Olympiads in Informatics, 2015, Vol. 9, 39–44
DOI: http://dx.doi.org/10.15388/ioi.2015.04


# Efficient Range Minimum Queries using Binary Indexed Trees

Mircea DIMA[1], Rodica CETERCHI[2]

[1] *Hickery, Martir Closca st., 600206 Bacau, Romania*
[2] *University of Bucharest, Faculty of Mathematics and Computer Science*
*14 Academiei st., 010014 Bucharest, Romania*
*e-mail: mircea@hickery.net, rceterchi@gmail.com*

**Abstract.** We present new results on Binary Indexed Trees in order to efficiently solve Range Minimum Queries. We introduce a way of using the Binary Indexed Trees so that we can answer different types of queries, e.g. the range minimum query, in $O(\log N)$ time complexity per operation, outperforming in speed similar data structures like Segment/Range Trees or the Sparse Table Algorithm.

**Keywords**: binary indexed tree (BIT), least significant non-zero bit (LSB), range minimum query (RMQ).

## 1. Introduction

The Binary Indexed Tree, introduced by Peter M. Fenwick in (Fenwick, 1994), is a data structure that maintains a sequence of elements (e.g. numbers) and is capable of computing the cumulative sum of consecutive elements, between any two given indexes, in time complexity $O(\log N)$ and also update the value at a given index.

We show how to use the structure of the Binary Indexed Tree so that it will support other types of operations besides summation, e.g. range minimum query, maintaining the same time complexity of $O(\log N)$.

## 2. Binary Indexed Trees

### 2.1. *Problem Presentation*

Consider an array $A$ indexed from $1$ with $N$ integers and the following types of operations:

1. Update – change the value at an index $i$, (e.g. $A[i] = v$).
2. Query – find the value of min(A[$i$], A[$i+1$], ... , A[$j$]), for $1 \le i \le j \le$ N.

The Binary Indexed Tree, as presented by Peter Fenwick, cannot efficiently answer these kinds of queries, because, for determining the sum of $A[i \ldots j]$, it needs to compute the difference between the sum of the first $j$ elements and the sum of the first $i-1$ elements.

## 2.2. *Defining the BIT*

A BIT is not a Binary Tree, the name "Binary Indexed" comes from the fact that the nodes are indexed from 1 to N with labels written in binary, and it uses this binary representation to define the parent node for each node.

BITs are in fact the binomial trees of (Cormen *et al.*, 1990). We construct them inductively, starting with $B_0$, a tree with a single node. We will construct two varieties, the *left* and the *right* binomial tree. The *left binomial tree* $B_{k+1}$ is obtained from two copies of left binomial trees $B_k$, by attaching the first of them as the leftmost child of the root of the second one. The *right binomial tree* is obtained in a mirror-like fashion, by attaching the second $B_k$ as a right child of the root of the first $B_k$.

Starting from the array $A$, from its set of indexes, we build a left binomial tree BIT1 (Fig. 2.1) and a right binomial tree BIT2 (Fig. 2.2).

The binomial tree $B_k$ (either left or right) has precisely $2^k$ nodes and height $k$. If we write the array indexes in binary, in the left binomial tree BIT1 we have $parent(i) = i + 2^{LSB(i)}$, and in the right binomial tree BIT2 we have $parent(i) = i - 2^{LSB(i)}$. This enables us to climb up either tree in $O(\log N)$.

Each node will keep aggregated data for all the nodes in its subtree. For instance in the first tree (Fig. 2.1) node 12 keeps the minimum value of nodes 9, 10, 11 and 12 which is the subarray $A[9 \ldots 12]$. Similarly, in the second tree (Fig. 2.2) node 12 keeps the minimum value the subarray $A[12 \ldots 15]$.

Since the parent of a node can be computed with a formula, we can store the trees in two arrays:

1. BIT1$[i]$ = minimum value of subarray $A[i - 2^{LSB(i)} + 1, i]$.
2. BIT2$[i]$ = minimum value of subarray $A[i, i + 2^{LSB(i)} - 1]$.

Computing these two arrays can be done in $O(N)$ with a bottom-up algorithm.

Let us consider the following array $A$ with 15 positive integers:

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A_i$ | 1 | 0 | 2 | 1 | 1 | 3 | 0 | 4 | 2 | 5 | 2 | 2 | 3 | 1 | 0 |

You can see below how we computed the data stored in node 12:

$$\text{BIT1}[12] = \min(\text{A}[9], \text{A}[10], \text{A}[11], \text{A}[12]) = \min(2, 5, 2, 2) = 2$$
$$\text{BIT2}[12] = \min(\text{A}[12], \text{A}[13], \text{A}[14], \text{A}[15]) = \min(2, 3, 1, 0) = 0$$

In the following figures Fig. 1 and Fig. 2, the number inside a node is the index associated with that node. The number below the node is the aggregated minimum value of its subtree.

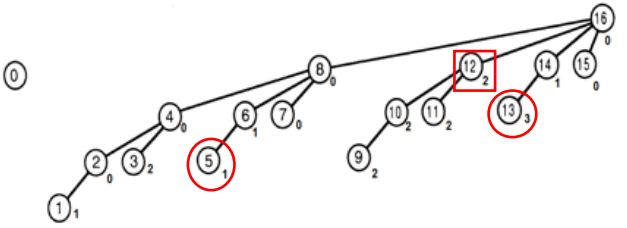



Fig. 2.1 Binomial Tree corresponding to BIT1 (node 16 is fictive) (Fenwick, 1994).

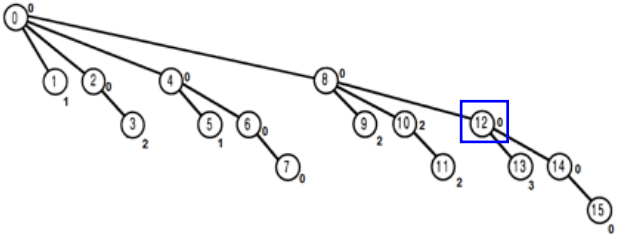



Fig. 2.2 Binomial Tree corresponding to BIT2 (node 0 is fictive) (Fenwick, 1994).

We shall exemplify the Least Significant Bit for better understanding:
LSB(216) = LSB(11011000) = 00001000 = 8 because there are 3 zeros at the end.

## 2.3. *Query operation*

For two given indexes $i$ and $j$ of the array $1 \le i \le j \le N$, we want to answer the question: What is the minimum value among $A[i], A[i+1], \ldots, A[j]$?

We start from node $i$ in the first tree (Fig. 2.1) and climb the tree through its parent as long as the node index is less than or equal to $j$. We do the same thing in the second tree starting from node $j$ and climbing the tree through the parent. In both cases we reach the same node and it splits $A[i \ldots j]$ in subarrays that are found either in BIT1, BIT2 or the value of the common stop node.

Let us exemplify by doing the query operation for the subarray $A[5 \ldots 13]$.

We start from node 5 and we climb the first tree (Fig. 2.1) while the current node's index is less than or equal to 13. We stop at node 8 because the next node, the parent of 8, is 16 which is greater than 13 and contains in its subtree the nodes 14, 15 and 16 which are not included in our subarray $A[5 \ldots 13]$. So far we passed by the nodes 5, 6 and 8. We take the minimum values corresponding to nodes 5 and 6 from the second tree found in BIT2. Looking in Fig. 2.2, node 5 keeps the minimum value for $A[5]$ and node 6 keeps the minimum value for $A[6 \ldots 7]$.

Similarly, we start from node 13 and climb the second tree (Fig. 2.2), passing by nodes 13, 12 and 8. We take the minimum values corresponding to nodes 13 and 12 from the first tree. Looking in Fig. 2.1, node 12 keeps the minimum value for $A[9 \ldots 12]$ and node 13 keeps the minimum value for $A[13]$.

We can observe that $A[5 \ldots 13]$ is now partitioned in the following subarrays: $A[5 \ldots 5]$, $A[6 \ldots 7]$, $A[8]$, $A[9 \ldots 12]$, $A[13]$.

An important thing is that we get to the same node $8$ for both traversals. We prove this happens every time:

Consider the subarray $A[i \ldots j]$ we want to make the query on. We know that $i < j$ and, because the order on integers is the same as the lexicographic order on their binary representation, we can write the indexes in binary like this (we consider that the indexes can be represented with $n$ bits and $p+1$ is the first bit on which $i$ and $j$ differ):

$i = c_1 c_2 \ldots c_p 0\, i_{p+2} \ldots i_n$
$j = c_1 c_2 \ldots c_p 1\, j_{p+2} \ldots j_n$

When we iteratively add $2^{LSB}$ to $i$ we will get at some point to $k = c_1 c_2 \ldots c_p 10 \ldots 0$ and if we iteratively subtract $2^{LSB}$ from $j$ we will get to the same $k$. This is the common node where we stop.

Because the query climbs the two trees by following the parent link and because the height of a binomial tree with $2^K$ nodes is $K$, the time complexity of the query operation for a subarray is $O(\log N)$ where $N$ is the size of the subarray.

## 2.4. *Update Operation*

Suppose we need to update the array at index $p$ with the value $v$ ($A[p] = v$).

We have to update all the tree nodes that have $p$ in their subtree. We start from node $p$ in the first tree (Fig. 2.1) and climb the tree until we reach the root (an index greater than $N$). For each node $i$ we pass by, we consider its associated interval that defines its subtree: $[i - 2^{LSB(i)} + 1, i]$ (e.g. [9,12] is the associated interval of node 12). We can observe that the generated intervals include the index $p$ because the parent's subtree expands and includes the node's subtree.

We want to update the minimum value of the associated interval of a node, be it $[x, y]$, where $y = x + 2^{LSB(x)} - 1$. If the minimum value of that interval is at an index $q$, $x \le q \le y$, different from $p$, then we update the interval by taking the minimum value between $v$ and $A[q]$. If the minimum value is at index $p$, then we have to take the minimum values of intervals $[x, p-1]$ and $[p+1, y]$.

If we compute the minimum values using two queries, the time complexity of the update will be $O(\log^2 N)$.

We make the following observation: when we generate the associated intervals of the nodes we pass by, we can cover the whole interval $[p+1, y]$ by starting from node $p+1$ and climbing the first tree (Fig. 2.1). So instead of doing a query for every node we update, we compute the results of the queries on the fly by climbing the tree once.

Analogously, we can update all the intervals of the form $[x, p-1]$ by starting from node $p-1$ and climbing the second tree (Fig. 2.2). The same algorithm is applied for updating both trees.

Since we are climbing each tree three times and the height of a binomial tree with $2^K$ nodes is $K$, the amortized time complexity of the Update operation is $O(\log N)$.

### 2.5. *Experiments and Results*

We wanted to find out how the Binary Indexed Tree compares to a similar data structure called Segment Tree (also known as Range Tree), since it supports both update and query operations in the same time complexity of $O(\log N)$.

We implemented these two data structures in C++ and ran them on a 3.5 GHz Intel Xeon-Haswell server with 8 GB of RAM on Ubuntu 14 operating system with gcc 4.8 compiler.

The initial array had 100K random integers and we ran 10M random updates and 10M random queries. In Table 1 is what we found (times are in seconds):

While there is not a big difference on the Build step, we see a 47% reduced time for updates and 77% reduced time on queries.

## 3. Conclusions

In the current paper we intended to adapt the Binary Indexed Tree so that we can solve different types of operations, using as an example the Range Minimum Query problem, and maintaining the original time complexity of $O(\log N)$. The RMQ can be solved using a Segment Tree or other data structures like quadtree, but the Binary Indexed Tree proved to be 2–4 times faster in practice due to its simple iterative implementation.

Due to the structure of the Binary Indexed Tree, it can be extended in multithreading and distributed environments obtaining $O(\log(\log N))$ time complexities per operations (Elhabashy *et al.*, 2009). Also the data can be distributed among multiple nodes. This data structure can be used as indexes for databases in a distributed manner.

In conclusion, the Binary Indexed Tree has the following advantages:

- Is faster than other data structures that allow the same types of operations.
- Can be adapted for a large number of distinct operations: sum, minimum, maximum, greatest common divisor (gcd), greatest common factor (gcf), etc.
- Can be extended on multi-core and distributed platforms.

## Acknowledgements

We thank dr. Florin Manea from the University of Bucharest and the University of Kiel for fruitful discussions and insightful comments on the topic of this paper.

Table 1

| Operation Type | Segment(Range) Tree time | Binary Indexed Tree time |
|---|---|---|
| Build 100K array | 0.0009 s | 0.0006 s |
| 10M Updates | 1.274 s | 0.672 s |
| 10M Queries | 2.397 s | 0.551 s |

## References

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. (1990). *Introduction to Algorithms*. MIT Press, McGraw-Hill, 1st edition.

Demaine, E., Sen, S., Lindy, J. *Advanced Data Structures*. Massachusetts Institute of Technology 6.897.

Elhabashy, A., Mohamed, A., Mohamad, A. (2009). An enhanced distributed system to improve thetime complexity of binary indexed trees. *World Academy of Science, Engineering and Technology*, 3(6), 121–126. http://waset.org/publications/5410/an-enhanced-distributed-system-to-improve-thetime-complexity-of-binary-indexed-trees

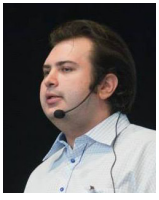
van Emde Boas, P., Kaas, R., Zijlstra, E. (1977 ). *D*esign and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10, 99–127.

Fenwick, P.M. (1994). A new data structure for cumulative frequency table, Software-Practice and Experience, 24(3), 327–336.

Fischer, J., Heunn, V. (2006). Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In: *CPM'06 Proceedings of the 17th Annual conference on Combinatorial Pattern Matching. Heidelberg*, Springer-Verlag Berlin, 36-48.

Topcoder Inc. (2014a). *Binary Indexed Trees*. https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/

Topcoder Inc. (2014b). *Range Minimum Query and Lowest Common Ancestor*. https://www.topcoder.com/community/data-science/data-science-tutorials/range-minimum-query-and-lowest-common-ancestor/



**M. Dima** (1989) – Co-Founder of Hickery.net, Senior Software Engineer, Former Software Engineer Intern at Facebook, Invited Host Scientific Committee Member at IOI 2013 Australia, problem setter at Romanian Olympiads, Programming Contest Veteran

**R. Ceterchi** (1953) – Associate Professor at the Faculty of Mathematics and Computer Science, University of Bucharest specialized on Algorithms and Data Structures, author of over 40 papers in national and international journals.