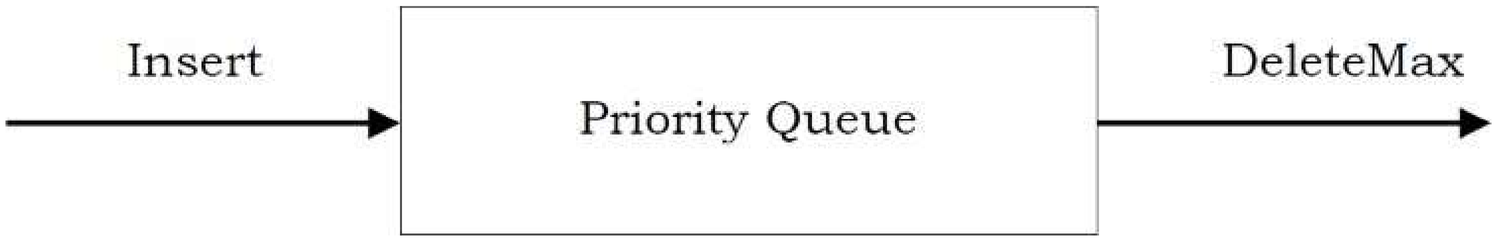
CHAPTER 7

PRIORITY QUEUES AND HEAPS

7.1 What is a Priority Queue?

In some situations we may need to find the minimum/maximum element among a collection of elements. We can do this with the help of Priority Queue ADT. A priority queue ADT is a data structure that supports the operations *Insert* and *DeleteMin* (which returns and removes the minimum element) or *DeleteMax* (which returns and removes the maximum element).

These operations are equivalent to *EnQueue* and *DeQueue* operations of a queue. The difference is that, in priority queues, the order in which the elements enter the queue may not be the same in which they were processed. An example application of a priority queue is job scheduling, which is prioritized instead of serving in first come first serve.



A priority queue is called an *ascending – priority* queue, if the item with the smallest key has the highest priority (that means, delete the smallest element always). Similarly, a priority queue is said to be a *descending –priority* queue if the item with the largest key has the highest priority (delete the maximum element always). Since these two types are symmetric we will be concentrating on one of them: ascending-priority queue.

7.2 Priority Queue ADT

The following operations make priority queues an ADT.

Main Priority Queues Operations

A priority queue is a container of elements, each having an associated key.

- Insert (key, data): Inserts data with *key* to the priority queue. Elements are ordered based on key.
- DeleteMin/DeleteMax: Remove and return the element with the smallest/largest key.
- GetMinimum/GetMaximum: Return the element with the smallest/largest key without deleting it.

Auxiliary Priority Queues Operations

- k^{th} - Smallest/k^{th} – Largest: Returns the k^{th} -Smallest/k^{th} –Largest key in priority queue.
- Size: Returns number of elements in priority queue.
- Heap Sort: Sorts the elements in the priority queue based on priority (key).

7.3 Priority Queue Applications

Priority queues have many applications - a few of them are listed below:

- Data compression: Huffman Coding algorithm
- Shortest path algorithms: Dijkstra’s algorithm
- Minimum spanning tree algorithms: Prim’s algorithm
- Event-driven simulation: customers in a line

- Selection problem: Finding k^{th}- smallest element

7.4 Priority Queue Implementations

Before discussing the actual implementation, let us enumerate the possible options.

Unordered Array Implementation

Elements are inserted into the array without bothering about the order. Deletions (DeleteMax) are performed by searching the key and then deleting.

Insertions complexity: O(1). DeleteMin complexity: O(n).

Unordered List Implementation

It is very similar to array implementation, but instead of using arrays, linked lists are used.

Insertions complexity: O(1). DeleteMin complexity: O(n).

Ordered Array Implementation

Elements are inserted into the array in sorted order based on key field. Deletions are performed at only one end.

Insertions complexity: O(n). DeleteMin complexity: O(1).

Ordered List Implementation

Elements are inserted into the list in sorted order based on key field. Deletions are performed at only one end, hence preserving the status of the priority queue. All other functionalities associated with a linked list ADT are performed without modification.

Insertions complexity: O(n). DeleteMin complexity: O(1).

Binary Search Trees Implementation

Both insertions and deletions take O($logn$) on average if insertions are random (refer to *Trees* chapter).

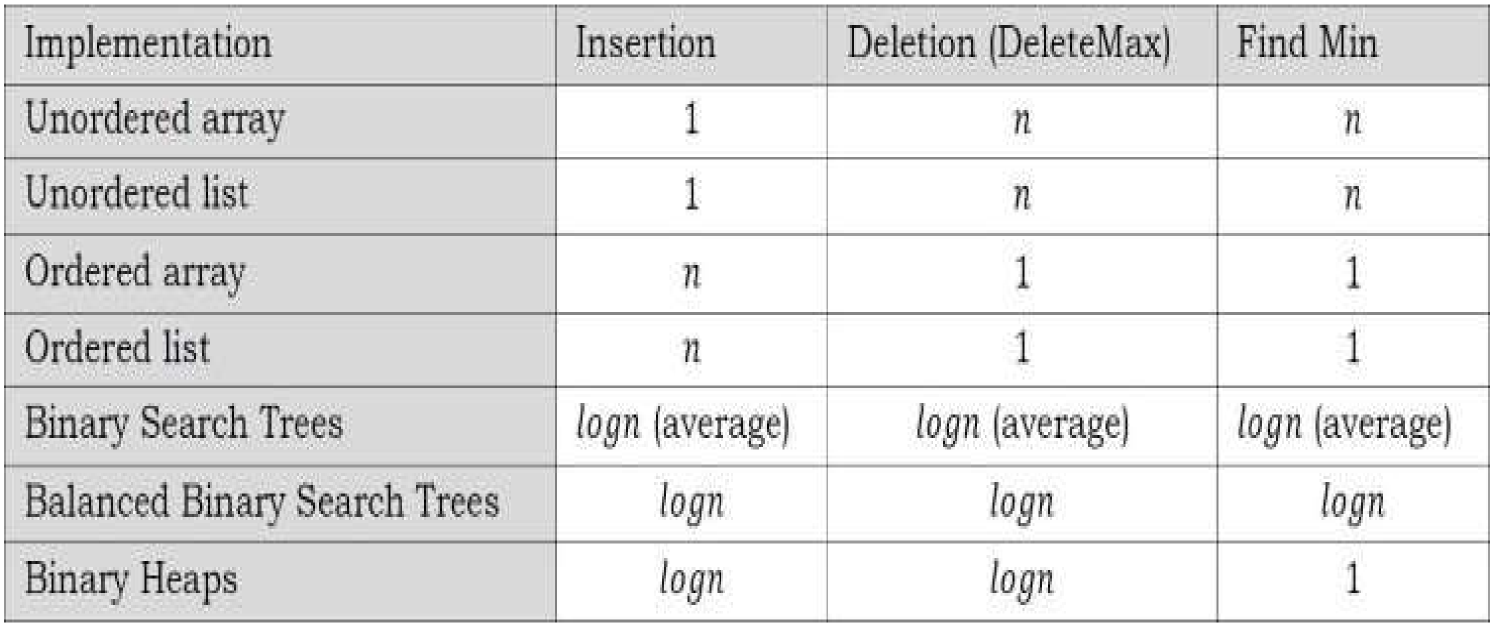
Balanced Binary Search Trees Implementation

Both insertions and deletion take O(*logn*) in the worst case (refer to *Trees* chapter).

Binary Heap Implementation

In subsequent sections we will discuss this in full detail. For now, assume that binary heap implementation gives O(*logn*) complexity for search, insertions and deletions and O(1) for finding the maximum or minimum element.

Comparing Implementations

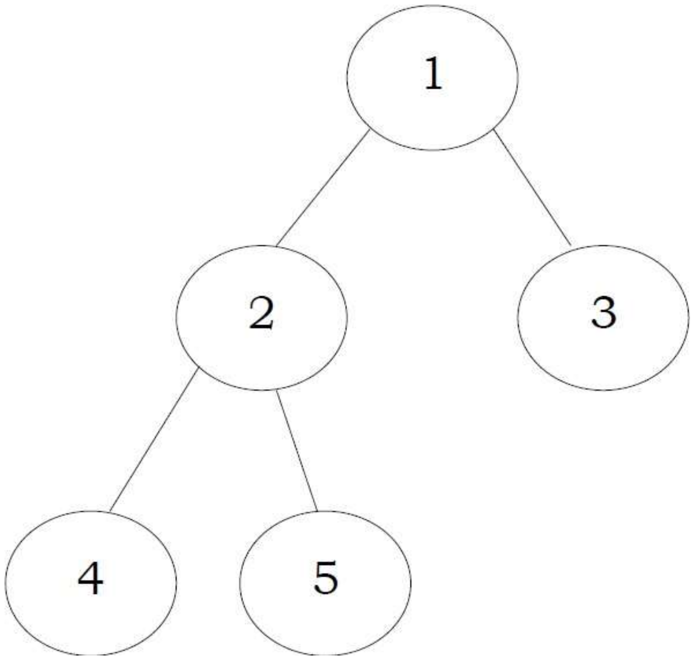


Implementation	Insertion	Deletion (DeleteMax)	Find Min
Unordered array	1	*n*	*n*
Unordered list	1	*n*	*n*
Ordered array	*n*	1	1
Ordered list	*n*	1	1
Binary Search Trees	*logn* (average)	*logn* (average)	*logn* (average)
Balanced Binary Search Trees	*logn*	*logn*	*logn*
Binary Heaps	*logn*	*logn*	1

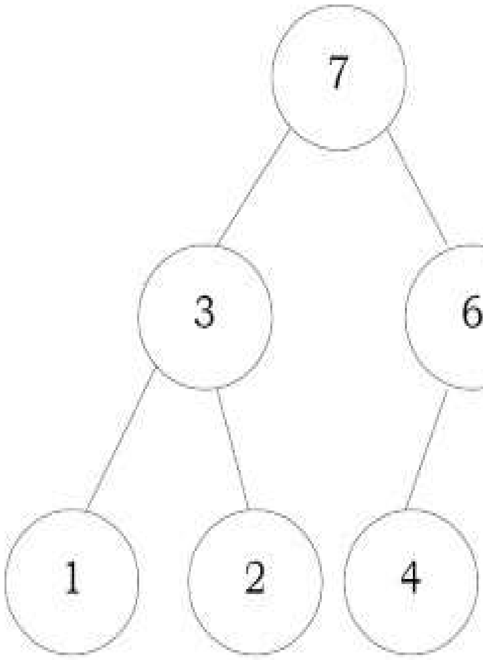
7.5 Heaps and Binary Heaps

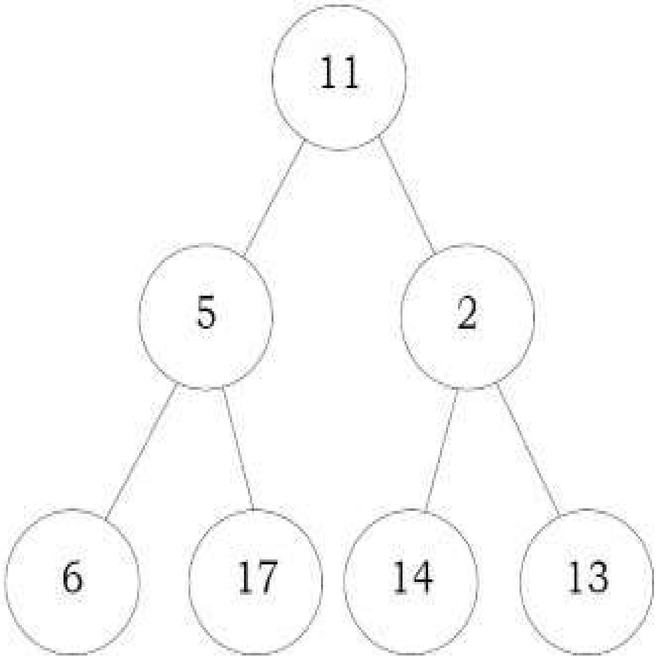
What is a Heap?

A heap is a tree with some special properties. The basic requirement of a heap is that the value of a node must be ≥ (or ≤) than the values of its children. This is called *heap property*. A heap also has the additional property that all leaves should be at h or $h - 1$ levels (where h is the height of the tree) for some $h > 0$ (*complete binary trees*). That means heap should form a *complete binary tree* (as shown below).



In the examples below, the left tree is a heap (each element is greater than its children) and the right tree is not a heap (since 11 is greater than 2).

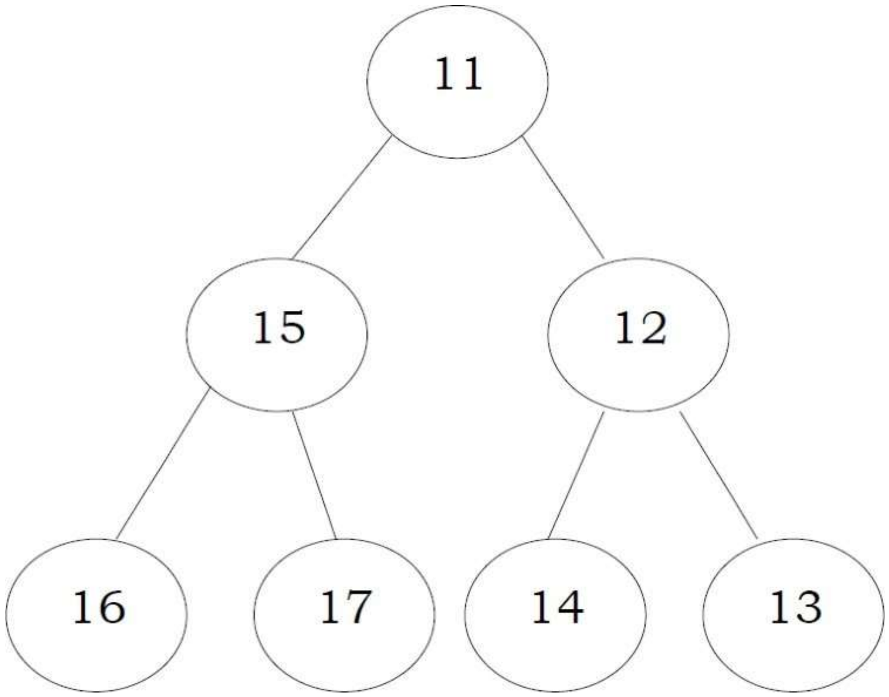




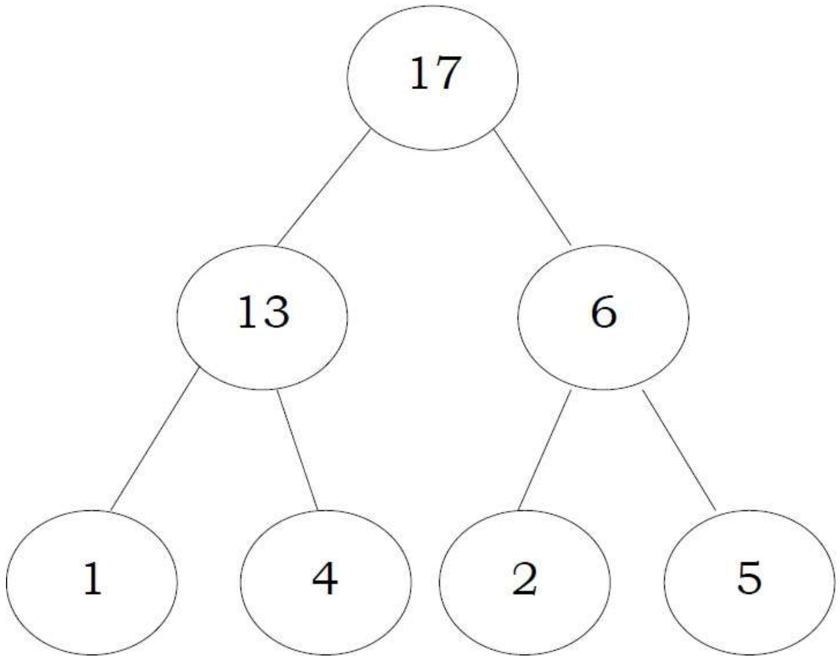
Types of Heaps?

Based on the property of a heap we can classify heaps into two types:

- **Min heap:** The value of a node must be less than or equal to the values of its children



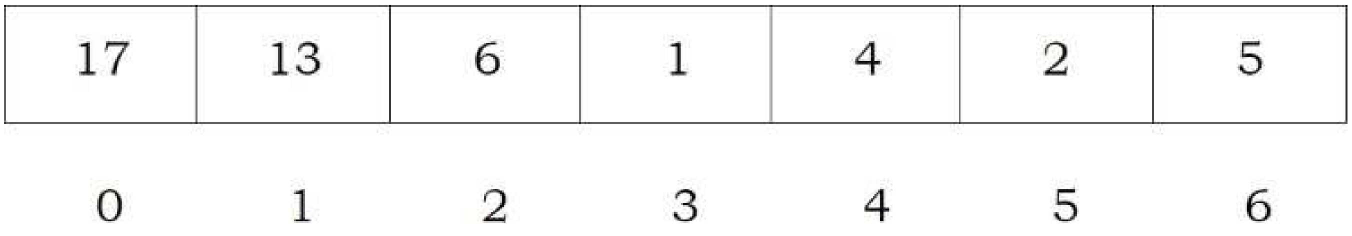
- **Max heap:** The value of a node must be greater than or equal to the values of its children



7.6 Binary Heaps

In binary heap each node may have up to two children. In practice, binary heaps are enough and we concentrate on binary min heaps and binary max heaps for the remaining discussion.

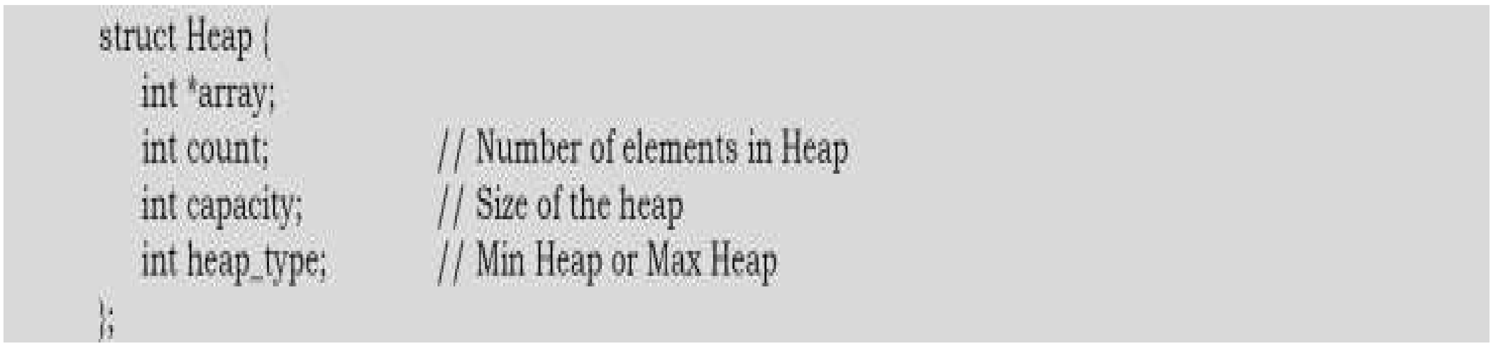
Representing Heaps: Before looking at heap operations, let us see how heaps can be represented. One possibility is using arrays. Since heaps are forming complete binary trees, there will not be any wastage of locations. For the discussion below let us assume that elements are stored in arrays, which starts at index 0. The previous max heap can be represented as:



17	13	6	1	4	2	5
0	1	2	3	4	5	6

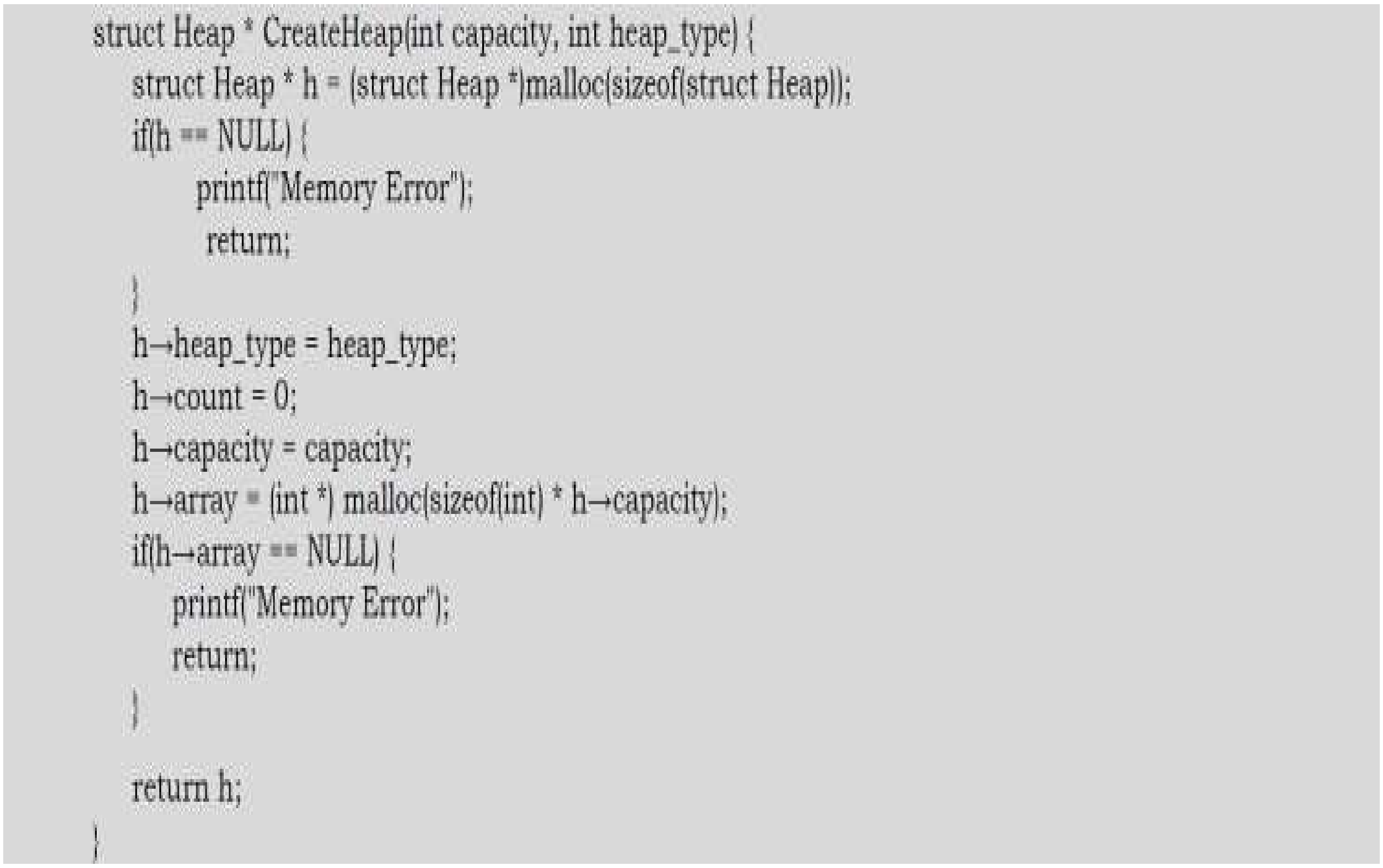
Note: For the remaining discussion let us assume that we are doing manipulations in max heap.

Declaration of Heap



```
struct Heap {
    int *array;
    int count;          // Number of elements in Heap
    int capacity;       // Size of the heap
    int heap_type;      // Min Heap or Max Heap
};
```

Creating Heap



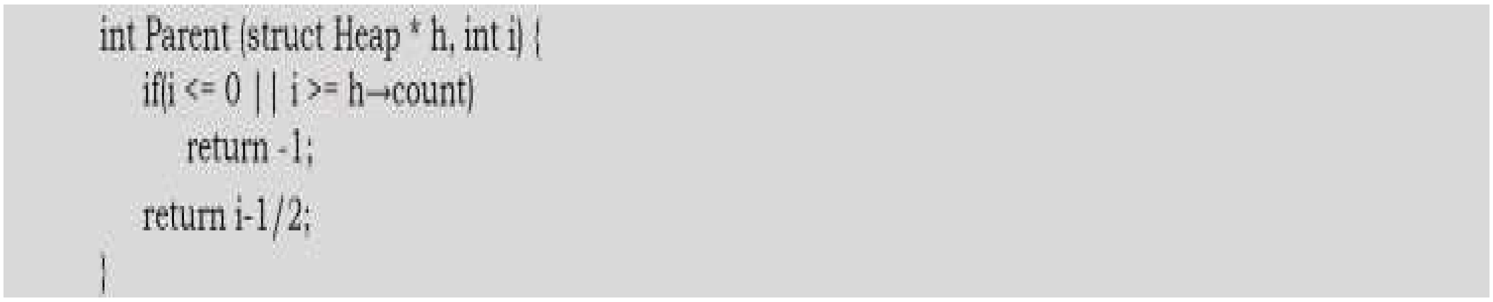
```
struct Heap * CreateHeap(int capacity, int heap_type) {
    struct Heap * h = (struct Heap *)malloc(sizeof(struct Heap));
    if(h == NULL) {
          printf("Memory Error");
           return;
    }
    h→heap_type = heap_type;
    h→count = 0;
    h→capacity = capacity;
    h→array = (int *) malloc(sizeof(int) * h→capacity);
    if(h→array == NULL) {
        printf("Memory Error");
        return;
    }

    return h;
}
```

Time Complexity: O(1).

Parent of a Node

For a node at i^{th} location, its parent is at $\frac{i-1}{2}$ location. In the previous example, the element 6 is at second location and its parent is at 0^{th} location.



```
int Parent (struct Heap * h, int i) {
    if(i <= 0 || i >= h→count)
        return -1;
    return i-1/2;
}
```

Time Complexity: O(1).

Children of a Node

Similar to the above discussion, for a node at i^{th} location, its children are at $2 * i + 1$ and $2 * i +$

2 locations. For example, in the above tree the element 6 is at second location and its children 2 and 5 are at 5 (2 * *i* + 1 = 2 * 2 + 1) and 6(2 * *i* + 2 = 2 * 2) locations.

```
int LeftChild(struct Heap *h, int i) {
    int left = 2 * i + 1;
    if(left >= h→count)
        return -1;
    return left;
}
Time Complexity: O(1).
```

```
int RightChild(struct Heap *h, int i) {
    int right = 2 * i + 2;
    if(right >= h→count)
        return -1;
    return right;
}
Time Complexity: O(1).
```

Getting the Maximum Element

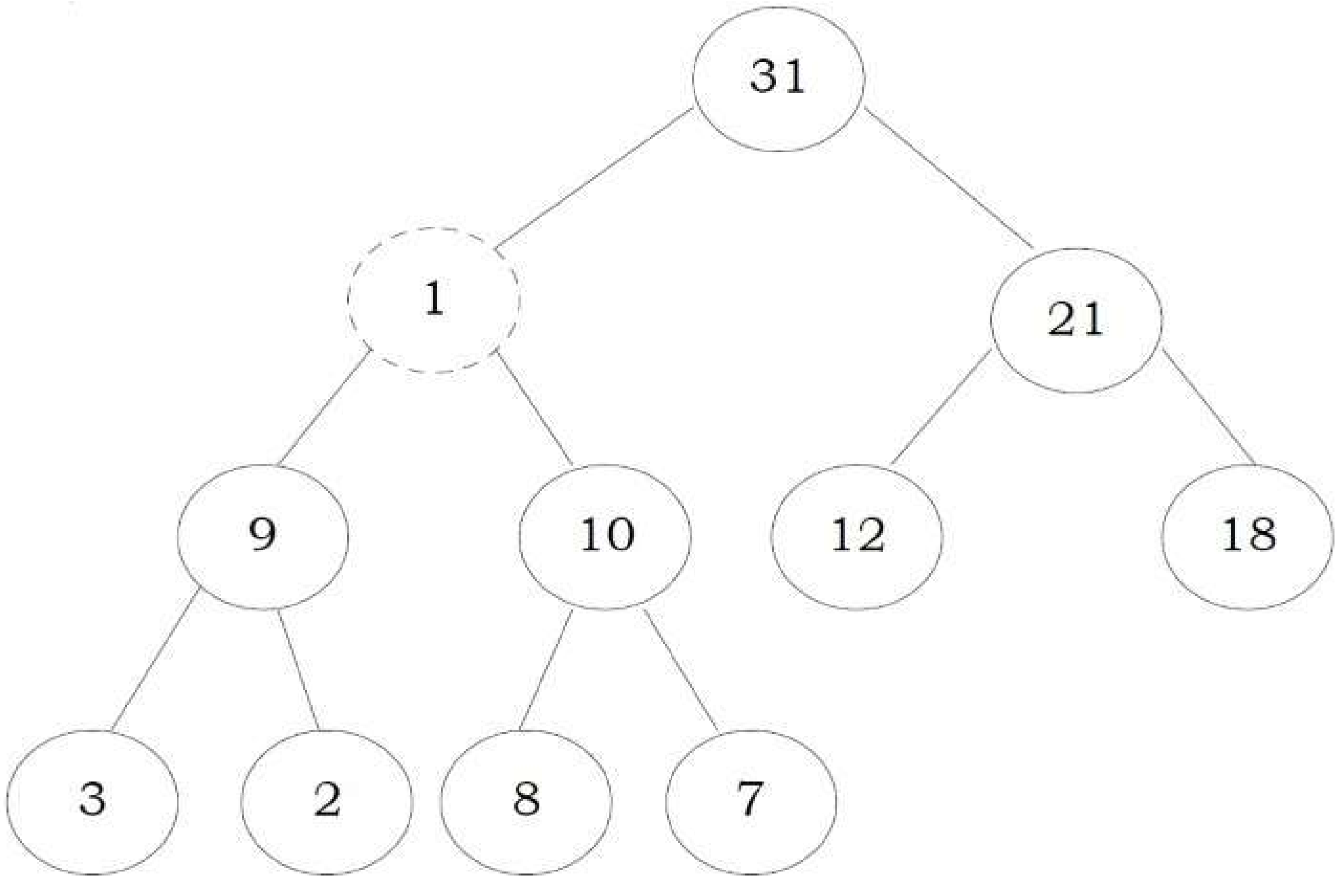
Since the maximum element in max heap is always at root, it will be stored at h→array[O].

```
int GetMaximum(Heap * h) {
    if(h→count == 0)
        return -1;
    return h→array[0];
}
```

Time Complexity: O(1).

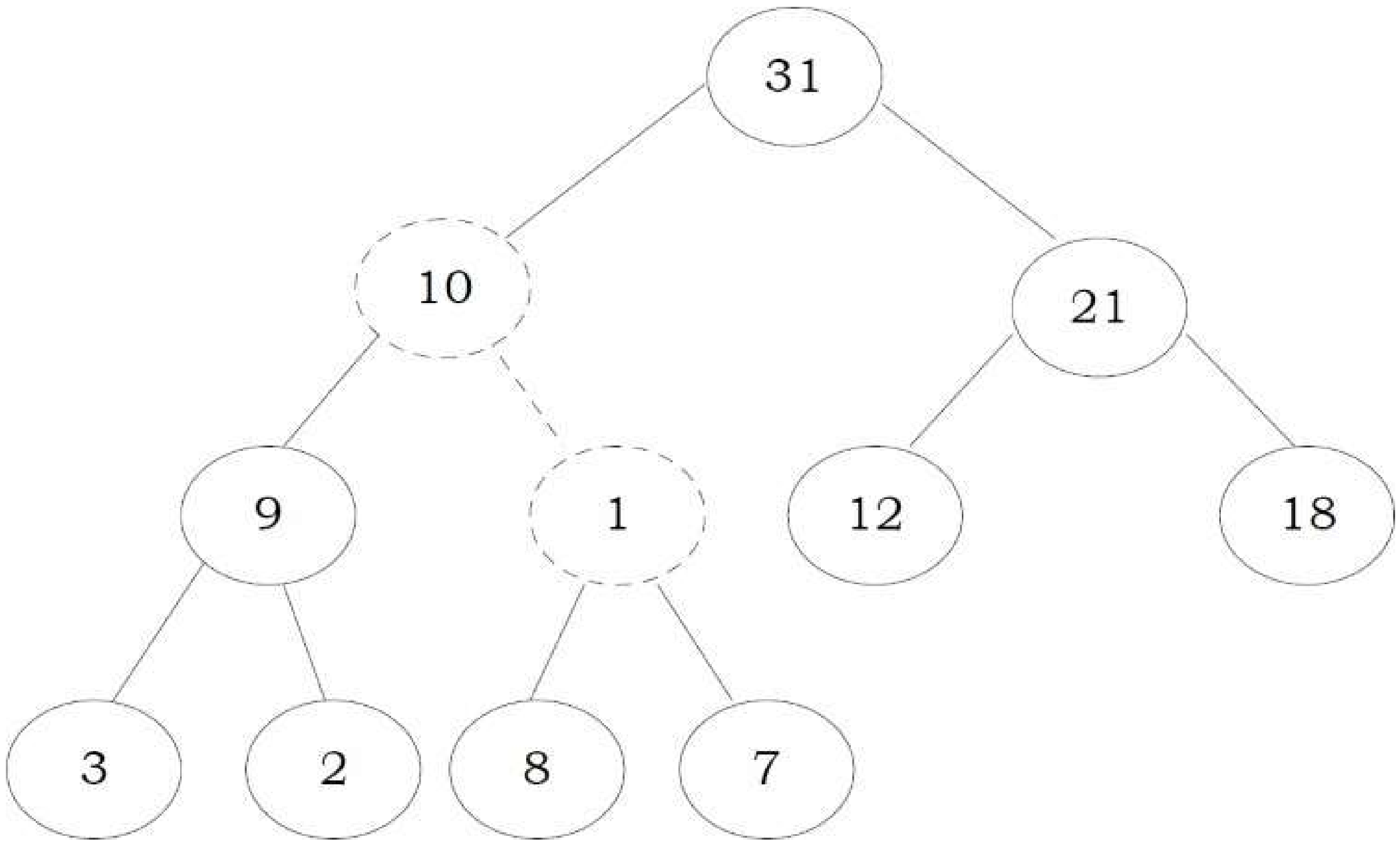
Heapifying an Element

After inserting an element into heap, it may not satisfy the heap property. In that case we need to adjust the locations of the heap to make it heap again. This process is called *heapifying*. In max-heap, to heapify an element, we have to find the maximum of its children and swap it with the current element and continue this process until the heap property is satisfied at every node.

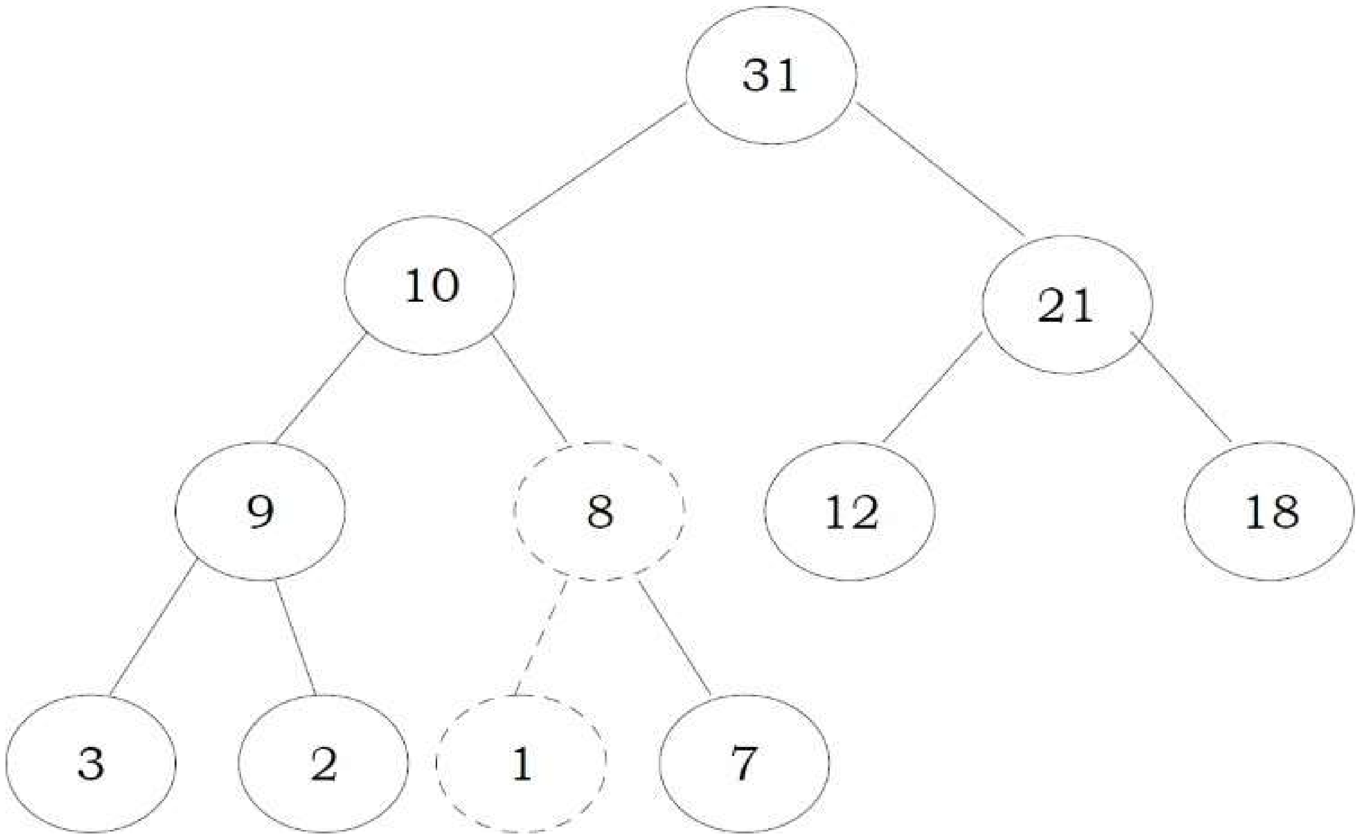


Observation: One important property of heap is that, if an element is not satisfying the heap property, then all the elements from that element to the root will have the same problem. In the example below, element 1 is not satisfying the heap property and its parent 31 is also having the issue. Similarly, if we heapify an element, then all the elements from that element to the root will also satisfy the heap property automatically. Let us go through an example. In the above heap, the element 1 is not satisfying the heap property. Let us try heapifying this element.

To heapify 1, find the maximum of its children and swap with that.



We need to continue this process until the element satisfies the heap properties. Now, swap 1 with 8.



Now the tree is satisfying the heap property. In the above heapifying process, since we are moving from top to bottom, this process is sometimes called *percolate down*. Similarly, if we start heapifying from any other node to root, we can that process *percolate up* as move from bottom to top.

```
//Heapifying the element at location i.
void PercolateDown(struct Heap *h, int i) {
    int l, r, max, temp;
    l = LeftChild(h, i);
    r = RightChild(h, i);
    if(l != -1 && h→array[l] > h→array[i])
        max = l;
    else
        max = i;
    if(r != -1&& h→array[r] > h→array[max])
        max = r;
    if(max != i) {
        //Swap h→array[i] and  h→array[max];
        temp = h→array[i];
        h→array[i] = h→array[max];
        h→array[max] = temp;
    }
    PercolateDown(h, max);
}
```

Time Complexity: O(*logn*). Heap is a complete binary tree and in the worst case we start at the root and come down to the leaf. This is equal to the height of the complete binary tree. Space Complexity: O(1).

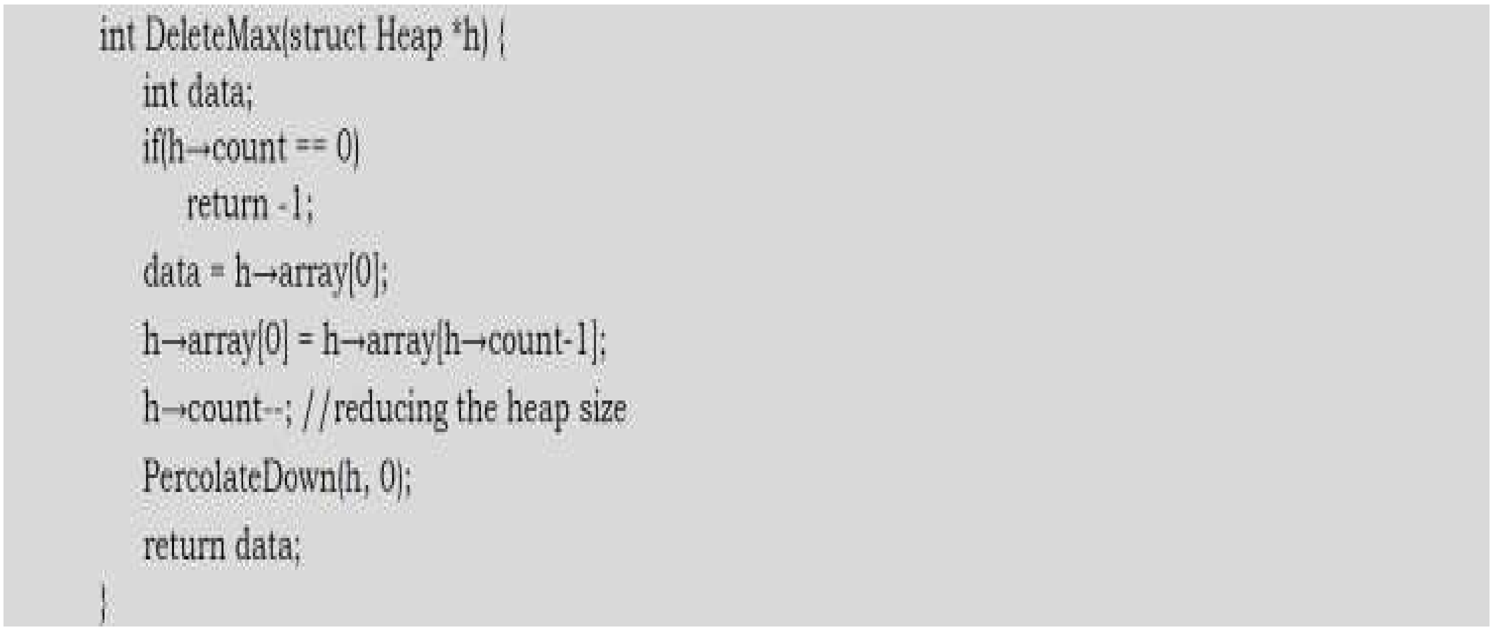
Deleting an Element

To delete an element from heap, we just need to delete the element from the root. This is the only operation (maximum element) supported by standard heap. After deleting the root element, copy the last element of the heap (tree) and delete that last element.

After replacing the last element, the tree may not satisfy the heap property. To make it heap again, call the *PercolateDown* function.

- Copy the first element into some variable

- Copy the last element into first element location
- *PercolateDown* the first element



```
int DeleteMax(struct Heap *h) {
    int data;
    if(h→count == 0)
        return -1;
    data = h→array[0];
    h→array[0] = h→array[h→count-1];
    h→count--; //reducing the heap size
    PercolateDown(h, 0);
    return data;
}
```

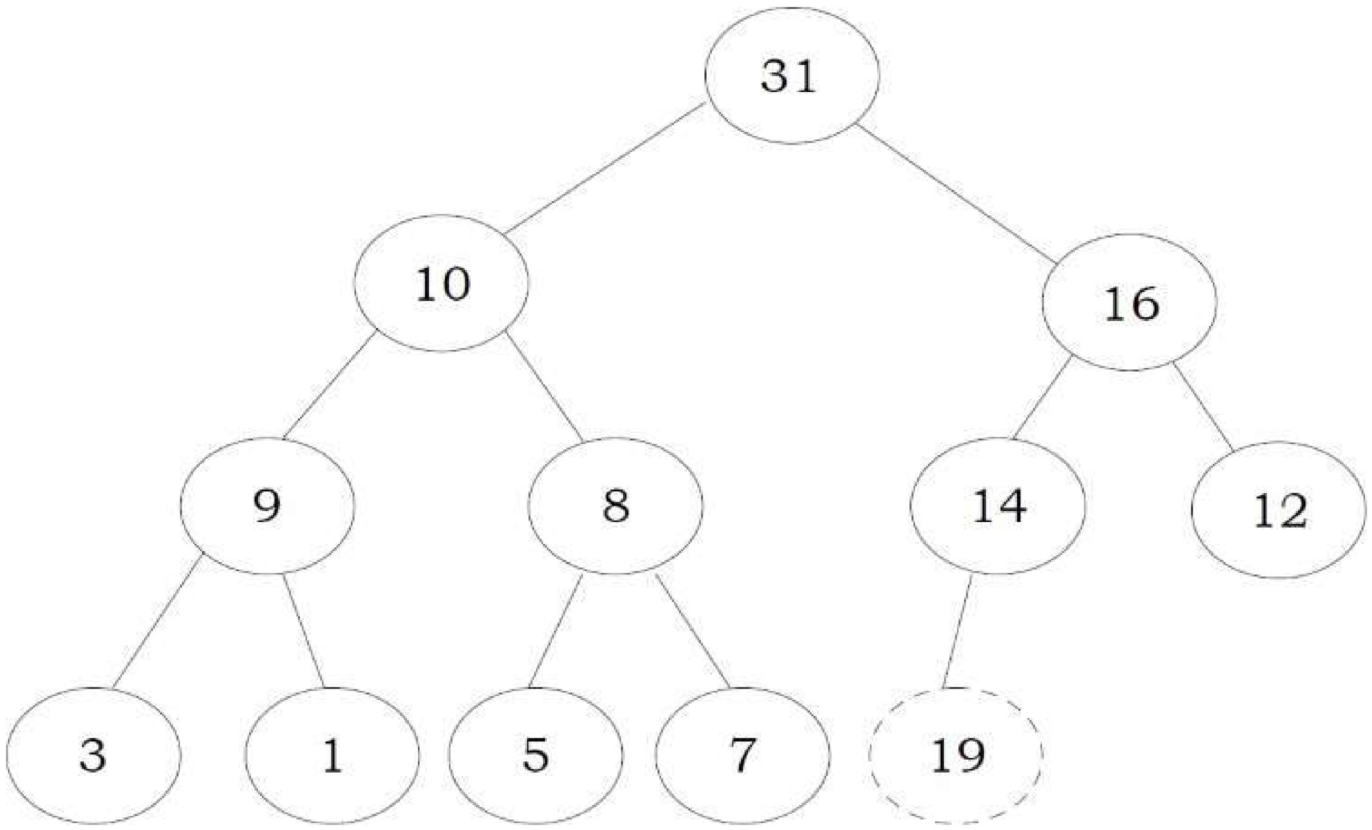
Note: Deleting an element uses *PercolateDown*, and inserting an element uses *PercolateUp*. Time Complexity: same as *Heapify* function and it is O(*logn*).

Inserting an Element

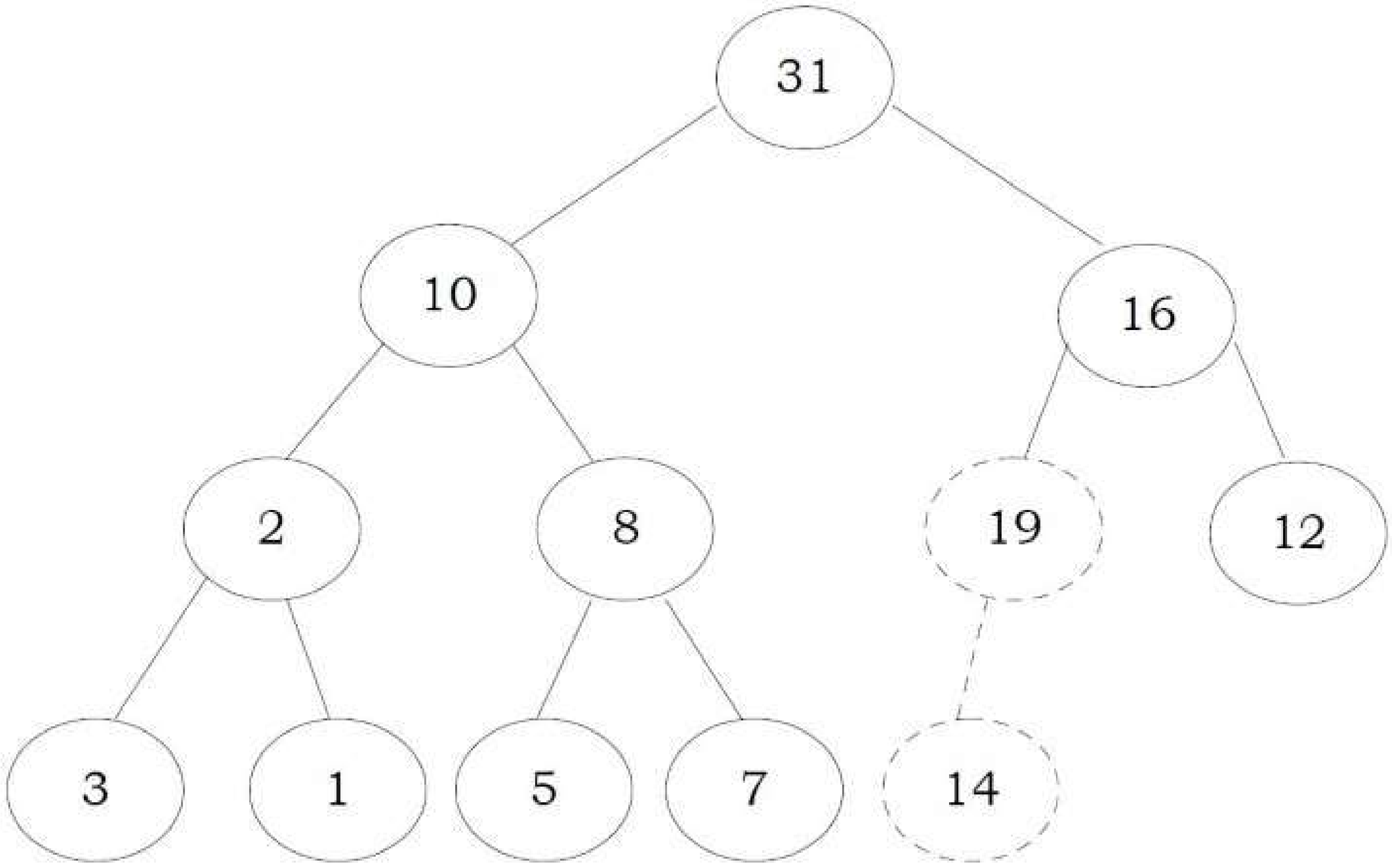
Insertion of an element is similar to the heapify and deletion process.

- Increase the heap size
- Keep the new element at the end of the heap (tree)
- Heapify the element from bottom to top (root)

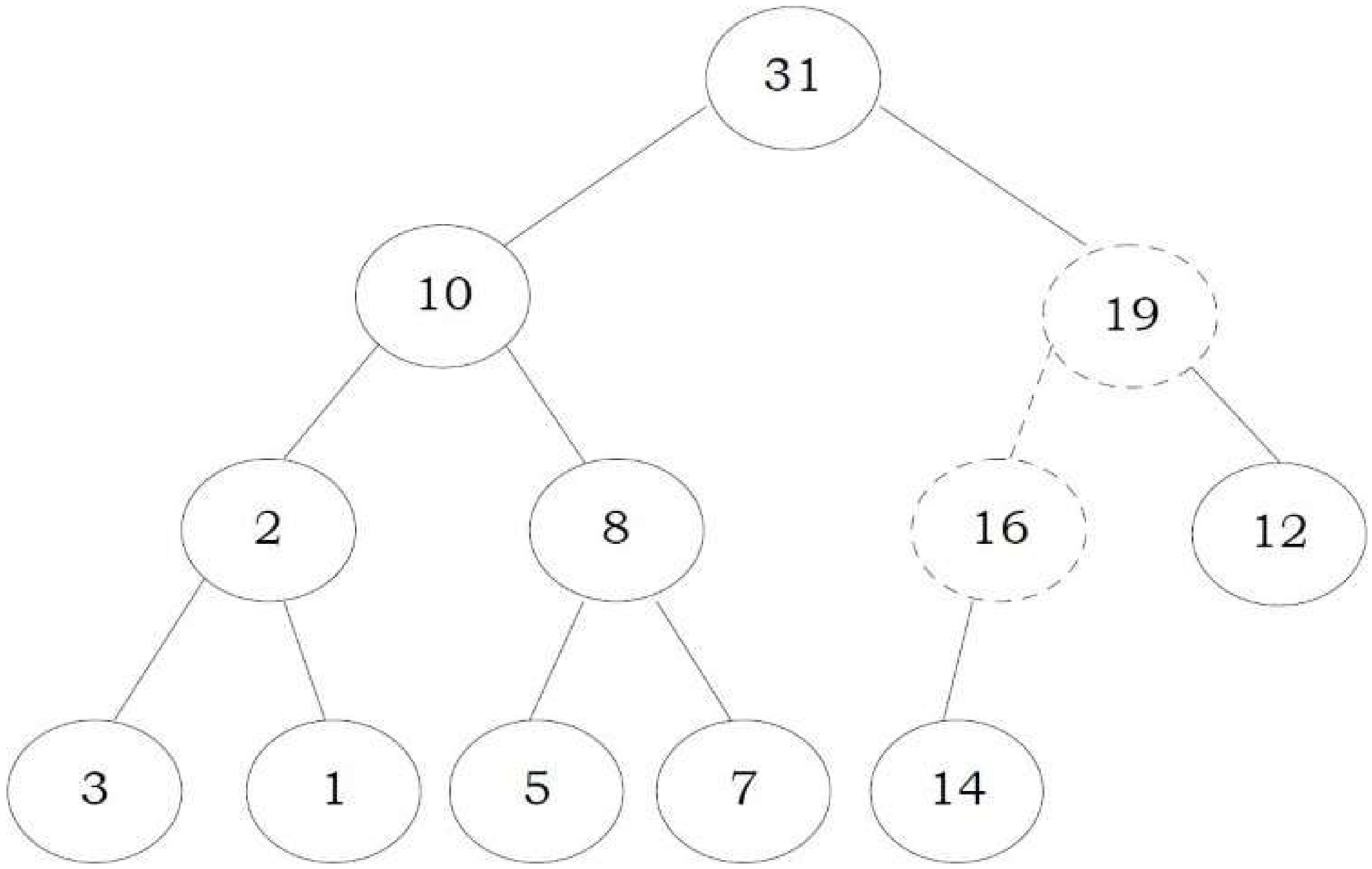
Before going through code, let us look at an example. We have inserted the element 19 at the end of the heap and this is not satisfying the heap property.



In order to heapify this element (19), we need to compare it with its parent and adjust them. Swapping 19 and 14 gives:



Again, swap 19 andl6:



Now the tree is satisfying the heap property. Since we are following the bottom-up approach we sometimes call this process *percolate up*.

```
int Insert(struct Heap *h, int data) {
    int i;
    if(h→count == h→capacity)
        ResizeHeap(h);
    h→count++;              //increasing the heap size to hold this new item
    i = h→count-1;
    while(i>=0 && data > h→array[(i-1)/2]) {
        h→array[i] = h→array[(i-1)/2];
        i = i-1/2;
    }
    h→array[i] = data;
}
void ResizeHeap(struct Heap * h) {
    int *array_old = h→array;
    h→array = (int *) malloc(sizeof(int) * h→capacity * 2);
    if(h→array == NULL) {
        printf("Memory Error");
        return;
    }
    for (int i = 0; i < h→capacity; i ++)
        h→array[i] = array_old[i];
    h→capacity *= 2;
    free(array_old);
}
```

Time Complexity: O(*logn*). The explanation is the same as that of the *Heapify* function.

Destroying Heap

```
void DestroyHeap (struct Heap *h) {
    if(h == NULL)
        return;
    free(h→array);
    free(h);
    h = NULL;
}
```

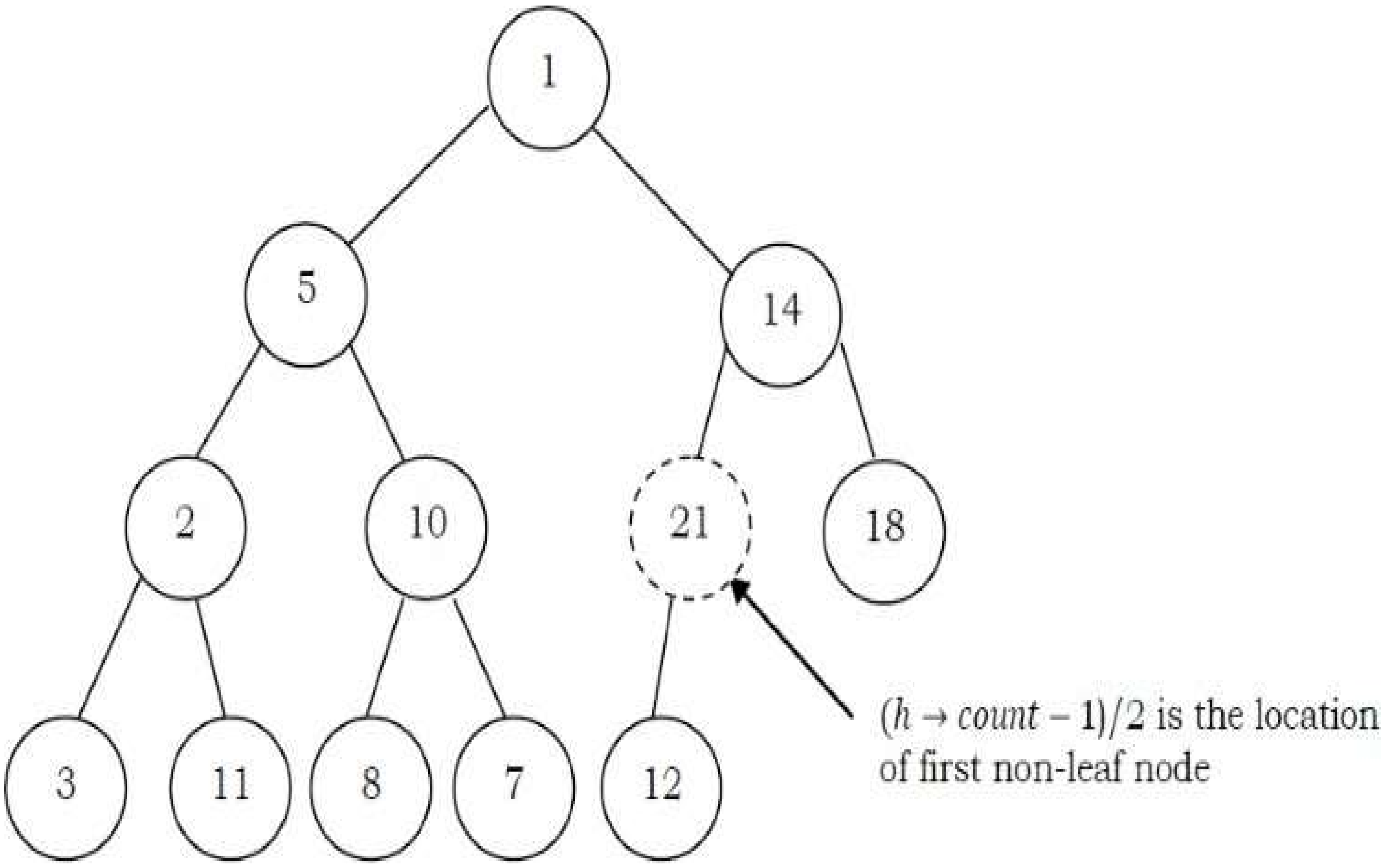
Heapifying the Array

One simple approach for building the heap is, take *n* input items and place them into an empty heap. This can be done with *n* successive inserts and takes O(*nlogn*) in the worst case. This is due to the fact that each insert operation takes O(*logn*).

To finish our discussion of binary heaps, we will look at a method to build an entire heap from a list of keys. The first method you might think of may be like the following. Given a list of keys, you could easily build a heap by inserting each key one at a time. Since you are starting with a list of one item, the list is sorted and you could use binary search to find the right position to insert the next key at a cost of approximately O(*logn*) operations.

However, remember that inserting an item in the middle of the list may require O(*n*) operations to shift the rest of the list over to make room for the new key. Therefore, to insert *n* keys into the heap would require a total of O(*nlogn*) operations. However, if we start with an entire list then we can build the whole heap in O(*n*) operations.

Observation: Leaf nodes always satisfy the heap property and do not need to care for them. The leaf elements are always at the end and to heapify the given array it should be enough if we heapify the non-leaf nodes. Now let us concentrate on finding the first non-leaf node. The last element of the heap is at location $h \rightarrow count - 1$, and to find the first non-leaf node it is enough to find the parent of the last element.


1
5
14
2
10
21
18
3
11
8
7
12
$(h \rightarrow count - 1)/2$ is the location of first non-leaf node

```
void BuildHeap(struct Heap *h, int A[], int n) {
    if(h == NULL)
        return;
    while (n > h→capacity)
        ResizeHeap(h);
    for (int i = 0; i < n; i ++)
        h→array[i] = A[i];
    h→count = n;
    for (int i = (n-1)/2; i >=0; i --)
        PercolateDown(h, i);
}
```

Time Complexity: The linear time bound of building heap can be shown by computing the sum of the heights of all the nodes. For a complete binary tree of height *h* containing $n = 2^{h+1} - 1$ nodes, the sum of the heights of the nodes is $n - h - 1 = n - logn - 1$ (for proof refer to *Problems Section*). That means, building the heap operation can be done in linear time (O(*n*)) by applying a *PercolateDown* function to the nodes in reverse level order.

7.7 Heapsort

One main application of heap ADT is sorting (heap sort). The heap sort algorithm inserts all elements (from an unsorted array) into a heap, then removes them from the root of a heap until the heap is empty. Note that heap sort can be done in place with the array to be sorted. Instead of deleting an element, exchange the first element (maximum) with the last element and reduce the heap size (array size). Then, we heapify the first element. Continue this process until the number of remaining elements is one.

```
void Heapsort(int A[], in n) {
    struct Heap *h = CreateHeap(n);
    int old_size,  i, temp;
    BuildHeap(h, A, n);
    old_size = h→count;
    for(i = n-1; i > 0; i--) {
        //h→array [0] is the largest element
        temp = h→array[0];
        h→array[0] = h→array[h→count-1];
        h→array[0] = temp;
        h→count--;
        PercolateDown(h, 0);
    }
    h→count = old_size;
}
```

Time complexity: As we remove the elements from the heap, the values become sorted (since maximum elements are always *root* only). Since the time complexity of both the insertion algorithm and deletion algorithm is O(*logn*) (where *n* is the number of items in the heap), the time complexity of the heap sort algorithm is O(*nlogn*).

7.8 Priority Queues [Heaps]: Problems & Solutions

Problem-1 What are the minimum and maximum number of elements in a heap of height *h?*

Solution: Since heap is a complete binary tree (all levels contain full nodes except possibly the lowest level), it has at most $2^{h+1} – 1$ elements (if it is complete). This is because, to get maximum nodes, we need to fill all the *h* levels completely and the maximum number of nodes is nothing but the sum of all nodes at all *h* levels.

To get minimum nodes, we should fill the $h – 1$ levels fully and the last level with only one element. As a result, the minimum number of nodes is nothing but the sum of all nodes from $h – 1$ levels plus 1 (for the last level) and we get $2^h – 1 + 1 = 2^h$ elements (if the lowest level has just 1 element and all the other levels are complete).

Problem-2 Is there a min-heap with seven distinct elements so that the preorder traversal of it gives the elements in sorted orde?

Solution: Yes. For the tree below, preorder traversal produces ascending order.