

Segment Trees
Min/Max and Summation Queries and Insertions

James McCaffrey

Shuichi Kameda

Evan Ung

Michael Gilkeson

University of Rhode Island

CSC-212

Jonathan Schrader

7/24/23

Table of Contents

1.	Introduction to Segment Trees	3
1.1.	Context & purpose / Surface level details	3
2.	Introduction to Our Project	4
3.	Methods	5
3.1.	Binary tree	5
3.2.	Construction	6
3.3.	Operations	8
3.4.	Space and time complexity	9
3.5.	Variants	10
3.6.	Applications	10
3.7.	Limitations	11
4.	Implementation	12
4.1.	Description of what the code does	12
4.2.	Noteworthy code portions	21
5.	Conclusion	31
6.	Contributions	32
7.	Works Cited	34

Segment Trees

1 INTRODUCTION TO SEGMENT TREES

1.1 Context & purpose / Surface level details

When working with some kind of collection, whether it be a vector, a list, an array, or some other type, it can be costly to get information on a certain segment of the collection. For example, if someone was working with an integer array of some arbitrary length and wanted to get the average of the elements of the vector starting at index 3 to index 7, that person would have to go through each element in that specified range in order to calculate the average. This operation would be done in linear time or $O(n)$. But there is a potentially more time-efficient way of doing such an operation when utilizing a segment tree, which could do it in logarithmic time or $O(\log(n))$.

A segment tree is a binary tree where each node stores some data that is relevant to its corresponding range of elements originating from the collection used to generate the binary tree. For example, each node could store data such as the minimum and/or maximum value or a summation of all of the elements in its specified range. The nodes will also have pointers to its left and right child as this is necessary in order to generate the segment tree.

The way a segment tree is structured helps facilitate the logarithmic time efficiency of queries. The root node holds data from every element of the collection from index 0 to the last index. Then (if possible) the root node will have left and right children nodes, where the left child node will hold data from the left half of the interval of its parent node and the right child

will hold the right half. Those child nodes will (if possible) have their own child nodes and the interval the node holds will be halved again and distributed to their corresponding child nodes. This will continue until a child node's interval is only one element, meaning that node can no longer have child nodes.

The way each node holds data relevant to a certain segment of a collection and the way the tree is structured allows for a logarithmic time complexity for queries about a certain portion of a collection. This, of course, will take up more memory than the collection used to generate the segment tree, so when making the decision on whether or not to implement this data structure, it is important to keep this trade off in mind.

2 INTRO TO PROJECT

The project our group worked on aimed to solve the problem of finding information about gas prices in a specified area. Using an inputted starting longitude and latitude coordinate along with information on different gas station prices and their coordinates the program is able to get the average, minimum, and maximum gas prices of gas stations in a specified range from the starting position. The program is also able to update the price of gas at a specific gas station.

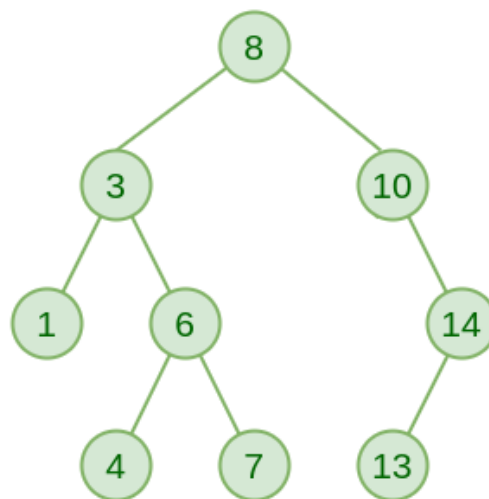
Our group chose this implementation because we thought it was relevant to solving a real world issue and that we could use a segment tree to our advantage when it comes to queries. Because our implementation solves this real world problem, and would benefit from the segment tree, we decided to create our program as we believed that the idea and data structure had synergy.

3 METHODS

As mentioned earlier in this paper, segment trees can be utilized to perform efficient range queries and element modifications over an array. Its efficiency is mainly due to how segment trees are functionally derived from a fundamental data structure known as the binary tree. Before going into more detail about segment trees, we will provide some basic information on its predecessor.

3.1 Binary tree

The binary tree is a tree-based data structure from which several other structures are derived. Its most defining feature is its hierarchical structure, which involves data being built via a parent-child relationship. A node is the element that drives this particular property, and at the very top level of a binary tree exists the root node. From the root node, the tree branches into at most two children, which go on to establish the left and right subtrees.



Binary tree - <https://www.geeksforgeeks.org/binary-search-tree-data-structure/>

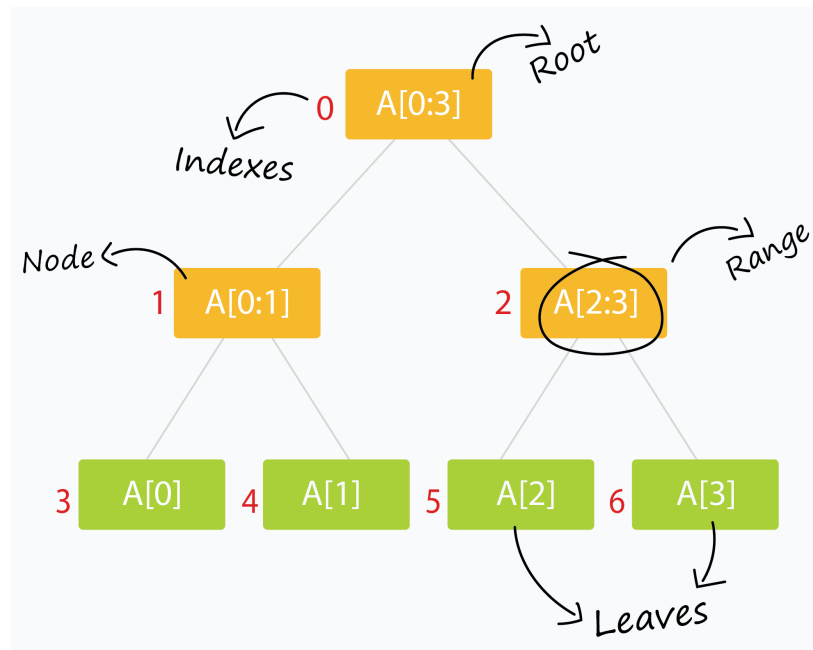
Each node is capable of holding a value from a given array as well as a pointer to a left and right child. The manner in which the values are inserted into a binary tree follows a distinct property. Whenever new nodes are created, they are first compared with the parent node. For values less than the parent, the child is inserted into the left subtree. For values that are greater, the child is inserted into the right subtree. Generally, the more nodes that are inserted into the tree, the larger the height, with height being defined as the largest distance between the root node and the leaf node at the very bottom.

Given the structure of a binary tree, certain functions can be performed at a higher efficiency compared to conventional methods like using a simple array. For example, if you were to traverse an array in order to find a particular value, this operation would typically take $O(n)$ time due to having to iterate one element at a time. This same search operation when performed on a relatively balanced binary search tree will only take, on average, $O(\log(n))$ time as a result of the nodes effectively splitting at every level. Binary trees do not always beat conventional arrays in every aspect regarding efficiency, but they are useful in certain use cases where large datasets are involved.

3.2 Construction

Segment trees are a specialized full binary tree that holds data regarding the range of elements within its nodes. At the root of the tree, the node holds the entire initial array. Every subsequent node holds half of the parent node's range. For example, given an array with the following elements $[0, \dots, N - 1]$, where N represents the size of the array, the left child will take

in the interval $[0, \dots, N/2 - 1]$, while the right child will take in $[N/2, \dots, N - 1]$. This format will continue until each leaf node corresponds to the individual elements of the initial array.



<https://leetcode.com/articles/a-recursive-approach-to-segment-trees-range-sum-queries-lazy-propagation/>

To display the values of each node in the segment tree as a linear array, you can access the indices of the tree using $(2*i + 1)$ and $(2*i + 2)$, where i represents the level. For example, if you wanted to display the contents of the nodes in the first level in the figure above, the indices would be $(2*1 + 1) = 1$ and $(2*1 + 2) = 2$, respectively. At index 1 of the segment tree, you would receive the value associated with interval $A[0:1]$. Generally, the size of the tree array will be $4*N$ since, by nature of a full binary tree, it will contain $2*N$ internal nodes and $2*N$ leaf nodes.

When constructing the segment tree, the divide and conquer method is often used. Starting at the root, you utilize two recursive calls to build the left and right subtree by computing the midpoint at each level in order to halve the current range. This top-down approach simply builds the nodes with their respective ranges. To actually include the values that the ranges correspond to, you will need to employ a bottom-up recursive approach. Since we know that each of the leaf nodes is an individual element of the initial array, we begin by performing the desired action, then propagating those values up the tree. For example, if we are interested in showing the sum at each node, we determine the individual values of the children and set the parent node's value equal to that sum. We continue moving up until you reach the root node, which should encompass the entire initial range.

3.3 Operations

Segment trees give the user access to two types of operations: queries and updates. For query operations, you can retrieve data without modifying any values in the tree. Some examples of commonly used queries involve finding the maximum, minimum, sum, or average of a specified interval. To perform these queries, the program starts from the root node, and recursively travels down the tree until the specified range is found. Since the segment tree is a binary tree, the height will be approximately $\log(n)$, where n is the size of the initial array. For root-to-leaf traversals, the program will take $O(\log(n))$ time to complete query operations.

For update operations, the program finds the node within the tree and updates the values. The program must then update all of the parent nodes up to the root node. These two processes

occur sequentially, and each takes approximately $O(\log(n))$ time to complete, therefore the time complexity of update operations is $O(\log(n) + \log(n)) = O(\log(n))$.

Operation	Time Complexity
Search	$O(\log(n))$
Min/Max	$O(\log(n))$
Get Average	$O(\log(n))$
Sum	$O(\log(n))$
Update	$O(\log(n))$

3.4 Space and time complexity

What makes segment trees a viable data structure to use is its space and time complexity. This space and time complexity will vary depending on the input size. With larger data sets, segment trees are a good choice because of efficiency. However, with smaller sets, it is much more viable to use another data structure.

Since segment trees are binary trees, they are similar in time and space complexities. As mentioned in the binary tree description, segment trees have a space complexity of $O(n \log n)$. The time complexity is also $O(n \log n)$. However, with the addition of persistent segment trees the space complexity will increase. Since persistent trees keep history of past versions of trees, the space complexity will increase depending on how many versions are saved. This translates to $O(n \log n + k \log n)$, K being the number of versions available.

3.5 Variants

As for variants of segment trees, there are several noteworthy ones. Segment trees are very versatile. As mentioned in a segment tree's time and space complexity, it can be made into a persistent tree. A persistent data structure is one that needs to retain both past and current data. This is useful when previous versions of the data need to be kept. For example, applied applications are using a redo and undo button. If past data is deleted, then changes cannot be redone.

Segment trees can also become persistent data structures. When modifying data, we would make use of the current tree. It would then be cloned and necessary changes would be made. We are then able to save both current and previous trees. However, it is important to know that this method has increased space usage. It will become more apparent with the use of larger data sets.

3.6 Applications

When it comes to segment trees, there are various applications where segment trees can be used. In our project, we decided to make use of how well segment trees can process ranges. Segment trees are efficient in storing and finding range queries.

A common application is the use of finding a minimum, maximum, sum, or average within a data range. This makes segment trees useful in database management systems. Databases usually have larger amounts of data so a data structure that can find an average over a specific range efficiently is essential. Other applications are image processing as well as analyzing a series of timed data. These all make use of finding minimum, maximum, sum, etc.

3.7 Limitations

There are things to keep in mind before pursuing the implementation of a segment tree. Segment trees will take up more memory than simple collection types like arrays or vectors, so if someone is working with constraints on memory, a segment tree might not be the best approach. A segment tree is probably not worth implementing, if the collection used to generate the segment tree is relatively small. For example, if someone was working with some array that has ten elements, it is probably not worth using a segment tree, even though it has a more efficient time complexity on paper. Another disadvantage segment trees have that hold them back is their insertions compared to an insertion on a linear collection type. Changing an element on a collection like an array, for example, would have a time complexity of $O(1)$ as only the element at the specified index would have to be updated. But doing the same thing in an insertion tree requires each node that contains the position that is being updated will have to be updated, making the insertion process have a relatively less time efficient time complexity of $O(\log(n))$.

But these disadvantages and certain situations where implementing a segment tree might not be the best approach does not mean that segment trees will not be useful. As well as the more general querying use cases mentioned earlier, segment trees have more uses in specific fields and applications. For example, segment trees are used in computational geometry, geographic information systems, and image processing. Segment trees when used in the before mentioned applications are quite useful.

4 IMPLEMENTATION

4.1 DESCRIPTION OF WHAT THE CODE DOES

Our implementation was to create a program that would take in a text file as input that contains a latitude and longitude coordinate which would be the starting position in the first line, and every line after that would contain a gas price and the longitude and latitude coordinates of a gas station. The program would read in this file and create a two-dimensional double vector that contains a gas station's gas price, and the distance between the gas station and the inputted starting position. The distance will be calculated using the haversine formula. Each subvector in the two-dimensional vector will be of length two as it only stores the price and distance.

Once this two-dimensional vector is created, it will then be sorted using a quick sort function and its auxiliary, which will sort the vector in ascending order based on its distance from the starting position. Quick sort was chosen due to its relatively quick time complexity of $n \log n$ for its average case, and we believed that the vector would not be short enough to benefit from sorting algorithms like bubble sort or insertion sort.

Once the vector is sorted, a segment tree object will be generated with a function that will take in the sorted vector and generate a segment tree with the intervals being taken from the vector that was taken in. The data that each node will hold will be as follows: the summation of all gas prices in the interval, the lowest gas price in the interval, and the highest gas price in the interval. The function that generates the segment tree is recursive and takes in a low index and high index. When it is first called it will take in a low index of zero and high index of the length of the vector minus one.

The function will do as follows: first, it will check if the low index is equal to the high index as if this happens, that means that the only node that can be created is a leaf node. If these values are equal, it will then generate a new node that which has no child nodes, so its left and right pointers will be null, and its low price, high price, and sum price will all be the same value, as there is only one element in the vector to work with. This node will then be returned. If the low index and high index are not equal to each other, a variable called middle will be instantiated with the value of the ceiling of the low index plus the high index divided by two. This middle variable will be used for generating more nodes later. Then a new node is generated that recursively calls the tree generation function to instantiate its left and right children. The left child will call gen tree with the same low index, but its high index will be the middle variable subtracted by one. The right child will call gen tree with the middle variable as its low, and the same high index. This is so the left child will contain the left half of its parent interval, and the right child will contain the right half of its parent interval.

This recursion will continue until the low index and high index are equal, as mentioned before, and will then return a leaf node. When this leaf node is returned, its sum gas price will be added to its parent node's sum price, and the parents low price will be set to the lowest low gas price of its children, and the high price will be set to the highest high price of its children. This parent node is then returned. Once the function is done, a segment tree is generated.

Once the Segment tree is generated, the user of the program will be given a list of options printed in the console to choose from. The options will be the following: changes the price of gas

at a specified gas station, get the average price of gas from the starting position to a specified distance, get the average price of gas from a specified low distance, to a specified high distance, get the lowest and highest gas prices from the starting position to a specified distance, and get the lowest and highest gas prices from a specified low distance to a specified high distance. The user can select which option they would like to choose by inputting the option's number into the console.

If option one is chosen, then a function will be called that prints out the contents of the input two-dimensional double vector along with the index of each subvector. The program will then ask the user to input the index number of the gas station they would like to change the price of. The program will then ask the user what the new price of gas should be set to, and takes in a user input. The program will then try to call the insertion function, but will have a try catch in case the user inputted an invalid input.

If option two is chosen, the program will ask the user to input a maximum distance. If the input was valid, the program will then perform a query on the segment tree to calculate the average price of gas in gas stations located in the radius starting from the starting position to the inputted maximum distance.

Option three is similar to option two due to both of them calling a get average query from a specified distance, but this time it will take in a minimum and maximum distance. The program will ask the user to input a minimum distance, and then a maximum distance. If the input is valid

the program will perform a query on the segment tree for the average price of gas in gas stations located from the minimum distance from the starting point to the maximum distance.

Option four is also similar to option two as it gets a high distance from the starting point as input and performs a query on the segment tree, but this time it gets the minimum gas price and maximum gas price in the specified range. The same can be said for option five as it will take in a low and high distance as input and perform an interval query of the tree to get the low and high gas prices.

Moving on from the options, the next function to be analyzed will be the modified binary search function in the `seg_tree.cpp` file. This is a very important function as it will take in a distance as a value and a boolean called `mode`, and find the index of the input vector which contains an element that has the closest distance to the inputted distance. The function will either round up or round down the index based on the `mode` boolean. The first part of this function is just a regular binary search, but since it is very unlikely the user will input a distance that happens to be exactly equal to a distance of some element in the input vector, the function requires a modification to approximate the distance. Where a regular binary search would return some value that indicates that the target is not contained in the collection, this binary search will instead round up the index if the `mode` is true, or round down if the `mode` is false. The reason why there are these two round modes is that when finding a low index, it is necessary to round up, and when finding the high index, it is necessary to round down so that the returned indexes will not be outside of the specified range.

Next are the query functions for getting the average gas price in a specified distance. There are three overloaded functions for this to handle different kinds of input, and setting up the inputs for the recursive get average function. The first of these overloaded get average functions is the one that takes in a double low and a double high as a parameter. First it is going to use the previously mentioned modified binary search function in order to get the low and high indexes based on its double low and high parameters, as they will contain the low and high distances for the query. Then a possible error is handled where if the high index is negative one, the program will return -999 as this index does not exist and this is a possible output for the binary search function as the high does round down. -999 will be the return value if there is an error with any inputs for the rest of the functions. But if the indexes are valid, the overloaded get average function that actually gets the average will be called now that the low and high indexes have been acquired.

The next get average function is the one that only takes in a high value. It functions exactly the same as the previously mentioned get average function, but it will have the low index value always be zero. It gets the high index the same way, handles the negative one high index error the same, and calls the overloaded get average function the same, but the low index will always be zero.

Next will be the actual get average function, which is a recursive function that takes in integer parameters current node low and current node high to keep track of the current nodes low and high indexes and search low and search high to store the low and high indexes for the segment the function is searching for. The function will also take in a node as a parameter to

keep track of the current node. When the function is called, it will first check if the current low/high are out of range of the search low/high or if the current node does not exist. If this is the case, the function will return zero as the current node is irrelevant to the search intervals. Next, the function will check if the current low/high indexes fall in between the search low/high indexes and return the current node's gas price. If this is not the case however, the function will recursively call itself, this time with the current node being the current node's left child with updated current low/high indexes, and add the result of that to another recursive call with the current node's right child and updated current low/high indexes. The result of the summation of these two recursive calls will then be returned. All of this together allows the function to traverse only the relevant nodes of the tree, and sum up the gas prices of nodes that fall within the search intervals.

Next up is the get low high functions which function very similarly to the get average functions, but instead of getting the average gas price in a given interval, it will get the lowest and highest gas prices. Its two functions that set up the overloaded function that actually gets the desired information functions almost exactly the same as the get average functions, but instead of returning -999 if an error is encountered, it will return the pair $\{-999, -999\}$ as the low and high gas prices will be returned as a pair of doubles. Getting the low and high indexes are exactly the same though, and both of the set up functions will call the "actual" get low high function with those indexes.

The actual get low high function is also very similar to the actual get average function, but it has a few key differences that make it unique. This function will return a pair of doubles,

with the first element being the low gas price, and the second element being the high gas price. The parameters the function takes in are almost identical to the parameters of get average, with the exception of a pair of doubles called low high. This parameter will be used to keep track of the current lowest and highest gas prices in the traversed intervals. When the function is called, it will first check if the current interval is out of range of the search interval or if the current node does not exist. If these conditions are met, the current low high pair will be returned. If the current interval falls in between the search interval however, if the current node's low price is lesser than the low high's low price, the low high's low price will be set to the current node's low price, and if the current node's high price is higher than the low high's high price, the low high's high price will be set to the current node's high price. Then low high will be returned. If these conditions are not met, then two recursive calls will happen with the first setting the low high to the recursive function call where its current node is the current node's left child, and its current low/high indexes will be updated. The second recursive call will set the low high to the result of the recursive call with the current node being the right child node of the current node, with updated current low/high indexes. After this recursion, the low high will be returned.

Another very important function is the insertion function for changing a gas price at a specific index. This function will not only update the sum gas prices with this updated price, it will also change the low and high gas prices. This is an overloaded function with the first set up function taking in an int position which will be the index that will be updated, and a double value which will be the new price of gas at that index. First the function will check if the position index actually exists in the input vector. If it does not exist, the function will output to the user that an input error has occurred, and return. If the position does exist however, the actual insertion

function will be called. After the actual insertion function is complete, the gas price in the input vector will be updated as well.

The actual insertion function will change a few things in the segment tree. It will need to change the sum gas prices and the low/high gas prices and does so in a kind of two part process. First it traverses all relevant nodes of the tree and updates the sum price, then once a leaf node is reached, the function backtracks to update the low/high gas prices once all of the sum prices have been updated. The function takes in the integer parameters pos, low, and high to store the position that is being updated, and the current low and high indexes respectively. The program also takes in a double called add which will be the updated gas price. Lastly, a node will be taken in to keep track of the current node being worked with. When the function is called, it will first check if the current node does not exist. If so, the function returns, but if not the current nodes sum gas price will be updated. After that, the function will recursively call itself to traverse to either the left or the right node, depending on which node's interval contains the position being updated. Once the leaf node has been reached, the function will then begin backtracking to update the low/high gas prices, starting with setting the leaf node's low and high price to the updated gas price. Then, with each backtrack the low/high gas prices of the current node will be set to the lowest and highest gas prices of their child nodes, and then return. After the backtracking is complete, the insertion is complete, and all relevant nodes have been updated.

The last group of functions are the functions for writing the dot file, starting with the write file function, which takes in a string as a parameter that stores the input file's name. First, the function will open a file in write mode, which will be the name of the input file with

“_output.dot” added to the end of it. Then a function that makes a string from a node’s data in the format of a dot file will be called to store the string of the root node. The function will then check if the root does not exist, and if so the function will just write an empty graph. The function will then check if the root node does not have any child nodes, and if so will only write the stringified root to the dot file. If none of these conditions are met however, the write node function will be called to write all of the information of the tree’s nodes to the dot file.

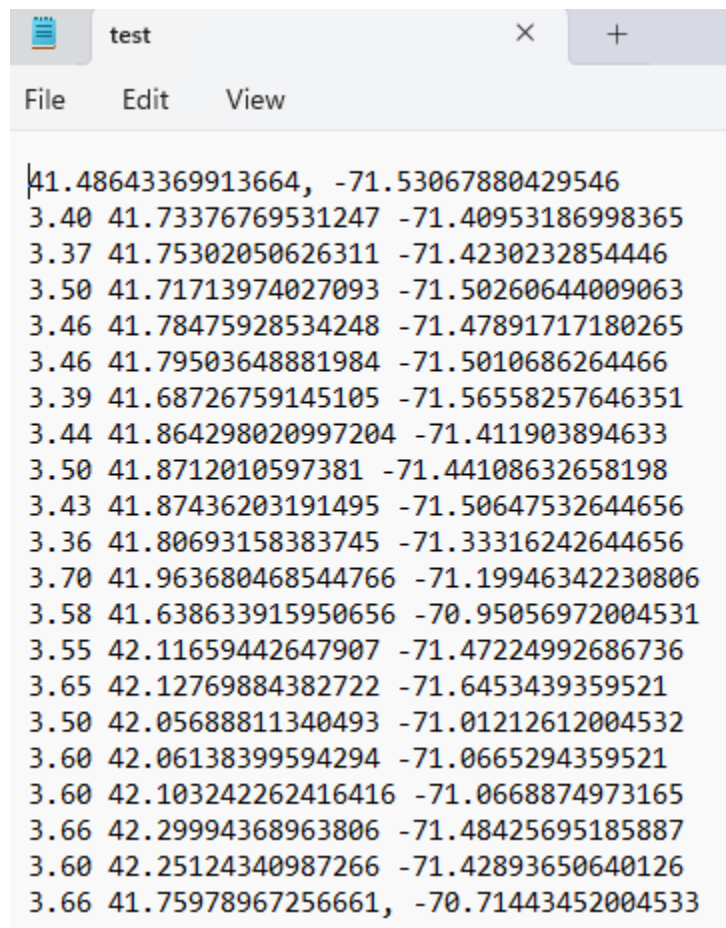
The write node function is a recursive function for writing data from every node in a binary tree to a dot file. The function will traverse the entire tree while writing the stringified nodes along the way. The function takes in a low and high integer for keeping track of the current node’s interval, a node that will be the current node, and the outfile streaming for the dot file being written to. First the current node will be stringified and the function will check if the current node’s left child exists. If that child exists, the left node will be stringified and the outfile will have the current node pointing to its left child node outputted to the dot file. Then the function will recursively call itself, but setting the current node to the current node’s left child and updating the low/high interval. The function also checks if the right child exists and does the same thing but instead works with the current node’s right child.

The get node string function is a simple function that just makes a string from the inputted node’s interval, sum gas price, and low/high gas prices. The function then returns the string containing that information.

4.2 NOTEWORTHY CODE AND PROOFS

Although our implementation program has a variety of functions each serving a unique and necessary purpose, we believe that there are three functions that are worth highlighting, and proving. These key functions are the get average function, the get low high function, and the insertion function. These functions are not only crucial to the program, but they are also crucial to the segment tree data structure. Because these functions are so important, it is imperative that they work consistently 100 percent of the time. Proving these functions will not only show how reliable the code is, but it will also give further insight into what specifically the function does.

All test inputs will be the following text file:



```
41.48643369913664, -71.53067880429546
3.40 41.73376769531247 -71.40953186998365
3.37 41.75302050626311 -71.4230232854446
3.50 41.71713974027093 -71.50260644009063
3.46 41.78475928534248 -71.47891717180265
3.46 41.79503648881984 -71.5010686264466
3.39 41.68726759145105 -71.56558257646351
3.44 41.864298020997204 -71.411903894633
3.50 41.8712010597381 -71.44108632658198
3.43 41.87436203191495 -71.50647532644656
3.36 41.80693158383745 -71.33316242644656
3.70 41.963680468544766 -71.19946342230806
3.58 41.638633915950656 -70.95056972004531
3.55 42.11659442647907 -71.47224992686736
3.65 42.12769884382722 -71.6453439359521
3.50 42.05688811340493 -71.01212612004532
3.60 42.06138399594294 -71.0665294359521
3.60 42.103242262416416 -71.0668874973165
3.66 42.29994368963806 -71.48425695185887
3.60 42.25124340987266 -71.42893650640126
3.66 41.75978967256661, -70.71443452004533
```

The program will produce a double vector that contains this information with this specific text file:

```
0 - $3.39 13.9932mi.  
1 - $3.5 16.0063mi.  
2 - $3.4 18.1993mi.  
3 - $3.37 19.2406mi.  
4 - $3.46 20.7851mi.  
5 - $3.46 21.3774mi.  
6 - $3.36 24.3799mi.  
7 - $3.44 26.8181mi.  
8 - $3.43 26.8326mi.  
9 - $3.5 26.9842mi.  
10 - $3.58 31.7809mi.  
11 - $3.7 37.1359mi.  
12 - $3.55 43.6443mi.  
13 - $3.65 44.6994mi.  
14 - $3.66 46.1961mi.  
15 - $3.6 46.3696mi.  
16 - $3.5 47.6189mi.  
17 - $3.6 48.8573mi.  
18 - $3.6 53.1025mi.  
19 - $3.66 56.2594mi.
```

First up is the get average function:

```
double seg_tree::getAvg(int currentNodeLow, int currentNodeHigh, int searchLow, int searchHigh, Node* currentNode){
    if (searchHigh < currentNodeLow || currentNodeHigh < searchLow || !currentNode){
        return 0;
    }

    if (searchLow <= currentNodeLow && currentNodeHigh <= searchHigh){
        return currentNode->gasPrice;
    }

    int mid = std::ceil((currentNodeHigh + currentNodeLow) / 2.0);

    return getAvg(currentNodeLow, currentNodeHigh: mid-1, searchLow, searchHigh, currentNode: currentNode->left)
        + getAvg(currentNodeLow: mid, currentNodeHigh, searchLow, searchHigh, currentNode: currentNode->right);
}
```

As mentioned before, this function is used for getting the average gas price in the search interval. The function will take in the current low/high and the search low/high as integer parameters and take in a node to keep track of the current node as a parameter. In order to make sure the current node is not out of bounds of the search interval, or if the current node does not exist, the program checks if these conditions are true with an `if` statement and returns zero and does not continue the function any further. This `if` statement makes sure that only relevant and existing nodes are worked with so the function outputs accurately and does not crash.

Next, the function checks if the current node's interval falls in between the search intervals and if so, returns the current node's sum gas price. This is so the function returns a gas price that is accurate to the specified search interval, and so the program does not have to do extra work as its child nodes would be irrelevant.

Lastly is the recursive part of the function, which first a mid point is calculated using the same method the mid point is calculated in the `gen tree` function, making sure the tree traversal is accurate. Then the function will recursively call itself but update the current low/high to the current node's left child's low/high and pass in the current node's left child as the current node.

The result of this function will then be added to another recursive call, but this time the parameters will hold data relevant to the current node's right child. The summation of these two will then be returned. This will always work as if an irrelevant node is passed through, then it will return zero, and if the node passed through is relevant, then it will either traverse the tree more or return its gas price. Because only gas prices from relevant node's are returned, the function will always work.

For a test input, I am going to input twenty seven miles as the high, and the low will be zero. If you manually calculate the average of the gas prices you will get 3.431. The program will output the following:

```
input the maximum distance: 27
the average gas price in this area is: $3.431
```

As you can see this calculated the average accurately.

Next I am going to get the average between a specified low and high distance. I will input 18.5 as the low distance and 25 as the high distance. If you calculate the average manually, you will get 3.4125. The program will output the following:

```
input the minimum distance: 18.5
input the maximum distance: 25
the average gas price in this area is: $3.4125
```

As you can see this calculated the average accurately.

Next up is the get low high function:


```

std::pair<double, double> seg_tree::getLowHigh(int currentNodeLow, int currentNodeHigh, int searchLow, int searchHigh,
Node *currentNode, std::pair<double, double> lowHigh) {
    if (searchHigh < currentNodeLow || currentNodeHigh < searchLow || !currentNode) {
        return lowHigh;
    }

    if (searchLow <= currentNodeLow && currentNodeHigh <= searchHigh){
        if (currentNode->lowPrice < lowHigh.first){
            lowHigh.first = currentNode->lowPrice;
        }
        if (currentNode->highPrice > lowHigh.second){
            lowHigh.second = currentNode->highPrice;
        }
        return lowHigh;
    }

    int mid = std::ceil( (currentNodeHigh + currentNodeLow) / 2.0);

    lowHigh = getLowHigh(currentNodeLow, currentNodeHigh: mid-1, searchLow, searchHigh, currentNode: currentNode->left, lowHigh);
    lowHigh = getLowHigh( currentNodeLow: mid, currentNodeHigh, searchLow, searchHigh, currentNode: currentNode->right, lowHigh);

    return lowHigh;
}

```

This function is quite similar to the get average function, but it has differences that set it apart. Its parameters are exactly the same as get average's with the exception of the pair of doubles called low high which will store the lowest and highest gas prices in a given interval. When called the function will perform the same check as get average but instead return low high, as the function does not get a sum. This check ensures only the right nodes are worked with leading to accurate output and avoids crashes.

Next up the function checks if the current node's interval falls in between the search's interval and will adjust the low of low high if the current node's low gas price is lower than low high's low, and will also adjust the high of low high if the current node's high gas price is higher than low high's high. This ensures that low high only has its values changed when the current node is relevant to the search interval, leading to an accurate output.

Next are some recursive calls which call themselves two times to get the low high of the current node's left and right children. This is necessary in order to traverse the tree to find every possible low and high gas price. After these recursions, low high is returned. Because of the checks and recursion, the function will always return an accurate low high of the inputted search interval.

For a test input, I am going to input twenty-seven miles as the high, and the low will be zero. If you manually find the lowest and highest gas prices, you will get \$3.36 as the low and \$3.50 as the high price. As you can see, the program outputted the lowest and highest gas prices accurately with the following:

```
input the maximum distance: 27  
  
the lowest gas price in this area is: $3.36  
  
the highest gas price in this area is: $3.5
```

Next, I am going to get the low and high gas prices between a specified low and high distance. I will input 18.5 as the low distance and 25 as the high distance. If you find the lowest and highest prices manually, you will get \$3.36 as the low and \$3.46 as the high. As you can see, the program outputted the lowest and highest gas prices accurately with the following:

```
input the minimum distance: 18.5  
  
input the maximum distance: 25  
  
the lowest gas price in this area is: $3.36  
  
the highest gas price in this area is: $3.46
```

Finally is the insertion function which can kind of be broken down into two parts, that do two separate things. The first part of the function updates the sum of gas prices:

```
void seg_tree::insert(int pos, int low, int high, double add, Node* currentNode){
    if (!currentNode){return;}

    currentNode->addGasPrice( add: add - inputVec[pos][0]);

    int mid = std::ceil( (low + high) / 2.0);

    if (low <= pos && mid-1 >= pos){
        insert(pos, low, high: mid-1, add, currentNode: currentNode->left);
    }
    else{
        insert(pos, low: mid, high, add, currentNode: currentNode->right);
    }
}
```

The second part of the function updates the low and high gas prices:

```
if (low == high){
    currentNode->lowPrice = add;
    currentNode->highPrice = add;
}
else {
    double tempLeftLow = currentNode->left->lowPrice;
    double tempLeftHigh = currentNode->left->highPrice;

    double tempRightLow = currentNode->right->lowPrice;
    double tempRightHigh = currentNode->right->highPrice;

    if (tempLeftLow < tempRightLow){
        currentNode->lowPrice = tempLeftLow;
    }
    else{
        currentNode->lowPrice = tempRightLow;
    }

    if (tempLeftHigh > tempRightHigh){
        currentNode->highPrice = tempLeftHigh;
    }
    else{
        currentNode->highPrice = tempRightHigh;
    }
}
return;
```

This function takes in an int parameter called position which is the index that is being updated, ints low and high which hold the data for the current node's interval, double add which is the updated gas price value at the position index, and a node current node to keep track of the current node being worked with. First the program checks if the current node does not exist and returns if this condition is true. This is to prevent crashes and to not waste time working with a node that does not exist.

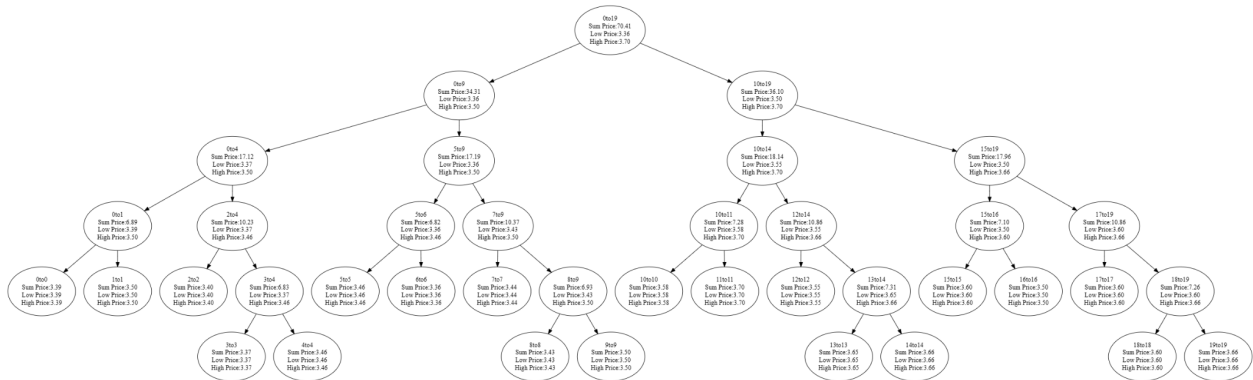
Next, the function updates the current node's gas sum price then recursively calls itself with parameters being relevant to either the current node's left child or its right child. The if statement will only call whichever child node's interval contains the position that is being updated. This ensures only relevant nodes are visited making the tree's nodes accurate to the update. This first part of the function will continue until reaching a leaf node at which point the backtracking part will begin to update the low and high prices.

The function will then check if its current node is a leaf node, and will update the current node's low and high price to the updated price. After the leaf node is updated, the new low and high gas prices can be determined the same way it was in the gen tree function, with each parent node setting its low and high price to the lowest and highest of their children's low and high prices.

If the current node is not a leaf node, then the current node's low price will be set to the lowest low price of its children and the current node's high price will be set to the highest high price of its children. Because the current node determines its low and high prices based on its children's low and high prices, each node will have its low and high prices be updated accurately. After the current node's low and high prices are updated, the function will return and continue backtracking until reaching the root node, at which point the segment tree is entirely updated and the insertion is complete.

Here is the dot file output when using the input file that is being used for example

input/output:

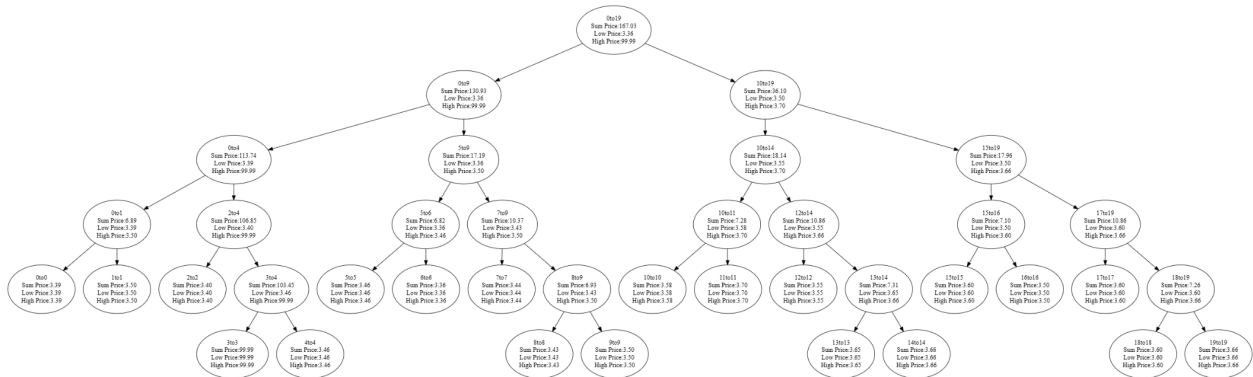


For a test input, I will have the program change the price of gas at index three to \$99.99. If you compare the outputted dot file visualization below, to the unaltered one above, you will see that only the correct nodes were updated with the correct information.

```

please input the index of which gas price you would like to update: 3
please input the new gas price: 99.99

input 'yes' to exit: yes
dot file has been written
  
```



5 CONCLUSION

Researching segment trees and their applications gave us a good starting point for understanding how segment trees work and where and when to use them. We gained a good understanding of how to approach functions like generating the segment tree, performing range queries on the segment tree, and updating an element in the segment tree. We also learned about the time complexity of these queries and insertions which further demonstrated to us how useful this data structure is. Even though it has some downsides which could make it a poor choice in specific situations, segment trees when used correctly can improve the performance of a program massively.

When programming our implementation, we learned much more about segment trees than what they can do on paper. With our implementation, we tried to take full advantage of the segment tree every step of the way, which taught us not only its applications in making programs, but how to efficiently and effectively solve real world problems. We saw how the logarithmic time complexity of queries to find average and low/high gas prices in a specific distance made our program run faster than an approach that would have a linear time complexity, which would make this program more useful to a user as it would not perform nearly as slowly with larger data sets. We believe that our implementation not only taught us about segment trees, it also taught us how to find the right real world application of data structures.

In terms of real world applications, segment trees have many different uses and functions. A good example of this real world usefulness is in our programming implementation, which would allow users to find out which gas stations they should go to and which gas stations to

avoid based on their distance and gas price. There are more real world applications for segment trees in specific fields such as computational geometry, geographic information systems, and image processing. Other than these specific use cases, segment trees can be used anywhere where range queries are made from a specific segment of a collection.

6 CONTRIBUTION TABLE

Name	Contributions
James McCaffrey	Worked on main.cpp, seg_tree.cpp/h, node.cpp/h Worked on presentation slides Worked on essay
Michael Gilkeson	Helped clean up main.cpp, the presentation slides, and the report
Shuichi Kameda	Cleaned console UI, Misc optimization in seg_tree.cpp, Set up repo and README.md, Worked on report
Evan Ung	Worked on essay

Works Cited

GeeksforGeeks. (2023). Applications Advantages and Disadvantages of Segment Tree.

GeeksforGeeks.

<https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-segment-tree/#>

GeeksforGeeks. (2023, May 8). *Binary search tree*. GeeksforGeeks.

<https://www.geeksforgeeks.org/binary-search-tree-data-structure/>

Ojha, S. (2021, December 10). SEGMENT TREES - Saurabh Ojha - Medium. *Medium*.

<https://medium.com/@ojhasaurabh2099/segment-trees-ccf461b73964>

Recursive Approach to Segment Trees. (n.d.).

<https://leetcode.com/articles/a-recursive-approach-to-segment-trees-range-sum-queries-lazy-propagation/>

Science, B. O. C., & Science, B. O. C. (2022). Segment Tree and its applications | Baeldung on Computer Science. *Baeldung on Computer Science*.

<https://www.baeldung.com/cs/segment-trees>

Segment tree. Segment Tree - Algorithms for Competitive Programming. (2023, June 23).

https://cp-algorithms.com/data_structures/segment_tree.html