Generic Splay Trees Using C++

Mrityunjay Jha - PES2201800
Tushar Dixit - PES2201800138

Abstract:

This library provides a container for a generic splay tree, similar to containers in the Standard Template Library.

Introduction:

A splay tree is a self-balancing binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log n) amortized time. It is used in many real world applications such as caches, as it follows the principle of locality. The recently accessed elements are accessible very fast. Every time, the element which is searched for becomes the root of the splay tree by following various rotations.

Splay trees have become the most widely used basic data structure invented in the last 30 years, because they're the fastest type of balanced search tree for many applications.

Implementation:

Splay Trees follow the property where the most recently accessed element becomes the root of the tree, which means if it's immediately accessed again, this can be done in O(1) time. This however means that Splay Trees need balancing after even a read only operation such as search to maintain their property.

A. Insertion

Insertion is a straightforward process. Insert the node just like any other binary search tree, i.e. find the right leaf node to insert the value at based on the values. Once inserted, the element is spalyed to the root. This process is described below.

B. Splaying and Rotations

Whenever a node is accessed a series of operations (rotations) are performed that brings a node to the root of the tree. This is known as splay operation. By performing a splay operation on the node of interest after every access, the recently accessed nodes are kept near the root and the tree remains roughly balanced, so that we achieve the desired amortized time bounds.

Let us call making a right rotation as zig, and making a left rotation as zag. Namely, there are 3 types of situations that are possible:

1) The node we are searching for is the root - No rotations have to take place as the node we are searching for is already at the root.

2) The node we are searching for is at a depth of one - In this situation, the node has only a parent and not a grandparent.

a) If the node is the left child of the root, we make a zig
rotation.
b) If the node is the right child of the root, we make a

zag rotation.

3) In all other situations, the node has both a parent (P) and
a grandparent (G). Here looking from the grandparent
down to the node, we have four combinations,
a) left-left : Zig(G) + Zig(P)
b) right-right : Zag(G) + Zag(P)
c) left-right : Zag(P) + Zig(G)
d) right-left : Zig(P) + Zag(G)

C. Deletion
To delete a node, we start by searching for it and finding it. The search process spalys the node to the root (as described above). Once the element to be deleted is at the root, we delete it. This leaves us with two unconnected trees (the root's left subtree and right subtree). We continue our process by searching for the largest element of the left subtree and splaying it to the top. This will become our new root. The final step is to attach the right subtree as the right child of this element (being the largest element in the left, it would not already have a right child).

Advantages:

Splay trees are self-optimising and frequently accessed nodes will always be nearer to the root. Worst case time complexity is O(n) and the average case is O(log(n)). This makes it particularly useful in caches and other applications.

Shortcomings:

1) The most significant disadvantage of splay trees is that the height of a splay tree can be linear (as compared to other self balanced trees like AVL Trees).

2) The representation of splay trees can change even when they are accessed in a ’read-only’ manner (i.e. by find operations). This complicates the use of such splay trees in a multi-threaded environment.

CONSTRUCTORS, ITERATORS AND PUBLIC METHODS:

Copy Constructor is supported to initialize one splay tree object with the same configuration as another .
Equality of two Splay Trees can be checked by using the ==operator. Two splay trees are said to be equal when both the trees are identical in value and position of their nodes.

Apart from this, a lot of standard methods and iterators are supported. The iterators supported are inorder, preorder and postorder traversal.
1) begin() : Return iterator to beginning (default traversal is inorder)
2) end() : Return iterator to end (default traversal is inorder)
3) rbegin(): Return reverse iterator to reverse beginning (default traversal is inorder)
4) rend(): Return reverse iterator to reverse end (default traversal is inorder)
5) begin in(): Return iterator to beginning (inorder traversal)
6) end in(): Return iterator to end (inorder traversal)
7) size(): Return size
8) empty(): Test whether splay tree is empty
9) insert(): Insert an element by value
10) erase():Test whether splay tree is empty
11) clear():Clear entire contents of splay tree

CONCLUSIONS

We have implemented a lightweight, easy to use Generic Splay Tree container which can be easily used by anyone working to write programs which deal with accessing similar elements frequently. It is similar to the containers in the STL library, and hence is easy to use. Supporting multiple iterators like inorder,preorder, postorder will make it easy for the user to traverse the splay tree.

REFERENCES

[1] Wikipedia : https://en.wikipedia.org/wiki/Splay tree
[2] GeeksForGeeks: https://www.geeksforgeeks.org/splay-tree-set-1-insert/