Project Report on:

Advanced Geometric Data Structures Based Independent Study

Student: Ari Bernstein
Advisor: Professor Ivona Bezakova

Table of Contents:

Introduction:

When I began this independent study with my partner, Efe Ozturkoglu, we were unsure of what it would become. We knew it would be data structure-centric and had a theory-based MIT course to follow, but also knew we loved implementation-based learning. What evolved was a learning experience that encapsulated the best of both worlds.

The MIT course in question was called *Advanced Data Structures* and taught by Professor Erik Demaine. It can be found at this address:
https://courses.csail.mit.edu/6.851/spring12/

The lectures covered different sub-fields of data structures, the theory behind them, and their practical realizations; each subfield was covered over between one and four consecutive lectures, each lecture totaling ~80 minutes. This translated into an overwhelming amount of content per lecture, with each one we watched requiring several viewings to understand; each subfield felt like an entire course in of itself.

For this reason, Efe and I decided to limit our focuses to specific subfields. I loved Erik Demaine's two-lecture overview of geometric data structures, specifically how they could be used in conjunction with a concept called *Fractional Cascading*. This became my primary focus for my independent study.

In the context of the approved plan for our independent study, I decided to implement some of the geometric data structure as part of the promised library. The following report will detail the concepts I learned, how they can be applied, and how they can be implemented. For the sections covering concepts I implemented, I will also detail my learning experience from a software engineering perspective. The implementations were in c# and can be viewed in the library GitHub repository at this address:
https://github.com/Oztaco/DataStruct/tree/master/DataStruct/Fractional%20Cascading

I would be remiss without also giving mention to Professor Philipp Kindermann of the University of Würzburg in Germany. While my implementations were inspired primarily by Erik Demaine's lectures, they were at times too theory-heavy and detail-light for me to implement the data structures described within them. To compensate, I used a variety of online sources, the most helpful being Kindermann's lectures on Computational Geometry. His university webpage and YouTube channel can be found at this address:
https://www.informatik.uni-wuerzburg.de/en/algo/staff/kindermann-philipp/
https://www.youtube.com/channel/UCuAzKw_VngkAsQh7ummYq0A

Citations for individual lectures/other learning resources used will be linked at the end of the sections for which they are relevant.

Conceptual Overview of Fractional Cascading in its most basic form and my implementation: Searching for the locations of an element (or its predecessor/successor) that exist in k lists, each of size n.

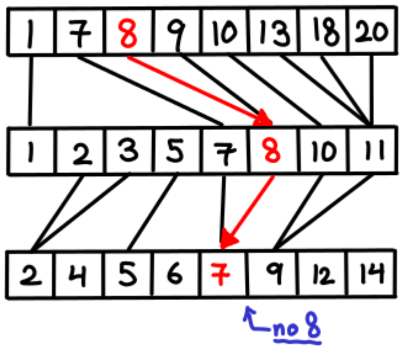
Note some terminology:

1. Dimension – generally the ith list; ranges from 1 to k.
2. Augmented list – a list built by merging a given dimension with half of the elements from the augmented list from the previous dimension. (Also referred to as a *prime* list in implementation.)
3. Promoted (node) - a node from a list that has been merged into an *augmented* list in a higher dimension.
4. Foreign (node) – given a node in an *augmented list,* if said node was promoted from a previous dimension, it is labelled as *foreign*.
5. Local (node) – given a node in an *augmented list,* if said node was initially from the coordNodeList from the current dimension it is labelled as *local*.

Conceptual overview of Fractional Cascading:

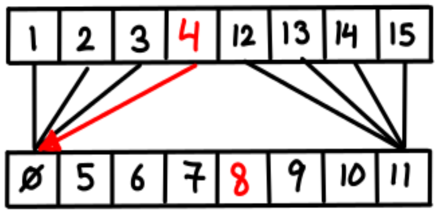
Suppose we have k ordered lists, each approximately size n. We want to find the location of an element x in each list. Trivially, we would accomplish this by performing a binary search on each list, leading to a time complexity of O(k log n). The concept of Fractional Cascading enables us to preprocess our matrix into a new data structure which, when used in conjunction with the initial matrix, enables the same search with a time complexity of O(k + log n) .

It is worth noting that, for the trivial solution, x must exist in each of the k lists. Fractional cascading does not necessitate this as it can be used to instead find x's would-be successor and/or predecessor. However, for simplicity, my implementation guarantees that integer x exists in each list. The rest of the values are randomized integers.



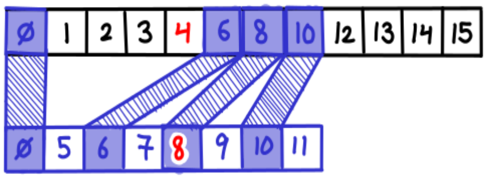
So how is this accomplished? Let's break it down into several different ideas:

1. For each element in each array, we store a pointer to the element with the same value in the next array. If said value does not exist, simply reference its predecessor or successor. Once we have found the first, we can follow the chain of pointers from list 1 to k, finding the location of the elements with the data we are looking for in each of the k arrays.

   This presents an issue when the element in question not only doesn't exist in the next array but is also outside of the range of all the array's values, as illustrated in the diagram to the right. In this case, we would have to redo our entire search for each such array. One often-suggested solution is to simply merge all of the arrays into one. Unfortunately, this would lead to an array of size k*n, which wouldn't help us given too large a k value.

2. Instead of merging all arrays, repeatedly merge every second element from arrays of previous dimensions. Start by merging every other element of list k into list (k-1). Call this list (k-1)’. Then, given i = k-1, for each list starting with (augmented) list i’, build *augmented* list (i - 1)’ by merging (*promoting*) every other element of list i' into list (I - 1). Decrement i and repeat.

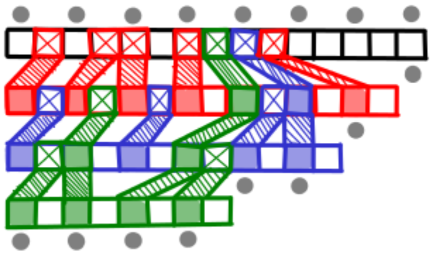


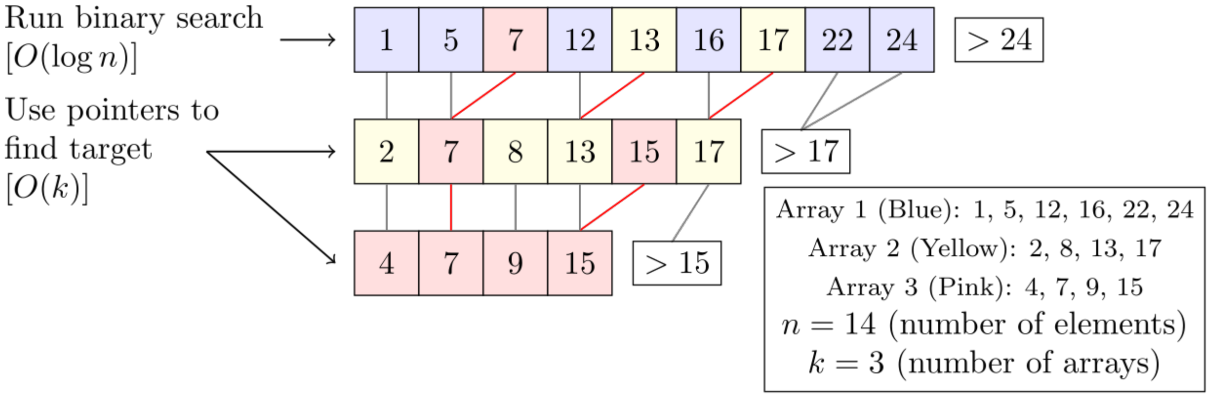
   a. So how large is the list 1’ ? Assuming each list is approximately size n, the recurrence relation:
      T(k) = n + T((k-1)/2) ,

      can be calculated with the summation of the geometric series:

      n + (n/2) + (n/4) + (n/8) + … ≈ 2n .

      The new lists are at most two times the size of their initial counterparts which represents only one extra step in the binary search!

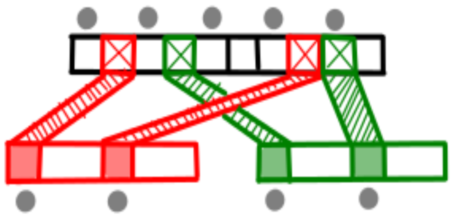




Run binary search $[O(\log n)]$
Use pointers to find target $[O(k)]$
1 5 7 12 13 16 17 22 24
> 24
2 7 8 13 15 17
> 17
4 7 9 15
> 15
Array 1 (Blue): 1, 5, 12, 16, 22, 24
Array 2 (Yellow): 2, 8, 13, 17
Array 3 (Pink): 4, 7, 9, 15
$n = 14$ (number of elements)
$k = 3$ (number of arrays)

3. There is one final detail to consider: when following a pointer through elements in each of the augmented arrays, there is the possibility of landing on a node which is not from the dimension in question. Moreover, there might be many such nodes crammed between the element at the end of the pointer and the one from the dimension we are looking for.

a. Fractional Cascading solves this problem by having nodes point, not to the closest element in the next list, but rather to the previous and next foreign/local nodes; if the node in question has been promoted from a previous list, store pointers to the previous and next local nodes; if the node in question was initially in the current-dimension list, store pointers to the previous and next foreign nodes.



This way, once the location of x is discovered in one list, we can simply use its pointers to (foreign) elements promoted from the previous dimension augmented list as a search range for x in the previous dimension augmented list. Note that this search range is usually only 3 if x exists in the next dimension list! Rinse and repeat!

Fractional Cascading is a versatile concept; for example, when applied using layered range trees, it can accomplish orthogonal range searches with O($\log^{d-2}$ n) such that 3D orthogonal range searching can be accomplished in O(log n) time. Fractional Cascading can also be generalized for almost any graph structure that represents an ordered world. Moreover, we do not have to limit ourselves to cascading ½ of the elements in each augmented list; the only limit is for this fraction is that its denominator must be less than 1/d, d being the in-degree of the node in question.

Discussion of my Fractional Cascading matrix implementation:
My implementation uses the C#. Before getting into the final details, note:

1. When performing my searches, I only populate the search-results dictionary if k is less than a given value, which I default to 10,000. This is for the sake of my computer's memory; the actual values found by the searches are only relevant for testing. Otherwise, for the sake of this learning experience, I only care about the duration of the Fractional cascading solution relative to that of the trivial solution.
2. Regarding querying methods, my randomized CoordNode list generation function sorts the list it returns on the data values of its elements. Their order is maintained when they are converted into FCNodes. The Fractional Cascading transformation (promotion of nodes into higher dimensions) works just like a merge operation such that, when complete, all the nodes in the augmented lists are sorted by their data values. This is why the binary search method calls work in the FractionalCascadingSearch method.
3. While relatively simple to implement on the surface, my real issues began when trying to test the queries against very large data sets (n and k values). Both searches are extremely fast but the construction of the CoordNode data set and its following transformation is slow. This implementation is the result of much iteration to improve performance of both CoordNode generation functionality (see Miscellaneous Functionality section -> Node Generator class -> RandUniqueInts method) and the

Fractional Cascading transformation. More details on this are located in the section following *Code Details*.

Code details:
View code here:
https://github.com/Oztaco/DataStruct/tree/master/DataStruct/Fractional%20Cascading/fractionalCascadingMatrix

For my implementation of Fractional Cascading Matrix, I wrote five classes:

1. CoordNode.cs – a class which contains a data attribute as well as up to three location attributes (x, y, z).

2. FractionalCascadingNode.cs (FCNode) - a class to hold all of the data necessary for the fractional cascading search.
    a. Attributes:
        i. BaseCoordNode – a CoordNode with a integer data and x-location values.
        ii. Index – index in its initial list
        iii. Dimension – integer between 1 and k (inclusive).
        iv. PrevNode – pointer the foreign node with the highest location value less than BaseCoordNode.Location
        v. NextNode – … lowest location value greater than …
        vi. Promoted – Boolean flag indicating whether or not a node has been promoted into an augmented list from a previous dimension. Used for pointer assignment.
        vii. PreviousAugmentedListIndex – if promoted from an augmented list of a lower dimension, this represents the nodes prior location in said list.

3. FractionalCascadingMatrixTransformation.cs
    a. Attributes:
        i. InputCoordMatrix – 2D array of CoordNodes to be transformed.
        ii. NodeMatrixPrimt – 2D array of FCNodes, populated as transformation occurs.
        iii. n – number of nodes per list.
        iv. k – number of lists (dimensions).
    b. Methods:
        i. SetCoordMatrix – use CoordNode random generation functionality to generate matrix of CoordNodes to be a) tested for the trivial solution and b) to be transformed into the Fractional Cascading data structure.
        ii. SetFCTransformationMatrix – Call BuildListPrime to build each row in the Fractional Cascading Matrix.
        iii. BuildListPrime – build the current dimension augmented list using the current-dimension non-augmented list and the previous dimension augmented list, assign pointers between local/foreign nodes.

    iv. SetPointers – helper function located within BuildListPrime that, given a FCNode, assigns prev and next pointers to closest foreign/local nodes.

    v.

4. FractionalCascadingMatrixQuery.cs
    a. Attributes: – identical to 3a.
    b. Methods:
        i. Trivial solution – runs a binary search on each list in InputCoordMatrix
        ii. FractionalCascadeSearch – Perform binary search to find the target node in first dimension. Then iteratively use the previous and/or next pointers to search the (tiny) range of the next dimension given by the prev and next node pointers.
        iii. FindNodeFromPointerRange – Helper function for FractionalCascadingSearch – find and search a range of indices for the FCNode with the target data value at the target dimension.

5. FCMatrixDemo.cs – for demonstration and testing purposes.
    a. (No attributes.)
    b. Methods:
        i. Demo – construct CoordNodeMatrix and CoordNodeMatrixPrime with given search value. Demonstrate differences in durations between trivial solution and Fractional Cascading solution.
        ii. CSV_Loop – same as demo but loop over a range of n and k values and write results to a CSV file.

Overview of transformation methods:

Method overview – void **SetCoordMatrix(**int **insertData):**
Parameter:

insertData – For both the trivial solution and my implementation of the fractional cascading solution, the node whose value for which we are searching must exist somewhere in each of the k lists. InsertData specifies such a location which will be placed in every (otherwise randomly-generated) CoordNode list.

Process:

1. Iterate from 0 to k. For each iteration call NodeGenerator.GetCoordNodeList() with insertData as a parameter. Set the return to coordNodeMatrix[i] .
    a. Note that the GetCoordNodeList returns a randomized list of CoordNodes which, by default, are sorted on their data (not location) values.

Method overview - void **SetFCTransformationMatrix(**int **unitFracDen):**
Parameter:

unitFracDen - denominator of the unit fraction indicating the size of the subset of promoted list (d-1)' into list d'. This value is traditionally 2 such that ½ of the elements in list (d-1)’ are promoted into list d’.

Process:

1. Instantiate global variable NodeMatrixPrime as an empty matrix of FCNodes of size k by n.
2. Iterate from 0 to k using index pointer i. At each step, instantiate an empty list of FCNodes of size n. Then iterate from 0 to n using index pointer j. At each sub-step, convert the CoordNode at InputCoordMatrix[i, j] to an FCNode at NodeMatrixPrime[i, j].
3. Perform the actual transformation by iterating through each list in NodeMatrixPrime in reverse order, starting with the second to last as the final augmented list is equivalent to its initial list.
    a. for (int i = k-2; i ≥ 0; i--):

```
            NodeMatrixPrime[i] =
                    BuildListPrime(NodeMatrixPrime[i], NodeMatrixPrime[i + 1], … )
```

Method overview –
FCNode[] **BuildListPrime(**FCNode[] **FCNodeList1**, FCNode[] **FCNodeList2**, int **unitFracDen**)**:**
Parameters:

1. FCNodeList1 – NodeMatrixPrime[i]
2. FCNodeList2 – NodeMatrixPrime[i + 1]
3. unitFracDen – See SetFCTransformationMatrix method description.

Return:

FCNodeList1’ – a new list of FCNodes with all of the values in FCNodeList1 and half (1 / unitFracDen) of the values in FCNodeList2. Each foreign node has pointers to its preceding and succeeding local nodes; each local node has pointers to its preceding and succeeding foreign nodes.

Process:

1. Instantiate null FCNode variables lastPromotedNode, lastNotPromotedNode.
2. Instantiate SetPointers method – see next method overview.
3. Instantiate a list to contain nodes to be promoted from FCNodeList2 into FCNodeMatrix1’.
    a. numPromotedNodes <- Math.Ceiling(FCNodeList2.Length / unitFracDen))

4. Instantiate list of FCNodes FCNodeListPrime, the new augmented list being constructed:
    a. FCNodeListPrime <- new FCNode[n + numPromotedNodes]

5. Instantiate list of FCNodes nodesToPromote, which will contain all of the nodes from FCNodeList2 to be promoted, of size numPromotedNodes .
6. Instantiate counter integer variables to 0:
    a. c - index counter for nodesToPromote.

   b. d - index counter for FCNodeList1.
   c. j – index counter for FCNodeListPrime.

7. Populate nodesToPromote – iterate through FCNodeList2 with increments of unitFracDen with index pointer i. At each iteration, copy the node at the ith index of FCNodeList2, mark it as promoted, and add it to nodesToPromote. Note that it is essential for nodes to be copied here because this is the only state at which we can easily both store their locations in their previous augmented lists and flag them as promoted.
   a.
```
for (i = 0; i < FCNodeList2.Length; i += unitFracDen):
        nodesToPromote[c++] <- FCNodeList2[i].Copy(setPromoted:true,
                                                    prevAugmentedIndex:i)
```

8. Perform Fractional Cascading transformation:
   a. Reset index counter c to 0.
   b. Merge elements from nodesToPromote and FCNodeList1 into FCNodeListPrime, assign pointers.
      i.
```
while(c < nodesToPromote.Length && d < FCNodeList1.Length):
    if (FCNodeList1[d].Location < nodesToPromote[c].Location):
        FCNodeListPrime[j] <- FCNodeList1[d++].Copy()
    else:
        FCNodeListPrime[j] <- nodesToPromote[c++]
    SetPointers(FCNodeListPrime[j++])
```

9. Add leftover values to augmented list – identical to the last step of the merging process in merge sort.
   a.
```
while(c < nodesToPromote.Length):
        FCNodeListPrime[j] <- nodesToPromote[c++]
        SetPointers(FCNodeListPrime[j++])
```
   b.
```
while(d < n):
        FCNodeListPrime[j] <- FCNodeList1[d++].Copy()
        SetPointers(FCNodeListPrime[j++])
```

10. Return FCNodeListPrime!

Method overview – void **SetPointers(**FCNode **currentNode):**
Parameter: currentNode – the node to which pointers will be assigned.

Process:
1. Note that, being located inside of BuildListPrime, SetPointers has access to (initially null) FCNodes lastPromotedNode and lastNotPromotedNode.
2. Assign prev and next pointers to closest promoted / non-promoted nodes
   a. Regarding nested if statements below, we must ensure that the successor of the currenNode's local predecessor does not already have a local / foreign successor.

```
    b. if (currentNode.IsPromoted()):
            lastPromotedNode <- currentNode
            currentNode.PrevPointer <- lastNotPromotedNode
            if (lastNotPromotedNode ≠ null &&
                    lastNotPromotedNode.NextPointer == null):
                        lastNotPromotedNode.NextPointer <- currentNode
    c. else:
            lastNotPromotedNode <- currentNode
            currentNode.PrevPointer <- lastPromotedNode
            if (lastPromotedNode ≠ null &&
                    lastPromotedNode.NextPointer == null):
                        lastPromotedNode.NextPointer <- currentNode
3. Mark currentNode as prime.
```

Overview of query methods:

Method overview – Dictionary<int, int> **TrivialSolution(**int **location):**
Parameter:
location – the location value of the node for which we are searching in each dimension.

Return: A dictionary with key = dimension, pair = location of data in dimension.

Process:
1. Instantiate dictionary locationsOfData.
2. Iterate from 0 to k with index pointer i. At each iteration, perform a binary search for location on the ith element of NodeMatrixPrime.
3. Add element's dimension and location to dictionary.
   a. locationsOfData[i] = dataNode.location
4. Return locationsOfData .

Method overview –
bool **TargetNodeCheck(**FCNode **node,** int **targetData,** int **targetDimension):**
Parameters:
1. node – the node which we are checking.
2. targetData – the location of the node (in targetDimension) that we are targeting.
3. targetDimension – the dimension of the location value we are targeting.

Process:
Return true if node ≠ null, node.GetData == targetData, and node.Dimension == targetDimension.

Method overview –
FCNode **FindNodeFromPointerRange(**FCNode **dataNode,** int **targetDimension):**

Parameters:

1. dataNode - FCNode from augmented list at targetDimension - 1 . DataNode contains pointers to FCNodes from the augmented list of the next dimension (TargetDimension). The search range is between the locations of DataNode.prev and DataNode.next in the augmented list in TargetDimension.
2. targetDimension – the dimension of the location of dataNode for which we are searching.

Return: dataNode from location in *previous-dimension* augmented list.

Process:

1. Instantiate variables using attributes of dataNode:
    a. targetData <- dataNode.GetData .
    b. prevNode <- dataNode.PrevPointer .
    c. nextNode <- dataNode.NextPointer .
2. Instantiate more variables:
    a. Integer targetDimIndex <- targetDimension – 1 .
    b. Empty integers lowRange, highRange .
        i. lowRange <- 0 if prevNode is null, else: prevNode.prevAugmentedIndex
        ii. highRange <-
            length of augmented list at target dimension if nextNode is null,
            else: prevNode.prevAugmentedIndex
3. Search the NodeMatrixPrime at targetDimension from indexes lowRange to highRange.
    a. Instantiate a Boolean variable found to false. This is so that we can check whether the FCNode with location in currentDimension of targetData was found.
    b. Instantiate FCNode dataNode
    c. Iterate through FCNodes in NodeMatrixPrime at index targetDimIndex
        i. Set dataNode <- current FCNode.
        ii. Call TargetNodeCheck on dataNode, targetData, and targetDimension.
        iii. If TargetNodeCheck evaluates to true, set found = to true and break.
4. Check if found is false. If so, throw an error stating that node was not found in range.
5. Return dataNode.

Method overview - Dictionary<int, int> **FractionalCascadingSearch(**int **data):**
Parameter: data – the data value whose location we are searching for in each dimension.

Return: A dictionary with key = dimension, pair = location of data in dimension.

Process:

1. Instantiate variables:
    a. locationsOfData - empty dictionary of types <int, int> .
    b. dataNode – null FCNode which will represent location of node with target data in each dimension.

```
2. Perform a binary search to find dataNode in first dimension:
    a. dataNode <- BinarySearch(NodeMatrixPrime[0], data)
        i. Note that, as all augmented lists (except for that of the highest
           dimension) contain nodes from higher dimensions, it is possible for this
           search to returns a node with the correct data value but located in a
           higher dimension.
3. Check if dataNode is in the correct dimension. If not, check its prevNode and nextNode
   pointers (both of which are guaranteed to be from the current dimension).
    a. Run TargetNodeCheck using parameters dataNode, data, and 1. If it returns true,
       then skip to step 4.
    b. Else, if TargetNodeCheck returns false:
        i. if (TargetNodeCheck(PrevNode, data, 1)): dataNode <- prevNode
        ii. else if (TargetNodeCheck(PrevNode, data, 1)): dataNode <- nextNode
        iii. else throw error
4. Add location in first dimension to locationsOfData
    a. locationsOfData[1] = dataNode.location
5. Walk through promoted node pointers, starting with the list 2' until list k' .
    a. Iterate from 1 to k with index pointer i .
        i. Increment currDim.
        ii. dataNode <- FindNodeFromPointerRange(dataNode, currentDim).
        iii. Add location in first dimension to locationsOfData
                - locationsOfData[currDim] = dataNode.location
6. return locationsOfData .
```

Further review of my implementation-based learning experience:

- There are relatively rare cases in which the trivial solution (k log(n) searches –> O(log n)) is faster than the Fractional Cascading solution (log(n) search + k steps –> O(log n)). This was concerning to me until I discussed it with Professor Bezakova. She explained that, as k is a constant that is generally much smaller than n, both solutions technically have the same time complexity. From here, she explained the invisible constant, which relates the number of processes that accompany each "step" in the execution. Given that the Fractional Cascading search does more work with each "step" than the trivial solution, it made sense that there would be some scenarios in which k binary searches could finish running in a shorter period of time than a single binary search followed by k steps. This also helped me to better understand the concept of classifications of time complexities which, while awesome, really tells me that I should have paid better attention in Professor Bezakova's wonderful Analysis of Algorithms course.

- As previously noted, the code related to the Fractional Cascading transformation is the result of a great deal of iteration; I had run into performance and memory issues when generating and transforming the data set for the queries. Most of these changes came down to performing operations in place to save memory. For example, initially, the FractionalCascadingTransformation constructor generated three matrices instead of two; revising this operation decreased the memory usage by almost 25% .

Another example: initially, the functionality to assign pointers did so in an extremely inefficient way; only after all nodes had been promoted (but pointers not-yet assigned), it would traverse the newly-generated augmented list twice – once for successors, once for predecessors. Revising this approach to the one under BuildListPrime Method Overview -> Process -> 8b & 9 dramatically reduced the duration of the Fractional Cascading transformation, though I never measured by exactly how much.

- I also ran into a small issue in attempting to save memory by using actual references to nodes in previous dimensions instead of copies. This just became too difficult to debug/manage as any promoted node could exist in any number of higher dimensions; currently my implementation duplicates the nodes' data, current dimension, and location values with null predecessor and successor pointers. If I had more time, I would go back and attempt a solution that could handle both predecessor and successor pointers to nodes in the current-dimension augmented list and references to FCNodes in previous dimensions.

Results:
Tested with a nested loop with k ranging from 250 to 290 and n from 250 to 450 with increments for both of 10. For each test, a new data set was randomly generated with locations between 0 and n*k . x was also a random value between 0 and n*k, which was inserted into each list at a random location if not already present.

As you can see, the durations of the Fractional cascading solution are generally much faster than those of the trivial solution. As mentioned above, there are relatively rare exceptions as can be seen in rows 2 and 8. As the values for n and k increase, not only do such exceptions become less common but the ratios between the trivial solution duration and the fractional cascading solution shrink as well.

	n	**k**	**x**	**FC solution duration (ms)**	**trivial solution duration (ms)**	**ratio**
1	200000	250	387	5	11	0.454545455
2	250000	250	164174	75	34	2.205882353
3	300000	250	294142	124	2382	0.052057095
4	350000	250	219791	220	2220	0.099099099
5	400000	250	33766	529	3144	0.168256997
6	450000	250	228017	166	2280	0.072807018
7	200000	260	64364	7	218	0.032110092
8	250000	260	51579	86	85	1.011764706
9	300000	260	177865	234	2730	0.085714286
10	350000	260	314446	202	2477	0.081550262
11	400000	260	216440	97	2877	0.033715676
12	450000	260	164778	273	3423	0.079754601

13	200000	270	142839	164	2006	0.081754736
14	250000	270	112039	306	1282	0.238689548
15	300000	270	88702	296	3076	0.096228869
16	350000	270	121252	643	3015	0.213266998
17	400000	270	120261	244	682	0.357771261
18	450000	270	363074	252	3306	0.076225045
19	200000	280	84372	157	1952	0.080430328
20	250000	280	146668	218	1255	0.173705179
21	300000	280	117774	564	2438	0.231337162
22	350000	280	82900	249	621	0.400966184
23	400000	280	113179	610	2209	0.276143051
24	450000	280	229974	273	3251	0.083974162
25	200000	290	28999	182	1962	0.092762487
26	250000	290	201806	5	529	0.009451796
27	300000	290	170023	254	1315	0.193155894
28	350000	290	199700	759	315	2.40952381
29	400000	290	294118	636	3886	0.163664436
30	450000	290	206020	486	2806	0.173200285

References:

1. https://courses.csail.mit.edu/6.851/spring12/lectures/L03.html
2. http://blog.ezyang.com/2012/03/you-could-have-invented-fractional-cascading/
    a. Most of the images
3. https://en.wikipedia.org/wiki/Fractional_cascading
    a. https://upload.wikimedia.org/wikipedia/commons/thumb/3/39/Fractional_cascading.svg/1280px-Fractional_cascading.svg.png

Red-Black Trees

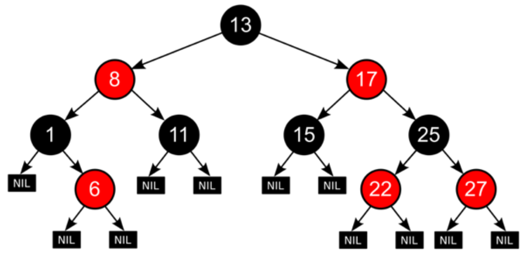
Conceptual overview of Red-Black Trees:
Red-Black Trees are one implementation of a self-balancing binary search tree. In Red-Black trees, each node is colored either red or black. Leaf nodes are always null. If the following color-related conditions are met, the tree will be balanced such that search, insertion, and deletion (only partial implemented here) can be accomplished in O(log n) time in all cases.

Color conditions:

1. The Root of the tree is always black.
2. Every leaf is black.
3. There can never be two successive red nodes.
   ie. no red node can have a red parent or red child.
4. Every path from a given node to any leaf in its subtree will always contain the same number of black nodes.

Meeting these conditions will not result in a perfectly balanced tree. They do however guarantee that every path from the root to any leaf is at most twice the height, or 2 * Log2(n+1). This translates to only one additional step during search than a perfectly balanced tree. Also note that this slight imbalance comes with the advantage of fewer rotations per operation relative to other balanced binary search tree implementations, as well as high likelihood of not needing to rebalance after any given operation. This in turn results in fast insertion and deletion relative to other self-balancing binary search trees.

Space complexity: O(n)
Search time complexity: O(log n)
Insertion time complexity: O(log n)
Deletion time complexity: O(log n)



Example from using randomly generated list from my implementation:

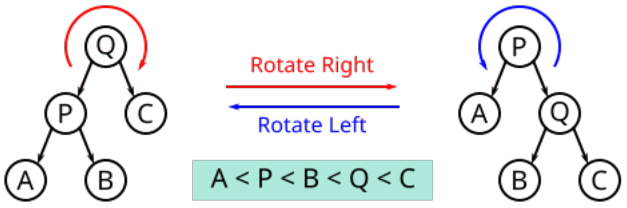
```
list = [22, 561, 7, 65, 100, 200, 300, 27, 3, 12, 29, 67, 6, 5, 123, 4, 2];
for each (int i in list): Red-BlackTree.Insert(i);
Red-BlackTree.GetRoot().Visualize();
29 (B)
├----- 7 (B)
│      ├----- 5 (R)
│      │      ├----- 3 (B)
│      │      │      ├----- 2 (R)
│      │      │      └----- 4 (R)
│      │      └----- 6 (B)
│      └----- 22 (R)
│             ├----- 12 (B)
│             └----- 27 (B)
└----- 100 (B)
       ├----- 65 (B)
       │      └----- 67 (R)
       └----- 300 (R)
              ├----- 200 (B)
              │      └----- 123 (R)
              └----- 561 (B)
```

General Operations

The following processes are used by both insertion and deletion to ensure that all the color conditions are met.

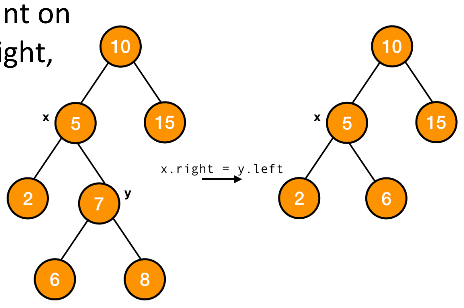
1. **Rotation:**
   Given a non-leaf node, swap its position with that of one of its children; if swapping with the left child, use left-rotation; if swapping with the right child, use right-rotation. This operation changes the structure of the tree without violating the rules related to the ordering of its nodes. In other words, an in-order tree traversal will return the same order after any number of tree rotations.

   In Red-Black Trees, rotation is primarily used for rebalancing the tree when its color conditions have been violated after some other operation, such as insertion or deletion.



   After a rotation, the size of the subtree of the side on which we rotate is increased by one; the size of the subtree on the other is decreased by one. This is part of how it is utilized by the methods which rebalance the Red-Black tree post insertion or deletion.

   a. **Left Rotation:**
   Given a non-lead node with at least one descendant on the left side and at least two generations on the right, label the non-leaf node as x:



   (Set the left child of y as the right child of x.)
   1. Assign y <- x.Right
   2. x.Right <- y.Left
   3. if (y ≠ null): y.Left.Parent = x

   (Adjust neighbors such that y is in the location of x.)

   4. y.Parent = x.Parent

   5. if (x.Parent == NULL): RBTree.Root <- y   (x is the root)
   6. else if x == x.parent.left: x.Parent.Left <- y   (x is a left child)
   7. else: x.Parent.Right <-y   (x is right child)

   (Now that y is in the location of x, we need only set x as the right child of y.)

   8. x.left <- x
   9. x.parent <- y

   b. **Right Rotation**: Symmetric to left rotation, just replace left with right.

2. **Transplanting:**
   Simply replacing one node z with another node y:

```
1. p = z.Parent
2. if (p == Leaf): RBTree.Root <- y   (x is the root)
3. else if (x == p.Left): p.Left <-y   (x is a left child)
4. else: p.Right <- y   (x is right child)
5. y.Parent <- p
```

   Note: In step 2, we check for leaf (empty node colored black) instead of null. This means that we don't have to bother with a null check, unlike a regular BST.

3. **Minimum:**
   Given a subtree with root z, traverse left children until the node with the smallest value is found. Remember that, in the case of a Red-Black Tree, leaves are nodes colored black with no value:

```
1.  while (z.Left ≠ leaf):
2.      z <- z.Left
3. return z
```

Insertion:
Insertion happens in two steps. First, insertion of the new node just like one would for a regular binary tree. The new node's new location maintains the tree's ordering but may violate the color conditions. New nodes are always instantiated as red. Second, use some tricks and helper operations to adjust the tree such that it does not violate the color conditions.

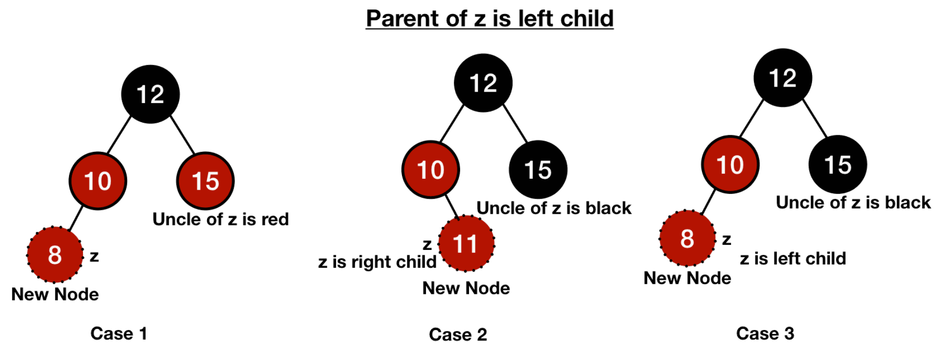
1. **Insertion function:**
   Given a new node z and a Red-Black tree T with attributes Root and Leaf (empty node colored BLACK):

```
1. temp  <- RBTree.Root
2. y <- T.Leaf
3. if (root == null):     (Check for empty tree)
       a. x.Color <- red
       b. Root <- z

   (Traverse until y is at the lowest level but also not a leaf in a location where x would
   not violate the tree's ordering)
4. while temp ≠ RBTree.Leaf:
       a. y <- temp
       b. if (x < temp): temp <- temp.Left
       c. else: temp <- temp.Right

   (Set x as the child of y with children as leaves)
5. x.Parent <- y
```

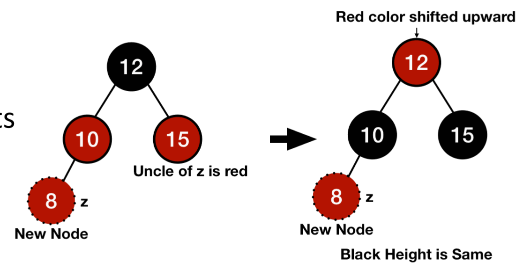
```
6.  if (y == null):  RBTree.Root <- z  (z is the root)
7.  else if (x < y): y.Left <- z
8.  else: y.Right <- z
9.  z.Left <- RBTree.Leaf
10. z.Right <- RBTree.Leaf
11. z.Color <- red
12. RebalancePostInsertion(z)
```

2. **Function to rebalance the tree post insertion:**
At this point, new node z is at the bottom of the tree and colored red. We need to check that it doesn't violate color conditions three (no two successive red nodes) or 4 (Same number of black nodes on every path from the root to a leaf).

We begin by checking color condition three, ie if the parent of z is red. We know that, in this case, the grandparent of z will be black. From here there are six possible cases, three if z's parent is a left child, three if z's parent is a right child. The following checks are performed inside of a while loop whose stopping condition is that z's parent is black. Note that, if this process is performed when the parent of z is not red, it can violate color condition four.



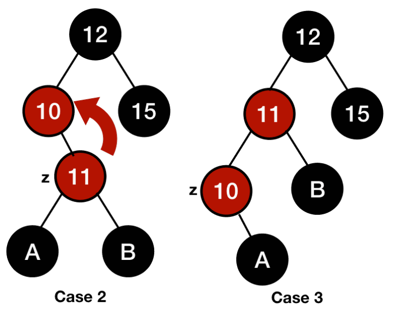
Case 1: The pibling of z is red.
In this case, "shift" the red color up the tree until the parent of the newly-red node is black and neither of its children are red. If we hit the root, color it black. Only handle the current node, its parent, pibling, and grandparent; let the while loop do the rest.



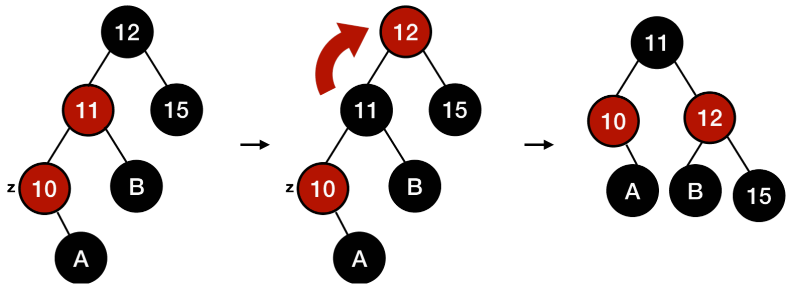
Case 2: The pibling of z is black and z is a right child.

Case 3: The pibling of z is black and z is a left child.

Case 2 can be converted into Case 3 with a left rotation on z's parent. Note that, as both z and its parent are red, the black-height of the tree will not be altered such that color condition four is not violated. **If we encounter case 2, left-rotate z's parent and handle case 3.**



Case 3: In this case, set the color of z's parent to black and z's grandparent to red. Then perform a right-rotation on z's grandparent:



Given that cases 2 and 3 guarantee that z's pibling is black, we can be certain that coloring z's parent black will make both children of z's grandparent black. Therefore, when we perform a right rotation on z's (now red) grandparent, we can be certain that color conditions three and four will be met.

The other three cases, which occur when z's parent is a right child, are symmetric; just swap left with right.

Notes on rebalancing:

1. Fixing cases 2 or three guarantees the red-black color conditions are met such that the while loop will hit its stopping condition. Given case 1, "shifting" the red node up a level can induce cases 2 or 3 (or the symmetric cases). In the case where cases 2 or 3 are never met, we end up with a red Root. To account for this, we set the root to black immediately after the while-loop finishes running.

2. Once can conceptualize this process by thinking solely in terms of ancestors; the tree is rebalanced from the bottom up with consideration only given to the current node, its parent, grandparent, and pibling. We progressively move up the tree, leaving an ever more balanced subtree in our wake!

3. Only case 1 will cause the rebalancing function to iterate. As this iteration is on the grandparent of z, the maximum number of steps it will take is (log n) / 2 , which is how the insertion process retains its O(log n) running time.

Process in pseudocode:

```
1.  while z.Parent.IsRed:
2.      if (z.Parent == z.GrandParent.Left):    (z.Parent is a left child)
3.              y <- z.GrandParent.Right    (pibling of z)
4.              if (y.IsRed):   (case 1)
5.                      z.Parent.Color <-black
6.                      y.Color <- black
7.                      z.GrandParent.Color <-red
8.                      z <= z.GrandParent

9.              else: (cases 2 or 3)
10.                     if z == z.Parent.Right   (z is a left child – case 2)
11.                             z = z.Parent
12.                             RotateLeft(z)  (case 2 is now case 3)

                        (case 3)
13.                     z.Parent.Color <- black
14.                     z.GrandParent.Color <- red
15.                     RotateRight(z.GrandParent)

16.     else:   (z's parent is a right child, code will be symmetric, just swap left and right)
17.             …
18. RBTree.Root.Color <- black  (case where case 1 is hit repeatedly until root is red)
```

Deletion:

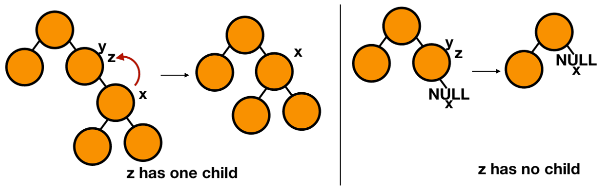
As with insertion, deletion works over two steps. First, deletion similar to that of a regular binary search tree such that the tree's ordering property is maintained but its color conditions may be violated, with the additional step of storing the color of the node to be deleted for imbalance-checking. Second, use some tricks and helper operations to adjust the tree such that it does not violate the color conditions.

1. **Deletion function:**

   Let's begin by considering the cases for deletion in a regular binary search tree. For each case, let's discuss their implications on the balance of a Red-Black tree. Given node-to-delete z:

   (For cases a and b, let *color* denote the color of z)

   a. z has no children
      i. in this case, simply delete z
   b. z has one child
      i. in this case, simply transplant (replace) z with its sole child.



(For cases c1 and c2, let y denote the in-order successor of z, let *color* denote the color of y, let x denote the right child of y; x will fill y's initial position).

c. z has two children and:

1. y is a direct successor (ie. z's right child):
   (In case x is a leaf such that its parent can be any arbitrary node.)
   i. set x.Parent <- y

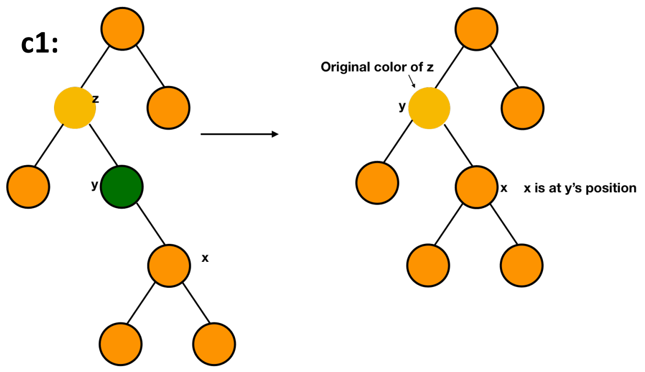
2. y is not a direct successor (ie. a left child of some node in z's right subtree)
   (After these steps, case c2 is converted to case c1.)
   i. Transplant y with its right child
   ii. y.Right <- z.Right
   iii. y.Right.Parent <- y

(In either case c1 or c2, after ensuring y the right child of z.)

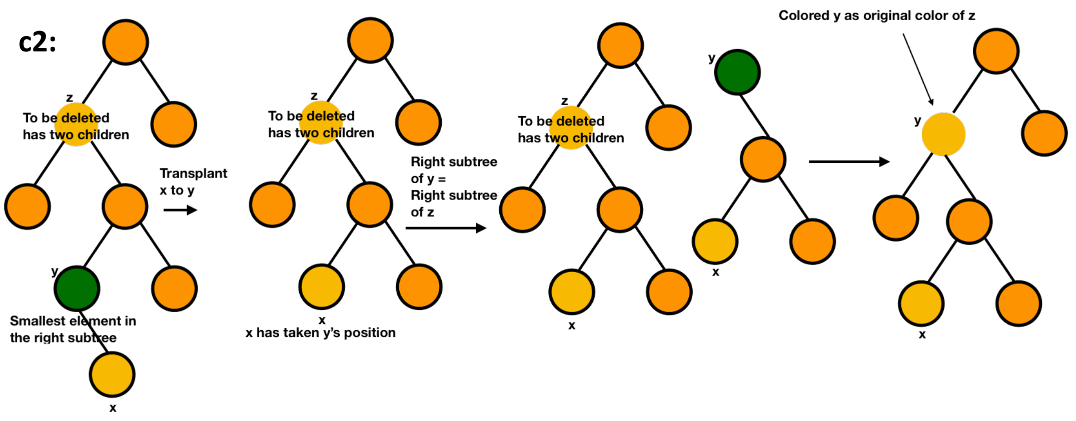
i. Transplant z with y.
ii. y.Left.Parent <- y
iii. y.Color <- z.Color

d. Not a case – just the final step regardless of cases:
   i. if (color == black): RebalancePostDeletion(x)

c1:

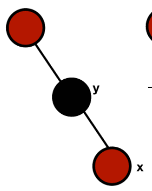


c2:



Possibilities for imbalance:

- For cases a and b, violation of color conditions is only possible if z is black.
- For cases c and d, node z is replaced by node y and node y is replaced by node x. As y is recolored to z's initial color, color condition violations can occur because of x taking y's position. This is the reason we store the initial color of y.
- So, ((given case a OR b) AND z is black) OR ((given case c1 OR c2) AND y was initially black)), the following violations are possible:
    - Violation of color condition 4 such that black-height of the path to any given leaf, which would either be z or one of any of its descendants, is shorter than the other paths.
    - Violation of color condition 3 such that the parent of z is red and its sole child which is replacing it is red (Only possible for case b).



- **Note:** given case c1 OR c2, if y was initially red, none of the color conditions would have been violated.
    - Removing y will not affect the black-height of the tree.
    - It would not have been root (color condition 1).
    - Cannot make any red colored nodes consecutive.

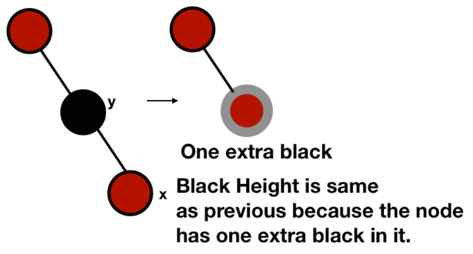
Process in pseudocode:

```
1.  originalColor <- z.Color
2.  y <- z

3.  if (z.Left == Leaf):  (case a or b)
4.      x <- z.Right
5.      Transplant(z, z.Right)

6.  else if (z.Right == Leaf):  (case b – left child only)
7.      x <- z.Left
8.      Transplant(z, z.Left

9.  else:  (case c1 or c2)
10.     y <- Minimum(c.Right)
11.     originalColor <- y.Color
12.     x = y.Right

13.     if (y.Parent == z):  (case c1)
14.             x.Parent <- y

15.     else:  (case c2)
16.             Transplant(y, y.Right)
17.             y.Right <- z.Right
18.             y.Right.Parent <- y

19.     Transplant(z, y)
```

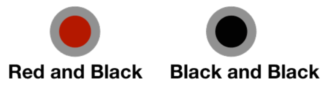
```
20.       y.Left.Parent <- y

21.       y.Color <- z.Color

22. if (originalColor == black):
23.       RebalancePostDeletion(x)
```

2. **Function for Rebalancing Post Deletion:**
At this point, z has been replaced by y, its in-order successor, ie. the smallest node in its right subtree. y has been set to the same color as z, such that it and its ancestors do not violate any color conditions. y's right child is x, which replaced y's initial location in the tree. x is where we may encounter color-condition violations.

We begin by addressing the possibility of violation of color-condition 4, which states that every path from the root of the tree to any leaf has an equal number of black nodes. In this case, any subtree which contains y may have a path with one-less black node than the others. To address this, we use something of a Jedi mind trick, ie. a weird hand-wave followed by the statement, "this is not the black node you are looking for." But instead of convincing some droids, we convince the Red Black tree in question.


y
x
One extra black
Black Height is same as previous because the node has one extra black in it.

In practice, we pretend that node x (now occupying y's initial position) has an extra black node inside of it, such that, if x was black, it is now "double black" and if it was red, it is now "red and black".



This fixes the height issue but violates the base condition of the entire data structure, which states that nodes must be colored **either** red or black. We also must account for color condition 1 (that the root of the tree must be black) and color condition 3 (that there cannot be two successive red nodes).

There are two possible cases that we can easily solve right off the bat:

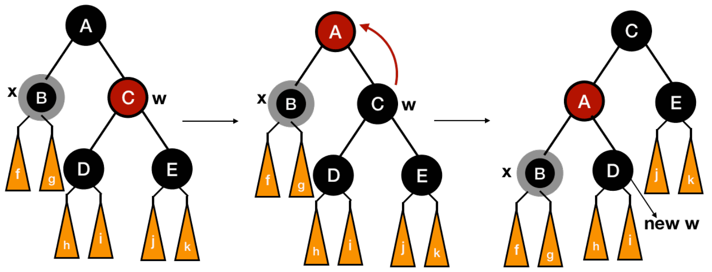
1. If x is "red and black", we simply color it black, fixing both the base condition and color condition 3.
2. If x is the root of the tree, simply turn it from "double black" to black.

After these conditions are checked, we can be sure that color conditions 1 and 3 are met. This leaves only the case of x being "double black" or "red and black". This results in four possible cases. **Let node w be x's sibling:**

1. w is red.
2. w is black and both of its children are black.
3. w is black and its right child is black and its left child is red.
4. w is black and its right child is red.

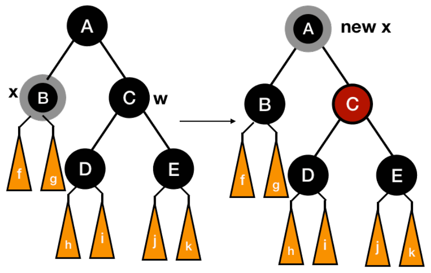
Note: the following cases are in the context of x being a left child and w being a right child. The process for the opposite case is symmetric, just swap left with right!

Given case 1 – w is red:
We swap color of w with that of its parent and then left rotate the parent of x. In this way, we will enter cases 2, 3 or 4.



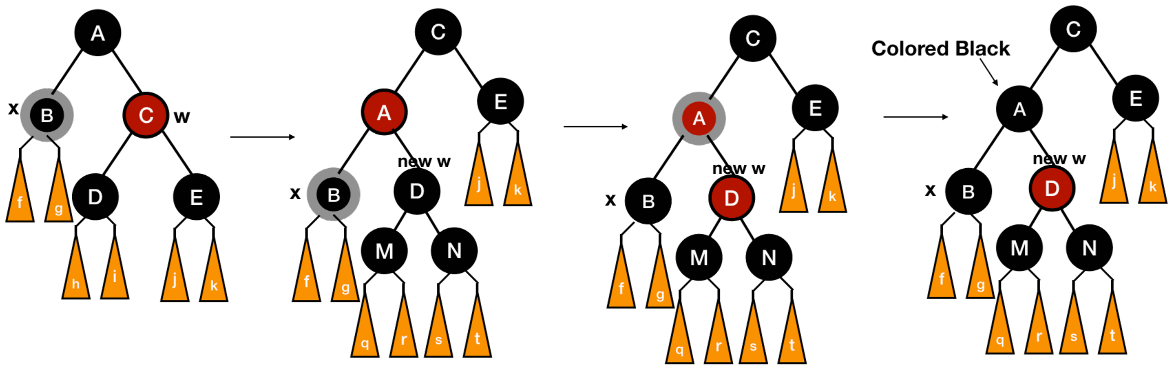
Case 1

Given case 2 – w and both of its children are black:
We will "remove a black" from both x and w. If x was "double black", it will become black; y will be red. To compensate for this, we add an extra black to the parent of x, and mark it as new x. We then repeat the process on the new x (ie. the parent of former x).



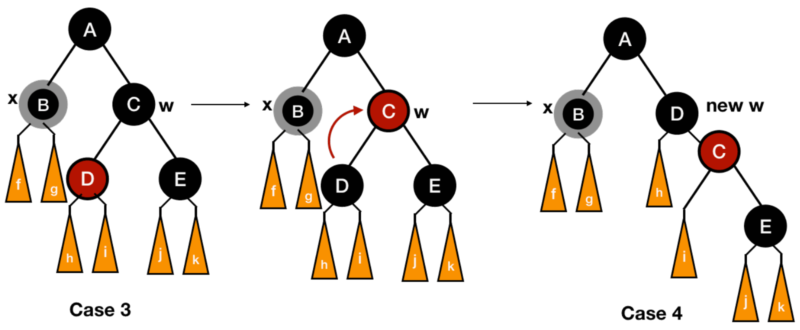
Case 2

Note: in performing this process, we will never induce a **new** color condition violation. **This is also the only case in which we iterate; all others will terminate the loop.** If we entered case 2 via case 1, the parent of x (new x) will have been red, such that it is now "red and black". We fix this by simply changing its color to black.

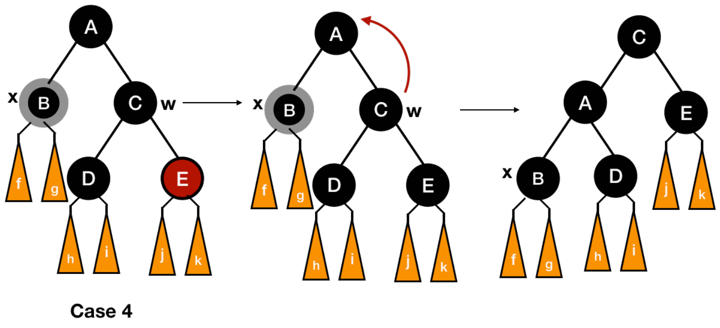


Case 1 Case 2

Given case 3 – w is black and its right child is black and its left child is red:
Simply transform the subtree into case by swapping the color of w with that of its left child. Them perform a right rotation on w.



Given case 4 - w is black and its right child is red:
Change the color of w to that of its parent, color its parent black, and color its right child black. Then perform a right rotation on the parent of w.



Note: Cases 3 and 4 will both result in a tree which meets all the color conditions. Case 1 can either result in such a tree as well or in case 2. As mentioned, case 2 may not result in a tree which meets all of the color conditions and as such is the only condition under which we will iterate. As this iteration is performed on the parent of x, it is how the deletion process retains its O(log n) running time.

Process in pseudocode:

```
1.   while x ≠ RBTree.Root and x.IsBlack:
          (If x is a left child st w is a right child)
2.        if (x == x.Parent.Left):
3.             w <- x.Parent.Right

4.             if (w.IsRed):   (Case 1)
5.                  w.Color <- black
6.                  x.Parent.Color <- red
7.                  RotateLeft(x.Parent)
8.                  w <- x.Parent.Right

               (not an else if as case 1 turns into case 2)
```

```
9.          if (w.Left.IsBlack && w.Right.IsBlack):  (case 2)
10.             w.Color <- red
11.             x <- x.Parent  (so that the while-loop iterates correctly)

12.         else:  (cases 3 or 4)
13.             if (w.Right.IsBlack):  (case 3)
14.                 w.Left.Color <- black
15.                 w.Color <- red
16.                 RotateRight(w)
17.                 w <- x.Parent.Right

                (case 4)
18.             w.Color <- x.Parent.Color
19.             x.Parent.Color <- black
20.             w.Right.Color  <-black
21.             RotateLeft(x.Parrent)

        (By this case we have reached the root of the tree and must stop the iteration.)
22.             x <- Root

        (If x is a right child st w is a left child)
23.     else:
24.         … code will be symmetric, just swap left and right

25. x.color = black
```

Code details:
To implement my Red-Black Tree, I used C# and wrote three classes:

1. RedBlackTree.cs – has all functionality related to insertion and deletion.
    a. Attributes:
        i. Root – RedBlackTreeNode representing the root of the entire tree
        ii. Leaf – RedBlackTreeNode with no information except that it is colored black
    b. Methods:
        i. RotateRight – given a RedBlackTreeNode, put its root in the location of its *right* child in such a way that the tree without violating the ordering of the tree's elements.
        ii. RotateLeft – given a RedBlackTreeNode, put its root in the location of its *left* child in such a way that the tree without violating the ordering of the tree's elements.Transplant – given RedBlackTreeNode x and RedBlackTreeNode y, replace y with x.

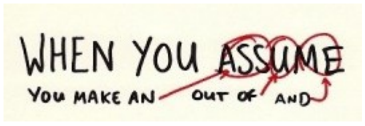
        iii. Minimum – given a RedBlackTreeNode x, return the RedBlackTreeNode with the lowest key in x's subtree.
        iv. Delete – given RedBlackTreeNode x, remove x from the RedBlack tree in such a way that maintains the tree's ordering while not necessarily maintaining all the color conditions. Then call RebalancePostDeletion.
        v. Insert – given RedBlackTreeNode node x, place x at the bottom of the tree in such a way that maintains the tree's ordering while not necessarily maintaining all the color conditions. Then call RebalancePostInsertion.
        vi. RebalancePostDeletion – given RedBlackTreeNode x, which is in the location of a previously-removed node, check for and address color conditions 3 and 4.
        vii. RebalancePostInsertion – given RedBlackTreeNode x, which has just been inserted, check for and address color conditions 3 and 4.

2. RedBlackTreeNode.cs – has all the functionality a binary tree node could ask for :)
    a. Attributes:
        i. SortAttribute – integer with which node is placed in tree.
        ii. Red – if true, node is red. Else, node is black.
        iii. EmptyNode - if true, node is a leaf.
        iv. LeftChild – RedBlackTreeNode for which SortAttribute of self and all children are less than SortAttribute.
        v. RightChild –RedBlackTreeNode for which SortAttribute of self and all children are greater than SortAttribute.
        vi. Parent – RedBlackTreeNode that I really need to call more often…
    b. Methods:
        i. Lots of getters and setters.
        ii. Grandparent – this.Parent().Parent()
        iii. Lots of comparison operators.

Unlike my other implementations, my code for Red-Black trees is close-enough to the pseudocode that going over my major methods would be redundant.

Further review of my implementation-based learning experience:
Much as I love binary search trees and computational geometry in general, this entire learning experience was an inadvertent lesson in patience and working to fully understand a problem prior to getting started. Erik Demaine's third MIT lecture on data structures was essentially a crash course in computational geometry which, while absolutely awesome to listen to, did not make for the most concise explanations. It was for this reason that I misunderstood self-balancing binary trees to be necessary for building Range Trees. As he had just covered Red-Black trees, and the lecture felt like a logical progression, I decided to implement them before learning more about Range Trees from other resources.

It was only after I had finished implementing insertion (having spent a few days on it) that I realized self-balancing binary trees are completely irrelevant when it comes to Range Trees. When I later mentioned this to my mom, she asked me if I knew how to spell the word *assume*.



Having said all this, I am very happy to have spent time learning about Red-Black trees; the most confusing aspect in my conceptual understanding was the idea of "null" leaf nodes which had color yet no value. It made thinking of the parents of leaves which, while technically having children, really didn't, especially weird. In implementing this, I also learned a great deal about comparison operators in C# as I wanted to be able to compare my node objects using "==", "!=", "<", "≤", etc.

Sources

1. https://courses.csail.mit.edu/6.851/spring12/lectures/L03.html
2. https://www.programiz.com/dsa/red-black-tree
3. https://en.wikipedia.org/wiki/Red%E2%80%93black_tree
4. https://en.wikipedia.org/wiki/Tree_rotation
5. https://www.codesdope.com/course/data-structures-red-black-trees-insertion/
    a. Most of the graphics came from here.

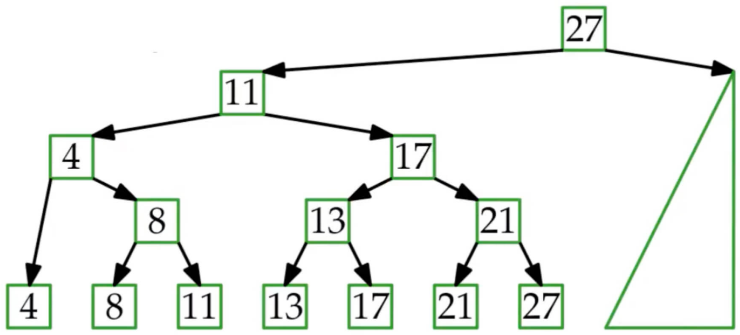
Range Trees

Conceptual Overview of Range Trees:
Range Trees are regular balanced binary search trees with two upgrades that make them extremely powerful.

1. All data is stored in the leaves, sorted on a given dimension.
2. All non-leaf nodes contain a pointer to a subtree with the same leaf nodes but sorted by their locations in the next dimension.

These changes make Range Trees ideal for representing objects with locations in any number of dimensions. Additionally, as the namesake implies, Range Trees are too ideal for range queries; trivially, they can be performed with time complexity $O(\log^d n + k)$, with n representing the number of nodes in the tree, d being the number of dimensions represented in the tree, and k being the number of points reported by the query in question. With some additional work, range tree queries can be accomplished with time complexity $O(\log^{d-2} n + k)$, which will be discussed in the *Fractional Cascading Section*.

In Range Trees, non-leaf nodes store the same data as one of the leaves in their subtree. The chosen leaf depends on the traversal pattern best for the problem in question, the only rule for selecting the leaf is that it *mostly* maintains integrity of the binary search tree – each non-leaf node has a left child and right child; the left child has values which are *mostly* less than the value at the root of the given subtree, the right has values *mostly* greater. The term mostly is used because, as the non-leaf nodes are also leaf nodes, their values must be compared using less than or equal to, or greater than or equal to; regular binary search trees do not check for equality as their leaves are distinct from their non-leaves. Traditionally, Range Trees are organized such that non-leaf nodes store the values of the node with the highest location in the current dimension in their left-child subtree. Below is a simple example in one dimension.



This way, when searching for a value **x**, a simple binary search is performed:

```
if node is leaf:
      return node
else if (node.data ≤ x):
      recurse left
else:
      recurse right
```

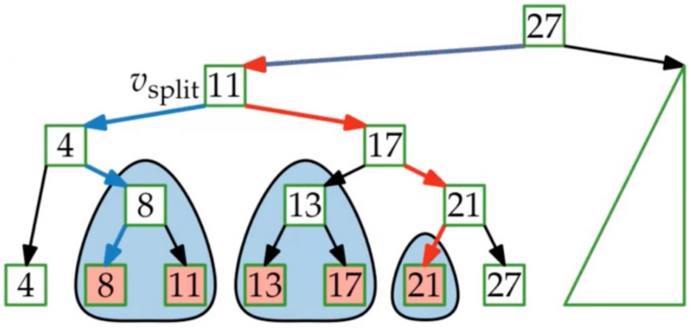
This search will return either x or the node with the lowest location value (in the current dimension) that is greater than x. As with regular binary trees, the time complexity of this search is O(log n).

Range Queries using Range Trees (single-dimension orthogonal range search):
The process for performing a (trivial) range query on a Range Tree is simple:

1. Search for the lower bound, store the path taken.

2. Search for the upper bound, store the path taken.
3. Traverse both paths until they diverge. Store this vertex (non-leaf node) as *vSplit.*
    a. *vSplit* represents the subtree whose set of leaf nodes is inclusive of the range being queried. It can still contain nodes with values out of range. We address this by finding *canonical subsets*, or subtrees of *vSplit* whose children are within our search range.
4. Starting at the index of *vSplit* in each of the two path lists, for each subsequent step:
    a. If the path is to the lower bound and the current step is to the left, store the right child as a canonical subset.
    b. If the path is to the upper bound and the current step is to the right, store the left child as a canonical subset.
5. Return all of the leaves in the stored canonical subsets.
6. Depending on the tree's organization methodology, remove the one or two outliers (clarified below).

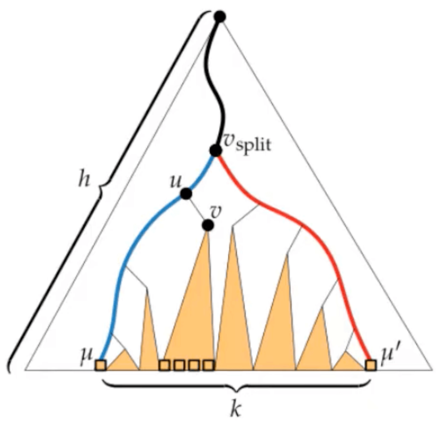
Here is the same Range Tree example illustrating a range search for nodes located in [6, 11]. The subtree rooted at vertex with location 11 is *vSplit*. Blue arrows represent the path to the node closest to the lower bound and red arrows represent the path closest to the upper bound. The blue boundaries surrounding some of the subtrees represent the canonical subsets.



In this case, 8 is selected for the lower bound because, in this implementation, search looks for either the node with value equal to the search value or the node with the smallest value greater than the search value. The same goes for the selection of 21; had the value in the right-most left subtree with root vSplit been higher than 21 (but lower than 27), it would have still been selected. This would be an example of an outlier which would need to be removed post-search.

Note that:
1. The number of canonical subsets returned is at most twice the height of the tree, or 2log(n).
2. Canonical subsets are technically defined as the set of nodes contained within any tree/subtree. We simply select the ones whose elements are in range.
3. Single-dimensional range trees can be built with time complexity O(n log n).
4. Single-dimensional range queries are accomplished with time complexity O (k + log n).

Orthogonal Range Search using Range Trees:
The process for extending range searching to multiple dimensions, ie. Orthogonal Range Searching is a simple extension of regular range searching. Instead of returning the canonical subsets found in the first dimension, iterate through them and perform the same process using each canonical subset's pointer to its subtree in the next dimension. Only return the list of canonical subsets when the dimension is equivalent to the dimensionality of the entire Range Tree.



This process can be thought of as narrowing down the search range, one dimension per recursive call. When the lists of canonical subsets are finally returned, the base case will be hit on recursive calls in the highest dimension; when these are passed to the function call from the first dimension, they will have been stripped of any node whose value is out of range in any dimension. As this approach applies to more data than that of a single dimension, we must modify the time complexity to include it such that queries are now accomplished with time complexity of O(k + log^{d}n), where d represents the dimensionality of the Range Tree.

Range Tree Construction:
Range Tree construction is even more simple than range querying. Given a data set with locations in one or more dimensions:

1. (If not at highest dimension) recurse on current dimension + 1 using the entire data set.
    a. Store the returned node as the root of the current subtree in the next dimension.
2. Check if data set contains one element.
    a. If so, store as a leaf with location attribute being the location of the leaf in its current dimension, return.
3. Sort data on its location in current dimension.
4. Find the middle index of newly sorted data set. Split data set into left and right sub-lists.
    a. (With traditional implementation) set the last element in the left sub-list as the location (sort attribute) of the non-leaf node being created.
5. Recurse using both sub-lists in the current dimension.
    a. Store returned nodes as left and right children.
6. Return self.

As we split the data set in two with each recursive call, we can be certain that the resulting tree is balanced. The process has a time and space complexity of O(n log^{d}n), with time being larger than space for the following reasons:

1. Regarding space complexity, each node is stored one or more times (non-leaf nodes always being leaf nodes in their subtrees) we have more than n possible nodes for each

level of the tree. The height of the tree is equivalent to log(n).

2. Regarding time complexity, at each recursive step into the next dimension, the nodes must be re-sorted, a process with time complexity O(n log n). While this does not change the value of the time complexity relative to space, it does increase the value of its hidden constant.

Discussion of My Range Tree Implementation:
Implementation of Range Trees and their range-searching functionality was extremely helpful in my conceptual understanding of their process. In doing so, I had a relatively easy time implementing the tree-construction functionality but an extremely difficult time implementing the orthogonal range searching functionality. Below I will demonstrate a randomly generated example of a 2-dimensional Range Tree. I will then go over the process of construction and search in the context of my implementation. Afterwards, I will review some of the errors I made along the way, how I resolved them, and the lessons I took. Note that the basis for my RangeTree implementation is my CoordNode class (see Fractional Cascading Matrix section).

(Examples on next page due to MS Word formatting issues.)

Example with 10-node, 2-dimensional Range Tree:

Note: upper subtree -> left child, lower subtree -> right child

```
Dimension 1:
[51] (10, 30, 45, 50, 51, 52, 55, 61, 70, 83)
├────────── [45] (10, 30, 45, 50, 51)
│           ├────────── [30] (10, 30, 45)
│           │           ├────────── [10] (10, 30)
│           │           │           ├────────── [10] - Data: 56, (x: 10, y: 79)
│           │           │           └────────── [30] - Data: 41, (x: 30, y: 39)
│           │           └────────── [45] - Data: 76, (x: 45, y: 83)
│           └────────── [50] (50, 51)
│                       ├────────── [50] - Data: 70, (x: 50, y: 70)
│                       └────────── [51] - Data: 28, (x: 51, y: 62)
└────────── [61] (52, 55, 61, 70, 83)
            ├────────── [55] (52, 55, 61)
            │           ├────────── [52] (52, 55)
            │           │           ├────────── [52] - Data: 87, (x: 52, y: 63)
            │           │           └────────── [55] - Data: 49, (x: 55, y: 18)
            │           └────────── [61] - Data: 10, (x: 61, y: 45)
            └────────── [70] (70, 83)
                        ├────────── [70] - Data: 50, (x: 70, y: 67)
                        └────────── [83] - Data: 74, (x: 83, y: 27)

Dimension 2:
[62] (18, 27, 39, 45, 62, 63, 67, 70, 79, 83)
├────────── [39] (18, 27, 39, 45, 62)
│           ├────────── [27] (18, 27, 39)
│           │           ├────────── [18] (18, 27)
│           │           │           ├────────── [18] - Data: 49, (x: 55, y: 18)
│           │           │           └────────── [27] - Data: 74, (x: 83, y: 27)
│           │           └────────── [39] - Data: 41, (x: 30, y: 39)
│           └────────── [45] (45, 62)
│                       ├────────── [45] - Data: 10, (x: 61, y: 45)
│                       └────────── [62] - Data: 28, (x: 51, y: 62)
└────────── [70] (63, 67, 70, 79, 83)
            ├────────── [67] (63, 67, 70)
            │           ├────────── [63] (63, 67)
            │           │           ├────────── [63] - Data: 87, (x: 52, y: 63)
            │           │           └────────── [67] - Data: 50, (x: 70, y: 67)
            │           └────────── [70] - Data: 70, (x: 50, y: 70)
            └────────── [79] (79, 83)
                        ├────────── [79] - Data: 56, (x: 10, y: 79)
                        └────────── [83] - Data: 76, (x: 45, y: 83)
```

Example with same tree as above showing left child of Root in 2-dimensions:

```
Dimension 1 – root.Left()
[45] (10, 30, 45, 50, 51)
├────────── [30] (10, 30, 45)
│           ├────────── [10] (10, 30)
│           │           ├────────── [10] - Data: 56, (x: 10, y: 79)
│           │           └────────── [30] - Data: 41, (x: 30, y: 39)
│           └────────── [45] - Data: 76, (x: 45, y: 83)
└────────── [50] (50, 51)
            ├────────── [50] - Data: 70, (x: 50, y: 70)
            └────────── [51] - Data: 28, (x: 51, y: 62)

Dimension 2 – root.Left().GetNextDimRoot()
[70] (39, 62, 70, 79, 83)
├────────── [62] (39, 62, 70)
│           ├────────── [39] (39, 62)
│           │           ├────────── [39] - Data: 41, (x: 30, y: 39)
│           │           └────────── [62] - Data: 28, (x: 51, y: 62)
│           └────────── [70] - Data: 70, (x: 50, y: 70)
└────────── [79] (79, 83)
            ├────────── [79] - Data: 56, (x: 10, y: 79)
            └────────── [83] - Data: 76, (x: 45, y: 83)
```

Code details:
To implement my Range Tree and Orthogonal Range Search, I used C# and wrote three classes:

1. RangeTree.cs – the class that represents the tree itself and querying functionality.
    a. Attributes:
        i. Dimensionality – number of location values per CoordNode in Range Tree
        ii. Root – root of the Range Tree in the X dimension
    b. Methods:
        i. BuildRangeTree(…) – Recursively constructs the Range Tree, and then populate the global variable *Root*.
        ii. FindNode(…) – Finds either the node with the target location value or that with the smallest location value greater than the target (ie. the successor).
        iii. OrthogonalRangeSearch(…) – Recursively narrows down the set of CoordNodes until their locations are all in the given range.
        iv. SearchRec (…) - Located within the OrthogonalRangeSearch method, a method that recurses on itself for most of the range-querying functionality.
        v. Getters and setters.

2. RangeTreeNode.cs - the class that contains all of the attributes and utilities needed for a given node in a range tree.
    a. Attributes:
        i. Data – the list of CoordNodes in its subtree.
        ii. Dimension – the dimension on which these nodes are sorted.
        iii. LocationVal – the highest location value of the current dimension in its left subtree.
        iv. NextDimensionRoot - the root of the Range Tree containing the same leafNodes sorted by Dimension + 1.
        v. ParentNode
        vi. LeftChild
        vii. RightChild
    b. Methods:
        i. Mostly getters and setters for above attributes.

3. RangeTreeHelper.cs – helper class with a visualization function for looking at the tree.

Method overview - RangeTreeNode **BuildRangeTree(**CoordNode[] **coordSubset,** int **currDim) :**
Parameters:

1. coordSubset - a list of CoordNodes with location values in between one and three dimensions.
2. currDim – an integer representing the current dimension of the tree being constructed (starts with 1).

Return: void, assigns global variable Root.

Process:

1. Instantiate the RangeTreeNode of the current dimension and subset of coordNodes.
    a. thisNode <- RangeTreeNode(coordSubset, currDim).
2. Recurse to build the RangeTree for the next dimension.
    a. If currDim < Dimensionality, thisNode.NextDimRoot = BuildRangeTree(coordSubset, currDim + 1).
3. Base case – check if thisNode is a leaf node (if coordSubset contains only one element).
    a. If coordSubset.Length == 1, set location value of thisNode to that of the lone CoordNode in coordSubset.
4. Sort coordSubset on the coordNodes' location values in currDim.
5. Find middle-most index of newly-sorted coordSubset.
6. Create two lists from each half, for both the left and right list:
    a. Recurse using the new subset and the current dimension.
    b. Set returned node as the left/right child of thisNode
    c. Set the parent of the left/right child as thisNode.
7. Add the sortAttribute of thisNode using the last element in the (sorted) left-half list, such that the sort attribute for thisNode is the highest location in currDim in its left subtree.
8. Return thisNode (to be set as Root by constructor).

Method overview - RangeTreeNode **FindNode(**RangeTreeNode **root,** int **target,** List<(int, RangeTreeNode)> **pathList=null) :**

Parameters:

1. root - the root of the current subtree in which we are searching.
2. target - the location value in the current dimension of the node for which we are searching.
3. pathList - list containing tuples with (int, RangeTreeNode) to denote the route taken by this search. Note that this list must already exist such that it will be populated after the method has finished running.
    a. int = 0 -> recurse left, int = 1 -> recurse right.
    b. RangeTreeNode - node on which we are recursing left or right.

Return:

The RangeTreeNode whose location value is equal to target or that whose location value is the lowest in the current dimension that is greater than target.

Process:

1. Check if root is a leaf (ie. it's subset of coordNodes contains only one).
    a. If so, and pathList is not null, add tuple (-1, root) to pathList.
    b. Return root.
2. Check if target is less than or equal to location at Root.

    a. If so, and pathList is not null, add tuple (left, root) to pathList.
    b. Recurse using the left child of root, target, and pathList.
3. Else:
    a. …. (right, root)…
    b. … right child …

Method overview - CoordNode[] **OrthogonalRangeSearch(**int[] **rangeMins,** int[] **rangeMaxes) :**
Parameters:

1. rangeMins – a list of integers of size=Dimensionality where each element represents the (inclusive) lower bound of the search in each dimension.
2. rangeMaxes - a list of integers of size=Dimensionality where each element represents the (inclusive) upper bound of the search in each dimension.

Return:

The subset of the coordNodes in the RangeTree whose locations in each dimension are within the inclusive bounds set by rangeMins and rangeMaxes.

Process:

1. Define method SearchRec to handle the majority of the functionality.
2. Call SearchRec using the Root of the entire tree and the first dimension
    a. canonicalSubsetsInSearchRange <- SearchRec(Root, 1) .
3. Instantiate an empty list of CoordNodes L .
4. For each of the RangeTreeNodes in canonicalSubsetsInSearchRange, append the given RangeTreeNode's list of CoordNodes to L .
5. Return L

Method overview - List<RangeTreeNode> **SearchRec(**RangeTreeNode **root, int** currDim**) :**
Note:

searchRec is contained within the OrthogonalRangeSearch method, meaning that it has access to all of the parameters of OrthogonalRangeSearch .

Parameters:

1. root – the root of the subtree in which we are searching.
2. currDim – the current dimension of the root of the subtree in which we are searching

Return:

A list of RangeTreeNodes whose subsets of CoordNodes are within the confines of the ranges in each dimension.

Process:

1. Assign variables rangeMin and rangeMax as the inclusive lower and upper location bounds in currDim.

2. Instantiate a quick helper function InRange(int data) that checks if data is greater than or equal to the current value of rangeMin and less than or equal to the current value of rangeMax.
3. Instantiate new lists rangeMinPath and rangeMaxPath containing tuples:
    a. (int, RangeTreeNode) – see *FindNode* method description.
4. Instantiate two RangeTreeNodes rangeMinNode and rangeMaxNode:
    a. rangeMinNode <- FindNode(root, rangeMin, rangeMinPath)
    b. rangeMaxNode <- FindNode(root, rangeMax, rangeMaxPath)
5. Instantiate a boolean variable, pathsDiverge:
    a. pathsDiverge <- rangeMinNode.Location ≠ rangeMaxNode.Location
    b. If this is false, it means that rangeMinPath is equal to rangeMaxPath, which means that there is only one node in the subtree which may or may not be in range.
6. Instantiate new list of RangeTreeNodes called canonicalSubsets, which will contain descendants of root representing its canonical subsets within the bounds of the current dimension.
7. Instantiate integer variable pathsDivergeIndex to zero. This represents the index at which the paths to the lower and upper bounds diverge, ie the index in either of the lists containing the RangeTreeNode equivalent to vSplit.
8. Check if the paths do not diverge. If this is the case, set pathsDivergeIndex to the second to last index in the shorter of the two paths.
    a. If (! pathsDiverge): pathsDivergeIndex <- rangeMinPath.Count - 2;
9. If the paths do diverge, traverse both simultaneously. At each iteration, check if:
    a. Both children are leaves. In this case, break, as the pathsDivergeIndex to be used is the last one to be checked.
    b. Check if their directions **do not** differ. In this case, assign pathsDivergeIndex to this index + 1, as both paths go into the same subtree.
    c. Else, if their directions **do** differ, break because the subtree at the current index in both path-lists represents vSplit.
10. Instantiate a list of RangeTreeNodes called canonicalSubsets.
11. Partially populate canonicalSubsets using the path to the lower bound:
    a. Check if rangeMinPath is of length 1. If so, check if the sole RangeTreeNode in rangeMinPath is a leaf. If so, check if the leaf's location is in range. If so, add the RangeTreeNode to canonicalSubsets.
    b. Else, if rangeMinPath is of length greater than one:
        i. Iterate through rangeMinPath, starting at index pathsDivergeIndex + 1. This way, the first node is on its own path to the lower bound, completely distinct from rangeMaxPath.
        ii. For each iteration, first check if the RangeTreeNode at the current index is a leaf. If so, check if it is in range. If so, add it to canonicalSubsets.
        iii. Else, if the RangeTreeNode at the current index is not a leaf, check its direction. If the next step is into the left subtree, add the right subtree to canonicalSubsets.
12. Finish populating canonicalSubsets using the path to the upper bound:

    a. Before starting, check if pathsDiverge is false. In this case, the route to the upper bound is contained within the path to the lower bound and our work here is done.
    b. If pathsDiverge is true, repeat the exact same process in part 11 with two differences:
        i. Iterate through rangeMaxPath instead of rangeMinPath.
        ii. For step 11.b.iii, do the following instead:
        Else, if the RangeTreeNode at the current index is not a leaf, check its direction. If the next step is into the *right* subtree, add the *left* subtree to canonicalSubsets.

13. Check for the case where no nodes are in range, such the only RangeTreeNode in canonical subsets is a successor of rangeMax.
    a.
    ```
    if (canonicalSubsets.Count == 1 && canonicalSubsets[0].IsLeaf()):
        if (! InRange(canonicalSubsets[0].Location )):
            canonicalSubsets = new empty list
    ```
14. Recurse on next dimension on each canonical subset.
    a. Instantiate empty list of RangeTreeNodes called nodesInRange
    b. Check if we are at the final dimension.
        i. If so, set nodesInRange <- canonicalSubsets.
    c. Else, if we are not in the final dimension, iterate through each canonical subset and recurse using the RangeTree's subset of nodes and the next dimension. At each step, append the coordNodes subsets returned by the recursive call to nodesInRange.
        i.
        ```
        foreach (RangeTreeNode canonicalRoot in canonicalSubsets):
            nodesInRange.append(
                SearchRec(canonicalRoot.NextDimRoot(), currDim + 1))
        ```
15. Return nodesInRange!

Further review of my implementation-based learning experience:

Implementation of the BuildRangeTree method was straightforward with one exception. Due to my testing with relatively small trees, it took me a long time to realize that I was making the mistake of sorting on the current dimension prior to the recursive call which builds the tree in the next dimensions. I only realized this was a problem when attempting a single-dimensional range query and, at one point, only had values exclusive of my range. My first instinct was that this was an issue with my search functionality, which made the actual problem take much-longer to diagnose.

Implementation of the SearchRec method, on the other hand, was probably my most difficult experience during this independent study due to several perfect storms of small mistakes. The first was a small error in which I had two variables: *Root*, which referenced the RangeTreeNode at the root of the entire Range Tree, and *root*, which referenced a RangeTreeNode representing the root of a subtree during a call to the method FindNode. This had me banging my head against the wall for hours as I tried to fix what turned out to be a great deal of other edge cases thinking they were the cause of my issues.

I also went through several iterations of the functionality which turned into steps 5, 8, and 12a under the process overview of *SearchRec*. At first, I was doing four separate checks for different circumstances in which the paths might not diverge yet still hit leafNodes, the paths were extremely short, etc. Eventually, I was able to understand the redundancy of these cases, resulting in the simple solution of checking for equality in the search results in the lower and upper bounds, as well as for cases in which children were leaves prior to recursion. This in turn enabled me to eliminate unnecessary processes and, as a result, an annoying issue in which duplicate coordNodes would be returned.

The most difficult aspect of this implementation, however, was in working with the paths to the lower and upper bounds. Errors in both my conceptual understanding of the theoretical process and confusion in my own implementation made for some issues which were both not always present and extremely difficult to debug. Below is one such issue from an email I sent to Professor Bezakova asking for help. It stemmed from my misunderstanding the point at which a subtree is added as a canonical subset (on next page due to MS Word formatting issues.)

During my own implementation of Orthogonal Range Search, I ran into the following conceptual issue. Take this 2D Range Tree with the upper branch representing the left subtree and lower branch representing the right. Let A = the node with x-value 49. Let's say we are doing an orthogonal range search for values between 50 and 70 (inclusive) on the first dimension. In this situation, the subtree containing A (out of range) would be selected. I believe that any other one or two leaves we add with x values greater than 41 and less than 49 would be children of A (as they would not imbalance the tree).

```
[50] (10, 28, 41, 49, 50, 56, 69, 70, 74, 87)
|├------------ (Veer left, take everything in right subtree r as a canonical subset)
├---------- [41] (10, 28, 41, 49, 50)
||            ||
||            ├---------- [28] (10, 28, 41)
||            ||            |
||            ||            ├---------- [10] (10, 28)
||            ||            |            |
||            ||            |            ├---------- [10] - Data: 8, (x: 10, y: 54)
||            ||            |            |
||            ||            |            └---------- [28] - Data: 4, (x: 28, y: 31)
||            ||            |
||            ||            └---------- [41] - Data: 2, (x: 41, y: 22)
||            |└----------- (Veer right, don't take anything)
||            └---------- [49] (49, 50) <- subtree r, take canonical subset containing 49 and 50, but 49 is out of range!
||                          ||
||                          ├---------- [49] - Data: 7, (x: 49, y: 53)
||                          ||
||                          |└--------- [50] (Veer right)
||                          └---------- [50] - Data: 9, (x: 50, y: 55)
|└------------ (Veer right, don't take anything)
└---------- [70] (56, 69, 70, 74, 87)
              |└------------
              ├---------- [69] (56, 69, 70)
              |             || …
```

At first, I attempted to remedy this by **adding a condition that, prior to adding a canonical subset, check if the value of the non-leaf node is in range**. This fixes the search in the above example but not in this one here (search range still between 50 and 70 inclusive):

```
[51] (10, 30, 45, 50, 51, 52, 55, 61, 70, 83)
|├------------ (Veer left, first check value of non-leaf node (45) - Out of range, don't take right subtree r as canonical
├---------- [45] (10, 30, 45, 50, 51)         subset!). This situation fails because subtree r includes data with x value
||            ||                              51, which is in range!
||            ├---------- [30] (10, 30, 45)
||            ||            |
||            ||            ├---------- [10] (10, 30)
||            ||            |            |
||            ||            |            ├---------- [10] - Data: 56, (x: 10, y: 79)
||            ||            |            |
||            ||            |            └---------- [30] - Data: 41, (x: 30, y: 39)
||            ||            |
||            ||            └---------- [45] - Data: 76, (x: 45, y: 83)
||            |└------------ (Veer left)
||            └---------- [50] (50, 51)  <- subtree r
||                          |└--------- [50]
||                          ├---------- [50] - Data: 70, (x: 50, y: 70)
||                          |
||                          └---------- [51] - Data: 28, (x: 51, y: 62) <- node in range but not taken
|└------------ (Veer left, don't take anything)
└---------- [61] (52, 55, 61, 70, 83)
              ||
```

As shown in the example above, the fix for the first situation does not apply to the second one.

Conclusion

This independent study was probably the second most enthralling learning experience I have ever had at RIT. (A close second to Professor Bezakova's wonderful Analysis of Algorithms course.) This semester, RIT was still bound to Covid-19 restrictions; this meant online courses, resulting in mass lack of retention/engagement, and lack of Spring Break, resulting in mass burnout. I felt these affects intensely and this independent study was my only reprieve, the only aspect of RIT this semester from which I feel I learned anything of value.

In basing the practical implementations independent study around theoretical lectures, I learned much more than expected. This was due to the problem solving surrounding the software engineering. For example, the concept of fractional cascading is relatively easy for a lecturer to demonstrate on a chalkboard; they can arbitrarily compare nodes in a small matrix with arrow between the representing pointers. Implementing something like this is much more of a challenge than conceptually understanding it. The same was true of range trees. As they are relatively obscure, very few of the online materials on the had implementation details. While I found one implementation online, and likely could have found more, I decided against using it for reference as I wanted to ensure my understanding of every single aspect.

I would like to use this final section to give thanks to my advisor, Professor Ivona Bezakova, who enabled myself and independent study partner, Efe Ozturkoglu, to have such a learning experience (for a second time). The learning experiences she gave us encapsulate the aspects of computer science that I love the most; I feel very lucky to have had another.

Appendix

Overview of time spent – 172 hours:

- 28 hours - Lectures from Erik Demaine's Advanced Data Structures Course at MIT.
  - Each lecture is ~80 minutes. We watched one to two per week.
  - Total: 80 minutes * 1.5 lectures per week * 14 weeks = ~19 hours
- 5 hours – Perlin Noise Project
  - Took around one week to implement.

- 80 hours – Fractional Cascading Matrix project
  - Took 3 – 5 weeks to implement
  - 35 hours- development of tools used in later implementations including:
    - CoordNode data structure which supported arbitrary data as well as location in up to three dimensions
    - Tools to generate data sets of nodes randomly, including a function to generate a list of non-repeating integers in a given range quickly
    - Tools to measure and record the performance of the process
    - Merge sort
    - Binary search
    - Visualization functionality for my nodes, lists of my nodes, and matrices of my nodes
  - 5 hours - re watching one of Erik Demaine's relevant several times
  - 30 hours – implementing the data structure/transformation itself
    - 5 hours – node structure specific to fractional cascading – each contains an instance of the node mentioned above.
    - 15 hours – the fractional cascading transformation itself
      - Matrix of CoordNodes -> Transformed matrix of FractionalCascading Nodes
    - 5 hours – query functionality
      - 4 hours – querying matrix of FractionalCascading Nodes
      - 1 hour – trivial solution using k binary searches.
  - 5 hours – testing
  - 5 hours – optimization for faster testing (see my *Further Review* section within my *Fractional Cascading Matrix* section)

- 35 hours – Red-Black tree implementation
  - 10 hours – learning the implementation and theory
  - 8 hours – insertion functionality
    - 1 on insertion, 7 on rebalancing post insertion
  - 12 hours – deletion functionality
    - 4 on deletion, 8 on rebalancing post deletion
  - 5 hours – functionality to visualize tree for testing/report

- 25 hours – Range Tree Implementation
    - 6 hours – construction, this one was relatively simple
    - 6 hours – watching and re-watching Philipp Kindermann’s lectures on the subject
        - https://www.youtube.com/playlist?list=PLubYOWSl9mIsOJW-u2ZusOJzZvhNi0xks
    - 13 hours – orthogonal range searching using range trees – I ran into many conceptual-understanding and software engineering related issues on this one.

- 10 hours – writing my project report