STL Implementation in C

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

November 7, 2022

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 ehich 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).

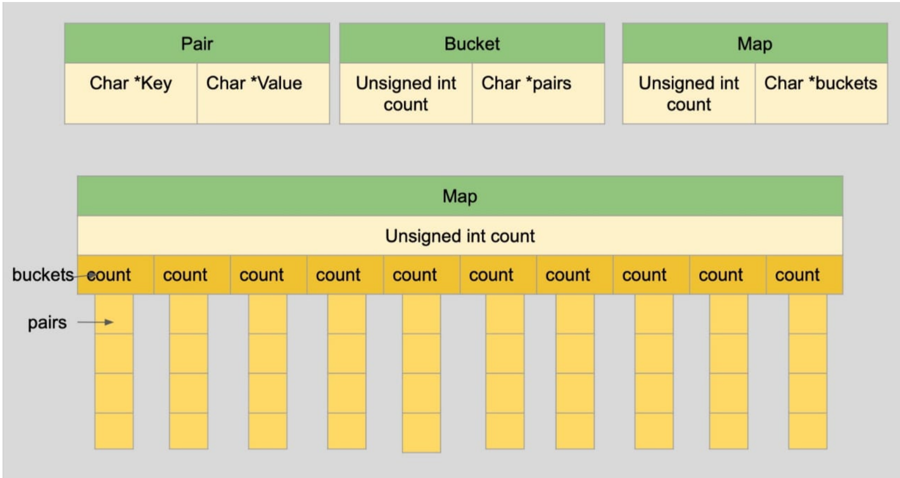
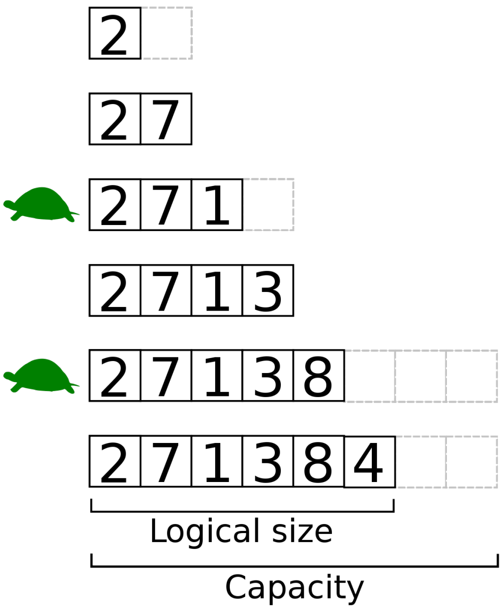
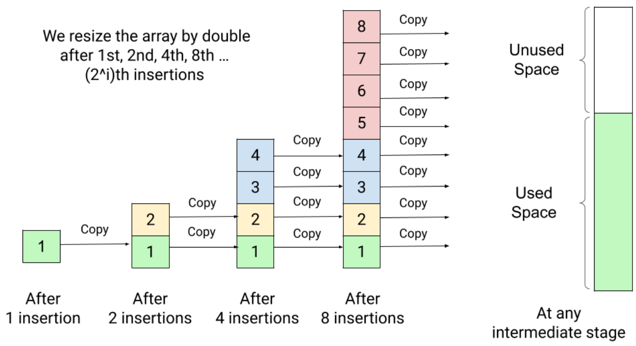


Figure 1: Demonstration of Map



(a)



(b)

Figure 2: Resizing in vector

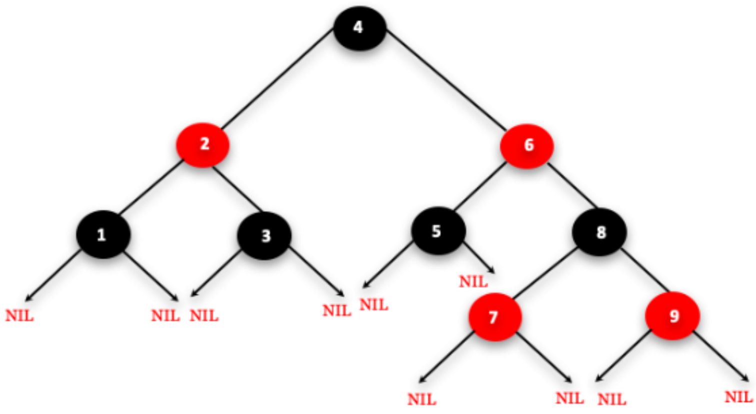


Figure 3: Demonstration of RB trees

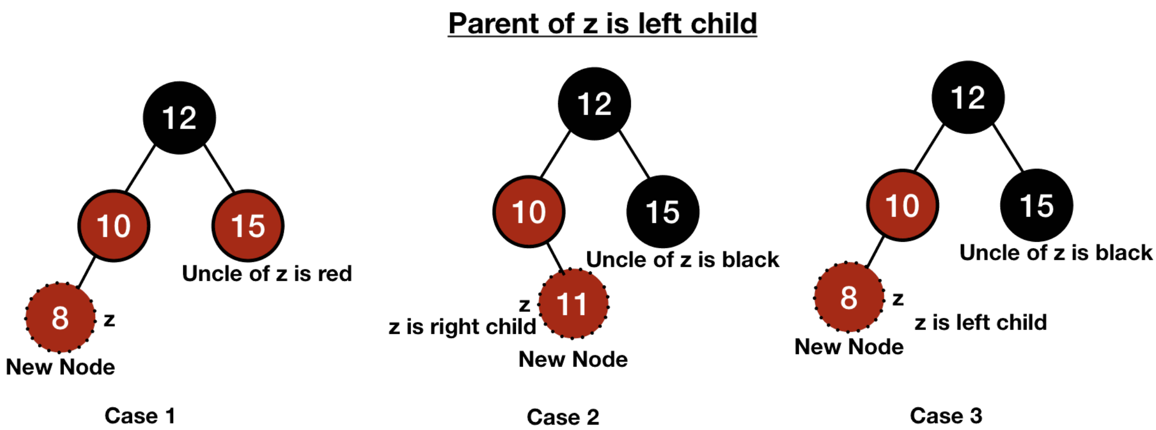


Figure 4: Insertion in RB trees

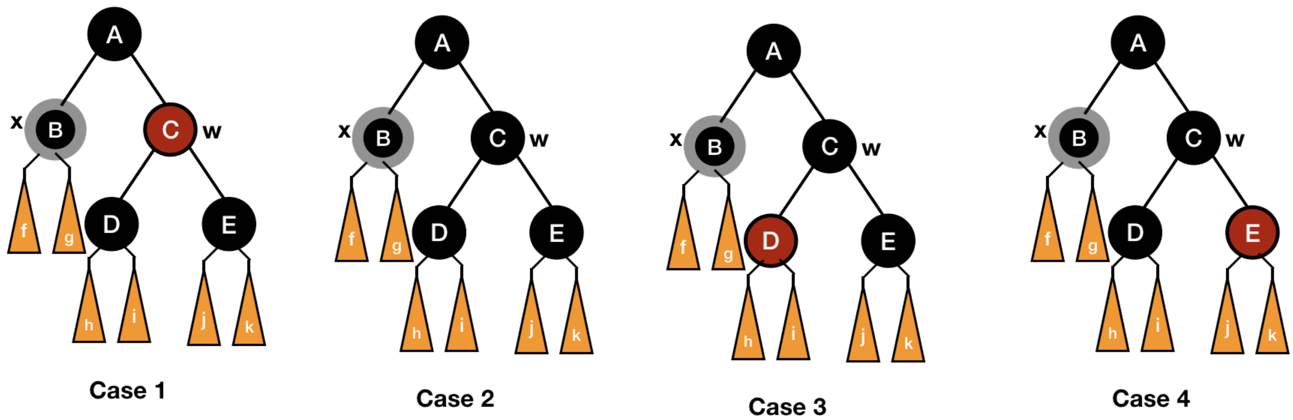


Figure 5: Deletion in RB trees

2.2. Time Complexity

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

SET

FUNCTION	Complexity
set_insert	O(log(n)
set_erase	O(log(n))
set_find	O(log(n))
set_size	O(1)
set_display	O(log(n))
set_begin	O(log(n))
set_end	O(log(n))

Table 1: Time Complexity of functions of Set

VECTOR

FUNCTION	Complexity
vector_push_back	O(1)
vector_pop_back	O(1)
vector_display	O(N)
vector_at	O(N)
vector_find	O(N)
vector_clear	O(1)

Table 2: Time Complexity of functions of Vector

MAP

FUNCTION	Complexity
map_insert	O(1)
map_at	O(1)
map_erase	O(1)
map_size	O(1)
map_display	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)

```
node*n = allocate memory to node n
n.color = red
n.data = key
n.left = NULL
n.right = NULL
n.parent = NULL
return n
```

Algorithm 2 create()

```
set t = allocate memory to t
node* null=allocate memory
n.color=black
n.left-NULL
n.right=NULL
n.parent=NULL
t.Null=NULL
t.root=t.Null
return t
```

Algorithm 3 LEFT ROTATE(T,X)

```
ptr = x.right
if x.right.left! = NULL
    x.right.left.parent=x
if x.parent = NULL
    t.root=x.right
else if x = s.parent.left
x.parent.right=x.right
else
    x.parent.right=x.right
x.right.parent=x.right
x.parent=x.right
x.right=x.right.left
ptr.left=x
```

Algorithm 4 RIGHT ROTATE(T,X)

```
ptr = x.left
if x.left.right! = NULL
    x.left.right.parent=x
if x.parent = NULL
    t.root=x.left
else if x = s.parent.right
x.parent.left=x.left
else
    x.parent.left=x.left
x.right.left=x.left
x.parent=x.left
x.left=x.left.right
ptr.right=x
```

Algorithm 5 INSERT FIXUP(T,x)

```
while x.parent.color = red
    if both x.parent.color and x.uncle.color is red
        color x's grandparent red
        color x's parent and uncle black
    else
        if x's parent is the left child
            if x is the right child
                x=x.p
                LEFT ROTATE(T,X)
            color x's parent black
            color x's grandparent red
            RIGHT ROTATE(T,x's grandparent)
        else
            if x is left child
                x=x.p
                RIGHT ROTATE(T,x)
            color x's parent black
            color x's grandparent red
            LEFT ROTATE(T,x's grandparent)
t.root.color=black
```

Algorithm 6 INSERT(T,n)

```
create(z)
ptr=t->Null
ptr=t->root
whileptr!=t->NULL
    y=ptr
    if z.data < ptr.data
        ptr=ptr.left
    else if z.data = ptr.data
        return
    else z.data > ptr.data
        ptr=ptr.right
z->p=y
if y=t.Null
    make z root
else if z.data<y.data
    y->l=z
else
    y->r=z
z.r=t.null
z.l=t.null
increment size
INSERT FIXUP(T,z)
```

Algorithm 7 rb TRANSPLANT(T,A,B)

```
if a is the root
    t.root=b
else if a is the right child
    a.parent.right=b
else
    a.parent.left=b
b.parent=a.parent
```

Algorithm 8 Successor(T,x)

```
while x.left!=NULL
    x=x.left
return x
```

Algorithm 9 swap c(x,y)

```
int k=x.color
x.color=y.color
y.color = k
```

Algorithm 10 find(t,data)

```
node * ptr=t.root
while ptr!=t.Null
    if ptr.data=data
        return ptr
    else if ptr.data<data
        ptr=ptr.right
    else
if ptr=ptr.Null
    print not found
return ptr
```

Algorithm 11 erase(T,n)

```
node z= find n in t
node y=z
node x
y.original.color=y.color
if z.left=t.NULL
    x=z.right
    rb TRANSPLANT(T,z,z.right
else if z.right=t.NULL
    x=z.left
    rb TRANSPLANT(T,z,z.left
else
    y=Successor(T,z.right)
    y.original.color=y.color
    x=y.right
    if y.parent=z
        x.parent=z
    else
        rb TRANSPLANT(T,y,y.right)
        y.right=z.right
        y.right.parent=y
    rb TRANSPLANT(T,z,y)
    y.left=z.left
    y.left.parent=y
    y.color=z.color
if y.original.color is black
    DELETE FIXUP(T,x)
decrement size
```

Algorithm 12 delete fixup(T,x)

```
while x is not root and x's color is double black
    if x is the left child
        node sibling = x.parent.right
            if sibling color is red
                swap c(color of x.parent and sibling)
                LEFT ROTATE(T,x.parent)
                sibling = x.parent.right
            if color of both sibling child is black
                color sibling red
                x=x.parent
            else
                if color of sibling right child is black
                    color sibling left child black
                    color sibling red
                    RIGHT ROTATE(T,x.parent)
                    sibling=x.parent.right
                color sibling right child black
                LEFT ROTATE(T,x.parent)
                sibling.color = x.parent.color
                color x's parent black
                x=t.root
    else
        node sibling = x.parent.left
            if sibling color is red
                swap c(color of x.parent and sibling)
                RIGHT ROTATE(T,x.parent)
                sibling = x.parent.left
            if color of both sibling child is black
                color sibling red
                x=x.parent
            else
                if color of sibling left child is black
                    color sibling right child black
                    color sibling red
                    LEFT ROTATE(T,x.parent)
                    sibling=x.parent.left
                color sibling left child black
                RIGHT ROTATE(T,x.parent)
                sibling.color = x.parent.color
                color x's parent black
                x=t.root
color x black
```

Algorithm 13 begin(T)

```
x=Successor(t,t.root)
return x
```

Algorithm 14 end(T)

```
x=t.root
while x.right!=t.Null
    x=x.right
    return x
```

Algorithm 15 display(T,n)

```
if n!=t.Null
    display(T,n.left)
    print n.data
    display(T,n.right)
```

MAP

Algorithm 16 struct map * map_new(size)

```
map=(struct map*)malloc(sizeof(struct map))
map->count=size
map->buckets=(struct bucket*)malloc(map->count * sizeof(struct bucket))
memset(map->buckets, 0, map->count * sizeof(struct bucket))
return map
```

Algorithm 17 map_insert(map,key,value)

```
index=hash(key)%map->count
bucket=&(map->buckets[index])
pair=get_pair(bucket, key)
if(pair!=NULL)
    if(length(pair->value)<length(value))
            pair->value=(char *)realloc(pair->value, (strlen(value)+1)*sizeof(char))
    pair->value=value
if(bucket->count==0)
    bucket->pairs=(struct pair*)malloc(sizeof(struct pair))
    bucket->count=1
else
    bucket->pair=(struct pair*)realloc(bucket->pairs, (bucket->count+1) *sizeof(struct pair))
    bucket->count++
pair=&(bucket->pairs[bucket->count-1])
pair->key=key
pair->value=value
```

Algorithm 18 char * map_at(map,key)

```
index=hash(key)%map->count
bucket=&(map->buckets[index])
pair=get_pair(bucket, key)
if(pair==NULL)
    return NULL
return pair->value
```

Algorithm 19 map_erase(map,key,value)

```
index=hash(key)%map->count
bucket=&(map->buckets[index])
pair=get_pair(bucket, key)
if(pair==NULL)
    return NULL
if(pair->value==value)
    temp=&(bucket->pairs[bucket->count-1])
    if(temp!=pair)
        pair->key=temp->key
        pair->value=temp->value
    bucket->pairs=(struct pair *)realloc(bucket->pairs,(bucket->count-1)*sizeof(struct pair))
    bucket->count- -
```

Algorithm 20 map_size(map)

```
size=0
for(i=0 to map->count)
    size+=(map->buckets[i]).count
return size
```

Algorithm 21 map_display(map)

```
size=0
for(i=0 to map->count)
    size=(map->buckets[i]).count
    bucket=&(map->buckets[i])
    pair=(bucket->pairs)
    for(j=0 to size)
        print(pair->key,pair->value)
```

Algorithm 22 hash(str)

```
hash=1
for(i=0 to length(str))
    c=*str
    hash = (hash*2) + c
    str++
return hash
```

Algorithm 23 get_pair(bucket,key)

```
ifbucket->count==0
    return NULL
pair=bucket->pairs
for(i=0 to bucket->count)
    if(pair->key!=NULL&&pair->value!=NULL)
        if(pair->key==key)
            return pair
        pair++
return NULL
```

VECTOR

Algorithm 24 vector_create(size)

```
Create vector node
Alloc memory
vec->capacity=size
vec->size=size
return vec
```

Algorithm 25 vector_push_back(vector,n)

```
if(vector->size==vector->capacity)
    (realloc 2*size*sizeof(int))
    (vector->capacity*=2)
vector->vector[vector->size]=n
vector->size++
```

Algorithm 26 vector_pop_back(vector,n)

```
vector->size--
```

Algorithm 27 vector_at(vector,index)

```
v=vector->vector
if(index>=vector->size)
    (return NULL)
return &v[index]
```

Algorithm 28 vector_find(vector,n)

```
ind=-1
v=vector->vector
for(i=0 to vector->size)
    (if v[i]==n)
            (ind=i)
            (break)
return ind
```

Algorithm 29 vector_clear(vector)

```
vector->size=0
```

Algorithm 30 merge(vector,left,mid,right)

```
l_size=mid-left+1
r_size=right-mid
int l_arr[l_size+1]
int r_arr[r_size+1]
for(i=0 to l_size)
    l_arr[i]=v[i+l]
for(i=0 to r_size)
    r_arr[i]=v[mid+1+i]
l_arr[l_size]=INT_MAX]
r_arr[r_size]=INT_MAX;
for(i=0 to r)
    if(l_arr[l_i]<=r_arr[r_i])
            v[i]=l_arr[l_i]
            l_i++
    else
            v[i]=r_arr[r_i]
            r_i++
```

Algorithm 31 merge_sort(vector,left,right)

```
if(left>=right)
    return
mid=(left+right)/2
merge_sort(vector,left,mid)
merge_sort(vector,mid+1,right)
merge(vector,left,mid,right)
```

Algorithm 32 vector_sort(vector)

```
merge_sort(vector->vector, 0, vector->size-1)
```

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.