PES University
Department of Computer Science and Engineering

UE19CS202- Data Structures and its Applications(4-0-0-4-4)

UNIT - 1 : INTRODUCTION, STATIC AND DYNAMIC MEMORY ALLOCATION, LINKED LIST AND RELATED CASE STUDY

Note : Pointers, Arrays, Structures to be revised for clear understanding.

Unit 1: Overview

Static and Dynamic Memory Allocation, Singly Linked List.**Linked List:** Doubly Linked List, Circular Linked List – Single and Double, Multilist : Introduction to sparse matrix (structure). **Application:** Case Study -**Text Editor , Assembler**- Creation of a Symbol Table.

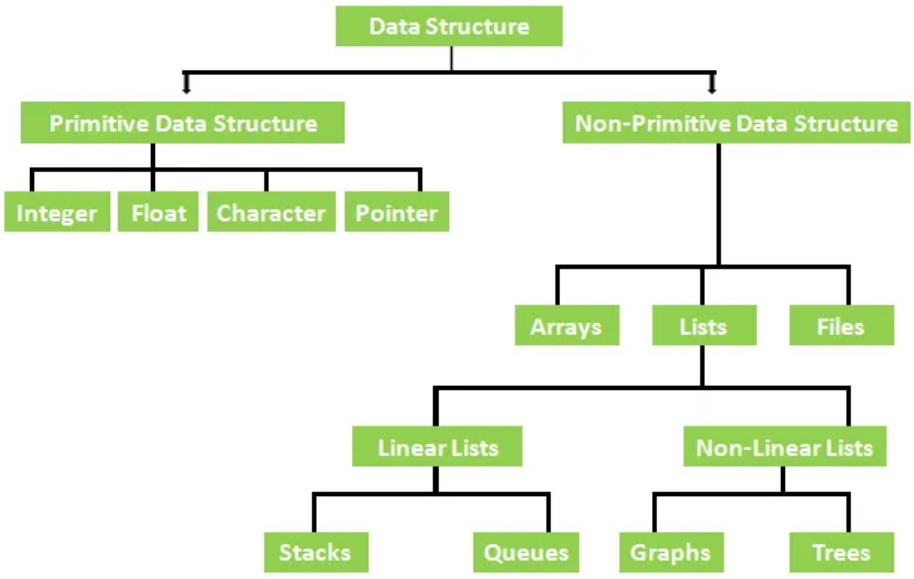
Contents :

Introduction :

- Data Structures is a way of organizing ,managing and storing the data that provides a way for a collection of different data values and the relationship among them.
- Choice of the data structure begins with the choice of Abstract Data Type

- Example : Arrays used to store list of elements having the same type, Structures are used to store the list of elements having different data types
- Other data Structures includes - Linked, Stacks, Queues, Trees, Graphs which will be discussed in detail throughout this course.
- Each of these data structures have a special way of organizing so that users can choose the data structure based on the requirement.
- In other words Data Structure is an arrangement of data in computer's memory in such a way that it could make the data quickly available to the processor for required calculations
- Data Structures should be seen as a logical perspective should address the 2 primary concerns:
    - How will the data be stored?
    - What operations will be performed on it?
- Since data structure is a way of organizing the data, so the functional definition of a data structure should be independent of its implementation.
- The functional definition of a data structure is known as ADT (Abstract Data Type) which is independent of implementation.
- The way in which the data is organized affects the performance of a program for different tasks.
- Programmers decide which data structures are to be used based on the nature of the data and the processes that need to be performed on that data.

Types of Data Structures



- Data Structures are classified into 2 types:
    - Primitive Data Structures

        - All primitive data types are primitive data structures
        - THey follow the machine instructions
    - Non Primitive Data Structures
        - Are built using primitive data types.
- **Linear Data Structures :** It is a type of data structure where the data elements are accessed in a sequential manner. However the elements can be stored in any order. Elements can be traversed in a single run. **Examples** - Arrays, Lists, Stacks
- **Non Linear Data Structures** : It is a type of structure where there is no physical adjacency between the elements, Elements can be accessed in a non-sequential manner. **Examples** - Trees, Graphs.

Data Types Vs Data Structures

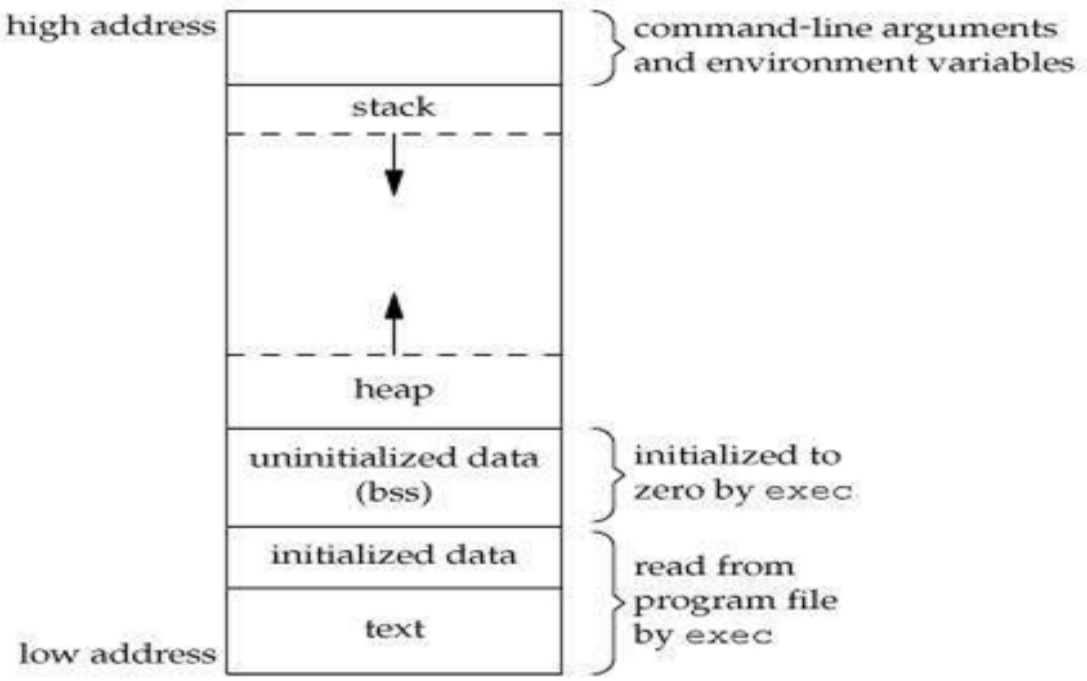
Data Type	**Data Structure**
Data type is a source to hold the variable	Collection of different kinds of data
Implementation - Abstract	Implementation - Concrete
Can hold data values	Can hold within a single object
Values can be directly assigned to types	Data assigned to operations through operations like - push, pop etc..
Example - int, float, char	**Example - Stack, queue, trees**

Abstract Data Type:

- It is a logical description of how the data and the operations are viewed without knowing how they will be implemented.
- Concerned only with what data is representing and not with how it will eventually be constructed.
- Provides level of abstraction and encapsulation around the data.
- The idea is that by encapsulating the details of the implementation,data is getting hidden from user view. This is called information hiding.
- 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.

Memory Management of C Programming

When a C program is executed, it's the executable image loaded into RAM in an organized manner.



Consider the figure above: Memory is divided into various segments as described below:

Text Segment :

- Text segment contains machine code of the compiled program.
- Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on.
- The text segment of an executable object file is often read-only segment that prevents a program from being accidentally modified

Initialized Data Segment:

- Stores global, static, constant, and external variables that are initialized beforehand.
- Data segment is not read-only, since the values of the variables can be altered at run time
- #include<stdio.h>
  Char s[]= “PES”;
  Char c[]=”University”;
  Int main()
  {
  Static int i = 11;
  }

Uninitialized Data Segment:

- Data in this segment is initialized to arithmetic 0 before the program starts executing.
- Uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to 0 or do not have explicit initialization in source code.
- #include<stdio.h>
  Char c;
  Int main()
  {
  Int i;
  Static int j;
  }

Heap:

- Heap is the segment where *dynamic memory allocation* usually takes place.
- When some more memory needs to be allocated using malloc and calloc function, heap grows upward.
- The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

Stack:

- Stack segment is used to store all local variables and is used for passing arguments to the functions along with the return address of the instruction which is to be executed after the function call is over.
- Local variables have a scope to the block which they are defined in, they are created when control enters into the block.
- Activation Records of Function calls are created in the stack.

Memory Management : Static and Dynamic Memory Allocation

- Memory allocation in programming is very important for storing values when you assign them to variables.
- The allocation is done either before or at the time of program execution.

Static Memory Allocation

- In static memory allocation, memory is allocated while the program is being compiled : Compile Time
- Each static or global variable defines one block of space, of a fixed size. The space is allocated once, when your program is executed(part of the exec operation), and is never freed.
- Memory size will be fixed and cannot be changed or modified.
- Memory is allocated in the stack..
- Example : Arrays : Declare static array to stored integer data, when we declare array of 500, required memory = 500*2 = 1000 bytes on 32 bit operating system and required memory = 500*4 = 2000 on a 64 bit operating system. 1000 (on Windows) and 2000 (on Linux) has reserved for the above declared array and we could not use that memory, even if that array contains only one integer or no data. Suppose if an array contains 500 integers, we have deleted 499 still we could not use that memory for other purposes.

Advantages of Static Allocation

- Allocation speed is faster.
- No extra algorithm needed.
- No fragmentation problem required to achieve allocation task.

Disadvantages of Static Allocation

- No memory reusability.
- Less efficient allocation.
- Memory once allocated cannot be reallocated.

Dynamic Memory Allocation

- In dynamic memory allocation, memory is allocated during the execution of the program - Run Time.
- Memory size can be modified while executing the program.
- Dynamic memory allocation manage an area of process Virtual memory Area called Heap
- Example : Linked List.
- ‘C’ offers 4 Dynamic memory allocation functions defined in stdlib.h- malloc(), calloc(), realloc(), free().

1. **malloc() - Memory Allocation**
    - Allocate a single large block of memory with the specified size.
    - It returns a pointer of type void which can be cast into a pointer of any form.

- It initializes each block with default garbage value.
- Syntax: ptr = (type*)malloc(sizeof(bytesize));
- Example : ptr=(int*)malloc(100*sizeof(int));
- In the above example, int occupies 4 bytes in memory, so the total memory occupied by ptr is 100*4 = 400 bytes. I.e a large memory block of size 400 bytes will be dynamically allocated to ptr.

2. calloc() - Contiguous allocation
    - Allocate the specified number of blocks of memory of the specified type.
    - It initializes each block with a default value '0'.
    - Syntax:ptr=(type*)calloc(n, sizeof(elementype));
    - Example : ptr = (float*) calloc(25, sizeof(float));
    - This allocated 25 continuous memory blocks of size of float(4 bytes).

3. free()
    - Deallocate the memory.
    - The memory allocated using functions malloc() and calloc() is not deallocated on their own.
    - Hence the free() method is used, whenever the dynamic memory allocation takes place.
    - It helps to reduce wastage of memory by freeing it.
    - Syntax : free(ptr);
    - Note : C does not support automatic garbage collection like java.

4. realloc()- Reallocation
    - Change the memory allocation of a previously allocated memory.
    - If the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory.
    - Re-allocation of memory maintains the already present value and new blocks will be initialized with default garbage value.
    - Syntax : realloc(ptr,newsize)
    - Ptr is now reallocated with a new size.

Advantages of Dynamic Memory Allocation

1. Can create additional storage whenever required
2. Can delete allocated storage whenever required.
3. Allocates memory during run time
4. Data structure can grow and shrink whenever needed.

Disadvantages of Dynamic Memory Allocation

1. Accessing heap is costly
2. Requires more time as memory is allocated during runtime.
3. Creates problems like - segmentation fault, dangling pointer issue, memory leak

LINKED LIST

- Linear Data Structure where each element is a separate object.
- Each element is called a node comprising 2 fields - data field and link field which is a reference to the next node.
- Last node has the 'null' reference.
- The entry point into a linked list is called the **head** of the list.
- Head is not a separate node, but it implicitly refers to the first node.
- If the list is empty then the head has a null reference.
- Linked list is a dynamic data structure, size of the node is not fixed, it can grow and shrink

Difference Between Arrays and Linked List

1. **Array** - a collection of similar type data elements
   Linked list -a collection of unordered linked elements known as nodes.

2. **Array**- traversal through indexes
   linked list - traversal through the head until we reach the node.

3. **Array** - Elements are stored in contiguous address space
   Linked List - Elements are at random address spaces

4. **Array** - Access is faster
   Linked List - Access is slower

5. **Array**- Insertion and Deletion of an element is not that efficient
   LInked List - Insertion and Deletion of an element is efficient

6. **Array** - Fixed Size.
   Linked List - Dynamic Size.

7. **Array** - memory assigned during compile time/ static allocation
   Linked List - Memory assigned during runtime / dynamic allocation

Types of Linked List

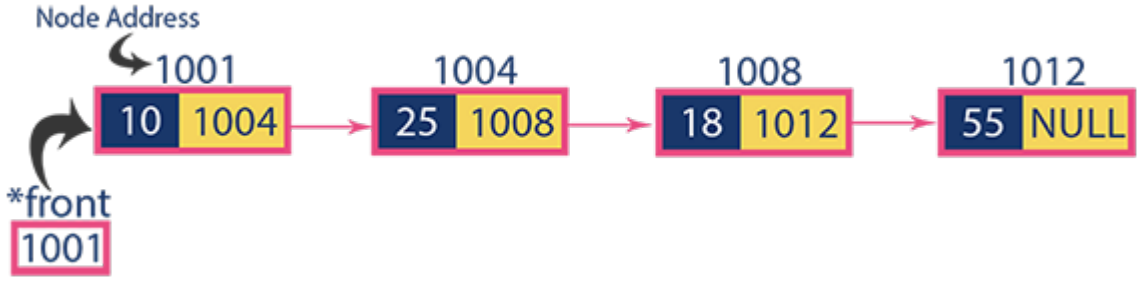
1. Singly Linked List
2. Doubly Linked List
3. Circular List

Singly Linked List

- A singly linked list is a type of linked list that is unidirectional, that is, it can be traversed in only one direction from head to the last node
- Each element in a singly linked list is called a Node.
- A Node contains the data , and a pointer to the next node which helps in maintaining the list structure
- Node structure is defined as:

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

- Linked List is pictorially represented as shown below.



In the above figure, a list contains data elements as 10,25,18,55 and 1004, 1008, 1012, are the address of the next consecutive nodes. First node has a data element as 3 and the address of the first node is stored in the next consecutive node.

Note : Head is only a reference to the first node.

- Head points to the first node of the list and it helps traverse, access other nodes in the list.
- Last node points to NULL and helps to determine the end of the list.

LIST as Abstract Data Type

- List as ADT holds the collection of nodes, which can be accessed only in sequential order.
- Nodes are connected to the next node and/or with the previous one, this gives the linked effect.
- If nodes are connected with only the next pointer the list is called **Singly Linked List - A -> B-> C-> D**
- and it is connected by the next and previous the list is called **Doubly Linked List - A <-> B<-> C<-> D.**

Note : Conventions Used for All Algorithms

- p = pointer variable of type struct node that points to the first node(head).
- temp = new node of type struct node for creating linked list.
- q = iterator of type struct node for traversing till end of list
- i = counter variable of int type, increments as we traverse every node in the list (used for position insertion and deletion to Check the position is correct).

ADT (Interface) - OPERATION PERFORMED ON A SINGLY LINKED LIST

1. Insertion
2. Deletion

Insertion: Insertion operation in a Singly Linked List can be performed in 3 ways:

1. Inserting At Beginning of the list :
    a. void insertbeginning(struct node *temp, int data, struct node *next);

2. Inserting At End of the list :
    a. void insertend(struct node *temp, int data, struct node *next);

3. Inserting At Specific location in the list :
    a. void insertposition(struct node *temp, int data, struct node *next);

Before any operation is performed, the following must be done:

Step 1 - Define a Node structure with two members data and next

Step 2 - Define a Node pointer 'head' and set it to NULL.

Step 3 - Implement the main method by displaying the operations menu and make suitable function calls in the main method to perform user selected operations

1. INSERT AT THE BEGINNING OF THE LIST

Step 1 - Create a new node **temp,** allocate memory dynamically and assign with a value.

Step 2 - Check whether list is Empty **(*p == NULL)**

Step 3 - If it is Empty then, set **temp→next = NULL and *p = temp.**

Step 4 - If it is Not Empty then, set **temp→next = *p and *p = temp**

1. INSERT AT THE END OF THE LIST

Step 1 to 3 // same as front insertion

Step 4 - If it is Not Empty then, define a node pointer q and initialize with head.

Step 5 - Keep moving the q to its next node until it reaches to the last node in the list (until **q → next = NULL).**

Step 6 - Set **q → next = temp.**

2. INSERT AT A SPECIFIC POSITION IN THE LIST

Step 1 - Create a new node temp with given value.

Step 2 - Check whether list is Empty(*p == NULL)

Step 3 - If it is Empty then, set temp→ next = NULL and *p = head.

Step 4 - If it is Not Empty then, define a node pointer temp and initialize with head.

3 cases to be considered :

- Insert at position 1//Front insertion
- Insert at position n//shown below
- Insert in between//shown below

Step 5 - Keep moving the q to its next node until it reaches to the node after which we want to insert the temp (until q → data is equal to location, here location is the node value after which we want to insert the newNode).

Step 6 - Every time check whether q is reached to the last node or not. If it is reached to the last node then display **'Insertion not possible!!!'** and terminate the function. Otherwise move the q to the next node.

Step 7 - Finally, Set 'newNode → next = temp → next' and 'temp → next = newNode'

ADT - Deletion: Deletion operation in a Singly Linked List can be performed in 3 ways

1. Deleting from Beginning of the list :
    a. Node * Deletefront(struct node *temp);

2. Deleting from End of the list :
    a. Node * Deletednd(struct node *temp);

3. Deleting fromSpecific location in the list :
    a. Node * Deletepos(struct node *temp);

1 DELETING FROM THE BEGINNING OF THE LIST

Step 1 - Check whether list is Empty (***p == NULL**)

Step 2 - If it is Empty then, **return 0** and terminate

Step 3 - If it is Not Empty then, define a Node pointer **'q' and initialize it with p.**

Step 4 - Check whether list is having only one node **(q→ next == NULL)**

Step 5 - If it is TRUE then set **p = NULL and delete q** //Empty list Condition

Step 6 - If it is FALSE then set **p = q → next, and delete q.**

2. DELETING FROM THE END OF THE LIST

Step 1 - Check whether list is **Empty** (**p** == **NULL**)

Step 2 - If it is **Empty** then, return and terminate the function.

Step 3 - If it is **Not Empty** then, define two Node pointers **'prev'** and '**temp'** and initialize '**prev**' with **p.**

Step 4 - Check whether list has only one Node (**prev → next** == **NULL**)

Step 5 - If it is **TRUE**. Then, set **p** = **NULL** and delete **prev**. And terminate the function. //List is empty

Step 6 - If it is **FALSE**. Then, set '**temp = prev** ' and move **prev** to its next node. Repeat the same until it reaches the last node in the list. (until **prev → next** == **NULL**)

Step 7 - Finally, Set **temp→ next** = **NULL** and delete **temp1**

3. DELETION AT A SPECIFIC POSITION

Step 1 - Check whether list is Empty **(p == NULL)**

Step 2 - If it is Empty then, return and terminate

Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize **'temp1'=p;.**

Step 4 - Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set **'temp2 = temp1'** before moving the **'temp1' to its next node.**

Step 5 - If it is reached to the last node then display **"Deletion not possible!!!'. / or return 0 and terminate.**

Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7 - If list has only one node and that is the node to be deleted, then set **p= NULL** and **delete temp1 (free(temp1)).**

Step 8 - If the list contains multiple nodes, then check whether temp1 is the first node in the list **(temp1 == head).**

Step 9 - If temp1 is the first node then move the p to the next node **(p = p→ next)** and delete temp1.

Step 10 - If temp1 is not the first node then check whether it is the last node in the list **(temp1 → next == NULL).**

Step 11 - If temp1 is the last node then set **temp2 → next = NULL** and **delete temp1 (free(temp1)).**

Step 12 - If temp1 is not the first node and not the last node then set **temp2 → next = temp1 → next and delete temp1 (free(temp1)).**

DISPLAY THE CONTENTS OF THE LIST

Step 1 - Check whether list is Empty **(p == NULL)**

Step 2 - If it is Empty then, **return 0 and terminate**

Step 3 - If it is Not Empty then, define a Node pointer **'q' and initialize it with a head.**

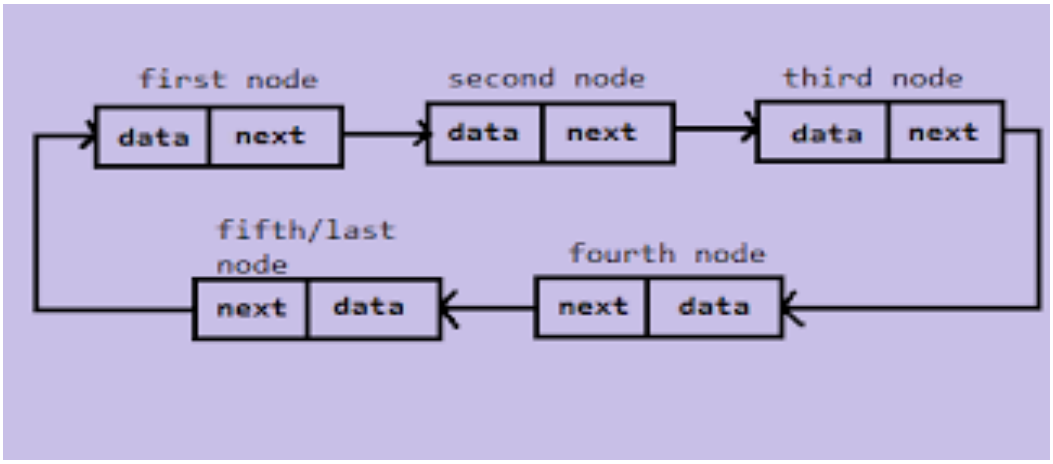
// Traverse using while or for to reach end of list

Step 4 - Keep displaying **q → data** with an arrow (--->) until q reaches to the last node

Step 5 - Finally display **q→ data** with an arrow pointing to NULL (q → data ---> NULL).

CIRCULAR LIST

- A type of linked list where all nodes are connected linearly in the form of a circle.
- It can be implemented using a singly linked list or doubly linked list.
- There is no null node unlike the singly linked list.
- The only difference between singly linked list and circular list is that, in a circular list the last node will point to the head rather than pointing to null.
- Pictorial representation of a circular list is as shown below:



In the above figure, the linked list contains five nodes. It can be observed that the last node(fifth node) points to the first node. Or in other words, the fifth node contains the address of the first node.

ADT of Circular List - BASIC OPERATIONS THAT CAN BE PERFORMED ON A CIRCULAR LIST

1. Insert
2. Delete
3. Display

INSERTION: 3 types of insertion operations on a singly linked list

1. Insertion at the beginning of the list
    a. Node* insertbeginning(struct node *temp, int data);

2. Insertion at the end of the list
    a. Node* endbeginning(struct node *temp, int data);

3. Insertion at a specific position
    a. Node* position_insert(struct node *temp, int data);

1. **INSERTION AT THE BEGINNING OF THE CIRCULAR LIST**

Step 1 - Create a new node as temp with given value.

Step 2 - Check whether list is Empty (p == NULL)

Step 3 - If it is Empty then, set p = temp and temp→next = p .

Step 4 - If it is Not Empty then, define a Node pointer 'q' and initialize with 'p''.

Step 5 - Keep moving the 'q' to its next node until it reaches the last node (until 'q → next == p').

Step 6 - Set temp→ next =p', 'p=temp' and 'q → next = p'

2. **INSERTION AT THE END OF THE CIRCULAR LIST**

Step 1 - Create a newNode with given value.

Step 2 - Check whether list is Empty (p== NULL).

Step 3 - If it is Empty then, set p = temp and temp → next = p.

Step 4 - If it is Not Empty then, define a node pointer q and initialize with head(q=p) .

Step 5 - Keep moving the q to its next node until it reaches to the last node in the list (until q → next ==p).//instead q->next =NULL

Step 6 - Set q → next = temp and temp→ next = p.

Deletion : Deletion operation can be performed in 3 ways

1. Delete from the beginning of the list
    a. Node * Deletefront(struct node *temp);

2. Delete from the end of the lis

a. Node * Deletefront(struct node *temp);

3. Delete from a specific position

   Node * Deletefront(struct node *temp);

1. DELETION AT THE BEGINNING OF THE LIST

Step 1 - Check whether list is Empty p == NULL)

Step 2 - If it is Empty then, display 'List is Empty!!! Or return 0 and terminate the function.

Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize both 'temp1' and 'temp2' with p.(temp1=p, temp2=p;)

Step 4 - Check whether list is having only one node (temp1 → next == p)

Step 5 - If it is TRUE then set p = NULL and delete temp1 (Setting Empty list conditions)

Step 6 - If it is FALSE move the temp1 until it reaches to the last node. (until

temp1 → next == p)

Step 7 - Then set p = temp2 → next, temp1 → next = p and delete temp2.

2. DELETION AT THE END OF THE LIST

Step 1 - Check whether list is **Empty** (**p** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty' or return 0 and** and terminate t

Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **p**.

Step 4 - Check whether list has only one Node (**temp1 → next** == **p**)

Step 5 - If it is **TRUE**. Then, set p = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)

Step 6 - If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches the last node in the list. (until **temp1** → **next** == **p**)

Step 7 - Set **temp2** → **next** = **p** and delete **temp1**

DOUBLY LINKED LIST

A Doubly linked list is a type of linked list in which traversal and access takes place in both the directions.

Node in a doubly linked list contains 3 fields:

1. Data Field
2. Left Link - which holds the address of the preceding node.
3. Right Link - which holds the address of the succeeding node.

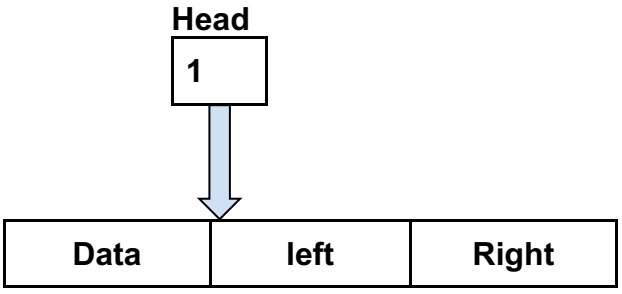
Dnode Structure definition of a Doubly Linked List is as follows

```
Struct Dnode
{
        Int data;
        Struct Dnode *prev;
        Struct Dnode *next;
}
```

Memory Representation of a doubly linked list

A doubly linked list consumes more space for every node.

But it's easy to manipulate the elements of the list since the list points both directions.



13	-1	4
15	1	6
19	4	8
57	6	-1

- First element of the list that is i.e. 13 stored at address 1.
- The head pointer points to the starting address 1.
- Since this is the first element being added to the list therefore the **prev** of the list **contains** null.
- The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.
- The list can be traversed in this way until it reaches the last Node pointing to NULL or -1.

ADT - BASIC OPERATIONS PERFORMED ON DOUBLY LINKED LIST

- Insertion.
- Deletion.

INSERTION : Insertion can be done in 3 ways

1. Insertion at the beginning of list
    a. Struct node *frontinsert(Struct node*temp, int value);

2. Insertion at the end of the list
    a. Struct node *frontend(Struct node*temp, int value);

3. Insertion at a specific position

a. Struct node *frontpos(Struct node*temp, int value);

1. INSERTION AT THE BEGINNING OF THE LIST

Step 1 - Create a new node as **temp** , allocate memory dynamically and assign with value and set **temp → prev and temp->next as NULL.**

Step 2 - Check whether list is Empty **(p == NULL)**

Step 3 - If it is Empty then, assign NULL to **temp→ next and temp to head.**

Step 4 - If it is not Empty then, assign p to **temp → next and temp**.

2. INSERTION AT THE END OF THE DOUBLY LINKED LIST

Steps 1 to 3 : same as above

Step 4 - If it is not Empty, then, define a node pointer q and initialize with p.

Step 5 - Keep moving the q to its next node until it reaches to the last node in the list (until tq → next is equal to NULL).

Step 6 - temp to q→ next and temp to temp → previous

3. INSERTION AT A SPECIFIC POSITION OF THE DOUBLY LINKED LIST

Step 1 - Create a new node as **temp** , allocate memory dynamically and assign with a value.

Step 2 - Check whether list is Empty **(p == NULL)**

Step 3 - If it is Empty then, assign **NULL to both temp → previous & temp →next and set temp to p.**

Step 4 - If it is not Empty then, define two node pointers **q & temp2** and initialize **q with p.**

Step 5 - Keep moving the **q** to its next node until it reaches to the node after which we want to insert the temp (until **q → data is equal to location,** here location is the node value **after** which we want to insert the temp).

Step 6 - Each time keep checking whether **q** is reached to the last node. If it is reached to the last node then display **Insertion not possible and terminate the function**. Otherwise move the **q to the next node.**

Step 7 - Assign q → next to temp2, newNode to temp1 → next, q to temp → previous, temp2 to temp → next and temp to temp2 → previous

//Note : Can avoid usage of temp2 by making **q->previous->next=p and continue accordingly so on…**

DELETION OPERATION PERFORMED ON A DOUBLY LINKED LIST

Deletion of a node from a Doubly Linked List can be performed in 3 ways:

1. Deletion from the front of the list
    a. Struct node *deletefront(struct node*temp, struct node *prev, struct node *next);

2. Deletion from the end of the list
    a. Struct node *deleteend(struct node*temp, struct node *prev, struct node *next);

3. Deletion from a specific position
    a. Struct node *deletefront(struct node*temp, struct node *prev, struct node *next, int pos)

1. DELETION AT THE FRONT OF THE LIST

Step 1 - Check whether list is Empty **(p == NULL)**

Step 2 - If it is Empty then, display '**List is Empty**' and return 0 / terminate the function.

Step 3 - If it is not Empty then, define a Node pointer **'q'** and initialize it with **p**

Step 4 - Check whether list is having only one node **(q → previous is equal to q → next)**

Step 5 - If it is TRUE, then set **p** to NULL and delete q **(free(q))**

Step 6 - If it is FALSE, then assign **q → next to p**, p→ previous to head and **delete q. free(q).**

2. DELETION AT THE END OF THE LIST

Step 1 - Check whether list is Empty **(p == NULL)**

Step 2 - If it is Empty, then display **'List is Empty'** and return 0 / terminate the function.

Step 3 - If it is not Empty then, define a Node pointer **'q'** and initialize it with a p

Step 4 - Check whether list has only one Node (**q→ previous and q→ next = NULL**)

Step 5 - If it is TRUE, then assign **p to NULL and delete q.** And terminate from the function // empty list

Step 6 - If it is FALSE, then keep moving temp until it reaches to the last node in the list. (until **q → next = NULL**)

Step 7 - Assign temp → previous → next to null and delete q. **(free q).**

3. DELETION AT A SPECIFIC POSITION

Step 1 - Check whether list is Empty **(p == NULL)**

Step 2 - If it is Empty then, display **'List is Empty'** or return 0 and terminate the function

Step 3 - If it is not Empty, then define a Node pointer **'q'** and initialize it with p

Step 4 - Keep moving the temp until it reaches to the exact node to be deleted or to the last node.

Step 5 - If it is reached to the **last node**, then display **'Not Found'** and terminate the function.

Step 6 - If it is reached to the same node which is to be deleted, then check whether list is having only one node or not

Step 7 - If list has only one node and that is the node which is to be deleted then set **p to NULL and delete q (free(q)).//Only 1 node)**

Step 8 - If the list contains more than 1 node, then check whether temp is the first node in the list **(q == head)**.

Step 9 - If q is the first node, then move the head to the next node (q = q → next), set p of previous to NULL **(p → previous = NULL)** and delete f**ree(q)** - (Front deletion).

Step 10 - If temp is not the first node, then check whether it is the last node in the list **(q→ next == NULL).**

Step 11 - If temp is the last node then set q of previous to next to NULL **(q → previous → next = NULL)** and delete q **(free(q))**.//End deletion

Step 12 - If temp is not the first node and not the last node, then set q of previous of next to q of next **(q → previous → next = q → next),** q of next of previous to q of previous **(q → next → previous = q → previous)** and **delete q (free(q)).**//Somewhere in between.

MULTI LIST

A Multilist is a variation of Doubly Linked which is described as follows:

- each node has just 2 pointers
- the pointers are exact inverses of each other

In other words, In a Multi-linked list each node can have any number of pointers to other nodes, and there may or may not be inverses for each pointer.

Structure definitation of a multilist is as follows:

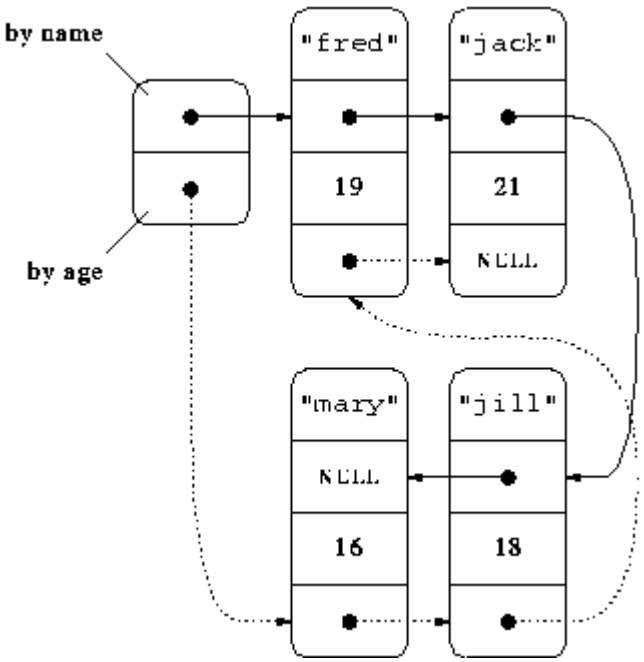
```
struct col_node {
        int col;
        int data;
        struct col_node *next_col;
};
struct row_node {
        int row;
        struct col_node *next_col;
        struct row_node *next_row;
};
```

Example 1: Multiorder of One set of Elements

The standard use of multi-linked lists is to organize a collection of elements in two different ways. For example, suppose my elements include the name of a person and his/her age.

E.g. (FRED, 19) (MARY,16) (JACK,21) (JILL,18)

The elements are ordered alphabetically and also by age. @ pointers are required : NEXT - alphabetically and NEXT - Age. List header will have 2 pointers one based on name, one based on age.



Example 2 : Sparse Matrix

- A sparse matrix is a matrix of numbers, as in mathematics, in which almost all the entries are zero.
- These arise frequently in engineering applications. the use of a normal Pascal array to store a sparse matrix is extremely wasteful of space - in an NxN sparse matrix typically only about N elements are non-zero.

Case Study : Design of Text Editor using SLL/DLL

Text editors are software programs that enable the user to create and edit text files. Notepad, Wordpad are some of the common editors used on Windows OS and vi, emacs, Jed, pico are the editors on UNIX OS.

Typical Features

- **Find and replace** – Text editors provide extensive facilities for searching and replacing text, either on groups of files or interactively. Advanced editors can use regular expressions to search and edit text or code.
- **Cut, copy, and paste** – most text editors provide methods to duplicate and move text within the file, or between files.

- Ability to handle UTF-8 encoded text.
- **Text formatting** – Text editors often provide basic visual formatting features like line wrap, auto-indentation, bullet list formatting using ASCII characters, comment formatting, syntax highlighting and so on. These are typically only for display and do not insert formatting codes into the file itself.
- **Undo and redo** – As with word processors, text editors provide a way to undo and redo the last edit, or more. Often—especially with older text editors—there is only one level of edit history remembered and successively issuing the undo command will only "toggle" the last change. Modern or more complex editors usually provide a multiple-level history such that issuing the undo command repeatedly will revert the document to successively older edits. A separate redo command will cycle the edits "forward" toward the most recent changes. The number of changes remembered depends upon the editor and is often configurable by the user.

Develop a miniature text editing program which will allow a few simple commands, and is, therefore, quite primitive in comparison with a modern text editor or word processor.

Specifications:

Our text editor will allow us to read a file into memory, where we shall say that it is stored in a buffer. We shall consider each line of text to be a string, and the buffer will be a list of these lines. We shall then devise editing commands that will do **list operations** on the lines in the buffer and will do **string operations** on the characters in a single line.

Since, at any moment, the user either may be typing characters to be inserted into a line or may be giving commands, a text editor should always be written to be as forgiving of invalid input as possible, recognizing illegal commands, and asking for confirmation before taking any drastic action like deleting the entire buffer.

Here is a list of commands to be included in the text editor. Each command is given by typing the letter shown in response to the prompt '?'. The command letter may be typed in either uppercase or lowercase.

'**R**': Read the text file, whose name was given in the command line, into the buffer. Any previous contents of the buffer are lost. At the conclusion, the current line will be the first line of the file.

‘**W**’: Write the contents of the buffer to the text file whose name was given in the command line. Neither the current line nor the buffer is changed.

‘**I**’: Insert a single line typed in by the user at the current line number. The prompt ‘I:’ requests the new line.

‘**D**’: Delete the current line and move to the next line.

‘**F**’: Find the first line, starting with the current line that contains a target string that will be requested from the user.

‘**L**’: Show the length in characters of the current line and the length in lines of the buffer.

‘**C**’: Change the string requested from the user to a replacement text, also requested from the user, working within the current line only.

‘**Q**’: Quit the editor; terminate immediately.

‘**H**’: Print out help messages explaining all the commands. The program will also accept ‘?’ as an alternative to ‘H’.

‘**N**’: Next line: advance one line through the buffer.

‘**P**’: Previous line: back up one line in the buffer.

‘**B**’: Beginning: go to the first line of the buffer.

‘**E**’: End: go to the last line of the buffer.

‘**G**’: Go to a user-specified line number in the buffer.

‘**S**’: Substitute a line typed in by the user for the current line. The function should ask for the line number to be changed, print out the line for verification, and then request the new line.

‘**V**’: View the entire contents of the buffer, print out to the terminal.

Case Study: Design of a Symbol Table in an Assembler

In computer science, a symbol table is a data structure used by a language translator such as an assembler, compiler or interpreter, where each identifier (a.k.a. symbol) in a program's source code is associated with information relating to its declaration or

appearance in the source. In other words, the entries of a symbol table store the information related to the entry's corresponding symbol.

The minimum information contained in a symbol table used by a translator includes the symbol's name, and its location or address.

The organization of the symbol table is the key to fast assembly. Even when working on a small program, the assembler may use the symbol table hundreds of times and, consequently, an efficient implementation of the table can cut the assembly time significantly even for short programs.

The symbol table is a dynamic structure. It starts empty and should support two operations, **insertion** and **search**. In a two-pass assembler, insertions are done only in the first pass and searches, only in the second. In a one-pass assembler, both insertions and searches occur in the single pass. The symbol table does not have to support deletions, and this fact affects the choice of data structure for implementing the table. A symbol table can be implemented in many different ways but the following methods are almost always used:

- A linear array
- A sorted array with binary search
- Buckets with linked lists
- A binary search tree
- A hash table

A Linear Array

The symbols are stored in the first N consecutive entries of an array, and a new symbol is inserted into the table by storing it in the first available entry (entry N + 1) of the array. The variable N is initially set to zero, and it always points to the last entry in the array. An insertion is done by:
Testing to make sure that N < lim (the symbol table is not full). Incrementing N by 1. Inserting the name, value (location) , and any other fields, using N as an index.

The insertion takes fixed time, independent of the number of symbols in the table. To search, the array of names is scanned entry by entry. The number of steps involved varies from a minimum of 1 to a maximum of N. Every search for a non-existent symbol involves N steps, thus a program with many undefined symbols will be slow to assemble because the average search time will be high. Assuming a program with only a few undefined symbols, the average search time is N/2. In a two-pass assembler, insertions are only done in the first pass so, at the end of that pass, N is fixed. All searches in the second pass are performed in a fixed table. In a one-pass assembler, N grows during the pass, and thus each search takes an average of N/2 steps, but the values of N are different.

Advantages: Fast insertion. Simple operations.
Disadvantages: Slow search, specially for large values of N. Fixed size

A Sorted Array

The same as a linear array, but the array (actually two arrays or more, for the name, value (location), and any other attributes) is sorted, by name, after the first pass is completed. This, of course, can only be done in a two-pass assembler. To find a symbol in such a table, binary search is used, which takes an average of $\log_2 N$ steps. The difference between N and $\log_2 N$ is small when N is small but, for large values of N, the difference can get large enough to justify the additional time spent on sorting the table.

Advantages: Fast insertion and fast search.
Disadvantages: The sort takes time, which makes this method useful only for a large number of symbols (at least a few hundred).

Buckets with Linked Lists

An array of 26 entries is declared, to serve as the start of the buckets. Each entry points to a bucket that is a linked list of all those symbols that start with the same letter. Thus all the symbols that start with a ‘C’ are linked together in a list that can be reached by following the pointer in the third entry of the array. Initially all the buckets are empty (all pointers in the array are null). As symbols are inserted, each bucket is kept sorted by symbol name. Notice that there is no need to actually sort the buckets. The buckets are kept in sorted order by carefully inserting each new symbol into its proper place in the bucket. When a new symbol is presented, to be inserted in a bucket, the bucket is first located by using the first character in the symbol’s name (one step). The symbol is then compared to the first symbol in the bucket (the symbol names are compared). If the new symbol is less (in lexicographic order) than the first, the new one becomes the first in the bucket. Otherwise, the new symbol is compared to the second symbol in the bucket, and so on. Assuming an even distribution of names over the alphabet, each bucket contains an average of N/26 symbols, and the average insertion time is thus 1+(N/26)/2=1+ N/52. For a typical program with a few hundred symbols, the average insertion requires just a few steps.

A search is done by first locating the bucket (one step), and then performing the same comparisons as in the insertion process above. The average search thus also takes 1 + N/52 steps.

Such a symbol table has a variable size. More nodes can be allocated and added to the buckets, and the table can, in principle, use the entire available memory.

Advantages: Fast operations. Flexible table size.

Disadvantages: Although the number of steps is small, each step involves the use of a pointer and is therefore slower than a step in the previous methods (that use arrays). Also, some programmers always tend to assign names that start with an A. In such a case all the symbols will go into the first bucket, and the table will behave essentially as a linear array.

Such an implementation is recommended only if the assembler is designed to assemble large programs, and the operating system makes it convenient to allocate storage for list nodes.

A Binary Search Tree

This is a general data structure used not just for symbol tables, and is quite efficient. It can be used by either a one pass or two pass assembler with the same efficiency.

The table starts as an empty binary tree, and the first symbol inserted into the table becomes the root of the tree. Every subsequent symbol is inserted into the table by (lexicographically) comparing it with the root. If the new symbol is less than the root, the program moves to the left son of the root and compares the new symbol with that son. If the new symbol is greater than the root, the program moves to the right son of the root and compares as above. If the new symbol turns out to be equal to any of the existing tree nodes, then it is a doubly-defined symbol. Otherwise, the comparisons continue until a node is reached that does not have a son. The new symbol becomes the (left or right) son of that node.

Example: Assuming that the following symbols are defined, in this order, in a program.

BGH, J12, MED, CC, ON, TOM, A345, ZIP, QUE, PETS

Symbol BGH becomes the root of the tree, and the final binary search tree is shown in below figure.

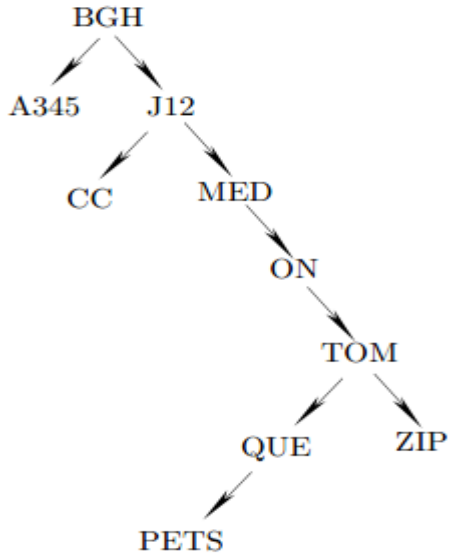


Figure: Binary Search Tree

The minimum number of steps for insertion or search is obviously 1. The maximum number of steps depends on the height of the tree. The tree in the above figure has a height of 7, so the next insertion will require from 1 to 7 steps. The height of a binary tree with N nodes varies between $\log_2 N$ (which is the height of a fully balanced tree), and N (the height of a skewed tree). It can be proved that an average binary tree is closer to a balanced tree than to a skewed tree, and this implies that the average time for insertion or search in a binary search tree is of the order of $\log_2 N$.

Advantages: Efficient operation (as measured by the average number of steps). Flexible size.
Disadvantages: Each step is more complex than in an array-based symbol table.

The recommendations for use are the same as for the previous method.

A Hash Table

This method comes in two varieties, closed hash, which is a fixed-size array and open hash, which uses pointers and has a variable size.

Closed hashing

A closed hash table is an array (actually two or more arrays, for the name, value (location), and any other attributes), normally of size 2^N , where each symbol is stored in an entry.

To insert a new symbol, it is necessary to obtain an index to the entry where the symbol will be stored. This is done by performing an operation on the name of the symbol, an operation that results in an N-bit number. An N-bit number has a value between 0 and $2^N - 1$ and can thus serve as an index to the array. The operation is called hashing and is done by hashing, or scrambling, the bits that constitute the name of the symbol. For example, consider 6-character names, such as abcdef. Each character is stored in memory as an 8-bit ASCII code. The name is divided into three groups of two characters (16-bits) each, ab cd ef. The three groups are added, producing an 18-bit sum. The sum is split into two 9-bit halves which are then multiplied to give an 18-bit product. Finally N bits are extracted from the middle of the product to serve as the hash index. The hashing operations are meaningless since they operate on codes of characters, not on numbers. However, they produce an N-bit number that depends on all the bits of the original name.

A good hash function should have the following two properties:

- It should consider all the bits in the original name. Thus when two names that are slightly different are hashed, there should be a good chance of producing different hash indexes.
- For a group of names that are uniformly distributed over the alphabet, the function should produce indexes uniformly distributed over the range 0 ... $2^N - 1$.

Once the hash index is produced, it is used to insert the symbol into the array. Searching for symbols is done in an identical way. The given name is hashed, and the hashed index is used to retrieve the value (location) and any other attributes from the array.

Ideally, a hash table requires fixed time for insert and search, and can be an excellent choice for a large symbol table. There are, however, two problems associated with this method namely, collisions and overflow, that make hash tables less than ideal.

Collisions involve the case where two entirely different symbol names are hashed into identical indices. Names such as SYMB and ZWYG6 can be hashed into the same value, say, 54. If SYMB is encountered first in the program, it will be inserted into entry 54 of the hash table. When ZWYG6 is found, it will be hashed, and the assembler should discover that entry 54 is already taken. The collision problem cannot be avoided just by designing

a better hash function. The problem stems from the fact that the set of all possible symbols is very large, but any given program uses a small part of it. Typically, symbol names start with a letter, and consist of letters and digits only. If such a name is limited to six characters, then there are 26 × 365 (≈ 1.572 billion) possible names. A typical program rarely contains more than, say, 500 names, and a hash table of size 512 (= 29) may be sufficient. When 1.572 billion names are mapped into 512 positions, more than 3 million names will map into each position. Thus even the best hash function will generate the same index for many different names, and a good solution to the collision problem is the key to an efficient hash table.

The simplest solution involves a linear search. All entries in the symbol table are originally marked as vacant. When the symbol SYMB is inserted into entry 54, that entry is marked occupied. If symbol ZWYG6 should be inserted into entry 54 and that entry is occupied, the assembler tries entries 55, 56 and so on. This implies that, in the case of a collision, the hash table degrades to a linear table.

Another solution involves trying entry 54 + P where P and the table size are relative primes. In either case, the assembler tries until a vacant entry is found or until the entire table is searched and found to be all occupied.

It is seen that when the hash table gets more than 50%–60% full, performance suffers, no matter how good the hashing function is. Thus a good hash table design makes sure that the table never gets more than 60% occupied. At that point the table is considered overflowed.

The problem of hash table overflow can be handled in a number of ways. Traditionally, a new, larger table is opened and the original table is moved to the new one by rehashing each element. The space taken by the original table is then released. A better solution, though, is to use open hashing.

Open hashing

An open hash table is a structure consisting of buckets, each of which is the start of a linked list of symbols. It is very similar to the buckets with linked lists discussed above. The principle of open hashing is to hash the name of the symbol and use the hash index to select a bucket. This is better than using the first character in the name, since a good

hash function can evenly distribute the names over the buckets, even in cases where many symbols start with the same letter.

OTHER OPERATION PERFORMED ON ANY LIST:

1. Sort the List
2. Addition of Two List
3. Display List in the reverse order
4. Display every alternate nodes in the list
5. Find the length of the list
6. Search for an element in the list.
7. Swap references without swapping the data
8. Remove the duplicates elements in list
9. Separate even and odd nodes in the list
10. Check for palindrome

APPLICATIONS OF LINKED LIST

1. Implementation of other Data Structures like - Stacks, Queues, Trees, Graphs, Hash Tables
2. Allocation and deallocation of memory
3. Performing arithmetic operations on long integers - Addition, Subtraction, Multiplication and Division
4. Representation of Sparse Matrix
5. Maintaining a Dictionary
6. Designing a Text Editor, Photo editor
7. Music Player.
8. Blockchain(Bitcoin)
9. Pattern Matching
10. Database applications - Student / employee record details, hotel management, reservation system etc…

REFERENCES

1. "Data Structures and Program Design in C", Robert Kruse, Cl Tondo, Pearson Education, 2nd Edition, 2007.
2. https://en.wikipedia.org/wiki/Data_structure#:~:text=In%20computer%20science%2C%20a%20data,be%20applied%20to%20the%20data.
3. https://www.programiz.com/c-programming/c-dynamic-memory-allocation.
4. http://staff.um.edu.mt/csta1/courses/lectures/csa2060/c8a.html
5. https://computer.howstuffworks.com/c-programming11.htm.
6. https://www.careerride.com/c-static-memory-dynamic-memory-allocation.aspx
7. https://www.geeksforgeeks.org/data-structures/
8. https://www.geeksforgeeks.org/editors-types-system-programming/
9. https://en.wikipedia.org/wiki/Text_editor
10. https://en.wikipedia.org/wiki/Symbol_table
11. https://www.davidsalomon.name/assem.advertis/asl.pdf

Write a C Program to implement a Singly Linked List with the following Operations:

1. **Insert at the beginning of the list**
2. **Insert at the end of the List**
3. **Delete at the beginning of the List**
4. **Delete at the end of the list**
5. **Insert at a specific position**
6. **Delete at a specific position**
7. **Reverse the list**
8. **Display the contents of the list**
9. **Search for a node.**

```
#include<stdio.h>
#include<stdlib.h>
//define structure
struct node
{
  int data;
  struct node* next;
};
//driver program
int main()
{
  struct node* first;
  int ch,x,y;
  void insert_tail(struct node **, int);
  void insert_head(struct node **, int);
  void display(struct node*);
  void delete_node(struct node **,int);
  void delete_pos(struct node **,int);
  void reverse(struct node **);
  void insert_pos(struct node **, int,int);
  first=NULL; // points to the first node
  while(1)
  {
    display(first);
    printf("\n1..Insert tail\n");
    printf("2..Insert head\n");
    printf("3..Display\n");
    printf("4..delete node\n");
```

```
    printf("5..delete position\n");
    printf("6..reverse list..\n");
    printf("7..insert position\n");
    printf("8..Exit\n");
    scanf("%d",&ch);
    switch(ch)
    {
      case 1:printf("Enter the number\n");
             scanf("%d",&x);
             insert_tail(&first,x);
             break;
      case 2:printf("Enter the number\n");
             scanf("%d",&x);
             insert_head(&first,x);
             break;
      case 3: display(first);
             break;
      case 4:printf("Enter the value of node to be deleted\n");
             scanf("%d",&x);
             delete_node(&first,x);
             break;
      case 5:printf("Enter the position of node to be deleted\n");
             scanf("%d",&x);
             delete_pos(&first,x);
             break;
       case 6:reverse(&first);
              break;
       case 7:printf("Enter the value & position \n");
             scanf("%d %d",&x,&y);
             insert_pos(&first,x,y);
             break;
     }
  }
}
```

//insert at a given position

```
void insert_pos(struct node **p , int x,int pos)
  {
    struct node *prev, *temp, *q;
    int i;
    //create node
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=x;
    temp->next=NULL;

     i=1;
    q=*p;
    prev=NULL;

    //go to the desired position
    while((q!=NULL)&&(i<pos))
   {
     prev=q;
     q=q->next;
     i++;
   }
   if(q!=NULL)//position found
   {
      if(prev==NULL)//check if it is first position
        {
         temp->next=*p;//insert at the first position
         *p=temp;
        }
       else// between 2 and the last position
        {
           prev->next=temp;
           temp->next=q;
        }
    }
    else//q==NULL
    {
     if(prev==NULL)//empty list, insert the first node
       *p=temp;
     else if(i==pos)//insert at the end
          prev->next=temp;
      else
```

```
        printf("Invalid position..\n");
    }
}
```

```
//function to reverse the list
void reverse(struct node **p)
{
   struct node *curr, *prev, *temp;

   prev=NULL;
   curr=*p;

   while(curr!=NULL)
   {
    temp=curr->next;
    curr->next=prev;
    prev=curr;
    curr=temp;
   }
   *p=prev;
 }
```

```
//delete a node at a given position
 void delete_pos(struct node **p, int pos)
 {
  int i;
  struct node *q,*prev;
  prev=NULL;

  i=1;
   q=*p;

 //move forward till the position is found or end of list is reached
  while((q!=NULL)&&(i<pos))
```

```
  {
    prev=q;
    q=q->next;
    i++;
  }
 if(q==NULL) // end of list reached
    printf("invalid position.\n");
   else if(prev==NULL)//first node is being deleted
    *p=q->next;//make second as the first node
   else
     prev->next=q->next;
   free(q);
}


 //delete a node given its value.
//deletes only the first occurrence of the node
 void delete_node(struct node **p, int x)
 {
   struct node *q,*prev;
   prev=NULL;

   //keep moving forward till the node to be deleted
   //is found or you go beyond the last node
   q=*p;
   while((q!=NULL)&&(q->data!=x))
   {
     prev=q;
     q=q->next;
    }
    if(q==NULL)
     printf("the node not found..\n");
    else if(prev==NULL)//first node is being deleted
     *p=q->next;//make second as the first node
    else
      prev->next=q->next;
    free(q);
   }
```

```
//insert the node at the front of the list
  void insert_head(struct node **p,int x)
  {
    struct node *temp;

    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=x;
     temp->next=NULL;

    //check if this is the first element
    if(*p==NULL)
      *p=temp;
    else
     {
        temp->next=*p;
        *p=temp;
     }
 }

//insert the node at the end of the list
  void insert_tail(struct node **p,int x)
  {
     struct node *temp,*q;
   //create node
     temp=(struct node*)malloc(sizeof(struct node));
   //copy the values in the node

    temp->data=x;
    temp->next=NULL;

    //check if this is the first node
    if(*p==NULL)//checking the content of first
      *p=temp;
    else
      {
         //go to the end of the list
       q=*p;
       while(q->next!=NULL)
          q=q->next;//move forward
```

```
            q->next=temp;//link the new node to the last node
          }
        }

       //displays the list
       void display(struct node *p)
       {
         if(p==NULL)
          printf("Empty List..\n");
        else
         {
          while(p!=NULL)
          {
            printf("%d ->",p->data);
            p=p->next;
          }
        }
      }
```

//delete the first node

```
void delete_first(struct node **p)
{
  struct node *q;
  //check for empty list
   if(*p==NULL)
        printf("Empty List..");
   else
    {
      q=*p;
     *p= q->next;
       free(q);
   }
}
```

//delete the last node

```
void delete_last(struct node **p)
{
  struct node *q,*prev;
     //check for empty list
```

```
  if(*p==NULL)
    printf("Empty List..\n");
 else
   {
     prev=NULL;
     q=*p;
    //go to the last node
     while( q->next!=NULL)
    {
       prev=q;
      q=q->next;
      }
   if(prev!=NULL) // check if there is only one node
       prev->next=NULL;
   free(q);
 }

//search for a key , using linear search
void search(struct node **p, int key)
{
  struct node *q;

  if(*p==NULL)
   printf("Empty List\n");
 else
   {
       q=*p;
       while((q!=NULL) &&(q->data!=key))
           q=q->next;
       if(q==NULL)
         printf("key found..\n")
else
    printf("key found\n")
  }
}
```

Program to create an ordered list, elements will be inserted in the ascending order

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

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

void display(struct node*);
void insert_order(struct node**,int);
int main()
{
  struct node *first;
  int x;
   first=NULL; //points to the first node

  while(1)
  {
    printf("\nEnter the number..\n");
    scanf("%d",&x);
     if(x==0)
       break;
    insert_order(&first,x);
    display(first);
  }
}

 void display(struct node *p)
 {
   printf("The list..\n");
   while(p!=NULL)
   {
     printf("%d->",p->data);
     p=p->next;
   }
}

  void insert_order(struct node **p, int x)
  {
    struct node *temp, *prev, *q;

    temp=(struct node*)malloc(sizeof(struct node));
```

```
  temp->data=x;
  temp->next=NULL;

  q=*p;
  prev=NULL;;

  //move forward until the position of the element is found
  while((q!=NULL)&&(x>q->data))
  {
    prev=q;
    q=q->next;
   }
  if(q!=NULL)
  {
    if(prev==NULL)//inserting the smallest number, insert first
    {
      temp->next=q;
      *p=temp;
     }
   else// insert somewhere in middle of the list
    {
      temp->next=q;
      prev->next=temp;
    }
  }
 else//q==NULL
 {
   if(prev==NULL)//empty list, first node inserted
        *p=temp;
  else
    prev->next=temp;//largest no, insert at end
 }
}
```

Write a C Program to perform the following operations on a Doubly Linked List:

1. **Insert at the beginning of the list**
2. **Insert at the end of the list**

3. **Insert at a specific position**
4. **Delete at the beginning of the list**
5. **Delete at the end of the list.**
6. **Delete at a specific position in the list.**
7. **Display the contents of the list.**

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *prev;
   struct node *next;
};
void insert_head(struct node**, int);
void insert_tail(struct node **, int);
void delete_node(struct node**, int);
void insert_pos(struct node **,int ,int);
void delete_pos(struct node**, int);
void display(struct node*);
void delete_last(struct node **);
void delete_first(struct node **);
int main()
{
  struct node *first;
  first=NULL;
  int x,ch,pos;
  while(1)
  {
    display(first);
    printf("\n1..Insert Head..\n");
    printf("2..Insert Tail..\n");
    printf("3..Delete First..\n");
    printf("4..Delete Last..\n");
    printf("5..Delete node..\n");
    printf("6..Delete at position\n");
    printf("7..Insert at position\n");
    scanf("%d",&ch);

    switch(ch)
```

```
   {
      case 1: printf("Enter the value..\n");
              scanf("%d",&x);
              insert_head(&first,x);
              break;
       case 2: printf("Enter the value..\n");
              scanf("%d",&x);
              insert_tail(&first,x);
              break;
       case 3:delete_first(&first);
               break;
        case 4:delete_last(&first);
               break;
        case 5: printf("Enter the value..\n");
              scanf("%d",&x);
              delete_node(&first,x);
              break;
       case 6: printf("Enter the position..\n");
              scanf("%d",&x);
              delete_pos(&first,x);
              break;
       case 7: printf("Enter the value and position..\n");
              scanf("%d %d",&x,&pos);
              insert_pos(&first,x,pos);
              break;

   }

  }
}
```

//delete the first node

```
void delete_first(struct node **p)
  {
    struct node *q;
    q=*p;

    if((q->prev==NULL)&&(q->next==NULL))//only one node
       *p=NULL;
```

```
    else //more than one node in the list
     {
        *p=q->next;
        (*p)->prev=NULL;
     }
    free(q);
   }
```

//delete the last node

```
 void delete_last(struct node **p)
 {
   struct node *q;
   q=*p;

   if((q->prev==NULL)&&(q->next==NULL))//only one node
      *p=NULL;
   else //more than one node in the list
    {
       while(q->next!=NULL)
         q=q->next;

     q->prev->next=NULL;
    }
   free(q);
  }
```

//insert at a given position

```
void insert_pos(struct node **p,int x,int pos)
{
  struct node *temp, *q;

 //create a node

    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=x;
    temp->prev=temp->next=NULL;

   q=*p;
   int i=1;
```

```
//go to the position

while((q->next!=NULL)&&(i<pos))
 {
   i++;
    q=q->next;
  }
if(q->next!=NULL)//position found
{
  //check if first position
   if(q->prev==NULL)
   {
      //insert in first position
      temp->next=q;
      q->prev=temp;
      *p=temp;
    }
  else
  {
   //insert somewhere in the middle of list
   //but not the last but one position
   q->prev->next=temp;
   temp->prev=q->prev;
   temp->next=q;
    q->prev=temp;
  }
}
else//q->next==NULL
 {
  if(i==pos)//insert at the last but one position
  {
    q->prev->next=temp;
   temp->prev=q->prev;
   temp->next=q;
    q->prev=temp;
 }
 else if(i==pos-1)//insert after the last node
 {
    q->next=temp;
     temp->prev=q;
```

```
    }
   else
     printf("Invalid position..\n");
   }
}

//delete at a given position
 void delete_pos(struct node**p, int pos)
  {
    struct node *q;

    //find the node to be deleted
    q=*p;

    int i=1;
    while((q!=NULL)&&(i<pos))
    {
      i++;
     q=q->next;
    }

   if(q!=NULL)//position found
   {
     if((q->prev==NULL)&&(q->next==NULL))//only one node
          *p=NULL;
     else if(q->prev==NULL)//first position
     {
        *p=q->next;
       (*p)->prev=NULL;
     }
      else if(q->next==NULL)//last position
         q->prev->next=NULL;
     else //somewhere in middle
       {
          q->prev->next=q->next;
           q->next->prev=q->prev;
       }
     free(q);
   }
   else//q=NULL
```

```
    printf("Invalid position.\n");
}
```

//delete the first occurrence of a node given its value

```
void delete_node(struct node**p, int x)
 {
   struct node *q;

   //find the node to be deleted
   q=*p;

   while((q!=NULL)&&(q->data!=x))
     q=q->next;

  if(q!=NULL)//node found
  {
    if((q->prev==NULL)&&(q->next==NULL))//only one node
        *p=NULL;
    else if(q->prev==NULL)//first node
    {
       *p=q->next;
      (*p)->prev=NULL;
    }
     else if(q->next==NULL)//last node
        q->prev->next=NULL;
    else //somewhere in middle
     {
        q->prev->next=q->next;
         q->next->prev=q->prev;
     }
    free(q);
  }
  else//q=NULL
    printf("Node not found..\n");
}
```

//insert a node at the front of the list

```
 void insert_head(struct node **p, int x)
```

```
 {
    struct node *temp;
   //create a node

   temp=(struct node*)malloc(sizeof(struct node));
   temp->data=x;
   temp->prev=temp->next=NULL;

  //if this is the first node

   if(*p==NULL)
      *p=temp;//make first point to temp
    else
    {
      temp->next=*p;//link the new node to first node
      (*p)->prev=temp;
      *p=temp;//make  first point to new node
     }
  }
```

//display the contents of the list

```
void display(struct node *p)
 {
    if(p==NULL)
      printf("\nEmpty List..\n");
    else
     {
       while(p!=NULL)
       {
         printf("%d<->",p->data);
         p=p->next;
       }
     }
 }
```

//insert the node at the end of the list

```
void insert_tail(struct node **p, int x)
{
```

```
    struct node *temp,*q;
    //create a node

    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=x;
    temp->prev=temp->next=NULL;

   //if this is the first node

    if(*p==NULL)
        *p=temp;//make first point to temp
    //go to the end of the list
   else
   {
    q=*p;
    while(q->next!=NULL)
      q=q->next;

    q->next=temp;//link the new node to last node
    temp->prev=q;
   }
 }
```

Implement the following operations on a Circular Singly Linked List:

1. **Insert at the beginning of the list**
2. **Insert at the end of the list**
3. **Delete a node given its value**
4. **Display the list**

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
  int data;
  struct node* next;
};
void insert_head(struct node**,int);
void insert_tail(struct node**,int);
```

```
void delete_node(struct node**,int);
void display(struct node*);

int main()
{
  struct node *last;
  int ch,x,pos;
  last=NULL;//pointer to the last node of the list

  while(1)
  {
    display(last);
    printf("\n1..Insert Head\n");
    printf("2..Insert Tail\n");
    printf("3..Delete a Node..\n");
    printf("4..Display\n");
    printf("5..Exit\n");
    scanf("%d",&ch);
    switch(ch)
    {
     case 1:printf("Enter the number...");
            scanf("%d",&x);
            insert_head(&last,x);
            break;
     case 2:printf("Enter the number...");
            scanf("%d",&x);
            insert_tail(&last,x);
            break;
    case 3: printf("Enter the value of the node to be deleted...");
            scanf("%d",&x);
            delete_node(&last,x);
            break;

     case 4:display(last);
            break;
     case 5:exit(0);
   }
 }
}
```

```
//delete node given its value
  void delete_node(struct node**p,int x)
{
  struct node *prev,*q,*r;

  q=*p;//copy of the last node address
  prev=q;//keep track the previous node

  r=q->next;//first node

  //move forward till you find the data
  //or you stop at the last node
  while((r!=q)&&(r->data!=x))
  {
    prev=r;
    r=r->next;
  }
  if(r->data==x)//node found
  {
      if(r->next==r)//only one node
        *p=NULL;
      else
       {
          prev->next=r->next;
          if(r==q)//deleting the last node
            *p=prev;
        }
       free(r);
   }
   else
     printf("Node not found..\n");
 }

//insert a node at the head of the list
 void insert_head(struct node**p,int x)
 {
    struct node *temp;

    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=x;
```

```
    temp->next=temp;

    if(*p==NULL)
      *p=temp;
    else
     {
        temp->next=(*p)->next;
        (*p)->next=temp;
     }
  }

//display the contents of the list
void display(struct node *p)
{
   struct node *q;

  if(p==NULL)
   printf("\nEmpty list..\n");
 else
  {
  q=p->next;
   while(q!=p)
   {
      printf("%d ->",q->data);
      q=q->next;
   }
  printf("%d ->",q->data);//last node
  }
}

//insert a node at the end of the list
  void insert_tail(struct node**p,int x)
 {
    struct node *temp;

    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=x;
    temp->next=temp;

    if(*p==NULL)
```

```
    *p=temp;
  else
   {
     temp->next=(*p)->next;
     (*p)->next=temp;
      *p=temp;
   }
}
```

Implement the following operations on a Circular Singly Linked List:
The list has a header node. The header node keeps the count of the number of nodes in the list. Empty List contains only the header node

1. **Insert at the beginning of the list**
2. **Insert at the end of the list**
3. **Delete a node given its value**
4. **Display the list**

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

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

void insert_head(struct node *,int);
void insert_tail(struct node *,int);
void display(struct node *);
struct node *create_head();
void delete_node(struct node*,int);
```

//implementing circular list with a header node

```
int main()
{
  struct node *head;
   int x,ch;
   head=create_head(); //head points to the header node
   while(1)
  {
    display(head);
    printf("\n1..Insert Head\n");
    printf("2..Insert Tail\n");
    printf("3..Delete a Node..\n");
    printf("4..Exit\n");
    scanf("%d",&ch);
    switch(ch)
    {
     case 1:printf("Enter the number...");
            scanf("%d",&x);
            insert_head(head,x);
            break;
     case 2:printf("Enter the number...");
            scanf("%d",&x);
            insert_tail(head,x);
            break;

     case 3: printf("Enter the value of the node to be deleted...");
            scanf("%d",&x);
            delete_node(head,x);
            break;

     case 4:exit(0);
   }
 }
}


//creates a header node

 struct node *create_head()
 {
```

```
  struct node *temp;
  temp=(struct node*)malloc(sizeof(struct node));
  temp->data=0;
  temp->next=temp;
  return temp;
 }

//insert a node at the front of the list
//i.e. after the header node
 void insert_head(struct node *p,int x)
 {
  struct node *temp;
  temp=(struct node*)malloc(sizeof(struct node));
  temp->data=x;
  temp->next=p->next; // insert after the header node
  p->next=temp;
  p->data++;//increment the count of the nodes
 }

 //display the contents of the list
 void display(struct node *p)
 {
  struct node *q;
  q=p;

  while(p->next!=q)
  {
   printf("%d-> ",p->data);
   p=p->next;
  }
  printf("%d ",p->data);
 }


//insert at the end of the list
 void insert_tail(struct node *p,int x)
 {
  struct node *temp,*q;
   temp=(struct node*)malloc(sizeof(struct node));
   temp->data=x;
```

```
   q=p->next;

   while(q->next!=p)
    q=q->next;

   temp->next=q->next;//or temp->next=p
   q->next=temp;
   p->data++;
 }



//delete a node given its value
void delete_node(struct node *p, int x)
 {
   struct node *prev,*q;

   q=p->next;
   prev=p;

  //move forward until the node to be deleted is found or the header node is reached
   while((q!=p)&&(q->data!=x))
   {
    prev=q;
    q=q->next;
   }
   if(q==p)
    printf("Node not found..\n");
  else
  {
   prev->next=q->next; //delete the node
   free(q);
   p->data--;//decrement the count in the header node
  }
}
```

Program to merge two singly linked list

```
#include<stdio.h>
```

```
#include<stdlib.h>
struct node
{
  int data;
  struct node* next;
};

void display(struct node*);
void insert_order(struct node**,int);
void createlist(struct node**);
void merge(struct node*,struct node*,struct node**);
int main()
{
  struct node *first,*second,*third;
  first=NULL;
  second=NULL;
  third=NULL;
 printf("Creating the first List..\n");
 createlist(&first);
 display(first);
 printf("\nCreating the second List..\n");
 createlist(&second);
 display(second);
 printf("\nMerging the lists..\n");
 merge(first,second,&third);
 display(third);
}

void display(struct node *p)
 {
   printf("The list..\n");
   while(p!=NULL)
   {
     printf("%d->",p->data);
     p=p->next;
   }
}

void createlist(struct node**p)
{
```

```
  int x;
  while(1)
  {
    printf("\nEnter the number..\n");
    scanf("%d",&x);
     if(x==0)
       break;
    insert_order(p,x);
  }
}

void insert_order(struct node** p,int x)
 {
  struct node *temp,*prev,*q;

  temp=(struct node*)malloc(sizeof(struct node));
  temp->data=x;
  temp->next=NULL;

  q=*p;//copy address of the first node
  prev=NULL;

  while((q!=NULL)&&(x>q->data))
  {
    prev=q;
    q=q->next;
  }
  if(q!=NULL)//position found,x<q->data
  {
   if(prev==NULL)//first position, insert as the first node
    { temp->next=q;
    *p=temp;
    }
   else//insert somewhere in middle
   {
    prev->next=temp;
    temp->next=q;
   }
  }
  else//q==NULL,
```

```
 {
   if(prev==NULL)//empty list,insert the node as the first node
     *p=temp;
   else
    prev->next=temp;//insert at the end of the list
}
}

void insert_tail(struct node **p, int x)
 {

  struct node *temp, *q;
   //create node
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data = x;
    temp->next=NULL;

    //if list empty
    if(*p==NULL)
      *p=temp;
    else
    {
      q=*p;//copy the address of teh first node of the list
    //keep moving till you stop at the last node
     while(q->next!=NULL)
      q=q->next;

     q->next=temp;//link the last node to new node
    }
 }
```

//merging the two lists

```
void merge(struct node* p,struct node* q,struct node** t)
{
  while((p!=NULL)&&(q!=NULL))
  {
    if(p->data <= q->data)
    {
      insert_tail(t,p->data);
      p=p->next;
```

```
    }
   else
   {
     insert_tail(t,q->data);
      q=q->next;
   }
  }
  if(p==NULL)//end of the first list
  {
    while(q!=NULL)//copy all the elements of the second list
    {
      insert_tail(t,q->data);
      q=q->next;
    }
  }
  else// q==NULL end of the second list
  {
    while(p!=NULL)//copy all the elements of the first list
    {
      insert_tail(t,p->data);
      p=p->next;
     }
   }
}
```

Implementation of Sparse Matrix Using Multilist:
Representing a sparse matrix as a multi list

```
#include<stdio.h>
#include<stdlib.h>
//used to store non zero value
struct col_node {
        int col; //column index
        int data; //non zero value
        struct col_node *next_col;
};

//represents the row
struct row_node {
        int row;
        struct col_node *next_col; // points to the first non zero value of that row
        struct row_node *next_row; //points to the next row
};
struct row_node *create_rows(int);
void insert_list(struct row_node*, int,int,int);
void display(struct row_node*);
int main()
{
  int a[10][10];
  int i,j,row,col;
  struct row_node *first,*p;
  first=NULL;
  printf("Enter the row and cols..\n");
  scanf("%d %d",&row, &col);


  printf("Enter the data for the matrix..\n");
  for(i=0;i<row;i++)
  {
     for(j=0;j<col;j++)
      scanf("%d",&a[i][j]);
  }
  //storing the matrix as a multi list...;

  first=create_rows(row);
  p=first;
```

```
 for(i=0;i<row;i++)
 {
    for(j=0;j<col;j++)
      if(a[i][j]!=0)
        insert_list(p,i,j,a[i][j]);
    p=p->next_row;

 }
 //displaying the matrix as a list

 display(first);
}


 //creates a linked list to represent the rows
 struct row_node* create_rows(int r)
 {
  struct row_node *p,*q;
  struct row_node *temp;

  int i;
  p=NULL; // points to the first row node
  q=NULL; // points to the last row node
  //create r number of row nodes
  for(i=0;i<r;i++)
  {
   temp=malloc(sizeof(struct row_node));
   temp->row=i;
   temp->next_row=NULL;
   temp->next_col=NULL;

   if (p==NULL)//first node
   {
     p=temp;
     q=temp;
    }
    else
    {
     q->next_row=temp;
     q=temp;
```

```
    }
   }
   return p;//return the address of the first row node
 }

  void insert_list(struct row_node *p,int row, int col, int x)
  {
    struct col_node *q,*prev,*temp;
    int i,j;


    temp=malloc(sizeof(struct col_node));
    temp->col=col;
    temp->data=x;
    temp->next_col=NULL;

    //insert each column node at the end of the list
    q=p->next_col;
    if(q==NULL)
     p->next_col=temp;
   else
   {
    while(q->next_col!=NULL)
     q=q->next_col;
    q->next_col=temp;
   }
}


  void  display(struct row_node *p)
  {
    struct col_node *q;
    printf("\n");
    while(p!=NULL)
    {
      printf("%d ->",p->row);
      q=p->next_col;
      while(q!=NULL)
      {
       printf("%d,",q->col);
```

```
      printf("%d -> ",q->data);
      q=q->next_col;
     }
    p=p->next_row;
    printf("\n");
   }
 }
```

Program to add two long numbers using circular lists with header node
The numbers are read as an array of characters, later each character converted to a integer

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

 struct node
 {
  int data;
  struct node *next;
 };
  void insert_first(int,struct node*);
  void create_head(struct node **);
  void sum(struct node *,struct node *, struct node *);
 void create_list(char*,struct node*);
 void display(struct node *);
 int main()
 {
  struct node *head1,*head2,*head3;
  char a[10],b[10];
  int x,ch,pos;
  head1=head2=head3=NULL;
  create_head(&head1);//creates the header node
  create_head(&head2);
  create_head(&head3);
  printf("Enter the first number\n");
  scanf("%s",a);
  printf("Enter the second number\n");
  scanf("%s",b);
  create_list(a,head1);
  printf("\n");
```

```
 display(head1);
 create_list(b,head2);
 printf("\n");
 display(head2);
 sum(head1,head2,head3);
 printf("\nsum=");
 display(head3);
}

 //convert each character into a number and insert into the list
//the list will be created in the reverse order
 void create_list(char *a, struct node *h)
 {
    int i=0;
    while(a[i]!='\0')
      {
       insert_first(a[i]-'0',h);
        i++;
      }

   }


  // create a header node
   void create_head(struct node **h)
   {
     struct node *temp;
     temp=(struct node*)malloc(sizeof(struct node));
     temp->data=0;
     temp->next=temp;
      *h=temp;
    }



//insert at the head of the list
void insert_first(int x, struct node *p)
  {
    struct node *temp;
    temp=(struct node*)malloc(sizeof(struct node));
```

```
 temp->data=x;
 temp->next=p->next;
 p->next=temp;
}

//finds the sum of the two lists
void sum(struct node *h1,struct node *h2, struct node *h3)
{
  struct node *p,*q;
  int sum,carry;
  sum=carry=0;
  p=h1->next;
  q=h2->next;

   while((p!=h1)&&(q!=h2))
   {
     sum=(p->data+q->data+carry)%10;
     carry=(p->data+q->data+carry)/10;
     insert_first(sum,h3);
     p=p->next;
     q=q->next;
    }
   if(p==h1)
   {
     while(q!=h2)
     {
        sum=(q->data+carry)%10;
        carry=(q->data+carry)/10;
        insert_first(sum,h3);
        q=q->next;

      }
   }
   else
    {
     while(p!=h1)
     {
      sum=(p->data+carry)%10;
       carry=(p->data+carry)/10;
       insert_first(sum,h3);
```

```
          p=p->next;
         }
        }
      if(carry!=0)
         insert_first(carry,h3);
     }

void display(struct node *p)
  {
    struct node *q;
    q=p;
    q=p->next;
    while(q!=p)
    {
      printf("->%d ",q->data);
      q=q->next;
     }
   }
```

Program to add two polynomials implemented as a singly linked lists

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

 struct node
 {
  int coeff;
  int px;
  int py;
  int flag;
  struct node *next;
 };
  void createpoly(struct node **);
  void insert_tail(int,int,int,struct node**);
  void display(struct node*);
  void polyadd(struct node*,struct node*, struct node**);
 main()
 {
  struct node *first,*second,*third;
```

```
 first = NULL;
 second=NULL;
 third=NULL;
 int cf,px,py,result;

 printf("\nCreating first polynomial..\n");
 createpoly(&first);
 printf("\nCreating the second polynomial..\n");
 createpoly(&second);
 printf("\nAdding the two polynomials & displaying the result..\n");
 polyadd(first,second,&third);
 display(third);
}

 void createpoly(struct node **p)
 {
   int cf,px,py;
  while(1)
  {
     printf("\nEnter the coefficient..");
     scanf("%d",&cf);
     if(cf==0)
       break;
     printf("\nEnter the power of x..");
     scanf("%d",&px);
     printf("\nEnter the power of y...");
     scanf("%d",&py);
     insert_tail(cf,px,py,p);
  }

   printf("\nThe polynomial created...\n");
   display(*p);
  }

  void display(struct node *q)
   {
     while(q!=NULL)
     {
        if(q->coeff>0)
```

```
      printf(" +%d ",q->coeff);
     else
      printf(" %d ",q->coeff);
     if(q->px>0)
     {
       if(q->px==1)
         printf("X");
       else
         printf("X^%d",q->px);;
      }
      if(q->py>0)
      {
       if(q->py==1)
        printf("Y");
       else
        printf("Y^%d",q->py);
      }
     q=q->next;
   }
 }


void insert_tail(int cf,int px,int py, struct node **p)
{
  struct node *q,*temp;
  temp=(struct node*)malloc(sizeof(struct node));
  temp->coeff=cf;
  temp->px=px;
  temp->py=py;
  temp->flag=1;
  temp->next=NULL;

  q=*p;
  if(q==NULL)//if it is the first node
    *p=temp;
  else
   {
    while(q->next!=NULL)//go to the last node
      q=q->next;
    q->next=temp;
```

```
  }
}

void polyadd(struct node *p,struct node *q,struct node **t)
{
  int x1,y1,cf,c1,x2,y2,c2;
  struct node *q1;
  while(p!=NULL)
  {
    c1=p->coeff;
    x1=p->px;
    y1=p->py;
    q1=q;
    while(q1!=NULL)
    {
      c2=q1->coeff;
      x2=q1->px;
      y2=q1->py;
      if((x1==x2)&&(y1==y2))
         break;
      q1=q1->next;
     }
     if(q1!=NULL)//still in mid of second poly and found the powers equal
     {
      cf=c1+c2;//add the coefficient
      q1->flag=0;
       if(cf!=0)
         insert_tail(cf,x1,y1,t);//add the sum coeff to the poly
      }
      else
         insert_tail(c1,x1,y1,t);//add the first term to poly;
      p=p->next;
  }

  q1=q;
   while(q1!=NULL)
   {
       if(q1->flag==1)
          insert_tail(q1->coeff,q1->px,q1->py,t);
        q1=q1->next;
```

```
    }
  }
```

Program to evaluate a polynomial implemented as a linked list

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

 struct node
 {
  int coeff;
  int px;
  int py;
  struct node *next;
 };
  void insert_tail(int,int,int,struct node**);
  void display(struct node *);
  int polyevaluate(struct node *);
 main()
 {
  struct node *first;

  first = NULL;
  int cf,px,py,result;
  while(1)
  {
     printf("\nEnter the coefficient..");
     scanf("%d",&cf);
     if(cf==0)
       break;
     printf("\nEnter the power of x..");
     scanf("%d",&px);
     printf("\nEnter the power of y...");
     scanf("%d",&py);
     insert_tail(cf,px,py,&first);
   }
   printf("\nThe polynomial created...\n");
```

```
  display(first);
 printf("\nEvaluating the polynomial..\n");
 result=polyevaluate(first);
 printf("Result=%d",result);
}

 int polyevaluate(struct node *p)
 {
   int x,y,sum;
   printf("\nEnter the value of x and y..");
   scanf("%d %d",&x,&y);
   sum=0;
   while(p!=NULL)
   {
     sum=sum+(p->coeff*pow(x,p->px)*pow(y,p->py));
     p=p->next;
   }
  return sum;
}

 void display(struct node *q)
  {
    while(q!=NULL)
    {
       if(q->coeff>0)
        printf("+%d",q->coeff);
       else
        printf("%d",q->coeff);
       if(q->px>0)
       {
         if(q->px==1)
           printf("X");
         else
           printf("X^%d",q->px);;
        }
        if(q->py>0)
        {
         if(q->py==1)
          printf("Y");
         else
```

```
        printf("Y^%d",q->py);
      }
     q=q->next;
   }
 }
```

```
void insert_tail(int cf,int px,int py, struct node **p)
{
  struct node *q,*temp;
  temp=(struct node*)malloc(sizeof(struct node));
  temp->coeff=cf;
  temp->px=px;
  temp->py=py;
  temp->next=NULL;

  q=*p;
  if(q==NULL)//if it is the first node
    *p=temp;
  else
   {
    while(q->next!=NULL)//go to the last node
      q=q->next;
    q->next=temp;
  }
}
```

References from the Text Book and Reference Book

Book	Chapter	Section	Topic	Page Number
	1	1.1	Data structures and C	22

Data Structures using C & C++, Yedidyah Langsam, Moshe J. Augenstein, Aaron M. Tenenbaum , 2nd Edition, 2015	**4**	**4.2**	**Linked Lists : Inserting and Removing Nodes from a list**	**187**
			Linked List as Data Structure	**195**
			Examples of List Operations	**198**
		4.3	**Allocating and Freeing Dynamic Variables**	**207**
			Linked Lists using Dynamic Variables	**215**
			Examples of List operations in C	
		4.5	**Circular Lists**	**229**
			Addition of Long positive integers using Circular Lists	**235**
			Doubly linked Lists	**237**
			Addition of Long positive integers using Doubly Linked Lists	**239**

Data Structures and Program Design in C, Robert L. Kruse, Clovis L. Tando, Bruce P. Leung, 2nd Edition, 2007	**4**	**4.5**	**Pointers and Linked List**	**151**
	5	**5.1**	**List Specifications**	**185**
		5.2	**Implementation of Lists**	**187**
		5.4	**Application : text Editor**	**201**
	7	**7.2.1**	**Ordered Lists**	**272**