

SCHOOL OF COMPUTER SCIENCES

ACADEMIC SESSION 2016 / 2017, SEMESTER 2

CPT 113 – PROGRAMMING METHODOLOGY & DATA STRUCTURES

ASSIGNMENT 3

DUE DATE : 15th MAY 2017

NAME : CHONG KAR HIE

MATRIC NUMBER : 132893

GROUP : D

LECTURER’S NAME : MOHD AZAM OSMAN

Table of content

|  |  |
| --- | --- |
| Content | Page number |
| Introduction / Problem explanation | 1 |
| Problem analysis   * Inputs * Outputs * Processes * Test data | 2  2  2  3-6 |
| Solution descriptions by using stacks and recursions | 6-7 |
| Discuss the relationship between stacks and recursions methods in solving this problem | 7-8 |
| Relate the way stack organizes the search for a sequence of flights to the way a recursive algorithms organize the search | 8-9 |
| Result analysis   * By using stacks method * By using recursions method | 9  10 |
| System design   * Interface Design * Module/Function/Methods diagram * Data Structure Design * Flight Diagram | 11  11  12-13  15 |
| Description of selected functions | 13-14 |
| C++ code (source code) | 16-28 |
| Print screen of the program | 29-31 |

**Introduction / Problem explanation**

The purpose of this assignment is to illustrate how stack and recursion work in finding a sequence of flights in an airline system. A company named Flying Traveller Airline Company (FTAir) wanted a program to aid customers, process the requests of those customers includes requested origins and requested destinations within available cities and available one way flights. The program will return a sequence of flights to the customer by 2 different types of approach, which are stack and recursion.

**Problem analysis:**

**Input:-**

Fixed input:

1. Text file for available cities and visit status
2. Text file for available one way flights
3. Text file for requested flights

**Output:-**

1. Sequence of flights by stack
2. Sequence of flights by recursion
3. List of all available cities
4. List of all one way flights

**Process:**

First, the program will read all the available cities and their visit status into list which is a linked list, origins and destinations of all one way flights into orindeslist which is a linked list, and requested origins and requested destinations of all scenarios into scenarios which is a linked list.

In the program’s main menu, there are 5 selections.

For the first selection, the program will then show the sequence of flights for all of the requested scenarios which the problem is solved by using stacks.

For the second selection, the program will then show the sequence of flights for all of the requested scenarios which is solved by using recursions.

For the third selection, the program will then show all available cities according to the data read from text file.

For the fourth selection, the program will then show all one way flights according to the data read from the text file.

For the fifth selection, user may terminate the program.

**Test data**

Cities.txt:

AOR

BTU

JHB

KBR

BKI

KUL

TGG

KCH

LBU

LGK

MYY

PEN

SDK

SBW

TWU

PairesOfCities.txt

KUL PEN

KUL JHB

KUL AOR

KUL KBR

KUL BKI

KUL KCH

KUL LGK

KUL TWU

KUL SBW

KUL MYY

KUL TGG

AOR KUL

BTU JHB

BTU KUL

BTU KCH

BTU BKI

JHB KUL

JHB KCH

JHB LBU

KBR SDK

KBR SBW

KBR KUL

BKI JHB

BKI KUL

BKI TWU

BKI PEN

TGG JHB

TGG KUL

KCH BTU

KCH KUL

KCH LGK

KCH PEN

KCH MYY

LBU JHB

LBU KUL

LBU MYY

LGK JHB

LGK KUL

LGK BKI

LGK PEN

LGK KCH

MYY BKI

MYY KUL

MYY TWU

MYY KBR

PEN KUL

PEN KCH

PEN JHB

PEN BKI

SDK JHB

SDK BKI

SDK KUL

SBW JHB

SBW KUL

SBW KCH

TWU JHB

TWU BKI

TWU KUL

TWU KCH

Scenario.txt

KUL MYY

AOR PEN

BTU PEN

PEN JHB

TWU JHB

KCH KUL

SDK PEN

KUL PAR

**Solution descriptions by using stacks and recursions methods:**

**Stacks:-**

First, all of the requested origins and requested destinations is read into scenarios which is a linked list object. There will be a for loop to determine which scenario will be execute first. A pair of requested origin and requested destination, and 2 linked list objects which are list and orindeslist will be passed into a function of stackType which is searchPath in order to make use of both of the linked list objects. First, the stack is initialized to make sure that nothing is inside the stack. Then, the visit status of all of the cities will be marked as “NO” which is not visited. After that, the requested origin will be assigned into the first variable of temporary struct x, and struct x will be pushed into the stack, and the visit status of the requested origin will be marked as “YES” which is visited. Then, there is a while loop with a condition of when the stack is not empty and the first variable of the struct pointed by the pointer stackTop in the stack is not destination, if the next city of the city which is in the struct pointed by stackTop is “NO\_CITY” which there is no next available city for the current city, the top element of the stack will be popped out, else, if there is next available city for the city in struct pointed by stackTop, that next city will be assigned into the first variable of temporary struct x, and the struct x will be pushed into the stack, and the visit status of the city will be marked as “YES”. Either when the stack is empty or the destination is found, the while loop will be exited. After the while loop, if the stack is empty, the program will tell the users which there is no such path from the requested city and requested destination, else, if the stack is not empty and the destination is found, the stack will be reversed by using a temporary pointer and the address of the temporary pointer will be assigned back to stackTop pointer, then the stack will be displayed out. Then, the program will assigned the next pair of requested origin and requested destination and the same 2 linked list objects into the searchPath function until the last pair of requested origin and requested destination.

**Recursion:-**

First, all off the requested origins and requested destinations is read into scenarios which is a linked list object. There will be a for loop to determine which scenario will execute first, visit status of all of the cities will be marked as “NO” which is mean they are not visited before each of the scenarios is executed, and all of the scenarios will be executed sequentially. A pair of requested origin and requested destination, and 2 linked list objects which are list and orindeslist will be passed into the searchrecursion function which is a boolean function in order to make use of both of the linked list objects. First, the requested origin is marked as visited which is “YES”. Because this method is a recursion method, if the origin in the searchrecursion function is same as the requested destination, the requested destination will be printed out, boolean of the variable result will be set as true, and it will return the boolean value to the previous recursion function which called the current recursion function. Else, the boolean of the variable done will be set to false, then the next city of origin of each recursion will be assigned to a variable next. Then, if the city assigned to the variable is not “NO\_CITY”, meaning that there are next available city, the program will print the origin of the respective recursion. Then, a while loop with conditions of when the boolean variable done is false (as !done == true) and the city in the variable next is not “NO\_CITY”, the searchrecursion function which is the recursive function with the parameter of the variable next, the requested destination, both of the linked list objects list and orindeslist, so that the program can search the sequence of the flight from the requested origin to the requested destination and return a boolean value to the boolean variable done. If it does not have next available city, which is “NO\_CITY”, it will return false, else if it does have next available city and arrives at the requested destination, it will return true. If the boolean value of the boolean variable done is true, it will not having the backtrack function which will search for the next available city and will exit the while loop, else if the boolean value of the boolean variable done is false, it will search for the next available city, and the while loop will not be exited and will execute the while loop until the a sequence of flights from requested origin and requested destination is found or there is no available sequence of flights. If the searchrecursion function returns a boolean value of false to the variable success in the main function, the program will print out a sentence that there is no path from the requested city to the requested destination.

**Discuss the relationship between stacks and recursions methods in solving this problem**

Both of the methods use the same algorithm to select an unvisited city. For example, the stacks method will find the unvisited city if it is available, push it into the stack until the requested destination is found at the stackTop pointer, and the recursion method will search for the unvisited city adjacent to the current city, print out the current city and returning a boolean value of true after it found the requested destination, thus returning the boolean value of true to the first call of the recursion method. The same thing is, they will always visit the same city because of the cities are in an order. So, the sequence of flights which is found by using stacks method as similar to the sequence of flights found by using recursions method.

Stacks method is actually used to implement the recursions method. When the recursion function is called, it must remember or store some information. So, the more the recursion function is being called, the more the information must be remembered in each function, so it will be like a stack, and stacks method is implemented in the use for recursions method.

In the recursion part of this program, when the current city is not the requested destination and it has next available cities, the searchrecursion function will be called again. At this point, the information such as the current city, the requested destination and the next city need to be kept before the execution of the searchrecursion function, and this will be repeated until the current city is same as the requested destination, then a boolean value of true will be return to previous searchrecursion function, and the recursions method will be ended. When the searchrecursion is being called, all the information will be stored and pushed into a stack, and when the searchrecursion function mades a return, the information will be popped. So, this is how stacks method is actually implemented in the recursions method.

**Relate the way stack organizes the search for a sequence of flights to the way a recursive algorithms organize the search**

For visiting a new city, the stacks method goes to or visits a new city by pushing the city to a stack first before marking the city as a visited city, while the recursions method goes to a new city by calling itself which is searchrecursion function again with the parameter of the next city and the requested destination.

For backtracking, both of the methods will visit a city which is not visited from the current city. The similarity for both of the methods is, the methods must backtrack to the previous city if there is no next available city for the current city. For stacks method, when there is no next available city left for the current city, the method will backtrack to the previous city by popping out the current city out from the stack. When there is next available cities for the current city, one of next available cities will be pushed into the stack. When the current city is the requested destination, the stack will be reversed and display out the sequence of flights from requested origin to requested destination. For recursions method, when there are available cities next to the current city, current city will be printed out, and the searchrecursion function will be called again with parameter of next city, requested destination, and both of the linked list objects. When there is no next available city next to the current city, current city will not be printed out, instead, returning a boolean value of false back to the previous searchrecursion function to search for next available city for the previous city. When the current city is the requested destination, the requested destination will be printed out and returning a boolean value of true. Thus, ending the recursions method.

For the termination of the both methods, the stacks method will be ended when the requested destination has been found or if the stack is empty, while the recursions method will be ended when the destination has been found and returning a boolean value of true to all the previous searchrecursion function.

**Result analysis**

Request: From PEN to JHB.

**By stacks method:**

JHB = JHB (Requested destination)

reversestack()

displaystack()

Origin = KUL

Destination = JHB

Visit status of JHB = “NO”

Return JHB

KUL != JHB

orindeslist.getnext(KUL, list) = JHB

push(JHB)

list.markvisited(JHB)

Origin = PEN

Destination = KUL

Visit status of KUL = “NO”

Return KUL

searchPath(PEN, JHB, list, orindeslist)

push(PEN)

list.markvisited(PEN)

orindeslist.getnext(PEN, list) = KUL

push(KUL)

list.markvisited(KUL)

**By recursions method:**

list.markvisited(JHB)

JHB == JHB

Print JHB

return true

Origin = KUL

Destination = JHB

Visit status of JHB = “NO”

return JHB

list.markvisited(KUL)

KUL != JHB

done = false

orindeslist.getnext(KUL, list) = JHB

JHB != “NO\_CITY”, print KUL

done = search recursion(JHB, JHB, list, orindeslist)

done = true

result = true

return result

Origin = PEN

Destination = KUL

Visit status of KUL = “NO”

return KUL

searchrecursion(PEN, JHB, list, orindeslist)

list.markvisited(PEN)

PEN != JHB

done = false

orindeslist.getnext(PEN, list) = KUL

KUL != “NO\_CITY”, print PEN

done = searchrecursion(KUL, JHB, list, orindeslist)

done = true

result = true

return result

**System Design**

**Interface Design:-**

Function name: menu()

* This function is used to display the main menu which all of the selections which are available in the program
* User may choose 1 for solving problem by stacks, 2 for solving problem by recursions, 3 for showing all available cities, 4 for showing all available one way flights, and 0 to exit the program

**Module/Function/Methods diagram:-**

END

Option 5: exit the program

Main menu

After searching the path,

reversestack()

displaystack()

Option 4: show all available one way flights

orindeslist.displayall2()

Option 3: show all available cities

list.displayall()

Option 2: solve problem by recursions

Get number of scenarios by scenarios.getcount()

list.unvisitall()

Get requested origin and destination

success = searchrecursion(…)

Option 1: solve problem by stacks

Get number of scenarios by scenarios.getcount()

Get requested origin and destination

stack.searchPath(…)

menu()

Read data from text files:

* void readFile(…)
* void readFile2(…)
* void readFilesce(,,,)

START

**Data Structure Design:-**

Class stackType

Description: to search a sequence of flights by using the stacks method, reverse the stack and then display all elements in the stack in sequence.

Variables:-

1. nodeTypes<type> \*stackTop – pointer which will always point to the top element
2. nodeTypes<type> \*tempstack – pointer used to reverse the stack

Functions:-

1. void initializeStack() – to initialize the stack
2. bool isEmptyStack() const – to check whether there is a stack or not
3. void push(const type&) – to push an element to the stack
4. type top() const – to return the top element of the stack
5. void pop() – to remove the top element of the stack
6. void searchPath(string, string, linkedListType<type>, linkedListType<type>) – to find a sequence of flights by using the stacks method
7. void displaystack() – to display the stack
8. void reversestack() – to reverse the stack
9. void pushtemp(const type&) – to push the information of the stack into a temporary pointer in order to reverse the stack

struct nodeTypes

Description: to store a structure which stores the name of the city and a pointer which is being pushed into the stack

Variables:-

1. type info – a struct which is used to store information
2. nodeTypes<type> \*link – a pointer to point to the next node

Class linkedListType

Description: to store all of the information, search information for the use of stacks method and recursions method

Variables:-

1. nodeType<type> \*first – pointer which points to the first node of linked list
2. nodeType<type> \*last – pointer which points to the last node of linked list
3. int count – to store the number of entries of the information

Functions:-

1. void insert(type) – to insert information as the last node of linked list
2. void displayall() – to display all of the available cities
3. void displayall2() – to display all of the available one way flights
4. void markvisited(string) – to mark the visit status of a city to visited
5. int getcount() – to get the number of entries of the information
6. string getori(int) – to get the requested origin based on the scenario
7. string getdesti(int) – to get the requested destination based on the scenario
8. void unvisitall() – to mark the visit status of all cities to not visited
9. bool checkvisited(string) – to check the city passed into this function is visited or not
10. string getnext(string, linkedListType<type>) – to get the next available city of the current city which is passed into this function

struct nodeType

Description: to store a structure which stores information and the pointer which points to the next node

Variables:-

1. type info – a structure to store information. For list, it stores all cities, visit status and counter, for list, it stores origin and destination of one way flights, and counter, for scenarios, it stores requested origins and destinations, and counter.
2. nodeType<type> \*link – a pointer for struct nodeType in order to point to the next node

struct info

Description: a generalized structure to store information that is read from the text files

Variables:-

1. string variable1 – For list: city, for orindeslist: origin, for scenario: requested origin, for stack: stored city
2. string variable 2 – For list: visit status, for orindeslist: destination, for scenario: requested destination
3. int variable3 – For list, orindeslist, scenario = counter for each node

**Description of selected functions**

void readFile(linkedListType<type>&)

* this function will read the first text file and save the information into the first linked list: list

void readFile2(linkedListType<type>&)

* this function will read the second text file and save the information into the second linked list: orindeslist

void readFilesce(linkedListType<type>&)

* this function will read the third text file and save the information into the third linked list: scenarios

bool searchrecursion(string, string, linkedListType<type>, linkedListType<type>)

* this function is a recursive function which the problem will be solved by using recursions method
* requested origins, requested destinations, list and orindeslist will be passed into this function
* the function will then solve the problem by using recursions methods
* the function will return a boolean value
* if the function returns true, it means that there is a possible sequence of flight
* if the function returns false, there is no such possible sequence of flight

**Route diagram**

BKI

KUL

PEN

SBW

TWU

LGK

TGG

SDK

KBR

LBU

AOR

JHB

BTU

MYY

KCH

**C++ code (source code):**

**main.cpp**

1. #include <iostream>
2. #include <cstdlib>
3. #include <windows.h>
4. #include <fstream>
5. #include "stackType.h"
6. #include "linkedListType.h"
8. **using** **namespace** std;
10. **template** <**class** type>
11. **bool** searchrecursion(string, string, linkedListType<type>, linkedListType<type>);
12. **template** <**class** type>
13. **void** readFile(linkedListType<type>&);
14. **template** <**class** type>
15. **void** readFile2(linkedListType<type>&);
16. **template** <**class** type>
17. **void** readFilesce(linkedListType<type>&);
18. **void** welcome();
19. **void** menu();
20. **int** main()
21. {
22. **int** selection, count;
23. string ori, desti;
24. **char** proceed = 'Y';
25. **bool** success;
26. welcome();
27. stackType<info> stack;
28. linkedListType<info> list; //list is for linked list which is used to store available cities and visit status
29. linkedListType<info> orindeslist; //orindeslist is for linked list which is used to store origin and destination
30. linkedListType<info> scenarios; //scenarios is for linked list which is used to store requested origin and destination
31. readFile(list); //Read textfile for list
32. readFile2(orindeslist); //Read textfile for orindeslist
33. readFilesce(scenarios); //Read textfile for scenarios
34. **while**(proceed == 'Y' || proceed == 'y')
35. {
36. menu(); //Function to show the main menu
37. cin >> selection;
38. **while**(selection < 0 || selection > 4 || !cin || cin.get() != '\n') //Validation for menu
39. {
40. cin.clear();
41. cin.ignore(100, '\n');
42. cout << "Invalid input. Please try again: ";
43. cin >> selection;
44. }
45. **switch**(selection)
46. {
47. **case** 0: system("cls");
48. cout << "Exiting program. Have a nice day!" << endl;
49. Sleep(600);
50. **return** 0;
51. **case** 1: system("cls");
52. cout << "By using stack: " << endl;
53. count = scenarios.getcount(); // Get total number of scenarios
54. **for**(**int** i = 0; i < count; i++)
55. {
56. cout << "Scenario " << i+1 << ":" << endl;
57. ori = scenarios.getori(i+1); //Get the origin of the requested scenario
58. desti = scenarios.getdesti(i+1); //Get the destination of the requested scenario
59. cout << "From " << ori << " to " << desti << endl;
60. stack.searchPath(ori, desti, list, orindeslist); //Function to find the path by using stack
61. cout << endl;
62. }
63. system("pause");
64. **continue**;
66. **case** 2: system("cls");
67. cout << "By using recursion: " << endl;
68. count = scenarios.getcount();
69. **for**(**int** j = 0; j < count; j++)
70. {
71. cout << "Scenario " << j+1 << ":" << endl;
72. ori = scenarios.getori(j+1); //get the origin of the requested scenario
73. desti = scenarios.getdesti(j+1); //get the destination of the requested scenario
74. cout << "From " << ori << " to " << desti << endl;
75. list.unvisitall(); //Function to ensure that all cities are not visited at beginning of the search path
76. success = searchrecursion(ori, desti, list, orindeslist); //Function to find the path by using recursion
77. **if**(success == **false**)
78. {
79. cout << endl;
80. cout << "The path is not exist." << endl;
81. }
82. cout << endl;
83. }
84. system("pause");
85. **continue**;
87. **case** 3: system("cls");
88. list.displayall(); //Function to display all available cities
89. system("pause");
90. **continue**;
92. **case** 4: system("cls");
93. orindeslist.displayall2(); //Function to display all of the available one way flights
94. system("pause");
95. **continue**;
96. }
98. cout << "Do you want to start again? Y/y for yes, any key to exit: "; //For the loop of the whole program
99. cin >> proceed;
100. }
101. **return** 0;
102. }
104. **void** welcome()
105. {
106. cout << endl << endl << endl << "\t\t\t";
107. **char** welcome1[] = {'W', 'e', 'l', 'c', 'o', 'm', 'e', ' ', 't', 'o', '.', '.', '.'};
108. **for**(**int** i = 0; i < 13; i++)
109. {
110. cout << welcome1[i];
111. Sleep(100);
112. }
113. **for**(**int** j = 0; j < 3; j++)
114. {
115. system("cls");
116. cout << endl << endl << endl << "\t\t\t";
117. cout << "Welcome to..." << endl << endl;
118. cout << "\t\t\teFTAir - Plane Routes Checking Program" << endl << endl;
119. Sleep(300);
120. system("cls");
121. Sleep(300);
123. }
124. cout << endl << endl << endl;
125. cout << "\t\t\tWelcome to..." << endl << endl;
126. cout << "\t\t\teFTAir - Plane Routes Checking Program" << endl << endl;
127. cout << "\t\t\tBy - Flying Traveller Airline Company (FTAir)" << endl;
128. Sleep(600);
129. }
131. **void** menu() //Function to show the main menu
132. {
133. system("cls");
134. cout << "Please choose one of the following methods: " << endl;
135. cout << "1. Using stack" << endl;
136. cout << "2. Using recursion" << endl << endl;
137. cout << "Others..." << endl;
138. cout << "3. Show all available cities" << endl;
139. cout << "4. Show all available one way flights" << endl << endl;
140. cout << "0. Exit" << endl;
141. cout << "Please enter your choice: ";
142. }
144. **template** <**class** type>
145. **void** readFile(linkedListType<type>& list) //Read textfile for list
146. {
147. **int** z = 0;
148. ifstream inFile;
149. info city;
150. inFile.open("Cities.txt");
151. **while**(!inFile.eof())
152. {
153. inFile >> city.variable1;
154. city.variable2 = "NO";
155. city.variable3 = ++z;
156. list.insert(city);
157. }
158. inFile.close();
159. }
161. **template** <**class** type>
162. **void** readFile2(linkedListType<type>& orindeslist) //Read textfile for orindeslist
163. {
164. **int** z = 0;
165. ifstream inFile;
166. info orindes;
167. inFile.open("PairsOfCities.txt");
168. **while**(!inFile.eof())
169. {
170. inFile >> orindes.variable1 >> orindes.variable2;
171. orindes.variable3 = ++z;
172. orindeslist.insert(orindes);
173. }
174. inFile.close();
175. }
177. **template** <**class** type>
178. **void** readFilesce(linkedListType<type>& scenarios) //Read textfile for scenarios
179. {
180. **int** z = 0;
181. ifstream inFile;
182. info scenario;
183. inFile.open("Scenario.txt");
184. **while**(!inFile.eof())
185. {
186. inFile >> scenario.variable1 >> scenario.variable2;
187. scenario.variable3 = ++z;
188. scenarios.insert(scenario);
189. }
190. inFile.close();
191. }
193. **template** <**class** type>
194. **bool** searchrecursion(string ori, string desti, linkedListType<type> list, linkedListType<type> orindeslist) //Function to search path by recursion
195. {
197. **bool** done, finalresult;
198. string next; //Variable of storing next city
199. **if**(!list.exist(ori))
200. {
201. cout << "Requested origin does not exist";
202. **return** **false**;
203. }
204. **if**(!list.exist(desti))
205. {
206. cout << "Requested destination does not exist.";
207. **return** **false**;
208. }
210. list.markvisited(ori); //Function to mark city as visited
211. **if**(ori == desti) //If the origin is same as the destination
212. {
213. cout << desti;
214. finalresult = **true**;
215. cout << endl;
216. }
217. **else**
218. {
219. done = **false**;
220. next = orindeslist.getnext(ori, list); //Function to get next city
221. **if**(next != "NO\_CITY") //Print origin if it has unvisited cities
222. cout << ori << "->";
223. **while**(!done && next != "NO\_CITY") //While loop if the condition is true
224. {
225. done = searchrecursion(next, desti, list, orindeslist); //Recursion starts here
226. **if**(!done) //This will be executed based on the boolean value returned by the recursion
227. next = orindeslist.getnext(ori, list); //Function to get next available city if the original next city has no any available cities
228. }
229. finalresult = done;
230. }
231. **return** finalresult;
232. }

**stackType.h**

1. #ifndef STACKTYPE\_H
2. #define STACKTYPE\_H
3. #include <iostream>
4. #include "linkedListType.h"
6. **using** **namespace** std;
8. **template** <**class** type>
9. **struct** nodeTypes
10. {
11. type info;
12. nodeTypes<type> \*link;
13. };
15. **template** <**class** type>
16. **class** stackType
17. {
18. **public**:
19. **void** initializeStack(); //Function to initialize the stack
20. **bool** isEmptyStack() **const**; //Function to check the existence of a stack pointed by stackTop
21. **void** push(**const** type&); //Function to push a variable/struct into a stack
22. type top() **const**; //Function to return the variable/struct pointed by stackTop pointer
23. **void** pop(); //Function to pop a variable/struct pointed by the stackTop pointer
24. stackType(); //Default constructor
25. ~stackType(); //Destructor
26. **void** searchPath(string, string, linkedListType<type>, linkedListType<type>); //Function in stack in order to search the flight path and store in a stack
27. **void** displaystack(); //Function to display a stack
28. **void** reversestack(); //Function to reverse a stack
29. **void** pushtemp(**const** type&); //Function to push a variable/struct into a temporary stack in order to reverse the stack
30. **private**:
31. nodeTypes<type> \*stackTop; //Pointer to point to the top element of the stack
32. nodeTypes<type> \*tempstack; //Temporary pointer in order to reverse the stack
33. };
35. **template** <**class** type>
36. stackType<type>::stackType() //Default constructor
37. {
38. stackTop = NULL;
39. tempstack = NULL;
40. }
42. **template** <**class** type>
43. stackType<type>::~stackType()
44. {
45. initializeStack();
46. }
48. **template** <**class** type>
49. **bool** stackType<type>::isEmptyStack() **const** //Function to check the existence of a stack pointed by stackTop
50. {
51. **return** (stackTop == NULL);
52. }
54. **template** <**class** type>
55. **void** stackType<type>::initializeStack() //Function to initialize the stack
56. {
57. nodeTypes<type> \*temp;
59. **while**(stackTop != NULL)
60. {
61. temp = stackTop;
62. stackTop = stackTop->link;
63. **delete** temp;
64. }
65. tempstack = NULL;
66. }
68. **template** <**class** type>
69. **void** stackType<type>::push(**const** type& newElement) //Function to push a variable/struct into a stack
70. {
71. nodeTypes<type> \*newNode;
72. newNode = **new** nodeTypes<type>;
73. newNode->info= newElement;
74. newNode->link = stackTop;
75. stackTop = newNode;
76. }
78. **template** <**class** type>
79. type stackType<type>::top() **const**
80. {
81. assert(stackTop != NULL);
83. **return** stackTop->info.variable1;
84. }
86. **template** <**class** type>
87. **void** stackType<type>::pop() //Function to pop a variable/struct pointed by the stackTop pointer
88. {
89. nodeTypes<type> \*temp;
91. **if**(stackTop != NULL)
92. {
93. temp = stackTop;
94. stackTop = stackTop->link;
95. **delete** temp;
96. }
97. **else**
98. cout << "Cannot remove from an empty stack." << endl;
99. }
101. **template** <**class** type>
102. **void** stackType<type>::searchPath(string ori, string desti, linkedListType<type> list, linkedListType<type> orindeslist) //Function in stack in order to search the flight path and store in a stack
103. {
104. info x;
105. **if**(!list.exist(ori))
106. {
107. cout << "Requested origin does not exist." << endl;
108. cout << "The path is not exist." << endl;
109. **return**;
110. }
111. **if**(!list.exist(desti))
112. {
113. cout << "Requested destination does not exist." << endl;
114. cout << "The path is not exist." << endl;
115. **return**;
116. }
117. x.variable1 = ori;
118. initializeStack(); //For every new scenario, need to make sure the stack is empty
119. list.unvisitall(); //For every new scenario, need to make sure all the cities are not visited
120. push(x); //Push the requested origin into the stack
121. list.markvisited(x.variable1); //Mark the requested origin as visited
122. **while**(!isEmptyStack() && stackTop->info.variable1 != desti) //While loop
123. {
124. **if**(orindeslist.getnext(stackTop->info.variable1, list) == "NO\_CITY") //Pop the variable/struct when it doesn't have any available cities
125. pop();
126. **else**
127. {
128. x.variable1 = orindeslist.getnext(stackTop->info.variable1, list); //Search for the next city
129. push(x); //Push the next city
130. list.markvisited(x.variable1); //Mark the next city as visited
131. }
132. }
133. **if**(isEmptyStack()) //If the stack is empty
134. {
135. cout << "The path is not exist." << endl;
136. }
137. **else**
138. {
139. reversestack(); //Function to reverse the stack
140. displaystack(); //Function to display the reversed stack
141. }

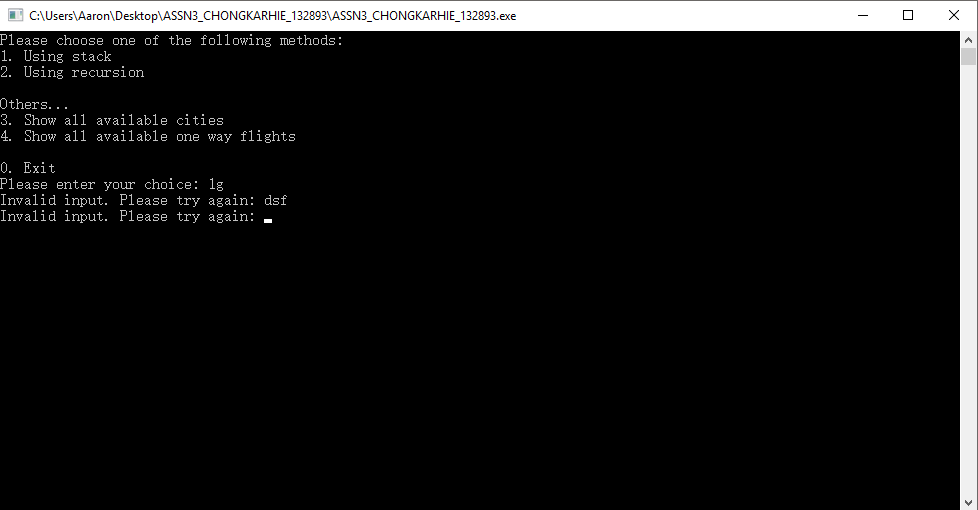
144. }
146. **template** <**class** type>
147. **void** stackType<type>::displaystack() //Function to display a stack
148. {
149. nodeTypes<type> \*current;
150. current = stackTop;
151. **while**(current != NULL)
152. {
153. cout << current->info.variable1 << "->";
154. current = current->link;
155. }
156. cout << '\b' << '\b';
157. cout <<  " " << " ";
158. cout << endl;
159. }
161. **template** <**class** type>
162. **void** stackType<type>::reversestack() //Function to reverse a stack
163. {
164. info item;
165. **while**(!isEmptyStack())
166. {
167. item = stackTop->info;
168. pop(); //Pop the element pointed by stackTop
169. pushtemp(item); //Push the element into another stack pointed by tempstack
170. }
172. stackTop = tempstack; //Assign the address of tempstack to stackTop so that stackTop will point to the reversed stack
173. }
175. **template** <**class** type>
176. **void** stackType<type>::pushtemp(**const** type& newElement) //Function to push a variable/struct into a temporary stack in order to reverse the stack
177. {
178. nodeTypes<type> \*newNode;
179. newNode = **new** nodeTypes<type>;
180. newNode->info = newElement;
181. newNode->link = tempstack;
182. tempstack = newNode;
183. }
185. #endif

**linkedListType.h**

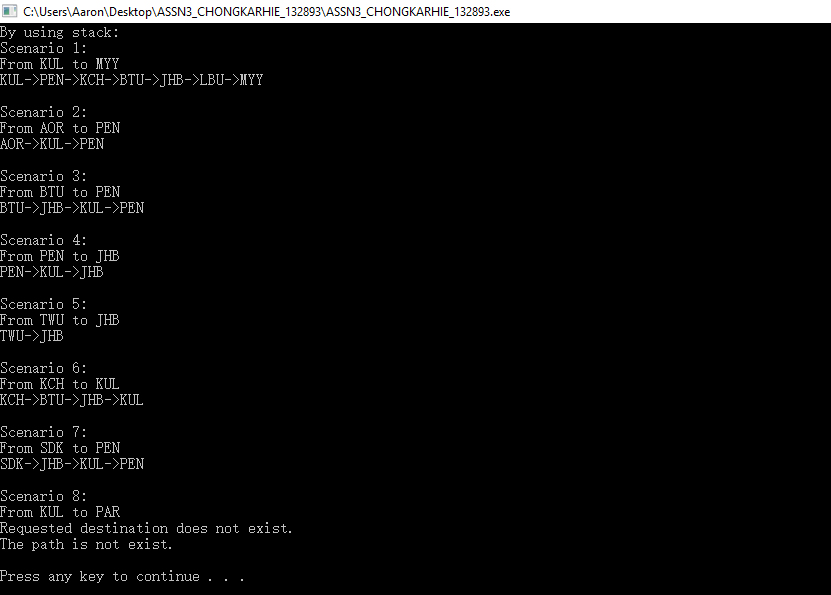
1. #ifndef LINKEDLISTTYPE\_H
2. #define LINKEDLISTTYPE\_H
3. #include <iostream>
4. #include <cassert>
6. **using** **namespace** std;
8. **struct** info //Generalized struct
9. {
10. string variable1; //For list: city, for orindeslist: origin, for scenario: requested origin, for stack: stored city
11. string variable2; //For list: visit status, for orindeslist: destination, for scenario: requested destination
12. **int** variable3; //For list, orindeslist, scenario = counter for each node
13. };
15. **template** <**class** type>
16. **struct** nodeType
17. {
18. type info;
19. nodeType<type> \*link;
20. };
22. **template** <**class** type>
23. **class** linkedListType
24. {
25. **public**:
26. linkedListType(); //Default constructor
27. ~linkedListType(); //Destructor
28. **void** insert(type); //Function to insert as the last node of the linked list
29. **void** displayall(); //Function to display all available cities
30. **void** displayall2(); //Function to display all available one way flights
31. **void** markvisited(string); //Function to mark visit status of a city as visited
32. **int** getcount(); //Function to return the number of nodes
33. string getori(**int**); //Function to return the origin
34. string getdesti(**int**); //Function to return the destination
35. **void** unvisitall(); //Function to mark visit status of all cities as not visited
36. **bool** checkvisited(string); //Function to check visit status of a city
37. string getnext(string, linkedListType<type>); //Function to get next city
38. **bool** exist(string);
39. **protected**:
40. nodeType<type> \*first;
41. nodeType<type> \*last;
42. **int** count;
43. };
45. **template** <**class** type>
46. linkedListType<type>::linkedListType()
47. {
48. first = NULL;
49. last = NULL;
50. count = 0;
51. }
53. **template** <**class** type>
54. linkedListType<type>::~linkedListType()
55. {
57. }
59. **template** <**class** type>
60. **void** linkedListType<type>::insert(type city) //Function to insert as the last node of the linked list
61. {
62. nodeType<type> \*newNode;
63. newNode = **new** nodeType<type>;
64. assert(newNode != NULL);
65. newNode->info = city;
66. **if**(first == NULL) //If there is no linked list
67. {
68. first = newNode;
69. last = newNode;
70. newNode->link = NULL;
71. }
72. **else**
73. {
74. last->link = newNode;
75. last = newNode;
76. newNode->link = NULL;
77. }
78. count++;
79. }
81. **template** <**class** type>
82. **void** linkedListType<type>::displayall() //Function to display all available cities
83. {
84. nodeType<type> \*current;
85. current = first;
86. cout << "List of cities available: " << endl << endl;
87. **while**(current != NULL)
88. {
89. cout << current->info.variable1 << endl;
90. current = current->link;
91. }
92. cout << endl << endl;
93. }
95. **template** <**class** type>
96. **void** linkedListType<type>::displayall2() //Function to display all available one way flights
97. {
98. nodeType<type> \*current;
99. current = first;
100. cout << "One way flights available: " << endl << endl;
101. **while**(current != NULL)
102. {
103. cout << current->info.variable1 << " -> " << current->info.variable2 << endl;
104. current = current->link;
105. }
106. cout << endl << endl;
107. }
109. **template** <**class** type>
110. **int** linkedListType<type>::getcount() //Function to return the number of nodes
111. {
112. **return** count;
113. }
115. **template** <**class** type>
116. string linkedListType<type>::getori(**int** zcount) //Function to return the origin
117. {
118. nodeType<type> \*current;
119. current = first;
120. **bool** found = **false**;
121. **while**(current != NULL && !found)
122. {
123. **if**(current->info.variable3 == zcount)
124. found = **true**;
125. **else**
126. current = current->link;
127. }
128. **if**(found)
129. **return** current->info.variable1;
130. }
132. **template** <**class** type>
133. string linkedListType<type>::getdesti(**int** zcount) //Function to return the destination
134. {
135. nodeType<type> \*current;
136. current = first;
137. **bool** found = **false**;
138. **while**(current != NULL && !found)
139. {
140. **if**(current->info.variable3 == zcount)
141. found = **true**;
142. **else**
143. current = current->link;
144. }
145. **if**(found)
146. **return** current->info.variable2;
147. }
149. **template** <**class** type>
150. **void** linkedListType<type>::unvisitall() //Function to mark visit status of all cities as not visited
151. {
152. nodeType<type> \*current;
153. current = first;
154. **while**(current != NULL)
155. {
156. current->info.variable2 = "NO";
157. current = current->link;
158. }
159. }
161. **template** <**class** type>
162. **void** linkedListType<type>::markvisited(string place) //Function to mark visit status of a city as visited
163. {
164. nodeType<type> \*current;
165. current = first;
166. **bool** found = **false**;
167. **while**(current != NULL && !found)
168. {
169. **if**(current->info.variable1 == place)
170. found = **true**;
171. **else**
172. current = current->link;
173. }
174. **if**(found)
175. current->info.variable2 = "YES";
177. }
179. **template** <**class** type>
180. **bool** linkedListType<type>::checkvisited(string place) //Function to check visit status of a city
181. {
182. nodeType<type> \*current;
183. current = first;
184. **bool** found = **false**;
185. **while**(current != NULL && !found)
186. {
187. **if**(current->info.variable1 == place)
188. found = **true**;
189. **else**
190. current = current->link;
191. }
192. **if**(found)
193. {
194. **if**(current->info.variable2 == "NO")
195. **return** **true**;
196. **else** **if**(current->info.variable2 == "YES")
197. **return** **false**;
198. }
199. }
201. **template** <**class** type>
202. string linkedListType<type>::getnext(string place, linkedListType<type> list) //Function to get next city
203. {
204. nodeType<type> \*current;
205. current = first;
206. **bool** found = **false**;
207. **while**(current != NULL && !found)
208. {
209. **if**(current->info.variable1 == place && list.checkvisited(current->info.variable2) == **true**)
210. found = **true**;
211. **else**
212. current = current->link;
213. }
214. **if**(found)
215. **return** current->info.variable2;
216. **else**
217. **return** "NO\_CITY";
218. }
220. **template** <**class** type>
221. **bool** linkedListType<type>::exist(string place)
222. {
223. nodeType<type> \*current;
224. current = first;
225. **bool** found = **false**;
226. **while**(current != NULL && !found)
227. {
228. **if**(current->info.variable1 == place)
229. found = **true**;
230. **else**
231. current = current->link;
232. }
233. **if**(found)
234. **return** **true**;
235. **else**
236. **return** **false**;
237. }
239. #endif

**Print screen of the program:**

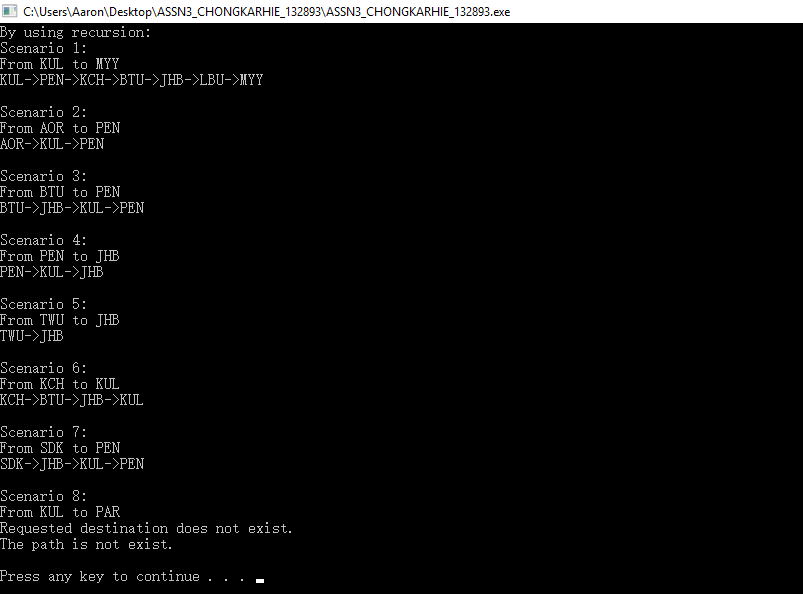
Main menu with invalid input



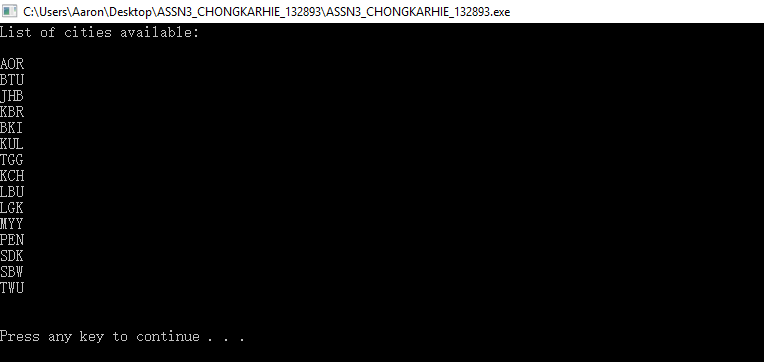
Option 1: Solve problem by using stacks



Option 2: Solve problem by using recursions



Option 3: Show all available cities



Option 4: Show all available one way flights

