

Data structures

**Research Papers 1**

**Red-Black trees**

Name : Abdulghaffar Abed

ID : 1180071

Instructor : Dr. Radi jarar

S# : 4

Red-black tree is a binary search tree with self balancing , every node in this type take either red or black color these colors help to ensure that the tree still approximately balanced during insertion and deletion .

* The root of the red-black tree is always black .
* The leaf node with NULL (without children) is black
* The red node has more restrictions they cannot have a red parent or red child ( nonadjacent ), if the node is red then its children are black .
* Every sequence from a node that consist root node to any of its descendants NULL node has the same number of black nodes.
* Nodes require one storage bit to keep track of color.

**Red-black tree algorithm :-**

The red-black used color for balancing the tree ,The limitations put on the node colors ensure that the longest path (from root to the last NULL) is no more than twice the length of the shortest path (from root to the first NULL),It helps in maintaining the self-balancing property of the red-black tree as a result of self-balancing the height of the red-black tree is always O(log n) where n is the number of nodes in the tree

**When it is best to use Red-Black trees :-**

to lookup , insertion and deletion Red-Black tree is good choice because of balancing and less cost with than the AVL trees that are more rigidly balanced than red black trees, leading to slower insertion and removal but faster retrieval , Red Black trees are good if there are similar number of insertions, deletions and lookups .

Red-black trees can be used in process schedulers, maps, sets .

**Rotations on Red-Black trees :-**

Red-black tree used rotations to maintain the properties of a red-black tree when insert or delete node to the tree and to decrease the height of the tree.

There are two types of rotations :-

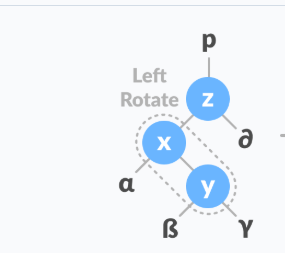
1. **Left rotate**
2. **Right rotate**

**Left rotate algorithm :**

Let the Y be the right subtree that we would to rotate it and X is the parent ( root ).

1. Rotate y to the left and X as a left child of Y (Y is the parent now).
2. If Y has right children keep them with Y .
3. If Y has left children put the left children as right children to X.

* **Left-right rotation :-**



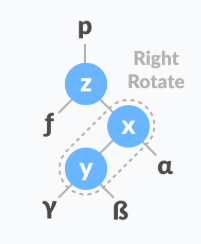
1. Do left rotation on y-x (now z x y all in the left in order from parent to its child respectively )
2. Do right rotation on y-z
3. The result that y is parent of left child x with its children and of right child z with its children .

**Right rotate algorithm :**

Let the Y be the left subtree that we would to rotate it and X is the parent ( root ).

1. Rotate y to the right and X to the right of Y (Y is the parent now).
2. If Y has left children keep them with Y .
3. If Y has right children put the right children as left children to X.

* **Right-left rotation :-**



1. Do right rotation on y-x (now z x y all in the right in order from parent to its child respectively )
2. Do left rotation on y-z
3. The result that y is parent of left child z with its children and of right child x with its children .

From the previous it’s clear that the Red-black rotation and AVL rotation are approximately the same .

## Red-Black tree operations :

**Algorithm to insert a node :-**

Insert may result in violation of Red-black tree properties .

* Let y be the leaf (y children are NULL) and x be the root of the tree.
* Check if the tree is empty (x is NULL).
* If yes, insert new node as a root node and color it black.
* Else, find the last node in the tree leaf whose children are NULL
* By comparing nodes in the tree:

Compare new key with root key (root value ).

* If new key is greater than root key, traverse through the right subtree because its binary search tree .

Else traverse through the left subtree.

* Assign the parent of the leaf as a parent of new node.
* If leaf key is greater than new key, make new node as right child.

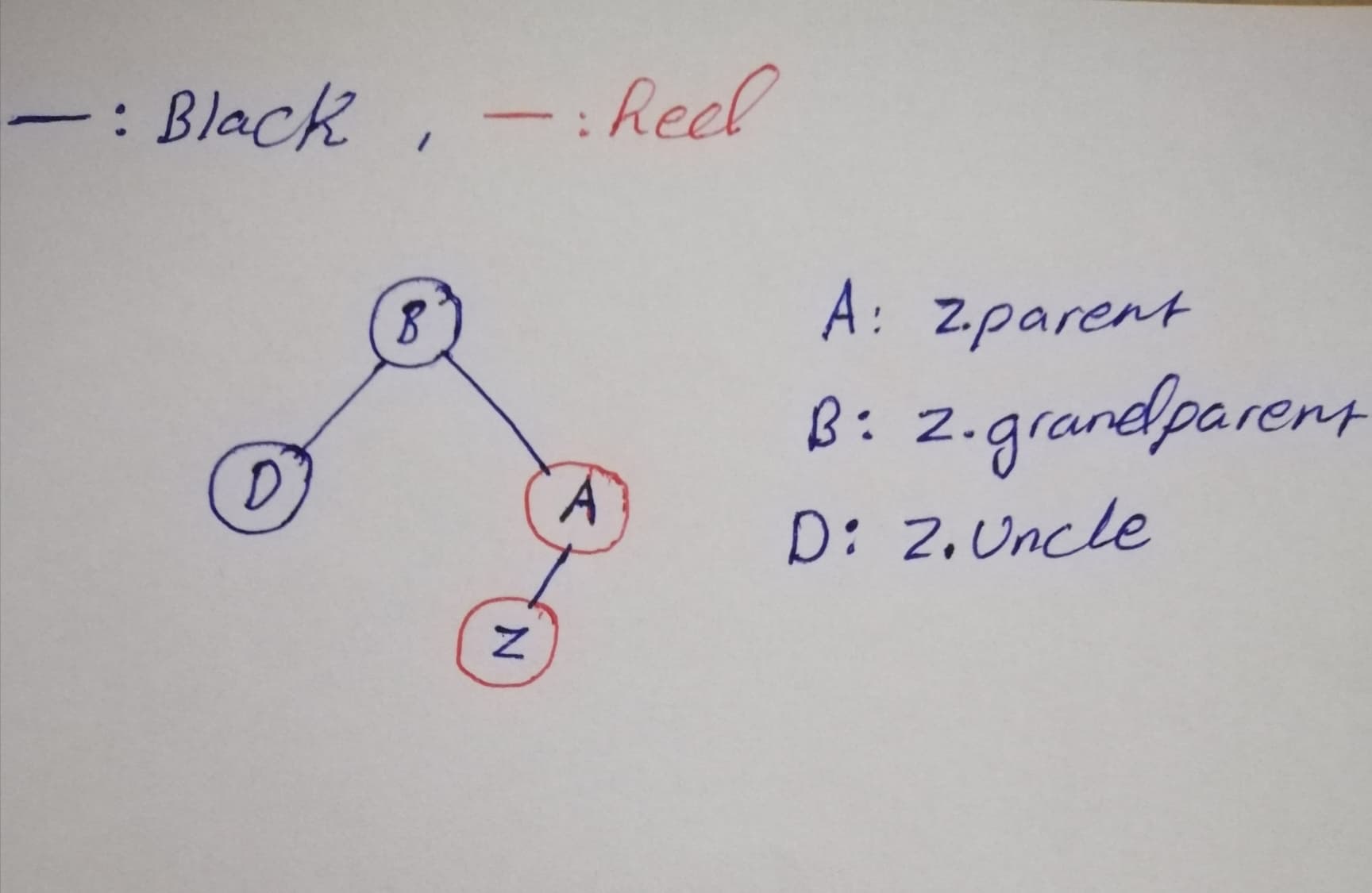
Else, make new node as left child.

* Assign NULL to the left and right child of new node.
* Assign RED color to new node ( to preserve the depth property of the tree in other word when the node is red fix the violent is easy ).
* Call Insert fix-algorithm to maintain the property of red-black tree if violated(that the tree by recolor and rotations ).

There are 4 sensor cases :

In all the following cases we review the properties for each step .

Firstly , let’s do refresher in the relationships in a binary tree .



Case 0 : Z ( insert node ) = root .

* Only we need to color Z to black and we are done .

Case 1 : Z has a red uncle .

* We recolor z parent , grandparent and uncle ( recolor means if the node red recolor it to black and vis versa ) .

Case 2 : Z uncle is black in triangle form ( triangle looks like the form of left-right or right-left rotation )

* Rotate Z parent in the opposite direction of Z .

Case 3 : Z uncle is black in line form .

* Rotate Z parent in the opposite direction of Z .
* Recolor the original parent and grandparent after the rotation .

**Algorithm to delete a node :-**

* **Initial Steps :**
* If the node we deleted has 2 Null children , its replacement x is Null .
* If the node we deleted has 1 Null child and 1 non-Null child ,its replacement x is the non-Null child.
* If the node we deleted has 2 non-Null children , set x to the replacement’s right child before the replacement is spliced out .
* If the node we deleted is red and its replacement is red or Null , we are done .
* If the node we deleted is red and its replacement is black , color the replacement red and proceed to the suitable case
* If the node we deleted is black and its replacement is red, color the replacement black and we are done .
* If the node we deleted is black and its replacement is Null or black , proceed to the **suitable case** .

* **Cases :**

1. Node x is red .
2. Node x is black and its sibling w is red .
3. Node x is black and its sibling w is black & both of w’s children are black .
4. Node x is black & its sibling w is black and :

* If x is the left child, w’s left child is red & w’s right child is black .
* If x is the right child, w’s right child is red & w’s left child is black .

1. Node x is black & its sibling w is black and :

* If x is the left child, w’s right child is red .
* If x is the right child, w’s left child is red .

**Case 1**

Node x is red :

1. Only color x black

**Case 2**

Node x is black and its sibling w is red :

1. Color w black
2. Color x parent red
3. Rotate x parent
4. If x is the left child do a left rotation .
5. If x is the right child do a right rotation .
6. Now we have to change w
7. If x is the left child set w = x parent right
8. If x is the right child set w = x parent left
9. With x and our new w , decide on case 3,4 or 4 from here.

**Case 3**

Node x is black and its sibling w is black and both of w’s children are black :

1. Color w red
2. Set x = x parent
3. If our new x is red , color x black and we are done .
4. If our new x is black , decide on case 1,2,3 or 4 from here ( we have new w now ) .

**Case 4**

Node x is black and its sibling w is black and

* If x is the left child ,w’s left child is red and w’s right child is black
* If x is the right child ,w’s right child is red and w’s left child is black

1. Color w’s child black
2. If x is the left child , color w. left black
3. If x is the right child , color w. right black
4. Color w red.
5. Rotate w
6. If x is the left child do a right rotation
7. If x is the right child do a left rotation
8. Now we have to change w
9. If x is the left child set w = x parent . right
10. If x is the right child set w = x parent . left
11. Proceed to case 5.

**Case 5**

Node x is black and its sibling w is black and

* If x is the left child ,w’s right child is red.
* If x is the right child ,w’s left child is red.

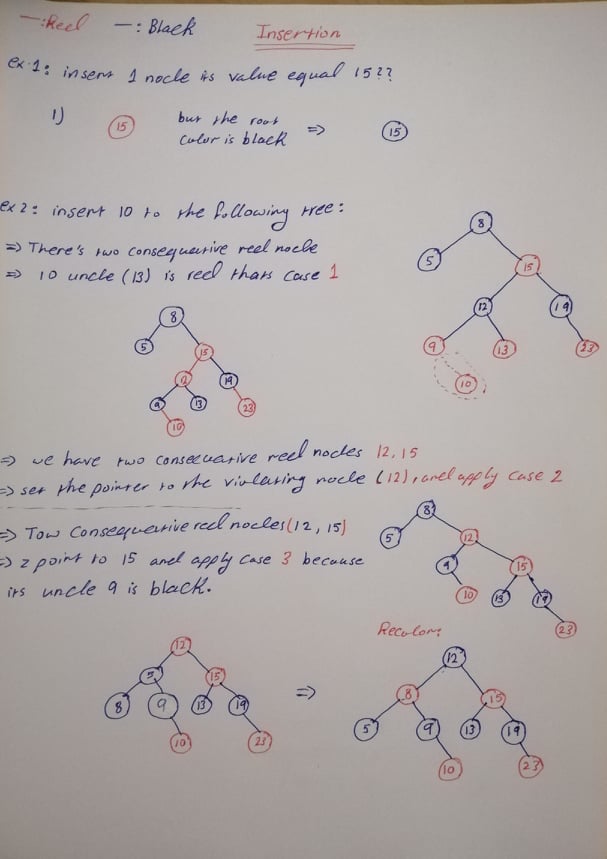
1. Color w the same color as x parent
2. Color x parent black
3. Color w’s child black
4. if x is the left child ,color w. right black
5. if x is the right child ,color w. left black
6. Rotate x parent
7. If x is the left child do a left rotation
8. If x is the right child do a right rotation
9. We are done .

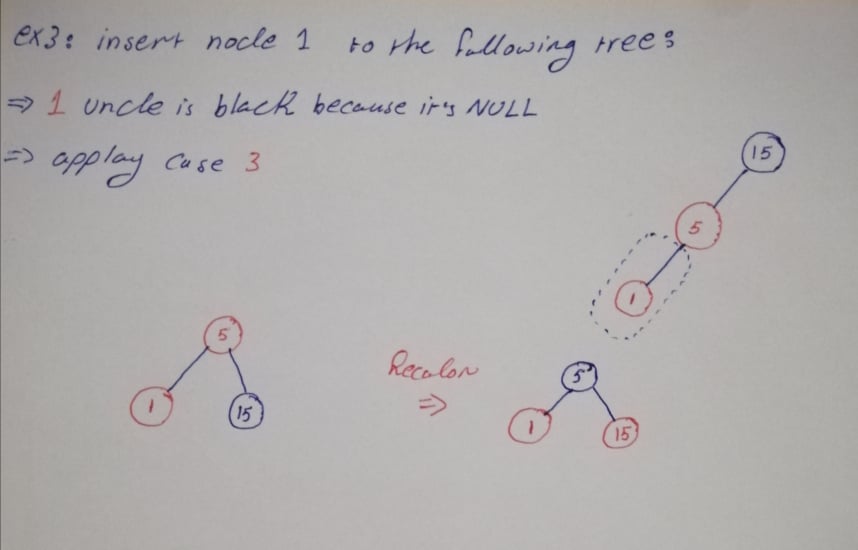
**Algorithm to search a node :**

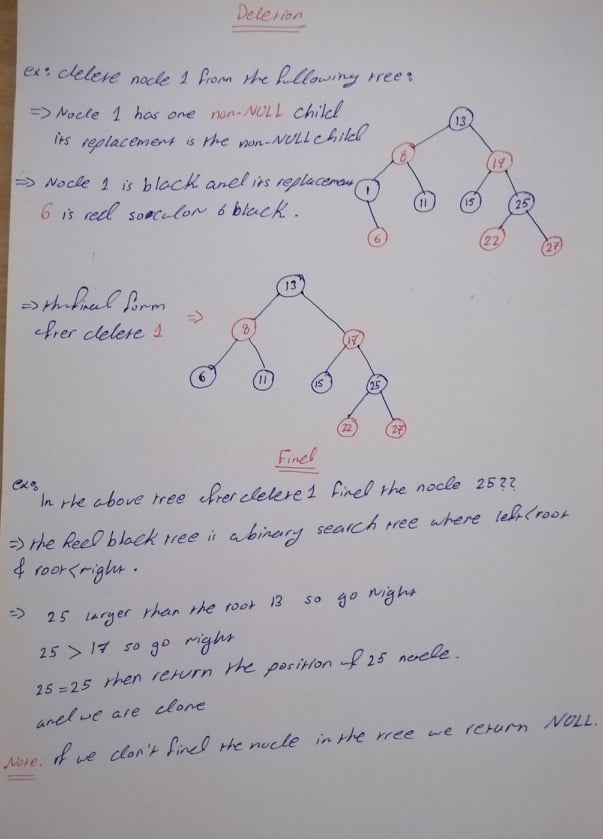
Reference - opengenus.org - .

* Perform a binary search on the records in the current node.
* If a record with the search key is found, then return that record.
* If the current node is a leaf node and the key is not found, then report an unsuccessful search.
* Otherwise, follow the proper branch and repeat the process.

# Examples :







# Red-Black trees time and space complexity

|  |  |  |
| --- | --- | --- |
| operation | Average case | Worst case |
| Insertion | O( log n ) | O( log n ) |
| Deletion | O( log n ) | O( log n ) |
| Search | O( log n ) | O( log n ) |

* **Space complexity :** Red-Black tree space complexity is O( n ) .
* Red-Black tree height is log n where n is the number of nodes .

## References

* <http://btechsmartclass.com/data_structures/red-black-trees.html>
* <https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/>
* <https://www.programiz.com/dsa/red-black-tree><https://www.youtube.com/channel/UCzDJwLWoYCUQowF_nG3m5OQ>
* <https://iq.opengenus.org/red-black-tree-search/>
* <https://en.wikipedia.org/wiki/Red%E2%80%93black_tree>