ТУ-Варна

Дисциплина:ООП

Курсов проект по обектно ориентирано програмиране №21

C++

Програма за информация за летище

Изготвил: Георги Красимиров Илиев

Факултетен №61460149

Специалност: КСТ 2а

Проверил: ас.Владимирова

Задача:

I. Да се дефинира клас FlightInfo, с член данни: Id на полета, тръгва от, пътува за, час на тръгване, час на пристигане, цена и необходимите методи, конструктори и оператори.

II. Да се дефинира клас FlightData, с информация за всички полети - съхранени в контейнер с указатеи към обекти от клас FlightInfo. Да се дефинира конструктор с параметър име на файл, с данните от който да се запълва контейнера, подходящи методи и операции, които подпомагат реализирането на опциите от менюто на информационната система. Като:

- Намира всички полети тръгващи от зададен като парамегьр град и записва указатели към тях в контейнер, който връща като резултат;
- Намира всички полети пътуващи за зададен като параметър град и записва указатели към тях в контейнер, който връща като резултат;
- Търси има ли полет от зададен град до друг град, като връща стойност показваща броя на прехвърлянията: -1 - няма такъв полет, 0— директен, 1— с едно прехвърляне и т.н.

III. Да се дефинира клас UserRequest, описващ желанието за полет на клиент, с член данни: Id на искането, пътува от, пътува за, тръгване (най-рано), пристигане (най-късно) и необходимите методи и операции.

IV. Да се дефинира клас Users, с информация за всички постъпили запитвания на клиенти, съхранени в контейнер. Да се дефинира конструктор с параметър име на файл, с данните от който да се запълва контейнера и подходящи методи и операции, които подпомагат реализирането на опциите от менюто на информационната система.

V. Да се създадат обекти от класовете FlightData u Users. Да се организира меню с подходящи опции за информационна система за летище, като:

- за всички потребителски заявки, да се намери и изведе най-евтиният полет без прекъсване (или с прекъсване - ако няма друг), отговарящ на искането на потребителя, или информация, че няма такава възможност;
- добавя нов полет;
- добавя ново потребителско искане;
- корегира данните за полет по зададено Id;
- извежда информация за всички полети;
- извежда информация за всички клиентски искания;
- за потребителска заявка със зададено Id, извежда най-евтиният полет без прекъсване или какъвто има;
- намира и извежда информация за полети тръгващи от зададен град, след зададен час;

- намира от кой град има най-много полети;
- намира за кой град има най-много желаещи да пътуват пътници;
- запис на обновените данни във файл и др.

Съдържание

Въведение в графите и начин на имплементация 3

Използвани алгоритми 5

- Алгоритъм на търсене в дълбочина в граф (DFS) 5
- Алгоритъм на Дейкстра 8

Описание на класовете и техните функции 16

I. class AjacencyMatrix 16

II. Class FlightInfo 18

III. Клас FlightData 19

IV. клас UserRequest 23

V. Клас Users 23

Сорс код 24

Хедърен файл Adjacencymatrix.h 24

Сорс файл AdjacencyMatrix.cpp 25

Хедърен файл FlightInfo.h 29

Сорс файл FlightInfo.cpp 30

Хедърен файл FlightData.h 32

Сорс файл FlightData.cpp 33

Хедърен файл UserRequest.h 39

Сорс файл UserRequest.cpp 39

Хедърен файл User.h 41

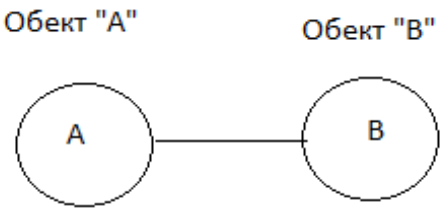
Сорс файл User.cpp 41

Главен файл Main.cpp 44

Въведение в графите и начин на имплементация

За решаване на проблема с намирането и извеждането на най-евтиният полет без прекъсване или с прекъсване е използвана структура от данни-граф.Графът представлява взаимовръзка между отделни обекти, където всички обекти в графа се явяват върхове на графа, а всички връзки между отделните върхове(обекти) се наричат ребра на графа.В този случай графът е неориентиран, претеглен.За по-голяма яснота ще дефинираме термините неориентиран и претеглен.Граф е неориентиран, когато

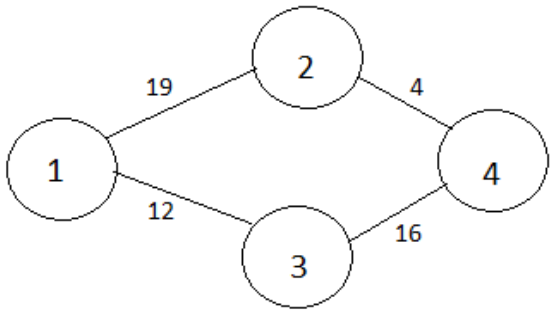
връзките между два обекта са двупосочни или например има възможност от обект „A” да се отиде към обект „B” и от обект „B” към обект „A“, докато при ориентиран граф посоката би била само една(само от „A“ към „B“ или само от „B“ към „A“).



Граф е претеглен когато връзката(реброто) между два обекта има тежест.Тежестта е реално число, което може например да представлява дължината на път между 2 града или пропусквателна способност на една тръба.

В конкретната задача върховете са всичките ни градове, ребрата са отделните полети между два града, а тежестта на ребрата са цените на конкретните полети.

Графът е представен с матрица на съседство.Матрицата се създава като по редовете и колоните се поставят номерата на върховете(в случаят- номерата на градовете), а при местата за връзка се записва теглото на съответното ребро.



Матрица на съседство на графът отляво

	1	2	3	4
1	0	19	12	0
2	19	0	0	4
3	12	0	0	16
4	0	4	16	0

Графът в тази задача е имплементиран в клас AdjacencyMatrix, който ще бъде разгледан по-късно.Тук ще обясня идеята на това как от всички полети се създава графът.Файлът в който се въвеждат всички полети Flights.txt, запълва контейнера от обекти от тип FlightInfo(това става с експлицитен конструктор на класа FlightData, който е разгледан по-късно в документацията).След като бъде запълнен контейнера се създава вектор от символни низове(string), в който се запълват всички имена на градове(излитащи и кацащи) от листа от тип FlightInfo.С този вектор с имената на градовете първоначално е в суров вид, тъй като са направо извлечени от полетите, то те могат да се повтарят и няма да са в азбучен ред.За това като са използвани функциите sort(), vector.erase() и unique() от STL библиотеката <algorithms>, стойностите във векторът стават сортирани по азбучен ред и са елиминирани повтарящите се градове(всеки град се среща точно един път).Сега вече този вектор от градове може да се използва за върховете на графа.Избран е контейнера вектор, защото елементите в него могат да се достъпват индексно, а **точно тези индекси се явяват като уникални номера на градовете.**

Използвани алгоритми

- ## Алгоритъм на търсене в дълбочина в граф (DFS)

Този алгоритъм позволява обхождане в граф като се тръгва от даден връх, който се обозначава като корен и последователно се посещават всички следващи върхове, докато се стигне до връх без наследници, след което се осъществява търсене в връщане назад (от англ. Backtracking) до достигане на крайна точка или при цялостно обхождане докато не стигне до корена от който е тръгнал.

В реализацията на текущата програма DFS е имплементиран с итеративна функция, а не рекурсивно.Използван е стек от STL библиотеката в който се съхраняват обходените върхове.

Алгоритъм:

1.Създава се масив от булев тип на който индекс отговаря съответният град.В този масив ще се следи кой град е обходен(true-обходен,false- не е обходен).Първоначално всички градове ще са необходени.

2.Тръгва се от корена, който се записва в стека и позицията му в масива се отбелязва като обходена.

3.Извлича се най-горният елемент от стека-from и се проверява в матрицата на ред from до всички останали върхове(от i=0 до броя на върховете-1) дали има връзка и дали вече е обходен текущият връх

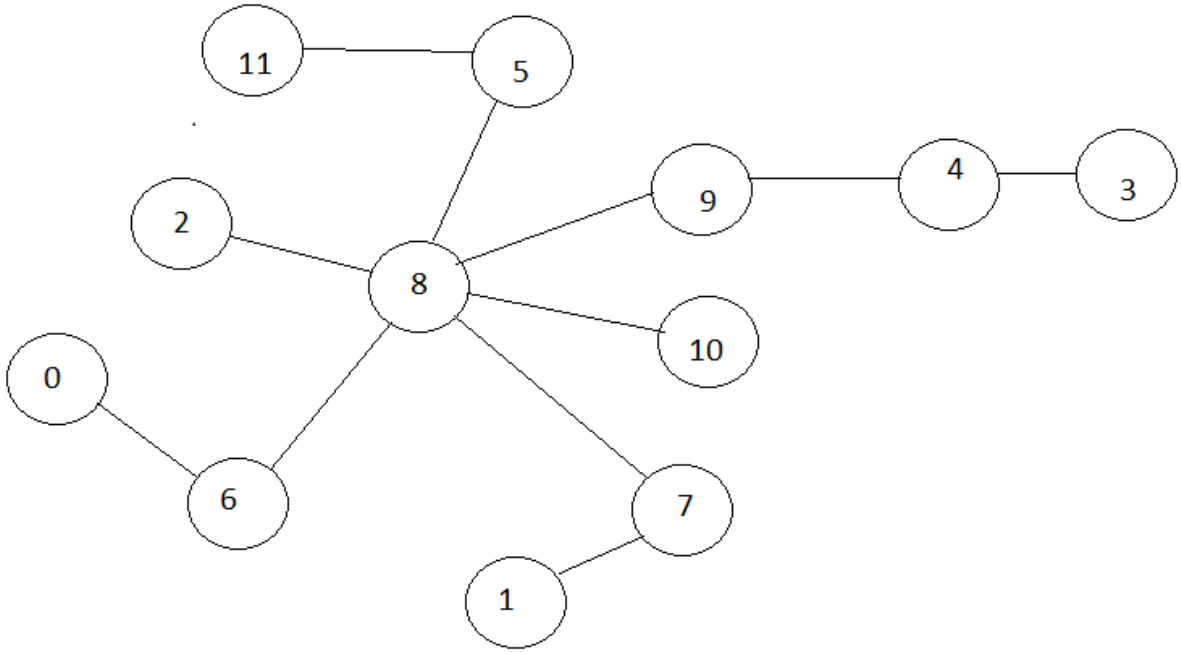
```
if (adjMatrix[from][i] >0 && !visited[i])
```

Ако условието е изпълнено върха i се записва в масива като обходен и се добавя в стека.

Когато се проверят всички връзки от един връх към всички останали се изпълнява отново стъпка 3

4.стъпка 3 се изпълнява докато текущият връх, който е излязъл от стека е и върха който търсим или докато всички върхове се обходят.

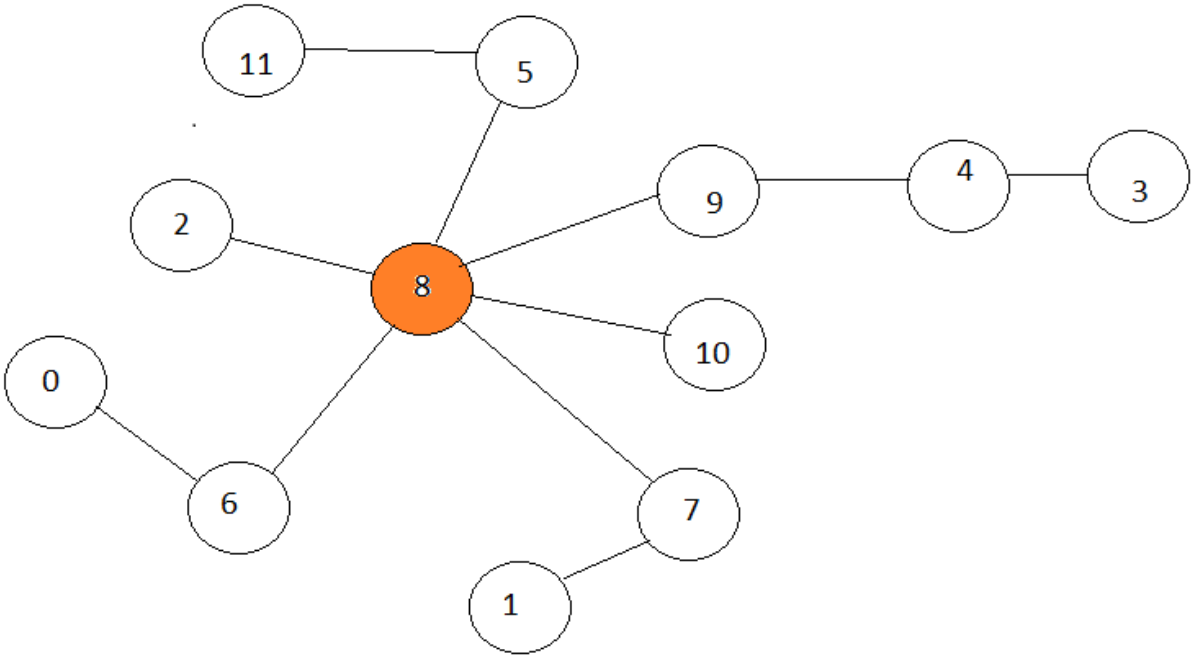
Пример, демонстриран и с графа от текущата програма:



Демонстрация на функцията за търсене в дълбочина от връх 8 до връх 3

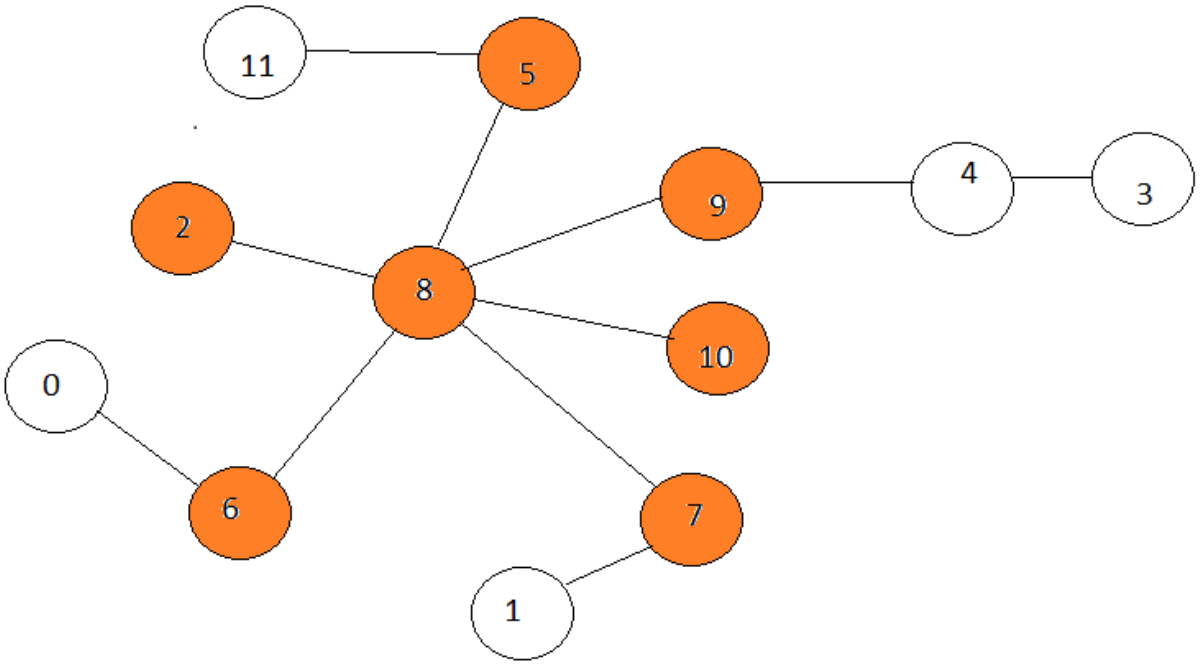
С оранжев цвят ще е обходеният връх, а със сиво в процес на обработка

Започвайки от връх 8 той ще е корена в случая.



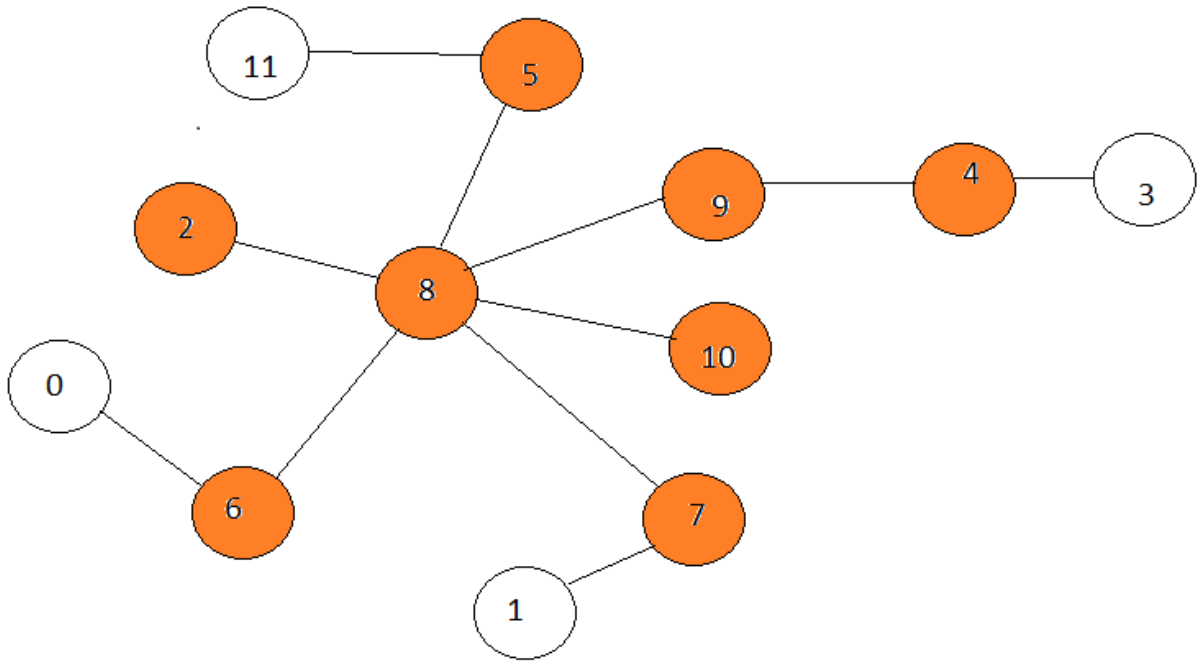
Състояние на стека:{8}

След като провери наследниците от текущият връх:



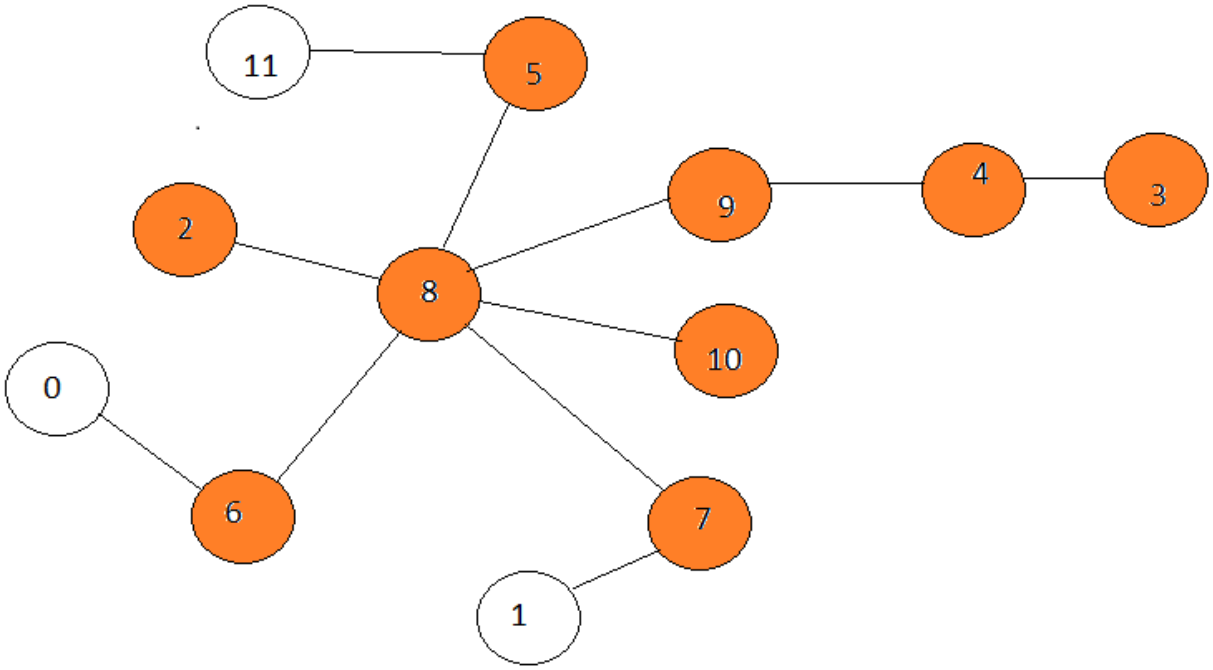
Състояние на стека:{2,5,6,7,9,10}

Тъй като последният добавен връх в стека е 10, но 10 единствено е свързан със върхът 8, който е вече обходен се извършва backtracking и от стека се извлича връх 9, който има необхододен наследник-връх 4.



Състояние на стека:{2,5,6,7,4}

Обхожда се връх 4 и се извлича от стека, от своя страна има необхододен наследник-връх 3,който се добавя в стека и се обхожда.



Състоянието на стека:{2,5,6,7,3}

Тъй като върхът 3, като се извлече от стека, съвпада с върхът който търсим, то функцията се прекратява и връща резултат, че е има път от корена до търсеният връх.

- Алгоритъм на Дейкстра-използван е за намиране на най-кратък път в граф от даден връх до всички останали върхове на граф с неотрицателни тегла на ребрата.Тъй като е подходящ за транспортни задачи като намиране на път в пътна карта или за намиране на път между два града за самолетни полети, какъвто е и случаят на текущата задача.

Стъпки на алгоритъма:

1.Създават се вектор, който съдържа „дистанцията“ от върха от който тръгваме, към всички останали върхове и вектор от булев тип,в който се записват срещу номера на всеки град дали същият е бил посетен или не.За индекс със стойност true е бил посетен съответният град, а за индекс със стойност false- не е бил посетен.

Инициализират се двата вектора, като този който пази стойностите на дистанцията към всички върхове се записват безкрайни стойности(в реализацията максималната стойност на integer), а във вектора с обходените върхове всички стойности са false(никой връх още не е обходен).Очевидно е, че размера на векторите е броят на върховете в графа.Декларира се още и променлива от тип map<key,value>, където ключът и стойността му са двойки върхове на обходеният от алгоритъма граф.Може да се представи като дърво от ребра(двойки върхове).От този map по-късно се изважда най-краткият път.

За улеснение вектора, който пази дистанцийте от корена до всички останали върхове ще се нарича-dist, вектора който пази обходените върхове-sptSet, а дървото съдържащо двойките върхове-parentMapl.

Важно! Градовете се достъпват по номер, като всеки индекс е отделен град!Върховете се явяват индекси!

2.Върхът, от който тръгва алгоритъма, във dist се присвоява стойност 0, защото логично дистанцията на връх към себе си е 0.Началният връх може да се разглежда като корен.

3.Във променлива, която сме нарекли ‘u’ се записва индекса/върха, който има най-малка дистанция и не е посетен.Този избор на индекс се извършва във функцията:

```
int  minDistance(vector<int> dist, vector<bool> sptSet)
```

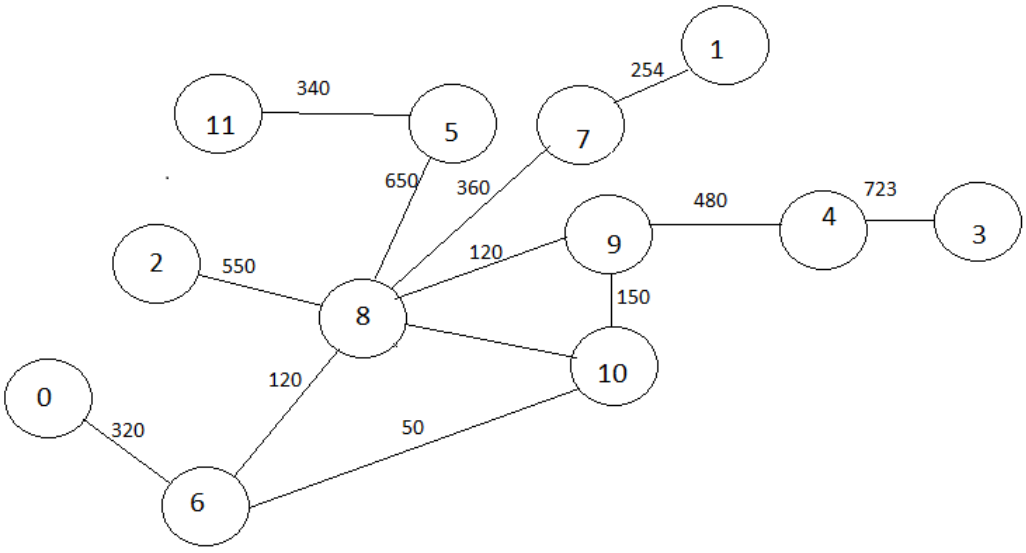
Във първата итерация този индекс винаги ще съответства на корена, защото разстоянието към всички останали върхове е безкрайно, а този на корена е 0.

4.Избраният връх се отбелязва като посетен.Подновява се стойността в dist на всички съседни върхове на ‘u’.За да се поднови дистанцията, се обхождат всички съседни върхове.Като за всеки съседен връх v, ако сумата от стойността на дистанцията от ‘u’ и теглото на връзката от графа u-v е по-малка от стойността на дистанцията при връх v, тогава се обновява.

5.Стъпка 3 и 4 се повтаря докато не се обходи всеки връх(във вектора sptSet всички върхове са true).Което означава, че дори при открит път към търсеният връх алогоритъмът не спира, докато и последният връх не бъде обходен.Това е така защото, не винаги първият срещнат път е оптимален/най-евтин.Съпоставка може да се направи и в реалният живот, където ако искаш да стигнеш от един град към друг, не винаги директният път е най-евтин, а по евтин може да се окаже през друго населено място.

За да се добие по-добра представа за алгоритъмът на Дейкстра ще бъде разгледан пример от програмата.В следващият пример върховете на графа с оранжев цвят ще са обходените градове а със бял, тези които не са обходени.

За даденият граф ще се покаже как работи имплементацията на алгоритъма от връх 0 до връх 4 .

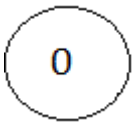


I. Първа стъпка.

Състояние на векторите:

	0	1	2	3	4	5	6	7	8	9	10	11
dist	0	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF	INF
sptSet	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

Състояние на графът:



Индкеса/върхът с минимална дистанция ‘u’ е връх 0, защото не е бил посещаван и има най-малка стойност от останалите стойности на разстоянието във вектора.Прави се проверка, като се обхождат върховете(,v‘) дали сумата от разстоянието от u със теглото на u-v е по-малко от разстоянието до v.

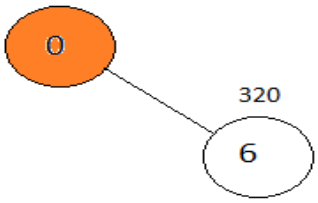
Намира се съседен на 0 връх, това е връх 6.Като преминава проверката dist[0]+ graph[0][6]< dist[6] или 0+320<INF.Добавя се в дървото двойката:parentMap={6,0} и се обновява разстоянието до връх 6 по следният начин-> dist[6]=dist[0]+ (теглото на реброто)graph[0][6].Така разстоянието до 6 dist[6]=0+320=320.

II. Втора стъпка

Състояние на векторите:

	0	1	2	3	4	5	6	7	8	9	10	11
dist	0	INF	INF	INF	INF	INF	320	INF	INF	INF	INF	INF
sptSet	**TRUE**	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

Състояние на графът:



Върхът с минимално разстояние ‘u’е 6, защото dist[6]=320<INF и не е посещаван.

Отбелязва се върхът 6 като посетен и отново се обхождат всички останали върхове.От обхождането се намират съседите на 6, които не са обходени и имат по малка сума на разстоянието.Първото преминаване на условието ще е за връх 8(dist[6]+ graph[6][8]<dist[8], което от таблицата може да се реши: 320+120<INF).Така в дървото ще се добави двойката {8,6} и ще добие вид parentMap={6,0},{8,6}, а обновеното разстояние до 8 ще е dist[8]=dist[6]+ graph[6][8], което е dist[8]=320+120=440.

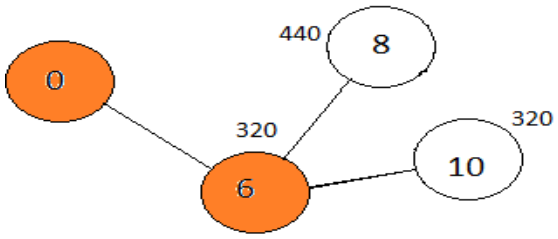
Второто преминаване на условието е при връх v=10(dist[6]+graph[6][10]<dist[10] или 320+ 50<INF).Дървото придобива вид parentMap={6,0},{8,6},{10,6} и обновеното разстояние до връх 10 е dist[10]=dist[6]+ graph[6][10] , dist[10]=320+50=370.

III. Трета стъпка.

Състояние на векторите:

	0	1	2	3	4	5	6	7	8	9	10	11
dist	0	INF	INF	INF	INF	INF	320	INF	440	INF	370	INF
sptSet	**TRUE**	FALSE	FALSE	FALSE	FALSE	FALSE	**TRUE**	FALSE	FALSE	FALSE	FALSE	FALSE

Състояние на графът:



Върхът с минимално разстояние, който не е бил обходен може да се види в графа, че е 10, тъй като 320<440 .

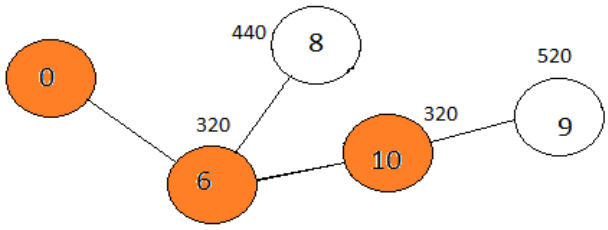
Върхът 10 се отбелязва като обходен.Първото преминаване на условието е при връх v=9(dist[10]+graph[10][9]<dist[9] или 370+150< INF).Дървото придобива следният вид-parentMap={6,0},{8,6},{9,10},{10,6} а разстоянието към връх 9 се обновява като dist[9]=dist[10]+graph[10][9], dist[9]=370+150=520.

IV. Четвърта стъпка.

Състояние на векторите:

	0	1	2	3	4	5	6	7	8	9	10	11
dist	0	INF	INF	INF	INF	INF	320	INF	440	520	370	INF
sptSet	**TRUE**	FALSE	FALSE	FALSE	FALSE	FALSE	**TRUE**	FALSE	FALSE	FALSE	**TRUE**	FALSE

Състояние на графът:



Избраният връх на минималното разстояние е 8 и го маркираме като обходен.

Първото преминаване на условието ще е при връх v=2(dist[8]+graph[8][2]<dist[2] -> 440+550<INF).В дървото се добавя връзката {2,8}->parentMap={2,8},{6,0},(8,6),{9,10},{10,6} и се обновява разстоянието до връх 2-> dist[2]=dist[8]+graph[8][2]=dist[2]=440+550=990 , така разстоянието от корена(0) до връх 2 е 990.

Второто преминаване на условието се случва при връх v=5(dist[8] +graph[8][5]<dist[5] ->440+650<INF).Дървото се обновява: parentMap={2,8},{5,8},{6,0},{8,6},{9,10},{10,6},

също и се обновява разстоянието до връх 5, dist[5]=dist[8]+graph[5][8]-> dist[5]=440+650=1090, разстоянието от корена-0 до връх 5 е 1090.

Трето преминаване на условието се случва при връх v=7.Дървото добавя двойка {7,8}:parentMap={2,8},{5,8},{6,0},{7,8},{8,6},{9,10},{10,6}, разстоянието до 7 се обновява като dist[7]=dist[8]+ graph[8][7] -> dist[7]=440+360=800.

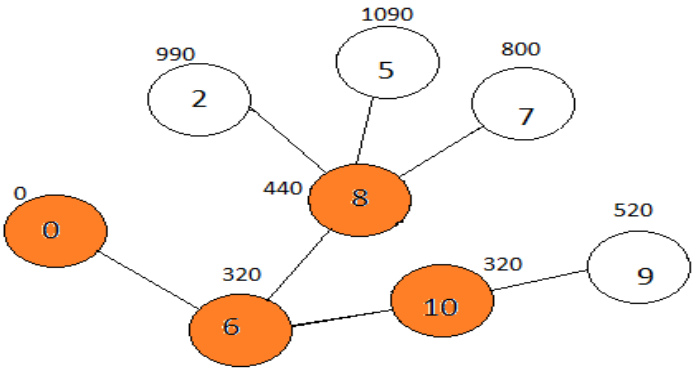
Забележка! Връх 8 е свързан с връх 9 във даденият граф,но тъй като разстоянието би станало по-голямо, то не преминава условието dist[8]+graph[8][9]<dist[9], което респективно е 440+120<520 ->560<520.Така и в дървото не се добавя връзка между 8 и 9!Дървото съдържа само двойките обходени от алгоритъма и съответно пътищата към всеки един връх.

V. Пета стъпка.

Състояние на векторите:

	0	1	2	3	4	5	6	7	8	9	10	11
dist	0	INF	990	INF	INF	1090	320	800	440	520	370	INF
sptSet	**TRUE**	FALSE	FALSE	FALSE	FALSE	FALSE	**TRUE**	FALSE	**TRUE**	FALSE	**TRUE**	FALSE

Състояние на графът:



Върхът с минимално разстояние е u=9, защото от всички които са със стойност false разстоянието до 9 е най-малко 520<990,1090,800,INF.Връх 9 се маркира като обходен.

Условието се изпълнява успешно при връх v=4 (dist[9]+graph[9][4]<dist[4] е 520+150<INF).В дървото се добавя двойката връх 4 към връх 9: parentMap={2,8},{4,9},},{5,8},{6,0},{7,8},{8,6},{9,10},{10,6}, а разстоянието към връх 4 от корена-0 е dist[4]=dist[9]+graph[9][4]=520+480=1000.

Всъщност може да се види, че и тук би могло да се прекъсне алгоритъма защото е открит път от корена 0 до връх 4, който се явява и най-кратък.Но тъй като принципа на алгоритъма е да открие всички върхове, понеже е възможно да се открие и по-

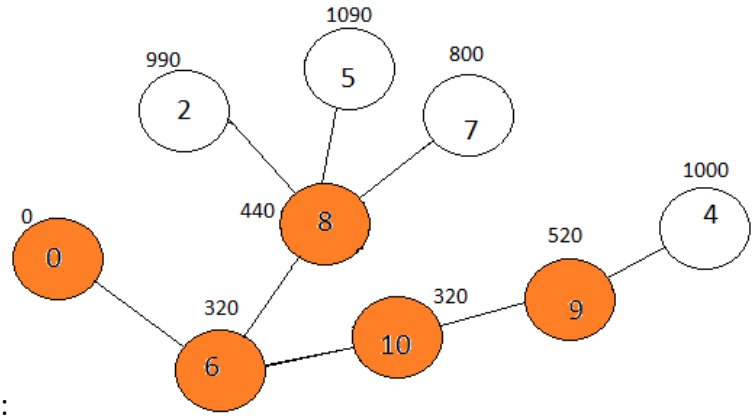
оптимален път, следващите стъпки ще довършат представата за това как работи алгоритъма.В този случай решението е открито.

VI. Шеста стъпка.

Състояние на векторите:

	0	1	2	3	4	5	6	7	8	9	10	11
dist	0	INF	990	INF	1000	1090	320	800	440	520	370	INF
sptSet	**TRUE**	FALSE	FALSE	FALSE	FALSE	FALSE	**TRUE**	FALSE	**TRUE**	**TRUE**	**TRUE**	FALSE

Състояние на графът:



Минималното разстояние до необходен връх е този на връх u=7.Отбелязва се 7 като обходен.

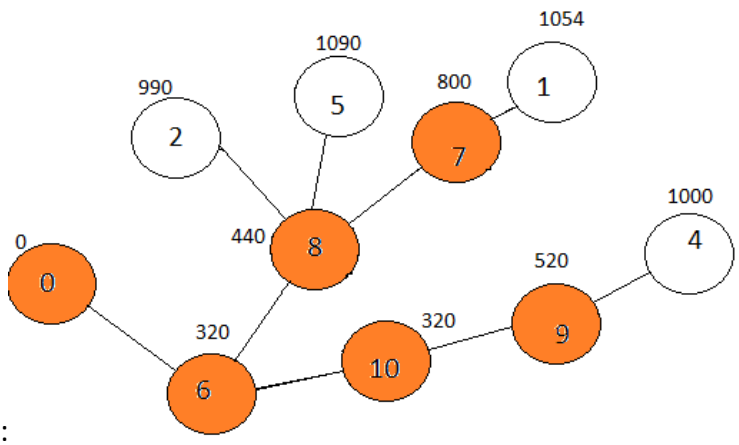
При връх v=1 условието е изпълнено,в дървото се добавя {1,7}: parentMap={1,7},{2,8},{4,9},},{5,8},{6,0},{7,8},{8,6},{9,10},{10,6}, обновява се разстоянието до връх 1 dist[1]=dist[7]+graph[7][1]=800+254=1054.

VII. Седма стъпка

Състояние на векторите:

	0	1	2	3	4	5	6	7	8	9	10	11
dist	0	1054	990	INF	1000	1090	320	800	440	520	370	INF
sptSet	**TRUE**	FALSE	FALSE	FALSE	FALSE	FALSE	**TRUE**	**TRUE**	**TRUE**	**TRUE**	**TRUE**	FALSE

Състояние на графът:



Минималното разстояние до необходен връх е този на връх u=2.Отбелязва се 2 като обходен.

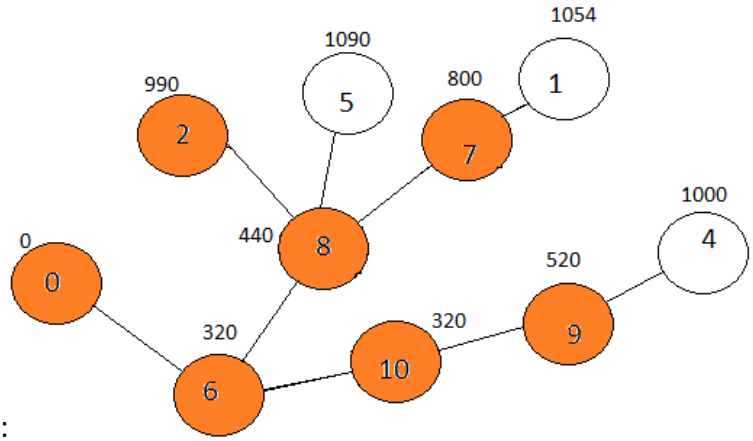
Връх 2 няма необходени съседи.

VIII. Осма стъпка.

Състояние на векторите:

	0	1	2	3	4	5	6	7	8	9	10	11
dist	0	1054	990	INF	1000	1090	320	800	440	520	370	INF
sptSet	**TRUE**	FALSE	**TRUE**	FALSE	FALSE	FALSE	**TRUE**	**TRUE**	**TRUE**	**TRUE**	**TRUE**	FALSE

Състояние на графът:



Минималното разстояние до необходен връх е този на връх u=4.Отбелязва се 4 като обходен.

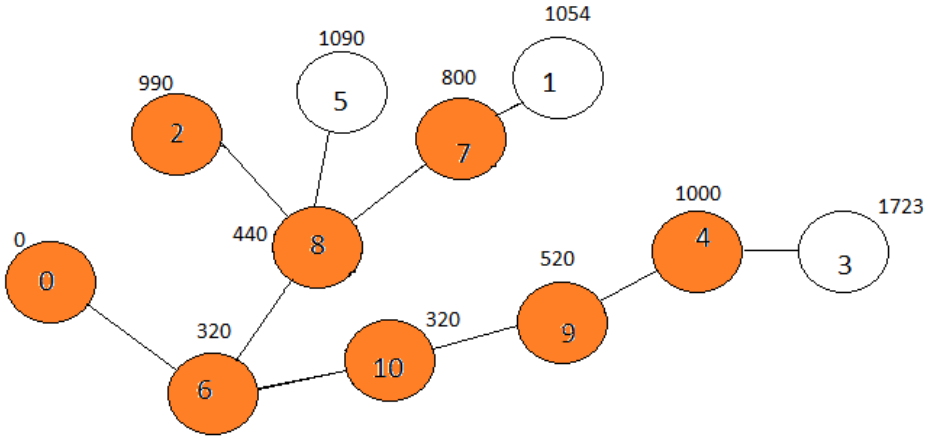
При връх v=3 условието е преминато успешно.В дървото се добавя връзка {3,4}: parentMap={1,7},{2,8},{3,4},{4,9},},{5,8},{6,0},{7,8},{8,6},{9,10},{10,6}, а разстоянието до връх 3 dist[3]=dist[4]+graph[4][3]=100+723=1723.

IX. Девета стъпка.

Състояние на векторите:

	0	1	2	3	4	5	6	7	8	9	10	11
dist	0	1054	990	1723	1000	1090	320	800	440	520	370	INF
sptSet	**TRUE**	FALSE	**TRUE**	FALSE	**TRUE**	FALSE	**TRUE**	**TRUE**	**TRUE**	**TRUE**	**TRUE**	FALSE

Състояние на графът:



Минималното разстояние до необходен връх е този на връх u=1.Отбелязва се връх 1 като обходен.

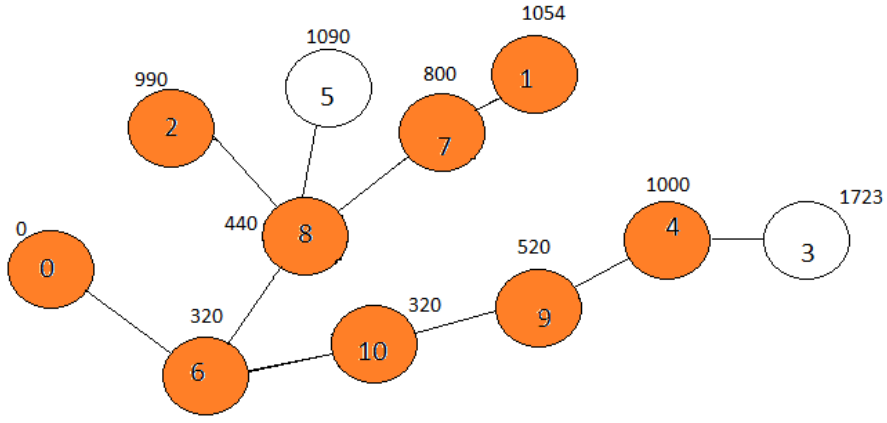
Връх 1 няма необходени съседи.

X. Десета стъпка.

Състояние на векторите:

	0	1	2	3	4	5	6	7	8	9	10	11
dist	0	1054	990	1723	1000	1090	320	800	440	520	370	INF
sptSet	**TRUE**	**TRUE**	**TRUE**	FALSE	**TRUE**	FALSE	**TRUE**	**TRUE**	**TRUE**	**TRUE**	**TRUE**	FALSE

Състояние на графът:



Минималното разстояние до необходен връх е този на връх u=5.Отбелязва се 5 като обходен.

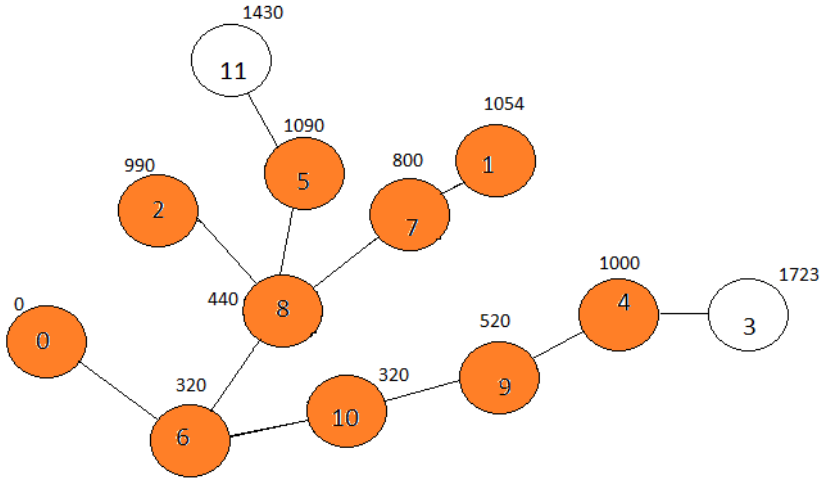
Необходен съсед на връх 5 е връх 11, така v=11 преминава условието.Добавя се връзката {11,5} в дървото: parentMap={1,7},{2,8},{3,4},{4,9},},{5,8},{6,0},{7,8},{8,6},{9,10},{10,6},{11,5}

XI. Единадесета стъпка.

Състояние на векторите:

	0	1	2	3	4	5	6	7	8	9	10	11
dist	0	1054	990	1723	1000	1090	320	800	440	520	370	1430
sptSet	**TRUE**	**TRUE**	**TRUE**	FALSE	**TRUE**	**TRUE**	**TRUE**	**TRUE**	**TRUE**	**TRUE**	**TRUE**	FALSE

Състояние на графът:



Минималното разстояние до необходен връх е този на връх u=11, защото 1430<1732.Отбелязва се 11 като обходен.

Това е и последна стъпка на алгоритъмът!

Описание на класовете и техните функции:

I. class AjacencyMatrix- този клас имплементира граф, представен като матрица на съседство.

Скрити членове :

`int **adjMatrix` -матрицата на съседство се инициализира в динамичната памет

`int nodes`-броят на върховете в графа

`bool *visited`-масив от булев тип, който съдържа дали даден връх е обходен или не, използва се в алгоритъма на Дейкстра и при търсене в дълбочина

`vector<int> listStops`-вектор от целочислен тип, съдържащ индексите на градовете, на най-краткият път в графа, използва се при алгоритъма на Дейкстра

Публични членове:

Експлицитен конструктор, който приема параматър, чиято стойност съответства на върховете на графа, с нея се инициализира матрицата на съседсвто и масива за обходените върхове:

```
AdjacencyMatrix(int init)
```

Метод за проверка дали съществува директна връзка между 2 върха подадени като параметри:

```
bool HasEdge(int source, int target)
```

Метод, с който се принтира матрицата на съседство, като параметри приема броят на върховете в графа и вектор от имената на върховете:

```
void PrintGraph(int counter, vector<string> nodes)
```

Метод за добавяне на връзка(ребро) между два върха.Като параметър се приема индекса/номера на върха(града) от който тръгва, индекса на върха(града) към който отива и цената или теглото на това ребро:

```
void AddEdge(int source, int dest, int cost)
```

Метод за търсене в дълбочина в графа.Използван се за да се намери дали има връзка между два града, които не са директно свързани.Като параметри са подадени индекса на върха от който тръгва, индекса на върха към който отива и общият брой върхове.Този алгоритъм ще бъде разгледан подробно в следващата глава.

```
int DepthFirstSearch(int from, int to, int size)
```

Метод за намиране на цената на път в граф, между два върха.Параметри са индекса на начален връх, индекса на крайният връх, вектора от имената на градовете.Този алгоритъм на Дейкстра ще бъде разгледан в следващата глава.

```
int Dijkstra(int src, int to, vector<string> cities)
```

В този метод е използван и външен за класа метод, който принтира на екрана градовете в най-краткият път намерен между два града.Като параметри приема индекса на града от който тръгва, индекса на града в който свършва, map който съдържа всеки връх “v”, върхът ”u”, който е открил “v”.Този map се запълва при изпълнението на Dijsktra.Последен параметър е вектор от имената на върховете в графа.В този метод се съставя вектор със индексите на градовете(съставящи най-краткият път) от map-а, който съдържа връзките между два върха.

```
void print(int source,int target,map<int,int> parentMap,vector<string> cities)
```

Метод за избиране на минималната дистанция между върховете, които не са посетени. Като параметър се подава вектор със стойностите на тежестите между града обозначен като корен и всички останали и вектор от булев тип отговарящ дали градовете са обходени или не.

```
int  AdjacencyMatrix::minDistance(vector<int> dist, vector<bool> sptSet)
```

Метод за намиране на най-краткият път в графа.Същият по същност като Dijkstra(), но вместо цялочисло(за разстоянието), връща най-краткият път, като вектор от индекси на върховете.

```
vector<int> ShortestPath(int src, int to, vector<string> cities)
```

Спомагащ метод на ShortestPath е метода.

```
vector<int> path(int source, int target, map<int, int> parentMap, vector<string>
cities)
```

В ShortestPath се запълва дървото от двойки, обходени от алгоритъма на Дейкстра, а в path() това дърво се предава като параметър, и в тази функция се запълва вектора, съдържащ индексите/върховете на най-краткият път.

II. Class FlightInfo със скрити частни членове: : Id на полета, тръгва от, пътува за, час на тръгване, час на пристигане са променливи от тип стринг, а цената от тип int.

Публични членове са:

Подразбиращ се конструктор:

```
FlightInfo()
 { flightID = ""; fliesTo = ""; fliesFrom = ""; timeAired="YYYY/MM/DD HH:MM";
timeArrived = "YYYY/MM/DD HH:MM:SS"; price = 0.0; };
```

Експлицитен конструктор:

```
FlightInfo(string flightid,string fromPort,string toPort,string timeaired,string
timearrived,int bgn)
```

Методи за запис на частните членове (Setters):

```
        void SetFlightID(string id);
        void SetFliesTo(string flight);
        void SetFliesFrom(string flight);
        void SetTimeAired(string time);
        void SetTimeArrived(string &time);
        void SetPrice(int bgn);
```

Методи за четена на частните членове(Getters):

```
        string GetFlightID();
        string GetFliesTo();
        string GetFliesFrom();
        string GetTimeAired();
        string GetTimeArrived();
        int GetPrice();
```

Оператори за вход и за изход:

```
        friend istream &operator>>(istream &fromStream, FlightInfo &obj);
        friend ostream &operator<<(ostream &toStream, list<FlightInfo> &obj);
```

В този клас при експлицитният конструктор и при методите за запис на полетата-час на тръгване и час на пристигане е използвана външна за класа функция за проверка дали датата е валидна

```
        bool CheckDate(string time)
```

Начинът по който проверява е като за всеки елемент от датата като година, месец, ден ,час и секунди, подаденият като параметър низ от символи се изрязва по индексите, които се очакват за отделните елементи, като се използва функцията

substr(index,lenght).Със новите изрязани стрингове се сравняват дали са легални,т.е. проверява се дали годината се състои от 4 символа, дали месеците са в област от „ 1“ до „12“, дали дните са от „1“ до „31“, дали часовете са в границата от „1“ до „24“ и дали секундите са от „0“ до „60“.Като така се гарантира формата на датата да е от вида „YYYY/MM/DD HH:MM:SS „

III. Клас FlightData частни членове: `list <FlightInfo> flights` –контейнер от клас FlightInfo.Публични членове:

Експлицитен конструктор, който приема параметър име на файл:

```
FlightData(string fileName)
```

Реализиран е като се създава файлов изходен поток с ifstream име на файла се явява подаденият параметър.Със функцията copy() предоставена от STL библиотеката <algorithm>, запълва контейнера с данни от задаеният файл.

Метод, който връща контейнера от клас FlightInfo:

```
list <FlightInfo> GetFlights();
```

Метод, който проверява дали има град(подаден като параметър) в контейнера:

```
bool HasCity(string city);
```

Итератор проверява дали има излитащ град или кацащ град в контейнера.При наличие на такъв град връща стойност true.Функцята се използва в главното меню за да ограничава потребителя да задава инвалидни градове.

Метод, който проверява дали има полет с ИД, зададено като параметър:

```
bool HasID(string id)
```

Аналогично на горната функция, итератор обхожда листа със полети и проверява дали има срещнат полет с такъв ID.

Метод за корекция на полет от зададен ID параметър:

```
void CorrectDataByID(string id)
```

По зададеният като параметър ID се интерира листа от полети и при срещнат полет с такъв ID се въвеждат от конзолата нови данни:пътува за, тръгва от, час на тръгване, час на кацане и цена, като данните заместват старите с извикване на функциите за запис на частните членове от клас FlightInfo.

Метод, който намира от кой град има най-много полети:

```
void MostFlightsFromCity()
```

Този метод изпълнява алгоритъм за най-често срещан елемент в масив.В този случай имаме масив от стрингове, в който записваме всеки град от който излита даден полет.

Чрез два вложени цикъла, първият сравнява елементите в масива от стрингове от i=0 до n-1, а вторият от i+1 до n.Като n е размера на масива от стрингове.При съвпадение на град, с индекс от първият, с град имащ индекс от вторият (вложен) цикъл се инкрементира променлива брояч.В първият цикъл(след вложеният) се проверява дали този брояч има най-много съвпадения(като сравнението става с променлива която е първоначално декларирана с минималната стойност на типа int, като ако стойността на брояча е по-голяма от тази на споменатата променлива, то тя приема стойността на брояча).Така сме сигурни че поредицата от съвпадения е наистина тази с максималният брой срещания на град.

Метод, който намира и извежда информация за полети тръгващи от зададен град, след зададен час, зададени като параметри:

```
void TakeOffFlightAfterTime(string time,string fromPort)
```

С интератор се обхожда листа от обекти и при съвпадение на град от който излита полет с този подаден като параметър се проверява дали часа на излитане е след часа зададен като параметър и след среща на такъв полет се извежда на конзолата чрез Get-функциите от класа FlightInfo.

Метод за извеждане информация за всички полети:

```
void PrintData()
```

Обхожда с итератор листа и печата всеки полет с Get-функциите.

Метод за добавяне на нови полети в контейнера:

```
	void AddFlight(string flightid, string fromPort, string toPort, string
timeaired, string timearrived, int bgn)
	void AddFlight(FlightInfo &obj)
```

Предефинирани функции, добавянето в контейнера се случва с функцията push_back(Object), като в първата функция се създава буферна променлива от тип FlightInfo, която се добавя в листа.

В следващите функции е използван графът описан в началото на документацията.

Метод който проверява дали има полет между два града, а ако има връща стойноста на неговата цена, ако няма връща -1:

```
int HasFlight(int fromPort, int toPort, vector<string> nodes)
```

Изпълнено е с цикъл, който обхожда с итератор контейнера от обекти от клас FlightInfo.Като параметри са подадени индексите на града от който излита полета, индекса на града в който каца полета, а вторият параметър е вектора от върховете на графа(всички градове), който е описан накрая на глава „**Въведение в графите и начин на имплементация**“.Прави се проверка дали има полет\връзка (за да се постави ребро в графа) между двата града, като се сравяват полета на индекс fromPort(nodes[fromPort]) и полета на индекс[toPort](nodes[toPort]) със градовете от полетите в листа от полети. Главната цел на този метод е при инциализирането на нашият граф.Именно този метод запълва матрицата на съседство като се извиква n*n брой пъти, за всяка една вариация на полет.n- броят на градовете в вектора от върхове.

Метод,който връща стойност int: -2 за полет, който излита от и пристига в един и същи град, -1 за полет който не съществува, 0 за директен полет и стойност 1 за полет с прикачвания.

```
int FindFlight(string fromPort, string toPort, vector<string> &cities, AdjacencyMatrix
                                                                              &matrix)
```

Приема параметри: стринг-град на излитане, стринг- град на пристигане, вектор от стрингове-върховете на графа, AdjacencyMatrix- самият граф.С градовете пристигане и тръгване, намираме съответните им индекси във вектора от върхове.Тези индекси са номерата и на самите градове в матрицата на съседство.Правим проверка дали има директен полет между 2та града като използваме функцията HasEdge() от AdjacencyMatrix, като ако има (true) връщаме 0, но ако няма директна връзка извикваме функцията DepthFirstSearch(), която обхожда графа в дълбочина и връща стойност -2,-1 или 1.

Метод, който връща цената на най-прекият път в графа.

```
int FindFlightPrice(string fromPort, string toPort, vector<string> &cities,
AdjacencyMatrix& matrix)
```

Приема параметри: стринг-град на излитане, стринг- град на пристигане, вектор от стрингове-върховете на графа, AdjacencyMatrix- самият граф.С градовете пристигане и тръгване, намираме съответните им индекси във вектора от върхове.Тези индекси са номерата и на самите градове в матрицата на съседство.Връща се с return резултатът от функцията Dijkstra() от AdjacencyMatrix.Алгоритъмът Дейкстра е обяснен при класът AdjacencyMatrix.

Метод, който проверява дали зададен като параметър полет удовлетворява изискванията на реален полет от листа с полети.

```
bool SatisfiesTime(string timeTakeoff,string fromPort,string timeLands,string
                   toPort,AdjacencyMatrix &matrix,vector<string> &cities)
```

Параметрите са както следва: стринг-датата и часа в който излита полета, стринг-града от който излита, стринг-дата и час на пристигане, града в който пристига, AdjacencyMatrix- графът, векторът от стрингове- върховете на графа. С градовете пристигане и тръгване, намираме съответните им индекси във вектора от върхове.Използван е същият алгоритъм на Дейкстра чрез функцията ShortestPath(), тази функция връща вектор от цели числа, който се явяват най-краткият път от един град до друг.Този вектор съдържа индексите на градовете и всеки град с този индекс има директна връзка със следващият индекс във вектора, така всеки индекс образува път, индексите на който са съседни със всеки следващ.Чрез цикъл, итератор обхожда листа от полети и се проверява дали съвпада с полет от вектора съдържащ най-краткият път.

```
if ((it.GetFliesFrom() == cities[path[idx]] && it.GetFliesTo() == cities[path[idx + 1]]))
```

Ако условието бъде изпълнено се преминава към следващото условие-дали полета между тези два града удовлетворява и зададените като параметри дата и час.

```
if (timeTakeoff<=it.GetTimeAired() && it.GetTimeArrived()<=timeLands)
```

Ако това условие е изпълнено се отпечатва на конзолата този полет, който отговаря на горните условия флага за удовлетворено време, който е от булев тип приема true и индекса с който достъпваме вектора с индексите с най-краткият път се инкрементира, за да достъпи следващият полет.Ако това условие не е е изпълнено, то флага за удовлетворено време приема false.Така се обхожда всеки полет и се проверява дали има полети отговарящи на най-краткият път.Фунцкията свършва с връщането на флага за удовлетворено време.

Предефиниране на оператор изходен поток, за лист от тип FlightData

```
friend ostream &operator<<(ostream &toStream, list<FlightData> &obj)
```

Метод, който връща лист от FlightData, съдържащ само полетите от град подаден като параметър

```
list<FlightInfo> GetFlightsFrom(string fromPort)
```

Метод, който връща лист от FlightData, съдържащ само полетите към град подаден като парамтър

```
list<FlightInfo> GetFlightsTo(string toPort)
```

IV. клас UserRequest скрити частни членове:

стрингове за: Id на искането, пътува от, пътува за, тръгване (най-рано), пристигане (най-късно)

публични полета:

Експлицитен конструктор:

```
UserRequest(string id,string fromPort,string toPort,string timeaired,string timearrived)
```

Методи за присвояване на стойност на член променливите(Setters):

```
void SetRequesttID(string id)
void SetFliesTo(string flight)
void SetFliesFrom(string flight)
void SetTimeAired(string time)
void SetTimeArrived(string &time)
```

Методи за четене на член променливите (Getters):

```
string GetFlightID();
string GetFliesTo();
string GetFliesFrom();
string GetTimeAired();
string GetTimeArrived();
```

Предефиниране на оператор за вход от поток на обект от тип UserRequest

```
friend istream &operator>>(istream &fromStream,UserRequest &obj)
```

Передефиниране на оператор за изход към поток, на лист от обекти от тип UserRequest:

```
friend ostream &operator<<(ostream &toStream, list<UserRequest> &obj)
```

V. Клас Users

Скрити член променливи : лист от UserRequest

Публични полета:

Експлицитен конструктор с параметър име на файл, запълва листа от UserRequest със съдържанието на файла:

```
Users(string fileName)
```

Метод за добавяне не ново потребителско желание, приемащ като параметър: id на полет, пътува от , пътува за, дата на тръгване, дата на пристигане:

```
void AddUserRequest(string flightid, string fromPort, string toPort, string
                                                    timeaired, string timearrived)
```

Метод за намиране на град, който има най-много желаещи да пътуват пътници

```
void MostRequestsToCity()
```

Този метод изпълнява алгоритъм за най-често срещан елемент в масив.В този случай имаме масив от стрингове, в който записваме всеки град от който излита даден полет.

Чрез два вложени цикъла, първият сравнява елементите в масива от стрингове от i=0 до n-1, а вторият от i+1 до n.Като n е размера на масива от стрингове.При съвпадение на град, с индекс от първият, с град имащ индекс от вторият (вложен) цикъл се инкрементира променлива брояч.В първият цикъл(след вложеният) се проверява дали

този брояч има най-много съвпадения(като сравнението става с променлива която е първоначално декларирана с минималната стойност на типа int, като ако стойността на брояча е по-голяма от тази на споменатата променлива, то тя приема стойността на брояча).Така сме сигурни че поредицата от съвпадения е наистина тази с максималният брой срещания на град.

Метод за принтиране на данните от контейнера от UserRequest, който съдържа желаещите потребителски полети

```
void PrintData()
```

Метод, който проверява дали при дадено като параметър ID, съвпада с ID от исканите от потребителя полети.Така сме сигурни, че при въвеждане на желан полет ID ще е уникално.

```
bool HasID(string id)
```

Метод, който намира най-евиният полет по зададено ID, като параметри приема съответно id и контейнер-свързан списък от FlightData(този който създаваме менюто), и матрицата(графът).

```
void CheapestFlightById(string id, list<FlightData> flights ,AdjacencyMatrix &matrix)
```

Извлича се желаният полет със подаденото id, от скритото поле(контейнера от UserRequest).Прави се проверка дали този желан полет има такива градове в графът,дали има път между двата града и дали удовлетворяват реални полети.Проверките стават със функциите от класа FlightData – HasCity(),FindFlight(),SatisfiesTime().Ако е намерен път между два града и желаният полет удовлетворява реален полен(функциите FindFlight и SatisfiesTime са изпълнени) се извиква фунцкията FindFlightPrice, която връща цената на най-краткият път.

Метод, който за всички потребителски заявки, намира и извежда най-евтиният полет без прекъсване или с прекъсване, ако няма друг, отговарящ на искането на потребителя, или информация, че няма такава възможност.

```
void CheapestFlightForAllReq( list<FlightData> flights,AdjacencyMatrix &matrix);
```

Всяко потребителско желание се обхожда и се извиква функцията CheapestFlightById() за него.

Оператор за извеждане в поток.

```
friend ostream &operator<<(ostream &toStream, list<Users> &obj)
```

Сорс код

Хедърен файл Adjacencymatrix.h

```
#pragma once
#ifndef ADJACENCY_MATRIX_H
#define ADJACENCY_MATRIX_H
#include <string>
#include <list>
#include <iterator>
#include <vector>
#include <iostream>
#include <queue>
```

```
#include <map>
#include <stack>
#include "FlightInfo.h"
class AdjacencyMatrix
{
public:
	AdjacencyMatrix() {};
	~AdjacencyMatrix()
	{
		for (int i = 0; i < nodes; i++)
		{
			delete []adjMatrix[i];

		}
		delete []visited;
		delete []adjMatrix;
	};
	AdjacencyMatrix(int init);
	bool HasEdge(int source, int target);
	void PrintGraph(int counter);
	void PrintGraph(int counter, vector<string> nodes);
	void AddEdge(int source, int dest, int cost);
	int DepthFirstSearch(int from, int to, int size);
	int minDistance(vector<int> dist, vector<bool> sptSet);
	int Dijkstra(int src, int to, vector<string> cities);
	vector<int> ShortestPath(int src, int to, vector<string> cities);
	vector<int> path(int source, int target, map<int, int> parentMap, vector<string>
cities);
private:
	vector<int> listStops;
	int nodes;
	int **adjMatrix;
	bool *visited;
};


#endif
```

Сорс файл AdjacencyMatrix.cpp

```
#include "AdjacencyMatrix.h"

void print(int source, int target, map<int, int> parentMap, vector<string> cities);

AdjacencyMatrix::AdjacencyMatrix(int init)
{
	nodes = init;
	visited = new bool[init];
	for (int i = 0; i < init; i++)
	{
		visited[i] = false;
	}
	adjMatrix = new int*[init];
	for (int i = 0; i < init; i++)
	{

		adjMatrix[i] = new int[init];

	}
	for (int i = 0; i < init; i++)
	{
```

```
                for (int j = 0; j < init; j++)
                {
                        adjMatrix[i][j] = 0;
                }
        }
}
bool AdjacencyMatrix::HasEdge(int source, int target)
{
        return adjMatrix[source][target] > 0;
}
void AdjacencyMatrix::PrintGraph(int counter)
{
        for (int i = 0; i < counter; i++)
        {
                for (int j = 0; j < counter; j++)
                {
                        std::cout << adjMatrix[i][j] << "\t";
                }
                std::cout << std::endl;
        }
};
void AdjacencyMatrix::PrintGraph(int counter, vector<string> nodes)
{
        cout << "\t";
        for each (auto var in nodes)
        {
                cout << var << "\t";
        }
        cout << endl;
        for (int i = 0; i < counter; i++)
        {
                cout << nodes[i] << "\t";
                for (int j = 0; j < counter; j++)
                {
                        std::cout << adjMatrix[i][j] << "\t";
                }
                std::cout << std::endl;
        }
}
void AdjacencyMatrix::AddEdge(int source, int dest, int cost)
{
        adjMatrix[source][dest] = cost;
}
int AdjacencyMatrix::DepthFirstSearch(int from, int to, int size)
{

        for (int i = 0; i < size; i++)
        {
                visited[i] = false;
        }
        stack<int> Stack;
        Stack.push(from);
        visited[from] = true;
        int c = 0;
        if (from == to)
        {
                return -2;
        }
        //cout << "Depth first Search starting from vertex ";
        //cout << from << " : " << endl;
        while (!Stack.empty())
        {
```

```
                from = Stack.top();
                //cout << from << " ";
                Stack.pop();
                if (from == to && c != 1)
                {
                        //cout << "Airport found" << endl;
                        return 1;
                }
                if (from == to && c == 1)
                {
                        //cout << "Airport found" << endl;
                        return 0;
                }
                for (int i = 0; i < size; i++)
                {
                        if (adjMatrix[from][i] >0 && !visited[i])
                        {
                                visited[i] = true;
                                Stack.push(i);
                        }
                }
                c++;
        }
        cout << endl;
        return -1;

};

int  AdjacencyMatrix::minDistance(vector<int> dist, vector<bool> sptSet)
{
        // Initialize min value
        int min = INT_MAX, min_index;

        for (int v = 0; v <nodes; v++)
                if (sptSet[v] == false && dist[v] <= min)
                        min = dist[v], min_index = v;

        return min_index;
}


int  AdjacencyMatrix::Dijkstra(int src, int to, vector<string> cities)
{
        map<int, int> parentMap;
        vector<int> dist(nodes); // The output array.  dist[i] will hold the shortest
                                                        // distance from src to i

        vector<bool> sptSet(nodes); // sptSet[i] will true if vertex i is included in
shortest
                                                                // path tree or shortest
distance from src to i is finalized

                                                                // Initialize all distances as
INFINITE and stpSet[] as false
        for (int i = 0; i < nodes; i++)
                dist[i] = INT_MAX, sptSet[i] = false;

        // Distance of source vertex from itself is always 0
        dist[src] = 0;

        // Find shortest path for all vertices
        for (int count = 0; count < nodes - 1; count++)
```

```
	{
		// Pick the minimum distance vertex from the set of vertices not
		// yet processed. u is always equal to src in first iteration.
		int u = minDistance(dist, sptSet);


		// Mark the picked vertex as processed
		sptSet[u] = true;

		// Update dist value of the adjacent vertices of the picked vertex.
		for (int v = 0; v <nodes; v++)

			// Update dist[v] only if is not in sptSet, there is an edge from
			// u to v, and total weight of path from src to  v through u is
			// smaller than current value of dist[v]
			if (!sptSet[v] && adjMatrix[u][v] && dist[u] != INT_MAX && dist[u]
+ adjMatrix[u][v] < dist[v])
			{
				parentMap[v] = u;
				dist[v] = dist[u] + adjMatrix[u][v];
				//cout << cities[v] << "-";
			}
	}

	print(src, to, parentMap,cities);

	// print the constructed distance array
	return dist[to];
}
void print(int source,int target,map<int,int> parentMap,vector<string> cities)
{
	vector<int> citiesVec;
	int current=target;
	while (current!=source)
	{

		citiesVec.push_back(current);
		current=parentMap[current];
	}
	citiesVec.push_back(current);
	cout << "\n\tCity stopovers:";
	for (int i =citiesVec.size()-1; i >=0;i--)
	{
		cout << cities[citiesVec[i]] << " ";
	}

	cout << endl;
}
vector<int> AdjacencyMatrix::path(int source, int target, map<int, int> parentMap,
vector<string> cities)
{
	vector<int> citiesVec;
	vector<int> citiesInOrder;
	int current = target;
	while (current != source)
	{

		citiesVec.push_back(current);
		current = parentMap[current];
	}
	citiesVec.push_back(current);
	for (int i = citiesVec.size() - 1; i >= 0;i--)
```

```
	{
		citiesInOrder.push_back(citiesVec[i]);
	}
	return citiesInOrder;
}
vector<int> AdjacencyMatrix::ShortestPath(int src, int to, vector<string> cities)
{
	map<int, int> parentMap;
	vector<int> dist(nodes); // The output array.  dist[i] will hold the shortest
                                                 // distance from src to i

	vector<bool> sptSet(nodes); // sptSet[i] will true if vertex i is included in
shortest
                                                         // path tree or shortest
distance from src to i is finalized

                                                         // Initialize all distances as
INFINITE and stpSet[] as false
	for (int i = 0; i < nodes; i++)
		dist[i] = INT_MAX, sptSet[i] = false;

	// Distance of source vertex from itself is always 0
	dist[src] = 0;

	// Find shortest path for all vertices
	for (int count = 0; count < nodes - 1; count++)
	{
		// Pick the minimum distance vertex from the set of vertices not
		// yet processed. u is always equal to src in first iteration.
		int u = minDistance(dist, sptSet);


		// Mark the picked vertex as processed
		sptSet[u] = true;

		// Update dist value of the adjacent vertices of the picked vertex.
		for (int v = 0; v <nodes; v++)

			// Update dist[v] only if is not in sptSet, there is an edge from
			// u to v, and total weight of path from src to  v through u is
			// smaller than current value of dist[v]
			if (!sptSet[v] && adjMatrix[u][v] && dist[u] != INT_MAX && dist[u]
+ adjMatrix[u][v] < dist[v])
			{
				parentMap[v] = u;
				dist[v] = dist[u] + adjMatrix[u][v];
				//cout << cities[v] << "-";
			}
	}

	return path(src, to, parentMap, cities);
}
```

Хедърен файл FlightInfo.h

```
#pragma once
#ifndef FLIGHTINFO_H_
#define FLIGHTINFO_H_
//Define class
#include <string>
#include <fstream>
#include <list>
using namespace std;
```

```
//A class to define flight information
class FlightInfo
{
public:
	FlightInfo() { flightID = ""; fliesTo = ""; fliesFrom = "";
timeAired="YYYY/MM/DD HH:MM"; timeArrived = "YYYY/MM/DD HH:MM:SS"; price = 0.0; };
	FlightInfo(string flightid,string fromPort,string toPort,string timeaired,string
timearrived,int bgn);
	~FlightInfo() {};
	void SetFlightID(string id);
	void SetFliesTo(string flight);
	void SetFliesFrom(string flight);
	void SetTimeAired(string time);
	void SetTimeArrived(string &time);
	void SetPrice(int bgn);
	string GetFlightID();
	string GetFliesTo();
	string GetFliesFrom();
	string GetTimeAired();
	string GetTimeArrived();
	int GetPrice();
	friend istream &operator>>(istream &fromStream, FlightInfo &obj);
	friend ostream &operator<<(ostream &toStream, list<FlightInfo> &obj);
private:
	string flightID;
	string fliesTo;
	string fliesFrom;
	string timeAired;
	string timeArrived;
	int price;

};


#endif
```

Сорс файл FlightInfo.cpp

```
#include "FlightInfo.h"

//implement funcion
#define FormatException -1

bool CheckDate(string time)
{
	if ((time.substr(0, 4).size() == 4) &&
	// Check YYYY
		(time.substr(5, 2) > "00"&&time.substr(5, 2) < "13") &&
	//Check MM
		(time.substr(8, 2) > "00"&&time.substr(8, 2) <= "31") &&	//Check DD
		(time.substr(11, 2) > "0"&&time.substr(11, 2) <= "24") &&	//Check HH
		(time.substr(14, 2) >= "00"&&time.substr(14, 2) <= "60") &&	//Check
Minutes
		(time.size() == 16))
	{
		return true;
	}
	else
	{
		return false;
	}
}
istream &operator>>(istream &fromStream, FlightInfo &obj)
```

```
{
	fromStream >> obj.flightID >> obj.fliesFrom >> obj.fliesTo >> obj.timeAired >>
obj.timeArrived >> obj.price;
	obj.SetTimeAired(obj.timeAired);
	obj.SetTimeArrived(obj.timeArrived);
	return fromStream;
}

FlightInfo::FlightInfo(string flightid, string fromPort, string toPort, string
timeaired, string timearrived, int bgn)
{
	flightID = flightid;
	fliesFrom = fromPort;
	fliesTo = toPort;
	if (CheckDate(timeaired))

	{
		timeAired = timeaired;
	}
	else
	{
		throw FormatException;
	}
	if (CheckDate(timearrived))
	{
		timeArrived = timearrived;
	}
	else
	{
		throw FormatException;
	}

	price = bgn;
}
void FlightInfo::SetFlightID(string id)
{
	flightID = id;
}
void FlightInfo::SetFliesTo(string flight)
{
	fliesTo = flight;
}
void FlightInfo::SetFliesFrom(string flight)
{
	fliesFrom = flight;
}
void FlightInfo::SetTimeAired(string time)
{
	if (CheckDate(time))
	{
		timeAired = time;
	}
	else
	{
		throw;
	}
}
void FlightInfo::SetTimeArrived(string &time)
{
	if (CheckDate(time))
	{
		timeArrived = time;
```

```
	}
	else
	{
		throw;
	}
}
void FlightInfo::SetPrice(int bgn)
{
	price = bgn;
}
string FlightInfo::GetFlightID()
{
	return flightID;
}
string FlightInfo::GetFliesTo()
{
	return fliesTo;
}
string FlightInfo::GetFliesFrom()
{
	return fliesFrom;
}
string FlightInfo::GetTimeAired()
{
	return timeAired;
}
string FlightInfo::GetTimeArrived()
{
	return this->timeArrived;
}
int FlightInfo::GetPrice()
{
	return price;
}
ostream &operator<<(ostream &toStream, list<FlightInfo> &obj)
{
	list<FlightInfo>::iterator it;
	for (it = obj.begin(); it != obj.end(); it++)
	{
		toStream<<it->GetFlightID() << " " << it->GetFliesFrom() << " " << it-
>GetFliesTo() << " " << it->GetTimeAired() << " " << it->GetTimeArrived() << " " <<
it->GetPrice() << endl;
	}

	return toStream;
}
```

Хедърен файл FlightData.h

```
#pragma once
#ifndef FLIGHTDATA_H_
#define FLIGHTDATA_H_
#include <list>
#include <string>
#include "FlightInfo.h"
#include <iterator>
#include <fstream>
#include <iostream>
#include <algorithm>
#include <vector>
#include <stack>
#include <map>
#include "Graph.h"
```

```
#include "AdjacencyMatrix.h"
using namespace std;
class FlightData
{
public:
	FlightData() {};
	~FlightData() {};
	FlightData(string fileName) ;//
	list<FlightInfo> GetFlightsFrom(string fromPort);//
	list<FlightInfo> GetFlightsTo(string toPort);//
	list <FlightInfo> GetFlights();//
	int HasFlight(int fromPort, int toPort, vector<string> nodes);//
	int FindFlight(string fromPort, string toPort, vector<string> &cities,
AdjacencyMatrix &matrix);//
	int FindFlightPrice(string fromPort, string toPort, vector<string> &cities,
AdjacencyMatrix& matrix);//
	void AddFlight(string flightid, string fromPort, string toPort, string
timeaired, string timearrived, int bgn);//
	void AddFlight(FlightInfo &obj);//
	void PrintData();//
	bool HasCity(string city);//
	bool HasID(string id);//
	bool SatisfiesTime(string timeTakeoff, string fromPort, string timeLands, string
toPort,AdjacencyMatrix &matrix,vector<string> &cities);//
	void CorrectDataByID(string id);//
	void MostFlightsFromCity();//
	void TakeOffFlightAfterTime(string time,string fromPort);//
	friend ostream &operator<<(ostream &toStream, list<FlightData> &obj);//

private:
	list <FlightInfo> flights;
};



#endif
```

Cорс файл FlightData.cpp

```
#include "FlightData.h"


FlightData::FlightData(string fileName)
{
	ifstream file(fileName.data());
	if (file.good())
		copy(istream_iterator<FlightInfo>(file), istream_iterator<FlightInfo>(),
back_inserter(flights));
	else throw ;
}
void FlightData::PrintData()
{
	list<FlightInfo>::iterator it;
	for (it = flights.begin(); it != flights.end();it++)
	{
		cout << it->GetFlightID() << " "
			<< it->GetFliesFrom() << " "
			<< it->GetFliesTo() << " "
			<< it->GetTimeAired() << " "
			<< it->GetTimeArrived() << " "
			<< it->GetPrice() << endl;
```

```
	}
}

list<FlightInfo> FlightData::GetFlightsFrom(string fromPort)
{
	list<FlightInfo> flightsFromPort;
	for each (auto var in flights)
	{
		if (var.GetFliesFrom()==fromPort)
		{
			FlightInfo
st(var.GetFlightID(),var.GetFliesFrom(),var.GetFliesTo(),var.GetTimeAired(),var.GetTim
eArrived(),var.GetPrice());
			flightsFromPort.push_back(st);
		}
	}
	return flightsFromPort;
}
list<FlightInfo>  FlightData::GetFlightsTo(string toPort)
{
	list<FlightInfo> flightsToPort;
	list<FlightInfo>::iterator it;
	for (it = flights.begin(); it != flights.end(); it++)
	{
		if (it->GetFliesTo() == toPort)
		{
			FlightInfo st(it->GetFlightID(), it->GetFliesFrom(), it-
>GetFliesTo(), it->GetTimeAired(), it->GetTimeArrived(), it->GetPrice());
			flightsToPort.push_back(st);
		}
	}
	return flightsToPort;
}
void  FlightData::AddFlight(string flightid, string fromPort, string toPort, string
timeaired, string timearrived, int bgn)
{

	FlightInfo info;
	info.SetFlightID(flightid);
	info.SetFliesFrom(fromPort);
	info.SetFliesTo(toPort);
	info.SetTimeAired(timeaired);
	info.SetTimeArrived(timearrived);
	info.SetPrice(bgn);
	flights.push_back(info);


}
void FlightData::CorrectDataByID(string id)
{
	list<FlightInfo>::iterator it;
	for (it=flights.begin(); it!=flights.end(); it++)
	{
		if (id==it->GetFlightID())
		{
			string inputInfo;
			int currency;

			cout << "\nEnter taking off city: ";
			cin >> inputInfo;
			it->SetFliesFrom(inputInfo);
```

```
                    cout << "\nEnter arrival city: ";
                    cin >> inputInfo;
                    it->SetFliesTo(inputInfo);

                    cout << "\nEnter aired date and time {ONLY IN FORMAT YYYY/MM/DD-
HH:MM}: ";
                    cin >> inputInfo;
                    it->SetTimeAired(inputInfo);
                    cout << "\nEnter arrival date and time {ONLY IN FORMAT YYYY/MM/DD-
HH:MM}: ";
                    cin >> inputInfo;
                    it->SetTimeArrived(inputInfo);

                    cout << "\nEnter price of the flight: ";
                    cin >> currency;
                    it->SetPrice(currency);
            }
        }
}

void FlightData::MostFlightsFromCity()
{
        string *stringOfCities;
        stringOfCities = new string[flights.size()];
        list<FlightInfo>::iterator it;
        int i = 0;
        for (it = flights.begin(); it != flights.end(); it++)
        {
                stringOfCities[i] = it->GetFliesFrom();
                i++;
        }
        int counter = 1;
        int len = INT32_MIN;
        string mostCommon;
        for (i=0;i<flights.size()-1;i++)
        {
                counter = 1;
                for (int j = i+1; j < flights.size(); j++)
                {
                        if (stringOfCities[i]==stringOfCities[j])
                        {

                                counter++;
                        }
                }
                if (len<counter)
                {
                        len = counter;
                        mostCommon = stringOfCities[i];
                }
        }
        cout << "\n\nMost Common City is " << mostCommon << " and occured " << len << "
times." << endl<<endl;
}
void FlightData::TakeOffFlightAfterTime(string time,string fromPort)
{
        list<FlightInfo>::iterator it;
        bool flag = false;
        for (it = flights.begin(); it != flights.end();it++)
        {
                if (it->GetTimeAired()>time && it->GetFliesFrom()==fromPort)
                {
```

```
                        flag = true;
                        cout << it->GetFlightID() << " "
                               << it->GetFliesFrom() << " "
                               << it->GetFliesTo() << " "
                               << it->GetTimeAired() << " "
                               << it->GetTimeArrived() << " "
                               << it->GetPrice() << endl;
                }

        }
        if (!flag)
        {
                cout << "\n\t\tNo flights found" << endl;
        }
}
ostream &operator<<(ostream &toStream, list<FlightData> &obj)
{
        list<FlightData>::iterator it;
        for (it = obj.begin(); it!=obj.end(); it++)
        {
                toStream << it->flights;
        }

        //toStream << obj.flights;
        return toStream;
}


int FlightData::FindFlight(string fromPort, string toPort,vector<string>
&cities,AdjacencyMatrix& matrix)
{


        int from;
        int to;
        for (int i = 0; i < cities.size(); i++)
        {

                if (cities[i] == fromPort)
                {
                        from = i;
                }
                if (cities[i] == toPort)
                {
                        to = i;
                }

        }
        if (matrix.HasEdge(from, to))
        {
                return 0;
        }
        cout << endl << endl;

        int n= matrix.DepthFirstSearch(from,to,cities.size());


        return n;
}

int FlightData::FindFlightPrice(string fromPort, string toPort, vector<string>
&cities, AdjacencyMatrix &matrix)
```

```
{


	int from;
	int to;
	for (int i = 0; i < cities.size(); i++)
	{

		if (cities[i] == fromPort)
		{
			from = i;
		}
		if (cities[i] == toPort)
		{
			to = i;
		}

	}


	return matrix.Dijkstra(from, to,cities);
}
int FlightData::HasFlight(int fromPort, int toPort,vector<string> nodes)
{
	for each (auto var in flights)
	{
		if ((nodes[fromPort]==var.GetFliesFrom() &&
nodes[toPort]==var.GetFliesTo())||(nodes[fromPort]==var.GetFliesTo() &&
nodes[toPort]==var.GetFliesFrom()))
		{
			return var.GetPrice();
		}
	}
	return -1;
}
void FlightData::AddFlight(FlightInfo &obj)
{
	flights.push_back(obj);
}
bool FlightData::HasCity(string city)
{
	for each (auto var in flights)
	{
		if (city==var.GetFliesFrom()||city==var.GetFliesTo())
		{
			return true;
		}
	}
	return false;
}
bool FlightData::HasID(string id)
{
	for each (auto var in flights)
	{
		if (id == var.GetFlightID() )
		{
			return true;
		}
	}
	return false;
}
```

```
bool FlightData::SatisfiesTime(string timeTakeoff,string fromPort,string
timeLands,string toPort, AdjacencyMatrix &matrix, vector<string> &cities)
{
	list<FlightInfo>::iterator itFlight;
	bool flag=false;
	vector<int> path;
	int from;
	int to;
	for (int i = 0; i < cities.size(); i++)
	{

		if (cities[i] == fromPort)
		{
			from = i;
		}
		if (cities[i] == toPort)
		{
			to = i;
		}

	}
	path=matrix.ShortestPath(from, to, cities);
	int idx=0;
	cout << "\tAvailable flights: " << endl;
	for (itFlight = flights.begin(); itFlight != flights.end();itFlight++)
	{


		if (idx < path.size() - 1)
		{


			if ((itFlight->GetFliesFrom() == cities[path[idx]] && itFlight-
>GetFliesTo() == cities[path[idx + 1]]))
			{
				//cout << "MATCH";
				if (timeTakeoff<=itFlight->GetTimeAired() && itFlight-
>GetTimeArrived()<=timeLands)
				{
					cout << "\t\t"<< itFlight->GetFliesFrom() << " to "
<< itFlight->GetFliesTo() << " takes off " << itFlight->GetTimeAired() << " arrives at
" << itFlight->GetTimeArrived() << " price: " << itFlight->GetPrice()<<endl;
					flag = true;
					itFlight = flights.begin();
					idx++;
				}
				else
				{
					flag = false;
				}

			}
		}
	}


	return flag;
}
list<FlightInfo> FlightData::GetFlights()
{
	return flights;
}
```

Хедърен файл UserRequest.h

```
#pragma once
#ifndef USERREQUEST_H_
#define USERREQUEST_H_
#include <string>
#include <fstream>
#include <list>
#include <iterator>
using namespace std;
class UserRequest
{
public:
	UserRequest() {};
	~UserRequest() {};
	UserRequest(string id,string fromPort,string toPort,string timeaired,string
timearrived);
	void SetRequesttID(string id);
	void SetFliesTo(string flight);
	void SetFliesFrom(string flight);
	void SetTimeAired(string time);
	void SetTimeArrived(string &time);
	string GetFlightID();
	string GetFliesTo();
	string GetFliesFrom();
	string GetTimeAired();
	string GetTimeArrived();
	friend istream &operator>>(istream &fromStream,UserRequest &obj);
	friend ostream &operator<<(ostream &toStream, list<UserRequest> &obj);
private:
	string requestID;
	string fliesFrom;
	string fliesTo;
	string timeAired;
	string timeArrived;
};


#endif
```

Сорс файл UserRequest.cpp

```
#include "UserRequest.h"

bool DateCheck(string time)
{
	if ((time.substr(0, 4).size() == 4) &&
	// Check YYYY
		(time.substr(5, 2) > "00"&&time.substr(5, 2) < "13") &&
	//Check MM
		(time.substr(8, 2) > "00"&&time.substr(8, 2) <= "31") &&	//Check DD
		(time.substr(11, 2) > "0"&&time.substr(11, 2) <= "24") &&	//Check HH
		(time.substr(14, 2) >= "00"&&time.substr(14, 2) < "60") &&	//Check
Minutes
		(time.size() == 16))
	{
		return true;
	}
	else
	{
		return false;
	}
}
```

```
UserRequest::UserRequest(string id, string fromPort, string toPort, string timeaired,
string timearrived)
{
	requestID = id;
	fliesFrom = fromPort;
	fliesTo = toPort;
	timeAired = timeaired;
	timeArrived = timearrived;
}

void UserRequest::SetRequesttID(string id) { requestID = id; }
void UserRequest::SetFliesTo(string flight) { fliesFrom = flight; }
void UserRequest::SetFliesFrom(string flight) { fliesTo = flight; }
void UserRequest::SetTimeAired(string time)
{
	if (DateCheck(time))
	{
		timeAired = time;
	}
	else
	{
		throw;
	}
}
void UserRequest::SetTimeArrived(string &time){
	if (DateCheck(time))
	{
		timeArrived = time;
	}
	else
	{
		throw;
	}
}
string UserRequest::GetFlightID() { return requestID; }
string UserRequest::GetFliesTo() { return fliesTo; }
string UserRequest::GetFliesFrom() { return fliesFrom; }
string UserRequest::GetTimeAired() { return timeAired; }
string UserRequest::GetTimeArrived() { return timeArrived; }
istream &operator>>(istream &fromStream, UserRequest &obj)
{
	fromStream >> obj.requestID >> obj.fliesFrom >> obj.fliesTo >> obj.timeAired >>
obj.timeArrived;
	obj.SetTimeAired(obj.timeAired);
	obj.SetTimeArrived(obj.timeArrived);
	return fromStream;
}
ostream &operator<<(ostream &toStream, list<UserRequest> &obj)
{
	list<UserRequest>::iterator it;
	for (it = obj.begin(); it != obj.end(); it++)
	{
		toStream << it->GetFlightID() << " " << it->GetFliesFrom() << " " << it-
>GetFliesTo() << " " << it->GetTimeAired() << " " << it->GetTimeArrived() << endl;
	}

	return toStream;
}
```

Хедърен файл User.h

```
#pragma once
#ifndef USERS_H_
#define USERS_H_
#include <list>
#include "UserRequest.h"
#include <fstream>
#include <iterator>
#include <string>
#include <iostream>
#include "FlightData.h"
#include "AdjacencyMatrix.h"
#include <ctime>
class Users
{
public:
	Users() {};
	~Users() {};
	Users(string fileName);
	void AddUserRequest(string flightid, string fromPort, string toPort, string
timeaired, string timearrived);
	void MostRequestsToCity();
	void PrintData();
	void CheapestFlightById(string id, list<FlightData> flights,AdjacencyMatrix
&matrix,vector<string> cities);
	void CheapestFlightForAllReq( list<FlightData> flights,AdjacencyMatrix
&matrix,vector<string> cities);
	bool HasID(string id);
	friend ostream &operator<<(ostream &toStream, list<Users> &obj);
private:
	list<UserRequest> users;
};


#endif
```

Сорс файл User.cpp

```
#include "Users.h"

Users::Users(string fileName)
{
	ifstream file(fileName.data());
	if (file.good())
		copy(istream_iterator<UserRequest>(file),
istream_iterator<UserRequest>(), back_inserter(users));
	else throw;
}
void Users::AddUserRequest(string requestId, string fromPort, string toPort, string
timeaired, string timearrived)
{
	UserRequest request;
	request.SetRequesttID(requestId);
	request.SetFliesFrom(fromPort);
	request.SetFliesTo(toPort);
	request.SetTimeAired(timeaired);
	request.SetTimeArrived(timearrived);
	users.push_back(request);
}
void Users::MostRequestsToCity()
{
```

```
	string *stringOfCities;
	stringOfCities = new string[users.size()];
	list<UserRequest>::iterator it;
	int i = 0;
	for (it = users.begin(); it != users.end(); it++)
	{
		stringOfCities[i] = it->GetFliesTo();
		i++;
	}
	int counter = 1;
	int len = INT32_MIN;
	string mostCommon;
	for (i = 0; i<users.size() - 1; i++)
	{
		counter = 1;
		for (int j = i + 1; j < users.size(); j++)
		{
			if (stringOfCities[i] == stringOfCities[j])
			{
				counter++;
			}
		}
		if (len<counter)
		{
			len = counter;
			mostCommon = stringOfCities[i];
		}
	}
	cout << "\n\nMost Common City is " << mostCommon << " and occured " << len << "
times." << endl << endl;
}
void Users::PrintData()
{
	cout << users;
}
void Users::CheapestFlightById(string id, list<FlightData> flights,AdjacencyMatrix
&matrix,vector<string> cities)
{
	list<UserRequest>::iterator itUsers;
	list<FlightData>::iterator itFlights=flights.begin();
	FlightData flightData("Flights.txt");
	UserRequest specifiedUser;
	int price=0;
	for (itUsers = users.begin(); itUsers != users.end(); itUsers++)
	{
		if (itUsers->GetFlightID() == id)
		{
			specifiedUser = UserRequest(itUsers->GetFlightID(), itUsers-
>GetFliesFrom(), itUsers->GetFliesTo(), itUsers->GetTimeAired(), itUsers-
>GetTimeArrived());
		}

	}


	if
(!(flightData.HasCity(specifiedUser.GetFliesFrom())&&flightData.HasCity(specifiedUser.
GetFliesTo())))
	{
		cout << "\tThis User Request ( "<<specifiedUser.GetFliesFrom()<<" to
"<<specifiedUser.GetFliesTo()<<" ) doesn't meet actual flights!" << endl<<endl<<endl;
		return;
```

```
	}
	if
(flightData.FindFlight(specifiedUser.GetFliesFrom(),specifiedUser.GetFliesTo(),cities,
matrix)>=0)
	{
		if (flightData.SatisfiesTime(specifiedUser.GetTimeAired(),
specifiedUser.GetFliesFrom(), specifiedUser.GetTimeArrived(),
specifiedUser.GetFliesTo(), matrix, cities))
		{
			price = flightData.FindFlightPrice(specifiedUser.GetFliesFrom(),
specifiedUser.GetFliesTo(), cities, matrix);
			cout << "\n\n\tPrice between " << specifiedUser.GetFliesFrom() <<
" and " << specifiedUser.GetFliesTo() << " is:" << price << endl << endl << endl <<
endl << endl << endl;
		}
		else
		{
			cout << "\t\tYour flight request don't meet any current flights!"
<< endl<<endl<<endl;
		}
	}
	else
	{
		cout << "\t\tThis User Request doesn't meet actual flights!" << endl;
	}
}
ostream &operator<<(ostream &toStream, list<Users> &obj)
{
	list<Users>::iterator it;
	for (it = obj.begin(); it != obj.end(); it++)
	{
		toStream << it->users;
	}

	//toStream << obj.flights;
	return toStream;
}
bool  Users::HasID(string id)
{
	for each (auto var in users)
	{
		if (id==var.GetFlightID())
		{
			return true;
		}
	}
	return false;
}
void Users::CheapestFlightForAllReq(list<FlightData> flights,AdjacencyMatrix
&matrix,vector<string> cities)
{
	clock_t begin = clock();
	list<FlightData>::iterator itFlights;
	list<UserRequest>::iterator itUsers;
	FlightData flightData("Flights.txt");

	for (itUsers = users.begin(); itUsers != users.end();itUsers++)
	{
		cout << "For User Request:\n\t"<<itUsers->GetFlightID()<<" from "<<
itUsers->GetFliesFrom()<<" to "<<itUsers->GetFliesTo()<<" sets off "<<itUsers-
>GetTimeAired()<<" lands "<<itUsers->GetTimeArrived();
		CheapestFlightById(itUsers->GetFlightID(), flights,matrix,cities);
```

```
	}
	clock_t end = clock();
	double elapsedtime = double(end - begin) / CLOCKS_PER_SEC;
	cout << elapsedtime;



}
```

Главен файл Main.cpp

```
#include <iostream>
#include <fstream>
#include <time.h>
#include "AdjacencyMatrix.h"
#include "FlightInfo.h"
#include "FlightData.h"
#include "UserRequest.h"
#include "Users.h"
#include <map>
#include <vector>
#include <set>
using namespace std;
bool CheckDateTime(string time)
{
	int size = time.size();
	if (size == 16)
	{


		if ((time.substr(0, 4).size() == 4) &&
		// Check YYYY
			(time.substr(5, 2) > "00"&&time.substr(5, 2) < "13") &&
	//Check MM
			(time.substr(8, 2) > "00"&&time.substr(8, 2) <= "31") &&
	//Check DD
			(time.substr(11, 2) > "0"&&time.substr(11, 2) <= "24") &&
	//Check HH
			(time.substr(14, 2) >= "00"&&time.substr(14, 2) <= "60") &&
	//Check Minutes
			(time.size() == 16))
		{
			return true;
		}
		else
		{
			return false;
		}
	}
	else if (size==4)
	{
		if ((time.substr(0, 4).size() == 4) && (time.substr(0, 4)>="1999"&&
time.substr(0, 4)<="2020"))
		{
			return true;
		}
		else
		{
			return false;
		}
	}
```

```
	else if (size < 8)
	{
		if ((time.substr(0, 4).size() == 4) && (time.substr(0, 4) >= "1999"&&
time.substr(0, 4) <= "2020")&& (time.substr(5, 2) > "00"&&time.substr(5, 2) < "13"))
		{
			return true;
		}
		else
		{
			return false;
		}
	}
	else if (size>=8 && size <12)
	{
		if ((time.substr(0, 4).size() == 4) &&
			(time.substr(0, 4) >= "1999"&& time.substr(0, 4) <= "2020") &&
			(time.substr(5, 2) > "00"&&time.substr(5, 2) < "13")&&
			(time.substr(8, 2) > "00"&&time.substr(8, 2) <= "31"))
		{
			return true;
		}
		else
		{
			return false;
		}
	}
	return true;
}
void Menu();
void SubMenu(list<FlightData>::iterator itFlight, list<FlightData>
flights,vector<string> &cities,AdjacencyMatrix &matrix);

int main()
{

	Menu();


}

void Menu()
{

	int choice = 0;
	list<FlightData> flights;
	list<Users> users;
	flights.push_back(FlightData("Flights.txt"));
	users.push_back(Users("Users.txt"));
	list<FlightData>::iterator itFlight=flights.begin();
	list<Users>::iterator itUser = users.begin();
	vector<string> cities;
	list<FlightInfo> flightInfo=itFlight->GetFlights();
	list<FlightInfo>::iterator it;
	for (it = flightInfo.begin(); it != flightInfo.end();it++ )
	{

		cities.push_back(it->GetFliesFrom());
		cities.push_back(it->GetFliesTo());
	}

	sort(cities.begin(), cities.end());
	cities.erase(unique(cities.begin(), cities.end()), cities.end());
```

```
	//leaving only our unique nodes



	AdjacencyMatrix matrix = AdjacencyMatrix(cities.size());


	cout << endl;

	for (int i = 0; i < cities.size(); i++)
	{
		for (int j = 0; j < cities.size(); j++)
		{
			if (itFlight->HasFlight(i, j, cities)>0)
			{
				matrix.AddEdge(i, j,itFlight->HasFlight(i, j, cities));
			}
		}
	}
	do
	{
		cout << "\n\n*             MENU          *" << endl;
		cout << "1.Add new Flight." << endl;
		cout << "2.Add new User Request." << endl;
		cout << "3.Change data by ID." << endl;
		cout << "4.Print information about all flights." << endl;
		cout << "5.Print information about all user request." << endl;
		cout << "6.Find and print information about flights from city after
specified time." << endl;
		cout << "7.Find from which city has most flights." << endl;
		cout << "8.Find to which city there are most requested flights." << endl;
		cout << "9.Display cheapest flight via user's request ID." << endl;
		cout << "10.Display cheapest flights for all user requests." << endl;
		cout << "11.Enter Submenu of class FlightData" << endl;
		cout << "12.Save updated data in file." << endl;

		cout << "\n Your choice: ";
		cin >> choice;
		switch (choice)
		{
		case 1:
		{
			cout << "\n**********" << endl;
			cout << "Enter Flight ID:";
			string info;
			cin >> info;
			cout << "\nEnter taking off city:";
			string fromTown;
			cin >> fromTown;
			cout << "\nEnter arrival city:";
			string toTown;
			cin >> toTown;
			cout << "\nEnter aired date and time {ONLY IN FORMAT YYYY/MM/DD-
HH:MM}:";
			string timeAired;
			cin >> timeAired;
			cout << "\nEnter arrival date and time {ONLY IN FORMAT YYYY/MM/DD-
HH:MM}:";
			string timeArrived;
			cin >> timeArrived;
			cout << "\nEnter price of the flight:";
			int price;
```

```
                    cin >> price;
                    itFlight = flights.begin();
                    itFlight-
>AddFlight(info,fromTown,toTown,timeAired,timeArrived,price);
             }
                    //flights.push_back("","","","","",5);
                    break;
             case 2:
             {
                    cout << "\n**********" << endl;
                    cout << "Enter User Request ID: ";
                    string info;
                    cin >> info;
                    cout << "\nEnter taking off city: ";
                    string fromTown;
                    cin >> fromTown;
                    cout << "\nEnter arrival city: ";
                    string toTown;
                    cin >> toTown;
                    cout << "\nEnter aired date and time {ONLY IN FORMAT YYYY/MM/DD-
HH:MM}: ";
                    string timeAired;
                    cin >> timeAired;
                    cout << "\nEnter arrival date and time {ONLY IN FORMAT YYYY/MM/DD-
HH:MM}: ";
                    string timeArrived;
                    cin >> timeArrived;

                    itUser =users.begin();
                    itUser->AddUserRequest(info, fromTown, toTown, timeAired,
timeArrived);
             }
                    break;
             case 3:
             {
                    itFlight = flights.begin();
                    string id;
                    cout << "\nEnter ID of a flight: ";
                    cin >> id;
                    if (!itFlight->HasID(id))
                    {
                           cout << "No such flight id was found!" << endl << endl;
                           break;
                    }
                    itFlight->CorrectDataByID(id);
             }
                    break;
             case 4:
             {
                    for (itFlight = flights.begin(); itFlight != flights.end();
itFlight++)
                    {
                           itFlight->PrintData();
                    }
             }
                    break;
             case 5:
             {
                    for (itUser = users.begin(); itUser != users.end();itUser++)
                    {
                           itUser->PrintData();
                    }
```

```
		}
			break;
		case 6:
		{
			itFlight = flights.begin();
			cout << "\n\nEnter city: ";
			string city;
			cin >> city;
			if (!itFlight->HasCity(city))
			{
				cout << "\n\t\tThere is no such city!"<<endl;
				break;
			}
			cout << "\nEnter aired date and time {ONLY IN FORMAT YYYY/MM/DD-
HH:MM}: ";
			string time;
			cin >> time;
			if (!CheckDateTime(time))
			{
				cout << "\n\nIncorrect time!" << endl;
				break;
			}

			itFlight->TakeOffFlightAfterTime(time,city);
		}

			break;
		case 7:
		{
			itFlight = flights.begin();
			itFlight->MostFlightsFromCity();
		}
			break;
		case 8:
		{
			itUser = users.begin();
			itUser->MostRequestsToCity();
		}
			break;
		case 9:
		{
			itUser = users.begin();
			cout << "\nEnter user ID:";
			string id;
			cin >> id;
			if (!itUser->HasID(id))
			{
				cout << "No such flight id was found!" << endl << endl;
				break;
			}
			itUser->CheapestFlightById(id, flights,matrix,cities);
			break;
		}
		case 10:
		{
			itUser = users.begin();
			itUser->CheapestFlightForAllReq(flights,matrix,cities);
			break;
		}
		case 11:
			SubMenu(itFlight,flights,cities,matrix);
			break;
```

```
		case 12:
		{

			ofstream fileWriter;
			fileWriter.open("Flights.txt");
			fileWriter << flights;
			fileWriter.close();

			fileWriter.open("Users.txt");
			fileWriter << users;
			fileWriter.close();

			cout << "\n\n\n\tUploaded data saved!" << endl << endl << endl;
			break;
		}

			break;
		default:
			cout << "\n\n\tWrong input!\n\n" << endl;
			break;
		}

	} while (choice>0 && choice<13);
}

void SubMenu(list<FlightData>::iterator itFlight,list<FlightData> flights,
vector<string> &cities, AdjacencyMatrix &matrix)
{

	int choice = 0;
	do
	{
		cout << "\n*        SUBMENU                    *" << endl;
		cout << "1.Find all flights airing from a given port."<<endl;
		cout << "2.Find all flights traveling to a given port." << endl;
		cout << "3. Check if there is flight between two given cities." << endl;
		cout << "4. Print Graph." << endl;
		cout << "5.Go back to main menu." << endl;
		cout << "Enter your choice:";
		cin >> choice;

		switch (choice)
		{
		case 1:
			{
				itFlight = flights.begin();
				string port;
				cout << "Enter port:";
				cin >> port;
				if (!itFlight->HasCity(port))
				{
					cout << "City doesn't exist!" << endl << endl;
					break;
				}
				list<FlightInfo>listPtr= itFlight->GetFlightsFrom(port);
				cout << "Flights Info:"<<endl;
//
				for each (auto var in listPtr)
				{
					cout<<var.GetFlightID() <<" "<<
var.GetFliesFrom()<<" "<<var.GetFliesTo()<<" "<<var.GetTimeAired()<<"
"<<var.GetTimeArrived()<<" "<<var.GetPrice()<<endl;
```

```
                        }
                    }
                    break;
                case 2:
                {
                    itFlight = flights.begin();
                    string port;
                    cout << "Enter port:";
                    cin >> port;
                    if (itFlight->HasCity(port))
                    {


                        list<FlightInfo>listPtr = itFlight->GetFlightsTo(port);
                        cout << "Flights Info:" << endl;

                        for each (auto var in listPtr)
                        {
                            cout << var.GetFlightID() << " " <<
var.GetFliesFrom() << " " << var.GetFliesTo() << " " << var.GetTimeAired() << " " <<
var.GetTimeArrived() << " " << var.GetPrice() << endl;
                        }
                    }
                    else
                    {
                        cout << "City doesn't exist!" << endl << endl;
                    }
                }
                    break;
                case 3: {
                    itFlight = flights.begin();
                    cout << "\nEnter airing port:";
                    string portOne = "";
                    cin >> portOne;
                    if (!itFlight->HasCity(portOne))
                    {
                        cout << "City doesn't exist!" << endl << endl;
                        break;
                    }
                    cout << "\nEnter decending port:";
                    string portTwo = "";
                    cin >> portTwo;
                    if (!itFlight->HasCity(portTwo))
                    {
                        cout << "City doesn't exist!" << endl << endl;
                        break;
                    }
                    cout << endl << endl;
                    int result=itFlight->FindFlight(portOne, portTwo,cities,matrix);
                    itFlight = flights.begin();
                    itFlight->FindFlightPrice(portOne, portTwo, cities, matrix);
                    switch (result)
                    {
                    case 1:
                    {
                        cout << "\n\tThis flight is a stopover!" <<
endl<<endl<<endl;
                        break;
                    }
                    case 0:
                    {
                        cout << "\n\tDirect flight!" << endl;
```

```
                        break;
                    }
                    case -1:
                    {
                        cout << "\n\tThis flight path doesn't exist!" << endl;
                        break;
                    }
                    case -2:
                    {
                        cout << "\n\tCannot fly to the same city!" << endl;
                        break;
                    }
                    default:
                        break;
                    }

                    break;

            }
            case 4:
                    matrix.PrintGraph(cities.size(),cities);
                    break;
            default:
            { }
                    break;
            }
        } while (choice>0&&choice<5);
}
```