2015 Exam

1)

```
// Returns whether the given string is a palindrome.
int checkpalindrome(char *buffer){
    int i,j;

    // We will use a kind of inverted quicksort partition idea to find values
    // which should be on both sides.
    i = -1;
    j = strlen(buffer);

    while(i < j){
        i++;
        j--;
        // Move over irrelevant characters
        while (i < j && (! isalpha(buffer[i]))){
            i++;
        }
        // Move over irrelevant characters
        while (i < j && (! isalpha(buffer[j]))){
            j--;
        }

        // Cross over (we know the character at i == j is the same for buffer[i]
        // and buffer[j])
        if (i >= j){
            // Palindrome
            break;
        }

        // Only need to compare lowercase letters as case insensitive.
        if(tolower(buffer[i]) != tolower(buffer[j])){
            // Not Palindrome
            return 0;
        }
    }

    // Palindrome
    return 1;
}
```

2a)

leaves

leaves

leaves

2b) New memory is not allocated for each string added to the array, this causes the same address to be added to the array in multiple places, leading to the original strings being overwritten. This is not the intended effect.

2c)
Between line 17 and 18 in original code, add `char *tempString;`
line 36 in original code replaced by:

```
// Allocate space for string
```

```
tempString = (char *) malloc(sizeof(char)*(strlen(buffer) + 1));
// Check we got memory
if(!tempString) return 1;
// Copy buffer into temp string
strcpy(tempString, buffer);
```

3a) O(1)

3b) Θ(n log n)

3c) Θ(n log n)

3d) O(n) extra memory

3e) O(n) extra memory

3f) O(log n) [Assuming equal elements are not a problem for our balanced tree]

4ai)

```
              100
         12           79
    11     13     77    11
 87   25

              100
         12           79
    11     13     77    11  <- level OK
 87   25                    <- level OK

downheap 12 and 79 (as they're not smaller than both their children)
              100
         11           11     <- level OK
    12     13     77    79
 87   25
downheap 100 [Option 1]
|              11
|         11           100
|    12     13     77    79
| 87   25
|
|              11               <- level OK
|         11            77
|    12     13     100    79
| 87   25
Array: 11 11 77 12 13 100 79 87 25
downheap 100 [option 2]
|              11
|       100           11
|    12     13     77    79
| 87   25
|
|              11
|       12            11
|    100    13     77    79
| 87    25
|
|              11             <- level OK
|       12            11
|    25     13     77    79
| 87    100
Array: 11 12 11 25 13 77 79 87 100
```

4aii) Bottom-up heap construction can complete in Θ(n) time, w hile inserting one by one and upheaping may take O(nlogn) time.

4bi) O((V + E)logV) - deletemin O(log V), update O(log V), O(E) maximum edge updates, O(V) deletemins

4bii) O(EV) - deletemin O(1), update O(V), O(E) maximum edge updates, O(V) deletemins.

4c) 4n + 1, 4n + 2, 4n + 3, 4n + 4, assuming beginning from 0.

5ai) θ(V^3)

5aii) θ(E^(3/2))

5aiii) θ(E^3)

5b - Note: A* wasn't covered this year, so you won't be expected to be able to answer these questions.

5bi) false, in a full search, this will be true (assuming cycle checking), however if the order of expansion is different due to involved data structures this may cause A* to search in a more or less efficient way where it is able to terminate early when any solution is found. Likewise, for a graph with all 0 weights, A* may terminate earlier.

5bii) true, any edge from a non-solution state has a cost > 0, and the solution state has a cost of 0 when reached. Hence all states are at least 0 distance away from the solution.

5biii) true, the next closest node expanded is the one with the shortest cost to that node left.

5c) Construct an MST (Prim's or Kruskal's), run Dijkstra's or A* on this graph.

5d) Check which nodes are in the MST, for any nodes not in the MST, rerun Prim's for one of those nodes. Repeat this process until all nodes are in an MST.

6ai) `[ - - - - - 17 53 41 29 - - - ]`
Multiples of 12:

12

24

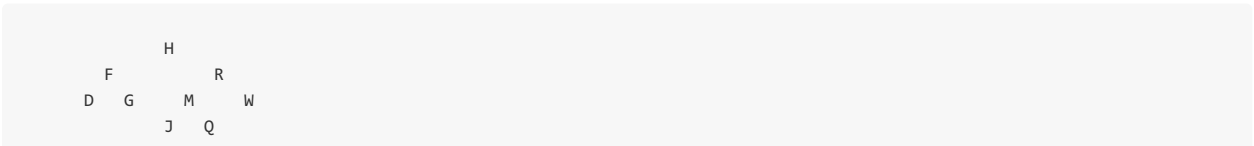
36

48

6aii) 125 % 12 = 5, 4 key comparisons (17 [5], 53 [6], 41 [7], 29 [8]), assuming free comparison to empty cells.

6aiii) Double Hashing, By using other properties of the keys being inserted, the values could be spread out more over the array.

6bi)



6bii)

```
// Swap "right" for "left" if the rotation occurs on the other side of the tree.
// Shift child up.
grandparent->right = child;
// Swap middle item.
parent->right = child->left;
// Shift old parent down.
child->left = parent;
```