# The Glasgow ADT library

J. Sventek, University of Glasgow

version 1.0
25 June 2012

When constructing sophisticated applications, one often needs to use well-known abstract data types. If one is using the Java programming language, the Java collection classes provide a rich set of such data type implementations. C++ has the Standard Template Library, along with many other contributed template libraries. When programming in C, one often reinvents just enough of a required ADT to get the job done. In addition to the requirement for general availability of complete implementations of these data types, 21st century programming often demands thread-safe versions, as well.

Through the judicious use of the **`void *`** type, C provides the ability to create generic abstract data types, similar to the way Java collection classes were done before the incorporation of generics. The Glasgow ADT library is a small set of abstract data types that have proved generally useful. Each such ADT is patterned after a corresponding Java collection class and is a complete implementation. Two versions are provided for each ADT, one thread-safe and the other not. Each ADT provides an iterator factory method; each such iterator can then be used to linearly progress through the ADT elements in a standard way.

The following describes the general structure for a non-thread-safe ADT, a thread-safe ADT, and how these link to and interact with the corresponding iterator classes.

## General Structure and Example

Assume we are building a simple, generic **`Stack`** ADT. We need to be able to perform create, destroy and purge operations on a stack, as well as push, pop, and peek elements to/from a stack. Since this is a generic **`Stack`**, the items pushed and popped will be **`void *`** elements. We also want the stack to resize itself when it is full and to be able to specify an initial size for the stack when it is created.

### The non-thread-safe interface

To enforce data hiding, we use opaque structure definitions in the corresponding header file:

```
typedef struct stack Stack;
```

A client of the ADT possesses variables of type **`Stack *`**, an approximate equivalent to the object reference used in Java.

#### Creation

The method for creating an instance of a stack has the following function signature:

```
Stack *stack_create(long size);
```

If **`size`** has a value of **`0L`**, then a stack with a default size is created; this default size is documented in a comment in the header file.

**`stack_create()`** allocates an instance of **`Stack`** on the heap, initializes its fields, and returns a pointer to that Stack instance as the value of the function. If there is a heap allocation failure, the value **`NULL`** is returned.

## Destruction

C requires that a programmer manage all memory allocated on the heap programmatically. The ADTs in the Glasgow library assume that the calling program has retained responsibility for the management of any `void *` elements stored in an ADT unless explicitly transferred to the ADT.

Destruction of a stack is one of those instances when the management responsibility is transferred to the ADT. The signature is as follows:

```
void stack_destroy(Stack *st, void (*freeFxn)(void*));
```

If the **`freeFxn`** function argument is non **`NULL`**, **`stack_destroy()`** visits each element on the stack and invokes **`(*freeFxn)()`** on that element; the presumption is that **`freeFxn`** knows how to return any heap storage associated with an element. **`stack_destroy()`** then returns any heap storage associated with the **`Stack`** implementation, and finally returns **`st`** to the heap. After **`stack_destroy()`** returns, it is illegal to attempt to invoke **`stack_*`** methods on **`st`**.

## Purge

Rather than destroy a stack, a program's use may require that it be able to purge all remaining elements from a stack. This is achieved through the use of the following method:

```
void stack_purge(Stack *s, void (*freeFxn)(void*));
```

**`freeFxn`** has the same meaning as for **`stack_destroy()`**. Upon return from **`stack_purge()`**, the stack represented by **`st`** is empty – i.e. a pop operation will generate an error.

## Push, pop, and peek operations

A common pattern used in C programs is to reserve the function return value to indicate success or failure of the function call. In most cases, a return value of 1 (which is true in C) indicates success, while a return value of 0 indicates failure; the pthreads API[1] is a notable exception that deserves universal condemnation for violating this norm. We will ***not*** violate this norm.

If a function is to return anything other than status, that return must be done through a pointer argument. This will be seen in the signatures below.

```
int stack_push(Stack *s, void *item);
int stack_pop(Stack *s, void **item);
int stack_peek(Stack *s, void **item);
```

In each case, if the return value of the function is 1, then the operation has been successful. For a successful **`pop`** or **`peek`**, the item popped or peeked is returned in **`*item`**. An unsuccessful **`push`** indicates that the stack is full ***and*** cannot be extended; an unsuccessful **`pop`** or **`peek`** indicates that the stack was empty.

## Other reasonable operations

Although not listed in our initial enumeration of required operations, three other operations are of general utility to programmers, and are often included in complete implementations of collection classes.

```
int stack_isEmpty(Stack *s);
```

**`stack_isEmpty`** returns 1 if the stack is empty, or 0 if it has at least one element.

```
long stack_size(Stack *s);
```

[1] OpenGroup standard C064, Extended API Set, Part 3.

**`stack_size`** returns the number of elements currently in the stack. Obviously, **`stack_isEmpty()`** returns 1 if **`stack_size() == 0L`**.

**`void **stack_toArray(Stack *s, long *len);`**

An array of **`void *`** pointers to the elements in the stack is returned, with the number of elements in the array returned in **`*len`**. The 0th element of the returned array points to the element that would be returned by a **`peek`** call to the **`Stack`**. The array of **`void *`** pointers is allocated on the heap, so must be returned by a call to **`free()`** when the caller has finished using it.

## Iterating over the Stack

Appendix A presents the interface for a generic iterator. The Stack ADT must provide a factory method to create an **`Iterator`** over the **`Stack`**; this **`Iterator`** can then be manipulated and destroyed using the methods for a generic iterator.

**`Iterator *stack_it_create(Stack *s);`**

If the **`it_create`** method is successful, a pointer to the iterator is returned; if not, **`NULL`** is returned. The first **`next`** call to the **`Iterator`** will return the same element as a **`peek`** call on the **`Stack`**; subsequent **`next`** calls will traverse down the stack.

## The complete header file

```
#include "iterator.h"

/*
 * interface definition for generic stack implementation
 *
 * patterned roughly after Java 6 Stack generic class
 */

typedef struct stack Stack;      /* opaque type definition */

/*
 * create a stack with the specified capacity; if capacity == 0, a
 * default initial capacity (50 elements) is used
 *
 * returns a pointer to the stack, or NULL if there are malloc() errors
 */
Stack *stack_create(long capacity);

/*
 * destroys the stack; for each occupied position, if freeFxn != NULL,
 * it is invoked on the element at that position; the storage associated with
 * the stack is then returned to the heap
 */
void stack_destroy(Stack *st, void (*freeFxn)(void *element));

/*
 * purges all elements from the stack; for each occupied position,
 * if freeFxn != NULL, it is invoked on the element at that position;
 * any storage associated with the element in the stack is then
 * returned to the heap
 *
 * upon return, the stack will be empty
 */
void stack_purge(Stack *st, void (*freeFxn)(void *element));

/*
 * pushes `element' onto the stack; if no more room in the stack, it is
 * dynamically resized
 *
 * returns 1 if successful, 0 if unsuccessful (malloc errors)
 */
int stack_push(Stack *st, void *element);
```

```
/*
 * pops the element at the top of the stack into `*element'
 *
 * returns 1 if successful, 0 if stack was empty
 */
int stack_pop(Stack *st, void **element);

/*
 * peeks at the top element of the stack without removing it;
 * returned in `*element'
 *
 * returns 1 if successful, 0 i stack was empty
 */
int stack_peek(Stack *st, void **element);

/*
 * returns 1 if stack is empty, 0 if it is not
 */
int stack_isEmpty(Stack *st);

/*
 * returns the number of elements in the stack
 */
long stack_size(Stack *st);

/*
 * returns an array containing all of the elements of the stack in
 * proper sequence (from top to bottom element); returns the length of
 * the list in `len'
 *
 * returns pointer to void * array of elements, or NULL if malloc failure
 */
void **stack_toArray(Stack *st, long *len);

/*
 * create generic iterator to this stack;
 * successive next calls return elements in proper sequence (top to bottom)
 *
 * returns pointer to the Iterator or NULL if failure
 */
Iterator *stack_it_create(Stack *st);
```

## The non-thread-safe implementation

### The preliminaries

```
        /*
         * implementation for generic stack
         */

        #include "stack.h"
        #include <stdlib.h>

        #define DEFAULT_CAPACITY 50L
        #define MAX_INIT_CAPACITY 1000L

        struct stack {
           long capacity;
           long delta;
           long next;
           void **theArray;
        };
```

As always with an ADT, the implementation must include the corresponding interface definition, thus assuring that we have the correct function signatures in the implementation. We then define the default capacity and flesh out the opaque structure type that represents a stack. From this we can see that we maintain the current **`capacity`** of the stack, the **`delta`** value by which to increment the capacity if we run out of room, and the **next** index into the stack to be used for a **push**. note that the top of stack is at index **next-1**; if **next** is 0, the stack is empty. Finally, we have an array of **void *** pointers for the elements on the stack.

## stack_create()

```
Stack *stack_create(long capacity) {
   Stack *st = (Stack *)malloc(sizeof(Stack));

   if (st != NULL) {
      long cap;
      void **array = NULL;

      cap = (capacity <= 0) ? DEFAULT_CAPACITY : capacity;
      cap = (cap > MAX_INIT_CAPACITY) ? MAX_INIT_CAPACITY : cap;
      array = (void **) malloc(cap * sizeof(void *));
      if (array == NULL) {
         free(st);
         st = NULL;
      } else {
         st->capacity = cap;
         st->delta = cap;
         st->next = 0L;
         st->theArray = array;
      }
   }
   return st;
}
```

First, we **malloc** a struct to represent the new stack. If that was successful, compute the initial capacity, and allocate an array of **void *** pointers of that size. If that was successful, fill in the various fields of the struct. Finally return the struct.

## stack_destroy() and stack_purge()

Both of these require that we traverse the current elements on the stack, invoking the user-specified function to free any memory associated with each element. Thus, we define a static function in the implementation to do this common processing, leaving any different processing to the two ADT methods.

```
/*
 * traverses stack, calling freeFxn on each element
 */
static void purge(Stack *st, void (*freeFxn)(void*)) {

   if (freeFxn != NULL) {
      long i;

      for (i = 0L; i < st->next; i++)
         (*freeFxn)(st->theArray[i]); /* user frees elem storage */
   }
}

void stack_destroy(Stack *st, void (*freeFxn)(void*)) {
   purge(st, freeFxn);
   free(st->theArray);              /* free array of pointers */
   free(st);                        /* free the Stack struct */
}

void stack_purge(Stack *st, void (*freeFxn)(void*)){
   purge(st, freeFxn);
   st->next = 0L;
}
```

## stack_push()

```
int stack_push(Stack *st, void *element) {
   int status = 1;
```

```
    if (st->capacity <= st->next) {      /* need to reallocate */
       size_t nbytes = (st->capacity + st->delta) * sizeof(void *);
       void **tmp = (void **)realloc(st->theArray, nbytes);
       if (tmp == NULL)
          status = 0;       /* allocation failure */
       else {
           st->theArray = tmp;
          st->capacity += st->delta;
       }
    }
    if (status)
       st->theArray[st->next++] = element;
    return status;
 }
```

The “complicated” code here is simply to detect when the stack needs to be resized. If the capacity has been exhausted, then **realloc** is invoked; if it is unsuccessful, then we will return a failure status, since the stack is full. If it is successful, appropriate fields are modified, and we then place **element** into the next location in the array and increment the **next** field.

## stack_pop() and stack_peek()

The logic here is nearly identical, except for the side effect of decrementing the **next** field in **pop**. The code is so simple that there is insufficient scope for placing the common logic in a static function, so the common code is duplicated in the two methods.

```
 int stack_pop(Stack *st, void **element) {
    int status = 0;

    if (st->next > 0L) {
       *element = st->theArray[--st->next];
       status = 1;
    }
    return status;
 }

 int stack_peek(Stack *st, void **element) {
    int status = 0;

    if (st->next > 0L) {
       *element = st->theArray[st->next - 1];
       status = 1;
    }
    return status;
 }
```

## stack_isEmpty() and stack_size()

No further discussion is needed. Here is the code.

```
 int stack_isEmpty(Stack *st) {
    return (st->next == 0L);
 }

 long stack_size(Stack *st) {
    return st->next;
 }
```

## stack_toArray() and stack_it_create()

As can be seen in Appendix A, the method that creates a generic iterator in the **Iterator** ADT requires an array of **void *** pointers and a length. Thus, **toArray** and **it_create** both require an array of **void *** pointers to the elements. A static function, **arraydupl**, is defined, and then used by the two public methods in the ADT.

```
 /*
  * local function - duplicates array of void * pointers on the heap
 *
 * returns pointer to duplicate array or NULL if malloc failure
 */
static void **arraydupl(Stack *st) {
   void **tmp = NULL;
   if (st->next > 0L) {
      size_t nbytes = st->next * sizeof(void *);
      tmp = (void **)malloc(nbytes);
      if (tmp != NULL) {
         long i;

          for (i = 0; i < st->next; i++)
            tmp[i] = st->theArray[i];
      }
   }
   return tmp;
}

void **stack_toArray(Stack *st, long *len) {
   void **tmp = arraydupl(st);

   if (tmp != NULL)
        *len = st->next;
   return tmp;
}

Iterator *stack_it_create(Stack *st) {
   Iterator *it = NULL;
   void **tmp = arraydupl(st);

   if (tmp != NULL) {
      it = it_create(st->next, tmp);
      if (it == NULL)
         free(tmp);
   }
   return it;
}
```

Note that both **`toArray`** and **`it_create`** return **`NULL`** if the stack is empty.

## A test program for the non-thread-safe ADT

The following program reads lines of text from a file specified in **`argv[1]`**, and then exercises most of the methods in the interface.

```
#include "stack.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
   char buf[1024];
   char *p;
   Stack *st;
   long i, n;
   FILE *fd;
   char **array;
   Iterator *it;

   if (argc != 2) {
      fprintf(stderr, "usage: ./sttest file\n");
   return -1;
   }
   if ((st = stack_create(0L)) == NULL) {
      fprintf(stderr, "Error creating stack of strings\n");
   return -1;
   }
   if ((fd = fopen(argv[1], "r")) == NULL) {
      fprintf(stderr, "Unable to open %s to read\n", argv[1]);
```

```
return -1;
}
/*
 * test of push()
 */
printf("===== test of push\n");
while (fgets(buf, 1024, fd) != NULL) {
   if ((p = strdup(buf)) == NULL) {
      fprintf(stderr, "Error duplicating string\n");
      return -1;
   }
   if (!stack_push(st, p)) {
      fprintf(stderr, "Error pushing string to stack\n");
      return -1;
   }
}
fclose(fd);
n = stack_size(st);
/*
 * test of pop()
 */
printf("===== test of pop\n");
for (i = 0; i < n; i++) {
   if (!stack_pop(st, (void **)&p)) {
      fprintf(stderr, "Error retrieving %ld'th element\n", i);
      return -1;
   }
   printf("%s", p);
   free(p);
}
printf("===== test of destroy(NULL)\n");
/*
 * test of destroy with NULL freeFxn
 */
stack_destroy(st, NULL);
if ((st = stack_create(0L)) == NULL) {
   fprintf(stderr, "Error creating stack of strings\n");
   return -1;
}
fd = fopen(argv[1], "r");            /* we know we can open it */
while (fgets(buf, 1024, fd) != NULL) {
   if ((p = strdup(buf)) == NULL) {
      fprintf(stderr, "Error duplicating string\n");
      return -1;
   }
   if (!stack_push(st, p)) {
      fprintf(stderr, "Error pushing string to stack\n");
      return -1;
   }
}
fclose(fd);
printf("===== test of toArray\n");
/*
 * test of toArray
 */
if ((array = (char **)stack_toArray(st, &n)) == NULL) {
   fprintf(stderr, "Error in invoking stack_toArray()\n");
   return -1;
}
for (i = 0; i < n; i++) {
   printf("%s", array[i]);
}
free(array);
printf("===== test of iterator\n");
/*
 * test of iterator
 */
if ((it = stack_it_create(st)) == NULL) {
      fprintf(stderr, "Error in creating iterator\n");
      return -1;
   }
   while (it_hasNext(it)) {
      char *p;
      (void) it_next(it, (void **)&p);
      printf("%s", p);
   }
   it_destroy(it);
   printf("===== test of destroy(free)\n");
   /*
    * test of destroy with free() as freeFxn
    */
   stack_destroy(st, free);

   return 0;
}
```

## Thread-safe interface

The non-thread-safe ADT gives us all of the functionality that we require. All we have to do now is create appropriate critical sections around calls to the non-thread-safe methods to guarantee the lack of race conditions.

Rather than create another complete implementation, but this time with appropriate pthread logic to create the critical sections, it is far easier and less error prone to simple encapsulate an instance of a non-thread-safe ADT inside of a thread-safe instance. Each method will then act like a Java synchronized method.

Note that one often wishes to create a transaction across two or more separate calls – e.g. insert a new entry into a collection ONLY if it is ***not*** already there. This has been tackled by enabling a client thread to obtain the lock associated with the thread-safe ADT; once possessing this lock, the client may invoke as many of the other methods as it wishes before releasing the lock. Thus, each of our "synchronized" methods must also be invokable in the scope of one of these larger transactions.

There is a general problem with using iterators if the structure can be change while you are traversing it. When you are given a thread-safe iterator by the factory method in one of these ADTs, you also possess the lock on the structure, and will retain that lock until you invoke **`tsit_destroy()`** on that iterator.

Here is the interface specification for the thread-safe Stack ADT.

```
#include "tsiterator.h"

/*
 * interface definition for generic type-safe stack implementation
 *
 * patterned roughly after Java 6 Stack generic class
 */

typedef struct tsstack TSStack;  /* opaque type definition */

/*
 * create a stack with the specified capacity; if capacity == 0, a
 * default initial capacity (50 elements) is used
 *
 * returns a pointer to the stack, or NULL if malloc() errors
 */
TSStack *tsstack_create(long capacity);

/*
 * destroys the stack; for each occupied position,
 * if freeFxn != NULL, it is invoked on the element at that position;
 * the storage associated with the stack is then returned to the heap
 */
void tsstack_destroy(TSStack *st, void (*freeFxn)(void *element));
```

```
/*
 * purges all elements from the stack; for each occupied position,
 * if freeFxn != NULL, it is invoked on the element at that position;
 * any storage associated with the element in the stack is then
 * returned to the heap
 *
 * upon return, the stack will be empty
 */
void tsstack_purge(TSStack *st, void (*freeFxn)(void *element));

/*
 * obtains the lock for exclusive access
 */
void tsstack_lock(TSStack *st);

/*
 * releases the lock
 */
void tsstack_unlock(TSStack *st);

/*
 * pushes `element' onto the stack; if no more room, the stack is
 * dynamically resized
 *
 * returns 1 if successful, 0 if unsuccessful (malloc errors)
 */
int tsstack_push(TSStack *st, void *element);

/*
 * pops the element at the top of the stack into `*element'
 *
 * returns 1 if successful, 0 if stack was empty
 */
int tsstack_pop(TSStack *st, void **element);

/*
 * peeks at the top element of the stack without removing it;
 * returned in `*element'
 *
 * returns 1 if successful, 0 i stack was empty
 */
int tsstack_peek(TSStack *st, void **element);

/*
 * returns 1 if stack is empty, 0 if it is not
 */
int tsstack_isEmpty(TSStack *st);

/*
 * returns the number of elements in the stack
 */
long tsstack_size(TSStack *st);

/*
 * returns an array containing all of the elements of the stack in
 * proper sequence (top to bottom element); returns the length of
 * the list in `len'
 *
 * returns pointer to void * element array, or NULL if malloc failure
 */
void **tsstack_toArray(TSStack *st, long *len);

/*
 * create generic iterator to this stack;
 * successive next calls return elements in sequence (top to bottom)
 *
 * returns pointer to the TSIterator or NULL if failure
 */
TSIterator *tsstack_it_create(TSStack *st);
```

## Thread-safe implementation

In order to enable the transaction capability over two or more method calls, we have used RECURSIVE pthread mutexes. This is why the code in tsstack_create is dominated by pthread calls. Other than that, the implementation is quite straight-forward.

```
#include "tsstack.h"
#include "stack.h"
#include <stdlib.h>
#include <pthread.h>

#define LOCK(st) &((st)->lock)

/*
 * implementation for thread-safe generic stack implementation
 */

struct tsstack {
   Stack *st;
   pthread_mutex_t lock;   /* this is a recursive lock */
};

TSStack *tsstack_create(long capacity) {
   TSStack *tsst = (TSStack *)malloc(sizeof(TSStack));

   if (tsst != NULL) {
      Stack *st = stack_create(capacity);

      if (st == NULL) {
         free(tsst);
          tsst = NULL;
      } else {
         pthread_mutexattr_t ma;
          pthread_mutexattr_init(&ma);
         pthread_mutexattr_settype(&ma, PTHREAD_MUTEX_RECURSIVE);
         tsst->st = st;
         pthread_mutex_init(LOCK(tsst), &ma);
         pthread_mutexattr_destroy(&ma);
      }
   }
   return tsst;
}

void tsstack_destroy(TSStack *st, void (*freeFxn)(void*)) {
   pthread_mutex_lock(LOCK(st));
   stack_destroy(st->st, freeFxn);
   pthread_mutex_unlock(LOCK(st));
   pthread_mutex_destroy(LOCK(st));
   free(st);
}

void tsstack_purge(TSStack *st, void (*freeFxn)(void*)) {
   pthread_mutex_lock(LOCK(st));
   stack_purge(st->st, freeFxn);
   pthread_mutex_unlock(LOCK(st));
}

void tsstack_lock(TSStack *st) {
   pthread_mutex_lock(LOCK(st));
}

void tsstack_unlock(TSStack *st) {
   pthread_mutex_unlock(LOCK(st));
}
```

```
int tsstack_push(TSStack *st, void *element) {
   int result;
   pthread_mutex_lock(LOCK(st));
   result = stack_push(st->st, element);
   pthread_mutex_unlock(LOCK(st));
   return result;
}

int tsstack_pop(TSStack *st, void **element) {
   int result;
   pthread_mutex_lock(LOCK(st));
   result = stack_pop(st->st, element);
   pthread_mutex_unlock(LOCK(st));
   return result;
}

int tsstack_peek(TSStack *st, void **element) {
   int result;
   pthread_mutex_lock(LOCK(st));
   result = stack_peek(st->st, element);
   pthread_mutex_unlock(LOCK(st));
   return result;
}

int tsstack_isEmpty(TSStack *st) {
   int result;
   pthread_mutex_lock(LOCK(st));
   result = stack_isEmpty(st->st);
   pthread_mutex_unlock(LOCK(st));
   return result;
}

long tsstack_size(TSStack *st) {
   long result;
   pthread_mutex_lock(LOCK(st));
   result = stack_size(st->st);
   pthread_mutex_unlock(LOCK(st));
   return result;
}

void **tsstack_toArray(TSStack *st, long *len) {
   void **result;
   pthread_mutex_lock(LOCK(st));
   result = stack_toArray(st->st, len);
   pthread_mutex_unlock(LOCK(st));
   return result;
}

TSIterator *tsstack_it_create(TSStack *st) {
   TSIterator *it = NULL;
   void **tmp;
   long len;

   pthread_mutex_lock(LOCK(st));
   tmp = stack_toArray(st->st, &len);
   if (tmp != NULL) {
      it = tsit_create(LOCK(st), len, tmp);
      if (it == NULL)
         free(tmp);
   }
   if (it == NULL)
      pthread_mutex_unlock(LOCK(st));
   return it;
}
```

## Thread-safe test program

This is a straightforward conversion of the non-thread-safe test program with the following edits:

- include “tsstack.h” instead of “stack.h”
- replace “Stack” declaration with “TSStack”
- replace “Iterator” declaration with “TSIterator”
- replace “stack_create” calls by “tsstack_create”
- replace “stack_push” calls by “tsstack_push”
- replace “stack_size” call by “tsstack_size”
- replace “stack_pop” call by “tsstack_pop”
- replace “stack_destroy” calls by “tsstack_destroy”
- replace “stack_toArray” call by “tsstack_toArray”
- replace “stack_it_create” call by “tsstack_it_create”
- replace “it_hasNext” call by “tsit_hasNet”
- replace “it_next” call by “tsit_next”
- replace “it_destroy” call by “tsit_destroy”

## The ADTs in the library

The ADTs provided in the library are the following (the letter for each enumerated item corresponds to the appendix in which the interface and implementation for that ADT are found):

A. Iterator and TSIterator – these are generic iterators that can be used to traverse a library ADT.
B. ArrayList and TSArrayList –provide most of the functionality found in the Java ArrayList generic class; natural order for iteration is the index order of the array list.
C. LinkedList and TSLinkedList – provide most of the functionality found in the Java LinkedList generic class; implemented using a doubly-linked list; natural order for iteration is head to tail order in the linked list.
D. HashMap and TSHashMap – provide most of the functionality found in the Java HashMap generic class ***EXCEPT*** that keys are restricted to strings; “natural” order for iteration is by hash bucket, and LIFO in each bucket.
E. Treeset and TSTreeset – provide most of the functionality found in the Java TreeSet generic class; implemented using an AVL tree; natural order for iteration is sort order for the set.
F. Stack and TSStack – while a stack can be implemented using a LinkedList or an ArrayList, the example shown in the previous section is also included as an ADT in the library; natural order for iteration is top to bottom in the stack.

Two test programs, one for non-thread-safe and the other for thread-safe versions of the ADT, are included in appendices B-F, as well. All of the source files can be obtained over the Internet at <where>.

## Appendix A – Iterator and TSIterator

### iterator.h

```
#ifndef _ITERATOR_H_
#define _ITERATOR_H_

/* BSD header removed to save space */

/*
 * interface definition for generic iterator
 *
 * patterned roughly after Java 6 Iterator class
 */

typedef struct iterator Iterator;

/*
 * creates an iterator from the supplied arguments; it is for use by the
 * iterator create methods in ADTs
 *
 * iterator assumes responsibility for elements[] if create is succesful
 * i.e. it_destroy will free the array of pointers
 *
 * returns pointer to iterator if successful, NULL otherwise
 */
Iterator *it_create(long size, void **elements);

/*
 * returns 1/0 if the iterator has a next element
 */
int it_hasNext(Iterator *it);

/*
 * returns the next element from the iterator in `*element'
 *
 * returns 1 if successful, 0 if unsuccessful (no next element)
 */
int it_next(Iterator *it, void **element);

/*
 * destroys the iterator
 */
void it_destroy(Iterator *it);

#endif /* _ITERATOR_H_ */
```

### tsiterator.h

```
#ifndef _TSITERATOR_H_
#define _TSITERATOR_H_

/* BSD header removed to save space */

/*
 * interface definition for thread safe generic iterator
 */

#include <pthread.h>

typedef struct tsiterator TSIterator;

/*
 * creates a thread-safe iterator from the supplied arguments; it is for use
 * by the iterator create methods in thread-safe ADTs
 *
 * at the time tsit_create is called, the calling ADT must already hold the
 * lock associated with the ADT instance to guarantee that the array reflects
 * the contents of the ADT instance; this lock is held until the application
 * destroys the thread-safe iterator, at which point the lock is released
 *
 * the iterator assumes responsibility for elements[] if create is successful
 * i.e. tsit_destroy will free the array of pointers
 *
 * returns pointer to iterator if successful, NULL otherwise
 */
TSIterator *tsit_create(pthread_mutex_t *lock, long size, void **elements);

/*
 * returns 1/0 if the iterator has a next element
 */
int tsit_hasNext(TSIterator *it);

/*
 * returns the next element from the iterator in `*element'
 *
 * returns 1 if successful, 0 if unsuccessful (no next element)
 */
int tsit_next(TSIterator *it, void **element);

/*
 * destroys the iterator
 */
void tsit_destroy(TSIterator *it);

#endif /* _TSITERATOR_H_ */
```

### iterator.c

```
/* BSD header removed to save space */

#include "iterator.h"
#include "stdlib.h"

/*
 * implementation for generic iterator
 *
 * patterned roughly after Java 6 Iterator class
 */

struct iterator {
   long next;
   long size;
   void **elements;
};

Iterator *it_create(long size, void **elements) {
   Iterator *it = (Iterator *)malloc(sizeof(Iterator));

   if (it != NULL) {
      it->next = 0L;
      it->size = size;
      it->elements = elements;
   }
   return it;
}

int it_hasNext(Iterator *it) {
   return (it->next < it->size) ? 1 : 0;
}

int it_next(Iterator *it, void **element) {
   int status = 0;
   if (it->next < it->size) {
      *element = it->elements[it->next++];
      status = 1;
   }
   return status;
}

void it_destroy(Iterator *it) {
   free(it->elements);
   free(it);
}
```

## tsiterator.c

```
/* BSD header removed to save space */

#include "tsiterator.h"
#include <stdlib.h>
#include <pthread.h>

/*
 * implementation for thread-safe generic iterator
 */

struct tsiterator {
   long next;
   long size;
   void **elements;
   pthread_mutex_t *lock;
};

TSIterator *tsit_create(pthread_mutex_t *lock, long size, void **elements) {
   TSIterator *it = (TSIterator *)malloc(sizeof(TSIterator));

   if (it != NULL) {
      it->next = 0L;
      it->size = size;
      it->elements = elements;
      it->lock = lock;
   }
   return it;
}

int tsit_hasNext(TSIterator *it) {
   return (it->next < it->size) ? 1 : 0;
}

int tsit_next(TSIterator *it, void **element) {
   int status = 0;
   if (it->next < it->size) {
      *element = it->elements[it->next++];
      status = 1;
   }
   return status;
}

void tsit_destroy(TSIterator *it) {
   free(it->elements);
   pthread_mutex_unlock(it->lock);
   free(it);
}
```

# Appendix B – ArrayList and TSArrayList

## arraylist.h

```
#ifndef _ARRAYLIST_H_
#define _ARRAYLIST_H_

/* BSD header removed to save space */

#include "iterator.h"

/*
 * interface definition for generic arraylist implementation
 *
 * patterned roughly after Java 6 ArrayList generic class
 */

typedef struct arraylist ArrayList;     /* opaque type definition */

/*
 * create an arraylist with the specified capacity; if capacity == 0, a
 * default initial capacity (10 elements) is used
 *
 * returns a pointer to the array list, or NULL if there are malloc() errors
 */
ArrayList *al_create(long capacity);

/*
 * destroys the arraylist; for each occupied index, if userFunction != NULL,
 * it is invoked on the element at that position; the storage associated with
 * the arraylist is then returned to the heap
 */
void al_destroy(ArrayList *al, void (*userFunction)(void *element));

/*
 * appends `element' to the arraylist; if no more room in the list, it is
 * dynamically resized
 *
 * returns 1 if successful, 0 if unsuccessful (malloc errors)
 */
int al_add(ArrayList *al, void *element);

/*
 * clears all elements from the arraylist; for each occupied index,
 * if userFunction != NULL, it is invoked on the element at that position;
 * any storage associated with the element in the arraylist is then
 * returned to the heap
 *
 * upon return, the arraylist will be empty
 */
void al_clear(ArrayList *al, void (*userFunction)(void *element));

/*
 * ensures that the arraylist can hold at least `minCapacity' elements
 *
 * returns 1 if successful, 0 if unsuccessful (malloc failure)
 */
int al_ensureCapacity(ArrayList *al, long minCapacity);

/*
 * returns the element at the specified position in this list in `*element'
 *
 * returns 1 if successful, 0 if no element at that position
 */
int al_get(ArrayList *al, long i, void **element);

/*
```

```
 * inserts `element' at the specified position in the arraylist;
 * all elements from `i' onwards are shifted one position to the right;
 * if no more room in the list, it is dynamically resized;
 * if the current size of the list is N, legal values of i are in the
 * interval [0, N]
 *
 * returns 1 if successful, 0 if unsuccessful (malloc errors)
 */
int al_insert(ArrayList *al, long i, void *element);

/*
 * returns 1 if arraylist is empty, 0 if it is not
 */
int al_isEmpty(ArrayList *al);

/*
 * removes the `i'th element from the list, returns the value that
 * occupied that position in `*element'; all elements from [i+1, size-1] are
 * shifted down one position
 *
 * returns 1 if successful, 0 if `i'th position was not occupied
 */
int al_remove(ArrayList *al, long i, void **element);

/*
 * relaces the `i'th element of the arraylist with `element';
 * returns the value that previously occupied that position in `previous'
 *
 * returns 1 if successful
 * returns 0 if `i'th position not currently occupied
 */
int al_set(ArrayList *al, void *element, long i, void **previous);

/*
 * returns the number of elements in the arraylist
 */
long al_size(ArrayList *al);

/*
 * returns an array containing all of the elements of the list in
 * proper sequence (from first to last element); returns the length of
 * the list in `len'
 *
 * returns pointer to void * array of elements, or NULL if malloc failure
 */
void **al_toArray(ArrayList *al, long *len);

/*
 * trims the capacity of the arraylist to be the list's current size
 *
 * returns 1 if successful, 0 if failure (malloc errors)
 */
int al_trimToSize(ArrayList *al);

/*
 * create generic iterator to this arraylist
 *
 * returns pointer to the Iterator or NULL if failure
 */
Iterator *al_it_create(ArrayList *al);

#endif /* _ARRAYLIST_H_ */
```

### tsarraylist.h

```
#ifndef _TSARRAYLIST_H_
#define _TSARRAYLIST_H_

/* BSD header removed to save space */
```

```
#include "tsiterator.h"

/*
 * interface definition for thread-safe generic arraylist implementation
 *
 * patterned roughly after Java 6 ArrayList generic class
 */

typedef struct tsarraylist TSArrayList; /* opaque type definition */

/*
 * create an arraylist with the specified capacity; if capacity == 0, a
 * default initial capacity (10 elements) is used
 *
 * returns a pointer to the array list, or NULL if there are malloc() errors
 */
TSArrayList *tsal_create(long capacity);

/*
 * destroys the arraylist; for each occupied index, if userFunction != NULL,
 * it is invoked on the element at that position; the storage associated with
 * the arraylist is then returned to the heap
 */
void tsal_destroy(TSArrayList *al, void (*userFunction)(void *element));

/*
 * obtains the lock for exclusive access
 */
void tsal_lock(TSArrayList *al);

/*
 * returns the lock
 */
void tsal_unlock(TSArrayList *al);

/*
 * appends `element' to the arraylist; if no more room in the list, it is
 * dynamically resized
 *
 * returns 1 if successful, 0 if unsuccessful (malloc errors)
 */
int tsal_add(TSArrayList *al, void *element);

/*
 * clears all elements from the arraylist; for each occupied index,
 * if userFunction != NULL, it is invoked on the element at that position;
 * any storage associated with the element in the arraylist is then
 * returned to the heap
 *
 * upon return, the arraylist will be empty
 */
void tsal_clear(TSArrayList *al, void (*userFunction)(void *element));

/*
 * ensures that the arraylist can hold at least `minCapacity' elements
 *
 * returns 1 if successful, 0 if unsuccessful (malloc failure)
 */
int tsal_ensureCapacity(TSArrayList *al, long minCapacity);

/*
 * returns the element at the specified position in this list in `*element'
 *
 * returns 1 if successful, 0 if no element at that position
 */
int tsal_get(TSArrayList *al, long i, void **element);
```

```
/*
 * inserts `element' at the specified position in the arraylist;
 * all elements from `i' onwards are shifted one position to the right;
 * if no more room in the list, it is dynamically resized;
 * if the current size of the list is N;
 * legal values of i are in the interval [0, N]
 *
 * returns 1 if successful, 0 if unsuccessful (malloc errors)
 */
int tsal_insert(TSArrayList *al, long i, void *element);

/*
 * returns 1 if list is empty, 0 if it is not
 */
int tsal_isEmpty(TSArrayList *al);

/*
 * removes the `i'th element from the list, returns the value that
 * occupied that position in `*element'
 *
 * returns 1 if successful, 0 if `i'th position was not occupied
 */
int tsal_remove(TSArrayList *al, long i, void **element);

/*
 * relaces the `i'th element of the arraylist with `element';
 * returns the value that previously occupied that position in `previous'
 *
 * returns 1 if successful
 * returns 0 if `i'th position not currently occupied
 */
int tsal_set(TSArrayList *al, void *element, long i, void **previous);

/*
 * returns the number of elements in the arraylist
 */
long tsal_size(TSArrayList *al);

/*
 * returns an array containing all of the elements of the list in
 * proper sequence (from first to last element); returns the length of
 * the list in `len'
 *
 * returns pointer to void * array of elements, or NULL if malloc failure
 */
void **tsal_toArray(TSArrayList *al, long *len);

/*
 * trims the capacity of the arraylist to be the list's current size
 *
 * returns 1 if successful, 0 if failure (malloc errors)
 */
int tsal_trimToSize(TSArrayList *al);

/*
 * create generic iterator to this arraylist
 *
 * returns pointer to the Iterator or NULL if failure
 */
TSIterator *tsal_it_create(TSArrayList *al);

#endif /* _TSARRAYLIST_H_ */
```

## arraylist.c

```
/* BSD header removed to save space */

/*
 * implementation for generic array list
 */

#include "arraylist.h"
#include <stdlib.h>

#define DEFAULT_CAPACITY 10L
#define MAX_INIT_CAPACITY 100000L

struct arraylist {
   long capacity;
   long delta;
   long size;
   void **theArray;
};

ArrayList *al_create(long capacity) {
   ArrayList *al = (ArrayList *)malloc(sizeof(ArrayList));

   if (al != NULL) {
      long cap;
      void **array = NULL;

      cap = (capacity <= 0) ? DEFAULT_CAPACITY : capacity;
      cap = (cap > MAX_INIT_CAPACITY) ? MAX_INIT_CAPACITY : cap;
      array = (void **) malloc(cap * sizeof(void *));
      if (array == NULL) {
         free(al);
        al = NULL;
      } else {
         al->capacity = cap;
        al->delta = cap;
        al->size = 0L;
        al->theArray = array;
      }
   }
   return al;
}

/*
 * traverses arraylist, calling userFunction on each element
 */
static void purge(ArrayList *al, void (*userFunction)(void *element)) {

   if (userFunction != NULL) {
      long i;

      for (i = 0L; i < al->size; i++)
         (*userFunction)(al->theArray[i]); /* user frees element storage */
   }
}

void al_destroy(ArrayList *al, void (*userFunction)(void *element)) {
   purge(al, userFunction);
   free(al->theArray);                    /* we free array of pointers */
   free(al);                       /* we free the ArrayList struct */
}

int al_add(ArrayList *al, void *element) {
   int status = 1;

   if (al->capacity <= al->size) {      /* need to reallocate */
      size_t nbytes = (al->capacity + al->delta) * sizeof(void *);
      void **tmp = (void **)realloc(al->theArray, nbytes);
      if (tmp == NULL)
         status = 0;      /* allocation failure */
      else {
         al->theArray = tmp;
        al->capacity += al->delta;
      }
   }
   if (status)
      al->theArray[al->size++] = element;
   return status;
}

void al_clear(ArrayList *al, void (*userFunction)(void *element)){
   purge(al, userFunction);
   al->size = 0L;
}

int al_ensureCapacity(ArrayList *al, long minCapacity) {
   int status = 1;

   if (al->capacity < minCapacity) {    /* must extend */
      void **tmp = (void **)realloc(al->theArray, minCapacity * sizeof(void *));
      if (tmp == NULL)
         status = 0;       /* allocation failure */
      else {
         al->theArray = tmp;
        al->capacity = minCapacity;
      }
   }
   return status;
}

int al_get(ArrayList *al, long i, void **element) {
   int status = 0;

   if (i >= 0L && i < al->size) {
      *element = al->theArray[i];
      status = 1;
   }
   return status;
}

int al_insert(ArrayList *al, long i, void *element) {
   int status = 1;

   if (i > al->size)
      return 0;                         /* 0 <= i <= size */
   if (al->capacity <= al->size) {      /* need to reallocate */
      size_t nbytes = (al->capacity + al->delta) * sizeof(void *);
      void **tmp = (void **)realloc(al->theArray, nbytes);
      if (tmp == NULL)
         status = 0;       /* allocation failure */
      else {
         al->theArray = tmp;
        al->capacity += al->delta;
      }
   }
   if (status) {
      long j;
      for (j = al->size; j > i; j--)           /* slide items up */
         al->theArray[j] = al->theArray[j-1];
      al->theArray[i] = element;
      al->size++;
   }
   return status;
}

int al_isEmpty(ArrayList *al) {
   return (al->size == 0L);
}

int al_remove(ArrayList *al, long i, void **element) {
   int status = 0;
   long j;

   if (i >= 0L && i < al->size) {
      *element = al->theArray[i];
      for (j = i + 1; j < al->size; j++)
         al->theArray[i++] = al->theArray[j];
      al->size--;
      status = 1;
   }
   return status;
}

int al_set(ArrayList *al, void *element, long i, void **previous) {
   int status = 0;

   if (i >= 0L && i < al->size) {
      *previous = al->theArray[i];
      al->theArray[i] = element;
      status = 1;
   }
   return status;
}

long al_size(ArrayList *al) {
   return al->size;
}

/*
 * local function that duplicates the array of void * pointers on the heap
 *
 * returns pointer to duplicate array or NULL if malloc failure
 */
static void **arraydupl(ArrayList *al) {
   void **tmp = NULL;
   if (al->size > 0L) {
      size_t nbytes = al->size * sizeof(void *);
      tmp = (void **)malloc(nbytes);
      if (tmp != NULL) {
         long i;

        for (i = 0; i < al->size; i++)
            tmp[i] = al->theArray[i];
      }
   }
   return tmp;
}

void **al_toArray(ArrayList *al, long *len) {
   void **tmp = arraydupl(al);

   if (tmp != NULL)
       *len = al->size;
   return tmp;
}

int al_trimToSize(ArrayList *al) {
   int status = 0;

   void **tmp = (void **)realloc(al->theArray, al->size * sizeof(void *));
   if (tmp != NULL) {
      status = 1;
      al->theArray = tmp;
      al->capacity = al->size;
   }
   return status;
}

Iterator *al_it_create(ArrayList *al) {
   Iterator *it = NULL;
   void **tmp = arraydupl(al);

   if (tmp != NULL) {
      it = it_create(al->size, tmp);
      if (it == NULL)
         free(tmp);
   }
   return it;
}
```

## tsarraylist.c

```
/* BSD header removed to save space */

#include "tsarraylist.h"
#include "arraylist.h"
#include <stdlib.h>
#include <pthread.h>

#define LOCK(al) &((al)->lock)

/*
 * implementation for thread-safe generic arraylist implementation
 */

struct tsarraylist {
   ArrayList *al;
   pthread_mutex_t lock;   /* this is a recursive lock */
};

TSArrayList *tsal_create(long capacity) {
   TSArrayList *tsal = (TSArrayList *)malloc(sizeof(TSArrayList));

   if (tsal != NULL) {
      ArrayList *al = al_create(capacity);

      if (al == NULL) {
         free(tsal);
        tsal = NULL;
      } else {
         pthread_mutexattr_t ma;
        pthread_mutexattr_init(&ma);
         pthread_mutexattr_settype(&ma, PTHREAD_MUTEX_RECURSIVE);
         tsal->al = al;
         pthread_mutex_init(LOCK(tsal), &ma);
         pthread_mutexattr_destroy(&ma);
      }
   }
   return tsal;
}

void tsal_destroy(TSArrayList *al, void (*userFunction)(void *element)) {
   pthread_mutex_lock(LOCK(al));
   al_destroy(al->al, userFunction);
   pthread_mutex_unlock(LOCK(al));
   pthread_mutex_destroy(LOCK(al));
   free(al);
}

void tsal_lock(TSArrayList *al) {
   pthread_mutex_lock(LOCK(al));
}

void tsal_unlock(TSArrayList *al) {
   pthread_mutex_unlock(LOCK(al));
}

int tsal_add(TSArrayList *al, void *element) {
   int result;
   pthread_mutex_lock(LOCK(al));
   result = al_add(al->al, element);
   pthread_mutex_unlock(LOCK(al));
   return result;
}

void tsal_clear(TSArrayList *al, void (*userFunction)(void *element)) {
   pthread_mutex_lock(LOCK(al));
   al_clear(al->al, userFunction);
   pthread_mutex_unlock(LOCK(al));
}

int tsal_ensureCapacity(TSArrayList *al, long minCapacity) {
   int result;
   pthread_mutex_lock(LOCK(al));
   result = al_ensureCapacity(al->al, minCapacity);
   pthread_mutex_unlock(LOCK(al));
   return result;
}

int tsal_get(TSArrayList *al, long i, void **element) {
   int result;
   pthread_mutex_lock(LOCK(al));
   result = al_get(al->al, i, element);
   pthread_mutex_unlock(LOCK(al));
   return result;
}

int tsal_insert(TSArrayList *al, long i, void *element) {
   int result;
   pthread_mutex_lock(LOCK(al));
   result = al_insert(al->al, i, element);
   pthread_mutex_unlock(LOCK(al));
   return result;
}

int tsal_isEmpty(TSArrayList *al) {
   int result;
   pthread_mutex_lock(LOCK(al));
   result = al_isEmpty(al->al);
   pthread_mutex_unlock(LOCK(al));
   return result;
}

int tsal_remove(TSArrayList *al, long i, void **element) {
   int result;
   pthread_mutex_lock(LOCK(al));
   result = al_remove(al->al, i, element);
   pthread_mutex_unlock(LOCK(al));
   return result;
}

int tsal_set(TSArrayList *al, void *element, long i, void **previous) {
   int result;
   pthread_mutex_lock(LOCK(al));
   result = al_set(al->al, element, i, previous);
   pthread_mutex_unlock(LOCK(al));
   return result;
}

long tsal_size(TSArrayList *al) {
   long result;
   pthread_mutex_lock(LOCK(al));
   result = al_size(al->al);
   pthread_mutex_unlock(LOCK(al));
   return result;
}
```

```
void **tsal_toArray(TSArrayList *al, long *len) {
   void **result;
   pthread_mutex_lock(LOCK(al));
   result = al_toArray(al->al, len);
   pthread_mutex_unlock(LOCK(al));
   return result;
}

int tsal_trimToSize(TSArrayList *al) {
   int result;
   pthread_mutex_lock(LOCK(al));
   result = al_trimToSize(al->al);
   pthread_mutex_unlock(LOCK(al));
   return result;
}

TSIterator *tsal_it_create(TSArrayList *al) {
   TSIterator *it = NULL;
   void **tmp;
   long len;

   pthread_mutex_lock(LOCK(al));
   tmp = al_toArray(al->al, &len);
   if (tmp != NULL) {
      it = tsit_create(LOCK(al), len, tmp);
      if (it == NULL)
         free(tmp);
   }
   if (it == NULL)
      pthread_mutex_unlock(LOCK(al));
   return it;
}
```

## altest.c (you can create your own tsaltest.c)

```
/* BSD header removed to save space */

#include "arraylist.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
   char buf[1024];
   char *p;
   ArrayList *al;
   long i, n;
   FILE *fd;
   char **array;
   Iterator *it;

   if (argc != 2) {
      fprintf(stderr, "usage: ./altest file\n");
      return -1;
   }
   if ((al = al_create(0L)) == NULL) {
      fprintf(stderr, "Error creating array list of strings\n");
      return -1;
   }
   if ((fd = fopen(argv[1], "r")) == NULL) {
      fprintf(stderr, "Unable to open %s to read\n", argv[1]);
      return -1;
   }
   /*
    * test of add()
    */
   printf("===== test of add\n");
   while (fgets(buf, 1024, fd) != NULL) {
      if ((p = strdup(buf)) == NULL) {
         fprintf(stderr, "Error duplicating string\n");
         return -1;
      }
      if (!al_add(al, p)) {
         fprintf(stderr, "Error adding string to array list\n");
         return -1;
      }
   }
   fclose(fd);
   n = al_size(al);
   /*
    * test of get()
    */
   printf("===== test of get\n");
   for (i = 0; i < n; i++) {
      if (!al_get(al, i, (void **)&p)) {
         fprintf(stderr, "Error retrieving %ld'th element\n", i);
         return -1;
      }
      printf("%s", p);
   }
   printf("===== test of remove\n");
   /*
    * test of remove
    */
   for (i = n - 1; i >= 0; i--) {
      if (!al_remove(al, i, (void **)&p)) {
         fprintf(stderr, "Error removing string from array list\n");
         return -1;
      }
      free(p);
   }
   printf("===== test of destroy(NULL)\n");
   /*
    * test of destroy with NULL userFunction
    */
   al_destroy(al, NULL);
   /*
    * test of insert
    */
   if ((al = al_create(0L)) == NULL) {
      fprintf(stderr, "Error creating array list of strings\n");
      return -1;
   }
   fd = fopen(argv[1], "r");            /* we know we can open it */
   printf("===== test of insert\n");
   while (fgets(buf, 1024, fd) != NULL) {
      if ((p = strdup(buf)) == NULL) {
         fprintf(stderr, "Error duplicating string\n");
         return -1;
      }
      if (!al_insert(al, 0, p)) {
         fprintf(stderr, "Error adding string to array list\n");
         return -1;
      }
   }
   fclose(fd);
   for (i = 0; i < n; i++) {
      if (!al_get(al, i, (void **)&p)) {
         fprintf(stderr, "Error retrieving %ld'th element\n", i);
         return -1;
      }
      printf("%s", p);
   }
   printf("===== test of set\n");
   /*
    * test of set
```

```
     */
    for (i = 0; i < n; i++) {
       char bf[1024], *q;
       sprintf(bf, "line %ld\n", i);
       if ((p = strdup(bf)) == NULL) {
          fprintf(stderr, "Error duplicating string\n");
          return -1;
       }
       if (!al_set(al, p, i, (void **)&q)) {
          fprintf(stderr, "Error replacing %ld'th element\n", i);
          return -1;
       }
       free(q);
    }
    printf("===== test of toArray\n");
    /*
     * test of toArray
     */
    if ((array = (char **)al_toArray(al, &n)) == NULL) {
       fprintf(stderr, "Error in invoking al_toArray()\n");
       return -1;
    }
    for (i = 0; i < n; i++) {
       printf("%s", array[i]);
    }
    free(array);
    printf("===== test of iterator\n");
    /*
     * test of iterator
     */
    if ((it = al_it_create(al)) == NULL) {
       fprintf(stderr, "Error in creating iterator\n");
       return -1;
    }
    while (it_hasNext(it)) {
       char *p;
       (void) it_next(it, (void **)&p);
       printf("%s", p);
    }
    it_destroy(it);
    printf("===== test of destroy(free)\n");
    /*
     * test of destroy with free() as userFunction
     */
    al_destroy(al, free);

    return 0;
 }
```

# Appendix C – LinkedList and TSLinkedList

## linkedlist.h

```
#ifndef _LINKEDLIST_H_
#define _LINKEDLIST_H_

/* BSD header removed to save space */

/*
 * interface definition for generic linked list
 *
 * patterned roughly after Java 6 LinkedList generic class, with many
 * duplicate methods removed
 */

#include "iterator.h"

typedef struct linkedlist LinkedList;          /* opaque type definition */

/*
 * create a linked list
 *
 * returns a pointer to the linked list, or NULL if there are malloc() errors
 */
LinkedList *ll_create(void);

/*
 * destroys the linked list; for each element, if userFunction != NULL, invokes
 * userFunction on the element; then returns any list structure associated with
 * the element; finally, deletes any remaining structures associated with the
 * list
 */
void ll_destroy(LinkedList *ll, void (*userFunction)(void *element));

/*
 * appends `element' to the end of the list
 *
 * returns 1 if successful, 0 if unsuccesful (malloc errors)
 */
int ll_add(LinkedList *ll, void *element);

/*
 * inserts `element' at the specified position in the list;
 * all elements from `index' upwards are shifted one position;
 * if current size is N, 0 <= index <= N must be true
 *
 * returns 1 if successful, 0 if unsuccessful (malloc errors)
 */
int ll_insert(LinkedList *ll, long i, void *element);

/*
 * inserts `element' at the beginning of the list
 * equivalent to ll_insert(ll, 0, element);
 */
int ll_addFirst(LinkedList *ll, void *element);

/*
 * appends `element' at the end of the list
 * equivalent to ll_add(ll, element);
 */
int ll_addLast(LinkedList *ll, void *element);

/*
 * clears the linked list; for each element, if userFunction != NULL, invokes
 * userFunction on the element; then returns any list structure associated with
 * the element
 *
 * upon return, the list is empty
 */
void ll_clear(LinkedList *ll, void (*userFunction)(void *element));

/*
 * Retrieves, but does not remove, the element at the specified index
 *
 * return 1 if successful, 0 if not
 */
int ll_get(LinkedList *ll, long index, void **element);

/*
 * Retrieves, but does not remove, the first element
 *
 * return 1 if successful, 0 if not
 */
int ll_getFirst(LinkedList *ll, void **element);

/*
 * Retrieves, but does not remove, the last element
 *
 * return 1 if successful, 0 if not
 */
int ll_getLast(LinkedList *ll, void **element);

/*
 * Retrieves, and removes, the element at the specified index
 *
 * return 1 if successful, 0 if not
 */
int ll_remove(LinkedList *ll, long index, void **element);

/*
 * Retrieves, and removes, the first element
 *
 * return 1 if successful, 0 if not
 */
int ll_removeFirst(LinkedList *ll, void **element);

/*
 * Retrieves, and removes, the last element
 *
 * return 1 if successful, 0 if not
 */
int ll_removeLast(LinkedList *ll, void **element);

/*
 * Replaces the element at the specified index; the previous element
 * is returned in `*previous'
 *
 * return 1 if successful, 0 if not
 */
int ll_set(LinkedList *ll, long index, void *element, void **previous);

/*
 * returns the number of elements in the linked list
 */
long ll_size(LinkedList *ll);

/*
 * returns an array containing all of the elements of the linked list in
 * proper sequence (from first to last element); returns the length of the
 * list in `len'
 *
 * returns pointer to void * array of elements, or NULL if malloc failure
 */
void **ll_toArray(LinkedList *ll, long *len);
```

```
/*
 * creates an iterator for running through the linked list
 *
 * returns pointer to the Iterator or NULL
 */
Iterator *ll_it_create(LinkedList *ll);

#endif /* _LINKEDLIST_H_ */
```

## tslinkedlist.h

```
#ifndef _TSLINKEDLIST_H_
#define _TSLINKEDLIST_H_

/* BSD header removed to save space */

/* interface definition for thread-safe generic linked list
 *
 * patterned roughly after Java 6 LinkedList generic class, with many
 * duplicate methods removed
 */

#include "tsiterator.h"

typedef struct tslinkedlist TSLinkedList;

/*
 * create a linked list
 *
 * returns a pointer to the linked list, or NULL if there are malloc() errors
 */
TSLinkedList *tsll_create(void);

/*
 * destroys the linked list; for each element, if userFunction != NULL, invokes
 * userFunction on the element and then returns any list structures to the heap
 * then completely deletes the list structures
 */
void tsll_destroy(TSLinkedList *ll, void (*userFunction)(void *element));

/*
 * obtains the lock for exclusive access
 */
void tsll_lock(TSLinkedList *ll);

/*
 * returns the lock
 */
void tsll_unlock(TSLinkedList *ll);

/*
 * appends `element' to the end of the list
 *
 * returns 1 if successful, 0 if unsuccesful (malloc errors)
 */
int tsll_add(TSLinkedList *ll, void *element);

/*
 * inserts `element' at the specified position in the list;
 * all elements from `index' upwards are shifted one position;
 * if current size is N, 0 <= index <= N must be true
 *
 * returns 1 if successful, 0 if unsuccessful (malloc errors)
 */
int tsll_insert(TSLinkedList *ll, long i, void *element);

/*
 * inserts `element' at the beginning of the list
 * equivalent to tsll_insert(ll, 0, element);
 */
int tsll_addFirst(TSLinkedList *ll, void *element);

/*
 * appends `element' at the end of the list
 * equivalent to tsll_add(ll, element);
 */
int tsll_addLast(TSLinkedList *ll, void *element);

/*
 * clears the linked list; for each element, if userFunction != NULL, invokes
 * userFunction on the element and then returns any list structures to the heap
 * upon return, the list is empty
 */
void tsll_clear(TSLinkedList *ll, void (*userFunction)(void *element));

/*
 * Retrieves, but does not remove, the element at the specified index
 *
 * return 1 if successful, 0 if not
 */
int tsll_get(TSLinkedList *ll, long index, void **element);

/*
 * Retrieves, but does not remove, the first element
 *
 * return 1 if successful, 0 if not
 */
int tsll_getFirst(TSLinkedList *ll, void **element);

/*
 * Retrieves, but does not remove, the last element
 *
 * return 1 if successful, 0 if not
 */
int tsll_getLast(TSLinkedList *ll, void **element);

/*
 * Retrieves, and removes, the element at the specified index
 *
 * return 1 if successful, 0 if not
 */
int tsll_remove(TSLinkedList *ll, long index, void **element);

/*
 * Retrieves, and removes, the first element
 *
 * return 1 if successful, 0 if not
 */
int tsll_removeFirst(TSLinkedList *ll, void **element);

/*
 * Retrieves, and removes, the last element
 *
 * return 1 if successful, 0 if not
 */
int tsll_removeLast(TSLinkedList *ll, void **element);

/*
 * Replaces the element at the specified index; the previous element
 * is returned in `*previous'
 *
 * return 1 if successful, 0 if not
 */
int tsll_set(TSLinkedList *ll, long index, void *element, void **previous);

/*
```

```
 * returns the number of elements in the linked list
 */
long tsll_size(TSLinkedList *ll);

/*
 * returns an array containing all of the elements of the linked list in
 * proper sequence (from first to last element); returns the length of the
 * list in `len'
 *
 * returns poijnter to void * array of elements, or NULL if malloc failure
 */
void **tsll_toArray(TSLinkedList *ll, long *len);

/*
 * creates an iterator for running through the linked list
 *
 * returns pointer to the Iterator or NULL
 */
TSIterator *tsll_it_create(TSLinkedList *ll);

#endif /* _TSLINKEDLIST_H_ */
```

## linkedlist.c

```
/* BSD header removed to save space */

/*
 * implementation for generic linked list
 */

#include "linkedlist.h"
#include <stdlib.h>

#define SENTINEL(p) (&(p)->sentinel)
#define FL_INCREMENT 128   /* number of entries to add to free list */

typedef struct llnode {
   struct llnode *next;
   struct llnode *prev;
   void *element;
} LLNode;

struct linkedlist {
   long size;
   LLNode *freel;
   LLNode sentinel;
};

/*
 * local routines for maintaining free list of LLNode's
 */

static void putEntry(LinkedList *ll, LLNode *p) {
   p->element = NULL;
   p->next = ll->freel;
   ll->freel = p;
}

static LLNode *getEntry(LinkedList *ll) {
   LLNode *p;

   if ((p = ll->freel) == NULL) {
      long i;
      for (i = 0; i < FL_INCREMENT; i++) {
         p = (LLNode *)malloc(sizeof(LLNode));
        if (p == NULL)
            break;
         putEntry(ll, p);
      }
      p = ll->freel;
   }
   if (p != NULL)
      ll->freel = p->next;
   return p;
}

LinkedList *ll_create(void) {
   LinkedList *ll;

   ll = (LinkedList *)malloc(sizeof(LinkedList));
   if (ll != NULL) {
      ll->size = 0l;
      ll->freel = NULL;
      ll->sentinel.next = SENTINEL(ll);
      ll->sentinel.prev = SENTINEL(ll);
   }
   return ll;
}

/*
 * traverses linked list, calling userFunction on each element and freeing
 * node associated with element
 */
static void purge(LinkedList *ll, void (*userFunction)(void *element)) {
   LLNode *cur = ll->sentinel.next;

   while (cur != SENTINEL(ll)) {
      LLNode *next;
      if (userFunction != NULL)
         (*userFunction)(cur->element);
      next = cur->next;
      putEntry(ll, cur);
      cur = next;
   }
}

void ll_destroy(LinkedList *ll, void (*userFunction)(void *element)) {
   LLNode *p;
   purge(ll, userFunction);
   p = ll->freel;
   while (p != NULL) {           /* return nodes on free list */
      LLNode *q;
      q = p->next;
      free(p);
      p = q;
   }
   free(ll);
}

/*
 * link `p' between `before' and `after'
 * must work correctly if `before' and `after' are the same node
 * (i.e. the sentinel)
 */
static void link(LLNode *before, LLNode *p, LLNode *after) {
   p->next = after;
   p->prev = before;
   after->prev = p;
   before->next = p;
}

int ll_add(LinkedList *ll, void *element) {
   return ll_addLast(ll, element);
}

int ll_insert(LinkedList *ll, long index, void *element) {
   int status = 0;
   LLNode *p;

   if (index <= ll->size && (p = getEntry(ll)) != NULL) {
      long n;
      LLNode *b;

      p->element = element;
      status = 1;
      for (n = 0, b = SENTINEL(ll); n < index; n++, b = b->next)
         ;
      link(b, p, b->next);
      ll->size++;
   }
   return status;
}

int ll_addFirst(LinkedList *ll, void *element) {
   int status = 0;
   LLNode *p = getEntry(ll);

   if (p != NULL) {
      p->element = element;
      status = 1;
      link(SENTINEL(ll), p, SENTINEL(ll)->next);
      ll->size++;
   }
   return status;
}

int ll_addLast(LinkedList *ll, void *element) {
   int status = 0;
   LLNode *p = getEntry(ll);

   if (p != NULL) {
      p->element = element;
      status = 1;
      link(SENTINEL(ll)->prev, p, SENTINEL(ll));
      ll->size++;
   }
   return status;
}

void ll_clear(LinkedList *ll, void (*userFunction)(void *element)) {
   purge(ll, userFunction);
   ll->size = 0L;
   ll->sentinel.next = SENTINEL(ll);
   ll->sentinel.prev = SENTINEL(ll);
}

int ll_get(LinkedList *ll, long index, void **element) {
   int status = 0;

   if (index < ll->size) {
      long n;
      LLNode *p;

      status = 1;
      for (n = 0, p = SENTINEL(ll)->next; n < index; n++, p = p->next)
         ;
      *element = p->element;
   }
   return status;
}

int ll_getFirst(LinkedList *ll, void **element) {
   int status = 0;
   LLNode *p = SENTINEL(ll)->next;
   if (p != SENTINEL(ll)) {
      status = 1;
      *element = p->element;
   }
   return status;
}

int ll_getLast(LinkedList *ll, void **element) {
   int status = 0;
   LLNode *p = SENTINEL(ll)->prev;

   if (p != SENTINEL(ll)) {
      status = 1;
      *element = p->element;
   }
   return status;
}

/*
 * unlinks the LLNode from the doubly-linked list
 */
static void unlink(LLNode *p) {
   p->prev->next = p->next;
   p->next->prev = p->prev;
}

int ll_remove(LinkedList *ll, long index, void **element) {
   int status = 0;

   if (index < ll->size) {
      long n;
      LLNode *p;

      status = 1;
      for (n = 0, p = SENTINEL(ll)->next; n < index; n++, p = p->next)
         ;
      *element = p->element;
      unlink(p);
      putEntry(ll, p);
      ll->size--;
   }
   return status;
}

int ll_removeFirst(LinkedList *ll, void **element) {
   int status = 0;
   LLNode *p = SENTINEL(ll)->next;

   if (p != SENTINEL(ll)) {
      status = 1;
      *element = p->element;
      unlink(p);
      putEntry(ll, p);
      ll->size--;
   }
   return status;
}

int ll_removeLast(LinkedList *ll, void **element) {
   int status = 0;
   LLNode *p = SENTINEL(ll)->prev;

   if (p != SENTINEL(ll)) {
      status = 1;
      *element = p->element;
      unlink(p);
      putEntry(ll, p);
      ll->size--;
   }
   return status;
}

int ll_set(LinkedList *ll, long index, void *element, void **previous) {
   int status = 0;

   if (index < ll->size) {
      long n;
      LLNode *p;

      status = 1;
      for (n = 0, p = SENTINEL(ll)->next; n < index; n++, p = p->next)
         ;
      *previous = p->element;
      p->element = element;
   }
   return status;
}

long ll_size(LinkedList *ll) {
   return ll->size;
}

/*
 * local function to generate array of element values on the heap
 *
 * returns pointer to array or NULL if malloc failure
 */
static void **genArray(LinkedList *ll) {
   void **tmp = NULL;
   if (ll->size > 0L) {
      size_t nbytes = ll->size * sizeof(void *);
      tmp = (void **)malloc(nbytes);
      if (tmp != NULL) {
         long i;
        LLNode *p;
        for (i = 0, p = SENTINEL(ll)->next; i < ll->size; i++, p = p->next)
            tmp[i] = p->element;
      }
   }
   return tmp;
}

void **ll_toArray(LinkedList *ll, long *len) {
   void **tmp = genArray(ll);

   if (tmp != NULL)
      *len = ll->size;
   return tmp;
}

Iterator *ll_it_create(LinkedList *ll) {
   Iterator *it = NULL;
   void **tmp = genArray(ll);

   if (tmp != NULL) {
      it = it_create(ll->size, tmp);
      if (it == NULL)
         free(tmp);
   }
   return it;
}
```

## tslinkedlist.c

```
/* BSD header removed to save space */

/*
```

```
 * implementation for thread-safe generic linked list
 */

#include "tslinkedlist.h"
#include "linkedlist.h"
#include <stdlib.h>
#include <pthread.h>

#define LOCK(ll) &((ll)->lock)

struct tslinkedlist {
   LinkedList *ll;
   pthread_mutex_t lock;   /* this is a recursive lock */
};

TSLinkedList *tsll_create(void) {
   TSLinkedList *tsll = (TSLinkedList *)malloc(sizeof(TSLinkedList));

   if (tsll != NULL) {
      LinkedList *ll = ll_create();

      if (ll == NULL) {
         free(tsll);
        tsll = NULL;
      } else {
         pthread_mutexattr_t ma;
        pthread_mutexattr_init(&ma);
        pthread_mutexattr_settype(&ma, PTHREAD_MUTEX_RECURSIVE);
        tsll->ll = ll;
        pthread_mutex_init(LOCK(tsll), &ma);
        pthread_mutexattr_destroy(&ma);
      }
   }
   return tsll;
}

void tsll_destroy(TSLinkedList *tsll, void (*userFunction)(void *element)) {
   pthread_mutex_lock(LOCK(tsll));
   ll_destroy(tsll->ll, userFunction);
   pthread_mutex_unlock(LOCK(tsll));
   pthread_mutex_destroy(LOCK(tsll));
   free(tsll);
}

void tsll_lock(TSLinkedList *tsll) {
   pthread_mutex_lock(LOCK(tsll));
}

void tsll_unlock(TSLinkedList *tsll) {
   pthread_mutex_unlock(LOCK(tsll));
}

int tsll_add(TSLinkedList *tsll, void *element) {
   int result;
   pthread_mutex_lock(LOCK(tsll));
   result = ll_add(tsll->ll, element);
   pthread_mutex_unlock(LOCK(tsll));
   return result;
}

int tsll_insert(TSLinkedList *tsll, long index, void *element) {
   int result;
   pthread_mutex_lock(LOCK(tsll));
   result = ll_insert(tsll->ll, index, element);
   pthread_mutex_unlock(LOCK(tsll));
   return result;
}
```

```
int tsll_addFirst(TSLinkedList *tsll, void *element) {
   int result;
   pthread_mutex_lock(LOCK(tsll));
   result = ll_addFirst(tsll->ll, element);
   pthread_mutex_unlock(LOCK(tsll));
   return result;
}

int tsll_addLast(TSLinkedList *tsll, void *element) {
   int result;
   pthread_mutex_lock(LOCK(tsll));
   result = ll_addLast(tsll->ll, element);
   pthread_mutex_unlock(LOCK(tsll));
   return result;
}

void tsll_clear(TSLinkedList *tsll, void (*userFunction)(void *element)) {
   pthread_mutex_lock(LOCK(tsll));
   ll_clear(tsll->ll, userFunction);
   pthread_mutex_unlock(LOCK(tsll));
}

int tsll_get(TSLinkedList *tsll, long index, void **element) {
   int result;
   pthread_mutex_lock(LOCK(tsll));
   result = ll_get(tsll->ll, index, element);
   pthread_mutex_unlock(LOCK(tsll));
   return result;
}

int tsll_getFirst(TSLinkedList *tsll, void **element) {
   int result;
   pthread_mutex_lock(LOCK(tsll));
   result = ll_getFirst(tsll->ll, element);
   pthread_mutex_unlock(LOCK(tsll));
   return result;
}

int tsll_getLast(TSLinkedList *tsll, void **element) {
   int result;
   pthread_mutex_lock(LOCK(tsll));
   result = ll_getLast(tsll->ll, element);
   pthread_mutex_unlock(LOCK(tsll));
   return result;
}

int tsll_remove(TSLinkedList *tsll, long index, void **element) {
   int result;
   pthread_mutex_lock(LOCK(tsll));
   result = ll_remove(tsll->ll, index, element);
   pthread_mutex_unlock(LOCK(tsll));
   return result;
}

int tsll_removeFirst(TSLinkedList *tsll, void **element) {
   int result;
   pthread_mutex_lock(LOCK(tsll));
   result = ll_removeFirst(tsll->ll, element);
   pthread_mutex_unlock(LOCK(tsll));
   return result;
}

int tsll_removeLast(TSLinkedList *tsll, void **element) {
   int result;
   pthread_mutex_lock(LOCK(tsll));
   result = ll_removeLast(tsll->ll, element);
   pthread_mutex_unlock(LOCK(tsll));
   return result;
}

int tsll_set(TSLinkedList *tsll, long index, void *element, void **previous) {
   int result;
   pthread_mutex_lock(LOCK(tsll));
   result = ll_set(tsll->ll, index, element, previous);
   pthread_mutex_unlock(LOCK(tsll));
   return result;
}

long tsll_size(TSLinkedList *tsll) {
   long result;
   pthread_mutex_lock(LOCK(tsll));
   result = ll_size(tsll->ll);
   pthread_mutex_unlock(LOCK(tsll));
   return result;
}

void **tsll_toArray(TSLinkedList *tsll, long *len) {
   void **result;
   pthread_mutex_lock(LOCK(tsll));
   result = ll_toArray(tsll->ll, len);
   pthread_mutex_unlock(LOCK(tsll));
   return result;
}

TSIterator *tsll_it_create(TSLinkedList *tsll) {
   TSIterator *it = NULL;
   void **tmp;
   long len;

   pthread_mutex_lock(LOCK(tsll));
   tmp = ll_toArray(tsll->ll, &len);
   if (tmp != NULL) {
      it = tsit_create(LOCK(tsll), len, tmp);
      if (it == NULL)
         free(tmp);
   }
   if (it == NULL)
      pthread_mutex_unlock(LOCK(tsll));
   return it;
}
```

## lltest.c (you can create your own tslltest.c)

```
/* BSD header removed to save space */

#include "linkedlist.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
   char buf[1024];
   char *p;
   LinkedList *ll;
   long i, n;
   FILE *fd;
   char **array;
   Iterator *it;

   if (argc != 2) {
      fprintf(stderr, "usage: ./lltest file\n");
      return -1;
   }
   if ((ll = ll_create()) == NULL) {
      fprintf(stderr, "Error creating array list of strings\n");
      return -1;
   }
```

```
if ((fd = fopen(argv[1], "r")) == NULL) {
   fprintf(stderr, "Unable to open %s to read\n", argv[1]);
   return -1;
}
/*
 * test of add()
 */
printf("===== test of add\n");
while (fgets(buf, 1024, fd) != NULL) {
   if ((p = strdup(buf)) == NULL) {
      fprintf(stderr, "Error duplicating string\n");
      return -1;
   }
   if (!ll_add(ll, p)) {
      fprintf(stderr, "Error adding string to array list\n");
      return -1;
   }
}
fclose(fd);
n = ll_size(ll);
/*
 * test of get()
 */
printf("===== test of get\n");
for (i = 0; i < n; i++) {
   if (!ll_get(ll, i, (void **)&p)) {
      fprintf(stderr, "Error retrieving %ld'th element\n", i);
      return -1;
   }
   printf("%s", p);
}
/*
 * test of remove
 */
printf("===== test of remove\n");
for (i = n - 1; i >= 0; i--) {
   if (!ll_remove(ll, i, (void **)&p)) {
      fprintf(stderr, "Error removing string from array list\n");
      return -1;
   }
   free(p);
}
/*
 * test of destroy with NULL userFunction
 */
printf("===== test of destroy(NULL)\n");
ll_destroy(ll, NULL);
/*
 * test of insert
 */
if ((ll = ll_create()) == NULL) {
   fprintf(stderr, "Error creating array list of strings\n");
   return -1;
}
fd = fopen(argv[1], "r");            /* we know we can open it */
printf("===== test of insert\n");
while (fgets(buf, 1024, fd) != NULL) {
   if ((p = strdup(buf)) == NULL) {
      fprintf(stderr, "Error duplicating string\n");
      return -1;
   }
   if (!ll_insert(ll, 0, p)) {
      fprintf(stderr, "Error adding string to array list\n");
      return -1;
   }
}
fclose(fd);
for (i = 0; i < n; i++) {
      if (!ll_get(ll, i, (void **)&p)) {
         fprintf(stderr, "Error retrieving %ld'th element\n", i);
         return -1;
      }
      printf("%s", p);
   }
   /*
    * test of set
    */
   printf("===== test of set\n");
   for (i = 0; i < n; i++) {
      char bf[1024], *q;
      sprintf(bf, "line %ld\n", i);
      if ((p = strdup(bf)) == NULL) {
         fprintf(stderr, "Error duplicating string\n");
         return -1;
      }
      if (!ll_set(ll, i, p, (void **)&q)) {
         fprintf(stderr, "Error replacing %ld'th element\n", i);
         return -1;
      }
      free(q);
   }
   /*
    * test of toArray
    */
   printf("===== test of toArray\n");
   if ((array = (char **)ll_toArray(ll, &n)) == NULL) {
      fprintf(stderr, "Error in invoking ll_toArray()\n");
      return -1;
   }
   for (i = 0; i < n; i++) {
      printf("%s", array[i]);
   }
   free(array);
   /*
    * test of iterator
    */
   printf("===== test of iterator\n");
   if ((it = ll_it_create(ll)) == NULL) {
      fprintf(stderr, "Error in creating iterator\n");
      return -1;
   }
   while (it_hasNext(it)) {
      char *p;
      (void) it_next(it, (void **)&p);
      printf("%s", p);
   }
   it_destroy(it);
   /*
    * test of destroy with free() as userFunction
    */
   printf("===== test of destroy(free)\n");
   ll_destroy(ll, free);

   return 0;
}
```

# Appendix D – HashMap and TSHashMap

## hashmap.h

```
#ifndef _HASHMAP_H_
#define _HASHMAP_H_

/* BSD header removed to save space */

#include "iterator.h"

/*
 * interface definition for generic hashmap implementation
 *
 * patterned roughly after Java 6 HashMap generic class with String keys
 */

typedef struct hashmap HashMap;   /* opaque type definition */
typedef struct hmentry HMEntry;

/*
 * create a hashmap with the specified capacity and load factor;
 * if capacity == 0, a default initial capacity (16 elements) is used
 * if loadFactor == 0.0, a default load factor (0.75) is used
 * if number of elements/number of buckets exceeds the load factor, the
 * table is resized, doubling the number of buckets, up to a max number
 * of buckets (134,217,728)
 *
 * returns a pointer to the hashmap, or NULL if there are malloc() errors
 */
HashMap *hm_create(long capacity, double loadFactor);

/*
 * destroys the hashmap; for each HMEntry, if userFunction != NULL,
 * it is invoked on the element in that entry ; the storage associated with
 * the hashmap is then returned to the heap
 */
void hm_destroy(HashMap *hm, void (*userFunction)(void *element));

/*
 * clears all elements from the hashmap; for each HMEntry,
 * if userFunction != NULL, it is invoked on the element in that entry;
 * any storage associated with the entry in the hashmap is then
 * returned to the heap
 *
 * upon return, the hashmap will be empty
 */
void hm_clear(HashMap *hm, void (*userFunction)(void *element));

/*
 * returns 1 if hashmap has an entry for `key', 0 otherwise
 */
int hm_containsKey(HashMap *hm, char *key);

/*
 * returns an array containing all of the entries of the hashmap in
 * an arbitrary order; returns the length of the list in `len'
 *
 * returns pointer to HMEntry * array of elements, or NULL if malloc failure
 */
HMEntry **hm_entryArray(HashMap *hm, long *len);

/*
 * returns the element to which the specified key is mapped in `*element'
 *
 * returns 1 if successful, 0 if no mapping for `key'
 */
```

```
int hm_get(HashMap *hm, char *key, void **element);

/*
 * returns 1 if hashmap is empty, 0 if it is not
 */
int hm_isEmpty(HashMap *hm);

/*
 * returns an array containing all of the keys in the hashmap in
 * an arbitrary order; returns the length of the list in `len'
 *
 * returns pointer to char * array of keys, or NULL if malloc failure
 */
char **hm_keyArray(HashMap *hm, long *len);

/*
 * associates `element' with key'; if this replaces an existing mapping, the
 * old value is returned in `*previous'; othersize *previous == NULL
 *
 *
 * returns 1 if successful, 0 if not (malloc failure)
 */
int hm_put(HashMap *hm, char *key, void *element, void **previous);

/*
 * removes the entry associated with `key' if one exists; returns element
 * associated with key in `*element'
 *
 * returns 1 if successful, 0 if `i'th position was not occupied
 */
int hm_remove(HashMap *hm, char *key, void **element);

/*
 * returns the number of mappings in the hashmap
 */
long hm_size(HashMap *hm);

/*
 * create generic iterator to this hashmap
 * note that iterator will return pointers to HMEntry's
 *
 * returns pointer to the Iterator or NULL if failure
 */
Iterator *hm_it_create(HashMap *hm);

/*
 * accessor methods for obtaining key and value from an HMEntry
 * used with return from it_next on iterator
 */
char *hmentry_key(HMEntry *hme);
void *hmentry_value(HMEntry *hme);

#endif /* _HASHMAP_H_ */
```

## tshashmap.h

```
#ifndef _TSHASHMAP_H_
#define _TSHASHMAP_H_

/* BSD header removed to save space */

#include "tsiterator.h"
#include "hashmap.h"

/*
 * interface definition for thread-safe generic hashmap implementation
 *
 * patterned roughly after Java 6 HashMap generic class with String keys
 */
```

```
typedef struct tshashmap TSHashMap;     /* opaque type definition */

/*
 * create a hashmap with the specified capacity and load factor;
 * if capacity == 0, a default initial capacity (16 elements) is used
 * if loadFactor == 0.0, a default load factor (0.75) is used
 * if number of elements/number of buckets ever exceeds the load factor,
 * the hashmap is resized by doubling the number of buckets, up to
 * a maximum number of buckets (134,217,728)
 *
 * returns a pointer to the hashmap, or NULL if there are malloc() errors
 */
TSHashMap *tshm_create(long capacity, double loadFactor);

/*
 * destroys the hashmap; for each HMEntry, if userFunction != NULL,
 * it is invoked on the element in that entry ; the storage associated with
 * the hashmap is then returned to the heap
 */
void tshm_destroy(TSHashMap *hm, void (*userFunction)(void *element));

/*
 * obtains the lock for exclusive access
 */
void tshm_lock(TSHashMap *hm);

/*
 * returns the lock
 */
void tshm_unlock(TSHashMap *hm);

/*
 * clears all elements from the hashmap; for each HMEntry,
 * if userFunction != NULL, it is invoked on the element in that entry;
 * any storage associated with the entry in the hashmap is then
 * returned to the heap
 *
 * upon return, the hashmap will be empty
 */
void tshm_clear(TSHashMap *hm, void (*userFunction)(void *element));

/*
 * returns 1 if hashmap has an entry for `key', 0 otherwise
 */
int tshm_containsKey(TSHashMap *hm, char *key);

/*
 * returns an array containing all of the entries of the hashmap in
 * an arbitrary order; returns the length of the list in `len'
 *
 * returns pointer to HMEntry * array of elements, or NULL if malloc failure
 */
HMEntry **tshm_entryArray(TSHashMap *hm, long *len);

/*
 * returns the element to which the specified key is mapped in `*element'
 *
 * returns 1 if successful, 0 if no mapping for `key'
 */
int tshm_get(TSHashMap *hm, char *key, void **element);

/*
 * returns 1 if hashmap is empty, 0 if it is not
 */
int tshm_isEmpty(TSHashMap *hm);

/*
```

```
 * returns an array containing all of the keys in the hashmap in
 * an arbitrary order; returns the length of the list in `len'
 *
 * returns pointer to char * array of keys, or NULL if malloc failure
 */
char **tshm_keyArray(TSHashMap *hm, long *len);

/*
 * associates `element' with key'; if this replaces an existing mapping, the
 * old value is returned in `*previous'
 *
 *
 * returns 1 if successful, 0 if not (malloc failure)
 */
int tshm_put(TSHashMap *hm, char *key, void *element, void **previous);

/*
 * removes the entry associated with `key' if one exists; returns element
 * associated with key in `*element'
 *
 * returns 1 if successful, 0 if `i'th position was not occupied
 */
int tshm_remove(TSHashMap *hm, char *key, void **element);

/*
 * returns the number of mappings in the hashmap
 */
long tshm_size(TSHashMap *hm);

/*
 * create generic iterator to this arraylist
 * note that iterator will return pointers to HMEntry's
 *
 * returns pointer to the Iterator or NULL if failure
 */
TSIterator *tshm_it_create(TSHashMap *hm);

#endif /* _TSHASHMAP_H_ */
```

## hashmap.c

```
/* BSD header removed to save space */

#include "hashmap.h"
#include <stdlib.h>
#include <string.h>

#define DEFAULT_CAPACITY 16
#define MAX_CAPACITY 134217728L
#define DEFAULT_LOAD_FACTOR 0.75
#define TRIGGER 100 /* number of changes that will trigger a load check */

struct hashmap {
   long size;
   long capacity;
   long changes;
   double load;
   double loadFactor;
   double increment;
   HMEntry **buckets;
};

struct hmentry {
   struct hmentry *next;
   char *key;
   void *element;
};

/*
```

```
 * generate hash value from key; value returned in range of 0..N-1
 */
#define SHIFT 7L                  /* should be prime */
static long hash(char *key, long N) {
   long ans = 0L;
   char *sp;

   for (sp = key; *sp != '\0'; sp++)
      ans = ((SHIFT * ans) + *sp) % N;
   return ans;
}

HashMap *hm_create(long capacity, double loadFactor) {
   HashMap *hm;
   long N;
   double lf;
   HMEntry **array;
   long i;

   hm = (HashMap *)malloc(sizeof(HashMap));
   if (hm != NULL) {
      N = ((capacity > 0) ? capacity : DEFAULT_CAPACITY);
      if (N > MAX_CAPACITY)
         N = MAX_CAPACITY;
      lf = ((loadFactor > 0.000001) ? loadFactor : DEFAULT_LOAD_FACTOR);
      array = (HMEntry **)malloc(N * sizeof(HMEntry *));
      if (array != NULL) {
         hm->capacity = N;
         hm->loadFactor = lf;
         hm->size = 0L;
         hm->load = 0.0;
         hm->changes = 0L;
         hm->increment = 1.0 / (double)N;
         hm->buckets = array;
         for (i = 0; i < N; i++)
            array[i] = NULL;
      } else {
         free(hm);
         hm = NULL;
      }
   }
   return hm;
}

/*
 * traverses the hashmap, calling userFunction on each element
 * then frees storage associated with the key and the HMEntry structure
 */
static void purge(HashMap *hm, void (*userFunction)(void *element)) {

   long i;

   for (i = 0L; i < hm->capacity; i++) {
      HMEntry *p, *q;
      p = hm->buckets[i];
      while (p != NULL) {
         if (userFunction != NULL)
            (*userFunction)(p->element);
         q = p->next;
         free(p->key);
         free(p);
         p = q;
      }
   }
}

void hm_destroy(HashMap *hm, void (*userFunction)(void *element)) {
   purge(hm, userFunction);
   free(hm->buckets);
   free(hm);
}

void hm_clear(HashMap *hm, void (*userFunction)(void *element)) {
   purge(hm, userFunction);
   hm->size = 0;
   hm->load = 0.0;
   hm->changes = 0;
}

/*
 * local function to locate key in a hashmap
 *
 * returns pointer to entry, if found, as function value; NULL if not found
 * returns bucket index in `bucket'
 */
static HMEntry *findKey(HashMap *hm, char *key, long *bucket) {
   long i = hash(key, hm->capacity);
   HMEntry *p;

   *bucket = i;
   for (p = hm->buckets[i]; p != NULL; p = p->next) {
      if (strcmp(p->key, key) == 0) {
         break;
      }
   }
   return p;
}

int hm_containsKey(HashMap *hm, char *key) {
   long bucket;

   return (findKey(hm, key, &bucket) != NULL);
}

/*
 * local function for generating an array of HMEntry * from a hashmap
 *
 * returns pointer to the array or NULL if malloc failure
 */
static HMEntry **entries(HashMap *hm) {
   HMEntry **tmp = NULL;
   if (hm->size > 0L) {
      size_t nbytes = hm->size * sizeof(HMEntry *);
      tmp = (HMEntry **)malloc(nbytes);
      if (tmp != NULL) {
         long i, n = 0L;
         for (i = 0L; i < hm->capacity; i++) {
            HMEntry *p;
            p = hm->buckets[i];
            while (p != NULL) {
               tmp[n++] = p;
               p = p->next;
            }
         }
      }
   }
   return tmp;
}

HMEntry **hm_entryArray(HashMap *hm, long *len) {
   HMEntry **tmp = entries(hm);

   if (tmp != NULL)
      *len = hm->size;
   return tmp;
}
```

```
int hm_get(HashMap *hm, char *key, void **element) {
   long i;
   HMEntry *p;
   int ans = 0;

   p = findKey(hm, key, &i);
   if (p != NULL) {
      ans = 1;
      *element = p->element;
   }
   return ans;
}

int hm_isEmpty(HashMap *hm) {
   return (hm->size == 0L);
}

/*
 * local function for generating an array of keys from a hashmap
 *
 * returns pointer to the array or NULL if malloc failure
 */
static char **keys(HashMap *hm) {
   char **tmp = NULL;
   if (hm->size > 0L) {
      size_t nbytes = hm->size * sizeof(char *);
      tmp = (char **)malloc(nbytes);
      if (tmp != NULL) {
         long i, n = 0L;
         for (i = 0L; i < hm->capacity; i++) {
            HMEntry *p;
            p = hm->buckets[i];
            while (p != NULL) {
               tmp[n++] = p->key;
               p = p->next;
            }
         }
      }
   }
   return tmp;
}

char **hm_keyArray(HashMap *hm, long *len) {
   char **tmp = keys(hm);

   if (tmp != NULL)
      *len = hm->size;
   return tmp;
}

/*
 * routine that resizes the hashmap
 */
void resize(HashMap *hm) {
   int N;
   HMEntry *p, *q, **array;
   long i, j;

   N = 2 * hm->capacity;
   if (N > MAX_CAPACITY)
      N = MAX_CAPACITY;
   array = (HMEntry **)malloc(N * sizeof(HMEntry *));
   if (array == NULL)
      return;
   for (j = 0; j < N; j++)
      array[j] = NULL;
   /*
```

```
    * now redistribute the entries into the new set of buckets
    */
   for (i = 0; i < hm->capacity; i++) {
      for (p = hm->buckets[i]; p != NULL; p = q) {
         q = p->next;
         j = hash(p->key, N);
         p->next = array[j];
         array[j] = p;
      }
   }
   free(hm->buckets);
   hm->buckets = array;
   hm->capacity = N;
   hm->load /= 2.0;
   hm->changes = 0;
   hm->increment = 1.0 / (double)N;
}

int hm_put(HashMap *hm, char *key, void *element, void **previous) {
   long i;
   HMEntry *p;
   int ans = 0;

   if (hm->changes > TRIGGER) {
      hm->changes = 0;
      if (hm->load > hm->loadFactor)
         resize(hm);
   }
   p = findKey(hm, key, &i);
   if (p != NULL) {
      *previous = p->element;
      p->element = element;
      ans = 1;
   } else {
      p = (HMEntry *)malloc(sizeof(HMEntry));
      if (p != NULL) {
         char *q = strdup(key);
         if (q != NULL) {
            p->key = q;
            p->element = element;
            p->next = hm->buckets[i];
            hm->buckets[i] = p;
            *previous = NULL;
            hm->size++;
            hm->load += hm->increment;
            hm->changes++;
            ans = 1;
        } else {
            free(p);
        }
      }
   }
   return ans;
}

int hm_remove(HashMap *hm, char *key, void **element) {
   long i;
   HMEntry *entry;
   int ans = 0;

   entry = findKey(hm, key, &i);
   if (entry != NULL) {
      HMEntry *p, *c;
      *element = entry->element;
      /* determine where the entry lives in the singly linked list */
      for (p = NULL, c = hm->buckets[i]; c != entry; p = c, c = c->next)
         ;
      if (p == NULL)
```

```
         hm->buckets[i] = entry->next;
      else
         p->next = entry->next;
      hm->size--;
      hm->load -= hm->increment;
      hm->changes++;
      free(entry->key);
      free(entry);
      ans = 1;
   }
   return ans;
}

long hm_size(HashMap *hm) {
   return hm->size;
}

Iterator *hm_it_create(HashMap *hm) {
   Iterator *it = NULL;
   void **tmp = (void **)entries(hm);

   if (tmp != NULL) {
      it = it_create(hm->size, tmp);
      if (it == NULL)
         free(tmp);
   }
   return it;
}

char *hmentry_key(HMEntry *hme) {
   return hme->key;
}

void *hmentry_value(HMEntry *hme) {
   return hme->element;
}
```

## tshashmap.c

```
/* BSD header removed to save space */

#include "tshashmap.h"
#include "hashmap.h"
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

#define LOCK(hm) &((hm)->lock)

struct tshashmap {
   HashMap *hm;
   pthread_mutex_t lock;   /* this is a recursive lock */
};

TSHashMap *tshm_create(long capacity, double loadFactor) {
   TSHashMap *tshm = (TSHashMap *)malloc(sizeof(TSHashMap));

   if (tshm != NULL) {
      HashMap *hm = hm_create(capacity, loadFactor);

      if (hm == NULL) {
         free(tshm);
         tshm = NULL;
      } else {
         pthread_mutexattr_t ma;
         pthread_mutexattr_init(&ma);
         pthread_mutexattr_settype(&ma, PTHREAD_MUTEX_RECURSIVE);
         tshm->hm = hm;
         pthread_mutex_init(LOCK(tshm), &ma);
         pthread_mutexattr_destroy(&ma);
      }
   }
   return tshm;
}

void tshm_destroy(TSHashMap *hm, void (*userFunction)(void *element)) {
   pthread_mutex_lock(LOCK(hm));
   hm_destroy(hm->hm, userFunction);
   pthread_mutex_unlock(LOCK(hm));
   pthread_mutex_destroy(LOCK(hm));
   free(hm);
}

void tshm_lock(TSHashMap *hm) {
   pthread_mutex_lock(LOCK(hm));
}

void tshm_unlock(TSHashMap *hm) {
   pthread_mutex_unlock(LOCK(hm));
}

void tshm_clear(TSHashMap *hm, void (*userFunction)(void *element)) {
   pthread_mutex_lock(LOCK(hm));
   hm_clear(hm->hm, userFunction);
   pthread_mutex_unlock(LOCK(hm));
}

int tshm_containsKey(TSHashMap *hm, char *key) {
   int result;
   pthread_mutex_lock(LOCK(hm));
   result = hm_containsKey(hm->hm, key);
   pthread_mutex_unlock(LOCK(hm));
   return  result;
}

HMEntry **tshm_entryArray(TSHashMap *hm, long *len) {
   HMEntry **result;
   pthread_mutex_lock(LOCK(hm));
   result = hm_entryArray(hm->hm, len);
   pthread_mutex_unlock(LOCK(hm));
   return  result;
}

int tshm_get(TSHashMap *hm, char *key, void **element) {
   int result;
   pthread_mutex_lock(LOCK(hm));
   result = hm_get(hm->hm, key, element);
   pthread_mutex_unlock(LOCK(hm));
   return  result;
}

int tshm_isEmpty(TSHashMap *hm) {
   int result;
   pthread_mutex_lock(LOCK(hm));
   result = hm_isEmpty(hm->hm);
   pthread_mutex_unlock(LOCK(hm));
   return  result;
}

char **tshm_keyArray(TSHashMap *hm, long *len) {
   char **result;
   pthread_mutex_lock(LOCK(hm));
   result = hm_keyArray(hm->hm, len);
   pthread_mutex_unlock(LOCK(hm));
   return  result;
}
```

```
int tshm_put(TSHashMap *hm, char *key, void *element, void **previous) {
   int result;
   pthread_mutex_lock(LOCK(hm));
   result = hm_put(hm->hm, key, element, previous);
   pthread_mutex_unlock(LOCK(hm));
   return  result;
}

int tshm_remove(TSHashMap *hm, char *key, void **element) {
   int result;
   pthread_mutex_lock(LOCK(hm));
   result = hm_remove(hm->hm, key, element);
   pthread_mutex_unlock(LOCK(hm));
   return  result;
}

long tshm_size(TSHashMap *hm) {
   long result;
   pthread_mutex_lock(LOCK(hm));
   result = hm_size(hm->hm);
   pthread_mutex_unlock(LOCK(hm));
   return  result;
}

TSIterator *tshm_it_create(TSHashMap *hm) {
   TSIterator *it = NULL;
   void **tmp;
   long len;

   pthread_mutex_lock(LOCK(hm));
   tmp = (void **)hm_entryArray(hm->hm, &len);
   if (tmp != NULL) {
      it = tsit_create(LOCK(hm), len, tmp);
      if (it == NULL)
         free(tmp);
   }
   if (it == NULL)
      pthread_mutex_unlock(LOCK(hm));
   return it;
}
```

## hmtest.c (you can create your own tshmtest.c)

```
/* BSD header removed to save space */

#include "hashmap.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
   char buf[1024];
   char key[20];
   char *p;
   HashMap *hm;
   long i, n;
   FILE *fd;
   HMEntry **array;
   Iterator *it;

   if (argc != 2) {
      fprintf(stderr, "usage: ./hmtest file\n");
      return -1;
   }
   if ((hm = hm_create(0L, 0.0)) == NULL) {
      fprintf(stderr, "Error creating hashmap of strings\n");
      return -1;
   }
   if ((fd = fopen(argv[1], "r")) == NULL) {
```

```
   fprintf(stderr, "Unable to open %s to read\n", argv[1]);
   return -1;
}
/*
 * test of put()
 */
printf("===== test of put when key not in hashmap\n");
i = 0;
while (fgets(buf, 1024, fd) != NULL) {
   char *prev;

   if ((p = strdup(buf)) == NULL) {
      fprintf(stderr, "Error duplicating string\n");
      return -1;
   }
   sprintf(key, "%ld", i++);
   if (!hm_put(hm, key, p, (void**)&prev)) {
      fprintf(stderr, "Error adding key,string to hashmap\n");
      return -1;
   }
}
fclose(fd);
n = hm_size(hm);
/*
 * test of get()
 */
printf("===== test of get\n");
for (i = 0; i < n; i++) {
   char *element;

   sprintf(key, "%ld", i);
   if (!hm_get(hm, key, (void **)&element)) {
      fprintf(stderr, "Error retrieving %ld'th element\n", i);
      return -1;
   }
   printf("%s,%s", key, element);
}
/*
 * test of remove
 */
printf("===== test of remove\n");
printf("Size before remove = %ld\n", n);
for (i = n - 1; i >= 0; i--) {
   sprintf(key, "%ld", i);
   if (!hm_remove(hm, key, (void **)&p)) {
      fprintf(stderr, "Error removing %ld'th element\n", i);
      return -1;
   }
   free(p);
}
printf("Size after remove = %ld\n", hm_size(hm));
/*
 * test of destroy with NULL userFunction
 */
printf("===== test of destroy(NULL)\n");
hm_destroy(hm, NULL);
/*
 * test of insert
 */
if ((hm = hm_create(0L, 3.0)) == NULL) {
   fprintf(stderr, "Error creating hashmap of strings\n");
   return -1;
}
fd = fopen(argv[1], "r");            /* we know we can open it */
i = 0L;
while (fgets(buf, 1024, fd) != NULL) {
   char *prev;
```

```
      if ((p = strdup(buf)) == NULL) {
         fprintf(stderr, "Error duplicating string\n");
         return -1;
      }
      sprintf(key, "%ld", i++);
      if (!hm_put(hm, key, p, (void **)&prev)) {
         fprintf(stderr, "Error adding key,value to hashmap\n");
         return -1;
      }
   }
   fclose(fd);
   /*
    * test of put replacing value associated with an existing key
    */
   printf("===== test of put (replace value associated with key)\n");
   for (i = 0; i < n; i++) {
      char bf[1024], *q;
      sprintf(bf, "line %ld\n", i);
      if ((p = strdup(bf)) == NULL) {
         fprintf(stderr, "Error duplicating string\n");
         return -1;
      }
      sprintf(key, "%ld", i);
      if (!hm_put(hm, key, p, (void **)&q)) {
         fprintf(stderr, "Error replacing %ld'th element\n", i);
         return -1;
      }
      free(q);
   }
   for (i = 0; i < n; i++) {
      char *element;

      sprintf(key, "%ld", i);
      if (!hm_get(hm, key, (void **)&element)) {
         fprintf(stderr, "Error retrieving %ld'th element\n", i);
         return -1;
      }
      printf("%s,%s", key, element);
   }
   /*
    * test of entryArray
    */
   printf("===== test of entryArray\n");
   if ((array = (HMEntry **)hm_entryArray(hm, &n)) == NULL) {
      fprintf(stderr, "Error in invoking hm_entryArray()\n");
      return -1;
   }
   for (i = 0; i < n; i++) {
      printf("%s,%s", hmentry_key(array[i]), (char *)hmentry_value(array[i]));
   }
   free(array);
   /*
    * test of iterator
    */
   printf("===== test of iterator\n");
   if ((it = hm_it_create(hm)) == NULL) {
      fprintf(stderr, "Error in creating iterator\n");
      return -1;
   }
   while (it_hasNext(it)) {
      HMEntry *p;
      (void) it_next(it, (void **)&p);
      printf("%s,%s", hmentry_key(p), (char *)hmentry_value(p));
   }
   it_destroy(it);
   /*
    * test of destroy with free() as userFunction
    */
   printf("===== test of destroy(free)\n");
   hm_destroy(hm, free);

   return 0;
}
```

# Appendix E – TreeSet and TSTreeSet

## treeset.h

```
#ifndef _TREESET_H_
#define _TREESET_H_

/* BSD header removed to save space */

#include "iterator.h"

/*
 * interface definition for generic treeset implementation
 *
 * patterned roughly after Java 6 TreeSet generic class
 */

typedef struct treeset TreeSet;  /* opaque type definition */

/*
 * create a treeset that is ordered using `cmpFunction' to compare two elements
 *
 * returns a pointer to the treeset, or NULL if there are malloc() errors
 */
TreeSet *ts_create(int (*cmpFunction)(void *, void *));

/*
 * destroys the treeset; for each element, if userFunction != NULL,
 * it is invoked on that element; the storage associated with
 * the treeset is then returned to the heap
 */
void ts_destroy(TreeSet *ts, void (*userFunction)(void *element));

/*
 * adds the specified element to the set if it is not already present
 *
 * returns 1 if the element was added, 0 if the element was already present
 */
int ts_add(TreeSet *ts, void *element);

/*
 * returns the least element in the set greater than or equal to `element'
 *
 * returns 1 if found, or 0 if no such element
 */
int ts_ceiling(TreeSet *ts, void *element, void **ceiling);

/*
 * clears all elements from the treeset; for each element,
 * if userFunction != NULL, it is invoked on that element;
 * any storage associated with that element in the treeset is then
 * returned to the heap
 *
 * upon return, the treeset will be empty
 */
void ts_clear(TreeSet *ts, void (*userFunction)(void *element));

/*
 * returns 1 if the set contains the specified element, 0 if not
 */
int ts_contains(TreeSet *ts, void *element);

/*
 * returns the first (lowest) element currently in the set
 *
 * returns 1 if non-empty, 0 if empty
 */
```

```
int ts_first(TreeSet *ts, void **element);

/*
 * returns the greatest element in the set less than or equal to `element'
 *
 * returns 1 if found, or 0 if no such element
 */
int ts_floor(TreeSet *ts, void *element, void **floor);

/*
 * returns the least element in the set strictly greater than `element'
 *
 * returns 1 if found, or 0 if no such element
 */
int ts_higher(TreeSet *ts, void *element, void **higher);

/*
 * returns 1 if the set contains no elements, 0 otherwise
 */
int ts_isEmpty(TreeSet *ts);

/*
 * returns the last (highest) element currently in the set
 *
 * returns 1 if non-empty, 0 if empty
 */
int ts_last(TreeSet *ts, void **element);

/*
 * returns the greatest element in the set strictly less than `element'
 *
 * returns 1 if found, or 0 if no such element
 */
int ts_lower(TreeSet *ts, void *element, void **lower);

/*
 * retrieves and removes the first (lowest) element
 *
 * returns 0 if set was empty, 1 otherwise
 */
int ts_pollFirst(TreeSet *ts, void **element);

/*
 * retrieves and removes the last (highest) element
 *
 * returns 0 if set was empty, 1 otherwise
 */
int ts_pollLast(TreeSet *ts, void **element);

/*
 * removes the specified element from the set if present
 * if userFunction != NULL, invokes it on the element before removing it
 *
 * returns 1 if successful, 0 if not present
 */
int ts_remove(TreeSet *ts, void *element, void (*userFunction)(void *element));

/*
 * returns the number of elements in the treeset
 */
long ts_size(TreeSet *ts);

/*
 * return the elements of the treeset as an array of void * pointers
 * the order of elements in the array is the as determined by the treeset's
 * compare function
 *
 * returns pointer to the array or NULL if error
 * returns number of elements in the array in len
 */
void **ts_toArray(TreeSet *ts, long *len);

/*
 * create generic iterator to this treeset
 *
 * returns pointer to the Iterator or NULL if failure
 */
Iterator *ts_it_create(TreeSet *ts);

#endif /* _TREESET_H_ */
```

## tstreeset.h

```
#ifndef _TSTREESET_H_
#define _TSTREESET_H_

/* BSD header removed to save space */

#include "tsiterator.h"

/*
 * interface definition for thread-safe generic treeset implementation
 *
 * patterned roughly after Java 6 TreeSet generic class
 */

typedef struct tstreeset TSTreeSet;     /* opaque type definition */

/*
 * create a treeset that is ordered using `cmpFunction' to compare two elements
 *
 * returns a pointer to the treeset, or NULL if there are malloc() errors
 */
TSTreeSet *tsts_create(int (*cmpFunction)(void *, void *));

/*
 * destroys the treeset; for each element, if userFunction != NULL,
 * it is invoked on that element; the storage associated with
 * the treeset is then returned to the heap
 */
void tsts_destroy(TSTreeSet *ts, void (*userFunction)(void *));

/*
 * obtains the lock for exclusive access
 */
void tsts_lock(TSTreeSet *ts);

/*
 * returns the lock
 */
void tsts_unlock(TSTreeSet *ts);

/*
 * adds the specified element to the set if it is not already present
 *
 * returns 1 if the element was added, 0 if the element was already present
 */
int tsts_add(TSTreeSet *ts, void *element);

/*
 * returns the least element in the set greater than or equal to `element'
 *
 * returns 1 if found, or 0 if no such element
 */
int tsts_ceiling(TSTreeSet *ts, void *element, void **ceiling);

/*
```

```c
 * clears all elements from the treeset; for each element,
 * if userFunction != NULL, it is invoked on that element;
 * any storage associated with that element in the treeset is then
 * returned to the heap
 *
 * upon return, the treeset will be empty
 */
void tsts_clear(TSTreeSet *ts, void (*userFunction)(void *));

/*
 * returns 1 if the set contains the specified element, 0 if not
 */
int tsts_contains(TSTreeSet *ts, void *element);

/*
 * returns the first (lowest) element currently in the set
 *
 * returns 1 if non-empty, 0 if empty
 */
int tsts_first(TSTreeSet *ts, void **element);

/*
 * returns the greatest element in the set less than or equal to `element'
 *
 * returns 1 if found, or 0 if no such element
 */
int tsts_floor(TSTreeSet *ts, void *element, void **floor);

/*
 * returns the least element in the set strictly greater than `element'
 *
 * returns 1 if found, or 0 if no such element
 */
int tsts_higher(TSTreeSet *ts, void *element, void **higher);

/*
 * returns 1 if set is empty, 0 if it is not
 */
int tsts_isEmpty(TSTreeSet *ts);

/*
 * returns the last (highest) element currently in the set
 *
 * returns 1 if non-empty, 0 if empty
 */
int tsts_last(TSTreeSet *ts, void **element);

/*
 * returns the greatest element in the set strictly less than `element'
 *
 * returns 1 if found, or 0 if no such element
 */
int tsts_lower(TSTreeSet *ts, void *element, void **lower);

/*
 * retrieves and removes the first (lowest) element
 *
 * returns 0 if set was empty, 1 otherwise
 */
int tsts_pollFirst(TSTreeSet *ts, void **element);

/*
 * retrieves and removes the last (highest) element
 *
 * returns 0 if set was empty, 1 otherwise
 */
int tsts_pollLast(TSTreeSet *ts, void **element);
```

```
/*
 * removes the specified element from the set if present
 * if userFunction != NULL, invokes it on the element before removing it
 *
 * returns 1 if successful, 0 if not present
 */
int tsts_remove(TSTreeSet *ts, void *element, void (*userFunction)(void *));

/*
 * returns the number of elements in the treeset
 */
long tsts_size(TSTreeSet *ts);

/*
 * return the elements of the treeset as an array of void * pointers
 * the order of elements in the array is the as determined by the treeset's
 * compare function
 *
 * returns pointer to the array or NULL if error
 * returns number of elements in the array in len
 */
void **tsts_toArray(TSTreeSet *ts, long *len);

/*
 * create generic iterator to this treeset
 *
 * returns pointer to the Iterator or NULL if failure
 */
TSIterator *tsts_it_create(TSTreeSet *ts);

#endif /* _TSTREESET_H_ */
```

## treeset.c

```
/* BSD header removed to save space */

#include "treeset.h"
#include <stdlib.h>

/*
 * implementation for generic treeset implementation
 * implemented as an AVL tree
 */

typedef struct tnode {
   struct tnode *link[2];  /* 0 is left, 1 is right */
   void *element;
   int balance;                  /* difference between heights of l and r subs */
} TNode;

struct treeset {
   long size;
   TNode *root;
   int (*cmp)(void *,void *);
};

/*
 * structure needed for recursive population of array of pointers
 */
typedef struct popstruct {
   void **a;
   long len;
} PopStruct;

/*
 * routines used in rotations when rebalancing the tree
 */

/*
```

```
 * allocates a new node with the given element and NULL left and right links
 */
static TNode *newNode(void *element) {
   TNode *node = (TNode *)malloc(sizeof(TNode));

   if (node != NULL) {
      node->element = element;
      node->link[0] = node->link[1] = NULL;
      node->balance = 0;
   }
   return node;
}

static TNode *singleRotate(TNode *root, int dir) {
   TNode *save = root->link[!dir];

   root->link[!dir] = save->link[dir];
   save->link[dir] = root;
   return save;
}

static TNode *doubleRotate(TNode *root, int dir) {
   TNode *save = root->link[!dir]->link[dir];

   root->link[!dir]->link[dir] = save->link[!dir];
   save->link[!dir] = root->link[!dir];
   root->link[!dir] = save;
   save = root->link[!dir];
   root->link[!dir] = save->link[dir];
   save->link[dir] = root;
   return save;
}

static void adjustBalance(TNode *root, int dir, int bal) {
   TNode *n = root->link[dir];
   TNode *nn = n->link[!dir];

   if (nn->balance == 0)
      root->balance = n->balance = 0;
   else if (nn->balance == bal) {
      root->balance = -bal;
      n->balance = 0;
   } else {  /* nn->balance == -bal */
      root->balance = 0;
      n->balance = bal;
   }
   nn->balance = 0;
}

static TNode *insertBalance(TNode *root, int dir) {
   TNode *n = root->link[dir];
   int bal = (dir == 0) ? -1 : +1;

   if (n->balance == bal) {
      root->balance = n->balance = 0;
      root = singleRotate(root, !dir);
   } else {  /* n->balance == -bal */
      adjustBalance(root, dir, bal);
      root = doubleRotate(root, !dir);
   }
   return root;
}

static TNode *insert(TNode *root, void *element, int *done,
                     int (*cmp)(void*,void*)) {
   if (root == NULL)
      root = newNode(element);
   else {
      int dir = ((*cmp)(root->element, element) < 0);

      root->link[dir] = insert(root->link[dir], element, done, cmp);
      if (! *done) {
         root->balance += (dir == 0) ? -1 : +1;
         if (root->balance == 0)
            *done = 1;
         else if (abs(root->balance) > 1) {
            root = insertBalance(root, dir);
            *done = 1;
         }
      }
   }
   return root;
}

static TNode *removeBalance(TNode *root, int dir, int *done) {
   TNode *n = root->link[!dir];
   int bal = (dir == 0) ? -1 : +1;

   if (n->balance == -bal) {
      root->balance = n->balance = 0;
      root = singleRotate(root, dir);
   } else if (n->balance == bal) {
      adjustBalance(root, !dir, -bal);
      root = doubleRotate(root, dir);
   } else {  /* n->balance == 0 */
      root->balance = -bal;
      n->balance = bal;
      root = singleRotate(root, dir);
      *done = 1;
   }
   return root;
}

static TNode *remove(TNode *root, void *element, int *done,
                     int (*cmp)(void*,void*), void (*uf)(void*)) {
   if (root != NULL) {
      int dir;

      if ((*cmp)(element, root->element) == 0) {
         if (root->link[0] == NULL || root->link[1] == NULL) {
            TNode *save;

            dir = (root->link[0] == NULL);
            save = root->link[dir];
           if (uf != NULL)
               (*uf)(root->element);
            free(root);
            return save;
         } else {
            TNode *heir = root->link[0];

            while (heir->link[1] != NULL)
               heir = heir->link[1];
            root->element = heir->element;
            element = heir->element;
         }
      }
      dir = ((*cmp)(root->element, element) < 0);
      root->link[dir] = remove(root->link[dir], element, done, cmp, uf);
      if (! *done) {
         root->balance += (dir != 0) ? -1 : +1;
         if (abs(root->balance) == 1)
            *done = 1;
         else if (abs(root->balance) > 1)
            root = removeBalance(root, dir, done);
      }
   }
   return root;
}

/*
 * finds element in the set; returns null if it cannot be found
 */
static TNode *find(void *element, TNode *tree, int (*cmp)(void*,void*)) {
   int result;

   if (tree == NULL)
      return NULL;
   result = (*cmp)(element, tree->element);
   if (result < 0)
      return find(element, tree->link[0], cmp);
   else if (result > 0)
      return find(element, tree->link[1], cmp);
   else
      return tree;
}

/*
 * infix traversal to populate array of pointers
 */
static void populate(PopStruct *ps, TNode *node) {
   if (node != NULL) {
      populate(ps, node->link[0]);
      (ps->a)[ps->len++] = node->element;
      populate(ps, node->link[1]);
   }
}

TreeSet *ts_create(int (*cmpFunction)(void *, void *)) {
   TreeSet *ts = (TreeSet *)malloc(sizeof(TreeSet));

   if (ts != NULL) {
      ts->size = 0L;
      ts->root = NULL;
      ts->cmp = cmpFunction;
   }
   return ts;
}

/*
 * postorder traversal, invoking userFunction and then freeing node
 */
static void postpurge(TNode *leaf, void (*userFunction)(void *element)) {
   if (leaf != NULL) {
      postpurge(leaf->link[0], userFunction);
      postpurge(leaf->link[1], userFunction);
      if (userFunction != NULL)
         (*userFunction)(leaf->element);
      free(leaf);
   }
}

void ts_destroy(TreeSet *ts, void (*userFunction)(void *element)) {
   postpurge(ts->root, userFunction);
   free(ts);
}

int ts_add(TreeSet *ts, void *element) {
   int done = 0;

   if (find(element, ts->root, ts->cmp) != NULL)
      return 0;
   ts->root = insert(ts->root, element, &done, ts->cmp);
   ts->size++;
   return 1;
}

static TNode *Min(TNode *n1, TNode *n2, int (*cmp)(void*,void*)) {
   TNode *ans = n1;
   if (n1 == NULL)
      return n2;
   if (n2 == NULL)
      return n1;
   if ((*cmp)(n1->element, n2->element) > 0)
      ans = n2;
   return ans;
}

static TNode *Max(TNode *n1, TNode *n2, int (*cmp)(void*,void*)) {
   TNode *ans = n1;
   if (n1 == NULL)
      return n2;
   if (n2 == NULL)
      return n1;
   if ((*cmp)(n1->element, n2->element) < 0)
      ans = n2;
   return ans;
}

int ts_ceiling(TreeSet *ts, void *element, void **ceiling) {
   TNode *t = ts->root;
   TNode *current = NULL;

   while (t != NULL) {
      int cmp = (*ts->cmp)(element, t->element);
      if (cmp == 0) {
         current = t;
        break;
      } else if (cmp < 0) {
         current = Min(t, current, ts->cmp);
        t = t->link[0];
      } else {
         t = t->link[1];
      }
   }
   if (current == NULL)
      return 0;
   *ceiling = current->element;
   return 1;
}

void ts_clear(TreeSet *ts, void (*userFunction)(void *element)) {
   postpurge(ts->root, userFunction);
   ts->root = NULL;
   ts->size = 0L;
}

int ts_contains(TreeSet *ts, void *element) {
   return (find(element, ts->root, ts->cmp) != NULL);
}

/*
 * find node with minimum value in subtree
 */
TNode *findMin(TNode *tree) {
   if (tree != NULL)
      while (tree->link[0] != NULL)
         tree = tree->link[0];
   return tree;
}

int ts_first(TreeSet *ts, void **element) {
   TNode *current = findMin(ts->root);

   if (current == NULL)
      return 0;
   *element = current->element;
   return 1;
}

int ts_floor(TreeSet *ts, void *element, void **floor) {
   TNode *t = ts->root;
   TNode *current = NULL;

   while (t != NULL) {
      int cmp = (*ts->cmp)(element, t->element);
      if (cmp == 0) {
         current = t;
        break;
      } else if (cmp > 0) {
         current = Max(t, current, ts->cmp);
        t = t->link[1];
      } else {
         t = t->link[0];
      }
   }
   if (current == NULL)
      return 0;
   *floor = current->element;
   return 1;
}

int ts_higher(TreeSet *ts, void *element, void **higher) {
   TNode *t = ts->root;
   TNode *current = NULL;

   while (t != NULL) {
      int cmp = (*ts->cmp)(element, t->element);
      if (cmp < 0) {
         current = Min(t, current, ts->cmp);
        t = t->link[0];
      } else {
         t = t->link[1];
      }
   }
   if (current == NULL)
      return 0;
   *higher = current->element;
   return 1;
}

int ts_isEmpty(TreeSet *ts) {
   return (ts->size == 0L);
}

/*
 * find node with maximum value in subtree
 */
TNode *findMax(TNode *tree) {
   if (tree != NULL)
      while (tree->link[1] != NULL)
         tree = tree->link[1];
   return tree;
}

int ts_last(TreeSet *ts, void **element) {
   TNode *current = findMax(ts->root);

   if (current == NULL)
      return 0;
   *element = current->element;
   return 1;
}

int ts_lower(TreeSet *ts, void *element, void **lower) {
   TNode *t = ts->root;
   TNode *current = NULL;

   while (t != NULL) {
      int cmp = (*ts->cmp)(element, t->element);
      if (cmp > 0) {
         current = Max(t, current, ts->cmp);
        t = t->link[1];
      } else {
         t = t->link[0];
      }
   }
   if (current == NULL)
      return 0;
   *lower = current->element;
   return 1;
}

int ts_pollFirst(TreeSet *ts, void **element) {
   TNode *node = findMin(ts->root);
   int done = 0;

   if (node == NULL)
      return 0;
   *element = node->element;
   ts->root = remove(ts->root, node->element, &done, ts->cmp, NULL);
   return 1;
}

int ts_pollLast(TreeSet *ts, void **element) {
   TNode *node = findMax(ts->root);
   int done = 0;

   if (node == NULL)
      return 0;
   *element = node->element;
   ts->root = remove(ts->root, node->element, &done, ts->cmp, NULL);
   return 1;
}

int ts_remove(TreeSet *ts, void *element, void (*userFunction)(void *element)) {
   int done = 0;

   if (find(element, ts->root, ts->cmp) == NULL)
      return 0;
   ts->root = remove(ts->root, element, &done, ts->cmp, userFunction);
   ts->size--;
   return 1;
}

long ts_size(TreeSet *ts) {
   return ts->size;
}

/*
 * generates an array of void * pointers on the heap and copies
 * tree elements into the array
 *
 * returns pointer to array or NULL if malloc failure
 */
static void **genArray(TreeSet *ts) {
   void **tmp = NULL;
   PopStruct ps;
   if (ts->size > 0L) {
      size_t nbytes = ts->size * sizeof(void *);
      tmp = (void **)malloc(nbytes);
      if (tmp != NULL) {
         ps.a = tmp;
        ps.len = 0;
        populate(&ps, ts->root);
      }
   }
   return tmp;
}

void **ts_toArray(TreeSet *ts, long *len) {
   void **array = genArray(ts);

   if (array != NULL)
      *len = ts->size;
   return array;
}

Iterator *ts_it_create(TreeSet *ts) {
   Iterator *it = NULL;
   void **tmp = genArray(ts);

   if (tmp != NULL) {
      it = it_create(ts->size, tmp);
      if (it == NULL)
         free(tmp);
   }
   return it;
}
```

### tstreeset.c

```
/* BSD header removed to save space */

#include "tstreeset.h"
#include "treeset.h"
#include <stdlib.h>
#include <pthread.h>

/*
 * implementation for generic thread-safe treeset implementation
 */

#define LOCK(ts) &((ts)->lock)

struct tstreeset {
   TreeSet *ts;
   pthread_mutex_t lock;
};

TSTreeSet *tsts_create(int (*cmpFunction)(void *, void *)) {
   TSTreeSet *tsts = (TSTreeSet *)malloc(sizeof(TSTreeSet));

   if (tsts != NULL) {
      TreeSet *ts = ts_create(cmpFunction);
      if (ts == NULL) {
         free(tsts);
        tsts = NULL;
      } else {
         pthread_mutexattr_t ma;
        pthread_mutexattr_init(&ma);
        pthread_mutexattr_settype(&ma, PTHREAD_MUTEX_RECURSIVE);
        tsts->ts = ts;
        pthread_mutex_init(LOCK(tsts), &ma);
        pthread_mutexattr_destroy(&ma);
      }
   }
   return tsts;
}

void tsts_destroy(TSTreeSet *ts, void (*userFunction)(void *element)) {
   pthread_mutex_lock(LOCK(ts));
   ts_destroy(ts->ts, userFunction);
   pthread_mutex_unlock(LOCK(ts));
   pthread_mutex_destroy(LOCK(ts));
   free(ts);
}

void tsts_lock(TSTreeSet *ts) {
   pthread_mutex_lock(LOCK(ts));
}

void tsts_unlock(TSTreeSet *ts) {
   pthread_mutex_unlock(LOCK(ts));
}

int tsts_add(TSTreeSet *ts, void *element) {
   int result;
   pthread_mutex_lock(LOCK(ts));
   result = ts_add(ts->ts, element);
   pthread_mutex_unlock(LOCK(ts));
   return result;
}

int tsts_ceiling(TSTreeSet *ts, void *element, void **ceiling) {
   int result;
   pthread_mutex_lock(LOCK(ts));
   result = ts_ceiling(ts->ts, element, ceiling);
   pthread_mutex_unlock(LOCK(ts));
   return result;
}

void tsts_clear(TSTreeSet *ts, void (*userFunction)(void *element)) {
   pthread_mutex_lock(LOCK(ts));
   ts_clear(ts->ts, userFunction);
   pthread_mutex_unlock(LOCK(ts));
}

int tsts_contains(TSTreeSet *ts, void *element) {
   int result;
   pthread_mutex_lock(LOCK(ts));
   result = ts_contains(ts->ts, element);
   pthread_mutex_unlock(LOCK(ts));
   return result;
}

int tsts_first(TSTreeSet *ts, void **element) {
   int result;
   pthread_mutex_lock(LOCK(ts));
   result = ts_first(ts->ts, element);
   pthread_mutex_unlock(LOCK(ts));
   return result;
}

int tsts_floor(TSTreeSet *ts, void *element, void **floor) {
   int result;
   pthread_mutex_lock(LOCK(ts));
   result = ts_floor(ts->ts, element, floor);
   pthread_mutex_unlock(LOCK(ts));
   return result;
}

int tsts_higher(TSTreeSet *ts, void *element, void **higher) {
   int result;
   pthread_mutex_lock(LOCK(ts));
   result = ts_higher(ts->ts, element, higher);
   pthread_mutex_unlock(LOCK(ts));
   return result;
}

int tsts_isEmpty(TSTreeSet *ts) {
   int result;
   pthread_mutex_lock(LOCK(ts));
   result = ts_isEmpty(ts->ts);
   pthread_mutex_unlock(LOCK(ts));
   return result;
}

int tsts_last(TSTreeSet *ts, void **element) {
   int result;
   pthread_mutex_lock(LOCK(ts));
   result = ts_last(ts->ts, element);
   pthread_mutex_unlock(LOCK(ts));
   return result;
}

int tsts_lower(TSTreeSet *ts, void *element, void **lower) {
   int result;
   pthread_mutex_lock(LOCK(ts));
   result = ts_lower(ts->ts, element, lower);
   pthread_mutex_unlock(LOCK(ts));
   return result;
}

int tsts_pollFirst(TSTreeSet *ts, void **element) {
   int result;
   pthread_mutex_lock(LOCK(ts));
   result = ts_pollFirst(ts->ts, element);
   pthread_mutex_unlock(LOCK(ts));
   return result;
}

int tsts_pollLast(TSTreeSet *ts, void **element) {
   int result;
   pthread_mutex_lock(LOCK(ts));
   result = ts_pollLast(ts->ts, element);
   pthread_mutex_unlock(LOCK(ts));
   return result;
}

int tsts_remove(TSTreeSet *ts, void *element, void (*userFunction)(void *element))
{
   int result;
   pthread_mutex_lock(LOCK(ts));
   result = ts_remove(ts->ts, element, userFunction);
   pthread_mutex_unlock(LOCK(ts));
   return result;
}

long tsts_size(TSTreeSet *ts) {
   long result;
   pthread_mutex_lock(LOCK(ts));
   result = ts_size(ts->ts);
   pthread_mutex_unlock(LOCK(ts));
   return result;
}

void **tsts_toArray(TSTreeSet *ts, long *len) {
   void **result;
   pthread_mutex_lock(LOCK(ts));
   result = ts_toArray(ts->ts, len);
   pthread_mutex_unlock(LOCK(ts));
   return result;
```

```
}

TSIterator *tsts_it_create(TSTreeSet *ts) {
   TSIterator *it = NULL;
   void **tmp;
   long len;

   pthread_mutex_lock(LOCK(ts));
   tmp = ts_toArray(ts->ts, &len);
   if (tmp != NULL) {
      it = tsit_create(LOCK(ts), len, tmp);
      if (it == NULL)
         free(tmp);
   }
   if (it == NULL)
      pthread_mutex_unlock(LOCK(ts));
   return it;
}
```

## tstest.c (you can create your own tststest.c)

```
/* BSD header removed to save space */

#include "treeset.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

static int scmp(void *a, void *b) {
   return strcmp((char *)a, (char *)b);
}

int main(int argc, char *argv[]) {
   char buf[1024];
   char *p;
   TreeSet *ts;
   long i, n;
   FILE *fd;
   Iterator *it;
   void **array;

   if (argc != 2) {
      fprintf(stderr, "usage: ./tstest file\n");
      return -1;
   }
   if ((ts = ts_create(scmp)) == NULL) {
      fprintf(stderr, "Error creating treeset of strings\n");
      return -1;
   }
   if ((fd = fopen(argv[1], "r")) == NULL) {
      fprintf(stderr, "Unable to open %s to read\n", argv[1]);
      return -1;
   }
   /*
    * test of add()
    */
   printf("===== test of add\n");
   i = 0;
   while (fgets(buf, 1024, fd) != NULL) {
      p = strchr(buf, '\n');
      *p = '\0';
      if ((p = strdup(buf)) == NULL) {
         fprintf(stderr, "Error duplicating string\n");
         return -1;
      }
      if (!ts_add(ts, p)) {
         fprintf(stderr, "Duplicate line: \"%s\"\n", p);
        free(p);
      }
   }
   fclose(fd);
   n = ts_size(ts);
   /*
    * test of get()
    */
   printf("===== test of first and remove\n");
   printf("Size before remove = %ld\n", n);
   for (i = 0; i < n; i++) {
      char *element;

      if (!ts_first(ts, (void **)&element)) {
         fprintf(stderr, "Error retrieving %ld'th element\n", i);
         return -1;
      }
      printf("%s\n", element);
      if (!ts_remove(ts, element, free)) {
         fprintf(stderr, "Error removing %ld'th element\n", i);
        return -1;
      }
   }
   printf("Size after remove = %ld\n", ts_size(ts));
   /*
    * test of destroy with NULL userFunction
    */
   printf("===== test of destroy(NULL)\n");
   ts_destroy(ts, NULL);
   /*
    * test of insert
    */
   if ((ts = ts_create(scmp)) == NULL) {
      fprintf(stderr, "Error creating treeset of strings\n");
      return -1;
   }
   fd = fopen(argv[1], "r");            /* we know we can open it */
   i = 0L;
   while (fgets(buf, 1024, fd) != NULL) {
      p = strchr(buf, '\n');
      *p = '\0';
      if ((p = strdup(buf)) == NULL) {
         fprintf(stderr, "Error duplicating string\n");
         return -1;
      }
      if (!ts_add(ts, p)) {
         free(p);
      }
   }
   fclose(fd);
   /*
    * test of toArray
    */
   printf("===== test of toArray\n");
   if ((array = ts_toArray(ts, &n)) == NULL) {
      fprintf(stderr, "Error in invoking ts_toArray()\n");
      return -1;
   }
   for (i = 0; i < n; i++) {
      printf("%s\n", (char *)array[i]);
   }
   free(array);
   /*
    * test of iterator
    */
   printf("===== test of iterator\n");
   if ((it = ts_it_create(ts)) == NULL) {
      fprintf(stderr, "Error in creating iterator\n");
      return -1;
   }
```

```
    while (it_hasNext(it)) {
       char *p;
       (void) it_next(it, (void **)&p);
       printf("%s\n", p);
    }
    it_destroy(it);
    /*
     * test of ceiling, floor, higher, lower
     */
    if (!ts_ceiling(ts, "0005", (void **)&p)) {
       fprintf(stderr, "No ceiling found relative to \"0005\"\n");
    } else
       printf("Ceiling relative to \"0005\" is \"%s\"\n", p);
    if (!ts_higher(ts, "0006", (void **)&p)) {
       fprintf(stderr, "No higher found relative to \"0006\"\n");
    } else
       printf("Higher relative to \"0006\" is \"%s\"\n", p);
    if (!ts_floor(ts, "0005", (void **)&p)) {
       fprintf(stderr, "No floor found relative to \"0005\"\n");
    } else
       printf("Floor relative to \"0005\" is \"%s\"\n", p);
    if (!ts_lower(ts, "0006", (void **)&p)) {
       fprintf(stderr, "No lower found relative to \"0006\"\n");
    } else
       printf("Lower relative to \"0006\" is \"%s\"\n", p);
    /*
     * test of pollFirst and pollLast
     */
    n = ts_size(ts) / 4;
    printf("===== test of pollFirst - first %ld elements of the set are\n", n);
    for (i = 0; i < n; i++) {
       char *p;
       (void) ts_first(ts, (void **)&p);
       printf("First element is: \"%s\"\n", p);
       (void) ts_last(ts, (void **)&p);
       printf("Last element is: \"%s\"\n", p);
       if (!ts_pollFirst(ts, (void **)&p)) {
          fprintf(stderr, "Error invoking pollFirst()\n");
         return -1;
       }
       printf("%s\n", p);
       free(p);
    }
    printf("===== test of pollLast - last %ld elements of the set are\n", n);
    for (i = 0; i < n; i++) {
       char *p;
       (void) ts_first(ts, (void **)&p);
       printf("First element is: \"%s\"\n", p);
       (void) ts_last(ts, (void **)&p);
       printf("Last element is: \"%s\"\n", p);
       if (!ts_pollLast(ts, (void **)&p)) {
          fprintf(stderr, "Error invoking pollLast()\n");
         return -1;
       }
       printf("%s\n", p);
       free(p);
    }
    /*
     * test of destroy with free() as userFunction
     */
    printf("===== test of destroy(free)\n");
    ts_destroy(ts, free);

    return 0;
}
```

# Appendix F – Stack and TSStack

## stack.h

```c
#ifndef _STACK_H_
#define _STACK_H_

/* BSD header removed to save space */

#include "iterator.h"

/*
 * interface definition for generic stack implementation
 *
 * patterned roughly after Java 6 Stack generic class
 */

typedef struct stack Stack;      /* opaque type definition */

/*
 * create an stack with the specified capacity; if capacity == 0, a
 * default initial capacity (50 elements) is used
 *
 * returns a pointer to the stack, or NULL if there are malloc() errors
 */
Stack *stack_create(long capacity);

/*
 * destroys the stack; for each occupied position, if freeFxn != NULL,
 * it is invoked on the element at that position; the storage associated with
 * the stack is then returned to the heap
 */
void stack_destroy(Stack *st, void (*freeFxn)(void *element));

/*
 * purges all elements from the stack; for each occupied position,
 * if freeFxn != NULL, it is invoked on the element at that position;
 * any storage associated with the element in the stack is then
 * returned to the heap
 *
 * upon return, the stack will be empty
 */
void stack_purge(Stack *st, void (*freeFxn)(void *element));

/*
 * pushes `element' onto the stack; if no more room in the stack, it is
 * dynamically resized
 *
 * returns 1 if successful, 0 if unsuccessful (malloc errors)
 */
int stack_push(Stack *st, void *element);

/*
 * pops the element at the top of the stack into `*element'
 *
 * returns 1 if successful, 0 if stack was empty
 */
int stack_pop(Stack *st, void **element);

/*
 * peeks at the top element of the stack without removing it;
 * returned in `*element'
 *
 * returns 1 if successful, 0 i stack was empty
 */
int stack_peek(Stack *st, void **element);
```

```
/*
 * returns 1 if stack is empty, 0 if it is not
 */
int stack_isEmpty(Stack *st);

/*
 * returns the number of elements in the stack
 */
long stack_size(Stack *st);

/*
 * returns an array containing all of the elements of the stack in
 * proper sequence (from top to bottom element); returns the length of
 * the list in `len'
 *
 * returns pointer to void * array of elements, or NULL if malloc failure
 */
void **stack_toArray(Stack *st, long *len);

/*
 * create generic iterator to this stack;
 * successive next calls return elements in proper sequence (top to bottom)
 *
 * returns pointer to the Iterator or NULL if failure
 */
Iterator *stack_it_create(Stack *st);

#endif /* _STACK_H_ */
```

## tsstack.h

```
/* BSD header removed to save space */
#ifndef _TSSTACK
#define _TSSTACK

/* BSD header removed to save space */

#include "tsiterator.h"

/*
 * interface definition for generic type-safe stack implementation
 *
 * patterned roughly after Java 6 Stack generic class
 */

typedef struct tsstack TSStack;  /* opaque type definition */

/*
 * create an stack with the specified capacity; if capacity == 0, a
 * default initial capacity (50 elements) is used
 *
 * returns a pointer to the stack, or NULL if there are malloc() errors
 */
TSStack *tsstack_create(long capacity);

/*
 * destroys the stack; for each occupied position, if freeFxn != NULL,
 * it is invoked on the element at that position; the storage associated with
 * the stack is then returned to the heap
 */
void tsstack_destroy(TSStack *st, void (*freeFxn)(void *element));

/*
 * purges all elements from the stack; for each occupied position,
 * if freeFxn != NULL, it is invoked on the element at that position;
 * any storage associated with the element in the stack is then
 * returned to the heap
 *
 * upon return, the stack will be empty
 */
void tsstack_purge(TSStack *st, void (*freeFxn)(void *element));

/*
 * obtains the lock for exclusive access
 */
void tsstack_lock(TSStack *st);

/*
 * returns the lock
 */
void tsstack_unlock(TSStack *st);

/*
 * pushes `element' onto the stack; if no more room in the stack, it is
 * dynamically resized
 *
 * returns 1 if successful, 0 if unsuccessful (malloc errors)
 */
int tsstack_push(TSStack *st, void *element);

/*
 * pops the element at the top of the stack into `*element'
 *
 * returns 1 if successful, 0 if stack was empty
 */
int tsstack_pop(TSStack *st, void **element);

/*
 * peeks at the top element of the stack without removing it;
 * returned in `*element'
 *
 * returns 1 if successful, 0 i stack was empty
 */
int tsstack_peek(TSStack *st, void **element);

/*
 * returns 1 if stack is empty, 0 if it is not
 */
int tsstack_isEmpty(TSStack *st);

/*
 * returns the number of elements in the stack
 */
long tsstack_size(TSStack *st);

/*
 * returns an array containing all of the elements of the stack in
 * proper sequence (from top to bottom element); returns the length of
 * the list in `len'
 *
 * returns pointer to void * array of elements, or NULL if malloc failure
 */
void **tsstack_toArray(TSStack *st, long *len);

/*
 * create generic iterator to this stack;
 * successive next calls return elements in proper sequence (top to bottom)
 *
 * returns pointer to the TSIterator or NULL if failure
 */
TSIterator *tsstack_it_create(TSStack *st);

#endif /* _TSSTACK */
```

### stack.c

```
/* BSD header removed to save space */
```

```
/*
 * implementation for generic stack
 */

#include "stack.h"
#include <stdlib.h>

#define DEFAULT_CAPACITY 50L
#define MAX_INIT_CAPACITY 1000L

struct stack {
   long capacity;
   long delta;
   long next;
   void **theArray;
};

Stack *stack_create(long capacity) {
   Stack *st = (Stack *)malloc(sizeof(Stack));

   if (st != NULL) {
      long cap;
      void **array = NULL;

      cap = (capacity <= 0) ? DEFAULT_CAPACITY : capacity;
      cap = (cap > MAX_INIT_CAPACITY) ? MAX_INIT_CAPACITY : cap;
      array = (void **) malloc(cap * sizeof(void *));
      if (array == NULL) {
         free(st);
        st = NULL;
      } else {
         st->capacity = cap;
        st->delta = cap;
        st->next = 0L;
        st->theArray = array;
      }
   }
   return st;
}

/*
 * traverses stack, calling freeFxn on each element
 */
static void purge(Stack *st, void (*freeFxn)(void*)) {

   if (freeFxn != NULL) {
      long i;

      for (i = 0L; i < st->next; i++)
         (*freeFxn)(st->theArray[i]); /* user frees element storage */
   }
}

void stack_destroy(Stack *st, void (*freeFxn)(void*)) {
   purge(st, freeFxn);
   free(st->theArray);                    /* we free array of pointers */
   free(st);                       /* we free the Stack struct */
}

void stack_purge(Stack *st, void (*freeFxn)(void*)){
   purge(st, freeFxn);
   st->next = 0L;
}

int stack_push(Stack *st, void *element) {
   int status = 1;

   if (st->capacity <= st->next) {      /* need to reallocate */
```

```
      size_t nbytes = (st->capacity + st->delta) * sizeof(void *);
      void **tmp = (void **)realloc(st->theArray, nbytes);
      if (tmp == NULL)
         status = 0;       /* allocation failure */
      else {
         st->theArray = tmp;
        st->capacity += st->delta;
      }
   }
   if (status)
      st->theArray[st->next++] = element;
   return status;
}

int stack_pop(Stack *st, void **element) {
   int status = 0;

   if (st->next > 0L) {
      *element = st->theArray[--st->next];
      status = 1;
   }
   return status;
}

int stack_peek(Stack *st, void **element) {
   int status = 0;

   if (st->next > 0L) {
      *element = st->theArray[st->next - 1];
      status = 1;
   }
   return status;
}

int stack_isEmpty(Stack *st) {
   return (st->next == 0L);
}

long stack_size(Stack *st) {
   return st->next;
}

/*
 * local function that duplicates the array of void * pointers on the heap
 *
 * returns pointer to duplicate array or NULL if malloc failure
 */
static void **arraydupl(Stack *st) {
   void **tmp = NULL;
   if (st->next > 0L) {
      size_t nbytes = st->next * sizeof(void *);
      tmp = (void **)malloc(nbytes);
      if (tmp != NULL) {
         long i;

        for (i = 0; i < st->next; i++)
            tmp[i] = st->theArray[i];
      }
   }
   return tmp;
}

void **stack_toArray(Stack *st, long *len) {
   void **tmp = arraydupl(st);

   if (tmp != NULL)
        *len = st->next;
   return tmp;
```

```c
}

Iterator *stack_it_create(Stack *st) {
   Iterator *it = NULL;
   void **tmp = arraydupl(st);

   if (tmp != NULL) {
      it = it_create(st->next, tmp);
      if (it == NULL)
         free(tmp);
   }
   return it;
}
```

## tsstack.c

```c
/* BSD header removed to save space */

#include "tsstack.h"
#include "stack.h"
#include <stdlib.h>
#include <pthread.h>

#define LOCK(st) &((st)->lock)

/*
 * implementation for thread-safe generic stack implementation
 */

struct tsstack {
   Stack *st;
   pthread_mutex_t lock;   /* this is a recursive lock */
};

TSStack *tsstack_create(long capacity) {
   TSStack *tsst = (TSStack *)malloc(sizeof(TSStack));

   if (tsst != NULL) {
      Stack *st = stack_create(capacity);

      if (st == NULL) {
         free(tsst);
        tsst = NULL;
      } else {
         pthread_mutexattr_t ma;
        pthread_mutexattr_init(&ma);
         pthread_mutexattr_settype(&ma, PTHREAD_MUTEX_RECURSIVE);
         tsst->st = st;
         pthread_mutex_init(LOCK(tsst), &ma);
         pthread_mutexattr_destroy(&ma);
      }
   }
   return tsst;
}

void tsstack_destroy(TSStack *st, void (*freeFxn)(void*)) {
   pthread_mutex_lock(LOCK(st));
   stack_destroy(st->st, freeFxn);
   pthread_mutex_unlock(LOCK(st));
   pthread_mutex_destroy(LOCK(st));
   free(st);
}

void tsstack_purge(TSStack *st, void (*freeFxn)(void*)) {
   pthread_mutex_lock(LOCK(st));
   stack_purge(st->st, freeFxn);
   pthread_mutex_unlock(LOCK(st));
}
```

```
void tsstack_lock(TSStack *st) {
   pthread_mutex_lock(LOCK(st));
}

void tsstack_unlock(TSStack *st) {
   pthread_mutex_unlock(LOCK(st));
}

int tsstack_push(TSStack *st, void *element) {
   int result;
   pthread_mutex_lock(LOCK(st));
   result = stack_push(st->st, element);
   pthread_mutex_unlock(LOCK(st));
   return result;
}

int tsstack_pop(TSStack *st, void **element) {
   int result;
   pthread_mutex_lock(LOCK(st));
   result = stack_pop(st->st, element);
   pthread_mutex_unlock(LOCK(st));
   return result;
}

int tsstack_peek(TSStack *st, void **element) {
   int result;
   pthread_mutex_lock(LOCK(st));
   result = stack_peek(st->st, element);
   pthread_mutex_unlock(LOCK(st));
   return result;
}

int tsstack_isEmpty(TSStack *st) {
   int result;
   pthread_mutex_lock(LOCK(st));
   result = stack_isEmpty(st->st);
   pthread_mutex_unlock(LOCK(st));
   return result;
}

long tsstack_size(TSStack *st) {
   long result;
   pthread_mutex_lock(LOCK(st));
   result = stack_size(st->st);
   pthread_mutex_unlock(LOCK(st));
   return result;
}

void **tsstack_toArray(TSStack *st, long *len) {
   void **result;
   pthread_mutex_lock(LOCK(st));
   result = stack_toArray(st->st, len);
   pthread_mutex_unlock(LOCK(st));
   return result;
}

TSIterator *tsstack_it_create(TSStack *st) {
   TSIterator *it = NULL;
   void **tmp;
   long len;

   pthread_mutex_lock(LOCK(st));
   tmp = stack_toArray(st->st, &len);
   if (tmp != NULL) {
      it = tsit_create(LOCK(st), len, tmp);
      if (it == NULL)
         free(tmp);
   }
   if (it == NULL)
      pthread_mutex_unlock(LOCK(st));
   return it;
}
```

## sttest.c (you can create your own tssttest.c)

```
/* BSD header removed to save space */

#include "stack.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
   char buf[1024];
   char *p;
   Stack *st;
   long i, n;
   FILE *fd;
   char **array;
   Iterator *it;

   if (argc != 2) {
      fprintf(stderr, "usage: ./sttest file\n");
   return -1;
   }
   if ((st = stack_create(0L)) == NULL) {
      fprintf(stderr, "Error creating stack of strings\n");
   return -1;
   }
   if ((fd = fopen(argv[1], "r")) == NULL) {
      fprintf(stderr, "Unable to open %s to read\n", argv[1]);
   return -1;
   }
   /*
    * test of push()
    */
   printf("===== test of push\n");
   while (fgets(buf, 1024, fd) != NULL) {
      if ((p = strdup(buf)) == NULL) {
         fprintf(stderr, "Error duplicating string\n");
         return -1;
      }
      if (!stack_push(st, p)) {
         fprintf(stderr, "Error pushing string to stack\n");
         return -1;
      }
   }
   fclose(fd);
   n = stack_size(st);
   /*
    * test of pop()
    */
   printf("===== test of pop\n");
   for (i = 0; i < n; i++) {
      if (!stack_pop(st, (void **)&p)) {
         fprintf(stderr, "Error retrieving %ld'th element\n", i);
         return -1;
      }
      printf("%s", p);
      free(p);
   }
   printf("===== test of destroy(NULL)\n");
   /*
    * test of destroy with NULL freeFxn
    */
   stack_destroy(st, NULL);
   if ((st = stack_create(0L)) == NULL) {
      fprintf(stderr, "Error creating stack of strings\n");
      return -1;
   }
   fd = fopen(argv[1], "r");            /* we know we can open it */
   while (fgets(buf, 1024, fd) != NULL) {
      if ((p = strdup(buf)) == NULL) {
         fprintf(stderr, "Error duplicating string\n");
         return -1;
      }
      if (!stack_push(st, p)) {
         fprintf(stderr, "Error pushing string to stack\n");
         return -1;
      }
   }
   fclose(fd);
   printf("===== test of toArray\n");
   /*
    * test of toArray
    */
   if ((array = (char **)stack_toArray(st, &n)) == NULL) {
      fprintf(stderr, "Error in invoking stack_toArray()\n");
      return -1;
   }
   for (i = 0; i < n; i++) {
      printf("%s", array[i]);
   }
   free(array);
   printf("===== test of iterator\n");
   /*
    * test of iterator
    */
   if ((it = stack_it_create(st)) == NULL) {
      fprintf(stderr, "Error in creating iterator\n");
      return -1;
   }
   while (it_hasNext(it)) {
      char *p;
      (void) it_next(it, (void **)&p);
      printf("%s", p);
   }
   it_destroy(it);
   printf("===== test of destroy(free)\n");
   /*
    * test of destroy with free() as freeFxn
    */
   stack_destroy(st, free);

   return 0;
}
```