ADTs

Queues
o Circular Queues
o Priority Queues
o Deques

Linked lists
o Single Linked Lists
o Double Linked Lists
o Circular Linked Lists
o Node-based storage with arrays

==========

ADTs
====

Abstract Data Type (ADT) is a mathematical model for data types, where the data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of its possible values, operations & the behavior of these operations.

We have already considered the following ADTs. Arrays, Stacks, Queues, etc.

Circular Queues

Queue (is also called as linear queue) supports typically following operations.

push(T)	Inserts given element at back
pop()	Removes the element from front
T front()	Returns the element in front
T back()	Returns the element in back
bool empty()	Returns true iff queue is empty
int size()	Returns number of elements

apart from its constructors and destructor.

Explain operation of queue (push(), pop()) from data structure point of view.

- The queue is initialized to zero elements(i.e. empty).
- Determine whether the queue is empty or not.
- Determine if the queue is full or not.
- Insertion of the new element from the rear end (Enqueue).
- Deletion of the element from the front end (Dequeue).

Therefore, if the operations are not implemented properly, it is quiet possible that queue may appear to be full even though it has some free space.

Circular queues are queue data type with one difference. The first element of the queue is connected with the last element of the queue through a pointer.

- Linear queue organizes the data elements in a sequential order.
- Circular queue arranges the data in the circular pattern.
- Order of execution(selection) of the element is FIFO in linear queue.
- Order of executing(selecting) element can be changed.
- For linear queue the new element is added from the rear end and removed from the front.
- For circular queue, insertion and deletion can be done at any position.

- Linear queue is inefficient when compared to circular queue.

Circular queue keeps track of the front and rear by implementing some extra logic so that it could trace the elements that are to be inserted and deleted. The circular queue does not generate the overflow condition until the queue is full in actual.

Some conditions followed by the circular queue:

- Front must point to the first element.
- The queue will be empty if Front = Rear.
- When a new element is added the queue is incremented by value one(Rear = Rear + 1).
- When an element is deleted from the queue the front is incremented by one (Front = Front + 1).

It also keeps track of the front and rear by implementing some extra logic so that it could trace the elements that are to be inserted and deleted. With this, the circular queue does not generate the overflow condition until the queue is full in actual.

Empty Circular Queue to begin with ==>
F represents Front, R represents Rear

Initially the queue is empty.

```
-1   0    1    2    3    4
F
R
```

Elements are inserted ==>

```
A
F
R

A    B
F    R

A    B    C    D    E
F                  R
```

Element Deletion(Dequeuing) ==>

```
-    -    C    D    E
          F         R
```

Element insertion(Enqueuing) ==>

```
X    -    C    D    E
R         F
```

Circular queue is also called as buffer or ring buffer.

Used in the OS to manage various queues - CPU scheduling, Print scheduling, Memory management. Traffic signal systems also makes use of circular queue.

Priority Queues

Priority Queue is an abstract data type which is an extension of queue with following characteristics.

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

Different data structures can be used to implement it including linked lists and heap. Regardless of how it is represented, its order of for certain operations at **max** is,

isEmpty() of O(1),
Top() of O(n),
push() of O(1),
pop() of O(n).

If Max Heap is used to represent it, then the order changes to

isEmpty() of O(1),
Top() of O(1),
push() of O(log(n)),
pop() of O(log(n)).

Deques

Double ended queues are sequence containers with the feature of expansion and contraction on both the ends. They are similar to vectors, but are more efficient in case of insertion and deletion of elements at the end, and also the beginning.

Unlike vectors, contiguous storage allocation may not be guaranteed.

The functions for Deque are same as vector, with an addition of push and pop operations for both front and back.

Methods supported by Deque are same as vector with addition of push & pop for both front and back.

Used in OS(Multiprocessor Systems)

Lab::

Implement Circular Queue using an array as well as linked list.

==========

List in C++
==========

List is a sequential container that can insert and delete elements locally in constant time, i.e., at a rate that is independent of the size of the container.

E.g., Train and its wagons/carriages

Operations supported

operator=

Iterators:

begin
end
rbegin
rend
cbegin Return const_iterator to beginning (public member function)
cend Return const_iterator to end (public member function)
crbegin Return const_reverse_iterator to reverse beginning (public member function)
crend Return const_reverse_iterator to reverse end (public member function)

Using List iterators

Suppose List cars is defined and has few elements in it.

```
auto it1 = cars.begin();
++it1;
cars.erase(it1);
```

****
where it1 points to?
****

const_iterator over iterator

```
std::vector<int> integers{ 3, 4, 56, 6, 778 };
```

If we were to read & write the members of a container we will use iterator:

```
for( auto it = integers.begin() ; it != integers.end() ; ++it )
       {*it = 4;  std::cout << *it << std::endl; }
```

If we were to only read the members of the container integers you use const_iterator which doesn't allow to write or modify members of container.

```
for( std::vector<int>::const_iterator it = integers.begin() ; it != integers.end() ; ++it )
       { cout << *it << endl; }
```

std::vector<int>::const_iterator returned by cbegin()
so one can only write auto it = integers.cbegin();

Capacity:

empty()
size()
max_size()

Element access:

front()
back()

Modifiers:

assign() (two variations available, (int, const T&=T()) and (iterator,iterator)
    Assign new content to container

Sample code
```
-----------
  std::list<int> first;
  std::list<int> second;

  first.assign (5,420);   // 5 ints with value 420
  second.assign (first.begin(),first.end()); // a copy of first

  int ia[]={1,2,3};
  first.assign (ia,ia+3);  // assigning from array

  std::cout << "Size of first: " << int (first.size()) << '\n';
  std::cout << "Size of second: " << int (second.size()) << '\n
```

****
Output?
3
5
****

emplace_front Insert element at beginning
emplace_back Construct and insert element at the end
push_front
push_back

pop_front Delete first element
pop_back Delete last element

erase Erase elements

swap Swap content (public member function)

      void swap (list& x);
      Exchanges the content of the container by the content of x, which is another list of the same type. Sizes may differ.
      All iterators, references and pointers remain valid for the swapped objects.

resize Change size

      void resize (size_type n);
      void resize (size_type n, const value_type& val);

      for (int i=1; i<10; ++i) mylist.push_back(i);

      mylist.resize(5);
      mylist.resize(8,100);
      mylist.resize(12);

clear() - Clears the content (new size becomes 0)

Operations
---

splicing is joining or connecting.

splice() Transfer elements from list to list

Three different forms.

entire list (1)

```
void splice (const_iterator position, list& x);
void splice (const_iterator position, list&& x);
```

single element (2)

transfers only the element pointed by i from x into the container.

```
void splice (const_iterator position, list& x, const_iterator i);
void splice (const_iterator position, list&& x, const_iterator i);
```

element range (3)

transfers the range [first,last) from x into the container.

```
void splice (const_iterator position, list& x, const_iterator first,
const_iterator last);
void splice (const_iterator position, list&& x, const_iterator first,
const_iterator last);
```

E.g.,

```
// splicing lists
#include <iostream>
#include <list>

int main ()
{
  std::list<int> mylist1, mylist2;
  std::list<int>::iterator it;

  // set some initial values:
  for (int i=1; i<=4; ++i)
     mylist1.push_back(i);      // mylist1: 1 2 3 4

  for (int i=1; i<=3; ++i)
     mylist2.push_back(i*10);   // mylist2: 10 20 30

  it = mylist1.begin();
  ++it;                         // points to 2

  /* First form of splice */
  mylist1.splice (it, mylist2); // mylist1: 1 10 20 30 2 3 4
                                // mylist2 (empty)
                                // "it" still points to 2 (the 5th element)

  /* Second form of splice */
  mylist2.splice (mylist2.begin(),mylist1, it);
                                // mylist1: 1 10 20 30 3 4
```

```
                                // mylist2: 2
                                // "it" is now invalid.
  it = mylist1.begin();
  std::advance(it,3);           // "it" points now to 30

  /* Third form of splice */
  mylist1.splice ( mylist1.begin(), mylist1, it, mylist1.end());
                                // mylist1: 30 3 4 1 10 20

  std::cout << "mylist1 contains:";
  for (it=mylist1.begin(); it!=mylist1.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  std::cout << "mylist2 contains:";
  for (it=mylist2.begin(); it!=mylist2.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}

Output:

mylist1 contains: 30 3 4 1 10 20
mylist2 contains: 2


remove()     Remove elements with specific value

remove_if()  Remove elements fulfilling condition

unique()     Remove duplicate values

merge()            Merge sorted lists

             void merge (list& x);
             void merge (list&& x);
             template <class Compare>
              void merge (list& x, Compare comp);
             template <class Compare>
                  void merge (list&& x, Compare comp);

            Merges x into the list by transferring all of its elements at their
respective ordered positions into the container

            This effectively removes all the elements in x (which becomes
empty), and inserts them into their ordered position within container (which
expands in size by the number of elements transferred).

// list::merge
#include <iostream>
#include <list>

// compare only integer part:
bool mycomparison (double first, double second)
{ return ( int(first)<int(second) ); }

int main ()
{
  std::list<double> first, second;
```

```
  first.push_back (3.1);
  first.push_back (2.2);
  first.push_back (2.9);

  second.push_back (3.7);
  second.push_back (7.1);
  second.push_back (1.4);

  first.sort();   // 2.2 2.9 3.1
  second.sort();  // 1.4 3.7 7.1

  first.merge(second);  // 1.4 2.2 2.9 3.1 3.7 7.1

  // (second is now empty)

  second.push_back (2.1);  // 2.1

  first.merge(second,mycomparison);

  std::cout << "first contains:";
  for (std::list<double>::iterator it=first.begin(); it!=first.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}

Output:

first contains: 1.4 2.2 2.9 2.1 3.1 3.7 7.1

Notice how in the second merger, the function mycomparison (which only compares
the integral parts) did not consider 2.1 lower than 2.2 or 2.9, so it was
inserted right after them, before 3.1.
==========================================================================
sort()            Sort elements in container

reverse()    Reverse the order of elements
```

To be done next

o Single Linked Lists
o Double Linked Lists
o Circular Linked Lists
o Node-based storage with arrays