

|  |  |  |  |
| --- | --- | --- | --- |
| Course Code | : | CST204 | |
| Course Name | : | Data Structures | |
| Lecturer | : | Dr. Raja Majid Mehmood | |
| Academic Session | : | 2023/09 | |
| Assessment Title | : | Assignment –  Sorting, Search, and Graph Algorithms | |
| Submission Due Date | : | 28th December 2023 | |
| Prepared by |  |  | |
| : | Student ID | Student Name |
|  | CST2209301 | Thet Paing Soe |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| Date Received | : |  | |
|  |  |  | |
| Feedback from Lecturer:  Mark: | | | |

**Own Work Declaration**

I hereby understand my work would be checked for plagiarism or other misconduct, and the softcopy would be saved for future comparison(s).

I hereby confirm that all the references or sources of citations have been correctly listed or presented and I clearly understand the serious consequence caused by any intentional or unintentional misconduct.

This work is not made on any work of other students (past or present), and it has not been submitted to any other courses or institutions before.

Signature:

A blue text on a black background

Description automatically generated

Date: 28th December 2023

**C++ Code**

**Question 1:**

#include <iostream>

#include <stack>

using namespace std;

int main() {

//Input elements 85, 75, 95, 80

int a[] = {85, 75, 95, 80}, n = 4;

//Create original stack (st)

stack<int> st;

//Push the elements into the stack

cout << "(1) push the elements ";

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

cout << a[i] << ",";

st.push(a[i]);

}

cout << " onto the stack st." << endl;

cout << "(2) the process of sorting elements of stack st is:" << endl;

//Create a temporary extra stack (tmpst)

stack<int> tmpst;

while (!st.empty()) {

//Obtain the top element from the st and assign to tmp

int tmp = st.top();

//Pop out the element from the st

st.pop();

cout << "\tst: pop out " << tmp << "=>" << endl;

//If stack tmpst is not empty

while (!tmpst.empty()) {

cout << "\t\ttmpst: get the top element " << tmpst.top() << endl;

//Making sure to push tmpst top element until tmpst top element becomes smaller than tmp.

if (tmpst.top() > tmp) {

cout << "\t\t\tSince " << tmpst.top() << " > " << tmp << " tmpst: pop out " << tmpst.top() << endl;

cout << "\t\t\tst: push " << tmpst.top() << endl;

//Push the element from tmpst onto st

st.push(tmpst.top());

//Pop out tmpst element

tmpst.pop();

}

else

{

cout << "\t\t\tSince " << tmpst.top() << " < " << tmp << ", exit the loop" << endl;

break;

}

}

cout << "\t\ttmpst: push " << tmp << endl;

//If stack tmpst is empty, push tmp onto tmpst

tmpst.push(tmp);

cout << endl;

}

//If tmpst is not empty

while (!tmpst.empty()) {

//Push remaining elements from tmpst to the st

st.push(tmpst.top());

//Pop the top element from tmpst

tmpst.pop();

}

cout << "(3) the sorting of stack st ends." << endl;

cout << "(4) the popping sequence of st is: ";

while (!st.empty()) {

//Print out the sorted numbers on screen

cout << st.top() << " ";

//Pop out the top element of st

st.pop();

}

cout << endl;

return 0;

}

**Question 2 - Case A:**

#include <iostream>

#include <stdlib.h>

#include <time.h>

using namespace std;

class Node {

public:

int val;

int weight;

Node\* next;

Node(int x, int w) //Constructor

{

val = x ;

weight = w;

next = NULL;

}

};

class List {

public:

Node\* head;

List(void) //Constructor

{

head = NULL;

}

};

class Graph {

public:

int vt;

List\* array;

Graph(int vertices) //Constructor

{

vt = vertices;

array = new List[vertices];

}

//addEdge is to add an edge between two vertices

//source = source vertex, dest = destination vertex, x=weight of the edge

void addEdge(int source, int dest, int x) {

//Create new Node with given vertex(dest) and weight (x)

Node\* temp = new Node(dest, x);

//Update next pointer of newly created Node object temp to point to current head of the adjacency list for source vertex

temp->next = array[source].head;

//Assign temp pointer to head member variable of the list object at index source in the array

array[source].head = temp;

}

//numEdge to count total number of edges in the graph

int numEdge() {

int counter = 0;

//To iterate over all vertices in the graph

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

//Declare temp and initialize with value of head member variable at index i in array

Node\* temp = array[i].head;

//Loop until there are no more Nodes

while (temp != NULL) {

temp = temp->next;

counter++;

}

}

return counter;

}

};

// Case A Sparse Graph - 100x100

int main(){

int vertices = 1000;

int edges;

Graph gr(vertices);

srand(time(0));

//Add edges with random source and destination vertices and random weight

for (int i = 0; i < (100 \* 100); i++) {

gr.addEdge(rand() % 1000, rand() % 1000, rand());

}

//Count total number of edges

edges = gr.numEdge();

cout << "The total number of edges for Case A is " << edges;

cout << endl;

return 0;

}

**Question 2 - Case B:**

#include <iostream>

#include <stdlib.h>

#include <time.h>

using namespace std;

class Node {

public:

int val;

int weight;

Node\* next;

Node(int x, int w) //Constructor

{

val = x ;

weight = w;

next = NULL;

}

};

class List {

public:

Node\* head;

List(void) //Constructor

{

head = NULL;

}

};

class Graph {

public:

int vt;

List\* array;

Graph(int vertices) //Constructor

{

vt = vertices;

array = new List[vertices];

}

//addEdge is to add an edge between two vertices

//source = source vertex, dest = destination vertex, x=weight of the edge

void addEdge(int source, int dest, int x) {

//Create new Node with given vertex(dest) and weight (x)

Node\* temp = new Node(dest, x);

//Update next pointer of newly created Node object temp to point to current head of the adjacency list for source vertex

temp->next = array[source].head;

//Assign temp pointer to head member variable of the list object at index source in the array

array[source].head = temp;

}

//numEdge to count total number of edges in the graph

int numEdge() {

int counter = 0;

//To iterate over all vertices in the graph

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

//Declare temp and initialize with value of head member variable at index i in array

Node\* temp = array[i].head;

//Loop until there are no more Nodes

while (temp != NULL) {

temp = temp->next;

counter++;

}

}

return counter;

}

};

// Case B Dense Graph - 600x600

int main(){

int vertices = 1000;

int edges;

Graph gr(vertices);

srand(time(0));

//add edges with random source and destination vertices and random weight

for (int i = 0; i < (600 \* 600); i++) {

gr.addEdge(rand() % 1000, rand() % 1000, rand());

}

//count total number of edges

edges = gr.numEdge();

cout << "The total number of edges for Case B is " << edges;

cout << endl;

return 0;

}

**Question 2 - Case C:**

#include <iostream>

#include <stdlib.h>

#include <time.h>

using namespace std;

class Node {

public:

int val;

int weight;

Node\* next;

Node(int x, int w) //Constructor

{

val = x ;

weight = w;

next = NULL;

}

};

class List {

public:

Node\* head;

List(void) //Constructor

{

head = NULL;

}

};

class Graph {

public:

int vt;

List\* array;

Graph(int vertices) //Constructor

{

vt = vertices;

array = new List[vertices];

}

//addEdge is to add an edge between two vertices

//source = source vertex, dest = destination vertex, x=weight of the edge

void addEdge(int source, int dest, int x) {

//Create new Node with given vertex(dest) and weight (x)

Node\* temp = new Node(dest, x);

//Update next pointer of newly created Node object temp to point to current head of the adjacency list for source vertex

temp->next = array[source].head;

//Assign temp pointer to head member variable of the list object at index source in the array

array[source].head = temp;

}

//numEdge to count total number of edges in the graph

int numEdge() {

int counter = 0;

//To iterate over all vertices in the graph

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

//Declare temp and initialize with value of head member variable at index i in array

Node\* temp = array[i].head;

//Loop until there are no more Nodes

while (temp != NULL) {

temp = temp->next;

counter++;

}

}

return counter;

}

};

// Case C Complete Graph - 1000 x 1000

int main(){

int vertices = 1000;

int edges;

Graph gr(vertices);

srand(time(0));

//Add edges with random source and destination vertices and random weight

for (int i = 0; i < 1000; i++) {

for (int j = 0; j < 1000; j++) {

gr.addEdge(i, j, rand());

}

}

//Count total number of edges

edges = gr.numEdge();

cout << "The total number of edges for Case C is " << edges;

cout << endl;

return 0;

}

**Question 3:**

#include <iostream>

#include <stdlib.h>

#include <time.h>

#include <algorithm>

#include <cmath>

using namespace std;

// **initialise array with random data**

void generateRandom(int a[], int SIZE)

{

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

{

a[i] = rand() % SIZE; // generate value in range from 0 to SIZE-1 inclusive

}

}

// **to sort an array**

void doSortedList(int a[], int SIZE)

{

sort(a, a + SIZE); //sort array a using the range from a's first element address to a+SIZE address(last element)

}

// **implement binary search and Count at every midpoint visit**

int findByBinSearch(int key, int a[], int SIZE)

{

int low = 0;

int high = SIZE - 1;

int mid;

int counter = 0;

while (low <= high)

{

counter++;

mid = (low + high) / 2;

if (a[mid] < key)

low = mid + 1;

else if (a[mid] > key)

high = mid - 1;

else

break;

}

if (a[mid] != key)

return -1;

else

return counter;

}

// **implement sequential search and Count at every array-index visit**

int findBySeqSearch(int key, int a[], int SIZE)

{

int n = SIZE;

int i;

int counter = 0;

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

{

counter++;

if (a[i] == key)

{

break;

}

else if (i == n - 1)

{

return -1;

}

}

return i + 1;

}

int main()

{

srand(time(0));

//long long int is used to store large values of SIZE

long long int SIZE = 10;

cout << "\nSize\t\tComplexity-Sequential\tComplexity-Binary\t\t(best/avg/worst)\n\t\t\t\t\t\t\t\tSeq. Cost\tBinary Cost\n\n";

//Loop again to get different sizes.

for (int i = 0; i < 8; i++)

{

int\* a = new int[SIZE];

// call generateRandom()

generateRandom(a, SIZE);

// call doSortedList()

doSortedList(a, SIZE);

int searchNum = rand() % SIZE;

// call both search functions and print information as in Table 1

int sequence = findBySeqSearch(searchNum, a, SIZE);

int binary = findByBinSearch(searchNum, a, SIZE);

string scost, bcost;

//Calculate sequence cost

if (sequence == 1)

scost = "best ";

else if (sequence == SIZE)

scost = "worst ";

else if (sequence > 1 && sequence < SIZE)

scost = "average ";

else

scost = "not found";

//Calculate binary cost

if (binary == 1)

bcost = "best ";

else if (binary == floor(log2(SIZE)) || binary == floor(log2(SIZE) + 1))

bcost = "worst ";

else if (binary > 1 && binary < floor(log2(SIZE)))

bcost = "average ";

else

bcost = "not found";

//Print out the values

cout << SIZE << "\t\t\t" << sequence << "\t\t\t" << binary << "\t\t" << scost << "\t\t" << bcost << endl;

SIZE = SIZE \* 10;

// Free dynamically allocated memory

delete[] a;

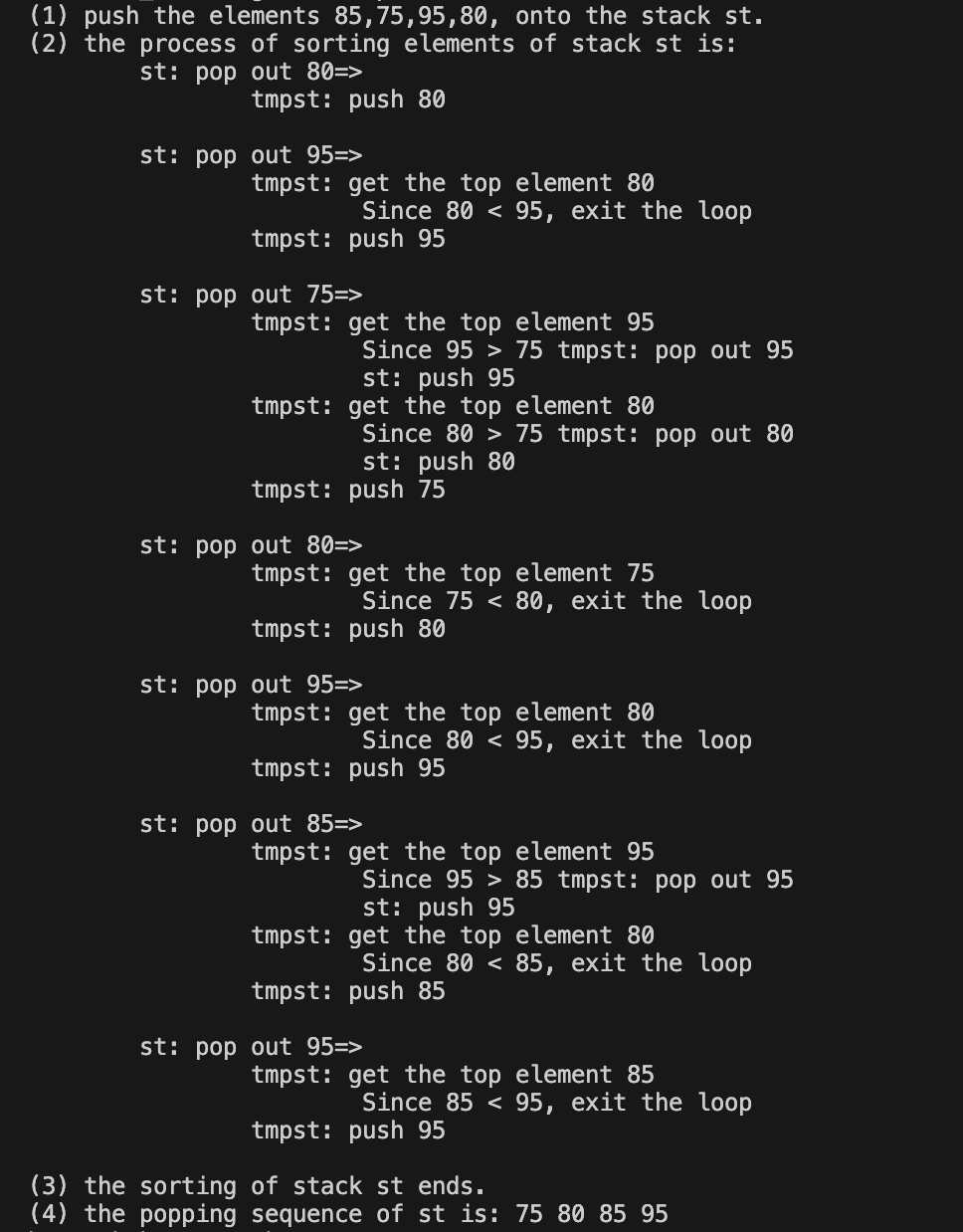
}

return 0;

}

**Results and Discussion**

**Question 1:**



**Explanation:**

1. Initialize an array, 'a,' with a size of 4, and input the elements 85, 75, 95, 80 into it.
2. Create a stack named 'st' and push the elements of array 'a' onto the stack.
3. Establish a temporary stack, 'tmpst.'
4. Execute the outer while loop to iterate through stack 'st,' extracting elements from the top one by one for sorting.
5. Retrieve and store the top element of 'st' in the variable 'tmp' using the expression "int tmp = st.top();"
6. Pop the top element from 'st.'
7. Utilize the inner while loop to compare the top element of 'tmpst' with the current variable 'tmp,' ensuring the elements in 'tmpst' are arranged in descending order.
8. Print the top element of 'tmpst.'
9. If the top element of 'tmpst' is greater than 'tmp,' move it back to 'st' since it should be positioned after the current element.
10. If the top of 'tmpst' is not greater than 'tmp,' let it remain in 'tmpst' as it is already in the correct location and exit the loop.
11. After the inner loop, push the current element 'tmp' onto 'tmpst.'
12. Continue this process until all numbers in 'st' have been moved to 'tmpst' in a sorted manner.
13. If there are any remaining numbers in 'tmpst,' move them back to 'st' in sorted order using a similar process in the last while loop.
14. Finally, print the sorted numbers and pop out the elements of 'st.'

**Question 2:**

**Case A**

A screen shot of a computer

Description automatically generated

**Explanation:**

To determine the total number of edges for Case A, the process involves the addition of the specified size of weighted edges, which is 100 x 100. This is achieved through a for loop that generates random source and destination vertices along with weights. Subsequently, a counter is initialized to keep track of the cumulative number of edges. The algorithm iterates through all vertices in the graph using a for loop. For each node encountered in the linked list, the counter is incremented, and this process is reiterated until all nodes in the linked list are processed. This cycle is then repeated for the next vertex in the graph. Once all vertices have been processed, the final value of the counter, which is 10,000, is returned.

**Case B**

A screen shot of a computer

Description automatically generated

**Explanation:**

To determine the total count of edges for Case B, the procedure involves adding the specified size of weighted edges, totalling 600 x 600, utilizing a for loop to systematically generate random source and destination vertices along with corresponding weights. A counter is then initialized to monitor the cumulative number of edges systematically. The algorithm iterates through all vertices in the graph using a for loop, incrementing the counter for each node encountered in the linked list until all nodes have been processed. Subsequently, the algorithm transitions to the next vertex, repeating the aforementioned steps. Upon completing the processing of all vertices, the final counter value, amounting to 360,000, is returned as the calculated total number of edges for Case B.

**Case C**

A screen shot of a computer

Description automatically generated

**Explanation:**

To determine the total number of edges for Case C, the method involves the summation of the specified size of weighted edges, set at 1000 x 1000, utilizing a for loop to generate random source and destination vertices with corresponding weights. A counter is initialized to systematically track the cumulative edge count as the algorithm iterates through all vertices in the graph. For each encountered node in the linked list, the counter increments iteratively until all nodes are processed. The algorithm then proceeds to the next vertex in the graph, repeating the steps. Once all vertices are processed, the final counter value of 1,000,000 is returned as the computed total number of edges for Case C.

**Question 3:**

**A screenshot of a computer

Description automatically generated**

d)

**Explanation of Best, Average, Worst and Not Found Cases in Binary Search and Sequential Search**

**Binary Search**

1. Best Case: The best-case scenario occurs when the target number is precisely situated at the middle index of the sorted array. In this case, the algorithm performs just a single comparison to identify the target number, representing the most efficient execution of binary search. The output table designates this scenario as the "best."
2. Average Case: The target number is located somewhere in the array but not necessarily at the middle index. With each comparison, the algorithm systematically eliminates approximately half of the remaining entries. While the average complexity is more favorable than the worst case, it is higher than the best case. The output table designates this scenario as the "average."
3. Worst Case: The worst-case scenario unfolds when the target number is positioned at either extreme end of the array, either the first or final index. In such instances, the algorithm necessitates the maximum number of comparisons to pinpoint the target. The worst-case complexity is represented by log2(SIZE), signifying the maximum number of iterations, where SIZE denotes the size of the array. The output table designates this scenario as the "worst."
4. Not Found Case: If the target element is not discovered during this process, the algorithm continues to iteratively refine the search space until it becomes an empty interval. In this situation, the algorithm returns a special value, represented as -1, signaling that the wanted value is not present in the sorted array.

**Sequential Search**

1. Best Case: The target number is located at the first index of the sorted array, which the time complexity is the lowest as the algorithm needs to run only one time to compare. The output table indicates "best" for this.
2. Average Case: The target number is situated somewhere in the middle of the sorted array. The time complexity is higher than the best case but lower than the worst case. To locate the desired number, the algorithm must traverse approximately half of the array. The output table designates this scenario as "average."
3. Worse Case: The target number is located at the last index of the sorted array. In this case, the algorithm must explore the entire array and perform SIZE comparisons. This result is the highest time complexity among all cases. The output table indicates “worst” for this.
4. Not Found Case: If the target element is not encountered throughout this process, the algorithm concludes that the element is not present in the collection. To signify this absence, the algorithm commonly returns a special value, denoted as -1, indicating that the search did not locate the desired element in the given collection.

**Explanation of the Output Based on Table 1**

1. Size = 10:
2. Complexity for sequential search is **1**. The **best** case as the target number is located at the first index.
3. Complexity for binary search is **3**. The **worst** case as the target number is found with the largest number of iterations.
4. Size = 100:
5. Complexity for sequential search is **-1**. The target number is **not found**.
6. Complexity for binary search is **-1**. The target number is **not found**.
7. Size = 1000:
8. Complexity for sequential search is **799**. The **average** case as the target number is located somewhere at the middle of the sorted array.
9. Complexity for binary search is **8**. The **average** case as the target number is located somewhere but not in the middle of the sorted array.
10. Size = 10000:
11. Complexity for sequential search is **8875**. The **average** case as the target number is located somewhere at the middle of the sorted array.
12. Complexity for binary search is **10**. The **average** case as the target number is located somewhere but not in the middle of the sorted array.
13. Size = 100000:
14. Complexity for sequential search is **-1**. The target number is **not found**.
15. Complexity for binary search is **-1**. The target number is **not found**.
16. Size = 1000000:
17. Complexity for sequential search is **849447**. The **average** case as the target number is located somewhere at the middle of the sorted array.
18. Complexity for binary search is **17**. The **average** case as the target number is located somewhere but not in the middle of the sorted array.
19. Size = 10000000:
20. Complexity for sequential search is **1948692**. The **average** case as the target number is located somewhere at the middle of the sorted array.
21. Complexity for binary search is **23**. The **worst** case as the target number is found with the largest number of iterations.
22. Size = 100000000:
23. Complexity for sequential search is **83952272**. The **average** case as the target number is located somewhere at the middle of the sorted array.
24. Complexity for binary search is **27**. The **worst** case as the target number is found with the largest number of iterations.

All in all, sequential search is more effective for smaller sizes due to its better sequential cost compared to binary search. It performs well in best and average cases, while binary search is preferable for larger sizes because it has a lower time complexity, making it more efficient in handling larger datasets, despite having more unfavourable worst-case scenarios.

**APPENDIX 1**

**Marking Rubrics**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Component Title** | Assignment: Tasks 1 to 3 | | | **Percentage (%)** | **15** | |
| **Criteria** | **Score and Descriptors** | | | | **Weight (%)** | **Marks** |
| **(5)** | **(3-4)** | **(0-2)** | |
| Task 1) Stack-based Sorting | Completed 100% of  task requirements  &  Delivered on time,  and in correct format  &  Student explained perfectly in results section | Completed 100% of  task requirements  &  Delivered on time,  and in correct format  &  Student has minor problem in explanation in result section | Completed less than 50% of the requirements.  or  Runtime issues  or  Does not comply with requirements (does something other than requirements).  or  Student does not explain enough in result section | | **5** |  |
| Task 2)  Graph Data structure | Completed 100% of  task requirements  &  Delivered on time,  and in correct format  &  Student explained perfectly in results section | Completed 100% of  task requirements  &  Delivered on time,  and in correct format  &  Student has minor problem in explanation in result section | Completed less than 50% of the requirements.  or  Runtime issues  or  Does not comply with requirements (does something other than requirements).  or  Student does not explain enough in result section | | **5** |  |
| Task 3) Search Algorithms | Completed 100% of  task requirements  &  Delivered on time,  and in correct format  &  Student explained perfectly in results section | Completed 100% of  task requirements  &  Delivered on time,  and in correct format  &  Student has minor problem in explanation in result section | Completed less than 50% of the requirements.  or  Runtime issues  or  Does not comply with requirements (does something other than requirements).  or  Student does not explain enough in result section | | **5** |  |
| **TOTAL** | | | | | **15** |  |

Note to students: Please print out and attach this appendix together with the submission of coursework.