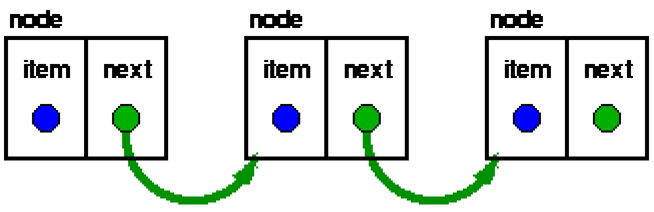
Practical no -6

Aim: Write a program for implementation of Singly Linked list.

Linked lists

The linked list is a very flexible **dynamic data structure**: items may be added to it or deleted from it at will. A programmer need not worry about how many items a program will have to accommodate: this allows us to write robust programs which require much less maintenance. A very common source of problems in program maintenance is the need to increase the capacity of a program to handle larger collections: even the most generous allowance for growth tends to prove inadequate over time!

In a linked list, each item is allocated space as it is added to the list. A link is kept with each item to the next item in the list.



Each node of the list has two elements

1. the item being stored in the list *and*
2. a pointer to the next item in the list

The last node in the list contains a NULL pointer to indicate that it is the end or *tail* of the list.

As items are added to a list, memory for a node is dynamically allocated. Thus the number of items that may be added to a list is limited only by the amount of memory available.

Handle for the list

The variable (or handle) which represents the list is simply a pointer to the node at the *head* of the list.

Adding to a list

The simplest strategy for adding an item to a list is to:

a. allocate space for a new node,
b. copy the item into it,
c. make the new node's next pointer point to the current head of the list *and*
d. make the head of the list point to the newly allocated node.

This strategy is fast and efficient, but each item is added to the head of the list.

An alternative is to create a structure for the list which contains both head and tail pointers:

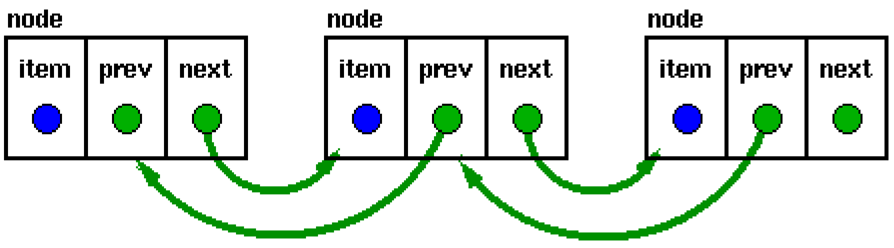
```
struct fifo_list {
        struct node *head;
        struct node *tail;
        };
```

List variants

Circularly Linked Lists

By ensuring that the tail of the list is always pointing to the head, we can build a circularly linked list. If the external pointer (the one in struct t_node in our implementation), points to the current "tail" of the list, then the "head" is found trivially via tail->next, permitting us to have either LIFO or FIFO lists with only one external pointer. In modern processors, the few bytes of memory saved in this way would probably not be regarded as significant. A circularly linked list would more likely be used in an application which required "round-robin" scheduling or processing.

Doubly Linked Lists



Doubly linked lists have a pointer to the preceding item as well as one to the next.

They permit scanning or searching of the list in both directions. (To go backwards in a simple list, it is necessary to go back to the start and scan forwards.) Many applications require searching backwards and forwards through sections of a list: for example, searching for a common name like "Kim" in a Korean telephone directory would probably need much scanning backwards and forwards through a small region of the whole list, so the backward links become very useful. In this case, the node structure is altered to have two links:

```
struct t_node {
    void *item;
    struct t_node *previous;
    struct t_node *next;
    } node;
```

<u>Algorithm:</u>

Steps:

1.[Initially list empty]

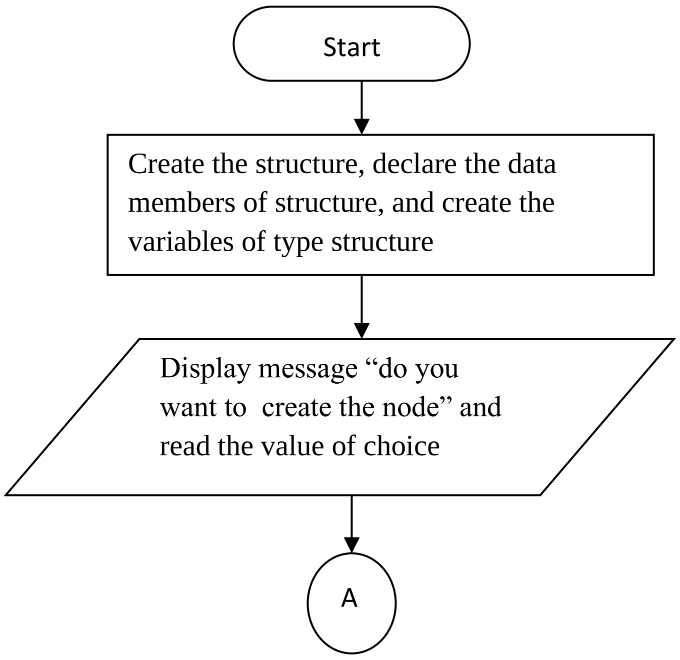
Start=Null.

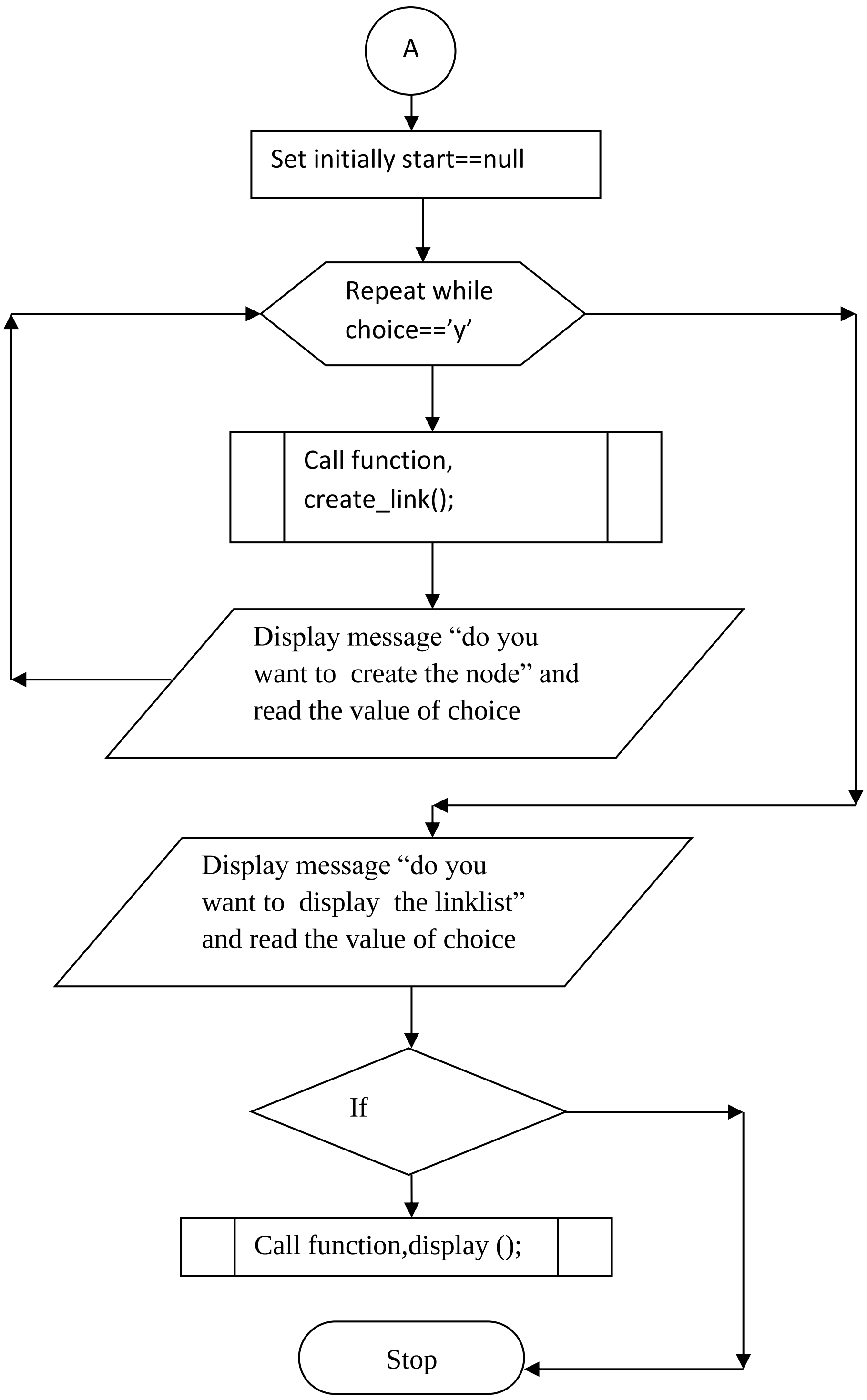
2.[Allocate a space to newly created node]

Node=create node ();

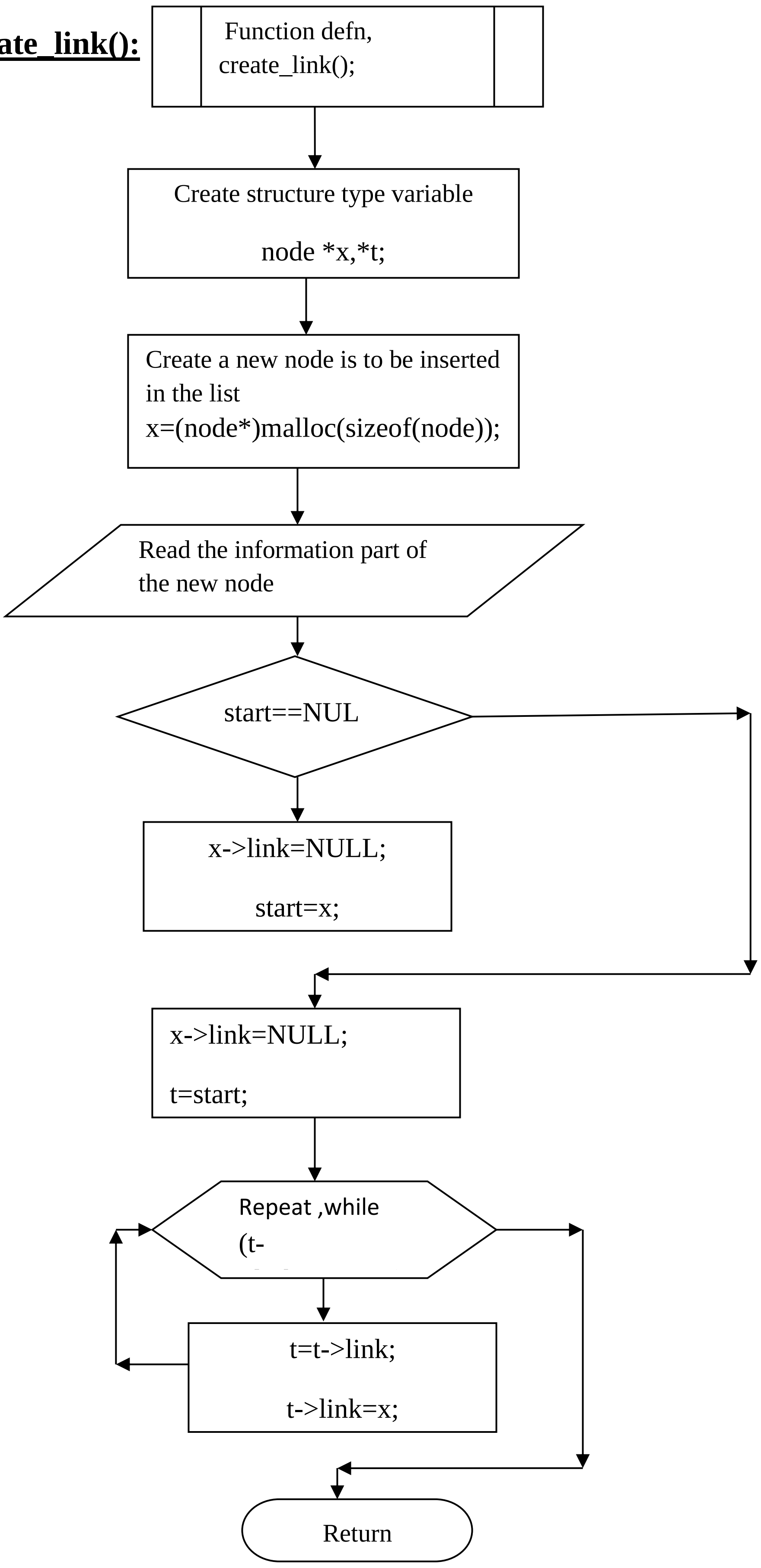
```
3.[Assign value to to information part of node]
Info[node]=data.
4.[Assign null to the address part for signaling end of the list]
next[node]=start
5.[Assign address of first node to start variable]
Start=node
6.[Return the created node]
Return (start)
```

Flowchart:


Start
Create the structure, declare the data members of structure, and create the variables of type structure
Display message “do you want to create the node” and read the value of choice
A

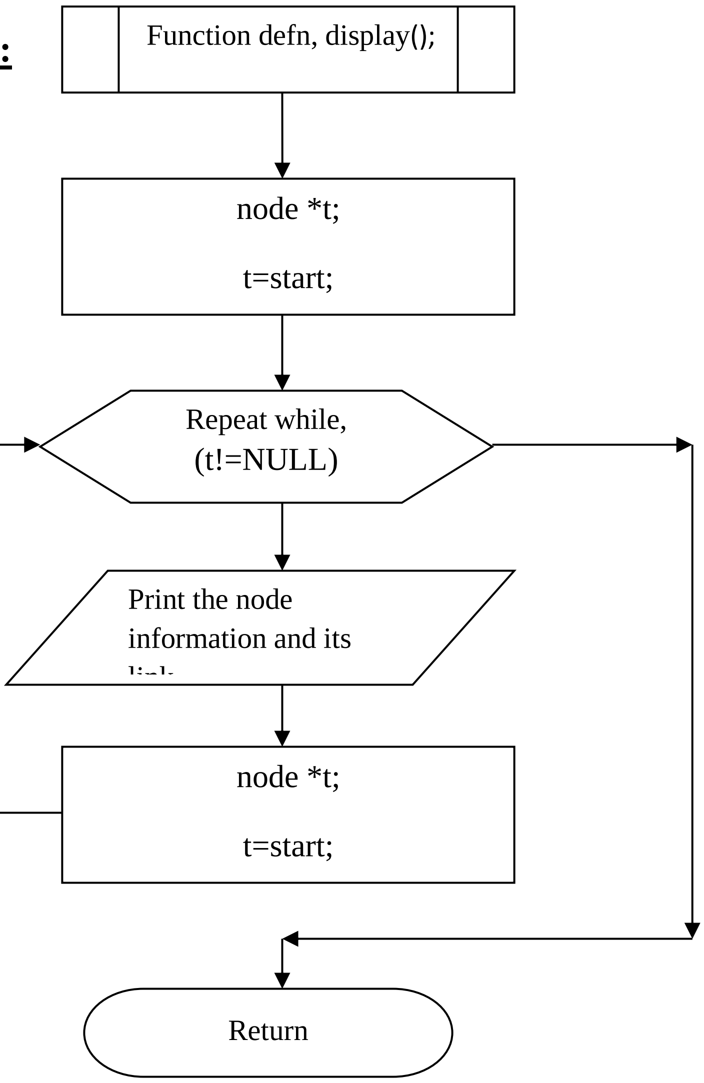

A
Set initially start==null
Repeat while choice=='y'
Call function, create_link();
Display message "do you want to create the node" and read the value of choice
Display message "do you want to display the linklist" and read the value of choice
If
Call function,display ();
Stop

Flowchart for create_link():



Function defn,
create_link();
Create structure type variable
node *x,*t;
Create a new node is to be inserted in the list
x=(node*)malloc(sizeof(node));
Read the information part of the new node
start==NUL
x->link=NULL;
start=x;
x->link=NULL;
t=start;
Repeat ,while
(t-
t=t->link;
t->link=x;
Return

Flowchart for display():



Program:

```
#include<stdio.h>

#include<conio.h>

#include<alloc.h>

typedef struct node

{

            int data;

            struct node*link;

}node;

node * start;
```

```
void create_link();
void display();
void main()
{
        char ch;
        clrscr();
        start=NULL;
        printf("want to create (y/n)");
        scanf("%c",&ch) ;
        while(ch=='y')
                {
                create_link();
                printf("want to create (y/n)");
                ch=getche();
                }
        printf("\n\nwant to display(y/n)");
        ch=getche();
        if(ch=='y')
                {
                display();
                }
        getch();
}
void create_link()
{
        node *x,*t;
        char ch;
```

```
          x=(node*)malloc(sizeof(node));
          printf("\nEnter the data");
          scanf("%d",&(x->data));
          if(start==NULL)
                    {
                    x->link=NULL;
                    start=x;
                    }
          else
                    {
                    x->link=NULL;
                    t=start;
                    while(t->link!=NULL)
                              t=t->link;
                              t->link=x;
                    }
}
void display()
          {
          node *t;
          t=start;
          while(t!=NULL)
                    {
                    printf("\n%d  %d",t->data,t->link);
                    t=t->link;
                    }
          }
```

Output:

```
Want to create (y/n):y

Enter the data:

10

Want to create (y/n):y

Enter the data:

20

Want to create (y/n):y

Enter the data:

30

Want to create (y/n):y

Enter the data:

40

Want to create (y/n):n

Want to display(y/n):y

10  1936

20  1944

30  1952

40  0
```