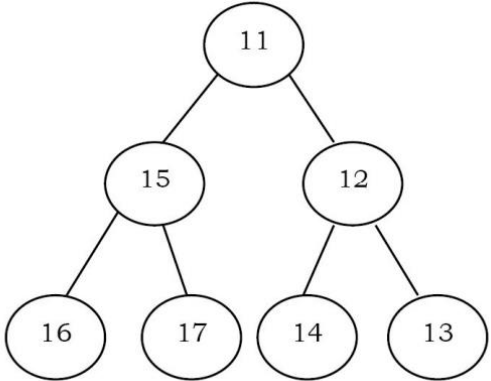
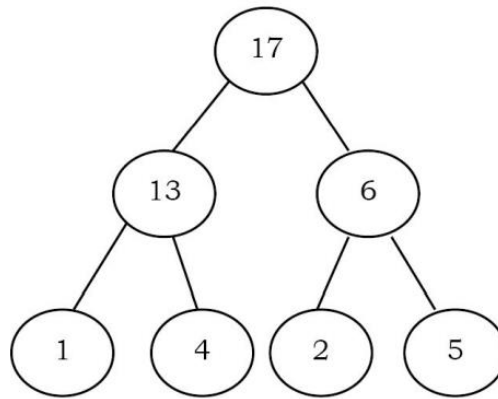


Name	Hatim Yusuf Sawai
UID no.	2021300108
Experiment No.	9

AIM:	Program on Binary Heaps
Program	
PROBLEM STATEMENT:	Write a program to demonstrate Heapsort using Max Heap implemented using arrays, using top-down approach
THEORY:	<p>Binary Heap:</p> <p>A heap is a tree with some special properties. The basic requirement of a heap is that the value of a node must be $>$ (or $<$) than the values of its children. This is called heap property. In binary heap each node may have up to two children. A heap should always form a Complete Binary Tree.</p> <p>Types of Heaps:</p> <p>Based on the property of a heap we can classify heaps into two types:</p> <p>1. Min heap: The value of a node must be less than or equal to the values of its children:</p>  <pre> graph TD 11((11)) --- 15((15)) 11 --- 12((12)) 15 --- 16((16)) 15 --- 17((17)) 12 --- 14((14)) 12 --- 13((13)) </pre>

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



Representation of Heap:

Since heaps are forming complete binary trees, there will not be any wastage of locations. Assume index starts at 0:

Parent: $(i-1)/2$

Left Child: $2i+1$ & **Right Child:** $2i+2$

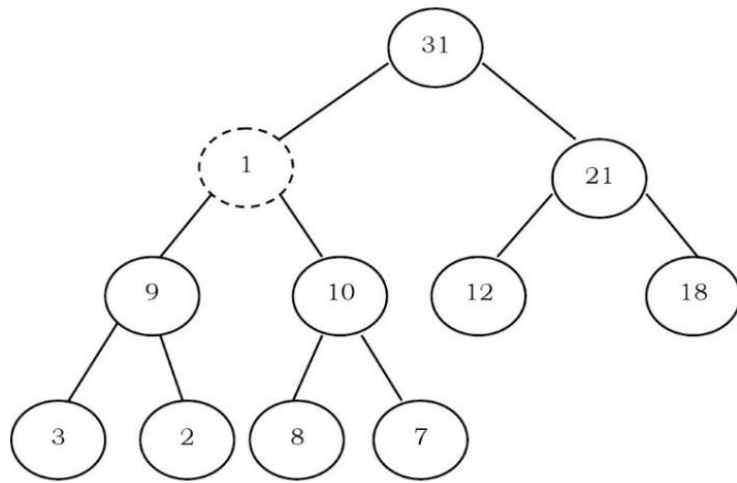
17	13	6	1	4	2	5
0	1	2	3	4	5	6

Heapifying the Tree:

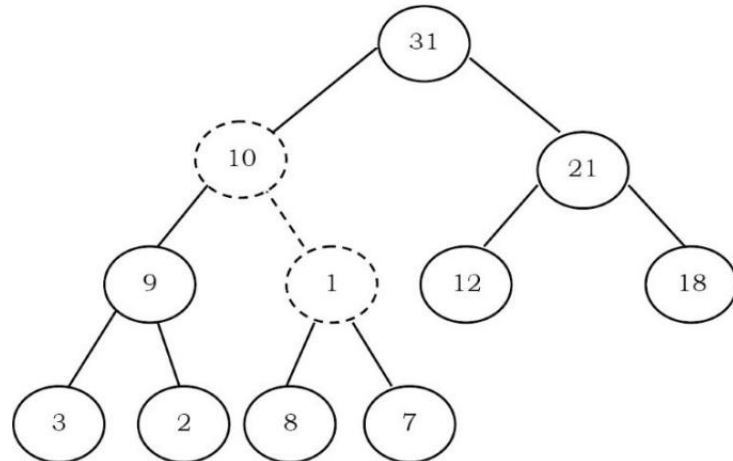
After inserting an element into heap, it may not satisfy the heap property. In that case we need to adjust the locations of the heap to make it heap again. This process is called heapifying. In **maxheap**, to heapify an element, we must find the maximum of its children and swap it with the current element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

Example:

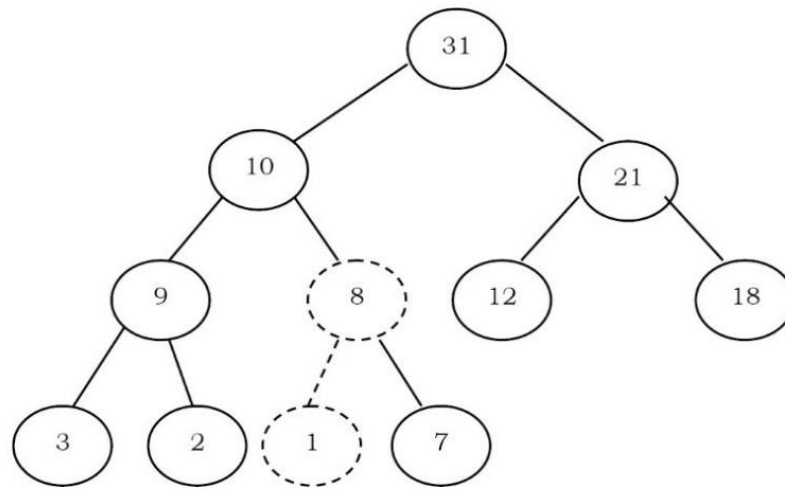
In the below heap, the element 1 is not satisfying the heap property.



To heapify 1, find the maximum of its children and swap with it:



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 shifting larger elements towards the root, this process is sometimes called **shift Up**.

Heapsort:

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

ALGORITHM:

1. Create **Heap** class with `int[] heap` Array as main member
2. Create `int count`, `capacity` members for traversing of array
3. Initialize constructor which takes `capacity` as argument and sets the `count` to 0 and initializes the heap array with "`capacity`" as size
4. Create Methods for Max Binary Heap including heapifying & Heapsort methods used for sorting

Parent Method:

1. if `i <= 0` or `i >= count`:
2. return -1

3. return $(i-1)/2$ → returns the parent of **[i]**

leftChild Method:

1. left = $2*i+1$
2. if left >= count then return -1
2. return left → returns left child of **[i]** if exists

rightChild Method:

1. right = $2*i+2$
2. if right >= count then return -1
2. return right → returns right child of **[i]** if exists

heapify Method:

1. Initialize maxIndex = i, left = **leftChild(i)** & right = **rightChild(i)**
2. if left is not equal to -1 and heap[left] is more than heap[maxIndex]
3. then: maxIndex = left
4. if right is not equal to -1 and heap[right] more than heap[maxIndex]
5. then: maxIndex = right
6. if maxIndex is not equal to i
7. then: initialize temp = heap[i]
8. Set heap[i] = heap[maxIndex]
9. Set heap[maxIndex] = temp
10. Recursive call to **heapify(maxIndex)**

insert Method:

1. if count equals capacity
2. then: **resizeHeap()** → adjust heap size manually
3. Now: increment count by 1
4. Set heap[count-1] = data → given as argument
5. shiftUp(count-1) → to heapify the tree

resizeHeap Method:

1. Initialize int[] array oldHeap = heap
2. Reinitialize heap = new int[2*capacity] → with new capacity
3. for i=0 to i<capacity → copies oldheap onto the new heap
4. Set heap[i] = oldHeap[i]

5. End for loop
6. capacity = 2*capacity -> update capacity variable
7. oldHeap = null -> destroy temp heap

heapSort Method:

1. initialize old_size = count;
2. for i=n-1 to i>0
3. initialize temp = heap[0] -> store max element
4. Set heap[0] = heap[i] -> swap last element with first
5. heap[i] = temp -> set first to last element
6. decrement count by 1 -> to leave out sorted elements
7. heapify(0) reheapify the remaining elements after swapping
9. print the heap
10. End for loop
11. update count = old_size

printHeap Method:

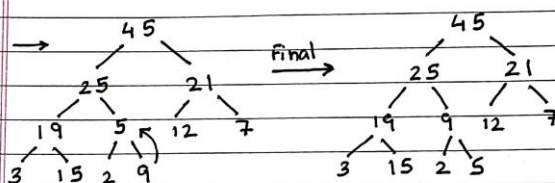
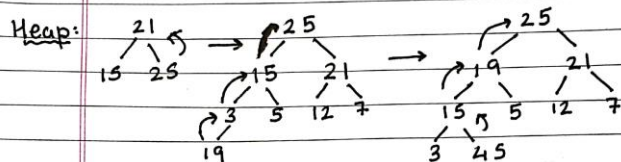
1. for i=0 to i<n -> for loop to traverse array
2. Apply if checks for formatting:
3. if I is more than or equal to count
4. print heap[i]
5. else if I equals count-1
6. print heap[i]
7. else
8. print heap[i]

PROBLEM
SOLVING:

HATIM SAWAI

9] MAX HEAPSORT

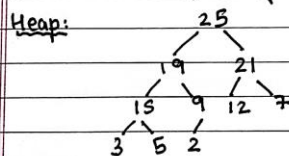
Input: 21, 15, 25, 3, 5, 12, 7, 19, 45, 2, 9



Heap: 45, 25, 21, 19, 9, 12, 7, 3, 15, 2, 5

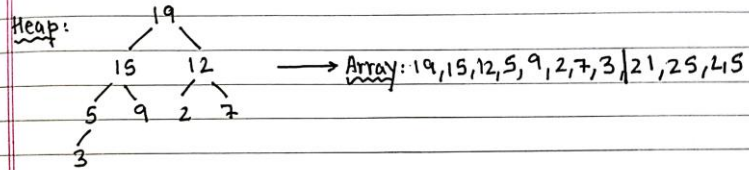
HeapSort

1. Remove 45 & swap with 5, Reheapify

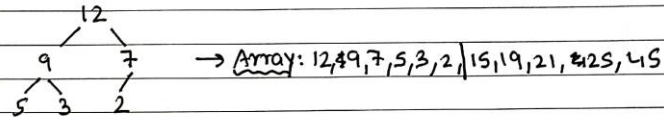


Array: 25, 19, 21, 15, 9, 12, 7, 3, 5, 2, 45

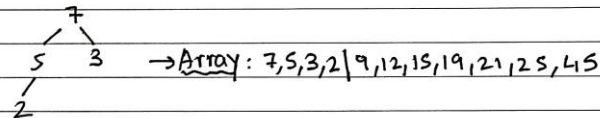
2. Remove 25, 21 & repeat process:



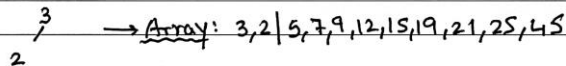
3. Remove 19, 15 & repeat process



4. Remove 12, 9 & repeat process



→ Remove 7, 5 & repeat process:



→ Remove 3, 2 → get sorted Array:

Array: 2, 3, 5, 7, 9, 12, 15, 19, 21, 25, 45

PROGRAM:

HeapCheck.java:

```

import java.util.Scanner;
import heapsd.Heap;
public class HeapCheck {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int capacity = 8;
        Heap heap = new Heap(capacity, 1);
        int choice, flag;
        while(true) {
            System.out.println("Select an option:\n1.Insert\n2.Build
Heap\n3.Heap Sort\n4.Destroy Heap");
            choice = sc.nextInt();
            switch(choice) {

```


case 1:

```
System.out.println("Enter the element to be inserted: ");
int element = sc.nextInt();
heap.insert(element);
System.out.print("Heap: ");
heap.printHeap();
break;
```

case 2:

```
System.out.print("Enter no. of elements to be inserted: ");
int n = sc.nextInt();
System.out.print("Enter the elements: ");
for(int i=0;i<n;i++) {
    element = sc.nextInt();
    heap.insert(element);
}
System.out.println("Constructed Heap: ");
heap.printHeap();
break;
```

case 3:

```
System.out.print("Enter no. of elements to be sorted: ");
n = sc.nextInt();
int[] A = new int[n];
System.out.print("Enter the elements: ");
for(int i=0;i<n;i++) {
    A[i] = sc.nextInt();
    heap.insert(A[i]);
}
System.out.print("Built Heap: ");
heap.printHeap();
System.out.println("Sorting Heap....");
heap.heapSort(A,n);
System.out.print("Sorted Array: ");
heap.printHeap();
break;
```

case 4:

```
heap.destroyHeap();
```

```

        System.out.println("Heap Destroyed");
        heap.destroy();
        Heap heap = new Heap(capacity,1);
        break;
    default:
        System.out.println("Invalid choice!");
    }
    System.out.print("Do you want to continue?(1/0): ");
    flag = sc.nextInt();
    if(flag==0) {
        break;
    }
}
sc.close();
}
}

```

Heap.java:

```

package heapds;
public class Heap {
    public int[] heap;
    int count,capacity;
    int heap_type;
    public Heap(int capacity,int heap_type) {
        this.heap_type = heap_type;
        this.count = 0;
        this.capacity = capacity;
        this.heap = new int[capacity];
    }
    public int parent(int i) {
        if(i<=0 || i>=count) {
            return -1;
        }
        return (i-1)/2;
    }
}

```

```

public int leftChild(int i) {
    int left = 2*i+1;
    if(left>=count) {
        return -1;
    }
    return left;
}
public int rightChild(int i) {
    int right = 2*i+2;
    if(right>=count) {
        return -1;
    }
    return right;
}
public int getMax() {
    if(count==0) {
        return -1;
    }
    return heap[0];
}
public void shiftUp(int i) {
    int temp = heap[i];
    while(i>0 && temp>heap[parent(i)]) {
        heap[i] = heap[parent(i)];
        i = parent(i);
    }
    heap[i] = temp;
}
public void heapify(int i) {
    int maxIndex = i;
    int left = leftChild(i);
    int right = rightChild(i);
    if(left!=-1 && heap[left]>heap[maxIndex]) {
        maxIndex = left;
    }
    if(right!=-1 && heap[right]>heap[maxIndex]) {

```

```

        maxIndex = right;
    }
    if(maxIndex!=i) {
        int temp = heap[i];
        heap[i] = heap[maxIndex];
        heap[maxIndex] = temp;
        heapify(maxIndex);
    }
}

public int deleteMax() {
    if(count==0) {
        return -1;
    }
    int data = heap[0];
    heap[0] = heap[count-1];
    count--;
    heapify(0);
    return data;
}

public void insert(int data) {
    if(count==capacity) {
        resizeHeap();
    }
    count++;
    heap[count-1] = data;
    shiftUp(count-1);
}

public void resizeHeap() {
    int[] oldHeap = heap;
    heap = new int[2*capacity];
    for(int i=0;i<capacity;i++) {
        heap[i] = oldHeap[i];
    }
    capacity = 2*capacity;
    oldHeap = null;
}

```

```

public void destroyHeap() {
    heap = null;
    count = 0;
}

public void heapSort(int[] A,int n) {
    int old_size = count;
    for(int i=n-1;i>0;i--) {
        int temp = heap[0];
        System.out.println("Swapping "+heap[0]+" and "+heap[i]);
        heap[0] = heap[i];
        heap[i] = temp;
        count--;
        heapify(0);
        printHeap(n);
    }
    count = old_size;
}

public void printHeap() {
    for(int i=0;i<count;i++) {
        System.out.print(heap[i]+" ");
    }
    System.out.println();
}

public void printHeap(int n) {
    for(int i=0;i<n;i++) {
        System.out.print(heap[i]+" ");
    }
    System.out.println();
}
}

```

OUTPUT:

Heap Sort:

```
● PS D:\Data Structures\Exp9> cd "d:\Data Structures"
Select an option:
1.Insert
2.Build Heap
3.Heap Sort
4.Destroy Heap
3
Enter no. of elements to be sorted: 11
Enter the elements: 21 15 25 3 5 12 7 19 45 2 9
Built Heap: 45 25 21 19 9 12 7 3 15 2 5
Sorting Heap....
45 < - > 5
25 19 21 15 9 12 7 3 5 2|45
25 < - > 2
21 19 12 15 9 2 7 3 5|25|45
21 < - > 5
19 15 12 5 9 2 7 3|21|25|45
19 < - > 3
15 9 12 5 3 2 7|19|21|25|45
15 < - > 7
12 9 7 5 3 2|15|19|21|25|45
12 < - > 2
9 5 7 2 3|12|15|19|21|25|45
9 < - > 3
7 5 3 2|9|12|15|19|21|25|45
7 < - > 2
5 2 3|7|9|12|15|19|21|25|45
5 < - > 3
3 2|5|7|9|12|15|19|21|25|45
3 < - > 2
2|3|5|7|9|12|15|19|21|25|45
Sorted Array: 2 3 5 7 9 12 15 19 21 25 45
Do you want to continue?(1/0): 0
○ PS D:\Data Structures\Exp9> █
```

CONCLUSION:

In this experiment, we learned how to implement Max Binary Heap using Array & how to use Max heap to perform heap sort on an array of elements.