MCS 360 EXAM 2 11 NOVEMBER 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.a	__________/15
Problem 1.b	__________/15
Problem 1.c	__________/20
Problem 2.a	__________/20
Problem 2.b	__________/15
Problem 2.c	__________/15

Total __________/100

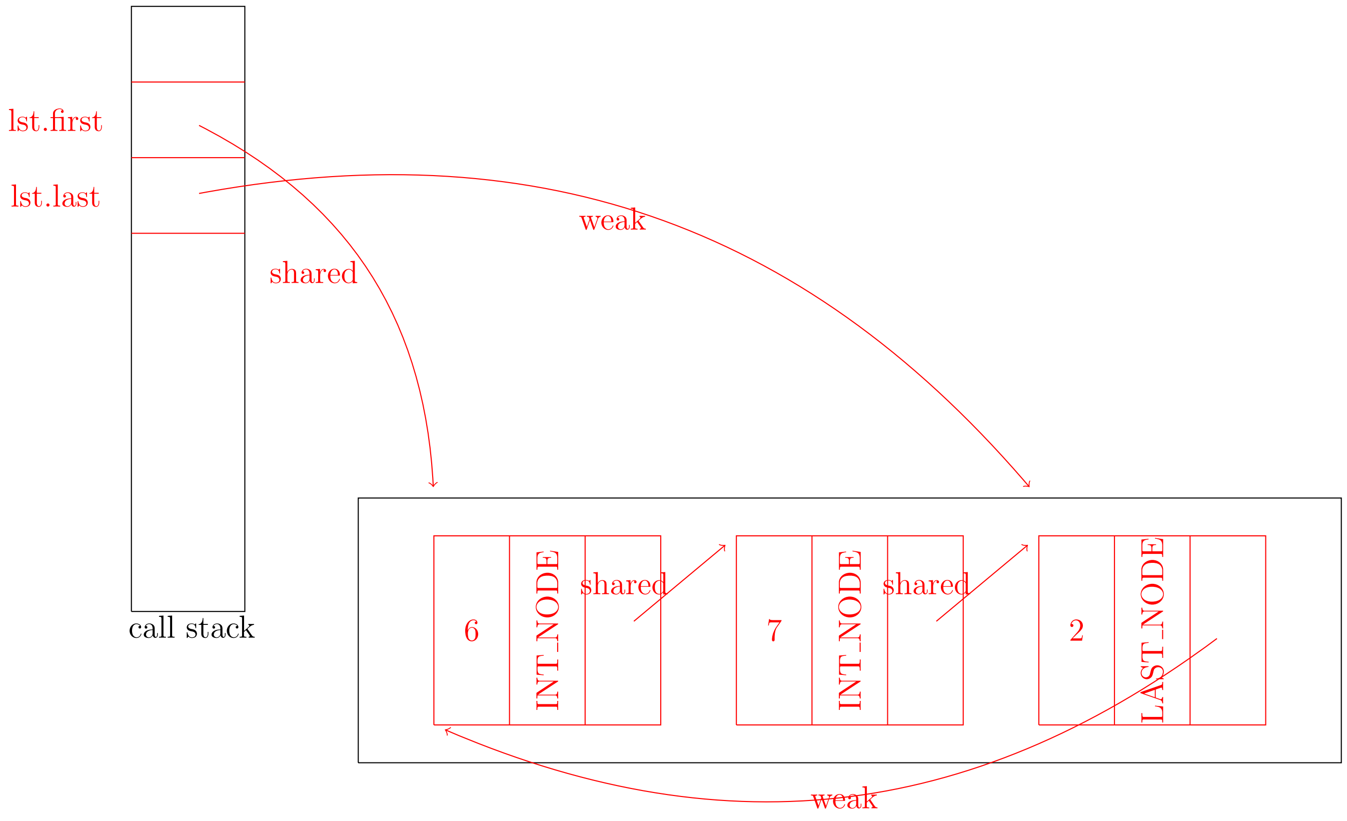
Problem 1 *(15 points)* Here is an attempt to construct a circular singly linked list of integers without a dummy node that is also properly deallocated.

```
struct IntLinkedList {

  struct Node {
    int val;
    enum {INTERNAL_NODE, LAST_NODE} tag;
    union {
        shared_ptr<Node> internalNext;
        weak_ptr<Node> fromLastToFirst;
    }
    Node(shared_ptr<Node> next) : tag(INTERNAL_NODE), internalNext(next) {};
    Node() : tag(LAST_NODE), fromLastToFirst(NULL) {};
    ~Node() {
        if (tag == INTERNAL_NODE)
            internalNext.~shared_ptr();
        else
            fromLastToFirst.~weak_ptr();
    }
  };

  shared_ptr<Node> first;
  weak_ptr<Node> last;

  void add_to_front(int v) {
      if (first) {
        first = make_shared<Node>(first);
        first->val = v;
        last->fromLastToFirst = first;
      } else {
        first = make_shared<Node>();
        last = first;
        first->val = v;
        last->fromLastToFirst = first;
      }
  }
};
```

(1.a) *(15 points)* Consider the following code:

```
lst = IntLinkedList();
lst.add_to_front(2);
lst.add_to_front(7);
lst.add_to_front(6);
```

Draw a picture of the call-stack and heap after executing this code. The call stack should have a single entry for the variable `lst` and the heap should show the memory for the three nodes. Draw the contents of memory and use an arrow for pointers. Label each pointer to show if it is shared or weak.



(1.b) *(15 points)* Will `IntLinkedList` have all the memory used by Nodes correctly deallocated? If yes, describe why memory is correctly deallocated. If no, describe the problem and how you could fix it. Be sure to mention the use of shared versus weak pointers. Your answer should be 4-5 sentences.

Answer. Yes, memory is properly deallocated. Shared pointers work by maintaining a reference count and will deallocate memory when the count reaches zero. Weak pointers do not contribute to the reference count so do not prevent a count from reaching zero. The reason that shared pointers would not properly deallocate memory is if there existed a pointer cycle. Examining the `IntLinkedList` definition, shared pointers do not create a cycle since the last node in the list does not use a shared pointer but a weak pointer.

(1.c) *(20 points)* Write a method `count_positive` for the `IntLinkedList` struct that takes no inputs and returns the number of positive entries in the list. Be sure to use a switch on the tag in each node.

Answer. There are several possible solutions, but here is mine:

```
int count_positive() {
    shared_ptr<Node> n = first;
    int cnt = 0;
    while (true) {
        if (n->val > 0)
            cnt += 1
        switch (n->tag) {
          case Node::INTERNAL_NODE:
            n = n->internalNext;
            break;

          case Node::LAST_NODE:
            return cnt;
    }
    return cnt;
}
```

Problem 2 Consider the following Treap definition.

```
template <typename Tkey, typename Tval>
struct Treap {
    struct Node {
        Tkey key;
        Tval val;
        int p;
        shared_ptr<Node> left;
        shared_ptr<Node> right;
    };

    shared_ptr<Node> root;
};
```

(2.a) *(20 points)* Write a method `are_priorities_valid` for the Treap struct which takes no parameters and returns true if the `p` properties satisfy the heap property of the treap and false otherwise. Your method should be recursive.

Answer. The heap property is that $v.p$ is larger than both $v.left.p$ and $v.right.p$. So we will visit each node checking this property, recursing to both the right and left. The return values from the right and left then are combined to produce my return value. We also need a helper function.

```
bool are_priorities_valid() {
    return valid_helper(root);
}
bool valid_helper(shared_ptr<Node> v) {
    if (!v) {
        return true;
    }
    bool leftOk = true;
    bool rightOk = true;
    if (v->left) {
        if (v->p > v->left->p)
            return false;
        leftOk = valid_helper(v->left);
    }
    if (v->right) {
        if (v->p > v->right->p)
            return false;
        rightOk = valid_helper(v->right);
    }
    return leftOk && rightOk;
}
```

(2.b) *(15 points)* Give the running time of your `are_priorities_valid` method in big-O notation. Is it possible to improve the running time?

Answer. The running time is $O(n)$, since each recursive instance is time $O(1)$ and we visit each node exactly once. It is impossible to improve the running time since we need to visit each node to check the heap property, and there are n nodes.

(2.c) *(15 points)* In a treap, if we call `add(x)` and then immediately call `remove(x)` with the same value x, do we always end up with the original treap? Argue why or why not.

Answer. Yes, the same tree will be produced. A structure of a treap is always as if we had formed a binary search tree by adding the elements in order of their priorities. The rotations we do during insert and remove are just fixuup we do in order to make the treap look as if it was formed by adding elements in order of priority. Thus if we add and then remove an element x, none of the other elements changed priority so we will get back the same treap. Thus any rotations that happen during the insert of x are exactly undone by the remove.