S4 DATA STRUCTURES (DS) Code: 4133

MODULE 2 - LIST AND LINKED LIST

Syllabus

2.1 Understanding list and its operations

2.1.1 Describe list (using array) and its operations – Find, MakeEmpty, PrintList etc.
2.1.2 Describe List ADT with find(), makeEmpty(), printList(), findKth() etc.

2.2 Understanding linked list and its operations

2.2.1 Explain linked list and its operations –
Find, MakeEmpty, PrintList, FindKth, Insert, Delete, Successor, Predecessor etc.
2.2.2 Describe the methods of memory allocation and deallocation for nodes.
2.2.3 Describe LinkedList ADT with find(), makeEmpty(), printList(), findKth(), insert(), delete() etc.
2.2.4 Describe algorithm for implementing stack with LinkedList ADT.
2.2.5 Describe algorithm for implementing queue with LinkedList ADT.
2.2.6 Describe about doubly linked lists and circular linked lists.

List

List is a linear data structure with a sequence of zero or more elements.
List is an ordered collection of elements. So it is also known as ordered list or linear list.

Examples
Days of the week:
(SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY)
Months of year:
(JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER)
Boolean list:
(TRUE, FALSE)
Letters:
(A, B, C, D……Z, a, b, c, d…..z)
Digits:
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

Consider a list A_1, A_2, A_3, … A_N

- N: length of the list
- A_1: first element
- A_N: last element
- A_i: element at position i
- **If N=0, then empty list**

- Successor is the next element in sequence. The last node has no successor.
  Successor of A_1 Is A_2
  Successor of A_2 Is A_3
  Successor of A_i is A_{i+1}
  i.e. A_i precedes A_{i+1}

- Predecessor is the previous element in sequence. The first node has no predecessor.
  Predecessor of A_2 is A_1
  Predecessor of A_3 is A_2
  Predecessor of A_i is A_{i-1}
  i.e. A_i follows A_{i-1}
- A list's length is the number of elements in it. A list may be empty (contain no elements)
- Two standard implementations for the list ADT
  - ✓ Array-based
  - ✓ Linked list

Operations on Linear List

1) Create a linear list
2) Destroy a linear list
3) Check whether list is empty
4) Size of list
5) Find the element with a given index
6) Find the index of a given element
7) Delete
8) Insert
9) Traverse

* printList : print the list
* makeEmpty : create an empty list
* find, get(index) : locate the position of an object in a list
  if list is 34,12, 52, 16, 12
  find(52) $\rightarrow$ 3
* insert : insert an object to a list
  insert(4,x) $\rightarrow$ 34, 12, 52, x, 16, 12
* remove : delete an element from the list
  remove(3) $\rightarrow$ 34, 12, x, 16, 12
  Elements with higher index of given index reduce by 1.
* findKth: return the element at Kth position

List ADT

AbstractDataType *LinearList*
{
Instances
Ordered finite collection of zero or more elements
}
{
Operations
isempty() : return true if the list is empty, otherwise false.
boolean datatype
size() : return the list size.
(i.e. number of elements in the list)
find, get(index) : return the index of element in list.
(i.e. locate the position of an object in a list)
indexof(x) : return the index of x in the list.

Return -1 if x not in list

remove(index), erase(index) : remove the element at given index.

Elements with higher index of given index reduce by 1

insert(index, x) : insert element x at given index.

Element with higher index of given index increased by 1

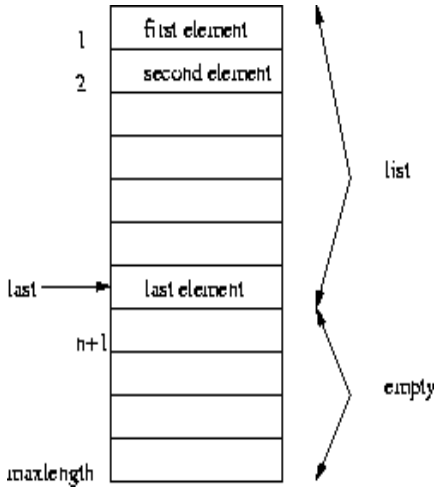
printList() : print the list

output he list elements from left to right

}

List using array

Elements are stored in continuous array positions.



Consider an Example:-

- First element of list is element[0], Second element of list is element[1]
- Last element of list is element[n-1], List (5, 2, 4, 8, 1)

Element[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
5	2	4	8	1				

insert(2, 12)

Element[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
5	2	12	4	8	1			

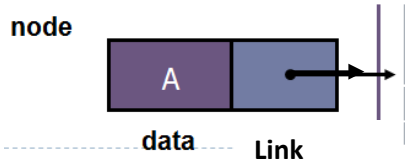
remove(2)

Element[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
5	2	4	8	1				

Insertion and deletion requires shifting of elements

Linked lists

- Linked list is a DS with linear collection of data elements with link.
- Simply, A linked list is a series of connected nodes



- Linked list can be used to implement other data structures such as stacks, queues and their types.
- Each node contains two parts:-
  1. Data (any type)
  2. Pointer to the next node in the list

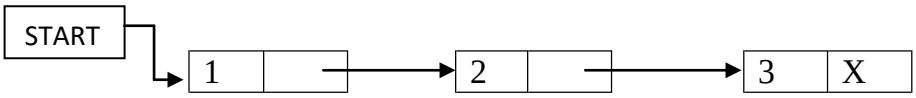


Figure: simple linked list

- The last node will have no next node and its link part will be NULL. The NULL pointer is represented by X or *nullptr*.
- Linked list contain a pointer variable *START* that stores the address of the **first node in the list**.
- We can traverse the entire list using a single pointer variable called *START.*
- If START=NULL, then the linked list is empty and contains no nodes.
- The figure shows below represents linked list in the memory

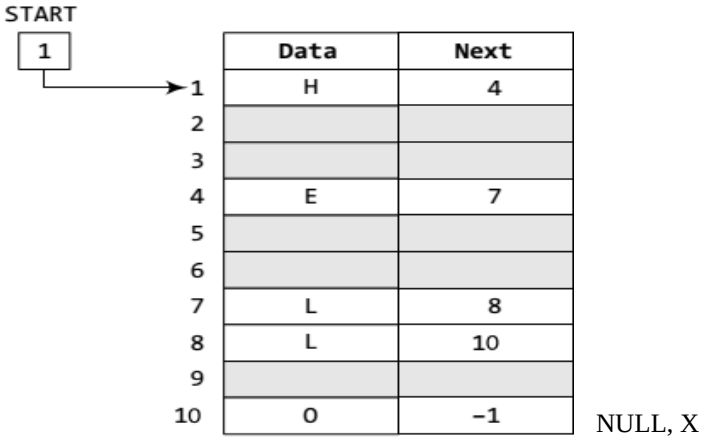


Figure 6.2 START pointing to the first element of the linked list in the memory

If start=1, the first data stored at address=1 which is H. The corresponding the NEXT stores the address of the next node, which is 4. The data element at address 4 is E. If the NEXT pointer contains -1 or NULL, this means that the end of the list.

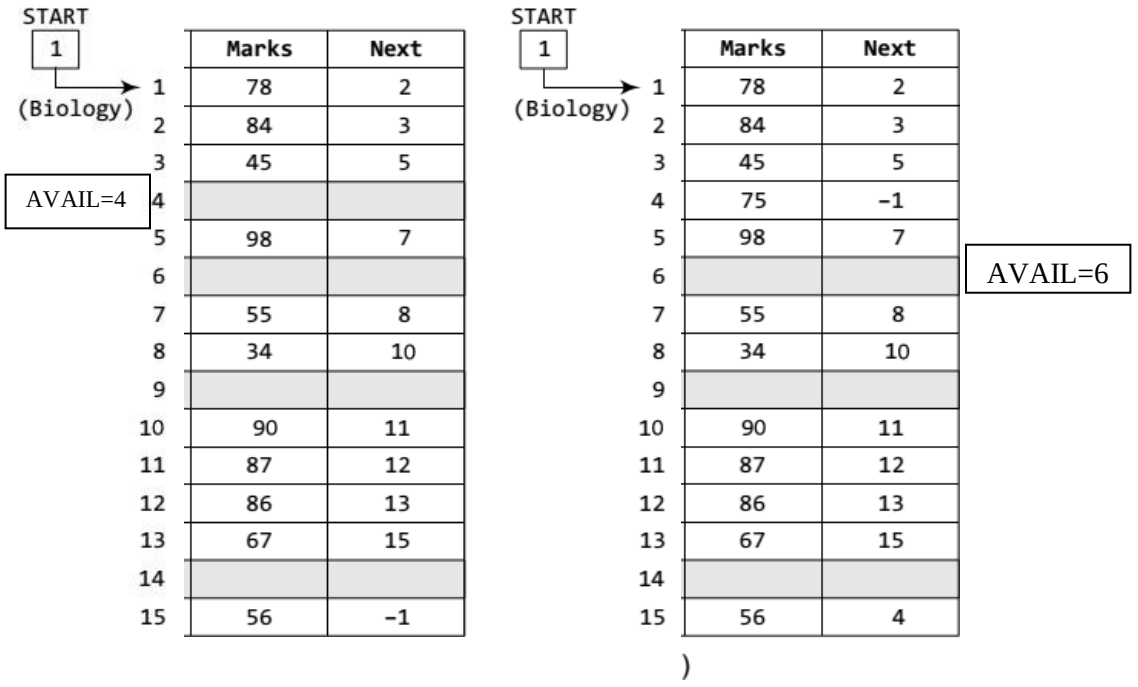
- Advantage : provides quick insert and delete operations
- Disadvantage: slow search operation and requires more memory space.
- Limitations:
  1. The number of nodes added to the list is limited only by the amount of memory available.
  2. It is very difficult to searching.

- **Comparison- Liked list and array**

Arrays	Linked list
1. Linear collection of data elements **2.** Data stored in **consecutive memory location** 3. Data can access randomly 4. An array can add a **limited number of elements** (Example: int a[20], can add 20 elements only) 5. No need of link for insertion and deletion. 6. An array is a **static** data structure. 7. This means the length of array cannot be altered.	**1.** Linear collection of **nodes** 2. Does not store nodes in consecutive memory location 3. It does not allow random access of data. Nodes can be accessed in a sequential manner. 4. We can add **any number** of elements in the list 5. Insertion and deletion can be done at any point. But change in link. 6. A linked list is a **dynamic** data structure. 7. Extra memory is needed to store the address of nodes.

Memory allocation and de-allocation for linked list (node)

NEXT field stores the address of the next node.
Grey shaded portion shows free space, and thus we have 4 memory locations available.



)

Figure 6.5 (a) Students linked list (b) linked list after the insertion of new student mark.

- If we want to insert a node to an already existing linked list in memory, **first find free space** in the memory and then use it to store the information.
- The computer maintains a list of all the free memory cells. This list is of available space is called the ***free pool***.
- Every linked list has a pointer variable START which stores the **address of the first node** of the list.
- Another pointer variable *AVAIL* which stores the **address of the first free space**.

- **Garbage collection**
  The operating system scans through all the memory cells and marks those cells that are being used by some other programs.
  Then, it collects all the cells which are not being used and adds their address to the free pool. So that it can be **reused by other programs. This process is called garbage collection**. The whole process of collecting unused memory cells is transparent to the programmer.

Different types of linked lists

1. Singly linked list
2. Circular linked list
3. Doubly linked list
4. Circular doubly linked list
5. Header linked list

1. Singly Linked List (Linked List) and its Operations

1. It is simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.
2. It allows traversal of data only in one way.

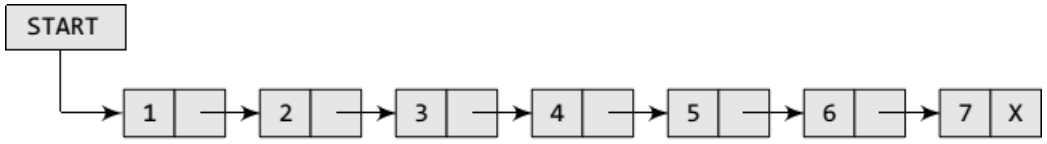


Figure 6.7 Singly linked list

Linked lists provide an efficient way of storing related data and perform basic operations such as **traverse, search, insert, delete of data**. But extra space required for storing address of the next node.

That is in detail:-

1) PrintList
2) Find
3) FindKth
4) Insert
5) Delete
6) MakeEmpty

7) Successor
8) Predecessor etc.

- A node's successor is the next node in the sequence
  - The last node has no successor
- A node's predecessor is the previous node in the sequence
  - The first node has no predecessor
- A list's length is the number of elements in it
  - A list may be empty (contain no elements)

I. Traversing

- Traversing means accessing (visiting) the nodes of the list in order to perform some process on them.
- The pointers used in the algorithms are START, PTR (which points to the node that is currently being accessed)

Algorithm for traversing a linked list

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
Step 5: exit

Algorithm to **print each node** of linked list

Step 1: (Initialize) SET PTR=START
Step 2: Repeat steps 3 and 4 while PTR!=NULL
Step 3: **print PTR->DATA**
Step 4: SET PTR=PTR->NEXT
Step 5: exit

Algorithm to **print number of nodes** in the linked list

Step 1: (Initialize) SET Count=0
Step 2: (Initialize) SET PTR=START
Step 3: Repeat steps 4 and 5 while PTR!=NULL
Step 4: **SET Count=Count+1**
Step 5: **SET PTR=PTR->NEXT**
Step 6: exit

II. Searching

- Searching means to find or search a particular element in the linked list.
- This algorithm **returns the address** of the node that contains the value.
- Two algorithms are:-
  1. Search an unsorted list
  2. Search a sorted list

1. Search an unsorted linked list

For example search value 4:-

Consider the linked list shown in Fig. 6.11. If we have VAL = 4, then the flow of the algorithm can be explained as shown in the figure.

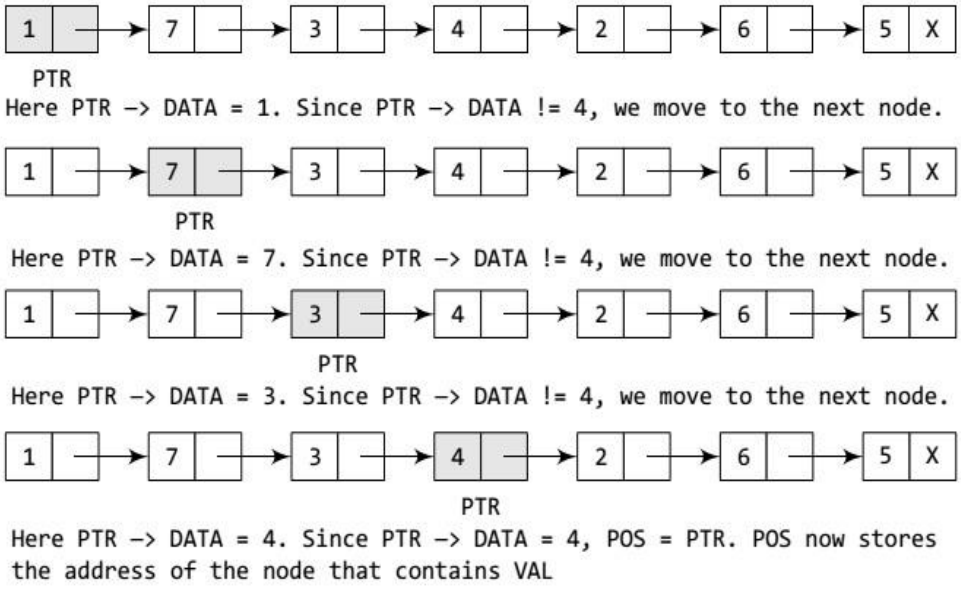


Figure 6.11 Searching a linked list

Algorithm to search an unsorted linked list (Algorithm Case 1):

Step 1: (initialize) SET PTR=START
Step 2: repeat steps3 while PTR!=NULL
Step 3: if VAL=PTR->DATA
SET POS=PTR
Print POS
Go to step 5
Else
SET PTR=PTR->NEXT
Step 4: SET POS=NULL
Print "value not present in Linked List"
Step 5: EXIT

2. Search a sorted linked list

Algorithm to search a sorted linked list (Algorithm Case 2):

Step 1: (initialize) SET PTR=START
Step 2: repeat steps3 while PTR!=NULL
Step 3: if PTR->DATA= VAL then
SET POS=PTR
Go to step 5
Else if PTR->DATA<VAL
SET PTR=PTR->NEXT

Else
Go to step 4
Step 4: SET POS=NULL
Step 5: EXIT

III. Insertion

✓ To do insertion, we will take 5 cases

1. The new node is inserted at the beginning
2. The new node is inserted at the end
3. The new node is inserted after a given node
4. The new node is inserted before a given node
5. The new node is inserted in a sorted linked list

✓ OVERFLOW is a condition occurs when AVAIL=NULL or no free memory cell. In this case, insertion not possible.

Unsorted Linked List

1. Insert at beginning

For example: Insert a new node with data 9 at beginning (first).

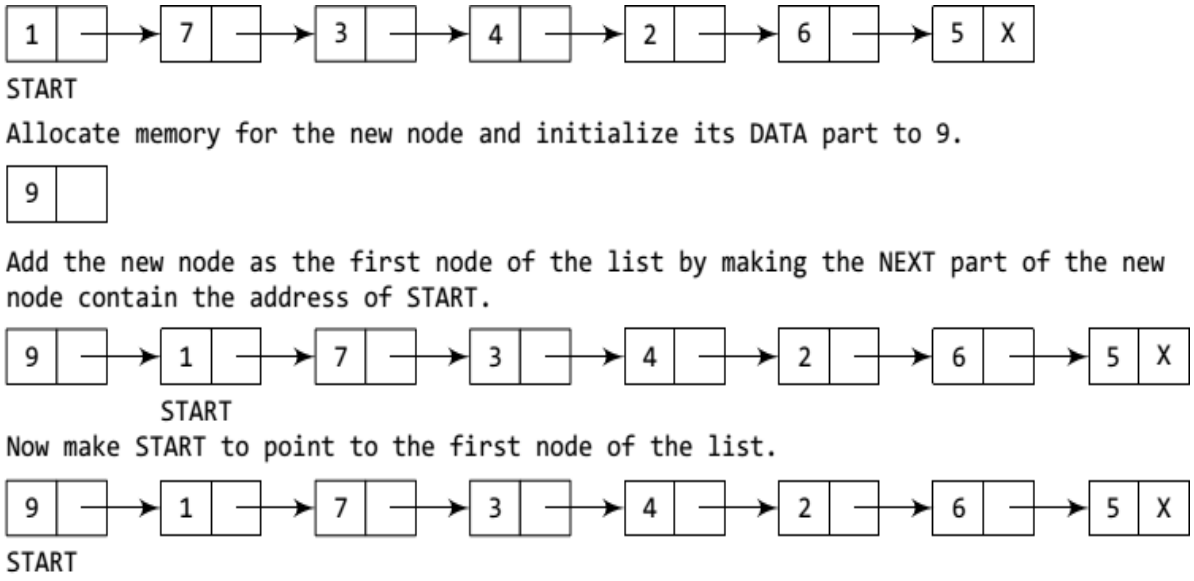


Figure 6.12 Inserting an element at the beginning of a linked list

Algorithm to insert a new node at the beginning of linked list (Algorithm case 1)

Step 1: if AVAIL=NULL, then print "overflow", go to step 7
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

2. Insert at end

For example: Insert a new node with data 9 at the end

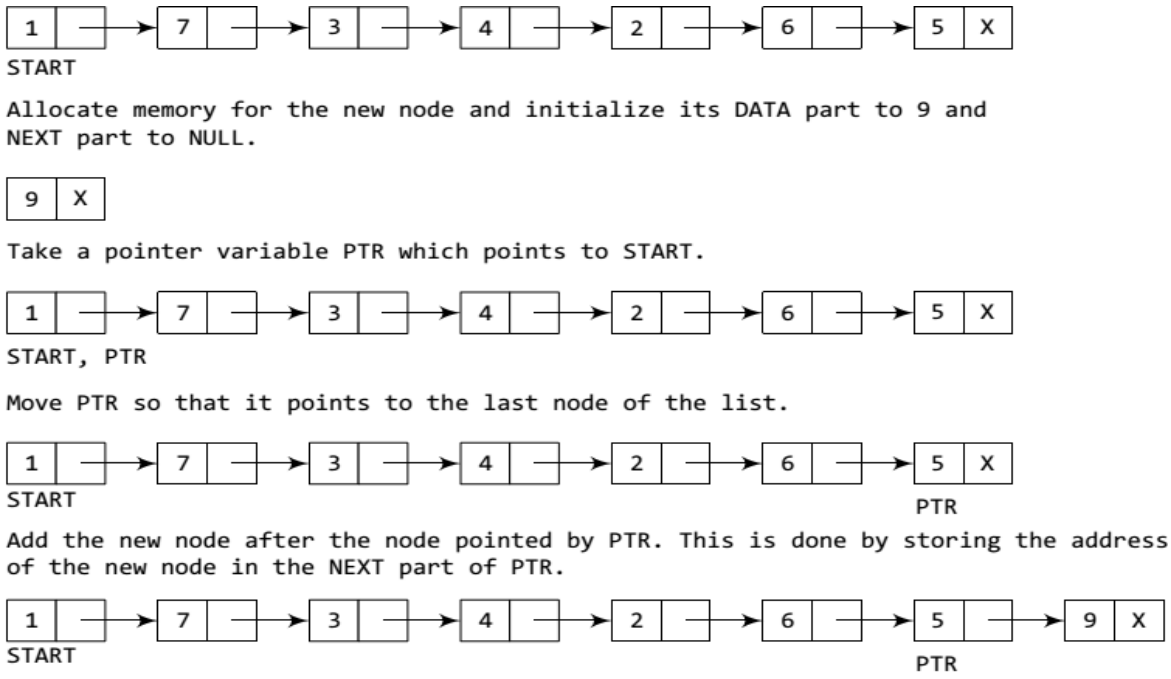


Figure 6.14 Inserting an element at the end of a linked list

Algorithm to insert a new node at the end of linked list (Algorithm case 2)

Step 1: if AVAIL=NULL, then write overflow go to step 10
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
Step 9: SET PTR->NEXT=New_Node
Step 10: Exit

3. Insert after a given node

- ✓ PREPTR is a pointer variable which stores the address of node preceding PTR. i.e. previous of PTR.
- ✓ In the *while* loop, traverse through the linked list to reach the node that has value equal to NUM.

For example: Insert a new node with value 9 after the node containing data 3.
VAL = 9, NUM = 3

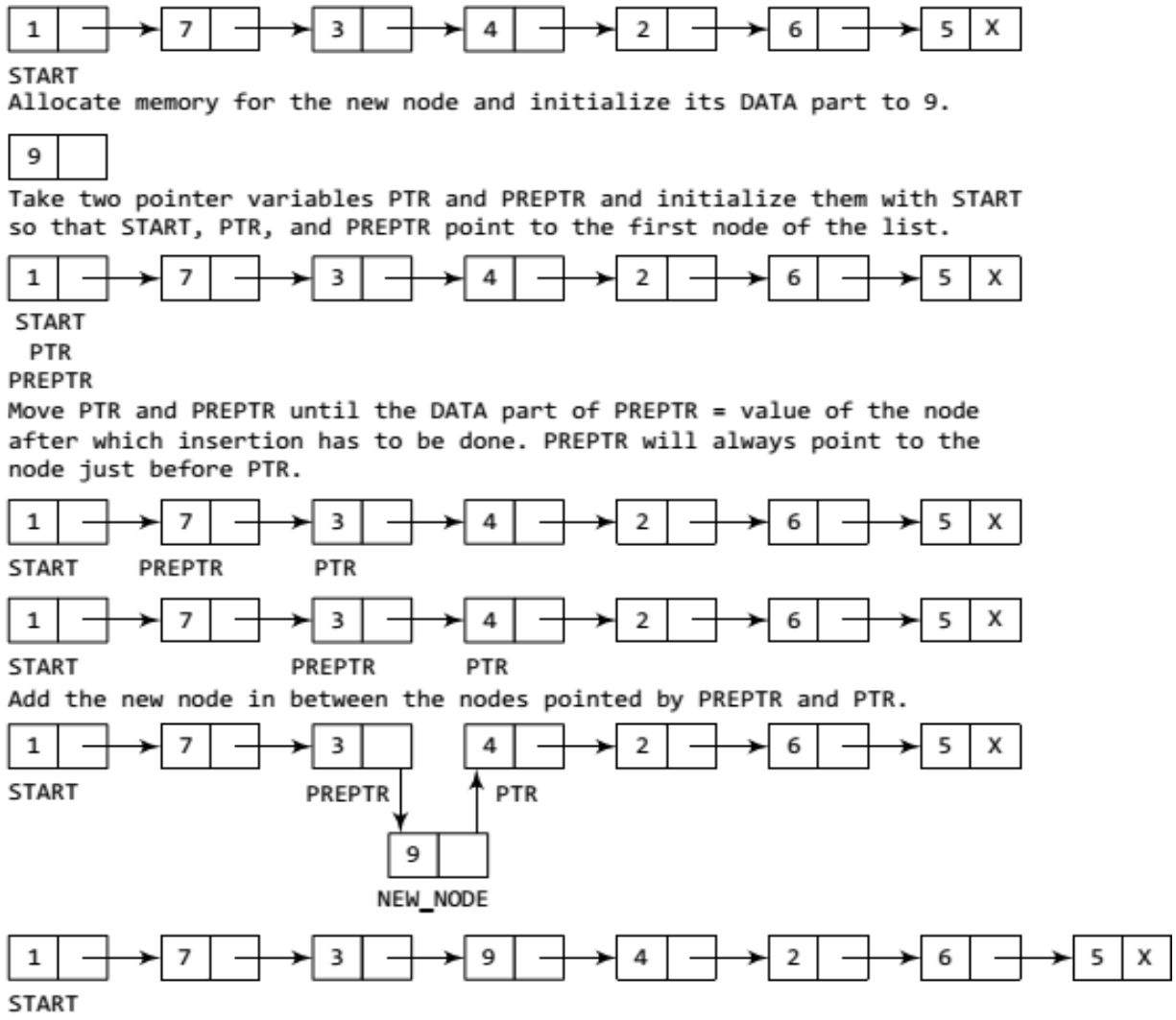


Figure 6.17 Inserting an element after a given node in a linked list

Algorithm to insert a new node after a node that has value NUM (Algorithm case 3)

Step 1: if AVAIL=NULL, then print "overflow" go to step 12
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 step 8 and 9 while **PREPTR->DATA!=NUM**
Step 8: SET PREPTR = PTR
Step 9: SET PTR=PTR->NEXT
Step 10: SET PREPTR->NEXT=New_Node
Step 11: SET New_Node->NEXT=PTR
Step 12: Exit

4. Insert before a given node

✓ PREPTR is a pointer variable which stores the address of node preceding PTR.
i.e. previous of PTR.
✓ In the *while* loop, traverse through the linked list to reach the node that has value equal to NUM.
For eg: Insert a new node with value 9 before the node containing data 3. VAL = 9, NUM = 3

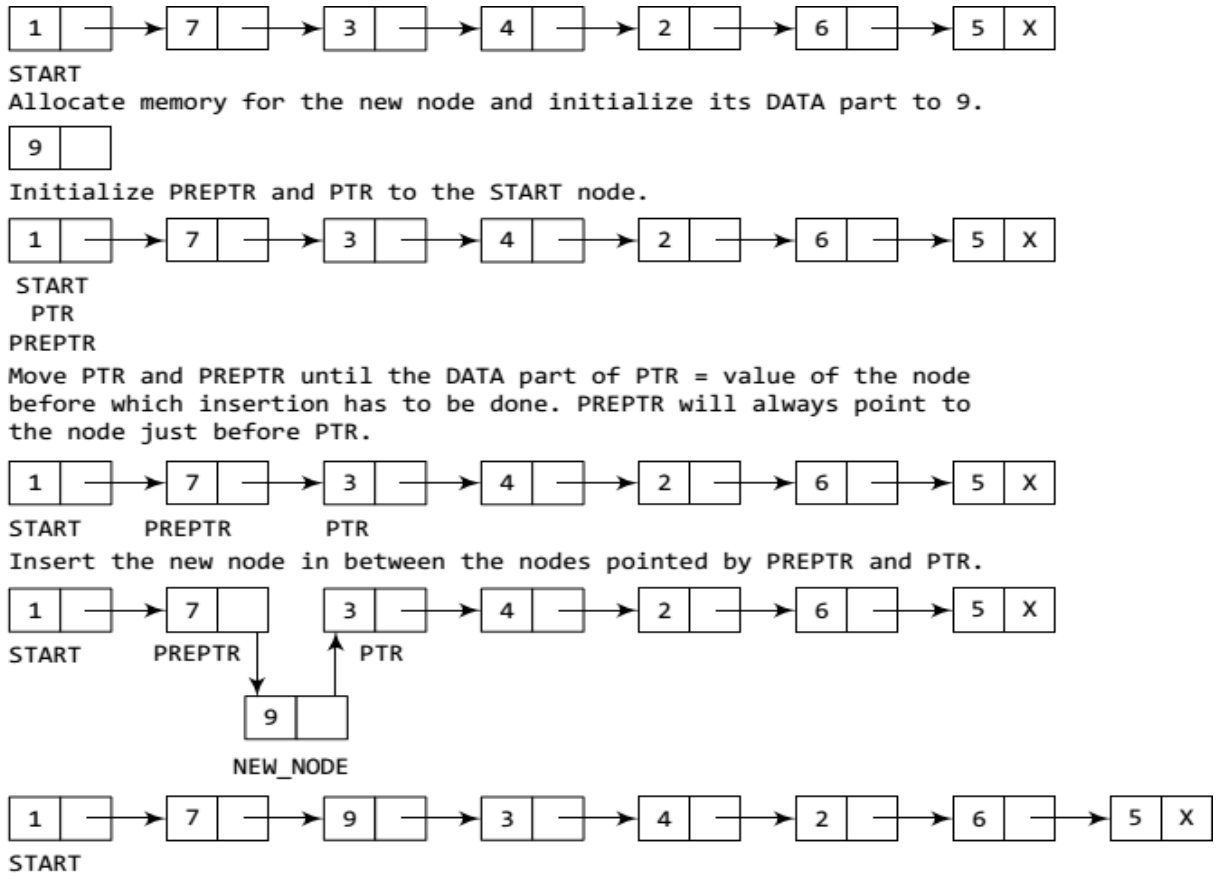


Figure 6.19 Inserting an element before a given node in a linked list

Algorithm to insert a new node before a node that has value NUM (Algorithm case 4)

Step 1: if AVAIL=NULL, then print "overflow" go to step 12
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 step 8 and 9 while **PTR->DATA!=NUM**
Step 8: SET PREPTR = PTR
Step 9: SET PTR=PTR->NEXT
Step 10: SET PREPTR->NEXT=New_Node
Step 11: SET New_Node->NEXT=PTR
Step 12: exit

5. Insert in Sorted Linked List

In Figure 8.22, insert a new node with value 5. VAL=5:-

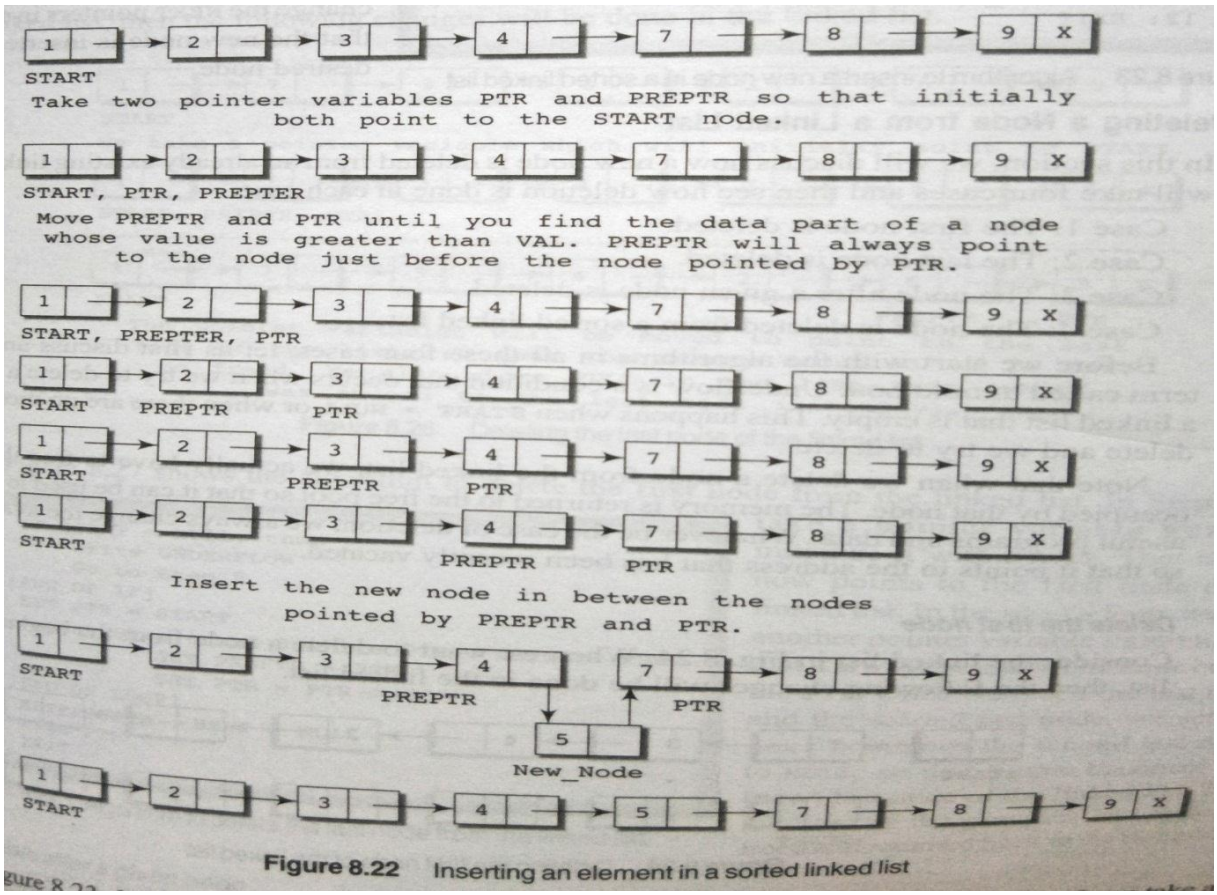


Figure 8.22 Inserting an element in a sorted linked list

Algorithm to insert a new node in Sorted Linked List (Algorithm case 5)

Step 1: if AVAIL=NULL, then print "overflow" go to step 12
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 step 8 and 9 while **PTR->DATA < VAL**
Step 8: SET PREPTR = PTR
Step 9: SET PTR=PTR->NEXT
Step 10: SET PREPTR->NEXT=New_Node
Step 11: SET New_Node->NEXT=PTR
Step 12: Exit

IV. Deleting a node

✓ For deleting a node, we will take four cases
Case 1: Delete the first node
Case 2: Delete the last node

Case 3: Delete after a given node
Case 4: Deleted the node from a sorted linked list

- ✓ UNDERFLOW is a condition that occurs when we try to delete a node from an empty linked list. This happens when START = NULL. i.e. no node to delete.
- ✓ After deletion, free the deleted space and return to free pool.

Unsorted Linked List

1. Delete the first node

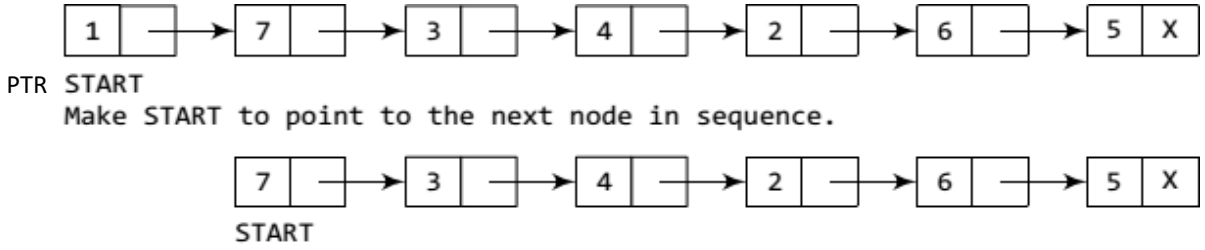


Figure 6.20 Deleting the first node of a linked list

START is made to point to the next node and finally the memory occupied by the node pointed by PTR (the first node of the list) is freed and returned to the free pool.

Algorithm to delete the first node of linked list (Algorithm case 1)

Step 1: If START=NULL, then print "underflow" go to step 5
Step 2: SET PTR=START
Step 3: SET START=START->NEXT
Step 4: **FREE PTR**
Step 5: EXIT

2. Delete the last node

- ✓ PREPTR is a pointer variable which stores the address of node before PTR. i.e. previous of PTR.
- ✓ SET PREPTR->NEXT=NULL.

So that PREPTR now becomes the (new) last node of linked list. The memory of previous last node is freed and returned to the free pool.

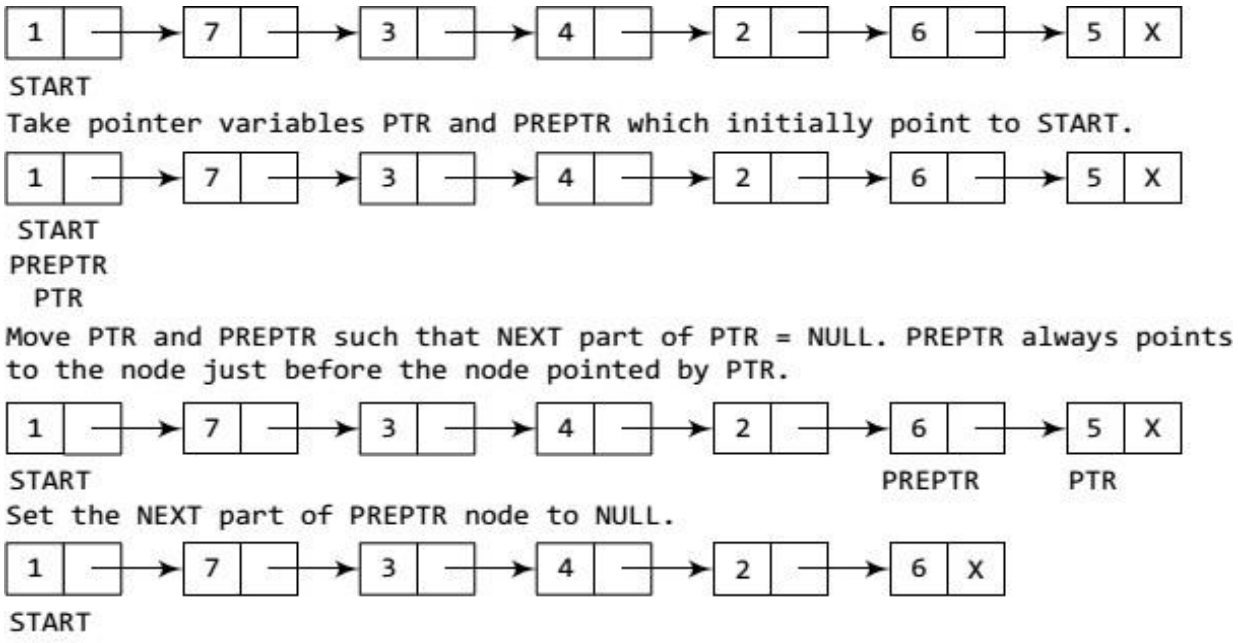


Figure 6.22 Deleting the last node of a linked list

Algorithm to delete the last node of linked list (Algorithm case 2)

Step 1: If START=NULL, then print "underflow" go to step 8
Step 2: SET PTR=START
Step 3: SET PREPTR=PTR
Step 4: Repeat step 5 and 6 while PTR->NEXT! =NULL
Step 5: SET PREPTR=PTR
Step 6: SET PTR=PTR->NEXT
Step 7: SET PREPTR->NEXT=NULL
Step 8: FREE PTR
Step 9: EXIT

3. Delete after a given node

- ✓ PREPTR is a pointer variable which stores the address of node preceding PTR. i.e. previous of PTR.
- ✓ The memory of the node after (succeeding) given node is freed and returned to the free pool.

For example: Delete the node after 4.

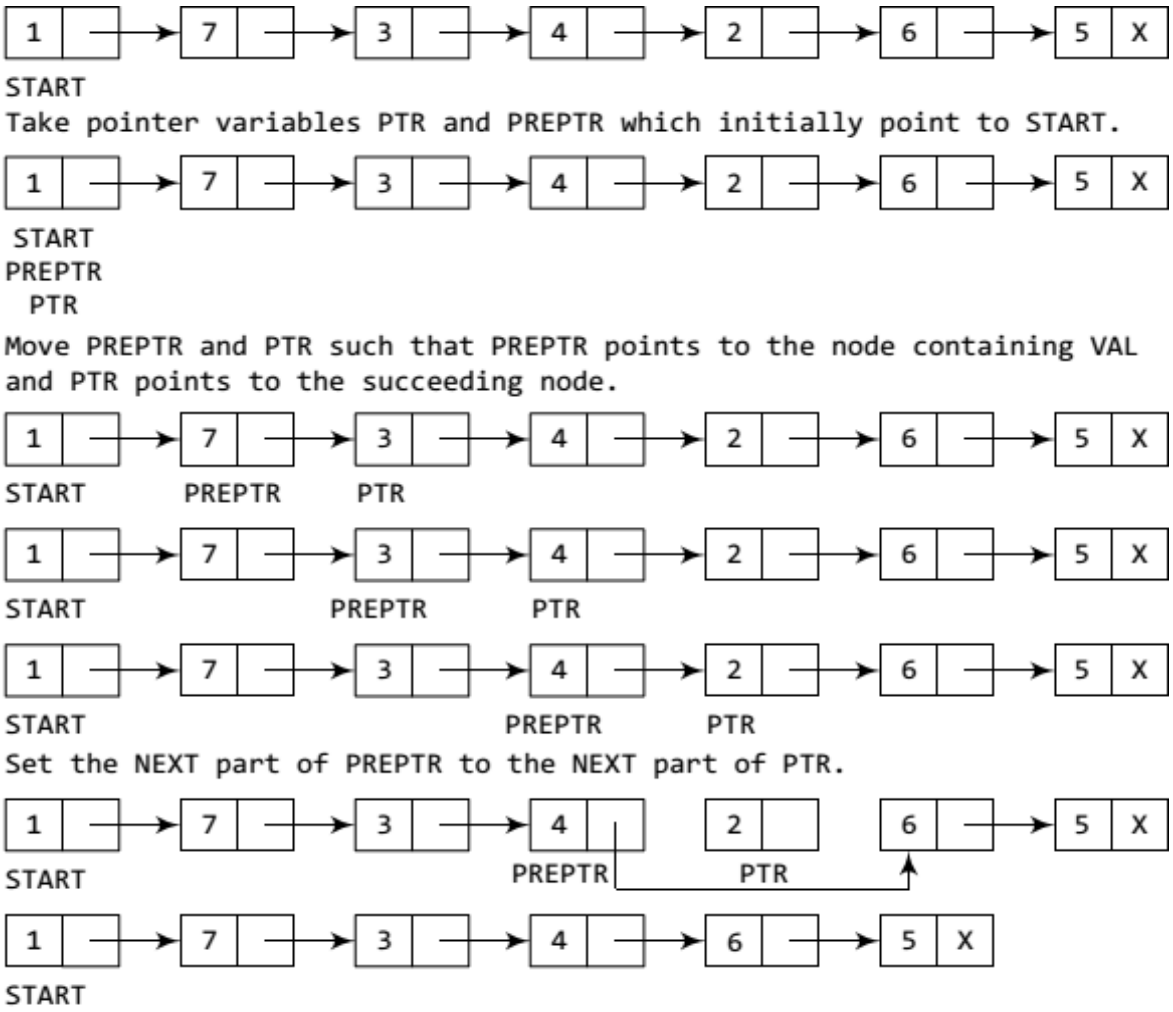


Figure 6.24 Deleting the node after a given node in a linked list

Algorithm to delete after a given node of linked list (Algorithm case 3)

Step 1: If START=NULL, then print "underflow" go to step 10
Step 2: SET PTR=START
Step 3: SET PREPTR=PTR
Step 4: Repeat step 5 and 6 while **PREPTR->DATA! =NUM**
Step 5: SET PREPTR=PTR
Step 6: SET PTR=PTR->NEXT
Step 7: SET TEMP=PTR
Step 8: SET PREPTR->NEXT=PTR->NEXT
Step 9: **FREE TEMP**
Step 10: EXIT

4. Delete from a sorted linked list

- PREPTR is a pointer variable which stores the address of node preceding PTR. i.e. previous of PTR.
- The memory of the node pointed by PTR is freed and returned to the free pool.

For example: Delete a node with value 4 in sorted linked list.

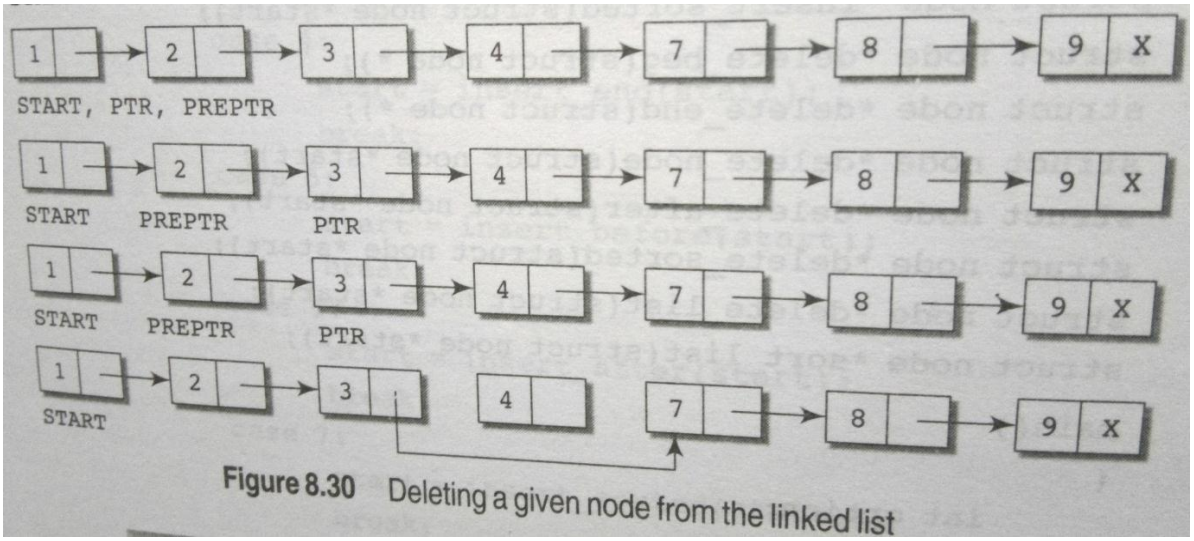


Figure 8.30 Deleting a given node from the linked list

Algorithm to delete the node from a sorted linked list (Algorithm case 4)

Step 1: If START=NULL, then print "underflow" go to step 9
Step 2: SET PTR=START
Step 3: SET PREPTR=PTR
Step 4: Repeat step 5 and 6 while **PTR->DATA!=NUM**
Step 5: SET PREPTR=PTR
Step 6: SET PTR=PTR->NEXT
Step 7:SET TEMP=PTR
Step 8: SET PREPTR->NEXT=PTR->NEXT
Step 9: **FREE TEMP**
Step 10: EXIT

LinkedList ADT

AbstractDataType *LinkedList*
{
Instances
Linear list of elements and node;
A pointer to the next node in the list called NEXT;
A pointer to the first node in the list called START, HEAD;
A pointer to the address of the first free space is called AVAIL.
}
{
Operations

- IsEmpty : Return true if linked list is empty (underflow), return false otherwise.
  If the list is initially empty, START is set to NULL.
  boolean data type
- Find(x) : Search for a node with the value equal to x in the list.
  If such a node is found, return its **position**.
  Otherwise (If x not in linked list), return 0.

- FindKth() : Search for a node with the position equal to K in the list.
  If such a node is found, return its **value**.
  If K=1, return First node value
- PrintList(), DisplayList() : print all the nodes in the list.
  Print the number of the nodes in the list
- InsertNode : Insert a new node at a particular position.
  Insert does not require moving the other elements
- InsertAfter() : Insert after a given node
- InsertBefore() : Insert before a given node
- DeleteNode( ) : Delete a node with the value equal to x from the list.
  Delete does not require moving the other elements
- DeleteAfter() : Delete after a given node
- MakeEmpty(): Delete all nodes in linked list.
  After this operation, linked list will be empty
- Successor(x) : Return the next node of x
  The last node has no successor
- Predecessor(x) : Return the previous node of x
  The first node has no predecessor

}

Linked Representation of Stack (Implementing stack with Linked list ADT)

- In a linked stack, every node has two parts-one that stores data and another that stores the address of the next node.
- The START pointer of the linked list is used as TOP.

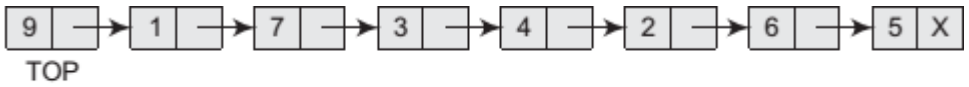


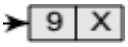
Figure 7.13 Linked stack

- All insertions and deletions are done at the node pointed by the TOP.
- If TOP=NULL, then it indicates that the stack is empty.

Operations on a linked stack:-

1. Push operation

✓ The push operation is used to insert an element. The new element is added at the topmost position.
✓ Allocate memory for a new node; store the value in its data part.
✓ To insert an element, first check TOP=NULL. If so, allocate memory for a new node, store the value in its data part and NULL in its next part



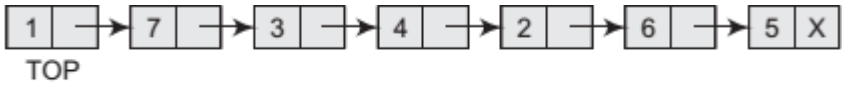


Figure 7.14 Linked stack

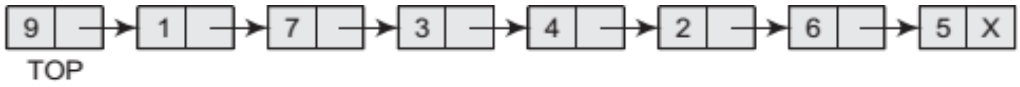


Figure 7.15 Linked stack after inserting a new node

Algorithm

Step 1: Allocate memory for the new node and name it as New_Node
Step 2: SET New_Node->DATA=VAL
Step 3: If TOP=NULL, then
SET New_Node->NEXT=NULL
Set TOP= New_Node
Else
SET New_Node->NEXT =TOP
SET TOP= New_Node
Step 4: end

2. Pop operation

- ✓ POP is used to delete the topmost element from the stack.
- ✓ If top=NULL, the stack is empty and deletion not possible. i.e. UNDERFLOW
- ✓ If top!=NULL, then we will **delete the first node pointed by the top**

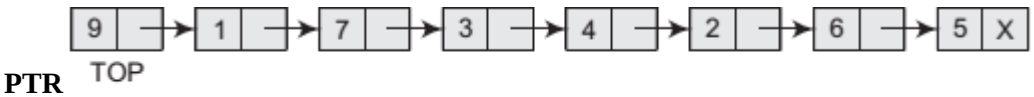


Figure 7.17 Linked stack

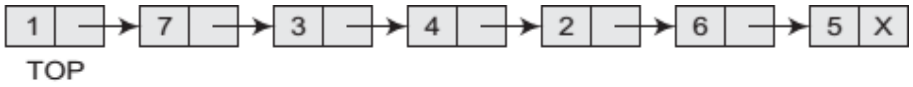


Figure 7.18 Linked stack after deletion of the topmost element

Algorithm

Step 1: If TOP=NULL, then print "UNDERFLOW" go to step 5
Step 2: SET PTR=TOP
Step 3: SET **TOP=TOP->NEXT**
Step 4: **FREE PTR**
Step 5: Exit

LinkedStack ADT

AbstractDataType *LinkedStack*

{

Instances

Linear list of elements;
Insertion, deletion take place at one end called *top*;

}

{

Operations

IsEmpty : Return true if linked stack is empty (underflow), return false otherwise.
If the linked stack is initially empty, START is set to NULL.
boolean data type

push(x) : insert element x at the top
check OVERFLOW condition
pop() : remove the top element i.e. first element
check IsEmpty()
top(), peep() : return the top element

}

Linked representation of a queue (implementing queue with Linked list ADT)

- In a linked queue, every node has two parts-one that stores data and another that stores the address of the next node.
- The START pointer of the linked list is used as FRONT.
- REAR store the address of last element in queue

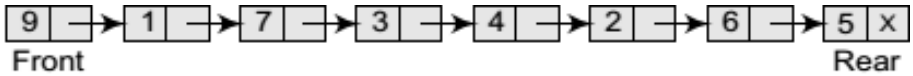


Figure 8.6 Linked queue

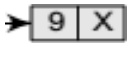
- **All insertions will be done at REAR end and all deletions will be done at FRONT end.**
- **If FRONT=REAR=NULL, then the queue is empty.**

Operations on a linked queue:-

1. **Insert operation**

- The new element is added as the last element of the queue.
- Allocate memory for a new node

If FRONT=NULL, the queue is empty. Store the value in its data part and NULL in its next part



PTR

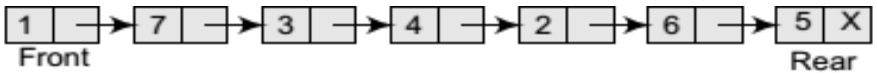


Figure 8.7 Linked queue

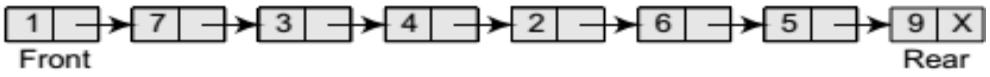


Figure 8.8 Linked queue after inserting a new node

Algorithm

Step 1: Allocate memory for the new node and name it as NEWNODE
Step 2: SET NEWNODE ->DATA=VAL
Step 3: if FRONT=NULL, then
Set FRONT=REAR= NEWNODE
Set FRONT->NEXT=REAR->NEXT=NULL
Else
Set REAR->NEXT= NEWNODE
Set REAR= NEWNODE
Set REAR->NEXT=NULL
Step 4: end

2. Delete operation

- The START pointer of the linked list is used as FRONT.
- Deletes the element whose address is stored in FRONT
- To delete an element, first check if FRONT=NULL, it indicates empty- UNDERFLOW. Otherwise delete the first node pointed by FRONT. Then FRONT is made to point to next node.

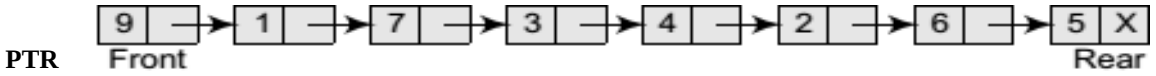


Figure 8.10 Linked queue

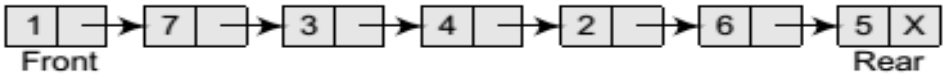


Figure 8.11 Linked queue after deletion of an element

Algorithm

Step 1: if FRONT=NULL, then print “underflow” go to step 5
Step 2: SET PTR=FRONT
Step 3: FRONT=FRONT->NEXT
Step 4: **FREE PTR**
Step 5: Exit

LinkedQueue ADT

AbstractDataType *LinkedQueue*

{

Instances

Linear list of elements;
Insertion take place at one end called *rear*;
Deletion take place at other end called front;

}

{

Operations

IsEmpty : Return true if linked queue is empty (underflow), return false otherwise.
boolean data type

insert(x) : insert element x at the rear;
check OVERFLOW condition

delete() : remove the front element ;
i.e. first check IsEmpty()

front(), peek() : return the front element

}

2. Circular Linked List

- ✓ **The last node contains a pointer to the first node of the list.**
- ✓ we can traverse the list in any direction, forward or backward
- ✓ A circular linked list has no beginning and no ending.
- ✓ There are Circular singly linked list and Circular doubly linked list.
- ✓ Circular list is shown below:-

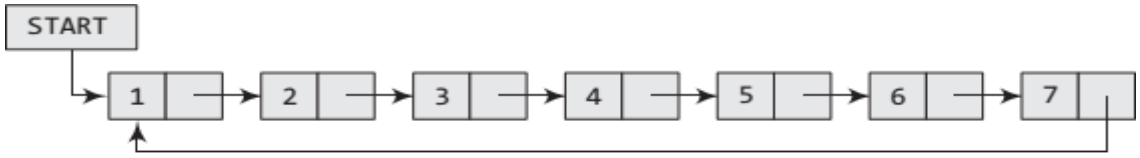


Figure 6.26 Circular linked list

- ✓ Application

1) widely used in the operating systems for task maintenance
2) Used to maintain the sequence of the web pages visited:
   Traversing this circular linked list either in forward or backward direction helps to revisit the pages again using back and forward buttons.

- ✓ Disadvantage -complexity of iteration

Memory representation of a circular linked list

- To form a Circular linked list, we need a structure called the node that has two fields- DATA

and NEXT. DATA will store the information and NEXT will store the address of next node.

- NEXT field does not contain NULL.
    - The figure shows memory representation of a circular linked list

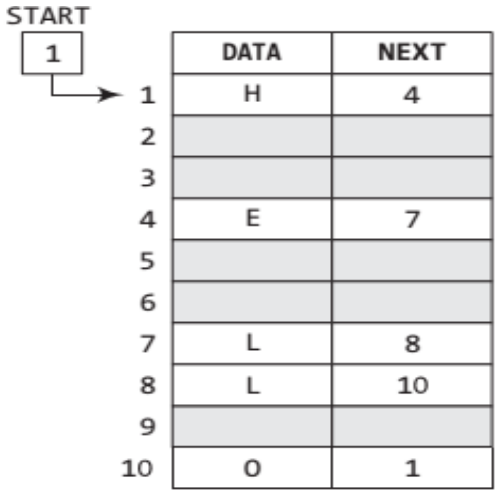


Figure 6.27 Memory representation of a circular linked list

From the figure, the START pointer will point the first node in the list. Here the data is H and NEXT pointing location is 4. Finally, P element point NEXT field to 1.

3. Doubly Linked List

- ✓ Doubly Linked List is also called **two-way linked list**
- ✓ It is more complex type of linked list in which contains a pointer to the next as well as the previous node in the sequence.
- ✓ It consists of three parts:-
    1) A pointer from the previous node,
    2) Data,
    3) A pointer to the next node

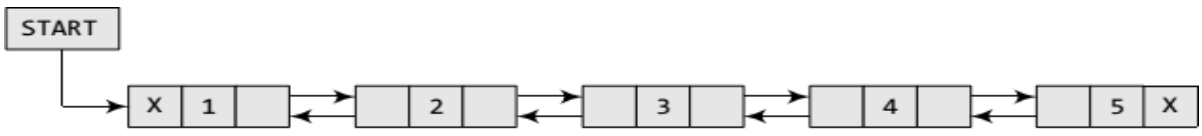


Figure 6.37 Doubly linked list

- The PREV field is used to store the address of the previous (preceding) node. This helps to traverse backward direction.
- The PREV field of the first node and the NEXT field of the last node will contain NULL or X or -1.
    - Advantage
        1) It makes searching twice as efficient
        2) Easy to manipulate the elements of the list, because it maintains pointers to nodes in both direction
    - Disadvantage

    Requires more space per node and more expensive basic operations

Memory representation of Doubly linked list

The figure shows memory representation of a circular linked list

START = 1

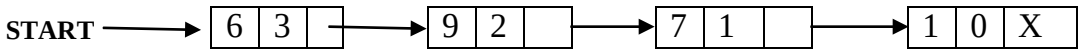
	DATA	PREV	NEXT
1	H	-1	3
2			
3	E	1	6
4			
5			
6	L	3	7
7	L	6	9
8			
9	O	7	-1

Figure 6.38 Memory representation of a doubly linked list

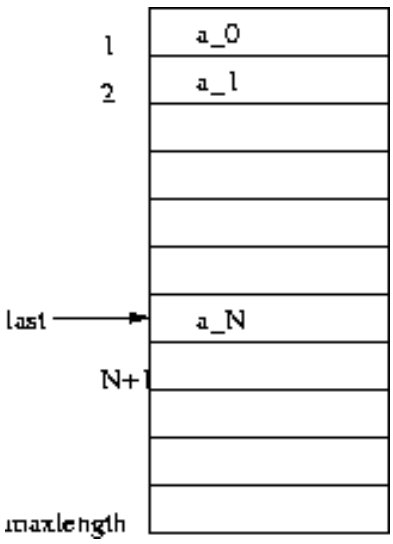
Here START=1, so first data is stored at address 1, which is H. This is the first node and its previous node PREV is -1. We will traverse the list until we reach a position where NEXT contains -1 or NULL or X.

Linked list Polynomial representation

- **Every individual term in a polynomial consist of two parts, a coefficient and a power**
- Every term of a polynomial can be represented as a node of linked list
- Linked representation of a polynomial 6x3+9x2+7x+1 is shown below



$$f(x) = \sum_{i=0}^{N} a_i x^i$$



Algorithm to Implement Polynomial Addition

Let p and q be the two polynomials represented by the linked list.

Step1: while p and q are not null, repeat step 2.

Step 2: If powers of the two terms are equal, then if the terms do not cancel then insert the sum of the terms into the sum Polynomial Advance p Advance q

Else if the power of the first polynomial> power of second ,

Then

Insert the term from first polynomial into sum polynomial Advance p

Else

Insert the term from second polynomial into sum polynomial Advance q

Step 3: Copy the remaining terms from the non-empty polynomial into the sum polynomial