Binary Indexed (Fenwick) Tree

1. Introduction

A Binary Indexed (Fenwick) Tree is a data structure that provides efficient methods for implementing **dynamic cumulative frequency tables**.

This Fenwick Tree data structure uses many bit manipulation techniques. In this visualization, we will refer to this data structure using the term Fenwick Tree as the abbreviation 'BIT' of Binary Indexed Tree is usually associated with the usual bit manipulation.

1-1. Cumulative Frequency Table

Suppose that we have a *multiset* of *integers* **s = {2,4,5,6,5,6,8,6,7,9,7}** (not necessarily sorted). There are **m = 11** elements in **s**. Also suppose that the largest integer that we will ever use is **n = 10** and we never use integer 0. For example, these integers represent student (integer) scores from [1..10]. Notice that **m** is independent of **n**.

We can create a frequency table **f** from **s** with a trivial **O(m)** time loop. We can then create cumulative frequency table **cf** from frequency table **f** in **O(n)** time using technique similar to DP 1D prefix sum.

Index/Score/Symbol	Frequency f	Cumulative Frequency cf
0	-	- (index 0 is ignored)
1	0	0
2	1	1
3	0	1
4	1	2
5	2	4 == **cf[4]+f[5]**
6	3	7 == **cf[5]+f[6]**
7	2	9
8	1	10
9	1	11
10 == **n**	0	11 == **m**

1-2. Range Sum Query: rsq(i, j)

With such cumulative frequency table **cf**, we can perform **Range Sum Query: rsq(i, j)** to return the sum of frequencies between index **i** and **j** (inclusive), in

efficient **O(1)** time, again using the DP 1D prefix sum (i.e., the inclusion-exclusion principle). For example, **rsq(5, 9) = rsq(1, 9) - rsq(1, 4) = 11-2 = 9**.

Index/Score/Symbol	Frequency f	Cumulative Frequency cf
0	-	- (index 0 is ignored)
1	0	0
2	1	1
3	0	1
4	1	2 == **rsq(1, 4)**
5	2	4
6	3	7
7	2	9
8	1	10
9	1	11 == **rsq(1, 9)**
10 == **n**	0	11 == **m**

1-3. Dynamic Cumulative Frequency Table

A dynamic data structure need to support (frequent) updates in between queries. For example, we may update (add) the frequency of score **7** from 2 → 5 and update (subtract) the frequency of score **9** from 1 → 0, thereby updating the table into:

Index/Score/Symbol	Frequency f	Cumulative Frequency cf
0	-	- (index 0 is ignored)
1	0	0
2	1	1
3	0	1
4	1	2
5	2	4
6	3	7
7	2 → 5	9 → 12
8	1	10 → 13
9	1 → 0	11 → 13
10 == **n**	0	11 → 13 == **m**

A pure array based data structure will need **O(n)** per update operation. Can we do better?

2. Modes and the First/Default Mode

Introducing: Fenwick Tree data structure.

There are three mode of usages of Fenwick Tree in this visualization.

The first mode is the default Fenwick Tree that can handle both **Point Update (PU)** and **Range Query (RQ)** in O(log **n**) where **n** is the largest index/key in the data structure. Remember that the actual number of keys in the data structure is denoted by another variable **m**. We abbreviate this default type as **PU RQ** that simply stands for **Point Update Range Query**.

This clever arrangement of integer keys idea is the one that originally appears in Peter M. Fenwick's 1994 paper.

3. Point Update Range Query (PU RQ)

You can click the '**Create**' menu to create a frequency array **f** where **f[i]** denotes the frequency of appearance of key **i** in our original array of keys **s**.

IMPORTANT: This frequency array **f** is not the original array of keys **s**. For example, if you enter {0,1,0,1,2,3,2,1,1,0}, it means that you are creating 0 one, 1 two, 0 three, 1 four, ..., 0 ten (1-based indexing). The largest index/integer key is **n = 10** in this example as in the earlier slides.

If you have the original array **s** of **m** elements, e.g., {2,4,5,6,5,6,8,6,7,9,7} from earlier slides (**s** does not need to be necessarily sorted), you can do an O(**m**) pass to convert **s** into frequency table **f** of **n** indices/integer keys. (*We will provide this alternative input method in the near future*).

You can click the '**Randomize**' button to generate random frequencies.

3-1. The Visualization - Part 1

Although conceptually this data structure is a **tree**, it will be implemented as an integer **array** called **ft** that ranges from index 1 to index **n** (we sacrifice index 0 of our **ft** array). The values inside the vertices of the Fenwick Tree shown above are the values stored in the 1-based Fenwick Tree **ft** array.

3-2. The Visualization - Part 2

The values inside the vertices at the bottom are the values of the data (the frequency array **f**).

3-3. The Visualization - Part 3

The value stored in index **i** in array **ft**, i.e., **ft[i]** is the cumulative frequency of keys in range **[i-LSOne(i)+1 .. i]**. Visually, this range is shown by the edges of the Fenwick Tree. For details of **LSOne(i)** operation, see our bitmask visualization page.

3-4. Range Query: rsq(j)

The function **rsq(j)** returns the cumulative frequencies from the first index 1 (ignoring index 0) to index **j**.

This value is the sum of sub-frequencies stored in array **ft** with indices related to **j** via this formula **j' = j-LSOne(j)**. This relationship forms a Fenwick Tree, specifically, the 'interrogation tree' of Fenwick Tree.

We apply this formula iteratively until **j** is 0. *(We will add that dummy vertex 0 later)*.

Discussion: Do you understand what does this function compute?

This function runs is O(log **n**), regardless of **m**. Discussion: Why?

3-5. Range Query: rsq(i, j)

rsq(i, j) returns the cumulative frequencies from index **i** to **j**, inclusive.

If **i = 1**, the previous slide is sufficient.
If **i > 1**, we simply need to return: **rsq(j)–rsq(i–1)**.

Discussion: Do you understand the reason?

This function also runs in O(log **n**), regardless of **m**. Discussion: Why?

3-6. Point Update: update(i, v)

To update the frequency of a key (an index) **i** by **v** (**v** is either positive or negative; |**v**| does not necessarily be one), we use **update(i, v)**.

Indices that are related to **i** via **i' = i+LSOne(i)** will be updated
by **v** when **i** < **ft.size()** (Note that **ft.size()** is **N+1** (as we ignore index 0). These relationships form a variant of Fenwick Tree structure called the 'updating tree'.

Discussion: Do you understand this operation and on why we avoided index 0?

This function also runs in O(log **n**), regardless of **m**. Discussion: Why?

4. Second Mode

The second mode of Fenwick Tree is the one that can handle **Range Update (RU)** but only able to handle **Point Query (PQ)** in O(log **n**).

We abbreviate this type as **RU PQ**.

5. Range Update Point Query (RU PQ)

Create the data and try running the **Range Update** or **Point Query** algorithms on it. Creating the data for this type means inserting several intervals. For example, if you enter [2,4],[3,5], it means that we are updating range 2 to 4 by +1 and then updating range 3 to 5 by +1, thus we have the following frequency table: 0,1,2,2,1 that means 0 one, 1 two, 2 threes, 2 fours, 1 five.

5-1. The RU PQ Visualization

The vertices at the top shows the values stored in the Fenwick Tree (the **ft** array).

The vertices at the bottom shows the values of the data (the frequency table **f**).

Notice the clever modification of Fenwick Tree used in this RU PQ type: We increase the start of the range by +1 but decrease one index after the end of the range by -1 to achieve this result.

6. Third Mode

The third mode of Fenwick Tree is the one that can handle both **Range Update (RU)** and **Range Query (RQ)** in O(log **n**), making this type on par with **Segment Tree with Lazy Update** that can also do RU RQ in O(log **n**).

7. Range Update Range Query (RU RQ)

Create the data and try running the **Range Update** or **Range Query** algorithms on it.

Creating the data is inserting several intervals, similar as RU PQ version. But this time, you can also do Range Query efficiently.

7-1. The RU RQ Visualization

In Range Update Range Query Fenwick Tree, we need to have two Fenwick Trees. The vertices at the top shows the values of the first Fenwick Tree (BIT1[] array), the vertices at the middle shows the values of the second Fenwick Tree (BIT2[] array), while the vertices at the bottom shows the values of the data (the frequency table). The first Fenwick Tree behaves the same as in RU PQ version. The second Fenwick Tree is used to do clever offset to allow Range Query again.

8. Extras

We have a few more extra stuffs involving this data structure.

8-1. Implementation

Unfortunately, this data structure is not yet available in C++ STL, Java API, Python or OCaml Standard Library as of 2020. Therefore, we have to write our own implementation.

Please look at the following C++/Python/Java/OCaml implementations of this Fenwick Tree data structure in Object-Oriented Programming (OOP) fashion: fenwicktree_ds.cpp | py | java | ml

Again, you are free to customize this custom library implementation to suit your needs.