**I BUBBLE SORT**

SOURCE CODE:

#include <stdio.h>

void bubbleSort(int array[], int size) {

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

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

if (array[i] > array[i + 1]) {

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[] = {-2, 45, 0, 11, -9};

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

bubbleSort(data, size);

printf("Sorted Array in Ascending Order:\n");

printArray(data, size);

}

OUTPUT:

Sorted Array in Ascending Order:

-9 -2 0 11 45

**SELECTION SORT**

SOURCE CODE:

#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++) {

if (array[i] < array[min\_idx])

min\_idx = i;

}

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]);

selectionSort(data, size);

printf("Sorted array in Acsending Order:\n");

printArray(data, size);

}

OUPUT:

Sorted array in Acsending Order:

2 10 12 15 20

**INSERTION SORT**

SOURCE CODE:

#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[] = {9, 5, 1, 4, 3};

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

insertionSort(data, size);

printf("Sorted array in ascending order:\n");

printArray(data, size);

}

OUTPUT:

Sorted array in ascending order:

1 3 4 5 9

**QUICK SORT**

SOURCE CODE:

#include <stdio.h>

void swap(int \*a, int \*b) {

int t = \*a;

\*a = \*b;

\*b = t;

}

int partition(int array[], int low, int high) {

int pivot = array[high];

int i = (low - 1);

for (int j = low; j < high; j++) {

if (array[j] <= pivot) {

i++;

swap(&array[i], &array[j]);

}

}

swap(&array[i + 1], &array[high]);

return (i + 1);

}

void quickSort(int array[], int low, int high) {

if (low < high) {

int pi = partition(array, low, high);

quickSort(array, low, pi - 1);

quickSort(array, pi + 1, high);

}

}

void printArray(int array[], int size) {

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

printf("%d ", array[i]);

}

printf("\n");

}

int main() {

int data[] = {8, 7, 2, 1, 0, 9, 6};

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

printf("Unsorted Array\n");

printArray(data, n);

quickSort(data, 0, n - 1);

printf("Sorted array in ascending order: \n");

printArray(data, n);

}

OUTPUT:

Unsorted Array

8 7 2 1 0 9 6

Sorted array in ascending order:

0 1 2 6 7 8 9

**MERGE SORT**

SOURCE CODE:

#include <stdio.h>

void merge(int arr[], int p, int q, int r) {

// Create L ← A[p..q] and M ← A[q+1..r]

int n1 = q - p + 1;

int n2 = r - q;

int L[n1], M[n2];

for (int i = 0; i < n1; i++)

L[i] = arr[p + i];

for (int j = 0; j < n2; j++)

M[j] = arr[q + 1 + j];

// Maintain current index of sub-arrays and main array

int i, j, k;

i = 0;

j = 0;

k = p;

// Until we reach either end of either L or M, pick larger among

// elements L and M and place them in the correct position at A[p..r]

while (i < n1 && j < n2) {

if (L[i] <= M[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = M[j];

j++;

}

k++;

}

// When we run out of elements in either L or M,

// pick up the remaining elements and put in A[p..r]

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = M[j];

j++;

k++;

}

}

// Divide the array into two subarrays, sort them and merge them

void mergeSort(int arr[], int l, int r) {

if (l < r) {

// m is the point where the array is divided into two subarrays

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

// Merge the sorted subarrays

merge(arr, l, m, r);

}

}

// Print the array

void printArray(int arr[], int size) {

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

printf("%d ", arr[i]);

printf("\n");

}

// Driver program

int main() {

int arr[] = {6, 5, 12, 10, 9, 1};

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

mergeSort(arr, 0, size - 1);

printf("Sorted array: \n");

printArray(arr, size);

}

OUTPUT:

Sorted array:

1 5 6 9 10 12

**LINKED LIST:**

**INSERTION AT BEGINNING**

SOURCE CODE:

#include<stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node \*next;

};

void insertStart (struct Node \*\*head, int data)

{

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

newNode->data = data;

newNode->next = \*head;

//changing the new head to this freshly entered node

\*head = newNode;

}

void display (struct Node \*node)

{

//as linked list will end when Node is Null

while (node != NULL)

{

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

node = node->next;

}

printf ("\n");

}

int main ()

{

//creating 4 pointers of type struct Node

//So these can point to address of struct type variable

struct Node \*head = NULL;

struct Node \*node2 = NULL;

struct Node \*node3 = NULL;

struct Node \*node4 = NULL;

// allocate 3 nodes in the heap

head = (struct Node \*) malloc (sizeof (struct Node));

node2 = (struct Node \*) malloc (sizeof (struct Node));

node3 = (struct Node \*) malloc (sizeof (struct Node));

node4 = (struct Node \*) malloc (sizeof (struct Node));

head->data = 15; // data set for head node

head->next = node2; // next pointer assigned to address of node2

node2->data = 10;

node2->next = node3;

node3->data = 12;

node3->next = node4;

node4->data = 3;

node4->next = NULL;

printf ("Linklist : ");

display (head);

// Need '&' i.e. address as we need to change head

insertStart (&head, 25);

printf ("\nAfter Inserting Element\n");

printf ("\nLinklist : ");

// no need for '&' as head need not be changed

// only doing traversal

display (head);

return 0;

}

OUTPUT:

Linklist : 15 10 12 3

After Inserting Element

Linklist : 25 15 10 12 3

INSERTION AT ENDING

SOURCE CODE:

#include<stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node \*next;

};

void insertStart (struct Node \*\*head, int data)

{

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

newNode->data = data;

newNode->next = \*head;

//changing the new head to this freshly entered node

\*head = newNode;

}

void insertLast (struct Node \*\*head, int data)

{

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

newNode->data = data;

newNode->next = NULL;

//need this if there is no node present in linked list at all

if (\*head == NULL)

{

\*head = newNode;

return;

}

struct Node \*temp = \*head;

while (temp->next != NULL)

temp = temp->next;

temp->next = newNode;

}

void display (struct Node \*node)

{

//as linked list will end when Node is Null

while (node != NULL)

{

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

node = node->next;

}

printf ("\n");

}

int main ()

{

struct Node \*head = NULL;

// Need '&' i.e. address as we need to change head

insertStart (&head, 12);

insertStart (&head, 16);

insertStart (&head, 20);

insertLast (&head, 10);

insertLast (&head, 14);

insertLast (&head, 18);

insertLast (&head, 11);

// no need for '&' as head need not be changed

// only doing traversal

display (head);

return 0;

}

OUTPUT:

20 16 12 10 14 18 11

INSERTION AT MIDDLE:

SOURCE CODE:

#include<stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node \*next;

};

int calcSize (struct Node \*node)

{

int size = 0;

while (node != NULL)

{

node = node->next;

size++;

}

return size;

}

void insertPosition (int pos, int data, struct Node \*\*head)

{

int size = calcSize (\*head);

//If pos is 0 then should use insertStart method

//If pos is less than 0 then can't enter at all

//If pos is greater than size then bufferbound issue

if (pos < 1 || size < pos)

{

printf ("Can't insert, %d is not a valid position\n", pos);

}

else

{

struct Node \*temp = \*head;

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

newNode->data = data;

newNode->next = NULL;

while (--pos)

{

temp = temp->next;

}

//(25)->next = 10 as 12->next is 10

newNode->next = temp->next;

// (12)->next = 25

temp->next = newNode;

//new node added in b/w 12 and 10

}

}

void insertStart (struct Node \*\*head, int data)

{

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

newNode->data = data;

newNode->next = \*head;

//changing the new head to this freshly entered node

\*head = newNode;

}

void insertLast (struct Node \*\*head, int data)

{

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

newNode->data = data;

newNode->next = NULL;

//need this if there is no node present in linked list at all

if (\*head == NULL)

{

\*head = newNode;

return;

}

struct Node \*temp = \*head;

while (temp->next != NULL)

temp = temp->next;

temp->next = newNode;

}

void display (struct Node \*node)

{

//as linked list will end when Node is Null

while (node != NULL)

{

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

node = node->next;

}

printf ("\n");

}

int main ()

{

struct Node \*head = NULL;

// Need '&' i.e. address as we need to change head

insertStart (&head, 12);

insertStart (&head, 16);

insertStart (&head, 20);

insertLast (&head, 10);

insertLast (&head, 14);

insertLast (&head, 18);

insertLast (&head, 11);

//Inserts after 3rd position

insertPosition (3, 25, &head);

// no need for '&' as head need not be changed

// only doing traversal

display (head);

return 0;

}

OUTPUT:

20 16 12 25 10 14 18 11

DELETION AT BEGINNING, MIDDLE, END

SOURCE CODE:

#include<stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node \*next;

};

void delete (struct Node \*\*head, int delVal)

{

struct Node \*temp = \*head;

struct Node \*previous;

//Case when there is only 1 node in the list

if (temp->next == NULL)

{

\*head = NULL;

free (temp);

printf ("Value %d, deleted \n", delVal);

return;

}

//if the head node itself needs to be deleted

if (temp != NULL && temp->data == delVal)

{

//Case 1 head becomes 30

\*head = temp->next; //changing head to next in the list

printf ("Value %d, deleted \n", delVal);

//case 1: 22 deleted and freed

free (temp);

return;

}

//run until we find the value to be deleted in the list

while (temp != NULL && temp->data != delVal)

{

//store previous link node as we need to change its next val

previous = temp;

temp = temp->next;

}

//if value is not present then

//temp will have traversed to last node NULL

if (temp == NULL)

{

printf ("Value not found\n");

return;

}

// Case 2: (24)->next = 16 (as 20->next = 16)

// Case 3: (16)->next = NULL (as 12->next = NULL)

previous->next = temp->next;

free (temp);

//case 2: 20 deleted and freed

//case 3: 12 deleted and freed

printf ("Value %d, deleted \n", delVal);

}

void display (struct Node \*node)

{

//as linked list will end when Node is Null

while (node != NULL)

{

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

node = node->next;

}

printf ("\n");

}

int main ()

{

//creating 4 pointers of type struct Node

//So these can point to address of struct type variable

struct Node \*head = NULL;

struct Node \*node2 = NULL;

struct Node \*node3 = NULL;

struct Node \*node4 = NULL;

struct Node \*node5 = NULL;

struct Node \*node6 = NULL;

// allocate 3 nodes in the heap

head = (struct Node \*) malloc (sizeof (struct Node));

node2 = (struct Node \*) malloc (sizeof (struct Node));

node3 = (struct Node \*) malloc (sizeof (struct Node));

node4 = (struct Node \*) malloc (sizeof (struct Node));

node5 = (struct Node \*) malloc (sizeof (struct Node));

node6 = (struct Node \*) malloc (sizeof (struct Node));

head->data = 22; // data set for head node

head->next = node2; // next pointer assigned to address of node2

node2->data = 30;

node2->next = node3;

node3->data = 24;

node3->next = node4;

node4->data = 20;

node4->next = node5;

node5->data = 16;

node5->next = node6;

node6->data = 12;

node6->next = NULL;

/\*No need for & i.e. address as we do not

need to change head address

\*/

printf ("Linked List Before Operations : ");

display (head);

//deleting first occurance of a value in linked list

delete (&head, 22);

delete (&head, 20);

delete (&head, 12);

printf ("Linked List After Operations : ");

display (head);

return 0;

}

OUTPUT:

Linked List Before Operations : 22 30 24 20 16 12

Value 22, deleted

Value 20, deleted

Value 12, deleted

Linked List After Operations : 30 24 16

**REVERSING A LINKED LIST**

SOURCE CODE:

#include <stdio.h>

struct Node {

   int data;

   struct Node\* next;

};

Node\* insertNode(int key) {

   Node\* temp = new Node;

   temp->data = key;

   temp->next = NULL;

   return temp;

}

void tailRecRevese(Node\* current, Node\* previous, Node\*\* head){

   if (!current->next) {

      \*head = current;

      current->next = previous;

      return;

   }

   Node\* next = current->next;

   current->next = previous;

   tailRecRevese(next, current, head);

}

void tailRecReveseLL(Node\*\* head){

   if (!head)

      return;

   tailRecRevese(\*head, NULL, head);

}

void printLinkedList(Node\* head){

   while (head != NULL) {

      printf("%d ", head->data);

      head = head->next;

   }

   printf("  
");

}

int main(){

   Node\* head1 = insertNode(9);

   head1->next = insertNode(32);

   head1->next->next = insertNode(65);

   head1->next->next->next = insertNode(10);

   head1->next->next->next->next = insertNode(85);

   printf("Linked list : \t");

   printLinkedList(head1);

   tailRecReveseLL(&head1);

   printf("Reversed linked list : \t");

   printLinkedList(head1);

   return 0;

}

OUTPUT:

Linked list : 9 32 65 10 85

Reversed linked list : 85 10 65 32 9

**DISPLAY THE ELEMENTS OF LINKED LIST**

SOURCE CODE:

#include <stdio.h>

#include <stdlib.h>

struct Node

{

int data;

struct Node \*next;

} \*first = NULL;

void create(int A[], int n)

{

int i;

struct Node \*t, \*last;

first = (struct Node \*) malloc(sizeof(struct Node));

first->data = A[0];

first->next = NULL;

last = first;

for (i = 1; i < n; i++)

{

t = (struct Node \*) malloc(sizeof(struct Node));

t->data = A[i];

t->next = NULL;

last->next = t;

last = t;

}

}

void Display(struct Node \*p)

{

while (p != NULL)

{

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

p = p->next;

}

}

int main()

{

struct Node \*temp;

int A[] = { 3, 5, 7, 10, 25, 8, 32, 2 };

create(A, 8);

Display (first);

return 0;

}

OUTPUT: 3 5 7 10 25 8 32 2