

TITLE:- Singly Linked List

In singly linked list, each node has a connection with next node.



Algorithm for inserting at the beginning of node in singly linked list:

Let *pHead be the pointer to the firstnode in the current list.

- 1) Create a new node using malloc
`pNode=(Nodetype*)malloc(sizeof(Newtype));`
- 2) Assign a data item to the new node
`pNode->info= a;`
- 3) If pHead=NULL then
`pNode->next=NULL`
goto 6.
- 4) Else
Set the pointer of next address in the field of new node
`pNode->next=pHead;`
- 5) Set the headpointer to new node
`pHead=pNode;`
- 6) End

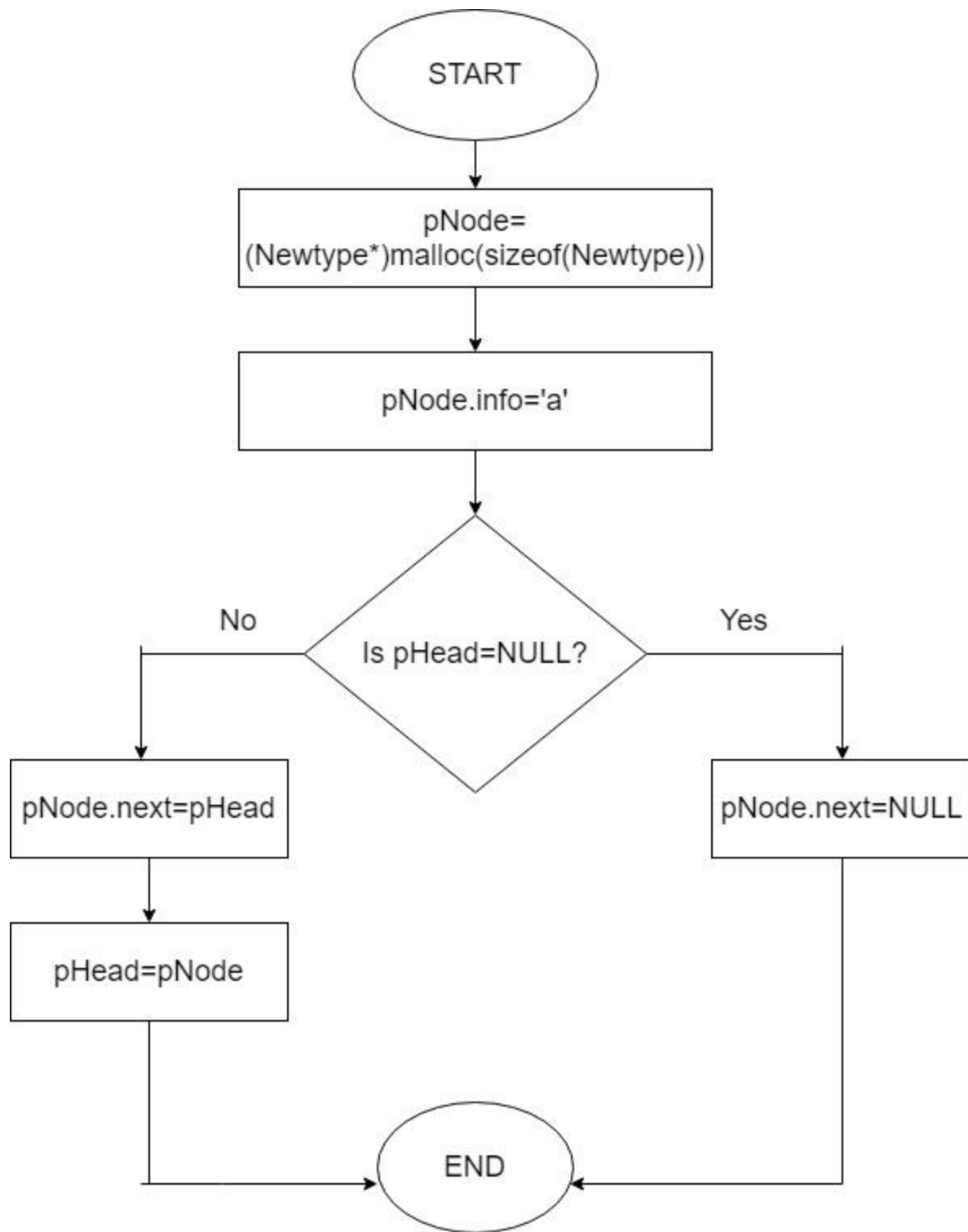


Figure: Flowchart for insert at beginning in singly linked list

Write a program that uses functions to perform the following operations on singly linked list i) Creation ii) Insertion iii) Deletion iv) Traversal.

```
#include<stdio.h>
#include<stdlib.h>
struct Node;
typedef struct Node * PtrToNode;
typedef PtrToNode List;
typedef PtrToNode Position;

struct Node
{
    int e;
    Position next;
};

void Insert(int x, List l, Position p)
{
    Position TmpCell;
    TmpCell = (struct Node*) malloc(sizeof(struct Node));
    if(TmpCell == NULL)
        printf("Memory out of space\n");
    else
    {
        TmpCell->e = x;
        TmpCell->next = p->next;
        p->next = TmpCell;
    }
}

int isLast(Position p)
{
    return (p->next == NULL);
}

Position FindPrevious(int x, List l)
{
    Position p = l;
    while(p->next != NULL && p->next->e != x)
        p = p->next;
    return p;
}

void Delete(int x, List l)
{
    Position p, TmpCell;
    p = FindPrevious(x, l);

    if(!isLast(p))
    {
```

```

        TmpCell = p->next;
        p->next = TmpCell->next;
        free(TmpCell);
    }
    else
        printf("Element does not exist!!!\n");
}

void Display(List l)
{
    printf("The list element are :: ");
    Position p = l->next;
    while(p != NULL)
    {
        printf("%d -> ", p->e);
        p = p->next;
    }
}

void Merge(List l, List l1)
{
    int i, n, x, j;
    Position p;
    printf("Enter the number of elements to be merged :: ");
    scanf("%d",&n);

    for(i = 1; i <= n; i++)
    {
        p = l1;
        scanf("%d", &x);
        for(j = 1; j < i; j++)
            p = p->next;
        Insert(x, l1, p);
    }
    printf("The new List :: ");
    Display(l1);
    printf("The merged List ::");
    p = l;
    while(p->next != NULL)
    {
        p = p->next;
    }
    p->next = l1->next;
    Display(l);
}

int main()
{
    int x, pos, ch, i;
    List l, l1;
    l = (struct Node *) malloc(sizeof(struct Node));
    l->next = NULL;

```

```

List p = l;
printf("LINKED LIST IMPLEMENTATION OF LIST ADT\n\n");
do
{
    printf("\n\n1. INSERT\t2. DELETE\t3. MERGE\t4. PRINT\t5. QUIT\n\nEnter the choice :: ");
    scanf("%d", &ch);
    switch(ch)
    {
        case 1:
            p = l;
            printf("Enter the element to be inserted :: ");
            scanf("%d",&x);
            printf("Enter the position of the element :: ");
            scanf("%d",&pos);

            for(i = 1; i < pos; i++)
            {
                p = p->next;
            }
            Insert(x,l,p);
            break;

        case 2:
            p = l;
            printf("Enter the element to be deleted :: ");
            scanf("%d",&x);
            Delete(x,p);
            break;

        case 3:
            l1 = (struct Node *) malloc(sizeof(struct Node));
            l1->next = NULL;
            Merge(l, l1);
            break;

        case 4:
            Display(l);
            break;
    }
}
while(ch<5);
return 0;
}

```

```

1. INSERT      2. DELETE      3. MERGE      4. PRINT      5. QUIT

Enter the choice :: 1
Enter the element to be inserted :: 5
Enter the position of the element :: 5

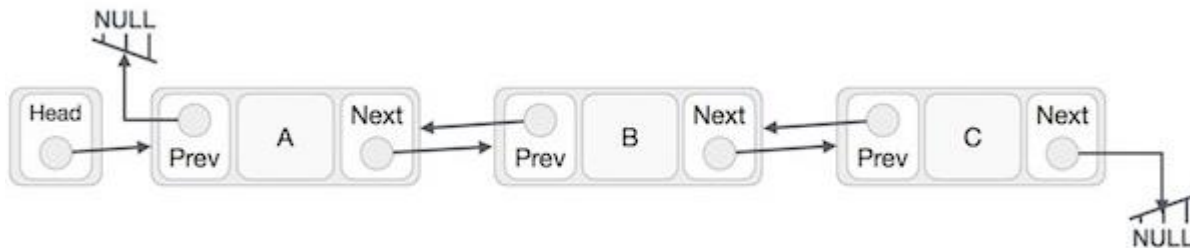
1. INSERT      2. DELETE      3. MERGE      4. PRINT      5. QUIT

Enter the choice :: 4
The list element are :: 1 -> 2 -> 3 -> 4 -> 5 ->

```

TITLE:- Doubly Linked List

Doubly linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence. So, we can traverse forward



by using next field and can traverse backward by using previous field.

Algorithm to insert a item at beginning in doubly linked list:

- 1) Create a new node using malloc
`pNode=(Nodetype*)malloc(sizeof(Newtype));`
- 2) Assign a data item to the new node
`pNode->info= a;`
- 3) If `pHead=NULL` then
`pNode->prev=NULL`
`pNode->next=NULL`
goto 6.
- 4) Else
Set the previous and next address in the field of new node
 - i. `pNode->prev=NULL`
 - ii. `pNode->next=pHead;`
- 5) Set the headpointer to new node
`pHead=pNode;`
- 6) End

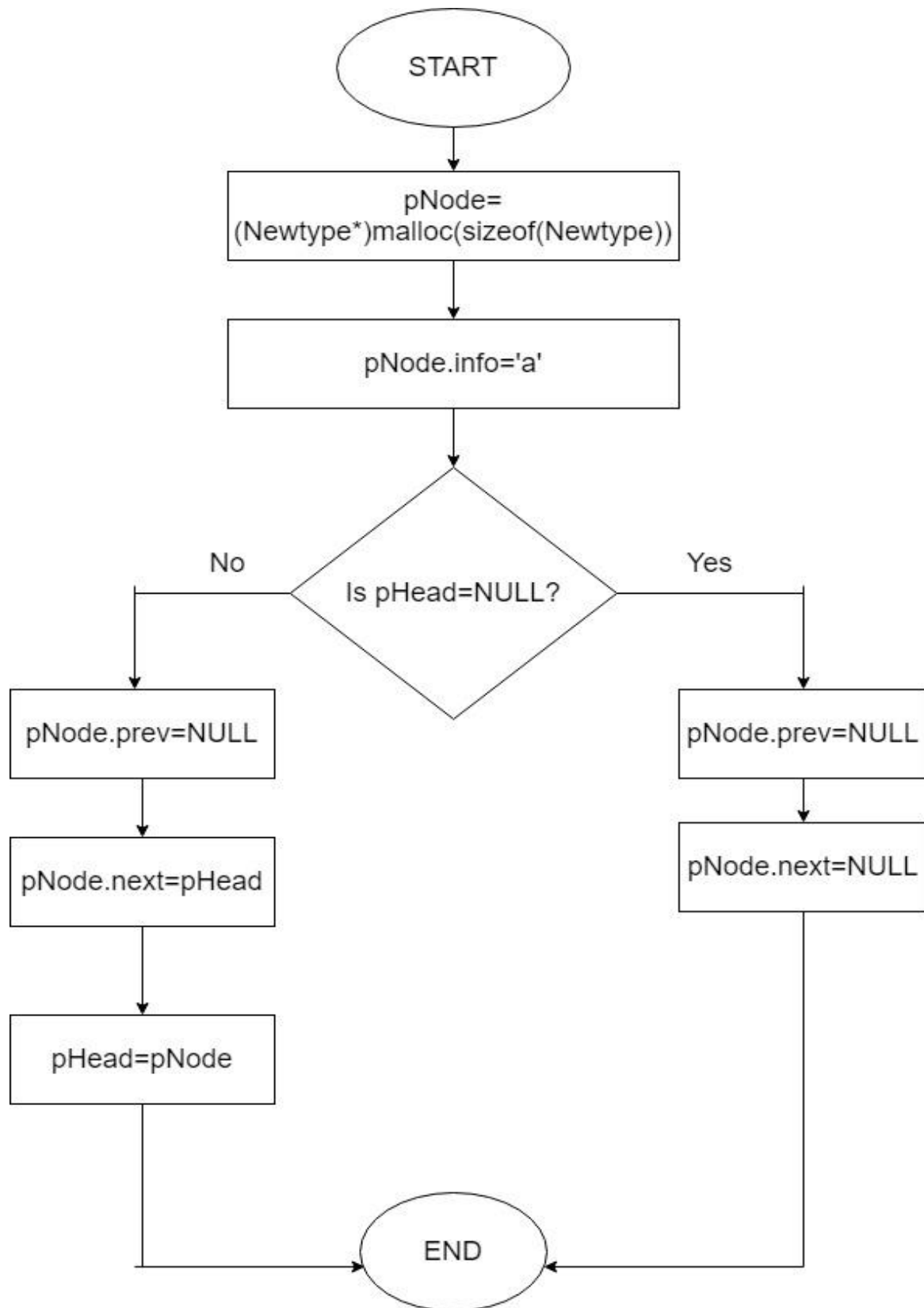


Figure: Flowchart to insert at beginning
in doubly linked list

Write a program that uses functions to perform the following operations on doubly linked list i) Creation ii) Insertion iii) Deletion iv) Traversal.

```
#include<stdio.h>
#include<stdlib.h>
struct node {
    struct node * prev;
    struct node * next;
    int data;
};
struct node * head;
void insertion_beginning();
void insertion_last();
void insertion_specified();
void deletion_beginning();
void deletion_last();
void deletion_specified();
void display();
void search();
main() {
    int choice = 0;
    while (choice != 9) {
        printf("\nChoose one option from the following list ...\n");
        printf("1.Insert in begining\t2.Insert at last\t3.Insert at any random location\n");
        printf("4.Delete from Beginning\t5.Delete from last\t6.Delete the node after the given\n");
        printf("7.Search\n8.Show\n9.Exit\n");
        printf("\nEnter your choice?\n");
        scanf("\n%d", & choice);
        switch (choice) {
            case 1:
                insertion_beginning();
                break;
            case 2:
                insertion_last();
                break;
            case 3:
                insertion_specified();
                break;
            case 4:
                deletion_beginning();
                break;
            case 5:
                deletion_last();
                break;
            case 6:
                deletion_specified();
                break;
            case 7:
                search();
                break;
```



```

    case 8:
        display();
        break;
    case 9:
        exit(0);
        break;
    default:
        printf("Please enter valid choice..");
    }
}
return 0;
}

void insertion_beginning() {
    struct node * ptr;
    int item;
    ptr = (struct node * ) malloc(sizeof(struct node));
    if (ptr == NULL) {
        printf("\nOVERFLOW");
    } else {
        printf("\nEnter Item value");
        scanf("%d", & item);

        if (head == NULL) {
            ptr -> next = NULL;
            ptr -> prev = NULL;
            ptr -> data = item;
            head = ptr;
        } else {
            ptr -> data = item;
            ptr -> prev = NULL;
            ptr -> next = head;
            head -> prev = ptr;
            head = ptr;
        }
        printf("\nNode inserted\n");
    }
}

}

void insertion_last() {
    struct node * ptr, * temp;
    int item;
    ptr = (struct node * ) malloc(sizeof(struct node));
    if (ptr == NULL) {
        printf("\nOVERFLOW");
    } else {
        printf("\nEnter value");
        scanf("%d", & item);
        ptr -> data = item;
        if (head == NULL) {
            ptr -> next = NULL;
            ptr -> prev = NULL;

```

```

    head = ptr;
} else {
    temp = head;
    while (temp -> next != NULL) {
        temp = temp -> next;
    }
    temp -> next = ptr;
    ptr -> prev = temp;
    ptr -> next = NULL;
}

}
printf("\nnode inserted\n");
}

void insertion_specified() {
    struct node * ptr, * temp;
    int item, loc, i;
    ptr = (struct node * ) malloc(sizeof(struct node));
    if (ptr == NULL) {
        printf("\n OVERFLOW");
    } else {
        temp = head;
        printf("Enter the location");
        scanf("%d", & loc);
        for (i = 0; i < loc; i++) {
            temp = temp -> next;
            if (temp == NULL) {
                printf("\n There are less than %d elements", loc);
                return;
            }
        }
        printf("Enter value");
        scanf("%d", & item);
        ptr -> data = item;
        ptr -> next = temp -> next;
        ptr -> prev = temp;
        temp -> next = ptr;
        temp -> next -> prev = ptr;
        printf("\nnode inserted\n");
    }
}

void deletion_beginning() {
    struct node * ptr;
    if (head == NULL) {
        printf("\n UNDERFLOW");
    } else if (head -> next == NULL) {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    } else {
        ptr = head;

```

```

    head = head -> next;
    head -> prev = NULL;
    free(ptr);
    printf("\nnode deleted\n");
}

}

void deletion_last() {
    struct node * ptr;
    if (head == NULL) {
        printf("\n UNDERFLOW");
    } else if (head -> next == NULL) {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    } else {
        ptr = head;
        if (ptr -> next != NULL) {
            ptr = ptr -> next;
        }
        ptr -> prev -> next = NULL;
        free(ptr);
        printf("\nnode deleted\n");
    }
}

void deletion_specified() {
    struct node * ptr, * temp;
    int val;
    printf("\n Enter the data after which the node is to be deleted : ");
    scanf("%d", & val);
    ptr = head;
    while (ptr -> data != val)
        ptr = ptr -> next;
    if (ptr -> next == NULL) {
        printf("\nCan't delete\n");
    } else if (ptr -> next -> next == NULL) {
        ptr -> next = NULL;
    } else {
        temp = ptr -> next;
        ptr -> next = temp -> next;
        temp -> next -> prev = ptr;
        free(temp);
        printf("\nnode deleted\n");
    }
}

void display() {
    struct node * ptr;
    printf("\nThe list:");
    ptr = head;
    while (ptr != NULL) {
        printf("%d->", ptr -> data);
    }
}

```

```

    ptr = ptr -> next;
}
}
void search() {
    struct node * ptr;
    int item, i = 0, flag;
    ptr = head;
    if (ptr == NULL) {
        printf("\nEmpty List\n");
    } else {
        printf("\nEnter item which you want to search?\n");
        scanf("%d", & item);
        while (ptr != NULL) {
            if (ptr -> data == item) {
                printf("\nitem found at location %d ", i + 1);
                flag = 0;
                break;
            } else {
                flag = 1;
            }
            i++;
            ptr = ptr -> next;
        }
        if (flag == 1) {
            printf("\nItem not found\n");
        }
    }
}

```

```

Choose one option from the following list ...
1.Insert in begining    2.Insert at last    3.Insert at any random location
4.Delete from Beginning 5.Delete from last    6.Delete the node after the given data
7.Search
8.Show
9.Exit

Enter your choice?
1

Enter Item value 5

Node inserted

Choose one option from the following list ...
1.Insert in begining    2.Insert at last    3.Insert at any random location
4.Delete from Beginning 5.Delete from last    6.Delete the node after the given data
7.Search
8.Show
9.Exit

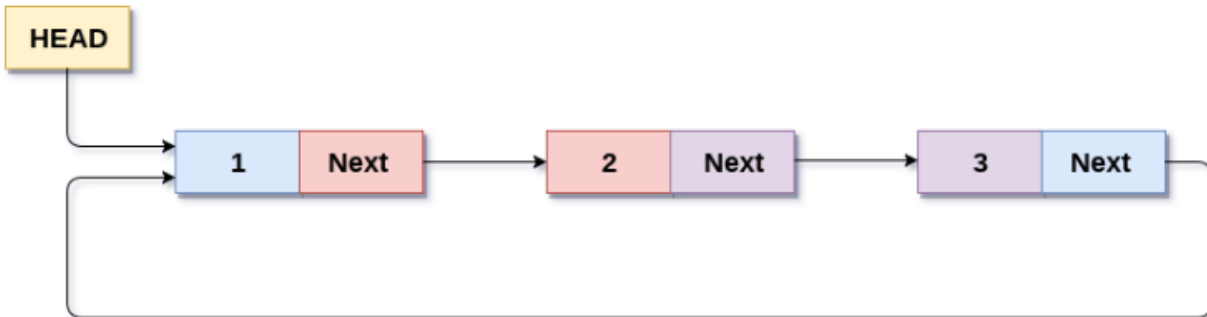
Enter your choice?
8

The list:5->4->3->2->1->

```

TITLE:- Circular Linked List

Circular linked list is a sequence of elements in which every element has link to its next element in the sequence and the last element has a link to the first element in the sequence. Circular linked lists can be used to help the traverse the same list again and again if needed. A circular list is very similar to the linear list where in the circular list the pointer of the last node points not NULL but the first node.



Algorithm to insert a item at the end of the circular linked list:

Let pHead be the pointer pointer to the headnode, qNode be the head node and tail be the tail node in the current list.

- 1) Create a new node using malloc
`pNode=(Nodetype*)malloc(sizeof(Newtype));`
- 2) Assign a data item to the new node
`pNode->info= a;`
- 3) If pHead=NULL then
`pNode->next=NULL`
`print("No elements in list so new element will be both head and tail node.")`
`goto 6.`
- 4) Else
Set the pointer of next address in the field of new node
`pNode->next=qNode;`
- 5) Set the next address of tail as address of new node
`tail->next=pNode;`
- 6) End

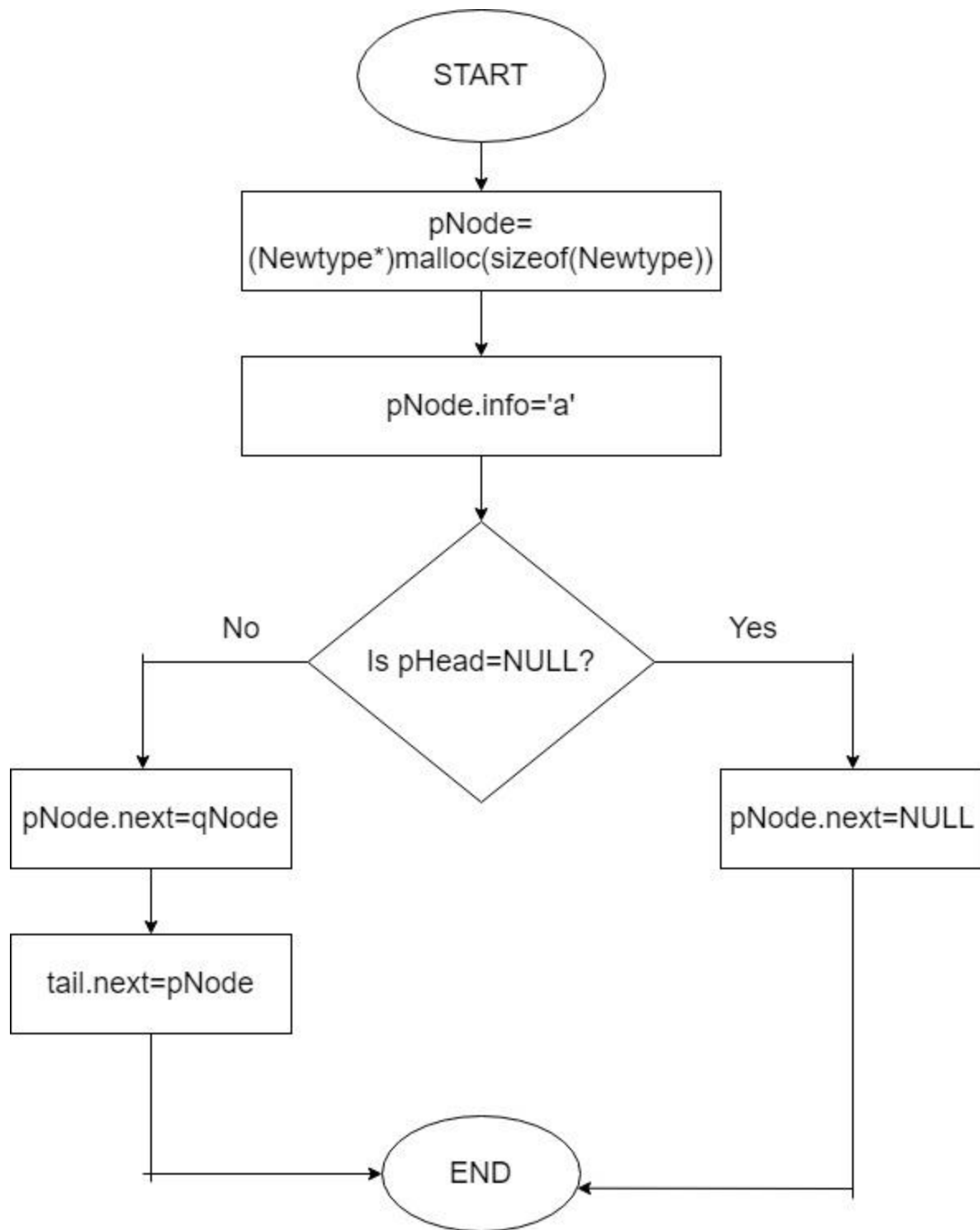


Figure: Flowchart to insert at tail in circular linked list

Write a program that uses functions to perform the following operations on circular linked List i) Creation ii) Insertion iii) Deletion iv) Traversal.

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

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

```
struct Node;
typedef struct Node * PtrToNode;
typedef PtrToNode List;
typedef PtrToNode Position;
```

```
struct Node {
    int e;
    Position next;
};
```

```
void Insert(int x, List l, Position p) {
    Position TmpCell;
    TmpCell = (struct Node * ) malloc(sizeof(struct Node));
    if (TmpCell == NULL)
        printf("Memory out of space\n");
    else {
        TmpCell -> e = x;
        TmpCell -> next = p -> next;
        p -> next = TmpCell;
    }
}
```

```
int isLast(Position p, List l) {
    return (p -> next == l);
}
```

```
Position FindPrevious(int x, List l) {
    Position p = l;
    while (p -> next != l && p -> next -> e != x)
        p = p -> next;
    return p;
}
```

```
Position Find(int x, List l) {
    Position p = l -> next;
    while (p != l && p -> e != x)
        p = p -> next;
    return p;
}
```

```
void Delete(int x, List l) {
```

```

Position p, TmpCell;
p = FindPrevious(x, l);
if (!isLast(p, l)) {
    TmpCell = p -> next;
    p -> next = TmpCell -> next;
    free(TmpCell);
} else
    printf("Element does not exist!!!\n");
}

```

```

void Display(List l) {
    printf("The list element are :: ");
    Position p = l -> next;
    while (p != l) {
        printf("%d -> ", p -> e);
        p = p -> next;
    }
}

```

```

int main() {
    int x, pos, ch, i;
    List l, l1;
    l = (struct Node * ) malloc(sizeof(struct Node));
    l -> next = l;
    List p = l;
    printf("CIRCULAR LINKED LIST IMPLEMENTATION OF LIST ADT\n\n");
    do {
        printf("\n\n1. INSERT\t2. DELETE\t3. FIND\t4. PRINT\t5. QUIT\n\nEnter the choice :: ");
        scanf("%d", & ch);
        switch (ch) {
            case 1:
                p = l;
                printf("Enter the element to be inserted :: ");
                scanf("%d", & x);
                printf("Enter the position of the element :: ");
                scanf("%d", & pos);
                for (i = 1; i < pos; i++) {
                    p = p -> next;
                }
                Insert(x, l, p);
                break;

            case 2:
                p = l;
                printf("Enter the element to be deleted :: ");
                scanf("%d", & x);
                Delete(x, p);
                break;

```



```

case 3:
    p = l;
    printf("Enter the element to be searched :: ");
    scanf("%d", & x);
    p = Find(x, p);
    if (p == l)
        printf("Element does not exist!!!\n");
    else
        printf("Element exist!!!\n");
    break;

case 4:
    Display(l);
    break;
}
} while (ch < 5);
return 0;
}

```

```

1. INSERT      2. DELETE      3. FIND      4. PRINT      5. QUIT

Enter the choice :: 1
Enter the element to be inserted :: 5
Enter the position of the element :: 5

1. INSERT      2. DELETE      3. FIND      4. PRINT      5. QUIT

Enter the choice :: 4
The list element are :: 1 -> 2 -> 3 -> 4 -> 5 ->

```

TITLE: SORTING

Sorting refers to ordering data in an increasing or decreasing manner according to some linear relationship among the data items. There are various types of sorting used. Some of them are:

Bubble Sorting: Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

Algorithm for Bubble sort:

Input: n, list[n]

begin bubblesort(n,list)

 for all elements of list

 if list[i]>list[i+1] then

 swap (list[i],list[i+1])

 end if

 end for

 return list

end bubblesort

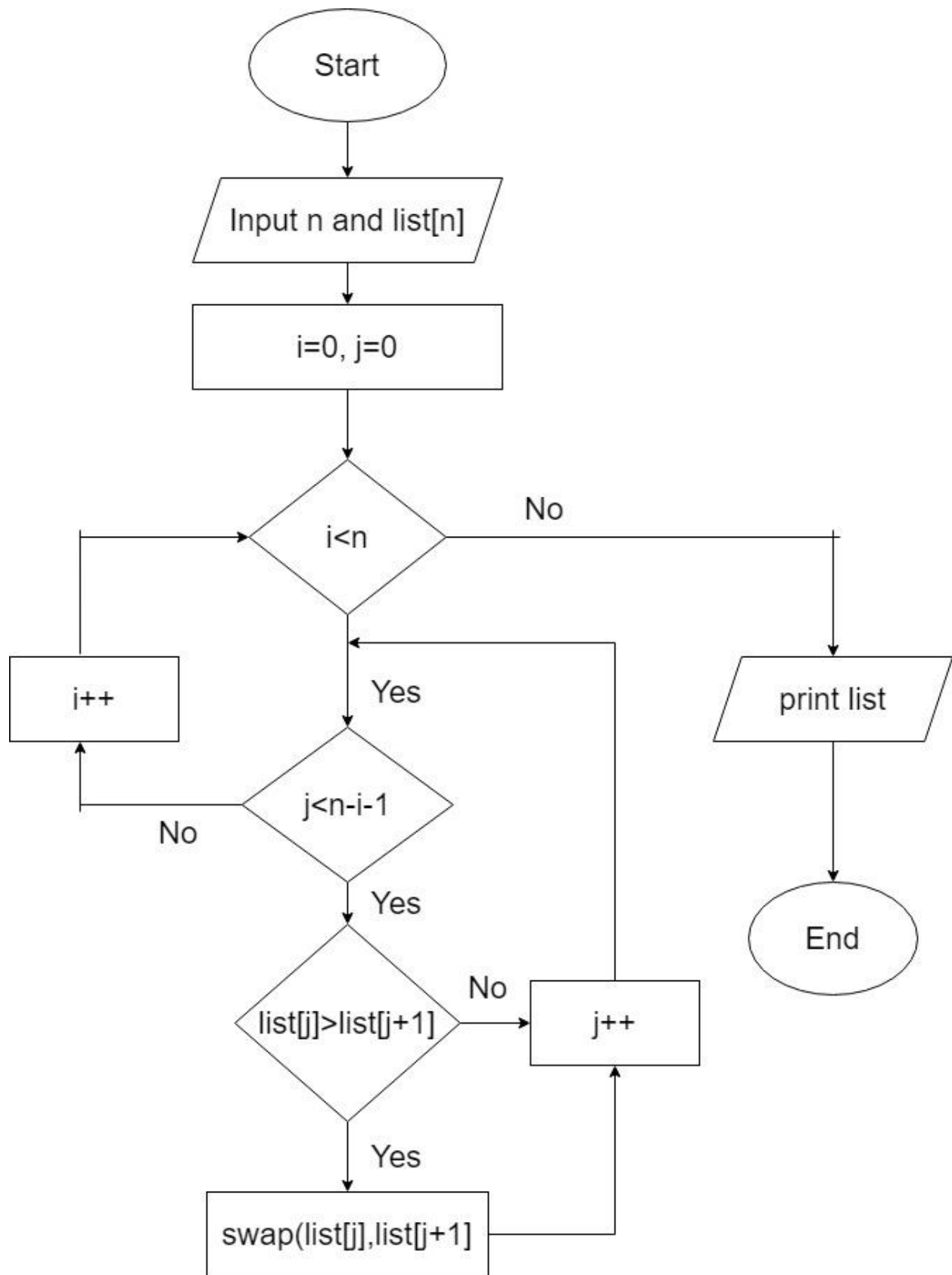


Figure: Bubble Sort

WAP to implement bubble sort.

```
#include <stdio.h>
// perform the bubble sort
void bubbleSort(int array[], int size) {

    // loop to access each array element
    for (int step = 0; step < size - 1; ++step) {

        // loop to compare array elements
        for (int i = 0; i < size - step - 1; ++i) {

            // compare two adjacent elements
            // change > to < to sort in descending order
            if (array[i] > array[i + 1]) {

                // swapping occurs if elements
                // are not in the intended order
                int temp = array[i];
                array[i] = array[i + 1];
                array[i + 1] = temp;
            }
        }
    }
}

void printArray(int array[], int size) {
    for (int i = 0; i < size; ++i) {
        printf("%d ", array[i]);
    }
    printf("\n");
}

int main() {
    int data[] = {
        9,
        45,
        8,
        5,
        80
    };

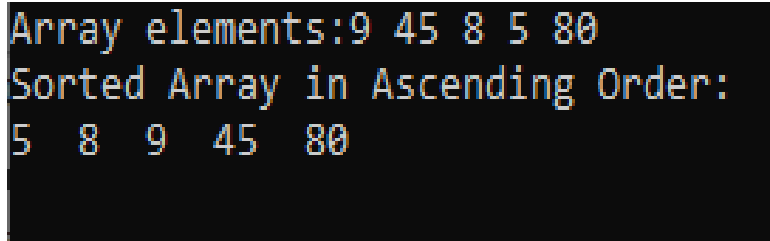
    // find the array's length
    int size = sizeof(data) / sizeof(data[0]);
    printf("Array elements:");
    for (int i = 0; i < size; i++) {
        printf("%d ", data[i]);
    }
}
```

```

bubbleSort(data, size);

printf("\nSorted Array in Ascending Order:\n");
printArray(data, size);
return 0;
}

```



```

Array elements:9 45 8 5 80
Sorted Array in Ascending Order:
5 8 9 45 80

```

Selection Sort: The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

Algorithm for selection sort:

- 1) Set MIN location to 0
- 2) Search the minimum value in the list
- 3) Swap the minimum value with location MIN
- 4) Increment MIN to point to next element
- 5) Repeat until the list is sorted

Pseudocode:

Input: n,list[n]

begin selectionsort(n,list[n])

for i=0 to n-1 do

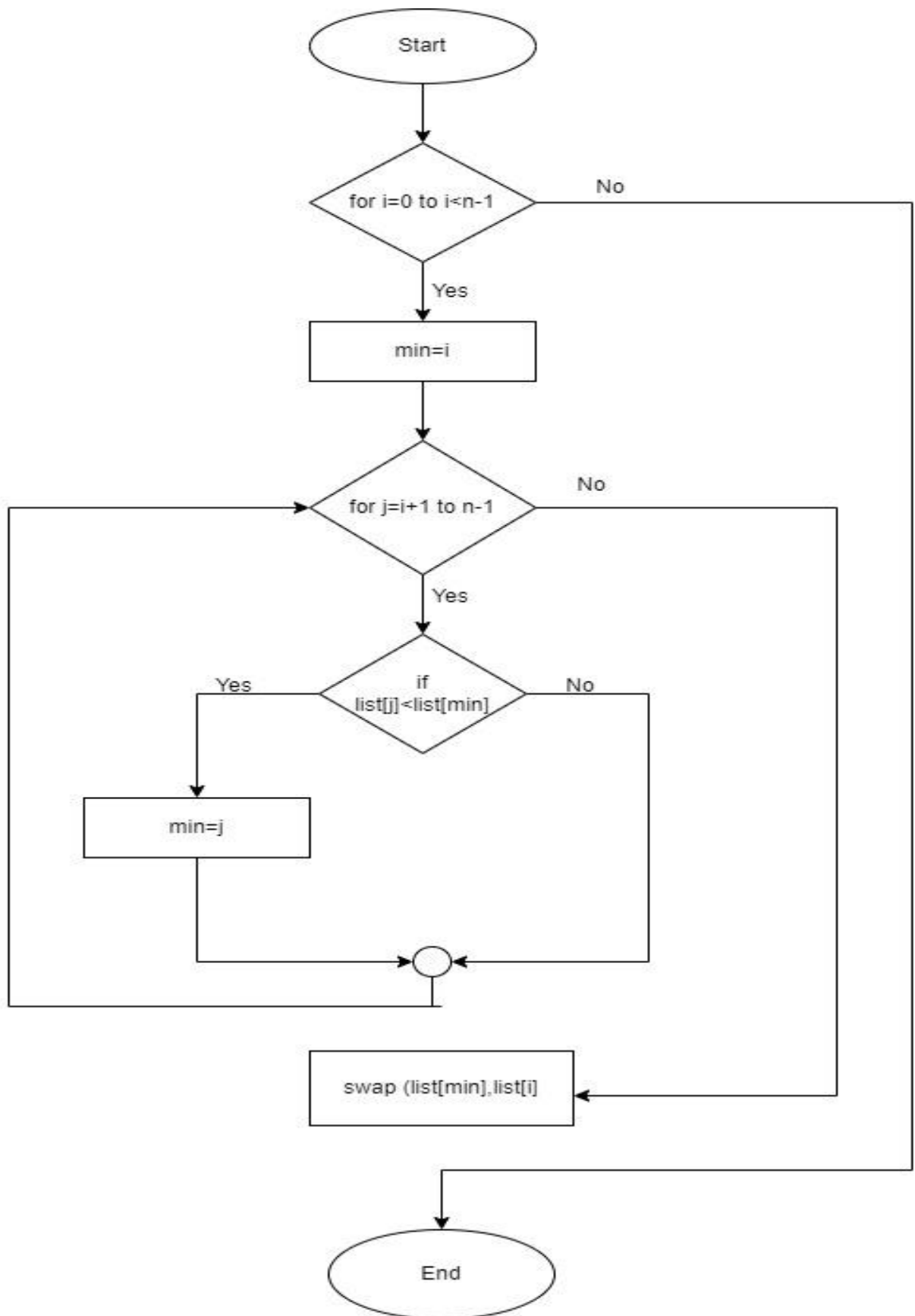
min=i

for j=i+1 to n-1 do

if list[j]<list[min] then

min=j

```
        end if
    end for
    if indexMin!=l then
        swap (list[min],list[i])
    end if
end for
end selectionsort
```



WAP to implement selection sort.

```
#include <stdio.h>

void swap(int * a, int * b) {
    int temp = * a;
    * a = * b;
    * b = temp;
}

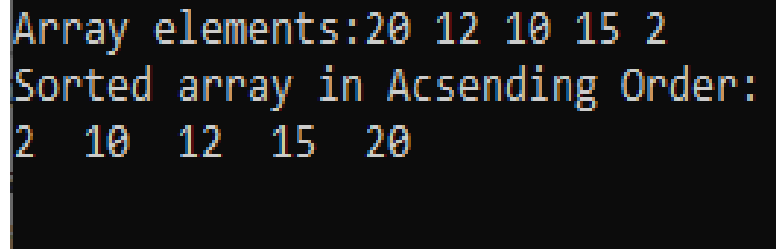
void selectionSort(int array[], int size) {
    for (int step = 0; step < size - 1; step++) {
        int min_idx = step;
        for (int i = step + 1; i < size; i++) {

            // To sort in descending order, change > to < in this line.
            // Select the minimum element in each loop.
            if (array[i] < array[min_idx])
                min_idx = i;
        }

        // put min at the correct position
        swap( & array[min_idx], & array[step]);
    }
}

void printArray(int array[], int size) {
    for (int i = 0; i < size; ++i) {
        printf("%d ", array[i]);
    }
    printf("\n");
}

int main() {
    int data[] = { 20, 12, 10, 15, 2 };
    int size = sizeof(data) / sizeof(data[0]);
    printf("Array elements:");
    for (int i = 0; i < size; i++) {
        printf("%d ", data[i]);
    }
    selectionSort(data, size);
    printf("\nSorted array in Ascending Order:\n");
    printArray(data, size);
    return 0;
}
```



```
Array elements:20 12 10 15 2
Sorted array in Ascending Order:
2 10 12 15 20
```


Insertion sort: Insertion sort is an in-place sorting algorithm. It uses no auxiliary data structures while sorting. Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration. Insertion sort works similarly as we sort cards in our hand in a card game. We assume that the first card is already sorted then, we select an unsorted card.

Algorithm for insertion sort:

- 1) If the element is the first element, assume that it is already sorted. Return 1.
- 2) Pick the next element, and store it separately in a key.
- 3) Now, compare the key with all elements in the sorted array.
- 4) If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.
- 5) Insert the value.
- 6) Repeat until the array is sorted.

Pseudocode:

for i=1 to n do

 key=list[i]

 j=i-1

 while j>0 and list[j]>key do

 list[j+1]=list[j]

 j=j-1

 end while

 list[j+1]=key

end for

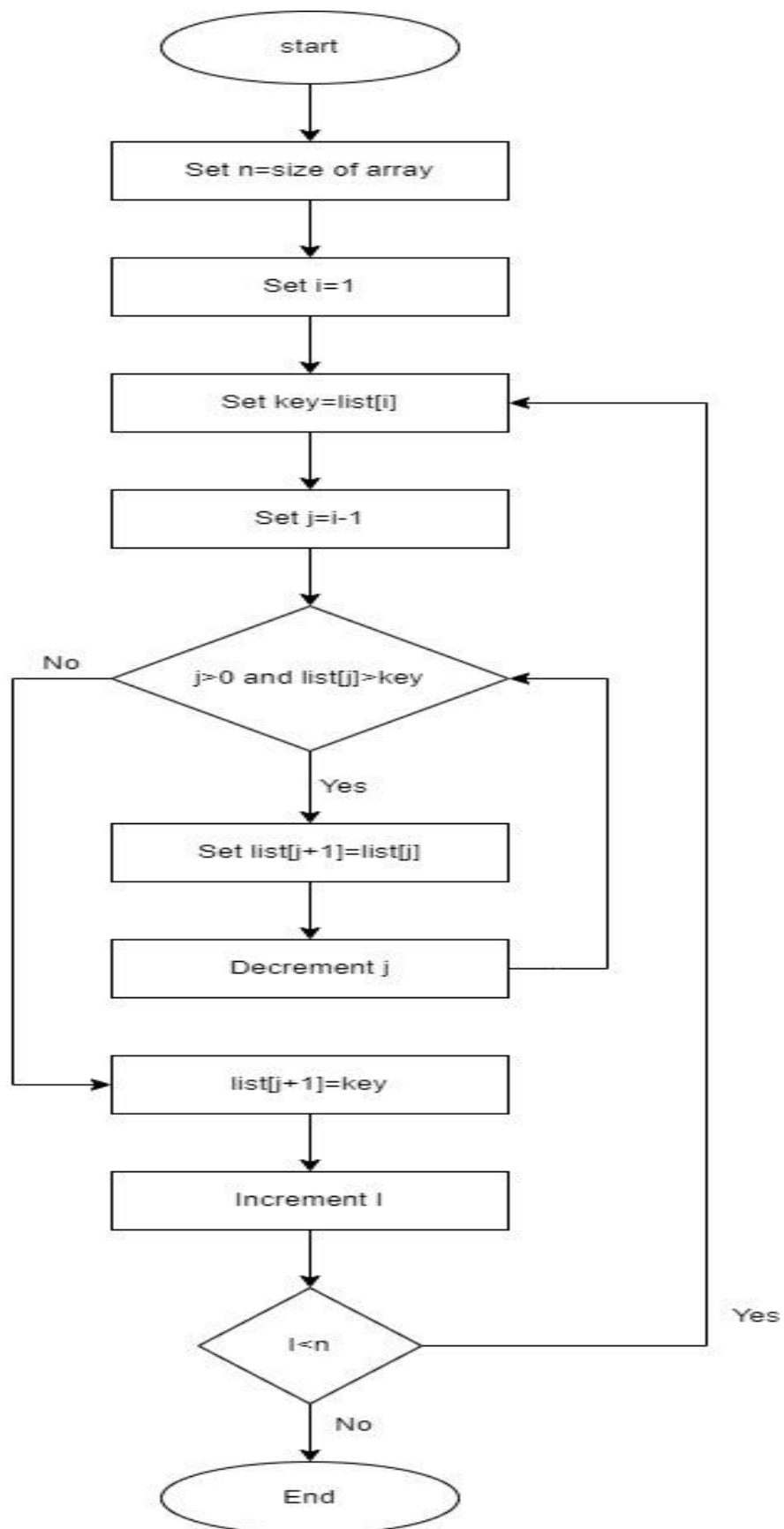


Figure: Insertion sort

WAP to implement insertion sort.

```
#include <stdio.h>

void printArray(int array[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
}

void insertionSort(int array[], int size) {
    for (int step = 1; step < size; step++) {
        int key = array[step];
        int j = step - 1;

        // Compare key with each element on the left of it until an element smaller than
        // it is found.
        // For descending order, change key<array[j] to key>array[j].
        while (key < array[j] && j >= 0) {
            array[j + 1] = array[j];
            --j;
        }
        array[j + 1] = key;
    }
}

int main() {
    int data[] = {9, 5, 1, 4, 3};
    int size = sizeof(data) / sizeof(data[0]);
    printf("Array elements:");
    for (int i = 0; i < size; i++) {
        printf("%d ", data[i]);
    }
    insertionSort(data, size);
    printf("\nSorted array in ascending order:\n");
    printArray(data, size);
    return 0;
}
```

```
Array elements:9 5 1 4 3
Sorted array in ascending order:
1 3 4 5 9
-----
```