Application of Trees in Range Query with Fenwick Tree and Segment Tree

Michel Fang 13518137
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13518137@std.stei.itb.ac.id

***Abstract*—Range queries are queries in which you are asked to gather information on the range asked, for example, the sum of a subarray in an array of numbers. There are naive approach to this simple problem, however these naive approach sometimes will not suffice, mostly due to time complexity. In this paper, we'll see how we can transform ranges of these sub-array to be nodes in a tree, then we can gain leverage of the natural structure of tree to process range queries effectively, reducing time complexity. We will take a look at two of the most well-known tree-based data structure in processing range queries, and the basic notion behind them.**

***Keywords*—fenwick tree, range queries, segment tree, time complexity.**

I. Introduction

Range queries come up very often in computer science, usually one kind of range queries is where we retrieve all records of some value between an upper or lower boundary in a database (list of records of employees biodata), and the other is where we are asked to answer a certain characteristic of a given range/subarray (minimum/maximum of a subarray, sum of a subarray, consecutive xor of a subarray). In this paper, we'll discuss the latter type of range queries exclusively.

There are naive approach in answering such range queries, such as iterating over the subarray asked and gathering the relevant information needed. These methods, more often than not, only suffice for one or few queries only, in multiple queries, these approach will take a toll on time complexity.

Usually when using trees to optimize range queries, we preprocess the array and then use the range as nodes in the tree, and the actual array as leaves in the tree, the tree itself is usually rooted, with the root representing the whole array. Storing these preprocessed information and the tree sacrifices space complexity in turn of time complexity, which more often than not, is a better approach than the naive one. In this paper, any code shown will be in C++.

II. Basic Theory

A. Graph

A graph is a representation of an object, consisting of two sets, vertices (often called nodes) set and edge set. The vertices is a nonempty set. Each edge has either one or two vertices associated with it, called its *endpoints* [1]. Formally,

$$G = (V, E)$$

where V is a nonempty set of vertices, and E is a set of edges (may be infinite).

In this paper, we will assume *simple* graphs, such graphs are graphs with no different edge connected to the same two vertex, and no edge connects a vertex to itself.

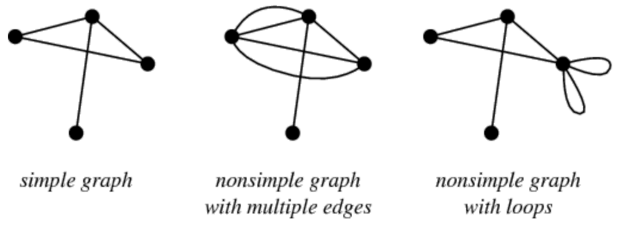

simple graph
nonsimple graph with multiple edges
nonsimple graph with loops

Figure 1. Simple graph and non simple graph [2]

Graphs can be directed and undirected. An undirected graph is a graph in which its edge doesn't have direction, the edges indicate a two-way relationship between its connected vertices. A directed graph is a graph in which its edge has a direction, the edges indicate a one-way relationship between its connected vertices [3].

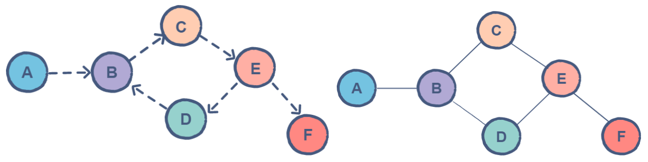


Figure 2. Directed graph (left) and undirected graph (right) [3]

B. Tree

A tree is a connected, *acyclic* graph that consists of n nodes and $n-1$ edges. Adding any distinct edge to the tree will make the tree have a cycle, and removing an edge will make the tree disconnected, dividing it into two sets of connected vertices [4].

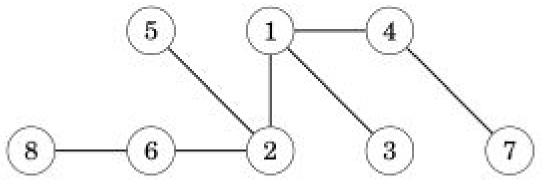


Figure 3. A tree with 8 vertices [4]

In a rooted tree, one of the nodes is appointed as the root of the tree, and all other nodes are placed underneath the root. In this tree, the children of a node are its lower neighbours, while the parent of a node is its upper neighbour. It is obvious that each node has exactly one parent, and that the leaves of a tree will not have any children. In the following figure, the tree is rooted at vertice 1, and its leaves are vertices 5, 8, and 7.

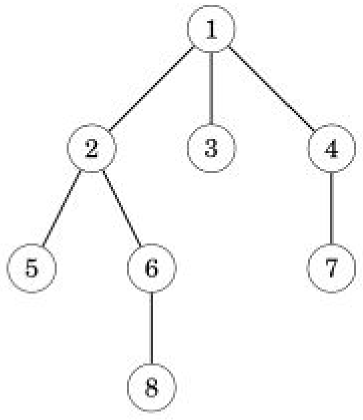


Figure 4. A rooted tree [4]

C. Time Complexity

The time complexity of an algorithm estimates how much time the algorithm will take for a given input. By calculating time complexity, we can often find out whether the algorithm is fast enough for solving a problem [4]. A time complexity is denoted by $O(...)$, where the three dots represent some function related to a given dimension, this notation is also known as the Big-O notation.

If a code consists of single commands, its time complexity is $O(1)$, these constant time operations are usually assignments, arithmetic operations, or comparisons.

The time complexity of a loop estimates the number of times the code inside the loop is executed. For example. the time complexity of the following C++ code snippet is $O(n)$, assuming the operation inside the loop is $O(1)$

```
for (int i = 0; i < n; i++) {
  /* Constant operation */
}
```

In general, a k nested for loops will have time complexity of $O(n^k)$.

By calculating time complexity, we can estimate how long our algorithm will run. We will start with an estimation; a modern processor can process 10^8 operation in a second. So if our algorithm runs in $O(n^2)$, and $n = 10^5$, then our algorithm will approximately run in 100 seconds. Such estimation is key in designing an algorithm with a constrained time limit. The following table lists some common algorithm complexity groups

Big-O	**Name**
$O(1)$	Constant
$O(log\ n)$	Logarithmic
$O(n)$	Linear
$O(n\ log\ n)$	n log n
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

Table 1. Common algorithm complexity groups

D. Binary Representation of Decimal Numbers

Binary numbers are written with only two symbols - 0 and 1. For example, $a = 1101$. Since symbols 0 and 1 are also a part of the decimal system and in fact of a positional system with any base, there's an ambiguity as to what 1101 actually stands for. We usually write the base explicitly to avoid confusion, like in $a = 1101_2$ or $b = 1101_{10}$ [5].

The decimal number uses powers of 10, the number b previously has a value of

$$b = 1 \cdot 10^3 + 1 \cdot 10^2 + 0 \cdot 10^1 + 1 \cdot 10^0$$
$$b = 1101$$

The binary number uses powers of 2, the number a previously has a value of

$$a = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$
$$a = 13$$

III. The Problem

A. Sum Query

On an array with n elements, we are tasked to calculate $sum_q(l, r)$, which is the sum of elements from index l to index r in an array. Consider the following array

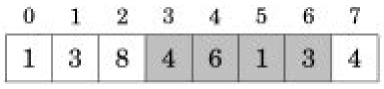


Figure 5. Our input array [4]

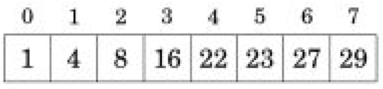
in this example, $sum_q(l, r) = 4 + 6 + 1 + 3 = 14$.

A simple approach is to iterate over l until r and sum up all the values of the array

```
int sum = 0
for (int i = l; i <= r; i++) {
  sum += array[i];
}
```

This approach works in $O(n)$ time, if we have q queries, then all queries will be processed in $O(qn)$ time, if n and q is large, then this approach will not suffice.

A better approach is to preprocess a prefix sum of the array, and then answer the query in $O(1)$



0	1	2	3	4	5	6	7
1	4	8	16	22	23	27	29

Figure 6. Prefix sum array [4]

Preprocessing the prefix sum array can be done in linear time

```
for (int i = 0; i < n; i++) {
  if (i == 0) {
      prefix[i] = array[i];
  } else {
    prefix[i] = prefix[i-1] + array[i];
  }
}
```

Then we can answer the query in $O(1)$ with inclusion-exclusion of the prefix sum array

$$sum_q(l,\ r) = prefix[r] - prefix[l-1]$$

with $prefix[l-1] = 0$ if l is 0

Assuming static queries; queries that do not alter the element of the array, this approach is much better. We have an $O(n)$ precomputation of the prefix array, then for q queries we answer in $O(q)$. If our queries involve updates however, then we need to rebuild our prefix sum array, this update of the prefix sum array will run in $O(n)$ time, reverting back to the slow $O(qn)$ time complexity.

B. Min/Max Query

Same problem as the previous one, except we need to find the minimum/maximum of elements in the range $[l,\ r]$.

The same simple approach as before can be applied here; instead of summing up the element over the range, we compare the current element with the best minimum element so far, the total time taken for q queries is still $O(qn)$.

```
int mn = array[l];
for (int i = l + 1; i <= r; i++) {
  mn = min(mn, array[i]);
}
```

Another approach is to preprocess all $[l,\ r]$ where $r - l + 1$ is a power of two [4]

l	r	$min_q(l,\ r)$
0	0	1
1	1	3
...	...	...
0	1	1
1	2	3
...	...	...
0	3	1
1	4	3
...	...	...

Table 2. Precomputed values for minimum query

Since there are $log\ n$ range lengths that are powers of two, the total time taken to compute these values is $O(n\ log\ n)$, all of these values can be computed with the following recurrence

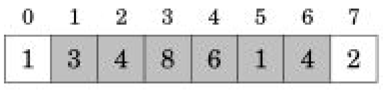
$$min_q(l,\ r) = min(min_q(l,\ l+m-1),\ min_q(a+m,\ r))$$

where m is $(r-l+1)/2$.

Answering the query can be done in $O(1)$ as a minimum of two precomputed values. Let k be the greatest power of two that does not exceed the length of $[l,\ r]$, then

$$min_q(l,\ r) = min(min_q(l,\ l+k-1),\ min_q(r-k+1,\ r))$$

For example, consider the query $min_q(1,\ 6)$



0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

Figure 7. Query over [1, 6] [4]

The largest power of two that doesn't exceed 6 is 4, then the range is the union of range [1,4] and [3,6]

$$min_q(1,\ 6) = min(min_q(1,\ 1+4-1),\ min_q(6-4+1,\ 6))$$

$$min_q(1,\ 6) = min(min_q(1,\ 4),\ min_q(3,\ 6))$$

$$min_q(1,\ 6) = min(3,\ 1)$$

$$min_q(1,\ 6) = 1$$

Same as the previous problem, this approach only works nicely if we are tasked with static queries, if the queries involve updates however, then we have to do the precomputation process again, which is expensive.

IV. Fenwick Tree

A. Theory

Fenwick Tree, also known as Binary Indexed Tree (BIT) is a data structure used to efficiently store information of an array, supporting a $O(log\ n)$ update, and a $O(log\ n)$ query. The implementation of a fenwick tree usually is in an array, whose size is the same as the input array we are processing. The basic idea of fenwick tree is the *responsibility* of each index of the array. In our previous input array, each index is responsible for the element at that index

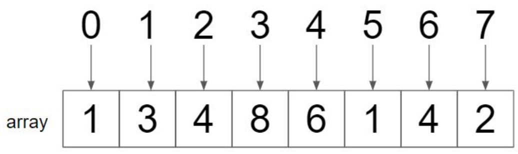


Figure 8. Index responsibility for an array

and in our prefix sum array, each index i is responsible for the array elements from index 0 to i

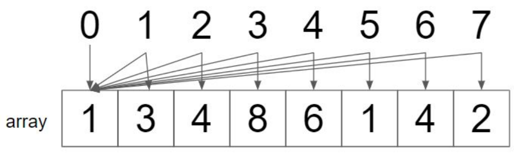


Figure 9. Index responsibility for prefix sum array

The range of responsibility of a fenwick array depends on the indexing convention, for an index i:

0-based indexing: consider the range for i+1

1-based indexing: consider the range for i itself

Suppose that the range of responsibility of an index i is computed by the function $p(i)$. The function $p(i)$ is the largest power of two that divides the considered index for i, let's see how we can compute this value with an example of $p(5)$

consider 6 instead of 5 (since 0 based indexing)

$$6 = 0110_2$$

$$\text{LSB}(0110) = 0010_2$$

$$\text{p}(5) = 0010_2 = 2$$

index 5 is then responsible for 2 elements (itself, and the elements before it), i.e. responsible for the range [4, 5]

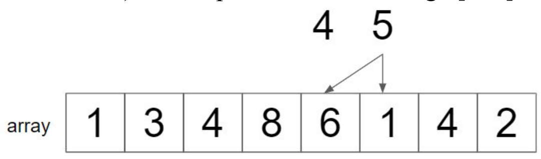


Figure 9. Index 5 responsibility in Fenwick Tree

We can also compute p(i) with the following bit manipulation

$$consider\ =\ i\ + 1(if\ 0\ based\ indexing)$$

$$consider\ =\ i\ (if\ 1\ based\ indexing)$$

$$p(i) = consider\ \&\ (-\ consider)$$

Formally, for any index i, it is responsible for the range $[i - p(i) + 1, i]$.

index$_{10}$	**binary considered**	**LSB**	**Range**
0	0001_2	0001_2=1_{10}	[0,0]
1	0010_2	0010_2=2_{10}	[0,1]
2	0011_2	0001_2=1_{10}	[2,2]
3	0100_2	0100_2=4_{10}	[0,3]
4	0101_2	0001_2=1_{10}	[4,4]
5	0110_2	0010_2=2_{10}	[4,5]
6	0111_2	0001_2=1_{10}	[6,6]
7	1000_2	1000_2=8_{10}	[0,7]

Table 3. Range of responsibility for indices of an array in Fenwick Tree

Now we can formally define the value of the fenwick tree array

$$tree[i] = sum_q(i - p(i) + 1, i)$$

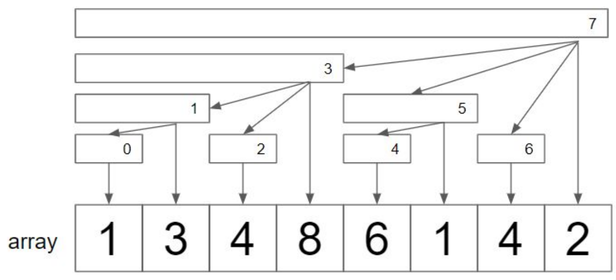


Figure 10. Fenwick Tree of an array

Notice that every index is responsible for k elements, where k is a power of two. Using a fenwick tree, any value of $sum_q(0,\ i)$ can be computed in $O(log\ n)$ time, since any range [0, i] can be divided into $log\ n$ intervals, whose sums are stored in the fenwick tree. For example, we want to compute $sum_q(0,\ 6)$

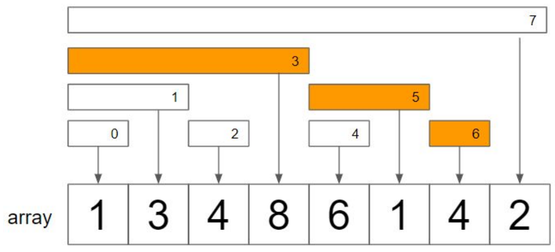


Figure 10. Answering query $sum_q(0,\ 6)$

The corresponding sum can be computed as follows

$$sum_q\ (0, 6)\ = sum_q\ (0, 3)\ + sum_q(4, 5)\ + sum_q\ (6, 6)$$

$$sum_q\ (0, 6)\ =\ 16 + 7 + 4 = 27$$

When updating an array value at index i, we also need to update every indice k where $tree[k]$ is responsible for the element in index i. Suppose we are updating the element at index 2, then the highlighted boxes are the values that need to be updated as well

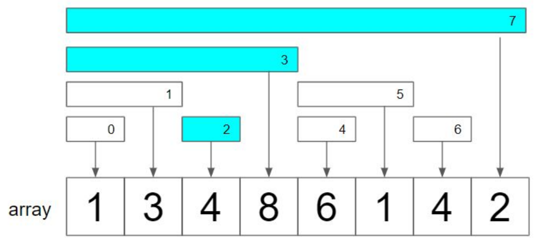


Figure 11. Updating array at index 2

$tree[2]$, $tree[3]$, and $tree[7]$ need to be updated since they cover $array[2]$. This operation takes $O(log\ n)$ time.

B. Implementation

We will allocate space for the fenwick tree array, whose size is the same as our input array

```
array[n];
tree[n];
```

Then, we will implement function $p(i)$ described previously

```
int p(int i) {
  int consider = i + 1; // if 0 based idx
  return (consider & -consider);
}
```

Now, we'll implement $sum_q(0, i)$

```
int sum(int i) {
  int ret = 0;
  while (i < n) {
    ret += tree[i];
    i += p(i);
  }
  return ret;
}
```

and how to update the value at index i

```
void add(int i, int val) {
//tree update after adding val to array[i]
  while (i < n) {
    tree[i] += val;
    i += p(i);
  }
}
```

To build the Fenwick Tree itself, we can fill the $tree$ array with zero, and pretend that we are updating each index i with the value $array[i]$

```
for (int i = 0; i < n; i++) {
  tree[i] = 0;
}

for (int i = 0; i < n; i++) {
  add(i, array[i]);
}
```

To answer the query $sum_q(l, r)$, we used the same inclusion-exclusion trick as before

```
int query(int l, int r) {
  int incl = sum(r);
  int excl;
  if (l == 0) excl = 0;
  else excl = sum(l - 1);
  return incl - excl;
}
```

The query takes $O(log\ n)$ time, the update takes $O(log\ n)$ time, and building the tree takes $O(n\ log\ n)$ time. This is a much better time complexity compared to the previous approach, other than that, we can answer queries with updates as well.

V. Segment Tree

A. Theory

A segment tree is a perfect binary tree whose nodes represent a range in an array, and the left child of a node represents the first half of the range, and the right child represents the second half of the range. The leaves of segment tree represent the actual array itself

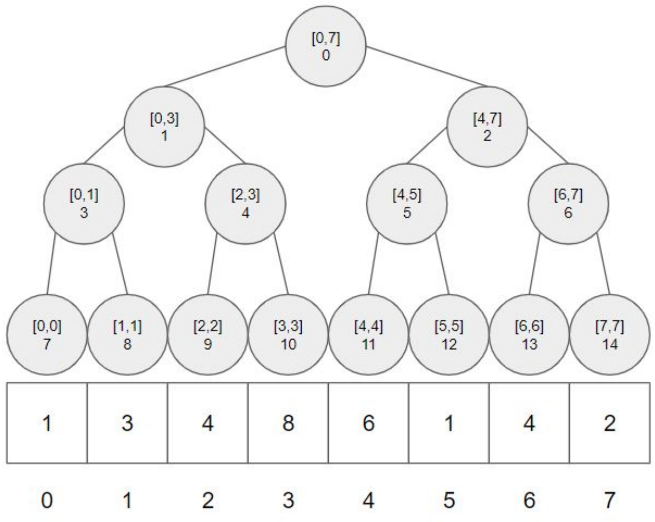


Figure 12. Segment tree of an array

The implementation of the segment tree is an array, whose size is $2n - 1$, if the size of the array is a power of two, and $4n - 1$ otherwise; since the tree is a perfect binary tree, we need to find the nearest power of two greater than the size of the array if it's not a perfect power of two, and build the tree from there, we will consider the extra values as garbage when we are dealing with updates and queries. In the figure, the number below the range is the index of the node in the array (e.g. range [0,7] corresponds to index 0, range [4,7] corresponds to index 2, etc).

When building the tree, we need to traverse the tree in postorder fashion (left-right-root) since a node's value is dependent on its children. The value of a node can be adjusted to meet our needs, it can be the minimum/maximum or the sum of its children. Since a node represents a range, then we can store these values in these nodes, and access these nodes for queries to come. Since the segment tree array is of size $2n - 1$ or $4n - 1$, and each node will be visited once, then building the tree will take $O(n)$ time.

To query a range $[ql, qr]$, we start at the root and do this procedure:

Let's denote $[l, r]$ the range of the current node

1. if $[l, r]$ is inside $[ql, qr]$, then simply return the value of the current node
2. if $[l, r]$ doesn't intersect with $[ql, qr]$, then we return a value that will be ignored
3. if $[l, r]$ partially intersects with $[ql, qr]$, then we recursively repeat this procedure for the left child and right child, and compute the value based on these results

Here is the visited state of the tree after we query range [2,4]

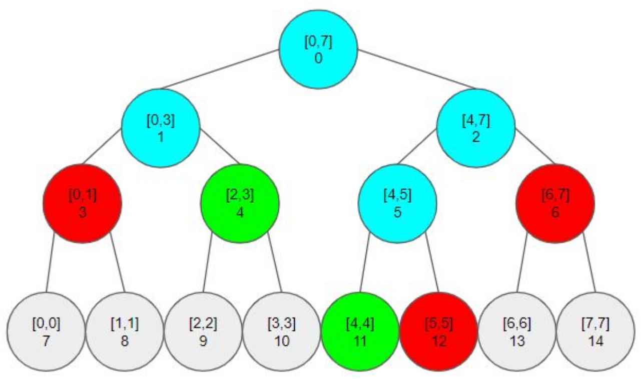


Figure 13. Visited state of the tree after query [2,4]

The green node's range is inside [2,4], the blue node's range partially intersects [2,4], the red node's range doesn't intersect with [2,4], and the grey node is unvisited during the search.

We return a garbage value when we visit a red node, this garbage value can be 0 if we're building a segment tree to answer a sum range query, since it will not contribute to the result, or an arbitrarily large/small value if we're building a segment tree to answer a minimum/maximum query (i.e. return a really big value if we're building a minimum query segment tree, and vice versa), since we will take the minimum/maximum respectively, this value will be ignored. When we visit a green or a blue node, we do the procedure mentioned before (see implementation for more details).

It can be proven that at each level, we only visit at most 4 nodes, since the level of the tree is $\log n$, the complexity of the query process is $O(4 \log n)$, which is the same as $O(\log n)$.

When we update an array at index i, we also need to update the segment tree, this update process is similar to the update process for fenwick tree; we update nodes that cover index i. Suppose we are updating the array at index 2, then the green nodes are the nodes that need to be updated

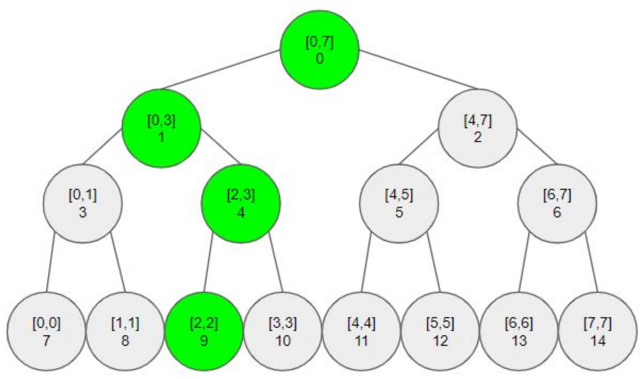


Figure 14. Visited state of the tree after updating index 2

We only visit one node per level in this update process, since the level of the tree is $\log n$, this update process takes $O(\log n)$ time.

B. Implementation

First, we need to allocate a segment tree array with an appropriate size, since in worst-case we'll have a segment tree array with $4n-1$ elements, we'll allocate that much in our program

```
int array[n];
int tree[4*n];
```

To build the tree, we will implement a recursive procedure. When we are at a leaf node, we simply assign the value of the array to the node, if not, we combine the value of the children to be the value of the node. We call the procedure `build(0, 0, n-1)` in our main program to build the entire tree.

```
void build(int v, int l, int r) {
  if (l == r) {
    tree[v] = array[l];
  } else {
    int m = (l + r) / 2;
    build(2*v + 1, l, m);
    build(2*v + 2, m + 1, r);
    tree[v]=combine(tree[2*v+1], tree[2*v+2]);
  }
}
```

The `combine` function above depends on what segment tree we are building

```
int combine(int val_left, int val_right) {
  // for sum query
  return val_left + val_right;
  // for min query
  return min(val_left, val_right);
  // for max query
  return max(val_left, val_right);
}
```

The query will be a recursive function as well. We will call the function `query(0, 0, n-1, ql, qr)` for a query [ql, qr]

```
int query(int v, int l, int r, int ql, int qr) {
    if (ql <= l && r <= qr) {
        return tree[v];
    } else if (qr < l || ql > r) {
        return garbage_value;
    } else {
        int m = (l + r) / 2;
        int val_left = query(2*v + 1,l,m,ql,qr);
           int val_right = query(2*v + 2,m+1,r,ql,
qr);
        return combine(val_left, val_right);
    }
}
```

The update procedure is similar to the build procedure

```
void update(int v, int l, int r, int idx, int val)
{
    if (l == r) {
        array[idx] += val;
        tree[v] += val;
    } else {
        int m = (l + r) / 2;
        if (l <= idx && idx <= m) {
        // idx lies in [l, m]
            update(2*v + 1,l,m,idx,val);
        } else {
        // idx lies in [m + 1, r]
            update(2*v + 2,m + 1,r,idx,val);
        }
        tree[v] = combine(tree[2*v+1],tree[2*v+2]);
    }
}
```

VI. Testing

We will compare the run time for each approach with multiple test cases generated with Mike Mirzayanov's test case generator, the queries in these test cases involve update queries, and the type of queries in these test cases are sum queries

N	Q	Naive	Preprocess	Fenwick	Segment
10	10	0.004 s	0.001 s	0.001 s	0.001 s
100	100	0.008 s	0.001 s	0.001 s	0.001 s
10000	10000	0.115 s	0.112 s	0.030 s	0.035 s
100000	300000	132.965 s	187.328 s	0.742 s	0.635 s

Table 4. Runtime comparison of different approaches

We can see that the runtime greatly differs when the input size is large. The naive and preprocess approach runs roughly in $O(qn)$, and the approach with query trees run roughly in $O(q \log n)$.

VII. Conclusion

We reduced time complexity for multiple range queries with the help of these query trees. These data structure can be further utilized and modified to fit our needs, for example we can find the Lowest Common Ancestor (LCA) of two nodes in a graph utilizing these query trees, one of LCA's applications in real life is in analysis of multiple species and their lowest common ancestor. This application of tree in solving a range query problem is just one of many applications of tree in computer science.

VIII. Acknowledgment

In this paper, the author thanks the Almighty God for His grace and guidance so that the author is able to complete this paper. The author also thanks Mrs. Harlili as a lecturer of Discrete Mathematics IF2120. The author also thanks his parents, his colleagues, and many other party related that has helped in the creation of this paper directly or indirectly. Lastly, the author apologizes for any mistakes in this paper.

References

[1] K. H. Rosen, *Discrete Mathematics and Its Applications*, 7th ed. New York: McGraw-Hill, 2012, pp. 641–802.

[2] Simple Graph. (n.d.). Retrieved December 5, 2019, from http://mathworld.wolfram.com/SimpleGraph.html.

[3] Directed Graphs vs. Undirected Graphs. (n.d.). Retrieved December 5, 2019, from https://www.educative.io/edpresso/directed-graphs-vs-undirected-graphs.

[4] A. Laaksonen, *Guide to Competitive Programming.* Switzerland: Springer, 2017, pp. 122–128 and pp. 131.

[5] A. Bogomolny. (n.d.). History of the Binary System. Retrieved December 3, 2019, from http://www.cut-the-knot.org/do_you_know/BinaryHistory.shtml.

[6] https://github.com/MikeMirzayanov/testlib

Statement

With this statement, I hereby declare that this thesis is a product of my own, not an adaptation, a translation from another person's work, nor formed by result of plagiarizing.

Bandung, 5 Desember 2019

[illegible]

Michel Fang 13518137