

Unit 2 DAA Notes

Red Black Tree

Why to study Red-Black tree?

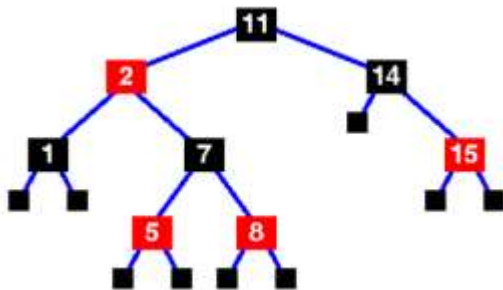
Most of the Binary Search Tree operations (e.g., search, max, min, insert, delete. etc.) take $O(h)$ time where h is the height of the BST. The cost of these operations may become $O(n)$ for a skewed Binary tree. If we make sure that the height of the tree remains $O(\log n)$ after every insertion and deletion, then we can guarantee an upper bound of $O(\log n)$ for all these operations. The height of a Red-Black tree is always $O(\log n)$ where n is the number of nodes in the tree.

S.No	Operation	Time Complexity
1	Insert	$O(\log n)$
2	Search	$O(\log n)$
3	Delete	$O(\log n)$

Introduction

A red-black tree is a self-balancing binary search tree with one extra bit at each node, which is commonly regarded as the color (red or black). These colors are used to keep the tree balanced as insertions and deletions are made.

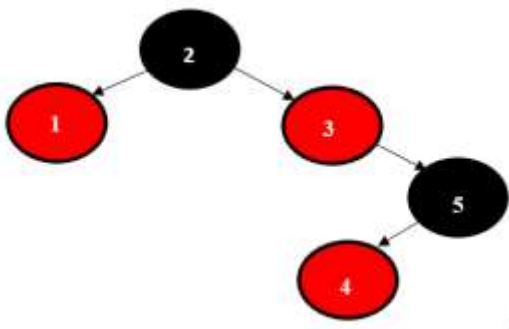
Red-Black tree=Binary Search Tree +One Extra Bit for color (either Red or Black)



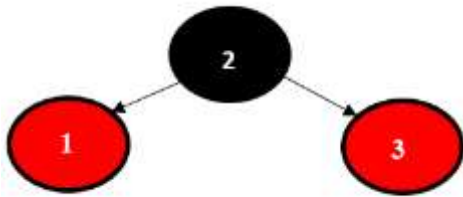
RB Tree is also a self-Balancing tree like B-Tree. But, it uses coloring scheme to balance itself.

Properties of Red- Black Tree

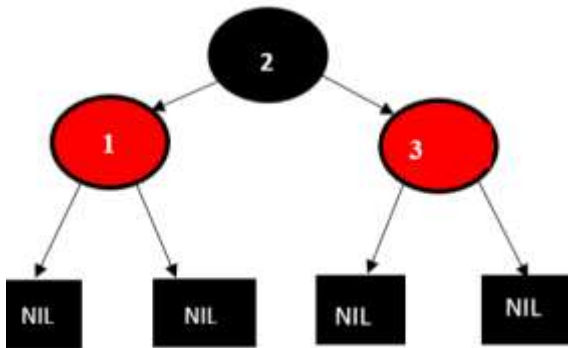
Property 1: Red black tree is the binary search tree



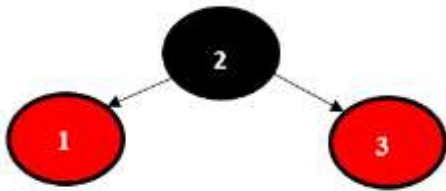
Property 2. Every node is either red or black



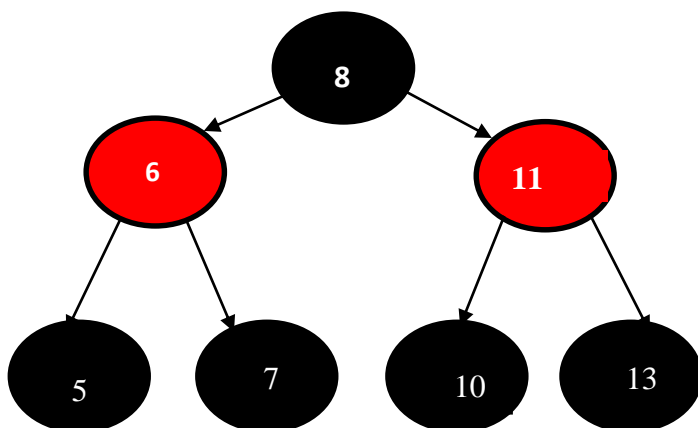
Property 3. Every leaf node (NIL) is black



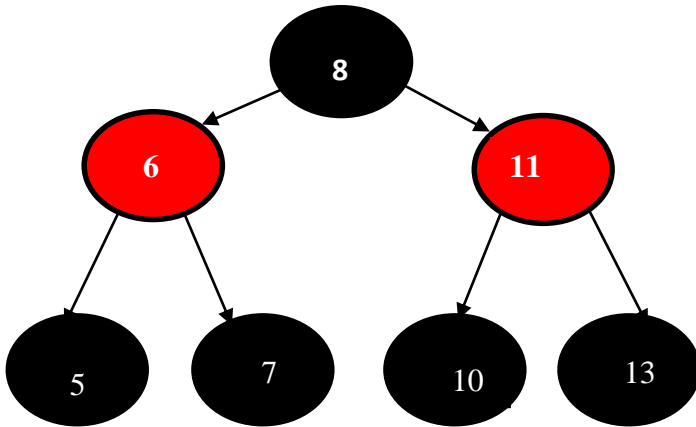
Property 4. Root Node should always be black



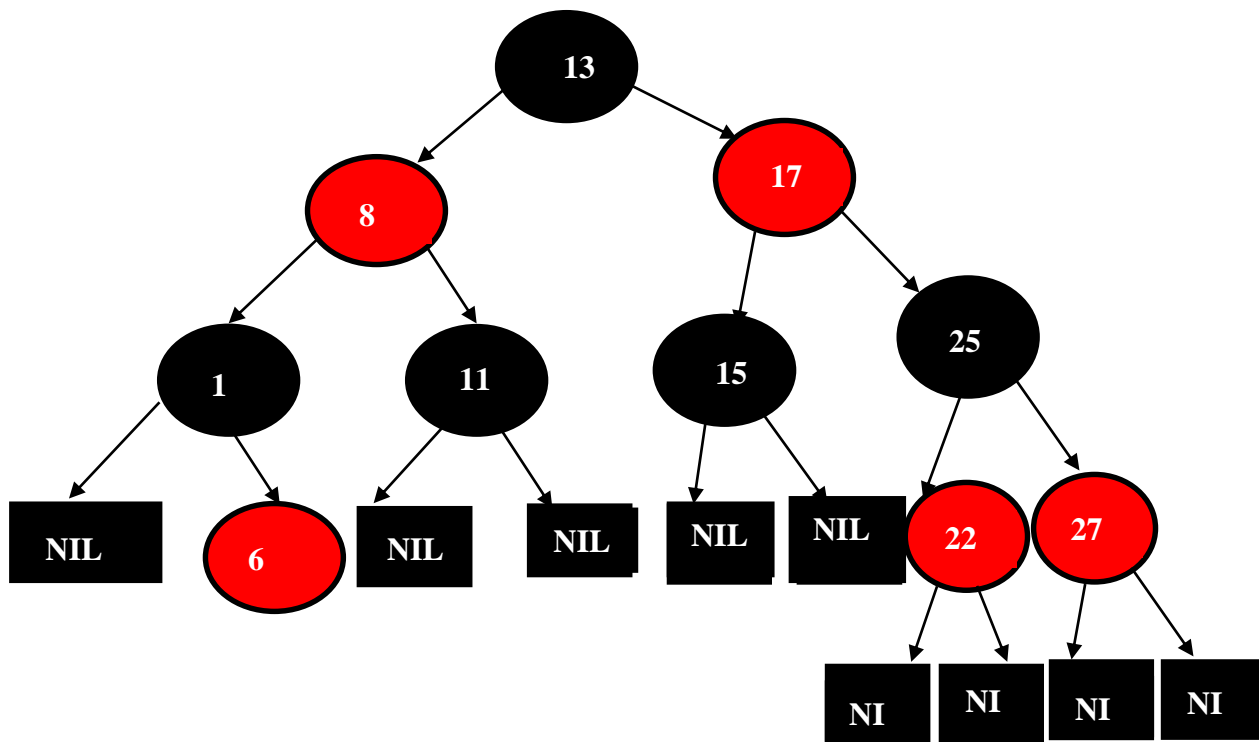
Property 5. If a node is red then both of its children are black



Property 6. Every simple path from a node to a descendant leaf contains the same number of black nodes.



Q. Is The below tree a Red Black Tree?



The above tree is a Red-Black tree if it satisfies all the properties of RB Tree.

- **Property 1** says that every red-black tree is a binary search tree i.e. left child node is smaller than root node and right child node is greater than the root node.

// given red-black tree fulfils this property

- **Property 2** says that there can be only two types of nodes i.e. red or black

// given tree fulfil this property as there are only red and black nodes present in the tree.

- **Property 3** says that every leaf node(NIL) should be black in colour

// in the above given tree all the NIL nodes are represented by black colour.

- **Property 4** tells that head node of the tree should be black in colour

// Here root node 13 is black in colour so it follows this property.

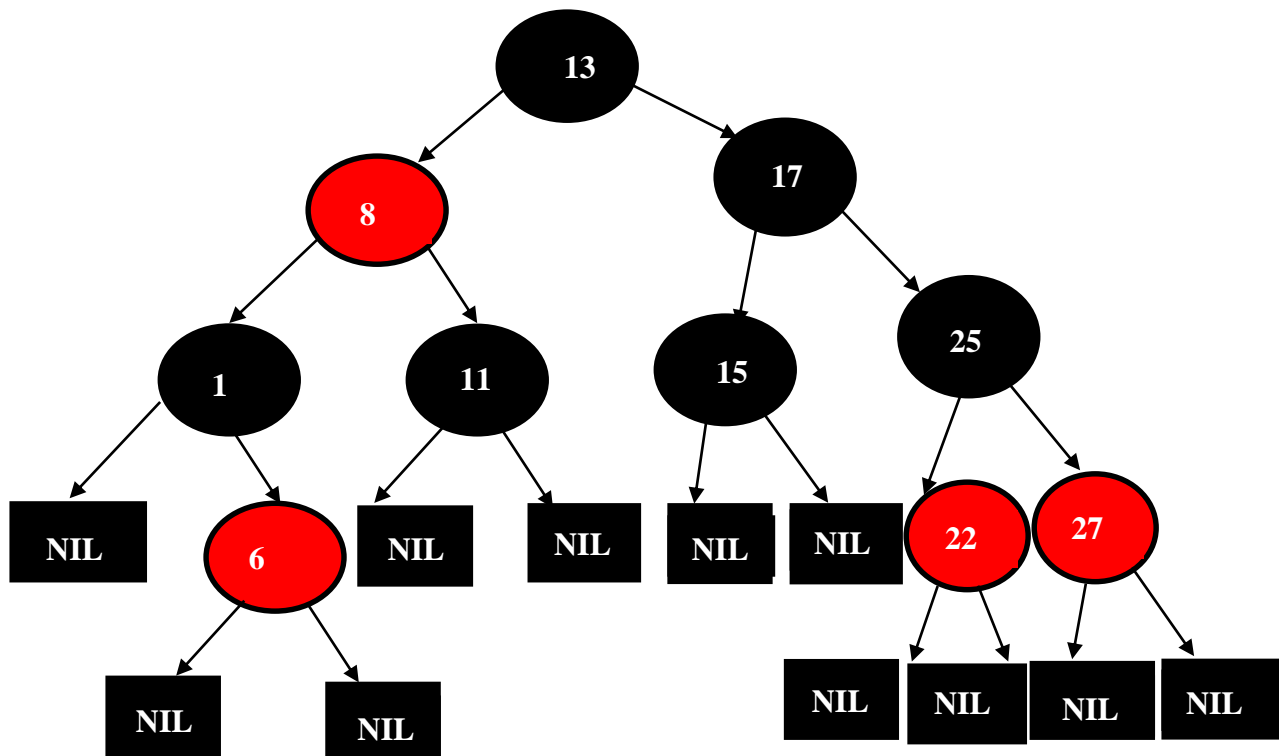
- **Property 5** says that there should be no red –red parent child relationship.

// in the above given tree all the red nodes has black children.

- **Property 6** says every path from node to leaf node has same no. of black nodes

// in the given tree every path from root node has 2 black nodes in total so it fulfils this property also.

After checking all the above properties, we can say that the given tree is a red-black tree
Is This a Red Black Tree?



The above tree is a Red-Black tree if it satisfies all the properties of RB Tree.

- **Property 1** says that every red-black tree is a binary search tree i.e. left child node is smaller than root node and right child node is greater than the root node.

// given red-black tree fulfils this property

- **Property 2** says that there can be only two types of nodes i.e. red or black

// given tree fulfil this property as there are only red and black nodes present in the tree.

- **Property 3** says that every leaf node(NIL) should be black in colour

// in the above given tree all the NIL nodes are represented by black colour.

- **Property 4** tells that head node of the tree should be black in colour

// Here root node 13 is black in colour so it follows this property.

- **Property 5** says that there should be no red –red parent child relationship.

// in the above given tree all the red nodes has black children.

- **Property 6** says every path from node to leaf node has same no. of black nodes

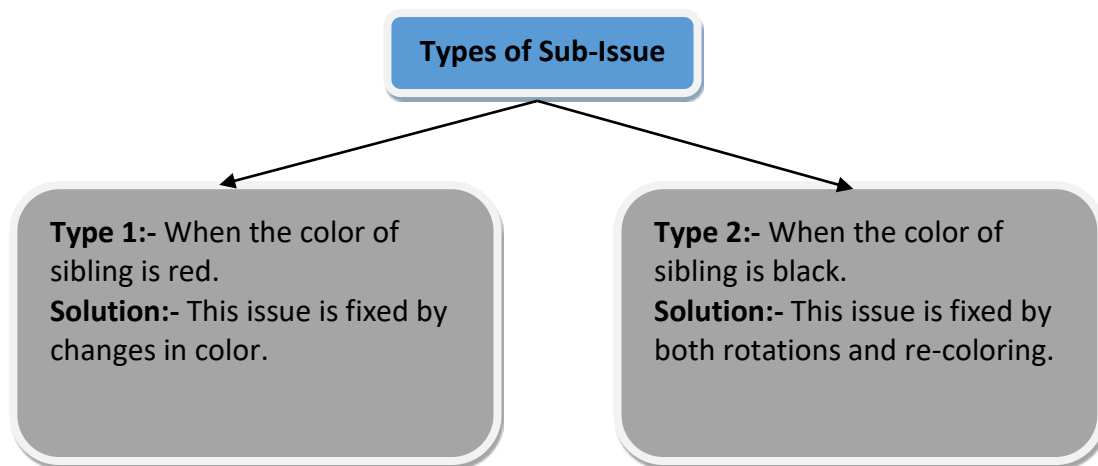
// in the given tree left path from the root node has total two black nodes and right path has total 3 black nodes. So $2 \neq 3$ due to which it violate this property.

After checking all the above properties we can say that the given tree is not a red-black tree.

RB Tree Creation / Insertion in RB Tree

Some points to be noted before proceeding towards cases.

1. A new node will be always inserted with red color.
2. Main Issue will arise when the parent of the newly inserted node is of red color. (**Violation of Black Children Property**) otherwise insert the node simply.
3. The type of sub-issue is now decided by the position of the newly inserted node in relation to its grandparent and the color of its parent's sibling. (Will be discussed in detail later on)



RB-INSERT algorithm will insert a new node with red color in the same manner as we have studied in binary search tree.

RB Tree INSERT-FIXUP algorithm will be called when the main issue will occur i.e. the parent's color of the newly inserted node is also red. After then, we will explore the type of sub-issue and will resolve it either by recoloring the nodes or by performing rotations.

RB-INSERT (T, z)

```

1. y ← nil[T]
2. x ← root[T]
3. while x ≠ nil[T]
4.   do y ← x
5.   If key[z] < key[x]
6.     then x ← left[x]
7.   else x ← right[x]
8. p[z] ← y
9. if y = nil[T]
10.  then root[T] ← z
11.  else if key[z] < key [y]
12.    then left[y] ← z
13.    else right[y] ← z
14. left[z] ← nil[T]
  
```

RB-INSERT-FIXUP (T, z)

```

1. while color[p[z]] = RED
2.   do if p[z] = left[p[p[z]]]           // MAIN
3.     CASE 1
4.       then y ← right[p[p[z]]]
5.       If color[y] = RED
6.         then color[p[z]] ← BLACK      //
7.         SUBCASE 1
8.           color[y] ← BLACK             //
9.         SUBCASE 1
10.        color[p[p[z]]] ← RED           //
11.        SUBCASE 1
12.        z ← p[p[z]]                   // SUBCASE
13.        1
14.     else if z = right[p[z]]
  
```

```

15. right[z] ← nil[T]
16. color[z] ← RED
17. RB-INSERT-FIXUP
    (T,z)

10.   then z ← p[z]           // SUBCASE
    2
11.   LEFT-ROTATE (T, z)      //
    SUBCASE 2
12.   color[p[z]] ← BLACK    // SUBCASE
    3
13.   color[p[p[z]]] ← RED   // SUBCASE
    3
14.   RIGHT-ROTATE (T, p[p[z]]) //
    SUBCASE 3
15.   else (same as then clause & exchange left
        with right and vice-versa) // MAIN
    CASE 2
16.   color[root[T]] ← BLACK

```

First of all, let's see the terminologies, we are using in RB tree insertion.

Terminology	Description
Z	Newly inserted node
P[z]	Parent of node z
P[P[z]]	Grandparent of node z
If P[z] ← left[P[P[z]]] then y ← right[P[P[z]]]	If parent of node z is left child of its parent, then uncle y will be right child of grandparent of z.
If P[z] ← right[P[P[z]]] then y ← left[P[P[z]]]	If parent of node z is right child of its parent, then uncle y will be left child of grandparent of z.
root[T]	Root of the tree
color[x]	Color of node x

NOTE: - Every NIL node is considered as of BLACK color.

RB-TREE-INSERT-FIXUP will be called
iff $\text{color}[p[z]] = \text{RED}$

MAIN CASE 1

If $p[z] = \text{left}[p[p[z]]]$ & $y = \text{right}[p[p[z]]]$

SUB-CASE 1 : When y is of red color
Then, change $\text{color}[p[z]] \leftarrow \text{BLACK}$
 $\text{color}[y] \leftarrow \text{BLACK}$
 $\text{color}[p[p[z]]] \leftarrow \text{RED}$
 $z \leftarrow [p[p[z]]]$
Call INSERT_FIXUP on new z i.e.
grand parent of z

Rectification is done by recoloring.

SUB-CASE 2 : When y is of black color
& z is right child of its parent
Then, exchange $p[z]$ with z and
perform
left rotation along z (which now
points to $p[z]$) i.e. LEFT-ROTATE (T, z)

Rectification is done by left rotation.

SUB-CASE 3 : When y is of black color
& z is left child of its parent
Then, change $\text{color}[p[z]] \leftarrow \text{BLACK}$
 $\text{color}[p[p[z]]] \leftarrow \text{RED}$
And perform right rotation along
grandparent of z
i.e. RIGHT-ROTATE($T, p[p[z]]$)

Rectification is done by recoloring as
well as right rotation.

MAIN CASE 2

If $p[z] = \text{right}[p[p[z]]]$ & $y = \text{left}[p[p[z]]]$

SUB-CASE 1 : When y is of red color
Then, change $\text{color}[p[z]] \leftarrow \text{BLACK}$
 $\text{color}[y] \leftarrow \text{BLACK}$
 $\text{color}[p[p[z]]] \leftarrow \text{RED}$
 $z \leftarrow [p[p[z]]]$
Call INSERT_FIXUP on new z i.e.
grand parent of z

Rectification is done by recoloring.

SUB-CASE 2 : When y is of black color
& z is left child of its parent
Then, exchange $p[z]$ with z and
perform
right rotation along z (which now
points to $p[z]$) i.e. RIGHT-ROTATE (T, z)

Rectification is done by right rotation.

SUB-CASE 3 : When y is of black color
& z is right child of its parent
Then, change $\text{color}[p[z]] \leftarrow \text{BLACK}$
 $\text{color}[p[p[z]]] \leftarrow \text{RED}$
And perform left rotation along
grandparent of z
i.e. LEFT-ROTATE($T, p[p[z]]$)

Rectification is done by recoloring as
well as left rotation.

NOTE:- MAIN CASE 2 is just a mirror image of MAIN CASE 1 (means left is exchanged with right and right is exchanged with left). And at last of both MAIN CASES, we will make color of root node as BLACK.

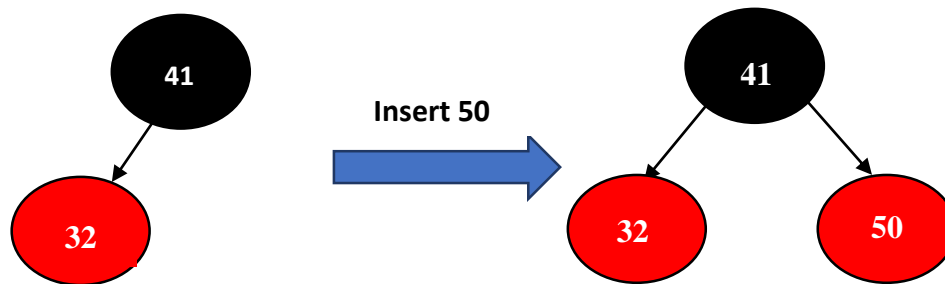
SUBCASES 2 & 3 use rotations and re-coloring to maintain the properties of RB tree after insertion of a new node into it. The following diagram will recall the LEFT & RIGHT rotations

(studied in Data Structure Course under topic AVL Tree)

After performing any of the rotation, the structural properties of BST should also be maintained.

Now, we will explore RB Tree Insertion using various examples complying with a particular case.

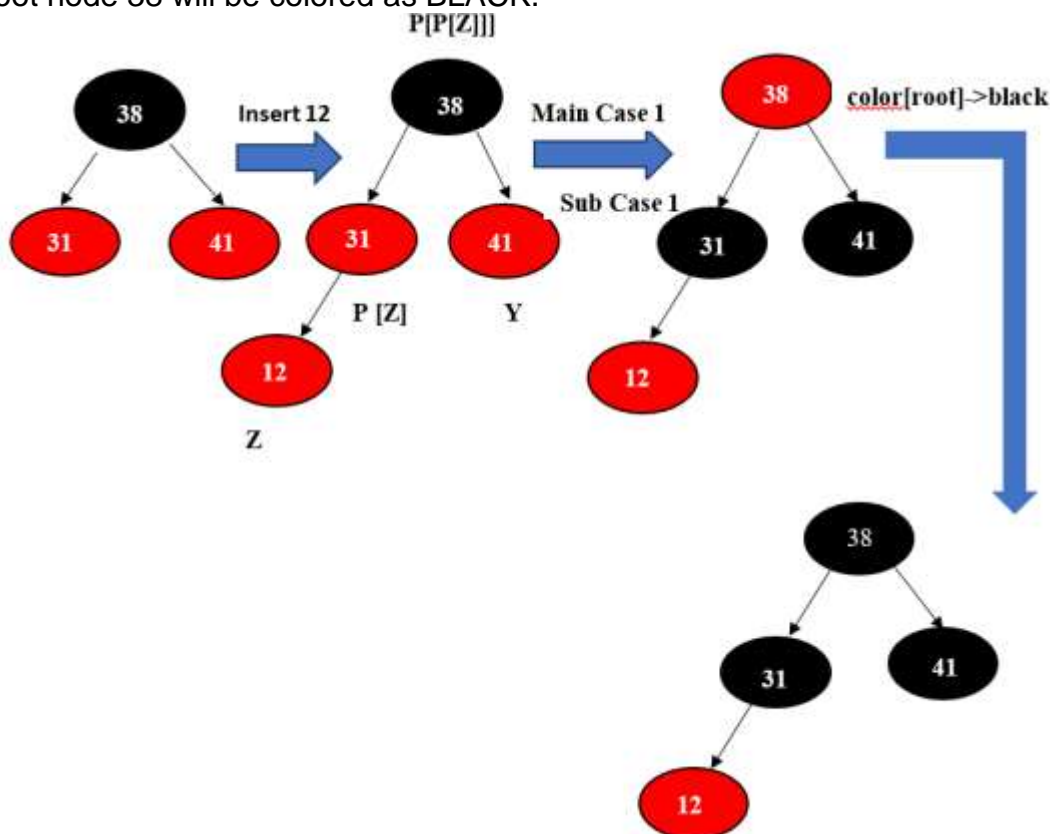
Example 1.



50 will be inserted as right child of node 41 following BST property. Here, $z = 50$ & $p[z] = 41$. Because, $color[p[z]] = RED$. So, there is no need to call INSERT-FIXUP. Node 50 will be inserted simply with red color.

Example 2.

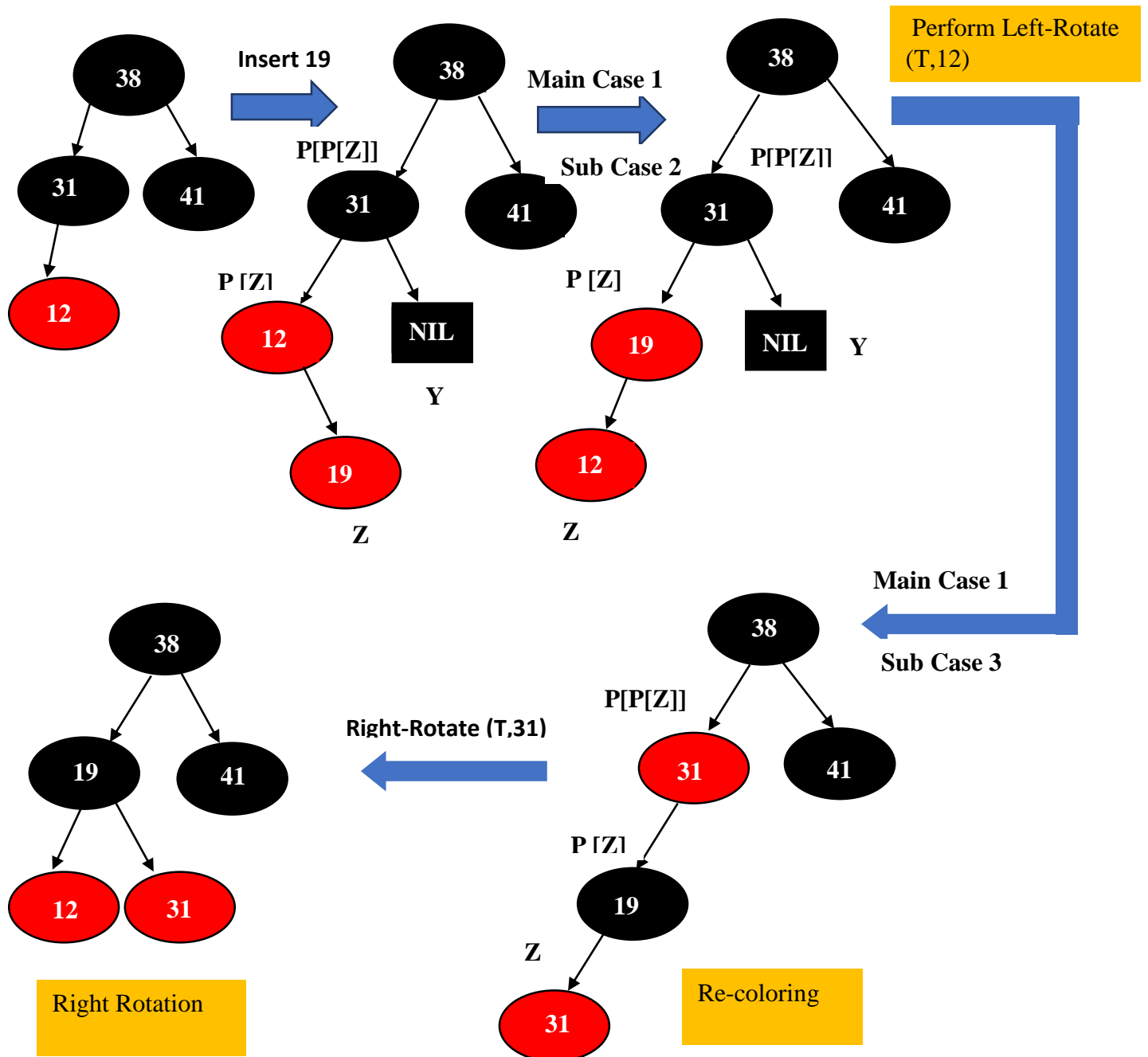
Insert 12 operation in given RB tree will lead to MAIN CASE 1 \rightarrow SUB CASE 1. Because, $P[z] = 31$ is left child of its parent i.e. $p[p[z]] = 38$ that complies to MAIN CASE 1. Further, the color of uncle $y = right[p[p[z]]] = 41$ is RED that complies to SUB CASE 1. So, change the color of both nodes 31 (parent) and 41 (uncle) to BLACK. Also, change the color of grandparent node 38 as RED. But, after executing MAIN CASE & SUB CASE, we will check the color of root node. If it is RED, make it BLACK. So, the root node 38 will be colored as BLACK.



Example 3.

Insert 19 will comply with MAIN CASE 1. Because, $P[z] = 12$ is left child of its parent node 31. Further,

right child of node 31 is NIL that means uncle $y = \text{NIL}$ of BLACK color. Z is right child of its parent. Further, it complies with SUBCASE 2. So, perform left rotation along new z i.e. 12. After this, SUBCASE 3 will be called. Change the color of node 19 (parent) as BLACK and of node 31 (grandparent) as RED. After re-coloring of nodes, perform right rotation along grandparent 31. At last, check for root node's color.



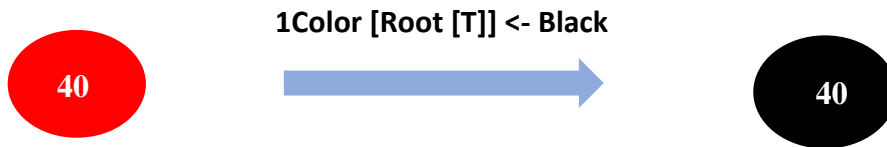
Example 4: Insert key sequence is as given below

40, 50, 70, 30, 42, 15, 20

Construct the Red Black tree for the above mentioned sequence.

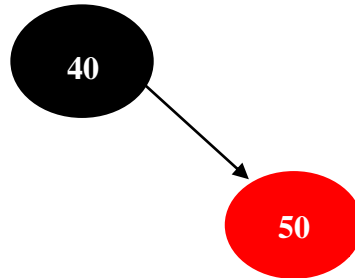
Step 1 Insert 40

// Property 4 violates here as every root node should be black in colour.



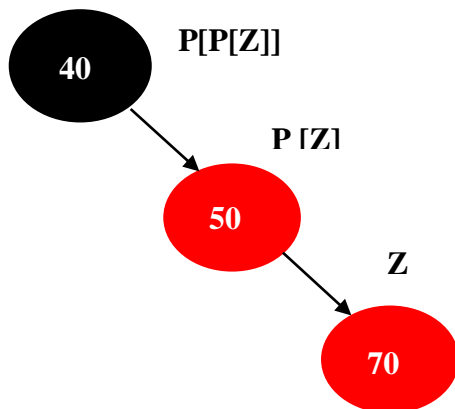
Step 2 Insert 50

// 50 > 40 so it is inserted as the right child of root node 40

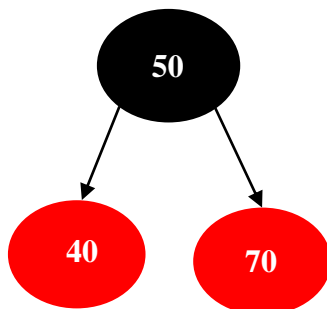


Step 3 Insert 70

// after inserting 70 property 5 violates as no red parent should have red child node.

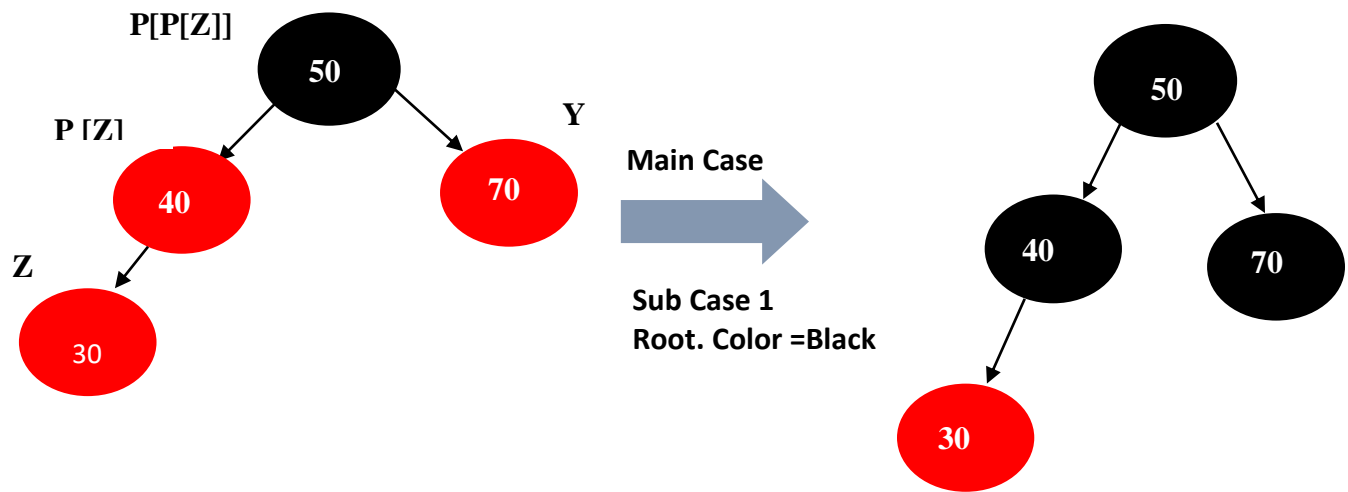


Case 3

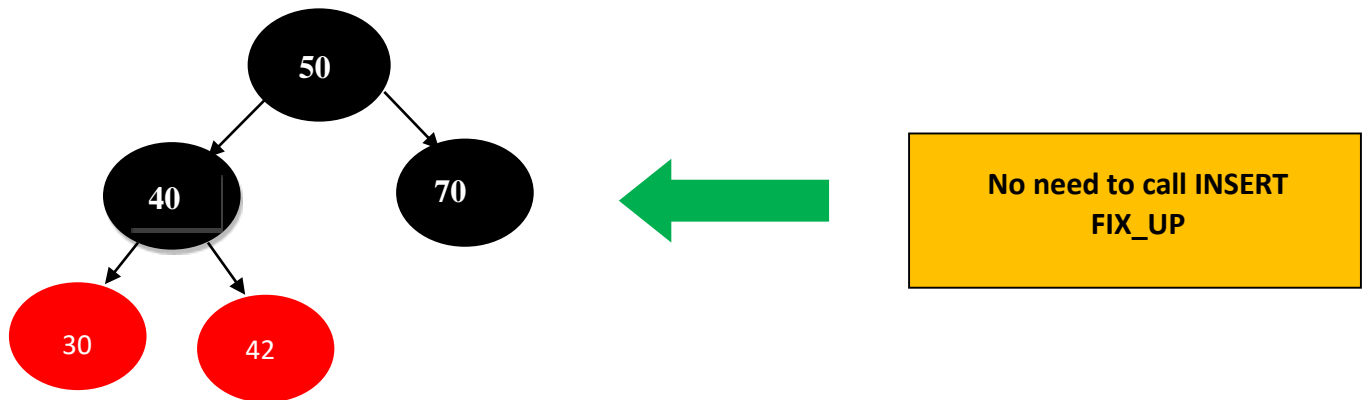


Here $P[Z] \leftarrow \text{right}[P[P[Z]]]$
 So, $y \leftarrow \text{left}[P[P[Z]]]$
 $Y \leftarrow \text{NIL (Black)}$
(Main Case 2)
 Y (uncle) is black & $Z = \text{right}[P[Z]]$
 So **(Sub case 3)**
 Color $P[Z] = \text{Black}$ & color $P[P[Z]] = \text{red}$
 Left Rotate (T, P [P[Z]])

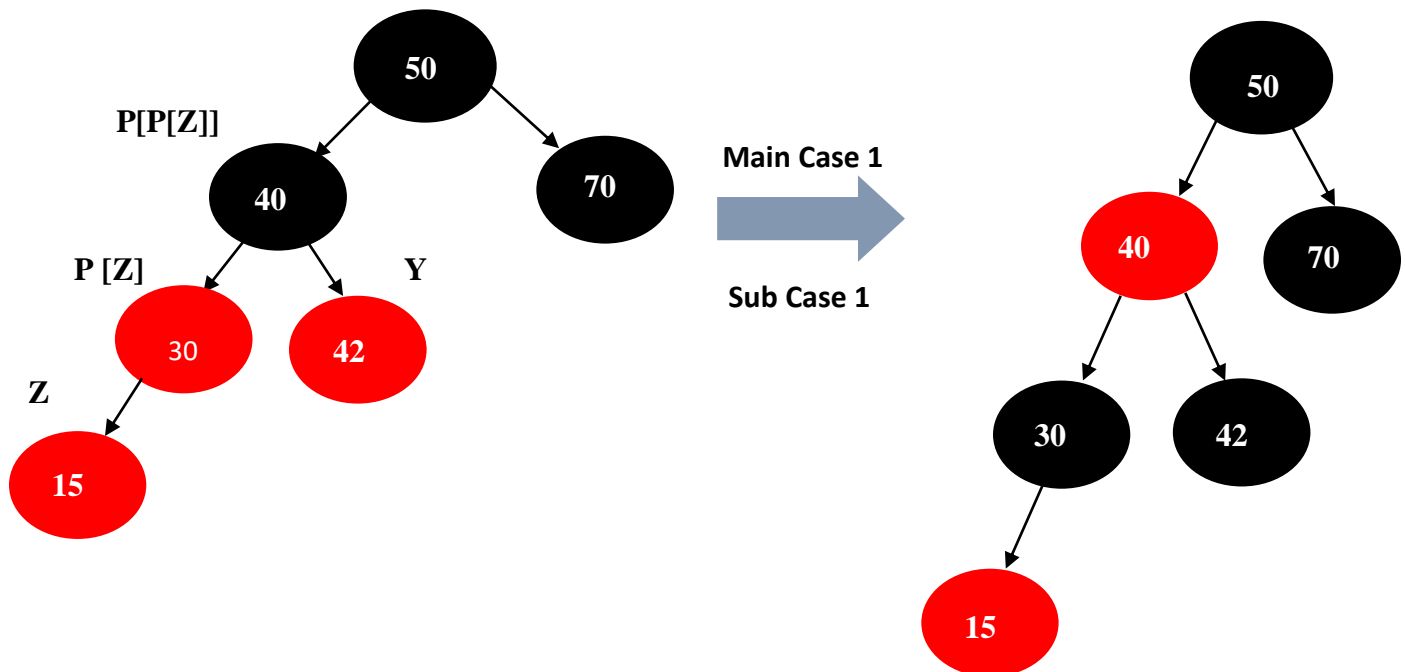
Step 4 Insert 30



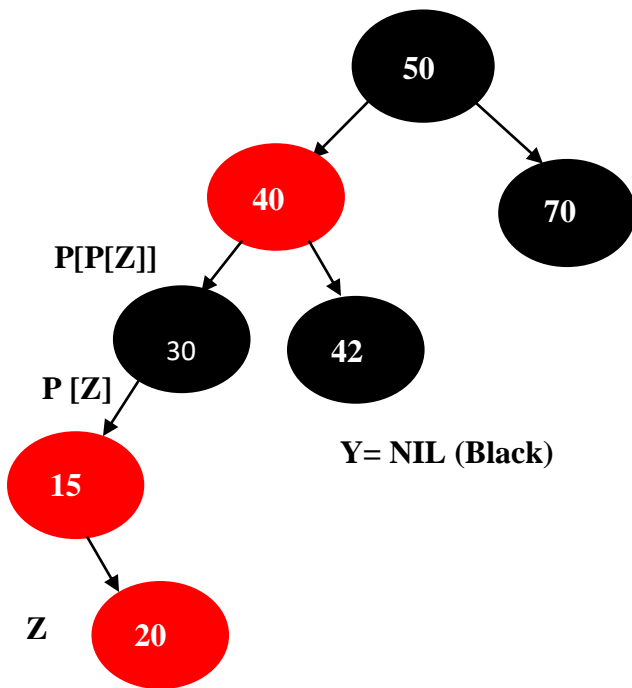
Step 5 Insert 42



Step 6 Insert 15



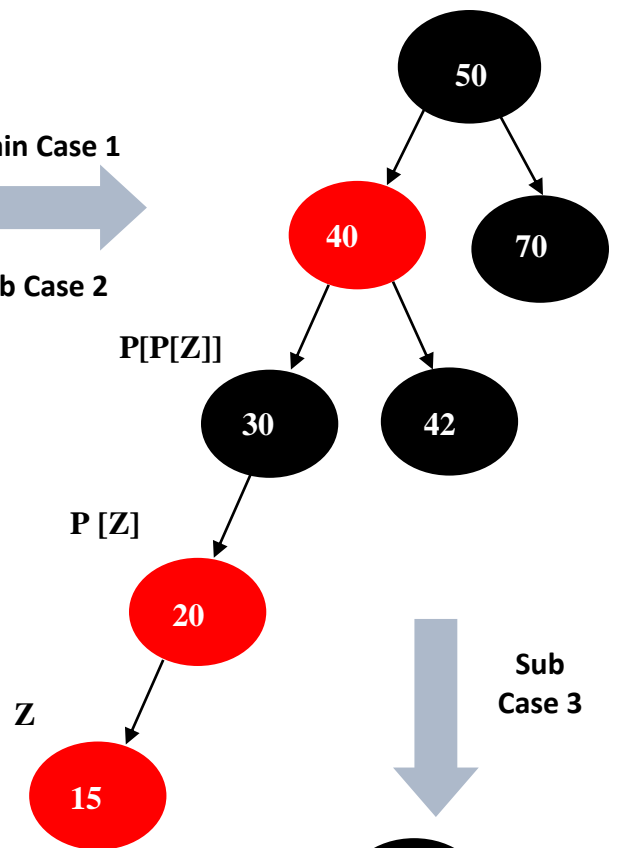
Step 7 Insert 20



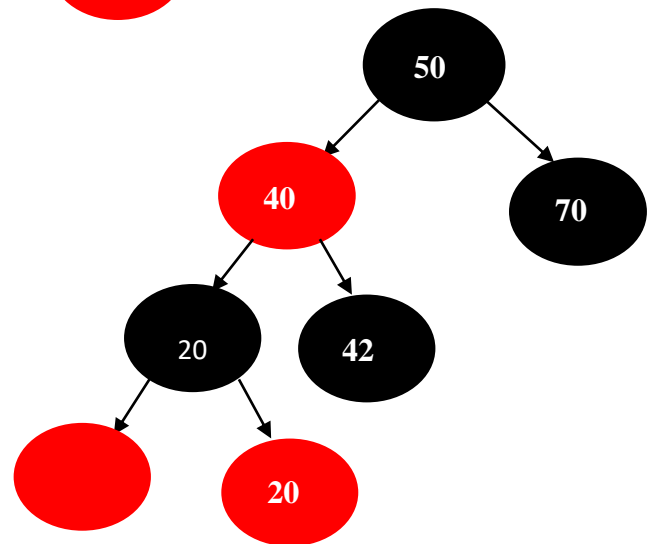
Main Case 1



Sub Case 2



Sub Case 3



Deletion in Red Black Tree

In insertion there is a problem of two consecutive red but in deletion change of black height in sub-tree as deletion of a black node may cause and reduce black height in any path from root to leaf.

Doubly Black: When a black node is deleted and replaced by a black child, the child is marked as doubly black. The main task is to convert doubly black to single black.

Note: First search the element that is required to replace the deleted element:

Types of Deletion

Type 1: If element to be deleted is in a node with only one left child.

Solution: Swap this node with the one containing the largest element in left sub-tree

Type 2: If element to be deleted is in a node with only one right child.


Solution: Swap this node with the one containing the smallest element in right sub-tree

Type 3: If element to be deleted is in a node with both a left child and right child.

Solution: Swap in any of the two ways

Note: While swapping, swap only the keys but not the colors.

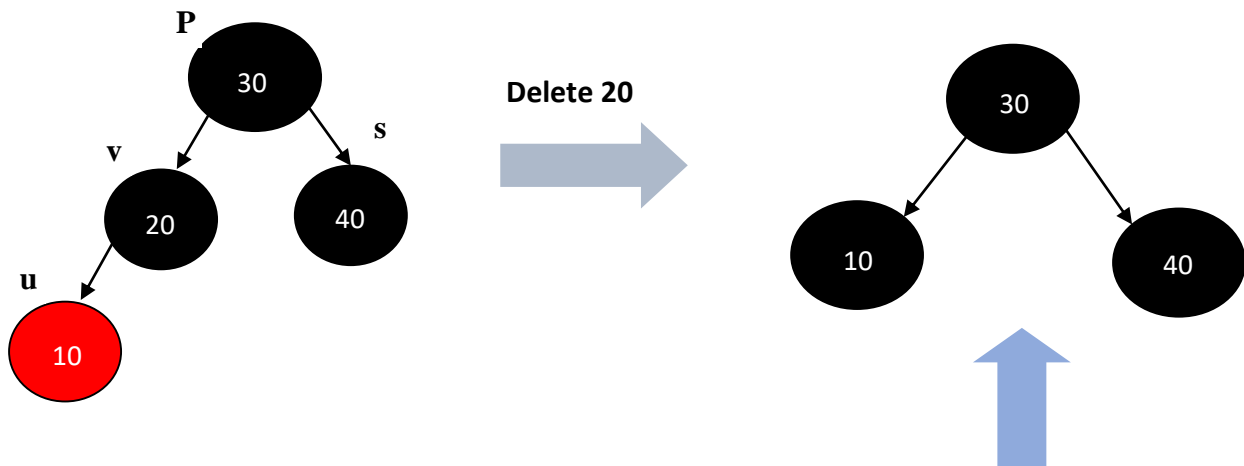
Terms used for in Red Black tree Deletion.

Terminology	Description
v	Node to be Deleted
u	Be the child of v that replaces v
s	Sibling of v
r	Child of sibling s
	Doubly Black

Important:

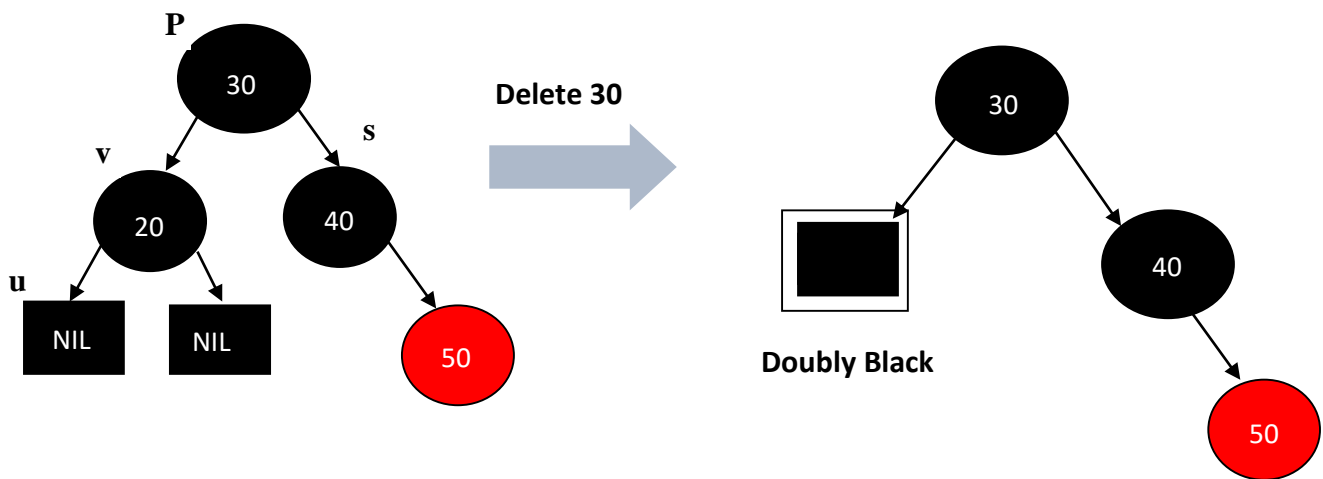
1. If v.color= Red then no issue
2. If v.color= Black then call Delete_FIXUP Procedure

Case 1: If u is red, we mark the replaced child as black.



Replaced child 10 is re-coloured as black to maintain black-height rule.

Case 2: If both u and v are black.



Case 2a: If sibling is black and atleast one of sibling's children is red then perform rotation.

Types of Rotation

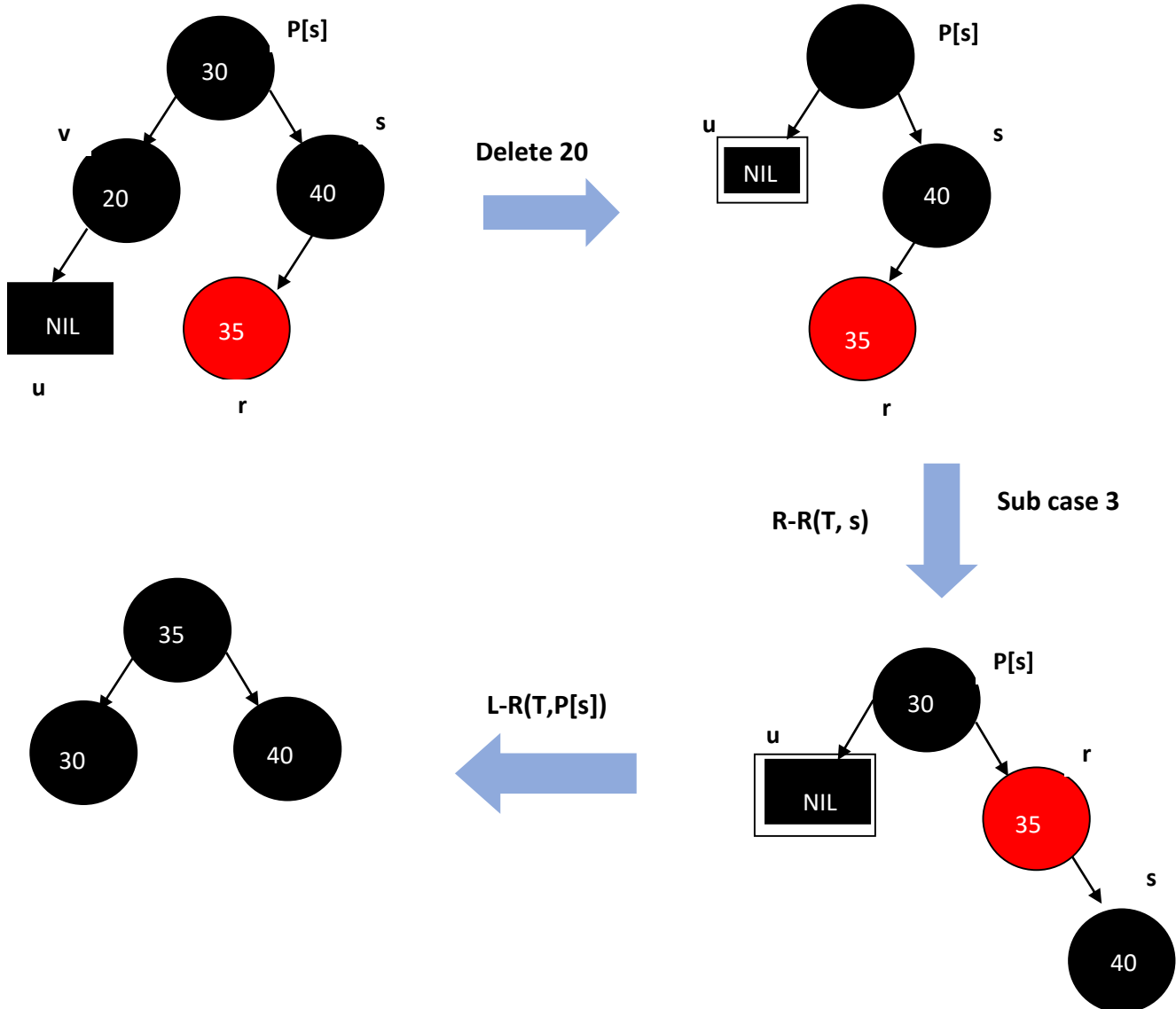
Sub case 1:
Left-Left case ->
Right rotation (T, P[s])

Sub case 2:
Left Right case ->
Left rotation then
right rotation

Sub case 3:
Right-Right case ->
Right rotation
then left rotation

Sub case 4:
Right- Right case ->
Left rotation (T, P[s])

Note: In sub case 2 and 3, first rotation is performed around siblings and second rotation is done around aren't of sibling P[s].

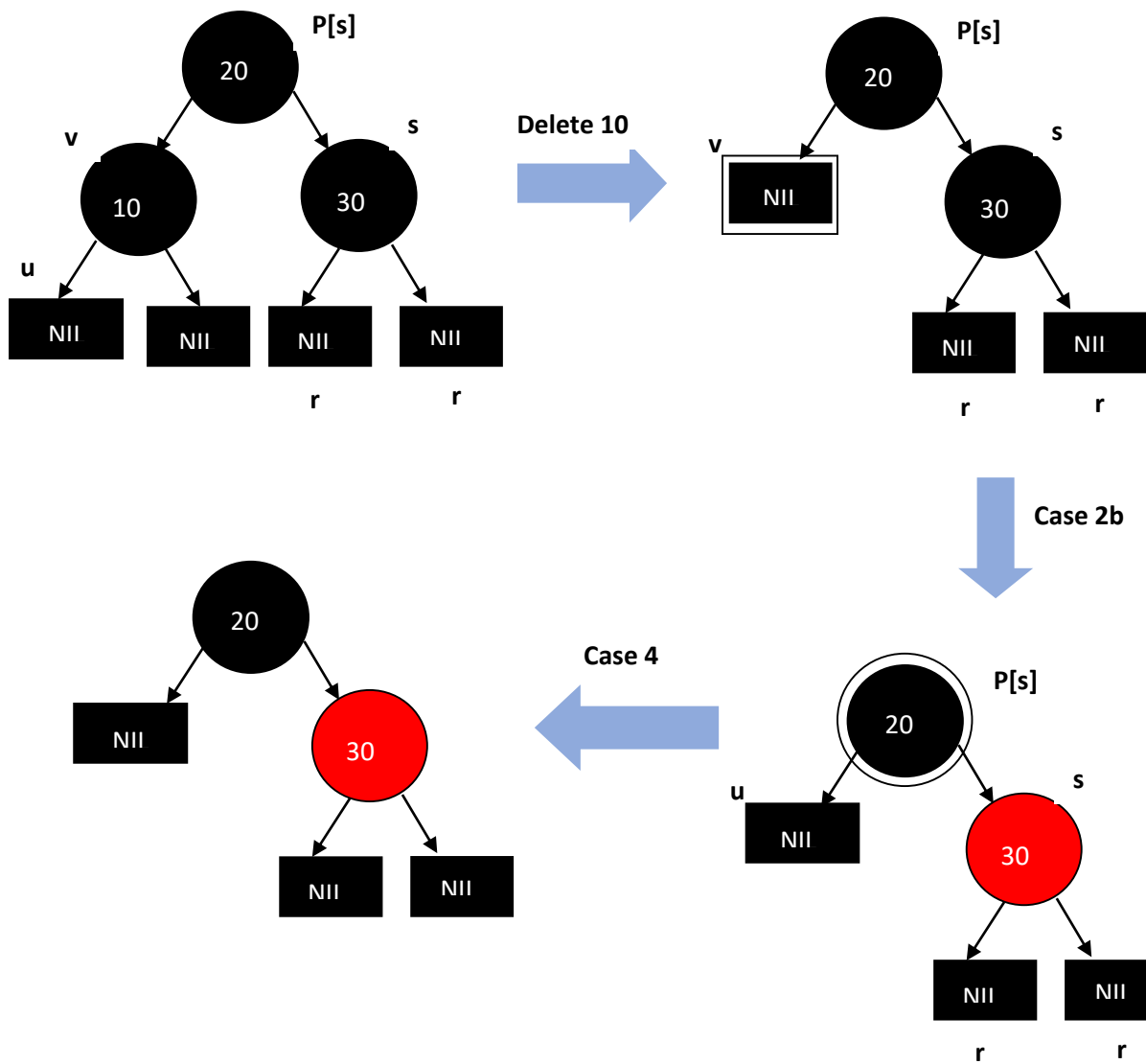


Case 2b: If sibling is black and both of it's children are also black.

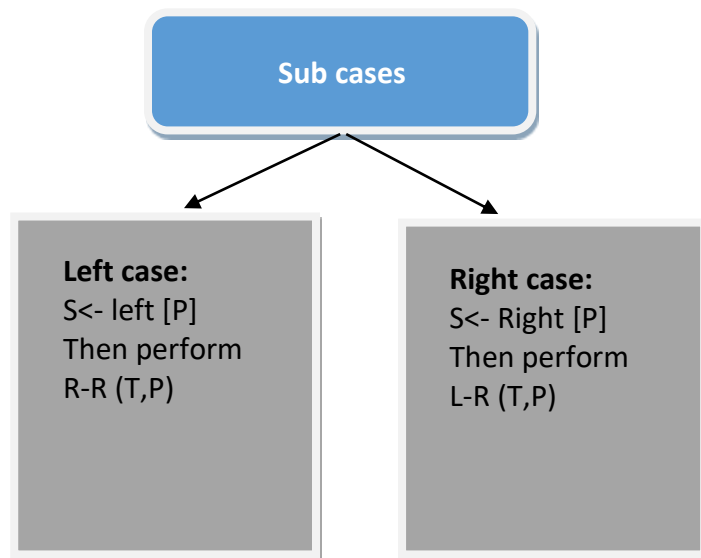
- Re-color sibling as red
- Recolor parent as doubly black and call DELETE_FIX_UP on parent.

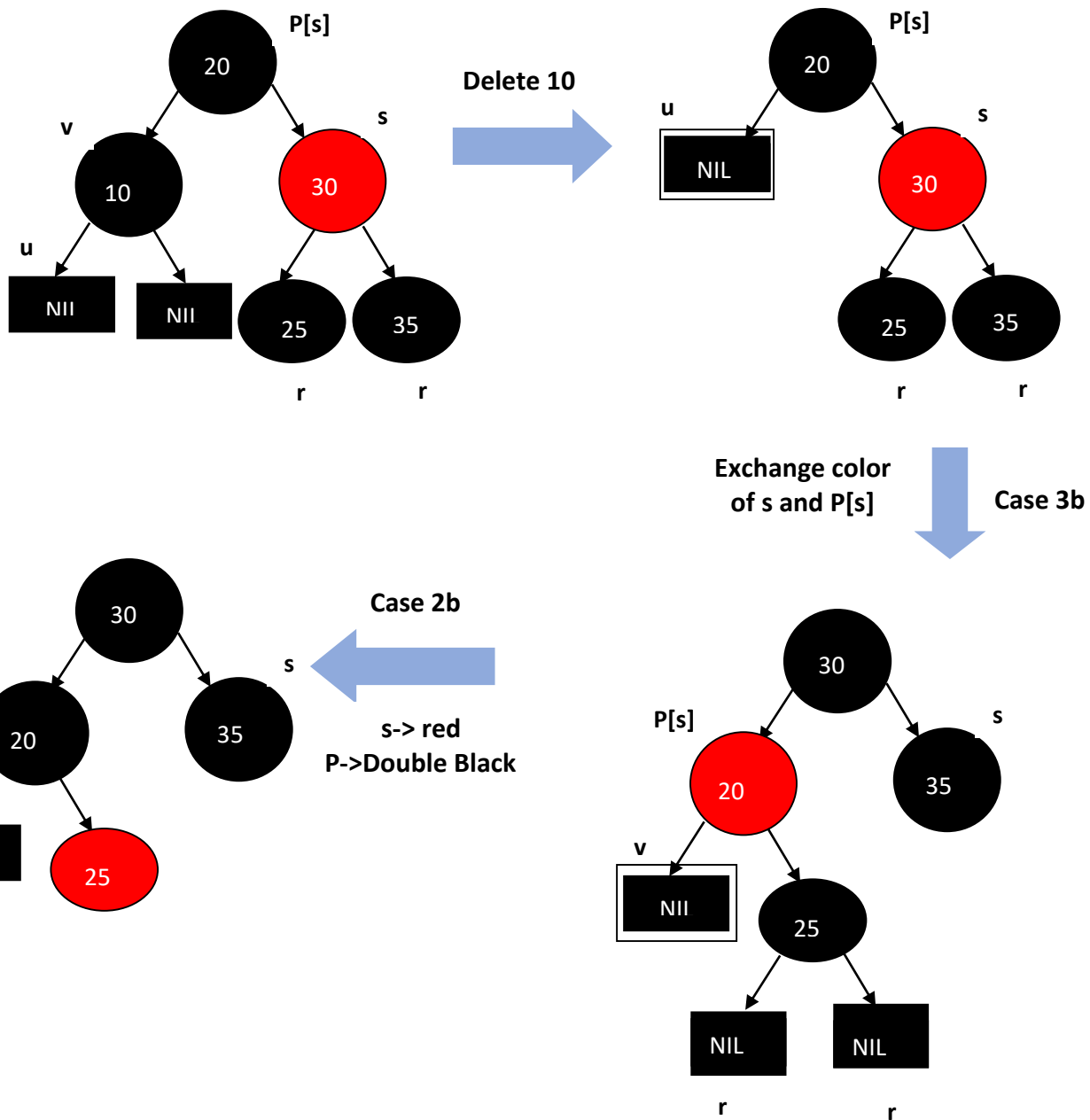
Note:

- B+B-> Double Black (Two black nodes)
- R+ Double Black-> (Single black node)



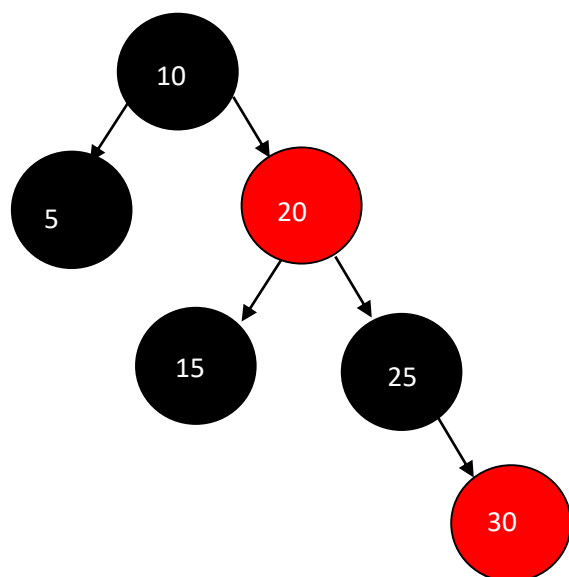
Case 3: When sibling is red



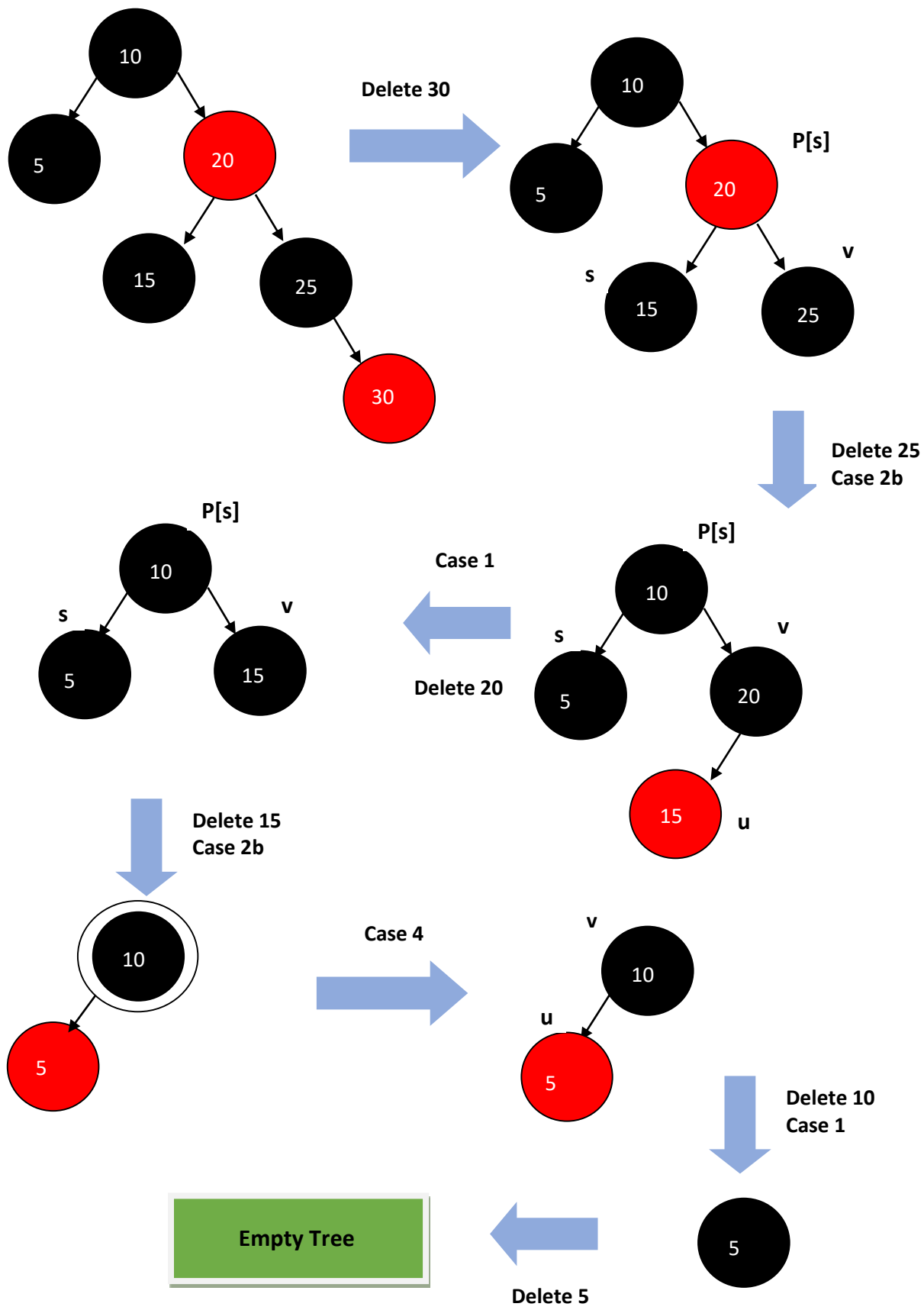


Case 4: If root is double black, make it single black.

Ques Perform deletion operation red black tree in order 30, 25, 20, 15, 10, 5.



Solution



Comparison between Red –Black Tree and AVL Tree

Parameter	Red Black Tree	AVL Tree
Searching	Red Black tree does not provide efficient searching as Red Black Trees are roughly balanced.	AVL trees provide efficient searching as it is strictly balanced tree.
Insertion and Deletion	Insertion and Deletion are easier in Red Black tree as it requires fewer rotations to balance the tree.	Insertion and Deletion are complex in AVL tree as it requires multiple rotations to balance the tree.
Color of the node	In the Red-Black tree, the color of the node is either Red or Black.	In the case of AVL trees, there is no color of the node.
Balance factor	It does not contain any balance factor. It stores only one bit of information that denotes either Red or Black color of the node.	Each node has a balance factor in AVL tree whose value can be 1, 0, or -1. It requires extra space to store the balance factor per node.
Strictly balanced	Red-black trees are not strictly balanced.	AVL trees are strictly balanced, i.e., the left subtree's height and the height of the right subtree differ by at most 1.

B Tree

INTRODUCTION

A B-tree is a self-balancing tree where all the leaf nodes are at the same level which allows for efficient searching, insertion and deletion of records. Because of all the leaf nodes being on the same level, the access time of data is fixed regardless of the size of the data set. B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least $m/2$ children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have $m/2$ number of nodes.

Characteristics of B-Tree

B-trees have several important characteristics that make them useful for storing and retrieving large amounts of data efficiently. Some of the key characteristics of B-trees are:

- **Balanced:** B-trees are balanced, meaning that all leaf nodes are at the same level. This ensures that the time required to access data in the tree remains constant, regardless of the size of the data set.
- **Self-balancing:** B-trees are self-balancing, which means that as new data is inserted or old data is deleted, the tree automatically adjusts to maintain its balance.
- **Multiple keys per node:** B-trees allow multiple keys to be stored in each node. This allows for efficient use of memory and reduces the height of the tree, which in turn reduces the number of disk accesses required to retrieve data.
- **Ordered:** B-trees maintain the order of the keys, which makes searching and range queries efficient.
- **Efficient for large data sets:** B-trees are particularly useful for storing and retrieving large amounts of data, as they minimize the number of disk accesses required to find a particular piece of data.

Need of a B Tree in a Data Structure

- The need for B-tree emerged as the demand for faster access to physical storage media such as a hard disk grew. With a bigger capacity, backup storage devices were slower. There was a demand for data structures that reduced the number of disc accesses.
- A binary search tree, an AVL tree, a red-black tree, and other data structures can only store one key in each node. When you need to store a significant number of keys, the height of such trees grows quite vast, and the time it takes to access them grows.
- B-tree, on the other hand, can store multiple keys inside a single node and has several child nodes. It considerably reduces the height, allowing for speedier disk access.

Application of B-Tree:

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process. Searching an un-indexed and unsorted database containing n key values needs $O(n)$ running time in worst case. However, if we use B Tree to index this database, it will be searched in $O(\log n)$ time in worst case.

The limitations of traditional binary search trees can be frustrating. Meet the B-Tree, the multi-talented data structure that can handle massive amounts of data with ease. When it comes to storing and searching large amounts of data, traditional binary search trees can become impractical due to their poor performance and high memory usage. B-Trees, also known as B-Tree or Balanced Tree, are a type of self-balancing tree that was specifically designed to overcome these limitations.

Unlike traditional binary search trees, B-Trees are characterized by the large number of keys that

they can store in a single node, which is why they are also known as “large key” trees. Each node in a B-Tree can contain multiple keys, which allows the tree to have a larger branching factor and thus a shallower height. This shallow height leads to less disk I/O, which results in faster search and insertion operations. B-Trees are particularly well suited for storage systems that have slow, bulky data access such as hard drives, flash memory, and CD-ROMs.

B-Trees maintain balance by ensuring that each node has a minimum number of keys, so the tree is always balanced. This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always $O(\log n)$, regardless of the initial shape of the tree.

B-trees are commonly used in applications where large amounts of data need to be stored and retrieved efficiently. Some of the specific applications of B-trees include:

- Databases: B-trees are widely used in databases to store indexes that allow for efficient searching and retrieval of data.
- File systems: B-trees are used in file systems to organize and store files efficiently.
- Operating systems: B-trees are used in operating systems to manage memory efficiently.
- Network routers: B-trees are used in network routers to efficiently route packets through the network.
- DNS servers: B-trees are used in Domain Name System (DNS) servers to store and retrieve information about domain names.
- Compiler symbol tables: B-trees are used in compilers to store symbol tables that allow for efficient compilation of code.

Advantages of B-Tree:

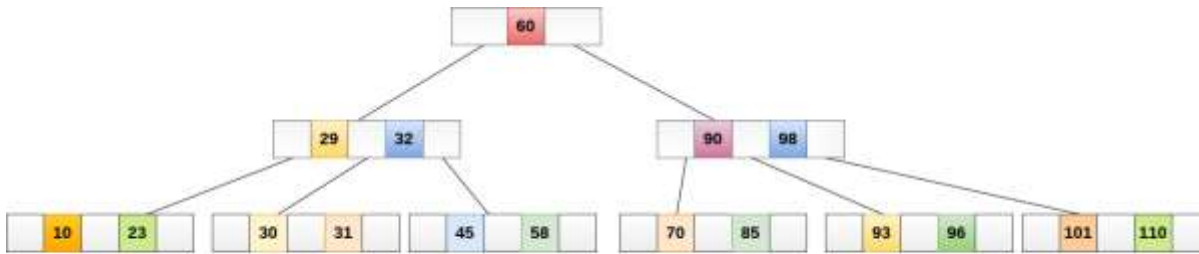
B-trees have several advantages over other data structures for storing and retrieving large amounts of data. Some of the key advantages of B-trees include:

- Sequential Traversing: As the keys are kept in sorted order, the tree can be traversed sequentially.
- Minimize disk reads: It is a hierarchical structure and thus minimizes disk reads.
- Partially full blocks: The B-tree has partially full blocks which speed up insertion and deletion.

Disadvantages of B-Tree:

- Complexity: B-trees can be complex to implement and can require a significant amount of programming effort to create and maintain.
- Overhead: B-trees can have significant overhead, both in terms of memory usage and processing time. This is because B-trees require additional metadata to maintain the tree structure and balance.
- Not optimal for small data sets: B-trees are most effective for storing and retrieving large amounts of data. For small data sets, other data structures may be more efficient.
- Limited branching factor: The branching factor of a B-tree determines the number of child nodes that each node can have. B-trees typically have a fixed branching factor, which can limit their performance for certain types of data.

A B tree of order 4 is shown in the following image.



While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

Operations

1.Searching:

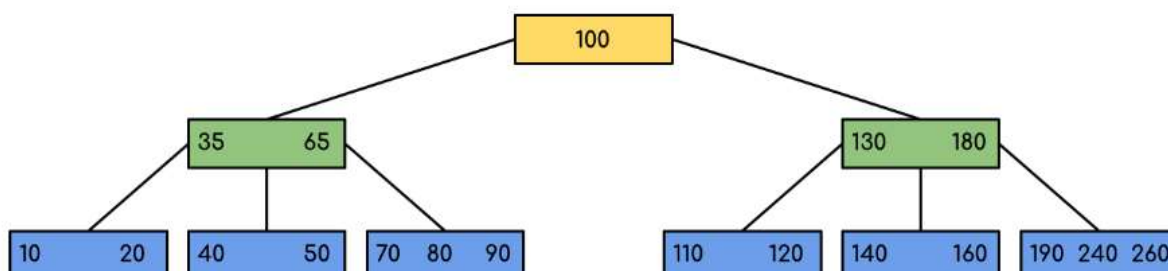
Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

1. Compare item 49 with root node 78. since $49 < 78$ hence, move to its left sub-tree.
2. Since, $40 < 49 < 56$, traverse right sub-tree of 40.
3. $49 > 45$, move to right. Compare 49.
4. match found, return.

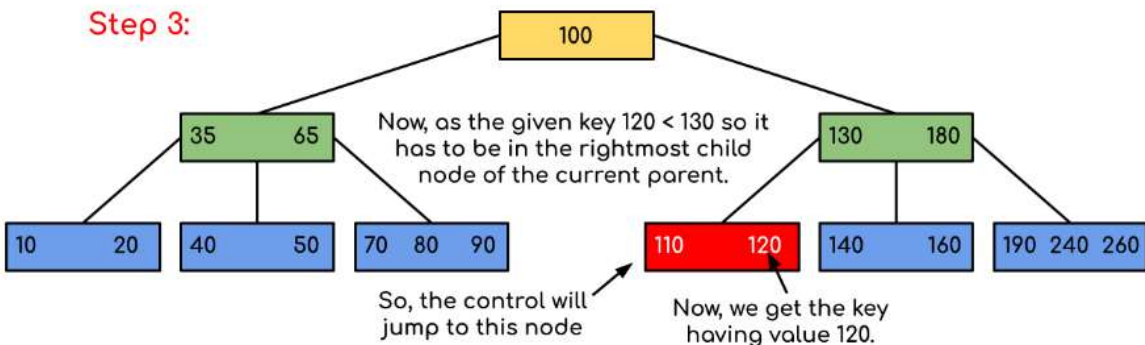
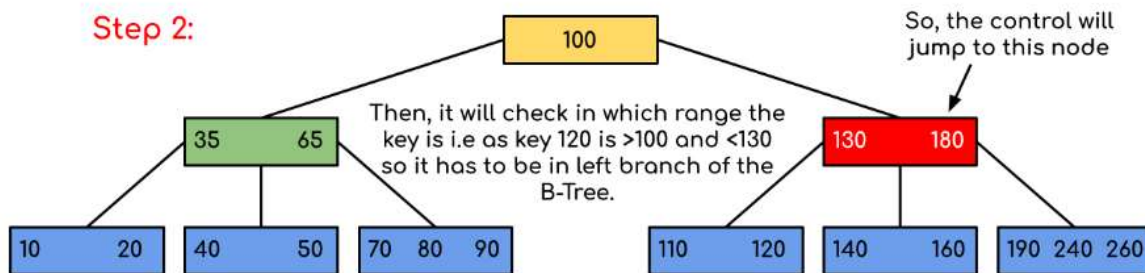
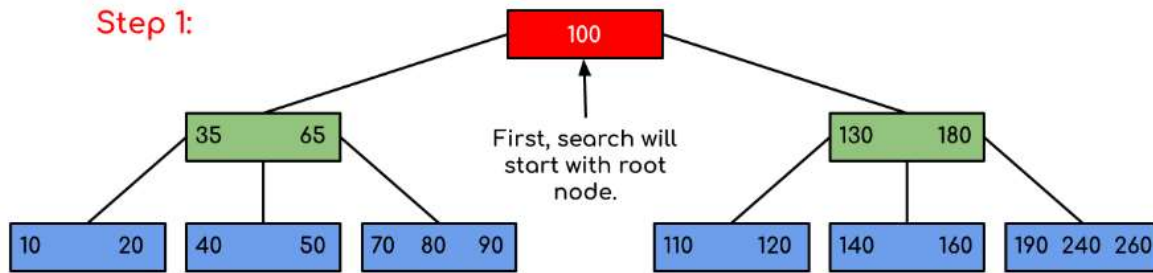
Searching in a B tree depends upon the height of the tree. The search algorithm takes $O(\log n)$ time to search any element in a B tree.

Examples:

Input: Search 120 in the given B-Tree.



Solution:



In this example, we can see that our search was reduced by just limiting the chances where the key containing the value could be present. Similarly if within the above example we've to look for 180, then the control will stop at step 2 because the program will find that the key 180 is present within the current node. And similarly, if it's to seek out 90 then as $90 < 100$ so it'll go to the left subtree automatically, and therefore the control flow will go similarly as shown within the above example.

2. B-Tree Insertion

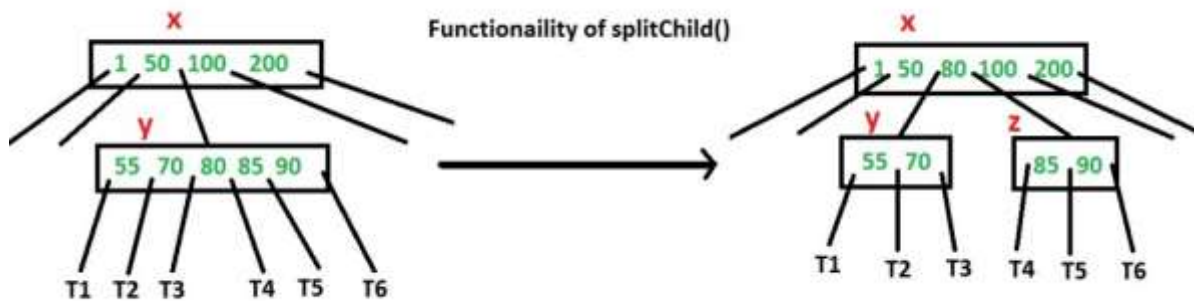
There are two conventions to define a B-Tree, one is to define by minimum degree, second is to define by order.

A new key is always inserted at the leaf node. Let the key to be inserted be k . Like BST, we start from the root and traverse down till we reach a leaf node. Once we reach a leaf node, we insert the key in that leaf node. Unlike BSTs, we have a predefined range on the number of keys that a node can contain. So before inserting a key to the node, we make sure that the node has extra space.

How to make sure that a node has space available for a key before the key is inserted?

We use an operation called `splitChild()` that is used to split a child of a node. See the following diagram

to understand split. In the following diagram, child y of x is being split into two nodes y and z. Note that the splitChild operation moves a key up and this is the reason B-Trees grow up, unlike BSTs which grow down.



As discussed above, to insert a new key, we go down from root to leaf. Before traversing down to a node, we first check if the node is full. If the node is full, we split it to create space. Following is the complete algorithm.

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contains less than $m-1$ keys then insert the element in the increasing order.
3. Else, if the leaf node contains $m-1$ keys, then follow the following steps.
 - Insert the new element in the increasing order of elements.
 - Split the node into the two nodes at the median.
 - Push the median element upto its parent node.
 - If the parent node also contain $m-1$ number of keys, then split it too by following the same steps.

Insertion

- 1) Initialize x as root.
- 2) While x is not leaf, do following
 - a) Find the child of x that is going to be traversed next. Let the child be y.
 - b) If y is not full, change x to point to y.
 - c) If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as the first part of y. Else second part of y. When we split y, we move a key from y to its parent x.
- 3) The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

EXAMPLE:

Let us understand the algorithm with an example tree of minimum degree 't' as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree. Initially root is NULL. Let us first insert 10.

Insert 10

10

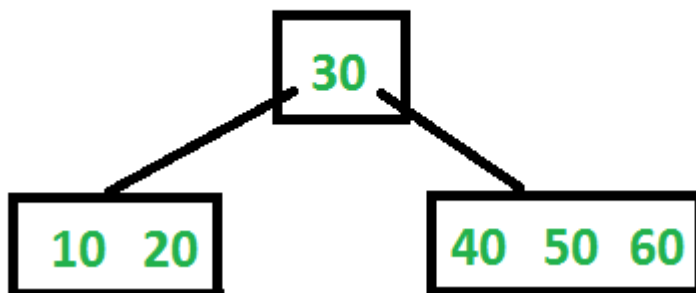
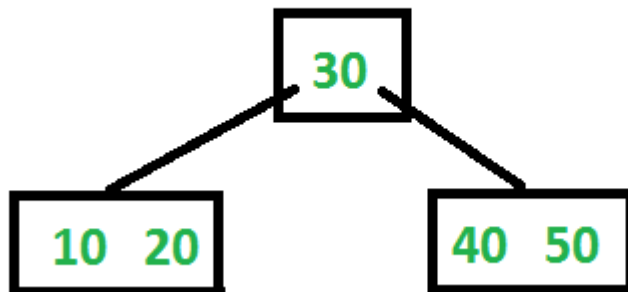
Let us now insert 20, 30, 40 and 50. They all will be inserted in root because the maximum number of keys a node can accommodate is $2^t - 1$ which is 5.

Insert 20, 30, 40 and 50

10 20 30 40 50

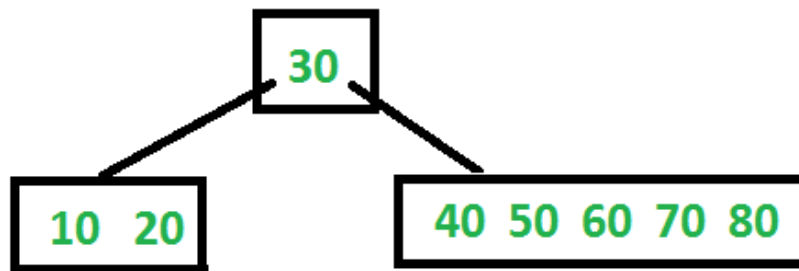
Let us now insert 60. Since root node is full, it will first split into two, then 60 will be inserted into the appropriate child.

Insert 60



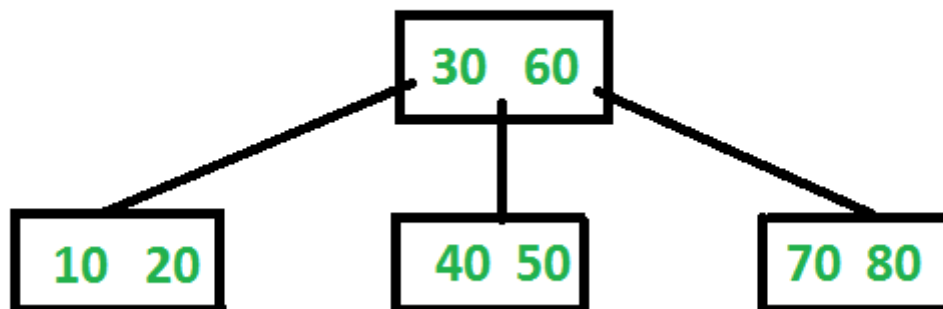
Let us now insert 70 and 80. These new keys will be inserted into the appropriate leaf without any split.

Insert 70 and 80



Let us now insert 90. This insertion will cause a split. The middle key will go up to the parent.

Insert 90

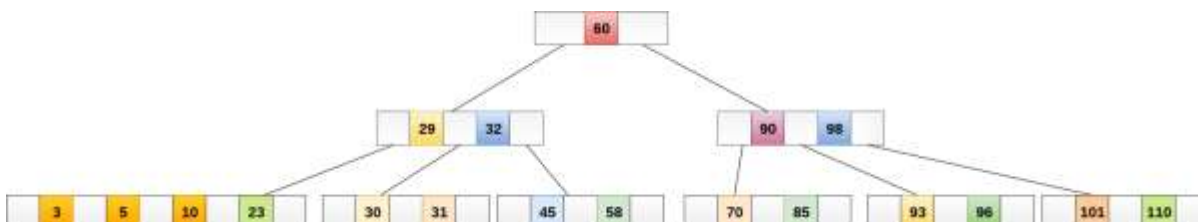


The B-tree is a data structure that is similar to a binary search tree, but allows multiple keys per node and has a higher fanout. This allows the B-tree to store a large amount of data in an efficient manner, and it is commonly used in database and file systems.

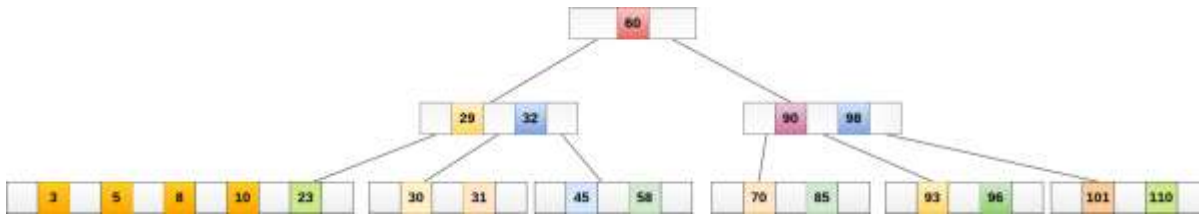
The B-tree is a balanced tree, which means that all paths from the root to a leaf have the same length. The tree has a minimum degree t , which is the minimum number of keys in a non-root node. Each node can have at most $2t-1$ keys and $2t$ children. The root can have at least one key and at most $2t-1$ keys. All non-root nodes have at least $t-1$ keys and at most $2t-1$ keys.

Example:2:

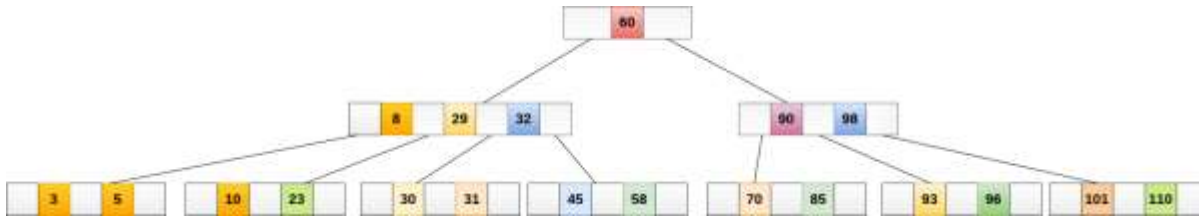
Insert the node 8 into the B Tree of order 5 shown in the following image.



8 will be inserted to the right of 5, therefore insert 8.



The node, now contain 5 keys which is greater than $(5 - 1 = 4)$ keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.



3.Deletion

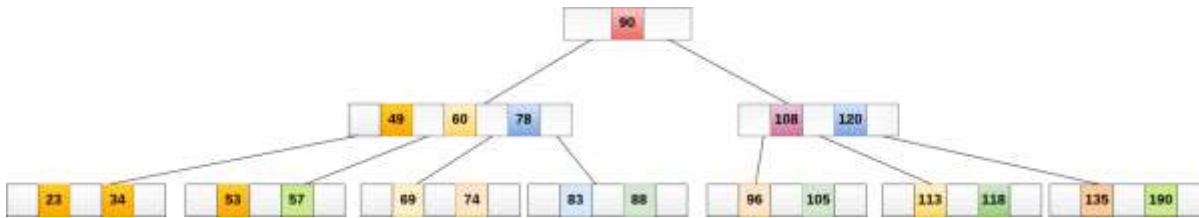
Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than $m/2$ keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain $m/2$ keys then complete the keys by taking the element from right or left sibling.
 - If the left sibling contains more than $m/2$ elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
 - If the right sibling contains more than $m/2$ elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
4. If neither of the sibling contain more than $m/2$ elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than $m/2$ nodes then, apply the above process on the parent too.

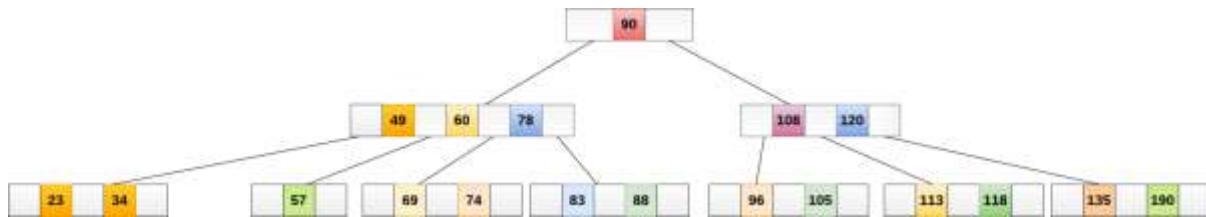
If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example 1

Delete the node 53 from the B Tree of order 5 shown in the following figure.

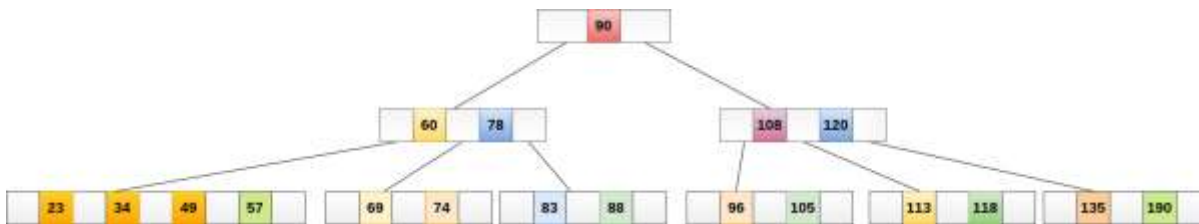


53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows.



Time Complexity of B-Tree:

-

Sr. No.	Algorithm	Time Complexity
1.	Search	$O(\log n)$
2.	Insert	$O(\log n)$
3.	Delete	$O(\log n)$

Note: “n” is the total number of elements in the B-tree

GATE QUESTION:

Q. The smallest number of keys that will force a B-Tree of order 3 to have a height of 3 are

Solution:

Consider that the root node is at height 0.

Since, order = 3

Therefore, minimum keys at root, internal node or leaf = 1

And, node pointers = 2

For height = 3,

The number of keys at height 0 is = 1 // root node

The number of keys at height 1 is = 2

Number of keys at height 2 = 4

Number of keys at height 3 = 8

Therefore, total keys = $8 + 4 + 2 + 1 = 15$

Q. the order of B-tree is 32 and the B-tree is 63% full. What will be the number of <key,data pointer> entries the 2 level B tree holds?

SOLUTION:

Here, order = 32 and Assuming that B tree node is 63% full.

so $P = \text{ceil} (0.63 * 32) = 21$

(If we take order $P = 20$, then utilization will be 62.5 % which is not allowed.)

so at level 0 (Root) : 20 keys

level 1 : $21 * 20 = 420$ keys

level 2 : $21 * 21 * 20 = 8820$ keys

Hence, total = $20 + 420 + 8820 = 9260$ keys

Ggggggghjgjhbxjkhjm

Binomial Heaps

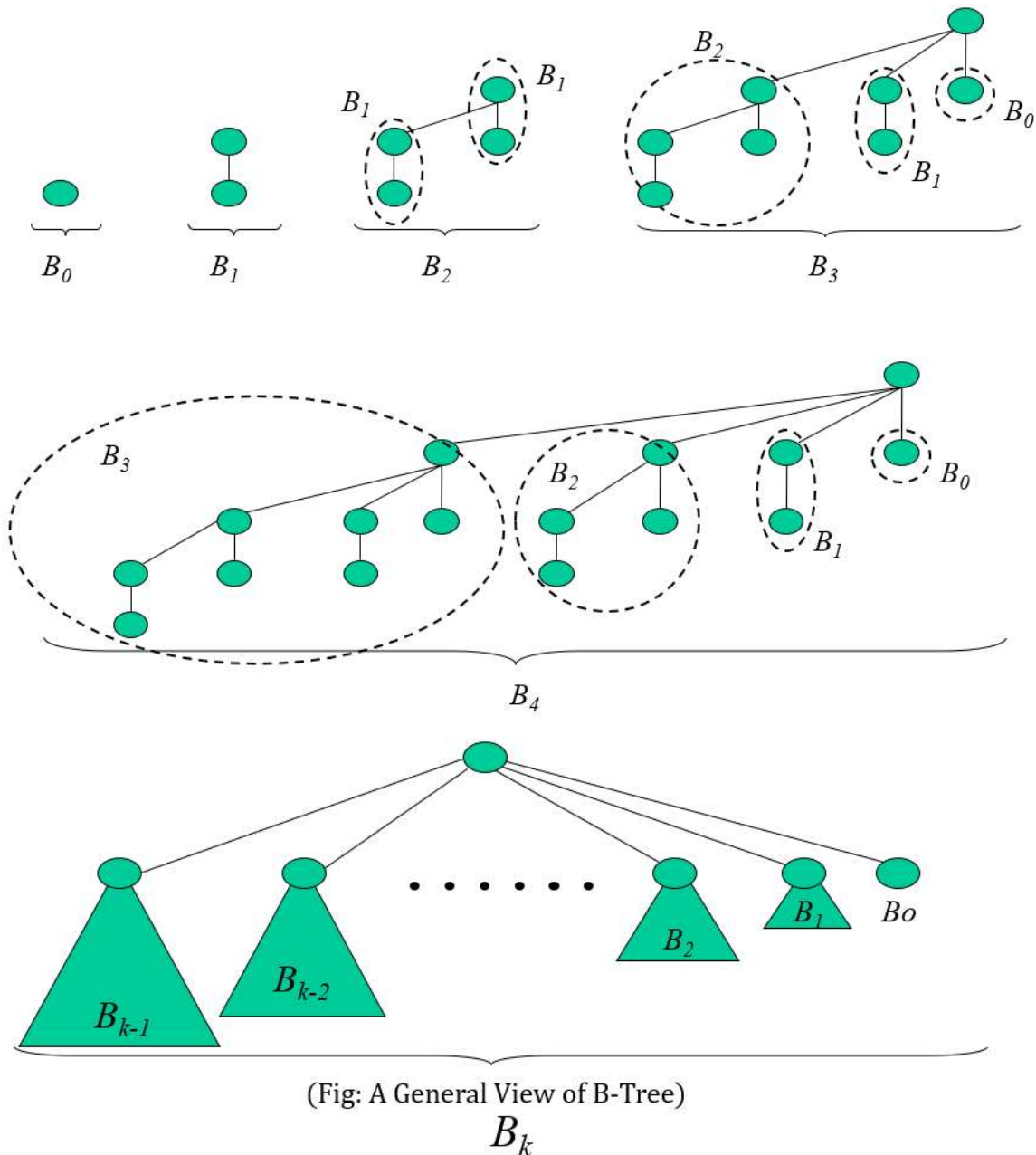
What is Binomial Heap?

Binomial heap was introduced in 1978 by Jean Vullemin. Jean Vullemin is a professor in mathematics and computer science. The other name of Binomial Heap is Mergeable heaps.

A binomial heap is a collection of binomial trees.

What is Binomial tree?

Binomial tree B_k is an ordered tree defined recursively. The binomial tree B_0 has one node. The binomial tree B_k consists of two binomial trees B_{k-1} and they are connected such that the root of one tree is the leftmost child of the other.



Binomial tree properties:

- B_k has 2^k nodes
- B_k has height k
- There are exactly $\binom{k}{i}$ nodes at depth i for $i=0, 1, 2, \dots, k$.

For Example:

Lets check in B_4 and depth 2 (i.e. $k = 4$ and $i = 2$)

$$\binom{k}{i} = \frac{k!}{i!(k-i)!} = \frac{4!}{2!(4-2)!} = \frac{4 \times 3 \times 2 \times 1}{2 \times 1 \times 2 \times 1} = 6$$

Hence 6 numbers of nodes are available in depth 2 of B_4 .

- The root has degree k which is greater than other node in the tree. Each of the root's child is the root of a subtree B_i .

Binomial Heap Properties

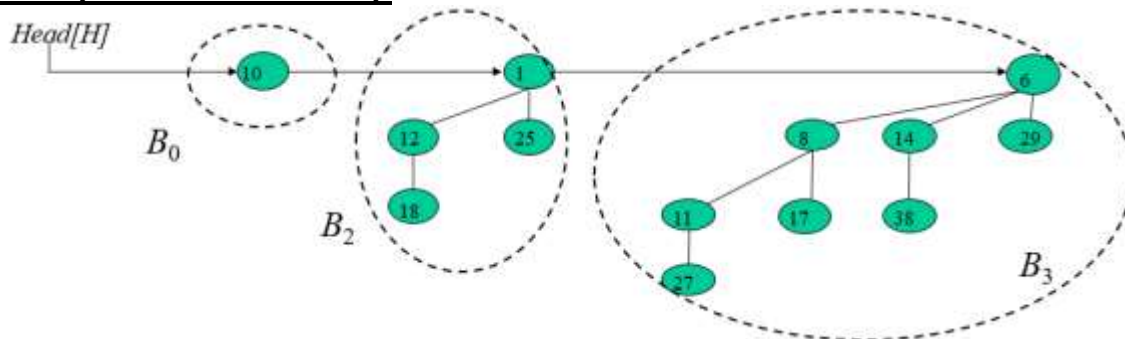
A Binomial Heap H is a set of binomial trees that satisfies the following properties:

- Each binomial tree in H obeys the **min heap property**.

(i.e. key of a node is greater or equal to the key of its parent. Hence the root has the smallest key in the tree).

- For any non negative integer k , there is at most one binomial tree whose root has degree k . (e.g. it implies that an n node Binomial heap H consists of at most $\lfloor \log n \rfloor + 1$ binomial Tree.
- The binomial trees in the binomial heap are arranged in increasing order of degree

Example of binomial Heap



Here $n=13$

So $\lfloor \log n \rfloor + 1 = \lfloor \log 13 \rfloor + 1 = 3 + 1 = 4$ (So at most 4). The Above Figure of Binomial Heap consists of B_0 , B_2 and B_3

Representation of Binomial Heap

- Each binomial tree within a binomial heap is stored in the left-child, right-sibling representation.
- Each node X contains POINTERS
 - $p[x]$ or $x \rightarrow parent$ to its parent
 - $key[x]$ or $x \rightarrow key$ to its key value
 - $child[x]$ or $x \rightarrow child$ to its leftmost child
 - $sibling[x]$ or $x \rightarrow sibling$ to its immediately right sibling
 - $degree[x]$ or $x \rightarrow degree$ to its degree value (i.e. denotes the number of children of X)

Operations on Binomial Heap

Binomial Heap support the following operations:

- MAKE-HEAP ()** creates and returns a new heap containing no elements.
- INSERT(H, x)** inserts node x , whose *key* field has already been filled in, into heap H .
- MINIMUM(H)** returns a pointer to the node in heap H whose key is minimum.
- EXTRACT-MIN(H)** deletes the node from heap H whose key is minimum, returning a pointer to the node.
- UNION (H1, H2)** creates and returns a new heap that contains all the nodes of heaps $H1$ and $H2$. Heaps $H1$ and $H2$ are "destroyed" by this operation.
- DECREASE-KEY(H, x, k)** assigns to node x within heap H the new key value k , which is assumed to be no greater than its current key value.
- DELETE (H, x)** deletes node x from heap H

1. MAKE-BINOMIAL-HEAP

This is the process of creating and returning a new heap containing no elements.

To make an empty binomial heap, the MAKE-BINOMIAL-HEAP procedure simply allocates and returns an object H , where $head[H] = NIL$.

The running time is $\Theta(1)$.

2. BINOMIAL-HEAP-INSERT

For inserting a new node x , Create a new heap H' and set $\text{Head}(H')$ to the new node x . Now, we can easily perform union of new Heap H' with the existing heap H .

Time complexity for this operation is:-

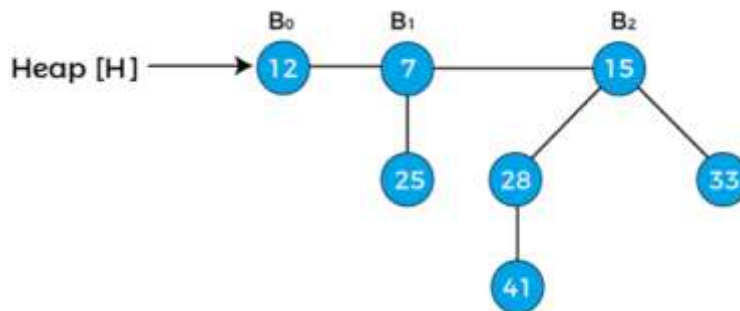
- For creating new heap = $O(1)$
- For union of two heaps = $O(\log n)$
- So, for inserting a new node = $O(1) + O(\log n) = O(\log n)$.

Binomial-Heap-Insert(H, x)

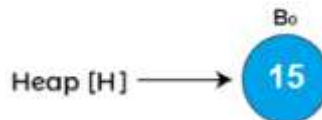
1. $H' \leftarrow \text{Make-Binomial-Heap}()$
2. $p[x] \leftarrow \text{NIL}$
3. $\text{child}[x] \leftarrow \text{NIL}$
4. $\text{sibling}[x] \leftarrow \text{NIL}$
5. $\text{degree}[x] \leftarrow 0$
6. $\text{Head}[H'] \leftarrow x$
7. $H \leftarrow \text{Binomial-Heap-Union}(H, H')$

Example demonstrating Insert operation-

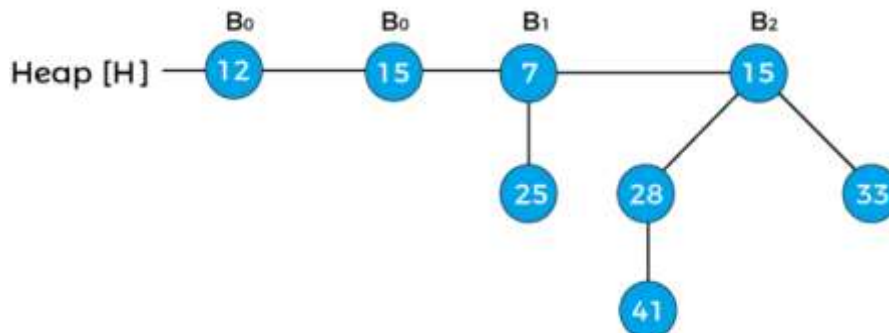
Inserting node 15, in given Binomial Heap.



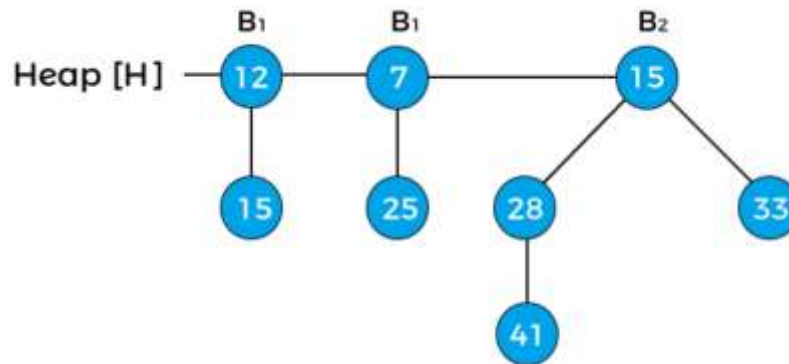
First step is to create a heap with key 15. First, we have to combine both of the heaps.



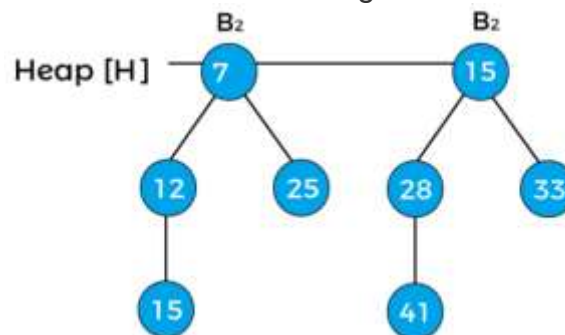
As both node 12 and node 15 are of degree 0, so node 15 is attached to node 12 as shown below.



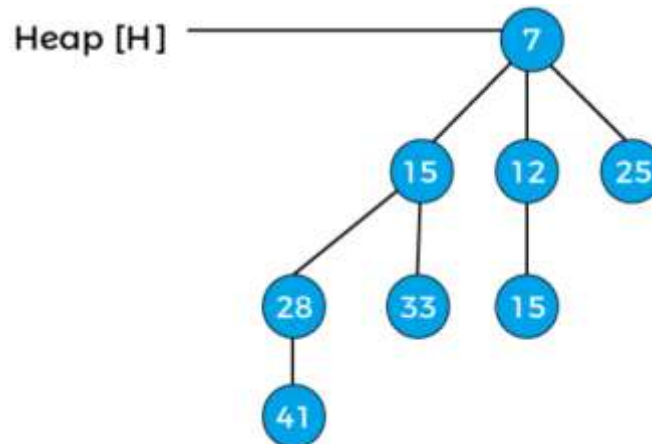
Now, assign x to B_0 with value 12, $\text{next}(x)$ to B_0 with value 15, and assign $\text{sibling}(\text{next}(x))$ to B_1 with value 7. As the degree of x and $\text{next}(x)$ is equal. The key value of x is smaller than the key value of $\text{next}(x)$, so $\text{next}(x)$ is removed and attached to the x . It is shown in the below image –



Now, x points to node 12 with degree B_1 , $\text{next}(x)$ to node 7 with degree B_1 , and $\text{sibling}(\text{next}(x))$ points to node 15 with degree B_2 . The degree of x is equal to the degree of $\text{next}(x)$ but not equal to the degree of $\text{sibling}(\text{next}(x))$. The key value of x is greater than the key value of $\text{next}(x)$; therefore, x is removed and attached to the $\text{next}(x)$ as shown in the below image -



Now, x points to node 7, and $\text{next}(x)$ points to node 15. The degree of both x and $\text{next}(x)$ is B_2 , and the key value of x is less than the key value of $\text{next}(x)$, so $\text{next}(x)$ will be removed and attached to x as shown in the below image -



Now, the degree of the above heap is B_3 , and it is the final binomial heap after inserting node 15.

3. BINOMIAL-HEAP-MINIMUM

The procedure **BINOMIAL-HEAP-MINIMUM()** returns a pointer to the node with the minimum key in an n -node binomial heap H . This implementation assumes that there are no keys with value ∞ .

Since the binomial heap is a min-heap-order, the minimum key of each binomial tree must be at the root. This operation checks all the roots to find the minimum key.

Pseudocode: This implementation assumes that there are no keys with value ∞ .

BINOMIAL-HEAP-MINIMUM(H)

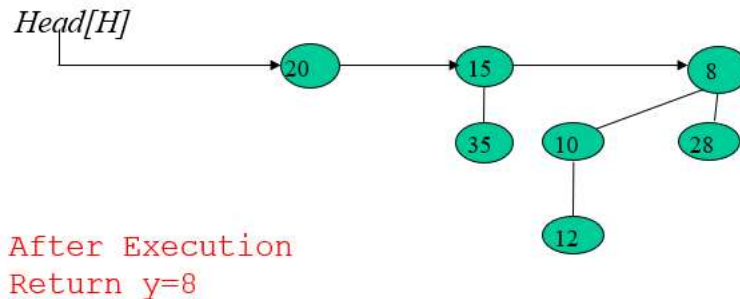
1	$y \leftarrow \text{NIL}$
2	$x \leftarrow \text{head}[H]$
3	$\text{min} \leftarrow \infty$
4	while $x \neq \text{NIL}$

```

5  do if key[x] < min
6  then min ← key[x]
7  y ← x
8  x ← sibling[x]
9  return y

```

Example: In the given example minimum value returned is 8.



Running Time- Since, binomial heap is Heap-ordered and the minimum key must reside in a ROOT node. The BINOMIAL-HEAP-MINIMUM(H) checks all roots in $O(\lg n)$. Because, Number of Roots in Binomial Heap is at least $\lfloor \lg n \rfloor + 1$ Hence $RUNNING-TIME = O(\lg n)$

4. Extracting the node with minimum key

This procedure extracts the node with the minimum key from binomial heap H and returns a pointer to the extracted node.

```

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.  $H' \leftarrow \text{MAKE-BINOMIAL-HEAP}()$ 
3. reverse the order of the linked list of  $x$ 's children,
   and set  $\text{head}[H']$  to point to the head of the
   resulting list
4.  $H \leftarrow \text{BINOMIAL-HEAP-UNION}(H, H')$ 
5. return  $x$ 

```

The procedure is shown in following example. The input binomial heap H is shown in Figure (a). Figure (b) shows the situation after line 1: the root x with the minimum key has been removed from the root list of H . If x is the root of a B_k -tree, then by property 4 of Lemma 20.1, x 's children, from left to right, are roots of B_{k-1} -, B_{k-2} -, ..., B_0 -trees. Figure (c) shows that by reversing the list of x 's children in line 3, we have a binomial heap H' that contains every node in x 's tree except for x itself. Because x 's tree is removed from H in line 1, the binomial heap that results from uniting H and H' in line 4, shown in Figure (d), contains all the nodes originally in H except for x . Finally, line 5 returns x .

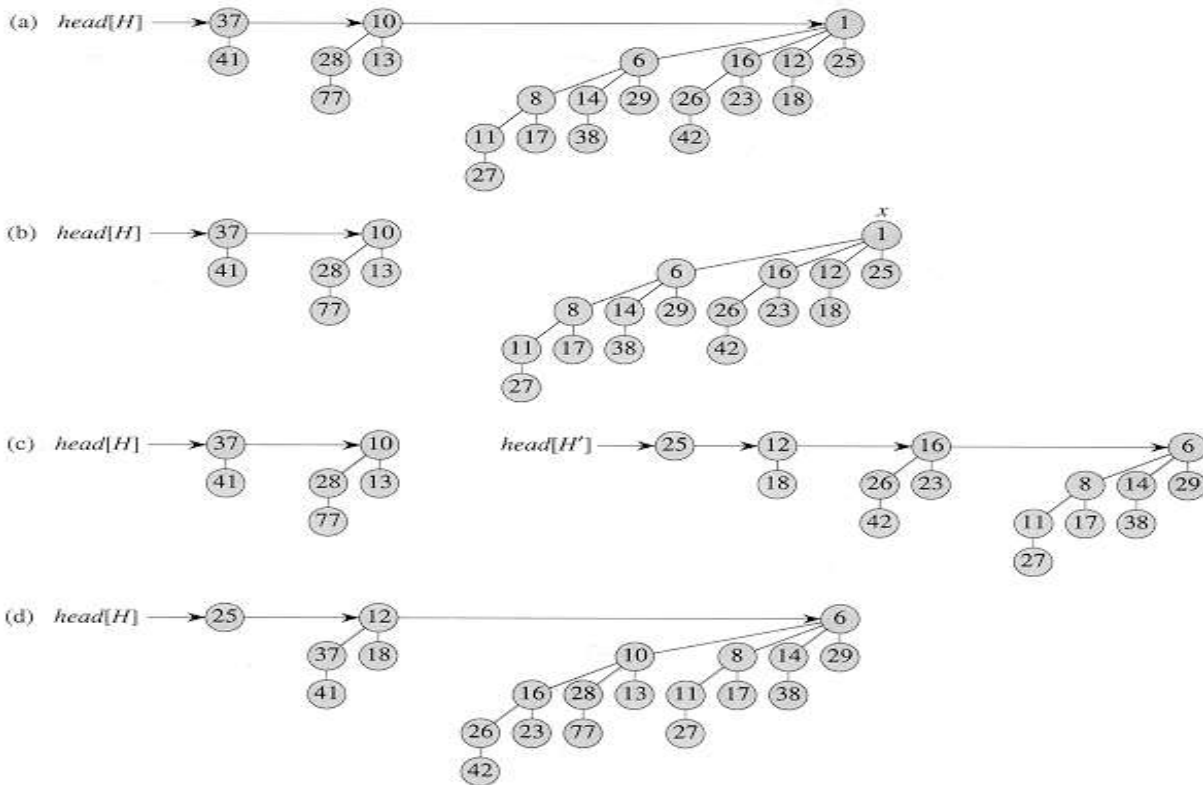


Figure. The action of BINOMIAL-HEAP-EXTRACT-MIN.

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

5. Uniting two binomial heaps

The operation of uniting two binomial heaps is used as a subroutine by most of the remaining operations. The BINOMIAL-HEAP-UNION procedure repeatedly links binomial trees whose roots have the same degree. The following procedure links the B_{k-1} tree rooted at node y 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)

- 1 $p[y] \leftarrow z$
- 2 $sibling[y] \leftarrow child[z]$
- 3 $child[z] \leftarrow y$
- 4 $degree[z] \leftarrow degree[z] + 1$

The BINOMIAL-LINK procedure makes node y the new head of the linked list of node z 's children in $O(1)$ time.

Union procedure- it unites binomial heaps H_1 and H_2 , returning the resulting heap. It destroys the representations of H_1 and H_2 in the process. Besides BINOMIAL-LINK, the procedure uses an auxiliary procedure BINOMIAL-HEAP-MERGE that merges the root lists of H_1 and H_2 into a single linked list that is sorted by degree into monotonically increasing order.

BINOMIAL-HEAP-UNION(H_1, H_2)

```

1   $H \leftarrow \text{MAKE-BINOMIAL-HEAP}()$ 
2   $\text{head}[H] \leftarrow \text{BINOMIAL-HEAP-MERGE}(H_1, H_2)$ 
3  free the objects  $H_1$  and  $H_2$  but not the lists they point to
4  if  $\text{head}[H] = \text{NIL}$ 
5      then return  $H$ 
6   $\text{prev-}x \leftarrow \text{NIL}$ 
7   $x \leftarrow \text{head}[H]$ 
8   $\text{next-}x \leftarrow \text{sibling}[x]$ 
9  while  $\text{next-}x \neq \text{NIL}$ 
10     do if ( $\text{degree}[x] \neq \text{degree}[\text{next-}x]$ ) or
           ( $\text{sibling}[\text{next-}x] \neq \text{NIL}$  and  $\text{degree}[\text{sibling}[\text{next-}x]] = \text{degree}[x]$ )
11         then  $\text{prev-}x \leftarrow x$                                 ▷ Cases 1 and 2
12              $x \leftarrow \text{next-}x$                                 ▷ Cases 1 and 2
13     else if  $\text{key}[x] \leq \text{key}[\text{next-}x]$ 
14         then  $\text{sibling}[x] \leftarrow \text{sibling}[\text{next-}x]$             ▷ Case 3
15             BINOMIAL-LINK( $\text{next-}x, x$ )                            ▷ Case 3
16     else if  $\text{prev-}x = \text{NIL}$                                        ▷ Case 4
17         then  $\text{head}[H] \leftarrow \text{next-}x$                         ▷ Case 4
18             else  $\text{sibling}[\text{prev-}x] \leftarrow \text{next-}x$           ▷ Case 4
19             BINOMIAL-LINK( $x, \text{next-}x$ )                            ▷ Case 4
20              $x \leftarrow \text{next-}x$                                 ▷ Case 4
21      $\text{next-}x \leftarrow \text{sibling}[x]$ 
22 return  $H$ 

```

BINOMIAL-HEAP-UNION procedure with four cases is given in the pseudocode above.

The BINOMIAL-HEAP-UNION procedure has two phases. **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. There might be as many as two roots (but no more) of each degree, however, so the **second phase** links roots 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 link operations quickly.

Lines 1-3 start by merging the root lists of binomial heaps H_1 and H_2 into a single root list H . The root lists of H_1 and H_2 are sorted by strictly increasing degree, and BINOMIAL-HEAP-MERGE returns a root list H that is sorted by monotonically increasing degree. If the root lists of H_1 and H_2 have m roots altogether, BINOMIAL-HEAP-MERGE runs in $O(m)$ time by repeatedly examining the roots at the heads of the two root lists and appending the root with the lower degree to the output root list, removing it from its input root list in the process.

The BINOMIAL-HEAP-UNION procedure next initializes some pointers into the root list of H . First, it simply returns in lines 4-5 if it happens to be uniting two empty binomial heaps. From line 6 on, therefore, we know that H has at least one root. Throughout the procedure, we maintain three pointers into the root list:

- ♦ x points to the root currently being examined,
- ♦ $\text{prev-}x$ points to the root preceding x on the root list: $\text{sibling}[\text{prev-}x] = x$, and
- ♦ $\text{next-}x$ points to the root following x on the root list: $\text{sibling}[x] = \text{next-}x$.

Initially, there are at most two roots on the root list H of a given degree: because H_1 and H_2 were binomial heaps, they each had only one root of a given degree. Moreover, BINOMIAL-HEAP-MERGE guarantees us that if two roots in H have the same degree, they are adjacent in the root list.

In fact, during the execution of BINOMIAL-HEAP-UNION, there may be three roots of a given degree appearing on the root list H at some time. We shall see in a moment how this situation could occur. At each iteration of the **while** loop of lines 9-21, therefore, we decide whether to link x and $\text{next-}x$ based on their degrees and possibly the degree of $\text{sibling}[\text{next-}x]$. An invariant of the loop is that each time we

start the body of the loop, both x and $next\text{-}x$ are non-NIL.

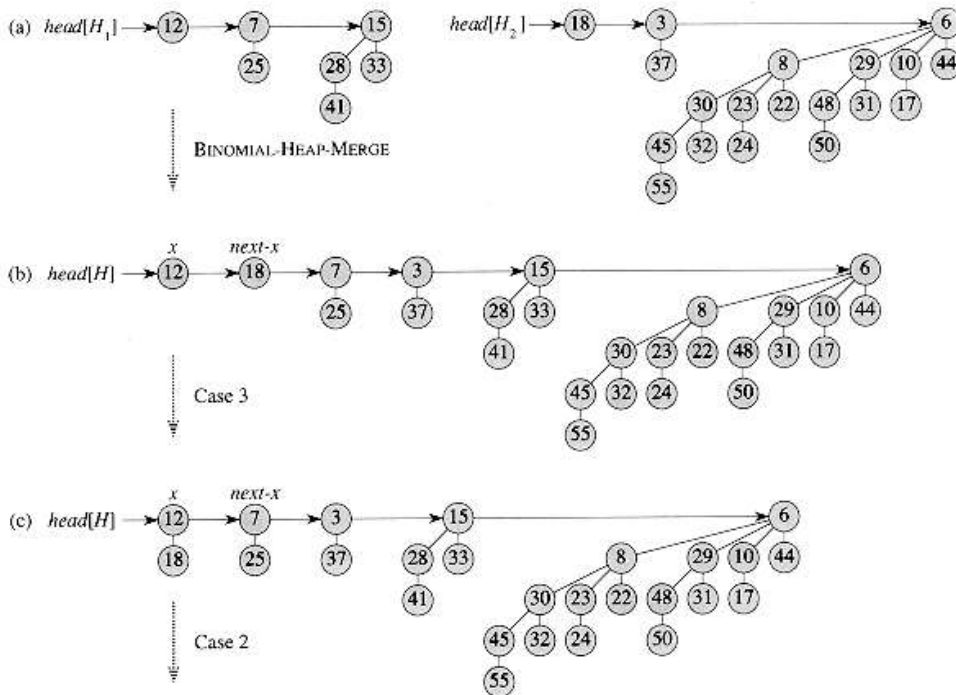
Case 1, shown in Figure 20.6(a), occurs when $degree[x] \neq degree[next\text{-}x]$, that is, when x is the root of a B_k -tree and $next\text{-}x$ is the root of a B_l -tree for some $l > k$. Lines 11-12 handle this case. We don't link x and $next\text{-}x$, so we simply march the pointers one position further down the list. Updating $next\text{-}x$ to point to the node following the new node x is handled in line 21, which is common to every case.

Case 2, shown in Figure 20.6(b), occurs when x is the first of three roots of equal degree, that is, when $degree[x] = degree[next\text{-}x] = degree[sibling[next\text{-}x]]$.

We handle this case in the same manner as case 1: we just march the pointers one position further down the list. Line 10 tests for both cases 1 and 2, and lines 11-12 handle both cases.

Cases 3 and 4 occur when x is the first of two roots of equal degree, that is, when $degree[x] = degree[next\text{-}x] \neq degree[sibling[next\text{-}x]]$.

These cases may occur on the next iteration after any case, but one of them always occurs immediately following case 2. In cases 3 and 4, we link x and $next\text{-}x$. The two cases are distinguished by whether x or $next\text{-}x$ has the smaller key, which determines the node that will be the root after the two are linked. In case 3, shown in Figure 20.6(c), $key[x] \leq key[next\text{-}x]$, so $next\text{-}x$ is linked to x . Line 14 removes $next\text{-}x$ from the root list, and line 15 makes $next\text{-}x$ the leftmost child of x .



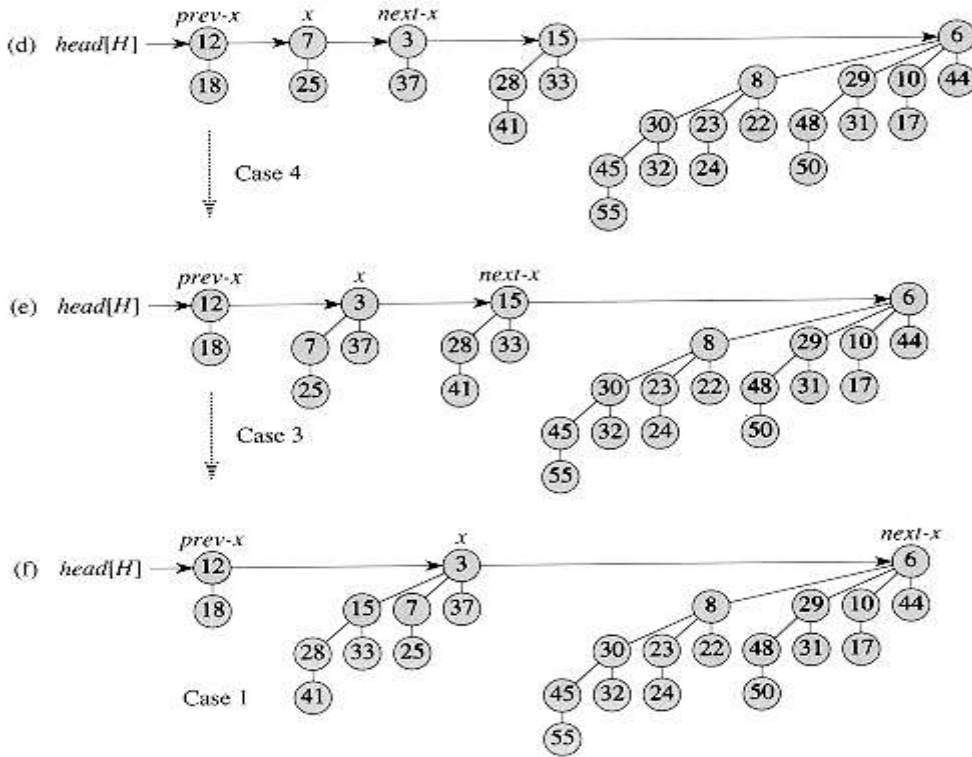


Figure. The execution of BINOMIAL-HEAP-UNION

In case 4, $next-x$ has the smaller key, so x is linked to $next-x$. Lines 16-18 remove x from the root list, which has two cases depending on whether x is the first root on the list (line 17) or is not (line 18). Line 19 then makes x the leftmost child of $next-x$, and line 20 updates x for the next iteration. Following either case 3 or case 4, the setup for the next iteration of the **while** loop is the same. We have just linked two B_k -trees to form a B_{k+1} -tree, which x now points to. There were already zero, one, or two other B_{k+1} -trees on the root list from the output of BINOMIAL-HEAP-MERGE, so x is now the first of either one, two, or three B_{k+1} -trees on the root list. If x is the only one, then we enter case 1 in the next iteration: $degree[x] \neq degree[next-x]$. If x is the first of two, then we enter either case 3 or case 4 in the next iteration. It is when x is the first of three that we enter case 2 in the next iteration. The running time of BINOMIAL-HEAP-UNION is $O(\lg n)$, where n is the total number of nodes in binomial heaps H_1 and H_2 . We can see this as follows. Let H_1 contain n_1 nodes and H_2 contain n_2 nodes, so that $n = n_1 + n_2$. Then, H_1 contains at most $\lfloor \lg n_1 \rfloor + 1$ roots and H_2 contains at most $\lfloor \lg n_2 \rfloor + 1$ roots, so H contains at most $\lfloor \lg n_2 \rfloor + \lfloor \lg n_1 \rfloor + 2 \leq 2 \lfloor \lg n \rfloor + 2 = O(\lg n)$ roots immediately after the call of BINOMIAL-HEAP-MERGE. The time to perform BINOMIAL-HEAP-MERGE is thus $O(\lg n)$. Each iteration of the **while** loop takes $O(1)$ time, and there are at most $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2$ iterations because each iteration either advances the pointers one position down the root list of H or removes a root from the root list. The total time is thus $O(\lg n)$.

6. Decreasing a key

The following procedure decreases the key of a node x in a binomial heap H to a new value k . It signals an error if k is greater than x 's current key.

BINOMIAL-HEAP-DECREASE-KEY (H, x, k)

1. **if** $k > key[x]$
2. **then 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 $key[y] \leftrightarrow key[z]$

8. \triangleright If y and z have satellite fields, exchange them, too.
9. $y \leftarrow z$
10. $z \leftarrow p[y]$

As shown in Figure 20.8, this procedure decreases a key in the same manner as in a binary heap: by "bubbling up" the key in the heap. After ensuring that the new key is in fact no greater than the current key and then assigning the new key to x , the procedure goes up the tree, with y initially pointing to node x . In each iteration of the **while** loop of lines 6-10, $key[y]$ is checked against the key of y 's parent z . If y is the root or $key[y] \geq key[z]$, the binomial tree is now heap-ordered. Otherwise, node y violates heap ordering, so its key is exchanged with the key of its parent z , along with any other satellite information. The procedure then sets y to z , going up one level in the tree, and continues with the next iteration.

The BINOMIAL-HEAP-DECREASE-KEY procedure takes $O(\lg n)$ time. By property 2 of Lemma 20.1, the maximum depth of x is $\lfloor \lg n \rfloor$, so the **while** loop of lines 6-10 iterates at most $\lfloor \lg n \rfloor$ times.

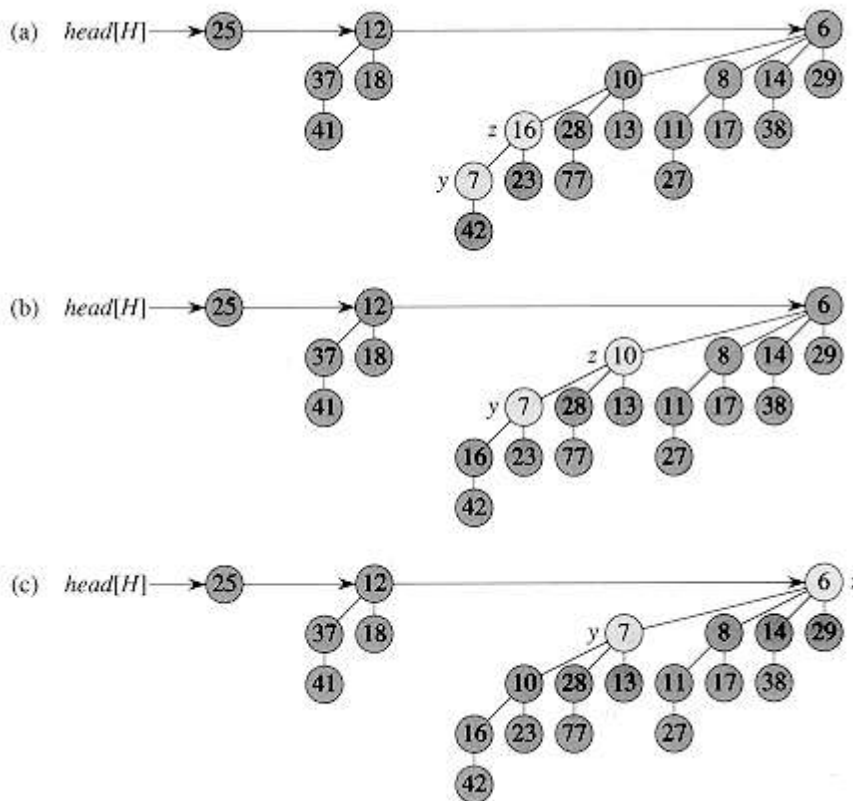


Figure. The action of BINOMIAL-HEAP-DECREASE-KEY. (a) The situation just before line 5 of the first iteration of the while loop. Node y has had its key decreased to 7, which is less than the key of y 's parent z . (b) The keys of the two nodes are exchanged, and the situation just before line 5 of the second iteration is shown. Pointers y and z have moved up one level in the tree, but heap order is still violated. (c) After another exchange and moving pointers y and z up one more level, we finally find that heap order is satisfied, so the while loop terminates.

7. Deleting a key

It is easy to delete a node x 's key and satellite information from binomial heap H in $O(\lg n)$ time. 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( $H$ )

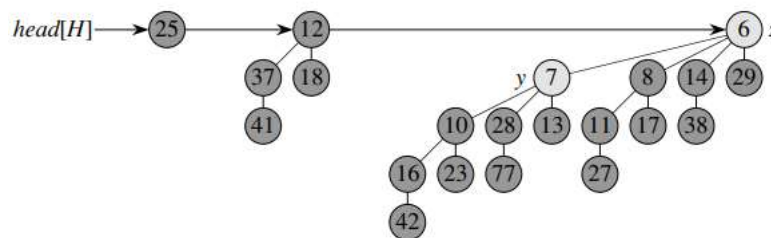
```


The BINOMIAL-HEAP-DELETE procedure makes node x have the unique minimum key in the entire binomial heap by giving it a key of $-\infty$. It then bubbles this key and the associated satellite information up to a root by calling BINOMIAL-HEAP-DECREASE-KEY. This root is then removed from H by a call of BINOMIAL-HEAP-EXTRACT-MIN.

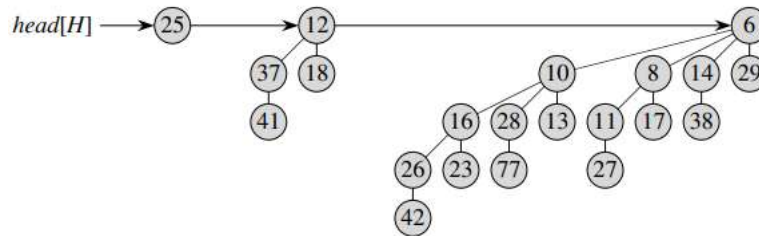
The BINOMIAL-HEAP-DELETE procedure takes $O(\lg n)$ time.

Important Questions

1. 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 2021-22, 2018-19, 2017-18]**
2. Explain properties of Binomial Heap. Write an algorithm to perform uniting two Binomial Heaps. And also, to find Minimum Key. **[AKTU 2017-18]**
3. Discuss the Cases and algorithm for Binomial Heap Union and Finding the Minimum Key in Binomial Heap with their Complexity. Also Perform Binomial Heap Insert on $A = \langle 10, 6, 7, 8, 9, 11, 14, 15, 16, 2, 3, 4, 5 \rangle$
4. Define Binomial Tree. Explain properties of Binomial Heap with example.
5. Discuss the Binomial Heap Insert, Binomial Heap Extract Min and Binomial Heap Delete Key Operation with its Complexity.
6. Show the binomial heap that results when the node with key 28 is deleted from the binomial heap shown in the given figure.



7. Show the binomial heap that results when a node with key 24 is inserted into the binomial heap shown in given figure.



8. Show that if root lists are kept in strictly decreasing order by degree (instead of strictly increasing order), each of the binomial heap operations can be implemented without changing its asymptotic running time.

Fibonacci Heap:

Fibonacci heap is designed and developed by Fredman and Tarjan in the year 1986.

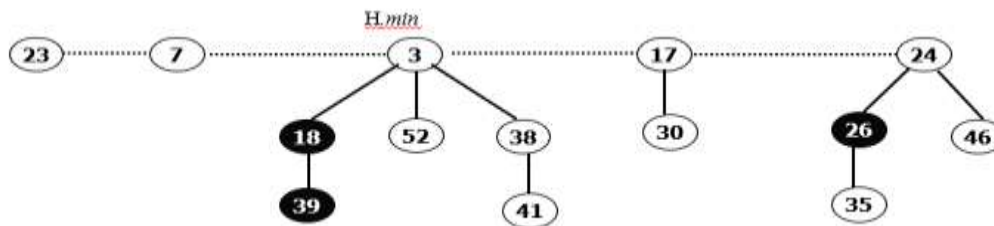
Ingenious data structure and analysis.

Like a binomial heap, a Fibonacci heap is a collection of trees with looser structure.

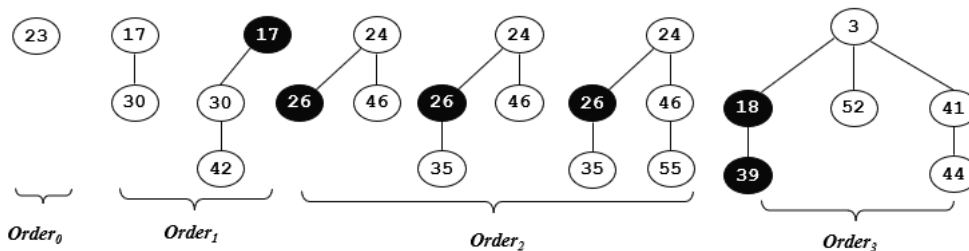
Fibonacci heaps are called lazy data structures because they delay work as long as possible using the field mark (i.e. black node).

Structure of Fibonacci heaps:

Like a binomial heap, a Fibonacci heap is a collection of min-heap-ordered trees. The trees in a Fibonacci heap are not constrained to be binomial trees. The picture shows below is an example of a Fibonacci heap.



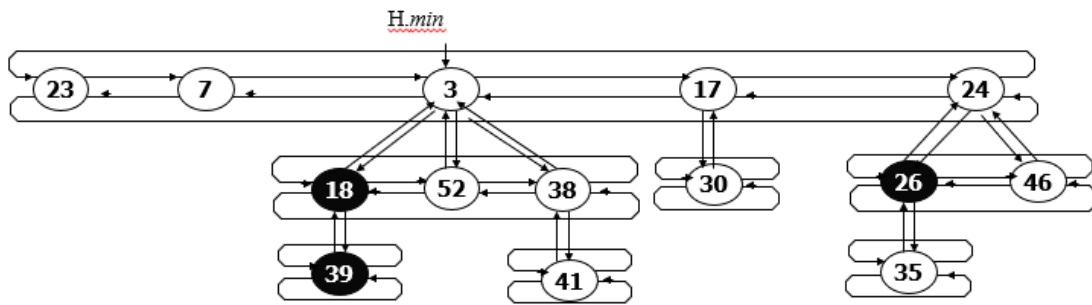
- Nodes within a Fibonacci heap can be removed from their tree without restructuring them.
- So the order does not necessarily indicate the maximum height of the tree or number of nodes it contain. Some examples of order 0, 1, 2 are given below for easy understanding



Each node x contain the following fields.

- $x.p \rightarrow$ points to its parent
- $x.child \rightarrow$ points to any one of its children and children of x are linked together in a circular doubly linked list.
- $x.left, x.right \rightarrow$ points to its left and right siblings.
- $x.degree \rightarrow$ number of children in the child list of x
- $x.mark \rightarrow$ indicate whether node x has lost a child since the last time x was made the child of another node
- $H.min \rightarrow$ points to the root of the tree containing a minimum key
- $H.n \rightarrow$ number of nodes in H

An example of Fibonacci Heap with the help of doubly circular link list is given below.



The Fibonacci heap data structure serves a dual purpose.

- First, it supports a set of operations that constitutes what is known as a “mergeable heap.”
- Second, several Fibonacci-heap operations run in constant amortized time, which makes this data structure well suited for applications that invoke these operations frequently.

Write down the properties of Fibonacci Heap.

Fibonacci Heap - A Fibonacci heap is defined as the collection of rooted-tree in which all the trees must hold the property of Min-heap. That is, for all the nodes, the key value of the parent node should be greater than the key value of the parent node:

Properties of Fibonacci Heap:

1. It can have multiple trees of equal degrees, and each tree doesn't need to have 2^k nodes.
2. All the trees in the Fibonacci Heap are rooted but not ordered.
3. All the roots and siblings are stored in a separated circular-doubly-linked list.
4. The degree of a node is the number of its children. Node X \rightarrow degree = Number of X's children.
5. Each node has a mark-attribute in which it is marked TRUE or FALSE. The FALSE indicates the node has not any of its children. The TRUE represents that the node has lost one child. The newly created node is marked FALSE.
6. The potential function of the Fibonacci heap is $F(FH) = t[FH] + 2 * m[FH]$
7. The Fibonacci Heap (FH) has some important technicalities listed below:
 1. $min[FH]$ - Pointer points to the minimum node in the Fibonacci Heap
 2. $n[FH]$ - Determines the number of nodes
 3. $t[FH]$ - Determines the number of rooted trees
 4. $m[FH]$ - Determines the number of marked nodes

$F(FH)$ - Potential Function

A mergeable heap is any data structure that supports the following seven operations, in which each element has a key:

1. **MAKE-HEAP()** creates and returns a new heap containing no elements.
2. **MINIMUM(H)** returns a pointer to the node in heap H whose key is minimum.
3. **UNION(H1, H2)** creates and returns a new heap that contains all the nodes of heaps H1 and H2. Heaps H1 and H2 are "destroyed" by this operation.
4. **EXTRACT-MIN(H)** deletes the node from heap H whose key is minimum, returning a pointer to the node.
5. **INSERT(H, x)** inserts node x, whose key field has already been filled in, into heap H.
6. **DECREASE-KEY(H, x, k)** assigns to node x within heap H the new key value k, which is assumed to be no greater than its current key value.
7. **DELETE(H, x)** deletes node x from heap H

Heaps Analysis			
Operation	Binary	Binomial	Fibonacci [†]
make-heap	1	1	1
insert	log N	log N	1
find-min	1	log N	1
delete-min	log N	log N	log N
union	N	log N	1
decrease-key	log N	log N	1
delete	log N	log N	log N
is-empty	1	1	1

Amortized Analysis

- Analyze a sequence of operations on a data structure.
- It shows that although some individual operations may be expensive, on average the cost per operation is small.

Average in this context does not mean that we're averaging over a distribution of inputs.

- No probability is involved.
- We're talking about average cost in the worst case.

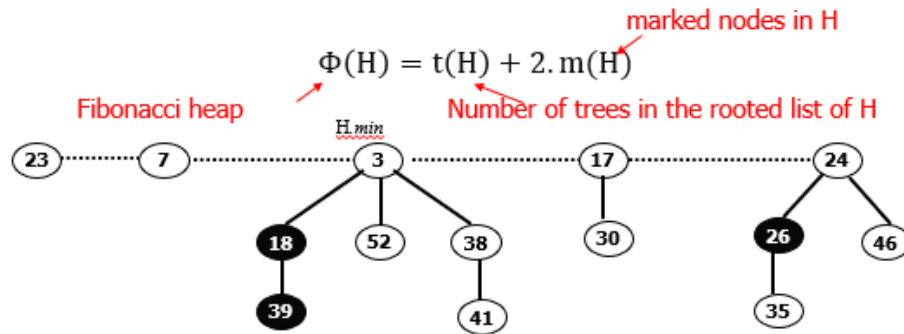
The Three most common techniques used are

- Aggregate Analysis
- Accounting Method

- Potential Method

The potential method is used to analyze the performance of Fibonacci heap operations.

- For a given Fibonacci heap H
 - $t(H)$ the number of trees in the root list of H .
 - $m(H)$ the number of marked nodes in H .
- The potential of Fibonacci heap H is then defined by



In the above example $t(H) = 5$ and $m(H) = 3$, Hence
 $\Phi(H) = 5 + 2 \cdot 3 = 11$

Fibonacci heap application begins with no heaps. Hence

- The initial potential is 0(zero) and the potential is nonnegative at all subsequent times.
- An upper bound on the total amortized cost is thus an upper bound on the total actual cost for the sequence of operations.

Maximum degree

- The amortized analyses was performed with a known upper bound $D(n)$ on the maximum degree of any node in an n -node Fibonacci heap. (i.e. $D(n) \leq \lceil \lg n \rceil$).

Fibonacci Heap (Operations)

1.) Creating a new Fibonacci Heap:

- To make an empty Fibonacci heap, the MAKE-FIB-HEAP procedure allocates and returns the Fibonacci heap object H ,
- where $n[H] = 0$ and $\text{min}[H] = \text{NIL}$; there are no trees in H .
- Because $t(H) = 0$ and $m(H) = 0$, the potential of the empty Fibonacci heap is
- $\Phi(H) = 0$.
- The amortized cost of MAKE-FIB-HEAP is thus equal to its $O(1)$ actual cost.

2.) Inserting a node:

3.) To insert a node in Fibonacci heap the following steps will be used.

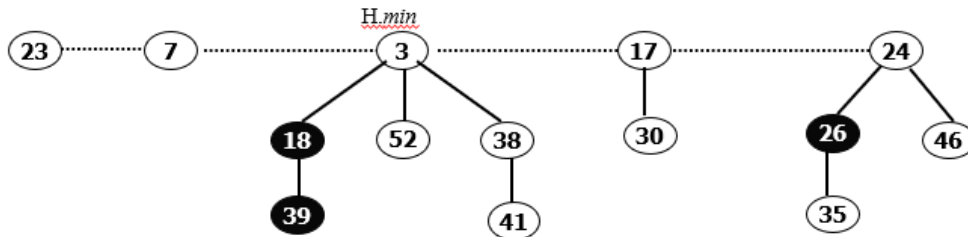
4.) Step1 :Create a new singleton tree.

5.) Step2 : Add to left of min pointer.

6.) Step3 : Update min pointer.

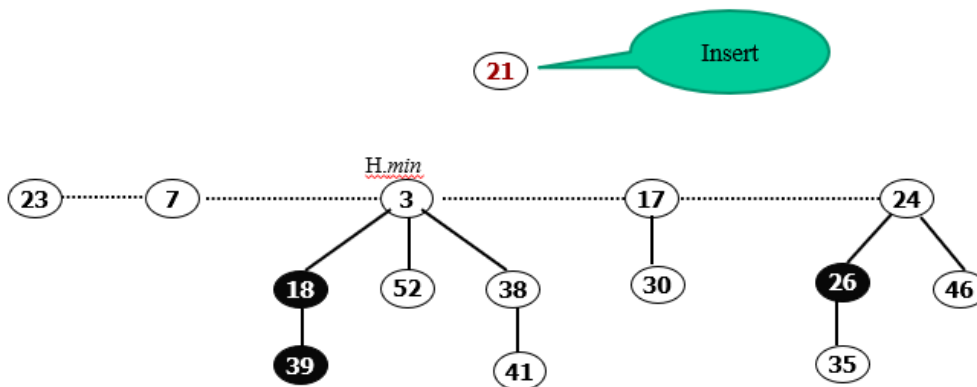
Inserting a node:

- Example: Insert the node 21 in following Fibonacci heap



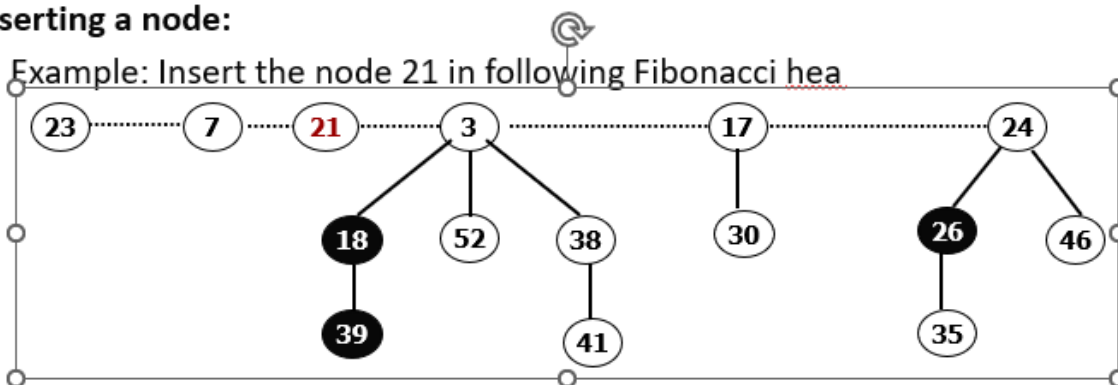
Inserting a node:

- Example: Insert the node 21 in following Fibonacci heap



Inserting a node:

- Example: Insert the node 21 in following Fibonacci heap



Inserting a node: The following procedure inserts node x into Fibonacci heap H , assuming that the node has already been allocated and that $\text{key}[x]$ has already been filled in.

FIB-HEAP-INSERT(H, x)

- 1 $\text{degree}[x] \leftarrow 0$
- 2 $p[x] \leftarrow \text{NIL}$
- 3 $\text{child}[x] \leftarrow \text{NIL}$
- 4 $\text{left}[x] \leftarrow x$

```

5  right[x] ← x
6  mark[x] ← 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] ← x
10 n[H] ← n[H] + 1

```

Inserting a node (Analysis)

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 difference in potential cost is=

$$((t(H) + 1) + 2 m(H)) - (t(H) + 2 m(H)) = 1.$$

Since the actual cost is $O(1)$,

the amortized cost = Actual cost+ Difference in potential cost ($\Delta(\Phi)$)
 $= O(1) + 1 = O(1)$.

3.) Finding the minimum node

The minimum node of a Fibonacci heap H is given by the pointer min[H],

So the actual time for finding the minimum node is $O(1)$. Because the potential of H does not change,
and the amortized cost of this operation is equal to its $O(1)$ actual cost.

4.) Uniting two Fibonacci heaps

This procedure unites Fibonacci heaps H_1 and H_2 , destroying H_1 and H_2 in the process. It simply
concatenates the root lists of H_1 and H_2 and then determines the new minimum node.

Basic Idea:

Step 1: Concatenate two Fibonacci heaps.

Step 2: Root lists are circular, doubly linked lists.

Uniting two Fibonacci heaps

FIB-HEAP-UNION(H_1 , H_2)

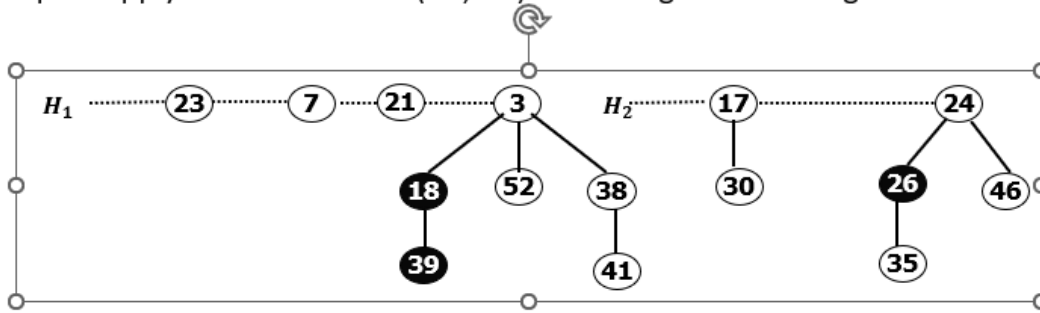
```

1  H ← MAKE-FIB-HEAP()
2  min[H] ← min[H1]
3  Concatenate the root list of H2 with the root list of H
4  if (min[H1] = NIL) or (min[H2] ≠ NIL and min[H2] < min[H1])
5    then min[H] ← min[H2]
6  n[H] ← n[H1] + n[H2]
7  free the objects H1 and H2
8  return H

```

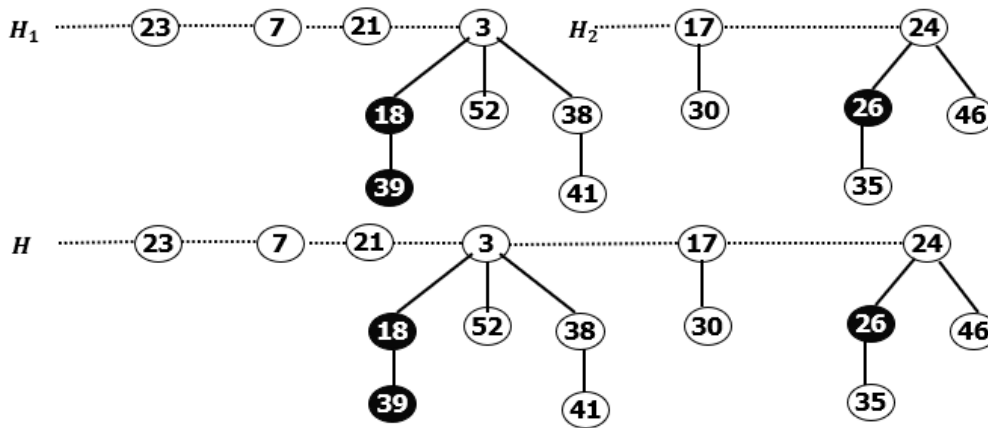
Uniting two Fibonacci heaps:

Example : Apply FIB-HEAP-UNION(H_1 , H_2) for uniting the following two Fibonacci Heaps



Uniting two Fibonacci heaps

Example : Apply FIB-HEAP-UNION(H_1 , H_2) for uniting the following two Fibonacci Heaps



Uniting two Fibonacci heaps (Analysis)

FIB-HEAP-UNION(H_1 , H_2)

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

Lines 1-3 concatenate the root lists of H_1 and H_2 into a new root list H .

Lines 2, 4, and 5 set the minimum node of H , and line 6 sets $n[H]$ to the total number of nodes.

The Fibonacci heap objects H_1 and H_2 are freed in line 7, and line 8 returns the resulting Fibonacci heap H .

As in the FIB-HEAP-INSERT procedure, no consolidation of trees occurs. The change in potential is

$$\begin{aligned}
 &= \Phi(H) - (\Phi(H_1) + \Phi(H_2)) \\
 &= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) \\
 &= 0,
 \end{aligned}$$

because $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$.

The amortized cost of FIB-HEAP-UNION is therefore equal to its $O(1)$ actual cost.

5.) Extracting the minimum node

The process of extracting the minimum node is the most complicated of the operations presented in this section.

Basic Idea:

Step 1: Delete min and concatenate its children into root list.

Step 2: Consolidate trees so that no two roots have same degree.

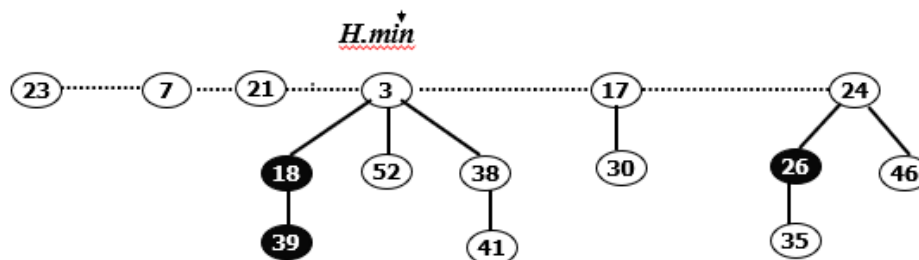
Extracting the minimum node

FIB-HEAP-EXTRACT-MIN(H)

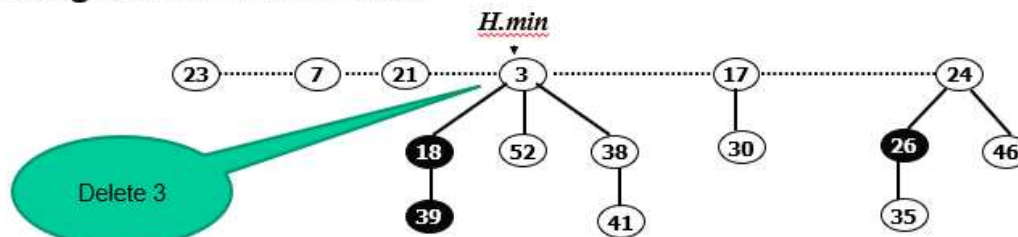
```
1  z ← min[H]
2  if z ≠ NIL
3      then for each child x of z
4          do add x to the root list of H
5          p[x] ← NIL
6          remove z from the root list of H
7  if z = right[z]
8      then min[H] ← NIL
9  else min[H] ← right[z]
10 CONSOLIDATE(H)
11 n[H] ← n[H] - 1
12 return z.
```

Extracting the minimum node

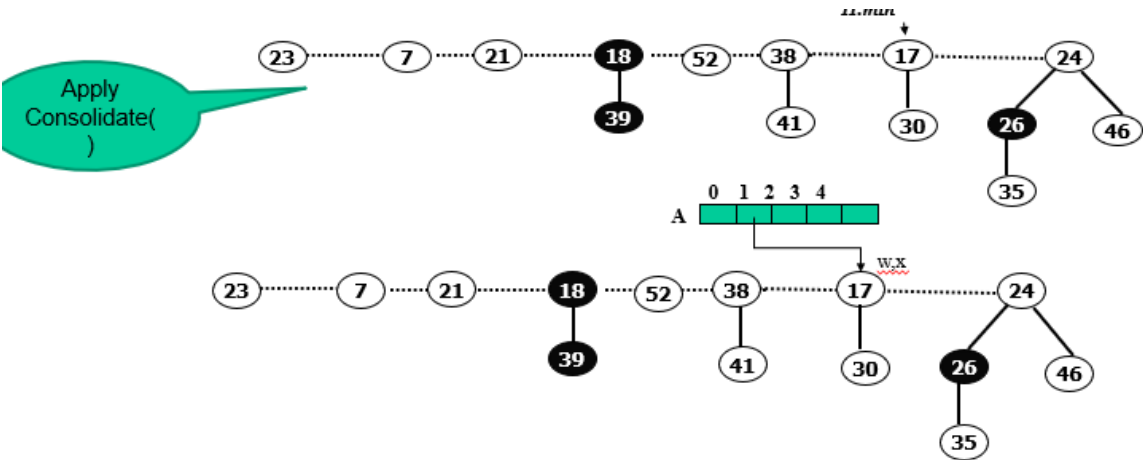
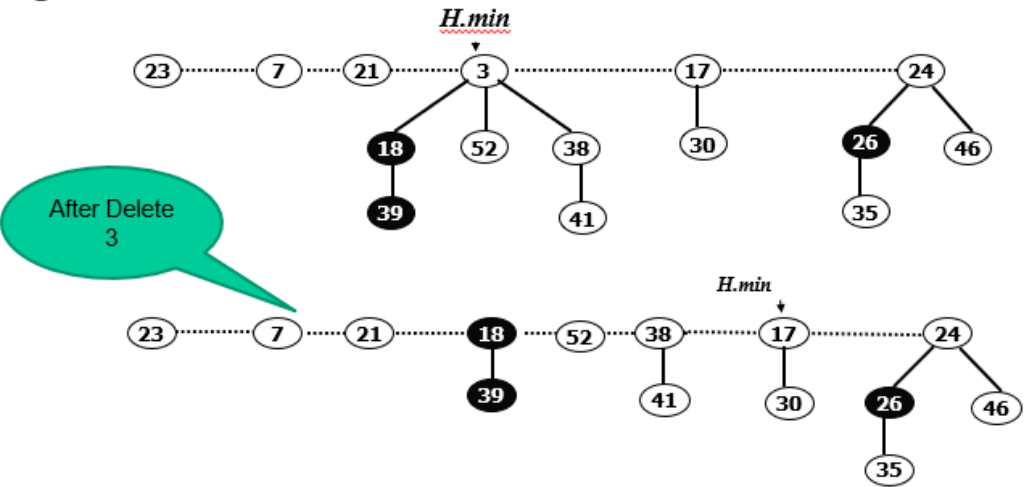
Example: Extract the minimum element from the following Fibonacci heap.



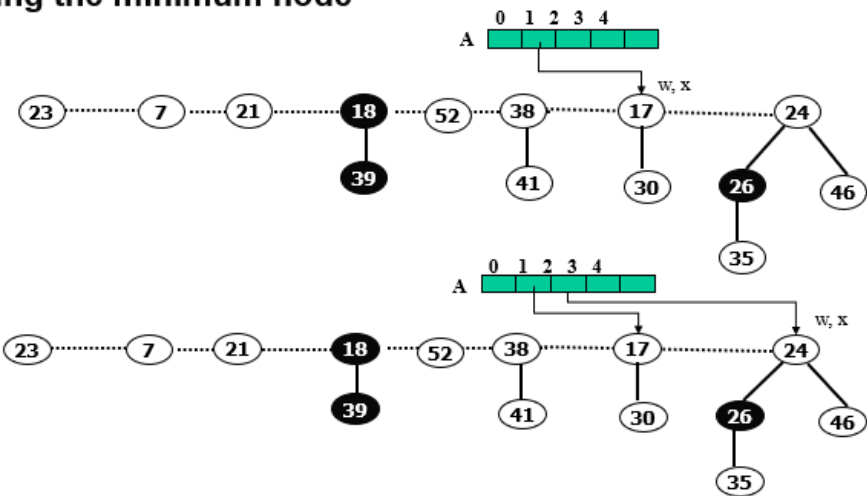
Extracting the minimum node:



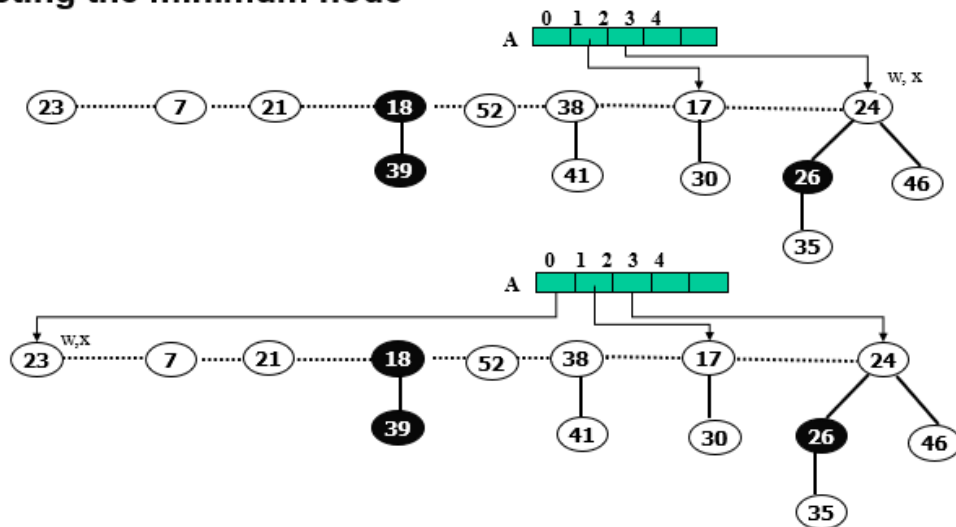
Extracting the minimum node



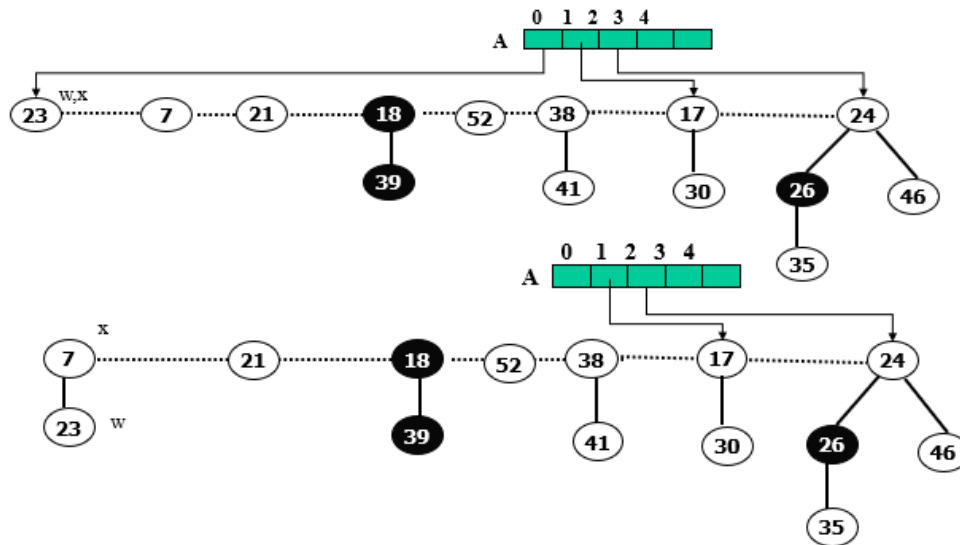
Extracting the minimum node



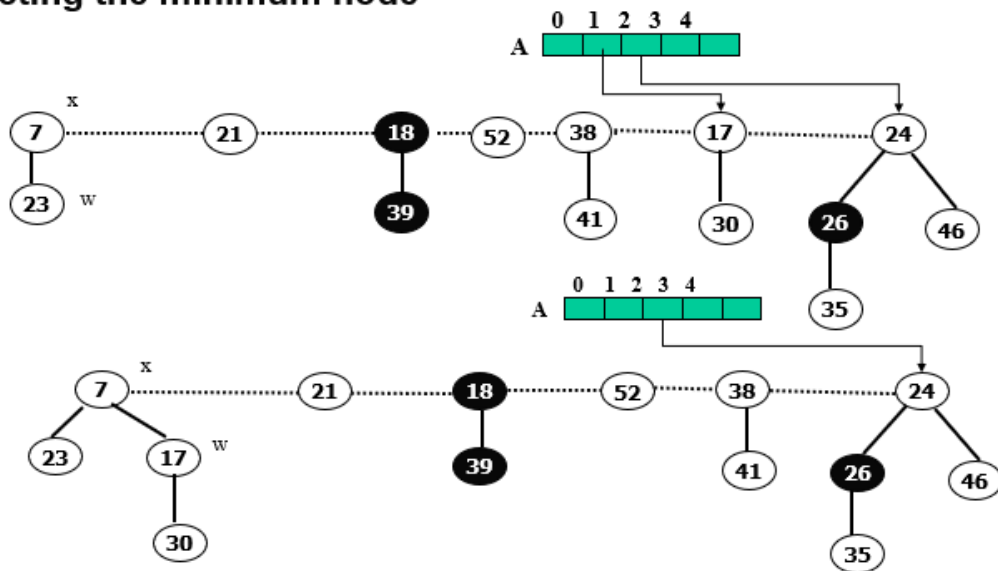
Extracting the minimum node



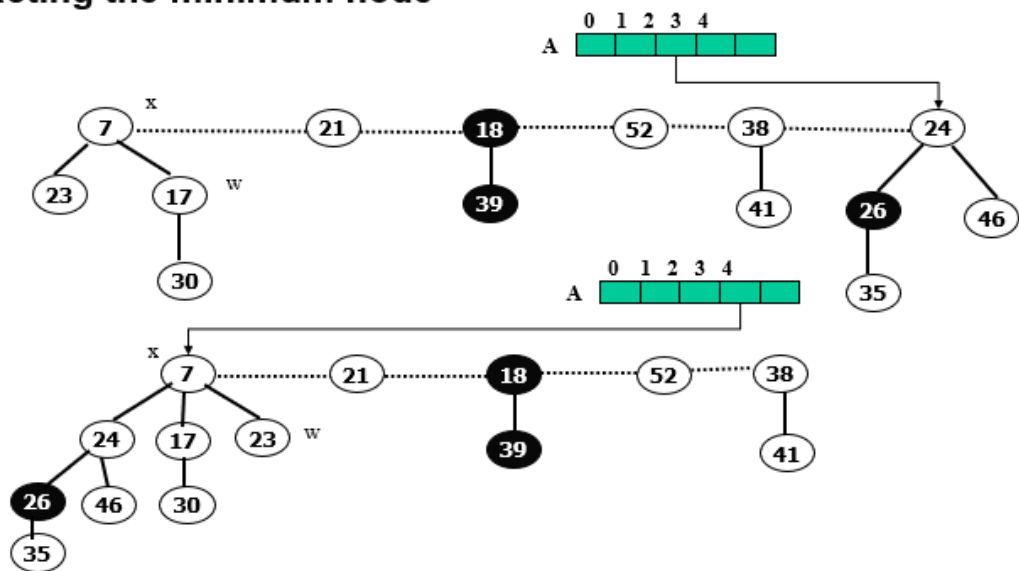
Extracting the minimum node



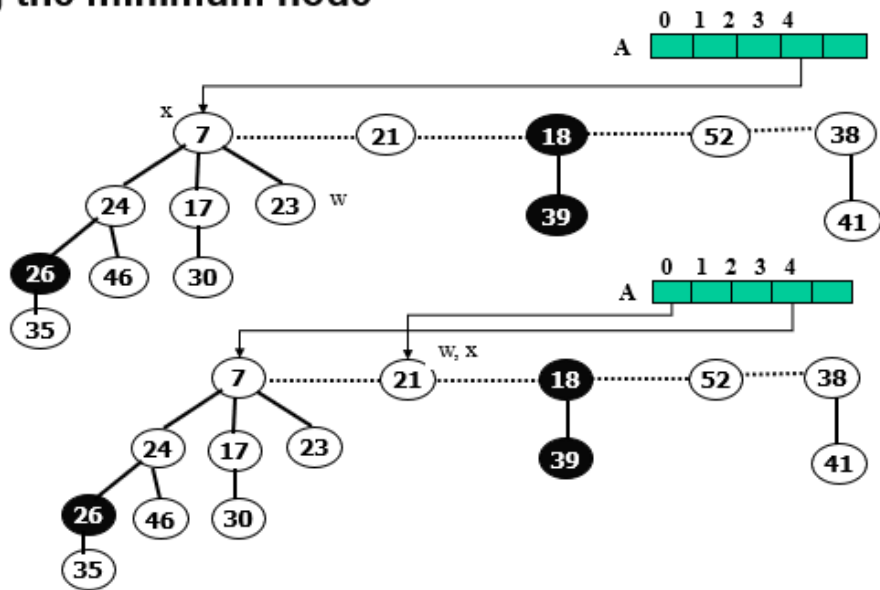
Extracting the minimum node



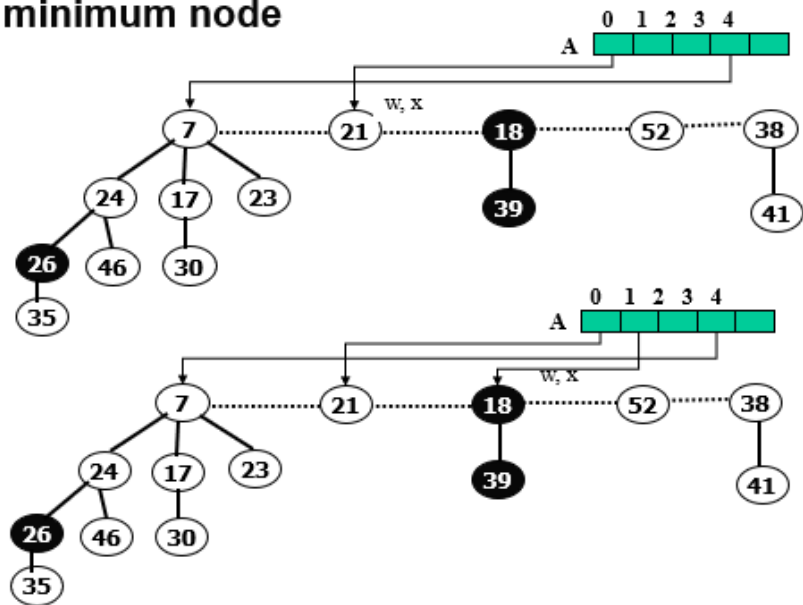
Extracting the minimum node



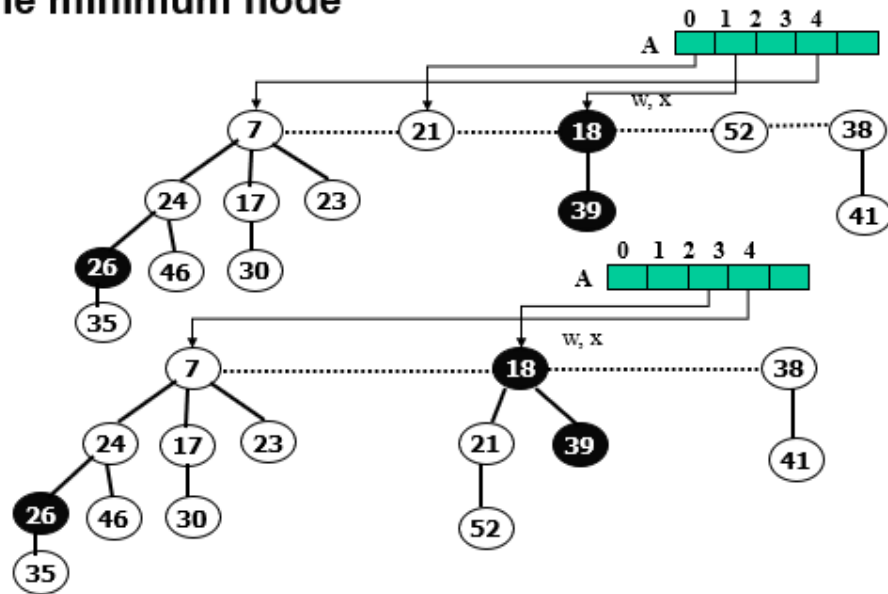
Extracting the minimum node



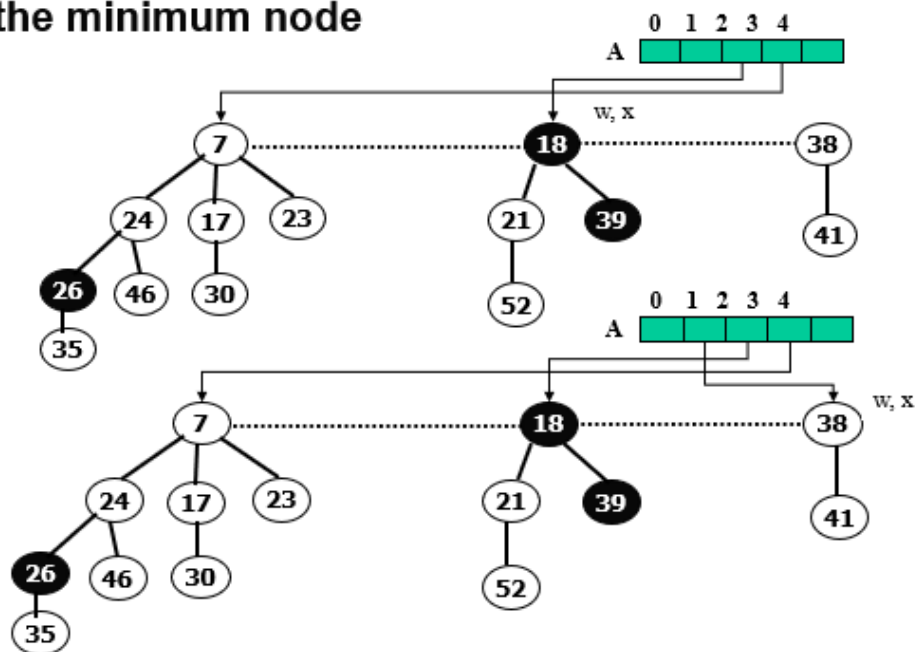
Extracting the minimum node



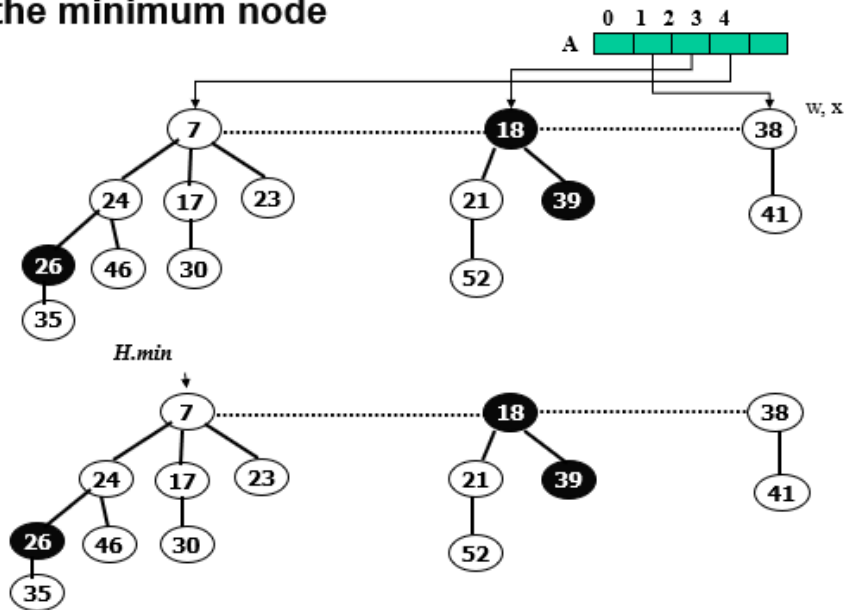
Extracting the minimum node



Extracting the minimum node



Extracting the minimum node



Extracting the minimum node

CONSOLIDATE(H)

```

1   for  $i \leftarrow 0$  to  $D(n[H])$ 
2       do  $A[i] \leftarrow \text{NIL}$ 
3   for each node  $w$  in the root list of  $H$ 
4       do  $x \leftarrow w$ 
5            $d \leftarrow \text{degree}[x]$ 
6           while  $A[d] \neq \text{NIL}$ 
7               do  $y \leftarrow A[d]$   $\triangleright$  Another node with the same degree as  $x$ .
8                   if  $\text{key}[x] > \text{key}[y]$ 
9                       then exchange  $x \leftrightarrow y$ 
10                  FIB-HEAP-LINK( $H, y, x$ )
11                   $A[d] \leftarrow \text{NIL}$ 
12                   $d \leftarrow d + 1$ 
13           $A[d] \leftarrow x$ 
14.  $\text{min}[H] \leftarrow \text{NIL}$ 
15. for  $i \leftarrow 0$  to  $D(n[H])$ 
16.     do if  $A[i] \neq \text{NIL}$ 
17.         then add  $A[i]$  to the root list of  $H$ 
18.         if  $\text{min}[H] = \text{NIL}$  or  $\text{key}[A[i]] < \text{key}[\text{min}[H]]$ 
19.             then  $\text{min}[H] \leftarrow A[i]$ 

```

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

```

1. remove  $y$  from the root list of  $H$ 
2. make  $y$  a child of  $x$ , incrementing  $\text{degree}[x]$ 
3.  $\text{mark}[y] \leftarrow \text{FALSE}$ 

```

Extracting the minimum node(Analysis)

Notation:

$D(n) = \text{max degree of any node in Fibonacci heap with } n \text{ nodes.}$

$t(H) = \text{Number of trees in heap } H.$

$m(H) = \text{the number of marked nodes in } H.$

$$\Phi(H) = t(H) + 2m(H).$$

Actual cost :

$O(D(n))$: for loop in Fib-Heap-Extract-Min

$D(n) + t(H) - 1$: size of the root list

Total actual cost: $O(D(n)) + t(H)$

Potential before extracting : $t(H) + 2m(H)$

Potential after extracting : $\leq D(n) + 1 + 2m(H)$

At most $D(n)+1$ nodes remain on the list and no nodes become marked.

Thus the amortized cost is at most:

$$\begin{aligned} &= O(D(n)) + t(H) + [(D(n) + 1 + 2m(H)) - (t(H) + 2m(H))] \\ &= O(D(n) + t(H) - t(H)) \\ &= O(D(n)) \\ &= O(\log n) \end{aligned}$$

[Note: An n node Binomial heap H consists of at most $\lfloor \log n \rfloor + 1$ binomial Tree]

6.) Decreasing a key

This pseudocode for the operation FIB-HEAP-DECREASE-KEY, work with an assumption that removing a node from a linked list does not change any of the structural fields in the removed node.

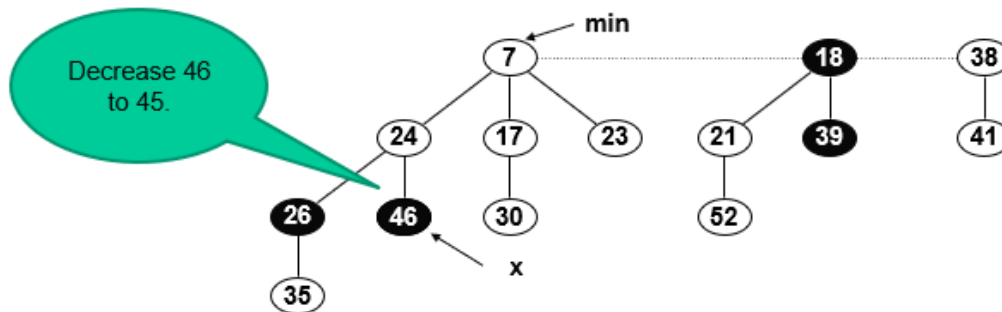
Decreasing a key

Basic Idea:

Case 0: min-heap property not violated.

- decrease key of x to k
- change heap min pointer if necessary

Example:



Basic Idea:

Case 1: parent of x is unmarked.

decrease key of x to k

cut off link between x and its parent

mark parent

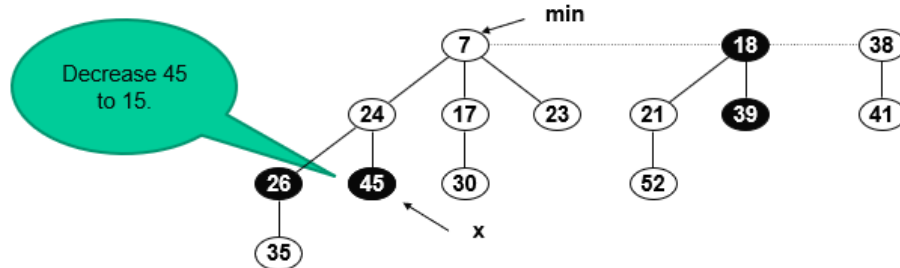
add tree rooted at x to root list, updating heap min pointer

Decreasing a key and deleting node

Basic Idea:

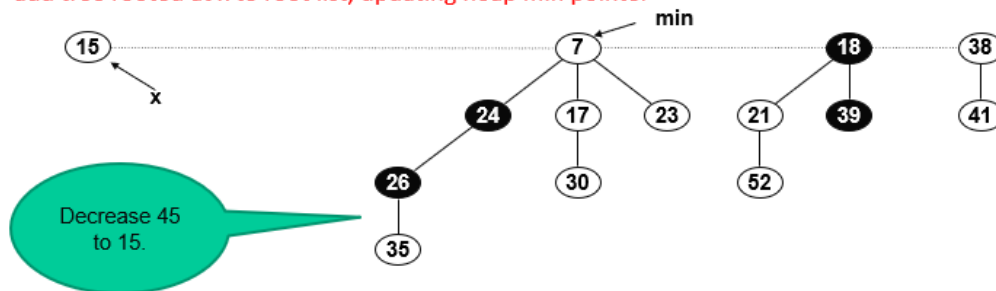
Case 1: parent of x is unmarked.

- decrease key of x to k
- cut off link between x and its parent
- mark parent
- add tree rooted at x to root list, updating heap min pointer



Case 1: parent of x is unmarked.

- decrease key of x to k
- cut off link between x and its parent
- mark parent
- add tree rooted at x to root list, updating heap min pointer



Decreasing a key

Basic Idea:

Case 2: parent of x is marked.

decrease key of x to k

cut off link between x and its parent $p[x]$, and add x to root list

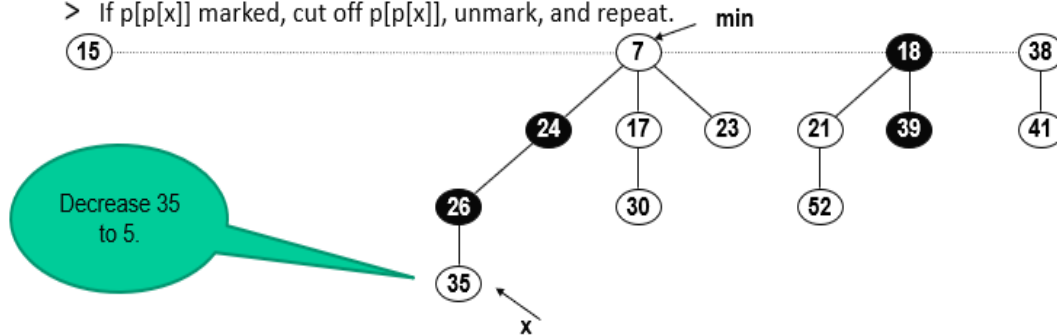
cut off link between $p[x]$ and $p[p[x]]$, add $p[x]$ to root list

> If $p[p[x]]$ unmarked, then mark it.

> If $p[p[x]]$ marked, cut off $p[p[x]]$, unmark, and repeat.

Case 2: parent of x is marked.

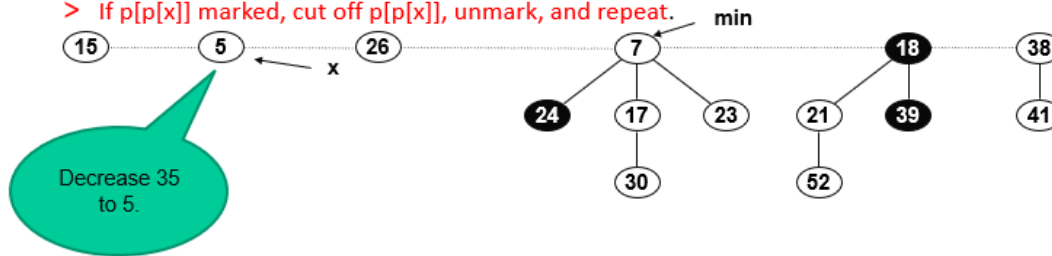
- decrease key of x to k
- cut off link between x and its parent $p[x]$, and add x to root list
- cut off link between $p[x]$ and $p[p[x]]$, add $p[x]$ to root list
 - > If $p[p[x]]$ unmarked, then mark it.
 - > If $p[p[x]]$ marked, cut off $p[p[x]]$, unmark, and repeat.



Decreasing a key: Basic Idea:

Case 2: parent of x is marked.

- decrease key of x to k
- cut off link between x and its parent $p[x]$, and add x to root list
- cut off link between $p[x]$ and $p[p[x]]$, add $p[x]$ to root list
 - > If $p[p[x]]$ unmarked, then mark it.
 - > If $p[p[x]]$ marked, cut off $p[p[x]]$, unmark, and repeat.



Decreasing a key

FIB-HEAP-DECREASE-KEY(H, x, k)

- 1 if $k > \text{key}[x]$
- 2 then error "new key is greater than current key"
- 3 $\text{key}[x] \leftarrow k$
- 4 $y \leftarrow p[x]$
- 5 if $y \neq \text{NIL}$ and $\text{key}[x] < \text{key}[y]$
- 6 then CUT(H, x, y)
- 7 CASCADING-CUT(H, y)
- 8 if $\text{key}[x] < \text{key}[\text{min}[H]]$
- 9 then $\text{min}[H] \leftarrow x$

Decreasing a key

CUT(H, x, y)

- 1 remove x from the child list of y, decrementing $\text{degree}[y]$
- 2 add x to the root list of H
- 3 $p[x] \leftarrow \text{NIL}$
- 4 $\text{mark}[x] \leftarrow \text{FALSE}$

CASCADING-CUT(H, y)

- 1 $z \leftarrow p[y]$

```

2      if z ≠ NIL
3          then if mark[y] = FALSE
4              then mark[y] ← TRUE
5              else   CUT(H, y, z)
6                  CASCADING-CUT(H, z)

```

Decreasing a key (Analysis)

The FIB-HEAP-DECREASE-KEY procedure takes $O(1)$ time, plus the time to perform the cascading cuts.

Suppose that CASCADING-CUT is recursively called c times from a given invocation of FIB-HEAP-DECREASE-KEY.

Each call of CASCADING-CUT takes $O(1)$ time exclusive of recursive calls.

Thus, the actual cost of FIB-HEAP-DECREASE-KEY, including all recursive calls, is $O(c)$

Each recursive call of CASCADING-CUT except for the last one, cuts a marked node and clears the mark bit.

[Note: Last call of CASCADING-CUT may have marked a node]

After Decrease-key, there are at most $t(H) + c$ trees, and at most $m(H) - c + 2$ marked nodes.

Thus the difference in potential cost is:

$$\begin{aligned}
 &= [t(H) + c + 2(m(H) - c + 2)] - [t(H) + 2m(H)] \\
 &= 4 - c
 \end{aligned}$$

Amortized cost:

$$O(c) + 4 - c = O(1)$$

7.) Deleting a node

It is easy to delete a node from an n -node Fibonacci heap in $O(D(n))$ amortized time, as is done by the following pseudocode. We assume that there is no key value of $-\infty$ currently in the Fibonacci heap.

FIB-HEAP-DELETE(H, x)

```

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

```

Deleting a node(Analysis)

The amortized time of FIB-HEAP-DELETE is $O(\lg n)$. [As $D(n) = \lg n$]

Application of Fibonacci heap

Fibonacci heap has a wide range of applications in computer science, particularly in algorithms and data processing related to:

- **Graph algorithms:** It is used in algorithms such as Dijkstra's shortest path algorithm and Prim's algorithm for finding minimum spanning trees. These algorithms are essential in network routing, such as finding the shortest path in a transportation network or designing efficient communication networks.
- **Network flow algorithms:** It is used in algorithms such as the Edmonds-Karp algorithm for finding maximum flow in a network.
- **Optimization Problems:** In fields like operations research, Fibonacci heaps can be employed for solving optimization problems. For example, in supply chain management, they can help find optimal routes for delivery trucks or minimize the cost of production.

- **Memory Management:** In real-time systems, such as those used in aerospace or automotive industries, memory management is crucial. Fibonacci heaps can help allocate and deallocate memory efficiently, which is important for real-time response and safety-critical systems.
- **Data compression:** It is used in Huffman encoding for compressing data. This is relevant in industries dealing with data storage and transmission, such as telecommunications and data centers.

Skip list in Data structure

Why Study Skip List

- ▶ The skip list is solid and trustworthy.
- ▶ To add a new node to it, it will be inserted extremely quickly.
- ▶ Easy to implement compared to the hash table and binary search tree
- ▶ The number of nodes in the skip list increases, and the possibility of the worst-case decreases
- ▶ Requires only $\Theta(\log n)$ time in the average case for all operations.
- ▶ Finding a node in the list is relatively straightforward.

What is a skip list?

A skip list is a probabilistic data structure. The skip list is used to store a sorted list of elements or data with a linked list. It allows the process of the elements or data to view efficiently. In one single step, it skips several elements of the entire list, which is why it is known as a skip list.

The skip list is an extended version of the linked list. It allows the user to search, remove, and insert the element very quickly. It consists of a base list that includes a set of elements which maintains the link hierarchy of the subsequent elements.

Skip list structure

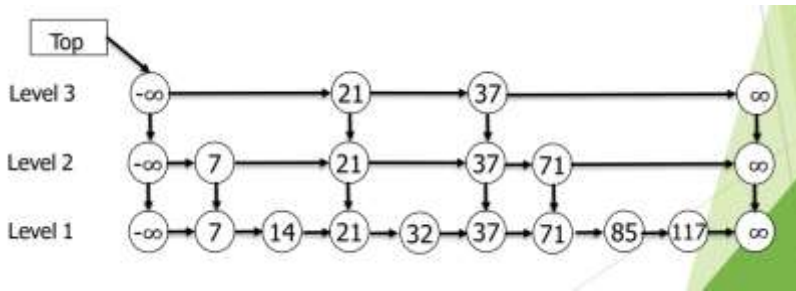
It is built in two layers: **The lowest layer and Top layer.**

The lowest layer of the skip list is a common sorted linked list, and the top layers of the skip list are like an "express line" where the elements are skipped.

Skip list is a data structure used for maintaining a set of keys in sorted order.

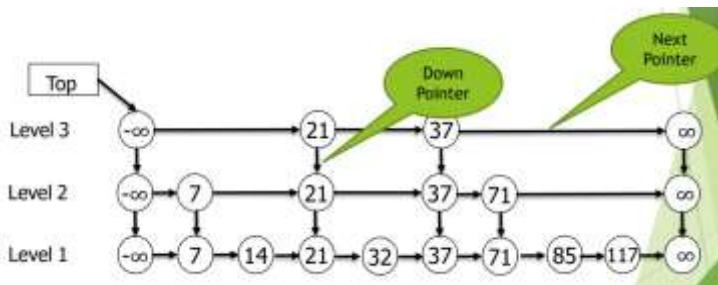
• Rules of Skip List

- It consists of several levels.
- In skip list all keys are appear in level 1.
- Each level of the skip list is a sorted list.
- In skip list if a key x appears in level i , then it also appears in all levels below i .



More Rules

- An element in level i points (via down pointer) to the element with the same key in the level below.
- In each level the keys $-\infty$ and ∞ appear.
- Top points to the smallest element in the highest level.



Finding an element with key x

$p = \text{top}$

While(1){

while ($p \rightarrow \text{next} \rightarrow \text{key} < x$)

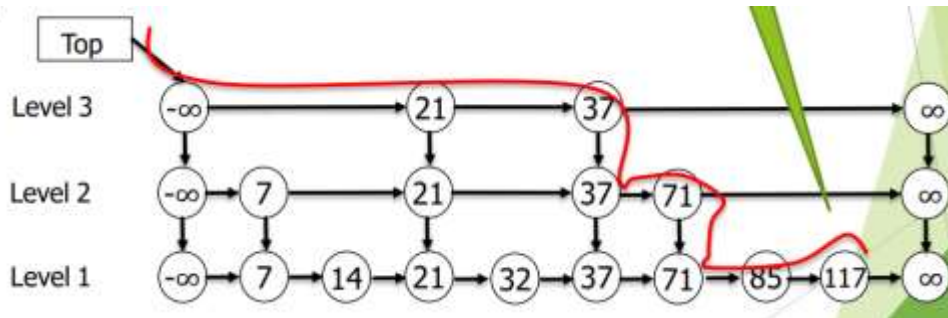
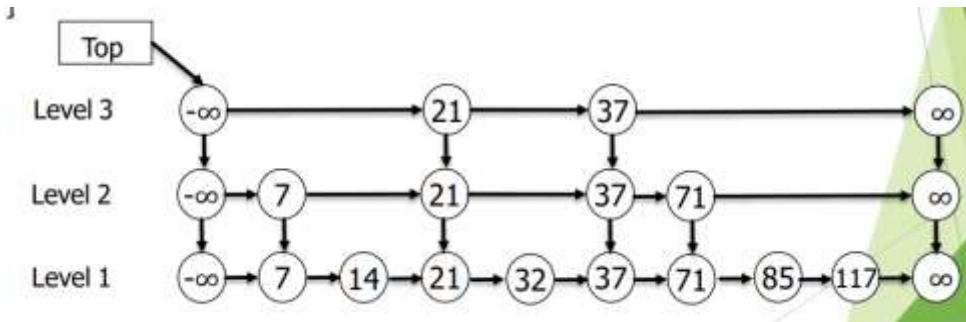
$p = p \rightarrow \text{next};$

If ($p \rightarrow \text{down} == \text{NULL}$)

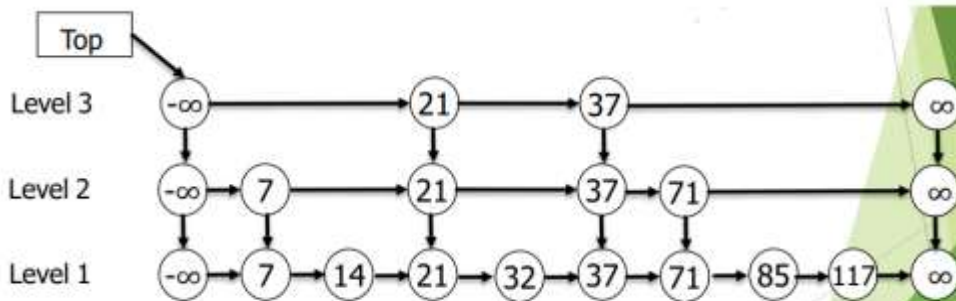
return $p \rightarrow \text{next};$

$p = p \rightarrow \text{down} ; \}$

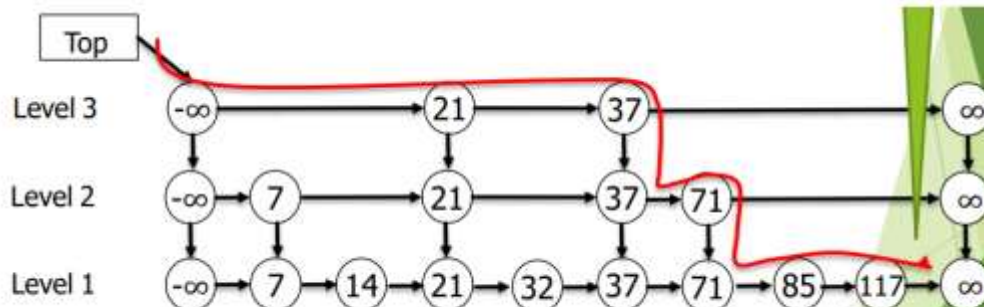
Find
117



Find
118



(Note: Observe that we return x , if exists, or $\text{succ}(x)$ if x is not in the SkipList)



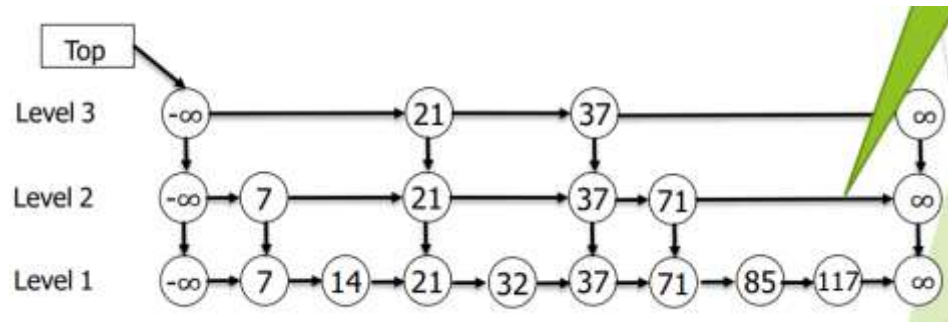
(Note: Observe that we return x , if exists, or $\text{succ}(x)$ if x is not in the Skip List)

It will return 117.

Inserting new element X:

Do find(x), and insert x to the appropriate places in the kth level

Example - inserting 119 at k=2

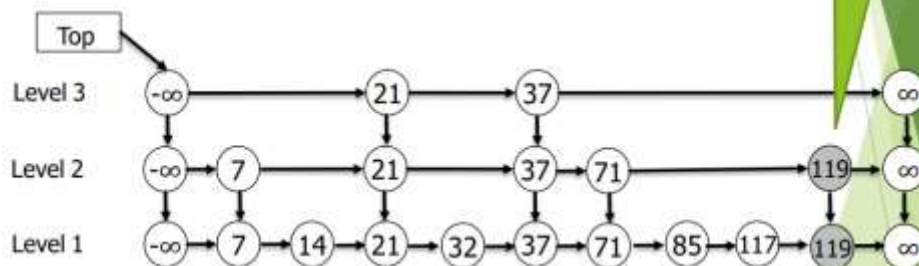


- Inserting new element X

Do find(x), and insert x to the appropriate places in the kth level

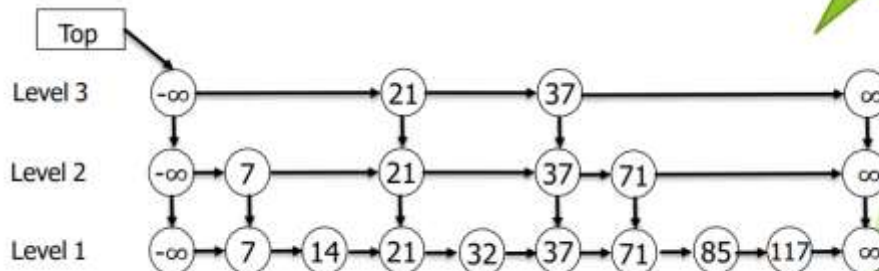
Example - inserting 119 at k=2

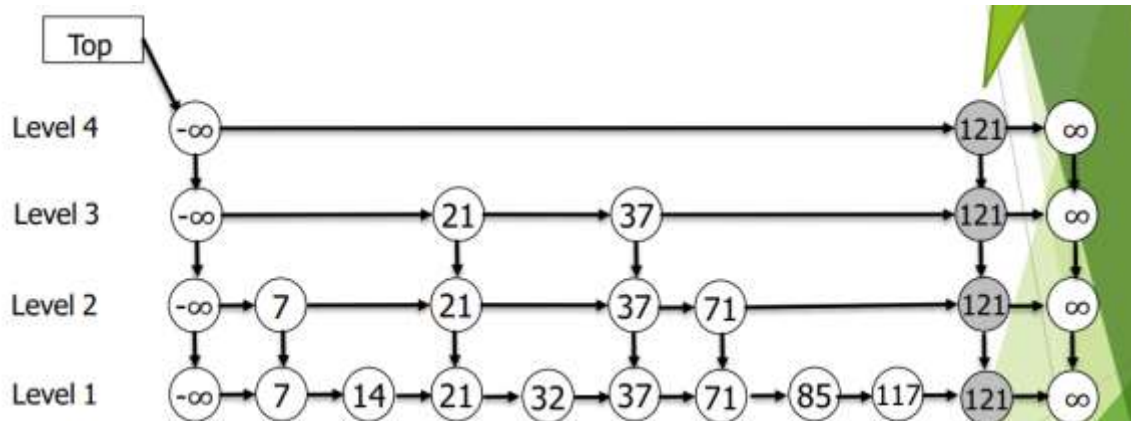
Insertion of 119
at level(k)=2 done
successfully



Example - inserting 121 at k=4

Insert 121 at
level(k)=4



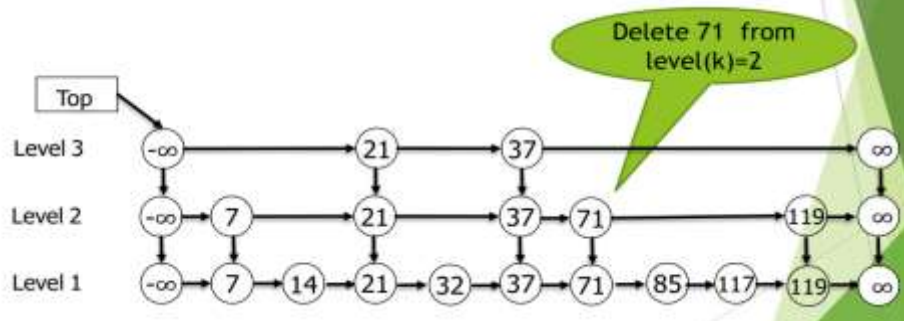


[Note: If k is larger than the current number of levels, add new levels and update the top pointer]

• Deleting a key X

- Apply Find x in all the levels, and delete the key X by using the standard 'delete from a linked list' method.
- If one or more of the upper levels are empty, remove them and update the top pointer.

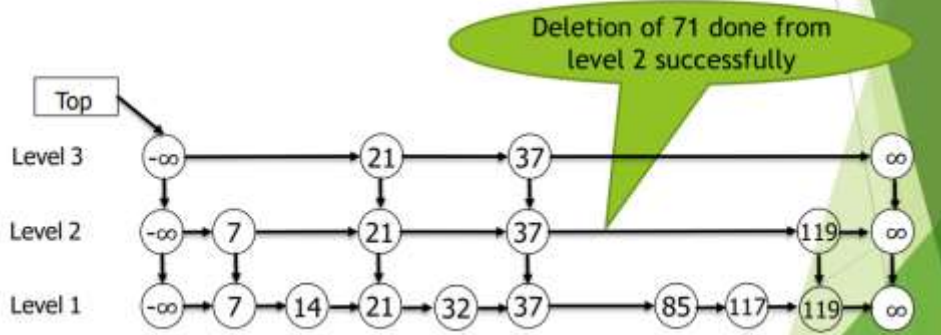
Example : Delete 71 from level 2



• Deleting a key X

- Apply Find x in all the levels, and delete the key X by using the standard 'delete from a linked list' method.
- If one or more of the upper levels are empty, remove them and update the top pointer.

Example : Delete 71 from level 2



Complexity table of the Skip list

S. No	Complexity	Average case	Worst case
1.	Access complexity	$O(\log n)$	$O(n)$
2.	Search complexity	$O(\log n)$	$O(n)$
3.	Delete complexity	$O(\log n)$	$O(n)$
4.	Insert complexity	$O(\log n)$	$O(n)$
5.	Space complexity	-	$O(n \log n)$

Working of the Skip list

Let's take an example to understand the working of the skip list. In this example, we have 14 nodes, such that these nodes are divided into two layers, as shown in the diagram.

The lower layer is a common line that links all nodes, and the top layer is an express line that links only the main nodes, as you can see in the diagram.

Suppose you want to find 47 in this example. You will start the search from the first node of the express line and continue running on the express line until you find a node that is equal to 47 or more than 47.

You can see in the example that 47 does not exist in the express line, so you search for a node of less than 47, which is 40. Now, you go to the normal line with the help of 40, and search the 47, as shown in the diagram.

Skip List Basic Operations

There are the following types of operations in the skip list.

- **Insertion operation:** It is used to add a new node to a particular location in a specific situation.
- **Deletion operation:** It is used to delete a node in a specific situation.
- **Search Operation:** The search operation is used to search a particular node in a skip list.

Algorithm of the insertion operation

1. Insertion (L, Key)
2. local update $[0 \dots \text{Max_Level} + 1]$
3. $a = L \rightarrow \text{header}$
4. **for** $i = L \rightarrow \text{level down to } 0$ **do**.
5. **while** $a \rightarrow \text{forward}[i] \rightarrow \text{key} < \text{forward}[i]$
6. $\text{update}[i] = a$
- 7.
8. $a = a \rightarrow \text{forward}[0]$

9. $lvl = \text{random_Level}()$
10. **if** $lvl > L \rightarrow \text{level}$ **then**
11. **for** $i = L \rightarrow \text{level} + 1$ **to** lvl **do**
12. $\text{update}[i] = L \rightarrow \text{header}$
13. $L \rightarrow \text{level} = lvl$
- 14.
15. $a = \text{makeNode}(lvl, \text{Key}, \text{value})$
16. **for** $i = 0$ **to** level **do**
17. $a \rightarrow \text{forward}[i] = \text{update}[i] \rightarrow \text{forward}[i]$
18. $\text{update}[i] \rightarrow \text{forward}[i] = a$

Algorithm of deletion operation

Deletion (L, Key)

1. local $\text{update} [0 \dots \text{Max_Level} + 1]$
2. $a = L \rightarrow \text{header}$
3. **for** $i = L \rightarrow \text{level}$ **down to** 0 **do**.
4. **while** $a \rightarrow \text{forward}[i] \rightarrow \text{key} \neq \text{Key}$
5. $\text{update}[i] = a$
6. $a = a \rightarrow \text{forward}[0]$
7. **if** $a \rightarrow \text{key} = \text{Key}$ **then**
8. **for** $i = 0$ **to** $L \rightarrow \text{level}$ **do**
9. **if** $\text{update}[i] \rightarrow \text{forward}[i] = a$ **then break**
10. $\text{update}[i] \rightarrow \text{forward}[i] = a \rightarrow \text{forward}[i]$
11. $\text{free}(a)$
12. **while** $L \rightarrow \text{level} > 0$ **and** $L \rightarrow \text{header} \rightarrow \text{forward}[L \rightarrow \text{level}] = \text{NIL}$ **do**
13. $L \rightarrow \text{level} = L \rightarrow \text{level} - 1$

Algorithm of searching operation

1. Searching (L, SKey)
2. $a = L \rightarrow \text{header}$
3. loop invariant: $a \rightarrow \text{key level down to } 0$ **do**.
4. **while** $a \rightarrow \text{forward}[i] \rightarrow \text{key} \neq \text{SKey}$
5. $a = a \rightarrow \text{forward}[0]$
6. **if** $a \rightarrow \text{key} = \text{SKey}$ **then return** $a \rightarrow \text{value}$
7. **else return** failure

Advantages of the Skip list

1. If you want to insert a new node in the skip list, then it will insert the node very fast because there are no rotations in the skip list.
2. The skip list is simple to implement as compared to the hash table and the binary search tree.
3. It is very simple to find a node in the list because it stores the nodes in sorted form.
4. The skip list algorithm can be modified very easily in a more specific structure, such as indexable skip lists, trees, or priority queues.
5. The skip list is a robust and reliable list.

Disadvantages of the Skip list

1. It requires more memory than the balanced tree.
2. Reverse searching is not allowed.
3. The skip list searches the node much slower than the linked list.

Applications of the Skip list

1. It is used in distributed applications, and it represents the pointers and system in the distributed applications.
2. It is used to implement a dynamic elastic concurrent queue with low lock contention.
3. It is also used with the QMap template class.
4. The indexing of the skip list is used in running median problems.
5. The skip list is used for the delta-encoding posting in the Lucene search.

List of application and framework that use skip list

- Apache Portable Runtime implements skip lists.
- MemSQL uses lock-free skip lists as its prime indexing structure for its database technology.
- MuQSS built on skip lists as a cpu scheduler.
- Cyrus IMAP server offers a "skiplist" backend DB implementation
- Lucene uses skip lists to search delta-encoded posting lists in logarithmic time.

Summary

1. A skip list is a data structure for maps that uses a randomized insertion algorithm
2. In a skip list with n entries
 - The expected space used is $O(n)$
 - The expected search, insertion and deletion time is $O(\log n)$
 - Using a more complex probabilistic analysis, one can show that these performance bounds also hold with high probability
 - Skip lists are fast and simple to implement in practice

Comparison of tries with hash table

1. Looking up data in a trie is faster in worst case as compared to imperfect hash table.
2. There are no collisions of different keys in a trie.
3. In trie if single key is associated with more than one value then it resembles buckets in hash table.
4. There is no hash function in trie.
5. Sometimes data retrieval from tries is very much slower than hashing.
6. Representation of keys a string is complex. For example, representing floating point numbers using strings is really complicated in tries.
7. Tries always take more space than hash tables.
8. Tries are not available in programming tool it. Hence implementation of tries has to be done from scratch.

Trie

Why study Trie

In computer science, a trie, also called digital tree or prefix tree, is a type of k-ary search tree, a tree data structure used for locating specific keys from within a set. These keys are most often strings, with links between nodes defined not by the entire key, but by individual characters. In order to access a key (to recover its value, change it, or remove it), the trie is traversed depth-first, following the links between nodes, which represent each character in the key.

Need of studying Trie:

Trie data structures are commonly used in predictive text or autocomplete dictionaries, and approximate matching algorithms. Tries enable faster searches, occupy less space, especially when the set contains large number of short strings, thus used in spell checking, hyphenation applications and longest prefix match algorithms. However, if storing dictionary words is all that is required (i.e. there is no need to store metadata associated with each word), a minimal deterministic acyclic finite state automaton

(DAFSA) or radix tree would use less storage space than a trie. This is because DAFSAs and radix trees can compress identical branches from the trie which correspond to the same suffixes (or parts) of different words being stored. String dictionaries are also utilized in natural language processing, such as finding lexicon of a text corpus.

Trie's Properties:

- A multi-way tree.
- Each node has from 1 to n children.
- Each edge of the tree is labeled with a character.

- Each leaf nodes corresponds to the stored string,

which is a concatenation of characters on a path

from the root to this node.

Trie Introduction

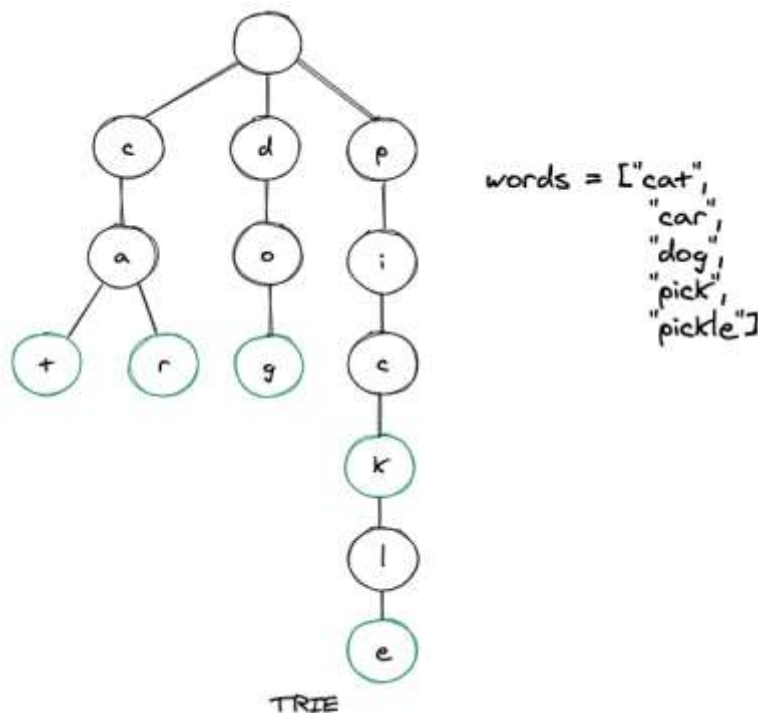
A **Trie is an advanced data structure** that is sometimes also known as **prefix tree or digital tree**. It is a tree that stores the data in an ordered and efficient way. We generally **use trie's to store strings**. Each node of a trie can have as many as 26 references (pointers).

Each node of a trie consists of two things:

- A character
- A boolean value is used to implement whether this character represents the end of the word.

Tries in general are used to store English characters, hence each character can have 26 references. Nodes in a trie do not store entire keys, instead, they store a part of the key(usually a character of the string). When we traverse down from the root node to the leaf node, we can build the key from these small parts of the key.

Let's build a trie by inserting some words in it. Below is a pictorial representation of the same, we have 5 words, and then we are inserting these words one by one in our trie.



an have at most 26 references. Tries are not balanced in nature, unlike AVL trees.

Why use Trie Data Structure?

When we talk about the **fastest ways to retrieve values** from a data structure, **hash tables generally comes to our mind**. Though very efficient in nature but still very less talked about as when compared to hash tables, **trie's are much more efficient than hash tables** and also they possess several advantages over the same. Mainly:

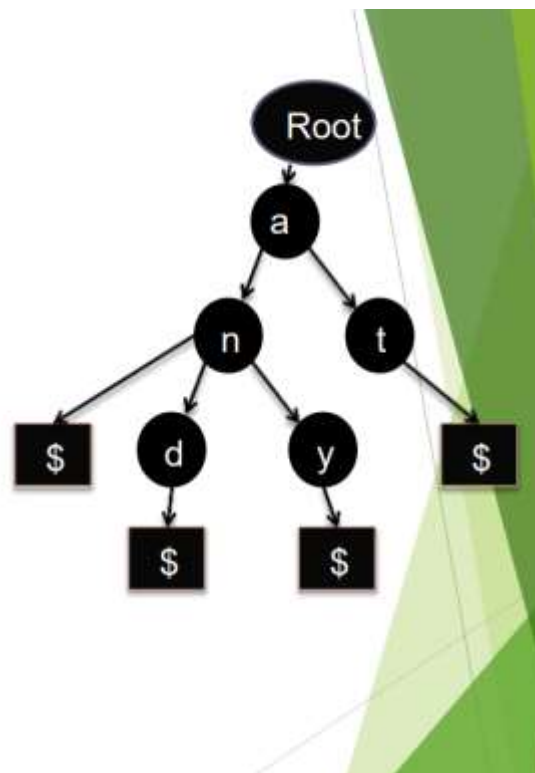
- There won't be any collisions hence making the worst performance better than a hash table that is not implemented properly.
- No need for hash functions.
- Lookup time for a string in trie is $O(k)$ where **k = length of the word**.
- It can take even less than $O(k)$ time when the word is not there in a trie.

Types:

- Standard Tries
- Compressed/Compact Tries
- Suffix Tries

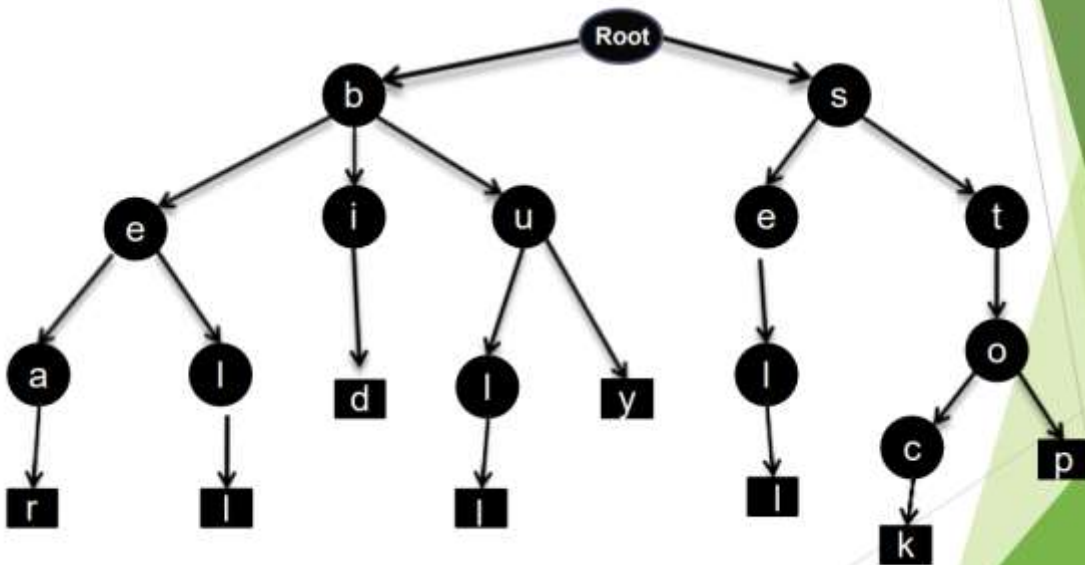
Standard Trie:

- The standard trie for a set of strings S is an ordered tree such that:
 - Each node but the root is labeled with a character.
 - The children of a node are alphabetically ordered.
 - The paths from the external nodes to the root yield the strings of S .
 - Example :Strings = {an, and, any, at}
 - append a special termination symbol "\$"



Standard Trie:

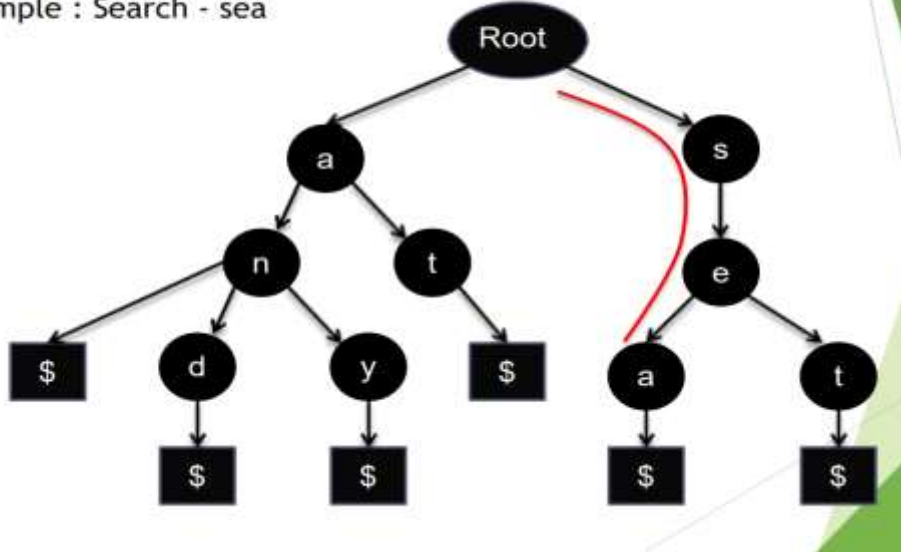
- Example: Standard trie for the set of strings
 $S = \{ \text{bear, bell, bid, bull, buy, sell, stock, stop} \}$



Standard Trie Searching

Search hit: Node where search ends has a \$ symbol

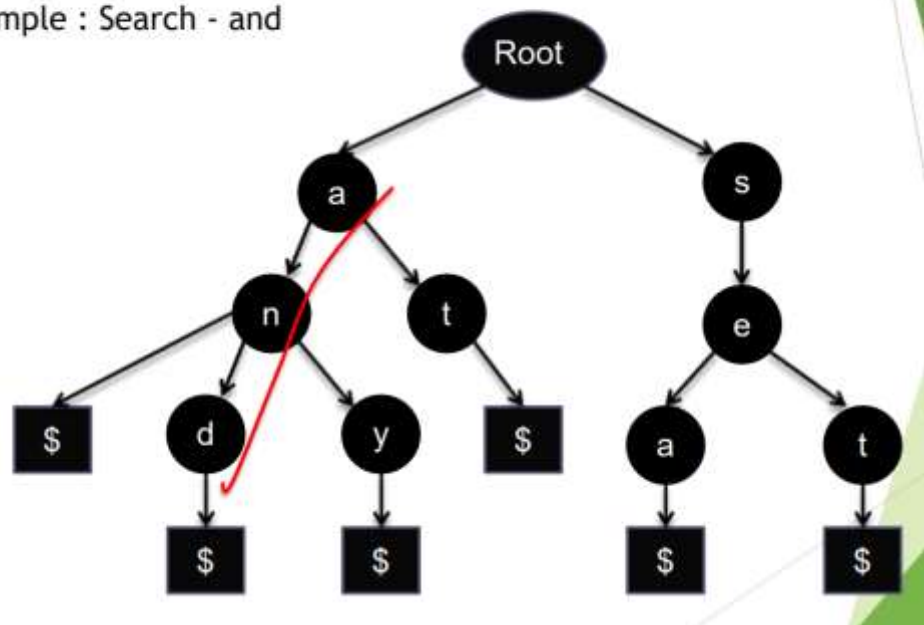
Example : Search - sea



Standard Trie Searching

Search hit: Node where search ends has a \$ symbol

Example : Search - and



Standard Trie Deletion

Three cases

Case 1: Word not found...!

then Return False

Case 2: Word exists as a stand alone word.

part of any other word

does not a part of any other word

Case 3: Word exists as a prefix of another word.

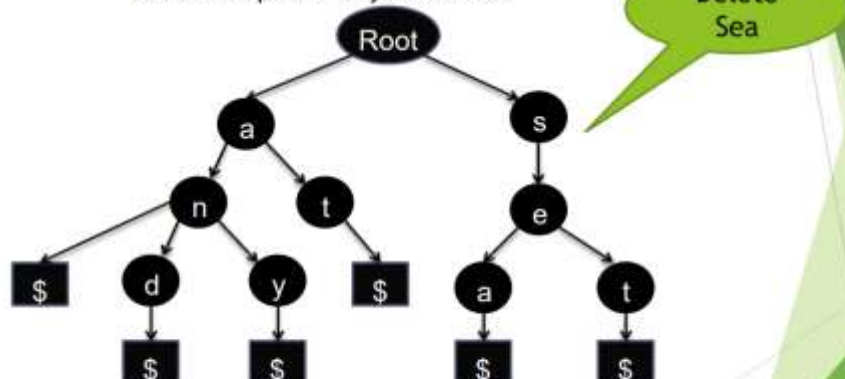
Standard Trie Deletion

- Three cases

Case 2: Word exists as a stand alone word.

part of any other word

does not a part of any other word



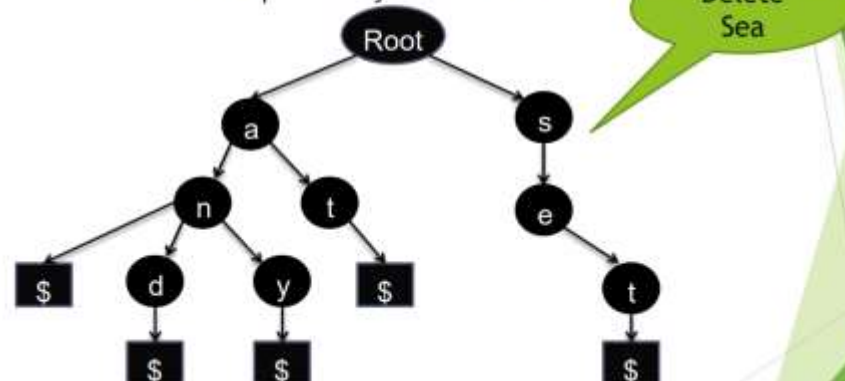
Standard Trie Deletion

- Three cases

Case 2: Word exists as a stand alone word.

part of any other word

does not a part of any other word



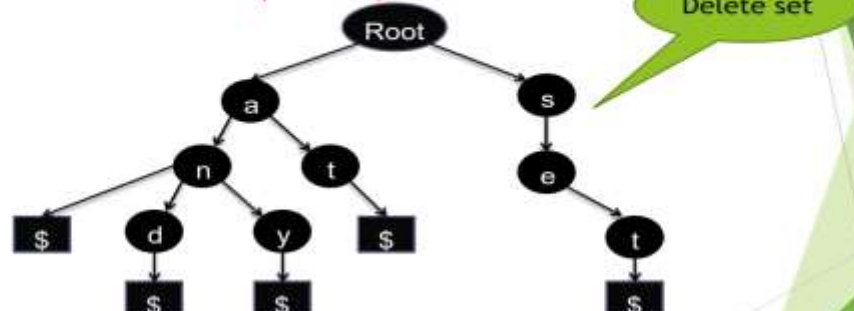
Standard Trie Deletion

- Three cases

Case 2: Word exists as a stand alone word.

part of any other word

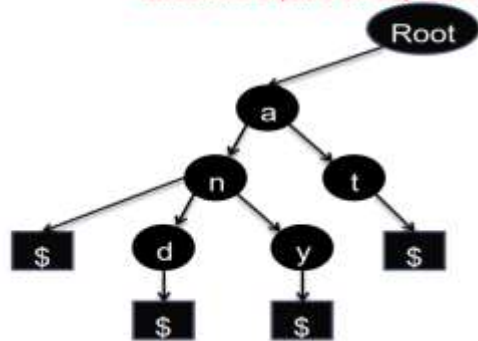
does not a part of any other word



Standard Trie Deletion

- Three cases

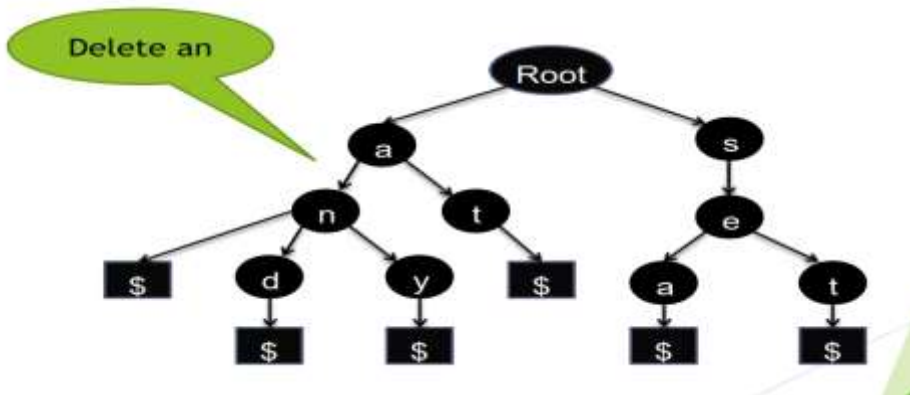
Case 2: Word exists as a stand alone word.
part of any other word
does not a part of any other word



Standard Trie Deletion

- Three cases

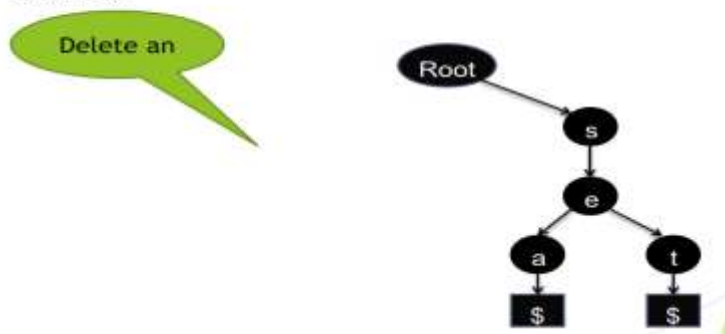
Case 3: Word exists as a prefix of any other word.
Delete - an



Standard Trie Deletion

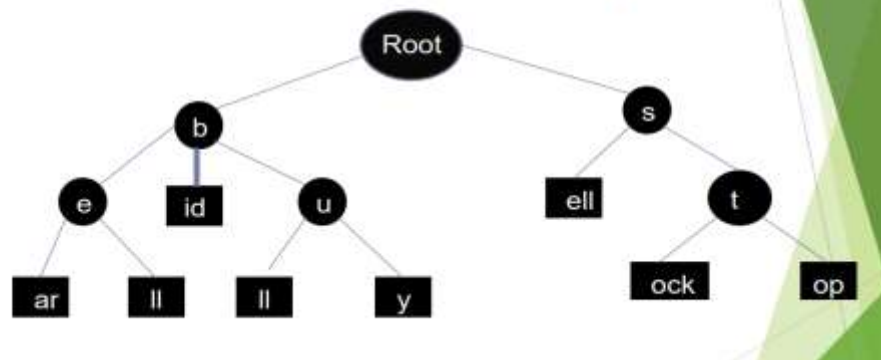
- Three cases

Case 3: Word exists as a prefix of any other word.
Delete - an



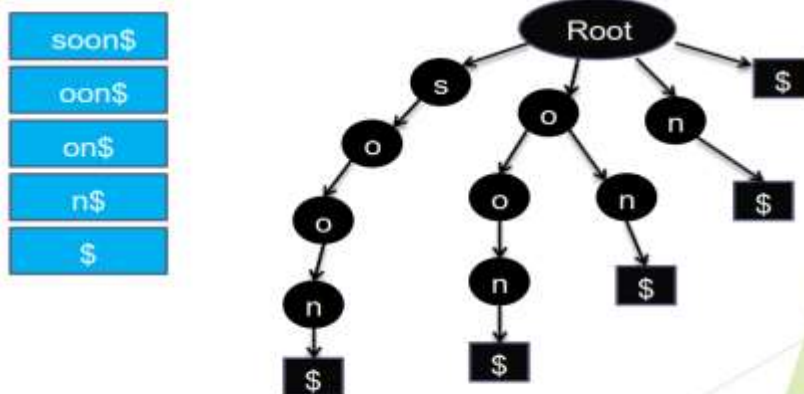
Compressed Trie

- Tries with nodes of degree at least 2
- Obtained by standard tries by compressing chains of redundant nodes
- Example: $S = \{ \text{bear, bell, bid, bull, buy, sell, stock, stop} \}$



Suffix Trie

- A suffix trie is a compressed trie for all the suffixes of a text.
- Suffix trie are a space-efficient data structure to store a string that allows many kinds of queries to be answered quickly.



Implementing Tries:

We will implement the Trie data structure in Java language.

Trie Node Declaration:

```
class TrieNode {  
  
    boolean isEndOfWord;  
  
    TrieNode children[];  
  
    public TrieNode(){
```

```
isEndOfWord = false;
```

```
children = new TrieNode[26]; } }
```

Note that we have two fields in the above TrieNode class as explained earlier, the boolean isEndOfWord keyword and an array of Trie nodes named children. Now let's initialize the root node of the trie class.

```
TrieNode root;
```

```
public Trie() {
```

```
    root = new TrieNode(); }
```

There are two key functions in a trie data structure, these are:

- **Search**
- **Insert**

Insert in trie:

When we insert a character(part of a key) into a trie, we start from the root node and then search for a reference, which corresponds to the first key character of the string whose character we are trying to insert in the trie. Two scenarios are possible:

- A reference exists, if, so then we traverse down the tree following the reference to the next children level.
- A reference does not exist, then we create a new node and refer it with parents reference matching the current key character. We repeat this step until we get to the last character of the key, then we mark the current node as an end node and the algorithm finishes.

Consider the code snippet below:

```
public void insert(String word) {
```

```
    TrieNode node = root;
```

```
    for (char c : word.toCharArray()) {
```

```
        if (node.children[c-'a'] == null) {
```

```
            node.children[c-'a'] = new TrieNode();
```

```
        }
```

```
        node = node.children[c-'a'];
```

```
}  
  
node.isEndOfWord = true;  
  
}
```

Search in trie:

A key in a trie is stored as a path that starts from the root node and it might go all the way to the leaf node or some intermediate node. If we want to search a key in a trie, we start with the root node and then traverses downwards if we get a reference match for the next character of the key we are searching, then there are two cases:

1. A reference of the next character exists, hence we move downwards following this link, and proceed to search for the next key character.
2. A reference does not exist for the next character. If there are no more characters of the key present and this character is marked as `isEndOfWord = true`, then we return **true**, implying that we have found the key. Otherwise, two more cases are possible, and in each of them we return **false**. These are:
 1. There are key characters left in the key, but we cannot traverse down as the path is terminated, hence the key doesn't exist.
 2. No characters in the key are left, but the last character is not marked as `isEndOfWord = false`. Therefore, the search key is just the prefix of the key we are trying to search in the trie.

Consider the code snippet below:

```
public boolean search(String word) {  
  
    return isMatch(word, root, 0, true);  
  
}  
  
public boolean startsWith(String prefix) {  
  
    return isMatch(prefix, root, 0, false);  
  
}  
  
public boolean isMatch( String s, TrieNode node, int index, boolean isFullMatch) {  
  
    if (node == null)  
  
        return false;  
  
    if (index == s.length())
```

```

return !isFullMatch || node.isEndOfWord;

return isMatch(s, node.children[s.charAt(index) - 'a'], index + 1, isFullMatch);
}

```

The method `startsWith()` is used to find if the desired key prefix is present in the trie or not. Also, both the `search()` and `startsWith()` methods make use of `isMatch()` method.

Tries and Web Search Engines

- The index of a search engine (collection of all searchable words) is stored into a compressed trie
- Each leaf of the trie is associated with a word and has a list of pages (URLs) containing that word, called occurrence list
- The trie is kept in internal memory
- The occurrence lists are kept in external memory and are ranked by relevance
- Boolean queries for sets of words (e.g., Java and coffee) correspond to set operations (e.g., intersection) on the occurrence lists
- Additional information retrieval techniques are used, such as – stopword elimination (e.g., ignore “the” “a” “is”) – stemming (e.g., identify “add” “adding” “added”) – link analysis (recognize authoritative pages)

University Questions

- Q1. Describe the properties of Red-Black tree.(10 marks)
- Q2. Write the properties of Red-Black Tree. Illustrate with an example, how the keys are inserted in an empty red-black tree. (10 marks)
- Q3. Discuss the various cases for insertion of key in red-black tree for given sequence of key in an empty red-black tree- {15,13,12,16,19,23,5,8}. Also show that a red-black tree with n internal nodes has height at most $2\lg(n+1)$.(10 marks)
- Q4. Explain about double black node problem in RB Tree.(2 Marks)
- Q5. 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 $t=3$ (7 Marks)
- Q6. Insert the nodes 15, 13, 12, 16, 19, 23, 5, 8 in empty red-black tree and delete in the reverse order of insertion (10 marks)
- Q7. Prove that if $n \geq 1$, then for any n-key B-Tree of height h and minimum degree $t \geq 2$, $h \leq \log_t((n+1)/2)$. (5 marks)
- Q8. Define a B-Tree of order m. Explain the searching operation in a B-Tree(5 marks)
- Q9. Discuss the properties of binomial trees(Heap).(5 marks)
- Q10. Construct the binomial heap for the following sequence of number 7, 2, 4, 17, 1, 11, 6, 8, 15. (7 marks)
- Q11. What is Binomial Heap? Write down the algorithm for Decrease key operation in Binomial Heap also write its time complexity. (10 marks)
- Q12. Explain the algorithm to delete a given element in a binomial Heap. Give an example for the same. (7 marks)

- Q13. Explain and write an algorithm for union of two binomial heaps and write its time complexity (10 marks)
- Q14. Explain the algorithm to extract the minimum elements in a binomial Heap. Give an example for the same. (5 marks)
- Q15. What is Fibonacci Heap? Discuss the application of Fibonacci Heap (5 marks)
- Q16. What is Fibonacci heap? (2 marks)
- Q17. Explain CONSOLIDATE operation with suitable example for Fibonacci heap. (5 marks)
- Q18. What are the various differences in Binomial and Fibonacci Heap? Explain (5 Marks)
- Q19. Discuss following operations of Fibonacci heap:
- i Insert
 - ii Extract-Min (5 marks)
- Q20: Discuss Skip list and its operation (Univ Exam 2022 – 23) (2 marks)
- Q21: Explain skip list in brief (Univ Exam 2020-21) (2 marks)
- Q22: What are the various advantages and disadvantages of Skip list. (7 marks)
- Q23: What are the various advantages and disadvantages of trie. (7 marks)
- Q24: Give any three fields where Skip list are used. (5 marks)
- Q25: Give any three fields where Trie are used. (5 marks)
- Q26: Explain the deletion operation in skip list with algorithm. (7 marks)
- Q27: What are the different types of Tries? (5 marks)
- Q28: Explain the insertion operation in skip list with algorithm. (7 marks)