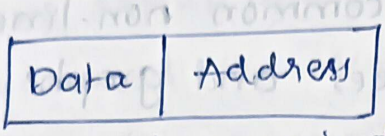
DATA STRUCTURES - UNIT-I Q.B. ANSWERS

SHORT ANSWERS (1 Mark)

1A) Applications of data structures:
- organizing and manipulating data
- storage and retreival of data
- managing data in web applications.

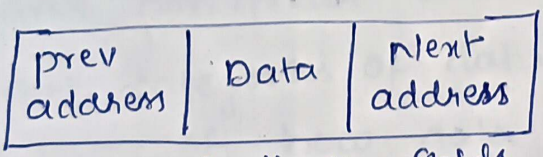
1B) Node structure of single linked list.



It consists of two main fields
1) Data:- This field stores the data or actual value
2) Address:- This field stores the address of next node.

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

Node structure of doubly linked list



It consists of three fields
- prev:- stores address of previous node
- Data:- stores actual value or data
- Next address :- stores next node address

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

2A) **Data structure:** A data structure is a way of organizing and storing data to perform operations efficiently. It provides a means to access and manipulate the data effectively.

2B) Recursion often involves the use of a data structure called the 'call stack'. The call stack is a last-in, first-out (LIFO) structure that keeps track of function calls. Each time a function is called, its information is pushed onto the stack, and when the function completes, it is popped off.

In recursive functions, each recursive call adds a new frame to the call stack. This allows the function to remember its state and return to it after the recursive call completes. The call stack essentially acts as a data structure that facilitates the recursive process.

3A) Applications of stacks:-

- function call management
- Expression evaluation
- Used in backtracking algorithms
- implementing undo in graphic design
- postfix expression evaluation.

3B) Queue operations using arrays typically involve two primary actions:

1) Insertion:- Adding new element to the rear of the queue, to implement you would increase the rear index and place the new element at that position.

2) Deletion:- Removing an element from the front of queue involves accessing the element at the front, incrementing the front index, and returning or using the element.

These basic operations ensure the first-in, first-out (FIFO) behaviour characteristic of a queue.

(3)

4A) Stack overflow:

- Array implementation: occurs when trying to push an element onto a full stack. In an array-based stack, there's a fixed size, and if the stack is full, attempting to push more elements leads to stack overflow.
- Linked list implementation: It's less likely to face overflow in linked lists hence they're dynamically allocate memory. In extreme cases, if system runs out of memory for node creation, a stack overflow can still occur.

Stack underflow:

- Array implementation:- When popping an element from an empty stack, it leads to stack underflow
- linked list implementation:- Underflow occurs when attempting to pop from an empty linked list-based stack. The list must contain atleast one node to perform pop operation.

4B) Linear data structure:

- Elements arranged in sequential orders
- Each element has a unique predecessor and succesor, except first and last elements.
- Examples : arrays, linked lists, queues and stacks

Non-linear data structure:

- Elements are arranged in sequential order.
- Elements may have multiple predecessors and successors, forming a branching or hierarchical structure.
- Examples include trees and graphs.

(4)

5A)

```
algorithm search (key)
{
    /* here start and last are two global
    identifiers to represent start and last nodes.
    These two are initially NULL. Here temp is
    also a node can be used to process data */

    temp = start;
    while (temp != NULL)
    {
        if (temp -> data == key)
        {
            print (element found);
            break;
        }
        temp = temp -> data
    }
    if (temp == NULL)
        print (element not found);
}
```

5B) Array:-

- Memory allocation: contiguous block of memory is allocated
- size : size is declared initially. Hence fixed
- Access time : Direct access to elements using index.
- Insertion/Deletion : inserting or deleting elements can be ineffecient as it may require shifting elements.

Linked list:

- Memory allocation:- Elements are stored in nodes, and each node can be scattered in memory. Dynamic memory allocation is used.
- size :- size can be dynamic, and memory is allocated as needed.
- Access time :- Access time is not constant ; it depends on the position ~~since it involves~~ of the element,

(5) . Insertion / Deletion:- Efficient for insertion and deletion at any position since it involves updating pointers.

6A)
```
algorithm sl_delete (key)
{
  /* here start and last are two identifiers to
     indicate start and last nodes. curr, prev
     indicates the curr node and previous nodes */
  slnode *prev, *curr;
  prev = curr = start;
  while (curr->data != key && curr->next != NULL)
  {
    prev = curr;
    curr = curr->next
  }
  if (curr == start)
  {
    start = start->next;
    free (curr);
  }
}
```

6B)
```
algorithm sl_display ()
{
  /* start and last are two global identifiers indicates
     the first and last nodes */
  temp = start;
  if (temp == NULL)
    print (list is empty);
  else
  { while (temp != NULL)
    {
      print (temp->data)
      temp = temp->next;
    }
  }
}
```

(6)

7A) 1) Order of elements

- Stacks: follows Last in first out order, where the last element added is the first one to be removed
- Queues: follows first in first out order, where the first element added is the first one to be removed

2) Insertion or deletion:

- Stacks: Elements are inserted and deleted from the same end, called top.
- Queues: Elements are inserted at rear and deleted from the front.

3) Access to elements:

- Stacks: Provides access to only the topmost element
- Queues: provides access to both the front and rear elements, but typical usage involves accessing elements from the front.

7B)

```
algorithm rev_display (
{
  /* start and last are two global identifiers to
     indicate first and last node */

     slnode *temp;
     int s[100], i, top=0;
     temp = start;
     while (temp != NULL) {
         s[top++] = temp -> data;
         temp = temp -> next;
       }
     print ("list data in reverse order:");
     for (i = top-1; i >=0 ; i--)
         print (s[i]);

}
```

7

LONG ANSWERS 5 & 10 marks

10) **Singly linked lists:-** A singly linked list is a common or standard type linked list. In this list every node stores the address of next node. If there is no next node it can store NULL value.

*) creation of single linked list:

```
Algorithm create()
{
/* here start and last are two global variables
used to represent starting and ending nodes of
list. These two identifiers contains initially NULL
values. Here temp is also a node it can be used
to processing the data */
int val;slnode *temp;
while (1)
{
    print ("enter node value");
    read (value);
    if (val ==0);
        break;
    else
        {
        temp = (slnode *) mallac (sizeof(slnode*));
        temp -> data = value;
        temp -> next = NULL;
        if (start == NULL)
            {
                start= last = temp;
            }
        else
            {   last -> next = temp;
                last = temp;
            }
    }
}   }
```

(8)

* Inserting elements into single linked list.

```
Algorithm- insert (key)
{
/* start is a global identifier which is key used to
represent starting node of list. Here the initial
values are NULL. key is the value to identify key
node */
slnode * temp, * curr, * prev;
curr = prev = start;
while (curr -> data != key && curr -> next != NULL)
{
    prev = curr;
    curr = curr -> next;
}
temp = (slnode *) malloc (sizeof (slnode));
printf (" enter value of new node ");
scanf ("%d", & temp -> data);
temp -> next = NULL;
if (curr = start)
{
    temp -> next = start;
    start = temp;
}
else
    if (curr == last && curr -> data != key)
    {
        last -> next = temp;
        last = temp;
    }
    else
    {
        prev -> next = temp;
        temp -> next = curr;
    }
}
```

(8)

2c) Implementation of stacks using single linked lists.

To implement a stack using the singly linked list concept, all the singly linked list operations should be performed based on stack operations LIFO (last in first out) and with the help of that knowledge, we are going to implement a stack using a singly linked list.

In the stack implementation, a stack contains a top pointer. which is the "head' of the stack where pushing and popping items ~~happened~~ happens at the head of the list. The first node has a ~~null~~ null in the link field and second node link has the first node address in the link field and so on and the last node address is in the "top" pointer. The Basic stack operations are push, pop and display.

```
Algorithm push (key)
{
/* top is a global identifier represents top of
   stack initiated with NULL value */
  stnode *temp;
  temp = (stnode*) mallac (sizeof (stnode));
    temp -> data = element;
    temp -> next = NULL;
    if (top == NULL)
        top = temp;
    else
        { temp->next = top;
          top = temp;
        }
}
```

```
Algorithm pop ()
{
/* top is a global identifier it's initially a
   NULL pointer */
  if (top == NULL)
      return (-1)      // stack empty
  else {
      temp = top;
      top = top -> next;
      top -> next = NULL;
      ele = temp -> data;
      free (temp);
      return (ele);
  }
}
```

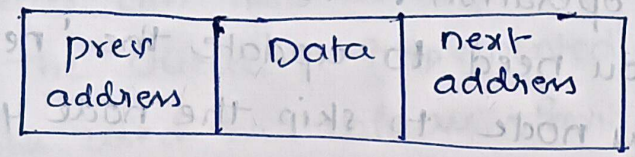
```
Algorithm st-display ()
{
    if (top == NULL)
        print (stack is empty);
    else {
        temp = top;
        while (temp != NULL)
        print (temp -> data);
        temp = temp -> next;
    }
}
```

(11)

3c) A linked list is a linear data structure that stores the collection of data items dynamically. In a linked list the data is represented as nodes. Each node consists of data feeld and address feeld.

Representation of double linked list

A double linked list is a data structure where each node contains data and two pointers: one pointing to previous node one pointing to the next node. It allows in efficient traversal in both directions.



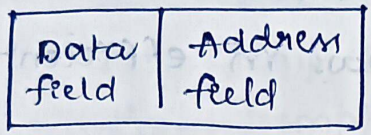
Operations on double linked list:

1. **Deletion operation** in a doubly linked list involves finding the node to delete, updating the 'next' pointer of the previous node to point to the node after the one to be deleted and updating the 'prev' pointer of the next node to point to the node before the one to be deleted.
2. Search operation in a double linked list is similar to that in a singly linked list. You start from the head node and traverse through the list, comparing the idea in each node with the data you are searching for. You can go forward and backward to find the element efficiently.
3. Traversing operation in a doubly linked list can be done in both forward and backward directions. You start at the head or tail and

follow the 'next' or 'prev' pointers respectively, moving from one node to another until you reach the end of the list in the desired direction.

Single linked list Representation:-

A single linked list consists of mainly two fields one 'address of next node' and data part field. The data part field stores the data value in node and the address field stores the address of next node.



Operations on single linked lists:

1. Deletion operation: To delete a node in a singly list list, you need to update the 'next' pointer of the previous node to skip the node you want to delete, essentially bypassing it. This operation can be less efficient when compared to a doubly linked list because you have to traverse the list from the head to find the previous node.

2. Search operation: You start from the head node and traverse through the list, comparing the data in each node with the data you are searching for. If the data matches, you have found the element. This operation can be efficient for searching in a forward direction.

3. Traversing operation: Traversing a single linked list can only be done in one direction, from head to tail. You follow the 'next' pointers from one node to another until you reach the end of the list.

4c) • Queues are the linear data structures that stores the data in continuous memory locations
• It uses the property of first in first out (FIFO).
• Queues are identified with two pointers front and rear.
• Queue can handle multiple data
• We can access both ends.
• They are fast and flexible

*) Queue Representation:

1. Array Representation of Queue:

Queues can also be represented in an array: In this, the Queue is implemented using the Array. Variables used in this are front, rear.

Algorithm to insert elements / enqueue:

```
algorithm insert ()
{
    if (rear == size)
        print (queue is full)
    else
    {
        ele = q [rear];
        rear++;
    }
}
```

Algorithm to delete / dequeue:

```
Algorithm delete ()
{
    if (front == rear)
        print (queue is empty);
    else
    {
```

```
        ele = q[front];
            front++;
          return[ele];
        }
    }
Algorithm display()
  {
      if (front == rear)
          print (queue is empty)
        else
          {
            for (i=front; i<rear; i++)
              print q[i];
          }
  }
```

2. Linked list representation of queue:

Node structure of queue:

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

Algorithm for inserting /enqueue:

```
Algorithm insert()
  {
     qnode * temp;
     temp = (qnode*) malloc (sizeof (qnode));
      temp -> data = ele;
      temp -> next = NULL;
      if (first == NULL)
        { rear = temp;
          front = temp;
        }
      else
        { rear -> next = temp;
          rear = temp;
        }
```

```
    }
}
```

Algorithm to delete / dequeue:

```
algorithm delete_qlist()
{
    if (first == NULL)
        return (-1);
    else
    {   temp = front;
        front = front -> next;
        temp -> next = NULL;
        ele = temp -> data;
        free (temp);
        return (ele);
    }
}
algorithm display ()
{
    if (front == NULL)
    print ("queue is empty");
    else
    {   print (queue data)
        temp = first;
        while (temp != NULL)
        {   print (temp -> data);
            temp = temp -> next;
        }
    }
}
```

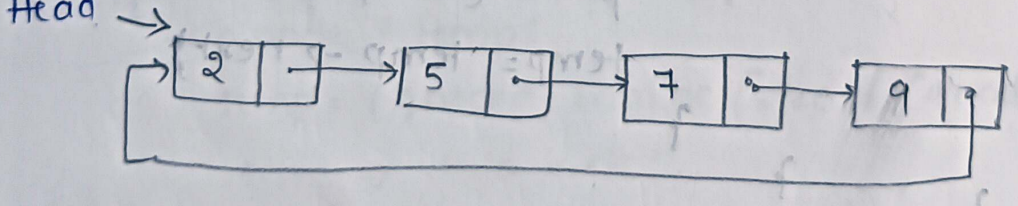
16

5c) The circular linked list is a linked list where all nodes are connected to form a circle. In a circular linked list, the first node and the last node are connected to each other which forms a circle. There is no NULL at the end.

There are generally two types of circular linked lists:

- Circular singly linked list: In a circular singly linked list, the last node of the list contains a pointer to the first node of the list. The circular single linked list has no beginning or end.
- Circular doubly linked list: Circular doubly linked list has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by the previous and next pointer and the last node points to the first node by the next pointer and also the first node points to the last node by the previous pointer.

Representation of single circular linked list:



Here Head node represents the starting node and which is helpful in traversal.

Node representation of a circular linked list:

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

(17) Operations on circular linked list:

We can do operations like:

i) Insertion
ii) Deletion on circular linked lists.
iii) Creation
iv) Display

Algorithms to creat and display a

i) Algorithm_create()

```
{
    // here temp, last, start are two global variables

    clnode *temp;
    int val;
    while (1)
    {
        print ("enter value to be insert");
        scanf("%d", &val);
        if (val==0)
            break;
        else
        {
            temp = (clnode*) malloc (sizeof (clnode));
            temp -> data = val;
            if (start = NULL)
            {
                start = temp;
                last = temp;
                last -> next = temp;
            }
            else
            {
                last -> next = temp;
                temp -> next = start;
                last = temp;
            }
        }
    }
}
```

(18)

Algorithm to display the elements of circular linked list.

```
algorithm display()
{
    clnode *temp;
    temp = start;
    if (temp == NULL)
        print("List is empty");
    else
    {
        print("List values are:");
        do{
            print(temp -> data);
            temp = temp -> next;
        } while (temp != start);
    }
}
```

6c i) =

*) (A+B)*C convert infix to postfix by using stack

To convert an infix expression (A+B)*C to postfix using a stack, you can follow these steps:

1). Initialize an empty stack and an empty output string.

2) Scan the infix expression from left to right

3) for each symbol in the infix expression:

- If it's an operand (A, B, C, etc.), add it to the output string.
- If it's an operator (+), pop operators from the stack and add them to the output until the stack is empty or the top operator has lower precedence. Then push

the current operator onto the stack.

- If it's an opening parenthesis '(', push it onto the stack.
- If it's a closing parenthesis ')', pop operators from the stack and add them to the output untill ~~the~~ an opening parenthesis is encountered. Pop and discard the opening parenthesis.

4) After scanning the entire expression, pop any remaining operators from the stack and add them to the output.

Applying these steps to (A+B)*C;

1. Empty stack, empty output.
2. scan (A+B)*C:
   - '(' - push onto stack
   - 'A' - append to output
   - '+' - push onto stack
   - 'B' - append to output
   - ')' - pop operators from the stack and append to output untill '(' is encountered. Discard '('.
   - '*' - push onto stack.
   - 'C' - append ~~output~~ to output
3. Pop any remaining operators from the stack and append to output.

The postfix expression for (A+B)*C is AB+C.

6C) Algorithm infix_postfix ()

```
{
    Read (infix);
    push (#);
    while ((ch = infix[i++]) != '\0')
    {
        ch = to upper (ch);
        if (ch == '(')
            push (ch);
        else if (is alnum (ch))
            postfix [k++] = ch;
        else
            if (ch == ')')
            {
                while (s[top-1] != '(')
                    postfix [k++] = pop();
                elem = pop(); /* Remove ( */
            }
        else
            { /* operator */
            while (pr(s[top-1]) >= pr(ch))
            {
            printf ("%d[top]");
            postfix [k++] = pop();
            }
        push (ch);
        }
    }
    while (s[top] != '#') /* pop from stack till empty */
    {
        postfix [k++] = pop();
    postfix [--k] = '\0'; /* make postfix as valid string */
    prnt ("infix, postfixExpn");
}
```

(2) ii) 456*+: Evaluate this expression using stack.

To evaluate the postfix expression 456*+, using stacks as follows:

*) ~~Intilia~~

*) Initialize an empty stack

*) Scan the expression from left to right.

*) for each symbol in the expression:

- If it's an operand (like 4, 5, 6), push it onto the stack.
- If it's an operator (* or +), pop the top two operands from the stack, perform the operation and push the result back onto the stack.

*) After scanning the entire expression, the result will be the only element left on the stack.

Applying these steps to 456*+:

1. Empty stack.
2. Scan 456*+:
   - '4' - push onto stack
   - '5' - push onto stack
   - '6' - push onto stack.
   - '*' - pop 5 and 6, multiply, push 30 onto the stack.
   - '+' - pop 4 and 30, add, push 34 onto the stack.
3. The result is 34.

So, the value of the postfix expression 456*+ is 34.

(22)

7 c)

```
#define SIZE 50  /* size of stack */
#include <ctype.h>
char s[SIZE];
int top = 0 ;  /* global declaration */
push (char elem)
    {    /* function for push operation */
         s[top++] = elem;
    }
char pop()
    { /* function for pop operation */
       return (s[--top]);
    }
int pr (char elem)
    {   /* function for precedence */
        switch (elem)
         {
             case '#': return 0;
             case '(' : return 1;
             case '+':
             case '-' : return 2;
             case '*':
             case '/':
                       return 3;
         }
   }
main ()
 { /* main pgrm */
    char infix[50], pofx[50] postfix[50], ch, elem;
    int i=0, k=0;
    printf("\n Read the infix expression ");
    scanf ("%s", infix);
    push ('#');
    while ((ch = infix[i++]) != '\0')
    {    ch = toupper(ch);
         if (ch == '(')
              push (ch);
```

7A)

```
else
    if (isalnum(ch))
        pofx[k++] = ch;
    else
    if (ch == ')')
    {
        while (s[top-1] != '(')
            pofx[k++] = pop();
        elem = pop();   /* Remove ( */
    }
    else
    { /* operator */
        while (pr(s[top-1]) >= pr(ch))
        {
            printf("top opr is %c", s[top]);
            pofx[k++] = pop();
        }
        push(ch);
    }
}
while (s[top] != '#')   /* pop from stack till empty */
{   pofx[k++] = pop();
pofx[--k] = '\0';   /* make pofx as valid string */
printf("\n Given Infix Expn : \t %s \n Postfix
        Expn: \t %s \n", infix, pofx);
}
```

7B)