Red Black Tree

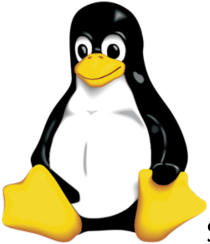
Properties and Visualization

Shashata Sawmya[1]
Abdullah Al Ishtiaq[1]

[1]Department of Computer Science and Engineering
BUET

July 15, 2018

A Practical Problem



Suppose we have a some programmes to be run by our linux OS scheduler. The linux schdeduler will need a data structure to sort them by their lowest spent execution time. So, it might need a Binary Search tree to sort the programmes. But, a BST might grow linearly in worst case as new programmes are inserted to be managed by the scheduler which will increase the run-time from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n^2)$. So, it will require a self height balancing BST which will always give the reduced searching time.

What is Red Black Tree?

- A red–black tree is a kind of self-balancing binary search tree in computer science.
- Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node.
- By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other.
- The tree is thus approximately height balanced.

What is Red Black Tree?

- A red–black tree is a kind of self-balancing binary search tree in computer science.
- Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node.
- By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other.
- The tree is thus approximately height balanced.

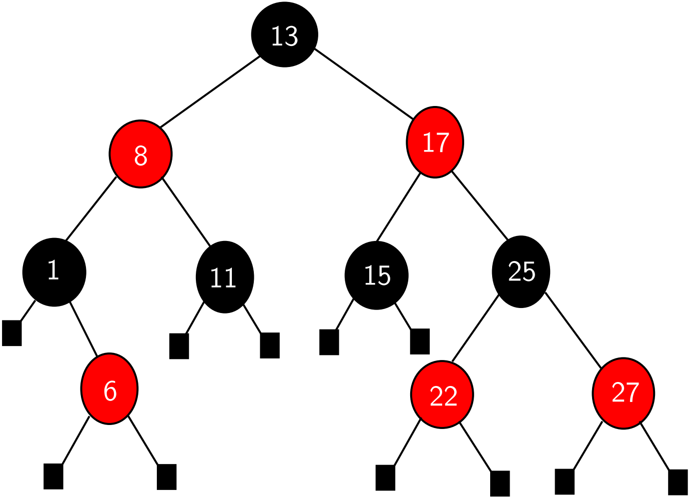
What is Red Black Tree?

- A red–black tree is a kind of self-balancing binary search tree in computer science.
- Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node.
- By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other.
- The tree is thus approximately height balanced.

What is Red Black Tree?

- A red–black tree is a kind of self-balancing binary search tree in computer science.
- Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node.
- By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other.
- The tree is thus approximately height balanced.

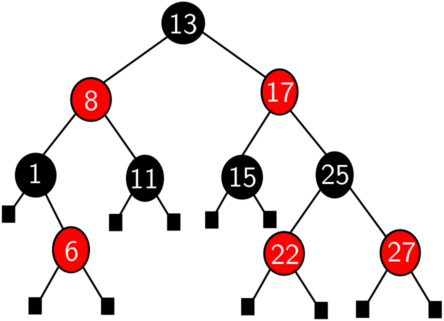
Properties of a Red Black Tree



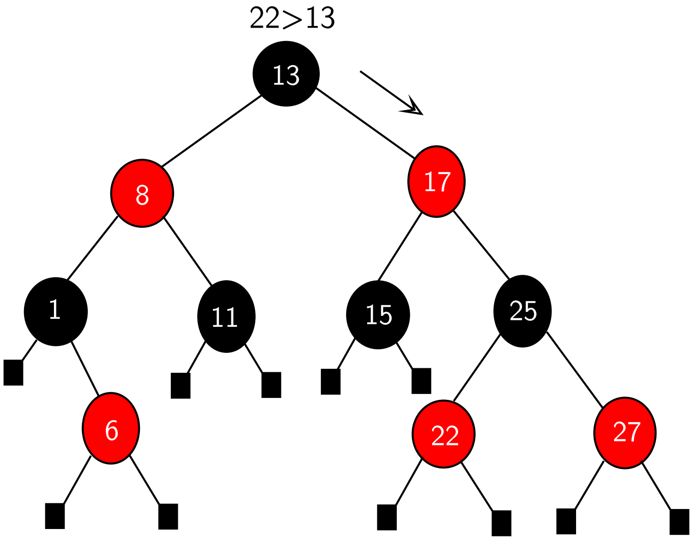
Properties of Red Black Tree

A Red Black Tree is a Binary Search Tree which have the following Red-Black properties:

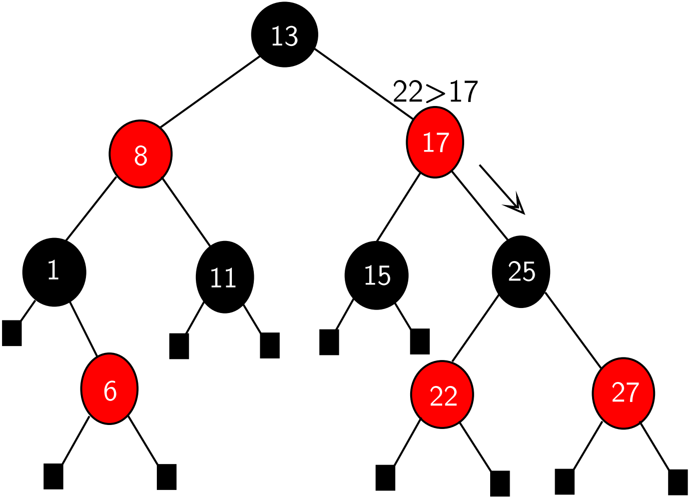
1. Color Property: Each node is either red or black.
2. Root Property: The root is black
3. External Property: Every external node is black
4. Internal Property: Both children of a red node are black.
5. Depth Property: For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.



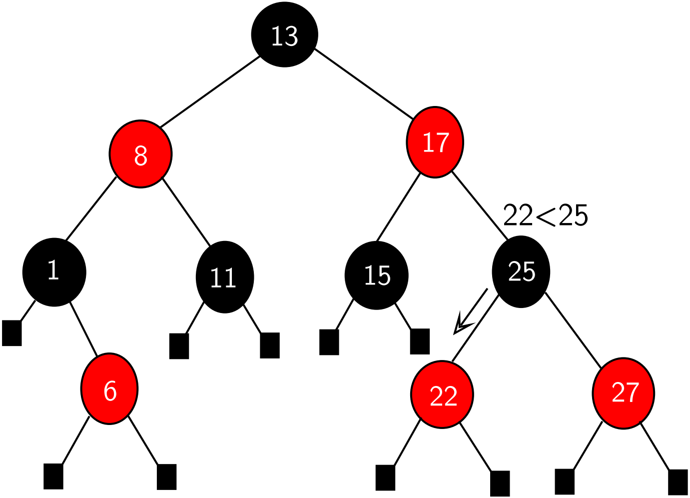
Searching 22



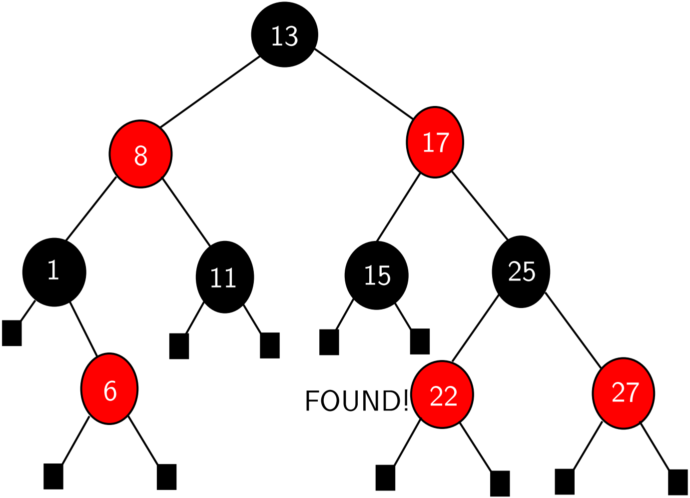
Searching 22



Searching 22



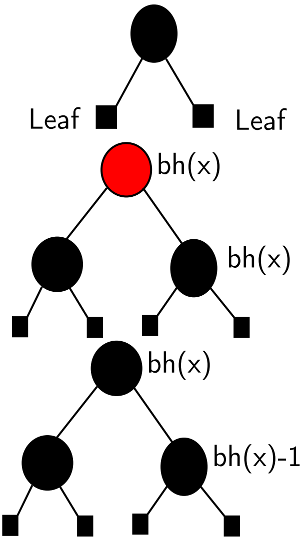
Searching 22



Why Red Black Tree is Height Balanced

Red Black Tree for storing n items will have a height of $\mathcal{O}(n)$. So, it will remain height balanced.

1. the subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.
2. If $h(x) = 0$ then x is a leaf, so the subtree rooted at x contains $2^0 - 1 = 0$ internal nodes.
3. For a node x with positive height and two children, the black height of the children will be $bh(x)$ (for red internal node) or $bh(x) - 1$ (for black internal node).



Why Red Black Tree is Height Balanced

- So, using induction, we can prove subtree rooted at x contains at least $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = (2^{bh(x)} - 1)$ internal nodes.
- According to the internal property the black-height of the root must be at least $\frac{h}{2}$; thus
- $n \geqslant 2^{\frac{h}{2}} - 1$
- $\log(n+1) \geqslant \frac{h}{2}$
- $h \geqslant 2\log(n+1)$
- Thus $h = \mathcal{O}(\log(n))$

Red Black Tree Insertion

- We can insert each node in Red Black Tree in $\mathcal{O}\log(n)$ time
- We use left rotation, right rotation and recoloring of nodes to fix the property violation which is caused by inserting a new node.
- There are roughly three cases in Red Black Tree Insertion.
- We are simulating the insertion of 5 keys: $2, 3, 5, 7, 4$ consecutively in an empty RBT.

Red Black Tree Insertion

- We can insert each node in Red Black Tree in $\mathcal{O}\log(n)$ time
- We use left rotation, right rotation and recoloring of nodes to fix the property violation which is caused by inserting a new node.
- There are roughly three cases in Red Black Tree Insertion.
- We are simulating the insertion of 5 keys: $2, 3, 5, 7, 4$ consecutively in an empty RBT.

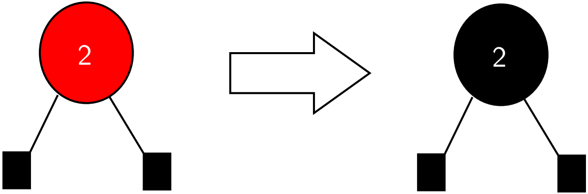
Red Black Tree Insertion

- We can insert each node in Red Black Tree in $\mathcal{O}\log(n)$ time
- We use left rotation, right rotation and recoloring of nodes to fix the property violation which is caused by inserting a new node.
- There are roughly three cases in Red Black Tree Insertion.
- We are simulating the insertion of 5 keys: $2, 3, 5, 7, 4$ consecutively in an empty RBT.

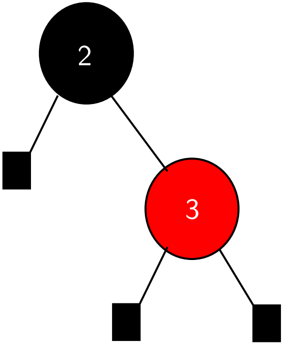
Red Black Tree Insertion

- We can insert each node in Red Black Tree in $\mathcal{O}\log(n)$ time
- We use left rotation, right rotation and recoloring of nodes to fix the property violation which is caused by inserting a new node.
- There are roughly three cases in Red Black Tree Insertion.
- We are simulating the insertion of 5 keys: $2, 3, 5, 7, 4$ consecutively in an empty RBT.

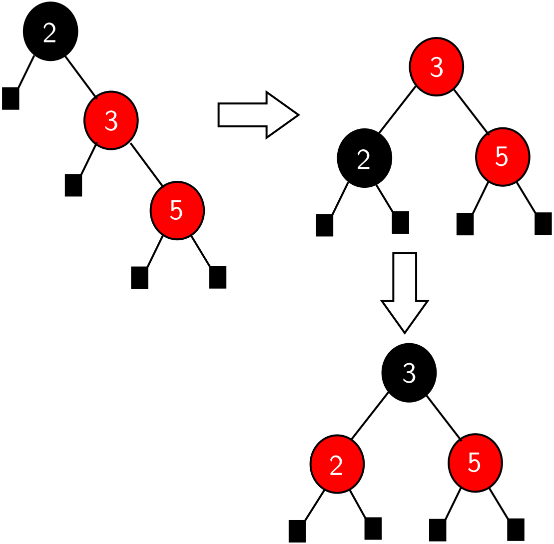
Insert 2



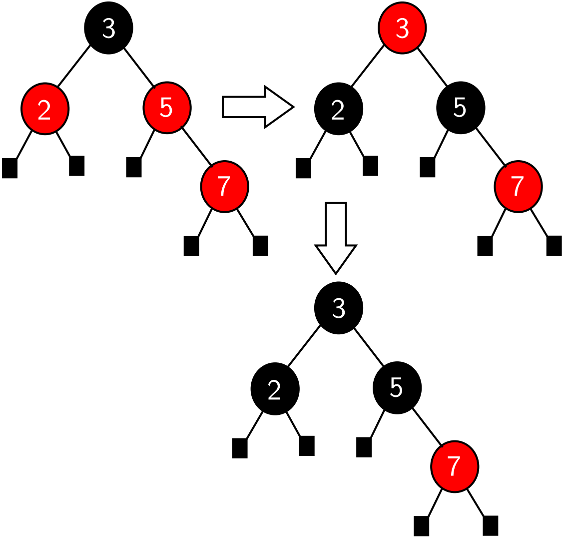
Insert 3



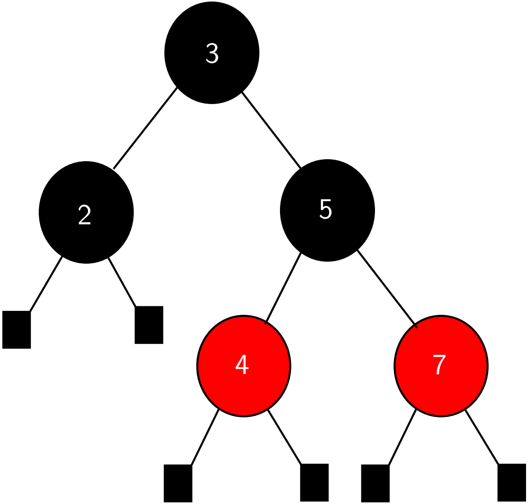
Insert 5



Insert 7



Insert 4



Red Black Tree Insert

```
RB Insert(T,z)
y ← T.nil
x ← T.root
while x ≠ T.nil do
    y ← x;
    if z.key < x.key then
        x ← x.left
    else
        x ← x.right
    end
end
z.p ← y
if y = T.nil then
    T.root ← z
else if z.key < y.key then
    y.left ← z
else
    y.right ← z
z.left ← T.nil
z.right ← T.nil
y.color ← red
RB-Insert-Fixup(T, z)
```

Red Black Tree Deletion

- Similar to Red Black Tree Insertion, Deletion also runs in $\mathcal{O}\log(n)$ time
- Deletion is a bit more complicated than Insertion
- There are mainly three cases in RBT Deletion; in one of the cases, a double black node is created which can be handled through six different cases.

Red Black Tree Deletion

- Similar to Red Black Tree Insertion, Deletion also runs in $\mathcal{O}\log(n)$ time
- Deletion is a bit more complicated than Insertion
- There are mainly three cases in RBT Deletion; in one of the cases, a double black node is created which can be handled through six different cases.

Red Black Tree Deletion

- Similar to Red Black Tree Insertion, Deletion also runs in $\mathcal{O}\log(n)$ time
- Deletion is a bit more complicated than Insertion
- There are mainly three cases in RBT Deletion; in one of the cases, a double black node is created which can be handled through six different cases.

THANK YOU
ANY QUESTIONS ?