On vEB Trees

David Keisar Schmidt

February 13, 2022

Abstract

A Predecessor algorithm defines a generic sorting algorithm, that is, find the elements set maximum and repeatedly find its predecessor. Thus, the elements number is n and the algorithm complexity is $O(f(n))$, the resulted sorting algorithm complexity is $O(n \cdot f(n))$. It is known that the sorting complexity has a spell bound of $n \log n$, so without any additional assumptions, it follows that the time complexity of any predecessor algorithm is bounded by $\log n$ since otherwise we could achieve sorting time better than $n \log n$. However, by bounding the size of the largest element u, we can achieve sorting in $O(u)$ and when $u = O(n)$, it becomes a linear time sorting.[1]. Hence, a question arises - can we achieve better predecessor algorithms by bounding u?

[1]See the counting sort algorithm

Contents

1 A constant height tree

Let u be an upper bound of an array M, which means the array contains elements in the range $\{0, \ldots, u-1\}$. We define an array $A[x] = \mathbb{1}_{x \in M}$, that is, A is a bitfield array containing 1 in the index x iff $x \in M$ and 0 otherwise.

In addition, we define a summary array of size $\sqrt{u}$, where summary $[i]$ tells us if there is a 1 in the range $A[i\sqrt{u}, \ldots, (i+1)\sqrt{u}-1]$, thus, summary $[i] = \bigvee_{j \in A\left[i\sqrt{u}, \ldots, (i+1)\sqrt{u}-1\right]} j$, the logical or of the array $A[i\sqrt{u}, \ldots, (i+1)\sqrt{u}-1]$. The $\sqrt{u}$ sized sub-array $A[i\sqrt{u}, \ldots, (i+1)\sqrt{u}-1]$ is called the i^{th} cluster of A. Besides, we add the logical or of summary - containing 1 iff the array is not empty.

Now, in order to find minimum and maximum we can do the following operations:

- Minimum: Find the left most index i in the summary with value 1, run a linear scan in the i^{th} cluster to find the left most positive value, which is, the minimum.

Besides, now, given an element x, we know that x is in cluster number $\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$, thus, in order to delete x, we update $A[x] = 0$, and recompute summary $\left[\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor\right]$, and the logical or of summary. a similar operation is required for insertion.

In both operations we needed to scan an array of size $\sqrt{u}$, so the time complexity if both cases is $O(\sqrt{u})$, that is, however, extremely poor, since AVL, Red&Black trees achieve better results - $O(\log u)$. Though we shall see we can develop it and achieve better running time, since this design will turn out as a key design.

2 A recursive data structure

Instead of storing only two tree levels, we create a recursive data structure. The array "summary" remains the same, however, summary $[i]$ contains also a smaller structure of size $\sqrt{u}$, of i^{th} cluster elements, with the same property. Which means, the first level of the tree contains u elements, the second level contains $u^{\frac{1}{2}}$structures each containing $u^{\frac{1}{2}}$ elements, the third contains $u^{\frac{1}{4}}$ of $u^{\frac{1}{4}}$ elements and so on. We should assume just for know that $u = 2^{2^k}$ so in each level we get an integer.

Consider the following recurrence $T(u) = T(\sqrt{u}) + O(1)$ that is, in each step we shrink the recursion by a factor of $\sqrt{u}$, substituting $m = \lg u$ we get $u = 2^m$ thus $T(2^m) = T\left(2^{\frac{m}{2}}\right) + O(1)$ thus the recurrence $S(m) = T(2^m) = S\left(\frac{m}{2}\right) + O(1)$ has the solution $S(m) = O(\lg m)$ this $T(u) = O(\lg m) = O(\lg\lg u)$, there we would like to design our tree to behave in a similar way - it requires a tree height of $\sqrt{u}$.

Now, consider the previous structure with constant height, given a value x, it is stored in cluster number $\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor = i$ and in x's cluster, the elements are $\sqrt{u}i, \ldots, \sqrt{u}(i+1) - 1$ and x between them, thus its index is $x \mod \sqrt{u}$,

Note that since $x < u = 2^{2^k}$, x is represented by at most $\lg u$ bits and if we view x as a $\lg u$ binary number, $\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$ is represented by the most significant $\frac{\lg u}{2}$ bits of x, and $x \mod \sqrt{u}$ is given by the least $\frac{\lg u}{2}$ bits of x. There is a connection between those two expressions:

$$\boxed{x = x \mod \sqrt{u} + \sqrt{u} \cdot \left\lfloor \frac{x}{\sqrt{u}} \right\rfloor}$$

Thus, we define the following functions:

$$\text{high}(x) = \left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$$
$$\text{low}(x) = x \mod \sqrt{u}$$
$$\text{index}(x, y) = x\sqrt{u} + y$$

and it follows that $\text{index}(\text{high}(x), \text{low}(x)) = x$, and that is how we can generate x back just from those two simple functions.

This discussion leads us to the vEB-prototype.

Definition 1. The universe size in each level in the tree is $u^{\frac{1}{2^k}}$ where k is the height of the tree ($k = 0$ is the leaves level).

3 Proto vEB Trees

We will now define the prototype of vEB tree, which we Denote as *proto-vEB*(u) where u is its universe size, as follows:

- if $u = 2$ this is the base case - we contain only an array of two bits $A[0 \ldots 1]$
- otherwise $u = 2^{2^k}$ for some integer $k \geq 1$, thus, $u \geq 4$. In addition to the universe size, proto-vEB contains the following attributes:
  - a pointer named summary to a *proto-vEB*$(\sqrt{u})$ structure.
  - an array cluster $[0 \ldots \sqrt{u} - 1]$ of $\sqrt{u}$ pointers, each points to a *proto-vEB*$(\sqrt{u})$ structure.

For a given element $x \in \{0, \ldots, u-1\}$, x is recursively stored in cluster number $\text{high}(x)$ as the $\text{low}(x)$ number.

As not as the simple structure we described in the previous section, the array summary does not contain explicit results, but allows us to compute the summary bit recursively. In particular, now, summary contains the indices of clusters that contains any array elements. Since the naive structure's summary at some index i contained 1 iff there was an array element in the range $\{i\sqrt{u}, \ldots, (i+1)\sqrt{u}-1\}$, now in the prototype, we have the same property, but it is achieved recursively.

Now, our goal is to describe algorithms for the following tasks:

1. Query operations - Member, Minimum
2. Successor - does not change the structure.
3. Insert and Delete.

Maximum and predecessor can be achieved in a similar manner. All those operations receive an element x and assume its validity, that is $0 \leq x < u$.

3.1 Determining Membership

In order to find x, we need to split the task into two cases:

- if $u = 2$, we only to check if $A[x] = 1$.
- otherwise, we need to search for $\text{low}(x)$ in $\text{cluster}[\text{high}(x)]$, a recursive process.

Assume the data structure is V, we get the following algorithm:

Algorithm 1 Proto-vEB-Member(V, x)

```
if V.u == 2 then
    return V.A[x]
return Proto-vEB-Member(V.cluster[high(x)], low(x))
```

The running time of this algorithm is

$$T(u) = T\left(\sqrt{u}\right) + O(1)$$

, as we have already analyzed, the solution is $T(u) = O(\lg \lg u)$, which is far better than $O(\sqrt{u})$ and $O(\lg u)$.

3.2 Finding The Minimum Element

In previous sections, we described a naive solution - scanning the summary array from the left till we find 1, and then scanning the corresponding cluster from the left to find the minimum. Now, when summary is not an array but a proto-vEB, we can achieve better results. Firstly, denote that the minimum element appears in the cluster with minimum index, thus, we can search for minimum in summary and get $min - cluster = \text{high}(x)$ index, and then again in the cluster to find the $offset$ which is $\text{low}(x)$. Thus, we shall return $\text{index}(\text{high}(x), \text{low}(x))$. Secondly, the base case is again when $u = 2$, then we just scan the array. Our algorithm returns NIL when the structure is empty, that is, there is not minimum.

Algorithm 2 Proto-vEB-Minimum(V)

```
if V.u == 2 then
    if V.A[0] == 1 then
        return 0
    else if V.A[1] == 1 then
        return 1
    else return NIL
else min-cluster=Proto-vEB-Minimum(V.summary)
    if min-cluster==NIL then
        return NIL
    else offset=Proto-vEB-Minimum(V.cluster[min − cluster])
        return index(min-cluster, offset)
```

The algorithm in the worst case performs two recursive calls and some constant operations, so the running time is

$$T\left(u\right)=2T\left(\sqrt{u}\right)+O\left(1\right)$$

To solve that we define $m=\lg u$ thus $u=2^{m}$ and the formula $S\left(m\right)=T\left(2^{m}\right)=2T\left(2^{\frac{m}{2}}\right)+O\left(1\right)=2S\left(\frac{m}{2}\right)+O\left(1\right)$, hence, from the master theorem, $S\left(m\right)=\Theta\left(m\right)=\Theta\left(\lg u\right)$. That is not better than a balanced bst, and we will see we can improve it.

3.3 Finding The Successor

In order to find the successor, we need, again to split to some cases.

1. Base case: if $u=2$, if $x=1$, there is no successor, and if $x=0$ we return 1 if $A\left[1\right]=1$ and 0 NIL otherwise.
2. We do know the cluster of x which is $\text{high}\left(x\right)$, so we can $\text{low}\left(x\right)$ successor in the $\text{high}\left(x\right)$ cluster and if we receive an index i we can return

index (high (x), i). otherwise, we move to the next cluster till we find the successor.

3. If now successor was found, we return NIL.

However, since the summary array is a proto-vEB structure itself, if no successor was found in cluster number high (x) we can find the successor of high (x) in summary, and then returns its minimum, if exists. This yields the following algorithm:

Algorithm 3 Proto-vEB-Successor(V, x)

```
if V.u == 2 then
    if x == 0 and V.A[0] == 1 then
        return 1
    else return NIL
else offset=Proto-vEB-Successor(V.cluster[high(x)], low(x))
    if offset ≠ NIL then
        return index(high(x), offset)
    else succ-cluster=Proto-vEB-Successor(V.summary, high(x))
        if succ-cluster == NIL then
            return NIL
        else offset=Proto-vEB-Minimum(V.cluster[succ − cluster])
            return index(succ − cluster, offset)
```

In the worst case, our algorithm does two recursive calls and a call to Proto-vEB-Minimum, thus the recurrence is

$$T(u) = 2T\left(\sqrt{u}\right) + \Theta(\lg u)$$

we can simply prove that $T(u) = \Theta(\lg u \lg\lg u)$.

3.4 Insertion

in order to insert an element we need to insert $\text{low}(x)$ to the cluster number $\text{high}(x)$ and insert $\text{high}(x)$ to summary.

Thus we get the following algorithm:

Algorithm 4 Proto-vEB-Insert(V, x)

if $V.u == 2$ **then**
  $V.A[x] = 1$
else Proto-vEB-Insert($V.cluster[high(x)], low(x)$)
  Proto-vEB-Insert($V.summary, high(x)$)

The recurrence is $T(u) = 2T(\sqrt{u}) + O(1)$ so as we proved in previous sections, $T(u) = \Theta(\lg u)$. not good enough.

3.5 Deletion

The current structure requires a linear scan over all $\sqrt{u}$ in $x's$ cluster, or, the addition of an element *count* that counts the number of elements in the structure.

Assuming n exists:

Algorithm 5 Proto-vEB-Delete(V, x)

if $V.u == 2$ **then**
  $V.A[x] = 0$
  $V.count = V.count - 1$
else Proto-vEB-Delete($V.cluster[high[x]], low(x)$)
  if $V.cluster[high[x]].count == 1$ **then**
    Proto-vEB-Delete($V.summary, high(x)$)

The recurrence in the worst case is $T(u) = 2T(\sqrt{u}) + O(1)$, thus $T(u) = O(\lg u)$.

4 The van Emde Boas tree

Our prototype suffers from an impractical assumption - $u = 2^{2^k}$. Our structure needs to be able to deal with more "flexible" numbers, we will assume they are just powers of two $u = 2^m$. if m is odd, that is $m = 2k+1$, we divide the bits of u to $\left\lceil \frac{\lg u}{2} \right\rceil$ most significant bits and $\left\lfloor \frac{\lg u}{2} \right\rfloor$ least significant bits.

For simplicity we denote $\sqrt[\downarrow]{u} = 2^{\left\lfloor \frac{\lg u}{2} \right\rfloor}$, $\sqrt[\uparrow]{u} = 2^{\left\lceil \frac{\lg u}{2} \right\rceil}$. It follows that $\sqrt[\downarrow]{u} \leq \sqrt{u} \leq \sqrt[\uparrow]{u}$ and $\sqrt[\downarrow]{u} \cdot \sqrt[\uparrow]{u} = u$. Thus, the fundamental methods we have already defined become:

$$\text{high}(x) = \left\lfloor \frac{x}{\sqrt[\downarrow]{u}} \right\rfloor$$
$$\text{low}(x) = x \mod \sqrt[\downarrow]{u}$$
$$\text{index}(x, y) = x \cdot \sqrt[\downarrow]{u} + y$$

We denote our structure as *vEB tree*, which is a modification to the Proto-vEB-tree we defined in the previous section. A vEB tree with universe size u is denoted as $vEB(u)$, and contains the following attributes:

- summary - points to a $vEB(\sqrt[\uparrow]{u})$ tree.
- cluster$[0 \ldots \sqrt[\uparrow]{u} - 1]$ - points to $\sqrt[\downarrow]{u} \cdot vEB(\sqrt[\uparrow]{u})$ trees.
- min - stores the minimum element in the vEB tree.
- max - stores the maximum element in the vEB tree.

Remark 1. min is stored only in the base vEB tree, and not in its recursive sub trees, thus, all the attributes stores in V is *min* plus all the elements recursively stored in the sub trees. That means, if $V.min = 0$ then cluster number 0 does not contain the value 0!

Those two additional attributes (min, max) reduce the running time significantly, and give us a total running time of $\Theta(\lg\lg u)$. For instance, to determine whether the tree is empty or not we can check $min \neq NIL$, and to see if there is only one element, we can compare $max == min$. Besides, in order to know whether an element x is the last element in the tree, we can compare $x < max$ or $x > min$. Moreover we can deduce if $x's$ successor is in the cluster number high (x), since this happens iff $x < max$ - that reduces the amount of recursive calls to only one.

In addition, inserting en element to an empty vEB can be done in $O(1)$ operations, since we just need to update min, max.

Thus, we would expect to see recurrences of the form:

$$T(u) = T\left(\sqrt{u}\right) + O(1)$$

However, $\sqrt{u}$ is not necessarily an integer, though, it holds that

$$T(u) \leq T\left(\sqrt[\uparrow]{u}\right) + O(1)$$

we can substitute $m = \lg u \Rightarrow 2^m = u$ thus we get

$$T(2^m) \leq T\left(2^{\left\lceil \frac{m}{2} \right\rceil}\right) + O(1)$$

it is quite hard to work with, Hence, note that $\left\lceil \frac{m}{2} \right\rceil \leq \frac{2}{3}m$ for all $m \geq 2$, thus, we get the following recurence:

$$S(m) = T(2^m) \leq T\left(2^{\frac{2m}{3}}\right) + O(1) = S\left(\frac{2}{3}m\right) + O(1)$$

the solution from the master theorem is $S(m) = O(\lg(m)) \Rightarrow T(u) = S(\lg u) = O(\lg \lg u)$. Therefor, extending the image of u to numbers of the form 2^k does not hurt our running time.

4.1 Finding the Minimum and Maximum Elements

Since we added min and max we get for free the following algorithms:

Algorithm 6 vEB-Tree-Minimum(V)

```
return V.min
```

Algorithm 7 vEB-Tree-Maximum(V)

```
return V.max
```

4.2 Determining Membership

We have two cases:

- if x is the minimum element or maximum, x is a member.
- else if $u = 2$: that means x can be only min or max, and thus, it is false.
- else check for membership in cluster number high (x) for low (x)

That yields the following algorithm:

Algorithm 8 vEB-Tree-Member(V, x)

1: **if** $V.min == x$ or $V.max == x$ **then**
2:     **return** True
3: **else if** $V.u == 2$ **then**
4:     **return** False
5: **else return** vEB-Tree-Member$(V.cluster[high(x)], low(x))$

In the worst case we have one recursive call, so the recurrence is $T(u) \le T(\sqrt[2]{u}) + O(1)$, which from a previous section, has the solution $T(u) = O(\lg \lg u)$.

4.3 Successor Query

We assume $0 \le x < u$ we does not care whether x is in the tree or not.

Lets split into some cases:

- Base case: if $u == 2$ then
  - if $x == 0$ and $V.max == 1$ we return 1.
  - otherwise, $x == 1$ or $V.max == 0$ Thus, x does not have any successor and we return NIL.
- else if $V.min \neq NIL$ and $x < V.min$ we know for sure, though x not in the tree, that its successor in the tree is $V.min$, so we return $V.min$.
- Otherwise, $x \ge V.min$ so if $x < V.max$ we return the successor of low (x) it its cluster, and otherwise, we search for high (x)'s successor in the summary array and return the minimum in this cluster.

Thus, we get the following algorithm:

Algorithm 9 vEB-Tree-Successor(V, x)

```
if u == 2 then
    if x == 0 and V.max == 1 then
        return 1
    else return NIL
else if V.min ≠ NIL and x < V.min then
    return V.min
else max-low = vEB-Tree-Maximum(V.cluster[high(x)])
    if x < max − low then
        offset = vEB-Tree-Successor(V.cluster[high(x)], low(x))
        return index(high(x), low(x)
    else succ-cluster = vEB-Tree-Successor(V.summary, high(x))
        if succ-cluster == NIL then
            return NIL
        else offset = vEB-Tree-Minimum(V.cluster[succ − cluster])
            return index(succ-cluster, offset)
```

There is only one recursive call and since vEB-Tree-Minimum costs $O(1)$ we get the recurrence $T(u) \leq T(\sqrt[\uparrow]{u}) + O(1)$ with the solution $T(u) = O(\lg \lg u)$.

4.4 Predecessor Query

It is quite similar to successor:

- if $u == 2$
    - if $x == 1$ and $V.min == 0$ we return 0.
    - else we return NIL since there is not predecessor for x.
- if $V.max \neq NIL$ and $x > V.max$ then $V.max$ is the predecessor so we return $V.max$.

- Else, we check if the minimum value in $x's$ cluster is smaller than x, and is so, we look for $x's$ successor in cluster number high (x). otherwise, we look for high (x) successor in the summary and then return its maximum.

Thus, we get an analogical algorithm, with one more if statement, since if there is no pred cluster, the predecessor might be $V.min$, because it is not stored in the tree's clusters:

Algorithm 10 vEB-Tree-Predecessor(V, x)

```
if u == 2 then
    if x == 1 and V.min == 0 then
        return 0
    else return NIL
else if V.max ≠ NIL and x > V.max then
    return V.max
else min-low = vEB-Tree-Minimum(V.cluster[high(x)])
    if x > min − low then
        offset = vEB-Tree-Predecessor(V.cluster[high(x)], low(x))
        return index(high(x), low(x)
    else pred-cluster = vEB-Tree-Predecessor(V.summary, high(x))
        if predcluster == NIL then
            if V.min ≠ NIL and x > V.min then
                return V.min
            return NIL
        else offset = vEB-Tree-Maximum(V.cluster[pred − cluster])
            return index(pred-cluster, offset)
```

The same recurrence we got from vEB-Tree-Successor holds as well.

4.5 Insertion

In order to insert an element, do the following steps:

- if $V.min == NIL$ then we update $V.min = V.max = x$

- else if $x < V.min$ we exchange x and $V.min$ so we insert only element that is not the minimum to its cluster (since $V.min$ is not stored in any cluster).
    - if $u > 2$ then we find the minimum in cluster number high $(V.min)$ and if it is NIL, we insert high $(V.min)$ to the summary array. In either case, we need to insert $V.min$ to cluster number high $(V.min)$
    - we do need to update max so if $x > V.max$ we change $V.max = x$.

We are going to use a helper procedure that handles empty trees:

Algorithm 11 vEB-Empty-Tree-Insert(V, x)

```
V.min=x
V.max=x
```

That yields the following algorithm

Algorithm 12 vEB-Tree-Insert(V, x)

```
if V.min == NIL then
    vEB-Empty-Tree-Insert(V, x)
else if x < V.min then
 Exchange x with V.min
if V.u > 2 then
    if vEB-Tree-Minimum(V.cluster[high(x)]) == NIL then
        vEB-Tree-Insert(V.summary, high(x))
        vEB-Empty-Tree-Insert(V.cluster[high(x)], low(x))
    else vEB-Tree-Insert(V.cluster[high(x)], low(x))
if V.max < x then
    V.max = x
```

All the cases contain only one recursive call so the running time, as we have seen in previous sections is $O(\lg \lg u)$

4.6 Deletion

We assume x is currently in the tree.

There are many cases, so lets go over them briefly

- if $V.min = V.max$ we can set $V.min = V.max = NIL$
- else if $V.u == 2$ then $V.min \neq V.max$ thus if $x == 0$ we can set $V.min = 1$ and otherwise we set $V.max = V.min$.
- else if $x == V.min$ then we need to find the second smallest element, delete it from the tree, and set it to be $V.min$.
    - Thus, we find the first cluster - which is the minimum of the summary array, and then set $V.min$ to be the minimum of the first cluster, since the $V.min$ itself is not stored in any cluster, this is the second smallest element.
- Now we delete this value from its cluster and need to check if we need to delete the cluster from summary, so the cluster is empty, we delete high (x) from the summary.
- In addition, if we deleted the maximum, we need to update. Thus, we find the maximum cluster - if it is NIL, then the tree contains only min, max so we set $V.min = V.max$, otherwise we set $V.max$ to be the maximum of the cluster.
- Finally, if $x's$ cluster didn't become empty, we might need to update $V.max$, so if we do, we update $V.max$ to be the maximum in the cluster of x.

All of this yields the following algorithm:

Algorithm 13 vEB-Tree-Delete(V, x)

```
if V.min == V.max then
    V.min=V.max=NIL
else if V.u == 2 then
    if x == 0 then
        V.min=1
    elseV.min=0
        V.max = V.min
else if x == V.min then
    first-cluster = vEB-Tree-Minimum(V.cluster[high(x)])
    x=index(first-cluster, vEB-Tree-Minimum(first − cluster))
    V.min = x
vEB-Tree-Delete(V.cluster[high(x)], low(x))
if vEB-Tree-Minimum(V.cluster[high(x)]) == NIL then
    vEB-Tree-Delete(V.summary, high(x))
    if x == V.max then
        summary-max=vEB-Tree-Maximum(V.summary)
        if summary-max == NIL then
            V.max=V.min
        else offset = vEB-Tree-Maximum(V.cluster[summary − max])
            V.max = index(summary-max, offset)
    else if  x == V.max then
        offset = vEB-Tree-Maximum(V.cluster[high(x)])
        V.max = index(high(x), offset)
```

It seems that there are two recursive calls in the worst case, one to delete from summary and one to delete x from the cluster. However, that happens only when x is the only element in its cluster, thus, the first recursive call terminates in the first if statement, so effectively we need to take care of only one call. Thus, the recurrence is again $T\left(u\right) \leq T\left(\sqrt[\uparrow]{u}\right) + O\left(1\right)$ and hence $T\left(u\right) = O\left(\lg\lg u\right)$.

4.7 Space Complexity

A vEB-Tree contains 2 simple attributes, and $\sqrt[\uparrow]{u}+1$ arrays of vEB-trees of size $\sqrt[\downarrow]{u}$ and array of pointers thus the recurrence is $P(u) = (\sqrt[\uparrow]{u}+1)\, P(\sqrt[\downarrow]{u}) + \Theta(\sqrt[\uparrow]{u})$, for simplicity, we assume $\sqrt{u}$ is always an integer so $P(u) = (\sqrt{u}+1)\, P(\sqrt{u}) + \Theta(\sqrt{u})$.

Which, apparently, has the solution $T(u) = O(u)$.

4.8 Building an Empty vEB tree

Assuming we would like to build a vEB-tree with universe size u, we need to create $\sqrt{u}+1\ vEB(\sqrt{u})$ and execute $\Theta(\sqrt{u})$ operations to store pointers in the array, thus the formula is exactly as the space complexity recurrence

$$T(u) = \left(\sqrt{u}+1\right) T\left(\sqrt{u}\right) + \Theta\left(\sqrt{u}\right)$$

Hence, $T(u) = O(u)$, so in cases when the amount of elements is small, we might prefer to use Red&Black Trees.

5 Bibliography

This scribe is mostly based on the book “Introduction to algorithms” by Thomas H. Cormen, without him, it would have been impossible. It is just a summary, and nothing more, it was written just for individual use, and contain parts summarized from Cormen’s book.

We encourage you to read Cormen’s vEB chapter for more examples and explanations.