5

HEAP

Unit Structure:

5.1. INTRODUCTION:

i. A heap is an efficient semi-ordered data structure for storing a collection of orderable data.

ii. The main difference between a heap and a binary tree is the heap property. In order for a data structure to be considered a heap, it must satisfy the heap property which is discussed in the further section

5.2. DEFINITION :

A Heap data structure is a binary tree with the following properties:

i. It is a ***complete binary tree***; that is, each level of the tree is completely filled, except possibly the bottom level. At this level, it is filled from left to right.

ii. It satisfies the ***heap-order property***: The data item stored in each node is greater than or equal to the data items stored in its children.

ie. If *A* and *B* are elements in the heap and *B* is a child of *A*, then key(*A*) ≥ key(*B*).

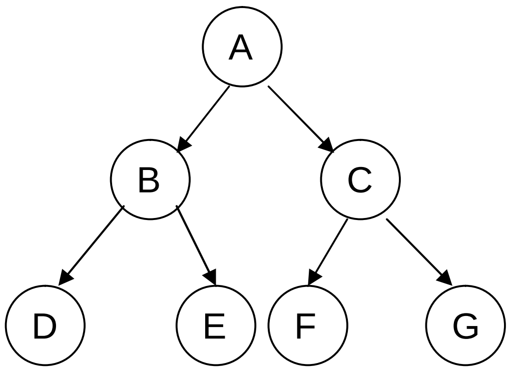


Fig 5.1

The above Tree is a Complete Binary tree and is a HEAP if we assume that the data item stored in each node is greater than or equal to the data items stored in its children

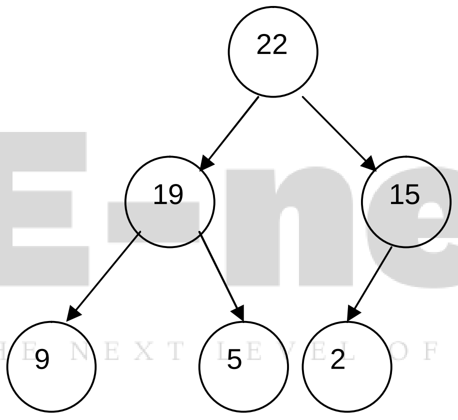


Fig 5.2

The above tree is a heap because

1) it is a complete binary tree
2) it satisfies the heap order property of heap

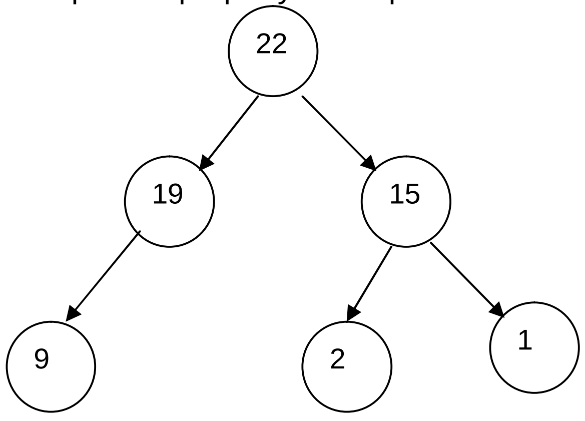


Fig 5.3

The above tree is not a heap since it is not a complete binary tree.

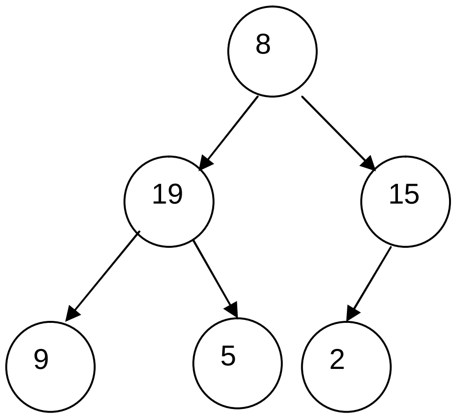


Fig 5.4

The above tree is not a heap since it does not satisfy the heap order property.i.e the value of data the root is less than the value at its child nodes.

5.3. TYPES OF HEAPS:

Depending upon the value of data items in the nodes of the binary tree, a heap could be of the following two types:

1) max-heap

2) min-heap

5.3.1 Max-Heap: A *max-heap* is often called as a Heap.

Definition

A max-heap is a binary tree structure with the following properties:

1) The tree is complete or nearly complete.
2) The key value of each node is greater than or equal to the key value of its children.
   .i.e If *A* and *B* are elements in the heap and *B* is a child of *A*, then key(*A*) ≥ key(*B*).

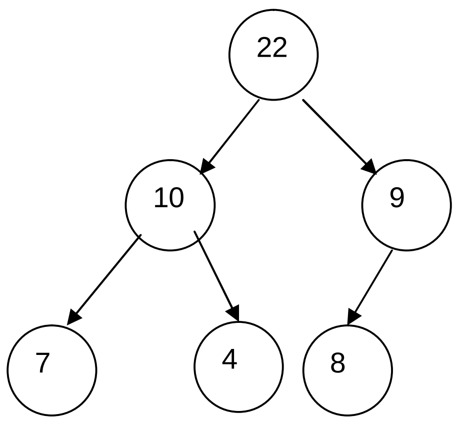
5.3.2 Min-Heap

Definition

A min-heap is a binary tree structure with the following properties:

1) The tree is complete or nearly complete.
2) The key value of each node is less than or equal to the key value of its children.

.i.e If *A* and *B* are elements in the heap and *B* is a child of *A*, then key(*A*) ≤ key(*B*).



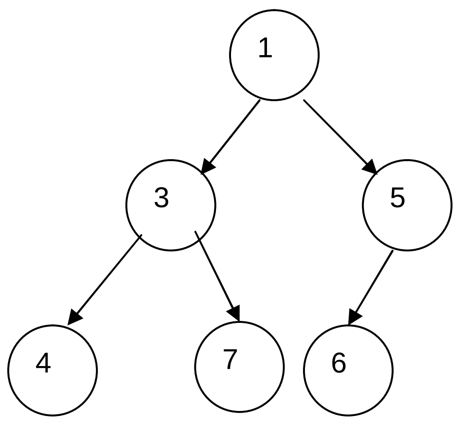


Fig 5.5 **Fig 5.6**

Max-heap (or heap) min-heap

5.4. REPRESENTATION OF HEAP IN MEMORY:

Heaps are represented in memory sequentially by means of an array.

Consider the following diagrams :

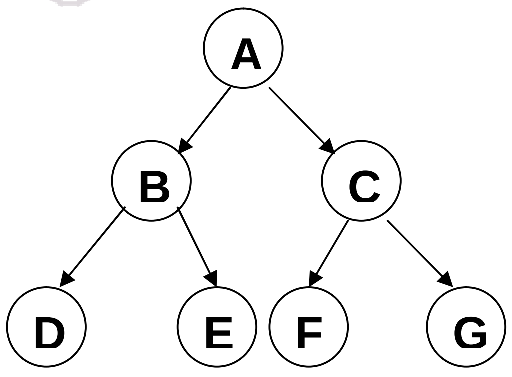


Fig 5.7.a

CONCEPTUAL Representation

1	2	3	4	5	6	7
A	B	C	D	E	F	G

Fig 5.7.b

PHYSICAL representation (in memory)

The root of the tree is TREE [1].

The left and right child of any node TREE [K] are TREE[2K] & TREE[2K+1] respectively

As in the diagram above,

- TREE [1] is the root node which contains A. (K=1)
- Left Node of B i.e left node of TREE [2] is given by TREE [2K] which is TREE [2] = B
- Right Node of A i.e. right node of TREE [1] is given by TREE [2K+1] which is TREE [3] = C

This representation could also be shown as

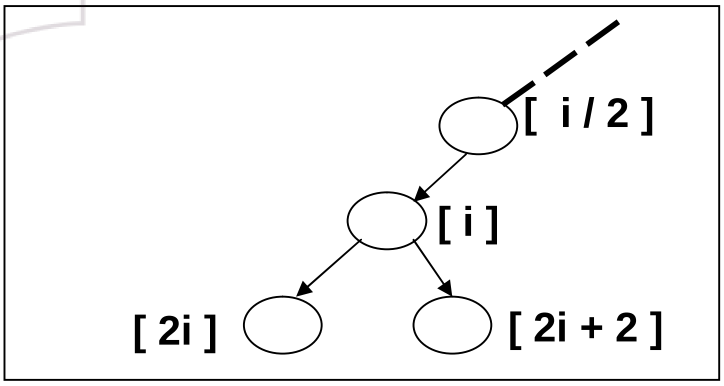


Fig 5.8

Note : In case the root node is Tree[0] i.e K=0, left node is given by 2K+1 and Right node is given by 2K+2

5.5. BASIC HEAP ALGORITHMS

5.5.1 ReheapUp: repairs a "broken" heap by floating the last element up the tree until it is in its correct location.

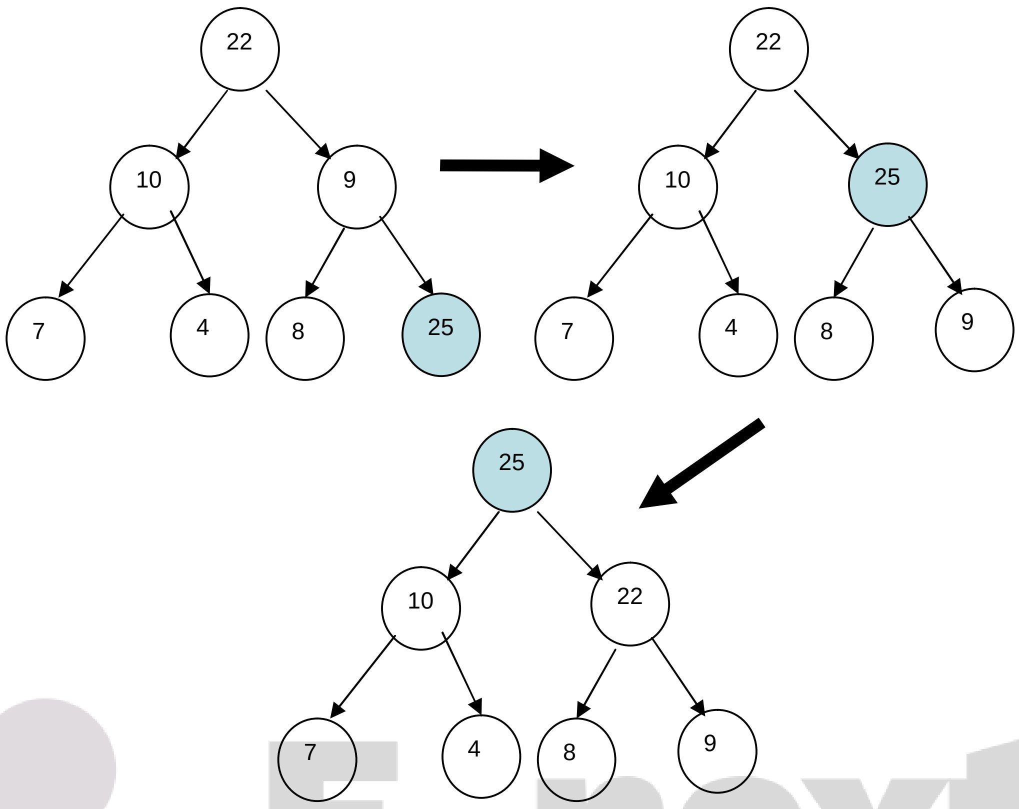


Fig 5.9 ReHeapUp

Algorithm ReheapUp(val position<int>)

This algorithm re-establishes heap by moving data in position up to its correct location. It is assumed that all data in the heap above this position satisfy key value order of a heap, except the data in position.

After execution Data in position has been moved up to its correct location.
The function ReheapUp is recursively used.

```
1.  if (position<> 0)// the parent of position exists.
        1. parent = (position-1)/2
        2. if (data[position].key > data[parent].key)
            1. swap(position, parent) // swap data at position with data
                at parent.
            2. ReheapUp(parent)
2. return
EndReheapUp
```

5.5.2 Reheapdown

Repairs a "broken" heap by pushing the root of the sub-tree down until it is in its correct location.

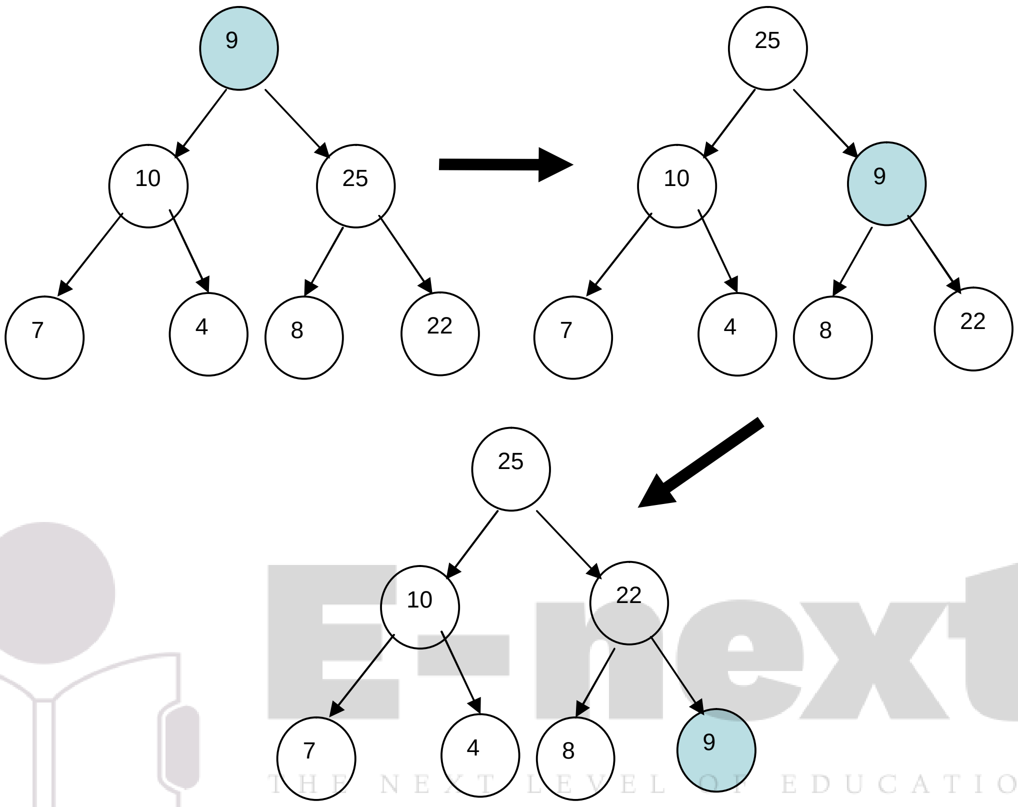


Fig 5.10 ReHeapDown

Algorithm: ReheapDown(val position<int>, val lastPosition<int>)

This algorithm re-establishes heap by moving data in position down to its correct location. It is assumed that all data in the sub-tree of position satisfy key value order of a heap, except the data in position.

After execution Data in position has been moved down to its correct location.

The function ReheapDown is recursively used.

```
1. leftChild= position*2 + 1
2. rightChild= position*2 + 2
3. if ( leftChild<= lastPosition)       // the left child of position exists.
```

```
        1. if (rightChild<= lastPosition) AND ( data[rightChild].key > data[leftChild].key )
                1. child = rightChild
        2. else
                1. child = leftChild  // choose larger child to compare with data in position
        3. if ( data[child].key > data[position].key )
                1. swap(child, position) // swap data at position with data at child.
                2. ReheapDown(child, lastPosition)
4. return
End ReheapDown
```

5.5.3 Inserting into a Heap

- A new node (say key k) is always inserted at the end of the heap. After the insertion of a new key k, the heap-order property may be violated.
- After the insertion of the new key k, the heap-order property may be violated.
- Algorithm heapUp restores the heap-order property by swapping k along an upward path from the insertion node
- heapUp terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k.

Algorithm: InsertHeap(val DataIn<DataType>) // *Recursive version.*

This algorithm inserts new data into the min-heap.

After execution DataIn has been inserted into the heap and the heap order property is maintained.

The algorithm makes use of the recursive function ReheapUp.

```
1. if(heap is full)
        1. return overflow
2. else
        1. data[count] = DataIn
        2. ReheapUp(count)
```

```
    3. count= count+ 1
    4. return success
End InsertHeap
```

5.5.4 Deleting from a Heap

- Removes the minimum element from the min-heap .i.e. root
- The element in the last position (say k) replaces the root.
- After the deletion of the root and its replacement by k, the heap-order property may be violated.
- Algorithm heapDown restores the heap-order property by swapping key k along a downward path from the root
- heapDown terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k

DeleteHeap(ref MinData<DataType>)
This algorithm removes the minimum element from the min-heap.

After execution MinData receives the minimum data in the heap and this data has been removed and the heap has been rearranged.

The algorithm makes use of the recursive function ReheapDown.

```
1. if(heap is empty)
    1. return underflow
2. else
    1. MinData= Data[0]
    2. Data[0] = Data[count -1]
    3. count= count-1
    4. ReheapDown(0, count -1)
    5. return success
End DeleteHeap
```

5.5.5 Building a Heap

BuildHeap(val listOfData<List>)

The following algorithm builds a heap from data from listOfData.

It is assumed that listOfData contains data that need to be inserted into an empty heap.

The algorithm makes use of the recursive function ReheapUp.

```
1. count= 0
2. repeat while (heap is not full) AND (more data in listOfData)
        1. listOfData.Retrieve(count, newData)
        2. data[count] = newData
        3. ReheapUp( count)
        4. count= count+ 1
3. if (count< listOfData.Size() )
        1. return overflow
4. else
        1. return success
End BuildHeap
```

Example:

Build a heap from the following list of numbers

44, 30, 50, 22, 60, 55, 77, 55

This can be done by inserting the eight elements one after the other into an empty heap H.

Fig 5.11.a to 6.11.h shows the respective pictures of the heap after each of the eight elements has been inserted.

The dotted lines indicate that an exchange has taken place during the insertion of the given item of information.

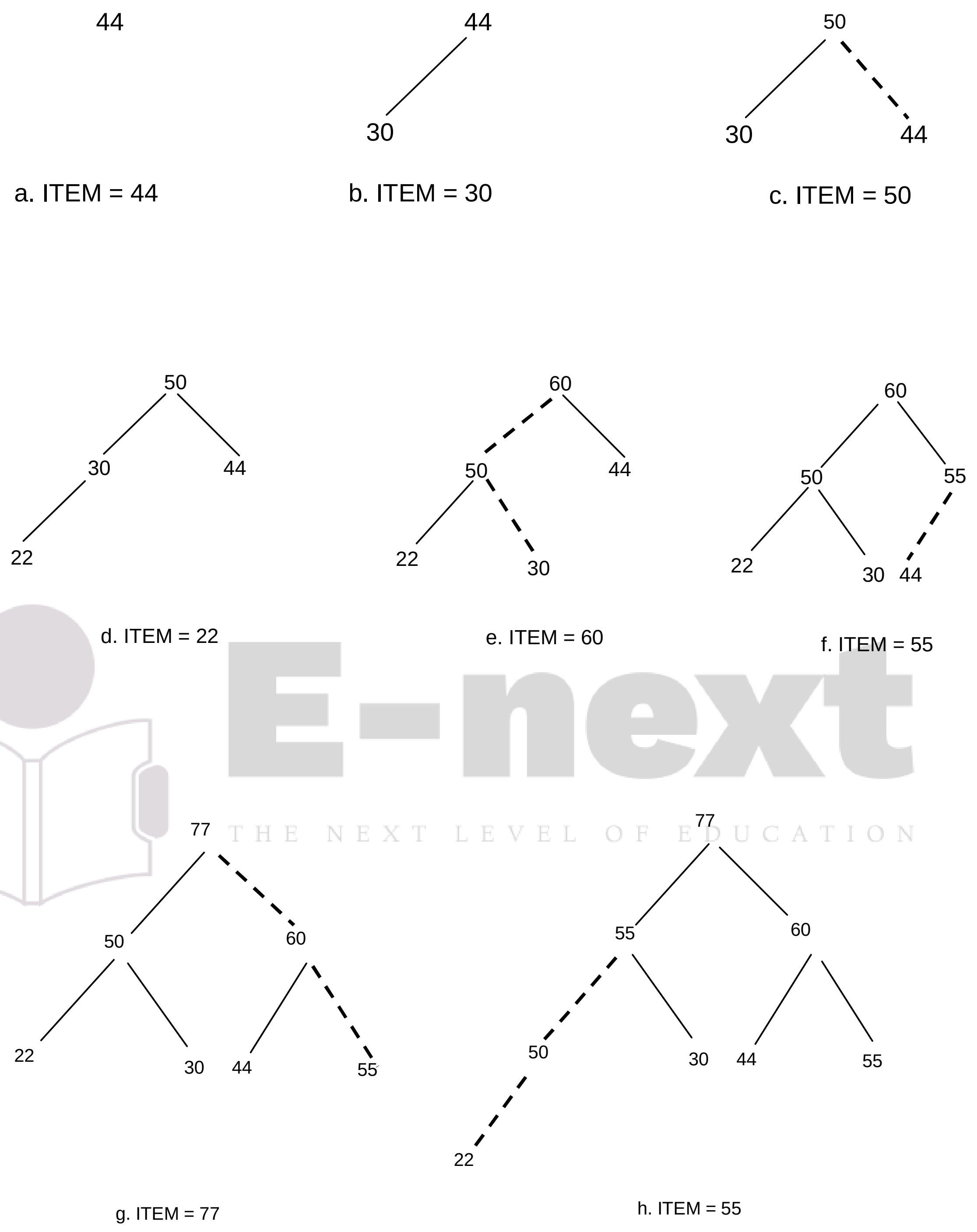


Fig 5.11 Building a heap

❖❖❖❖