Wavelet Trees

Julian Steinmann

January 21, 2019

1 Concept

The idea behind a Wavelet Tree is to partition a sequence `X` into specific subsets and connect those to form a tree-shaped structure. Thanks to the properties of these subsets we are then able to answer various queries quickly. Our root node is the complete sequence `X`. As long as there are distinct values in a node `S`, the sequence is split into two sets `L` (left child of `S`) and `R` (right child of `S`). Denote the median value in `S` as `m(S)`. All values below `m(S)` in `S` will go to `L` while the others will go to `R`. The order of values is preserved, so when splitting for example $\texttt{N} = \{1, 4, 2, 3, 1\}$ the new nodes will be $\texttt{L} = \{1, 2, 1\}$ and $\texttt{R} = \{4, 3\}$. This process is done recursively until a leaf is reached. The set at the leaf contains per definition only one value (otherwise it'd be possible to split the set again).

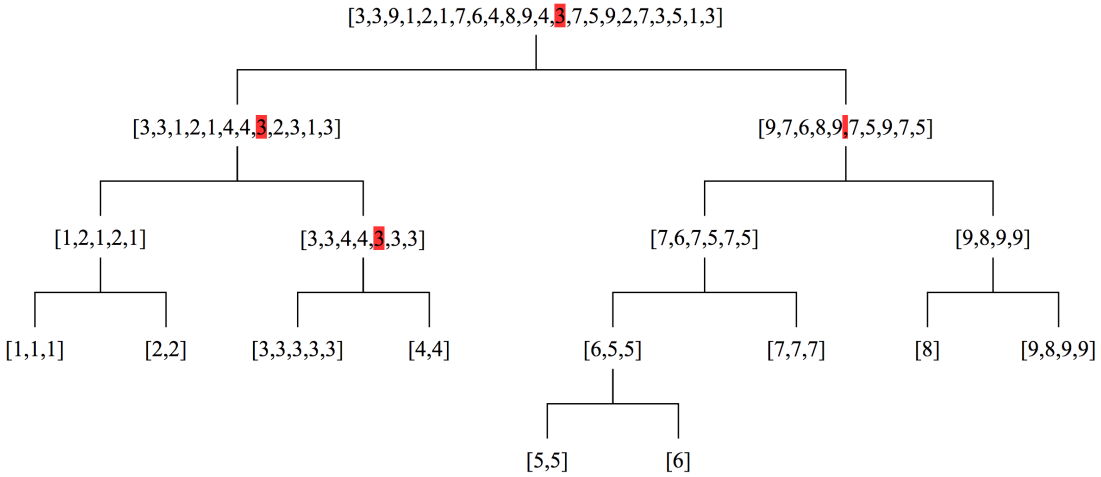

Figure 1: Wavelet Tree with the sequence $\texttt{X} = \{3, 3, 9, 1, 2, 1, 7, 6, 4, 8, 9, 4, 3, 7, 5, 9, 2, 7, 3, 5, 1, 3\}$

The alphabet of the tree is the set of all different values that the tree contains. It is assumed that the size of the alphabet is σ. The tree has a depth of $O(\log \sigma)$, because the alphabet is split into halves on every layer.

1.1 Construction

The construction of a Wavelet Tree is done recursively. For every node we first check if it is a leaf. If not, we process the elements of the node for the traversal (see subsection 1.2). Afterwards all elements are partitioned while preserving the order. Finally, the `build` function is called again twice: once with the elements bigger than $m(S)$ and once with the elements smaller than $m(S)$. The nodes of the tree are numbered from 1 to `n` such that the left child of node `i` is `2i` and the right child is `2i + 1`.

Implementation

```
vector<vector<int>> C;
int size;

WaveletTree(vector<int>& S, int sigma) : C(sigma*2), size(sigma){
    input = S;
    build(S.begin(),S.end(), 0, size-1,1);
}

void build(iter begin, iter end, int left, int right, int node){
    //-> leaf
    if(left == right)
        return;

    int mid = left+(right-left)/2;

    //here we will prepare the tree for the traversal, see next subsection

    iter pivot = stable_partition(begin, end, [=](int i){return i <= mid;});

    //call recursively for left and right side
    build(begin,pivot,left,mid,node*2);
    build(pivot,end,mid+1,right,node*2+1);
}
```

1.2 Traversal

The fundamental operation for traversing a Wavelet Tree is finding the indexes to which an index in the parent node is mapped to. Let's look at an example from Figure 1. The index 13 in the root node (marked in red) gets mapped to the index 8 in the left child and between index 5 and 6 in the right child. If a parent index is mapped between two child indexes, we will pick the first one (so index 5 in this case). We can define the functions `mapLeft(S,i)` and `mapRight(S,i)` which take a node `S` and an index `i` as an input and return the index in the left or right child respectively.

These are both elementary functions that will be used often, so it would be useful to be able to compute them as quickly as possible. Luckily, it is possible to do this in constant time.

Algorithm For quickly calculating `mapLeft` and `mapRight` we use prefix sums. An array C_S is used, which will hold the amount of elements that go to the left child up until a certain index. This is equivalent to checking if the element is bigger than $m(S)$ or not. Whenever a new node `S` is built, we iterate over all elements of the node and check whether they will end up in the left or right child. If the element at index `i` would end up in the left child, $C_S[i] = C_S[i-1] + 1$. Otherwise, $C_S[i] = C_S[i-1]$.

It is now possible to query `mapLeft` with the help of C_S. The number of elements that go the left up until index `i` is $C_S[i] - 1$ (because we start indexing from 0). Now `mapRight` can also be

computed, as the elements that go to the right are $i - C_S[i]$.

Running time Precomputing C with an original sequence of length N takes $O(N \log N)$ time, because we iterate once over all elements of N in every layer & there are $\log N$ layers. We can answer all mapLeft and mapRight queries in $O(1)$.

Implementation

```
C[node].reserve(end-begin+1);
C[node].push_back(0);
//this is a prefixsum, which allows us to get the number of elements
//that go to the left or right in O(1)
for(iter it = begin; it != end; ++it)
    C[node].push_back(C[node].back() + (*it<=mid));

//mapLeft(S,i) is now C[S][i]
```

2 Queries

2.1 Rank query

The rank operation counts the occurrences of a value x up until a certain index i. We can define the rank operation to be equal to

$$\text{rank}_x(S, i) = |\{k \in \{1, \ldots, x\} | S[i] = x\}|.$$

As an example, $\text{rank}_3(S, 14) = 3$ when taking the sequence from Figure 1.

Algorithm If $x \leq m(S)$ it is guaranteed that all occurrences of x in S appear in the left child of S. In this case

$$\text{rank}_x(S, i) = \text{rank}_x(\text{LeftChild}(S), \text{mapLeft}(S, i))$$

Similarly, if $x > m(S)$ then

$$\text{rank}_x(S, i) = \text{rank}_x(\text{RightChild}(S), \text{mapRight}(S, i))$$

This process can be applied recursively until a leaf is reached. If S is a leaf, $\text{rank}_x(S, i) = i$.[1]

Notice that it is possible to compute the occurrences of a value x between two indexes i and j through $\text{rank}_x(S, j) - \text{rank}_x(S, i - 1)$.

Running time The running time of the rank operation is $O(\log \sigma)$, as we traverse the tree down to a leaf and therefore call mapLeft or mapRight $O(\log \sigma)$ times. Computing the rank for a range between two indexes i and j has the same running time.

Implementation

```
//Count occurences of x until position i
//occurences between i and j: rank(x,j) - rank(x,i)
int rank(int x, int i) const{
    //open the interval on the left as it makes the processing easier
    ++i;
```

1. Rodrigo González et al., "Practical implementation of rank and select queries," 2005, 27–38.

```
    int left = 0, right = size-1, node = 1, mid;
    while(left != right){
        mid = left+(right-left)/2;

        if(x <= mid){
            i = C[node][i]; right = mid; node = 2*node;
        } else {
            i -= C[node][i]; left = mid+1; node = 2*node+1;
        }
    }
    return i;
}
```

2.2 Quantile query

$\texttt{quantile}_k(S, i, j)$ returns the k-th smallest element in the range $[i, j]$. Taking the sequence from Figure 1 again, $\texttt{quantile}_6(S, 7, 16) = 7$.

Algorithm For now, let $i = 1$, so an operation would look like $\texttt{quantile}_k(S, 1, j)$. This is a simpler problem, as we are now only searching for the k-th smallest element up until the index j.[2] Assume that $c = \texttt{mapLeft}(S, j)$. c is equal to the amount of elements that get mapped to the left child L up until index j. It is therefore guaranteed that if $k \leq c$, the element we are searching for is in the left subtree, so

$$\texttt{quantile}_k(S, 1, j) = \texttt{quantile}_k(\texttt{LeftChild}(S), 1, \texttt{mapLeft}(S, j))$$

If $k > c$, the element is in the right subtree and `quantile` can be computed with

$$\texttt{quantile}_k(S, 1, j) = \texttt{quantile}_{k-c}(\texttt{RightChild}(S), 1, \texttt{mapRight}(S, j))$$

This process can again be repeated until we reach a leaf. The answer is then the value stored in the leaf.

To generalize this process, we need to ignore all elements up until index i. In order to do this, we set $c = \texttt{mapLeft}(S, j) - \texttt{mapLeft}(S, i - 1)$. This means that c now only holds the number of elements that get mapped to the left between i and j. We also need to map i to the left or right, resulting in the following two equations:

$$\texttt{quantile}_k(S, i, j) = \texttt{quantile}_k(\texttt{LeftChild}(S), \texttt{mapLeft}(S, i - 1) + 1, \texttt{mapLeft}(S, j))$$

$$\texttt{quantile}_k(S, i, j) = \texttt{quantile}_{k-c}(\texttt{RightChild}(S), \texttt{mapRight}(S, i - 1) + 1, \texttt{mapRight}(S, j))$$

Running time The running time of the `quantile` operations is again $O(\log \sigma)$ and is only slightly bigger than the `rank` operation.

Implementation

```
//Find the k-th smallest element in [i,j]
int quantile(int k, int i, int j){
    int left = 0, right = size-1, node = 1, mid;
    while(left != right){
        mid = left+(right-left)/2;

```

2. Travis Gagie, Simon J. Puglisi, and Andrew Turpin, "Range Quantile Queries: Another Virtue of Wavelet Trees," arXiv: 0903.4726, *arXiv:0903.4726 [cs]* 5721 (2009): 1–6, accessed January 7, 2019, doi:10.1007/978-3-642-03784-9_1, http://arxiv.org/abs/0903.4726.

```
        if(k <= C[node][j]-C[node][i]){
            i = C[node][i]; j = C[node][j]; right = mid; node = 2*node;
        } else {
            k -= C[node][j]-C[node][i]; i -= C[node][i]; j -= C[node][j]; left = mid+1; node = 2*node+1;
        }
    }
    return right;
}
```

2.3 Range query

The `range` query returns the number of values that are between `x` and `y` and appear in the range `[i, j]`[3].

Algorithm The idea behind this algorithm can be visualized by displaying all entries in the sequence `X` in a grid, where the x-axis is the index and the y-axis the actual value. We are now looking for the amount of points within a rectangle that starts at `(i, x)` and ends at `(j, y)`.

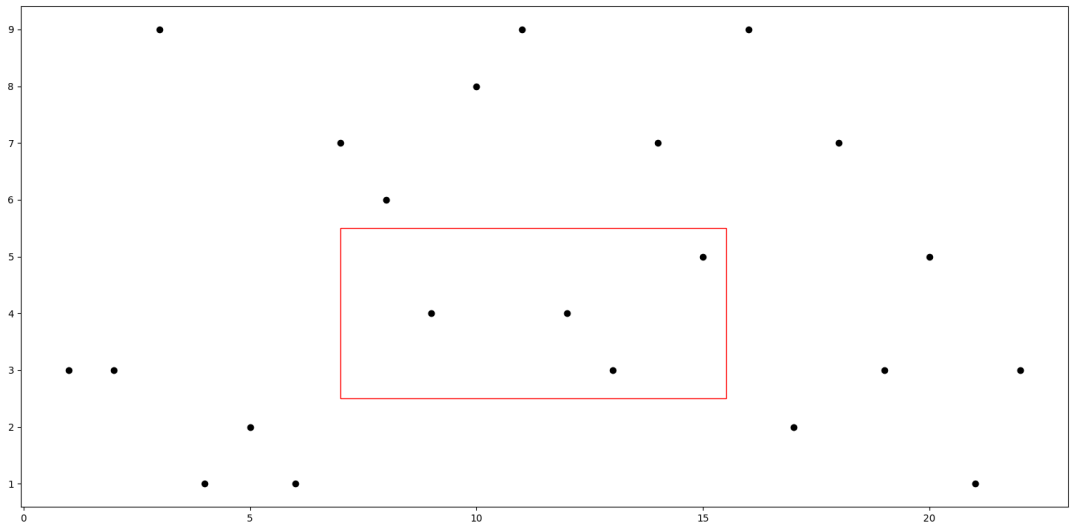

Figure 2: Illustration of the idea behind the range query algorithm

If we now consider an interval `[i, j]` where we want to search for the number of values betweeen `x` and `y`, there are three possible options:

- The interval does not intersect `[x, y]`. The returned value is therefore zero.
- The interval is completely contained in `[x, y]`. The returned value is $|[i, j]| = j - i + 1$ because all elements are included.
- The interval is partially contained in `[x, y]`. Now, the returned value is the sum of the range query of both children. We therefore need to call `range` with the same `x` and `y`, but `i` and `j` should be mapped to the children's indexes:

$$\begin{aligned} \texttt{range}_{x,y}(S, i, j) = \texttt{range}_{x,y}(\texttt{LeftChild}(S), \texttt{mapLeft}(S, i), \texttt{mapRight}(S, j)) \\ + \texttt{range}_{x,y}(\texttt{RightChild}(S), \texttt{mapRight}(S, i), \texttt{mapRight}(S, j)) \end{aligned} \tag{1}$$

3. Gonzalo Navarro, "Wavelet Trees for All," `https://users.dcc.uchile.cl/~gnavarro/ps/cpm12.pdf`.

These cases also apply when called on a leaf, so no additional code is necessary.

Running time It can be shown that the `range` operation runs in $O(\log \sigma)$.[4] The constant factors for this operation are higher than the ones for the `range` and `quantile` queries, but it is still possible to call `range` 10^6 times in one second if the alphabet has size 10^5.

Implementation

```
int originalLeft, originalRight;
//Count number of values in range [a,b] between positions [i,j]
int range(int i, int j, int a, int b){
    //empty range
    if(i > j || a > b){
        return 0;
    }
    originalLeft = i; originalRight = j;
    return range(i,j+1,0,size-1,1);
}

int range(int i, int j, int a, int b, int node){
    //ranges do not intersect
    if(originalLeft > b || originalRight < a){
        return 0;
    }
    //completely contained
    if(originalLeft <= a && b <= originalRight){
        return j-i;
    }
    //partially contained, call recursively
    int mid = (a+b)/2, mapLeft = C[node][i], mapRight = C[node][j];
    return range(mapLeft,mapRight,a,mid,node*2) +
           range(i-mapLeft,j-mapRight,mid+1,b,node*2+1);
}
```

3 Update queries

This document will only discuss two basic update queries. Please refer to Castro et al.[5] for more advanced update queries such as toggling elements.

3.1 Adding and deleting elements at the end of the sequence

The operations will be named identically to their array counterpart: `pushBack(S,x)` appends the element `x` at the end of the sequence `X` while `popBack(S)` removes the last element in the sequence.

Algorithm It is possible that performing these operations will alter the alphabet of the sequence `X`. To circumvent this problem, we will simply allow empty leafs and additionally build the tree with a size big enough to be able to contain any element `x` within the limits of the problem we

4. Travis Gagie, Gonzalo Navarro, and Simon J. Puglisi, "New Algorithms on Wavelet Trees and Applications to Information Retrieval," arXiv: 1011.4532, *arXiv:1011.4532 [cs]*, November 2010, accessed January 4, 2019, `http://arxiv.org/abs/1011.4532`.

5. Robinson Castro et al., "Wavelet Trees for Competitive Programming," *Olympiads in Informatics* 10, no. 1 (July 2016): 19–37, ISSN: 18227732, 23358955, accessed January 4, 2019, doi:`10.15388/ioi.2016.02`, `http://www.ioinformatics.org/oi/pdf/v10_2016_19_37.pdf`.

are solving. `pushBack(S,x)` needs to do two things: First, actually add the element `x` to the end of the sequence `X`. Second, we must add new entries in `C` such that it is possible to map the new index down through the tree.

If `x <= m(S)`, it follows that `mapLeft(S,i)` = `mapLeft(S,i-1)+1`. This automatically also means that `mapRight(S,i)` = `mapRight(S,i-1)`.

If `x > m(S)`, we know that `mapLeft(S,i)` = `mapLeft(S,i-1)` and `mapRight(S,i)` = `mapRight(S,i-1)+1`.

This process is now repeated recursively until we reach a leaf. For `popBack(S)`, a similar approach can be used. Instead of adding new elements to `C`, we simply pop the last element and traverse down the tree until a leaf is reached.

Running time Both `pushBack(S,i)` and `popBack(S)` run in $O(\log \sigma)$ time - we traverse down the tree with height $\log \sigma$ and delete/add a new element, which can be done in amortized constant time.

Implementation

```
void pushBack(int x){
    input.push_back(x);
    int left = 0, right = size-1, node = 1, mid;
    while(left != right){
        mid = left+(right-left)/2;

        C[node].push_back(C[node].back() + (x<=mid));
        if(x <= mid){
            right = mid;
        } else {
            left = mid+1;
        }
        node = 2 * node + (x > mid);
    }
}
void popBack(){
    int x = input.back();
    input.pop_back();
    int left = 0, right = size-1, node = 1, mid;
    while(left != right){
        mid = left+(right-left)/2;

        C[node].pop_back();
        if(x <= mid){
            right = mid;
        } else {
            left = mid+1;
        }
        node = 2 * node + (x > mid);
    }
}
```

References

Castro, Robinson, Nico Lehmann, Bernardo Subercaseaux, and Jorge Perez. "Wavelet Trees for Competitive Programming." *Olympiads in Informatics* 10, no. 1 (July 2016): 19–37. ISSN: 18227732, 23358955, accessed January 4, 2019. doi:10.15388/ioi.2016.02. http://www.ioinformatics.org/oi/pdf/v10_2016_19_37.pdf.

Gagie, Travis, Gonzalo Navarro, and Simon J. Puglisi. "New Algorithms on Wavelet Trees and Applications to Information Retrieval." ArXiv: 1011.4532, *arXiv:1011.4532 [cs]*, November 2010. Accessed January 4, 2019. http://arxiv.org/abs/1011.4532.

Gagie, Travis, Simon J. Puglisi, and Andrew Turpin. "Range Quantile Queries: Another Virtue of Wavelet Trees." ArXiv: 0903.4726, *arXiv:0903.4726 [cs]* 5721 (2009): 1–6. Accessed January 7, 2019. doi:10.1007/978-3-642-03784-9_1. http://arxiv.org/abs/0903.4726.

González, Rodrigo, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. "Practical implementation of rank and select queries," 2005, 27–38.

Navarro, Gonzalo. "Wavelet Trees for All." https://users.dcc.uchile.cl/~gnavarro/ps/cpm12.pdf.