Topics Covered

Structs

A structure is basically a super-variable. We use structures to logically group together a collection of elements (which we will call fields) that together comprise a larger element. For example, heres the potential definition of a structure (struct) for a student:

```
struct student_t {
  string name;
  int year;
  float gpa;
};
```

Once weve defined the structure, we can then use it like any other type. We access fields of the structure with the dot operator (.). We can also have pointers to the structure like we can have pointers to any other type. If we have a pointer to a structure, we access its fields with the arrow operator ($->$).

```
struct student_t john;
struct student_t * mary = malloc(sizeof(struct student_t));
john.name = "John";
mary->name = "Mary";
john.year = 2014;
mary->year = 2015;
john.gpa = 3.48;
mary->gpa = 3.55;
```

Linked Lists

How do you create a linked list?
Insert into a linked list (at head, tail, middle)?
Delete out of a linked list?
Delete an entire linked list?
And iterate over a linked list?

```
typedef struct node
{
    // just some form of data; could be a char* or whatever
    int i;

    // pointer to next node; have to include ‘struct‘ since this is a recursive definition
```

Linked Lists continued on next page...

```
    struct node *next;

    node;
}
```

How do you alter and iterate over linked lists? The end of a linked list will always be marked by a pointer to NULL. To iterate over a list, then, use a $node * head_ptr$ and simply loop through the list until its next field points to NULL. Similarly, to insert or delete a node, simply change where the $node * next$ points. Be careful about the order of operations, though! Especially when inserting, if done in the wrong order you could lose track of your list!

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. Lets 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 dont 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:

Binary Trees and Tries continued on next page. . .

1. A tree node can point at its children (named left and right, in a binary tree), or at NULL.
2. 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. Lets 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_t;

// Binary trie
typedef struct _btrie
{
  bool valid;
  struct _btrie *kids[2];
}
btrie_t;

// 6-child trie
typedef struct _trie6
{
  bool valid;
  struct _trie6 *kids[6];
}
trie6_t;
```

Huffman Encoding

Take notes below from examples in class from study.cs50.net!