SPLAY Tree

Mirza Mohammad Azwad (Student ID: 200042121)

CSE 4303 Data Structures
BScEngg in Software Engineering(SWE), Department Computer Science and Engineering, Islamic University of Technology(IUT)
Board Bazar, Gazipur, Dhaka, Bangladesh
E-mail: mirzaazwad@iut-dhaka.edu

11th December, 2022

I Topics

- What is a Splay Tree?
- What are the operations we can perform on a Standard Splay Tree?
- Advantages and Disadvantages
- Sample Code of a basic implementation of Splay Tree
- Complexity Analysis
- Usecases (Describe a problem where we can use Splay Tree)

II What is a Splay Tree?

1 Introduction

Splay trees are another form of self-adjusted Binary Search Trees similar to Red Black Trees, AVL Trees, and B Trees. Earlier we learned in class that in certain cases a Binary Search Tree can become degenerate Binary Trees which would in other words be skewed as left-skewed or right-skewed binary trees. This creates an issue as the best case that we are looking for has a time complexity of $\mathbf{O(log_2 n)}$ but this cannot be attained as a result of skewing which causes the worst case that being $\mathbf{O(n)}$.

In this semester, we came across red-black trees and AVL trees that are self-balancing. AVL Trees are more strictly balanced than red-black trees. Now we come across splay trees, which are another form of balanced trees, which are also **roughly balanced** similar to red-black trees.

III What are the operations we can perform on a Standard Splay Tree?

In splay trees, the fundamental property is that after every operation, the value it is operated on becomes the root. This property is called splaying, splaying is defined as the operations performed on any element which cause that element to become the root upon certain rearrangements performed in the form of rotations.

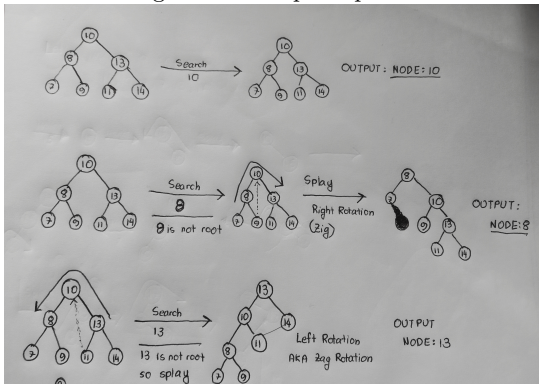
1 Searching

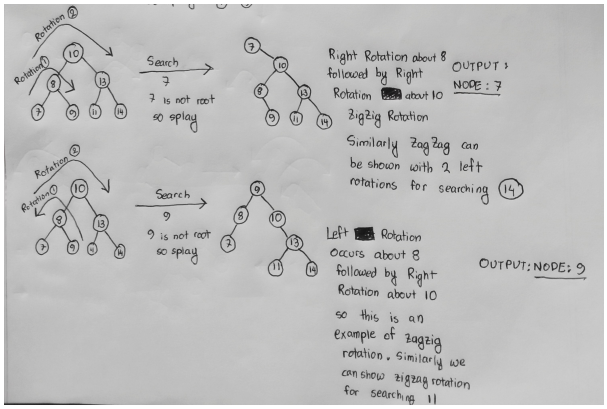
In splay trees, the searched value becomes the root upon being found, so essentially the difference with BST lies in the fact that splay trees when searching have 2 operations being performed, namely the basic BST searching followed by splaying to rotate the tree and making the searched element root by the use of rotations. Now let's observe searching in splay trees, the following points should be kept in mind:

[1]Definitions acquired from Javapoint at https://www.javatpoint.com/splay-tree

- If we are searching for root, then no splaying is required
- if child is to the left of the root, we do a right rotation and make child the root of the tree, this is an example of zig rotation.
- if child is to the right of the root, we do a left rotation and make child the root of the tree, this is an example of zag rotation.
- Now let's say that the child's parent is not root, and maybe the grandparent isn't root either, we might need to do multiple rotations, say child is left child of left subtree, we might need to do 2 right rotations, which would result into 2 zig rotations, so its called zigzig rotation. Similarly, we can observe a similar case for zagzag rotations.
- Now there are cases where we would need to do zigzag rotations or zigzig rotations. For instance, let's say we are searching for the right child of the left subtree, we have to first do a left rotation about the root of the left subtree, followed by a left rotation about the root. Similar cases can be shown for zagzig rotations.

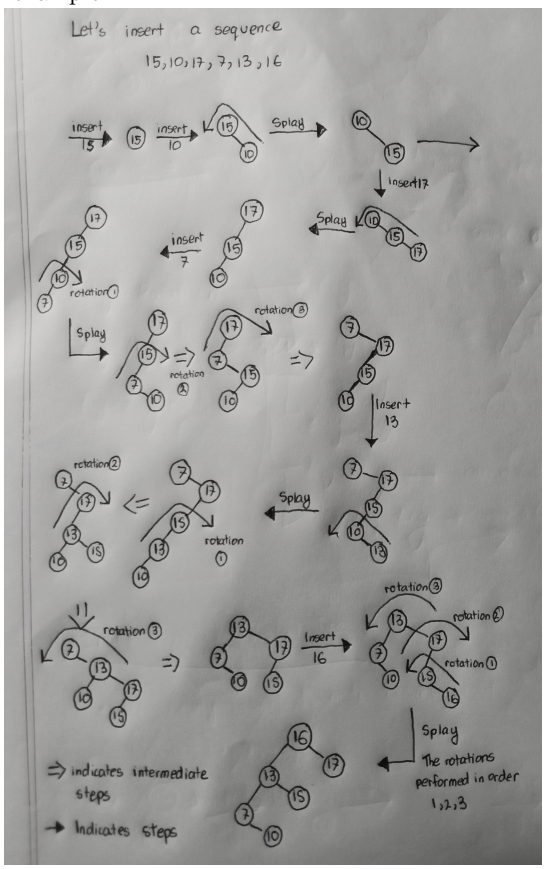
The terms may be defined differently in different resources, some resources argue that zig rotations are both zig and zag, with zigzig being both zigzig and zagzag. As well as zagzig and zigzag being both defined as zig-zag rotations. But regardless the principle remains nearly the same.





2 Insertion

In Insertion similar to searching we need to perform splaying operations in the manner mentioned above. The newly inserted element becomes the root. So in order to do that we perform various types of rotation as per the requirement at that state. First and foremost the insertion is done similarly to Binary Search Tree, the difference lies in the splaying operation performed following insertion. It can be illustrated with an example:

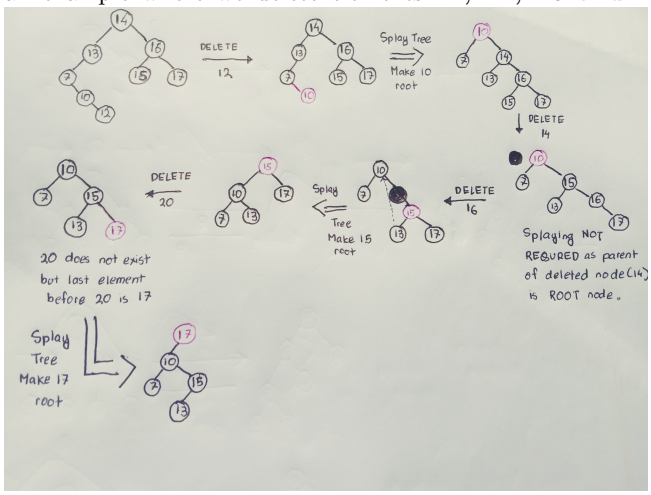


3 Deletion

Deletion is done similarly to Binary Search Tree with the added splaying operation following the deletion operation. Now the question remains since we are deleting the element, which element are we supposed to splay on? This is where we get the two different types of splaying:

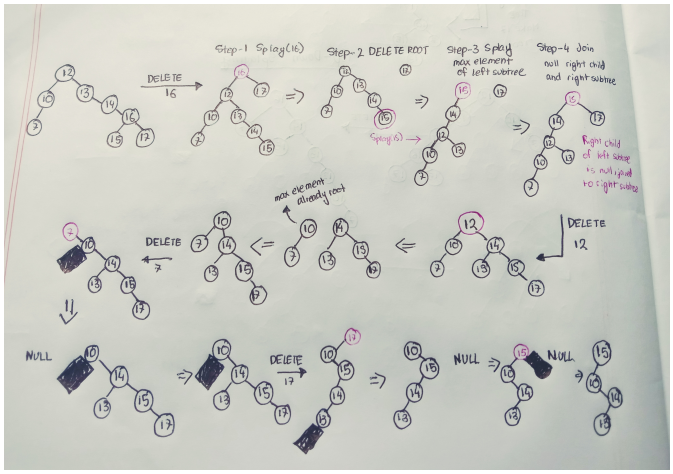
A Bottom-Up Splaying

Previously we splayed on the new element or searched element. Both of these have an element existing but in this case, the element to be deleted is deleted, and the splaying is performed on the parent of the element to be deleted. Furthermore, we have to keep in mind that if an element to be deleted is not found, then we still have to perform splaying on the last node visited before concluding that this element does not exist. These cases can be better illustrated with an example where we delete elements **12, 14, 16 and 20**:



B Top-Down Splaying

This method is slightly more complicated than its predecessor in this document. But nevertheless based on use cases, either type of deletion may be applied. In the top-down approach, the idea of top-down comes from the fact that we are first performing splaying and then we are performing deletion as compared to the previous approach where we performed deletion followed by splaying. The sequence of operations is reversed. We splay the tree and make the element we want to delete the root node. Then we delete the node which results in two different subtrees. We then splay the left subtree to obtain the maximum element of the left subtree and join it to the right subtree. If the left subtree is null, we can simply state that the right subtree is the answer. But if the right subtree is null, we have to first splay the left subtree and then the final result after splaying is the answer. All of these cases can be better illustrated with an example where we delete the elements **16, 12, 7 and 17**:



IV Splitting

In this operation, a tree is essentially split into two about a node. This is similar to how we performed top-down deletion except we don't delete the node, we can have a left-inclusive split and right-inclusive split where the left subtree contains the root and the right subtree contains the root respectively.

V Advantages and Disadvantages

1 Advantages

The most frequently accessed element is either at the root or near the root, which makes that value available at O(1) time complexity in the best case. Or in other words, frequently accessed elements being near the root reduces access time and thus makes it efficient for such values. The fast access for frequently accessed values makes it more advantageous than red-black trees or AVL trees in certain practical scenarios such as caches. The advantages can be summarised as follows:

- no extra pieces of information stored such as in Red Black Tree we store the color of the node or in AVL tree we need to store the balance factor, thus making it relatively memory efficient(as it only stores the value) compared to the other two discussed types of Binary Search Tree.
- fastest type of Binary Search Tree for certain practical situations such as Windows NT, GCC compilers, and garbage collection in various other compilers.
- easy implementation which makes it helpful in certain general cases of problem-solving such as being able to do most versions of range query problems
- better performance because frequently searched values move closer to the root, thus it can be used in implementations of caches to deal with temporal locality

2 Disadvantages

The main disadvantage of the splay tree is the left frequently accessed elements may take up to O(n) time to access especially because the tree might get skewed. Although it's not something that occurs very frequently in practical scenarios it is still a possibility nevertheless.

[2]Operations on Splay Tree acquired from Jenny's Lectures CS IT at
https://www.youtube.com

VI Sample Code of a basic implementation of Splay Tree

The language used here to carry out this implementation is **C++**.

1 Nodes Used in Tree

```
template <typename T>class Node
{
public:
    T data;
    Node<T> *parent;
    Node<T> *right;
    Node<T> *left;
    Node(T value)
    {
        data = value;
        left = nullptr;
        right = nullptr;
        parent = nullptr;
    }
};
```

2 Tree Class

```
template <typename T>class SplayTree
{
private:
    Node<T> *root;

public:
    SplayTree()
    {
        root = nullptr;
    }
    void Insert(T Key);
    Node<T>* Search(T key, Node<T> *t = root);
    void Delete(T Key);
    void Splay(Node<T>* node);
    void RightRotation(Node<T> node);
    void LeftRotation(Node<T> node);
};
```

3 Insert Node Into Tree

```
void Insert(T key)
{
  Node<T> *insertNode = new Node<T>(key);
  Node<T> *t = root;
  Node<T> *temp = nullptr;
  while (t != nullptr)
  {
    temp = t;
    if (insertNode->data < t->data)
    {
      t = t->left;
    }
    else
    {
      t = t->right;
    }
  }
  insertNode->parent = temp;
```

```
  if (temp == nullptr)
  {
    root = insertNode;
  }
  else if (insertNode->data < temp->data)
  {
    temp->left = insertNode;
  }
  else
    temp->right = insertNode;
  Splay(insertNode);
}
```

4 Splayed Search Node In Tree

```
Node<T> *Search(T key,
Node<T> *currentNode = root,
Node<T> *prevNode = nullptr)
{
  if (currentNode == nullptr)
    return prevNode;
  if (key == currentNode->data)
  {
    Splay(currentNode);
    return currentNode;
  }
  else if (key < currentNode->data)
  {
    return Search(key,
    currentNode->left, currentNode);
  }
  else
    return Search(key,
    currentNode->right, currentNode);
}
```

5 Unsplayed Search Node In Tree

```
Node<T> *UnsplayedSearch(T key,
Node<T> *currentNode = root,
Node<T> *prevNode = nullptr)
{
  if (currentNode == nullptr)
    return prevNode;
  if (key == currentNode->data)
  {
    return currentNode;
  }
  else if (key < currentNode->data)
  {
    return UnsplayedSearch(key,
    currentNode->left, currentNode);
  }
  else
    return UnsplayedSearch(key,
    currentNode->right, currentNode);
}
```

6 Bottm Up Delete in Splay Tree

```
void BottomUpDelete(T key)
{
  Node<T> *deleteNode =
  UnsplayedSearch(key);
  if(deleteNode->data!=key){
    Splay(deleteNode);
    return;
    }
  Node<T> *parent =
  deleteNode->parent;
  if (deleteNode->left == nullptr)
  {
    Transplant(deleteNode,
    deleteNode->right);
  }
  else if (deleteNode->right == nullptr)
  {
    Transplant(deleteNode,
    deleteNode->left);
  }
  else
  {
    Node<T> *InorderSuccessor =
    Successor(deleteNode->right);
    if (InorderSuccessor->parent !=
    deleteNode)
    {
      Transplant(InorderSuccessor,
      InorderSuccessor->right);
      InorderSuccessor->right =
      deleteNode->right;
      InorderSuccessor->right->parent =
      InorderSuccessor;
    }
    Transplant(deleteNode,
    InorderSuccessor);
    InorderSuccessor->left =
    deleteNode->left;
    InorderSuccessor->left->parent =
    InorderSuccessor;
    delete (deleteNode);
  }
  Splay(parent);
}
```

7 Top Down Delete in Splay Tree

```
void TopDownDelete(T key)
{
  if (root == nullptr)
  {
    return;
  }
  Node<T> *deleteNode = Search(key);
  if (deleteNode->data != key)
  {
    Splay(deleteNode);
    return;
  }
  if (root->left == nullptr)
  {
    root = root->right;
  }
  else
  {
    Node<T> *leftMax =
```

```
    Predecessor(root->left);
    Node<T> *temp = root;
    Node<T> *root = root->left;
    Splay(leftMax);
    root->right = temp->right;
    delete (temp);
  }
  return root;
}
```

8 Transplant

```
void Transplant(Node<T> *u, Node<T> *v)
{
  if (u->parent == nullptr)
  {
    root = v;
  }
  else if (u->parent->left == nullptr)
  {
    u->parent->left = v;
  }
  else
    u->parent->right = v;
  if (v != nullptr)
  {
    v->parent = u->parent;
  }
}
```

9 Inorder Successor

```
Node<T> *Successor(Node<T> *p)
{
  while (p && p->left != nullptr)
  {
    p = p->left;
  }
  return p;
}
```

10 Inorder Predecessor

```
Node<T> *Predecessor(Node<T> *p)
{
  while (p && p->right != nullptr)
  {
    p = p->right;
  }
  return p;
}
```

11 Splaying in Splay Tree

```
void Splay(Node<T> node)
{
  while (node->parent != nullptr)
  {
    if (node->parent == root)
    {
      if (node == node->parent->left)
      {
        RightRotation(node->parent);
      }
      else
      {
        LeftRotation(node->parent);
      }
    }
    else
    {
      Node<T> *parent = node->parent;
      Node<T> *grandparent = parent->parent;
      if (node == node->parent
      && parent == parent->left)
      {
        RightRotation(grandparent);
        RightRotation(parent);
      }
      else if (node == node->parent->right
      && parent == parent->parent->right)
      {
        LeftRotation(grandparent);
        LeftRotation(parent);
      }
      else if (node == node->parent->left
      && parent == parent->parent->right)
      {
        RightRotation(parent);
        LeftRotation(grandparent);
      }
      else
      {
        LeftRotation(parent);
        RightRotation(grandparent);
      }
    }
  }
}
```

12 Left Rotation

```
Node<T> *LeftRotation(Node<T> node)
{
  Node<T> *rightChild = node->right;
  node->right = rightChild->left;
  rightChild->left = node;
  return rightChild;
}
```

13 Right Rotation

```
Node<T> *RightRotation(Node<T> node)
{
  Node<T> *leftChild = node->left;
  node->left = leftChild->right;
  leftChild->right = node;
  return leftChild;
}
```

VII Complexity Analysis

1 Space Complexity

The **space complexity** of building a splay tree is O(n) where n represents the number of elements of the tree. The space can although be more efficiently used provided we are using a pointer-based approach as shown above.

2 Time Complexity

The **time complexity** of a splay tree is $O(\log_2 n)$ by amortized analysis. The worst-case complexity can be $\theta(\mathbf{n})$. When the tree becomes degenerate and we are searching for a leaf node. This mainly occurs when we are searching for the less frequently searched values.

VIII Usecases

The splay tree may be required as the one tree to rule them all when it comes to problems that involve multiple queries such as RMQ(Range Minimum Queries), and essentially we have been previously introduced to similar concepts such as Segment Trees. But although this may be regarded by some experts as the swiss knife of solving these types of problems a general caution may be that the use of a splay tree is mostly just applicable for those with more lax test cases as strict problems have a high-risk of TLE(Time Limited Exceeded) when it comes to using splay trees. A sample problem that involved a splay tree can be shown here:

SPOJ AdaList Here we basically have a problem of multiple insertions, deletions, and searching near the same localized area of the tree. So we basically end up having frequently accessed numbers being accessed which introduces the idea of splay tree being applicable here. To solve this problem first implement a splay tree that has two added features:

- Nodes contain 2 values instead of 1 value, the 2 values being the value provided in the initial array of numbers and the index of the value.
- The deletion function used should be a form of top-down deletion
- An extra function should be there to update the position of the right subtree by incrementing by 1 upon insertion of the number
- Another extra function should be used during deletion to update the position of the right subtree by decrementing by 1 upon deletion of a number.

This works essentially because repetitive numbers being searched causes these repetitive numbers to be grouped together and this thus helps to search there numbers faster.

IX References

- Splay Tree Code View on Github
- Zhtluo's Explanation of using Splay Tree in Competitive Programming Zhtluo Blog
- Video Tutorial by Jenny's IT CS Lectures Youtube Jenny's Lectures CS IT in Youtube
- Proof For Amortized Analysis of Splay Tree cornel cs3110

[4]ammortized analysis aquired from Cornell CS at https://www.cs.cornell.edu/courses/cs3110

[4]Usecases acquired from Zhtluo at https://zhtluo.com/cp/splay-tree-one-tree-to-rule-them-all.html