JUNIORS 2

SFE ACPC CLUB
CODE YOUR WAY

Linear Data Structures

Session 3

To declare the libraries of everything:

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

Pairs

This type couples together a pair of values, which may be of different types and each value can be accessed separately from the other. (int, int) , (string, double) , etc…

Then to declare a pair we do the following.

We can create a pair of pairs and so on.

```
//To Declare a pair
//pair<type1, type2> variableName;
pair<int, int> p1;
pair<string, int> p2;
pair< pair<int, int>, string> p3; //And so on
```

To assign values to a pair we use the following function

make_pair(val1, val2)	Creates a pair with the given values (val1 and val2)

We can use {val1 , val2} as a shortcut instead of make_pair(val1 ,val2).

```
pair<int, int> p1 = make_pair(1, 52);
pair<string, int> p1 = make_pair(“test”, 52);
pair< pair<int, int>, string> p3 = make_pair(make_pair(1,1), “test”);  //And so on
//-------------------------------------------------
pair<int, int> p1 = {1, 52};
pair<string, int> p1 = {“test”, 52};
pair< pair<int, int>, string> p3 = { {1,1}, “test”};  //And so on
```

Now if we want to access the values inside the pair

Note that if (myPair.first) is a pair and we want to access it's values we use these following notations
myPair.first.first
myPair.first.second
and so on if the value is still a pair.

```
//To Access values inside a pair
//pair.first & pair.second are used
pair<string, int> myPair;
myPair = ("testing", 98};
cout << "first = " << myPair.first << " & second = " << myPair.second << endl;
//Output:first = testing & second = 98
```

STLs (Linear Structures)

Intro into STL (Standard Template Library)

To get familiar with STL we can discuss one of it's basic elements which is Containers.
You're most likely to have encountered a container before which is the array.
Despite having fast access time but sometimes we need more control or flexibility.
For example:
-we can't resize an array meaning that we can't add a new element to it after it was initialized.

```
int arr[5] = {0, 1, 2, 3, 4};
Arr[5] = 5; //Runtime Error
```

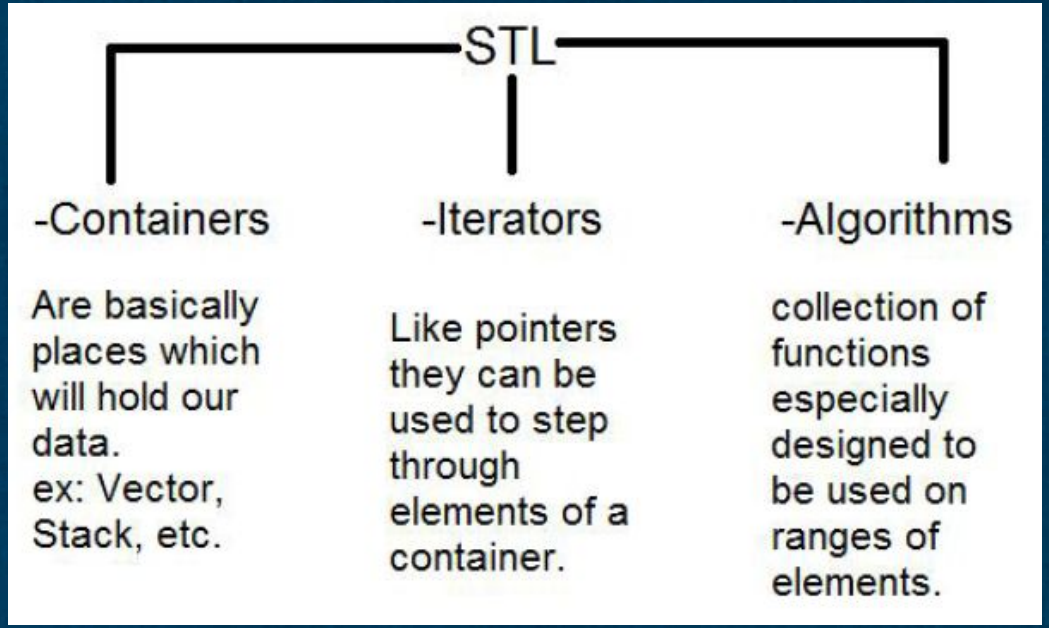
-you can't determine which element was added last which sometime is helpful.
So as the result of the previous problems we came up with new containers that fit in specific positions having properties that the regular array doesn't have.

What is a Template?

A template is simply a piece of code written independent of any data type, meaning that this piece of code could be applied on (Integers, Characters, Other template, …, etc.).
Just for an example the sort(), it can be used to sort any type of data.

```
//Sort function used to sort integers and chars
Int intArray[5] = {4, 5, 1, 3, 2};
Char charArray[5] = {‘e’, ‘a’, ‘d’, ‘b’, ‘c’};
sort(intArray, intArray+5); //Now intArray = {1, 2, 3, 4, 5}
sort(charArray, charArray+5); //And charArray = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’};
```

But keep in mind that the sort() is not a template.
STL consists of 3 basic elements:



Containers (Templates)

While explaining the containers we will encounter the iterators and algorithms.
There are several types of containers as shown below.

Sequential Containers	Container Adaptors	Associative Containers
Array	Stack	Priority Queue
Vector	Queue	Set/Multiset
Deque		Map/Multimap
List		Unordered Map

-Sequential Containers (Linear data structure):
Implement data structures which can be accessed in a sequential manner (array[0], array[1] and so on).

-Container Adaptors (Linear data structure):
They are basically modified version of the sequential containers.

-Associative Containers (Non-Linear data structure):
The data in them is sorted always, which means they can be searched quickly (O(log n) complexity).

Vector

A Vector is basically a resizable array, meaning that we don’t have to give it a size at declaration.

Improvements:

-Size is not fixed and can be changed during runtime
-We can add/delete elements to/from it.

Notes:

-If a new element is added it’s added after the last element of the vector

Important functions:

Assume vector<int> v; is declared

v.at(i) or v[i]	Return the element at the index i
v.push_back(x)	X is added to end of the vector
v.pop_back()	Deletes the last element
v.size()	Returns the size of the vector (the number of elements)
v.empty()	Returns True if empty and False otherwise
v.clear()	Empties the vector (new size = 0)
v.swap(v2)	Swap the elements of v and v2

```
vector<int> v;
cout << “Start - Empty? “ << v.empty() << endl; //Start - Empty? 1
Cout << “Start - Size = “ << v.size() << endl; //Start - Size = 0
//Adding 5 elements to the vector
for(int i=1; i<=5; i++)
    v.push_back(i*i); //Has 1 4 9 16 25
cout << “Empty? “ << v.empty() << endl; //Empty? 0
cout << “Size = “ << v.size() << endl; //Size = 5
v.pop_back(); //Deletes 25
v.pop_back(); //Deletes 16
cout << “After pop - Empty? “ << v.empty() << endl; //After pop - Empty? 0
Cout << “After pop - Size = “ << v.size() << endl; //After pop - Size = 3
```

```
//Showing the elements of the vector
cout << “Using [] operator.” << endl; //Using [] operator.
for(int i=0; i<v.size(); i++)
    cout << v[i] << “ “;              //1 4 9
cout << endl;
cout << “Using .at()” << endl; //Using .at()
for(int i=0; i<v.size(); i++)
    cout << v.at(i) << “ “;          //1 4 9
Cout << endl;
v.clear(); //Clearing the vector
cout << “After clear() - Empty? “ << v.empty() << endl; //After clear() -
Empty? 1
Cout << “After clear() - Size = “ <<v.size()<< endl; //After clear() - Size = 0
```

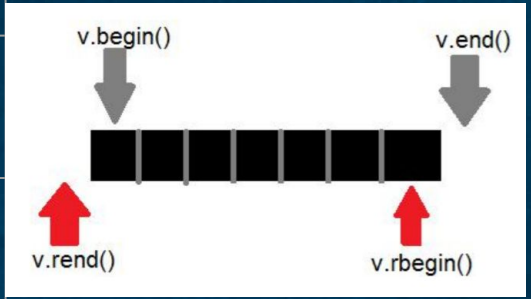
Iterators

Iterators are basically like pointers meaning that they point at a specific location but keep in mind that they have some differences from a pointer.
The STL has Iterators which are custom made for each container in different ways and these ways are defined in the template of the container itself.

Note:
-assuming that x is an iterator
then *x gives the data stored at the place which x points to.

The Following are the most important iterators

v.begin()	Returns an iterator at the beginning of the vector
v.end()	Returns an iterator at the first place after the end of the vector
v.rbegin()	Returns an iterator at the end of the vector
v.rend()	Return an iterator at the first place before the beginning of the vector



Notes

-v.begin() & v.end() are called forward iterators which means they are used to go from element with index 0 to element (n-1).

-v.rbegin() & v.rend() are called reverse iterators and used to go through the elements from index (n-1) to element 0.

As stated before we can use the iterators to go through the elements of a container. This is how.

```
//Defining a forward iterator -> ContainerName<Data Type>::iterator variableName;
//Defining a reverse iterator -> ContainerName<Data Type>::reverse_iterator variableName;
//IMP: The iterator must match the type of the container

vector<int> myVec;
for(int i=1; i<=5; i++)
     myVec.push_back(i);   //myVec now has 1 2 3 4 5
vector<int>::iterator itr = myVec.begin();
//Using iterator to show the element of the vector
for(itr; itr!=myVec.end(); itr++)
cout << *itr << " ";   //Output:1 2 3 4 5
//Using reverse iterator to show the element of the vector
vector<int>::reverse_iterator itr2 = myVec.rbegin();
for(itr2; itr2!=myVec.rend(); itr2++)
cout << *itr2 << " ";   //Output:5 4 3 2 1
```

Notes

All the mentioned functions have Constant complexity “O(1)” but there is special case with v.push_back().

As the vector is dynamic and changes it’s size every once in a while sometimes it needs a bigger space so it moves all of it’s elements to another place when that happens the complexity is linear O(n) where n is the number of moved elements.

Algorithms

Which are functions that operate on a range of elements in a container, Here are some of the most used in competitive programming.

- These functions are under <algorithm> header (this can be neglected if you use the <bits/stdc++.h> header)

Finding Elements

- Note that start & end are iterators
- The Complexity for each of these functions is linear O(n)

find(start, end, value)	Returns an iterator to the place of the first occurrence of the value *returns an iterator to end if the value is not found
find_if(start, end, function)	An iterator to the first element in the range for which the function returns true to. *returns an iterator to end if the value is not found
count(start, end, value)	Returns the number of how many times value was found in that range
count_if(start, end, function)	Returns the number of values that satisfy the function

```
//using find with array and pointer
int myints[] = {10, 20, 30, 40};
int *p;
p = find(myints, myints+4, 30);
if(p != myints+4)
      cout << “Element found in myints: “ << *p << endl; //Element found in myints: 30
else
      cout << “Element not found in myints\n”
//using find with vector and iterator
//New way to initialize a vector
vector<int> myVector(myints, myints+4);
vector<int>::iterator it;
it = find(myVector.begin(), myVector.end(), 50);
if(it != myVector.end())
      cout << “Element found in myVector: “ << *it << endl;
else
      cout << “Element not found in myVector\n” //Element not found in myVector
```

```cpp
bool IsOdd (int i) {
  if(i % 2)
    return true;
  else
    return false;
}

bool Large(int i){
    return (i > 100);
}
```

```cpp
int main () {

  vector<int> myvector;
  myvector.push_back(10);
  myvector.push_back(3);
  myvector.push_back(40);
  myvector.push_back(57);

  vector<int>::iterator it = find_if(myvector.begin(), myvector.end(), IsOdd);
  if(it != myvector.end())
    cout << "The first odd value is " << *it << '\n';
  else
    cout << "No Odd values found" << endl;

  it = find_if(myvector.begin(), myvector. end(), Large);
  if(it != myvector.end())
    cout << "element greater than 100 is " << *it << '\n';
  else
    cout << "No elements greater than 100" << endl;

  return 0;
}
```

```
The first odd value is 3
No elements greater than 100
```

```cpp
int arr[11] = {1,6,9,4,1,3,1,7,6,2,1};
vector<int> myVec(arr,arr+11);
int numOfOnesFound = count(myVec.begin(),myVec.end(),1);
cout << "1 was found " << numOfOnesFound << " times" << endl << endl;

int oddNumbers = count_if(myVec.begin(),myVec.end(),IsOdd);
cout << "There is " << oddNumbers << " Odd values in myVec" << endl;
```

```
1 was found 4 times

There is 7 Odd values in myVec
```

Elements Modification and Reorder

remove(start, end, value)	Removes every element that is equal to value in the given range • Returns an iterator to new end • O(n)
sort(start, end, optional function)	-Sorts the element according to the function -If there is no function elements are sorted in ascending manner • O(n log n)
Reverse(start, end)	Reverses the order of the elements in the given range • O(n / 2) ?!

```cpp
int arr[11] = {1,6,9,4,1,3,1,7,6,2,1};
vector<int> myVec(arr,arr+11);
for(int i = 0; i < 11; i++)
    cout << myVec[i] << " ";
cout << endl << endl;
vector<int>::iterator newEnd = remove(myVec.begin(),myVec.end(),1);
vector<int>::iterator itr = myVec.begin();
for(itr; itr != newEnd; itr++)
    cout << *itr << " ";
```

```
1 6 9 4 1 3 1 7 6 2 1

6 9 4 3 7 6 2
```

```
bool largerOddFirst(int i, int j)
{
    if(i%2 && !(j%2))
        return true;
    else if(i%2 && j%2)
        return i > j;
    else if(!(i%2) && !(j%2))
        return i > j;
    else
        return false;
}
int main ()
{
    int arr[10] = {1,7,10,2,5,3,6,9,4,8};
    vector<int> myVec(arr,arr+10);
    for(int i = 0; i < 10; i++)
        cout << myVec[i] << " ";
```

```
1 7 10 2 5 3 6 9 4 8

After sort
1 2 3 4 5 6 7 8 9 10

After sorting with custom function
9 7 5 3 1 10 8 6 4 2

Reversing Elements
2 4 6 8 10 1 3 5 7 9
```

```
    cout << endl << endl << "After sort" << endl;
    sort(myVec.begin(), myVec.end());
    for(int i = 0; i < 10; i++)
        cout << myVec[i] << " ";

    cout << endl << endl << "After sorting with custom function" << endl;
    sort(myVec.begin(), myVec.end(),largerOddFirst);
    for(int i = 0; i < 10; i++)
        cout << myVec[i] << " ";
    cout << endl << endl << "Reversing Elements" << endl;
    reverse(myVec.begin(), myVec.end());
    for(int i = 0; i < 10; i++)
        cout << myVec[i] << " ";
    cout << endl;
    return 0;
}
```

Minimum and Maximum Elements

min_element(start, end)	Returns an iterator at the position of the minimum element
max_element(start, end)	Returns an iterator at the position of the maximum element
minmax_element(start,end)	Returns a pair of iterators 1 st to minimum element 2 nd to maximum element

- Complexity is O(n)

```
int arr[10] = {20,40,30,80,60,10,100,20,135,270};
vector<int> myVec(arr,arr+10);

int minimum = *min_element(myVec.begin(), myVec.end());
int maxiimum = *max_element(myVec.begin(), myVec.end());

pair< vector<int>::iterator , vector<int>::iterator > minAndMax;
minAndMax = minmax_element(myVec.begin(),myVec.end());

cout << "Minimum = " << minimum << endl;
cout << "Maximum = " << maxiimum << endl;
cout << "Minimum = " << *minAndMax.first << " - And max =  " << *minAndMax.second <<  endl
```

```
Minimum = 10
Maximum = 270
Minimum = 10 - And max =  270
```

Sort based functions

- YOU CAN'T USE THEM ON UNSORTED RANGE!

lower_bound(start,end,value)	Returns an iterator to the first element that >= value • If no such element returns vec.end()
upper_bound(start,end,value)	Returns an iterator to the first element that > value • If no such element returns vec.end()
equal_range(start,end,value)	Returns pair of iterators that are lower and upper bound
binary_search(start,end,value)	Returns True if value exists and False otherwise

- Complexity is O(log n)

Notes

-We can use lower and upper to determine the number of occurrences of value in the range and the index of the wanted value

lower_bound() – vec.begin() = index of lower bound value

upper_bound() – vec.begin() = index of upper bound value

- Why is it better to use this method to determine how many time the value is found instead of count() ?
  “ON SORTED RANGE”.
  Look for the complexity of each ;)

```
int myints[] = {10,20,30,30,20,10,10,20};
vector<int> v(myints,myints+8);                 /// 10 20 30 30 20 10 10 20

sort (v.begin(), v.end());                      /// 10 10 10 20 20 20 30 30

vector<int>::iterator low,up;
low = lower_bound (v.begin(), v.end(), 20);///          ^
up  = upper_bound (v.begin(), v.end(), 20);///                ^

if(low != v.end() && up != v.end()){
cout << "lower_bound at position " << (low- v.begin()) << '\n';
cout << "upper_bound at position " << (up - v.begin()) << '\n';

cout << "20 is found " << up - low <<  " times" << endl;
}
else
    cout << "20 is not found" << endl;
```

```
///Using Equal Range
cout << endl << "Using Equal_range() " << endl;
pair<vector<int>::iterator,vector<int>::iterator> bounds;
bounds = equal_range(v.begin(),v.end(),20);
if(bounds.first != v.end() && bounds.second != v.end()){
cout << "bounds.first  at " << (bounds.first- v.begin()) << '\n';
cout << "bounds.second at " << (bounds.second - v.begin()) << '\n';

cout << "20 is found " << bounds.second - bounds.first <<  " times" << endl;
}
else
    cout << "20 is not found" << endl;

///Binary Search
bool check = binary_search(v.begin(),v.end(),40);
cout << endl << "40 is " << ((check)? "found" : "not found") << endl;
```

List

Lists are sequence containers that allow constant time insert and erase operations anywhere within the sequence, and iteration in both directions.

- Implemented as Doubly-Linked Lists

Improvements:

O(1) insertion and deletion from middle once iterator for that position is obtained.

Disadvantage:

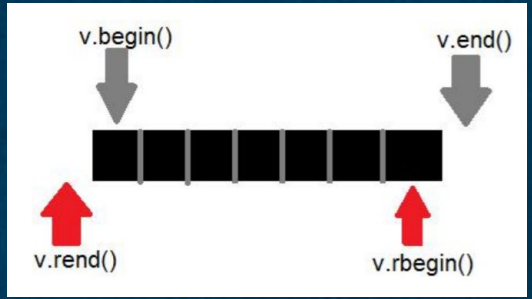
Unlike arrays and vectors access time is too slow as there is no direct access to elements i.e.(list[10] is not valid).So to get an element we have to iterate through all the list.

- Access time is linear O(n).
- Like the Vector, a List has it’s own iterators and functions.

Iterators

- They behave the same way as in a vector

v.begin()	Returns an iterator at the beginning of the vector
v.end()	Returns an iterator at the first place after the end of the vector
v.rbegin()	Returns an iterator at the end of the vector
v.rend()	Return an iterator at the first place before the beginning of the vector



Functions

- Assuming list l is defined

l.push_front(x)	Adds x to beginning of list
l.pop_front()	Deletes the first element
l.push_back(x)	Adds x to the end of the list
l.pop_back()	Deletes the last element
l.size()	Returns the size of the list
l.clear()	Deletes all elements in the list
l.swap(l2)	Swaps elements of l and l2
l.empty()	Returns true if the list is empty false otherwise

These the functions behave the same way as in a vector but there is some extra function with the list

l.insert(position, val)	Position is an iterator Adds val at the given position • O(1) ?
l.insert(position, n, val)	Adds val n-times at the given position • O(n of added elements)
l.erase(position)	Deletes element at given position • O(1)
l.erase(pos1, pos2)	Deletes all elements from pos1 to (pos2-1) • O(n of deleted elements)
l.remove(x)	Deletes all elements with value == x

```cpp
list<int> mylist;
list<int>::iterator it;

/// set some initial values:
for (int i=1; i<=5; ++i) mylist.push_back(i); /// 1 2 3 4 5

it = mylist.begin();
++it;        /// it points now to number 2            ^

mylist.insert (it,10);                          /// 1 10 2 3 4 5

/// "it" still points to number 2                      ^
mylist.insert (it,2,20);                        /// 1 10 20 20 2 3 4 5


cout << "mylist contains:";
for (it=mylist.begin(); it!=mylist.end(); ++it)
  cout << ' ' << *it;
///Output is
///mylist contains: 1 10 20 20 2 3 4 5
```

```
list<int> mylist;
list<int>::iterator it1,it2;
/// set some values:
for (int i=1; i<10; ++i) mylist.push_back(i*10);
                                /// 10 20 30 40 50 60 70 80 90
it1 = it2 = mylist.begin(); /// ^^
advance(it2,6);             /// ^                 ^
++it1;                      ///    ^              ^

it1 = mylist.erase (it1);   /// 10 30 40 50 60 70 80 90
                            ///    ^           ^

it2 = mylist.erase (it2);   /// 10 30 40 50 60 80 90
                            ///    ^           ^

++it1;                      ///       ^        ^
--it2;                      ///       ^     ^

mylist.erase (it1,it2);     /// 10 30 60 80 90
                            ///       ^

cout << "mylist contains:";
for (it1=mylist.begin(); it1!=mylist.end(); ++it1)
  cout << ' ' << *it1;
///mylist contains: 10 30 60 80 90
```

Stack

Stacks are a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from one end of the container.

- To declare a stack:

```
Stack <data_type> name;
Ex:
    stack<int> st1;
    stack<char> st2;
    stack<string> st3;
```

And so on.

SFE ACPC CLUB
CODE YOUR WAY

Important built-in functions of Stack

empty	Test whether stack is empty	O(1)
size	Return size of the stack	O(1)
top	Access top element of the stack	O(1)
push	Insert an element on the top of the stack	O(1)
pop	Remove the top element of the stack	O(1)
swap	Swap contents of two stacks	O(1)

```cpp
int main()
{
    stack <int> st1,st2;
    st1.push(1);
    st1.push(2);
    st1.push(3);
    cout<<"Size of Stack 1 :"<<st1.size()<<endl;
    st2.push(4);
    st2.push(5);
    st2.push(6);
    cout<<"Size of Stack 2 :"<<st1.size()<<endl;
    st1.swap(st2);
    while(!st1.empty()) {
        cout<<st1.top()<<" "<<st2.top()<<endl;
        st1.pop();
        st2.pop(); }
    return 0;
}
```

```
Size of Stack 1 :3
Size of Stack 2 :3
6 3
5 2
4 1
```

Queue

Queues are a type of container adaptor, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other.

- To declare a queue:

```
queue <data_type> name;
```

Ex:

```
queue<int> qu1;
queue<char> qu2;
queue<string> qu3;
```

And so on.

Important built-in functions of Queue

empty	Test whether queue is empty	O(1)
size	Return size of the queue	O(1)
front	Access first element of the queue	O(1)
back	Access last element of the queue	O(1)
push	Insert an element on the back of the queue	O(1)
pop	Swap contents of two queues	O(1)

```cpp
int main()
{
    queue<int> q;
    q.push(1);
    q.push(2);
    q.push(3);
    cout<<"The front of the queue:"<<q.front()<<endl;
    cout<<"The back of the queue:"<<q.back()<<endl;
    q.front()=20;
    q.back()=100;
    while(!q.empty())
    {
        cout<<q.front()<<endl;
        q.pop();
    }
    return 0;
}
```

```
The front of the queue:1
The back of the queue:3
20
2
100
```

Deque

Deques are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).

- To declare a deque:

```
deque <data_type> name;
```

Ex:

```
deque<int> dqu1;
deque<char> dqu2;
deque<string> dqu3;
```

And so on.

Important features and built-in functions of the Deque is the same of the List

Except

- List has “remove()” function
- Deque has “operator[]” feature.

```cpp
int main()
{
    deque<int> dq;
    dq.push_back(4);
    dq.push_back(5);
    dq.push_back(6);
    dq.push_front(3);
    dq.push_front(2);
    dq.push_front(1);
    for(int i=0;i<6;++i)
        cout<<dq[i]<<endl;
    cout<<endl;
    while(!dq.empty()) {
        cout<<dq.front()<<endl;
        dq.pop_front(); }
    return 0;
}
```

```
1
2
3
4
5
6

1
2
3
4
5
6
```

Deques vs Vectors

They provide a functionality similar to vectors, but with efficient insertion and deletion of elements also at the beginning of the sequence, and not only at its end.
Performance of addition and deletion at end for vector is better than deque.
Performance of random access operations i.e. operator [] and at() function in deque will be little slower than vector.

Deques vs Lists

For operations that involve frequent insertion or removals of elements at positions other than the beginning or the end, deques perform worse and have less consistent iterators and references than lists

Assignment: https://a2oj.com/contest?ID=35189