بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ

In the name of Allah, Most Gracious, Most Merciful

CSE 4303
Data Structure

Topic:Abstract Data Type, Arrays & Linked List

Asaduzzaman Herok
Lecturer | CSE | IUT
asaduzzaman34@iut-dhaka.edu

What is Abstract Data Type?

An abstract data type (ADT) is the way we look at a data structure, focusing on what it does and ignoring how it does its job.

For example, stacks and queues are perfect examples of an ADT

Data type: Data type of a variable is the set of values that the variable can take. We have already read the basic data types in C include int, char, float, and double.

Abstract; The word ‘abstract’ in the context of data structures means considered apart from the detailed specifications or implementation.

Advantage of using ADTs: In the real world, programs evolve as a result of new requirements or constraints, so a modification to a program commonly requires a change in one or more of its data structures. So it is better to separate the use of a data structure from the details of its implementation.

Arrays

Scenario: There are 20 students in a class, we have write a program that reads and prints the marks of all the students. How can we do it?

- We need 20 integers to store and print the marks.
- One way is to have 20 integer variables with different names like, mark01, mark02, mark03…
- For reading we need 20 input (i.e scanf) statements.
- For printing we need 20 output (i.e print) statements.

Now what if we have the same scenario for a

- Entire course? (say 63 students)
- Entire 3rd semester cse+swe (180 students)
- Entire university (10,000 students)

???

Arrays

What we need is a data structure, Simplest form an Array.

- a collection of similar data elements
- elements are stored in consecutive memory locations.

1st element	2nd element	3rd element	4th element	5th element	6th element

Advantage:

- An array is a pretty obvious way to store a list, with a big advantage: it enables very fast access of each item.

Disadvantage:

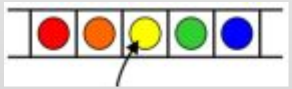
- First, if we want to insert an item at the beginning or middle of an array, we have to slide a lot of items over one place to make room. This takes time proportional to the length of the array.
- Second, an array has a fixed length that can’t be changed. If we want to add items to the list, but the array is full, we have to allocate a whole new array and move all the items from the old array to the new one.

Operations on Arrays / Linked List

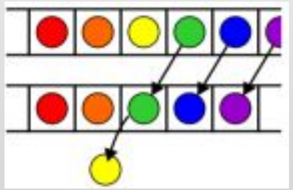
- Traversing
- Find / Search
- Insert
- Erase / Delete
- Update / Replace
- Sorting
- Merging

Operations at the kth entry of the list include:

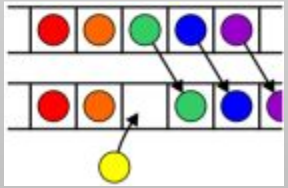
Accessing



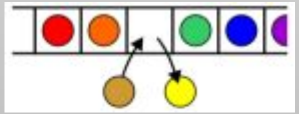
Erasing/Deleting



Inserting



Updating



Complexity of Operations on Arrays

Given a sorted array, we have the following run times:

Operation \ Position	Front / 1st	Arbitrary location in terms of particular value	Back / nth
Find	Good	Okay	Good
Insert	Bad	Bad	Good* Bad
Erase	Bad	Bad	Good

Good = Θ(1), Okay = Θ(log(n)), Bad = Θ(n)

* only if the array is not full

Complexity of Operations on Arrays

Given an unsorted array, we have the following run times:

Operation \ Position	Front / 1st	Arbitrary location in terms of particular value	Back / nth
Find	Good	Bad	Good
Insert	Bad	Bad	Good* Bad
Erase	Bad	Bad	Good

Good = Θ(1), Okay = Θ(log(n)), Bad = Θ(n)

* only if the array is not full

Linked List

Why don't we just keep using Array?

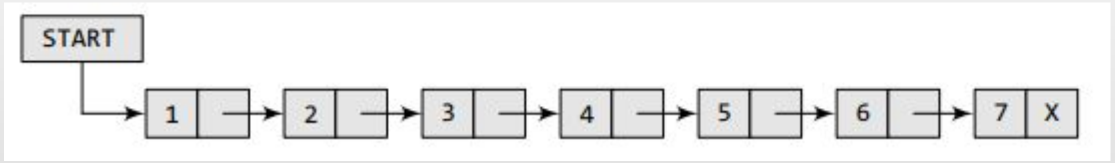
- ➢ While declaring arrays, size should be predefined.
- ➢ Extending arrays is more costlier.
  - ○ Can we just create array with large size?
  - ○ What if after declaring large size, there is not enough data to fill that size?

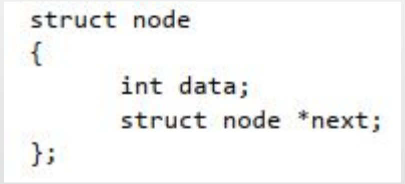
???

There must be a data structure that removes the restrictions on the maximum number of elements and the storage condition to write efficient programs … …

Linked List

A ***linked list***, in simple terms, is a linear collection of data elements. These data elements are called ***nodes***.





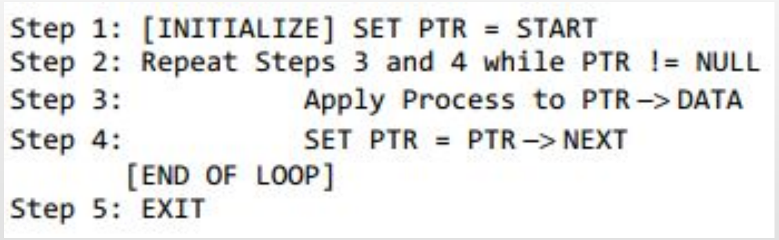
```
struct node
{
        int data;
        struct node *next;
};
```

- ➔ Linked lists are appropriate when the number of data elements to be represented in the data structure at once is unpredictable.
- ➔ Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- ➔ Each node does not necessarily follow the previous one physically in the memory.
- ➔ Linked lists can be maintained in sorted order by inserting an element at the proper point in the list.
- ➔ Can be used to implement other data structures like binary trees.

❑ However, linked lists have a big disadvantage compared to arrays. Finding the kth item of a linked list takes time proportional to k. You have to start at the head of the list and walk forward k - 1 nodes, one "next" at a time.

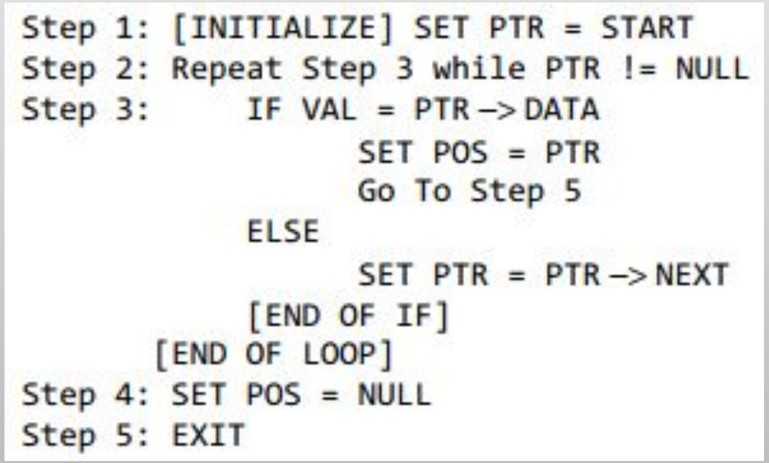
Operations on Linked List

Traversing: accessing the nodes of the list in order to perform some processing on them.



```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:             Apply Process to PTR->DATA
Step 4:             SET PTR = PTR->NEXT
        [END OF LOOP]
Step 5: EXIT
```

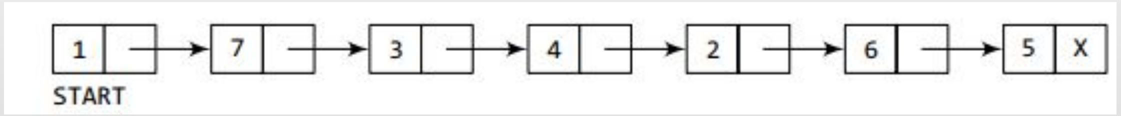
Searching: finding a particular element in the linked list. If it is present, the algorithm returns the address of the node that contains the value.



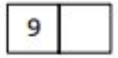
```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3:         IF VAL = PTR->DATA
                      SET POS = PTR
                      Go To Step 5
                ELSE
                      SET PTR = PTR->NEXT
                [END OF IF]
           [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

Insert Operations on Linked List

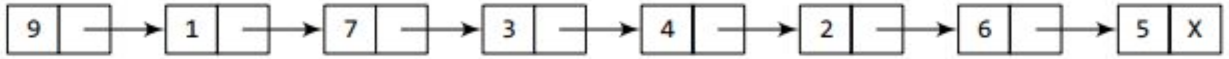
Case 1: The new node is inserted at the beginning.



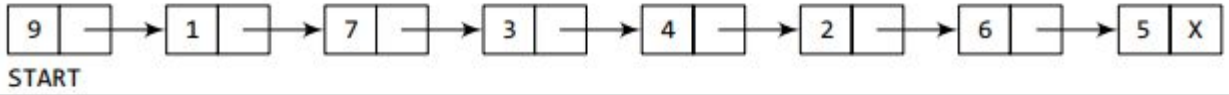
Allocate memory for the new node and initialize its DATA part to 9.



Add the new node as the first node of the list by making the NEXT part of the new node contain the address of START.



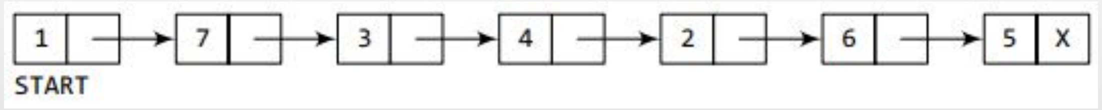
Now make START to point to the first node of the list.



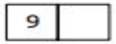
```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET START = NEW_NODE
Step 7: EXIT
```

Insert Operations on Linked List

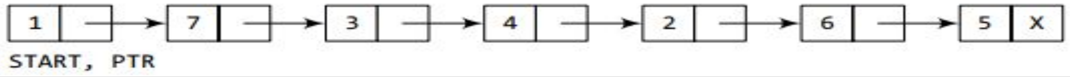
Case 2: The new node is inserted at the end.



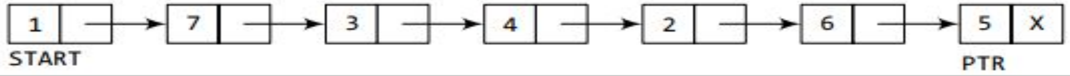
Allocate memory for the new node and initialize its DATA part to 9.



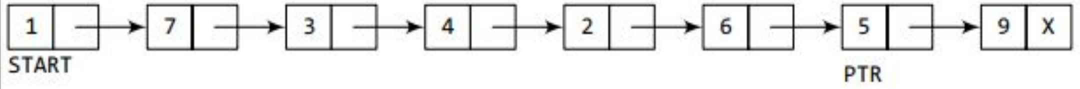
Take a pointer variable PTR which points to START.



Move PTR so that it points to the last node of the list.



Add the new node after the node pointed by PTR. This is done by storing the address of the new node in the NEXT part of PTR.

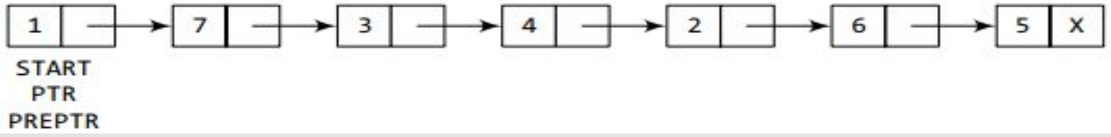


```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

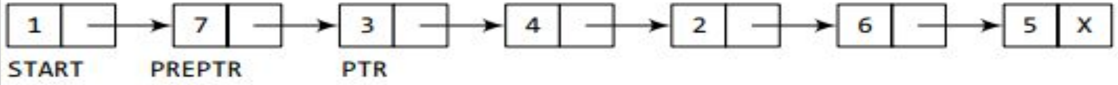
Insert Operations on Linked List

Case 3: The new node is inserted after a given node.

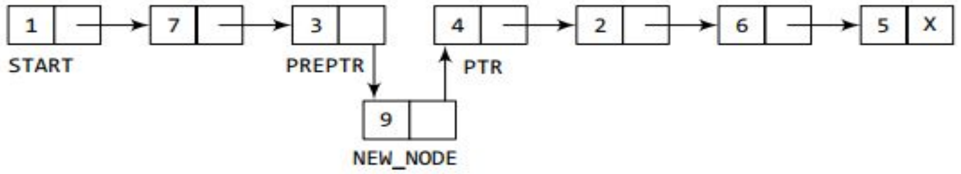
Take two pointer variables PTR and PREPTR and initialize them with START so that START, PTR, and PREPTR point to the first node of the list.

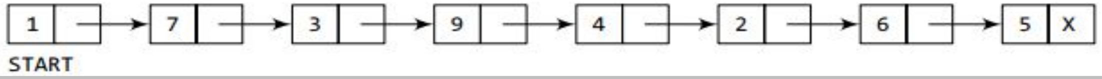


Move PTR and PREPTR until the DATA part of PREPTR = value of the node after which insertion has to be done. PREPTR will always point to the node just before PTR.



Add the new node in between the nodes pointed by PREPTR and PTR.

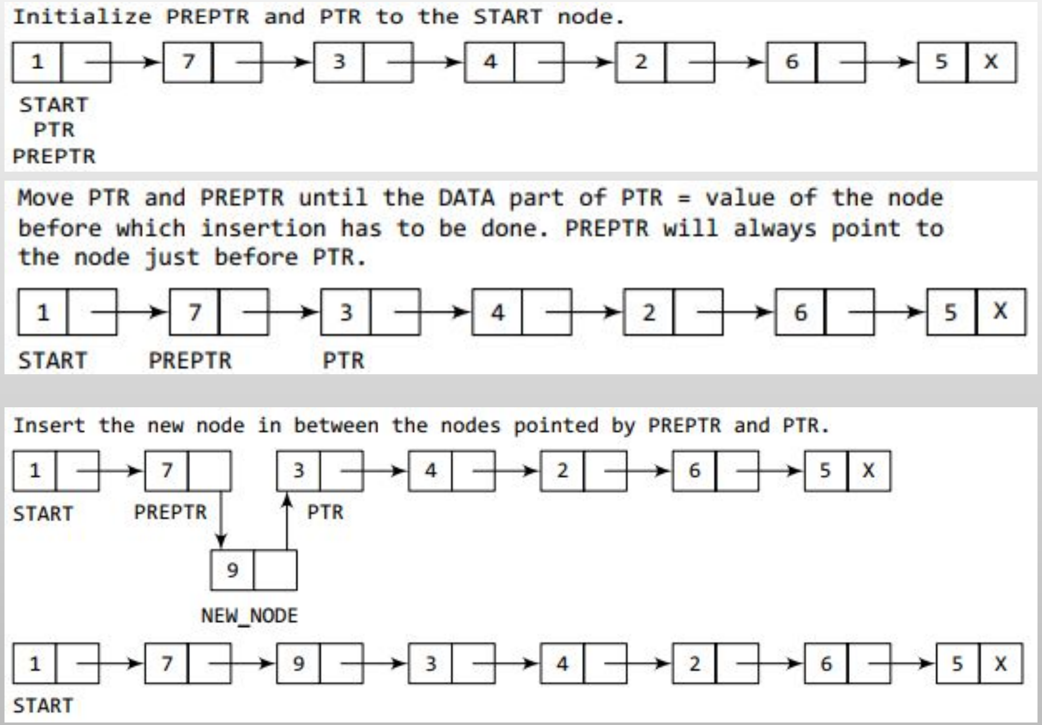




```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA
        != NUM
Step 8:     SET PREPTR = PTR
Step 9:     SET PTR = PTR -> NEXT
          [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

Insert Operations on Linked List

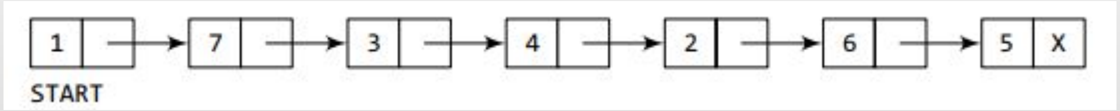
Case 4: The new node is inserted before a given node


Initialize PREPTR and PTR to the START node.
1
7
3
4
2
6
5
X
START
PTR
PREPTR
Move PTR and PREPTR until the DATA part of PTR = value of the node before which insertion has to be done. PREPTR will always point to the node just before PTR.
START
PREPTR
PTR
Insert the new node in between the nodes pointed by PREPTR and PTR.
9
NEW_NODE
START

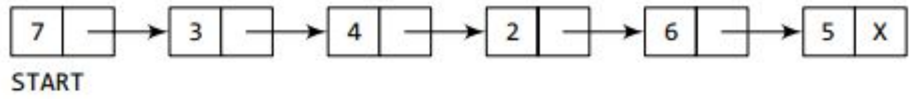
```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PTR -> DATA != NUM
Step 8:     SET PREPTR = PTR
Step 9:     SET PTR = PTR -> NEXT
         [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

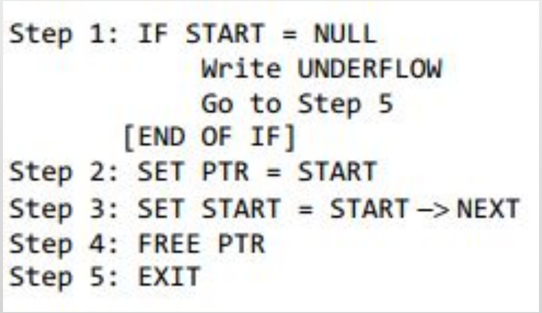
Delete Operations on Linked List

Case 1: The first node is deleted.



Make START to point to the next node in sequence.



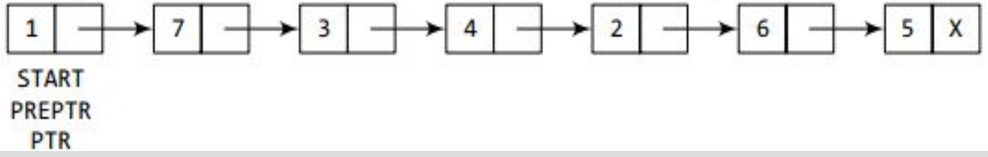


```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 5
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

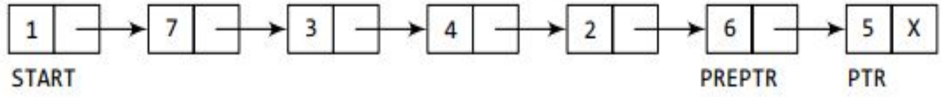
Delete Operations on Linked List

Case 2: The last node is deleted.

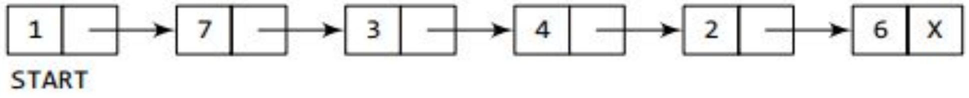
Take pointer variables PTR and PREPTR which initially point to START.



Move PTR and PREPTR such that NEXT part of PTR = NULL. PREPTR always points to the node just before the node pointed by PTR.



Set the NEXT part of PREPTR node to NULL.

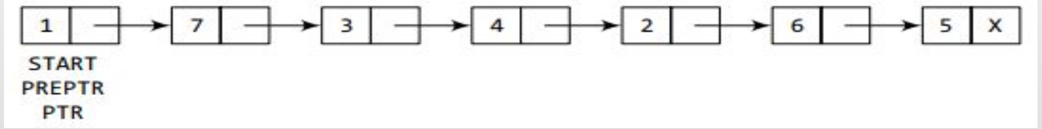


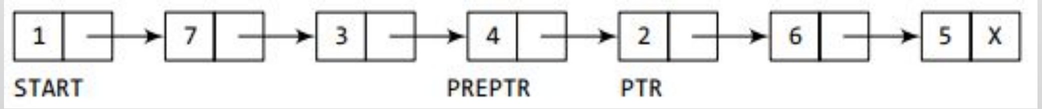
```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL
Step 4:     SET PREPTR = PTR
Step 5:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: SET PREPTR -> NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

Delete Operations on Linked List

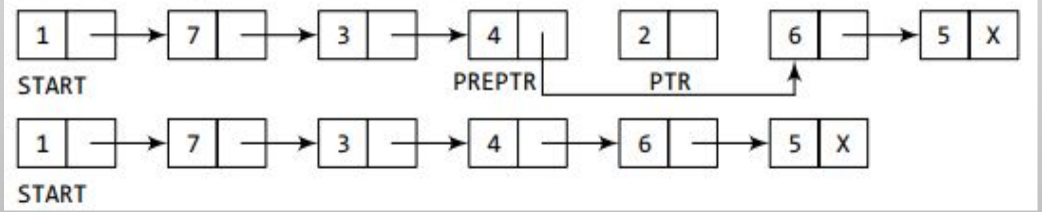
Case 3: The node after a given node is deleted

Take pointer variables PTR and PREPTR which initially point to START.





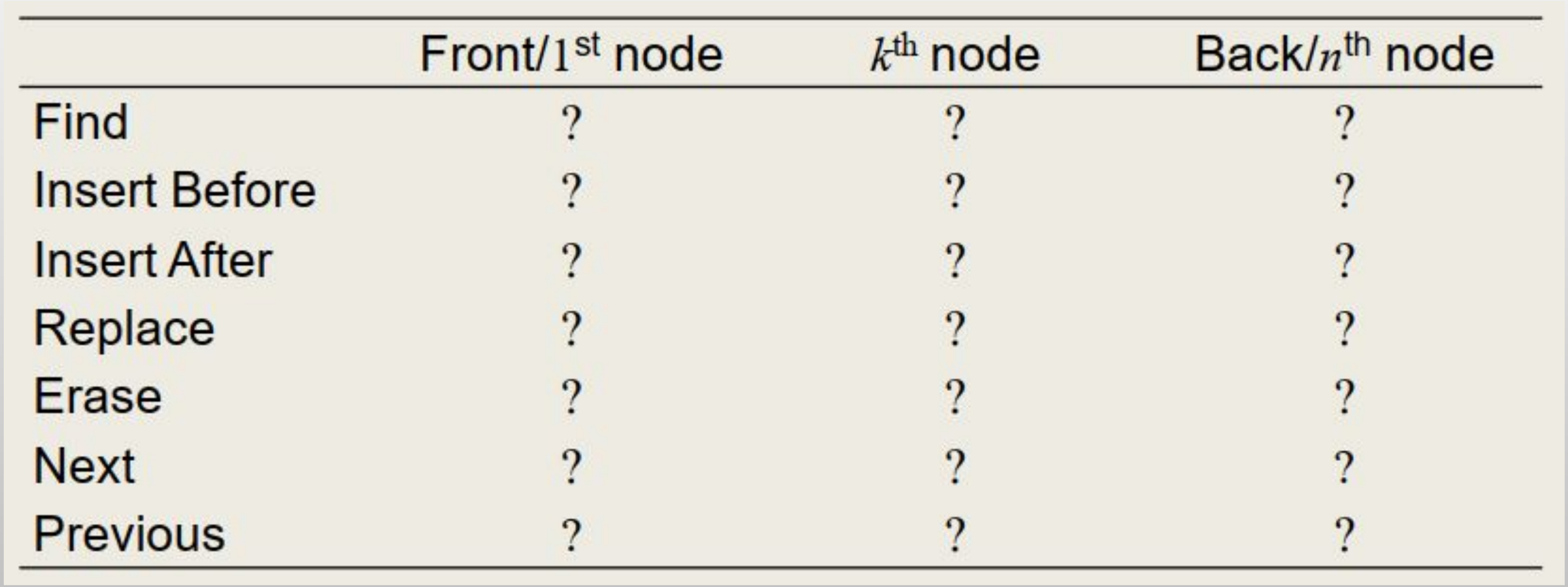
Set the NEXT part of PREPTR to the NEXT part of PTR.



```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET PREPTR = PTR
Step 4: Repeat Steps 5 and 6 while PREPTR -> DATA != NUM
Step 5:     SET PREPTR = PTR
Step 6:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 7: SET TEMP = PTR
Step 8: SET PREPTR -> NEXT = PTR -> NEXT
Step 9: FREE TEMP
Step 10: EXIT
```

Complexity of Operations on Linked List

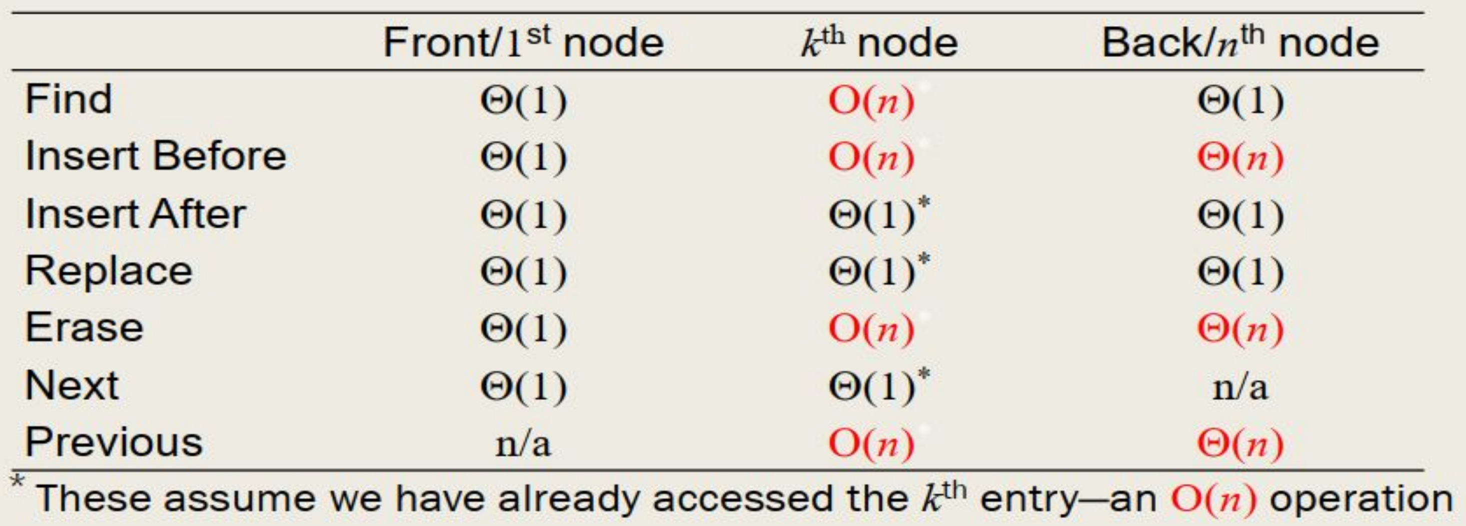
We will use the following matrix to describe operations at the locations within the structure



	Front/1^{st} node	k^{th} node	Back/n^{th} node
Find	?	?	?
Insert Before	?	?	?
Insert After	?	?	?
Replace	?	?	?
Erase	?	?	?
Next	?	?	?
Previous	?	?	?

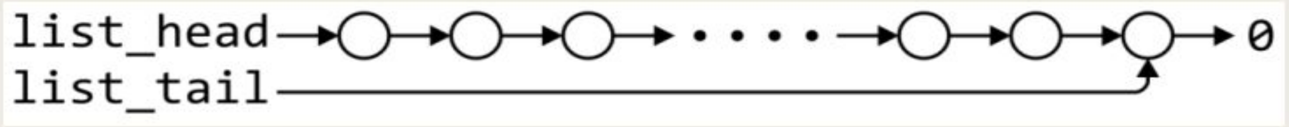
Complexity of Operations on Singly linked list

Using only a single pointer PTR. No PREPTR is used. (I repeat, Using only single pointer PTR)



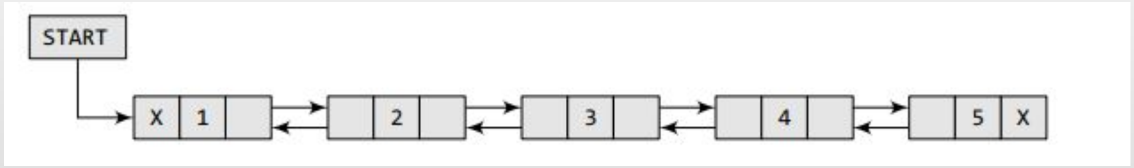
	Front/1st node	kth node	Back/nth node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$O(n)$	$\Theta(n)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$O(n)$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$\Theta(n)$

* These assume we have already accessed the kth entry—an $O(n)$ operation



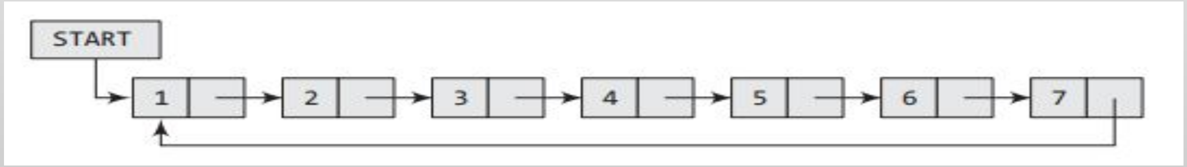
Doubly and Circular Linked List

A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence.

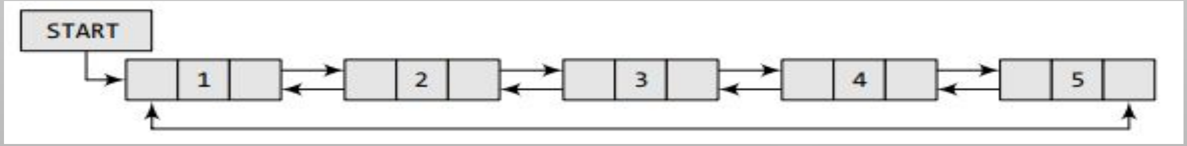


```
struct node
{
        struct node *prev;
        int data;
        struct node *next;
};
```

In a circular linked list, the last node contains a pointer to the first node of the list.



A circular doubly linked list or a circular two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence.



Pros and Cons of Circular Linked / Circular Doubly Linked Lists

Advantage

- ❑ If we are at a node, then we can go to any node. But in linear linked list it is not possible to go to previous node.
- ❑ It saves time when we have to go to the first node from the last node. It can be done in single step because there is no need to traverse the in between nodes. But in double linked list, we will have to go through in between nodes.

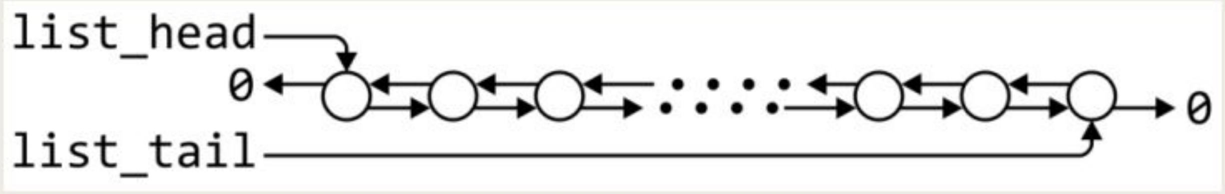
Disadvantage

- ❑ Finding end of list and loop control is harder (no NULL's to mark the end)
- ❑ If we at a node and need to go back to the previous node, then we can not do it in single step (In case of singly circular list). Instead we have to complete the entire circle by going through the in between nodes.

Complexity of Operations on Doubly linked list

	Front/1st node	kth node	Back/nth node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$\Theta(1)^*$	$\Theta(1)$

* These assume we have already accessed the kth entry—an $O(n)$ operation



Applications of Linked Lists

- ❑ Process Scheduling in Operating Systems
- ❑ Hashing
- ❑ Stack and Queue
- ❑ Graph Algorithms
- ❑ Music Player Application
- ❑ Consider the history section of web browsers, where it creates a linked list of webpages visited
- ❑ Our brain is also a good example of linked list. In the initial stages of learning something by heart, the natural process is to link one item to next. Also, when we forget something and try to remember, than our brain follows association and tries to link one memory with another and so on and we finally recall the lost memory

Acknowledgement

Rafsanjany Kushol
PhD Student, Dept. of Computing Science,
University of Alberta

Sabbir Ahmed
Assistant Professor
Department of CSE, IUT