STRUCTURE

A structure is a composite data type that defines a grouped list of variables that are to be placed under one name in a block of memory. It allows different variables to be accessed by using a single pointer to the structure.

struct structure\_name

{

    data\_type member1;

    data\_type member2;

    .

    .

    data\_type member;

};

ADVANTAGES:

* It can hold variables of different data types.
* We can create objects containing different types of attributes.
* It allows us to re-use the data layout across programs.
* It is used to implement other data structures like linked lists, stacks, queues, trees, graphs etc.

EXAMPLE

struct employee

{

**int** id ;

**float** salary ;

**int** mobile ;

} ;

**DYNAMIC MEMORY ALLOCATION**

**C malloc() method**

The **“malloc”** or **“memory allocation”** method in C is used to dynamically 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 doesn’t Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

**Syntax of malloc() in C**

ptr = (cast-type\*) malloc(byte-size)

**For Example:**

***ptr = (int\*) malloc(100\*sizeof(int));***

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

## C calloc() method

1. **“calloc”** or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:
2. It initializes each block with a default value ‘0’.
3. It has two parameters or arguments as compare to malloc().

### Syntax of calloc() in C

ptr = (cast-type\*)calloc(n, element-size);

here, n is the no. of elements and element-size is the size of each element.

***ptr = (float\*) calloc(25, sizeof(float));*** *This statement allocates contiguous space in memory for 25 elements each with the size of the float.*

LINKED LIST

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

In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

**Memory representation of LL**

The efficient way of representing a linked list is using the free pool of storage.

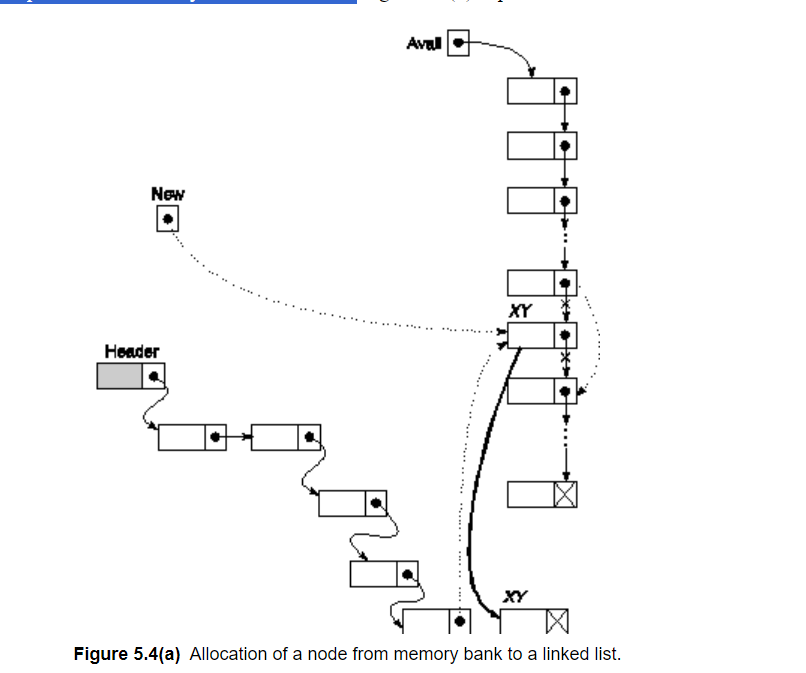
In this method, there is a *memory bank* (which is nothing but a collection of free memory spaces) and a *memory manager* (a program, in fact).

During the creation of a linked list, whenever a node is required the request is placed to the memory manager; the memory manager will then search the memory bank for the block requested and, if found, grants the desired block to the caller.

Again, there is also another program called the *garbage collector*;it plays whenever a node is no more in use; it returns the unused node to the memory bank.

It may be noted that memory bank is basically a list of memory spaces which is available to a programmer. Such a memory management is known as *dynamic* memory management.

The dynamic representation of linked list uses the dynamic memory management policy.



Program to create a linked list

#include <stdio.h>

#include <stdlib.h>

//Represent a node of singly linked list

struct node{

int data; //specifying the data field

struct node \*next; //specifying the address field

};

//Represent the head and tail of the singly linked list

struct node \*head, \*tail = NULL; //initializing them to null initially

//addNode() will add a new node to the list

void addNode(int data) {

//Create a new node

struct node \*newNode = (struct node\*)malloc(sizeof(struct node));

newNode->data = data;

newNode->next = NULL;

//Checks if the list is empty

if(head == NULL) {

//If list is empty, both head and tail will point to new node

head = newNode;

tail = newNode;

}

else {

//newNode will be added after tail such that tail's next will point to newNode

tail->next = newNode;

//newNode will become new tail of the list

tail = newNode;

}

}

//display() will display all the nodes present in the list

void display() {

//Node current will point to head

struct node \*current = head;

if(head == NULL) {

printf("List is empty\n");

return;

}

printf("Nodes of singly linked list: \n");

while(current != NULL) {

//Prints each node by incrementing pointer

printf("%d ", current->data);

current = current->next;

}

printf("\n");

}

int main()

{

//Add nodes to the list

addNode(1);

addNode(2);

addNode(3);

addNode(4);

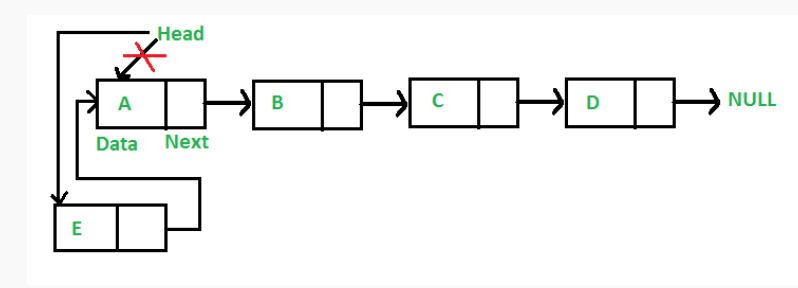
//Displays the nodes present in the list

display();

return 0;

}

ADDING A NEW NODE IN LL- in the beginning



void push(Node\*\* head\_ref, int new\_data)

{

    // 1. allocate node

    Node\* new\_node = new Node();

    // 2. put in the data

    new\_node->data = new\_data;

    // 3. Make next of new node as head

    new\_node->next = (\*head\_ref);

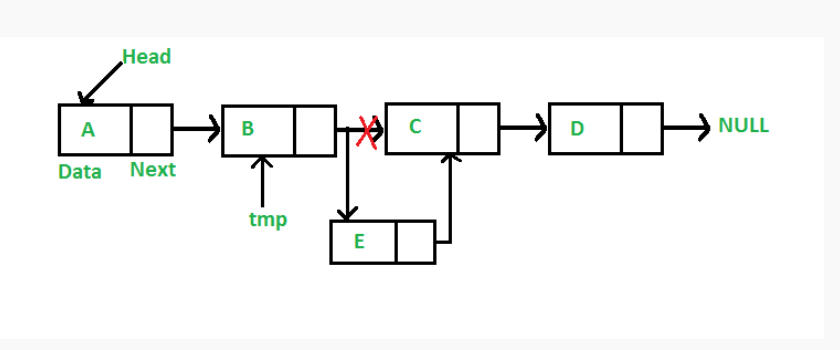
    // 4. Move the head to point to

    // the new node

    (\*head\_ref) = new\_node;

}

INBETWEEN



void insertAfter(Node\* prev\_node, int new\_data)

{

    // 1. Check if the given prev\_node is NULL

    if (prev\_node == NULL) {

        cout << "The given previous node cannot be NULL";

        return;

    }

    // 2. Allocate new node

    Node\* new\_node = new Node();

    // 3. Put in the data

    new\_node->data = new\_data;

    // 4. Make next of new node as

    // next of prev\_node

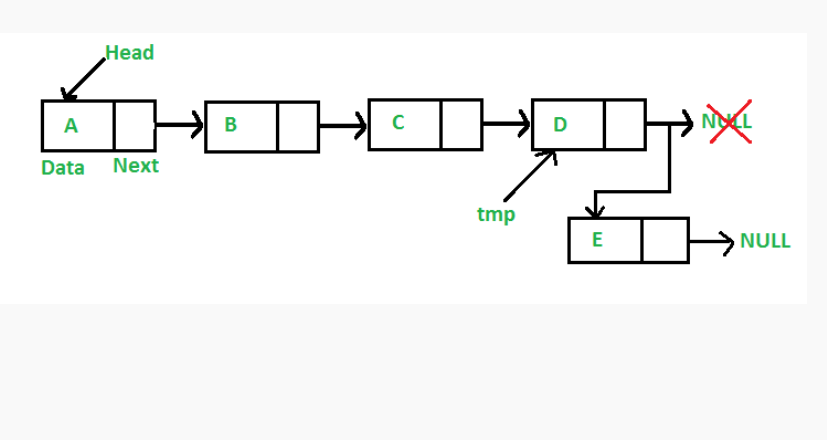
    new\_node->next = prev\_node->next;

    // 5. move the next of prev\_node

    // as new\_node

    prev\_node->next = new\_node;

}

AT THE END:  


while (last->next != NULL) {

        last = last->next;

    }

    // Change the next of last node

    last->next = new\_node;

    return;

}