A Dynamic Array Class

Version 1b

The *DA* class

For your dynamic array class, the header file, *da.h*, should look like:

```
#ifndef __DA_INCLUDED__
#define __DA_INCLUDED__

#include <stdio.h>

typedef struct da DA;

extern DA   *newDA(void);
extern void  setDAdisplay(DA *,void (*)(void *,FILE *));
extern void  setDAfree(DA *,void (*)(void *));
extern void  insertDA(DA *items,int index,void *value);
extern void *removeDA(DA *items,int index);
extern void  unionDA(DA *recipient,DA *donor);
extern void *getDA(DA *items,int index);
extern void *setDA(DA *items,int index,void *value);
extern int   sizeDA(DA *items);
extern void  displayDA(DA *items,FILE *fp);
extern int   debugDA(DA *items,int level);
extern void  freeDA(DA *items);

#define insertDAback(items,value) insertDA(items,sizeDA(items),value)
#define removeDAback(items)       removeDA(items,sizeDA(items)-1)
#endif
```

The header file contains the function signatures of your public methods while the code module, *da.c*, contains their implementations.

The only local include that *da.c* should have is *da.h*.

Method behavior

Here are some of the behaviors your methods should have. This listing is not exhaustive; you are expected, as a computer scientist, to complete the implementation in the best possible and most logical manner. The dynamic array uses zero-based indexing.

- *newDA* - The constructor returns an initialized *DA* object.
- *setDAdisplay* - This method is passed a function that knows how to display the generic value stored in an array slot.
- *setDAfree* - This method is passed a function that knows how to free the generic value stored in an array slot.
- *insertDA* - The insert method places the given item at the slot named by given index. The previous item at that slot shifts to the next higher slot (and so on). If there is no room for the insertion, the array grows by doubling. It should run in amortized constant time when inserting an item a constant distance from the back and in linear time otherwise.
- *removeDA* - The remove method removes the item at the given index. The item at the next higher slot shifts to that slot (and so on). The method returns the removed item. If the ratio of the size to the capacity drops below 25%, the array shrinks by half. The array should never shrink below a capacity of one. Also, an empty array should always have a capacity of one. It should run in amortized constant time when removing an item a constant distance from the back and in linear time otherwise.
- *unionDA* - The union method takes two array and moves all the items in the donor array to the recipient array. If the recipient array has the items `[3,4,5]` and the donor array has the items `[1,2]`, then, after the union, the donor array will be empty and recipient array will have the items `[3,4,5,1,2]`. It should run in amortized linear time with respect to the size of the donor array.
- *getDA* - The get method returns the value at the given index. It should run in constant time.
- *setDA* - If the given index is equal to the size, the value is inserted via the insert method. The method returns the replaced value or the null pointer if no value was replaced. It should run in constant or amortized constant time (in the case of the insert method being called).
- *sizeDA* - The size method returns the number of items stored in the array. It should run in amortized constant time.

- *displayDA* - This visualizing method prints out the filled region, enclosed in brackets and separated by commas. If the integers 5, 6, 2, 9, and 1 are stored in the array (listed from index 0 upwards) and the debug level is zero (see *debugDA*), the method would generate this output:

  ```
  [5,6,2,9,1]
  ```

  with no preceding or following whitespace. If no display method is set (see *setDAdisplay*), the address of each item is printed (using `%p`). Each address is preceded by an @ sign.
- *debugDA* - The debug method sets an internal flag in the object to the given value. If the flag is greater than zero, the display method adds a comma (if needed) and the number of empty slots (in brackets) immediately after the last element. If the above array had a capacity of 8, the display would be `[5,6,2,9,1,[3]]`. An empty array with capacity 1 displays as `[[1]]`. The method returns the previous debug value.
- *freeDA* - If no free method is set, the individual items are not freed. In any case, the array and its supporting allocations are freed.

Constraints

The array should never have a capacity of 0; An empty array should have a capacity of 1.

If there is no room for an insertion, the array grows by doubling.

If the ratio of the size (i.e. the number of items in the array) to the capacity drops below 25%, the array shrinks by half.

Assertions

Include the following assertions in your methods:

- *newDA* - The memory allocated shall not be zero.
- *insertDA* - The memory allocated shall not be zero.
- *removeDA* - The size shall be greater than zero.
- *getDA* - The index shall be greater than or equal to zero and less than the size.
- *setDA* - The index shall be greater than or equal to zero and less than or equal to the size.

Testing your DA class

Here is a sample testing program that uses the *displayINTEGER* function found in the *integer* class. Note how it is passed to the *newDA* constructor.

```
#include <stdio.h>
#include <stdlib.h>
#include "integer.h"
#include "da.h"

static void showItems(DA *items);

int
main(void)
    {
    srandom(1);
    DA *items = newDA();
    setDAfree(items,freeINTEGER);
    showItems(items);
    insertDA(items,0,newINTEGER(3));                 //insert at front
    insertDA(items,sizeDA(items),newINTEGER(2));     //insert at back
    insertDA(items,1,newINTEGER(1));                 //insert at middle
    showItems(items);
    printf("The value ");
    INTEGER *i = removeDA(items,0);                  //remove from front
    displayINTEGER(i,stdout);
    printf(" was removed.\n");
    freeINTEGER(i);
    showItems(items);
    int x = getINTEGER((INTEGER *) getDA(items,0));  //get the first item
    printf("The first item is %d.\n",x);
    printf("Freeing the list.\n");
```

```
        freeDA(items);
        return 0;
        }

    static void
    showItems(DA *items)
        {
        int old;
        setDAdisplay(items,displayINTEGER);
        printf("The items are ");
        displayDA(items,stdout);
        printf(".\n");
        printf("The items (debugged) are ");
        old = debugDA(items,1);
        setDAdisplay(items,0);
        displayDA(items,stdout);
        printf(".\n");
        debugDA(items,old);
        }
```

The output of this program should be:

```
    The items are [].
    The items (debugged) are [[1]].
    The items are [3,1,2].
    The items (debugged) are [@0x420c4d8,@0x420c580,@0x420c510,[1]].
    The value 3 was removed.
    The items are [1,2].
    The items (debugged) are [@0x420c580,@0x420c510,[2]].
    The first item is 1.
    Freeing the list.
```

You may download the integer class with these commands:

```
    wget beastie.cs.ua.edu/cs201/testing/integer.c
    wget beastie.cs.ua.edu/cs201/testing/integer.h
    wget beastie.cs.ua.edu/cs201/testing/makefile
    wget beastie.cs.ua.edu/cs201/testing/test-integer.c
```

The last command retrieves a program for testing the integer class. Compile and run this program with the commands:

```
    make test
```

You should model all your classes and makefiles after the the integer class.

The superior student will test his or her dynamic array class with other data types besides *INTEGER*s by implementing such classes as *REAL* and *STRING*.

Change log

`1b`	fixed erroneous calls to sizeCDA in the .h macros
`1a`	clarified the explanation of *displayDA*