VAN EMDE BOAS TREE (VEB-TREE)

AVL TREE

GROUP : 3

Department of Computer Science

The University of Alabama, Tuscaloosa

OVERVIEW

Three important data structures used in computer science:

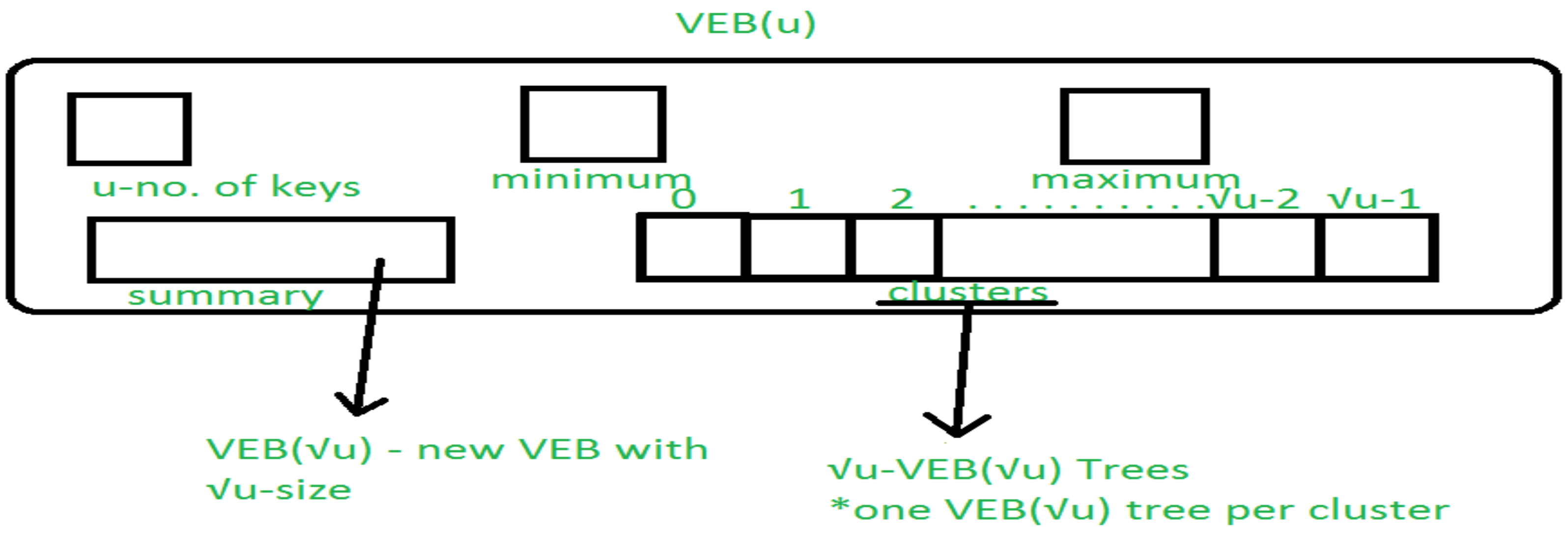
- ❑ Van Emde Boas Tree (VEB-Tree)
- ❑ AVL Tree and
- ❑ Red-Black Tree.

We will discuss their basic features, advantages and disadvantages, and applications.

VAN EMDE BOAS TREE

A VEB-Tree is a tree data structure that can store elements from a finite set of size n efficiently in O(log log n) time.

Structure :


VEB(u)
u-no. of keys
minimum
maximum
0
1
2
. . . .√u-2
√u-1
summary
clusters
VEB(√u) - new VEB with √u-size
√u-VEB(√u) Trees
*one VEB(√u) tree per cluster

VEB CONCEPTS:

1. Bit Vectors: Bit vectors provide constant time insert search and delete, but have high time complexity for successor and predecessor operations as well as high memory complexity
2. Summary Vectors: Summary vectors allow for simplifications of successor and predecessor especially in sparse arrays, but updating them complicates other operations
3. Lazy Propagation: To reduce single operation time complexity, some functions only perform the bare minimum number of changes required and procrastinate on other logical manipulations
4. Caching: Storing additional information in constant time can save significant amounts of effort down the road by reducing repetitive operations

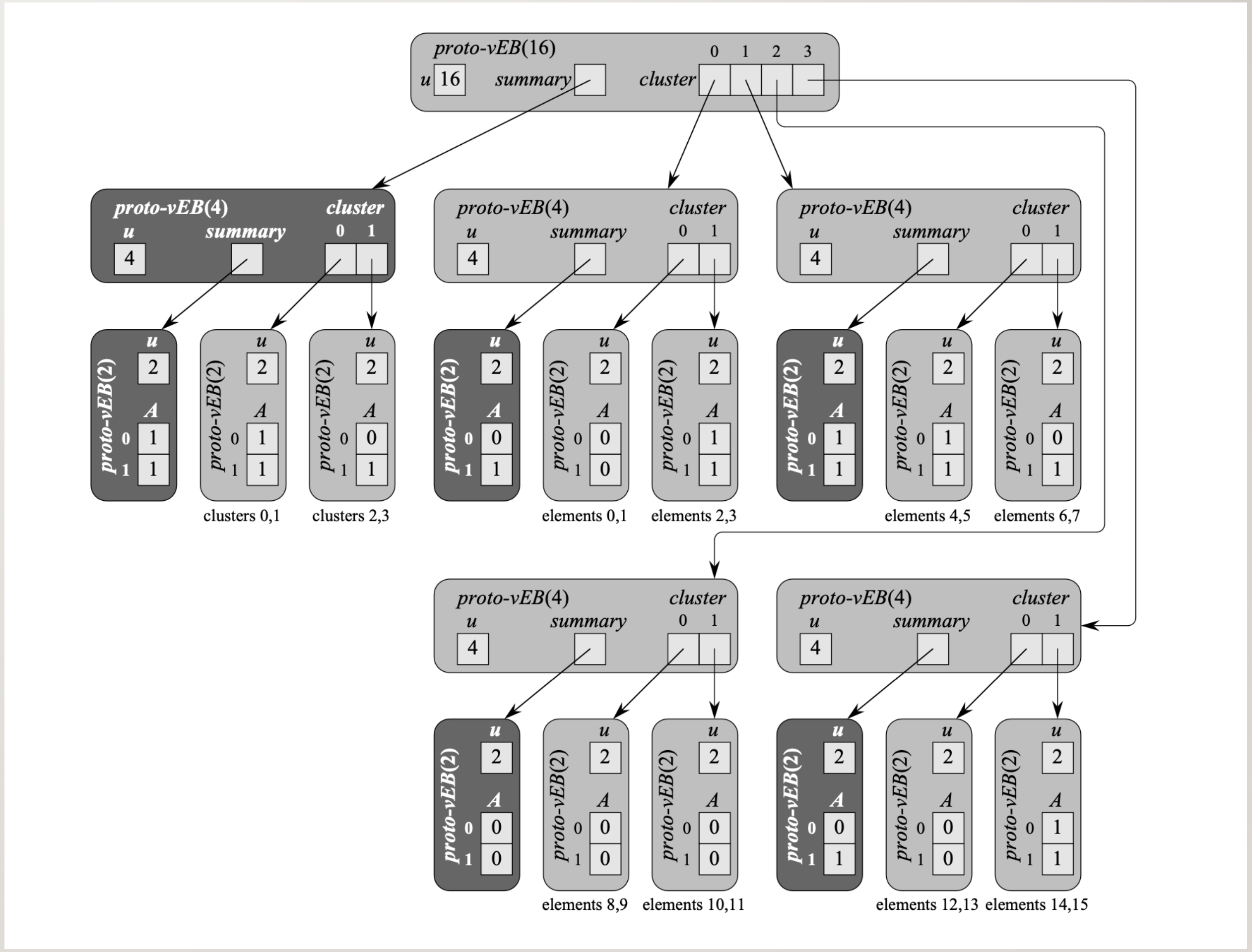
ADVANTAGES

- Efficient search and update operations
- Allows for the fast retrieval of elements based on their value, rather than their position in the tree, making it well-suited for applications such as range queries or priority queues
- Can be used to implement dynamic sets with search, insertion, and deletion operations that are exponentially faster than most other data structures that provide these functions (such as Red-Black trees or other balanced BSTs)
- Well-suited for applications that require a large number of elements to be stored, as they have an overall logarithmic time complexity
- Minimum and maximum are stored implicitly and can be found in constant time

DISADVANTAGES

1. Must have predefined fixed integer ranges
2. Due to their hierarchical structure, VEB trees may require large amounts of memory.
3. Complexity is defined by the size of the universe of values, not by the size of the stored integers

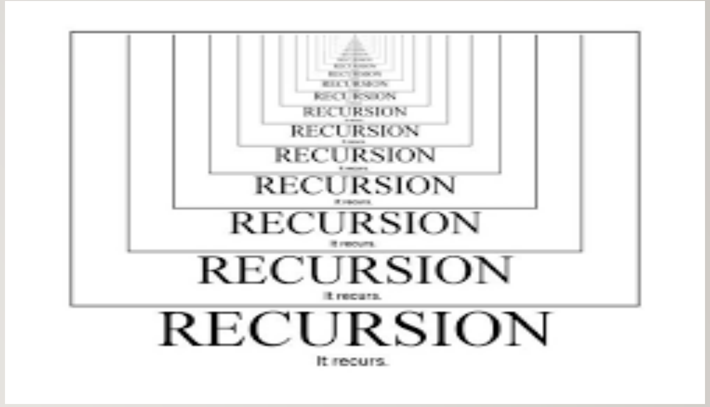
PROTO VAN EMDE BOAS TREE STRUCTURE



VEB IMPLEMENTATION:

1. VEB is defined recursively, where a parent contains a summary VEB tree and child 'clusters' AKA 'galaxies' which are smaller VEB trees
2. VEB requires a predefined universe size *u*, equal to a power of 2 such as *u* = 2^32 for 32bit integers
3. Each child VEB tree has sqrt(*u*) elements, down to the last layer where blocks of 2 are used
4. Utility functions high(*x*), low(*x*) and index(*i, j*) are used to find the cluster in the same layer above *x*, below *x*, and at a specific position
5. The bottom layer of the tree is handled distinctly as a break case for recursive updates, and stores 0s or 1s in the min/max positions
6. Memory complexity of VEB varies significantly based on implementation choices, either with O(*u*), O(*n*lglg*n*) or O(*n*) storage

VEB INSERT:

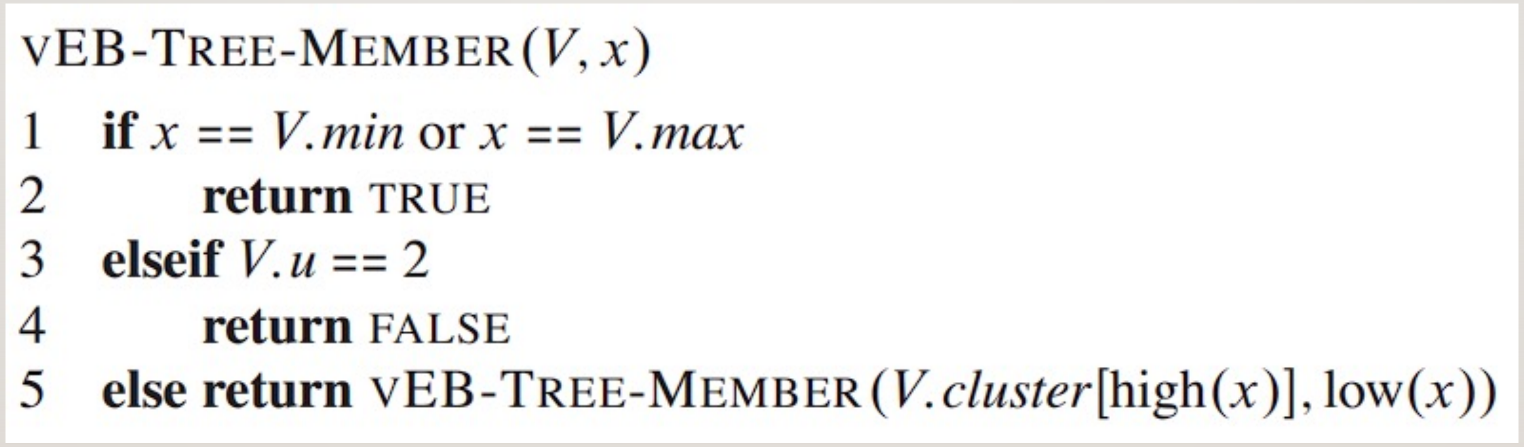


1. If VEB layer is empty, *lazily* insert the value as both the min and the max
2. If insertion value *x* is less than the minimum, set the *x* as the minimum and continue the algorithm with the old minimum value in place of *x*
3. If *x* is greater than the maximum, set the max equal to *x*
4. Insert *x* into the next level of the tree if it hasn't been inserted already

Because minimum and maximum values are not propagated down the tree until absolutely necessary, any follow up insertions after recursive operations will only require constant time

VEB SEARCH:

1. If the current layer min or max equals *x*, return true
2. If the current layer is the last layer and min or max do not equal *x*, return false
3. Otherwise, call search on the next relevant layer



```
vEB-TREE-MEMBER($V, x$)
1  if $x == V.min$ or $x == V.max$
2      return TRUE
3  elseif $V.u == 2$
4      return FALSE
5  else return vEB-TREE-MEMBER($V.cluster[\text{high}(x)], \text{low}(x)$)
```

VEB DELETE:

1. If min and max are the same (only 1 value in the tree), simply delete the min and max
2. If you've reached the bottom layer and the min/max values are different, set min and max both to the remaining element
3. If neither of the above conditions apply, something must be deleted after accounting for changes in minimums after the expected delete. If *x* is equal to min then the next smallest value must be set to min
4. Call delete recursively on the next layer
5. If min is empty, update the corresponding summary vector entry and reset min
6. If *x* is equal to max, reset max to the corresponding max value within the cluster, without updating the summary vector

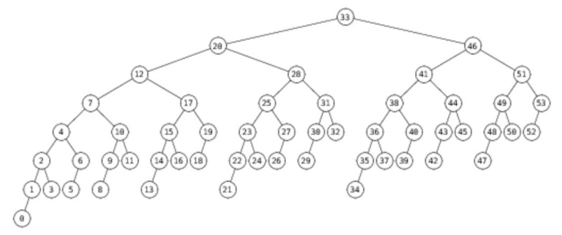
AVL TREE

BALANCED BINARY SEARCH TREE

OVERVIEW

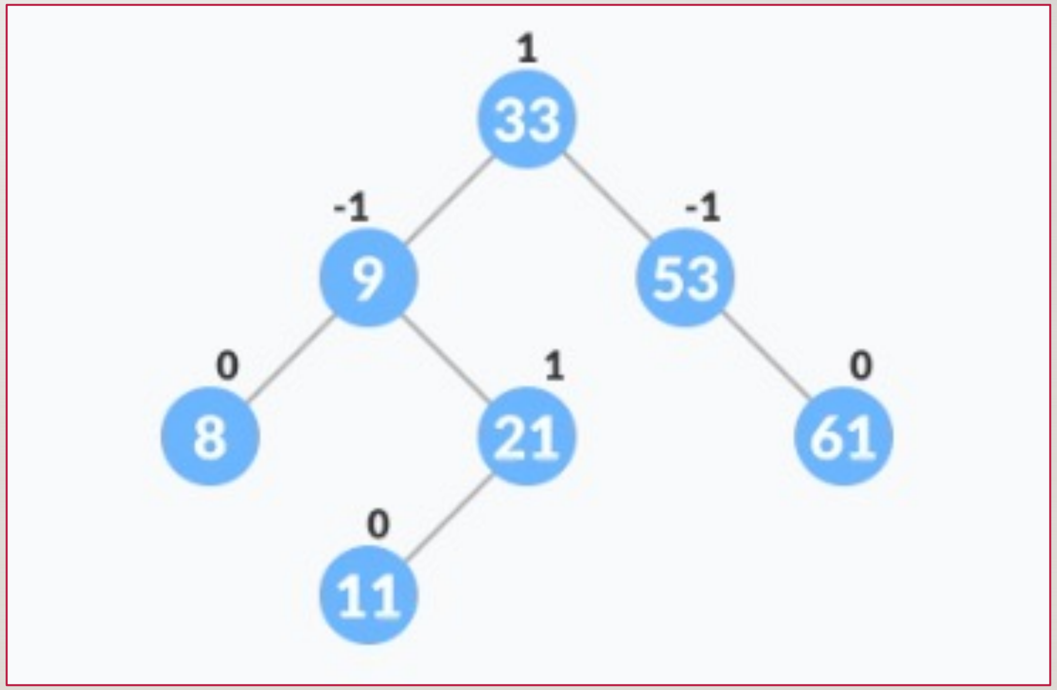
An AVL tree is a self-balancing binary search tree which uses the concept of height balance to ensure that all nodes have the same depth or level, thus making it more efficient than other trees in terms of searching time.

- Invented by and named after Georgy Adelson-Velsky and Landis
- First balanced binary search tree to be invented
- The sub-trees of every node differ in height by at most one.
- Every sub-tree is an AVL tree.



BALANCING

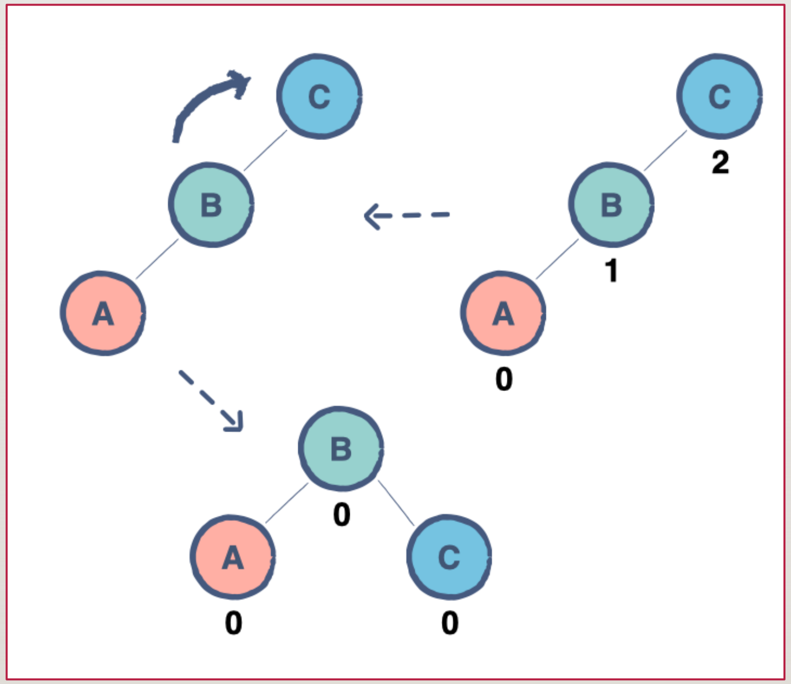
- Each node keeps track of its maximum height
- A node’s height is the maximum distance between it and the leaf node of its subtree.
- Balance Factor is based on the height of the node’s children’s subtrees.
- $Balance = Height\ of\ Left\ Subtree - Height\ of\ Right\ Subtree$
- Each node is allowed to have a balance factor of either -1, 0, or 1.
- Once a node becomes too unbalanced, left/right rotations will need to be performed to rebalance it.

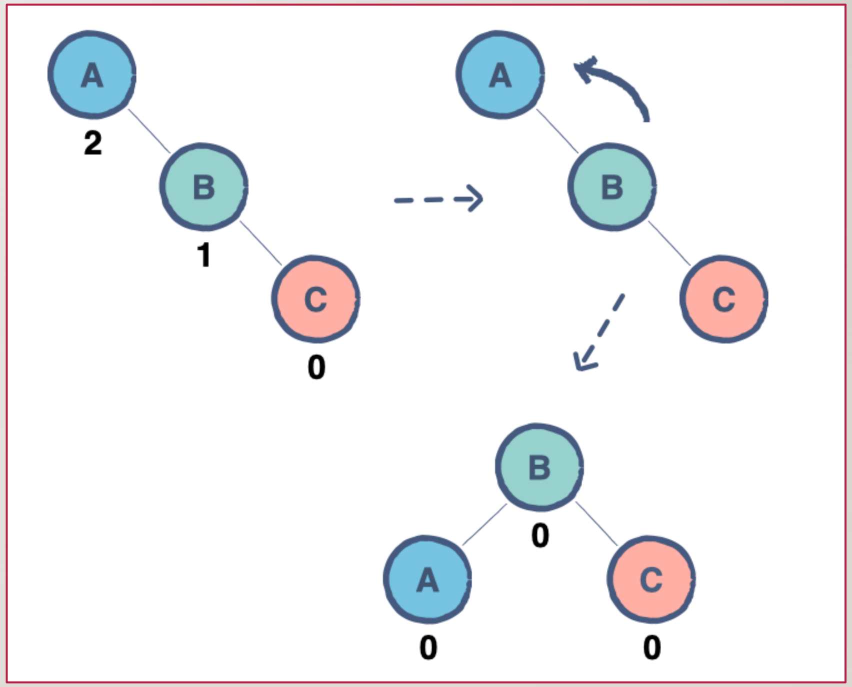


https://www.programiz.com/dsa/avl-tree

RIGHT AND LEFT ROTATIONS

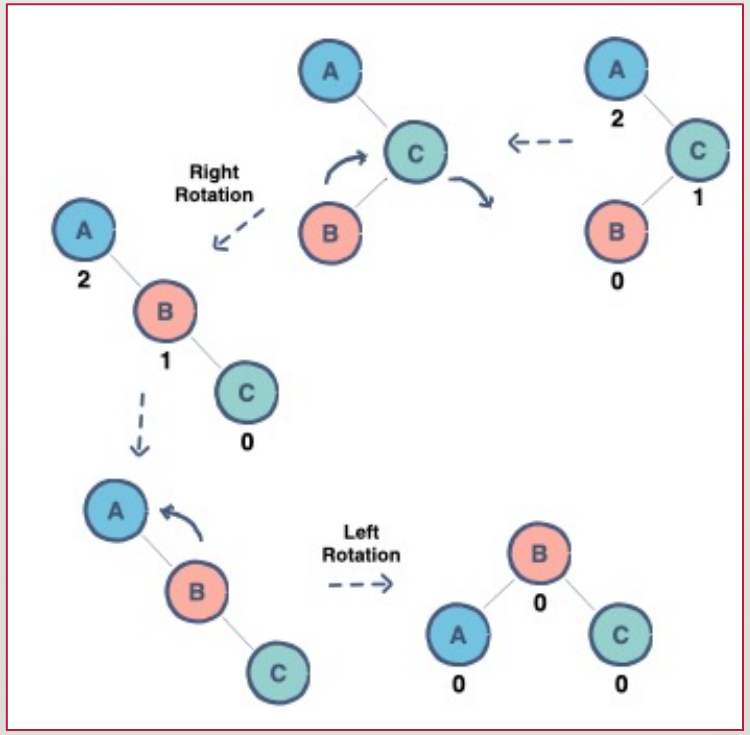
- Imbalance in the right child's right sub-tree, perform a left rotation.
- Imbalance in the left child's left subtree. Perform a right rotation.

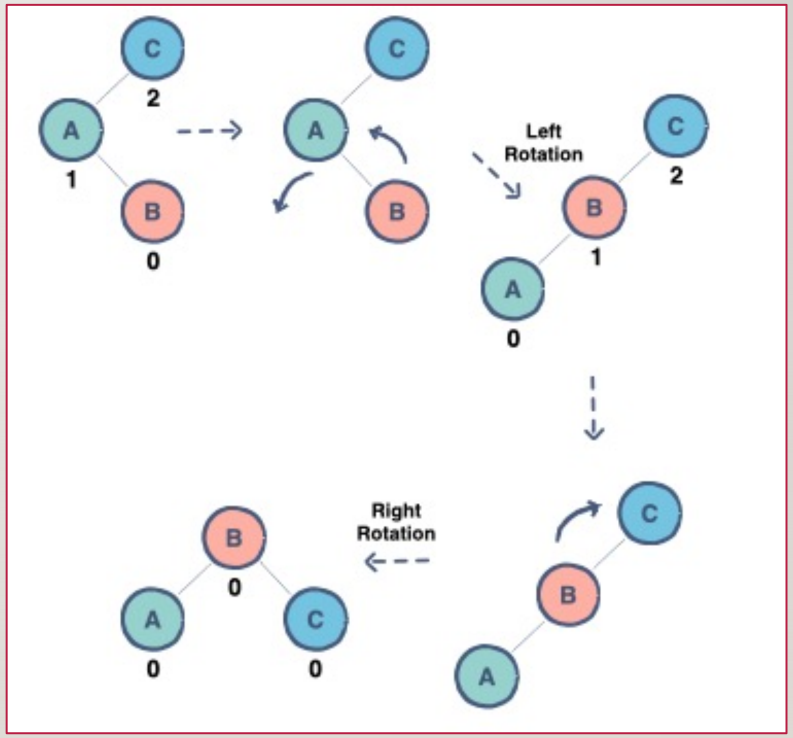




RIGHT-LEFT AND LEFT-RIGHT ROTATIONS

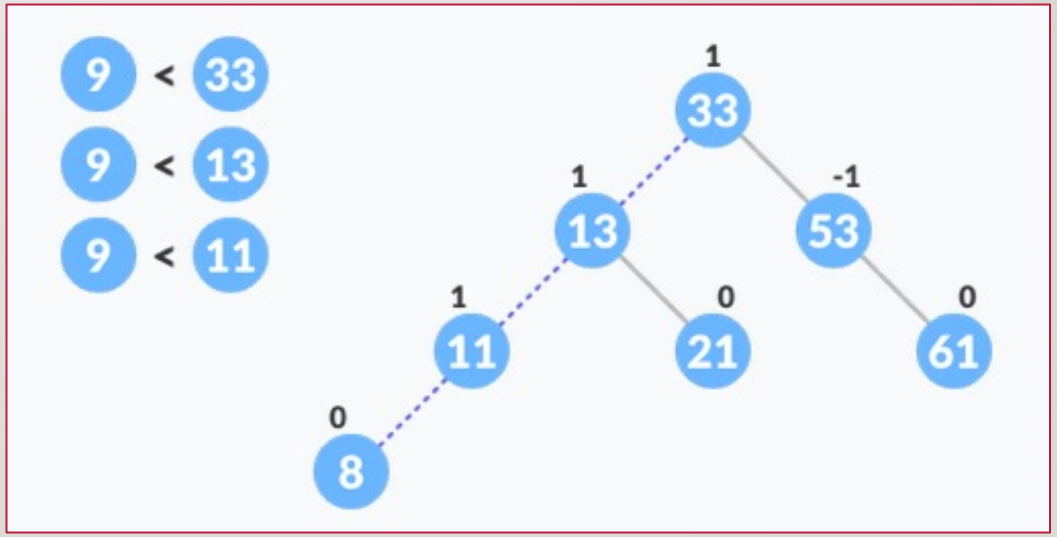
- Imbalance in the right child's left sub-tree, perform a right-left rotation.
- Imbalance in the left child's right subtree, a left-right rotation is required.





INSERT

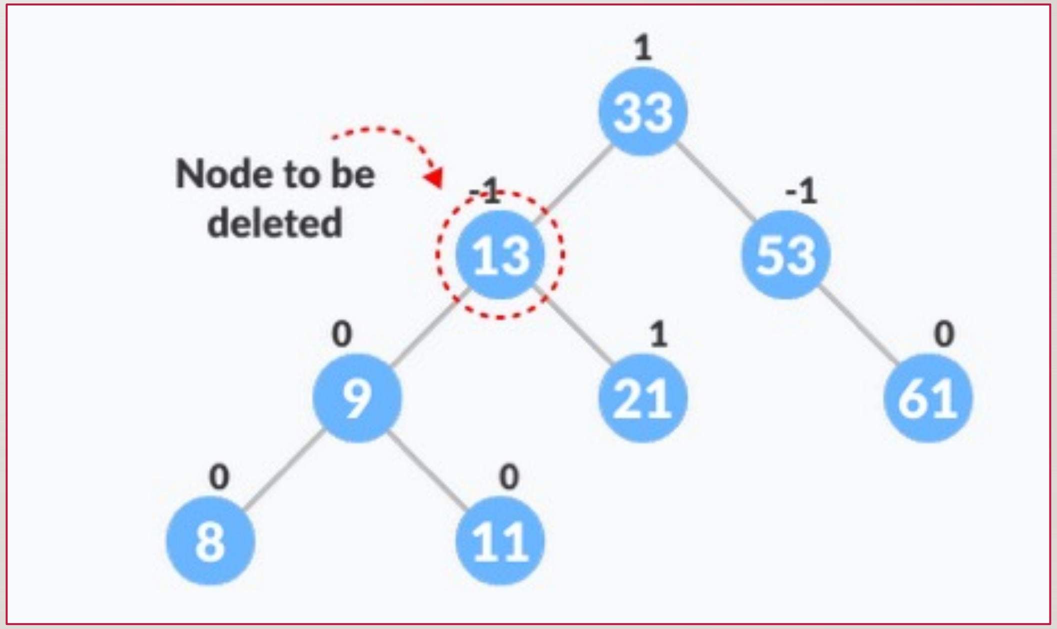
- AVL trees perform the standard binary search tree insert algorithm.
- The new key value is compared to a node at each level until the end of the tree is reached, which is where the node will be inserted.
- Once the new node has been inserted, we must traverse back up the tree to ensure that it is still balanced.
- Once a node becomes too unbalanced, left/right rotations will need to be performed to rebalance it.
- O(log(n))



https://www.programiz.com/dsa/avl-tree

DELETE

- AVL trees perform the standard binary search tree delete algorithm.
- The new key value is compared to a node at each level until we have found the node to be deleted.
- Once the node has been found, we will need to find its successor.
- The successor will replace the node that we want to delete, and the leaf successor will be deleted.
- Once a node becomes too unbalanced, left/right rotations will need to be performed to rebalance it.
- O(log(n))



https://www.programiz.com/dsa/avl-tree

ADVANTAGES & DISADVANTAGES:

- The height of an AVL tree is always kept at most log(n), ensuring that it remains relatively shallow compared to other types of binary search trees.
- Inserting and deleting elements can be quite expensive because the tree must be rebalanced frequently.

COMPARISON:

	VEB TREE	AVL TREE	RED BLACK TREE
Insert	O(lglg(u))	O(lg(n))	O(lg(n))
Delete	O(lglg(u))	O(lg(n))	O(lg(n))
Search	O(lglg(u))	O(lg(n))	O(lg(n))
Successor	O(lglg(u))	O(lg(n))	O(lg(n))
Space	S(u)	S(n)	S(n)

Thank You!