UNIT 2

Advanced Data Structure

CONTENTS

PART-1

Red-Black Trees.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 2.1. **Define a red-black tree with its properties. Explain the insertion operation in a red-black tree.**

Answer

Red-black tree :

A red-black tree is a binary tree where each node has colour as an extra attribute, either red or black. It is a self-balancing Binary Search Tree (BST) where every node follows following properties :

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendent leave contain the same number of black nodes.

Insertion :

i. We begin by adding the node as we do in a simple binary search tree and colouring it red.

RB-INSERT(*T*, *z*)

```
y ← nil [T]
x ← root [T]
while x ≠ nil [T]
    do y ← x
    if key[z] < key [x]
    then x ← left [x]
    else x ← right [x]
p[z] ← y
if y = nil [T]
    then root [T] ← z
    else if key [z] < key[y]
    then left [y]← z
    else right [y] ← z
left [z] ← nil[T]
right [z] ← nil[T]
colour [z] ← RED
RB-INSERT-FIXUP(T, z)
```

ii. Now, for any colour violation, RB-INSERT-FIXUP procedure is used.

RB-INSERT-FIXUP(*T*, z)

```
while colour [p[z]] = RED
    do if p[z] = left[p [p [z]]]
       then y ← right[p [p [z]]]
          if colour[y] = RED
             then colour[p [z]] ← BLACK                       ⇒ case 1
             colour[y] ← BLACK                                ⇒ case 1
             colour[p [p [z]]]← RED                           ⇒ case 1
             z ← p [p [ z]]                                   ⇒ case 1
          else if z = right[p[z]]
             then z ← p[z]                                    ⇒ case 2
                LEFT-ROTATE(T, z)                             ⇒ case 2
             colour[p[z]] ← BLACK                             ⇒ case 3
             colour[p[p[z]]] ← RED                            ⇒ case 3
             RIGHT-ROTATE(T, p[p[z]])                         ⇒ case 3
          else (same as then clause with "right" and "left" exchanged)
colour[root[T]] ← BLACK
```

Cases of RB-tree for insertion :

Case 1 : *z*'s uncle is red :

$$P[z] = \text{left}[p[p[z]]]$$

then uncle ← right[p[p[z]]]

a. change z's grandparent to red.

b. change z's uncle and parent to black.

c. change z to z's grandparent.

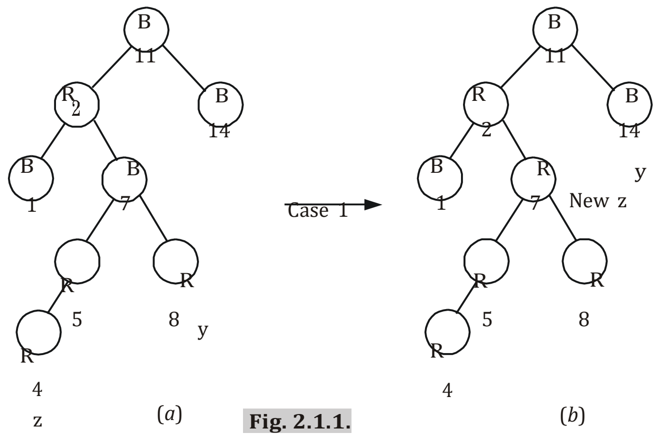
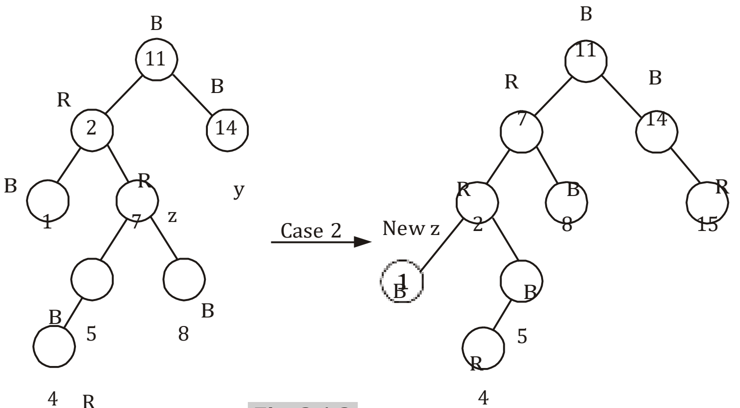


Fig. 2.1.1.

Now, in this case violation of property 4 occurs, because z's uncle y is red, then case 1 is applied.

Case 2 : *z*'s uncle is black, *z* is the right of its parent :

a. Change z to z's parent.
b. Rotate z's parent left to make case 3.



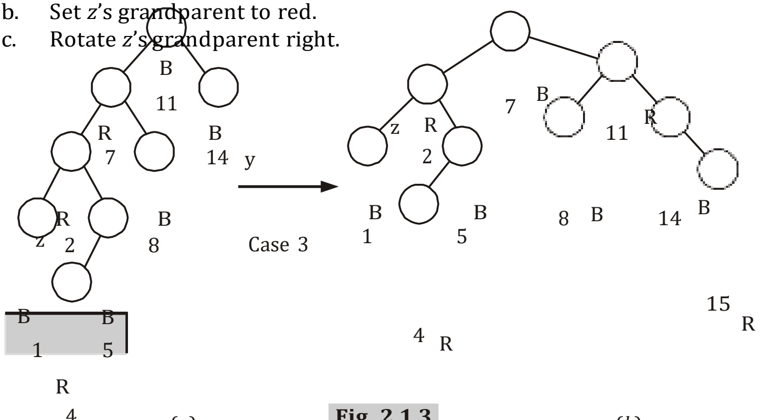
(*a*) **Fig. 2.1.2.** (*b*)

Case 3 : *z*'s uncle is black, *z* is the left child of its parent :

a. Set *z*'s parent black.

b. Set *z*'s grandparent to red.

c. Rotate *z*'s grandparent right.



(*a*) **Fig. 2.1.3.** (*b*)

Que 2.2. **What are the advantages of red-black tree over binary search tree ? Write algorithms to insert a key in a red-black tree insert the following sequence of information in an empty red-black tree 1, 2, 3, 4, 5, 5.**

Answer

Advantages of RB-tree over binary search tree :

1. The main advantage of red-black trees over AVL trees is that a single top-down pass may be used in both insertion and deletion operations.
2. Red-black trees are self-balancing while on the other hand, simple binary

search trees are unbalanced.

3. It is particularly useful when inserts and/or deletes are relatively frequent.
4. Time complexity of red-black tree is $O(\log n)$ while on the other hand, a simple BST has time complexity of $O(n)$.

Algorithm to insert a key in a red-black tree : Refer Q. 2.1, Page 2–2B, Unit-2.

Numerical :

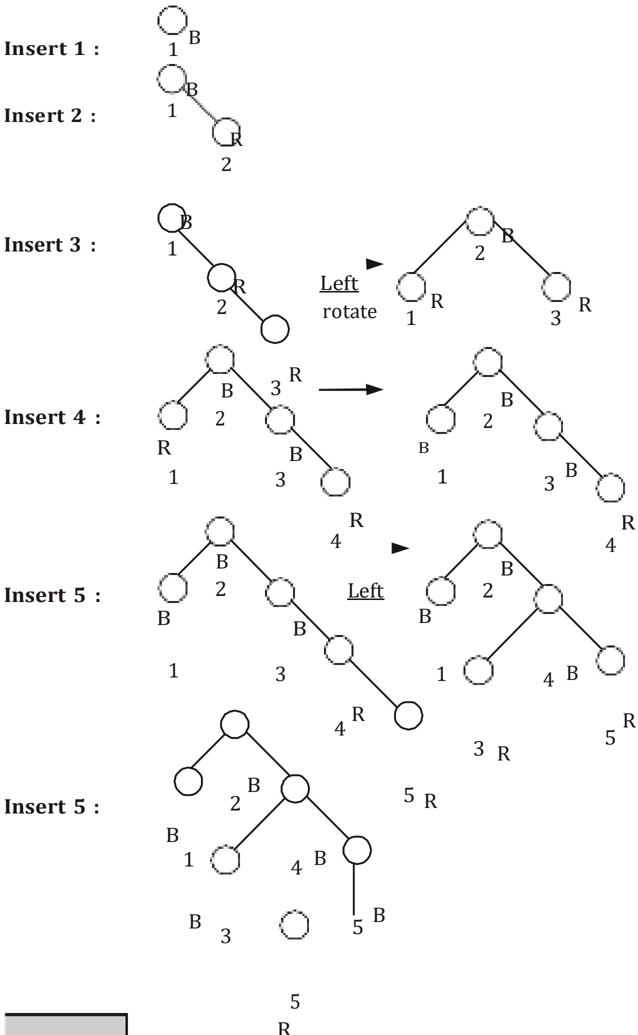


Fig. 2.2.1.

Que 2.3. **Explain red-black tree. Show steps of inserting the keys 41, 38, 31, 12, 19, 8 into initially empty red-black tree.**

OR

What is red-black tree ? Write an algorithm to insert a node in an

AKTU 2017-18, Marks 10

empty red-black tree explain with suitable example.

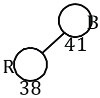
Answer

Red-black tree and insertion algorithm : Refer Q. 2.1, Page 2–2B, Unit-2.

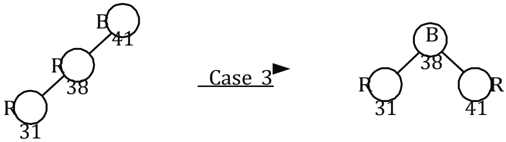
Numerical :

Insert 41 :

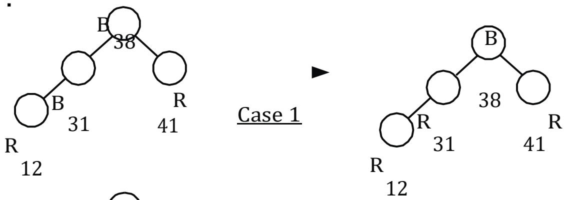
Insert 38 :



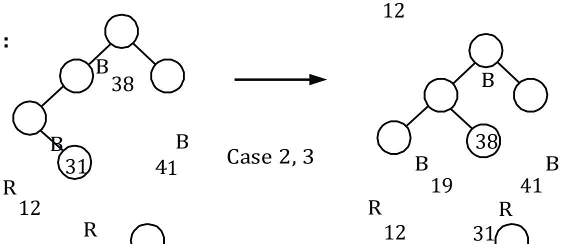
Insert 31 :



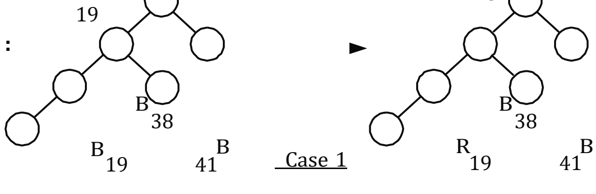
Insert 12 :



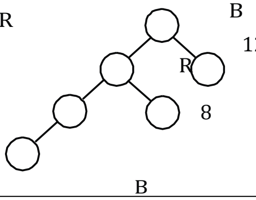
Insert 19 :



Insert 8 :



Thus final tree is



38

R B

19 41

B

12 B

31

R

8

Que 2.4. **Explain insertion in red-black tree. Show steps for inserting 1, 2, 3, 4, 5, 6, 7, 8 and 9 into empty RB-tree.**

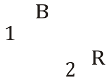
AKTU 2015-16, Marks 10

Answer

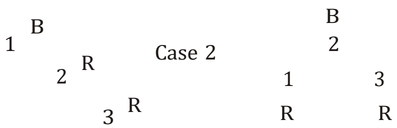
Insertion in red-black tree : Refer Q. 2.1, Page 2–2B, Unit-2.

Insert 1 :

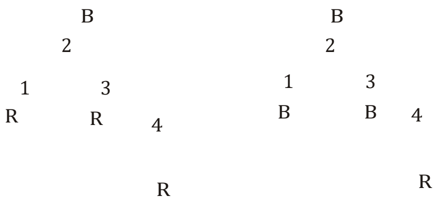
Insert 2 :



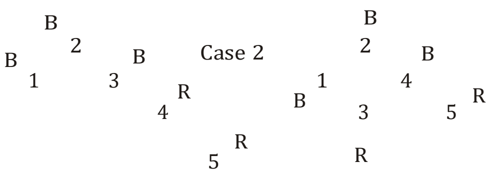
Insert 3 :



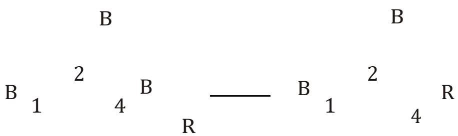
Insert 4 :



Insert 5 :



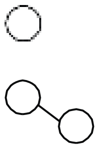
Insert 6 :

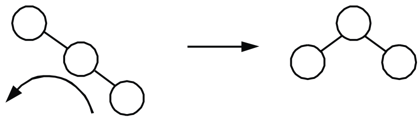


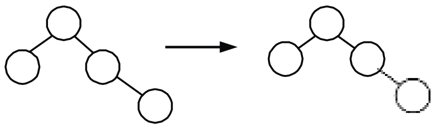
3 5 R 6

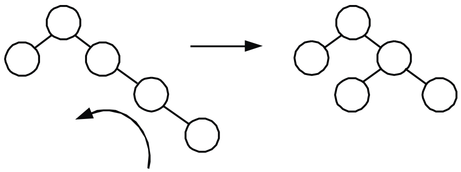
R

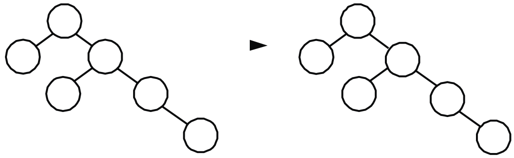
3 5 B B 6 R



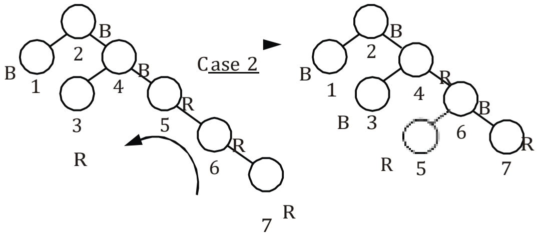




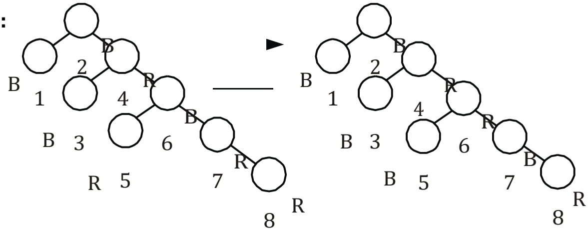




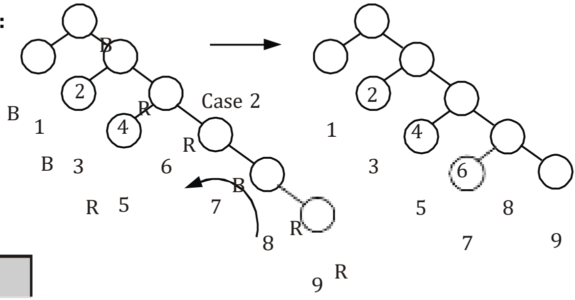
Insert 7 :



Insert 8 :



Insert 9 :



Que 2.5. **How to remove a node from RB-tree ? Discuss all cases and write down the algorithm.**

Answer

To remove a node from RB-tree RB-DELETE procedure is used. In RB-DELETE procedure, after splitting out a node, it calls an auxiliary procedure RB-DELETE-FIXUP that changes colours and performs rotations to restore the red-black properties.

RB-DELETE(T, z)

```
if left[z] = nil[T] or right[z] = nil[T]
    then y ← z
    else y ← TREE-SUCCESSOR(z)
if left[y] ≠ nil[T]
    then x ← left[y]
    else x ← right[y]
```

$p[x] \leftarrow p[y]$
if $p[y]$ = nil$[T]$

```
        then root[T] ← x
        else if y = left[p[y]]
          then left[p[y]] ← x
          else right[p[y]] ← x
    if y ≠ z
        then key[z] ← key[y]
          copy y's sibling data into z
    if colour[y] = BLACK
        then RB-DELETE-FIXUP(T, x)
    return y
```

RB-DELETE-FIXUP(*T*, *x*)

```
    while x ≠ root[T] and colour[x] = BLACK
        do if x = left[p[x]]
          then w ← right[p[x]]
            if colour[w] = RED

              then colour[w] ← BLACK                              ⇒ case 1
                colour[p[x]] ← RED                                ⇒ case 1
                LEFT-ROTATE(T, p[x])                              ⇒ case 1
                w ← right[p[x]]                                   ⇒ case 1
            if colour[left[w]] = BLACK and colour[right[w]] = BLACK
              then colour[w] ← RED                                ⇒ case 2
                x ← p[x]                                          ⇒ case 2
              else if colour[right[w]] = BLACK
                then colour[left[w]] ← BLACK                      ⇒ case 3
                  colour[w] ← RED                                 ⇒ case 3
                  RIGHT-ROTATE(T, w)                              ⇒ case 3
                  w ← right[p[x]]                                 ⇒ case 3
                colour[w] ← colour[p[x]]                          ⇒ case 4
                colour[p[x]] ← BLACK                              ⇒ case 4
                colour[right[w]] ← BLACK                          ⇒ case 4
                LEFT-ROTATE(T, p[x])                              ⇒ case 4
                x ← root[T]                                       ⇒ case 4
            else (same as then clause with "right" and "left" exchanged).
    colour[x] ← BLACK
```

Cases of RB-tree for deletion :

Case 1 : *x*'s sibling *w* is red :

1. It occurs when node w the sibling of node x, is red.
2. Since w must have black children, we can switch the colours of w and $p[x]$ and then perform a left-rotation on $p[x]$ without violating any of the red-black properties.
3. The new sibling of x, which is one of w's children prior to the rotation, is now black, thus we have converted case 1 into case 2, 3 or 4.
4. Case 2, 3 and 4 occur when node w is black. They are distinguished by colours of w's children.

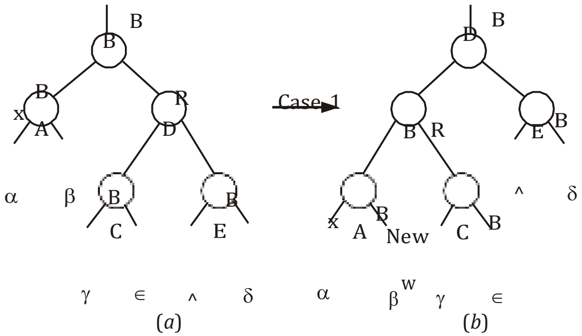


Fig. 2.5.1.

Case 2 : *x*'s sibling *w* is black, and both of *w*'s children are black :

1. Both of *w*'s children are black. Since *w* is also black, we take one black of both *x* and *w*, leaving *x* with only one black and leaving *w* red.
2. For removing one black from *x* and *w*, we add an extra black to *p*[*x*], which was originally either red or black.
3. We do so by repeating the while loop with *p*[*x*] as the new node *x*.
4. If we enter in case 2 through case 1, the new node *x* is red and black, the original *p*[*x*] was red.
5. The value *c* of the colour attribute of the new node *x* is red, and the loop terminates when it tests the loop condition. The new node *x* is then coloured black.

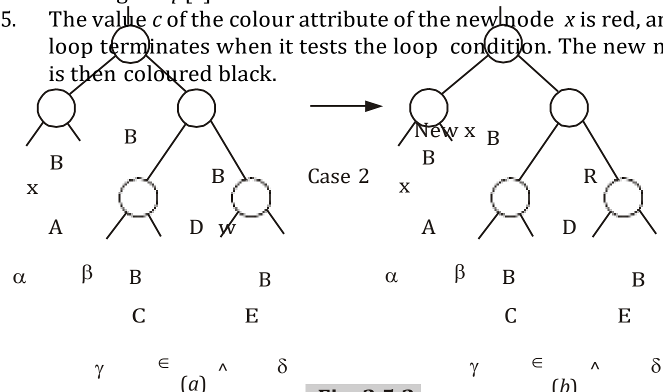


Fig. 2.5.2.

Case 3 : *x*'s sibling *w* is black, *w*'s left child is red, and *w*'s right child is black :

1. Case 3 occurs when *w* is black, its left child is red and its right child is black.
2. We can switch the colours of *w* and its left child left[*w*] and then

perform a right rotation on w without violating any of the red-black properties, the new sibling w of x is a black node with a red right child and thus we have transformed case 3 into case 4.

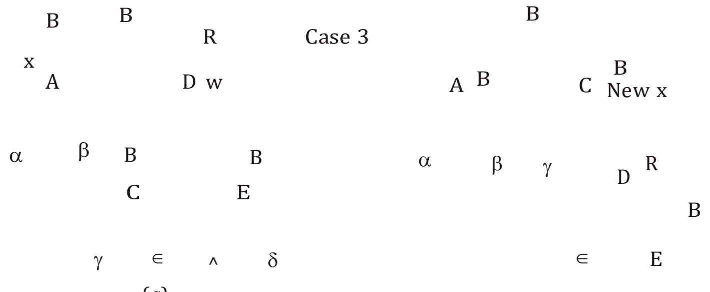


Fig. 2.5.3.

Case 4 : *x*'s sibling *w* is black, and *w*'s right child is red :

1. When node *x*'s sibling *w* is black and *w*'s right child is red.
2. By making some colour changes and performing a left rotation on *p*[*x*], we can remove the extra black on *x*, making it singly black, without violating any of the red-black properties.

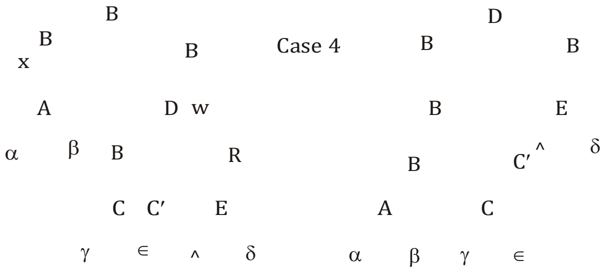


Fig. 2.5.4.

Que 2.6. **Insert the nodes 15, 13, 12, 16, 19, 23, 5, 8 in empty red-black tree and delete in the reverse order of insertion.**

AKTU 2016-17, Marks 10

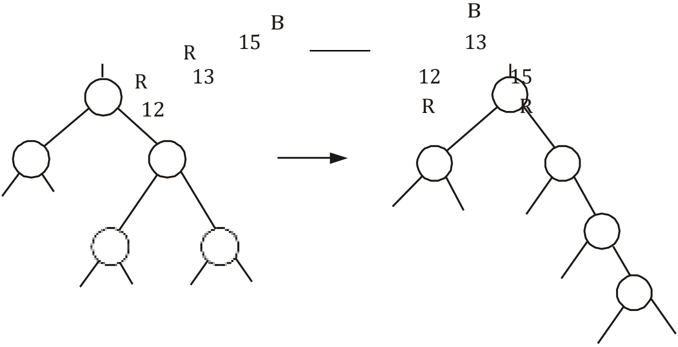
Answer

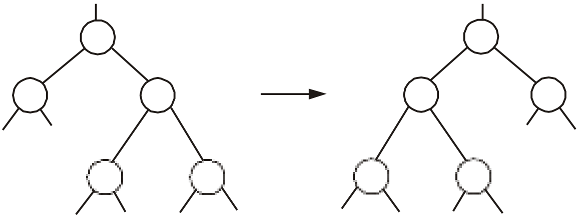
Insertion :

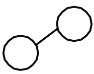
Insert 15 : 15 B

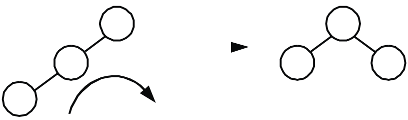
Insert 13 : 15 B, 13 R

Insert 12 :

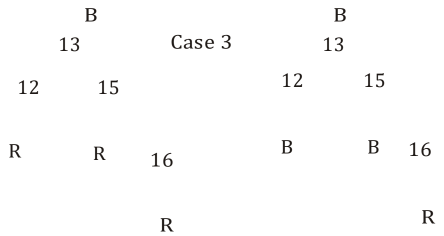

B
15
R
13
R
12
B
13
12
R
15
R



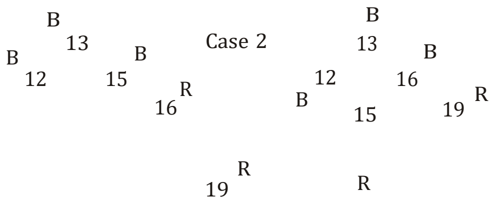




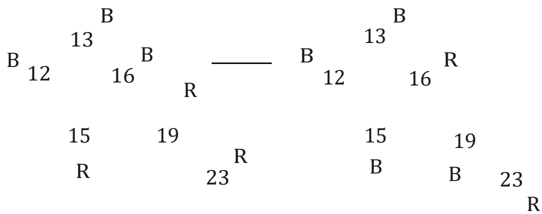
Insert 16 :



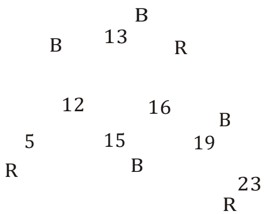
Insert 19 :



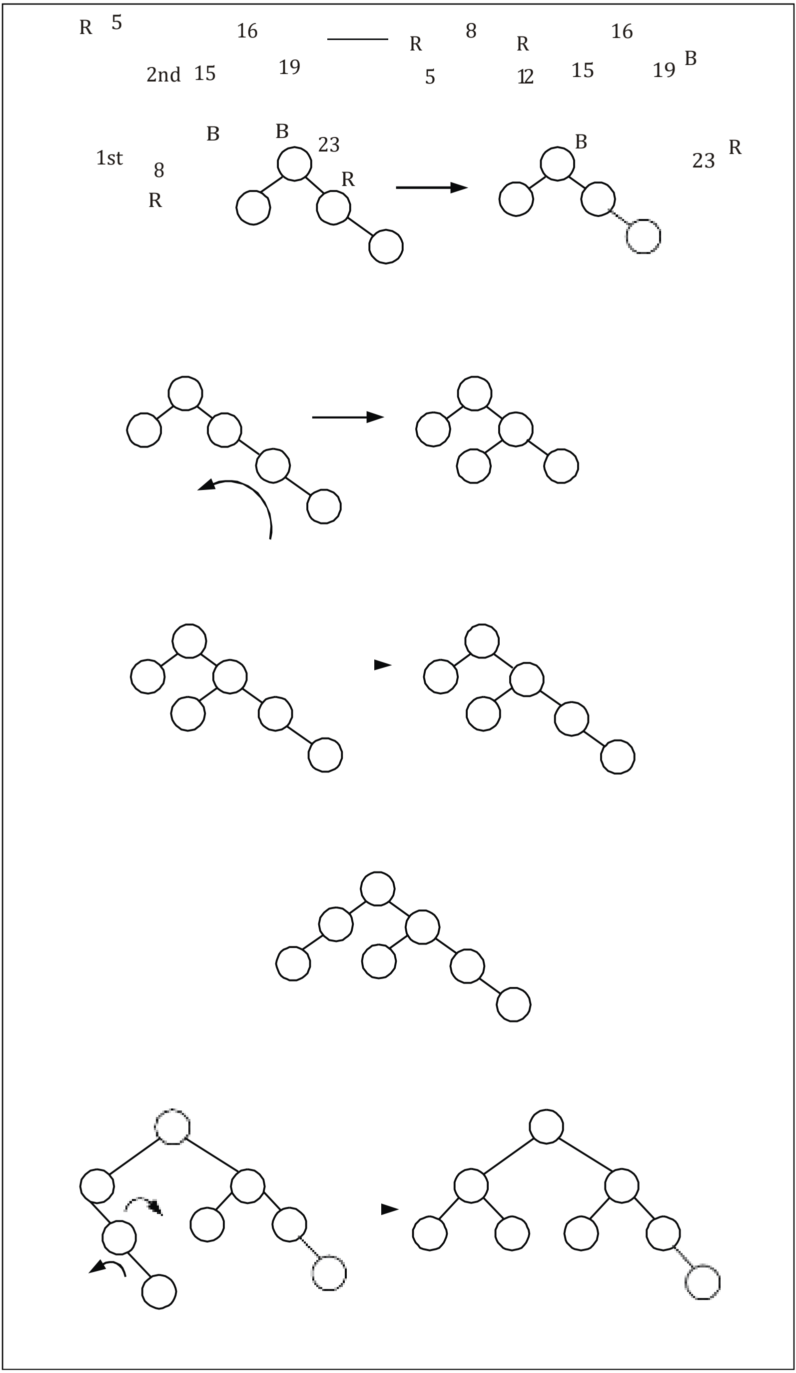
Insert 23 :



Insert 5 :



Insert 8 :


R
5
16
8
R
16
R
2nd
15
19
5
12
15
19
B
B
B
23
1st
8
R
R
B
23
R

Deletions :

Delete 8 :

B
13 B
B R 13
8 16 ——— B R
R 12 16

5 12 15 19 B 5 R B 15 19 B

R B
R 23 23 R

Delete 5 :

B
B
13 13

R R B R

12 16 ——— 12 16

R B B B
5 15 19 15 19
B

23 R 23 R

Delete 23 :

13 B 13 B

B 12 16 R —— B 12 16 R

B 15 19 B B 15 19 B

23 R

Delete 19 :

13 B B
13 B 13
R Case 2 B
B 12 16 —— B 12 16 12 16 B

R
B 15 19 B B 15 15 R

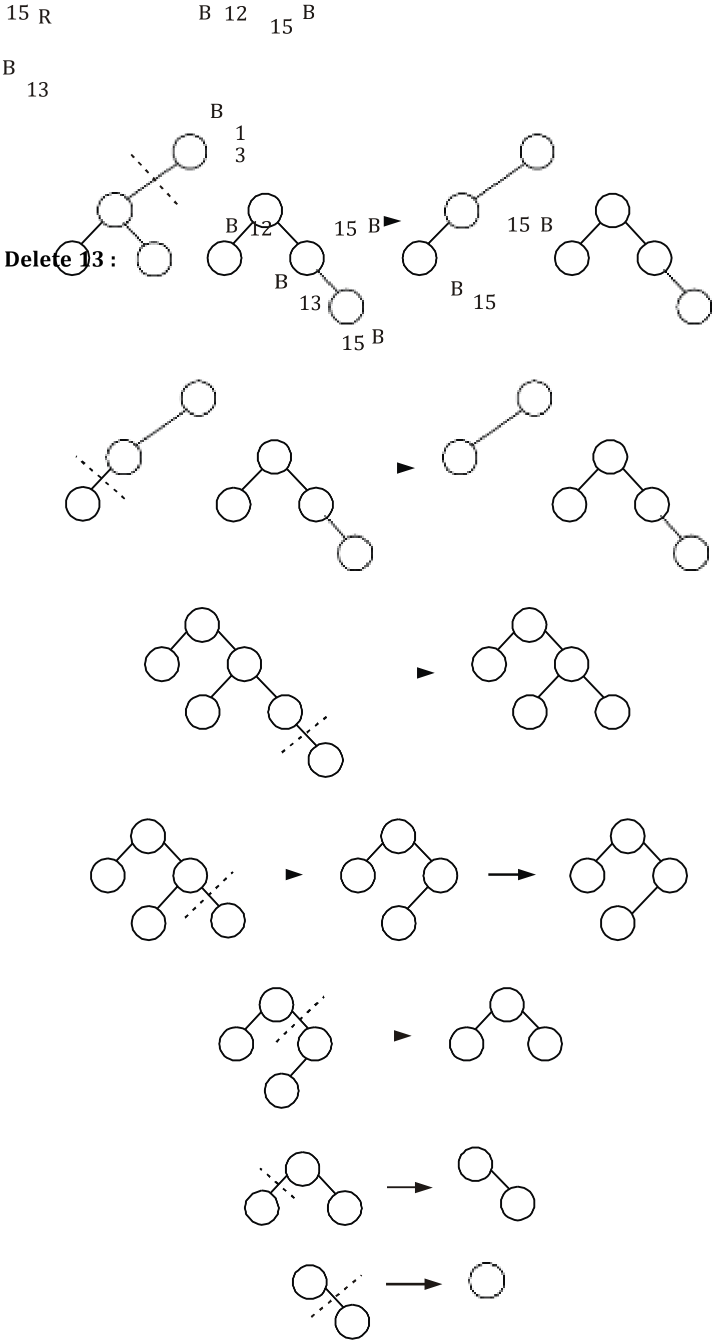
Delete 16 :

B B
13 13

Delete 12 :

B 12 16 B ——

Delete 13 :



Delete 15 :

No tree

Que 2.7. **Insert the following element in an initially empty RB-Tree.**

12, 9, 81, 76, 23, 43, 65, 88, 76, 32, 54. Now delete 23 and 81.

AKTU 2019-20, Marks 07

Answer

Insert 12 : 12 B

Insert 9 : 12 B

R
9

Insert 81 : 12 B

R 9 R 81

Insert 76 :

12 B
R R Case 1
9 81
R 76

12 B
B B
9 81
76 R

Insert 23 :

12 B
B B Case 3
9 81
R 76
R 23

12 B
B B
9 76
23 81
R R

Insert 43 :

12 B
B B Case 1
9 76
R R

12 B
B R
9 76
23 81

B

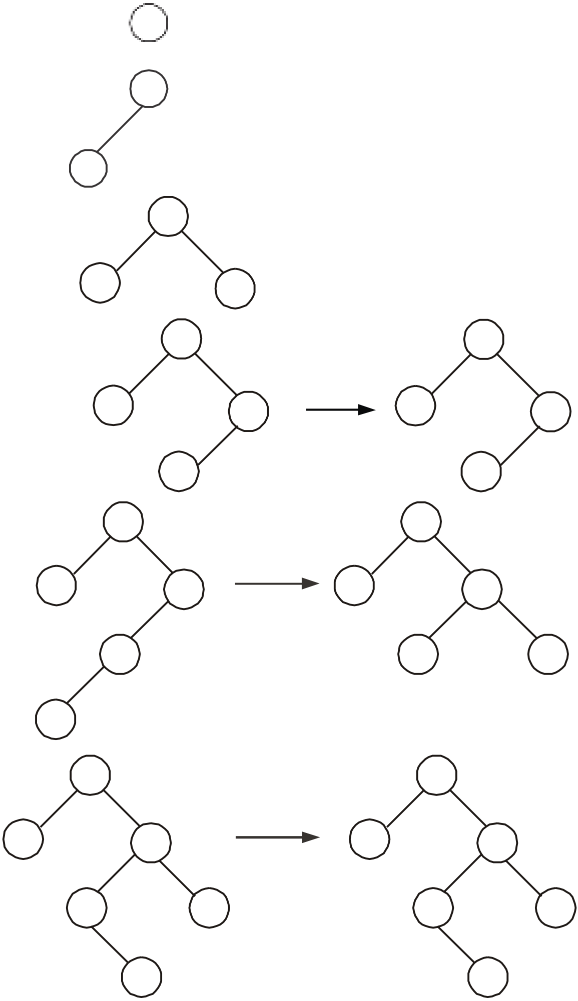
23

81

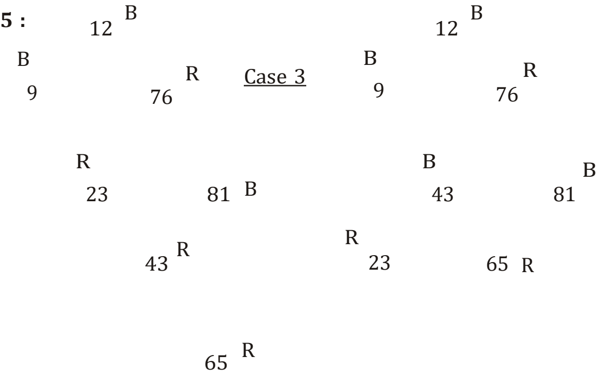
B

R 43

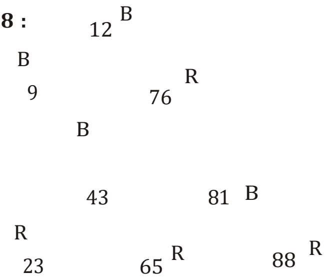
43 R



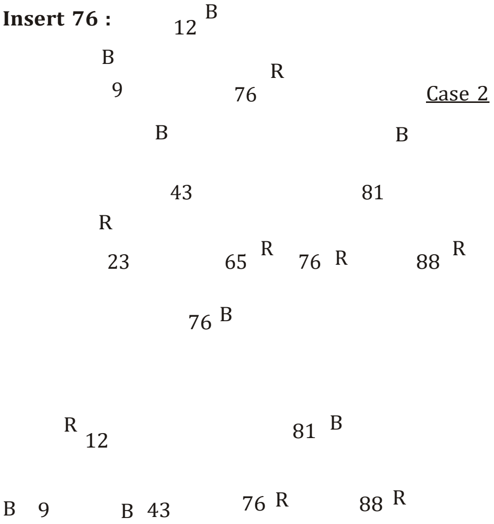
Insert 65 :



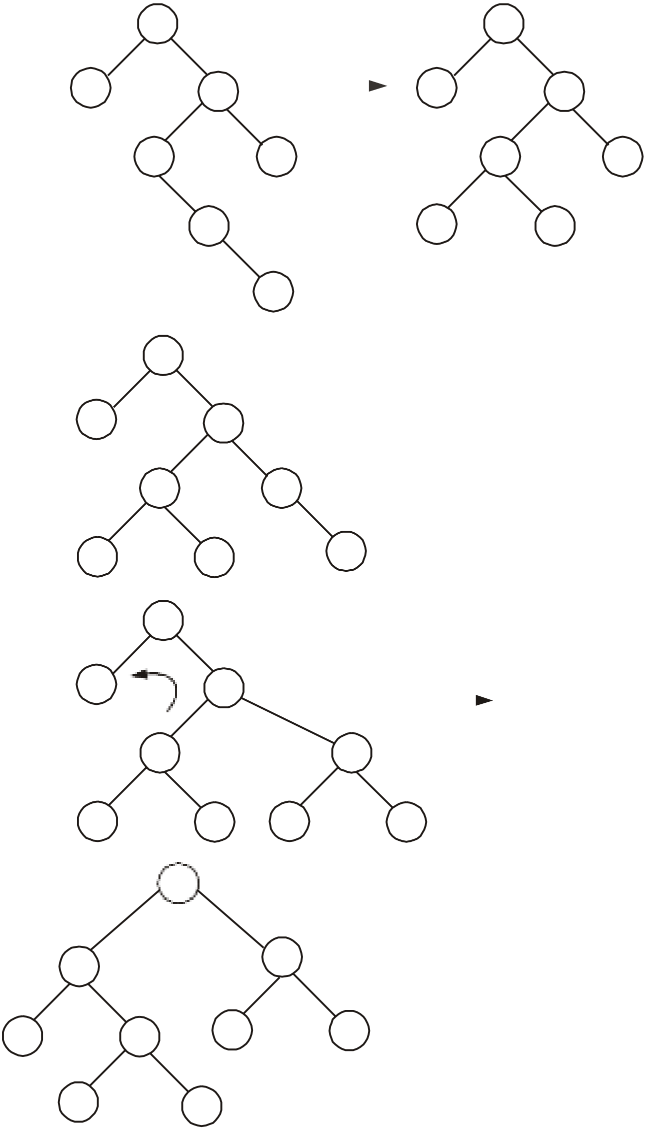
Insert 88 :



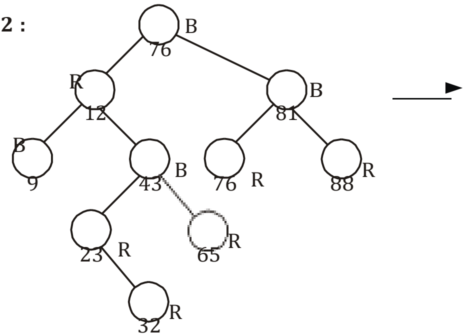
Insert 76 :

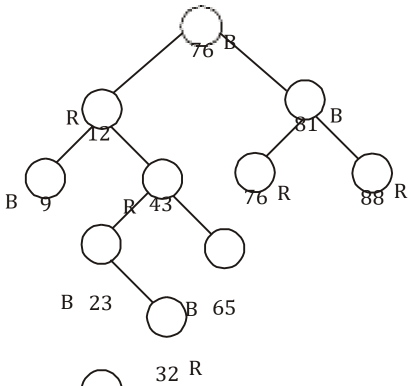


R 23 R 65

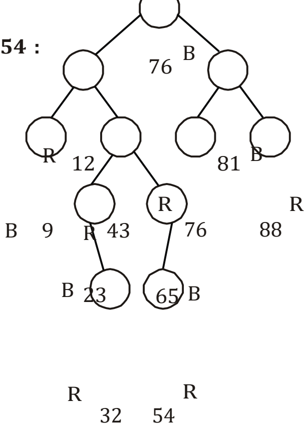


Insert 32 :

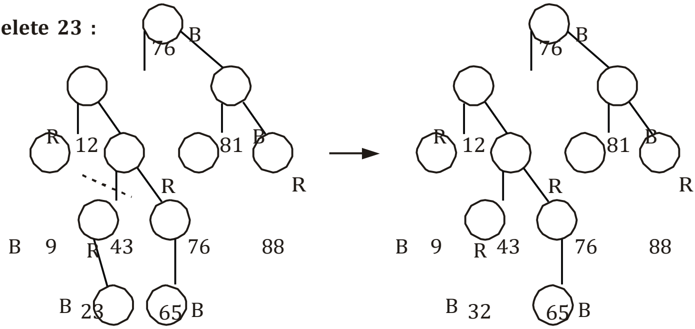




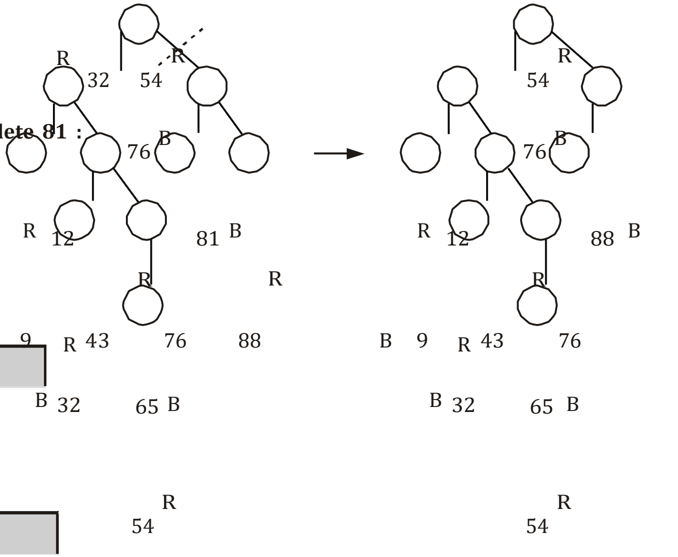
Insert 54 :



Delete 23 :



Delete 81 :



Que 2.8. Describe the properties of red-black tree. Show the red-black tree with *n* internal nodes has height at most 2 log (*n* + 1).

OR

Prove the height *h* of a red-black tree with *n* internal nodes is not greater than 2 log (*n* + 1).

Answer

Properties of red-black tree : Refer Q. 2.1, Page 2–2B, Unit-2.

1. By property 5 of RB-tree, every root-to-leaf path in the tree has the same number of black nodes, let this number be *B*.
2. So there are no leaves in this tree at depth less than *B*, which means the

tree has at least as many internal nodes as a complete binary tree of height B.

3. Therefore, $n \leq 2^B - 1$. This implies $B \leq \log(n + 1)$.
4. By property 4 of RB-tree, at most every other node on a root-to-leaf path is red, therefore, $h \leq 2B$.

   Putting these together, we have

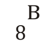
$$h \leq 2 \log(n + 1).$$

Que 2.9. **Insert the elements 8, 20, 11, 14, 9, 4, 12 in a Red-Black tree and delete 12, 4, 9, 14 respectively.**

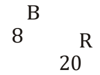
AKTU 2018-19, Marks 10

Answer

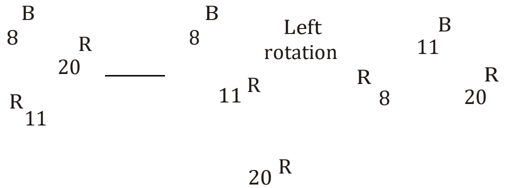
Insert 8 :



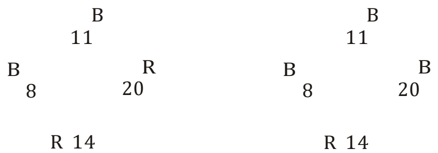
Insert 20 :



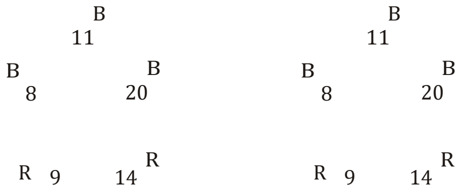
Insert 11 : Since, parent of node 11 is red. Check the colour of uncle of node 11. Since uncle of node 11 is nil than do rotation and recolouring.



Insert 14 : Uncle of node 14 is red. Recolour the parent of node 14 *i.e.*, 20 and uncle of node 14 *i.e.*, 8. No rotation is required.

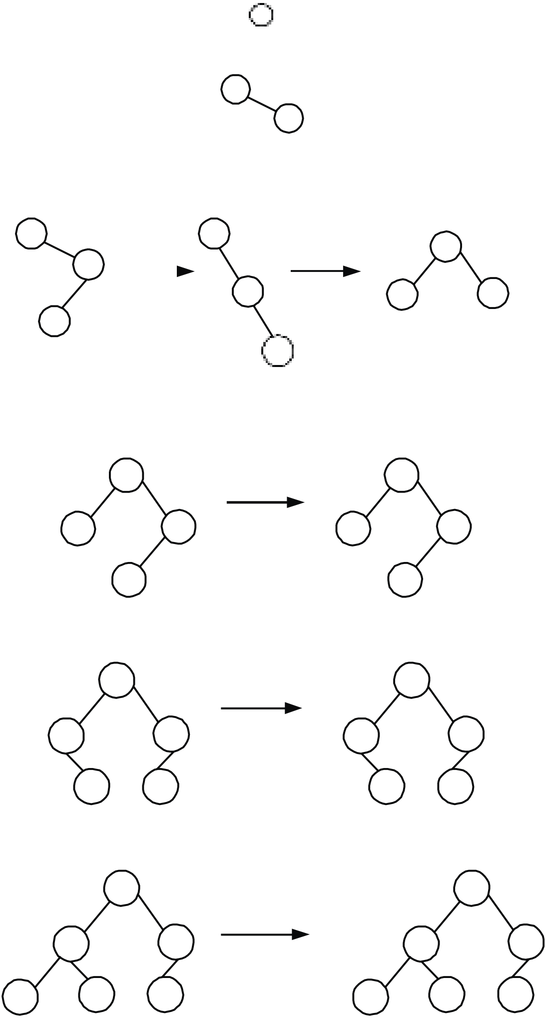


Insert 9 : Parent of node 9 is black. So no rotation and no recolouring.



Insert 4 : Parent of node 4 is black. So no rotation and no recolouring.


B
B
B
1
8
20
1
R
R
R 4
R 9
14
R 4
R 9
14



Insert 12 : Parent of node 12 is red. Check the colour of uncle of node 12, which is nil. So do rotation and recolouring.

B 11 B 11

B 8 B 20 Right rotation B 8 B 14

R 4 R 9 14 R R 4 R 9 12 R 20 R

12 R

Delete 12 : Node 12 is red and leaf node. So simply delete node 12.

B 11 B 11

B 8 B 14 B 8 B 14

R 4 R 9 12 R 20 R R 4 R 9 20 R

Delete 4 : Node 4 is red and leaf node. So simply delete node 4.

B 11 B 11

B 8 B 14 B 8 B 14

R 4 9 R 20 R R 9 20 R

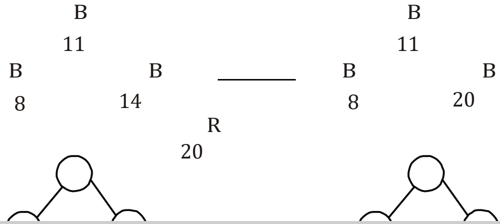
Delete 9 : Node 9 is red and leaf node. So simply delete node 9.

B 11 B 11

B 8 B 14 Case 1 B 8 B 14

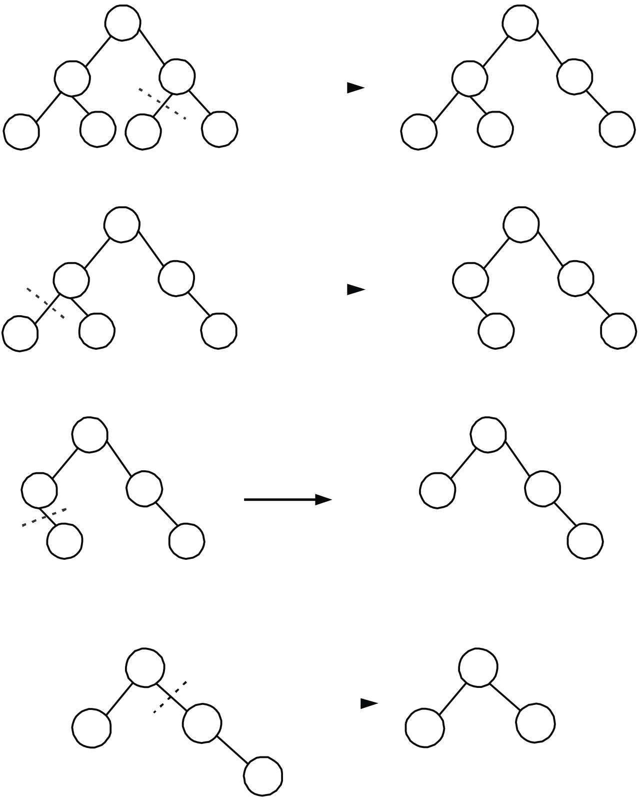
9 R 20 R 20 R

Delete 14 : Node 14 is internal node replace node 14 with node 20 and do not change the colour.



PART-2

B-Trees.



Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 2.10. Define a B-tree of order *m*. Explain the searching and insertion algorithm in a B-tree.

Answer

A B-tree of order m is an m-ary search tree with the following properties :

1. The root is either leaf or has atleast two children.
2. Each node, except for the root and the leaves, has between $m/2$ and m children.
3. Each path from the root to a leaf has the same length.
4. The root, each internal node and each leaf is typically a disk block.
5. Each internal node has upto $(m - 1)$ key values and upto m children.

SEARCH(x, k)

1. $i \leftarrow 1$
2. while $i \le n[x]$ and $k > \text{key}_i[x]$
3. do $i \leftarrow i + 1$
4. if $i \le n[x]$ and $k = \text{key}_i[x]$
5. then return(x, i)
6. if leaf[x]
7. then return NIL
8. else DISK-READ($c_i[x]$)
9. return B-TREE-SEARCH ($c_i[x]$, k)

B-TREE-INSERT(T, k)

1. $r \leftarrow \text{root}[T]$
2. if $n[r] = 2t - 1$
3. then $s \leftarrow$ ALLOCATE-NODE ( )
4. root[T] $\leftarrow S$
5. leaf[s] $\leftarrow$ FALSE
6. $n[s] \leftarrow 0$
7. $c_1[s] \leftarrow r$
8. B-TREE SPLIT CHILD(S, l, r)
9. B-TREE-INSERT-NONFULL(s, k)
10. else B-TREE-INSERT-NONFULL(r, k)

B-TREE SPLIT CHILD(x, i, y)

1. $z \leftarrow$ ALLOCATE-NODE ( )
2. leaf[z] $\leftarrow$ leaf[y]
3. $n[z] \leftarrow t - 1$

for $j \leftarrow 1$ to $t - 1$
do $key_j[z] \leftarrow key_{j+t}[y]$
if not leaf$[y]$
then for $j \leftarrow 1$ to t
do $c_j[z] \leftarrow c_{j+t}[y]$
$n[y] \leftarrow t - 1$
for $j \leftarrow n[x] + 1$ down to $i + 1$
do $c_{j+1}[x] \leftarrow c_j[x]$
$c_{i+1}[x] \leftarrow z$
for $j \leftarrow n[x]$ down to i
do $key_{j+1}[x] \leftarrow key_j[x]$
$key_i[x] \leftarrow key_t[y]$
$n[x] \leftarrow n[x] + 1$
DISK-WRITE$[y]$
DISK-WRITE$[z]$
DISK-WRITE$[x]$

The CPU time used by B-TREE SPLIT CHILD is $\theta(t)$. The procedure performs $\theta(1)$ disk operations.

B-TREE-INSERT-NONFULL(x, k)

$i \leftarrow n[x]$
if leaf$[x]$
then while $i \geq 1$ and $k < key_i[x]$
do $key_{i+1}[x] \leftarrow key_i[x]$
$i \leftarrow i - 1$
$key_{i+1}[x] \leftarrow k$
$n[x] \leftarrow n[x] + 1$
DISK-WRITE(x)
else while $i \geq 1$ and $k < key_i[x]$
do $i \leftarrow i - 1$
$i \leftarrow i + 1$
DISK-READ$(c_i[x])$
if $n[c_i[x]] = 2t - 1$
then B-TREE-SPLIT-CHILD$(x, i, c_i[x])$
if $k > key_i[x]$
then $i \leftarrow i + 1$
B-TREE INSERT NONFULL$(c_i[x], k)$

The total CPU time use is $O(th) = O(t \log_t n)$

Que 2.11. What are the characteristics of B-tree ? Write down the steps for insertion operation in B-tree.

Answer

Characteristic of B-tree :

1. Each node of the tree, except the root node and leaves has at least $m/2$ subtrees and no more than m subtrees.
2. Root of tree has at least two subtree unless it is a leaf node.
3. All leaves of the tree are at same level.

Insertion operation in B-tree :

In a B-tree, the new element must be added only at leaf node. The insertion operation is performed as follows :

Step 1 : Check whether tree is empty.

Step 2 : If tree is empty, then create a new node with new key value and insert into the tree as a root node.

Step 3 : If tree is not empty, then find a leaf node to which the new key value can be added using binary search tree logic.

Step 4 : If that leaf node has an empty position, then add the new key value to that leaf node by maintaining ascending order of key value within the node.

Step 5 : If that leaf node is already full, then split that leaf node by sending middle value to its parent node. Repeat the same until sending value is fixed into a node.

Step 6 : If the splitting is occurring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Que 2.12. **Describe a method to delete an item from B-tree.**

Answer

There are three possible cases for deletion in B-tree as follows :

Let k be the key to be deleted, x be the node containing the key.

Case 1 : If the key is already in a leaf node, and removing it does not cause that leaf node to have too few keys, then simply remove the key to be deleted. Key k is in node x and x is a leaf, simply delete k from x.

Case 2 : If key k is in node x and x is an internal node, there are three cases to consider :

a. If the child y that precedes k in node x has at least t keys (more than the minimum), then find the predecessor key k' in the subtree rooted at y. Recursively delete k' and replace k with k' in x.
b. Symmetrically, if the child z that follows k in node x has at least t keys, find the successor k' and delete and replace as before.
c. Otherwise, if both y and z have only $t - 1$ (minimum number) keys, merge k and all of z into y, so that both k and the pointer to z are removed from x, y now contains $2t - 1$ keys, and subsequently k is deleted.

Case 3 : If key k is not present in an internal node x, determine the root of the appropriate subtree that must contain k. If the root has only $t - 1$ keys, execute either of the following two cases to ensure that we descend to a node containing at least t keys. Finally, recurse to the appropriate child of x.

a. If the root has only $t - 1$ keys but has a sibling with t keys, give the root an extra key by moving a key from x to the root, moving a key from the roots immediate left or right sibling up into x, and moving the appropriate child from the sibling to x.

b. If the root and all of its siblings have $t - 1$ keys, merge the root with one sibling. This involves moving a key down from x into the new merged node to become the median key for that node.

Que 2.13. **How B-tree differs with other tree structures ?**

Answer

1. In B-tree, the maximum number of child nodes a non-terminal node can have is m where m is the order of the B-tree. On the other hand, other tree can have at most two subtrees or child nodes.
2. B-tree is used when data is stored in disk whereas other tree is used when data is stored in fast memory like RAM.
3. B-tree is employed in code indexing data structure in DBMS, while, other tree is employed in code optimization, Huffman coding, etc.
4. The maximum height of a B-tree is log mn (m is the order of tree and n is the number of nodes) and maximum height of other tree is $\log_2 n$ (base is 2 because it is for binary).
5. A binary tree is allowed to have zero nodes whereas any other tree must have atleast one node. Thus binary tree is really a different kind of object than any other tree.

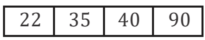
Que 2.14. **Insert the following key in a 2-3-4 B-tree : 40, 35, 22, 90, 12, 45, 58, 78, 67, 60 and then delete key 35 and 22 one after other.**

AKTU 2018-19, Marks 07

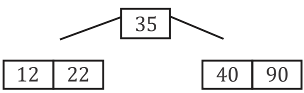
Answer

In 2-3-4 B-trees, non-leaf node can have minimum 2 keys and maximum 4 keys so the order of tree is 5.

Insert 40, 35, 22, 90 :



Insert 12 :

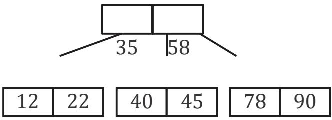


Insert 45, 58 :

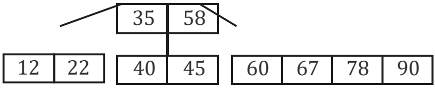
12	22

40	45	58	90

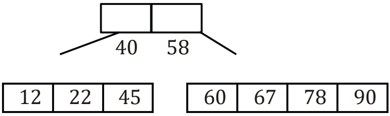
Insert 78 :



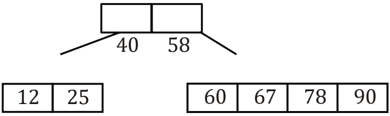
Insert 67, 60 :



Delete 35 :



Delete 22 :



Que 2.15. Explain B-tree and insert elements *B, Q, L, F* into B-tree Fig. 2.15.1 then apply deletion of elements *F, M, G, D, B* on resulting B-tree.

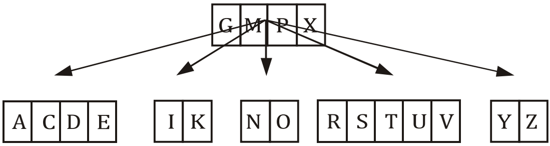


Fig. 2.15.1.

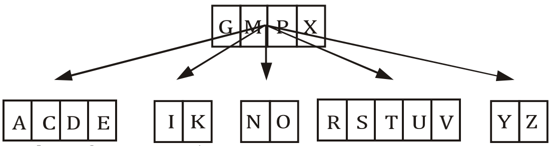
AKTU 2015-16, Marks 10

Answer

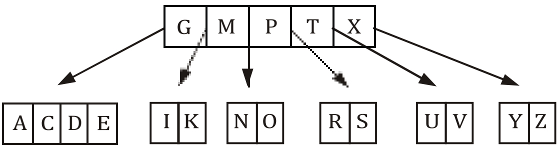
B-tree : Refer Q. 2.10, Page 2–20B, Unit-2.

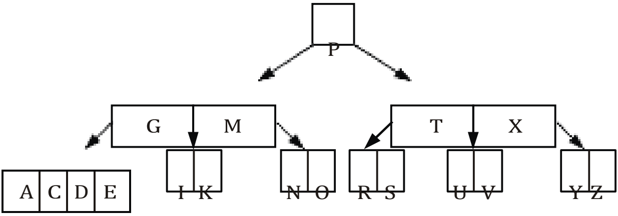
Numerical :

Insertion :

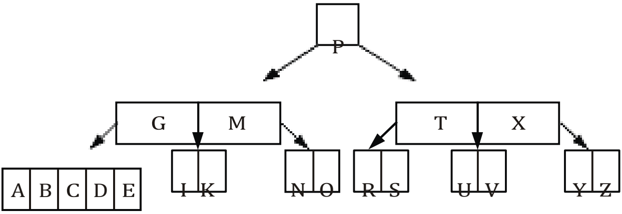


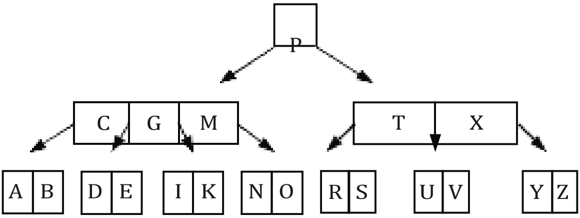
Assuming, order of B-tree = 5



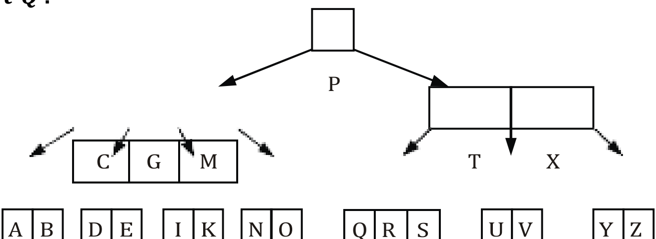


Insert *B* :

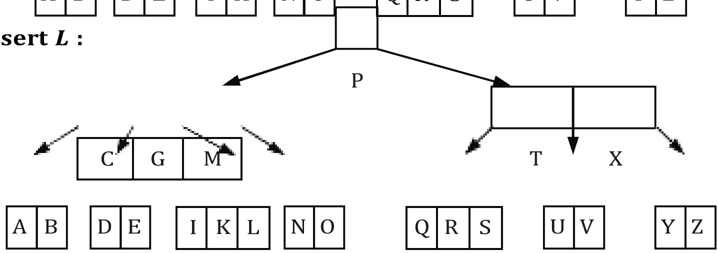




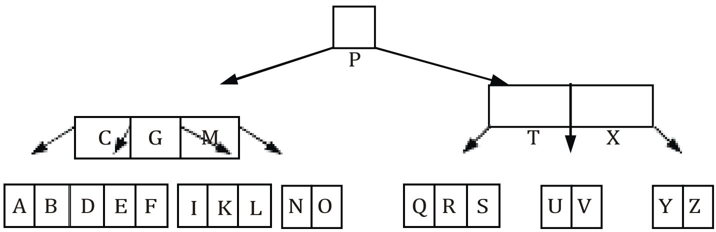
Insert *Q* :



Insert *L* :

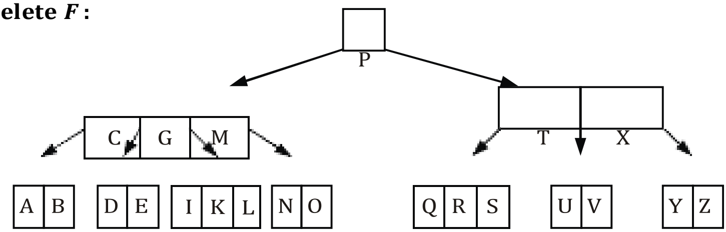


Insert *F* :

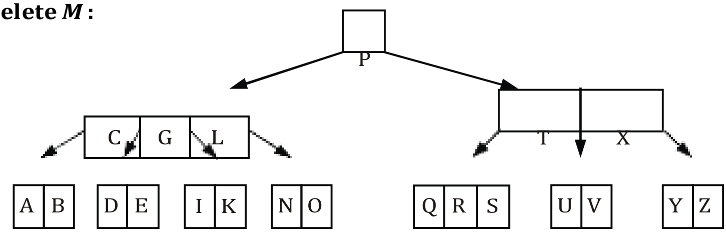


Deletion :

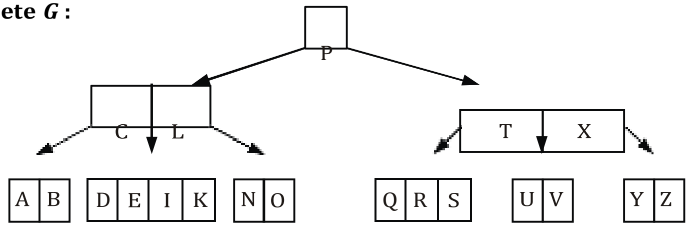
Delete *F* :



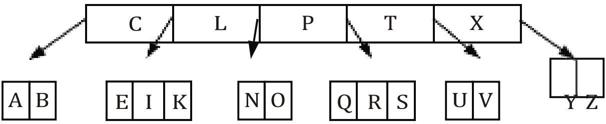
Delete *M* :



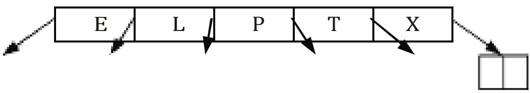
Delete *G* :



Delete *D* :



Delete *B* :



Que 2.16. **Insert the following information, *F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E, G, I* into an empty B-tree with degree *t* = 3.**

AKTU 2017-18, Marks 10

Answer

Assume that $t = 3$

$$2t - 1 = 2 \times 3 - 1 = 6 - 1 = 5$$

and $t - 1 = 3 - 1 = 2$

So, maximum of 5 keys and minimum of 2 keys can be inserted in a node.

Now, apply insertion process as :

Insert *F* : F

Insert *S* : | F | S |

Insert *Q* : | F | Q | S |

Insert *K* : | F | K | Q | S |

Insert *C* : | C | F | K | Q | S |

Insert *L* : | C | F | K | L | Q | S |

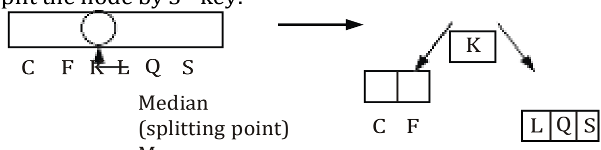
As, there are more than 5 keys in this node.

∴ Find median, $n[x] = 6$ (even)

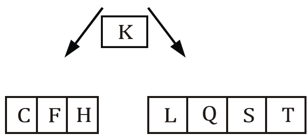
$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3$$

Now, median = 3,

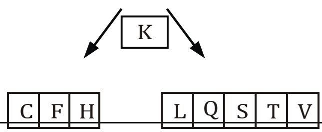
So, we split the node by 3rd key.



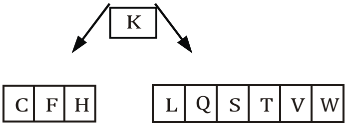
Insert *H, T* :



Insert *V* :



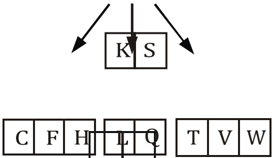
Insert *W* :



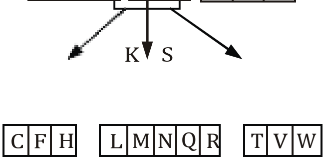
More than 5 keys split node from Median.

$n[x] = 6$ [even]

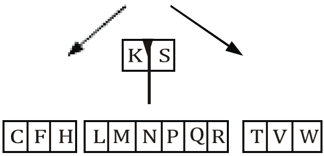
Median $= \frac{n[x]}{2} = \frac{6}{2} = 3$ (*i.e.*, 3rd key move up)



Insert *M, R, N* :



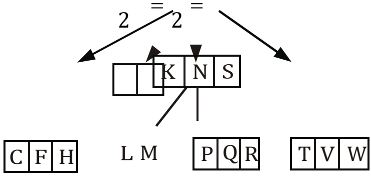
Insert *P* :



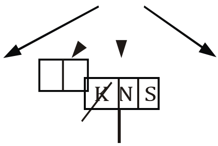
More than 5 key
split the node

$n[x] = 6$

Median $= \frac{n[x]}{2} = \frac{6}{2} =$ 3 (*i.e.*, 3rd key move up)



Insert *A, B* :



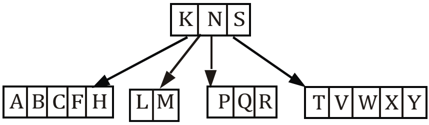
A	B	C	F	H

L M

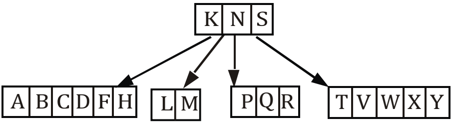
P	Q	R

T	V	W

Insert *X, Y* :



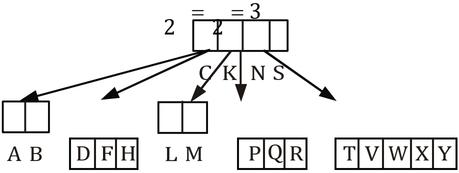
Insert *D* :



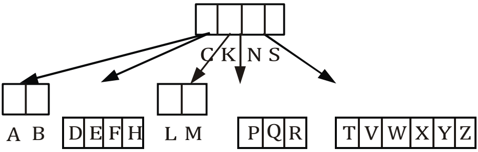
More than 5 key
split the node

$n[x] = 6$ (even)

$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3$$ (*i.e.*, 3rd key move up)



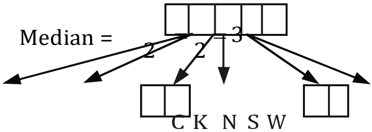
Insert *Z, E* :



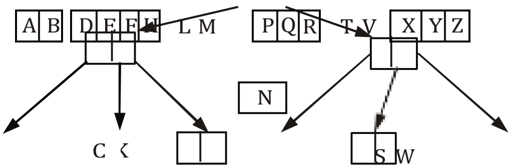
More than 5 key
split the node

$n[x] = 6$

$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3$$ (*i.e.*, 3rd key move up)



Insert *G, I* :



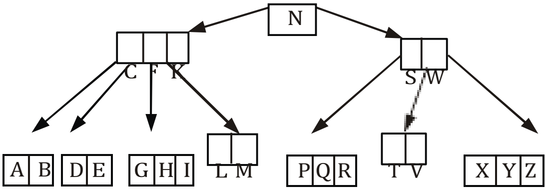


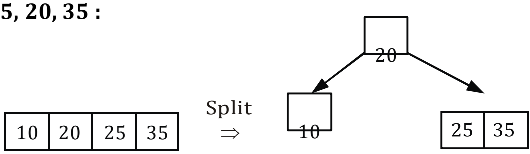
Fig. 2.16.1. Inserted all given information with degree t = 3.

Que 2.17. **Using minimum degree '*t*' as 3, insert following sequence of integers 10, 25, 20, 35, 30, 55, 40, 45, 50, 55, 60, 75, 70, 65, 80, 85 and 90 in an initially empty B-Tree. Give the number of nodes splitting operations that take place.**

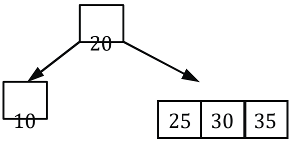
AKTU 2019-20, Marks 07

Answer

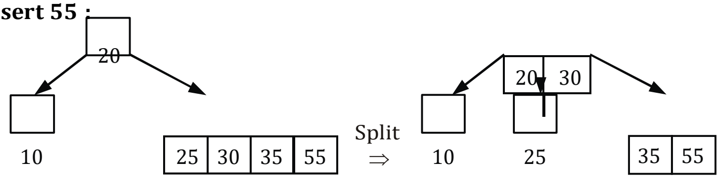
Insert 10, 25, 20, 35 :



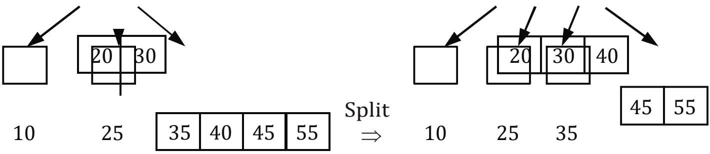
Insert 30 :



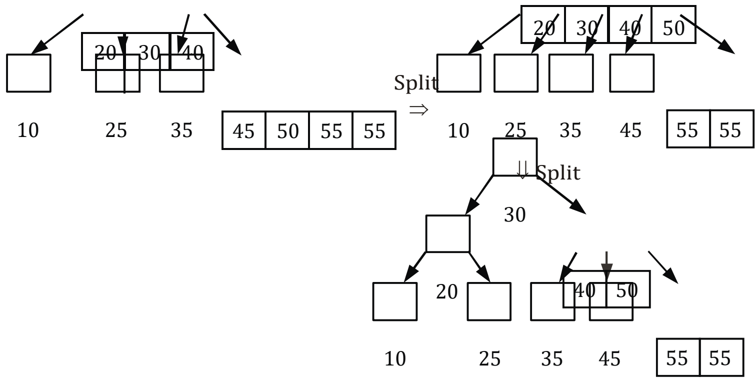
Insert 55 :



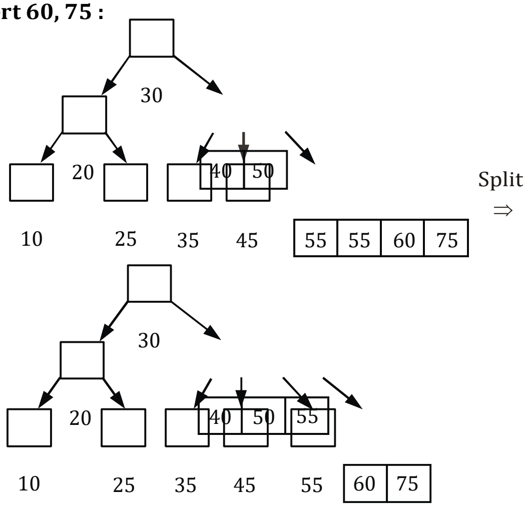
Insert 40, 45 :



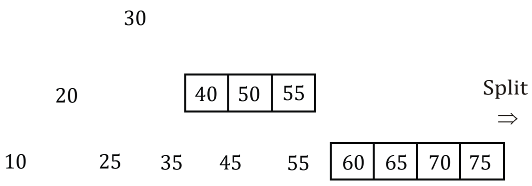
Insert 50, 55 :

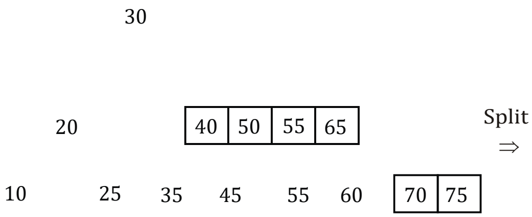


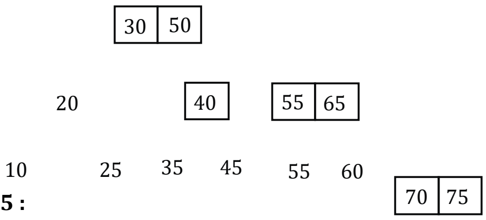
Insert 60, 75 :



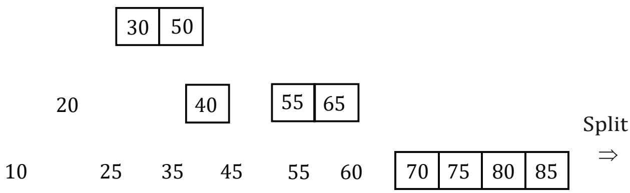
Insert 70, 65 :

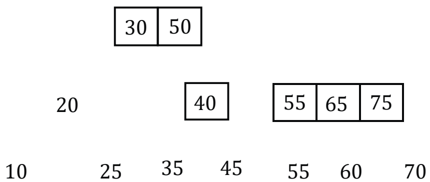




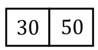


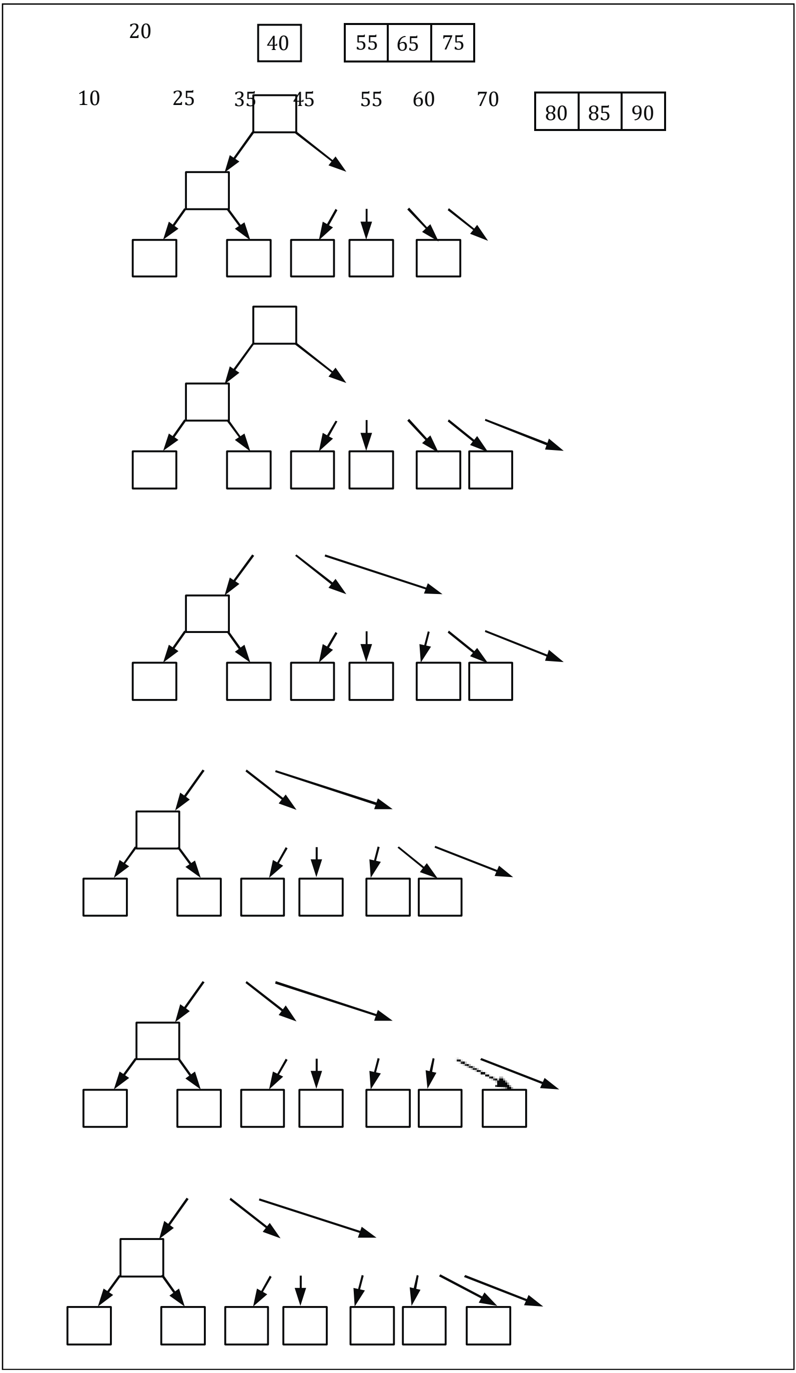
Insert 80, 85 :





Insert 90 :




20
40
55 65 75
10
25
35
45
55
60
70
80 85 90

Number of nodes splitting operations = 9.

PART-3

Binomial Heaps.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 2.18. Explain binomial heap and properties of binomial tree.

Answer

Binomial heap :

1. Binomial heap is a type of data structure which keeps data sorted and allows insertion and deletion in amortized time.
2. A binomial heap is implemented as a collection of binomial tree.

Properties of binomial tree :

1. The total number of nodes at order k are 2^k.
2. The height of the tree is k.
3. There are exactly $\binom{k}{i}$ *i.e.*, ${}^kC_i$ nodes at depth i for $i = 0, 1, \ldots, k$ (this is why the tree is called a "binomial" tree).
4. Root has degree k (children) and its children are B_{k-1}, B_{k-2}, ..., B_0 from left to right.

Que 2.19. What is a binomial heap ? Describe the union of binomial heap.

OR

Explain the different conditions of getting union of two existing binomial heaps. Also write algorithm for union of two binomial heaps. What is its complexity ?

AKTU 2018-19, Marks 10

Answer

Binomial heap : Refer Q. 2.18, Page 2–33B, Unit-2.

Union of binomial heap :

1. The BINOMIAL-HEAP-UNION procedure repeatedly links binomial trees where roots have the same degree.
2. The following procedure links the B_{k-1} tree rooted at node to the B_{k-1} tree rooted at node z, that is, it makes z the parent of y. Node z thus becomes the root of a B_k tree.

BINOMIAL-LINK (*y*, *z*)

```
p[y] ← z
sibling [y] ← child[z]
child[z] ← y
degree[z] ← degree[z] + 1
```

3. The BINOMIAL-HEAP-UNION procedure has two phases :
   a. The first phase, performed by the call of BINOMIAL-HEAP-MERGE, merges the root lists of binomial heaps H_1 and H_2 into a single linked list H that is sorted by degree into monotonically increasing order.
   b. The second phase links root of equal degree until at most one root remains of each degree. Because the linked list H is sorted by degree, we can perform all the like operations quickly.

BINOMIAL-HEAP-UNION(H_1, H_2)

```
H ← MAKE-BINOMIAL-HEAP ( )
head[H] ← BINOMIAL-HEAP-MERGE(H1, H2)
Free the objects H1 and H2 but not the lists they point to
if head[H] = NIL
then return H
prev-x ← NIL
x ← head[H]
next-x ← sibling[x]
while next-x ≠ NIL
do if (degree[x] ≠ degree[next-x]) or
(sibling[next-x] ≠ NIL and degree[sibling[next-x]] = degree[x])
then prev-x ← x                                  ⇒ case 1 and 2
x ← next-x                                       ⇒ case 1 and 2
else if key[x] ≤ key[next-x]
then sibling[x] ← sibling[next-x]                ⇒ case 3
BINOMIAL-LINK(next-x, x)                         ⇒ case 3
else if prev-x = NIL                             ⇒ case 4
then head[H] ← next-x                            ⇒ case 4
else sibling[prev-x] ← next-x                    ⇒ case 4
BINOMIAL-LINK(x, next-x)                         ⇒ case 4
x ← next-x                                       ⇒ case 4
next-x ← sibling[x]
return H
```

BINOMIAL-HEAP-MERGE(H_1, H_2)

```
a ← head[H1]
b ← head[H2]
head[H1] ← min-degree (a, b)
if head[H1] = NIL
return
if head[H1] = b
then b ← a
a ← head[H1]
while b ≠ NIL
```

```
do if sibling[a] = NIL
then sibling[a] ← b
return
else if degree [sibling[a]] < degree[b]
then a ← sibling[a]
else c ← sibling[b]
sibling[b] ← sibling[a]
sibling[a] ← b
a ← sibling[a]
b ← c
```

There are four cases/conditions that occur while performing union on binomial heaps.

Case 1 : When degree[x] ≠ degree[next-x] = degree [sibling[next-x]], then pointers moves one position further down the root list.

sibling
prev-x x next-x [next-x] prev-x x next-x
a b c d a b c d

Case1

B_k B_L B_k B_L

(*a*) (*b*)

Fig. 2.19.1.

Case 2 : It occurs when x is the first of three roots of equal degree, that is, degree[x] = degree[next-x] = degree[sibling[next-x]], then again pointer move one position further down the list, and next iteration executes either case 3 or case 4.

sibling
prev-x x next-x [next-x] prev-x x next-x
a b c d a b c d

Case 2

B_k B_k B_k B_k B_k B_k

(*a*) (*b*)

Fig. 2.19.2.

Case 3 : If degree[x] = degree[next-x] ≠ degree [sibling[next-x]] and key[x] ≤ key[next-x], we remove next-x from the root list and link it to x, creating B_{k+1} tree.

sibling
prev-x x next-x [next-x] prev-x x next-x
a b c d

B
k B_k

B_L key [x] ≥ a b d

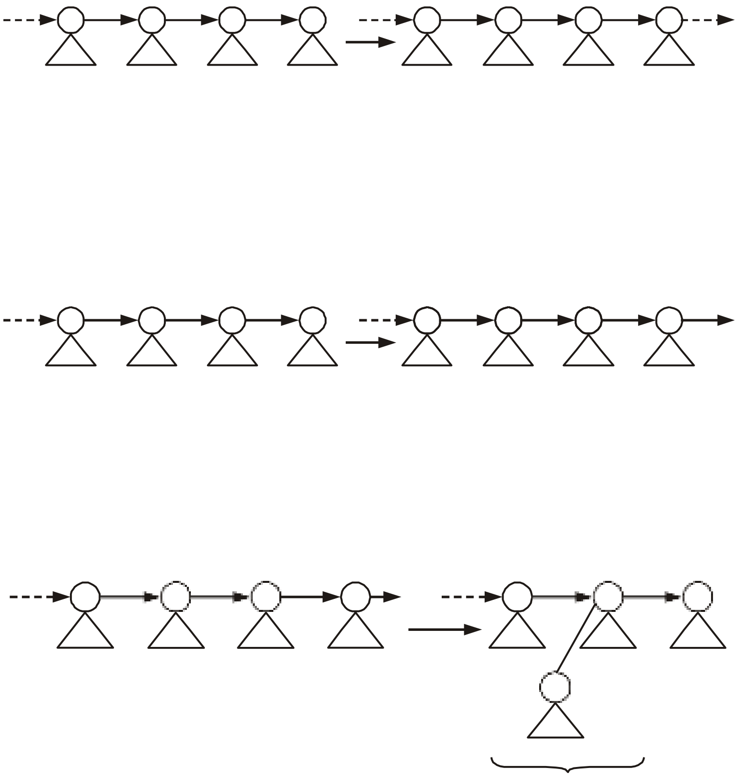
key [next-x]

(*a*) Case 3 c B_k B_L

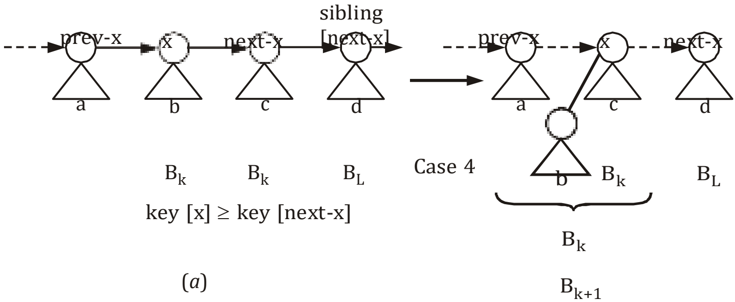
B_k

B_{k+1}

Fig. 2.19.3. (*b*)



Case 4 : degree[*x*] = degree[next-*x*] ≠ degree[sibling[next-*x*] and key[next-*x*] ≤ key *x*, we remove *x* from the root list and link it to next-*x*, again creating a B_{k+1} tree.



(*a*) (*b*)

Fig. 2.19.4.

Time complexity of union of two binomial heap is O(log *n*).

Que 2.20. Explain properties of binomial heap. Write an algorithm to perform uniting two binomial heaps. And also to find Minimum key.

AKTU 2017-18, Marks 10

Answer

Properties of binomial heap : Refer Q. 2.18, Page 2–33B, Unit-2.

Algorithm for union of binomial heap : Refer Q. 2.19, Page 2–33B, Unit-2.

Minimum key :

BINOMIAL-HEAP-EXTRACT-MIN (*H*) :

1. Find the root *x* with the minimum key in the root list of *H*, and remove *x* from the root list of *H*.
2. *H*' ← MAKE-BINOMIAL-HEAP( ).
3. Reverse the order of the linked list of *x*'s children, and set head[*H*'] to point to the head of the resulting list.
4. *H* ← BINOMIAL-HEAP-UNION(*H*, *H*').
5. Return *x*

Since each of lines 1-4 takes O(log *n*) time of *H* has *n* nodes, BINOMIAL-HEAP-EXTRACT-MIN runs in O(log *n*) time.

Que 2.21. Construct the binomial heap for the following sequence of number 7, 2, 4, 17, 1, 11, 6, 8, 15.

Answer

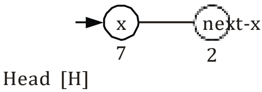
Numerical :

Insert 7 :

Head [H]

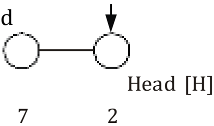
7

Insert 2 :



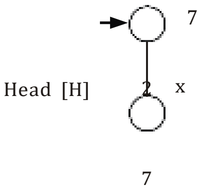
prev-x = NIL

degree [x] = 0. So, degree [x] ≠ degree [next-x] is false.
degree [next-x] = 0 and Sibling [next-x] = NIL
So, case 1 and 2 are false here.
Now key [x] = 7 and key [next-x] = 2
Now prev-x = NIL
then Head [H] ← next-x and
i.e.,



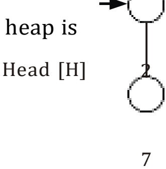
and BINOMIAL-LINK (x, next-x)
i.e.,

Now

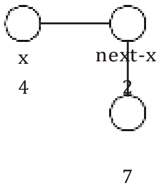


and next-x = NIL

So, after inserting 2, binomial heap is

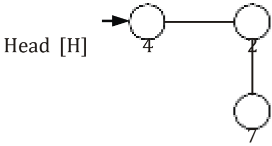


Insert 4 :



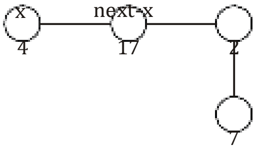
degree [x] ≠ degree [next-x]
So, Now next-x makes x and x makes prev-x.

Now next-x = NIL
So, after inserting 4, final binomial heap is :



Insert 17 :

After Binomial-Heap-Merge, we get



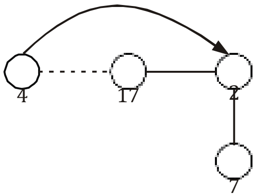
degree [x] = degree [next-x]

degree [Sibling-[next-x]] $\neq$ degree [x]

key [x] $\leq$ key [next-x]

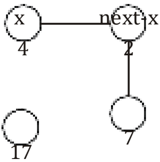
$4 \leq 17$ [True]

So,



and call Binomial-Link [next-x, x]

We get



degree [x] = degree [next-x]

Sibling [next-x] = NIL

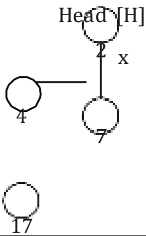
Key [x] $\leq$ key [next-x] [False]

prev-x = NIL then

Head [H] $\leftarrow$ [next-x]

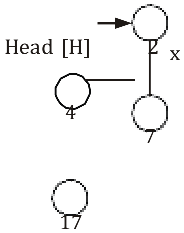
Binomial-Link [x, next-x]

$x \leftarrow$ next-x

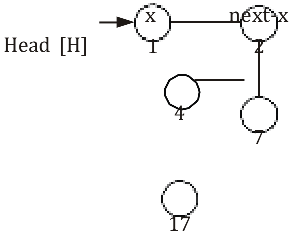


next-x = NIL

So, after inserting 17, final binomial heap is :



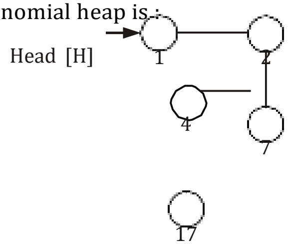
Insert 1 :



degree [x] ≠ degree [next-x]

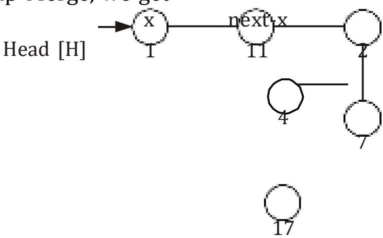
So, next-x makes x and next-x = NIL

and after inserting 1, binomial heap is :



Insert 11 :

After Binomial-Heap-Merge, we get

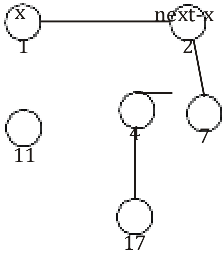


degree [x] = degree [next-x]

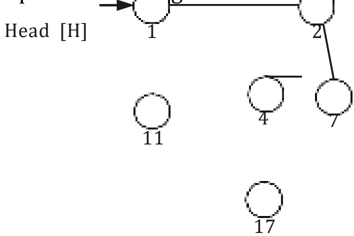
degree [Sibling [next-x]] ≠ degree [x]

key [x] $\leq$ key [next-x] [True]

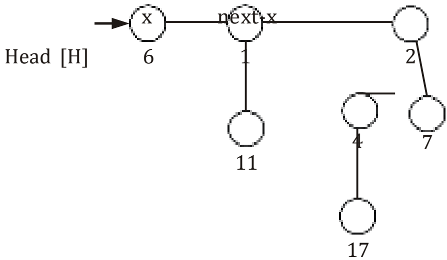
So,



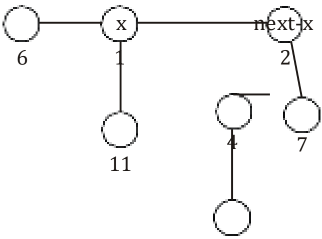
degree [x] ≠ degree [next-x]
So, next-x makes x and next-x = NIL
and final binomial heap after inserting 11 is



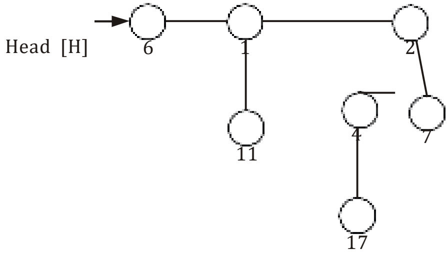
Insert 6 :



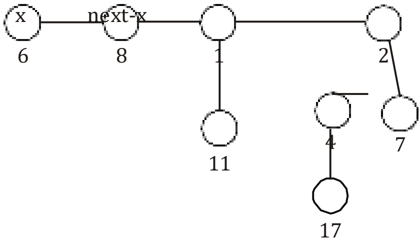
degree [x] ≠ degree [next-x]
So, next-x becomes x
Sibling [next-x] becomes next-x.
i.e.,



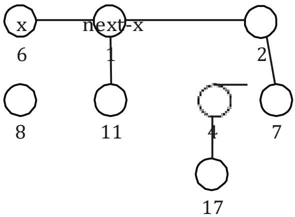
degree [x] ≠ degree [next-x]
So, no change and final heap is :



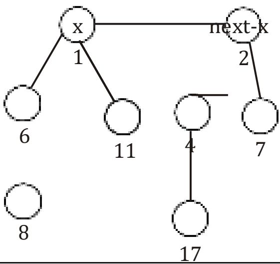
Insert 8 :



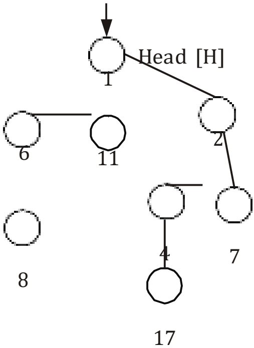
degree [x] = degree [next-x]
degree [Sibling [next-x]] ≠ degree [x]
key [x] ≤ key [next-x] [True]
So,



degree [x] = degree [next-x]
degree [Sibling p[next-x]] ≤ degree [x]
key [x] ≤ key [next-x] [False]
prev-x = NIL
So,

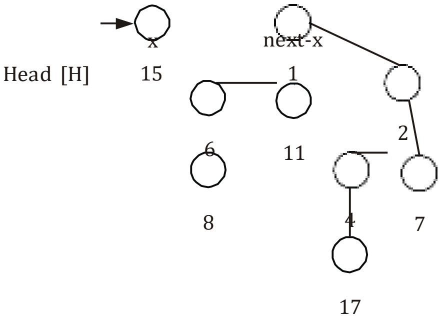


degree [x] = degree [next-x]
Sibling [next-x] = NIL
key [x] $\leq$ key [next-x] [True]
So, Sibling [x] = NIL.
and



next [x] = NIL
So, this is the final binomial heap after inserting 8.

Insert 15 :



degree [x] $\neq$ degree [next-x]
So, no change and this is the final binomial heap after inserting 15.

Que 2.22. **Explain the algorithm to delete a given element in a binomial heap. Give an example for the same.**

AKTU 2019-20, Marks 07

Answer

Deletion of key from binomial heap :
The operation BINOMIAL-HEAP-DECREASE (H, x, k) assigns a new key 'k' to a node 'x' in a binomial heap H.

BINOMIAL-HEAP-DECREASE-KEY (H, x, k)
1. if k > key [x] then
2. Message "error new key is greater than current key"
3. key [x] $\leftarrow$ k

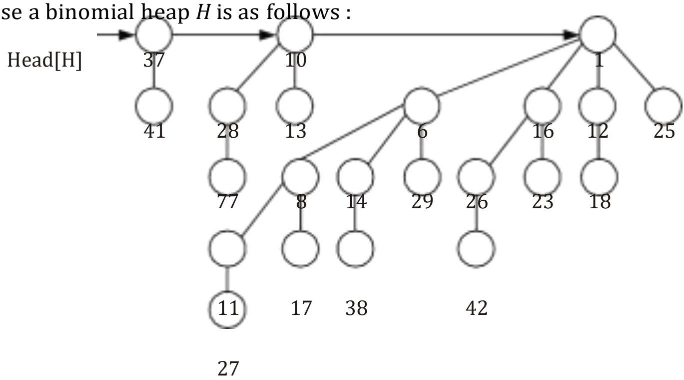
4. $y \leftarrow x$
5. $z \leftarrow P[y]$
6. While ($z \neq$ NIL) and key [y] < key [z]
7. do exchange key [y] $\leftrightarrow$ key [z]
9. $y \leftarrow z$
10. $z \leftarrow P[y]$

Deleting a key : The operation BINOMIAL- HEAP-DELETE (H, x) is used to delete a node x's key from the given binomial heap H. The following implementation assumes that no node currently in the binomial heap has a key of $-\infty$.

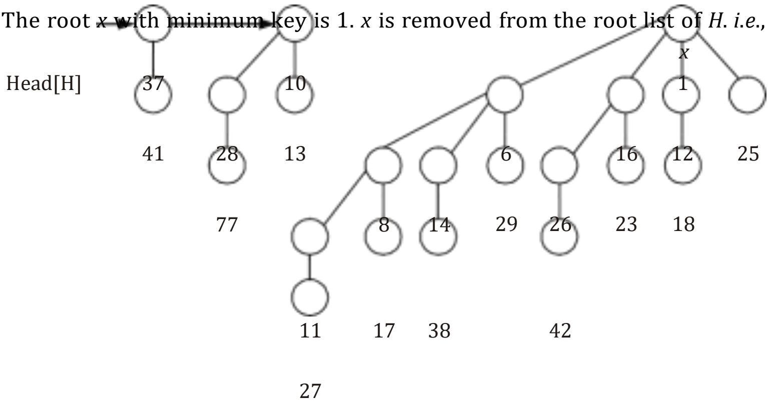
BINOMIAL-HEAP-DELETE (H, x)
1. BINOMIAL-HEAP-DECREASE-KEY (H, x, $-\infty$)
2. BINOMIAL-HEAP-EXTRACT-MIN(H)

For example : Operation of Binomial-Heap-Decrease (H, x, k) on the following given binomial heap :

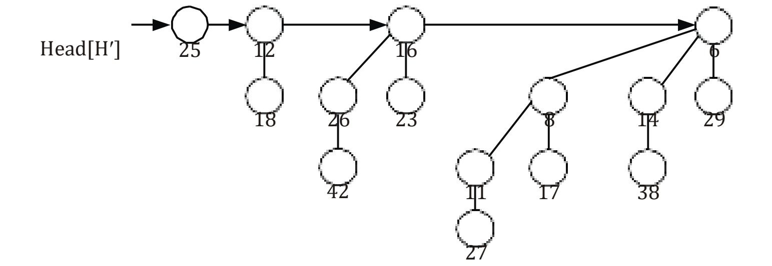
Suppose a binomial heap H is as follows :



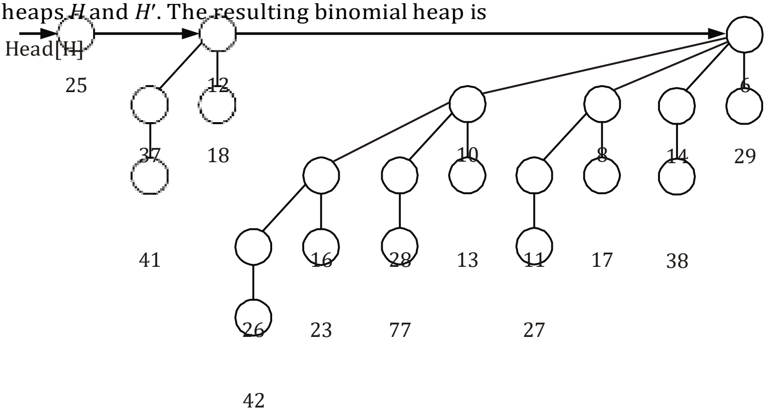
The root x with minimum key is 1. x is removed from the root list of H. *i.e.*,



Now, the linked list of x's children is reversed and set head[H'] to point to the head of the resulting list, *i.e.*, another binomial heap H'.



Now, call BINOMIAL-HEAP-UNION (H, H') to uniting the two binomial heaps H and H'. The resulting binomial heap is



PART-4

Fibonacci Heaps.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 2.23. **What is a Fibonacci heap ? Discuss the applications of Fibonacci heaps.**

Answer

1. A Fibonacci heap is a set of min-heap-ordered trees.
2. Trees are not ordered binomial trees, because

a. Children of a node are unordered.

b. Deleting nodes may destroy binomial construction.

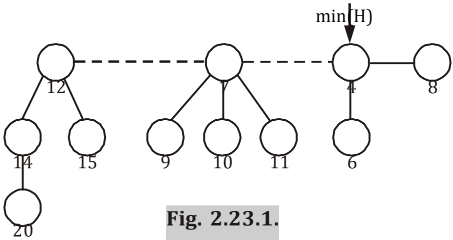


Fig. 2.23.1.

3. Fibonacci heap H is accessed by a pointer min[H] to the root of a tree containing a minimum key. This node is called the minimum node.
4. If Fibonacci heap H is empty, then min[H] = NIL.

Applications of Fibonacci heap :

1. Fibonacci heap is used for Dijkstra's algorithm because it improves the asymptotic running time of this algorithm.
2. It is used in finding the shortest path. These algorithms run in O(n^2) time if the storage for nodes is maintained as a linear array.

Que 2.24. **What is Fibonacci heap ? Explain CONSOLIDATE operation with suitable example for Fibonacci heap.**

AKTU 2015-16, Marks 15

Answer

Fibonacci heap : Refer Q. 2.23, Page 2–44B, Unit-2.

CONSOLIDATE operation :

CONSOLIDATE(*H*)

```
for i ← 0 to D(n[H])
do A[i] ← NIL
for each node w in the root list of H
do x ← w
d ← degree[x]
while A[d] ≠ NIL
do y ← A[d]   ◁ Another node with the same degree as x.
if key[x] > key[y]
then exchange x ↔ y
FIB-HEAP-LINK(H, y, x)
A[d] ← NIL
d ← d + 1
A[d] ← x
min[H] ← NIL
for i ← 0 to D(n[H])
do if A[i] ≠ NIL
then add A[i] to the root list of H
```

```
if min[H] = NIL or key [A[i]] < key[min[H]]
then min[H] ← A[i]
```

FIB-HEAP-LINK(*H*, *y*, *x*)

```
remove y from the root list of H
make y a child of x, incrementing degree[x]
mark[y] ← FALSE
```

Que 2.25. **Define Fibonacci heap. Discuss the structure of a Fibonacci heap with the help of a diagram. Write a function for uniting two Fibonacci heaps.**

Answer

Fibonacci heap : Refer Q. 2.23, Page 2–44B, Unit-2.

Structure of Fibonacci heap :

1. **Node structure :**
   a. The field "mark" is True if the node has lost a child since the node became a child of another node.
   b. The field "degree" contains the number of children of this node. The structure contains a doubly-linked list of sibling nodes.

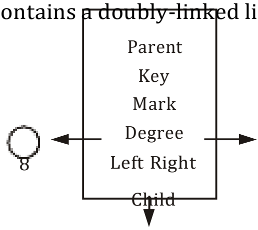


Fig. 2.25.1. Node structure.

2. **Heap structure :**
   min(*H*) : Fibonacci heap *H* is accessed by a pointer min[*H*] to the root of a tree containing a minimum key; this node is called the minimum node. If Fibonacci heap *H* is empty, then min[*H*] = NIL.
   ***n*(*H*) :** Number of nodes in heap *H*

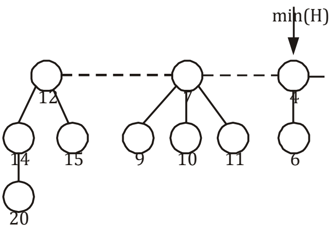


Fig. 2.25.2. Heap structure.

Function for uniting two Fibonacci heap :

Make-Heap :

MAKE-FIB-HEAP( )

allocate(H)
min(H) = NIL
$n(H) = 0$

FIB-HEAP-UNION(H_1, H_2)

1. $H \leftarrow$ MAKE-FIB-HEAP( )
2. min[H] $\leftarrow$ min[H_1]
3. Concatenate the root list of H_2 with the root list of H
4. if (min[H_1] = NIL) or (min[H_2] $\neq$ NIL and min[H_2] < min[H_1])
5. then min[H] $\leftarrow$ min[H_2]
6. $n[H] \leftarrow n[H_1] + n[H_2]$
7. Free the objects H_1 and H_2
8. return H

Que 2.26. Discuss following operations of Fibonacci heap :

i. Make-Heap
ii. Insert
iii. Minimum
iv. Extract-Min

Answer

i. Make-Heap : Refer Q. 2.25, Page 2–46B, Unit-2.

ii. Insert : (H, x)

1. degree[x] $\leftarrow$ 0
2. $p[x] \leftarrow$ NIL
3. child[x] $\leftarrow$ NIL
4. left[x] $\leftarrow x$
5. right[x] $\leftarrow x$
6. mark[x] $\leftarrow$ FALSE
7. concatenate the root list containing x with root list H
8. if min[H] = NIL or key[x] < key[min[H]]
9. then min[H] $\leftarrow x$
10. $n[H] \leftarrow n[H] + 1$

To determine the amortized cost of FIB-HEAP-INSERT, Let H be the input Fibonacci heap and H' be the resulting Fibonacci heap, then $t(H) = t(H) + 1$ and $m(H') = m(H)$, and the increase in potential is,
$(t(H) + 1) + 2m(H) - (t(H) + 2m(H)) = 1$
Since the actual cost is O(1), the amortized cost is O(1) + 1 = O(1)

iii. Minimum :

The minimum node of a Fibonacci heap H is always the root node given by the pointer min[H], so we can find the minimum node in O(1) actual time. Because the potential of H does not change, the amortized cost of this operation is equal to its O(1) actual cost.

iv. FIB-HEAP-EXTRACT-MIN(H)

1. $z \leftarrow$ min[H]
2. if $z \neq$ NIL

3. then for each child x of z

4. do add x to the root list of H
5. $p[x] \leftarrow$ NIL
6. remove z from the root list of H
7. if z = right[z]
8. then min[H] $\leftarrow$ NIL
9. else min[H] $\leftarrow$ right[z]
10. CONSOLIDATE (H)
11. $n[H] \leftarrow n[H] - 1$
12. return z

PART-5

Tries, Skip List.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 2.27. What is trie ? What are the properties of trie ?

Answer

1. A trie (digital tree / radix tree / prefix free) is a kind of search tree *i.e.*, an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings.
2. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated.
3. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string.
4. Values are not necessarily associated with every node. Rather, values tend only to be associated with leaves, and with some inner nodes that correspond to keys of interest.

Properties of a trie :

1. Trie is a multi-way tree.
2. Each node has from 1 to d children.
3. Each edge of the tree is labeled with a character.
4. Each leaf node corresponds to the stored string, which is a concatenation of characters on a path from the root to this node.

Que 2.28. Write an algorithm to search and insert a key in trie data structure.

Answer

Search a key in trie :

Trie-Search(t, $P[k..m]$) // inserts string P into t

1. if t is leaf then return true
2. else if t.child($P[k]$) = nil then return false
3. else return Trie-Search(t.child($P[k]$), $P[k + 1..m]$)

Insert a key in trie :

Trie-Insert(t, $P[k..m]$)

1. if t is not leaf then //otherwise P is already present
2. if t.child($P[k]$) = nil then

   //Create a new child of t and a "branch" starting with that child and storing $P[k..m]$
3. else Trie-Insert(t.child($P[k]$), $P[k + 1..m]$)

Que 2.29. **What is skip list ? What are its properties ?**

Answer

1. A skip list is built in layers.
2. The bottom layer is an ordinary ordered linked list.
3. Each higher layer acts as an "express lane", where an element in layer i appears in layer $(i + 1)$ with some fixed probability p (two commonly used values for p are ½ and ¼.).
4. On average, each element appears in $1/(1- p)$ lists, and the tallest element (usually a special head element at the front of the skip list) in all the lists.
5. The skip list contains $\log_{1/p} n$ (*i.e.*, logarithm base $1/p$ of n).

Properties of skip list :

1. Some elements, in addition to pointing to the next element, also point to elements even further down the list.
2. A level k element is a list element that has k forward pointers.
3. The first pointer points to the next element in the list, the second pointer points to the next level 2 element, and in general, the i^{th} pointer points to the next level i element.

Que 2.30. **Explain insertion, searching and deletion operation in skip list.**

Answer

Insertion in skip list :

1. We will start from highest level in the list and compare key of next node of the current node with the key to be inserted.

2. If key of next node is less than key to be inserted then we keep on moving forward on the same level.
3. If key of next node is greater than the key to be inserted then we store the pointer to current node *i* at update[*i*] and move one level down and continue our search.

At the level 0, we will definitely find a position to insert given key.

Insert(list, searchKey)

1. local update[0...MaxLevel+1]
2. x : = list → header
3. for i : = list → level down to 0 do
4. while x → forward[i] → key forward[i]
5. update[i] : = x
6. x : = x → forward[0]
7. lvl : = randomLevel()
8. if lvl > list → level then
9. for i : = list → level + 1 to lvl do
10. update[i] : = list → header
11. list → level : = lvl
12. x : = makeNode(lvl, searchKey, value)
13. for i : = 0 to level do
14. x → forward[i] : = update[i] → forward[i]
15. update[i] → forward[i] : = x

Searching in skip list :
Search(list, searchKey)

1. x : = list → header
2. loop invariant : x → key level down to 0 do
3. while x → forward[i] → key forward[i]
4. x : = x → forward[0]
5. if x → key = searchKey then return x → value
6. else return failure

Deletion in skip list :
Delete(list, searchKey)

1. local update[0..MaxLevel+1]
2. x : = list → header
3. for i : = list → level down to 0 do
4. while x → forward[i] → key forward[i]
5. update[i] : = x

6. $x := x \rightarrow$ forward[0]
7. if $x \rightarrow$ key = searchKey then
8. for $i := 0$ to list $\rightarrow$ level do
9. if update[i] $\rightarrow$ forward[i] $\neq x$ then break
10. update[i] $\rightarrow$ forward[i] := $x \rightarrow$ forward[i]
11. free(x)
12. while list $\rightarrow$ level > 0 and list $\rightarrow$ header $\rightarrow$ forward[list $\rightarrow$ level] = NIL do
13. list $\rightarrow$ level := list $\rightarrow$ level – 1

Que 2.31. Given an integer x and a positive number n, use divide and conquer approach to write a function that computes x^n with time complexity O(log n).

AKTU 2018-19, Marks 10

Answer

Function to calculate x^n with time complexity O(log n) :

```
int power(int x, unsigned int y)
{
int temp;
if(y == 0)
return 1;
temp = power(x, y/2);
if (y%2 == 0)
return temp * temp;
else
return x * temp * temp;
}
```

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. Define red-black tree and give its properties.

Ans. Refer Q. 2.1.

Q. 2. Insert the following element in an initially empty RB-Tree. 12, 9, 81, 76, 23, 43, 65, 88, 76, 32, 54. Now delete 23 and 81.

Ans. Refer Q. 2.7.

Q. 3. Define a B-tree of order *m*. Explain the searching and insertion algorithm in a B-tree.

Ans. Refer Q. 2.10.

Q. 4 Explain the insertion and deletion algorithm in a red-blacktree.

Ans. **Insertion algorithm :** Refer Q. 2.1.

Deletion algorithm : Refer Q. 2.5.

Q. 5. Using minimum degree '*t*' as 3, insert following sequence of integers 10, 25, 20, 35, 30, 55, 40, 45, 50, 55, 60, 75, 70, 65, 80, 85 and 90 in an initially empty B-Tree. Give the number of nodes splitting operations that take place.

Ans. Refer Q. 2.17.

Q. 6. What is binomial heap ? Describe the union of binomial heap.

Ans. Refer Q. 2.19.

Q. 7. What is a Fibonacci heap ? Discuss the applications of Fibonacci heaps.

Ans. Refer Q. 2.23.

Q. 8. What is trie ? Give the properties of trie.

Ans. Refer Q. 2.27.

Q. 9. Explain the algorithm to delete a given element in a binomial heap. Give an example for the same.

Ans. Q. 2.22.

Q. 10. Explain skip list. Explain its operations.

Ans. **Skip list :** Refer Q. 2.29.

Operations : Refer Q. 2.30.