**TITLE:- Singly Linked List**

A singly linked list is a particular kind of linked list that can only be traversed in one direction, from head to final node (tail). A node is the name given to each entry in a linked list. A single node keeps the list's structure by including data and a pointer to the 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));

1. Assign a data item to the new node

pNode->info= a;

1. If pHead=NULL then

pNode->next=NULL

goto 6.

1. Else

Set the pointer of next address in the field of new node

pNode->next=pHead;

1. Set the headpointer to new node

pHead=pNode;

1. End

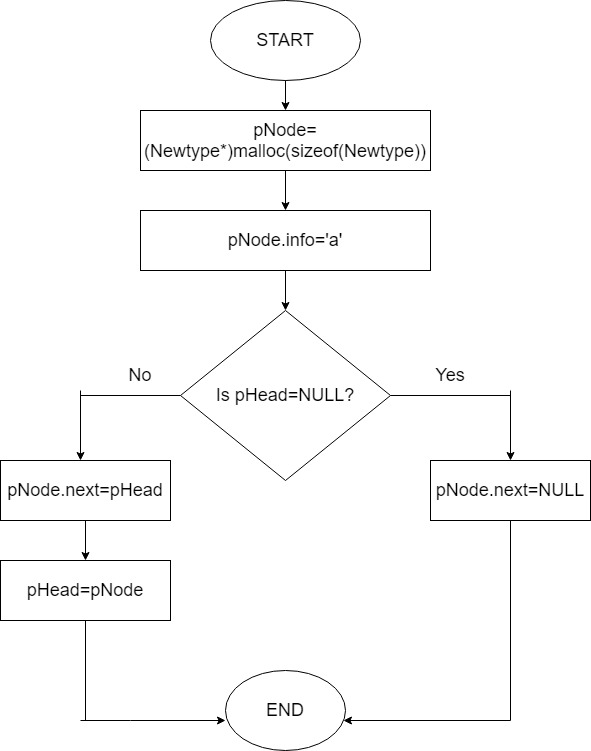


Figure: Flowchart for insert at beginning in singly linked list

**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{

    int data;

    struct Node \*next;

};

typedef struct Node NodeType;

NodeType \*head = NULL;

NodeType \*getnode(){

    NodeType \*p;

    p=(NodeType\*)malloc(sizeof(NodeType));

*return*(p);

}

void display(){

*// printf("linked list elements:\n");*

*if* (head == NULL){

        printf("Linked list is empty\n");

    }

*else*{

        NodeType \*ptr;

        ptr = head;

        printf("[ ");

*while* (ptr != NULL){

*if* (ptr->next == NULL){

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

            }*else*{

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

            }

            ptr = ptr->next;

        }

        printf(" ]\n");

    }

}

void insertEnd(int *x*){

    NodeType \*p;

    p = (NodeType\*)malloc(sizeof(NodeType));

    p->data = *x*;

    p->next = NULL;

*if* (head == NULL){

        head = p;

    }*else*{

        NodeType \*ptr = head;

*while* (ptr->next!=NULL){

            ptr = ptr->next;

        }

        ptr->next = p;

    }

}

void insertBeg(int *x*){

    NodeType \*ptr = getnode();

    ptr->data = *x*;

    ptr->next = head;

    head = ptr;

}

void insert(int *x*,int *index*){

*if* (head == NULL){

        printf("Empty List: index out of range");

*return*;

    }

*if* (*index* ==0){

        insertBeg(*x*);

*return*;

    }

    NodeType \*prev = getnode();

    NodeType \*ptr = head;

*// ptr = head;*

*for* (int i=0;i<*index*;i++){

        prev = ptr;

        ptr = ptr->next;

    }

    NodeType \*p =getnode();

    p->data = *x*;

    p->next = ptr;

    prev->next = p;

}

void deleteEnd(){

*if* (head==NULL){

        printf("Cannot delete. List is already empty\n");

*return*;

    }

    NodeType \*prev = getnode();

    NodeType \*ptr = head;

*while* (ptr->next!=NULL){

        prev = ptr;

        ptr = ptr->next;

    }

    prev->next = NULL;

    free(ptr);

}

void deleteBeg(){

*if* (head==NULL){

        printf("Cannot delete. List is already empty\n");

*return*;

    }

    NodeType \*ptr = head;

    head = ptr->next;

    free(ptr);

}

void del(int *index*){

*if* (head==NULL){

        printf("Cannot delete. List is already empty\n");

*return*;

    }

*if* (*index* == 0){

        deleteBeg;

*return*;

    }

    NodeType \*ptr = head;

    NodeType \*prev;

*for* (int i=0;i<*index*;i++){

        prev = ptr;

        ptr = ptr->next;

    }

    prev->next = ptr->next;

    free(ptr);

}

void destroy(){

*while* (head!=NULL){

        NodeType \*toDel = head;

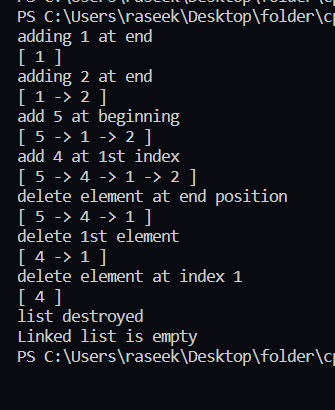
        head = head->next;

        free(toDel);

    }

    printf("list destroyed\n");

}

int main(){

    printf("adding 1 at end\n");

    insertEnd(1);

    display();

    printf("adding 2 at end\n");

    insertEnd(2);

    display();

    printf("add 5 at beginning\n");

    insertBeg(5);

    display();

    printf("add 4 at 1st index\n");

    insert(4,1);

    display();

    printf("delete element at end position\n");

    deleteEnd();

    display();

    printf("delete 1st element\n");

    deleteBeg();

    display();

    printf("delete element at index 1\n");

    del(1);

    display();

    destroy(); *// destroy list*

    display();

}

**Conclusion:**

­

**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));

1. Assign a data item to the new node

pNode->info= a;

1. If pHead=NULL then

pNode->prev=NULL

pNode->next=NULL

goto 6.

1. Else

Set the previous and next address in the field of new node

* + 1. pNode->prev=NULL
    2. pNode->next=pHead;

1. Set the headpointer to new node

pHead=pNode;

1. End

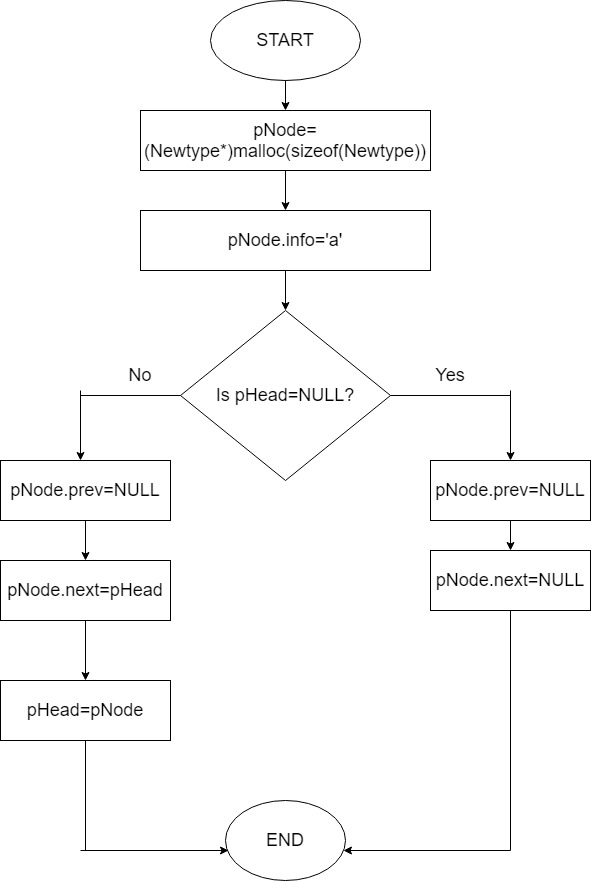


Figure: Flowchart to insert at beginning in doubly linked list

**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 theNode{

    int data;

    struct theNode \*prev;

    struct theNode \*next;

};

typedef struct theNode Node;

Node \*head = NULL;

Node \*tail = NULL;

Node \*getNode(){

    Node \*n1;

    n1 = (Node\*)malloc(sizeof(Node));

*return* n1;

}

void insertBeg(int *x*){

    Node \*ptr = getNode();

    ptr->data = *x*;

    ptr->next = head;

    ptr->prev = NULL;

*if* (head==NULL){

        tail = ptr;

    }

*else*{

        head->prev = ptr;

    }

    head = ptr;

}

void insertEnd(int *x*){

*if* (head == NULL){

        insertBeg(*x*);

*return*;

    }

    Node \*newNode = getNode();

    newNode->data = *x*;

    newNode->next = NULL;

    newNode->prev = tail;

    tail->next = newNode;

    tail = newNode;

}

void display(){

*if* (head == NULL){

        printf("Empty list. Nothing to display\n");

    }

    Node \*ptr = head;

    printf("\nForward Traverse: ");

*while*(ptr!=NULL){

        printf("%d ",ptr->data);

        ptr = ptr->next;

    }

    printf("\nBackward Traverse: ");

    ptr = tail;

*while* (ptr != NULL){

        printf("%d ",ptr->data);

        ptr = ptr->prev;

    }

}

void insert(int *x*,int *index*){

*if* (head == NULL){

        printf("Empty List: index out of range");

*return*;

    }

*if* (*index* == 0){

        insertBeg(*x*);

*return*;

    }

    Node \*currentNode = head;

*for* (int i=0;i<*index*;i++){

        currentNode = currentNode->next;

    }

*if* (currentNode == NULL){

        printf("Error: index out of range\n");

        exit(0);

    }

    Node \*newNode = getNode();

    newNode->data = *x*;

    newNode->next = currentNode;

    newNode->prev = currentNode->prev;

    currentNode->prev->next = newNode;

    currentNode->prev = newNode;

}

void deleteEnd(){

*if* (head==NULL){

        printf("Cannot delete. List is already empty\n");

*return*;

    }

    Node \*currentNode = head;

*while* (currentNode->next!=NULL){

        currentNode = currentNode->next;

    }

    tail = currentNode->prev;

    currentNode->prev->next = NULL;

    free(currentNode);

}

int main(){

    printf("add 5,6,3 to end and 110 to index 1\n");

    insertEnd(5);

    insertEnd(6);

    insertEnd(3);

    insert(110,1);

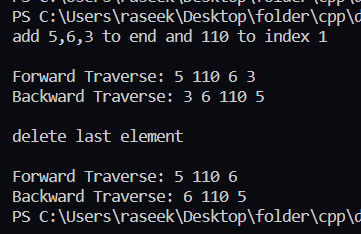
    display();

    printf("\n\ndelete last element\n");

    deleteEnd();

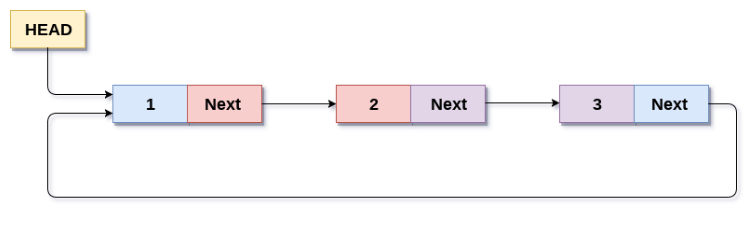
    display();

}

****

**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));

1. Assign a data item to the new node

pNode->info= a;

1. If pHead=NULL then

pNode->next=NULL

print(“No elements in list so new element will be both head and tail node.”)

goto 6.

1. Else

Set the pointer of next address in the field of new node

pNode->next=qNode;

1. Set the next address of tail as address of new node

tail->next=pNode;

1. End

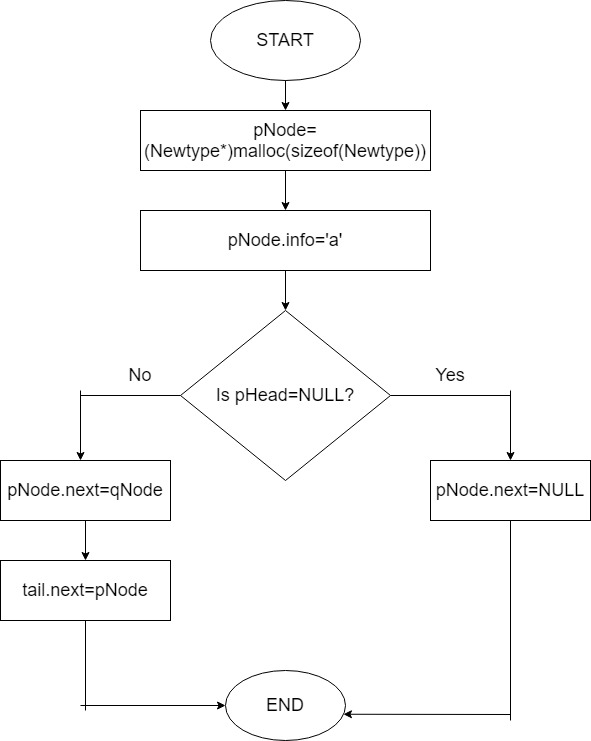


Figure: Flowchart to insert at tail in circular linked list

**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\t 2. DELETE\t 3. FIND\t 4. PRINT\t 5. 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);

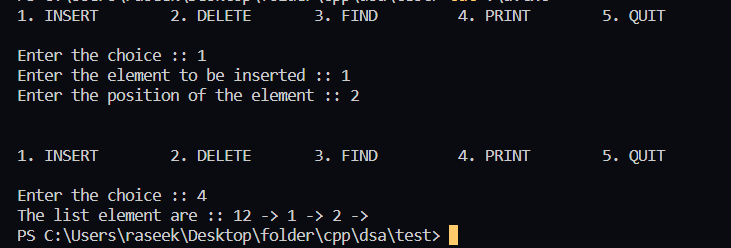
break;

}

} while (ch < 5);

return 0;

}



**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](https://www.geeksforgeeks.org/sorting-algorithms/) 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

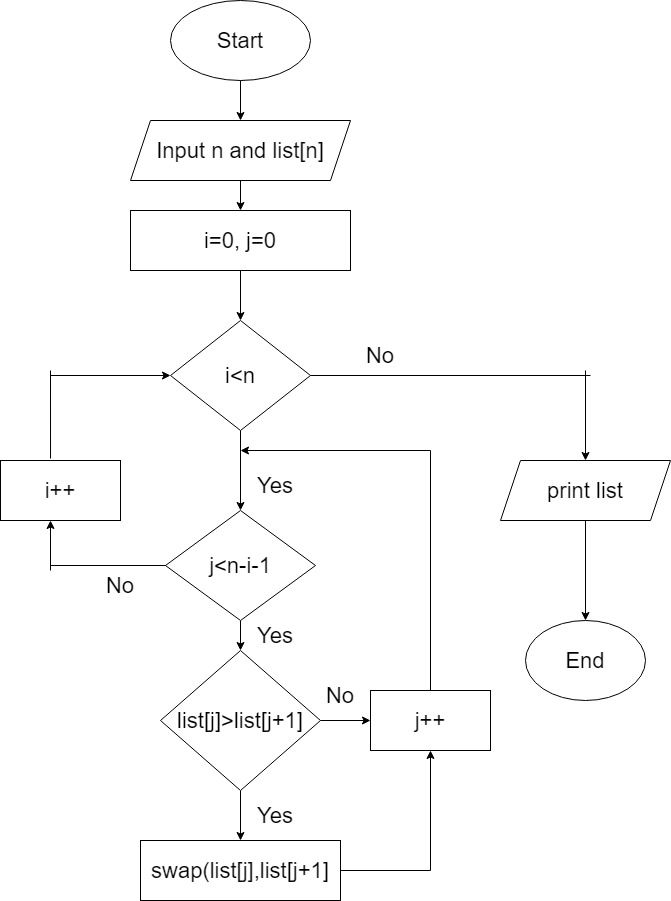
****

Figure: Bubble Sort

**WAP to implement bubble sort.**

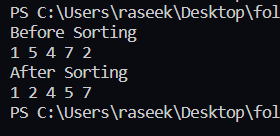
*#include* <stdio.h>

int linearSearch(int *arr*[],int *size*,int *searchFor*){

*for* (int i=0;i<size;i++){

*if* (arr[i] == searchFor){

*return* i;

        }

    }

*return* -1;

}

int main(){

    int arr[] = {1,3,5,7,9,6,8,3,2,6};

    int size = sizeof(arr) / sizeof(int);

    int searchFor = 2;

    int indexof = linearSearch(arr,size,searchFor);

*for* (int i=0;i<size;i++){

        printf("%d ",arr[i]);

    }

    printf("\n");

*if* (indexof >=0){

        printf("found at index %d",indexof);

    }*else*{

        printf("not found");

    }

}

**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!=I then

swap (list[min],list[i])

end if

end for

end selectionsort

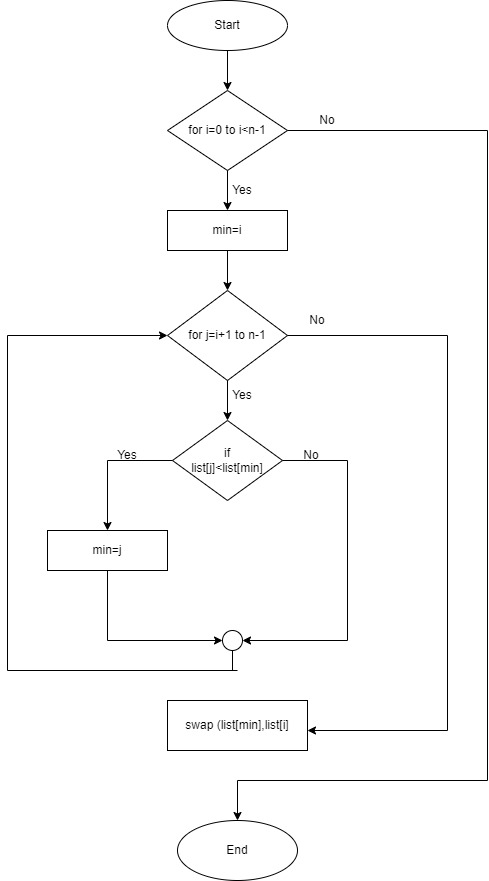


Figure: Selection sort

**WAP to implement selection sort.**

*#include*<stdio.h>

void printarr(int \**arr*,int *size*){

*for* (int i=0;i<*size*;i++){

        printf("%d ",*arr*[i]);

    }

}

int main(){

    int size=10,min=0,temp;

    int arr[size] = {23,10,20,3,45,76,67,24,2,11};

    printf("Bofore sorting:\n");

    printarr(&arr[0],size);

*for* (int i=0;i<size-1;i++){

        min = i;

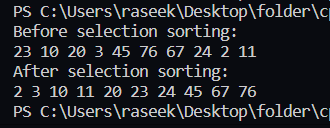
*for* (int j=i+1;j<size;j++){

*if* (arr[j] < arr[min]){

                min = j;

            }

        }

        *if* (min!=i){

            temp = arr[min];

            arr[min] = arr[i];

            arr[i] = temp;

        }

    }

    printf("\nAfter sorting:\n");

    printarr(&arr[0],size);

}

**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

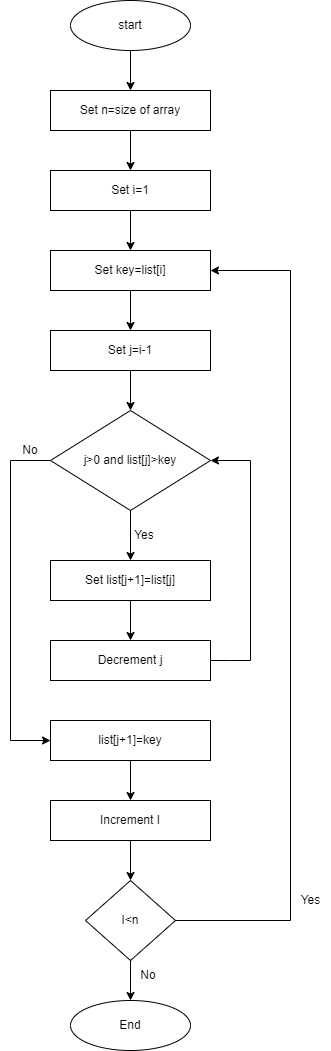


Figure: Insertion sort

**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;

*while* (key < *array*[j] && j >= 0) {

*array*[j + 1] = *array*[j];

      --j;

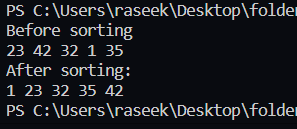
    }

*array*[j + 1] = key;

  }

}

int main() {

  int data[] = {23,42,32,1,35};

  int size = sizeof(data) / sizeof(data[0]);

  printf("Before Insertion sorting\n");

*for*(int i=0;i<size;i++){

    printf("%d ",data[i]);

  }

  insertionSort(data, size);

  printf("\nAfter Insertion sorting:\n");

  printArray(data, size);

*return* 0;

}