2020 Semester 1
COMP20007 Design of Algorithms
School of Computing and Information Systems
The University of Melbourne

Assignment 1

Due: 5:00 PM AEST Wednesday 29 April

Introduction

This assignment contains 3 problems. You will be required to write pseudocode and C code, as well as provide a detailed analysis of your algorithms. You will submit your solutions for the C programming component of this assignment via `dimefox submit` and the written component via the LMS.

This assignment has a total of 10 marks and will contribute 10% to your final grade for this subject.

Problem 1

2 Marks – 0.5 - 1 Page

In this task you need to design an algorithm to evaluate arithmetic expressions that consist of integer numbers, opening and closing parentheses and the operators `+`, `-`, `*` and `/`. Here `/` is integer division.

A well-defined arithmetic expression is defined as:

- A digit `d` $\in$ {`0, 1, 2, 3, 4, 5, 6, 7, 8, 9`} is a well-formed arithmetic expression.
- If `X` and `Y` are well-formed arithmetic expressions, then the four expressions `(X + Y)`, `(X - Y)`, `(X * Y)` and `(X / Y)` are also well-formed arithmetic expressions.

Examples of well-formed arithmetic expressions are `"((8+7)*3)"`, `"(4-(9-2))"` and `"0"`. However, the expressions `) "3+) 2 (())"`, `"(9 + ())"`, `"-4"`, `"5-8"`, `"108"` or `"(8)"` are not well-formed arithmetic expressions.

For this problem you will be required to write *pseudocode*, *i.e.*, human understandable "code" which provides enough detail to make it clear which operations your algorithm is performing but doesn't let the syntax get in the way of presenting your algorithm – the lecture slides and solutions to the workshops contain good examples of pseudocode.

You must write an algorithm, in pseudocode, which checks whether an input string is a well-formed arithmetic expression. If it is not well-formed then your algorithm must return NotWellFormed, otherwise it must evaluate the expression and return the integer result.

Your algorithm does not have to deal with division by 0.

Your algorithm should process the input from left to right, one character at a time. You should make use of two stacks, one stack for operators (including `(` and `)`) and another stack for values. Algorithms which do not make use of the stack abstract data structure will not be awarded full marks.

Problem 2

In this problem you will be implementing three functions in C. These functions deal with the abstract data type called a **deque**, pronounced *deck*. A deque is a double-ended queue in which elements can be inserted and removed from both ends (which we'll call the top and the bottom). In particular a deque must provide the following functions:

- PUSH(x) – inserts the element x at the top of the deque,
- INSERT(x) – inserts the element x at the bottom of the deque,
- POP() – removes and returns the element at the top of the deque, and
- REMOVE() – removes and returns the element at the bottom of the deque.

In the provided C code we have implemented the type `Deque` for you in the deque module (consisting of `deque.h` and `deque.c`) using a **doubly-linked list**.

Your task is to read and understand how the deque module works and implement the functions described in Parts (a), (b) and (c).

Your implementations must go in `deque.c`. You can add any additional functions required to solve this problem.

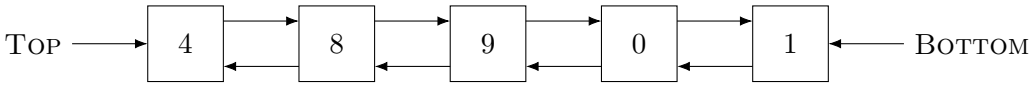
In each of these functions you should not return a new `Deque` but rather alter the original `Deque` data structure. You may make use of additional data structures during these functions, however they must be freed before your function returns.

Part (a)

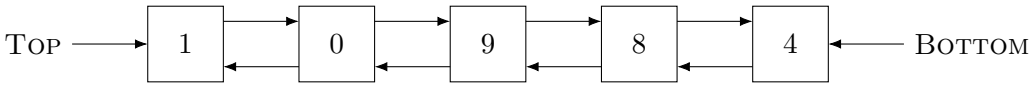
1 Mark

You must implement the function `iterative_reverse(Deque *deque)` which takes a pointer to a `Deque` and reverses this deque using an *iterative approach*.

For example if the deque initially looks like:



Then after calling `iterative_reverse()` the deque should look like:



For full marks your function must run in $O(n)$ time for a deque with n elements.

Part (b)

1 Mark

As in *Part (a)* you must write a function to reverse the deque, although this time you must use a recursive approach. The function you must implement is `recursive_reverse(Deque *deque)`.

Again, your function must run in $O(n)$ time for a deque with n elements.

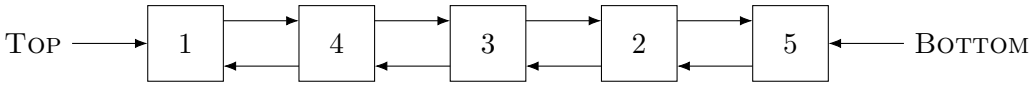
Part (c)

1 Mark

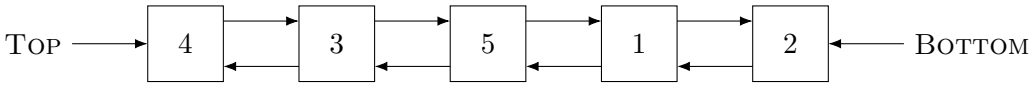
Write a function `split(Deque *deque, int k)` that rearranges the elements in the deque such that each element with a value greater than or equal to the critical value (*i.e.*, $\geq$ `k`) precedes (*i.e.*, is closer to the `top` of the deque) each element with a value less than `k`.

Within the two groups of elements (*i.e.*, $\geq$ `k` and $<$ `k`) the elements must be in their original order.

For example if we were to split the following deque with a critical value `k` = 3,



then the resulting deque would be,



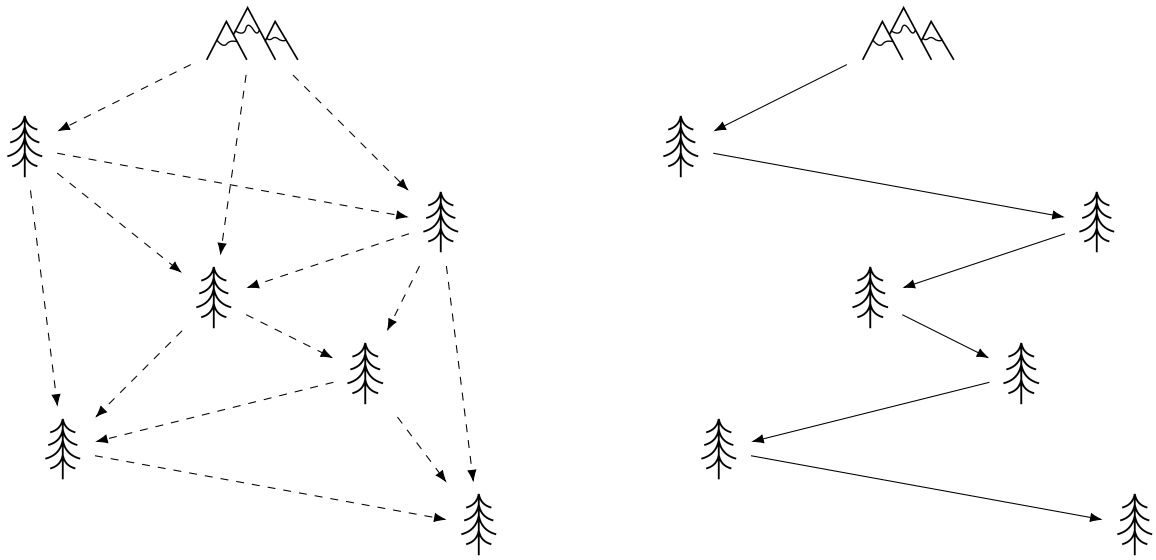
This function must run in $O(n)$ time, where n is the number of elements in the deque.

Problem 3

A park ranger has to trim all the trees on a mountain slope. Since it is winter, the slope is covered in snow and it is not safe to climb it directly. Therefore, the ranger plans to chairlift to the top of the mountain and ski down the slope, trimming the trees as they go down.

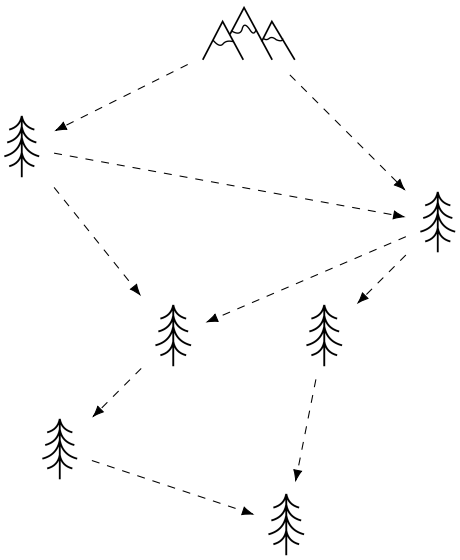
You are provided with a map of the trees on the ski slope with the possible routes between trees. You must write an algorithm to answer the following question: **can the park ranger trim all of the trees using the chairlift only once?** In other words, is there a single run the park ranger can perform from the top of the mountain which goes past every tree, or does the park ranger require multiple trips down the mountain?

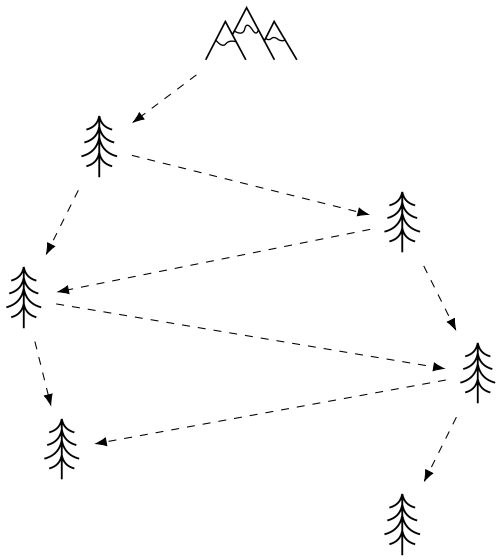
For example, the ski slope map on the left indicates which routes between trees are possible (with the mountain icon indicating the top of the mountain where the park ranger must begin his traversal of the mountain). The answer in this case is **yes, the park ranger can trim all trees in one run**. This run is shown on the right.



Note that your algorithm doesn't need to produce the order in which the trees should be visited, only determine whether or not there does exist a valid run from the top of the mountain that visits all the trees.

The following two maps, however, do not have a single run from the top to the bottom which visits all trees.





You will notice that these trees and the possible routes between them form a directed graph. Since the park ranger can only ski down the mountain (and cannot climb up it during a run) the graph will not have any cycles. Thus, the graph is a directed acyclic graph (DAG).

Part (a)

2 Marks

Devise a linear time algorithm which determines whether or not the park ranger can ski down and trim all of the trees in a single run. If there are n trees and m edges between trees then your algorithm must run in $O(n + m)$.

You must implement your algorithm in C by completing the function `is_single_run_possible()` in the park ranger module provided for you (`parkranger.c` and `parkranger.h`). You should add any types and functions required to solve this problem.

Your program must provide an appropriate data structure to store the directed graph.

Your program will be provided with the ski slope map via `stdin` in the following format:

- One line containing integers n and m. n indicates the number of trees, and m indicates the number of possible routes (*i.e.*, edges) between trees (or between the top of the mountain and a tree).
- m lines containing two integers u and v, which indicates that it's possible for the park ranger to ski from u directly to v.

  The trees are labelled 1 to n, while the top of the mountain is always labelled 0.

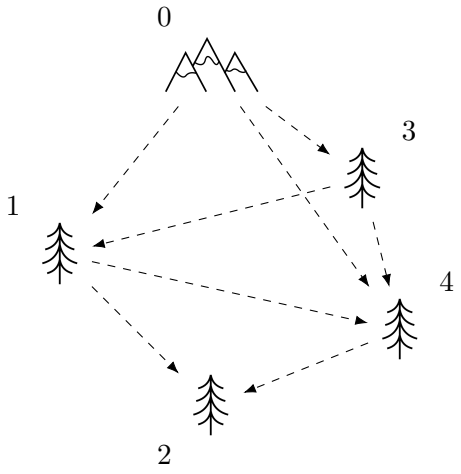
Your function must perform the following steps:

- Read in the data from `stdin`.
- Store the data in an appropriate data structure.
- Determine whether the park ranger can trim all the trees in a single run.
- Do any clean up required (*e.g.*, free any allocated memory).

- Return `true` or `false` depending on the result of your algorithm. Note that `true` and `false` are of type `bool`, which is included from the `stdbool.h` library.

The following ski slope is an example of the input format your program will be given.

```
4 8
0 1
0 3
0 4
1 2
1 4
3 1
3 4
4 2
```



Part (b)

2 Marks – 0.5 - 1 Page

Explain your algorithm, and explain why your algorithm runs in linear time.

Clear pseudocode may help you convey your algorithm to the reader, but is not required for full marks for this part. You should not include very detailed pseudocode if you do decide to present this, just an overview of your algorithm.

Completing the Programming Problems

We have provided you with the skeleton code for this assignment. The provided files has the following directory structure:

```
provided_files/
  Makefile
  main.c
  deque.c
  deque.h
  parkranger.c
  parkranger.h
  util.c
  util.h
  tests/
    p2a-in-1.txt
    p2a-out-1.txt
    ...
    p3a-in-6.txt
    p3a-out-6.txt
```

You must not change the `main.c` file, however you are allowed to change any of the other files (without changing the existing type declarations and function signatures).

If you create new C modules (which you are encouraged to do) then you must change the `Makefile` so that your program can be compiled on `dimefox` using the `make` command.

To run the program you must first compile with `make`, and then run the `a1` command with one of these command line arguments: `p2a`, `p2b`, `p2c`, `p3a`.

For example:

```
$ make
$ ./a1 p2a < tests/input.txt
$ ./a1 p2b < tests/input.txt
$ ./a1 p2c < tests/input.txt
$ ./a1 p3a < tests/input.txt
```

The `pXX-in-X.txt` files contain the input your program will be provided for each test case, and the `pXX-out-X.txt` files contain the expected output. Your program must match the expected output exactly.

Your functions **must not print to** `stdout`. Your program should change the `Deque` provided and return nothing in the case of Problem 2 and return `true` or `false` in the case of Problem 3.

Testing your Program

Problems 2 (a) and 2 (b) can be tested by using `./a1 p2a` and `./a1 p2b` respectively. The program expects input via `stdin` which is a list of integers (one per line). These integers will be inserted into the deque (inserting each element at the bottom), then your function will be called.

For example if `tests/input.txt` contains the numbers 10, 15, 20, 30 on their own lines then after implementing `iterative_reverse()` your program should behave like so:

```
$ ./a1 p2a < tests/input.txt
read 4 elements into the deque
original deque: [10, 15, 20, 30]
reversed deque: [30, 20, 15, 10]
successfully freed the deque
```

Again, note that your program should not print this output, this is handled by `test_reverse()` in `main.c` which has been written for you.

Testing Problem 2 (c) is similar, you should use the `p2c` option for `./a1`. Here the text input includes a list of integers on their own lines as well, with the first integer being the critical value, and the remaining integers being inserted into the deque before it is split.

Testing Problem 3 (a) requires the input to be provided to `stdin` as described in the problem description. If the example input from the problem description is in `tests/input.txt` then your program should behave like so:

```
$ ./a1 p3a < tests/input.txt
the trees on the ski slope CAN be trimmed in one run
```

Again, your function should just return `true` or `false` and should not write to `stdout`. The output is handled by `main.c`.

Programming Problem Submission

You will submit your program via `dimefox submit`. Instructions for how to connect to the `dimefox` server can be found on the LMS.

You should copy all files required to compile and run your code to `dimefox`, this includes the `Makefile` and all `.c` and `.h` files.

It's recommended that you test your program on `dimefox` before submitting to make sure that your program compiles without warnings and runs as expected.

From the directory containing these files you should run `submit` and **list all files required to compile and run your program**. For example, to submit only the provided files we would run:

```
$ submit comp20007 a1 Makefile main.c deque.c deque.h
   ... util.c util.h parkranger.c parkranger.h
```

Note that you can also list all `.c` and `.h` files in the current directory with `*.c` and `*.h`, so the following command will be equivalent:

```
$ submit comp20007 a1 Makefile *.c *.h
```

For this to work correctly you should have your Assignment 1 code in its own subdirectory (*i.e.*, the only files in the directory should be the files you wish to submit).

You must then `verify` your submission, which will provide you with the outcome of each of the tests, and the number of marks your submission will receive.

```
$ verify comp20007 a1 > a1-receipt.txt
```

To view the result of the `verify` command you must run:

```
$ less a1-receipt.txt
```

`less` is a tool which allows you to read files using the command line. You can press `Q` on your keyboard to exit the `less` view.

You can submit as many times as you would like.

Note that programs which do not implement the algorithm they claim to (*e.g.*, a solution for `iterative_reverse()` which uses recursion) or does not run the required time bound will receive fewer marks than the receipt may suggest.

Any attempt to manipulate the submission system and/or hard-code solutions to pass the specific test cases we have provided will **result in a mark of 0 for the whole assignment.**

Completing the Written Problems

You will submit your solutions to Problems 1 and 3 (c) via the LMS.

For Problem 1, which asks for pseudocode, we expect you to provide the same level of detail as the lectures and workshops do. Pseudocode which lacks sufficient detail or is too detailed (*e.g.*, looks like C code) will be subject to a mark deduction.

Your submission should be **typed and not handwritten** and submitted as a **.pdf** file. Pseudocode should be formatted similarly to lectures/workshops, or presented in a monospace font.

The page limit for each problem is given (half to one full page per written problem).

Make sure you confirm that your written submission has been submitted correctly.

Mark Allocation

The total number of marks in this assignment is 10. The maximum marks for each problem are:

Problem 1 2 marks

Problem 2 3 marks (1 per part)

Problem 3 4 marks (2 per part)

There is one additional mark awarded for "structure and style" for the C programming component. Of particular importance will be the structure of your C program in terms of modules (`.c` and `.h` files), and how you separate your types and functions across these files.

In total there are 4 marks for the written problems (Problem 1 and Problem 3 (b)) and 6 marks for the C programming problems (Problem 2 and Problem 3 (a)).

We have provided **3 test cases** for each part of Problem 2 and **6 test cases** for Problem 3 (a).

The marks awarded for each part of Problem 2 will be calculated by:

$$\text{Marks} = \max\left\{1 - 0.5 \times \text{Test Cases Failed}, 0\right\}.$$

So passing 0 or 1 of the tests results in a mark of 0, passing 2 results in a mark of 0.5 and passing all three results in 1 mark (the maximum available).

For Problem 3 the marks are calculated by:

$$\text{Marks} = \max\left\{2 - 0.5 \times \text{Test Cases Failed}, 0\right\}.$$

So your submission will get 0.5 marks for passing 3 tests, 1 mark for passing 4 tests, 1.5 for 5 and a maximum of 2 marks for passing all 6.

Late Policy

A late penalty of 20% per day will be applied to submissions made after the deadline. The 20% applies to the *number of total marks*. This applies *per component*, *i.e.*,

$$\begin{aligned}\text{Grade} = &\max\Big\{\text{Programming Grade} - 0.2 \times \text{Days Late} \times 6, 0\Big\} \\ &+ \max\Big\{\text{Written Submission Grade} - 0.2 \times \text{Days Late} \times 4, 0\Big\}.\end{aligned}$$

For example, if you are 2 days late with the programming component but only 1 day late with the analysis component your grade for the programming component will be reduced by $0.4 \times 6 = 2.4$ and the grade for the analysis component will be reduced by $0.2 \times 4 = 0.8$.

Academic Honesty

You may make use of code provided as part of this subject's workshops or their solutions (with proper attribution), however you **may not** use code sourced from the Internet or elsewhere. Using code from the Internet is grounds for academic misconduct.

All work is to be done on an individual basis. All submissions will be subject to automated similarity detection. Where academic misconduct is detected, all parties involved will be referred to the School of Engineering for handling under the University Discipline procedures. Please see the Subject Guide and the "Academic Integrity" section of the LMS for more information.