NATIONAL INSTITUTE OF TECHNOLOGY SIKKIM
Department of Computer Science and Engineering

CS13201 Data Structure and Algorithms Laboratory

Odd Semester, July–December 2024

Laboratory Assignment #4

NB: Solve the following programming problems according to instructions given during the laboratory sessions. Along with the requirements mentioned in the text of a question, additional instructions may be given by the course instructor to satisfy the requirements of the problem. All programs should be written in the C programming language. Do not use any external variable unless allowed by the instructor. Do not use any header other than *stdio.h* and *stdlib.h*.

1. Write a program to create a singly linked list (a chain) and then display the odd numbers and the even numbers of the list in two separate lines. Create a linked list L containing n nodes, where each node contains one data item called *key* which stores an integer, by taking n and the values of *key* for each node of L as input from the user. Print all the *key* values of L in the correct order, from the first node to the last node, in the 1st line, all the odd key values of L in the 2nd line, and all the even key values of L in the 3rd line of the final output of your program. Display the output in the format given below.

   Sample Final Output

   Linked list: 20 → 14 → 15 → 35 → 25 → 56 → 98 → 45 → 17 → 40

   Odd values: 15, 35, 25, 45, 17

   Even values: 20, 14, 56, 98, 40

2. Write a program to print the nodes located at odd positions of a singly linked list. Take a singly linked list L as input from the user, print all the nodes of L, and then print only the 1^{st} node of L, 3^{rd} node of L, 5^{th} node of L, and so on.

3. Write a program to swap the first node, called the head, and last node, called the tail, of a singly linked list L with a function called **swapHeadAndTail**. **swapHeadAndTail** will take a pointer to the first node of L as an argument, swap the head and the tail of L, and return a pointer to the first node of the modified L. Take L as input from the user, and print L both before and after modification as the final output of your program.

4. Modify the program from Question no. 3 above to make L a doubly linked list. All other requirements for the problem remain the same.

5. Write a program to implement the operations given below on a given singly linked list L, where each node of L contains one data item called *key*. Each operation should be implemented with a function according to the abstract definition of the operation given below, where L is a singly linked list, n is a node that can be a member of L, and the *key* contained in any node n_i is denoted as $n_i.key$. After initially creating L with m nodes by taking m and the *key* values of L as input from the user, the user will be shown a menu-based interface where the user will choose to execute an operation. The menu will be repeatedly displayed until the user chooses to quit. The first node of L is called the head, and the last node of L is called the tail.

   (i) **search**(L, k) := Searches for the value k in L; returns n if n is the first occurrence of a node in L such that $n.key = k$; returns NIL if there is no such node in L.

   (ii) **count**(L) := Returns the number of nodes in L.

   (iii) **addAtHead**(L, n) := Adds a node n at the beginning of L, making n the new head of L.

   (iv) **addAtTail**(L, n) := Adds a node n at the end of L, making n the new tail of L.

   (v) **insert**(L, n, p) := Inserts a node n in L at position p.

   (vi) **deleteByPosition**(L, p) := Removes node n from position p of L, and frees the storage held by n.

(vii) **deleteByValue**(L, k) := Removes every node n_i from L where $n_i.key = k$, and frees the storage held by the removed nodes.

(viii) **maximum**(L) := Returns the node with the largest *key* value in L (returns the first such node if multple nodes store the largest *key* value in L).

(ix) **minimum**(L) := Returns the node with the smallest *key* value in L (similar to **maximum**).

(x) **empty**(L) := Removes all nodes of L, and frees the storage held by the removed nodes.

(xi) **print**(L) := Prints the *key* values of L elegantly in the correct order, from head to tail.

Also, define the following function (abstract definition given) to create a node, and call this function from other functions whenever a new node needs to be created.

- **createNode**(k) := creates and returns a new node n with $n.key = k$; returns NIL if new node could not be created.

6. Write a program to implement a stack with a singly linked list. Given below are abstract definitions of functions that you need to implement, where S is a stack, x is an element that can be stored in S, *top* is the end of stack where push and pop operations are allowed, *bottom* is the other end of the stack. Each node of the linked list contains a data item which represents an element of the stack. After initially creating a stack S with k elements and with an upper limit on the number of elements as *max*, taking k, *max*, and the elements of S as input from the user, your program will show a menu-based interface, similar to Question no. 5, where the user can choose from the following options: i) Push, ii) Pop, iii) Display, iv) Quit. The menu will be displayed repeatedly until the user chooses option (iv). The **push** and **pop** functions must call the functions **isFull** and **isEmpty**, respectively, to check the overflow and underflow conditions, for which appropriate messages should be displayed. Note that deleting a node of the linked list means removing the node from the linked list and then freeing the storage held by the removed node.

   isEmpty(S) := returns True if S is empty, otherwise returns False.

   isFull(S) := returns True if S is full (i.e. S contains the maximum number of elements), otherwise returns False.

   push(S, x) := pushes x to S.

   pop(S) := pops an element x from S and returns x, i.e. returns the popped data item after deleting the popped node.

   display(S) := prints all elements of S elegantly from *top* to *bottom* (clearly showing *top*).

7. Modify the program of Question no. 6 above to make the linked list a doubly linked list, i.e. implement the same functions, as given in Question no. 6, with a doubly linked list L, where each node of L contains one data item representing an element of a stack S. All other details remain the same (re-read Question no. 6).

8. Write a program to implement a queue with a doubly linked list. Given below are abstract definitions of functions that you need to implement, where Q is a queue, x is an element that can be stored in Q, *front* is the end of Q where only the delete operation is allowed, *rear* is the other end of Q where only the add operation is allowed. Each node of the linked list contains a data item which represents an element of the queue. After initially creating a queue Q with k elements and with maximum size *max*, taking k, *max*, and the elements of Q as input from the user, your program will show a menu-based interface, similar to Question no. 5, where the user can choose from the following options: i) Add, ii) Delete, iii) Display, iv) Quit. The menu will be displayed repeatedly until the user chooses option (iv). The **add** and **delete** functions must call the functions **isFull** and **isEmpty**, respectively, to check the overflow and underflow conditions, for which appropriate messages should be displayed.

   isEmpty(Q) := returns True if Q is empty, otherwise returns False.

   isFull(Q) := returns True if Q is full (i.e. Q contains the maximum number of elements), otherwise returns False.

add(Q, x) := adds x to the rear of Q.

delete(Q) := deletes an element x from the front of Q and returns x, i.e. removes from Q the node containing x, frees the storage held by the removed node, and returns x.

display(Q) := prints all elements of Q elegantly from *front* to *rear* (clearly showing *front* and *rear*).

9. Write a program to delete the middle node of a doubly linked list, i.e. given a doubly linked list L of n nodes, delete the $\lceil \frac{n}{2} \rceil^{\text{th}}$ node of L. Create a doubly linked list L of n nodes by taking n and the data items of every node of L as input from the user. Display the nodes of L from head to tail before and after deleting the middle node as shown in the sample final output given below.

   Sample Final Output

   Original linked list: 50 ↔ 40 ↔ 20 ↔ 15 ↔ 10

   Modified linked list: 50 ↔ 40 ↔ 15 ↔ 10

10. Write a program to delete every alternate node of a doubly linked list L, starting with the 2nd node. After taking L as input from the user, modify L so that the nodes at even positions, i.e. 2nd node, 4th node, 6th node, and so on, of L are deleted. Assume that every node of L contains one data item. Note that deleting a node of a linked list means removing the node from the linked list and then freeing the storage held by that node. Display L both before and after the modification as the final output of your program.

    Sample Final Output

    Original list: 40 ↔ 30 ↔ 10 ↔ 20 ↔ 50 ↔ 25 ↔ 70

    Modified list: 40 ↔ 10 ↔ 50 ↔ 70

11. Modify the program of Question no. 10 above to make L a singly linked list. All other requirements for the problem remain the same.

12. Write a program to invert/reverse a singly linked list. To elaborate, given a singly linked list L with k nodes, where each node n_i contains one data item called *key*, and one pointer called *next* denoted as n_i.*next*, and where n_1 is the first node, called the head, and n_k is the last node, called the tail, with n_i.*next* pointing to n_{i+1}, for $i = 1, 2, \ldots k\text{-}1$, and n_k.*next* pointing to NIL, you need to modify L such that n_k becomes the new head, n_1 becomes the new tail, with n_i.*next* pointing to n_{i-1} for $i = 2, 3, \ldots k$, and n_1.*next* pointing to NIL. Take the original list L as input from the user, and then display both the original and the modified list elegantly as the final output of your program. Do not modify the key value of any node; you need to only modify the links of the nodes. The sample final output given below is self-explanatory.

    Sample Final Output

    Original list: 40 → 30 → 10 → 20 → 50 → 25 → 70

    Modified list: 70 → 25 → 50 → 20 → 10 → 30 → 40

13. Write a program to implement search and display operations, as defined below, on a doubly linked circular list. Take as input from the user a doubly linked circular list L of k nodes where every node contains the following attributes: i) *roll_number* which should be unique for each node, ii) *name* of student, and iii) *academic_stream* of student. Let the oldest node of L be denoted *head*, and the newest node of L be denoted *tail*, and every new node *nw* be added to L at the position after *tail*, i.e. the *next* pointer of *tail* will be made to point to *nw*, before *tail* is updated. After creating L, print elegantly the attributes of all nodes of L from *head* to *tail*, and then implement the following functions (abstract definitions given) with a menu-based interface, where n is a node in a doubly linked circular list L, r is a value of a type the same as that of *roll_number*, “forward direction” means the direction in which the *next* pointer of each node of L points, and “backward direction” means the direction in which the *previous* pointer of each node of L points. The menu is displays repeatedly in a loop until the user chooses to quit.

i) **search**(r, L) := Returns the node n in L where $n.roll\_number = r$ if such a node exists in L, otherwise returns NIL.

ii) **displayFrom**(r, L) := Prints the attributes of all nodes of L starting from the node n where $n.roll\_number = r$ and ending at the node m where m is the previous node of n, traversing the circular list L in the forward direction. Displays nothing if **search**(r, L) is NIL.

iii) **displayFrom_InReverse**(r, L) := Prints the attributes of all nodes of L starting from the node n where $n.roll\_number = r$ and ending at the node q where q is the next node of n, traversing the circular list L in the backward direction. Prints nothing if **search**(r, L) is NIL.

A representation of the results of the three aforementioned functions is given below.

If L is represented by the following arrangement of nodes, where n_1 is the next node of n_5 and n_5 is the previous node of n_1,

$\ldots \leftrightarrow n_1 \leftrightarrow n_2 \leftrightarrow n_3 \leftrightarrow n_4 \leftrightarrow n_5 \leftrightarrow \ldots$

and $n_3.roll\_number = r$, then

search(r, L) returns n_3

and

displayFrom(r, L) prints

n_3, n_4, n_5, n_1, n_2

and

displayFrom_InReverse(r, L) prints

n_3, n_2, n_1, n_5, n_4

14. Write a program to print the nodes of a singly linked list in reverse order (from tail to head), with a non-recursive function as well as with a recursive function. Define two functions, **printTailToHead** and **printTailToHeadRec**. **printTailToHead** will take only one parameter, a pointer to the head of a singly linked list L, and will print the key values of the nodes of L from tail to head, without any recursion (hint: use a stack). **printTailToHeadRec** does the same task as the other function but with recursion. Take a singly linked list L as input from the user, print L from head to tail, and then print L in reverse with both the aforesaid functions as the final output of your program.

15. Write a function called **merge** to merge two sorted singly linked lists L_1 and L_2 into one sorted singly linked list M. Assume that the nodes of L_1, L_2, and M are sorted in ascending order with repect to the nodes' key values. **merge** will take two arguments: i) $head_1$, a pointer to the first node of L_1, and ii) $head_2$, a pointer to the first node of L_2. After merging L_1 and L_2 into a sorted list M, **merge** will return a pointer to the first node of M. Take L_1 and L_2 as input from the user in your **main** function, print L_1 and L_2, and call **merge** from your **main** function to produce M. Print M as the final output of your program. Note that M will be made up of the nodes of L_1 and L_2 (and, L_1 and L_2 will become empty), and no new node should be created for M.

16. Modify the program of Question no. 15 above to make L_1, L_2, and M doubly linked lists. All other requirements for the problem remain the same.

17. Write a program to sort in ascending order the nodes of a doubly linked list L with respect to the key values of L's nodes by insertion sort. Write two functions called **remove** and **insert**, as defined below (with abstract definitions), which you will call in your implementation of insertion sort on L to place the nodes of L at their correct positions as you traverse L. Create L by taking the key values of L as input from the user, print the unmodified L, sort the nodes of L by insertion sort, and then print the sorted L as the final output of your program. In the function definitions below, p and q are pointers to nodes of a doubly linked list.

remove(p, L) := removes the node p from the doubly linked list L.

insert(p, q, L) := inserts the node p before the node q in the doubly linked list L (where q is a node of L)

Note that no node's key value may be changed; the nodes will be rearranged by changing their links.

18. Write a program to store the letters of a word w in a doubly linked list L, where w is entered as a command-line argument, and then remove nodes of L and re-link them to create two doubly linked lists C and V containing consonants and vowels of w respectively. All the letters of w will be stored in the nodes of L in the correct sequence where every node of L stores one letter of w. The nodes of L containing consonants and vowels will then be removed from L and then be re-linked to create linked lists C and V, where C is made up of nodes containing consonants and V is made up of nodes containing vowels. Print L before modification, and then print C and V as the final output of your program. Do not create new nodes to make C and V; make C and V with the removed nodes of L.

    Sample Execution

    ```
    $ ./myProgram.out Computer-Networks ↵
    L: C o m p u t e r – N e t w o r k s
    C: C m p t r N t w r k s
    V: o u e e o
    ```

19. Write a program to enter a sentence through the command line, store the sentence in a singly linked list L, and then delete the nodes of L that contain a given word. To elaborate, after taking a sentence s as input through the command line, store the words of s in a singly linked list L in the correct order such that every node of L stores one word. Then, take a word w as input from the user, and delete all the nodes of L that contain w. Deleting a node means removing the node from the linked list and then freeing the storage held by the removed node. Print L both before and after modification.

    Sample Execution

    ```
    $ ./myProgram.out Tom chases Jerry and then Jerry eats cheese. ↵
    L: Tom chases Jerry and then Jerry eats cheese.
    Word to delete: Jerry ↵
    L: Tom chases and then eats cheese.
    ```

20. Write a program to create the game of Tower of Hanoi with doubly-linked lists. The game consists of three stacks, say s_1, s_2, s_3, where every stack has its elements arranged in a non-decreasing order from top to bottom, i.e. in any stack s, a push operation can be performed only if the element x to be pushed to s is less than or equal to the top element t of s (i.e. if $x > t$, then push is not allowed in s). Initially, s_1 has n elements, and s_2 and s_3 are empty. The objective of the game is to transfer all the n elements from s_1 to s_3 by a series of moves where a move is defined as popping an element x from a stack s_i and then pushing x to another stack s_j (where, $1 \le i \le 3$, $1 \le j \le 3$, and $i \ne j$). Note that only one move can be made at a time. Your program will initially take the value of n and the elements of s_1 as input from the user, and then will repeatedly display the contents of s_1, s_2, and s_3, along with a menu where the user is given the option to make a move. Each time the menu is displayed, the user will choose the source stack for the pop operation and destination stack for the push operation for one single move. The menu and the stack contents will be elegantly displayed repeatedly until either the game is solved or the user chooses to quit. If the game is solved, your program will print the number of moves taken by the user to solve the game. Represent each of the three stacks s_1, s_2, s_3 with a doubly linked list where each node of the linked list stores a key value (of type *int*) representing one element of the stack represented by the linked list. Do not use any array.

--------------------------------------x--------------------------------------