"Cheat Sheet" - Week 6

CS50 — Fall 2012

Prepared by: Doug Lloyd '09

October 22, 2012

Linked Lists

Linked lists come in singly- and doubly-linked flavors. This sheet focuses only on singly-linked lists, but the concepts herein can easily be generalized to doubly-linked lists. Linked lists give us a new data structure for holding, arranging, searching for, sorting, deleting, and inserting values. Previously, all we had was the array! Linked lists are a very special structure, where the fields of the structure are some data and a pointer to another structure of the same type. In code form:

```
typedef struct _sllist {
   int data;
   struct _sllist *next;
} sllist;
```

Let's create a singly linked list using a few of these nodes!

```
int main() {
   sllist first;
   sllist second;
   sllist *third = malloc(sizeof(sllist));
   first.data = 5;
   second.data = 6;
   third->data = 8;
   first.next = &second;
   second.next = third;
   third->next = NULL;
   ...
   free(third);
   return 0;
}
```

Now let's write a function to traverse the list and print out all the values! This function will take a pointer to the first node in the list (usually called the *head* of the list), and will continue printing until it hits a NULL value (which we typically use to represent the *tail* of the list). Let's do it both iteratively and recursively!

```
void printList_I(sllist *head) {
   while(head != NULL) {
      printf("%d ", head->data);
      head = head->next;
   }
   return;
}
```

```
void printList_R(sllist *head) {
   if(head == NULL)
      return;
   else {
      printf("%d ", head->data);
      printList_R(head->next);
   }
   return;
}
```

And above, we'd just replace the `...` line in the previously-defined `main()` with `printList_I(&first)` or `printList_R(&first)`, as appropriate. The things you need to know how to do with a linked list are: (1) create them, (2) insert an item after another item already in the list, (3) insert an item before another item already in the list (doubly-linked lists only), (4) add an item to the end of a list, (5) delete a single node from the list, and (6) delete the entire linked list. Try playing around with them a bit to see if you can get these things to work! (Hint: The last one is best done recursively with a singly-linked list. Think about it!)

Hash Tables

Hash tables are a great data structure to store information in a semi-organized way. A hash table is made up of at least two parts: (1) an array and (2) a hash function. More advanced hash tables, which we will be using, also make use of (3) linked lists. Say we want to hash some strings. Let's create a hash table of linked lists nodes, where each of those nodes contains a string and a pointer to another node.

```
typedef struct _node {
   char word[50]; // max length of a word is 50
   struct _node *next;
} node;

node *hashtable[3]; // our hashtable has 3 possible hash values
```

We can then hash a string to compute a hash code, which is entirely derived from the string itself. For example, `hash("hello")` may be 298. We then mod that to be in the range [0, 2] (for our hashtable) and insert it.

```
[0]
[1] -> "hello"
[2]
```

Then, if we `hash("world")` and we get 325, we can mod that to be in the range [0, 2] and insert it, rearranging the nodes of the linked list as necessary so we don't lose what was there before.

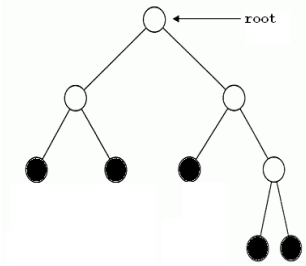
```
[0]
[1] -> "world" -> "hello"
[2]
```

We can look up strings easily by hashing them again, and then searching through the list pointed to by that hash code to find whether or not they are in the list.

Binary Trees and Tries

Notice how with hash tables we are combining simple data structures (arrays and linked lists) in a way so as to create a new data structures that has an even higher degree of utility. A trie, discussed in just a moment, uses those same two simple structures in a different way to do something completely different.
A tree is a data structure very similar to a linked list, but with a different interpretation. The rules of a tree (into which category binary trees and tries both fall) are as follows:

- A tree node can point at its children (named left and right, in a binary tree), or at NULL.
- A tree node may not point at any other node other than those listed above, including itself.



We typically visualize binary trees upside-down from how a regular tree looks. The root of the tree is the node at the very top. The leaves are the nodes where both children point to NULL (indicated here as colored-in circles). The branches are the nodes that are not leaves or the root. And what is a trie, then? A trie is simply a tree with an arbitrary number of children. This means that a binary tree is simply a special case of a trie, where the number of children is 2. Let's declare three structures. One for a binary tree, one for a binary tree with trie syntax, and one for a trie with 6 children.

```
// Binary tree
typedef struct _btree {
   bool valid;
   struct _btree *left;
   struct _btree *right;
} btree;
```

```
// Binary trie
typedef struct _btrie {
   bool valid;
   struct _btrie *children[2];
} btrie;
```

```
// 6-child trie
typedef struct _trie {
   bool valid;
   struct _trie *children[6];
} trie;
```