Week 01 Lab Exercise

Linked Lists

Objectives

- To re-acquaint you with C programming
- To manipulate a linked list data structure

Admin

Marks	5 (see the Assessment section for more details)
Demo	in the Week 1, 2 or 3 lab session
Submit	see the Submission section
Deadline to submit to give	5pm Monday of Week 2
Late penalty	0.2% per hour or part thereof, submissions later than 5 days not accepted

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

or "cs2521", and then creating a subdirectory under that called "labs", and then subdirectories "lab01", "lab02", etc.

Create a directory for this lab, change into it, and run the following command:

```
$ unzip /web/cs2521/23T2/labs/week01/downloads/files.zip
```

If you're working at home, download `files.zip` by clicking on the above link and then run the `unzip` command on the downloaded file.

NOTE:

In the example interactions, we assume that you are typing at a terminal emulator running a shell, and the shell is giving you a `$` prompt. All the text that you type is in **`monospace bold`**; and all the text that is printed to you is in `monospace`.

If you've done the above correctly, you should now have the following files:

`Makefile`	a set of dependencies used to control compilation
`IntList.h`	interface definition for the IntList ADT
`IntList.c`	implementation of the IntList ADT (incomplete)
`testIntList.c`	a program for testing the IntList ADT
`sortIntList.c`	a program that uses the IntList ADT to sort a list of numbers

Before you start using these programs, it's worth looking at the code. If there are any constructs you don't understand, ask your tutor.

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

```
$ make
```

Don't worry if you don't understand the `Makefile`; we'll be taking a closer look at `make` in lectures. Note: you will need to run `make` to recompile the program each time you make changes to the code.

The `make` command will produce messages which show the commands it is running, and will eventually leave two executable files in your working directory (along with some `.o` files):

`testIntList`

keeping the list ordered), and then prints out the resulting list. It doesn't work at the moment because the code to insert the integer is incomplete. Here is an example interaction with the program:

```
$ ./testIntList
Enter some numbers (must be in ascending order): 2 3 5 7
Enter number to insert: 4

Original list:
2
3
5
7
List after inserting 4:
2
3
5
7
```

Note that the program in its current state does not actually insert the number. Once you've got the program working, it should behave like so:

```
$ ./testIntList
Enter some numbers (must be in ascending order): 2 3 5 7
Enter number to insert: 4

Original list:
2
3
5
7
List after inserting 4:
2
3
4
5
7
```

sortIntList

This is the executable for the `sortIntList.c` program. It reads a list of integers (which can be unordered) from standard input, then prints that list, then attempts to make a sorted (in ascending order) copy of the list, and then prints the sorted list. It doesn't work at the moment because the code to produce the sorted list is incomplete. Here is an example interaction with the program (once it is implemented correctly):

```
Original:
3
7
2
5
Sorted:
2
3
5
7
```

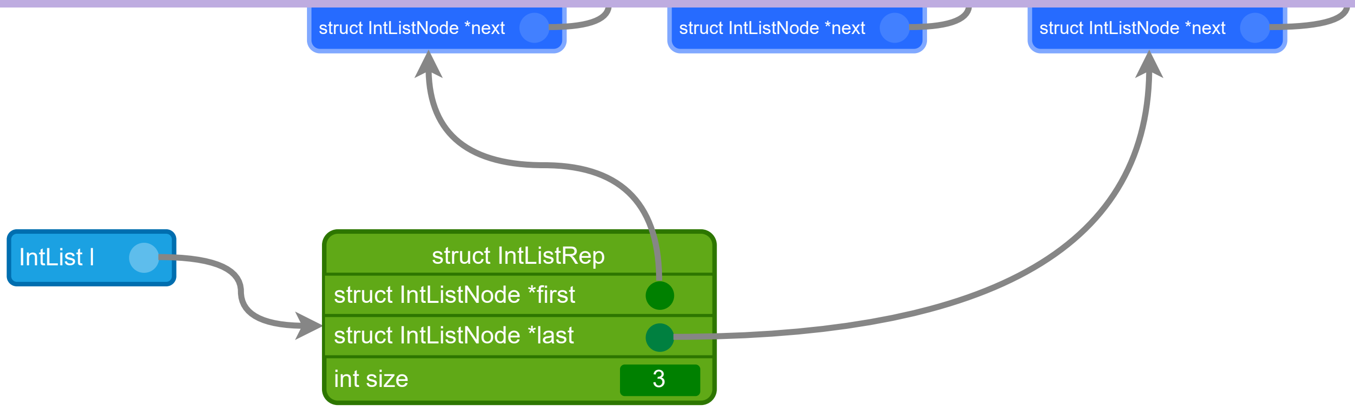
If you omit the `-v` command-line parameter (the v stands for verbose), the `sortIntList` program will only display the final sorted list.

```
$ ./sortIntList
3 7 2 5
Ctrl-D
2
3
5
7
```

Task 1

Implement the `IntListInsertInOrder()` function in `IntList.c`, which takes an `IntList` and an integer and inserts the integer into the appropriate place in the list, so that the list remains sorted (in ascending order). The function can assume that the given list is sorted (and you don't need to check that it is sorted). The function should accept and insert duplicate values. You'll need to handle a number of different cases, such as: (a) empty list, (b) smallest value, (c) largest value, (d) value somewhere in the middle. Discuss with your classmates and propose test cases that test these and any other cases that you can think of.

To help you visualise the problem, here is a diagram of an `IntList` that contains some items:



When you think you're done, use the `make` command to compile the `testIntList` program, and then run it to test your code. Run it multiple times to test different cases. For example, does it work when the original list is empty? (To test for an empty list, just press enter when you are prompted for the list.)

Here's an example test run:

```
$ ./testIntList
Enter some numbers (must be in ascending order): 2 3 5
Enter number to insert: 4

Original list:
2
3
5
List after inserting 4:
2
3
4
5
```

The `IntListRep` struct contains multiple fields that must be kept up to date as the list is changed. If you don't maintain all of these fields correctly, even if you inserted the number into the right place, a loud error message will be printed, and you should go back and fix your code.

```
$ ./testIntList
Enter some numbers (must be in ascending order): 2 3 5
Enter number to insert: 4

Original list:
2
3
5
```

```
4
5

#####  ####   ####   #####  ####
#      #   #  #   #  #   #  #   #
####   ####   ####   #   #  ####
#      #  #   #  #   #   #  #  #
#####  #   #  #   #  #####  #   #

error: IntListOK returned false, which means the list was not updated correctly. Please see
the IntListOK function for details.
```

Try to be thorough with your testing and test as many cases as you can think of, as the next task relies on `IntListInsertInOrder` working.

Task 2

Implement the `IntListSortedCopy()` function in `IntList.c`, which takes an `IntList` (which may be unordered), and returns an sorted copy of the list. **You must use the `IntListInsertInOrder()` function that you implemented in Task 1.**

HINT:

Recall from Task 1 that `IntListInsertInOrder()` assumes that the given list is sorted, and ensures that the list remains sorted (if you've implemented it correctly).

When you think you're done, use the `make` command to recompile the `sortIntList` program, and then run it to test your code. Here's an example test run:

```
$ ./sortIntList
3 7 2 5
Ctrl-D
2
3
5
7
```

files containing your test cases in that directory with one test case in each file. A useful naming strategy is to call the test files `01`, `02`, etc. Here's an example test file which corresponds to the test run above:

```
3
7
2
5
```

WARNING:

Make sure your test files contain exactly one integer per line and don't contain any unnecessary whitespace (such as leading/trailing spaces). Also make sure your test files do not contain any blank lines at the end.

Now we can run tests by making the `sortIntList` program read from our test files instead of from the terminal. For example:

```
$ ./sortIntList < tests/01
2
3
5
7
```

To check that your program is producing the correct results (without eyeballing the output every time), you can compare it to the output of a known correct sorting program. For example, you could run both your `sortIntList` and the built-in `sort` command on a test case, and then compare the results using the `diff` command (see `man diff` for details). If your program is correct, `diff` will produce no output, as there will be no difference. The following shows an example of how to do this:

```
$ sort -n < tests/01 > tests/01.exp          # generate correct result
$ ./sortIntList < tests/01 > tests/01.out    # generate *your* result
$ diff -Bb tests/01.exp tests/01.out    # if correct, no output
```

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

```
#!/bin/sh

for t in 01 02 03 04 05 # TODO: add the rest of your test files to this list
```

```
    ./sortIntList < tests/$t > tests/$t.out 2>&1

    if diff -Bb tests/$t.exp tests/$t.out > /dev/null 2>&1
    then
        echo "Passed"
    else
        echo "Failed - check differences between tests/$t.exp and tests/$t.out"
    fi
done
```

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

```
$ sh run_tests
```

The script will run all your test cases and tell you whether each test case passed or failed. If any of your tests fail, then you should go back and fix your code. Note that bugs could reside in either `IntListInsertInOrder` or `IntListSortedCopy` - you'll need to check both functions.

NOTE:

If you get error messages like:

```
$'\r': command not found
syntax error near unexpected token `$'do\r''
```

then run the command `dos2unix` *`script-name`* (where *`script-name`* is the name of the script) and try again.

Submission

You need to submit one file: `IntList.c`. You can submit via the command line using the `give` command:

```
$ give cs2521 lab01 IntList.c
```

You can also submit via give's web interface. You can submit multiple times. Only your last submission will be marked. You can check the files you have submitted here.

After you submit, you **must** check that your submission was successful by going to your submissions page. Check that the timestamp is correct. If your submission does not appear under Last Submission or the timestamp is not correct, then resubmit.

Assessment

Most of the marks for this lab will come from automarking. To receive the rest of the marks, you *must* show your work to your tutor during your Week 1, 2 or 3 lab session. You will be marked based on the following criteria:

Code correctness (4 marks)
These marks will come from automarking. Automarking will be run after submissions have closed. After automarking is run you will be able to view your results here.

Code style (1 mark)
Code with good style should have these qualities: consistent indentation and spacing, no repetition of code, no overly complicated logic, no overly long functions, correct use of C constructs (such as `if` statements and `while` loops), and comments where appropriate. See the style guide.

COMP2521 23T2: Data Structures and Algorithms is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs2521@cse.unsw.edu.au
CRICOS Provider 00098G

Week 02 Lab Exercise

Recursion

Objectives

- To practice recursion
- To practice using linked lists
- To practice defining recursive helper functions

Admin

Marks	5 (see the Assessment section for more details)
Demo	no demo required
Submit	see the Submission section
Deadline to submit to give	5pm Monday of Week 3
Late penalty	0.2% per hour or part thereof, submissions later than 5 days not accepted

Setting Up

Create a directory for this lab, change into it, and run the following command:

```
$ unzip /web/cs2521/23T2/labs/week02/downloads/files.zip
```

If you're working at home, download `files.zip` by clicking on the above link and then unzip the downloaded file.

Task 1 - GCD

The greatest common divisor, or GCD, of two integers a and b is the largest integer that divides both a and b with no remainder. For example, the GCD of 16 and 28 is 4 because $16 = 4 \times 4$ and $28 = 4 \times 7$, and there is no larger integer than divides both.

One way to calculate the GCD would be to totally factor both numbers and find common factors, but there is a much faster and easier way to do it.

If r is the remainder when we divide a by b, then the common divisors of a and b are precisely the same as the common divisors of b and r, so the following identity holds:

$$\gcd(a, b) = \gcd(b, r)$$

If we start with any two positive integers, and apply this identity repeatedly, r will eventually become zero, and the other number in the pair is the greatest common divisor.

This is an amazing method known as Euclid's algorithm, and is probably the oldest known non-trivial algorithm; it was first described in Euclid's *Elements* in around 300 BC.

Your task is to implement the following function in `gcd.c`:

```
int gcd(int a, int b);
```

This function should find the GCD of two integers using Euclid's algorithm as described above. You can assume that `a` and `b` are non-negative, and that at most one of them is 0.

Note: You must use recursion. A non-recursive solution will not receive any marks.

Note: We have provided hints for some of the tasks in this lab, which can be found in the appendix at the bottom of this page. Only look at them if you're stuck and have **no** ideas - if you have an idea but you're not sure if it will work, try it first!

Testing

`gcd.c` contains a main function which allows you to test your `gcd` function. The main function takes two command-line arguments which are the two integers. Here are some

```
...
$ ./gcd 16 28
The GCD of 16 and 28 is 4
$ ./gcd 25 15
The GCD of 25 and 15 is 5
$ ./gcd 12 72
The GCD of 12 and 72 is 12
$ ./gcd 64 25
The GCD of 64 and 25 is 1
$ ./gcd 0 42
The GCD of 0 and 42 is 42
```

Task 2 - A Plague of Rabbits

I have a terrible rabbit problem.

I used to have a pair of baby rabbits; they were extremely cute and fluffy, so of course I got them. But the shopkeeper I got them from - a guy named Leonardo, of Pisa Pets - didn't tell me they would mature very fast, and breed even faster.

After a month, I had a mature pair of rabbits... and, of course, they bred. Damn.

So, a month later, I had a pair of adults and a pair of baby rabbits.

And a month later, I had two pairs of adults, and another pair of baby rabbits.

And a month later, I had three pairs of adults, and two pairs of baby rabbits.

And a month later, I had five pairs of adults, and three pairs of baby rabbits.

HELP! I HAVE SO MANY RABBITS, I'M GOING HOPPING MAD!

Can you help me figure out how many rabbits I'll have? Given that I started with one pair of baby rabbits, implement the following function in `rabbits.c` that tells me how many rabbits I'll have after a given number of months.

```
long long rabbits(int months);
```

You can assume I won't ask about any time longer than 60 months - surely the rabbits will have taken over the world by then...

A `long long` is like an `int`, but is 8 bytes in size, so it can store larger values than

Note: You must use recursion. A non-recursive solution will not receive any marks.

Testing

`rabbits.c` contains a main function which allows you to test your `rabbits` function. The main function accepts one command-line argument which is the number of months. Here are some examples of its usage:

```
$ make
...
$ ./rabbits 0
Number of rabbits after 0 month(s): 2
$ ./rabbits 1
Number of rabbits after 1 month(s): 2
$ ./rabbits 2
Number of rabbits after 2 month(s): 4
$ ./rabbits 12
Number of rabbits after 12 month(s): 466
$ ./rabbits 42
... after a long pause ...
Number of rabbits after 42 month(s): 866988874
$ ./rabbits 60
... after several hours ...
Number of rabbits after 60 month(s): 5009461563922
```

Task 3 - Find the Last Element of a Linked List

Recursion works really well with linked lists, because a linked list can be defined recursively as follows:

A linked list is either:

- Empty, or
- A node containing a value followed by a (smaller) linked list

This means we usually have a base case where the node pointer is `NULL`, and a recursive case where we do something with the current value and recursively call the function on

Your task is to implement this function in `listTail.c`:

```
int listTail(struct node *list);
```

This function should use recursion to find the last value in the given list. You can assume that the list is not empty.

Note: You must use recursion. A non-recursive solution will not receive any marks.

Testing

`listTail.c` contains a main function which allows you to test your `listTail` function.

The main function:

- Reads the size of the list
- Reads the values of the list
- Displays the list
- Calls `listTail`
- Displays the result

Here are some examples of its usage:

```
$ make
...
$ ./listTail
Enter list size: 7
Enter list values: 6 8 9 2 5 1 3
List: [6, 8, 9, 2, 5, 1, 3]
The last element is: 3
$ ./listTail
Enter list size: 4
Enter list values: 2 5 2 1
List: [2, 5, 2, 1]
The last element is: 1
$ ./listTail
Enter list size: 1
Enter list values: 42
List: [42]
The last element is: 42
```

Your task is to implement this function in `listMax.c`:

```
int listMax(struct node *list);
```

This function should use recursion to find the largest value in the given list. You can assume that the list is not empty.

Note: You must use recursion. A non-recursive solution will not receive any marks.

Testing

`listMax.c` contains a main function which allows you to test your `listMax` function. The main function:

- Reads the size of the list
- Reads the values of the list
- Displays the list
- Calls `listMax`
- Displays the result

Here are some examples of its usage:

```
$ make
...
$ ./listMax
Enter list size: 5
Enter list values: 7 2 6 8 0
List: [7, 2, 6, 8, 0]
The maximum element is: 8
$ ./listMax
Enter list size: 5
Enter list values: 9 6 1 8 8
List: [9, 6, 1, 8, 8]
The maximum element is: 9
$ ./listMax
Enter list size: 4
Enter list values: 2 5 2 1
List: [2, 5, 2, 1]
The maximum element is: 5
$ ./listMax
Enter list size: 1
Enter list values: 42
```

Task 5 - Using Recursive Helper Functions

Often in this course, a list will be represented by two structures, one for the usual list node, and one which contains a pointer to the head of the list (along with other data about the list such as its size), usually called a wrapper or container struct. In this case, since we want to recurse on the nodes, not the wrapper struct, we need to implement a helper function which takes in a node pointer and then call it from the original function. For example:

```c
int listFunc(struct list *list) {
    return listFuncHelper(list->head);
}
```

Your task is to implement this function in `listSum.c`:

```c
int listSum(struct list *list);
```

This function should use a recursive helper function that takes in a node pointer to find the sum of a linked list.

Note: You must use recursion. A non-recursive solution will not receive any marks.

Testing

`listSum.c` contains a main function which allows you to test your `listSum` function. The main function:

- Reads the size of the list
- Reads the values of the list
- Displays the list
- Calls `listSum`
- Displays the result

Here are some examples of its usage:

```
$ make
...
$ ./listSum
Enter list size: 9
```

```
$ ./listSum
Enter list size: 6
Enter list values: 2 4 3 7 0 4
List: [2, 4, 3, 7, 0, 4]
The sum of the values in the list is: 20
$ ./listSum
Enter list size: 5
Enter list values: 3 5 2 4 1
List: [3, 5, 2, 4, 1]
The sum of the values in the list is: 15
$ ./listSum
Enter list size: 2
Enter list values: 42 -4
List: [42, -4]
The sum of the values in the list is: 38
$ ./listSum
Enter list size: 0
List: []
The sum of the values in the list is: 0
```

Task 6 - Insert into an Ordered Linked List

Your task is to implement this function in `listInsertOrdered.c`:

```
void listInsertOrdered(struct list *list, int value);
```

This function should use recursion to insert the given value into an ordered linked list. The list should remain ordered after inserting the value. The function should be able to insert duplicates.

The given file contains a `newNode` function. You can use this in your solution.

Note: You must use recursion. A non-recursive solution will not receive any marks.

Testing

`listInsertOrdered.c` contains a main function which allows you to test your `listInsertOrdered` function. The main function:

- Reads the size of the list

- Reads the value to insert
- Calls `listInsertOrdered`
- Displays the updated list

Here are some examples of its usage:

```
$ make
...
$ ./listInsertOrdered
Enter list size: 3
Enter list values (must be in ascending order): 2 5 7
List: [2, 5, 7]
Enter value to insert: 1
List after inserting 1: [1, 2, 5, 7]
$ ./listInsertOrdered
Enter list size: 3
Enter list values (must be in ascending order): 2 5 7
List: [2, 5, 7]
Enter value to insert: 3
List after inserting 3: [2, 3, 5, 7]
$ ./listInsertOrdered
Enter list size: 3
Enter list values (must be in ascending order): 2 5 7
List: [2, 5, 7]
Enter value to insert: 5
List after inserting 5: [2, 5, 5, 7]
$ ./listInsertOrdered
Enter list size: 3
Enter list values (must be in ascending order): 2 5 7
List: [2, 5, 7]
Enter value to insert: 6
List after inserting 6: [2, 5, 6, 7]
$ ./listInsertOrdered
Enter list size: 3
Enter list values (must be in ascending order): 2 5 7
List: [2, 5, 7]
Enter value to insert: 8
List after inserting 8: [2, 5, 7, 8]
$ ./listInsertOrdered
Enter list size: 0
List: []
Enter value to insert: 42
List after inserting 42: [42]
```

Your task is to implement this function in `listInsertNth.c`:

```
void listInsertNth(struct list *list, int n, int value);
```

This function should use recursion to insert the given value before position `n` of the linked list. The list elements are numbered in the same manner as array elements, so the first element in the list is considered to be at position 0, the second element is considered to be at position 1, and so on. If there are less than `n` elements in the list, the new value should be inserted at the end of the list. You can assume that `n` is non-negative.

The given file contains a `newNode` function. You can use this in your solution.

Note: You must use recursion. A non-recursive solution will not receive any marks.

Testing

`listInsertNth.c` contains a main function which allows you to test your `listInsertNth` function. The main function:

- Reads the size of the list
- Reads the values of the list
- Displays the list
- Reads the values of `n` and `value`
- Calls `listInsertNth`
- Displays the updated list

Here are some examples of its usage:

```
$ make
...
$ ./listInsertNth
Enter list size: 3
Enter list values: 16 7 8
List: [16, 7, 8]
Enter position and value to insert: 0 12
List after inserting 12 at position 0: [12, 16, 7, 8]
$ ./listInsertNth
Enter list size: 3
Enter list values: 16 7 8
List: [16, 7, 8]
```

```
Enter list size: 3
Enter list values: 16 7 8
List: [16, 7, 8]
Enter position and value to insert: 2 12
List after inserting 12 at position 2: [16, 7, 12, 8]
$ ./listInsertNth
Enter list size: 3
Enter list values: 16 7 8
List: [16, 7, 8]
Enter position and value to insert: 3 12
List after inserting 12 at position 3: [16, 7, 8, 12]
$ ./listInsertNth
Enter list size: 3
Enter list values: 16 7 8
List: [16, 7, 8]
Enter position and value to insert: 4 12
List after inserting 12 at position 4: [16, 7, 8, 12]
$ ./listInsertNth
Enter list size: 1
Enter list values: 42
List: [42]
Enter position and value to insert: 0 16
List after inserting 16 at position 0: [16, 42]
$ ./listInsertNth
Enter list size: 0
List: []
Enter position and value to insert: 0 2
List after inserting 2 at position 0: [2]
$ ./listInsertNth
Enter list size: 0
List: []
Enter position and value to insert: 10 2
List after inserting 2 at position 10: [2]
```

Submission

Submit via the command line using the `give` command:

```
$ give cs2521 lab02 gcd.c rabbits.c listTail.c listMax.c listSum.c listInsertOrdered.c
listInsertNth.c
```

WARNING:

After you submit, you must check that your submission was successful by going to your submissions page. Check that the timestamp is correct. If your submission does not appear under Last Submission or the timestamp is not correct, then resubmit.

Assessment

All of the marks for this lab will come from automarking, so there is no need to show your code, however, your tutors may still come around and check whether you are comfortable with using recursion. The marks will be distributed as follows:

Task	Mark
Task 1 - `gcd`	0.75
Task 2 - `rabbits`	0.75
Task 3 - `listTail`	0.70
Task 4 - `listMax`	0.70
Task 5 - `listSum`	0.70
Task 6 - `listInsertOrdered`	0.70
Task 7 - `listInsertNth`	0.70

Automarking will be run after submissions have closed. After automarking is run you will be able to view your results here.

Appendix

Hints

Hints for `gcd`

Hint 1 (base case)

Hint 2 (recursive case)

Hints for `rabbits`

Hint 1

Hints for `listTail`

Hint 1 (base case)

Hints for `listMax`

Hint 1 (base case)

Hint 2 (recursive case)

Hints for `listInsertOrdered`

Hint 1 (base case)

Hint 2 (base case)

Hint 3 (recursive case)

COMP2521 23T2: Data Structures and Algorithms is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.

Week 03 Lab Exercise

Algorithm Analysis and ADTs

Objectives

- To perform theoretical and empirical complexity analysis
- To get practice with linked lists

Admin

Marks	5 (see the Assessment section for more details)
Demo	in the Week 3, 4 or 5 lab session
Submit	see the Submission section
Deadline to submit to give	5pm Monday of Week 4
Late penalty	0.2% per hour or part thereof, submissions later than 5 days not accepted

Setting Up

Create a directory for this lab, change into it, and run the following command:

```
$ unzip /web/cs2521/23T2/labs/week03/downloads/files.zip
```

If you're working at home, download `files.zip` by clicking on the above link and then run the `unzip` command on the downloaded file.

If you've done the above correctly, you should now have the following files:

`Makefile`	a set of dependencies used to control compilation
`theoretical-analysis.txt`	a template to fill in your answers for Task 1

`List.c`	incomplete implementation of the List ADT for Task 3
`runList.c`	a main program for testing the List ADT

You'll also need to copy in some files from the week 1 lab. You can use the `cp` command to do this (or you can use the file browser). The files you need from the week 1 lab are `IntList.h`, `IntList.c` and `sortIntList.c`.

WARNING:

This lab assumes that your `sortIntList` program from the week 1 lab mostly works, and that your `IntListSortedCopy` function uses `IntListInsertInOrder`. If this is not the case, then you should use the sample solution.

Now run the command:

```
$ make
```

This will leave the following executable files in your working directory:

`sortIntList`
This is the same `sortIntList` executable as in the Week 1 lab. It reads a list of integers from standard input, creates a sorted (in ascending order) copy of the list, and then prints the sorted list.

`runList`
This is an interactive command-line program for testing the List ADT. More details about this will be given in Task 3.

Task 1 - Theoretical Analysis

In this task, we will perform **theoretical analysis** on the `IntListInsertInOrder` function from the week 1 lab.

Theoretical analysis is one method for analysing the time complexity of an algorithm. It involves looking at the high-level description of the algorithm (i.e., the pseudocode) or the code itself (if it is implemented) and counting the number of operations (e.g., assignments, arithmetic operations, array accesses) performed in terms of the input

Open `theoretical-analysis.txt`. Your task is to analyse the *best* case, the *worst* case, and the *average* case time complexities of the `IntListInsertInOrder` function, and enter your answers and explanations into the corresponding sections of the file.

Example

To explain best case, worst case, and average case, we will refer to the following function that performs a linear search on an unordered array of size n (this is just an example, you need to analyse the `IntListInsertInOrder` function as stated above):

```
int linearSearch(int A[], int n, int elem) {
    for (int i = 0; i < n; i++) {
        if (A[i] == elem) {
            return i;
        }
    }
    return -1;
}
```

The **best case** time complexity occurs when the input has some property which enables the algorithm to complete as quickly as possible. For linear search, the best case would occur when the element being searched for is the first element of the array, because the function would return `0` straight away without needing to examine any of the other elements. Since the function only needs to examine one item, regardless of the size of the array, the best case time complexity is $O(1)$.

The **worst case** time complexity occurs when the input has some property which causes the algorithm to take as long as possible to complete. For linear search, the worst case would be where the element being searched for is not present in the array. Since the function must examine all n items, the worst case time complexity is $O(n)$.

The **average case** time complexity is determined by making a reasonable assumption about all the possible kinds of input for the algorithm, and then finding the average number of operations/line executions over all of these inputs. For linear search, one possible reasonable assumption is that each element has an equal chance of being searched. Under this assumption, the average number of elements examined by the function will be $n/2$, so the average case time complexity is $O(n)$.

NOTE:

then use it to analyse the time complexity. For example, it is not valid to reason that "if the array has only one item, then linear search only needs to examine one item, so the best case time complexity of the algorithm is $O(1)$" because this explanation does not apply to a larger array. Notice that our explanation for the best case time complexity of linear search does not make any assumptions about the size of the array.

NOTE:

You can assume that *malloc* is $O(1)$.

Task 2 - Empirical Analysis

In this task, we will perform **empirical analysis** on the `sortIntList` program from the week 1 lab and compare the time taken by this program with the time taken by the built-in Unix `sort` command.

Empirical analysis involves running the algorithm on inputs of varying size and composition, recording the amount of time it takes on each run, and plotting the results. The shape of the plot directly corresponds to the time complexity of the algorithm.

The types of inputs we are interested in are random, sorted and reverse-sorted.

We will use the built-in `seq` command to generate numbers. Try out the command - it will generate all integers from 1 up to whatever number you give it.

```
$ seq 10
$ seq 100
```

Since we also want random and reverse-sorted inputs, we will pass the output of `seq` to a second command, `sort`, to reorder the numbers the way we want them.

```
$ seq 100 | sort -R    # random
$ seq 100              # sorted
$ seq 100 | sort -nr   # reverse-sorted
```

NOTE:

directly connected to the standard input (`stdin`) of the command C_2, so that whatever C_1 writes, C_2 reads.

Now that we know how to generate all the types of inputs, we need to pass the inputs to our `sortIntList` program and the built-in `sort` program. This can be achieved by using another pipe symbol. The following commands are for random input - you should be able to deduce the commands for sorted and reverse-sorted inputs. Try the commands yourself - do they correctly sort the numbers?

```
$ seq 100 | sort -R | ./sortIntList    # pass random input to sortIntList
$ seq 100 | sort -R | sort -n          # pass random input to built-in sort program
```

Finally, we need to use the `time` command so we can actually obtain the timing data. The `time` command takes another command, runs it, and then displays how long it took. The following command times the `ls` command:

```
$ /usr/bin/time ls
...
0.00user 0.00system 0:00.00elapsed 66%CPU (0avgtext+0avgdata 2516maxresident)k
0inputs+0outputs (0major+141minor)pagefaults 0swaps
```

Similarly, we can add `/usr/bin/time` to the above commands, but since we are only interested in how long the sorting takes (and not the input generation), we only add it to the final part of the command.

```
# pass random input to sortIntList and time it
$ seq 100 | sort -R | /usr/bin/time ./sortIntList
# pass random input to built-in sort program and time it
$ seq 100 | sort -R | /usr/bin/time sort -n
```

Notice that the `time` command produces three times: user, system and elapsed. (Since the `ls` command is very fast, it will say 0.00 for all of them, but for a complex program like `sortIntList`, they will be different.) Here is what the different times mean:

user time	time spent doing normal computation
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 is most useful is the user time, which represents the time that the program at different points during the different executions. Hence, you should run the same timing test multiple times and take an average.

Before you start collecting timing data, there are two issues we need to resolve. Firstly, the `sortIntList` and `sort` programs sort their input and print the result to `stdout`. But we are not interested in the output of these programs, since we know that the programs are correct. All we care about is the time that the programs take. To ignore the output, we can redirect it to `/dev/null`, like so:

```
# pass random input to sortIntList and time it
$ seq 100 | sort -R | /usr/bin/time ./sortIntList > /dev/null
# pass random input to built-in sort program and time it
$ seq 100 | sort -R | /usr/bin/time sort -n > /dev/null
```

Secondly, the `time` command produces a bunch of extra information that we don't need. In fact, we only care about the user time. Thankfully, the `time` command allows us to customise how its output is formatted so that we can make it only show user time.

```
# pass random input to sortIntList and time it
$ seq 100 | sort -R | /usr/bin/time -f "%U" ./sortIntList > /dev/null
# pass random input to built-in sort program and time it
$ seq 100 | sort -R | /usr/bin/time -f "%U" sort -n > /dev/null
```

Now the commands produce just a single time, which makes the data easier to collate.

Great! Now you can start collecting timing data using commands similar to the ones above. If you think you'll get tired of entering commands over and over again, we have provided a script at the end of this lab exercise that will collect data for you.

You'll find that both `sortIntList` and `sort` sort small lists (e.g., up to 10000) very quickly, so there'll be very little difference between them in the timing data for these input sizes. You should use progressively larger list sizes such as 20000, 40000, 80000 and 100000 until there is an appreciable difference between them.

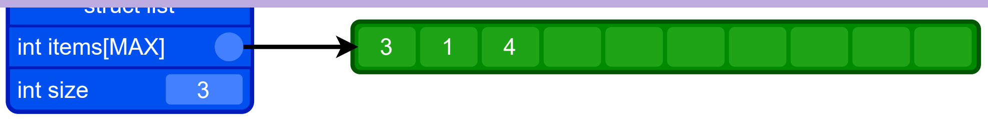
If you find that the `sortIntList` program takes a long time on some input types but is very fast on other input types as the input size increases, you can stop testing the slow input types and test the program with the fast input types only. You should keep increasing the input size until `sortIntList` takes at least 0.5 seconds on the fastest input type.

Put your results in the table in `empirical-analysis.txt`. You need to fill in three columns: (1) *Number of Runs*, which indicates how many runs of each program were used to compute the average time-cost (5 runs is sufficient), (2) *Avg Time for sortIntList*, Write some short paragraphs under the table to *explain* any patterns and trends that you notice in the timing results. Don't just re-state the timing result in words; try to explain *why* they happened. Also explain what you think the time complexity of the `sortIntList` program is for each of the input types (random, sorted and reverse-sorted). If you want to double-check your answers, you could plot the results (e.g., on Google Sheets or Excel).

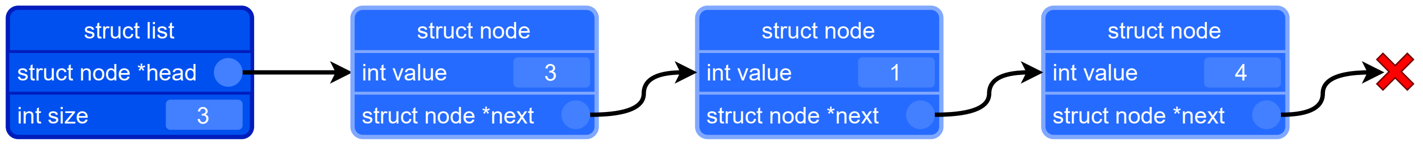
Hint: Since the `sortIntList` program was compiled from *your* code in `IntList.c`, you should refer to your code in your explanation.

Task 3 - List ADT

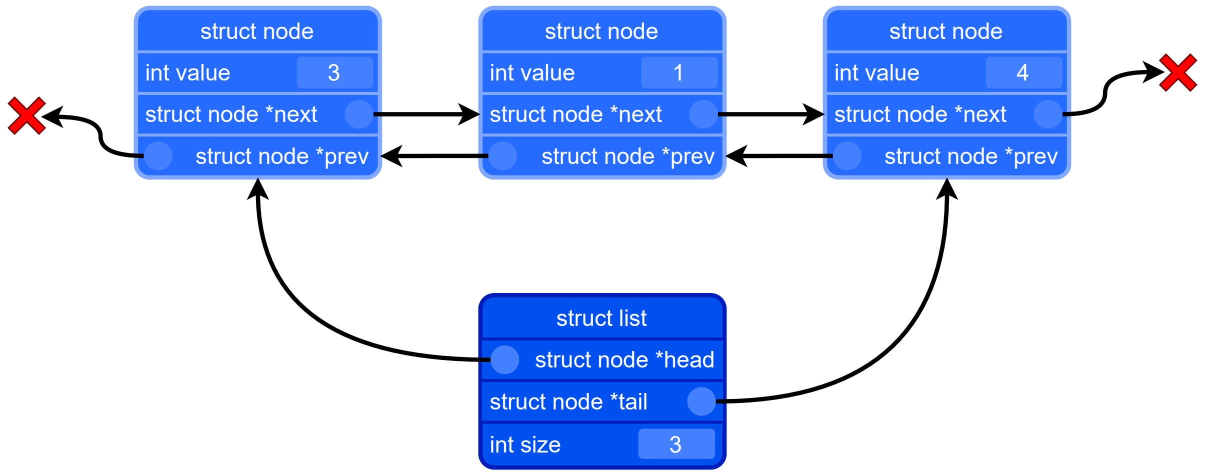
In lectures, we considered three different ways of implementing a List ADT: an array, a singly linked list, and a doubly linked list.



Singly Linked List Implementation



Doubly Linked List Implementation



Note: A List ADT is not to be confused with a linked list. A List ADT represents a linear collection of elements which could be implemented in a number of ways, including a linked list.

In a doubly linked list, each node contains *two* pointers: one to the next node (as in a singly linked list) and one to the previous node, and the list struct contains pointers to the first and last nodes.

The doubly linked list implementation has a clear advantage over the other two, in that we can very efficiently insert and remove items at the start and end if we needed to. With the array implementation, we would not be able to efficiently insert and remove items at the start (unless it was implemented as a *circular array*), and with the singly linked list implementation, we would not be able to efficiently insert and remove items at the end - even if the list struct had a `tail` pointer, removing from the end would still be inefficient.

approaches in general is that we can't efficiently get the element at a specific index. However, it is well worth the cost if we need to frequently operate on both the start and end of the list.

Your task is to finish the doubly linked list implementation of the List ADT in `List.c` that was started in lectures. Most of the functions are complete already, but you need to implement the following:

- `ListAddStart` - adds an element to the start of the list
- `ListAddEnd` - adds an element to the end of the list
- `ListDeleteEnd` - deletes an element from the end of the list

To help you test your implementation, we have provided a command-line program `runList` which lets you perform operations on a list such as adding to the start, deleting from the end, and so on. Here is an example run of the program once it is working correctly:

```
$ make
...
$ ./runList
Interactive List Tester
Enter ? to see the list of commands.
> +e 1
Added 1 to the end of the list
> +s 3
Added 3 to the start of the list
> +e 4
Added 4 to the end of the list
> p
[3, 1, 4]
> s
The size of the list is 3
> -s
Deleted 3 from the start of the list
> f
The first element in the list is 1
> -e
Deleted 4 from the end of the list
> p
[1]
> q
$
```

small mistakes which break the list (as was discovered by yours truly!).

Submission

You need to submit three files: `theoretical-analysis.txt`, `empirical-analysis.txt` and `List.c`. **You must submit all of these files, even if you did not complete all of the tasks.** You can submit via the command line using the `give` command:

```
$ give cs2521 lab03 theoretical-analysis.txt empirical-analysis.txt List.c
```

You can also submit via give's web interface. You can submit multiple times. Only your last submission will be marked. You can check the files you have submitted here.

WARNING:

After you submit, you must check that your submission was successful by going to your submissions page. Check that the timestamp is correct. If your submission does not appear under Last Submission or the timestamp is not correct, then resubmit.

Assessment

Tasks 1 and 2 are handmarked. To receive a mark for these tasks, you *must* show your work to your tutor during your Week 3, 4 or 5 lab session. You will be marked based on the following criteria:

Theoretical analysis (1.5 marks)
These marks are for how accurate you are with the time complexities that you obtained in Task 1 and the quality of your explanations.

Empirical analysis (2 marks)
These marks are for (1) whether you collected an adequate amount of timing data and (2) how well you can explain the trends/patterns in the timing results you collected in Task 2. Note: you don't need to understand or explain how Unix `sort` works, but you

List ADT (1.5 marks)

These marks are for the correctness of your List ADT implementation in Task 3, and will come from automarking. Automarking will be run after submissions have closed. After automarking is run you will be able to view your results here.

Appendix

Task 2 script

This is a shell script to help you collect timing data for Task 2. You likely won't fully understand it unless you've taken COMP2041 or had prior experience with shell scripting, but feel free to use it anyway.

To use this script, copy it to a file, make sure `sortIntList` has been compiled, and then run `sh` *`script-name`*, where *`script-name`* is the name of the file you copied the script to.

```
#!/bin/sh

num_runs=5

for order in random sorted reverse # NOTE: feel free to change this list
do
    for input_size in 10000 20000 40000 # TODO: add more input sizes
    do
        for program in ./sortIntList "sort -n"
        do
            echo
            echo "Input size: $input_size, Order: $order, Program: $program"
            for i in $(seq 1 $num_runs)
            do
                case $order in
                    random)
                        seq -f %1.0f $input_size | sort -R | /usr/bin/time -f "%U" $program
> /dev/null
                        ;;
                    sorted)
                        seq -f %1.0f $input_size | /usr/bin/time -f "%U" $program >
/dev/null
                        ;;
                    reverse)
                esac
            done
        done
    done
done
```

NOTE:

If you get error messages like:

```
$'\r': command not found
syntax error near unexpected token `$'do\r''
```

then run the command `dos2unix` *`script-name`* (where *`script-name`* is the name of the script) and try again.

COMP2521 23T2: Data Structures and Algorithms is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs2521@cse.unsw.edu.au
CRICOS Provider 00098G

Week 04 Lab Exercise

Binary Search Trees

Objectives

- To explore the implementation of binary search trees
- To get some practice with binary search trees
- To implement a level-order traversal
- To get some practice with complexity analysis

Admin

Marks	5 (see the Assessment section for more details)
Demo	in the Week 4, 5 or 7 lab session
Submit	see the Submission section
Deadline to submit to give	5pm Monday of Week 5
Late penalty	0.2% per hour or part thereof, submissions later than 5 days not accepted

Background

In lectures, we introduced the binary search tree data type and implemented some operations on it. In this week's lab, we will implement some additional operations on it.

Recall that a binary search tree is an ordered binary tree with the following properties:

- the tree consists of a (possibly empty) collection of linked *nodes*
- each node contains a single integer value, and has links to two subtrees
- either or both subtrees of a given node may be empty

Setting Up

Create a directory for this lab, change into it, and run the following command:

```
$ unzip /web/cs2521/23T2/labs/week04/downloads/files.zip
```

If you're working at home, download `files.zip` by clicking on the above link and then run the `unzip` command on the downloaded file.

If you've done the above correctly, you should now have the following files:

`Makefile`	a set of dependencies used to control compilation
`bst.h`	interface for the `BST` module
`bst.c`	implementation of the `BST` module (incomplete)
`Queue.h`	interface for the `Queue` ADT
`Queue.c`	implementation of the `Queue` ADT (complete)
`testBst.c`	a main program to read values into a tree and then display the tree
`runBst.c`	interactive test program for the `BST` module
`tests/`	a sub-directory containing some basic test cases
`analysis.txt`	a template for you to enter your time complexity analysis

There is quite a lot of code provided, but most of it is complete, and you don't necessarily need to read it... although reading other people's code is generally a useful exercise. The main code to look at initially is `testBst.c`. This is the main program that will be used for testing the additions you make to the `BST` module.

The next files you should look at are the header files, to find what operations they provide. Finally, you should open the `bst.c` file, since that's the file you need to modify for the tasks below.

Compile the initial version of the files with `make`:

```
$ make
clang -Wall -Werror -g -fsanitize=address,leak,undefined   -c -o runBst.o runBst.c
clang -Wall -Werror -g -fsanitize=address,leak,undefined   -c -o bst.o bst.c
clang -Wall -Werror -g -fsanitize=address,leak,undefined   -c -o Queue.o Queue.c
```

This will produce two executables:

testBst

This program reads in numbers from standard input, inserts the numbers into a binary search tree, and then runs each of the BST functions once on the given tree. Once you've completed all the tasks, the program should behave like the following:

```
$ ./testBst
5 3 7 1 4 6 9
Ctrl-D
BST:
      5
     / \
    /   \
   /     \
  3       7
 / \     / \
1   4   6   9

#nodes:  7
#leaves: 4
Range:   8

In-order:    1 3 4 5 6 7 9
Pre-order:   5 3 1 4 7 6 9
Post-order:  1 4 3 6 9 7 5
Level-order: 5 3 7 1 4 6 9

Deleting all the leaves in the BST...
New BST:
  5
 / \
3   7
```

The tests directory contains some sample inputs and expected outputs for the testBst program. The *t*.in files contain the sample inputs and the *t*.exp files contain the corresponding expected outputs. You can make the testBst program read input from the input files by using input redirection:

```
$ ./testBst < tests/2.in
BST:
      5
     / \
 / \     / \
1   4   6   9

#nodes:  7
#leaves: 4
Range:   8
...
```

We strongly advise you to do your own testing in addition to using the given tests, as the given tests are (intentionally) not very extensive.

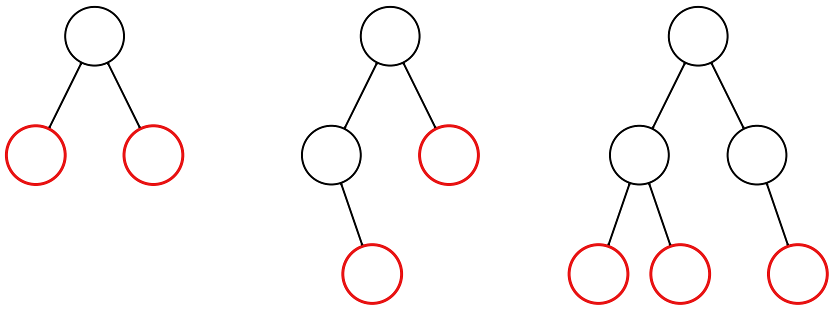
runBst

This program allows you to test the BST functions interactively by entering commands in the terminal. Here is an example run of the program (once you've completed all the tasks):

```
$ ./runBst
Interactive BST Tester
Enter ? to see the list of commands.
> + 6 8 4 1 9 3 5
Inserted 6
Inserted 8
Inserted 4
Inserted 1
Inserted 9
Inserted 3
Inserted 5
> p
    6
   / \
  4   8
 / \   \
1   5   9
 \
  3
> n
The BST contains 7 nodes
> l
The BST contains 3 leaves
> r
The range of the BST is 8
> P
Pre-order traversal: 6 4 1 3 5 8 9
> L
> p
    6
   / \
  4   8
 /
1
> l
The tree contains 2 leaves
> L
Level-order traversal: 6 4 8 1
> d
Deleted all the leaves in the BST
> p
  6
 /
4
> d
Deleted all the leaves in the BST
> p
6
> q
```

Task 1 - Count Leaves in a BST

Implement `bstNumLeaves()` which returns a count of the number of leaf nodes in the given `BST`. A leaf node is any node whose left and right subtrees are empty. The following diagram shows some sample trees, with the leaf nodes highlighted in red.



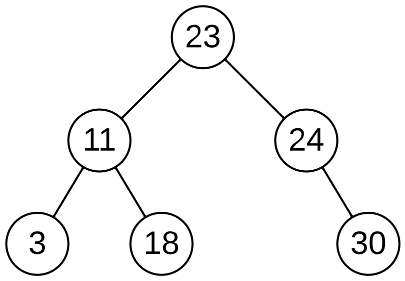
Once you think you've got the function working, test it by recompiling with `make` and running either (or both) of the test programs.

should be in terms of , where is the number of nodes in the tree, or , where is the height of the tree.

> Since the binary search trees in this lab are not guaranteed to be balanced, the height of a tree with nodes can vary significantly. This means for some algorithms, expressing their time complexity in terms of can give a more accurate measure of efficiency. An example of this is searching a BST - an efficient implementation involves only going down one branch of the tree, so its efficiency depends on the height of the tree, rather than the total number of nodes.

Task 2 - Find the Range of a BST

Implement `bstRange()` which returns the range of the given `BST`. The range of a BST is the difference between its smallest and largest values. If the given tree is empty, the function should return -1. For example, consider the following BST:

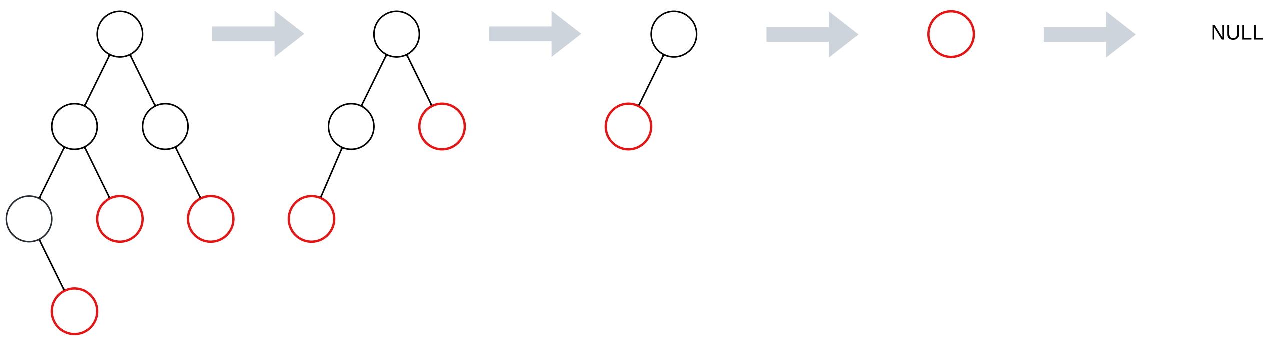


The range of this BST is 27, because the largest value is 30, the smallest value is 3, and
.

Once you think you've got the function working, test it by recompiling with `make` and running either (or both) of the test programs.

When you're certain that the function works correctly, determine its worst case time complexity and write it in `analysis.txt` along with an explanation. The time complexity should be in terms of , where is the number of nodes in the tree, or , where is the height of the tree.

Implement `bstDeleteLeaves()` which deletes all the leaf nodes in the given `BST` and returns the root of the updated tree. The following diagram shows the result of repeatedly applying the function to a sample tree.

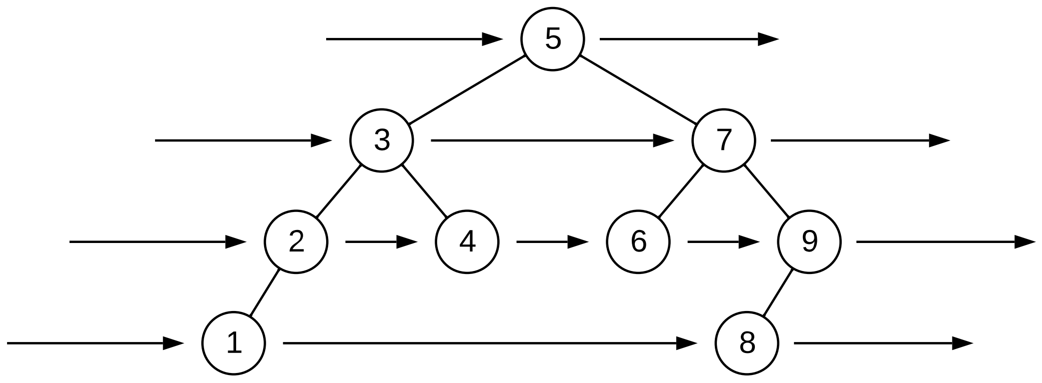


Once you think you've got the function working, test it by recompiling with `make` and running either (or both) of the test programs.

When you're certain that the function works correctly, determine its worst case time complexity and write it in `analysis.txt` along with an explanation. The time complexity should be in terms of , where   is the number of nodes in the tree, or , where   is the height of the tree.

Task 4 - Level-Order Traversal of a BST

Implement `bstLevelOrder()` which prints the values in the `BST` in level-order on a single line separated by spaces (i.e., the same format as the other traverse-and-print functions). **Do not print a newline.** The following diagram aims to give an idea of how level-order traversal scans the nodes in a tree:



The output from this traversal would be: `5 3 7 2 4 6 9 1 8`.

```
Level Order Traversal(BST t):
    initialise an empty queue
    add t's root node to the queue
    while the queue is not empty do
        remove the node at the front of the queue
        print the value in the node
        add its left child (if any) to the queue
        add its right child (if any) to the queue
    end while
```

You must implement this algorithm by making use of the `Queue` ADT provided. Note that the `Queue` ADT stores pointers (`void *` is a generic pointer type). This is because you should store node pointers in the queue rather than integers - if the queue stored integers then after dequeuing an integer, there would be no easy way to add its children to the queue.

When you're certain that the function works correctly, determine its worst case time complexity and write it in `analysis.txt` along with an explanation. The time complexity should be in terms of , where is the number of nodes in the tree, or , where is the height of the tree.

Submission

You need to submit two files: `bst.c` and `analysis.txt`. **You must submit all of these files, even if you did not complete all of the tasks.** You can submit via the command line using the `give` command:

```
$ give cs2521 lab04 bst.c analysis.txt
```

You can also submit via give's web interface. You can submit multiple times. Only your last submission will be marked. You can check the files you have submitted here.

WARNING:

not appear under Last Submission or the timestamp is not correct, then resubmit.

Assessment

Most of the marks for this lab will come from automarking. To receive the rest of the marks, you *must* show your work to your tutor during your Week 4, 5 or 7 lab session. You will be marked based on the following criteria:

Code correctness (4 marks)
These marks will come from automarking. Automarking will be run after submissions have closed. After automarking is run you will be able to view your results here.

Complexity analysis (1 mark)
This mark is based on how accurate you were with your time complexity analysis and the quality of your explanations in `analysis.txt`.

COMP2521 23T2: Data Structures and Algorithms is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs2521@cse.unsw.edu.au
CRICOS Provider 00098G

Week 05 Lab Exercise

Graphs and Social Networks

Objectives

- To explore an application of graphs
- To get some practice with graph problems
- To perform complexity analysis on graph algorithms
- To implement some basic features of social networks

Admin

Marks	5 (see the Assessment section for more details)
Demo	in the Week 5, 7 or 8 lab session
Submit	see the Submission section
Deadline to submit to give	5pm Monday of Week 7
Late penalty	0.2% per hour or part thereof, submissions later than 5 days not accepted

Background

In lectures, we learned that a graph is a collection of vertices and edges between them. This very abstract definition allows for many real-world scenarios and systems to be modelled by graphs - for example, maps, social networks, and the web. In this lab, we will explore an application of graphs in a simple social network app called Friendbook.

Friendbook

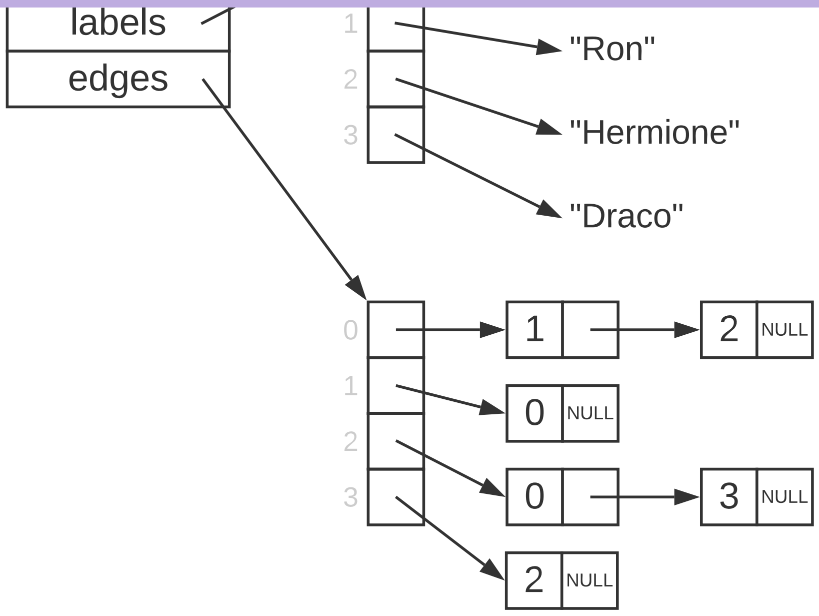
Friendbook is a very simple social network app with the following features:

- People can friend other people (i.e., add them as friends). Friending goes both ways, so if you add someone as a friend, you become their friend as well.
- People can unfriend their friends (i.e., remove them from their friends list). This also goes both ways.
- People can see a count of how many friends they have.
- People can see a list of their friends.
- People can see a list of the mutual friends that they share with someone else.
- People can receive friend recommendations. Friendbook has two different methods of generating recommendations:
  1. The first method only recommends friends of friends, and ranks friend recommendations in order of the number of mutual friends, so people who you share more mutual friends with will be recommended first.
  2. The second method recommends friends of friends first, and then friends of friends of friends next, and then friends of friends of friends of friends, and so on. Anyone who can be reached by following friendship links can be recommended.

Names as Vertices

All of the graph implementations we have seen so far have used integer vertices numbered from 0 to $n-1$, where n is the number of vertices. This is convenient, as vertex numbers can double as indices into the adjacency matrix or adjacency list. But in Friendbook, the vertices are people (names), so how do we represent this internally?

It turns out we don't need to do that much more work. If we give each person an integer ID between 0 and $n-1$ and store a mapping between names and IDs, then we can continue to use the graph representations that we are familiar with. A simple way to implement this mapping would be to store all the names in an array, and let the ID of each person be the index containing their name in the array. The first person in the array would have an ID of 0, the second person in the array would have an ID of 1, and so on. If we wanted to answer a question involving one or more people, we can scan this array to determine their ID, and then use this ID to query the matrix/list. For example, suppose this is our internal representation:



Now suppose we wanted to find out if Harry and Draco are friends. First, we need to find the vertex numbers associated with Harry and Draco, so we perform a linear scan of the array of names (called `labels`), and find that Harry is associated with a vertex number of 0, and Draco is associated with a vertex number of 3. The adjacency list for vertex 0 does not contain vertex 3, so we can conclude that Harry and Draco are **not** friends.

Unfortunately, this translation between names and vertex numbers adds quite a bit of overhead to our graph operations. Converting from vertex numbers to names is easy, as we can go straight to the relevant index in the array ($O(1)$), but converting from names to vertex numbers requires a linear scan of the array, which is $O(n)$. So what was once an $O(1)$ operation (checking if an edge exists) is now an $O(n)$ operation. We can improve the efficiency of the name to vertex number conversion by using a data structure that allows for efficient searching, such as a binary search tree.

Setting Up

Create a directory for this lab, change into it, and run the following command:

```
$ unzip /web/cs2521/23T2/labs/week05/downloads/files.zip
```

If you're working at home, download `files.zip` by clicking on the above link and then run the `unzip` command on the downloaded file.

If you've done the above correctly, you should now have the following files:

`Fb.h`	the interface for the Friendbook ADT
`List.c`	a complete implementation of the List ADT
`List.h`	the interface for the List ADT
`Map.c`	a complete implementation of the Map ADT
`Map.h`	the interface for the Map ADT
`Queue.c`	a complete implementation of the Queue ADT
`Queue.h`	the interface for the Queue ADT
`runFb.c`	a program that provides a command-line interface to the Friendbook ADT
`analysis.txt`	a template for you to enter your time complexity analysis

Once you've got these files, the first thing to do is to run the command

```
$ make
```

This will compile the initial version of the files, and produce the `./runFb` executable.

File Walkthrough

`runFb.c`

`runFb.c` provides a command-line interface to the Friendbook ADT. It creates a Friendbook instance, and then accepts commands to interact with it. Here is an example session with the program once it is working correctly:

```
$ ./runFb
Friendbook v1.0
Enter ? to see the list of commands.
> ?
Commands:
    + <name>                add a new person
    l                       list the names of all people
    f <name1> <name2>       friend two people
    u <name1> <name2>       unfriend two people
    s <name1> <name2>       get the friendship status of two people
    n <name>                get the number of friends a person has
    m <name1> <name2>       list all mutual friends of two people
    r <name>                get friend recommendations for a person based on mutual friends
    R <name>                get friend recommendations for a person based on friendship
closeness
    ?                       show this message
    q                       quit
```

```
> + Ron
Ron was successfully added to Friendbook!
> + Hermione
Hermione was successfully added to Friendbook!
> f Harry Ron
Successfully friended Harry and Ron!
> f Ron Hermione
Successfully friended Ron and Hermione!
> s Harry Ron
Harry and Ron are friends.
> n Harry
Harry has 1 friend.
> n Ron
Ron has 2 friends.
> s Harry Hermione
Harry and Hermione are not friends.
> m Harry Hermione
Harry and Hermione's mutual friends:
    Ron
> r Harry
Harry's friend recommendations
    Hermione               1 mutual friends
> u Harry Ron
Successfully unfriended Harry and Ron!
> s Harry Ron
Harry and Ron are not friends.
> q
$
```

`Fb.c`

`Fb.c` implements the Friendbook ADT. Most of the functions are complete, however, it would be helpful to read through these functions to get a good idea of how they manipulate and obtain information from the graph representation, how they create and return lists of names, and how they convert people's names to vertex numbers. You should also read the definition of `struct fb` and make sure you understand the purpose of each field.

`List.h`

`List.h` defines the interface to the List ADT. Some operations require a list of names to be returned to the user, and the List ADT is used for this purpose. To see how to create a list and add names to the list, you should read some of the already-completed functions in the Friendbook ADT.

An important thing to note is that the Map ADT is not strictly necessary - it is only used for efficiency reasons. If we didn't have access to the Map ADT and wanted to know the ID of a particular person, we could simply scan the `names` array until we found the index containing that person's name, and their ID would be that index.

`Queue.h`

`Queue.h` defines the interface to the Queue ADT. The Queue ADT is currently not used.

Task 1 - Counting Friends

Implement the `FbNumFriends()` function in `Fb.c`, which takes the name of a person and returns the number of friends they have.

Once you think you've got the function working, test it by recompiling with `make` and running the `runFb` program.

When you're certain that the function works correctly, determine its worst case time complexity and write in `analysis.txt` along with an explanation. The time complexity should be in terms of n, where n is the total number of people.

Important: The Map ADT uses an inefficient binary search tree implementation, but you should assume that it uses an AVL tree for complexity analysis.

NOTE:

It is possible to speed up testing by entering your commands into a file and making the `runFb` program read in commands from the file. Here's an example command file that tests a very simple case:

```
+ Harry
+ Ron
+ Hermione
f Harry Ron
f Ron Hermione
n Harry
n Ron
n Hermione
```

terminal to make it easy to see which operations are being performed.

```
$ ./runFb -e < num-friends-1.txt
```

If you've implemented the function correctly, you should get the following output:

```
Friendbook v1.0
Enter ? to see the list of commands.
> + Harry
Harry was successfully added to Friendbook!
> + Ron
Ron was successfully added to Friendbook!
> + Hermione
Hermione was successfully added to Friendbook!
> f Harry Ron
Successfully friended Harry and Ron!
> f Ron Hermione
Successfully friended Ron and Hermione!
> n Harry
Harry has 1 friend.
> n Ron
Ron has 2 friends.
> n Hermione
Hermione has 1 friend.
```

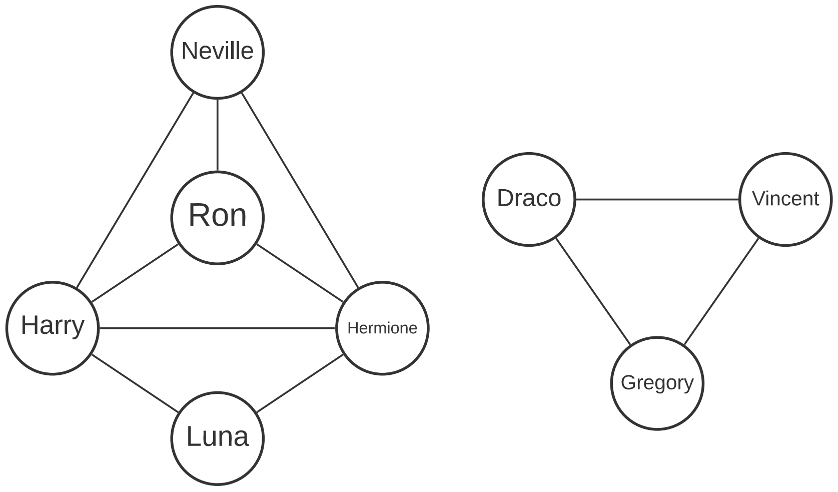
Task 2 - Unfriending :(

Implement the `FbUnfriend()` function in `Fb.c`, which takes the names of two people, and unfriends them if they are friends. The function should return true if the people were friends and were successfully unfriended, and false if the two people were not friends (and so they could not be unfriended).

Once you think you've got the function working, test it by recompiling with `make` and running the `runFb` program.

When you're certain that the function works correctly, determine its worst case time complexity and write in `analysis.txt` along with an explanation. The time complexity should be in terms of n, where n is the total number of people.

Implement the `FbMutualFriends()` function in `Fb.c`, which takes the names of two people, and returns a list of all their mutual friends. A person is a mutual friend of two people if that person is friends with both of those people. To illustrate this, here is an example:



In the example, Harry and Hermione have three mutual friends: Neville, Ron and Luna. Draco and Vincent have one mutual friend: Gregory. Harry and Draco have no mutual friends.

HINT:

To find out how to create a list and add names to it, see the comments in `List.h`, or read one of the existing functions in `Fb.c` that use the List ADT.

Once you think you've got the function working, test it by recompiling with `make` and running the `runFb` program.

When you're certain that the function works correctly, determine its worst case time complexity and write in `analysis.txt` along with an explanation. The time complexity should be in terms of n, where n is the total number of people.

Task 4 - Generating Friend Recommendations

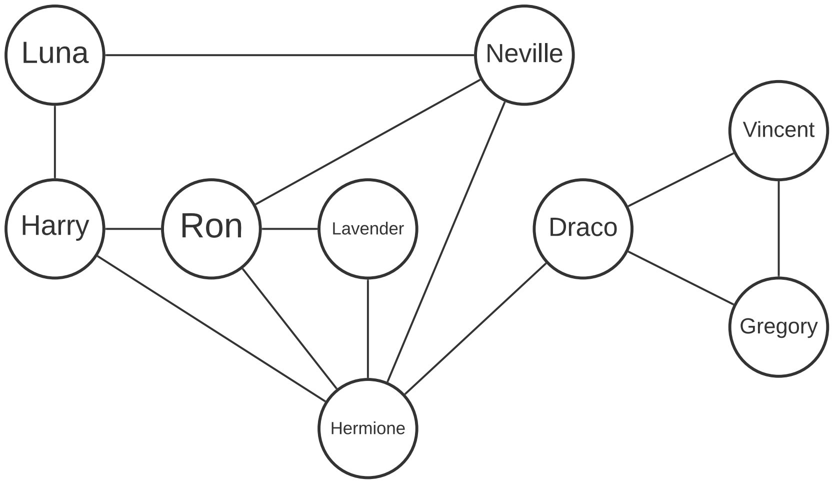
for them in the given `recs` array and return the number of recommendations stored in the array.

The function should only recommend people who are friends of friends of the person. In other words, it should only recommend people who share at least one mutual friend with the person. Obviously, it should not recommend someone who is already the person's friend.

Each recommendation consists of the name of the person being recommended and the number of mutual friends they share with the given person.

The recommendations should be sorted in descending order on the number of mutual friends shared, since someone with more mutual friends is more likely to be known by the person, and is therefore more likely to be added as a friend. If two people share the same number of mutual friends, they may be sorted in any order.

For example, consider the following scenario:



If `FbFriendRecs1()` is called with the name "Harry", the following output should be produced:

```
Harry's friend recommendations:
        Neville                 3 mutual friends
        Lavender                2 mutual friends
        Draco                   1 mutual friends
```

Explanation: Neville should be recommended first as he shares three mutual friends with Harry: Luna, Ron and Hermione. Lavender should be recommended next as she shares two mutual friends with Harry: Ron and Hermione. Draco should be recommended last as he shares just one mutual friend with Harry: Hermione. (*Note:* There is no typo on the last line - it is left as "friends" for simplicity's sake.)

When you're certain that the function works correctly, determine its worst case time complexity and write in `analysis.txt` along with an explanation. The time complexity should be in terms of n, where n is the total number of people.

HINT:

If you're not sure how to order the recommendations properly, consider the following: Given that there are n people, what is the largest number of mutual friends two people could have? What is the smallest number of mutual friends two people could have? Your solution doesn't need to be efficient, so you can consider a brute-force approach.

Optional Task

NOTE:

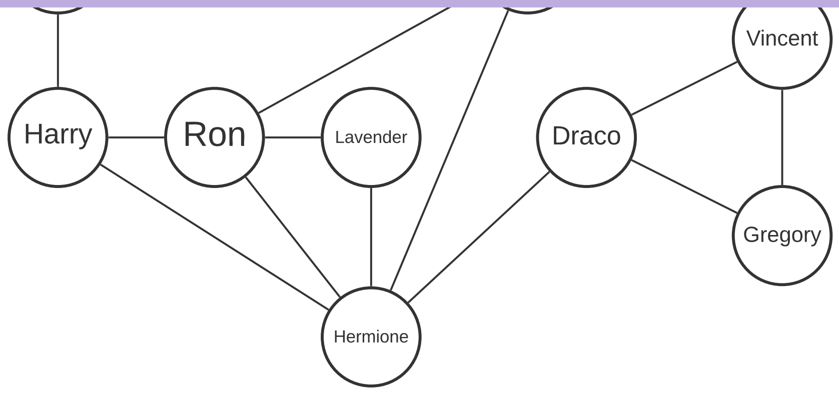
This task is **optional**. It is not worth any marks.

Implement the `FbFriendRecs2()` function in `Fb.c`, which takes the name of a person and finds friend recommendations for them. The function should return a list containing the names of all the people being recommended, with the names being ordered as described below.

Unlike `FbFriendRecs1`, this function should recommend all people who are reachable from the given person via friendship links (not just people who share a mutual friend), and should recommend people who are "closer" to the person first. In other words, friends of friends of the person should be recommended first, then friends of friends of friends, and so on. Obviously, it should not recommend someone who is already the person's friend. If multiple people are the same "distance" from the person, they can be recommended in any order.

Limit the number of recommendations to 20 to avoid generating too many recommendations.

For example, consider the same scenario as in Part 2:



If `FbFriendRecs2()` was called with the name "Luna", the following is one possible valid output:

```
Luna's friend recommendations:
        Ron
        Hermione
        Draco
        Lavender
        Vincent
        Gregory
```

Explanation: Ron and Hermione are the closest people to Luna who are not also her friends, so they are recommended first. The example output recommends Ron first and then Hermione, but it would be equally valid to recommend Hermione first and then Ron. Draco and Lavender are the next furthest away, so they are recommended next. It would be valid to recommend Lavender before Draco. Vincent and Gregory are the next furthest away, so they are printed next. Once again, it would be valid to recommend Gregory before Vincent.

When you think you are done, use the `make` command to recompile the `runFb` program and then develop some scenarios to test your code.

HINT:

You will need to use a graph traversal algorithm to complete this task. But which one? You can review the graph traversal algorithms here, and then follow the pseudocode of your chosen algorithm.

HINT:

Submission

You need to submit two files: `Fb.c` and `analysis.txt`. **You must submit all of these files, even if you did not complete all of the tasks.** You can submit via the command line using the `give` command:

```
$ give cs2521 lab05 Fb.c analysis.txt
```

You can also submit via give's web interface. You can submit multiple times. Only your last submission will be marked. You can check the files you have submitted here.

WARNING:

After you submit, you **must** check that your submission was successful by going to your submissions page. Check that the timestamp is correct. If your submission does not appear under Last Submission or the timestamp is not correct, then resubmit.

Assessment

Most of the marks for this lab will come from automarking. To receive the rest of the marks, you *must* show your work to your tutor during Week 5, 7 or 8 lab session. You will be marked based on the following criteria:

Code correctness (4 marks)
These marks will come from automarking. Automarking will be run after submissions have closed. After automarking is run you will be able to view your results here.

Complexity analysis (1 mark)
This mark is based on how accurate you were with the time complexities that you obtained in Part 4 and the quality of your explanations.

the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs2521@cse.unsw.edu.au
CRICOS Provider 00098G

Week 07 Lab Exercise

Graph Search Algorithms and Maze Solvers

Objectives

- To explore an application of graphs
- To get some practice implementing graph search algorithms
- To perform complexity analysis on graph algorithms
- To understand the difference between BFS and DFS

Admin

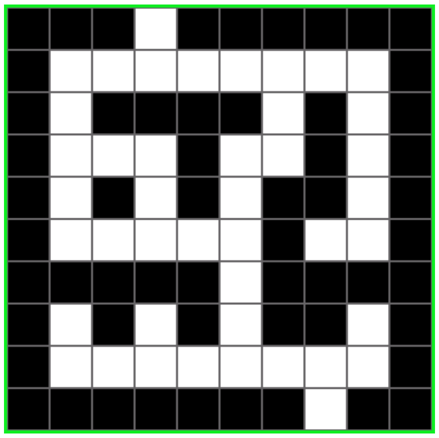
Marks	5 (see the Assessment section for more details)
Demo	in the Week 7, 8 or 9 lab session
Submit	see the Submission section
Deadline to submit to give	5pm Monday of Week 8
Late penalty	0.2% per hour or part thereof, submissions later than 5 days not accepted

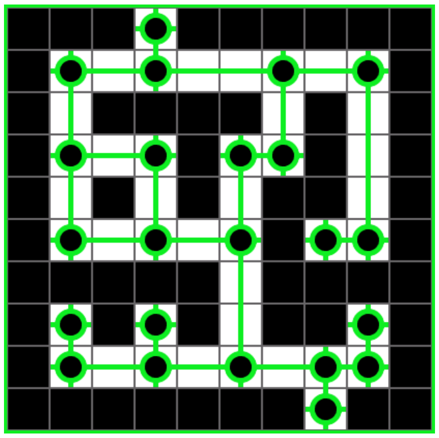
Background

In lectures, we learned about two basic graph search algorithms: breadth-first search (BFS) and depth-first search (DFS). BFS explores the vertices in order of distance from the starting vertex, and is guaranteed to find the shortest path to any vertex (in an unweighted graph). Meanwhile, DFS tries to explore as far as possible by following edges to unvisited vertices before backtracking. In this lab, we will explore the differences between these algorithms by implementing our own maze solvers!

typically with a start and finish. But how can we represent a maze using a graph?

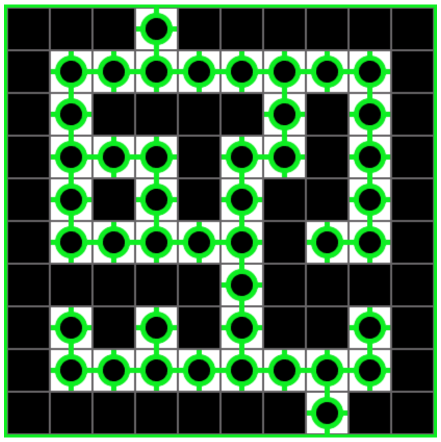
Visually, it's quite easy - given a maze, we can treat each uninterrupted stretch (uninterrupted meaning no turns or intersections) as an edge, and create vertices at the ends of each edge. For example, here is a maze and its graph equivalent:





This maze was taken from this Computerphile video

In this graph representation, each vertex would consist of an identifier, a set of coordinates and a list of neighbours (up to four). However, producing this graph representation is quite complex. For mazes that have a grid layout, it is simpler to treat each every cell of the grid as a potential vertex. Then, identifying neighbours for each vertex is easy - all we need to do is to check the four adjacent cells. If an adjacent cell is a wall cell, then it is not a vertex (and hence not a neighbour), otherwise it must be a neighbour.



In this new representation, all we need is a 2D matrix (array) of booleans to indicate whether a cell is a wall or a path, and it is up to the user of the maze to determine where the vertices and edges are. This is the representation that we will be using in the lab.

Important: This lab must be completed on CSE systems.

Create a directory for this lab, change into it, and run the following command:

```
$ unzip /web/cs2521/23T2/labs/week07/downloads/files.zip
```

If you've done the above correctly, you should now have the following files:

`Makefile`	a set of dependencies used to control compilation
`cell.h`	the definition of the cell data type used by the rest of the code
`Maze.h`	the interface to the Maze ADT
`Maze.o`	a precompiled implementation of the Maze ADT
`Queue.h`	the interface to the Queue ADT
`Queue.o`	a precompiled implementation of the Queue ADT
`Stack.h`	the interface to the Stack ADT
`Stack.o`	a precompiled implementation of the Stack ADT
`matrix.h`	the interface to utility functions for creating matrices (2D arrays)
`matrix.o`	a precompiled implementation of the utility functions for matrices
`solve.h`	the interface to the maze solver, used by `solver.c`
`solveBfs.c`	an implementation of the maze solver using breadth-first search
`solveDfs.c`	an implementation of the maze solver using depth-first search
`solveDfsBacktrack.c`	an implementation of the maze solver using recursive depth-first search
`solveKeepLeft.c`	an implementation of the maze solver using the "keep left" strategy
`solver.c`	a driver program that creates a maze and runs a maze-solving algorithm on it
`mazes/`	a directory containing example mazes
`analysis.txt`	a template for you to fill in your answers for Task 3

Once you've got these files, the first thing to do is to run the command

./solveBfs, ./solveDfs, ./solveDfsBacktrack and ./solveKeepLeft.

File Walkthrough

`solver.c`

`solver.c` is the entry point of the program. Given the path to a maze file, it opens the file, creates a maze from it and then calls `solve()`, which should solve the maze.

`cell.h`

`cell.h` contains the definition of the cell data type used by the rest of the code. A cell is simply represented by two integers: a row number and a column number.

NOTE:

A `struct cell` is a struct, not a struct pointer, so you should access a `struct cell` using the dot operator rather than the arrow operator. You can also easily create new `struct cell` instances without using `malloc`. Here are some examples of its usage:

```
struct cell x = {1, 2};                 // cell (1, 2)
struct cell a = {.row = 1, .col = 2}; // also cell (1, 2) but more explicit
struct cell y = {x.row, x.col + 1};   // this is the cell to the right of x
struct cell z = x;                      // this is a copy of x
z.row = z.row + 1;                      // this modifies z to be the cell under it
```

`Maze.h`

`Maze.h` defines the interface to the Maze ADT, which provides all the functionality required to access information about the maze. In addition to storing the structure of the maze, the Maze ADT also keeps track of the state of all of the cells during a traversal so it can display the maze and produce an animation. You should read `Maze.h`, as you will be using many of its interface functions.

`Queue.h` and `Stack.h`

`Queue.h` and `Stack.h` define the interfaces to the Queue and Stack ADTs respectively. You will need these ADTs to implement your solvers, so you should read the header files to find out how to use them.

`matrix.h`

`matrix.h` defines the interface to some useful functions involving matrices/2D arrays that you may want to use to implement your solvers.

`solveBfs.c`, `solveDfs.c`, `solveDfsBacktrack.c` and `solveKeepLeft.c`

looking for a challenge.

Maze files

The `mazes/` directory contains some example mazes that you can use to test your maze solvers. You can create your own mazes, but to ensure that the mazes are read in correctly, you must follow the format specified in `Maze.h`.

Note that you will **not** be able to access the implementation of the Maze, Queue and Stack ADTs. This is because we want to reinforce the notion that users of an ADT should treat it as a black box, and need not and should not care about its internal representation and implementation. As we have only provided the object files, which are machine specific, this lab can only be completed reliably on CSE systems.

Task 1

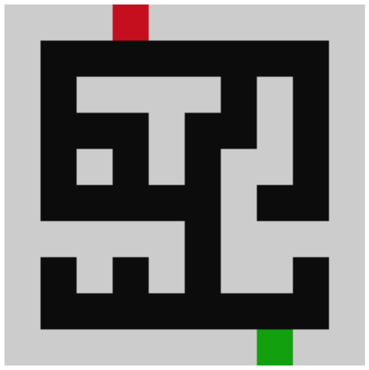
Implement the `solve()` function in `solveBfs.c` which takes in a maze and tries to find a path from start to finish using the breadth-first search algorithm. If there is a path, the function should mark the path on the maze using the `MazeMarkPath()` function and return `true`. Otherwise, the function should return `false`.

While searching the maze, you should call `MazeVisit()` every time you visit a cell. This will cause the maze to be redisplayed with the most recently visited cell marked. **Important:** Additionally, `MazeVisit` will return `true` if the cell you passed it was the finishing cell.

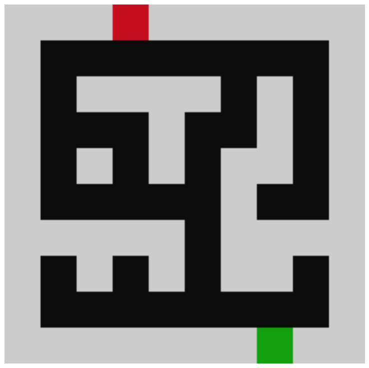
When you want to test your function, use the `make` command to recompile the program and then run `./solveBfs` *`maze-file`*, where *`maze-file`* is one of the maze files in the `mazes/` directory.

WARNING:

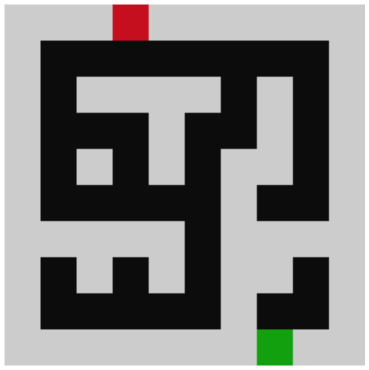
In order for the maze to display properly, you **must** ensure that your terminal window is large enough to fit the entire maze. Otherwise, you may observe strange behaviour, such as wall cells being visited. You may need to reduce your terminal's font size for the largest mazes.



Your code does not have to produce this exact animation - other animations are possible depending on the order in which you visit neighbours. Here is another possible animation produced from a different visit order:



If there is no path from start to finish, then eventually, every cell that is reachable from the starting cell will be visited. Here is a possible animation for `small-2.txt`, which is not solveable:



You can adjust the speed of the animation by providing an additional command-line argument to the program - a number between 1 and 11, where 1 is the slowest and 11 is the fastest. The default speed is 3.

HINT:

```
Inputs: graph g
        starting vertex src

create visited and predecessor arrays
create queue and enqueue src

mark src as visited
while the queue is not empty:
    dequeue v

    for all edges (v, w) where w has not been visited:
        mark w as visited
        set predecessor of w to v
        enqueue w
```

You can follow this pattern in this task, however note that you will need to adapt it for mazes and you should stop the search as soon as you have found the exit. You also need to mark the path after you have found the exit.

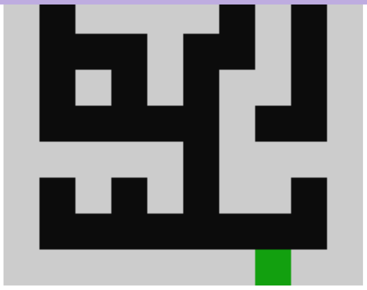
HINT:

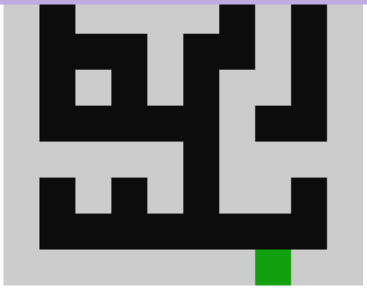
You may find the `create`*`XYZ`* functions in `matrix.h` useful for creating visited and predecessor arrays. If you do use these functions, make sure to free the arrays once you no longer need them with the corresponding `free` functions. Avoid memory leaks!

Task 2

Implement the `solve()` function in `solveDfs.c` which also tries to solve the given maze but instead uses the depth-first search algorithm. Use the iterative implementation of the algorithm (i.e., the version that uses a stack), **do not use recursion**. When you think you are done, use the `make` command to recompile the program and then run `./solveDfs` *`maze-file`*.

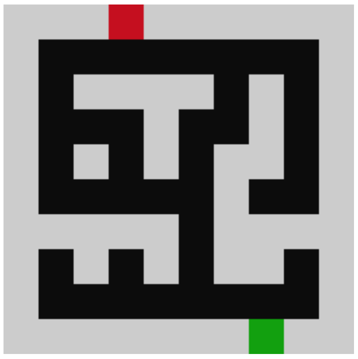
Here are some possible animations produced from `./solveDfs mazes/small-1.txt`:





NOTE:

You should avoid creating small offshoots that don't get explored like in the following animation:



This is not proper DFS behaviour and you may be penalised slightly if your code produces an animation like this.

HINT:

Here is the pseudocode for the DFS algorithm for general graphs:

```
dfs(g, src):
    Inputs: graph g
            starting vertex src

    create visited and predecessor arrays
    create stack and push src

    while the stack is not empty:
        pop v

        if v has been visited:
            continue (i.e., return to beginning of loop)
        for all edges (v, w) where w has not been visited:
            set predecessor of w to v
            push w
```

Task 3

Congratulations for completing your maze solver! Now it's time to analyse the complexity of your algorithms. **Given a maze with n cells in total** (i.e., $n = \text{width} \times \text{height}$), what would be the time complexity of your BFS and DFS algorithms? Enter your answers into `analysis.txt`, along with an explanation for each answer. **Important:** you should ignore the maze-displaying code when analysing the time complexity, as that code only exists to produce an animation.

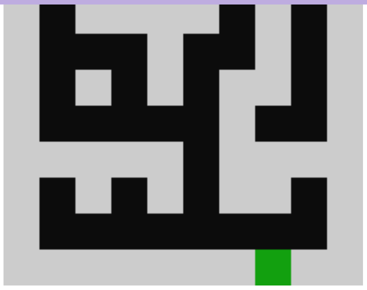
Optional Challenge Task 1

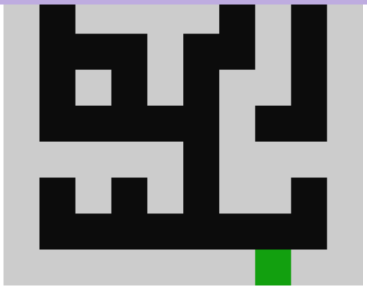
NOTE:

This task is **optional**. It is not worth any marks, and there is no submission for it.

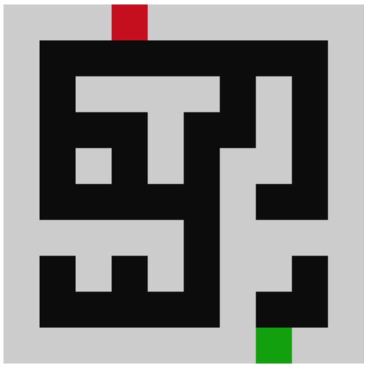
One slight advantage of depth-first search over breadth-first search is that it can be easily implemented recursively, and therefore does not require any additional data structures such as a queue or stack. Recursive depth-first search also induces backtracking behaviour, which occurs when there are no more new vertices to visit from a particular vertex (say B), and you backtrack to the previous vertex (say A) on your current path to continue searching. In the code, backtracking behaviour occurs when the function call where B was visited returns to the call where A was visited.

The backtracking behaviour of recursive depth-first search allows us to produce more natural movement in our animations. Instead of immediately jumping to a new cell once we reach a dead end (like in iterative depth-first search), we show the backwards movement along the path until a new cell is found. Here are some examples of recursive depth-first search and backtracking in action:





If there is no path from start to finish, then eventually, every cell that is reachable from the starting cell will be visited, and the algorithm will backtrack all the way back to the starting cell, like so:



Implement this algorithm in `solveDfsBacktrack.c`. Once you are done, use the `make` command to recompile the program and then run `./solveDfsBacktrack` `*maze-file*`.

HINT:

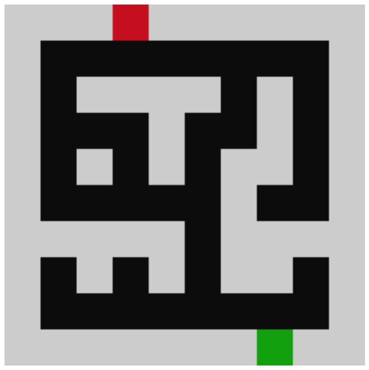
To show the backtracking movement, you will need to call `MazeVisit` more than once on the same cell.

Optional Challenge Task 2

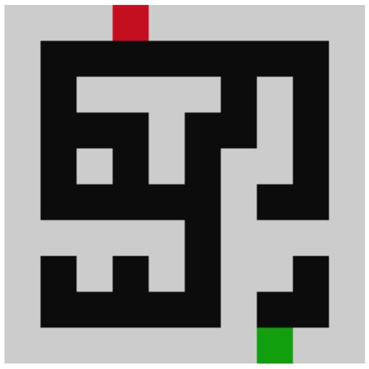
NOTE:

This task is **optional**. It is not worth any marks, and there is no submission for it.

all around the maze. If your goal is simply to find the finish, then this algorithm requires only constant $O(1)$ memory (!), as all you need to keep track of is which cell you are at and what direction you are moving. (If you want to show a path that doesn't involve moving backwards over where you've already been, however, then you will still need a predecessor array.) Note that this is *not* a graph-search algorithm - it's just a simple algorithm that works for certain mazes. Here is the keep-left algorithm in action:



If there is no path from start to finish, then eventually, you will loop back to the start and you can use this to deduce that there is no path.



Notice that if there are cycles in the maze (like the one above), this algorithm may not visit all the cells in the maze - this is why it is only guaranteed to work if the start and finish are on the edges of the maze.

Implement this algorithm in `solveKeepLeft.c`. Once you are done, use the `make` command to recompile the program and then run `./solveKeepLeft maze-file`.

Submission

You need to submit three files: `solveBfs.c`, `solveDfs.c` and `analysis.txt`. **You must submit all of these files, even if you did not complete all of the tasks.** You can

You can also submit via give's web interface. You can submit multiple times. Only your last submission will be marked. You can check the files you have submitted here.

WARNING:

After you submit, you must check that your submission was successful by going to your submissions page. Check that the timestamp is correct. If your submission does not appear under Last Submission or the timestamp is not correct, then resubmit.

Assessment

There is no automarking for this lab. To receive a mark, you *must* show your work to your tutor during your Week 7, 8 or 9 lab session. You will be marked based on the following criteria:

Code correctness (3 marks)
These marks are for the correctness of your code for Parts 1 and 2. To demonstrate the correctness of your code, your tutor will run your submission on a few select mazes.

Complexity analysis (1 mark)
This mark is for how accurate you were with the time complexities that you obtained in Part 3 and the quality of your explanations.

Code style (1 mark)
Code with good style should have these qualities: consistent indentation and spacing, no repetition of code, no overly complicated logic, no overly long functions, correct use of C constructs (such as `if` statements and `while` loops), and comments where appropriate. See the style guide.

Week 08 Lab Exercise

Weighted Graphs and Grid Planning

Objectives

- To explore an application of weighted graphs and minimum spanning trees
- To gain a better understanding of minimum spanning trees and MST algorithms
- To see how graphs might be used with real-world data

Admin

Marks	5 (see the Assessment section for more details)
Demo	in the Week 8, 9 or 10 lab session
Submit	see the Submission section
Deadline to submit to give	5pm Monday of Week 9
Late penalty	0.2% per hour or part thereof, submissions later than 5 days not accepted

Background

When dealing with weighted graphs in the real world, especially networks, there is often a desire to connect nodes in the cheapest way possible, as connecting or traversing nodes usually incurs material, labour or time costs. This is where minimum spanning trees are most useful.

A minimum spanning tree is a subset of the edges of a weighted graph that connects all the vertices together with the minimum possible total edge weight. Minimum spanning trees have applications in the design of networks, such as computer networks, focus on their use in designing electrical networks.

Setting Up

Create a directory for this lab, change into it, and run the following command:

```
$ unzip /web/cs2521/23T2/labs/week08/downloads/files.zip
```

If you're working at home, download `files.zip` by clicking on the above link and then run the `unzip` command on the downloaded file.

If you've done the above correctly, you should now have the following files:

`Makefile`	a set of dependencies used to control compilation
`Graph.h`	the interface for the Graph ADT
`Graph.c`	an incomplete implementation of the Graph ADT
`Pq.h`	the interface for the Priority Queue ADT
`Pq.c`	a complete implementation of the Priority Queue ADT
`place.h`	the definition of the place and power line data types
`gridPlanner.c`	a main program to read in locations and design electrical grids
`planner.c`	an implementation of the grid-planning algorithm (incomplete)
`planner.h`	the interface to the grid-planning algorithm, used by `gridPlanner.c`
`testGraphMst.c`	a main program to test the minimum spanning tree algorithm
`tests/`	a sub-directory containing a collection of test cases

Once you've got these files, the first thing to do is to run the command

```
$ make
```

This will compile the initial version of the files, and produce the `./testGraphMst` and `./gridPlanner` executables.

File Walkthrough

`Graph.h`

`Graph.h` defines the interface to the Graph ADT. The graphs produced by this ADT are undirected, which means that edges are always bidirectional, and weighted, which positive, and self-loops (edges going from a vertex to itself) aren't allowed.

Graph.c

Graph.c contains the implementation of the Graph ADT. The implementation is almost complete except for the GraphMst function, which computes the minimum spanning tree of a given graph. Completing this will be one of the tasks in this lab.

testGraphMst.c

testGraphMst.c reads in graph data from standard input, calls GraphMst to produce a minimum spanning tree of the graph, and then displays the result.

To read in a graph, the program first reads in an integer representing the number of vertices in the graph, and then reads in edges. Each edge is represented by three comma-separated values: two integers which are the endpoints of the edge and a double which is the weight of the edge. Here are some examples of entering graph data into the program:

```
$ ./testGraphMst
3
0, 1, 5.0
0, 2, 4.0
1, 2, 3.0
Ctrl-D
...
$ ./testGraphMst < tests/graphMst/1.in     # from a file
...
```

The program has two output modes. If not given any command-line arguments, the program will output the original graph and the minimum spanning tree in plain text.

```
$ ./testGraphMst < tests/graphMst/1.in
Graph:
Number of vertices: 3
Number of edges: 3
Edge 0 - 1: 5.000000
Edge 0 - 2: 4.000000
Edge 1 - 2: 3.000000

Minimum Spanning Tree:
Number of vertices: 3
Number of edges: 2
Edge 0 - 2: 4.000000
Edge 1 - 2: 3.000000
```

will give you a nice visualisation of the result.

```
$ ./testGraphMst -v < tests/graphMst/1.in > tests/graphMst/1.html
$ # now open the file in a web browser
```

`Pq.h`

`Pq.h` defines the interface to the priority queue ADT. A priority queue is somewhat like a queue except that each item has a priority, and items are removed in priority order. This particular priority queue stores edges and removes the edge with the lowest weight first (i.e., lower weight = higher priority).

`place.h`

`place.h` contains the definition of the place and power line data types used by the rest of the code. A place is represented by a name and two integers `x` and `y` representing its coordinates on a map. A power line is represented by the two locations which it goes between.

NOTE:

`struct place`s and `struct powerLine`s are structs, not struct pointers, so you should access them using the dot operator rather than the arrow operator. You can also easily create copies of `struct place`s and new `struct powerLine`s without using `malloc`.

`gridPlanner.c`

`gridPlanner.c` reads in a series of places from standard input (one per line), calls `planGrid1` or `planGrid2`, which determine where to build power lines to minimise cost, and then displays the result. Like `testGraphMst.c`, this program also has two output modes: plain text and visualiser.

Every place is either a city or a power plant. Places are represented by four comma-separated values: (1) a string which is either "city" or "plant", (2) the name of the place, (3) its x-coordinate, and (4) its y-coordinate. See the input files in the `tests/gridPlanner` directory for examples.

`planner.c`

This file will contain the implementation of your grid planning algorithm. `planGrid1` designs an electrical grid for the simple scenario where there may be many cities but just one power plant. `planGrid2` handles more complex (but realistic) scenarios where there may be many power plants, and is left as a challenge task.

Task 1

Implement the `GraphMst()` function in `Graph.c` which takes in a graph and returns a minimum spanning tree of the graph as a new graph. If the given graph has no minimum spanning tree, the function should return NULL. If the graph has multiple minimum spanning trees, you can return any of them.

You may use any minimum spanning tree algorithm you like. You can (and are encouraged to) make use of the priority queue ADT that we've supplied.

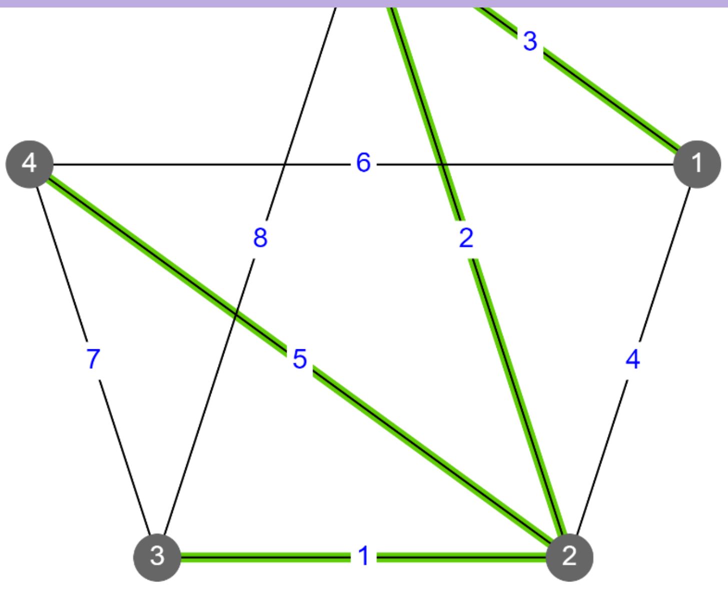
When you think you are done, use the `make` command to recompile the `testGraphMst` program and use the provided input files (and/or your own) in the `tests/graphMst` directory to test your code. Each input file *`t`*`.in` has a corresponding expected output file *`t`*`.exp`. For example:

```
$ ./testGraphMst < tests/graphMst/1.in
Graph:
Number of vertices: 3
Number of edges: 3
Edge 0 - 1: 5.000000
Edge 0 - 2: 4.000000
Edge 1 - 2: 3.000000

Minimum Spanning Tree:
Number of vertices: 3
Number of edges: 2
Edge 0 - 2: 4.000000
Edge 1 - 2: 3.000000
$ ./testGraphMst < tests/graphMst/1.in > tests/graphMst/1.out
$ # compare 1.out with 1.exp manually or with diff
```

A more interesting way to test is to add the **`-v`** flag to toggle the program's visualiser mode. This will cause the program to spew a bunch of HTML that isn't very useful on the terminal, so redirect the output to an `.html` file instead:

```
$ ./testGraphMst -v < tests/graphMst/1.in > tests/graphMst/1.html
```

Now open the file in your web browser to get a nice visualisation of the result. For example, here is what the visualisation for `3.in` should look like:



The edges of the MST (if it exists) will be highlighted in green. The vertices will always be initially arranged in a circle, but you can change how the graph looks by dragging the vertices around the canvas.

NOTE:

The `.html` file must be in the `tests/graphMst` directory for the visualiser to work, because the visualiser depends on other files in that directory (`style.css` and `visualiser.js`). Alternatively, you can move `style.css` and `visualiser.js` to the same directory as the `.html` file.

Task 2

Implement the `planGrid1()` function in `planner.c` which takes an array of cities (of size `numCities`) and a power plant, and determines the most optimal (i.e., cost-minimising) configuration of power lines such that every city has access to electricity. The function should fill the given `powerLines` array with the required power lines and return the number of power lines stored in the array. If there are multiple optimal configurations, you can choose any of them.

You can make the following simplifying assumptions:

- Power lines can only be built between pairs of cities or between a city and a power plant, so you can't create a new place and build a power line to it.
- You can assume the land is flat so that there is no additional cost from a difference in altitude. The cost of a power line between two locations and will therefore be directly proportional to

Please note that you are not required to calculate the total cost - you are only required to find the configuration of power lines that would minimise the cost.

When you think you are done, use the `make` command to recompile the `gridPlanner` program and use the provided input files (and/or your own) in the `tests/gridPlanner` directory to test your code. For example:

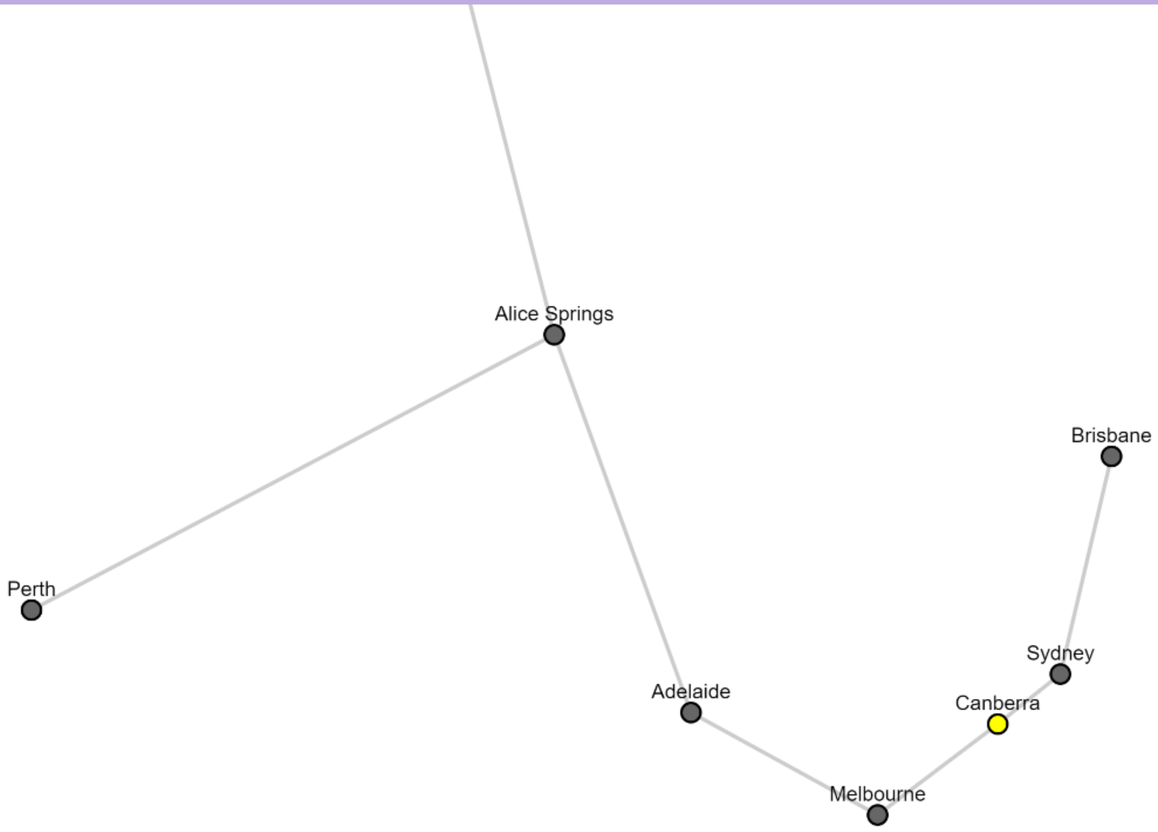
```
$ ./gridPlanner < tests/gridPlanner/1.in
City Sydney at (870, 204)
City Melbourne at (727, 94)
City Adelaide at (581, 174)
City Alice Springs at (474, 469)
City Darwin at (406, 736)
City Perth at (65, 254)
City Brisbane at (910, 374)
Power Plant Canberra at (821, 165)
Power line between Sydney (870, 204) and Canberra (821, 165)
Power line between Sydney (870, 204) and Brisbane (910, 374)
Power line between Melbourne (727, 94) and Canberra (821, 165)
Power line between Melbourne (727, 94) and Adelaide (581, 174)
Power line between Adelaide (581, 174) and Alice Springs (474, 469)
Power line between Alice Springs (474, 469) and Darwin (406, 736)
Power line between Alice Springs (474, 469) and Perth (65, 254)
$ ./gridPlanner < tests/gridPlanner/1.in > tests/gridPlanner/1.out
$ # compare 1.out with 1.exp
```

Each input file *`t`*`.in` has a corresponding expected output file *`t`*`.exp`, but you are not required to match the exact order of power lines or match the order of place names for each power line. For example, it would be valid for the power line between Sydney and Canberra to read:

```
Power line between Canberra (821, 165) and Sydney (870, 204)
```

You can also test by using the program's visualiser mode:

```
$ ./gridPlanner -v < tests/gridPlanner/1.in > tests/gridPlanner/1.html
```



NOTE:

The `.html` file must be in the `tests/gridPlanner` directory for the visualiser to work, because the visualiser depends on other files in that directory (`style.css` and `visualiser.js`). Alternatively, you can move `style.css` and `visualiser.js` to the same directory as the `.html` file.

NOTE:

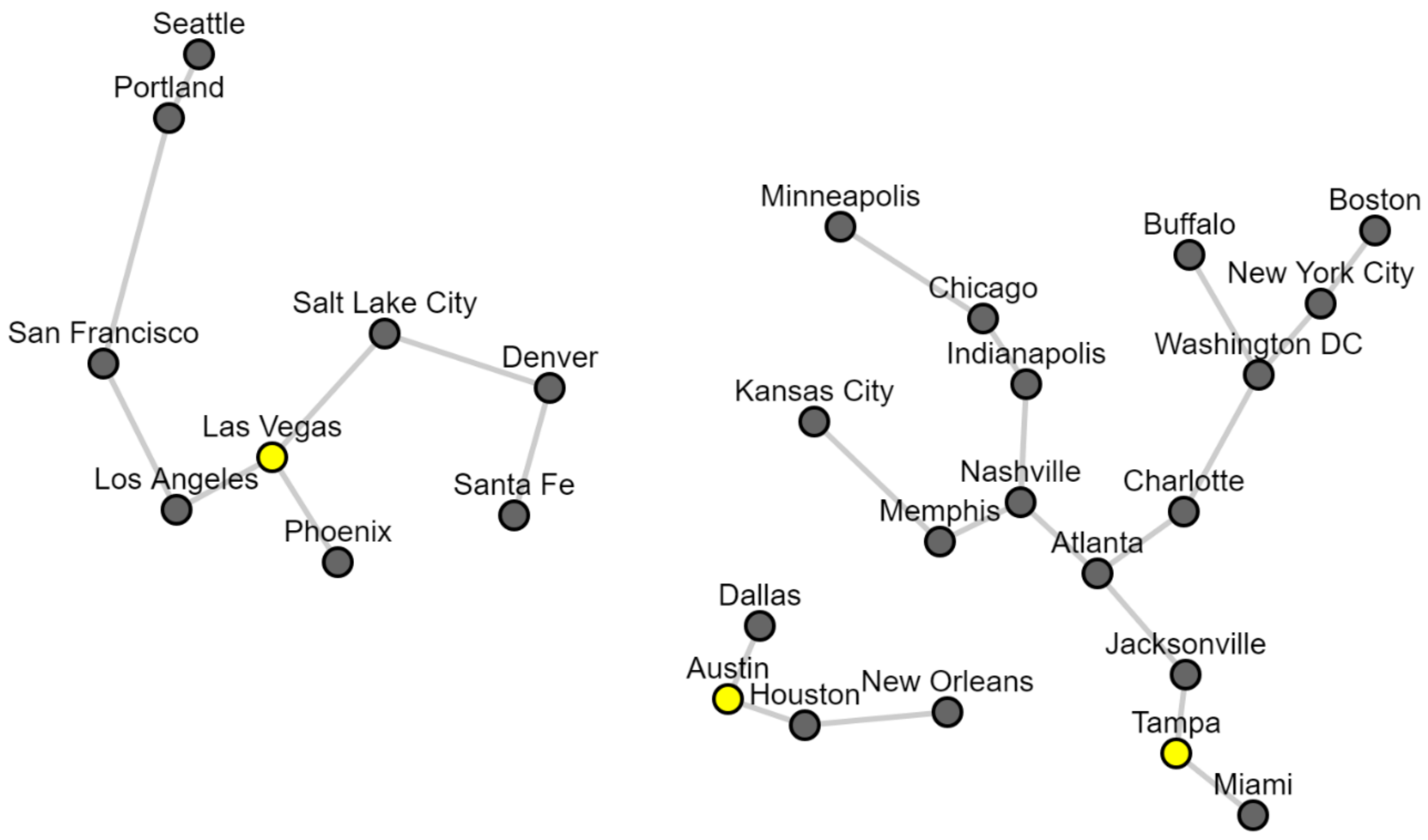
Only the input files `1.in`, `2.in` and `3.in` are relevant for this task. The other input files (`challenge1.in` and `challenge2.in`) are relevant for the challenge task.

Optional Challenge Task

This task is **optional**. It is not worth any marks.

Implement the `planGrid2()` function in `planner.c` which also designs a minimal cost electrical grid, but can take into account multiple power plants, unlike `planGrid1()`. You can make the same simplifying assumptions as in the previous task.

When you think you are done, use the `make` command to recompile the `gridPlanner` program and run it with the provided input files `tests/gridPlanner/challenge1.in` and `tests/gridPlanner/challenge2.in`. For example, here's what the visualisation for `8.in` should look like:



Submission

You need to submit two files: `Graph.c` and `planner.c`. **You must submit all of these files, even if you did not complete all of the tasks.** You can submit via the command line using the `give` command:

```
$ give cs2521 lab08 Graph.c planner.c
```

WARNING:

After you submit, you **must** check that your submission was successful by going to your submissions page. Check that the timestamp is correct. If your submission does not appear under Last Submission or the timestamp is not correct, then resubmit.

Assessment

Most of the marks for this lab will come from automarking. To receive the rest of the marks, you *must* show your work to your tutor during your Week 8, 9 or 10 lab session. You will be marked based on the following criteria:

Code correctness (4 marks)
These marks will come from automarking. Automarking will be run after submissions have closed. After automarking is run you will be able to view your results here.

Code style (1 mark)
This mark is based on your code style. Code with good style should have these qualities: consistent indentation and spacing, no repetition of code, no overly complicated logic, no overly long functions, correct use of C constructs (such as `if` statements and `while` loops), and comments where appropriate. See the style guide.

COMP2521 23T2: Data Structures and Algorithms is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs2521@cse.unsw.edu.au
CRICOS Provider 00098G

Week 09 Lab Exercise

Sort Detective

Objectives

- To familiarise yourself with practical aspects of computational complexity
- To practice a systematic approach to problem solving
- To apply sound scientific reasoning in reaching conclusions
- To hone your analysis skills
- To identify algorithms from their behaviour

Admin

Marks	5 (see the Assessment section for more details)
Demo	in the Week 9 or 10 lab session
Submit	see the Submission section
Deadline to submit to give	5pm Monday of Week 10
Late penalty	0.2% per hour or part thereof, submissions later than 5 days not accepted

NOTE:

In this lab, you are allowed to work with a partner. If you do, you must:

- Include both names and zids in your report
- Indicate whose sort programs you used in your report
- Both submit the report
- Demo the report as a pair at your lab session in Week 9 or 10 to get marks

Background

A very long time ago, in a burst of enthusiasm, Richard Buckland wrote a collection of sort programs. Sadly, he forgot to give the programs meaningful names, so when he later passed them on to COMP2521 lecturers to use for this lab, he couldn't tell us which program used which algorithm. All that he could remember is that the sorting algorithms he used came from the following list:

- Bubble sort
    - standard bubble sort (bubble-up)
- Insertion sort
    - standard insertion sort
- Selection sort
    - standard unstable selection sort
- Merge sort
    - standard merge sort
- Naive quicksort
    - uses the leftmost element as the pivot
- Median-of-three quicksort
    - uses the median of the first, middle and last elements as the pivot
- Randomised quicksort
    - uses a randomly chosen element as the pivot
- Bogosort
    - repeatedly generates permutations of its input until one is found that is sorted

Despite not knowing which program uses which algorithm, Richard did remember a few things about the programs:

- All of the programs read their input from stdin and write the sorted version of the data to stdout
- Sorting happens line-by-line, like the Unix sort program
- There is a limit on the size of the input the programs can process (10,000,000 lines), because they read their input into a fixed size array and sort it there, before writing out the sorted result
- The programs all expect each line to start with a number, which acts as the sorting key. The sorting is numeric, which means that the programs all behave something like the Unix sort program run with the -n option. If no number is present at the start of a line, it will be treated as a zero value.

```
$ sort -n < data > sorted_data
```

Your task is to help us identify the specific sorting algorithms used in two of these programs. You will not be able to view the source code. Instead, you will have to try to identify the algorithms using only the programs' observable behaviour when sorting different data. Note that since the programs are only available in binary format, they will most likely only run on the CSE machines, so you'll have to do your work there.

In the setup phase, we will give you access to two different sort programs; each lab pair or individual gets a different (randomly chosen) pair of programs. The first phase of the task is to design and write up the experimental framework that you plan to use to solve the problem. In the second phase, you should gather data on the execution behaviour of the two programs according to your exerimental setup. You then add the data to your report and analyse it to reach a conclusion on which algorithm each of the supplied programs contains.

To make your task a little simpler, we've supplied a program to generate data in the correct format for the sorting programs to use.

```
$ /web/cs2521/23T2/labs/week09/scripts/gen 5 R
1 nwl
5 arz
4 hcd
2 rbb
3 mqb
$ /web/cs2521/23T2/labs/week09/scripts/gen
Not enough arguments
Usage: /web/cs2521/23T2/labs/week09/scripts/gen  N  A|D|R  [S]
       N = number of lines
       A|D|R = Ascending|Descending|Random
       S = seed for Random
$
```

Use the `gen` program however you like, or not at all. Note that the `gen` program always generates a unique set of keys (from 1..N). This won't enable you to test stability, so you'll need to come up with some more data of your own. Note that the seed argument allows you to generate the same sequence of "random" numbers; if you want to test both sort programs on the same random sequence, use the same seed.

Note that the `setup` script (below) will put a copy of the `gen` executable into your lab directory, so you can run it as `./gen` rather than having to type the long file name.

Create a directory for this lab, change into it, and run the following command:

```
$ /web/cs2521/23T2/labs/week09/setup
```

This must be run on the CSE machines, either via VLAB or SSH. This command will set up two symbolic links in your directory called `sortA` and `sortB` which reference executable programs under the class account. The `setup` command also gives you a copy of the `gen` program and `timeit` shell script.

If you are working in a pair and you are using your partner's sort programs, run the `setup` command above, and then delete your own sort programs with the `rm` command:

```
$ rm sortA sortB
```

Then get your partner to run the following command in their lab directory:

```
$ ls -l sortA sortB
lrwxrwxrwx 1 ... sortA -> long-path-ending-with-sort...A
lrwxrwxrwx 1 ... sortB -> long-path-ending-with-sort...B
```

Finally, copy the long paths that you obtained to create symlinks (shortcuts) to the sort programs:

```
$ ln -s long-path-ending-with-sort...A sortA
$ ln -s long-path-ending-with-sort...B sortB
```

This will give you the same symbolic link as your partner so you can both investigate the same sort programs. **Make sure you remember whose sort programs you are using.**

You can check that the sortA and sortB programs actually do sorting by running something like the following:

```
$ ./gen 5 R
... unsorted output ...
$ ./gen 5 R | ./sortA
... sorted output ...
$ ./gen 5 R | ./sortB
... sorted output ...
```

Phase 1: Designing your Experiment

experimenting with the actual sort programs you have thoroughly thought about what kind of behaviour to look for and what further experimentation might be necessary when analysing your findings.

We expect this will involve coming up with numerous sequences of test data to use, and what differences (and why) you expect to be able to observe from different types of sorting algorithms. Typical properties to look for are execution time and output stability.

Of course, when designing tests you cannot anticipate all possible results which might occur during your experiment. This is the nature of scientific experimentation. But by formalising what you expect to occur and how you will respond, you can better account for unexpected behavior and sensibly revise your design or create new tests once the experiment is under way.

Your experimental design should detail the tests you have devised and explain, with clear reasons, how you will be able to distinguish each algorithm you might be given. You do not need to include all the input data you intend to use, only a description or small sample of it (you may put this in the appendix if you wish).

Write up the experimental design as Part 1 of your report. You can produce the report using whatever tools you want (e.g., OpenOffice, Google Docs, raw HTML, etc.), but it must eventually be submitted as a PDF. Most document processing systems and Web browsers can produce PDF.

There is no size requirement for the report; it is the quality of the report which matters. If you want to include detailed reporting of timing results, then put these in an appendix. Your report should be clear, scientific/systematic in approach, and all reasoning and assumptions should be explicit. Make sure you ask your tutor if you are unclear about what is expected.

To help you get started, a template for the report is available. Note that a fault in many of the reports in the past is that they simply report observations without attempting to analyze them or explain why these results occurred. For this lab try to get beyond just stating observations and explain them.

Phase 2: Run Experiment and Analyse Results

The `setup` command has given you two sort programs to identify. As noted earlier, each sort program reads from standard input and writes to standard output, and assumes that each input line contains a numeric key (first field) and an arbitrary string following the unique.

The following examples show some useful ways of running the sort programs, and auxiliary commands to help collect useful data on their behaviour:

```
# generate some data, put in a file called "mydata"
$ ./gen 100 R > mydata
# sort the data using sortA, put the result in "sortedA"
$ ./sortA < mydata > sortedA
# sort the data using sortB, put the result in "sortedB"
$ ./sortB < mydata > sortedB
# sort the data using Unix sort
$ sort -n < mydata > sorted
# check that the sortA and sortB programs actually sorted
$ diff sorted sortedA # should show no diffs
$ diff sorted sortedB # should show no diffs
# run a large sort and throw away the result
$ ./gen 100000 R | ./sortA > /dev/null
# repeat the above, but get timing data on sortA
$ ./gen 100000 R | time ./sortA > /dev/null
# repeat the timing, but with better output format
$ ./gen 100000 R | /usr/bin/time --format="%U seconds" ./sortA > /dev/null
```

You should now carry out the experiment you designed in Phase 1. Collect and record all of the data, and then summarize it in your report. You can use whatever tools you like to produce useful summaries (e.g. plot graphs of time vs data size). Then analyze the data, draw conclusions, and explain them.

To help with the experiments, we have provided a shell script called `timeit` to collect timing data. As supplied, this script times the built-in sort program, which is not helpful for you, so you'll need to modify it to use one of your `sortA` or `sortB` programs. You could use it as follows:

```
# set up appropriate testing for sortA
$ gedit timeit
# collect timing data
$ sh timeit
# set up appropriate testing for sortB
$ gedit timeit
# collect timing data
$ sh timeit
```

Note that some tests will take a long time to run with large data. You can remove the large data sizes from the outer `for` loop if you can't wait, but you should probably add more smaller sizes to get more data points to try to determine execution cost trends.

Tips for measuring: the Unix `time` command works by sampling and will likely produce different results for the same program run multiple times (the `timeit` script will do this for you). Take an average over a number of timings to account for this. Also, beware of claiming too much accuracy. You can't really claim more than one or two significant digits on an average from the `time` command.

The precise format of your report is up to you, but it must include:

- a summary of the results for each program
- an argument, based on the observed behaviour, for what sorting algorithm you think each program is using

Submission

Your submission for this lab is a report containing an experimental design and results/analysis from carrying out the experiment. It should be submitted as a PDF file called `report.pdf`. You can submit via the command line using the `give` command:

```
$ give cs2521 lab09 report.pdf
```

You can also submit via give's web interface. You can submit multiple times. Only your last submission will be marked. You can check the files you have submitted here.

WARNING:

After you submit, you **must** check that your submission was successful by going to your submissions page. Check that the timestamp is correct. If your submission does not appear under Last Submission or the timestamp is not correct, then resubmit.

Assessment

There is no automarking for this lab. To receive a mark, you *must* show your report to your tutor during your Week 9 or 10 lab session. You will be marked based on the following criteria:

methodology. What experiments and tests did you run, how did you run them, and why?

Results and Analysis (2 marks)

Your report should include the results you obtained from your experiments, including a table of timing results (and possibly some graphs) from your timing experiments, sample input and output from your stability tests, and results from any other experiments you ran. You should also include an analysis of these results: what did the results tell you about what the sorting algorithms could/couldn't be?

Accuracy (1 mark)

This mark is for whether you were able to accurately identify the sorting algorithms used by your sort programs. *Your conclusions must logically follow from your results* - if they are inconsistent with your results or if multiple sorting algorithms were possible based on your results and you didn't properly narrow it down to one algorithm, you may be penalised. If you worked with someone else, make sure to let your tutor know which one of you ran the `setup` command at the start of the lab (and hence whose sort programs you used), since each student is given a different pair of sort programs.

COMP2521 23T2: Data Structures and Algorithms is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs2521@cse.unsw.edu.au
CRICOS Provider 00098G

Extra Lab Exercise

Debugging with GDB and Valgrind

Objectives

- To learn about debugging with GDB
- To debug memory errors and leaks using Valgrind

Admin

This lab is not marked and there is no submission for it. However, we highly recommend that you attempt it so that you can use GDB and Valgrind in your future labs and assignments.

Resources

You may want to consult the following resources:

- GDB Quickstart: Breakpoints and Printing Values
- Breaking, Stepping Over, and Stepping into Functions
- Debugging - GDB Tutorial (another great tutorial)

This very helpful guide was written by tutors of this course.

- Guide to Memory Management and Debugging in C

Setting Up

Create a directory for this lab, change into it, and run the following command:

run the unzip command on the downloaded file.

If you've done the above correctly, you should now have the following files:

Makefile	a set of dependencies used to control compilation
sorter.c	a buggy program with a simple sorting function
Set.h	interface definition for a Set ADT
Set.c	buggy implementation of the Set ADT using a binary search tree
testSet.c	main program for testing the Set ADT

If you run the make command, it will build two executables: sorter and testSet. Both of these programs are buggy. Before you fix the bugs in the programs, make copies of sorter.c and Set.c as follows:

```
$ cp sorter.c sorter.bad.c
$ cp Set.c Set.bad.c
```

NOTE:

We usually compile our programs with AddressSanitizer, but we are not using it in this lab to demonstrate how to use GDB. After you complete this lab, you can try compiling with AddressSanitizer to see just how helpful it is :)

Task 1 - Debugging the Sorter

The aim of the sorter program is to generate a small array containing random numbers, print it, sort the array using bubble sort, and then print the sorted array. It repeats this process five times, generating different random array contents each time.

If the sorter were correct, you would observe something like the following:

```
$ ./sorter
Test #1
Sorting: 83 86 77 15 93 35 86 92 49 21
Sorted : 15 21 35 49 77 83 86 86 92 93
Test #2
Sorting: 62 27 90 59 63 26 40 26 72 36
```

```
Sorted : 11 23 29 30 35 62 67 67 68 82
Test #4
Sorting: 29 02 22 58 69 67 93 56 11 42
Sorted : 02 11 22 29 42 56 58 67 69 93
Test #5
Sorting: 29 73 21 19 84 37 98 24 15 70
Sorted : 15 19 21 24 29 37 70 73 84 98
```

Unfortunately, what you actually observe is:

```
$ ./sorter
Test #1
Sorting: 83 86 77 15 93 35 86 92 49 21
Segmentation fault
```

You may get a different set of random numbers to the above, and maybe even a different error message depending on which machine you're working on, but that doesn't affect the exercise. The program should be able to sort any set of random numbers, but clearly there's a problem.

So, what to do...? You may have noticed that when the programs were compiled, they used the **-g** flag, which sets them up to be used with **gdb**. Run the program under control of `gdb` to find out where it is crashing.

```
$ gdb -q ./sorter
Reading symbols from ./sorter...
(gdb) run
Starting program: some-long-path-name-ending-in/sorter
Test #1
Sorting: 83 86 77 15 93 35 86 92 49 21

Program received signal SIGSEGV, Segmentation fault.
0xXXX...XXX in sort (a=0xXXX...XXX, n=10) at sorter.c:37
37                 if (a[j] < a[j - 1]) {
(gdb)
```

where the *XXX...XXX* are large hexadecimal numbers, which may vary from machine to machine. The exact value is not important.

The `gdb` command can be quite verbose, and part of the skill of using it is working what to ignore. I've highlighted the critical output in red above.

In the sample output above, you can see the line where the error has occured. Sometimes it's useful to get more context than just a single line. You can do this from within `gdb` using the `list` command, e.g.

```
33      int i, j, nswaps;
34      for (i = 0; i < n; i++) {
35          nswaps = 0;
36          for (j = n - 1; j > i; j++) {
37              if (a[j] < a[j - 1]) {
38                  int tmp;
39                  tmp = a[j];
40                  a[j] = a[j - 1];
41                  a[j - 1] = tmp;
(gdb)
```

An alternative to using `list` is simply to keep the program open in an edit window while you run `gdb` in a separate window. GDB also provides a mode so that you can monitor the code and do debugging in a single terminal window; run the `gdb` command with the `-tui` option. Yet another alternative is to use a program like `ddd`, which provides a GUI front-end to `gdb`. For this lab, it may be simpler to stick with plain `gdb`, which has the advantage that it will be available on all Linux machines.

Use `gdb` to find out more information about the state of the program at the point where it crashed. You can find out about the current state of your program in `gdb` using commands like `where` and `print`:

```
(gdb) where         // verify where the program was executing when it crashed
...                 // - gdb gave you a line number above; this will tell you which
function
(gdb) print n       // show the value of the parameter 'n'
...
(gdb) print a       // show the value of the parameter 'a'
...                 // - this is the address of the start of the array
(gdb) print *a      // show the first element in the array
...
(gdb) print a[0]    // show the first element in the array
...
(gdb) print a[2]    // show the third element in the array
...
(gdb) print *a@5    // show the first 5 elements in the array
...
(gdb) print a[j]    // show the j'th element of the array
...
```

Keep examining variables until you find something that looks anomalous. You will then need to find out how it got that way. You could look at the code again and you might spot the error. If not, continue...

executes.

```
(gdb) break sort
...
(gdb) run
...                  // stops at breakpoint ... start of sort function
(gdb) next
...                  // execute next statement, then check variable values
(gdb) next
...
```

If examining the variables at each step doesn't help you to find the problem quickly, then try adding a breakpoint on line 39 (where the error occurs), and re-running the program. After it stops each time, check the value of variables. After each stop/check, you can continue the program with the **continue** command.

Once you've found the problem, change the code to try to fix it, recompile, and see whether the program now exhibits the expected behaviour.

Task 2 - Debugging the Set

If you simply compile the testSet program without change, it will behave as follows:

```
$ ./testSet
Test 1: Create set
Passed
Test 2: Add to set
Segmentation fault
```

Note that this program uses assertions to aid debugging. While assertions provide some information, they may not provide enough to work out what the problem is (e.g. "what is the value of variable i?").

Now run the program under gdb's control, observe the values of variables when it crashes, and use this information to determine the causes of the problems.

Note that **this program has multiple bugs**, so after you fixed one, another will probably manifest itself when you recompile and test. Repeat the above until testSet exhibits the expected behaviour.

Once the Set ADT has been implemented correctly, then the testSet program should produce something like the following:

```
Passed
Test 2: Add to set
Passed
Test 3: Add duplicates
Passed
Test 4: Add more to set
Passed
Test 5: Print set
2 4 6 7 9
Check manually
Test 6: Free set
Now check for memory errors and leaks using valgrind
```

Note that even though a program behaves as expected, this does not guarantee that the code is correct. The code may contain memory errors, which occur when your program tries to read from or write to a memory location that it shouldn't. The code may also contain memory leaks, which occur when your program dynamically allocates memory (using *malloc*), but doesn't free it once it's no longer needed. Memory errors are more difficult to debug, as they don't always manifest themselves, so a program with memory errors may run normally one time, but abnormally the next. They also often lead to strange behaviour that occurs far away from the source of the actual problem.

Run the program in `valgrind` to see if the code contains any memory errors or leaks. If your code *does* contain memory errors or leaks, you might get output that looks like the following:

```
$ valgrind ./testSet
==15336== Memcheck, a memory error detector
==15336== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==15336== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==15336== Command: ./testSet
==15336==
Test 1: Create set
==15336== Invalid write of size 4
==15336==    at 0x1098A4: SetNew (Set.c:38)
==15336==    by 0x1091BD: main (testSet.c:18)
==15336==  Address 0x4a43488 is 0 bytes after a block of size 8 alloc'd
==15336==    at 0x48357BF: malloc (vg_replace_malloc.c:299)
==15336==    by 0x10985F: SetNew (Set.c:31)
==15336==    by 0x1091BD: main (testSet.c:18)
...
...
...
==15336== HEAP SUMMARY:
```

```
==15336== LEAK SUMMARY:
==15336==    definitely lost: 56 bytes in 3 blocks
==15336==    indirectly lost: 0 bytes in 0 blocks
==15336==      possibly lost: 0 bytes in 0 blocks
==15336==    still reachable: 0 bytes in 0 blocks
==15336==         suppressed: 0 bytes in 0 blocks
==15336== Rerun with --leak-check=full to see details of leaked memory
==15336==
==15336== For counts of detected and suppressed errors, rerun with: -v
==15336== ERROR SUMMARY: 19 errors from 19 contexts (suppressed: 0 from 0)
```

Valgrind gives detailed information of memory errors. If your program tried to read from an invalid memory location, Valgrind will report an invalid read. If your program tried to write to an invalid memory location, Valgrind will report an invalid write and tell you the size of the data item that your program wrote. In the above example, Valgrind reported an 'Invalid write of size 4', which means the program likely tried to write an `int` to an invalid memory address (since an `int` is 4 bytes). Valgrind will also tell you the line on which the error occurred. For example, `(Set.c:38)` means Line 38 in `Set.c`.

From this output, it is up to you to figure out the cause of the error and fix it. Here are some common causes of memory errors:

- Not allocating enough memory - this is common with strings/structs
- Trying to access an index beyond the end of an array
- Reading and using an uninitialised value
- Use after free - this is where you free a block memory and then try to access it afterwards.
- Double free - this is where you free the same block of memory twice.

Valgrind also summarises memory leaks. You can get more detailed information about memory leaks by using the `--leak-check=full` option, as the output above suggests.

```
$ valgrind --leak-check=full ./testSet
...
...
...
==15280==
==15280== HEAP SUMMARY:
==15280==     in use at exit: 56 bytes in 3 blocks
==15280==   total heap usage: 7 allocs, 4 frees, 1,152 bytes allocated
==15280==
==15280== 8 bytes in 1 blocks are definitely lost in loss record 1 of 3
==15280==    at 0x48357BF: malloc (vg_replace_malloc.c:299)
```

```
==15280== 24 bytes in 1 blocks are definitely lost in loss record 2 of 3
==15280==    at 0x48357BF: malloc (vg_replace_malloc.c:299)
==15280==    by 0x1099F7: newNode (Set.c:80)
==15280==    by 0x109982: doSetAdd (Set.c:68)
==15280==    by 0x1099D1: doSetAdd (Set.c:74)
==15280==    by 0x1099D1: doSetAdd (Set.c:74)
==15280==    by 0x109944: SetAdd (Set.c:61)
==15280==    by 0x1095B2: main (testSet.c:66)
==15280==
==15280== 24 bytes in 1 blocks are definitely lost in loss record 3 of 3
==15280==    at 0x48357BF: malloc (vg_replace_malloc.c:299)
==15280==    by 0x1099F7: newNode (Set.c:80)
==15280==    by 0x109982: doSetAdd (Set.c:68)
==15280==    by 0x1099A4: doSetAdd (Set.c:72)
==15280==    by 0x1099D1: doSetAdd (Set.c:74)
==15280==    by 0x109944: SetAdd (Set.c:61)
==15280==    by 0x1095C3: main (testSet.c:67)
==15280==
==15280== LEAK SUMMARY:
==15280==    definitely lost: 56 bytes in 3 blocks
==15280==    indirectly lost: 0 bytes in 0 blocks
==15280==      possibly lost: 0 bytes in 0 blocks
==15280==    still reachable: 0 bytes in 0 blocks
==15280==         suppressed: 0 bytes in 0 blocks
==15280==
==15280== For counts of detected and suppressed errors, rerun with: -v
==15280== ERROR SUMMARY: 22 errors from 22 contexts (suppressed: 0 from 0)
```

Valgrind will tell you where the memory that was leaked was allocated. From there, you should be able to figure out why you aren't freeing the memory and where you should be freeing it.

Once you have fixed all the memory errors and leaks, Valgrind should output something like the following:

```
$ valgrind ./testSet
==22805== Memcheck, a memory error detector
==22805== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==22805== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==22805== Command: ./testSet
==22805==
Test 1: Create set
Passed
Test 2: Add to set
Test 4: Add more to set
Passed
Test 5: Print set
2 4 6 7 9
Check manually
Test 6: Free set
Now check for memory errors and leaks using valgrind
==22805==
==22805== HEAP SUMMARY:
==22805==     in use at exit: 0 bytes in 0 blocks
==22805==   total heap usage: 7 allocs, 7 frees, 1,160 bytes allocated
==22805==
==22805== All heap blocks were freed -- no leaks are possible
==22805==
==22805== For counts of detected and suppressed errors, rerun with: -v
==22805== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Extras

Command-Line Arguments

You can run a program that requires command-line arguments in GDB by including the command-line arguments in the run command. For example:

```
$ gdb -q ./args
Reading symbols from ./args...done.
(gdb) run hello world
Starting program: some-long-path-name-ending-in/args
...
args[0] = ./args
args[1] = hello
args[2] = world
(gdb)
```

IO Redirection

To redirect input and output when running programs in GDB, you can use < and > the same way as you would when running a program in the shell. For example:

```
$ gdb -q ./prog
Reading symbols from ./prog...done.
```

```
(gdb)
```

Frame Switching

In GDB, the `where` command produces a stack trace of the current state of the program, with functions numbered starting from 0. Sometimes, you may want to know the value of an argument, variable or expression in a function other than the currently executing function (i.e., one of its callers). You can switch to the stack frame of another function on the stack by using the `frame` command. For example, the command `frame 2` will cause you to switch to the stack frame of the function numbered 2.

COMP2521 23T2: Data Structures and Algorithms is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs2521@cse.unsw.edu.au
CRICOS Provider 00098G

Extra Lab Exercise

ADTs and Queues

Objectives

- To acquaint you with the queue ADT
- To understand the difference between interface and implementation in ADTs
- To manipulate a list-based data structure
- To manipulate an array-based data structure
- To implement core queue operations

Admin

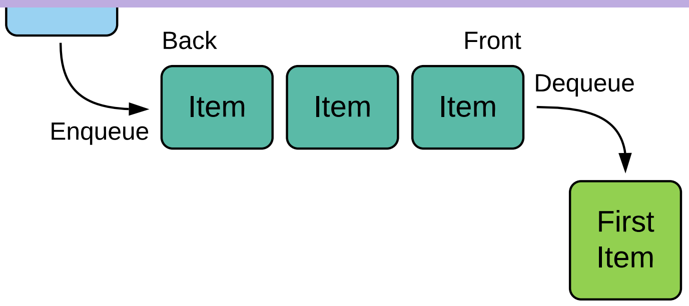
This lab is not marked and there is no submission for it.

Background

At the end of COMP1511, you may have learned about the stack ADT, a last-in first-out (LIFO) collection of elements. Very closely related is the queue ADT, which is first-in first-out (FIFO). Since the queue is an important data structure that we'll be using in some tree and graph algorithms later, it will be useful to have a look at it in more detail.

Queues

You should be reasonably familiar with the concept of a queue in the real world. The way it works in computing is exactly the same! A queue is a linear data structure, with a front and a back. Items enter the queue by joining the back of the queue, and items are removed one at a time from the front of the queue. The operations of adding an item to the back and removing an item from the front are known as enqueuing and dequeuing respectively.



ADTs

A queue is an example of an **abstract data type (ADT)**, which is a data type whose implementation details are hidden from the user. This means that users of an ADT do not have access to its internal representation, and thus cannot access or manipulate the data directly. Instead, users must interact with the ADT through its interface, which defines the operations that are allowed to be performed on the ADT.

In terms of code, the interface to an ADT is defined in its header (*.h) file, and consists of a collection of function declarations. The implementation of these functions are contained in the corresponding *.c file. Only the header file is #included in the main program, which means that users of the ADT do not have access to the internal representation of the ADT (i.e., the fields contained in the struct) and do not know how the functions declared in the header file are implemented.

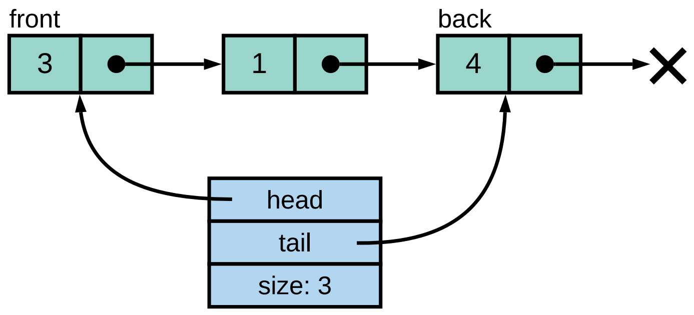
You can think of an ADT as a box with buttons on the outside corresponding to each of the operations that can be performed on the ADT. For example, the queue ADT has two main operations: enqueue and dequeue. If a user enqueues some items, they can expect that when they dequeue, they will get the items back in the same order, because that is how a queue is expected to work. The user is not concerned with the mechanism inside the box (the implementation).

Queue Implementations

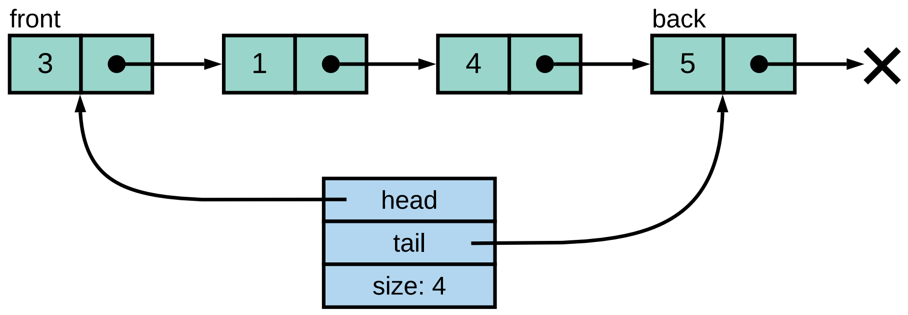
A queue can be implemented in several different ways, for example, using a linked list or using an array. Although each implementation should produce the same queue-like behaviour, some implementations may be more efficient than others so it is important to analyse the implementations and choose the right one to use. Here are the different queue implementations we will explore:

List Queue

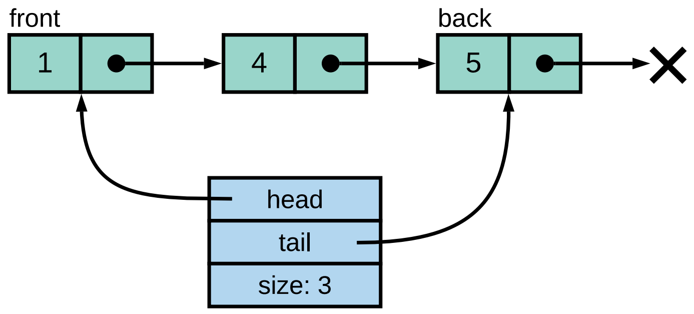
In this implementation, the items in the queue are stored in a linked list. To enqueue an item, the item is added to the end of the list, and to dequeue an item, the item is removed from the beginning of the list. To enable both operations to be efficient, the



After enqueuing 5:

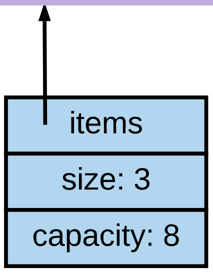


After dequeuing 3:

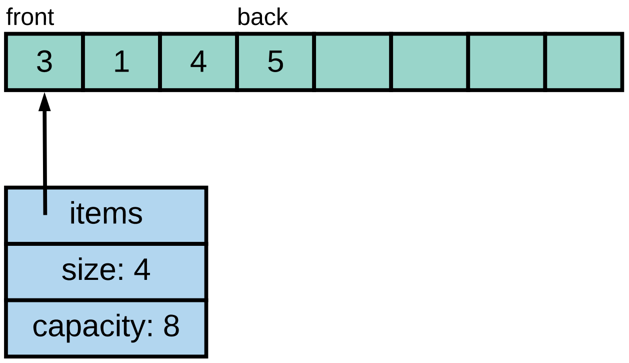


Array Queue

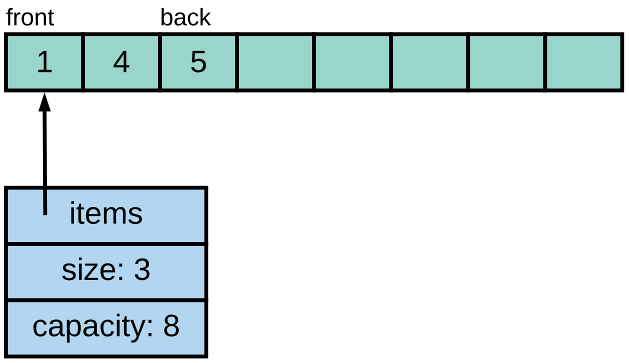
In this implementation, the items in the queue are stored in an array. To enqueue an item, the item is simply placed in the next available slot in the array, and to dequeue an item, the item at index 0 is removed and the rest of the items are shifted down. Since the implementation uses an array, the array may become full and if more items need to be inserted, the array will need to be expanded. Also, since the array will usually not be full, we will need to keep track of both the number of items (i.e., the size of the queue) and the size of the array (i.e., the capacity). Here are some diagrams demonstrating the enqueue and dequeue operations on the array queue:



After enqueuing 5:



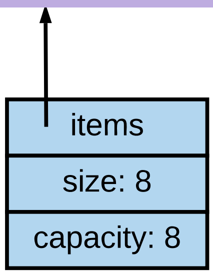
After dequeuing 3:



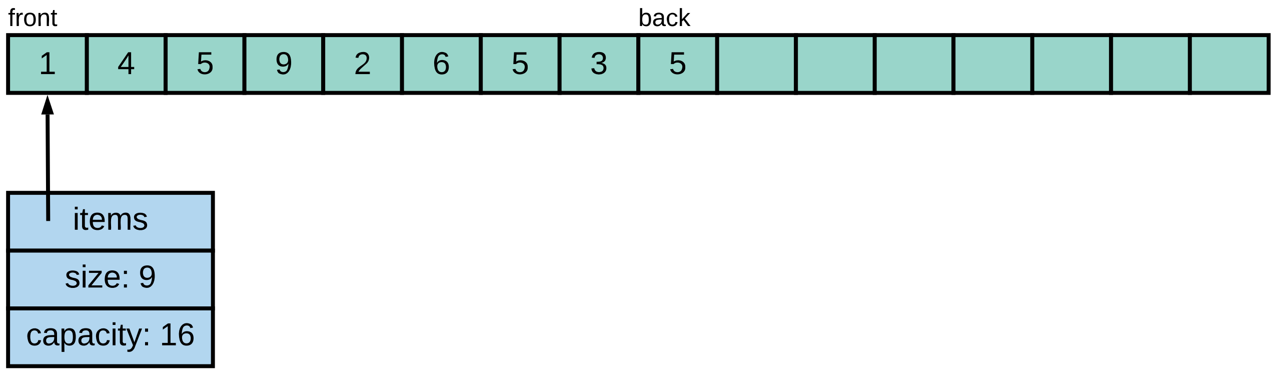
NOTE:

In these diagrams, when an array index is empty, that simply means that the value at that index is irrelevant, so it could contain any value.

The following diagrams show the behaviour of the array queue when the queue becomes full and more items need to be inserted:



After enqueuing 5:



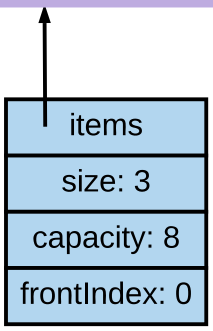
To make room for the new item, the array is resized to double its original size. Note that the capacity field is updated to reflect the new size of the array.

Circular Array Queue

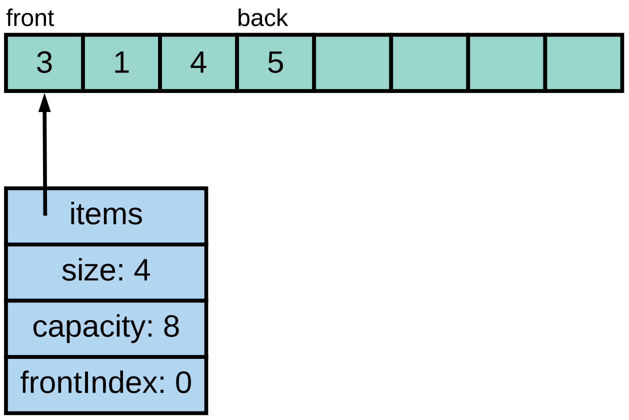
This implementation is similar to the array queue, except that when we dequeue, we don't shift the rest of the items down - we simply leave them and the next index becomes the front of the queue. If the front of the queue was at the end of the array, then it circles back to the start of the array (which is where the name *circular array* comes from). If enough enqueues and dequeues are performed, the queue may circle around the array many times. Since the front of the queue is not necessarily at index 0 anymore, another variable is needed to keep track of the index containing the item at the front of the queue. The following diagrams demonstrate the behaviour of the queue:

NOTE:

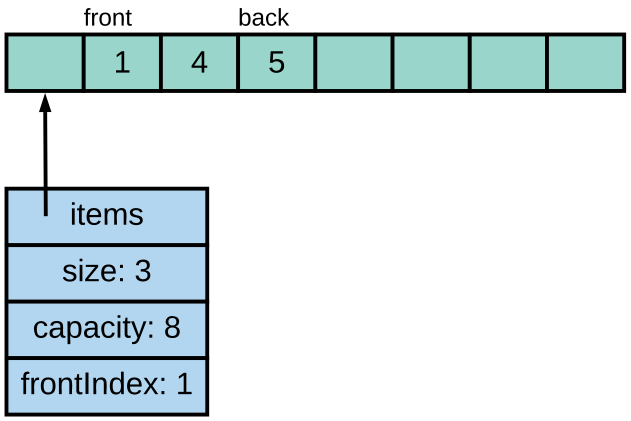
Just like with the ArrayQueue, in these diagrams, when an array index is empty, that simply means that the value at that index is irrelevant, so it could contain any value.



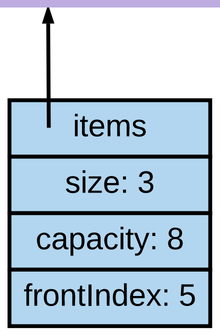
After enqueuing 5:



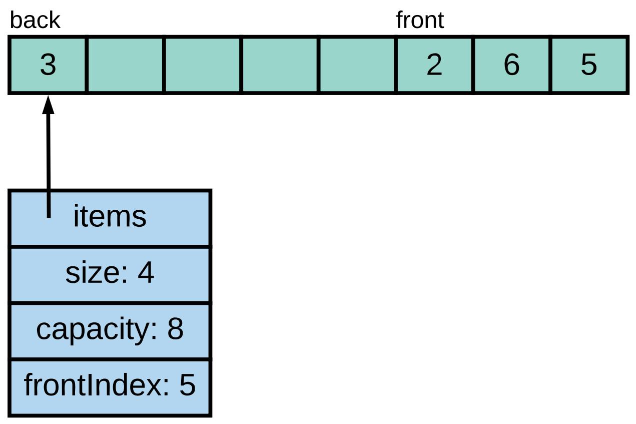
After dequeuing 3:



When enough enqueues and dequeues are performed, the back of the queue will wrap around the end of the array, as shown in the following diagrams:



After enqueuing 3:



Note that when the array becomes full, we will still need to resize the array if another item needs to be enqueued. One of the tasks in this lab will involve working out what needs to be done during a resizing.

Setting Up

Create a directory for this lab, change into it, and run the following command:

```
$ unzip /web/cs2521/23T2/labs/week12/downloads/files.zip
```

If you're working at home, download `files.zip` by clicking on the above link and then run the `unzip` command on the downloaded file.

If you've done the above correctly, you should now have the following files:

`Makefile`	a set of dependencies used to control compilation
`Queue.h`	interface to the Queue ADT
`ListQueue.c`	implementation of the Queue ADT using a linked list (incomplete)

	(incomplete)
`testQueue.c`	tests for the Queue ADT
`runQueue.c`	interactive test program for the Queue ADT

Before you start using these programs, it's worth having a quick look at the code, especially the Queue ADT interface and its various implementations. If there are any constructs you don't understand, ask your tutor.

Once you've understood the programs, run the command:

```
$ make
```

This will leave these executable files in your working directory (along with some `.o` files):

`testListQueue`, `testArrayQueue`, `testCircularArrayQueue`

These executables run tests on each of the queue implementations. All of the tests come from the same source file `testQueue.c` - you should read this file to see what the tests do. Currently, only `testArrayQueue` passes all the tests, as the `ListQueue` and `CircularArrayQueue` implementations are incomplete. Once you've got both of them working, all programs should produce the following output:

```
$ ./testListQueue
Test 1...
Passed!
Test 2...
Passed!
Test 3...
Passed!
Test 4...
Passed!
Test 5...
This is left blank for you to add your own test.
```

As the output suggests, we recommend you to add your own test to the `testQueue5()` function in `testQueue.c`, as the given tests may not cover all the possible edge cases.

`runListQueue`, `runArrayQueue`, `runCircularArrayQueue`

These executables allow you to test the queue implementations interactively by entering commands in the terminal. Here is an example run of the program (once you've got the queue implementations working):

```
$ ./runListQueue
Interactive Queue Tester
```

```
    + <num>                 enqueue an element
    -                       dequeue an element
    f                       get the front element
    s                       get the queue size
    d                       call QueueDebugPrint
    ?                       show this message
    q                       quit

> + 3
Enqueued 3
> + 1
Enqueued 1
> + 4
Enqueued 4
> s
Queue size is 3
> f
Front element is 3
> -
Dequeued 3
> -
Dequeued 1
> f
Front element is 4
> -
Dequeued 4
> s
Queue size is 0
> q
```

Task 1

Complete the `ListQueue` implementation by implementing the enqueuing and dequeuing functions in `ListQueue.c`.

To get an idea of how these functions should behave, you can refer to the diagrams above. **You must implement the functions as described above.**

Once you think you have got the functions working, recompile and run the tests for `ListQueue`:

```
$ ./testListQueue
```

If you get an assertion failed message, that means you didn't pass all the tests, and you'll need to fix your code. For example, here is one error message you might get:

```
$ ./testListQueue
testListQueue: testQueue.c:32: testQueue1: Assertion `QueueSize(q) == 10' failed.
Aborted
```

This particular message says that the assertion on line 32 of `testQueue.c` failed, and hence you should go to line 32 of `testQueue.c`, see what the test does, and then try to figure out where your code is going wrong.

Once you pass tests 1 to 4, you can be fairly certain that your `ListQueue` implementation works.

NOTE:

The Queue ADT provides a debugging function, `QueueDebugPrint`, which you can modify to print any information about the queue that you think would be useful for debugging. You can then call this in your other functions to help debug. Of course, you should remove the calls once you have got everything working.

Task 2

Complete the `CircularArrayQueue` implementation by implementing the enqueuing and dequeuing functions in `CircularArrayQueue.c`.

To get an idea of how these functions should behave, you can refer to the diagrams above. **You must implement the functions as described above.**

Keep in mind that your enqueuing function must be able to handle the situation where the array containing the items is full and needs to be expanded. You may implement the expansion in any way you want, as long as it allows the queue to continue behaving as expected (i.e., items should always be dequeued in the same order as they were enqueued). You *do not* need to reduce the size of the array if the number of items goes below a certain threshold.

A good rule of thumb is to double the size of the array each time it needs to be expanded. You may want to look into *realloc(3)* for expanding the array. (For an example of how to use `realloc`, you can look at `ArrayQueue.c`).

WARNING:

Don't just blindly copy code from `ArrayQueue.c`, as that code is specific to the array queue implementation. You should first *think* about whether the code is appropriate for the circular array queue.

Once you think that you have got the functions working, recompile and run the tests for `CircularArrayQueue`:

```
$ make
...
$ ./testCircularArrayQueue
```

Unlike in the previous task, passing tests 1 to 4 does not guarantee that your implementation is correct. We strongly recommend that you add at least one more test to `testQueue.c` to cover some additional cases, especially those that exercise the array-resizing part of your implementation.

You may add as many tests or as few tests as you like. You will not be submitting your tests, and you will not be marked on your tests. However, note that the given tests do not cover all the cases that your code is expected to handle, so if you choose not to write any tests, you may not know whether your code handles these cases correctly.

COMP2521 23T2: Data Structures and Algorithms is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs2521@cse.unsw.edu.au
CRICOS Provider 00098G

Extra Lab Exercise

Binary Search Trees and Student Records

Objectives

- To learn how structs can be stored in binary search trees
- To get some practice with binary search trees
- To explore a use of function pointers

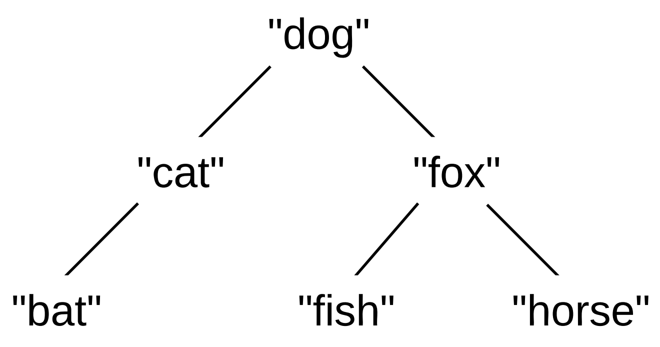
Admin

This lab is not marked and there is no submission for it.

Background

Comparison Functions

So far, most of the examples you have seen of binary search trees have been of integers. But binary search trees don't have to contain integers - any data type whose values can be ordered from smallest to largest can be stored in a binary search tree. For example, here is an example of a binary search tree that contains strings:



programming languages (such as Python), the same operators can be used to compare strings, but in C, string comparison is usually performed using the `strcmp` function. This function takes in the two strings to be compared and returns:

- a negative integer if the first string is lexicographically (alphabetically) less than the second string
- zero if the two strings are equal (i.e., they contain the exact same sequence of characters)
- a positive integer if the first string is lexicographically greater than the second string

This kind of function that takes in two values of the same data type and "compares" them by returning a negative, zero or positive integer is typically called a *comparison function* or comparator. `strcmp`, mentioned above, is a comparison function for strings.

In the context of binary search trees, the return value of a comparison function is very useful, as it determines how we should proceed when inserting into or searching a binary search tree. Suppose we have a binary search tree of strings. If we call `strcmp(s, n->value)`, where `s` is the string being inserted/searched for and `n->value` is the string value in the current node, a negative return value indicates that we should go left (as `s` is "less than" the string in the current node), a positive return value indicates that we should go right, and a return value of zero indicates that `s` is already in the tree. Here is what a search function for a tree of strings might look like:

```c
bool doTreeSearch(Node n, char *s) {
    // not found
    if (n == NULL) {
        return false;
    }

    int cmp = strcmp(s, n->value);
    if (cmp < 0) {
        return doTreeSearch(n->left, s);
    } else if (cmp > 0) {
        return doTreeSearch(n->right, s);
    } else { // (cmp == 0)
        return true;
    }
}
```

We can write comparison functions for any data type. We can even write one for integers, even though integers can already be easily compared with the < and > operators.

```
        return -1;
    } else if (a > b) {
        return  1;
    } else {
        return  0;
    }
}
```

Why not just return `a - b`? Because `a - b` is susceptible to integer overflow/underflows!

Comparison functions are especially useful when we have defined our own data type (by defining our own struct), because they allow us to isolate the logic for comparisons (which could be complex) into a separate function, which improves readability. It also allows us to take advantage of function pointers if we have defined multiple comparison functions for the same data type, which we will see later.

A Custom Data Type - The Student Record

Suppose we have defined our own student record data type that consists of a zid and name. Its definition looks like:

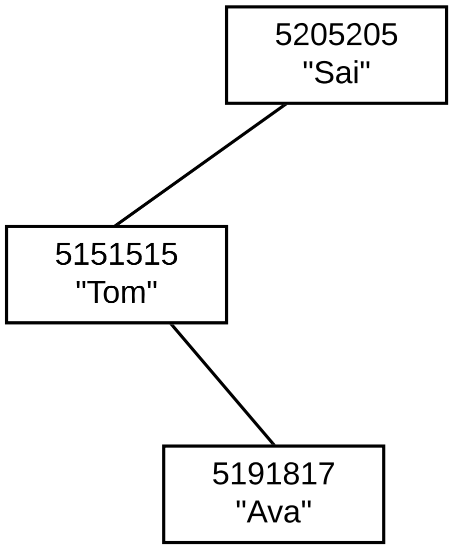
```
struct record {
    int  zid;      // must be between 1 and 9999999
    char name[16]; // must be at most 15 chars in length
};
```

Suppose we now want to implement a binary search tree of records, ordered on zid, for efficient searching. How do we define the comparison function for this tree? Firstly, since the binary search tree is ordered on zid and zids are unique (we'll never have two students with the same zid), the comparison function will never need to be concerned with names. Furthermore, since zids are integers, our comparison function should be similar to a comparison function for integers, except that it takes in records, rather than integers. Here is one possible implementation, based on the `compareInts` function above:

```
int compareByZid(struct record a, struct record b) {
    if (a.zid < b.zid) {
        return -1;
    } else if (a.zid > b.zid) {
        return  1;
    } else {
        return  0;
    }
}
```

```c
int compareByZid(struct record a, struct record b) {
    return a.zid - b.zid;
}
```

With this comparison function, we can now easily implement the search, insertion and deletion algorithms for our tree.



Records ordered on zid

We can now search efficiently by zid. But what if we want to be able to search by name? The first tree won't be able to help us, as it is ordered on zid. We would need another tree ordered on the name field, which means we need another comparison function! Think about how you would implement this function. It should have the same interface as `compareByZid`:

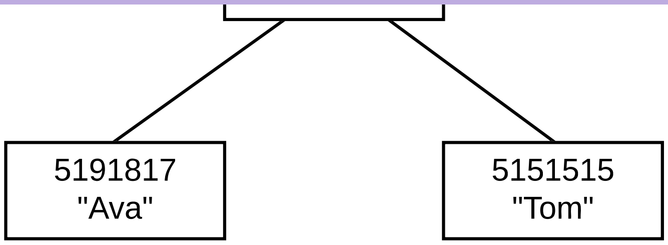
```c
int compareByName(struct record a, struct record b);
```

If you thought of this comparison function:

```c
int compareByName(struct record a, struct record b) {
    return strcmp(a.name, b.name);
}
```

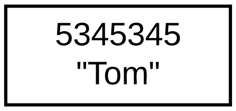
You're on the right track, but there is a problem. Zids are unique, so if `compareByZid` returns 0, then we *know* we have found the right record, as there can't be another record with the same zid. On the other hand, names are *not* unique, so it is possible for `compareByName` to return 0 for two records that have the same name, but different zids.

Suppose we had the following binary search tree, which uses `compareByName` as its comparison function:



Records ordered on name

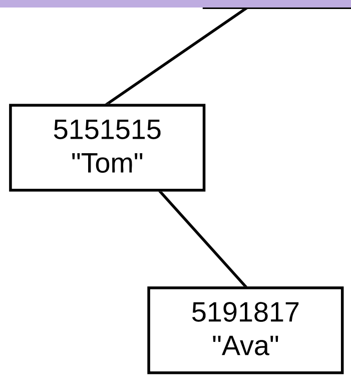
If we wanted to insert the record of another student named "Tom", our insertion function (which uses `compareByName`) would not insert the record, as `compareByName` would say that the new record is equivalent to the existing record with the name "Tom".

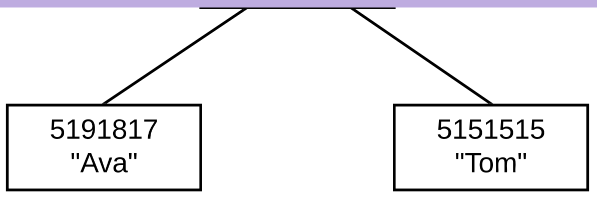


Thus, any comparison function must use a combination of fields/attributes that is guaranteed to be unique. `compareByZid`, which we implemented above, only needs to use the zid field, as zids are unique. For `compareByName`, this means also comparing the records' zids in case the names happened to match. Here is the improved `compareByName`:

```c
int compareByName(struct record a, struct record b) {
    int cmp = strcmp(a.name, b.name);

    // if names are not the same, return the result of strcmp
    if (cmp != 0) {
        return cmp;

    // names are the same, so compare zids
    } else {
        return compareByZid(a, b);
    }
}
```

Now that we have comparison functions for zid and name, we can implement two binary search trees - one ordered on zid and another ordered on name.





Ordered on zid

Ordered on name

Since we now have two trees containing the same data, it is important to note that any operation that modifies one of the trees must be performed on both trees, otherwise there would be inconsistencies. For example, if a record is inserted into the zid-ordered tree, it must also be inserted into the name-ordered tree. The same goes for deletion.

Function Pointers

We can now search records efficiently by zid and by name, but let's have a closer look at the implementation of these trees. Here are the insertion functions:

```c
Node doTreeZidInsert(Node n, struct record
r) {
    if (n == NULL) {
        return newNode(r);
    }

    int cmp = compareByZid(r, n->value);
    if (cmp < 0) {
        n->left = doTreeZidInsert(n->left,
r);
    } else if (cmp > 0) {
        n->right = doTreeZidInsert(n->right,
r);
    }
    return n;
}
```

```c
Node doTreeNameInsert(Node n, struct record
r) {
    if (n == NULL) {
        return newNode(r);
    }

    int cmp = compareByName(r, n->value);
    if (cmp < 0) {
        n->left = doTreeNameInsert(n->left,
r);
    } else if (cmp > 0) {
        n->right = doTreeNameInsert(n-
>right, r);
    }
    return n;
}
```

Can you see a problem with the above code? If you guessed code duplication, you'd be right! Other than the comparison function used, these functions are virtually identical. This problem only becomes worse when we consider that we also need functions for search and deletion.

code duplication), we can let the user decide what comparison function to use by passing in a pointer to the function when they create the tree. To create a tree ordered on zid, the user can pass in a pointer to `compareByZid`, and to create a tree ordered on name, the user can pass in a pointer to `compareByName`.

In this new implementation, each tree struct now contains a pointer to the root node, as well as a pointer to the comparison function that should be used in that tree.

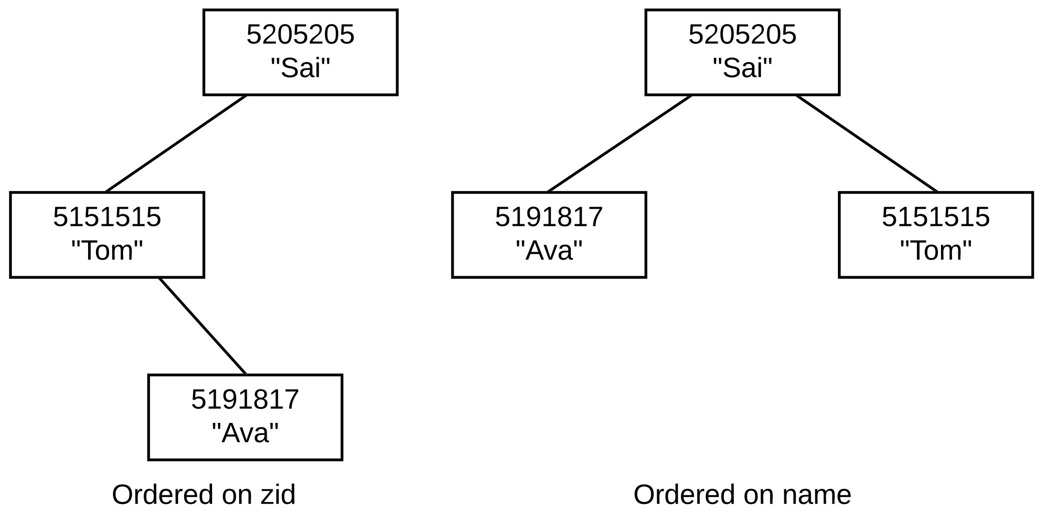
```c
// the tree struct
struct tree {
    Node root;
    int (*compare)(struct record, struct record);
};

Tree TreeNew(int (*compare)(struct record, struct record)) {
    Tree t = malloc(sizeof(*t));
    // malloc error checking here

    t->root = NULL;
    t->compare = compare;
    return t;
}
```

Now that each tree has a configurable comparison function, both of our trees can use the same tree implementation. Here is the new insertion function. Take notice of how the comparison function is called.

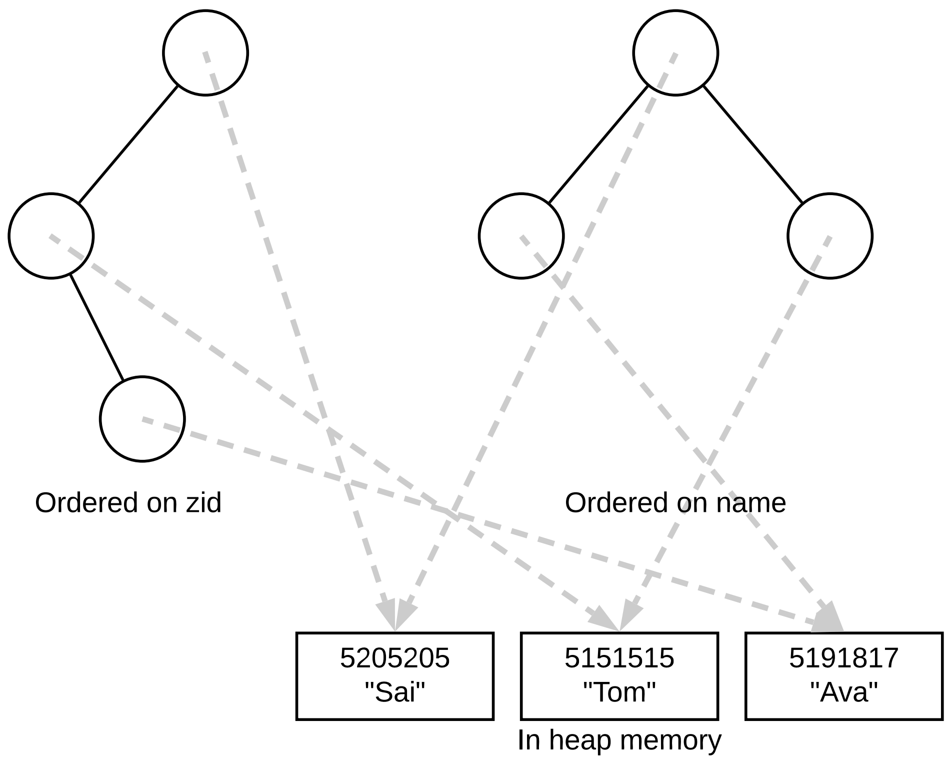
```c
Node doTreeInsert(Tree t, Node n, struct record r) {
    if (n == NULL) {
        return newNode(r);
    }

    int cmp = t->compare(r, n->value);
    if (cmp < 0) {
        n->left = doTreeInsert(t, n->left, r);
    } else if (cmp > 0) {
        n->right = doTreeInsert(t, n->right, r);
    }
    return n;
}
```

Saving Space

multiple copies of each record, which will consume a lot of space.



To save space, we can let each tree node store a pointer to a record rather than the full record itself. That way, only one copy of each record needs to exist.



Setting Up

Create a directory for this lab, change into it, and run the following command:

run the unzip command on the downloaded file.

If you've done the above correctly, you should now have the following files:

Makefile	a set of dependencies used to control compilation
runDb.c	a program that provides a command-line interface to the StudentDb ADT
runTest.c	a program that produces output for a few tests
List.c	implementation of the List ADT (complete)
List.h	interface to the List ADT
Record.c	implementation of the Record ADT (complete)
Record.h	interface to the Record ADT
StudentDb.c	implementation of the StudentDb ADT (incomplete)
StudentDb.h	interface to the StudentDb ADT
Tree.c	implementation of the Tree ADT (incomplete)
Tree.h	interface to the Tree ADT
tests/	a sub-directory containing expected output for some tests

Once you've got these files, the first thing to do is to run the command

```
$ make
```

This will compile the initial version of the files, and produce two executables: ./runDb and ./runTest.

File Walkthrough

runDb.c

runDb.c provides a command-line interface to the StudentDb ADT. It creates a StudentDb object, and then accepts commands to interact with it, including inserting student records into the database, searching for records, and deleting records. Here is an example session with the program:

```
$ ./runDb
StudentDB v1.0
Enter ? to see the list of commands.
> ?
Commands:
    + <zid> <family name> <given name>     add a student record
    fz <zid>                                find a student record by zid
    fn <family name> <given name>           find student records by name
     ?                                      show this message
     q                                      quit

> + 1 Stark Tony
Successfully inserted record!
> + 2 Banner Bruce
Successfully inserted record!
> + 6 Rogers Steve
Successfully inserted record!
> + 4 Pym Hank
Successfully inserted record!
> + 3 Romanoff Natasha
Successfully inserted record!
> lz
1|Stark|Tony
2|Banner|Bruce
3|Romanoff|Natasha
4|Pym|Hank
6|Rogers|Steve
> ln
> fz 1
Found a record:
1|Stark|Tony
> fz 3
Found a record:
3|Romanoff|Natasha
> fn Rogers Steve
No records found
> d 1
Successfully deleted record!
> lz
2|Banner|Bruce
3|Romanoff|Natasha
4|Pym|Hank
6|Rogers|Steve
> fz 1
No records with zid '1'
> q
$
```

`StudentDb.c`

`StudentDb.c` implements the StudentDb ADT, which handles student records. A user can attempt to insert a student record by calling the `DbInsert()` function, and the StudentDb ADT will insert the record if there is not already a record with the same zid. The StudentDb ADT also provides other useful operations, such as deleting a record with a given zid, finding a record with a given zid or name, and listing all the records in order of zid or name.

You should have a read of the functions in `StudentDb.c` to understand how they use the Tree and Record ADTs. Note that at the moment, only the operations involving insertion and zid work.

`Tree.c`

`Tree.c` implements the Tree ADT, which is used to enable efficient searching of records. Notice that the `TreeNew` function (which creates a new tree) takes in a function pointer to a comparison function which determines how the records should be ordered. You should read the `DbNew` function in `StudentDb.c` to see how it passes the comparison functions to `TreeNew`. You should also have a quick read of the search, insertion and deletion algorithms in `Tree.c` to see how they access and call the comparison function.

`List.c`

`List.c` implements the List ADT, which is used to create a list of records. Note that the list has no sorting capabilities, so if you want the records to be ordered in a particular way, you must ensure they are appended in that order.

`Record.c`

`Record.c` implements the Record ADT. You should read the definition of `struct record` to see what a student record consists of, but since this is an ADT, you won't be able to access the fields directly - you'll need to use the `RecordGet`*`XYZ`* functions listed in `Record.h` to access them. This ADT is fully implemented, so you do not need to modify it.

Header files

StudentDb, Tree, List and Record are ADTs, and their associated header files contain extensive descriptions of their interface functions. If you are unsure about what a certain ADT function does or is supposed to do, you should read its description in the relevant header file.

Dummy Records

If you have read through the functions in `StudentDb.c`, you would have noticed the use of so-called "dummy records" in `DbFindByZid` and `DbDeleteByZid`. What are they and why do we need them?

Suppose that the StudentDb contains a number of records and we want to find a record with a given zid. We should search the tree that is ordered on zid, but `TreeSearch` takes in a record, not a zid. So what should we pass into `TreeSearch`?

That is where the dummy record comes in! Because the comparison function used by the zid-ordered tree only inspects the zid of each record (see `compareByZid`), two records that have the same zid will be considered equal, no matter what names they contain. Hence, we can create a dummy record that contains the zid we are searching for, along with some dummy names (an empty string is fine), and pass this into `TreeSearch`. If the tree does contain a record with that zid, it will consider this record as being equal to the dummy record, and return the real record.

This is why it is important that `compareByZid` only compares the zids of the records, and not any other fields. If `compareByZid` also compared the names of the records, this wouldn't work.

Task 1

Implement the `compareByName()` comparison function in `StudentDb.c`. The function should compare the two given records by name (family name first, then given name) and then by zid if the names are equal. Name comparison should be case-sensitive, which means you should use `strcmp`.

When you think you're done, run the following command to test your code:

```
$ ./runTest 1
```

This command runs some tests for the `compareByName` function and prints the results to `stdout`. You should compare the results to the expected output in `tests/1.exp`.

function.

Task 2

First, make sure you have modified the `DbNew` function in `StudentDb.c` so that the `db->byName` tree uses the new comparison function.

Now implement the `TreeSearchBetween()` function in `Tree.c`. This function should search the tree for all records that are considered (by the tree's comparison function) to be between the records `lower` and `upper` (inclusive), and return them all in a list in order. In order for the search to be as efficient as possible, the function should visit as few nodes as possible.

When you think you're done, run the following command to test your code:

```
$ ./runTest 2
```

This command runs some tests for the `TreeSearchBetween` function and prints the results to `stdout`. You should compare the results to the expected output in `tests/2.exp`.

HINT:

We have provided a stub helper function, `doTreeSearchBetween()`, to help you get started. It is recommended that you implement and use this function, but you can write your own helper function if you want. If you want to use the function, you should uncomment it and also uncomment the function prototype at the top of the file.

NOTE:

What does "visit as few nodes as possible" mean? Essentially, it means that you shouldn't visit nodes unnecessarily. A basic in-order traversal that adds the requested records to the list would be extremely easy to implement, but also very inefficient, as it would need to visit all nodes in the tree.

Implement the `DbFindByName()` function in `StudentDb.c`. This function should find all the records that have the same family name *and* given name as the provided family name and given name, and return them in a list in increasing order of zid.

Also implement the `DbListByName()` function in `StudentDb.c`. This function should display all the records in the database in order of name (family name first). It should print the records in the same format as the format produced by the list-by-zid command, for example:

```
2|Banner|Bruce
4|Pym|Hank
6|Rogers|Steve
3|Romanoff|Natasha
1|Stark|Tony
```

HINT:

Use the `TreeSearchBetween` function that you implemented in Task 2. If you're trying to use `TreeSearchBetween` but you are unsure about what `lower` and `upper` should be, think about how you could use dummy records to extract just the range of records that you need (see the Additional Notes section above).

Once you are done, recompile the `runDb` program with the `make` command and run it. Use commands to insert records, search for records (by zid and by name), and list records (by zid and by name). Does your `DbFindByName` function work when there are multiple records with the same name? Do the records get returned in the correct order (i.e., in increasing order of zid)? What if there are no records with the given name? Test thoroughly.

NOTE:

It is possible to speed up testing by entering your commands into a file and making the `runDb` program read in commands from the file. Here's an example command file that tests a very simple case:

```
+ 1 Stark Tony
fn Stark Tony
```

terminal to make it easy to see which operations are being performed.

```
$ ./runDb -e < commands.txt
```

COMP2521 23T2: Data Structures and Algorithms is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs2521@cse.unsw.edu.au
CRICOS Provider 00098G

Extra Lab Exercise

Graphs and Social Networks (Adjacency Matrix)

Objectives

- To explore an application of graphs
- To get some practice with graph problems
- To perform complexity analysis on graph algorithms
- To implement some basic features of social networks

Admin

This lab is not marked and there is no submission for it.

Background

In lectures, we learned that a graph is a collection of vertices and edges between them. This very abstract definition allows for many real-world scenarios and systems to be modelled by graphs - for example, maps, social networks, and the web. In this lab, we will explore an application of graphs in a simple social network app called Friendbook.

Friendbook

Friendbook is a very simple social network app with the following features:

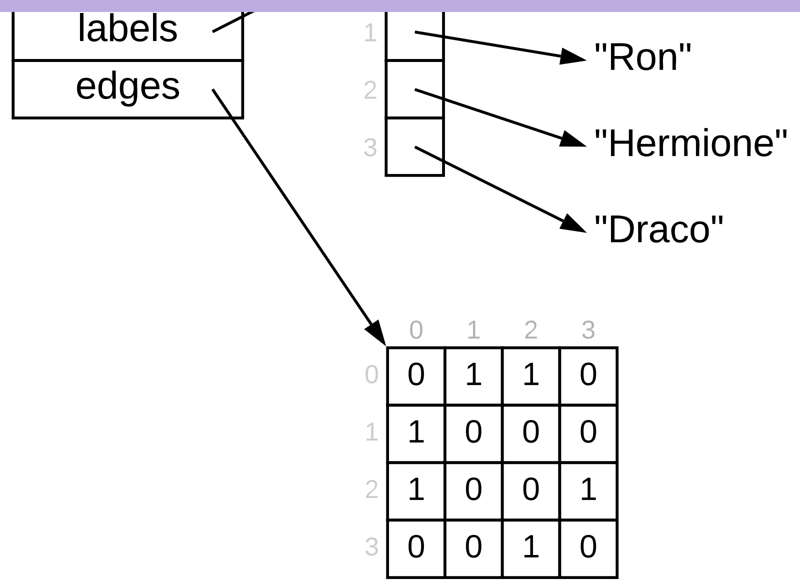
- People can sign up with their name. For simplicity, people are identified by their names, so two people cannot have the same name.
- People can friend other people (i.e., add them as friends). Friending goes both ways, so if you add someone as a friend, you become their friend as well.

- People can see a count of how many friends they have.
- People can see a list of their friends.
- People can see a list of the mutual friends that they share with someone else.
- People can receive friend recommendations. Friendbook has two different methods of generating recommendations:
  1. The first method only recommends friends of friends, and ranks friend recommendations in order of the number of mutual friends, so people who you share more mutual friends with will be recommended first.
  2. The second method recommends friends of friends first, and then friends of friends of friends next, and then friends of friends of friends of friends, and so on. Anyone who can be reached by following friendship links can be recommended.

Names as Vertices

All of the graph implementations we have seen so far have used integer vertices numbered from to , where is the number of vertices. This is convenient, as vertex numbers can double as indices into the adjacency matrix or adjacency list. But in Friendbook, the vertices are people (names), so how do we represent this internally?

It turns out we don't need to do that much more work. If we give each person an integer ID between and and store a mapping between names and IDs, then we can continue to use the graph representations that we are familiar with. A simple way to implement this mapping would be to store all the names in an array, and let the ID of each person be the index containing their name in the array. The first person in the array would have an ID of , the second person in the array would have an ID of , and so on. If we wanted to answer a question involving one or more people, we can scan this array to determine their ID, and then use this ID to query the matrix/list. For example, suppose this is our internal representation:



Now suppose we wanted to find out if Harry and Draco are friends. First, we need to find the vertex numbers associated with Harry and Draco, so we perform a linear scan of the array of names (called `labels`), and find that Harry is associated with a vertex number of , and Draco is associated with a vertex number of . Entry of the adjacency matrix contains a , so we can conclude that Harry and Draco are **not** friends.

Unfortunately, this translation between names and vertex numbers adds quite a bit of overhead to our graph operations. Converting from vertex numbers to names is easy, as we can go straight to the relevant index in the array ( ), but converting from names to vertex numbers requires a linear scan of the array, which is . So what was once an operation (checking if an edge exists) is now an operation. We can improve the efficiency of the name to vertex number conversion by using a data structure that allows for efficient searching, such as a binary search tree.

Setting Up

Create a directory for this lab, change into it, and run the following command:

```
$ unzip /web/cs2521/23T2/labs/week14/downloads/files.zip
```

If you're working at home, download `files.zip` by clicking on the above link and then run the `unzip` command on the downloaded file.

If you've done the above correctly, you should now have the following files:

`Makefile`	a set of dependencies used to control compilation

`List.c`	a complete implementation of the List ADT
`List.h`	the interface for the List ADT
`Map.c`	a complete implementation of the Map ADT
`Map.h`	the interface for the Map ADT
`Queue.c`	a complete implementation of the Queue ADT
`Queue.h`	the interface for the Queue ADT
`runFb.c`	a program that provides a command-line interface to the Friendbook ADT
`analysis.txt`	a template for you to fill in your answers for Task 4

Once you've got these files, the first thing to do is to run the command

```
$ make
```

This will compile the initial version of the files, and produce the `./runFb` executable.

File Walkthrough

runFb.c

`runFb.c` provides a command-line interface to the Friendbook ADT. It creates a Friendbook instance, and then accepts commands to interact with it. Here is an example session with the program:

```
$ ./runFb
Friendbook v1.0
Enter ? to see the list of commands.
> ?
Commands:
    + <name>                add a new person
    l                       list the names of all people
    f <name1> <name2>       friend two people
    u <name1> <name2>       unfriend two people
    s <name1> <name2>       get the friendship status of two people
    n <name>                get the number of friends a person has
    m <name1> <name2>       list all mutual friends of two people
    r <name>                get friend recommendations for a person based on mutual friends
    R <name>                get friend recommendations for a person based on friendship
closeness
    ?                       show this message
    q                       quit
```

```
Ron was successfully added to Friendbook!
> + Hermione
Hermione was successfully added to Friendbook!
> f Harry Ron
Successfully friended Harry and Ron!
> f Ron Hermione
Successfully friended Ron and Hermione!
> s Harry Ron
Harry and Ron are friends.
> u Harry Ron
Could not unfriend Harry and Ron - they are not friends.
> s Harry Ron
Harry and Ron are friends.
> s Harry Hermione
Harry and Hermione are not friends.
> m Harry Hermione
Harry and Hermione's mutual friends:

> r Harry
> R Harry
> q
$
```

Note that the program currently does not correctly unfriend people. It also does not compute mutual friends or generate friend recommendations. In the above example, Ron should be a mutual friend of Harry and Hermione, and Harry should receive Hermione as a friend recommendation. Your task will be to implement these operations.

`Fb.c`

`Fb.c` implements the Friendbook ADT. Most of the functions are complete, however, it would be helpful to read through these functions to get a good idea of how they manipulate and obtain information from the graph representation, how they create and return lists of names, and how they convert people's names to vertex numbers. You should also read the definition of `struct fb` and make sure you understand the purpose of each field.

`List.h`

`List.h` defines the interface to the List ADT. Some operations require a list of names to be returned to the user, and the List ADT is used for this purpose. To see how to create a list and add names to the list, you should read some of the already-completed functions in the Friendbook ADT.

An important thing to note is that the Map ADT is not strictly necessary - it is only used for efficiency reasons. If we didn't have access to the Map ADT and wanted to know the ID of a particular person, we could simply scan the `names` array until we found the index containing that person's name, and their ID would be that index.

`Queue.h`

`Queue.h` defines the interface to the Queue ADT. The Queue ADT is currently not used.

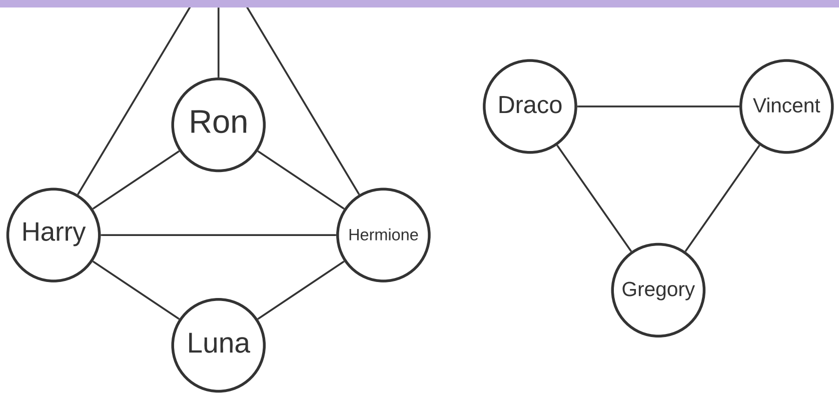
Task 1

Implement the `FbUnfriend()` function in `Fb.c`, which takes the names of two people, and unfriends them if they are friends. The function should return true if the people were friends and were successfully unfriended, and false if the two people were not friends (so they could not be unfriended).

When you think you are done, use the `make` command to recompile the `runFb` program and then run it to test your code.

Task 2

Implement the `FbMutualFriends()` function in `Fb.c`, which takes the names of two people, and returns a list of all their mutual friends. A person is a mutual friend of two people if that person is friends with both of those people. To illustrate this, here is an example:



In the example, Harry and Hermione have three mutual friends: Neville, Ron and Luna. Draco and Vincent have one mutual friend: Gregory. Harry and Draco have no mutual friends.

HINT:

To find out how to create a list and add names to it, see the comments in `List.h`, or read one of the existing functions in `Fb.c` that use the List ADT.

WARNING:

You are forbidden from using the list iterator functions (the functions whose names start with `ListIt`). These functions are meant to be used by `runFb.c` only.

When you think you are done, use the `make` command to recompile the `runFb` program and then develop some scenarios to test your code. You can even use the example from above.

NOTE:

It is possible to speed up testing by entering your commands into a file and making the `runFb` program read in commands from the file. Here's an example command file that tests a very simple case:

```
+ Harry
+ Ron
+ Hermione
f Harry Ron
f Ron Hermione
```

Suppose the file is called `mutual-friends-1.txt`. Then, the following command will run the `runFb` program using the commands from the file and echo the commands to the terminal to make it easy to see which operations are being performed.

```
$ ./runFb -e < mutual-friends-1.txt
```

If you've implemented the function correctly, you should get Ron as the only mutual friend of Harry and Hermione. However, this is a very simple case - you should add more people and friendships to test other cases.

NOTE:

In this task and the next, ensure that you don't introduce memory leaks into the program by freeing data structures that you create once they are no longer needed. For example, if you malloc a temporary array or create a temporary list in your function that is no longer needed after the function returns, you should free it. You can use the `valgrind` command to check for memory leaks:

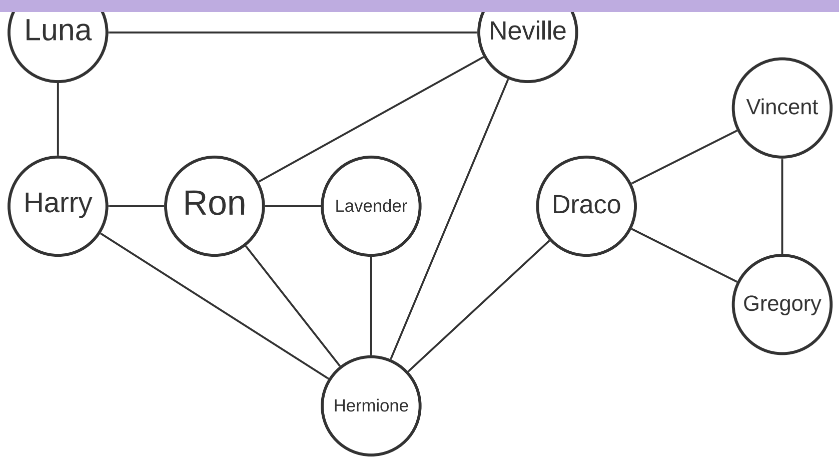
```
$ valgrind -q --leak-check=full ./runFb -e < mutual-friends-1.txt
...
```

If the program contains memory leaks, `valgrind` will report an error after the program exits. You can find out more about `valgrind` and memory leaks in the debugging lab.

Task 3

Implement the `FbFriendRecs1()` function in `Fb.c`, which takes the name of a person and generates and prints friend recommendations for them. This function should only recommend people who are friends of friends of the person. In other words, it should only recommend people who share at least one mutual friend with the person. Obviously, it should not recommend someone who is already the person's friend.

The recommendations should be printed in descending order on the number of mutual friends shared, since someone with more mutual friends is more likely to be known by the person, and is therefore more likely to be added as a friend. If two people share the same number of mutual friends, they may be printed in any order. The number of mutual friends should also be displayed next to each recommendation.



If `FbFriendRecs1()` is called with the name "Harry", the function should produce the following output:

```
Harry's friend recommendations
	Neville                3 mutual friends
	Lavender               2 mutual friends
	Draco                  1 mutual friends
```

Explanation: Neville should be recommended first as he shares three mutual friends with Harry: Luna, Ron and Hermione. Lavender should be recommended next as she shares two mutual friends with Harry: Ron and Hermione. Draco should be recommended last as he shares just one mutual friend with Harry: Hermione. (*Note:* There is no typo on the last line - it is left as "friends" for simplicity's sake.)

When you think you are done, use the `make` command to recompile the `runFb` program and then develop some scenarios to test your code. You can even use the example from above.

NOTE:

To ensure that your output matches the expected output format, use the following `printf` format: `"\t%-20s%4d mutual friends\n"`

HINT:

If you're unsure about how to print the recommendations in sorted order, consider the following: Given that there are people, what is the largest number of mutual friends two people could have? What is the smallest number of mutual friends two people could have? Finally, if you know how many mutual friends each person has, how can

Task 4

Congratulations on completing the above functions! Now, it's time to do some complexity analysis.

Open `analysis.txt`. This file contains three sections - one for each of the functions you implemented. You need to determine the worst case time complexities of each of these functions and enter them in the file under the corresponding section, along with an explanation of your answer.

Important: The Map ADT uses an inefficient binary search tree implementation, but you should assume that it uses an AVL tree for complexity analysis.

HINT:

If you used any functions from the List, Map or Queue ADTs, either directly or indirectly (i.e., via a helper function), the time complexities of these functions will affect the time complexity of your solution. You should consult the header files of these ADTs to find out these time complexities.

Optional Task

NOTE:

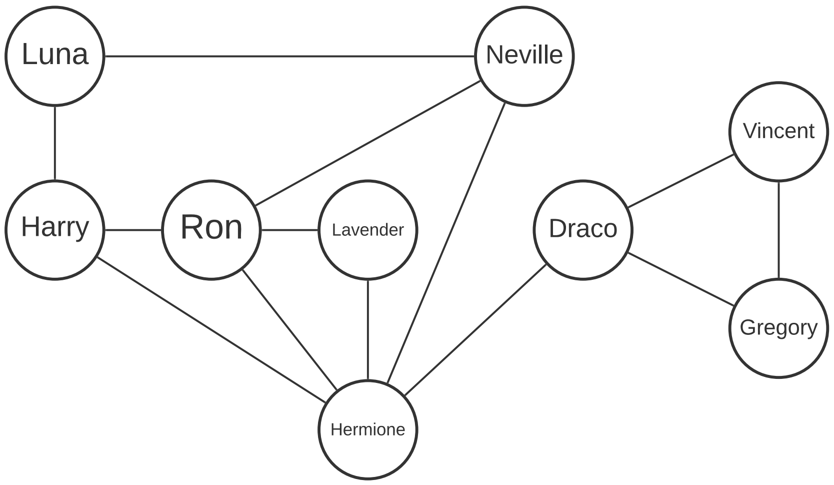
This task is **optional**. It is not worth any marks.

Implement the `FbFriendRecs2()` function in `Fb.c`, which takes the name of a person and generates and prints friend recommendations for them. Unlike `FbFriendRecs1`, this function can recommend all people who are reachable from the person via friendship links (not just people who share a mutual friend), and should recommend people who are "closer" to the person first. In other words, friends of friends of the person should be "distance" from the person, they can be recommended in any order.

Limit the number of recommendations to 20 to avoid generating too many recommendations.

This recommendation method only needs to list people's names - no additional information is required.

For example, consider the same scenario as in Part 2:



If `FbFriendRecs2()` was called with the name "Luna", the following is one possible valid output:

```
Luna's friend recommendations
        Ron
        Hermione
        Draco
        Lavender
        Vincent
        Gregory
```

Explanation: Ron and Hermione are the closest people to Luna who are not also her friends, so they are recommended first. The example output recommends Ron first and then Hermione, but it would be equally valid to recommend Hermione first and then Ron. Draco and Lavender are the next furthest away, so they are recommended next. It would be valid to recommend Lavender before Draco. Vincent and Gregory are the next furthest away, so they are printed next. Once again, it would be valid to recommend Gregory before Vincent.

When you think you are done, use the `make` command to recompile the `runFb` program and then develop some scenarios to test your code.

You will need to use a graph traversal algorithm to complete this task. But which one? You can review the graph traversal algorithms here, and then follow the pseudocode of your chosen algorithm.

HINT:

Use the following `printf` format: `"\t%s\n"`

COMP2521 23T2: Data Structures and Algorithms is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs2521@cse.unsw.edu.au
CRICOS Provider 00098G

Extra Lab Exercise

Directed Graphs and Web Crawlers

Objectives

- To implement a web crawler
- To see how directed graphs might be used with real-world data

Admin

This lab is not marked and there is no submission for it.

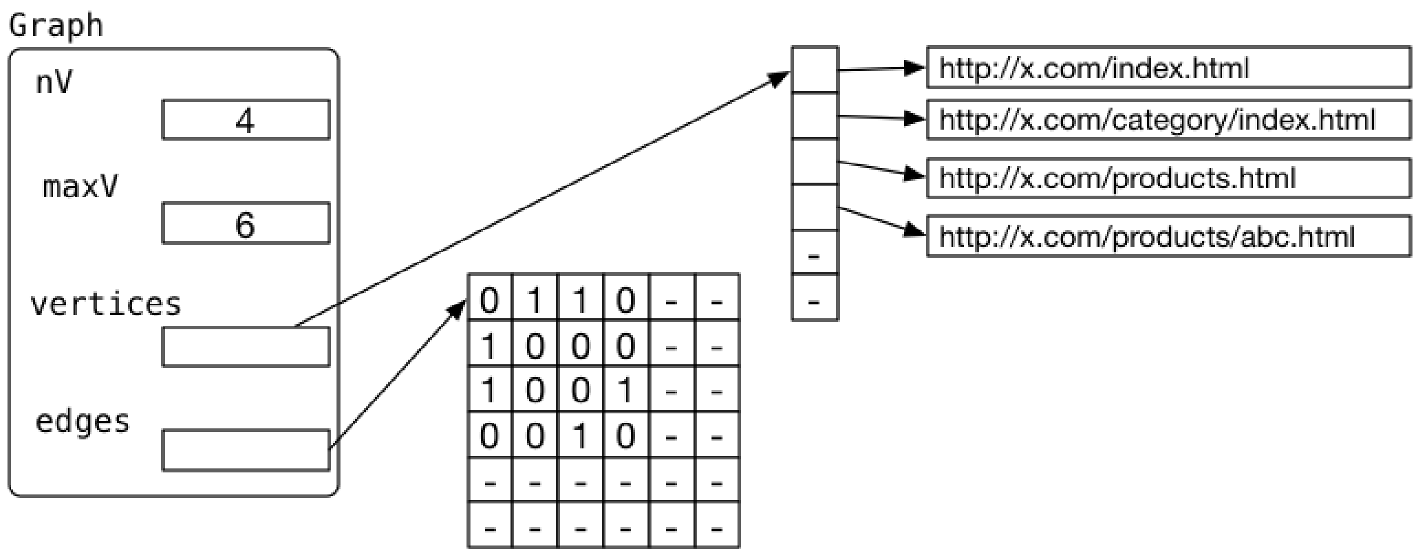
Background

We can view the World Wide Web as a massive directed graph, where *pages* (identified by URLs) are the vertices and *hyperlinks* are the directed edges. Unlike the graphs we have studied in lectures, there is not a single central representation (e.g. adjacency matrix) for all the edges in the graph of the web; such a data structure would clearly be way too large to store. Instead, the "edges" are embedded in the "vertices". Despite the unusual representation, the Web is clearly a graph, so the aim of this lab exercise is to build an in-memory graph structure for a very, very small subset of the Web.

Web crawlers are programs that navigate the Web automatically, moving from page to page, processing each page they visit. Crawlers typically use a standard graph traversal algorithm to:

- maintain a list of pages to visit (a ToDo list)
- "visit" the next page by grabbing its HTML content
- scan the HTML to extract whatever features they are interested in
- collect hyperlinks from the visited page, and add these to the ToDo list
- repeat the above steps (until there are no more pages to visit :-)

You need to implement such a crawler, using a collection of supplied ADTs and a partially complete main program. Your crawler processes each page by finding any hyperlinks and inserting the implied edges into a directed `Graph` ADT based on an adjacency matrix. One difference between this `Graph` ADT and the ones we have looked at in lectures is that you


Graph
nV
4
maxV
6
vertices
edges
http://x.com/index.html
http://x.com/category/index.html
http://x.com/products.html
http://x.com/products/abc.html

0	1	1	0	-	-
1	0	0	0	-	-
1	0	0	1	-	-
0	0	1	0	-	-
-	-	-	-	-	-
-	-	-	-	-	-

The `Graph` data structure allows for `maxV` vertices (URLs), where `maxV` is supplied when graph is created. Initially, there are no vertices or edges, but as the crawler examines the web, it adds the URLs of any pages that it visits and records the hyperlinks between them. This diagram shows what a web crawler might have discovered had it started crawling from the URL `http://x.com/index.html`, and so far examined four web pages.

If we number the four pages from 0..3, with

- page (vertex) 0 being `http://x.com/index.html`
- page (vertex) 1 being `http://x.com/category/index.html`
- page (vertex) 2 being `http://x.com/products.html`
- page (vertex) 3 being `http://x.com/products/abc.html`

The `vertices` array holds the actual URL strings and also, effectively, provides the mapping between URLs and vertex numbers. The `edges` array is a standard adjacency matrix. The top row tells us that page 0 has hyperlinks to pages 1 and 2. The second row tells us that page 1 has a hyperlink back to page 0. The third row similarly shows a hyperlink from page 2 back to page 0, but also a hyperlink to page 3.

Setting Up

Set up a directory for this lab, change into that directory, and run the following command:

```
$ unzip /web/cs2521/23T2/labs/week15/downloads/files.zip
```

If you're working at home, download `files.zip` by clicking on the above link and then run the `unzip` command on the downloaded file.

`crawl.c`	a main program that implements a web crawler (incomplete)
`url_file.h`, `url_file.c`	provides a file-like interface to webpages
`html.h`, `html.c`	provides functions for extracting URLs from HTML
`Graph.h`, `Graph.c`	a Graph ADT interface and implementation
`Queue.h`, `Queue.c`	a Queue ADT interface and implementation
`Stack.h`, `Stack.c`	a Stack ADT interface and implementation
`Set.h`, `Set.c`	a Set ADT interface and implementation

The only file you need to modify is `crawl.c`, but you need to understand at least the interfaces to the functions in the various ADTs. This is described in comments in the `.h` files. You can also see examples of using the ADT functions in the `t?.c` files. Note that there's no test file for the HTML parsing and URL-extracting code, because the supplied version of `crawl.c` effectively provides this.

Note that HTML parsing code was borrowed from Dartmouth College. If you looks at the code, it has quite a different style to the rest of the code. This provides an interesting comparison with our code.

The `crawl` program is used as follows:

```
$ ./crawl StartingURL MaxURLsInGraph
```

The *StartingURL* tells you which URL to start the crawl from, and should be of the form `http://x.y.z/`. The crawler uses this URL as both the starting point and uses a normalised version as the base against which to interpret other URLs.

The *MaxURLsInGraph* specifies the maximum number of URLs that can be stored in the `Graph`. Once this many URLs have been scanned, the crawling will stop, or will stop earlier if there are no more URLs left in the ToDo list.

If you compile then run the supplied crawler on the UNSW handbook, you would see something like:

```
$ ./crawl http://www.handbook.unsw.edu.au/2017/ 100
Found: 'http://www.unsw.edu.au'
Found: 'https://my.unsw.edu.au/'
Found: 'https://student.unsw.edu.au/'
Found: 'http://www.futurestudents.unsw.edu.au/'
Found: 'http://timetable.unsw.edu.au/'
Found: 'https://student.unsw.edu.au/node/62'
Found: 'http://www.library.unsw.edu.au/'
Found: 'http://www.handbook.unsw.edu.au/2017/'
Found: 'http://www.unsw.edu.au/faculties'
Found: 'https://student.unsw.edu.au/node/4431'
Found: 'https://student.unsw.edu.au/node/128'
```

```
Found: 'http://www.handbook.unsw.edu.au/postgraduate/2017/'
Found: 'http://www.handbook.unsw.edu.au/research/2017/'
Found: 'http://www.handbook.unsw.edu.au/nonaward/2017/'
Found: 'http://www.unsw.edu.au/future-students/domestic-undergraduate'
Found: 'http://www.unsw.edu.au/future-students/postgraduate-coursework'
Found: 'http://research.unsw.edu.au/future-students'
Found: 'http://www.international.unsw.edu.au/#1'
Found: 'https://student.unsw.edu.au/node/1334'
Found: 'https://moodle.telt.unsw.edu.au/login/index.php'
Found: 'https://student.unsw.edu.au/node/943'
Found: 'https://apply.unsw.edu.au/'
Found: 'https://student.unsw.edu.au/node/5450'
Found: 'http://cgi.cse.unsw.edu.au/~nss/feest/'
Found: 'http://www.unsw.edu.au/privacy'
Found: 'http://www.unsw.edu.au/copyright-disclaimer'
Found: 'http://www.unsw.edu.au/accessibility'
```

The supplied crawler simply scans the URL given on the command line, prints out any URLs that it finds, and then stops. It does not attempt to traverse any further than the supplied page. The second command-line argument, which limits the size of the `Graph`, is effectively ignored here, since the supplied code does not build a `Graph`; you need to add the code to do this.

If you run the supplied "crawler" on `http://www.cse.unsw.edu.au`, you don't get very much, because the CSE website recently moved under the Engineering Faculty system and the above URL is just a redirection page to the new site. Copying/pasting the redirection URL gives you more interesting results. Before you go running the "crawler" on other websites ... DON'T! See the comments below.

HTML is a language which is difficult to parse given the way it is frequently used, and the `GetNextURL()` make some approximations which, while they wouldn't be acceptable in a *real* Web crawler, are OK for this lab.

Exercise

Your task is to modify `crawl.c` so that it follows any URLs it finds and builds a `Graph` of the small region of the Web that it traverses before bumping in to the `MaxURLsInGraph` limit.

Important: running crawlers outside the UNSW domain is problematic. Running crawlers that make very frequent URL requests is problematic. So...

- DO NOT run your crawler on any website outside UNSW
- YOU MUST include a delay (`sleep(1)`) between each URL access

disciplinary action.

Your crawler can do either a breadth-first or depth-first search, and should follow roughly the graph traversal strategy described in lectures and tutes:

```
add firstURL to the ToDo list
initialise Graph to hold up to maxURLs
initialise set of Seen URLs
while (ToDo list not empty and Graph not filled) {
    grab Next URL from ToDo list
    if (not allowed) continue
    foreach line in the opened URL {
        foreach URL on that line {
            if (Graph not filled  or both URLs in Graph)
                add an edge from Next to this URL
            if (this URL not Seen already) {
                add it to the Seen set
                add it to the ToDo list
            }
        }
    }
    close the opened URL
    sleep(1)
}
```

This does not give all the details. You need to work them out yourself, based on the supplied ADT functions and your understanding of generic graph traversal. If you use a `Stack` for the ToDo list, your crawler will end up doing a depth-first search. If you use a `Queue` for the ToDo list, your crawler will end up doing a breadth-first search.

A couple more things to note:

- `(not allowed)` refers to not using URLs outside UNSW
- the `ToDo list` is a `Stack` or `Queue` rather than a List
- if you don't include the `sleep(1)` you will smash whatever web server you access

If you test the crawler out on `www.cse.unsw.edu.au`, you don't get particularly interesting results, because you'll build a large adjacency matrix, most of which is empty, before you bump into `MaxURLsInGraph`. To assist in doing some feasible crawling and getting some more interesting output, I have set up a tiny set of self-contained web pages that you can crawl, starting from:

```
$ ./crawl https://cgi.cse.unsw.edu.au/~cs2521/mini-web/ 30
```

You should use `GraphShow()` to check whether you are building a plausible `Graph`. Note that `GraphShow()` has two modes:

- `GraphShow(g, 0)` shows the URL for each vertex, followed by an indented list of connected vertices

Challenges

There are several aspects of the crawler that you could look to improve:

- The existing crawler grabs all sorts of URLs that do not represent Web pages. Modify the code so that it filters out non-HTML output.
- The supplied `GetNextURL()` function does a reasonable job on finding URLs, but doesn't handle relative URLs. Find online or write your own, or modify the existing code, to make a new `GetNextURL()` that handles a wider range of URLs.
- Modify `GraphShow()` so that it can (also) produce output (JSON) that could be fed into a graph drawing tool such as sigmajs, to produce beautiful graph diagrams.

COMP2521 23T2: Data Structures and Algorithms is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs2521@cse.unsw.edu.au
CRICOS Provider 00098G

Extra Lab Exercise

Weighted Graphs and Geo Data

Objectives

- To implement a variant of path finding in a weighted graph
- To see how graphs might be used with real-world data

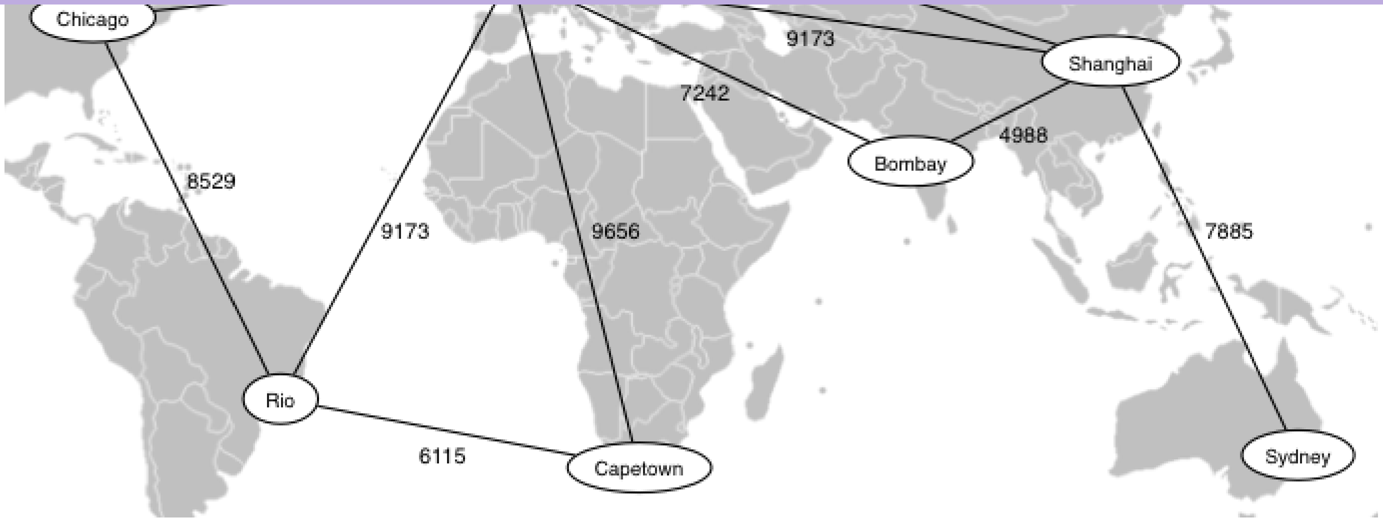
Admin

This lab is not marked and there is no submission for it.

Background

Geographic data is widely available, thanks to sites such as GeoNames. For this lab, we have downloaded data files from the City Distance Dataset by John Burkardt from the Department of Scientific Computing at Florida State University. The dataset that we will use contains information about distances between 30 prominent cities/locations around the world. These are great-circle distances; we'll assume that these are the distances that an aircraft would fly between two cities. The data that we have forms a complete graph in that there is a distance recorded for every pair of cities.

The following diagram shows a subset of the data from the City Distance Dataset.



Map from "BlankMap-World-v2" by original uploader: Roke
Own work. Licensed under Creative Commons Attribution-Share Alike 3.0 via Wikimedia Commons

The data comes in two files:

`ha30_dist.txt`

This file contains a matrix of distances between cities. This is essentially the adjacency matrix for a weighted graph where the vertices are cities and the edge weights correspond to distances between cities. As you would expect for an adjacency matrix, the leading diagonal contains all zeroes (in this case, corresponding to the fact that a city has a distance of zero from itself).

`ha30_name.txt`

This file contains one city name per line. If we number the lines starting from zero, then the line number corresponds to the vertex number for the city on that line. For example, the Azores is on line 0, so vertex 0 corresponds to the Azores, and the first line in the distance file gives distances from the Azores to the other 29 cities. The last line (line 29) tells us that Tokyo is vertex 29, and the last line in the distance files gives distances between Tokyo and all other cities.

Setting Up

Set up a directory for this lab, change into that directory, and run the following command:

```
$ unzip /web/cs2521/23T2/labs/week16/downloads/files.zip
```

If you've done the above correctly, you should now have the following files:

`Makefile`	a set of dependencies used to control compilation
`travel.c`	main program to load and manipulate the graph
`Graph.h`	the interface for the Graph ADT
`Graph.c`	an incomplete implementation of the Graph ADT
`Queue.h`	the interface for the Queue ADT
`Queue.c`	a complete implementation of the Queue ADT
`ha30_name.txt`	the city name file described above
`ha30_dist.txt`	the distance matrix file described above

The `Makefile` produces a file called `travel` based on the main program in `travel.c` and the functions in `Graph.c`. The `travel` program takes either zero or two command line arguments:

```
$ ./travel
... displays the entire graph ...
... produces lots of output, so either redirect to a file or use less ...
$ ./travel from-city to-city
... display a route to fly between specified cities ...
```

If given zero arguments, it simply displays the graph. If given two arguments, it treats the first city as a starting point and the second city as a destination, and determines a route between the two cities, based on "hops" between cities with direct flights.

Read the `main()` function so that you understand how it works, and, in particular, how it invokes the functions that you need to implement.

The `Graph` ADT in this week's lab has a `GraphRep` data structure that is a standard adjacency matrix representation of the kind we looked at in lectures. However, some aspects of it are different to the `GraphRep` from lectures.

Note that city names are not stored as part of the `GraphRep` data structure. The `Graph` ADT deals with vertices using their numeric ID. The `main` program maintains the list of city names and passes this list to the `showGraph()` function when it is called to display the graph. This means that the calling interface for the `showGraph()` function is different to the `showGraph()` function from the `Graph` ADT in lectures.

between vertices. In other words, we're dealing with a *weighted* graph. A weight value of zero indicates that there is no edge between two vertices, while a non-zero weight indicates that there is an edge.

The main program makes some changes to the edges implied by the distance matrix as it copies them into the `Graph`. The values in the `ha30_dist.txt` file are given in units of "hundreds of miles"; we want them in units of kilometres so each distance is converted before it is added to the graph as the weight of an edge. Since every city has a distance to every other city (except itself), this gives us a *complete graph*.

As supplied, `Graph.c` has an incomplete implementation of the `findPath()` function. If you compile the `travel` program and try to find any route, it will simply tell you that there isn't one. Your task will be to implement this function.

Exercise

Implement the `findPath(g, src, dest, max, path)` function. This function takes a graph `g`, two vertex numbers `src` and `dest`, a maximum flight distance `max`, and fills the `path` array with a sequence of vertex numbers giving the "shortest" path from `src` to `dest` such that no edge on the path has weight larger than `max`. The function returns the number of vertices stored in the `path` array; if it cannot find a path, it returns zero. The `path` array is assumed to have enough space to hold the longest possible path (which would include all vertices).

This could be solved with a standard BFS graph traversal algorithm, but there are two twists for this application:

- The edges in the graph represent real distances, but the user of the `travel` program (the traveller) isn't necessarily worried about real distances. They are more worried about the number of take-offs and landings (which they find scary), so the length of a path is measured in terms of the number of edges, *not* the sum of the edge weights. Thus, the "shortest" path is the one with the minimum number of edges.
- While the traveller isn't concerned about how far a single flight goes, aircrafts *are* affected by this (because they hold a limited amount of fuel). The `max` parameter for `findPath()` allows a user to specify that they only want to consider flights whose length is at most `max` kilometres (i.e. only edges whose weight is not more than `max`).

Note that the default value for `max`, set in the `main` function, is 10000km. Making the maximum flight distance smaller produces more interesting paths (see below), but if you make it too small (e.g., 5000km) you will end up isolating Australia from the rest of the world. With a maximum flight distance of 6000km, the only way out of Australia is via Guam. If you make the maximum flight distance large enough (e.g., possibly reflecting an improvement in aircraft technology), every city will be reachable from every other city in a single hop.

Here are some example routes (don't expect them to closely match reality):

```
# when no max distance is given on the command line,
# we assume that planes can fly up to 10000km before refuelling
$ ./travel Berlin Chicago
Least-hops route:
Berlin
-> Chicago
$ ./travel Sydney Chicago
Least-hops route:
Sydney
-> Honolulu
-> Chicago
$ ./travel Sydney London
Least-hops route:
Sydney
-> Shanghai
-> London
$ ./travel London Sydney
Least-hops route:
London
-> Shanghai
-> Sydney
$ ./travel Sydney 'Buenos Aires'
Least-hops route:
Sydney
-> Honolulu
-> Chicago
-> Buenos Aires
# if planes can fly up to  6000km before refuelling
$ ./travel Sydney London 6000
Least-hops route:
Sydney
-> Guam
-> Manila
```

```
# if planes can fly up to  5000km before refuelling
# you can't leave Australia under this scenario
$ ./travel Sydney 'Buenos Aires' 5000
No route from Sydney to Buenos Aires
# if planes can fly up to  7000km before refuelling
$ ./travel Sydney 'Buenos Aires' 7000
Least-hops route:
Sydney
-> Guam
-> Honolulu
-> Chicago
-> Panama City
-> Buenos Aires
# if planes can fly up to  8000km before refuelling
$ ./travel Sydney 'Buenos Aires' 8000
Least-hops route:
Sydney
-> Guam
-> Honolulu
-> Mexico City
-> Buenos Aires
# if planes can fly up to 11000km before refuelling
$ ./travel Sydney 'Buenos Aires' 11000
Least-hops route:
Sydney
-> Bombay
-> Azores
-> Buenos Aires
# if planes can fly up to 12000km before refuelling
# we can reach Buenos Aires in a single flight
$ ./travel Sydney 'Buenos Aires' 12000
Least-hops route:
Sydney
-> Buenos Aires
$ ./travel Bombay Chicago 5000
Least-hops route:
Bombay
-> Istanbul
-> Azores
-> Montreal
-> Chicago
$ ./travel Sydney Sydney
```

The above routes were generated using an algorithm that checks vertices in numeric order (vertex 0 before vertex 1 before vertex 2, etc.). If you check vertices in a different order, you may generate different, but possibly equally valid, routes.

COMP2521 23T2: Data Structures and Algorithms is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs2521@cse.unsw.edu.au
CRICOS Provider 00098G

Extra Lab Exercise

Hashing and Profiling

Objectives

- To learn about hash tables and the effectiveness of hash functions
- To learn about analysis of program performance via profiling

Admin

This lab is not marked and there is no submission for it.

Background

In lectures, we examined hashing as a key-based search/storage data structure. Under ideal circumstances, hashing can give access to stored items via a key value. Two critical components in a hashing scheme were: the hash function (which converts key values into indexes), and the collision resolution method (which deals with several keys hashing to the same index value). In this lab, we deal with a hash table implementation that uses Sedgewick's "universal" hashing function on character strings, and uses the separate chaining mechanism for collision resolution.

Setting Up

Set up a directory for this lab, change into that directory, and run the following command:

```
$ unzip /web/cs2521/23T2/labs/week17/downloads/files.zip
```

If you've done the above correctly, you should now have the following files:

`Makefile`	a set of dependencies used to control compilation
`words.c`	a main program that reads words and builds a hash table from the words
`HashTable.h`	the interface for the Hash Table ADT
`HashTable.c`	an incomplete implementation of the Hash Table ADT
`List.h`	the interface for the List ADT
`List.c`	a complete implementation of the List ADT
`Item.h`	the interface for the Item ADT
`Item.c`	a complete implementation of the Item ADT
`mkwords.c`	a main program that generates random words

Compiling with make will produce two executables: `mkwords` and `words`. The `mkwords` program is fully functional and produces sequences of words using a random number generator with an (optional) seed. For example:

```
$ ./mkwords 10 3
allpnl
ahebpveeloatic
ualoubyy
hssaif
rywt
tiehaelsh
oheom
vmoe
jeabzsa
zqa
```

The example above produces 10 "words" using a random number generator seed of 3. If you want to generate really random (i.e. unreproduceable) word lists, don't supply a seed parameter and mkwords will use an effectively random seed.

Note that mkwords will inevitably produce duplicate words for any reasonable-sized N; the insertion function for the hash table checks for and handles duplicates.

The words program also executes to completion, but since the `HashTableStats()` function is incomplete, you won't get particularly interesting output. See below for examples of what the output from words looks like.

You should begin by looking primarily at the code in these files:

table, and then prints statistics about the hash table. After inserting each word into the hash table, the program immediately searches for it, to make sure that it was actually inserted. It then searches for several "words" not in the input; these should not be found in the hash table. Finally, it clears all of the memory used by the hash table and exits.

The words program takes two command-line parameters: the name of the file to read from; the number of slots in the hash table (best if this is a prime number).

The words program can also read from its standard input if the filename is given as a single minus sign. However, when reading from standard input, it only performs the tests for words not in the input.

`HashTable.c`

This provides an implementation of a hash table that uses separate chaining for collision resolution. The hash table consists of an indexed array of lists, which are based on the `List` data type. The core hash table functions (insert, delete, search) are all quite simple, and consist of using the hash function to find the appropriate list, and then carrying out the relevant operation using the appropriate `List` function. All of the hash table function are complete except for `HashTableStats()`.

Other relevant code is in `List.c` which provides a standard implementation of a linked list of items, which you can assume is correct. Similarly, the file `Item.c` contains an implementation for items; normally, this would be done simply as a set of macros, but we have used functions for some of the operations to create entries in the profile.

Task 1

Your first task is to complete the `HashTableStats()` function in `HashTable.c`. This function should print the following information about the hash table:

- the number of slots in the hash table array
- the number of items stored in the (lists of the) hash table
- information about the lengths of chains in a table containing
  - the length of chains (use zero length for unused array slots)
  - the number (frequency) of chains with this length

The table should have a row for each chain length from 0 up to the maximum chain length. You'll need to work out the maximum chain length, and then allocate an array of counters of the appropriate size.

```
$ ./words wordsfile 1777
Reading words from wordsfile
Hash Table Stats:
Number of slots = 1777
Number of items = 1970
Chain length distribution
  Length  #Chains
       0      585
       1      657
       2      351
       3      139
       4       33
       5       11
       7        1
Testing completed OK
```

Note that you could produce the same output, without needing an intermediate file, using the command:

```
$ ./mkwords 2000 13 | ./words - 1777
```

The above commands insert 2000 words (1970 distinct words) into a hash table with 1777 slots. The output tells us that there are 585 unused slots in the hash table (chain length 0), and 657 slots with chains of length 1, etc. If there are no chains of a given length, then nothing is written for that length, e.g.

```
$ ./mkwords 1000 7 | ./words - 101
Reading words from stdin
Hash Table Stats:
Number of slots = 101
Number of items = 991
Chain length distribution
  Length  #Chains
       2        1
       4        2
       5        2
       6        6
       7       16
       8       12
       9       13
      10       15
      11        8
      12        6
      13        5
      17        1
      18        2
      19        1
Testing completed OK
$
```

This output is for a small hash table with just 101 slots. Since there is no entry for 0, all of the slots in the hash tables have been used (i.e. no slots with zero items). Since there are no entries for 1 and 3, this tells us there are no chains of length 1 or 3; this is not in itself interesting, and is just way that it works out for this data. Many of the slots have overflow chains of length 10; hits on any of these hash table entries will result in us examining up to 10 items. We can also see that the longest chain contains 19 items. Ideally, we would like each chain to contain a small number of items; this means that the hash table needs to have a number of slots that is proportional to the number of items (e.g. if the has function works perfectly and we have *n* items and *n/3* slots, we'd expect each chain to be of length ~3).

Execution Profiling

Execution profilers measure statistics like the number of times each function is called and the average time spent executing each function, in order to identify which functions are contributing most to the cost of executing the program. Raw timing data (e.g. using the `time` command) gives you an overview of how fast a program is, but an execution profile gives you detailed information about where it's spending all of its time, and thus where you should focus any attempts at improving its performance.

The standard execution profiler on Linux is `gprof`. In order to profile a C program, the program needs to be compiled *and* linked with the `-pg` flag. When you compiled the `words` program with `make` earlier, you may have noticed that `-pg` appeared on each line with `gcc`.

Programs compiled with `-pg` include additional code that monitors the execution of the program and measures:

- overall execution time
- how many times each function is called
- which functions call which other functions
- how much time it takes to execute each function

To generate an execution profile, run commands like the following:

```
$ ./mkwords 100000 3 | ./words - 50033
....
```

which inserts them into a hash table with 50033 slots. Since `mkwords` may produce duplicates when generating large numbers of words, the actual number of distinct words added to the hash table may be less than the number of words requested (in this case, 90893 distinct items will be inserted). Remember that for each word in the input, there will be one insert operation (to add it to the hash table) and one search operation (to check that it was added).

Since `gprof` produces quite a lot of output, it is useful to pipe it through the `less` command, which allows us to scroll through output one screenful at a time.

The output from `gprof` has two parts:

- flat profile: how much time was spent in each funtion; how many times it was called
- graph profile: how many times was each function called from which other functions

For this lab, we consider only the flat profile, although you might want to check the graph profile to see if it gives you any information that might be useful to tune the program.

If you execute the `words` program as above, then the flat profile will look *approximately* like:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 44.50      0.04     0.04   200004   200.24   200.24  hash
 33.37      0.07     0.03   200004   150.18   150.18  ListSearch
 11.12      0.08     0.01   100004   100.12   450.53  HashTableSearch
 11.12      0.09     0.01   100000   100.12   450.53  HashTableInsert
  0.00      0.09     0.00   378051     0.00     0.00  cmp
  0.00      0.09     0.00   100001     0.00     0.00  ItemGet
  0.00      0.09     0.00   100000     0.00     0.00  newItem
  0.00      0.09     0.00    90936     0.00     0.00  ListInsert
  0.00      0.09     0.00    90936     0.00     0.00  dropItem
  0.00      0.09     0.00    50033     0.00     0.00  ListLength
  0.00      0.09     0.00    50033     0.00     0.00  dropList
  0.00      0.09     0.00    50033     0.00     0.00  newList
  0.00      0.09     0.00        1     0.00     0.00  HashTableStats
  0.00      0.09     0.00        1     0.00     0.00  dropHashTable
  0.00      0.09     0.00        1     0.00     0.00  newHashTable
```

code) where the counts come from. However, the timing (being sampling-based) may differ from machine to machine, and even differ between executions of the program on the same machine. This may result in significantly different times, different percentages, and even a different ordering of functions. Given such variations, is this kind of profile even useful? Yes, because the most time-consuming functions will be consistently higher than all of the others, and thus give you a basis for tuning.

Each line gives some statistics for one function in the program. The "% time" column indicates the percentage of execution time that was spent in the function. The (approximate) total running time of the program can be obtained by reading down the "cumulative seconds" column; the final value is the total time. The "self seconds" column gives the total time spent executing that function, during the entire course of program execution; the values in this column are added to obtain the "cumulative seconds" column. The "calls" column indicates the total number of times that the function was called during program execution. The "self ms/call" gives the average time spent *in each call* of the function, while the "total ms/call" gives the time spent in this function plus any of the functions it calls.

In the above profile, you can see that the program takes a total of 0.06 seconds to run (the last value in the "cumulative seconds" column). Function-wise, most of the time is spent in the `hash()` function. The next most expensive consumers of execution time are `ListSearch()` and `ListLength()`. (However, as noted above, you may observe different functions and quite different percentages.)

You might be surprised to see that most of the functions appear to cost 0.00ms to run. The most likely explanation here is that the cost of executing the function is, on average, less than 0.005 ms, which is rounded down to zero. Such a small cost may mean that the function itself is not inherently inefficient; if it features prominently in the cumulative time, you need to consider why it's being called so many times (which is where the graph profile helps).

In the above example, the `hash()` function is called many times, but can't be called less times (why not?), so if we want to tune the performance of this program, we would need to improve the speed of the `hash()` function (but without sacrificing its good performance in spreading hash values across the index range). If we make the hashing distribution worse while making the `hash()` function faster, we might simply end up moving the cost from the `hash()` function to some other function such as `ListSearch()`.

For testing the `words` program, you will want to use some reasonably large inputs. Try running your program as follows:

```
$ ./mkwords 1000000 | ./words - 424247
```

Note that the above command inserts 857424 distinct words into a hash table with 424247 slots. Clearly, since there are more words than slots, chains of length greater than one will occur frequently in this table. You could try adding more slots to the table to see how this improves the average/maximum chain length. You could also try inserting more words, to do some serious stress testing. We suggest keeping the ratio of words to slots at less than 2:1, and ideally closer to 1:1 (which is aiming at one slot per word).

Remember that the whole point of minimising chain lengths is that the worst case lookup cost is proportional to the maximum chain length, and the average case cost is proportional to the average chain length. You will quickly notice, while running with large inputs, that different slot numbers produce different average chain lengths, and those examples with shorter average chain lengths run much faster than those with longer average chain lengths.

If you want an alternative word set, there is a dictionary of 90,000 English words and a dictionary of 5 million place names (from the GeoNames database) which you could use on the CSE workstations. Try the following commands:

```
$ ./words /web/cs2521/23T2/labs/week17/data/dict 49999
$ ./words /web/cs2521/23T2/labs/week17/data/dict 50000
$ ./words /web/cs2521/23T2/labs/week17/data/places 1048576
$ ./words /web/cs2521/23T2/labs/week17/data/places 1048573
```

Don't copy these files to your own directory as they are quite large.

Consider the questions below, using the above command and variations on it (see below). Put your answers in a text file called `answers.txt`.

Your answers file should (ultimately) contain:

- answers to the questions below
- output from `words` to illustrate your answers (where appropriate)
- flat profiles to illustrate your answers (where appropriate)
- analyses/explanations for all answers

the output of the `words` program.

a. The `mkwords 1000000 3` command produces 857424 distinct words. What is the maximum chain length if a hash table size of 85711 is used? How does the chain length distribution change if the hash table size is 100000? 214283? 400000? 400837? 857144? 857137?

b. Every other number above (i.e. 85711, 214283, 400837, 857137) is prime. It is often noted that using prime numbers appropriately in the hash function leads to a better distribution of hash values, and thus generally shorter chains. Does this appear to be the case for the hash table sizes in the previous question?

c. An "optimal" hash table would have all slots occupied and have all chains of length roughly (nwords/nslots). In practice, this is impossible to achieve in the general case, and what we want is a table with relatively short chains, with as few slots as possible (small size of hash table), and not too many empty slots. Can you find a suitable hash table size that keeps the maximum chain length under 10, and has most chains with length 1 or 2, but also uses more than 2/3 of the slots?

d. Compare both the outputs and the profiles for the two commands:

```
$ ./words /web/cs2521/23T2/labs/week17/data/places 1048576
$ ./words /web/cs2521/23T2/labs/week17/data/places 1048573
```

What does this tell you about hash table search performance when the hash function is significantly sub-optimal?

e. Examine the profiles from running the command:

```
$ ./mkwords 1000000 | ./words - N
```

For a number of different values of *N*, what are the most costly functions (in terms of overall time)?

f. Suggest how the individual functions might be improved. Suggest how the overall performance might be improved.

g. Implement your suggestions and then give a new profile to show the improvement, and explain how the profile shows the improvement.

If you want some prime numbers to use for testing different table sizes, there are a large number of them in:

```
/web/cs2521/23T2/labs/week17/data/primes
```

the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs2521@cse.unsw.edu.au
CRICOS Provider 00098G