Chapter 4

Linked Lists, Stacks and Queues

LEARNING OBJECTIVES

- *Data structure*
- *Linked list*
- *Single-linked list*
- *Double-linked list*
- *Circular linked list*
- *Double circular-linked list*
- *Stack*
- *Queue*
- *Double-ended queue*
- *Circular queue*
- *Priority queue*
- *Array implementation*
- *Linked list implementation*
- *Linked list implementation of priority queue*

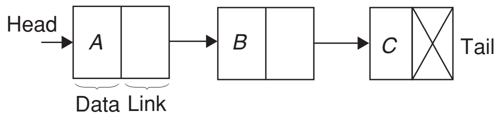
Data Structure

Data structure represents the logical arrangement of data in computer memory for easily accessing and maintenance.

Linked List

A linked list is a data structure that consists of a sequence of nodes, each of which contains data field and a reference (i.e., link) to next node in sequence.

- Generally node of linked list is represented as self-referential structure.
- The linked list elements are accessed with special pointer(s) called head and tail.



- The principal benefit of a linked list over a conventional array is that the list elements can easily be added or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk.
- Linked lists allow insertion and removal of nodes at any point in the list.
- Finding a node that contains a given data, or locating the place where a new node should be inserted may require scanning most or all of the list elements.
- The list element does not have to occupy contiguous memory.
- Adding, insertion or deletion of list elements can be accomplished with the minimal disruption of neighbouring nodes.

Single-linked List

List in which each node contains only one link field.

Node structure

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

typedef struct node Node;

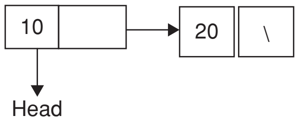
Creating a linked list with two nodes of type list node

Creating a linked list with 2 nodes
struct node

```
{
Int ele;
struct node * next ;
};
typedef struct node Node ;
Node * ptr1, * ptr2;
ptr1 = getnode ();
ptr2 = getnode ();
if((ptr1) && (ptr2))
{
Printf("No memory");
exit(1);
}
Ptr1 → ele = 10;
```

```
Ptr1 → next = ptr2;
Ptr2 → ele = 20;
Ptr2 → next = NULL;
Head = ptr1;
```

the linked list appears as below



Operations on SLL (single-linked list)

- Insert at Head
- Insert at Tail
- Insert in Middle
- Delete Head
- Delete Tail
- Delete Middle
- Search
- Display

Declare two special pointers called head and tail as follows:
Node *Head, *Tail;
Head = Tail = NULL;

Head or tail is NULL represents list is empty.
Steps for Insertion:

1. Allocate memory
2. Read data
3. Adjust references

Insert head element

```
1. void ins _ Head (int x)
2. {
3. Node *temp;
4. temp = (Node *) malloc(sizeof (Node));
5. temp → ele = x;
6. temp → next = Head;
7. Head = temp;
8. if (Tail = = NULL)
9. Tail = Head;
10 }
```

- Step 4 allocates memory
- Step 5 read data
- Steps from 6 to 9 adjust reference
- 'if' condition represents first insertion

Insert tail element

```
1. void ins_tail (int x)
2. {
3. Node *temp;
4. temp = (Node *) malloc (sizeof (Node));
5. temp → ele = x;
6. temp → next = NULL;
7. Tail = temp;
8. if (Head = = NULL)
9. Head = Tail;
10. }
```

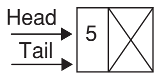
- Step 4 allocates memory
- Step 5 read data
- Steps from 6 to 9 adjust reference
- 'if' condition represents first insertion

Insert in middle/random position of list

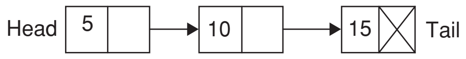
```
1. void ins _ mid (int n, int pos)
2. {
   int i = 1;
3. Node  *  temp,  N,  P;  //N,P  represent
   previous //& next nodes
4. if (Head == NULL)
5. {
6. ins _ head(n);
7. return;
8. }
9. temp = (Node *) malloc(sizeof(Node));
10. temp → ele = n;
11. P = head;
12. while (i < pos -1)
13. {
   P = P → next;
   i++;
   }
14. N = P → next;
15. temp → next = N;
16. P → next = temp;
17. }
```

- step 4 checks, whether the insertion is into an empty list.
- If list is empty, invokes ins–head( ) function.
- If list is not empty, then step 9 allocates memory.
- Step 10 reads data.
- Steps from 11 to 14 make the reference to the previous and next nodes of new node to be inserted.
- Steps 15 and 16 create the reference to new node from previous node and from new node to next node.

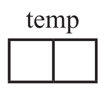
Example 1: Head = Tail = NULL
$n = 5$, P = NULL;
Here the list is empty. So,



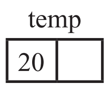
Example 2:



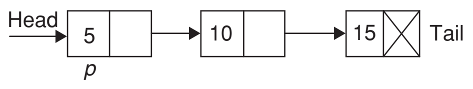
Insert element (n) 20 at position(pos) 3.
In current list, element 5 is the first element, 10 is the second and 15 is the third element.
To insert an element at pos = 3, the new node has to be placed between elements 10 and 15.
Condition in step 4 is false so step 9 executes and allocates memory.



On completion of step 10 –



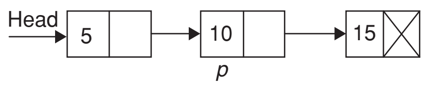
Step 11



Step 12, 13

```
While (i < pos - 1)
{
P = P → next;
i++;
}
i < pos
1 < 2
```

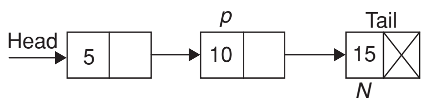
Condition true, so



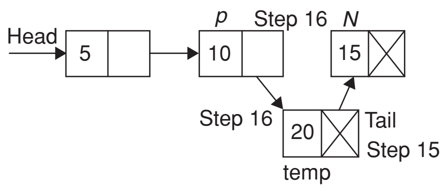
i becomes 2,

2 < 2 // condition false

Step 14 makes a reference to next of previous element.



Steps 15 and 16 execute as follows:



Now the element 20 becomes the 3rd element in the list.

Deletion

- Identify the node
- Adjust the links, such that deallocation of that node does not make the list as unconnected components.
- Return/display element to delete.
- Deallocate memory.

Delete head element

```
1. void del _ head()
2. {
3. int x;
   Node * temp;
4. if (Head = = NULL)
5. {
6. printf("List empty");
7. return;
8. }
9. x = Head → ele;
10. temp = Head;
11. if (Head = = Tail)
12. Head = Tail = NULL;
13. else
14. Head = Head → next;
15. printf ("Deleted element "%d", x);
16. free(temp);
17. }
```

Step 4	–	Checks for list empty
Step 9	–	Reads element to delete
Step 10	–	Head referred by temp pointer
Step 11	–	Checks for last deletion
Step 14	–	Moves the head pointer to next element in the list
Step 15	–	Displays element to delete
Step 16	–	Deallocates memory

Delete tail element

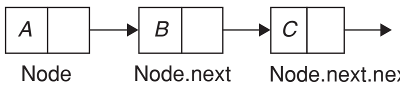
```
1. void del _ tail()
2. {
3. int x;
4. Node * temp;
5. if (Head = = NULL)
6. {
7. printf("\n list empty")
8. return ;
9. }
10. temp = Head;
11. while(temp → next ! = Tail)
12. temp = temp → next;
13. x = Tail → ele;
14. Tail = temp;
15. temp = temp → next;
16. Tail → next = NULL;
17. printf("\n Deleted element : %d", x)
18. free (temp);
19. }
```

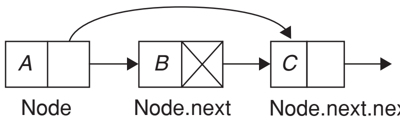
Step 4	– Checks for list empty
Step 10, 11, 12	– Move the temp pointer to last but one node of the list
Step 13	– Reads tail element to delete
Step 14	– Moves tail pointer to last but one node
Step 15	– Moves the temp pointer to last node of the list
Step 16	– Removes the reference from tail node to temp node, i.e., tail node becomes the last element
Step 17	– Displays elements to delete
Step 18	– Deallocate memory

Delete middle element

```
 1. void del _ mid (int pos)
 2. {
 3. int i = 1, x;
 4. Node * temp P, N;
 5. if(Head = = NULL)
 6. {
 7. printf ("\n list empty")
 8. return;
 9. }
10. P = head;
11. while (i < pos -1)
12. {
    P = P → next;
    i++ ;
    }
13. temp = P → next;
14. N = temp → next;
15. P → next = N;
16. x = temp → ele;
17. printf("\n Element to Delete %d", x);
18. free(temp);
19. }
```

Step 5 – Checks for empty list
Step 10, 11, 12 – Move previous pointer *P* to previous node of node to delete.
Step 13 – Temp points to node to delete
Step 14 – N points to temp next
Step 15 – Creates link from *P* to *N*
Steps 16, 17, 18 – Read and display elements to delete and deallocate memory.





Linked list using dynamic variables

Node in the linked list contains data part that is ele and link part which points to the next node, and some other external pointer will be pointing to this as these take some storage, a programmer when creating a list, should check with the available storage. For this we make use of get node ()

Function which is defined as follows:

```
struct node
{
int ele
struct node * next ;
};
typedef struct node Node;
Node getnode ()
{
Node ptr;
ptr = (Node *) malloc (size of (struct node)):
return (ptr);
}
```

If ptr returns NULL, then it is underflow (there is no available memory) otherwise, it returns start address of memory location.

Search an element

```
 1. void search (int x)
 2. {
 3. Node * temp = head;
 4. int  c = 1;
 5. while (temp! = NULL)
 6. {
 7. if (temp → ele = = x)
 8. {
 9. printf("\n Element found at % d", c);
10. break;
11. }
12. c++;
13. }
14. if (temp = = NULL)
15. printf("\n search unsuccessful");
16. }
```

Step 7 – Checks temp data with search element. Repeats this step until the element is found or reaches the last node
Step 9 – Displays the position of search element in the list, if found
Step 14, 15 – Represents search element not exists in list

Display

```
 1. void display ( )
 2. {
 3. Node *temp = Head;
 4. printf("\n list elements: ");
 5. while (temp ! = NULL)
 6. {
 7. printf("%d", temp → ele);
 8. temp = temp → next;
 9. }
10. }
```

Step 7 – Displays temp data
Step 8 – Moves temp pointer to next node

Algorithm to reverse direction of all links of singly liked list

Consider a linked list ‘*L*’ with head as pointer pointing to the first node contains data element ‘ele’ and a pointer called ‘next’ which points to the next node.

Reverse is the routine which will reverse the list, there are three node pointers *P*, *Q*, *R* with *P* pointing to the first node, *Q* pointing to NULL.

```
 1. START
 2. if (P = NULL)
    1. print ("List is null");
    2. Exit
 3. While (P)
 4. R = Q;
 5. Q = P;
 6. P = P → next;
 7. Q → next = R
 8. End While
 9. Head = Q;
10. STOP
```

Double-linked List (DLL)

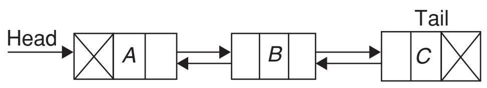
Double-linked list is a linked list in which, each node contains data part and two link fields.

Node structure:

```
struct Dnode
{
struct Dnode *prev;
int ele;
struct Dnode *next;
};
```

- prev – points to previous node in list
- next – points to next node in list
- The operations which can be performed in SLL can also be preformed on DLL.
- The major difference is that we have to adjust double reference as compared to SLL.
- We can traverse or display the list elements in forward as well as in reverse direction.

Example:



Circular-linked List (CLL)

Circular-linked list is completely same as SLL, except, in CLL the last (Tail) node points to first (Head) node of list.

So, the Insertion and Deletion operation at Head and Tail are little different from SLL.

Double Circular-linked List (DCL)

Double circular-linked list can be traversed in both directions again and again. DCL is very similar to DLL, except the last node's next pointer points to first node of list and first node's previous pointer points to last node of list.

So, the insertion and deletion operations at head and tail in DCL are little different in adjusting the reference as compared to DLL.

Storing ordered table as linked list: The table is stored as a linked list, it is retrieved and stored with two pointers, one pointer will point to node holding a record having the smallest key and other pointer performs the search.

Stack

A stack is a last in first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterized by only two fundamental operations.

√ PUSH
√ POP

- The PUSH operation adds an item to the top of the stack, hiding any items already on the stack or initializing the stack if it is empty.
- The POP operation removes an item from the top of the stack, and returns the poped value to the caller.
- Elements are removed from the stack in the reverse order to the order of their insertion. Therefore, the lower elements are those that have been on the stack for longest period.

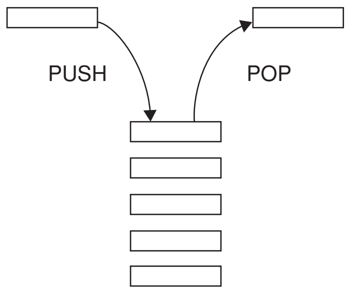


Figure 1 Simple representation of a stack

Implementation

A stack can be easily implemented either through an array or a linked list. The user is only allowed to POP or PUSH items onto the array (or) linked list.

1. **Array Implementation:** Array implementation aims to create an array where the first element inserted is placed st[0] which will be deleted last.

   The program must keep track of position top (last) element of stack.

   Operations

   Initially Top = –1;//represents stack empty

```
 (i) Push (S, N, TOP, x)
     {
     if (TOP = = N - 1)
     printf("overflow");
     else
     TOP = TOP + 1;
     S[TOP] = x;
     }
(ii) POP (S, N, TOP, x)
     {
     if (TOP = = -1)
     printf("underflow");
     else
     x = S[TOP]
     TOP = TOP - 1
     return x;
     }
```

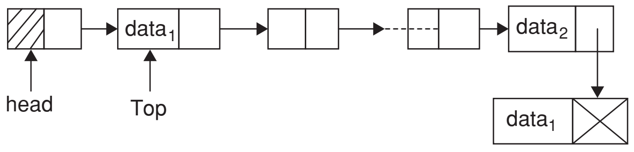
2. **Dynamic Implementation:** The Array implementation is also called static implementation, because the stack size is fixed.

   The stack implementation using linked list is called dynamic implementation, because the stack size can grow and shrink as the elements added or removed from the stack.

   - The PUSH operation on stack is same as insert head in SLL.
   - The POP operation is same as delete head in SLL.

Algorithm to add and delete to a link stack and link queue

Link stack:



The linked stack with head and top pointers is shown above

The algorithm to push the elements into stack is given below, the method push (item)

Steps:

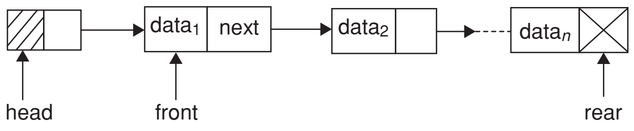
```
1. ptr = getnode (Node)
2. ptr.data = item
3. ptr.next = Top
4. Top = new
5. Head.next = Top
6. Stop.
```

for deletion of elements from stack, its algorithm is pop(), it is given below

Steps:

```
1. if (Top = NULL)
   1. print "stack is empty"
   2. exit
2. Else
   1. ptr = Top.next
   2. item = Top.data
   3. Head.next = ptr
   4. Top = ptr
3. End if
4. Stop.
```

Linked queue representation



The linked queue with head, front and rear point is shown above.

The algorithm to enqueue the elements into queue is given below, the method enqueue (item)

Steps:

```
1. ptr = getNode (Node)
2. ptr.data = item
3. ptr.next = NULL
4. if (front = NULL)
      front = ptr
      else
      rear.next = ptr;
5. end if
6. rear = ptr
7. Stop
```

For deletion of elements from queue that is ptr dequeue () is given below

Steps:

```
1. if (front = NULL)
   1. print "underflow".
   2. exit
2. ptr = front;
3. front = ptr.next
4. Head.next = front
5. item = ptr.data
6. free(ptr)
7. end.
```

Uses of Stack

- Function calls: When a function is called all local storage for the function is allocated on system 'stack', and return address also pushed on to system stack.
- Recursion stacks can be used to implement recursion if the programming language does not provide recursion facility.
- Reversing a list
- Parsing: Stacks are used by compilers to check the syntax of program.
- For evaluating expressions.

Expression Notations

Infix expression: Here binary operator comes between the operands.

Postfix expression: Here the binary operator comes after both the operands.

Example: *ab+*

Prefix expression: Here the binary operator comes before both the operands.

Example: *+ab*

Infix to postfix conversion

- If operand, output to postfix expression
- If operator, push it onto stack
- In case of parenthesis, when an opening parenthesis is read, it is pushed onto stack and when a closing parenthesis is read, all operators up to the first opening parenthesis must be popped from the stack into the post fix notation.

Example: $(A + (B - C))*D$

i/p	Postfix notation	Stack
(		(
A	*A*	(
+	*A*	(+
(	*A*	(+(
B	*AB*	(+(
–	*AB*	(+(–
C	*ABC*	(+(–
)	*ABC–*	(+
)	*ABC–+*	–
*	*ABC–+*	*
D	*ABC–+D*	*
	*ABC–+D**	

Evaluation of postfix expression

We use operand stack for evaluation. Scan the post fix expression,

- When an operand encounters while scanning, push on to stack.
- While scanning post fix expression, if operator found then
    - Pop top two operands from stack
    - Perform the operation on those two operands
    - Push, result on to stack top
- Finally, the stack contains only one value, which represents result of the expression.

Example: 6 2 3 + – 3 8 2 / + 2 3 +

Symbol	OP1	OP2	Value	Operand stack
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6,5
–	6	5		1
3				1, 3
8				1, 3, 8
2				1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2				7, 2
*	7	2	14	14
3				14, 3
+	14	3		17

Result is 17.

Performing add, delete operations on stack (multiple stack)

Let us consider an array whose size is '<u>max</u>'
with multiple stack *A*, *B* having top *A* and top *B*, push and pop operations on one stack *A* is given below.

Algorithm for push A(x)

Initially *A*[Max], top *A* = –1, top *B* = MAX;

```
1. if (top A = top B)
   a. print " overflow"
   b. exit
2. top A = top A + 1
3. A[top A] = x
4. stop
```

Algorithm for pop A(x)

```
1. if (top A = - 1)
   a. print "underflow"
   b. exit
2. y = A[top A]
3. top A = top A - 1
4. return y
5. stop
```

Algorithm for push B(x)

```
1. if (top B - 1 = top A)
   a. print "overflow"
   b. exit
2. top B = top B -1
3. A[top B] = x
4. stop
```

Algorithm for pop B(x)

```
1. if (top B = max)
   a. print "underflow"
   b. exit
2. y = A [top B]
3. top B = top B - 1
4. return y
5. stop
```

QUEUE

A queue is an ordered collection of items from which items may be deleted at one end (called that front of queue) and into which items may be inserted at the other end (called <u>rear</u> of queue).

Queue is a linear data structure maintains the data in first in−first out (FIFO) order.

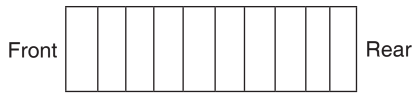
Implementation

Queue can be implemented in the following ways:

1. Array static implementation: queue cannot be extended beyond the array size.
2. Linked list dynamic implementation: Queue size increases as the elements added/inserted to queue. Queue shrinks when an element deleted from queue.

Array Implementation

const int SIZE = 10;
int q[SIZE];
int $f = -1$, $r = -1$; //$f = r = -1$ represents queue empty



Insertion

```
 1. void insert (int x)
 2. {
 3. if (r = = SIZE -1)
 4. {
 5. printf("Q FULL")
 6. return;
 7. }
 8. r++;
 9. q[r] = x;
10. if (f = = -1)
11. f = r;
12. }
```

Step 3 – Checks for queue full
Step 8 – Increments rear (r)
Step 9 – Inserts 'x' into queue
Step 10 – Checks whether insertion is first
Step 11 – If first insertion, updates front (f)

Deletion

```
 1. void deletion()
 2. {
 3. int x;
 4. if (f = = -1)
 5. {
 6. if ("\n Q Empty");
 7. return;
 8. }
 9. x = q[f];
10. if (f = = r)
11. f = r = -1;
12. else
13. f++;
14. printf("\n deleted element %d", x);
15. }
```

Step 4 – Checks for queue empty
Step 9 – Deletes 'q' front element
Step 10 – Checks whether queue having only one element
Step 11 – Rear and front initializes to –1, if queue is having only one element
Step 13 – Queue front points to next element
Step 14 – Deleted element is printed

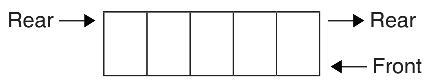
Display

```
 1. void display( )
 2. {
 3. int i = f;
 4. if (f = = -1)
 5. {
 6. printf("Queue Empty");
 7. return;
 8. }
 9. printf ("\n Queue Elemetns");
10. for(; i < = r; i++)
11. printf(" %d", q[i]);
12. }
```

Step 4 – Checks for 'q' empty
Step 10 and 11 – Display 'q' elements

Double-ended Queue

A double-ended queue (deque) is an abstract data structure that implements a queue for which elements can only be added to or removed from the front (head) (or) rear (tail) end.



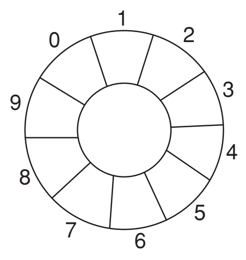
Insertions and deletions are possible at both ends.

Linked List Implementation Double-ended Queue

- Insert – Front is same as insert – Head
- Insert – Rear is same as insert – Tail
- Delete front is same as delete – Head
- Delete – Rear is same as delete – Tail

Circular Queue

As the items from a queue get deleted, the space for that item is reclaimed. Those queue positions continue to be empty. This problem is solved by circular queues. Instead of using a linear approach, a circular queue takes a circular approach; this is why a circular queue does not have a beginning or end.



The advantage of using circular queue over linear queue is efficient usage of memory.

Algorithm to implement addition and deletion from circular queue

Circular Queue Insertion:
To add an element 'X' to a Queue 'Q' of size 'N' with front and rear pointers as 'F' and 'R' is done with insert (X), Initially $F = R = 0$.
Insert (X)

Steps:

```
1. if (((R = N ) & & (F = 1)) or ((R +
   1) = F))
   a. print "overflow"
   b. exit
2. if (R = N)
   then R = 0;
   Else
      R = R + 1;
3. Q[R] = x;
4. if (F = 0)
   F = 1
5. Stop.
```

To delete an element we implement an algorithm delete (). '*y*' contains the deleted element.

delete()

Steps:

```
1. if (F = 0)
   a. print "underflow"
   b. exit
2. y = Q[F]
3. if (F = R)
   F = R = 0
   else
   If (F = N)
   F = 1
   Else
   F = F + 1
4. Return y
5. Stop.
```

Priority Queue

In priority queue, the intrinsic ordering of elements does determine the results of its basic operations.

There are two types of priority queues.

- Ascending priority queue is a collection of items in which items can be inserted arbitrarily and from which only the smallest items can be removed.
- Descending priority queue is similar but allows deletion of the largest item.

Array Implementation

- The insertion operation on priority queue selects the position to the element to insert.
- Makes the position empty/free by moving the existing element (if required).
- Place the element in required position.
- Deletion operation simply deletes front of queue.

Linked-list Implementation

- Insertion operation create a node
- Reads element into node
- Find out the location
- Insert the node into list, by adjusting the reference
- Deletion operation simply deletes head elements, making the head next as head element

Linked-list Implementation of Priority Queue

- Insertion in queue is same as insert-tail of queue
- Deletion from queue is same as delete head

Exercises

Practice Problems I

Directions for questions 1 to 16: Select the correct alternative from the given choices.

1. If the array representation of a circular queue contains only one element then

(A) front = rear (B) front = rear + 1
(C) front = rear − 1 (D) front = rear = NULL

2. The five items *P*, *Q*, *R*, *S* and *T* are pushed in a stack, one after another starting from *P*. The stack is popped four times, and each element is inserted in a queue. The two elements are deleted from the queue and pushed back on the stack. Now one item is popped from the stack. The popped item is ______.

(A) *P* (B) *Q*
(C) *R* (D) *S*

3. What are the contents of the stack (initially the stack is empty) after the following operations?

PUSH (A)
PUSH (B)
PUSH (C)
POP
PUSH(D); POP; POP;
PUSH(E)
PUSH(F)
POP

(A) ABE (B) AE
(C) A (D) ABCE

4. Consider the below code, which deletes a node from the beginning of a list:

```
void deletefront()
{
if(head == NULL)
return;
else
{
..........
.........
.........
}
}
```

Which lines will correctly implement else part of above code?

(A)
```
if (head → next = = NULL)
head = head → next;
```
(B)
```
if (head = = tail)
head = tail = NULL;
else
head = head → next;
```
(C)
```
if (head = = tail = = NULL)
head = head → next;
```
(D)
```
head = head → next;
```

5. When a new element is inserted in the middle of linked list, then the references of _____ to be adjusted/updated.

(A) those nodes that appear after the new node
(B) those nodes that appear before the new node
(C) head and tail nodes
(D) those nodes that appear just before and after the new node

6. The following C function takes double-linked list as an argument. It modifies the list by moving the head (first) element to tail of the list.

```
typedef struct node
{
    struct node *p;
    int data;
    struct node *n;
} Node;
Node * Move - to - last (Node *head)
{
Node * temp, * prev, * next;
if (head = = NULL)||(head → n = = NULL))
return head;
temp = head;
prev = head;
head = head → n;
while (prev → n! = NULL)
{
X;
}
Y;
return head;
}
```

(A)
```
X: prev = prev → n;
Y: prev → n = temp;
temp → p = prev;
temp → n = NULL;
head → P = NULL;
```
(B)
```
X: next = prev → n;
Y: prev → n = temp;
temp → p = prev;
```
(C)
```
X: prev = prev → n;
Y: prev → n = temp;
temp → n = NULL;
head → p = NULL;
```
(D)
```
X: next = prev → n;
prev = prev → n;
Y: prev → n = Next;
next → n = head;
temp → n = NULL;
```

7. Which of the following program segment correctly inserts an element at the front of the linked list. Assume that Node represents linked list node structure, value is the element to be inserted.

(A)
```
temp = (Node *)malloc (sizeof (Node));
temp → data = value;
temp → next = head;
head = temp;
```
(B)
```
temp = (Node *)malloc(sizeof (Node*)
);
temp → data = value;
temp → next = head;
head = temp;
```
(C)
```
temp = (Node *)malloc (sizeof (Node));
head = temp;
temp → next = head;
temp → data = value;
```
(D)
```
temp = (Node *)malloc (sizeof (Node
*);
temp → data = value;
head = temp;
temp → next = head;
```

8. Consider the following program segment:

```
struct element
{
    int x;
    struct element *link;
}
void shuffle(struct element *head)
{
struct *p, *q;
int t;
if (!head || !head → link) return;
p= head ; q = head → link;
while(q)
{
t = p → x;
p→ x = q → x;
q → x = t;
p = q → link;
q = p? p : 0;
}
}
```

The function called with list containing 10, 15, 20, 25, 30, 35, 40 in given order. What will the order of elements of the list, after executing the function shuffle?

(A)	10	15	20	25	30	35	40
(B)	40	35	30	25	20	15	10
(C)	20	15	10	25	40	35	30
(D)	15	10	25	20	35	30	40

9. Primary ADT's are

(A) Linked list only (B) Stack only
(C) Queue only (D) All of these

10. Linked list uses NULL pointers to signal
(A) end of list (B) start of list
(C) Either (A) or (B) (D) Neither (A) nor (B)

11. Which of the following is essential for converting an infix to postfix form efficiently?
(A) Operator stack (B) Operand stack
(C) Both (A) and (B) (D) Parse tree

12. Stacks cannot be used to
(A) Evaluate postfix expression
(B) Implement recursion
(C) Convert infix to postfix
(D) Allocate resource like CPU by the operating system

13. Linked list can be sorted
(A) By swapping data only
(B) By swapping address only
(C) Both (A) and (B)
(D) None of these

14. Linked list are not suitable for implementing
(A) Insertion sort
(B) Binary search
(C) Radix sort
(D) Polynomial manipulation

15. Insertion of node in a double-linked list requires how many changes to previous (prev) and next pointers?
(A) No changes (B) 2 next and 2 prev
(C) 1 next and 1 prev (D) 3 next and 3 prev

16. Minimum number of stacks required to implement a queue is
(A) 1 (B) 2
(C) 3 (D) 4

Practice Problems 2

Directions for questions 1 to 11: Select the correct alternative from the given choices.

1. Stack is useful for implementing _____.
(A) radix sort
(B) breadth first search
(C) quick sort
(D) recursion

2. Which is true about linked list?
(A) A linked list is a dynamic data structure.
(B) A linked list is a static structure.
(C) A stack cannot be implemented by a linear linked list.
(D) None of the above

3. The process of accessing the data stored in a tape is similar to manipulating data on a _____.
(A) stack (B) list
(C) queue (D) heap

4. Which of the following is used to aid in evaluating a prefix expression?
(A) Queue (B) Heap
(C) Stack (D) Hash

5. Select the statement which best completes the sentence — 'Abstract data type is…'
(A) a data type which is abstract in nature
(B) a kind of data type
(C) data structure
(D) a mathematical model together with a set of operations defined on it

6. Which of the following data structures may give an overflow error, even through the current number of elements in it is less than its size?
(A) Simple queue (B) Circular queue
(C) Stack (D) None of these

7. In a circular linked list, insertion of a record involves the modification of _____.
(A) no pointer (B) four pointers
(C) two pointers (D) All of the above

8. Among the following, which one is not the right operation on a stack?
(A) Remove the item that is inserted latest into the stack.
(B) Add an item to the stack.
(C) Remove the first item that is inserted into the stack, without deleting other elements.
(D) None of the above

9. Among the following which one is not the right operation on dequeue?
(A) Inserting an element in the middle of a dequeue.
(B) Inserting an element at the front of a dequeue.
(C) Inserting an element at the rear of a dequeue.
(D) None of the above

10. A linear list in which elements can be added or removed at either end but not in the middle is _____.
(A) queue
(B) dequeue
(C) array
(D) tree

11. The post fix notation of $A/B**C+D*E-A*C$ is
(A) $ABC**/DE*+AC*-$
(B) $ABC**D/E*+AC+-$
(C) $ABC**/DE*AC+-$
(D) $ABC**/DE*+AC+-$

Previous Years' Questions

1. An abstract data type (ADT) is **[2005]**
   (A) same as an abstract class.
   (B) a data type that cannot be instantiated.
   (C) a data type for which only the operations defined on it can be used, but none else.
   (D) All of the above

2. An implementation of a queue Q, using two stacks S_1 and S_2, is given below:

```
    void insert (Q, x)    {
      push (S1, x);
    }
    void delete (Q)   {
      if (stack-empty (S2)) then
        if (stack-empty (S1)) then    {
          print ("Q is empty");
          return;
        }
        else while(!(stack-empty(S1)))
        {
            x = pop (S1);
            push(S2, x);
        }
      x = pop (S2);
    }
```

   Let n insert and m (″ n) delete operations be performed in an arbitrary order on an empty queue Q. Let x and y be the number of push and pop operations performed respectively in the process. Which one of the following is true for all m and n? **[2006]**
   (A) $n + m$ ″ $x < 2n$ and $2m$ ″ y $n + m$
   (B) $n + m$ ″ $x < 2n$ and $2m$ ″ y $2n$
   (C) $2m$ ″ $x < 2n$ and $2m$ ″ $y\ n + m$
   (D) $2m$ ″ $x < 2n$ and $2m$ ″ y $2n$

3. The following postfix expression with single digit operands is evaluated using a stack:
   8 2 3 ∧ / 2 3 ∗ + 5 1 ∗ –
   Note that ∧ is the exponentiation operator. The top two elements of the stack after the first ∗ is evaluated are: **[2007]**
   (A) 6 and 1 (B) 5 and 7
   (C) 3 and 2 (D) 1 and 5

4. The following C function takes a single-linked list of integers as a parameter and rearranges the elements of the list. The function is called with the list containing the integers 1, 2, 3, 4, 5, 6, 7 in the given order. What will be the contents of the list after the function completes execution?

```
struct node {
int value;
struct node *next;
};
void rearrange (struct node *list) {
struct node *p, *q;
int temp;
if (!list || !list -> next) return;
p = list; q = list -> next;
while (q) {
temp = p -> value; p -> value = q ->
value;
    q -> value = temp; p = q → next;
    q = p?p -> next : 0;
}
}
```

   [2008]
   (A) 1, 2, 3, 4, 5, 6, 7 (B) 2, 1, 4, 3, 6, 5, 7
   (C) 1, 3, 2, 5, 4, 7, 6 (D) 2, 3, 4, 5, 6, 7, 1

5. Suppose a circular queue of capacity (n – 1) elements is implemented with an array of n elements. Assume that the insertion and deletion operations are carried out using REAR and FRONT as array index variables, respectively. Initially, REAR = FRONT = 0. The conditions to detect queue full and queue empty are **[2012]**
   (A) Full: (REAR + 1) mod n = = FRONT
   Empty: REAR = = FRONT
   (B) Full: (REAR + 1) mod n = = FRONT
   Empty: (FRONT + 1) mod n = = REAR
   (C) Full: REAR = = FRONT
   Empty: (REAR + 1) mod n = = FRONT
   (D) Full: (FRONT + 1) mod n = = REAR
   Empty: REAR = = FRONT

6. Consider the C program below **[2015]**

```
#include <stdio.h>
int *A, stkTop;
int stkFunc (int opcode, int val)
{
  static int size=0, stkTop=0;
  switch (opcode) {
    case -1: size = val; break;
    case 0: if (stkTop < size)
  A[stkTop++] =
        val; break;
    default: if (stkTop) return A[-
    -stkTop];
  }
  return -1;
}
   int main ( )
   {
   int B[20]; A = B; stkTop = -1;
   stkFunc (-1, 10);
   stkFunc (0, 5);
   stkFunc (0, 10);
   printf ("%d\n", stkFunc(1, 0) +
   stkFunc(1, 0));
   }
```

   The value printed by the above program is _____

7. The result of evaluating the postfix expression 10 5 + 60 6/* 8 – is **[2015]**
(A) 284 (B) 213
(C) 142 (D) 71

8. Let Q denote a queue containing sixteen numbers and S be an empty stack.

Head (Q) returns the element at the head of the queue **Q without** removing it from **Q**. Similarly Top(S) returns the element at the top of **S without** removing it from S.

Consider the algorithm given below.

```
while Q is not Empty do
        if S is Empty OR Top(S) ≤ Head (Q)
        then
                x : = Dequeue (Q)
                Push (S, x);
        else
                x : = Pop (S);
                enqueue (Q, x);
        end
end
```

The maximum possible number of iterations of the while loop in the algorithm is ____ . **[2016]**

9. The attributes of three arithmetic operators in some programming language are given below.

Operator	Precedence	Associativity	Arity
+	High	Left	Binary
–	Medium	Right	Binary
*	Low	Left	Binary

The value of the expression

2 – 5 + 1 – 7 * 3 in this language is ______. **[2016]**

10. A circular queue has been implemented using a singly linked list where each node consists of a value and a single pointer pointing to the next node. We maintain exactly two external pointers **FRONT** and **REAR** pointing to the front node and the rear node of the queue, respectively. Which of the following statements is/are CORRECT for such a circular queue, so that insertion and deletion operations can be performed in O (1) time?

I. Next pointer of front node points to the rear node.
II. Next pointer of rear node points to the front node.

[2017]

(A) I only (B) II only
(C) Both I and II (D) Neither I nor II

Answer Keys

Exercises

Practice Problems 1

1. A	**2.** C	**3.** B	**4.** B	**5.** D	**6.** A	**7.** A	**8.** D	**9.** D	**10.** A
11. A	**12.** D	**13.** C	**14.** B	**15.** B	**16.** B				

Practice Problems 2

1. D	**2.** A	**3.** C	**4.** C	**5.** D	**6.** A	**7.** C	**8.** C	**9.** A	**10.** B
11. A									

Previous Years' Questions

1. C	**2.** A	**3.** A	**4.** B	**5.**	**6.** 15	**7.** C	**8.** 256	**9.** 9	**10.** B