Create Array:

void create\_array(int\* arr,int\* n)

{

int i,j,temp;

printf("Enter size of an array\n");

scanf("%d",n);

printf("Enter array elements\n");

for(i=0; i<\*n; i++)

{

scanf("%d",&arr[i]);

}

printf("Given array is\n");

display\_array(arr,\*n);

}

Display Array:

void display\_array(int\* arr,int size)

{

int i;

printf("[\t");

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

{

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

}

printf("]");

printf("\n");

}

Bubble Sort:

void bubble\_sort(int\* arr,int size)

{

int i,j,temp;

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

{

for(j=0; j<size-i; j++)

{

if(arr[j]>arr[j+1])

{

temp = arr[j];

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

arr[j+1] = temp;

}

}

}

}

Selection Sort:

void selection\_sort(int\* arr, int size)

{

int i, j, temp;

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

{

for(j=i+1; j<size; j++)

{

if(arr[i]>arr[j])

{

temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

}

}

Insertion Sort:

void insertion\_sort(int\* arr,int size)

{

int temp,empty,i;

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

{

empty=i;

temp = arr[i];

while(empty>0 && arr[empty-1]>temp)

{

arr[empty]=arr[empty-1];

empty--;

}

arr[empty]=temp;

}

}

QuickSort:

void quick\_sort(int\* arr,int L,int H)

{

int low,high,pivot,i,temp;

pivot = arr[L];

low = L+1;

high = H;

while(low<=high)

{

while(arr[low]<pivot)

{

low++;

}

while(arr[high]>pivot)

{

high--;

}

if(low<high)

{

temp=arr[low];

arr[low]=arr[high];

arr[high]=temp;

high--;

low++;

}

}

temp = arr[L];

arr[L] = arr[high];

arr[high] = temp;

if(L<high)

{

quick\_sort(arr,L,high-1);

}

if(low<H)

{

quick\_sort(arr,low,H);

}

}

Merge:

void merge(int\* arr, int low, int mid, int high)

{

int i = low, j = mid +1, k = 0, brr[100];

while(i<=mid && j<=high)

{

if(arr[i]<arr[j])

{

brr[k] = arr[i];

k++;

i++;

}

else

{

brr[k] = arr[j];

k++;

j++;

}

}

while(i<=mid)

{

brr[k] = arr[i];

i++;

k++;

}

while(j<=high)

{

brr[k] = arr[j];

k++;

j++;

}

k=0;

for(i=low; i<=high; i++)

{

arr[i] = brr[k];

k++;

}

}

MergeSort:

void merge\_sort(int\* arr, int low, int high)

{

int mid;

if(low < high)

{

mid = (low + high)/2;

merge\_sort(arr, low, mid);

merge\_sort(arr, mid+1, high);

merge(arr, low, mid, high);

}

}

Binary Search:

int binary\_search(int\* arr, int low, int high, int search)

{

int mid;

if(low<=high)

{

mid = (low + high)/2;

if(search == arr[mid])

{

return mid;

}

else if(search < arr[mid])

{

return binary\_search(arr,low,mid-1,search);

}

else if(search > arr[mid])

{

return binary\_search(arr,mid+1,high,search);

}

}

else return -1;

}

Create LinkedList:

struct node

{

int data;

struct node\* next;

};

struct node\* createnode()

{

int value;

struct node\* newnode = NULL;

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

if(newnode==NULL)

{

printf("Memory not allocated\n");

}

else

{

printf("Enter data value\n");

scanf("%d",&value);

newnode->data = value;

newnode->next = NULL;

}

}

void createLinkedList(struct node\*\* head)

{

struct node\* tempnode = NULL;

struct node\* travnode = NULL;

tempnode = createnode();

if(\*head == NULL)

{

\*head = tempnode;

}

else

{

travnode = \*head;

while(travnode->next!=NULL)

{

travnode=travnode->next;

}

travnode->next = tempnode;

}

}

Display LinkedList:

void displayLinkedList(struct node\* head)

{

if((head)==NULL)

{

printf("Linked List is Empty\n");

}

else

{

while(head!=NULL)

{

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

head = head->next;

}

printf(" NULL\n\n");

}

}

Sort LinkedList:

void sortLinkedList(struct node\* head)

{

int i,j,n;

int temp = 0;

struct node\* trav1=head;

struct node\* trav2=head;

n = countnode(head);

if(head==NULL)

{

printf("Linked List is empry\n");

}

else

{

while(trav1->next!=NULL)

{

trav2 = trav1->next;

while(trav2!=NULL)

{

if(trav1->data>=trav2->data)

{

temp=trav1->data;

trav1->data=trav2->data;

trav2->data=temp;

}

trav2=trav2->next;

}

trav1=trav1->next;

}

}

}

Create Stack:

struct Stack

{

int capacity;

char\* arr;

int top;

};

void create\_stack(struct Stack\*\* stack)

{

(\*stack) = (struct Stack\*) malloc(sizeof(struct Stack));

printf("Enter capacity of stack\n");

scanf("%d",&((\*stack)->capacity));

(\*stack)->arr=(char\*)malloc(((\*stack)->capacity)\*sizeof(char));

(\*stack)->top = -1;

}

Push Stack:

void push(struct Stack\* stack,char element)

{

if(isFull(stack))

{

printf("Stack is Full\n");

}

else

{

(stack->top)++;

stack->arr[stack->top]=element;

}

}

Pop Stack:

void pop(struct Stack\* stack)

{

if(isEmpty)

{

printf("Stack is empty\n");

}

else

{

printf("Popped element is %d\n",stack->arr[stack->top]);

(stack->top)--;

}

}

Create Queue:

struct Que

{

int capacity;

int\* arr;

int front;

int rear;

};

void create\_que(struct Que\*\* que)

{

(\*que) = (struct Que\*) malloc(sizeof(struct Que));

printf("Enter capacity of Qeue\n");

scanf("%d",&((\*que)->capacity));

(\*que)->arr=(int\*)malloc(((\*que)->capacity)\*sizeof(int));

(\*que)->front = -1;

(\*que)->rear = -1;

}

Enqueue :

void enqueue(struct Que\* que,int element)

{

if(isFull(que))

{

printf("Que is Full\n");

}

else

{

(que->rear)++;

que->arr[que->rear]=element;

}

}

Dequeue:

void dequeue(struct Que\* que)

{

if(isEmpty)

{

printf("Que is empty\n");

que->rear = -1;

que->front = -1;

}

else

{

(que->front)++;

printf("Deleted element is %d\n",que->arr[que->front]);

}

}

Create Binary Search Tree:

void create\_BST(struct node\*\* root,int data)

{

struct node\* tempnode = createnode(data);

struct node\* temp = NULL;

if(tempnode==NULL)

{

printf("Memory not allocated\n");

}

else

{

if((\*root)==NULL)

{

(\*root)=tempnode;

}

else

{

temp = \*root;

while(temp!=NULL)

{

if(((temp)->data)>tempnode->data)

{

if((temp)->left == NULL)

{

(temp)->left = tempnode;

break;

}

else

{

temp = temp->left;

}

}

else

{

if(temp->right == NULL)

{

temp->right = tempnode;

break;

}

else

{

temp = temp->right;

}

}

}

}

}

}

Tree Traversal:

void inorder(struct node\* root) //LDR

{

if(root==NULL)

{

return;

}

inorder(root->left);

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

inorder(root->right);

}

void preorder(struct node\* root) //DLR

{

if(root==NULL)

{

return;

}

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

preorder(root->left);

preorder(root->right);

}

void postorder(struct node\* root) //LRD

{

if(root==NULL)

{

return;

}

postorder(root->left);

postorder(root->right);

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

}