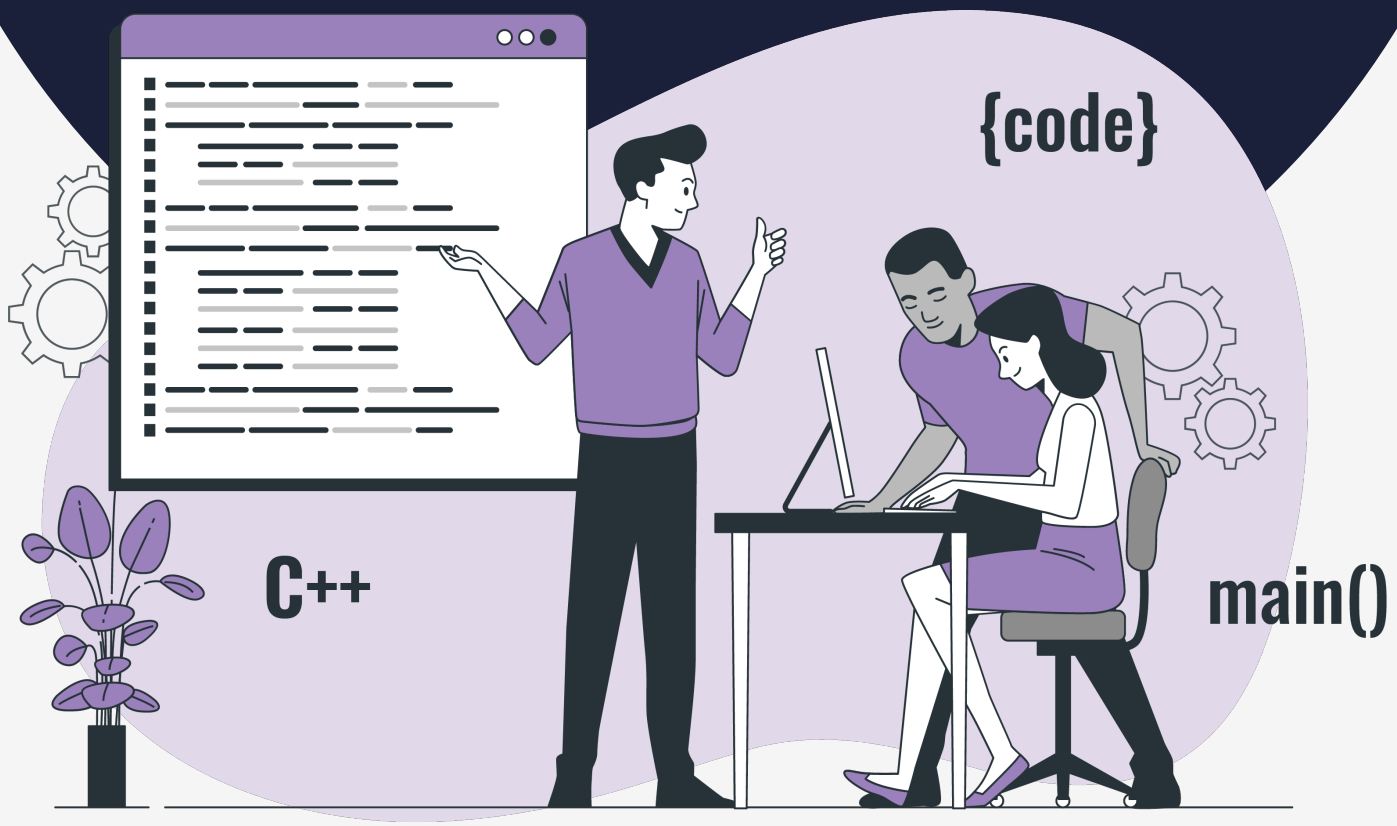
Lesson:

Circular linked list and List in STL



PW SKILLS

Concepts involved

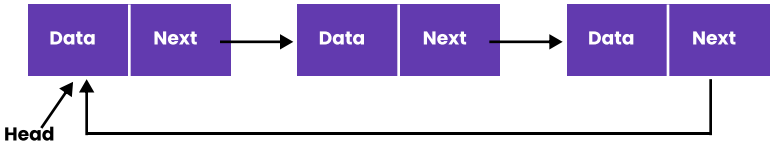
- Pointers
- Singly linked list

List of Concepts Involved:

- What is a circular linked list?
- Advantages of a circular linked list over a singly linked list
- Types of circular linked lists
- Implementation of a node in a circular linked list
- Traversal in a circular linked list
- Insertion at kth position in a circular linked list
- Updation at kth position in a circular linked list
- Deletion at kth position in a circular linked list
- What is a list in C++ STL?
- Advantage of using lists
- Declaration of a list
- Iterator functions
- Traversal of a list
- Inserting elements into the list
- Updating elements of a list
- Deleting elements from the list
- Other member functions of the list container

Topic 1: What is a circular linked list?

A circular linked list is a normal linked list with its tail node being connected to the head node, thus forming a circle. There is no NULL pointer at the end.



Topic 2: Advantages of a circular linked list over a singly linked list

1. We can start our traversal from any node of the list. We'll stop when we encounter a visited node for the second time.
2. It is useful in cases where we have to traverse a list multiple times in a circular fashion.

Topic 3: Types of circular linked lists

1. Circular singly linked list. It is a modification of a singly linked list, where the tail node is connected to the head node of the linked list.
2. Circular doubly linked list. It is a modification of a doubly linked list, where the tail node is connected to the head node and vice versa. In this the tail node's next pointer points to the head node and the head node's prev pointer points to the tail node.

Topic 4: Implementation of a node in a circular linked list

- Since the only difference between a circular linked list and a normal linked list is that the tail is connected to the head, the implementation of the node remains the same.

Topic 5: Traversal in a circular linked list

- The forward traversal is the same for both circular singly linked list and circular doubly linked list.
- Steps-
    1. Make a new node pointer for the traversal and make it point to the head of the linked list.
    2. Loop till you don't encounter the head of the list again. A. process the dataB. update the value of the pointer being used for traversal.

Code

Time complexity- O(n)

- In the case of circular doubly linked lists, we have backward traversal too. The steps are the same as in forward traversal with the only change in the node value updation. In the case of backward traversal, we use the prev pointer of the node instead of the next pointer.

Code

Time complexity- O(n)

- Since all the nodes in a circular list are symmetrical, it doesn't matter much if you start your traversal from the head or the tail unless specified otherwise.

Topic 6: Insertion at k^{th} position in a circular linked list

- Due to the symmetric nature of a circle, the process to insert a node at any point is going to be the same.

Circular singly linked list

- Steps-

1. Make a new node to insert in the list and make a pointer that is pointing to it.
2. Check if the list is empty. If yes, then make the new_node the head of the list and make it point to itself to make a circle.
3. Traverse the linked list till you reach the position after which the new node has to be inserted.
4. Make the new_node point to the node after the prev_node.
5. Make the prev_node point to the new_node.
6. If the position of insertion is 0, then update the head of the list.

Time complexity- O(n)

Code

Circular doubly linked list

- Steps-

1. Make a new node to insert in the list and make a pointer that is pointing to it.
2. Check if the list is empty. If yes, then make the new_node the head of the list and make it point to itself to make a circle using both its pointers.
3. Traverse the linked list till you reach the position after which the new node has to be inserted.
4. Make a new pointer to represent the next node after the prev_node.
5. Make the prev_node and new_node point to the new_node and the next_node respectively using their next pointers.
6. Make the new_node and the next_node point to the prev_node and the new_node respectively using their prev_node.
7. If the position of insertion is 0, then update the head of the list.

Time complexity- O(n)

Code

Topic 7: Updation at k^{th} position in a circular linked list

- The way of updation is the same in both circular singly linked list and circular doubly linked list.
- Steps-

1. Make a new node to traverse the list.
2. Traverse the list till you reach the required node.
3. Update the value of the node.

Code

Time complexity- O(n)

Topic 8: Deletion at kth position in a singly linked list

- Same as insertion, the deletion of a node is symmetric for all nodes in the list.

Circular singly linked list

- Steps-

1. Traverse the linked list till you reach the position after which the node that has to be deleted is present.
2. Make a temporary node pointer to point to the node that has to be deleted.
3. Make the prev_node point to the node after the node to be deleted.
4. Free the space taken up by the deleted node.
5. If the position of deletion is 0, update the head pointer.
6. Update the size of the list.

Time complexity- O(n)

Code

Circular doubly linked list

- Steps-

1. Traverse the linked list till you reach the position after which the node that has to be deleted is present.
2. Make a temporary node pointer to point to the node that has to be deleted.
3. Make a new pointer representing the node after the node to be deleted.
4. Make the prev_node point to the next_node using its next pointer and make the next_node point to the prev_node using its prev pointer.
5. Free the space taken up by the deleted node.
6. If the position of deletion is 0, update the head pointer.
7. Update the size of the list.

Time complexity- O(n)

Code

Topic 9: What is a list in C++ STL?

We have already studied about what a doubly linked list is. It is a linked list in which each node points to 2 other nodes- the previous node and the next node in the list sequence.

So, what is a list in C++ STL? A list is the template class of doubly linked lists that is present in the C++ STL. It supports all the functions that are supported by doubly linked lists and have the same limitations.

In simple words, it is a doubly linked list that has already been defined for us in the form of a class with all the functionalities. We just have to use it in our code as we desire.

Topic 10: Advantage of using lists

The major advantage of using the list in STL is that it makes the implementation of a doubly linked list easy. In this we don't have to define insertion, deletion or any other function. They have already been defined for us. We can directly call them in our program.

Topic 11: Declaration of a list

To use lists in a program, we need to start by including the list header.

```
#include <list>
```

Syntax for declaring a list

```
ist<data_type> name_of_list;
```

Example:

```
1. list<int> list1;
```

This creates a list that can store integer values, and the name of the list is 'list1'.

```
2. list<char> list2;
```

This creates a list that can store character values, and the name of the list is 'list2'.
Note that the data type can be a struct/class as well.

Furthermore, if we want to initialize the list with values, we have 2 ways to do so-

```
list<int> list{1, 2, 3, 4};
```

or

```
list<int> list = {1, 2, 3, 4};
```

Both create a list that contains 4 values- 1, 2, 3 and 4.

Topic 12: Iterator functions

An iterator acts like a pointer that indicates the items inside the list. Iterators are used mainly to move through the data inside a list. Iterators are also used to indicate the memory address of the containers.

There are 5 major types of iterators that we need to learn-

1. **begin():** Returns a bidirectional iterator for the first element
   CODE LINK: https://pastebin.com/wYzwBuXK
   Output: 1

2. end(): Returns a bidirectional iterator for the position after the last element.
If we consider the list nums = {1, 2, 3, 4} then nums.end() will return an iterator pointing to the memory location after 4.

3. rbegin(): Returns a reverse iterator for the first element of a reverse iteration

CODE LINK: https://pastebin.com/9cEEsJLc
Output: 4

4. rend(): Returns a reverse iterator for the position after the last element of a reverse iteration
If we consider the list nums = {1, 2, 3, 4}. rend() is similar to end(). It points to the location after the last element. However, in this case the last element is 1 as the iteration is reversed.

5. advance(): Advances the iterator by 'n' element positions.
Syntax
advance(iterator, n);
Example:

```
list<int> nums = {1, 2, 3, 4};
  auto itr = nums.begin();
  advance(itr, 2);
  cout << *itr;
```

Output: 3
itr is initialized by pointing it to the first element of the list. The advance function makes it move forward by 2 positions. Thus we get 3 as output.

Topic 13: Traversal of a list

There are 2 ways by which we can traverse a list. Let's try to understand both the ways with the help of an example-

1. **Range-based for loop**

```
list<int> nums = {1, 2, 3, 4};
  for (int num : nums)
  {
      cout << num << ' ';
  };
```

Output : 1 2 3 4

In this we iterate through all of the elements of the list nums in order starting with the first element. At each iteration, an instance of the object stored in the list is created and is named num(we can name this anything of our choice). Inside the curly braces, we can process this variable num in whatever way we want.

2. Using iterators

There are 2 types of traversals that we can do using iterators-

a. Forward traversal

```
ist<int> nums = {1, 2, 3, 4};
    for (auto itr = nums.begin(); itr ≠ nums.end(); itr++)
    {
        cout << *itr << ' ';
    }
```

Output: 1 2 3 4

The iterator is initialized by nums.begin() i.e. it points to the first element of the list. We increment the value of itr till it is inside the list i.e. it is not equal to nums.end(). Incrementing the value of the iterators makes us move forward in the list.

b. Backward traversal

```
list<int> nums = {1, 2, 3, 4};
    for (auto itr = nums.rbegin(); itr ≠ nums.rend(); itr++)
    {
        cout << *itr << ' ';
    }
```

Output: 4 3 2 1

The iterator is initialized by nums.rbegin() i.e. it points to the last element of the list. We increment the value of itr till it is inside the list i.e. it is not equal to nums.rend(). Since itr here is a reverse iterator, when we increment its value, we move backwards in the list. This is what differentiates begin() and end() from rbegin() and rend().

Topic 14: Inserting elements to the list

There are 3 ways by which we can insert an element into a list-

- **insert(iterator, element):** inserts element in the list before the position pointed by the iterator.

Example:

```
   list<int> nums = {1, 2, 3, 4};
    auto itr = nums.begin();
    advance(itr, 2); // itr points to 3
    nums.insert(itr, 5); // inserts 5 before 3
    for (auto num : nums)
        cout << num << ' ';
```

Output: 1 2 5 3 4

- **insert(iterator, count, element):** inserts element in the list before the position pointed by the iterator, count number of times.

Example:

```
list<int> nums = {1, 2, 3, 4};
    auto itr = nums.begin();
    advance(itr, 2); // itr points to 3
    nums.insert(itr, 3, 5); // inserts three 5s before 3
    for (auto num : nums)
        cout << num << ' ';
```

Output: 1 2 5 5 5 3 4

- **insert(iterator, start_iterator, end_iterator):** insert the element pointed by start_iterator to the element pointed by end_iterator before the position pointed by iterator

Example:
CODE LINK: https://pastebin.com/ZWhTRSkH
Output: **1 2 1 2 3 4**

Topic 15: Updating elements of a list

To update the elements of a list, we take the help of iterators.For example if we have a list of 4 elements, {1, 2, 3, 4}, and we want to change the value of the third element to 5.

CODE LINK: https://pastebin.com/iCagCxfD

Topic 16: Deleting elements from the list

There are 2 ways to delete an element from the list-

- **erase(iterator):** deletes the element pointed to by the iterator

CODE LINK: https://pastebin.com/iiq90hqy
Output: 1 2 4

- **erase(start_iterator, end_iterator):** deletes the elements between the start_iterator and the end_iterator, excluding the element pointed by the end_iterator

CODE LINK: https://pastebin.com/2k23ufbk
Output: **1 2 5 6**

Topic 17: Other member functions of the list container

1. front(): returns the value of the first element of the list
2. back(): returns the value of the last element of the list
3. push_front(val): adds a new value val to the start of the list
4. push_back(val): adds a new value val to the end of the list
5. pop_front(): removes the first element of the list
6. pop_back(): removes the last element of the list
7. size(): returns the size of the list
8. empty(): returns 1 if the list is empty else returns 0
9. reverse(): reverses the list

Upcoming Class Teasers

- Stack