DOUBLY LINKED LIST

INSERTION

case1:- New node ~~instan~~ inserted at the beginning.

INITIALIZE

Step1:- IF AVAIL = NULL
Write Overflow
Goto to Step8,9.
[END OF IF]

AVAIL:- to check memory
NEWNODE:- node to be inserted
VAL:- value to be inserted
PREV:- to point the previous node add location

Step2:- SET NEWNODE = AVAIL
Step3:- SET AVAIL = AVAIL → NEXT.
Step4:- SET NEWNODE → DATA = VAL
Step5:- SET NEWNODE → PREV = NULL
Step6:- SET NEWNODE → NEXT = START.
Step7:- SET START → PREV = NEWNODE
Step8:- SET START = NEWNODE.
Step9:- EXIT.

Start

Allocate memory to newnode & initialize its data part to 9 and prev field to NULL.

Add new node before the start node. Now the new node becomes first node of list.

Case: Inserting at the End of Doubly Linked List

INITIALIZE

STEP 1:- IF AVAIL = NULL
Write Overflow
Goto step 11
[END OF IF]

Step 2:- SET NEW_NODE = AVAIL

Step 3:- SET AVAIL = AVAIL → NEXT.

Step 4:- SET NEWNODE → DATA = VAL

Step 5:- SET NEWNODE → NEXT = NULL.

Step 6:- SET PTR = START.

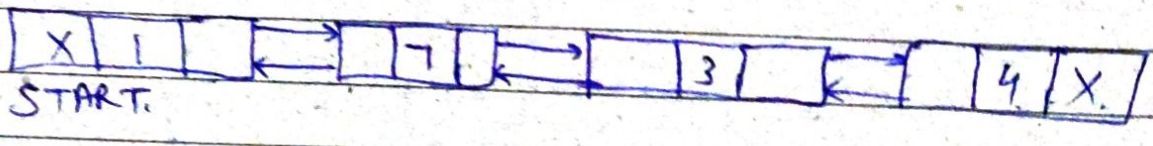
Step 7:- Repeat Step 8 while PTR → NEXT != NULL

Step 8:- SET PTR = PTR → NEXT.
[END OF LOOP]

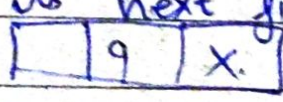
Step 9:- SET PTR → NEXT = NEW NODE

Step 10:- SET NEWNODE → PREV = PTR

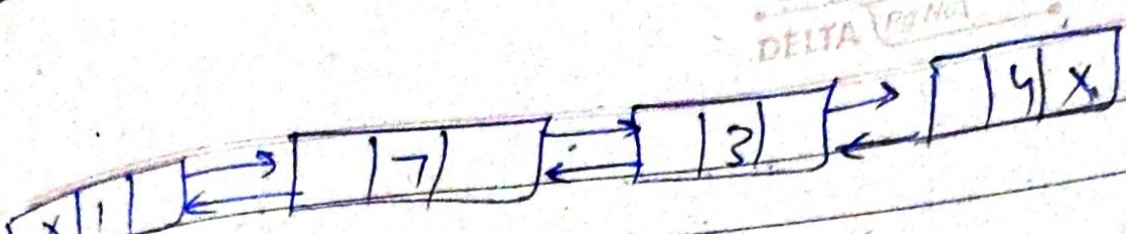
Step 11:- EXIT.



Allocate memory for new node & initialize its data part to 9 and its next field to NULL.

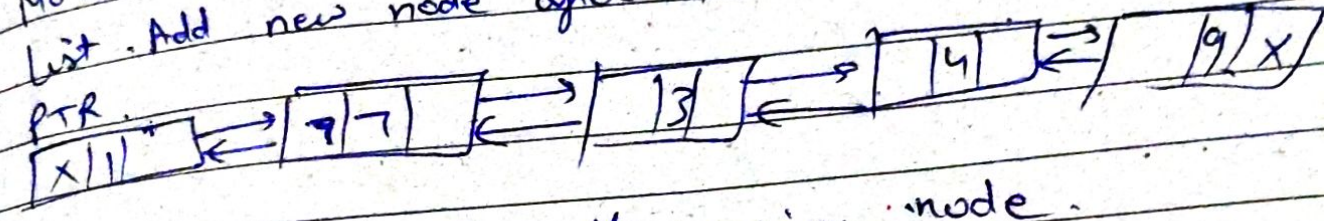


Take a pointer variable PTR & make it point to first node of list.



Start PTR.

Move PTR so that it points to last node of list. Add new node after the node pointed by PTR.



Case 3 Inserting node after a given node.

Initialize.

Step1 IF AVAIL = NULL
Write Overflow
Goto Step 12.
[END OF IF]

Step2. SET NEW NODE = AVAIL

Step3 SET AVAIL = AVAIL → NEXT.

Step4 SET NEWNODE → ~~NEXT~~ DATA = VAL

Step5 SET PTR = START.

Step6 Repeat Step 7 while PTR → DATA != NUM.

Step7 SET PTR = PTR → NEXT
[END OF LOOP]

Step8 SET NEW NODE → NEXT = PTR → NEXT.

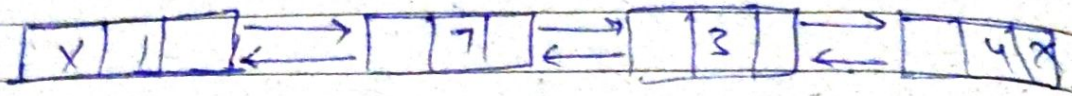
Step9 SET NEW NODE → PREV = PTR

Step10 SET PTR → NEXT → PREV → NEWNODE

Step11 SET PTR → NEXT → PREV → NEWNODE

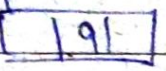
Step12 EXIT

ROUGH

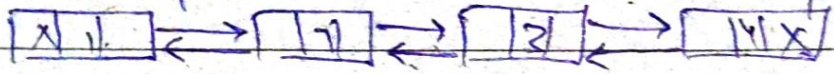


START

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

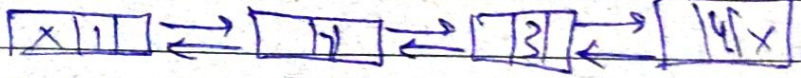


Take a pointer variable PTR and make it point to first node



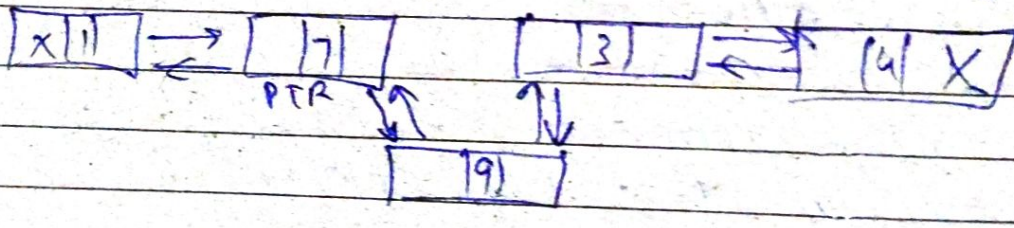
START, PTR.

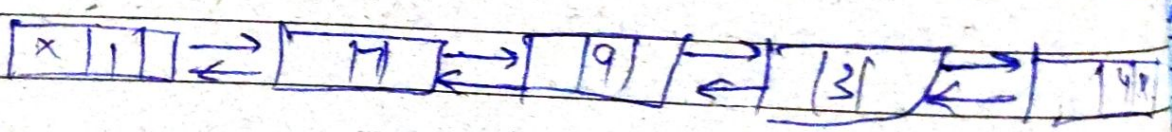
Move PTR further until data part of PTR = value after which the node has to be inserted.



START PTR.

Insert the new node b/w PTR & node succeeding it





ROUGH

Deleting Node

Initialize

Step 1 If START = N

WRITE Underf

Goto step

Step [END OF

Step 2 SET PTR

Step 3 Repeat ste

Step 4 PTR =

[END O

Step 5 SET TE

Step 6 SET PTR

Step 7 SET TE

Step 8 FREE T

Step 9 EXIT

Store

Take a pointer

PTR

Move PTR further

after which the

ROUGH

Deleting Node after a given node

Initialize.

Step 1 If START = NULL
WRITE Underflow
Goto step 9
[END OF IF]

Step 2 SET PTR = START.

Step 3 Repeat Step 4 while PTR → DATA != NUM

Step 4 PTR = PTR → NEXT
[END OF LOOP]

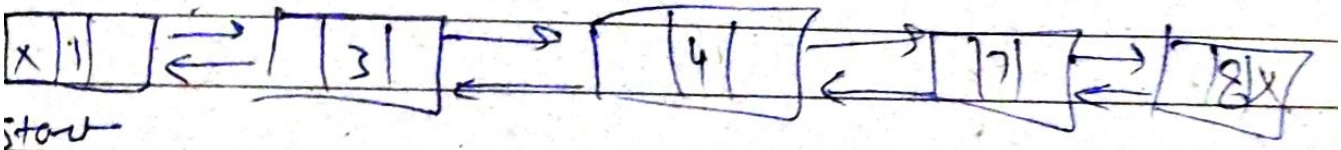
Step 5 SET TEMP = PTR → NEXT

Step 6 SET PTR → NEXT = TEMP → NEXT

Step 7 SET TEMP → NEXT → PREV = PTR

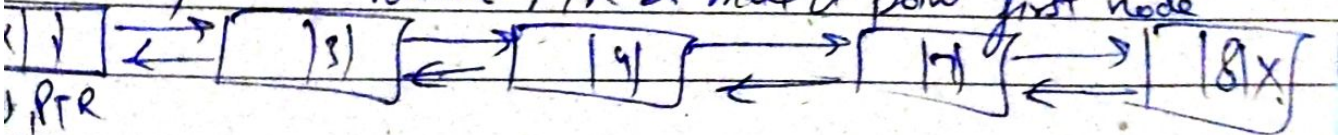
Step 8 FREE TEMP

Step 9 EXIT

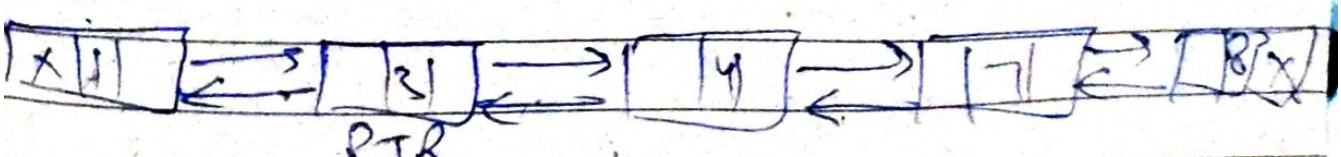


start

Take a pointer variable PTR & make it point first node

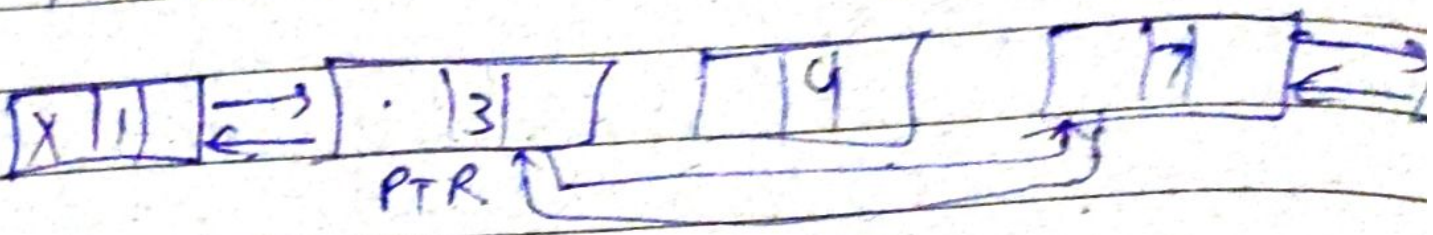


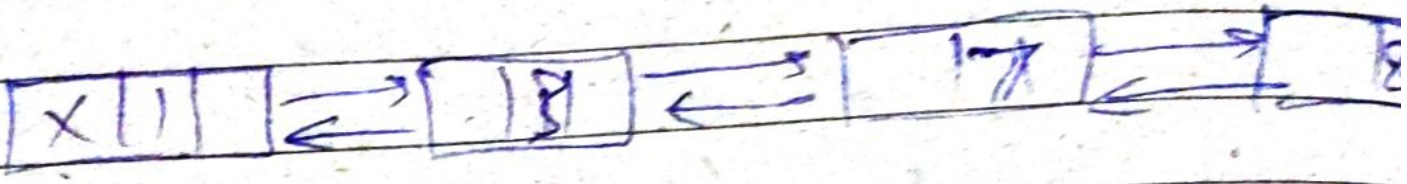
Move PTR further so that its data part is equal to value upto which the node has to be deleted



ROUGH

Delete the node succeeding PTR.

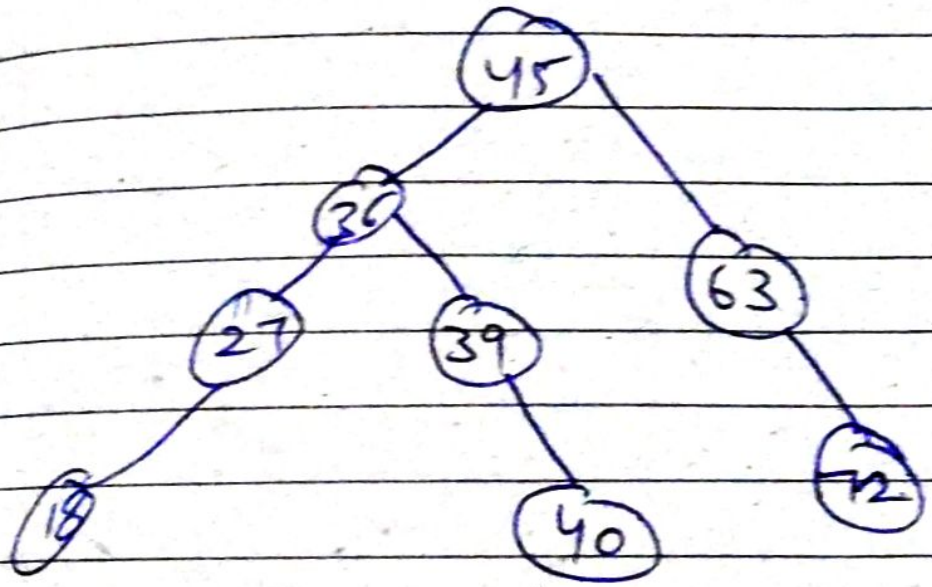




AVL TREES

AVL tree is a self balancing binary search tree: The height of two sub trees of a node may differ by only 1
for all operation it takes O(log n)

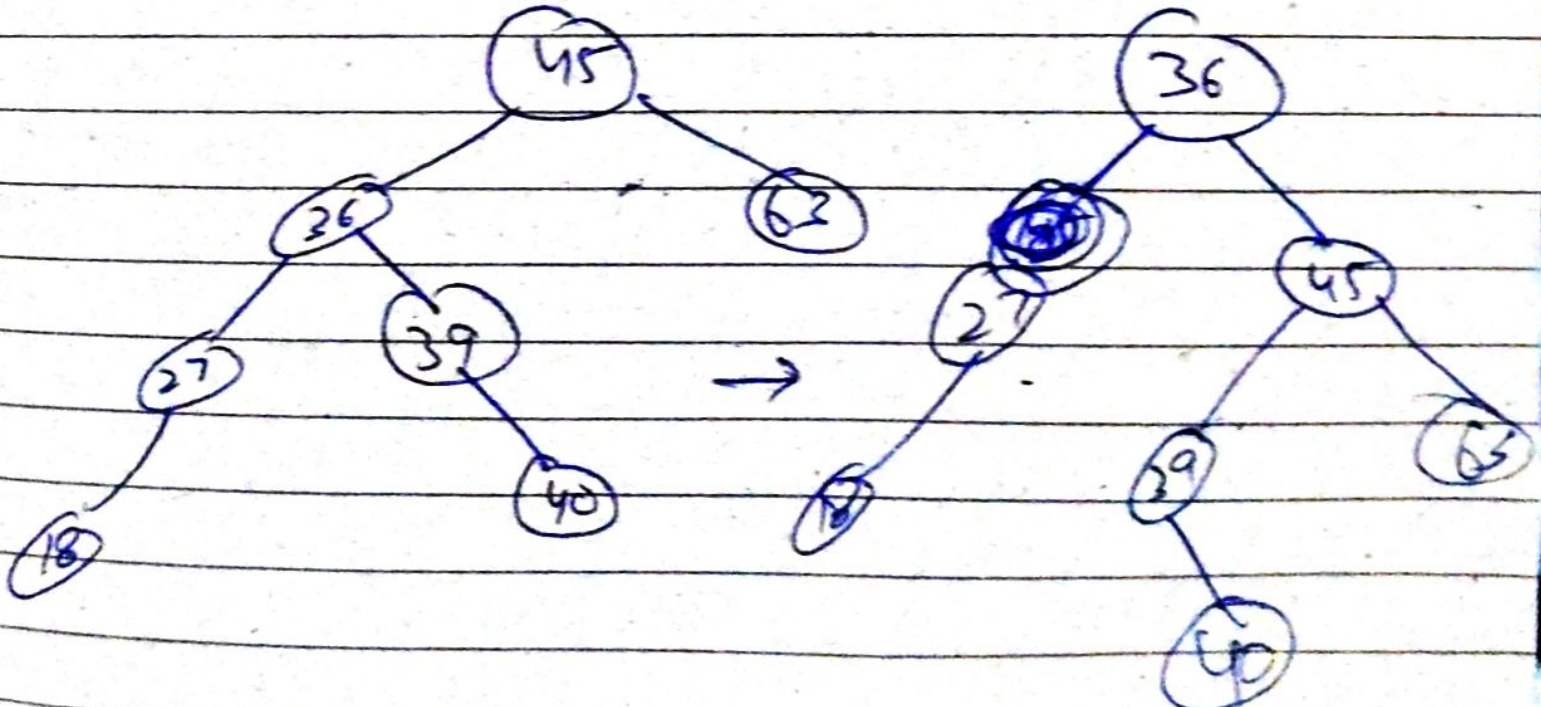
Balance factor = height of left subtree − height of right subtree



RO
Rotation

Deletion

If balance factor of B is 0.



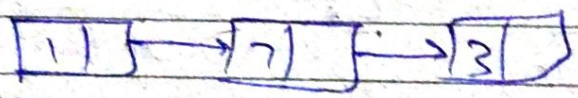
Algorithm to insert an element in a Stack

Step1 Allocate memory for new node & name it as NEW_NODE

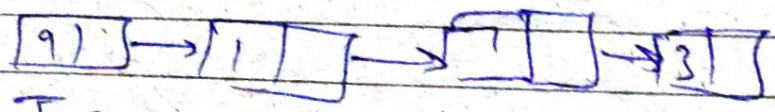
Step2 NEW_NODE → DATA = VAL

Step3 IF TOP = NULL
SET NEWNODE → NEXT = NULL.
SET TOP = NEWNODE.
ELSE
SET NEWNODE → NEXT := TOP
SET TOP = NEWNODE.
[END OF IF]

Step4 END



Top



Top.

To delete an

Step1 IF TOP

Step2 SET PTR

Step3 SET TOP

Step4 FREE P

Step5 EXIT

Program to

#include <conio

#include <std

int stk[10];

int top = -1;

int pop();

void push(int)

int main()

{ int val, n

int arr[10

printf("\n Ente

scanf("%d",

printf("Enter

for(i=0, i

{ scanf(

To delete an element in stack.

Step1 IF TOP = NULL
Print Write Underflow
Goto Step 5.
Step2 SET PTR = TOP
Step3 SET TOP = TOP → NEXT
Step 4 FREE PTR.
Step5 EXIT

Program to reverse a string using stack.

```
#include <conio.h>
#include <stdio.h>
int stk[10];          //Stack.
int top = -1;         // top of stack.
int pop();            // func. prototype for Pop.
void push(int);       // func. prototype for Push op.
int main()
{ int val, n, i;
  int arr[10];
printf("\n Enter no. of elements in array");
scanf("%d", &n);
printf("Enter element of array:")
for(i=0; i<n; i++)
{ scanf("%d", &arr[i]);
```

```
for(i=0; i<n; i++)
{ push(arr[i]); }  // push func. called

for(i=0; i<n; i++)
{
    val = pop();          // Pop func. called
    arr[i] = val; }
printf("\n the reversed array is:");

for (i=0; i<n; i++)
  printf("\n%d", arr[i]);
  return 0;
}
void push (int val)
{
  stk[++top] = val;
}
int pop ()
{ return (stk[top--]);
}
```

Algorithm to convert infix to postfix.

tep1: Add ")" to the end of infix expression
tep2: Push "(" to the stack.
tep3 Repeat until each character in infix expression is scanned.
If a "(" is encountered, push it on stack.
If an operand is encounter, add it to postfix expression.
If ")" is encountered, then
a) Repeatedly pop from stack & add to postfix expression till a "(" is encountered.
b) Discard "(", remove "(" from stack.

f an operator O is encountered then.
a) Repeatedly pop from stack & add each operator to the expression which has same precedence or higher precedence than O.
b) Push the operator O to stack.
[END of IF]
ep4:- Repeatedly pop from the stack & add to postfix expression until the stack is empty.
ep5 Exit.

Date / /
Page No.

Unit - IV

Sorting and Hashing

Bubble Sort

- Sorts the array element by repeatedly moving the largest element to the highest index position of the array. (in case of arranging in asc. order)
- Consecutive adjacent pairs of elements in the array are compared with each other.
- If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so the elements will be interchanged.

Note:- If element are to be sorted in descending order, then in first pass the ~~same~~ smallest element is moved to the highest index of the array.

Technique:

(a) In Pass 1, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N-2] element is compared with A[N-1]. Pass 1 involves n-1 comparisions and places the biggest element at the highest index of the array.

(b) In Pass 2, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on, finally A[N-3] is compared with A[N-2]. Pass 2 involves n-2 comparisons.

Date / /
Page No.

(c) In Pass n-1, A[0] and A[1] are compared so that A[0] < A[1]. After this step, all the elements of the array are arranged in ascending order.

[Algorithm]

BUBBLE SORT. (A, N) A = Array to be sorted.
STEP1 INITIALISE N = Num of Elements in array
STEP2 Repeat for i = 0 to N-1
STEP3 Repeat for j = 0 to N-i
STEP4 If A[j] > A[j+1]
SWAP A[j] and A[j+1]
[END OF INNER LOOP]
[END OF OUTER LOOP]
STEP5 EXIT.

Complexity of Bubble Sort

The complexity of any sorting algorithm depends upon the number of comparisions

$f(n) = (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1$

$f(n) = n(n-1)/2.$

$f(n) = n^2/2 + O(n) = O(n^2).$

Therefore complexity of bubble sort is $O(n^2)$

i.e. time taken to sort is proportional to n^2 where 'n' is no. of elements in array.

Merge Sort

Merge Sort is a sorting algorithm that uses the divide, conquer and combine algorithmic paradigm.

Divide :- It means partitioning the n-element array to be sorted into two sub arrays of n/2 elements. If A is an array containing 0 or 1 element then it is already sorted. If there are more elements in the array, divide A into two sub-arrays, A_1 and A_2, each containing about half of the elements of A.

Conquer :- means sorting the two subarrays recursively using merge sort.

Combine :- means merging the two sorted arrays of size $n/2$ to produce the sorted array of 'n' elements.

Merge sort algorithm focuses on two main concepts to improve its performance (running time)

- A smaller list takes fewer steps and thus less time to sort than a large list.
- As number of steps is relatively less, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.

Date / /
Page No.

The basic steps of a merge sort algorithm are as follows:-

- If the array is of length 0 or 1, then it is already sorted.
- Otherwise, divide the unsorted array into two sub arrays of about half the size.
- Use merge sort algorithm recursively to sort each sub-array.
- Merge the two sub-arrays to form a single sorted list

```
MERGE SORT (ARR, BEG, MID, END)
ARR → ARRAY NAME
BEG → Starting Value of Array.
END → Last value of array.
MID → Index val Middle index Value of Array.
[INITIALIZE]
STEP 1  SET I = BEG, J = MID+1, INDEX = 0
STEP 2  Repeat while (I <= MID) AND (J <= END)
          IF ARR[I] < ARR[J]
            SET TEMP[INDEX] = ARR[I]
            SET I = I+1
          ELSE
            SET TEMP[INDEX] = ARR[J]
            SET J = J+1
          [END OF IF]
          SET INDEX = INDEX+1
          [END OF LOOP]
STEP 3  (Copy the remaing elements of right subarray, if any)
          IF I > MID
          Repeat while J <= END
```

SET TEMP[INDEX] = ARR[J]
SET INDEX = INDEX + 1 ; SET J = J + 1
[END OF LOOP]
[Copy the Element of left sub array, if any]
ELSE
Repeat while I <= MID.
SET TEMP[INDEX] = ARR[I]
SET INDEX = INDEX + 1, SET I = I + 1
[END OF LOOP]
[END OF IF]

STEP 4 [Copy the contents of TEMP back to ARR] SET K = 0

STEP 5 Repeat while K < INDEX.
SET ARR[K] = TEMP[K]
SET K = K + 1
[END OF LOOP]

STEP 6 IF BEG < END
SET MID = (BEG + END) / 2
CALL MERGE_SORT (ARR, BEG, MID)
CALL MERGE_SORT (ARR, MID + 1, END)
MERGE (ARR, BEG, MID, END)
[END OF IF]

STEP 7 END.

COMPEXITY O(n log n)

Date / /
Page No.

Algorithm	Average	Worst
Bubble Sort	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n^2)$
Merge Sort	$\Omega(n \log n)$	$O(n \log n)$
Heap Sort	$\Omega(n \log n)$	$O(n \log n)$
Quick Sort	$\Omega(n \log n)$	$O(n^2)$

SELECTION SORT

Selection Sort is a sorting algorithm that has a quadratic running time complexity $O(n^2)$, making it inefficient to be used on large list. It performs worse than insertion sort algorithm, it is noted for its simplicity and has performance advantages over more complicated algorithms in certain situations. Selection sort is generally used for sorting files with very large object & small keys.

Technique

- In pass 1, find the position POS of the smallest value in the array and then swap ARR[POS] and ARR[0]. Thus ARR[0] is sorted.
- In pass 2, find the position of the smallest value in sub-array of N-1 elements. Swap ARR[POS] with ARR[1]. Now ARR[0] & ARR[1] is sorted.

- In pass N-1, find the position of the elements ARR[N-2] and ARR[N-1]. ARR[POS] and ARR[N-2] so that ARR[0], ARR[1] ... ARR[N-1] is sorted.

Algorithm

SMALLEST (ARR, K, N, POS)

STEP1 [INITIALIZE] SET SMALL = K

STEP2 [INITIALIZE] SET POS = K.

STEP3 Repeat for J = K+1 to N-1

IF SMALL > ARR[J]

SET SMALL = ARR[J]

SET POS = J.

[END OF IF]

[END OF LOOP]

STEP4 RETURN POS

SELECTION SORT (ARR, N)

STEP1 (Initialize) Repeat step 2 and 3 for K = 1 to N-1

STEP2 CALL SMALLEST (ARR, K, N, POS)

STEP3 SWAP[K] with ARR[POS]

[END OF LOOP]

STEP4 EXIT

Date / /
Page No.

COMPLEXITY

Selection Sort is a sorting algorithm that is independent of the original order of elements in the array.

$(n-1) + (n-2) + \ldots + 2 + 1$
$= n(n-1)/2 = O(n^2)$.

Advantages of Selection Sorts

- It is simple and easy to implement
- It can used for small data sets.
- It is 60 percent more ~~effective~~ efficient than bubble sort

Insertion Sort

It is avery simple sorting algorithm in which the sorted array is built one element at a time. The main idea behind insertion sort is that it inserts each item into its proper place in the final list.
It is less efficient then other advanced algorithms.

* Technique

- The array of values to be stored sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially, the element with index 0 (zero) is in the sorted set. Rest of the elements are in unsorted set
- The first element of the unsorted partition has array index 1 (if LB = 0)
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Algorithm

Insertion-SORT (ARR, N)
Initialize
Step 1 Repeat Step 2 to 5 for K=1 to N-1
Step 2 set temp = ARR[K]
Step 3 Set J = K - 1
Step 4 Repeat while TEMP <= ARR[J]
SET J = J - 1
[END OF INNER LOOP]
STEP 5 SET ARR[J+1] = TEMP
[END OF LOOP]
STEP 6 EXIT.

Technique

- The array of values to be ~~stored~~ sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are 'n' elements in the array. Initially, the element with index 0 (assuming LB = 0) is in the sorted ~~set~~ set. Rest of the elements are in unsorted set
- The first element of the unsorted partition has array index 1 (if LB = 0)
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Algorithm

Insertion-SORT (ARR, N)
Initialize
Step 1 Repeat Step 2 to 5 for K = 1 to N-1
Step 2 SET TEMP = ARR[K]
Step 3 Set J = K - 1
Step 4 Repeat while TEMP <= ARR[J]
SET J = J - 1
[END OF INNER LOOP]
STEP 5 SET ARR[J+1] = TEMP
[END OF LOOP]
STEP 6 EXIT.

Date / /
Page No.

Complexity.

Best = $O(n)$
Worst Case = $O(n^2)$
Avg Case = $O(n^2)$

Advantages of Insertion Sort

- It is easy to implement & efficient for small sets of data
- It can be implemented efficiently on the data sets that are already substantially sorted.
- It requires less memory space (only $O(1)$ of additional memory space)
- It is said to be online, as it can sort a list as & when it receives new elements.
- It performs better than selection sort & bubble sort.

Quick Sort

It makes $O(n \log n)$ comparisons in the avg. case to sort an array of n elements. In worst case, it has a quadratic running time given as $O(n^2)$. It is faster than other $O(n \log n)$ algorithms, because its efficient implementation can minimize the probability of requiring quadratic time. Quick sort is also known as partition exchange sort.

It works by using a divide and conquer strategy to divide a single unsorted array into two smaller sub arrays.

Working

1- Select an element pivot from the array elements.

2- Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all the elements greater than the pivot element come after it (equal values can go either way). After such a partitioning the pivot is placed in its final position. This is called the partition operation.

3- Recursively sort the two sub arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements)

Date / /
Page No.

Technique

1) Set the index of the first element in the array to 'loc' and 'left' variables. Also set the index of the last element of the array to the 'right' variable. That is, loc = 0, left = 0 and right = n-1 (n is no. of elements in the array)

2) Start from the element pointed by right and scan the array from right to left, comparing each element on the way with the element pointed by the variable loc. That is, a[loc] should be less than a[right]

(a) If that is the case, then simply continue comparing until 'right' becomes equal to 'loc'. Once right = loc, it means the pivot has been placed in its correct position.

(b) However, if at any point, we have a[loc] > a[right], then interchange the two values and jump to step 3.

(c) Set 'loc' = 'right'.

3) Start from the element pointed by left and scan the array from left to right, comparing each element on the way with the element pointed by 'loc'.
That is, a[loc] should be greater than a[left]

(a) If that is the case, then simply continue comparing until left becomes equal to loc. Once left = loc, it means the pivot has been placed in its correct position

(b) However, if at any point, we have a[loc] < a[left], then interchange the two values and jump to step 2.

(c) Set 'loc' = 'left'

Algorithm

PARTITION (ARR, BEG, END, LOC]

step 1 INITIALIZE SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0

step 2 Repeat steps 3 to 6 while Flag = 0.

step 3 Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT.
SET RIGHT = RIGHT - 1
[END OF LOOP]

step 4 IF LOC = RIGHT
SET FLAG = 1
ELSE IF ARR[LOC] > ARR[RIGHT]
SWAP ARR[LOC] with ARR[RIGHT]
SET LOC = RIGHT
[END OF IF]

step 5 IF FLAG = 0
Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
SET LEFT = LEFT + 1
[END OF LOOP]

step 6 IF LOC = LEFT.
SET FLAG = 1
ELSE IF ARR[LOC] < ARR[LEFT]
SWAP ARR[LOC] with ARR[LEFT]
SET LOC = LEFT.
[END OF IF]
[END OF IF]

STEP 7 [END OF LOOP]

Step 8 END.

Date / /
Page No.

QUICK_SORT (ARR, BEG, END)

Step 1 IF (BEG < END).
CALL PARTITION (ARR, BEG, END, LOC).
CALL QUICKSORT (ARR, BEG, LOC-1).
CALL QUICK SORT (ARR, LOC+1, END).
[END of IF]

Step 2 END.

Complexity
WORST CASE = $O(n^2)$
AVG. CASE = $O(n \log n)$

PROS & CON's

It is faster than other algorithms such as bubble sort, Selection Sort and insertion Sort. Quick Sort can be used to sort arrays of small size, medium size, or large size. On the other side, quick Sort is complex and massively recursive.

Heap Sort

Complexity :- $n \log n$

+ − → Associativity (Left to Right)
− + → Associativity
*/ → Associativity
/* → Associativity

Date / /
Page No.

so pop first. then push operator

Linked List

Traversing a Linked List

Initialize
Step1 SET PTR = START.
Step2 Repeat Steps 3 and 4 while PTR != NULL
Step3 Apply process to PTR → DATA
Step4 Set PTR = PTR → NEXT
[END OF LOOP]

Searching in a Linked List

[INITALIZE]
Step1 SET PTR = START.
Step2 Repeat step 3 while PTR != ~~VAL~~ NULL.
Step3 IF VAL = PTR → ~~NEXT~~ DATA
SET POS = PTR.
Goto Step 5
~~Step4~~ ELSE
SET PTR = PTR → NEXT.
[END OF IF]
[END OF LOOP]
Step4 SET POS = NULL
Step5 EXIT.

Date / /
Page No.

INSERTION

Case 1 NEW NODE At the beginning

VAL → Value to be inserted
New node → new node to be inserted
AVAIL → To check if there is memory available

ALGO
[INITIALIZE]

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

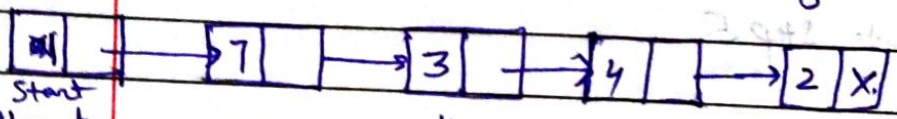
Step 4 SET NEW NODE → DATA = VAL

Step 5 SET NEW NODE → NEXT = START

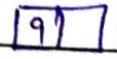
Step 6 SET START = NEW NODE

Step 7 EXIT.

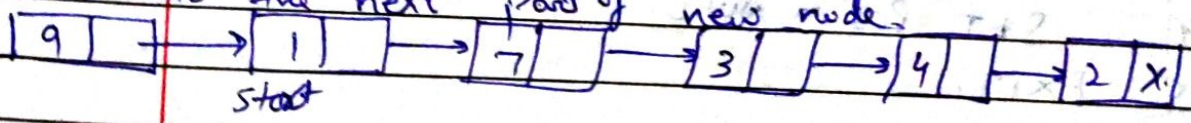
Let we need to add 9 in the given linked list.



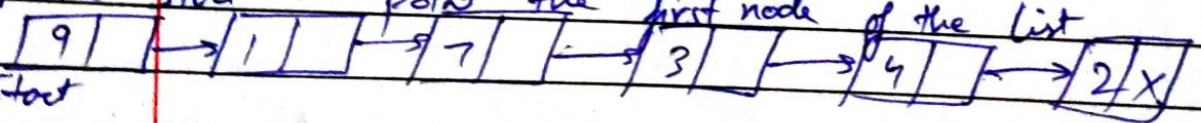
Allocate memory for the new node & initialize its data part to 9.



Add the new node as the first node & store the address of start to the next part of new node.



Make start to point the first node of the list

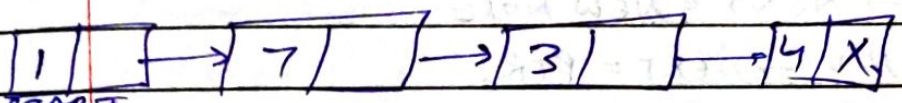


Date / /
Page No.

Case2 Inserting node at the end of Linked List

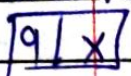
Step1 IF AVAIL = 'NULL'
Write Overflow
Goto to Step 10.
[END of IF]
Step2 Set NEW_NODE = AVAIL
Step3 Set AVAIL = AVAIL → NEXT.
Step4 Set NEW NODE → DATA = VAL
Step5 Set New NODE → NEXT = 'NULL'
Step6 Set PTR = START.
Step7 Repeat Step 8 while PTR != NULL
Step8 SET PTR = PTR → NEXT.
Step9 [END OF LOOP]
Step9 SET PTR → NEXT = NEW NODE
Step10 EXIT.

NEW NODE → node to be inserted
VAL → Value to be inserted
PTR → variable to traverse

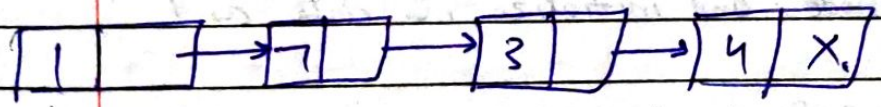


START

Allocate memory for the new node & initialize its data part as 9 and NEXT part 'NULL'

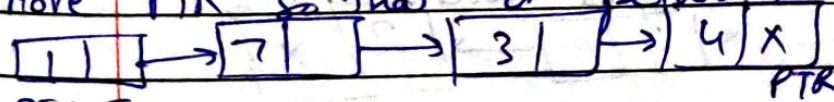


Take a pointer PTR. which points to START.



START, PTR.

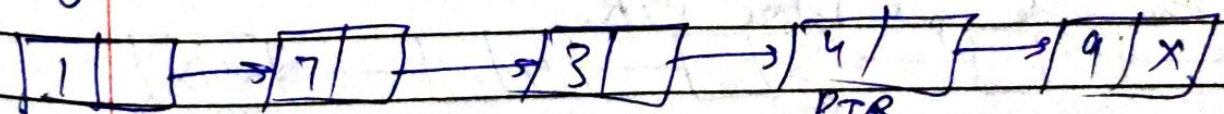
Move PTR so that it points to the last node of list



START

PTR.

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



PTR.

Date / /

Page No.

Case 3 Inserting a Node after a given node.

[Initialize]

AVAIL → Variable to check memory

New node → node to be inserted

VAL → Value to be inserted

PTR, PREPTR → Variables to traverse

NUM → Value of node after which new node is inserted

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 Step 8 and 9 while PREPTR → DATA ! = NUM.

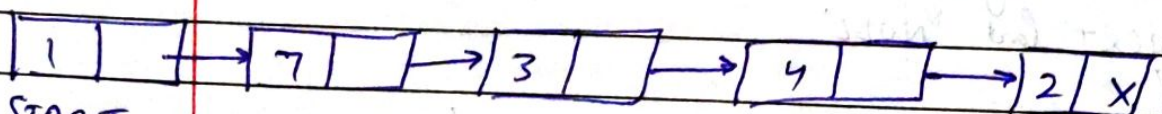
Step 8 Set PREPTR = PTR.

Step 9 SET PTR = PTR → NEXT.
[END OF LOOP]

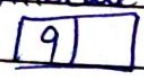
Step 10 PREPTR → NEXT = NEW NODE

Step 11 SET NEW NOD → NEXT = PTR.

Step 12 EXIT.



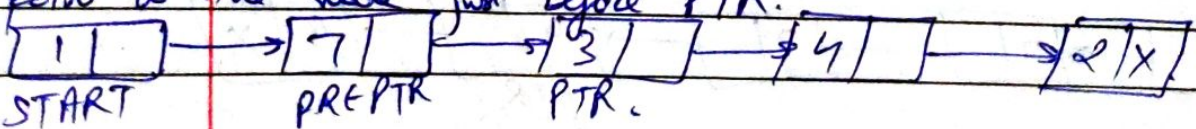
Allocate memory for new node and initialize its data as 9.



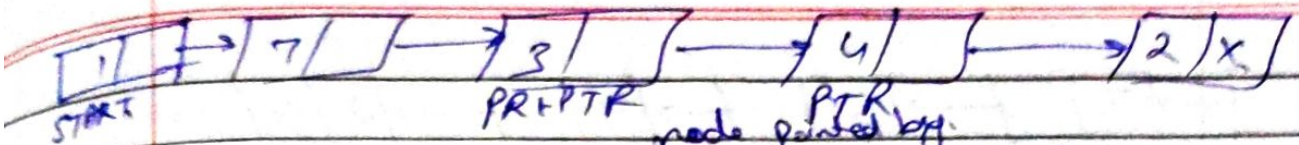
Take two pointer variable PTR & PREPTR & initialize them with start.

So PTR, PREPTR & start point to first node.

Move PTR & 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.

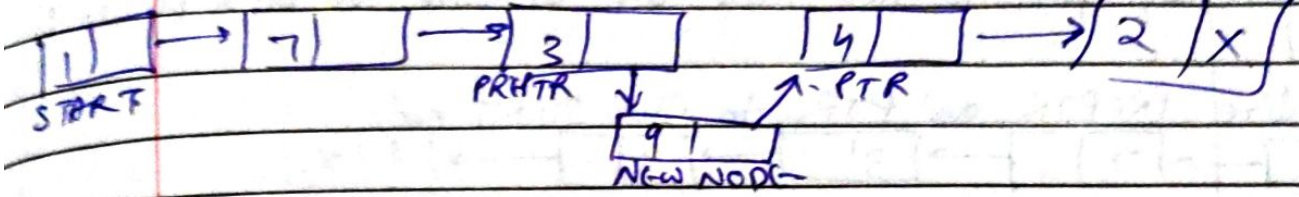


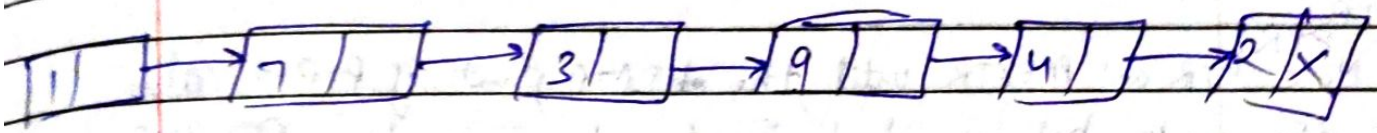
Date / /
Page No.



node pointed by.

Add new node between a PREPTR & PTR





Case 4. Inserting a node before a given node

Initialize.

Step 1 IF AVAIL = NULL
Write 'overflow'
Goto Step
[END OF IF]

Step 2 Set NEW NODE = AVAIL

Step 3 Set AVAIL = AVAIL → NEXT.

Step 4 Set NEWNODE → DATA = VALUE

Step 5 Set PTR = Start.

Step 6 Set PREPTR = PTR.

Step 7 Repeat Step and while PTR → DATA != NUM.

Step 8 SET. PREPTR = PTR.

Step 9 SET PTR = PTR → NEXT

Step 10: PREPTR → NEXT = NEW NODE

Step 11 Set NEWNODE → NEXT = PTR.

Step 12 EXIT.

AVAIL → to check memory
Newnode → node to be inserted.
VAL → Value to be inserted
NUM → Value of node before which newnode to be inserted
PTR, PREPTR → Pointer variable to traverse.

Date
Page No.

Step 1:
Allocate memory for the new node and initialize its data part to 9.

Initialize PREPTR and PTR to the START node.

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.

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

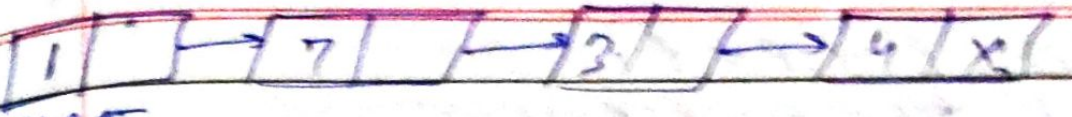
DELETION

Case 1:- Deleting the first node from a linked list

Initialize
Step 1 IF START = NULL
write UNDERFLOW
Go to Step 5
Step 2 SET PTR = START.
Step 3 SET START = START → NEXT.
Step 4 FREE PTR.
Step 5 EXIT.

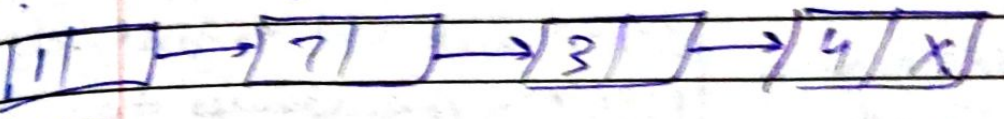
START → to point to first node
PTR → to point to the node to be deleted

Date / /
Page No.

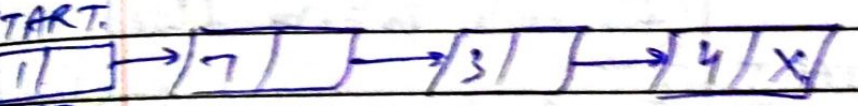


Make Start to point to the next node in sequence

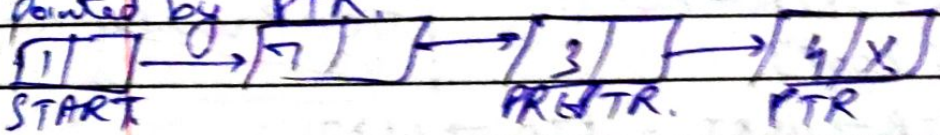
Case 2 Deleting the last node.



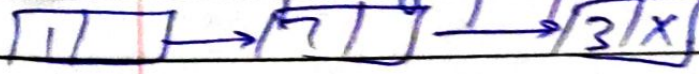
Take Pointer variable PTR and PREPTR. which initially point to START.



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



SET Next part of preptr to NULL



Initialize

Step 1 If START = NULL
Write Underflow
Go to Step 8
End of

Step 2. SET PTR = START.

Step 3 ~~SET PREPTR = PTR~~ Repeat step 4 and 5 while

Step 4 PTR → NEXT != NULL

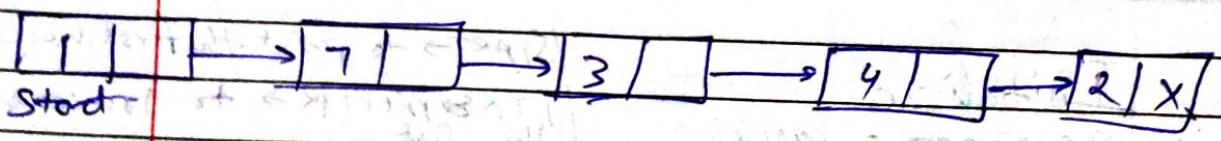
START → to point the first node
PTR & PREPTR → to traverse the list

Date / /
Page No.

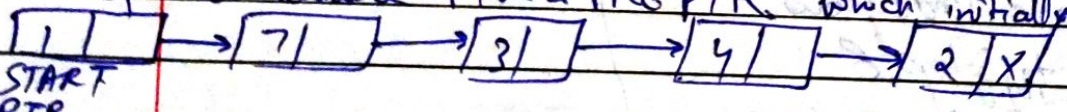
Step 4 Set PREPTR = PTR.
Step 5 set PREPTR = PTR → NEXT.
[END OF LOOP]
Step 6 set PREPTR → NEXT = NULL
Step 7 FREE PTR.
Step 8 EXIT

Case 3 Deleting node after a given node.
Initialize.
Step 1 IF START = NULL
Write 'Underflow'
Goto Step. 9.
[END OF IF]
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.
[END OF LOOP]
Step 7 SET. PREPTR → NEXT = PTR → NEXT.
Step 8 FREE PTR.
Step 9. EXIT.

START → to point to first node
PTR & PREPTR ⇒ Pointer variables to traverse the list
NUM ⇒ Value of node after which the node is to be deleted
If used
TEMP = to store value of node after which needs to be
We can add PTR to TEMP variable

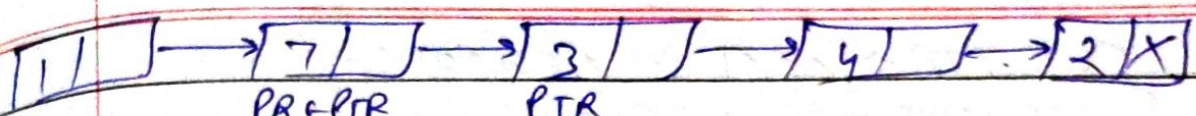


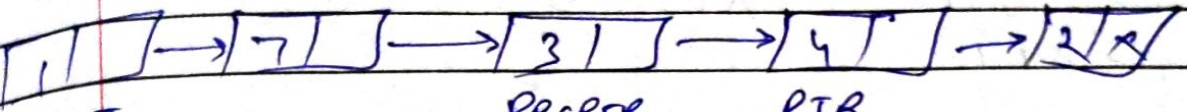
Take pointer variable PTR & PREPTR which initially point to start



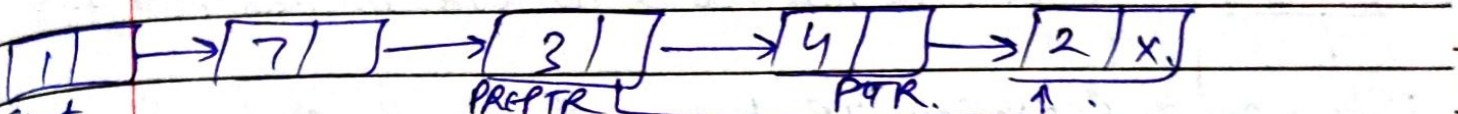
Move PTR & PREPTR such that PREPTR points the node whose succeeding node needs to be deleted.

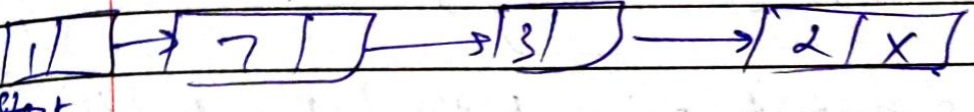
Date / /
Page No.





Set next part of PREPTR to next part of PTR





B. Circular linked list

Insertion

Case 1 New node needs to be inserted at the beginning.

INITIALIZE

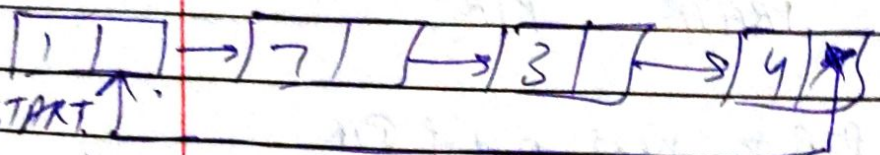
Step 1 IF AVAIL = NULL
Write Overflow
Goto Step. 11
~~Step 2~~ [END OF IF]
Step 2 SET. NEWNODE = AVAIL
Step 3 SET AVAIL = AVAIL → NEXT.
Step 4 SET NEWNODE → DATA = VAL
Step 5 SET PTR = START.
Step 6 Repeat Step while PTR → NEXT != START.
Step 7 SET PTR = PTR → NEXT
[END OF LOOP]
Step 8 SET NEWNODE → NEXT = START.

AVAIL → to check memory.
NEWNODE → node to inserted
VAL → value to be inserted
PTR → Pointer variable to traverse the list.

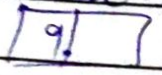
Pg 182

Date / /
Page No.

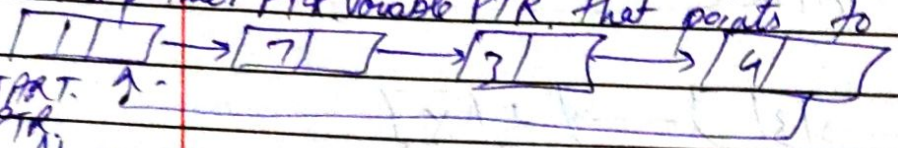
Step 9 SET. PTR → NEXT = NEWNODE.
~~Step 10 NEWNODE = START~~
Step 10. SET. START = NEWNODE.
Step 11 EXIT.



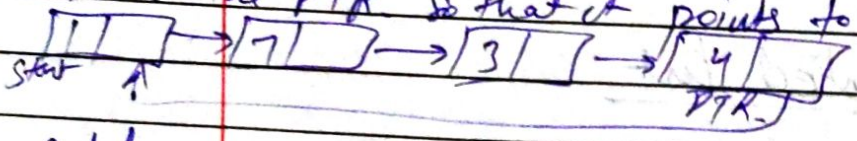
Allocate memory for the new node & initialize its data part to 9



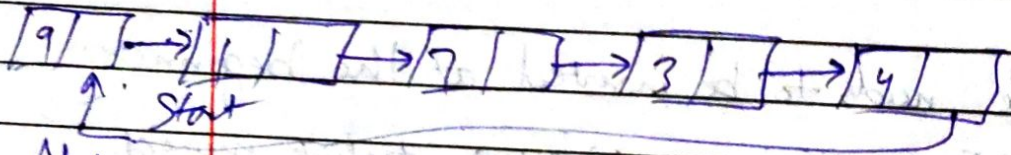
Take a pointer ~~PTR~~ variable PTR that points to the start node.



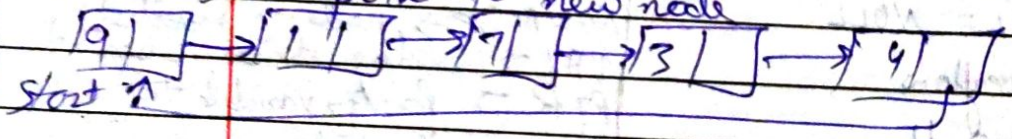
Move the PTR so that it points to the last node of list



Add new node inbetween PTR & Start



Make Start point to new node



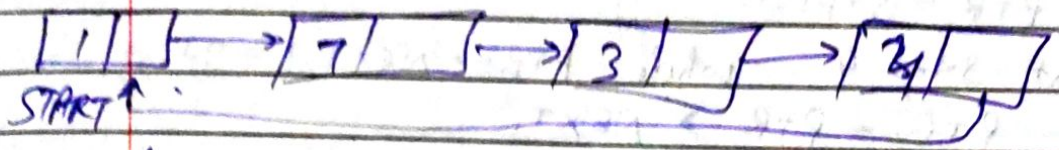
Case 2. Inserting a node at the end of a list

Initialize
Step 1 IF AVAIL = NULL
Write Overflow
Goto Step 10
[END OF IF]
Step 2 Set NewNODE = AVAIL
Step 3 Set AVAIL = AVAIL → NEXT

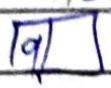
AVAIL :- to check memory.
Newnode :- node to be inserted
VAL :- Value to be inserted
PTR :- Pointer variable to traverse the list.

Date / /
Page No.

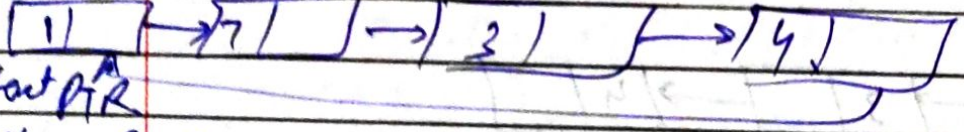
Step4 Set NewNode → DATA = VAL.
Step5 Set NewNode → NEXT = START.
Step6 Set PTR = START
Step7 Repeat Step while PTR → NEXT != START
Step8 SET PTR = PTR → NEXT.
[END OF LOOP]
Step9. SET PTR → NEXT = NEW NODE
Step10 EXIT



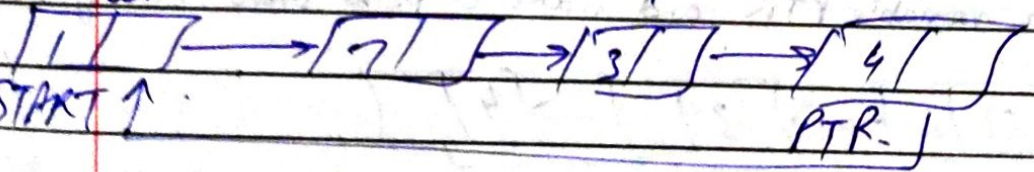
Allocate memory to newnode & initiate the data part to 9.



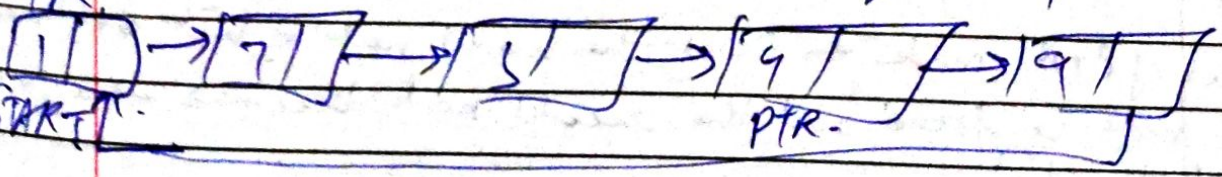
Take a pointer variable PTR which will initially point to START



Move PTR so that now it points the last node of the list



~~The next part of PTR is made to point the second node of the list and the memory of the first node is freed~~
Add the new node after the node pointed by PTR.



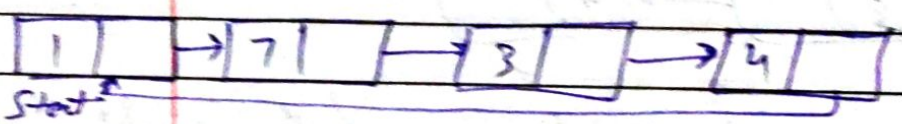
Date / /
Page No.

DELETION

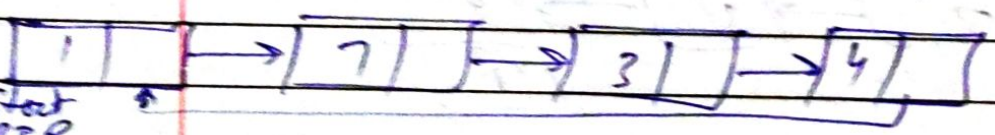
Case 1 | Deleting the first Node | Start :- to point the starting node
PTR = pointer variable to traverse the list

Step 1 | If Start = NULL
Write Underflow
Goto Step 8.
[End of If.]

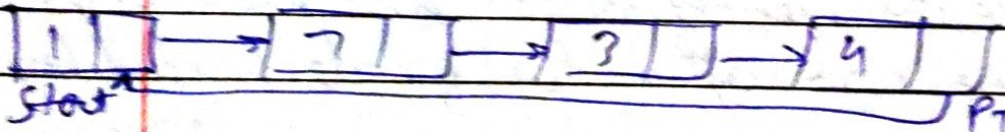
Step 2 | SET PTR = START.
Step 3 | Repeat Step 4 while PTR → NEXT != Start.
Step 4 | Set PTR = PTR → NEXT.
[END OF LOOP]
STEP 5 | SET PTR → NEXT. = START → NEXT
Step 6 | FREE START.
Step 7 | SET START = PTR → NEXT.
Step 8 | EXIT.



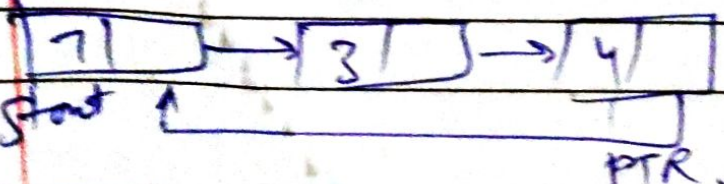
Take a pointer variable PTR and make it point start node of list.



Move PTR further so that now it points the last node of list



The next of PTR made to point the second node of list and memory of first node is freed and the second node becomes the first node of the list



Date / /
Page No.

Case 2 Deleting the last Node

Initialize

Step 1 If START = 'NULL'
Write 'Underflow'
Goto ~~Point~~ Step 8 [END OF IF]

Step 2 Set PTR = START

Step 3 Repeat Step while PTR → NEXT ≠ START

Step 4 SET PREPTR = PTR.

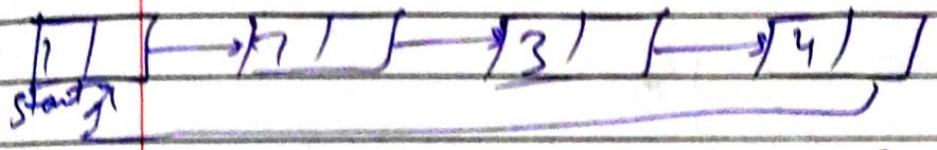
Step 5 SET PTR = PTR → NEXT

Step 6 [END OF LOOP]

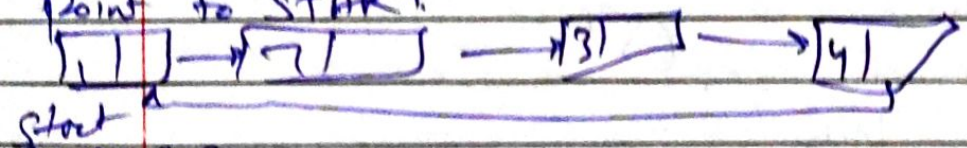
Step 7 Set PREPTR → NEXT = START.

Step 7 FREE PTR.

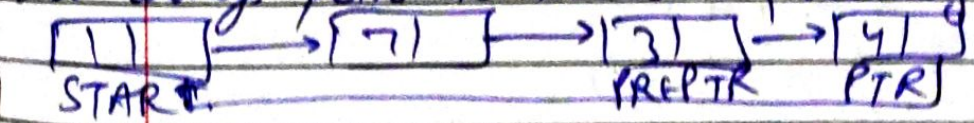
Step 8 EXIT



Take two pointers PTR & PREPTR which will initially Point to START.



Move PTR so that it points the last node of list, PREPTR will always Point to the node preceding PTR.



Make the prepTR next part store START node's address and free the space allocated to PTR, Now PREPTR is the last node of list