LABORATORY EXPERIMENTS

1. Implementation of randomized quicksort algorithm

2. Implementation of hash functions and associated algorithms

3. Implementation of operations on splay trees

4. Implementation of operations on Fibonacci heaps

5. Implementation of operations on binary heaps

6. Implementation on operations on B-Trees

7. Implementation of operations on partition ADT and union-find data structures

8. Implementation of Bellman-Ford algorithm

9. Implementation of Ford-Fulkerson algorithm

10. Implementation of Edmonds-Karp algorithm

1. Randomized Quick sort

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

int PARTITION(int [], int, int);
void R_QUICKSORT(int [], int, int);
int R_PARTITION(int [], int, int);

int main() {
  int n;
  printf("Enter the size of the array\n");
  scanf("%d", &n);

  int a[n],i;
  printf("Enter the elements in the array\n");
  for ( i = 0; i < n; i++) {
    scanf("%d", &a[i]);
  }

  printf("Sorting using randomized quicksort\n");
  int p = 0, r = n - 1;

  R_QUICKSORT(a, p, r);

  printf("Sorted form\n");
  for (i = 0; i < n; i++) {
    printf("%d\n", a[i]);
  }

  return 0;
}

void R_QUICKSORT(int a[], int p, int r) {
  int q;
  if (p < r) {
    q = R_PARTITION(a, p, r);
    R_QUICKSORT(a, p, q - 1);
    R_QUICKSORT(a, q + 1, r);
  }
}

int R_PARTITION(int a[], int p, int r) {
  int i = p + rand() % (r - p + 1);
  int temp;
```

```
    temp = a[r];
    a[r] = a[i];
    a[i] = temp;
    return PARTITION(a, p, r);
}

int PARTITION(int a[], int p, int r) {
    int temp, temp1,j;
    int x = a[r];
    int i = p - 1;
    for ( j = p; j <= r - 1; j++) {
        if (a[j] <= x) {
            i = i + 1;
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }
    temp1 = a[i + 1];
    a[i + 1] = a[r];
    a[r] = temp1;
    return i + 1;
}
```

2. Hash Function:

```
#include <stdio.h>
#include <conio.h>
int tsize;

int hasht(int key)
{
 int i ;
 i = key%tsize ;
 return i;
}

//-------LINEAR PROBING-------
int rehashl(int key)
{
 int i ;
 i = (key+1)%tsize ;
 return i ;
}

//-------QUADRATIC PROBING-------
int rehashq(int key, int j)
{
 int i ;
 i = (key+(j*j))%tsize ;
 return i ;
}

void main()
{
    int key,arr[20],hash[20],i,n,s,op,j,k ;

    printf ("Enter the size of the hash table:  ");
    scanf ("%d",&tsize);

    printf ("\nEnter the number of elements: ");
    scanf ("%d",&n);

    for (i=0;i<tsize;i++)
 hash[i]=-1 ;

    printf ("Enter Elements: ");
    for (i=0;i<n;i++)
    {
```

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

  do
  {
printf("\n\n1.Linear Probing\n2.Quadratic Probing \n3.Exit \nEnter your option: ");
scanf("%d",&op);
switch(op)
{
case 1:
   for (i=0;i<tsize;i++)
   hash[i]=-1 ;

   for(k=0;k<n;k++)
   {
 key=arr[k] ;
 i = hasht(key);
 while (hash[i]!=-1)
 {
    i = rehashl(i);
 }
 hash[i]=key ;
   }
   printf("\nThe elements in the array are: ");
   for (i=0;i<tsize;i++)
   {
 printf("\n  Element at position %d: %d",i,hash[i]);
   }
   break ;

case 2:
   for (i=0;i<tsize;i++)
 hash[i]=-1 ;

   for(k=0;k<n;k++)
   {
 j=1;
 key=arr[k] ;
 i = hasht(key);
 while (hash[i]!=-1)
 {
    i = rehashq(i,j);
    j++ ;
 }
 hash[i]=key ;
   }
   printf("\nThe elements in the array are: ");
   for (i=0;i<tsize;i++)
   {
 printf("\n Element at position %d: %d",i,hash[i]);
   }
   break ;

}
  }while(op!=3);

  getch() ;
}
```

Output:

```
Enter the size of the hash table:  10

Enter the number of elements: 8
Enter Elements: 72
27
36
24
```

63
81
92
101

1.Linear Probing
2.Quadratic Probing
3.Exit
Enter your option: 1

The elements in the array are:
 Element at position 0: -1
 Element at position 1: 81
 Element at position 2: 72
 Element at position 3: 63
 Element at position 4: 24
 Element at position 5: 92
 Element at position 6: 36
 Element at position 7: 27
 Element at position 8: 101
 Element at position 9: -1

1.Linear Probing
2.Quadratic Probing
3.Exit
Enter your option: 2

The elements in the array are:
 Element at position 0: -1
 Element at position 1: 81
 Element at position 2: 72
 Element at position 3: 63
 Element at position 4: 24
 Element at position 5: 101
 Element at position 6: 36
 Element at position 7: 27
 Element at position 8: 92
 Element at position 9: -1

1.Linear Probing
2.Quadratic Probing
3.Exit
Enter your option: 3

4.Fibonacci Heap

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<math.h>
#include<malloc.h>
#define NIL 0
 int nNodes;

/* structure of a node */

struct fheap_node
{
struct fheap_node *parent;
struct fheap_node *left;
struct fheap_node *right;
struct fheap_node *child;
int degree;
int mark;
int key; };
struct fheap_node *min;

/* creating fibonacci heap */
```

```
void create_fib()
{
min=NULL;
nNodes=0;
}

/* inserting in fibonacci heap */

void Finsert(int val)
{
struct fheap_node *fheap;
fheap=malloc(sizeof(struct fheap_node));
fheap->key=val;
fheap->parent=NULL;
fheap->left=NULL;
fheap->right=NULL;
fheap->child=NULL;
if(min!=NULL)
{
fheap->right=min;
fheap->left=min->left;
(min)->left=fheap;
(fheap->left)->right=fheap;

if(val<min->key)
min=fheap;
}
else
{
min=fheap;
min->left=min;
min->right=min;
}
nNodes++;
}

/* Displaying Fibonacci heap*/

void display(struct fheap_node *min)
{
struct fheap_node *q=min;
if(q==NIL)
{
printf("\n Fibonacci heap is empty");
return;
}
q=min;
printf("\n Fibonacci heap display");
do
{
printf("\t%d ",q->key);
if(q->child!=NIL)
{
display(q->child);
}
q=q->right;
}
while(q!=min);
}

/* finding minimum key in heap */

void findmin()
{
        if(min!=NULL)
printf("\nminimum is %d: ",min->key);
else
printf("\n Fibonacci heap is empty");
```

```
}
int main ()
{
int option,n;
create_fib();
min=NIL;
while(1)
{printf("\nmenu\n");
printf("1:create fibonacci heap\n");
printf("2:insert in fibonacci heap\n");
printf("3: find min in fibonacci heap \n");
printf("4:display\n");
printf("5: exit \n");
scanf ("%d",&option);
switch(option)
{
case 1 :create_fib();
break;
case 2: printf("\nenter the element= ");
scanf("%d",&n);
Finsert(n);
break;
case 3: findmin();
break;
case 4: display(min);
break;
case 5 :exit(0);
default: printf("\nwrong choice... try again \n ");
  }
 }
 return 0;
}
```

<u>OUTPU T :</u>

```
menu
1 : create fibonacci heap
2 : insert in fibonacci heap
3: find min in fibonacci heap
4 : display
5: exit
1

menu
1: create fibonacci heap
2: insert in fibonacci heap
3: find min in fibonacci heap
4: display
5: exit
2

enter the element= 2

menu
1: create fibonacci heap
2: insert in fibonacci heap
3: find min in fibonacci heap
4: display
5: exit
2

enter the element= 3

menu
1: create fibonacci heap
2: insert in fibonacci heap
3: find min in fibonacci heap
4: display
5: exit
```

4

Fibonacci heap display 2 3
menu
1: create fibonacci heap
2: insert in fibonacci heap
3: find min in fibonacci heap
4: display
5: exit
2

enter the element= 10

menu
1: create fibonacci heap
2: insert in fibonacci heap
3: find min in fibonacci heap
4: display
5: exit
4

Fibonacci heap display 2 3 10
menu
1: create fibonacci heap
2: insert in fibonacci heap
3: find min in fibonacci heap
4: display
5: exit
3

minimum is 2:
menu
1: create fibonacci heap
2: insert in fibonacci heap
3: find min in fibonacci heap
4: display
5: exit
2

enter the element= 1

menu
1: create fibonacci heap
2: insert in fibonacci heap
3: find min in fibonacci heap
4: display
5: exit
4

Fibonacci heap display 1 2 3 10
menu
1: create fibonacci heap
2: Insert in fibonacci heap
3: find min in fibonacci heap
4: display
5: exit
3

minimum is 1:
menu
1: create fibonacci heap
2: insert in fibonacci heap
3: find min in fibonacci heap
4: display
5: exit

5. Binary Heap :

```
/* C program to build a binary heap */
#include <stdio.h>
#include <stdlib.h>
#define MAX 20
void maxheapify(int *, int, int);
int* buildmaxheap(int *, int);
void main()
{
  int i, t, n;
        int *a = calloc(MAX, sizeof(int));
  int *m = calloc(MAX, sizeof(int));
  printf("Enter no of elements in the array\n");
  scanf("%d", &n);
  printf("Enter the array\n");
  for (i = 0; i < n; i++) {
    scanf("%d", &a[i]);
  }
  m = buildmaxheap(a, n);
  printf("The  Binary heap is\n");
  for (t = 0; t < n; t++) {
    printf("%d\n", m[t]);
  }
}
int* buildmaxheap(int a[], int n)
{
  int heapsize = n;
  int j;
  for (j = n/2; j >= 0; j--) {
    maxheapify(a, j, heapsize);
  }
  return a;
}
void maxheapify(int a[], int i, int heapsize)
{
  int temp, largest, left, right, k;
  left = (2*i+1);
  right = ((2*i)+2);
  if (left >= heapsize)
    return;
  else {
    if (left < (heapsize) && a[left] > a[i])
      largest = left;
    else
      largest = i;
    if (right < (heapsize) && a[right] > a[largest])
      largest = right;
    if (largest != i) {
      temp = a[i];
      a[i] = a[largest];
      a[largest] = temp;
      maxheapify(a, largest, heapsize);
    }
  }
}
```

OUTPUT:

```
Enter no of elements in the array
5
Enter the array
20
70
10
90
50
```

The Binay heap is
90
70
10
20
50

6. **ADT and union-find data structures**

```
#include <stdio.h>

#define MAX_SIZE 100

int parent[MAX_SIZE];

// Initialize disjoint sets
void initialize(int n) {
    int i;
    for (i = 1; i <= n; i++) {
        parent[i] = i;  // Each node is its own parent initially
    }
}

// Find operation with path compression
int find(int x) {
    if (parent[x] != x) {
        parent[x] = find(parent[x]); // Path compression
    }
    return parent[x];
}

// Union operation
void unionSets(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);
    if (rootX != rootY) {
        parent[rootX] = rootY; // Make rootY the parent of rootX
    }
}

int main() {
    int n; // Number of elements
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    if (n > MAX_SIZE) {
        printf("Number of elements exceeds the maximum size (%d).\n", MAX_SIZE);
        return 1;
    }

    initialize(n);

    // Perform some union operations
    unionSets(0, 1);
    unionSets(2, 3);
    unionSets(0, 2);

    // Test find operation
    printf("Find(1) = %d\n", find(1)); // Should print the root of the set containing 1
    printf("Find(3) = %d\n", find(3)); // Should print the root of the set containing 3

    return 0;
}
```

OUTPUT:

Enter the number of elements: 3
Find(1) = 3

Find(3) = 3

8.Bellman Ford

```
#include<stdlib.h>
#include<conio.h>
#include<string.h>
#include<limits.h>

struct Edges

{

// This structure is equal to an edge. Edge contains two endpoints. These edges are directed edges so they

//contain source and destination and some weight. These 3 are elements in this structure

int src, dest, wt;

};

// a structure to represent a graph

struct Graph

{

int Vertex, Edge;

//Vertex is the number of vertices, and Edge is the number of edges

struct Edges* edge;

// This structure contains another structure that we have already created.

};

struct Graph* designGraph(int Vertex, int Edge)

{

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

//Allocating space to structure graph

graph->Vertex = Vertex; //assigning values to structure elements that taken form user.

graph->Edge = Edge;

graph->edge = (struct Edges*) malloc( graph->Edge * sizeof( struct Edges ) );

//Creating "Edge" type structures inside "Graph" structure, the number of edge type structures are equal to number
of edges

return graph;

}

void Solution(int dist[], int n)

{

// This function prints the last solution

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

int i;

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

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

}

}

void BellmanFordalgorithm(struct Graph* graph, int src)

{

int Vertex = graph->Vertex;

int Edge = graph->Edge;

int Distance[Vertex];

int i,j;

// This is the initial step that we know, and we initialize all distances to infinity except the source vertex.

// We assign source distance as 0

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

Distance[i] = INT_MAX;

Distance[src] = 0;

//The shortest path of graph that contain Vertex vertices, never contain "Veretx-1" edges. So we do here "Vertex-1" relaxations

for (i = 1; i <= Vertex-1; i++)

{

for (j = 0; j < Edge; j++)

{

int u = graph->edge[j].src;

int v = graph->edge[j].dest;

int wt = graph->edge[j].wt;

if (Distance[u] + wt < Distance[v])

Distance[v] = Distance[u] + wt;

}

}

//, up to now, the shortest path found. But BellmanFordalgorithm checks for negative edge cycles. In this step, we check for that

// shortest path if the graph doesn't contain any negative weight cycle in the graph.

// If we get a shorter path, then there is a negative edge cycle.

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

{

int u = graph->edge[i].src;

int v = graph->edge[i].dest;
```

```
int wt = graph->edge[i].wt;

if (Distance[u] + wt < Distance[v])

printf("This graph contains negative edge cycle\n");

}

Solution(Distance, Vertex);

return;

}

int main()

{

int V,E,S; //V = no.of Vertices, E = no.of Edges, S is source vertex

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

scanf("%d",&V);

printf("Enter number of edges\n");

scanf("%d",&E);

printf("Enter the source vertex number\n");

scanf("%d",&S);

struct Graph* graph = designGraph(V, E); //calling the function to allocate space to these many vertices and edges

int i;

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

printf("\nEnter edge %d properties Source, destination, weight respectively\n",i+1);

scanf("%d",&graph->edge[i].src);

scanf("%d",&graph->edge[i].dest);

scanf("%d",&graph->edge[i].wt);

}

BellmanFordalgorithm(graph, S);

//passing created graph and source vertex to BellmanFord Algorithm function

return 0;

}
```

OUTPUT:

```
Enter number of vertices
4
Enter number of edges
5
Enter the source vertex number
0

Enter edge 1 properties Source, destination, weight respectively
0 1 5

Enter edge 2 properties Source, destination, weight respectively
```

0 1 6

Enter edge 3 properties Source, destination, weight respectively
1 2 7

Enter edge 4 properties Source, destination, weight respectively
1 4 -5

Enter edge 5 properties Source, destination, weight respectively
1 3 6

Vertex Distance from Source Vertex
0 0
1 5
2 12
3 11

9.Ford Fulkerson

```
#include<stdio.h>

#include<stdlib.h>

#include<conio.h>

#include<string.h>

#include<limits.h>

#define A 0

#define B 1

#define C 2

#define MAX_NODES 1000

#define O 1000000000

int n;

int e;

int capacity[MAX_NODES][MAX_NODES];

int flow[MAX_NODES][MAX_NODES];

int color[MAX_NODES];

int pred[MAX_NODES];

int min(int x, int y) {

return x < y ? x : y;

}

int head, tail;

int q[MAX_NODES + 2];

void enqueue(int x) {

q[tail] = x;

tail++;

color[x] = B;
```

```
}
int dequeue() {
int x = q[head];
head++;
color[x] = C;
return x;
}
// Using BFS as a searching algorithm
int bfs(int start, int target) {
int u, v;
for (u = 0; u < n; u++) {
color[u] = A;
}
head = tail = 0;
enqueue(start);
pred[start] = -1;
while (head != tail) {
u = dequeue();
for (v = 0; v < n; v++) {
if (color[v] == A && capacity[u][v] - flow[u][v] > 0) {
enqueue(v);
pred[v] = u;
}
}
}
return color[target] == C;
}
// Applying fordfulkerson algorithm
int fordFulkerson(int source, int sink) {
int i, j, u;
int max_flow = 0;
for (i = 0; i < n; i++) {
for (j = 0; j < n; j++) {
flow[i][j] = 0;
}
```

```
}
// Updating the residual values of edges
while (bfs(source, sink)) {
int increment = O;
for (u = n - 1; pred[u] >= 0; u = pred[u]) {
increment = min(increment, capacity[pred[u]][u] - flow[pred[u]][u]);
}
for (u = n - 1; pred[u] >= 0; u = pred[u]) {
flow[pred[u]][u] += increment;
flow[u][pred[u]] -= increment;
}
// Adding the path flows
max_flow += increment;
}
return max_flow;
}
int main() {
int i, j;
printf("Enter the number of nodes and edges:\n");
scanf("%d%d", &n, &e);
// Initialize capacity matrix
for (i = 0; i < n; i++) {
for (j = 0; j < n; j++) {
capacity[i][j] = 0;
}
}
printf("Enter the edges and their capacities:\n");
for (i = 0; i < e; i++) {
int u, v, cap;
scanf("%d%d%d", &u, &v, &cap);
capacity[u][v] = cap;
}
int source, sink;
printf("Enter the source and sink nodes:\n");
```

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

printf("Max Flow: %d\n", fordFulkerson(source, sink));

return 0;

}
```

OUTPUT:

```
Enter the number of nodes and edges:
2 1
Enter the edges and their capacities:
0 1 5
Enter the source and sink nodes:
0 1
Max Flow: 5
```

10.E-karp

```
#include <stdio.h>

#include <stdbool.h>

#include <string.h>

#define MAX_NODES 100

int capacities[MAX_NODES][MAX_NODES];

int flowPassed[MAX_NODES][MAX_NODES];

int graph[MAX_NODES][MAX_NODES];

int parentsList[MAX_NODES];

int currentPathCapacity[MAX_NODES];

int i;

int bfs(int startNode, int endNode, int nodesCount)

{

memset(parentsList, -1, sizeof(parentsList));

memset(currentPathCapacity, 0, sizeof(currentPathCapacity));

int queue[MAX_NODES];

int front = 0, rear = 0;

bool visited[MAX_NODES];

memset(visited, false, sizeof(visited));

queue[rear++] = startNode;

parentsList[startNode] = -2;

currentPathCapacity[startNode] = 999;

while (front != rear)

{

int currentNode = queue[front++];

visited[currentNode] = true;
```

```
for (i= 0; i < nodesCount; i++)

{

if (!visited[i] && capacities[currentNode][i] - flowPassed[currentNode][i] > 0)

{

parentsList[i] = currentNode;

currentPathCapacity[i] = (currentPathCapacity[currentNode] < capacities[currentNode][i] -
flowPassed[currentNode][i]) ? currentPathCapacity[currentNode] : capacities[currentNode][i] -
flowPassed[currentNode][i];

if (i == endNode)

{

return currentPathCapacity[endNode];

}

queue[rear++] = i;

}

}

}

return 0;

}

int edmondsKarp(int startNode, int endNode, int nodesCount)

{

int maxFlow = 0;

while (true)

{

int flow = bfs(startNode, endNode, nodesCount);

if (flow == 0)

{

break;

}

maxFlow += flow;

int currentNode = endNode;

while (currentNode != startNode)

{

int previousNode = parentsList[currentNode];

flowPassed[previousNode][currentNode] += flow;

flowPassed[currentNode][previousNode] -= flow;
```

```
currentNode = previousNode;

}

}

return maxFlow;

}

int main()

{

int nodesCount, edgesCount;

printf("Enter the number of nodes and edges: ");

scanf("%d%d", &nodesCount, &edgesCount);

int source, sink,edge;

printf("Enter the source and sink: ");

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

for ( edge = 0; edge < edgesCount; edge++)

{

printf("Enter the start and end vertex along with capacity: ");

int from, to, capacity;

scanf("%d%d%d", &from, &to, &capacity);

capacities[from][to] = capacity;

graph[from][to] = 1;

graph[to][from] = 1;

}

int maxFlow = edmondsKarp(source, sink, nodesCount);

printf("\n\nMax Flow is: %d\n", maxFlow);

return 0;

}
```

OUTPUT:

```
Enter the number of nodes and edges:  4 5
Enter the source and sink: 0 3
Enter the start and end vertex along with capacity: 0 1 10
Enter the start and end vertex along with capacity: 0 2 5
Enter the start and end vertex along with capacity: 1 2 15
Enter the start and end vertex along with capacity: 1 3 10
Enter the start and end vertex along with capacity: 2 3 10


Max Flow is: 15
```