CSE 4304-Data Structures Lab. Winter 23-24
Batch: CSE 22
Date: October 02, 2024
Target Group: All
Topic: Linked Lists

Instructions:

- Regardless of how you finish the lab tasks, you must submit the solutions in Google Classroom. In case I forget to upload the tasks there, CR should contact me. The deadline will always be 11:59 PM on the day the lab took place.
- Task naming format: fullID_T01L01_2A.c/cpp
- If you find any issues in the problem description/test cases, comment in the Google Classroom.
- If you find any tricky test cases that I didn't include but that others might forget to handle, please comment! I'll be happy to add them.
- Use appropriate comments in your code. This will help you to recall the solution in the future easily.
- Obtained marks will vary based on the efficiency of the solution.
- Do not use <bits/stdc++.h> library.
- Modified sections will be marked with BLUE color.
- You are allowed to use the STL stack unless it's specifically mentioned to use manual functions.

Group	**Tasks**
2A	1 2 5 6
1B	1 2 5 6
1A	2 3 4 8
2B	2 3 4 8
Assignments	2A/1B: 3 4 8 1A/2B: 5 6 9

Task-01:

Implement the basic operations using a 'Singly Linked list'. Your program should include the following functions:

1. **Insert_front**(int key):
    - Insert the element with the 'key' at the beginning of the list.
    - Time Complexity: O(1)
2. **Insert_back**(int key):
    - Insert the element with the 'key' at the end of the list.
    - **Time Complexity: O(1)**
3. **Insert_after_node** (int key, int v):
    - Insert a node with the 'key' after the node containing the value 'v' if it exists. (shows error message otherwise).
    - Time complexity: $O(n)$
4. **Update_node** (int key, int v):
    - Looks for the node with value v and updates it with the new value 'key' (error message if the node doesn't exist)
    - Time complexity: O(n)
5. **Remove_head** ():
    - Remove the first node from the linked list.
    - Time complexity: O(1)
6. **Remove_element** (int key):
    - Removes the node containing the 'key' if it exists (else throw an error message).
    - Time complexity O(n)
7. **Remove_end** ():
    - Remove the last node from the linked list.
    - Time complexity: O(n)

Input format:

- The program will offer the user the following operations (as long as the user doesn't use option 7):
    - Press 1 to insert at front
    - Press 2 to insert at back
    - Press 3 to insert after a node
    - Press 4 to update a node
    - Press 5 to remove the first node
    - Press 6 to remove a node
    - Press 7 to remove the last node
    - Press 8 to exit.
- After the user chooses an operation, the program takes necessary actions (or asks for further info if required).

Output format:

- After each operation, the status of the linked list is printed with the head and tail nodes.

Sample input	Sample Output
1 10	Head=10, Tail=10, 10
7	Head=Null, Tail=Null, Empty
7	Underflow Head=Null, Tail=Null, Empty
6 10	Value Not found Head=Null, Tail=Null, Empty
5	Head=Null, Tail=Null, Empty
5	Underflow Head=Null, Tail=Null, Empty
2 20	Head=20, Tail=20, 20
1 30	Head=30, Tail=20, 30 20
2 40	Head=30, Tail=40, 30 20 40
3 50 20	Head=30, Tail=40, 30 20 50 40
3 60 40	Head=30, Tail=60, 30 20 50 40 60
5	Head=20, Tail=60, 20 50 40 60
7	Head=20, Tail=40, 20 50 40
4 70 50	Head=20, Tail=40, 20 70 40
4 80 50	Value Not found Head=20, Tail=40, 20 70 40
4 80 40	Head=20, Tail=80, 20 70 80
4 90 20	Head=90, Tail=80, 90 70 80
6 70	Head=90, Tail=80, 90 80
6 70	Value Not found. Head=90, Tail=80, 90 80
3 100 90	Head=90, Tail=80, 90 100 80

Note: You must follow the prescribed input-output format. Otherwise, 50% marks will be discarded.

Task 2

- Satisfy the requirements of Task 1 using a **'Doubly linked list'**.
- One additional requirement is that you must print the linked list twice after each operation:
  - From head to tail.
  - From the tail towards the head (don't use recursive implementation; rather, utilize the 'previous' pointers).

The function that needs to be implemented -

- **Insert_front**(int key):
  - Insert the element with the '*key*' at the beginning of the list.
  - Time Complexity: O(1)
- **Insert_back**(int key):
  - Insert the element with the '*key*' at the end of the list.
  - **Time Complexity: O(1)**
- **Insert_after_node** (int key, int v):
  - Insert a node with the '*key*' after the node containing the value '*v*' if it exists. (shows error message otherwise).
  - Time complexity: $O(n)$
- **Update_node** (int key, int v):
  - Looks for the node with value *v* and updates it with the new value 'key' (error message if the node doesn't exist)
  - Time complexity: O(n)
- **Remove_head** ():
  - Remove the first node from the linked list.
  - Time complexity: O(1)
- **Remove_element** (int key):
  - Removes the node containing the 'key' if it exists (else throw an error message).
  - Time complexity O(n)
- **Remove_end** ():
  - Remove the last node from the linked list.
  - Time complexity: O(1)

Input format:

- The program will offer the user the following operations (as long as the user doesn't use option 7):
  - INSERT_FRONT to insert at front
  - INSERT_BACK to insert at back
  - INSERT_AFTER to insert after a node
  - UPDATE to update a node
  - REMOVE_HEAD 5 to remove the first node
  - REMOVE to remove a node
  - REMOVE_TAIL to remove the last node
  - EXIT to exit.
- After the user chooses an operation, the program takes necessary actions (or asks for further info if required).

Output format:

- After each operation, the status of the linked list is printed with the head and tail nodes.

Sample input	Sample Output
INSERT_FRONT 10	10 (HEAD) (TAIL)
REMOVE_TAIL	Empty NULL (HEAD) (TAIL)
REMOVE_TAIL	Underflow Empty NULL (HEAD) (TAIL)
REMOVE 10	Value Not found Empty NULL (HEAD) (TAIL)
REMOVE_HEAD	Underflow Empty NULL (HEAD) (TAIL)
REMOVE_HEAD	Underflow Empty NULL (HEAD) (TAIL)
INSERT_BACK 20	20 (HEAD) (TAIL)
INSERT_FRONT 30	30 (HEAD) -> 20 (TAIL)
INSERT_BACK 40	30 (HEAD) -> 20 -> 40 (TAIL)
INSERT_AFTER 50 20	30 (HEAD) -> 20 -> 50 -> 40 (TAIL)
INSERT_AFTER 60 40	30 (HEAD) -> 20 -> 50 -> 40 -> 60 (TAIL)
REMOVE_HEAD	20 (HEAD) -> 50 -> 40 -> 60 (TAIL)
REMOVE_TAIL	20 (HEAD) -> 50 -> 40 (TAIL)
UPDATE 70 50	20 (HEAD) -> 70 -> 40 (TAIL)
UPDATE 80 50	Value Not found 20 (HEAD) -> 70 -> 40 (TAIL)
UPDATE 80 40	20 (HEAD) -> 70 -> 80 (TAIL)
UPDATE 90 20	90 (HEAD) -> 70 -> 80 (TAIL)
REMOVE 70	90 (HEAD) -> 80 (TAIL)
REMOVE 70	Value Not found. 90 (HEAD) -> 80 (TAIL)
INSERT_AFTER 100 90	90 (HEAD) -> 100 -> 80 (TAIL)

Note: You must follow the prescribed input-output format. Otherwise, 50% marks will be

Task 5

Given two sorted linked lists in increasing order, your task is to create a **new linked list** that stores the item that intersects in both lists.

The first two lines represent the two lists (input stops with -1). Store them in a linked list, and show their intersection as output. Print 'empty' if there is no intersection.

Input	**Output**
1 2 3 4 5 6 -1 3 4 5 7 8 -1	3 4 5
10 20 30 40 50 60 70 80 90 -1 1 5 10 15 30 25 40 43 77 80 99 -1	10 30 40 80
2 2 3 3 3 5 5 5 5 5 9 9 9 -1 0 1 2 3 4 5 6 7 -1	2 3 5
1 2 3 4 5 -1 6 7 8 9 -1	Empty
6 7 8 9 -1 1 2 3 4 5 -1	Empty

Task 6 Implementing the basic operations of **Stack with Linked lists**

Stacks is a linear data structure that follows the Last In First Out (LIFO) principle. The last item to be inserted is the first one to be deleted. For example, you have a stack of trays on a table. The tray at the top of the stack is the first item to be moved if you require a tray from that stack.

The Insertion and Deletion of an element from the stack are a little bit different from the traditional operation. We define the two corresponding operations as Push() and Pop() from the stack.

The first line contains *N* representing the size of the stack. The lines contain the 'function IDs' and the required parameter (if applicable). Function ID 1, 2, 3, 4, 5, and 6 corresponds to push, pop, isEmpty, isFull, size, and top. The return type of inEmpty and isFull is Boolean. Stop taking input once given -1.

Input	**Output**
10 3 2 1 10 1 20 5 1 30 6 2 1 40 1 50 4 1 60 4 5 1 70 5 2 6 -1	 True Underflow 10 10 20 2 10 20 30 30 10 20 10 20 40 10 20 40 50 False 10 20 40 50 60 False 5 10 20 40 50 60 70 6 10 20 40 50 60 60

Task 3

Implement a ***'Deque'*** using ***'Double linked list'***. Your program should offer the user the following options:

1. void **push_front**(int key): Insert an element at the beginning of the list.
2. void **push_back**(int key): Insert an element at the end of the list.
3. int **pop_front**(): Extracts the first element from the list.
4. int **pop_back**(): Extracts the last element from the list.
5. int **size()**: Returns the total number of items in the Deque.

Note:

- The maximum time complexity for any operations is **O(1)**.
- For options 3,4: the program shows an error message if the list is empty.

Input format:

- The program will offer the user the following operations (as long as the user doesn't use option 6):
  - Press 1 to push_front
  - Press 2 to push_back
  - Press 3 to pop_front
  - Press 4 to pop_back
  - Press 5 for size
  - Press 6 to exit.
- After the user chooses an operation, the program takes necessary actions (or asks for further values if required).

Output format:

- After each operation, the status of the list is printed.

Input	**Output**
1 10	10
1 20	20 10
2 30	20 10 30
5	3
2 40	20 10 30 40
3	10 30 40
1 50	50 10 30 40
4	50 10 30
5	3

Task 4

A set of sorted numbers is stored in a linked list. Your task is to keep the first occurrence of a number and remove the other duplicate values from the list.
(Stop taking input when the user enters -1)

Input	Output
2 7 7 10 12 18 25 25 25 27 -1	2 7 10 12 18 25 27
5 5 5 5 5 -1	5
1 2 3 4 5 -1	1 2 3 4 5
10 20 20 20 20 20 20 -1	10 20

Note: Your solution should remove the node from the existing linked list instead of using a new linked list to store unique elements.

Task 8

You are given a singly linked list. Your task is to rearrange the nodes so that all nodes at odd indices come first, followed by those at even indices. The first node is considered odd, the second is even, and so forth.

Ensure that the relative order of nodes within the odd and even groups remains unchanged from the original list.

Your solution should achieve this with O(1) extra space complexity and **O(n) time complexity**.

Input:

A linked list

Output:

Linked list with elements rearranged

Input	**Output**
1 2 3 4 5 NULL	1 3 5 2 4 NULL
2 1 3 5 6 4 7 NULL	2 3 6 7 1 5 4 NULL
5 2 6 1 9 3 NULL	5 6 9 2 1 3 NULL

Task 9 Implementing the basic operations of **Queue with Linked lists**

Queue is a linear data structure that follows the First In First Out (FIFO) principle. The first item to be inserted is the first one to be removed. The Insertion and Deletion of an element from a queue are defined as EnQueue() and DeQueue().

The first line contains *N* representing the size of a **Queue**. The lines contain the 'function IDs' and the required parameter (if applicable). Function ID 1, 2, 3, 4, 5, and 6 correspond to EnQueue, DeQueue, isEmpty, isFull (assume the max size of Queue=5), size, and front. The return type of inEmpty and isFull is Boolean. Stop taking input once given -1.

Input	**Output**
5 3 2 1 10 1 20 5 1 30 6 2 1 40 1 50 4 1 60 4 5 1 60 5 2 6 -1	 isEmpty: True DeQueue: Underflow EnQueue: 10 EnQueue: 10 20 Size: 2 EnQueue: 10 20 30 Front: 10 DeQueue: 20 30 EnQueue: 20 30 40 EnQueue: 20 30 40 50 isFull: False EnQueue: 20 30 40 50 60 isFull: True Size: 5 EnQueue: Overflow Size: 5 DeQueue: 30 40 50 60 Front: 30 Exit