Assignment 1b: DArray and Hashtable

Assignment written by Julie Zelenski.

And now, on to part b!

Part a was designed as a warm-up, the next installment is going to require quite a bit more of your attention. To build data structures which are efficient and highly reusable in C requires very careful use of `void*`, pointer arithmetic, and typecasts along with dedicated memory management through `malloc`, `realloc`, and `free`. This is exactly the experience you will get this week writing a generic dynamic array and hashtable. This assignment requires a strong understanding of C pointers, memory layout, and function pointers, and will demand your finest debugging skills. Many consider this to be the most difficult program assigned in all of CS107, so don't delay getting started.

This assignment will challenge and frustrate you, but you will find it be quite rewarding in terms of solidifying your understanding of many important and complex programming topics. The ADTs that you implement in HW1a and b are not just useful as a learning experience either—they're actually generic container structures that CS107 students have gone on to use for CS108, CS143, CS140—even industry! Write it once, do a really nice job on it, and reap the benefits for years to come!

Due: Wednesday, October 18th at 11:59 p.m.

The Dynamic Array ADT

Most programs require storing things. The ultimate solution to all these storage needs is to construct a single "generic" storage ADT that can hold a dynamic number of objects, wherein the elements themselves can be of any size. That last point is particularly important, because in the past (i.e. CS106) we resorted to always storing an array of pointers to objects as a work-around for storing objects of differing size, such as when we wanted to use the same code to store a stack of integers and a stack of characters. In this case, we want to construct an array that truly accommodates elements of any base type, without requiring the overhead of a pointer to each element. Enter the DArray!

The DArray is a more flexible extension of C's built-in array type. It has some of the same basic properties: it is managed in memory as a contiguous region of elements, all the elements of any particular array must be homogeneous (i.e. same size and type), and element indexing starts from zero. But our dynamic array differs in that it can automatically grow when elements are appended, it can insert and delete elements in the middle, and it does bounds-checking when accessing elements. In addition, through the use of client-supplied function pointers, it can iterate over all elements, search for an element, and sort the array elements.

The specific requirements of the DArray interface are detailed in its interface file that is attached at the end of this handout. The short summary of its functionality includes:

```
DArray ArrayNew(int elemSize, int numElemsToAllocate, ElemFreeFn fn);
 void  ArrayFree(DArray array);
  int  ArrayLength(DArray array);
 void *ArrayNth(DArray array, int n);
 void  ArrayAppend(DArray array, const void *newElem);
 void  ArrayInsertAt(DArray array, const void *newElem, int n);
 void  ArrayReplaceAt(DArray array, const void *newElem, int n);
 void  ArrayDeleteAt(DArray array, int n);
 void  ArraySort(DArray array, ArrayCompareFn comparator);
  int  ArraySearch(DArray array, const void *key, ArrayCompareFn comp,
                   int fromIndex, bool isSorted);
 void  ArrayMap(DArray array, ArrayMapFn fn, void *clientData);
```

You should make liberal use of the standard C libraries to help you write your code—think of this new abstraction as a layer on top of what the standard facilities provide. In particular, you should use `qsort` and `bsearch` to implement sorting and binary search. You will also want `malloc`, `realloc`, `memcpy`, and `memmove` for memory management. The DArray will be so heavily used later that making it efficient is a moderately interesting goal.

The Generic HashTable ADT

The second powerful tool you will build is a generic HashTable ADT. Hashing is a particularly efficient method for storing and retrieving elements by partitioning the elements into buckets based on a hash function that maps elements to integers. Often we use strings as keys to identify and hash elements, but as with the DArray, we're trying to remove such restrictions on the element type and allow elements of any type to be stored in our hashtable.

When creating a HashTable, the client specifies the element size and the number of hash buckets. Because the implementor has no knowledge about the structure of the elements, the client must also supply the hash function and the comparison function to be used on the elements. Hash collisions should be resolved by chaining (i.e. you should keep all elements which collide in one bucket).

You will use the DArray when implementing the HashTable. The list of elements in each bucket, for example, should be kept in a DArray. Be sure you leverage as much of the DArray functionality as you can. You've already written that code once and it's much more fun (and a better design) to just call functions that work than to re-implement the behavior.

It is not necessary to use a DArray for the list of buckets themselves. Since that array doesn't change size once created and is always the same base type, an ordinary C array works fine.
The detailed interface specification for the HashTable ADT is attached to the end of the handout. As a preview, the list of operations supported will be:

```
HashTable TableNew(int elemSize, int nBuckets, TableHashFn hashFn,
                   TableCompareFn compFn, TableElemFreeFn freeFn);
void  TableFree(HashTable table);
void  TableEnter(HashTable table, const void *newElem);
void *TableLookup(HashTable table, const void *elemKey);
void  TableMap(HashTable table, TableMapFn fn, void *clientData);
 int  TableCount(HashTable table);
```

Read the interface files!

The descriptions in this handout are sparse—all of the important facts about the workings of the DArray and HashTable are specified in their header files (which are reprinted at the end of this handout). It is important that you become familiar with the skill of reading a specification from a `.h` file and then creating the required implementation to match it. Be sure to very carefully read the header files before doing any implementation! I mean it!

The home page search engine

For the final part of this assignment, you'll put on your client hat and work with the collections you have just written. To first help you test them in isolation, we've provided a simple test file with code that exercises the basic functionality. You can use these test routines to verify that your ADTs are doing their jobs. Also, if you don't understand the usage or behavior of a particular function, this test code a good place to look for example usage.

Once they pass our simple tests, you can bring these fancy new ADTs to the rescue of the poor simple-minded home page parser from HW1a. Now with some sophisticated functionality to build on, you can re-visit this task and do a much better job of it. This time, rather than just counting the words that appear, you are going to construct a little "mini-WebCrawler" that will let the user search for a given word and see a list of all the users whose home page include that word. Much of the code you have written for `analyze` will get re-used. This is yet another chance to try your hand at creating a new program out of an already-written program that does something close to what you want, but needs some rearrangement and some additional features. This pattern is incredibly common in industry work, so this is a good skill to sharpen.

Take the same approach as before: your search program takes one argument, the file of usernames to analyze. You pull the usernames out of this file, open the pages one by one and extract the tokens. This time instead of keeping the frequency count for each word, you now keep a DArray that lists the names of the users that contain that word.

Don't allow duplicates— no matter how many times the word "beer" appears in Homer's page or "trouble" in Bart's, just enter their name once in the DArray of matches. Don't worry about keeping the names in any particular order while building the table, but after indexing all the pages, go back and sort each of the arrays of names into alphabetical order so you can print results that way.

In order to make it quick to store and lookup by word, you will use your ever-handy and very efficient HashTable as the data structure for matching a word to the DArray of users that reference it. Unlike HW1a, you won't need to introduce any arbitrary maximum on the number of words you track or number of users tracked per word because your underlying data structures are much more flexible this time.

Please use the same set of delimiters and maximum token length as we gave you to use in HW1a. As before, only store words that are at least 4 characters long and contain only ASCII characters. You should continue to skip over all words inside HTML tokens. You will have an upper bound on the large, temporary buffer used to scan in the usernames and tokens, but the data structure (the combination of hashtable / darrays) you use to store the words and usernames should not have any required minimum or hard maximum on the string length. This means you must make a copy of the string trimmed down to just the right size and store the entries as `char *` instead of storing the words or usernames using a fixed buffer for each entry.

After you have scanned all the pages, your program will examine the hashtable and print a summary of the index statistics: the number of unique words (i.e. the number of entries in the hashtable) and the total number of references (each word has one or more pages that reference it, the sum of those counts over all the words is the total number of references). Finally you need to build a simple interactive loop that asks the user to enter a word which you will then look up in the hashtable. (Hint: when you need to parse input typed in by the user, don't forget what's a good way to scan tokens from a file…) Report the number of matches and then ask the user if they would like to see the full list. Use the `ArrayMap` function to print the list containing the names of the users, one per line, numbered in order, with no name should be repeated. (It's a little brainteaser to figure out how to use `ArrayMap` to number the items.) The names should be printed in alphabetical order.

For example, here is an excerpt that shows what we expect, your output should look just like this:

```
% search /usr/class/cs107/assignments/students

Welcome to the leland homepage frequency service!
Hold on while I process the home pages (may take a while)...
Done. Table contains 2455 unique words and 5125 total references.

Please enter a word to search for ("quit" to quit): apple
Found 3 matches for "weird". See the list? (y/n): y
      1) anakken
      2) andygray
      3) briehl

Please enter a word to search for ("quit" to quit): stanford
Found 51 matches for "stanford". See the list? (y/n): n
```

Although there is still the slow file opening and reading that can't be avoided, the smarter data structures and better underlying algorithms will definitely trim time off the handling of the word tabulation, even though it is doing much more work this time than just counting. And notice that once the table is built, the queries are instantaneous.

It's fun to experiment with various allocation chunk sizes for the DArray, different hash strategies, changing the number of buckets in the hash table, etc. to learn how those factors affect the performance of your ADTs. As a few starting suggestions, since the number of unique words is about 3,000, you want to use about that number of buckets in order to get solid performance from your hashtable. For the DArrays storing the users, each array is usually pretty small (just one or two entries) so you probably want to use a fairly small chunk allocation size (say 2 or 4) to minimize waste. If you're feeling ambitious, you can try indexing many more pages that just the 107 users to prove to yourself how well your data structures will stand up under heavy use.

A note on hashing and comparing strings

For the home page searching client, you will need a hash function that can do a string hash on the word. The construction of an appropriate hash function is a topic beyond the scope of this course, thus for hashing strings, please just take the simple hash function you'll find waiting for you in the `search.c` file. It is based on the standard linear congruence hash which is quite effective and this adaptation is designed to be case-insensitive ("Stanford" and "sTaNFOrd" will hash to same thing). You also should be using the case-insensitive `strcasecmp` comparison routine instead of `strcmp`. Relying on the two of them, your home page search program should be case-insensitive (which is really what you want).

C library functions

Our class position on using library functions will be "You can always use anything from the standard ANSI libraries (i.e. if it's in Kernighan & Ritchie, it's fine by us)." So all the

string functions, I/O functions, memory functions, etc. are at your disposal. For this assignment, you are going to want to look at least these functions: `malloc`, `realloc`, `free`, `memcpy`, `memmove`, `qsort`, `bsearch`, `assert`. You have probably seen some, but not all, of these functions before. Please read the man pages on-line or refer to a copy of K & R for the function interfaces and implementation details of the functions you aren't yet familiar with. We may talk about these functions very briefly in class, but mostly you'll need to rely on your own initiative and resourcefulness to get acquainted with them.

Other non-standard library functions that you find on your particular system are not okay unless we specifically state otherwise. For example, the non-standard `lfind` and `lsearch` are off-limits for this program. If you run across other non-ANSI functions that you would like to use, you should clear it with one of the course staff first.

Helpful suggestions from students past

Although the handout describes the assignment requirements in the order of DArray, HashTable, and then search client, students from the past have suggested we recommend you write the client program first, then go back and develop the ADTs. Many have found the client to be the trickiest part and writing it when you still aren't sure if the earlier parts are entirely correct is especially challenging. Using the ADTs as a client helps you to understand how they operate externally which is valuable experience for when you put your implementor hat on and need to fill in the guts.

We provide compiled versions of the completed ADTs that you can use right away and develop the client on (see `/usr/class/cs107/assignments/hw1b_sample`). Read the `.h` files carefully and write your client code to use the functions in accordance with the public interface specification and you might find it a particularly good way to get started.

We suggested this approach before and many folks tried and found it to be a good strategy. But others believed that starting with the ADTs made more sense to them. The jury appears to still be out on this issue. We encourage you to consider starting with the client, but ultimately, you are free to do what works best for you. Another suggestion of past students we heartily recommend is drawing a picture of what you expect your data structures to look like, where are the pointers and what do they point to. It is essential you understand how your memory is being laid out so that you can properly manage it.

Optional (but extremely valuable) fill-in-the-blank exercise

Any C program that requires use of `void *` is guaranteed to have tricky issues and being a client of these generic ADTs is no exception. To help you think about how to be a successful client, see if you can lead yourself through the following examples and fill in the blanks.

Let's say you want to use the DArray to store a list of integers taken from a normal C array. Take a look at this starting code:

```
DArray collection;
int i, last, entries[] = {34, 15, 17}; // put these 3 values in the DArray

collection = ArrayNew(____, 3, ____);
for (i = 0; i < 3; i++)                    // add members from array
   ArrayAppend(collection, ____);

                     // get last element back out of DArray
last = ____ ArrayNth(collection, ArrayLength(collection) - 1);
```

The tricky parts are left blank for you to fill in. There are three critical things you need to understand to work with the DArray:

- How do you properly create a DArray that will store the desired type?
- When you are inserting or appending a new value, how do you pass the element as a parameter?
- When you are retrieving an element via ArrayNth or manipulating it in a callback function during a search or map operation, what do you need to cast the `void *` to? Do you need to dereference it? How many times?

Getting this correct for the basic types such as integers, floats, or structs never seems to be too much trouble for students. But now consider when the type you want to store in the DArray is a already a pointer, for example, if you were storing strings:

```
DArray collection;
int i;
char *last, *entries[] = {"Apple", "Banana", "Pear"};

collection = ArrayNew(____, 3, ____);
for (i = 0; i < 3; i++)                    // add members from array
   ArrayAppend(collection, ____);

last = ____ ArrayNth(collection, ArrayLength(collection) - 1);
```

How do you fill the blanks in this one? Is there anything different about how you store and retrieve an element that is a pointer type? Be careful! What is an easy mistake to make in this case that didn't come up in the integer example? What will happen if you make that mistake?

Think through what is going on, maybe even draw out a memory diagram and try typing in the above code and getting it to work. The goal of this exercise is for you to get a solid understanding of how to use of the ADT, for both pointer and non-pointer types. Also by realizing where the most likely pitfalls are, you are in a better position to find and fix those errors if they somehow wander into your own code.

A few last thoughts

- As mentioned before in this handout, HW1b is considered by many to be the most challenging of all the CS107 assignments. In particular, the searching client has a few tricky little details that often cause trouble for students, so make sure you start early enough that you have plenty of time to work through those issues and can take advantage of our help in office hours and email.
- HW1a was straightforward enough you may not have ever needed the help of the debugger, but starting with this program, you want to start getting familiar with gdb to help you sort out things when difficult bugs surface. Information on the gdb debugger was included in the big UNIX handout and in the Thursday section. If you missed it, you might want to bug someone who went and get them to help you tackle gdb. It's an extremely good debugger.
- Don't forget Purify— it can be a real help in tracking down those pesky memory problems that are easy to run into when you are doing a lot of pointer and memory manipulations, like you will for this program. If your program has some undesired behavior or surprise crashes, running it under Purify may help you pinpoint the root cause. The `man` page on Purify is chock full of good information as well.
- The DArray and HashTable implementations are rife with opportunities for using `assert` to ensure no important assumptions are being violated. See the interface specifications for some suggestions about conditions appropriate for asserts. In fact, for any unrecoverable error, an assert is much preferred to blundering on into a segmentation fault.
- All the same expectations apply about design, decomposition, code factoring, no globals, good style, commenting, and so on. By now this is second nature to you...
- As always, we expect that your code compile cleanly— i.e. without any warnings— and that it run under Purify without finding any errors (other than any unfortunate mis-reports from Purify that we will try to identify for you in the FAQ.)

Getting started

The starting project is in the leland directory `/usr/class/cs107/assignments/hw1b`**.** This directory contains the .h files, the skeleton .c files, the simple adttest client code, and a makefile that builds the project. You'll note that the makefile makes two projects: the `adttest` program (our sample client code) and the `search` program (which will be your home page searching client) and purify versions of both. Refer to the `Makefile` for a little more information about how to use the various targets.

You want to make your own copy of the project directory
(i.e. `cp -r /usr/class/cs107/assignments/hw1b ~`) to get started. You can also get the starting files via anonymous ftp to leland or from the class Web site `http://cse.stanford.edu/classes/cs107/`. Be sure to get all the files, including the necessary `.purify` configuration file.

HW1b Deliverables

You should electronically submit your entire project using the leland submit script (described on submit policy on web site). What you submit should make cleanly with no changes, so be sure to include all necessary files (including your scanner). As far as hard copies are concerned, my sense is that we won't be needing them any more, since we expect to grade the electronic submissions by online testing, and we'll eyeball the sections of the code we know to be tough and provide feedback via email. I may change my mind on this, but it seems like an awfully huge waste of paper to make you all print everything out.

```
#ifndef _DARRAY_H
#define _DARRAY_H

/*
 * File: darray.h
 * --------------
 * Defines the interface for the DynamicArray ADT.
 * The DArray allows the client to store any number of elements of any desired
 * base type and is appropriate for a wide variety of storage problems. It
 * supports efficient element access, and appending/inserting/deleting elements
 * as well as optional sorting and searching. In all cases, the DArray imposes
 * no upper bound on the number of elements and deals with all its own memory
 * management. The client specifies the size (in bytes) of the elements that
 * will be stored in the array when it is created. Thereafter the client and
 * the DArray can refer to elements via (void*) ptrs. The memory management
 * alone more than justifies the cost of the ADT- it should provide you with
 * years of faithful service. Just wipe occasionally with a damp cloth.
 */

/*
 * Type: DArray
 * ----------------
 * Defines the DArray type itself. The client can declare variables of type
 * DArray, but these variables must be initialized with the result of ArrayNew.
 * The DArray is implemented with pointers, so all client copies in variables
 * or parameters will be "shallow" -- they will all actually point to the
 * same DArray structure.  Only calls to ArrayNew create new arrays.
 * The struct declaration below is "incomplete"- the implementation
 * details are literally not visible in the client .h file.
 */

typedef struct DArrayImplementation *DArray;

/*
 * ArrayCompareFn
 * --------------
 * ArrayCompareFn is a pointer to a client-supplied function which the
 * DArray uses to sort or search the elements. The comparator takes two
 * (const void*) pointers (these will point to elements) and returns an int.
 * The comparator should indicate the ordering of the two elements
 * using the same convention as the strcmp library function:
 * If elem1 is "less than" elem2, return a negative number.
 * If elem1 is "greater than" elem2, return a positive number.
 * If the two elements are "equal", return 0.
 */

typedef int (*ArrayCompareFn)(const void *elem1, const void *elem2);

/*
 * ArrayMapFn
 * ----------
 * ArrayMapFn defines the space of functions that can be used to map over
 * the elements in a DArray.  A map function is called with a pointer to
 * the element and a client data pointer passed in from the original
 * caller.
 */

typedef void (*ArrayMapFn)(void *elem, void *clientData);
```

```
/*
 * ArrayElementFreeFn
 * ------------------
 * ArrayElementFreeFn defines the space of functions that can be used as the
 * clean-up function for an element as it is deleted from the array
 * or when the entire array of elements is freed.  The cleanup function is
 * called with a pointer to an element about to be deleted.
 */

typedef void (*ArrayElementFreeFn)(void *elem);

/*
 * ArrayNew
 * --------
 * Creates a new DArray and returns it. There are zero elements in the array.
 * to start. The elemSize parameter specifies the number of bytes that a single
 * element of this array should take up.  For example, if you want to store
 * elements of type Binky, you would pass sizeof(Binky) as this parameter.
 * An assert is raised if the size is not greater than zero.
 *
 * The numElemsToAllocate parameter specifies the initial allocated length
 * of the array, as well as the dynamic reallocation increment for when the
 * array grows.  Rather than growing the array one element at a time as
 * elements are added (which is rather inefficient), you will grow the array
 * in chunks of numElemsToAllocate size.  The "allocated length" is the number
 * of elements that have been allocated, the "logical length" is the number of
 * those slots actually being currently used.
 *
 * A new array is initially allocated to the size of numElemsToAllocate, the
 * logical length is zero.  As elements are added, those allocated slots fill
 * up and when the initial allocation is all used, grow the array by another
 * numElemsToAllocate elements. You will continue growing the array in chunks
 * like this as needed.  Thus the allocated length will always be a multiple
 * of numElemsToAllocate.  Don't worry about using realloc to shrink the array
 * allocation if many elements are deleted from the array.  It turns out that
 * many implementations of realloc don't even pay attention to such a request
 * so there is little point in asking.  Just leave the array over-allocated.
 *
 * The numElemsToAllocate is the client's opportunity to tune the resizing
 * behavior for their particular needs.  If constructing large arrays,
 * specifying a large allocation chunk size will result in fewer resizing
 * operations.  If using small arrays, a small allocation chunk size will
 * result in less space going unused. If the client passes 0 for
 * numElemsToAllocate, the implementation will use the default value of 8.
 *
 * The elemFreeFn is the function that will be called on an element that
 * is about to be deleted (using ArrayDeleteAt) or on each element in the
 * array when the entire array is being freed (using ArrayFree).  This function
 * is your chance to do any deallocation/cleanup required for the element
 * (such as freeing any pointers contained in the element). The client can pass
 * NULL for the cleanupFn if the elements don't require any handling on free.
 */

DArray ArrayNew(int elemSize, int numElemsToAllocate,
                ArrayElementFreeFn elemFreeFn);
```

```
/* ArrayFree
 * ----------
 * Frees up all the memory for the array and elements. It DOES NOT
 * automatically free memory owned by pointers embedded in the elements.
 * This would require knowledge of the structure of the elements which the
 * DArray does not have. However, it will iterate over the elements calling
 * the elementFreeFn earlier supplied to ArrayNew and therefore, the client,
 * who knows what the elements are, can do the appropriate deallocation of any
 * embedded pointers through that function.  After calling this, the value of
 * what array is pointing to is undefined.
 */
void ArrayFree(DArray array);


/* ArrayLength
 * -----------
 * Returns the logical length of the array, i.e. the number of elements
 * currently in the array.  Must run in constant time.
 */
int ArrayLength(const DArray array);


/* ArrayNth
 * --------
 * Returns a pointer to the element numbered n in the specified array.
 * Numbering begins with 0.  An assert is raised if n is less than 0 or greater
 * than the logical length minus 1. Note this function returns a pointer into
 * the DArray's element storage, so the pointer should be used with care.
 * This function must operate in constant time.
 *
 * We could have written the DArray without this sort of access, but it
 * is useful and efficient to offer it, although the client needs to be
 * careful when using it. In particular, a pointer returned by ArrayNth
 * becomes invalid after any calls which involve insertion, deletion or
 * sorting the array, as all of these may rearrange the element storage.
 */
void *ArrayNth(DArray array, int n);


/* ArrayAppend
 * -----------
 * Adds a new element to the end of the specified array.  The element is
 * passed by address, the element contents are copied from the memory pointed
 * to by newElem.  Note that right after this call, the new element will be
 * the last in the array; i.e. its element number will be the logical length
 * minus 1. This function must run in constant time (neglecting
 * the memory reallocation time which may be required occasionally).
 */
void ArrayAppend(DArray array, const void *newElem);

/* ArrayInsertAt
 * -------------
 * Inserts a new element into the array, placing it at the position n.
 * An assert is raised if n is less than 0 or greater than the logical length.
 * The array elements after position n will be shifted over to make room. The
 * element is passed by address, the new element's contents are copied from
 * the memory pointed to by newElem. This function runs in linear time.
 */
void ArrayInsertAt(DArray array, const void *newElem, int n);
```

```
/* ArrayDeleteAt
 * -------------
 * Deletes the element numbered n from the array. Before being removed,
 * the elemFreeFn that was supplied to ArrayNew will be called on the element.
 * An assert is raised if n is less than 0 or greater than the logical length
 * minus one. All the elements after position n will be shifted over to fill
 * the gap.  This function runs in linear time. It does not shrink the
 * allocated size of the array when an element is deleted, the array just
 * stays over-allocated.
 */
void ArrayDeleteAt(DArray array, int n);


/* ArrayReplaceAt
 * -------------
 * Overwrites the element numbered n from the array with a new value. Before
 * being overwritten, the elemFreeFn that was supplied to ArrayNew is called
 * on the old element. Then that position in the array will get a new value by
 * copying the new element's contents from the memory pointed to by newElem.
 * An assert is raised if n is less than 0 or greater than the logical length
 * minus one. None of the other elements are affected or rearranged by this
 * operation and the size of the array remains constant. This function must
 * operate in constant time.
 */
void ArrayReplaceAt(DArray array, const void *newElem, int n);


/* ArraySort
 * ---------
 * Sorts the specified array into ascending order according to the supplied
 * comparator.  The numbering of the elements will change to reflect the
 * new ordering. An assert is raised if the comparator is NULL.
 */
void ArraySort(DArray array, ArrayCompareFn comparator);


#define NOT_FOUND -1    // returned when a search fails to find the key

/* ArraySearch
 * -----------
 * Searches the specified array for an element whose contents match
 * the element passed as the key.  Uses the comparator argument to test
 * for equality. The "fromIndex" parameter controls where the search
 * starts looking from. If the client desires to search the entire array,
 * they should pass 0 as the fromIndex. The function will search from
 * there to the end of the array. The "isSorted" parameter allows the client
 * to specify that the array is already in sorted order, and thus it uses a
 * faster binary search.  If isSorted is false, a simple linear search is
 * used. If a match is found, the position of the matching element is returned
 * else the function returns NOT_FOUND.  Calling this function does not
 * re-arrange or change contents of DArray or modify the key in any way.
 * An assert is raised if fromIndex is less than 0 or greater than
 * the logical length (although searching from logical length will never
 * find anything, allowing this case means you can search an entirely empty
 * array from 0 without getting an assert). An assert is raised if the
 * comparator is NULL.
 */
int ArraySearch(DArray array, const void *key, ArrayCompareFn comparator,
                int fromIndex, bool isSorted);
```

```
/* ArrayMap
 * -----------
 * Iterates through each element in the array in order (from element 0 to
 * element n-1) and calls the function fn for that element.  The function is
 * called with the address of the array element and the clientData pointer.
 * The clientData value allows the client to pass extra state information to
 * the client-supplied function, if necessary.  If no client data is required,
 * this argument should be NULL. An assert is raised if map function is NULL.
 */
void ArrayMap(DArray array, ArrayMapFn fn, void *clientData);

#endif _DARRAY_
```

```
#ifndef _HASHTABLE_H
#define _HASHTABLE_H

/* File: hashtable.h
 * -----------------
 * Defines the interface for the HashTable ADT.
 * The HashTable allows the client to store any number of elements of any
 * type in a hash table for fast storage and retrieval.  The client specifies
 * the size (in bytes) of the elements that will be stored in the table when
 * it is created. Thereafter the client and the HashTable refer to elements
 * via (void*) ptrs. The HashTable imposes no upper bound on the number of
 * elements and deals with all its own memory management.
 *
 * The client-supplied information (in the form of the number of buckets
 * to use and the hashing function to be applied to each element) is employed
 * to divide elements in buckets with hopefully only few collisions, resulting
 * in Enter & Lookup performance in constant-time.  The HashTable also supports
 * iterating over all elements by use of mapping function.
 */

/* Type: HashTable
 * ---------------
 * Defines the HashTable type itself. The client can declare variables of type
 * HashTable, but these variables must be initialized with the result of
 * TableNew.  The HashTable is implemented with pointers, so all client
 * copies in variables or parameters will be "shallow" -- they will all
 * actually point to the same HashTable structure.  Only calls to TableNew
 * create new tables. The struct declaration below is "incomplete"- the
 * implementation details are literally not visible in the client .h file.
 */
typedef struct HashImplementation *HashTable;


/* TableHashFn
 * -----------
 * TableHashFn is a pointer to a client-supplied function which the
 * HashTable uses to hash elements. The hash function takes a (const void*)
 * pointer to an element and the number of buckets and returns an int,
 * which represents the hash code for this element.  The returned hash code
 * should be within the range 0 to numBuckets-1 and should be stable (i.e.
 * an element's hash code should not change over time).
 * For best performance, the hash function should be designed to
 * uniformly distribute elements over the available number of buckets.
 */
typedef int (*TableHashFn)(const void *elem, int numBuckets);


/* TableCompareFn
 * --------------
 * TableCompareFn is a pointer to a client-supplied function which the
 * HashTable uses to compare elements. The comparator takes two
 * (const void*) pointers (these will point to elements) and returns an int.
 * The comparator should indicate the ordering of the two elements
 * using the same convention as the strcmp library function:
 * If elem1 is "less than" elem2, return a negative number.
 * If elem1 is "greater than" elem2, return a positive number.
 * If the two elements are "equal", return 0.
 */
typedef int (*TableCompareFn)(const void *elem1, const void *elem2);
```

```
/* TableMapFn
 * ----------
 * TableMapFn defines the space of functions that can be used to map over
 * the elements in a HashTable.  A map function is called with a pointer to
 * the element and a client data pointer passed in from the original caller.
 */
typedef void (*TableMapFn)(void *elem, void *clientData);



/* TableElementFreeFn
 * ------------------
 * TableElementFreeFn defines the space of functions that can be used as the
 * clean-up function for each element as it is deleted from the array
 * or when the entire array of elements is freed.  The cleanup function is
 * called with a pointer to an element about to be deleted.
 */
typedef void (*TableElementFreeFn)(void *elem);



/* TableNew
 * --------
 * Creates a new HashTable with no entries and returns it. The elemSize
 * parameter specifies the number of bytes that a single element of the
 * table should take up. For example, if you want to store elements of type
 * Binky, you would pass sizeof(Binky) as this parameter. An assert is
 * raised if this size is not greater than 0.
 *
 * The nBuckets parameter specifies the number of buckets that the elements
 * will be partitioned into.  Once a HashTable is created, this number does
 * not change.  The nBuckets parameter must be in synch with the behavior of
 * the hashFn, which must return a hash code between 0 and nBuckets-1.
 * The hashFn parameter specifies the function that is called to retrieve the
 * hash code for a given element.  See the type declaration of TableHashFn
 * above for more information.  An assert is raised if nBuckets is not
 * greater than 0.
 *
 * The compFn is used for testing equality between elements.  See the
 * type declaration for TableCompareFn above for more information.
 *
 * The elemFreeFn is the function that will be called on an element that is
 * about to be overwritten (by a new entry in TableEnter) or on each element
 * in the table when the entire table is being freed (using TableFree).  This
 * function is your chance to do any deallocation/cleanup required,
 * (such as freeing any pointers contained in the element). The client can pass
 * NULL for the cleanupFn if the elements don't require any handling on free.
 * An assert is raised if either the hash or compare functions are NULL.
 */

HashTable TableNew(int elemSize, int nBuckets,
                   TableHashFn hashFn, TableCompareFn compFn,
                   TableElementFreeFn freeFn);
```

```
/* TableFree
 * ----------
 * Frees up all the memory for the table and its elements. It DOES NOT
 * automatically free memory owned by pointers embedded in the elements. This
 * would require knowledge of the structure of the elements which the HashTable
 * does not have.  However, it will iterate over the elements calling
 * the elementFreeFn earlier supplied to TableNew and therefore, the client,
 * who knows what the elements are,can do the appropriate deallocation of any
 * embedded pointers through that function.
 * After calling this, the value of what table points to is undefined.
 */
void TableFree(HashTable table);


/* TableCount
 * ----------
 * Returns the number of elements currently in the table.
 */
int TableCount(HashTable table);



/* TableEnter
 * ----------
 * Enters a new element into the table. Uses the hash function to determine
 * which bucket to place the new element. Its contents are copied from the
 * memory pointed to by newElem. If there is already an element in the table
 * which is determined to be equal (using the comparison function) this will
 * use the contents of the new element to replace the previous element,
 * calling the free function on the replaced element.
 */
void TableEnter(HashTable table, const void *newElem);



/* TableLookup
 * ----------
 * Returns a pointer to the table element which matches the elemKey parameter
 * (equality is determined by the comparison function).  If there is no
 * matching element, returns NULL. Calling this function does not
 * re-arrange or change contents of the table or modify elemKey in any way.
 */
void *TableLookup(HashTable table, const void *elemKey);



/* TableMap
 * -----------
 * Iterates through each element in the table (in any order) and calls the
 * function fn for that element. The function is called with the address of
 * the table element and the clientData pointer. The clientData value allows
 * the client to pass extra state information to the client-supplied function,
 * if necessary.  If no client data is required, this argument should be NULL.
 * An assert is raised if the map function is NULL.
 */
void TableMap(HashTable table, TableMapFn fn, void *clientData);


#endif _HASHTABLE_H
```