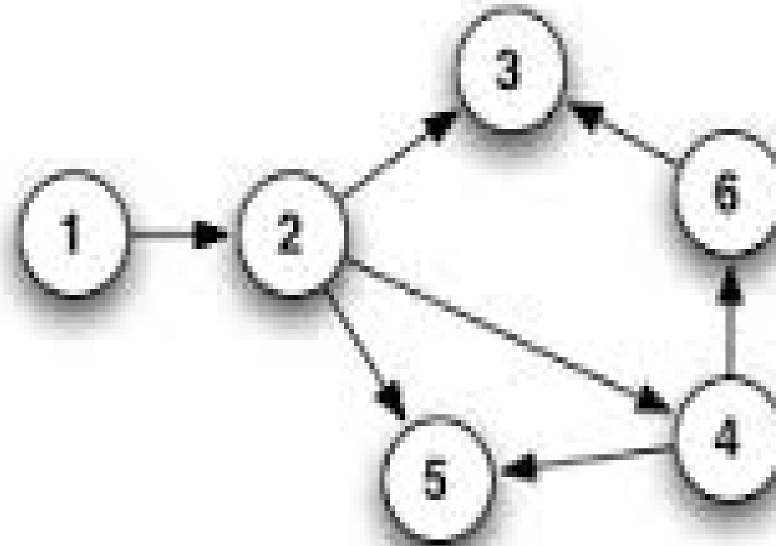
Travel Network
Project 6

CS 211 – Fall 2018

Travel Network

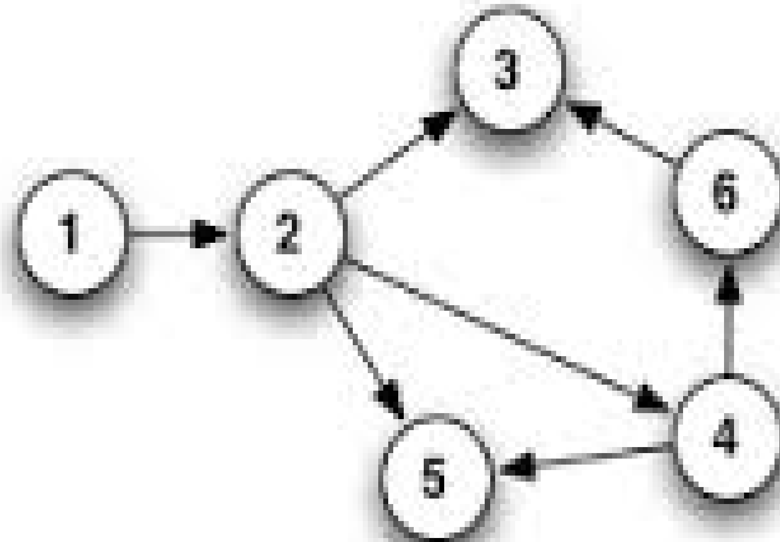
- A group of N “airport” with flights from airport X to airport Y

- 6 Airports
- Flights from:
    - Airport 1 to 2
    - Airport 2 to 3
    - Airport 2 to 4
    - Airport 2 to 5
    - Airport 4 to 5
    - etc



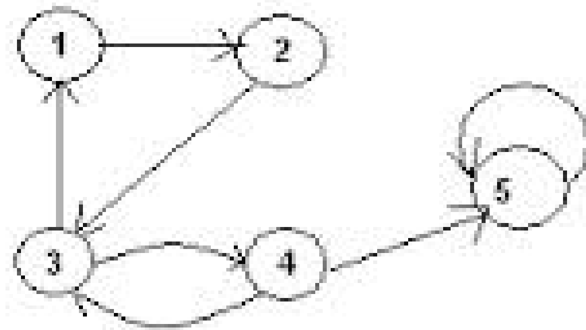
Travel Network

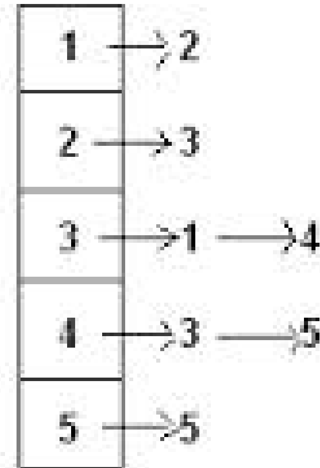
- Flights from Airport X are maintained in an “Adjacency List”
- Adjacency List is a Linked List
- Adjacency List for Airport X
  contains which airports can
  be Traveled to in one Flight
- Adjacency List for Airport 1
  Contains: 2
- Adjacency List for Airport 2
  Contains: 3, 4, 5



Travel Network

- All of the “Adjacency List” are contained in a Dynamic Array of Linked Lists
- This example needs 5 lists:
- The dynamic array in my
  solution is of size 6.
- Location 0 is not used.
- So Airport Number will match Array Location





Travel Network Commands

- q – quit the program (already implemented in proj6Base.cpp)
- ? – display the list of commands (already implemented)
- # – ignore this line of input (already implemented)
- t <int1> <int2>
  - Determine if a person can get from airport <int1> to airport <int2> in one or more flights (use the depth first search algorithm provided)
- r <int>
  - Remove all values from travel network and resize the dynamic array to allow for <int> airports (make sure you have no memory leaks!)

Travel Network Commands

- i <int1> <int2>
  - Insert a flight from airport <int1> to airport <int2>
  - Add the value <int2> in the adjacency list for airport <int1>
- d <int1> <int2>
  - Remove the flight from airport <int1> to airport <int2>
  - Remove the value <int2> from the adjacency list for airport <int1>
- l
  - List all information contained in the travel network
  - For each airport, list all airports in its adjacency list

Travel Network Commands

- f <filename>
  - Open the file indicated by <filename> and read commands from this file
  - Assume the file is in the current directory
  - When the end of the file is reached, continue reading commands from the previous input source. This must be done using recursion.
  - Beware of the possible case of an infinite recursive loop.
    - Don’t let the f command call a file that is currently in use.

Travel Network Required Classes

The write-up discusses 4 classes:

- Node class for the List (see Lab11)
- List Class (see Lab11) – don't name it "List"
- Airport Class
  - Contains a List Instance – the adjacency list
  - Contains the Boolean "visited" as needed for the Depth First Search
- TravelNetwork class
  - Contains the dynamic array of Airport Instances (initially of size 10)
  - Plus other supporting information (in-use filenames, size of dynamic array)

Travel Network Required Files

- The write-up discusses (at least) 2 source code files,
  - plus header file and makefile
- Source code file #1: Airport.cpp
  - Airport class for sure
  - Mine also includes my List and Node classes
  - Header file “Airport.h” contains information for these 3 classes
- Source code file #2: NetidProj6.cpp
  - main() and TravelNetwork class
  - Includes “Airport.h”

Travel Network Command Line Arguments

- None required
- We leave it up to you if you want a “debug mode flag” to help in testing your program
- We will grade your program without giving any command line arguments

Travel Network t Command

t <int1> <int2> - call the depth first search algorithm

- Depth First Search must be methods in the TravelNetwork class

```
void  TravelNetwork::depthFirstSearchHelper (int x, int y);
bool  TravelNetwork::dfs (int a, int b)
```

- <int1> becomes value for x parameter (and initial a parameter)
- <int2> becomes value for y parameter (and initial b parameter)
- Access to the dynamic array of airports via the **this** pointer
- You MUST use the given recursive algorithm

Travel Network t Command

```
void depthFirstSearchHelper (int x, int y)
{
   mark all airports as unvisited;
   if ( dfs (x, y) = = TRUE)   // call the recursive code
       print ("You can get from airport " + x + " to airport " + y +
                  " in one or more flights");
   else
       print ("You can NOT get from airport " + x + " to airport " + y +
                  " in one or more flights");
}
```

Depth First Search Recursive method

```
boolean dfs (int a, int b)   {                     // trying to get from a to b
   for (each airport c that can be reached from a in one flight)
      {
         if (c = = b)                              // Check if I can get directly from a to b
               return TRUE;                        //   if so, report success!
         if ( airport c is unvisited )             // needed to stop possible infinite loop!!
            {
              mark airport c as visited;           // make sure we don't return to c
             if ( dfs (c, b) = = TRUE )            //  see if I can get from c to b
                 return TRUE;                      //   if so, report success!
            }
      }
  return FALSE;                                    //  otherwise report failure
}
```

Travel Network t Command

```
boolean dfs (int a, int b)
{
   for (each airport c that can be reached from a in one flight)
        {
           …   // do stuff with airport c
        }
   return FALSE;  // so far, no path to target airport
}
```

Travel Network t Command

```
boolean dfs (int a, int b)
{
   for (each airport c that can be reached from a in one flight)
```

- Values for c come from the values in the adjacency list for airport a
- Assumes use of the following methods from your List class:
  - getNumberOfCurrentsValues( )   (named getListLength( ) in Lab 11)
  - getNthValue( )
- Values in adjacency list could be in any order – so paths may vary!

Travel Network t Command

```
boolean dfs (int a, int b)
{
   for (each airport c that can be reached from a in one flight)
      {
         …   // do stuff with airport c
      }
   return FALSE;  // so far, no path to target airport
}
```

Travel Network t Command

```
boolean dfs (int a, int b)
{
  for (each airport c that can be reached from a in one flight)
      {
          if (c = = b)      // we have reached the target airport
             return TRUE;
          …
      }
  return FALSE;
}
```

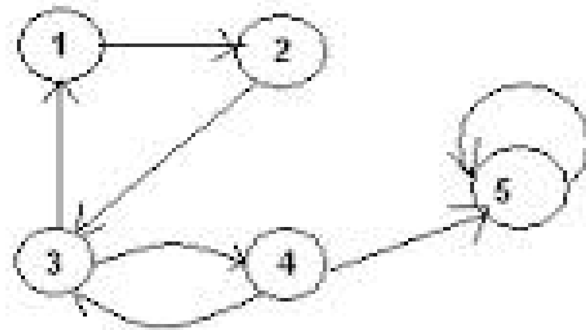
Travel Network t Command

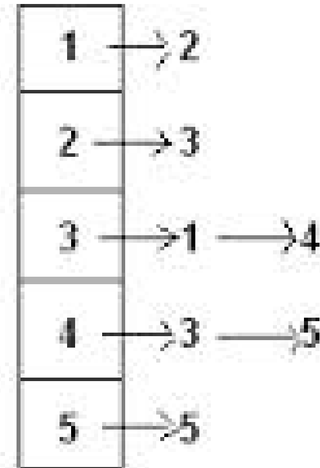
```
boolean dfs (int a, int b)
{
  for (each airport c that can be reached from a in one flight)
      {
          if (c = = b)
             return TRUE;
          if ( airport c is unvisited )  // needed to stop possible infinite loop!!
          …
      }
  return FALSE;
}
```

Travel Network – Possible Infinite Loop

- If we don't mark airports as visited: the following could occur for
  t 2 4

- First, go from 2 to 3
- Then, go from 3 to 1
- Then, go from 1 to 2
- Then, go from 2 to 3
- Then, go from 3 to 1
- Then, go …





Travel Network t Command

```
boolean dfs (int a, int b)
{
  for (each airport c that can be reached from a in one flight)
      {
          if (c = = b)
              return TRUE;
          if ( airport c is unvisited )  // needed to stop possible infinite loop!!
          …
      }
  return FALSE;
}
```

Travel Network t Command

```
boolean dfs (int a, int b)
 {    …
          if ( airport c is unvisited )  // needed to stop possible infinite loop!!
           {
             mark airport c as visited;     // make sure we don't return to c
             if ( dfs (c, b) = = TRUE )      //  see if I can get from c to b
                  return TRUE;               //   if so, report success!
           }
        }
   return FALSE;
}
```

Travel Network t Command

```
boolean dfs (int a, int b)                          // we want to get from a to b
{    …
         if ( airport c is unvisited )
           {
              mark airport c as visited;            //  we can get from a to c
              if ( dfs (c, b) = = TRUE )            //  see if I can get from c to b
                   return TRUE;                     //   report success from a to b
           }                                        //   because we can get from a to c
         }                                          //   and from c to b
  return FALSE;
}
```

Depth First Search Recursive method

```
boolean dfs (int a, int b)   {                         // trying to get from a to b
   for (each airport c that can be reached from a in one flight)
      {
          if (c = = b)                                 // Check if I can get directly from a to b
               return TRUE;                            //   if so, report success!
          if ( airport c is unvisited )                // needed to stop possible infinite loop!!
              {
               mark airport c as visited;              // make sure we don't return to c
               if ( dfs (c, b) = = TRUE )              //  see if I can get from c to b
                   return TRUE;                        //   if so, report success!
              }
       }
  return FALSE;                                        //  otherwise report failure
}
```

The FILE Command: f

f <filename>

- Open the file indicated by <filename> and read commands from this file
- Assume the file is in the current directory
- When the end of the file is reached, continue reading commands from the previous input source.   This must be done using recursion.
- Beware of the possible case of an infinite recursive loop.
    - Don’t let the f command call a file that is currently in use.

- Code already exists in proj6Base.cpp to set up for this

The FILE Command: f

- First note the prototype for processCommandLoop ( )
  - void TravelNetwork::processCommandLoop (FILE* inFile);
  - A file pointer as its parameter
- Note how this is called from main( ):

```
FILE* inFile = stdin;
…
TravelNetwork airportData;
airportData.processCommandLoop (inFile);
```

- Sends a “file pointer” that refers to Standard Input

The FILE Command: f

When initially called from main( ), the method

```
void processCommandLoop (FILE* inFile)
```

reads from Standard Input

So, the line of code that reads the input:

```
input = fgets ( buffer, 300, inFile );
```

Gets input from the user via the keyboard/Standard Input

So input comes from what the user types in at the keyboard

The FILE Command: f

What if processCommandLoop( ) is called with the **inFile** parameter that refers to a File rather than Standard Input?

So, the line of code that reads the input:

```
input = fgets ( buffer, 300, inFile );
```

Gets input from the file

So the same code can get input comes from either the keyboard or from a file, depending on the parameter value used at the call!

The FILE Command: f

doFile( ) indicates the following steps once the filename is read in:

```
// next steps:  (if any step fails: print an error message and return )
//  1. verify the file name is not currently in use
//  2. open the file using fopen creating a new instance of FILE*
//  3. recursively call processCommandLoop() with this new instance of
//               FILE* as the parameter
//  4. close the file when processCommandLoop() returns
```

The FILE Command: f - Step-by-Step

1. verify the file name is not currently in use
2. open the file using fopen creating a new instance of FILE*
3. recursively call processCommandLoop() with this new instance of FILE* as the parameter
4. close the file when processCommandLoop() returns

Skip Step 1 for now (it is the most complicated), come back to it

The FILE Command: f - Step 2

open the file FOR READING using fopen( )

From: http://www.cplusplus.com/reference/cstdio/fopen/

```
FILE * fopen ( const char * filename, const char * mode );
```

Opens the file whose name is specified in the parameter filename and associates it with a stream that can be identified in future operations by the FILE pointer returned.

```
FILE * pFile;
pFile = fopen ("myfile.txt","r");   // open file for reading
```

The FILE Command: f - Step 2

What if filename does not exist?

From: http://www.cplusplus.com/reference/cstdio/fopen/

```
FILE * fopen ( const char * filename, const char * mode );
```

If the file is successfully opened, the function returns a pointer to a FILE object that can be used to identify the stream on future operations.

Otherwise, a null pointer is returned.

```
FILE* pFile = fopen ("myfile.txt","r");   // open file for reading
if ( pFile == NULL ) { … }                // file did not open
```

The FILE Command: f - Step 2

What if filename does not exist or fails to open?

```
FILE* pFile = fopen ("myfile.txt","r");   // open file for reading
if ( pFile == NULL )                      // file did not open
{
        print error message
        return from doFile( )
}
```

The FILE Command: f - Step 3

recursively call processCommandLoop() with this new instance of FILE* as the parameter from doFile( )

- Since both doFile( ) and processCommandLoop( ) are methods in the same TravelNetwork class instance, use the this pointer at the call.

```
FILE* inFile = fopen ( …   );            // Step 2
…
this->processCommandLoop (inFile);   // Step 3
```

The FILE Command: f - Step 3

recursively call processCommandLoop() with this new instance of FILE* as the parameter from doFile( )

```
FILE* inFile = fopen ( …   );                 // Step 2
…
this->processCommandLoop (inFile);  // Step 3
```

Now the parameter given to processCommandLoop( ) by this call will have it read from the file opened in Step 2

The FILE Command: f - Step 3

Note the while loop in processCommandLoop ( ):

```
while (input != NULL)
    …
```

When the **end of the file** is reached, the loop terminates, and the execution of processCommandLoop( ) ends returning back to doFile( )

The FILE Command: f - Step 4

close the file when processCommandLoop() returns

```
FILE* inFile = fopen ( …   );               // Step 2
…
this->processCommandLoop (inFile);  // Step 3
…
fclose (inFile);                            // Step 4
```

doFile( ) then returns back to the processCommandLoop( ) that continues reading from Standard Input

The FILE Command: f

When reading from one file, can we start reading from another file?

Yes! (If we are careful)

Data file proj6data2.txt makes calls to data file proj6data3.txt

However:

Data file proj6data4.txt makes calls to data file proj6data4.txt

And

Data file proj6data5.txt makes calls to data file proj6data4.txt

Thus why we need Step 1!!!

The FILE Command: f - Step 1

1. Verify the file name is not currently in use
    - If the file name is currently in use
        - Print error message
        - Return from doFile( ) stopping the infinite recursion
2. open the file using fopen creating a new instance of FILE*
3. recursively call processCommandLoop() with this new instance of FILE* as the parameter
4. close the file when processCommandLoop() returns

The FILE Command: f - Step 1

1. Verify the file name is not currently in use
    - If the file name is currently in use
        - Print error message
        - Return from doFile( )

How do we do this??

- We need a list of filenames that are “currently in use”
- A new data member for the TravelNetwork class

The doFile( ) method - Rethinking the steps

0. Get the filename to be used

1a. Check if filename is in the list of files “currently in use”

1b. If so, Print error and return (avoiding infinite recursion)

2a. open the file using fopen( ) creating a new instance of FILE*

2b. If file does not open properly, Print error and return

2c. Add the filename to list of files “currently in use”

3. recursively call processCommandLoop() with this new instance of FILE* as the parameter

4. close the file when processCommandLoop() returns

5. Remove the filename from list of files “currently in use”