

7

1 2

1 3

2 3

2 4

4 1

4 3

4 6

5 3

5 7

6 1

6 3

7 4

7 5

a

Enter the number of vertices:

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Print graph matrix

0 1 1 1 0 1 0

1 0 1 1 0 0 0

1 1 0 1 1 1 0

1 1 1 0 0 1 1

0 0 1 0 0 0 1

1 0 1 1 0 0 0

0 0 0 1 1 0 0

Print vertex's degree

1: 4 degree

2: 3 degree

3: 5 degree

4: 5 degree

5: 2 degree

6: 3 degree

7: 2 degree

#include <stdio.h>

#include <stdlib.h>

enum GraphType {ADJ\_MATRIX, ADJ\_LIST}; // Types of Graph Representation

typedef struct \_listnode

{

int vertex;

    struct \_listnode \*next;

} ListNode;

struct GraphForm{

int \*\*matrix;

ListNode \*\*list;

};

typedef struct \_graph{

int V;

int E;

enum GraphType type;

struct GraphForm adj;

}Graph;

void printGraphMatrix(Graph );

void calDegreeV(Graph,int \*);

void printDegreeV(int \*,int );

int main()

{

Graph g;

int i,j;

int\* degreeV;

printf("Enter the number of vertices:\n");

scanf("%d",&g.V);

g.E = 0;

g.adj.matrix = (int \*\*)malloc(g.V\*sizeof(int \*));

for(i=0;i<g.V;i++)

g.adj.matrix[i] = (int \*)malloc(g.V\*sizeof(int));

for(i=0;i<g.V;i++)

for(j=0;j<g.V;j++)

g.adj.matrix[i][j] = 0;

g.type = ADJ\_MATRIX;

degreeV = (int \*) malloc(g.V\*sizeof(int));

for(i=0;i<g.V;i++)

degreeV[i]=0;

int V1, V2;

printf("Enter two vertices which are adjacent to each other: (enter a to stop)\n");

while(scanf("%d %d",&V1,&V2)==2)

{

if(V1>0 && V1<=g.V && V2>0 && V2<=g.V)

{

g.adj.matrix[V1-1][V2-1] = 1;

g.adj.matrix[V2-1][V1-1] = 1;

g.E++;

}

else

break;

printf("Enter two vertices which are adjacent to each other: (enter a to stop)\n");

}

printGraphMatrix(g);

calDegreeV(g,degreeV);

printDegreeV(degreeV,g.V);

return 0;

}

void printGraphMatrix(Graph g)

{

int i,j;

printf("Print graph matrix \n");

if(g.type == ADJ\_LIST) {printf("Error"); return;}

for(i=0;i<g.V;i++){

for(j=0;j<g.V;j++)

printf("%d\t",g.adj.matrix[i][j]);

printf("\n");

}

printf("\n");

}

void calDegreeV(Graph g, int \*degreeV)

{

// Write your code here

    // Similar to lab but nested loop

int i, j;

ListNode \*temp = NULL;

for(i=0; i<g.V; i++)

    {

        degreeV[i]=0;

for(j=0; j<g.V; j++)

        {

ListNode \*temp = g.adj.matrix[i][j];

            //printf("%d \n", degreeV[i]);

if(temp != NULL)

            {

degreeV[i]++;

                temp = temp->next;

            }

}

}

}

void printDegreeV(int \*degreeV,int V)

{

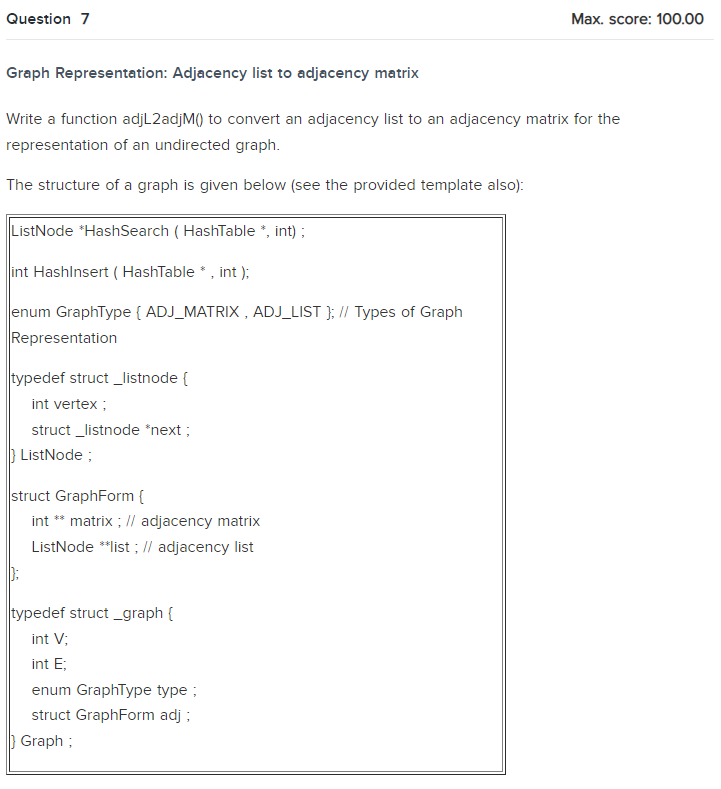
int i;

printf("Print vertex's degree \n");

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

printf("%d: %d degree\n",i+1,degreeV[i]);

}



7

1 2

1 3

2 3

2 4

4 1

4 3

4 6

5 3

5 7

6 1

6 3

7 4

7 5

a

Enter the number of vertices:

Enter two vertices which are adjacent to each other (enter a to stop):

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Enter two vertices which are adjacent to each other: (enter a to stop)

Adjacency matrix:

0 1 1 1 0 1 0

1 0 1 1 0 0 0

1 1 0 1 1 1 0

1 1 1 0 0 1 1

0 0 1 0 0 0 1

1 0 1 1 0 0 0

0 0 0 1 1 0 0

#include <stdio.h>

#include <stdlib.h>

enum GraphType {ADJ\_MATRIX, ADJ\_LIST}; // Types of Graph Representation

typedef struct \_listnode

{

int vertex;

    struct \_listnode \*next;

} ListNode;

struct GraphForm{

int \*\*matrix;

ListNode \*\*list;

};

typedef struct \_graph{

int V;

int E;

enum GraphType type;

struct GraphForm adj;

}Graph;

void printGraphMatrix(Graph );

void adjL2adjM(Graph \*);

void printGraphList(Graph );

void calDegreeV(Graph,int \*);

void printDegreeV(int \*,int );

int main()

{

Graph g;

int i,j;

int\* degreeV;

ListNode \*temp;

printf("Enter the number of vertices:\n");

scanf("%d",&g.V);

g.E = 0;

g.adj.list = (ListNode \*\*) malloc(g.V\*sizeof(ListNode \*));

g.type = ADJ\_LIST;

degreeV = (int \*) malloc(g.V\*sizeof(int));

for(i=0;i<g.V;i++)

degreeV[i]=0;

int V1, V2;

printf("Enter two vertices which are adjacent to each other (enter a to stop):\n");

while(scanf("%d %d",&V1,&V2)==2)

{

if(V1>0 && V1<=g.V && V2>0 && V2<=g.V)

{

i = V1-1;

j = V2-1;

if(g.adj.list[i]==NULL){

g.adj.list[i] = (ListNode \*)malloc(sizeof(ListNode));

g.adj.list[i]->vertex = j+1;

g.adj.list[i]->next = NULL;

            }

            else{

                temp = (ListNode \*)malloc(sizeof(ListNode));

                temp->vertex = j+1;

                temp->next =g.adj.list[i];

                g.adj.list[i] = temp;

            }

            if(g.adj.list[j]==NULL){

g.adj.list[j] = (ListNode \*)malloc(sizeof(ListNode));

g.adj.list[j]->vertex = i+1;

g.adj.list[j]->next = NULL;

            }

            else{

                temp = (ListNode \*)malloc(sizeof(ListNode));

                temp->vertex = i+1;

                temp->next =g.adj.list[j];

                g.adj.list[j] = temp;

            }

g.E++;

}

else

break;

printf("Enter two vertices which are adjacent to each other: (enter a to stop)\n");

}

adjL2adjM(&g);

printGraphMatrix(g);

return 0;

}

void printGraphMatrix(Graph g)

{

int i,j;

if(g.type == ADJ\_LIST) {printf("Error"); return;}

printf("Adjacency matrix:\n");

for(i=0;i<g.V;i++){

for(j=0;j<g.V;j++)

printf("%d\t",g.adj.matrix[i][j]);

printf("\n");

}

}

void adjL2adjM(Graph \*g){

// add your code here

    // Similar to Q23, but matrix

int i, j;

int \*\*matrix;

ListNode \*temp1, \*temp2;

if(g->type == ADJ\_MATRIX) {printf("Error"); return;}

if(g->V<=0){printf("Empty graph!"); return;}

matrix = (int \*\*)malloc(g->V\*sizeof(ListNode\*));

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

    {

        matrix[i] = (int \*)malloc(g->V\*sizeof(int));

for(j=0; j<g->V ; j++)

{

            matrix[i][j] = 0;

        }

}

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

{

temp1 = g->adj.list[i];

while(temp1!= NULL)

{

            temp2 = temp1;

matrix[(temp2->vertex)-1][i] = 1;

matrix[i][(temp2->vertex)-1] = 1;

temp1 = temp1->next;

}

}

g->type = ADJ\_MATRIX; // change representation form

g->adj.matrix = matrix;

}

void printGraphList(Graph g){

int i;

ListNode\* temp;

if(g.type == ADJ\_MATRIX) {printf("Error"); return;}

printf("Adjacency list:\n");

for(i=0;i<g.V;i++)

{

printf("%d:\t",i+1);

temp = g.adj.list[i];

while(temp!=NULL){

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

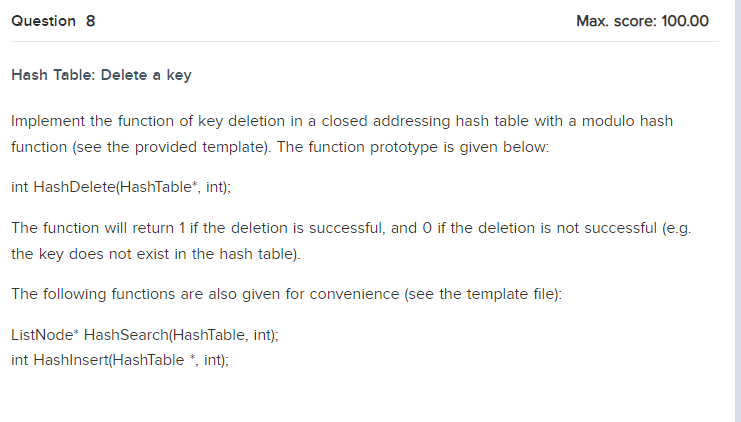
temp = temp->next;

}

printf("\n");

}

}



1

3

2

1

2

2

2

3

2

4

2

5

2

6

2

7

2

8

2

9

3

5

4

5

3

5

3

4

3

6

6

============= Hash Table ============

|1. Create a hash table |

|2. Insert a key to the hash table |

|3. Search a key in the hash table |

|4. Delete a key in the hash table |

|5. Print the hash table |

|6. Quit |

=====================================

Enter selection: Enter the size of hash table:

HashTable is created.

Enter selection: Enter a key to be inserted:

1 is inserted.

Enter selection: Enter a key to be inserted:

2 is inserted.

Enter selection: Enter a key to be inserted:

3 is inserted.

Enter selection: Enter a key to be inserted:

4 is inserted.

Enter selection: Enter a key to be inserted:

5 is inserted.

Enter selection: Enter a key to be inserted:

6 is inserted.

Enter selection: Enter a key to be inserted:

7 is inserted.

Enter selection: Enter a key to be inserted:

8 is inserted.

Enter selection: Enter a key to be inserted:

9 is inserted.

Enter selection: Enter a key for searching in the HashTable:

5 is found.

Enter selection: Enter a key to be deleted:

5 is deleted.

Enter selection: Enter a key for searching in the HashTable:

5 is not found.

Enter selection: Enter a key for searching in the HashTable:

4 is found.

Enter selection: Enter a key for searching in the HashTable:

6 is found.

Enter selection:

#include <stdio.h>

#include <stdlib.h>

typedef struct \_listnode{

int key;

struct \_listnode \*next;

} ListNode;

typedef struct \_linkedlist{

int size;

ListNode \*head;

} HashTableNode;

typedef struct \_hashTable{

int hSize; //size of hash table

int nSize; //number of inserted keys

HashTableNode \*Table;

} HashTable;

int Hash(int,int);

ListNode\* HashSearch(HashTable, int);

int HashInsert(HashTable \*, int);

int HashDelete(HashTable\*, int);

//In Practice, we will not do it

void HashPrint(HashTable);

int main()

{

int opt;

int size;

int i;

int key;

//Create a HashTable

HashTable Q3Hash;

Q3Hash.hSize = 0;

Q3Hash.nSize = 0;

Q3Hash.Table = NULL;

printf("============= Hash Table ============\n");

printf("|1. Create a hash table |\n");

printf("|2. Insert a key to the hash table |\n");

printf("|3. Search a key in the hash table |\n");

printf("|4. Delete a key in the hash table |\n");

printf("|5. Print the hash table |\n");

printf("|6. Quit |\n");

printf("=====================================\n");

printf("Enter selection: ");

scanf("%d",&opt);

while(opt>=1 && opt <=5){

switch(opt){

case 1:

printf("Enter the size of hash table:\n");

scanf("%d",&Q3Hash.hSize);

Q3Hash.nSize = 0;

Q3Hash.Table = (HashTableNode \*) malloc(sizeof(HashTableNode)\*(Q3Hash.hSize));

for(i=0;i<Q3Hash.hSize;i++){

Q3Hash.Table[i].head = NULL;

Q3Hash.Table[i].size = 0;

}

printf("HashTable is created.\n");

break;

case 2: //Insertion

printf("Enter a key to be inserted:\n");

scanf("%d",&key);

if(HashInsert(&Q3Hash,key))

printf("%d is inserted.\n",key);

else

printf("%d is a duplicate. No key is inserted.\n",key);

break;

case 3: //Search

printf("Enter a key for searching in the HashTable:\n");

scanf("%d",&key);

ListNode\* node = HashSearch(Q3Hash, key);

if(node!=NULL)

printf("%d is found.\n",key);

else

printf("%d is not found.\n",key);

break;

case 4: //Deletion

printf("Enter a key to be deleted:\n");

scanf("%d",&key);

if(HashDelete(&Q3Hash,key))

printf("%d is deleted.\n",key);

else

printf("%d is not existing.\n",key);

break;

case 5:

HashPrint(Q3Hash);

break;

}

printf("Enter selection: ");

scanf("%d",&opt);

}

for(i=0;i<Q3Hash.hSize;i++)

{

while(Q3Hash.Table[i].head)

{

ListNode \*temp;

temp = Q3Hash.Table[i].head;

Q3Hash.Table[i].head = Q3Hash.Table[i].head->next;

free(temp);

}

}

free(Q3Hash.Table);

return 0;

}

int Hash(int key,int hSize)

{

return key%hSize;

}

ListNode\* HashSearch(HashTable Q3Hash, int key)

{

int index;

ListNode \*temp;

//we may use Q3Hash.Table != NULL

if(Q3Hash.hSize!=0)

index = Hash(key,Q3Hash.hSize);

else

return NULL;

temp = Q3Hash.Table[index].head;

while(temp !=NULL){

if(temp->key == key)

return temp;

temp = temp->next;

}

return NULL;

}

int HashInsert(HashTable\* Q3HashPtr, int key)

{

int index;

ListNode \*newNode;

if(HashSearch(\*Q3HashPtr, key)!=NULL) //duplicate

return 0;

if(Q3HashPtr->hSize!=0)

index = Hash(key,Q3HashPtr->hSize);

//The key is inserted from the front. It is not the same approach discussed in lecture

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

newNode->key = key;

newNode->next= Q3HashPtr->Table[index].head;

Q3HashPtr->Table[index].head = newNode;

Q3HashPtr->Table[index].size++;

Q3HashPtr->nSize++;

return 1; //insertion is done successfully

}

int HashDelete(HashTable\* Q3HashPtr, int key)

{

// Write your code here.

if(HashSearch(\*Q3HashPtr, key) != NULL)

{

int i;

ListNode \*pre, \*temp;

        i = Hash(key, Q3HashPtr->hSize);

temp = Q3HashPtr->Table[i].head;

        // head deleted

        if(temp->key == key)

{

Q3HashPtr->Table[i].head = temp->next;

temp->key = 0;

temp->next = NULL;

free(temp);

Q3HashPtr->Table[i].size = Q3HashPtr->Table[i].size - 1;

return 1;

}

        // 1 only and deleted

else if(temp->next == NULL)

{

Q3HashPtr->Table[i].head = NULL;

Q3HashPtr->Table[i].size = 0;

return 1;

}

        // norm, relink previous with next

else

{

            // find the one we want to delete

while(temp->key != key)

{

pre = temp;

temp = temp->next;

}

            // if tail

if(temp->next == NULL)

{

pre->next = NULL;

temp->key = 0;

free(temp);

Q3HashPtr->Table[i].size = Q3HashPtr->Table[i].size - 1;

return 1;

}

            // else normal

else

{

pre->next = temp->next;

temp->key = 0;

temp->next = NULL;

free(temp);

Q3HashPtr->Table[i].size = Q3HashPtr->Table[i].size - 1;

return 1;

}

}

}

return 0;

}

void HashPrint(HashTable Q3Hash)

{

int i;

ListNode \*temp;

for(i=0;i<Q3Hash.hSize;i++)

{

temp =Q3Hash.Table[i].head;

printf("%d: ",i);

while(temp !=NULL)

{

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

temp = temp->next;

}

printf("\n");

}

}