Concept of Data Structures

1. Define Data structure and algorithm. Why is it important to study?

In computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.

Note: Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, B-trees are particularly well-suited for implementation of databases.

An algorithm is well defined set of rules. Recipe for solving some types of computational problems.
Importance of DSA as follows:

- it provides a means to manage large amounts of data efficiently, such as large databases
- important for all other branches of computer science
- plays a vital role in modern technological innovation
- provides novel “lens” on processes outside of computer science and technology

2. What is ADT? Write down the method of specifying ADT. How do you define Stack and Queue as ADT?

An abstract data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects.

Note:

- ADT is a useful tool for specifying the logical properties of a data type.
- ADT is a mathematical concept that defines the data type.
- ADT defines the values the data type can store and the operations that can be performed.
- Space and Time Complexity do not matter in ADT. Those are implementation issues.
- The definition of ADT is not concerned with implementation details at all. It may not even be possible to implement an ADT in a piece of hardware or using a software.

Specifying an ADT consists of two parts:

- Value Definition: It defines the collection of values for the ADT and consists of two parts:
    a. Definition clause: The keyword ***abstract typedef*** is used to specify the value definition. The definition of a value can take place as follows:
        - *abstract typedef <tp> adt-name* (this defines an ADT having an element of type ‘tp’)
        - *abstract typedef <tp1,tp2,...> adt-name* (this defines an ADT having elements of types ‘tp1’, ‘tp2’, …)
        - *abstract typedef <<tp>> adt-name* (this defines an ADT having sequence of type ‘tp’ with any length)
        - *abstract typedef <<tp,n>> adt-name* (this defines an ADT having a sequence of elements of type ‘tp’ of length ‘n’ where ‘n’ is an integer)
        - *abstract typedef <<,n>> adt-name* (this defines an ADT having an arbitrary sequence of length ‘n’)
    b. Condition clause: The keyword **condition** is used to specify any conditions on the newly

defined type.

- Operator Definition: Each operator is defined as an abstract function with three parts:
    a. Header: It starts with keyword **abstract** to indicate it is an abstract function and it also specifies the name of the function and the data types involved.
    b. Preconditions: The lines starting with keyword **precondition** are the preconditions and they specify any restrictions that must be satisfied before the operation can take place.
    c. Postconditions: They start with keyword **postcondition** and they specify what the operation does.

The ADT specification of a simple rational number is shown below:

```
/*value definition*/
abstract typedef <int, int> RATIONAL;
condition RATIONAL[1] != 0;
/*an adt RATIONAL is defined having two integers, one for numerator and other for denominator.
  the condition specifies that RATIONAL[1], i.e., the denominator may not be zero */

/* operator definition */
abstract RATIONAL makerational(a,b)
int a,b;
precondition b!=0
postcondition makerational[0] == a;
              makerational[1] == b;
/* an operator 'makerational' is defined. it takes two integers 'a' and 'b'. The precondition is that b
cannot be zero. The operator inserts 'a' to makerational[0] and 'b' to makerational[1]. Thus it gives
a rational number 'a/b'. */

/*other operations like add, subtract, multiply, etc can be specified */
```

Stack:
An abstract stack could be defined by three operations:

- push, that inserts some data item onto the structure
- pop, that extracts the last item pushed
- peek(optional), that allows data on top of the structure to be examined without removal.
- empty(optional, that returns 1 if there are no elements, otherwise 0.

The constraint in stack is that element is always pushed or popped from the top of the stack.
ADT specification of stack

```
abstract typedef <<eltype>> STACK (eltype);

abstract empty(s)
STACK(eltype) s;
postcondition empty == (len(s) == 0)        // len(s) returns the length of s

abstract eltype pop(s)
STACK(eltype) s;
precondition   empty(s) == FALSE
postcondition pop == first(s);              //first(s) gives the first element of s
              s == sub(s, 1, len(s) -1)     //sub(s,1,len(s)-1) gives substring of s (1 to len(s)-1)

abstract push(s, elt)
```

```
STACK(eltype) s;
eltype elt;
postcondition s == <elt> + s          //<elt> + s concatenates 'elt' and s

abstract eltype peek(s)
STACK(eltype) s;
precondition   empty(s) == FALSE
postcondition peek == first(s)
```

Queue:
An abstract queue could be defined by two operations:

- enqueue (addition of entities to the rear terminal position)
- dequeue (removal of entities from the front terminal position)
- peek (optional) (returning the value of the front element without dequeuing)
- empty (optional) (returns 1 if queue is empty, else returns 0)

ADT specification of queue

```
abstract typedef <<eltype>> QUEUE (eltype);

abstract empty(q)
QUEUE(eltype) q;
postcondition empty == (len(q) == 0)        // len(q) returns the length of q

abstract eltype remove(s)
QUEUE(eltype) q;
precondition   empty(q) == FALSE
postcondition remove == first(q);           //first(q) gives the first element of s
              q == sub(q, 1, len(q) -1)     //sub(q,1,len(q)-1) gives substring of q (1 to len(q)-1)

abstract insert(q, elt)
QUEUE(eltype) q;
eltype elt;
postcondition q == q + <elt>          //q + <elt> concatenates q and 'elt'

abstract eltype peek(q)
QUEUE(eltype) q;
precondition   empty(q) == FALSE
postcondition peek == first(q)
```

3. Define array as an ADT. Explain the concept of multidimensional array.

An array is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key. An array is stored so that the position of each element can be computed from its index tuple by a mathematical formula. An array is usually bounded from 0 to (ub-1) and sometimes from 1 to ub. The specification of array is as follows:

```
abstract typedef <<eltype, ub>> ARRAY(ub,eltype);
```

```
condition type(ub) == int;

abstract eltype extract(a,i)            /*     written a[i]     */
ARRAY(ub,eltype) a;
int i;
precondition 0 <= i < ub;
postcondition extract == a_i ;

abstract store (a, i, elt)              /*     written a[i] = elt        */
ARRAY (ub, eltype) a;
int i;
eltype elt;
precondition 0 <= i < ub;
postcondition a[i] == elt;
```

Multidimensional Arrays
Arrays can have more than one dimension.
For example, a two dimensional array can be declared by

type array[s1][s2]   / s1,s2 = 1,2,3,... */*

Here 'array' has s1 rows and s2 columns and hence has (s1*s2) elements.
Similarly, a three dimensional array can be declared by

type array[s1][s2][s3]   / s1,s2,s3 = 1,2,3,... */*

Here 'array' has s1 planes, s2 rows, and s3 columns and hence has (s1*s2*s3) elements.
In general, an n-dimensional array can be declared by

*type array[s1][s2][s3]...[sn]   /*s1,s2,s3,...sn = 1,2,3,... */*

The array will have a dimension (s1xs2xs3x...xsn). It will consist(s1*s2*s3*...*sn) elements. Though the geometric analogy breaks down beyond three dimensions, multidimensional arrays are allowed.
Multidimensional arrays are an important data structure. For example, a coordinate of a city may be defined in two-dimensions,i.e., longitude and latitude. Similarly, an array of temperatures may be indexed by latitude, longitude and altitude. Again, the array of temperatures can be modified to store temperatures of different time periods by adding another dimension for time, and so on.
An element can be accessed by

array[i1][i2][i3]...[in]

Here i1,i2,i3,...,in are the index of the element. The location of this element can be written as

*base(array) + esize * [(i1 * r2 * … *rn) + (i2 * r3 * … *rn) + … + (i(n-1) * rn) +in]*

which can be more evaluated more efficiently by using the equivalent formula:

*base(array) + esize * [in + rn * (i(n-1) + r(n-1) * (... + r3 * (i2 + r2 * i1)))]*

4. What is stack? How stack can be implemented?

A stack is a particular kind of abstract data type or collection in which the principal (or only) operations on the collection are the addition of an entity to the collection, known as push and removal of an entity, known as pop. The relation between the push and pop operations is such that the stack is a Last-In-First-Out (LIFO) data structure.

In a LIFO data structure, the last element added to the structure must be the first one to be removed. This is equivalent to the requirement that, considered as a linear data structure, or more abstractly a sequential collection, the push and pop operations occur only at one end of the structure, referred to as the top of the stack. Often a peek or top operation is also implemented, returning the value of the top element without removing it.

A stack may be implemented in two ways:

- using an array: it will have a bounded capacity (it cannot exceed the size of the array and we will need to check the overflow condition) and we use the index of the array to represent the top of the stack.
- using a linked list: it will not have a bounded capacity and we use a pointer of the structure as the top pointer

5. What are the primitive operations of stack? Write down the implementations in C.

Primitive functions of stack as follows:

- push, that inserts some data item onto the stack
- pop, that extracts an item from stack (with the constraint that each pop always returns the most recently pushed item that has not been popped yet)
- peek, that allows data on top of the structure to be examined without removal
- overflow, if stack is full (optional)
- underflow, if stack is empty(optional)

```
/**
 * Implementation of stack using array
 */

#include <stdio.h>

#define N 3

int stack[N];
int top = -1;

void push(){
      int n;

      if(top >= N-1){
            printf("Stack is full\n");
      }else{
            printf("enter a number: ");
            scanf("%d", &n);
            top++;
            stack[top] = n;
      }
}

void pop(){
      if(top<0){
            printf("Stack empty\n");
      }else{
            int x = stack[top];
            top--;
            printf("%d is popped\n",x);
      }
}

void peek(){
      printf("Top Element = %d\n", stack[top]);
      return;
}

void display(){
      int i;

      for(i=0; i<=top; i++)
            printf("%d ", stack[i]);
      printf("\n");
}

int main(){
      int c;

      while(1){
            printf("\n");
            printf("1. Push\n");
            printf("2. Pop\n");
            printf("3. Peek\n");
            printf("4. Display\n");
            printf("5. Exit\n");

            printf("Option: ");
            scanf("%d",&c);

            if(c==1)
                  push();
            else if(c==2)
                  pop();
            else if(c==3)
                  peek();
            else if(c==4)
                  display();
            else if(c==5)
                  return 0;
      }

return 0;
}
```

```
/**
 * Implementation of stack using linked list
 */

#include<stdio.h>
#include<stdlib.h>

typedef struct{
      int data;
      struct node *link;
}node;

node *top=NULL;

void push(){
      int n;
      node *ptr = malloc(sizeof(node));
      printf("Enter the value to Push : ");
      scanf("%d", &n);
      ptr->data=n;
	ptr->link = top;
	top = ptr;
}

void pop(){
	int n;
	node *ptr;
	if(top !=NULL){
		ptr = top;
		n = top->data;
		top = top->link;
		free(ptr);
		printf("%d Popped\n",n);
	}
	else{
		printf("\nStack Empty\n");
	}
}

int main(){
	int choice, flag=0;
	while(!flag){
		printf("1. Push\n2. Pop\n3. Quit\nEnter your choice : ");
		scanf("%d", &choice);
		switch(choice){
			case 1:	push();
					break;
			case 2:	pop();
					break;
			case 3:	flag=1;
		}
	}
return 0;
}
```

6. Write down the algorithm of evaluating postfix operation using stack.

In normal algebra, we use the infix notation like a+b*c. The corresponding postfix notation is abc*+.
The algorithm for evaluating postfix operation is as follows :

- Initialise an empty stack.
- Scan the Postfix string from left to right.
    - If the scanned character is an operand, add it to the stack.
    - If the scanned character is an operator, then pop the top two elements to temporary variable. Now evaluate the operation defined by the operator on the two popped elements and push the result to the stack

  Repeat this step till all the characters are scanned.
- After all characters are scanned, we will have only one element in the stack. Return topStack.

7. How do you convert an expression from infix to postfix using stack. Explain with the help of example.

Algorithm

1. Read an character input

2. Actions to be performed at end of each input

○ Opening parentheses	(2.1) Push into stack and then Go to step (1)
○ Number	(2.2) Print and then Go to step (1)
○ Operator	(2.3) Push into stack and then Go to step (1)
○ Closing parentheses	(2.4) Pop it from the stack (2.4.1) If it is an operator, print it, Go to step (2.4) (2.4.2) If the popped element is an opening parentheses, discard it and go to step (1)
○ New line character	(2.5) STOP

For example, (((8 + 1) - (7 - 4)) / (11 - 9))

Input	Operation	Stack	Output on monitor
(	(2.1) Push operand into stack	(	
(	(2.1) Push operand into stack	((	
(	(2.1) Push operand into stack	(((	
8	(2.2) Print it		8
+	(2.3) Push operator into stack	(((+	8
1	(2.2) Print it		8 1
)	(2.4) Pop from the stack: Since popped element is '+' print it	(((	8 1 +
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	((	8 1 +
-	(2.3) Push operator into stack	((-	
(	(2.1) Push operand into stack	((- (	
7	(2.2) Print it		8 1 + 7
-	(2.3) Push the operator in the stack	((- (-	
4	(2.2) Print it		8 1 + 7 4

)	(2.4) Pop from the stack: Since popped element is '-' print it	((- (	8 1 + 7 4 -
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	((-	
)	(2.4) Pop from the stack: Since popped element is '-' print it	((	8 1 + 7 4 - -
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	(	
/	(2.3) Push the operand into the stack	(/	
(	(2.1) Push into the stack	(/ (	
11	(2.2) Print it		8 1 + 7 4 - - 11
-	(2.3) Push the operand into the stack	(/ (-	
9	(2.2) Print it		8 1 + 7 4 - - 11 9
)	(2.4) Pop from the stack: Since popped element is '-' print it	(/ (	8 1 + 7 4 - - 11 9 -
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	(/	
)	(2.4) Pop from the stack: Since popped element is '/' print it	(	8 1 + 7 4 - - 11 9 - /
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	Stack is empty	
New line character	(2.5) STOP		

8. What is queue? What are the different types of queue?

A queue is a particular kind of abstract data type or collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position, known as enqueue, and removal of entities from the front terminal position, known as dequeue. This makes the queue a First-In-First-Out (FIFO) data structure.

Note:
In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed.
Often a peek or front operation is also implemented, returning the value of the front element without dequeuing it.
A queue is an example of a linear data structure, or more abstractly a sequential collection.

Types of queue are as follows:
- Simple Queue
- Circular Queue
- Priority Queue
- DEQueue (Double Ended Queue)

9. Write down the algorithm of Enqueue and Dequeue in linear and circular queue separately.

Linear Queue

Enqueue
```
1) Check for overflow
        if(rp >= n-1) then
                write "queue is full"
                return
2) Increment rear pointer
        r ← r+1
3) Insert the element
        LQ[rp] ← x
4) Check for front pointer
        if(fp == -1)
                f = 0
5) FINISH
```

Dequeue
```
1) Check for underflow
        if(fp == -1)
                write "queue is underflow"
                return
2) Delete the element
        x ← LQ[fp]
3) Increment the front pointer
        if(rp == fp)
                fp ← rp ← -1
        else
                fp ← fp + 1
4) FINISH
```

Circular Queue

```
Enqueue
1) Check for overflow
        if(rp == fp-1) OR (fp == 0 AND rp == n-1)
                write “queue is full”
                return
2) Increment rear pointer
        if(fp>0 AND rp == n-1)
                rp ← 0
        else
                rp ← r + 1
3) Insert the element
        CQ[rp] ← x
4) Check for front pointer
        if(fp == -1)
                fp ← 0
5) FINISH
```

```
Dequeue

1) Check for Underflow
        if(fp == -1)
                write “queue is full”
                return
2) Delete the element
        x ← CQ[fp]
3) Increment the front pointer
        if(fp == n-1)
                fp ← 0
        else if (fp == rp)
                fp ← rp ← -1
        else
                fp ← fp + 1
4) FINISH
```

10. Write the types of priority queue? Is it always necessary that the order of priority queue be in terms of values? If not give some examples?

The types of priority queue are:

- ascending priority queue: items can be inserted arbitrarily and from which only the smallest element can be removed.
- descending priority queue:items can be inserted arbitrarily and from which only the largest element can be removed.

It is not necessary that the order of priority queue is in terms of values. It can be ordered on any parameters. For example, a stack may be viewed as a descending priority queue whose elements are ordered by time of insertion. The element that was inserted at last has the greatest insertion time value and is the only item that can be retrieved. A queue may similarly be viewed as an ascending priority queue whose elements are ordered by time of insertion.

11. What are the issues for the deletion of items in an array implementation of

priority queue? What could be the possible solutions?

The issues for the deletion of items in an array implementation of priority queue are:

- The element to be deleted may occur at the middle of the array.
- All the elements need to be examined in order to find the element to be deleted.

The possible solutions are:

- A special indicator to indicate an empty element may be used. The insertion operation can also be modified to fill these empty places
- The elements of the array that occur right to the deleted element may be shifted to remove the empty spaces.
- We can also use ordered arrays or order the array before deletion so that the array is not affected by deletion of elements.

12. What is circular queue? How can you implement a circular queue? Explain empty queue and full queue?

A circular queue is a data structure that uses a single, fixed-size buffer as if it were connected end-to-end. If a small change is made to a queue, so that the last node contains a pointer back to the first node instead of the null pointer, the queue is known as priority queue. This structure lends itself easily to buffering data streams.

Like a normal queue, a circular queue may be implemented in two ways:

- using an array
- using a linked list

Empty Queue: Empty queue is a queue having no elements. For array implementation, it generally means that the index ‘top’ has value -1. For linked list implementation, it means that the front and the rear pointers have value NULL.

Full Queue: It is a queue in which no more elements can be added. This case arises generally in array implementation as the size of array is bounded. In linked list implementation, this case can be avoided as the size of the queue becomes dynamic. Nonetheless, the size of queue in linked list implementation can also be restricted by the programmer.

13. What is the condition of overflow and underflow in a queue and Stack? Explain with suitable example.

Stack

Condition for overflow
```
    if(top >= N)
        write “stack overflow”
        return
```

Condition for underflow
```
    if(top < 0)
        write “stack underflow”
        return
```

Queue

```
Check for overflow
        if(rp >= n-1) then
                write "queue is full"

Check for underflow
        if(fp == -1)
                write "queue is underflow"
```

Let us consider a queue ‘Q’ and stack ‘S’ which are both empty. That means for array implementation, the top of S and front(and rear) of Q are equal to -1, and for linked list implementation, top pointer of S and front & rear of Q are NULL. Thus, for array implementation, top of S and front & rear of Q are -1, S and Q are empty and thus we get the condition of underflow. Similarly, for linked list implementation, if top pointer of S and front & rear of Q are NULL, then underflow occurs
The overflow condition doesn’t occur in case of linked list implementation as the memory allocation is dynamic. But for array, when the top of S or rear of Q try to exceed the maximum size of array, i.e., it they are greater than or equal to maximum size of array, then overflow occurs as the array cannot hold more elements

14. What is drawback of using sequential storage to represent stack and queue? What are the possible alternatives and its advantage?

The drawbacks of using sequential storage to represent stack and queue are as follows:

- it will require a contiguous memory and if there is no consecutive free space in memory, memory cannot be allocated and the data cannot be stored.
- inserting or deleting an element may require all of its elements to be shifted
- during sequential storage, the size of the data structure often becomes fixed and the number of data becomes bounded

The possible alternatives of would be use of Link List to store data.
Some of the advantages of Linked List over Sequential memory storage are as follows:

- Does not require consecutive free space in memory to allocate memory (free space in memory can be fully utilized)
- We don’t need to shift any element in the list to insert/delete an element
- Memory for each node can be allocated dynamically whenever required (thus unnecessary memory is not allocated)

15. What is list? Write down about the array implementation of list.

A list is an abstract data type that implements an ordered collection of values, where the same value may occur more than once. It is a collection of homogenous set of elements or objects.

Array implementation of list
The easiest way of implementing list is to keep the elements in an array. The elements of the list can be accessed with an index.
The advantages of array implementation are:

- Random access is possible
- Implementation of list using array is easier

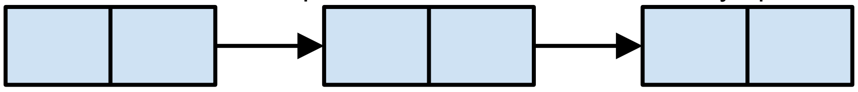
The disadvantages of such implementation are

- Elements are stored in contiguous memory and inserting/deleting an element may require shifting of

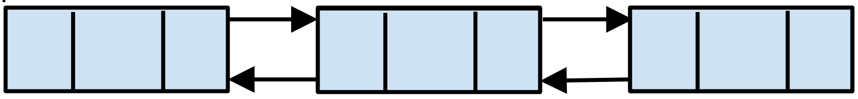
all the elements.
- The size of an array is usually fixed which means that the size of the list becomes bounded and memory of array is reserved even if only a part of it is used.

16. What are different types of list? Explain.

The different types of lists are
- Singly linked lists: Nodes in singly linked lists have only one pointer to reference to the next node. Traversing between nodes is unidirectional. Since they involve only one link, nodes can easily be inserted or deleted. But we require a header node to start every operation.



- Doubly linked lists: Nodes have two pointers. The two consecutive nodes point to each other. Thus, traversing is bidirectional. Header node is not important in doubly linked list as we can reach any node by traversing backward or forward. But they are harder to implement because they involve more operation on nodes for insertion and deletion.



17. Compare singly linked lists with doubly linked lists.

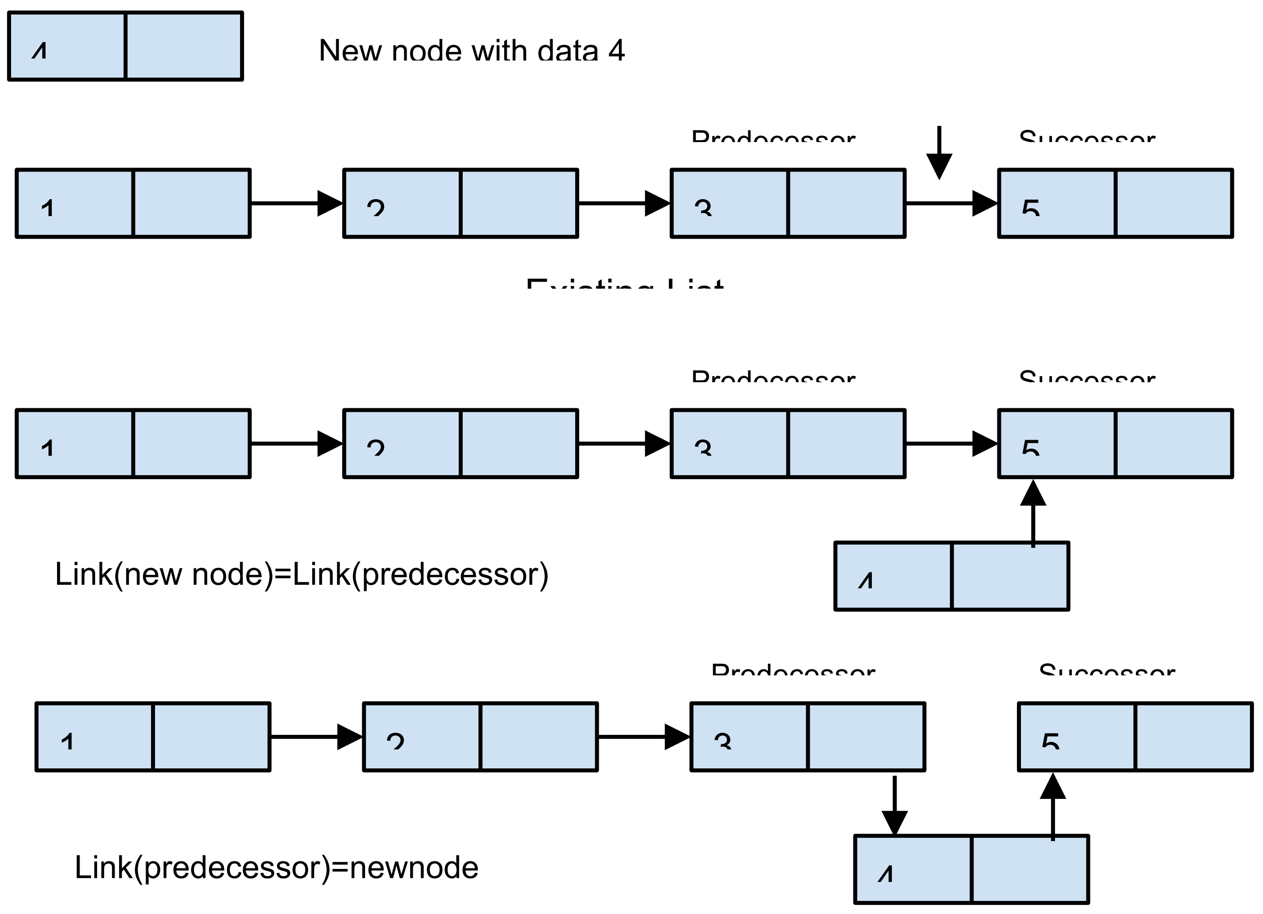
Singly linked lists	Doubly linked lists
Navigation of elements is unidirectional and hence traversing is harder	Navigation of elements is bidirectional and hence traversing is easier
Each node has one pointer and takes lesser space	Each node has two pointers and takes more space
Insertion and deletion of node is fairly easy	Insertion and deletion of nodes is more complex

18. Explain why linked list is called dynamic list.

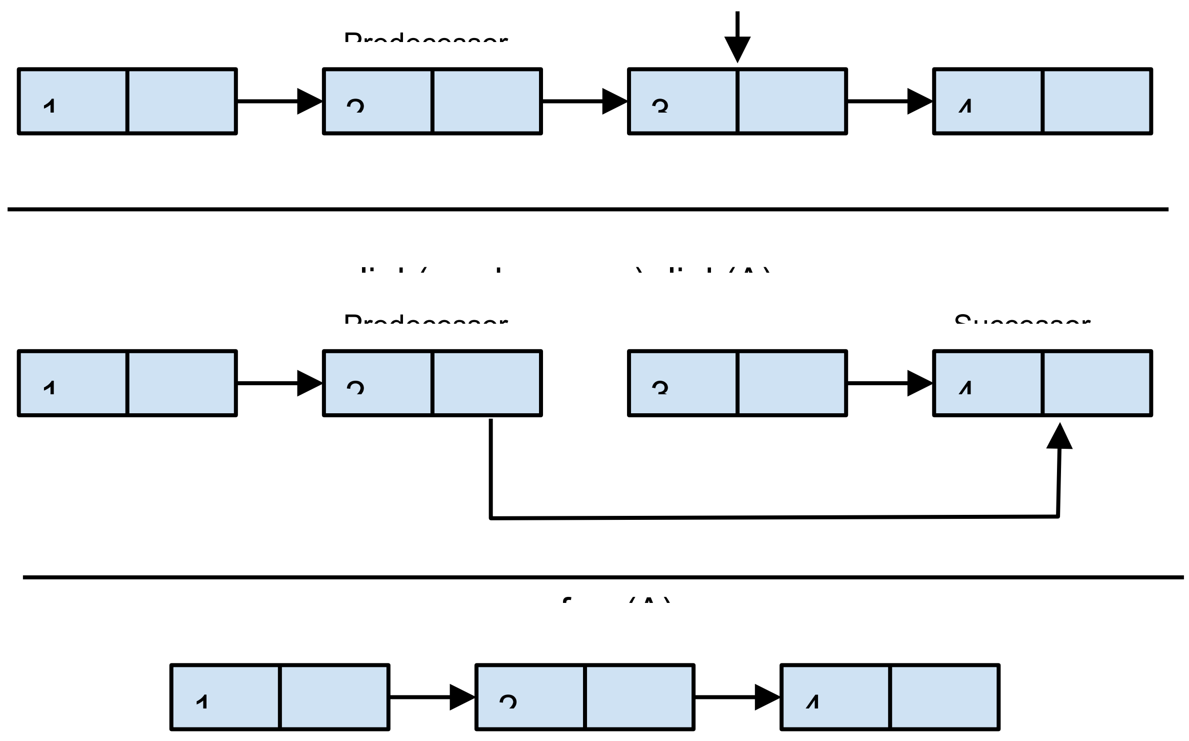
Linked list is called dynamic list because its size is dynamic. Memory is allocated for each element dynamically whenever required and once the element is deleted, the memory is also freed.

19. Write an algorithm for insertion and deletion of nodes form a list. Explain with figure.

- Insertion of Node:
    a. Allocate memory for the new node and move data to the node.
    b. Point the new node to its successor: link(newnode)=link(predecessor)
    c. Point the new node's predecessor to the new node: link(predecessor)=newnode



- Deletion of Node:
    a. If the node to be deleted is ‘A’, point A’s predecessor to A’s successor.
    b. Free memory space occupied by ‘A’



20. Write down the steps involved in inserting and deleting a node in a doubly linked list.

- Insertion of Node:

    a. Allocate memory for the new node and move data to the node.
    b. Point next pointer of the new node to its successor
    c. Point prev pointer of the new node to its predecessor
    d. Point next pointer of the predecessor to the new node
    e. Point prev pointer of the successor to the new node

- Deletion of Node:
    a. Let ‘A’ be the pointer to be deleted.
    b. Point next of A’s predecessor to A’s successor
    c. Point prev of A’s successor to A’s predecessor
    d. Free memory reserved by ‘A’

21. Write the merits and demerits of contiguous list and linked list.

Merits of linked list:

- Memory is allocated dynamically and thus there is no overflow problem unless memory runs out.
- It provides flexibility in terms of addition and deletion of elements.

Demerits of linked list:

- They are not very suitable for random access. We need to traverse many nodes to get to the desired node.
- Extra space is required for the pointer variable. (But in practical purpose, the size of links or pointers is negligible considering the size of the node and number of nodes)

Merits of contiguous list:

- Elements of a contiguous list can be accessed randomly by referring to their position.
- They are easier to use and maintain.

Demerits of contiguous list:

- It requires a continuous memory for allocation.
- Insertion or Deletion of elements of the list may involve shifting of other elements as well.
- Memory is allocated at the beginning. Thus we need to make sure that the size of the allocation is right. Memory will be wasted if large memory is allocated but never used. If small amount of memory is allocated, it may run out.

22. What is the role of Header node in linked list? Explain.

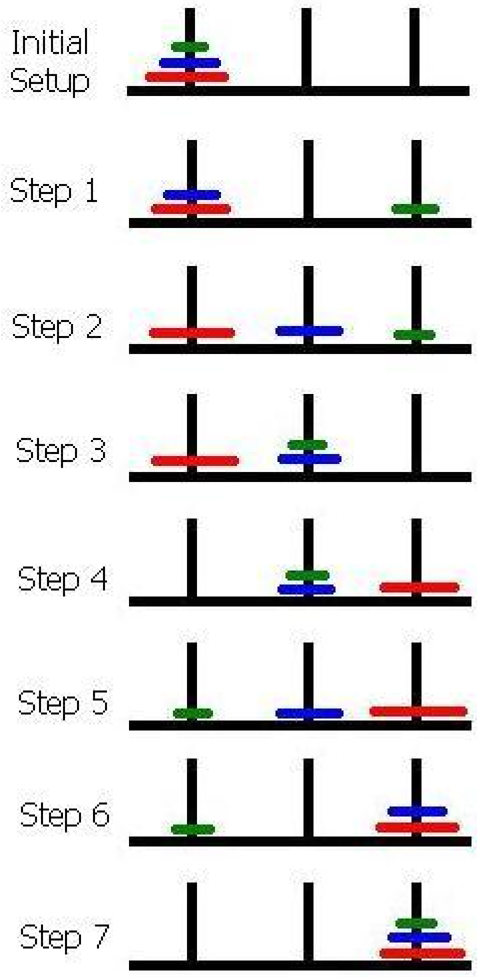
The header node is needed in linked list to keep track of the nodes. It is used to gain the initial access to the entire list and hence is required for every operation like, insertion, deletion, traversing, displaying, etc.
Suppose that we wish to traverse in a list. This can be done by initializing p using header node and repeatedly executing p=p->next until p is NULL. Thus header node provides the starting point.
In circular lists, the header node can have even greater importance. It can also be used to halt the repeated execution of p=p->next.
We need at least one starting point to perform any operation to the linked lists and the header node provides that initial reference. Additional pointers can be added besides the header node for various purposes but a header node is a must.

Recursion

23. What is recursion? Explain with Towers of Hanoi example.

Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to iteration). The approach can be applied to many types of problems, and recursion is one of the central ideas of computer science.

A key to solving Towers of Hanoi puzzle is to recognize that it can be solved by breaking the problem down into a collection of smaller problems and further breaking those problems down into even smaller problems until a solution is reached. For example:
label the pegs A, B, C — these labels may move at different steps
let n be the total number of discs
number the discs from 1 (smallest, topmost) to n (largest, bottommost)
To move n discs from peg A to peg C:
move n−1 discs from A to B. This leaves disc n alone on peg A
move disc n from A to C
move n−1 discs from B to C so they sit on disc n
The above is a recursive algorithm: to carry out steps 1 and 3, apply the same algorithm again for n−1. The entire procedure is a finite number of steps, since at some point the algorithm will be required for n = 1. This step, moving a single disc from peg A to peg B, is trivial. This approach can be given a rigorous mathematical formalism with the theory of dynamic programming, and is often used as an example of recursion when teaching programming.



4. How can recursive algorithm makes program effective? Write merits and demerits of recursion in programming.

Recursion is a powerful problem-solving technique that often produces very clean solutions to even the most complex problems.
Recursive algorithm lowers the length of program. It helps to easily manage codes.

Merits of recursion:

- the recursion is very flexible in data structure like stacks, queues, linked list and quick sort.
- using recursion, the length of the program can be reduced.

Demerits of recursion:

- if base condition is not provided in recursive function, the program will execute out of memory
- it is difficult to design recursive algorithm for a problem
- it is less effective in terms of space and time complexity

25. Write down the recursive definition of Algebraic Expression Multiplication.

The product *a* * *b*, where *a* and *b* are positive integers, may be defined as *a* added to itself *b* times. Thus we have the recursive definition as follows

*a * b = a if b == 1*
*a * b = (a * (b-1) + a) if b > 1*

26. Write a function to compute the nth fibonacci number using recursion. Explain the working mechanism with the help of recursion stack.

C function to compute the nth fibonacci number using recursion:

```
int fib(int n) {
        int x,y;
        if (n <= 1)
                return n;
        x = fib(n-1);
        y = fib(n-2);
        return (x+y);
}
```

ALTERNATIVE

```
int fib(int n) {
        int x,y;
        if (n <= 1)
                return n;
        return (fib(n-1)+fib(n-2));
}
```

The working mechanism (for n=5) can be explained as follows:

Step	Stack for n	Stack for x	Stack for y	Remark
Initial	-	-	-	Initially all the stacks are empty
fib(5) call	**5**	-	-	we have n = 5, so 5 is pushed to Stack for n
fib(4) call	**4**,5	-	-	fib(5) calls fib(4) in line x=fib(n-1)
fib(3) call	**3**,4,5	-	-	fib(4) calls fib(3) in line x=fib(n-1)
fib(2) call	**2**,3,4,5	-	-	fib(3) calls fib(2) in line x=fib(n-1)
fib(1) call	**1**,2,3,4,5	-	-	fib(2) calls fib(1) in line x=fib(n-1)
fib(1) return	**2**,3,4,5	**1**	-	fib(1) returns 1 to x of fib(2)
fib(0) call	**0**,2,3,4,5	**1**	-	fib(2) calls fib(0) in line y=fib(n-2)
fib(0) return	**2**,3,4,5	**1**	**0**	fib(0) returns 0 to y of fib(2)
fib(2) return	**3**,4,5	**1**	-	fib(2) returns x+y=1+0=1 to x of fib(3)
fib(1) call	**1**,3,4,5	**1**	-	fib(3) calls fib(1) in line y=fib(n-2)
fib(1) return	**3**,4,5	**1**	**1**	fib(1) returns 1 to y of fib(3)

fib(3) return	**4**,5	**2**	-	fib(3) returns x+y=1+1=2 to x of fib(4)
fib(2) call	**2**,4,5	**2**	-	fib(4) calls fib(2) in line y=fib(n-2)
fib(1) call	**1**,2,4,5	**2**	-	fib(2) calls fib(1) in line x=fib(n-1)
fib(1) return	**2**,4,5	**1**,2	-	fib(1) returns 1 to x of fib(2)
fib(0) call	**0**,2,4,5	**1**,2	-	fib(2) calls fib(0) in line y=fib(n-2)
fib(0) return	**2**,4,5	**1**,2	**0**	fib(0) returns 0 to y of fib(2)
fib(2) return	**4**,5	**2**	**1**	fib(2) returns x+y=1+0=1 to y of fib(4)
fib(4) return	**5**	**3**	-	fib(4) returns x+y=2+1=3 to x of fib(5)
fib(3) call	**3**,5	**3**	-	fib(5) calls fib(3) in line y=fib(n-2)
fib(2) call	**2**,3,5	**3**	-	fib(3) calls fib(2) in line x=fib(n-1)
fib(1) call	**1**,2,3,5	**3**	-	fib(2) calls fib(1) in line x=fib(n-1)
fib(1) return	**2**,3,5	**1**,3	-	fib(1) returns 1 to x of fib(2)
fib(0) call	**0**,2,3,5	**1**,3	-	fib(2) calls fib(0) in line y=fib(n-2)
fib(0) return	**2**,3,5	**1**,3	**0**	fib(0) returns 1 to y of fib(2)
fib(2) return	**3**,5	**1**,3	-	fib(2) returns x+y=1+0=1 to x of fib(3)
fib(1) call	**1**,3,5	**1**,3	-	fib(3) calls fib(1) in line y=fib(n-2)
fib(1) return	**3**,5	**1**,3	**1**	fib(1) returns 1 to y of fib(3)
fib(3) return	**5**	**3**	**2**	fib(3) returns x+y=1+1=2 to y of fib(5)
fib(5) return	-	-	-	fib(5) returns x+y=3+2=5

Hence, fib(5) = 5.

The execution of the function is as follows:

fib(5) = fib(4) + fib(3)
= (fib(3) + fib(2)) + fib(3)
= ((fib(2) + fib(1)) + fib(2)) + fib(3)
= (((fib(1) + fib(0)) + fib(1)) + fib(2)) + fib(3)
= (((1 + fib(0)) + fib(1)) + fib(2)) + fib(3)
= (((1+0) + fib(1)) +fib(2)) + fib(3)
= ((1 + fib(1)) +fib(2)) + fib(3)
= ((1 + 1) + fib(2)) + fib(3)
= (2 + fib(2)) + fib(3)
= (2 + (fib(1) + fib(0))) + fib(3)
= (2 + (1 + fib(0))) + fib(3)
= (2 + (1 + 0)) + fib(3)
= (2 + 1) + fib(3)
= 3 + fib(3)
= 3 + (fib(2) + fib(1))
= 3 + ((fib(1) + fib(0)) + fib(1))
= 3 + ((1 + fib(0)) + fib(1))
= 3 + ((1 + 0) + fib(1))

```
= 3 + (1 + fib(1))
= 3 + (1 + 1)
= 3 + 2
= 5
```