Implementation of a Set ADT

Due at 11:59pm on Monday, 11 April 2022

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 a linked list or a hash table. If you implement it using 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. **Note that you MAY NOT implement the Set ADT using any other ADT other than Iterator; in particular, you may not use a Map ADT to implement the Set ADT.**

1 Requirements

1.1 Creating an instance of a `Set`

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

```
#include "hashset.h"
const Set *s = HashSet(freeValue, cmpFxn, 0L, 0.0, hashFxn);
```

or

```
#include "llistset.h"
const Set *s = LListSet(freeValue, cmpFxn);
```

The 1st argument to each constructor is a function that is used by `destroy()`, `clear()`, and `remove()` methods to return heap memory associated with set elements. This function is applied to each affected element; if the elements inserted into the set are ***NOT*** from the heap, you specify `doNothing` as the `freeValue` function argument. The `freeValue` argument has the signature

```
void (*freeValue)(void *element);
```

The 2nd argument 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`.

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

If you invoke HashSet(), the additional arguments have the following meaning:

- the 3rd argument is an indication of the initial capacity; if specified as 0L, a default capacity is used;
- the 4th argument 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 5th 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 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 one of the constructors 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.

`s->destroy(s);`
Destroys `s`; for each element in the set, the constructor-supplied `freeValue` function is applied; the storage associated with `s` is then returned to the heap; no return value.

`s->clear(s);`
Clears all elements from `s`; for each element in the set, the constructor-supplied `freeValue` function is applied; upon return, `s` will be empty; no return value.

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

`bool status = s->contains(s, value);`
Returns true if the `value` is contained in the set, false if not.

`bool value = s->isEmpty(s);`
Returns true if `s` is empty, false if not.

`bool status = s->remove(s, value);`
Removes the `value` from the set; the constructor-supplied `freeValue` function, is applied to that value before removing it from the set; returns true if `value` was present and removed, false 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 hashset.h

```
#ifndef _HASHSET_H_
#define _HASHSET_H_

/* BSD header removed to conserve space */

/*
 * constructor definition for generic hashset implementation
 */

#include "set.h"            /* dispatch table */

/*
 * create a HashSet
 *
 * freeValue is a function that returns any heap memory associated with
 * a set element to the heap; specify doNothing if the elements are not
 * from the heap
 *
 * cmp is used to determine equality between two objects, with
 * `cmp(first, second)' returning 0 if first==second, <>0 otherwise
 *
 * if capacity == 0L, the initial capacity is set to DEFAULT_SET_CAPACITY
 *
 * 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_SET_CAPACITY 16L
#define DEFAULT_LOAD_FACTOR 0.75

const Set *HashSet(void (*freeValue)(void *), int (*cmp)(void*, void*),
                   long capacity, double loadFactor,
                   long (*hashFunction)(void *, long)
                  );

#endif /* _HASHSET_H_ */
```

1.4 llistset.h

```
#ifndef _LLISTSET_H_
#define _LLISTSET_H_

/* BSD header removed to conserve space */

/*
 * constructor definition for generic llistset implementation
 */

#include "set.h"             /* dispatch table */

/*
 * create a LListSet
 *
 * freeValue is a function that returns any heap memory associated with
 * a set element to the heap; specify doNothing if the elements are not
 * from the heap
 *
 * cmp is used to determine equality between two objects, with
 * `cmp(first, second)' returning 0 if first==second, <>0 otherwise
 *
 * returns a pointer to the set, or NULL if there are malloc() errors
 */

const Set *LListSet(void (*freeValue)(void *), int (*cmp)(void*, void*));

#endif /* _LLISTSET_H_ */
```

1.5 set.h

```
#ifndef _SET_H_
#define _SET_H_

/* BSD header removed to conserve space */

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

/*
 * interface definition for generic set implementation
 */

typedef struct set Set;  /* forward reference */

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

/*
```

```
 * destroys the  set; applies freeValue function to each element
 * the storage associated with the set is then returned to the heap
 */
    void (*destroy)(const Set *s);

/*
 * clears all elements from the set; applies freeValue function to each element
 *
 * upon return, the set will be empty
 */
    void (*clear)(const Set *s);

/*
 * adds the specified element to the set if it is not already present
 *
 * returns true if the element was added, false if the element was already
 * present
 */
    bool (*add)(const Set *s, void *element);

/*
 * returns true if the set contains the specified element, false if not
 */
    bool (*contains)(const Set *s, void *element);

/*
 * returns true if set is empty, false if it is not
 */
    bool (*isEmpty)(const Set *s);

/*
 * removes the specified element from the set, if present
 *
 * applies freeValue function to element before removing it from the set
 *
 * returns true if successful, false if not present
 */
    bool (*remove)(const Set *s, void *element);

/*
 * returns the number of elements in the set
 */
    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'
 *
 * N.B. the caller is responsible for freeing the array of void* pointers
 *      when finished with it.
 */
    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 `P0start.tgz`; this file contains the following files:

- `hashset.h, llistset.h, set.h` – the files shown above
- `hashset.c` and `llistset.c` - stub implementations
- `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` files to create the `longtest` executable image; it must also be linked with `stringtest.o` and your `*set.o` files to create the `stringtest` executable image
- `Makefile` – a makefile for creating your `longtest` and `stringtest` executable image files[2]

3 Hints

If you are using a linked list, I suggest that you thoroughly review the Deque implementation in Appendix B of `Canvas/Files/Readings/Sventek-CaDS21F.pdf` (CaDS) to see how doubly-linked lists using a sentinel node are programmed; and if you are using a hash table, I suggest that you thoroughly review Chapter 12 on Maps, especially sections 12.4 and 12.5, and Appendix B, of CaDS.

4 Checking that your solution works correctly

The synopses for `longtest` and `stringtest` are:

[2] Note that the Makefile links both hashset.o **and** llistset.o with sort.o to create longtest and stringtest. The stub implementations of hashset.c and llistset.c will compile without error or warning as is. If you choose to implement the hashset, you are expected to modify hashset.c; if the llistset, you are expected to modify llistset.c. Do not modify the other implementation file, but it must be present in your working directory so that make can compile it and link it with your implementation.

```
./longtest -l|-h test# . . .
./stringtest -l|-h test# . . .
```

`longtest` and `stringtest` each take two or more arguments; the first argument, an option, indicated whether you have implemented `hashset.c` or `llistset.c`; each following 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 -h|-l 1
# if the output from the above command indicates success, then
valgrind ./longtest -h|-l 1
```

If the result of test 1 indicates *failure*, or if `valgrind` reports warnings or lost memory, track down the cause in your `*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 -h|-l 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 -h 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
Test sorting of elements  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)
```

where the above dialog assumes that `longtest` has been invoked with the `-h` option.

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

5 Submission[3]

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

Your TGZ archive should be named `<duckid>-project0.tgz`, where `<duckid>` is your "duckid". It should contain your file, either `hashset.c` or `llistset.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 a linked list or a hash table**. The archive should ***not*** include `hashset.h, llistset.h, set.h, sort.h, sort.c, longtest.c, stringtest.c,` or `Makefile`.

6 A professional way to create your TGZ archives

As described in CaDS section 2.7.3.1, you can create your TGZ archive using a command of the form:

```
tar –zcvf duckid-project0.tgz report.txt hashset.c
```

OR

```
tar -zcvf duckid-project0.tgz report.txt llistset.c
```

Of course `'duckid'` above has to be replaced by **your** DuckID. This a lot to type, and future assignments might require that you put 10-15 files into an archive. By creating a manifest file, one can simplify the creation of the TGZ file, and avoid including extraneous files.

A **manifest**, customs **manifest,** or cargo document is a document listing the cargo, passengers, and crew of a ship, aircraft, or vehicle, for the use of customs and other officials.[4] In software terms, a manifest is a list of files that should be included in an archive.

We recommend that you put all of your files for each project in a separate directory in your Linux image; `project0` would seem to be a good name for this project, but you may name it anything you wish. You should probably put the contents of `P0start.tgz` from Canvas in this directory, and should edit `hashset.c (or llistset.c)` and `report.txt` files there, as well.

Since the submitted archive has to contain files named `hashset.c (or llistset.c) and report.txt`, you should create a file in this directory named `manifest` that contains the following lines (the example here assumes you are submitting the `hashset.c` implementation):

```
report.txt
hashset.c
```

Armed with the `manifest`, `report.txt`, `and hashset.c/llistset.c` files, creation of the TGZ file becomes

---

[3] ***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 no points.***

[4] This definition is taken from Wikipedia.

```
tar –zcvf duckid-project0.tgz $(cat manifest)
```

The `$(cat manifest)` expression causes the shell to invoke `cat` on the file named `manifest`; instead of displaying the contents on the standard output, the output from `cat` is captured by the shell, and it merges all of the lines into a sequence of blank separated words; this merged list then provides the additional arguments given to `tar`.

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 a doubly-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	llistset.c successfully compiles
1	llistset.c successfully compiles with no warnings
1	There are no memory usage errors or 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	hashset.c successfully compiles
1	hashset.c successfully compiles with no warnings
1	There are no memory usage errors or 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.