Data Structures and Algorithms — CSci 230
Chapter 5 — Hashing

Class Outline

Presumably, you have already studied hashing in CS II, so our coverage of hashing will be brief. The quiz on hashing will be merged into the quiz on trees, creating a double weighted quiz.

- In hashing, a **key** is mapped to a location in a table, and the key and an associated entry are then stored at that location.
- A primary application is in building **symbol tables** for compilers.
- The crucial issues are the mapping or **hash function** and how to handle **collisions**, e.g. when multiple keys map to the same table location.

Hash Functions

- An appropriate hash function should be efficiently computable and spread the keys (especially similar keys) as uniformly as possible throughout the table. The ideal hash function depends therefore on the keys themselves.
- Fortunately, in the majority of hashing applications strings are used as keys, and decent general purposing hashing functions may be developed for strings. Here's one

```
unsigned int
Hash(const std::string& key, int tbl_size)
{
  unsigned int value = 0;
  for ( int i=0; i<key.length(); i++ )
    value = ( value << 5 ) + key[i];
  return value % tbl_size;
}
```

Handling Collisions

- Handling collisions is usually done via *separate chaining*, where each table entry points to a (distinct) linked list.
- Earlier, when conserving memory was more of an issue, *closed hashing* techniques were heavily studied. In closed hashing collisions are resolved through a search for empty locations in the table.
- The attached code gives one implementation of *separate chaining*. It differs from the implementation given in the textbook in several ways. Most importantly, it provides a simple mechanism for dynamically changing the size of the table!

In-class Exercises

1. Consider computational efficiency issues:

   (a) What are the average case and worst-case times for inserting, finding and removing an entry from a hash table containing n entries?

   (b) Are these worst-case times likely to occur?

   (c) How do these compare to those of balanced trees?

2. What operations can be done on trees that can not be done on hash tables?

3. How would you modify the public interface and the implementation of the `HashTable` class to allow multiple instances of the same key?

Review Problems

1. Consider the following hashing function, similar to hash functions discussed in class.

```
unsigned int
Hash( const string & key, const int h_size )
{
  unsigned int value = 0;
  for ( int i=0; i<key.length(); i++ )
    value = (value + key[i]) << 3;  // multiply by 8
  return value % h_size;
}
```

   Is this a good hash function when `h_size = 128`? Why or why not?

2. Suppose you need a data structure to support several types of operations on a set of strings: insert a string, find a string, delete a string, and print the strings in order. Consider the possibilities of using either an unbalanced binary search tree, an AVL tree, or a hash table with separate chaining to represent the strings. Even under these assumptions, which data structure is best will depend on the data and on the relative frequencies of insert, find, delete and print operations.

   (a) Under what conditions should you choose a hash table? Why?

   (b) Under what conditions should you choose a binary search tree? Why?

   (c) Under what conditions should you choose an AVL tree? Why?