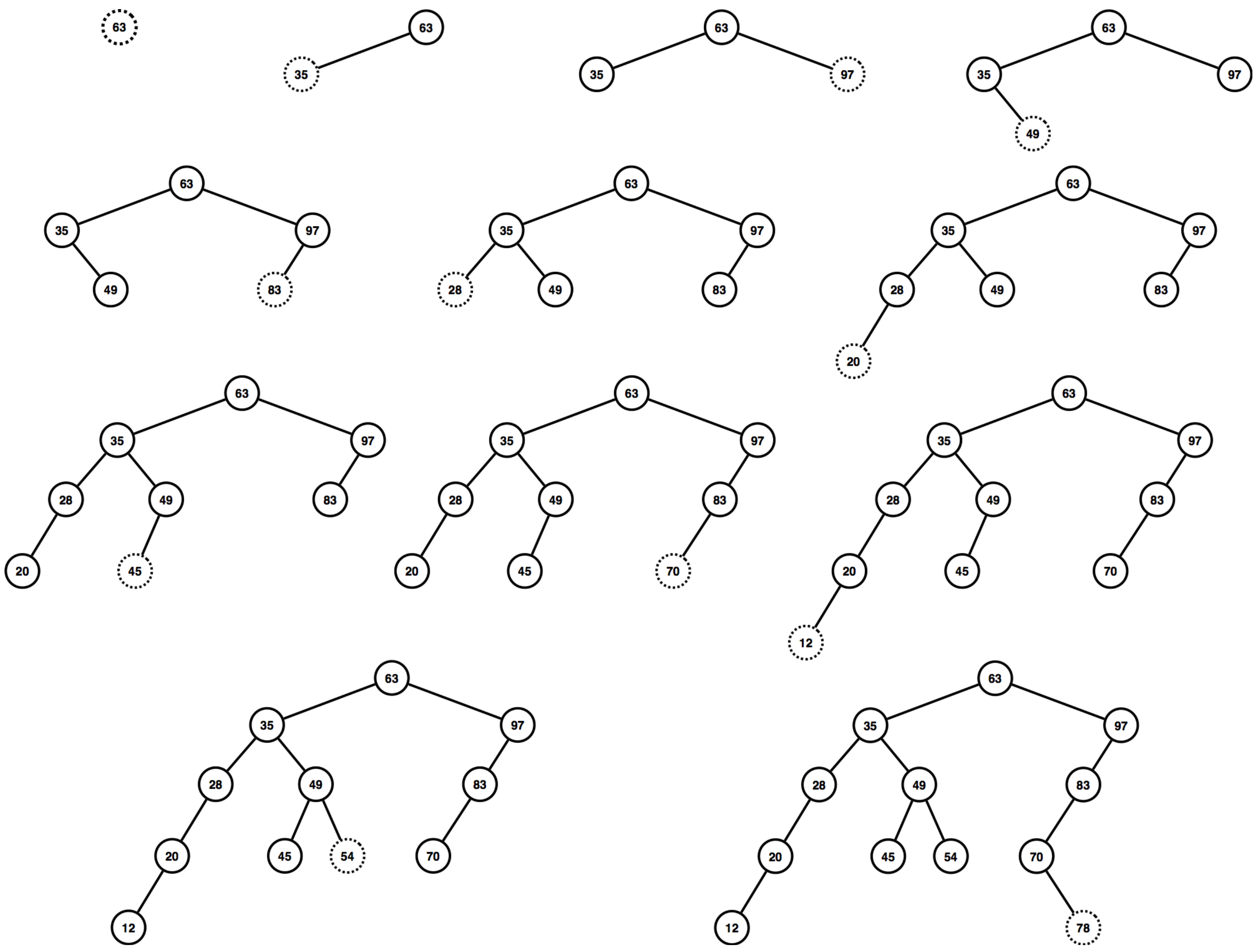
Data Structure Engineering for Algorithm Design

1. **[Data structure conversion]**
Design and analyze an on-the-fly algorithm for converting a binary search tree to an ordered doubly linked list following the inorder traversal of the data. Utilize a loop invariant (text p.17) to argue the correctness of your algorithm. Have your algorithm use as little extra space as possible, with "in place" the goal. Perform a walkthrough for the sequence S where the elements of S are first processed left-to-right to build the search tree. Then apply your algorithm to convert the binary search tree to a doubly linked list, showing the status of the conversion at each node of the search as it is consumed and moved into the doubly linked list.

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

BINARY SEARCH TREE CREATION
Nodes being inserted are highlighted by a dashed border



DISCUSSION

The structure of a binary search tree is very similar to the structure of a doubly linked list. Each node in the structure has the following composition:

```
struct Node {
     var data            // This is typically an int
     Node* left          // Pointer to the left child node
     Node* right         // Pointer to the right child node
}
```

Binary Search Trees utilize a structure where each node is greater than its left child node and less than it's right child node. As nodes are inserted, comparisons are made to determine where the new node should be placed. The end result is a structure where every node to the left of the root node is less than the value of the root node, and every value to the right is greater.

Nodes within a doubly linked list have a very similar structure to nodes of a binary search tree, the only notable difference is the variable naming. Instead of taking a hierarchical shape, doubly linked lists are arranged in a linear fashion where each node in the list has a pointer to the previous node and the next node. The structure of these nodes has the following composition:

```
struct Node {
     var data            // This is typically an int
     Node* previous      // Pointer to the previous node in the list
     Node* next          // Pointer to the next node in the list
}
```

Furthermore, these nodes are typically represented as a rectangular object with 3 sections; the first and last sections contain the previous and next pointers respectively, and the center section contains the payload (data).

Because of the strong similarities between these two structures, the conversion from a binary search tree to a doubly linked list is rather simple in theory. In order to make this conversion, each node must have its left and right child node pointers changed to point to that specific node's previous and next nodes. Finally, the first node, otherwise known as the *Head*, and last node must point to one another to complete the list.

IMPLEMENTATION

When considering the implementation of an algorithm to handle this conversion, a decision must be made to either utilize recursion or to utilize iteration. Due to the hierarchical nature of the tree and the structure of each node, recursion is an alluring option. In addition, the implementation of this algorithm must also take space utilization into consideration. An efficient solution should be able to convert the tree with minimal additional storage, resulting in an "in-place" efficiency.

By utilizing recursion and three node pointers, the conversion can be accomplished in-place. The recursive function will receive three node pointers; one pointer which will point to the current node being examined and two remaining pointers, which will be passed by reference, which will point to the head node and the previous node.

The first step in the conversion process is to recur until a specified base case is reached. The base case for this process is reaching the smallest, left-most node of the tree which will become the h*ead* of the doubly linked list. At this point, the pointer for the *head* will be updated and the *previous* pointer will be also be updated to indicate the current node has been examined. At the end of this process, a new doubly linked list will exist with only a single node pointing to itself with both the *previous* and *next* pointers.

After the base case, the tree conversion begins to take place. At each node, a series of rearrangements will occur which will update the linked list by adding a single node and rearranging where the previous node and the head node point. In essence, each recursive step will make the following changes:

1) Update the current node's left pointer to point to the previous node
2) Update the previous node's right pointer to point to the current node
3) Update the head's left pointer to point to the current node (which is now the last node in the list)
4) And finally, update the current node's right pointer to point to the beginning of the list (head)

After making these changes, the recursive function then calls itself and passes in the right node (the next largest node) and continues.

LOOP INVARIATION

In order to determine the correctness of this algorithm, loop invariation will be addressed. For this algorithm, we will use the following loop invariant: At the beginning of each recursion, all elements of the doubly linked list (those between *head* and *prev*, which is essentially the tail) are in sorted order. Furthermore, because of the nature of the original data structure, at the beginning of each recursion, all items in the doubly linked list will have a data value less than the data values in the remaining nodes of the binary search tree (which may include detached sub-tree(s) depending on the current state). To prove correctness, the invariant must hold during the *Initialization*, *Maintenance*, and *Termination* stages.

Initialization: Before the first recursion, all pre-existing nodes are still stored in the binary search tree and the pointers utilized by the function (*head* and *prev*) are both null. The invariant holds.

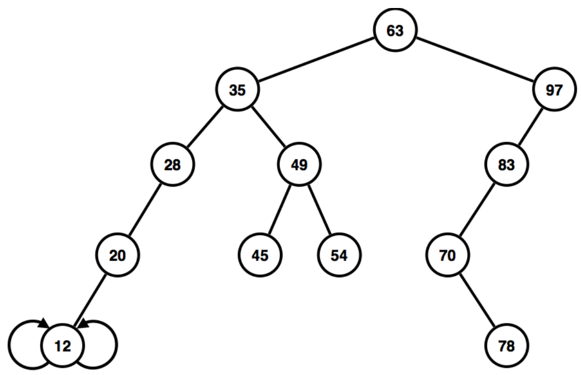
Maintenance: At any given recursion level, the doubly linked list will contain the nodes belonging to the original left sub-tree of the current node in sorted order, including the current node. Furthermore, any right nodes or right sub-trees belonging to the current node will have a greater value than all of the values in the doubly linked list. When a new node is detached from the binary search tree, the pointer of the tail of the structure (*prev*) is adjusted to point to the new node, which becomes the new tail. This node then points back to the previous node, and points to the head. Finally, the head's left pointer is adjusted to point at the current node, which becomes the new tail, ensuring sorted order and appropriate structure. The invariant holds.

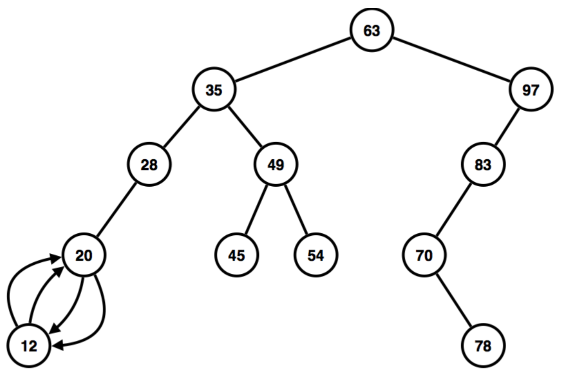
Termination: The outcome of the termination of the recursive sequence is no different than the termination of any specific level of the recursion; at the time of termination, the structure will contain all nodes from the binary search tree in sorted order with the *head*'s left pointer pointing to the tail of the structure (which is the final node stored in *prev*), and vice versa, with every node between the two having a pointer to the previous and next nodes. At this point, the binary search tree root is null, indicating the tree is empty and the conversion has completed. The invariant holds and the algorithm is correct.

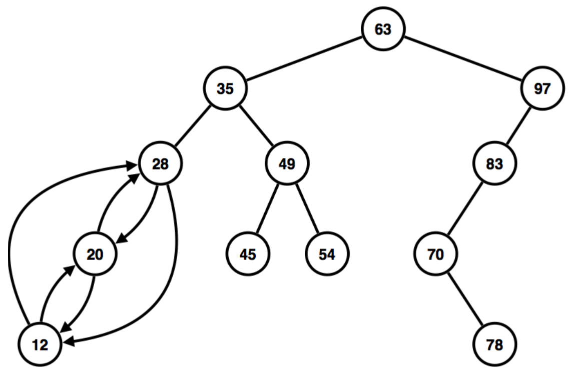
ALGORITHM PSEUDOCODE

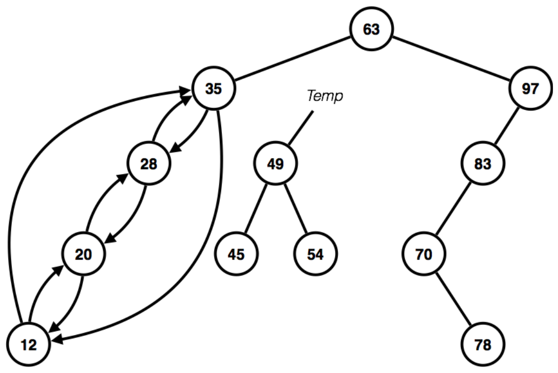
```
convertTreeToList(Node* curr, Node*& prev, Node*& head) {
     // Check to see if node exists, if not, return
     if(!curr) {
          return;
     }

     // Recur with left node
     convertTreeToList(curr->left, prev, head);

     // Once spawned recursion completes…
     // Set the left pointer of the current node to the previous node
     curr->left = prev;

     if(prev) {
          // RECURSIVE CASE
          // Set the right pointer of the previous node to the current
          // node
          prev->right = curr;
     } else {
          // BASE CASE
          // If smallest node reached, set head to curr
          head = curr;
     }

     // Change the left pointer of the head (which points to the end
     // of the linked list) to point to the current node
     head->left = curr;

     // Save a reference to the right node because the right
     // pointer of the current node will be changed shortly
     Node* temp = curr->right;

     // Because this is now the new end of the list, change the right
     // pointer to point to the beginning of the list (head)
     curr->right = head;

     // Mark the current node as visited by making prev point to it
     prev = curr;

     // Repeat with right node (which will return if it doesn’t exist)
     convertTreeToList(right, prev, head);
}
```

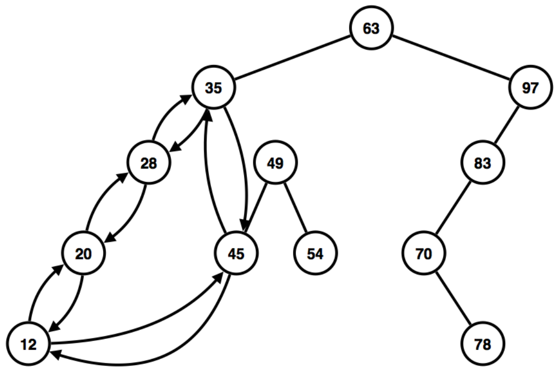
CONVERSION PROCESS
The diagrams below illustrate the conversion process. The diagrams progress from left to right and then from top to bottom.

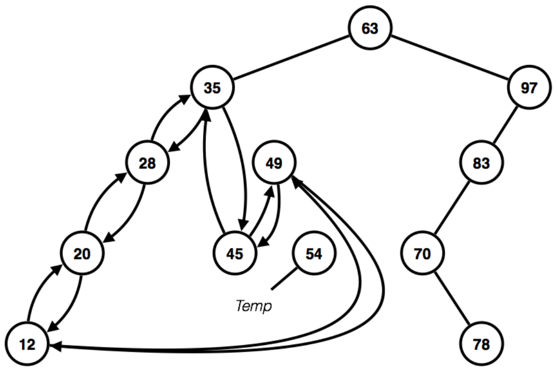


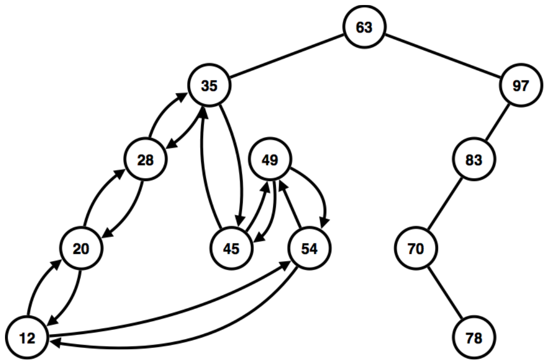


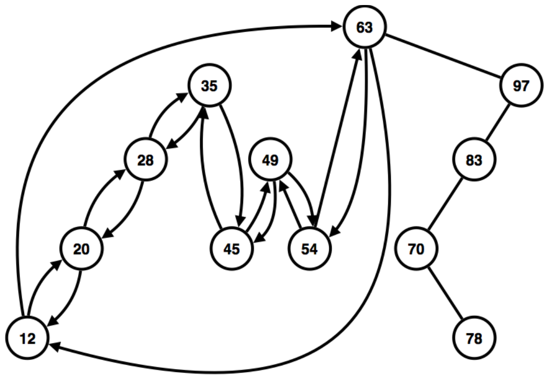


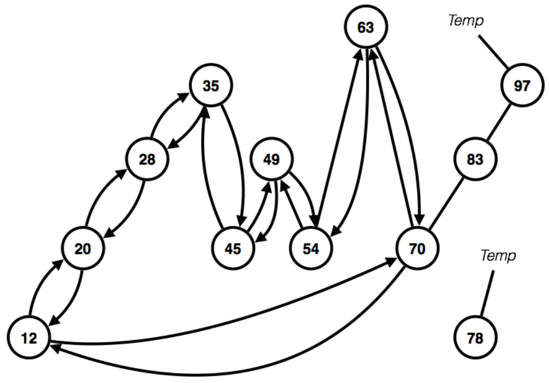


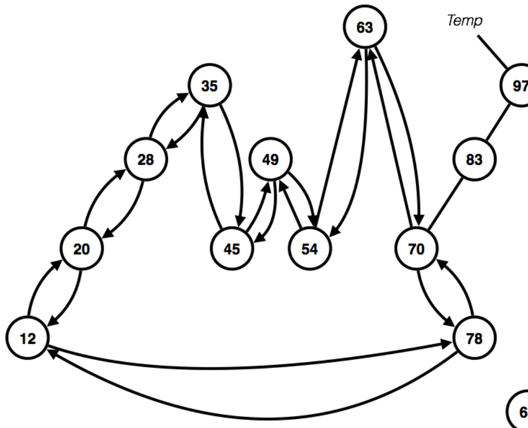


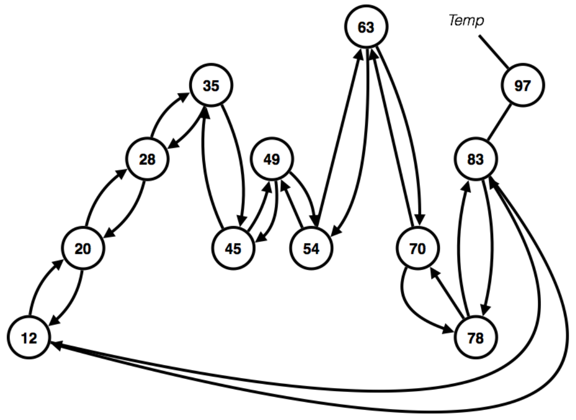


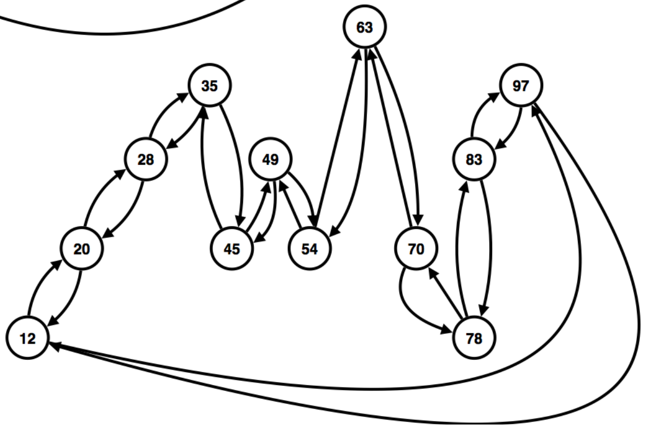

63
35
97
28
49
83
20
45
54
70
12
78


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


63
Temp
35
97
28
49
83
20
45
54
70
Temp
12
78


63
Temp
35
97
28
49
83
20
45
54
70
12
78

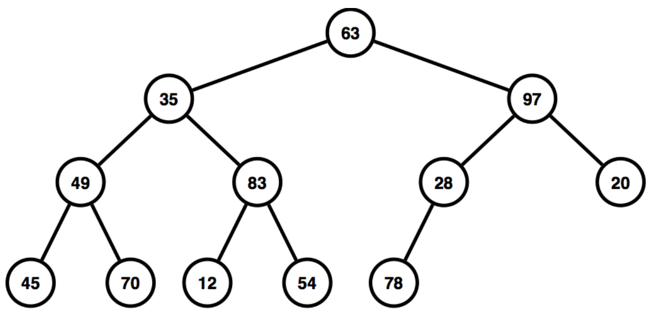

63
Temp
35
97
28
49
83
20
45
54
70
12
78


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

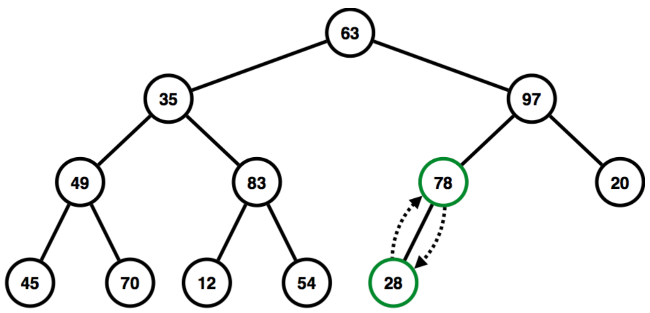
2. **[Heapify]**

The array form of a heap (i.e. a binary heap array) has the children of the element at position i in positions (left @ 2i, right @ 2i+1). For a balanced ternary heap array the three children of the element at position i are at positions (left @ 3i-1, middle @ 3i, rig why ht @ 3i+1). The term "Heapify" refers to the algorithm that builds a binary heap array from right-to-left (leaf-to-root in binary tree form) given the size of the array to be built. The term "Ternary Heapify" here refers to the analogous algorithm that builds a balanced ternary heap array in right-to-left order.

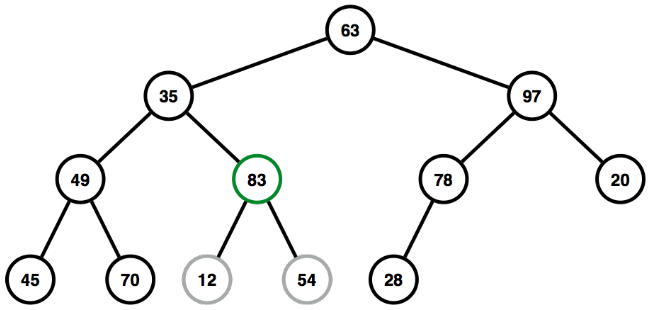
a. Heapify the sequence S of Problem 1, and give the number of comparisons required.



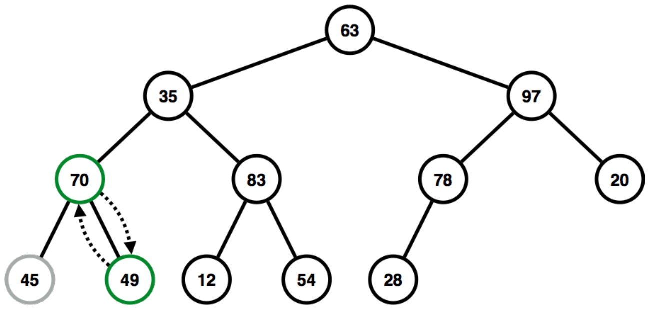
Initial Structure



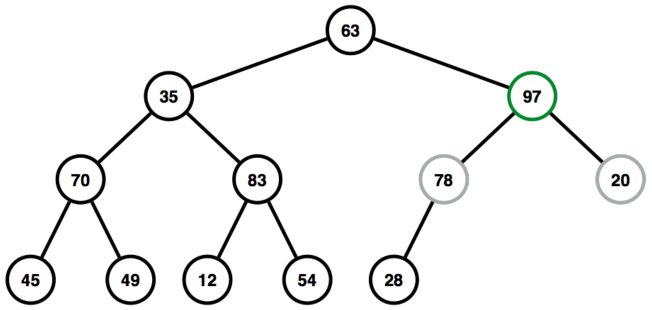
+1 Comparison



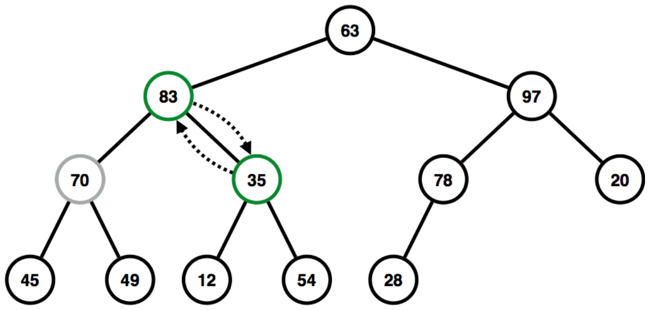
+2 Comparisons



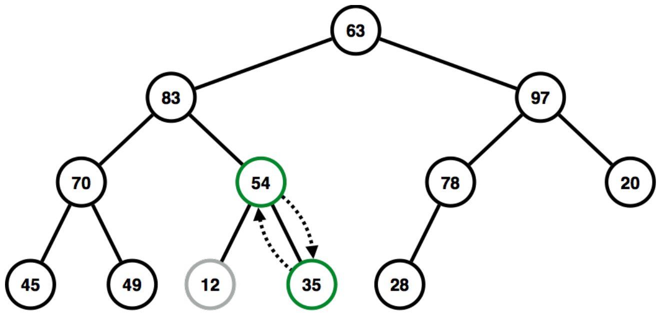
+2 Comparisons



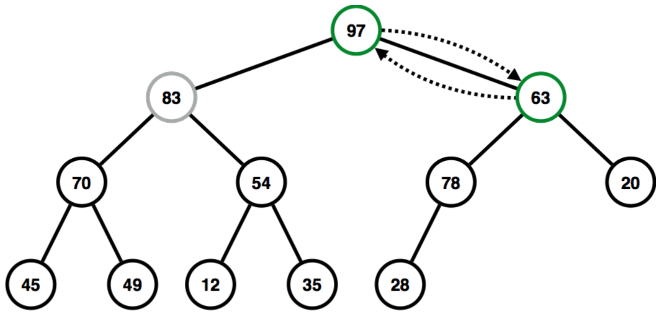
+2 Comparisons



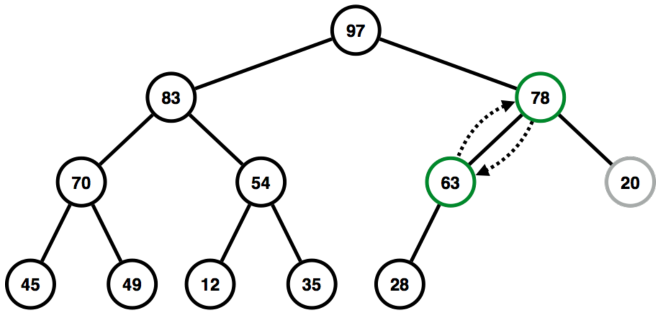
+2 Comparisons



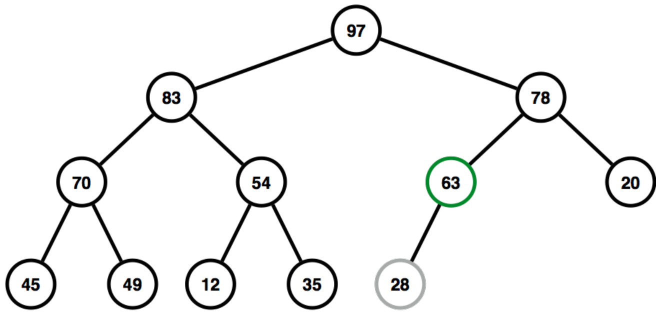
+2 Comparisons



+2 Comparisons



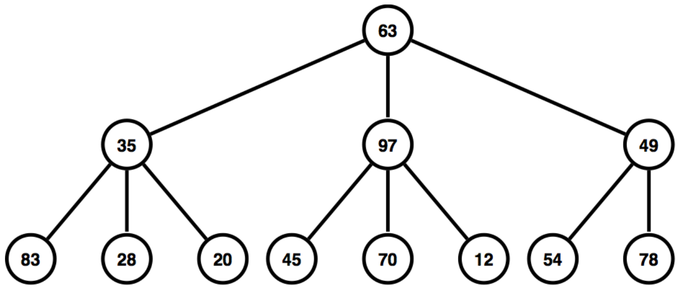
+2 Comparisons



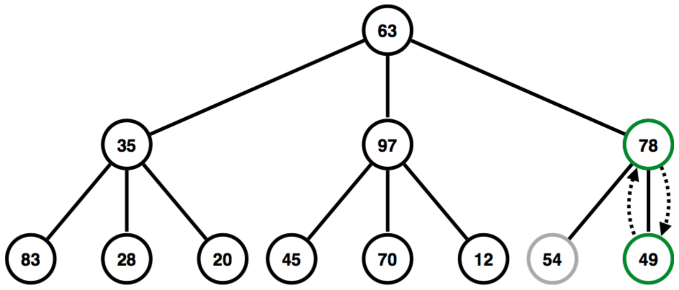
+1 Comparisons

Total Comparisons: 16

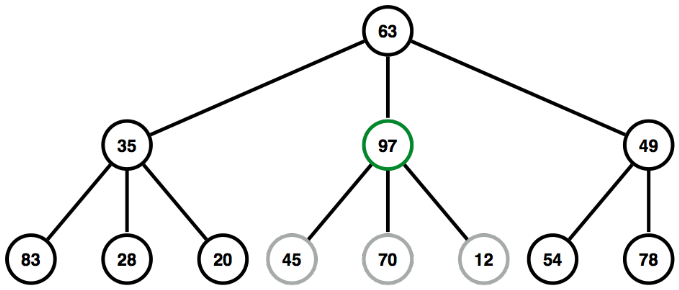
b. Do the same as (a) for Ternary Heapify.



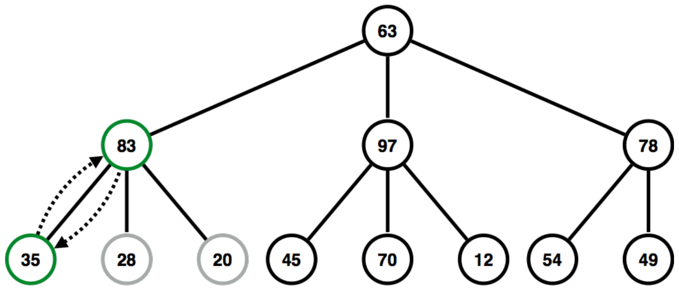
Initial Structure



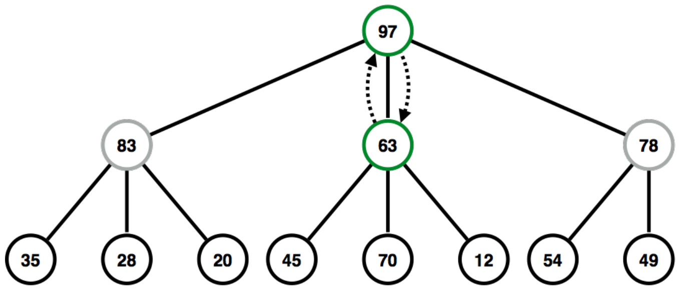
+2 Comparisons



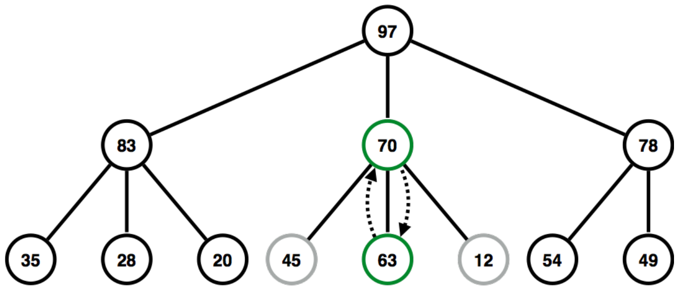
+3 Comparisons



+3 Comparisons



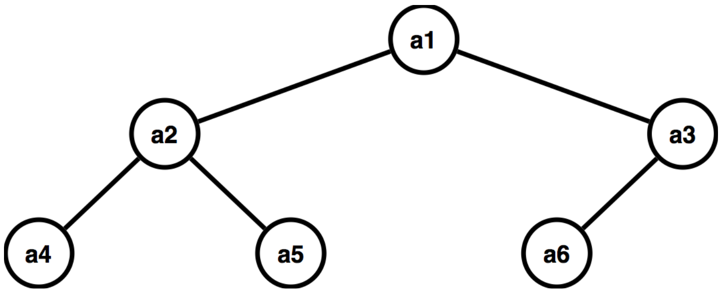
+3 Comparisons

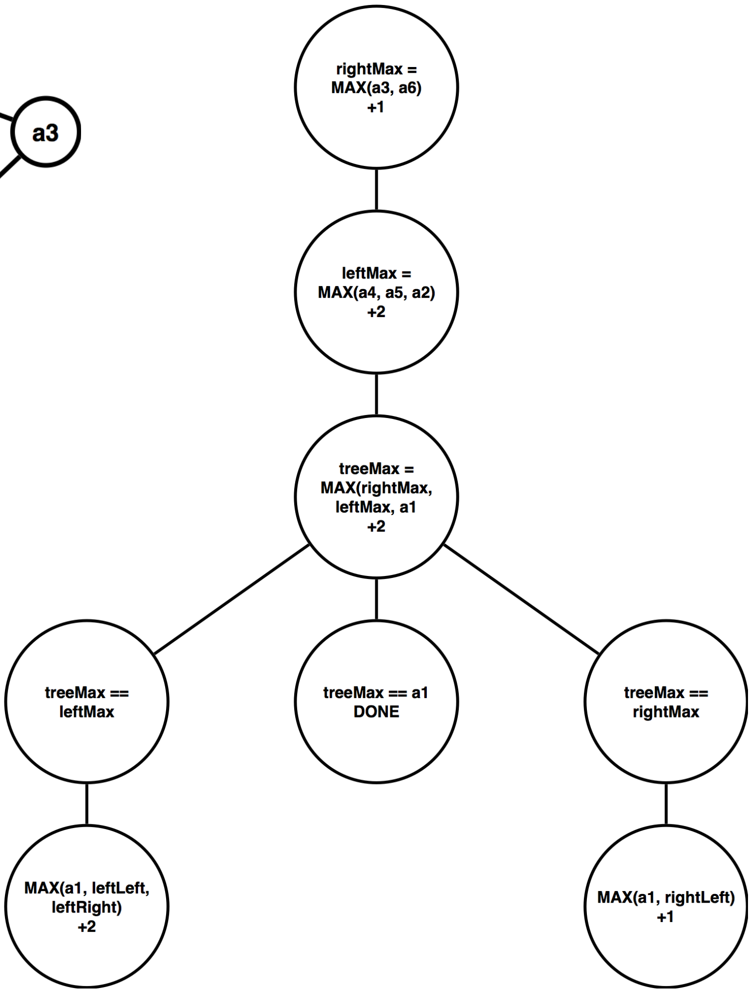


+3 Comparisons

Total Comparisons: 14

c. Build the decision tree determined by the comparisons employed to Heapify a six element list: a1, a2, a3, a4, a5, a6.What is then the worst case and average case number of comparisons to Heapify six elements? Is this optimal in either case?




rightMax = MAX(a3, a6) +1
leftMax = MAX(a4, a5, a2) +2
treeMax = MAX(rightMax, leftMax, a1 +2
treeMax == leftMax
treeMax == a1 DONE
treeMax == rightMax
MAX(a1, leftLeft, leftRight) +2
MAX(a1, rightLeft) +1

A decision tree for the binary heapification of a1 through a6 is a representation of the possible comparisons that could be made during the process of heapifying the elements. This tree takes the approach of quantifying all possible number of comparisons.

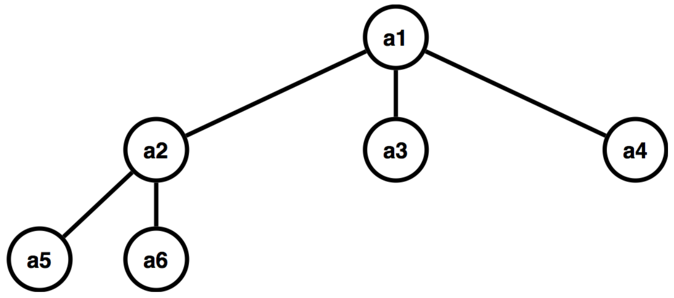
The first step in the process is to determine the largest value in the right sub-tree. If a6 is greater than a3, the two are swapped. Next, if the larger value between a4 and a5 is larger than a2, they swap as well. Once at the root of the tree, another comparison takes place. This comparison dictates whether the number of comparisons will be equal to the min, average, or max number of comparisons If the root is the max of those three, the heapification is complete, taking only 5 comparisons. If the maximum of the right tree is larger than the root, the two swap, requiring an additional comparison between the new parent of the right sub-tree its child. Finally, if the maximum value of the left sub-tree is larger than the root, the two values are swapped. This swap, however, results in 2 more comparisons before the heapification is complete.

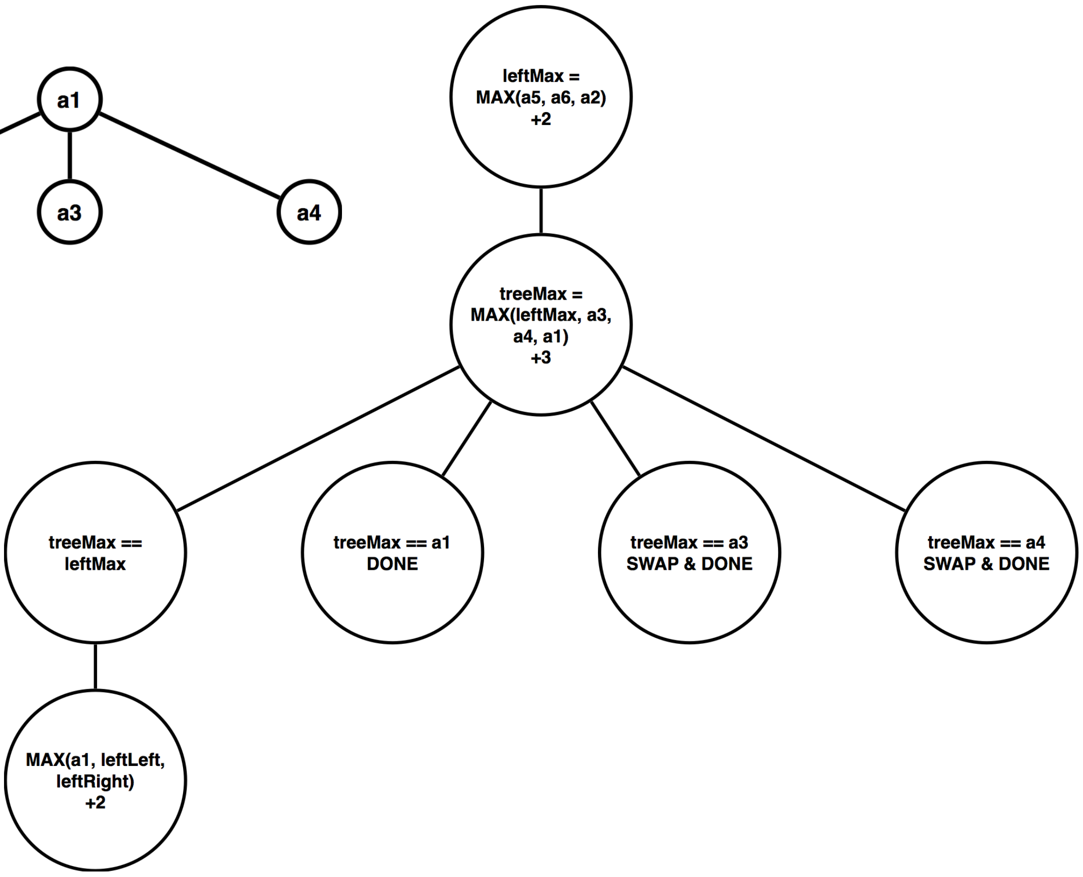
Assuming random values of elements a1 through a6, all three possible root-comparison outcomes are equally likely. As a result, the worst and average case comparisons are as follows:

Worst Case Comparisons: 7
Average Case Comparisons: 6

Neither the worst case nor the average case are optimal. The optimal solution for the heapification is that the data is already sorted appropriately and none of the comparison, including the root comparison, result in a swap. This outcome would result in 5 comparisons.

d. Do the same as (c) for Ternary Heapify.




leftMax = MAX(a5, a6, a2) +2
treeMax = MAX(leftMax, a3, a4, a1) +3
treeMax == leftMax
treeMax == a1 DONE
treeMax == a3 SWAP & DONE
treeMax == a4 SWAP & DONE
MAX(a1, leftLeft, leftRight) +2

A decision tree for the ternary heapification of a1 through a6 is a representation of the possible comparisons that could be made during the process of heapifying the elements. This tree takes the approach of quantifying all possible number of comparisons.

This decision tree is very similar to that of the binary heapification, however, now there is only one sub-tree comparison prior to the root-comparison. After the root comparison, 3 of the 4 outcomes results in either a sorted heap or requires a single swap, without additional comparison(s), to finish sorting. The 4th outcome, occurring when the max value of the left sub-tree is greater than a1, requires a swap and an additional 2 comparisons.
Again, assuming random values of elements a1 through a6, all four possible root-comparison outcomes are equally likely. As a result, the worst and average case comparisons are as follows:

Worst Case Comparisons: 7
Average Case Comparisons: 5

Because the the only non-optimal outcome occurs when the left sub-tree is greater than the root, and because this outcome only accounts for approximately 1/4th of the total outcomes, the average case *is* optimal. This case can be reached whenever the root is the max value, or when the max value of the tree is a3 or a4.

3. **[Graph (Symmetric Matrix) Reordering]**

Let the adjacency list representation of a graph (0, 1 symmetric matrix) be given by a packed edge list. Describe an algorithm for reordering the edge list in place so all adjacency lists follow a new vertex ordering given by a permutation p(1), p(2), ..., p(n). Apply your algorithm to the graph given by the packed edge list stored as example HW2 - 3 graph. Reorder by determining a maximum adjacency search order with "ties" broken lexicographically.

HW2 - 3 graph in adjacency list form:

- 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;

Given the representation of a graph as a packed edge list, it's a relatively straight forward process to reorder the edge list following a new vertex ordering, such as maximum adjacency. The only consideration for the implementation of the algorithm, is how to complete such a reordering without using additional space, which is "in-place".

The algorithm is based on the principle that edge of the graph is listed twice in the adjacency list; each edge from *n* to *m* is listed as an edge for both *n* and *m*. In other words, there are duplicates of each edge. The algorithm can take advantage of this redundancy in order to free up space for the algorithm to utilize in order to ensure the process is completed "in-place".

The vertex adjacency representation of the packed edge list is as follows:

Vertex	a	b	c	d	e	f	g	h
Adjacency (Degree)	4	2	4	4	3	4	4	3

When ordered according to maximum adjacency and then lexicographically, the result is a vertex adjacency of vertices ordered from highest adjacency to least, which, when renamed lexicographically (i.e., from "A" to "H"), is the desired end result of the algorithm in vertex adjacency form:

Vertex	a	c	d	f	g	e	h	b
Adjacency (Degree)	4	4	4	4	4	3	3	2
Final Vertex Name	A	B	C	D	E	F	G	H

The first step of the algorithm is to create the data structure containing the packed edge list:

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

Now, the algorithm will sort the packed edge list according to the specified permutation; in this case, the packed edge list will be sorted according to maximum adjacency, with "ties' broken lexicographically:

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

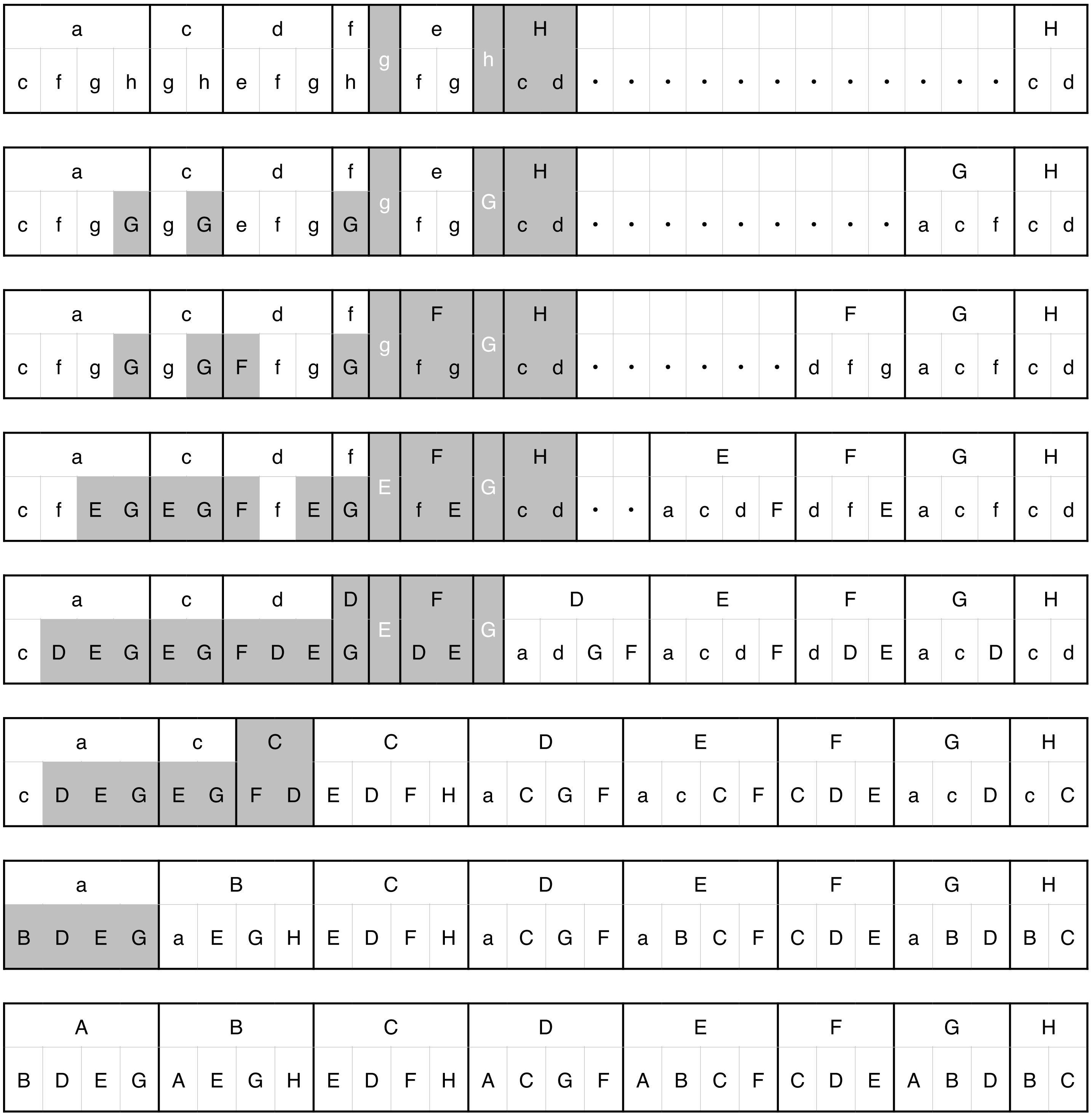
Next, the algorithm will "trim" the sorted packed edge list by iterating though the list and removing the duplicate edge listings lexicographically. In order to ensure that only one half of an edge pair are is removed, edges are removed only when the edge's associated vertex's name is lexicographically *less* than the that of the current vertex. The trimmed, packed edge list is illustrated below:

a				c				d				f				g				e			h			b	
c	f	g	h	•	•	g	h	•	e	f	g	•	•	•	h	•	•	•	•	•	f	g	•	•	•	c	d

At this point, the packed edge list contains half of the original nodes. Next, the algorithm will "compresses" the packed edge list by pushing the data to the front of the structure and sorting their respective edge connections. This process has a two-fold purpose. By moving the packed edge list to the front half of the data structure, the second half of the structure presents a contiguous block of space which the algorithm can utilize to ensure the operation is performed in-place. In addition, sorting the packed edge list lexicographically will allow the decompression process to occur without overwriting unique data. The sorted and compressed packed edge list is as follows:

a				c		d			f	g	e		h	b															
c	f	g	h	g	h	e	f	g	h		f	g		c	d	•	•	•	•	•	•	•	•	•	•	•	•	•	•

After the compression has finished, the algorithm must begin decompressing the packed edge list. At the same time, it must begin converting the previous names of the vertices to the new, sorted order. This process takes place from the last lexicographic character to the first, and at each vertex, references to the old vertex name is replaced with the new vertex name. The decompression process is illustrated below with uppercase characters representing the new vertex names. Furthermore, as the packed edge list is decompressed, data that has been consumed will be greyed out to illustrate that no unique data is overwritten.


a c d f e H H
c f g h g h e f g h g f g h c d • • • • • • • • • • • • c d
a c d f e H G H
c f g G g G e f g G g f g G c d • • • • • • • • • a c f c d
a c d f F H F G H
c f g G g G F f g G g f g G c d • • • • • • d f g a c f c d
a c d f F H E F G H
c f E G E G F f E G E f E G c d • • a c d F d f E a c f c d
a c d D F D E F G H
c D E G E G F D E G E D E G a d G F a c d F d D E a c D c d
a c C C D E F G H
c D E G E G F D E D F H a C G F a c C F C D E a c D c C
a B C D E F G H
B D E G a E G H E D F H a C G F a B C F C D E a B D B C
A B C D E F G H
B D E G A E G H E D F H A C G F A B C F C D E A B D B C

After the algorithm completes the decompression process, the resulting packed edge list is sorted by the appropriate permutation, maximum adjacency in this case, and is renamed lexicographically accordingly.

4. **[Sparse Matrix Multiplication]**

Describe an efficient sparse matrix multiplication procedure for two n x n matrices stored in "adjacency list" form. Assume matrix A has m_A non zero entries and matrix B has m_B non zero entries, with $n << m_A << m_B << n^2$. Describe why your algorithm will generate the n × n matrix product in adjacency list form in time O (n m_A).

Apply your algorithm for multiplying the two 8 × 8 0,1 matrices A_1, B_1 where A_1 is given by the following adjacency list of unit entries and B_1 is given by the adjacency list of unit entries of Problem 3. Count the actual number of multiplications used in this matrix product and explain why it is considerably less than the bound n m_A (which is 8 × 19 =152) in this case.

Adjacency List For 0,1 Matrix A_1

- 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

A sparse matrix different from a traditional matrix in that the majority of it's elements are zero. While this may not appear to have a significant impact on matrix multiplication, in actuality, it allows for a wide variety of algorithms to exploit the sparsely populated matrix to make matrix multiplications significantly more efficient.

Assuming that matrix A is an n x m matrix and that matrix B is an m x p matrix, denoted by the following[1]:

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix}$$

The product of the A and B results in an n x p matrix, *C*, with the following composition[2]:

$$\mathbf{AB} = \begin{pmatrix} (\mathbf{AB})_{11} & (\mathbf{AB})_{12} & \cdots & (\mathbf{AB})_{1p} \\ (\mathbf{AB})_{21} & (\mathbf{AB})_{22} & \cdots & (\mathbf{AB})_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ (\mathbf{AB})_{n1} & (\mathbf{AB})_{n2} & \cdots & (\mathbf{AB})_{np} \end{pmatrix}$$

The value for any given *i,j* of *C*, represented by C_{ij}, is calculated by multiplying A_{ik} (which is the entire *i* row of A) by B_{jk} (which is the entire *j* column of *B*) where *k* = 1, 2, … *m*. This calculation is represented by the following equation[3]:

$$(\mathbf{AB})_{ij} = \sum_{k=1}^{m} A_{ik}B_{kj} \,.$$

[1] https://upload.wikimedia.org/math/a/d/6/ad67996e9dec70cd1564b02808ee0e86.png

[2] https://upload.wikimedia.org/math/5/f/e/5fee367c1a614b2f1b6096ccecfe12c6.png

[3] https://en.wikipedia.org/wiki/Matrix_multiplication

Traditional matrix multiplication like that described above requires an overall complexity of O(n^3), which represented both best and worst cases. It also takes O(n) to calculate the value of C_{ij}. As a result, a significant amount of research has been conducted into methods of increasing the efficiency of matrix multiplication, specifically in regards to spare matrices.

One method of increasing the efficiency of this involves storing only the non-zero elements within an adjacency list as the elements with a value of 0 do not affect the final values of C. Essentially, the algorithm begins by creating an empty matrix, C_1. Next, the algorithm iterates through all elements for A_1 and performs a specific calculation at each non-zero element of A_1, which in make up m_a. At each non-zero element, you identify which values in C_1 are affected by the value, perform the necessary calculations against values in B_1, and add the result to that element within C_1. After iterating through all ma non-zero elements of A_1, the results stored within C_1 are equivalent to the product of A_1 and B_1.

Given adjacency lists A$_1$ and B$_1$, this algorithm can demonstrate the considerable increase in efficiency by counting the number of multiplications that must take place. A_1 can be represented in the following adjacency list:

A₁ = 1: 3, 5, 7	4: 2, 6	7: 1, 5, 6
2: 4, 8	5: 1, 2, 7, 8	8: 6, 7
3: 5, 8	6: 3	

The adjacency list for matrix B_1 can be calculated from the adjacency list from Question 3 buy substituting each character's numerical value (between 1 and 26) for each character. The resulting adjacency list is as follows:

B₁ =		c: a, b, g, h;	= 3: 1, 2, 7, 8	f: a, d, e, h;	= 6: 1, 4, 5, 8
a: c, f, g, h;	= 1: 3, 6, 7, 8	d: b, e, f, g;	= 4: 2, 5, 6, 7	g: a, c, d, e	= 7: 1, 3, 4, 5
b: c, d;	= 2: 3, 4	e: d, f, g;	= 5: 4, 6, 7	h: a, c, f;	= 8: 1, 3, 6

For these matrices and this calculation, n = 8, m_a = 19, and m_b = 28. By applying the above algorithm to A_1 and B_1, the resulting matrix is as follows:

$$C_1 = \begin{bmatrix} 0&1&1&0&1&1&0&1 \\ 1&1&1&0&1&0&1&0 \\ 1&0&1&1&0&0&1&0 \\ 1&0&1&0&1&0&0&1 \\ 0&0&0&0&1&0&1&1 \\ 1&1&0&0&0&0&1&1 \\ 1&0&1&0&1&0&0&0 \\ 0&0&1&0&0&0&0&1 \end{bmatrix}$$

Utilizing the above algorithm to take advantage of the sparsity of both A_1 and B_1, the product of the two matrices can be completed in approximately 60 multiplications, which is consistent with O (n m_a). This is a substantial improvement over more traditional matrix multiplication complexities which typically have a time complexity of O(n^3).

5. **[Floating Point Number Formats]**

A floating point number is an IEEE standardized formulaic, binary representation of a real-number approximation. Floating point numbers are used in order to increase the effective range of values that can be stored, however, the trade off for this extended range is a reduction in precision. This format reserves specific bit ranges of the total bits allocated and uses those reserved bits to raise a *significand* (or *mantissa*) to an *exponent* with a *sign*.

In order to further extend the range of representable values, the IEEE standardized floating point number has several reserved combinations of bits to represent "special values". Depending on the bit combination of the exponent and the mantissa, the resulting value can be either a "special value" (e.g., ± ∞, *NaN*) or a finite number. Finite numbers in turn have two distinct groups of values: normalized numbers and denormalized numbers. A denormalized number is a non-zero significand which does not use the "assumed 1" which typically precedes the decimal; instead, the decimal is preceded by a 0. Normalized numbers consist of values which have a 1 preceding the decimal.

a) What is the probability that a random 64-bit word represents a:

i. finite binary floating point number, and

Continuing the discussion above, Finite Binary Floating Point Numbers (henceforth referred to as FBFPNs) are the floating point values in the finite 'bucket', which include both normalized and denormalized values. Because there are a finite, and relatively small, number of combinations which generate invalid FBFPNs, the following analysis will calculate the probability that a 64-bit word will result in an *invalid* FBFPN (which may be either a positive or negative 0 or one of the other special numbers discussed above) in order to calculate the probability it will be valid.

Invalid FBFPNs consist of the following possible bit combinations[4]:

Type	Exponent	Fraction	Sign	Probability
Positive Zero	0 (All 0s)	0	0	$1 \div 2^{64}$
Negative Zero	0 (All 0s)	0	1	$1 \div 2^{64}$
Infinity (±)	$2^e - 1$ (All 1s)	0	*	$2^{53} \div 2^{64}$
NaN	$2^e - 1$ (All 1s)	non-zero	*	

In order to calculate the probability that a given 64-bit word is an invalid FBFPN, the total probability one of the above combinations occurring must be calculated. Positive and negative zero make up 2 possible combinations out of 2^{64} combinations, resulting in a total probability of $2 \div 2^{64}$. This is included under the assumption that signed 0 is a non-finite number because it is a special reserved number under the IEEE standard. Additionally, infinity and NaN occur when

[4] https://en.wikibooks.org/wiki/Floating_Point/Special_Numbers

the exponent is all 1s and the bit combination of the mantissa determines which of the two occurs. Because the mantissa combinations are exhaustively covered between the two and because the sign can be either 0 or 1, the probability of either of these occurring is entirely dependent upon the remaining 53 bits. As a result, the cumulative probability of either occurring is $2^{53} \div 2^{64}$.

Overall, the total probability of any of these four combinations occurring is the sum of $2 \div 2^{64}$ and $2^{53} \div 2^{64}$, or $\mathbf{(2^{53} + 2) \div 2^{64}}$.

In order to determine the probability of a *valid* FBFPN occurring can be calculated by subtracting the above calculation from one.

Probability that a random 64-bit word represents a finite binary floating point number:

$1 - \mathbf{(2^{53} + 2) \div 2^{64}} = .999511719 \asymp$ **99.95%**.

ii. a normalized binary floating point number, and,

Normalized Binary Floating Point Numbers (NBFPN) are a subsection of finite numbers where the number preceding the decimal point is an “assumed 1”.

Similar to the calculation from the section above, in order to calculate the probability that a random 64-bit word represents a valid NBFPN, first the probability of a denormalized number will be calculated. In order for a random 64-bit word to be a denormalized number, it must be a finite number, it must have an exponent of 0 (all bits of the exponent are 0s), and it must have a non-zero mantissa.

Type	Exponent	Fraction	Sign	Probability
Positive Zero	0 (All 0s)	0	0	$1 \div 2^{64}$
Negative Zero	0 (All 0s)	0	1	$1 \div 2^{64}$
Infinity (±)	2^e - 1 (All 1s)	0	*	$2^{53} \div 2^{64}$
NaN	2^e - 1 (All 1s)	non-zero	*	
Denormalized	0 (All 0s)	non-zero	*	$(2^{53} - 2) \div 2^{64}$

In order to generate a denormalized number, all exponent bits (11) must be zero and at least one of the mantissa bits must be 1, there are two cases where an exponent with all 0s can result in a non-denormalized number. As a result, the probability of a denormalized number can be represented as the following: $(2^{(64-11)} - 1) \div 2^{64}$, or $(2^{53} - 2) \div 2^{64}$.

Overall, the total probability of any of these four combinations occurring is the sum of $2 \div 2^{64}$, $2^{53} \div 2^{64}$, and $(2^{53} - 2) \div 2^{64}$, or $\mathbf{(2^{53} + 2^{53}) \div 2^{64}}$.

In order to determine the probability of a *valid* NBFPN occurring can be calculated by subtracting the above calculation from one.

Probability that a random 64-bit word represents a normalized binary floating point number:

$1 - \mathbf{(2^{53} + 2^{52} + 2) \div 2^{64}} = .999023438 \approx$ **99.90%**.

b) Do the same as (a) for a random 128-bit word.

The following calculations will use the same logic discussed in *part (a)* above.

i. finite binary floating point number, and

Invalid FBFPNs consist of the following possible bit combinations:

Type	Exponent	Fraction	Sign	Probability
Positive Zero	0 (All 0s)	0	0	$1 \div 2^{128}$
Negative Zero	0 (All 0s)	0	1	$1 \div 2^{128}$
Infinity (±)	2^e - 1 (All 1s)	0	*	$2^{113} \div 2^{128}$
NaN	2^e - 1 (All 1s)	non-zero	*	

Overall, the total probability of any of these four combinations occurring is the sum of $2 \div 2^{128}$ and $2^{113} \div 2^{128}$, or $\mathbf{(2^{113} + 2) \div 2^{128}}$.

In order to determine the probability of a *valid* FBFPN occurring can be calculated by subtracting the above calculation from one.

Probability that a random 128-bit word represents a finite binary floating point number:

$1 - \mathbf{(2^{113} + 2) \div 2^{128}} = .999969482 \approx$ **99.997%**.

ii. a normalized binary floating point number, and,

Invalid NBFPNs consist of the following possible bit combinations:

Type	Exponent	Fraction	Sign	Probability
Positive Zero	0 (All 0s)	0	0	$1 \div 2^{128}$
Negative Zero	0 (All 0s)	0	1	$1 \div 2^{128}$
Infinity (±)	2^e - 1 (All 1s)	0	*	$2^{113} \div 2^{128}$
NaN	2^e - 1 (All 1s)	non-zero	*	
Denormalized	0 (All 0s)	non-zero	*	$(2^{113} - 2) \div 2^{64}$

Overall, the total probability of any of these four combinations occurring is the sum of $2 \div 2^{128}$, $2^{113} \div 2^{128}$, and $(2^{113} - 2) \div 2^{64}$, or $\mathbf{(2^{113} + 2^{113}) \div 2^{128}}$.

In order to determine the probability of a *valid* NBFPN occurring can be calculated by subtracting the above calculation from one.

Probability that a random 128-bit word represents a normalized binary floating point number:
$1 - \mathbf{(2^{113} + 2^{112} + 2) \div 2^{128}} = .999938965 \asymp$ **99.994%**.