CSCI 104L Lecture 3: Recursion and Linked Lists

```
//t = target element.  b = array.
//lo = index of first element in array (pass in 0 when you call this function).
//hi = index of last element in array (initially array length-1).
int binarySearch(int t, int *b, int lo, int hi) {
    if (hi < lo) return -1; //nothing to search, it's not in the array.
    else {
        int mid = (hi+lo)/2;  //the middle of the array, rounded down.
        if (t == b[mid]) return mid;  //found it!
        else if (t < b[mid]) return binarySearch(t, b, lo, mid-1); //search left.
        else return binarySearch(t, b, mid+1, hi); //search right.
    }
}

//t = target element.  b = array.  len = length of array.
int iterativeBinarySearch(int t, int *b, int len) {
    int lo = 0, hi = len-1, mid;
    while(lo <= hi) {
        mid = (hi+lo)/2;
        if (b[mid]==t) return mid;
        else if (t < b[mid]) hi = mid-1;
        else lo = mid+1;
    }
    return -1;
}
```

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) { }
};
```

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;
}
```

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.