ADVANCED DATA STRUCTURE AND ALGORITHMS

CSPE43

Group -18

Dominic John (033) | Haemanth V (043) | Jeremiah T. (055)

TOPIC: *A simple Performance Analysis and Comparison of 3 popular multi-way search trees: B+ Tree, B* Tree and Van Emde Boas Tree*

OBJECTIVE: -

The aim of our project is to compare the performances of the three important multiway search trees namely B+ tree, B* tree and Van Emde Boas tree with respect to 5 specific cases: -

1. Insertion
2. Deletion
3. Search
4. find min
5. extract min

The m-way search trees are m-ary trees which are generalised versions of binary trees where each node contains multiple elements.

In an m-Way tree of order m, each node contains a maximum of m – 1 elements and m children. An m-Way search tree of height h calls for O(h) number of accesses for an insert/delete/retrieval operation.

Hence, it ensures that the height h is close to logm(n + 1). The advantage is their easy-to-maintain balance.

Multi-way search trees have uses that range from effective faster searching all to way real time applications like effective multi-level indexing in databases and file systems and that is what initially caught our attention.

Through our project, we wish to satiate our curiosity regarding which multiway search tree that we've considered is better and in which scenario. For instance, does B* outperform B+ and if so, when does it do? Does it do it for all orders? For all input sizes?

We've put in an honest and decent bit of effort towards discovering what's better and tried reasoning out the observations from the graphs we got. However, we do admit there might be some errors but we sure would love to hear your feedback on the same.

2. Data structure 1: B+ Tree

2.1 Description/ definition

In a nutshell, a B+ tree is an advanced form of a self-balancing multiway search tree in which all the values are present in the leaf level. It can be viewed as a **B-tree** in which each node contains only keys (not key–value pairs), and to which an additional level is added at the bottom with linked leaves.

A B+ tree consists of a root, internal nodes and leaves. The root may be either a leaf or a node with two or more children

For a B+ tree of degree m: -

- A node can have a maximum of m children.
- A node can contain a maximum of m - 1 keys.
- A node should have a minimum of ⌈m/2⌉ children.
- A node (except root node) should contain a minimum of ⌈m/2⌉ - 1 keys.

2.2. Real Time Application: -

- B+ Tree are used to store the large amount of data which cannot be stored in the main memory.
- Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.
- Thus the primary value of a B+ tree is in **storing data for efficient retrieval in a block-oriented storage context** — in particular, filesystems. This is primarily because unlike binary search trees, **B+ trees have very high fanout** (number of pointers to child nodes in a node,[1] typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the tree.
- The ReiserFS, NSS, XFS, JFS, ReFS, and BFS filesystems all use this type of tree for metadata indexing; BFS also uses B+ trees for storing directories.
- NTFS uses B+ trees for directory and security-related metadata indexing. EXT4 uses extent trees (a modified B+ tree data structure) for file extent indexing

2.3. Requirements/ Limitations: -

The data in B+ trees need not be sorted or such but there are few caveats and restrictions to keep in mind while implementing a B+ tree: -

1. Order must be pre-defined either through console or initializing in the code
2. The maximum order for our implementation is 100.
3. There is no limit to the no of elements you can enter

2.4.1 Operation 1 : Insertion:-

Explanation: -

Inserting an element into a B+ tree consists of three main events:

- searching the appropriate leaf
- inserting the element
- balancing/splitting the tree if required

Algorithm: -

The following steps are followed for inserting an element.

Since every element is inserted into the leaf node, go to the appropriate leaf node (Similar to search function.)

Insert the key into the leaf node.

Case I

If the leaf is not full, insert the key into the leaf node in increasing order. No other change required. node_count will increase by 1.

Case II

If the leaf is full, insert the key into the leaf node in increasing order and balance the tree in the following way, depending on whether it's a lead node or not:-

If leaf node: -

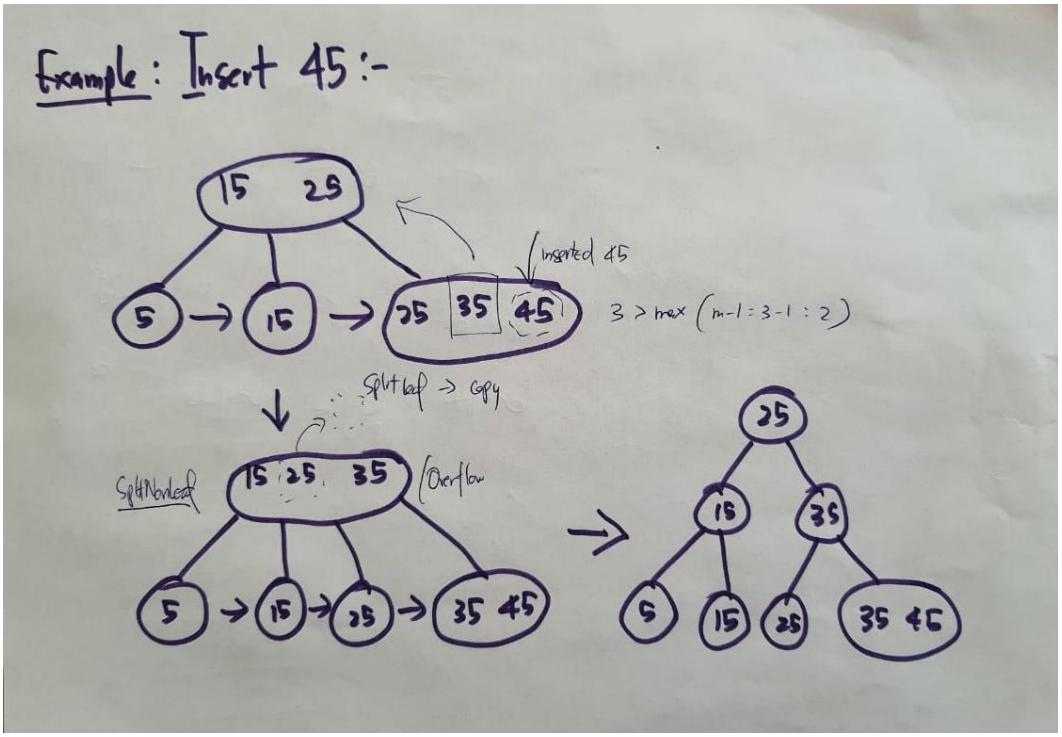
- Break the node at m/2th position.
- Copy m/2th key to the parent node as well. The two halves will be the left and right child of the m/2th element in parent node. (Note: the 2nd half begins with the m/2th element)
- If the parent node is already full, follow steps 2 to 3 but a

If non-leaf node: -

Slight caveat here: when splitting for a non-leaf node, ensure that you move the beginning element of the 1st half and not simply copy it.

- Break the node at m/2th position.
- Move m/2th key to the parent node. The 2 halves formed will be the left and right child of m/2th element
- If the parent node is already full, follow the above steps

Example: -



2.4.2 Operation 2: Deletion: -

Explanation: -

Deleting an element on a B+ tree consists of three main events:

- searching the node where the key to be deleted exists,
- deleting the key
- balancing the tree if required.

Underflow is a situation when there is a smaller number of keys in a node than the minimum number of keys it should hold.

While deleting a key, we have to take care of the keys present in the internal nodes (i.e., indexes) as well because the values are redundant in a B+ tree.

Algorithm: -

First searching for the key to be deleted must be done, followed which the following steps for deletion must be taken in accordance to which case the node containing key to be deleted falls in: -

Case I

The key to be deleted is present only at the leaf node not in the internal nodes.

There are two cases for it:

1. There is more than the minimum number of keys in the node. Simply delete the key. node_count--. Use memcpy() to shift accordingly.
2. There is an exact minimum number of keys in the node. Delete the key and borrow a key from the immediate sibling. Add the median key of the sibling node to the parent.

Case II

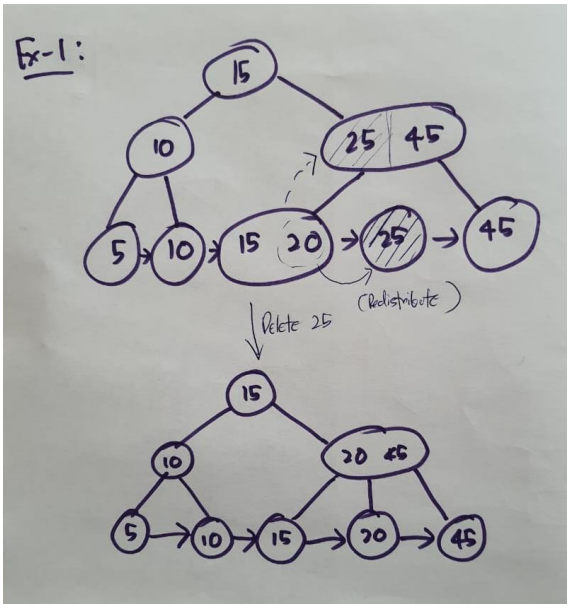
The key to be deleted is present in the internal nodes as well. Then we have to remove them from the internal nodes as well. There are the following cases for this situation: -

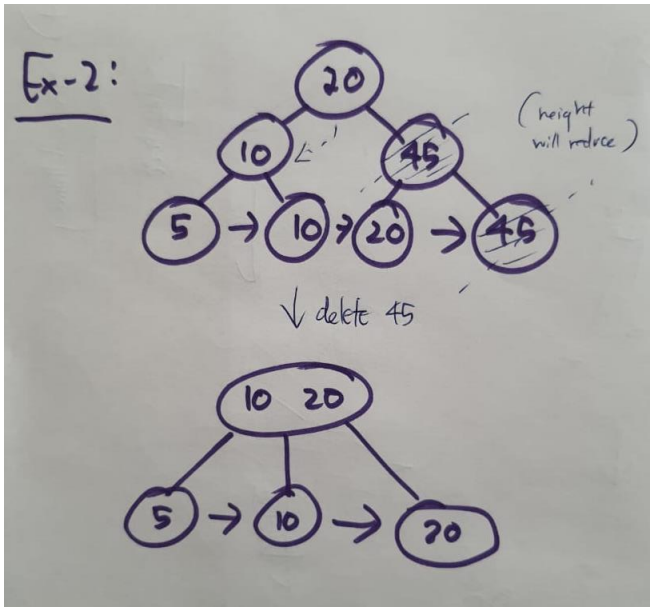
1. If there is more than the minimum number of keys in the node, simply delete the key from the leaf node and delete the key from the internal node as well. Fill the empty space in the internal node with the in order successor.
2. If there is an exact minimum number of keys in the node, then delete the key and borrow a key from it immediate sibling (through the parent). Fill the empty space created in the index (internal node) with the borrowed key.
3. If empty space is generated above the immediate parent node, then after deleting the key, merge the empty space with its sibling.
   Fill the empty space in the grandparent node with the in order successor.

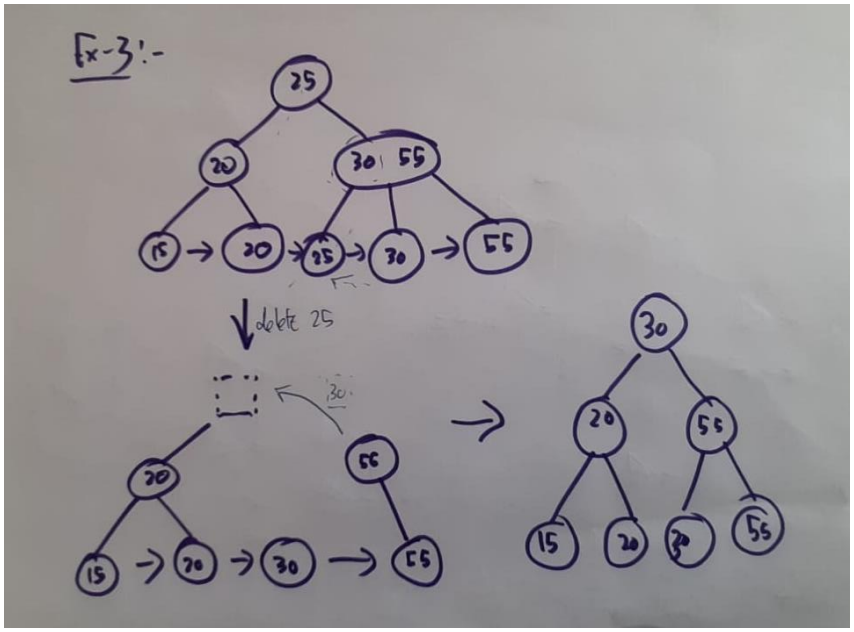
Case III

In this case, the height of the tree gets shrinked. (Ex -2 is a good illustration to get one to grips with this case)

Examples: -







2.4.3. Search: -

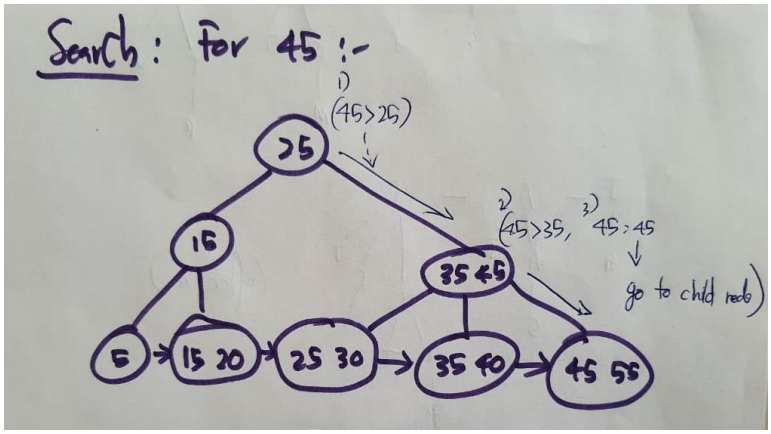
Since it is a multiway search tree with sorted values in every node and with smaller elements as left child and larger elements as the right child, we can use O(logn) search, comparing each key of tree with the element to be searched and move to its left or right descendant depending on the logical relationship bw the current key of tree and the searching value.

Algorithm: -

The following steps are followed to search for data in a B+ Tree of order m. Let the data to be searched be k.

1. Start from the root node. Compare k with the keys at the root node [k1, k2, k3,.....km - 1].
2. If k < k1, go to the left child of the root node.
3. Else if k == k1, compare k2. If k < k2, k lies between k1 and k2. So, search in the left child of k2.
4. If k > k2, go for k3, k4,.. km-1 as in steps 2 and 3.
5. Repeat the above steps until a leaf node is reached.
6. If k exists in the leaf node, return true else return false.

Example: -



2.4.4 Operation 4: Find Min: -

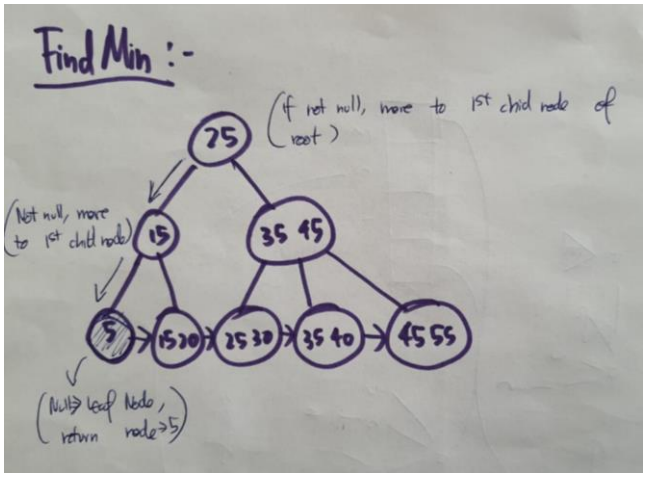
Explanation: -

A fact that we have made use of while implementing findMin() is that the smallest element is always the leftmost node's 1st key no matter what type of tree or data we take. Thus, our implementation involves recursively seeking the 1st child node of every node in conidiation if not null. If null, then it means we have reached the last node containing the minimum element.

Algorithm:-

1. Start
2. Recursively loop through the tree starting from the node
3. Check if node is not null
   If not null, then call findmin recursively on its leftmost first child
   If null, then we have reached the leaf node, move to step 4
4. Return the first key in the node. This will be the minimum value
5. End.

Example: -



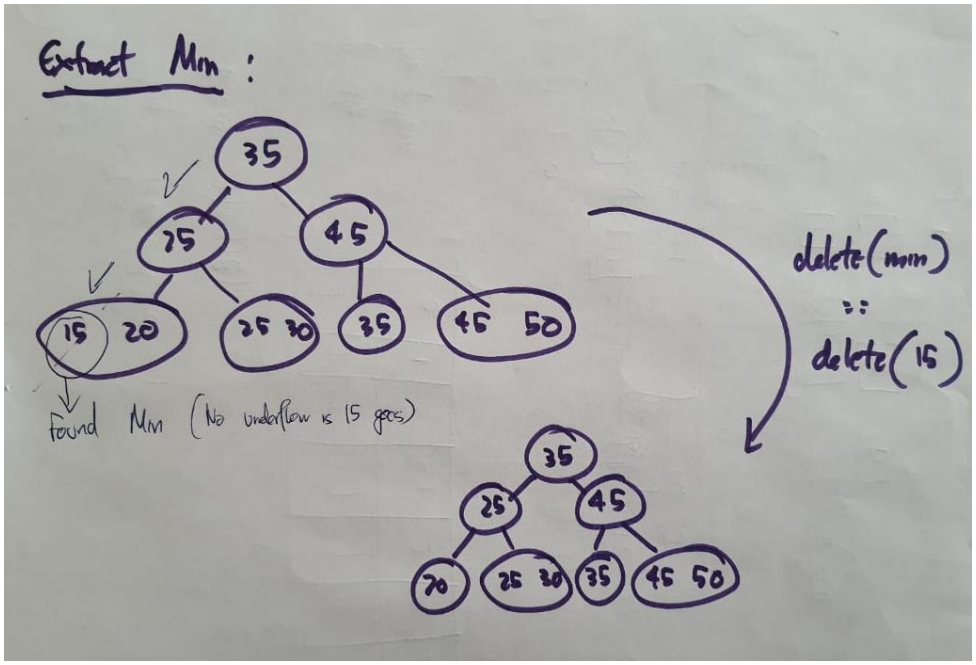
2.4.5. Operation 5 : Extract Min

Extract Min involves two component functions: 1. First run the findmin() function to find the minimum value in the tree. Then, 2) Run the delete node function for that particular key of minimum element. The heavier function is deletenode, hence its time complexity revolves on deletenode().

Algorithm: -

1. Start
2. Find the minimum value of the tree using the search function
3. Run the deletenode() function to extract the minimum element.(The delete node simply deletes the given element)
4. End

Example: -



---

3. Data Structure: B* Tree: -

3.1 Definition: -

B*-tree of order m is a self-balancing multiway search tree that is either empty or that satisfies the following properties:

- Every n-node B*-tree has height O(lg n)

- If an internal B*-tree node contains x keys, then it has x+1 children. The keys in the node serve as dividing points separating the range of keys handled by it into n+1 subranges, each handled by one of its children.
- The keys are stored in each node in an increasing fashion.
- All leaves have the same depth, which is the tree's height h
- Nodes have lower and upper bounds on the number of keys they can contain. For a B*-tree of order m:
    - Every node other than the root must have at least ceil((2*m-1)/3.0)-1 keys and at least ceil((2*m-1)/3.0) children. If the tree is non-empty, the root must have at least one key.
    - Every node other than the root must have at most m-1 keys and hence can have at most m children.
    - The root node must contain at least 1 key and at least 2 children (if root node is not the only node in the tree).
    - The root node must contain at most 2*floor((2*m-2)/3.0) keys and at most 2*floor((2*m-2)/3.0) + 1 children.

	Root Node	Non-root node
Maximum keys	2*floor((2*m-2)/3.0)	m-1
Maximum subtrees	2*floor((2*m-2)/3.0)+1	m
Minimum keys	1	ceil((2*m-1)/3.0)-1
Minimum subtrees	2	ceil((2*m-1)/3.0)

3.2. Real Life Applications: -

- B*-trees can be used to implement many dynamic-set operations in time O(lg n) as height of the tree is O(lg n) for nodes.
- B*-trees are balanced search trees designed to work well on disks or other direct access secondary storage devices as they are better at minimizing disk I/O operations.
- B*-trees are used to store information in many database systems. Advantages of using a B*-tree for databases are that a B*-tree:
    - keeps keys in sorted order for sequential traversing
    - uses a hierarchical index to minimize the number of disk reads
    - uses partially full blocks to speed up insertions and deletions
    - keeps the index balanced with a recursive algorithm

  In addition, a B*-tree minimizes waste by making sure the interior nodes are at least two-third full. A B*-tree can handle an arbitrary number of insertions and deletions.

- B*-trees are also used in the HFS (a proprietary file system developed by Apple Inc. for use in computer systems running Mac OS) and Reiser4 file systems as they allow quick random access to an arbitrary block in a particular file.

3.3. Requirements: -

- The order of the tree must be provided.
- The splitting factor (typically 2/3rd) must be decided upon before hand

Benefits and Limitations:

- The advantage of using B* trees over B-trees is a unique feature called the '2-3' or 'two-to-three' split. By this, the minimum number of keys in each node is not half the maximum number, but two-thirds of it, making data far more compact. Thus, The B* tree balances more neighbouring internal nodes to keep the internal nodes more densely packed.
- B* tree guarantees that storage utilization is at least 66%, while requiring only moderate adjustment of the maintenance algorithms. It should be pointed out that increasing storage utilization has the side effect of speeding up the search since the height of the resulting tree is smaller.
- However, the disadvantage of B* trees is a complex deletion operation. The difficulties in practically implementing a B* algorithm contribute to why it's not as regularly used as its B and B+ counterparts.

Operation 1: Insertion: -

Explanation:

The B* tree balances more neighbouring internal nodes to keep the internal nodes more densely packed. This variant ensures non-root nodes are at least 2/3rd full instead of 1/2.

As the most-costly part of operation of inserting the node in B-tree is splitting the node, B*-trees are created to postpone splitting operation as long as they can. To maintain this, instead of immediately splitting up a node when it gets full, its keys are shared with a sibling that is not full. This spill operation is less costly to do than split, because it requires only shifting the keys between existing nodes, not allocating memory for a new one.

Algorithm:

All insertions start at a leaf node.

1) Search the tree to find the leaf node where the new element should be added using Binary search tree logic or search logic described below
2) Insert the new element at the appropriate position in the node so that the keys remain sorted.
3) If the node contains at most maximum allowed number of elements, return
4) Else if the node overflows or exceeds maximum capacity then
    4.1 If the right sibling exists and is not full (has lesser than maximum number of keys allowed)
        - shift all the keys and children in the right sibling to the right
        - move the previous parent to the beginning of the right sibling.

- shift the last element of the current node to the position of the parent
- move the last child node of current node at the beginning of right sibling's children

4.2 Else if the left sibling exists and is not full (has lesser than maximum number of keys allowed) then

- move the parent element to the end of the left sibling
- move the first element in the current node to the parent
- move the first child of the current node to the end of left sibling's children
- shift all the keys in the current node one position to the left.

4.3 Else if siblings are also full (whichever exists) and node is not the root node then

- Create an array of keys and children containing the keys and children of the current node, one of its siblings(left/right) and the parent key alone. The array of keys should be in sorted order.
- split the current node, one of its siblings (left/right) and the parent key into 3 nodes and 2 parent keys such that
    - Insert the first (2m – 2)/3 keys and (2m-2)/3+1 children in the first node
    - The key at index (2m – 2)/3 in the keys array is stored as parent1
    - Insert the next (2m – 1)/3 keys and (2m-1)/3+1 children in the second node
    - The key at index (4m)/3 is stored as 'parent2'
    - Insert the remaining 2m/3 keys and 2m/3+1 children in the arrays in the third node
    - Make the three nodes the children of the two new parent
    - Repeat step 4 for the parent if the parent also overflows

4.4 Else if node is the root node then

- create two new child nodes and an array of keys and children containing all the keys and children of the root node with keys in sorted order.
- Split the root node into the 2 new child nodes and a parent node with a single key.
    - The first floor((2*m-2)/3.0) keys and floor((2*m-2)/3.0) + 1 children are stored in the left child
    - The key at index floor((2*m-2)/3.0) is stored in the parent
    - the remaining floor((2*m-2)/3.0) keys are stored in the right child of the root node.
    - Store the pointers to the two new child nodes in the root node's children list.

Consider an empty B^* tree of order 4

Max keys (root node) $= 2 \times \left\lfloor \frac{2m-2}{3} \right\rfloor$

$= 2 \times \left\lfloor \frac{8-2}{3} \right\rfloor$

$= 4.$

Max keys (non-root node) $= m-1$

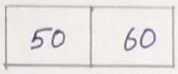
$= 3.$

Let us insert 50, 60, 70, 80, 90, 100, 110, 120

50 →

Root is empty. So we just insert the value

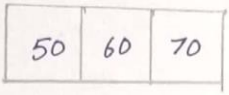
60 →



Since root is a leaf node and is not full we just insert 60.

Number of keys = $2 \leq 4$. So we finish

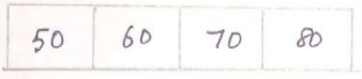
70 →



Since root node is a leaf node, we just insert 70 into the sorted array of keys

Number of keys = $3 \leq 4$. So we finish

80 →

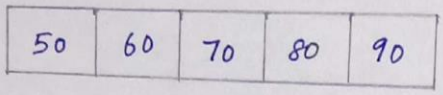


Since root is leaf node we simply insert 80

Number of keys = $4 \leq 4$. So we finish

Root node is full

90 →



Since root is leaf node we just insert

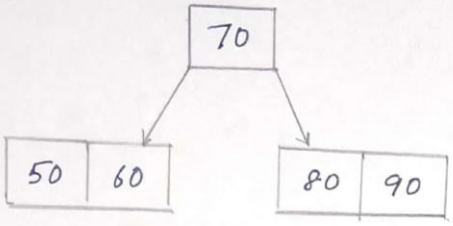
Number of keys = $5 > 4$.

So root node has overflown.

We now Split the root as in step 4.4.

Left child will have first $\left\lfloor \frac{2m-2}{3} \right\rfloor = 2$ keys

3rd key goes to the parent node and remaining keys to the right child.

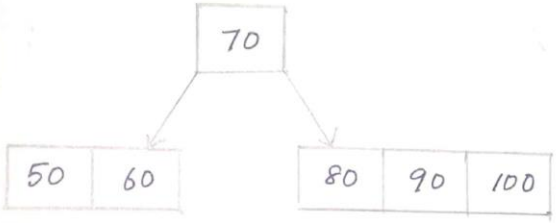


100 →

$100 > 70$

So we go to the right child

The child node is a leaf. So we simply insert 100



Number of keys after insertion = $3 \leq 3$

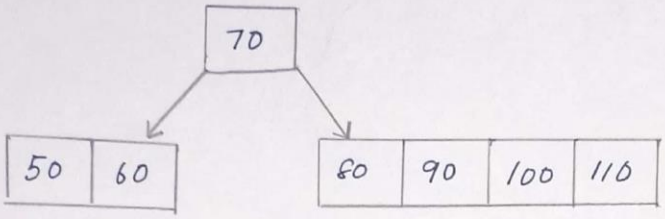
So we finish

110 →

110 > 70

So we go to the right child

The node is a leaf node. So we simply insert



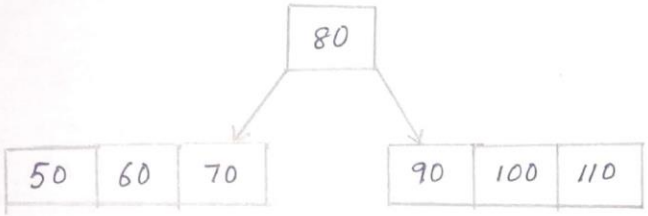
Number of keys after insertion = 4 > 3

So the node has overflown.

It does not have a right sibling. So we check the left sibling

number of keys = 2 < 3. So it is not full

So we rearrange the keys as in step 4.2



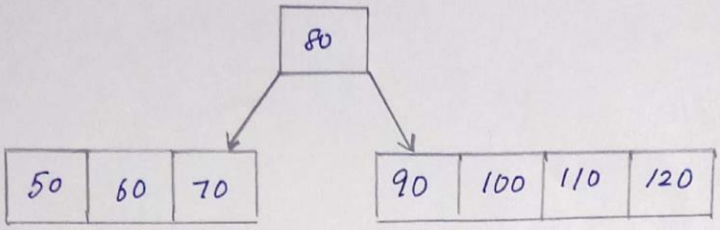
120 →

120 > 80

So we go to the right child

Right child is leaf node

So we simply insert 120.



Number of keys after insertion = 4 > 3.

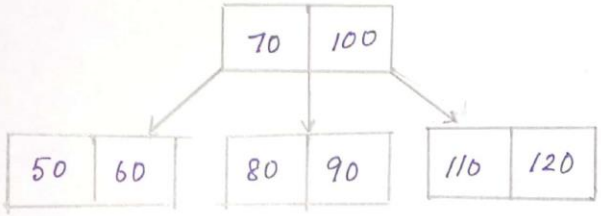
So the node has overflown.

It does not have a right sibling, left sibling is also full and it is not the root. So we split the 2 leaf nodes into 3 leaf nodes using step 4.3.

child 1 has $\lfloor \frac{2m-2}{3} \rfloor$ = 2 ~~not~~ keys

child 2 has $\lfloor \frac{2m-1}{3} \rfloor$ = 2 keys

child 3 has $\lfloor \frac{2m}{3} \rfloor$ = 2 keys



Parent node has 2 keys $\leq$ 4.

So it has not overflown and hence we finish

Operation 2: Deletion: -

Explanation:

Deletion from a B*-tree is analogous to insertion but a little more complicated, because we can delete a key from any node—not just a leaf—and when we delete a key from an internal node, we will have to rearrange the node's children. As in insertion, we must guard against deletion producing a tree whose structure violates the B*-tree properties. Just as we had to ensure that a node didn't get too big due to insertion, we must ensure that a node doesn't get too small during deletion. Just as a simple insertion algorithm might have to back up if a node on the path to where the key was to be inserted was full, a simple approach to deletion might have to back up if a node (other than the root) along the path to where the key is to be deleted has the minimum number of keys already.

We always delete a value only from the leaf node. Even if the key to be deleted is found in an internal node we move a key from a leaf node to its place and delete that key from the leaf node. If the node has less than the minimum required number of keys after deletion then we perform rebalancing. Rebalancing starts from a leaf and proceeds toward the root until the tree is balanced. If deleting an element from a node has brought it under the minimum size, then some elements must be redistributed to bring all nodes up to the minimum.

Algorithm:

To delete a key k from B*-tree we follow the following procedure:

1. We first search for the node that contains the key k to be deleted using search logic (similar to searching in a Binary search tree)
2. If key is not in the tree then we return
3. Else we consider the following cases for deletion:
   3.1. Deletion from a leaf node
      - Simply delete it from the node by shifting all the keys after it one position to the left.
      - If underflow happens, rebalance the tree as described in step 4
   3.2. Deletion from an internal node
      - Find the smallest element greater than k present in any leaf node of the right subtree of the key k. This key is called the greater_of_minor key of k (similar to inorder successor).
      - Replace the key k in the node with the key greater_of_minor
      - Delete the key greater_of_minor from the right subtree of the deleted key by repeating this procedure
4. If the current node has less than minimum number of keys required then
   4.1. If the right sibling exists and has more than minimum number of elements
      - shift the parent element to the end of the current node
      - move the first element in the right sibling to the parent
      - move the first child of the right sibling to the end of current node's children
      - shift all the keys in the right sibling one position to the left.
   4.2. Else if the left sibling exists and has more than minimum number of keys

- shift all the keys and children in the current node one position to the right
- insert the previous parent at the beginning of the current node.
- move the last element of the left sibling to the position of the parent
- move the last child node of left sibling to the beginning of current node's children

4.3. Else if parent of current node is root and root has only 1 key then:

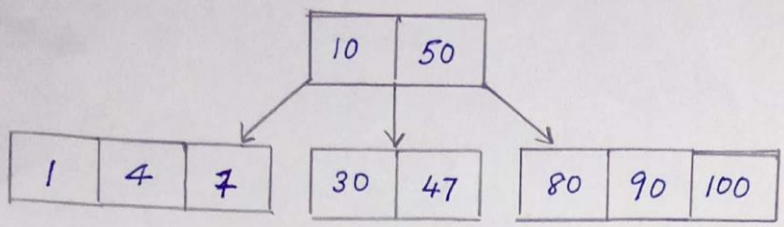
- Merge all the keys of the two children and parent and store them in the root node
- Make all the children of the left and right nodes as the children of the root node
- Delete the previous two children of the root node

4.4. Else:

- Create an array of keys and children with all the keys and child nodes of the current node, its left and right siblings (if the node is the leftmost or rightmost child consider the 2 nearest siblings i.e if the node is leftmost child consider the right sibling and right sibling of the right sibling and similarly consider the 2 left siblings for rightmost node) and the 2 parent keys. Let the size of the array of keys be n
- Merge the 3 child nodes and 2 parent keys into 2 child nodes and a single parent key such that:
    - Store the first n/2 keys and n/2+1 children in the first child node
    - Store the key t index n/2 in the array in the parent
    - Store the remaining n-n/2-1 keys and n-n/2 children in the second child node
    - Delete the third child node and corresponding parent key from the parent by shifting all the elements after them in the array one position to the left
    - Repeat the step 4 for the parent node if there is an underflow

<u>Example: (Attached below)</u>

Consider the following B^* tree of order 4 :



Delete 4, 50, 10, 30, 1, 80

Minimum keys (root node) = 1

Minimum keys (non-root node) $= \left\lceil \frac{2m-1}{3} \right\rceil - 1$

$= \left\lceil \frac{8-1}{3} \right\rceil - 1$

$= 3 - 1$

$= 2$

Delete 4 :

$4 < 10$

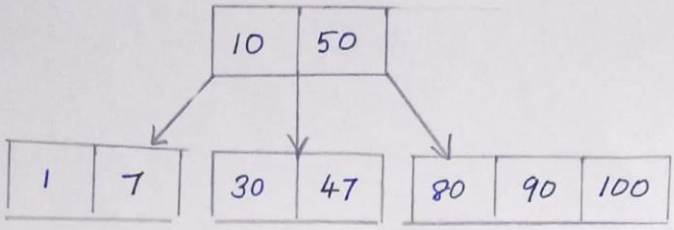
Go to left subtree of 10

~~left~~ child ~~is leaf node.~~

Search for 4 in left child

$4 > 1$, $4 = 4$ Found.

Since left child is leaf node, simply delete 4.



Node after deletion has 2 keys $\geq$ 2. (minimum keys).
So we finish

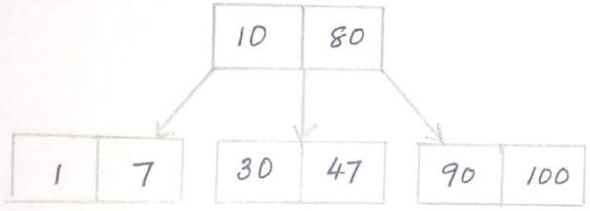
Delete 50

$10 < 50$, $50 = 50$ Found

The node is not a leaf node

So we find the smallest element greater than 50 in the right subtree which is 80.

Replace 50 by 80 and delete 80 from the right subtree.



Leaf node after deleting 80 has 2 $\geq$ 2 keys. So we finish

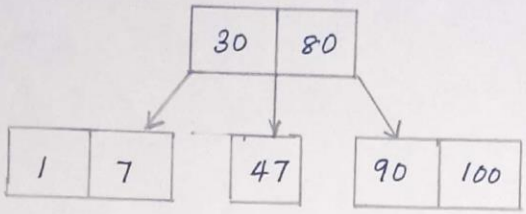
Delete 10

10 = 10 (Found)

Node is not a leaf node

Greater of minor of 10 is 30.

Replace 10 with 30 and delete 30 from its right subtree.



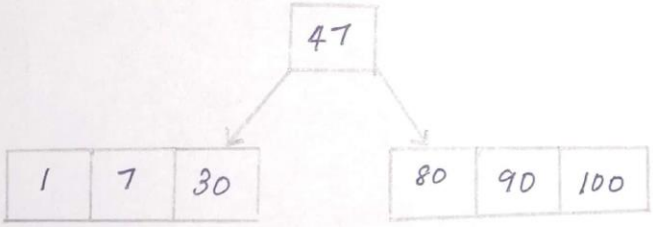
Leaf node after deleting 30 has 1 key < 2 (minimum) so we need to rebalance

Right sibling and left sibling have minimum keys (= 2). Parent is root node but it has more than 1 key, so we perform 2-3 merge as in step 4.4.

$n = 2+1+1+1+2 = 7.$

child 1 has $n/2 = 7/2 = 3$ keys (first 3 keys).

child 2 has $n - n/2 - 1 = 7-3-1 = 3$ keys. (last 3 keys)



Parent node i.e root node has 1 key ≥ 1

so we finish

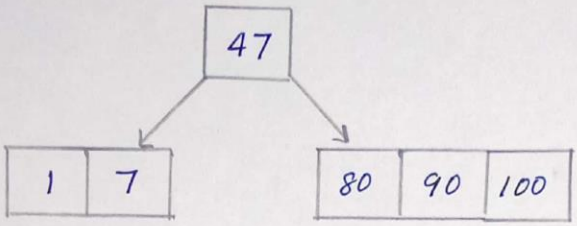
Delete 30

30 < 47

Go to left child

30 > 1, 30 > 7, 30 = 30 Found

Node is leaf node. So just delete 30.



Number of keys after deletion = 2 ≥ 2 keys

So we finish

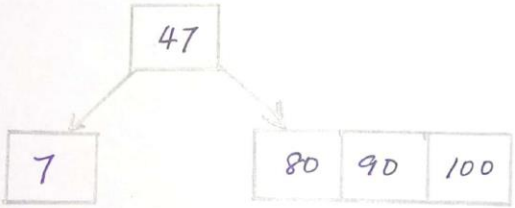
Delete 1

1 < 47

Go to left child

1 = 1 (Found)

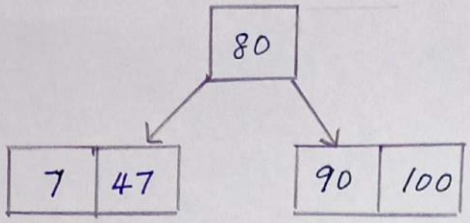
Node is leaf node. So we just delete 1.



Number of keys after deletion = 1 < 2 keys

Right sibling exists and has 3 > 2 keys.

So we borrow a key from ~~a~~ right sibling



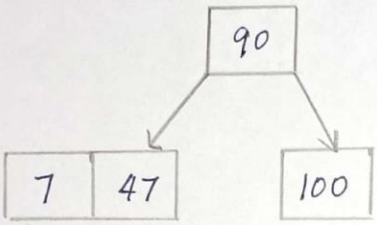
Delete 80.

80 = 80 (Found)

Node is root node (non-leaf node)

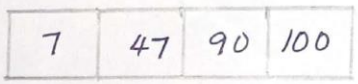
Greater of minor of 80 = 90

Replace 80 with 90 and delete 90 from right subtree.



Number of keys after deleting 90 = 1 < 2 keys.
Right sibling does not exist. Left sibling has minimum number keys (2).

Parent is root node and has only 1 key.
So we perform merge root as in step 4.3



Operation 3: Search: -

Explanation:

Searching a B*-tree is much like searching a binary search tree, except that instead of making a binary, or "two-way," branching decision at each node, we make a multiway branching decision according to the number of the node's children. More precisely, at each internal node with x keys, we make an *x+1*-way branching decision.

Starting at the root, the tree is recursively traversed from top to bottom. At each level, the search reduces its field of view to the child pointer (subtree) whose range includes the search value. A subtree's range is defined by the values, or keys, contained in its parent node. These limiting values are also known as separation values.

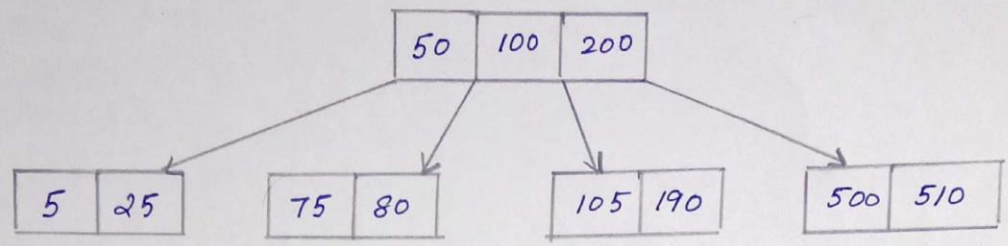
Algorithm:

To search for a key k we do the following:

1) Start searching from root node
2) If k is present as a key in current node return true.
3) If k is not present in current node then
    - 1.1 If node is a leaf node return false
    - 1.2 Else search for k in the left child of the first key in root node which is greater than k i.e find an index i such that node->keys[i] > k and node->keys[i-1] <= k (if it exists) and search for k in the child node at index i (left child of node->keys[i]). If all keys in the node are lesser than k, then search for k in the rightmost child using the same procedure.

Example: (Attached below)

Consider the following B* tree of order 4.



Search for 100, 75 and 300

i) 100

100 > 50, 100 = 100 (Found)

ii) 75

75 > 50, 75 < 100

Go to left child of 100

75 = 75 (Found).

iii) 300

300 > 50, 300 > 100, 300 > 200

All nodes are lesser than 300

so we go to rightmost child node

300 < 500 and ~~node~~

so we go to left child

But left child = NULL (node is leaf node)

So Not found.

Operation 4: Find min: -

Explanation:

This operation finds the minimum valued key in the B*-tree and the node that contains it.

Since a B*-tree is a multiway search tree, the left-most node in the tree has the minimum element in the entire tree and the first key in it is the minimum key as keys within a node are also arranged in increasing order

Algorithm:

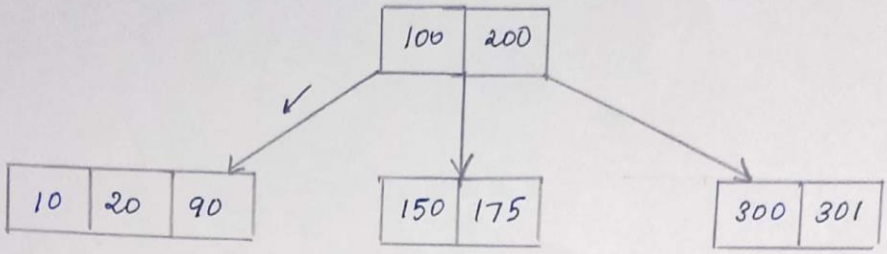
1) Start from root node
2) If current node is leaf node return the current node (the first key in this node i.e current_node->keys[0] is the minimum key).
3) Else repeat the procedure for the leftmost child of current node.

Example: (Attached below)

Consider the following B^* tree of order 4.

Find the minimum key/node

1.



Leaf node

↓

Minimum node

Minimum key is

key[0] = 10

Node - |100|200| Not a leaf node

Go to left child

Node - |10|20|90| Leaf node

Return this node as we are at the leftmost node

Minimum key is key[0] = 10 as keys are arranged in increasing order in the node.

Operation 5: Extract min: -

Explanation:

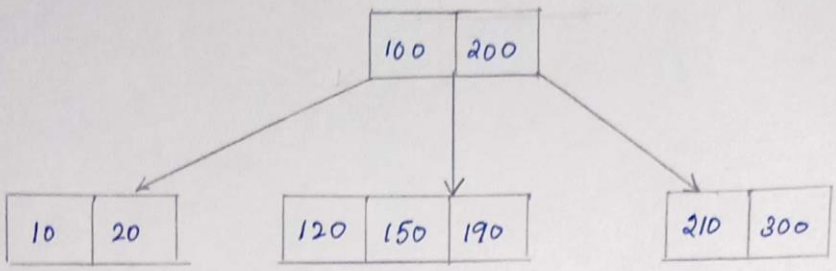
Delete the minimum valued key from the tree

Algorithm:

1) Find the minimum key in the tree using the FindMin procedure described above
2) Delete the minimum key found in the first step by calling the delete function described above.

Example: (Attached below)

Consider the following B* tree of order 4 and extract minimum from the tree.



Find Minimum :

Node - [100 | 200]

Not a leaf node. Go to left child

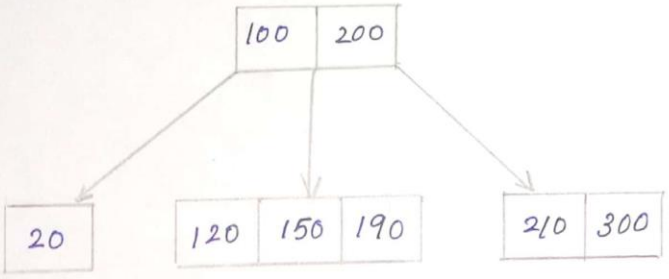
Node - [10 | 20]

Leaf node. Stop.

Delete Minimum -

Delete key[0] = 10

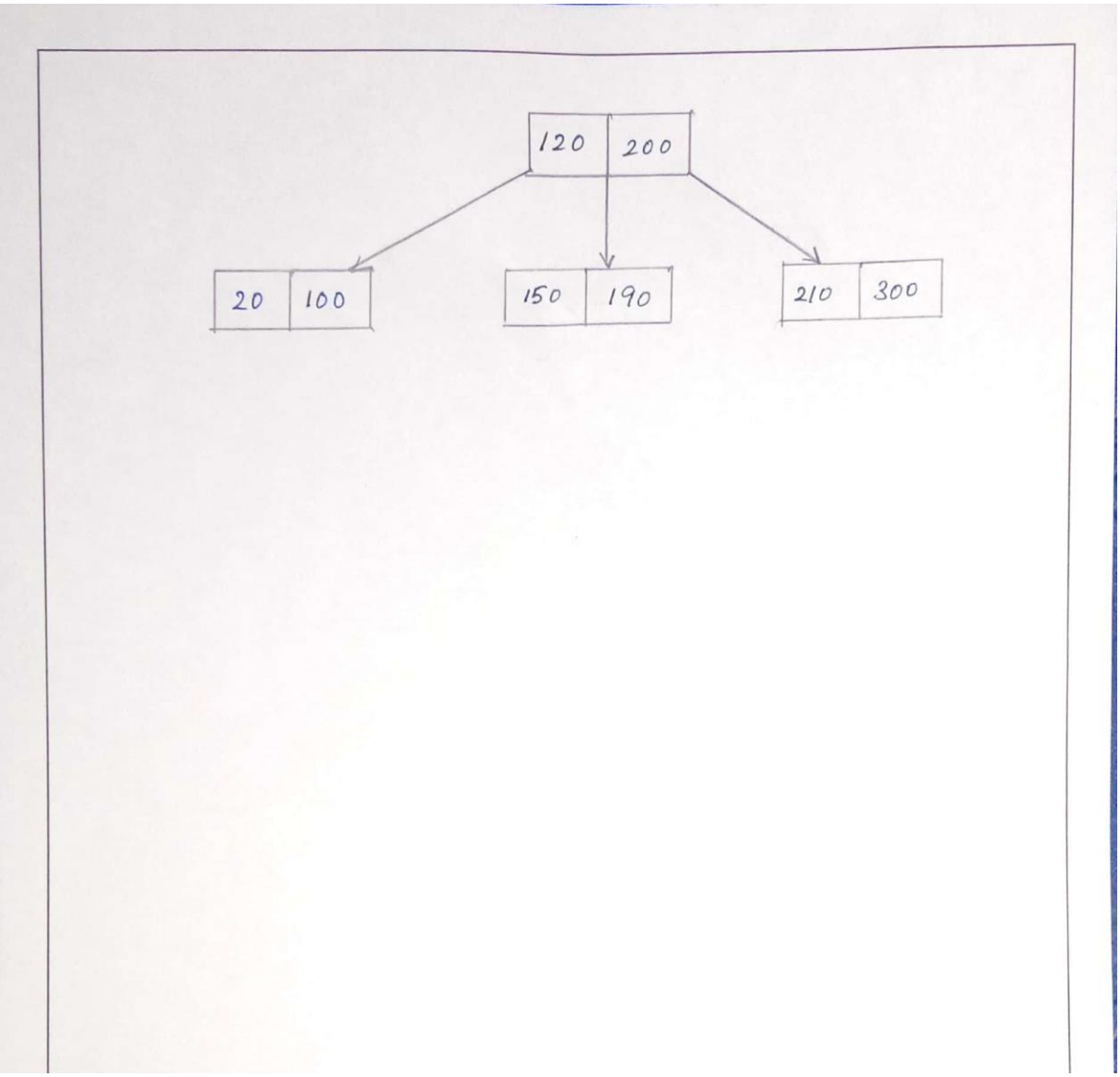
Node is leaf node. So simply delete 10



Number of keys in node after deletion = 1 < 2

So we redistribute the key

Right sibling exists and has more than 2 keys (= 3 keys). So borrow a key from right sibling



4.Data Structure III: Van Emde Boas Tree: -

4.1 Description/ definition:

- The Van Emde Boas tree (VEB tree) is a multi-way search tree, that can store elements from 0 to u-1 without any duplicates, where u is called the universal set size. At each level, the node points to $\sqrt{u}$ cluster nodes, and each cluster node will have a new universal size u'= $\sqrt{u}$, and points to $\sqrt{u'}$ cluster nodes and so on until $u = 2$, thus the degree of the tree varies at each level.

  However, the above description is for the prototype VEB tree, and we would

then need to have the size of the tree as $u = 2^{2^k}$ to be able to take the square root successively, where k is an integer greater than 0. So, it is optimised further to take $u = 2^k$ in the actual VEB tree, which is accomplished by taking an upper square root and a lower square root, whose result is u when multiplied together.

- Each node that points to cluster contains a summary member, which tells if a cluster is empty or not. This summary is updated when a node is inserted or deleted from the cluster.

- Another small modification from the prototype VEB that was done in the improved VEB was the addition of min and max data members. These store the value of minimum and maximum element in the subtree. Note that the value stored in minimum and maximum data member is not the actual value, but rather the value by which it is referenced by.

  An element, say 'x', is not referred using its value in the children subtrees. Instead we use the lowest $\sqrt{u}$ significant bits of x to refer to it in the children subtrees (calculated in the function low(x)). And the highest $\sqrt{u}$ significant bits of x to find in which cluster the x belongs to (calculated using the function high(x)). And the index function is used to get the value x, for the given high(x) and low(x) values.

4.2 Real Time Application /Where it is used?

It can be used as a set with a fixed universal set, as it can insert and delete elements that are not duplicates. Another application is in double ended priority queues, because it stores the minimum and maximum at the root node all times.

4.3. Requirements/ limitations

The limitations with using the VEB tree are:

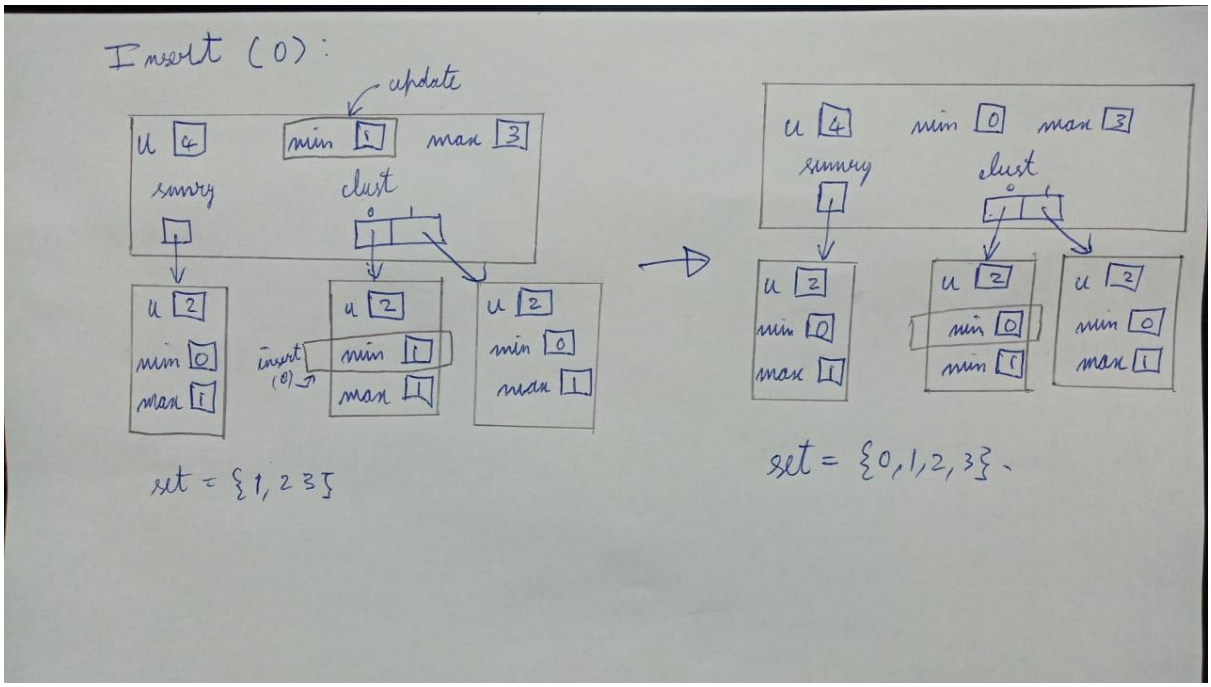
a) Once a size for the universal set is chosen, then is not possible to insert an element beyond that range, without reconstructing the tree.
b) The universal set size u must be a power of 2.

Operation I: Insertion: -

- Explanation

  When we insert an element, either the cluster that it goes into already has another element or it does not. If the cluster already has another element, then the cluster number is already in the summary, and so we do not need to make a recursive call. Otherwise, the element is just inserted, by updating the min and max members.

- Algorithm
  1) If min= NIL, set min and max to key, else go to step 2
  2) If key is less than min, the swap them. After that, if node is not a leaf node, and key is not present in the cluster, then update summary, min and max to contain that element, but if a key is present, then recurse further, with low(key) as the new index.
  3) If the key is greater than the max, set it as the new max
- Example



Operation 2: Deletion: -

- Explanation

  If the VEB tree V contains only one element, then it's just as easy to delete it as it was to insert an element into an empty VEB tree: just set *min* and *max* to NIL. Otherwise, V has at least two elements. Then we test whether V is a base-case and, if so, we set *min* and *max* to the one remaining element. If V has two or more elements and that u = 4 then, we will have to delete an element from a cluster. The element we delete from a cluster might not be x, however, because if x equals *min*, then once we have deleted x, some other element within one of V 's clusters becomes the new *min*, and we have to delete that other element from its cluster.

  If this test reveals that we are in this case, then we set *first-cluster* to the number of the cluster that contains the lowest element other than *min*, and then set x to the value of the lowest element in that cluster. This element becomes the new *min* and, because we set x to its value, it is the element that will be deleted from its

cluster.

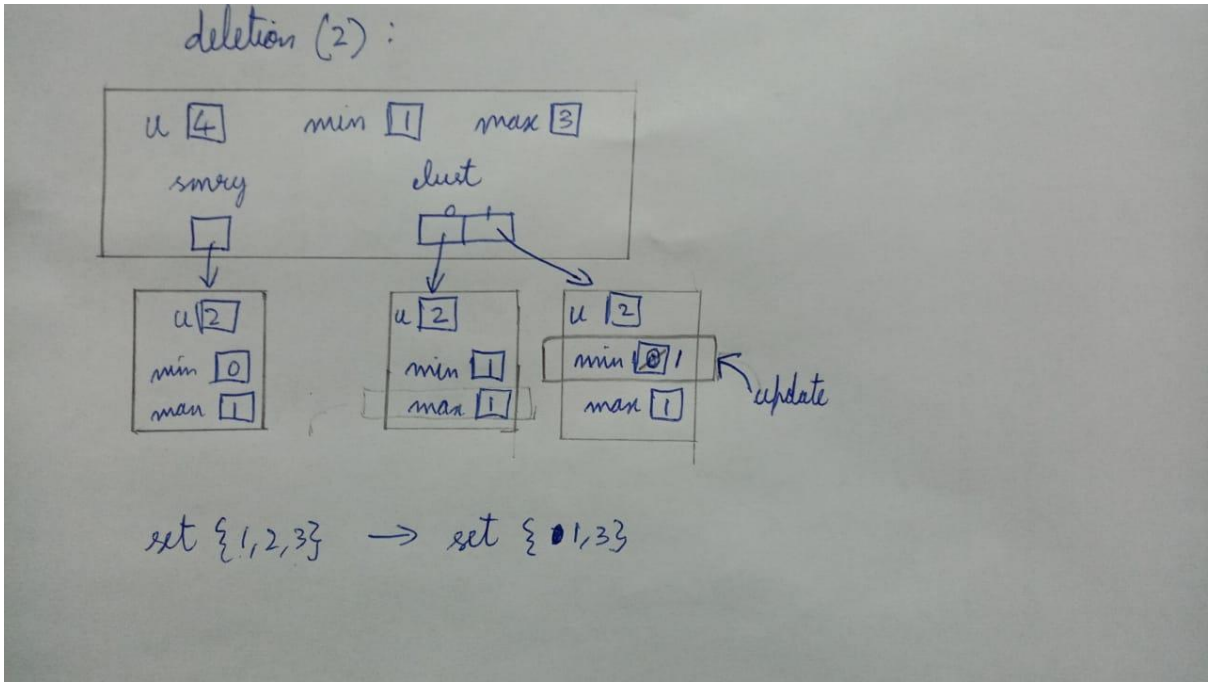
We then delete x from its cluster. That cluster might now become empty, which we check, and if it is, then we need to remove x's cluster number from the summary. After updating the summary, we might need to update *max*. We check to see whether we are deleting the maximum element in V and, if we are, we set *summary-max* to the number of the highest-numbered nonempty cluster. If all of V 's clusters are empty, then the only remaining element in V is *min*; line for this case, and updates *max* appropriately. Otherwise, sets *max* to the maximum element in the highest-numbered cluster.

Finally, we have to handle the case in which x's cluster did not become empty due to x being deleted. Although we do not have to update the summary in this case, we may have to update *max*, if we have to update *max*, we done so, and then we end.

- Algorithm (pseudo code same as program)

```
if V.min == V.max
    V.min = NIL
    V.max = NIL
elseif V.u == 2
    if x == 0
        V.min = 1
    else V.min = 0
    V.max = V.min
else if x == V.min
        first-cluster = VEB-TREE-MINIMUM(V.summary)
        x = index(first-cluster,
            VEB-TREE-MINIMUM(V.cluster[first-cluster]))
        V.min = x
    VEB-TREE-DELETE(V.cluster[high(x)], low(x))
    if VEB-TREE-MINIMUM(V.cluster[high(x)]) == NIL
        VEB-TREE-DELETE(V.summary, high(x))
        if x == V.max
            summary-max = VEB-TREE-MAXIMUM(V.summary)
            if summary-max == NIL
                V.max = V.min
            else V.max = index(summary-max,
                    VEB-TREE-MAXIMUM(V.cluster[summary-max]))
    elseif x == V.max
        V.max = index(high(x),
            VEB-TREE-MAXIMUM(V.cluster[high(x)]))
```

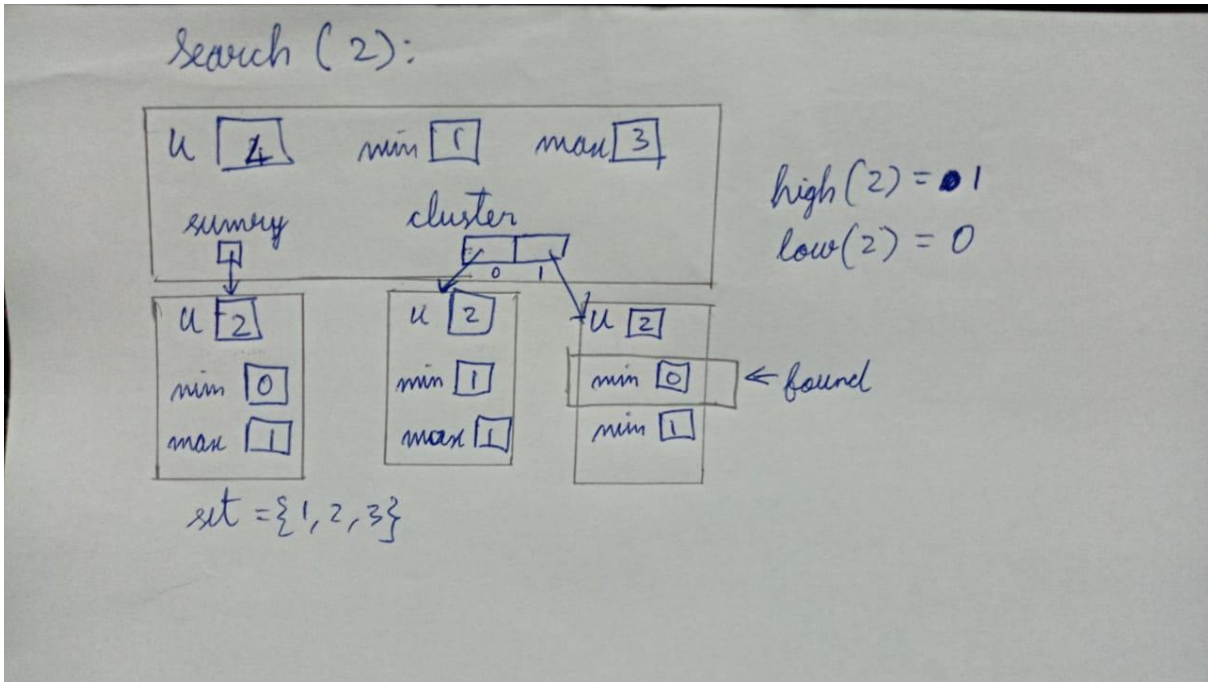
- Example:-



Operation 3: Search: -

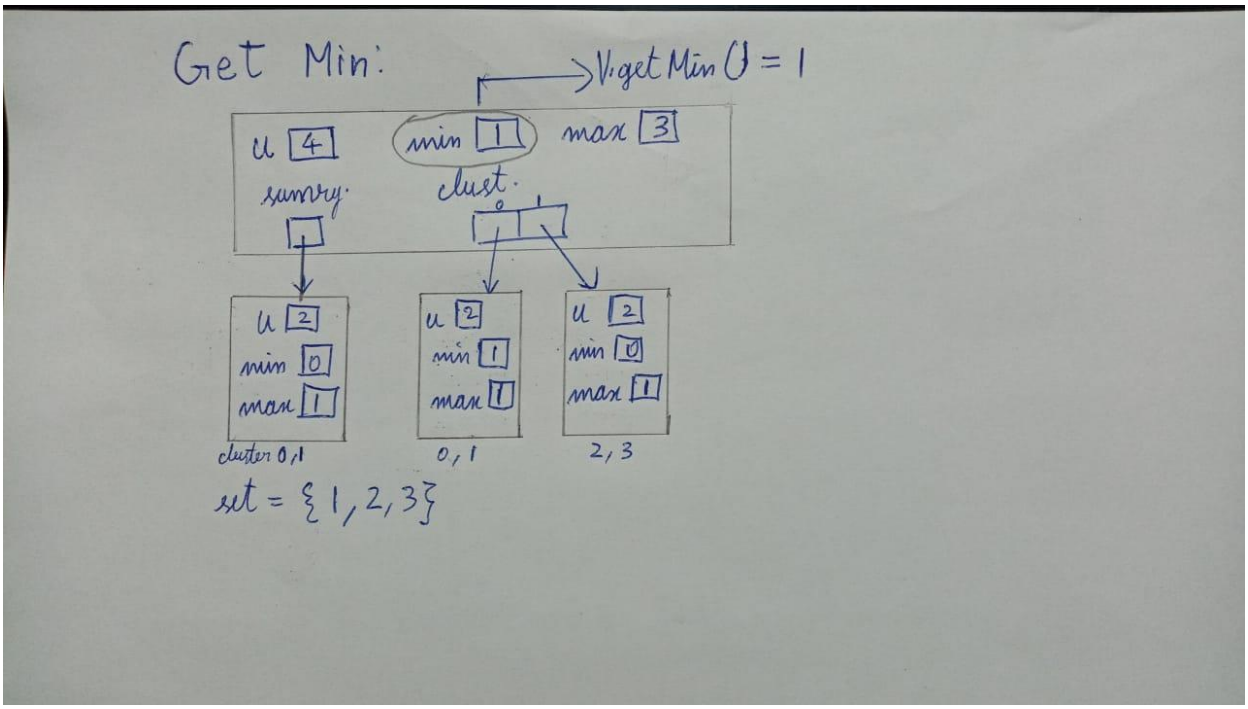
- Explanation

  The search operation is done by checking if the element is already in the min or max members, if not, then recursing deeper into the tree and checking min and max members, until u =2.
- Algorithm
  1) if the key value is less than min, return false
  2) if key matches either min or max data members, then we found the element, return true
  3) else, we recurse further, by calling search on the cluster that the key is, and passing low(key) as the new key.
- Example (Attached below) :-



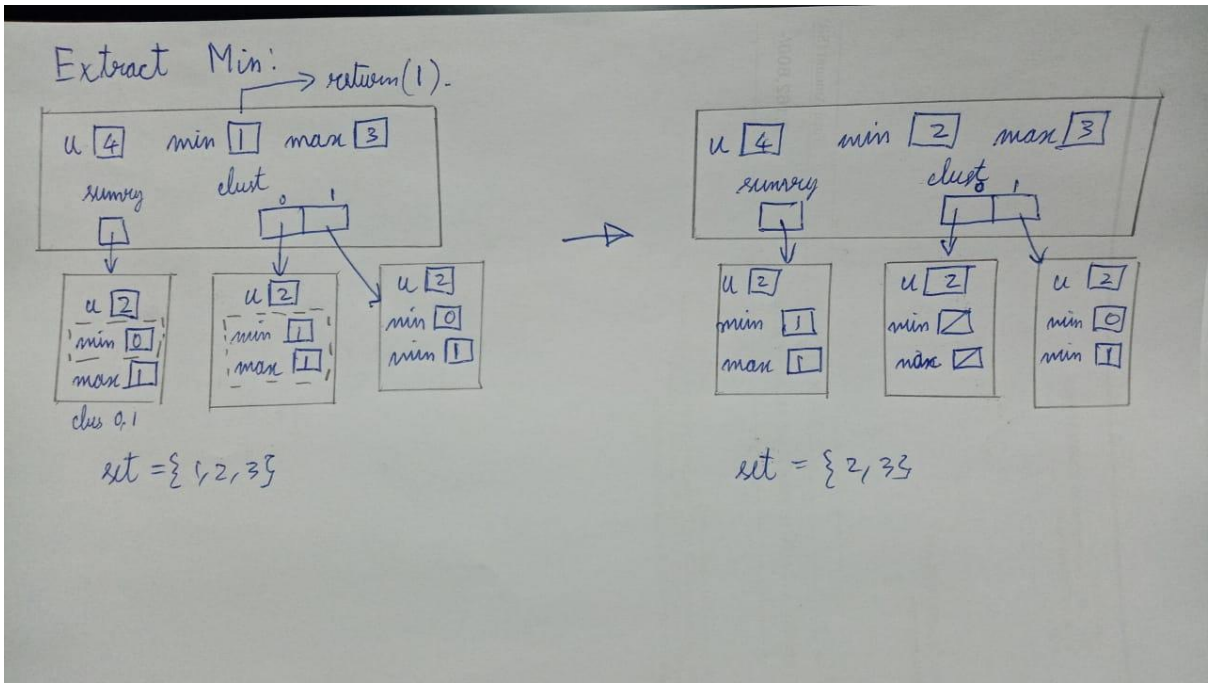
Operation 4: FindMin: -

- Explanation:
  The find minimum operation is very easy to perform in the case of VEB tree, as it stores the minimum element in the root node itself, which needs to be accessed.
- Algorithm
  1) Return value stored min member in root node.
- Example



Operation 5: ExtractMin: -

- Explanation
  The Extract min operation is just performing find min and then performing delete on the min element.
- Algorithm
  1) Store the result of Find min in x.
  2) delete x from the tree
  3) return x
- Example : (Attached Below) :-



---

5. Summary Table: -

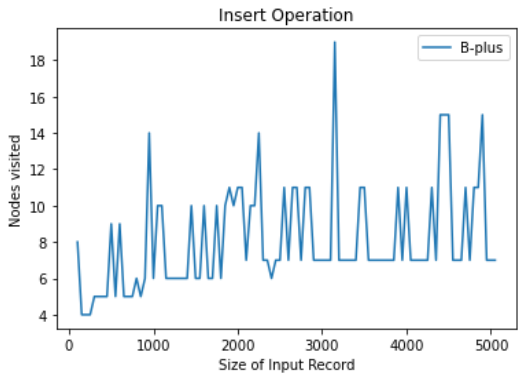
DATA STRUCTURE	INSERTION	DELETE	SEARCH	FINDMIN	EXTRACTMIN

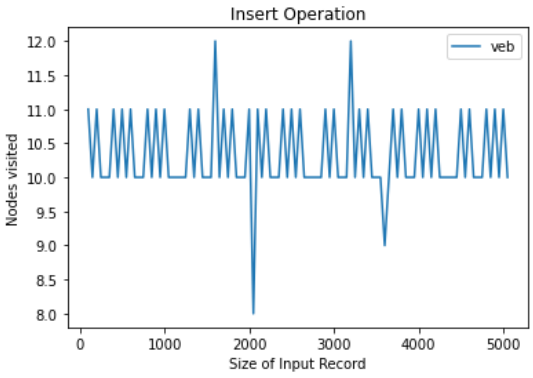
B+ TREE	O(logn)	O(logn)	O(logn)	O(logn)	O(logn)
B* TREE	O(logn)	O(logn)	O(logn)	O(logn)	O(logn)
VAN EMDE BOAS TREE				O(1)	

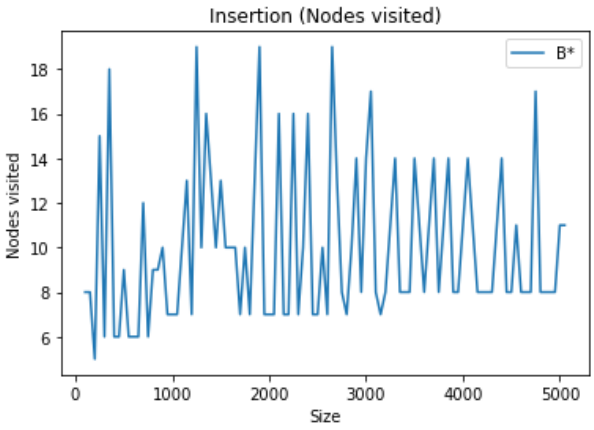
6. PERFORMANCE COMPARISON: -

- ### 6.1. Insertion: -

Individual Performances: -

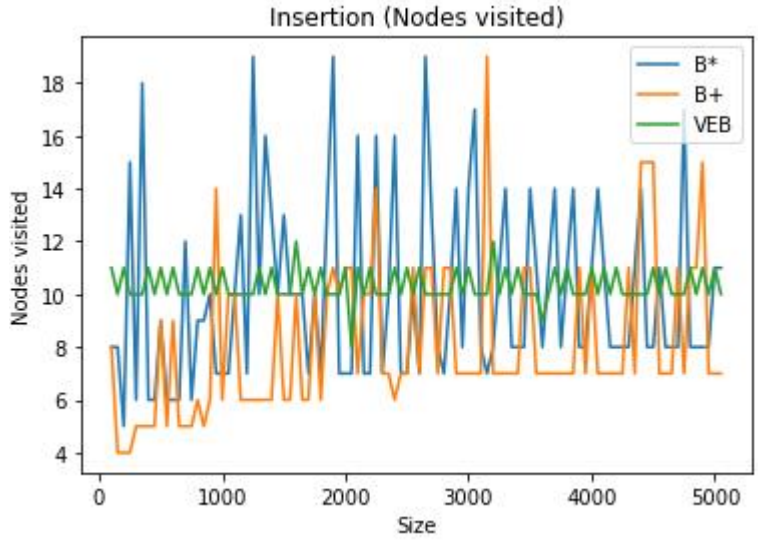




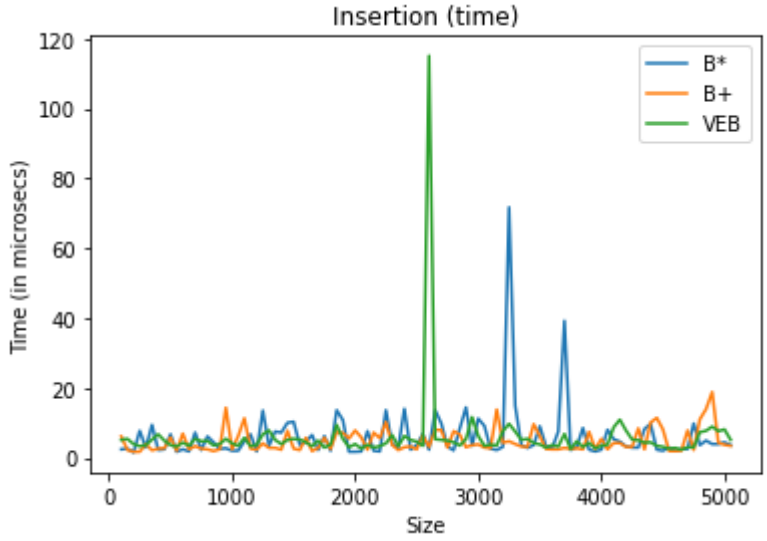


Graph of Comparison: -

- Comparison with count of nodes visited: -



- Comparison with time: -



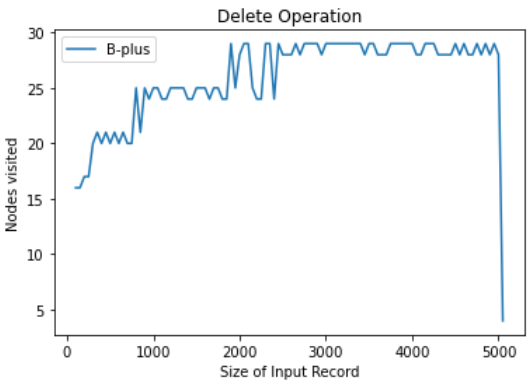
Inference: -

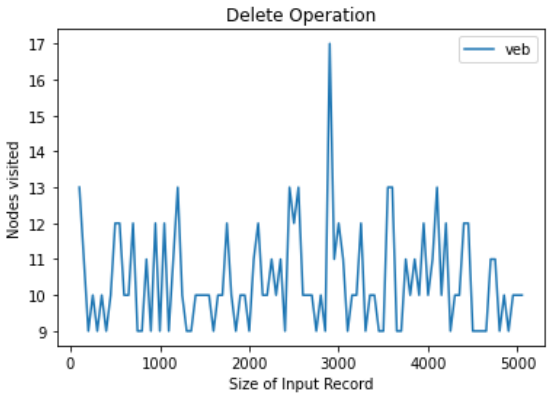
- B+ outperforms B* on insertion as the order is small (4 here) and 2-3 split operation is costlier than splitting a node into 2.
- While both traverse to the leaf node to insert a particular key, in the case of an overflow, B+ splits and moves/copies to the parent (essentially visiting one extra node at each iteration), however, B* deals with an overflow by combining a node with one of its siblings and then splitting, inherently visiting more nodes to deal with the overflow.
- Moreover, the number of comparisons performed in a B* tree might be large owing to the densely packed nodes.
- As the order increases the difference in heights might become significant and hence B* tree might outperform B+ tree.

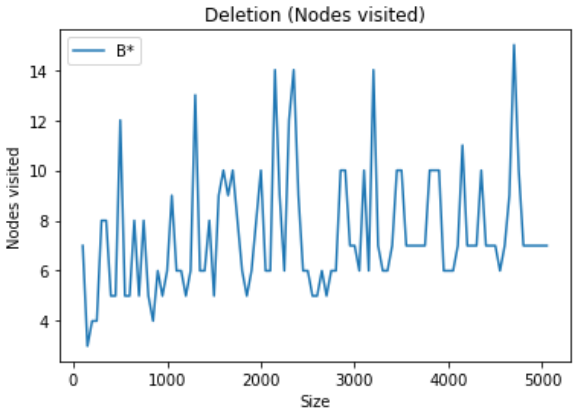
- Since the value of u for the VEB tree is 2^17 for these graphs, the height of the tree would be 17-1 = 16. So, while inserting, due to lazy propagation (due to the use of min and max members), the number of nodes accessed would be some amount less than the height on average. In this case, we get it around 10 and 11 node accesses.

- ## 6.2. Deletion: -

Individual Performances: -

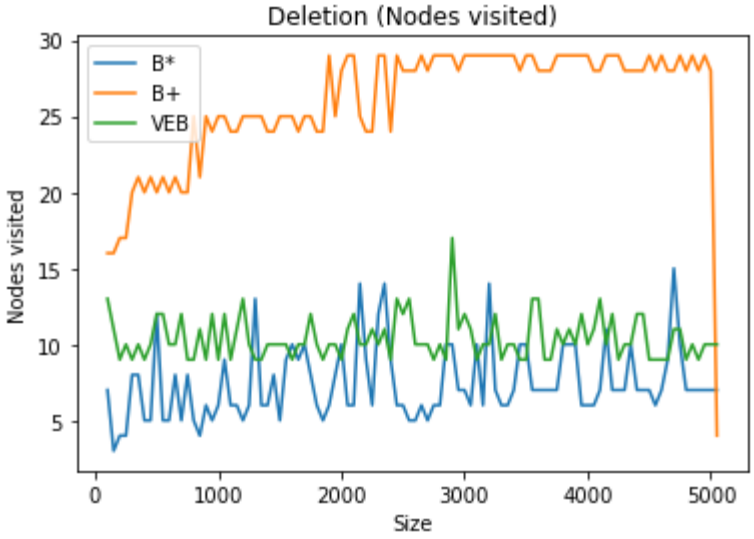




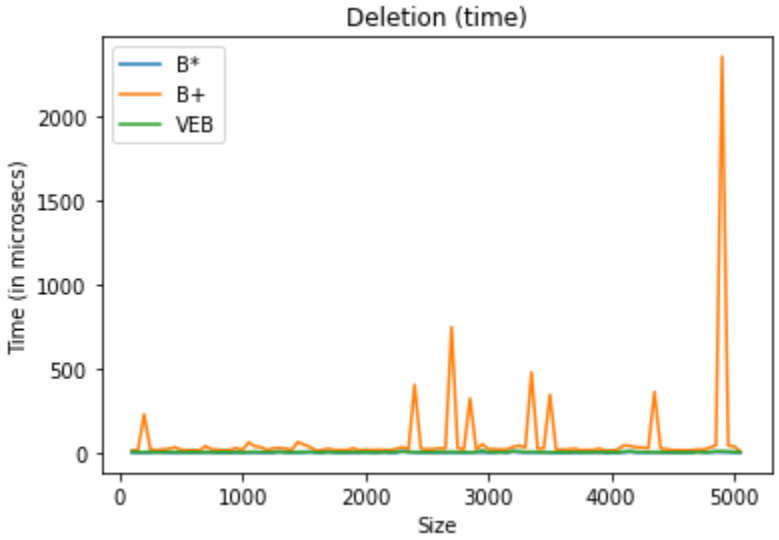


Graph of Comparison: -

- Comparison with count of nodes visited: -



- Comparison with time: -



Inference: -

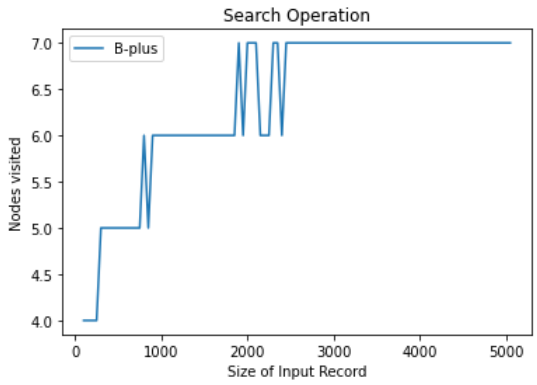
- B+ could have duplicate elements in the non-leaf nodes that need to be deleted unlike B* which deals with unique elements.
- True, underflow can happen in B* too, causing a need for merging/redistribution but on an overall level, there is an increased chance for B+ to take more time as it would have

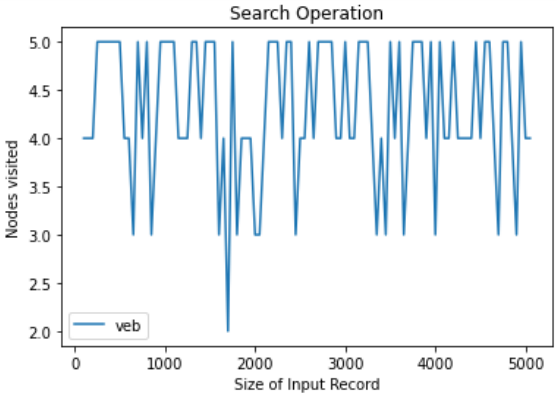
to iterate back through the parents to the root not just for underflow cases but also for duplicates.

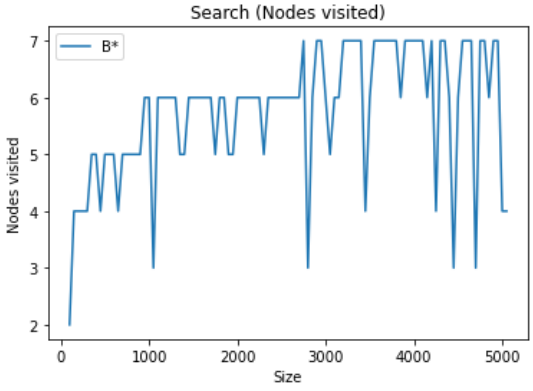
- Moreover, even though both the trees might have to perform merging repeatedly up to the root, B* has much lesser height than B+ and ends up visiting a lesser number of nodes
- As mentioned earlier for insertion, the height of the VEB tree is 16 for the value of u chosen, and during deletion, all we need to do is update the min and max members used in the lazy propagation. So only rarely, do we have to traverse all the way to the leaves to set the min and max to NIL.
- Spike cases: -
  Deletion in B+ has a few spikes as can be seen from the graph. This can be explained with the understanding that the key deleted might have its duplicate equivalent close to the root or its immediate children.
- Thus, the difference in depth between the leaf and the node that contains the duplicate is extremely large rendering a lot more shifts. Now the spikes occur on a much lower level because it should be made known that the probability of a deleted to node to have its duplicated close to the tree is low.

• 6.3. Search: -

Individual Performances: -

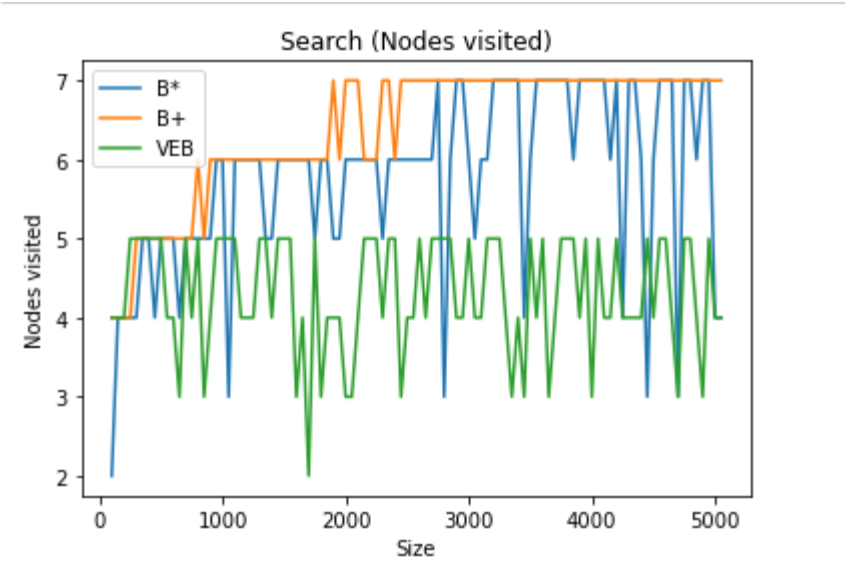




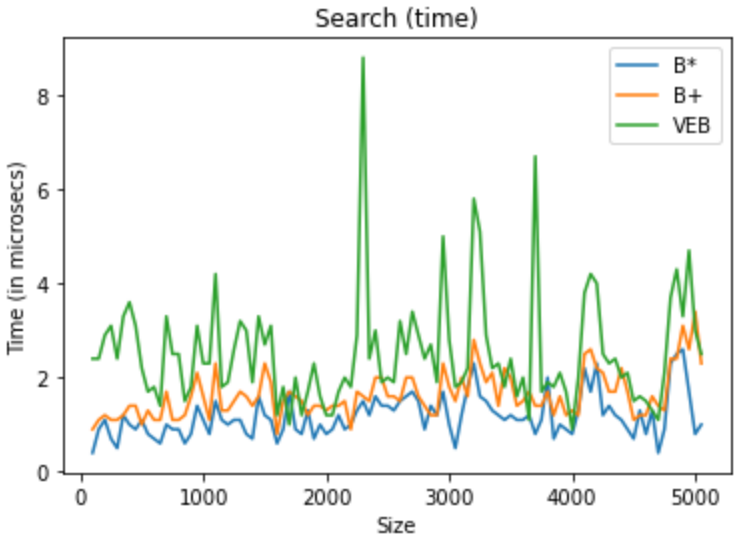


Graph of Comparison: -

- Comparison with count of nodes visited: -



- Comparison with time: -



Inference: -

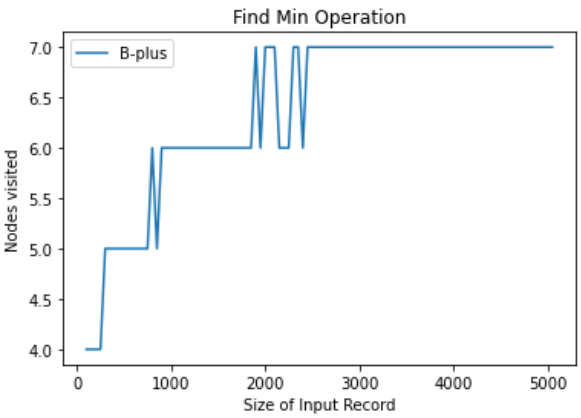
- Search for B+ and B* are dependent on the height of the tree. So, for smaller input sizes, it is rather hard to see a noticeable difference in search times as the heights of the trees are small.
- However, as input size increases parallelly, B* will outperform B+ to a fair bit of an extent as again, the height of B* will be less than that of its equivalent B+ tree (same elements & same size) as its more compact.

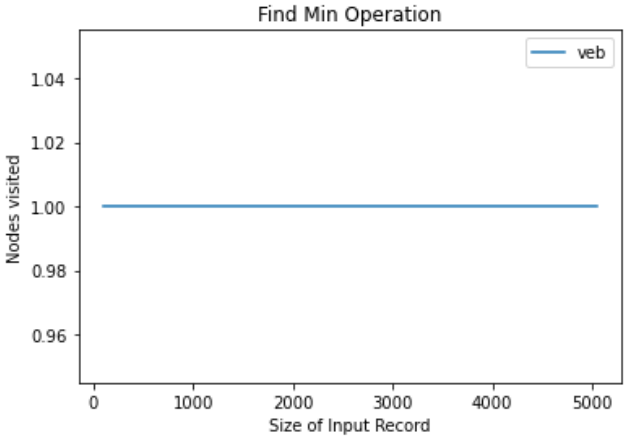
- It is observed that the number of nodes accessed during insertion is about 10 or 11, so the number of nodes to be accessed for the search operation is either equal or lesser, as it only has to go up to where the min or max is equal to the key.
- Otherwise, if a cluster is empty where the key is to be found, then it returns false immediately, which enables it to do the search operation much quicker, as we know which clusters are using the summary date member.
- And for the purpose of graphing, we picked a random key to search for, which likely doesn't exist in the tree, so it returns false with relatively low number of node accesses

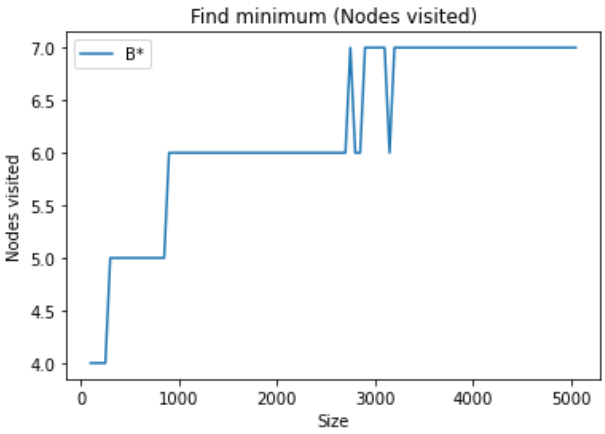
---

- ## 6.4. FindMin: -

Individual Performances: -

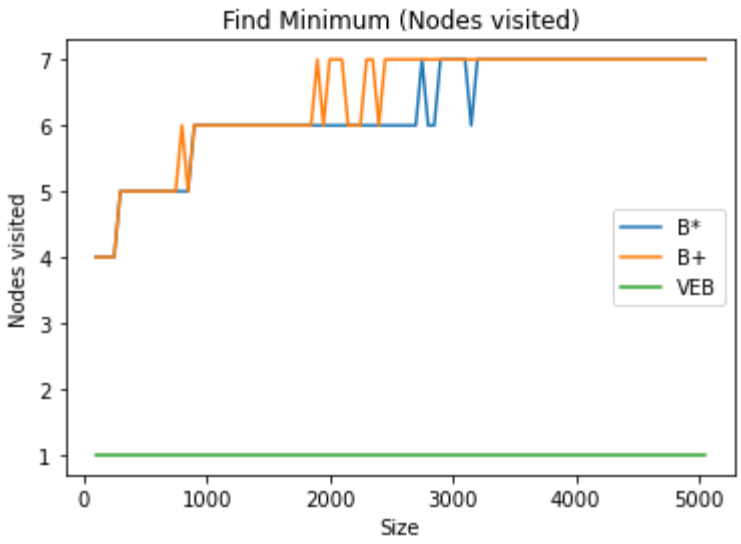




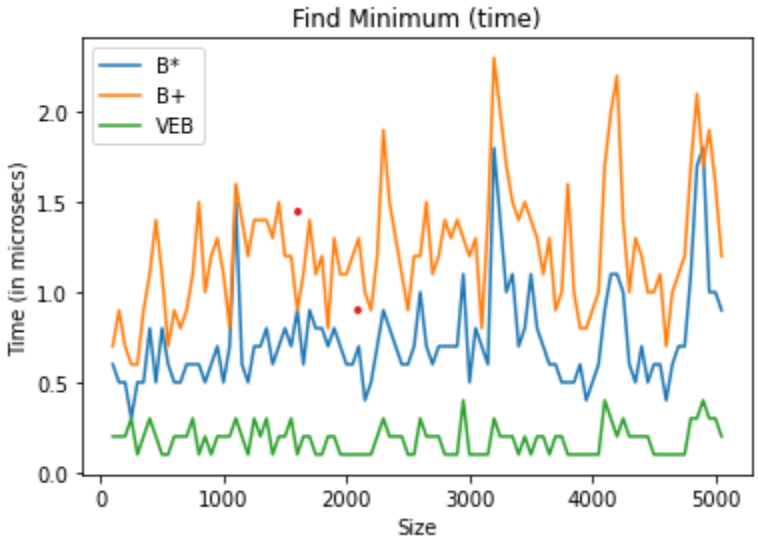


Graph of Comparison: -

- Comparison with count of nodes visited: -



- Comparison with time: -

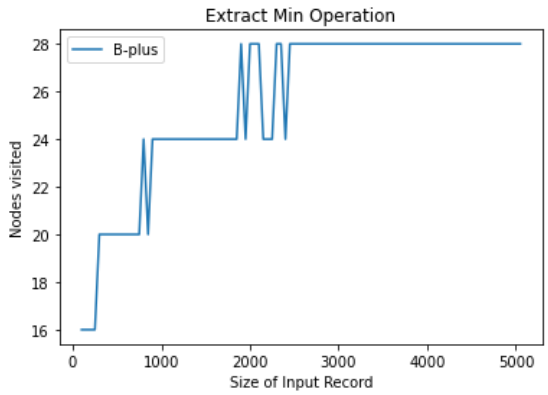


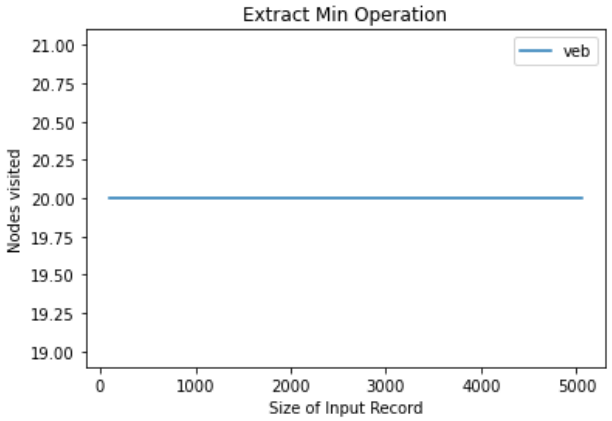
Inference: -

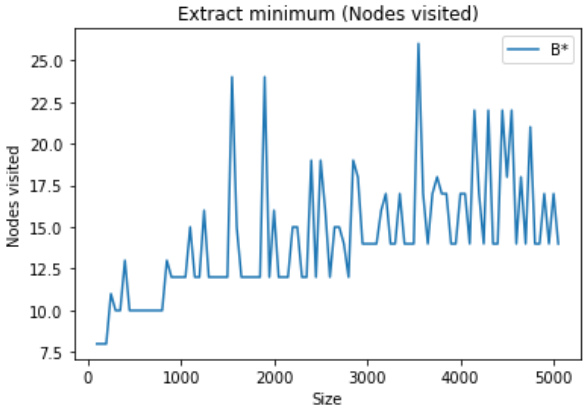
- We visit the leftmost leaf node to find the minimum key. As B* has lesser height than B+ (nodes are more densely packed) it visits lesser number of nodes
- The minimum number in the tree is stored in the root node, so it only needs one access in the case of VEB tree.

6.5. ExtractMin: -

Individual Performances: -

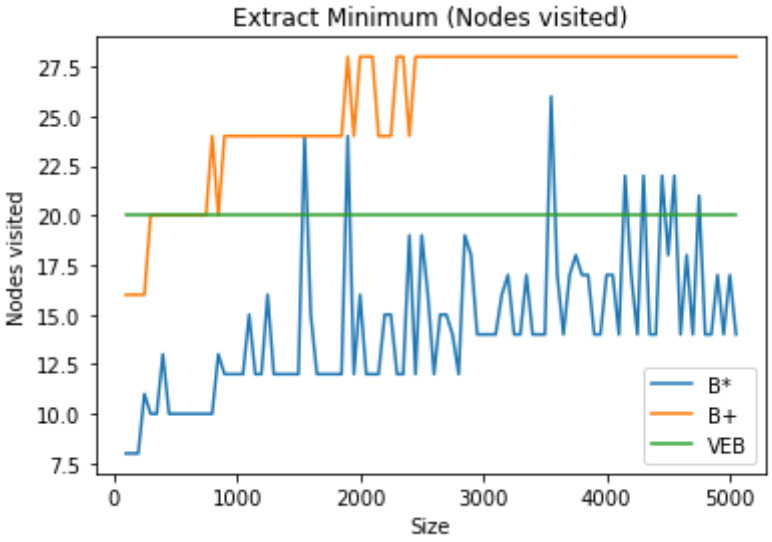




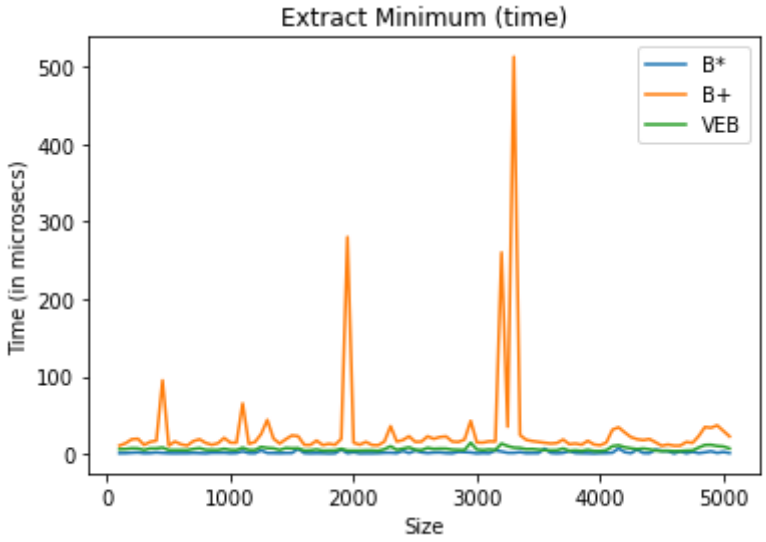


Graph of Comparison: -

- Comparison with count of nodes visited: -



- Comparison with time: -



Inference: -

- As observed before, B* outperforms B+ in both finding min and deletion. Hence B* visits lesser number of nodes than B+
- The extract min operation in VEB is simply the find minimum operation, which needs about 2 accesses, and the deletion operation, which in this case, the minimum element always exists in the tree, which results in traversal to the leaf always and a few summary nodes accesses.
- So, the number of nodes accessed is the sum of node accesses to the deletion and find min operation, which is 20 in this case.

CONCLUSION: -

- B* outperforms B+ tree most of the times but the implementation is complex. So, B+ is used commonly.
- The performance of VEB tree depends on the universal set. For smaller universal set sizes, the overheads incurred are high.
- But as the size of the universal set increases these overheads become much less significant and the log(log(u)) complexity of VEB turns out to be much better than the O (log n) complexity of B+ and B* trees.
- However, the two trees were designed for different purposes and the choice of the tree to be used depends on the application, the size of the dataset and the complexity of the project.

THANK YOU, MA'AM!