Ben Reichert
June 3rd, 2015
CS322 Compilers II

Homework IIII

Question 1:

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?

We have a size S of 100, and N objects. Since we can generalize the formula for finding the amount of N objects as N = S/(objLength+1) we can calculate the max number of allocated N objects before we overfill the heap and are forced to do garbage collection. In this case we can have a max N=11, and a N=12 results in heap overflow. If we had working garbage collection, we would forward and scavenge the objects, and remove the old garbage from the new heap to create space, and thus give us room for allocating the new object. If there was no garbage, we would return an error that no memory on the heap can be filled (something like an Out Of Memory error). These observations are seemingly correct for all positive, valid values of N and S.

Question 2:

forward and scavenge code follows:

```
class TwoSpace extends Heap {
  ...

    /** Forward a reachable object from the current heap into toSpace.
   */
  private int forward(int obj) {
    // There are three cases that you will need to handle here.
    // 1) If obj is not a pointer (i.e., if it is greater than
    //    or equal to zero), then you should just return obj
    //    directly.
    if(obj >= 0){
      return obj;
    }

    if(obj < 0){
    // 2) If obj is a pointer, but the length field of the object
    //    that it points to is negative, then (a) we can conclude
    //    that the object has already been forwarded; and (b) we
    //    can just return the (negative) length value, which is a
    //    pointer to the object's new location in toSpace.
            //a
            int len = heap[obj+size]; //forwarded
            if(len < 0){
                //b
```

```
                return len;
            }

            else if(len >= 0){
            // 3) If obj is a pointer and the length field is non negative,
            //    then it points to an object that needs to be forwarded to
            //    toSpace.  This requires us to copy the length word and
            //    the associated fields from the heap into the toSpace.
            //    After that, we overwrite the length field with a pointer
            //    to the location of the object in toSpace (because it is
            //    a pointer, this will be a negative number).
            // The description here is longer than my code!
            int toSpaceLocation = hp;

              //need to forward
              //obj is negative since it is a pointer
              for(int i = 0; i < len+1; i++){
                //take what that part of the object in the heap and put it into the toSpace
                toSpace[hp++] = heap[obj+size+i];
                //copy element to new heap
              }
              //overwite length in heap to pointer in toSpace
              heap[obj+size] = toSpaceLocation - size;

              //need to return address in relation to toSpace
              obj = toSpaceLocation - size;

            }
        }

   return obj;
  }

  /* Scavenge an object in toSpace for pointers to reachable objects
   *  that are still in the current heap.  Return the total number of
   *  words that are used to represent the object, which is just the
   *  total number of fields plus one (for the length field at the
   *  start of the object).
   */
  private int scavenge(int obj) {
    int len = toSpace[obj];

    for(int i = 0; i < len+1; i++){
        //we could do a check to see if obj is a ptr here, but we already do that
        //check in forward, so we have some additional overhead of calling it on every
        //object, but the code is easier to read.
        //gotta do the offset
        toSpace[obj+i] = forward(toSpace[obj+i]);
    }
    // Scan the fields in this object, using forward on
    // any pointer fields that we find to make sure the
    // objects that they refer to are copied into toSpace.
    return 1+len;
  }
}
```

I used TestHeap3 and TestHeap4 to verify that the register a references an object, and reachable objects get scavenged. In the case of TestHeap3, we have a reference to an array of size 9 in register a:

```
Object at address -96, length=9, data=[-82, -70, 0, 0, 0, 0, 0, 0, 0]
```

and two objects at addresses -82 and -70 respectively:

```
Object at address -82, length=5, data=[0, 0, 0, 0, 0]
Object at address -70, length=4, data=[0, 0, 0, 0]
```

which should be scavenged when run. Everything else is garbage and can be removed from the new heap since it is unreachable. We see this in the output of the new heap after forwarding and scavenging:

```
breic2@ada:~/homework/322/hw4$ java TestHeap3
Object at address -100, length=3, data=[0, 0, 0] #garbage
Object at address -96, length=9, data=[-82, -70, 0, 0, 0, 0, 0, 0, 0]
Object at address -86, length=3, data=[0, 0, 0] #garbage
Object at address -82, length=5, data=[0, 0, 0, 0, 0]
Object at address -76, length=5, data=[0, 0, 0, 0, 0] #garbage
Object at address -70, length=4, data=[0, 0, 0, 0]
Object at address -65, length=2, data=[0, 0] #garbage
Heap allocation pointer: 38
Free space remaining = 62
Object at address -100, length=9, data=[-90, -84, 0, 0, 0, 0, 0, 0, 0]
Object at address -90, length=5, data=[0, 0, 0, 0, 0]
Object at address -84, length=4, data=[0, 0, 0, 0]
Heap allocation pointer: 21
Free space remaining = 79
#yay we get the expected output
```

Question 3A:

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

We can see from the stdout that the garbage collector is removing objects that are not reachable. We can see that objects in the four registers a,b,c,d and their pointers to other objects are scavenged, and everything else is left behind in the garbage collection process. We can also see that we replace the original heap with the new reduced toSpace heap during the swap and the end of garbage collection.

Question 3B:

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.

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

  public static void main(String[] args) {
    Heap h = new TwoSpace(S);
    h.alloc(3);
    int t = h.alloc(9); //new alloc that doesnt save in register
    //h.a = h.alloc(9); original alloc that saves in a
    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());
  }
}
```

Since we are not saving the result of the alloc() method into the local register a, we lose the reference to the object in t and never save a pointer to it. Therefore we create unusable allocated space, but since t is the only reference to it, we can't access it without saving t's value in a register. Running the Fail program results in the error:

```
Fatal error: Invalid object reference
```

This is because we lost the reference to the place of allocation in the heap and aren't sure where to allocate the next object when needed.

Question 4A

How does the behavior of this program vary in response to changes in the values of S and N?

When the size of the heap (S) is changed, you can allocate more objects on the heap of the same size. N is the number of objects that you allocate and if it violates N = S/(objLength+1) then you cannot allocate that many objects and therefore must do garbage collection. Allocation of increasingly sized objects of size 1+i is happening in the for loop. As we increase N, we have a growing need for space (S) with each increasing object. Each object requires one more element than the previous and therefore we must account for this in the size of of the heap as S.

Question 4B:

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

The garbageCollect() method in this program reverses the heap due to the fact that we forward registers and then scavenge. The register a points to the last heap element, so when we scavenge we travel backwards due to pointers are reverse the order of the heap.

Question 4C:

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

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

  public static void main(String[] args) {
    Heap h = new TwoSpace(S);
    int t = h.alloc(1);     //create two 1 element objects
    int z = h.alloc(1);
    h.store(t, 1, z);  //store oppositional pointers to each object so we get cyclic action
    h.store(z, 1, t);      //so z -> a -> z -> a…...
    h.a = t;
    h.a = z;

    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());
  }
}
```

Preserved structures:

```
breic2@ada:~/homework/322/hw4$ java Cyclic
Before garbage collection;
Object at address -100, length=1, data=[-98]
Object at address -98, length=1, data=[-100]
Heap allocation pointer: 4
Free space remaining = 96
After garbage collection;
Object at address -100, length=1, data=[-98]
Object at address -98, length=1, data=[-100]
Heap allocation pointer: 4
Free space remaining = 96
```

We break the cyclic motion when forwarding and scavenging, but create a new cyclic toSpace so the original cyclic structure is preserved.