# Algorithm Analysis & Design

Merge Sort

# 2-Way Merge Sort

```
Function sort(arr[0...n-1], c[0...n-1], low, high)
    if 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
```

**Time Complexity:** $n\log_2(n) - n + 1 \in \theta(n\log(n))$

```
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
```

# 3-Way Merge Sort

```
Function sort(arr[0...n-1], c[0...n-1], low, high)
    if 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
```

**Time Complexity:** $n\log_3(n) - n + 1 \in \theta(n\log(n))$

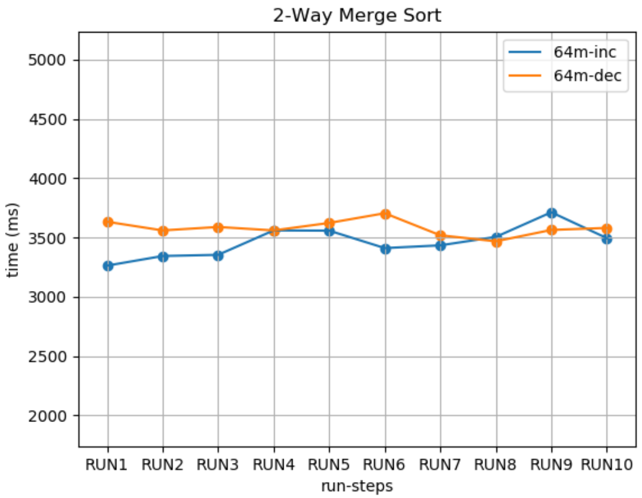
# Bottom-Up Merge Sort

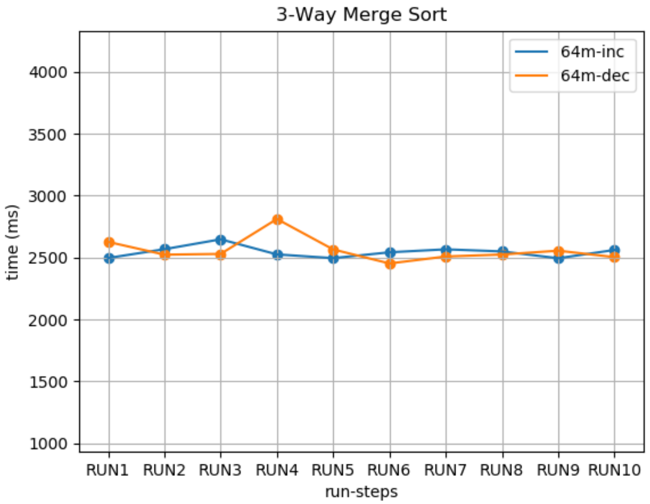
```
Function sort(arr[0...n-1])
    n ← size of arr
    create c[0...n-1]
    len ← 1
    while len < n do          // merge subarrays size 1, size = 2, size = 4, ...
        low ← 0
        while low < n – len do   // pick starting point of different subarrays of current size
            mid ← low + 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
```

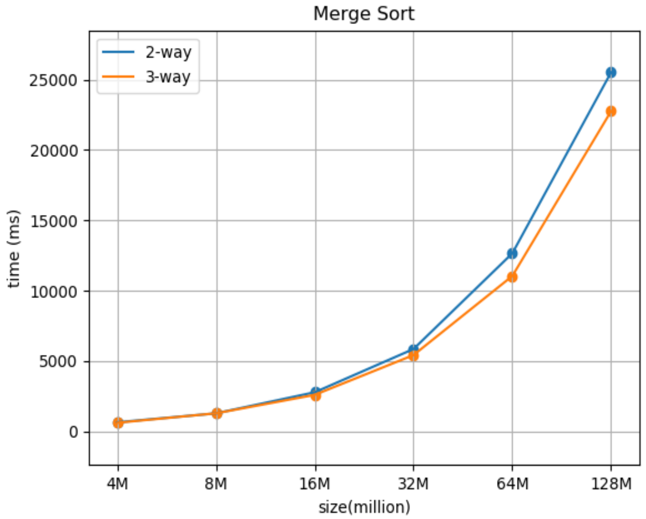
**NOTE:** Merge Algorithm is same with 2-Way Merge Sort.

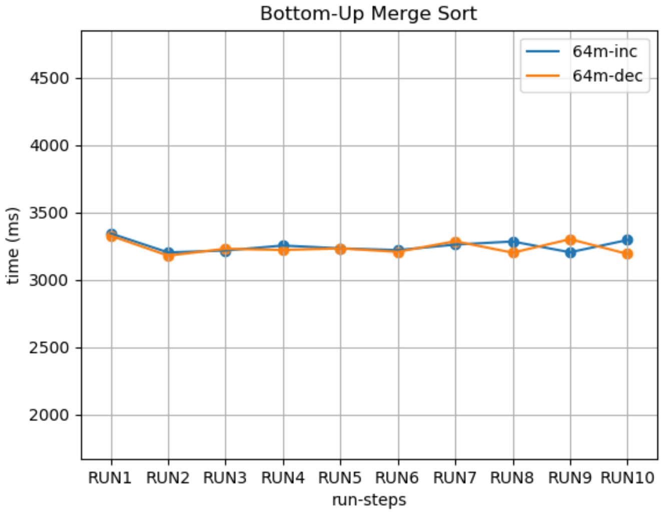
**Time Complexity:** $n\log_2(n) - n + 1 \in \theta(n\log(n))$

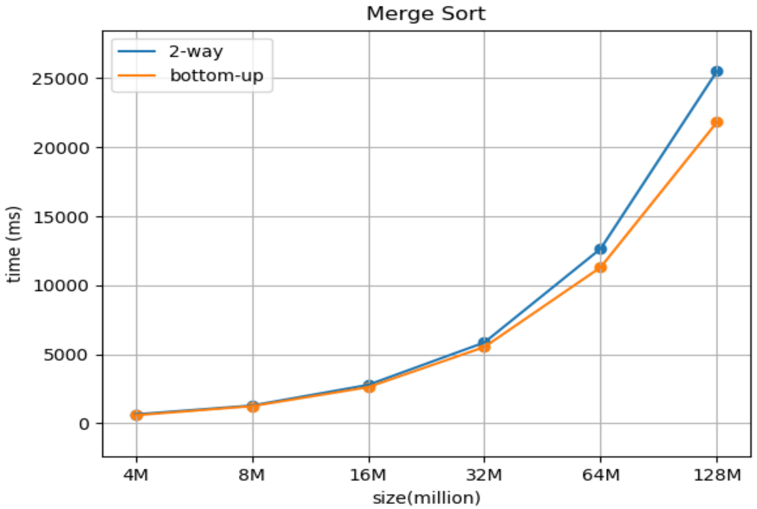
# Test Results



# Red-Black Trees

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**)

## Why Red-Black Trees

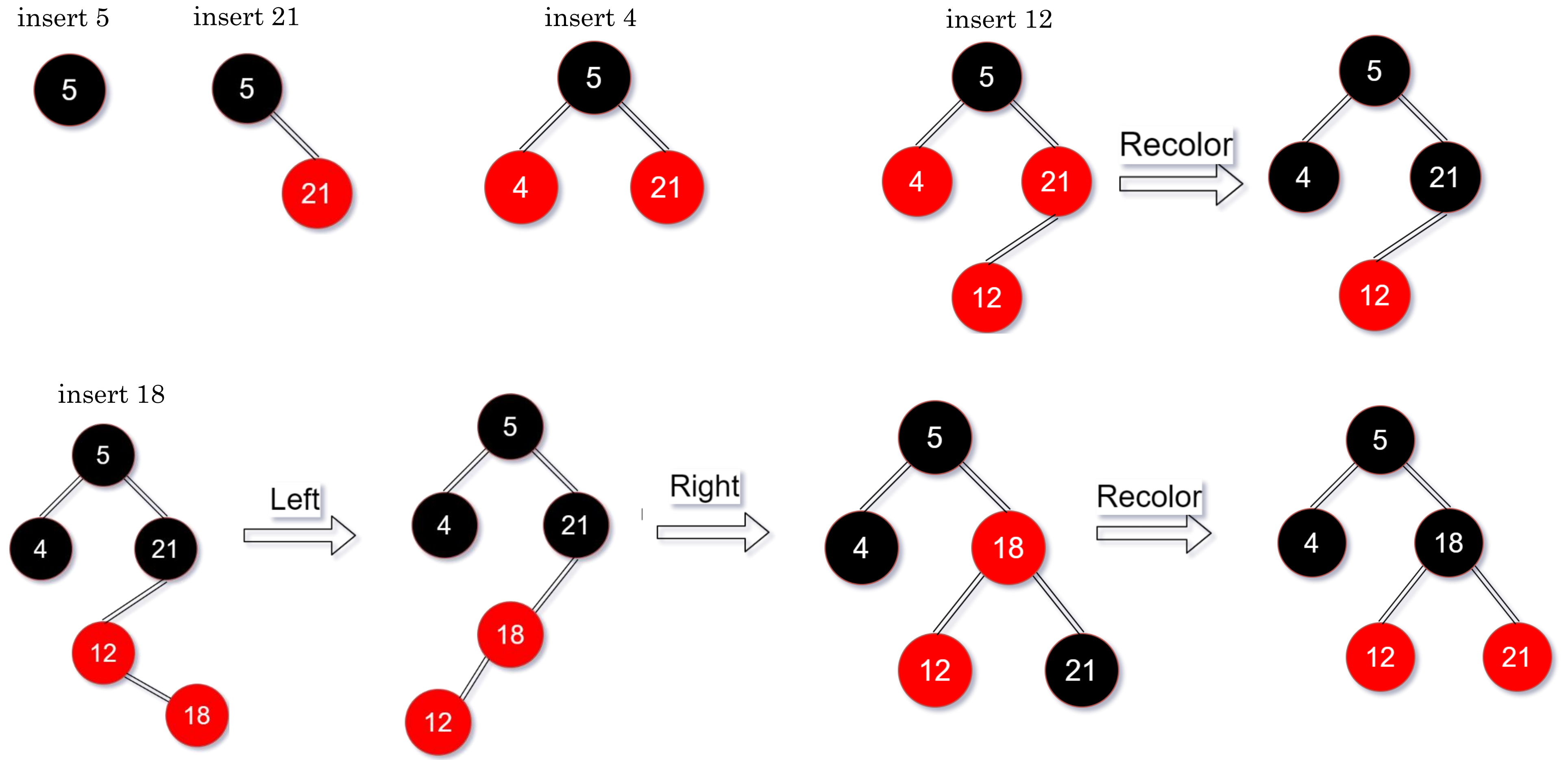
Most of the BST operations (search, insert, delete) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary Tree. If we make sure that the height of the tree remains **O(log n)** after every tree operation, then we can guarantee an upper bound of O(log n) for all these operations

## 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**.

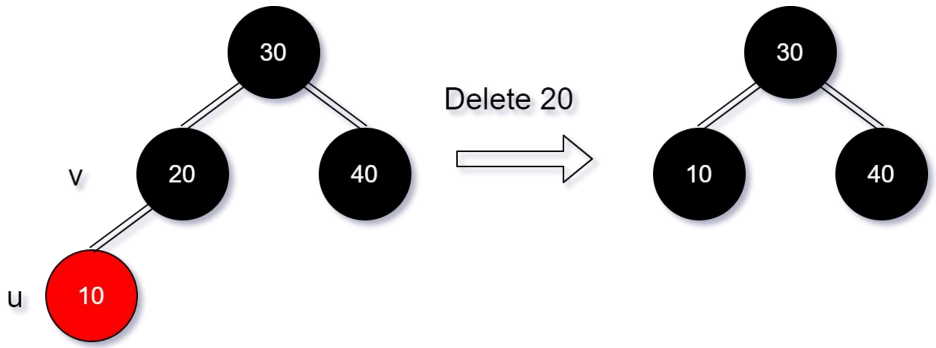
# Insertion

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

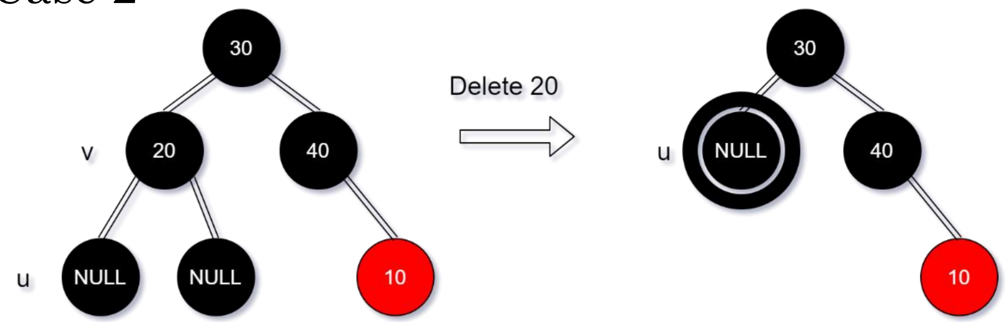


# Deletion

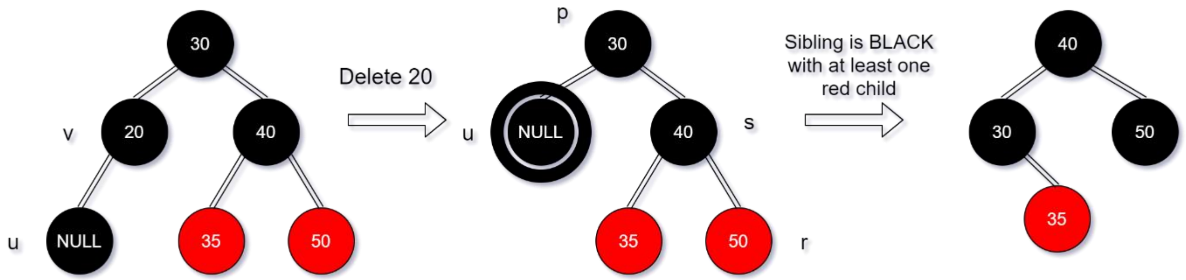
## Case 1



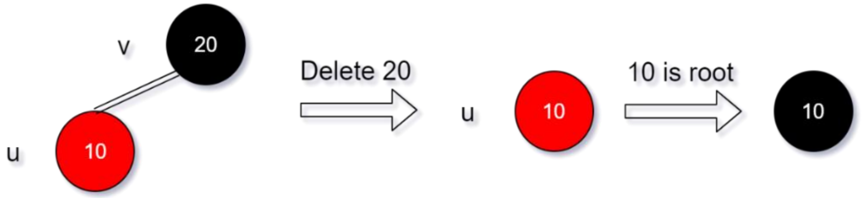

## Case 2




## Case 3



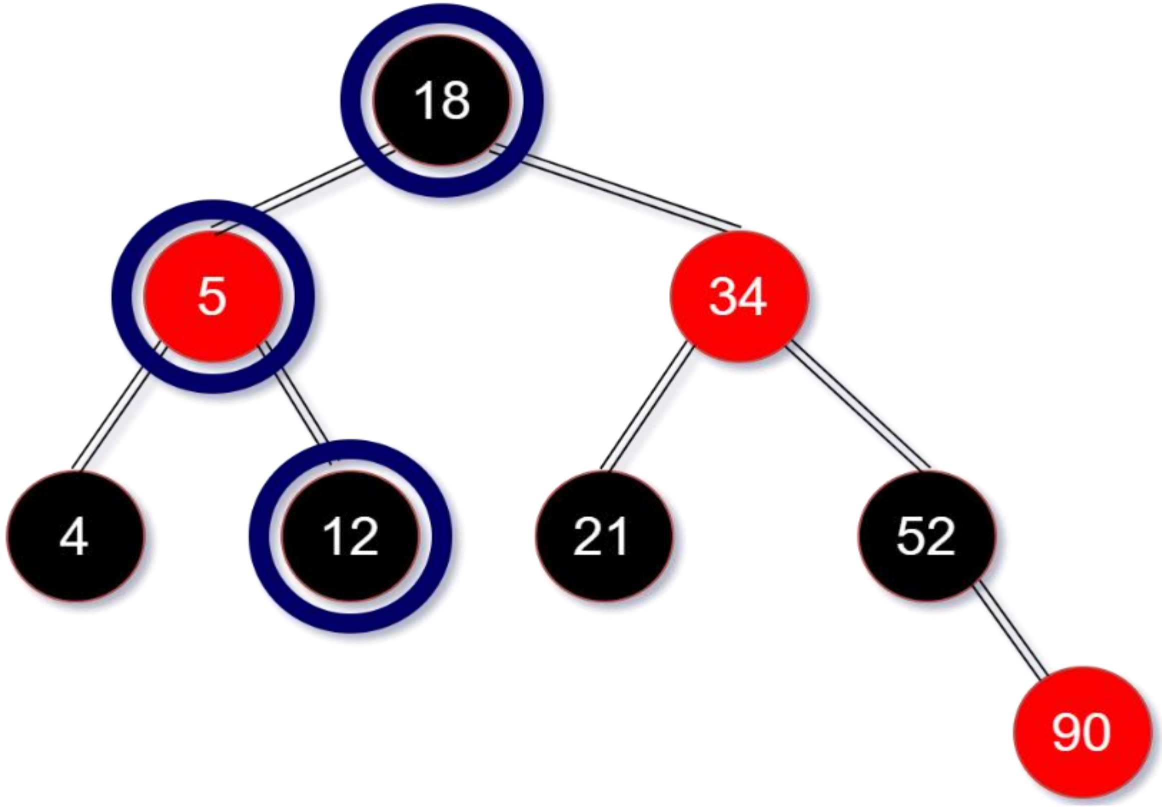

## Case 4



# Searching

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

Search 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/