CSCI 104L Lecture 4: 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).

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

This allows you to create a new item as such:

```
Item *it = new Item(5, NULL);
```

There are two different ways to traverse a linked list:

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

Doubly-Linked Lists:

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

Adding to the front of the list:

```
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 back of the list (if no tail pointer):

```
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:

```
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;
}
```

Figure 1: XKCD # 379: Of course, the assert doesn't work.