Task 1

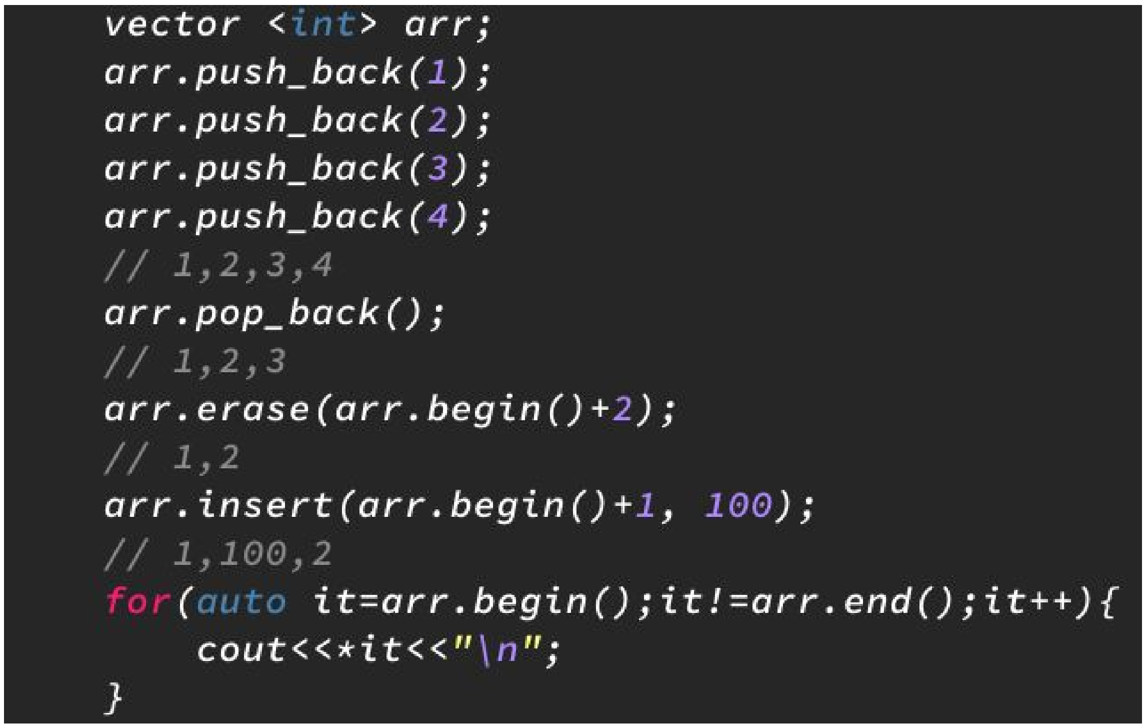
Vector is dynamic array.It can be resized unlike static arrays.
Methods of vector are

Methods

pop_back()
push_back(value)
insert(index,value)
erase(index)

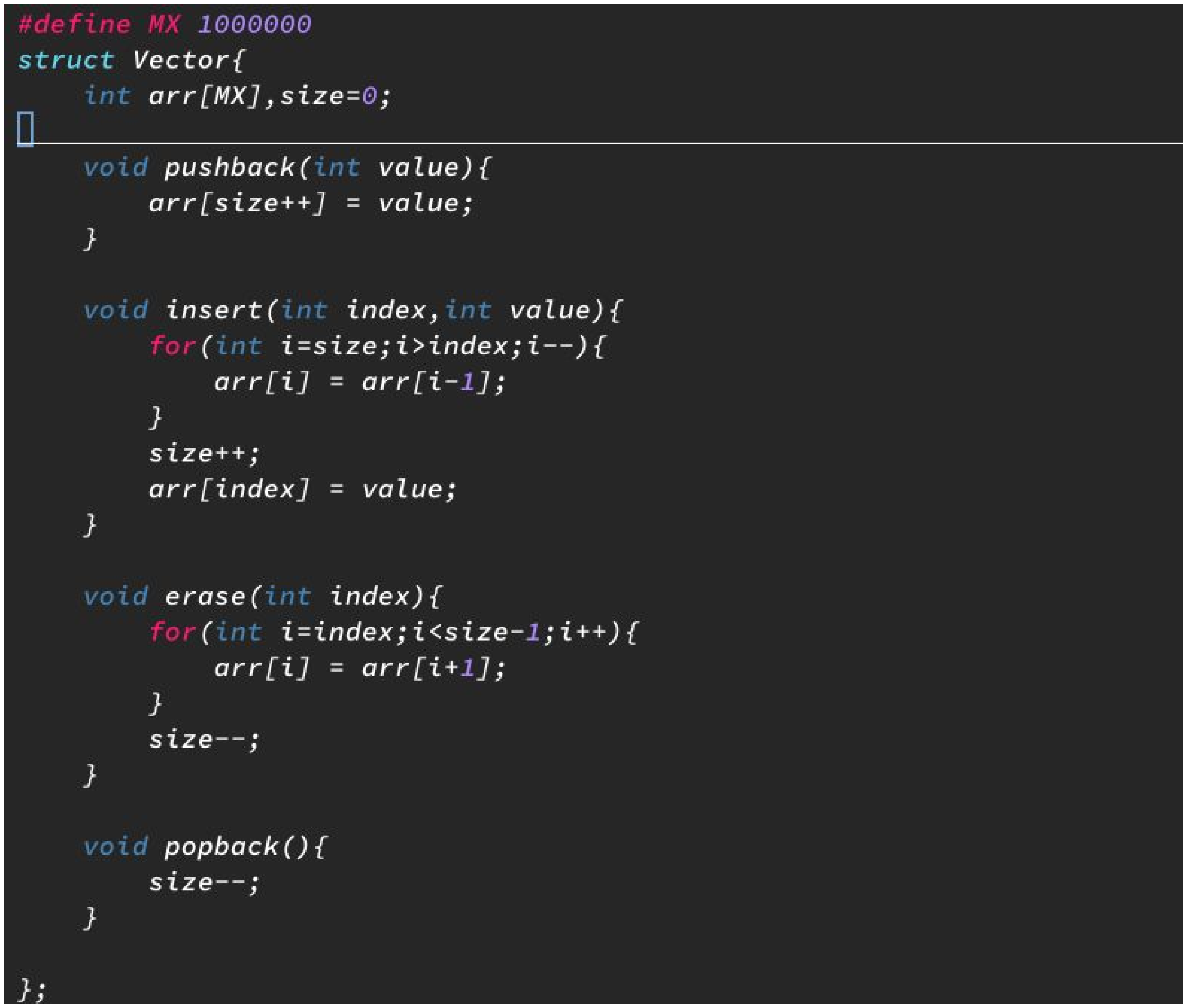
Iterators:

begin()
end()
rbegin()
rend()
Static arrays don’t have any of these functionalities



```
vector <int> arr;
arr.push_back(1);
arr.push_back(2);
arr.push_back(3);
arr.push_back(4);
// 1,2,3,4
arr.pop_back();
// 1,2,3
arr.erase(arr.begin()+2);
// 1,2
arr.insert(arr.begin()+1, 100);
// 1,100,2
for(auto it=arr.begin();it!=arr.end();it++){
    cout<<*it<<"\n";
}
```

Implementing vector functionalities using static arrays



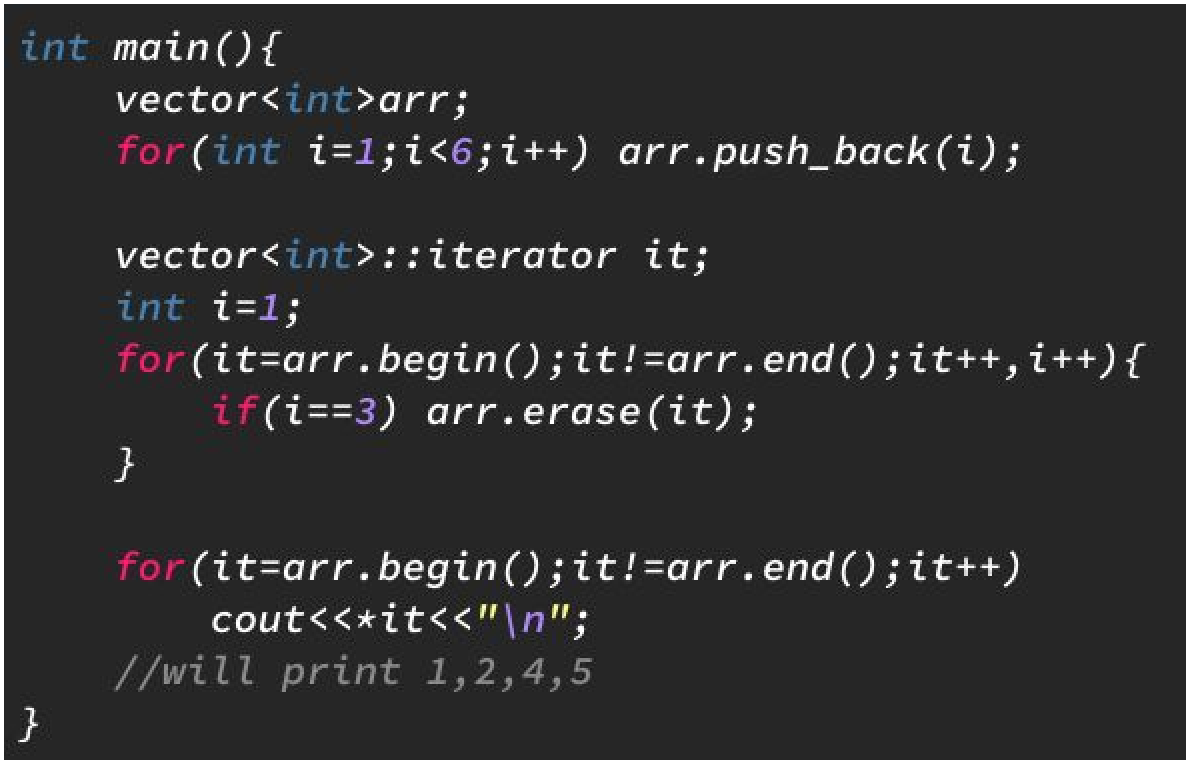
```
#define MX 1000000
struct Vector{
    int arr[MX],size=0;

    void pushback(int value){
        arr[size++] = value;
    }

    void insert(int index,int value){
        for(int i=size;i>index;i--){
            arr[i] = arr[i-1];
        }
        size++;
        arr[index] = value;
    }

    void erase(int index){
        for(int i=index;i<size-1;i++){
            arr[i] = arr[i+1];
        }
        size--;
    }

    void popback(){
        size--;
    }

};
```

pushback() and popback() are O(1)
erase and insert are O(n)

Task 2

Iterators points to a specific element of various stl containers such as vector and it can be used to traverse a range of elements,while a pointer is nothing but an address of memory.
A pointer can be used in various arithmetic operations but not all iterators can do that because iterators can point to something that is very complex
Pointers can be deleted using free,but in case of iterators they cant be deleted rather we can delete the value that the iterator is holding

Code:

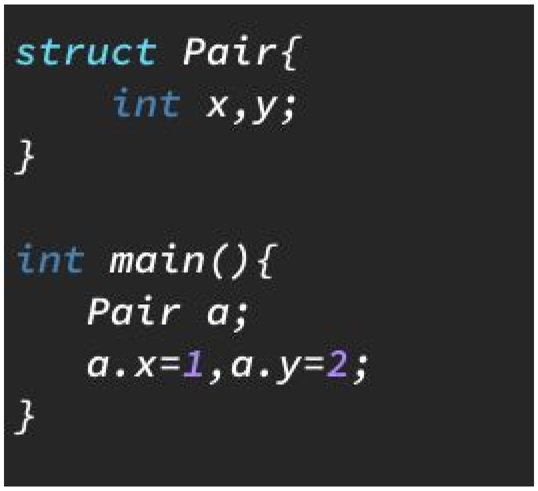


```
int main(){
    vector<int>arr;
    for(int i=1;i<6;i++) arr.push_back(i);

    vector<int>::iterator it;
    int i=1;
    for(it=arr.begin();it!=arr.end();it++,i++){
        if(i==3) arr.erase(it);
    }

    for(it=arr.begin();it!=arr.end();it++)
        cout<<*it<<"\n";
    //will print 1,2,4,5
}
```

Task3

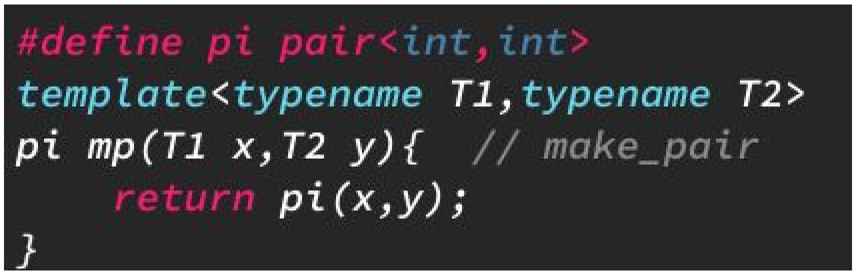
Alternative of pair

A replacement of pair can be structures



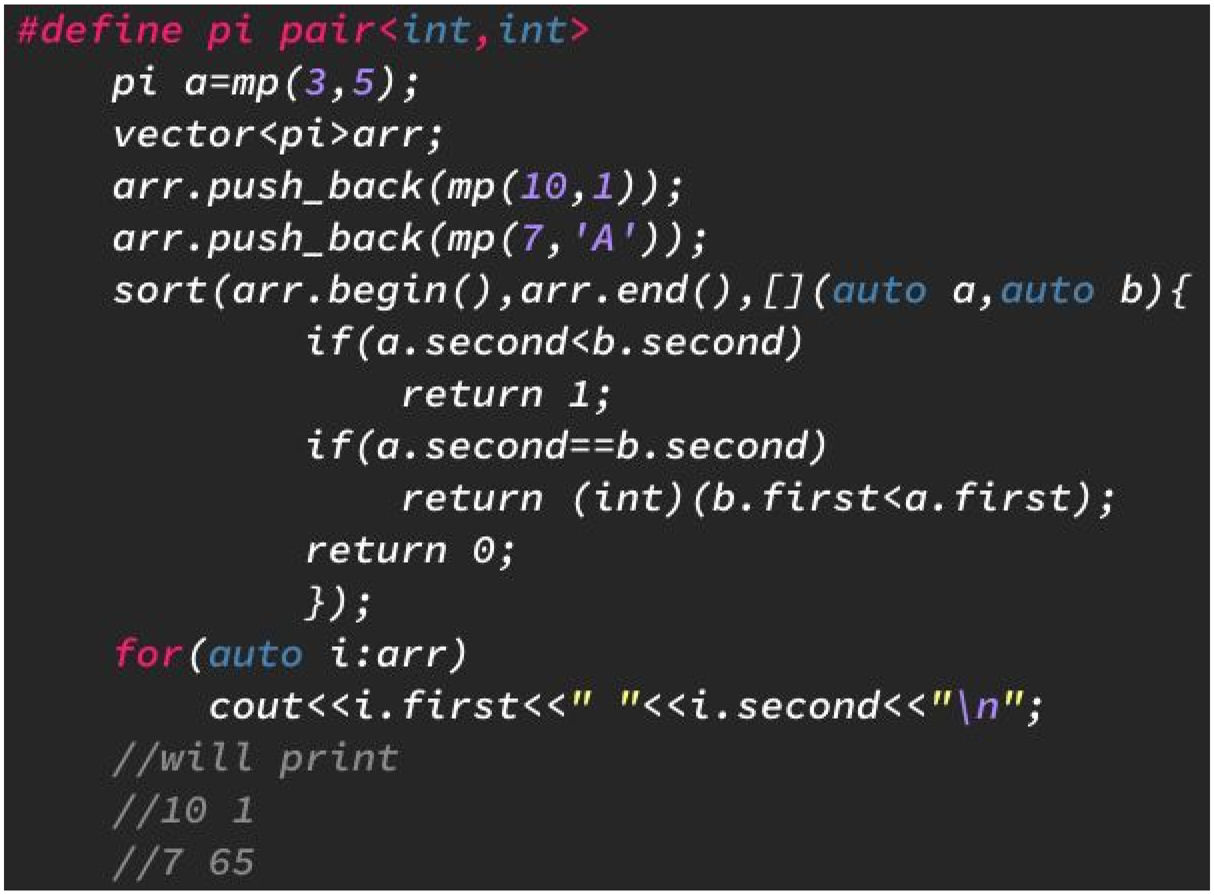
```
struct Pair{
    int x,y;
}

int main(){
   Pair a;
   a.x=1,a.y=2;
}
```

Implementing the make_pair functionalities of std::pair



```
#define pi pair<int,int>
template<typename T1,typename T2>
pi mp(T1 x,T2 y){  // make_pair
    return pi(x,y);
}
```

Sorting pairs based on the second element

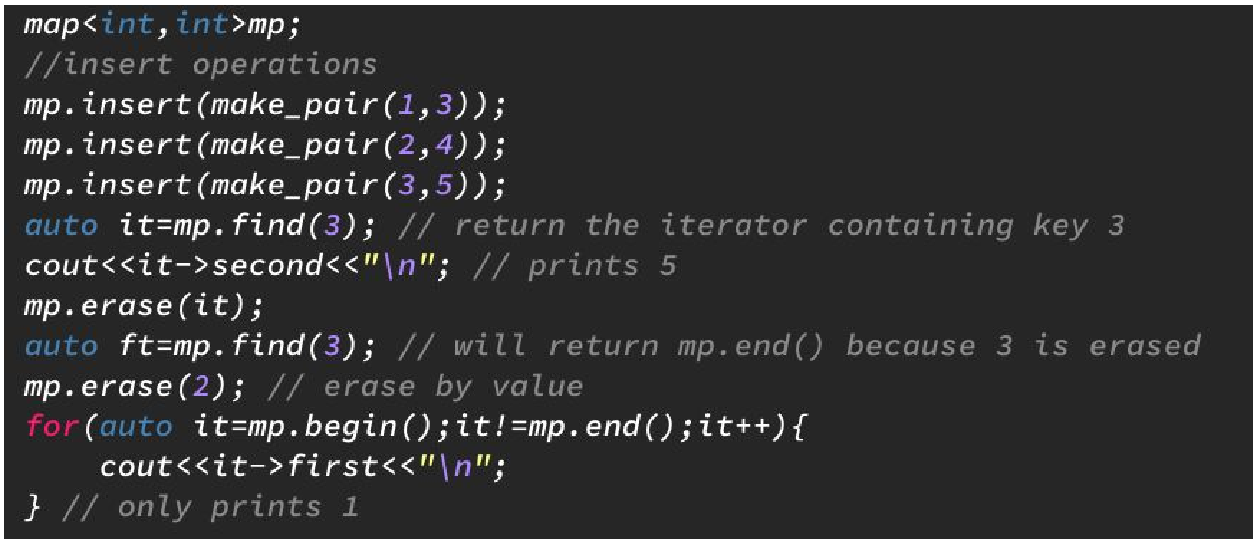


```
#define pi pair<int,int>
    pi a=mp(3,5);
    vector<pi>arr;
    arr.push_back(mp(10,1));
    arr.push_back(mp(7,'A'));
    sort(arr.begin(),arr.end(),[](auto a,auto b){
            if(a.second<b.second)
                return 1;
            if(a.second==b.second)
                return (int)(b.first<a.first);
            return 0;
            });
    for(auto i:arr)
        cout<<i.first<<" "<<i.second<<"\n";
    //will print
    //10 1
    //7 65
```

Task 4

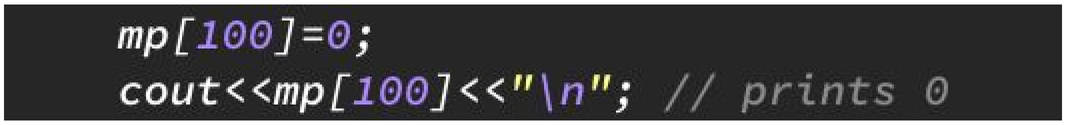
Maps store data as key and value pairs.
The key values are unique
Map is by default,sorted by key in ascending order
Functionalities of map are :
Insert(key)
erase(key/iterator)
clear()
count(key)
empty()
Iterators : begin()/rbegin(),end()/rend()

Using the stl maps various functionalities



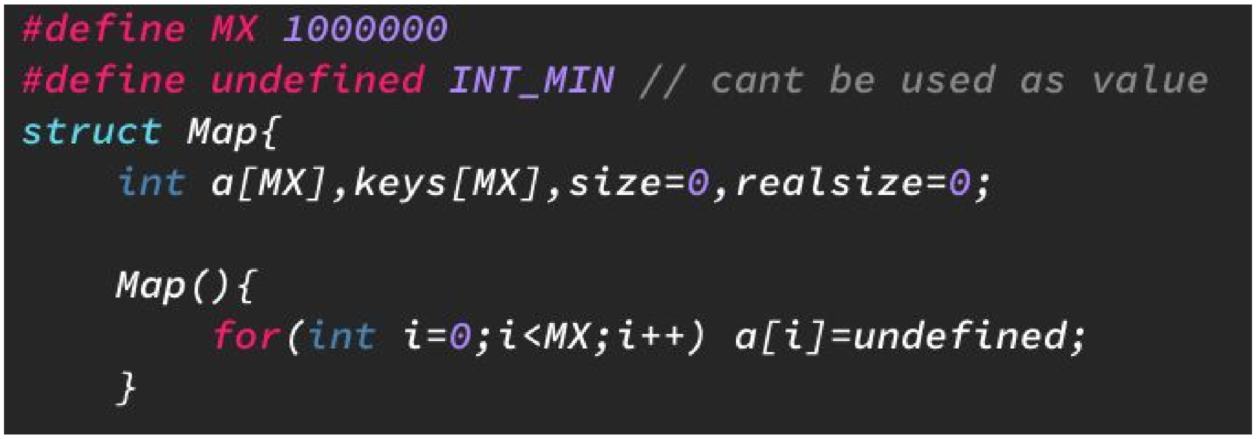
```
map<int,int>mp;
//insert operations
mp.insert(make_pair(1,3));
mp.insert(make_pair(2,4));
mp.insert(make_pair(3,5));
auto it=mp.find(3); // return the iterator containing key 3
cout<<it->second<<"\n"; // prints 5
mp.erase(it);
auto ft=mp.find(3); // will return mp.end() because 3 is erased
mp.erase(2); // erase by value
for(auto it=mp.begin();it!=mp.end();it++){
    cout<<it->first<<"\n";
} // only prints 1
```

Alternative method to insert



```
    mp[100]=0;
    cout<<mp[100]<<"\n"; // prints 0
```

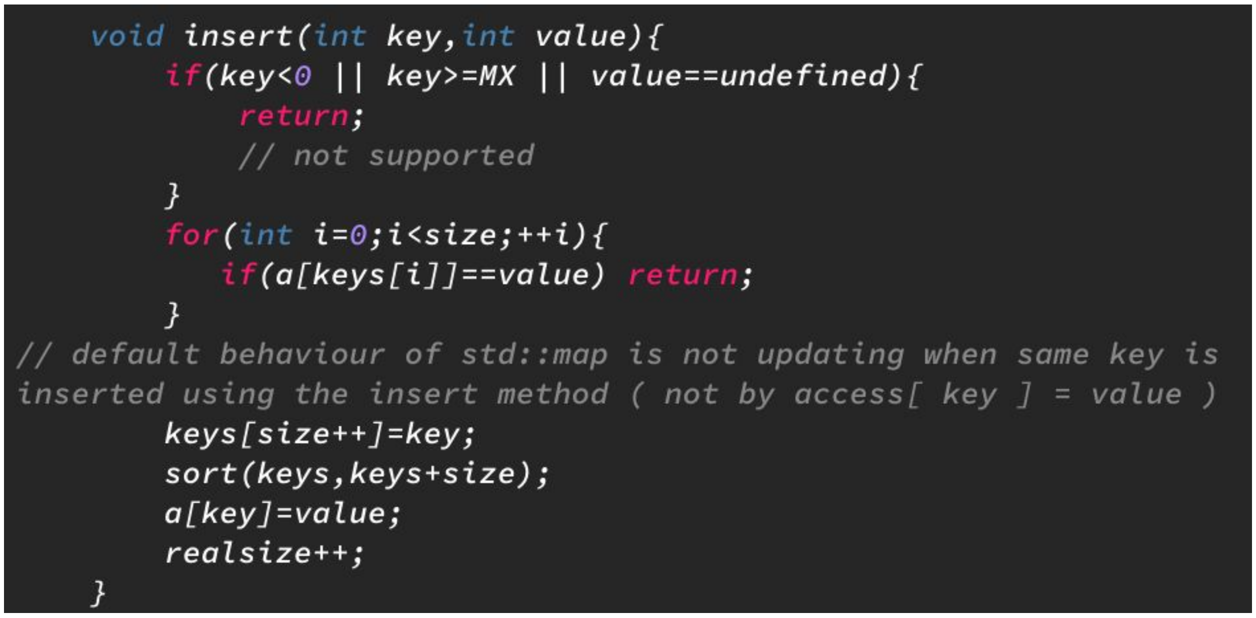
Implantation of map using static arrays



```
#define MX 1000000
#define undefined INT_MIN // cant be used as value
struct Map{
    int a[MX],keys[MX],size=0,realsize=0;

    Map(){
        for(int i=0;i<MX;i++) a[i]=undefined;
    }
```

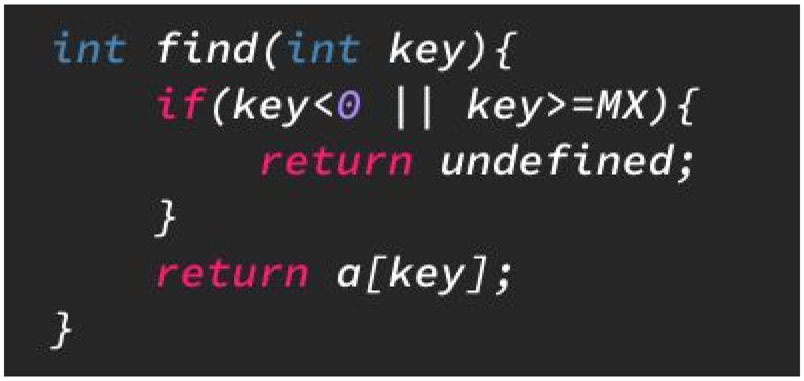
Initialising all values as undefined

Insertion



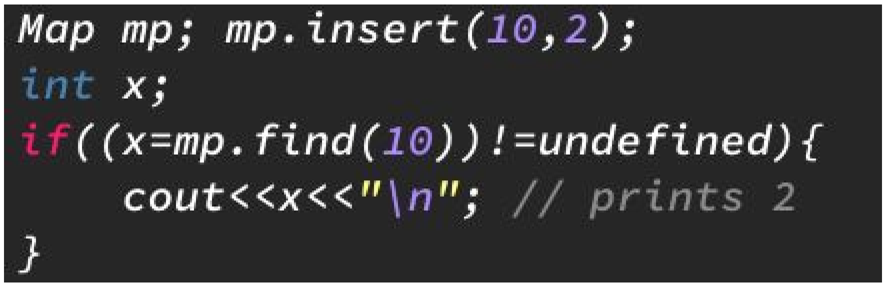
```
    void insert(int key,int value){
        if(key<0 || key>=MX || value==undefined){
            return;
            // not supported
        }
        for(int i=0;i<size;++i){
           if(a[keys[i]]==value) return;
        }
// default behaviour of std::map is not updating when same key is
inserted using the insert method ( not by access[ key ] = value )
        keys[size++]=key;
        sort(keys,keys+size);
        a[key]=value;
        realsize++;
    }
```

Find method



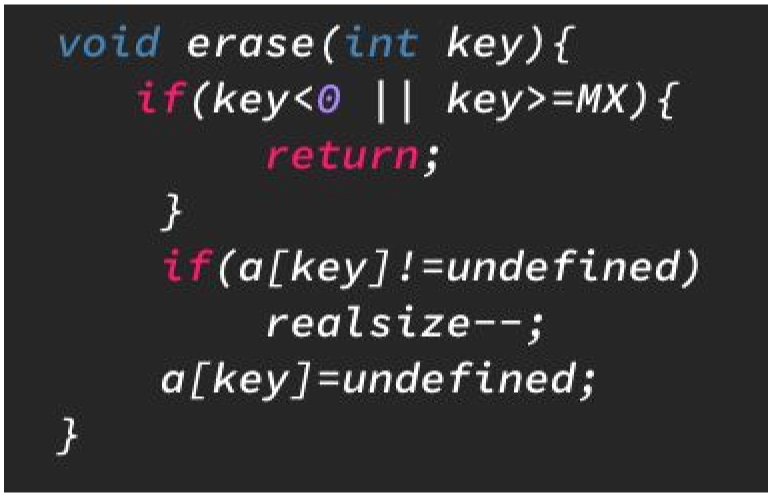
```
int find(int key){
    if(key<0 || key>=MX){
        return undefined;
    }
    return a[key];
}
```

Using this find method



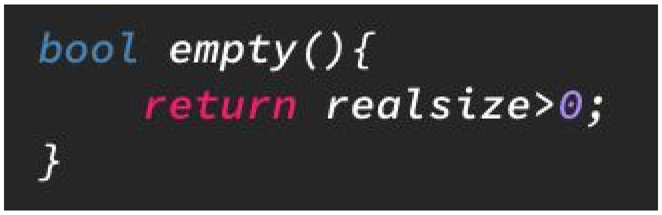
```
Map mp; mp.insert(10,2);
int x;
if((x=mp.find(10))!=undefined){
    cout<<x<<"\n"; // prints 2
}
```

Erasing



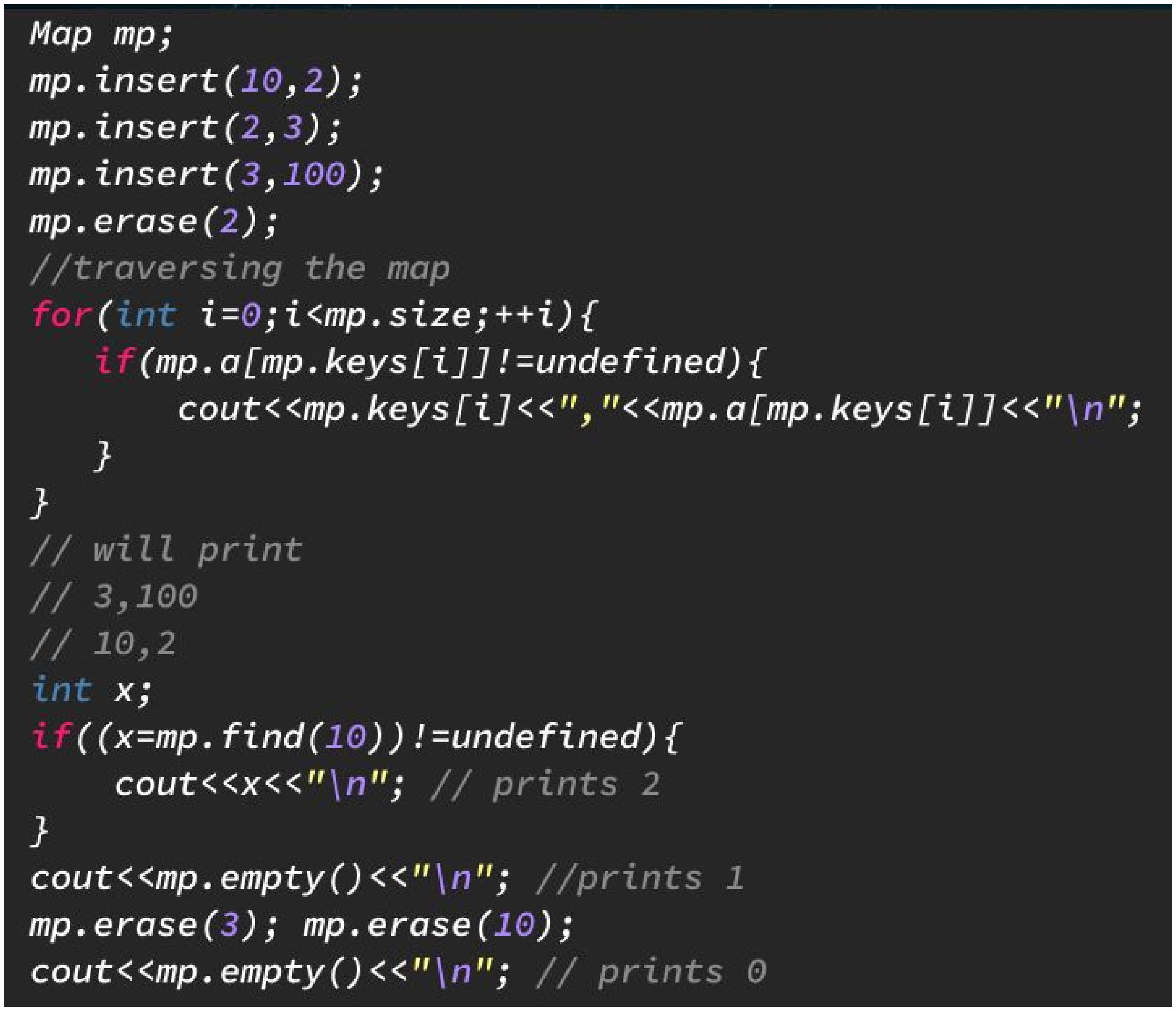
```
void erase(int key){
    if(key<0 || key>=MX){
        return;
    }
    if(a[key]!=undefined)
        realsize--;
    a[key]=undefined;
}
```

Checking if its Empty



```
bool empty(){
    return realsize>0;
}
```

Using this implementation of map



```
Map mp;
mp.insert(10,2);
mp.insert(2,3);
mp.insert(3,100);
mp.erase(2);
//traversing the map
for(int i=0;i<mp.size;++i){
   if(mp.a[mp.keys[i]]!=undefined){
       cout<<mp.keys[i]<<","<<mp.a[mp.keys[i]]<<"\n";
   }
}
// will print
// 3,100
// 10,2
int x;
if((x=mp.find(10))!=undefined){
    cout<<x<<"\n"; // prints 2
}
cout<<mp.empty()<<"\n"; //prints 1
mp.erase(3); mp.erase(10);
cout<<mp.empty()<<"\n"; // prints 0
```

Description of this implementation

Cons:

1. INT_MIN cant be used as a value
2. Negative key and key>MX are not supported
3. Values are not really deleted upon erase
4. Maximum MX number of insertion possible

Pros:

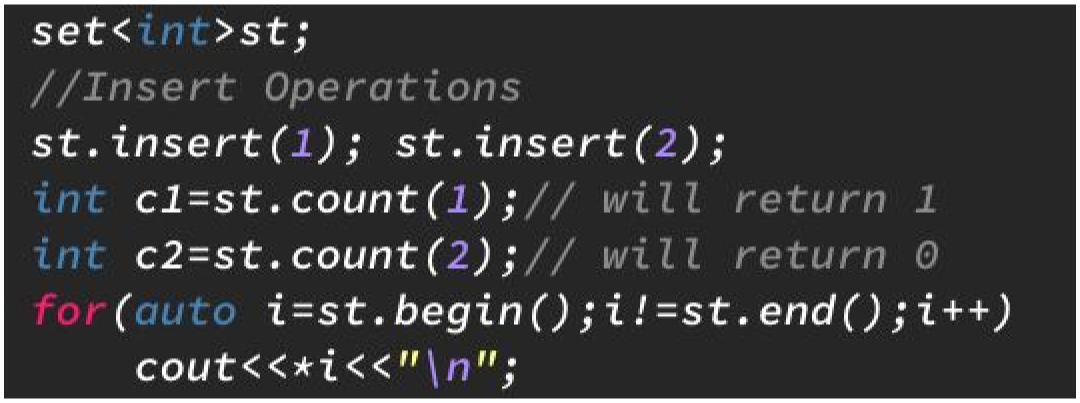
1. All operations beside insertions are O(1)

Task 5

Set stores unique values.It is used to identify existence of a certain value
Values can be deleted from the set.
std::set is by default sorted in ascending order.

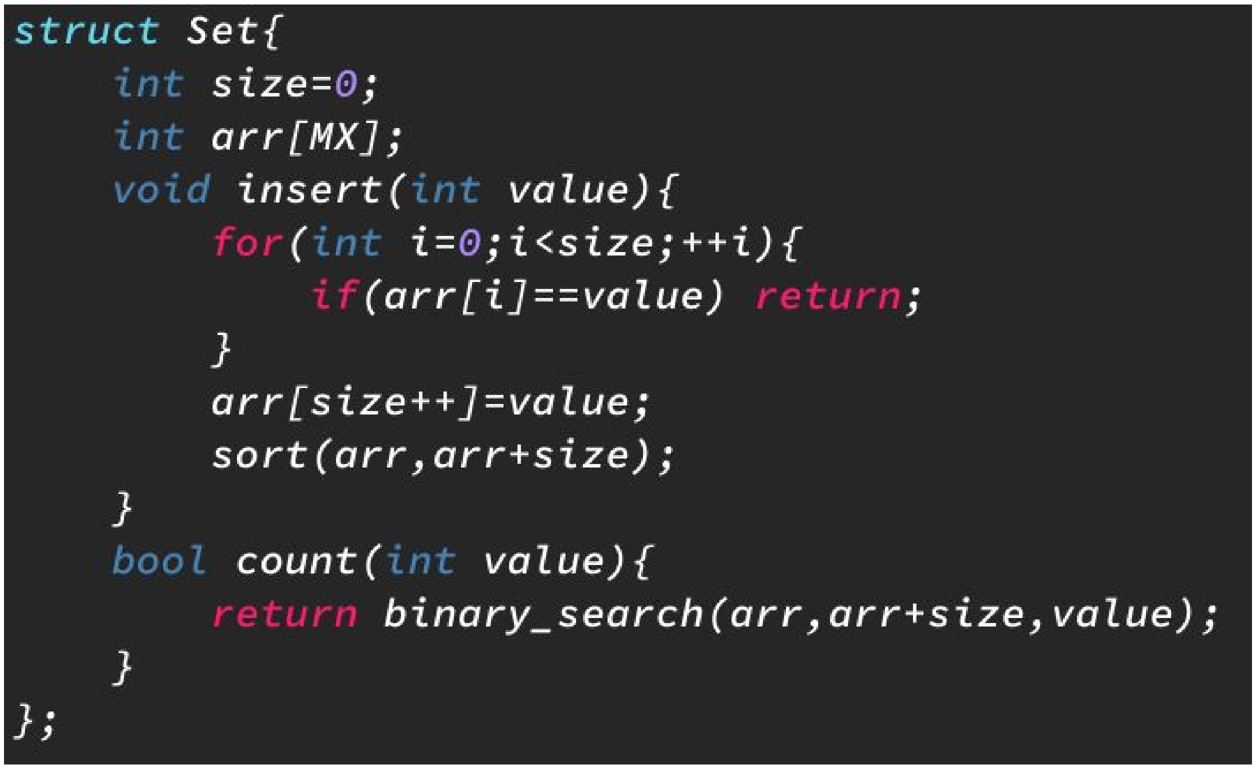
Methods of std::set :
insert(value)
erase(value/iterator)
size()
empty()
Iterators :
begin()/rbegin()
end()/rend()

Using the std::set container



```
set<int>st;
//Insert Operations
st.insert(1); st.insert(2);
int c1=st.count(1);// will return 1
int c2=st.count(2);// will return 0
for(auto i=st.begin();i!=st.end();i++)
    cout<<*i<<"\n";
```

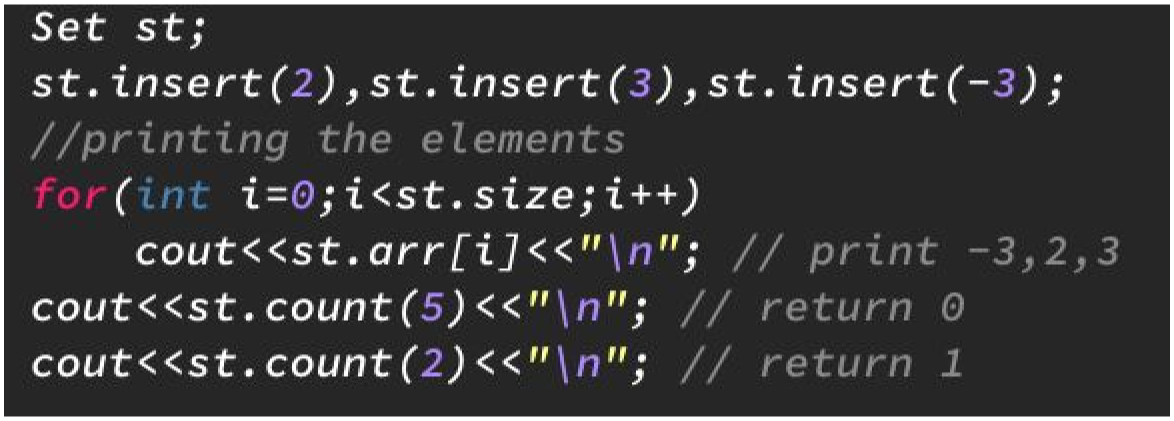
Implementation of set using arrays



```
struct Set{
    int size=0;
    int arr[MX];
    void insert(int value){
        for(int i=0;i<size;++i){
            if(arr[i]==value) return;
        }
        arr[size++]=value;
        sort(arr,arr+size);
    }
    bool count(int value){
        return binary_search(arr,arr+size,value);
    }
};
```

Insertion : O(nlogn)
Count : O(logn)

Using this implementation of set



```
Set st;
st.insert(2),st.insert(3),st.insert(-3);
//printing the elements
for(int i=0;i<st.size;i++)
    cout<<st.arr[i]<<"\n"; // print -3,2,3
cout<<st.count(5)<<"\n"; // return 0
cout<<st.count(2)<<"\n"; // return 1
```