CSCI 104L Lecture 3: Linked Lists

- Advantage: they are easy to grow and shrink.
- Disadvantage: you can’t search a sorted list efficiently (binary search doesn’t work when you don’t know where the middle element resides in memory).

```c
struct Item {
    int value;
    Item *next;
    Item (int val, Item *n) : value(val), next(n) { }
};
```

There are two different ways to traverse a linked list:

```c
void traverse (Item *head) {
    for (Item *p = head; p != NULL; p = p->next) {
        // do whatever you’re going to do here.
    }
}
```

```c
void traverse (Item *head) {
    // do whatever you’re going to do here.
    traverse (head->next);
}
```

Doubly-Linked Lists:

```c
struct Item {
    int value;
    Item *next;
    Item *prev;
    Item (int val, Item *n, Item *p) { ... }
};
```

Adding to the front of the list:

```c
void prepend (Item *head, int n) {
    Item *newElement = new Item (n, head, NULL);
    head = newElement;
    if (head->next != NULL) head->next->prev = head;
}
```

Adding to the end of the list (without a tail pointer):

```c
void append(Item *head, int n) {
    if (head == NULL) head = new Item (n, NULL, NULL);
    else if (head->next == NULL) head->next = new Item (n, NULL, head);
    else append (head->next, n);
}
```
Removing, when given a pointer to the item to be removed:

```c
void remove(Item *&head, Item *toRemove) {
    if (toRemove != head) toRemove->prev->next = toRemove->next;
    else head = toRemove->next;
    if (toRemove->next != NULL) toRemove->next->prev = toRemove->prev;
    delete toRemove;
}
```

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. insert (int position, T value): inserts value at the specified position, moving all later elements one position to the right.
  2. remove(int position): removes the value at the specified position, moving all later elements one position to the left.
  3. 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. add (T item): adds item to the set.
  2. 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. add (keyType key, valueType value): adds a mapping from key to value.
  2. 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.