Computer Engineering 12
Project 2: Sets and Arrays

1 Introduction

In this project, you will implement a set abstract data type for strings. Your interface and implementation must be kept separate. Separate source files that provide `main` will be provided for testing your data type.

2 Interface

The interface to your abstract data type must provide the following operations:

- `SET *createSet(int maxElts);`
  return a pointer to a new set with a maximum capacity of *maxElts*
- `void destroySet(SET *sp);`
  deallocate memory associated with the set pointed to by *sp*
- `int numElements(SET *sp);`
  return the number of elements in the set pointed to by *sp*
- `void addElement(SET *sp, char *elt);`
  add *elt* to the set pointed to by *sp*
- `void removeElement(SET *sp, char *elt);`
  remove *elt* from the set pointed to by *sp*
- `char *findElement(SET *sp, char *elt);`
  if *elt* is present in the set pointed to by *sp* then return the matching element, otherwise return `NULL`
- `char **getElements(SET *sp);`
  allocate and return an array of elements in the set pointed to by *sp*

3 Implementation

You will write two different implementations of the data type for this assignment. First, implement a set using an unsorted array of length $m > 0$, in which the first $n \leq m$ slots are used to hold n strings in some arbitrary order. Use sequential search to locate an element in the array. Second, implement a set using a sorted array of length $m > 0$, in which the first $n \leq m$ slots are used to hold n strings in ascending order. Use binary search to locate an element in the array.

For both implementations, rather than duplicating the search logic in several functions, write an auxiliary function `search` that returns the location of an element in an array. Declare `search` as static so it is not visible outside the source file. Use `search` to implement the functions in the interface. Your implementation should allocate memory and copy the string when adding, and therefore also deallocate memory when removing.

By the end of the first lab, you should have finished the first implementation. You should verify that the ADT works with both test programs (`unique`, and then `parity`) on all of the files. Note that `unique` takes an optional second file, whose words it ***removes*** from the set, thus allowing you to test insertion followed by deletion. In contrast, `parity` interweaves insertion and deletion, so it is a tougher test. By the end of the second lab, you should have finished both implementations. Before the due date, make sure you finish commenting and testing your code. You should use the `assert` macro defined in `<assert.h>` where appropriate.

4 Submission

Download the `project2.tar` file from the course website to get started. Call your source files `unsorted.c` and `sorted.c`. Complete the file `report.txt` with the results requested below. Submit a `tar` file containing the `project2` directory using the online submission system. Although a `Makefile` is not required, you may find it beneficial to write one. To build an executable program, you will need to combine one of the two test programs with one of your implementations. For example:

```
$ gcc -o unique unique.c unsorted.c
```

If you are using `Xcode`, you can simply include both test programs and implementations as part of your project and then select which files to build. Alternatively, you can have multiple projects that share the same source files.

5 Grading

Your implementation will be graded in terms of correctness, clarity of implementation, and commenting and style. Your implementation ***must*** compile and run on the workstations in the lab. The algorithmic complexity of each function ***must*** be documented. For both implementations, report the execution times of the test programs on each of the sample input files by using the `time` command. (Report the average of the "real" times of at least three runs on each input file.) For example:

```
$ time ./unique /scratch/coen12/Macbeth.txt
```