[Solutions to this homework assignment are due at the start of class on Thursday, March 14, 2013, or at noon the same day if submitted by email. Please be sure to follow the homework submission guidelines that are included on the class web page.]

Introduction: This assignment is about the implementation of dynamic memory allocation and garbage collection. Ideally, it would be nice to study these topics in the context of a real-world, practical implementation. However, even a fairly simple copying collector implementation for the Mini compiler requires attention to more details than we have time for in this assignment, so we will make do instead with a less realistic, but simpler implementation that, nevertheless, still illustrates some of the key ideas.

To start this assignment, you should download the file `hw6.zip` from the class web page. This file contains the skeleton of an implementation for a two space, copying garbage collector that you will work with in the rest of this assignment. The most important file is called `Heap.java`, and it contains the code for a corresponding class called `Heap`. Roughly speaking, the `Heap` class provides the following interface, details of which are explained in the following text:

```
class Heap {
  public static Heap make(int size);
  int alloc(int len);
  public void garbageCollect();
  public int load(int obj, int offs);
  public void store(int obj, int offs, int val);
  public void dump();
  int freeSpace();
}
```

Some key points:

- A *heap* is used to store a set of *objects*, each of which spans one or more words. A new heap with the capacity to store a total of `S` words can be created by calling the method `Heap.make(S)`. The first word in each object is a *length*, which specifies how many *fields* that object has. These fields are stored in consecutive words immediately after the initial word. So an object that has `len` fields will span `(len+1)` consecutive words in the heap.
- We can allocate heap space for a new object with `len` fields by calling the method `alloc(len)`, which also ensures that all of the fields in the new object are initialized to zero.
- If there is not enough room in the heap for an object with `len` fields, then `alloc(len)` will call the `garbageCollect()` method in an attempt to recover and reclaim any portions of the heap that are no longer in use. The `garbageCollect()` method can also be called directly to force a garbage collection when the heap is not full. In fact, the version of this method in `Heap.java` simply displays an error message that garbage collection is not supported and then terminates the program.
- The value of the `i`th field in an object at address `obj` can be read by calling `load(obj, i)`. The value of the `i`th field in an object at address `obj` can be set to `val` by calling `store(obj, i, val)`. These methods perform some limited checking on the validity of the `obj` and `i` arguments, and will abort the program if the values appear to be incorrect.
- The values that are stored in heap objects are all integers, but negative integers in the range `S`, ..., `-1` are interpreted specially as pointers to the individual locations (assuming a heap with `S` words). Positive numbers are not interpreted in any special way.

- The `dump()` method can be used to print a description for each of the objects in the current heap; the `freeSpace()` method can be used to determine the number of free (unused) words in the current heap.

The following program shows how (most of) these operations can be used in simple example program (the source code for this is included in `hw6.zip` as `TestHeap1.java`):

```
class TestHeap1 {
  static final int S = 100;

  public static void main(String[] args) {
    Heap h    = Heap.make(S);
    int root  = h.alloc(5);
    h.store(root, 1, h.alloc(3));
    h.store(root, 2, h.alloc(2));
    h.dump();
    System.out.println("Free space remaining = "
                           + h.freeSpace());
  }
}
```

The output from this program is as follows:

```
Object at address -100, length=5, data=[-94, -90, 0, 0, 0]
Object at address -94, length=3, data=[0, 0, 0]
Object at address -90, length=2, data=[0, 0]
Heap allocation pointer: 13
Free space remaining = 87
```

Before proceeding, be sure that you understand this output, checking with the source in `Heap.java` where necessary for more details about the implementation of the heap operators. (You may also find it useful to draw a diagram illustrating the structure of the heap that is constructed in the program above.)

Question 1: Consider the following program (source code in `TestHeap2.java`):

```
class TestHeap2 {
  static final int S = 100;
  static final int N = 6;

  public static void main(String[] args) {
    Heap h = Heap.make(S);
    for (int i=0; i<N; i++) {
      System.out.println("Allocating object " + i
                         + " at address " + h.alloc(8));
    }
    h.dump();
    System.out.println("Free space remaining = "
                       + h.freeSpace());
  }
}
```

What is the smallest value of `N` that will cause this program to report a fatal error due to lack of available memory in the heap? What behavior would you expect if the implementation of `Heap` included a working

garbage collector? Are these observations true for all values of N and S? (Note: In answering each of these questions, be sure to include appropriate explanation or justification.) **(6)**

Question 2: The file TwoSpace.java contains the skeleton of an implementation for a garbage collected heap, using a two space, copying collector in the style that was discussed in class. The TwoSpace constructor allocates memory for the toSpace array into which reachable objects will be forwarded; the heap array that is inherited from Heap plays the role of the fromSpace. The supplied version of the code, however, is missing the implementation of two key functionsforward() and \verbscavenge()that perform essentially the same roles as the functions of the same names that were described in class. Note, however, that instead of using a special tag value, FORWARDED, to mark objects in fromSpace that have already been forwarded, our implementation simply stores the toSpace address of the forwarded object in the initial word of the object. Such objects are easily distinguished from objects that have not been forwarded: pointers to specific locations in the heap are represented by negative numbers, which are easily distinguished from the positive len values at the beginning of an unforwarded object. Using these ideas, complete the implementation of TwoSpace.java.

Be sure to subject your implementation to careful and appropriate testing. Note, in particular, that you can test your TwoSpace implementation by: changing the code in the Heap.make() method to use the TwoSpace constructor in place of the Heap constructor; recompiling the sources (using javac *.java); and then running test programs (e.g., java TestHeap2). **(24)**

Question 3: Consider the following test program, a modified version of TestHeap1.java (source code for this program is in TestHeap3.java):

```
class TestHeap3 {
  static final int S = 100;

  public static void main(String[] args) {
    Heap h = Heap.make(S);
    h.alloc(3);
    h.a = h.alloc(6);
    h.alloc(3);
    h.store(h.a, 1, h.alloc(5));
    h.alloc(5);
    h.store(h.a, 2, h.alloc(4));
    h.alloc(2);

    h.dump();
    System.out.println("Free space remaining = "
                       + h.freeSpace());

    h.garbageCollect();

    h.dump();
    System.out.println("Free space remaining = "
                       + h.freeSpace());
  }
}
```

What aspects of the behavior of our garbage collector are illustrated by the output that is produced by running this program? **(2)**

The expression h.a in the code above refers to a field a of the heap object h that is included as a root in the

implementation of `garbageCollect()` in `TwoSpace.java`. Explaining your method, construct a lightly modified version of the above program to demonstrate how the garbage collector can fail if we use a simple local integer variable `t` in place of `h.a`. **(4)**

The code for `alloc()` in `Heap.java` includes a loop to ensure that the fields in each newly allocated object are initialized to zero:

```
for (heap[hp++]=len; len>0; len--) {
  heap[hp++] = 0;
}
```

How might the garbage collector fail if we were to omit this code? **(4)**

Question 4: Consider the following program (sources in `TestHeap4.java`):

```
class TestHeap4 {
  static final int S = 100;
  static final int N = 10;

  public static void main(String[] args) {
    Heap h = Heap.make(S);
    for (int i=0; i<N; i++) {
      int t = h.alloc(1+i);
      System.out.println("Allocating object " + i
                         + " at address " + t);
      h.store(t, 1, h.a);
      h.a = t;
    }
    System.out.println("Before garbage collection;");
    h.dump();
    System.out.println("Free space remaining = "
                       + h.freeSpace());

    h.garbageCollect();

    System.out.println("After garbage collection;");
    h.dump();
    System.out.println("Free space remaining = "
                       + h.freeSpace());
  }
}
```

How does the behavior of this program vary in response to changes in the values of `S` and `N`? **(4)**

What effect does the `garbageCollect()` call in this program have on heap layout? **(2)**

Show that it is possible to construct cyclic structures in the heap and that these structures are preserved by the copying garbage collector implementation. **(4)**

End!