TEMPLATE

Templates:Allow to write generic programs.It means we create single function or a class to work with different data types using templates.

There are two types of templates:

- Function based
- Class based

Function Based Template:

```
#include<iostream>
using namespace std;
template <class t>
t sum(t a,t b)
{
return  a+b;
}
int main()

{
   cout<<"sum"<<sum(5,6)<<"\n";
   cout<<"float sum="<<sum(12.4,34.5);


   return 0;

}
```

Class Based Template:

```
#include<iostream>
using namespace std;
template <class c>
class vik
{
    c a,b;
    public:vik(c n1,c n2)
    {
        a=n1;
        b=n2;
    }
    void grt()
    {
        if(a>b)
        cout<<a<<"is large\n";
        else
        cout<<b<<"is large\n";
    }
};
int main()

{
    vik <float>k(12.34,12.42);
    k.grt();
    vik<int>j(3,79);
    j.grt();
   vik<char>m('a','A');
    m.grt();
    return 0;

}
```

Example:We can use more than two arguments

```
#include<iostream>
using namespace std;
template <class T, class U>
class A {
   T x;
   U y;

public:
   A()
{
cout << "Constructor Called" << endl;
}
};

int main()
{
   A<char, char> a;
   A<int, double> b;
   return 0;
}
```

Default Arguments:

```
#include<iostream>
using namespace std;
template <class p=float,class q=int>
class joy
{
public:p sum(p a,q b)
{
   return a+b;
}
};
int main()
{
   Joy k;//we need not pass template bcz default argument,it will execute
   cout<<k.sum(12.45,34);
   return 0;
}
```

Without Default Arguments:we have to specify datatype during creation of an object else will generate an deduction error.

```
#include<iostream>
using namespace std;
template <class p,class q>
class joy
{
public:p sum(p a,q b)
{
   return a+b;
}
};



int main()
{
   Joy k;//generate error
   cout<<k.sum(12.45,34);
   return 0;
}
```

INHERITANCE 1:

```
#include<iostream>
using namespace std;
template<class t>
class boss
{

public:void show(t v)
{
   cout<<"boss code is="<<v<<endl;
}
void fun()
{
   cout<<"good\n";
}
};
```

```
template<class t1>
class emp:public boss<t1>
{
    public:void eshow(t1 v1)
    {
        cout<<"emp code is="<<v1<<endl;
    }
};
int main()
{
    emp<int>e;
    e.show(101);
   e.eshow(201);
   e.fun();

}
```

INHERITANCE 2:

```
#include<iostream>
using namespace std;
template <typename t>
class base{
   public:void show()
   {
      cout<<"base\n";
   }
void joy()
{

cout<<"joy\n";
}
};
template <typename t2>
class der:public base<t2>
{
   using base<t2>::show;
using base<t2>::joy;
   public:void get()
   {
      show();
     joy();
   }
   void set()
   {
      cout<<"derived class\n";
   }
};
int main()
{

der <int> d;
d.get();
d.set();
}
```
-----------*****---------************_****************-----------------------------

The C++ STL (Standard Template Library) 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 like vectors, lists, queues, and stacks.

It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized. Working knowledge of template classes is a prerequisite for working with STL.

Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one

C++ Standard Template Library has following three well-structured components −

1 Containers

Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map, etc.

- **Sequence Containers** - These containers implement data structures which can be accessed in a sequential manner.
    - array
    - vector
    - list
    - deque

    - forward_list

- **Container Adaptors** - They provide a different interface for sequential containers.
    - queue
    - priority_queue
    - stack
- **Associative Containers** - They implement sorted data structures that can be quickly searched (O(log n) complexity).
    - set
    - multiset
    - map
    - multimap
- **Unordered Associative Containers** - These containers implement unordered data structures that can be quickly searched
    - unordered_set
    - unordered_multiset
    - unordered_map
    - unordered_multimap

2 Algorithms

Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming the contents of containers.

Algorithm

- Sorting
- Searching
- Important STL Algorithms
- Useful Array algorithms
- Partition Operations

3 Iterators

Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

---**************---

Sequence Containers :

ARRAY

```
#include<iostream>
using namespace std;
#include<array>
int main()
{
    array<int,6>v{10,20,30,40,50,60};
    array<int,6>v2{100,200,300,400,500,600};
  int k=sizeof(v)/sizeof(v[0]);
cout<<"size of array="<<k;
```

MEMBER FUNCTIONS OF ARRAY:

```
//sizeof()
    for(int i=0;i<v.sizeof();i++)
    {
        cout<<v.at(i)<<endl;
    }
//at()
    cout<<"values of 2nd array\n";
     for(int i=0;i<k;i++)
    {
        cout<<v2.at(i)<<endl;
    }
//front() & back()
        cout<<"first element="<<v.front()<<endl;
        cout<<"last element="<<v.back();

//fill(value)
     v.fill(20);
        cout<<endl;
//swap():array1.swap(array2)
        v.swap(v2);
        cout<<"after swapping values of 1st array\n";
        for(int i=0;i<k;i++)
    {
        cout<<v.at(i)<<endl;
    }
    cout<<"after swapping values of 2nd array\n";
     for(int i=0;i<k;i++)
    {
       cout<<v2.at(i)<<endl;
    }
//iterator
    //array <int>::iterator it=v.begin();
    for(int value:v)//range based value //value is a copy of actual value
    {
       cout<<value++<<endl;
    }
    //range function:for(datatype variable:arrayname)
    //get copy of actual value
    for(int value:v)//value is not get incremented,its remain same
    {
       cout<<value<<endl;
    }
//range function:for(datatype &variable:arrayname)
 //now we work on reference,so values get incremented
        for(int &value:v)
    {
       cout<<value++<<endl;
    }

    //Now print the updated values
    for(int value:v)//value get incremented
    {
       cout<<value<<endl;
    }*/

//auto  :auto variable=value  //it dynamically determine data type
 auto a=9;//
    cout<<a;
    auto it=v.begin();//auto replaces iterator
    cout<<*it;
    cout<<*it+3;
}
```

--************------------------------------

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. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. 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. Inserting and erasing at the beginning or in the middle is linear in time.

Example 1:

```
#include<iostream>
using namespace std;
#include<vector>
#include<iterator>
```

```
int main()
{
    vector<int>v;
    v.push_back(101);
    v.push_back(102);
    v.push_back(103);
    v.push_back(104);
    v.push_back(105);
    cout<<"size="<<v.size()<<endl;
    cout<<"capacity="<<v.capacity()<<endl;
    for(int i=0;i<v.size();i++)
    {
        cout<<v[i]<<endl;
    }
    cout<<"front="<<v.front()<<endl;
     cout<<"back="<<v.back()<<endl;
     vector<int>::iterator it=v.end();
     v.pop_back();
     cout<<"pop"<<endl;
     for(int i=0;i<v.size();i++)
    {
        cout<<v[i]<<endl;
    }
    v.insert(it-2,108);
    cout<<"last="<<v.back()<<endl;
      v.insert(it-3,109);
       for(int i=0;i<v.size();i++)
    {
        cout<<v[i]<<endl;
    }
     vector<int>::iterator it2=v.begin();
      v.insert(it2+1,200);
      for(int i=0;i<v.size();i++)
    {
        cout<<v[i]<<endl;
    }
v.erase(v+2);//delete value at second position
v.clear();//delete all data
    vector <int> v1;//length 0
    vector <char> v2(5);//length 5//this is static array
    vector<int> v3(5,1);//5 blocks contain 1// this is static array
    vector <string> v4(3)//contains 3 strings// this is static array
    vector <string> v5(5,"hi")//each contain hi// this is static array
  const vector<int>k(5)//vector size can not be changed
vector<int> j1{1,2,3,4,5};
     vector<int> j2{10,20,30};
```

```
 cout<<"first"<<endl;
for(int i=0;i<j1.size();i++)

{
   cout<<j1.at(i)<<"\t";
}
  cout<<"second"<<endl;
for(int i=0;i<j2.size();i++)

{
   cout<<j2.at(i)<<"\t";
}
cout<<endl;
j1.swap(j2);
cout<<"after swappingfirst"<<endl;
for(int i=0;i<j1.size();i++)

{
   cout<<j1.at(i)<<"\t";
}
cout<<endl;
  cout<<"after swapping second"<<endl;
for(int i=0;i<j2.size();i++)
  {
   cout<<j2.at(i)<<"\t";
}
vector<int>::iterator t=j1.begin();
//insert(pointer,times,value)
j1.insert(t,2,222);//from 2 position 222 insert two times
cout<<endl;
 for(int i=0;i<j1.size();i++)
    {
   cout<<j1.at(i)<<"\t";
}
vector<int>::iterator t2=j1.end();
j1.insert(t2,{11,22,33,44,45});//vector.insert(pointer,{values})
cout<<"multi values inserted at end of vector\n";
 for(int i=0;i<j1.size();i++)
{
   cout<<j1.at(i)<<"\t";
}
}
vector<int>::iterator t3=j1.begin();
 j1.erase(t3,t3+3);//erase from range
cout<<"\n after erasing\n";
 for(int i=0;i<j1.size();i++)
```

```
        {
        cout<<j1.at(i)<<"\t";
    }
    1.assign({40});//assign({replace with value})
    j1.assign(7,{5});//assign(times,{replace with value})
    cout<<"assign new values\n";

     for(int i=0;i<j1.size();i++)

    {
        cout<<j1.at(i)<<"\t";
    }
vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

        cout << "\nOutput of rbegin and rend: ";
    for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir)//rbegin()=reverse begin
        cout << *ir << " ";

    cout << "\nOutput of crbegin and crend : ";
    for (auto ir = g1.crbegin(); ir != g1.crend(); ++ir)//rend()=reverse end
        cout << *ir << " ";

   vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    cout << "Size : " << g1.size();
    cout << "\nCapacity : " << g1.capacity();
    cout << "\nMax_Size : " << g1.max_size();

    // resizes the vector size to 4
    g1.resize(3);

    // prints the vector size after resize()
    cout << "\nSize : " << g1.size();

    // checks if the vector is empty or not
    if (g1.empty() == false)
        cout << "\nVector is not empty";
    else
        cout << "\nVector is empty";
    // Shrinks the vector
    g1.shrink_to_fit();
    cout << "\nVector elements are: ";
    for (auto it = g1.begin(); it != g1.end(); it++)
        cout << *it << " ";
```

---*****--

Example 2:

```
#include<iostream>
using namespace std;
#include<vector>
void show(vector<int> &arr)
{
     for (int i=0;i<arr.size();i++)
     {
         cout<<arr[i]<<" ";
     }
     cout<<endl;
}

int main()
{
    vector<int> v;
    int x;
    cout<<"values are----------";
    for (int i=0;i<4;i++)
    {

        cin>>x;
        v.push_back(x);
    }
    show(v);
    return 0;
}
```

PAIR:

Example 1

```
#include<iostream>

#include<vector>

#include<algorithm>
```

```
using namespace std;
int main()
{
    pair<int,int> pr;
    pr.first=10;
    pr.second=78;
    cout<<pr.first<<","<<pr.second;

}
```

Example 2:

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
int main()
{
    pair<int,pair<int,string>> pr;
    pr.first=1;
    pr.second.first=1;
    pr.second.second=”rohan”;
    cout<<pr.first<<","<<pr.second.first<<","<<pr.second.second;

}
```

Example 3:

```
#include<iostream>

#include<vector>

#include<algorithm>

using namespace std;

int main()

{

   vector <pair<int,int>> pr;

   int n,a,b;

   cout<<"enter records\n";

   cin>>n;

   for(int i=0;i<n;i++)

   {

      cin>>a>>b;

      pr.push_back(make_pair(a,b));

   }

   for(int j=0;j<n;j++)

   {

     cout<<pr[j].first<<","<<pr[j].second<<endl;

   }
```

Example 3:

```
#include<iostream>

#include<vector>
```

```
using namespace std;

int main()

{

   vector<pair<int,pair<int,int>>>v;

   v.push_back({1,make_pair(50,60)});

   for(int i=0;i<1;i++)

   {

       cout<<v[i].first<<v[i].second.first<<v[i].second.second;

   }

}


}
```

TUPLE:

Example 1:

```
#include<iostream>

#include<vector>

#include<algorithm>

#include<tuple>

using namespace std;

int main()

{

   tuple <int,int,int>tr;
```

```
    tuple <int,int,int>tr2{10,20,30};

    int a,b,c;

    cin>>a;

    cin>>b;

    cin>>c;

    tr=make_tuple(a,b,c);

    cout<<get<0>(tr)<<",";

     cout<<get<1>(tr)<<",";

     cout<<get<2>(tr)<<endl;


}
```

Example 2:

```
#include<iostream>

#include<vector>

#include<algorithm>

#include<tuple>

using namespace std;

int main()

{

    tuple <int,int,int>tr;

    tuple <int,int,int>tr2{23,2,56};

    int a,b,c;

    cin>>a;

    cin>>b;
    cin>>c;

    tr=make_tuple(a,b,c);

    cout<<get<0>(tr)<<",";

     cout<<get<1>(tr)<<",";

     cout<<get<2>(tr)<<endl;

     if (tr==tr2)

     {

        cout<<"same";  }

     else

     {

        cout<<"not same";   }


}
```

Forward_list

Member function of list

- **Insert_after(startpos,value)**
- **Emplace_after(startpos,value)**
- **Obj.Reverse()**
- **Obj.unique()**
- **Obj.sort()**
- **Obj1.merge(obj2)**
- **Obj.splice_after(startpos,listobj)**
- **Obj.remove(value)**

- **Obj.remove_if([] (){statement;})**
- **Obj.resize(value)**
- **Obj.clear()**

```
#include<iostream>
#include<forward_list>
#define flist1 forward_list<int>
#define flist2 forward_list<int>
using namespace std;
int main()
{
   flist1 f1={10,200,30,40,500};
   flist2 f2={1001,202,300,4001,500};
   cout<<"values of list1\n";
   for(int &k1:f1)
   {
      cout<<k1<<"\t";
   }
   cout<<"\nvalues of list2\n";
   for(auto k2:f2)
   {
      cout<<k2<<"\t";
   }
   //f1.insert_after(f1.begin(),9);//insert_after(val1,val2)//it creates object then copy and assign
   f1.emplace_after(f1.begin(),8);//similar to insert but it will create object and assign
   cout<<endl;
   for(int &k1:f1)
   {
      cout<<k1<<"\t";
   }
   cout<<"\n";
   f1.reverse();//reverse()
   for(int &k1:f1)
   {
      cout<<k1<<"\t";
   }
   f1.sort();
   f2.sort();
   cout<<"\n";
   for(int &k1:f1)
   {
      cout<<k1<<"\t";
   }
   cout<<"\nvalues of list2\n";
   for(auto k2:f2)
   {
      cout<<k2<<"\t";
   }
cout<<"\nafter merge\n";
   f1.merge(f2);//merge()will apply only after the list is to be sorted otherwise it will act as a
concatenation

   for(int &k1:f1)
   {
      cout<<k1<<"\t";
   }
   cout<<"\n using splice_after"<<endl;
   flist1 f3={10,200,30,40,500};
   flist2 f4={1001,202,300,4001,500};
   f3.splice_after(f3.begin(),f4);//we can insert entire list into another list in any position by
using it.
   for(int &kt:f3)
   {
      cout<<kt<<"\t";
   }
   flist1 f5={10,200,40,40,500};
   cout<<endl;
   f5.unique();//it will remove matching element but that should be adjacent
   for(int &kt:f5)
   {
      cout<<kt<<"\t";
   }
   flist1 f6={40,200,400,40,500};
   cout<<endl;
   f6.unique();//it will not remove matching element bcz 40 is not  adjacent
  forward_list<int>k;
   forward_list<int>k2;
   k.assign({12,3,4,5,67,8});
   for(auto a:k)
   {
      cout<<a<<endl;
   }
   k.push_front(200);
   for(auto a:k)
   {
      cout<<a<<endl;
```

```
    }
    k.pop_front();
    for(auto a:k)
    {
       cout<<a<<endl;
    }
    k2.assign(4,22);
    for(auto a:k2)
    {
       cout<<a<<endl;
    }
    for(int &kt:f6)
    {
       cout<<kt<<"\t";
    }
    cout<<"\n";
    f6.remove(400);//remove particular value
     for(int &kt:f6)
    {
       cout<<kt<<"\t";
    }
    f6.remove_if([](int n){ return n>200;});//remove_if() by using lamda function
    cout<<"after using lamda function\n";
     for(int &kt:f6)
    {
       cout<<kt<<"\t";
    }
    cout<<endl;
    f6.resize(2);//it will delete over two elements
for(int &kt:f6)
    {
       cout<<kt<<"\t";
    }
    cout<<"\n";
    f6.resize(6);//it wil set value 0 rest of the values
     for(int &kt:f6)
    {
       cout<<kt<<"\t";
    }
    f1.clear();
   cout<<"after clear\n";
     for(int &kt:f1)
    {
       cout<<kt<<"\t";
    }

}
```

LIST:

- List class supports a bidirectional ,linear list
- Vector supports random access but a list can be accessed sequentially only.
- List can be accessed front to back or back to front.
- Member functions of List
    - Push_back()
    - Push_front()
    - Pop_back()
    - Pop_front()

- List.reverse()
- List.sort()
- List.remove(value)
- List.clear()

```
#include<iostream>

using namespace std;

#include<list>

int main()

{

   list <int> p;

   list <int> l1{11,2,30,4,50};

   list <string> l2{"sun","mon","tue"};

   l2.push_back("wed");

   l2.push_front("thur");

cout<<l1[1];//it[] does not support in list

l2.reverse();


   //witout iterator we cant access the value of list

   list<string>::iterator t=l2.begin();

   while(t!=l2.end())

   {

      cout<<*t<<"\n";

      t++;
```

```
    }


    list<int>::iterator t2=l1.begin();
     while(t2!=l1.end())
    {
        cout<<*t2<<"\n";
        t2++;
    }
    l1.pop_front();
cout<<"after pop_front()"<<"\n";
     list<int>::iterator t3=l1.begin();
     while(t3!=l1.end())
    {
        cout<<*t3<<"\n";
        t3++;
    }
    cout<<"after pop_back()"<<"\n";
    l1.pop_back();
    list<int>::iterator t4=l1.begin();
    while(t4!=l1.end())
    {
        cout<<*t4<<"\n";
        t4++;
    }
```

```
            l1.sort();
            cout<<"after sorting\n";
            list<int>::iterator t5=l1.begin();
            while(t5!=l1.end())
            {
                cout<<*t5<<"\n";
                t5++;
            }
            l1.reverse();
            cout<<"after reverse\n";
            list<int>::iterator t6=l1.begin();
            while(t6!=l1.end())
            {
                cout<<*t6<<"\n";
                t6++;
            }

cout<<"after remove\n";
    //remove particular value
    l1.remove(4);
    list<int>::iterator t7=l1.begin();
    while(t7!=l1.end())
    {
        cout<<*t7<<"\n";
        t7++;
    }
    l2.clear();
    cout<<"size="<<l2.size();

    return 0;
}
```

QUEUE

```
#include<iostream>
using namespace std;
#include<queue>
void show(queue <int> i)
{  queue<int>j=i;
    while(!j.empty())
{
    cout<<j.front()<<endl;
```

```
   j.pop();}  }
int main()
{
queue<int> q;
q.push(10);
q.push(-23);
q.push(34);
q.push(20);
show(q);
cout<<endl;
q.pop();
cout<<q.front()<<endl;
cout<<q.size()<<endl;
cout<<q.back()<<endl;}
```

DEQUEUE(Double Ended Queue)

```
#include<bits/stdc++.h>
using namespace std;
void showdq(deque<int> dq)
{
  deque<int>::iterator it;
  for(it=dq.begin();it!=dq.end();it++)
  {  cout<<*it<<endl;  }}
int main()
{
   deque<int> dq;

   dq.push_back(20);

   dq.push_front(2);

   dq.push_back(-97);

   dq.push_front(54);

   showdq(dq);

   cout<<"size="<<dq.size()<<endl;

    cout<<"maxsize="<<dq.max_size()<<endl;

    cout<<dq.front()<<endl;

    cout<<dq.back()<<endl;

    dq.pop_back();

    showdq(dq);

    dq.pop_front();

    showdq(dq);}
```

STACK(LIFO)

Ex:valid parenthesis,next greater element

Member functions:

- size()
- push()
- pop()
- empty()
- =

```
#include<iostream>

using namespace std;

#include<stack>

#include<iterator>

int main()
```

```
{
   stack<int> s;
   s.push(10);
   s.push(34);
   s.push(56);
   s.push(43);
   s.push(78);
cout<<"size="<<s.size()<<endl;
   while(!s.empty())
   {
      cout<<s.top()<<endl;
      s.pop();
   }
}
```

LAMDA FUNCTION:

```
#include<iostream>
using namespace std;
#include<vector>
#include<algorithm>
bool jj(int x)
{
   return x>20;
}
int main()
```

```
{
   //all_of();none_of(),any_of();
   cout<<[](int a){return a+2;}(4)<<endl;
   auto p=[](int a,int b){return a+b;};
   cout<<p(45,6)<<endl;
   vector<int> v={23,10,59};
   vector<int> v2={2,1,5};
  cout<<all_of(v.begin(),v.end(),jj)<<endl;
   cout<<all_of(v.begin(),v.end(),[](int x){return x>6;})<<endl;
   cout<<any_of(v.begin(),v.end(),[](int x){return x>16;})<<endl;
    cout<<none_of(v2.begin(),v2.end(),[](int x){return x>6;})<<endl;
    sort(v.begin(),v.end());
    cout<<"sorted array\n";
    for(auto j:v)
    {
        cout<<j<<endl;
    }
    auto cmp=[](int a,int b)
    {
        return a>b;    };
    sort(v.begin(),v.end(),cmp);
    cout<<"sorted array in descending\n";
    for(auto j:v)
    {
        cout<<j<<endl;
    }
    }
auto my=[&v](int t)
    {
        for(int i=0;i<v.size();i++)
        {
            v[i]+=t;    }
    };
    my(2);
    for(auto p:v)
    {   cout<<p<<endl;
    }
```

BITSET(static allocation in bit)

Member functions:

- **Obj.count()**
- **Obj.size()**
- **Obj.test()**
- **Obj.any()**
- **Obj.none()**
- **Obj.set()**
- **Obj.set(pos,value)**
- **Obj.reset()**

- **Obj.flip()**

```
#include<iostream>

using namespace std;

#include<bitset>

int main()

{

    bitset<8> b(35);//bitset<size> objectname//size should be static,we can not create
when it is dynamic

    cout<<b<<endl;

    cout<<"count of 1="<<b.count()<<endl;

    cout<<"size="<<b.size()<<endl;

    cout<<"position="<<b.test(3)<<endl;//read form right to left//for position value

    cout<<"any set 1="<<b.any()<<endl;//return true(1) if found any 1

    cout<<"none set 0="<<b.none()<<endl;//return 1 if all are zeros but now it returns 0

   // cout<<"set 1="<<b.set()<<endl;//return set all 1

    cout<<"set particular position value="<<b.set(4,0)<<endl;//set posito 4=0

    cout<<"reset=all value 0"<<b.reset()<<endl;

    cout<<"flip="<<bflip()<<endl;

    cout<<"flip particular position="<<b.flip(3)<<endl;

cout<<"b.all()<<endl;//if all bits are 1 return 1 else 0

}
```

TYPE CASTING:

- Static_cast
- Reinterpret_cast

- Constant_cast
- Dynamic_cast

```
#include<iostream>

using namespace std;

void fun(int* p)

{

    cout<<*p+1;

}

int main()

{

    unsigned int t=-1;

    cout<<static_cast<int>(t);

    cout<<"\npointer value=";

    int *p=new int(97);

    cout<<"address of int pointer=";

    cout<<p<<endl;

    cout<<"value of int pointer=";

    cout<<p<<endl;

    cout<<"convert into char pointer\n";

    char *c=reinterpret_cast<char *>(p);

     cout<<"\naddress of char pointer=";

    cout<<c;

     cout<<"\nvalue of char pointer=";

     cout<<*c<<endl;
     const int num=200;

     const int *ptr=&num;

     int *w=const_cast<int*>(ptr);

     fun(w);


}
```

Dynamic_cast:

```
#include<iostream>

using namespace std;

class base

{

    public:virtual void show();

};


class der:public base

{

    public:void show()

    {

        cout<<"derive\n"; }

};

int main()

{

    der d;

      base *bp=dynamic_cast<base*>(&d);
```

```
if(bp==nullptr)
{
   cout<<"null"<<endl;
}
else
{   cout<<"not null"<<endl;
    bp->show();   }
der *d2=dynamic_cast<der*>(bp);
 if(d2==nullptr)
{
   cout<<"null"<<endl;  }
else
{cout<<"not null"<<endl;
    d2->show();
}
```

MAP

- Works in key,value pair
- Sorted
- log(n) times
- Red black tree(self balance tree)
- Uses in graph and lexography order
- we can perform insert,erase and find operation

EXAMPLE:

```
#include<iostream>
using namespace std;
```

```
#include<map>

int main()

{

    map<int,int> mp;

    mp.insert({101,65});

    mp.insert({104,90});

    mp.insert({106,73});

    mp.insert({105,90});

    mp.insert({109,69});

    mp.insert({102,73});

    map<int,int>::iterator it;

    for(it=mp.begin();it!=mp.end();it++)

    {

        cout<<it->first<<","<<it->second<<endl;

    }


mp.erase(105);

    cout<<"after erasing\n";

    for(it=mp.begin();it!=mp.end();it++)

    {

        cout<<it->first<<","<<it->second<<endl;

    }

mp.erase(it.begin(),mp.find(104));

      cout<<"after erasing\n";

    for(it=mp.begin();it!=mp.end();it++)
    {
        cout<<it->first<<","<<it->second<<endl;
    }
cout<<mp.size()<<endl;
    cout<<mp.clear()<<endl;
    cout<<mp.max_size()<<endl;
}
```

INRODUCTION OF RED BLACK TREE:

When it comes to searching and sorting data, one of the most fundamental data structures is the binary search tree. However, the performance of a binary search tree is highly dependent on its shape, and in the worst case, it can degenerate into a linear structure with a time complexity of O(n). This is where Red Black Trees come in, they are a type of balanced binary search tree that use a specific set of rules to ensure that the tree is always balanced. This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.

Red Black Trees are self-balancing, meaning that the tree adjusts itself automatically after each insertion or deletion operation. It uses a simple but powerful mechanism to maintain balance, by coloring each node in the tree either red or black.

Red Black Tree-

Red-Black tree is a binary search tree in which every node is colored with either red or black. It is a type of self balancing binary search tree. It has a good efficient worst case running time complexity.

Properties of Red Black Tree:

The Red-Black tree satisfies all the properties of binary search tree in addition to that it satisfies following additional properties –

1. **Root property:** The root is black.

2. **External property:** Every leaf (Leaf is a NULL child of a node) is black in Red-Black tree.

3. **Internal property:** The children of a red node are black. Hence possible parent of red node is a black node.

4. **Depth property:** All the leaves have the same black depth.

5. **Path property:** Every simple path from root to descendant leaf node contains same number of black nodes.

The result of all these above-mentioned properties is that the **Red-Black tree** is roughly balanced.

Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O(log n) after every insertion and deletion, then we can guarantee an upper bound of O(log n) for all these operations. The height of a Red-Black tree is always O(log n) where n is the number of nodes in the tree.

Sr. No.	Algorithm	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)

Sr. No.	Algorithm	Time Complexity
3.	Delete	O(log n)

Comparison with AVL Tree:

The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves frequent insertions and deletions, then Red-Black trees should be preferred. And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over the Red-Black Tree.

Rules That Every Red-Black Tree Follows:

- It is a self balancing BST
- Every node is either Black or Red
- Root is always Black
- Every leaf which is Nil is Black
- If node is Red then its children are Black
- Every path from a node to any of its descendent Nil node has same number of Black nodes

Red Black Tree Insertion rules:

- If tree is empty,create newnode as root node with color **BLACK.**
- If tree I not empty,create new node as leafnode with color **RED.**
- If parent of new node is **BLACK** then exit.
- If parent of new node is **RED,**then check the color of parent sibling of new node
    1. If color is **BLACK or NIL** ,there do rotation and recolor
    2. If color is **RED** then recolor & also check if parent's parent of new node is not root then recolor it & recheck

Note:

- Root:**Black**
- No two adjacent node will be **RED**
- Count number of BLACK nodes in each path should be same

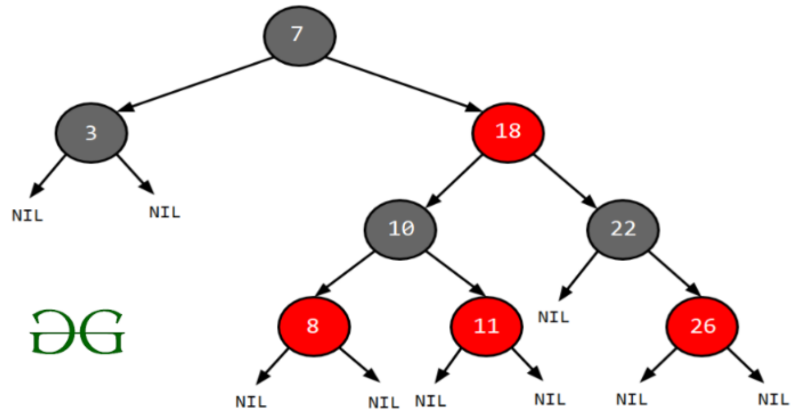
Search Operation in Red-black Tree:

As every red-black tree is a special case of a binary tree so the searching algorithm of a red-black tree is similar to that of a binary tree.

Algorithm:

```
searchElement (tree, val)

Step 1:
If tree -> data = val OR tree = NULL
    Return tree
Else
If val < data
        Return searchElement (tree -> left, val)
    Else
        Return searchElement (tree -> right, val)
    [ End of if ]
[ End of if ]

Step 2: END
```

Example: Searching 11 in the following red-black tree.

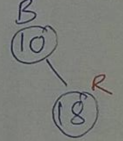


Solution:

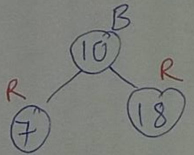
1. Start from the root.
2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.
3. If the element to search is found anywhere, return true, else return false.

10, 18, 7, 15, 16, 30, 25, 40, 2

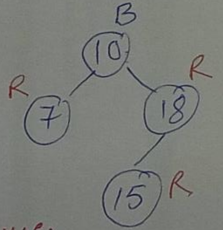
(1)



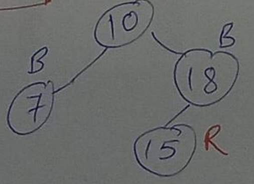
(2)



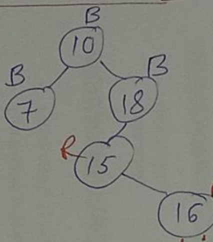
(3)



RECULURING:-

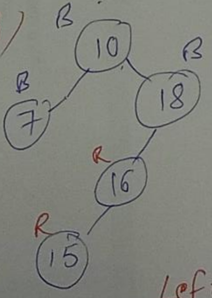


(4)



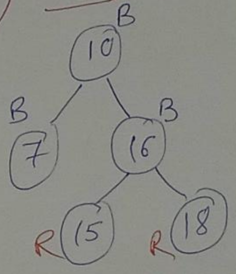
Parent's parent is Black so do ratation (LR) & recolor

(i)



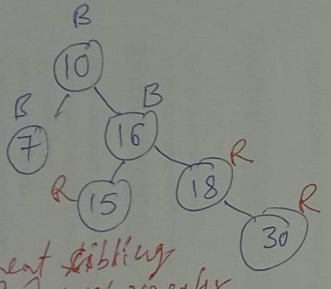
Left Rotation

(ii) RIGHT ROTATION



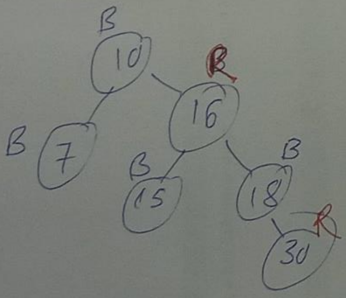
Recolor it.

(5)

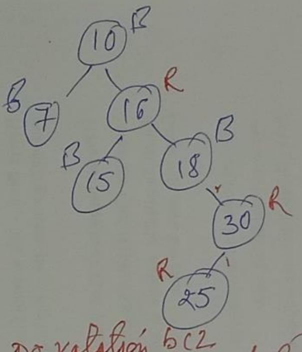


Parent's Parent sibling is Red now recolor

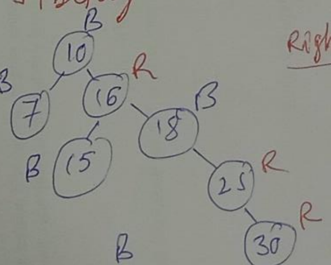
(i)



(6i)

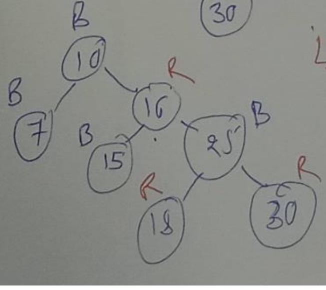


(i) Do rotation bcz sibling is not Red



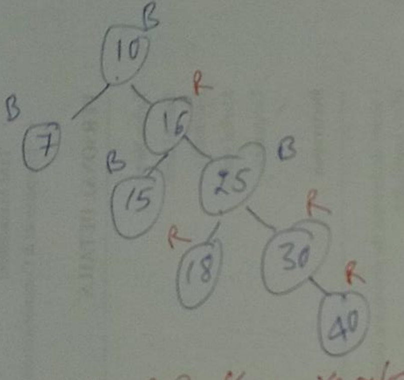
Right Rotate

(ii)

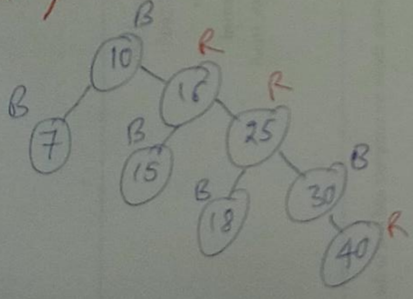


Left Rotate

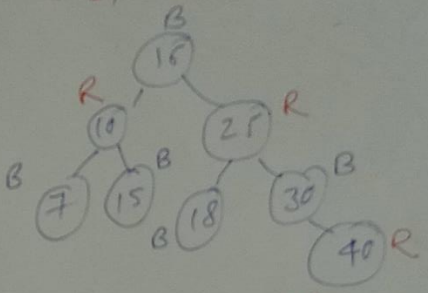
(7)



(i) Sibling is Red then recolor



(ii) Now Parents Parent Sibling is not Red, so do rotation



↗ Now recolor it

[THIS IS YOUR FINAL RED BLACK TREE]

UNORDERED MAP

EXAMPLE:

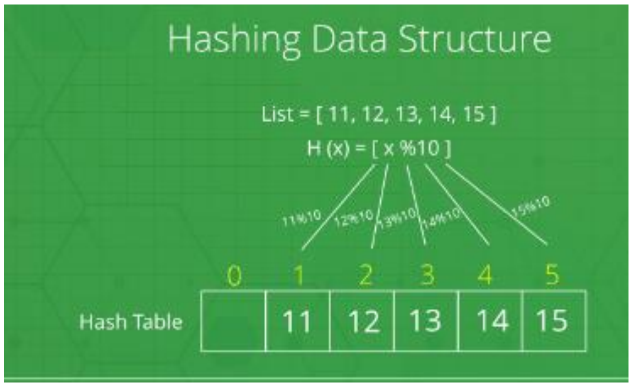
```
#include<iostream>

using namespace std;

#include<unordered_map>

#include<iterator>

int main()

{

    unordered_map<string,int> up;

    up.insert({"a",65});

    up.insert({"z",90});

    up.insert({"e",73});

    up.insert({"k",90});

    up.insert({"l",69});

    up.insert({"x",73});


   for(auto x:up)

    {

        cout<<x.first<<","<<x.second<<endl;

    }

    //or

    /*for(auto it:up.begin();it!=up.end();it++)

    {

        cout<<it->first<<","<<it->second<<endl;

    }*/
string k="l";
if(up.find(k)!=up.end())
{
    auto t=up.find(k);
    cout<<"keys is "<<t->first<<" value is= "<<t->second;
}
else
{
    cout<<"not found";
}

cout<<"\n"<<up.size()<<endl;
    up.clear();
    cout<<up.max_size()<<endl;
}
```

HASH TECHNIQUE

What is Hashing

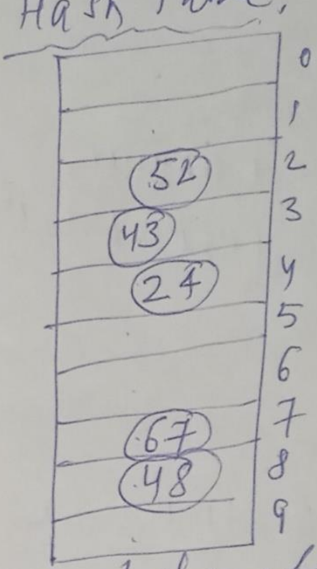
Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.

Let a hash function H(x) maps the value **x** at the index x%10 in an Array. For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.



HASHING - MAPPING TECHNIQUE
(Storing & Retrieving Data in O(1) times)

Hash Table:



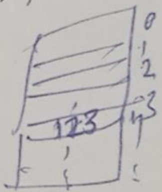
1. Search Key
2. Hash Table
3. Hash Function
   - → k mod 10
   - → k mod n
   - → Mid square
   - → Folding Method

EX: (24, 52, 91, 67, 48, 43) n = No. of elements

INSERT/DELETE decides position of Hash Tabl

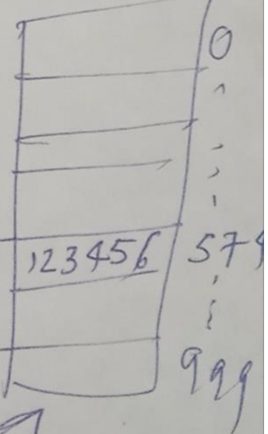
1. 24 mod 10 → 4 O(1)
2. 52 mod 10 → 2
   ⋮

Similary for Search



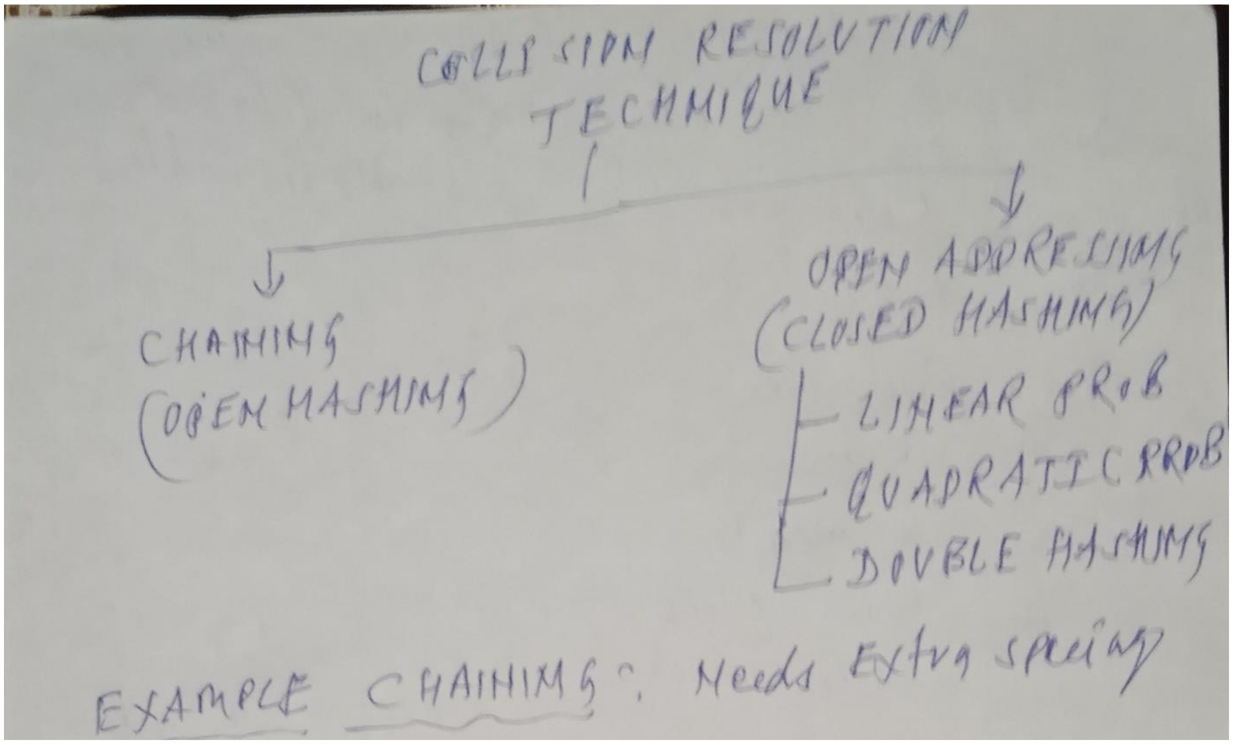
Mid Square EX:- 123

$2^2 = 4$ - Pos



Folding Method EX: 123456

$$\begin{array}{r} 123 \\ +\ 456 \\ \hline 5\cancel{7}9 \end{array}$$ - position



COLLISION RESOLUTION TECHNIQUE

- CHAINING (OPEN HASHING)
- OPEN ADDRESSING (CLOSED HASHING)
  - LINEAR PROB
  - QUADRATIC PROB
  - DOUBLE HASHING

EXAMPLE CHAINING : Needs Extra spacing

COLLISION TECHNIQUE

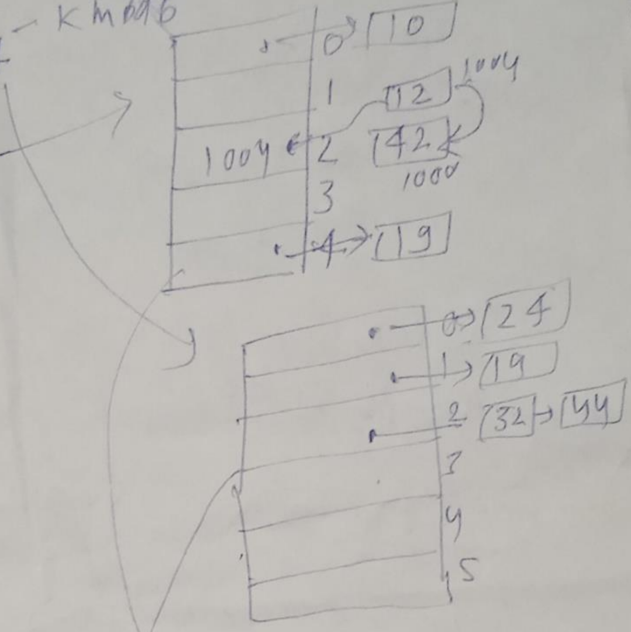
* CHAINING (OPEN ~~HA~~ HASHING)

KEYS: 24, 19, 32, 44

KEYS: 42, 19, 10, 12

K mod 5

1. 42 % 5 = 2
2. 19 % 5 = 4
3. 10 % 5 = 2
4. 12 % 5 = 2 *



Advantages
1. INSERT O(1)
2. DELETION EASY

Disadvantages
1. SEARCHING - O(n)
2. DELETION — O(n)
3. EXTRA SPACE

LOAD FACTOR

$\alpha = \frac{\text{TOTAL KEYS}}{\text{TOTAL SLOTS}}$

$\boxed{\alpha = \frac{4}{5}}$

LOAD FACTOR

$\alpha = \frac{4}{6}$

$\boxed{\alpha = \frac{2}{3}}$

* OPEN ADDRESSING (CLOSED HASHING) (2)

i = collision no/probe no/ Attempt no

(1) LINEAR PROBING

KEYS: 43, 135, 72, 23, 99, 19, 82

$h(k)$: K mod 10

$h(k,i)$: $(h(k)+i)$ mod 10

. 43 mod 10, . 135 mod 10,

. 72 mod 10, . 23 mod 10 *

COLLISION ← $h(k)=3$

$h(k,i) = (3+1)$ mod 10

= 4 mod 10

= 4

. 99 mod 10; 19 mod 10 *

. 82 mod 10 * collision

$h(K)=2$

$h(k,i)$: $(2+1)$ mod 10   i=1 prob

= 2 + 2 mod 10   i=2 prob

i=3 prob

= $\boxed{2+3}$ mod 10   i=4 prob

= 5 mod 10

= $\boxed{6 \text{ mod } 10}$ i=6 prob.

19	0
	1
72	2
43	3
23	4
135	5
82	6
	7
	8
99	9

Adv: NO EXTRA SPACE

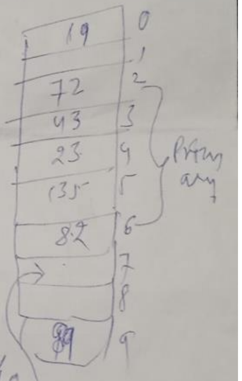
DIS: (1) search time O(n) worst case { Avg / Best → O(1)

(2) Deletion difficult — if some values deleted in middle means not available in this case we use extra keyword/space to continue delete

(3) Primary clustring

(4) Secondary clustring

Primary clustering:
↳ group of elements



→ Tendency or probability of element will increase at some particular location

For any position
Probability = 1/10

probability EX: 92 = 6/10

→ searching time increases

secondary clustering
follow the same prob sequence
1st → 1, 2, 3, 4, 5, 6 . . .
2nd → 1, 2, 3, 4, 5, 6 . . .

(4) QUADRATIC HASHING

$h(k) = k \bmod 10$

$h'(h(k) + i^2) \bmod 10$

Keys : 42, 16, 91, 33, 18, 27, 36, 62

36 mod 10

$h(k) = 6$

$h' = (h(k) + i^2) \bmod 10$

$= 6 + (1)^2 \bmod 10$

$= 7 \bmod 10$

$h' = 7$ already elements

now $= h(k) + (i)^2 \bmod 10$

$= 6 + 2^2 \bmod 10$

$= 10 \bmod 10$

$h(k) = 62 \bmod 10 = 2$

3 $= 2 + 1^2 \bmod 10$

6 $= 2 + 2^2 \bmod 10$

1 $= 2 + 3^2 \bmod 10$

8 $= 2 + 4^2 \bmod 10$

7 $= 2 + (5)^2 \bmod 10$

⋮

loop prob

36	0
91	1
42	2
33	3
	4
	5
16	6
27	7
18	8
	9
	10

* Adv :- No Extra space

Primary clustering Resolved

* DIS : search O(n)

: secondary clust

: No Guarantee of finding slot

: insert O(n)

: Delete

DOUBLE HASHING (Two Hash())

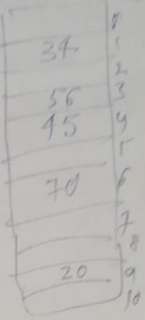
$h_1(k) = k \bmod 11$

$h_2(k) = 8 - (k \bmod 8)$

$= (h_1(k) + i \cdot h_2(k)) \bmod 11$

{here i is not fix}

For uniform distribution



Keys = 20, 34, 45, 70, 56

$h_1(k) =$ 20 mod 11

= 34 mod 11

45 mod 11

= 1 occupied K = 45

$h(k) =$ 1 occupied

$h_2(k) = 8 - (45 \bmod 8)$

$= 8 - 5$

$= 3$

uniform distribution key

$= h_1(k) + i \cdot h_2(k) \bmod 11$

$1 + 1 \cdot 3$

=

$\boxed{h_1'(k) = 4}$

$h(k) = 70 \bmod 11 = 4$

$h_2(k) = 8 - (70 \bmod 8)$

$= 8 - 6$

$= 2$

$(h_1 k + i \cdot h_2 k) \bmod 11$

$= (4 + 2 \cdot 1) \% 11$

$h_1'(k) = 6$

Now 56 mod 11

= 1 already

$h_2(k) = 8 - 56 \bmod 8$

$= 8 - 0$

$= 8$

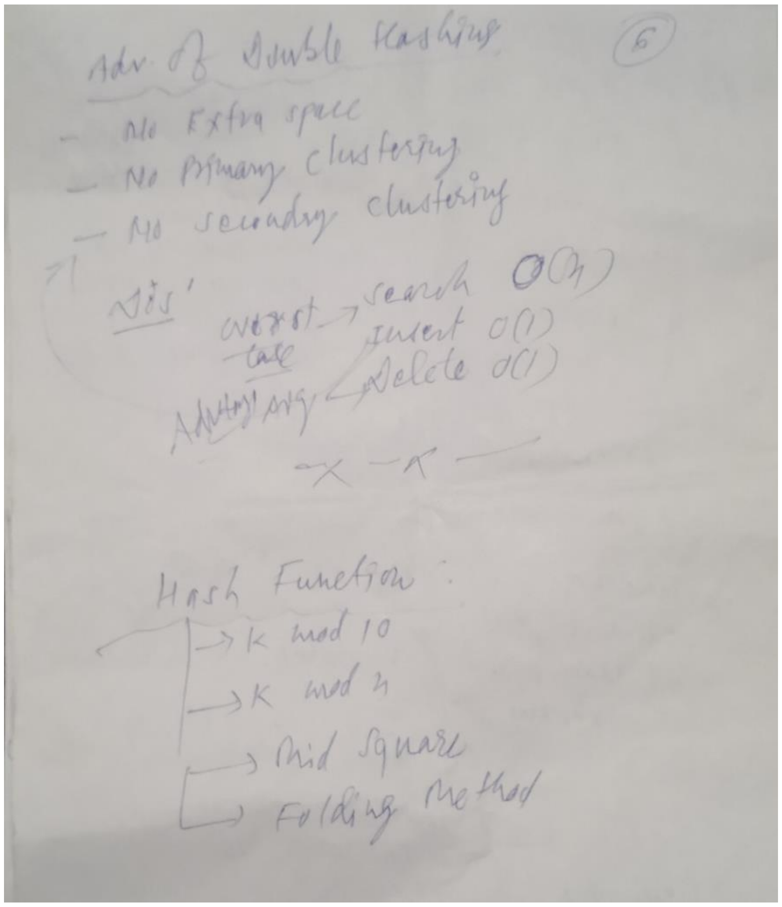
$= 1 + 8 \bmod 11$

$= 9 \bmod 11$

$= 1 + (16 \bmod 11)$

$= 1 + 24 \bmod 11$

$= \boxed{13}$ ✓



Adv. of Double Hashing (6)

- No Extra space
- No Primary clustering
- No secondary clustering

Dis'

worst case → search O(1)

Insert O(1)

Admtsy avg ← Delete O(1)

—x—x—

Hash Function:

→ K mod 10

→ K mod n

→ Mid Square

→ Folding Method

SET

- **SORTED**
- **Ologn(Avg) and O(n)(worst)**
- **DIJKSTRA ALGO(BST)**

UNORDER_SET

- **UNSORTED**
- **O(1) (Avg)and O(n)(worst)**
- **Hash Technique**

Example of SET:

```
#include<bits/stdc++.h>

using namespace std;

int main()

{

   set<int> st;

   st.insert(101);

    st.insert(12);

    st.insert(1);

    st.insert(10);

   set<int>::iterator it;

   for(it=st.begin();it!=st.end();it++)

   {

       cout<<*it<<endl;

   }


/*st.erase(1);
```

```
    cout<<"after erasing\n";
   for(it=st.begin();it!=st.end();it++)
    {
        cout<<*it<<endl;
    }*/

st.erase(st.begin(),st.find(12));
      cout<<"after erasing\n";
    for(it=st.begin();it!=st.end();it++)
    {
        cout<<*it<<endl;
    }
cout<<st.size()<<endl;

    cout<<st.max_size()<<endl;
    st.clear();
set <int> s{23,4,6,8,5,90,45};
auto y=s.find(8);//find return iterator//O(logn)
    cout<<"found="<<*y<<"\n";
    if(s.count(452))//O(n)
    {
        cout<<"found";
    }
    else
   {
      cout<<"not found";
   }

}
```

Example of unordered_set

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    unordered_set <int> s{23,4,6,8,5,90,45};
 s.insert(21);
 auto k=0;
    for(auto &p:s)
    {
        cout<<p<<endl;
    }
    cout<<"iterator\n";
    set<int>::iterator t;

    /*for(t=s.begin();t!=s.end();t++)
    {

    cout<<*t<<endl;
    }*/
    auto y=s.find(8);//find return iterator//O(logn)
    cout<<"found="<<*y<<"\n";
    if(s.count(452))//O(n)
    {
        cout<<"found";
    }
    else
    {
        cout<<"not found";
    }
}
```