CSE 333 – Section 4

MultiSet, SortableArray, and File I/O

Johnny Yan

Department of Computer Science & Engineering
University of Washington

October 17, 2013

Administrivia

- Exercise 4 graded, feedback returned via email
- Exercise 5 & 6 will be graded together
- Next section is a quiz section

Buggy 1

MultiSet.h

```
typedef struct {
  size_t  size;
  int valueVec[20];
} MultiSet;
```

MultiSet.c

```
MultiSet *multiset_new(size_t size, int *elements) {
  int i;
  MultiSet *this = (MultiSet*) malloc(sizeof(MultiSet));

  if (elements != NULL) {
    for (i = 0; i < size; i ++) {
      this->valueVec[i] = elements[i];
    }
  }
  return this;
}

void multiset_destroy(MultiSet *this){
  free(this);
}
```

Buggy 2

MultiSet.h

```c
typedef struct {
  size_t  size;
  int *valueVec;
} MultiSet;
```

MultiSet.c

```c
MultiSet *multiset_new(size_t size, int *elements){
  MultiSet *newMulti = (MultiSet *)malloc(sizeof(MultiSet));
  newMulti->size = size;
  int *allElements = malloc(sizeof(int) * size);
  for(int i=0; i<size; i++){
    allElements[i] = elements[i];
  }
  newMulti->valueVec = allElements;
  free(allElements);

  return newMulti;
}

void multiset_destroy(MultiSet *this){
  free(this);
}
```

Buggy 3

MultiSet.c

```c
MultiSet* multiset_new(size_t size, int *elements) {
  int *array = (int*) malloc(size * sizeof(int));
  array = elements;
  MultiSet *result = (MultiSet*) malloc(sizeof(MultiSet));
  result->size = size;
  result->valueVec = array;
  return result;
}
```

Buggy 4

MultiSet.c

```c
MultiSet *multiset_new(size_t size, int *elements) {
  MultiSet *multiSet = (MultiSet*)malloc(sizeof(MultiSet));
  int *valueVec = (int*)malloc(size * sizeof(int));
  for (size_t i = 0; i < size; i++) {
    valueVec[i] = elements[i];
  }
  multiSet->size = size;
  multiSet->valueVec = valueVec;
  return multiSet;
}

MultiSet multiset_union(MultiSet *A, MultiSet *B) {
  MultiSet *C = multiset_new(A->size + B->size, B->valueVec);
  for (size_t i = 0; i < A->size; i++) {
    C->valueVec[i] = A->valueVec[i];
  }

  for (size_t i = 0; i < B->size; i++) {
    C->valueVec[i + A->size] = B->valueVec[i];
  }

  return *C;
}
```

Buggy 5

MultiSet.c

```c
MultiSet *multiset_new(size_t size, int *elements) {
  MultiSet *m = (MultiSet *) malloc(sizeof(size_t) + (size * sizeof(int)));
  m->size = size;
  for(int i = 0; i < size; i++)
    m->valueVec[i] = elements[i];

  return m;
}

void multiset_destroy(MultiSet *this) {
  free(this);
}

MultiSet multiset_union(MultiSet *A, MultiSet *B) {
  size_t newsize = A->size + B->size;
  MultiSet C = {newsize, {}};
  int i;
  for(i = 0; i < A->size; i++)
    C.valueVec[i] = A->valueVec[i];

  for(; i < C.size; i++)
    C.valueVec[i] = B->valueVec[i - A->size];

  return C;
}
```

Buggy 5 cont.

MultiSet.h

```c
typedef struct {
  size_t  size;
  int valueVec[20];
} MultiSet;
```

Inspired a new way of implementation!

Buggy 5 cont.

MultiSet.h

```c
typedef struct {
  size_t  size;
  int valueVec[20];
} MultiSet;
```

Inspired a new way of implementation!

What is SortableArray?

- How do we design SortableArray, data structure and interface?
- What is our goal?

SortableArray struct

```c
typedef struct sortable_array_t {
  unsigned int  size;
  void**        data;
} *SortableArray;
```

- The size field is required by the documentation
- Array of void pointers to achieve generic

Interface

SortableArray.h

```c
#ifndef _SORTABLEARRAY_H
#define _SORTABLEARRAY_H

#include <stdbool.h>

struct sortable_array_t;
typedef struct sortable_array_t *SortableArray;

SortableArray SortableArray_new(unsigned int size);
bool SortableArray_delete(SortableArray sa);
unsigned int SortableArray_size(SortableArray sa);
bool SortableArray_set(SortableArray sa,
                       unsigned int index,
                       void *dataIn);  // returns data
bool SortableArray_get(SortableArray sa,
                       unsigned int index,
                       void **dataOut);

// SortableArray_Map invokes a caller-supplied function on each
// element of the array.  It guarantees to iterate over the elements
// in index order.  The user supplied callback function should
// return false to continue the iteration, and true to stop it.
typedef bool (*SortableArray_Map_Callback)(void *el);
void SortableArray_map(SortableArray sa, SortableArray_Map_Callback callback);

// A SortableArray_Comparator function returns -1 if a < b,
// 0 if a == b, and 1 if a > b
typedef int (*SortableArray_Comparator)(const void *a, const void *b);
bool SortableArray_sort(SortableArray sa, SortableArray_Comparator compare);

#endif // _SORTABLEARRAY_H
```

Implementation

SortableArray.c

```
SortableArray SortableArray_new(unsigned int size) {
  if ( size == 0 ) return NULL;
  SortableArray sa = (SortableArray)malloc(sizeof(struct sortable_array_t));
  if ( sa == NULL ) return NULL;
  sa->size = size;
  sa->data = (void**)malloc(sizeof(void*) * size);
  return sa;
}

void SortableArray_map(SortableArray sa, SortableArray_Map_Callback callback) {
  int i;
  if ( sa == NULL ) return;
  for ( i=0; i<sa->size; i++ ) callback(sa->data[i]);
}

bool SortableArray_sort(SortableArray sa, SortableArray_Comparator compare) {
  if ( sa == NULL ) return false;
  qsort(sa->data, sa->size, sizeof(void*), compare);
  return true;
}
```

main.c

```
int string_compare(const void *a, const void *b) {
  return strcmp( *(const char**)a, *(const char**)b);
}
```

Stdio I/O

```
#include <stdio.h>
FILE *fopen(const char *path, const char *mode);
int fclose(FILE *fp);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- *fopen* opens the file whose name is the string pointed to by *path* and associates a stream with it.
- *mode* can be r–read, w–write or a–append combined with b–binary or t–text. All POSIX conforming systems, including Linux, will ignore b. Adding mode b may be a good idea to make your program portalbe to non-UNIX environments.
- *fclose* flushes the stream pointed to by *fp* and closes the underlying file descriptor.
- *fread/fwrite* reads/writes *nmemb* elements of data, each *size* bytes long, from/to the stream pointed to by *stream*, storing them at/obtaining them from the location given by *ptr*.

Stdio I/O cont.

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
int fflush(FILE *stream);
```

- *fseek* sets the file position indicator for the stream pointed to by *stream*. The new position, measured in bytes, is obtained by adding *offset* bytes to the position specified by *whence*. If *whence* is set to *SEEK_SET*, *SEE_CUR*, or *SEEK_END*, the *offset* is relative to the start of the file, the current position indicator, or end-of-file, respectively.
- For output streams, *fflush* forces a write of all user-space buffered data for the given output or update stream via the stream's underlying write function. For input streams, *fflush* discards any buffered data that has been fetched from the underlying file, but has not been consumed by the application. The open status of the stream is unaffected. If the *stream* argument is NULL, *fflush* flushes all open output streams.

System I/O

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

#include <unistd.h>
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

- Similar to their stdio counterparts, but at system level, non-buffered
- Notice that number of bytes read/write is not necessary the same as desired.