Containers in STL

View this page efficiently here:

Containers in STL

Vectors

Vectors are 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.

Using The Inbuilt vector Data-Structure in STL

Syntax: `vector <int> vector_name;`

Vectors provides 7 basic operations for interaction

1. `begin()-` Return iterator to beginning
2. `end()` - Return iterator to end
3. `push_back()` - Add element at the end
4. `pop_back()` - Delete last element
5. `erase()` - Removes a range of elements ([first,last))
6. `clear()` - Clear contents
7. `empty()` - Returns a boolean value depending whether vector is empty or not

```
int main() {

  // Declearing a vector
  vector <int> vec;

  // inserting values in a vector
  vec.push_back(1);
  vec.push_back(2);
  vec.push_back(3);
  vec.push_back(4);
  vec.push_back(5);

  // printing the 1st value of vector unging begin()
  cout<<"1st Element: "<<*vec.begin()<<endl;

  // printing the last value of vector unging begin()
  cout<<"last Element: "<<*vec.end()<<endl;

  // Checking if vector is empty
  cout<<"Is vector empty: "<<vec.empty()<<endl;

  // Clear contents of vector
  vec.clear();

  // Checking if vector is empty
  cout<<"Is vector empty: "<<vec.empty()<<endl;
}
```

Queue

Queue is an abstract linear data structure in which operations are performed in a particular manner

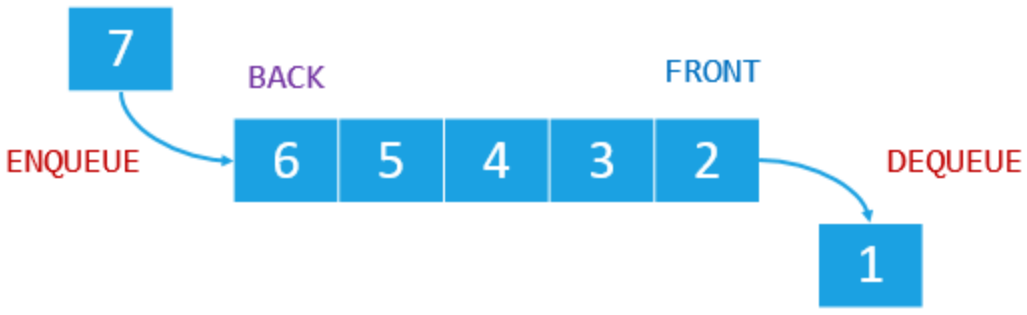
The Manner in which operations are performed in queue is First in First Out (FIFO) or first come first served.

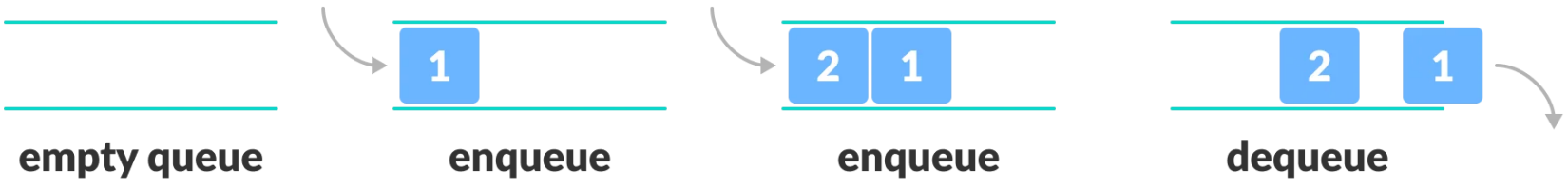
A queue can be implemented using array as well as using Linked List.

Stacks provides 3 basic operations for interaction

1. `enqueue()-` Adds an element at the end of queue
2. `dequeue()` - Removes an element from from to the queue

3. `isEmpty()` - Check if queue is empty or not.
4. `top()` - Return the element present at the front of queue





Copy of Time complexity for Queue

Aa Operation	Complexity
Untitled	
Enque (Insertion)	O(1)
Deque (Deletion)	O(1)
Top (Get Front)	O(1)
Searching	O(N)

Using Inbuilt Queue Data Structure from STL

Syntax: `queue <int> queue_name;`

Some functions of Queue:

- `push()` Inserts an element (at end) `O(1)`
- `pop()` Removes/delete an element (from front) `O(1)`
- `front()` Acess the element at the front of queue `O(1)`
- `empty()` Checks if queue is empty or not `O(1)`
- `size()` Returns the size of queue `O(1)`

```
#include <bits/stdc++.h>
using namespace std;

int main() {

  queue <int> que_name; // Decleare a queue

  for (int i = 0; i < 5; i++) {//This loop takes input and adds it to queue (at end)
    int temp; cin >> temp;
    que_name.push(temp);
  }

//This prints the element of queue and clears the queue at the end of loop
  while (!que_name.empty()) {
    cout << que_name.front() << " ";
    que_name.pop();
  }

  return 0;
}
```

Stack

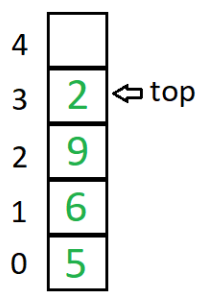
What is Stack?

Stack is an linear data structure in which operations are performed in a particular manner.

operations are performed on `(LIFO) Last in first out` or `(FILO) First in Last out` manner

A new data element is stored by pushing it on the top of the stack. And an data element is retrieved by popping the top element off the stack and returning it.

An stack can be implemented using array as well as linked list but using a linked list is more efficient due to its dynamic nature.



Stack

In the above image, although item 2 was kept last, it was removed first. This is exactly how the LIFO (Last In First Out) Principle works.

Copy of Time-Complexity

Aa Operation	Big O	Explanation
Insertion (push()) at top	O(1)	We always have to insert at top of stack which is a one step process
Deletion (pop()) at top	O(1)	We always have to remove from the top of stack which is a one step process
Searching	O(N)	We have to iterate from the top most element to the Nth element
Accessing a particular element	O(N)	We have to iterate from the top most element to the Nth element

Using The Inbuilt Stack Data-Structure in STL

Syntax: `stack <string> stack_name;`

Some functions of stack

- `push()` Insert an element at the top of stack `O(1)`
- `pop()` removes a element from the top of stack `O(1)`
- `top()` Acess the top element of the stack `O(1)`
- `empty()` Returns whether the stack is empty or not `O(1)`
- `size()` Returns size of stack `O(1)`

```
#include <bits/stdc++.h>
using namespace std;

int main() {
  stack <int> stack_name; // Decleare a stack

  for (int i = 0; i < 5; i++) { //This loop takes input and adds it to stack
    int temp;
    cin >> temp;
    stack_name.push(temp);
  }

//This loop prints the element of stack and clears the stack at the ending of loop
  while (!stack_name.empty()) {
    cout << stack_name.top() << " ";
    stack_name.pop();
  }
```

```
  return 0;
}
```

Priority Queue

What is Stack?

Priority queues are a type of container , specifically designed such that the first element of the queue is the greatest of all elements in the queue and elements are in non increasing order (hence we can see that each element of the queue has a priority {fixed order}).

Using The Inbuilt priority queue Data-Structure in STL

Syntax: `stack <string> stack_name;`

Some functions of stack

- `push()` Insert an element `O(log(n))`
- `pop()` removes a element `O(log(n))`
- `top()` Acess the top element `O(1)`
- `empty()` Returns whether the priority queue is empty or not `O(1)`
- `size()` Returns size of priority queue `O(1)`

```
int main() {


priority_queue<int> pq; // greater to low (descending order)
pq.push(10);
pq.push(20);

//Iterating over the pq and poping and acessing the elements simaltanously

while (!(pq.empty())) {
  cout << pq.top() << "\n";
  pq.pop(); // O(logn)
}
cout << pq.top() << "\n"; // O(1)

pq.push(10);
pq.push(20);
pq.push(40);
pq2 = pq;
while (!(pq.empty())) {
  cout << pq.top() << "\n";
  pq.pop();
}
while (!(pq2.empty())) {
  cout << pq2.top() << "\n";
  pq2.pop();
}
cout << pq.top() << "\n";
  cout << pq.size() << "\n";
  cout << pq.empty() << "\n"; // O(1)
}
```

List

Lists are sequence containers that allow non-contiguous memory allocation. As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick.

The elements of a list cannot be accesed directly we have to itrate through the list to acess that element

Syntax: `list <int> list_name;`

`list <int> list_name(5,100)` This list will contain 5 elements of value 100

Some functions of list

- `begin()` It returns an iterator pointing to the first element of the list. complexity is `O(1)`
- `end()` This function returns an iterator to the element past the last element of the list. not the last element `O(1)`
- `empty()` It returns whether the list is empty or not. It returns 1 if it is empty otherwise 0 `O(1)`
- `back()` It Returns the value of the last element in the list . `O(1)`
- `assign()` Assigns new elements to list by replacing current elements and resizes the list. `O(n)`
- `erase()` Removes a single element or a range of elements from the list. `O(N)`
- `sort()` list_name.sort() it sorts the list
- `front()` Returns the value of the first element in the list.
- `back()` Returns the value of the last element in the list .
- `push_front(g)` Adds a new element 'g' at the beginning of the list
- `push_back(g)` Adds a new element 'g' at the end of the list.
- `pop_front()` Removes the first element of the list
- `pop_back()` Removes the last element of the list

```
#include <bits/stdc++.h>
using namespace std;

int main() {
  list <int> list1; // Decleare a list
  list <string> subjects{"phy", "chem", "maths", "pcom"}; // Initialize a list

  subjects.sort();
  subjects.reverse();

  cout << subjects.front() << endl; // Displays the 1st element(index=0)

  subjects.pop_front(); // Removes the 1st element(index=0)

  subjects.push_front("phy"); // Adds element to 0th index or front
  subjects.push_back("bio"); // Adds element to last of the list

  subjects.remove("maths"); // Removes maths from list

  auto var = subjects.begin();
  var++; var++;
  subjects.insert(var, "Maths");  // Adds maths to the list

  for (string i : subjects) { //Itrate Over the list
    cout << i << " ";
  }

  cout << endl;
  for ( auto j = subjects.begin(); j != subjects.end(); j++) {//Itrate Over the list
    cout << *j << "-->";
  }
  return 0;
}
```

Sets

Sets are a type of associative containers in which each element has to be unique, because the value of the element identifies it. The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.

Some functions of sets:

- `begin()` – Returns an iterator to the first element in the set.
- `end()` – Returns an iterator to the theoretical element that follows last element in the set.

- `size()` – Returns the number of elements in the set.
- `empty()` – Returns whether the set is empty.
- `insert()` – Inserts an element in set
- `find()` – Finds the element in set, If present returns an iterator pointing to that element of the set and if absent returns an iterator pointing to last element of set
- `erase()` – Removes an element from the set

```
#include <bits/stdc++.h>
using namespace std;

int main() {
  set <int> set_name; // Set which stores from small to large
  set <int, greater<int>> set_name2; // Set which stores from large to small

  int n; cin >> n; // take input into a set
  for (int i = 0; i < n; i++) {
    int temp; cin >> temp;
    set_name.insert(temp);
  }

  set_name.erase(3);// Deletes 3 from the set

  for (auto i : set_name) { // Iterate over the set and print it
    cout << i << " ";
  }
  cout << endl;
  auto i = set_name.find(2);// Searches for 2 in set
  cout << *i << endl; // Prints 2

  if (i != set_name.end()) {cout << "Found" << endl;}// If element is found prints found
  else {cout << "Not found" << endl;}// If element is not found prints not found

}
```

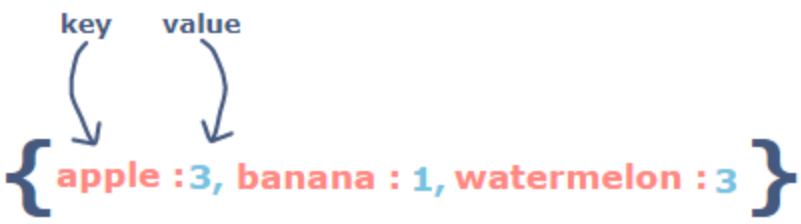
Maps

maps are the containers that strores a Key-Value pair, Each element has a key and a mapped value

No two mapped values can have the same key values.

In C++, maps store the key values in *ascending order* by default.

Maps takes a pair as its argument



A visual representation of a C++ map.

Syntax: `map<string,int> map_name;`

Some functions of map:

- `begin()` Returns an iterator to the first element in the map
- `end()` Returns an iterator to the theoretical element that follows last element in the map
- `size()` Returns the number of elements in the map
- `empty()` Returns whether the map is empty
- `pair insert( Key, value )` Adds a new element to the map
- `erase(iterator position)` Removes the element at the position pointed by the iterator
- `erase()` Removes a single element or a range of elements

- `clear()` Removes all the elements from the map

```
#include <bits/stdc++.h>
using namespace std;

int main() {
  map <string, int> map_name;

// Insert a key-value pair in map
  map_name.insert(pair<string, int> ("Apple", 120)); // Way 1
  map_name.insert(make_pair("Mango ", 90)); //Way 2
  map_name["banana"] = 40;

  // Searching in a Map
  string key;
  cin >> key;
  auto i = map_name.find(key);
  cout << map_name[key] << endl; // Prints 120
  if (i != map_name.end()) {
    cout << "price of " << key << " is " << map_name[key] << endl;
  }
  else {cout << "Not found" << endl;}

  //Delete a key-Value pair
  map_name.erase(key);

  map_name["Kiwi  "] = 50;
  map_name["Orange"] = 80;
  map_name["Grapes"] = 70;
  cout << endl;

  //Itrate over the map using for each loop
  cout << "Key" << "    " << "Value" << endl;
  for (auto i : map_name) {

    cout << i.first << "    " << i.second << endl;
  }

}
```

Multimaps

Multimap is similar to map with the only difference that a multimap can store more than one value against a key. Also, it is NOT required that the key value and mapped value pair has to be unique in this case. One important thing to note about multimap is that multimap keeps all the keys in sorted order always

Some functions of multi map:

- `begin()` – Returns an iterator to the first element in the multimap
- `end()` – Returns an iterator to the theoretical element that follows last element in the multimap
- `size()` – Returns the number of elements in the multimap
- `empty()` – Returns whether the multimap is empty

Syntax: `multimap <string,int> map_name;`

```
#include <bits/stdc++.h>
using namespace std;

int main() {
  multimap <char, string> map_name; // Declare a Multi-map
  int n; cin >> n; // No of elements multimap should contain

  for (int i = 0; i < n; i++) { //This loop takes input and stores it in multimap
    char character_var; cin >> character_var;
    string string_var; cin >> string_var;
    map_name.insert(make_pair(character_var, string_var));
  }

  for (auto i : map_name) {// This loop itrates Over multimap and print its elements
    cout << i.first << "  " << i.second << endl;
  }

  map_name.erase(map_name.begin());// removes "a apple" from multimap
  cout << endl;

  for (auto i : map_name) {//again prints elements of multimap
    cout << i.first << "  " << i.second << endl;
```

```
  }
  cout << endl;

  auto var1 = map_name.find('c');// searching in multimap
  cout << var1->second << endl;// Prints Cat
  cout << var1->first << endl;// Prints c
}
```