CS515 – Lab 5

"His debugging skills are exceptional."

The objective of this lab is to give you practice implementing linked list data structures in C++ and work with the Set ADT.

Note: the second task in this lab can be used as the basis for assignment 6P—You'll be at a disadvantage if you skip it!

Download all files from `~cs515/public/5L`

Task 1. Jumble debug

Starter files needed in this task: `jumble.cpp charlist.h charlist.cpp input`

The `jumble.cpp` program randomly mixes the letters of a string to create a word jumble puzzle. The program will read a string and a list of 0s and 1s (in token format) as the jumble keys. The total number of 0s and 1s equals the length of the string. The program inserts each character into an initially empty list following this rule: if the key is 1, the character is inserted at the front of the list; if the key is 0, the character is inserted at the rear of the list. The program outputs a new string formed by all the characters in the jumbled list. For example, if the key were 1111 and the word were "**word**", then 'w', 'o', 'r', and 'd' would each be inserted to the front of the list one by one and the output string would be "**drow**". If the key were 1100, the jumbled word would be "**owrd**" since 'w' and 'o' are inserted to the front of the list, while 'r' and 'd' are inserted at the rear.

To compile and run the starter code, use this:

```
$ make jumble
$ ./jumble < input
```

After running, you would *expect* the program to output the following:

```
Input word and key to jumble: word is jumbled by key 1111 to drow
Input word and key to jumble: word is jumbled by key 1110 to rowd
Input word and key to jumble: word is jumbled by key 1100 to owrd
Input word and key to jumble: word is jumbled by key 1000 to word
Input word and key to jumble: word is jumbled by key 0011 to drwo
Input word and key to jumble: word is jumbled by key 0101 to dowr
Input word and key to jumble: word is jumbled by key 0000 to word
Input word and key to jumble:  ^D
```

Unfortunately, there are some bugs in `charlist.cpp`, and you will get some funny output followed by a segmentation fault. Your task is to locate and fix the bugs so the jumble program runs as expected. You also need to make sure the program doesn't contain memory leaks.

You should only modify the `charlist.cpp` file. You must **not** change the header `charlist.h` or the game driver `jumble.cpp`.

Task 2. Set ADT

Starter files needed in this task: `set.h settest.cpp`

You will implement a Set ADT that behaves like a mathematical set that does not contain duplicate items. A Set can be viewed as a container such that items can be inserted into and removed from the container. However, the items cannot be modified once in the container. The Sets will contain items that can be compared, and items are kept sorted in order from least to greatest. The set ADT supports the following operations.

void **insert**(element): insert an element into the set.
void **erase**(element): remove an element from the set.
void **clear()**: remove all elements from the set. The set is empty but not destroyed.
bool **find**(element): find an element from the set; return true if found.
int **size():** return the size of current elements in the set

The Set also has the following overloaded operators:

- Logical equality (==): two sets are equal if they contain the same elements.
- Insertion operator (<<): prints all set elements to the output stream in ascending order. One single space is used to separate the elements.

The Set ADT can be implemented with various concrete data structures. In this lab, you must use a linked-list is to implement the Set. Keep in mind the linked-list implementation is not time efficient since the search operation takes ***O***(***n***) for both sorted and unsorted linked lists. In the future, we will use balanced search trees for more efficient implementations.

You are provided with starter code for a partial implementation of the set ADT using a doubly linked list with dummy head and tail nodes (sentinels). In a doubly-linked list, every node is connected with its next and previous nodes, thus we can traverse the list backwards and forwards easily. Figures 1-3 show some examples of such a list (note that they don't properly insert numbers in order).

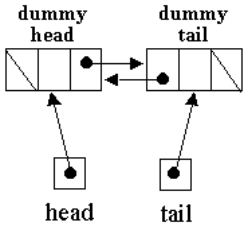

Figure 1*. An empty list*

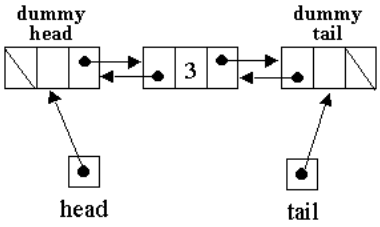

Figure 2*. A list with one node*

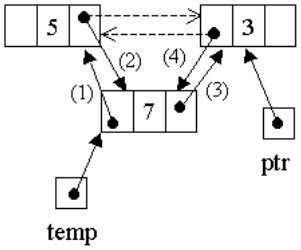

Figure 3. *Insert node 7 after 5*

It is assumed that the elements to be inserted into the set are only integers as specified with `typedef` in the `set.h` file. You need to finish the implementation of the ADT in a file named `set.cpp`. You may modify the header file `set.h`. However, you should **not** change the public interface of the Set ADT, which means you cannot add any new public methods. However, you are allowed to add new private methods or declare friends for the class. Also note that since the `set` class contains a pointer field that points to dynamic memory, you need to provide a constructor and at least 3 of the Big-5 methods for the class: copy constructor, destructor, and overloaded assignment operator.

Once you finish, you should be able to run the test driver `settest.cpp` with your completed implementation. You need to write additional test files to test all methods in your implementation. All valgrind errors need to be checked.

You are not allowed to use any STL containers in your implementation.

We are no longer providing you with the tests we will be using when we grade. You are now responsible for coming up with your own tests to ensure you have properly implemented your class. Note that you should not submit your tests, just use them in the process of writing your code.

Submission:

Submit the source files **`charlist.h`**, **`charlist.cpp`**, **`set.h`**, and **`set.cpp`**. You may change the provided `makefile` locally, but we will be testing with our own—be sure your `makefile` has something like the following lines to ensure proper compilation with our grading test cases.

```
all: charlist.o set.o

charlist.o: charlist.cpp charlist.h
	$(CXX) -c charlist.cpp

set.o: set.cpp set.h
	$(CXX) -c set.cpp
```

You should also fill out the README file and submit it as well. To submit the files in Mimir, follow the link for this lab in MyCourses. Here is a list of files you are expected to submit:

`charlist.h charlist.cpp set.h set.cpp README`

As always, do not turn in executables or object code, and make sure your submission compiles successfully on Agate. Programs that fail to compile will receive a zero mark (even if it might work perfectly on your home computer.) To check this, use the make command with the provided makefile, and check that your submission passes tests on Mimir.

Important:

You must include the standard comment block in each of your source files, including your name, section, date and collaboration details. You must also finish the README file along with your programs.

You should also include detailed collaboration declaration information in your comments. If you worked in pairs in this lab, each of you must include the partner's name in your program comment. Both you and your partner must complete an individual submission in order to earn a grade. If you work by yourself, you must indicate in the program comments that you have worked on your own independently.

You can resubmit as needed—your last submission will be graded. Here is a tentative grading scheme:

jumble test run 1	10
jumble test run 2	10
jumble Valgrind check	5
set test run 1	15
set test run 2	15
set test run 3	15
set test run 4	15
set test Valgrind checks	15