“Cheat Sheet” - Week 7

CS50 — Fall 2010

Prepared by: Doug Lloyd ’09

October 25, 2010

Valgrind

Valgrind is an incredibly useful tool for finding *memory leaks*. Run it from the command line in the following way:

```
valgrind -v --leak-check=full <executable>
```

When you do that, you’ll get a messy bunch of stuff that prints out. What you want (or, really, don’t want!) to see is something like the following:

```
==nnnnn== LEAK SUMMARY:
==nnnnn==    definitely lost: 8 bytes in 1 blocks
==nnnnn==    indirectly lost: 72 bytes in 9 blocks
==nnnnn==      possibly lost: 0 bytes in 0 blocks
==nnnnn==    still reachable: 0 bytes in 0 blocks
==nnnnn==         suppressed: 0 bytes in 0 blocks
```

What happened here? In this case, I `free()`d the head of a linked list containing eleven nodes, without first traversing through that linked list to free from the end to the beginning. The “definitely lost” refers to the node that the head pointed to. The “indirectly lost” refers to the chain from that point forward. Be sure to `man valgrind` for an explanation of other commands (like `--leak-check=full`) that you can use to trace where your memory leak comes from!

Bitwise Operators

There are four bitwise operators in C that you can use to compare integers at the *bit* level. They are (`&`) (bitwise AND), (`|`) (bitwise OR), (`~`) (bitwise NOT), and (`^`) (bitwise XOR). As an example, here are three equivalent ways to test if a number (`num`) is odd (note that, since they are stored in binary, a number is odd if it’s lowest bit is a 1, instead of a 0):

```
// Modulo
if(num % 2 == 1)
   printf("Odd\n");
```

```
// Fancier modulo
if(num % 2)
   printf("Odd\n");
```

```
// Bitwise
if(num & 1)
   printf("Odd\n");
```

You can also bit-shift, which shift bits left (`<<`) or right (`>>`). With integers, this has the effect of multiplying or dividing by 2^i, where `i` is the number on the right side of the operand. (e.g. if `x = 400`, then `x << 2` is 1600; `x >> 4` is 25.) The following table summarizes the outcomes of the other operators. Can you figure out, then, what they all mean?

`a`	`b`	`a & b`	`a | b`	`~a`	`a ^ b`
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	0	0

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]
```

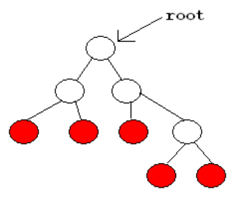
We can look up strings easily by hashing them again, and then searching through the list pointed to by that hash code to find whether or not they are in the list.

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.



Here is an example of a binary tree. We typically visualize them 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;
```