COMP2521 19T1 Data Structures and Algorithms

Week 01 Lab Exercise

Linked Lists, Performance

Objectives

- to re-acquaint you with C programming and ADTs
- to manipulate a linked-list data structure
- to learn about the COMP2521 programming style
- to learn (or remind yourself) about Makefiles
- to learn about shell scripts to automate repetitive tasks
- to do some simple performance analysis

Admin

Marks	5=outstanding, 4=very good, 3=adequate, 2=sub-standard, 1=hopeless
Demo	in the Week01 Lab or in the Week02 Lab
Submit	`give cs2521 lab01 IntList.c timing.txt` or via WebCMS
Deadline	must be submitted by 11:59pm Sunday of Week-01 (24/Feb)

Note: you need to do something *truly* outstanding, above and beyond the "call of duty" to get 5 marks. Doing the exercise just as specified will generally get you 4-4.5 marks.

Background

At the end of COMP1511, you dealt with linked lists. Over the break, you haven't forgotten linked lists (have you?), but a bit of revision never hurts, especially when many of the data structures we'll deal with later are based on linked lists. So ... on with this simple linked list exercise ...

Setting Up

To keep your files manageable, it's worth doing each lab exercise in a separate directory (folder). I'd suggest creating a subdirectory in your home directory called "cs2521", and then creating a subdirectory under that called "labs", and then subdirectories "week01", "week02", etc. Let's assume that the directory you finally set up for this lab is *Week01LabDir*.

Change into your *Week01LabDir* directory and run the following command:

```
$ unzip /home/cs2521/web/19T1/labs/week01/lab.zip
```

If you're working at home, download `lab.zip` by right-clicking on the above link and then run the above command on your local machine.

> In the example interactions, we assume that you are in a Linux shell window, and the shell is giving you a `$` prompt. All the text that you type is in **`bold`** font and all the text that the shell types at you is in `normal` font.

If you've done the above correctly, you should now find the following files in the directory:

`Makefile`	a set of dependencies used to control compilation

`IntList.h`	interface definition for the IntList ADT
`IntList.c`	implementation for the IntList ADT
`useIntList.c`	main program for testing the IntList ADT
`randList.c`	main program for generating random sequences of numbers
`timing.txt`	template for your results file; you need to add more rows

Before you start using these programs, it's worth looking at the code. Are there any constructs that you don't understand? Try to work them out with your lab partner, or ask your tutor.

Once you've understood the programs, the next thing to do is to run the command:

```
$ make
```

It's worth taking a look at the `Makefile` to see if you can work out what it's doing. Don't worry if you don't understand it all; we'll be taking a longer look at `make` in later labs. Note: you will need to run `make` to recompile the system each time you make changes to the source code file and are ready to test the program again.

The `make` command will produce messages about compiling with `gcc` and will eventually leave two executable files in your directory (along with some `.o` files).

`useIntList`
: This is the executable for the `useIntList.c` program. It reads a list of integers from standard input, then attempts to make a sorted (ascending order) copy of the list, and then prints both the original and the sorted lists. It doesn't work at the moment because the function to produce the sorted list is incomplete.

`randList`
: This is the executable for the `randList.c` program. It writes, on its standard output, a list of random numbers in the range 1..999999. It takes one command-line argument to indicate how many numbers it should write, and another optional argument to give a seed for the random number generator. Note that it does not attempt to eliminate any duplicate values produced; if you generate a large enough number of values, duplicates are inevitable.

You can run the `useIntList` command by typing the command name, followed by return. It will then patiently wait for you to type some numbers, and will store them in a list, display the list, then call the function that is supposed to sort the list, and then display the "sorted" list. Of course, this doesn't currently work, because the list sorting function is incomplete. You can, however, fake it by typing the numbers in in order, e.g.

```
$ ./useIntList -v
1 2 3 4
Here you type control-D to finish your input
Original:
1
2
3
4
Sorted:
1
2
3
4
```

When `useIntList` eventually works properly, this is the kind of output you'll expect to see ... except that you

won't be giving it sorted lists as input. To see the existing behaviour on an unsorted list, type a few numbers not sorted in ascending order, e.g.

```
$ ./useIntList -v
1 3 2
Once again, you type control-D to finish your input
Original:
1
3
2
Sorted
1
3
2
useIntList: useIntList.c:20: main: Assertion `IntListIsSorted(myOtherList)' failed.
Aborted
```

If you omit the `-v` command-line parameter, the the `useIntList` program only displays the final sorted list (which is not yet sorted, of course).

```
$ ./useIntList
1 3 2
Once again, you type control-D to finish your input
1
3
2
useIntList: useIntList.c:20: main: Assertion `IntListIsSorted(myOtherList)' failed.
Aborted
```

Now it's time to play with the list generator. If you execute the command:

```
$ /.randList 10
```

it should display a list of 10 random numbers. If you run it again, you'll get a different list of random numbers. Enjoy generating small lists of random numbers until you are bored.

If you then execute the command:

```
$ ./randList 10 | ./useIntList
```

the `randList` command will generate 10 random numbers and give them to the `useIntList` command which will print the list of numbers twice (separated by the word `Sorted`) and then fail the assertion as above.

> If you're not familiar with Unix/Linux conventions, the `|` is a "pipe" that connects two commands, e.g. $C_1 | C_2$. The standard output (`stdout`) of the command C_1 will be directly connected to the standard input (`stdin`) of the command C_2, so that whatever C_1 writes, C_2 reads.

You can adjust the number of numbers that `randList` generates via the command-line argument, e.g.

```
$ ./randList 100 | less
```

will produce 100 random numbers. If you want to change the range of numbers produced (e.g. to only make numbers in the range 1..10) edit the `randList.c` code and add extra command-line arguments to set the range.

If you supply only one command-line argument, `randList` will generate a completely random (well, pseudo-

random) sequence each time you run it. If you want to generate a large sequence of pseudo-random numbers, and be able to generate the same sequence consistently, you can use the second command-line argument to specify a seed for the random number generator. If you give the same seed each time you run `randList`, you'll get the same sequence each time. To see the difference between using and not using the second command-line argument, try the following commands:

```
$ ./randList 10
$ ./randList 10
$ ./randList 10
$ ./randList 10 13
$ ./randList 10 13
$ ./randList 10 13
```

The first three commands will generate different sequences. The second three commands all generate the same sequence.

While `./randList` is useful for producing large amounts of data, you don't need to use `./randList` to produce input to `./useIntList`. An alternative is to use the `echo` command and enter the numbers yourself, e.g.

```
$ echo 5 4 3 2 1 | ./useIntList
```

Another alternative, which will be more useful for testing, is to use the `seq` command, in combination with the `sort` command, e.g.

```
$ seq 10
# gives 1 2 3 4 5 6 7 8 9 10
$ seq 10 | sort -n
# gives 1 2 3 4 5 6 7 8 9 10
$ seq 10 | sort -nr
# gives 10 9 8 7 6 5 4 3 2 1
$ seq 10 | sort -R
# gives 1..10 in a random order
```

The `-nr` argument to the `sort` command tells it to treat the input as numbers and sort them in reverse order. The `-R` argument to the `sort` command tells it to put the input in random order. You can find more details about `sort` via the command `man sort`.

One thing to remember is that `sort` orders items lexically, not numerically, by default. The following pipeline may not produce what you expect. Try it.

```
$ seq 10 | sort
```

Task 1

The `IntListInsertInorder()` function is incomplete. In fact, it's just a *stub* function that invokes the `IntListInsert()` function, which inserts the number, but not in order. You should re-write the `IntListInsertInorder()` function so that it takes an `IntList` and an integer and inserts the value into the appropriate place in the list, so that the list always remains sorted (in ascending order). The function should fail if the original list (before insertion) is not sorted. Don't forget to handle the cases of (a) empty list, (b) smallest value, (c) largest value, (d) second-smallest value, (e) second-largest value, (f) value somewhere in the middle. Why were these kinds of values chosen? Discuss with your partner and propose test cases that test these and any other cases that you can think of.

Make a directory called `tests`, and place files containing your test cases in that directory with one test case in

each file. A useful name strategy is to call the test files `01`, `02`, etc. You could create test files in various ways, e.g.

```
$ mkdir tests
$ seq 10 | sort -R > tests/01
$ seq 10 | sort -nr > tests/02
$ randList 10 11 > tests/03
etc. etc. etc.
```

In order to check that your program is producing the correct results, you could compare it to the output of a known correct sorting program. The `diff` command can help here (see `man diff` for details). For example, you could put each test case in a file, then run both your `useIntList` program and the built-in `sort` command, save the results of each output in a file, and then compare the files using `diff`. If your program is correct, there should be no difference. The following shows an example of how to do this:

```
$ sort -n < tests/01 > tests/01.expected    # generate correct result
$ ./useIntList < tests/01 > tests/01.observed     # generate *your* result
$ diff tests/01.expected tests/01.observed  # if correct, no output
```

If you produce a decent number of tests (10-20), as you should, then testing them one by one using the above is a bit tedious. You could simplify carrying out the testing by writing a small shell script like:

```
#!/bin/sh

for t in 01 02 03 04 05 ... and the rest of your test files
do
        echo === Test $t ===
        sort -n < tests/$t > tests/$t.expected
        ./useIntList < tests/$t > tests/$t.observed
        diff tests/$t.expected tests/$t.observed
done
rm tests/*.expected tests/*.observed
```

If you put the above in a file called e.g. `run_tests`, and then run the command:

$ **sh run_tests**

the script will run all your test cases and any that fail will either produce an assertion message or show you the difference between your output and the expected output.

You will have some confidence that your `IntListInsertInorder()` function is working properly when, for all test cases, the assertions in `useIntList.c` no longer fail.

Task 2

Once `IntListInsertInorder()` is working correctly, modify `useIntList.c` so that it behaves as a sorting program: it reads a list of numbers from its standard input, and writes a sorted version of the numbers to its standard output. This is a simple change to the existing program, and can be accomplished by commenting a couple of lines.

Recompile and check that the new version of `useIntList` behaves correctly. Once you're satisfied, run some tests to compare the time taken by `useIntList` with the time taken by the Unix `sort` command for the same task.

The first thing to do is to check that both commands are producing the same result (otherwise, it's not useful to compare them). Try the following commands:

```
$ ./randList 1000 > nums
$ ./useIntList < nums > out1
$ sort -n < nums > out2
$ diff out1 out2
```

If the `diff` command gives *no* output, then the files have no difference (i.e. the observed output is he same as the expected output). If you try the above a number of times with different sets of numbers each time and get the same result, then you can be more confident that both commands are producing the same result. (However, you cannot be absolutely certain that they will always produce the same result for any input. Why not?)

Note that you need to use the `-n` option to `sort` because the default sorting order is lexcial, not numeric. If you don't use `-n`, then e.g. 111 is treated as being less than 12. (See `man sort` for details.)

The next thing to do is to devise a set of test cases and collect some timing data. There is a command called `time` that will produce the timing data for you. Use it as follows:

```
$ ./randList 100000 > nums
$ time ./useIntList < nums > out1
$ time sort -n < nums > out2
```

As before, check that both commands are producing the same result. This time, however, you will also see output that contains some timing information (note that the output format may vary depending on which shell you're using):

user time	time spent executing the code of the command
system time	time spent executing system operations, such as input/output
elapsed/real time	wall-clock time between when the command starts and finishes

The elapsed/real time is affected by the load on the machine, and so is not reliable. The system time should be similar for both commands (same amount of input/output). The value that's most useful is the `user time`, which represents the time that the program spent doing computation. Note that the time will vary if you run the same command multiple times. This is because the timing uses sampling and may sample the program at different points during the different executions. Collect results from a number of executions on the same output and take an average.

You will need to use large numbers of values (large argument to `randList`) to observe any appreciable difference. Of course, if you use very large numbers of values, the `nums` file may become too large to store in your directory. In this case, you could store `nums` under the `/tmp` directory:

```
$ ./randList 10000 > /tmp/nums
$ time ./useIntList < /tmp/nums > /tmp/out1
$ time sort -n < /tmp/nums > /tmp/out2
$ diff /tmp/out1 /tmp/out2
```

It would also be worth taking your `/tmp/nums` file and modifying it to check the timings for sorted and reverse sorted inputs:

```
$ sort -n /tmp/nums > alreadySortedWithDuplicates
$ sort -nr /tmp/nums > reverseSortedWithDuplicates
```

As well as lists produced by `randList`, it would be worth trying lists produced using the `seq`/`sort` combinations mentioned above. These will be different to the lists produced by `randList` in that they won't have any duplicate values. For example, you could generate data files like:

```
$ seq 10000 > alreadySortedNoDuplicates
$ seq 10000 | sort -nr > reverseSortedNoDuplicates
$ seq 10000 | sort -R > randomOrderNoDuplicates
```

and use these in place of `/tmp/nums` to generate more timing data.

Once the data files get big enough, and once you've checked that both programs are producing the same output for a wide selection of cases, you can avoid generating and comparing large output files by running the timing commands as follows:

```
$ time ./useIntList < /tmp/nums > /dev/null
$ time sort -n < /tmp/nums > /dev/null
```

The above commands do all of the computation that we want to measure, but send their very large output to the Unix/Linux "data sink" (`/dev/null`), so it never takes up space on the file system.

If you do write results to files, don't forget to remove them after you've finished the lab.

You should take note of the timing output and build a collection of test results for different sizes/types of input and determine roughly the point at which it becomes impractical to use `useIntList` as a sort program (rather than using the Unix `sort` command).

You'll find that the timing data for relatively small lists (e.g. up to 10000) doesn't show much difference between `useIntList` and `sort`, and shows time pretty close to zero. Try using list sizes such as 5000, 10000, 20000, 50000, 100000, and as high as you dare to go after that. Also, try varying the kinds of input that you supply. Consider the cases for the order of the input: random, already sorted, reverse sorted. Also, consider the proportion of duplicate values in the input, e.g. all distinct values, some duplicates. Note that `./randList` will most likely include duplicates as soon as you specify a relatively large list size.

Put your results and some brief comments explaining the results, in a file called `timing.txt`. The file should contain a table of results, with major rows dealing with a particular size of data, and sub-rows dealing with the order of the input. You should have three columns: one to indicate how many runs of each program were used to compute the average time-cost, one to give the average time-cost for `useIntList` and the other to give the average time-cost for `sort`. The table should look something like the following:

Input Size	Initial Order	Has Duplicates	Number of runs	Avg Time for useIntList	Avg Time for sort
5000	random	no	*N*	T_1sec	T_2sec
5000	sorted	no	*N*	T_1sec	T_2sec
5000	reverse	no	*N*	T_1sec	T_2sec
5000	random	yes	*N*	T_1sec	T_2sec
5000	sorted	yes	*N*	T_1sec	T_2sec
5000	reverse	yes	*N*	T_1sec	T_2sec
10000	random	no	*N*	T_1sec	T_2sec
10000	sorted	no	*N*	T_1sec	T_2sec
10000	reverse	no	*N*	T_1sec	T_2sec
10000	random	yes	*N*	T_1sec	T_2sec
10000	sorted	yes	*N*	T_1sec	T_2sec
10000	reverse	yes	*N*	T_1sec	T_2sec
etc. etc.					

Note that, given the variation in timing output, it's not worth considering more than two significant figures in your averages.

You should also write a short paragraph to *explain* any patterns that you notice in the timing results. Don't just re-state the timing result in words; try to explain *why* they happened.

If you're looking for a challenge, and know how to write scripts, write e.g. a Linux shell script to automate the testing for you, and maybe even produce the timing file for you.

Another interesting challenge would be to plot graphs using the R system, available as the Linux command `R` (a single upper-case letter). I'd suggest plotting a separate graph for each type (random, ascending, descending) for a range of values (e.g. 5000, 10000, 20000, 50000, ... as far as you can be bothered, given how long larger data files take to sort). Alternatively, produce your `timing.txt` as a tab-separated file and load it into a spreadsheet, from where you could also produce graphs.

If you're feeling brave, try to find out how large input `sort` can deal with. However, you should try such testing on a machine where you are the sole user. Also, you shouldn't try to store the data files; use commands like:

```
$ seq SomeVeryLargeNumber | time sort -n > /dev/null
$ seq SomeVeryLargeNumber | sort -nr  | time sort -n > /dev/null
$ seq SomeVeryLargeNumber | sort -R | time sort -n > /dev/null
```

Submission

You need to submit two files: `IntList.c` and `timing.txt`. You can submit these via the command line using `give` or you can submit them from within WebCMS. After submitting them (either in this lab class or the next) show your tutor, who'll give you feedback on your coding style, your timing results, and award a mark.

Have fun, *jas*