Birla Institute of Technology and Science Pilani, Hyderabad Campus

CS F211: Data Structures and Algorithms

2nd Semester 2024-25, Lab Sheet No: 4 (Linked Lists)

General Tips

- The use of STL is strictly prohibited in this lab. You should not use any inbuilt data-structure/ algorithm to do the tasks listed.
- Indent your code appropriately and use proper variable names, if not already provided in the question. Also, use comments wherever necessary.
- Make sure to debug your code after every task or after every code block to avoid having to debug your entire file at once in the last minute.

Overview:

The goal of this lab is to learn how to implement basic functionalities in a Singly and Doubly linked list. You will be given a piece of code which is incomplete. Based on your learnings in the lecture classes and tutorials, try to complete the code and get the expected output. You should finish your task in the given lab timings.

Program 1 (SinglyLinkedList.cpp attached):

Task 1:

The below **search ()** method is a newly added feature to the GenericSinglyLinkedList implementation discussed in the class. The method basically searches whether there is a node in the list containing the given item. Complete the code highlighted as Task 1. Try to utilize the given **isEqual()** method to compare two elements of generic type. You should get the output as shown below:

```
SNode<E> *SLinkedList<E>::search(const E &e)
{
    for (SNode<E> *curr = head; curr != NULL; curr = curr->next)
    {
        // Complete the missing code
     }
    return NULL; // not found
}

bool SLinkedList<E>::isEqual(const E &a, const E &b)
{
    return ((string)a).compare((string)b) == 0;
}
```

Output:

```
Traversing the list : Mayank Ayush Amit
+--------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Get frontmost element
3 : Remove front element
4 : Check if list is empty
5 : Traverse the list
6 : Search for an element
7 : Swap two nodes
8 : Exit
6
Enter the element to search: Mayank
Mayank is present in the list.
+--------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Get frontmost element
3 : Remove front element
4 : Check if list is empty
5 : Traverse the list
6 : Search for an element
7 : Swap two nodes
8 : Exit
6
Enter the element to search: Manas
Manas is NOT present in the list.
```

Task 2:

Newly added method **swap ()** swaps the nodes containing the two given elements in the singly linked list. Again, you are given an incomplete code. You are free to refer to lecture slides to revise the concept of interchanging node pointers and the technique to swap two nodes. Do a thorough understanding of the code before you jump to coding. You should get the output as shown below:

```
void SLinkedList<E>::swap(const E &a, const E &b)
{
    // first search for element 'a' in the list.
    SNode<E> *node1 = search(a);

    if (node1 == NULL)
        return;

    // next, search for element 'b'
    SNode<E> *node2 = search(b);
```

```
    if (node2 == NULL)
        return;

    SNode<E> *prevNode1 = NULL; // pointer to the previous node of [node1]
    SNode<E> *prevNode2 = NULL; // pointer to the previous node of [node2]

    // find previous node of node1 and update prevNode1 pointer
    for (SNode<E> *curr = head; curr != node1; curr = curr->next)
        prevNode1 = curr;

    // find previous node of node2 and update prevNode2 pointer
    for (SNode<E> *curr = head; curr != node2; curr = curr->next)
        prevNode2 = curr;

    // first, we need to fix head pointer
    if (head == node1 || head == node2)
    {
        if (head == node1)
            head = node2;
        else
            head = node1;
    }

    // Handling the case:-
    // head -> ... -> node1 -> node2 -> ... -> NULL
    if (prevNode2 == node1)
    {
        node1->next = node2->next;
        node2->next = node1;
        if (prevNode1 != NULL)
            prevNode1->next = node2;
        return;
    }

    // Handling the case: -
    // head -> ... -> node2 -> node1 -> ... -> NULL
    // Complete the missing code

    // Hold the next pointer of [node1]
    SNode<E> *nextNode1 = node1->next;

    // next pointer of [node1] points to the next pointer of [node2]
    node1->next = node2->next;

    // next pointer of [node2] points to [nextNode1]
    node2->next = nextNode1;

    // next pointer of [prevNode1] points to [node2]
    if (prevNode1 != NULL)
        prevNode1->next = node2;

    // next pointer of [prevNode2] points to [node1]
    // Complete the missing code

}
```

Output:

```
Traversing the list : Tom Smith Mayank Ayush Amit
+--------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Get frontmost element
3 : Remove front element
4 : Check if list is empty
5 : Traverse the list
6 : Search for an element
7 : Swap two nodes
8 : Exit
7
Enter the first element: Smith
Enter the second element: Amit
+--------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Get frontmost element
3 : Remove front element
4 : Check if list is empty
5 : Traverse the list
6 : Search for an element
7 : Swap two nodes
8 : Exit
5
Traversing the list : Tom Amit Mayank Ayush Smith
```

Task 3:

Add a new method **reverse ()** which will reverse the nodes of the linked list. Note that the reversal of linked list should happen in-place i.e. using constant space complexity. Complete the code given below. Hints are given in as comments in the code.

```
template <typename E>
void SLinkedList<E>::reverse()
{
  if (head == NULL || head->next == NULL)
    return;

  SNode<E> *curr = head;
  SNode<E> *prev=NULL;
  while (curr != NULL)
  {
    // Store the next node using a temporary pointer
    // Update the next pointer of current node to point to previous node
    // Update the curr and prev pointers in each iteration
  }
```

```
  // Update the head pointer of the list
}
```

Output:

```
Traversing the list : Tom Smith Mayank Ayush Amit
+--------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Get frontmost element
3 : Remove front element
4 : Check if list is empty
5 : Traverse the list
6 : Search for an element
7 : Swap two nodes
8 : Reverse nodes in list
9 : Exit
8
List reversed successfully!+--------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Get frontmost element
3 : Remove front element
4 : Check if list is empty
5 : Traverse the list
6 : Search for an element
7 : Swap two nodes
8 : Reverse nodes in list
9 : Exit
5
Traversing the list : Amit Ayush Mayank Smith Tom
```

Program 2 (DoublyLinkedList.cpp attached):

Task 4:

Add a new method **isPalindrome()** to the DoublyLinkedList class. The method will check if the list is a palindrome or not. A palindrome list is a type of linked list where the elements read the same forwards as they do backwards. Complete the code given below:

```
bool DLinkedList::isPalindrome()
{
  // Complete the initialization of begin and end pointers.
  // begin is the first node and end is the last node of the list.
  DNode *begin;
  DNode *end;

  while (begin != end)
  {
    // Check if the values of begin and end pointers are different.

    if (begin->next == end)
      break;

    //Update the begin and end pointers for the next iteration.
  }
```

```
    return 1;
  }
```

Output:

```
Traversing the list :  1 2 2 1
+-------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Add at the end
3 : Remove from the front
4 : Remove from the end
5 : Get front-most element
6 : Get end-most element
7 : Get Middle element
8 : Check if list is empty
9 : Reverse List
10 : Traverse the list
11 : Check if list is palindrome
12 : Exit
11
List is  palindrome
```

Task 5:

Method **middle()** is newly added feature to DoublyLinkedList class discussed in the class. The concept of finding middle node in a linked list will turn out to be useful before we jump into Divide and Conquer techniques, specifically applying merge-sort algorithm in linked list, which we shall learn later in the course. Merge Sort always tries to chop the input list into two equal halves. Thus, finding middle node in a linked list is important to understand. As discussed in the class, complete the remaining portion of the code. You should get the output as shown below:

```
const Elem &DLinkedList::middle() const // get middle element
{
    // Using Two pointer Technique

    DNode *slowPtr, *fastPtr;

    // initially both pointers point to the head node
    slowPtr = fastPtr = header;

    // Complete the missing code

    // [slowPtr] is the middle node
    return slowPtr->elem;
}
```

Task 6:

Using pointer technique, find out if the doubly linked list has odd number of nodes or even number of nodes. Print "Odd" if number of nodes are odd, and "Even" otherwise.

(hint: If the fast pointer reaches NULL, it is even, else if it reaches the last node then it is odd.)

```
Traversing the list :  E D C B A
+--------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Add at the end
3 : Remove from the front
4 : Remove from the end
5 : Get front-most element
6 : Get end-most element
7 : Get Middle element
8 : Check if list is empty
9 : Reverse List
10 : Traverse the list
11 : Exit
7
Element at the middle is : C
+--------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Add at the end
3 : Remove from the front
4 : Remove from the end
5 : Get front-most element
6 : Get end-most element
7 : Get Middle element
8 : Check if list is empty
9 : Reverse List
10 : Traverse the list
11 : Exit
1
X
Adding the following element to the front : X
Traversing the list :  X E D C B A
+--------------------------------------+
Please enter one of the following choices:
1 : Add at the front
2 : Add at the end
3 : Remove from the front
4 : Remove from the end
5 : Get front-most element
6 : Get end-most element
7 : Get Middle element
8 : Check if list is empty
9 : Reverse List
10 : Traverse the list
11 : Exit
7
Element at the middle is : C
```

Program 3 (Circular Linked List: CircularLL.cpp is given for your reference):

Implement a scheduling algorithm which you will read in Operating Systems course later. Round Robin is a CPU scheduling algorithm where each process gets a fixed amount of time (time slice) to execute. If a process doesn't complete within its time slice, it's pre-empted and added to the end of the ready queue. This process repeats until all processes are completed. Implement this algorithm using a Circular linked list as discussed in the class.

You may consider the below assumptions:

- Each process can be represented as a node in the circular linked list.
- The "next" pointer of each node points to the next process in the ready queue.
- The CPU scheduler can easily traverse the list in a circular fashion, giving each process its time slice.
- When a process completes or is pre-empted, it's simply removed from its current position and added to the end of the list.
- Take a set of processes with their CPU burst (time required with CPU) and a time slice (or time quantum) and produce an order of their completion as output.
