MIDTERM 2
CSCI 2270 Fall 2014

Name: BATMAN

Honor code: I promise, once again, not to cheat.

Sign: Batman never lies!

1. When we wrote the linked-list version of a queue, we chose to add items at one end of the list and remove them from the other (something every queue must do).

1a. [2 pts] Which end of the list do we add to?

TAIL

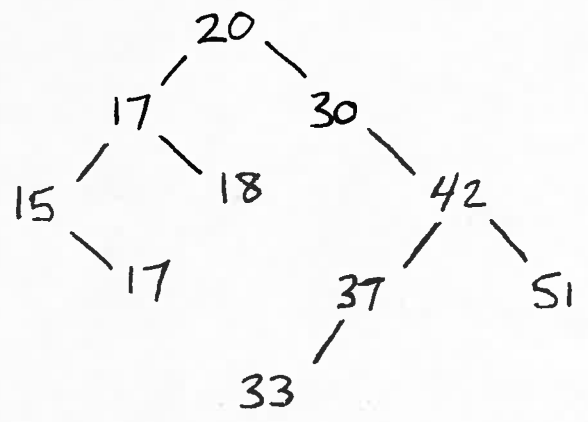
1b. [2 pts] Which end of the list do we remove from?

HEAD

1c. [6 pts] Why do we do it this way and not at the opposite ends?

IF WE REMOVE FROM THE TAIL, WE'LL NEED TO REDEFINE THE TAIL AS THE PREVIOUS NODE. SINCE THE LIST IS SINGLY LINKED, THIS IS O(N), NOT O(1). VERY SLOW.

2. Here is a binary search tree:



Print it out in

2a. [4 pts] Preorder

20, 17, 15, 17, 18, 30, 42, 37, 33, 51

2b. [4 pts] Postorder

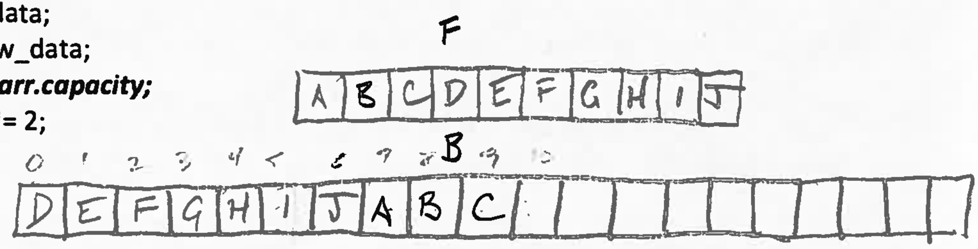
17, 15, 18, 17, 33, 37, 51, 42, 30, 20

2b. [2 pts] In-order

15, 17, 17, 18, 20, 30, 33, 37, 42, 51

3. [10 pts] Here is the code for the array-based queue's add method. It resizes when the back and the front of the queue are the same.

```
void add(array_queue& que, const int& payload)
{
        que.arr.data[que.back] = payload;
        que.back = (que.back + 1) % que.arr.capacity;
        if (que.back == que.front)
        {
                int* new_data = new int[que.arr.capacity * 2];
  ①             if (que.front > 0)
                        copy(que.arr.data + 0, que.arr.data + que.front, new_data + 0);
  ②             copy(que.arr.data + que.front, que.arr.data + que.arr.capacity,
                        new_data + que.front + que.arr.capacity);
                delete [] que.arr.data;
                que.arr.data = new_data;
  ③             que.front += que.arr.capacity;
                que.arr.capacity *= 2;
        }
}
```



Rewrite the italicized code here so the resized queue has a front = 0, there's no wraparound in the array, and the back value is set correctly. Use the copy command, and make sure you land these elements at the correct locations in the larger array.

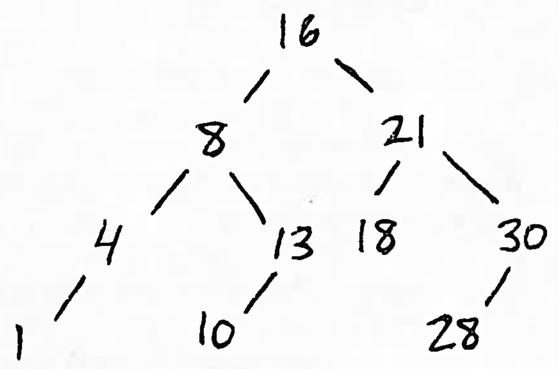
```
① if (que.front > 0)
      copy (que.arr.data + 0, que.arr.data
          + que.front, new_data +
          que.arr.capacity - que.front);

② copy (que.arr.data + que.front,
        que.arr.data + que.arr.capacity,
        new_data + 0);

③ que.front = 0;
   que.back = que.arr.capacity;
```

4. [10 pts] Recall the tree_clear function:

```
void tree_clear(binary_tree_node*& root_ptr)
 // Library facilities used: cstdlib
 {
                binary_tree_node* child;
                if (root_ptr != nullptr)
                {
                        child = root_ptr->left;
                        tree_clear( child );
                        child = root_ptr->right;
                        tree_clear( child );
                        delete root_ptr;
                        root_ptr = nullptr;
                }
 }
```



Trace out the tree_clear function for the above binary tree, and write down the order in which each node is deleted.

1, 4, 10, 13, 8, 18, 28, 30, 21, 16

5. Binary search. Here's the code from the binary search lecture.

```
bool binary_contains1(const int* arr, unsigned int first, unsigned int last, int target)
{
        cout << "Searching from " << first << " to " << last << endl;
        if (first > last)
        {
                cout << target << " is not here" << endl;
                return false; // target not in original array
        }
        if (first == last)
        {
                if (arr[first] == target)
                        cout << target << " Found " << target << " at " << first << endl;
                else
                        cout << target << " is not here" << endl;
                return arr[first] == target;
        }
        else
        {
                int mid = first + (last - first) / 2;
                cout << "Checking item " << arr[mid] << endl;
                if (target == arr[mid])
                {
                        cout << "Found " << target << " at " << mid << endl;
                        return true;
                }
                else if (target < arr[mid])
                        return binary_contains1(arr, first, mid - 1, target);
                else
                        return binary_contains1(arr, mid + 1, last, target);
        }
        return false;
}
```

Given this array: 2 19 30 76 77 91 100 105

5a. [2 pts] Tell me what middle element binary search for a 51 will check first.

76

5b. [4 pts] What middle elements after that will the search examine until it stops?

19 and 30

5c. [2 pts] When will it know to give up looking?

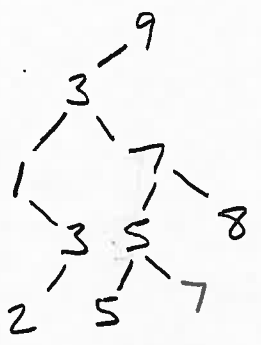
first >= last

5d. [2 pts] Could we do binary search on a doubly linked list? Justify your answer.

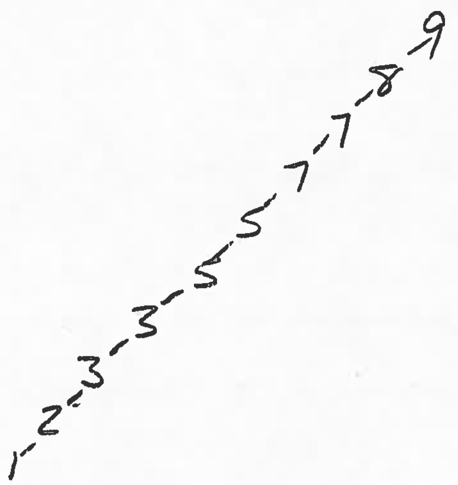
No: Can't find the middle element in reasonable time.

6. Binary search trees

6a. [4 pts] Insert the numbers 9, 3, 7, 1, 3, 5, 2, 8, 7, 5, in that order, into a binary search tree.



6b. [4 pts] Insert the numbers 9, 8, 7, 7, 5, 5, 3, 3, 2, 1, in that order, into a new binary search tree.



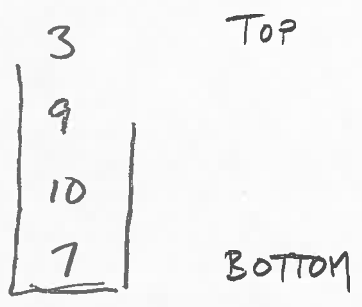
6c. [2 pts] Which tree is preferable? Justify this answer.

6a - more complete

7. Stacks and queues

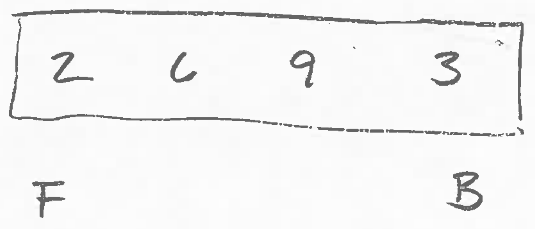
7a. [5 pts] Assume you have an empty stack called s. Draw s after these operations, indicating the top and bottom of the stack.

```
push(s, 7);
push(s, 8);
pop(s);
push(s, 10);
push(s, 2);
pop(s);
push(s, 6);
pop(s);
push(s, 9);
push(s, 3);
```



7b. [5 pts] Assume you have an empty queue called q. Draw q after these operations, indicating the front and back of the queue.

```
add(q, 7);
add(q, 8);
remove(q);
add(q, 10);
add(q, 2);
remove(q);
add(q, 6);
remove(q);
add(q, 9);
add(q, 3);
```



8. [10 pts] Tell me about one really good book I should read.

Batman books!