**Heap And Priority Queue**

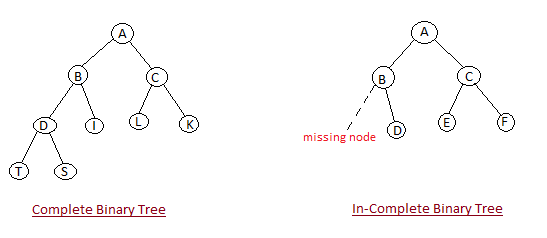
**Heap**

A heap is a specific tree based data structure in which all the nodes of tree are in a specific order. Let’s say if X is a parent node of Y, then the value of X follows some specific order with respect to value of Y and the same order will be followed across the tree.

The maximum number of children of a node in the heap depends on the type of heap. However in the more commonly used heap type, there are at most 2 children of a note and it's known as a Binary heap. We will study only binary heap.

Heap is a special tree-based data structure, that satisfies the following special heap properties :

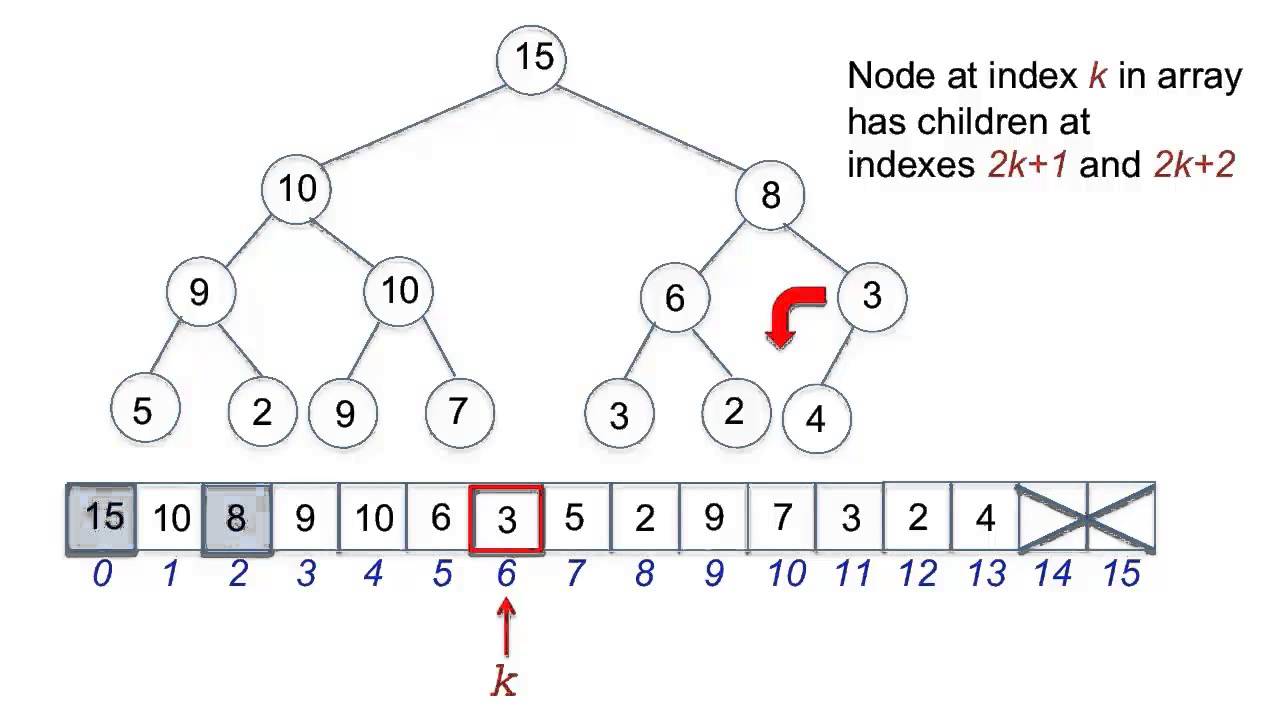
Shape Property : Heap data structure is always a Complete Binary Tree, which means all levels(except last level) of the tree are fully filled. The last level is filled from left to right. Smallest possible height can be log(base 2)N .



Heap Property : All nodes are either [greater than or equal to] or [less than or equal to] each of its children. If the parent nodes are greater than their children, heap is called a Max-Heap, and if the parent nodes are smalled than their child nodes, heap is called Min-Heap.

Binary Heap data structure

An array can be used to simulate a tree in the following way. If we are storing one element at index K in array Ar, then its parent will be stored at index (K - 1)/2 (unless its a root, as root has no parent) and can be access by Ar[ K/2 ], and its left child can be accessed by Ar[ 2*K + 1 ] and its right child can be accessed by Ar[ 2*K + 2 ]. Index of root will be 0 in an array.



Note:

1. If the root is present at index 1 then its parent will be at K/2 and left child will be at 2*K and right child will be at 2*K+1.
2. Generally heap stores (key,value) pairs where key is the priority of the node. The higher is the key value the more is it's priority.

Why don't we use Binary Tree data structure to implement heap?

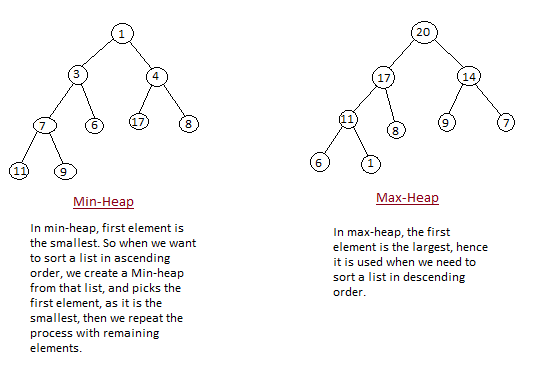
Heaps are usually simpler to implement using arrays than complete binary trees. Arrays require less memory overhead (elements can be stored directly in an array, without having to allocate tree nodes and pointers and everything), potentially speedier performance (largely due to the memory locality of using a single contiguous array). The time to reach parent node in array is O(1) where as in trees it will be O(N). Using binary tree :

1. Search minimum/maximum element : O(1)
2. Adding new element : O(log(N))
3. Removing minimum/maximum element : O(log(N))
4. Accessing parent : O(log(N))
5. Space complexity : O(N) + overhead

Using arrays :

1. Search minimum/maximum element : O(1)
2. Adding new element : O(log(N))
3. Removing minimum/maximum element : O(log(N))
4. Accessing parent : O(1)
5. Space complexity : O(N)

Types of Heap



**Max heap**

In this type of heap, the value of parent node will always be greater than or equal to the value of child node across the tree and the node with highest value will be the root node of the tree.

**Max heap implementation**

We shall use this example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

Building heap :

We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

For Input → 35 33 42 10 14 19 27 44 26 3

Step 1 − Create a new node at the end of heap.

Step 2 − Assign new value to the node.

Step 3 − Compare the value of this child node with its parent.

Step 4 − If value of parent is less than child, then swap them.

Step 5 − Repeat step 3 & 4 until Heap property holds.

Do the above steps for all input nodes.

Complexity of building heap : At the bottommost level, there are 2^(h) nodes (h is height), but we do not call heapify on any of these, so the work is 0. At the next to level there are 2^(h − 1) nodes, and each might move down by 1 level. At the 3rd level from the bottom, there are 2^(h − 2) nodes, and each might move down by 2 levels. As you can see not all heapify operations are O(log n), this is why we get O(n) time complexity and not O(nlogn).

Removing a node :

Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

Step 1 − Remove root node.

Step 2 − Move the last element of last level to root and decrement the heap size.

Step 3 − Compare the value of this node with the children.

Step 4 − Swap parent node with the larger child.

Step 5 − Repeat step 3 & 4 until Heap property holds.

Complexity of removing a node : As we move from root to last level the time complexity will be O(height) i.e. O(logN).

import java.util.\*;

class Pair<T> {

int p; // priority

T data; // value

Pair(int priority, T data){

p = priority;

this.data = data;

}

}

public class HeapClass {

ArrayList<Pair<Integer>> heap = new ArrayList<>();

public HeapClass(Pair<Integer> arr[]){

//initial capacity of heap array = arr.length

heap = new ArrayList<>(arr.length);

createMaxHeap(arr);

}

public boolean isEmpty(){

return heap.isEmpty();

}

// to find the maximum element

public Pair<Integer> findMax() throws Exception{

if(heap.isEmpty()){

throw new Exception();

}

return heap.get(0);

}

private void createMaxHeap(Pair<Integer>[] arr){

for(int i = 0;i < arr.length;i++){

heap.add(arr[i]);

createHeap(i);

}

}

//maintains heap property

private void createHeap(int index){

if(index <= 0){

return;

}

int parent = (index - 1)/2;

Pair<Integer> temp = heap.get(index);

if(heap.get(parent).p < heap.get(index).p){

// if priority of parent element is less than that of child swap them

temp = heap.get(index);

heap.set(index, heap.get(parent));

heap.set(parent, temp);

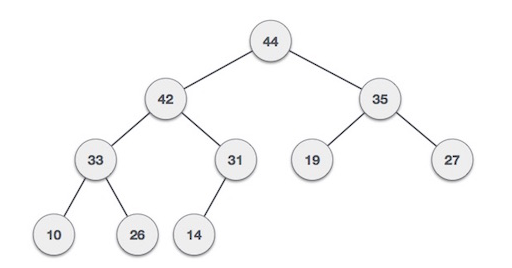
//recursive call for parent

createHeap(parent);

}

}

}



**Min Heap**

In this type of heap, the value of parent node will always be less than or equal to the value of child node across the tree and the node with lowest value will be the root node of tree.

Applications

**1. Priority Queue**

Priority Queue is similar to queue where we insert an element from the back and remove an element from front, but with a one difference that the logical order of elements in the priority queue depends on the priority of the elements. The element with highest priority will be moved to the front of the queue and one with lowest priority will move to the back of the queue. Thus it is possible that when you enqueue an element at the back in the queue, it can move to front because of its highest priority.

import java.util.\*;

class Pair<T> {

int p; // priority

T data; // value

Pair(int priority, T data){

p = priority;

this.data = data;

}

}

class PriorityQueue {

ArrayList<Pair<Integer>> heap = new ArrayList<>()

public PriorityQueue(){

heap = new ArrayList<>();

}

public boolean isEmpty(){

return heap.isEmpty();

}

public Pair<Integer> findMax() throws Exception{

if(heap.isEmpty()){

throw new Exception();

}

return heap.get(0);

}

private void createHeap(int index){

if(index <= 0){

return;

}

int parent = (index - 1)/2;

Pair<Integer> temp = heap.get(index);

if(heap.get(parent).p < heap.get(index).p){

temp = heap.get(index);

heap.set(index, heap.get(parent));

heap.set(parent, temp);

createHeap(parent);

}

}

public Pair<Integer> remove() throws Exception{

if(heap.isEmpty()){

throw new Exception();

}

Pair<Integer> temp = heap.get(0);

heap.set(0, heap.get(heap.size()-1));

heap.remove(heap.size()-1);

downHeapify(0);

return temp;

}

public void add(Pair<Integer> newElement){

heap.add(newElement);

createHeap(heap.size()-1);

}

//maintains heap property

private void downHeapify(int index){

int left = index\*2 +1;

int right = index\*2 +2;

int greatest = index;

if(left < heap.size()){

if(heap.get(index).p < heap.get(left).p) {

greatest = left;

}

if(right < heap.size() && heap.get(greatest).p < heap.get(right).p){

greatest = right;

}

if(greatest != index){

Pair<Integer> temp = heap.get(index);

heap.set(index, heap.get(greatest));

heap.set(greatest, temp);

downHeapify(greatest);

}

}

}

}

public class usePriorityQueue {

private static Scanner s = new Scanner(System.in);

public static void main(String[] args) throws Exception {

PriorityQueue queue = new PriorityQueue();

System.out.println("Enter priority and value : ");

// taking random input

int key = s.nextInt();

while(key != -1){

int value = s.nextInt();

PQPair<Integer> newElement = new PQPair<>(key,value);

queue.add(newElement);

System.out.println("Enter priority and value : ");

key = s.nextInt();

}

// maximum priority

PQPair<Integer> temp = queue.findMax();

System.out.println("Maximum : "+temp.p +" "+temp.data);

//adding new element

queue.add(new PQPair<Integer>(12,20));

temp = queue.findMax();

System.out.println("\nMaximum after addition : "+temp.p +" "+temp.data);

//removing maximum priority

queue.remove();

temp = queue.findMax();

System.out.println("\nMaximum after dequeue : "+temp.p +" "+temp.data);

}

}

Example :

Enter priority and value :

2 100

Enter priority and value :

6 12

Enter priority and value :

0 23

Enter priority and value :

10 33

Enter priority and value :

9 74

Enter priority and value :

-1

Maximum : 10 33

Maximum after addition : 12 20

Maximum after dequeue : 10 33

**2. HeapSort**

We can use heaps in sorting the elements in a specific order in efficient time. Let’s say we want to sort elements of array Arr in ascending order. We can use max heap to perform this operation.

We build the max heap of elements stored in Arr, and the maximum element of Arr will always be at the root of the heap.

Leveraging this idea we can sort an array in the following manner.

Algorithm :

Step 1 - Initially we will build a max heap of elements in Arr.

step 2 - Now the root element that is Arr[ 0 ] contains maximum element of Arr. After that, we will exchange this element with the last element of Arr and will again build a max heap excluding the last element which is already in its correct position and will decrease the length of heap by one.

Step 3 - We will repeat the step 2, until we get all the elements in their correct position.

Step 4 - We will get a sorted array.

Complexity of heapsort : First we will build heap (O(n)) and then heapify for every element (O(n\*logn)). Space complexity will be O(n).

public class HeapSort {

static void heapsort(int a[], int length){

//build heap O(n)

buildheap(a, length);

int heapsize, temp;

heapsize = length - 1;

//sorting n\*logn

for(int i=heapsize; i >= 0; i--){

temp = a[0];

a[0] = a[heapsize];

a[heapsize] = temp;

heapsize--;

downHeapify(a, 0, heapsize);

}

for(int i:a){

System.out.print(i + " ");

}

}

static void buildheap(int a[], int length){

int i, heapsize;

heapsize = length - 1;

for( i=(length/2); i >= 0; i--){

downHeapify(a, i, heapsize);

}

}

static void downHeapify(int a[], int i, int heapsize){

int l, r, largest, temp;

l = 2\*i;

r = 2\*i + 1;

if(l <= heapsize && a[l] > a[i]){

largest = l;

}

else{

largest = i;

}

if( r <= heapsize && a[r] > a[largest]){

largest = r;

}

if(largest != i){

temp = a[i];

a[i] = a[largest];

a[largest] = temp;

downHeapify(a, largest, heapsize);

}

}

public static void main(String[] args) {

int arr[] = {35, 33, 42, 10, 14, 19, 27, 44, 26, 3};

heapsort(arr,arr.length);

}

}

OUTPUT :

3 10 14 19 26 27 33 35 42 44

