Implementation of a Set ADT

Due at 11:59pm on Friday, 7 June 2019

A set is an abstract data type that can store unique values without any particular order. It is a computer implementation of the mathematical concept of a finite set. Unlike most other collections classes/types, rather than retrieving a specific element from a set, one typically tests a value for membership in a set. We are interested in mutable sets, in which insertion and deletion of elements from the set are permitted.[1]

The interface to a Set ADT is defined in section 1 below. You are to completely implement the constructor and the ADT methods. You have a choice of implementing the set using an array, a linked list, or a hash table. If you implement it using an array or a linked list, the project will be worth ***at most*** 30 of the 50 points. If you implement it using a hash table, the project will be able to access all 50 points for the project.

1 Requirements

1.1 Creating an instance of a `Set`

You can create a `Set` instance with a declaration of the following form:

```
const Set *s = Set_create(0L, cmpFxn, 0.0, hashFxn);
```

The first argument to the constructor is an indication of the initial capacity; if specified as 0L, a default capacity is chosen. The second parameter is a function used by the implementation to compare two elements in the set; it has the signature

```
int (*cmpFxn)(void *first, void *second);
```

this function returns a value <0 if `first < second`, 0 if `first == second`, and >0 if `first > second.`

If the set is implemented using an array or a singly-linked list, the third and fourth parameters to `Set_create()` are ignored by the implementation; the caller must still specify them, and the latter function can be specified as `NULL`.

If the set is implemented using a hash table, the additional parameters have the following meaning:

- the third parameter is the load factor to be maintained in the hash table that underlies the set implementation; if specified as 0.0, a default load factor is used;
- the fourth parameter is a function that will produce a hash value for an element in the set; it has the signature `long (*hashFxn)(void *, long)`; the function will produce a long integer that is a hash of the first argument, modulo the value of the second argument

[1] This definition is largely taken from https://en.wikipedia.org/wiki/Set_(abstract_data_type).

to the hash function – i.e., if the second argument is `N`, it will return a value in the range `[0, N)`.

As you can see in the next subsection, there are a large number of methods that one can invoke on a `Set` instance. All of these methods ***must*** be implemented.

1.2 Methods of the `Set` ADT

In each of the following, it is assumed that you have invoked `Set_create()` to create a pointer to a Set instance named `s`. Invocation of each method is shown using this pointer prior to a description of what the method does.

If the signature indicates a `value`, this is a `void *` pointer. If the signature indicates a `freeFxn`, this indicates a pointer to a function that satisfies the following signature:
`void (*freeFxn)(void*)`.

`s->destroy(s, freeFxn);`
Destroys `s`; for each element in the set, if `freeFxn != NULL`, that function is invoked on that element; the storage associated with `s` is then returned to the heap; no return value.

`s->clear(s, freeFxn);`
Clears all elements from `s`; for each element in the set, if `freeFxn != NULL`, that function is invoked on that element; upon return, `s` will be empty; no return value.

`int status = s->add(s, value);`
Adds the specified `value` to the set if it is not already present. Returns 1 if `value` was added, 0 if the value was already present.

`int status = s->contains(s, value);`
Returns 1 if the `value` is contained in the set, 0 if not.

`int value = s->isEmpty(s);`
Returns 1 if `s` is empty, 0 if not.

`int status = s->remove(s, value, freeFxn);`
Removes the `value` from the set; if freeFxn != NULL, that function is invoked on that value before removing it from the set; returns 1 if `value` was present and removed, 0 if not.

`long length = s->size(s);`
Returns the number of elements in the set.

`void **array = s->toArray(s, long *len);`
Returns the elements of the set as an array of `void*` pointers in an arbitrary order; returns a pointer to the array or `NULL` if error; returns the number of elements in the array in `*len`.
N.B. the caller is responsible for freeing the array of `void*` pointers when finished with it.

`const Iterator *it = s->itCreate(s);`
Creates a generic iterator to `s`; returns pointer to the Iterator instance or `NULL` if failure.

1.3 set.h

```
#ifndef _SET_H_
#define _SET_H_

/* BSD Header removed to conserve space */

#include "ADTs/iterator.h"                /* needed for factory method */

/*
 * interface definition for generic set implementation
 */

typedef struct set Set;  /* forward reference */

/*
 * create a Set with the specified initial capacity
 * if capacity == 0L, the initial capacity is set to DEFAULT_CAPACITY
 *
 * cmpFunction is used to determine equality between two objects, with
 * `cmpFunction(first, second)' returning 0 if first==second, <>0 otherwise
 *
 * if you are implementing the Set using an array or a singly-linked list:
 *    ignore the loadFactor and hashFunction arguments.
 *
 * if you are implementing the Set using a hash table:
 *    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 must be resized, doubling the number of buckets, up to a max
 *    number of buckets (134,217,728)
 *
 *    hashFunction is used to hash a value into the hash table that underlies
 *    the set, with `hashFunction(value, N)' returning a number in [0,N)
 *
 * returns a pointer to the set, or NULL if there are malloc() errors
 */
#define DEFAULT_CAPACITY 16L

const Set *Set_create( long capacity. int (*cmpFunction)(void *, void *),
                       double loadFactor, long (*hashFunction)(void *, long)
                     );

/*
 * now define struct set
 */
struct set {
/*
 * the private data of the set
 */
    void *self;

/*
 * destroys the set; for each element, if freeFxn != NULL,
 * it is invoked on the element; the storage associated with
 * the set is then returned to the heap
 */
```

```
    void (*destroy)(const Set *s, void (*freeFxn)(void *element));

/*
 * clears all elements from the set; for each element,
 * if freeFxn != NULL, it is invoked on the element;
 * any storage associated with the entry in the set is then
 * returned to the heap
 *
 * upon return, the set will be empty
 */
    void (*clear)(const Set *s, void (*freeFxn)(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 (*add)(const Set *s, void *element);

/*
 * returns 1 if the set contains the specified element, 0 if not
 */
    int (*contains)(const Set *s, void *element);

/*
 * returns 1 if hashset is empty, 0 if it is not
 */
    int (*isEmpty)(const Set *s);

/*
 * removes the specified element from the set, if present
 *
 * if freeFxn != NULL, invokes it on the element before removing it
 *
 * returns 1 if successful, 0 if not present
 */
    int (*remove)(const Set *s, void *element, void (*freeFxn)(void *));

/*
 * returns the number of elements in the hashset
 */
    long (*size)(const Set *s);

/*
 * return the elements of the set as an array of void * pointers in an
 * arbitrary order
 *
 * returns pointer to the array or NULL if error
 * returns the number of elements in the array in `*len'
 */
    void **(*toArray)(const Set *s, long *len);

/*
 * create generic iterator to this hashset
 *
 * returns pointer to the Iterator or NULL if failure
 */
```

```
    const Iterator *(*itCreate)(const Set *s);
};

#endif /* _SET_H_ */
```

2 Starting files

In Canvas, in Files/Projects, you will find a gzipped tar archive named `projectECstart.tgz`; this file contains the following files:

- `longtest.c` – a program that thoroughly tests your Set implementation using long values
- `stringtest.c` – a program that thoroughly tests your Set implementation using C string values
- `sort.h` – a header file defining the function signature for a `sort()` function
- `sort.c` – the source file for the `sort()` function; you must compile this and link it to `longtest.o` and your `set.o` file to create the `longtest` executable image; it must also be linked with `stringtest.o` and your `set.o` file to create the `stringtest` executable image
- `Makefile` – a makefile for creating your `longtest` and `stringtest` executable image files
- `set.c` – a template source file that you should expand for your implementation.

3 Hints

If you are implementing using an array, I suggest that you thoroughly review Section 5.4 of the textbook; if you are using a linked list, I suggest that you thoroughly review the model solution for queue.c from project 7; and if you are using a hash table, I suggest that you thoroughly review Chapter 13 on Maps, especially sections13.4 and 13.5.

4 Checking that your solution works correctly

`longtest and stringtest` each take one or more arguments; each argument specifies the number of a test to perform. The tests build on each other, so during development of your Set implementation, you are encouraged to first make sure that you pass test 1, then test 2, then test 3, … .

The tests that `longtest and stringtest` performs are:

1	creation and destruction of a set
2	addition of a single value
3	addition of a duplicate value
4	`isEmpty()` on an empty set
5	`isEmpty()` on a non-empty set
6	`contains()` on an empty set

7	`contains()` on a non-empty set and the value is present
8	`contains()` on a non-empty set and the value is not present
9	`remove()` on an empty set
10	`remove()` on a non-empty set and the value is present
11	`remove()` on a non-empty set and the value is not present
12	`size()` on an empty set
13	`size()` on a non-empty set
14	addition of multiple, unique values
15	`toArray()` on a non-empty set
16	`itCreate()` on a non-empty set
17	use of `toArray()` results with `sort()` to generate sorted output

You are encouraged to start your testing with longtest, only going to stringtest if longtest passes each of the 17 tests. Furthermore, you are encouraged to conduct the tests one at a time, as in the following:

```
./longtest 1
# if the output from the above command indicates success, then
valgrind ./longtest 1
```

If the result of test 1 indicates *failure*, or if `valgrind` reports warnings or lost memory, track down the cause in `set.c` and fix it. Keep conducting this pair of tests until test 1 indicates *success* and there are no warnings or lost memory in the `valgrind` report, then move on to test 2. Keep testing one at a time until all tests are passed and there are no warnings or lost memory in the `valgrind` reports.

If your code works correctly, then the report from executing

```
valgrind ./longtest 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
```

should look as follows:

```
==6354== Memcheck, a memory error detector
==6354== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6354== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==6354== Command: ./longtest 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
==6354==
Test creation and destruction of a set ... success
Test addition of a single value ... success
Test addition of a duplicate value ... success
Test isEmpty() on an empty set ... success
Test isEmpty() on a non-empty set ... success
Test contains() on an empty set ... success
Test contains() on a non-empty set and value is present ... success
Test contains() on a non-empty set and value is not present ... success
Test remove() on an empty set ... success
Test remove() on a non-empty set and value is present ... success
Test remove() on a non-empty set and value is not present ... success
Test size() on an empty set ... success
Test size() on a non-empty set ... success
Test addition of multiple, unique values ... success
Test toArray() ... success
Test itCreate() ... success
Show set elements sorted 1 2 3 4 5 6 7 8 9 10 ... success
==6354==
==6354== HEAP SUMMARY:
==6354==     in use at exit: 0 bytes in 0 blocks
==6354==   total heap usage: 91 allocs, 91 frees, 9,680 bytes allocated
==6354==
==6354== All heap blocks were freed -- no leaks are possible
==6354==
==6354== For counts of detected and suppressed errors, rerun with: -v
==6354== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

After you have reached this state using `longtest`, do the same with `stringtest`.

5 Submission[2]

You will submit your solutions electronically by uploading a gzipped tar archive[3] via Canvas.

Your TGZ archive should be named `<duckid>-projectEC.tgz`, where `<duckid>` is your "duckid". It should contain your file `set.c`; it should also contain a file named `report.txt`; this file should contain your name, duckid, the names of any classmates who helped you and what assistance they provided, and the current state of your solution; **in particular, it MUST indicate whether you have implemented the Set using an array, linked list, or a hash table**. It should ***not*** include `sort.h`, `sort.c`, `longtest.c`, `stringtest.c`, or `Makefile`.

[2] ***A 100% penalty will be assessed if you do not follow these submission instructions – i.e., I will not mark your submission and you receive a no extra credit points.***

[3] *See section 7 of Canvas/Files/Projects/P1Handout.pdf for instructions if you do not remember how to create a gzipped tar archive. Obviously, the filenames used for this project will be different.*

Grading Rubric

Your submission will be marked on a 30/50 point scale. Substantial emphasis is placed upon **WORKING** submissions, and you will note that a large fraction of the points are reserved for this aspect. It is to your advantage to ensure that whatever you submit compiles, links, and runs correctly. The information returned to you will indicate the number of points awarded for the submission.

You must be sure that your code works correctly on the virtual machine under VirtualBox, regardless of which platform you use for development and testing. Leave enough time in your development to fully test on the virtual machine before submission.

The marking scheme is as follows:

Points	Description
5	Your report – honestly describes the state of your submission
	If implemented using an array or singly-linked list
1	`longtest` and `stringtest`, linked to your set implementation, pass test 1
1	`longtest` and `stringtest`, linked to your set implementation, pass test 2
1	`longtest` and `stringtest`, linked to your set implementation, pass test 3
1	`longtest` and `stringtest`, linked to your set implementation, pass test 4
1	`longtest` and `stringtest`, linked to your set implementation, pass test 5
1	`longtest` and `stringtest`, linked to your set implementation, pass test 6
1	`longtest` and `stringtest`, linked to your set implementation, pass test 7
1	`longtest` and `stringtest`, linked to your set implementation, pass test 8
1	`longtest` and `stringtest`, linked to your set implementation, pass test 9
1	`longtest` and `stringtest`, linked to your set implementation, pass test 10
1	`longtest` and `stringtest`, linked to your set implementation, pass test 11
1	`longtest` and `stringtest`, linked to your set implementation, pass test 12
1	`longtest` and `stringtest`, linked to your set implementation, pass test 13
1	`longtest` and `stringtest`, linked to your set implementation, pass test 14
1	`longtest` and `stringtest`, linked to your set implementation, pass test 15
1	`longtest` and `stringtest`, linked to your set implementation, pass test 16
1	`longtest` and `stringtest`, linked to your set implementation, pass test 17
1	set.c successfully compiles
1	set.c successfully compiles with no warnings
1	There are no memory leaks in your program
5	The code could have worked with minor modification to the source.
	If implemented using a hash table
2	`longtest` and `stringtest`, linked to your set implementation, pass test 1
2	`longtest` and `stringtest`, linked to your set implementation, pass test 2
2	`longtest` and `stringtest`, linked to your set implementation, pass test 3
2	`longtest` and `stringtest`, linked to your set implementation, pass test 4
2	`longtest` and `stringtest`, linked to your set implementation, pass test 5
2	`longtest` and `stringtest`, linked to your set implementation, pass test 6
2	`longtest` and `stringtest`, linked to your set implementation, pass test 7
2	`longtest` and `stringtest`, linked to your set implementation, pass test 8
2	`longtest` and `stringtest`, linked to your set implementation, pass test 9
2	`longtest` and `stringtest`, linked to your set implementation, pass test 10

2	`longtest` and `stringtest`, linked to your set implementation, pass test 11
2	`longtest` and `stringtest`, linked to your set implementation, pass test 12
2	`longtest` and `stringtest`, linked to your set implementation, pass test 13
2	`longtest` and `stringtest`, linked to your set implementation, pass test 14
2	`longtest` and `stringtest`, linked to your set implementation, pass test 15
2	`longtest` and `stringtest`, linked to your set implementation, pass test 16
2	`longtest` and `stringtest`, linked to your set implementation, pass test 17
1	set.c successfully compiles
1	set.c successfully compiles with no warnings
1	There are no memory leaks in your program
8	The code could have worked with minor modification to the source.

Note that:

- Your report needs to be honest. Stating that everything works and then finding that it doesn't is offensive. The 5 points associated with the report are probably the easiest 5 points you will ever earn as long as you are honest.
- The points for "could have worked" is the maximum awarded in this category; your mark in this category may be lower depending upon how close you were to a working implementation.