### Operations On Binary Search Trees

These operations are available for all Binary Search Trees. All these operations require comparisons to be made between the nodes.

### Searching for a Node in a Binary Search Tree

The search function is used to find whether a given value is present in the tree or not. The searching process begins at the root node. The function checks if the binary search tree is empty. If it is empty, then the value we are searching for is not present in the tree. The search algorithm terminates by displaying an appropriate message. However, if there are nodes in the tree, then the search function checks to see if the key value of the current node is equal to the value of the value to be searched. If not, it checks if the value to be searched for is less than the value of the current node, in which case it be recursively called on the left child node. In case the value is greater than the value of the current node, it should be recursively called on the right child node.

### Inserting a New Node in a Binary Search Tree

The insert function is used to add a new node with a given value at the correct position in the binary search tree. Adding the node at the correct position means that the new node should not violate the properties of the binary search tree.

### Deleting a Node from a Binary Search Tree

The delete function deletes a node from the binary search tree. However, utmost care should be taken that the properties of the binary search tree are not violated and nodes are not lost in the process. Case in Which a node is deleted from a binary search tree

#### Case 1: Deleting a Node that has no Children

Simply remove the node. This is the simplist case.

#### Case 2: Deleting a Node with One Child

To hanlde this case, the node's child is set as the childe of the node's parent. In other words, replace the node with its child. Now, if the node is the left child of its parent, the node's child becomes the left child of the node's parent. Correspondingly, if the node is the right child of its parent, the node's child becomes the right child of the node's parent.

#### Case 3: Deleting a Node with Two Children

Th handle this case, replace the node's value with its **in-order predecessor** (largest value in the left sub-tree). The **in-order** predecessor or the successor can then be deleted using any of the above cases.

### Determing the Height of a Binary Search Tree

In order to determine the height of a binary search tree, we calculate the height of the left sub-trr and the right sub-tree. Whichever height is greater, 1 is added to it.

**Determining the Number of Nodes**

Determining the number of nodes in a binary search tree is similar to determining its height. To calculate the total number of elements/nodes in the tree, we count the number of nodes in the left sub-tree and the right sub-tree.

```
Number of nodes = totalNodes(left sub-tree) + totalNodes(right sub-tree) + 1
```

**Determing the Number of Internal Nodes**

To calculate the total number of internal nodes or non-leaf nodes, we count the number of internal nodes in the left sub-tree and the right sub-tree and add 1 to it ( 1 is added for the root node).

```
Number of internal nodes =

totalInternalNodes(left sub-tree + totalInteralNodes(right sub-tree) + 1
```

**Determinging the Number of External Nodes**

To calculate the total number of external nodes or leaf nodes, we add the number of external nodes in the left sub-tree and the right sub-tree. However if the tree is empty, that is `TREE = NULL`, then the number of external nodes will be zero. But if there is only one node in the tree, then the number of external nodes will be one.

```
Number of external nodes =

totalExternalNoses(left sub-tree) + totalExternalNodes(right sub-tree)
```

**Finding the Mirror Image of a Binary Search Tree**

Mirror image of a binary search tree is obtained by interchanging the left sub-tree with the right sub-tree at every node of the tree. Given a tree $T$, the mirror image of $T$ can be obtained as $T'$.

**Deleting a Binary Search Tree**

To delete/remove an entire binary search tree from the memory, we first delete the elements/nodes in the left sub-tree and then delete the nodes in the right sub-tree.

**Finding the Smallest Node in a Binary Search Tree**

The very basic property of the binary search tree states that the smaller value will occur in the left sub-tree. If the left sub-tree is `NULL`, then the value of the root node will be smallest as compared to the nodes in the right sub-tree. So, to find the node with the smallest value, we find the value of the leftmost node of the left sub-tree.

**Finding the Largest Node in a Binary Search Tree**

To find the node with the largest value, we find the value of the right most node of the right sub-tree. However, if the right sub-tree is empty, then the root node will be the largest value in the tree.

**Threaded Binary Trees**

A threaded binary tree is the same as that of a binary tree but with a difference in storing the `NULL` pointers. In the linked representation, a number of nodes contain a `NULL` pointer, either in their lef or right fields or in both. This space that is wasted in storing a `NULL` pointer can be efficiently used to store other useful information. For example, the `NULL` entries can be replaced to store a pointer to the in-order predecessor or the in-order successor of the node. These special pointers are called **threads** and binary trees containing threads are called **threaded trees**. A threaded binary tree may correspond to one-way threading or a two-way threading.

In one-way threading, a thread will appear either in the right field or the left field of the node. A one-way threaded tree is also called a single-threaded tree. If the thread appears in the left field, then the left field will be made to point to the in-order predecessor of the node. Such a threaded binary a one-way threaded tree is called a left-threaded binary tree. On the contrary, if the thread appears in the right field, then it will point to the in-order successor of the node. Such a one-way threaded tree is called a tight-threaded binary tree.

In a two-way threaded tree, also called a double-threaded tree, threads will appear in both the left and the right field of the node. While the left field will point to the in-order predecessor of the node, the, the right field will point to its sucessor. A two-way threaded binary tree is also called a fully threaded binary tree.

**Traversing a Threaded Binary Tree**

For every node, visit the left sub-tree firs, provided if one exists and has not been visited earlier. Then the node (root) itself is followed by visiting its right sub-tree (if one exists). In case there is no right sub-tree, check for the threaded link and make the threaded node the current node in consideration.

**Advantages of Threaded Binary Tree**

- It enables linear traversal of elements in the tree.

- Linear traversal eliminates the use of stacks which in turn consume a lot of memory space and computing time.

- I enables to find the parent of a given element without explicit use of parent pointers.

- Since nodes contain pointers to in-order predecessor and successor, the threaded tree enables forward and backward traversal of the nodes as given by in-order fashion.

Thus, the basic difference between a binary tree and a threaded binary tree is that in binary trees a node stores a `NULL` pointer if it has no child and there is no way to traverse back.

**AVL Trees**

AVL tree is a self-balancing binary search tree. In an AVL tree, the heights of the two sub-trees of a node may differ by a most one. Due to this property , the AVL tree is also known as a height-balance tree.

The structure of an AVL tree is the same as that of a binary search tree. In its structure, it stores an additional variable called the `BalanceFactor`. Thus, every node has a balance factor assocaiated with it. The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree. A binary search tree in which every node has a balance $-1, 0$ or 1 is said to be height balanced. A node with any other balance factor is considered to be unbalanced and requires rebalancing of the tree.

• If the balance factor of a node is 1, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is therefore called as a **left-heavy tree**

• If the balance factor of a node is 0, then it means that the height of the left sub-tree (longest path in the left sub-tree) is equal to the height of the right sub-tree.

• If the balance factor of a node is $-1$, then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is therefore called a **right-heavy tree**.

```
Balance factor = Height(left sub-tree) - Height(right sub-tree)
```

Insertions and deletions from an AVL tree may disturb the balance factor of the nodes and thus, rebalancing of the tree may have to be done. The tree is rebalanced by performing rotation at the critical node. There are four types of rotations: `LL` rotation, `RR` rotation, `LR` rotation, and `RL` rotation. The type of rotation that has to be donw will vary depending on the particular situation.

**Operations on AVL Trees**

**Searching for a Node in an AVL Tree**

Searching in an AVL tree is also done in the same way it is done in a binary tree. In the AVL tree, the new node is always inserted as the leaf node. But the step of insertion is usually followed by an additional step of rotation. Rotation is done to restore the balance of the tree. However, if insertion of the new node does not distrub the balance factor, that is, if the balance factor of every node is still $-1, 0$ or 1, then rotations are not required.

During insertion, the new node is inserted as the leaf node, so it will have a balance factor equal to zero. The only nodes whose balance factors will change are those which lie in the path between the root of the tree and the newly inserted node. The possibility changes which may take place in any node on the path as follow:

- Initially, the node was either left- or right-heavy and after insertion, it becomes balanced.
- Initially, the node was balanced and after insertion, it becomes either left- or right-heavy.
- Initially, the node was heavy (either left or right) and the new node has been inserted in the heavy sub-tree, thereby creating an unbalanced sub-tree. Such a node is said to be a critical node.

To perform a rotation, our first task is to find the critical node. Critical node is the nearest ancestor node on the path from the inserted node to the root whose balance factor is neither $-1, 0$ nor 1. The second task in rebalancing the tree is to determine which type of rotation has to be done. There are four types of rebalancing rotations and applications of these rotations depends on the position of the inserted node with reference to the criticsl node. The four categories are:

- `LL` rotation The new node is inserted in the left sub-tree of the left sub-tree of the critical node.
- `RR` rotation The new node is inserted in the right sub-tree of the right sub-tree of the critical node.
- `LR` rotation The new node is inserted in the right sub-tree of the left sub-tree of the critical node.
- `RL` rotation The new node is inserted in the left sub-tree of the right sub-tree of the critical node.

**Deleting a Node from an AVL Tree**

Deletion of a node in a AVL tree is similar to that of a binary search trees. Deletion may disturb the AVLness of the tree, so to rebalance the AVL tree, we need to perform rotations. There are tewo classes of rotations that can be performed on an AVL tree after deleting a given node. There rotations are $R$ and $L$ rotation.

On deletion of node $x$ from the AVL tree, if node $A$ becomes the critical node (closest ancestor node on the path from $x$ to the root node that does not have its balance factor as $1, 0$ or $-1$, then the type of rotation depdens on whether $x$ is in the left sub-tree of $A$ or in its right sub-tree. If the node to be deleted is present in the left sub-tree of $A$, then $L$ rotation is applied, else if $x$ is in the right sub-tree, $R$ rotation is performed.

Further, there are three categories of $L$ and $R$ rotations. The variations of $L$ rotation are $L-1$, $L0$, and $L1$ rotation. Correspondingly for $R$ rotation, there are $R0$, $R-1$, and $R1$ rotations.

**$R0$ Rotation**

Let $B$ be the root of the left or right sub-tree of $A$ (critical node). $R0$ is applied if the balance factor of $B$ is 0.

**$R1$ Rotation**

Let $B$ be the root of the left or right sub-tree of $A$ (critical node). $R1$ rotation is applied if the balance factor of $B$ is 1. Observe that $R0$ and $R1$ rotations are similiar to $LL$ rotations; the only difference is that $R0$ and $R1$ rotations yield different balance factors.

### $R-1$ Rotations

Let $B$ be the root of the left or right sub-tree of $A$ (critical node). $R-1$ rotation is applied if the balance factor of $B$ is $-1$ rotation is similar to $LR$ rotation.

### Red-Black Trees

A red-black tree is a self-balancing binary search tree. Practically, a red-black tree is a binary search tree which inserts and removes intelligently, to keep the tree reasonably balanced. A special point to note about the red-black tree is that in this tree, no data is store in the leaf nodes.

### Properties of Red-Black Trees

A red-black tree is a binary search tree in which every node has a color which is either red or black. Apart from the other restrictions of a binary search tree, the red-black tree has the following additional requirements:

(1) The color of a node is either red or black

(2) The color of the root node is always black

(3) All leaf nodes are black

(4) Every red node has both the children colored in black.

(5) Every simple path from a given node to any of its leaf nodes has an equal number of black nodes.

These constraints enforce a critical property of red-black trees. The longest path from the root node to any leaf is no more than twice as long as the shortest path from the root to any other leaf in that tree.

**Operations on Red-Black Trees**

Preforming a read-only operation (like traversing the nodes in a tree) on a red-black tree is a special case of a binary tree. However, insertion and deletion operations may violate the properties of a red-black search tree.

**Inserting a Node in a Red-Black Tree**

The insertion operation starts in the same way as we add a new node in the binary search tree. However, in a binary search tree, we always add the new node as a leaf. while in a red-black tree, leaf nodes contain no data. So instead of adding a new leaf node, we add a red interior node that has two black leaf nodes. Note that the color of the new node is red and its leaf nodes are colored black.

Once a new node is added, it may violate some properties of the red-black tree. So in order to restore their property, we check for certain cases and restore the property depdending on the case that turns up after insertion.

**Grandparent node** ($G$) of a node ($N$) refers to the parent of $N$'s parent ($P$). The C code to find a node's grandparent can be given as follows:

```
struct node* grand_parent(struct node *n) {
       // No parent means no grandparent
       if((n ≠ NULL && (n -> parent ≠ NULL))
            return n-> parent -> parent;
else
            return NULL;
}
```

**Uncle node** ($U$) of a node ($N$) refers to the sibling of $N$'s parent ($P$)

```
struct node* g {
g = grand_parent(n);
// With no grandparent, there cannot be any uncle
if(g == NULL)
       return NULL;
if(n -> parent == g -> left)
       return g -> right;
else
       return g -> left;
}
```

When we insert a new node in a red-black tree, note the following

- All leaf nodes are always black. So property 3 always holds true.

- Property 4 (both children of every red node are black) is threatened only by adding a red node, repainting a black node red, or a rotation,

- Property 5 (all paths from any given node to its leaf node has equal number of black nodes) is threatened only by adding a black node, repainting a red node black, or a rotation.

**Case 1: The New Node $N$ is Added as the Root of the tree**

In this case, $N$ is repainted black, as the root of the tree is always black. Since $N$ adds one black node to every path at once, Property 5 is not violated. The C code for case 1 can be given as

```
void case1(struct node* n) {
     if(n-> parent == NULL    // Root node
         n -> color = BLACK;
     else
         case2(n)
}
```

**Case 2: The New Node's Parent $P$ is Black**

In this case, both children of every red node are black, so Property 4 is not invalidated. Property 5 is also not threatened. This is because the new node $N$ has two black leaf children, but because $N$ is red, the paths through each of its children have the same number of black nodes. The C code to check for case 2 can be given as:

```
void case2(struct node* n) {
     if(n-> parent == BLACK
         return;    /* Red black tree property is not violated */
     else
         case3(n)
}
```

In the following cases, it is assumed that $N$ has a grandparent nde $G$, because its parent $P$ is red, and if it were the root, it would be black. Thus, $N$ also has an uncle node $U$

**Case 3: If Both the Parent ($P$) and the Uncle ($U$) are Red**

In this case, Property 5 which says all paths from any given node to its leaf nodes have an equal number of black nodes is violated. In order to restore Property 5, both nodes ($P$ and $U$) are repainted black and the grandparent $G$ is repainted red. Now, the new red node $N$ has a black parent. Since any path through the parent or uncle must pass through the grandparent, the number of black nodes on these paths has not changed.

However, the grandparent $G$ may now violate Property 2 which says that the root node is always black or Property 4 which states that both children of every red node are black. Property 4 will be violated when $G$ has a red parent. In order to fix this problem, this entire procedure is recursively performed on $G$ from Case 1. The $C$ code to deal with Case 3 insertion can be given as:

```
void case3(struct node* n) {
     struct node *u, *g;
     u = uncle(n);
     g = grand_parent(n)
     if((i≠ NULL) && (u -> color == RED)) {
          n -> parent -> color = BLACK;
          u -> color = BLACK;
          g -> color = RED;
          Case1(g);
     }
     else {
          insert_case4(n);
     }
}
```

**Case 4: The Parent $P$ is Red but the Uncle $U$ is Black and $N$ is the Right Child of $P$ and $P$ is the Left Child of $G$**

In order to fix this problem, a left rotation is done to switch the roles of the new node $N$ and its parent $P$. After the rotation, note in the C code, we have re-labelled $N$ and $P$ and then, Case 5 is called to deal with the node's parent. This is done because Property 4 which says both children of every node should be black is still violated. Noted that in case $N$ is the left child of $P$ and $P$ is the right child of $G$, we have to perform a right rotation. In the C code that handles Case 4, we check for $P$ and $N$ and then, perform either a left or right rotation.

```
void case4(struct node* n) {
     struct node *g = grand_parent(n);
     if((n== n -> parent -> right) && (n -> parent == g -> left)) {
          rotate_left(n -> parent);
          n = n -> left;
     }
     else if((n == n-> left) && (n -> parent == g -> right)) {
          rotate_right(n -> parent);
          n = n -> right;
     }
     case5(n);
}
```

**Case 5: The Parent $P$ is Red but the Uncle $U$ is Black and the New Node is the Left Child of $P$, and $P$ is the Left Child of its Parent $G$**

In order to fix this problem, a right rotation on $G$ (the grandparent of $N$) is performed. After this rotation, the former parent $P$ is now the parent of both the new node $N$ and the former grandparent $G$. We know that the color of $G$ is black (because otherwise its former child $P$ could not have been red), so now switch the colors of $P$ and $G$ so that the resulting tree satisfies Property 4 which states that both children of a red node are black. Note that in case $N$ is the right child of $P$ and $P$ is the right child of $G$, we perform a left rotation. In the C code that handles Case 5, we check for $P$ and $N$ and then, perform either a left or a right rotation.

```
void case5(struct node* n) {
     struct node *g;
     if((n== n -> parent -> left) && (n -> parent == g -> left))
          rotate_right(g);
     else if((n == n-> left) && (n -> parent == g -> right))
          rotate_left(g);
     n -> parent -> color = BLACK;
     g -> color = RED;
}
```

**Deleting a Node from a Red-Black Tree**

We start deleting a node from a red-black tree in the same way we do in case of a binary search tree. In a binary seach tree, when we delete a node with non-leaf children, we find either the maximum element in its left sub-tree of the node or the minimum element in its right sub-tree, and move its value into the node being deleted. After that, we delete the node from which we had copied the value. Note that this node must have less than two non-leaf children. Therefore, merely copying a value does not violate any red-black properties, but it just reduces the problem of deleting to the problem of deleting a node with at most one non-leaf child.

While deleting a node, if its color is red, then we can simply replace it with its child, which must be black. All paths through the deleted node will simply pass through one less red node, and both the deleted node's parent and child must be black, so none of the properties will be violated.

Another simple case is when we delete a black node that has a red child. In this case, Property 4 and Property 5 could be violated, so to restore them, just repaint the deleted node's child with black.

A complex situation arises when both the node to be deleted as well as its child is label the child node as (in its new position) $N$, and its siblings (its new parent's other child) as $S$. The C code to find the sibling of a node can be given as:

```
struct node *sibling(struct node *n) {
       if(n== n-> parent ->)
           return n -> parent -> right;
       else
           return n -> parent -> left;
}
```

We can start the deletion process by using the following code, where the function `replace_node` substitutes the `child` into $N$'s place in the tree. We assume null leaves are represented by actual node object, rather than `NULL`.

```
void delete_child(struct node *n) {
     /* If N has at most one non-null child */
     struct node *child;
     if(is_leaf(n -> right))
               child = n -> left;
     else
               child = n -> right;
     replace_node(n, child);
     if(n -> color == BLACK) {
               if(child-> color == RED)
                       child -> color == BLACK;
               else
                       del_case1(child);
               }
               free(n);
}
```

When both $N$ and its parent $P$ are black, then deleting $P$ will cause paths which precede through $N$ to have one fewer black nodes than the other paths. This will violate Property 5. Therefore, the tree needs to be reblanced. There are several cases to consider.

**Case 1: $N$ is the New Root**

In this case, we hve removed one black node from every path, and the new root is black, so none of the properties are violated.

```
void del_case1(struct node *n) {
     if( n-> parent ≠ NULL)
          del_case2(n);
}
```

**Case 2: Sibling $S$ is Red**

In this case, interchange the colors of $P$ and $S$, and then rotate left $P$. In the resultant tree, $S$ will become $N$'s grandparent. The C code that handles Case 2 deletion can be given as

```
void del_case2(struct node *n) {
     struct node *s;
     s = sibling(n);
     if(s-> color == RED) {
         if(n== n -> parent -> left)
             rotate_left(n -> parent);
         else
             rotate_right(n -> parent);
         n -> parent -> color = RED;
         s -> color = BLACK;
     }
     del_case3(n);
}
```

**Case 3: $P, S$ and $S$'s Children are Black**

In this case, simply repaint $s$ with red. In the resultant tree, all the paths passing through $S$ will have one less black node. Therefore, all the paths that pass through $P$ now have one fewer black nodes than the path that do not pass through $P$, So Property 5 is still violated. To fix this problem, we perform the rebalancing procedure on $P$, starting at Case 1. The C code fo Case 3 can be given as:

```
void del_case3(struct node *n) {
     struct node *s;
     s = sibling(n);

     if((n-> parent -> color == BLACK) && (s -> color == Black) &&
     (s -> left -> color == BLACK && (s -> right-color == BLACK)) {
          s -> color = RED;
          del_case1(n -> parent);
          } else
                del_case4(n);
}
```

**Case 4: $S$ and $S$'s Children are Black, but $P$ is Red**

In this case, we interchange the colors of $S$ and $P$. Although this will not affect the number of black nodes on the paths going through $S$, it will add one black node to the paths going through $N$, making up for the deleted black node on those paths. The C code handle Case 4 can be given as:

```
void del_case4(struct node *n) {
     struct node *s;
     s = sibling(n);

     if((n-> parent -> color == RED) && (s -> color == Black) &&
     (s -> left -> color == BLACK && (s -> right-color == BLACK)) {
          s -> color = RED;
          n -> parent -> color = BLACK;
          } else
                del_case5(n);
}
```

**Case 5: $N$ is the Left Child or $P$ and $S$ is Black, $S$'s Left Child is Red, $S$'s Right Child is Black**

In this case, perform a right rotation at $S$. After the rotation, $s$'s left child becomes $S$'s parent and $N$'s new sibling. Also, interchange the colors of $S$ and its new parent. Note that all paths still have equal number of black nodes, but $N$ has a black sibling whose right child is red, So we fall into Case 6. The C code to hanle Case 5 is given as:

```
void del_case5(struct node *n) {
     struct node *s;
     s = sibling(n);

     if(s-> color == BLACK) {
     /* the following code forces the red to be on the left of the left of the parent
     * or right of the right, to be correctly operated in Case 6.
         if((n-> parent -> left) && (s -> right -> color == BLACK) &&
              (s -> left -> color == RED))
         else if ((n == n -> parent -> right) && (s -> left -> color == BLACK) &&
         s -> right -> color == RED))
         rotate_left(s);
         s -> color == RED;
         s -> right -> color = BLACK;
     }
     del_case6(n);
}
```

**Case 6: $S$ is Black, $S$'s Right Child is Red, and $N$ is the Left Child of its Parent $P$**

In this case, a left rotation is done at $P$ to make $S$ the parent of $P$ and $S$'s right child. After the rotation, the colors of $P$ and $S$ are interchanged and $S$'s right child is colored black. One these steps are followed, you will observed that Property 4 and Property 5 remain vaild. The C code to fix ase 6 can be given as:

```
void del_case5(struct node *n) {
     struct node *s;
     s = sibling(n);
     s -> color = n -> parent -> color;
     n -> parent -> color = BLACK;

     if(n== n -> left) {
         s -> right -> color = BLACK;
         rotate_left(n -> parent);
     } else {
         s -> right -> color == BLACK
         rotate_right(n -> parent);
     }
}
```

**Applications of Red-Black Trees**

Red-black trees are not only valuable in time-sensitive applications such as real-time applications, but are also preferred to be used as a building block in other data structures which provide worst-case guarantee.

**Splay Trees**

A splay tree is a self-balancing binary search tree with an additional property that recently accessed elements can be re-accessed fast. For many non-uniform sequences of operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown.

A splay tree consists of a binary tree, with no additional fields. When a node in a splay tree is accessed, it is rotated or 'splayed' to the root, thereby changing the structure of the tree. Since the most frequently accessed node is always moved closer to the starting point (the root node), these nodes are therefore located faster.

In a splay tree, operations such as insertions, search, and deletion are combined with one basic operation called **splaying**. Splaying the tree for a particular node rearranges the tree to place that node at the root. A technique to do this is to first perform a standard binary tree search for that node and then use rotation in a specific order to bring the node on top.

**Operations on Splay Trees**

The four main operations that are performed on a splay tree. These include splaying, insertion, search, and deletion.

**Splaying**

When we access a node $N$, splaying is performed on $N$ to move it to the root. To perform a splay operation, certain **splay steps** are performed where each step moves $N$ to the root. Splaying a particular node of interest after the tree remains roughly balanced.

Each splay step depends on three facts:

- Whether $N$ is the left or right child of its parent $P$
- Whether $P$ is the root or not, and if not
- Whether $P$ is the left or right child of its parent, $G$ ($N$'s grandparent).

Depending on these factors, we have one splay step on each factor.

**Zig step**

The `zig` operation is done when $P$ (the parent of $N$) is the root of the splay tree. In the `zig` step, the tree is rotated on the edge between $N$ and $P$. `zig` step is usually performed as the last step in a splay operation and only when $N$ has an odd depth at the beginning of the operation.

**Zig-zig step**

The `zig-zig` operation is performed when $P$ is not the root. In addition to this, $N$ and $P$ are either both right or left children of their parents. During the `zig-zig` step, first the tree is rotated on the edge joining $P$ and its parent $G$, and then again rotated on the edge $N$ and $P$.

**Zig-zag step** The `zig-zag` operation is performed when $P$ is not the root. In addition to this, $N$ is the right child of $P$ and $P$ is the left child of $G$ or vice verse. In `zig-zag`, the tree is first rotated on the edge between $N$ and $P$, and then rotated on the edge between $N$ and $G$.

**Inserting a Node in a Sply Tree**

Although the process of inserting a new node into a splay tree begins in the same way as we insert a node in a binary search tree, but after the insertion, $N$ is made the new root of the splay tree. The steps performed to insert a new node $N$ in a splay tree can be given as:

**Step 1** Search $N$ in the splay tree. If the search is successful, splay at the node $N$.

**Step 2** If the search is unsuccessful, add the new node $N$ in such a way that it replaces the `NULL` pointer reached during the search by a piinter to a new node $N$. Splay the tree at $N$.

**Searching a Node in a Splay Tree**

If a particular node $N$ is present in the splay tree, then a pointer to the null node is returned. The steps performed to search a node $N$ in a splay tree include:

- Search down the root of the splay tree looking for $N$.
- If the search is successful, and we reach $N$, then splay the tree at $N$ and return a pointer to $N$.
- If the search is unsuccessful, i.e., the splay tree does not contain $N$, then we reach a null node. Splay the tree during the search and return a pointer to null.

**Deleting a Node from a Splay Tree**

To delete a node $N$ from a splay tree, we perform the following steps:

- Search for $N$ that has to be deleted. If the search us unsuccessful, splay the tree at the last non-null node encountered during the search.

• If the search is successful and $N$ is not the root node, then let $P$ be the prent of $N$. Replace $N$ by an appropriate descendent of $P$. Finally splay the tree.

**Advantages and Disadvantages of Splay Trees**

The advantages of using a splay tree are:

• A splay tree gives good performance for search, insertion, and deletion operations. This advantage centers on the fact that the splay tree is a self-balancing and a self-optimizing data structure in which the frequently accessed nodes are moved closer to the root so that they can be accessed quickly. This advantage is particularly useful for implementing caches and garbage collection algorithms.

• Splay trees are considerably simpler to implement than the other self-balancing binary search trees, such as red-black trees or AVL trees, while their average-case performance is just as efficient.

• Splay trees minimize memory requirements as they do not store any book-keeping data.

• Unlike other types of self-balancing trees, splay trees provide good performance with nodes containing identical keys.

However, here are some disadvantages of splay trees:

• While sequentially accessing all the nodes of a tree in a sorted order, the resultant tree becomes completely unbalanced. This takes $n$ accesses of the tree in which each access in turn takes $O(\log n)$ time. For example. re-accessing the first node triggers an operation that in turn takes $O(n)$ operations to rebalance the tree before returning the first node. Although this creates a signigicant delay for the final operation, the amortized performance over the entire sequence is still $O(\log n)$.

• For uniform access, the performance of a splay tree will be considerably worse than a somewhat balanced simple binary search tree. For uniform access, unlike splay trees, these other data structures provide worst-case time guarantess and can be more efficient to use.

**Points to Rememeber**

• A binary search tree, also known as an ordered binary tree, is a variant of binary tree in which all the nodes in the left sub-tree have a value less than that of the root and all the nodes in the right sub-tree have a value either equal to or greater than the root node.

• The average running time of a search operation is $O(\log_2 n)$. However, in the worst case, a binary search tree will take $O(n)$ time to search an element from the tree.

• Mirror image of a binary search tree is obtained by interchanging the left sub-tree with the right sub-tree at every node of the tree.

• In a threaded binary tree, null entries can be replaced to store a pointer to either the in-order predecessor or in-order successor of a node.

• A one-way threaded tree is also called a single threaded tree. In a two-way threaded tree, also called a double threaded tree, threads will appear in both the left and the right field of the node.

• An AVL tree is a self-balancing tree which is also known as a height-balanced tree. Each node has a balance factor associated with it, which is calculated by subtracting the height of the right sub-tree from the height of the left sub-tree. In a height balanced tree, every node has a balance factor of either $0, 1$, or $-1$.

• A red-black tree is a self-balacing binary search tree which is also called as a 'symmetric binary B-tree'. Although a red-black tree is complex, it has good worst case running time for its operations and is efficient to use, as searching, insertion, and deletion can all be done int $O(\log n)$ time.

• A splay tree is a self-balancing binary search tree with an additional property that recently accessed elements can be re-accessed fast.