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]](#footnote-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);**

**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 simply 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 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 thread-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):

1. Iterator and TSIterator – these are generic iterators that can be used to traverse a library ADT.
2. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. BQueue and TSBQueue – provides most of the functionality found in the Java Queue generic interface; the queues are bounded FIFO; natural order for iteration is head to tail.
8. UQueue and TSUQueue – provides most of the functionality found in the Java Queue generic interface; the queues are unbounded FIFO; natural order for iteration is head to tail.

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);

/\*

\* 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

#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

#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;

}

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

/\* 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;

}

# Appendix G – BQueue and TSBQueue

## bqueue.h

#ifndef \_BQUEUE\_H\_

#define \_BQUEUE\_H\_

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

/\*

\* interface definition for generic bounded FIFO queue

\*

\* patterned roughly after Java 6 Queue interface

\*/

#include "iterator.h"

#define DEFAULT\_CAPACITY 25L

#define MAX\_CAPACITY 10240L

typedef struct bqueue BQueue; /\* opaque type definition \*/

/\*

\* create a bounded queue; if capacity is 0L, give it a default capacity

\*

\* returns a pointer to the queue, or NULL if there are malloc() errors

\*/

BQueue \*bq\_create(long capacity);

/\*

\* destroys the bounded queue; for each element, if userFunction != NULL,

\* invokes userFunction on the element; then returns any queue structure

\* associated with the element; finally, deletes any remaining structures

\* associated with the queue

\*/

void bq\_destroy(BQueue \*bq, void (\*userFunction)(void \*element));

/\*

\* clears the queue; for each element, if userFunction != NULL, invokes

\* userFunction on the element; then returns any queue structure associated with

\* the element

\*

\* upon return, the queue is empty

\*/

void bq\_clear(BQueue \*bq, void (\*userFunction)(void \*element));

/\*

\* appends `element' to the end of the bounded queue

\*

\* returns 1 if successful, 0 if unsuccesful (queue is full)

\*/

int bq\_add(BQueue \*bq, void \*element);

/\*

\* retrieves, but does not remove, the head of the queue

\*

\* returns 1 if successful, 0 if unsuccessful (queue is empty)

\*/

int bq\_peek(BQueue \*bq, void \*\*element);

/\*

\* Retrieves, and removes, the head of the queue

\*

\* return 1 if successful, 0 if not (queue is empty)

\*/

int bq\_remove(BQueue \*bq, void \*\*element);

/\*

\* returns the number of elements in the queue

\*/

long bq\_size(BQueue \*bq);

/\*

\* returns true if the queue is empty, false if not

\*/

int bq\_isEmpty(BQueue \*bq);

/\*

\* returns an array containing all of the elements of the queue in

\* proper sequence (from first to last element); returns the length of the

\* queue in `len'

\*

\* returns pointer to void \* array of elements, or NULL if malloc failure

\*/

void \*\*bq\_toArray(BQueue \*bq, long \*len);

/\*

\* creates an iterator for running through the queue

\*

\* returns pointer to the Iterator or NULL

\*/

Iterator \*bq\_it\_create(BQueue \*bq);

#endif /\* \_BQUEUE\_H\_ \*/

## tsbqueue.h

#ifndef \_TSBQUEUE\_H\_

#define \_TSBQUEUE\_H\_

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

/\*

\* interface definition for generic, thread-safe bounded FIFO queue

\*

\* patterned roughly after Java 6 Queue interface

\*/

#include "tsiterator.h"

typedef struct tsbqueue TSBQueue; /\* opaque type definition \*/

/\*

\* create an bounded queue; if capacity is 0L, give it a default capacity

\*

\* returns a pointer to the queue, or NULL if there are malloc() errors

\*/

TSBQueue \*tsbq\_create(long capacity);

/\*

\* destroys the bounded queue; for each element, if userFunction != NULL,

\* invokes userFunction on the element; then returns any queue structure

\* associated with the element; finally, deletes any remaining structures

\* associated with the queue

\*/

void tsbq\_destroy(TSBQueue \*tsbq, void (\*userFunction)(void \*element));

/\*

\* clears the queue; for each element, if userFunction != NULL, invokes

\* userFunction on the element; then returns any queue structure associated with

\* the element

\*

\* upon return, the queue is empty

\*/

void tsbq\_clear(TSBQueue \*tsbq, void (\*userFunction)(void \*element));

/\*

\* obtains the lock for exclusive access

\*/

void tsuq\_lock(TSBQueue \*uq);

/\*

\* returns the lock

\*/

void tsuq\_unlock(TSBQueue \*uq);

/\*

\* non-blocking append of `element' to the end of the bounded queue

\*

\* returns 1 if successful, 0 if unsuccesful (queue is full)

\*/

int tsbq\_add(TSBQueue \*tsbq, void \*element);

/\*

\* blocking append of `element' to the end of the bounded queue

\*/

void tsbq\_put(TSBQueue \*tsbq, void \*element);

/\*

\* non-blocking retrieval, but does not remove, the head of the queue

\*

\* returns 1 if successful, 0 if unsuccessful (queue is empty)

\*/

int tsbq\_peek(TSBQueue \*tsbq, void \*\*element);

/\*

\* non-blocking retrieval, and removal, of the head of the queue

\*

\* return 1 if successful, 0 if not (queue is empty)

\*/

int tsbq\_remove(TSBQueue \*tsbq, void \*\*element);

/\*

\* blocking retrieval, and removal, of the head of the queue

\*/

void tsbq\_take(TSBQueue \*tsbq, void \*\*element);

/\*

\* returns the number of elements in the queue

\*/

long tsbq\_size(TSBQueue \*tsbq);

/\*

\* returns true if the queue is empty, false if not

\*/

int tsbq\_isEmpty(TSBQueue \*tsbq);

/\*

\* returns an array containing all of the elements of the queue in

\* proper sequence (from first to last element); returns the length of the

\* queue in `len'

\*

\* returns pointer to void \* array of elements, or NULL if malloc failure

\*/

void \*\*tsbq\_toArray(TSBQueue \*tsbq, long \*len);

/\*

\* creates an iterator for running through the queue

\*

\* returns pointer to the Iterator or NULL

\*/

TSIterator \*tsbq\_it\_create(TSBQueue \*tsbq);

#endif /\* \_TSBQUEUE\_H\_ \*/

## bqueue.c

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

/\*

\* implementation for generic bounded FIFO queue

\*/

#include "bqueue.h"

#include <stdlib.h>

struct bqueue {

long count;

long size;

int in;

int out;

void \*\*buffer;

};

BQueue \*bq\_create(long capacity) {

BQueue \*bq = (BQueue \*)malloc(sizeof(BQueue));

if (bq != NULL) {

long cap = capacity;

void \*\*tmp;

if (cap <= 0L)

cap = DEFAULT\_CAPACITY;

else if (cap > MAX\_CAPACITY)

cap = MAX\_CAPACITY;

tmp = (void \*\*)malloc(cap \* sizeof(void \*));

if (tmp == NULL) {

free(bq);

bq = NULL;

} else {

int i;

bq->count = 0;

bq->size = cap;

bq->in = 0;

bq->out = 0;

for (i = 0; i < cap; i++)

tmp[i] = NULL;

bq->buffer = tmp;

}

}

return bq;

}

static void purge(BQueue \*bq, void (\*userFunction)(void \*element)) {

if (userFunction != NULL) {

int i, n;

for (i = bq->out, n = bq->count; n > 0; i = (i + 1) % bq->size, n--)

(\*userFunction)(bq->buffer[i]);

}

}

void bq\_destroy(BQueue \*bq, void (\*userFunction)(void \*element)) {

purge(bq, userFunction);

free(bq->buffer);

free(bq);

}

void bq\_clear(BQueue \*bq, void (\*userFunction)(void \*element)) {

int i;

purge(bq, userFunction);

for (i = 0; i < bq->size; i++)

bq->buffer[i] = NULL;

bq->count = 0;

bq->in = 0;

bq->out = 0;

}

int bq\_add(BQueue \*bq, void \*element) {

int i;

if (bq->count == bq->size)

return 0;

i = bq->in;

bq->buffer[i] = element;

bq->in = (i + 1) % bq->size;

bq->count++;

return 1;

}

static int retrieve(BQueue \*bq, void \*\*element, int ifRemove) {

int i;

if (bq->count <= 0)

return 0;

i = bq->out;

\*element = bq->buffer[i];

if (ifRemove) {

bq->out = (i + 1) % bq->size;

bq->count--;

}

return 1;

}

int bq\_peek(BQueue \*bq, void \*\*element) {

return retrieve(bq, element, 0);

}

int bq\_remove(BQueue \*bq, void \*\*element) {

return retrieve(bq, element, 1);

}

long bq\_size(BQueue \*bq) {

return bq->count;

}

int bq\_isEmpty(BQueue \*bq) {

return (bq->count == 0L);

}

static void \*\*toArray(BQueue \*bq) {

void \*\*tmp = NULL;

if (bq->count > 0L) {

tmp = (void \*\*)malloc(bq->count \* sizeof(void \*));

if (tmp != NULL) {

int i, j, n;

n = bq->count;

for (i = bq->out, j = 0; n > 0; i = (i+1) % bq->size, j++, n--) {

tmp[j] = bq->buffer[i];

}

}

}

return tmp;

}

void \*\*bq\_toArray(BQueue \*bq, long \*len) {

void \*\*tmp = toArray(bq);

if (tmp != NULL)

\*len = bq->count;

return tmp;

}

Iterator \*bq\_it\_create(BQueue \*bq) {

Iterator \*it = NULL;

void \*\*tmp = toArray(bq);

if (tmp != NULL) {

it = it\_create(bq->count, tmp);

if (it == NULL)

free(tmp);

}

return it;

}

## tsbqueue.c

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

/\*

\* implementation for thread-safe generic bounded queue

\*/

#include "tsbqueue.h"

#include "bqueue.h"

#include <stdlib.h>

#include <pthread.h>

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

#define COND(bq) &((bq)->cond)

struct tsbqueue {

long cap;

BQueue \*bq;

pthread\_mutex\_t lock; /\* this is a recursive lock \*/

pthread\_cond\_t cond; /\* needed for take \*/

};

TSBQueue \*tsbq\_create(long capacity) {

TSBQueue \*tsbq = (TSBQueue \*)malloc(sizeof(TSBQueue));

if (tsbq != NULL) {

BQueue \*bq = bq\_create(capacity);

if (bq == NULL) {

free(tsbq);

tsbq = NULL;

} else {

pthread\_mutexattr\_t ma;

long cap = capacity;

if (cap <= 0L)

cap = DEFAULT\_CAPACITY;

else if (cap > MAX\_CAPACITY)

cap = MAX\_CAPACITY;

tsbq->cap = cap;

pthread\_mutexattr\_init(&ma);

pthread\_mutexattr\_settype(&ma, PTHREAD\_MUTEX\_RECURSIVE);

tsbq->bq = bq;

pthread\_mutex\_init(LOCK(tsbq), &ma);

pthread\_mutexattr\_destroy(&ma);

pthread\_cond\_init(COND(tsbq), NULL);

}

}

return tsbq;

}

void tsbq\_destroy(TSBQueue \*tsbq, void (\*userFunction)(void \*element)) {

pthread\_mutex\_lock(LOCK(tsbq));

bq\_destroy(tsbq->bq, userFunction);

pthread\_mutex\_unlock(LOCK(tsbq));

pthread\_mutex\_destroy(LOCK(tsbq));

pthread\_cond\_destroy(COND(tsbq));

free(tsbq);

}

void tsbq\_clear(TSBQueue \*tsbq, void (\*userFunction)(void \*element)) {

pthread\_mutex\_lock(LOCK(tsbq));

bq\_clear(tsbq->bq, userFunction);

pthread\_mutex\_unlock(LOCK(tsbq));

}

void tsbq\_lock(TSBQueue \*tsbq) {

pthread\_mutex\_lock(LOCK(tsbq));

}

void tsbq\_unlock(TSBQueue \*tsbq) {

pthread\_mutex\_unlock(LOCK(tsbq));

}

int tsbq\_add(TSBQueue \*tsbq, void \*element) {

int result;

pthread\_mutex\_lock(LOCK(tsbq));

result = bq\_add(tsbq->bq, element);

if (result)

pthread\_cond\_signal(COND(tsbq));

pthread\_mutex\_unlock(LOCK(tsbq));

return result;

}

void tsbq\_put(TSBQueue \*tsbq, void \*element) {

pthread\_mutex\_lock(LOCK(tsbq));

while (bq\_size(tsbq->bq) == tsbq->cap)

pthread\_cond\_wait(COND(tsbq), LOCK(tsbq));

(void)bq\_add(tsbq->bq, element);

pthread\_cond\_signal(COND(tsbq));

pthread\_mutex\_unlock(LOCK(tsbq));

}

int tsbq\_peek(TSBQueue \*tsbq, void \*\*element) {

int result;

pthread\_mutex\_lock(LOCK(tsbq));

result = bq\_peek(tsbq->bq, element);

pthread\_mutex\_unlock(LOCK(tsbq));

return result;

}

int tsbq\_remove(TSBQueue \*tsbq, void \*\*element) {

int result;

pthread\_mutex\_lock(LOCK(tsbq));

result = bq\_remove(tsbq->bq, element);

if (result)

pthread\_cond\_signal(COND(tsbq));

pthread\_mutex\_unlock(LOCK(tsbq));

return result;

}

void tsbq\_take(TSBQueue \*tsbq, void \*\*element) {

pthread\_mutex\_lock(LOCK(tsbq));

while (bq\_size(tsbq->bq) == 0L)

pthread\_cond\_wait(COND(tsbq), LOCK(tsbq));

(void)bq\_remove(tsbq->bq, element);

pthread\_cond\_signal(COND(tsbq));

pthread\_mutex\_unlock(LOCK(tsbq));

}

long tsbq\_size(TSBQueue \*tsbq) {

long result;

pthread\_mutex\_lock(LOCK(tsbq));

result = bq\_size(tsbq->bq);

pthread\_mutex\_unlock(LOCK(tsbq));

return result;

}

int tsbq\_isEmpty(TSBQueue \*tsbq) {

int result;

pthread\_mutex\_lock(LOCK(tsbq));

result = bq\_isEmpty(tsbq->bq);

pthread\_mutex\_unlock(LOCK(tsbq));

return result;

}

void \*\*tsbq\_toArray(TSBQueue \*tsbq, long \*len) {

void \*\*result;

pthread\_mutex\_lock(LOCK(tsbq));

result = bq\_toArray(tsbq->bq, len);

pthread\_mutex\_unlock(LOCK(tsbq));

return result;

}

TSIterator \*tsbq\_it\_create(TSBQueue \*tsbq) {

TSIterator \*it = NULL;

void \*\*tmp;

long len;

pthread\_mutex\_lock(LOCK(tsbq));

tmp = bq\_toArray(tsbq->bq, &len);

if (tmp != NULL) {

it = tsit\_create(LOCK(tsbq), len, tmp);

if (it == NULL)

free(tmp);

}

if (it == NULL)

pthread\_mutex\_unlock(LOCK(tsbq));

return it;

}

## bqtest.c (you can create your own tsbqtest.c)

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

#include "bqueue.h"

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

int main(int argc, char \*argv[]) {

char buf[1024];

char \*p, \*q;

BQueue \*bq;

long i, n;

FILE \*fd;

char \*\*array;

Iterator \*it;

if (argc != 2) {

fprintf(stderr, "usage: ./bqtest file\n");

return -1;

}

/\*

\* test of queue exhaustion

\*/

printf("===== test of exhaustion of small queue\n");

if ((bq = bq\_create(10L)) == NULL) {

fprintf(stderr, "Error creating bounded queue, size 10, of strings\n");

return -1;

}

for (i = 0; i < 100; i++) {

sprintf(buf, "Line %ld\n", i);

if ((p = strdup(buf)) == NULL) {

fprintf(stderr, "Unable to duplicate string on heap\n");

return -1;

}

if (! bq\_add(bq, p)) {

free(p);

break;

}

}

printf("bounded queue filled after %ld additions\n", i);

while (! bq\_isEmpty(bq)) {

(void)bq\_remove(bq, (void \*\*)&p);

printf("%s", p);

free(p);

}

/\*

\* test of bq\_destroy(bq, NULL);

\*/

printf("===== test of destroy(NULL)\n");

bq\_destroy(bq, NULL);

if ((fd = fopen(argv[1], "r")) == NULL) {

fprintf(stderr, "Unable to open %s to read\n", argv[1]);

return -1;

}

if ((bq = bq\_create(10000L)) == NULL) {

fprintf(stderr, "Error creating bounded queue, size 10000, of strings\n");

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 (!bq\_add(bq, p)) {

fprintf(stderr, "Error adding string to bounded queue\n");

return -1;

}

}

fclose(fd);

n = bq\_size(bq);

/\*

\* test of remove()

\*/

printf("===== test of peek and remove\n");

for (i = 0; i < n; i++) {

if (!bq\_peek(bq, (void \*\*)&p)) {

fprintf(stderr, "Error retrieving %ld'th element\n", i);

return -1;

}

if (!bq\_remove(bq, (void \*\*)&q)) {

fprintf(stderr, "Error retrieving %ld'th element\n", i);

return -1;

}

if (strcmp(p, q) != 0) {

fprintf(stderr, "Returns from peek and remove are not the same\n");

return -1;

}

printf("%s", p);

}

/\*

\* test of destroy with free userFunction

\*/

printf("===== test of destroy(free)\n");

bq\_destroy(bq, free);

if ((bq = bq\_create(10000L)) == NULL) {

fprintf(stderr, "Error creating bounded queue 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 (!bq\_add(bq, p)) {

fprintf(stderr, "Error adding string to bounded queue\n");

return -1;

}

}

fclose(fd);

/\*

\* test of toArray

\*/

printf("===== test of toArray\n");

if ((array = (char \*\*)bq\_toArray(bq, &n)) == NULL) {

fprintf(stderr, "Error in invoking bq\_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 = bq\_it\_create(bq)) == 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);

bq\_destroy(bq, free);

return 0;

}

# Appendix H – UQueue and TSUQueue

## uqueue.h

#ifndef \_UQUEUE\_H\_

#define \_UQUEUE\_H\_

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

/\*

\* interface definition for generic unbounded FIFO queue

\*

\* patterned roughly after Java 6 Queue interface

\*/

#include "iterator.h"

typedef struct uqueue UQueue; /\* opaque type definition \*/

/\*

\* create an unbounded queue

\*

\* returns a pointer to the queue, or NULL if there are malloc() errors

\*/

UQueue \*uq\_create(void);

/\*

\* destroys the unbounded queue; for each element, if userFunction != NULL,

\* invokes userFunction on the element; then returns any queue structure

\* associated with the element; finally, deletes any remaining structures

\* associated with the queue

\*/

void uq\_destroy(UQueue \*uq, void (\*userFunction)(void \*element));

/\*

\* clears the queue; for each element, if userFunction != NULL, invokes

\* userFunction on the element; then returns any queue structure associated with

\* the element

\*

\* upon return, the queue is empty

\*/

void uq\_clear(UQueue \*uq, void (\*userFunction)(void \*element));

/\*

\* appends `element' to the end of the unbounded queue

\*

\* returns 1 if successful, 0 if unsuccesful (malloc errors)

\*/

int uq\_add(UQueue \*uq, void \*element);

/\*

\* retrieves, but does not remove, the head of the queue

\*

\* returns 1 if successful, 0 if unsuccessful (queue is empty)

\*/

int uq\_peek(UQueue \*uq, void \*\*element);

/\*

\* Retrieves, and removes, the head of the queue

\*

\* return 1 if successful, 0 if not

\*/

int uq\_remove(UQueue \*uq, void \*\*element);

/\*

\* returns the number of elements in the queue

\*/

long uq\_size(UQueue \*uq);

/\*

\* returns true if the queue is empty, false if not

\*/

int uq\_isEmpty(UQueue \*uq);

/\*

\* returns an array containing all of the elements of the queue in

\* proper sequence (from first to last element); returns the length of the

\* queue in `len'

\*

\* returns pointer to void \* array of elements, or NULL if malloc failure

\*/

void \*\*uq\_toArray(UQueue \*uq, long \*len);

/\*

\* creates an iterator for running through the queue

\*

\* returns pointer to the Iterator or NULL

\*/

Iterator \*uq\_it\_create(UQueue \*uq);

#endif /\* \_UQUEUE\_H\_ \*/

## tsuqueue.h

#ifndef \_TSUQUEUE\_H\_

#define \_TSUQUEUE\_H\_

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

/\*

\* interface definition for generic threadsafe unbounded FIFO queue

\*

\* patterned roughly after Java 6 Queue interface

\*/

#include "tsiterator.h"

typedef struct tsuqueue TSUQueue; /\* opaque type definition \*/

/\*

\* create an unbounded queue

\*

\* returns a pointer to the queue, or NULL if there are malloc() errors

\*/

TSUQueue \*tsuq\_create(void);

/\*

\* destroys the unbounded queue; for each element, if userFunction != NULL,

\* invokes userFunction on the element; then returns any queue structure

\* associated with the element; finally, deletes any remaining structures

\* associated with the queue

\*/

void tsuq\_destroy(TSUQueue \*uq, void (\*userFunction)(void \*element));

/\*

\* clears the queue; for each element, if userFunction != NULL, invokes

\* userFunction on the element; then returns any queue structure associated with

\* the element

\*

\* upon return, the queue is empty

\*/

void tsuq\_clear(TSUQueue \*uq, void (\*userFunction)(void \*element));

/\*

\* obtains the lock for exclusive access

\*/

void tsuq\_lock(TSUQueue \*uq);

/\*

\* returns the lock

\*/

void tsuq\_unlock(TSUQueue \*uq);

/\*

\* append of `element' to the end of the unbounded queue

\*

\* returns 1 if successful, 0 if unsuccesful (malloc errors)

\*/

int tsuq\_add(TSUQueue \*uq, void \*element);

/\*

\* retrieves, but does not remove, the head of the queue

\*

\* returns 1 if successful, 0 if unsuccessful (queue is empty)

\*/

int tsuq\_peek(TSUQueue \*uq, void \*\*element);

/\*

\* Nonblocking retrieval and removal of the head of the queue

\*

\* return 1 if successful, 0 if not

\*/

int tsuq\_remove(TSUQueue \*uq, void \*\*element);

/\*

\* Blocking retrieval and removal of the head of the queue

\*

\* return 1 if successful, 0 if not

\*/

void tsuq\_take(TSUQueue \*uq, void \*\*element);

/\*

\* returns the number of elements in the queue

\*/

long tsuq\_size(TSUQueue \*uq);

/\*

\* returns true if the queue is empty, false if not

\*/

int tsuq\_isEmpty(TSUQueue \*uq);

/\*

\* returns an array containing all of the elements of the queue in

\* proper sequence (from first to last element); returns the length of the

\* queue in `len'

\*

\* returns pointer to void \* array of elements, or NULL if malloc failure

\*/

void \*\*tsuq\_toArray(TSUQueue \*uq, long \*len);

/\*

\* creates an iterator for running through the queue

\*

\* returns pointer to the Iterator or NULL

\*/

TSIterator \*tsuq\_it\_create(TSUQueue \*uq);

#endif /\* \_TSUQUEUE\_H\_ \*/

## uqueue.c

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

/\*

\* implementation for generic unbounded FIFO queue

\*/

#include "uqueue.h"

#include "linkedlist.h"

#include <stdlib.h>

struct uqueue {

LinkedList \*ll;

};

UQueue \*uq\_create(void) {

UQueue \*uq = (UQueue \*)malloc(sizeof(UQueue));

if (uq != NULL) {

LinkedList \*ll = ll\_create();

if (ll == NULL) {

free(uq);

uq = NULL;

} else {

uq->ll = ll;

}

}

return uq;

}

void uq\_destroy(UQueue \*uq, void (\*userFunction)(void \*element)) {

ll\_destroy(uq->ll, userFunction);

free(uq);

}

void uq\_clear(UQueue \*uq, void (\*userFunction)(void \*element)) {

ll\_clear(uq->ll, userFunction);

}

int uq\_add(UQueue \*uq, void \*element) {

int result;

result = ll\_add(uq->ll, element);

return result;

}

int uq\_peek(UQueue \*uq, void \*\*element) {

int result;

result = ll\_getFirst(uq->ll, element);

return result;

}

int uq\_remove(UQueue \*uq, void \*\*element) {

int result;

result = ll\_removeFirst(uq->ll, element);

return result;

}

long uq\_size(UQueue \*uq) {

long result;

result = ll\_size(uq->ll);

return result;

}

int uq\_isEmpty(UQueue \*uq) {

return (ll\_size(uq->ll) == 0L);

}

void \*\*uq\_toArray(UQueue \*uq, long \*len) {

void \*\*result;

result = ll\_toArray(uq->ll, len);

return result;

}

Iterator \*uq\_it\_create(UQueue \*uq) {

Iterator \*it = NULL;

void \*\*tmp;

long len;

tmp = ll\_toArray(uq->ll, &len);

if (tmp != NULL) {

it = it\_create(len, tmp);

if (it == NULL)

free(tmp);

}

return it;

}

## tsuqueue.c

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

/\*

\* implementation for thread-safe generic unbounded queue

\*/

#include "tsuqueue.h"

#include "uqueue.h"

#include <stdlib.h>

#include <pthread.h>

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

#define COND(uq) &((uq)->cond)

struct tsuqueue {

UQueue \*uq;

pthread\_mutex\_t lock; /\* this is a recursive lock \*/

pthread\_cond\_t cond; /\* needed for take \*/

};

TSUQueue \*tsuq\_create(void) {

TSUQueue \*tsuq = (TSUQueue \*)malloc(sizeof(TSUQueue));

if (tsuq != NULL) {

UQueue \*uq = uq\_create();

if (uq == NULL) {

free(tsuq);

tsuq = NULL;

} else {

pthread\_mutexattr\_t ma;

pthread\_mutexattr\_init(&ma);

pthread\_mutexattr\_settype(&ma, PTHREAD\_MUTEX\_RECURSIVE);

tsuq->uq = uq;

pthread\_mutex\_init(LOCK(tsuq), &ma);

pthread\_mutexattr\_destroy(&ma);

pthread\_cond\_init(COND(tsuq), NULL);

}

}

return tsuq;

}

void tsuq\_destroy(TSUQueue \*tsuq, void (\*userFunction)(void \*element)) {

pthread\_mutex\_lock(LOCK(tsuq));

uq\_destroy(tsuq->uq, userFunction);

pthread\_mutex\_unlock(LOCK(tsuq));

pthread\_mutex\_destroy(LOCK(tsuq));

pthread\_cond\_destroy(COND(tsuq));

free(tsuq);

}

void tsuq\_clear(TSUQueue \*tsuq, void (\*userFunction)(void \*element)) {

pthread\_mutex\_lock(LOCK(tsuq));

uq\_clear(tsuq->uq, userFunction);

pthread\_mutex\_unlock(LOCK(tsuq));

}

void tsuq\_lock(TSUQueue \*tsuq) {

pthread\_mutex\_lock(LOCK(tsuq));

}

void tsuq\_unlock(TSUQueue \*tsuq) {

pthread\_mutex\_unlock(LOCK(tsuq));

}

int tsuq\_add(TSUQueue \*tsuq, void \*element) {

int result;

pthread\_mutex\_lock(LOCK(tsuq));

result = uq\_add(tsuq->uq, element);

pthread\_cond\_signal(COND(tsuq));

pthread\_mutex\_unlock(LOCK(tsuq));

return result;

}

int tsuq\_peek(TSUQueue \*tsuq, void \*\*element) {

int result;

pthread\_mutex\_lock(LOCK(tsuq));

result = uq\_peek(tsuq->uq, element);

pthread\_mutex\_unlock(LOCK(tsuq));

return result;

}

int tsuq\_remove(TSUQueue \*tsuq, void \*\*element) {

int result;

pthread\_mutex\_lock(LOCK(tsuq));

result = uq\_remove(tsuq->uq, element);

pthread\_mutex\_unlock(LOCK(tsuq));

return result;

}

void tsuq\_take(TSUQueue \*tsuq, void \*\*element) {

pthread\_mutex\_lock(LOCK(tsuq));

while (uq\_size(tsuq->uq) == 0L)

pthread\_cond\_wait(COND(tsuq), LOCK(tsuq));

(void)uq\_remove(tsuq->uq, element);

pthread\_mutex\_unlock(LOCK(tsuq));

}

long tsuq\_size(TSUQueue \*tsuq) {

long result;

pthread\_mutex\_lock(LOCK(tsuq));

result = uq\_size(tsuq->uq);

pthread\_mutex\_unlock(LOCK(tsuq));

return result;

}

int tsuq\_isEmpty(TSUQueue \*tsuq) {

int result;

pthread\_mutex\_lock(LOCK(tsuq));

result = uq\_isEmpty(tsuq->uq);

pthread\_mutex\_unlock(LOCK(tsuq));

return result;

}

void \*\*tsuq\_toArray(TSUQueue \*tsuq, long \*len) {

void \*\*result;

pthread\_mutex\_lock(LOCK(tsuq));

result = uq\_toArray(tsuq->uq, len);

pthread\_mutex\_unlock(LOCK(tsuq));

return result;

}

TSIterator \*tsuq\_it\_create(TSUQueue \*tsuq) {

TSIterator \*it = NULL;

void \*\*tmp;

long len;

pthread\_mutex\_lock(LOCK(tsuq));

tmp = uq\_toArray(tsuq->uq, &len);

if (tmp != NULL) {

it = tsit\_create(LOCK(tsuq), len, tmp);

if (it == NULL)

free(tmp);

}

if (it == NULL)

pthread\_mutex\_unlock(LOCK(tsuq));

return it;

}

## uqtest.c (you can create your own tsuqtest.c)

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

#include "uqueue.h"

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

int main(int argc, char \*argv[]) {

char buf[1024];

char \*p, \*q;

UQueue \*uq;

long i, n;

FILE \*fd;

char \*\*array;

Iterator \*it;

if (argc != 2) {

fprintf(stderr, "usage: ./uqtest file\n");

return -1;

}

if ((fd = fopen(argv[1], "r")) == NULL) {

fprintf(stderr, "Unable to open %s to read\n", argv[1]);

return -1;

}

if ((uq = uq\_create()) == NULL) {

fprintf(stderr, "Error creating unbounded queue of strings\n");

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 (!uq\_add(uq, p)) {

fprintf(stderr, "Error adding string to unbounded queue\n");

return -1;

}

}

fclose(fd);

n = uq\_size(uq);

/\*

\* test of peek and remove()

\*/

printf("===== test of peek and remove\n");

for (i = 0; i < n; i++) {

if (!uq\_peek(uq, (void \*\*)&p)) {

fprintf(stderr, "Error retrieving %ld'th element\n", i);

return -1;

}

if (!uq\_remove(uq, (void \*\*)&q)) {

fprintf(stderr, "Error retrieving %ld'th element\n", i);

return -1;

}

if (strcmp(p, q) != 0) {

fprintf(stderr, "Returns from peek and remove are not the same\n");

return -1;

}

printf("%s", p);

}

/\*

\* test of destroy with free userFunction

\*/

printf("===== test of destroy(free)\n");

uq\_destroy(uq, free);

if ((uq = uq\_create()) == NULL) {

fprintf(stderr, "Error creating unbounded queue 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 (!uq\_add(uq, p)) {

fprintf(stderr, "Error adding string to unbounded queue\n");

return -1;

}

}

fclose(fd);

/\*

\* test of toArray

\*/

printf("===== test of toArray\n");

if ((array = (char \*\*)uq\_toArray(uq, &n)) == NULL) {

fprintf(stderr, "Error in invoking uq\_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 = uq\_it\_create(uq)) == 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 uq\_clear(free)

\*/

printf("===== test of uq\_clear(free)\n");

uq\_clear(uq, free);

printf("===== test of uq\_destroy(NULL)\n");

uq\_destroy(uq, NULL);

return 0;

}

1. OpenGroup standard C064, Extended API Set, Part 3. [↑](#footnote-ref-1)