Computer Engineering 12
Project 3: Sets, Arrays, and Hash Tables

Due: Sunday, Feb 7th at 5:00 pm

1 Introduction

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

2 An ADT for Strings

2.1 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*

2.2 Implementation

Implement a set using a hash table of length $m > 0$ and linear probing to resolve collisions. Create an auxiliary function `search` that contains all of the search logic as you did for the previous assignment, and use `search` to implement the functions in your interface. The following hash function should be used:

```
unsigned strhash(char *s) {
    unsigned hash = 0;

    while (*s != '\0')
        hash = 31 * hash + *s ++;

    return hash;
}
```

As in the previous assignment, your implementation should allocate memory and copy the string when adding, and therefore also deallocate memory when removing.

3 An ADT for Generic Pointer Types

So far, we have only developed ADTs for strings. If we wanted to store another type of data, we would need to copy our implementation and change "`char *`" to the new type, which is both tedious and error-prone. Fortunately, C provides a ***generic pointer type***: a pointer to `void` can be assigned to or from any other pointer type. For example, the function `malloc` returns a pointer to `void` so its result can be assigned to any pointer type. Similarly, the function `free` takes a pointer to `void` as a parameter so it can be passed any pointer type. By changing "`char *`" to "`void *`" in our implementation, we can write an ADT that works on generic pointer types, allowing us to store strings, pointers to structures, or whatever we like. The test program `counts` uses a generic set ADT to store structures rather than just strings in order to count the number of times each word occurs in a file.

Unfortunately, we need to do a little more than just replace "`char *`" with "`void *`" in our implementation. Our implementation needs to be told how to compare two elements as well as compute the hash value for an element. After all, `strcmp` and `strhash` only work on strings. Therefore, our `createSet` function must take extra parameters that are now pointers to these two functions:

```
SET *createSet(int maxElts, int (*compare)(), unsigned (*hash)());
```

These two functions must be stored internally as part of the set structure. Rather than calling `strcmp` and `strhash` in the function `search`, you will need to call the client-provided functions instead. Assuming that the pointer to the set is `sp` and the comparison function is stored as a member called `compare`:

```
(*sp->compare)(...);
```

Our new generic set ADT also does not know how to allocate or deallocate the elements it stores, so rather than calling `strdup` and `free`, it simply copies the pointers themselves. The client is responsible for managing memory. Such is the price we pay for using a generic ADT.

Once you have completely finished your set ADT for strings, copy it to the `generic` directory and modify it so it uses generic pointer types instead of strings. You can test it against the `unique` and `counts` programs in that directory. Note that since the interface is slightly different, you cannot test your implementation against the programs in the `strings` directory. Moving forward, all future ADTs we design will be generic ADTs.

4 Submission

Download the `project3.tar` file from the course website to get started. Call your source file `table.c`. Place the solution for the ADT for strings in the subdirectory `strings`. Complete the file `report.txt` containing the results requested for below. Place the solution for the ADT for generic pointer types in the subdirectory `generic`. Submit a `tar` file containing the entire `project3` directory using the online submission system.

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. 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.)