adding/deleting a single node, this can only increase/decrease the height of some subtree by 1. So, if the AVL tree property is violated at a node X, it means that the heights of left(X) and right(X) differ by exactly 2. This is because, if we balance the AVL tree every time, then at any point, the difference in heights of left(X) and right(X) differ by exactly 2. Rotations is the technique used for restoring the AVL tree property. That means, we need to apply the rotations for the node X.

Observation: One important observation is that, after an insertion, only nodes that are on the path from the insertion point to the root might have their balances altered because only those nodes have their subtrees altered. To restore the AVL tree property, we start at the insertion point and keep going to root of the tree. While moving to root, we need to consider the first node whichever is not satisfying the AVL property. From that node onwards every node on the path to root will have the issue. Also, if we fix the issue for that first node, then all other nodes on the path to root will automatically satisfy the AVL tree property. That means we always need to care for the first node whichever is not satisfying the AVL property on the patl: from insertion point to root and fix it.

Types of Violations

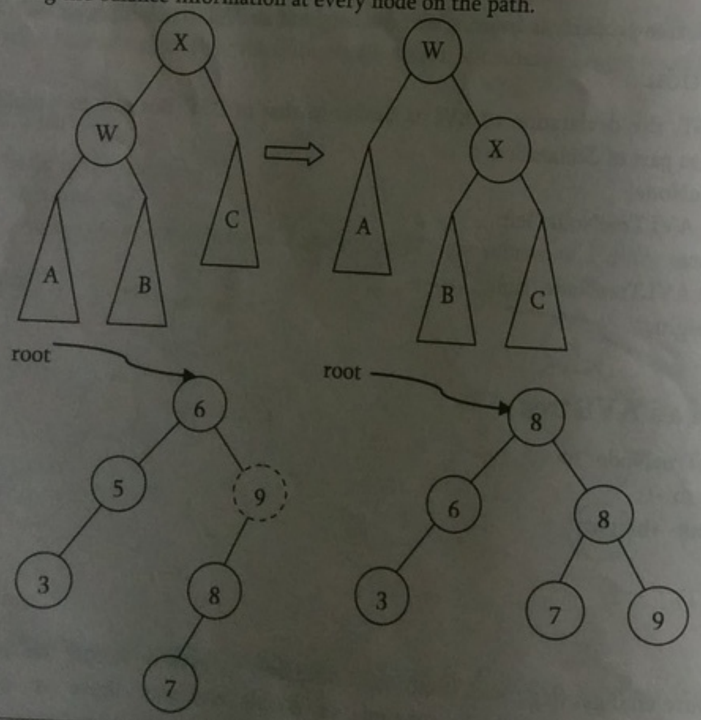
Let us assume the node that must be rebalanced is X. Since any node has at most two children, and a height imbalance requires that X's two subtree heights differ by two. We can easily observe that a violation might occur in four cases:

1. An insertion into the left subtree of the left child of X.
2. An insertion into the right subtree of the left child of X.
3. An insertion into the left subtree of the right child of X.
4. An insertion into the right subtree of the right child of X.

Cases 1 and 4 are symmetric and easily solved with single rotations. Similarly, cases 2 and 3 are also symmetric and can be solved with double rotations (needs two single rotations).

Single Rotations

Left Left Rotation (LL Rotation) [Case-1]: In the below case, at node X, the AVL tree property is not satisfying. As discussed earlier, rotation does not have to be done at the root of a tree. In general, we start at the node inserted and travel up the tree, updating the balance information at every node on the path.

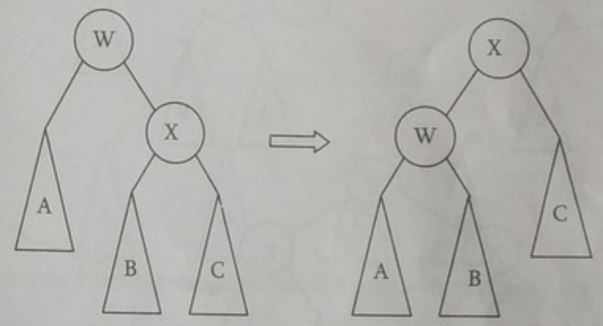


For example, in above figure, after the insertion of 7 in the original AVL tree on the left, node 9 becomes unbalanced. So, we do a single left-left rotation at 9. As a result we get the tree on the right.

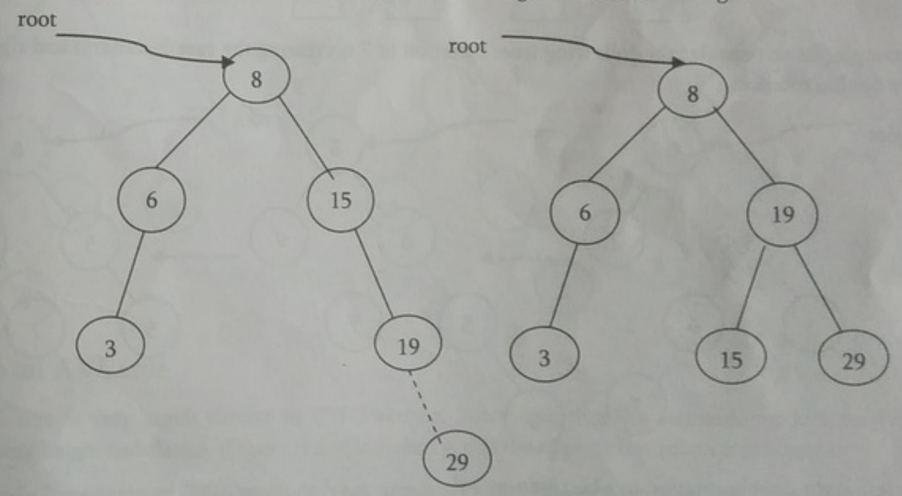
```
struct AVLTreeNode *SingleRotateLeft(struct AVLTreeNode *X ){
        struct AVLTreeNode *W = X→left;
        X→left = W→right;
        W→right = X;
        X→height = max( Height(X→left), Height(X→right) ) + 1;
        W→height = max( Height(W→left), X→height ) + 1;
        return W;  /* New root */
}
```

Time Complexity: O(1). Space Complexity: O(1).

Right Right Rotation (RR Rotation) [Case-4]: In this case, the node X is not satisfying the AVL tree property.



For example, in above figure, after the insertion of 29 in the original AVL tree on the left, node 15 becomes unbalanced. So, we do a single right-right rotation at 15. As a result we get the tree on the right.

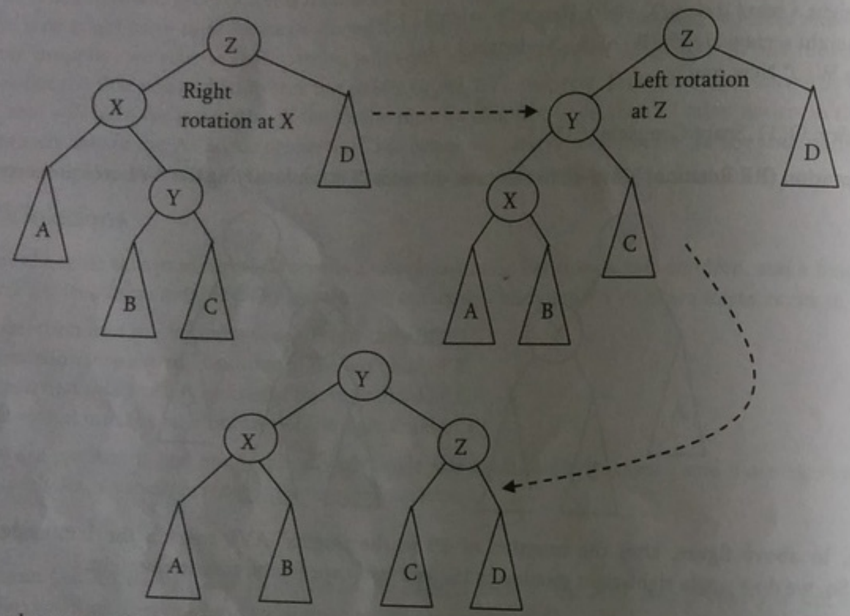



```
struct AVLTreeNode *SingleRotateRight(struct AVLTreeNode *W ) {
        struct AVLTreeNode *X  = W→right;
        W→right = X→left;
        X→left = W;
        W→height = max( Height(W→right), Height(W→left) ) + 1;
        X→height = max( Height(X→right), W→height) + 1;
        return X;
```

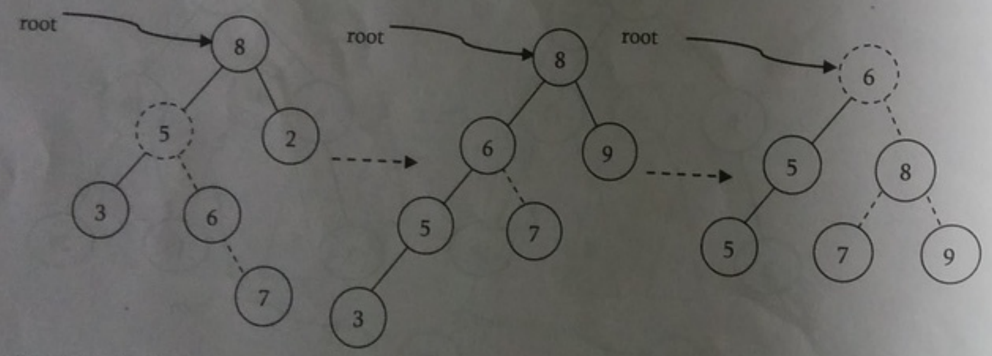
}
Time Complexity: O(1). Space Complexity: O(1).

Double Rotations

Left Right Rotation (LR Rotation) [Case-2]: For case-2 and case-3 single rotation does not fix the problem. We need to perform two rotations.



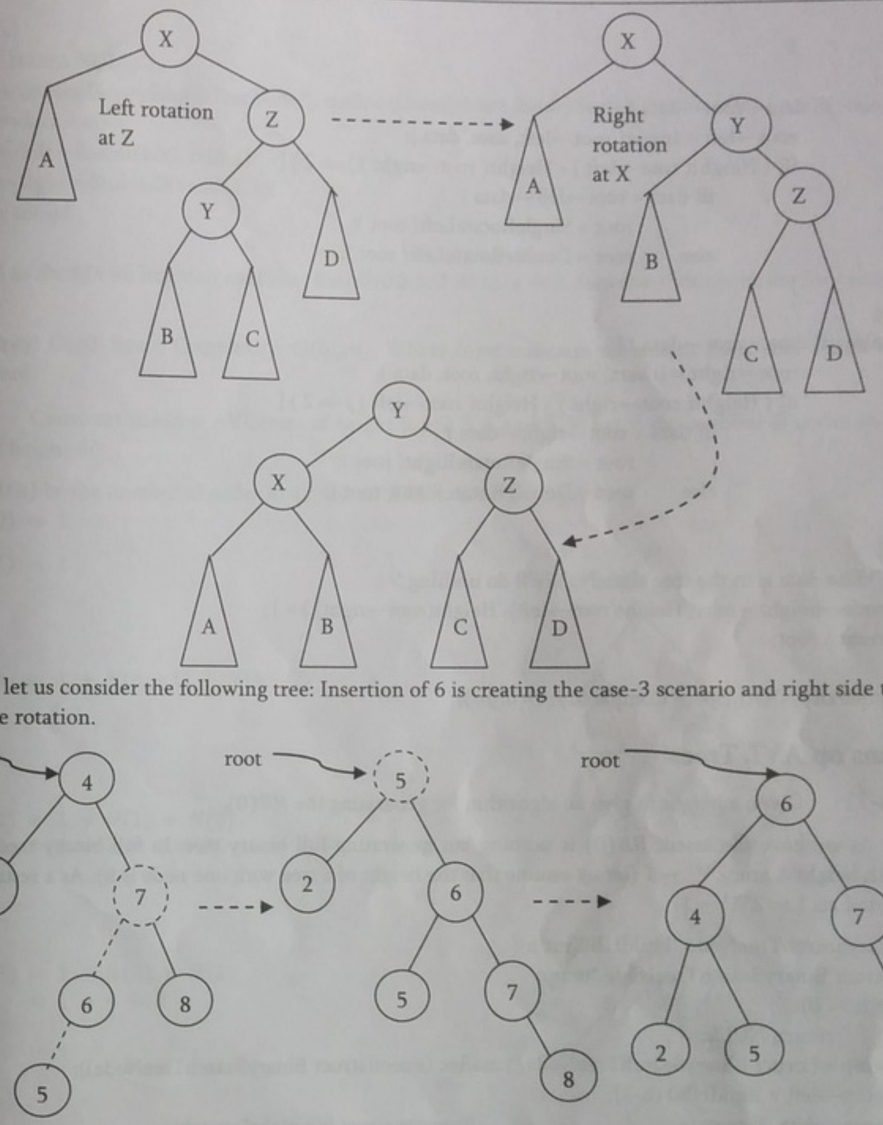
As an example, let us consider the following tree: Insertion of 7 is creating the case-2 scenario and right side tree is the one after double rotation.



Code for left-right double rotation can be given as:

```
struct AVLTreeNode *DoubleRotatewithLeft( struct AVLTreeNode *Z ){
        Z→left = SingleRotateRight( Z→left );
        return SingleRotateLeft(Z);
}
```

Right Left Rotation (RL Rotation) [Case-3]: As similar to case-2, we need to perform two rotations for fixing this scenario.



As an example, let us consider the following tree: Insertion of 6 is creating the case-3 scenario and right side tree is the one after double rotation.

Insertion into an AVL tree

Insertion in AVL tree is very much similar to BST insertion. After inserting the element, we just need to check whether there is any height imbalance. If there is any imbalance, call the appropriate rotation functions.

```
struct AVLTreeNode *Insert( struct AVLTreeNode *root, struct AVLTreeNode *parent, int data){
        if( !root) {
                root = (struct AVLTreeNode*) malloc(sizeof (struct AVLTreeNode*) );
                if(!root) { printf("Memory Error");
                        return NULL;
                }
                else {  root→data = data;
                        root→height = 0;
                        root→left = root→right = NULL;
```

```
            }
        }
        else if( data < root→data ) {
                root→left = Insert( root→left, root, data );
                if( ( Height( root→left ) - Height( root→right ) ) == 2 ) {
                        if( data < root→left→data )
                                root = SingleRotateLeft( root );
                        else    root = DoubleRotateLeft( root );
                }
        }
        else if( data > root→data ) {
                root→right = Insert( root→right, root, data );
                if( ( Height( root→right ) - Height( root→left ) ) == 2 ) {
                        if( data < root→right→data )
                                root = SingleRotateRight( root );
                        else    root = DoubleRotateRight( root );
                }
        }
        /* Else data is in the tree already. We'll do nothing */
        root→height = max( Height(root→left), Height(root→right) ) + 1;
        return root;
}
```

Time Complexity: O(n). Space Complexity: O($logn$).

Problems on AVL Trees

Problem-71 Given a height h, give an algorithm for generating the $HB(0)$.

Solution: As we have discussed, $HB(0)$ is nothing but generating full binary tree. In full binary tree the number of nodes with height h are: $2^{h+1} - 1$ (let us assume that the height of a tree with one node is 0). As a result the nodes can be numbered as: 1 to $2^{h+1} - 1$.

```
struct BinarySearchTreeNode *BuildHB0(int h){
        struct BinarySearchTreeNode *temp;
        if(h == 0)
                return NULL;
        temp = (struct BinarySearchTreeNode *) malloc (sizeof(struct BinarySearchTreeNode));
        temp→left = BuildHB0 (h-1);
        temp→data = count++;                    //assume count is a global variable
        temp→right = BuildHB0 (h-1);
        return temp;
}
```

Time Complexity: O(n). Space Complexity: O($logn$), where $logn$ indicates the maximum stack size which is equal to height of tree.

Problem-72 Is there any alternative way of solving Problem-71?

Solution: Yes, we can solve following Mergesort logic. That means, instead of working with height, we can take the range. With this approach we do not need any global counter to be maintained.

```
Struct BinarySearchTreeNode *BuildHB0(int l, int r){
        struct BinarySearchTreeNode *temp;
        int mid = l + (r-l)/2;
        if( l > r)
                return NULL;
        temp = (struct BinarySearchTreeNode *) malloc (sizeof(struct BinarySearchTreeNode));
        temp→data = mid;
        temp→left = BuildHB0(l, mid-1);
        temp→right = BuildHB0(mid+1, r);
        return temp;
}
```

The initial call to $BuildHB0$ function could be: $BuildHB0(1, 1 \ll h)$. $1 \ll h$ does the shift operation for calculating the $2^{h+1} - 1$.

Time Complexity: O(n). Space Complexity: O($logn$). Where $logn$ indicates maximum stack size which is equal to height of the tree.

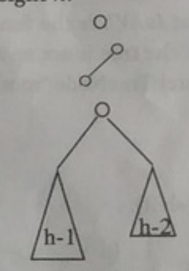
Problem-73 Construct minimal AVL trees of height 0, 1, 2, 3, 4, and 5. What is the number of nodes in a minimal AVL tree of height 6?

Solution Let $N(h)$ be the number of nodes in a minimal AVL tree with height h.

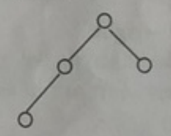
$N(0) = 1$

$N(1) = 2$

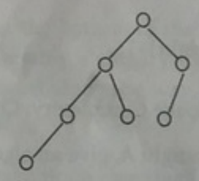
$N(h) = 1 + N(h-1) + N(h-2)$



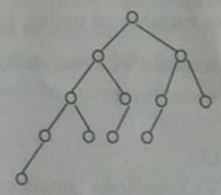
$N(2) = 1 + N(1) + N(0)$
$= 1 + 2 + 1 = 4$



$N(3) = 1 + N(2) + N(1)$
$= 1 + 4 + 2 = 7$



$N(4) = 1 + N(3) + N(2)$
$= 1 + 7 + 4 = 12$



$N(5) = 1 + N(4) + N(3)$
$= 1 + 12 + 7 = 20$

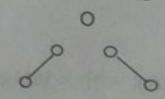
...

Problem-74 For the Problem-71 how many different shapes of a minimal AVL tree of height h can have?

Solution: Let $NS(h)$ be the number of different shapes of a minimal AVL tree of height h.

$NS(0) = 1$

$NS(1) = 2$

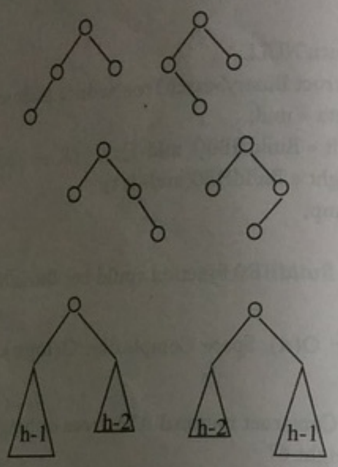


$NS(2) = 2 * NS(1) * NS(0)$
$= 2 * 2 * 1 = 4$

$NS(3) = 2 * NS(2) * NS(1)$
$= 2 * 4 * 1 = 8$

...

$NS(h) = 2 * NS(h-1) * NS(h-2)$



Problem-75 Given a binary search tree check whether the tree is an AVL tree or not?

Solution: Let us assume that *IsAVL* is the function which checks whether the given binary search tree is an AVL tree or not. *IsAVL* returns -1 if the tree is not an AVL tree. During the checks each node sends height of it to their parent.

```
int IsAVL(struct BinarySearchTreeNode *root){
    int left, right;
    if(!root)
        return 0;
    left = IsAVL(root→left);
    if(left == -1)
        return left;
    right = IsAVL(root→right);
    if(right == -1)
        return right;
    if(abs(left-right)>1)
        return -1;
    return Max(left, right)+1;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$.

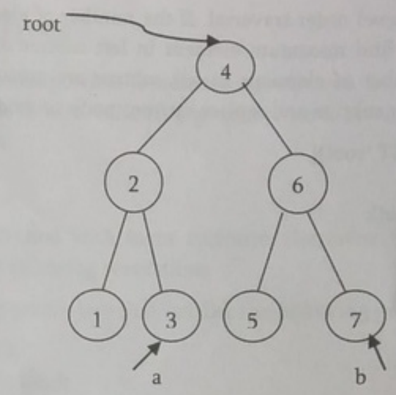
Problem-76 Given a height h, give an algorithm to generate an AVL tree with min number of nodes.

Solution: To get minimum number of nodes, fill one level with $h-1$ and other with $h-2$.

```
struct AVLTreeNode *GenerateAVLTree(int h){
    struct AVLTreeNode *temp;
    if(h == 0)
        return NULL;
    temp = (struct AVLTreeNode *)malloc (sizeof(struct AVLTreeNode));
    temp→left = GenerateAVLTree(h-1);
    temp→data = count++; //assume count is a global variable
    temp→right = GenerateAVLTree(h-2);
    temp→height = temp→left→height+1; // or temp→height = h;
    return temp;
}
```

Problem-77 Given an AVL tree with n integer items and two integers a and b, where a and b can be any integers with $a <= b$. Implement an algorithm to count the number of nodes in the range $[a, b]$.

Solution:



The idea is to make use of the recursive property of binary search trees. There are three cases to consider, whether the current node is in the range $[a, b]$, on the left side of the range $[a, b]$ or on the right side of the range $[a, b]$. Only subtrees that possibly contain the nodes will be processed under each of the three cases.

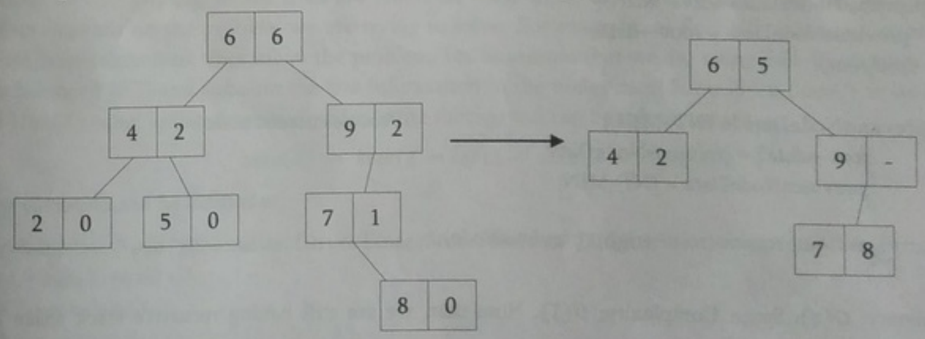
```
int RangeCount(struct AVLNode *root, int a, int b) {
    if(root == NULL) return 0;
    else if(root→data > b)
        return RangeCount(curr→left, a, b);
    else if(root→data < a)
        return RangeCount(root→right, a, b);
    else if(root→data >= a && root→data <= b)
        return RangeCount(root→left, a, b) + RangeCount(root→right, a, b) + 1;
}
```

The complexity is similar to $in-order$ traversal of the tree but skipping left or right sub-trees when they do not contain any answers. So in the worst case, if the range covers all the nodes in the tree, we need to traverse all the n nodes to get the answer. The worst time complexity is therefore $O(n)$.

If the range is small, which only covers few elements in a small subtree at the bottom of the tree, the time complexity will be $O(h) = O(logn)$, where h is the height of the tree. This is because only a single path is traversed to reach the small subtree at the bottom and many higher level subtrees have been pruned along the way.

Note: Refer similar problem in BST.

Problem-78 Given an BST (applicable to AVL trees as well) where each node contains two data elements (its data and also number of nodes in its subtrees) as shown below. Convert the tree to another BST by replacing the second data (number of nodes in its subtrees) with previous node data in inorder traversal. Note that, each node is merged with $inorder$ previous node data. Also make sure that coversion happens in-place.



Solution: One simplest way is to use level order traversal. If the number of elements in left subtree are greater than number of elements in right subtree, find maximum element in left subtree and replace current node second data element with it. Similarly, if the number of elements in left subtree are greater than number of elements in right subtree, find minimum element in right subtree and replace current node *second* data element with it.

```
struct BST *TreeCompression (struct BST *root){
        struct BST *temp, *temp2;
        struct Queue *Q = CreateQueue();
        if(!root)
                return;
        EnQueue(Q, root);
        while(!IsEmptyQueue(Q)) {
                temp = DeQueue(Q);

                if(temp→left && temp→right && temp→left→data2 > temp→right→data2)
                    temp2 = FindMax(temp);
                else temp2 = FindMin(temp);

                temp→data2 = temp2→data2;          //Process current node
                //Remember to delete this node.
                DeleteNodeInBST(temp2);
                if(temp→left)
                        EnQueue(Q, temp→left);
                if(temp→right)
                        EnQueue(Q, temp→right);
    }
    DeleteQueue(Q);
}
```

Time Complexity: $O(nlogn)$ on average since BST takes O($logn$) on average to find maximum or minimum element. Space Complexity: $O(n)$. Since, in the worst case, all the nodes on the entire last level could be in the queue simultaneously.

Problem-79 Can we reduce time complexity for the previous problem?

Solution: Let us try using an appaoch similar to Problem-59. The idea behind this solution is that, inorder traversal of BST produces sorted lists. While traversing the BST in inorder, keep track elements visited and merge them.

```
struct BinarySearchTreeNode * TreeCompression(struct BinarySearchTreeNode *root,  int *previousNodeData){
        if(!root)
            return NULL;
        TreeCompression(root→left, previousNode);
        if(*previousNodeData == INT_MIN){
            *previousNodeData = root→data;
            free(root);
        }
        if(*previousNodeData != INT_MIN){          //Process current node
                root→data2 = previousNodeData;
                *previousNodeData = INT_MIN;
        }
        return TreeCompression(root→right, previousNode);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(1)$. Note that, we are still having recursive stack space for inorder traversal.

6.14 Other Variations in Trees

In this section, let us enumerate the other possible representations of trees. In the earlier sections, we have seen AVL trees which is a binary search tree (BST) with balancing property. Now, let us see few more balanced binary search trees: Red-Black Trees and Splay Trees.

Red-Black Trees

In red-black trees each node is associated with extra attribute: the color, which is either red or black. To get logarithmic complexity we impose the following restrictions.

Definition: A red-black tree is a binary search tree that satisfies the following properties:

- Root Property: the root is black
- External Property: every leaf is black
- Internal Property: the children of a red node are black
- Depth Property: all the leaves have the same black

As similar to AVL trees, if the Red-black tree becomes imbalanced then we perform rotations to reinforce the balancing property. With Red-black trees, we can perform the following operations in O($logn$) in worst case, where n is the number of nodes in the trees.

- Insertion, Deletion
- Finding predecessor, successor
- Finding minimum, maximum

Splay Trees

Splay-trees are BSTs with self-adjusting property. Another interesting property of splay-trees is: starting with empty tree, any sequence of K operations with maximum of n nodes takes O($Klogn$) time complexity in worst case.

Splay trees are easier to program and also ensures faster access to recently accessed items. As similar to *AVL* and Red-Black trees, at any point if the splay tree becomes imbalanced then we perform rotations to reinforce the balancing property.

Splay-trees cannot guarantee the O($logn$) complexity in worst case. But it gives amortized O($logn$) complexity. Even though individual operations can be expensive, any sequence of operations gets the complexity of logarithmic behavior. One operation may take more time (a single operation may take O(n) time) but the subsequent operations may not take worst case complexity and on the average *per operation* complexity is O($logn$).

Augmented Trees

In earlier sections, we have seen the problems like finding K^{th} –smallest element in the tree and many other similar problems. For all those problems the worst complexity is O(n), where n is the number of nodes in the tree. To perform such operations in O($logn$) augmented trees are useful. In these trees, extra information is added to each node and that extra data depends on the problem we are trying to solve. For example, to find K^{th} –smallest in binary search tree, let us see how augmented trees solves the problem. Let us assume that we are using Red-Black trees as balanced BST (or any balanced BST) and augment the size information in the nodes data. For a given node X in Red-Black tree with a field $size(X)$ equal to the number of nodes in the subtree and can be calculated as:

$$size(X) = size(X \rightarrow left) + size(X \rightarrow right)) + 1$$

Kth-smallest operation can be defined as:

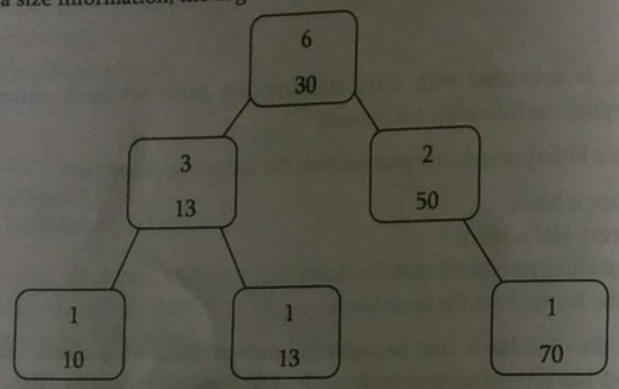
```
struct BinarySearcTreeNode *KthSmallest (struct BinarySearcTreeNode *X, int K) {
        int r = size(X→left) + 1;
        if(K == r) return X;
        if(K < r) return KthSmallest (X→left, K);
```

if(K > r) return KthSmallest (X→right, K-r);

}

Time Complexity: O($logn$). Space Complexity: O($logn$).

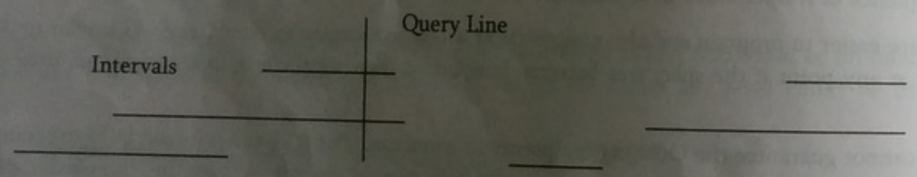
Example: With the extra size information, the augmented tree will look like:



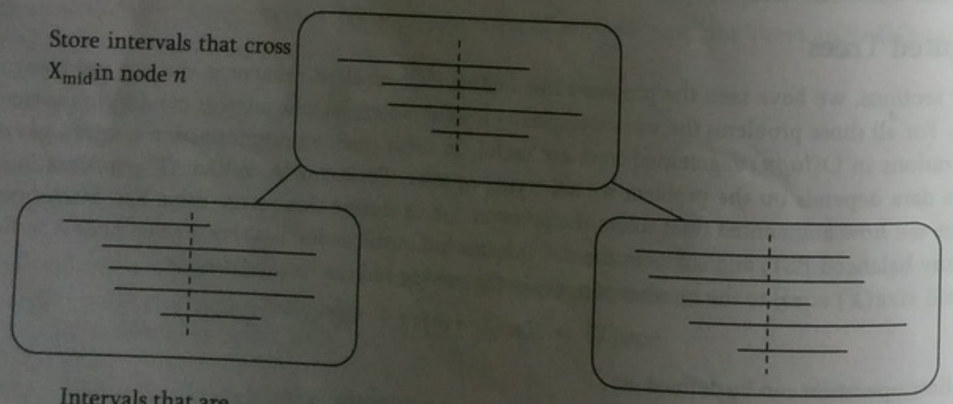
Interval Trees

Interval trees are also binary search trees and stores interval information in the node structure. That means, we maintain a set of n intervals $[i_1, i_2]$ such that one of the intervals containing a query point Q (if any) can be found efficiently. Interval trees are used for performing range queries efficiently.

Example: Given a set of intervals: S = {[2-5], [6-7], [6-10], [8-9], [12-15], [15-23], [25-30]}. A query with $Q = 9$ returns $[6, 10]$ or $[8, 9]$ (assume these are the intervals which contains 9 among all the intervals). A query with $Q = 23$ returns $[15, 23]$.



Construction of Interval Trees: Let us assume that we are given a set S of n intervals (also called segments). These n intervals will have $2n$ endpoints. Now, let us see how to construct the interval tree.



Algorithm:

Recursively build tree on interval set S as follows:

- Sort the $2n$ endpoints
- Let X_{mid} be the median point

Time Complexity for building interval trees: O($nlogn$). Since we are choosing the median, Interval Trees will be approximately balanced. This ensures that, we split the set of end points up in half each time. The depth of the tree is O($logn$). To simplify the search process, generally X_{mid} is stored with each node.

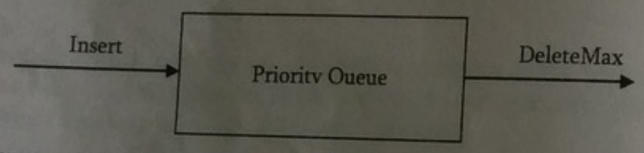
Chapter-7

PRIORITY QUEUE AND HEAPS

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^{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($logn$) on average if insertions are random (refer *Trees* chapter).

Balanced Binary Search Trees Implementation

Both insertions and deletion take O($logn$) 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($logn$) complexity for search, insertions and deletions and O(1) for finding the maximum or minimum element.

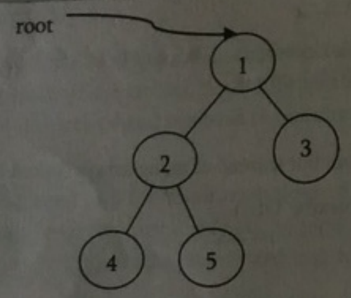
Comparing Implementations

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

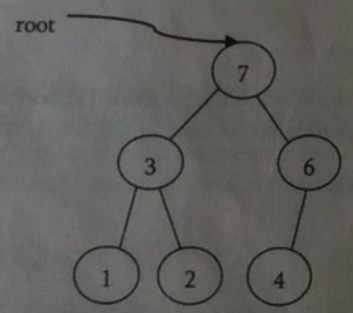
7.5 Heaps and Binary Heap

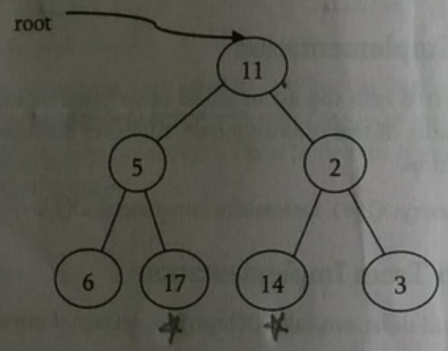
What is a Heap?

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).

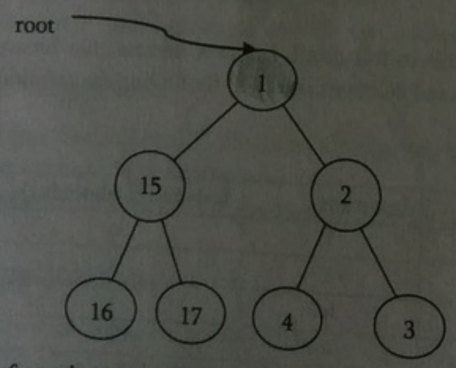




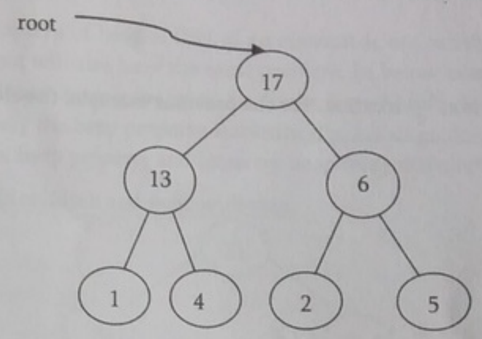
Types of Heaps?

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

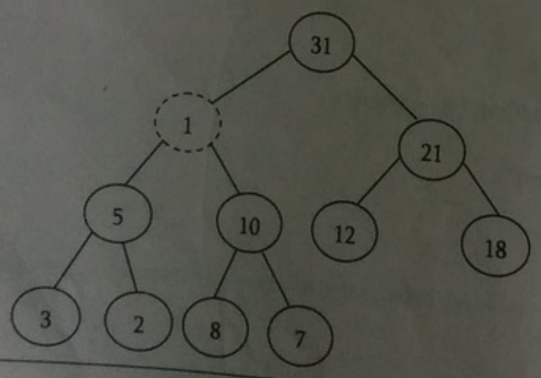
Since the maximum element in max heap is always at root, it will be stored at h→array[0].

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

Time Complexity: O(1).

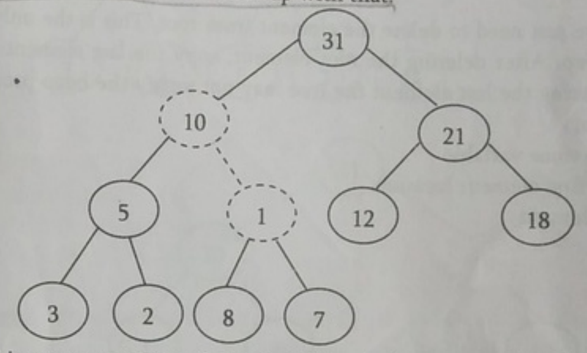
Heapifying an Element

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.

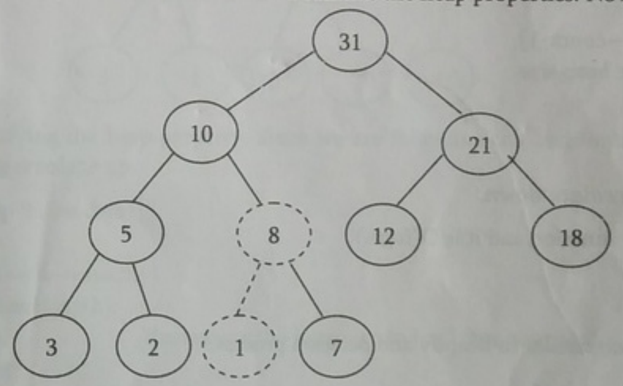


Observation: One important property of heap is that, if an element is not satisfying the heap property then all the elements from that element to root will also have the same problem. In below example, element 1 is not satisfying the heap property and its parent 31 is also having the issue. Similarly,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

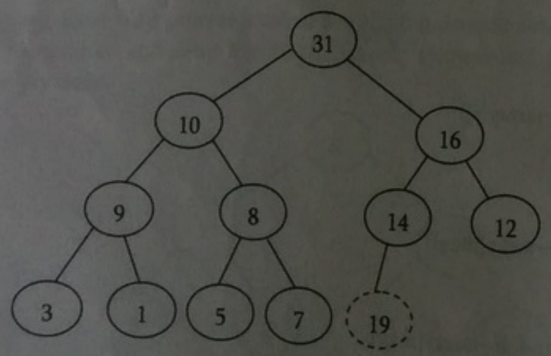
Time Complexity: same as Heapify function and it is O(*logn*).

Inserting an Element

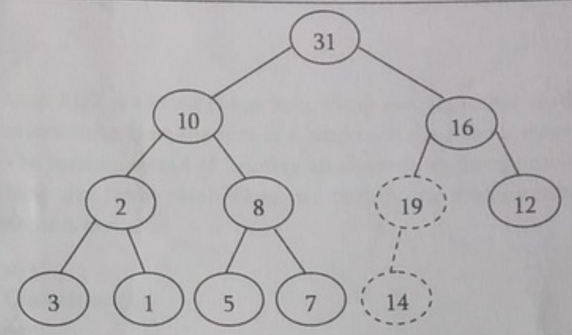
Insertion of an element is very much similar to heapify and deletion process.

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

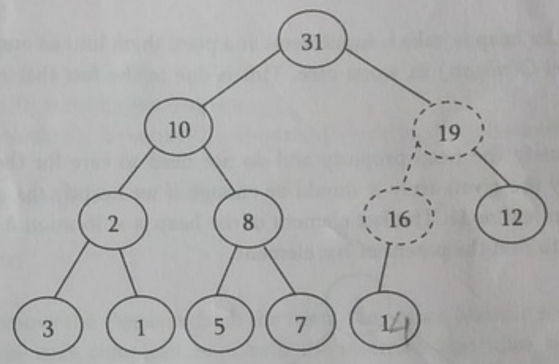
Before going through code, let us take an example. We have inserted the element 19 at the end of the heap and it's not satisfying the heap property.



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 and16:



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(*logn*). The explanation is same as that of Heapify function.

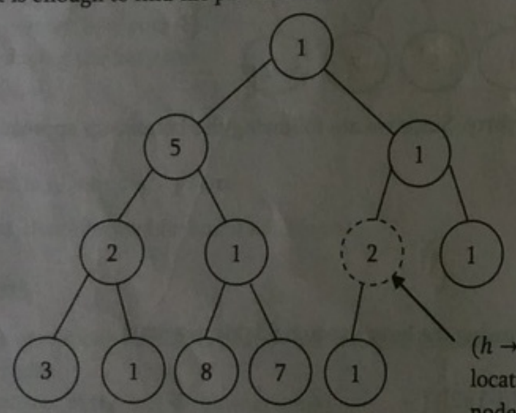
Destroying Heap

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

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 $h \rightarrow count - 1$, and to find the first non-leaf node it is enough to find the parent of last element.



$(h \rightarrow count - 1)/2$ 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 *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[0] = 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 $O(logn)$ (where n is the number of items in the heap), the time complexity of the heap sort algorithm is $O(nlogn)$.

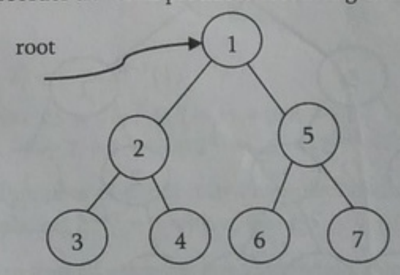
7.7 Problems on Priority Queues [Heaps]

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).

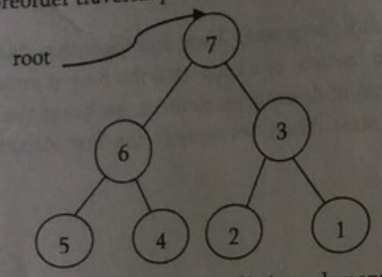
Problem-2 Is there a min-heap with seven distinct elements so that, the preorder traversal of it gives the elements in sorted order?

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.

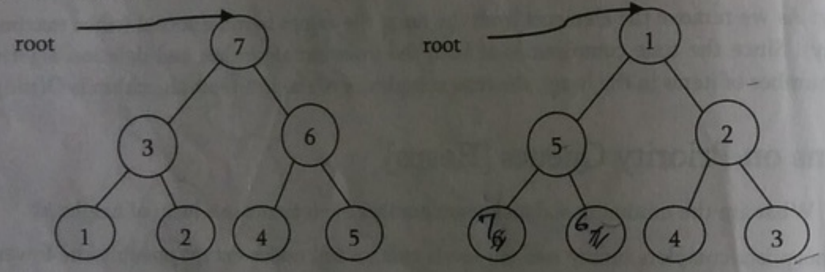


Problem-4 Is there a min-heap/max-heap with seven distinct elements so that, the inorder traversal of it gives the elements in sorted order?

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).

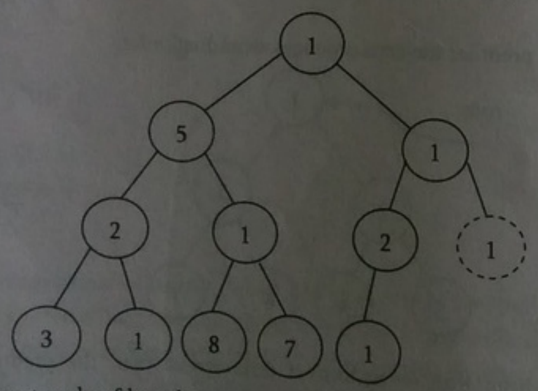


Problem-6 Show that the height of a heap with n elements is $logn$?

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, $h \leq logn \leq h + 1$. Since h is integer, $h = logn$.

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:

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

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: $O(\frac{n}{2}) \approx O(n)$.

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
$= O(n) + O(logn) \approx O(n)$. //Time for searching is dominated.

Problem-9 Give an algorithm for deleting the i^{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 = $O(logn)$.

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:

$$S = \sum_{i=0}^{h} 2^i(h - i)$$
$$S = h + 2(h - 1) + 4(h - 2) + \cdots + 2^{h-1}(1)$$

Multiplying with 2 on both sides gives: $2S = 2h + 4(h - 1) + 8(h - 2) + \cdots + 2^h(1)$
Now, subtract S from $2S$: $2S - S = -h + 2 + 4 + \cdots + 2^h \Rightarrow S = (2^{h+1} - 1) - (h - 1)$

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: $S = n - (h - 1) = O(n - logn) = O(n - h)$.

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 $O(logm \times logn)$ 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: $O(klogn)$. Since we are performing deletion operation k times and each deletion takes $O(logn)$.

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^{th} smallest element and we need k loops to get the k^{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 $O(klogk)$.

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.

$T(n)$ = DeleteMin from heap k times = $\Theta(klogn)$.

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 implemention 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^{nd} 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: $O(n) = n/1000 \times$(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: $= \frac{2n}{k} + \frac{3n}{k} + \frac{4n}{k} + \cdots \cdot \frac{kn}{k} = \sum_{i=2}^{n} \frac{in}{k} = \frac{n}{k} \sum_{i=2}^{n} i \approx \frac{n(k^2)}{k} \approx O(nk)$. 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 $logk$ 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 $O(nlogn)$. 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 = $O(n\ logk\ )$. At a time we have k elements max heap and for all n elements we have to read just the heap in $logk$ time so total time = $O(nlogk)$. 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 $O(2logn) \approx O(logn)$.

Total Time complexity: $O(nlogn)$.

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($logn$) time. That means, design a data structure which supports the following operations:

Operation	Complexity
Init	O(n)
Insert	O($logn$)
FindMin	O(1)
FindMax	O(1)
DeleteMin	O($logn$)
DeleteMax	O($logn$)

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}.

Init	Build H_{min} in O(n) and H_{max} in O(n)
Insert(x)	Insert x to H_{min} in O($logn$). Insert x to H_{max} in O($logn$). Update the pointers in O(1)
FindMin()	Return root(H_{min}) in O(1)
FindMax	Return root(H_{max}) in O(1)
DeleteMin	Delete the minimum from H_{min} in O($logn$). Delete the same element from H_{max} by using the mutual pointer in O($logn$)
DeleteMax	Delete the maximum from H_{max} in O($logn$). Delete the same element from H_{min} by using the mutual pointer in O($logn$)

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($nlogw$). Insert operation takes O($logw$) 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)];
}
```

Chapter-8

DISJOINT SETS ADT

8.1 Introduction

In this chapter, we will represent an important mathematics concept *sets*. That means how to represent a group of elements which do not need any order. The disjoint sets ADT is the one used for this purpose. It is used for solving the equivalence problem. It is very simple to implement and a simple array can be used for the implementation and each function takes only a few lines of code. Disjoint sets ADT acts as an auxiliary data structure for many other algorithms (for example, Kruskal's algorithm in graph theory). Before starting our discussion on disjoint sets ADT, let us see some basic properties of sets.

8.2 Equivalence Relations and Equivalence Classes

For the below discussion let us assume that S is a set containing the elements and a relation R is defined on it. That means for every pair of elements in $a, b \in S$, $a\ R\ b$ is either true or false. If $a\ R\ b$ is true, then we say a is related to b, otherwise a is not related to b. A relation R is called an *equivalence relation* if it satisfies the following properties:

- *Reflexive*: For every element $a \in S$, $a\ R\ a$ is true.
- *Symmetric*: For any two elements $a, b \in S$, if $a\ R\ b$ is true then $b\ R\ a$ is true.
- *Transitive*: For any three elements a, b, c ∈ S, if a R b and $b\ R\ c$ are true then $a\ R\ c$ is true.

As an example, relations $\leq$ (less than or equal to) and $\geq$ (greater than or equal to) on a set of integers are not equivalence relations. They are reflexive (since $a \leq a$) and transitive ($a \leq b$ and $b \leq c$ implies $a \leq c$) but it is not symmetric ($a \leq b$ does not imply $b \leq a$).

Similarly, *rail connectivity* is an equivalence relation. This relation is reflexive because any location is connected to itself. If there is a connectivity from city a to city b, then city b also has connectivity to city a, so the relation is symmetric. Finally, if city a is connected to city b and city b is connected to city c, then city a is also connected to city c.

The *equivalence class* of an element $a \in S$ is a subset of S that contains all the elements that are related to a. Equivalence classes creates a *partition* of S. Every member of S appears in exactly one equivalence class. To decide if $a\ R\ b$, we just need to check whether a and b are in the same equivalence class (group) or not.

In the above example, two cities will be in same equivalence class if they have rail connectivity. If they are not having connectivity then they will be part of different equivalence classes.

Since the intersection of any two equivalence classes is empty (ϕ), the equivalence classes are sometimes called *disjoint sets*. In the subsequent sections, we will try to see the operations that can be performed on equivalence classes. The possible operations are:

- Creating an equivalence class (making a set)
- Finding the equivalence class name (Find)
- Combining the equivalence classes (Union)

8.3 Disjoint Sets ADT

To manipulate the set elements we need basic operations defined on sets. In this chapter, we concentrate on following set operations:

- MAKESET(X): Creates a new set containing a single element X.
- UNION(X, Y): Creates a new set containing the elements X and Y with their union and deletes the sets containing the elements X and Y.
- FIND(X): Returns the name of the set containing the element X.

8.4 Applications

Disjoint sets ADT has many applications and few of them are:

- To represent network connectivity
- Image processing
- For finding least common ancestor
- For defining equivalence of finite state automata
- Kruskal's minimum spanning tree algorithm (graph theory)
- In game algorithms

8.5 Tradeoffs in Implementing Disjoint Sets ADT

Let us see the possibilities for implementing disjoint set operations. Initially, assume the input elements are collection of n sets, each with one element. That means, initial representation assumes all relations (except reflexive relations) are false. Each set has a different element, so that $S_i \cap S_j = \phi$. This makes the sets *disjoint*.

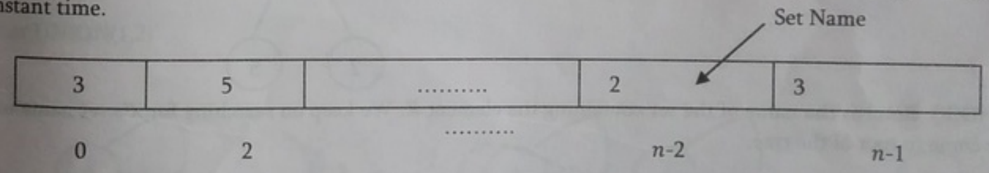
To add the relation $a\ R\ b$ (UNION), we first need to check whether a and b are already related or not. This can be verified by performing FINDs on both a and b and checking whether they are in the same equivalence class (set) or not. If they are not, then we apply UNION. This operation merges the two equivalence classes containing a and b into a new equivalence class by creating a new set $S_k = S_i \cup S_j$ and deletes S_i and S_j. Basically there are two ways to implement the above FIND/UNION operations:

- Fast FIND implementation (also called Quick FIND)
- Fast UNION operation implementation (also called Quick UNION)

Fast FIND Implementation (Quick FIND)

In this method, we use an array. As an example, in the below representation the array contains the set name for each element. For simplicity, let us assume that all the elements numbered sequentially from 0 to $n - 1$.

For the below example, element 0 has the set name 3, for element 1 the set name is 5 and so on. With this representation FIND takes only O(1) since for any element we can find the set name by accessing its array location in constant time.



In this representation, to perform UNION(a, b) [assuming that a is in set i and b is in set j] we need to scan the complete array and change all i's to j. This takes O(n).
A sequence of $n - 1$ unions take O(n^2) time in the worst case. If there are O(n^2) FIND operations, this performance is fine, as the average time complexity comes to O(1) for each UNION or FIND operation. If there are fewer FINDs, this complexity is not acceptable.

Fast UNION Implementation (Quick UNION)

In this and subsequent sections, we will discuss the faster *UNION* implementations and its variants. There are different ways of implementing this approach and below are few of them.

- Fast UNION implementation (Slow FIND)

- Fast UNION implementations (Quick FIND)
- Fast UNION implementations with path compression

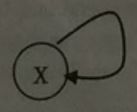
Fast UNION implementation (Slow FIND)

As we have discussed, FIND operation returns same answer (set name) if and only if they are in the same set. In representing disjoint sets, our main objective is to give different set name for each group. In general we do not care for the name of the set. One possibility for implementing the set is *tree* as each element has only one *root* and we can use it as the set name.

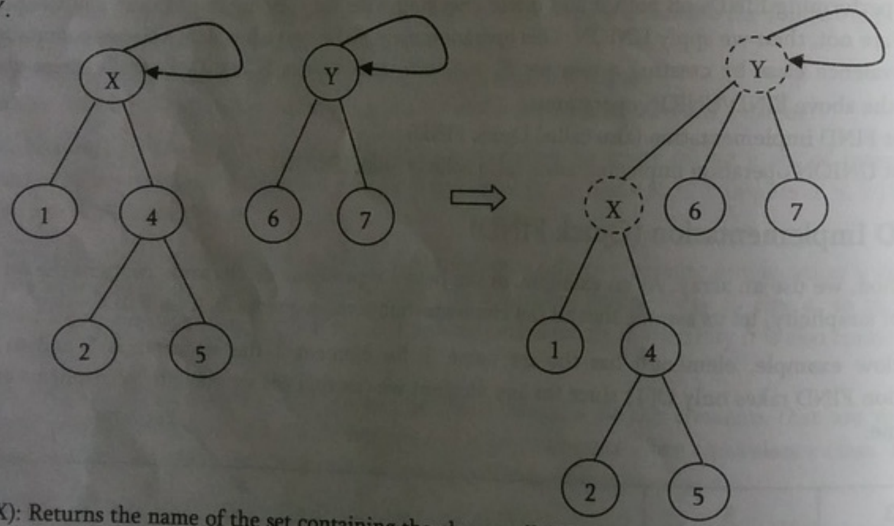
How to represent them?

One possibility is using an array: for each element keep the *root* as its set name. But with this representation, we will have the same problem as that of FIND array implementation. To solve this problem, instead of storing the *root* we can keep parent of element. Therefore, using an array which stores the parent of each element solves our problem. To differentiate the root node, let us assume its parent is same as that of element in the array. Based on this representation, MAKESET, FIND, UNION operations can be defined as:

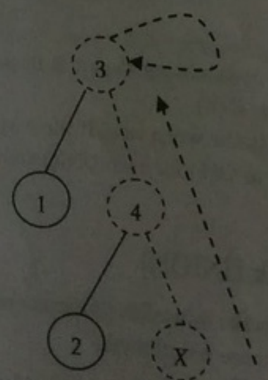
- MAKESET(X): Creates a new set containing a single element X and in the array update the parent of X as X. That means root (set name) of X is X.



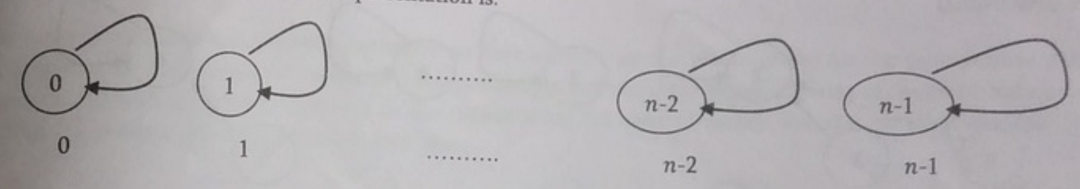
- UNION(X, Y): Replaces the two sets containing X and Y by their union and in the array update the parent of X as Y.



- FIND(X): Returns the name of the set containing the element X. We keep on searching for X's set name until we come to root of the tree.



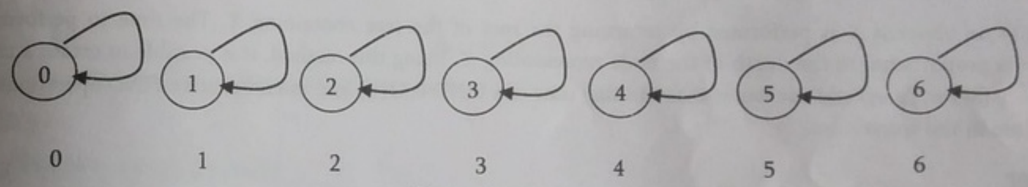
For the elements 0 to $n-1$ the initial representation is:



Parent Array

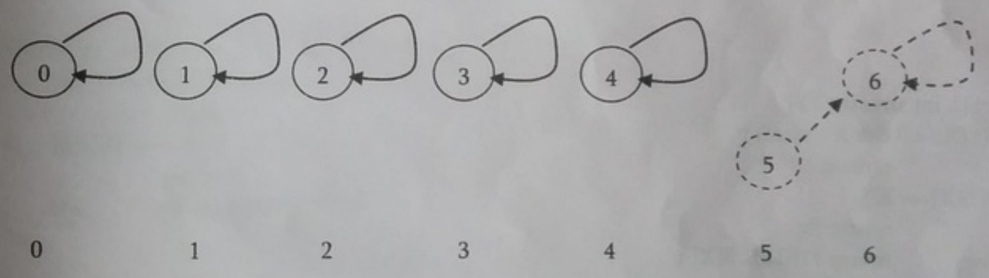
To perform a UNION on two sets, we merge the two trees by making the root of one tree point to the root of the other.

Initial Configuration for the elements 0 to 6



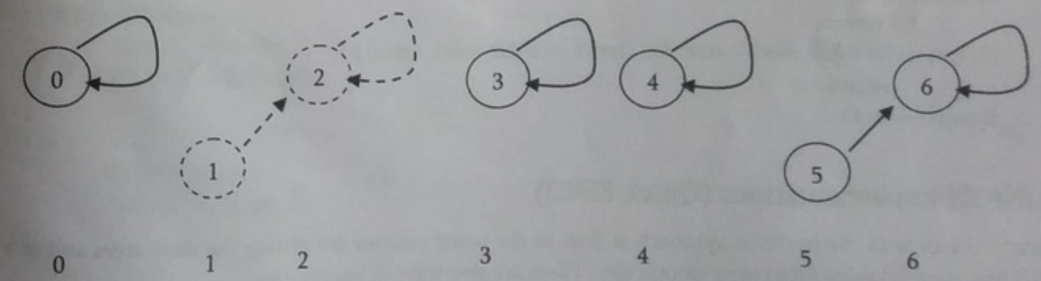
Parent Array

After UNION(5,6)



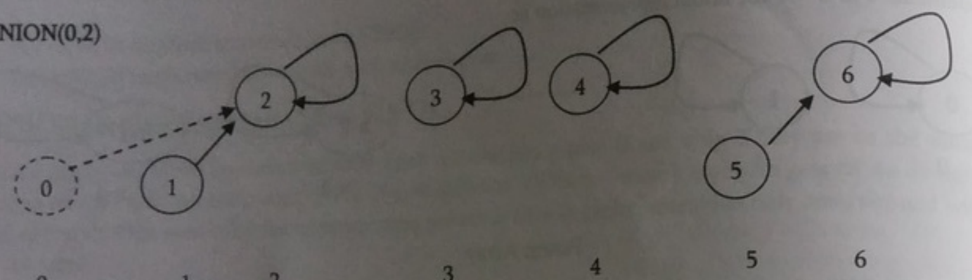
Parent Array

After UNION(1,2)



Parent Array

After UNION(0,2)



0 1 2 3 4 5 6

Parent Array

One important thing to observe here is, UNION operation is changing the roots parent only but not for all the elements in the sets. Due to this, the time complexity of UNION operation is O(1).

A FIND(X) on element X is performed by returning the root of the tree containing X. The time to perform this operation is proportional to the depth of the node representing X. Using this method, it is possible to create a tree of depth $n - 1$ (Skew Trees) and the worst-case running time of a FIND is O(n) and m consecutive FIND operations take O(mn) time in the worst case.

MAKESET

```
void MAKESET( int S[], int size) {
        for(int  i = size-1; i >=0; i-- )
                S[i] = i;
}
```

FIND

```
int FIND(int S[], int size, int X) {
        if(! (X >= 0 && X < size)) )
                return;
        if( S[X] == X )
                return X;
        else    return FIND(S, S[X]);
}
```

UNION

```
void UNION( int S[], int size, int root1, int root2 ) {
        if(FIND(root1) == FIND(root2))
                return;
        if(! ((root1 >= 0 && root1 < size) && (root1 >= 0 && root1 < size) ) )
                return;
        S[root1] = root2;
}
```

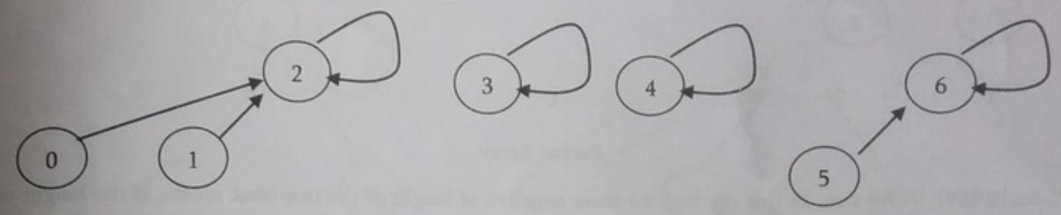
Fast UNION implementations (Quick FIND)

The main problem with the previous approach is that, in the worst case we are getting the skew trees and as a result the FIND operation is taking O(n) time complexity. There are two ways to improve it:

- UNION by Size (also called UNION by Weight): Make the smaller tree as a subtree of the larger tree
- UNION by Height (also called UNION by Rank): Make the tree with smaller height as a subtree of the tree with larger height

UNION by Size

In the earlier representation, for each element i we have stored i (in the parent array) for the root element and for other elements we have stored the parent of i. But in this approach we store negative of size of the tree (that means, if the size of the tree is 3 then store -3 in the parent array for root element). For the previous example (after UNION(0,2)), the new representation will look like:



2 2 -3 -1 -1 6 -2

Parent Array

Assume that the size of one element set is 1 and store -1. Other than this there is no change.

MAKESET

```
void MAKESET( int S[], int size) {
        for(int  i = size-1; i >= 0; i-- )
                S[i] = -1;
}
```

FIND

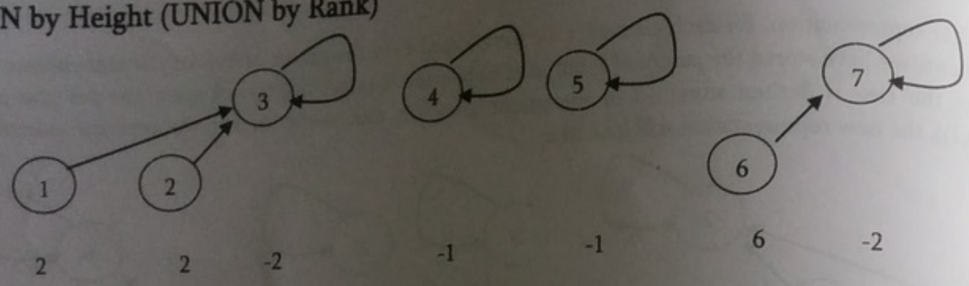
```
int FIND(int S[], int size, int X) {
        if(! (X >= 0 && X < size)) )
                return;
        if( S[X] == -1 )
                return X;
        else    return FIND(S, S[X]);
}
```

UNION by Size

```
void UNIONBySize(int S[], int size, int root1, int root2) {
        if(FIND(root1) == FIND(root2))
                return;
        if( S[root2] < S[root1] )    {
                S[root1] = root2;
                S[root2] += S[root1];
        }
        else {
                S[root2] = root1;
                S[root1] += S[root2];
        }
}
```

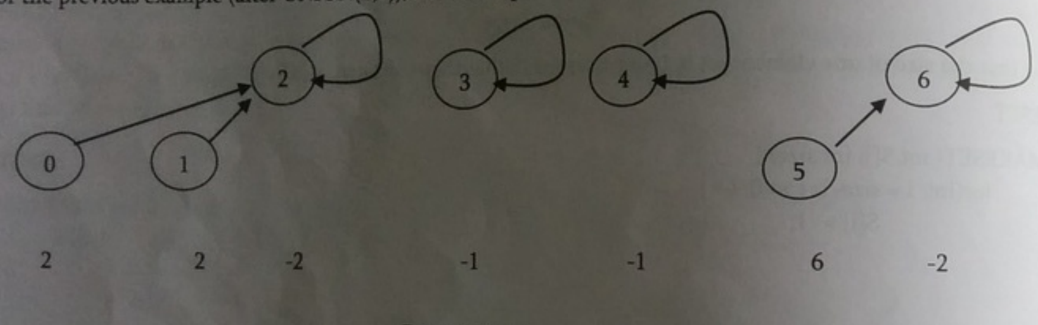
Note: There is no change in FIND operation implementation.

UNION by Height (UNION by Rank)



Parent Array

As similar to UNION by size, in this method we store negative of height of the tree (that means, if the height of the tree is 3 then we store −3 in the parent array for root element). We assume the height of a tree with one element set is 1. For the previous example (after UNION(0,2)), the new representation will look like:



Parent Array

UNION by Height

```
void UNIONByHeight(int S[], int size, int root1, int root2) {
        if(FIND(root1) == FIND(root2))
                return;
        if( S[root2] < S[root1] )
                S[root1] = root2;
        else {  if( S[root2] == S[root1] )
                        S[root1]--;
                S[root2] = root1;
        }
}
```

Note: For FIND operation there is no change in the implementation.

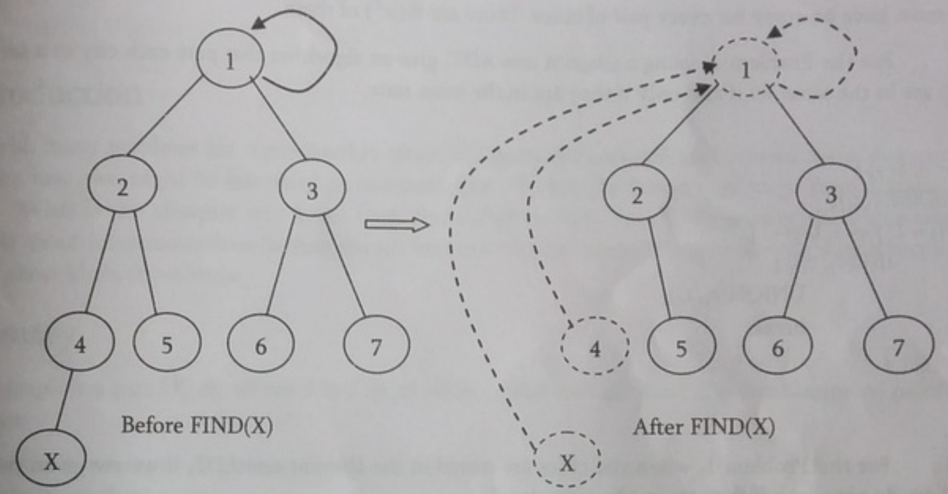
Comparing UNION by Size and UNION by Height

With UNION by size, the depth of any node is never more than $logn$. This is because a node is initially at depth 0. When its depth increases as a result of a UNION, it is placed in a tree that is at least twice as large as before. That means its depth can be increased at most $logn$ times. This says that the running time for a FIND operation is $O(logn)$, and a sequence of m operations takes $O(m\ logn)$.

Similarly with UNION by height, if we take the UNION of two trees of the same height, the height of the UNION is one larger than the common height, and otherwise equal to the max of the two heights. This will keep the height of tree of n nodes from growing past $O(logn)$. A sequence of m UNIONs and FINDs can then still cost $O(m\ logn)$.

Path Compression

FIND operation traverses list of nodes on the way to the root. We can make later FIND operations efficient by making each of these vertices point directly to the root. This process is called *path compression*. For example, in the FIND(X) operation, we travels from X till the root of the tree. The effect of path compression is that every node on the path from X to the root has its parent changed to the root.



With path compression the only change to the FIND function is that $S[X]$ is made equal to the value returned by FIND. That means, after the root of the set is found recursively, X is made to point directly to it. This happen recursively to every node on the path to the root.

FIND with path compression

```
int FIND(int S[], int size, int X) {
        if(! (X >= 0 && X < size)) )
                return;
        if( S[X] <= 0 )
                return X;
        else    return( S[X] = FIND( S, S[X] ) );
}
```

Note: Path compression is compatible with UNION by size but not with UNION by height as there is no efficient way to change the height of the tree.

8.6 Summary

Performing m union-find operations on a set of n objects.

Algorithm	Worst-case time
Quick-find	mn
Quick-union	mn
Quick-Union by Size/Height	$n + m\ logn$
Path compression	$n + m\ logn$
Quick-Union by Size/Height + Path Compression	$(m + n)\ logn$