10. ADTs and Algorithms I

10.1 Indexed Arrays

In the usual places are the files `arr.h`, `arr.c`, `testarr.c` and a makefile. These files enable you to build, and test, the ADT for a 1D Indexed Array. This simple replacement for C arrays is 'safe' in the sense that if you write the array out-of-bounds, it will be automatically resized correctly (using `realloc()`). The interface to this ADT is in `arr.h` and its implementation is in `arr.c`. Typing `make testarr`, compiles these files together with the test file `testarr.c`. Executing `./testarr` should result in all tests passing correctly.

```
% make -f 1d_adt.mk run
Basic Array Tests ... Start
Basic Array Tests ... Stop
```

Exercise 10.1 Build `testarr`, and check that you understand the use of the functions, including initialization, reading, writing and freeing. Use the makefile provided to run the code, and do some memory-leak checking etc. ■

10.2 Sets

Sets are an important concept in Computer Science. They enable the storage of elements (members), guaranteeing that no element appears more than once. Operations on sets include initializing them, copying them, inserting an element, returning their size (cardinality), finding if they contain a particular element, removing an element if it exists, and removing one element from a random position (since sets have no particular ordering, this could be the first element). Other set operations include union (combining two sets to include all elements), and intersection (the set containing elements common to both sets).

www `https://www.mathsisfun.com/sets/sets-introduction.html`

www `https://en.wikipedia.org/wiki/Set_(mathematics)`

The definition of a Set ADT is given in `set.h`, and a file to test it is given in `testset.c`.

Exercise 10.2 Write `set.c`, so that:

```
% make -f set_adt.mk run
./testset
Basic Set Tests ... Start
Basic Set Tests ... Stop
```

works correctly. Your Set ADT will build on top of the Indexed Array ADT introduced in Exercise 10.1. Only write `set.c`. Alter no other files, including `arr.c`, `arr.h`, `set.h` or the Makefile. ■

10.3 Towards Polymorphism

Polymorphism is the concept of writing functions (or ADTs), without needing to specify which particular type is being used/stored. To understand the quicksort algorithm, for instance, doesn't really require you to know whether you're using integers, doubles or some other type. C is not very good at dealing with polymorphism - you'd need something like Python, Java or C++ for that. However, it does allow the use of `void*` pointers for us to approximate it.

Exercise 10.3 Extend the array ADT discussed in Exercise 10.1, so that any type can be used - files `varr.h` and `testvarr.c` are available in the usual place - use the Makefile used there, simply swapping `arr` for `varr` at the top. ■

10.4 Double Hashing

Here we use double hashing, a technique for resolving collisions in a hash table.

Exercise 10.4 Use double hashing to create a spelling checker, which reads in a dictionary file from `argv[1]`, and stores the words.

Make sure the program:

- Use double hashing to achieve this.
- Makes no assumptions about the maximum size of the dictionary files. Choose an initial (prime) array size, created via malloc(). If this gets more than 60% full, creates a new array, roughly twice the size (but still prime). Rehash all the words into this new array from the old one. This may need to be done many times as more and more words are added.
- Uses a hash, and double hash, function of your choosing.
- Once the hash table is built, reads another list of words from `argv[2]` and reports on the *average* number of look-ups required. A *perfect* hash will require exactly 1.0 look-up. Assuming the program works correctly, this number is the only output required from the program.

■

10.5 Separate Chaining

Separate chaining deals with collisions by forming (hopefully small) linked lists out from a base array.

Exercise 10.5 Adapt Exercise 10.4 so that:

- A linked-list style approach is used.
- No assumptions about the maximum size of the dictionary file is made.

- The same hash function as before is used.
- Once the hash table is built, reads another list of words from `argv[2]` and reports on the *average* number of look-ups required. A *perfect* hash will require exactly 1.0 look-up, on average. Assuming the program works correctly, this number is the only output required from the program.

■