Christian Ayala
Algorithm Engineering

Homework 2

Before beginning with the problem, I will define the structure of a binary search tree. A binary search tree is composed of nodes, with the following attributes (written in C++):

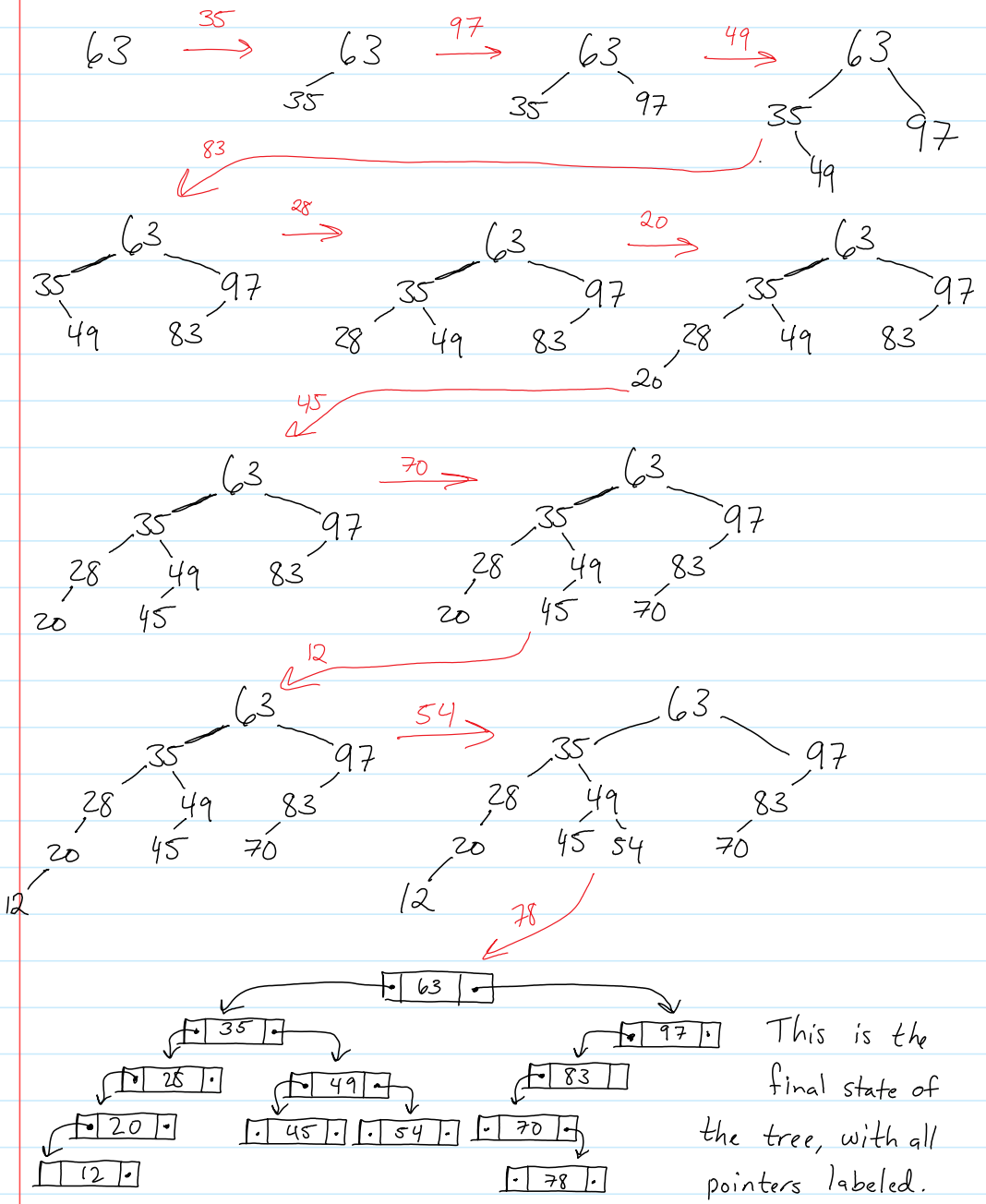
```
class Node {
public:
    Node *left;
    Node *right;
    int payload;
};
```

A binary search tree is such that, at a particular node n_i, all nodes in the left sub-tree of n_i contain payloads less than n_i. All nodes in the right sub-tree of n_i contain payloads greater than n_i. A typical recursive solution for inserting a new element is as follows:

```
void insert(Node*& root, int data) {
    if (root == nullptr) {
        root = new Node();
        root -> payload = data;
    } else if (data < root -> payload) {
        insert(root -> left, data);
    } else {
    } insert(root -> right, data);
}
```

Question 1 Page 2

Using the insert function, we can build a tree:
S = 63, 35, 97, 49, 83, 28, 20, 45, 70, 12, 54, 78



This is the final state of the tree, with all pointers labeled.

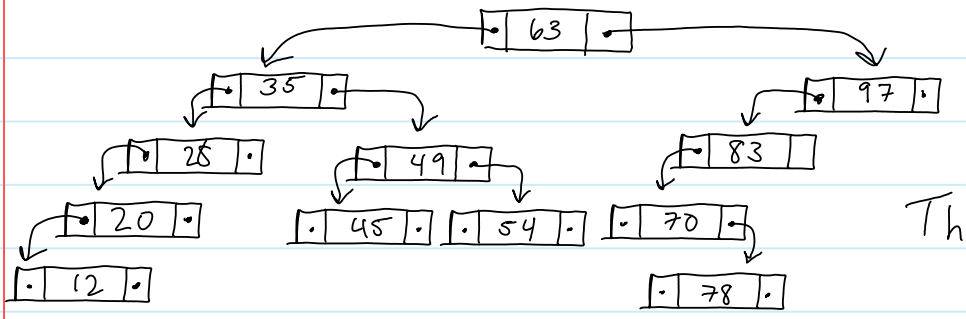
Question 1 Page 3

When converting from a binary search tree to a doubly linked list, the natural inclination is to use recursion. By modifying a typical in-order traversal, we can rearrange the nodes' left and right pointers to form a list. In order for this recursive strategy to work, each function call must maintain a pointer to the previously visited node, and the head of the currently built list. Code for the function is shown below:

```
Node* convertToList(){
        Node* head = nullptr;
        convertToList(root, head, nullptr)
        return head;
}

void convertToList(Node* root, Node*& head, Node*& previous){
1       if(root == nullptr) return;
2       convertToList(root->left);
3       root->left = previous;
4       if(previous != nullptr){
5           previous->right = root;
6       }else{
7           head = root;
8       }
9       previous = root;
10      convertToList(root->right, head, previous);
}
```

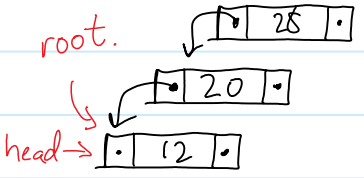
Question 1 Page 4



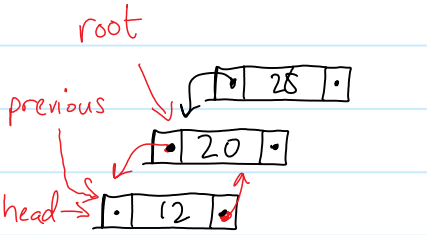
The tree is copied here.

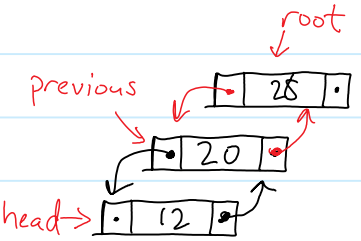
The function operates on the use of the previous pointer: once a node has been visited, it is marked as the "previous" node, and we then know that it will never be visited again (by the nature of an in-order traversal). Therefore, instead of modifying the current root node, we only ever modify the previous' next pointer to the current root noot. Stepping through the algorithm with the sample tree above will clarify.

We start by recurring until we reach the left-most node in the tree, 12. The recursion stops, and at line 3 we have root = 12 node. In this case, previous == nullptr, so we know the current root is our new head of the list. We then assign previous to 12, and recur right. Since there are no children to the right, we recur back to the 20.

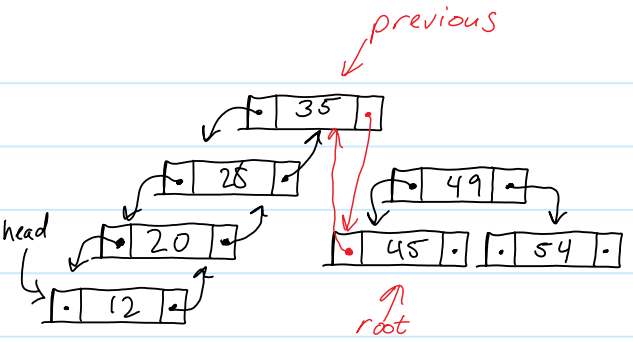


At 20, previous != nullptr, so we assign root->left = previous, and previous->right = root. This makes the 12 point to the 20, and vice-versa. By continuing the recursion once more, the 20 then points to the 28, and vice-versa

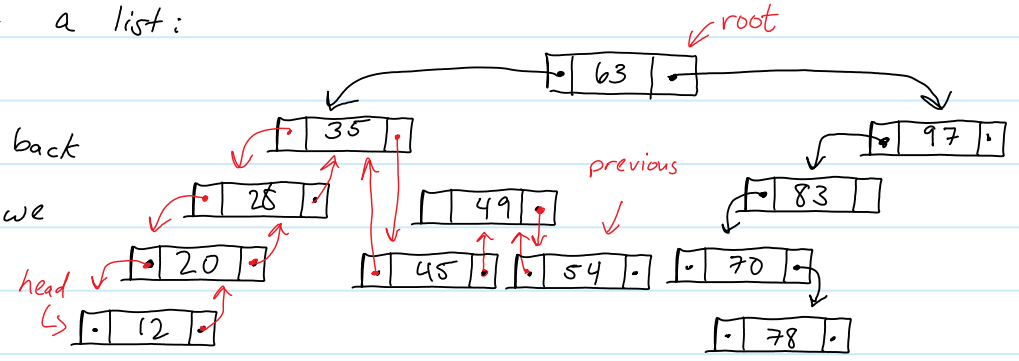




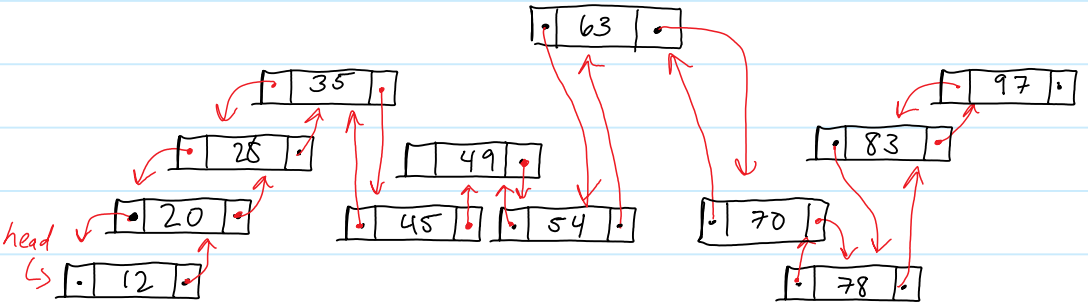
Question 1 Page 5



When we reach the 35 node, and line 9 has finished, we then recur to the right. Since 49 has a left child, we recur left, and root points to 45 (with previous still at 35). Once again, lines 3-5 execute to reassign pointers. 35 then points to 45, and vice-versa. We continue this until the entire left subtree is a list:



The root is now back to the 63, and we continue the recursion until the entire tree has turned into a list.



Using this algorithm, we have a constant number of additional pointers: head and previous. However, since this is a recursive solution, we have the overhead of the call stack. On average, this would require an additional $O(\lg_2 n)$ space, meaning that this algorithm is not truly in-place. The benefit to this particular function is that it is very simple to implement and understand.

Question 1 Page 6

In an attempt to make the algorithm truly in-place, I have removed the recursive element in the following function:

```
Node* convertToList(Node* root) {
1     Node *parent =nullptr, *head = nullptr, *tail = nullptr;
2     while (root !=nullptr) {
3         while (parent != root ) {
4             Node * smallestElement = getSmallestElement(root);
5             parent = smallestElement -> left;
6             if (head == nullptr)  head = tail = smallestElement;
7             else {
8                 smallestElement -> left = tail;
9                 tail -> right = smallestElement;
10                tail = smallestElement;
11            }
12            if (parent == nullptr) break;
13            else if (smallestElement -> right != nullptr)
14                parent -> left =smallestElement ->right;
15            else  parent -> left = nullptr;
16        }
17        if (parent ==nullptr) root = root -> right;
18        else if ( root -> left ==nullptr) {
19            root -> left = tail;
20            tail -> right = root;
21            tail = root;
22            root = root -> right;
23        }
24        parent = nullptr;
25    }
26    head -> left = tail -> right = nullptr;
27    return head;
}
```

Question 1 Page 7

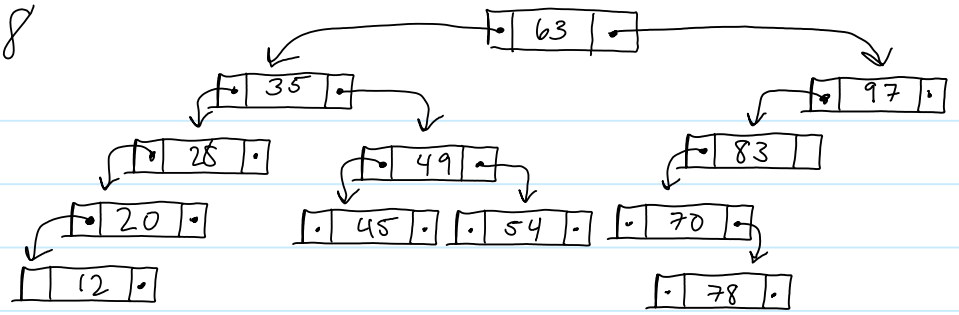
The getSmallestElement function in line 4 is shown below:

```
Node * getSmallestElement(Node * root) {
1       Node* leftChild = root -> left;
2       if (leftChild == nullptr) return root;
3       while (leftChild -> left != nullptr) {
4           root = leftChild;
5           leftChild = leftChild -> left;
6       }
7       leftChild -> left = root;
8       return leftChild;
}
```

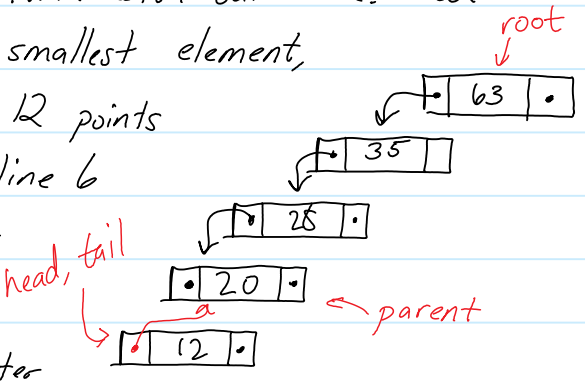
The function operates on the knowledge that, for any BST, the smallest element in the tree (a.k.a. the left-most node) has a left child pointing to null. Therefore, when building the list from smallest to largest, the smallest element can have its left pointer point back to its parent. This is what happens in line 7 of getSmallestElement. Once we have the smallest element in the tree (and it points to its parent), we can then extract it from the tree, and add it to the tail of our list. We continuously extract elements from the left subtree of root, until we have exhaustively removed all elements. We then add the root to the list, and then repeat the process with the right subtree of root.

Question 1 Page 8

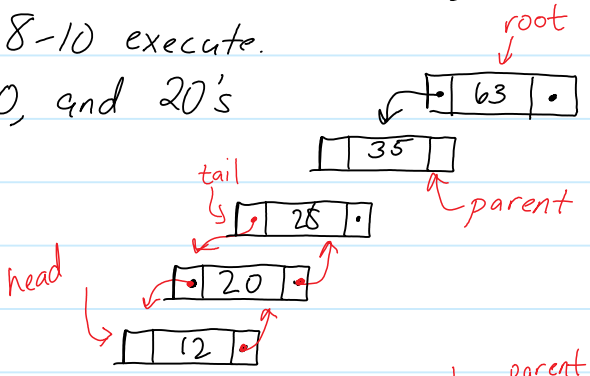
Once again, the tree is copied here



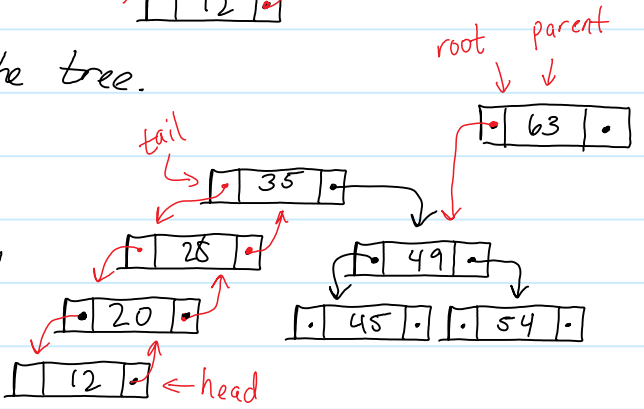
We will now walk through the algorithm with our tree. root begins at 63. We then pull the smallest element, which is 12. The left pointer of 12 points to 20. Head is currently null, so line 6 executes, and 12 is the head of our list. Since parent is not null, and 12 has no children, the left pointer of 20 is set to null, and 12 has effectively been removed from the tree.



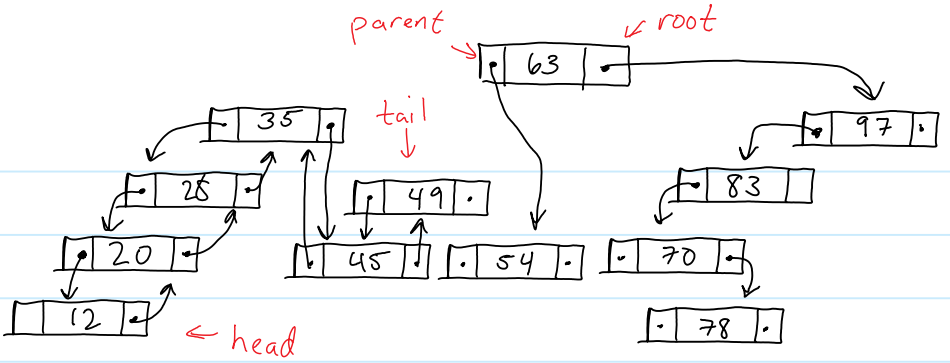
We then loop again, getting the smallest element (20). Since head is no longer null, lines 8-10 execute. This makes 12's right point to 20, and 20's left point to 12. Once again, 20 has no right tree, so 28's left gets set to null, and 20 is removed from the tree. Repeat again, and 28 is removed from the tree.



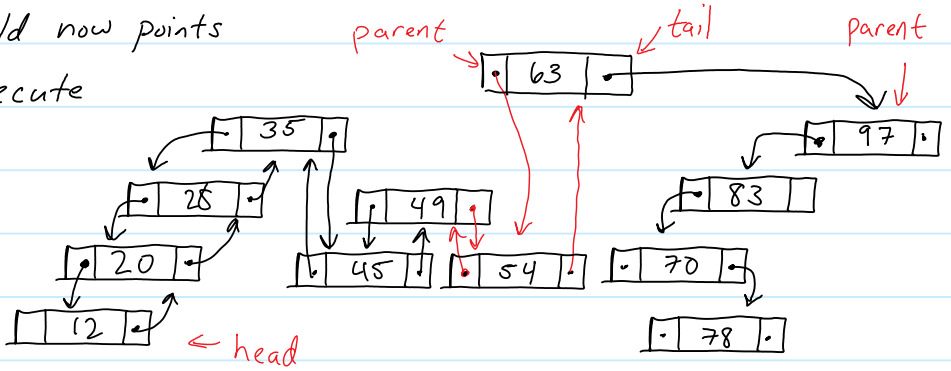
When we attempt to remove 35 from the tree, lines 8-10 happen normally, but since 35 has a right child, line 14 executes and reassigns 63's left pointer to the right subtree of 35. We have removed 35 from the tree, we reset parent to null on line 24, and continue adding elements.



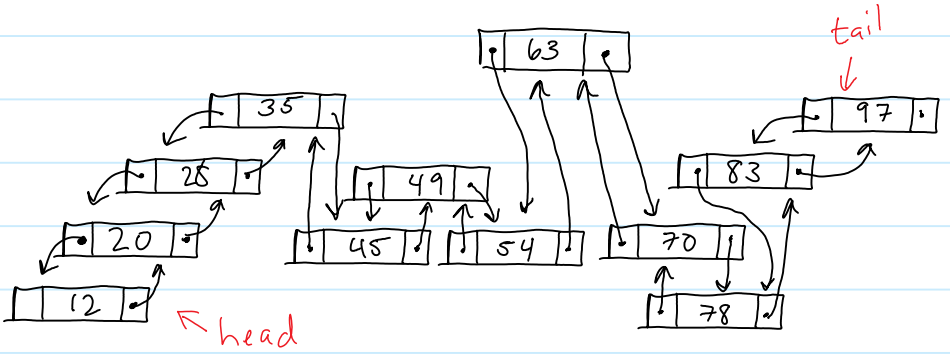
Question 1 Page 9



We now fast-forward to this state. The root node 63 has a single left child, 54. Lines 8-10 execute normally, and the root node's left child now points to null. We then execute lines 18-22, thereby adding the root to the list. Line 22 makes the new



root of the tree be node 97. The entire left subtree (and root) of the original subtree has been converted to a list, and the process repeats with the new subtree at 97. Eventually, 97 will be the last element to add. Line 22 will execute, setting root = null. The large loop ends, and the entire tree has transformed to the following structure:



The tree is now a doubly linked list with constant overhead: a parent pointer, head and tail pointers, and a temporary pointer in getSmallestElement. Therefore, the revised function is fully in-place. To make the list circular, change line 26 to:

```
head -> left = tail;
tail -> right = head;
```

Runtime complexity is $O(n)$, since each node is visited once.

Question 1 Page 10

We will now demonstrate correctness of the algorithm through the use of a loop invariant. I argue that at each iteration of the while loop on lines 3-16, all elements between the head and tail pointers are in sorted order, and contain elements that are less than the remaining elements in the tree. At each of the iterations of the larger loop on lines 2-24, the above statement holds true, and the left subtree has been completely processed.

Initialization: Before the loops start, head and tail are both null. Empty lists are inherently sorted, so our invariant holds.
Maintenance: At iteration K, we have removed the k-1 smallest elements from the tree. We then add the k^{th} smallest element to the end of our list, thereby maintaining sorted order. This element is removed from the tree, which leaves n-k elements in our tree, which are guaranteed to be larger than the just-removed element. Thus we preserve our loop invariant.
Termination: When the loop terminates, tail is pointing to the largest element in the original tree, and head is pointing to the smallest element. Our root is null, meaning our tree is empty. All elements are in sorted order (as in, element k < element k+1 in our list), and we conclude that our final list is sorted. Thus, our algorithm is correct and in-place.

Question 2 Page 1

Question 2a

The heapify algorithm is shown below:

```
void heapify (int* data, int size) {
    for (int i = size-1; i >= 0; ++i) {
        bubbleDown(data, size, i);
    }
}

void bubbleDown (int* data, int size, int current) {
1       int childIndex = (current * 2) + 1;
2       if (childIndex >= size) return
3       if (childIndex != size-1)
4           childIndex += (data[childIndex] > data[childIndex+1]) ? 0 : 1;
5       if (data[childIndex] > data[current]) {
6           swap (data[childIndex], data[current]);
7           bubbleDown (data, size, childIndex);
8       }
}
```

The heapify function is straightforward: starting at the right-most element, we call bubbleDown on that elements' index, to see if it needs to travel down the heap. The bubbleDown element checks to see if a given element is less than a child element. In particular, it finds the largest child element of that node. If it is, then they are swapped, and we recur with the child element to see if it needs to travel farther down the tree.

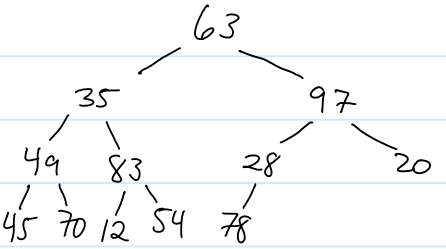
Question 2 Page 2

Input Sequence ⇒ S:

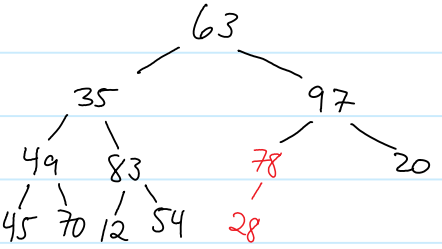
0	1	2	3	4	5	6	7	8	9	10	11
63	35	97	49	83	28	20	45	70	12	54	~~78~~

The children of a node at index i are at positions $2i+1$ $2i+2$ (since indices in C++ are 0 based).

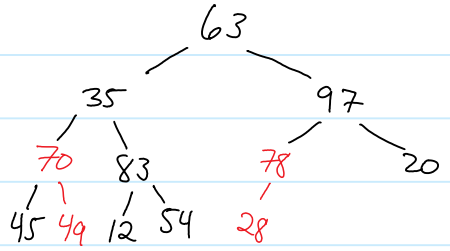
Therefore, as a tree, it looks like:



Starting the heapify function, we attempt to bubbleDown the 78. However, since it is a "leaf" node, bubbleDown immediately returns. This happens for 54, 12, 70, 45, and 20. Then we get to 28. In this case, there is a child node (78). Since line 5 of bubbleDown is true, 28 is swapped with 78. We then try to bubbleDown 28 again, and stop. Thus, our tree looks like:



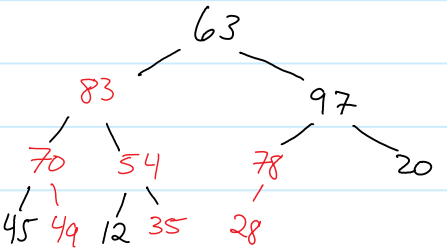
Continuing with the 83, it does not move since it is larger than both of its children. At 49, we see that 70 is the largest child, and therefore needs to be swapped. The tree is now:



We are now at 97, which also does not move. At 35, we see that it is smaller than the right child, so it swaps. We then bubble down again, and we see that $35 < 54$, so we swap those two. 35 is now in the right place, as are the 54 and 83.

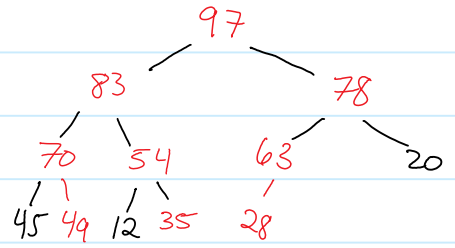
Question 2 Page 3

At this point, the tree is:



Finally, we examine the 63. It is swapped with the 97, and again with the 78, where it stays. The sequence has now been completely heapified. The final sequence and tree are:

S: 97, 83, 78, 70, 54, 63, 20, 45, 49, 12, 35, 28



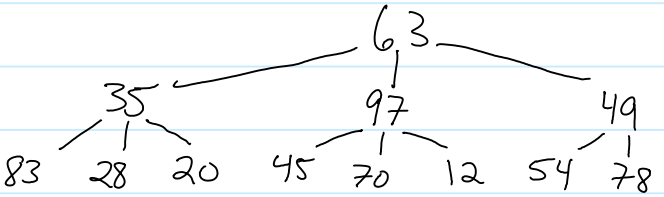
We did 6 swaps and 15 comparisons.

Question 2b

The algorithm for Ternary Heapify is identical to Binary Heapify, except for the fact that we will be finding the largest element of 3 items, instead of 2. This requires 2 comparisons.

The original sequence as an array and a tree is shown here:

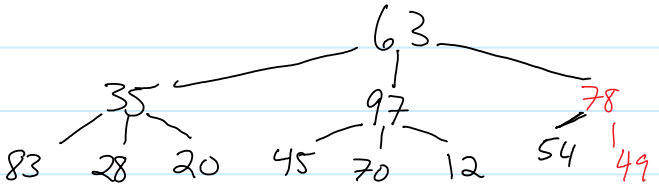
0	1	2	3	4	5	6	7	8	9	10	11
63	35	97	49	83	28	20	45	70	12	54	78



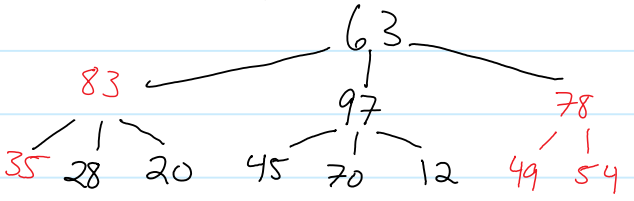
Question 2 Page 4

We will now step through the algorithm. Starting with the right-most element (78), we realize that it does not have any children. Therefore, it does not move. This happens, in sequence, for 54, 12, 70, 45, 20, 28, and 83.

Finally, we reach element 49. Its largest child is 78 and therefore it is swapped. 49 is now a leaf node, and does not move. The 78 is also in a proper place. Now both elements are in proper locations, and our heap is now:



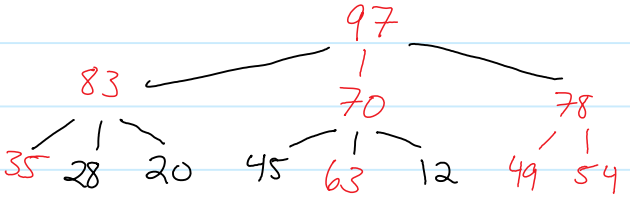
Continuing with 97, we compare it with the largest of its children, but do not swap. Next, we examine 35, and swap it with 83. No additional swaps are needed, and our heap is now:



The last element we examine is 63. Since 97 > 63, they are swapped. By recurring, we see that 63 < 70, and must therefore be swapped. The 63, 70, and 97 elements, are in proper locations, and the sequence has been heapified. The final sequence and trees are shown below

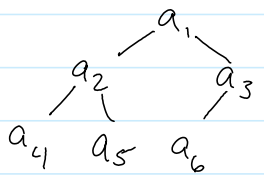
S: 97, 83, 70, 78, 35, 28, 20, 45, 63, 12, 49, 54

We did 4 swaps and 11 comparisons

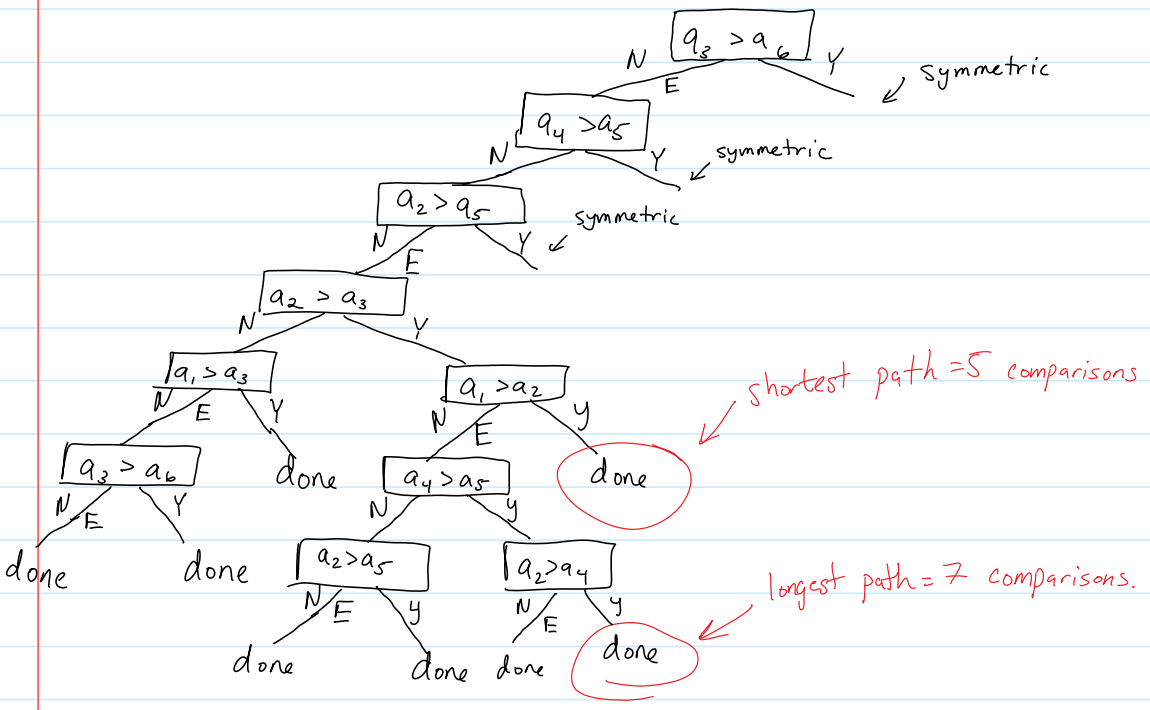


Question 2c

Input sequence: $a_1\ a_2\ a_3\ a_4\ a_5\ a_6$



Since we examine starting at the leaf nodes, we don't have any comparisons for the first three iterations. Then, we have the following decision structure (Y is yes, N is no, E is exchange):



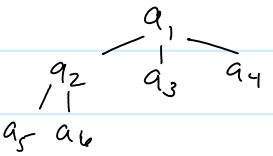
As can be seen above, the worst case to heapify 6 elements is 7 comparisons. This happens in half of all cases (since the above tree is entirely symmetric where labeled). 6 comparisons occurs in 1/4 of all cases, and 5 comparisons occurs in 1/4 of all cases as well. As such, on average it takes

$$\left(\frac{1}{2}\cdot 7\right) + \left(\frac{1}{4}\cdot 6\right) + \left(\frac{1}{4}\cdot 5\right) = 6.25 \text{ comparisons.}$$

This is optimal.

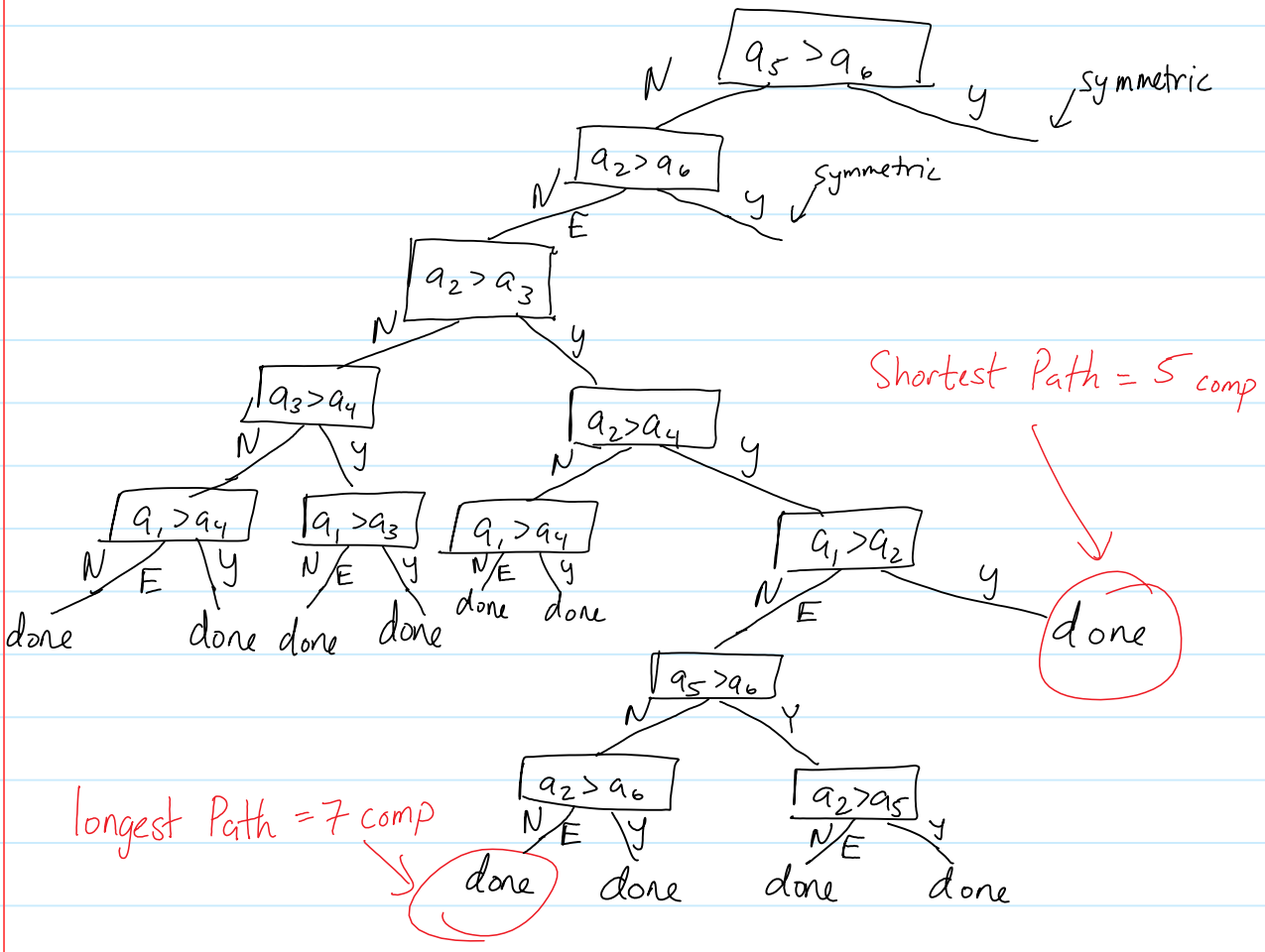
Question 2 Page 6

Question 2d



Input sequence: a_1 a_2 a_3 a_4 a_5 a_6

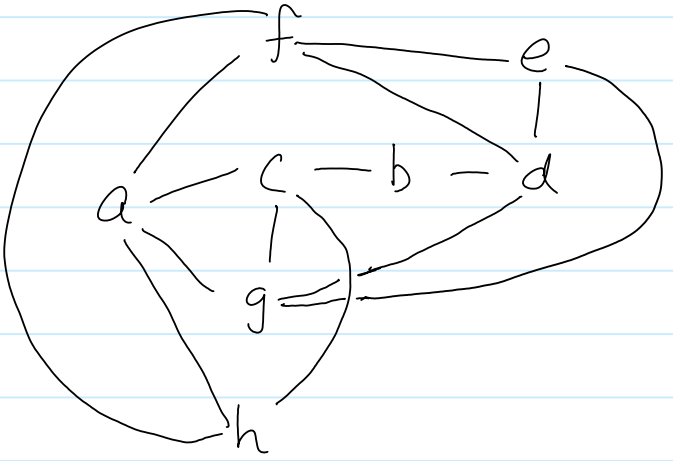
Similar to the previous question, a_6 and a_5 initially do not require comparisons. We then build the following structure:



Once again, the worst case to heapify this sequence is 7 comparisons. This happens in 4/11 of all cases. In all other cases it only takes 5 comparisons. On average it takes $\left(\frac{7}{11} \cdot 5\right) + \left(\frac{4}{11} \cdot 7\right) = 5.72$ comparisons. This is optimal as well

Question 3 Page 1

a: c, f, g, h
b: c, d
c: a, b, g, h
d: b, e, f, g
e: d, f, g
f: a, d, e, h
g: a, c, d, e
h: a, c, f



The algorithm for determining the smallest last search order is as follows:

Given a graph G on n vertices, the following determines an ordering $v_1, v_2, \ldots, v_n$ of the vertices of G, where $\deg(v_i | G_i) = \delta(G_i)$ for $1 \leq i \leq n$

SL1: [Initialize]: $j \leftarrow n$, $H \leftarrow G$

SL2: [Find min degree vertex]: Let v_j be a vertex of minimum degree in H

SL3: [Delete min degree vertex]: $H \leftarrow H - v_j$, $j \leftarrow j - 1$

SL4: [Finished?]: If $j \geq 1$, return to step SL2, otherwise terminate with sequence $v_1, v_2, \ldots, v_n$

The ordering $v_1, v_2, \ldots, v_n$ of the vertices of G are the smallest-last ordering for G, since $\deg(v_i | G_i) = \delta(G_i)$ for $1 \leq i \leq n$, where $\delta(G_i) = \min_{v \in V(H)} \{\deg(v | H)\}$

(This algorithm was retrieved from an article published by David Matula and Leland Beck, published in Vol. 30 #3 Of the ACM Journal, July 1983).

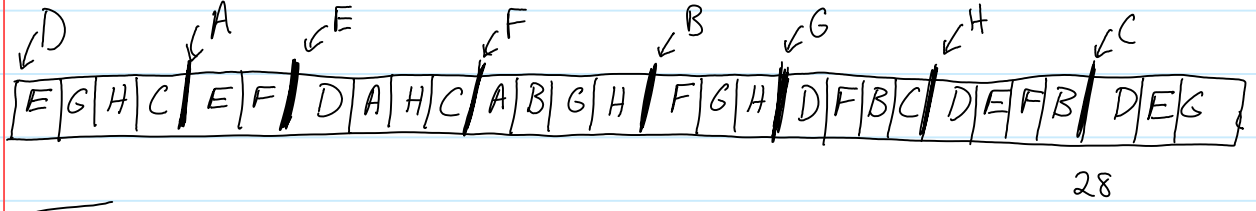
Question 3 Page 2

With the algorithm mentioned previously, we can determine the smallest last search order as:

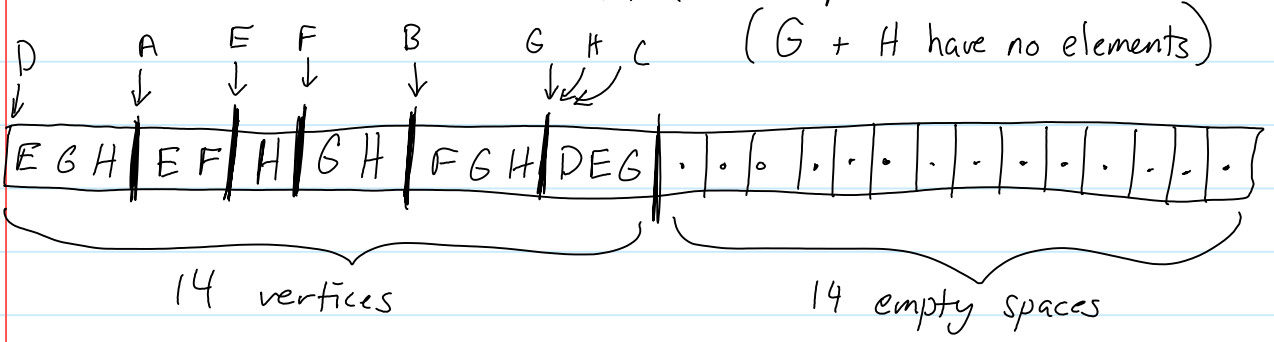
Degree:	b	e	h	a	c	d	f	g
Vertex:	2	3	3	4	4	4	4	4
Renamed:	A	B	C	D	E	F	G	H

We have renamed the vertices based on this new ordering, and we are done with the first step

Next, we compress the list. If we examine the edge list, we see that a given edge is represented twice in the list, which is unnecessary. Therefore, we can remove one of the two representations, without losing any information. Below is our renamed edge list:



To compress this list, we remove a pair (v_1, v_2) if v_i is larger lexicographically. With this, we have only pairs of (v_1, v_2), where $v_1 < v_2$, and we lose no information. Compressed: (G + H have no elements)

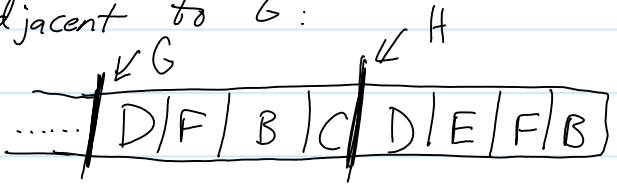


Question 3 Page 3

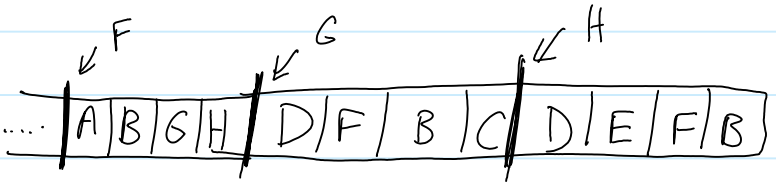
Next, we expand the compressed list into a full list. We process vertices in reverse alphabetical order, filling the list from right to left (transferring vertices if necessary). In this way, we can do it all in place, without deleting any of the compressed information.

First look at the last vertex H. In our compressed list it does not have any other vertices it connects to (which makes sense, since H is the last letter in our set). Therefore, no elements are decompressed yet. Then, starting from the right of the compressed list, we move left and record all vertices that contain H as a paired vertex. So, the right of our list looks like:

Next, we look at G. Again, it has no adjacent vertices in our compressed list. We then record all vertices that are adjacent to G:

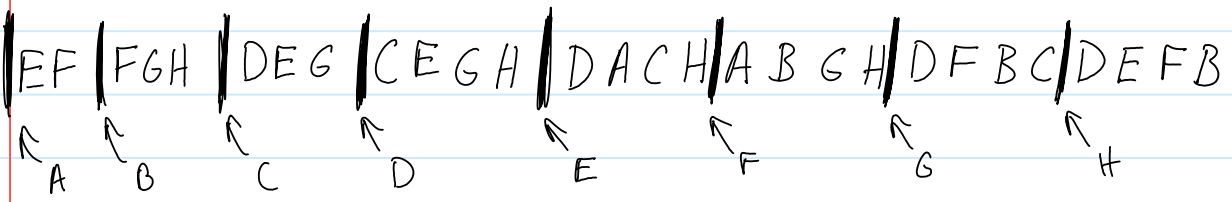


Next is F. Since F has adjacent vertices, we move the list into our uncompressed version. Once again, we then sweep left on the remaining vertices to record all who are connected to F. Our list looks like:

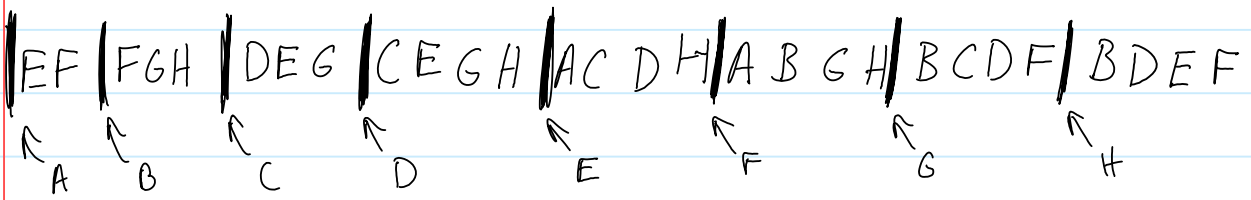


Question 3 Page 4

Vertices E, D, C, B, and A are all functionally the same as what we did for F. Processing these remaining vertices leads to this final adjacency list:



Finally, we sort each of the vertices' adjacency lists, which leads to the final ordering:



Our algorithm is now complete. We did not use any additional space, so space is constant. Renaming requires $O(|E|+|V|)$ time, as defined by the smallest last search order algorithm. Compression requires a full sweep of all edges, which is $O(|E|)$. Decompression also requires $O(|V|+|E|)$ operations. Sorting is negligible, because we assume that the size of a single vertex's adjacency list is insignificant when compared to the size of the entire list. Therefore, the overall complexity is $O(|V|+|E|)$.

Question 4 Page 1

Given 2 input matrices A and B, both of which are $n \times n$ matrices, computing a single entry in the result matrix C is fundamentally described as:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

This means that computing a single element C_{ij} takes n multiplications and additions, which is $O(n)$. Computing $n \times n$ cells in the matrix C has an overall complexity of $O(n^3)$ (best and worst-case).

However, with sparse matrices, most of the multiplications are $\emptyset$, and therefore do not contribute to the overall value of C_{ij}. By only storing the non-zero elements in an adjacency list, we do not need to do all $O(n^3)$ operations.

What we can see is that for a given row C_i, we need all of the elements in A_i. Similarly, for a given column C_j, we need all of the elements in B_j. Therefore, when computing a single row C_i, we can traverse the same row in A (A_i). For each element in the row A_i, we multiply by each B_{jk}, and the resulting value is stored in C_{ik}. So, a generalized formula is:

$$C_{ik} = A_{ij} B_{jk} + \underbrace{\text{Old value of } C_{ik}}_{\text{Initially } \emptyset}$$

Question 4 Page 2

A general algorithm is shown below:

```
void multiply(Matrix A, Matrix B, Matrix C, int n){
1     for (int i=1; i≤n; ++i) {
2         rowAi = A[i];   // Look at the i'th row of A
3         while (rowAi.hasNext()) { //Go through all elements
4             j = rowAi.currentColumn;
5             rowBj = B[j]; //Look @ j'th row of B
6             while (rowBj.hasNext()) {
7                 k = rowBj.currentColumn;
8                 product= rowAi.currentValue * rowBj.currentValue;
9                 C[i][k] += product;
10                rowBj.moveToNext();
11            }
12            rowAi.moveToNext()
13        }
14    }
15    // Remove all Ø elements from C
}
```

If A has m_A elements, we describe $m_A = \sum_{i=1}^{n} m_i$, where n is the number of rows, and m_i is the number of elements in row i of A. For each element m_{ik} in m_i, we go through each element in row m_k in B, which has at most n multiplications. So for each row in A, we have $m_i \cdot n$ multiplications at most. Expanding this to all rows, we have $\sum_{i=1}^{n} (m_i \cdot n) = m_A n$, which is our upper bound $O(m_A n)$.

Question 4 Page 3

We will now demonstrate the algorithm on example adjacency list matrices. (Where each non-zero entry is 1).
We translate B from letters to numbers.

A =
1: 3, 5, 7
2: 4, 8
3: 5, 8
4: 2, 6
5: 1, 2, 7, 8
6: 3
7: 1, 5, 6
8: 6, 7

B =
a 1: 3, 6, 7, 8
b 2: 3, 4
c 3: 1, 2, 7, 8
d 4: 2, 5, 6, 7
e 5: 4, 6, 7
f 6: 1, 4, 5, 8
g 7: 1, 3, 4, 5
h 8: 1, 3, 6

In this case, $n = 8$, $m_A = 19$, and $m_b = 28$. We start by looking at the first row of A, and iterating across each element. Taking the 3, we look at row 3 of B. We take each non-zero element of row three, and multiply by the value in row A_1 (since these are unit matrices, this is always 1.). We add that value to element to C_{ik}. The first row of C looks like:

$$C_1 = 1\ 1\ 0\ 0\ 0\ 0\ 1\ 1$$

Here we did 4 multiplications. Repeating with the remaining elements in row A_1 leads to:

$$C_1 = 2\ 1\ 1\ 2\ 1\ 1\ 2\ 1$$

So far we have 11 multiplications.

Question 4 Page 4

We are now finished with row C_1. Continuing with row A_2, we calculate C_2 as:

$C_2 = 1\ 1\ 1\ 0\ 1\ 2\ 1\ 0$

We took an additional 7 multiplications, for a current total of 18 multiplications. Continuing with the algorithm for the remainder of rows A_3 - A_8 yields the following final matrix

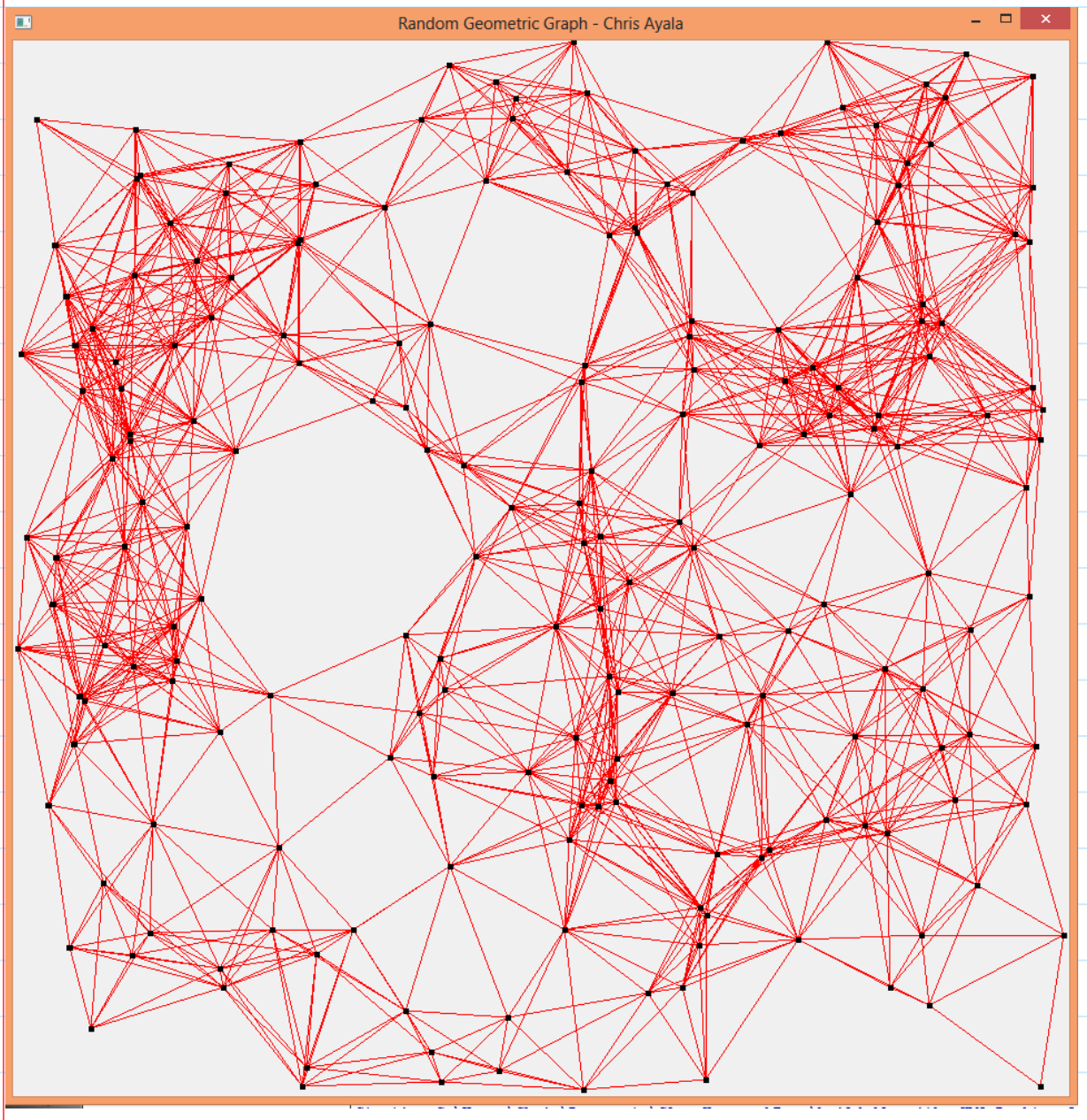
$$C = \begin{bmatrix} 2 & 1 & 1 & 2 & 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 0 & 1 & 2 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 2 & 1 & 0 \\ 1 & 0 & 1 & 2 & 1 & 0 & 0 & 1 \\ 2 & 0 & 4 & 2 & 1 & 2 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 2 & 1 & 2 & 2 & 2 \\ 2 & 0 & 1 & 2 & 2 & 0 & 0 & 8 \end{bmatrix}$$

Multiplications	
11	
7	24
6	
6	
13	23
4	
11	
8	19
66	

In the classic multiplication algorithm (which is $O(n^3)$), computing C would have taken $n^3 = 8^3 = 512$ multiplications. However, this algorithm demonstrates that, with sparse matrices stored as adjacency lists, we can achieve the same result in only 66 multiplications, which is beneath the upper bound of 152 multiplications.

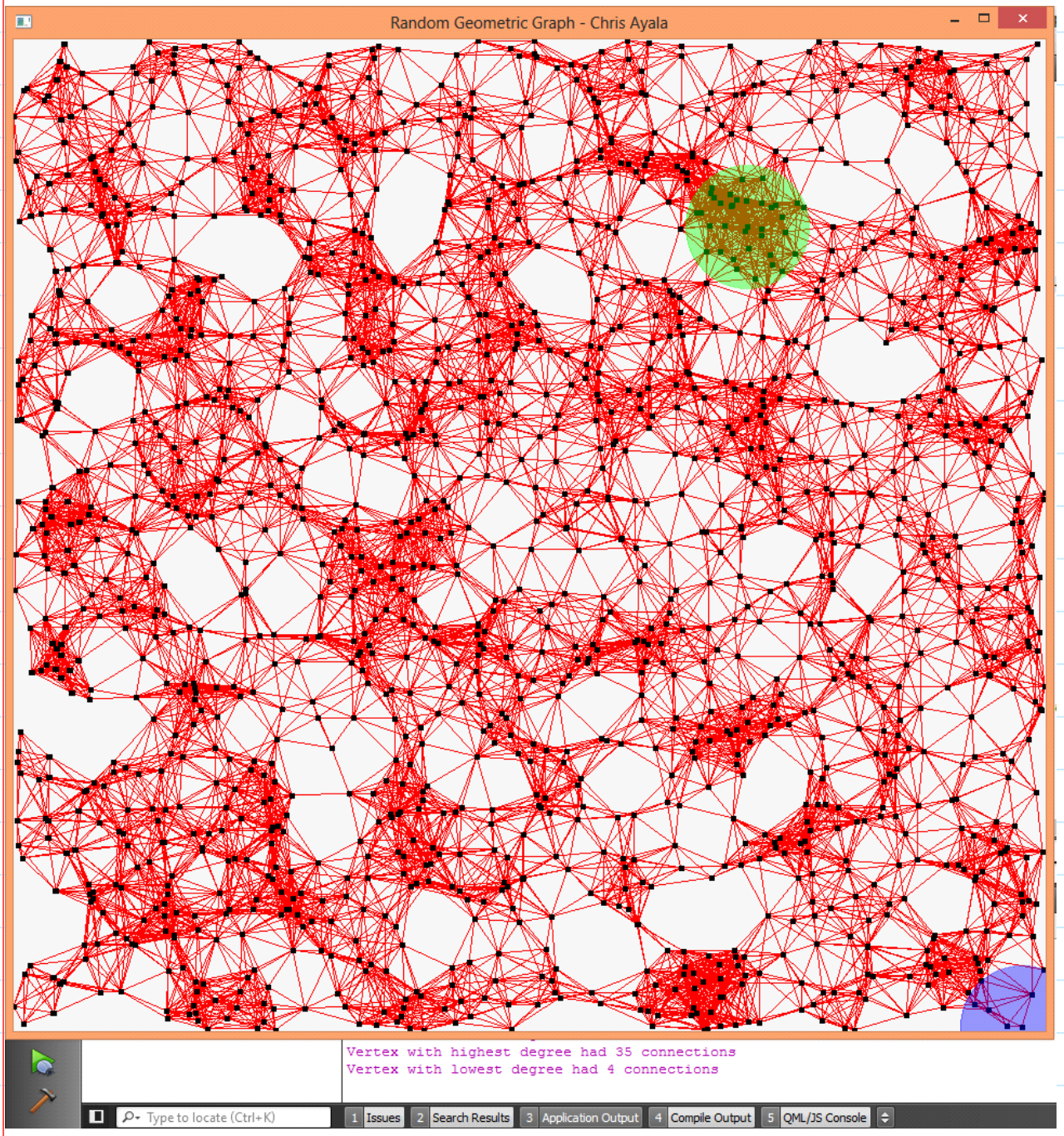
Question 5 Page 1

Below is a screenshot of an RGG with 200 random points and $r = 0.16$. This was done in the Qt C++ framework, and was designed to have a similar representation to an example shown in class.



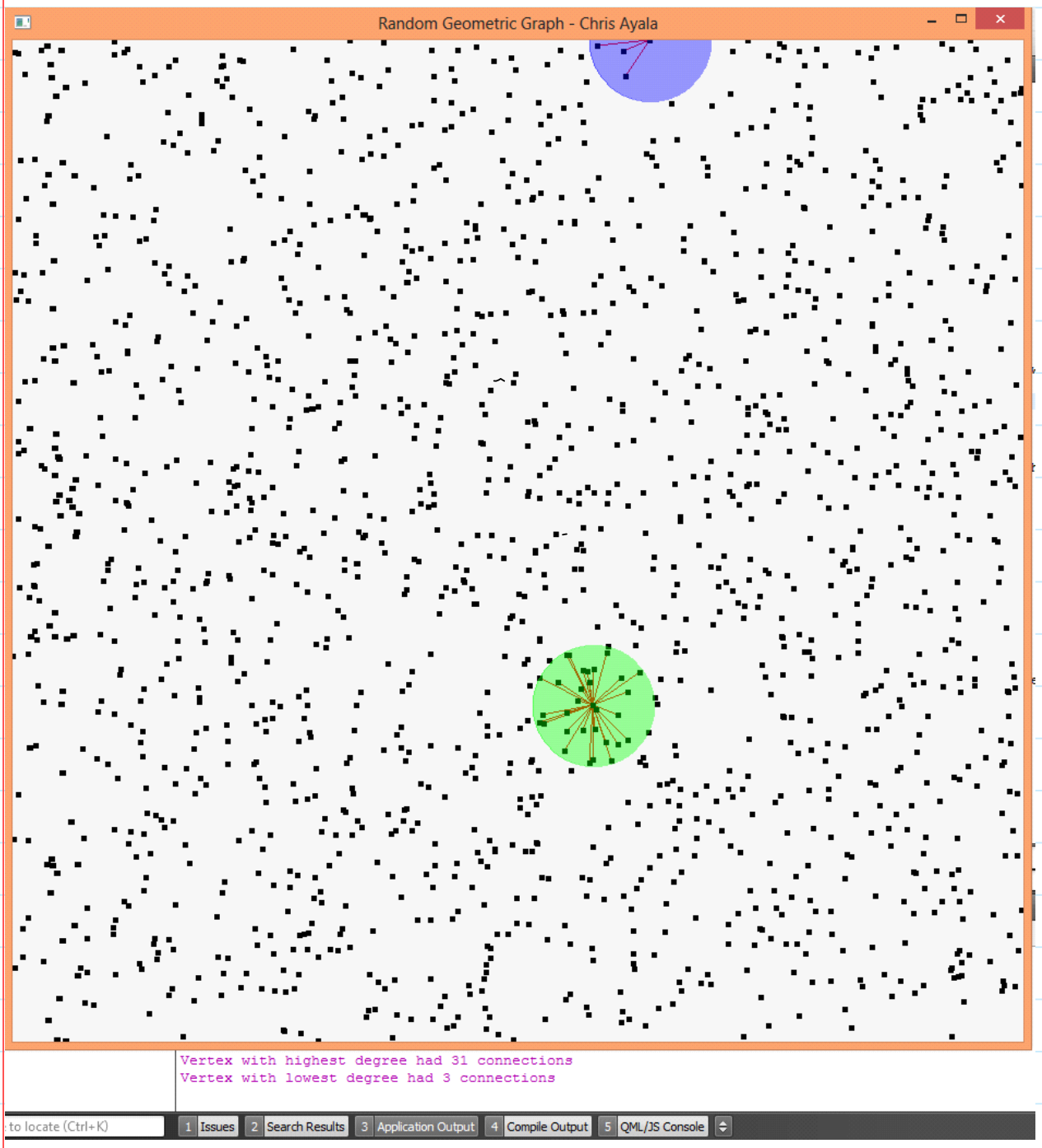
Question 5 Page 2

Below is an RGG with 1600 points and $r = 0.06$. The vertex with the highest degree is highlighted in green, and the vertex with the lowest degree is highlighted in blue. In this case, all edges are shown. The next page has a graph where only max/min degree vertices have edges. Max degree = 35, min degree = 4



Question 5 Page 3

Below is another RGG with 1600 vertices and r = 0.06. In this case, only max/min degree vertices have edges. Max degree = 31, min degree = 3. Note this is a different RGG than the previous example, so max and min degree vertices are in different locations



Question 6 Page 1

Question 6ai

It is simpler to consider the cases where a word is not a finite binary floating point number (FBFPN). In the case where the exponent is all 0's, the number is still a valid FBFPN, though not normalized (this will be important later). In the case where the exponent is a mix of 1's and 0's, the number is always a FBFPN.

Therefore, the only case of a word not being a FBFPN is when the exponent is all 1's (this can represent either $+\infty$, $-\infty$, or NaN). In a 64 bit number, the exponent takes 11 bits of space. In the case where the exponent is all 1's, the other 53 bits can be anything. Therefore, the odds of triggering this case is:

$$2^{53}/2^{64} = \frac{1}{2^{11}} = \frac{1}{2048} \approx .04883\%$$

Since this is the probability that a random word is NOT a FBFPN, the odds that it IS is

$$1 - \frac{1}{2048} = \boxed{\frac{2047}{2048} \approx 99.9512\%}$$

Question 6 Page 2

Question 6aii

The only difference between FBFPN and NBFPN (the first N = normalized) is when the exponent is all 0's. In this case it is an FBFPN, but not normalized. Therfore, we must add all cases where the exponent is all Ø's.

$$\left(\underbrace{2^{53}}_{\text{Probability that exponent is all Ø's}} + \underbrace{2^{53}}_{\text{Probability that exponent is all 1's}}\right) / \underbrace{2^{64}}_{\text{All possible combinations.}} = \frac{1}{1024} \approx 0.09766\,\%$$

This makes sense, since we are exactly double the number of input combinations that fail to be NBFPN. Therefore, the probability that a word is indeed NBFPN is:

$$1 - \frac{1}{1024} = \frac{1023}{1024} \approx 99.90234\,\%$$

Question 6bi

This question is exactly the same as 6ai, but with a different number of exponent bits. In a 128-bit word, 15 bits are for the exponent. Therefore, the probability of finding an FBFPN is:

$$1 - \left(\frac{2^{113}}{2^{128}}\right) = 1 - \left(\frac{1}{2^{15}}\right) = \frac{32767}{32768} \approx 99.996948\,\%$$

Question 6bii

Once again, this is fundamentally the same problem as 6aii. In this case, all 15 exponent bits must be all 0's or all 1's, so our probability of finding a NBFPN is

$$1 - \left(\frac{2^{113} + 2^{113}}{2^{128}}\right) = 1 - \left(\frac{1}{2^{14}}\right) = \frac{16383}{16384} \approx 99.993896\%$$