STL (STACKS,QUEUES,PRIORITY QUEUES)

Prerequisites:

1) Comfort in programming in C++.

STACKS

MOTIVATION PROBLEM:

Let us consider the following problem.
There are N students in a classroom. These students will submit their Exercise Books for correction to the teacher. Each student will come and put his/her exercise book on top of the existing pile of copies and enter his/her name in the software you are about to write.
Now the teacher wants to start checking exercise books from the top of the pile(this would obviously be easier for her). Before picking an exercise book she asks our software the name of the student whose exercise book she is about to check. After checking each exercise book she tells the software “Checked”and gives the copy away. Then she picks up the next copy on the pile asks the software for the name of the student whose book she is about to check. This process goes on till the pile becomes empty.
Can you write a program to do it?
Use arrays may be…..?
Complicated?

C++ HAS A PREWRITTEN DATA STRUCTURE FOR THIS PURPOSE!!

Don’t worry. Our C++ STL(Standard Template Library) has a data structure called stack which allows us to do exactly this. When we use the stack data structure, the computer creates a pile just like the pile of books in the computer’s memory. Each time we enter(technically called push) a name to the stack, the computer adds this name to the top of that pile in the memory. Let us see how to use STL stack.

CREATING A STACK:

Before using a stack, we have to create it(the pile in the computer memory). Remember how we created(declared) an array? Similarly we have to create a stack as well, but the syntax is slightly different. Let’s see how to create a stack:

stack <datatype> name;

**Data Type can be int, long long, float, double,.... anything we covered in the tutorial on data types.

Eg:

stack <string> S;

The above statement creates a pile named S in the computer's memory which will contain strings (in our problem, it will contain the names of the student). This pile is now empty.

PUSHING ELEMENTS TO THE STACK:

When we push an element to a stack, the element is always placed at the top.
Let's understand this with the following example:
First, A comes. Since the pile was empty, A's exercise copy is now at the top of the pile(stack).
Next B comes. According to our stack data structure where will B's exercise copy be placed? It will be placed on the top of A's exercise copy. Now B's exercise copy is at the top of the pile.
Next comes C. Where will his exercise copy be placed according to our stack data structure? C's exercise copy will now be placed at the top of the pile just above B's exercise copy. Next comes D, then E,F and last G. We will have a pile like this:

G's Exercise copy
F's Exercise copy
E's Exercise copy
D's Exercise copy
C's Exercise copy
B's Exercise copy
A's Exercise copy

We want to do a simple thing. The student who comes will enter his name and we will push his/her name in our stack S, so that at the end, the student who submitted last will have his/her name at the top of the stack S.Here goes the syntax to push an element to the top of the stack.:

<name of stack>.push(<some element>)

Eg:

S.push("Jack");

This will add the name Jack at the top of the pile/stack.

ACCESSING THE TOPMOST ELEMENT:

Let's assume the submission process is done. Now the teacher wants to start correcting. She draws the exercise copy at the top of the pile and asks our program to print the name of the student to whom this exercise copy belongs.
In other word we have to access the topmost element in our stack S. This is the syntax:

<name of stack>.top()

Eg:

cout<<S.top();

REMOVING THE TOPMOST ELEMENT:

And when the teacher has finished checking a student's copy, she has to tell the program to remove the top most name. The following command removes the topmost element.

<stack name>.pop();

Eg:
S.pop();

Now whose name is at the top of the stack? The student who submitted before the last student. Doesn't this make our work easy?
Again when the teacher takes an exercise copy from the top of the pile, she will get the appropriate name(whose copy she is about to correct) from our program by accessing the top most element in the stack.
If we have the following pile:

G's Exercise copy
F's Exercise copy
E's Exercise copy
D's Exercise copy
C's Exercise copy
B's Exercise copy
A's Exercise copy

And we keep popping elements till the stack becomes empty, in what order are they popped/removed?
First G is popped(this is the top most element) then
F,E,D,C,B,A.
Right?

CHECKING WHETHER STACK HAS BECOME EMPTY:

To check whether the stack is empty at any point we write the following code:

```
if(<stack name>.empty()==true)
{
        cout<<"Stack Empty. No Books left to be checked";
}
```

Eg:

```
if(S.empty()==true)
{
        cout<<"Stack Empty. No Books left to be checked";
}
```

PRINCIPLE OF STACK DATA STRUCTURE(LIFO):

You have seen that the last element that was inserted on the pile was removed first i .e the student who submitted last had his/her copy checked first. So we say that stack is a LIFO(Last In First Out) data structure.

VISUALIZATION OF THE STACK DATA STRUCTURE:

You can visit this site for excellent visualization of the stack data structure:
https://visualgo.net/en/list
Instructions to visualize stack data structure:

1. Press ESC to close the dialogue box.
2. On the top black bar click on Stack.
3. See the left bar. You have things like Create, Peek, Push, Pop. If you cannot see these click on the green > arrow and you should be able to see these.
4. Click on Create -> Random.
5. Let's first visualize the push operation. Click push -> Go.
6. You should now see the element at the top of the pile.
7. Feel free to push some more elements.
8. Let's visualize the pop operation. Click pop.
9. You will now see how the topmost element gets removed from the stack.
10. Peek is same as top() i.e to get the top most element of the stack.

CONCLUSION:

That's it. You now know what is the stack data structure. There are many useful things that can be done using stacks. This is just the basic! Can you think of some other real world examples where the concept of stack is used?

QUEUES

MOTIVATION PROBLEM

Now consider the same problem as above. Now only thing is the teacher is a bit more fair and starts checking the exercise books from the bottom of the pile i.e she first checks the copy submitted by the first student, then the copy of the student who submitted second and so on. She draws the exercise copy at the bottom of the pile. She asks the name of the student whose copy she is about to check. And mind, here is the difference! Our program should return the

name of the student at the bottom of the pile and not the one at the top(as was in the case of stack). Once corrected, she tells the software to remove the name from the bottom of the pile. Again, she draws another copy from the bottom of the pile and asks our program to give the name of the student and this process continues till the pile becomes empty. Can you write a program to do it?

AGAIN C++ HAS A PREWRITTEN DATA STRUCTURE FOR THIS PURPOSE!!

Now we have another data structure by the name of queue provided by STL which does exactly this. When we create a queue data structure, the computer maintains a pile in the memory just as it did in the case of a stack. It can add elements to the top of the pile (just like stack). The only difference is elements are removed from the bottom of the pile and not top (as in the case of stack).

CREATING A QUEUE

Here is the syntax to create a queue:

queue <data type> name;

Eg:
queue <string> Q;

This creates a pile by the name Q in the computer's memory. This pile is now empty. It can store strings.

PUSHING ELEMENTS TO THE TOP OF THE QUEUE

When we push an element into a queue, the element gets added at the top of the pile maintained by the computer's memory.
This is same as pushing elements to a stack. Here is the syntax:

<queue name>.push(some element);

Eg:
Q.push("Jack");

This will add Jack to the top of the pile Q in the computer's memory.

ACCESSING THE BOTTOM MOST ELEMENT:

Now the teacher wants to know the name of the student whose exercise copy is at the bottom. To do so we write:

<queue name> . front()

Eg:

```
cout<<Q.front();
```

REMOVING ELEMENTS FROM THE BOTTOM OF THE QUEUE:

Every time the teacher finishes correcting a copy she removes it from the bottom of the pile. Let's write code to remove the corresponding name from our queue Q in the computer's memory:

```
<queue name> .pop()
```

Eg:

```
Q.pop()
```

This removes the bottom most element from the pile. Next when the teacher asks for the bottom most element, she gets the name of the student who submitted second after the first one. Great!

CHECKING WHETHER QUEUE IS EMPTY:

Syntax:

```
if(<queue name>.empty()==true)
{
        cout<<"Queue empty!";
}
```

Eg:

```
if(Q.empty()==true)
{
        cout<<"Queue empty!";
}
```

PRINCIPLE OF QUEUE DATA STRUCTURE(FIFO):

You have seen that the first element that was inserted on the pile was removed first i .e the student who submitted first had his/her copy checked first. So we say that queue is a FIFO(First In First Out) data structure.

VISUALIZATION OF THE QUEUE DATA STRUCTURE:

Refer to the following link for an excellent visualization: https://visualgo.net/en/list
Instructions to visualize queue data structure:

1. Press ESC to close the dialogue box.

2. On the top black bar click on Queue
3. See the left bar. You have things like Create, Peek, Enqueue, Dequeue. If you cannot see these click on the green > arrow and you should be able to see these.
4. Click on Create -> Random.
5. Let's first visualize the push operation. Click Enqueue -> Go. The enqueue operation is same as our push operation.
6. You should now see the element at the end of the pile.
7. Feel free to push some more elements.
8. Let's visualize the pop operation. Click Dequeue.
9. You will now see how the frontmost element gets removed from the queue.
10. Peek is same as front() i.e to get the bottom most element in the pile.

***The bottom of the pile is the left end and the top of the pile is the right end in this visualization. In other words the pile is rotated 90 degrees towards right. :P
So don't get confused!

PRIORITY QUEUE:

MOTIVATION PROBLEM:
Let us again consider the same example of submitting exercise copies. Now, when the students come to submit their exercise copies, they don't enter their name but their score in the last test. And the exercise copy is not placed at the top of the pile. It is placed somewhere in the pile depending on the score of the student. In the end, the student who got the highest score has his exercise copy at the top of the pile whereas the student who got the lowest score has his copy at the bottom of the pile. Again, the teacher starts correcting copies and she draws the copy that is at the top, checks it and removes it from the pile. Then she takes the next copy, corrects it and removes it from the pile. Basically her aim is to correct copies in such a way that the student with the highest grade gets his/her copy back first and the student with lowest grade gets his copy at last.
Think of writing a program!

AS YOU CAN GUESS, C++ AGAIN HAS A PREWRITTEN DATA STRUCTURE FOR THIS PURPOSE!!
This data structure goes by the name priority queue. Like stack and queue, when we create a priority queue a pile gets created in computer's memory. When we insert an element, the element is not necessarily placed at the top of the pile. It is placed somewhere in the pile so that the pile is sorted i.e the smallest element is at the bottom while the largest element is at the top. Removing elements is from the top of the pile just like stack.

Let us see how to implement heaps(min heap and max heap) using the C++ STL priority queues. Max Heaps allow us to access the maximum element in O(1) and allows us to insert an element in O(log N) and allows us to delete the maximum element in O(log N).
Min Heaps allows us to access the minimum element in O(1) and allows us to insert an element in O(log N) and allows us to delete the minimum element O(log N).

How do we create a maximum heap using priority queue?

CREATING A PRIORITY QUEUE:
Syntax:

prioirty_queue <data type> name;

Eg:

priority_queue <int> P;

This creates a pile P in the computer's memory in which pile insertion is done as mentioned above.

PUSHING ELEMENT INTO A PRIORITY QUEUE:
When we push an element, the element is placed somewhere in the pile so that the pile remains sorted i.e the smallest element is at the bottom while the largest element is at the top.
Here goes the syntax:

<Name of Priority Queue> .push(<some value>);

Eg:
P.push(5);

Let us take a small example to see how elements are inserted. Say the first student comes and he enters his score which happens to be 2. Since our pile is empty, therefore this score is placed at the top of the pile. Our pile looks like:
2
Next comes another student whose score happens to be 1. What will our pile look like now?
2
1
Next comes a student whose score is 5. Our pile becomes:
5
2
1
Next comes a student who enters his score as 4. The pile becomes:
5
4
2
1
You can see clearly how the score depending on it's value is placed somewhere in the pile so that the pile is always sorted.

ACCESSING THE TOP MOST (MAXIMUM) ELEMENT:

At any point of time the teacher may wish to know the marks of the student at the top of the pile whose exercise copy she will correct. We have a function for this as well.

Syntax:

<Name of Priority Queue> .top();

Eg:

P.top();

DELETING THE TOP MOST (MAXIMUM) ELEMENT:

After correcting the copy of the student with the maximum score, the teacher removes it from the pile. Then she gets the exercise copy of the student who secured the second highest.

How do we remove an element from the top of the pile?

Syntax:

<Name of Priority Queue>.pop()

Eg:

P.pop()

CHECKING WHETHER PRIORITY QUEUE IS EMPTY:

At any point of time you may s=check whether our priority queue has any elements or not?

Syntax:

```
if(<Name of Priority Queue>.empty()==true)
{
        cout<<"Empty!";
}
```

Eg:

```
if(P.empty()==true)
{
        cout<<"Empty!";
}
```

Think:

Are you wondering is it possible to create a priority queue so that the smallest element is at the top of the pile and the largest element is at the bottom of the pile? Well it turns out that there is a way to do it. The only difference is you have to tell while creating the priority queue that you want the pile sorted in descending order i.e the smallest element is at the top of the pile and the largest element is at the bottom of the pile.

If you are interested, please feel free to Google out how to do it? Only a small change is required!

CONCLUSION:

You know understand priority queue. Great! Can you guess the complexity when an element is being inserted in the pile every time? It is O(Log N), which is quite fast. The proof of this and how C++ implements it is beyond the scope of this tutorial.