



STL Implementation in C

November 7, 2022

Sakshi Bansal (2021MCB1244) ,
Ashima Goyal (2021CSB1075) ,
Harshit Gupta (2021CSB1092)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Monisha Singh

Summary: The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as maps, sets, vectors, etc. In this project we are trying to implement STL like containers in C. Their implementation includes memory allocation, inserting an element, searching for an element, deletion and many such operations. Further then we have done the time analysis and tried to optimise the complexities of our functions.

1. Introduction

The Standard Template Library (STL) is a well known set of classes which offer a variety of set of functions such as sets, maps, vectors which make many structures easy to use. The STL is a powerful set of C++ template classes to provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures. A container is a holder object that stores a collection of other objects (its elements). The container manages the storage space for its elements and provides member functions to access them.

Since C does not provide us with any such library or template, implementation of these container elements of C++ in C is an interesting but also a challenging task. Each container element requires the implementation of some or the data structure. Various basic functions have been created like memory allocation, inserting an element, deletion, searching for an element, etc. All functions have been created maintaining optimised time complexities.

Many containers have several member functions in common, and share functionalities. The decision of which type of container to use for a specific need does not generally depend only on the functionality offered by the container, but also on the efficiency of some of its members (complexity).

VECTOR

Vectors are the same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes the array may need to be extended. Removing the last element takes only constant time because no resizing happens. Unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

SETS

Sets are a type of associative container in which each element has to be unique because the value of the element identifies it. The values are stored in a specific sorted order i.e. either ascending or descending. Sets are containers that store unique elements following a specific order. In a set, the value of an element also identifies it and each value must be unique. They can be inserted or removed from the container. Set implementation is based on implementing rb trees, a well recognised data structure.

MAPS

Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have the same key values. The types of key and mapped value may differ, and are grouped together. Internally unordered map is implemented using Hash Table, the key provided to map is hashed into indices of a hash table which is why the performance of data structure depends on the hash function.

2. Figures, Tables and Algorithms

2.1. Figures

The following figures demonstrate the working of containers like vector, map, set(using rb trees).

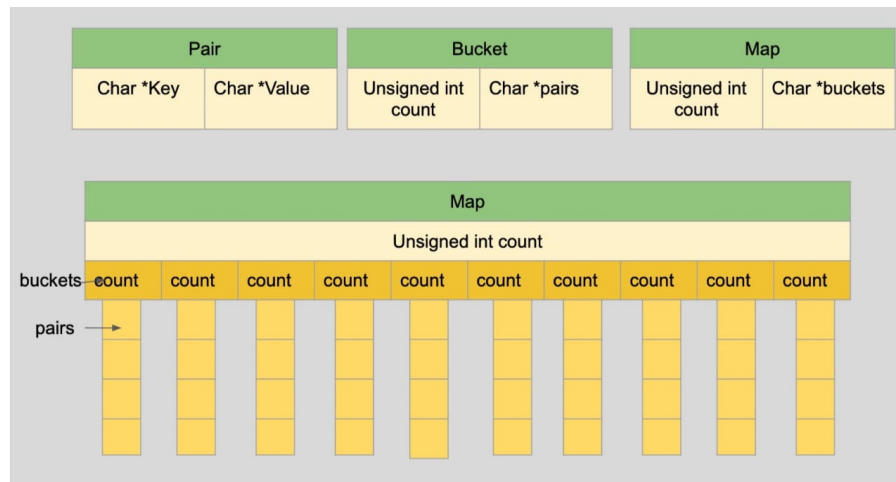


Figure 1: Demonstration of Map

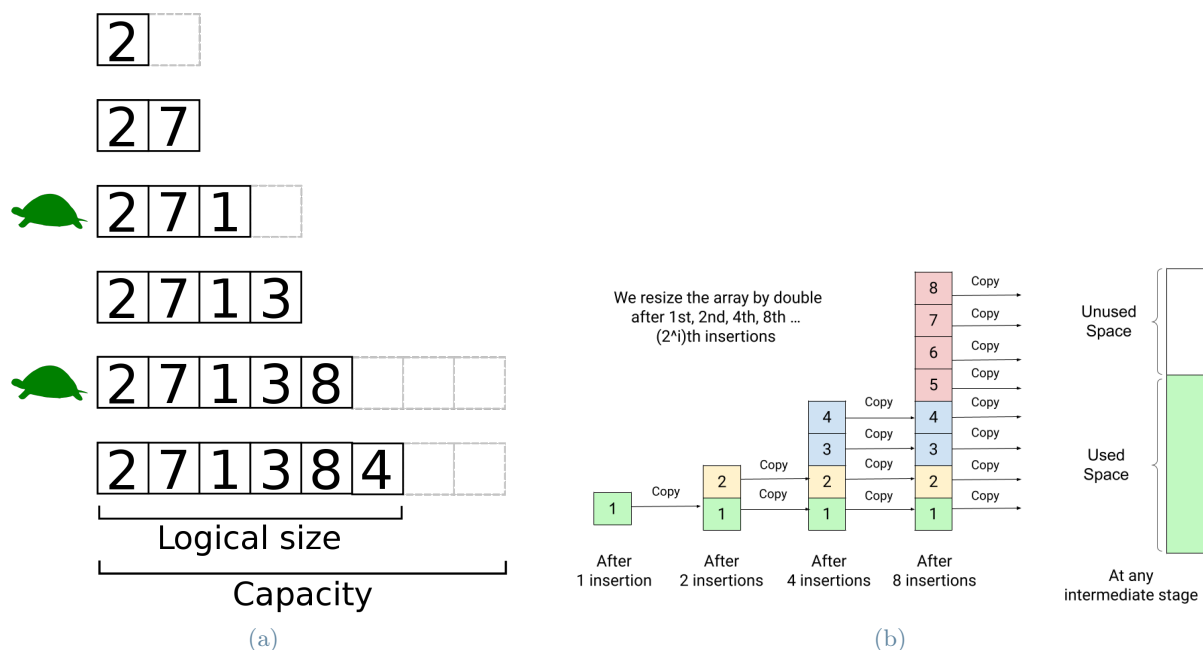


Figure 2: Resizing in vector

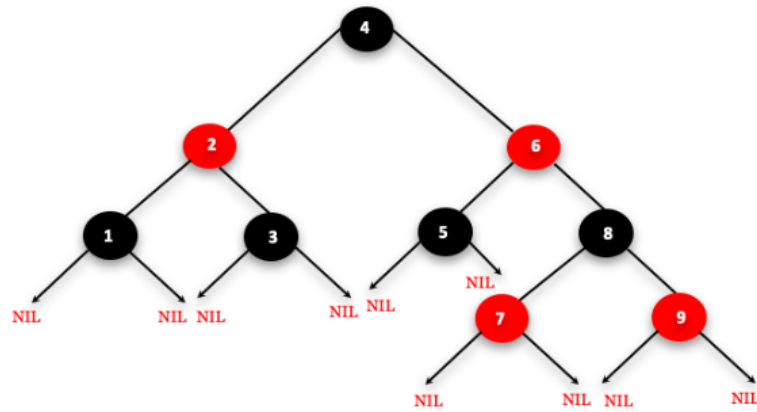


Figure 3: Demonstration of RB trees

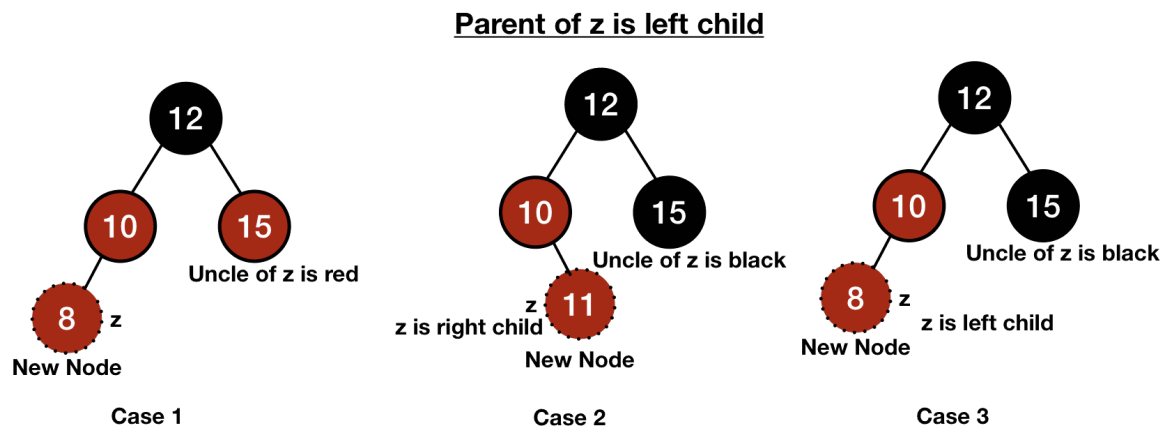


Figure 4: Insertion in RB trees

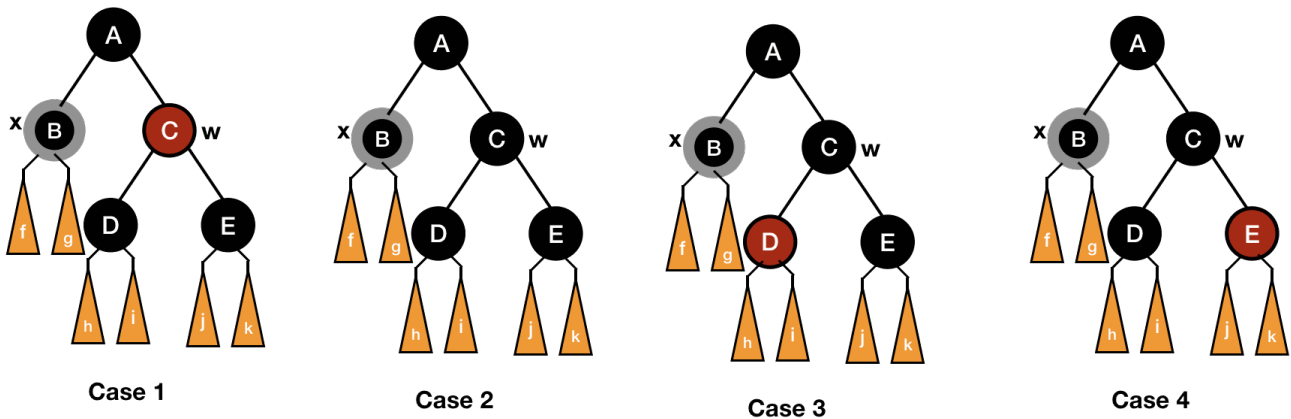


Figure 5: Deletion in RB trees

2.2. Time Complexity

Given below is the time complexity of various operation performed in vector, set and unordered map.

SET

FUNCTION	Complexity
<code>set_insert</code>	$O(\log(n))$
<code>set_erase</code>	$O(\log(n))$
<code>set_find</code>	$O(\log(n))$
<code>set_size</code>	$O(1)$
<code>set_display</code>	$O(\log(n))$
<code>set_begin</code>	$O(\log(n))$
<code>set_end</code>	$O(\log(n))$

Table 1: Time Complexity of functions of Set

VECTOR

FUNCTION	Complexity
<code>vector_push_back</code>	$O(1)$
<code>vector_pop_back</code>	$O(1)$
<code>vector_display</code>	$O(N)$
<code>vector_at</code>	$O(N)$
<code>vector_find</code>	$O(N)$
<code>vector_clear</code>	$O(1)$

Table 2: Time Complexity of functions of Vector

MAP

FUNCTION	Complexity
<code>map_insert</code>	$O(1)$
<code>map_at</code>	$O(1)$
<code>map_erase</code>	$O(1)$
<code>map_size</code>	$O(1)$
<code>map_display</code>	$O(N)$

Table 3: Time Complexity of functions of Map

2.3. Algorithms

Pseudo-algorithms of our code are as follows:

SET

Algorithm 1 new node(key)

```
1: node*n = allocate memory to node n
2: n.color = red
3: n.data = key
4: n.left = NULL
5: n.right = NULL
6: n.parent = NULL
7: return n
```

Algorithm 2 create()

```
1: set t = allocate memory to t
2: node* null=allocate memory
3: n.color=black
4: n.left=NULL
5: n.right=NULL
6: n.parent=NULL
7: t.Null=NULL
8: t.root=t.Null
9: return t
```

Algorithm 3 LEFT ROTATE(T,X)

```
1: ptr = x.right
2: if  $x.right.left \neq NULL$ 
3:   x.right.left.parent=x
4: if  $x.parent = NULL$ 
5:   t.root=x.right
6: else if  $x = s.parent.left$ 
7: x.parent.right=x.right
8: else
9:   x.parent.right=x.right
10: x.right.parent=x.right
11: x.parent=x.right
12: x.right=x.right.left
13: ptr.left=x
```

Algorithm 4 RIGHT ROTATE(T,X)

```
1: ptr = x.left
2: if  $x.left.right \neq NULL$ 
3:   x.left.right.parent=x
4: if  $x.parent = NULL$ 
5:   t.root=x.left
6: else if  $x = s.parent.right$ 
7: x.parent.left=x.left
8: else
9:   x.parent.left=x.left
10: x.right.left=x.left
11: x.parent=x.left
12: x.left=x.left.right
13: ptr.right=x
```

Algorithm 5 INSERT FIXUP(T,x)

```
1: while x.parent.color = red
2:   if both x.parent.color and x.uncle.color is red
3:     color x's grandparent red
4:     color x's parent and uncle black
5:   else
6:     if x's parent is the left child
7:       if x is the right child
8:         x=x.p
9:         LEFT ROTATE(T,X)
10:      color x's parent black
11:      color x's grandparent red
12:      RIGHT ROTATE(T,x's grandparent)
13:     else
14:       if x is left child
15:         x=x.p
16:         RIGHT ROTATE(T,x)
17:       color x's parent black
18:       color x's grandparent red
19:       LEFT ROTATE(T,x's grandparent)
20: t.root.color=black
```

Algorithm 6 INSERT(T,n)

```
1: create(z)
2: ptr=t->Null
3: ptr=t->root
4: while ptr!=t->NULL
5:   y=ptr
6:   if z.data < ptr.data
7:     ptr=ptr.left
8:   else if z.data = ptr.data
9:     return
10:  else z.data > ptr.data
11:    ptr=ptr.right
12: z->p=y
13: if y=t.Null
14:   make z root
15: else if z.data<y.data
16:   y->l=z
17: else
18:   y->r=z
19: z.r=t.null
20: z.l=t.null
21: increment size
22: INSERT FIXUP(T,z)
```

Algorithm 7 rb TRANSPLANT(T,A,B)

```
1: if a is the root
2:   t.root=b
3: else if a is the right child
4:   a.parent.right=b
5: else
6:   a.parent.left=b
7: b.parent=a.parent
```

Algorithm 8 Successor(T,x)

```
1: while x.left!=NULL
2:   x=x.left
3: return x
```

Algorithm 9 swap c(x,y)

```
1: int k=x.color
2: x.color=y.color
3: y.color = k
```

Algorithm 10 find(t,data)

```
1: node * ptr=t.root
2: while ptr!=t.Null
3:   if ptr.data=data
4:     return ptr
5:   else if ptr.data<data
6:     ptr=ptr.right
7:   else
8:     if ptr=ptr.Null
9:       print not found
10: return ptr
```

Algorithm 11 erase(T, n)

```
1: node  $z = \text{find } n \text{ in } t$ 
2: node  $y = z$ 
3: node  $x$ 
4:  $y.\text{original.color} = y.\text{color}$ 
5: if  $z.\text{left} = t.NULL$ 
6:    $x = z.\text{right}$ 
7:   rb TRANSPLANT( $T, z, z.\text{right}$ )
8: else if  $z.\text{right} = t.NULL$ 
9:    $x = z.\text{left}$ 
10:  rb TRANSPLANT( $T, z, z.\text{left}$ )
11: else
12:    $y = \text{Successor}(T, z.\text{right})$ 
13:    $y.\text{original.color} = y.\text{color}$ 
14:    $x = y.\text{right}$ 
15:   if  $y.\text{parent} = z$ 
16:      $x.\text{parent} = z$ 
17:   else
18:     rb TRANSPLANT( $T, y, y.\text{right}$ )
19:      $y.\text{right} = z.\text{right}$ 
20:      $y.\text{right.parent} = y$ 
21:     rb TRANSPLANT( $T, z, y$ )
22:      $y.\text{left} = z.\text{left}$ 
23:      $y.\text{left.parent} = y$ 
24:      $y.\text{color} = z.\text{color}$ 
25: if  $y.\text{original.color}$  is black
26:   DELETE FIXUP( $T, x$ )
27: decrement size
```

Algorithm 12 delete fixup(T,x)

```
1: while x is not root and x's color is double black
2:   if x is the left child
3:     node sibling = x.parent.right
4:     if sibling color is red
5:       swap c(color of x.parent and sibling)
6:       LEFT ROTATE(T,x.parent)
7:       sibling = x.parent.right
8:     if color of both sibling child is black
9:       color sibling red
10:      x=x.parent
11:    else
12:      if color of sibling right child is black
13:        color sibling left child black
14:        color sibling red
15:        RIGHT ROTATE(T,x.parent)
16:        sibling=x.parent.right
17:        color sibling right child black
18:        LEFT ROTATE(T,x.parent)
19:        sibling.color = x.parent.color
20:        color x's parent black
21:        x=t.root
22:    else
23:      node sibling = x.parent.left
24:      if sibling color is red
25:        swap c(color of x.parent and sibling)
26:        RIGHT ROTATE(T,x.parent)
27:        sibling = x.parent.left
28:      if color of both sibling child is black
29:        color sibling red
30:        x=x.parent
31:      else
32:        if color of sibling left child is black
33:          color sibling right child black
34:          color sibling red
35:          LEFT ROTATE(T,x.parent)
36:          sibling=x.parent.left
37:          color sibling left child black
38:          RIGHT ROTATE(T,x.parent)
39:          sibling.color = x.parent.color
40:          color x's parent black
41:          x=t.root
42:    color x black
```

Algorithm 13 begin(T)

```
1: x=Successor(t,t.root)
2: return x
```

Algorithm 14 end(*T*)

```
1: x=t.root
2: while x.right!=t.Null
3:   x=x.right
4:   return x
```

Algorithm 15 display(*T*,*n*)

```
1: if n!=t.Null
2:   display(T,n.left)
3:   print n.data
4:   display(T,n.right)
```

MAP

Algorithm 16 struct map * map_new(size)

```
1: map=(struct map*)malloc(sizeof(struct map))
2: map->count=size
3: map->buckets=(struct bucket*)malloc(map->count * sizeof(struct bucket))
4: memset(map->buckets, 0, map->count * sizeof(struct bucket))
5: return map
```

Algorithm 17 map_insert(map,key,value)

```
1: index=hash(key)%map->count
2: bucket=&(map->buckets[index])
3: pair=get_pair(bucket, key)
4: if (pair!=NULL)
5:   if (length(pair->value)<length(value))
6:     pair->value=(char *)realloc(pair->value, (strlen(value)+1)*sizeof(char))
7:   pair->value=value
8: if (bucket->count==0)
9:   bucket->pairs=(struct pair*)malloc(sizeof(struct pair))
10:  bucket->count=1
11: else
12:  bucket->pair=(struct pair*)realloc(bucket->pairs, (bucket->count+1) *sizeof(struct pair))
13:  bucket->count++
14: pair=&(bucket->pairs[bucket->count-1])
15: pair->key=key
16: pair->value=value
```

Algorithm 18 char * map_at(map,key)

```
1: index=hash(key)%map->count
2: bucket=&(map->buckets[index])
3: pair=get_pair(bucket, key)
4: if (pair==NULL)
5:   return NULL
6: return pair->value
```

Algorithm 19 map_erase(map, key, value)

```
1: index=hash(key)%map->count
2: bucket=&(map->buckets[index])
3: pair=get_pair(bucket, key)
4: if(pair==NULL)
5:     return NULL
6: if(pair->value==value)
7:     temp=&(bucket->pairs[bucket->count-1])
8:     if(temp!=pair)
9:         pair->key=temp->key
10:        pair->value=temp->value
11:    bucket->pairs=(struct pair *)realloc(bucket->pairs,(bucket->count-1)*sizeof(struct pair))
12:    bucket->count--
```

Algorithm 20 map_size(map)

```
1: size=0
2: for(i=0 to map->count)
3:     size+=(map->buckets[i]).count
4: return size
```

Algorithm 21 map_display(map)

```
1: size=0
2: for(i=0 to map->count)
3:     size=(map->buckets[i]).count
4:     bucket=&(map->buckets[i])
5:     pair=(bucket->pairs)
6:     for(j=0 to size)
7:         print(pair->key, pair->value)
```

Algorithm 22 hash(str)

```
1: hash=1
2: for(i=0 to length(str))
3:     c=*str
4:     hash = (hash*2) + c
5:     str++
6: return hash
```

Algorithm 23 get_pair(bucket, key)

```
1: if(bucket->count==0)
2:     return NULL
3: pair=bucket->pairs
4: for(i=0 to bucket->count)
5:     if(pair->key!=NULL&&pair->value!=NULL)
6:         if(pair->key==key)
7:             return pair
8:     pair++
9: return NULL
```

VECTOR

Algorithm 24 `vector_create(size)`

```
1: Create vector node
2: Alloc memory
3: vec->capacity=size
4: vec->size=size
5: return vec
```

Algorithm 25 `vector_push_back(vector,n)`

```
1: if(vector->size==vector->capacity)
2:   (realloc 2*size*sizeof(int))
3:   (vector->capacity*=2)
4: vector->vector[vector->size]=n
5: vector->size++
```

Algorithm 26 `vector_pop_back(vector,n)`

```
1: vector->size--
```

Algorithm 27 `vector_at(vector,index)`

```
1: v=vector->vector
2: if(index>=vector->size)
3:   (return NULL)
4: return &v[index]
```

Algorithm 28 `vector_find(vector,n)`

```
1: ind=-1
2: v=vector->vector
3: for(i=0 to vector->size)
4:   (if v[i]==n)
5:     (ind=i)
6:     (break)
7: return ind
```

Algorithm 29 `vector_clear(vector)`

```
1: vector->size=0
```

3. Conclusions

We have implemented C++ STL containers like structures like unordered map, sets and vectors in C. Their time complexities are as per the actual algorithms in the STL library.

4. Bibliography and citations

The following reference links were used:

<https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/>
<https://medium.com/@info.gildacademy/an-introduction-to-red-black-tree-2a13407abc6c>
<https://www.geeksforgeeks.org/c-program-red-black-tree-insertion/>

https://en.wikipedia.org/wiki/Dynamic_array

Introduction to Algorithms is a book on computer programming by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

Acknowledgements

We acknowledge our professor Dr. Anil Shukla and our mentor Ms. Monisha Singh to support us in the project and cleared our doubts at every stage.