VAN EMDE BOAS TREE & TREAP

TIMELINE

1

April 13-17

We start researching and learning about the data structures we're supposed to implement for this project.

2

April 18-19

We meet in-person and pitch in what we each have to offer in this project, and split the workload between ourselves.

3

April 18-19

Business Analyst provides requirement document. Initial development phase begins

TIMELINE

1

April 13-17

We start researching and learning about the data structures we're supposed to implement for this project

2

April 18-19

We meet in-person and pitch in what we each have to offer in this project, and split the workload between ourselves

3

April 19-23

Business Analyst provides requirement document. Initial development phase begins

4

April 24-26

Testing and feedback, with parallel improvements and implementation from dev team

5

April 27

Meet to discuss state of project and final touchups to it.

6

April 27-28

Final development and testing phase, with a focus in polishing.

TIMELINE

7

April 28

GitHub Repository made Public!

8

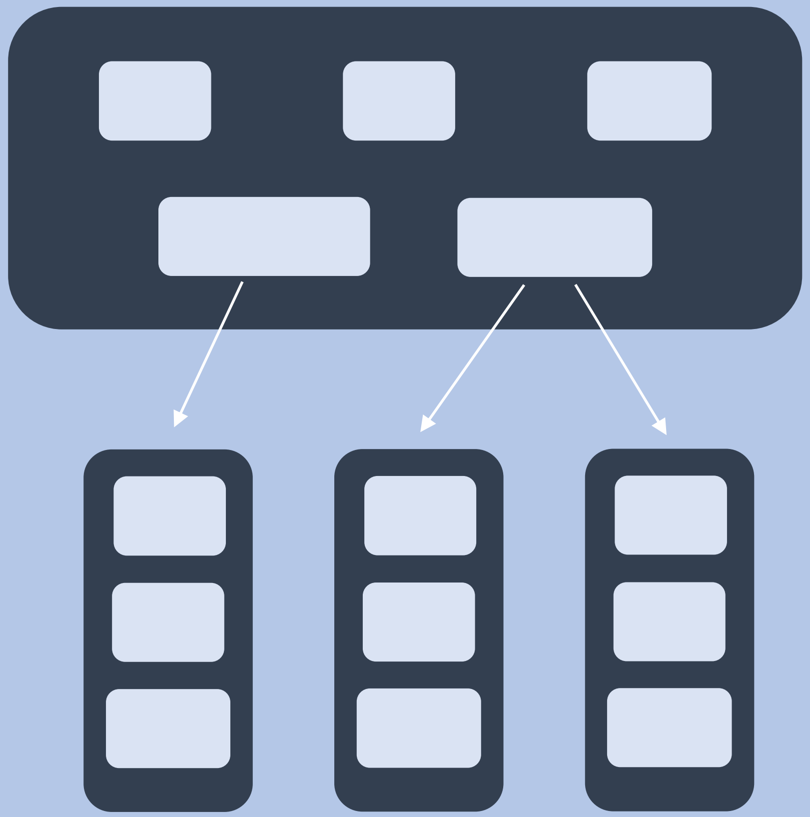
May 3

Presentation designed by Project Manager

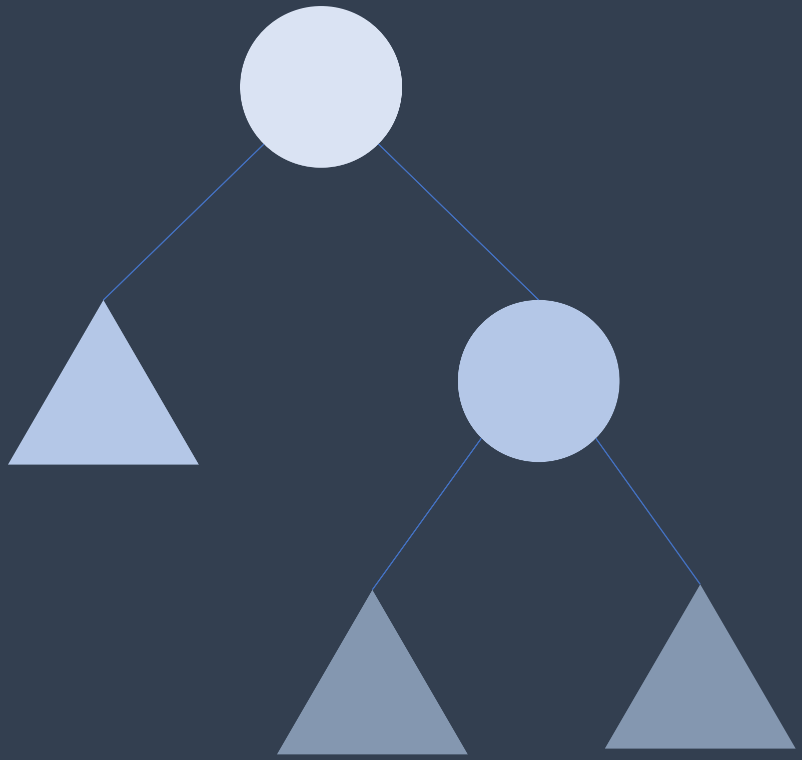
9

May 10

Changes made across the board (Presentation, Documentation, Repository) to keep things in line with new instructions.



VAN EMDE BOAS TREE



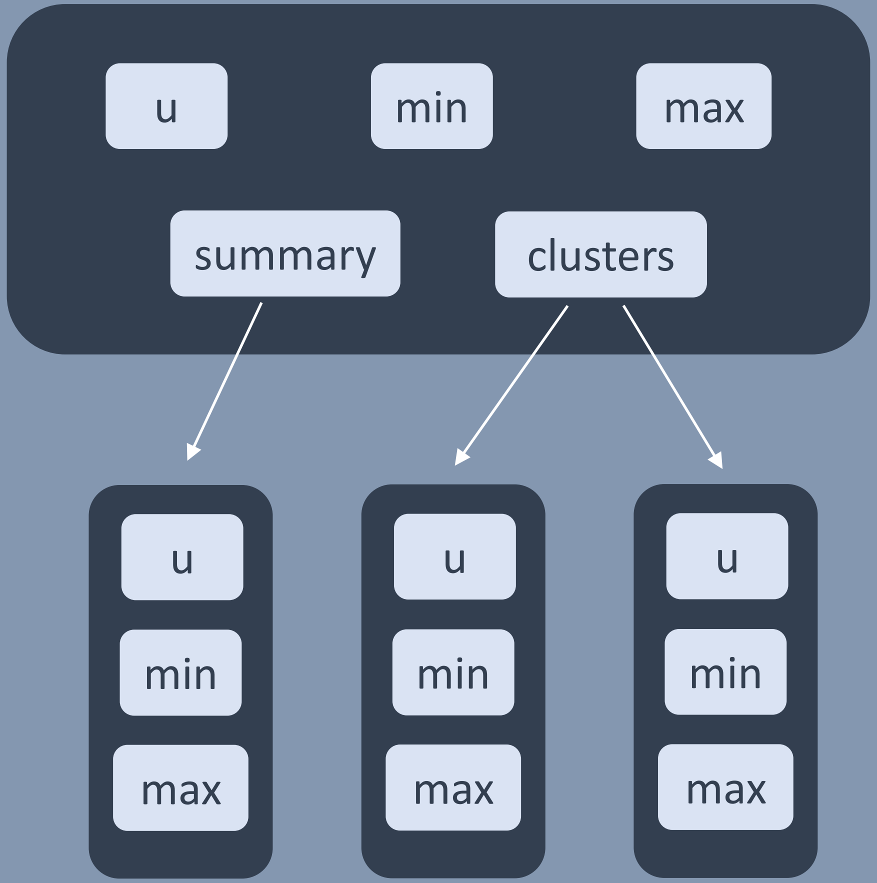
TREAP

VAN EMDE BOAS TREE

- vEB tree is a data structure that supports insanely fast operations like search, successor, predecessor, insertion and deletion, on a universe of keys.

- It uses a recursive structure where each node represents a sub-universe of keys. The root node maintains summary information about the sub-universes, while the children nodes handle details for each sub-universe.

- vEB trees are useful when dealing with a large universe of keys where quick and efficient operations would immensely help in retrieval of data.

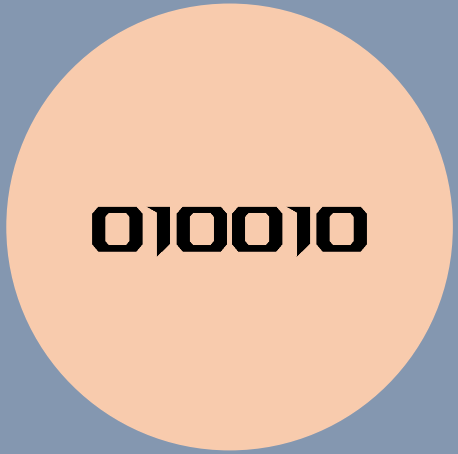


Basic structure of a vEB tree

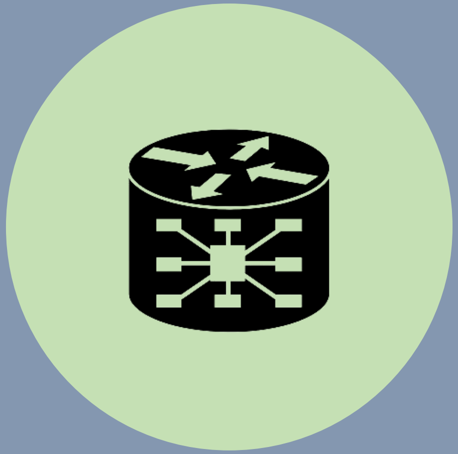
Structurally, vEB tree is a cluster of data recursively containing more clusters. Each sub-tree consists of:

- No. of keys - u
- Minimum key in the cluster.
- Maximum key in the cluster.
- A summary of keys present in the clusters array.
- An array of clusters (sub-vEB trees)

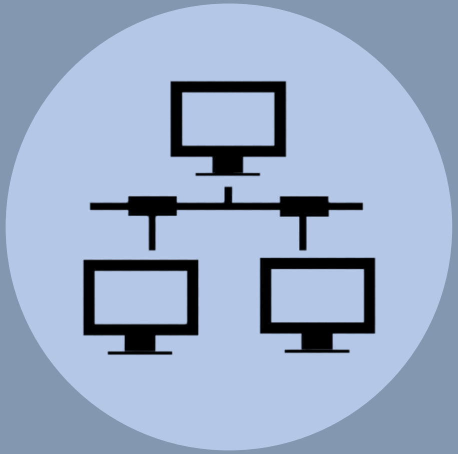
vEB trees show the quickest retrieval of successing values, which could find its use in places like



Data fetching



Routing



Packet management

Legend

Function	Formula
root(x)	sqrt(x)
high(x)	x / root(u)
low(x)	x % root(u)
index(x)	high(x) * root(u) + low(x)

Insert

Input – A (Bit array to insert to), x (Number to insert)
Output – Item is inserted in the array

```
1. A[x] <- True
```

Time Complexity
O(1)

Successor

Input – A (Bit array), x (Number of which to find successor)
Output – Returns the successor of the number, else None

```
1. i <- x + 1
2. Repeat until i > A.universe_size
    1. If A[i] = True
        1. Return i
    2. i <- i + 1
3. Return None
```

Time Complexity
O(u)

Insert

Input – A (Bit array), S (Summary array), x (Number to insert)
Output – Number is inserted to both arrays

```
1. A[x] <- True
2. S[high(x)] <- True
```

Time Complexity
O(1)

Successor

Input – A (Bit array), x (Number to find summary of)
Output – Returns the successor of n in A, else None

```
1. high_x <- high(x)
2. low_x <- low(x)
3. offset <- low_x
4. Repeat until offset < root(u_size)
    1. If V[index(high_x), offset]
        1. return index(high_x, offset)
    2. offset <- offset + 1
5. For each i from high_x + 1 to root(u_size)
    1. If summary[i] = True
        1. For each j from 0 to root(u_size)
            1. If bitvec[index(i, j)]
                1. return index(i, j)
6. Return None
```

Time Complexity
$O(\sqrt{u})$

Insert

Input – V (Van Emde Boas Tree), x (Number to insert)
Output – x is inserted into the tree

```
1. Insert(V.cluster[high(x)], low(x))
2. Insert(V.summary, high(x))
```

```
T(u) = 3 T(√u) + O(1)

T(u) ~ O((log u) ^ 1.585)
```

Time Complexity
O(log u)

Successor

Input – V (Van Emde Boas Tree), x (Number to find successor of)
Output – Returns successor of x

```
1. i <- high(x)
2. j <- Successor(V.cluster[i], j)
3. if j = ∞
    1. i <- Successor(V.summary, i)
    2. j <- Successor(V.cluster[i], -∞ )
4. return index(i, j)
```

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

$$T(u) = O(\log u)$$

Time Complexity

O(log u ^ 1.585)

Insert

Input – V (Van Emde Boas Tree), x (Number to insert)
Output – x is inserted into the tree

```
1. If V.min = None
    1. V.min <- x
    2. V.max <- x
    3. Return
2. if x < V.min
    1. swap x <-> V.min
3. if x > V.max
    1. V.max <- x
4. if V.cluster[high(x)] = None
    1. Insert(V.summary, high(x))
5. Insert(V.cluster[high(x)], low(x))
```

Time Complexity
O(log log u)

Successor

Input – V (Van Emde Boas Tree), x (Number to find successor of)
Output – Returns successor of x

```
1. i <- high(x)
2. If low(x) < V.cluster[i].max
    1. j <- Successor(V.cluster[i], low(x))
3. Else
    1. i <- Successor(V.summary, high(x))
    2. j <- V.cluster[i].min
4. Return index(i, j)
```

Time Complexity
O(log log u)

Delete

Input – V (Van Emde Boas Tree), x (Number to delete)
Output – x is deleted from the tree

```
1. If x = V.min
    1. i <- V.Summary.min
    2. If i = None
        1. V.min <- None
        2. V.max <- None
        3. return
    4. x <- index(i, V.cluster[i].min)
    3. V.min <- index(i, V.cluster[i].min)
2. Delete(V.cluster[high(x)], low(x))
3. If V.cluster[high(x)].min = None
    1. Delete(V.summary, high(x))
4. If x = V.max
    1. If V.Summary.max = None
        1. V.max <- V.min
    2. Else:
        1. i <- V.summary.max
        2. V.max <- index(i, V.cluster[i].max)
```

Time Complexity

O(log log u)

Usage of unordered maps instead of vectors

Basically, when using vectors, the amount of memory utilized will be **ASTRONOMICAL!** (u – universe size)

(pun intended)

Unordered maps utilize **hashmaps** to store the clusters and **only stores nonempty clusters**, saving up on space by reducing space complexity from
O(u log log u) to **O(n log log u)**

*This is an extremely important addition that significantly reduces memory space from **Gigabytes** to mere **Bytes***

TREAP

Tree + Heap

- Treap is a hybrid data structure that combines the properties of a binary search tree (BST) and a max heap. Each node in the treap has a key (BST property) and a priority (heap property).

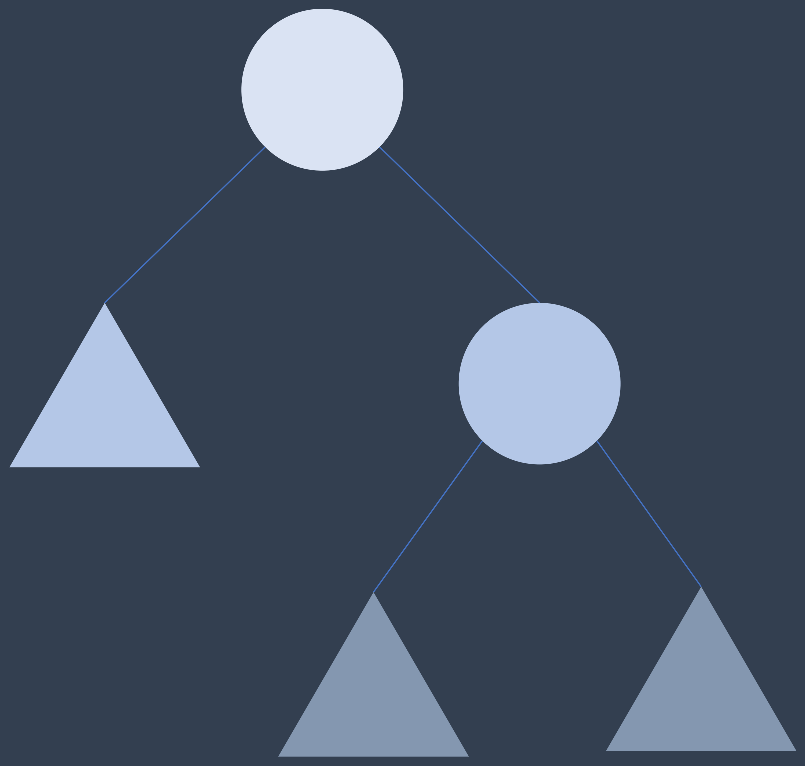
- Nodes are inserted based on their keys following BST rules, and priorities are assigned randomly but maintain the max heap property. This randomness helps balance the tree.

- Treaps are useful for scenarios where you need both the properties of a BST (sorted keys) and a max heap (priority-based operations), such as priority queues.

WHAT

HOW

WHEN

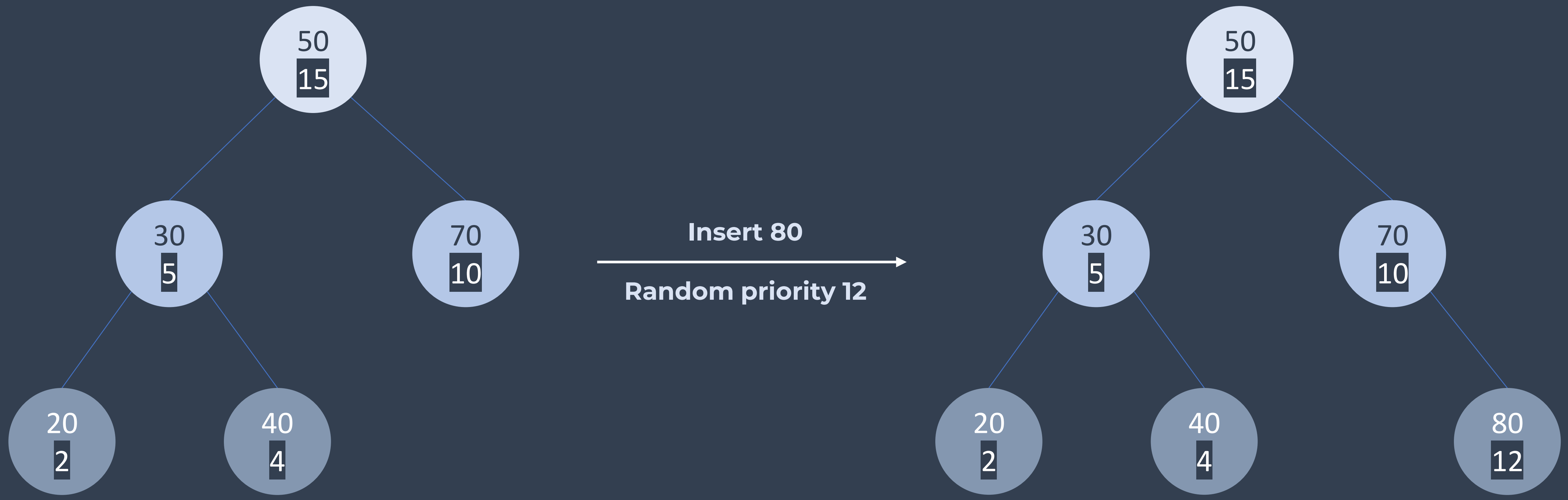


Basic structure of a Treap

Each node contains two values:

- A value corresponding to its BST property.
- A priority value corresponding to its Heap property.
- Operations are usually either from BST, Heap or a sequence of both, to maintain the "Tree + Heap" properties.

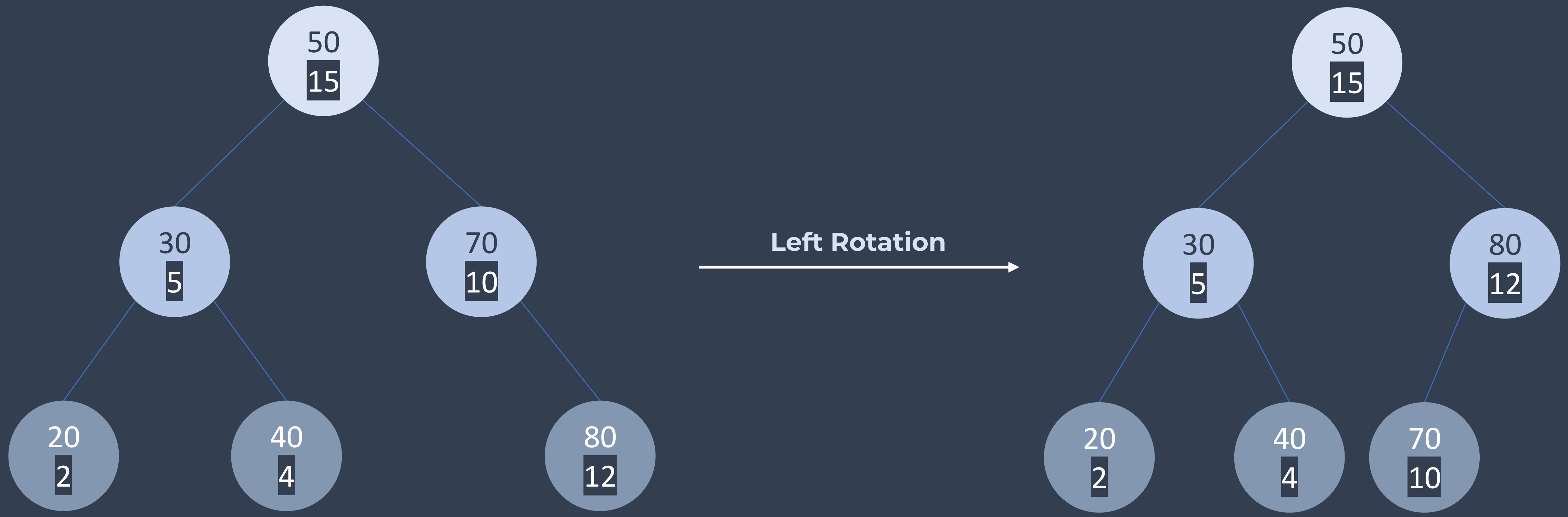
Insert



Treap before insertion

Treap after insertion

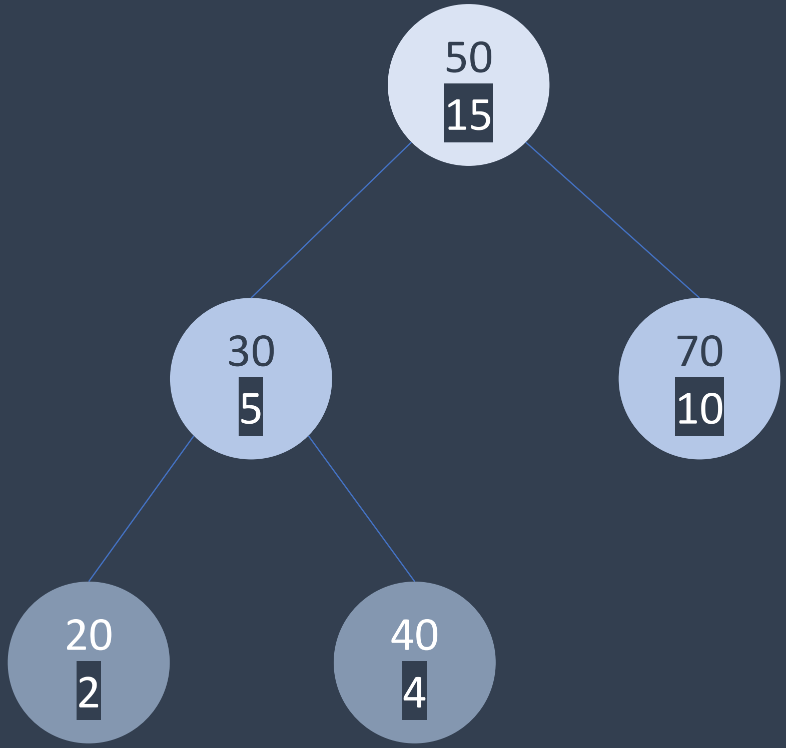
Insert



80's priority higher than 70

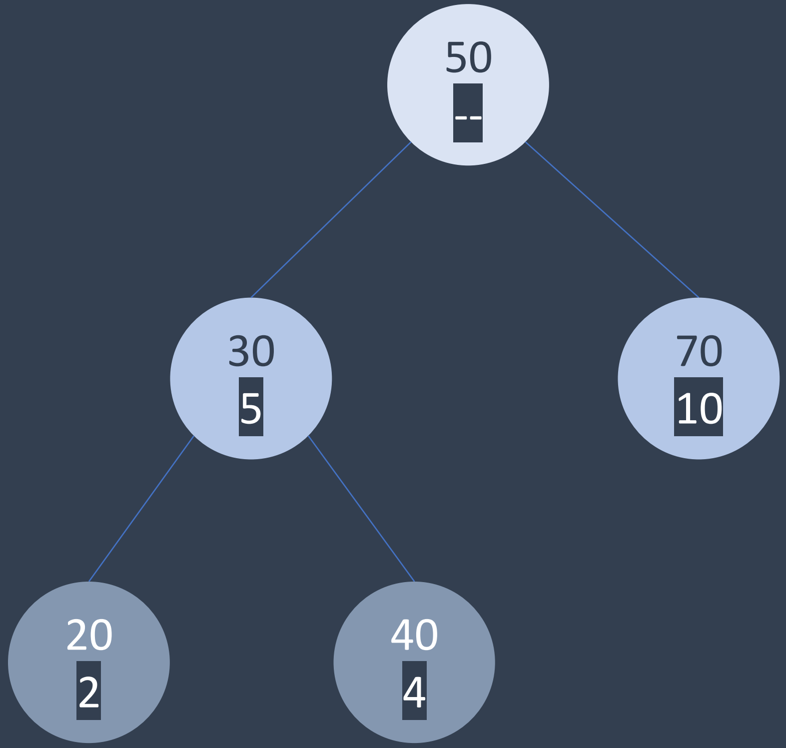
Left rotated and nodes changed according to heap property

Delete

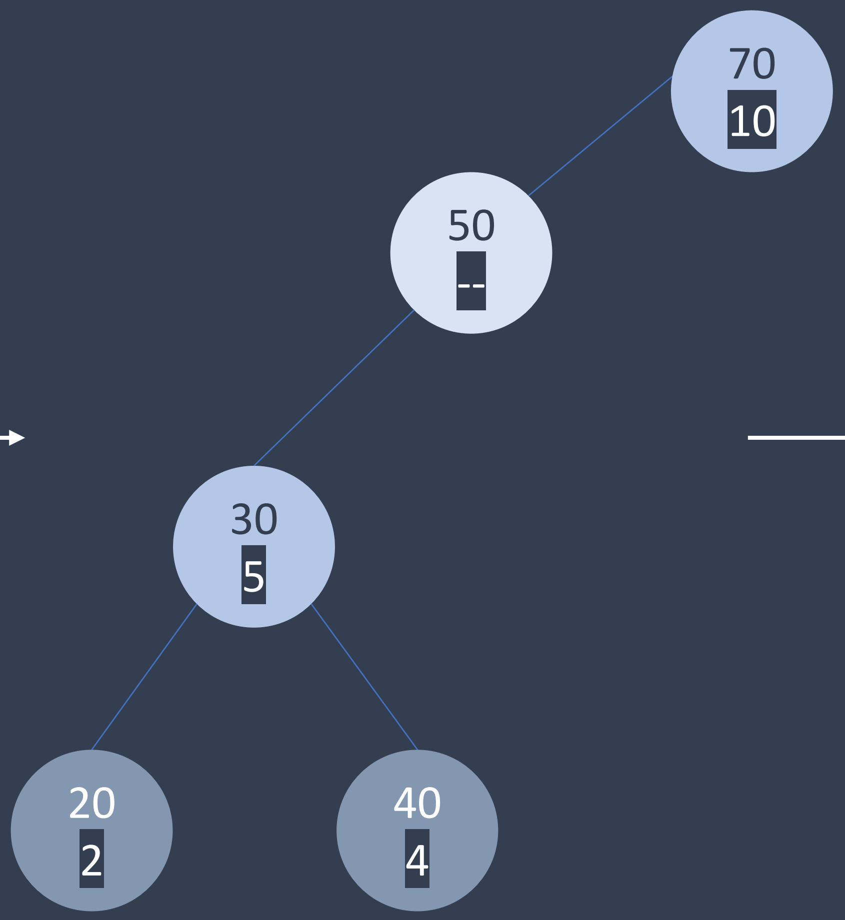


Treap initially

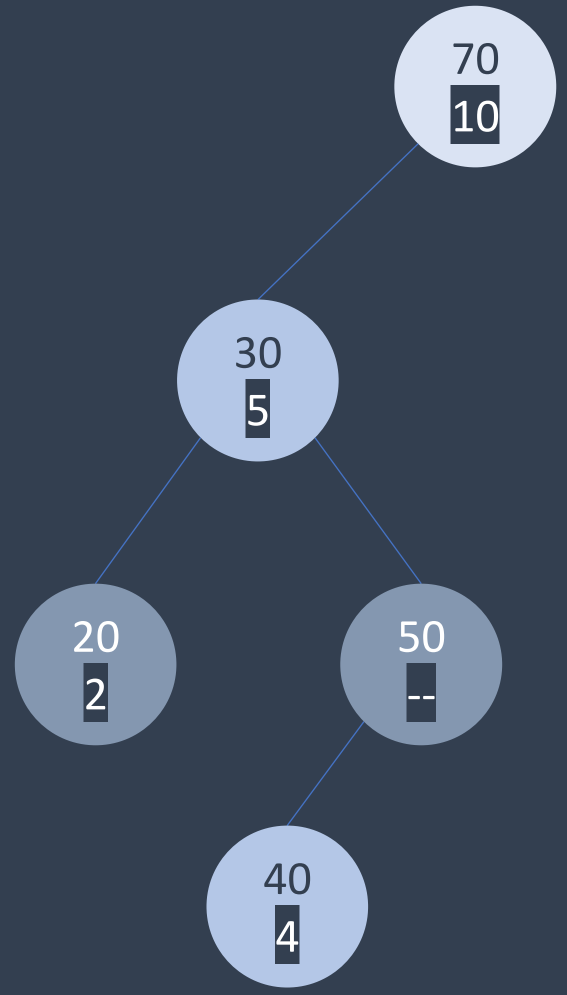
Delete 50



50’s priority removed

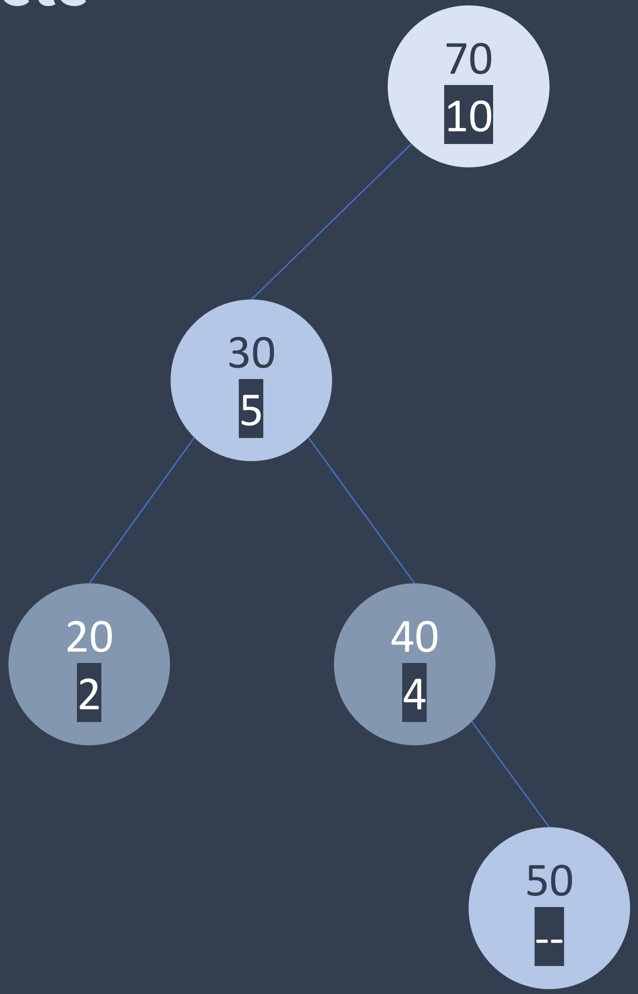


50-70 left rotated



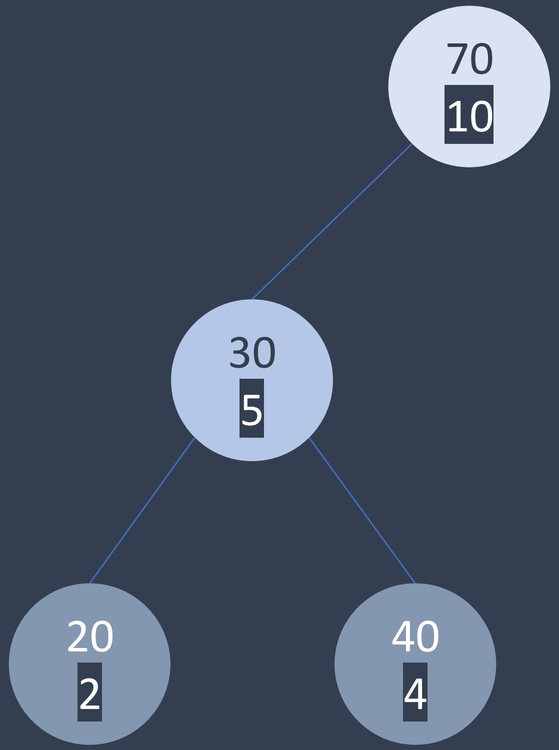
30-50 right rotated

Delete



40-50 right rotated

Since 50 node is now a leaf, it can be removed from the treap



Treap post deletion

Probability of formation of skewed trees in treap is extremely low (owing to its heap properties), hence search does not go towards O(n), making search...

EXTREMELY FAST!

Secure

Faster databases

Quicker Scheduling

Insert

Input – Key to be inserted, Priority of Key, Root, Copy of Root
Output – Root of updated treap

Time Complexity
O(log n)

```
1. if temp = NULL, do
    1. Create a new node “newnode”
    2. newnode→key ← key
    3. newnode→data ← data
    4. newnode→left ← NULL
    5. newnode→right ← NULL
    6. if root = NULL, do
        1. root ← newnode
    7. return newnode
2. else if temp→key > key, do
    1. temp→left ← repeat the algorithm with key, data and temp→left
    2. if temp→left is not NULL and temp→left→data > temp→data, do
        1. temp ← algorithm 3 with temp as input
3. else, do
    1. temp→right ← repeat the algorithm with key, data and temp→right
    2. if temp→right is not NULL and temp→right→data > temp→data, do
        1. temp ← algorithm 2 with temp as input
4. return temp
```

Left/Right rotation

Input – Node to be rotated
Output – Rotated Node

```
1. rnode ← temp→right
2. lnode ← rnode→left
3. rnode→left ← temp
4. temp→right ← lnode
5. return rnode
```

Left Rotation

```
1. lnode ← temp→left
2. rnode ← lnode→right
3. lnode→right ← temp
4. temp→left ← rnode
5. return lnode
```

Right Rotation

Time Complexity

O(1)

Delete

Input – Key to be deleted, Root, Copy of Root
Output – Root of updated treap

Time Complexity
O(log n)

```
1. if temp is NULL, do
    1. return temp
2. if temp→key > num, do
    1. temp→left ← repeat algorithm 4 with num and temp→left as input
3. else if temp→key < num, do
    1. temp→right ← repeat algorithm 4 with num and temp→right as input
4. else, do
    1. if temp is root and (temp→left or temp→right is NULL), do
        1. if temp→left and temp→right are both NULL, do
            1. root ← NULL
        2. else if temp→left is NULL, do
            1. root ← temp→right
        3. else, do
            1. root ← temp→left
        4. return root
    2. else, do
        1. if temp→left is NULL, do
            1. return temp→right
```

Delete

Input – Key to be deleted, Root, Copy of Root
Output – Root of updated treap

Time Complexity
O(log n)

```
        2. else if temp→right is NULL, do
            1. return temp→left
        3. else, do
            1. succparent ← temp
            2. succ ← temp→right
            3. repeat until succ→left is NULL,
                1. succparent ← succ
                2. succ ← succ→left
            4. temp→key ← succ→key
            5. temp→data ← succ→data
            6. if succparent is not temp, do
                1. succparent→left ← repeat the algorithm with succ→key and succparent→left as
                   inputs
            7. else, do
                1. succparent→right ← repeat the algorithm with succ→key and succparent→right as
                   inputs
            8. repeat until temp→left or temp→right is NULL or (temp→left→data <= temp→data and
               temp→right→data <= temp→data)
                1. if temp→left→data > temp→right→data, do
                    1. temp ← algorithm 3 with temp as input
                2. else, do
                    1. temp ← algorithm 2 with temp as input
5. return temp
```

Search

Input – Key to be searched, Copy of root
Output – 1 if key is found, 0 otherwise

Time Complexity
O(log n)

```
1. if temp is NULL, do
    1. return 0
2. if num = temp→key, do
    1. return 1
3. else if num < temp→key, do
    1. if temp→left is not NULL, do
        1. return output of algorithm 5 with num and temp→left as input
4. else, do
    1. if temp→right is not NULL, do
        1. return output of algorithm 5 with num and temp→right as input
5. return 0
```

Inorder display

Input – Copy of root
Output – Elements of treap

```
1. if temp is NULL,
    1. return
2. repeat the algorithm with temp→left as input
3. display temp→key and temp→data
4. repeat the algorithm with temp→right as input
```

Time Complexity

O(n)

Level order display

Input – Copy of root
Output – Elements of treap

Time Complexity
O(n)

```
1. if temp is NULL, do
    1. return
2. Create a queue “q”
3. enqueue temp into q
4. repeat until q is empty,
    1. count ← number of elements in q
    2. repeat until count  = 0,
        1. node ← frontpeek of q
        2. display node→key and node→data
        3. dequeue an element from q
        4. if node→left is not NULL,
            1. enqueue node→left into q
        5. if node→right is not NULL,
            1. enqueue node→right into q
            2. count ← count - 1
    3. display empty line
```

Citations

1. **Treaps – Complete Introduction** *Uzair Javed Akhtar*
   https://www.youtube.com/watch?v=ZNtC4oUaQ8A
2. **Treap – A randomized BST** *Geeks4Geeks*
   https://www.geeksforgeeks.org/treap-a-randomized-binary-search-tree/
3. **Divide & Conquer – van Emde Boas Tree** *Eric Demaine*
   https://ocw.mit.edu/courses/6-046j-design-and-analysis-of-algorithms-spring-2015/resources/lecture-4-divide-conquer-van-emde-boas-trees/
4. **van Emde Boas Trees** *Sam McCauley*
   https://williams-cs.github.io/cs358-f21/lectures/lecture23/veb.pdf

THE TEAM

devs

Treap

- nikhilesh h
- prem danasekaran
- nighil natarajan

van Emde Boas Tree

- saran shankar r
- raghav sridharan

testers

sathya narayanan vEB Tree

sanjeev krishna s vEB Tree

nithilan m Treap

Project Manager	ramana k s
Business Analyst	prajesh raam h s