**What is Binary Heap?**

A Binary heap is a complete binary tree. There are two types of it.

1. Max Heap is a complete binary tree where value of every internal node is greater than its child.
2. Min Heap is a complete binary tree where value of every internal node is less than its child

e.g.

Max\_Heap Min\_Heap

The benefit of designing this structures is that you will always get biggest element in case of Max Heap and Smallest element in case of Min Heap as a root node.

**How binary tree is represented in simple data Structure like array?**

We will represent same examples of binary trees in arrays.

1. Max Heap

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 25 | 15 | 10 | 7 |  |

1. Min Heap

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 7 | 15 | 26 | 77 | 24 |

* Root node is always placed at 0th location in array.
* For any node at location ‘i’ ,

I’s parent would be at location : i/2.

I’s left child would be at location : I\*2

I’s right child would be at location : I \*2 + 1

Since arrays are indexed from 0 instead of 1, the formulas will change as follows:

e.g for node 15 at location i=2,

Parent = (1/2) = 2

Left= (2\*2)=4 (77 is left child of )

right= (2\*2)+1=5 (24 is left child of )

**Heap Sort:**

By heap sort we mean max heap sort. i.e. The algorithm uses the concept of heapify(A,length) to get the maximum element at 0th element in an array then replaces it with the last element. After this decrease the index, now call heapify (A,length-1) again. Repeat the procedure until all elements are in sorted order. Since we are using max heap , the elements would be sorted in descending order. For sorting in Ascending order, use min heap.

**Pseudocode:**

Heap\_Sort ( Unsorted\_array A )

1. mainBuild Heap(A)
2. For I = length of array down to 1 do
3. Swap element 0 (biggest) with the last element in the array.
4. Call Heapify(A , length-1) to get the next biggest in the array.
5. End for
6. Print Sorted Array.

**Implementation:**

**-------------------------------------------------------------------------------------------------------------------------------------------------**

**//Main class : This class includes all the necessary Heap functions.**

**public class HeapSorter implements sort** {

private int[] A;

//Constructor

public HeapSorter(int[] a) {

//this.A = a;

A=new int[a.length+1];

//System.out.println("Length of A: "+A.length);

A[0] = 0;

for (int i = 1; i<= a.length; i++) {

A[i] = a[i-1];

}

}

@Override

public void sorter() {

Heapsort();

}

private void Heapsort() {

int len = A.length - 1;

build\_max\_heap(len); ……………………. (2)

int temp;

for (int i=len; i>=2; i--) {

//exchange A[1} and A[i}

temp = A[1];

A[1]= A[i];

A[i]= temp;

max\_heapify(i, i-1);

}

}

@Override

public void printData() {

for(int i=1; i < A.length; i++) {

System.out.println(A[i]);

}

}

public int getLength() {

return A.length;

}

public void build\_max\_heap(int len) { …………………….(3)

for (int i = (len/2)-1; i >= 1; i--) {

max\_heapify(i, len);

}

}

private void max\_heapify(int parent, int len) { …………………….(4)

**int** left=parent\*2;

**int** right= left+1;

**int** childToCompare=parent;

**if**(left<=len && right<=len){

//System.out.println(arr[parent]+" left:"+arr[left]+" right:"+arr[right]+" ");

**if**(arr[left]> arr[right]){

childToCompare=left;

}**else**{

childToCompare=right;

}

}**else** **if**(right<=len){

//System.out.println(arr[parent]+" right:"+arr[right]+" ");

childToCompare=right;

}**else** **if**(left<=len){

//System.out.println(arr[parent]+" left:"+arr[left]+" left:"+left);

childToCompare=left;

}**else**{

//System.out.println("hmm");

}

**if**(childToCompare!= parent){

System.***out***.println("swap:"+parent+" :"+childToCompare);

printArray(arr);

**if**(arr[childToCompare]>arr[parent]){

temp = A[parent];

A[parent] = A[childToCompare];

A[childToCompare] = temp;

max\_heapify(childToCompare,len-1); ……………………… (5)

}

}

}

}

**//Client Class: This class is the calling class which randomly generates an array and calls sort of the main class Sorter.**

import java.util.Random;

public class ClientClass {

public static void main(String[] args) { …………………………..Execution begins here.

final int LEN = 10;

Random randomGenerator = new Random();

int[] a = new int[LEN];

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

a[i] = randomGenerator.nextInt(100); ….. ……………(1)

}

System.out.println("Array Length:"+a.length);

System.out.println("Original Array:");

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

System.out.println(a[i]);

}

sort s = new HeapSorter(a);

s.sorter();

System.out.println("After sort");

s.printData();

}

}

**//Sort Interface : considering the reusability and for future use**

public interface Sort {

//int[] a;

public void sorter();

public void printData();

}

----------------------------------------------------------------------------------------------------------------------------------------------------------------

We will take an example to explain the above implementation. Lets assume that the random generator in the class Client method main() has generated the following array.of length 6.(Marked as (1) in the code)

**Unsorted array:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 33 | 10 | 7 | 65 | 23 |

**Binary Tree Representation :**

The array does not follow the max heap property. So we will call build\_max\_heap(len); (marked by (2) in the code)

The build\_max\_heap(len) function will structure the tree such that every internal node’s value is more than its children.

i.e no node violates max heap property.

**Lets understand implementation of the function build\_max\_heap(len)** **(marked by (3) in code)**

To check whether a internal node is bigger than its children , we only need to iterate over internal nodes only as leaf nodes have no children and hence they always satisfy max heap property. Hence we only call max\_heapify(i, len); , for the nodes i/2 down to 0. (refere code marked by (3)) The internal nodes can be easily computed by the formula (i/2) for binary trees .

Lets check the formula for the given array, i=6 .

so (i/2)= 3.

Our formula is correct as nodes at locations 1,2,3 are the internal nodes and at 4,5,6 are the leaf nodes.

**Lets visualize the iterations over max\_heapify(i, len); (marked by (4) in code)**

As mentioned the function checks if each internal node satisfies the max heap property , if not corrects it.

**Iteration 1:** Values passed to the function are len=6 ; parent=3

arr[parent]=10; left=6 ; arr[left]=23 , right child is null . hence compare the parent with left child.

Since the left child is bigger than parent. Swap.

**Unsorted array:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 33 | 23 | 7 | 65 | 10 |

**Binary Tree Representation :**

Now the function would be called recursively for value 10.(refer code ) since it is a leaf node , the recursion will end.

**Iteration 2:**

Values passed to the function are len=5 ; parent=2

arr[parent]=33 ; left=3 ; arr[left]=7, right=4 , arr[right]=65 . Since 65> 7, hence compare the parent with right child.

Since the right child is bigger than parent. swap

**Unsorted array:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 65 | 23 | 7 | 33 | 10 |

**Binary Tree Representation :**

Recursive call to **max\_heapify(i, len)** for value 33. Since it is a leaf, the recursion will break.

**Iteration 3:**

Values passed to the function are len=5 ; parent=1

arr[parent]=1 ; left=2 ; arr[left]=65, right=3 , arr[right]=23 . Since 65> 23, hence compare the parent with left child.

Since the left child is bigger than parent. swap

**Unsorted array:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 65 | 1 | 23 | 7 | 33 | 10 |

**Binary Tree Representation :**

**Iteration 3 :Recursive call** : (refer code marked by (5))

Values passed to the function are len=5 ; parent=2

arr[parent]=1 ; left=4 ; arr[left]=7, right=5 , arr[right]=33 . Since 33> 7, hence compare the parent with right child.

Since the right child is bigger than parent. swap

**Unsorted array:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 65 | 33 | 23 | 7 | 1 | 10 |

**Binary Tree Representation :**

Now our tree satisfies MAX HEAP property . but the array is still unsorted.