MCS 360 Exam 1 14 October 2015

Name: _____

- Do not start until instructed to do so.
- In order to get full credit, you need to show your work.
- You have 50 minutes to complete the exam.
- Good Luck!

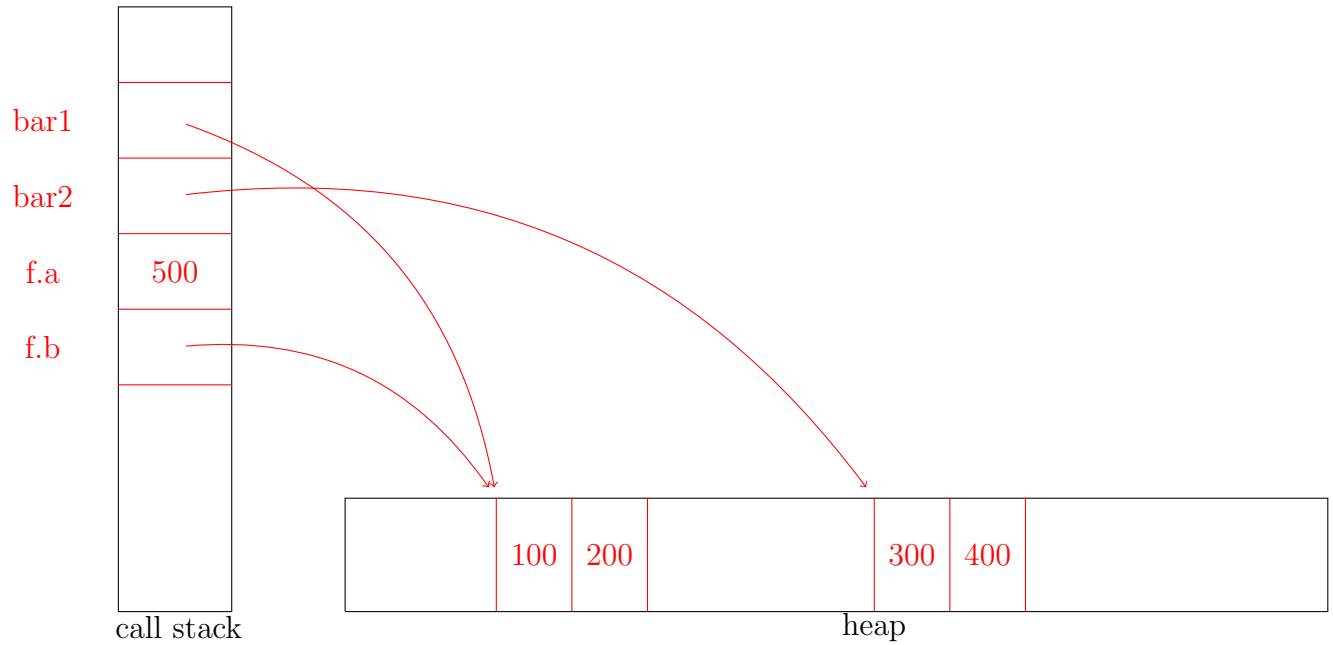
Problem 1 _____/15
Problem 2.a _____/15
Problem 2.b _____/10
Problem 2.c _____/25
Problem 3 _____/20
Problem 4 _____/15

Total _____/100

Problem 1 *(15 points)* Consider the following C++ code:

```
1  struct Foo {
2      int a;
3      shared_ptr<Bar> b;
4  };
5  struct Bar {
6      int c;
7      int d;
8  }
9  int main() {
10     shared_ptr<Bar> bar1 = make_shared<Bar>();
11     bar1->c = 100;
12     bar1->d = 200;
13     shared_ptr<Bar> bar2 = make_shared<Bar>();
14     bar2->c = 300;
15     bar2->d = 400;
16     Foo f;
17     f.a = 500;
18     f.b = bar1;
19
20 }
```

Draw a picture of the call stack and heap as it exists when execution reaches line 19. That is, draw the memory used by the frame on the call stack, plus all memory used by structures on the heap. For each memory location drawn, fill it in with either an integer value or use an arrow to show the pointer.



Problem 2 The list intersperse operation takes a list plus a new element, and inserts the new element between all the existing elements in the list. For example, calling intersperse with the element `"::"` on the list `["Think", "twice", "code", "once"]` would result in the list `["Think", "::", "twice", "::", "code", "::", "once"]`. Here is pseudocode for an intersperse which works with the List ADT.

```
intersperse(x, lst):
  for i from lst.size() - 1 downto 1:
      lst.insert(i, x)
```

(Part a, 15 points) Briefly (2-3 sentences) describe why the above code works. In particular, why does the loop end at `1` and why is the loop starting at the end and moving down?

Answer. (Note: insert and add are the same thing. Also, this answer here is a little longer than I expected, but I wanted to explain it in detail.) First, consider the example above. The size of `["Think", "twice", "code", "once"]` is 4 so the first call to insert is `lst.insert(3, "::")`. This inserts the colons into position three, pushing once to index four. That is, the list is now `["Think", "twice", "code", "::", "once"]`. Next, `lst.insert(2, "::")` is called which puts colons in position 2 and pushes all the later elements to higher indices.
In general, we can think of a call to insert i as adding the new element right before the element indexed by i. We therefore want to call insert on the index of every single element in the list except the zeroth element, since we don't want x at the beginning of the list. Also, calling insert only disturbs the indices of elements later in the list. For that reason, we start at the end at index `lst.size() - 1` and work our way down the list, inserting x before every element. We stop at index 1 because x should not go at the very beginning of the list.

(Part b, 10 point) If `lst` is an ArrayStack, what is the amortized cost running time of the intersperse pseudocode above? Briefly justify your answer, don't just write down a big-O time.

Answer. Each call to insert takes time $O(n-i)$, so the total time is

$$O\left(\sum_{i=1}^{n-1}(n-i)\right) = O\left(\sum_{j=1}^{n-1} j\right)$$

. Since the sum of the first $n-1$ numbers is $\frac{n(n-1)}{2}$, the total time is $O(n^2)$.

(Part c, continuing from previous page, 25 points) Is it possible to produce a faster (in the big-O sense) implementation of intersperse on ArrayStacks? If so, write pseudocode for an intersperse method for an ArrayStack struct that has properties `n`, `array_size`, and `arr`. You can assume that a method `resize` already exists that does not change the elements but doubles the size of `arr`. If you do not think a faster implementation is possible, explain why you think that is the case.

Answer. Yes, it is possible to improve the running time. The reason is identical to the `add_all` method you had to write for an exercise. When adding elements one by one, there is a lot of repeated work since elements must be copied out of the way for each call to add. In both `add_all` and `intersperse`, the improved running time version moves elements to their final position once before any elements are added. Specifically, for intersperse notice that after a call to intersperse, all odd indices contain the new element and all even indices are the original elements from the list. The total length of the new list is $2n-1$, so the first step is to check if the list needs to be resized. Next, we loop through all new indices. If the index is even the element from the original list is copied, and if the index is odd the new element x is set.

```
intersperse(x, lst):
    n = 2*n-1
    if n > array_size then resize()
    for i from n-1 down to 0:
        if i is even:
            arr[i] = arr[i/2]
        else:
            arr[i] = x
```

The amortized running time is $O(n)$, much better than the running time of the first intersperse on the previous page.

Problem 3 *(20 points)* This problem asks you to compare an ArrayDeque to a RootishArrayStack. First, use two diagrams to show how data is stored in an ArrayDeque and a RootishArrayStack. For example, draw how the list `[1,2,3,4,5,6,7,8,9]` could be stored in each implementation. Next write 3-4 sentences explaining what are the drawbacks of the ArrayDeque compared to a RootishArrayStack, and what are the drawbacks of a RootishArrayStack compared to an ArrayDeque. Your answer must discuss the space usage and amortized big-O execution time of the `get` and `insert` operations (you do not need to discuss the cost of `set` or `remove`). I do not want an essay here (you have limited time), so I will just be looking for highlights which you can cover with 3-4 sentences.

Answer. Look in the book for example drawings of the two structures. ArrayDeque can be drawn as a circular array or like the book does with a linear array. A RootishArrayStack has an array of block pointers and then each block is an array of increasing size.
First, get and set run in time $O(1)$ in both an ArrayDeque and a RootishArrayStack, so there is no difference for these methods. An ArrayDeque uses $O(n)$ wasted space, but has the advantage that add and remove run in amortized time $O(\min(i, n-i))$ so are efficient in add and remove on both ends of the list. To contrast, a RootishArrayStack has only $O(\sqrt{n})$ wasted space, much better than the ArrayDeque, but add and remove run in amotized time $O(n-i)$ so only operations on the end of the list (when $i = n-1$) are efficient.

Problem 4 *(15 points)* Here is a C++ structure for a simple singly-linked list of integers:

```
struct IntLinkedList {
    struct ListEntry {
        int val;
        shared_ptr<ListEntry> next;
    };
    int n;
    shared_ptr<ListEntry> head; //pointer to the first entry in the list
    shared_ptr<ListEntry> tail; //pointer to the last entry in the list
};
```

Write either psuedocode or C++ (whichever you are more comfortable with) to implement a method `count_positive` that has no parameters and returns the number of positive values in the list. The list itself is unchanged. What is the big-O running time of your method?

Answer.

```
int count_positive() {
    shared_ptr<ListEntry> u = head;
    int cnt = 0;
    //this checks if u is nil, but you could also compare u to tail
    while (u) {
        if (u->val > 0) {
            cnt += 1;
        }
        u = u->next;
    }
    return cnt;
}
```

The running time is $O(n)$ since the loop visits each node exactly once.