The main objective of this project is to learn some ADT sequence operations (insertion, search, traversal, deletion, etc.) implemented with dynamic storage allocation and the doubly linked list data structure.

The specified software will manage sequences of “Cities” to help a traveller plan a trip. The executable file to be built by your build.sh script should be named triplanner; and the directory containing your source files, RCS database, and build script (minus core, non-shell-script executable files, and object files) should be submitted to project Proj2.

The software will store some simple data representing and pertaining to each City inputted: Name, the floating point numeric Cost of a hypothetical room there, and the integer Population. Hence a concrete class, class City should be designed so each instance represents one City.

The software works with sequences of Cities (see DSO chapters 3, 4 and 5). Hence a container class, class CSequence, should be designed to so each CSequence stores one sequence of Cities.

The class CSequence has all the data and function members for operating on a sequence of Cities required for the functionality of the software. The data members include a cursor that either has the “undefined” value, or, when “defined” locates a selected sequence element.

At this point, it would be very wise to read the specification below. After it, read the section on how to do the project. Then make sure you are familiar enough with the course contents featured in Main and Savitch’s DSO Chapters 2, 4 and 5 required for the data structure implementation. All the Chapter 5 operations or certain variants of them will be needed for this project, except for list copying. Study the pseudo-code and sample implementations as necessary. The essentials will be covered in the upcoming lectures and in Lab 3. When you begin draft design, creating some C++ files (RCS use required!) and the build script, and continue to complete the project, keep the this assignment sheet handy so you can consult it for details. (Maybe also use it to write notes and a “to-do” list.)

1 Specification of Commands, other inputs and actions

This software maintains for you (the user) a main sequence of Cities. The software has a primitive text-based user interface: You are prompted to type commands and data input, and the software responds by printing more prompts and/or results. (However, the operations behind the interface are similar to those behind the common graphical user interfaces to modern software.)

We will evaluate most of the functionality of your project work in the order of the description of the commands below.

For each command, the exact command name and a general syntactic description of any argument are given first in Italic Font Type. The exact texts or forms of messages the program must print are given next. Explanations of these printed results, errors to detect and report, and of the data operations follow.

What the computer should print, such as the Command> prompt, and code samples are given in Fixed Font Type. The explanations are Roman Font Type1.

---

1Software specifications and manuals often explain a typeface usage convention like this.
Each command and message text is available in the file below on itsunix
~acsi310/Proj2/messages.txt
so you can copy and paste them into your program instead of typing them. For full credit, 
messages must be printed exactly as shown.

Unlike those in a practical “user manual” type specifications, the explanations include information about how each command should be implemented because learning these implementation techniques is a project objective.

First implementation information item: There must be 3 separate modules: a “user interface” module for inputting commands and displaying some or all of the results, a separate module for sequences of Cities implemented by C++ class CSequence, and a third separate module for the C++ class City, where each instance represents one City.

Since class CSequence implements doubly-linked lists, each instance must have two pointer-to-node variables head_ptr and tail_ptr to quickly locate the first and the last node of the list. (You can use alternative names for them if you want.)

For the user interface module: Program with strcmp (covered in Ch. 4) to distinguish cases 1-12. For cases 5 and 9 you can use cin.get to “swallow” and check the space, and use cin.peek to distinguish the 3 argument cases. The code int Pos; cin >> Pos; will read a position number. Finally, you can use cin.getline() to read the search string in case 13. (Consult DSO’s appendix on Input and Output.)

1.1 Commands one by one

1. Command><misspelled or unimplemented command>
<misspelled or unimplemented command>
ERR: I don’t know that command.

If the characters at the beginning of the line does not match any of the commands defined below, or they match a command you didn’t implement, reprint the command and then print the indicated error message. (A very small amount of credit will be awarded for “flushing away” the remainder of the bad command line after the first token.) The program should continue to input and process lines, (except of course when the last input causes the program to exit).

Input of the command must be done with code like the following, or, if you want try for full error recovery, the code using cin.getline() which appears later.

```
#include <iostream>
#include <cstdlib> //supplies exit()
using package std;
...
const int INBUFSIZE = <your figure out a suitable number!!>
char inbuf[INBUFSIZE];
...

    cin.width(INBUFSIZE);
    cin >> inbuf;
```
Warning: One of the grading tests will make your program read an excessively long command line. If it crashes instead of printing the error message and going on to the next command, you will lose credit for this case.

2. Command> quit
   Happy Travelling! Bon Voyage!

   The quit command makes the editor exit. Errors are impossible!

3. Command> prepend
   Name: New York City, USA
   Population: 8100000
   Cost: 62.50
   (1)New York City pop. 8100000, $62.50
   Command> prepend
   Name: Cape Town, South Africa
   Population: 2980000
   Cost: 42.73
   (1)Cape Town, South Africa pop. 2980000, $42.73
   Command>

   The prepend command makes the computer prompt for the input of the City name, Population and typical Cost in the form exactly like the examples above.

   (Internally, the user interface module can activate the City module, and the City module can take care of obtaining the user data and constructing a new City object. Then, the user interface can pass a pointer to an appropriate member function of the CSequence object holding the main sequence. Finally, the user interface can call another member function of CSequence to return a pointer to the City at position 1, and then use it to print that City.)

   First the user must type the name beginning with a letter on one line of 80 characters or fewer, including the newline character. Next, the computer prompts for the population and cost exactly as shown, and, for each, separately, the user must respond with a positive number expressed in decimal.

   It would be OK for this project if the program refuses to go further until the input of a City is complete. (So City detail input can be done by the City module. For simplicity, that module need not report input aborts or failures to its caller.)

   The user, of course, must spell the command name prepend exactly as shown, and the your program must print the computer prompts (Name:, Population:, Cost:).

   Upon successful input of a City, that City is inserted at the beginning of the main sequence. Finally, the first sequence element is printed.
The examples above illustrate the exact format for printing a sequence element: The element number in ( ) first, and population and cost delimited as shown. Note the element numbers begin with one (not zero) because this is a user’s (not programmer’s) view.

Future commands will require City element printing in exactly the same format. Hint: Overload operator<<() for class City.

Input of the City name must be read using the following code: (For simplicity, don’t bother removing any leading or trailing whitespace from it.)

```cpp
#include <iostream>
#include <cstdlib> //supplies exit()
using package std;
...
const int INBUFSIZE = 80;
char inbuf[INBUFSIZE];

if( !cin.getline(inbuf, INBUFSIZE) )
{
    cout << "ERR:Line input failed(too long?end of file?)..exiting." << endl;
    exit( 0 );
}
```

The call to cin.getline() reads an input line up to INBUFSIZE characters long including the terminating newline character. The line includes any leading or trailing whitespace characters (spaces or tabs here). The cin.getline() call fails if the input is too long. This call stores the inputted characters into the given array except that it replaces the terminating newline with the NULL char value. That replacement turns the inputted characters into a C-string. Ask yourself and become sure of the answer: What is the maximum length of a line this software will handle?

These rules apply even to the empty input line. Users will type that in by pressing the enter key only. It’s important to understand the empty C-string: It’s a NON-EMPTY array of the one character ‘\0’.

In order to dynamically allocate the properly sized char array for the C-string to hold a copy of the input line, the program must determine its length and add 1 to this length. (Why? What can happen if I isn’t added?) After dynamically allocating (using new[] of course..) the array, the line must be copied into the array. The resulting new C-string will eventually become part of the data structure within instance of class City. The input buffer inbuf above is NOT part of City’s data structure! (What would happen if each City object held the same pointer to this buffer in its Name field?)

**TIP:** It would be really smart to design, implement and test a concrete class City for most of the above functionality all by itself. See the DSO Point class example in Chapter 2 because a City is essentially a fancy point with a name!

4. **Command> printall**
   Prints all the Cities, one per output line, preceeded by the position of each City in the
sequence, in forward order. For example, assuming just New York and Cape Town were input exactly as above, the result of the `printall` command would be:

(1) Cape Town, South Africa pop. 2980000, $42.73
(2) New York City pop. 8100000, $62.50

Note and explain to yourself why Cape Town has position 1 and is printed first.

If sequence (of Cities) is empty, print:

(empty sequence)

5. **Command>** `select <position number>`
   - **Command>** `select +`
   - **Command>** `select −`
   - ERR: Too few Cities.
   - ERR: + or − requires a selection be made first.
   - ERR: Missing or badly formatted position number.
   - ERR: Cannot select City before position 1.
   - **Command>** `select 2`
   - (2) New York City pop. 8100000, $62.50

The first variant selects the City at the given position number. (What are its cases for failure?)

The + and − variant require that a selection had already been made. They change the selection to the next and previous positions respectively.

The prior selection remains unchanged when an erroneous select command is given.

In cases of successful selection, the program will print the position number and the selected City.

Like in most graphical user interface software (GUI), once a City is selected, various operations can be performed. They are described below.

6. **Command>** `print`
   - ERR: No City is selected.
   - Print the selected position number and the City in that position.

7. **Command>** `insert`
   - ERR: No City is selected.
   - (See interaction example under prepend description above.)

   This requires a City to be selected. First, prompt for and obtain the input of a new City (just like for the prepend operation.) Then, insert that new City into the sequence so that the new City is at the same position of the selected City. The newly inserted City should become the selected City. Therefore, the selected City and all Cities after it will be moved one position towards the end. Finally, (just like for prepend) the newly inputted and selected position and City should be printed.

8. **Command>** `delete`
   - ERR: No City is selected.
   - (n) name, population and cost of selected city
Delete selected City: Are you sure? (y or n): y
Delete selected City: Are you sure? (y or n): n
ERR: Answer y or n.
Deletion not done.

(n) name, population and cost of selected city
Last City removed. No current selection.

Delete is the inverse of insert. After the user-friendly (or user-safe, or user-annoying) confirmation of y is given, the selected City is deleted, (all its storage must be recycled by use of C++ delete and delete[]. Naturally, the position of each City after the deleted one will then become one step less.

The City, if any, just after the selected City should become the new selected City.

The appropriate messages above should be printed for each situation. In case of errors or the “n” confirmation, the selected City should remain unchanged.

Finally, in all cases, if there is a selected City, it and its position should be printed (just as done for the print command.)

9. Command> swap <position number>
   Command> swap +
   Command> swap -
   ERR: No City is selected.
   ERR: Too few Cities.
   ERR: Trying to swap a thing with itself?? Waste of time!
   ERR: Cannot swap City before position 1.
   ERR: Missing or badly formatted position number.

The selected City and the City indicated by the swap command should be repositioned so their positions are interchanged. This should be implemented by changing all the values of the relevant pointer variables. They include variables within the swapped nodes, within their neighbors, and sometimes the the head_ptr and/or tail_ptr variables.

In the case of success, the selection should be the City indicated by the swap command. This means the selection index (position number) remains unchanged, but a different City is now at that position.

As with the other commands, the currently selected City, if any, with its position number is printed just before going on for the next command.

10. Command> cut
    ERR: No City is selected.
    One City added to the Clipboard.
    (n) name, population and cost of selected city
    Last City removed. No current selection.

This and the next few commands will use a second sequence of Cities called the “Clipboard”. Try to implement it with a second instance of Class CSequence
The *cut* command behaves like *delete* except (1) no confirmation is done and (2) instead of the selected City being deleted permanently, it is merely removed from the main sequence data structure and inserted into the Clipboard.

The “cut” City should be prepended to the Clipboard. The City that had been just after it in the main sequence, if any, should become the newly selected City. As usual, the newly selected City and its position should be printed.

11. **Command>printcuts**

   Print the sequence in the Clipboard in the same manner as the *printall* command prints the main sequence.

12. **Command>paste** (You figure out the error situations and choose messages consistent with the previous ones.)

   This requires a City be selected. If this is true and there no other errors, remove the City from the beginning of the Clipboard and insert it at the selected position. As with the *insert* command, the selected position remains unchanged but a different City is now in it.

13. Enhanced arguments to *select* and *swap*— Implement 3 new ways for these commands to target a City in the sequence. The enhanced argument will begin (like the original argument) one space after the *select* or *swap* token.

   - $ and $$: Target the cheapest (lowest cost) or highest cost City respectively. (Target any of them if there are multiple minima or maxima.)
   - `<Prefix Search string>`

     ERR: Line input failed(too long? end of file?)...exiting.
     ERR: Empty search string?

     If there are no errors, print the first City and its position number among those Cities whose names have the greatest length prefix in common with the the given `<Prefix Search String>`.

     You will have to write your own code to determine the length of the longest common prefix of two C-strings.

2   **How to Do This Project; Some Graded Items!**

   C-string literals of the exact (and other) strings that your program should print or recognize have been pre-typed for you. I also prepared sample compile and link commands for you to use to start your build.sh file. Copy them from `~acsi310/Proj2/messages.txt` or download (same thing??) them from

   http://www.cs.albany.edu/~sdc/CSI310/Proj/Proj2/messages.txt

   For design practice, you must design a class named *class CSequece* whose public member functions provide the core operations a list of *class City* instances. This class must implement one sequence of Cities using one doubly linked lists of nodes, where each node holds a pointer to a dynamically allocated City instance. The implementation’s data structure consists of private data
members of CSequence and dynamically allocated linked list nodes that hold pointers to Citys. The private data members must of course include head and tail pointers to the linked list. They should also include a pointer to implement the cursor. You might choose to include other private data members for such things as the number of Cities, etc.

It is up to you how much of the rest of the program is designed and implemented using classes and other object oriented practices. One choice is to make use of Main and Savitch’s “linked list toolkit”. Another (my preference for CS2 at Albany) is to use a struct for list nodes and let CSequence’s member functions manipulate node fields directly. For example,

```c
#include "City.h"
struct node {
    class City *pData;
    node *pNext;
    node *pPrev;
};
```

For simplicity, I think it would be a good idea to provide public functions of class CSequence to print using cout.

It is your design choice about whether class CSequence’s functions detect and/or print messages about user errors described below, or whether the CSequence should enforce preconditions and its caller detect and act on user errors. It is smart to think about what would happen with user errors before you commit much time into the design of CSequence.

The job of (an instance of) class CSequence is to hold the data being edited, carry out changes to it, and print (at your design option) copies of selected contents. Following this design requirement, class CSequence MUST NOT obtain input from cin or a file, NOR figure out commands by examining input strings! If the user interface language changes, the CSequence class must be reusable without change.

One critical design choice for CSequence is “Will CSequence call new to allocate space for each new City then store the name and coordinates given to it, or (alternatively) will the caller (probably the user interface module) call new and pass to CSequence the address of the new City?”

Another design choice involves which class will call delete[] on pointers to C-strings and call delete on pointers to Citys to recycle dynamic memory when a City is deleted.

Part of the project grade will come from your CSequence and City classes’ header files CSequence.h and City.h to document their interfaces in terms of pre and post conditions on their public function members. In addition to this documentation of the public function members, comments to document the data members (where the data structure begins), preferably written in the form of invariants, will also be graded.

I suggest you write the user interface in a “main” module much like the test driver of the previous project and Main/Savitch’s sample programs. This main module, or perhaps classes of your own design if you choose that route, but not of class CSequence, should analyze the user input commands! One main difference is that the user interface module should read in whole lines of input, not just single characters. It could use C-string library function strcmp(), described in DSO Chapter 4, to test if the string given by the user equals a legal command (see below).

1. All the software development practices given on the project 1 assignment must also be followed
for project 2 in order to earn full credit. Some details and additional requirements are given below.

2. Submission of the RCS revision history database for non-trivial files, showing your development process, is REQUIRED. It was ok (with points off) to omit it or start it late for the first project, but now, **PROJECT 2 (and the rest) will earn 0 points WITHOUT IT.** Right now is the time to learn basic RCS use, if you haven’t already.

3. Organization of source files must reflect the software’s design, or points will be taken off. In particular, at the very least, `CSequence.h`, `CSequence.cxx`; `City.h`, `City.cxx` and the the `.cxx` file containing the user interface code must be separate files. The file `CSequence.h` must document your `CSequence` class with pre and postconditions for all public functions. Similarly for `City.h` and class `City`. You may write the definition of `struct node` either in `CSequence.h` or in a separate header file.

4. An executable “build” script named `build.sh` is required for full credit. When run, the build script should compile and link the software to create an executable file named “`triplanner`” in the same directory.

   At this point, it would be wise to also write and test a “cleanup” script that automatically deletes any “core”, object or executable program files so that you can use it before submitting the project.

5. Submit a single directory to project name Proj2 using
   
   “`turnin-csi310 -c csi310 -p Proj2 Directory`”. Verify with
   
   “`turnin-csi310 -v -c csi310 -p Proj2`”.

   As before, points will be deducted if object, “core” or executable files are submitted.