CS558 Programming Languages – Fall 2023 – Suggested Study Question Solutions for Lecture 3b

1. Fragmentation can arise in the heap because deallocations may occur in arbitrary order relative to allocations, leading to the possibility of "holes" that are too small individually to satisfy an allocation request, even if the sum of their sizes is sufficient. Suppose we have a heap of size 1024 bytes, and we execute the following code in C (where `malloc` requests bytes of memory from the heap and `free` returns it):

```
char *a = malloc(256);   // allocates at heap offset 0
char *b = malloc(512);   // allocates at heap offset 256
char *c = malloc(256);   // allocates at heap offset 768; heap is full
free(a);                 // deallocates at offset 0; heap has 256 free bytes
free(c);                 // deallocates at offset 768; heap has 512 free bytes, but
char *d = malloc(512);   // this request fails because those bytes are not contiguous
```

This problem cannot occur with the stack, because all the free space in a stack is *always* contiguous. This is a result of the fact that stack deallocations always occur in reverse order of allocations.

2. Since `ST` values are not boxed, the semantics of assignment is to make a copy of each field. Thus, changing the value of the `D` field in `st2` does not affect `st1`. On the other hand, since `CL` values *are* boxed, the `C` field is effectively a pointer to the contents of `cl`, so changes to `cl` are visible in both `st1` and `st2`. So we get this:

```
st1.C.A = 13 st1.C.B = 2 st1.D = 3
st2.C.A = 13 st2.C.B = 2 st2.D = 33
```

If you want to play with C# code without installing the language on your machine, try `https://dotnetfiddle.net/`.

3. Here is a simple example, exercised by running `Foo.main()`. Since each call to the constructor of `P` allocates a fresh object at a new address, `a` and `b` are structurally equal, but not reference equal.

```
case class P(x:Int)
object Foo {
  def main() = {
    val a = P(1)
    val b = P(1)
    println("are a and b structurally equal? answer:" + (a==b));
    println("are a and b reference equal? answer:" + a.eq(b))
  }
}
```

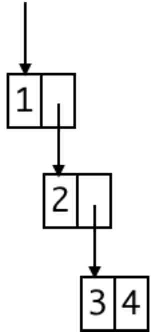
4. (a) Following the suggestion on slide 23, we can write a 4-tuple using nested pairs, nesting either to the right:

```
(1.(2.(3.4)))
```

or to the left:

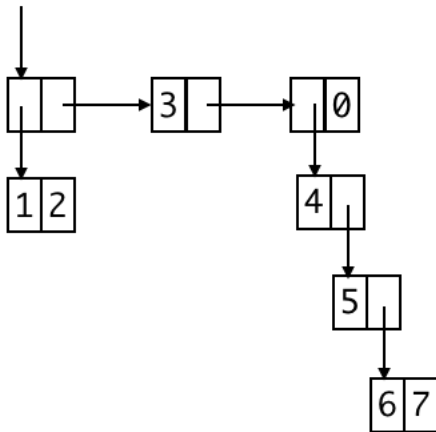
```
(((1.2).3).4)
```

The right-nested version looks like this as a tree (the left-nested version is similar):



(b) Using right-nested 4-tuples, we get

```
((1.2).(3.((4.(5.(6.7))).0)))
```



Notice the difference between the encoding of fixed-length records and arbitrary-length lists.