1.POST ORDER WITHOUT RECURSION

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

// Definition of a binary tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function for iterative postorder traversal
void iterativePostorder(struct Node* root) {
    if (root == NULL)
        return;

    struct Node* stack1[100];
    struct Node* stack2[100];
    int top1 = -1;
    int top2 = -1;
```

```
    stack1[++top1] = root;

    while (top1 >= 0) {
        struct Node* curr = stack1[top1--];
        stack2[++top2] = curr;

        if (curr->left != NULL)
            stack1[++top1] = curr->left;
        if (curr->right != NULL)
            stack1[++top1] = curr->right;
    }

    while (top2 >= 0) {
        struct Node* curr = stack2[top2--];
        printf("%d ", curr->data);
    }
}

int main() {
    // Create the binary tree
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    printf("Postorder traversal: ");
    iterativePostorder(root);
```

```
    return 0;
}
```

2. PREORDER WITHOUT RECURSION

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

// Definition of a binary tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function for iterative preorder traversal
void iterativePreorder(struct Node* root) {
    if (root == NULL)
        return;

    struct Node* stack[100];
```

```
    int top = -1;
    stack[++top] = root;

    while (top >= 0) {
        struct Node* curr = stack[top--];
        printf("%d ", curr->data);

        if (curr->right != NULL)
            stack[++top] = curr->right;
        if (curr->left != NULL)
            stack[++top] = curr->left;
    }
}

int main() {
    // Create the binary tree
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    printf("Preorder traversal: ");
    iterativePreorder(root);

    return 0;
}
```

3. COINS PROBLEM USING GREEDY METHOD

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

```
void countNotes(int amount) {
    int denominations[] = {2000, 500, 200, 100, 50, 20, 10, 5, 2, 1};
    int count[10] = {0};
    int i;

    for (i = 0; i < 10; i++) {
        if (amount >= denominations[i]) {
            count[i] = amount / denominations[i];
            amount = amount - (count[i] * denominations[i]);
        }
    }

    printf("Denomination\tNumber of Notes\n");
    for (i = 0; i < 10; i++) {
        if (count[i] != 0) {
            printf("%d\t\t%d\n", denominations[i], count[i]);
        }
    }
}

int main() {
    int amount;

    printf("Enter the amount to withdraw: ");
    scanf("%d", &amount);

    printf("Minimum number of coins/notes:\n");
    countNotes(amount);
```

```
    return 0;
}
```

4. RANDOMIZED QUICK SORT

```
#include <stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>
void quicksort(int x[20],int p,int q)
{
    int temp,i,j,pivot;
    if(p<q)
    {
        i=p;
        j=q;
        pivot=p+rand()%(q-p);
        temp=x[pivot];
        x[pivot]=x[p];
        x[p]=temp;
        while(i<j)
        {
            while(x[i]<=x[pivot] && i<q)
            {
                i++;
            }
            while(x[j]>x[pivot])
            {
                j--;
            }
```

```
            if(i<j)
            {
               temp=x[i];
               x[i]=x[j];
               x[j]=temp;
            }}
         temp=x[pivot];
         x[pivot]=x[j];
         x[j]=temp;
         quicksort(x,p,j-1);
         quicksort(x,j+1,q);
      }
   }

   int main()
   {
      int n;
      printf("Enter the no. of elements:");
      scanf("%d",&n);
      int a[n];
      int i;
      for(i=0;i<n;i++)
      {
         scanf("%d",&a[i]);
      }
      quicksort(a,0,n-1);
      printf("Sorted:");
      for(i=0;i<n;i++)
      {
```

```
        printf("%d\t",a[i]);
    }

    return 0;
}
```

5. DOUBLE HASHING

```
#include <stdio.h>
#include<stdlib.h>
#define TABLE_SIZE 10

int h[TABLE_SIZE]={NULL};

void insert()
{

 int key,index,i,flag=0,hkey,hash2;
 printf("\nenter a value to insert into hash table\n");
 scanf("%d",&key);
 hkey=key%TABLE_SIZE;
 hash2 = 7-(key %7);
 for(i=0;i<TABLE_SIZE;i++)
 {
    index=(hkey+i*hash2)%TABLE_SIZE;
    if(h[index] == NULL)
    {
        h[index]=key;
        break;
    }
 }
```

```
 if(i == TABLE_SIZE)
     printf("\nelement cannot be inserted\n");
}
void search()
{

 int key,index,i,flag=0,hash2,hkey;
 printf("\nenter search element\n");
 scanf("%d",&key);
  hkey=key%TABLE_SIZE;
 hash2 = 7-(key %7);
 for(i=0;i<TABLE_SIZE; i++)
 {
   index=(hkey+i*hash2)%TABLE_SIZE;
   if(h[index]==key)
   {
     printf("value is found at index %d",index);
     break;
   }
  }
  if(i == TABLE_SIZE)
   printf("\n value is not found\n");
}
void display()
{

  int i;
  printf("\nelements in the hash table are \n");
  for(i=0;i< TABLE_SIZE; i++)
```

```
    printf("\nat index %d \t value =  %d",i,h[i]);


}
main()
{
    int opt,i;
    while(1)
    {
        printf("\nPress 1. Insert\t 2. Display \t3. Search \t4.Exit \n");
        scanf("%d",&opt);
        switch(opt)
        {
            case 1:
                insert();
                break;
            case 2:
                display();
                break;
            case 3:
                search();
                break;
            case 4:exit(0);
        }
    }
}
```

6. BINARY SEARCH USING DIVIDE AND CONQUER

```
#include<stdio.h>


int binary_search(int A[], int key, int len) {
```

```
  int low = 0;
  int high = len -1;

  while (low <= high) {
    int mid = low + ((high - low) / 2);

    if (A[mid] == key) {
      return mid;
    }

    if (key < A[mid]) {
      high = mid - 1;
    }
    else {
      low = mid + 1;
    }
  }
  return -1;
}

int main() {
  int a[10]={1,3,5,7,9,11,13,15,17,21};
  int key = 3;

  int position = binary_search(a, key, 10);
  if (position == -1){
    printf("Not found");
    return 0;
```

```
  }
  printf("Found it at %d", position);
  return 0;
}
```

7. INORDER TRAVERSAL WITHOUT RECURSION

```
#include<stdio.h>
#include<stdlib.h>
struct node // node defining for tree
{
	struct node* left;
	struct node* right;
	int data;
};
struct stack // node defining for stack
{
	struct node* data;
	struct stack* next;
};

void push(struct stack** top,struct node* n); //function declation
struct node* pop(struct stack** top);
int isEmpty(struct stack* top);

int Inorder(struct node* root) //Inorder Traversing function
{
	struct node* temp = root;
	struct stack* s_temp = NULL;
	int flag = 1;
	while(flag) //Loop run untill temp is null and stack is empty
```

```
	{
		if(temp){
			push(&s_temp,temp);
			temp = temp->left;
		}
		else{
			if(!isEmpty(s_temp)){
				temp = pop(&s_temp);
				printf("%d ",temp->data);
				temp = temp->right;
			}
			else
				flag = 0;
		}
	 }
}
void push(struct stack** top,struct node* n) //push node in stack
{
	struct stack* new_n = (struct stack*)malloc(sizeof(struct stack));
	new_n->data = n;
	new_n->next = (*top);
	(*top) = new_n;
}
int isEmpty(struct stack* top) // check if stack is empty
{
	if(top==NULL)
		return 1;
	else
		return 0;
```

```
}
struct node* pop(struct stack** top_n) // pop the node from stack
{
	struct node* item;
	struct stack* top;
	top = *top_n;
	item = top->data;
	*top_n = top->next;
	free(top);
	return item;
}
struct node* create_node(int data) // create a node for tree
{
	struct node* new_n = (struct node*)malloc(sizeof(struct node));
	new_n->data = data;
	new_n->left = NULL;
	new_n->right = NULL;
	return (new_n);
}

int main()
{
	struct node* root;
	root = create_node(8);
	root->left = create_node(5);
	root->right = create_node(4);
	root->left->left = create_node(7);
	root->left->right = create_node(6);
	Inorder(root);
```

```
        return 0;
}
```

8. JOB SEQUENCING USING GREEDY ALGORITHM

```
#include <stdio.h>

#define MAX 100

typedef struct Job {
  char id[5];
  int deadline;
  int profit;
} Job;

void jobSequencingWithDeadline(Job jobs[], int n);

int minValue(int x, int y) {
  if(x < y) return x;
  return y;
}

int main(void) {
  //variables
  int i, j;

  //jobs with deadline and profit
  Job jobs[5] = {
    {"j1", 2,  60},
    {"j2", 1, 100},
    {"j3", 3,  20},
```

```
  {"j4", 2,  40},
  {"j5", 1,  20},
};

//temp
Job temp;

//number of jobs
int n = 5;

//sort the jobs profit wise in descending order
for(i = 1; i < n; i++) {
  for(j = 0; j < n - i; j++) {
    if(jobs[j+1].profit > jobs[j].profit) {
      temp = jobs[j+1];
      jobs[j+1] = jobs[j];
      jobs[j] = temp;
    }
  }
}

printf("%10s %10s %10s\n", "Job", "Deadline", "Profit");
for(i = 0; i < n; i++) {
  printf("%10s %10i %10i\n", jobs[i].id, jobs[i].deadline, jobs[i].profit);
}

jobSequencingWithDeadline(jobs, n);

return 0;
```

```
}

void jobSequencingWithDeadline(Job jobs[], int n) {
  //variables
  int i, j, k, maxprofit;

  //free time slots
  int timeslot[MAX];

  //filled time slots
  int filledTimeSlot = 0;

  //find max deadline value
  int dmax = 0;
  for(i = 0; i < n; i++) {
    if(jobs[i].deadline > dmax) {
      dmax = jobs[i].deadline;
    }
  }

  //free time slots initially set to -1 [-1 denotes EMPTY]
  for(i = 1; i <= dmax; i++) {
    timeslot[i] = -1;
  }

  printf("dmax: %d\n", dmax);

  for(i = 1; i <= n; i++) {
    k = minValue(dmax, jobs[i - 1].deadline);
```

```
  while(k >= 1) {
    if(timeslot[k] == -1) {
      timeslot[k] = i-1;
      filledTimeSlot++;
      break;
    }
    k--;
  }

  //if all time slots are filled then stop
  if(filledTimeSlot == dmax) {
    break;
  }
}

//required jobs
printf("\nRequired Jobs: ");
for(i = 1; i <= dmax; i++) {
  printf("%s", jobs[timeslot[i]].id);

  if(i < dmax) {
    printf(" --> ");
  }
}

//required profit
maxprofit = 0;
for(i = 1; i <= dmax; i++) {
  maxprofit += jobs[timeslot[i]].profit;
```

```
  }
  printf("\nMax Profit: %d\n", maxprofit);
}
```

9. MAXMIN USING DIVIDE AND CONQUER

```
#include<stdio.h>
#include<stdio.h>
int max, min;
int a[100];
void maxmin(int i, int j)
{
 int max1, min1, mid;
 if(i==j)
 {
  max = min = a[i];
 }
 else
 {
  if(i == j-1)
  {
   if(a[i] <a[j])
   {
    max = a[j];
    min = a[i];
   }
   else
   {
    max = a[i];
    min = a[j];
   }
```

```
 }
 else
 {
  mid = (i+j)/2;
  maxmin(i, mid);
  max1 = max; min1 = min;
  maxmin(mid+1, j);
  if(max <max1)
   max = max1;
  if(min > min1)
   min = min1;
 }
}
}
int main ()
{
 int i, num;
 printf ("\nEnter the total number of numbers : ");
 scanf ("%d",&num);
 printf ("Enter the numbers : \n");
 for (i=1;i<=num;i++)
  scanf ("%d",&a[i]);

 max = a[0];
 min = a[0];
 maxmin(1, num);
 printf ("Minimum element in an array : %d\n", min);
 printf ("Maximum element in an array : %d\n", max);
 return 0;
```

```
}
```

10. OPTIMAL STORAGE ON TAPES

```
#include <stdio.h>


// Function to calculate the optimal storage on tapes
void optimalTapeStorage(int tapes[], int numTapes, int files[], int numFiles) {
  int tapeIndex = 0;  // Current tape index


  // Loop through each file
  for (int i = 0; i < numFiles; i++) {
    // Check if the file can fit in the current tape
    if (files[i] <= tapes[tapeIndex]) {
      // Store the file in the current tape
      tapes[tapeIndex] -= files[i];
      printf("File %d stored in Tape %d\n", i+1, tapeIndex+1);
    } else {
      // Move to the next tape
      tapeIndex++;
      // Check if all tapes have been used
      if (tapeIndex >= numTapes) {
        printf("Error: Not enough tapes to store all files\n");
        return;
      }
      // Store the file in the next tape
      tapes[tapeIndex] -= files[i];
      printf("File %d stored in Tape %d\n", i+1, tapeIndex+1);
    }
  }
}
```

```
int main() {
    // Number of tapes and files
    int numTapes = 3;
    int numFiles = 5;

    // Sizes of tapes and files
    int tapes[] = {100, 200, 150};
    int files[] = {80, 50, 30, 70, 120};

    // Calculate the optimal storage on tapes
    optimalTapeStorage(tapes, numTapes, files, numFiles);

    return 0;
}
```

11. LINEAR PROBING HASHING

```
#include <stdio.h>
#include<stdlib.h>
#define TABLE_SIZE 10

int h[TABLE_SIZE]={NULL};

void insert()
{

 int key,index,i,flag=0,hkey;
 printf("\nenter a value to insert into hash table\n");
 scanf("%d",&key);
 hkey=key%TABLE_SIZE;
```

```
 for(i=0;i<TABLE_SIZE;i++)
   {

    index=(hkey+i)%TABLE_SIZE;

    if(h[index] == NULL)
    {
       h[index]=key;
        break;
    }

   }

   if(i == TABLE_SIZE)

    printf("\nelement cannot be inserted\n");
}
void search()
{

 int key,index,i,flag=0,hkey;
 printf("\nenter search element\n");
 scanf("%d",&key);
 hkey=key%TABLE_SIZE;
 for(i=0;i<TABLE_SIZE; i++)
 {
   index=(hkey+i)%TABLE_SIZE;
   if(h[index]==key)
   {
```

```
      printf("value is found at index %d",index);
      break;
    }
  }
  if(i == TABLE_SIZE)
    printf("\n value is not found\n");
}
void display()
{

  int i;

  printf("\nelements in the hash table are \n");

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

  printf("\nat index %d \t value =  %d",i,h[i]);

}
int main()
{
    int opt,i;
    while(1)
    {
        printf("\nPress 1. Insert\t 2. Display \t3. Search \t4.Exit \n");
        scanf("%d",&opt);
        switch(opt)
        {
            case 1:
```

```
        insert();
        break;
      case 2:
        display();
        break;
      case 3:
        search();
        break;
      case 4:exit(0);
    }
  }
  return 0;
}
```

12. QUICK SORT

```
#include<stdio.h>
void quicksort(int number[25],int first,int last){
  int i, j, pivot, temp;
  if(first<last){
    pivot=first;
    i=first;
    j=last;
    while(i<j){
      while(number[i]<=number[pivot]&&i<last)
      i++;
      while(number[j]>number[pivot])
      j--;
      if(i<j){
        temp=number[i];
        number[i]=number[j];
```

```
            number[j]=temp;
         }
       }
       temp=number[pivot];
       number[pivot]=number[j];
       number[j]=temp;
       quicksort(number,first,j-1);
       quicksort(number,j+1,last);
   }
}
int main(){
   int i, count, number[25];
   printf("How many elements are u going to enter?: ");
   scanf("%d",&count);
   printf("Enter %d elements: ", count);
   for(i=0;i<count;i++)
   scanf("%d",&number[i]);
   quicksort(number,0,count-1);
   printf("Order of Sorted elements: ");
   for(i=0;i<count;i++)
   printf(" %d",number[i]);
   return 0;
}
```

13. TOPOLOGICAL SORT

```
#include<stdio.h>
int vis[10]={0};
int a[10][10]={0};
void fun1(int n)
{
```

```
    int i,j,count;
    for(i=0;i<n;i++)
    {
        count=0;
        if(vis[i]==0)
        {
            for(j=0;j<n;j++)
            {
                if((a[j][i]==1)&&(vis[j]!=1))
                {
                    count+=1;
                }
            }
            if(count==0)
            {
                printf("%d",i+1);
                vis[i]=1;
            }
        }
    }
    return;
}

int main()
{
    int n,i,j;
    printf("Enter no.of vertices:");
    scanf("%d",&n);
    printf("Enter the adjacency matrix:");
```

```
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    fun1(n);
    for(i=0;i<n;i++)
    {
        if(vis[i]==0)
        {
            fun1(n);
        }
    }
    return 0;
}
```

14. FRACTIONAL KNAPSACK

```
#include <stdio.h>

int n = 5; /* The number of objects */
int c[10] = {12, 1, 2, 1, 4}; /* c[i] is the *COST* of the ith object; i.e. what
                YOU PAY to take the object */
int v[10] = {4, 2, 2, 1, 10}; /* v[i] is the *VALUE* of the ith object; i.e.
                what YOU GET for taking the object */
int W = 15; /* The maximum weight you can take */

void simple_fill() {
    int cur_w;
```

```
    float tot_v;

    int i, maxi;

    int used[10];


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

        used[i] = 0; /* I have not used the ith object yet */


    cur_w = W;

    while (cur_w > 0) { /* while there's still room*/

        /* Find the best object */

        maxi = -1;

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

            if ((used[i] == 0) &&

                ((maxi == -1) || ((float)v[i]/c[i] > (float)v[maxi]/c[maxi])))

                maxi = i;


        used[maxi] = 1; /* mark the maxi-th object as used */

        cur_w -= c[maxi]; /* with the object in the bag, I can carry less */

        tot_v += v[maxi];

        if (cur_w >= 0)

            printf("Added object %d (%d$, %dKg) completely in the bag. Space left: %d.\n", maxi + 1, v[maxi], c[maxi],
cur_w);

        else {

            printf("Added %d%% (%d$, %dKg) of object %d in the bag.\n", (int)((1 + (float)cur_w/c[maxi]) * 100),
v[maxi], c[maxi], maxi + 1);

            tot_v -= v[maxi];

            tot_v += (1 + (float)cur_w/c[maxi]) * v[maxi];

        }

    }
```

```
    printf("Filled the bag with objects worth %.2f$.\n", tot_v);
}

int main(int argc, char *argv[]) {
    simple_fill();

    return 0;
}
```

15. KRUSKALS ALGORITHM

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

#define MAX_EDGES 100
#define MAX_VERTICES 100

typedef struct {
    int u, v, weight;
} Edge;

typedef struct {
    int parent, rank;
} Subset;

int find(Subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}
```

```
void union_set(Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

int cmp(const void* a, const void* b) {
    Edge* a1 = (Edge*)a;
    Edge* b1 = (Edge*)b;
    return a1->weight - b1->weight;
}

void kruskal(int n, int m, Edge edges[]) {
    Subset subsets[MAX_VERTICES];
    int i, j, k;
    Edge result[MAX_EDGES];

    qsort(edges, m, sizeof(Edge), cmp);

    for (i = 0; i < n; i++) {
        subsets[i].parent = i;
```

```
        subsets[i].rank = 0;
    }

    j = 0;
    k = 0;
    while (j < n - 1 && k < m) {
        Edge next_edge = edges[k++];

        int x = find(subsets, next_edge.u);
        int y = find(subsets, next_edge.v);

        if (x != y) {
            result[j++] = next_edge;
            union_set(subsets, x, y);
        }
    }

    printf("Minimum spanning tree:\n");
    for (i = 0; i < j; i++)
        printf("(%d, %d) -> %d\n", result[i].u, result[i].v, result[i].weight);
}

int main() {
    int n, m, i;
    Edge edges[MAX_EDGES];

    printf("Enter number of vertices and edges: ");
    scanf("%d %d", &n, &m);
```

```
    printf("Enter edges in the format (u, v, weight):\n");
    for (i = 0; i < m; i++)
        scanf("%d %d %d", &edges[i].u, &edges[i].v, &edges[i].weight);

    kruskal(n, m, edges);

    return 0;
}
```

16. PRIMS ALGORITHM

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

#define MAX 100

int cost[MAX][MAX], near[MAX], min_cost = 0;

void prim(int n) {
    int i, j, k, u, v, min;

    // initialize near array
    for (i = 2; i <= n; i++) {
        near[i] = 1;
    }

    // iterate n-1 times to construct tree
    for (i = 1; i < n; i++) {
        min = INT_MAX;

        // find edge with minimum cost
```

```
        for (j = 2; j <= n; j++) {
            if (near[j] != 0 && cost[j][near[j]] < min) {
                min = cost[j][near[j]];
                u = j;
                v = near[j];
            }
        }

        // add edge to tree
        printf("(%d, %d) cost: %d\n", u, v, min);
        min_cost += min;
        near[u] = 0;

        // update near array
        for (k = 2; k <= n; k++) {
            if (near[k] != 0 && cost[k][near[k]] > cost[k][u]) {
                near[k] = u;
            }
        }
    }

    printf("Minimum cost = %d\n", min_cost);
}

int main() {
    int n, i, j;

    printf("Enter the number of vertices: ");
    scanf("%d", &n);
```

```c
    printf("Enter the cost adjacency matrix:\n");
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= n; j++) {
            scanf("%d", &cost[i][j]);
        }
    }

    printf("Minimum cost spanning tree edges:\n");
    prim(n);

    return 0;
}
```

17. HEAP SORT

```c
#include <stdio.h>
/* function to heapify a subtree. Here 'i' is the
index of root node in array a[], and 'n' is the size of heap. */
void heapify(int a[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // left child
    int right = 2 * i + 2; // right child
    // If left child is larger than root
    if (left < n && a[left] > a[largest])
        largest = left;
    // If right child is larger than root
    if (right < n && a[right] > a[largest])
        largest = right;
    // If root is not largest
```

```
    if (largest != i) {
        // swap a[i] with a[largest]
        int temp = a[i];
        a[i] = a[largest];
        a[largest] = temp;

        heapify(a, n, largest);
    }
}
/*Function to implement the heap sort*/
void heapSort(int a[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(a, n, i);
    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i--) {
        /* Move current root element to end*/
        // swap a[0] with a[i]
        int temp = a[0];
        a[0] = a[i];
        a[i] = temp;

        heapify(a, i, 0);
    }
}
/* function to print the array elements */
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
```

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

}
int main()
{
    int a[] = {48, 10, 23, 43, 28, 26, 1};
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArr(a, n);
    heapSort(a, n);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);
    return 0;
}
```

18. MERGE SORT

```
#include <stdio.h>

#define max 10

int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };
int b[10];

void merging(int low, int mid, int high) {
   int l1, l2, i;

   for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
```

```
      if(a[l1] <= a[l2])
         b[i] = a[l1++];
      else
         b[i] = a[l2++];
   }

   while(l1 <= mid)
      b[i++] = a[l1++];

   while(l2 <= high)
      b[i++] = a[l2++];

   for(i = low; i <= high; i++)
      a[i] = b[i];
}

void sort(int low, int high) {
   int mid;

   if(low < high) {
      mid = (low + high) / 2;
      sort(low, mid);
      sort(mid+1, high);
      merging(low, mid, high);
   } else {
      return;
   }
}
```

```
int main() {
  int i;

  printf("List before sorting\n");

  for(i = 0; i <= max; i++)
    printf("%d ", a[i]);

  sort(0, max);

  printf("\nList after sorting\n");

  for(i = 0; i <= max; i++)
    printf("%d ", a[i]);
   return 0;
}
```

19. BFS

```
#include <stdio.h>
#define MAX_SIZE 100

// Queue implementation
int queue[MAX_SIZE];
int front = -1, rear = -1;

void enqueue(int value) {
   if (rear == MAX_SIZE - 1) {
       printf("Queue is full.\n");
       return;
   }
```

```
    if (front == -1)
        front = 0;
    rear++;
    queue[rear] = value;
}

int dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue is empty.\n");
        return -1;
    }
    int value = queue[front];
    front++;
    return value;
}

int isQueueEmpty() {
    if (front == -1 || front > rear)
        return 1;
    else
        return 0;
}

// BFS implementation
void BFS(int adjacencyMatrix[][MAX_SIZE], int vertices, int source) {
    int visited[MAX_SIZE] = {0};
    enqueue(source);
    visited[source] = 1;
```

```c
    printf("BFS traversal: ");
    while (!isQueueEmpty()) {
        int currentVertex = dequeue();
        printf("%d ", currentVertex);

        for (int i = 0; i < vertices; i++) {
            if (adjacencyMatrix[currentVertex][i] && !visited[i]) {
                enqueue(i);
                visited[i] = 1;
            }
        }
    }
}

// Test the BFS function
int main() {
    int vertices, source;
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);

    int adjacencyMatrix[MAX_SIZE][MAX_SIZE];
    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            scanf("%d", &adjacencyMatrix[i][j]);
        }
    }

    printf("Enter the source vertex: ");
```

```
    scanf("%d", &source);

    BFS(adjacencyMatrix, vertices, source);

    return 0;
}
```

20. DFS

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

#define MAX_SIZE 100

// Data structure to represent a graph node
typedef struct Node {
    int data;
    struct Node* next;
} Node;

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to add an edge between two nodes in the graph
void addEdge(Node* graph[], int src, int dest) {
    Node* newNode = createNode(dest);
```

```
    newNode->next = graph[src];
    graph[src] = newNode;

    newNode = createNode(src);
    newNode->next = graph[dest];
    graph[dest] = newNode;
}

// Function to perform Depth-First Search traversal
void DFS(Node* graph[], int vertex, int visited[]) {
    visited[vertex] = 1;
    printf("%d ", vertex);

    Node* adjNode = graph[vertex];
    while (adjNode != NULL) {
        int adjVertex = adjNode->data;
        if (visited[adjVertex] == 0) {
            DFS(graph, adjVertex, visited);
        }
        adjNode = adjNode->next;
    }
}

// Function to initialize visited array with all zeros
void initializeVisited(int visited[], int size) {
    for (int i = 0; i < size; i++) {
        visited[i] = 0;
    }
}
```

```
// Driver code
int main() {
    int numVertices = 6;
    Node* graph[MAX_SIZE];
    int visited[MAX_SIZE];

    // Initialize graph and visited array
    for (int i = 0; i < MAX_SIZE; i++) {
        graph[i] = NULL;
        visited[i] = 0;
    }

    // Add edges to the graph
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);
    addEdge(graph, 3, 5);

    printf("Depth-First Search Traversal: ");
    initializeVisited(visited, numVertices);
    DFS(graph, 0, visited);

    return 0;
}
```