CSCI 104L: Lecture 5

Encapsulation and Classes

We will group together all data and functions that interact with that data into a common element, or *class*. This idea is called **encapsulation**. Here is an example class signature for a linked list of integers:

```
class IntLinkedList {
        public:
                IntLinkedList();
                IntLinkedList(int n);
                ~IntLinkedList();
                void prepend(int n);
                void remove(int toRemove);
                void printList();
                void printReverse ();
        private:
                void printReverseHelper (Item *p);
                Item *head;
};
```

Here is how we will implement `printReverse()`.

```
void IntLinkedList::printReverse() {
  if (head != NULL) printReverseHelper(head);
}
void IntLinkedList::printReverseHelper(Item *p) {
  if (p->next != NULL) printReverseHelper(p->next);
  cout << p->value;
}
```

Here is a possible usage of our IntLinkedList:

```
int main () {
        IntLinkedList *myList = new IntLinkedList;
        myList->printList();
        delete myList;
        return 0;
}
```

A destructor is necessary to prevent memory leaks:

```
IntLinkedList::~IntLinkedList () {
        Item *p = head, *q;
        while (p != NULL) {
                q = p->next;
                delete p;
                p = q;
        }
}
```

You can have multiple constructors. You could allow a user to start a Linked List with a single node with value n. Both constructors can be used, one constructor does not replace the other.

Runtime Analysis

- $f(n)$ is $O(g(n))$ means $f(n) \leq c \cdot g(n)$, for all $n \geq n_0$, for some constants c, n_0. That is, our algorithm never takes more time than $g(n)$ times a constant for sufficiently large inputs.
- $f(n)$ is $\Theta(g(n))$ means $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$. This means that, up to constant factors, f and g are the same function.
- Big-Oh notation measures the growth rate of an algorithm, ignoring those pesky constant factors, and concerning itself with what happens for very large inputs. We typically consider the worst-case.

Calculating worst-case

- Do not assume a specific input. If some inputs do one thing, and other inputs do the other, assume the worst of the two happens.
- An elementary statement such as `a[i]++`, is a constant number of steps, so write it as $\Theta(1)$.
- If you have two code blocks with runtime $T_1(n)$ and $T_2(n)$, and you run them in sequence, then add the runtimes. The runtime would be $T_1(n) + T_2(n)$.
- When you have a for loop, add up each element. So if a block takes $T_i(n)$ time, for i ranging from $i = 0$ up to $i = n - 1$, then the runtime would be $\sum_{i=0}^{n-1} T_i(n)$. Note the variable i is saying that the block may take a different amount of time for each iteration.

Exercise 1. Calculate the runtime:

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    a[i][j] = i*j;
```

The following sums may come up in analysis and may prove useful to you.

- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \theta(n^2)$. This is called the arithmetic series.
- $\sum_{i=0}^{n} \theta(i^p) = \Theta(n^{p+1})$. This is a general form of the arithmetic series.
- $\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1} = \Theta(c^n)$. This is called the geometic series.
- $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$. This is called the harmonic series.

Exercise 2. Calculate the runtime:

```
for (int i = 0; i < n; i++)
  if (a[i][0] == 0)
    for (int j = 0; j < i; j++)
      a[i][j] = i*j;
```

Exercise 3. Calculate the runtime:

```
for (int i = 1; i < n; i *= 2)
  for (int j = 0; j < i; j++)
    a[i][j] = i*j;
```