**TREES**

**Learning Material**

**Basic Terminology:**

**1. Node**: It is a main component of tree. It stores the actual data and links to other nodes.

Data

Link

Link

**Structure of a node in Tree**

**2. Link / Edge / Branch:** Link is point to other nodes in a tree.

Data

LC

RC

Here *LC* Points To **Left Child** and *RC* Points To **Right Child**.

**3. Parent Node:** The Immediate Predecessor of a Node is called as Parent Node.

X

Y

Z

Here X is Parent Node to Node Y and Z.

**4. Child Node:** The Immediate Successor of a Node is called as Child Node.

In the above diagram Node Y and Z are child nodes to node X.

**5. Root Node:** Which is a specially designated node, and does not have any parent node.

A

B

C

D

E

F

I

G

H

J

0

1

2

3

Level

In the above diagram node A is a Root Node.

**6. Leaf node:** The node which does not have any child nodes is called leaf node.

* In the above diagram node H, I, E, J, G are Leaf nodes.

**7. Level:** It is the rank of the hierarchy and the Root node is present at level **0**. If a node is present at level ***l*** then its parent node will be at the level ***l-*1**and child nodes will present at level ***l*+1**.

**8. Height / Depth:** The number of nodes in the longest path from Root node to the Leaf node is called the height of a tree.

* Height of above tree is 4.
* Height of a tree can be easily obtained as *l*max + 1. Where *l*max is the maximum level of a tree.
* In the above example *l*max= 3. So height = 3 + 1 = 4

**9. Siblings:** The nodes which have same parent node are called as siblings.

* In the above example nodes B and C are siblings, nodes D and E are siblings , nodes F and G are siblings , nodes H and I are siblings.

**10. Degree / Arity**: Maximum number of child nodes possible for anode is called as degree of a node.

**TREE:**

Tree is a nonlinear data structure.

**Definition**

A tree T is a finite set of one or more nodes such that:

(i) There is a special node called as *root* node.

(ii) The remaining nodes are partitioned into ***n*** disjoint sets T1, T2, T3. . . Tn. where n>0.Where each disjoint set is a tree.

T1, T2, T3. . .Tn are called as *sub trees*.

A

B

C

D

E

F

I

G

H

J

T1

T2

T3

**A sample Tree**

**BINARY TREE**: Is a special form of a tree.

**Definition:**

A binary tree is T is a finite set of nodes such that,

(i) T is empty called as empty binary tree.

(ii) T has specially designed node called as Root Node and remaining node of binary tree are partitioned into 2 disjoint sets. One is Left sub tree and another one is Right sub tree.

A

B

C

D

E

F

I

G

H

J

**A sample Binary Tree**

**TWO SPECIAL CASES OF BINAERY TREE**

1. Full binary tree

2. Complete binary tree

**1. Full Binary Tree:** a binary tree is said to be full binary tree, if each level has maximum number of possible nodes.

**Eg:**

A

B

C

D

E

F

I

G

H

J

0

1

2

3

Level

K

O

N

M

L

Node’s

20= 1

21=2

22=4

23=8

**A Full Binary Tree of height 4**

**2. Complete Binary Tree:** A binary tree is said to be complete binary tree, if all levels except the last level has maximum number of possible nodes, last level nodes are appeared as far left as possible.

**Eg:**

A

B

C

D

E

F

I

G

H

J

0

1

2

3

Level

K

L

Node’s

20= 1

21=2

22=4

23=8

**A Complete Binary Tree of height 4**

**Properties of Binary Trees**

1. In any binary tree, maximum number of nodes on level *l* is 2*l* (where *l*>= 0).
2. Maximum number of nodes possible in a binary tree of height *h* is 2*h*-1.
3. Minimum number of nodes possible in a binary tree of height *h* is *h.*

.

A

B

C

Height = 3,

Nodes = 3

A

B

C

Height = 3,

Nodes = 3

Height = 4,

Nodes = 4

A

B

C

D

* Whenever every parent node has only one child, such kind of binary trees are called as ***Skew binary trees***.

1. For any non-empty binary tree, if ***n*** is the number of nodes and ***e*** is the number of edges, then ***n = e + 1***.

i.e. number of nodes = number of edges +1.

1. For any non-empty binary tree T, if ***n0*** is the number of leaf nodes (degree = 0) and **n2** is the number of intermediate nodes (degree = 2) then **n0 = n2 + 1**.

i.e. number of leaf nodes = number of non-leaf nodes + 1.

1. The height of a complete binary tree with **n** nodes is
2. Total number of binary trees possible with **n** number of nodes is
3. The maximum and minimum size that an array may require to store a binary tree with ***n*** number of nodes are:

Maximum size =**2n – 1**

Minimum size =

1. In a linked list representation of binary tree, if there are ***n*** number of nodes, then the number of *NULL* link are ***λ = n +1.***

**Representation of Binary Tree**

Binary tree can be represented in two ways.

1. Linear (or) Sequential representation using arrays.
2. Linked List representation using pointers.

**1. Linear (or) Sequential representation using arrays**

* In this representation, a block of memory for an array is to be allocated before going to store the actual tree in it.
* Once the memory is allocated, the size of the tree will be restricted to memory allocated.
* In this representation, the nodes are stored level by level starting from zero level, where only *ROOT* node is present.
* The *ROOT* node is stored in the first memory location. i.e. first element in the array.

The following rules are used to decide the location on any node of tree in the array. (Assume the array index start from 1)

1. The ROOT node is at index**1**.
2. For any node with index i, 1 ≤i ≤ n.

(a) **Parent (i)** =

For the node when i = 1, there is ***no parent node***.

(b) **LCHILD (i)** = 2 \* i

If 2 \* i > n then **i** has ***no left child***.

(c) **RCHILD (i)** = 2\*i +1

If 2\*i + 1 > n then **i** has ***no right child***.

**Eg**. (A-B) + C \* (D / E)

+

-

\*

A

B

C

/

E

D

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| + | - | \* | A | B | C | / |  |  |  |  |  |  | D | E |

**Array representation of above binary tree.**

**Advantages of Linear representation of Binary Tree**

1. Any node can be accessed from any other node by calculating the index and this is efficient from execution point of view.
2. Here only data is stored without any pointers to their successor (or) predecessor.

**Disadvantages of Linear representation of Binary Tree**

1. Other than full binary tree, majority of entries may be empty.
2. It allows only static memory allocation.

**2. Linked List representation of Binary Tree using pointers**

* Linked list representation of assumes structure of a node as shown in the following figure.

Data

Right Child

Left Child Link

* With linked list representation, if one knows the address of *ROOT* node, then any other node can be accessed.

**+**

-

**\***

A

B

C

**/**

E

D

**Binary Tree**

**+**

-

**\***

**X A X**

**X B X**

**X C X**

**/**

**X D X**

**X E X**

**Linked List representation of Binary Tree**

**Advantages of Linked List representation of Binary Tree**

1. It allows dynamic memory allocation.
2. We can overcome the drawbacks of linear representation.

**Disadvantages of Linked List representation of Binary Tree**

1. It requires more memory than linear representation. i.e. Linked list representation requires extra memory to maintain pointers.

**Tree Traversals**

Traversal operation is used to visit each node present in binary tree exactly once.

A Binary tree can be traversed in 3 ways.

1. Preorder traversal
2. Inorder traversal
3. Postorder traversal

**1. Preorder traversal**

Here first **ROOT** node is visited, then **LEFT** sub tree is visited in *Preorder* fashion and then **RIGHT** sub tree is visited in *Preorder* fashion.

i.e. *ROOT*, LEFT, RIGHT

**2. Inorder traversal**

Here first **LEFT** subtree is visited in *inorder* fashion, then **ROOT** node is visited and then **RIGHT** sub tree is visited in *inorder* fashion.

i.e. LEFT, *ROOT*, RIGHT

**3. Postorder traversal**

Here first **LEFT** subtree is visited in *postorder* fashion, then **RIGHT** sub tree is visited in *postorder* fashion and then **ROOT** node is visited.

i.e. LEFT, RIGHT, *ROOT*

**Eg:**

+

-

\*

A

B

C

/

E

D

Preorder traversal : + - A B \* C / D E

Inorder traversal : A – B + C \* D / E

Postorder traversal : A B – C D E / \* +

**Recursive implementation of Binary Tree Traversals:**

**Algorithm preorder(ptr)**

**Input:** Binary Tree with some nodes.

**Output:** *preorder* traversal of given Binary Tree.

1. if(ptr != NULL)

a) print(ptr.data)

b) preorder(ptr.lchild)

c) preorder(ptr.rchild)

2. end if

**End preorder**

**Algorithm inorder(ptr)**

**Input:** Binary Tree with some nodes.

**Output**: *inorder* traversal of given Binary Tree.

1. if(ptr != NULL)

a) inorder(ptr.lchild)

b) print(ptr.data)

c) inorder(ptr.rchild)

2. end if

**End inorder**

**Algorithm postorder(ptr)**

**Input:** Binary Tree with some nodes.

**Output:** *postorder* traversal of given Binary Tree.

1. if(ptr != NULL)

a) postorder(ptr.lchild)

b) postorder(ptr.rchild)

c) print(ptr.data)

2. end if

**End postorder**

**Non recursive implementation of Binary Tree Traversals**

To implement non recursive implementation of binary tree traversals, here we are using single array for Inorder and Preordere traversals and two stacks for Postorder traversals.

**Algorithm Inorder\_Stack(i)**

**Input:** *i* is root node **index** and a stack is used.

**Output:** Inorder traversal of binary tree.

1. while(stack is not empty || a[i] != NULL)

a) if(a[i] !=NULL)

i) push(i)

ii) I = 2\*i

b) else

i) i = pop()

ii) print(a[i])

iii) I = 2\*i+1

c) end if

2. end loop

**End Inorder\_Stack**

The following operations are performed to traverse a binary tree in in-order using a stack:

1. Start from the root, call it PTR.
2. Push PTR onto stack if PTR is not NULL.
3. Move to left of PTR and repeat step 2.
4. If PTR is NULL and stack is not empty, then Pop element from stack and set as PTR.
5. Process PTR and move to right of PTR , go to step 2.

**Algorithm Preorder\_Stack(i)**

**Input:** *i* is root node **index** and a stack used.

**Output:** Preorder traversal of binary tree

1. push(i)

2. while(stack is not empty)

a) i = pop()

b) if(a[i] != NULL)

i) print(a[i])

ii) push(2\*i+1)

iii) push(2\*i)

c) end if

3. end loop

**End Preorder\_Stack**

The following operations are performed to traverse a binary tree in pre-order using a stack:

1. Start with root node and push onto stack.
2. Repeat while the stack is not empty
   1. POP the top element (PTR) from the stack and process the node.
   2. PUSH the right child of PTR onto to stack.
   3. PUSH the left child of PTR onto to stack.

**Algorithm Postorder\_Stack(ROOT)**

**Input:** *ROOT* is root node and two stacks named as stack1 and stack2 are used.

**Output:** Postorder traversal of binary tree.

1. PUSH the ROOT node into stack1.
2. While the stack1 is not empty

a) POP a node from stack1 and PUSH it into the stack2.

b) PUSH the LEFT and RIGHT Childs of poped nodes into stack1.

1. End loop
2. Now print the content of stack2. i.e. POP every node from stack2 and print them.

**End Postorder\_Stack**

1. Start from the root, call it PTR.
2. Push PTR onto stack if PTR is not NULL.
3. Move to left of PTR and repeat step 2.
4. If PTR has a right child R, then PUSH -R onto the stack.
5. Pop and process positive element from stack and set as PTR.
6. If a negative node is popped, (PTR = -N), then set PTR = N and go to step 2.

**Formation of Binary Tree from its Tree traversals**

* A binary tree can be constructed from its traversals.
* If the *Preorder* traversals is given, then the ***first node*** is *ROOT* node and *Postorder* traversal is given then ***last node*** is the *ROOT* node.
* For construction of a binary tree from its traversals, two traversals are essentials. Out of which one should be *inorder* traversal and another one is either *preorder* (or) *postorder* traversal.
* If preorder and post order is given to construct a binary tree, then binary tree can’t be obtain *uniquely*.

**Eg.1.**

Inorder: D B H E A I F J C G

Preorder: A B D E H C F I J G

1. From the preorder traversal, *A* is the *ROOT* node.
2. In the inorder traversal, all the nodes which are LEFT side of **A** belongs to LEFT sub tree and those node which are RIGHT side of **A** belongs to RIGHT sub tree.
3. Now the problem is reduced to two sub trees and same procedure can be applied repeatedly.

A

Inorder : D B H E

Preorder: B D E H

Inorder : I F J C G

Preorder: C F I J G

B

Inorder : D Preorder: D

Inorder : H E Preorder: E H

Inorder : I F J

Preorder: F I J

Inorder : G Preorder: G

C

E

Inorder : H Preorder: H

C

Inorder : I Preorder: I

Inorder : J Preorder: J

Final Binary tree from the Inorder and Preorder as follows:

A

B

C

D

E

F

G

J

I

H

**Eg. 2.**

Inorder: B C A E D G H F I

Postorder: C B E H G I F D A

A

Inorder : B C

Postorder: C B

Inorder : E D G H F I

Postorder: E H G I F D

B

Inorder : C Postorder: C

Inorder : E

Postorder: E

Inorder : G H F I Postorder: H G I F

D

F

Inorder : G H Postorder: H G

Inorder : I Postorder: I

G

Inorder : H Postorder: H

Final Binary tree from the Inorder and Postorder as follows:

A

B

D

C

E

F

I

G

H

**Types of Binary Tress**

1. Expression Trees
2. Binary Search Trees
3. Threaded Binary Trees
4. Heap Tree
5. Height Balanced Binary Trees
6. Decision Trees
7. Huffman Tree

**Binary Search Tree**

**Definition:**

A binary tree T is termed binary search tree (or binary sorted tree) if each node N of T satisfies the following property:

The value at N is greater than every value in the left sub-tree of N and is less than every value in the right sub-tree of N.

**A BST with numeric data**

**Operations on Binary Search Trees:**

1. Searching data
2. Inserting data
3. Deleting Data
4. Traversing the Tree

**1. Searching for data a Binary Search Tree**

* Searching data in a binary search tree is much faster than searching data in arrays or linked lists.
* To find an element ITEM from the BST first it should be compared with the root node R, and then one of the following conditions may be true:

1. ITEM == R
2. ITEM < R
3. ITEM > R

* If ITEM == R, the required element is found at Root, and the search is successful.
* If ITEM < R means that the required element is less than Root , so ITEM may reside in the left sub tree of the root, therefore we continue the search in the left sub tree of the root.
* If ITEM > R means that the required element is greater than Root, so ITEM may reside in the right sub tree of the root, therefore we continue the search in the right sub tree of the root.

**Algorithm BST\_Search(item)**

**Input**: item is data part of new node to be search in BST.

**Output**: if found then pointer to node containing data part as item. Otherwise error message.

1. ptr = Root

2. flag = 0

3. while(ptr != NULL and flag == 0 )

a) if( item == ptr.data)

i) flag = 1

ii) print “item found at ptr”

b) else if( item <ptr.data)

i) ptr = ptr.LCHILD goto step 3

c) else if( item >ptr.data)

i) ptr = ptr.RCHILD goto step 3

d) end if

4. end loop

5. if(flag == 0)

a) print “item not found”

6. end if

**End BST\_Search**

**2. Inserting data into a binary search tree**

To insert a node with data say item into a binary search tree, first binary search tree is searched starting from ROOT node for the item. If the item is found, do nothing. Otherwise item is to be inserted as LEAF node where search is halt.

**Inserting 80 into the above figure**

**After insertion of new node 80**

**Algorithm BST\_Insert(item)**

**Input:** *item* is data part of new node to be insert into BST.

**Output:** BST with new node has data part *item*.

1. ptr = Root

2. flag = 0

3. while(ptr != NULL && flag == 0 )

a) if( item == ptr.data)

i) flag = 1

ii) print “item already exist”

b) else if( item < ptr.data)

i) ptr1 = ptr

ii) ptr = ptr.LCHILD

c) else if( item > ptr.data)

i) ptr1 = ptr

ii) ptr = ptr.RCHILD

d) end if

4. end loop

5. if(ptr == NULL)

a) new = getnewnode()

b) new.data = item

c) new.lchild = NULL

d) new.rchild = NULL

e) if(root.data == NULL)

i) root = new

A) print “New node inserted successfully as ROOT Node”

f) else if( item < ptr1.data) /\* inserting new node as left child to its parent\*/

i) ptr1.lchild = new

ii) print “New Node is inserted successfully as LEFT child”

g) else /\* inserting new node as right child to its parent\*/

i) ptr1.rchild = new;

ii) print “New Node is inserted successfully as Right Child”

h) end if

6. end if

**End BST\_Insert**

**3. Deleting data from a Binary Search Tree**

* If ITEM is the information given which is to be deleted from a BST. Let N be the node which contains the information ITEM. Assume PARENT(N) denotes the parent node of N and SUCC(N) denotes the inorder successor of N.
* Then the deletion of the node N depends on the number of its children. Hence, 3 cases may arise and they are:

Case 1: N is the leaf node. i.e. no child nodes.

Case 2: N has exactly one child.

Case 3: N has two children.

**Case 2**

**Case 3**

**Case 1**

**Three cases from deleting a node from BST**

In the above Binary Search Tree (BST) deletion of a node *29* leads to *Case 1*. Here node *29* is a leaf node i.e. which does not have any child nodes. Deletion of node *94* leads to *Case 2*. Here node *94* has only one child node. Deletion of a node *19* leads to *Case 3*. Here node 19 has two child nodes.

**Case 1:**

N is a leaf node, this node is to be delete. N is deleted from T by simply setting the pointer of N in the parent node PARENT(N) by **NULL** value.

**Deletion of node 29**

**After deletion of node 29 form given BST**

**Case 2: N** has exactly one child.

N is deleted from T by simply replacing the pointer of N in PARENT(N) by the pointer of the only child of N.

**Deletion of node 94**

**After deletion of node 94 form given BST**

**Case 3: N** has two child nodes.

N is deleted from T by first deleting SUCC(N) from T(by using Case 1 or Case 2 it can be verified that SUCC(N) never has a left child) and then replacing the data content in node N by the data content in node SUCC(N).

SUCC(N)

N

**Deletion of node 19**

**After deletion of node 19 form given BST**

**Algorithm BST\_Delete(item)**

**Input**: *item* is the data part of node to be delete.

**Output**: BST without the node as data part *item*.

1. flag = 0

2. ptr = root;

3. while((ptr != NULL) && (flag == 0))

a) if(item == ptr.data)

i) flag=1

ii) break

b) else if(item < ptr.data)

i) parent = ptr

ii) ptr = ptr.lchild

c) else if(item > ptr.data)

i) parent = ptr

ii) ptr = ptr.child

d) end if

4. end loop

5. if(flag == 0)

a) printf “The given item is not exist in BST, So Deletion is not possible”

6. end if

7. if((ptr.lchild == NULL) && (ptr.rchild == NULL))

a) ch=1

8. else if((ptr.lchild != NULL) && (ptr.rchild != NULL))

a) ch=3

9. else

a) ch=2

10. end if

/\* DELETION FROM CASE 1 \*/

11. if(ch==1)

a) if(root.data == item) /\* BST has only one node \*/

i) root = NULL

b) else if(parent.lchild == ptr)

i) parent.lchild =NULL

c) else

i) parent.rchild = NULL

c) end if

d) print “Node is deleted successfully”

e) free(ptr)

12. end if

/\* DELETION FROM CASE 2 \*/

13. if(ch == 2)

a) if(root.data == item) /\* Node to be deleted as ROOT node \*/

i) if(root.rchild == NULL)

A) root = root.lchild

ii) else if(root.lchild == NULL)

A) root = root.rchild

iii) end if

b) else if(parent.lchild == ptr) /\* Node to be deleted is left child of parent node \*/

i) if(ptr. Lchild == NULL)

A) parent.lchild = ptr.rchild

ii) else

A) parent.lchild = ptr.lchild

iii) end if

b) else /\* Node to be deleted is right child of parent node \*/

i) if(ptr.rchild == NULL)

A) parent.rchild = ptr.lchild

ii) else

A) parent.rchild = ptr.rchild

iii) end if

c) end if

d) print “Node is deleted successfully”

e) free(ptr)

14. end if

/\* DELETION FROM CASE 3 \*/

15. if(ch == 3)

a) ptr4 = ptr /\* to store the address of the node to be delete \*/

b) ptr1= succ(ptr)

c) item1 = ptr1.data

d) deletion(item1)

e) ptr4.data = item1

16. end if

**End BST\_Delete**

**Algorithm SUCC(ptr2)**

**Input:** Pointer to a node *ptr2* who’s in order successor is to be found.

**Output:** Pointer to the inorder successor of *ptr2*.

1. ptr3 = ptr2.RCHILD

2. if(ptr3 != NULL)

a) While(ptr3.LCHILD != NULL)

i) ptr3 = ptr3.LCHILD

b) end loop

3. end if

4. return(ptr3)

**End SUCC**

**Threaded Binary Trees**

* In a linked representation of binary trees with n (n>0) nodes, n+1 link fields contain **NULL** entries.
* From this fact: in a binary tree more than 50% of link fields are with **NULL** values, thereby wasting memory space.
* A clever idea to utilize these **NULL** link fields is to store pointers of some nodes in these link fields; these extra pointers are called **threads** (in contrast to links) and the tree is specially termed **Threaded Binary Tree.**
* Thus in a Threaded Binary Tree each node contains links to its child (or) threaded to some other node in the tree.

**Representation of Threaded Binary Trees**

There are 3 ways to represent a Threaded Binary Tree. They are:

1. Inorder Threading of a Binary Tree.
2. Preorder Threading of a Binary Tree.
3. Postorder Threading of a Binary Tree.

**Example:**

A

B

**C**

**D**

**F**

**G**

**E**

**Advantages of Threaded Binary Tree:**

* Traversal is faster in Threaded Binary Trees.
* Predecessor and Successor can be determined in Threaded Binary Trees.
* Any node can be accessed from any other node.
* Though insertion and deletion are time consuming still these are easy to implement.

**Various operations on Threaded Binary Tree:**

1. To find the inorder successor of any node.
2. To find the inorder predecessor of any node.
3. To find the inorder traversal of any node.

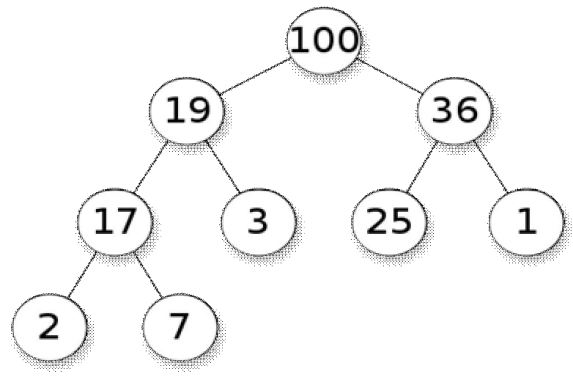
**Binary Heap/Heap Tree**

* ***Binary Heap/Heap Tree:*** A Heap is a Complete Binary tree with elements from partially ordered set which satisfies Heap Ordering property. The ordering can be of 2 types:
  1. ***Min Heap:*** For each node N in a complete binary tree, the value of N is less than or equal to the value of its children’s, such a heap tree is called a Min Heap.

**

*Fig: Min heap*

* 1. ***Max Heap:*** For each node N in a complete binary tree, the value of N is greater than or equal to the value of its children’s, such a heap tree is called a Max Heap.

**

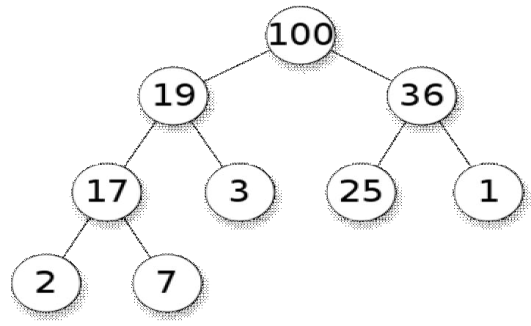
*Fig: Max Heap*

***Properties of Binary Heap:***

* + 1. *Structure Property:* A Heap is a binary tree that is also a Complete Binary tree.
    2. *Heap Ordering property:*
       - Min Heap property
       - Max Heap property

**Array Representation of Heap Tree/Binary Heap:**

* A heap tree can be represented using array and linked list.
* For an element in array position i, the left child is in the position 2i, the right child is at position (2i+1) i.e in the cell after the left child.
* The parent is in position floor(i/2).



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **100** | **19** | **36** | **17** | **3** | **25** | **1** | **2** | **7** | **-** | **-** | **-** | **-** | **-** | **-** |

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Heap ADT:

1. Inserting a new element
2. Deleting an element

**INSERTION OPERATION ON BINARY HEAP:**

* To insert an element say x, into the heap with n elements, we first create a hole in position (n+1) and see if the heap property is violated by putting x into the hole.
* If the heap property is not violated, then we have found the correct position for x. Otherwise, we “Reheap –up”' or “percolate-up”' x until the heap property is restored.
* *Percolate-up / Reheap-up:* we slide the element that is in the hole's parent node into the hole, thus bubbling the hole up toward the root. We continue this process until x can be placed in the whole.

***Algorithm:***

**Algorithm Insert\_Maxheap( A[1…N], N, X )**

{

N=N+1; A[N]=X;

Reheap\_up(N); /\* rebuild heap tree if the heap ordering property is violated \*/

}

Algorithm Reheap\_up(node M)

{

while(M>1)

{

parent=M/2;

if(A[parent] < A[M]) /\* if(A[parent] > A[M]) in case of Min-heap \*/

{

temp=A[M];

A[M]=A[parent];

A[parent]=temp;

M=parent;

}

}

Example:

* In the below example, 4 is inserted at the last position, which satisfies complete binary tree property.
* Now assume newly inserted node as current node and x is the parent of the current node.
* Now compare current node value with its parent(x). If current node is less than parent(x) node value, then interchange the current node value and x value.
* Continue the re-heap up to the current node is less than the child node value.v

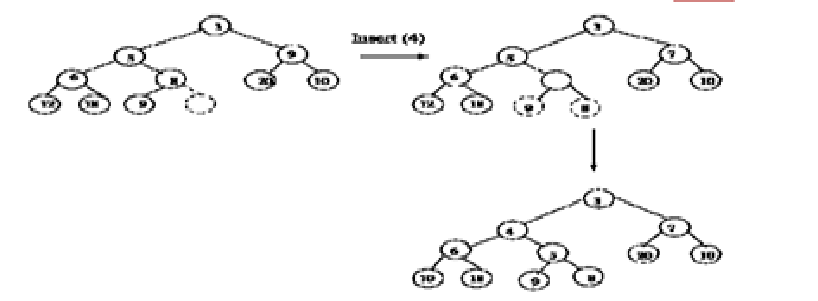


Fig: Insertion of node **4** into Heap tree

* Worst-case complexity of insert is O (h) where h is the height of the heap.
* Thus insertions are O (log n) where n is the number of elements in the heap.

**DELETION OPERATION ON BINARY HEAP:**

* When the minimum is deleted, a hole is created at the root level. Since the heap now has one less element and the heap is a complete binary tree, the element in the least position is to be relocated.
* First place the last element in the hole created at the root.
* This will leave the heap property possibly violated at the root level. We now “Reheap-down” or ``percolate-down'' the hole at the root until the violation of heap property is stopped.
* *Percolate-down/Reheap-down:* we slide the smaller of the hole’s children into the hole, thus pushing the hole down one level.

***Algorithm:***

**Algorithm DeleteMin( A[1…N], N, X )**

**{**

**i**f ( N == 0)

print(“Heap tree is empty, deletion not possible”)

else

{

A[1]=A[N];

A[N]=0;

N--;

Reheap\_down(1);**/\* rebuild heap tree if the heap ordering property is violated \*/**



}

}

Algorithm Reheap\_down(node M)

{

while(2\*M <= N)

{

left=2\*M; right=2\*M+1;

if(right <=N and A[right] < A[left]) target=right;

else

target=left;

if(A[target] < A[M]) /\* if (A[target] > A[M]) in case of Max-heap \*/

{

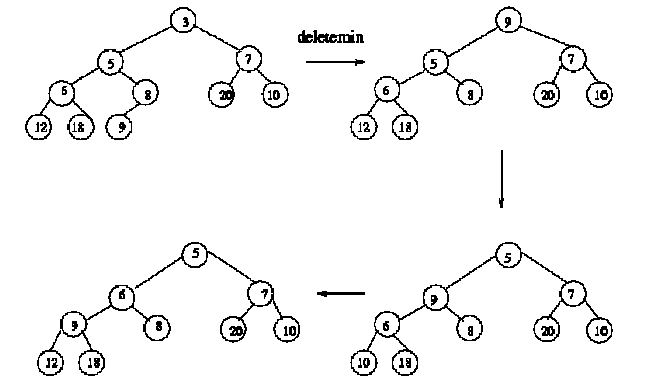
temp=A[M]; A[M]=A[target]; A[target]=temp; M=target;

}

}

}

***Example:***

******

* The worst-case time complexity of deletemin/deletemax is O (log n) where n is the number of elements in the heap