1) Impliment Binomial Heaps and operations (Create, Insert, Delete)

Source code

#include<stdio.h>

#include<malloc.h>

struct node

{

int n;

int degree;

struct node\* parent;

struct node\* child;

struct node\* sibling;

};

struct node\* MAKE\_bin\_HEAP();

int bin\_LINK(struct node\*, struct node\*);

struct node\* CREATE\_NODE(int);

struct node\* bin\_HEAP\_UNION(struct node\*, struct node\*);

struct node\* bin\_HEAP\_INSERT(struct node\*, struct node\*);

struct node\* bin\_HEAP\_MERGE(struct node\*, struct node\*);

struct node\* bin\_HEAP\_EXTRACT\_MIN(struct node\*);

int REVERT\_LIST(struct node\*);

int DISPLAY(struct node\*);

struct node\* FIND\_NODE(struct node\*, int);

int bin\_HEAP\_DECREASE\_KEY(struct node\*, int, int);

int bin\_HEAP\_DELETE(struct node\*, int);

int count = 1;

struct node\* MAKE\_bin\_HEAP()

{

struct node\* np;

np = NULL;

return np;

}

struct node \* H = NULL;

struct node \*Hr = NULL;

int bin\_LINK(struct node\* y, struct node\* z)

{

y->parent = z;

y->sibling = z->child;

z->child = y;

z->degree = z->degree + 1;

}

struct node\* CREATE\_NODE(int k)

{

struct node\* p;//new node;

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

p->n = k;

return p;

}

struct node\* bin\_HEAP\_UNION(struct node\* H1, struct node\* H2)

{

struct node\* prev\_x;

struct node\* next\_x;

struct node\* x;

struct node\* H = MAKE\_bin\_HEAP();

H = bin\_HEAP\_MERGE(H1, H2);

if (H == NULL)

return H;

prev\_x = NULL;

x = H;

next\_x = x->sibling;

while (next\_x != NULL)

{

if ((x->degree != next\_x->degree) || ((next\_x->sibling != NULL)

&& (next\_x->sibling)->degree == x->degree))

{

prev\_x = x;

x = next\_x;

}

else

{

if (x->n <= next\_x->n)

{

x->sibling = next\_x->sibling;

bin\_LINK(next\_x, x);

}

else

{

if (prev\_x == NULL)

H = next\_x;

else

prev\_x->sibling = next\_x;

bin\_LINK(x, next\_x);

x = next\_x;

}

}

next\_x = x->sibling;

}

return H;

}

struct node\* bin\_HEAP\_INSERT(struct node\* H, struct node\* x)

{

struct node\* H1 = MAKE\_bin\_HEAP();

x->parent = NULL;

x->child = NULL;

x->sibling = NULL;

x->degree = 0;

H1 = x;

H = bin\_HEAP\_UNION(H, H1);

return H;

}

struct node\* bin\_HEAP\_MERGE(struct node\* H1, struct node\* H2)

{

struct node\* H = MAKE\_bin\_HEAP();

struct node\* y;

struct node\* z;

struct node\* a;

struct node\* b;

y = H1;

z = H2;

if (y != NULL)

{

if (z != NULL && y->degree <= z->degree)

H = y;

else if (z != NULL && y->degree > z->degree)

/\* need some modifications here;the first and the else conditions

can be merged together!!!! \*/

H = z;

else

H = y;

} else

H = z;

while (y != NULL && z != NULL)

{

if (y->degree < z->degree)

{

y = y->sibling;

}

else if (y->degree == z->degree)

{

a = y->sibling;

y->sibling = z;

y = a;

} else

{

b = z->sibling;

z->sibling = y;

z = b;

}

}

return H;

}

int DISPLAY(struct node\* H)

{

struct node\* p;

if (H == NULL)

{

printf("\nHEAP EMPTY");

return 0;

}

printf("\nTHE ROOT NODES ARE:-\n");

p = H;

while (p != NULL)

{

printf("%d", p->n);

if (p->sibling != NULL)

printf("-->");

p = p->sibling;

}

printf("\n");

}

struct node\* bin\_HEAP\_EXTRACT\_MIN(struct node\* H1)

{

int min;

struct node\* t = NULL;

struct node\* x = H1;

struct node \*Hr;

struct node\* p;

Hr = NULL;

if (x == NULL)

{

printf("\nNOTHING TO EXTRACT");

return x;

}

// int min=x->n;

p = x;

while (p->sibling != NULL)

{

if ((p->sibling)->n < min)

{

min = (p->sibling)->n;

t = p;

x = p->sibling;

}

p = p->sibling;

}

if (t == NULL && x->sibling == NULL)

H1 = NULL;

else if (t == NULL)

H1 = x->sibling;

else if (t->sibling == NULL)

t = NULL;

else

t->sibling = x->sibling;

if (x->child != NULL)

{

REVERT\_LIST(x->child);

(x->child)->sibling = NULL;

}

H = bin\_HEAP\_UNION(H1, Hr);

return x;

}

int REVERT\_LIST(struct node\* y)

{

if (y->sibling != NULL)

{

REVERT\_LIST(y->sibling);

(y->sibling)->sibling = y;

} else

{

Hr = y;

}

}

struct node\* FIND\_NODE(struct node\* H, int k)

{

struct node\* x = H;

struct node\* p = NULL;

if (x->n == k)

{

p = x;

return p;

}

if (x->child != NULL && p == NULL)

{

p = FIND\_NODE(x->child, k);

}

if (x->sibling != NULL && p == NULL)

{

p = FIND\_NODE(x->sibling, k);

}

return p;

}

int bin\_HEAP\_DECREASE\_KEY(struct node\* H, int i, int k)

{

int temp;

struct node\* p;

struct node\* y;

struct node\* z;

p = FIND\_NODE(H, i);

if (p == NULL)

{

printf("\nINVALID CHOICE OF KEY TO BE REDUCED");

return 0;

}

if (k > p->n)

{

printf("\nSORY!THE NEW KEY IS GREATER THAN CURRENT ONE");

return 0;

}

p->n = k;

y = p;

z = p->parent;

while (z != NULL && y->n < z->n)

{

temp = y->n;

y->n = z->n;

z->n = temp;

y = z;

z = z->parent;

}

printf("\nKEY REDUCED SUCCESSFULLY!");

}

int bin\_HEAP\_DELETE(struct node\* H, int k)

{

struct node\* np;

if (H == NULL)

{

printf("\nHEAP EMPTY");

return 0;

}

bin\_HEAP\_DECREASE\_KEY(H, k, -1000);

np = bin\_HEAP\_EXTRACT\_MIN(H);

if (np != NULL)

printf("\nNODE DELETED SUCCESSFULLY");

}

int main()

{

int i, n, m, l;

struct node\* p;

struct node\* np;

char ch;

printf("\nENTER THE NUMBER OF ELEMENTS:");

scanf("%d", &n);

printf("\nENTER THE ELEMENTS:\n");

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

{

scanf("%d", &m);

np = CREATE\_NODE(m);

H = bin\_HEAP\_INSERT(H, np);

}

DISPLAY(H);

do

{

printf("\nMENU:-\n");

printf(

"\n1)INSERT AN ELEMENT\n2)EXTRACT THE MINIMUM KEY NODE\n3)DECREASE A NODE KEY\n4)DELETE A NODE\n5)QUIT\n");

printf("ENTER YOUR CHOICE:");

scanf("%d", &l);

switch (l)

{

case 1:

do

{

printf("\nENTER THE ELEMENT TO BE INSERTED:");

scanf("%d", &m);

p = CREATE\_NODE(m);

H = bin\_HEAP\_INSERT(H, p);

printf("\nNOW THE HEAP IS:\n");

DISPLAY(H);

printf("\nINSERT MORE(y/Y)= \n");

fflush(stdin);

scanf("%c", &ch);

} while (ch == 'Y' || ch == 'y');

break;

case 2:

do

{

printf("\nEXTRACTING THE MINIMUM KEY NODE");

p = bin\_HEAP\_EXTRACT\_MIN(H);

if (p != NULL)

printf("\nTHE EXTRACTED NODE IS %d", p->n);

printf("\nNOW THE HEAP IS:\n");

DISPLAY(H);

printf("\nEXTRACT MORE(y/Y)\n");

fflush(stdin);

scanf("%c", &ch);

} while (ch == 'Y' || ch == 'y');

break;

case 3:

do

{

printf("\nENTER THE KEY OF THE NODE TO BE DECREASED:");

scanf("%d", &m);

printf("\nENTER THE NEW KEY : ");

scanf("%d", &l);

bin\_HEAP\_DECREASE\_KEY(H, m, l);

printf("\nNOW THE HEAP IS:\n");

DISPLAY(H);

printf("\nDECREASE MORE(y/Y)\n");

fflush(stdin);

scanf("%c", &ch);

} while (ch == 'Y' || ch == 'y');

break;

case 4:

do

{

printf("\nENTER THE KEY TO BE DELETED: ");

scanf("%d", &m);

bin\_HEAP\_DELETE(H, m);

printf("\nDELETE MORE(y/Y)\n");

fflush(stdin);

scanf("%c", &ch);

} while (ch == 'y' || ch == 'Y');

break;

case 5:

printf("\nExiting....\n");

break;

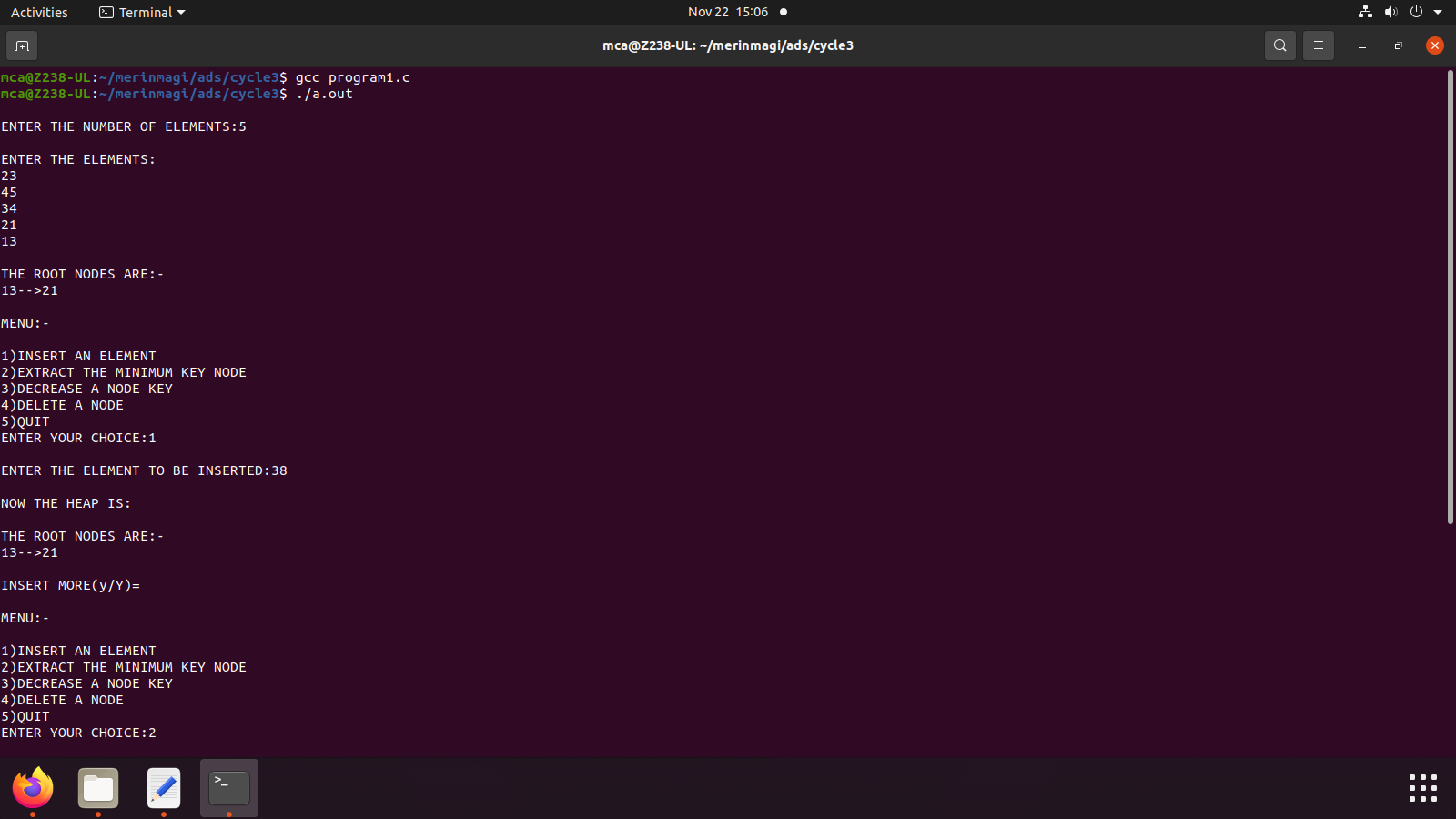
default:

printf("\nINVALID ENTRY...TRY AGAIN....\n");

}

} while (l != 5);

}



2) Impliment B Trees and its operations

Source code

#include <stdio.h>

#include <stdlib.h>

#define MAX 4

#define MIN 2

struct btreeNode

{

int val[MAX + 1], count;

struct btreeNode \*link[MAX + 1];

};

struct btreeNode \*root;

/\* creating new node \*/

struct btreeNode \* createNode(int val, struct btreeNode \*child)

{

struct btreeNode \*newNode;

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

newNode->val[1] = val;

newNode->count = 1;

newNode->link[0] = root;

newNode->link[1] = child;

return newNode;

}

/\* Places the value in appropriate position \*/

void addValToNode(int val, int pos, struct btreeNode \*node,

struct btreeNode \*child)

{

int j = node->count;

while (j > pos)

{

node->val[j + 1] = node->val[j];

node->link[j + 1] = node->link[j];

j--;

}

node->val[j + 1] = val;

node->link[j + 1] = child;

node->count++;

}

/\* split the node \*/

void splitNode (int val, int \*pval, int pos, struct btreeNode \*node,

struct btreeNode \*child, struct btreeNode \*\*newNode)

{

int median, j;

if (pos > MIN)

median = MIN + 1;

else

median = MIN;

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

j = median + 1;

while (j <= MAX)

{

(\*newNode)->val[j - median] = node->val[j];

(\*newNode)->link[j - median] = node->link[j];

j++;

}

node->count = median;

(\*newNode)->count = MAX - median;

if (pos <= MIN)

{

addValToNode(val, pos, node, child);

}

else

{

addValToNode(val, pos - median, \*newNode, child);

}

\*pval = node->val[node->count];

(\*newNode)->link[0] = node->link[node->count];

node->count--;

}

/\* sets the value val in the node \*/

int setValueInNode(int val, int \*pval,

struct btreeNode \*node, struct btreeNode \*\*child)

{

int pos;

if (!node)

{

\*pval = val;

\*child = NULL;

return 1;

}

if (val < node->val[1])

{

pos = 0;

}

else

{

for (pos = node->count;

(val < node->val[pos] && pos > 1); pos--);

if (val == node->val[pos])

{

printf("Duplicates not allowed\n");

return 0;

}

}

if (setValueInNode(val, pval, node->link[pos], child))

{

if (node->count < MAX)

{

addValToNode(\*pval, pos, node, \*child);

} else

{

splitNode(\*pval, pval, pos, node, \*child, child);

return 1;

}

}

return 0;

}

/\* insert val in B-Tree \*/

void insertion(int val)

{

int flag, i;

struct btreeNode \*child;

flag = setValueInNode(val, &i, root, &child);

if (flag)

root = createNode(i, child);

}

/\* copy successor for the value to be deleted \*/

void copySuccessor(struct btreeNode \*myNode, int pos)

{

struct btreeNode \*dummy;

dummy = myNode->link[pos];

for (;dummy->link[0] != NULL;)

dummy = dummy->link[0];

myNode->val[pos] = dummy->val[1];

}

/\* removes the value from the given node and rearrange values \*/

void removeVal(struct btreeNode \*myNode, int pos)

{

int i = pos + 1;

while (i <= myNode->count) {

myNode->val[i - 1] = myNode->val[i];

myNode->link[i - 1] = myNode->link[i];

i++;

}

myNode->count--;

}

/\* shifts value from parent to right child \*/

void doRightShift(struct btreeNode \*myNode, int pos)

{

struct btreeNode \*x = myNode->link[pos];

int j = x->count;

while (j > 0) {

x->val[j + 1] = x->val[j];

x->link[j + 1] = x->link[j];

}

x->val[1] = myNode->val[pos];

x->link[1] = x->link[0];

x->count++;

x = myNode->link[pos - 1];

myNode->val[pos] = x->val[x->count];

myNode->link[pos] = x->link[x->count];

x->count--;

return;

}

/\* shifts value from parent to left child \*/

void doLeftShift(struct btreeNode \*myNode, int pos)

{

int j = 1;

struct btreeNode \*x = myNode->link[pos - 1];

x->count++;

x->val[x->count] = myNode->val[pos];

x->link[x->count] = myNode->link[pos]->link[0];

x = myNode->link[pos];

myNode->val[pos] = x->val[1];

x->link[0] = x->link[1];

x->count--;

while (j <= x->count) {

x->val[j] = x->val[j + 1];

x->link[j] = x->link[j + 1];

j++;

}

return;

}

/\* merge nodes \*/

void mergeNodes(struct btreeNode \*myNode, int pos)

{

int j = 1;

struct btreeNode \*x1 = myNode->link[pos], \*x2 = myNode->link[pos - 1];

x2->count++;

x2->val[x2->count] = myNode->val[pos];

x2->link[x2->count] = myNode->link[0];

while (j <= x1->count)

{

x2->count++;

x2->val[x2->count] = x1->val[j];

x2->link[x2->count] = x1->link[j];

j++;

}

j = pos;

while (j < myNode->count)

{

myNode->val[j] = myNode->val[j + 1];

myNode->link[j] = myNode->link[j + 1];

j++;

}

myNode->count--;

free(x1);

}

/\* adjusts the given node \*/

void adjustNode(struct btreeNode \*myNode, int pos)

{

if (!pos) {

if (myNode->link[1]->count > MIN)

{

doLeftShift(myNode, 1);

} else

{

mergeNodes(myNode, 1);

}

} else

{

if (myNode->count != pos)

{

if(myNode->link[pos - 1]->count > MIN)

{

doRightShift(myNode, pos);

} else

{

if (myNode->link[pos + 1]->count > MIN)

{

doLeftShift(myNode, pos + 1);

} else

{

mergeNodes(myNode, pos);

}

}

} else

{

if (myNode->link[pos - 1]->count > MIN)

doRightShift(myNode, pos);

else

mergeNodes(myNode, pos);

}

}

}

/\* delete val from the node \*/

int delValFromNode(int val, struct btreeNode \*myNode)

{

int pos, flag = 0;

if (myNode) {

if (val < myNode->val[1])

{

pos = 0;

flag = 0;

} else

{

for (pos = myNode->count;

(val < myNode->val[pos] && pos > 1); pos--);

if (val == myNode->val[pos])

{

flag = 1;

}

else

{

flag = 0;

}

}

if (flag)

{

if (myNode->link[pos - 1])

{

copySuccessor(myNode, pos);

flag = delValFromNode(myNode->val[pos], myNode->link[pos]);

if (flag == 0)

{

printf("Given data is not present in B-Tree\n");

}

} else

{

removeVal(myNode, pos);

}

} else

{

flag = delValFromNode(val, myNode->link[pos]);

}

if (myNode->link[pos])

{

if (myNode->link[pos]->count < MIN)

adjustNode(myNode, pos);

}

}

return flag;

}

/\* delete val from B-tree \*/

void deletion(int val, struct btreeNode \*myNode)

{

struct btreeNode \*tmp;

if (!delValFromNode(val, myNode))

{

printf("Given value is not present in B-Tree\n");

return;

}

else

{

if (myNode->count == 0)

{

tmp = myNode;

myNode = myNode->link[0];

free(tmp);

}

}

root = myNode;

return;

}

/\* search val in B-Tree \*/

void searching(int val, int \*pos, struct btreeNode \*myNode)

{

if (!myNode)

{

return;

}

if (val < myNode->val[1])

{

\*pos = 0;

} else

{

for (\*pos = myNode->count;

(val < myNode->val[\*pos] && \*pos > 1); (\*pos)--);

if (val == myNode->val[\*pos]) {

printf("Given data %d is present in B-Tree", val);

return;

}

else

{

printf("%d not found!",val);

}

}

searching(val, pos, myNode->link[\*pos]);

return;

}

/\* B-Tree Traversal \*/

void traversal(struct btreeNode \*myNode)

{

int i;

if (myNode)

{

for (i = 0; i < myNode->count; i++)

{

traversal(myNode->link[i]);

printf("%d ", myNode->val[i + 1]);

}

traversal(myNode->link[i]);

}

}

int main()

{

int val, ch;

while (1)

{

printf("\n1. Insertion\n2. Deletion\n3. Searching\n4. Traversal\n5. Exit\nEnter your choice:\n");

scanf("%d", &ch);

switch (ch)

{

case 1:

printf("Enter your element:");

scanf("%d", &val);

insertion(val);

break;

case 2:

printf("Enter the element to delete:");

scanf("%d", &val);

deletion(val, root);

break;

case 3:

printf("Enter the element to search:");

scanf("%d", &val);

searching(val, &ch, root);

break;

case 4:

traversal(root);

break;

case 5:

exit(0);

default:

printf("U have entered wrong option!!\n");

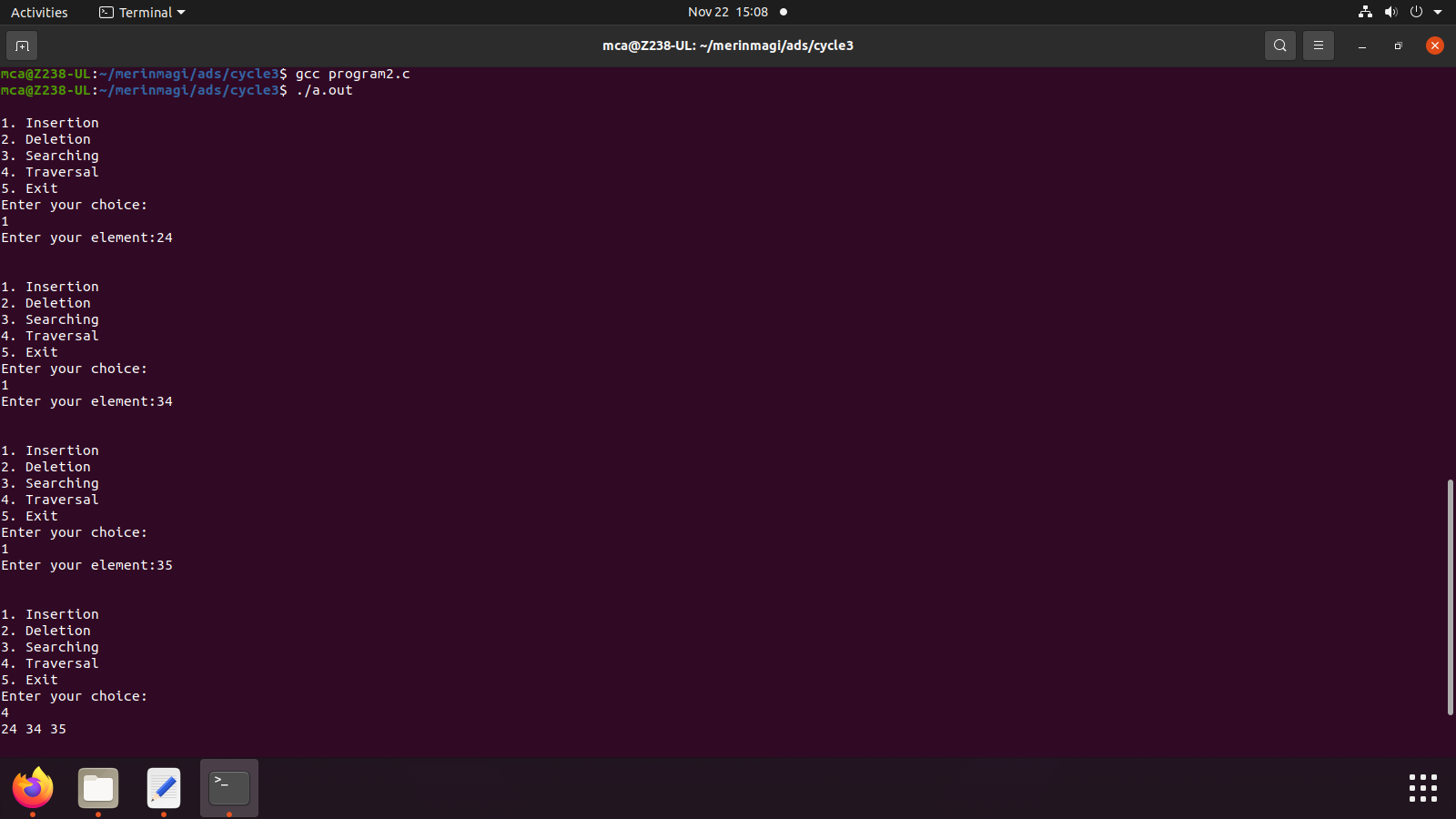
break;

}

printf("\n");

}

}



3) Impliment Red Black Trees and its operations

Source Code

// Implementing Red-Black Tree in C

#include <stdio.h>

#include <stdlib.h>

enum nodeColor {

RED,

BLACK

};

struct rbNode {

int data, color;

struct rbNode \*link[2];

};

struct rbNode \*root = NULL;

// Create a red-black tree

struct rbNode \*createNode(int data) {

struct rbNode \*newnode;

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

newnode->data = data;

newnode->color = RED;

newnode->link[0] = newnode->link[1] = NULL;

return newnode;

}

// Insert an node

void insertion(int data) {

struct rbNode \*stack[98], \*ptr, \*newnode, \*xPtr, \*yPtr;

int dir[98], ht = 0, index;

ptr = root;

if (!root) {

root = createNode(data);

return;

}

stack[ht] = root;

dir[ht++] = 0;

while (ptr != NULL) {

if (ptr->data == data) {

printf("Duplicates Not Allowed!!\n");

return;

}

index = (data - ptr->data) > 0 ? 1 : 0;

stack[ht] = ptr;

ptr = ptr->link[index];

dir[ht++] = index;

}

stack[ht - 1]->link[index] = newnode = createNode(data);

while ((ht >= 3) && (stack[ht - 1]->color == RED)) {

if (dir[ht - 2] == 0) {

yPtr = stack[ht - 2]->link[1];

if (yPtr != NULL && yPtr->color == RED) {

stack[ht - 2]->color = RED;

stack[ht - 1]->color = yPtr->color = BLACK;

ht = ht - 2;

} else {

if (dir[ht - 1] == 0) {

yPtr = stack[ht - 1];

} else {

xPtr = stack[ht - 1];

yPtr = xPtr->link[1];

xPtr->link[1] = yPtr->link[0];

yPtr->link[0] = xPtr;

stack[ht - 2]->link[0] = yPtr;

}

xPtr = stack[ht - 2];

xPtr->color = RED;

yPtr->color = BLACK;

xPtr->link[0] = yPtr->link[1];

yPtr->link[1] = xPtr;

if (xPtr == root) {

root = yPtr;

} else {

stack[ht - 3]->link[dir[ht - 3]] = yPtr;

}

break;

}

} else {

yPtr = stack[ht - 2]->link[0];

if ((yPtr != NULL) && (yPtr->color == RED)) {

stack[ht - 2]->color = RED;

stack[ht - 1]->color = yPtr->color = BLACK;

ht = ht - 2;

} else {

if (dir[ht - 1] == 1) {

yPtr = stack[ht - 1];

} else {

xPtr = stack[ht - 1];

yPtr = xPtr->link[0];

xPtr->link[0] = yPtr->link[1];

yPtr->link[1] = xPtr;

stack[ht - 2]->link[1] = yPtr;

}

xPtr = stack[ht - 2];

yPtr->color = BLACK;

xPtr->color = RED;

xPtr->link[1] = yPtr->link[0];

yPtr->link[0] = xPtr;

if (xPtr == root) {

root = yPtr;

} else {

stack[ht - 3]->link[dir[ht - 3]] = yPtr;

}

break;

}

}

}

root->color = BLACK;

}

// Delete a node

void deletion(int data) {

struct rbNode \*stack[98], \*ptr, \*xPtr, \*yPtr;

struct rbNode \*pPtr, \*qPtr, \*rPtr;

int dir[98], ht = 0, diff, i;

enum nodeColor color;

if (!root) {

printf("Tree not available\n");

return;

}

ptr = root;

while (ptr != NULL) {

if ((data - ptr->data) == 0)

break;

diff = (data - ptr->data) > 0 ? 1 : 0;

stack[ht] = ptr;

dir[ht++] = diff;

ptr = ptr->link[diff];

}

if (ptr->link[1] == NULL) {

if ((ptr == root) && (ptr->link[0] == NULL)) {

free(ptr);

root = NULL;

} else if (ptr == root) {

root = ptr->link[0];

free(ptr);

} else {

stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];

}

} else {

xPtr = ptr->link[1];

if (xPtr->link[0] == NULL) {

xPtr->link[0] = ptr->link[0];

color = xPtr->color;

xPtr->color = ptr->color;

ptr->color = color;

if (ptr == root) {

root = xPtr;

} else {

stack[ht - 1]->link[dir[ht - 1]] = xPtr;

}

dir[ht] = 1;

stack[ht++] = xPtr;

} else {

i = ht++;

while (1) {

dir[ht] = 0;

stack[ht++] = xPtr;

yPtr = xPtr->link[0];

if (!yPtr->link[0])

break;

xPtr = yPtr;

}

dir[i] = 1;

stack[i] = yPtr;

if (i > 0)

stack[i - 1]->link[dir[i - 1]] = yPtr;

yPtr->link[0] = ptr->link[0];

xPtr->link[0] = yPtr->link[1];

yPtr->link[1] = ptr->link[1];

if (ptr == root) {

root = yPtr;

}

color = yPtr->color;

yPtr->color = ptr->color;

ptr->color = color;

}

}

if (ht < 1)

return;

if (ptr->color == BLACK) {

while (1) {

pPtr = stack[ht - 1]->link[dir[ht - 1]];

if (pPtr && pPtr->color == RED) {

pPtr->color = BLACK;

break;

}

if (ht < 2)

break;

if (dir[ht - 2] == 0) {

rPtr = stack[ht - 1]->link[1];

if (!rPtr)

break;

if (rPtr->color == RED) {

stack[ht - 1]->color = RED;

rPtr->color = BLACK;

stack[ht - 1]->link[1] = rPtr->link[0];

rPtr->link[0] = stack[ht - 1];

if (stack[ht - 1] == root) {

root = rPtr;

} else {

stack[ht - 2]->link[dir[ht - 2]] = rPtr;

}

dir[ht] = 0;

stack[ht] = stack[ht - 1];

stack[ht - 1] = rPtr;

ht++;

rPtr = stack[ht - 1]->link[1];

}

if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&

(!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {

rPtr->color = RED;

} else {

if (!rPtr->link[1] || rPtr->link[1]->color == BLACK) {

qPtr = rPtr->link[0];

rPtr->color = RED;

qPtr->color = BLACK;

rPtr->link[0] = qPtr->link[1];

qPtr->link[1] = rPtr;

rPtr = stack[ht - 1]->link[1] = qPtr;

}

rPtr->color = stack[ht - 1]->color;

stack[ht - 1]->color = BLACK;

rPtr->link[1]->color = BLACK;

stack[ht - 1]->link[1] = rPtr->link[0];

rPtr->link[0] = stack[ht - 1];

if (stack[ht - 1] == root) {

root = rPtr;

} else {

stack[ht - 2]->link[dir[ht - 2]] = rPtr;

}

break;

}

} else {

rPtr = stack[ht - 1]->link[0];

if (!rPtr)

break;

if (rPtr->color == RED) {

stack[ht - 1]->color = RED;

rPtr->color = BLACK;

stack[ht - 1]->link[0] = rPtr->link[1];

rPtr->link[1] = stack[ht - 1];

if (stack[ht - 1] == root) {

root = rPtr;

} else {

stack[ht - 2]->link[dir[ht - 2]] = rPtr;

}

dir[ht] = 1;

stack[ht] = stack[ht - 1];

stack[ht - 1] = rPtr;

ht++;

rPtr = stack[ht - 1]->link[0];

}

if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&

(!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {

rPtr->color = RED;

} else {

if (!rPtr->link[0] || rPtr->link[0]->color == BLACK) {

qPtr = rPtr->link[1];

rPtr->color = RED;

qPtr->color = BLACK;

rPtr->link[1] = qPtr->link[0];

qPtr->link[0] = rPtr;

rPtr = stack[ht - 1]->link[0] = qPtr;

}

rPtr->color = stack[ht - 1]->color;

stack[ht - 1]->color = BLACK;

rPtr->link[0]->color = BLACK;

stack[ht - 1]->link[0] = rPtr->link[1];

rPtr->link[1] = stack[ht - 1];

if (stack[ht - 1] == root) {

root = rPtr;

} else {

stack[ht - 2]->link[dir[ht - 2]] = rPtr;

}

break;

}

}

ht--;

}

}

}

// Print the inorder traversal of the tree

void inorderTraversal(struct rbNode \*node) {

if (node) {

inorderTraversal(node->link[0]);

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

inorderTraversal(node->link[1]);

}

return;

}

// Driver code

int main() {

int ch, data;

while (1) {

printf("1. Insertion\t2. Deletion\n");

printf("3. Traverse\t4. Exit");

printf("\nEnter your choice:");

scanf("%d", &ch);

switch (ch) {

case 1:

printf("Enter the element to insert:");

scanf("%d", &data);

insertion(data);

break;

case 2:

printf("Enter the element to delete:");

scanf("%d", &data);

deletion(data);

break;

case 3:

inorderTraversal(root);

printf("\n");

break;

case 4:

exit(0);

default:

printf("Not available\n");

break;

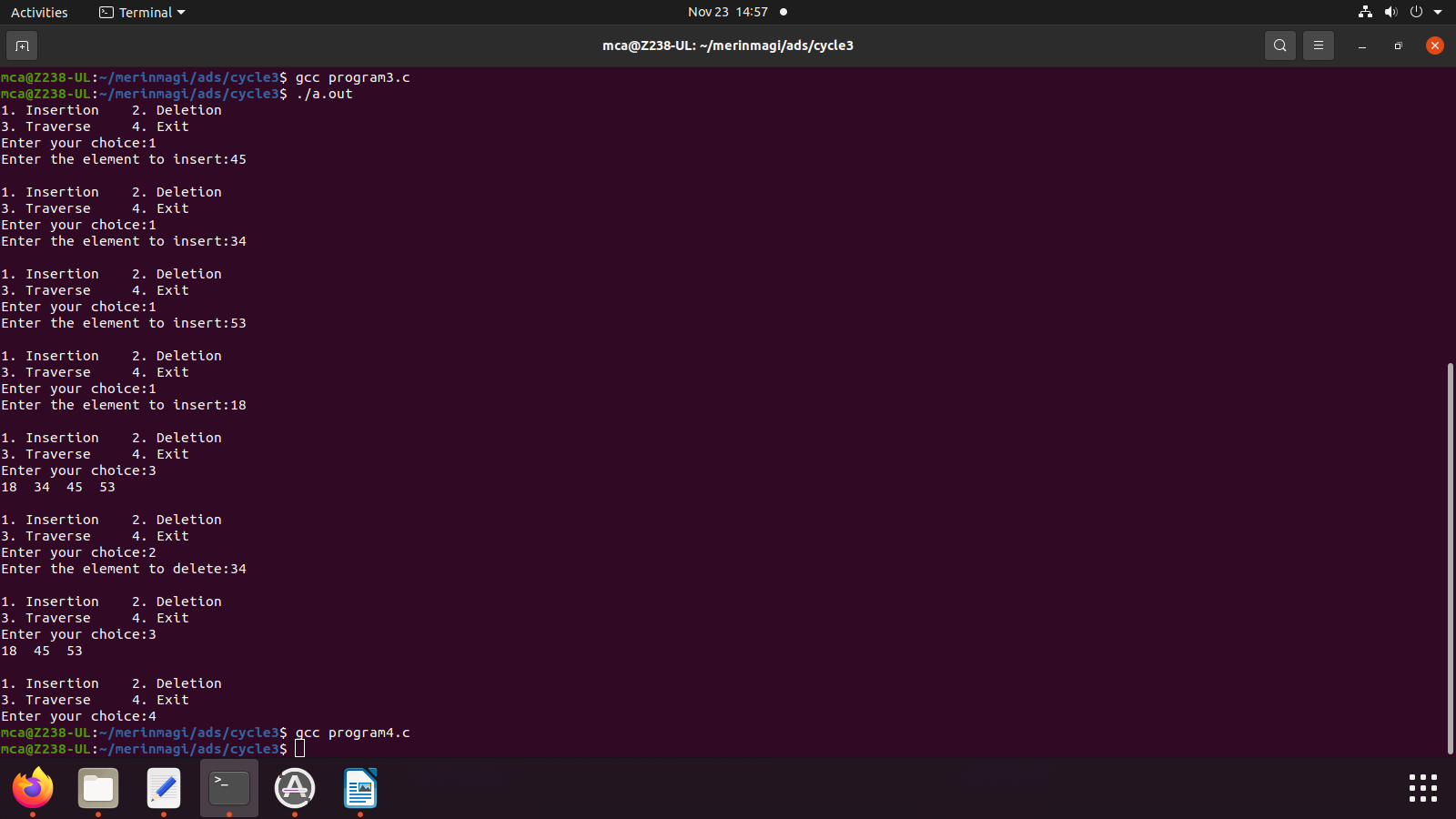
}

printf("\n");

}

return 0;

}



4) Graph Traversal techniques (DFS and BFS) and Topological Sorting

Source code

//graph traversal

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_VERTICES 5

typedef struct Node {

int vertex;

struct Node\* next;

} Node;

typedef struct Graph {

Node\* adjList[MAX\_VERTICES];

int numVertices;

} Graph;

// Function to create a new graph with a given number of vertices

Graph\* createGraph(int numVertices) {

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

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

exit(EXIT\_FAILURE);

}

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

graph->numVertices = numVertices;

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

graph->adjList[i] = NULL;

return graph;

}

// Function to add an edge to a directed graph

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

if (src < 0 || src >= graph->numVertices || dest < 0 || dest >= graph->numVertices) {

printf("Invalid source or destination vertex. Ignoring edge...\n");

return;

}

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

newNode->vertex = dest;

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

graph->adjList[src] = newNode;

}

// Function to perform Depth-First Search (DFS)

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

visited[vertex] = true;

printf("%d ", vertex);

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

while (adjNode != NULL) {

int adjVertex = adjNode->vertex;

if (!visited[adjVertex])

DFS(graph, adjVertex, visited);

adjNode = adjNode->next;

}

}

// Function to perform Breadth-First Search (BFS)

void BFS(Graph\* graph, int start) {

if (start < 0 || start >= graph->numVertices) {

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

exit(EXIT\_FAILURE);

}

bool visited[MAX\_VERTICES] = { false };

int queue[MAX\_VERTICES];

int front = 0, rear = -1;

visited[start] = true;

queue[++rear] = start;

while (front <= rear) {

int vertex = queue[front++];

printf("%d ", vertex);

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

while (adjNode != NULL) {

int adjVertex = adjNode->vertex;

if (!visited[adjVertex]) {

visited[adjVertex] = true;

queue[(++rear) % MAX\_VERTICES] = adjVertex; // Circular array for the queue

}

adjNode = adjNode->next;

}

}

}

// Function to perform Topological Sorting using DFS

void topologicalSortUtil(Graph\* graph, int vertex, bool visited[], int stack[], int\* stackIndex) {

visited[vertex] = true;

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

while (adjNode != NULL) {

int adjVertex = adjNode->vertex;

if (!visited[adjVertex])

topologicalSortUtil(graph, adjVertex, visited, stack, stackIndex);

adjNode = adjNode->next;

}

stack[++(\*stackIndex)] = vertex;

}

// Function to perform Topological Sorting

void topologicalSort(Graph\* graph) {

bool visited[MAX\_VERTICES] = { false };

int stack[MAX\_VERTICES];

int stackIndex = -1;

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

if (!visited[i])

topologicalSortUtil(graph, i, visited, stack, &stackIndex);

}

// Print the topological order

printf("Topological Sorting: ");

while (stackIndex >= 0)

printf("%d ", stack[stackIndex--]);

}

int main() {

Graph\* graph = createGraph(MAX\_VERTICES);

// Pre-define edges for demonstration

addEdge(graph, 0, 1);

addEdge(graph, 0, 3);

addEdge(graph, 1, 2);

addEdge(graph, 2, 3);

addEdge(graph, 3, 4);

int choice;

printf("Choose a technique to demonstrate:\n");

printf("1. DFS\n2. BFS\n3. Topological Sort\n4. Exit\n");

while (1) {

printf("Enter your choice (1-4): ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("DFS Traversal: ");

DFS(graph, 0, (bool[MAX\_VERTICES]){ false });

printf("\n");

break;

case 2:

printf("BFS Traversal: ");

BFS(graph, 0);

printf("\n");

break;

case 3:

topologicalSort(graph);

printf("\n");

break;

case 4:

exit(EXIT\_SUCCESS);

default:

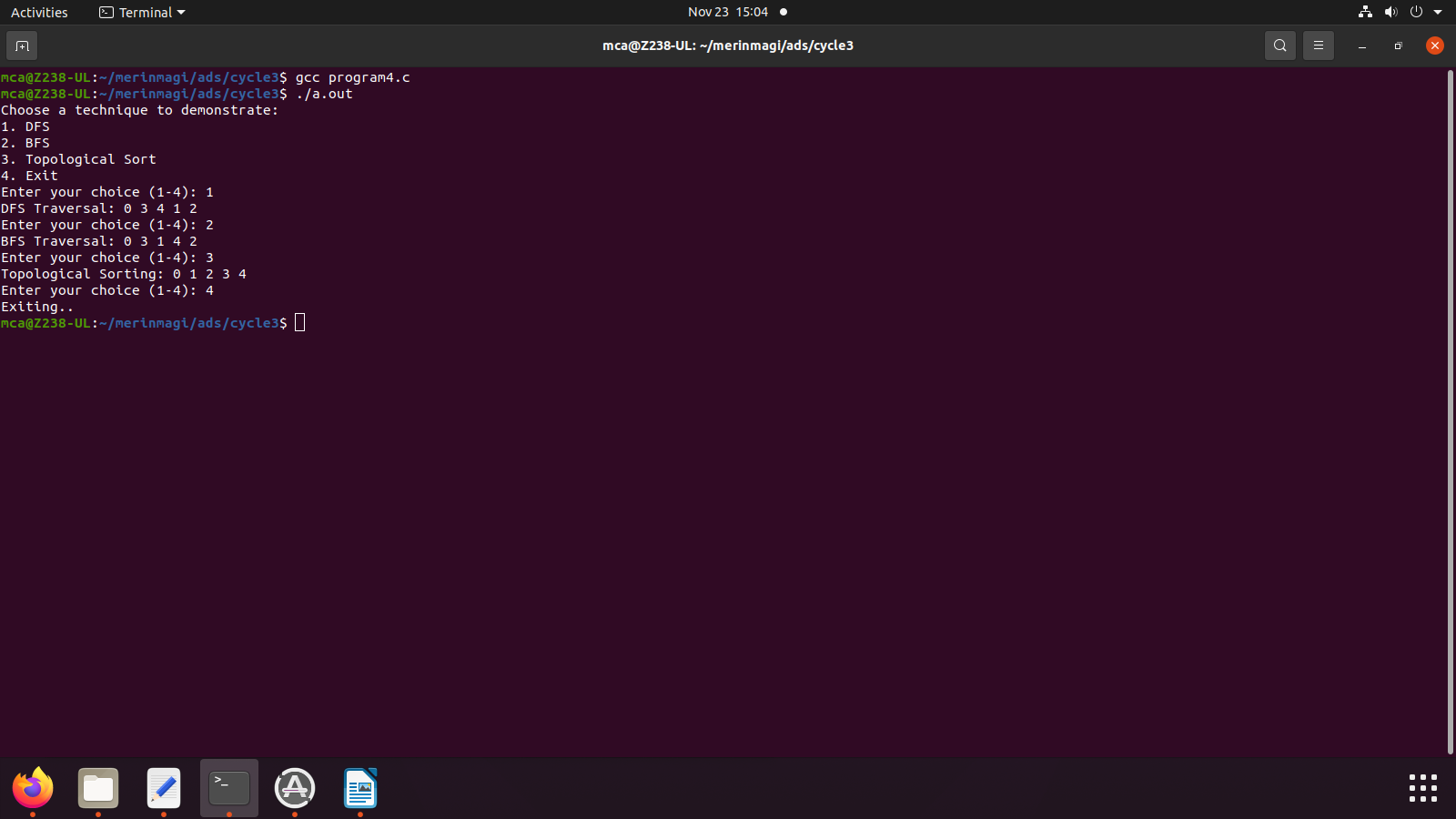
printf("Invalid choice!\n");

}

}

return 0;

}



5) Finding the Strongly connected Components in a directed graph

Source code

//strongly connected components

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_VERTICES 100

typedef struct Node {

int vertex;

struct Node\* next;

} Node;

typedef struct Graph {

Node\* adjList[MAX\_VERTICES];

Node\* revAdjList[MAX\_VERTICES];

int numVertices;

} Graph;

Graph\* createGraph(int numVertices) {

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

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

exit(EXIT\_FAILURE);

}

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

graph->numVertices = numVertices;

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

graph->adjList[i] = NULL;

graph->revAdjList[i] = NULL;

}

return graph;

}

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

if (src < 0 || src >= graph->numVertices || dest < 0 || dest >= graph->numVertices) {

printf("Invalid source or destination vertex. Ignoring edge...\n");

return;

}

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

newNode->vertex = dest;

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

graph->adjList[src] = newNode;

// Reverse graph for Kosaraju's algorithm

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

newNode->vertex = src;

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

graph->revAdjList[dest] = newNode;

}

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

visited[vertex] = true;

printf("%d ", vertex);

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

while (adjNode != NULL) {

int adjVertex = adjNode->vertex;

if (!visited[adjVertex])

DFSUtil(graph, adjVertex, visited);

adjNode = adjNode->next;

}

}

void fillOrder(Graph\* graph, int vertex, bool visited[], int stack[], int\* stackIndex) {

visited[vertex] = true;

Node\* adjNode = graph->revAdjList[vertex];

while (adjNode != NULL) {

int adjVertex = adjNode->vertex;

if (!visited[adjVertex])

fillOrder(graph, adjVertex, visited, stack, stackIndex);

adjNode = adjNode->next;

}

stack[++(\*stackIndex)] = vertex;

}

Graph\* getTranspose(Graph\* graph) {

Graph\* transposedGraph = createGraph(graph->numVertices);

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

Node\* current = graph->adjList[i];

while (current != NULL) {

addEdge(transposedGraph, current->vertex, i);

current = current->next;

}

}

return transposedGraph;

}

void printSCCs(Graph\* graph) {

int stack[MAX\_VERTICES];

int stackIndex = -1;

bool visited[MAX\_VERTICES] = { false };

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

if (!visited[i])

fillOrder(graph, i, visited, stack, &stackIndex);

}

Graph\* transposedGraph = getTranspose(graph);

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

visited[i] = false;

while (stackIndex >= 0) {

int vertex = stack[stackIndex--];

if (!visited[vertex]) {

DFSUtil(transposedGraph, vertex, visited);

printf("\n");

}

}

free(transposedGraph);

}

int main() {

Graph\* graph = createGraph(8);

// Define edges for demonstration

addEdge(graph, 0, 1);

addEdge(graph, 1, 2);

addEdge(graph, 2, 0);

addEdge(graph, 2, 3);

addEdge(graph, 3, 4);

addEdge(graph, 4, 5);

addEdge(graph, 5, 3);

addEdge(graph, 6, 5);

addEdge(graph, 6, 7);

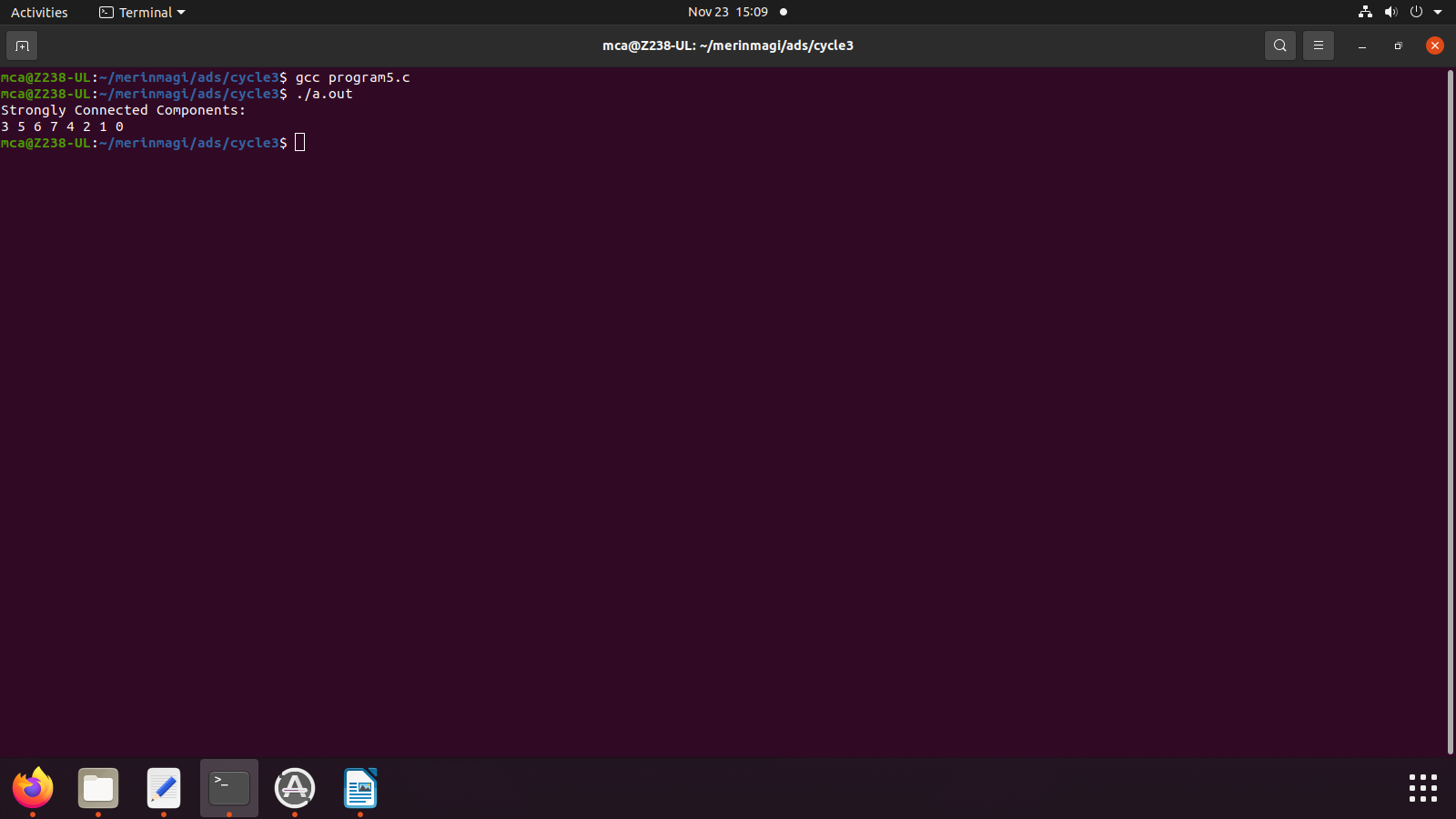
addEdge(graph, 7, 6);

printf("Strongly Connected Components:\n");

printSCCs(graph);

return 0;

}



6) Prim’s Algorithm for finding the minimum cost spanning tree

Source code

//prims algorithm

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_VERTICES 100

#define INF 999999

typedef struct {

int parent;

int key;

bool inMST;

} Vertex;

typedef struct Graph {

int numVertices;

int adjacencyMatrix[MAX\_VERTICES][MAX\_VERTICES];

Vertex vertices[MAX\_VERTICES];

} Graph;

Graph\* createGraph(int numVertices) {

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

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

exit(EXIT\_FAILURE);

}

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

graph->numVertices = numVertices;

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

graph->vertices[i].parent = -1;

graph->vertices[i].key = INF;

graph->vertices[i].inMST = false;

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

graph->adjacencyMatrix[i][j] = INF;

}

return graph;

}

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

if (src >= 0 && src < graph->numVertices && dest >= 0 && dest < graph->numVertices) {

graph->adjacencyMatrix[src][dest] = weight;

graph->adjacencyMatrix[dest][src] = weight;

} else {

printf("Invalid source or destination vertex. Ignoring edge...\n");

}

}

int findMinKeyVertex(Graph\* graph) {

int minKey = INF;

int minIndex = -1;

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

if (!graph->vertices[i].inMST && graph->vertices[i].key < minKey) {

minKey = graph->vertices[i].key;

minIndex = i;

}

}

return minIndex;

}

void primMST(Graph\* graph) {

graph->vertices[0].key = 0;

for (int count = 0; count < graph->numVertices - 1; ++count) {

int u = findMinKeyVertex(graph);

graph->vertices[u].inMST = true;

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

if (graph->adjacencyMatrix[u][v] != INF && !graph->vertices[v].inMST &&

graph->adjacencyMatrix[u][v] < graph->vertices[v].key) {

graph->vertices[v].key = graph->adjacencyMatrix[u][v];

graph->vertices[v].parent = u;

}

}

}

printf("Minimum Cost Spanning Tree (Prim's Algorithm):\n");

for (int i = 1; i < graph->numVertices; ++i)

printf("Edge: %d - %d, Weight: %d\n", graph->vertices[i].parent, i, graph->vertices[i].key);

}

int main() {

Graph\* graph = createGraph(5);

// Define edges for demonstration

addEdge(graph, 0, 1, 2);

addEdge(graph, 0, 3, 6);

addEdge(graph, 1, 2, 3);

addEdge(graph, 1, 3, 8);

addEdge(graph, 1, 4, 5);

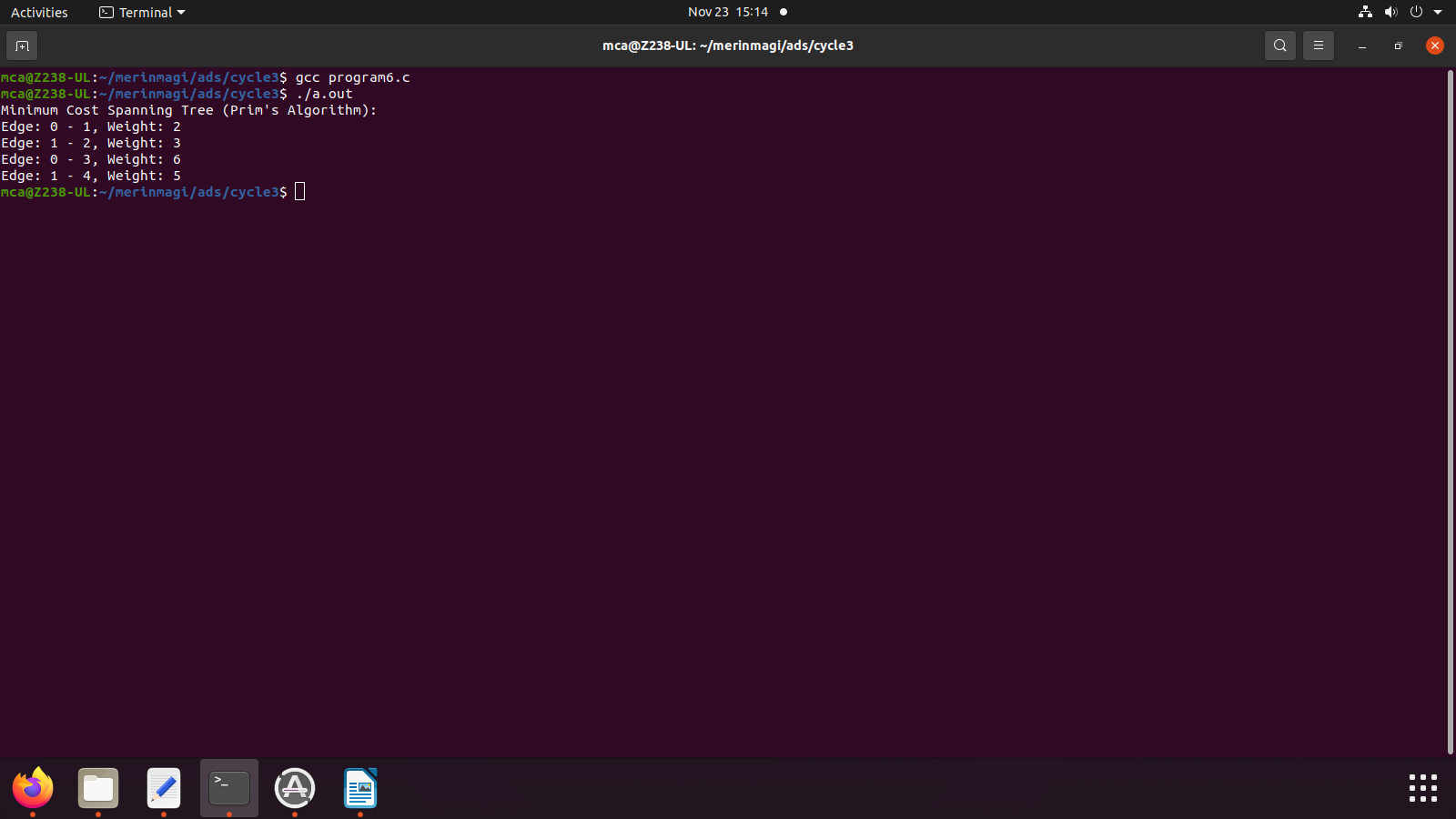
addEdge(graph, 2, 4, 7);

addEdge(graph, 3, 4, 9);

primMST(graph);

return 0;

}



7) Kruskal’s algorithm using the Disjoint set data structure

Source code

//kruskals algorithm

#include <stdio.h>

#include <stdlib.h>

#define MAX\_VERTICES 100

typedef struct Edge {

int src, dest, weight;

} Edge;

typedef struct {

int parent, rank;

} Subset;

typedef struct {

int numVertices, numEdges;

Edge edges[MAX\_VERTICES \* MAX\_VERTICES];

} Graph;

Graph\* createGraph(int numVertices, int numEdges) {

if (numVertices <= 0 || numVertices > MAX\_VERTICES || numEdges <= 0 || numEdges > MAX\_VERTICES \* MAX\_VERTICES) {

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

exit(EXIT\_FAILURE);

}

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

graph->numVertices = numVertices;

graph->numEdges = numEdges;

return graph;

}

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

if (index >= 0 && index < graph->numEdges && src >= 0 && src < graph->numVertices &&

dest >= 0 && dest < graph->numVertices) {

graph->edges[index].src = src;

graph->edges[index].dest = dest;

graph->edges[index].weight = weight;

} else {

printf("Invalid edge information. Exiting...\n");

exit(EXIT\_FAILURE);

}

}

int compareEdges(const void\* a, const void\* b) {

return ((Edge\*)a)->weight - ((Edge\*)b)->weight;

}

int find(Subset subsets[], int i) {

if (subsets[i].parent != i)

subsets[i].parent = find(subsets, subsets[i].parent);

return subsets[i].parent;

}

void unionSets(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++;

}

}

void kruskalMST(Graph\* graph) {

Edge result[graph->numVertices];

int e = 0;

int i = 0;

qsort(graph->edges, graph->numEdges, sizeof(graph->edges[0]), compareEdges);

Subset subsets[graph->numVertices];

for (i = 0; i < graph->numVertices; ++i) {

subsets[i].parent = i;

subsets[i].rank = 0;

}

i = 0;

while (e < graph->numVertices - 1 && i < graph->numEdges) {

Edge nextEdge = graph->edges[i++];

int x = find(subsets, nextEdge.src);

int y = find(subsets, nextEdge.dest);

if (x != y) {

result[e++] = nextEdge;

unionSets(subsets, x, y);

}

}

printf("Minimum Cost Spanning Tree (Kruskal's Algorithm):\n");

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

printf("Edge: %d - %d, Weight: %d\n", result[i].src, result[i].dest, result[i].weight);

}

int main() {

Graph\* graph = createGraph(4, 5);

// Define edges for demonstration

addEdge(graph, 0, 0, 1, 10);

addEdge(graph, 1, 0, 2, 6);

addEdge(graph, 2, 0, 3, 5);

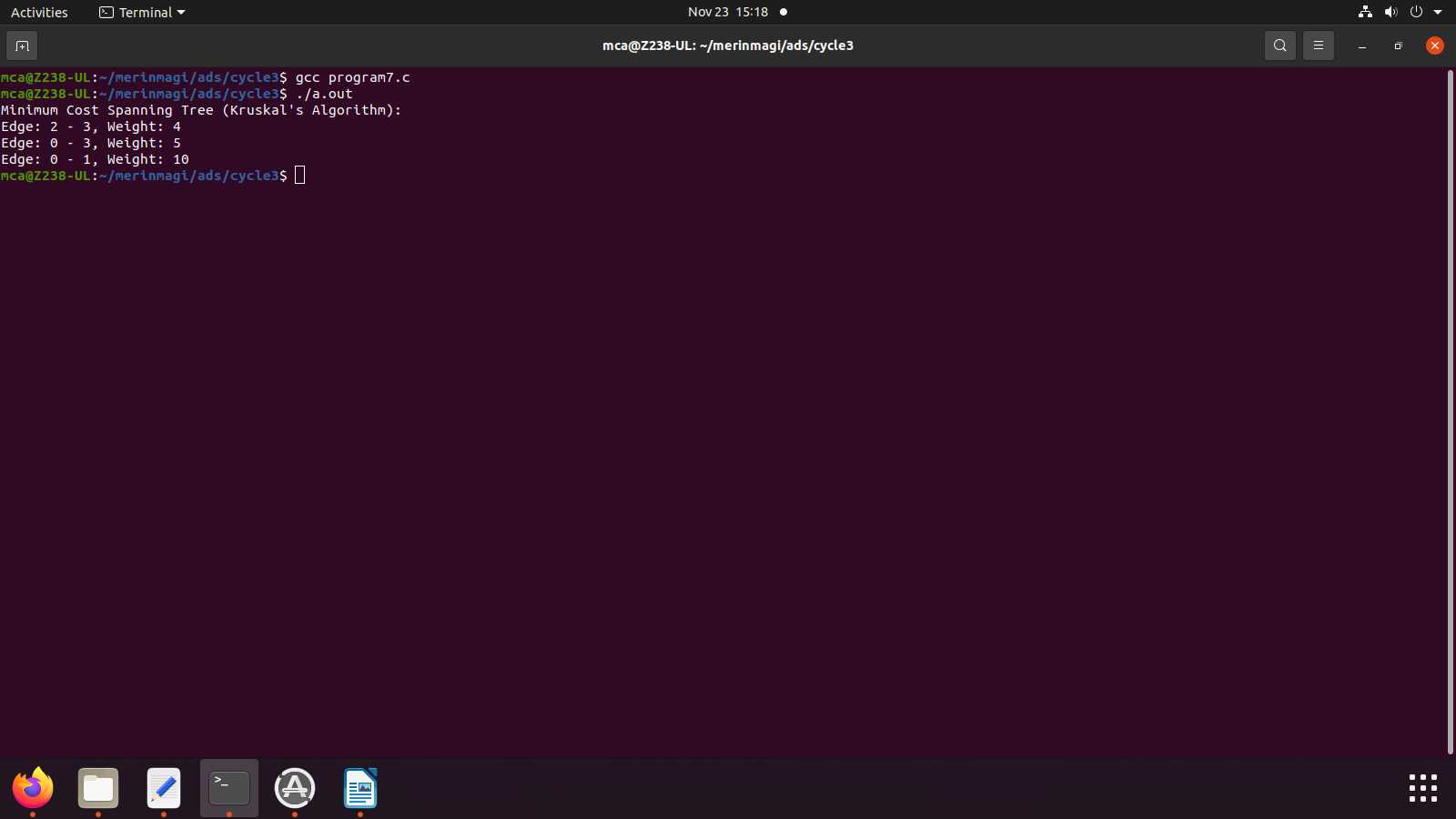
addEdge(graph, 3, 1, 3, 15);

addEdge(graph, 4, 2, 3, 4);

kruskalMST(graph);

return 0;

}



8) Single Source shortest path algorithm using any heap structure that supports mergeable heap

operations

Source Code

//single source shortest path

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <time.h>

#define MAX\_VERTICES 5

#define MAX\_DISTANCE 50 // Adjust this based on the expected maximum distance between nodes

typedef struct {

int vertex, distance;

} Node;

typedef struct {

Node\* heap;

int capacity, size;

} MinHeap;

typedef struct {

int numVertices;

int adjacencyMatrix[MAX\_VERTICES][MAX\_VERTICES];

} Graph;

Graph\* createGraph(int numVertices) {

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

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

exit(EXIT\_FAILURE);

}

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

graph->numVertices = numVertices;

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

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

if (i == j) {

graph->adjacencyMatrix[i][j] = 0; // Distance from a vertex to itself is 0

} else {

graph->adjacencyMatrix[i][j] = rand() % MAX\_DISTANCE + 1; // Random distance between 1 and MAX\_DISTANCE

}

}

}

return graph;

}

MinHeap\* createMinHeap(int capacity) {

MinHeap\* heap = (MinHeap\*)malloc(sizeof(MinHeap));

heap->capacity = capacity;

heap->size = 0;

heap->heap = (Node\*)malloc(capacity \* sizeof(Node));

return heap;

}

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

Node temp = \*a;

\*a = \*b;

\*b = temp;

}

void minHeapify(MinHeap\* heap, int idx) {

int smallest = idx;

int left = 2 \* idx + 1;

int right = 2 \* idx + 2;

if (left < heap->size && heap->heap[left].distance < heap->heap[smallest].distance)

smallest = left;

if (right < heap->size && heap->heap[right].distance < heap->heap[smallest].distance)

smallest = right;

if (smallest != idx) {

swap(&heap->heap[idx], &heap->heap[smallest]);

minHeapify(heap, smallest);

}

}

bool isEmpty(MinHeap\* heap) {

return heap->size == 0;

}

Node extractMin(MinHeap\* heap) {

if (isEmpty(heap))

exit(EXIT\_FAILURE);

Node root = heap->heap[0];

heap->heap[0] = heap->heap[heap->size - 1];

heap->size--;

minHeapify(heap, 0);

return root;

}

void decreaseKey(MinHeap\* heap, int vertex, int distance) {

int i;

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

if (heap->heap[i].vertex == vertex) {

heap->heap[i].distance = distance;

break;

}

}

while (i != 0 && heap->heap[i].distance < heap->heap[(i - 1) / 2].distance) {

swap(&heap->heap[i], &heap->heap[(i - 1) / 2]);

i = (i - 1) / 2;

}

}

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

MinHeap\* heap = createMinHeap(graph->numVertices);

Node\* result = (Node\*)malloc(graph->numVertices \* sizeof(Node));

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

heap->heap[i].vertex = i;

heap->heap[i].distance = MAX\_DISTANCE \* graph->numVertices + 1; // A value larger than the sum of all possible distances

result[i].vertex = -1;

result[i].distance = MAX\_DISTANCE \* graph->numVertices + 1;

}

heap->heap[src].distance = 0;

result[src].distance = 0;

clock\_t start\_time = clock();

while (!isEmpty(heap)) {

Node current = extractMin(heap);

int u = current.vertex;

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

if (graph->adjacencyMatrix[u][v] != 0) { // Consider only non-zero distances

int alt = result[u].distance + graph->adjacencyMatrix[u][v];

if (alt < result[v].distance) {

result[v].distance = alt;

result[v].vertex = u;

decreaseKey(heap, v, alt);

}

}

}

}

clock\_t end\_time = clock();

double execution\_time = ((double)(end\_time - start\_time)) / CLOCKS\_PER\_SEC;

printf("Paths from source %d to destination %d:\n", src, dest);

printf("Shortest Distance: %d\n", result[dest].distance);

printf("Execution Time: %f seconds\n", execution\_time);

printf("Path: ");

int temp = dest;

while (temp != -1) {

printf("%d ", temp);

temp = result[temp].vertex;

}

printf("\n");

printf("All Paths Traversed:\n");

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

if (i == src) continue;

printf("To vertex %d: Distance = %d, Path = ", i, result[i].distance);

temp = i;

while (temp != -1) {

printf("%d ", temp);

temp = result[temp].vertex;

}

printf("\n");

}

free(heap->heap);

free(heap);

free(result);

}

int main() {

srand(time(NULL));

Graph\* graph = createGraph(5);

printf("Random Distance Matrix:\n");

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

for (int j = 0; j < graph->numVertices; ++j) {

printf("%2d ", graph->adjacencyMatrix[i][j]);

}

printf("\n");

}

int sourceVertex, destVertex;

printf("\nEnter source vertex (0-%d): ", graph->numVertices - 1);

scanf("%d", &sourceVertex);

printf("Enter destination vertex (0-%d): ", graph->numVertices - 1);

scanf("%d", &destVertex);

if (sourceVertex < 0 || sourceVertex >= graph->numVertices ||

destVertex < 0 || destVertex >= graph->numVertices) {

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

return EXIT\_FAILURE;

}

dijkstra(graph, sourceVertex, destVertex);

free(graph);

return 0;

}

