

# PRIORITY QUEUE AND HEAPS

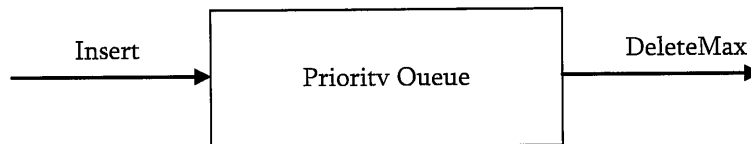
## Chapter-7



### 7.1 What is a Priority Queue?

In some situations we may need to find minimum/maximum element among a collection of elements. Priority Queue ADT is the one which support these kinds of operations. 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 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 smallest key has the highest priority (that means, delete smallest element always). Similarly, a priority queue is said to be a *descending – priority* queue if the item with largest key has the highest priority (delete maximum element always). Since these two types are symmetric we will be concentrating on one of them, say, 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 and below are few of them:

- 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  $k^{\text{th}}$ -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 followed by deletion.

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

### Unordered List Implementation

It is very much 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  $O(\log n)$  on average if insertions are random (refer *Trees* chapter).

### Balanced Binary Search Trees Implementation

Both insertions and deletion take  $O(\log n)$  in the worst case (refer *Trees* chapter).

### Binary Heap Implementation

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

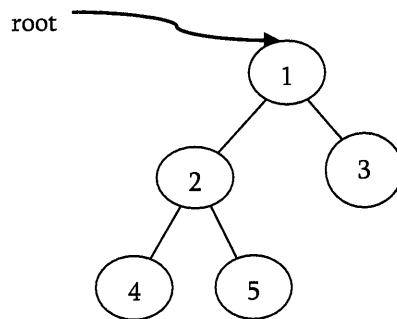
### Comparing Implementations

| 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          | $\log n$ (average) | $\log n$ (average)   | $\log n$ (average) |
| Balanced Binary Search Trees | $\log n$           | $\log n$             | $\log n$           |
| Binary Heaps                 | $\log n$           | $\log n$             | 1                  |

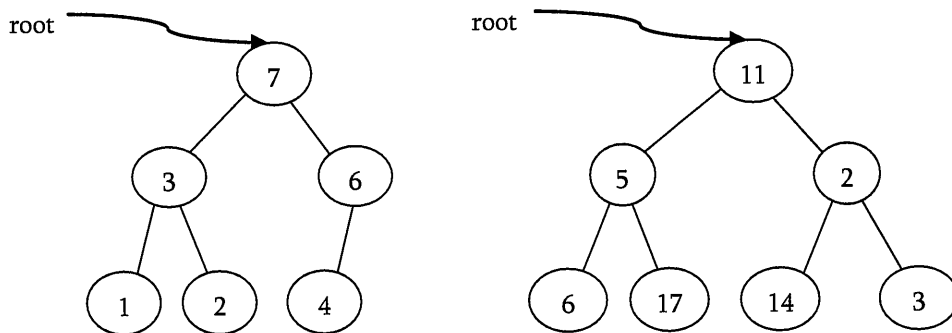
## 7.5 Heaps and Binary Heap

### 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  $\geq$  (or  $\leq$ ) to 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).



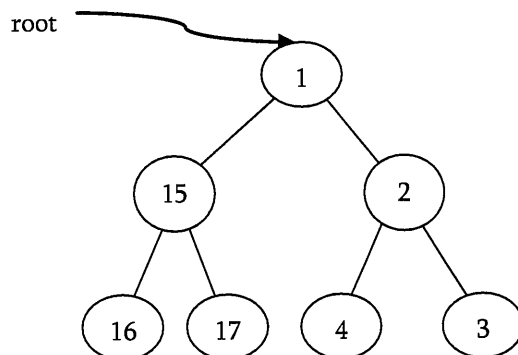
In the below examples, the left tree is a heap (each element is greater than its children) and right tree is not a heap (since, 11 is greater than 2).



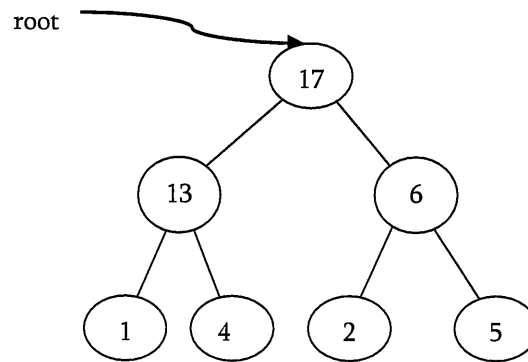
### Types of Heaps?

Based on the heap property we can classify the 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 remaining discussion.

**Representing Heaps:** Before looking at heap operations, let us see how to represent heaps. One possibility is using arrays. Since heaps are forming complete binary trees, there will not be any wastage of locations. For the below discussion 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. For 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 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 ( $2 * i + 2 = 2 * 2 + 2$ ) 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

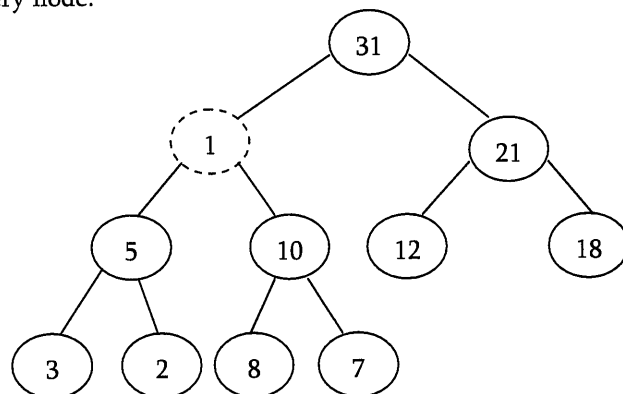
Since the maximum element in max heap is always at root, it will be stored at  $h \rightarrow array[0]$ .

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

Time Complexity:  $O(1)$ .

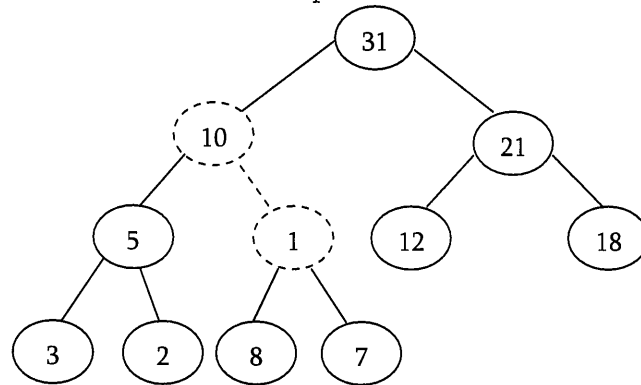
## Heapifying an Element

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.

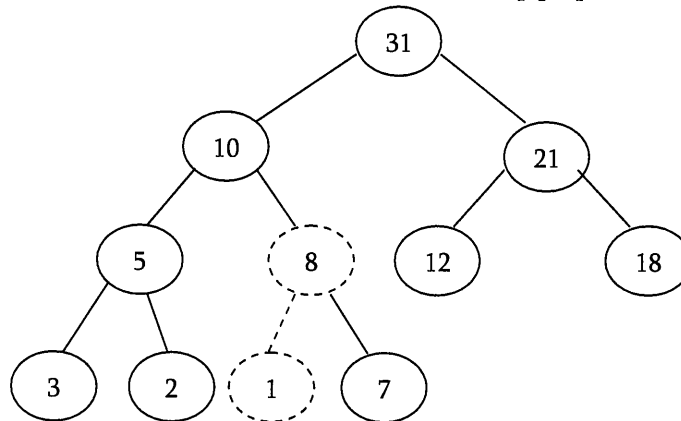


**Observation:** One important property of heap is that, if an element is not satisfying the heap property then all the elements from that element to root will also have the same problem. In below example, element 1 is not satisfying the heap property and its parent 31 is also having the issue. Similarly,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

To heapify 1, find 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 as *percolate down*.

```
//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:  $O(\log n)$ . Heap is a complete binary tree and in the worst we start at root and coming down till 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 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 *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 *percolate down*.

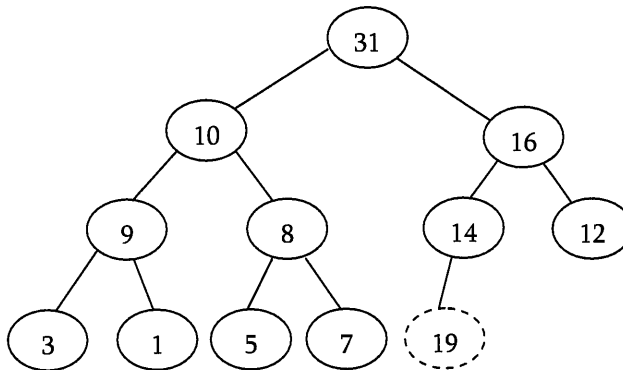
Time Complexity: same as Heapify function and it is  $O(\log n)$ .

## Inserting an Element

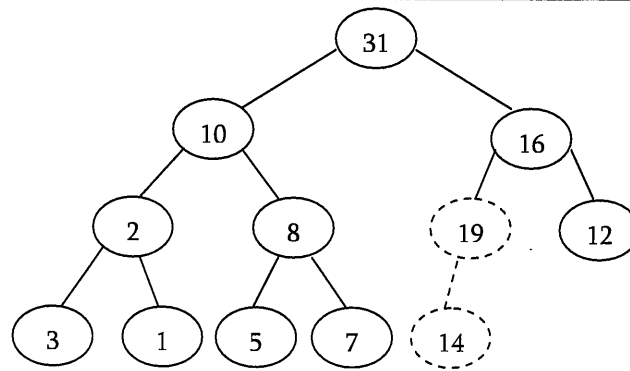
Insertion of an element is very much similar to heapify and deletion process.

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

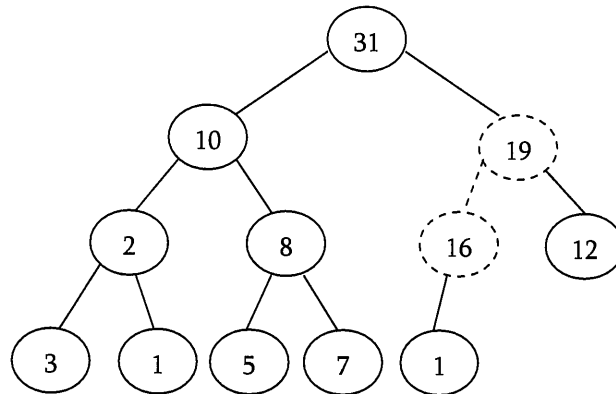
Before going through code, let us take an example. We have inserted the element 19 at the end of the heap and it's not satisfying the heap property.



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 and 16:



Now the tree is satisfying the heap property. Since we are following the bottom-up approach we call this process is sometimes called as *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:  $O(\log n)$ . The explanation is same as that of Heapify function.



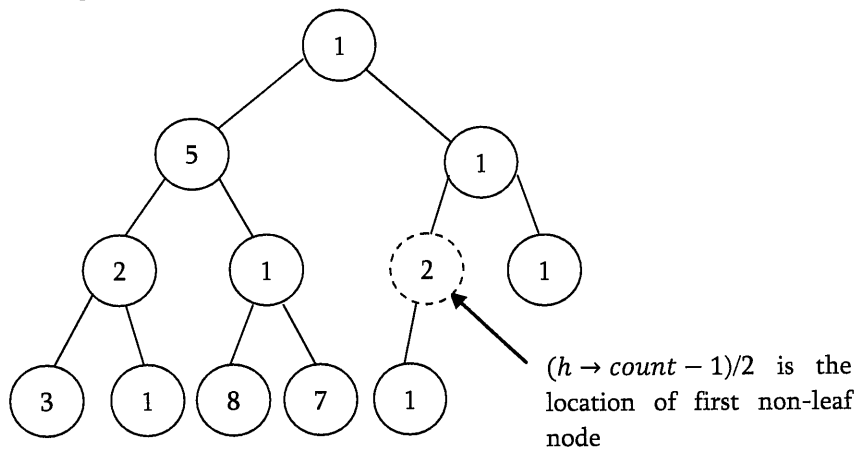
## Destroying Heap

```
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  $O(n \log n)$  in worst case. This is due to the fact that each insert operation takes  $O(\log n)$ .

**Observation:** Leaf nodes always satisfy the heap property and do not need to care for them. The leaf elements are always at the ending 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  $h \rightarrow \text{count} - 1$ , and to find the first non-leaf node it is enough to find the parent of last element.



```
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  $n - h - 1 = n - \log n - 1$  (for proof refer *Problems Section*). That means, building heap operation can be done in linear time ( $O(n)$ ) by applying a *PercolateDown* function to nodes in reverse level order.

## Heapsort

One main application of heap ADT is sorting (heap sort). Heap sort algorithm inserts all elements (from an unsorted array) into a heap, then remove 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 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[h->count-1] = temp;
        h->count--;
        PercolateDown(h, i);
    }
    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 complexities of both the insertion algorithm and deletion algorithms is  $O(\log n)$  (where  $n$  is the number of items in the heap), the time complexity of the heap sort algorithm is  $O(n \log n)$ .

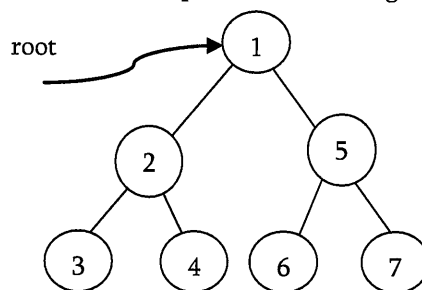
## 7.7 Problems on Priority Queues [Heaps]

**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 sum of all nodes at all  $h$  levels. To get minimum nodes, we should fill the  $h - 1$  levels fully and last level with only one element. As a result, the minimum number of nodes is nothing but sum of all nodes from  $h - 1$  levels plus 1 (for 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).

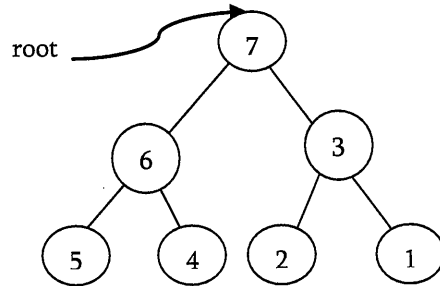
**Problem-2** Is there a min-heap with seven distinct elements so that, the preorder traversal of it gives the elements in sorted order?

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



**Problem-3** Is there a max-heap with seven distinct elements so that, the preorder traversal of it gives the elements in sorted order?

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

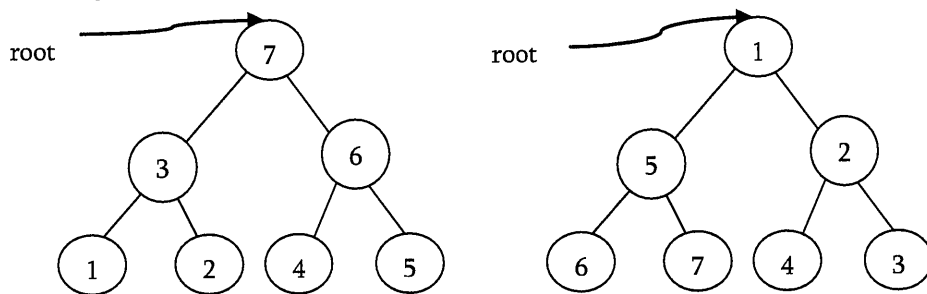


**Problem-4** Is there a min-heap/max-heap with seven distinct elements so that, the inorder traversal of it gives the elements in sorted order?

**Solution:** No, since a heap must be either a min-heap or a max-heap, the root will hold the smallest element or the largest. An inorder traversal will visit the root of tree as its second step, which is not the appropriate place if trees root contains the smallest or largest element.

**Problem-5** Is there a min-heap/max-heap with seven distinct elements so that, the postorder traversal of it gives the elements in sorted order?

**Solution:** Yes, if tree is a max-heap and we want descending order (below left), or if tree is a min-heap and we want ascending order (below right).

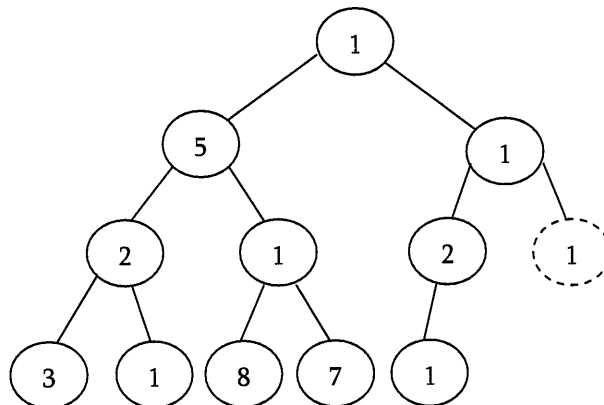


**Problem-6** Show that the height of a heap with  $n$  elements is  $\log n$ ?

**Solution:** A heap is a complete binary tree. All the levels, except the lowest, are completely full. A heap has at least  $2^h$  element and atmost elements.  $2^h \leq n \leq 2^{h+1} - 1$ . This implies,  $h \leq \log n \leq h + 1$ . Since  $h$  is integer,  $h = \log n$ .

**Problem-7** Given a min-heap, give an algorithm for finding the maximum element.

**Solution:** For a given min heap the maximum element will always be at leaf only. Now, the next question is how to find the leaf nodes in tree?



If we carefully observe, the next node of last elements parent is the first leaf node. Since the last element is always at  $h \rightarrow count - 1^{th}$  location, the next node of its parent (parent at location  $\frac{h \rightarrow count - 1}{2}$ ) can be calculated as:

$$\frac{h \rightarrow \text{count} - 1}{2} + 1 \approx \frac{h \rightarrow \text{count} + 1}{2}$$

Now, the only step remaining is scanning the leaf nodes and finding the maximum among them.

```
int FindMaxInMinHeap(struct Heap *h) {
    int Max = -1;
    for(int i = (h->count+1)/2; i < h->count; i++)
        if(h->array[i] > Max)
            Max = h->array[i];
}
```

Time Complexity:  $O\left(\frac{n}{2}\right) \approx O(n)$ .

**Problem-8** Give an algorithm for deleting an arbitrary element from min heap.

**Solution:** To delete an element, first we need to search for an element. Let us assume that we are using level order traversal for finding the element. After finding the element we need to follow the DeleteMin process.

Time Complexity = Time for finding the element + Time for deleting an element  
 $= O(n) + O(\log n) \approx O(n)$ . //Time for searching is dominated.

**Problem-9** Give an algorithm for deleting the  $i^{\text{th}}$  indexed element in a given min-heap.

**Solution:**

```
Int Delete(struct Heap *h, int i) {
    int key;
    if(n < i) {
        printf("Wrong position");
        return;
    }
    key = h->array[i];
    h->array[i] = h->array[h->count-1];
    h->count--;
    PercolateDown(h, i);
    return key;
}
```

Time Complexity =  $O(\log n)$ .

**Problem-10** Prove that, for a complete binary tree of height  $h$  the sum of the heights of all nodes is  $O(n - h)$ .

**Solution:** A complete binary tree has  $2^i$  nodes on level  $i$ . Also, a node on level  $i$  has depth  $i$  and height  $h - i$ . Let us assume that  $S$  denotes the sum of the heights of all these nodes and  $S$  can be calculated as:

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

$$S = h + 2(h - 1) + 4(h - 2) + \dots + 2^{h-1}(1)$$

Multiplying with 2 on both sides gives:  $2S = 2h + 4(h - 1) + 8(h - 2) + \dots + 2^h(1)$

Now, subtract  $S$  from  $2S$ :  $2S - S = -h + 2 + 4 + \dots + 2^h \Rightarrow S = (2^{h+1} - 1) - (h - 1)$

But, we already know that the total number of nodes  $n$  in a complete binary tree with height  $h$  is  $n = 2^{h+1} - 1$ . This gives us:  $h = \log(n + 1)$ . Finally, replacing  $2^{h+1} - 1$  with  $n$ , gives:  $S = n - (h - 1) = O(n - \log n) = O(n - h)$ .

**Problem-11** Give an algorithm to find all elements less than some value  $k$  in a binary heap.

**Solution:** Start from the root of the heap. If the value of the root is smaller than  $k$  then print its value and call recursively once for its left child and once for its right child. If the value of a node is greater or equal than  $k$  then the function stops without printing that value.

The complexity of this algorithm is  $O(n)$ , where  $n$  is the total number of nodes in the heap. This bound takes place in the worst case, where the value of every node in the heap will be smaller than  $k$ , so the function has to call each node of the heap.

**Problem-12** Give an algorithm for merging two binary max-heaps. Let us assume that the size of first heap is  $m + n$  and size of second heap is  $n$ .

**Solution:** One simple way of solving this problem is:

- Assume that elements of first array (with size  $m + n$ ) are at the beginning. That means, first  $m$  cells are filled and remaining  $n$  cells are empty.
- Without changing the first heap, just append the second heap and heapify the array.
- Since the total number of elements in the new array are  $m + n$ , each heapify operation takes  $O(\log(m + n))$ .

The complexity of this algorithm is :  $O((m + n)\log(m + n))$ .

**Problem-13** Can we improve the complexity of Problem-12?

**Solution:** Instead of heapifying all the elements of the  $m + n$  array, we can use technique of “building heap with an array of elements (heapifying array)”. We can start at non leaf nodes and heapify them. The algorithm can be given as:

- Assume that elements of first array (with size  $m + n$ ) are at the beginning. That means, first  $m$  cells are filled and remaining  $n$  cells are empty.
- Without changing the first heap, just append the second heap.
- Now, find the first non leaf node and start heapifying from that element.

In theory section, we have already seen that, building a heap with  $n$  elements takes  $O(n)$  complexity. The complexity of merging with this technique is:  $O(m + n)$ .

**Problem-14** Is there an efficient algorithm for merging 2 max-heaps (stored as an array)? Assume both arrays are having  $n$  elements.

**Solution:** The alternative solution for this problem depends on the type of the heap is. If it's a standard heap where every node has up to two children and which gets filled up that the leaves are on a maximum of two different rows, we cannot get better than  $O(n)$  for merge.

There is an  $O(\log m \times \log n)$  algorithm for merging two binary heaps with sizes  $m$  and  $n$ . For  $m = n$ , this algorithm takes  $O(\log^2 n)$  time complexity. We will be skipping it due to its difficulty and scope.

For better merging performance, we can use another variant of binary heap like a *Fibonacci-Heap* which can merge in  $O(1)$  on average (amortized).

**Problem-15** Give an algorithm for finding the  $k^{th}$  smallest element in max-heap.

**Solution:** One simple solution to this problem is: perform deletion  $k$  times from max-heap.

```
int FindKthLargestEle(struct Heap *h, int k) {
    //Just delete first k-1 elements and return the kth element.
    for(int i=0;i<k-1;i++)
        DeleteMin(h);
    return DeleteMin(h);
}
```

Time Complexity:  $O(k\log n)$ . Since we are performing deletion operation  $k$  times and each deletion takes  $O(\log n)$ .

**Problem-16** For the Problem-15, can we improve the time complexity?

**Solution:** Assume that the original min-heap is called *HOrig* and the auxiliary min-heap is named *HAux*. Initially, the element at the top of *HOrig*, the minimum one, is inserted into the *HAux*. Here we don't do the operation of *DeleteMin* with *HOrig*.

```

Heap HOrig;
Heap HAux;
int FindKthLargestEle( int k ) {
    int heapElement;//Assuming heap data is of integers
    int count=1;
    HAux.Insert(HOrig.Min());
    while( true ) {
        //return the minimum element and delete it from the HA heap
        heapElement = HAux.DeleteMin();
        if(++count == k ) {
            return heapElement;
        }
        else { //insert the left and right children in HO into the HA
            HAux.Insert(heapElement.LeftChild());
            HAux.Insert(heapElement.RightChild());
        }
    }
}

```

Every while-loop iteration gives the  $k^{\text{th}}$  smallest element and we need  $k$  loops to get the  $k^{\text{th}}$  smallest elements. Because the size of the auxiliary heap is always less than  $k$ , every while-loop iteration the size of the auxiliary heap increases by one, and the original heap  $HOrig$  has no operation during the finding, the running time is  $O(k \log k)$ .

**Problem-17** Find  $k$  max elements from max heap.

**Solution:** One simple solution to this problem is: build max-heap and perform deletion  $k$  times.

$$T(n) = \text{DeleteMin from heap } k \text{ times} = \Theta(k \log n).$$

**Problem-18** For Problem-17, is there any alternative solution?

**Solution:** We can use the Problem-16 solution. At the end the auxiliary heap contains the  $k$ -largest elements. Without deleting the elements we should keep on adding elements to  $HAux$ .

**Problem-19** How do we implement stack using heap.

**Solution:** To implement a stack using a priority queue PQ (using min heap), let us assume that we are using one extra integer variable  $c$ . Also, assume that  $c$  is initialized equal to any known value (e.g. 0). The implementation of the stack ADT is given below. Here  $c$  is used as the priority while inserting/deleting the elements from PQ.

```

void Push(int element) {
    PQ.Insert(c, element);
    c--;
}
int Pop() {
    return PQ.DeleteMin();
}
int Top() {
    return PQ.Min();
}
int Size() {
    return PQ.Size();
}
int IsEmpty() {
    return PQ.IsEmpty();
}

```

```
}

```

We could also increment  $c$  back when popping.

**Observation:** We could use the negative of the current system time instead of  $c$  (to avoid overflow). The implementation based on this can be given as:

```
void Push(int element) {
    PQ.insert(-gettime(),element);
}

```

**Problem-20** How do we implement Queue using heap?

**Solution:** To implement a queue using a priority queue PQ (using min heap), as similar to stacks simulation, let us assume that we are using one extra integer variable,  $c$ . Also, assume that  $c$  is initialized equal to any known value (e.g. 0). The implementation of the queue ADT is given below. Here the  $c$ , is used as the priority while inserting/deleting the elements from PQ.

```
void Push(int element) {
    PQ.Insert(c, element);
    c++;
}
int Pop() {
    return PQ.DeleteMin();
}
int Top() {
    return PQ.Min();
}
int Size() {
    return PQ.Size();
}
int IsEmpty() {
    return PQ.IsEmpty();
}

```

**Note:** We could also decrement  $c$  back when popping.

**Observation:** We could use just the negative of the current system time instead of  $c$  (to avoid overflow). The implementation based on this can be given as:

```
void Push(int element) {
    PQ.insert(gettime(),element);
}

```

**Note:** The only change is that we need to take positive  $c$  value instead of negative.

**Problem-21** Given a big file containing billions of numbers. How to find maximum 10 numbers from those file?

**Solution:** Always remember that when we are asked about these types of questions where we need to find max  $n$  elements, best data structure to use is priority queues.

One solution for this problem is to divide the data in some sets of 1000 elements (let's say 1000), make a heap of them, and take 10 elements from each heap one by one. Finally heap sort all the sets of 10 elements and take top 10 among those. But the problem in this approach is where to store 10 elements from each heap. That may require a large amount of memory as we have billions of numbers.

Reuse top 10 elements from earlier heap in subsequent elements can solve this problem. That means to take first block of 1000 elements and subsequent blocks of 990 elements each. Initially Heapsort first set of 1000 numbers, took max

10 elements and mix them with 990 elements of  $2^{nd}$  set. Again Heapsort these 1000 numbers (10 from first set and 990 from  $2^{nd}$  set), take 10 max element and mix those with 990 elements of  $3^{rd}$  set. Repeat the same till last set of 990 (or less) elements and take max 10 elements from final heap. Those 10 elements will be your answer.

Time Complexity:  $O(n) = n/1000 \times (\text{complexity of Heapsort 1000 elements})$  Since complexity of heap sorting 1000 elements will be a constant so the  $O(n) = n$  i.e. linear complexity.

**Problem-22 Merge  $k$  sorted lists with total of  $n$  elements:** We are given  $k$  sorted lists with total  $n$  inputs in all the lists. Give an algorithm to merge them into one single sorted.

**Solution:** Since there are  $k$  equal size lists with a total of  $n$  elements, size of each list is  $\frac{n}{k}$ . One simple way of solving this problem is:

- Take the first list and merge it with second list. Since the size of each list is  $\frac{n}{k}$ , this step produces a sorted list with size  $\frac{2n}{k}$ . This is very much similar to merge sort logic. Time complexity of this step is:  $\frac{2n}{k}$ . This is because we need to scan all the elements of both the lists.
- Then, merge the second list output with third list. As a result this step produces the sorted list with size  $\frac{3n}{k}$ . Time complexity of this step is:  $\frac{3n}{k}$ . This is because we need to scan all the elements of both the lists (one with size  $\frac{2n}{k}$  and other with size  $\frac{n}{k}$ ).
- Continue this process until all the lists are merged to one list.

Total time complexity:  $= \frac{2n}{k} + \frac{3n}{k} + \frac{4n}{k} + \dots + \frac{kn}{k} = \sum_{i=2}^n \frac{in}{k} = \frac{n}{k} \sum_{i=2}^n i \approx \frac{n(k^2)}{k} \approx O(nk)$ . Space Complexity:  $O(1)$ .

**Problem-23** For the Problem-22, Can we improve the time complexity?

**Solution:**

- 1 Divide the lists into pairs and merge them. That means, first take two lists at a time and merge them so that the total elements parsed for all lists is  $O(n)$ . This operation gives  $k/2$  lists.
- 2 Repeat step-1 until the number of lists becomes one.

Time complexity: Step-1 executes  $\log k$  times and each operation parses all  $n$  elements in all the lists for making  $k/2$  lists. For example, if we have 8 lists then first pass would make 4 lists by parsing all  $n$  elements. Second pass would make 2 lists by parsing again  $n$  elements and third pass would give 1 list again by parsing  $n$  elements. As a result the total time complexity is  $O(n \log n)$ . Space Complexity:  $O(n)$ .

**Problem-24** For the Problem-23, can we improve the space complexity?

**Solution:** Let us use heaps for reducing the space complexity.

1. Build the max-heap with all first elements from each list in  $O(k)$ .
2. In each step extract the maximum element of the heap and add it at the end of the output.
3. Add the next element from the list of the one extracted. That means, we need to select the next element of the list which contains the extracted element of the previous step.
4. Repeat step-2 and step-3 until all the elements are completed from all the lists.

Time Complexity =  $O(n \log k)$ . At a time we have  $k$  elements max heap and for all  $n$  elements we have to read just the heap in  $\log k$  time so total time =  $O(n \log k)$ . Space Complexity:  $O(k)$  [for Max-heap].

**Problem-25** Given 2 arrays  $A$  and  $B$  each with  $n$  elements. Give an algorithm for finding largest  $n$  pairs  $(A[i], B[j])$ .

**Solution:**

**Algorithm:**

- Heapify  $A$  and  $B$ . This step takes  $O(2n) \approx O(n)$ .
- Then keep on deleting the elements from both the heaps. Each of this step takes  $O(2 \log n) \approx O(\log n)$ .

Total Time complexity:  $O(n \log n)$ .



**Problem-26** **Min-Max heap:** Give an algorithm that supports min and max in  $O(1)$  time, insert, delete min, and delete max in  $O(\log n)$  time. That means, design a data structure which supports the following operations:

| Operation | Complexity  |
|-----------|-------------|
| Init      | $O(n)$      |
| Insert    | $O(\log n)$ |
| FindMin   | $O(1)$      |
| FindMax   | $O(1)$      |
| DeleteMin | $O(\log n)$ |
| DeleteMax | $O(\log n)$ |

**Solution:** This problem can be solved using two heaps. Let us say two heaps are: Minimum-Heap  $H_{\min}$  and Maximum-Heap  $H_{\max}$ . Also, assume that elements in both the arrays are having mutual pointers. That means, an element in  $H_{\min}$  will have a pointer to the same element in  $H_{\max}$  and an element in  $H_{\max}$  will have a pointer to the same element in  $H_{\min}$ .

|                  |  |
|------------------|--|
| <b>Init</b>      | Build $H_{\min}$ in $O(n)$ and $H_{\max}$ in $O(n)$  |
| <b>Insert(x)</b> | Insert $x$ to $H_{\min}$ in $O(\log n)$ . Insert $x$ to $H_{\max}$ in $O(\log n)$ . Update the pointers in $O(1)$                      |
| <b>FindMin()</b> | Return $\text{root}(H_{\min})$ in $O(1)$   |
| <b>FindMax</b>   | Return $\text{root}(H_{\max})$ in $O(1)$   |
| <b>DeleteMin</b> | Delete the minimum from $H_{\min}$ in $O(\log n)$ . Delete the same element from $H_{\max}$ by using the mutual pointer in $O(\log n)$ |
| <b>DeleteMax</b> | Delete the maximum from $H_{\max}$ in $O(\log n)$ . Delete the same element from $H_{\min}$ by using the mutual pointer in $O(\log n)$ |

**Problem-27** **Dynamic median finding.** Design a heap data structure that supports finding the median.

**Solution:** In a set of  $n$  elements, median is the middle element, such that the number of elements smaller than the median is equal to the number of elements larger than the median. If  $n$  is odd, we can find the median by sorting the set and taking the middle element. If  $n$  is even, the median is usually defined as the average of the two middle elements. This algorithm work even when some of the elements in the list are equal. For example, the median of the multiset  $\{1, 1, 2, 3, 5\}$  is 2, and the median of the multiset  $\{1, 1, 2, 3, 5, 8\}$  is 2.5.

“Median heaps” are the variant of heaps that give access to the median element. A median heap can be implemented using two heaps, each containing half the elements. One is a max-heap, containing the smallest elements, the other is a min-heap, containing the largest elements. The size of the max-heap may be equal to the size of the min-heap, if the total number of elements is even. In this case, the median is the average of the maximum element of the max-heap and the minimum element of the min-heap. If there are an odd number of elements, the max-heap will contain one more element than the min-heap. The median in this case is simply the maximum element of the max-heap.

**Problem-28** **Maximum sum in sliding window:** Given array  $A[]$  with sliding window of size  $w$  which is moving from the very left of the array to the very right. Assume that we can only see the  $w$  numbers in the window. Each time the sliding window moves rightwards by one position. For example: The array is  $[1\ 3\ -1\ -3\ 5\ 3\ 6\ 7]$ , and  $w$  is 3.

| Window position         | Max |
|-------------------------|-----|
| $[1\ 3\ -1]$ -3 5 3 6 7 | 3   |
| 1 $[3\ -1\ -3]$ 5 3 6 7 | 3   |
| 1 3 $[-1\ -3\ 5]$ 3 6 7 | 5   |
| 1 3 -1 $[-3\ 5\ 3]$ 6 7 | 5   |
| 1 3 -1 -3 $[5\ 3\ 6]$ 7 | 6   |
| 1 3 -1 -3 5 $[3\ 6\ 7]$ | 7   |

**Input:** A long array  $A[]$ , and a window width  $w$ . **Output:** An array  $B[]$ ,  $B[i]$  is the maximum value of from  $A[i]$  to  $A[i+w-1]$

**Requirement:** Find a good optimal way to get  $B[i]$

**Solution:** Brute force solution is, every time the window is moved, we can search for a total of  $w$  elements in the window.

Time complexity:  $O(nw)$ .

**Problem-29** For Problem-28, can we reduce the complexity?

**Solution:** Yes, we can use heap data structure. This reduces the time complexity to  $O(n \log w)$ . Insert operation takes  $O(\log w)$  time, where  $w$  is the size of the heap. However, getting the maximum value is cheap, it merely takes constant time as the maximum value is always kept in the root (head) of the heap. As the window slides to the right, some elements in the heap might not be valid anymore (range is outside of the current window). How should we remove them? We would need to be somewhat careful here. Since we only remove elements that are out of the window's range, we would need to keep track of the elements' indices too.

**Problem-30** For Problem-28, can we further reduce the complexity?

**Solution:** Yes, The double-ended queue is the perfect data structure for this problem. It supports insertion/deletion from the front and back. The trick is to find a way such that the largest element in the window would always appear in the front of the queue. How would you maintain this requirement as you push and pop elements in and out of the queue?

Besides, you might notice that there are some redundant elements in the queue that we shouldn't even consider about. For example, if the current queue has the elements: [10 5 3], and a new element in the window has the element 11. Now, we could have emptied the queue without considering elements 10, 5, and 3, and insert only element 11 into the queue.

A natural way most people would think is to try to maintain the queue size the same as the window's size. Try to break away from this thought and try to think outside of the box. Removing redundant elements and storing only elements that need to be considered in the queue is the key to achieve the efficient  $O(n)$  solution below. This is because each element in the list is being inserted and then removed at most once. Therefore, the total number of insert + delete operations is  $2n$ .

```
void MaxSlidingWindow(int A[], int n, int w, int B[]) {
    struct DoubleEndQueue *Q = CreateDoubleEndQueue();
    for (int i = 0; i < w; i++) {
        while (!IsEmptyQueue(Q) && A[i] >= A[QBack(Q)])
            PopBack(Q);
        PushBack(Q, i);
    }
    for (int i = w; i < n; i++) {
        B[i-w] = A[QFront(Q)];
        while (!IsEmptyQueue(Q) && A[i] >= A[QBack(Q)])
            PopBack(Q);
        while (!IsEmptyQueue(Q) && QFront(Q) <= i-w)
            PopFront(Q);
        PushBack(Q, i);
    }
    B[n-w] = A[QFront(Q)];
}
```