

# CS 14: Data Structures and Algorithms

## Lab 10: Hash Tables

Peter H. Fröhlich  
phf@cs.ucr.edu

Wagner Truppel  
wagner@cs.ucr.edu

Out on: March 10, 2003  
Due by: March 14, 2003

**Instructions:** Remember to follow the instructions and policies for assignments given on the course website. This week’s assignment consists of **two** separate parts:

- an **in lab** programming exercise (1)
- an **at home** programming exercise (2)

If available, your TA **may** provide skeleton code or test drivers, but otherwise you are on your own. The “in lab” part is due at the **end** of your **lab section**. The “at home” parts are due **before 8:00 pm** on the date given above. You have to use the department’s electronic turnin service for **everything**.

### 1 In-Lab: Basic Hash Tables

Your task for this week is to implement the **hash table** data structure discussed in the lecture. Your implementation should be in the form of a **template** class `HashTable` in a file `hashbasic`. You **can** reuse both `SimpleArray` from Lab 3 and `DoubleList` from Lab 5 for this part, but you can also start from scratch (if you are sure you have enough time).

Since a hash table maintains a **set** of elements, it should adhere to ADT `Set` and provide the following public operations:

```
void add( const T &t );
```

```
void rem( const T &t );
bool has( const T &t ) const;
bool empty() const;
```

Clients should be able to supply an **estimate** of the expected size to the hash table, and you should in turn choose a **prime number** close to that size for the length of the underlying `Array`. For clients that need to process the elements of a hash table one-by-one, you must also implement an iterator, which has to support the following operations:

```
bool valid();
T get() const;
void next();
```

The order in which elements are returned in a sequence of `next(); get();` operations is undefined, so you can choose the order most convenient for your implementation. However, you must return **each** element of the table **exactly** once. Finally, your implementation should constrain the type parameter `T` as little as possible.

### 2 At-Home: Fancy Hash Tables

You now will extend `hashbasic` to `hashfancy` and implement hash tables in which the underlying `Array` grows and shrinks depending on how “full” the data structure is. The goal is to keep the average

length of each list short enough to achieve in the **expected** time complexity of  $O(1)$  for each operation, but at the same time to not “waste” memory for too many empty array slots.

You should add code to measure a “load factor” between 0 and 1, for example by dividing the number of elements in the hash table by the length of the `Array`. Once a certain load factor is reached, you should create a new (bigger or smaller) `Array` instance, **re-hash** all the elements into it, and then destroy the previous `Array` instance.

## Famous Last Words

Thanks for “holding out” in the course until the “bitter end,” all of us hope that you had some fun programming the various data structures this quarter. Good luck for the final exam!