CSCI 104L: Lecture 5

Abstract Data Types

- If we are precise about what we want to do (the operations we want to implement), then we have specified an **Abstract Data Type** or ADT.
- A **List** is defined by the following operations, where T denotes any one type (such as int, string, etc).
  1. void insert (int position, T value): inserts value at the specified position, moving all later elements one position to the right.
  2. void remove(int position): removes the value at the specified position, moving all later elements one position to the left.
  3. void set(int position, T value): overwrites the specified position with the given value.
  4. T get (int position): returns the value at the specified position.
- A **Set** (called a Bag in the textbook) supports the following:
  1. void add (T item): adds item to the set.
  2. void remove (T item): removes item from the set.
  3. bool contains (T item): determines whether the set contains item.
- A **Map** (sometimes referred to as a Dictionary) associates values with keys. keyType can be any individual data type, as can valueType.
  1. void add (keyType key, valueType value): adds a mapping from key to value.
  2. void remove (keyType key): removes the mapping for key.
  3. valueType get (keyType key): returns the value that key maps to.
- All of the ADTs support storing and accessing data. It would be kind of pointless to make an ADT which did not support this.
- A List cares about order, whereas the others do not.

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.