CMPT 125: Practice Midterm Answer Key

Linked Lists

Suppose you have a singly-linked list whose nodes are defined like this:

```
struct Node {
    int val;
    Node* next;
};
```

There's a global variable `H` of type `Node*` that should always point to the first node of the list. If the list is empty, then `H == nullptr`. The `next` pointer of the last node on the list has the value `nullptr`.

The following questions ask you to write functions that process this singly-linked list. After each function is finished, `H` should either be `nullptr` (if the list is empty), or pointing to the first node of the list. You can write helper functions if necessary, but **don't** use any other data structures, such as arrays or vectors, in your answer.

a) Write two versions of a function called `pop_tail()` that deletes the last node on the list and returns its value. If the list is empty, use `cmpt::error` to cause an error. The first version of `pop_tail()` should use a loop, and the second version should use recursion.

```
int pop_tail_loop() {
    if (H == nullptr) cmpt::error("error!");
    if (H->next == nullptr) {  // list has 1 node
        int result = H->val;
        delete H;
        H = nullptr;
        return result;
    } else { // list has 2 or more nodes
        Node* p = H;
        while (p->next->next != nullptr)
            p = p->next;
        // p->next->next == nullptr
        int result = p->next->val;
        delete p->next;
        p->next = null;
        return result;
    }
}
```

```
////////////////////////////////////////////
// Pre-condition: both p and p->next are
// not nullptr
int pop_tail_rec(Node* p) {
    if (p->next->next == nullptr) {
        int result = p->next->val;
        delete p->next;
        p->next = null;
        return result;
    } else
        return pop_tail_rec(p->next);
}
```

```
int pop_tail_rec() {
    if (H == nullptr) cmpt::error("error!");
    if (H->next == nullptr) { // list has 1 node
    int result = H->val;
        delete H;
        H = nullptr;
        return result;
    } else { // list has 2 or more nodes
        return pop_tail_rec(H);
    }
}
```

b) Write a function called `get(n)` that returns the value of the nth node. The index of the first node is 0, so `get(0)` returns the value of the first node, `get(1)` returns the value of the second node, and so on.

If `n` < 0, or `n` >= (the number of nodes on the list), then cause an error with `cmpt::error`.

```
int get_loop(int i) {
    if (i < 0) cmpt::error("error!");
    Node* p = H;
    while (!(p == nullptr || i == 0)) {
        p = p->next;
        i--;
    }

    // p == nullptr || i == 0
    if (p == nullptr) cmpt::error("error!");
    return p->val;
}
```

```
////////////////////////////////////////////

int get_rec(int i, Node* p) {
    if (i < 0 || p == nullptr)
        cmpt::error("error");
    if (i == 0) {
        return p->val;
    } else {
        return get_rec(i - 1, p->next);
    }
}

int get_rec(int i) {
    return get_rec(i, H);
}
```

O-notation

a) Give the mathematical definition of "f(n) is O(g(n))".

If f(n) is O(g(n)), then there exists $c > 0$ and $n_0 > 1$ such that

$f(n) <= cg(n), \quad n >= n_0$

b) Using the definition of O-notation, prove that n^2 is $O(2^n)$.

To show n^2 is $O(2^n)$, we must find $c > 0$ and $n_0 > 1$ such that

$n^2 <= c2^n, \quad$ for $n >= n_0$

By creating a table of values you can see that, eventually, 2^n always greater than n^2:

n	2^n	n^2
1	2	1
2	4	4
3	8	9
4	16	16
5	32	25
6	64	36
7	128	49

So, if we set $c = 1$ and $n_0 = 4$, then inequality (1) above is satisfied, and so n^2 is $O(2^n)$.

c) Prove or disprove: 500 is O(1).

This is true: 500 is O(1). To prove this, you must find $c > 0$ and $n_0 >= 1$ such that

$500 <= c*1$, for $n >= n_0$

Setting $c = 500$ and $n_0 = 1$ satisfies the inequality, thus proving 500 is O(1).

d) Suppose algorithm A does $O(n^2)$ primitive operations when run on an input of size n. Experiments show that for n = 1000, it takes about 2 seconds to run. About how many seconds would you expect to wait for A to process an input of size n = 10,000?

10,000 is 100 times bigger than 1000 and, since A is $O(n^2)$, if n is replaced by 10n, then it does about $(10n)^2=100n^2$ primitive operations, i.e. 100 times more primitive operations. So you would expect A to take about **100*2 = 200 seconds** on an input of size n = 10,000.

e) Suppose f(n) is O(g(n)). Is it also possible that g(n) is O(f(n))? If so, give examples of functions for f and g that make it true. If not, explain why it's impossible.

Answer yes: e.g. f(n) = 2n, g(n) = n

Stacks and ADTs

a) Define **abstract data type** (**ADT**).

An ADT is a mathematical model of data structure that specifies the type of the data stored and the operations on that type. The exact implementation of the type, and the exact details of the algorithms, are not usually given. The ADT only specifies how the operations and the type behaves.

b) Write an abstract data type `Queue` for a queue using C++. Include at least 5 basic functions, and use precise English to write the specifications.

```
// returns a new, empty queue
Queue make_empty()

// adds x as the new back of queue Q
void enqueue(Queue Q, int x)

// removes and returns the front element of the queue
// Q; if Q has no elements, then causes an error
int dequeue(Queue Q)

// returns true if, and only if, Q has no items
bool is_empty(Queue Q)

// returns the number of elements in Q
int size(Queue Q)
```

c) Show how you can simulate a stack using only one queue. You only need to show how to implement push and pop in no worse then O(n) time. Don't use any other arrays, lists, stacks, etc. in your answer.

Let Q be an initially empty queue as defined in the previous question. Then push and pop work like this:

push(x):
enqueue(x), then do the following n-1 times (where n is the # of elements in Q): enqueue(dequeue()).

pop(): dequeue()