Computer Science & IT

Programming and Data Structures

Comprehensive Theory

with Solved Examples and Practice Questions

MADE EASY
Publications

MADE EASY Publications

Corporate Office: 44-A/4, Kalu Sarai (Near Hauz Khas Metro Station), New Delhi-110016
E-mail: infomep@madeeasy.in
Contact: 011-45124660, 8860378007
Visit us at: www.madeeasypublications.org

Programming and Data Structures

First Edition: 2015
Second Edition : 2016
Third Edition : 2017
Fourth Edition : 2018
Fifth Edition : 2019
Sixth Edition : 2020

Contents

Programming & Data Structures

■■■■

Programming and Data Structures

Goal of the Subject

Computer Science is not the study of programming. Programming, however, is an important part of what a computer scientist does. Programming is often the way that we create a representation for our solutions. Therefore, this language representation and the process of creating it becomes a fundamental part of the discipline.

A data structure is a specialized format for organizing and storing data. To manage the complexity of problems and the problem-solving process, computer scientists use abstractions to allow them to focus on the "big picture" without getting lost in the details. By creating models of the problem domain, we are able to utilize a better and more efficient problem-solving process. The implementation of an abstract data type, often referred to as a data structure, will require that we provide a physical view of the data using some collection of programming constructs and primitive data types.

General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways. In computer programming, a data structure may be selected or designed to store data for the purpose of working on it with various algorithms.

Introduction

In this book we tried to keep the syllabus of Software Programming and Data structures around the GATE syllabus. Each topic required for GATE is crisply covered with illustrative examples and each chapter is provided with Student Assignment at the end of each chapter so that the students get the thorough revision of the topics that he/she had studied. This subject is carefully divided into seven chapters as described below.

1. **Programming Methodology:** In this chapter we will study about the different segments and their organization, variables and their scope, flow of control in a program, function evaluation types, storage classes, and pointers and finally we discuss the application of pointers.
2. **Arrays:** In this chapter we will study properties and application of arrays, accessing methods for two and three dimensional arrays and finally we discuss the arrays in the form of special matrices.
3. **Stack:** In this chapter we will study the ADT of stack, operations on stack, applications and different types of notations evaluated by stack and finally we discuss the tower of Hanoi (application).
4. **Queue:** In this chapter we will study about the Queue, operations on queue, applications and finally we discuss different types of queues.
5. **Linked List:** In this chapter we will study types and applications of linked list, operations on linked list, priority queue and finally we discuss implementation of stack, queue and priority queue using lists.
6. **Trees:** In this chapter we introduce trees, their applications, types of trees (BST, B-tree, and AVL), different types tree traversals and finally we discuss operations on trees.
7. **Hashing Techniques:** In this chapter we introduce the Hash function, collision resolution techniques and comparisons of different collision techniques.

■■■■

CHAPTER 05

Linked Lists

5.1 Introduction

We have seen representation of linear data structures by using sequential allocation method of storage, as in, arrays. But this is unacceptable in cases like:

(a) Unpredictable storage requirements: The exact amount of data storage required by the program varies with the amount of data being processed. This may not be available at the time we write programs but are to be determined later.

Example: Linked allocations are very beneficial in case of polynomials. When we add two polynomials, and none of their degrees match, the resulting polynomial has the size equal to the sum of the two polynomials to be added. In such cases, we can generate nodes (allocate memory to the data member) whenever required, if we use linked representation (dynamic memory allocation).

(b) Extensive data manipulation takes place: Frequently many operations like insertion, deletion etc., are to be performed on the linked list.

Comparison of Arrays, Vectors and Linked Lists

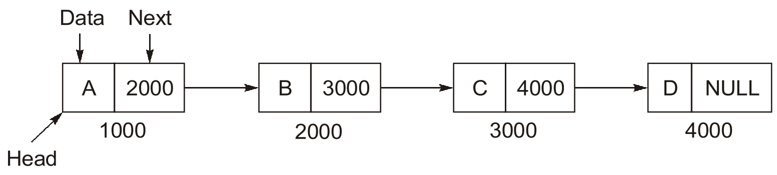
1. Arrays can be useful because:
   - The convenient “[ ]” notation allows immediate access to any element in the array.
   - Arrays can directly hold primitive types, as well as objects.
2. Arrays can be a problem because:
   - The size of the array must be known or at least estimated before the array can be used.
   - Increasing the size of an array can be very time consuming (create another array and copy contents).
   - Arrays can only use a contiguous block of memory. When a new element is inserted into an array, all elements above the new one must be shifted up.
3. Vectors are useful because:
   - Sizing is no longer a problem.
   - They are built-in to the java.util class.
   - They inherit many useful methods.

4. A Vector can be a **problem** because:
   - Arrays are still used (in the background).
   - Every re-sizing of the Vector causes a time lag, since a new array must be created in another block of memory and all the elements copied over.
   - Element insertion is just as time-consuming as with arrays, and possibly even worse if the Vector has to be resized.
   - It can only store Objects, not primitive types.
   - Do not have the handy "[ ]" notation, only methods.

5.2 Linked Lists

A linked list is a linear data structure in which elements are not stored at contiguous memory locations. The elements in linked list are linked using pointers.

A linked list consists of node where each node contains a data field and a reference (link) to the next node in the list.



Next → Contains address where next data is stored.
Null → Denotes end of list.
Head → Points to the start of list (stores address of first node of list).

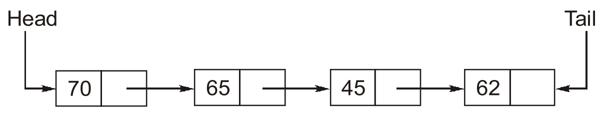
5.3 Uses of Linked lists

1. Link list can store primitive types or Objects.
2. A node can be anywhere in memory. The list does not have to occupy a contiguous memory space.
3. List size is only limited by available memory, and does not have to be declared first.
4. No empty nodes.
5. The adding, insertion or deletion of list elements or nodes can be accomplished with the minimal disruption of neighbouring nodes.

5.4 Singly Linked List or One Way Chain

This is a list, which may consist of an ordered set of elements that may vary in number. Each element in this linked list is called as node. A node in a singly linked list consists of two parts, a information part where the actual data is stored and a link part, which stores the address of the successor (next) node in the list. The order of the elements is maintained by this explicit link between them. The typical node is as shown:

Consider an example where the marks obtained by the students are stored in a linked list as shown in the figure:



In figure, the arrows represent the links. The **data** part of each node consists of the marks obtained by a student and the **next** part is a pointer to the next node. The NULL in the last node indicates that this node is the last node in the list and has no successors at present. In the above example the data part has a single element marks but you can have as many elements as you require, like his name, class etc.

5.4.1 Creation of a Singly Linked List

```
struct node* main( ) //returning a pointer of type struct node
{
   struct node
   {
      int data;
      struct node* next;
   };
   struct node a = {10, NULL} //creating and initializing all nodes
   struct node b = {20, NULL}
   struct node c = {30, NULL}
   struct node d = {40, NULL}
   struct node e = {50, NULL}
   struct node f = {60, NULL}
   a.next = &b;   //linking nodes by storing address of next node
   b.next = &c;
   c.next = &d;
   d.next = &e;
   e.next = &f;
   struct node* head = &a //contains address of first node
   return head;
}
```

Remember

- malloc( ) and calloc( ) are library functions that allocate memory dynamically. It means that memory is allocated during run time in heap.
- malloc( ) allocates memory block of given size and returns a pointer to the beginning of the block. It does not initialize the allocated memory. If we try to access to content of memory block (before initializing) then we will get segmentation fault error (or garbage values).

Syntax: void malloc (size_t size);

- calloc allocates memory and also initializes the allocated memory block to 0. If we try to access the content of these blocks, then we will get 0.

void malloc (size_r num, size_t size);

calloc takes 2 arguments:

- Number of blocks to be allocated
- Size of each block.
- Both malloc( ) and calloc( ) return pointer to allocated memory, if memory is allocated successfully else return NULL.
- malloc( ) is faster than calloc( ).

5.4.2 Operations on Singly Linked List

Structure declaration:

```
typedof struct node
{  int data;
   struct node *link;
};
   # include<stdio.h>
   # include<alloc.h>   /*required for dynamic memory allocation*/
```

Function to add a node at the end of the linked list

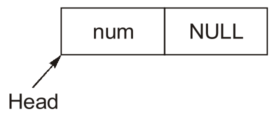
```
oppend(struct node** head, int num)
{
   //create a new node using malloc
   struct node* new_node = (struct node*) malloc(size of struct node))
   //malloc returns address of type void to convert it into struct node type
   //new_node contains address of newly created node.
         new_node →data = num;
         new_node →link = NULL;
      if(*head == NULL)//check if linked list is empty
   {
      *head → new_node; //head contains address of first node
      return* head;
   }
   else //if linked list already contains nodes
   {
      struct node* temp = *head;
      while (temp → link! null) //loop to go to end of list
   {
      temp = temp → link;
   }
   //after coming out of while loop, temp contains address of last node
   temp → link = new node; //temp link now points to new node
   return* head;
   }
}
```

The append( ) function has to deal with two situations:

(a) The node is being added to an empty list.

(b) The node is being added to the end of the linked list.

- In first case, if `(*head == NULL)` gets satisfied, then the newly created node (new_node) is the only node in list and head points to that node.



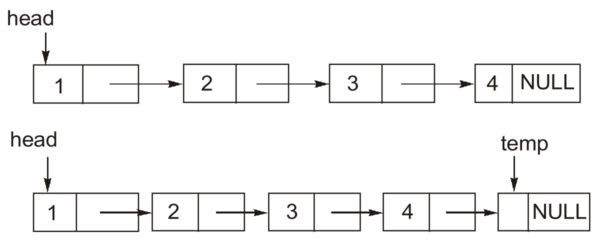
- In second case, if `(*head == NULL)` fail i.e., linked list is not empty. Temp is made to point to the first node of the list using the statement.

$$\underbrace{\text{struct node *temp}}_{\substack{\text{Create a new pointer}\\ \text{temp of type struct node}}} = \text{*head}$$

Then using temp we have traversed through the entire linked list using the statements:

```
while(temp→link !=NULL)
temp=temp→link;
```

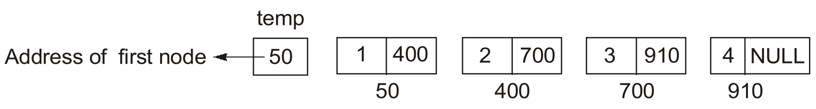
The position of the pointer before and after traversing the linked list is shown below:



Node being added at the end of Singly Linked List

- Each time through the loop the statement `temp=temp→link` makes temp point to the next node in the list. When temp reaches the last node the condition `temp→link != NULL` would fail.
- Once outside the loop, the temp contains address of last node.
- To add new node at last, the link part of last node has to be changed. The link part of last node should contain address of newly created node (new_node). This is done through the statement:

  `temp → link = new_node;`
- There is often a confusion among the beginners as to how the statement `temp=temp→link` makes temp point to the next node in the linked list. Let us understand this with the help of an example. Suppose in a linked list containing 4 nodes temp is pointing to the first node. This is shown in the figure below:



Actual representation of a Singly Linked List in Memory

- Instead of showing the links to the next node the above diagram shows the addresses of the next node in the link part of each node. When we execute the statement `temp=temp→link`, the right hand side yields 50. This address is now stored in temp. As a result, temp starts positioning nodes present at address 50. In effect the statement has shifted temp so that it has started positioning to the next node in the linked list.

5.4.3 Function to Add a Node at the beginning of the Linked List

```
add_at_beginning(struct node **head, int num)
{
   //create a new node using malloc
   struct node* new_node = (struct node*) malloc(size of (struct node));
1. new_node → data = num;
2. new_node → link = *head;
3. *head = new_node;
   return *head;
}
```

Student's Assignments

Q.1 In a linked list

(*i*) Each node contains a pointer to the next node.

(*ii*) An array of pointers point to the links.

(*iii*) Each node containing data and pointer to next.

(*iv*) The links are stored in an array.

Which of the following is correct.

(a) (*i*, *iii*) (b) only (*i*)

(c) only (*iii*) (d) (*ii*, *iv*)

Q.2 Which of the following operation is performed more efficiently by double linked list than by linear linked list

(a) deleting nodes whose location is given

(b) searching an unsorted list for a given item

(c) inserting a node after the node with a given location

(d) traveling the list to process each node

Q.3 Linked lists are not suitable for

(a) Insertion sort

(b) Binary search

(c) Polynomial addition

(d) Polynomial multiplication

Q.4 A list can be initialized to the empty list by which operation

(a) list = 1; (b) list = 0;

(c) list = null; (d) None of these

Q.5 In the given n elements linear linked list to find k^{th} node from the end, how much time it will take?

(a) $O(n)$ (b) $O(n^2)$

(c) $O(n \log n)$ (d) None of these

Q.6 If we want to find last node of a linked list then the correct coding is :

(a) if (temp → link ! = null)
temp = temp → link;

(b) if (temp → data = num
temp = temp → link;

(c) while (temp → link ! = data)
temp = temp → link;

(d) while (temp → link! = null)
temp = temp → link;

Q.7 Consider the following program with a linked list called head:

```
int func(list *head){
  list *a, *b;
  if(head==null||head→next==null)
  return 0;
  a=head;
  b=head→next;
  while (a!=b)
  {
    if(a==null||b==null||b→next==
    null)
    return 0;
    a=a→next;
    b=b→next;
    if(b→next==NULL)
    return 0;
    else
    b=b→next;
  }
  return 1;
}
```

What does the function func( ) do?

(a) Finds if the length of the loop is odd

(b) Finds whether any loop is present in linked list

(c) Finds whether the length of the linked list is even

(d) None of these

Q.8 The most appropriate matching for the following pairs:

List-I

A. m = malloc(5); m = null;

B. free (n); n → value = 5;

C. char * p; *p = 'a';

List-II

1. Using dangling pointer
2. Using uninitialized pointers
3. Lost memory

	A	B	C
(a)	1	3	2
(b)	2	1	3
(c)	3	2	1
(d)	3	1	2

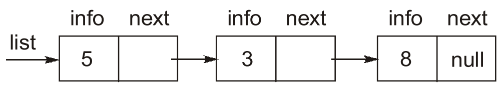
Q.9 Consider the following given code:

```
Void deletenode from LL(struct
LLnode**head)
{
  struct LLnode*temp=*head;
  struct LLnode*current=*head;
  if(*head==NULL)
  {
    printf("list empty");
    return;
  }
  while(current→next!=head)
  {
    current=current→next;
  }
  current→next=*head→next;
  *head=*head→next;
  free(temp);
  return;
}
```

Which of the following is true for the above given code?

(a) The given code is used to delete the last node in a circular linked list.
(b) The given code is used to delete all the nodes in a circular linked list
(c) The given code is used to delete the first node in a circular linked list
(d) The given code is used to delete the middle node in a circular linked list.

Q.10 Consider the following sequence of operation on given link list:



```
P=getnode();
info(P)=6;
next(P)=list;
list=P;
  :
  :
P=list;
list=next(P);
X=info(P);
freenode(P);
```

The above sequence of operation to linked list results into

(a) Updation of a node
(b) Deletion of a node
(c) Linked list remains the same
(d) None of the above

Q.11
```
Void main()
{
  int *mptr, *cptr;
  mptr=(int*)malloc(size of(int));
  printf("%d",*mptr);
  int*cptr=(int*)calloc(size of (int));
  printf("%d",*cptr);
}
```

What is the output of the above program?

(a) grabagl, 0 (b) 0, grabagl
(c) grabagl, grah (d) 0, 0

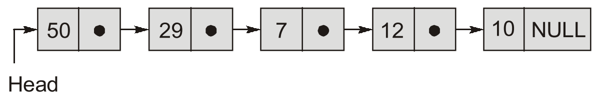
Q.12 The following C function takes single linked list of integers as parameter and rearrange the elements of the list. The function is called with the list containing integers 10, 20, 30, 40, 50, 60, 70 in given order and generate output as 20, 10, 40, 30, 60, 50, 70 after completion. What will be the correct options files so the we get desired output?

```
struct node
{
    int val;
    struct node * next;
}
void Altswap(struct node * list)
{
    struct node *p, *q;
    int temp;
    if (! list || list → next) return;
    p = list; q = list → next;
    while (q)
    {
            W
            X
            Y
            Z
            A
    }
}
```

Which of the following represent the time complexity of above two operations respectively?

(a) $O(n), O(n)$ (b) $O(n), O(1)$
(c) $O(1), O(1)$ (d) $O(1), O(n)$

Q.17 Consider the linked list of integers represented by the following diagram.



Run the following code with the above list of integers.

```
Node * Prev, * nodeToDele;
Prev = Head → next;
nodeToDele = (struct node *) malloc(sizeof
(struct node));
nodeToDele → item = 28;
nodeToDele → next = Prev → next;
Prev → next = nodeToDele;
```

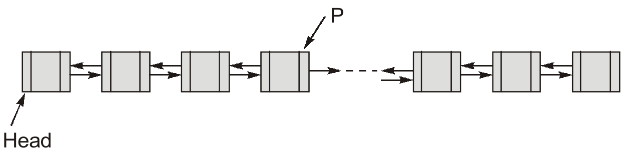
Assume that the following structure is used to create a node in the list.

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

Which of the following is the effect of code?

(a) Element 28 becomes the first elements of the list.
(b) Element 28 becomes the second element of the list.
(c) One element is deleted from the list.
(d) Element 28 becomes the 3rd element of the list.

Q.18 Consider a containing *n* nodes given below :



We want to insert an new node in the given linked list after node P. Which of the following sequence of operation is correct?

(a) 1. new → next = P → next;
2. new = P → next;
3. new → prev = P;
4. (P → next) → prev = next;

(b) 1. new → next = P → next;
2. new → next = P;
3. new → prev → P;
4. (new → next) → prev = new;

(c) 1. new → next = P → next;
2. P → next = new;
3. new → prev = P;
4. (new → next) → prev = new;

(d) 1. new → next → next;
2. P → next = new;
3. new → prev = P;
4. (P → next) → prev = new;

Q.19 In a doubly linked list organization, insertion of a record in end involves modification of ______ for existing list.

(a) one pointer (b) two pointer
(c) multiple pointer (d) no pointer

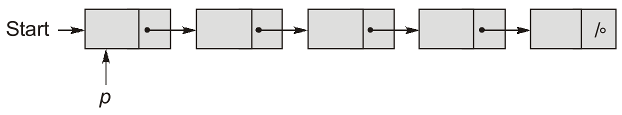
Q.20 Consider the following code for single linked list:

```
Struct void Modified (Struct node ** head)
{
    Struct node *X = *head;
    Struct node *Y;
    Struct node *Z = NULL;
    while (X != NULL)
    {
    Y = X → next;
        X → next = Z;
        Z = X;
        X = Y;
    }
    *head = Z;
}
```

If head is a pointer to a pointer to the first node of the list and it is passed to the 'Modified' function then find the list after executing the function.

(a) It adds a new node at the first
(b) It adds a new node at the last
(c) It keeps the list as same
(d) It reverses the list

Q.24 Let '*p*' be a pointer of NODE (str data, NODE * next) as shown in the figure in a singly linked list.



Consider '*q*' to be another pointer of same data type as '*p*'.

$q = p \rightarrow$ next

$p \rightarrow$ next $\rightarrow$ next $= q \rightarrow$ next $\rightarrow$ next

$p \rightarrow$ next $\rightarrow$ next $= q \rightarrow$ next $\rightarrow$ next

After performing above operations, what will be remaining number of nodes accessible from start pointer.

(a) 5 (b) 3
(c) 4 (d) 2

Q.25 Consider the following C program execute on a singly linked list numbered from 1 to *n* containing atleast 2 nodes:

```
struct Listnode
{
    int data;
    struct Listnode *next;
} ;
void fun (struct Listnode *head)
{
    if (head == NULL || head → next ==NULL)
    return;
    struct Listnode * tmp = head → next;
    head → next = tmp → next;
    free (tmp);
    fun (head → next);
}
```

Which of the following represents the output of above function 'fun' ?

(a) It reverses the every 2 adjacent nodes linked list
(b) Every odd number nodes of given linked list will be deleted
(c) Every even number nodes of given linked list will be deleted
(d) It reverses the linked list and delete alternate nodes

Q.26 The following C function takes a singly linked list as input argument. It prints its elements from the end using with the help of some other data structure. Some part of the code is left blank.

```
typdef struct node
{
    int value;
    struct node * next;
}
Node;
void printlist (node * head)
{
    if (! head) return;
    ________
}
```

Choose the correct alternative to replace the blank line.

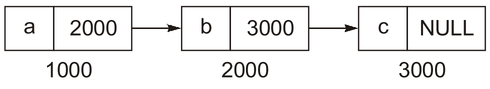
(a) printf ("%d", head → data);
    printlist (head → next);
(b) while (node! = Null)
    { Node = Node → next
    printf("%d", node → data);
    }
(c) printlist (head → next);
    printf("%d", head → data);
(d) None of these

Answer Key:

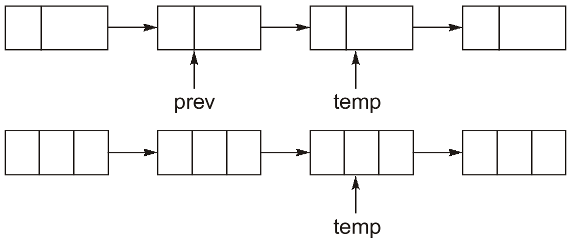
1. (a)	**2.** (a)	**3.** (b)	**4.** (c)	**5.** (a)
6. (d)	**7.** (b)	**8.** (d)	**9.** (c)	**10.** (c)
11. (a)	**12.** (d)	**13.** (c)	**14.** (b)	**15.** (b)
16. (b)	**17.** (d)	**18.** (c)	**19.** (a)	**20.** (d)
21. (c)	**22.** (c)	**23.** (4)	**24.** (b)	**25.** (c)
26. (c)				

Student's Assignments Explanations

1. (a)
In a linked list, a node consists of 2 parts: data and next pointer which stores address of next node.



2. (a)
To delete a node whose location is given because before we remove the node (say of locations) from the linked list, we need to store the address of next node (i.e., 4th node) into the link part of 2nd node.
If this deletion is performed using singly linked list, we need two pointers, prev (to store address of second node) and temp (to store address of third node). While in doubly linked list, this deletion can be performed using only one pointer.



3. (b)
Binary search using linked list takes $O(n)$ time because finding middle takes $O(n)$ time as we have to till last node.

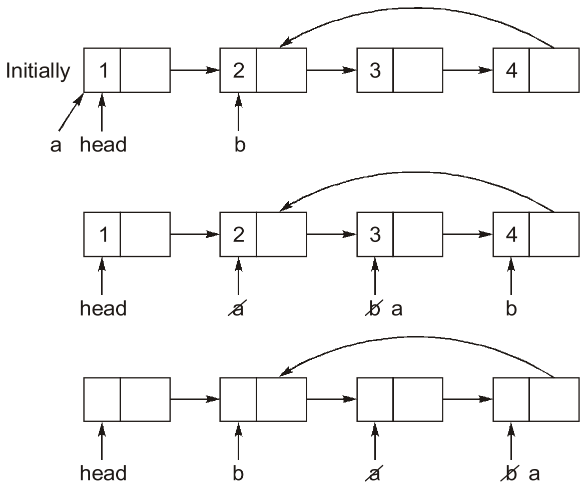
5. (a)
To find kth node from the node

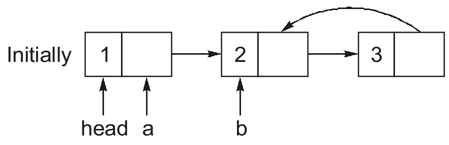
```
struct node {
   int data;
   struct node *next;
};
struct node* getkNode(strcut node*
head int k)
{
   int n=0;
   struct node *temp=head;
   //count total nodes in linked list
   while (temp)
   {
      temp=temp→next;
      n++;
   }
   //if total nodes more than k
   if(n>=k)
   {
   //return(n-k+1)th node from
   beginning
  temp=head;
   for (int i=n, i<n-k+1, i++)
   {
      temp=temp→next;
   }
  }
  return temp;
}
```

6. (d)

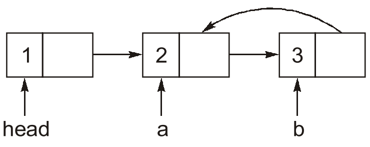
```
while(temp→link!=null)
{
   temp=temp→link;
}
```

7. (b)





After while loop runs once:



if(b → next == Null) - false
b = b → next