FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ

# ALGORITHM ANALYSIS & DESIGN PROJECT REPORT

## MERGE SORT

ABDUSSAMET KACI

1721221007

## Question 1: Comparison of 3-way Mergesort vs 2-way mergesort both theoretically and empirically

**1) Write the pseudo-code of each algorithm and explain using your own words, briefly**

**Merge Sort 2-Way**

```
Function sort(arr[0...n-1], c[0...n-1], low, high)
    İf low <= high return     // low index equlas or lower than high index
    mid ← low + (high - low) / 2     // compute mid point
    sort(arr, c, low, mid)     // left part
    sort(arr, c, mid + 1, high)          // right part
    merge(arr, c, low, mid, high)     // merge parts


Function merge(arr[0...n-1], c[0...n-1], low, mid, high)
    copy arr[low...high] to c[low...high]
    i ← low          // index of first subarray
    j ← mid + 1      // index of second subarry
    k ← low          // index of merged subarray
    while i <= mid and j <= high do
        if c[i] <= c[j]     arr[k] ← c[i];    i ← i + 1
        else     arr[k] ← c[j];    j ← j + 1
        k ← k + 1
    while i <= mid   // copy remaining elements of left part
        arr[k] ← c[i];    k ← k + 1;   i ← i + 1
    while j <= high  // copy remaining elements of right part
        arr[k] ← c[j];    k ← k + 1;   j ← j + 1
```

**Merge Sort 3-Way**

```
Function sort(arr[0...n-1], c[0...n-1], low, high)
    İf low – high  <  2    return        // if array size 1, then do nothing
    mid1 ← low + ((high - low) / 3)  // first 1/3 part
    mid2 ← low + 2 * ((high - low) / 3) + 1   // second 1/3 part
    sort(arr, c, low, mid1)    // first 1/3 part
    sort(arr, c, mid1, mid2) // second 1/3 part
```

```
        sort(arr, c, mid2, high)   // last 1/3 part
        merge(arr, c, low, mid1, mid2, high)       // merge parts
Function merge(arr[0...n-1], c[0...n-1], low, mid1, mid2, high)
        copy arr[low...high - 1] to c[low...high - 1]
        i ← low;    j ← mid1;    k ← mid2;    l ← high
        while i < mid1 and j < mid2 and k < high do       // find the smallest element from 3 ranges
                if c[i] < c[j]
                        if c[i] < c[k]    arr[l] ← c[i];    i ← i + 1
                        else    arr[l] ← c[k];    k ← k + 1
                else
                        if c[j] < c[k]        arr[l] ← c[j];      j ← j + 1
                        else      arr[l] ← c[k];     k ← k + 1
                l ← l + 1
        while i < mid1 and j < mid2 do   // case where first and second ranges have remaining values
                if c[i] < c[j]    arr[l] ← c[i];    i ← i + 1
                else    arr[l] ← c[j];    j ← j + 1
                l ← l + 1
        while j < mid2 and k < high do   // case where second and third ranges have remaining values
                if c[j] < c[k]       arr[l] ← c[j];      j ← j + 1
                else      arr[l] ← c[k];     k ← k + 1
                l ← l + 1
        while i < mid1 and k < high do   // case where first and third ranges have remaining values
                if c[i] < c[k]       arr[l] ← c[i];      i ← i + 1
                else      arr[l] ← c[k];     k ← k + 1
                l ← l + 1
        while i < mid1 do          // copy remaining values from first part
                arr[l] ← c[i];      l ← l + 1;    i ← i + 1
        while j < mid2 do          // copy remaining values from second part
                arr[l] ← c[j];      l ← l + 1;    j ← j + 1
        while k < high do          // copy remaining values from third part
                arr[l] ← c[k];      l ← l + 1;    k ← k + 1
```

**2) Write and solve the recurrence relation for the number of key comparisons made by each of those algorithms in the worst case**

**Merge Sort 2 Way**

C(n) = 2C(n/2) + n - 1 , for n > 1, C(1) = 0, n $= 2^k$

= 2C( $2^{k-1}$) + $2^k$ - 1

= 2[2C($2^{k-2}$) + $2^{k-1}$ - 1] + $2^k$ - 1 = $2^2$C($2^{k-2}$) + $2^k$ – 2 + $2^k$ - 1

= $2^2$[2C($2^{k-3}$) + $2^{k-2}$ - 1] + 2*$2^k$ - 2 - 1= $2^3$C($2^{k-3}$) + $2^k$ - $2^2$+ 2*$2^k$ – 2 - 1

...

= $2^i$C($2^{k-i}$) + i$2^i$ - ($2^i$ – 1)  $\quad$ $i = k$

= $2^k$C(1) + k$2^k$ - $2^k$ + 1  $\quad$ C(1) = 0, n = $2^k$, k = $\log_2(n)$

= k$2^k$ - $2^k$ + 1 = $n\log_2(n)$ – n + 1

= $n\log_2(n)$ – n + 1 ∈ $\theta(n\log(n))$

**Merge Sort 3 Way**

C(n) = 3C(n/3) + n - 1 , for n > 1, C(1) = 0, n $= 3^k$

= 3C( $3^{k-1}$) + $3^k$ - 1

= 3[3C($3^{k-2}$) + $3^{k-1}$ - 1] + $3^k$ - 1 = $3^2$C($3^{k-2}$) + $3^k$ – 3 + $3^k$ - 1

= $3^2$[3C($3^{k-3}$) + $3^{k-2}$ - 1] + 2*$3^k$ - 3 - 1= $3^3$C($3^{k-3}$) + $3^k$ - $3^2$+ 2*$3^k$ – 3 - 1

...

= $3^i$C($3^{k-i}$) + i$3^i$ - ($3^i$ – 1)  $\quad$ $i = k$

= $3^k$C(1) + k$3^k$ - $3^k$ + 1  $\quad$ C(1) = 0, n = $3^k$, k = $\log_3(n)$

= k$3^k$ - $3^k$ + 1 = $n\log_3(n)$ – n + 1

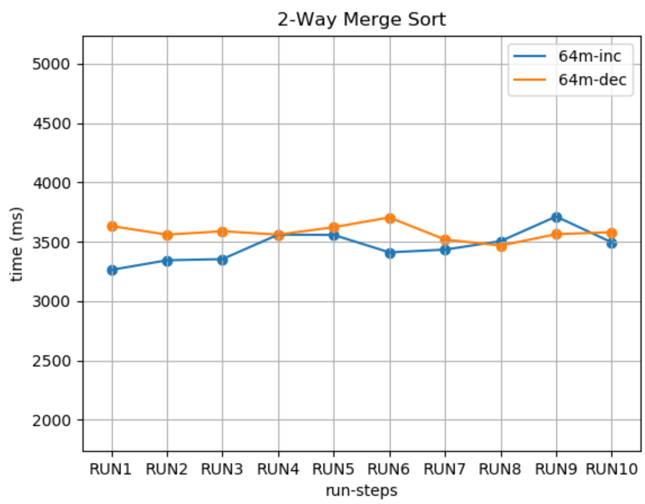
= $n\log_3(n)$ – n + 1 ∈ $\theta(n\log(n))$

**4) Investigate the performance of those algorithms on random arrays of sizes**

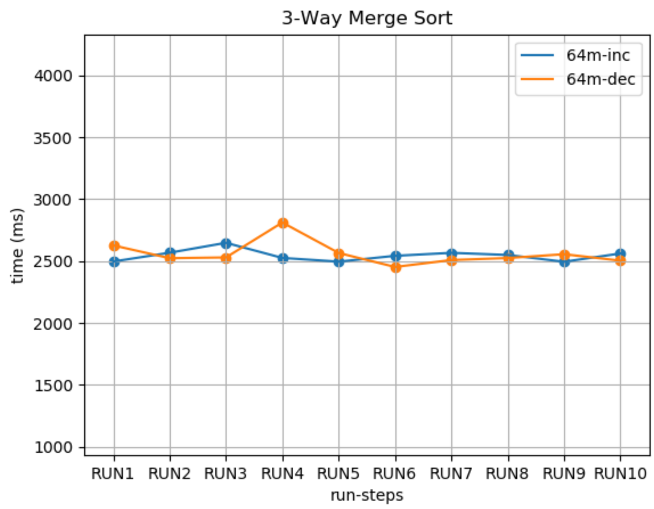
| 2-Way | seed | 4M (ms) | 8M (ms) | 16M (ms) | 32M (ms) | 64M (ms) | 128M (ms) | 64M-Inc (ms) | 64M-Dec (ms) |
|---|---|---|---|---|---|---|---|---|---|
| Run 1 | 0 | 640 | 1265 | 2658 | 5765 | 12146 | 25705 | 3263 | 3633 |
| Run 2 | 1000 | 670 | 1303 | 2624 | 5739 | 12470 | 25525 | 3344 | 3560 |
| Run 3 | 2000 | 682 | 1255 | 2661 | 5801 | 12392 | 25644 | 3354 | 3589 |
| Run 4 | 3000 | 664 | 1282 | 2789 | 5943 | 12692 | 25255 | 3560 | 3560 |
| Run 5 | 4000 | 664 | 1262 | 2852 | 6004 | 12688 | 25866 | 3558 | 3623 |
| Run 6 | 5000 | 674 | 1335 | 2886 | 5938 | 12742 | 25021 | 3411 | 3705 |
| Run 7 | 6000 | 613 | 1326 | 2943 | 5887 | 12839 | 25706 | 3434 | 3519 |
| Run 8 | 7000 | 648 | 1325 | 2850 | 5822 | 12970 | 25973 | 3505 | 3468 |
| Run 9 | 8000 | 652 | 1284 | 2814 | 5875 | 12981 | 26169 | 3713 | 3564 |
| Run 10 | 9000 | 618 | 1249 | 2765 | 5806 | 12613 | 24264 | 3495 | 3581 |
| Average | | 652.5 | 1288.6 | 2784.2 | 5858 | 12653.3 | 25512.8 | 3463.7 | 3580.2 |

| 3-Way | seed | 4M (ms) | 8M (ms) | 16M (ms) | 32M (ms) | 64M (ms) | 128M (ms) | 64M-Inc (ms) | 64M-Dec (ms) |
|---|---|---|---|---|---|---|---|---|---|
| Run 1 | 0 | 621 | 1292 | 2568 | 5394 | 11123 | 23670 | 2496 | 2626 |
| Run 2 | 1000 | 578 | 1323 | 2563 | 5465 | 11049 | 23569 | 2567 | 2523 |
| Run 3 | 2000 | 608 | 1313 | 2537 | 5474 | 11170 | 22648 | 2647 | 2529 |
| Run 4 | 3000 | 617 | 1324 | 2745 | 5585 | 11158 | 22737 | 2525 | 2811 |
| Run 5 | 4000 | 619 | 1384 | 2541 | 5617 | 11044 | 22844 | 2495 | 2565 |
| Run 6 | 5000 | 620 | 1273 | 2630 | 5417 | 10762 | 22972 | 2542 | 2452 |
| Run 7 | 6000 | 627 | 1301 | 2622 | 5476 | 11035 | 22501 | 2566 | 2508 |
| Run 8 | 7000 | 679 | 1222 | 2610 | 5260 | 11073 | 22280 | 2549 | 2524 |
| Run 9 | 8000 | 622 | 1260 | 2572 | 5295 | 11008 | 22363 | 2494 | 2555 |
| Run 10 | 9000 | 597 | 1218 | 2632 | 5287 | 11006 | 21919 | 2561 | 2503 |
| Average | | 618.8 | 1291 | 2602 | 5427 | 11042.8 | 22750.3 | 2544.2 | 2559.6 |

**5) Compare the performance of the algorithms on increasing and decreasing arrays of the size 64 M only**



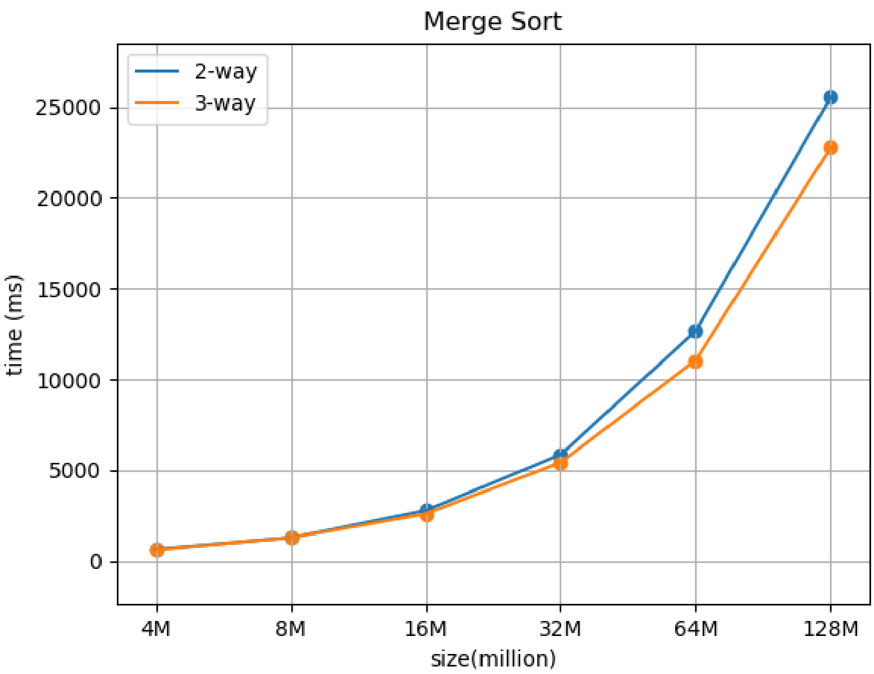

Graph 1




Graph 2

**6) Compare these two algorithms visually on a scatter plot using the average results obtained for the random arrays of sizes 4M to 128M**

Graph 3



**7) According to the results, what can you say about the theoretical assertions about each of the algorithm's efficiency?**

For 64 million array size, increasing and decreasing arrays are created and then arrays are sorted with 2-way and 3-way Merge Sort Algorithms. According to graph 1 and graph 2, increasing and decreasing arrays are sorted at similar time. This situation is same for both algorithms.

Random arrays with between 4 million to 128 million are created. According to gragh 3, 3-Way Merge Sort Algorithm works less time than 2-Way Merge Sort Algorithm. Because, 3-Way Merge Sort's time efficiency is $\mathrm{nlog}_3(\mathrm{n})$ and 2-Way Merge Sort's time efficieny is $\mathrm{nlog}_2(\mathrm{n})$. So, we can say that 3-Way Merge Sort is more efficient for big array size. This result is as expected with theoretical assertions.

**8) Repeat all the analysis for the Bottom-up mergesort. Does it provide any performance gain over ordinary mergesort?**

## 8.1) Pseudo Code

**Bottom-Up Merge Sort**

**Function** sort(arr[0...n-1])

n ← size of arr

create c[0...n-1]

len ← 1

**while** len < n **do** // merge subarrays size 1, size = 2, size = 4, ...

low ← 0

**while** low < n – len **do** // pick starting point of different subarrays of current size

mid ← low + len – 1 // assign mid point

high ← min(low + 2*len – 1, n - 1) // assign high point

merge(arr, c, low, mid, high)

low ← low + 2 * len

len ← len * 2

**NOTE:** Merge Algorithm is same with 2-Way Merge Sort.

## 8.2) Time Complexity

The time complexity of Bottom Up Algorithm is same with 2-Way Merge Sort Algorithm.

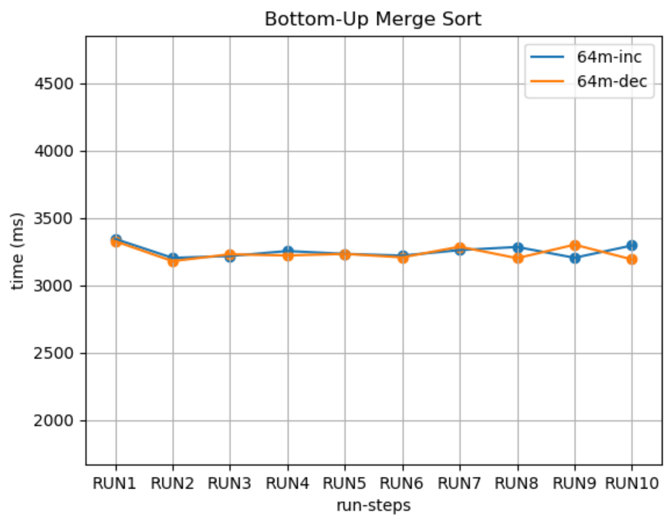
**Time Complexity:** $\mathrm{nlog}_2(\mathrm{n}) - n + 1 \in \theta(\mathrm{nlog}(\mathrm{n}))$

Bottom Up Merge Sort is an iterative merge sort. It does not need to store function calls in the stack. So, Bottom Up Merge Sort has more space efficient at %25.

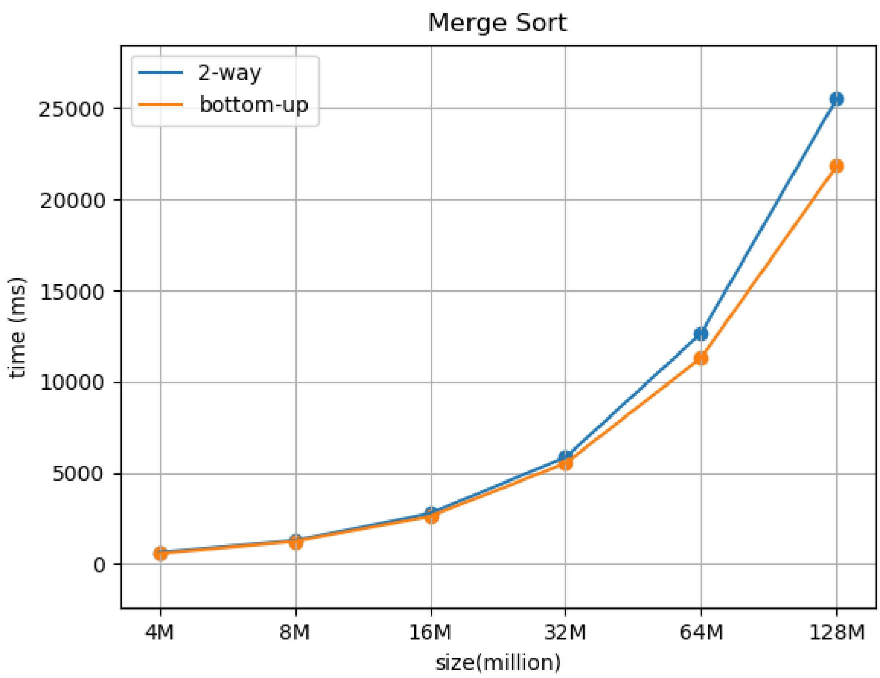
### 8.4) Bottom-Up Merge Sort Performances

| Bottom-up | seed | 4M (ms) | 8M (ms) | 16M (ms) | 32M (ms) | 64M (ms) | 128M (ms) | 64M-Inc (ms) | 64M-Dec (ms) |
|---|---|---|---|---|---|---|---|---|---|
| Run 1 | 0 | 572 | 1245 | 2650 | 5579 | 11552 | 24495 | 3343 | 3326 |
| Run 2 | 1000 | 561 | 1252 | 2582 | 5621 | 11783 | 24225 | 3202 | 3178 |
| Run 3 | 2000 | 594 | 1225 | 2575 | 5534 | 11339 | 22263 | 3216 | 3229 |
| Run 4 | 3000 | 601 | 1256 | 2581 | 5629 | 10949 | 21238 | 3253 | 3220 |
| Run 5 | 4000 | 589 | 1237 | 2603 | 5552 | 11057 | 21137 | 3232 | 3232 |
| Run 6 | 5000 | 585 | 1234 | 2556 | 5584 | 11171 | 20753 | 3219 | 3206 |
| Run 7 | 6000 | 600 | 1232 | 2652 | 5621 | 11107 | 21077 | 3260 | 3285 |
| Run 8 | 7000 | 586 | 1262 | 2767 | 5388 | 11356 | 21101 | 3283 | 3200 |
| Run 9 | 8000 | 586 | 1292 | 2603 | 5393 | 11412 | 20897 | 3203 | 3300 |
| Run 10 | 9000 | 601 | 1214 | 2641 | 5401 | 11361 | 21027 | 3294 | 3191 |
| Average | | 587.5 | 1244.9 | 2621 | 5530.2 | 11308.7 | 21821.3 | 3250.5 | 3236.7 |

### 8.5)Buttom-Up Merge Sort 64M Increasing and Decreasing Comparison




### 8.6)2-Way Merge Sort and Bottom-Up Merge Sort Comparison



### 8.7)Does it provide any performance gain over ordinary mergesort?

According to results, short answer is yes. Botom-Up merge Sort Algorithm provides performance gain over ordinary Merge Sort Algorithm. Altough both algorithm has same time complexity, Bottom-Up Merge Sort works less time algorithm. Because, Bottom-Up is an iterative algorithm and Ordinary Merge Sort is a recursive algorithm. Recursive algorithms need to store function calls in the satck, but iterative algorithms don not need to it. Also, iterative alorithms are generally faster than recursive algorithms. Because of this, we can say that Bottom-Up Merge Sort is more efficient. This result as expected.

# Question 2: Balanced Search Trees

**Red-Black Trees:** Provide a definition of Red-Black Trees, show by examples the construction of Red-Black and the basic dictionary operations on a given Red-Black Trees. Then, provide an efficiency analysis (worst/case) for those dictionary operations.

## Red-Black Tree

A red-black tree is a kind of self-balancing Binary Search Tree where each node has an extra bit, and that bit is often interpreted as the color (red or **black**). These colours are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree.

## Why Red-Black Trees?

Most of the BST operations (search, insert, delete) 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 tree operation, 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.

| Insert | $\mathbf{O(\log(n))}$ |
|---|---|
| Delete | $O(\log(n))$ |
| Search | $O(\log(n))$ |

## Properties of Red Black Tree

- Red - Black Tree must be a Binary Search Tree.
- The root node must be colored **BLACK**.
- The children of Red colored node must be colored **BLACK**. (There should not be two consecutive RED nodes).
- In all the paths of the tree, there should be same number of **BLACK** colored nodes.
- Every new node must be inserted with RED color.
- Every leaf (NULL node) must be colored **BLACK**.

## Insertion

First, you have to insert the node similarly to that in a binary tree and assign a RED colour to it. Now, if the node is a root node then change its color to **BLACK**, but if it does not then check the color of the parent node. If its color is **BLACK** then don’t change the color but if it is not i.e. it is RED then check the color of the node’s uncle. If the node’s uncle has a RED colour then change the color of the node’s parent and uncle to **BLACK** and that of grandfather to RED colour and repeat the same process for him (i.e. grandfather)

## Algorithm

1)Perform BST and make the color of newly inserted nodes as RED.

2)If new node is the root, change the color of it as **BLACK**

3)Do the following if the color of new node's parent is not **BLACK** and new node is not the root.
**a)** If new node's uncle is RED
**i)** Change the color of parent and uncle as **BLACK**.
**ii)** Color of a grandparent as RED.
**iii)** Change new node = new node's grandparent, repeat steps 2 and 3 for new nodes.

**b**) If new node's uncle is **BLACK**, then there can be four configurations for new node, new node's parent (p) and new node's grandparent (g)
**i)** Left Left Case (p is left child of g and new node is left child of p)
**ii)** Left Right Case (p is left child of g and new node is the right child of p)
**iii)** Right Right Case (Mirror of case i)
**iv)** Right Left Case (Mirror of case ii)

## Example

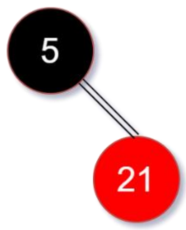
Create a Red Black Tree by inserting following sequence of number: 5, 21, 4, 12, 18, 34, 52, 90

**insert(5)**

Tree is empty. So insert new node as root node with **BLACK** colored.

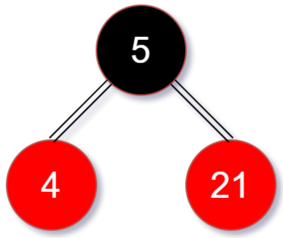

**insert(21)**

Tree is not empty. So insert new node with RED colored.




**insert(4)**

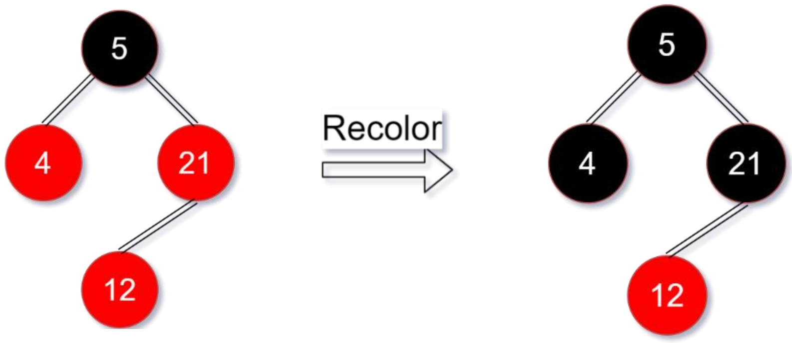
Tree is not empty. So insert new node with RED colored.



**insert(12)**

Tree is not empty. So insert new node with RED colored.

There are 2 consecutive RED nodes (21 and 12). The new nodes's parent's sibling's (new node's uncle) color is RED and parent's parent is root. So tree is recolored.

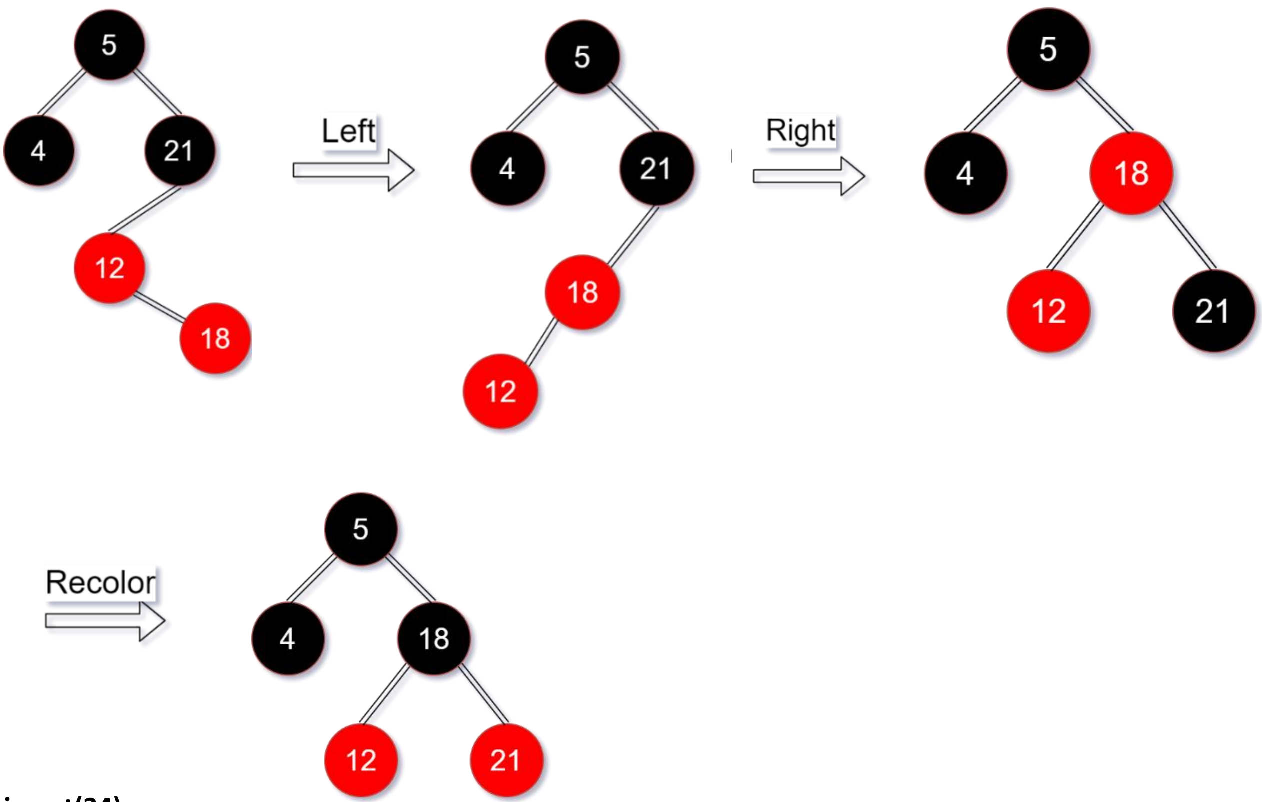



After recolor operation, the tree is satisfying all Red-Black Tree properties.

**insert(18)**

Tree is not empty. So insert new node with RED colored.

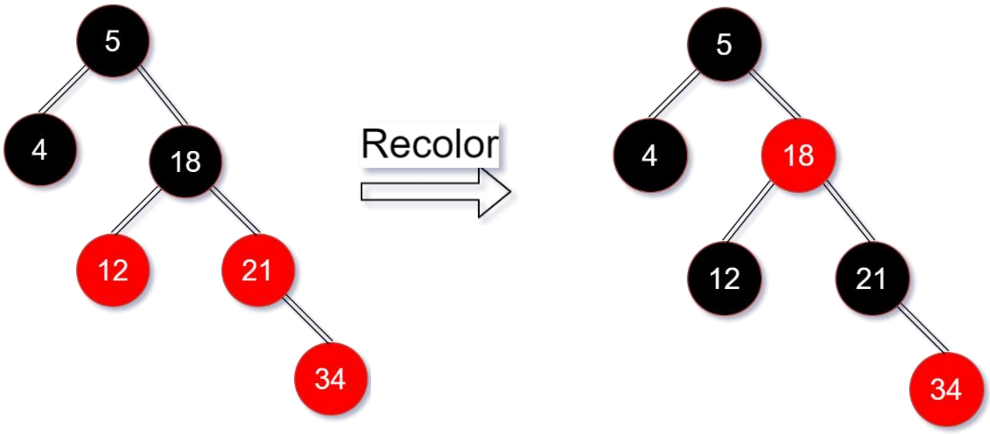
There are 2 consecutive RED nodes (12 and 18). The new nodes's uncle is NULL (color of NULL is **BLACK**). So we need rotation. (Left-Right rotation and Recolor)




**insert(34)**

Tree is not empty. So insert new node with RED colored.

There are 2 consecutive RED nodes (21 and 34). The new nodes's uncle's color is RED and parent's parent is not root. So we need recolor and check.

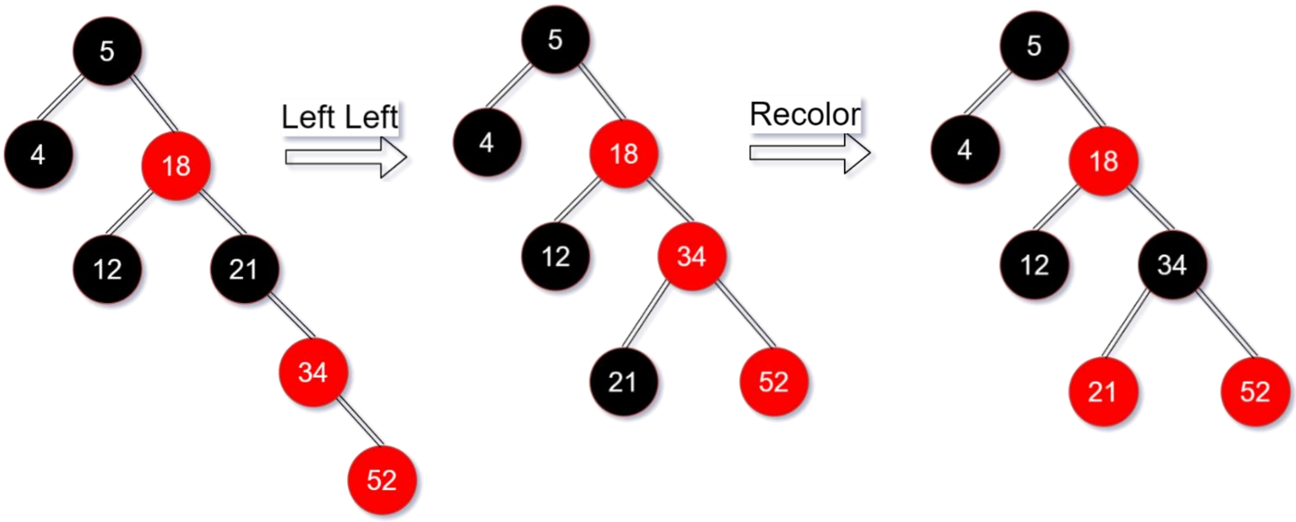



After recolor operation, the tree is satisfying all Red-Black Tree properties.

**insert(52)**

Tree is not empty. So insert new node with RED colored.

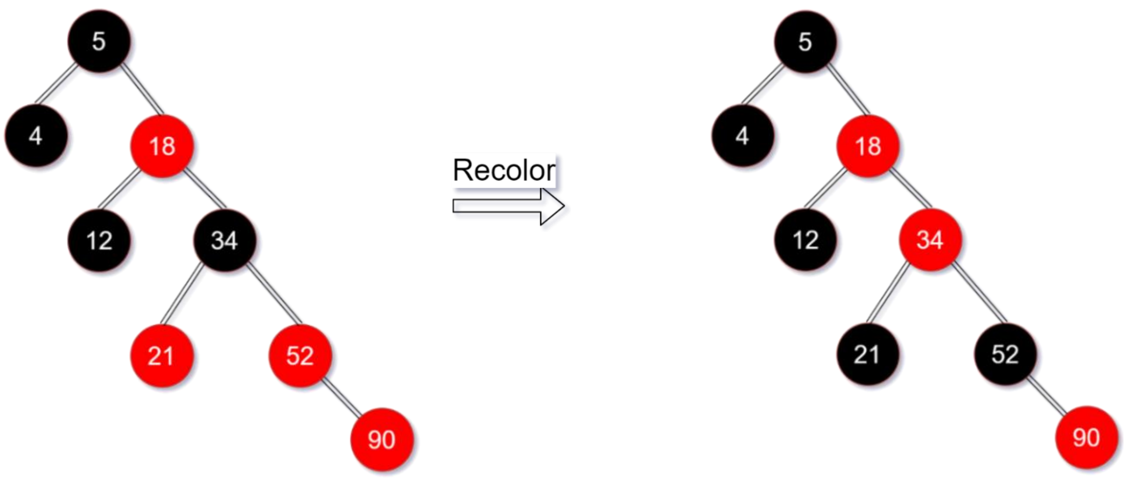
There are 2 consecutive RED nodes (34 and 52). The new nodes's uncle is NULL (**BLACK**). So we need rotation. (Left Left rotation and Recolor)



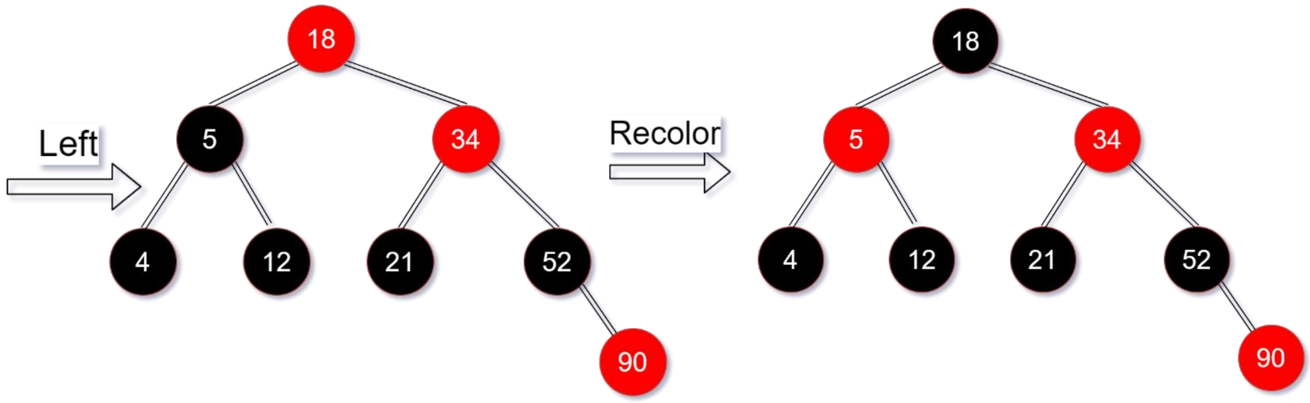

**insert(90)**

Tree is not empty. So insert new node with RED colored.

There are 2 consecutive RED nodes (52 and 90). The new nodes's uncle's color is RED and parent's parent is not root. So we need recolor and recheck



After recolor operation, again there are 2 consecutive RED nodes (18 and 34). Node 34's uncle's color is **BLACK**. So we need rotation. (Left rotation and recolor)



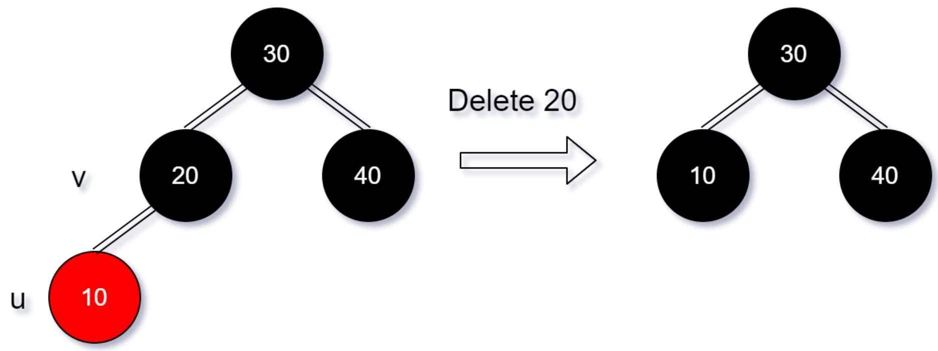

## Deletion

To understand deletion, notion of **double black** is used.  When a black node is deleted and replaced by a black child, the child is marked as ***double black***. The main task now becomes to convert this double black to single black.

### Deletion Steps

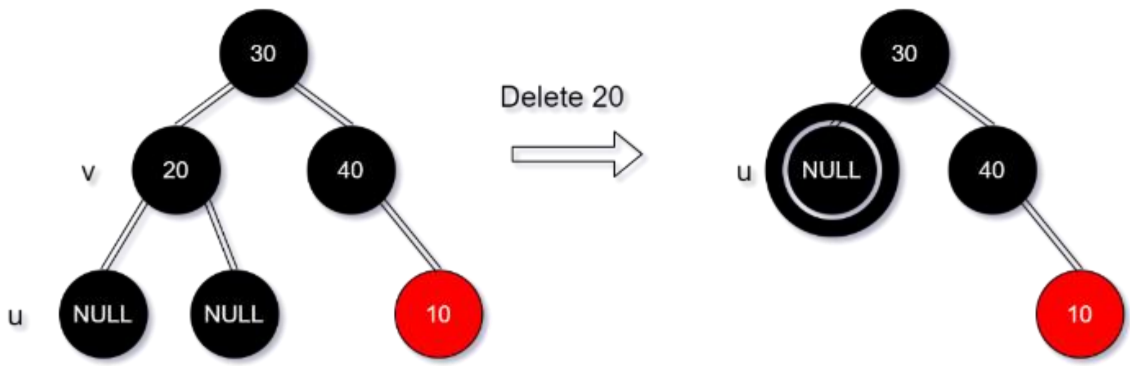
Let v be the node to be deleted and u be the child that replaces v (Note that u is NULL when v is a leaf and color of NULL is considered as **BLACK**).

**Step 1) Simplest Case:** If either u or v is RED, we mark the replaced child as **BLACK** (No change in black height). Note that both u and v cannot be RED as v is parent of u and two consecutive REDS are not allowed in Red Black Tree.




**Step 2)** If both u and v are **BLACK**

**2.1)** Color u as double **BLACK**.  Now our task reduces to convert this double black to single black. Note that If v is leaf, then u is NULL and color of NULL is considered as **BLACK**. So the deletion of a **BLACK** leaf also causes a double **BLACK**.



When 20 is deleted, it is replaced by a NULL, so the NULL becomes double **BLACK**. Note that deletion is not done yet, this double **BLACK** must become single **BLACK**.

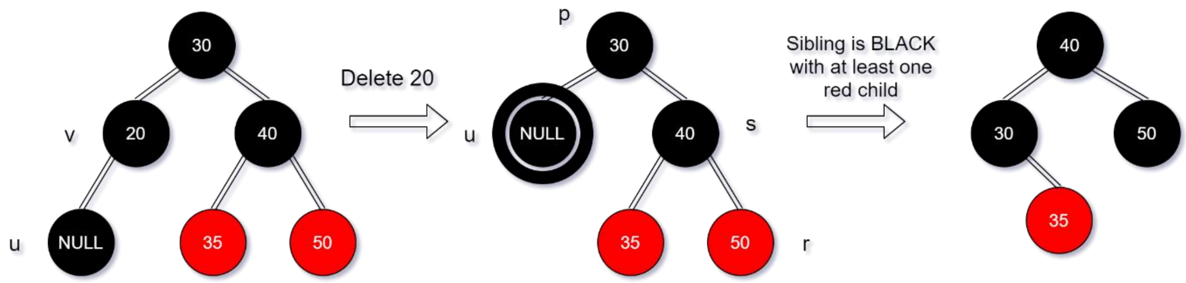
**2.2)** Do following while the current node u is double **BLACK** and it is not root. Let sibling of node be s.

**2.2.1)** If sibling s is **BLACK** and at least one of sibling's children is RED, perform rotation(s). Let the RED child of s be r. This case can be divided in 4 subcases depending upon positions of s and r.

**i) Left Left Case** (s is left child of its parent and r is left child of s or both children of s are **RED**). This is mirror of right right case shown in below diagram (iii).

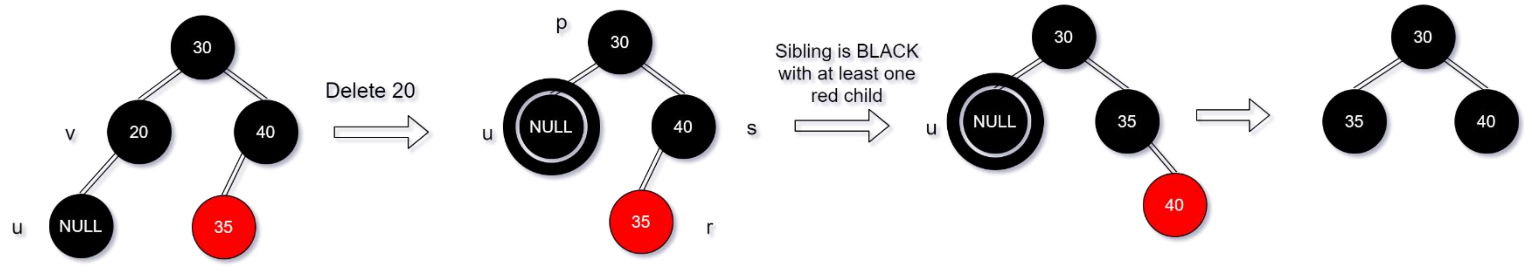
**ii) Left Right Case** (s is left child of its parent and r is right child). This is mirror of right left case shown in below diagram (iv).

**iii) Right Right Case** (s is right child of its parent and r is right child of s or both children of s are RED)

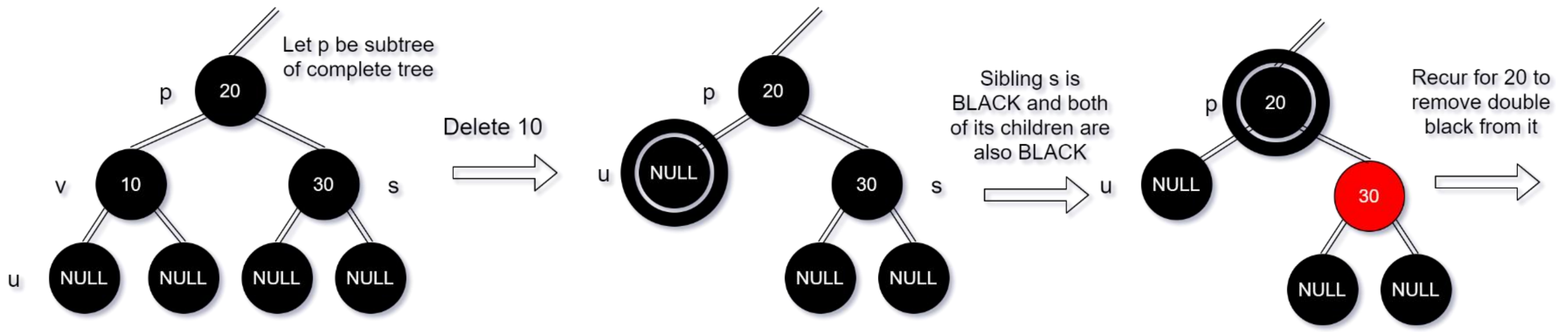



Sibling s is right child of its parent and right child of s is RED (RR Case)

**iv) Right Left Case** (s is right child of its parent and r is left child of s)



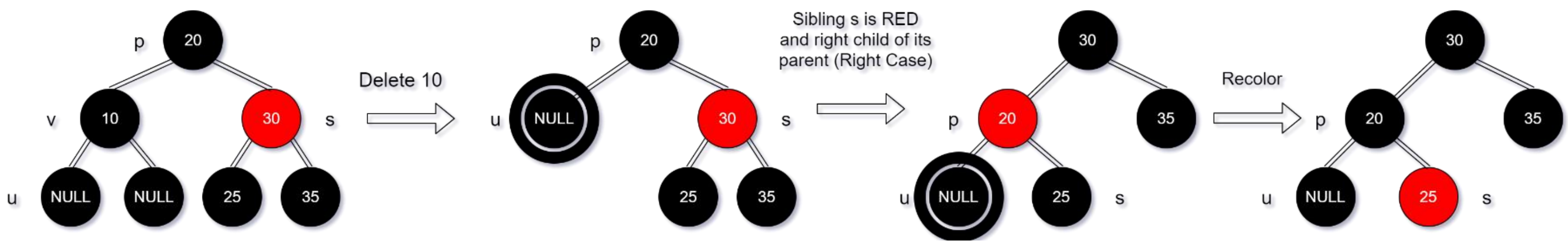

**2.2.2)** If sibling is **BLACK** and its both children are **BLACK**, perform recoloring, and recur for the parent if parent is black.



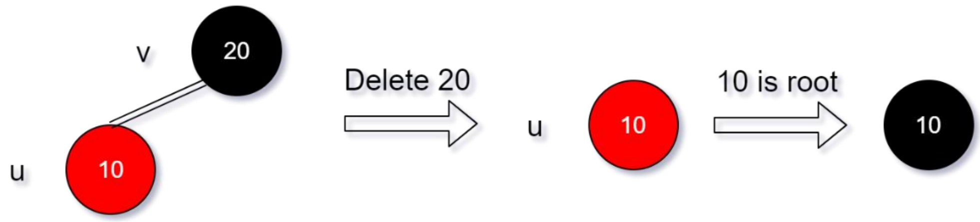

In this case, if parent was RED, then we didn't need to recur for parent, we can simply make it **BLACK** (red + double black = single black)

**3.2.3)** If sibling is RED, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always **BLACK**. This mainly converts the tree to **BLACK** sibling case (by rotation) and leads to case (2.2.1) or (2.2.2). This case can be divided in two subcases.

**i) Left Case** (s is left child of its parent). This is mirror of right right case shown in below diagram (ii). We right rotate the parent p.
**ii) Right Case** (s is right child of its parent). We left rotate the parent




**2.3)** If u is root, make it single **BLACK** and return (Black height of complete tree reduces by 1)




## Searching

Red-Black Tree is actually is an Binary Search Tree. So searching operation is same with BST.

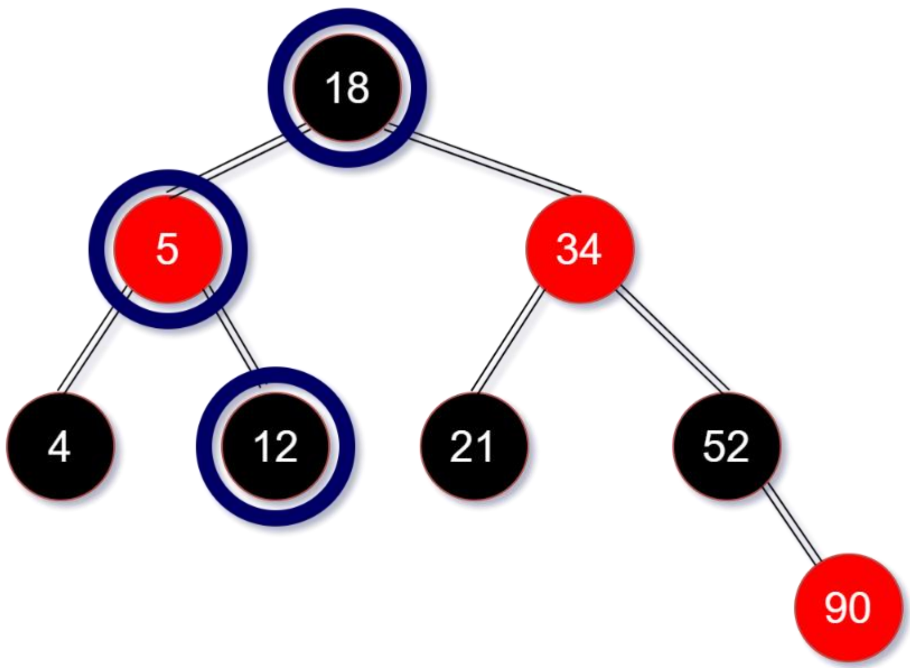
**Searching Steps:**

**1)** Start from the root.

**2)** Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.

**3)** If the element to search is found anywhere, return true, else return false.

**Example:** Search element 12



**References**


https://www.geeksforgeeks.org/merge-sort/

https://www.geeksforgeeks.org/3-way-merge-sort/

https://www.geeksforgeeks.org/iterative-merge-sort/

https://stackoverflow.com/questions/14713468/why-should-we-use-n-way-merge-what-are-its-advantages-over-2-way-merge

https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/

https://www.geeksforgeeks.org/red-black-tree-set-3-delete-2/

http://www.btechsmartclass.com/data_structures/red-black-trees.html

https://www.programiz.com/dsa/red-black-tree

https://www.programiz.com/dsa/insertion-in-a-red-black-tree

https://www.youtube.com/watch?v=2Ww4FMMJwp8

https://www.geeksforgeeks.org/red-black-tree-set-2-insert/