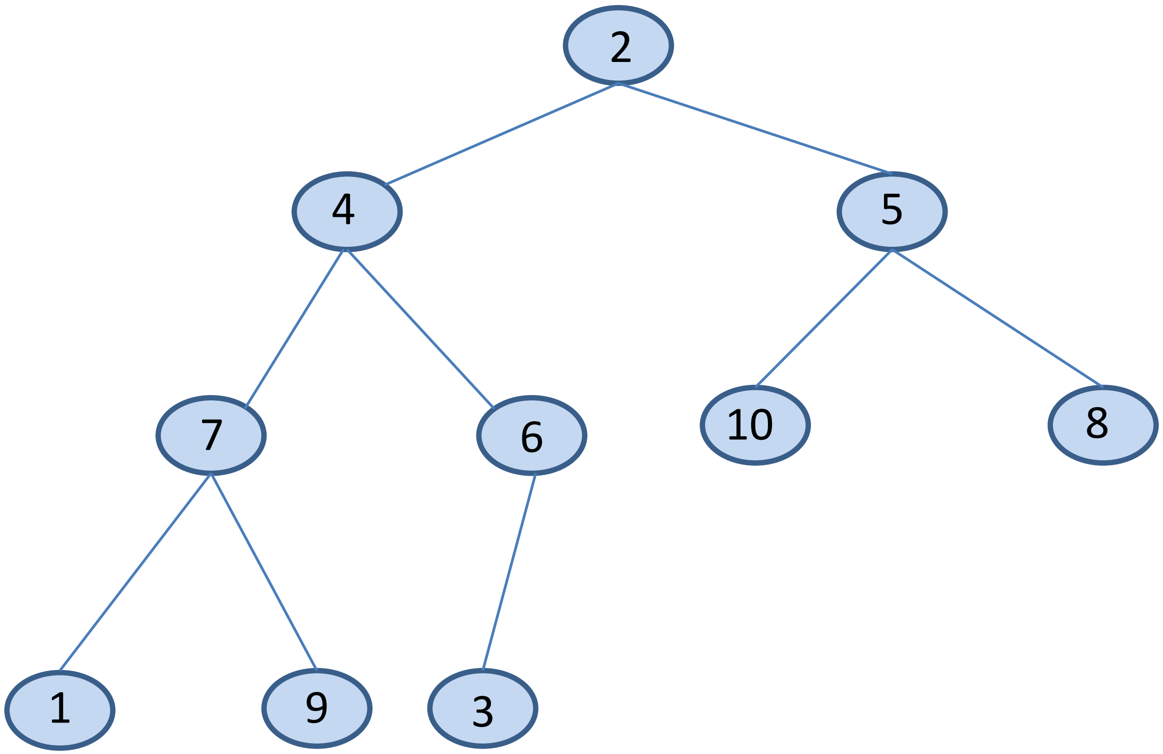
The internal nodes in reverse level order are 3, 7, 5, 4 and 2.

Heapifying the Tree



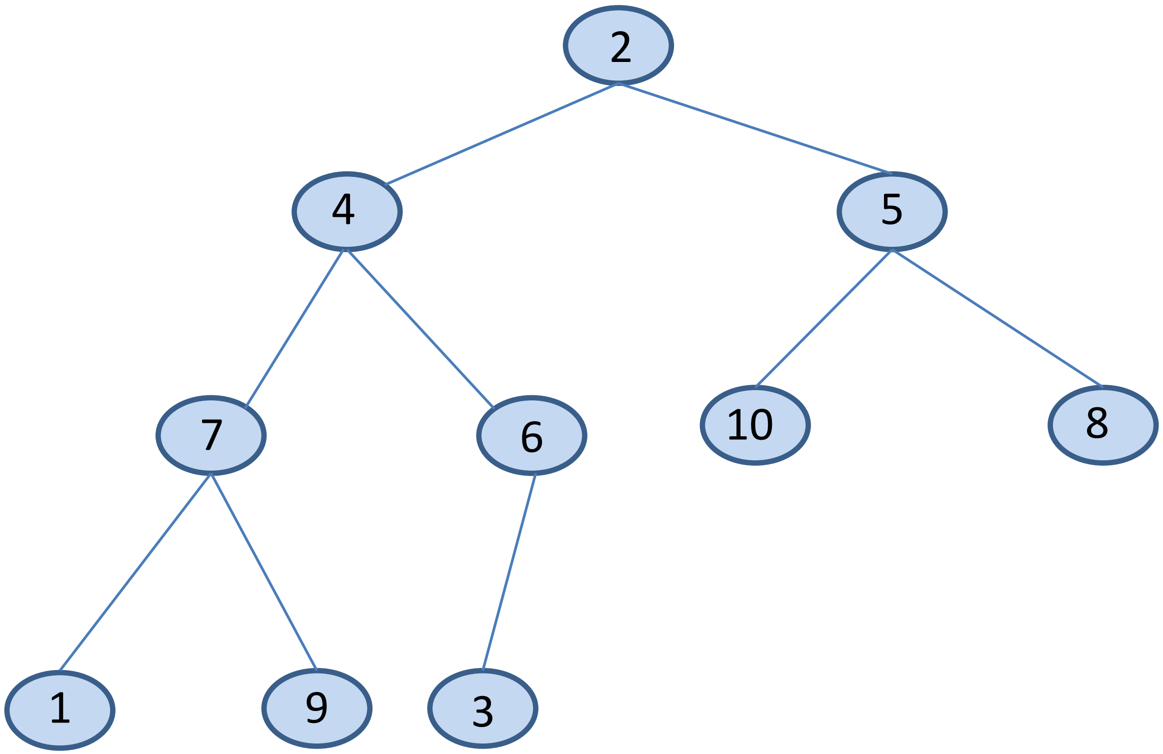
We start with the node with value 3. Exchange 3 with 6.

Heapifying the Tree (cont'd)



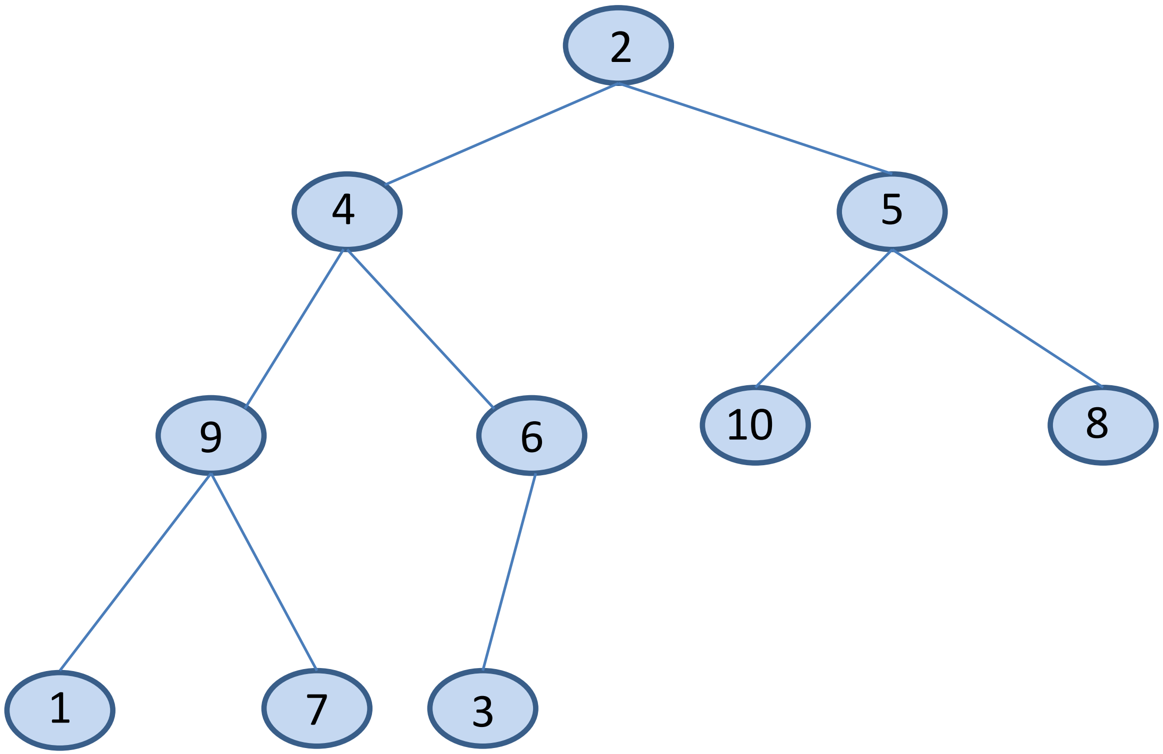
Since the node with value 3 has no children, we have nothing more to do in this subtree.

Heapifying the Tree (cont’d)



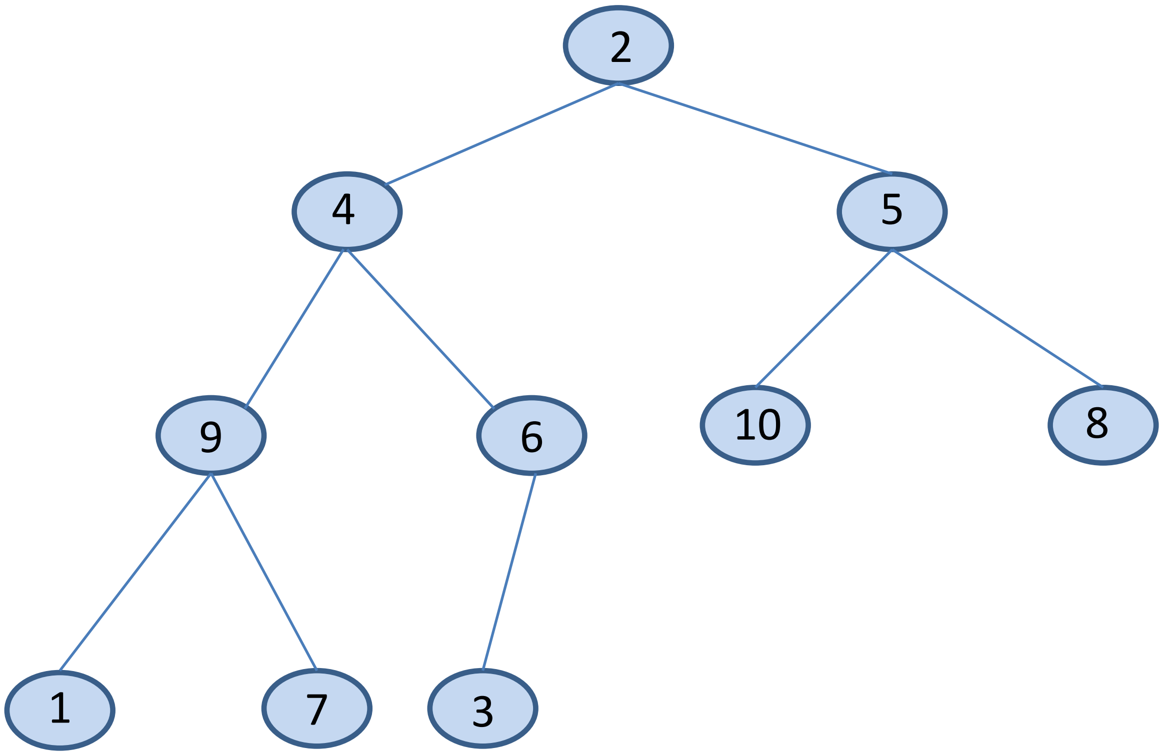
We continue with the subtree rooted at 7. We exchange 7 with 9.

Heapifying the Tree (cont'd)



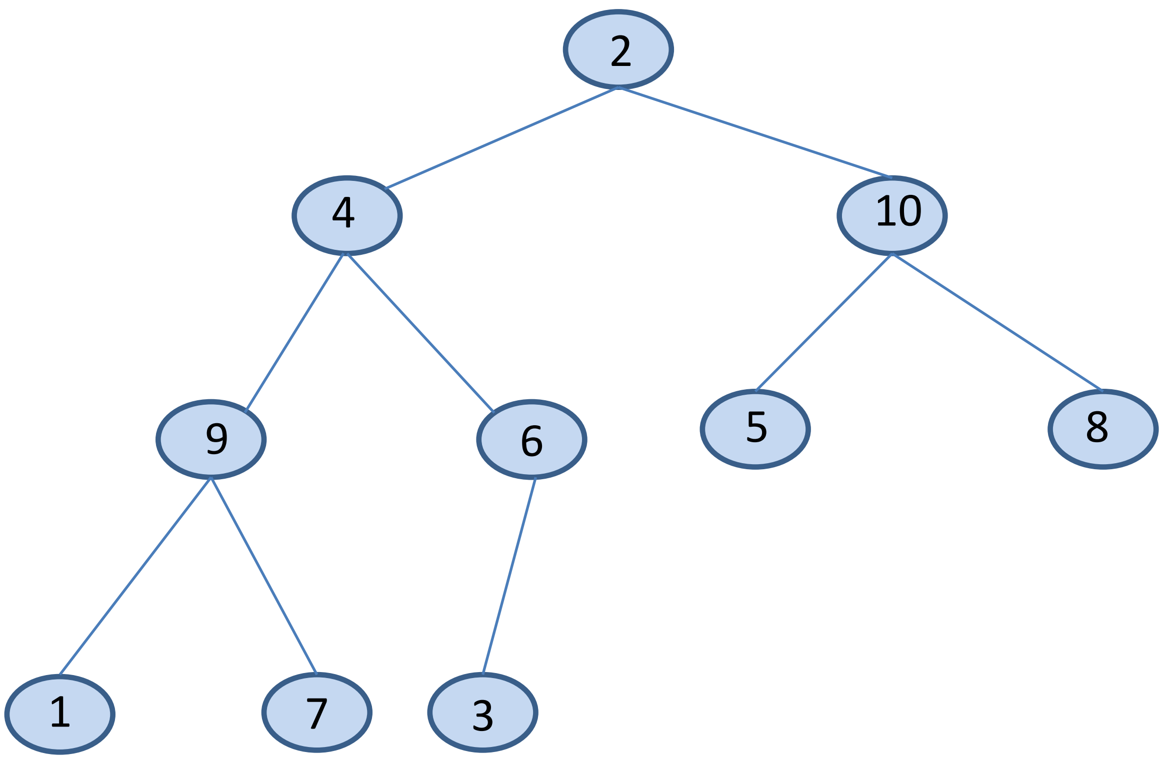
There is nothing more to do in this subtree.

Heapifying the Tree (cont'd)



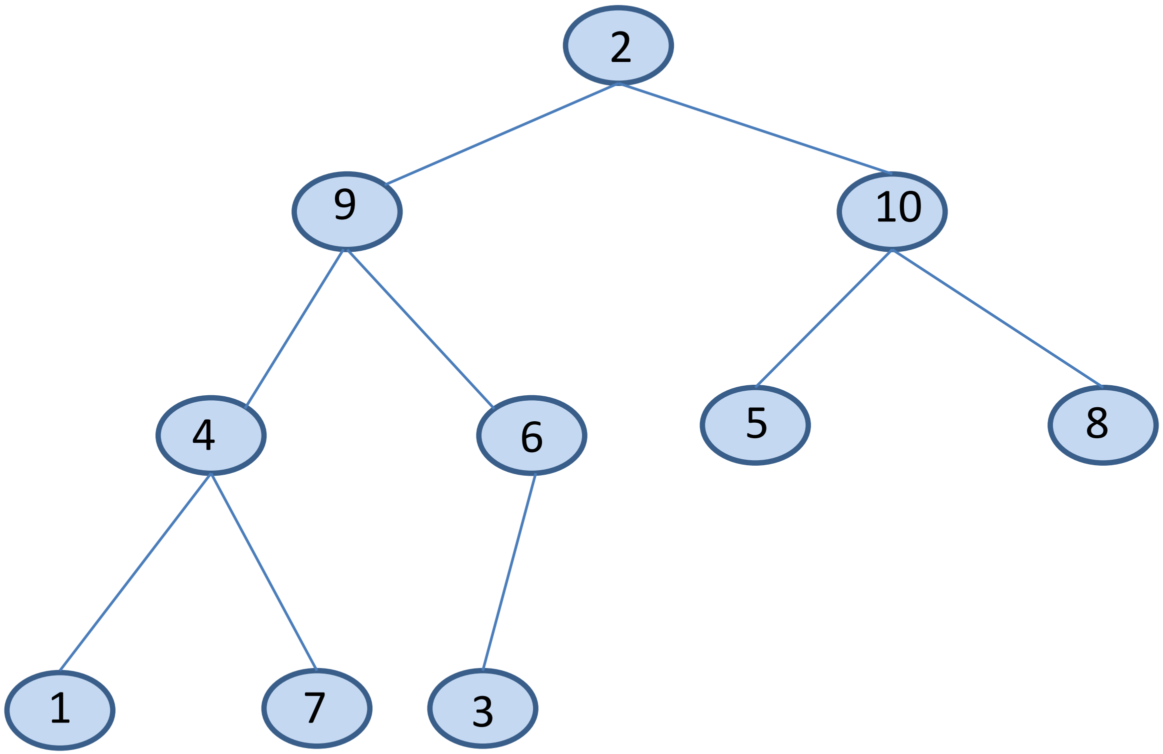
We continue with the subtree rooted at 5. We exchange 5 with 10.

Heapifying the Tree (cont'd)



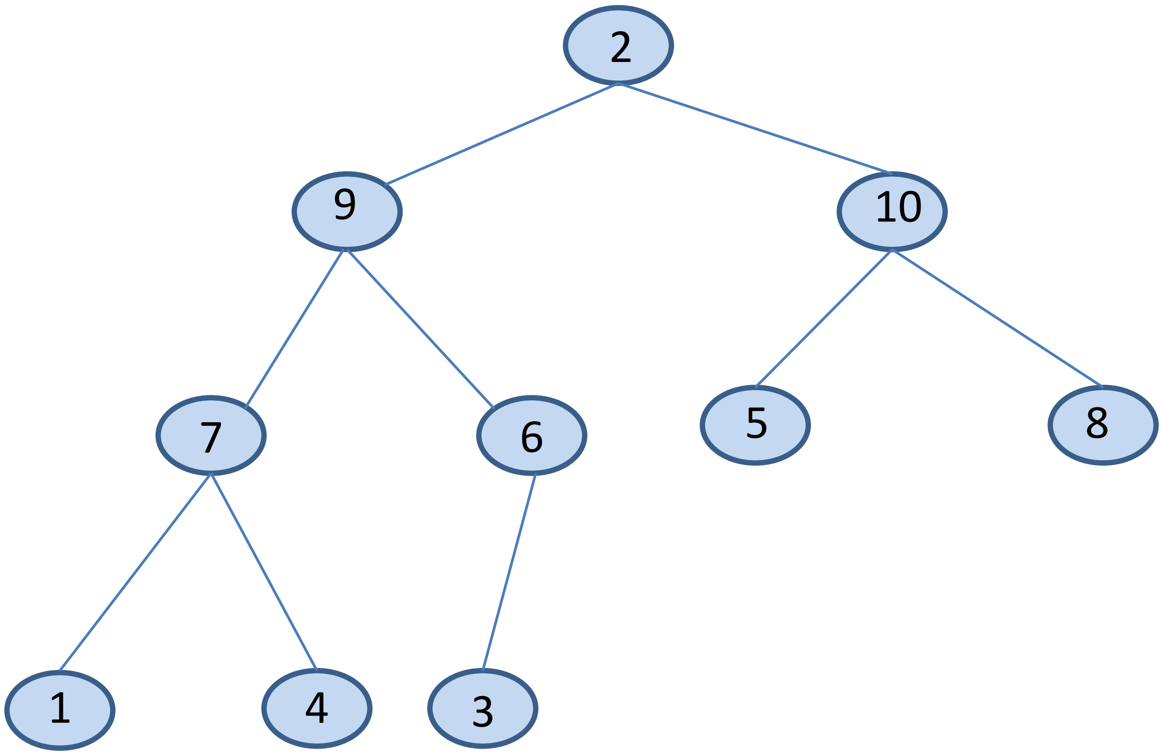
We have nothing more to do in this subtree. We continue with the subtree rooted at 4. We exchange 4 with 9.

Heapifying the Tree (cont'd)



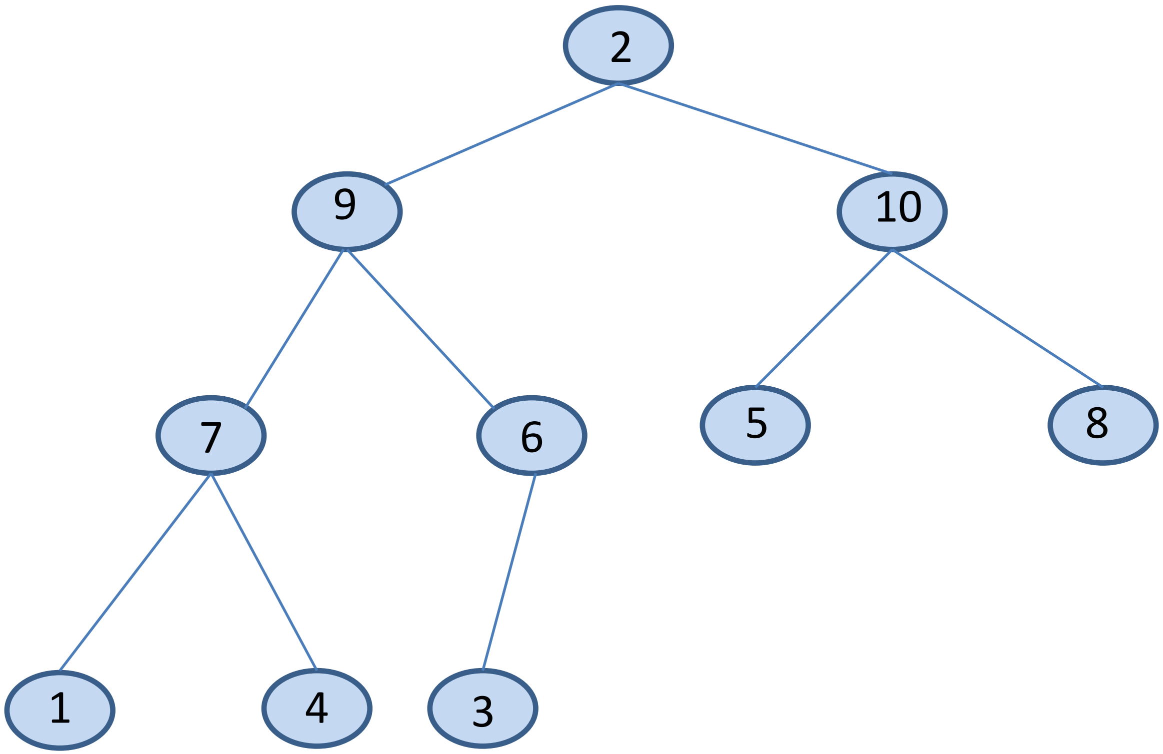
We now exchange 4 with 7.

Heapifying the Tree (cont'd)



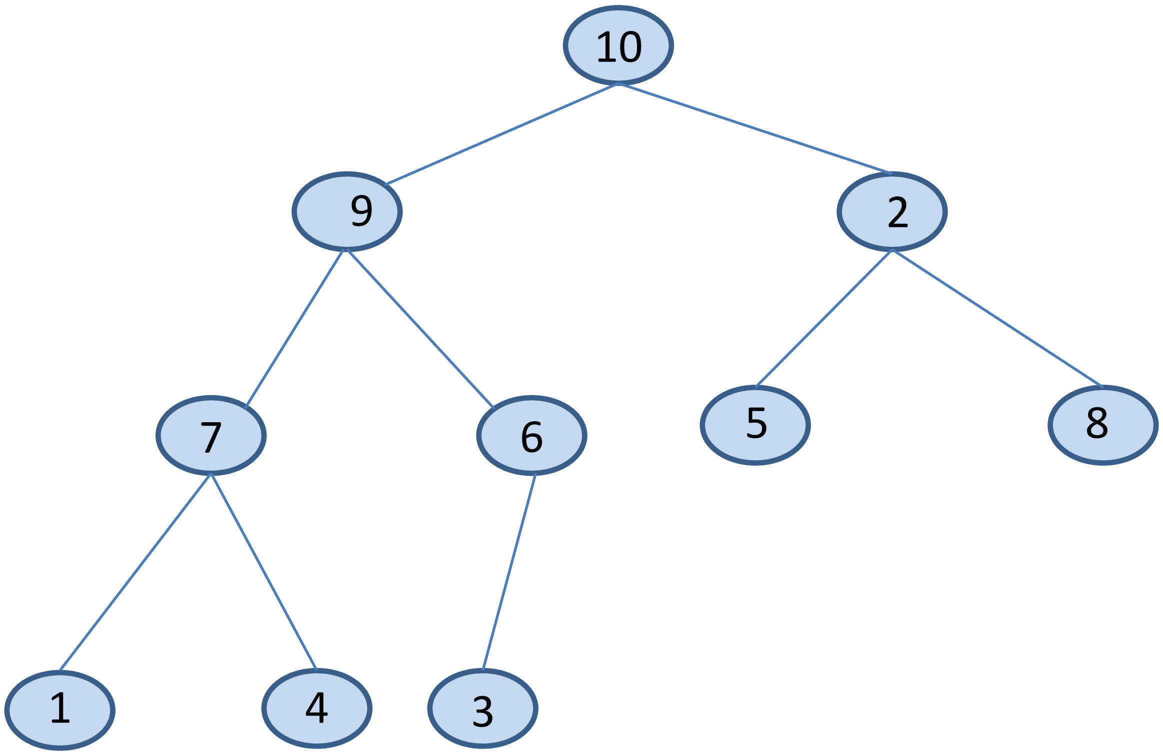
We have nothing more to do in this subtree. We now consider the root having value 2.

Heapifying the Tree (cont'd)



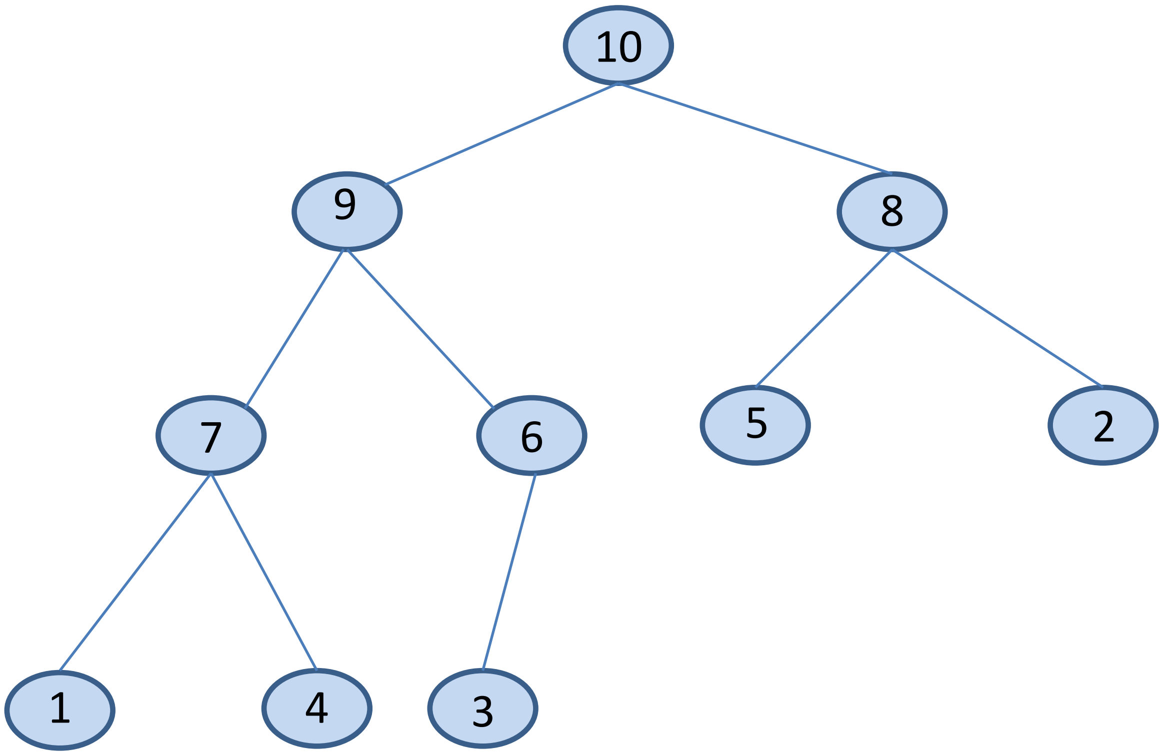
We exchange 2 with 10.

Heapifying the Tree (cont’d)



We exchange 2 with 8.

Heapifying the Tree (cont'd)



There is nothing more to do in this subtree. The tree has now been turned into a heap.

Heapifying the Tree (cont’d)

- The order the nodes are processed guarantees that **subtrees rooted at the children of node i are heaps** before the algorithm of heapification runs at that node.

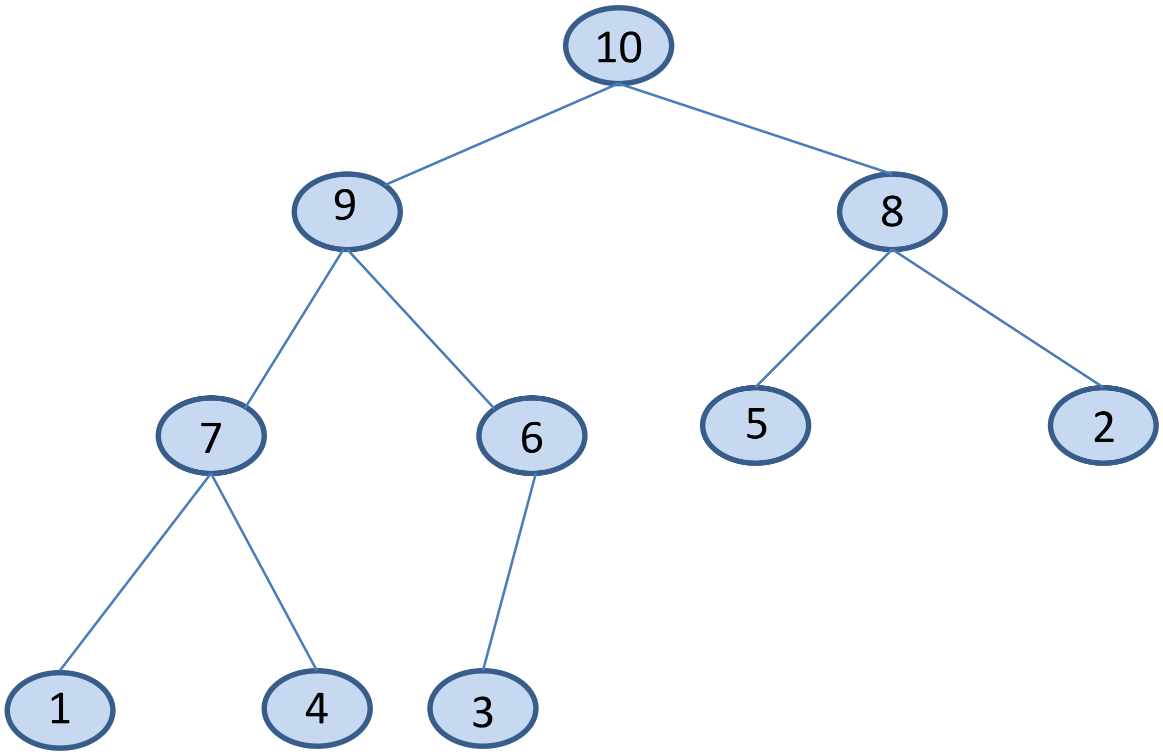
heapify-up (auxiliary function)

- Input: a **leaf n**
- Preconditions:
  - the all nodes **except n** satisfy the heap property
- Postcondition:
  - the whole tree is a heap
- Algorithm:
  - If **n** is <= its parent stop
  - Otherwise **exchange n** with its **parent**
  - Continue the process with the new position of **n**

insert

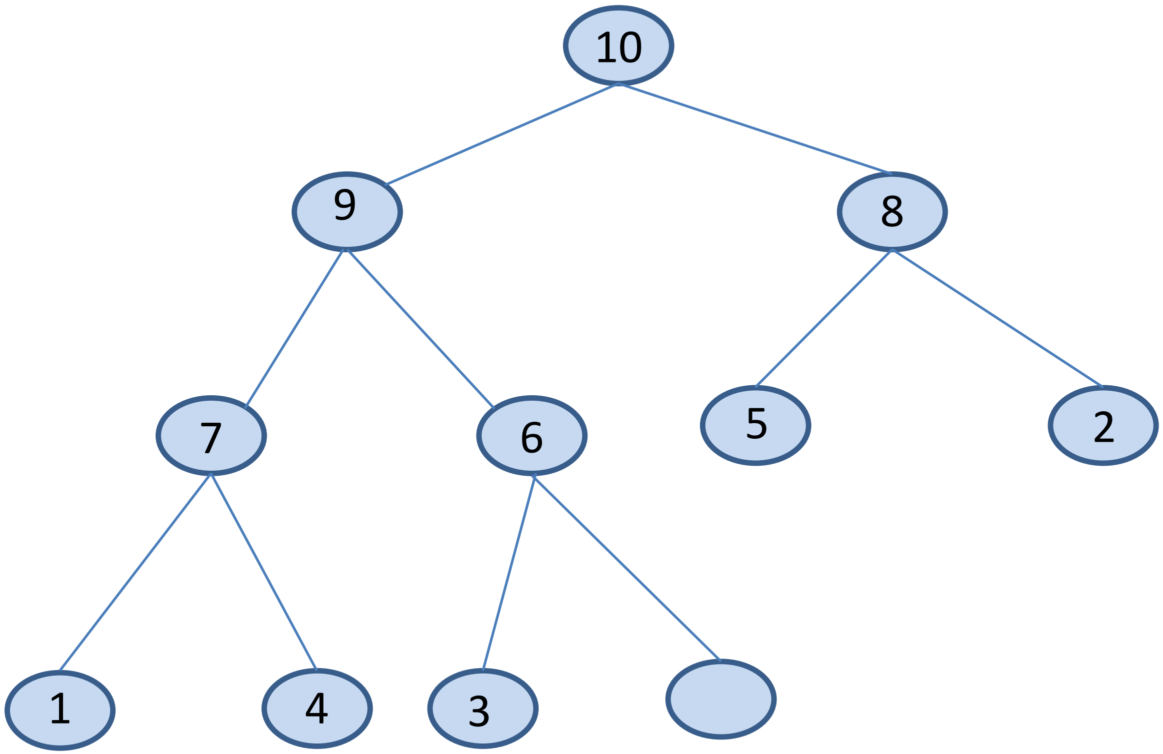
- Insert a new element
- Restores the heap property
- Algorithm:
  - Insert as a **leaf n** at the end of the complete tree
    - The tree is **not a heap** anymore
    - **BUT:** only n violates the property! so...
  - Call **heapify-up(n)**

Example Insertion



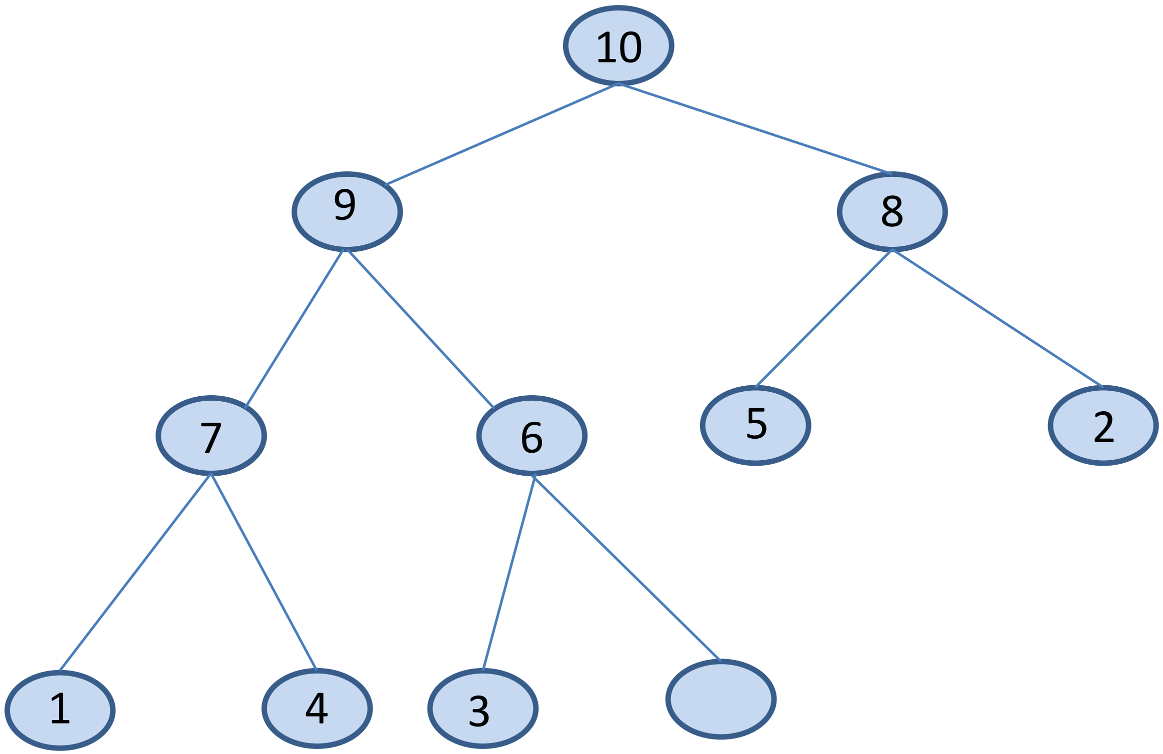
Let us insert the new element 15 in the heap.

Example Insertion (cont'd)



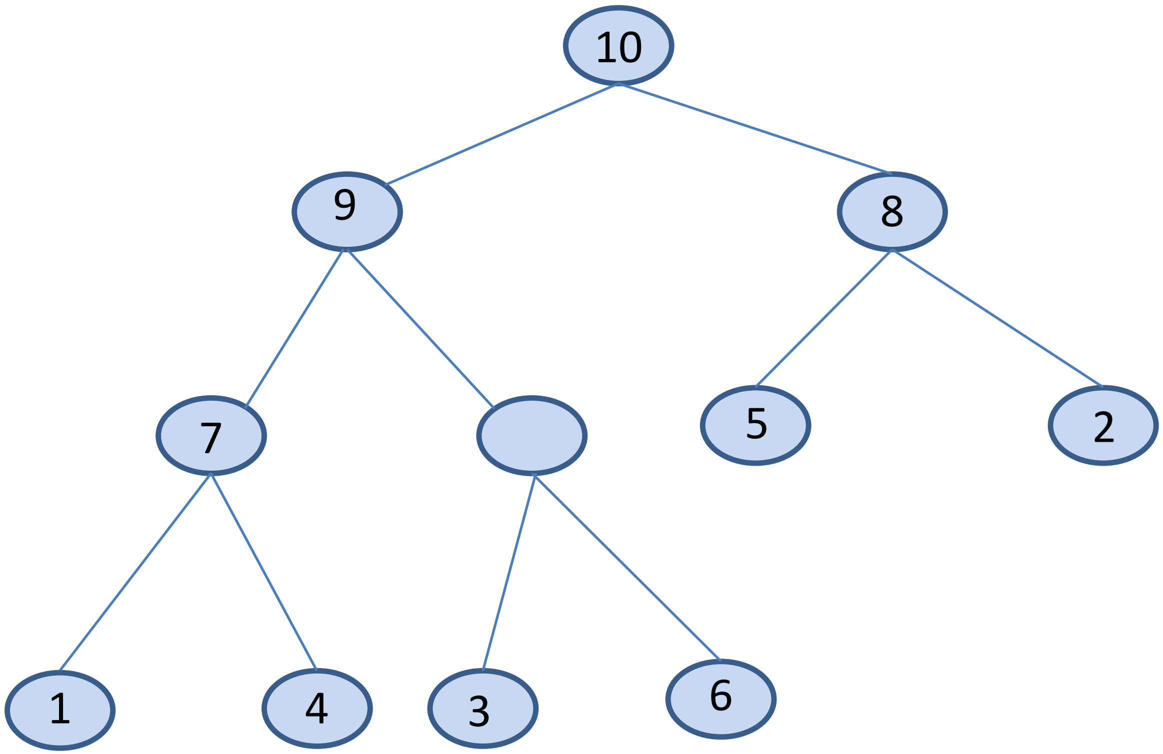
A new leaf is added to the tree at the first available position at the bottom level.

Example Insertion (cont'd)



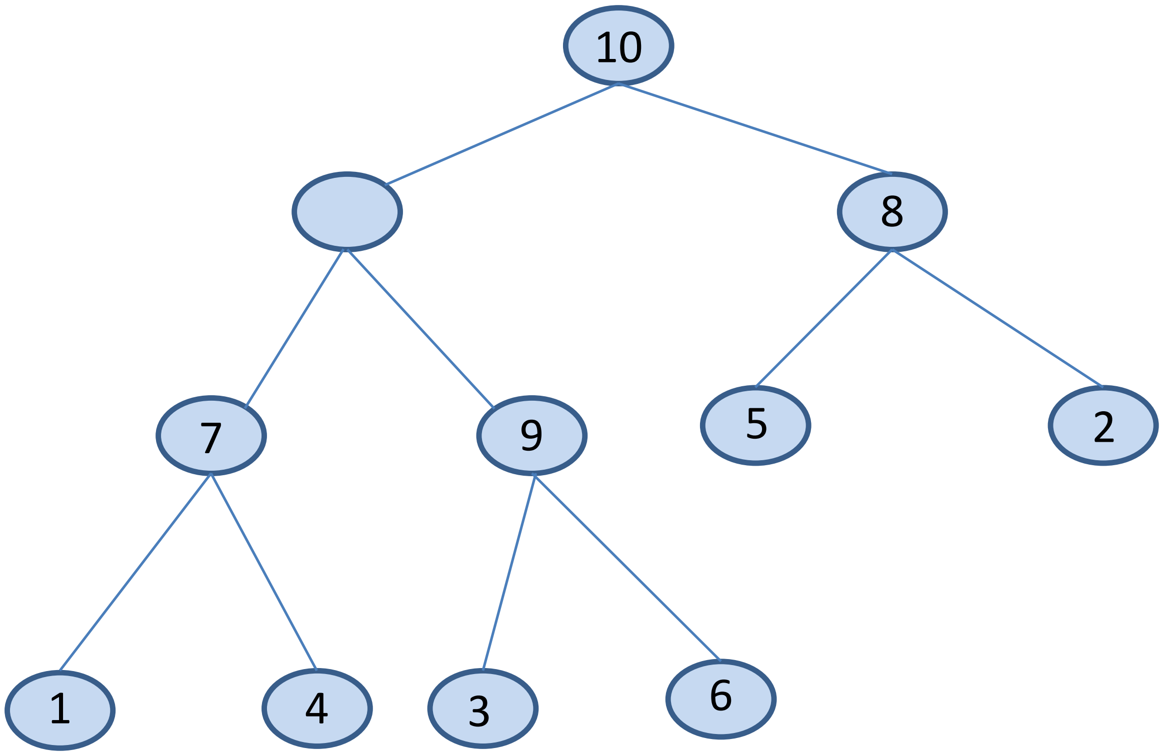
Values on the path from the new leaf node to the root are copied down until a place for the key 15 is found.

Example Insertion (cont'd)



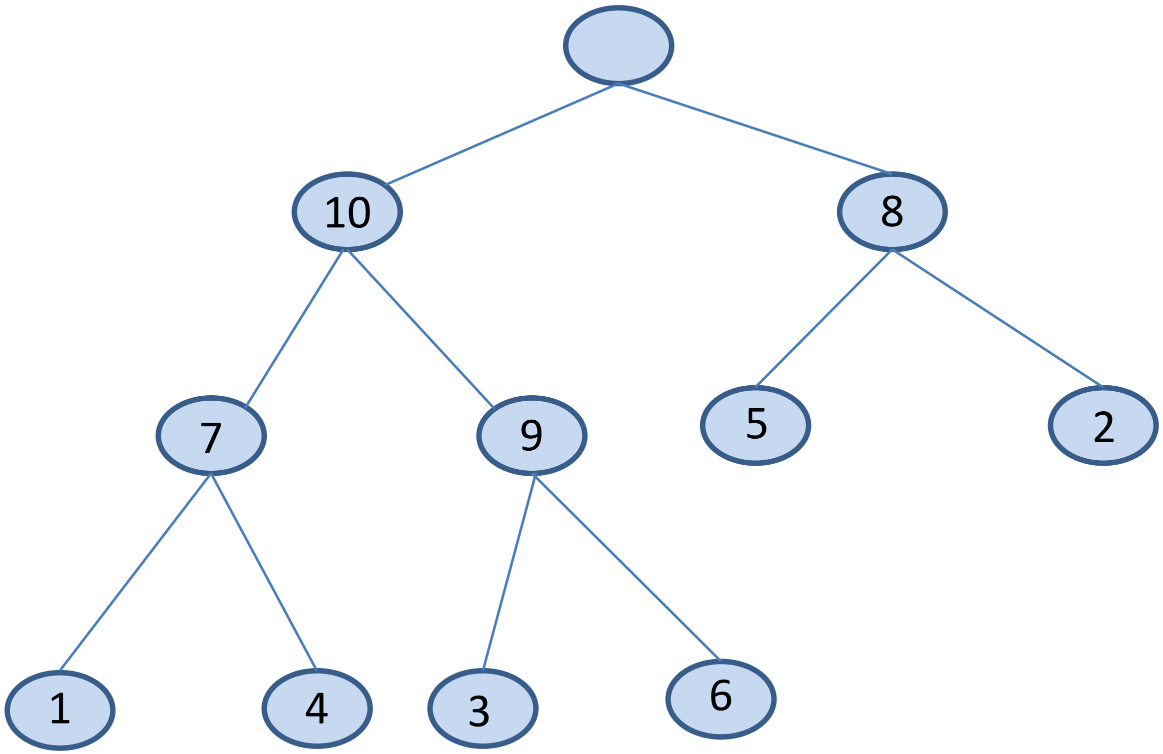
6 is copied down.

Example Insertion (cont'd)



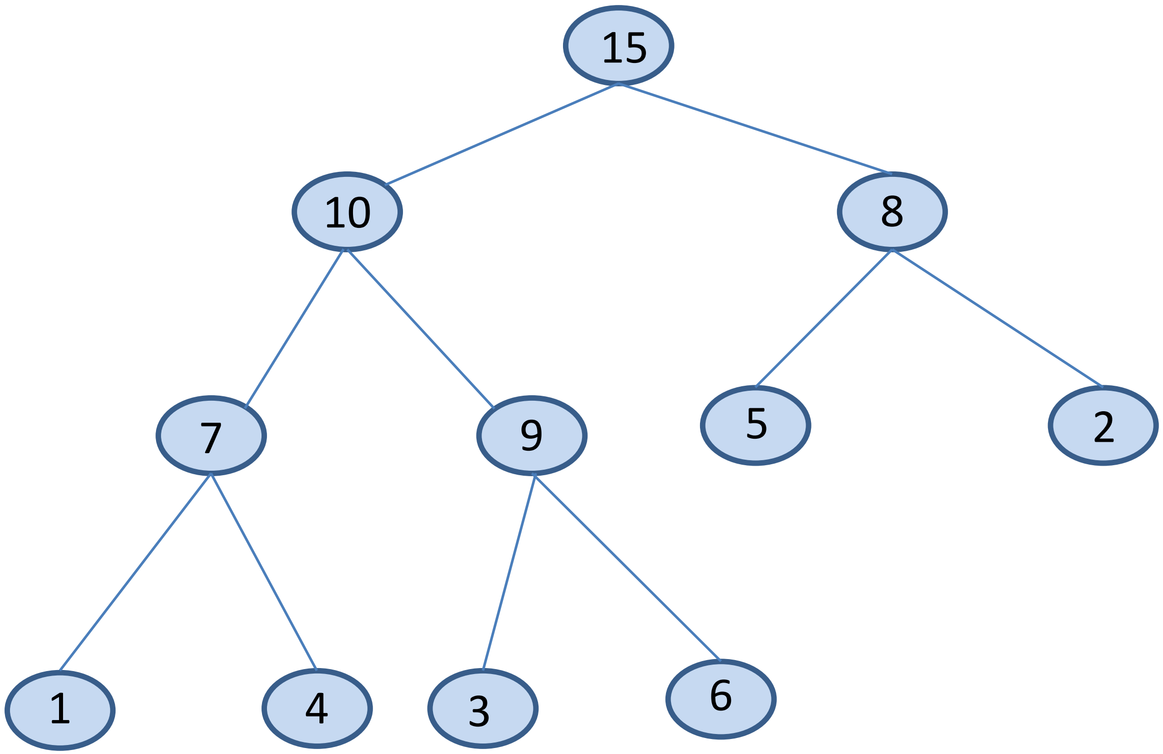
9 is copied down.

Example Insertion (cont’d)

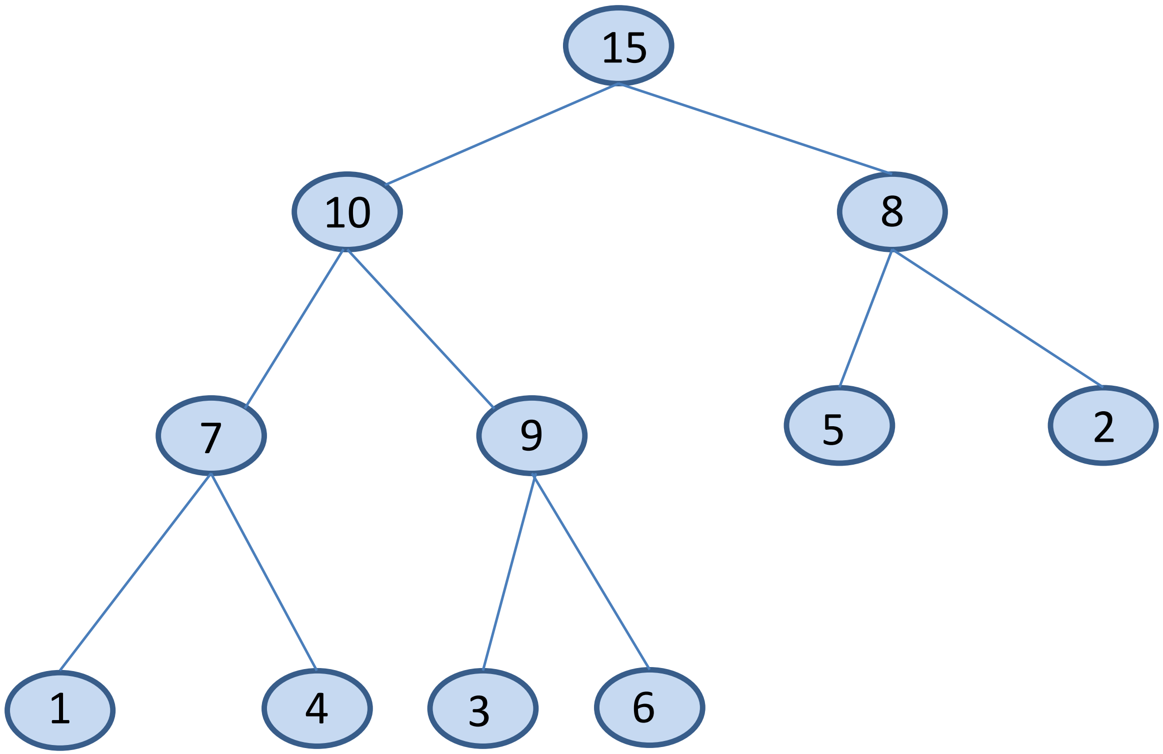


10 is copied down. Now a place for 15 has been found.

Example Insertion (cont'd)

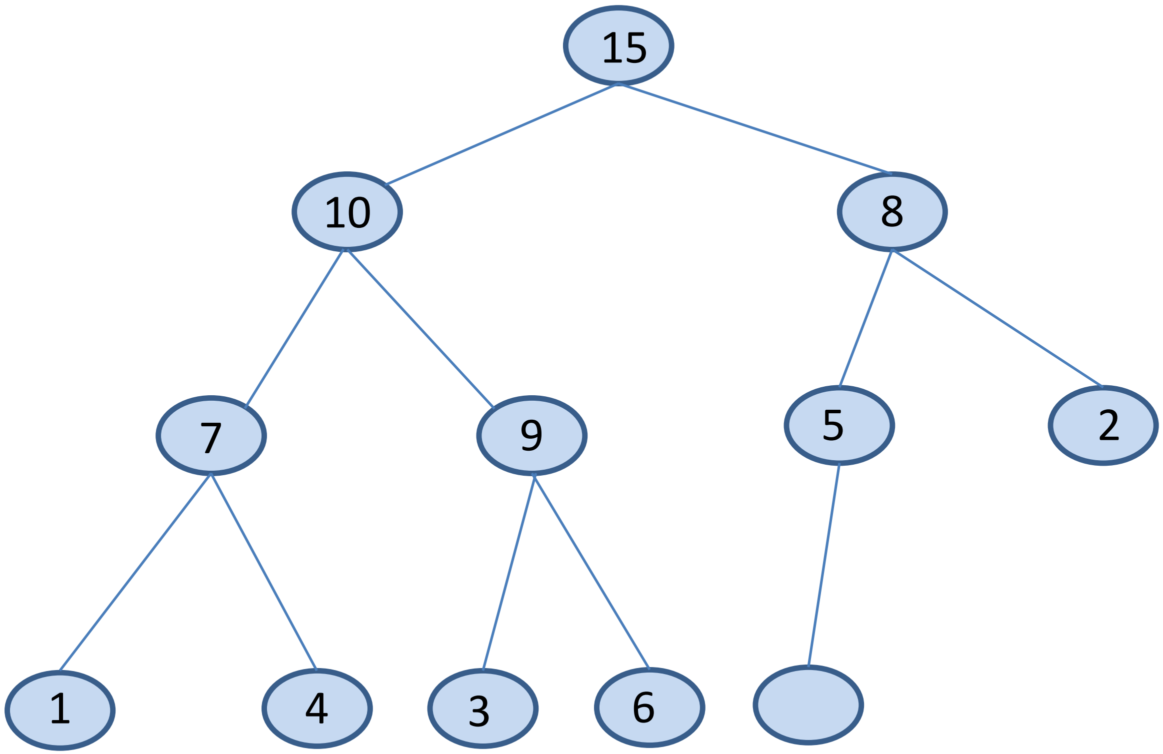


Example Insertion



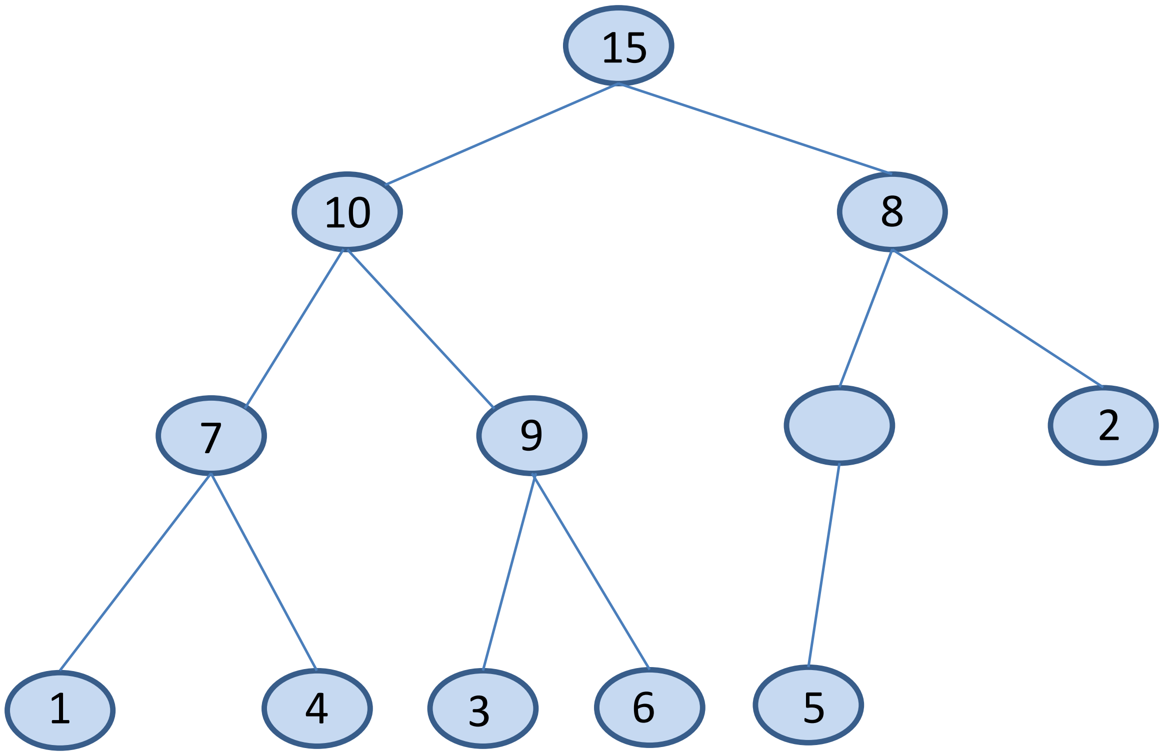
Let us now insert the key 12 in the heap.

Example Insertion (cont'd)



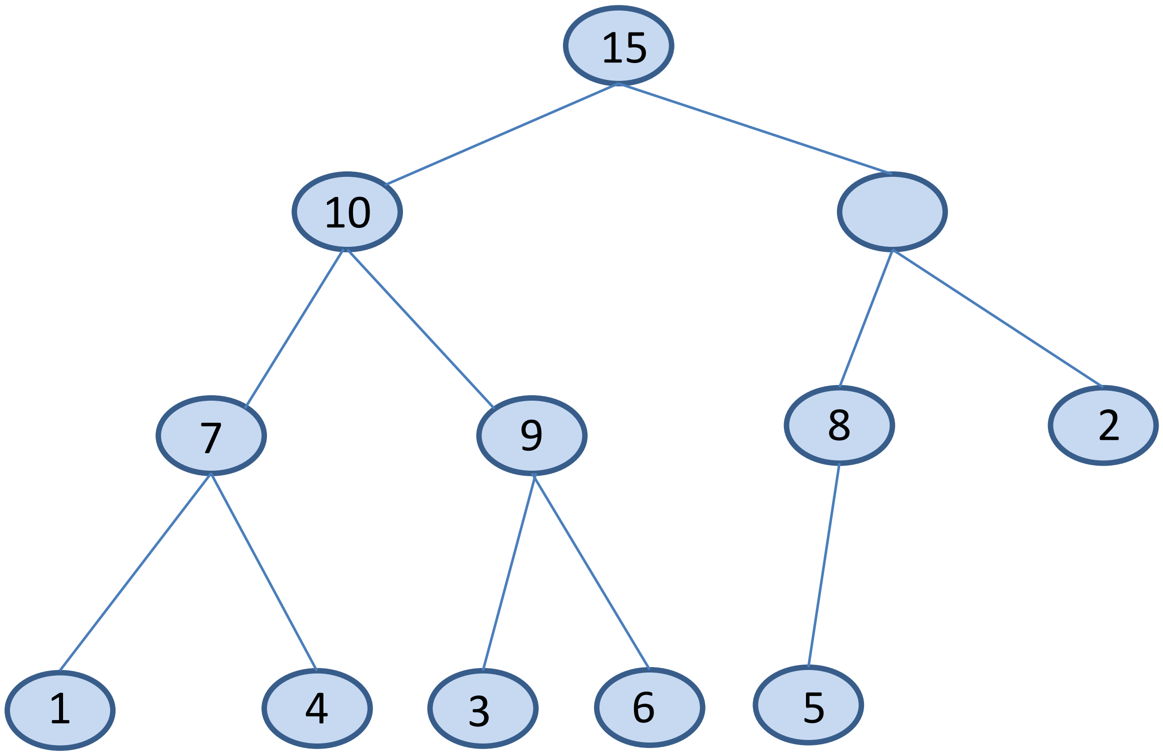
A new empty leaf is added to the tree at the first available place in the bottom level.

Example Insertion (cont'd)



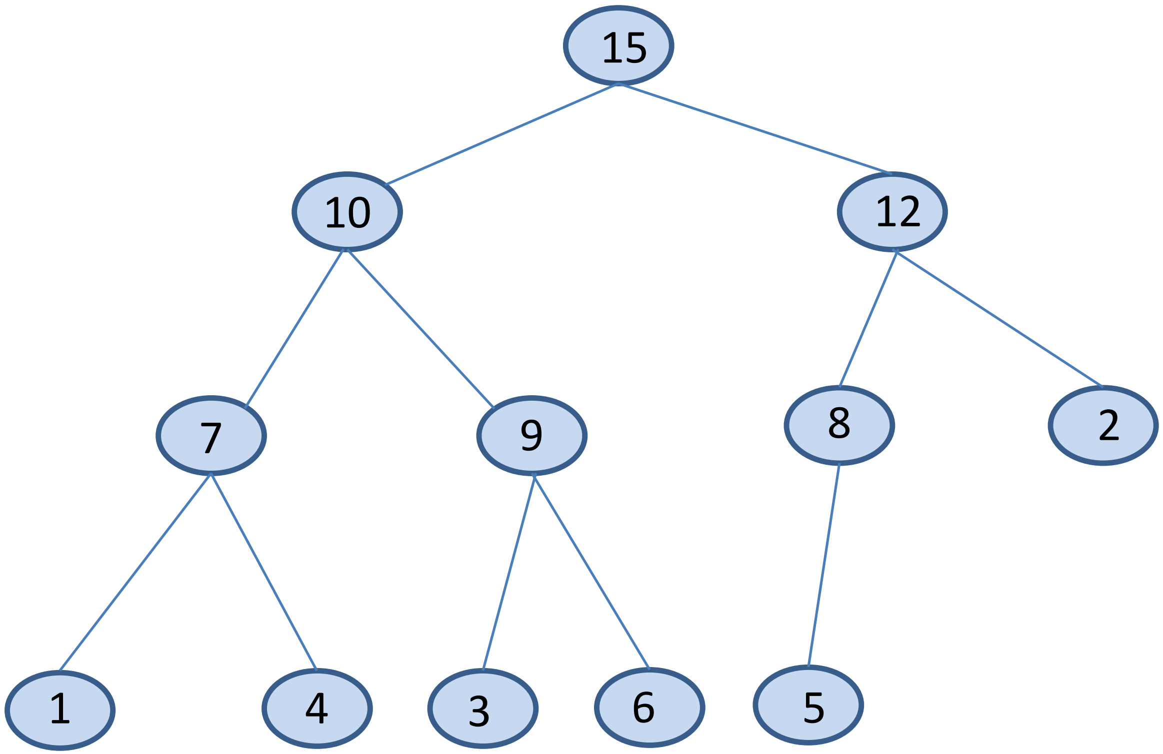
5 is copied down.

Example Insertion (cont'd)



8 is copied down.

Example Insertion (cont'd)



Now a place for 12 has been found.

Implementation of Priority Queues Using Heaps

- Let us now develop a third implementation of the Priority Queue ADT using heaps implemented by the sequential representation of binary trees.

The Priority Queue ADT

- A **priority queue** is a finite collection of items for which the following operations are defined:
  - **Initialize** the priority queue, *PQ*, to the empty priority queue.
  - Determine whether or not the priority queue, *PQ*, is **empty**.
  - Determine whether or not the priority queue, *PQ*, is **full**.
  - **Insert** a new item, *X*, into the priority queue, *PQ*.
  - If *PQ* is non-empty, **remove** from *PQ* an item *X* of highest priority in *PQ*.

The Priority Queue Data Types

```
/* This is the file "PQTypes.h" */

#define MAXCOUNT 10
typedef int PQItem;
typedef PQItem PQArray[MAXCOUNT+1];

typedef struct {
            int Count;
            PQArray ItemArray;
        } PriorityQueue;
```

The Priority Queue Interface File

```
/* this is the file “PQInterface.h”      */

#include “PQTypes.h”
/* defines types PQItem and PriorityQueue */

void Initialize (PriorityQueue *);
int Empty (PriorityQueue *);
int Full (PriorityQueue *);
void Insert (PQItem, PriorityQueue *);
PQItem Remove (PriorityQueue *);
```

The Priority Queue Implementation File

```
/* This is the file "PQImplementation.c" */

#include "PQInterface.h"

void Initialize(PriorityQueue *PQ)
{
    PQ->Count=0;
}

int Empty(PriorityQueue *PQ)
{
    return(PQ->Count==0);
}

int Full(PriorityQueue *PQ)
{
    return(PQ->Count==MAXCOUNT);
}
```

The Priority Queue Implementation File (cont'd)

```
void Insert(PQItem Item, PriorityQueue *PQ)
{
    int ChildLoc;
    int ParentLoc;

    (PQ->Count)++;
    ChildLoc=PQ->Count;
    ParentLoc=ChildLoc/2;
    while (ParentLoc != 0){
       if (Item <= PQ->ItemArray[ParentLoc]){
         PQ->ItemArray[ChildLoc]=Item;
         return;
       } else {
         PQ->ItemArray[ChildLoc]=PQ->ItemArray[ParentLoc];
         ChildLoc=ParentLoc;
         ParentLoc=ParentLoc/2;
       }
    }
    PQ->ItemArray[ChildLoc]=Item;
}
```

Notes

- The previous algorithm first **introduces a new empty node** in the first available position in the complete binary tree.
- Then, it **propagates this empty node upwards** on the path towards the root, until the correct location is found where the new value can be inserted without violating the heap property.

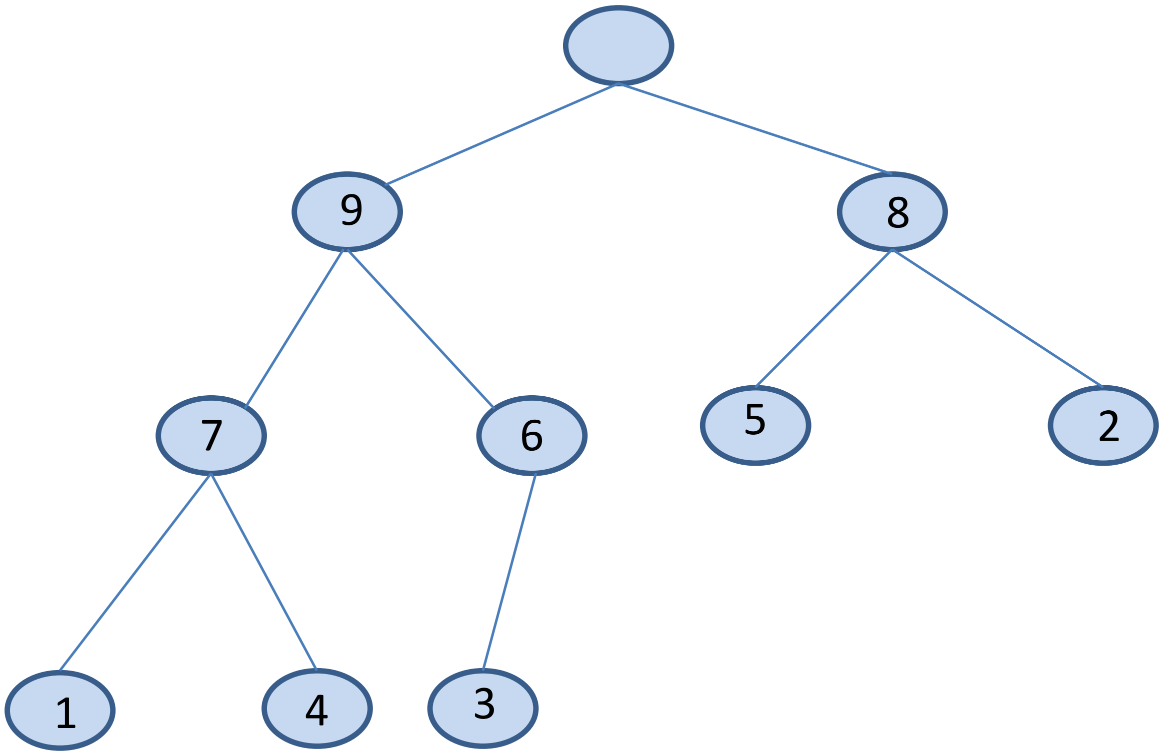
The Priority Queue Implementation File (cont'd)

```
PQItem Remove(PriorityQueue *PQ)
{
   int CurrentLoc;
   int ChildLoc;
   PQItem ItemToPlace;
   PQItem ItemToReturn;

   if(Empty(PQ)) return;

   ItemToReturn=PQ->ItemArray[1];
   ItemtoPlace=PQ->ItemArray[PQ->Count];
   (PQ->Count)--;
   CurrentLoc=1;
   ChildLoc=2*CurrentLoc;
```

The Priority Queue Implementation File (cont'd)

```
    while (ChildLoc <= PQ->Count){
       if (ChildLoc < PQ->Count){
          if (PQ->ItemArray[ChildLoc+1] > PQ->ItemArray[ChildLoc]){
             ChildLoc++;
          }
       }
       if (PQ->ItemArray[ChildLoc] <= ItemToPlace){
           PQ->ItemArray[CurrentLoc]=ItemToPlace;
           return(ItemToReturn);
       } else {
           PQ->ItemArray[CurrentLoc]=PQ->ItemArray[ChildLoc];
           CurrentLoc=ChildLoc;
           ChildLoc=2*CurrentLoc;
       }
    }
    PQ->ItemArray[CurrentLoc]=ItemToPlace;

    return(ItemToReturn);
}
```

Comments on the `Remove` Function



The `Remove` function optimizes our earlier method by moving 9, 7 and 4 upward to make a hole where 3 will come to be inserted.

Complexity of Removing an Item from a Heap

- To remove an item from a heap *H*, we must delete the last leaf *L* in level order, and then reheapify the tree that results from replacing the root's value with *L*'s value *V*.
- During reheapification, we repeatedly exchange the value *V* with the larger values of the children nodes on some path from the root downward toward *V*'s final resting place.
- The **longest possible path** for these pairwise exchanges is a path from the root to some leaf on the bottommost row of *H*.
- The longest path from the root to a bottom leaf is the **height** of *H* (equivalently, the **level number** of the bottom row in *H*).
- For a complete binary tree with n items, the height is given by $\lfloor \mathbf{log_2\, n} \rfloor$ i.e., the **largest integer smaller than or equal to $\mathbf{log_2\, n}$**.
- Therefore, **removal of an item from a heap takes $\boldsymbol{O(\log n)}$ time**.

Proposition

- A heap T storing n entries has height $h = \lfloor \log n \rfloor$.
- Proof?

Proof

- From the fact that T is a complete binary tree, we know that there are 2^i nodes in level i, for $0 \leq i \leq h-1$, and level h has at least 1 node.

- Thus, the number of nodes of T is at least $(1+2+4+\cdots+2^{h-1})+1=(2^h-1)+1$ $=2^h$

Proof (cont'd)

- Level h has at most 2^h nodes, and thus the number of nodes of T is at most

$$\left(1 + 2 + 4 + \cdots + 2^{h-1}\right) + 2^h = 2^{h+1} - 1.$$

- Since the number of nodes is equal to the number of entries n, we obtain $2^h \leq n$ and $n \leq 2^{h+1} - 1$.

Proof (cont'd)

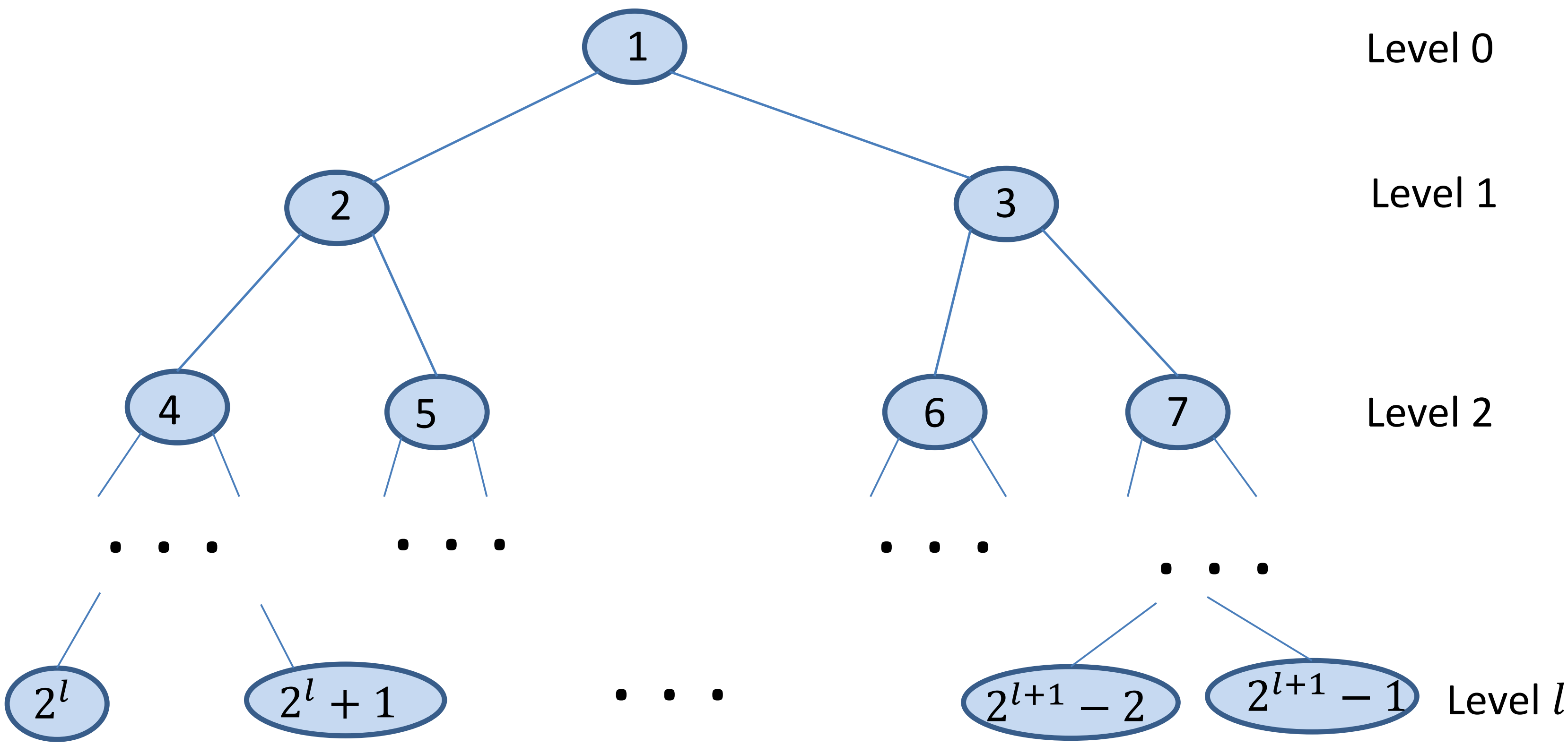
- Thus, by taking logarithms of both sides of these two inequalities, we see that $h \leq \log n$ and $\log(n+1) - 1 \leq h$.

- Since h is an integer, the two inequalities above imply that $h = \lfloor \log n \rfloor$.

Complexity of Inserting an Item in a Heap

- Similarly, an insertion can cause pairwise exchanges of node values to occur along a path from a leaf in the bottom row upward all the way to the root.
- Thinking in a similar way, we can see that **insertion can also be done in $\boldsymbol{O(\log n)}$ time**.

Complexity of Making a Heap



Complexity of Making a Heap (cont'd)

- Suppose the tree we are considering is like the one on the previous slide and has l levels.
- An item at level i could be exchanged with children along any downward path at most $(l - i)$ times before coming to rest.
- The tree contains 2^i nodes on level i.
- Since each of the 2^i nodes on level i could be exchanged downward at most $(l - i)$ times, the cost of processing the nodes on level i is $(l - i) * 2^i$.

Complexity of Making a Heap (cont'd)

- Therefore, the total number of exchanges needed to apply the heapifying process to all nodes on all levels except the bottom level could not exceed the sum S below:

$$S = \sum_{i=0}^{(l-1)} (l - i) * 2^i$$

- S can be shown to be less than $2n$ (exercise!).
- Therefore **the heapifying process has complexity $\boldsymbol{O(n)}$**.

Comparing Running Times of Priority Queue Operations for the Three Representations

Priority Queue Operation	Heap Representation	Sorted List Representation	Unsorted Array Representation
Organize a priority queue	$O(n)$	$O(n^2)$	$O(1)$
Remove highest priority item	$O(\log n)$	$O(1)$	$O(n)$
Insert a new item	$O(\log n)$	$O(n)$	$O(1)$

Heapsort

- If we use the PriorityQueue ADT implementation developed here to sort an array (as we have done in the past with the other two representations), then we have a version of the sorting algorithm **heapsort**.

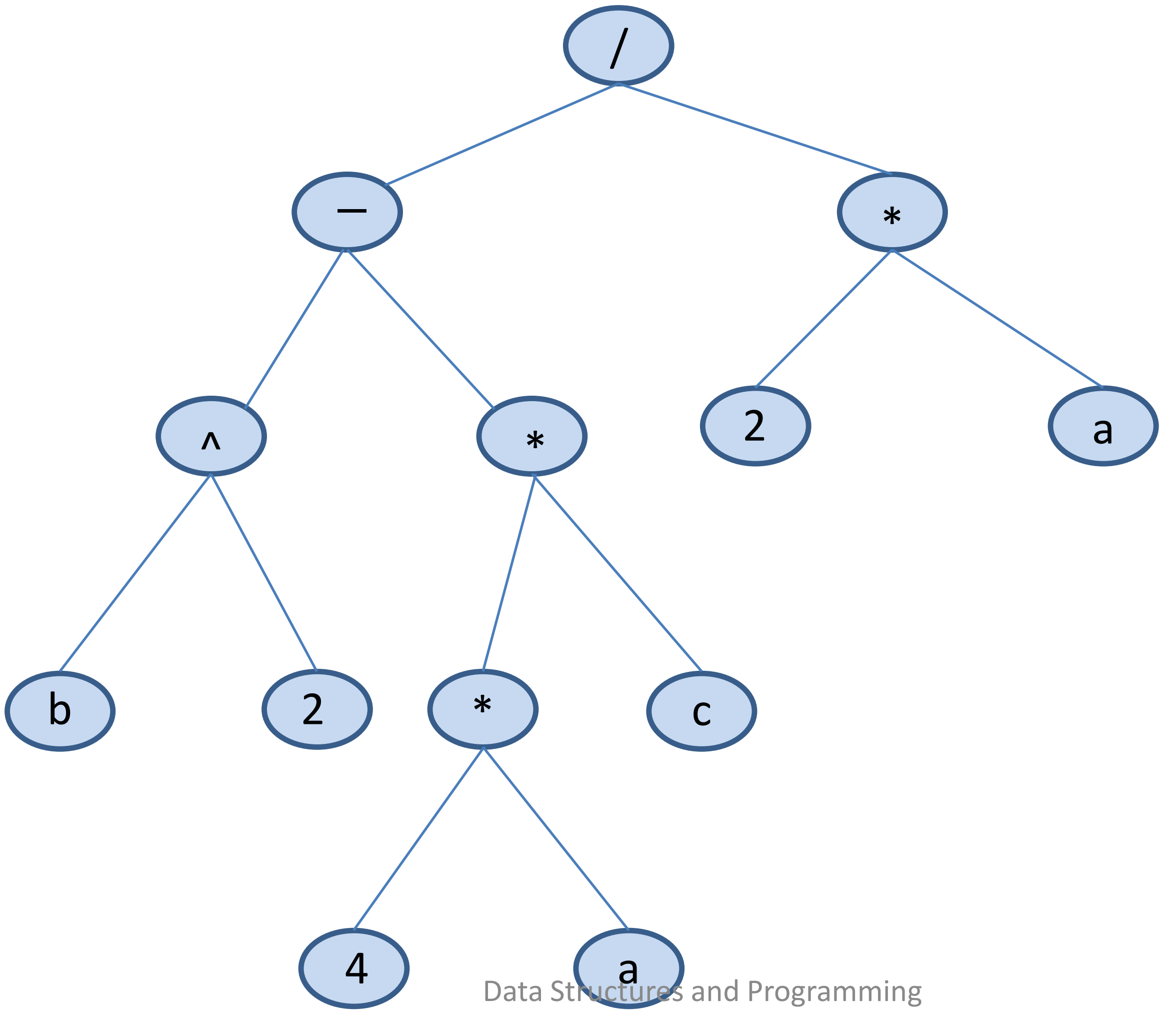
- While sorting using the other representations of priority queues takes time $O(n^2)$, **heapsort can be shown to take time $\boldsymbol{O(n\log n)}$.**

Expression Trees

- **Expression trees** (δένδρα εκφράσεων) are binary trees used to represent algebraic expressions formed with binary operators.

- Example: the tree on the next slide is an expression tree for the following algebraic expression: $(b^2 - 4 * a * c)/(2 * a)$

Example (cont’d)



Parse Trees

- Compilers parse algebraic expressions (and, in general, program statements) and build **parse trees** (δένδρα συντακτικής ανάλυσης) such as the expression tree of the previous example.
- Parse trees are then traversed by code generators to produce assembly code.

Traversing Binary Trees

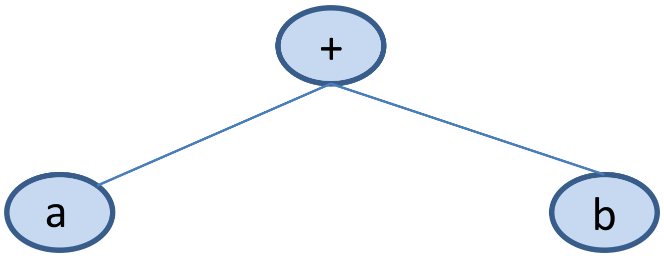
- A traversal of a tree is a process that visits each node in the tree exactly once in some particular order.

- Three popular traversal orders for binary trees are **preorder**, **inorder** and **postorder** (προδιατεταγμένη, ενδοδιατεταγμένη και μεταδιατεταγμένη διάταξη).

Traversal Orders for Binary Trees

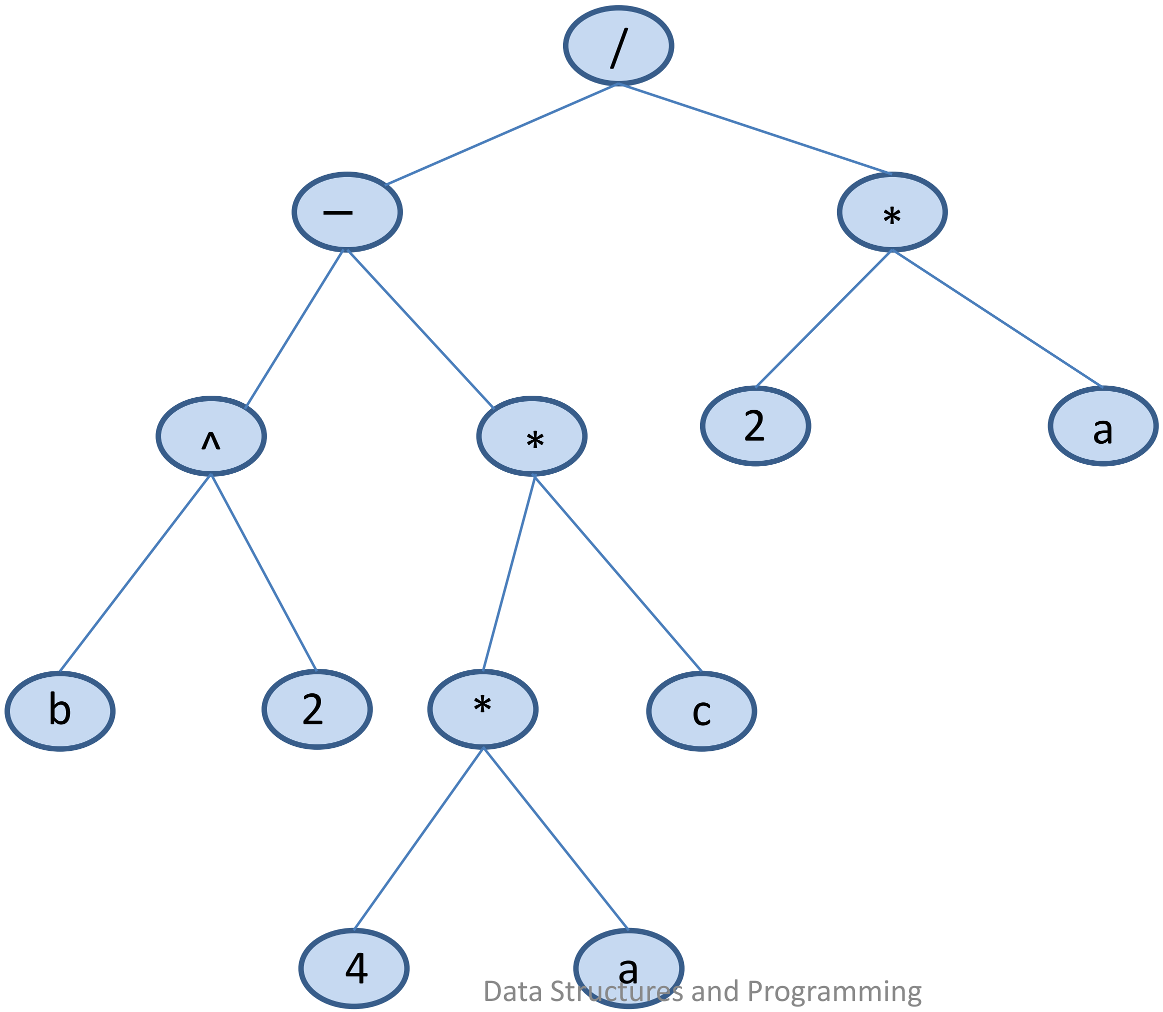
PreOrder	InOrder	PostOrder
Visit the root	Traverse left subtree in InOrder	Traverse left subtree in PostOrder
Traverse left subtree in PreOrder	Visit the root	Traverse right subtree in PostOrder
Traverse right subtree in PreOrder	Traverse right subtree in InOrder	Visit the root

Example



- If, when we visit a node, we print the character contained in the node, then, for the above example tree, we have:
  - PreOrder: *+a b*
  - InOrder: *a+b*
  - PostOrder: *ab+*

Example



Example (cont'd)

- PreOrder: */ – ^ b 2 * * 4 a c * 2 a*
- InOrder: *b ^ 2 – 4 * a * c / 2 * a*
- PostOrder: *b 2 ^ 4 a * c * – 2 a * /*

- The preorder traversal of an expression tree gives the **prefix** (προθεματική) representation of the expression.
- The postorder traversal of an expression tree gives the **postfix** (μεταθεματική) representation of the expression.
- The inorder traversal of an expression tree gives the **infix**(ενδοθεματική) representation of the expression without parentheses.

Prefix Representation

- The prefix expression for a single operand A is A itself.
- The prefix expression for $(E_1)\ \theta\ (E_2)$ is $\theta P_1 P_2$ where P_1 and P_2 are the prefix expressions for E_1 and E_2 respectively.

- Note that **no parentheses are necessary** in the prefix expression, since we can scan the prefix expression $\theta P_1 P_2$ and uniquely identify P_1 as the shortest (and only) prefix of $P_1 P_2$ that is a legal prefix expression.

Postfix Representation

- The postfix expression for a single operand A is A itself.
- The postfix expression for $(E_1)\ \theta\ (E_2)$ is $P_1P_2\ \theta$ where P_1 and P_2 are the postfix expressions for E_1 and E_2 respectively.

- Note that **no parentheses are necessary** in the postfix expression, since we can deduce what P_2 is by looking for the shorter suffix of P_1P_2 that is a legal postfix expression.

Infix Representation

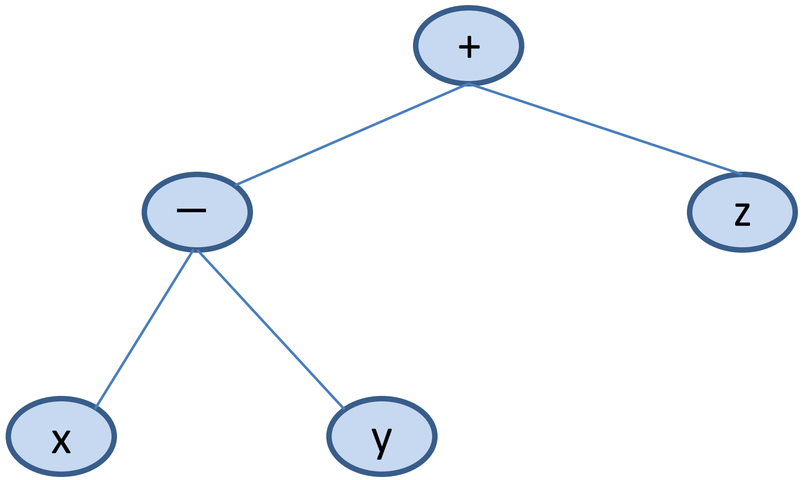
- **Question**: What is the algorithm for producing an infix representation with parentheses of a given arithmetic expression represented by an expression tree?

Traversals Using the Linked Representation of Binary Trees

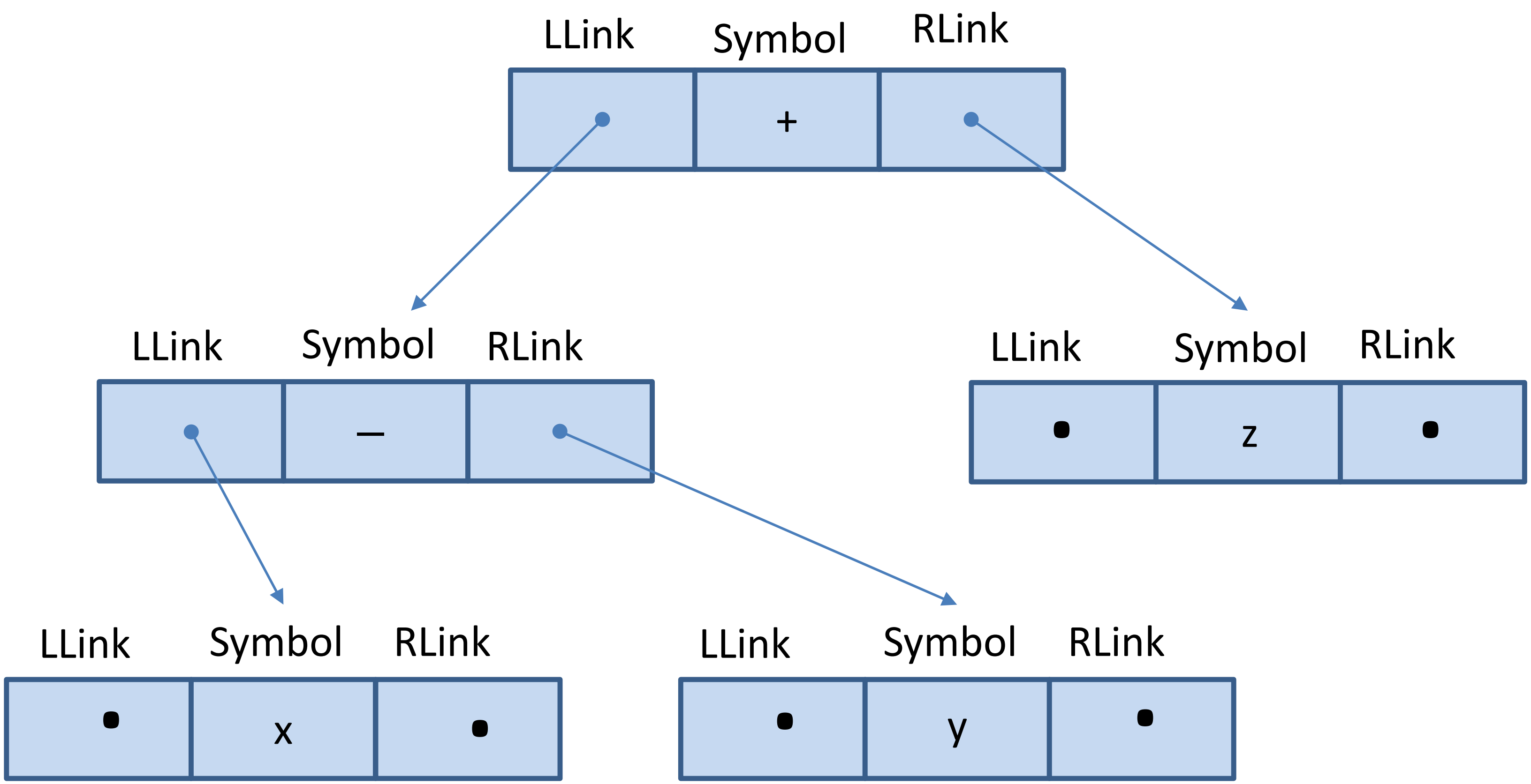
- We can declare the type for tree nodes to use in the linked representation as follows:

```
typedef struct NodeTag{
                char Symbol;
                struct NodeTag *LLink;
                struct NodeTag *RLink;
        } TreeNode;
```

Example Expression Tree



Its Linked Representation



The Linked Representation

- The linked representation of binary trees presented in the previous example can also be used when we consider **non-empty proper binary trees (extended trees).**
- In this case, **`NULL` links represent the special external nodes** we denote by a square.
- In the next lecture we will also see the alternative representation where **external nodes are represented by a special dummy node**.

Traversals Using the Linked Representation (cont'd)

- Let us define the folowing enumeration type:

```
typedef enum {PreOrder, InOrder, PostOrder} OrderOfTraversal;
```

- We can now write a general recursive function to perform traversals in the various traversal orders.

Generalized Recursive Traversal Function

```
void Traverse(TreeNode *T, OrderOfTraversal TraversalOrder)
{
     if (T!=NULL){
        if (TraversalOrder==PreOrder){
           Visit(T);
           Traverse(T->LLink, PreOrder);
           Traverse(T->RLink, PreOrder);
        } else if (TraversalOrder==InOrder){
           Traverse(T->LLink, InOrder);
           Visit(T);
           Traverse(T->RLink, InOrder);
       } else if (TraversalOrder==PostOrder){
           Traverse(T->LLink, PostOrder);
           Traverse(T->RLink, PostOrder);
           Visit(T);
      }
   }
}
```

Generalized Recursive Traversal Function (cont'd)

```
void Visit(TreeNode *T)
{
     printf("%c\n", T->Symbol);
}
```

The Main Program

```
#include <stdio.h>
#include <stdlib.h>

typedef struct NodeTag{
                char Symbol;
                struct NodeTag *LLink;
                struct NodeTag *RLink;
          } TreeNode;

typedef enum {PreOrder, InOrder, PostOrder} OrderOfTraversal;

/* code for Visit */

/* code for Traverse */

int main(void)
{
    TreeNode *T;

    /* code to construct a binary tree to which T points */

    Traverse(T, PreOrder);
}
```

Using a Stack

- We can use the linked representation of binary trees and the stack ADT to write **non-recursive** traversal functions.
- A stack is used to hold pointers to subtrees awaiting further traversal.

PreOrder Traversal of an Expression Tree Using a Stack

```
#include <stdio.h>
#include “StackInterface.h”

void PreOrderTraversal(TreeNode *T)
{
      Stack S;
      TreeNode *N;

      InitializeStack(&S);
      Push(T,&S);

      while (!Empty(&S)){
            Pop(&S, &N);

             if (N!=NULL){
                  printf(“%c\n”, N->Symbol);
                  Push(N->RLink, &S);
                  Push(N->LLink, &S);
              }
       }
}
```

Comments

- Note that the stack in the previous function contains pointers to `TreeNode`.
- Therefore, to have a working program with our earlier stack implementations, we need to change the file "`StackTypes.h`" as we show on the next slide.

The Stack Data Types

```
/* This is the new file StackTypes.h   */

#define MAXSTACKSIZE 100

typedef struct NodeTag{
                char Symbol;
                struct NodeTag *LLink;
                struct NodeTag *RLink;
          } TreeNode;

typedef TreeNode *ItemType;


typedef struct{
           int Count;
           ItemType Items[MAXSTACKSIZE];
        } Stack;
```

The Main Program

```
#include <stdio.h>
#include <stdlib.h>
#include "StackInterface.h"

/* code for PreOrderTraversal */

int main(void)
{
    TreeNode *T;

    /* code to construct a binary tree to which T points*/

    PreOrderTraversal(T);
}
```

Using a Queue

- We can use the linked representation of binary trees and the ADT Queue to write a non-recursive function that prints the nodes of the tree in **level order.**

Level Order Binary Tree Traversal Using Queues

```
#include <stdio.h>
#include “QueueInterface.h”

void LevelOrderTraversal(TreeNode *T)
{
    Queue Q;
    TreeNode *N;

     InitializeQueue(&Q);
     Insert(T, &Q);
     while (!Empty(&Q)){
          Remove(&Q, &N);
          if (N!=NULL){
               printf(“%c\n”, N->Symbol);
               Insert(N->LLink, &Q);
               Insert(N->RLink, &Q);
          }
     }
}
```

Comments

- Note that the queue in the previous function contains pointers to `TreeNode`.
- Therefore, to have a working program with our earlier queue implementations, we need to change the file "`QueueTypes.h`" as we show on the next slide.

The Queue Data Types

```
/* This is the new file QueueTypes.h */

#define MAXQUEUESIZE 100

typedef struct NodeTag{
                char Symbol;
                struct NodeTag *LLink;
                struct NodeTag *RLink;
          } TreeNode;

typedef TreeNode *ItemType;

typedef struct {
           int Count;
           int Front;
           int Rear;
           ItemType Items[MAXQUEUESIZE];
         } Queue;
```

The Main Program

```
#include <stdio.h>
#include <stdlib.h>
#include “QueueInterface.h”

/* code for LevelOrderTraversal */

int main(void)
{
    TreeNode *T;

    /* code to construct a binary tree to which T points*/

    LevelOrderTraversal(T);
}
```

The Abstract Data Type Binary Tree

- We can now define the ADT **Binary Tree** with the following operations:
  - **Create**: create an empty binary tree.
  - **IsEmpty**: return true if the tree is empty, otherwise return false.
  - **MakeTree(*Root*, *Left*, *Right*):** create a binary tree with ***Root*** as the root element and ***Left*** (resp. ***Right***) as the left (resp. right) subtree.
  - **Delete**: delete the tree by freeing all its nodes.
  - **PreOrder, InOrder, PostOrder, LevelOrder**: traverse the tree and visit its nodes in the respective order.
  - **Print**: print the tree using an intuitive representation
  - **Height**: return the height of the tree.
  - **Size**: return the number of elements in the tree.

The ADT Binary Tree (cont'd)

- For some of the operations, we have already shown how to implement them.
- The implementation of the remaining operations is left as an exercise.

Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C*.
  - Chapter 9. Sections 9.1 to 9.6.
- R. Sedgewick. *Αλγόριθμοι σε C*.

  Κεφ. 5 και 9.
- The formal propositions we have seen appear in the following book:
  - M. T. Goodrich, R. Tamassia and D. Mount. *Data Structures and Algorithms in C++.* 2nd edition. John Wiley and Sons, 2011.