

Generic Implementation of Segment Trees
PROJECT ID : 8

UE18CS331 : Generic Programming

Offered By
Professor N. S. Kumar

Project By

Name	SRN	Email	Section
Tanishq Vyas	PES1201800125	tanishqvyas069@gmail.com	H
Nitin Bindal	PES1201800199	nitinbindal100@gmail.com	D

Table of Contents

Serial No.	Chapter		Page No.	
1	Introduction		3	
	1.1	What are Segment Trees ?	3	
	1.2	Generic Segment Trees	4	
		1.2.1	What are Generic Segment Trees ?	4
		1.2.2	Our Contributions	5
2	Methodology		6	
	2.1	Implementation	6	
	2.2	Generic Seg_tree Class	6	
	2.3	Generic Create Function	7	
	2.4	Generic Range Query Function	7	
	2.5	Generic Update Query Function	7	
3	Usage & Documentation		8	
	3.1	Usage	8	
		3.1.1	Importing Header File	8

		3.1.2	Creating a Segment Tree	8
		3.1.3	Displaying the Segment Tree	9
		3.1.4	Update Query	10
		3.1.5	Range Query	11
	3.2	Documentation		12
		3.2.1	Segment Tree Creation	12
		3.2.2	Display	12
		3.2.3	Performing Range Query	13
		3.2.4	Performing Update Query	13
		3.2.5	In-built Functors	14
		3.2.6	User Defined Functors	15
4	Conclusion			16
	References			16

Chapter 1 : Introduction

1.1 What are Segment Trees ?

Segment Tree is essentially a binary tree in whose nodes we store information about segments of a linear data structure such as an array.

A segment tree proves to be very useful in cases where we have a long range of numbers/data points and we need to perform numerous range based queries on the same along with certain kinds of range based updates. Using a segment tree in such cases helps reduce the time by a large margin.

Consider an array, $\mathbf{A} = [a_1, a_2, a_3, \dots, a_n]$ of length \mathbf{N} .

A possible range based query would be to get the sum of elements from index i to index j such that $0 \leq i \leq j \leq N-1$.

A possible update query would be to add 7 to all the elements from index i to index j such that $0 \leq i \leq j \leq N-1$.

In order to understand the creation of a segment tree better, let us take an example of range sum queries. In figure 1, we can see the segment tree for array $\mathbf{A} = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$, for range sum queries.

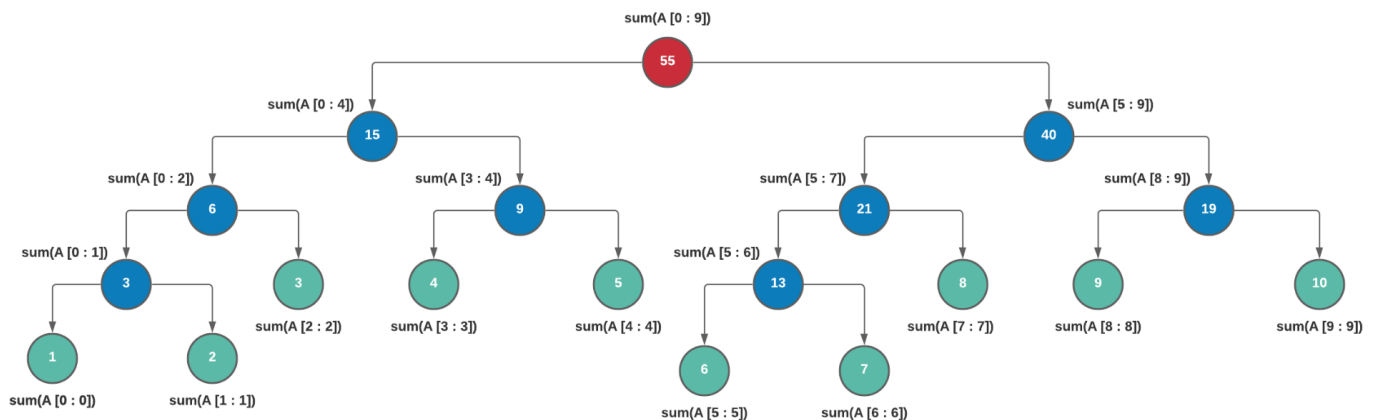


Figure 1 : Segment Tree for Range Sum

The above image clearly depicts the resulting segment tree for the example problem at hand.

- Speaking in general, each node contains the sum of elements of the array for in a given range **L** (lower bound) to **R** (upper bound). The left child of the node contains the sum of elements in the range $[L, (R+L)/2]$ whereas the right child of the node contains the sum of elements in the range $((R+L)/2, R]$.
- Thus starting from the root node, which contains the sum of the elements of the whole array and moving down towards the leaf node which contains the elements of the array itself, the segment tree can be formed by recursively splitting the segment at each node.
- Thus the size of the segment tree, assuming we have an array of size **N** can be easily calculated to be $2*N - 1$.
- Each node either has 2 children (internal nodes) or no children at all (leaf nodes).

1.2 Generic Segment Trees

This section illustrates a Generic **Segment Tree** and the implementation of the same in our project. The implementation not only allows for a generic container but also gives the user flexibility with the kind of updates they can perform using in-built functors provided by us or by writing their own custom functors. About which we have provided detailed description in the upcoming sections.

1.2.1 What are Generic Segment Trees ?

Generic Segment Trees, as the name suggests, are generic in nature where the user provides different types of iterators and a container on which a user-defined or built-in (update & query) operations need to be performed.

1.2.2 Our Contributions

In this project we worked on developing a completely generic implementation of a Segment Tree, allowing the users to fully leverage the use of Segment Tree as a container for various types of linear data structures. The following are the contributions that we have made :

- **CreateSegTree_**
Function which creates the segment tree for the provided linear container.
- **Update**
Function which performs an operation (user defined functor or an inbuilt functor provided by us) and modifies values in a given range **[L, R]**.
- **Query**
Function which is used to calculate the output of the operation for a range of values. Example : Range sum query.
- **Inbuilt-Functors**
These are some helper functors which are divided into 2 types in which one set of functors are used for creating and querying the segment trees and another type is used for updating the segment.
- **Overloaded Print**
Function which will print the entire segment tree created. This is an overloaded function for ostream object class .

Chapter 2 : Methodology

The following section illustrates the methodology used while implementing the generic segment trees.

2.1 Implementation

Implementation for generic segment tree is done keeping in mind to have code as generic as possible giving the user the freedom to do anything he/she wants in which a segment tree is required. Segment tree is stored in a linear data structure namely arrays. Inside every function the ***orthogonality of types*** are used wherever possible. Our generic implementation also supports different kinds of iterators like forward , bidirectional and random access . Besides this there are several builtin functors for basic operations (add ,multiply ,divide etc.) which user can directly use.

2.2 Generic Seg_tree Class

This class is the crux of our implementation as it will act as a type for segment tree. This is a generic class template and user will have to explicitly provide the type by writing them inside angular brackets after class name and before instance name. Once these type are provided it contains a constructor in which start ,end iterators and size of the containers needs to be provided. After that this container will allocate memory to seg_tree array which is used to store segment tree and it will also call create_seg_tree function to build the segment tree for the container provided. It contains 4 function out of which 3 are described below fourth function is just a overloaded function for printing the entire segment tree.

2.3 Generic Create Function

This function as the name suggests creates a segment tree of a given container. This function is private and is automatically called when `seg_tree` class is instantiated and it takes 3 arguments in total out of which first two are given by user which is start and end iterators of a container and third argument is always 0 which just denotes the start position of segment tree array. Here operation which needs to be performed on the segment tree is provided by the user as a functor when instantiating the `seg_tree` class.

2.4 Generic Range Query Function

This function will output the result of query range and the operation which will be used in this function will be a user-defined functor. This function expects 4 arguments the start and end of the container and the range i.e **[L,R]** on which query function needs to be executed. Here the operation which will be performed on the any two nodes of segment tree has to be same operation that is used while building the segment tree and hence the functor provided during class instantiation is used in range query function as well.

2.5 Generic Update Query Function

This function will perform an operation (via a functor which has to be explicitly provided by the user) on a range of values and modifies them. During modification of the container values the segment tree values will also be updated from leaves to root but point to note here is updation operation is performed via user-provided functor given in update function while segment tree updation is done with another functor that user has provided during the class instantiation (the one used to build segment tree) . By having this behaviour it makes our implementation much more flexible and generic as the user can perform whatever operation he wants on the container values regardless of the operation he used to build the segment tree !

Chapter 3 : Usage & Documentation

3.1 Usage

*For exhaustive documentation please refer to the section 3.2

3.1.1 Importing Header File

Download the **src** folder from [Generic Segment Tree github repository](#) and place it in your project folder (The repository as of now is private so link may show 404). Then write the following line of code in order to import the generic implementation

```
#include "../src/segTree.h"
```

3.1.2 Creating a Segment Tree

Now in order to create a segment tree **s**, make use of the following function

```
#include <iostream>
#include <vector>
#include "../src/segTree.h"

using namespace std;

int main(int argc, char const *argv[])
{
    vector<int> v = {1, 1, 1, 1, 1};

    vector<int>::iterator it;

    // Create a Segment Tree for Range Sum query
    seg_tree<vector<int>::iterator, Add<int>> s(begin(v), end(v) - 1, v.size());
}
```

3.1.3 Displaying the Segment Tree

Now that we have created, use the following line of code to display/view the created segment tree

```
#include <iostream>
#include <vector>
#include "../src/segTree.h"

using namespace std;

int main(int argc, char const *argv[])
{
    vector<int> v = {1, 1, 1, 1, 1};

    vector<int>::iterator it;

    // Create a Segment Tree for Range Sum query
    seg_tree<vector<int>::iterator, Add<int>> s(begin(v), end(v) - 1, v.size());

    // Display created Segment Tree
    cout << s << "\n";
}
```

3.1.4 Update Query

Now use the following piece of code to in order to perform the updates on a range of items

```
#include <iostream>
#include <vector>
#include "../src/segTree.h"

using namespace std;

int main(int argc, char const *argv[])
{
    vector<int> v = {1, 1, 1, 1, 1};

    vector<int>::iterator it;

    // Create a Segment Tree for Range Sum query
    seg_tree<vector<int>::iterator, Add<int>> s(begin(v), end(v) - 1, v.size());

    // Display created Segment Tree
    cout << s << "\n";

    // Constant
    int const_val1 = 6;
    int const_val2 = 4;

    // Update Query for increasing all values by 6
    s.update(begin(v), begin(v), end(v) - 1, 1, 2, 0, const_val1, addEq<int>());

    // Update query to set all values in the range as 4
    s.update(begin(v), begin(v), end(v) - 1, 3, 4, 0, const_val2, Eq<int>());
}
```

3.1.5 Range Query

Now use the following piece of code to in order to perform the Range query on a range of items

```
#include <iostream>
#include <vector>
#include "../src/segTree.h"

using namespace std;

int main(int argc, char const *argv[])
{
    vector<int> v = {1, 1, 1, 1, 1};

    vector<int>::iterator it;

    // Create a Segment Tree for Range Sum query
    seg_tree<vector<int>::iterator, Add<int>> s(begin(v), end(v) - 1, v.size());

    // Display created Segment Tree
    cout << s << "\n";

    // Constant
    int const_val1 = 6;
    int const_val2 = 4;

    // Update Query for increasing all values by 6
    s.update(begin(v), begin(v), end(v) - 1, 1, 2, 0, const_val1, addEq<int>());

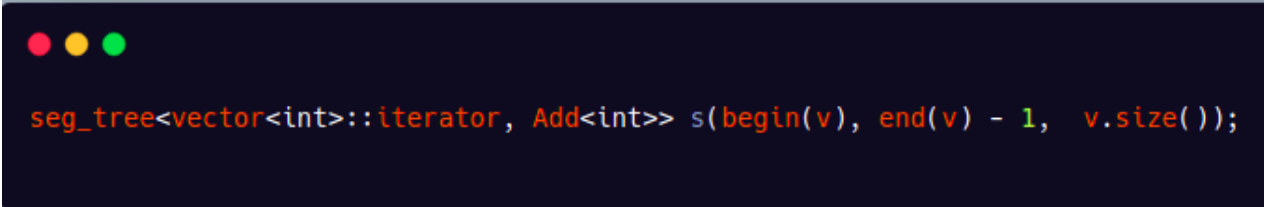
    // Update query to set all values in the range as 4
    s.update(begin(v), begin(v), end(v) - 1, 3, 4, 0, const_val2, Eq<int>());

    // Performing Range Sum Query
    auto val = s.query(v.begin(), v.begin(), end(v) - 1, 1, 4, 0);
    cout << val << "\n";
}
```

3.2 Documentation

3.2.1 Segment Tree Creation

Code snippet :

A code snippet in a dark-themed editor with three colored window control buttons (red, yellow, green) in the top-left corner. The code is:


```
seg_tree<vector<int>::iterator, Add<int>> s(begin(v), end(v) - 1, v.size());
```

```
seg_tree<vector<int>::iterator, Add<int>> s(begin(v), end(v) - 1, v.size());
```

Here in order to create a segment tree first it expects the type of container and the type of functor to be provided by the user after that the constructor for the class expects 3 arguments → start , end and size of the container. Once provided it will create a segment tree on the provided container.

3.2.2 Display

Code Snippet :

A code snippet in a dark-themed editor with three colored window control buttons (red, yellow, green) in the top-left corner. The code is:

```
|cout << s << "\n";
```

```
|cout << s << "\n";
```

Display function is an overloaded function of ostream class in which the user can print the segment tree created.

3.2.3 Performing Range Query

Code Snippet :

```
auto val = s.query(v.begin(), v.begin(), end(v) - 1, 1, 4, 0);
```

Query function requires 6 parameters: the first two are starting iterators, then ending iterator for container, then a range needs to be specified like **[L,R]** on which query needs to be executed and the node value which generally is zero.

3.2.4 Performing Update Query

Code Snippet

```
s.update(begin(v), begin(v), end(v) - 1, 1, 2, 0, 6, addEq<int>());
```

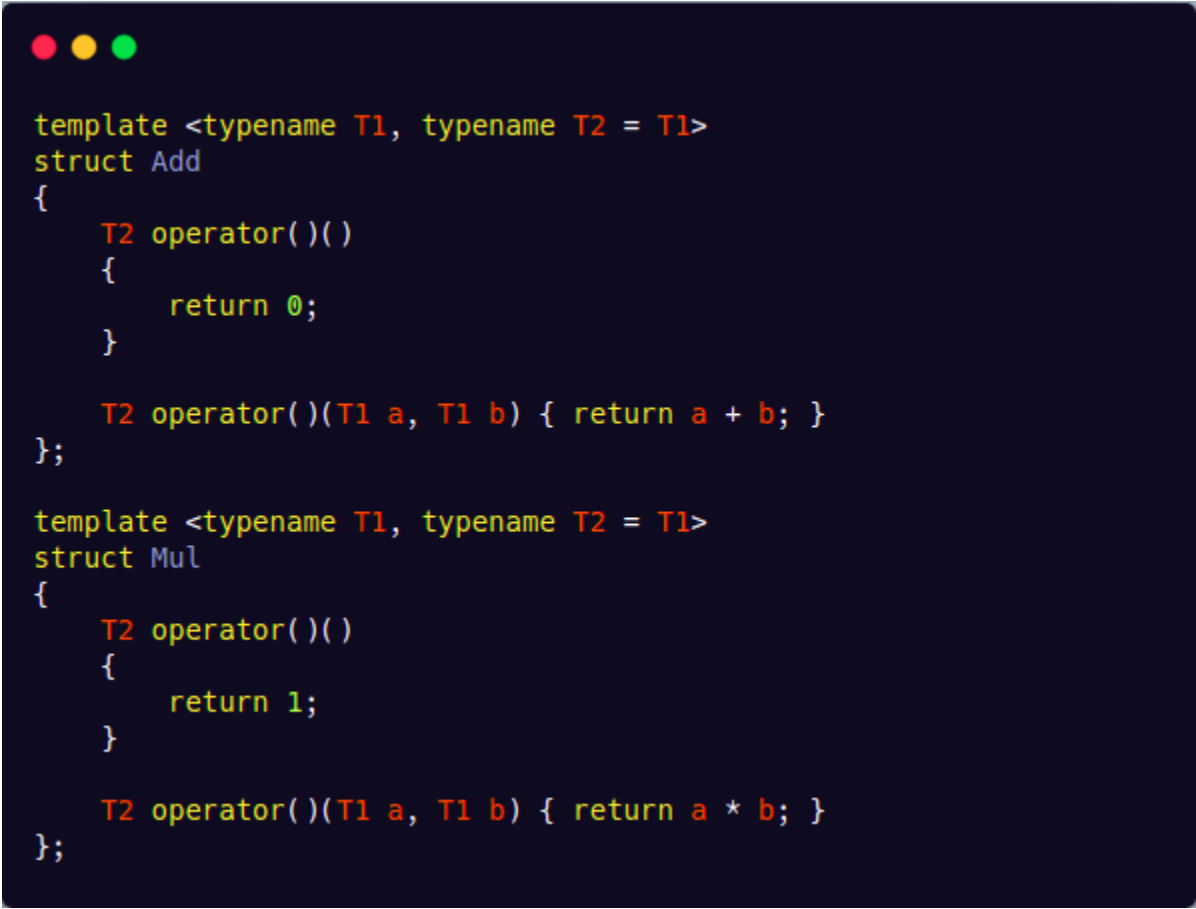
In order to perform the update on an already created segment tree one must pass the pointer to the head of the container as the first argument and second argument and end iterator as the third argument then a start and end position of the range on which updation needs to be performed as fourth and fifth argument sixth argument will be node which is the starting value of the segment trees, seventh argument is the value which is required to update the previous value and finally the update functor which contains the logic to modify the current value.

3.2.5 In-built Functors

These are the inbuilt functors for basic operators like add, multiply, divide, max, min etc. There are two categories for these functors in which the first category can be used only for creation and updation and the second category can be only used for updation.

Some examples of Inbuilt functors are

1.> Building or query functors



```

template <typename T1, typename T2 = T1>
struct Add
{
    T2 operator()()
    {
        return 0;
    }

    T2 operator()(T1 a, T1 b) { return a + b; }
};

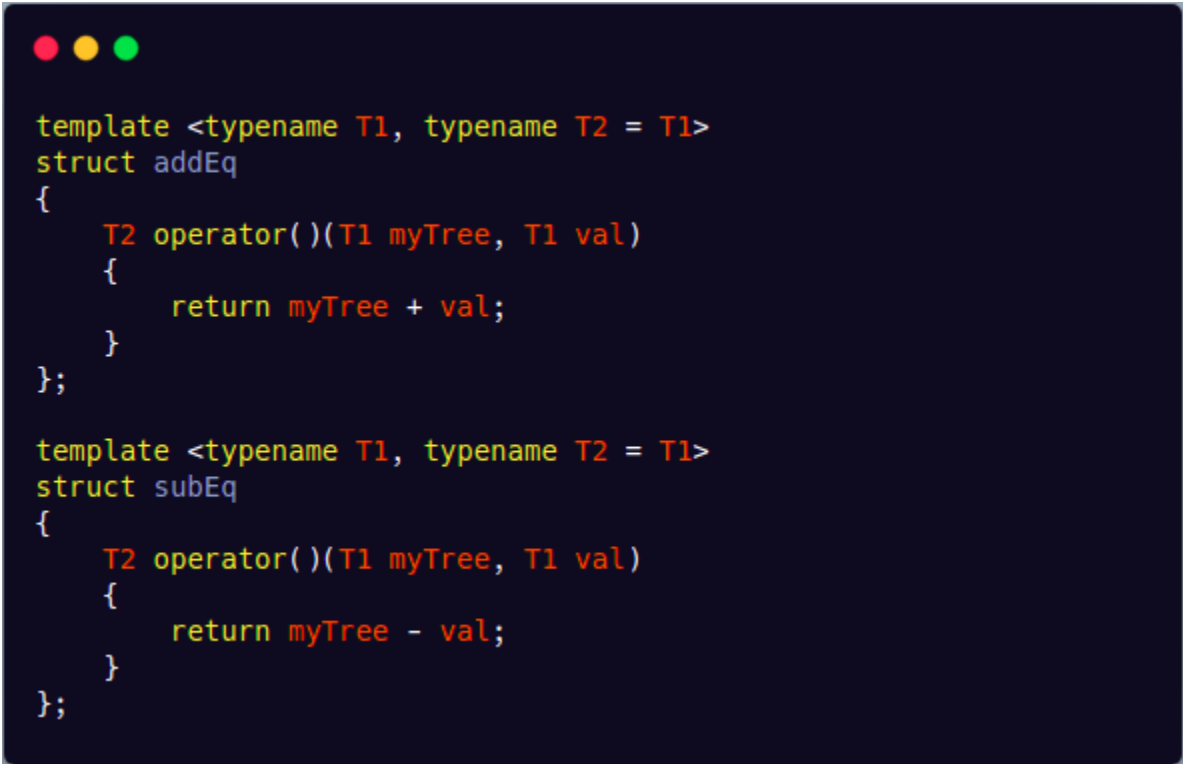
template <typename T1, typename T2 = T1>
struct Mul
{
    T2 operator()()
    {
        return 1;
    }

    T2 operator()(T1 a, T1 b) { return a * b; }
};

```

Here in the image two types of functors are shown which performs addition and multiplication.

3.2.6 User Defined Functors



```
template <typename T1, typename T2 = T1>
struct addEq
{
    T2 operator()(T1 myTree, T1 val)
    {
        return myTree + val;
    }
};

template <typename T1, typename T2 = T1>
struct subEq
{
    T2 operator()(T1 myTree, T1 val)
    {
        return myTree - val;
    }
};
```

Here the image shown provides an example of two updation functors in which first one adds **val** to the existing value and second subtracts **val** from the existing value.

Chapter 4 : Conclusion

This project helps us learn about the concept of generic programming in depth by implementing the same in order to develop a generic segment tree. The implementation is flexible, robust and allows the user the freedom to make modifications on the range in any way he wishes to. All the possible places where segment trees can be used have been taken into consideration for problems such as range sum queries, maximum queries, min queries, etc. The segment tree has been successfully built as a container that has a simple interface and can be extensively used by the user. The current implementation does not account for lazy propagation as the provision of flexibility in terms of update queries does not align well with the current design considerations.

References

- [1] Segment Tree | Set 1 (Sum of given range) .Available Online at URL: (www.geeksforgeeks.org/segment-tree-set-1-sum-of-given-range/)
- [2] Segment Trees tutorial. Available online at URL: (www.hackerearth.com/practice/data-structures/advanced-data-structures/segment-trees/tutorial/)
- [3] Get the return type of a method from a member function pointer .Available online at URL: (stackoverflow.com/questions/29184944/get-the-return-type-of-a-method-from-a-member-function-pointer)
- [4] Type Inference in C++ (auto and decltype). Available online at URL: (www.geeksforgeeks.org/type-inference-in-c-auto-and-decltype/)
- [5] Template function inside a template class. Available online at URL: (stackoverflow.com/questions/8640390/template-function-inside-template-class)
- [6] Lazy Propagation in Segment Trees. Available online at URL: (www.hackerearth.com/practice/notes/segment-tree-and-lazy-propagation/)