Trees-I

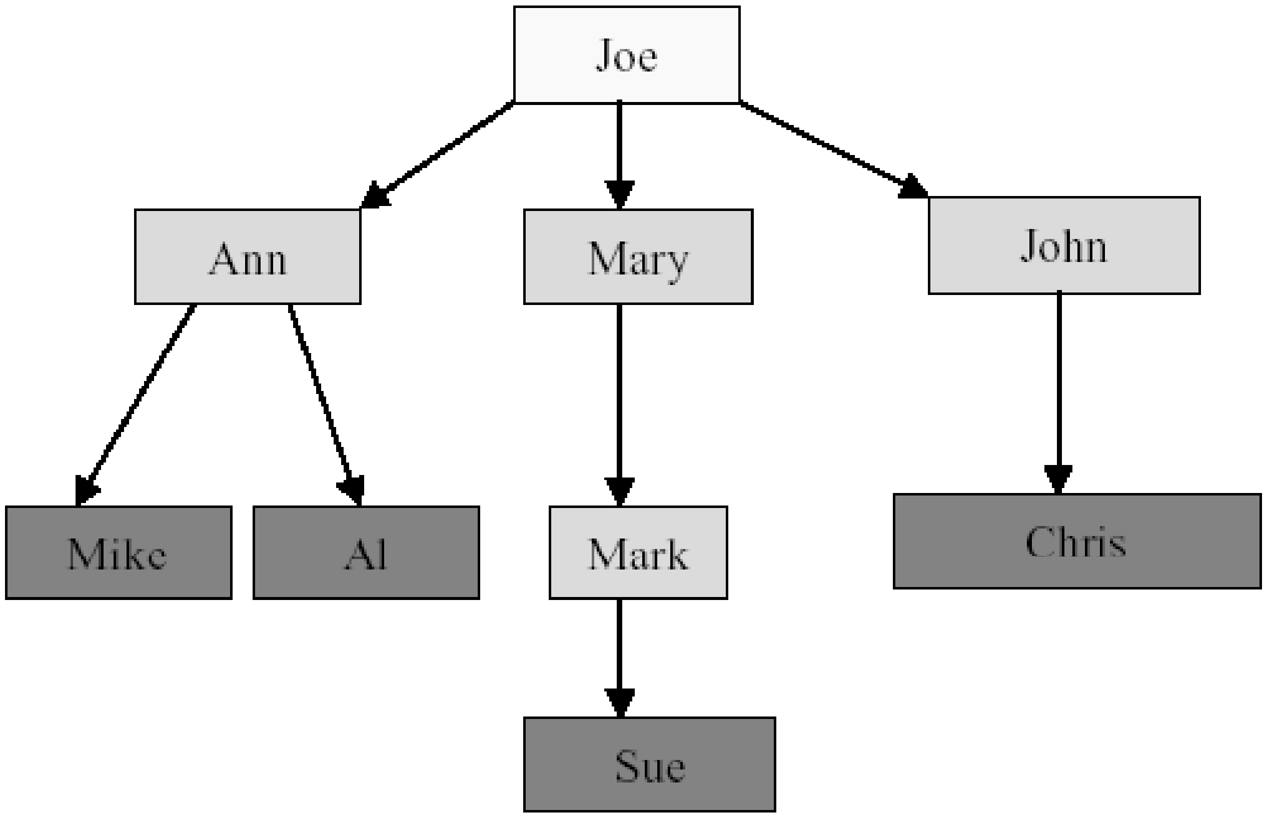
Linear Lists and Trees

- Linear lists are useful for serially ordered data
  - $(e_1, e_2, e_3, \ldots, e_n)$
  - Days of week
  - Months in a year
  - Students in a class

Link list.

- Trees are useful for hierarchically ordered data
  - Joe's descendants
  - Corporate structure
  - Government Subdivisions
  - Software structure

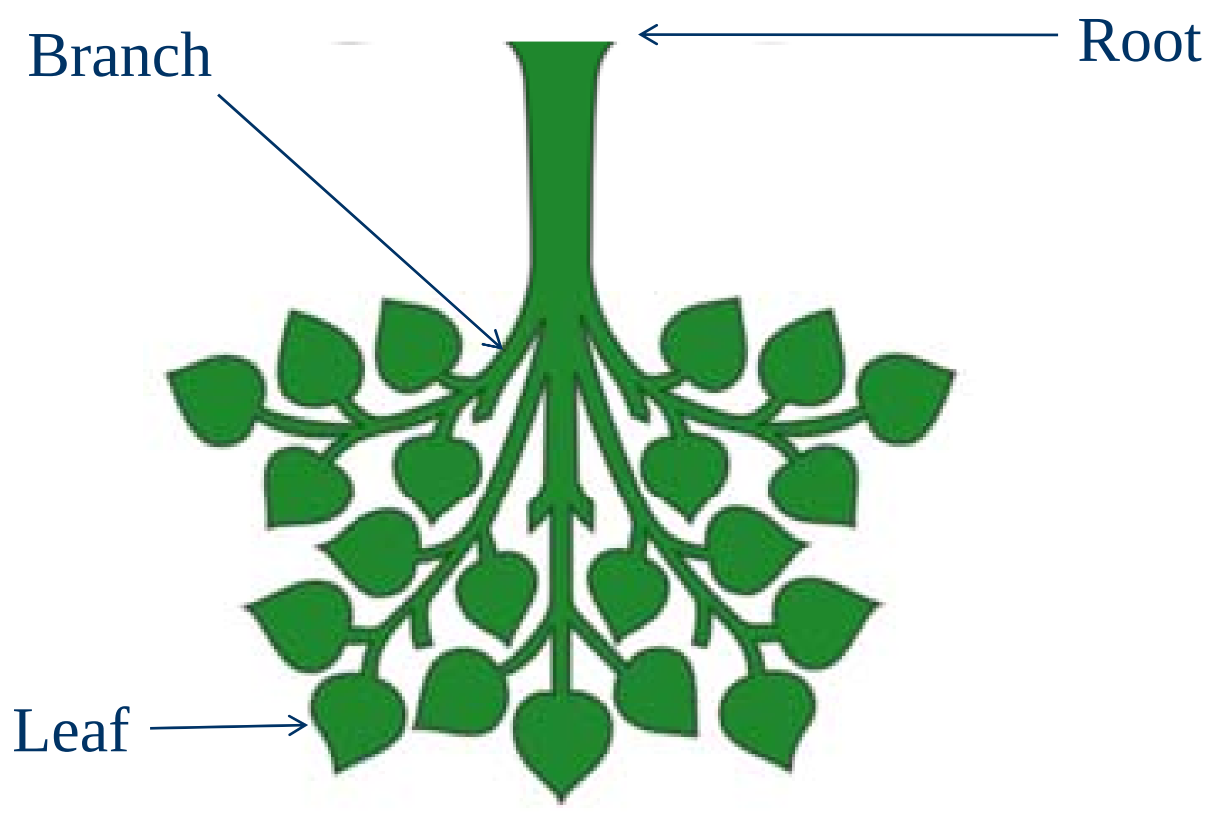
Joe's Descendants



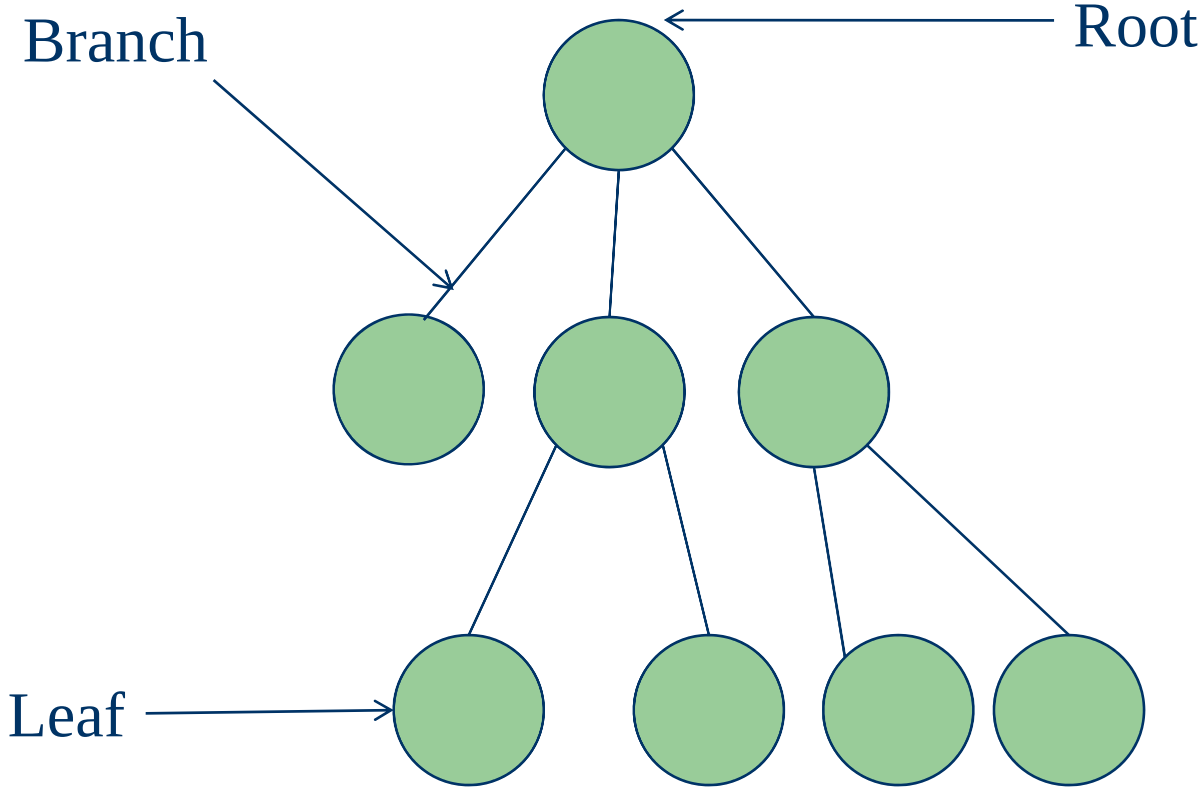
What are other examples of hierarchically ordered data?

Real Tree vs Tree Data Structure

Real Tree vs Tree Data Structure



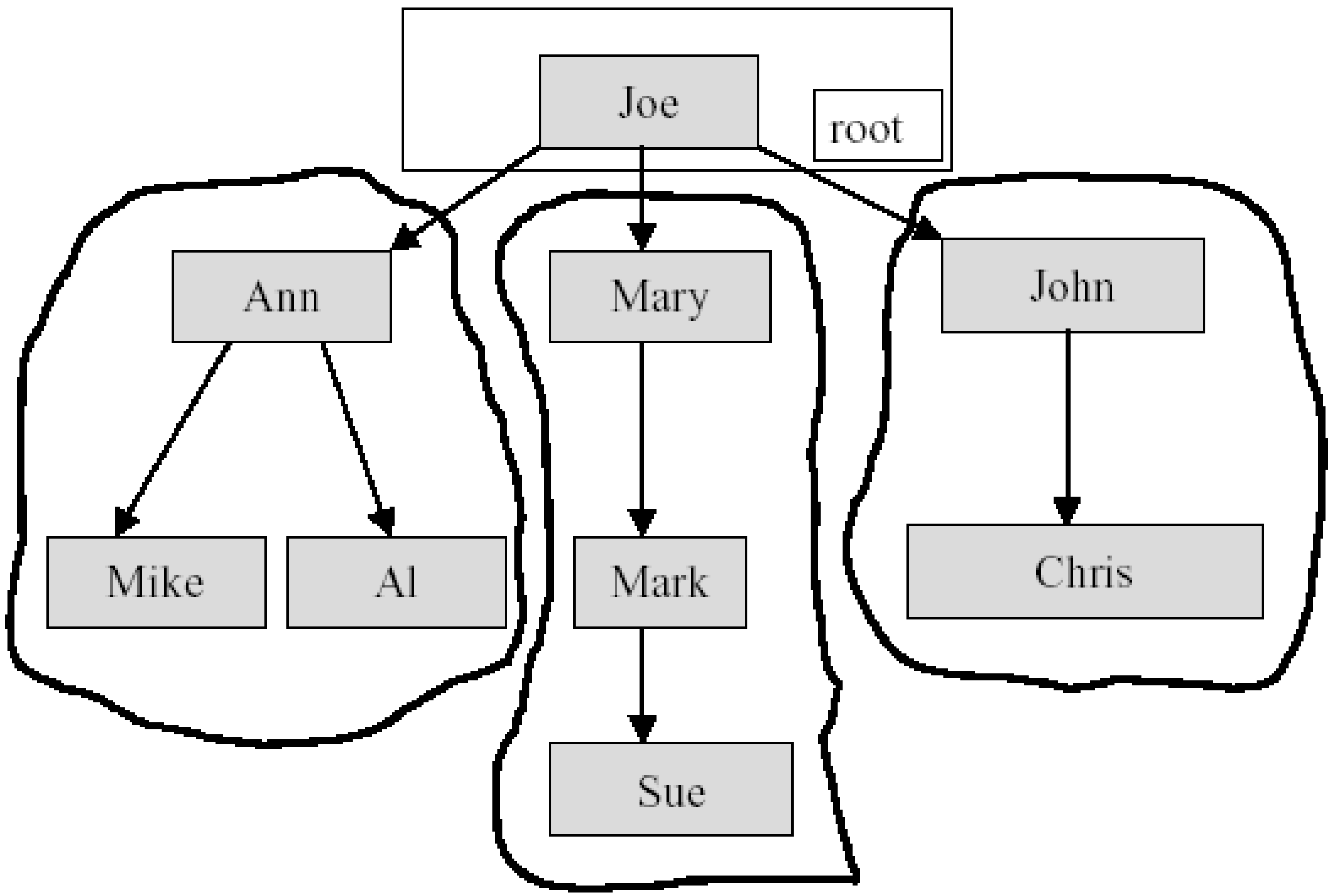
Real Tree vs Tree Data Structure



Definition of Tree

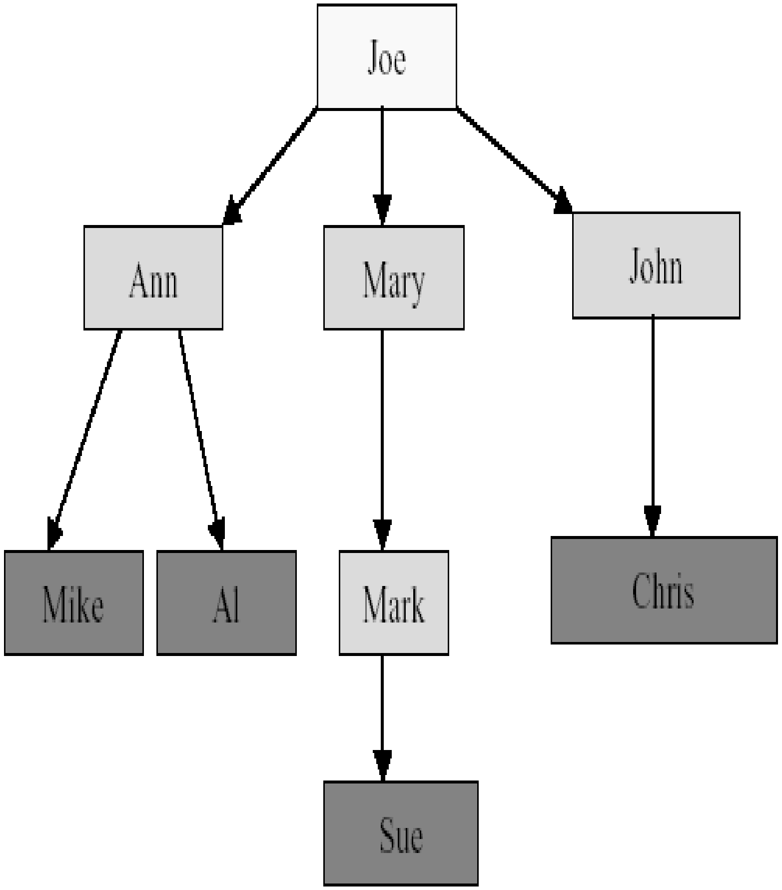
- A tree *t* is a finite nonempty set of elements
- One of these elements is called the root
- The remaining elements, if any, are partitioned into trees, which are called the subtrees of *t*.

Subtrees



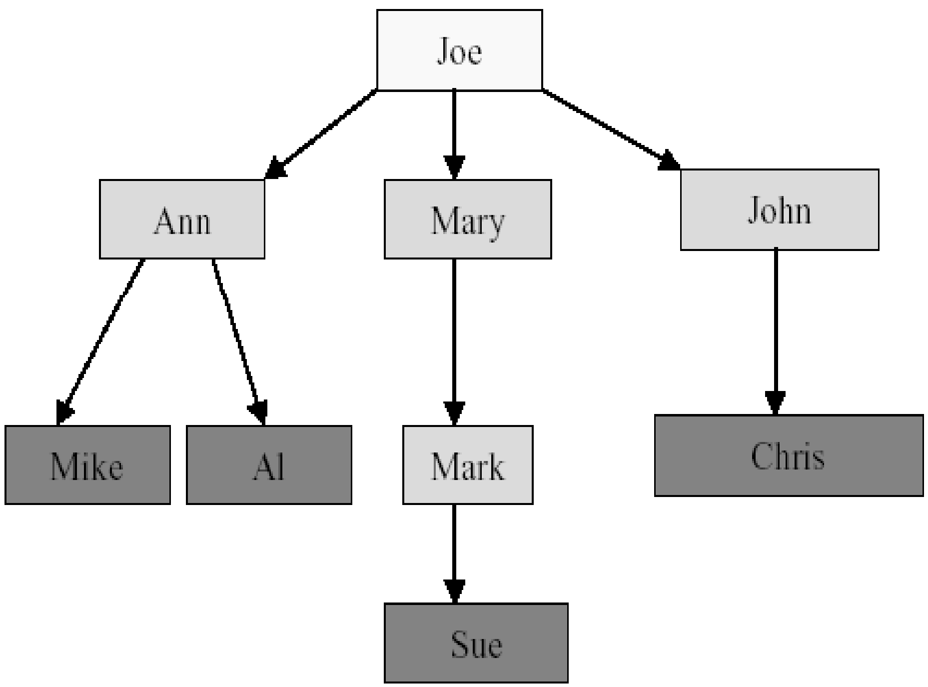
Tree Terminology

- The element at the top of the hierarchy is the **root**.
- Elements next in the hierarchy are the **children** of the root.
- Elements next in the hierarchy are the **grandchildren** of the root, and so on.
- Elements at the lowest level of the hierarchy are the **leaves**.



Other Definitions

- Leaves, Parent, Grandparent, Siblings, Ancestors, Descendents



Leaves = {Mike,AI,Sue,Chris}

Parent(Mary) = Joe

Grandparent(Sue) = Mary

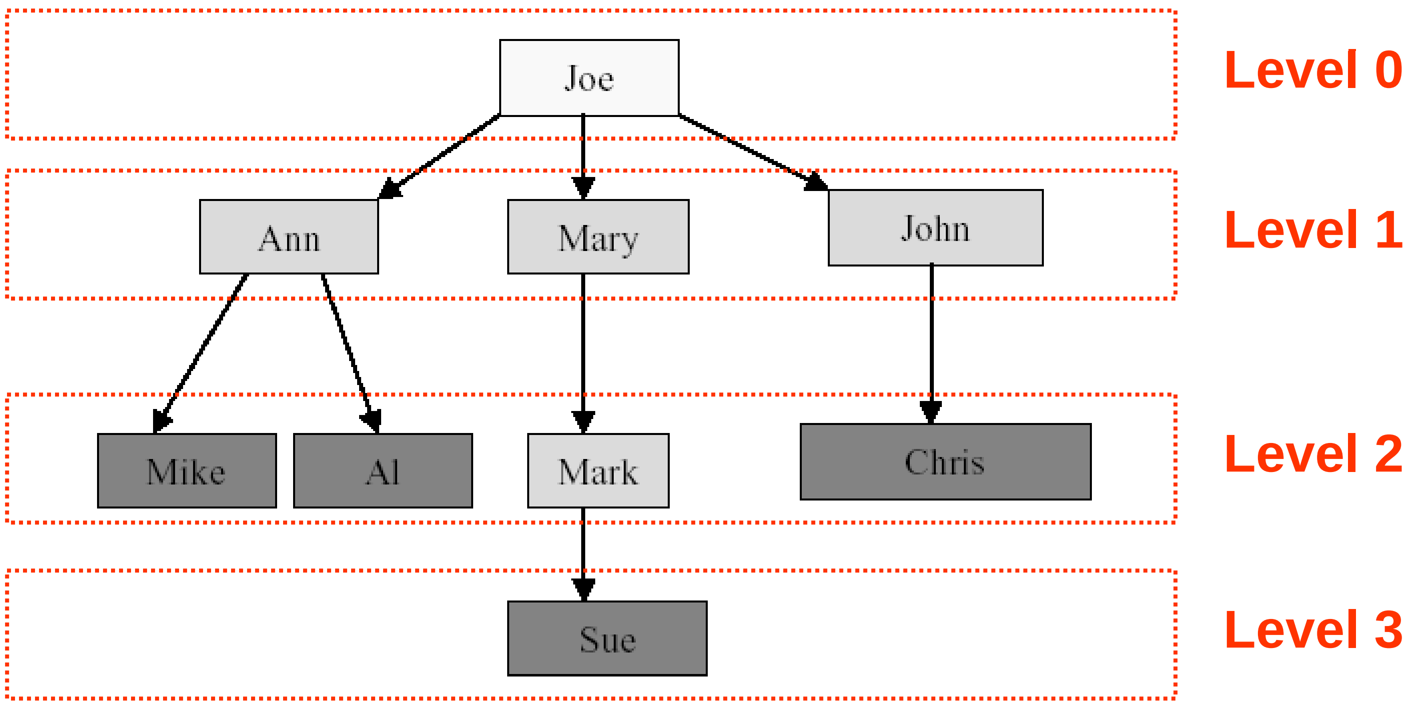
Siblings(Mary) = {Ann,John}

Ancestors(Mike) = {Ann,Joe}

Descendents(Mary)={Mark,Sue}

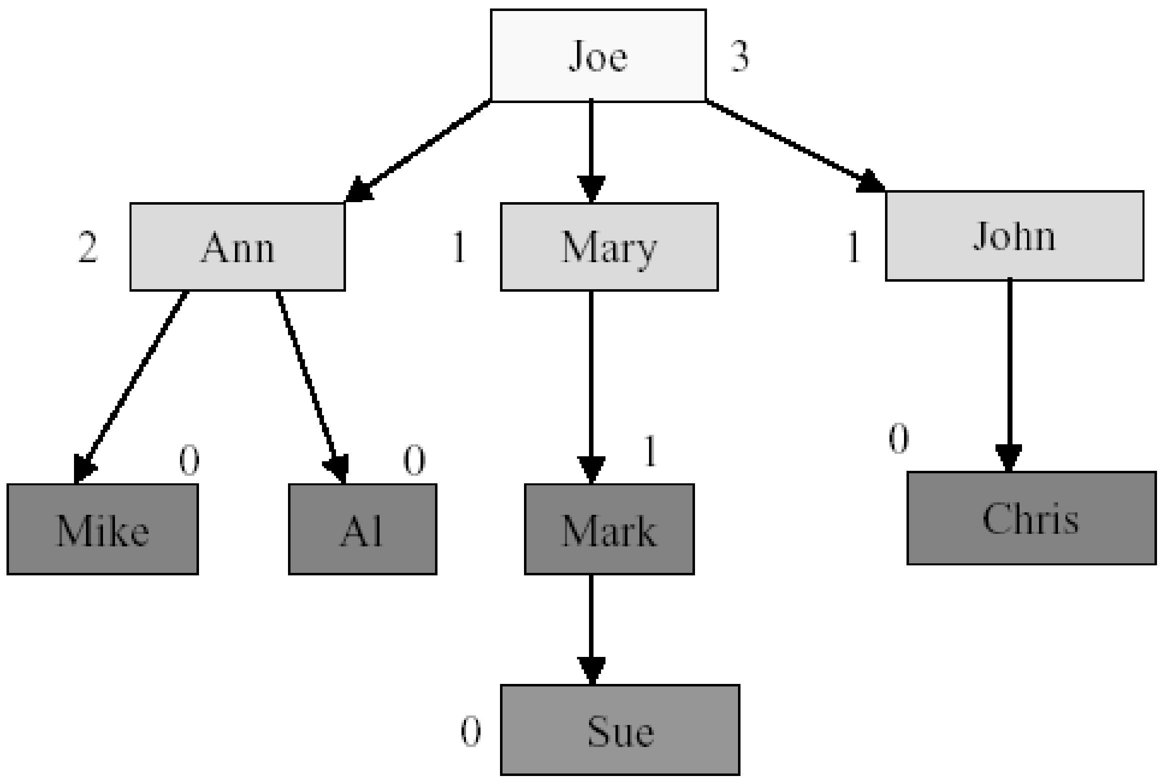
Levels and Height

- Root is at level 0 and its children are at level 1.
- Height = depth = maximum level index



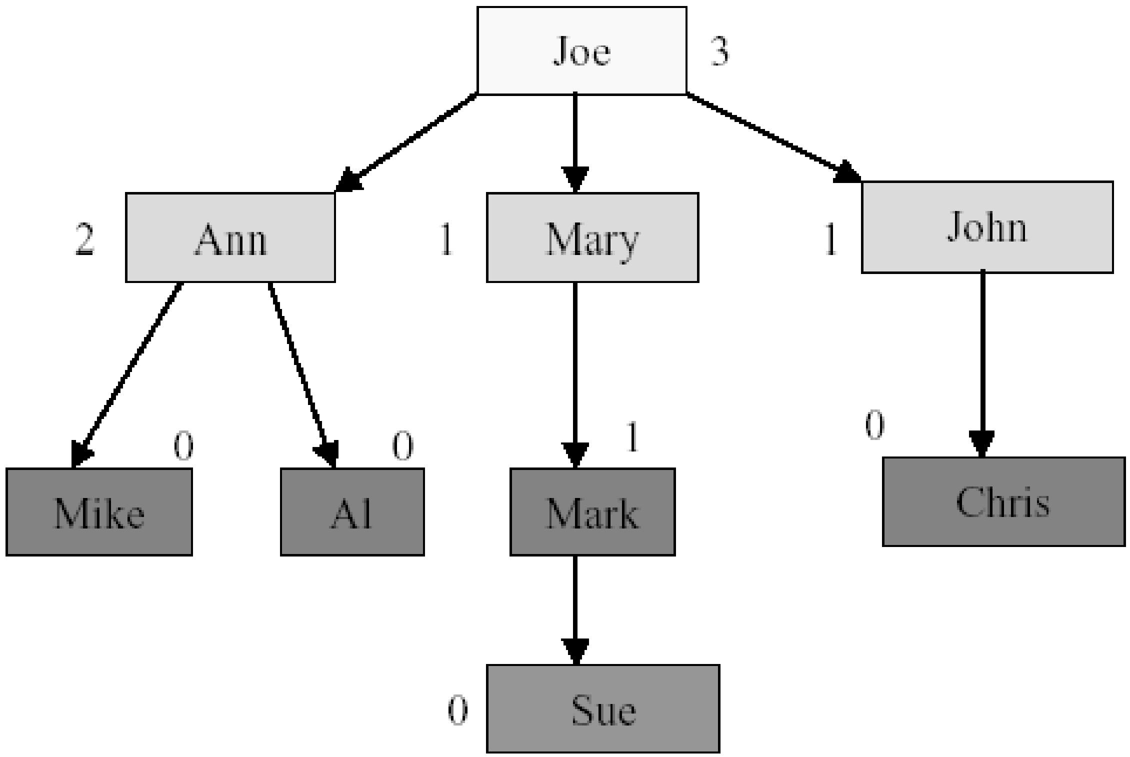
Node Degree

- Node degree is the number of children it has



Tree Degree

- Tree degree is the maximum of node degrees

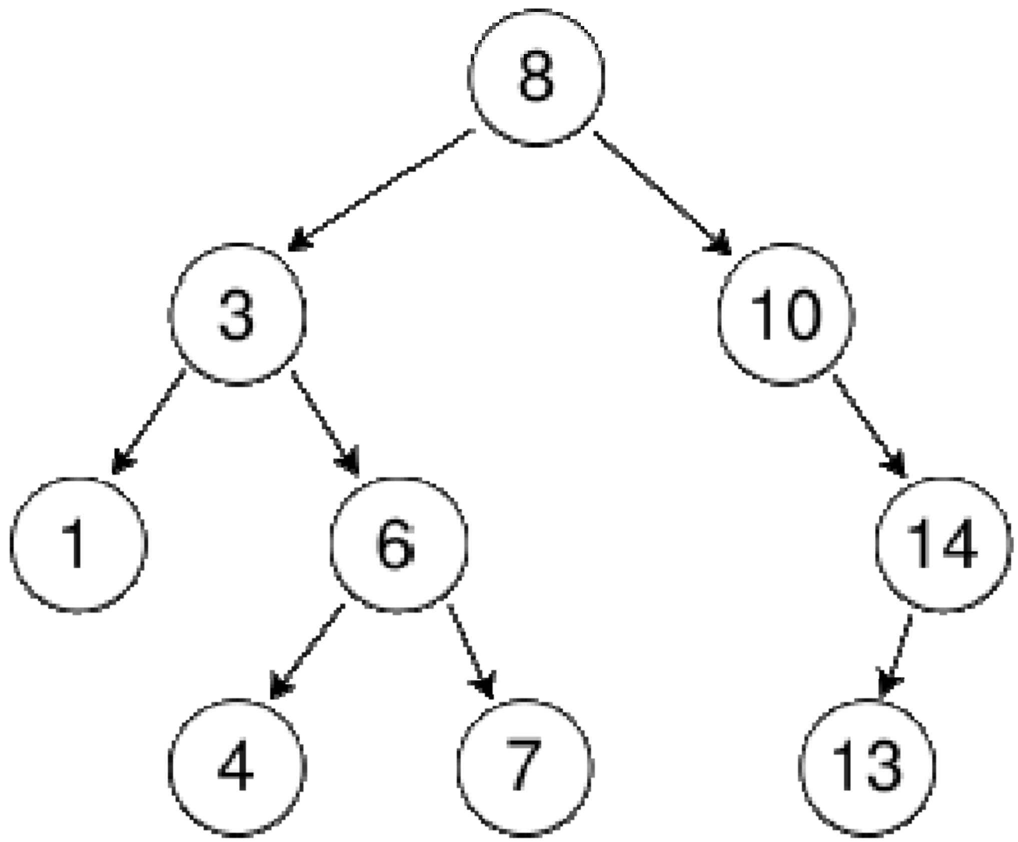


tree degree = 3

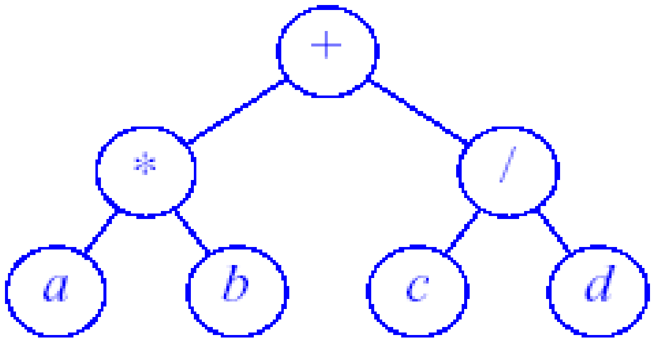
Binary Tree

- A finite (possibly empty) collection of elements
- A nonempty binary tree has a root element and the remaining elements (if any) are partitioned into two binary trees
- They are called the left and right subtrees of the binary tree
- All the nodes in a binary tree have 0, 1 or 2 child/children.

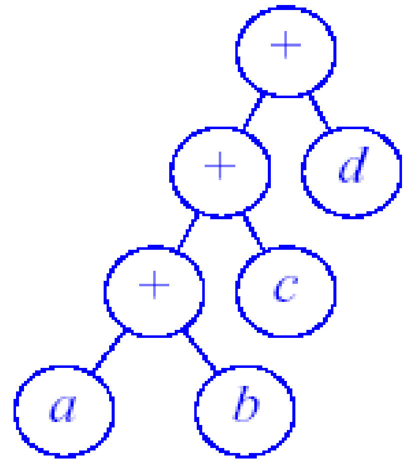
Binary Tree (Example)



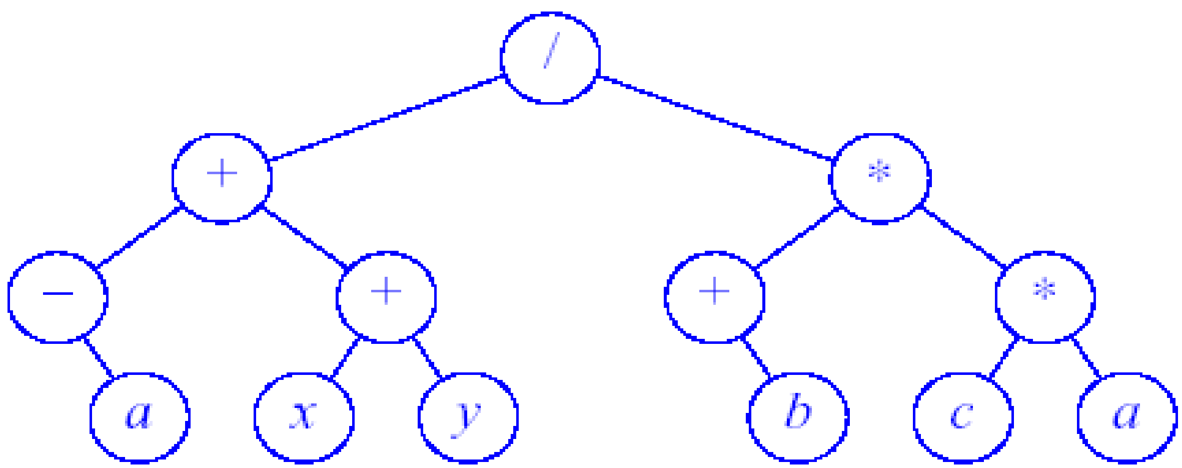
Binary Tree for Expressions



(a) $(a * b) + (c / d)$



(b) $((a + b) + c) + d)$

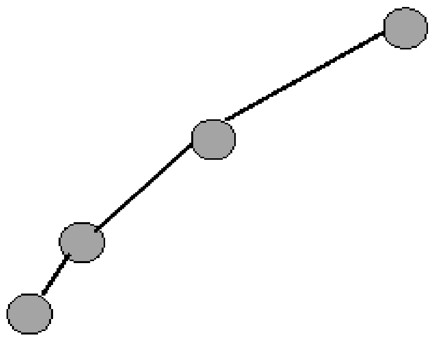


(c) $((-a) + (x + y)) / ((+b) * (c * a))$

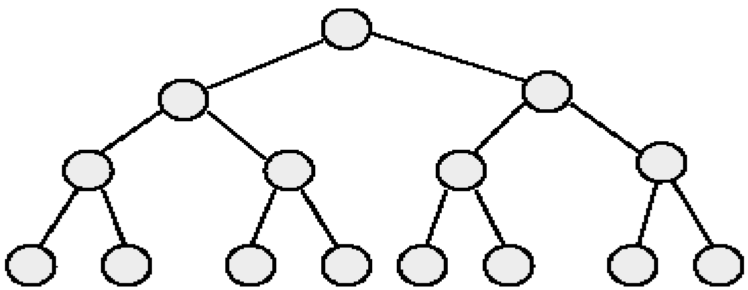
Binary Tree Properties

1. Every binary tree with n elements, n > 0, has exactly n-1 edges.

2. A binary tree of height h, h >= 0, has <u>at least</u> $h+1$ and <u>at most</u> 2^{h+1}-1 elements in it.



minimum number of elements



maximum number of elements

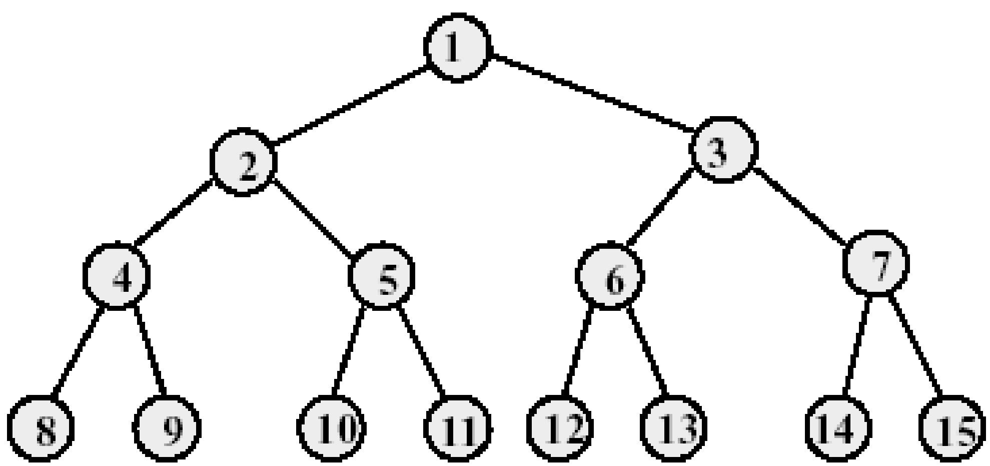
Binary Tree Properties

3. The height of a binary tree that contains n elements, $n >= 0$, is <u>at least</u> $\lceil (log_2(n+1)) \rceil -1$ and <u>at most</u> $n-1$.

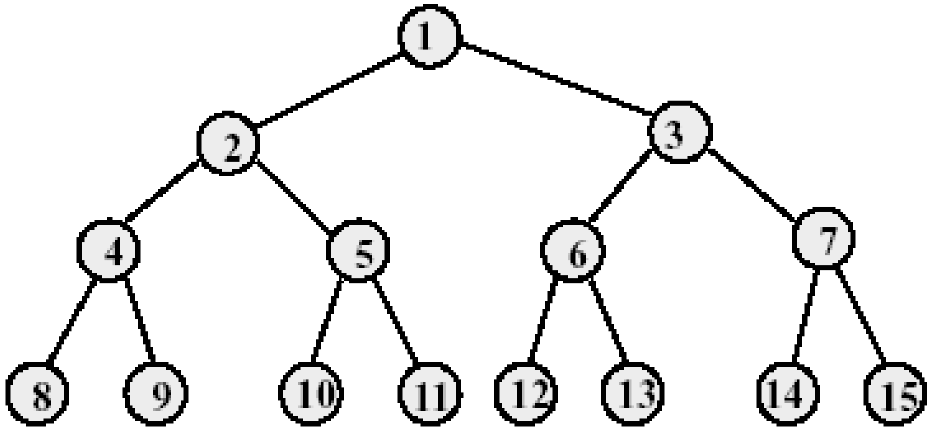
4. For any nonempty binary tree, T, if n0 is the number of leaf nodes and n2 the number of nodes of degree 2, then n0=n2+1.

Full Binary Tree

- A full binary tree of height *h* has exactly $2^{h+!}$-1 nodes.
- Numbering the nodes in a full binary tree
  - Number the nodes 1 through 2^{h+1}-1
  - Number by levels from top to bottom
  - Within a level, number from left to right

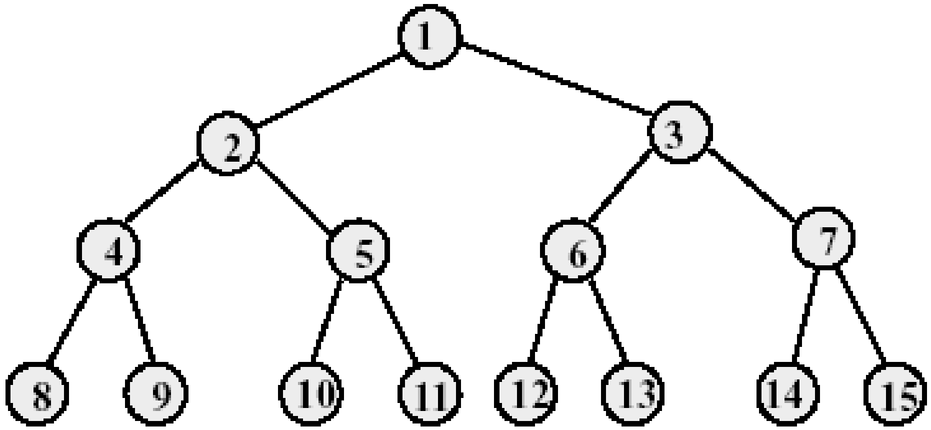


Node Number Property of Full Binary Tree



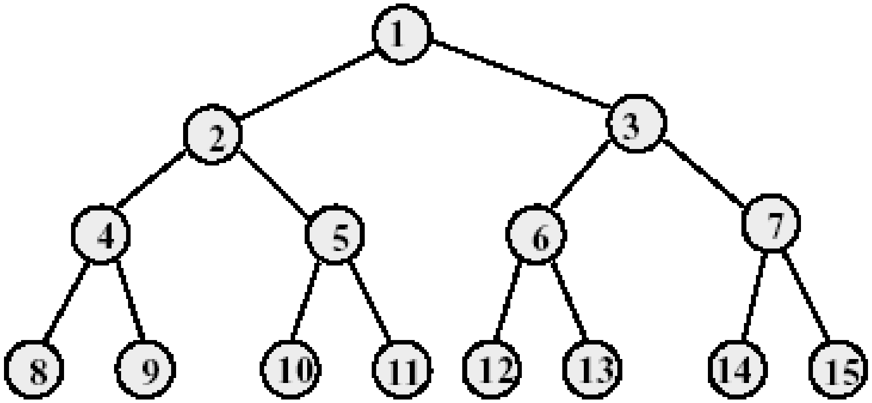
- Parent of node *i* is node $\lfloor (i/2) \rfloor$, unless i = 1
- Node 1 is the root and has no parent

Node Number Property of Full Binary Tree



- Left child of node *i* is node *2i*, unless *2i* > *n*, where *n* is the total number of nodes.
- If *2i* > *n*, node *i* has no left child.

Node Number Property of Full Binary Tree



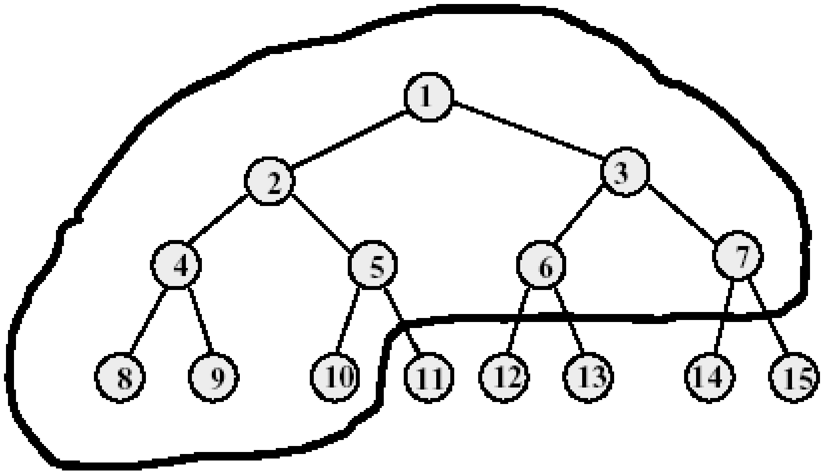
- Right child of node *i* is node $2i+1$, unless $2i+1 > n$, where n is the total number of nodes.
- If $2i+1 > n$, node *i* has no right child.

Complete Binary Tree with n Nodes

- Start with a full binary tree that has at least *n* nodes
- Number the nodes as described earlier.
- The binary tree defined by the nodes numbered 1 through *n* is the *n*-node complete binary tree.
- A full binary tree is a special case of a complete binary tree

- A complete binary tree is a binary tree every level of which has the maximum possible number of nodes except possibly the last level.

Example of Complete Binary Tree



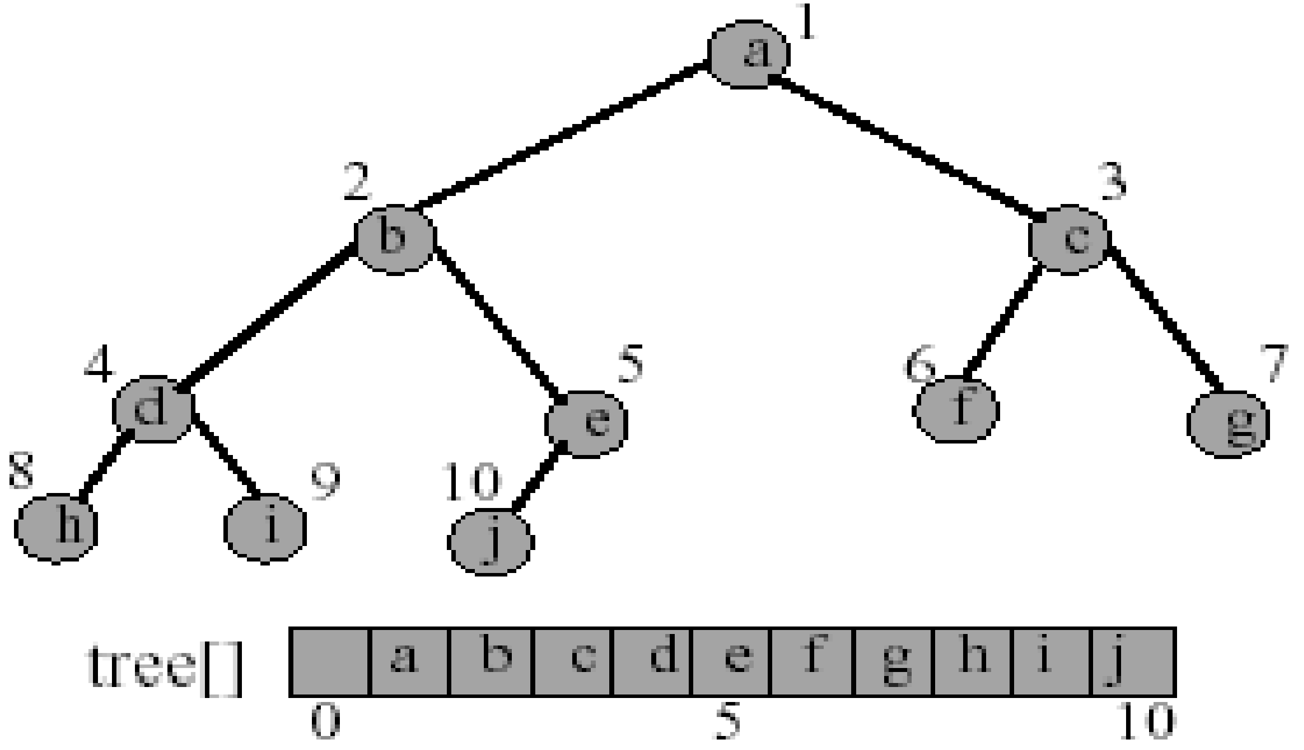
- Complete binary tree with 10 nodes.
- Same node number properties (as in full binary tree) also hold here.

Binary Tree Representation

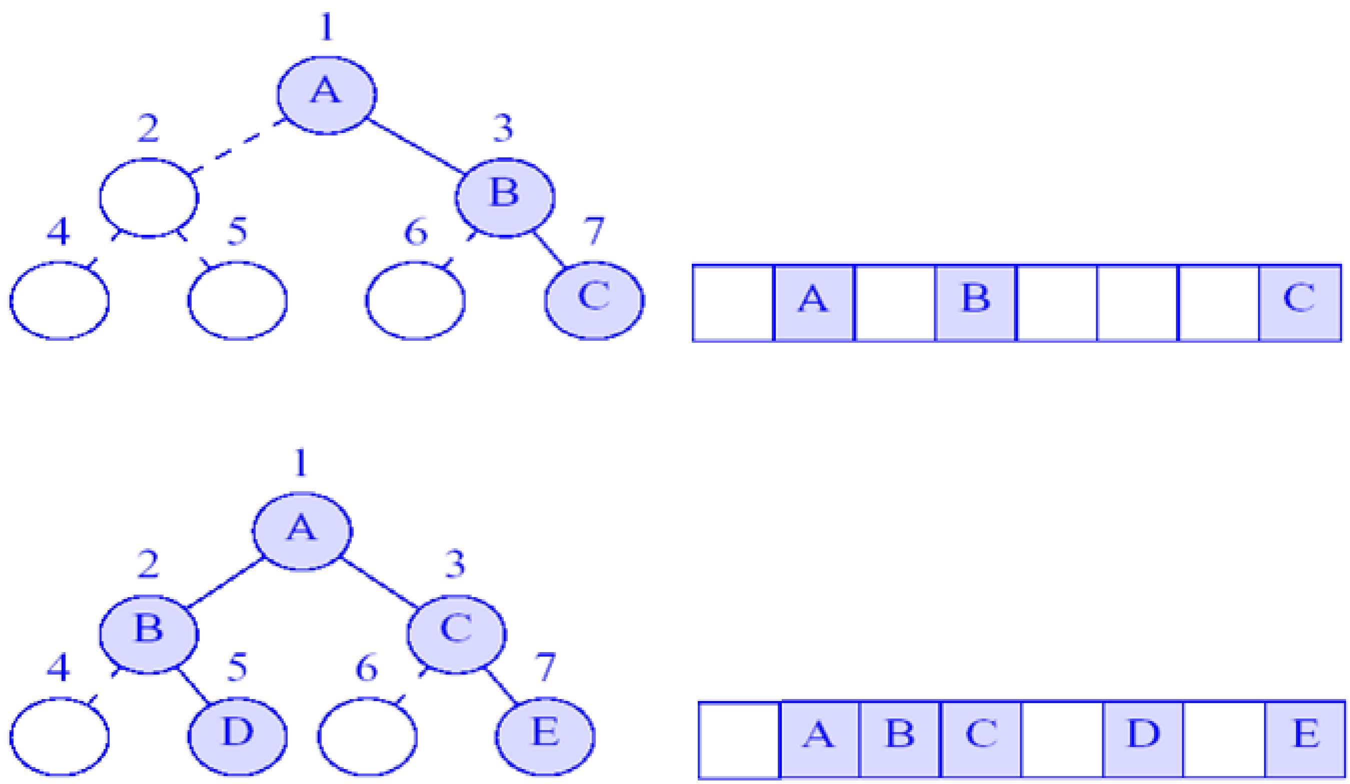
- Array representation
- Linked representation

Array Representation of Binary Tree

- The binary tree is represented in an array by storing each element at the array position corresponding to the number assigned to it.



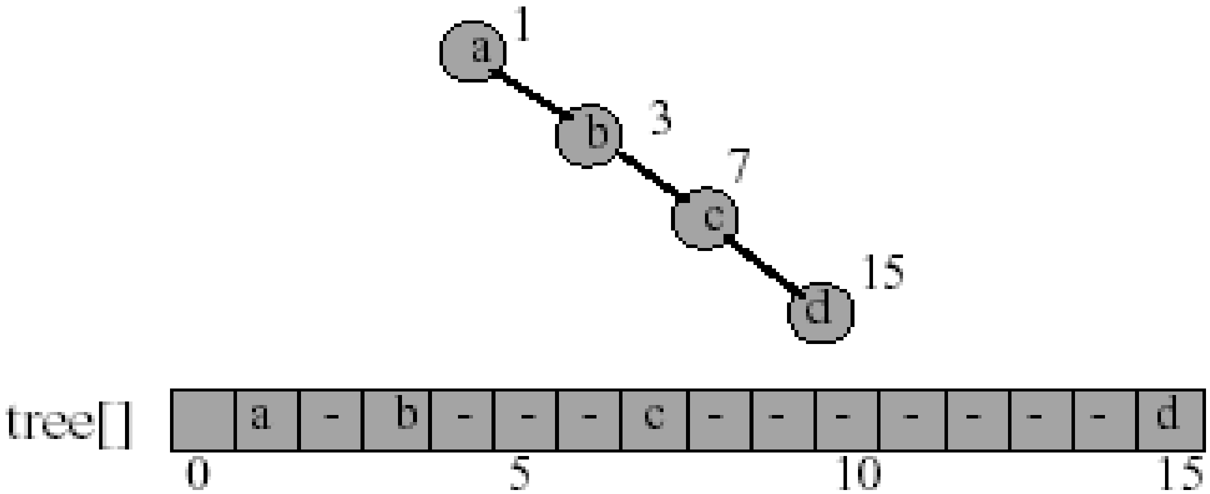
Incomplete Binary Trees



Incomplete binary trees

- Complete binary tree with some missing elements

Right-Skewed Binary Tree



- Right-skewed binary tree wastes the most space
- What about left-skewed binary tree?

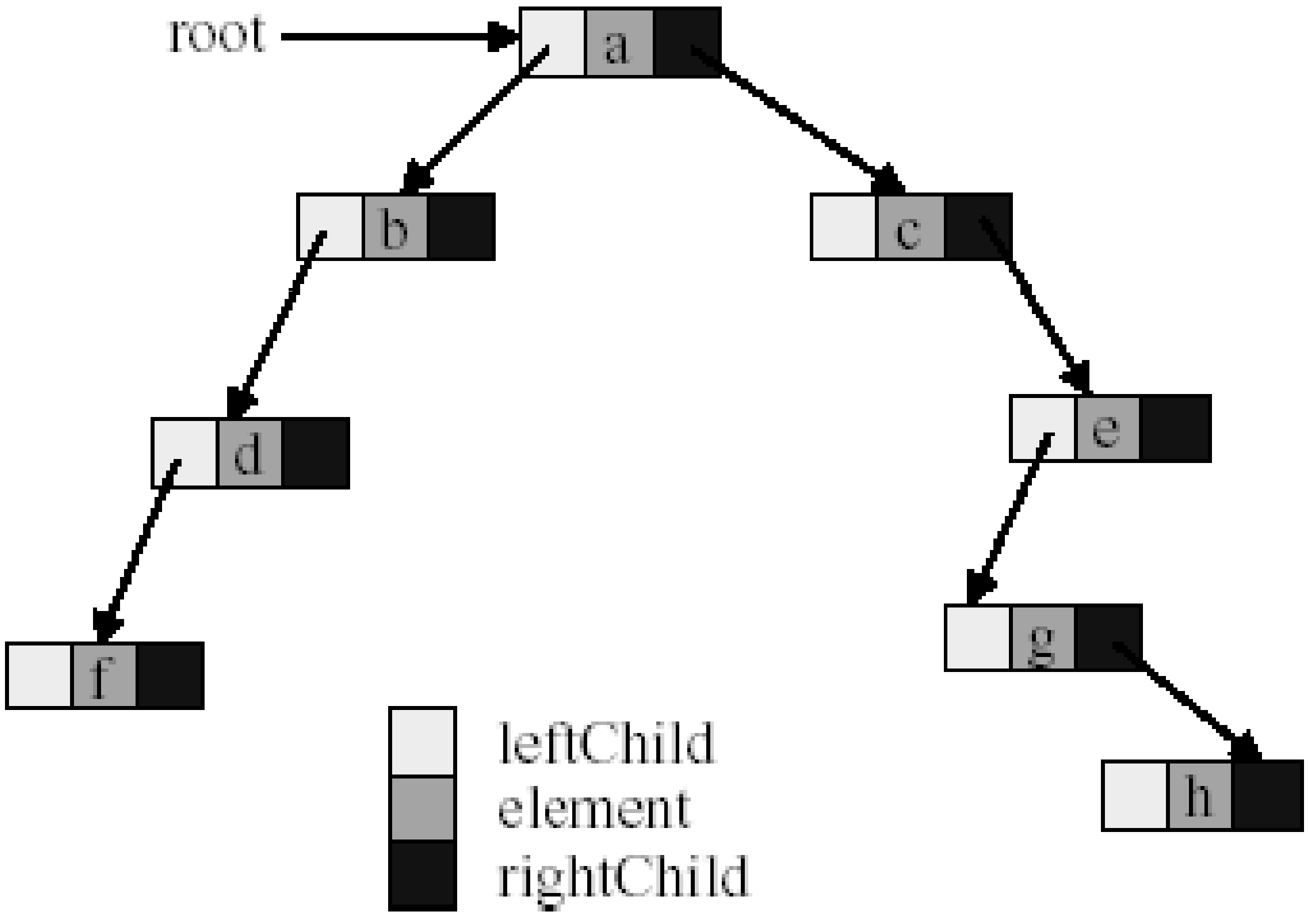
Linked Representation of Binary Tree

- The most popular way to present a binary tree
- Each element is represented by a node that has two link fields (leftChild and rightChild) plus an element field
- Each binary tree node is represented as an structure/object whose data type is **Node**
- The space required by an *n* node binary tree is *n* * sizeof(Node)

Linked Representation of Binary Tree


```
class Node
{
private:
int key;
Node* left;
Node* right;
public:
Node() { key=-1; left=NULL; right=NULL; };
void setKey(int aKey) { key = aKey; };
void setLeft(Node* aLeft) { left = aLeft; };
void setRight(Node* aRight) { right = aRight; };
int Key() { return key; };
Node* Left() { return left; };
Node* Right() { return right; };
};
```

Linked Representation of Binary Tree



Linked Representation of Binary Tree

```
class Tree
{
private:
Node* root;
public:
Tree(){root = NULL;};
~Tree(){freeNode(root);};
Node* Root() { return root; };
...//other methods
...//other methods
void inOrder(Node* n); //inOrder traversal
void preOrder(Node* n); //preOrder traversal
void postOrder(Node* n); //postOrder traversal
private:
void freeNode(Node* nd);
};
```

Linked Representation of Binary Tree

```
void Tree::freeNode(Node* nd)
{
     if ( nd != NULL )
     {
          freeNode(nd->Left());
          freeNode(nd->Right());
          delete nd;
     }
}
```

Common Binary Tree Operations

- Determine the height
- Determine the number of nodes
- Make a copy
- Determine if two binary trees are identical
- Display the binary tree
- Delete a tree
- If it is an expression tree, evaluate the expression
- If it is an expression tree, obtain the parenthesized form of the expression

Binary Tree Traversal

- Many binary tree operations are done by performing a **traversal** of the binary tree

- In a traversal, each element of the binary tree is **visited** exactly once

- During the visit of an element, all actions (make a copy, display, evaluate the operator, etc.) with respect to this element are taken

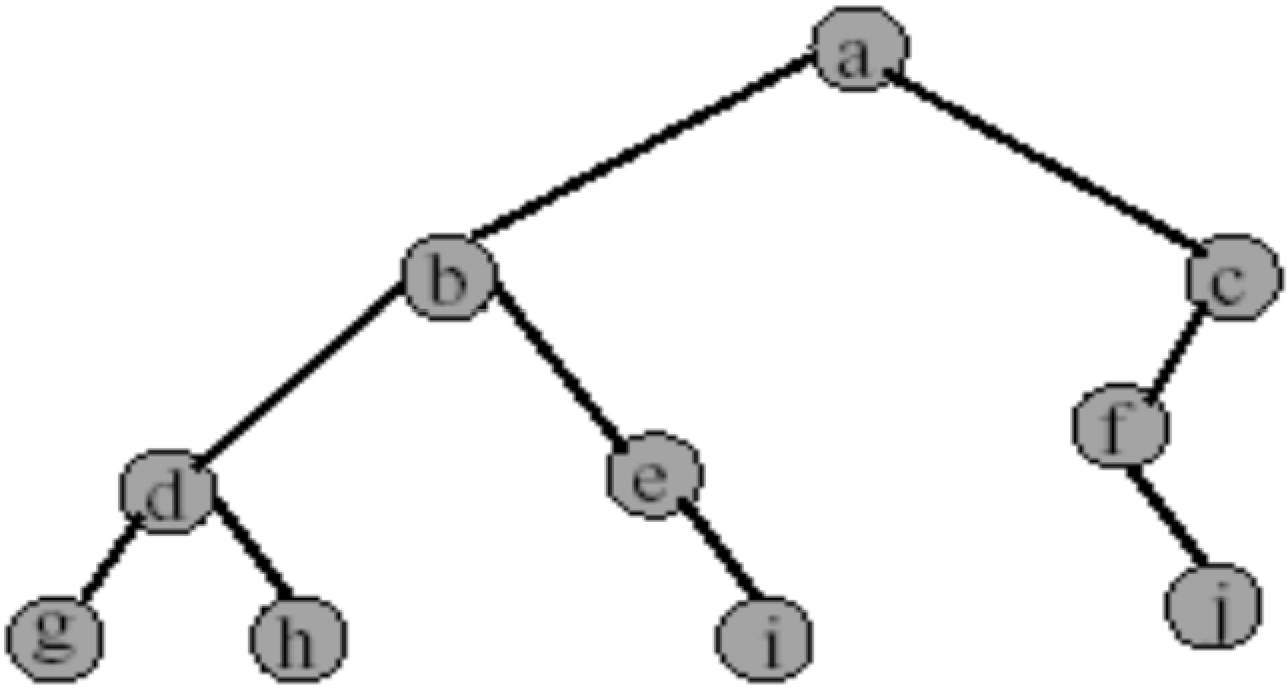
Binary Tree Traversal Methods

- **Preorder**
  - The root of the subtree is processed first before going into the left then right subtree (root, left, right).
- **Inorder**
  - After the complete processing of the left subtree the root is processed followed by the processing of the complete right subtree (left, root, right).
- **Postorder**
  - The root is processed only after the complete processing of the left and right subtree (left, right, root).
- **Level order**
  - The tree is processed by levels. So first all nodes on level i are processed from left to right before the first node of level i+1 is visited

Preorder Traversal

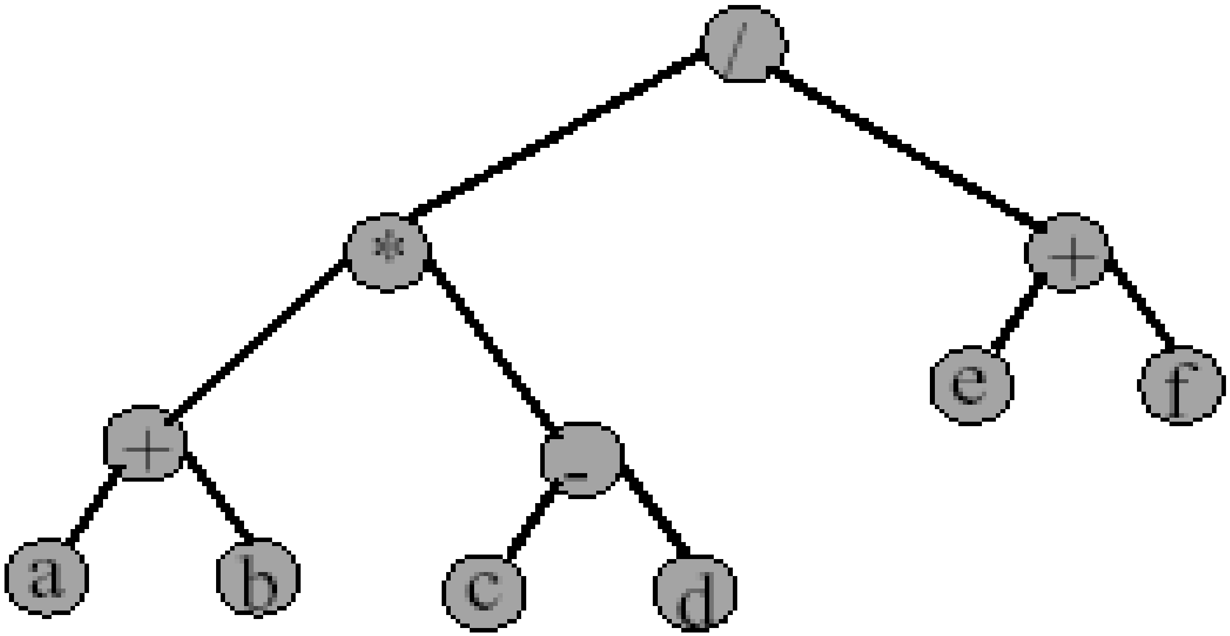
```
void Tree::preOrder(Node* n)
{
     if ( n!=NULL )
     {
          cout << n->Key() << " ";
          preOrder(n->Left());
          preOrder(n->Right());
     }
}
```

Preorder Example (visit = print)



a b d g h e i c f j

Preorder of Expression Tree



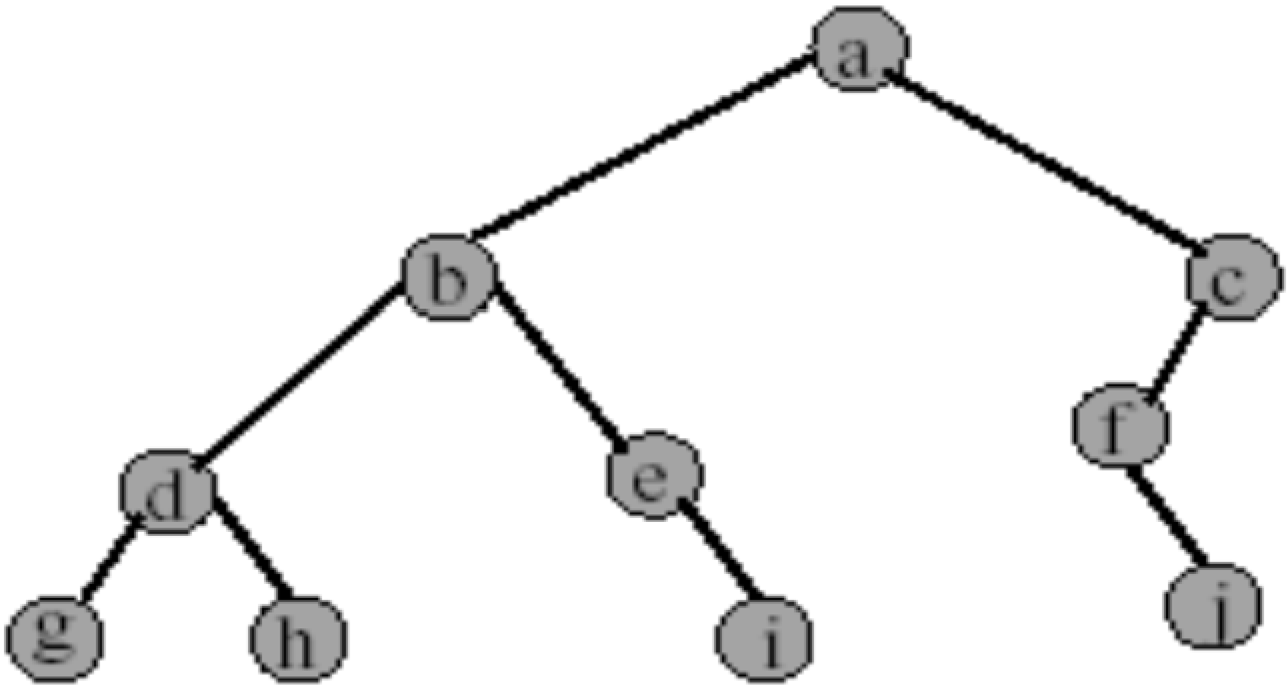
/ * + a b - c d + e f

Gives prefix form of expression.

Inorder Traversal

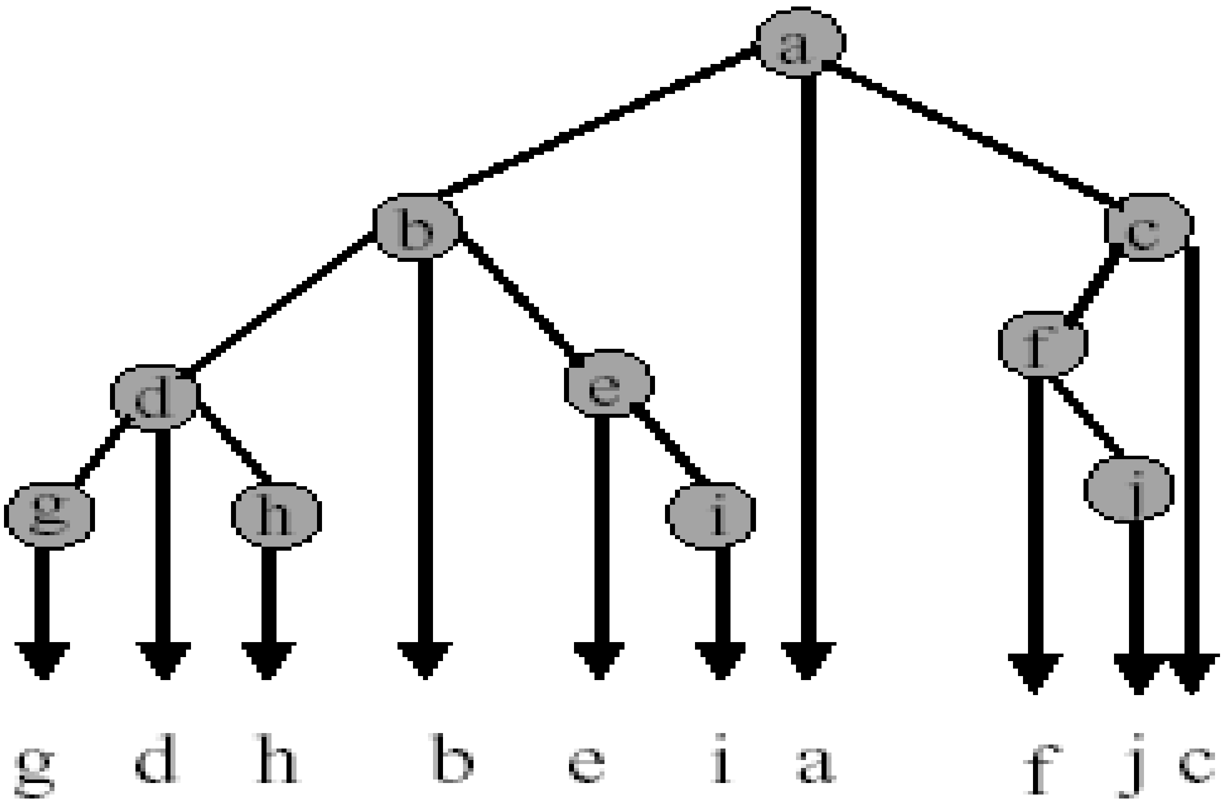
```
void Tree::inOrder(Node* n)
{
      if ( n!=NULL )
      {
            inOrder(n->Left());
            cout << n->Key() << " ";
            inOrder(n->Right());
      }
}
```

Inorder Example (visit = print)

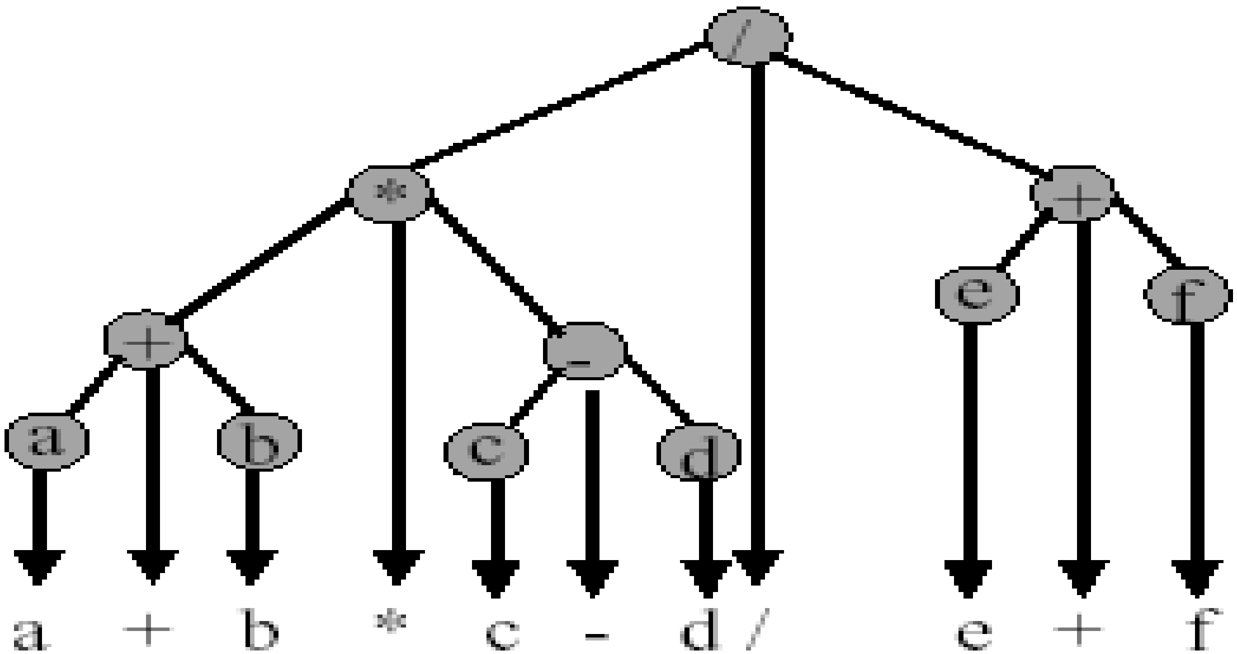


g d h b e i a f j c

Inorder by Projection (Squishing)



Inorder of Expression Tree

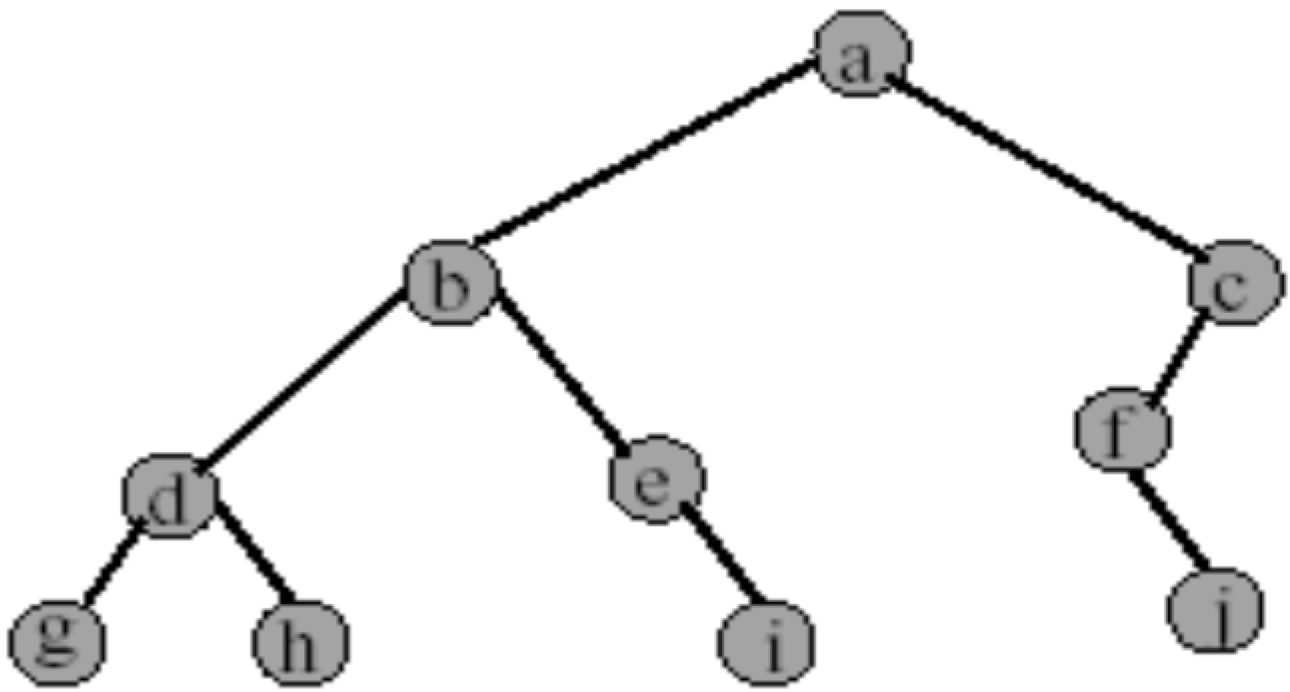


- Gives infix form of expression, which is how we normally write math expressions.

Postorder Traversal

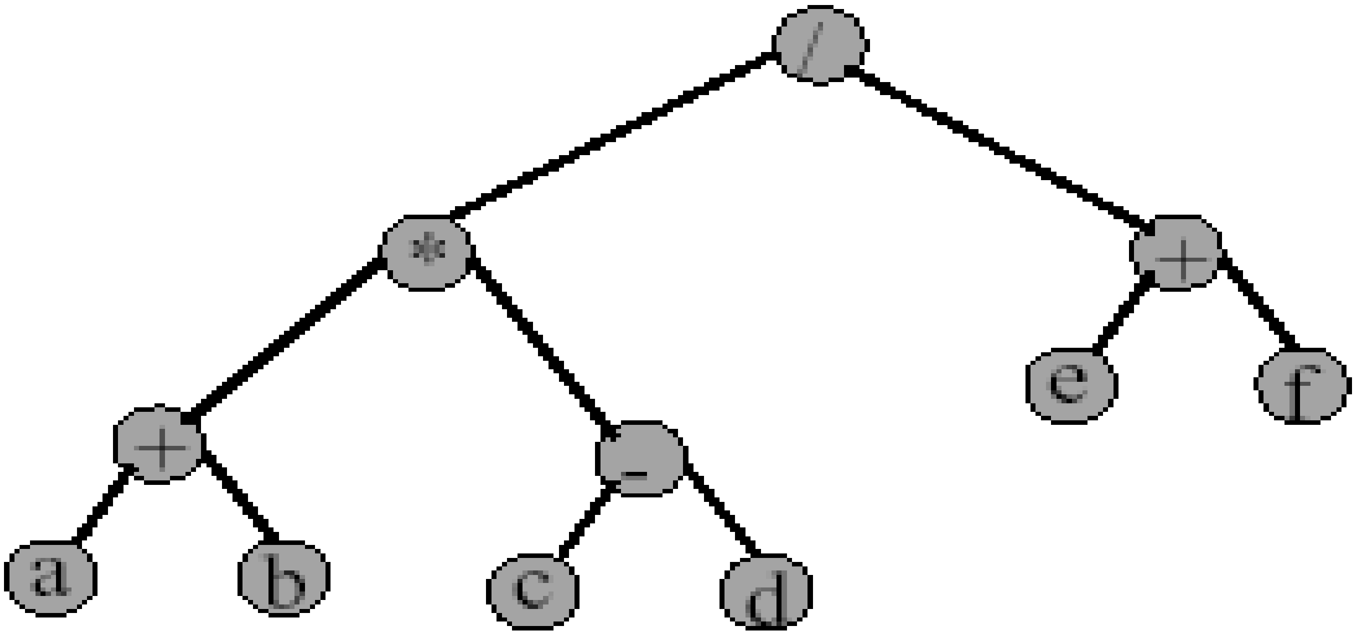
```
void Tree::postOrder(Node* n)
{
     if ( n!=NULL )
     {
          postOrder(n->Left());
          postOrder(n->Right());
          cout << n->Key() << " “;
     }
}
```

Postorder Example (visit = print)



g h d i e b j f c a

Postorder of Expression Tree



a b + c d - * e f + /

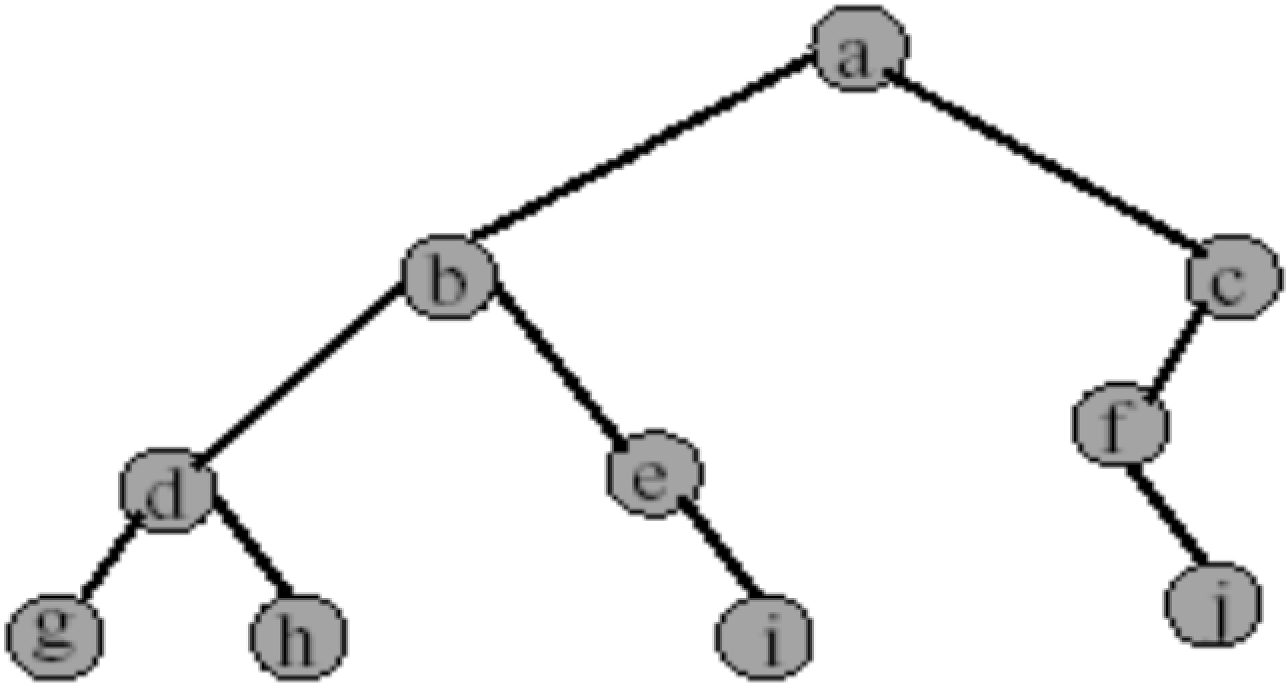
Gives postfix form of expression.

Level Order Traversal

Try to write the code yourself

– Visit all nodes in the i^{th} level before going into $(i+1)^{th}$ level.

Level Order Example (visit = print)



- Add and delete nodes from a queue
- Output: a b c d e f g h i j

Time Complexity

- The time complexity of each of the four traversal algorithm is **O(n)** because each node is visited exactly once.

- **Recurrence relation for preorder, inorder and postorder traversals:**

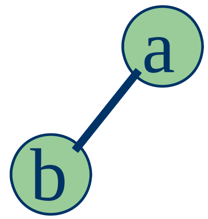
$$T(n) = 2T(n/2) + c$$

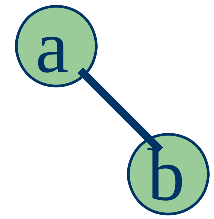
Binary Tree Construction

- Suppose that the elements in a binary tree are distinct.
- Can you construct the binary tree from which a given traversal sequence came?
- When a traversal sequence has more than one element, the binary tree is not uniquely defined.
- Therefore, the tree from which the sequence was obtained cannot be reconstructed uniquely.

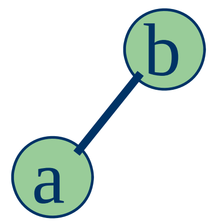
Some Examples

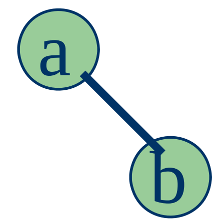
preorder = ab



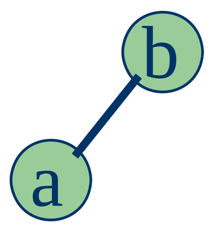


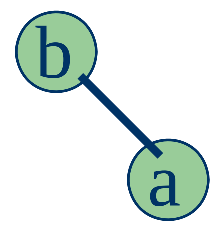
inorder = ab



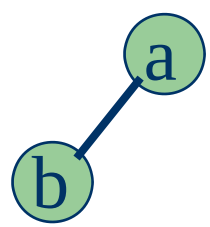


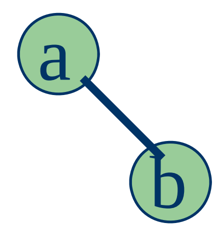
postorder = ab





level order = ab





Binary Tree Construction

- Can you construct the binary tree, given two traversal sequences?
- Depends on which two sequences are given.

Preorder And Postorder

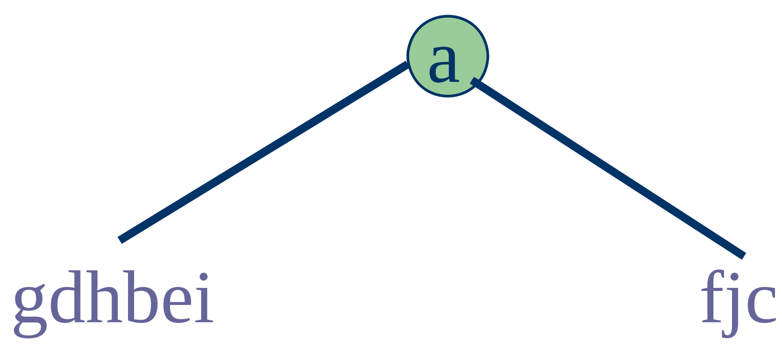
preorder = ab

postorder = ba

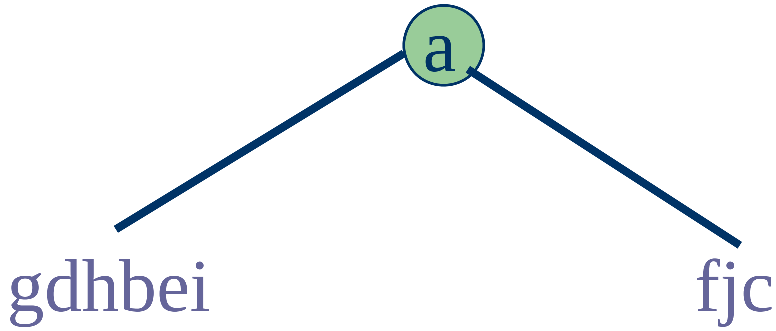
- Preorder and postorder do not uniquely define a binary tree.
- Nor do preorder and level order (same example).
- Nor do postorder and level order (same example).

Inorder And Preorder

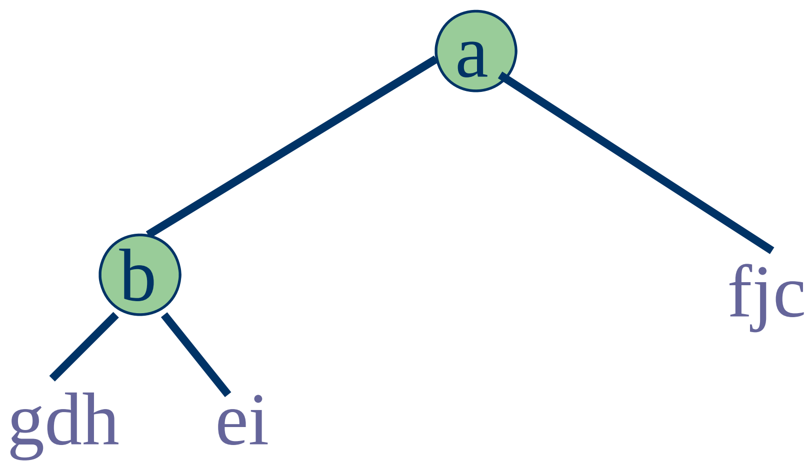
- inorder = g d h b e i a f j c
- preorder = a b d g h e i c f j
- Scan the preorder left to right using the inorder to separate left and right subtrees.
- a is the root of the tree; gdhbei are in the left subtree; fjc are in the right subtree.



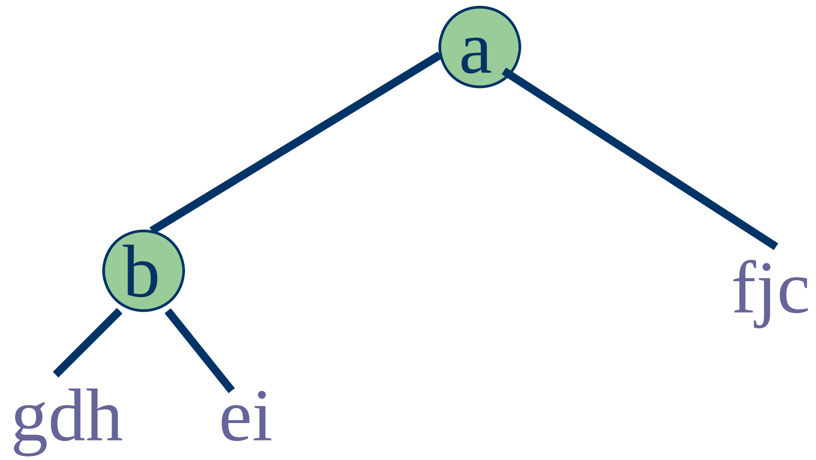
Inorder And Preorder



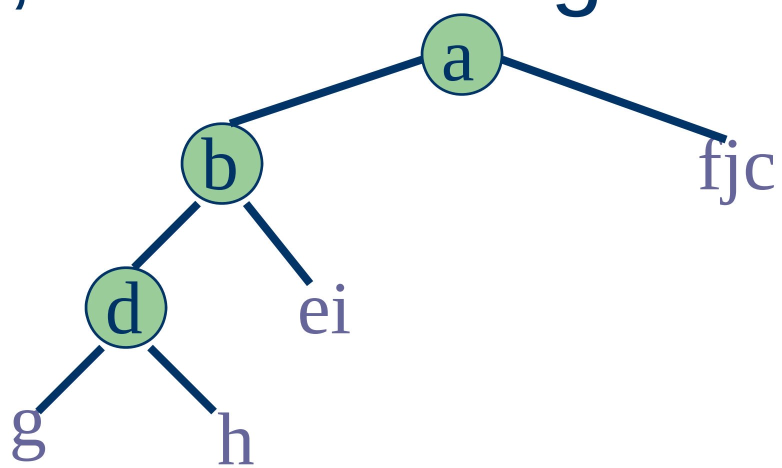
- preorder = b d g h e i c f j
- b is the next root; gdh are in the left subtree; ei are in the right subtree.



Inorder And Preorder



- preorder = d g h e i c f j
- d is the next root; g is in the left sub tree; h is in the right subtree.



Inorder And Postorder

- Scan postorder from right to left using inorder to separate left and right subtrees.
- inorder = g d h b e i a f j c
- postorder = g h d i e b j f c a
- Tree root is a; gdhbei are in left subtree; fjc are in right subtree.

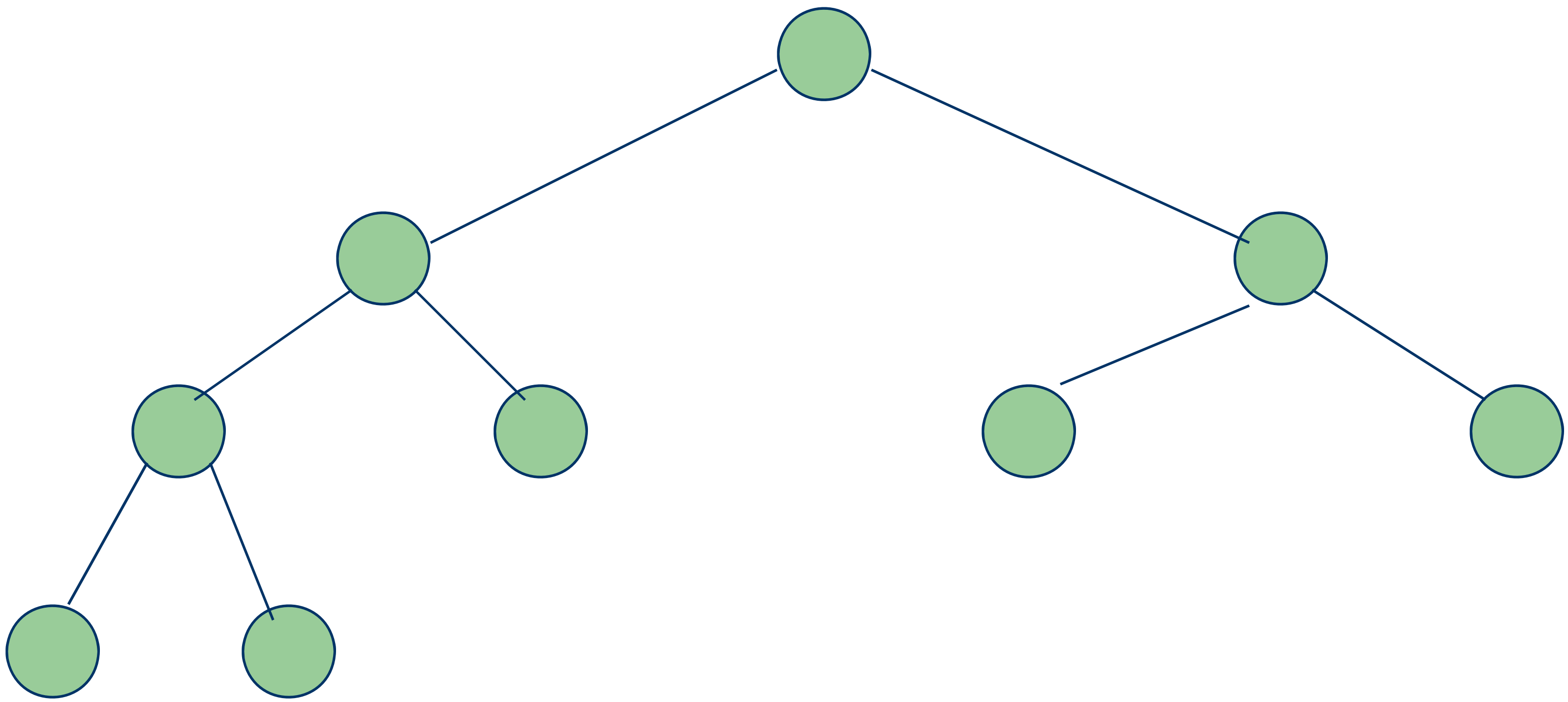
Inorder And Level Order

- Scan level order from left to right using inorder to separate left and right subtrees.
- inorder = g d h b e i a f j c
- level order = a b c d e f g h i j
- Tree root is a; gdhbei are in left subtree; fjc are in right subtree.

Heap

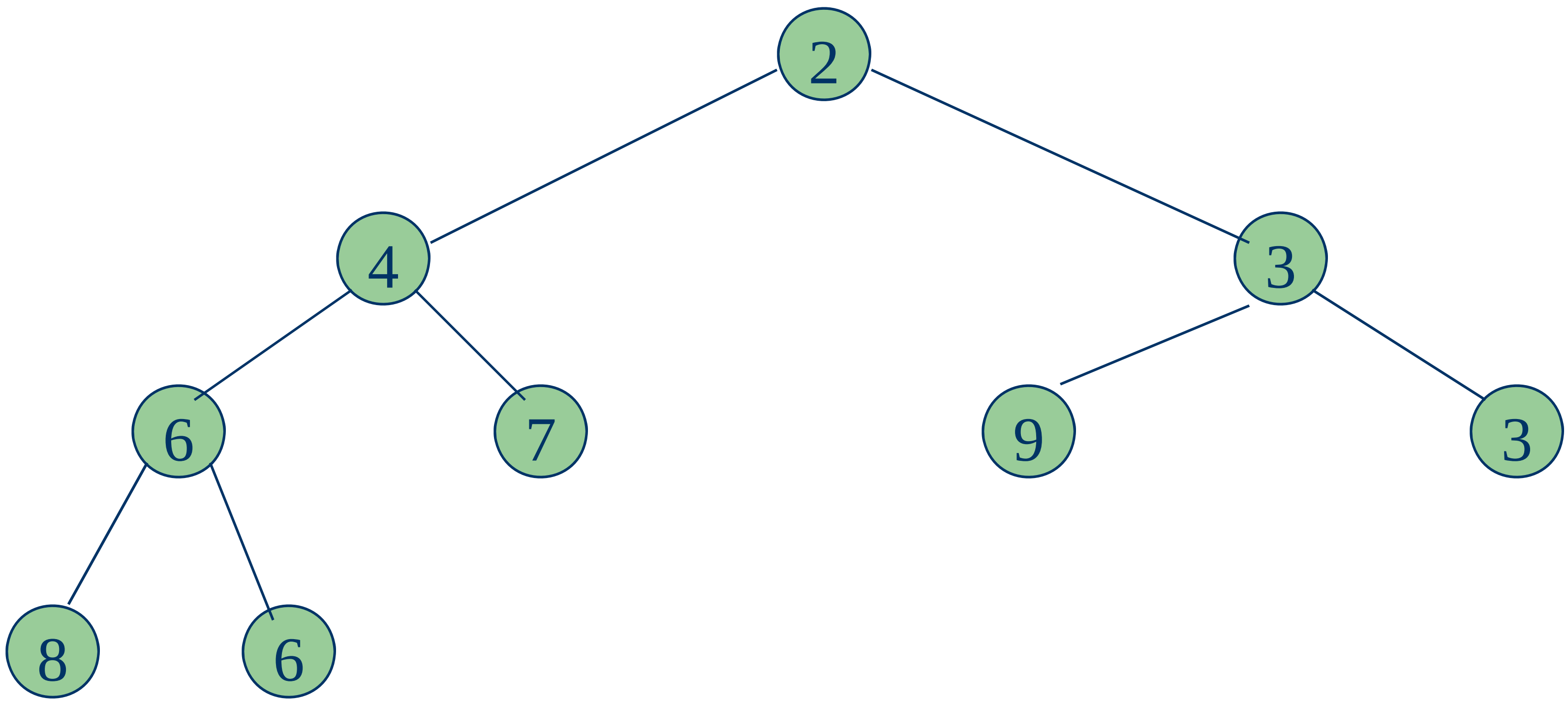
- An array of objects than can be viewed as a complete binary tree such that:
  - Each tree node corresponds to elements of the array
  - The tree is complete except possibly the lowest level, filled from left to right
  - The max-heap property for all nodes I in the tree must be maintained except for the root:
    - Value(Parent(I)) $\geq$ value(I)
  - Similarly for min-heap:
    - Value(Parent(I)) $\leq$ value(I)

Min Heap With 9 Nodes



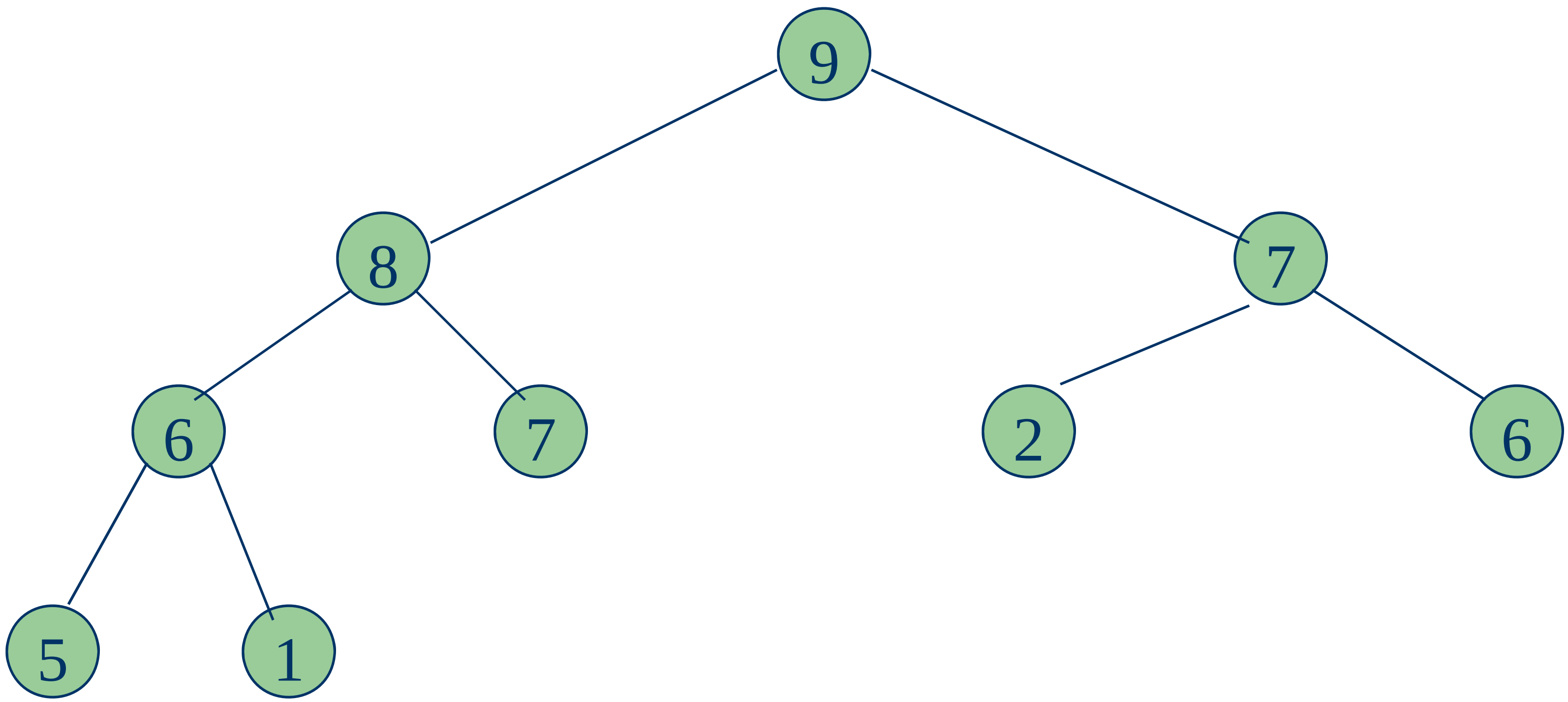
Complete binary tree with 9 nodes.

Min Heap With 9 Nodes



Complete binary tree with 9 nodes

Max Heap With 9 Nodes



Complete binary tree with 9 nodes

Heap Height

Since a heap is a complete binary tree, the height of an *n* node heap is $\lceil \log_2(n+1) \rceil - 1$ or $O(\log n)$.

Application of Heaps

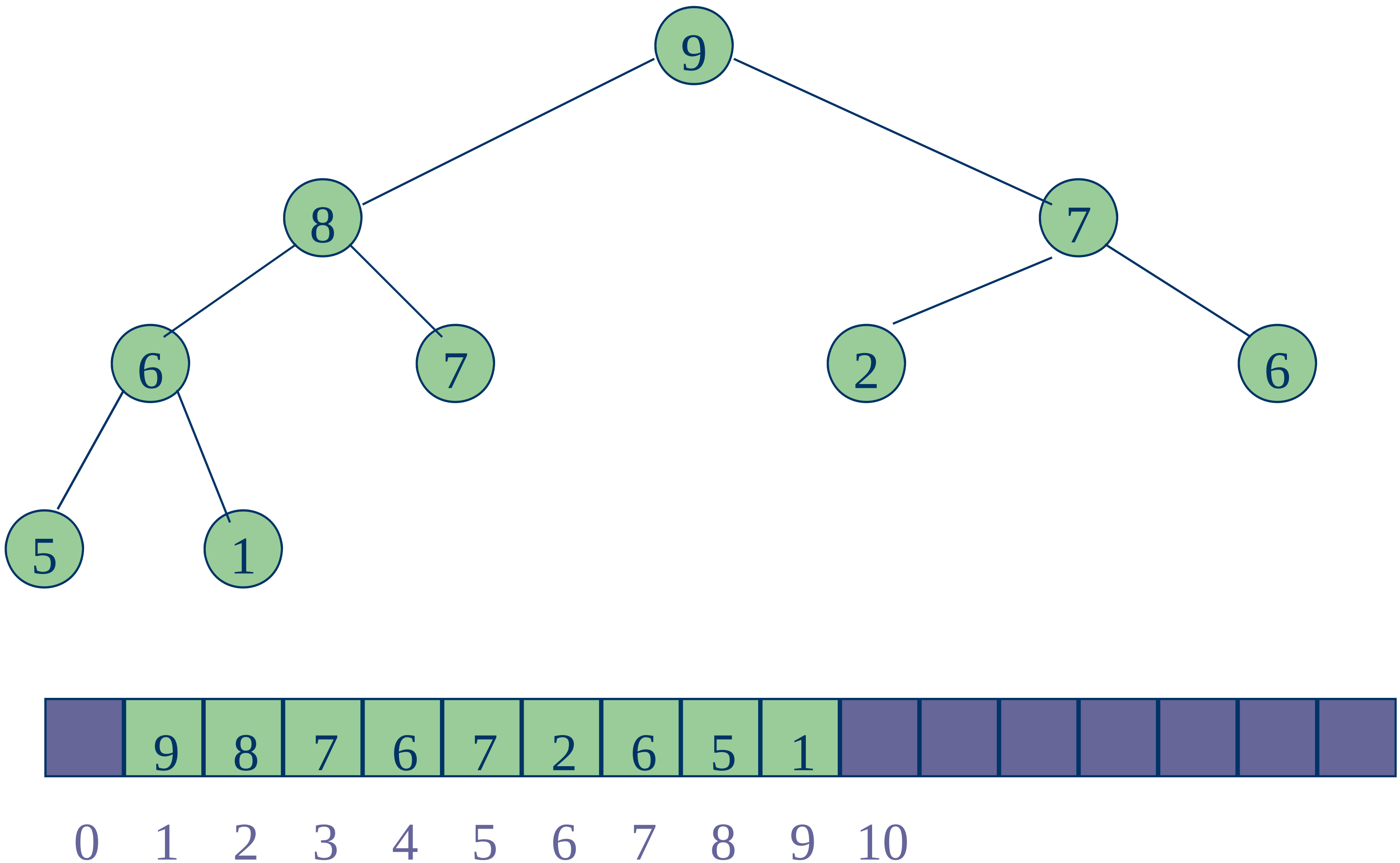
- Delete the minimum/maximum value and return it. This operation is called
  - deleteMin / deleteMax.
- Insert a new data value

Applications of Heaps:

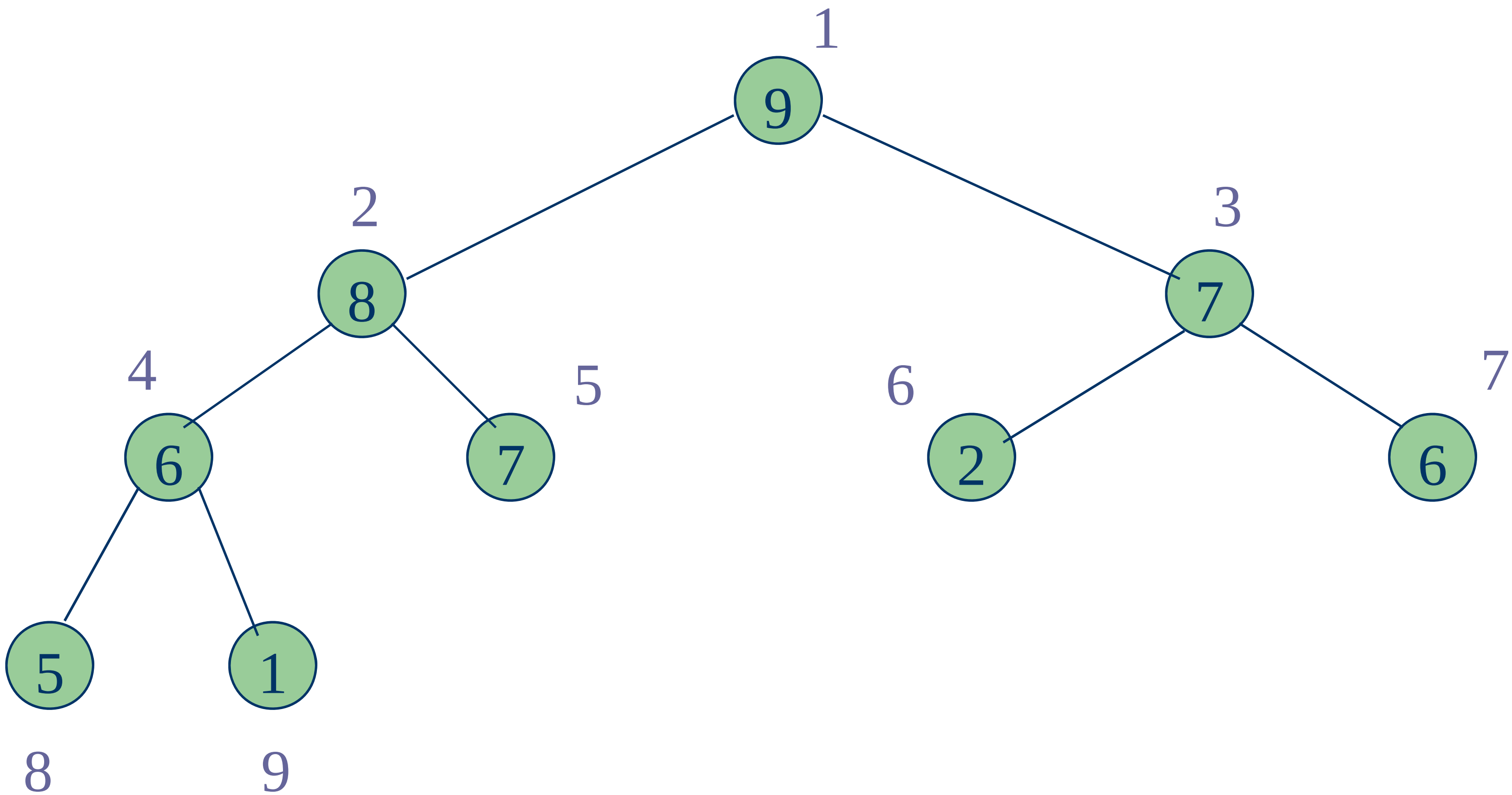
- A heap implements a **priority queue**, which is a queue that orders entities not a on first-come first-serve basis, but on a priority basis: the item of highest priority is at the head, and the item of the lowest priority is at the tail

- Another application: sorting, which will be seen later

A Heap Is Efficiently Represented as An Array



Moving Up And Down A Heap



Inserting into a max-heap

- Suppose you want to insert a new value x into the heap

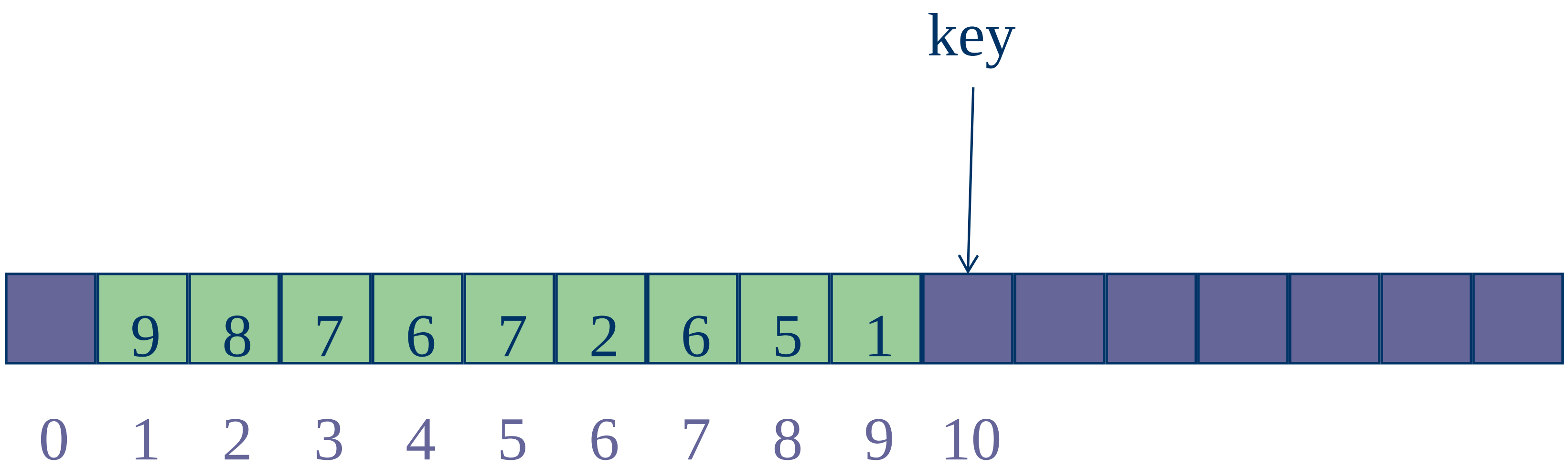
- Create a new node at the “end” of the heap (or insert *x* at the end of the array)

- If x is <= its parent, done

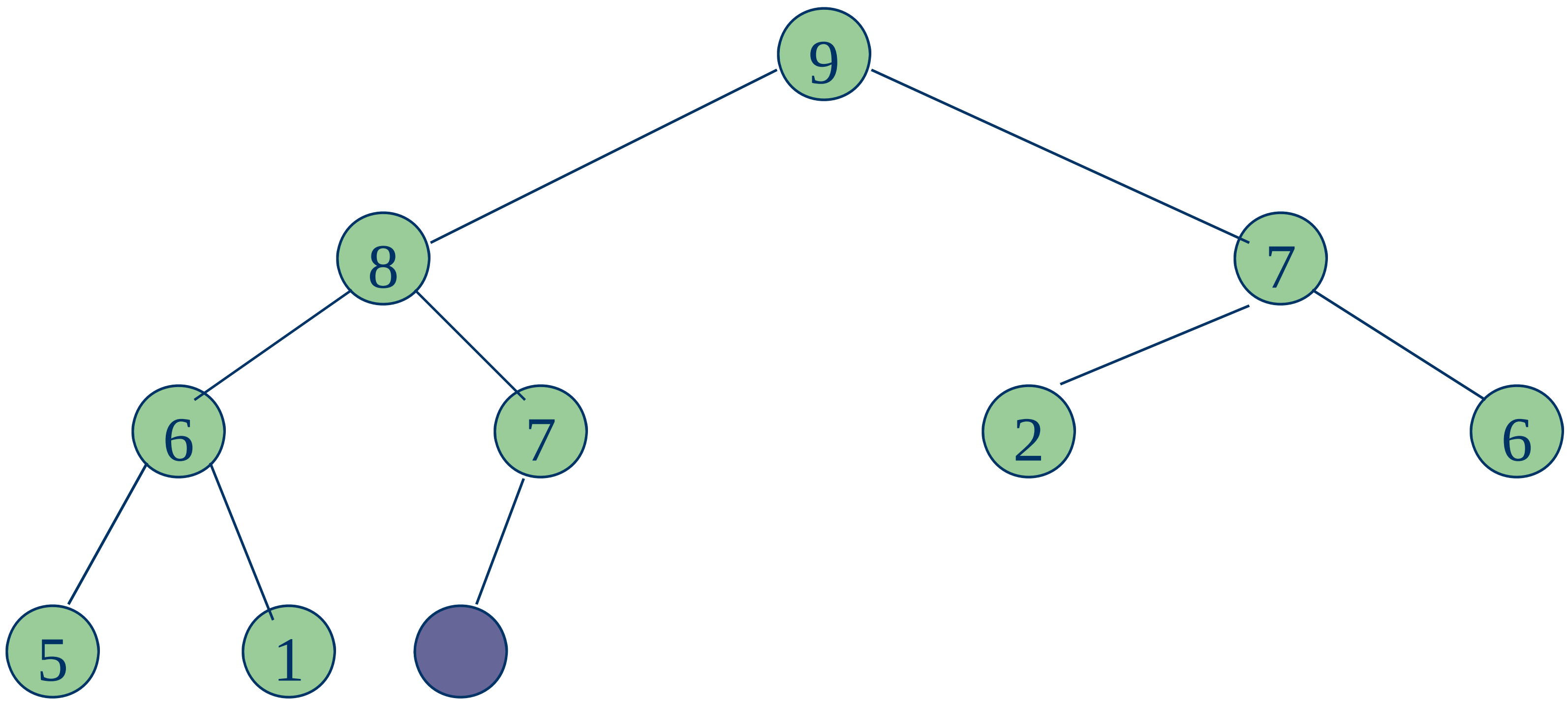
- Otherwise, we have to restore the heap:

  – Repeatedly swap *x* with its parent until either *x* reaches the root or *x* becomes <= its parent

Inserting into a max-heap



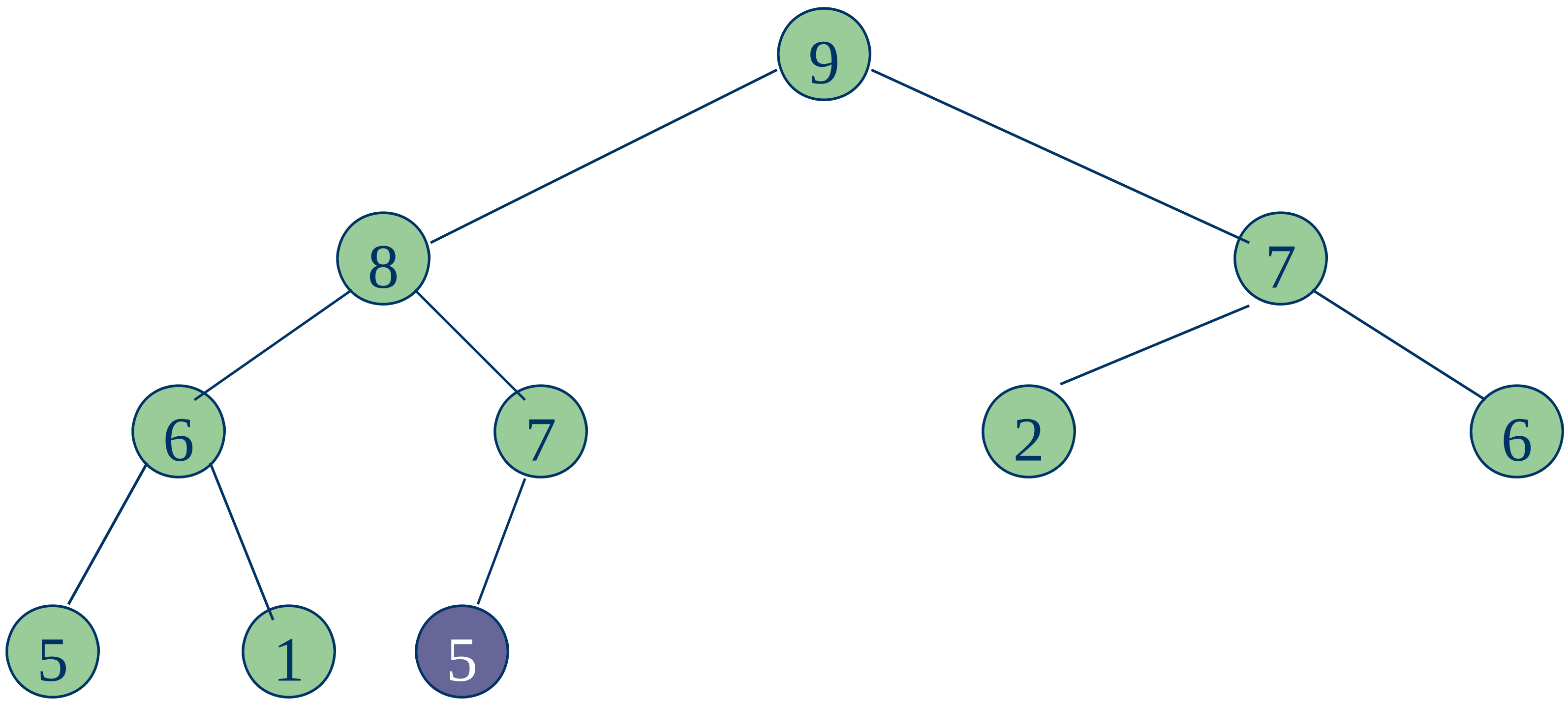
Inserting An Element Into A Max Heap



Complete binary tree with 10 nodes.

	9	8	7	6	7	2	6	5	1							
0	1	2	3	4	5	6	7	8	9	10						

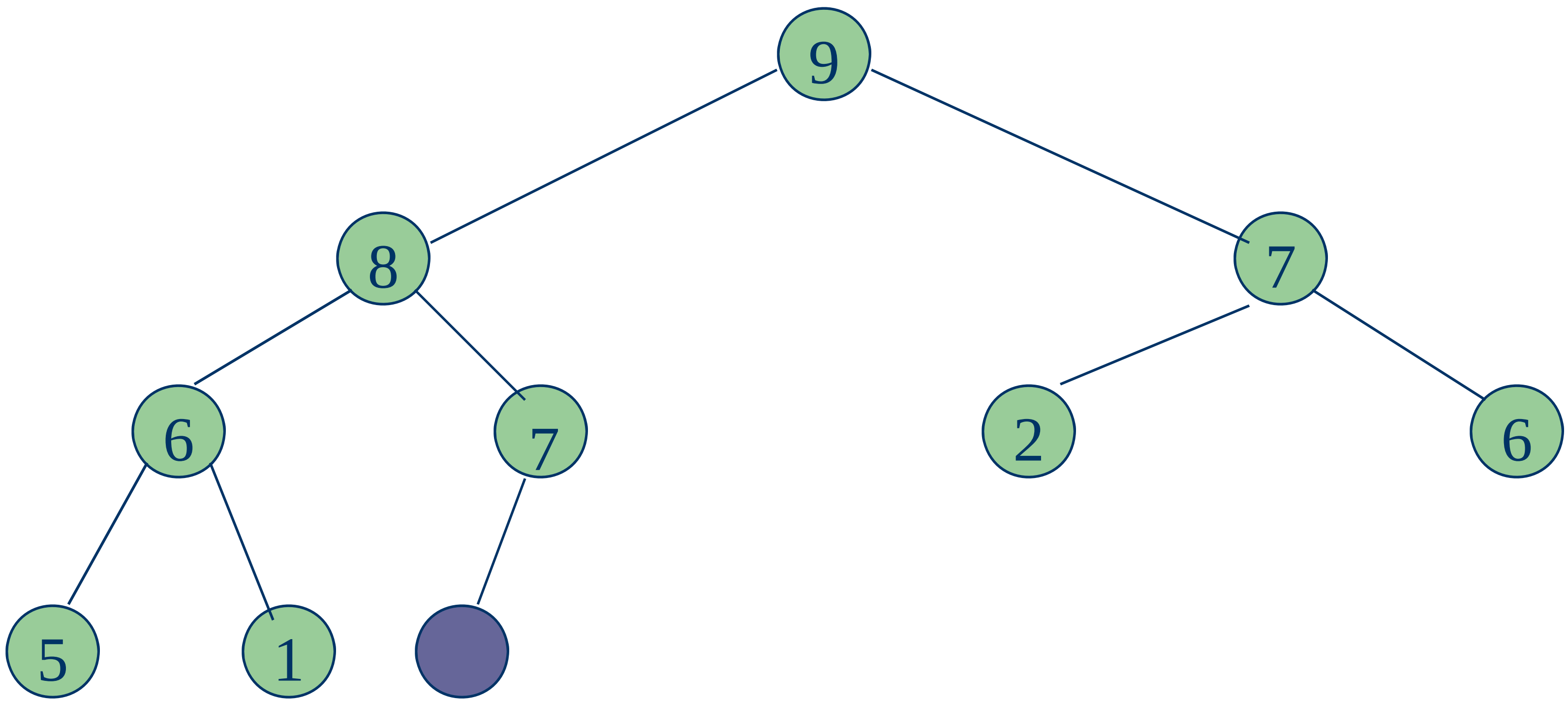
Inserting An Element Into A Max Heap



New element is 5.

	9	8	7	6	7	2	6	5	1	5						
0	1	2	3	4	5	6	7	8	9	10						

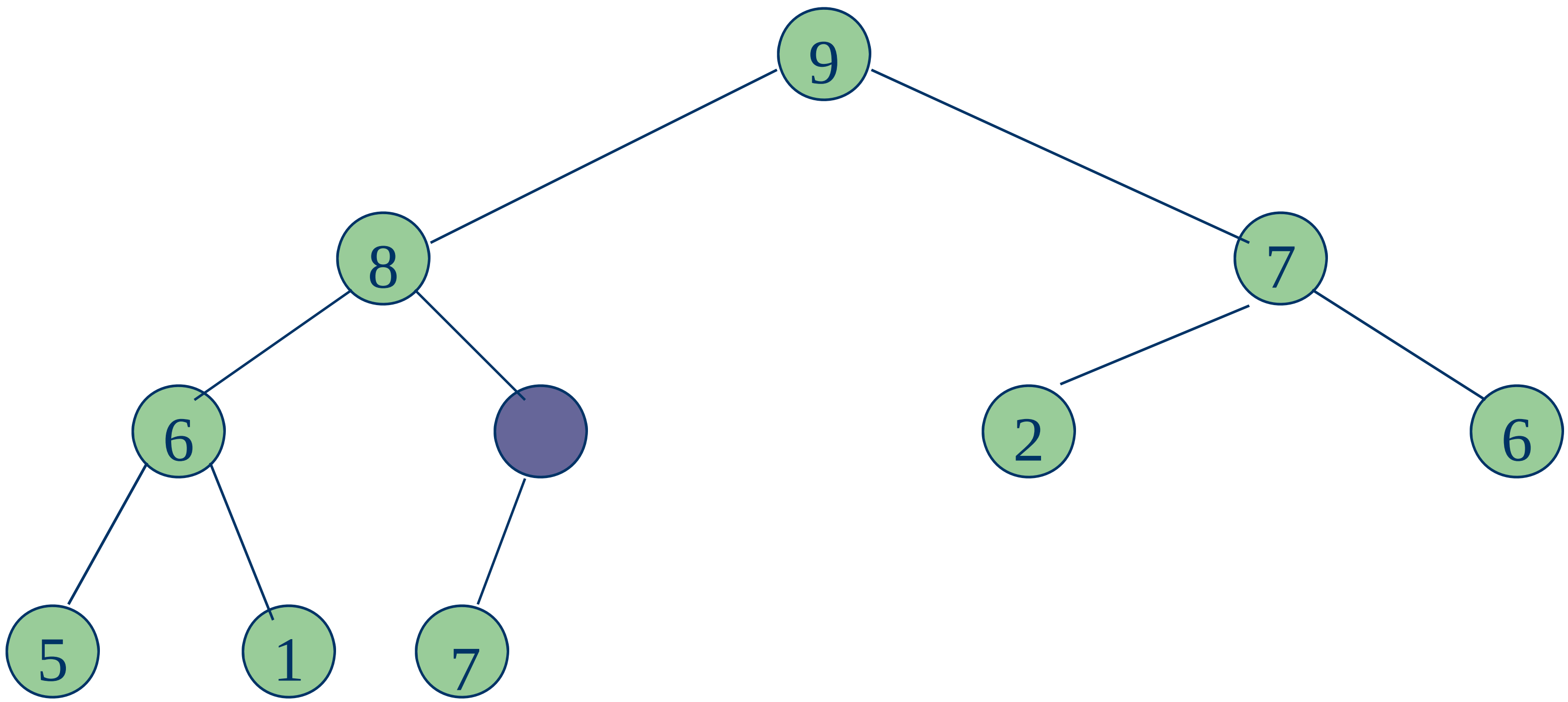
Inserting An Element Into A Max Heap



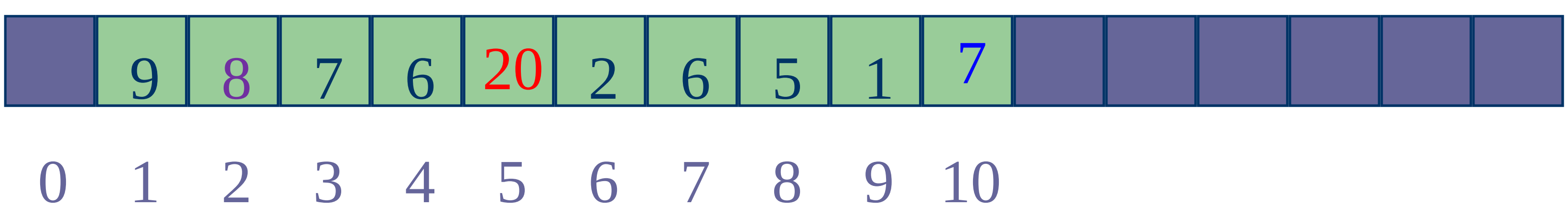
New element is 20.

0	1	2	3	4	5	6	7	8	9	10						
	9	8	7	6	7	2	6	5	1	20						

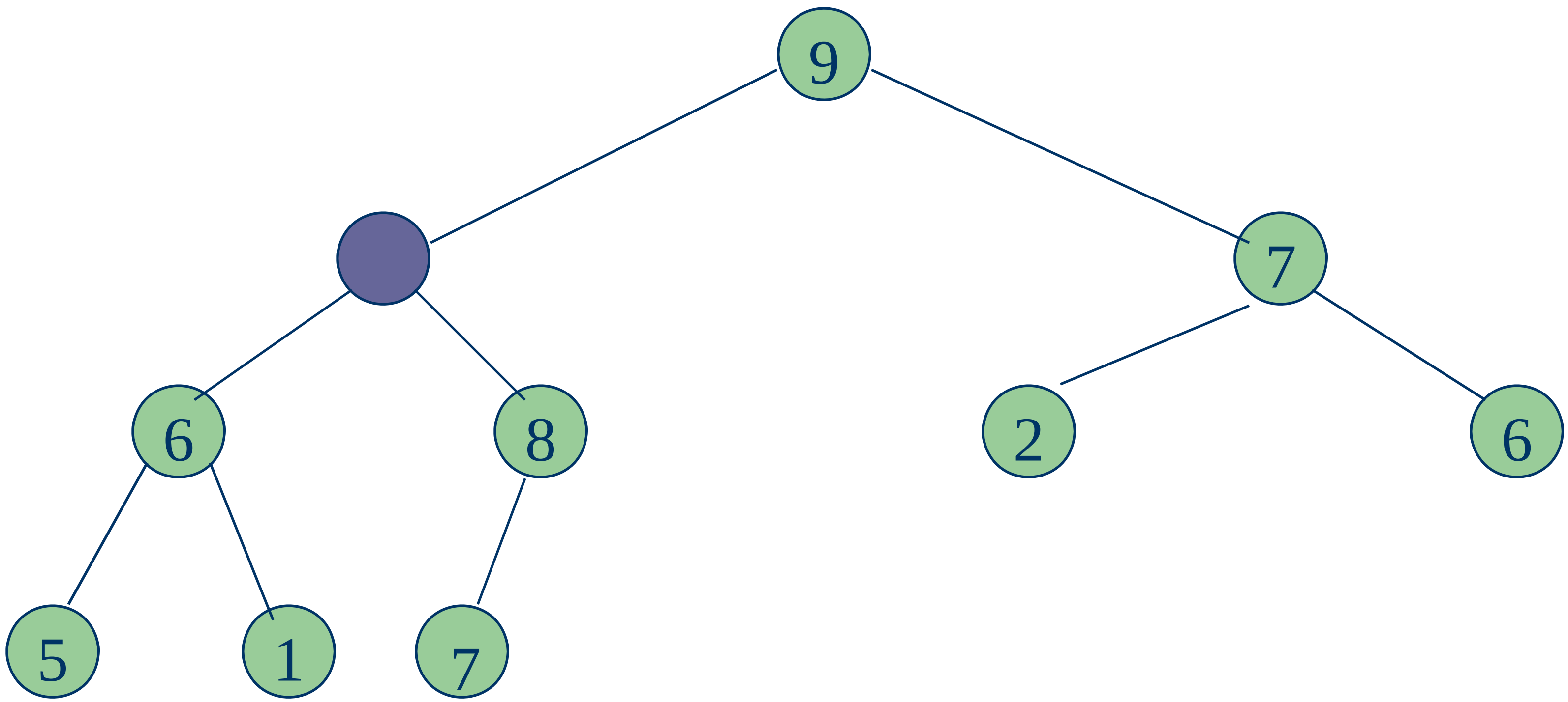
Inserting An Element Into A Max Heap



New element is 20.



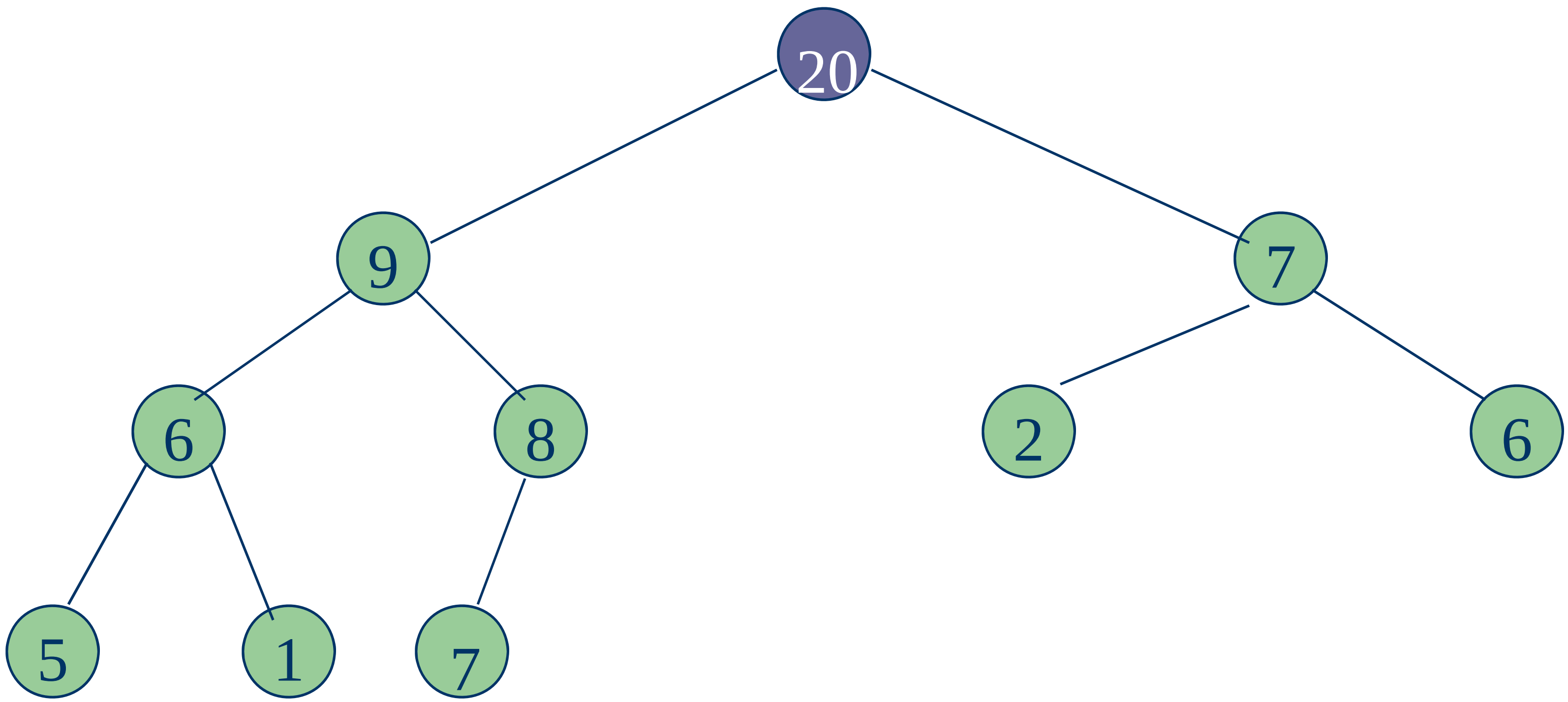
Inserting An Element Into A Max Heap



New element is 20.

	9	20	7	6	8	2	6	5	1	7							
0	1	2	3	4	5	6	7	8	9	10							

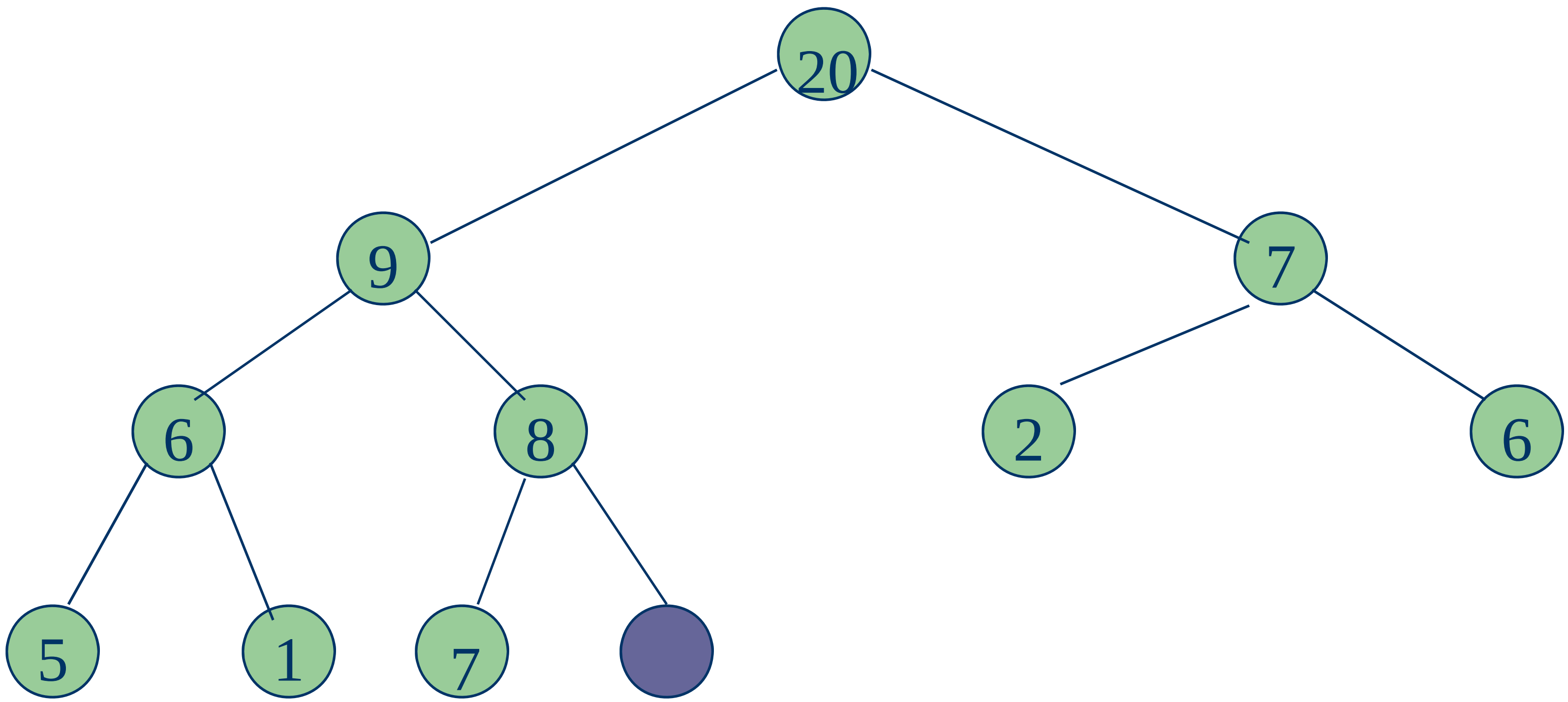
Inserting An Element Into A Max Heap



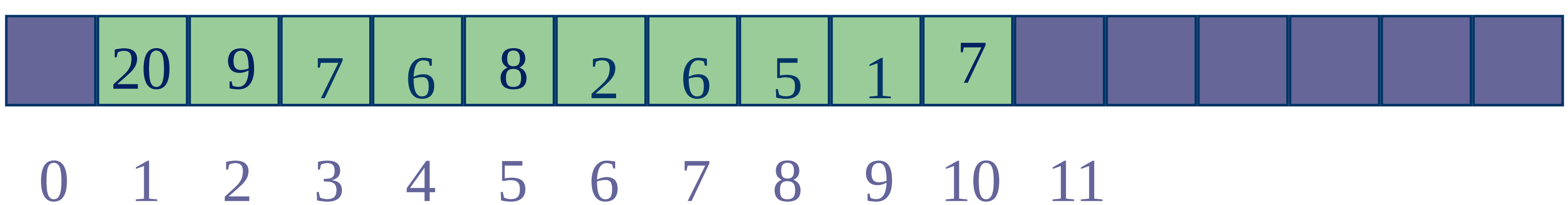
New element is 20.

0	1	2	3	4	5	6	7	8	9	10						
	20	9	7	6	8	2	6	5	1	7						

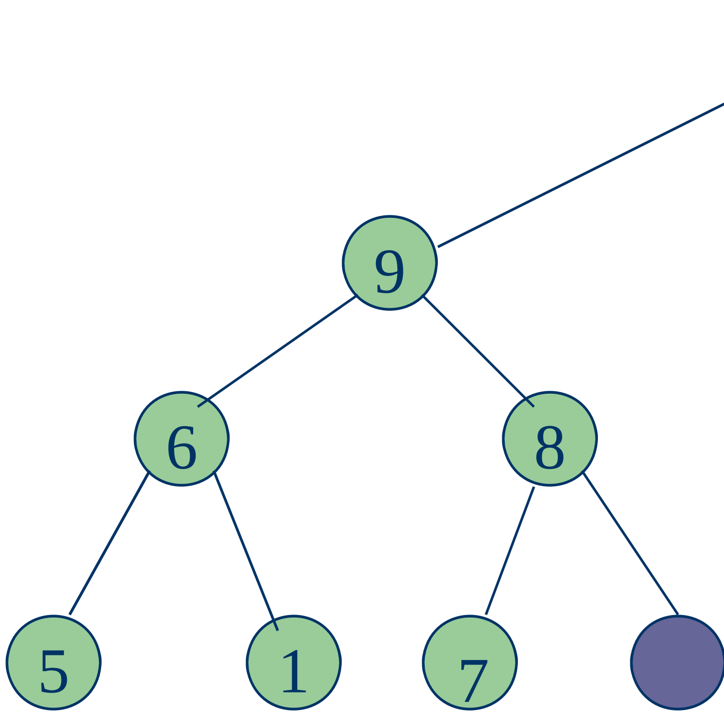
Inserting An Element Into A Max Heap



Complete binary tree with 11 nodes.



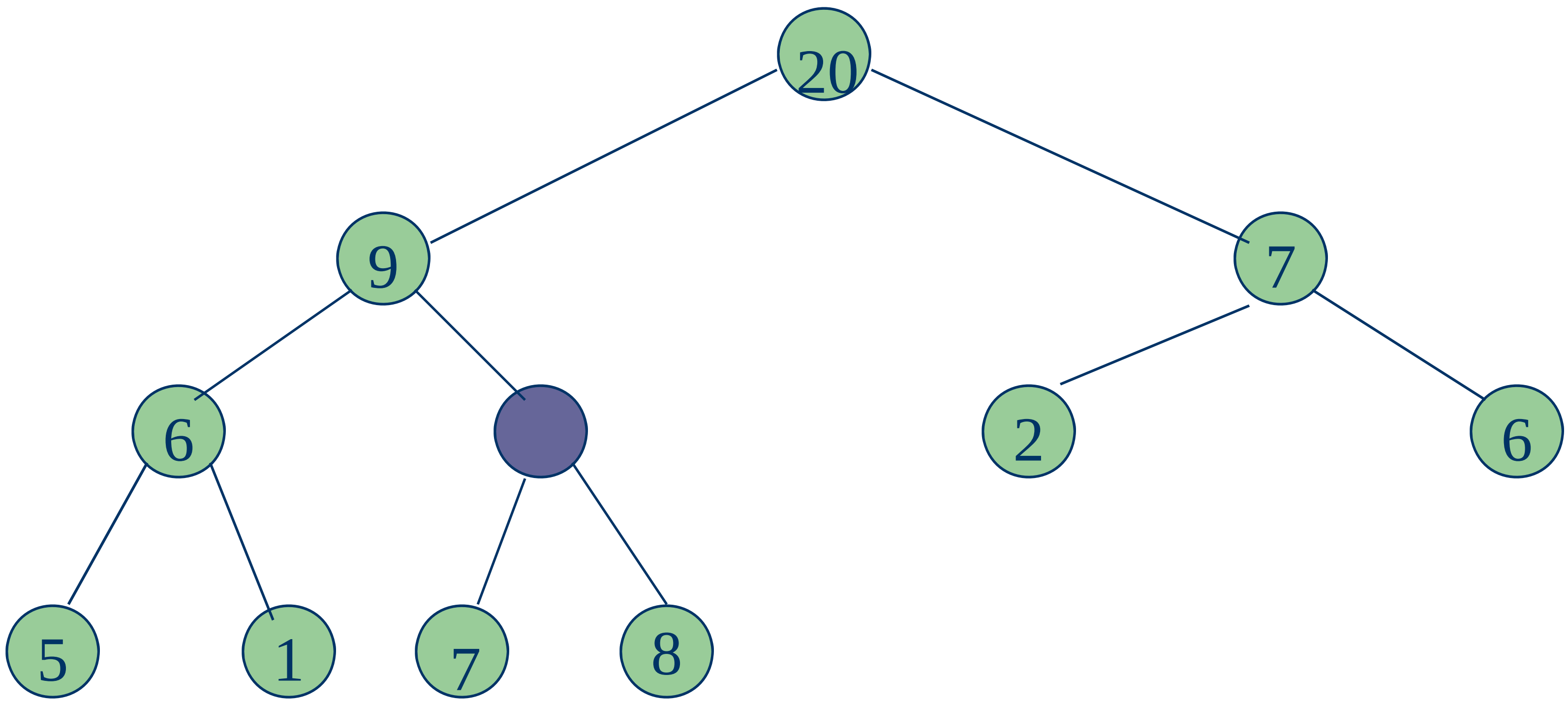
Inserting An Element Into A Max Heap



New element is 15.

	20	9	7	6	8	2	6	5	1	7	15					
0	1	2	3	4	5	6	7	8	9	10	11					

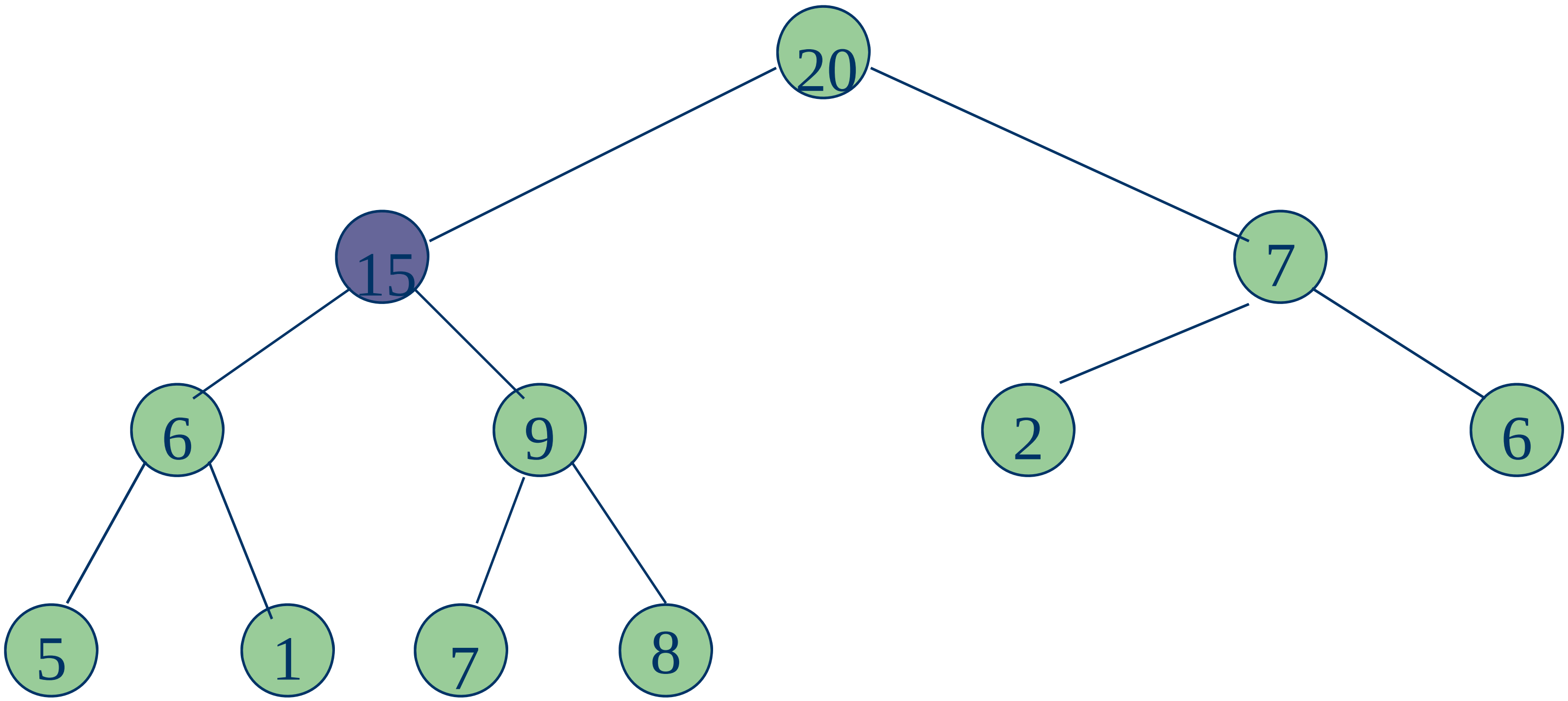
Inserting An Element Into A Max Heap



New element is 15.

	20	9	7	6	15	2	6	5	1	7	8					
0	1	2	3	4	5	6	7	8	9	10	11					

Inserting An Element Into A Max Heap



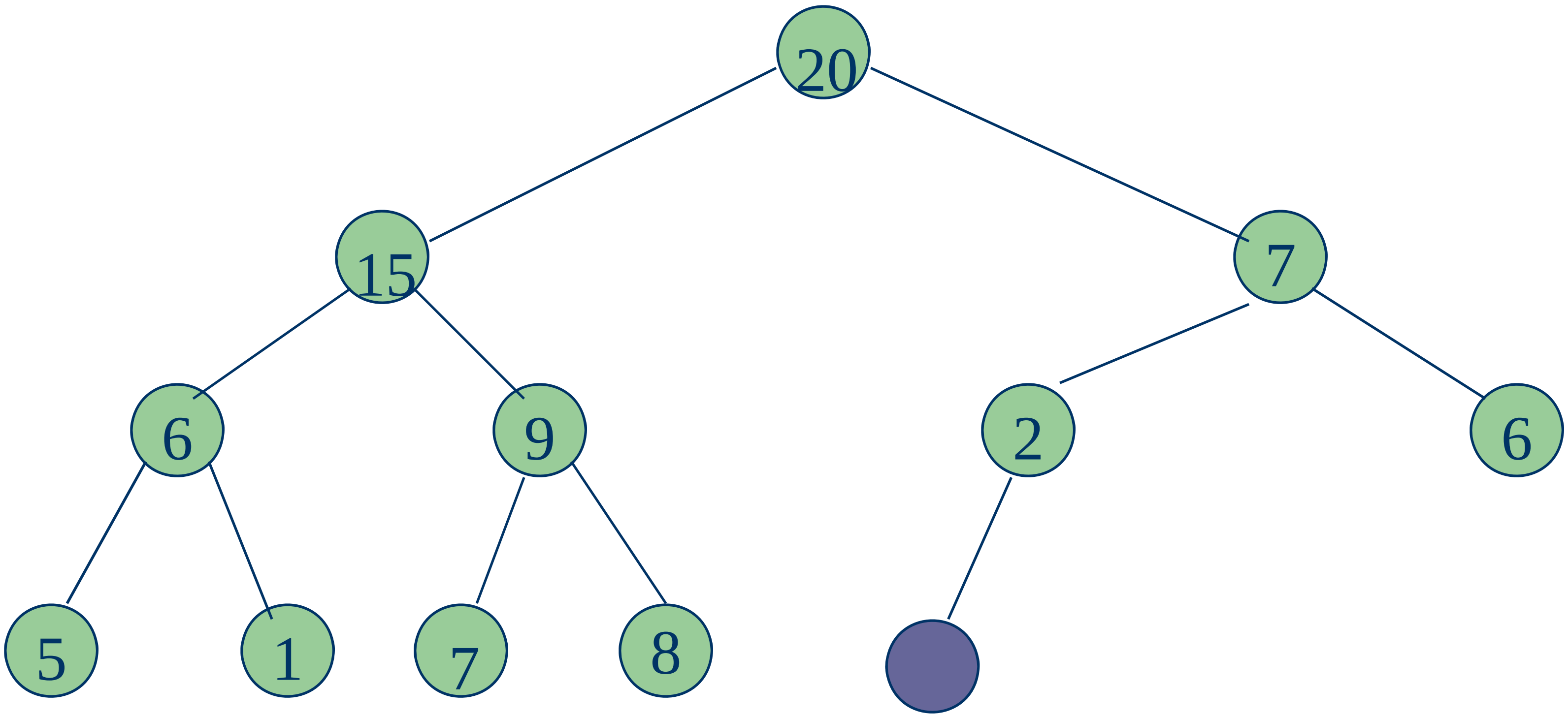
New element is 15.

	20	15	7	6	9	2	6	5	1	7	8					
0	1	2	3	4	5	6	7	8	9	10	11					

Insert

```
Heap-Insert(A,key)
        n ← n+1
        I ← n
        while I > 1 and A[⌊I / 2⌋] < key
                do      A[I] ← A[⌊I / 2⌋]
                        I ← ⌊I / 2⌋
        A[I] ← key
```

Complexity Of Insertion



Complexity is O(log n), where n is heap size.

DeleteMax in max-heaps

- The maximum value in a max-heap is at the root!
- To delete the max, you can’t just remove the data value of the root, because every node must hold a key
- Instead, take the last node from the heap, move its key to the root, and delete that last node
- But now, the tree is no longer a heap (still complete, but the root key value may no longer be $\leq$ the keys of its children

Restore Heap

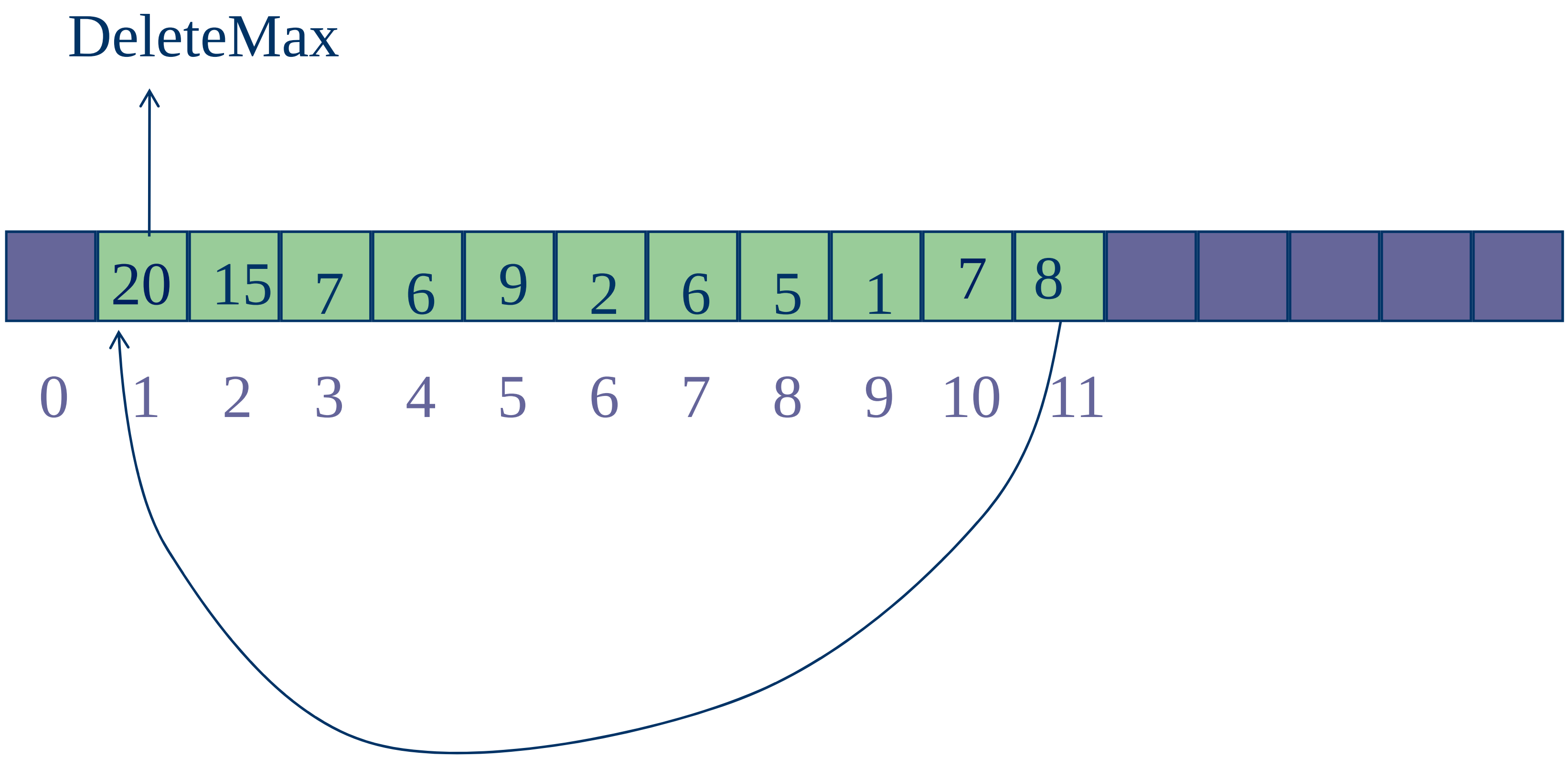
- To bring the structure back to its “heapness”, we restore the heap

- Swap the new root key with the smaller child.

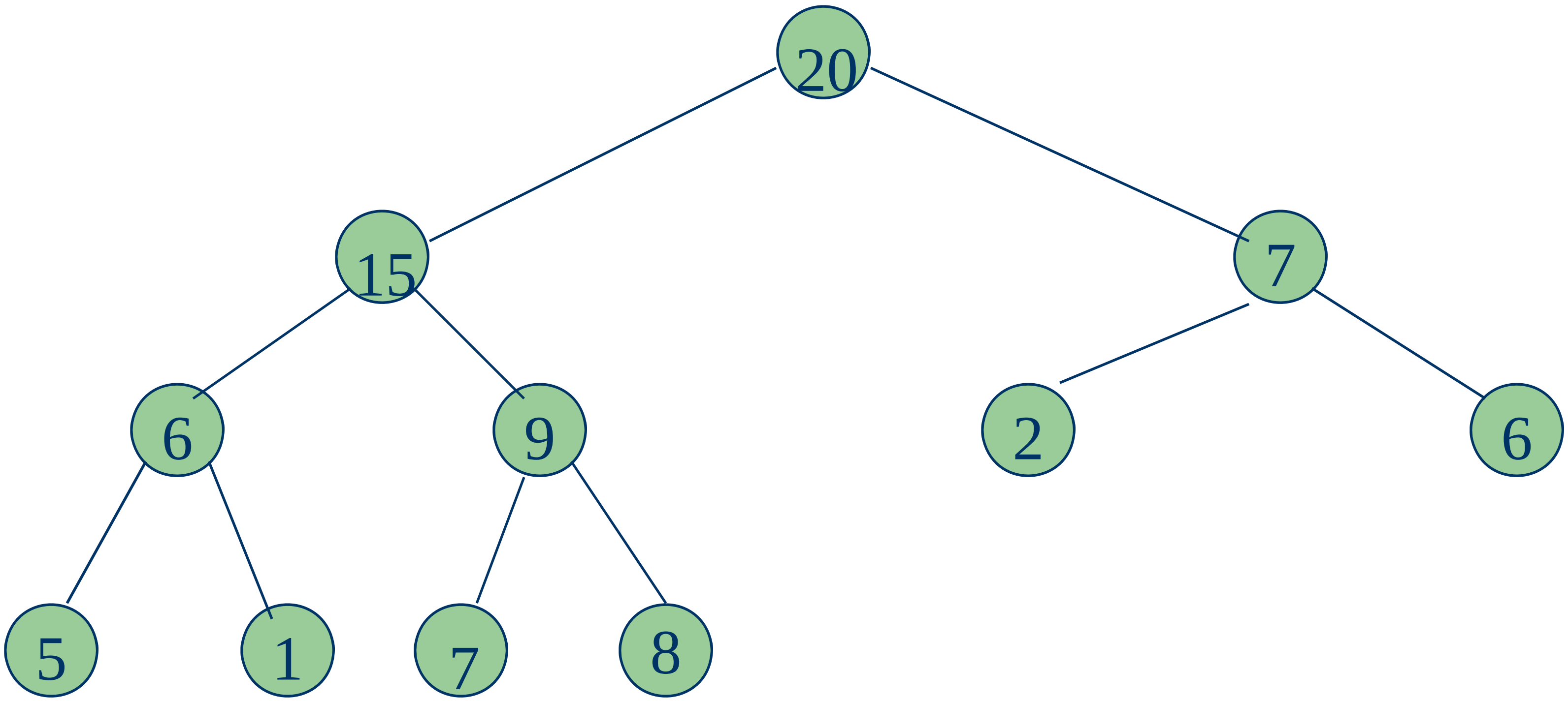
- Now the potential bug is at the one level down. If it is not already $\geq$ the keys of its children, **swap it with its larger child**

- Keep repeating the last step until the “bug” key becomes $\geq$ its children, or the it becomes a leaf

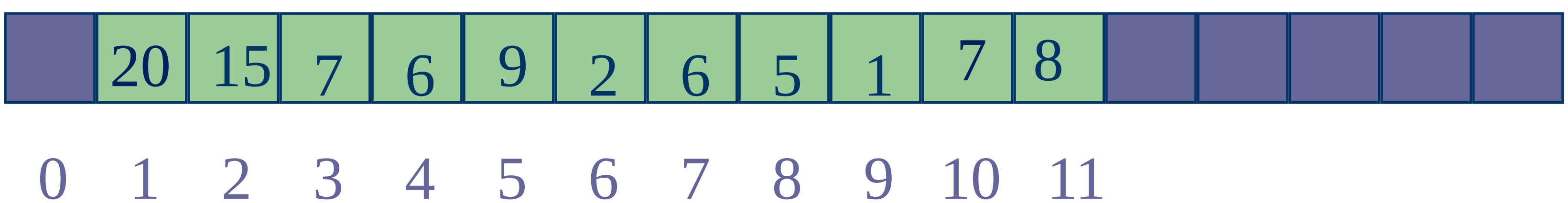
Removing The Max Element



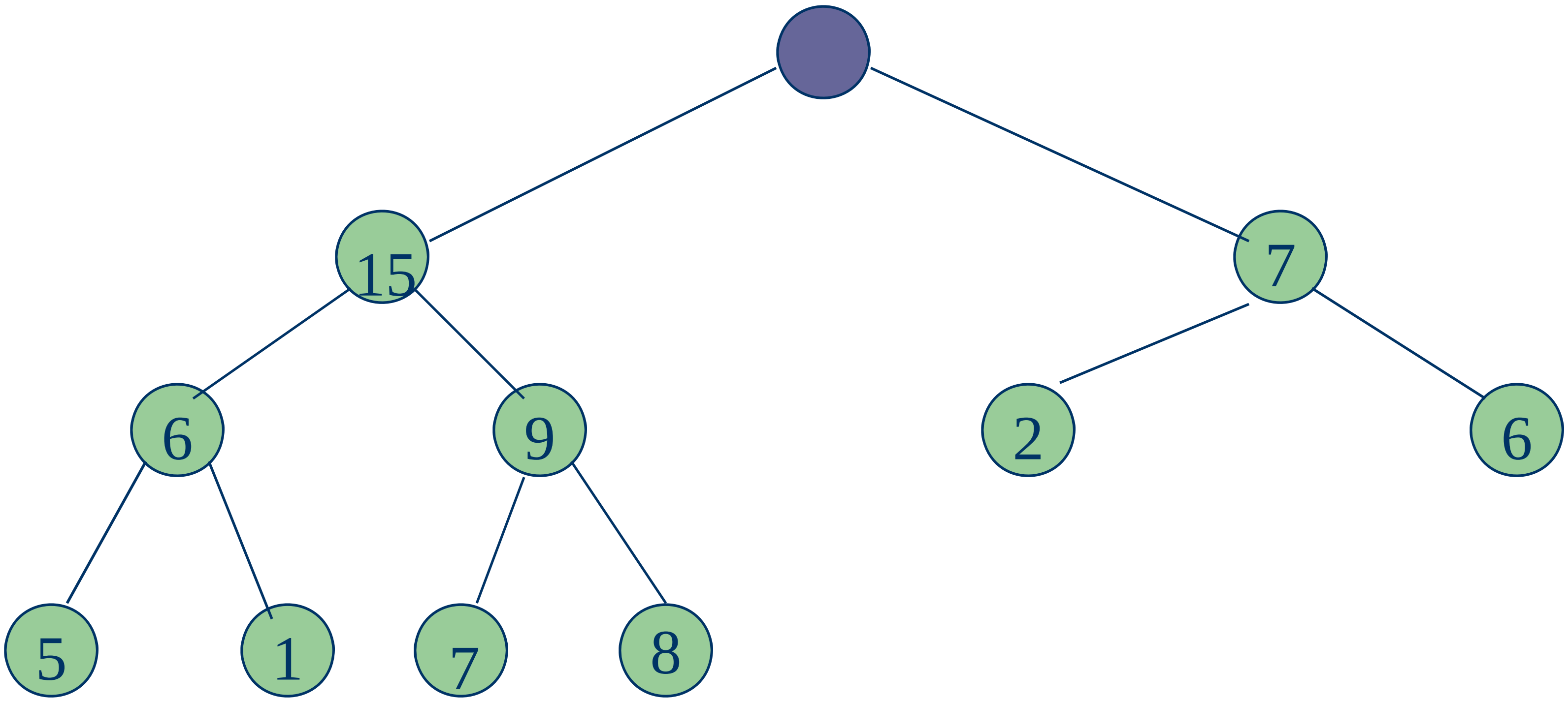
Removing The Max Element



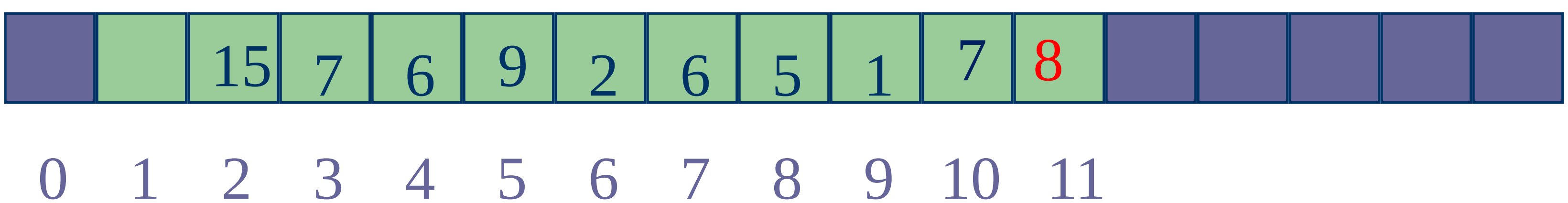
Max element is in the root.



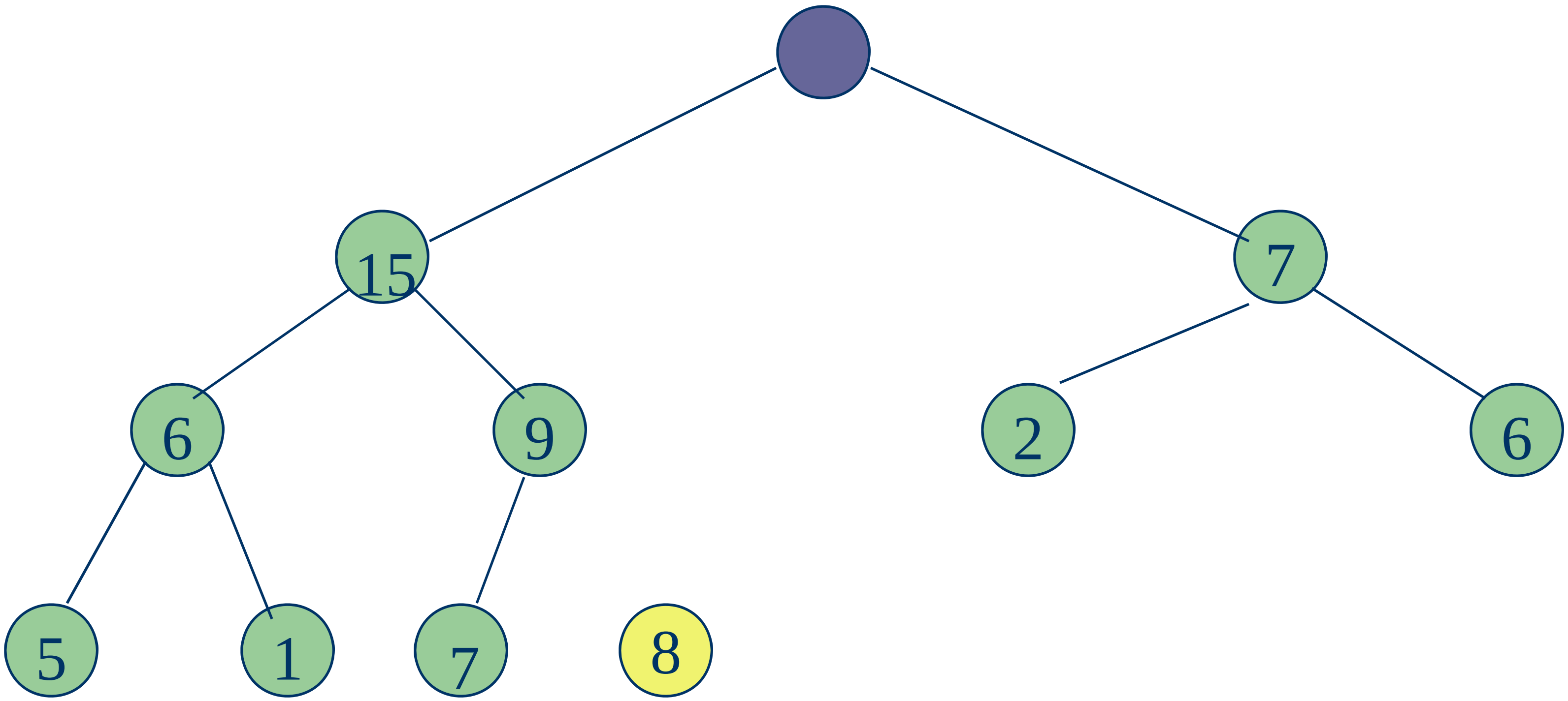
Removing The Max Element



After max element is removed.



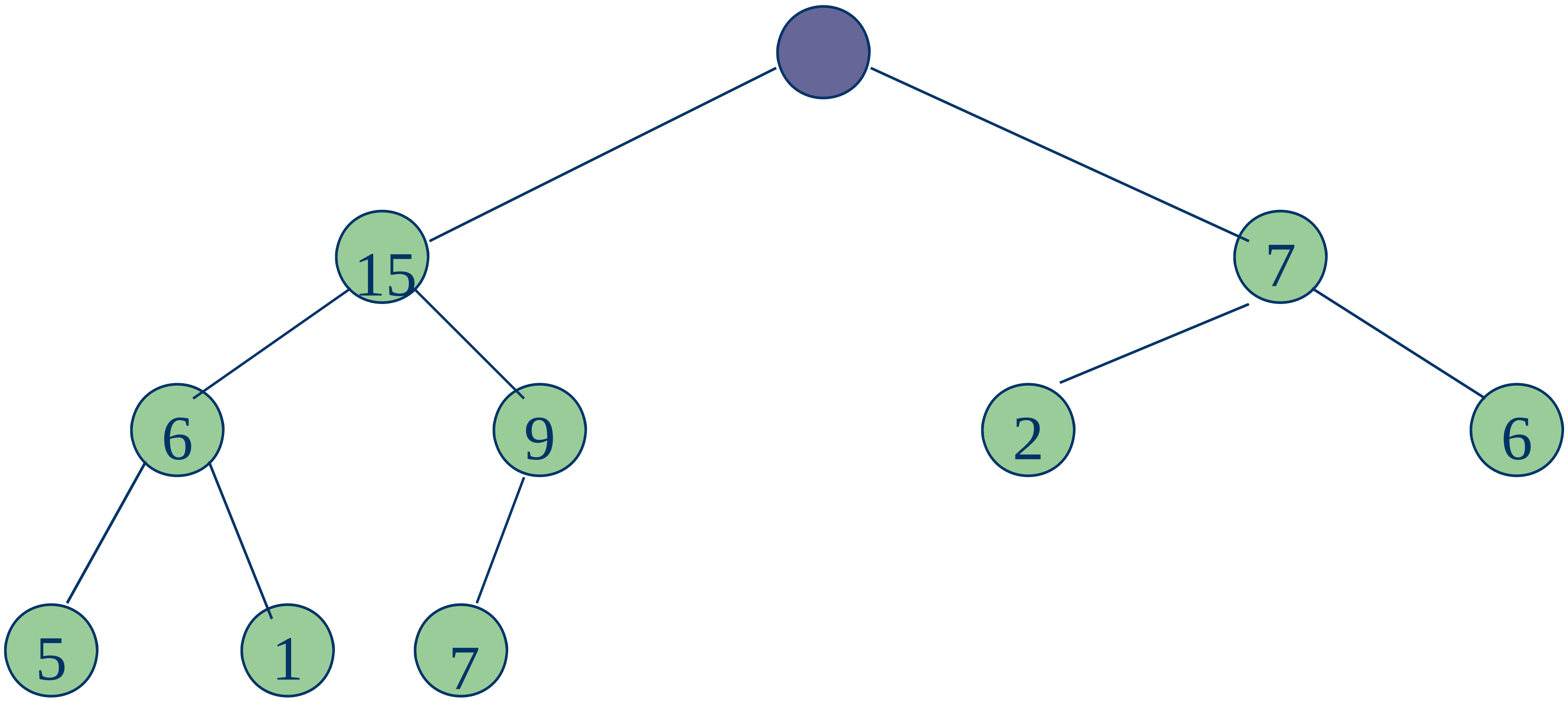
Removing The Max Element



Heap with 10 nodes.

Reinsert 8 into the heap.

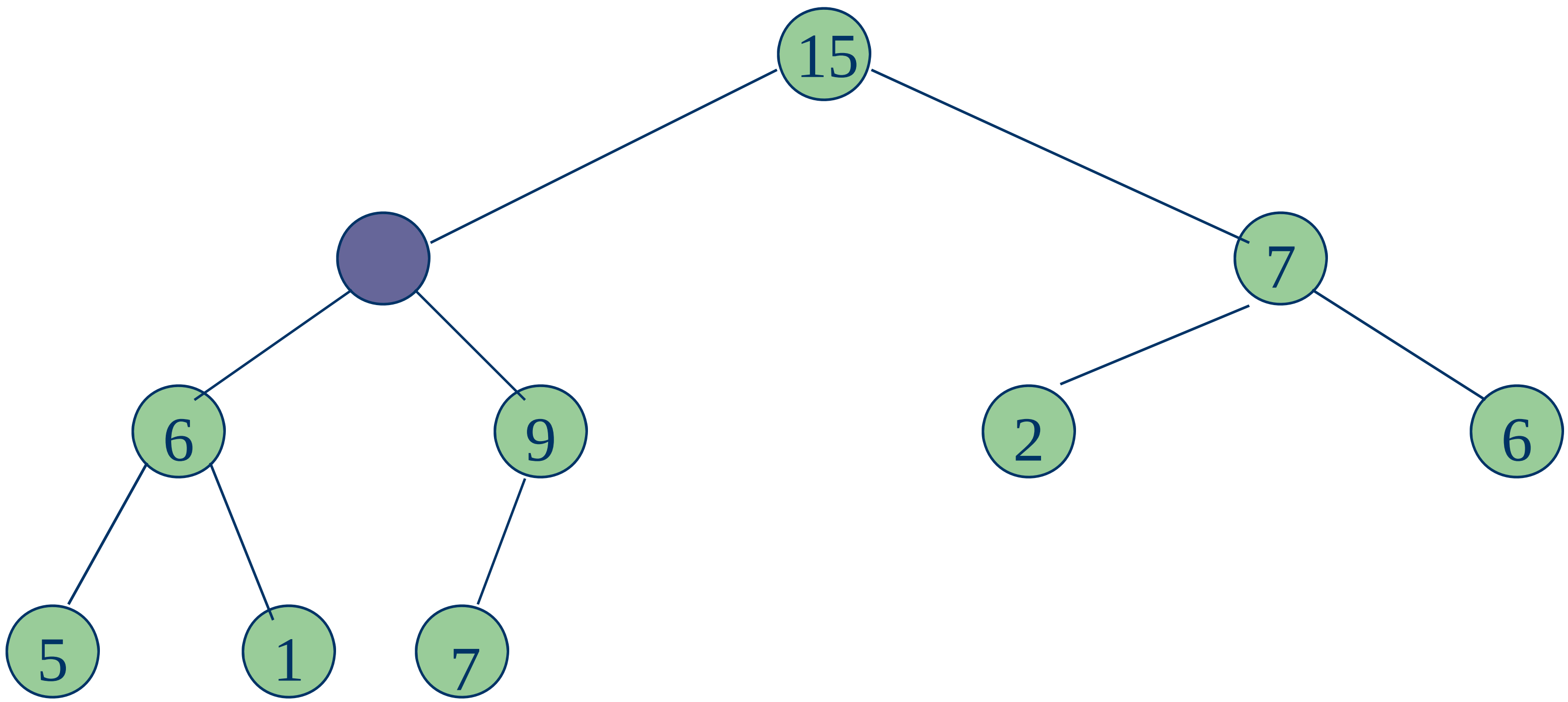
Removing The Max Element



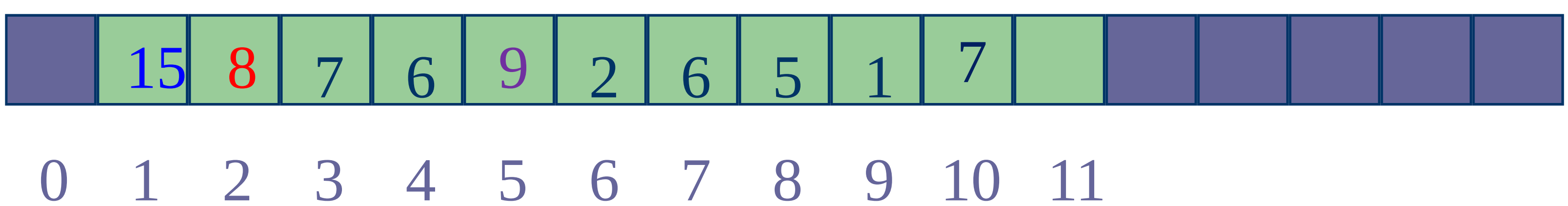
Reinsert 8 into the heap.

	8	15	7	6	9	2	6	5	1	7						
0	1	2	3	4	5	6	7	8	9	10	11					

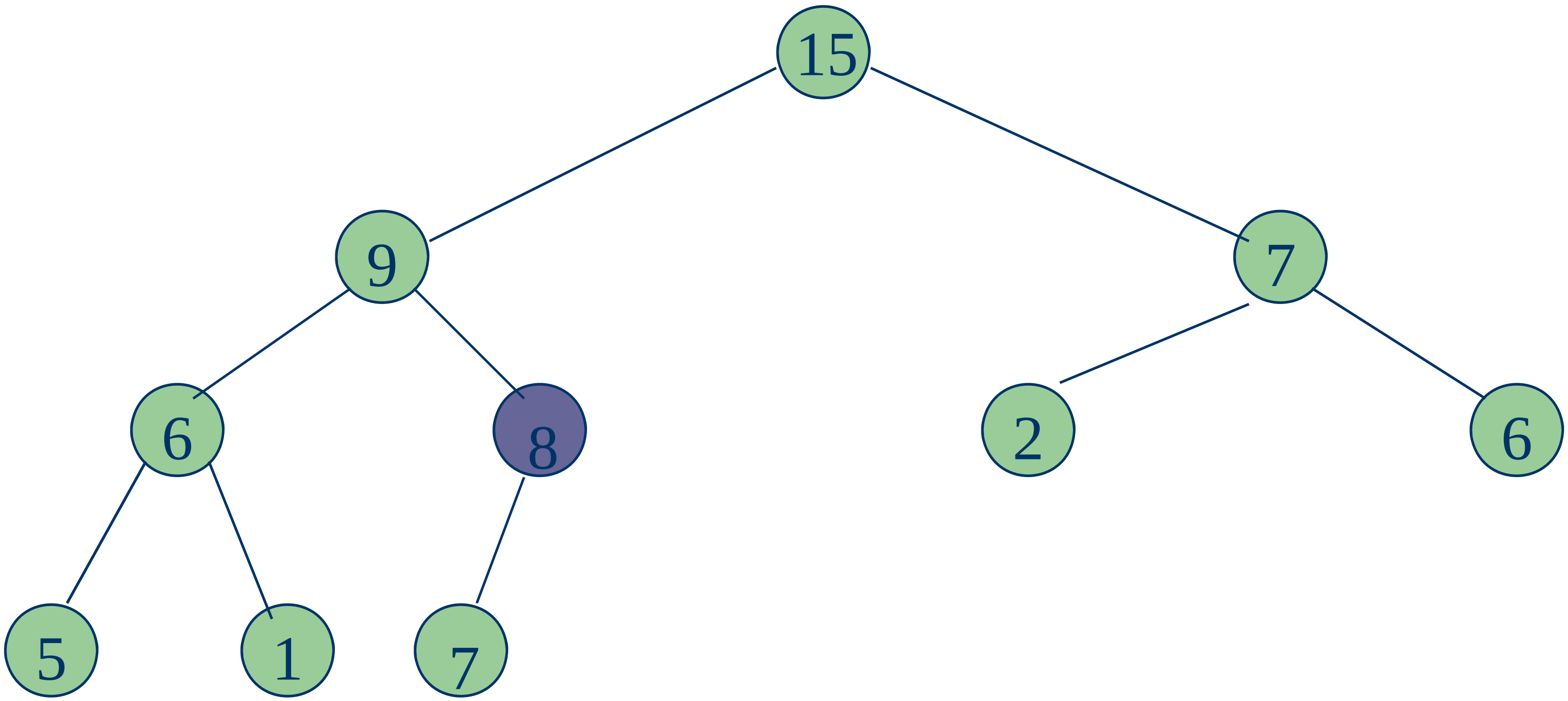
Removing The Max Element



Reinsert 8 into the heap.



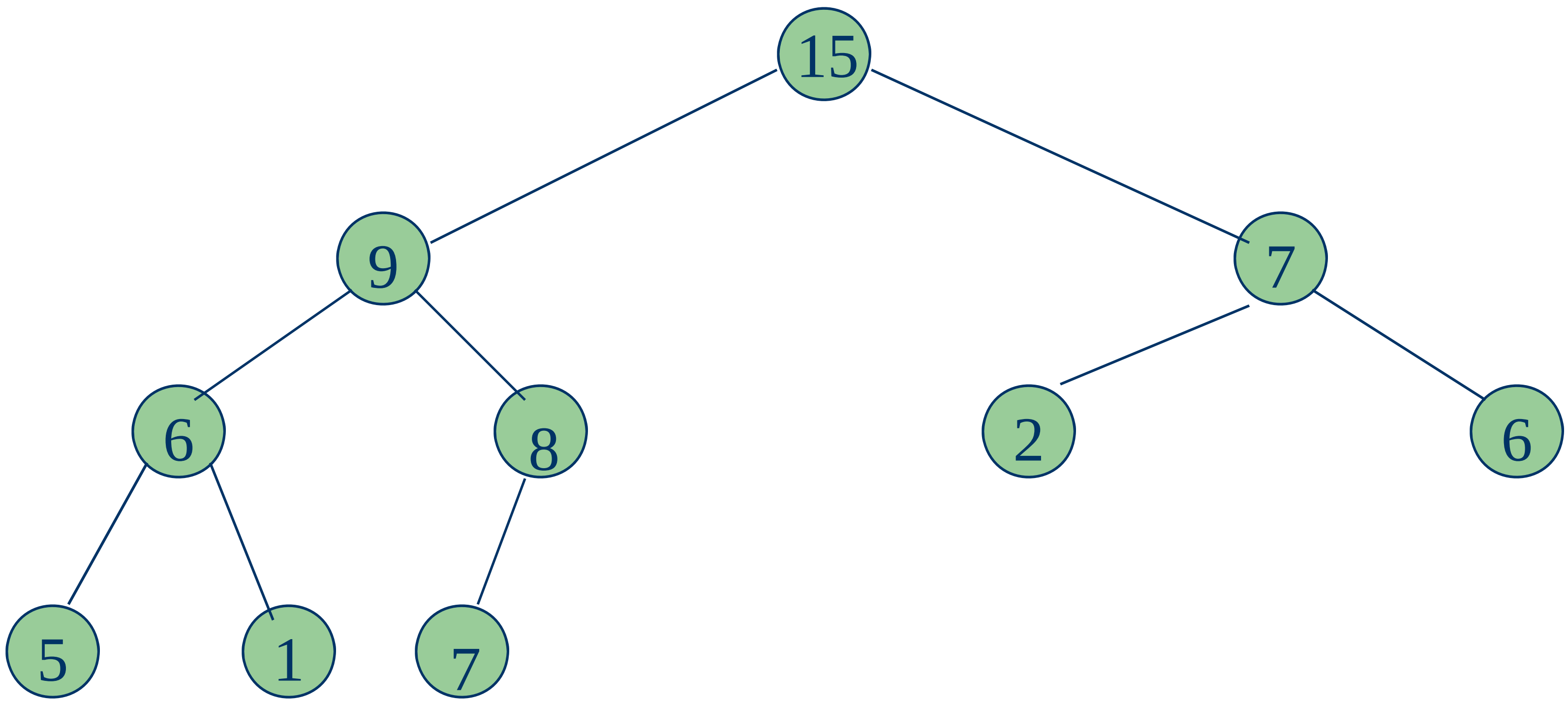
Removing The Max Element



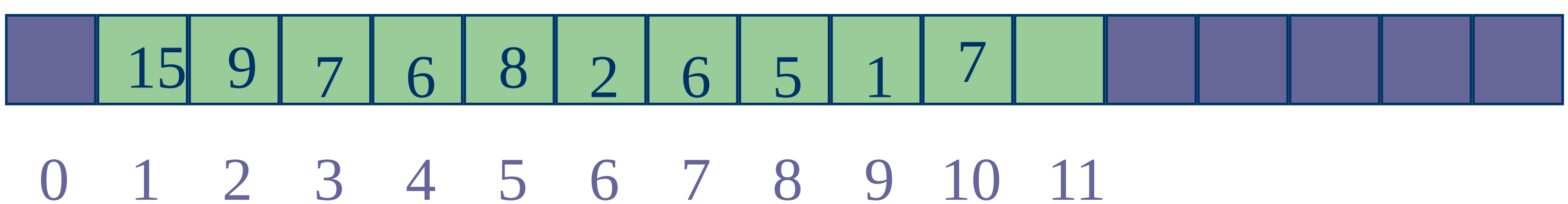
Reinsert 8 into the heap.

0	1	2	3	4	5	6	7	8	9	10	11					
	15	9	7	6	8	2	6	5	1	7						

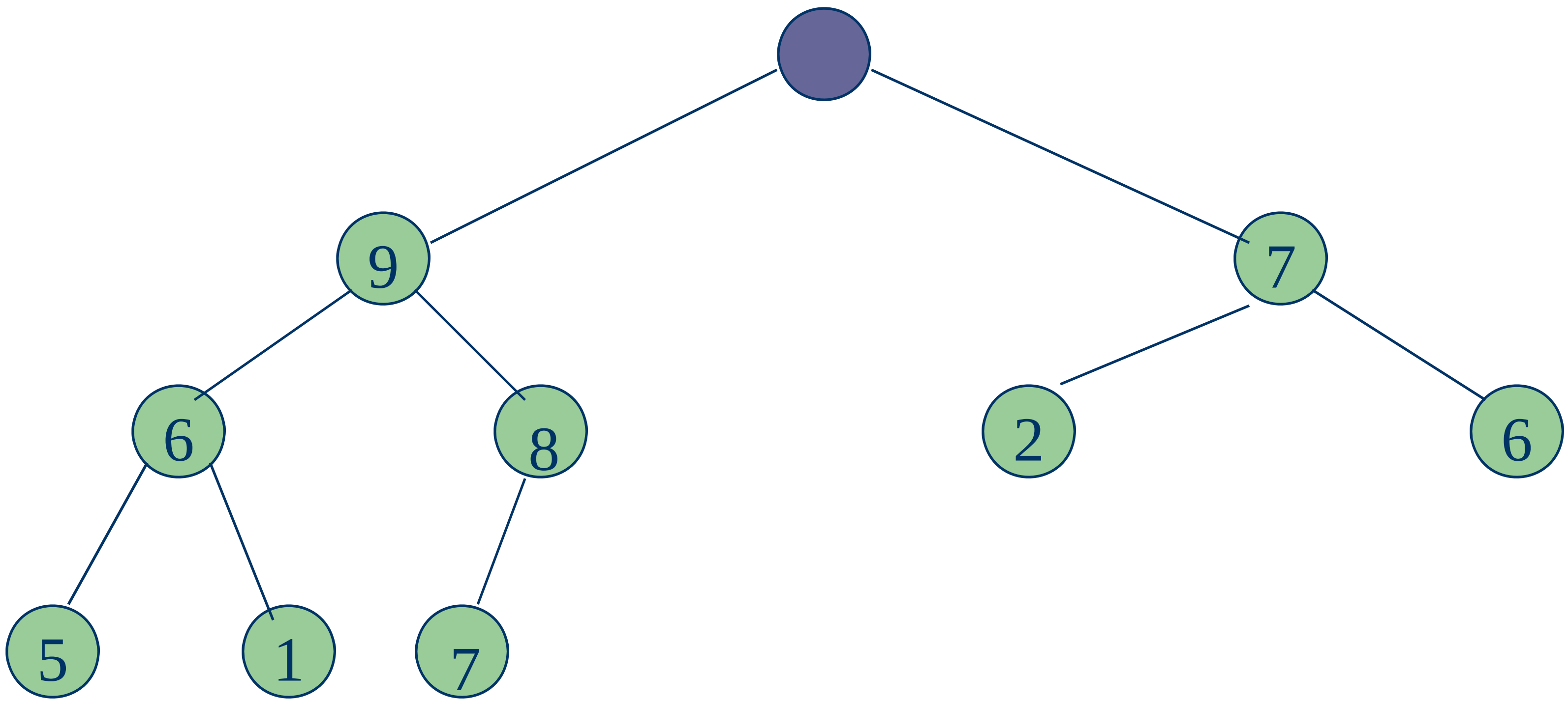
Removing The Max Element



Max element is 15.



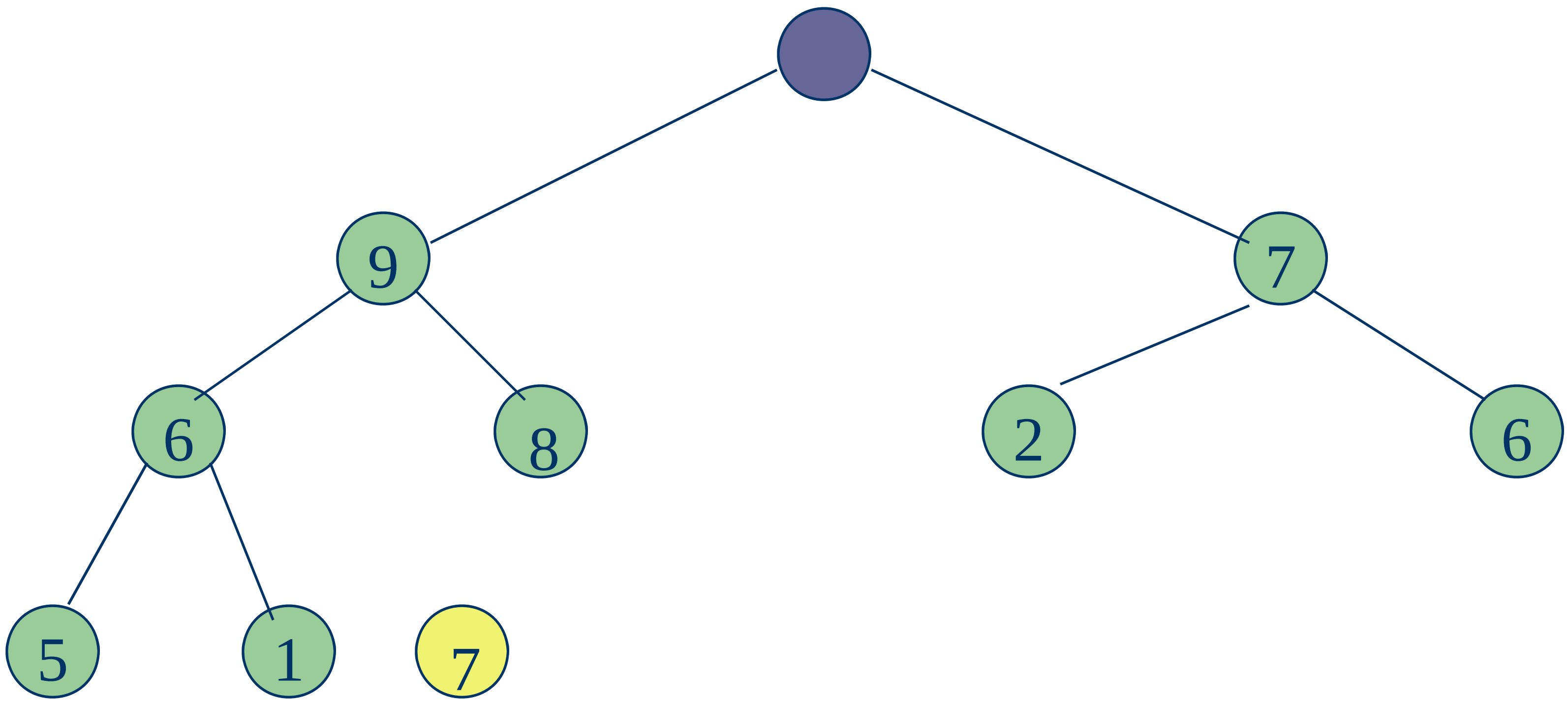
Removing The Max Element



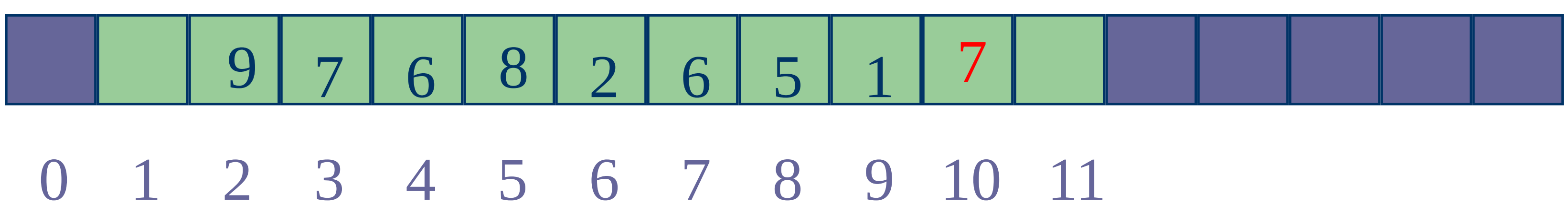
After max element is removed.

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

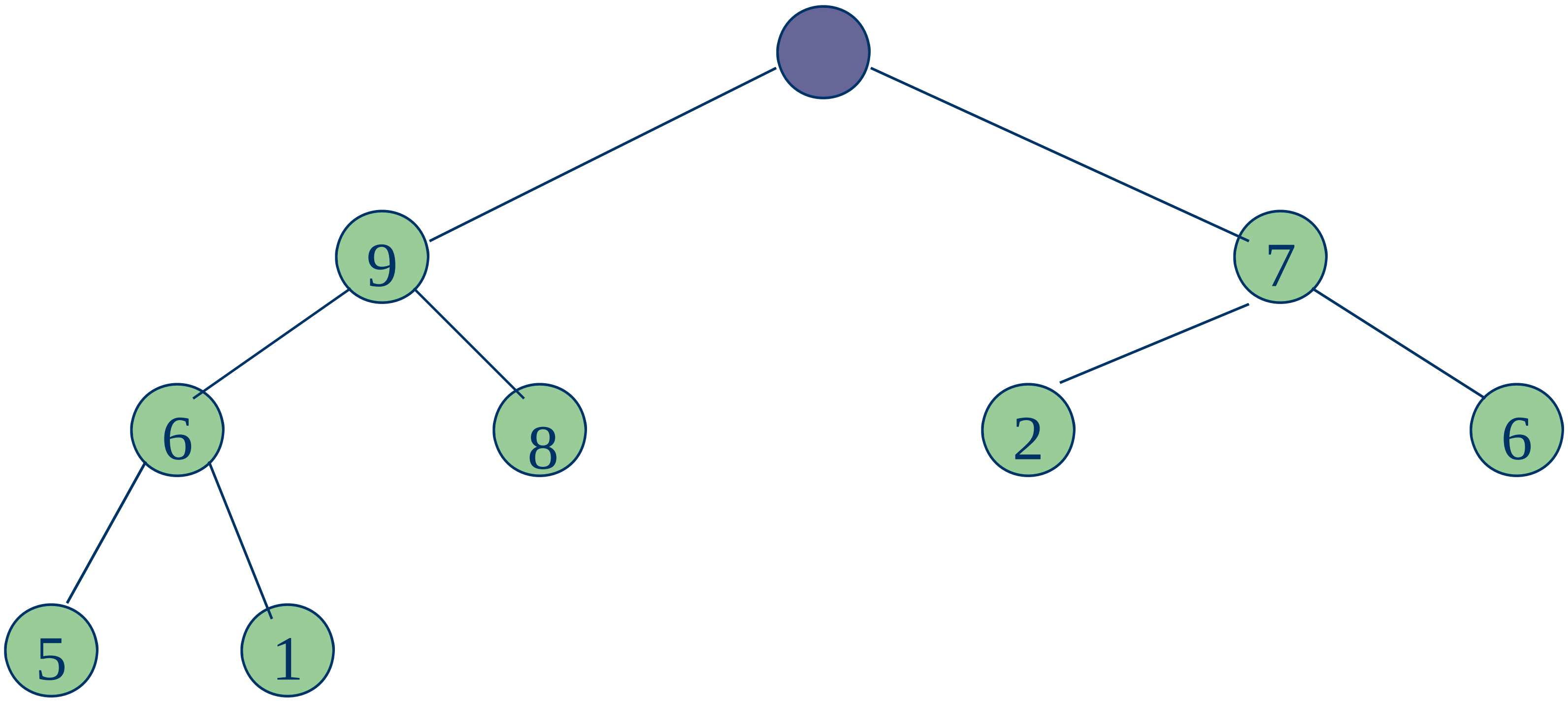
Removing The Max Element



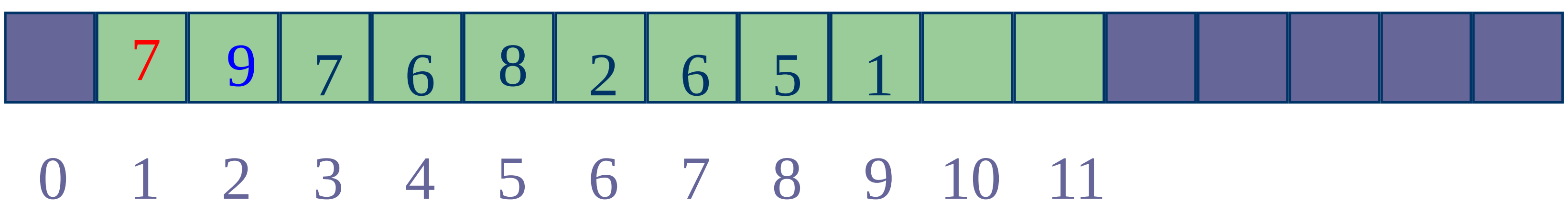
Heap with 9 nodes.



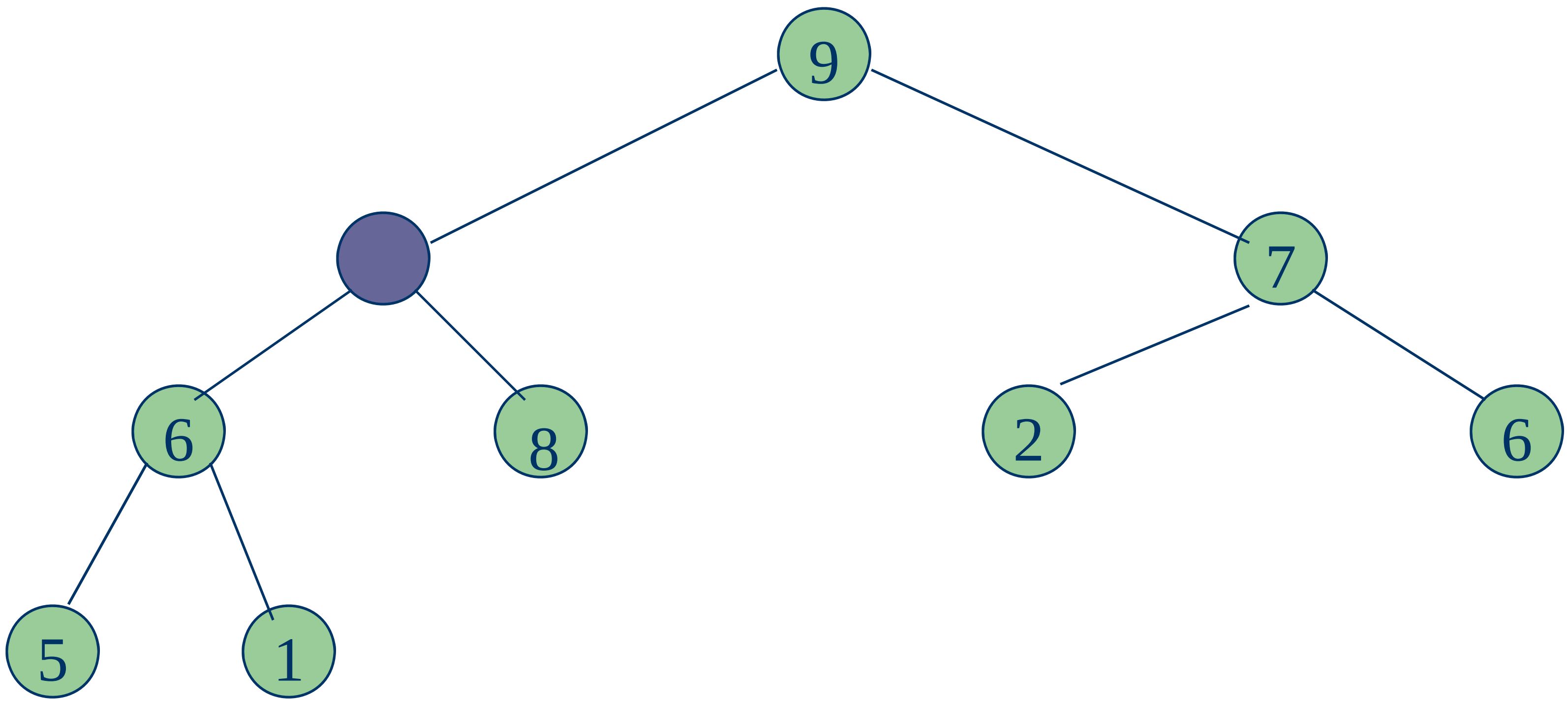
Removing The Max Element



Reinsert 7.



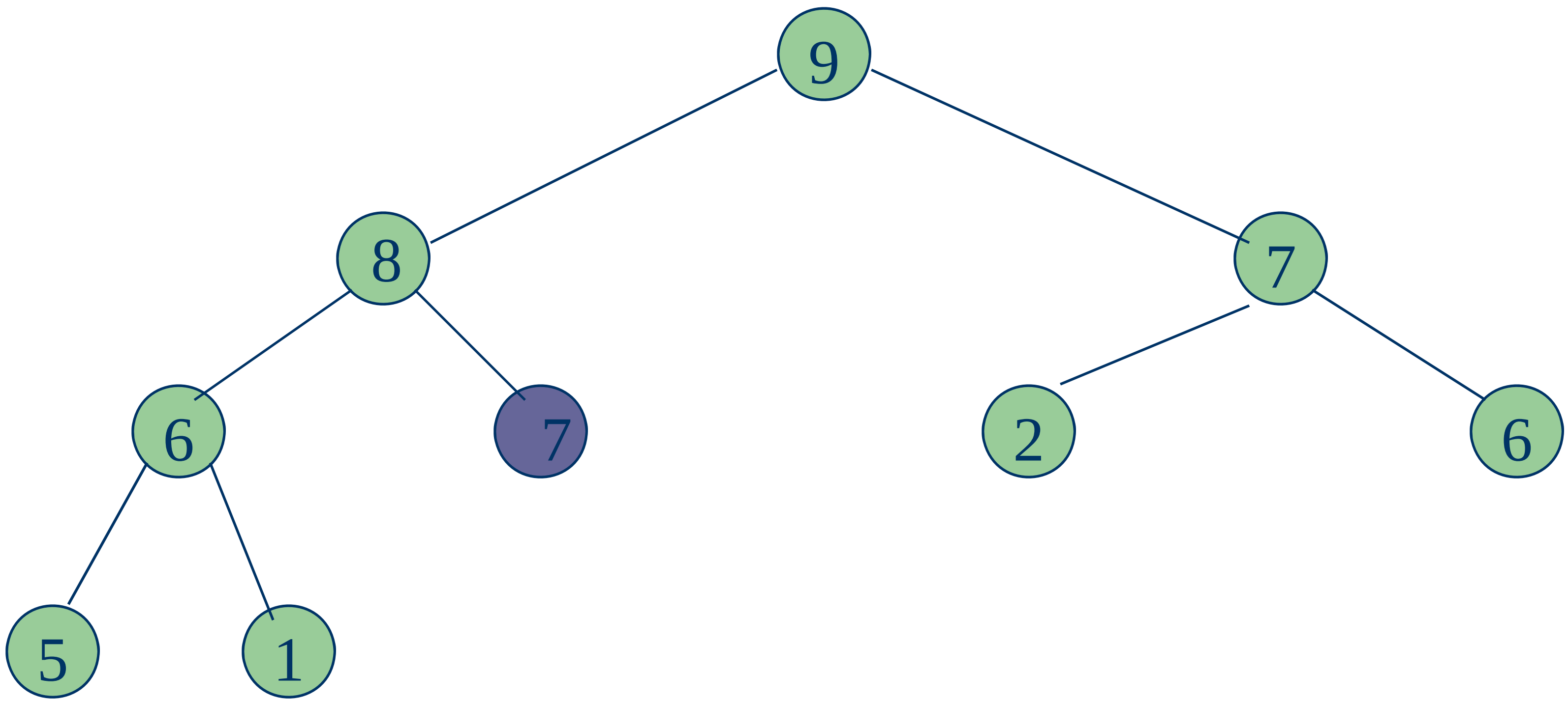
Removing The Max Element



Reinsert 7.

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

Removing The Max Element



Reinsert 7.

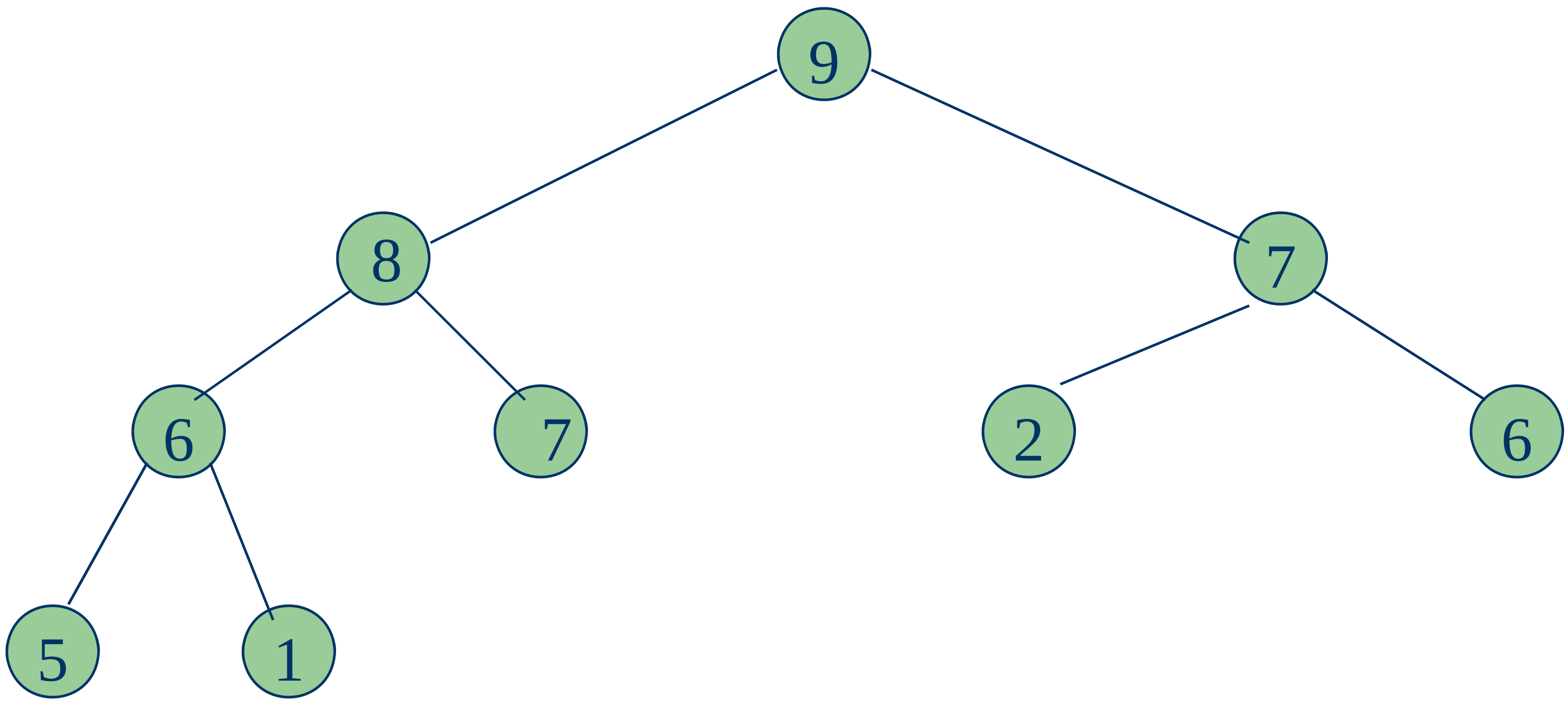
	9	8	7	6	7	2	6	5	1							
0	1	2	3	4	5	6	7	8	9	10	11					

Remove Max

```
HEAP-EXTRACT-MAX(A)
        remove A[1]
        A[1]←A[n]           ; n is HeapSize(A), the length of the heap, not array
        n←n-1               ; decrease size of heap
        Heapify(A,1,n)      ; Remake heap to conform to heap properties
```

```
Heapify(A,I,n)          ; Array A, heapify node I, heapsize is n
        ; Note that the left and right subtrees of I are also heaps
        ; Make I's subtree be a heap.
        If 2I≤n and A[2I]>A[I]
                ; see which is largest of current node and its children
                then largest←2I
                else largest← I
        If 2I+1≤n and A[2I+1]>A[largest]
                then largest←2I+1
        If largest ≠ I
                then swap A[I]↔ A[largest]
                     Heapify(A,largest,n)
```

Complexity Of Remove Max Element



Complexity is O(log n).