##############################################

11) BFS

#include <stdio.h>

#include <stdlib.h>

#define SIZE 40

struct queue {

int items[SIZE];

int front;

int rear;

};

struct queue\* createQueue();

void enqueue(struct queue\* q, int);

int dequeue(struct queue\* q);

void display(struct queue\* q);

int isEmpty(struct queue\* q);

void printQueue(struct queue\* q);

struct node {

int vertex;

struct node\* next;

};

struct node\* createNode(int);

struct Graph {

int numVertices;

struct node\*\* adjLists;

int\* visited;

};

void bfs(struct Graph\* graph, int startVertex) {

struct queue\* q = createQueue();

graph->visited[startVertex] = 1;

enqueue(q, startVertex);

while (!isEmpty(q)) {

printQueue(q);

int currentVertex = dequeue(q);

printf("Visited %d\n", currentVertex);

struct node\* temp = graph->adjLists[currentVertex];

while (temp) {

int adjVertex = temp->vertex;

if (graph->visited[adjVertex] == 0) {

graph->visited[adjVertex] = 1;

enqueue(q, adjVertex);

}

temp = temp->next;

}

}

}

struct node\* createNode(int v) {

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

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

struct Graph\* createGraph(int vertices) {

struct Graph\* graph = malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = malloc(vertices \* sizeof(struct node\*));

graph->visited = malloc(vertices \* sizeof(int));

int i;

for (i = 0; i < vertices; i++) {

graph->adjLists[i] = NULL;

graph->visited[i] = 0;

}

return graph;

}

void addEdge(struct Graph\* graph, int src, int dest) {

struct node\* newNode = createNode(dest);

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

graph->adjLists[src] = newNode;

newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;

}

struct queue\* createQueue() {

struct queue\* q = malloc(sizeof(struct queue));

q->front = -1;

q->rear = -1;

return q;

}

int isEmpty(struct queue\* q) {

if (q->rear == -1)

return 1;

else

return 0;

}

void enqueue(struct queue\* q, int value) {

if (q->rear == SIZE - 1)

printf("\nQueue is Full!!");

else {

if (q->front == -1)

q->front = 0;

q->rear++;

q->items[q->rear] = value;

}

}

int dequeue(struct queue\* q) {

int item;

if (isEmpty(q)) {

printf("Queue is empty");

item = -1;

} else {

item = q->items[q->front];

q->front++;

if (q->front > q->rear) {

printf("Resetting queue ");

q->front = q->rear = -1;

}

}

return item;

}

void printQueue(struct queue\* q) {

int i = q->front;

if (isEmpty(q)) {

printf("Queue is empty");

} else {

printf("\nQueue contains \n");

for (i = q->front; i < q->rear + 1; i++) {

printf("%d ", q->items[i]);

}

}

}

int main() {

struct Graph\* graph = createGraph(6);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 1, 4);

addEdge(graph, 1, 3);

addEdge(graph, 2, 4);

addEdge(graph, 3, 4);

bfs(graph, 0);

return 0;

}

############################################################

11) DFS

#include <stdio.h>

#include <stdlib.h>

#define MAX\_VERTICES 100

struct Node {

int vertex;

struct Node\* next;

};

struct Graph {

int numVertices;

struct Node\* adjList[MAX\_VERTICES];

};

struct Graph\* createGraph(int numVertices) {

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->numVertices = numVertices;

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

graph->adjList[i] = NULL;

}

return graph;

}

void addEdge(struct Graph\* graph, int src, int dest) {

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

newNode->vertex = dest;

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

graph->adjList[src] = newNode;

}

void DFS(struct Graph\* graph, int vertex, int visited[]) {

visited[vertex] = 1;

printf("%d ", vertex);

struct Node\* temp = graph->adjList[vertex];

while (temp != NULL) {

int adjVertex = temp->vertex;

if (!visited[adjVertex]) {

DFS(graph, adjVertex, visited);

}

temp = temp->next;

}

}

int main() {

int numVertices = 4;

struct Graph\* graph = createGraph(numVertices);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 2, 0);

addEdge(graph, 2, 3);

addEdge(graph, 3, 3);

int visited[MAX\_VERTICES] = {0};

printf("Depth First Traversal (starting from vertex 2):\n");

DFS(graph, 2, visited);

return 0;

}

##################################################################

12) Topo Sort

#include <stdio.h>

#include <stdlib.h>

typedef struct AdjListNode {

int dest;

struct AdjListNode\* next;

} AdjListNode;

typedef struct AdjList {

AdjListNode\* head;

} AdjList;

typedef struct Graph {

int V;

AdjList\* array;

} Graph;

AdjListNode\* newAdjListNode(int dest) {

AdjListNode\* newNode = (AdjListNode\*)malloc(sizeof(AdjListNode));

newNode->dest = dest;

newNode->next = NULL;

return newNode;

}

Graph\* createGraph(int V) {

Graph\* graph = (Graph\*)malloc(sizeof(Graph));

graph->V = V;

graph->array = (AdjList\*)malloc(V \* sizeof(AdjList));

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

graph->array[i].head = NULL;

return graph;

}

void addEdge(Graph\* graph, int src, int dest) {

AdjListNode\* newNode = newAdjListNode(dest);

newNode->next = graph->array[src].head;

graph->array[src].head = newNode;

newNode = newAdjListNode(src);

newNode->next = graph->array[dest].head;

graph->array[dest].head = newNode;

}

void printGraph(Graph\* graph) {

for (int v = 0; v < graph->V; ++v) {

AdjListNode\* pCrawl = graph->array[v].head;

printf("\nAdjacency list of vertex %d\nhead", v);

while (pCrawl) {

printf(" -> %d", pCrawl->dest);

pCrawl = pCrawl->next;

}

printf("\n");

}

}

void DFSUtil(Graph\* graph, int v, int visited[]) {

visited[v] = 1;

printf("%d ", v);

AdjListNode\* adjList = graph->array[v].head;

while (adjList) {

int connectedVertex = adjList->dest;

if (!visited[connectedVertex])

DFSUtil(graph, connectedVertex, visited);

adjList = adjList->next;

}

}

void DFS(Graph\* graph, int startVertex) {

int\* visited = (int\*)malloc(graph->V \* sizeof(int));

for (int i = 0; i < graph->V; i++)

visited[i] = 0;

DFSUtil(graph, startVertex, visited);

free(visited);

}

int main() {

int V = 5;

Graph\* graph = createGraph(V);

addEdge(graph, 0, 1);

addEdge(graph, 0, 4);

addEdge(graph, 1, 2);

addEdge(graph, 1, 3);

addEdge(graph, 1, 4);

addEdge(graph, 2, 3);

addEdge(graph, 3, 4);

printf("Graph adjacency list representation:\n");

printGraph(graph);

printf("\nDFS starting from vertex 0:\n");

DFS(graph, 0);

return 0;

}

####################################################

13) Prim's

#include <stdio.h>

#include <limits.h>

#define MAX\_VERTICES 100

int minKey(int key[], int mstSet[], int vertices) {

int min = INT\_MAX, minIndex;

for (int v = 0; v < vertices; v++) {

if (!mstSet[v] && key[v] < min) {

min = key[v];

minIndex = v;

}

}

return minIndex;

}

void printMST(int parent[], int graph[MAX\_VERTICES][MAX\_VERTICES], int vertices) {

printf("Edge \tWeight\n");

for (int i = 1; i < vertices; i++) {

printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);

}

}

void primMST(int graph[MAX\_VERTICES][MAX\_VERTICES], int vertices) {

int parent[MAX\_VERTICES];

int key[MAX\_VERTICES];

int mstSet[MAX\_VERTICES];

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

key[i] = INT\_MAX;

mstSet[i] = 0;

}

key[0] = 0;

parent[0] = -1;

for (int count = 0; count < vertices - 1; count++) {

int u = minKey(key, mstSet, vertices);

mstSet[u] = 1;

for (int v = 0; v < vertices; v++) {

if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {

parent[v] = u;

key[v] = graph[u][v];

}

}

}

printMST(parent, graph, vertices);

}

int main() {

int vertices;

printf("Input the number of vertices: ");

scanf("%d", &vertices);

if (vertices <= 0 || vertices > MAX\_VERTICES) {

printf("Invalid number of vertices. Exiting...\n");

return 1;

}

int graph[MAX\_VERTICES][MAX\_VERTICES];

printf("Input the adjacency matrix for the graph:\n");

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

for (int j = 0; j < vertices; j++) {

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

}

}

primMST(graph, vertices);

return 0;

}

##############################################################

14) DJ Algo

#include <stdio.h>

#include <limits.h>

#define MAX\_VERTICES 100

int minDistance(int dist[], int sptSet[], int vertices) {

int min = INT\_MAX, minIndex;

for (int v = 0; v < vertices; v++) {

if (!sptSet[v] && dist[v] < min) {

min = dist[v];

minIndex = v;

}

}

return minIndex;

}

void printSolution(int dist[], int vertices) {

printf("Vertex \tDistance from Source\n");

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

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

}

}

void dijkstra(int graph[MAX\_VERTICES][MAX\_VERTICES], int src, int vertices) {

int dist[MAX\_VERTICES];

int sptSet[MAX\_VERTICES]; for (int i = 0; i < vertices; i++) {

dist[i] = INT\_MAX;

sptSet[i] = 0;

}

dist[src] = 0;

for (int count = 0; count < vertices - 1; count++) {

int u = minDistance(dist, sptSet, vertices);

sptSet[u] = 1;

for (int v = 0; v < vertices; v++) {

if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v]) {

dist[v] = dist[u] + graph[u][v];

}

}

}

printSolution(dist, vertices);

}

int main() {

int vertices;

printf("Input the number of vertices: ");

scanf("%d", &vertices);

if (vertices <= 0 || vertices > MAX\_VERTICES) {

printf("Invalid number of vertices. Exiting...\n");

return 1;

}

int graph[MAX\_VERTICES][MAX\_VERTICES];

printf("Input the adjacency matrix for the graph (use INT\_MAX for infinity):\n");

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

for (int j = 0; j < vertices; j++) {

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

}

}

int source;

printf("Input the source vertex: ");

scanf("%d", &source);

if (source < 0 || source >= vertices) {

printf("Invalid source vertex. Exiting...\n");

return 1;

}

dijkstra(graph, source, vertices);

return 0;

}

###########################################

15)

#include <stdio.h>

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

int t = \*a;

\*a = \*b;

\*b = t;

}

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

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j <= high - 1; j++) {

if (arr[j] < pivot) {

i++;

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

}

}

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

return (i + 1);

}

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

if (low < high) {

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

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

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

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

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

printf("\n");

}

int main() {

int arr[] = {10, 7, 8, 9, 1, 5};

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

printf("Original array: \n");

printArray(arr, n);

quickSort(arr, 0, n - 1);

printf("Sorted array with Quick Sort: \n");

printArray(arr, n);

return 0;

}

#include <stdio.h>

void merge(int arr[], int l, int m, int r) {

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

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

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

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

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

int i = 0, j = 0, k = l;

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

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

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

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

if (l < r) {

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

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

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

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

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

printf("\n");

}

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

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

printf("Original array: \n");

printArray(arr, n);

mergeSort(arr, 0, n - 1);

printf("Sorted array with Merge Sort: \n");

printArray(arr, n);

return 0;

}

###########################################################

16)

#include <stdio.h>

#include <stdlib.h>

#define TABLE\_SIZE 10

int hash(int key) {

return key % TABLE\_SIZE;

}

void insert(int table[], int key) {

int index = hash(key);

int i = 0;

while (table[(index + i) % TABLE\_SIZE] != -1) {

i++;

}

table[(index + i) % TABLE\_SIZE] = key;

}

void display(int table[]) {

for (int i = 0; i < TABLE\_SIZE; i++) {

if (table[i] != -1)

printf("%d -> %d\n", i, table[i]);

else

printf("%d -> \n", i);

}

}

int main() {

int table[TABLE\_SIZE];

for (int i = 0; i < TABLE\_SIZE; i++)

table[i] = -1; // Initialize table with -1 (indicating empty slots)

insert(table, 10);

insert(table, 20);

insert(table, 30);

insert(table, 25);

printf("Open Addressing Hash Table:\n");

display(table);

return 0;

}

#include <stdio.h>

#include <stdlib.h>

#define TABLE\_SIZE 10

typedef struct Node {

int key;

struct Node\* next;

} Node;

Node\* createNode(int key) {

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

newNode->key = key;

newNode->next = NULL;

return newNode;

}

int hash(int key) {

return key % TABLE\_SIZE;

}

void insert(Node\* table[], int key) {

int index = hash(key);

Node\* newNode = createNode(key);

newNode->next = table[index];

table[index] = newNode;

}

void display(Node\* table[]) {

for (int i = 0; i < TABLE\_SIZE; i++) {

printf("%d -> ", i);

Node\* temp = table[i];

while (temp) {

printf("%d -> ", temp->key);

temp = temp->next;

}

printf("NULL\n");

}

}

int main() {

Node\* table[TABLE\_SIZE] = { NULL }; // Initialize hash table with NULL pointers

insert(table, 10);

insert(table, 20);

insert(table, 30);

insert(table, 25);

printf("Closed Addressing (Chaining) Hash Table:\n");

display(table);

return 0;

}

#include <stdio.h>

#include <stdlib.h>

#define INITIAL\_TABLE\_SIZE 10

#define LOAD\_FACTOR\_THRESHOLD 0.7

typedef struct {

int\* table;

int size;

int count;

} HashTable;

int hash(int key, int size) {

return key % size;

}

HashTable\* createHashTable(int size) {

HashTable\* ht = (HashTable\*)malloc(sizeof(HashTable));

ht->size = size;

ht->count = 0;

ht->table = (int\*)malloc(size \* sizeof(int));

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

ht->table[i] = -1; // Initialize table with -1 (indicating empty slots)

return ht;

}

void rehash(HashTable\* ht) {

int oldSize = ht->size;

int\* oldTable = ht->table;

ht->size \*= 2;

ht->count = 0;

ht->table = (int\*)malloc(ht->size \* sizeof(int));

for (int i = 0; i < ht->size; i++)

ht->table[i] = -1;

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

if (oldTable[i] != -1)

insert(ht, oldTable[i]);

}

free(oldTable);

}

void insert(HashTable\* ht, int key) {

if ((float)ht->count / ht->size > LOAD\_FACTOR\_THRESHOLD)

rehash(ht);

int index = hash(key, ht->size);

int i = 0;

while (ht->table[(index + i) % ht->size] != -1) {

i++;

}

ht->table[(index + i) % ht->size] = key;

ht->count++;

}

void display(HashTable\* ht) {

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

if (ht->table[i] != -1)

printf("%d -> %d\n", i, ht->table[i]);

else

printf("%d -> \n", i);

}

}

int main() {

HashTable\* ht = createHashTable(INITIAL\_TABLE\_SIZE);

insert(ht, 10);

insert(ht, 20);

insert(ht, 30);

insert(ht, 25);

insert(ht, 15); // Trigger rehashing by exceeding the load factor

insert(ht, 35);

insert(ht, 40);

printf("Rehashing Hash Table:\n");

display(ht);

free(ht->table);

free(ht);

return 0;

}

#####################################################################