Apr 6, 2023
Anurag Verma

BTree Design Document

Contents

1 Introduction

A BTree is a self-balancing tree data structure that stores and maintains the data in a sorted manner and allows insertion, searching, and deletions in logarithmic time. The nature of BTree is that the insertions are at the leaf level but the growth of BTree happens from leaf to root i.e. bottom to top. This is different from the existing self balancing data structures such as AVL Tree and Red Black tree where the tree grows from top to bottom. Due to its growth from Bottom to Top, its root node is not always fixed. This is to maintain its core property of a Self Balancing Tree.

Here are some basic rules of a BTree:

1. A Btree can be an order of some integer.
   Minimum order is 3 and maximum order is 512.
2. A BTree node can have a maximum of m children.
3. A node can contain a maximum of m - 1 keys.
4. A node should have a minimum of [m/2] children.
5. A node (except root node) should contain a minimum of [m/2] - 1 keys.

The target audience of this document includes Developer, Architect, TechnoManager and enthusiastic members.

The current implementation of BTree is done in NIM language so all the code snippets and other data structures mentioned below are in NIM language format.

2 Data Structures

Normally in a BTreeNode, key datatype is defined statically through a fixed size array.

The current implementation is having some variation. Keys are stored in a linked list named as ListNode and ListNode is encapsulated inside a BTreeNode. The left and right child of the current BTreeNode are pointers from ListNode.

A BTree of of Order 3 (m = 3) will have following attributes:

1. A node can have a maximum of 3 children. (i.e. m)
2. A node can contain a maximum of 2 i.e m - 1 keys.
3. A node should have a minimum of 2 children i.e. [m/2]
4. A node (except root node) should contain a minimum of 1 key ie. [m/2] - 1 keys.

The above information is represented by these following global variables:

BT_ORDER	Order of a BTree. Default order is 3 and Max order is 512
BT_MAXKEYS	One less than the order i.e. BT_ORDER -1 It is also called the max no. of children a BTreeNode can have
BT_MINKEYS	Minimum no. of keys a BTreenode can handle. It should be ceil[m/2] -1 where m is the order
BT_ROOT	The BTreeNode is a type of Root. Searching begins from here.
BT_LEAF	A BTreeNode which does not have any children. Addition of new keys takes place at this level
BT_NODE	An internal BTreeNode in the BTree which is neither Root nor Leaf.

Multiple NIM object types are defined to denote the BTreeNode, its keys, search results etc.

ListNode	It is a normal linked list which is the representation of user input keys. All keys are stored in a sorted sequential manner in the form of a linked list
BTreeNode	It is a logical grouping of all ListNodes in a single BTreeNode. In a nutshell each BTreeNode contains a linked list of ListNode objects
SearchResult	It is used when a successful search operation is performed. The search function returns the BTreeNode and ListNode as part of SearchResult

Here is the pictorial representation of BTreeNode having ListNodes as keys

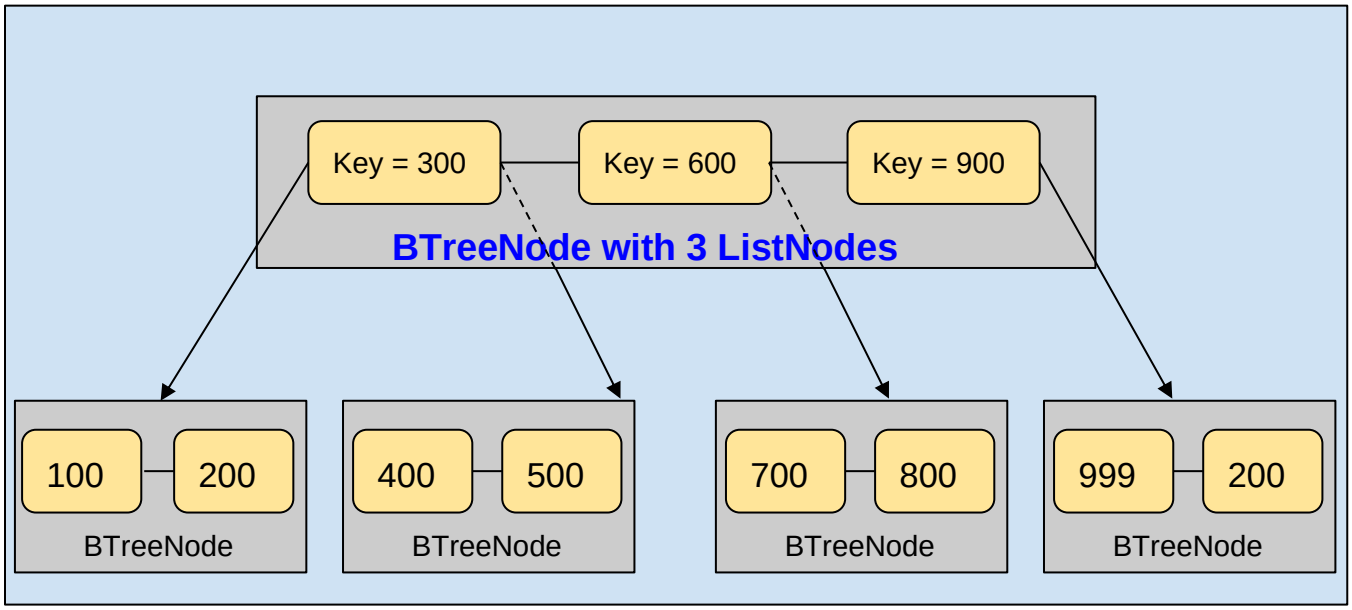


Figure 1

The ListNode and BTreeNode data structures are defined as below:

2.1 ListNode data structure.

It is a doubly linked list with prev and next items and pointers to left and right sub BtreeNodes as well. It also contains a pointer to the current BTreeNode as well. In figure-1, Key 300 is having left BtreeNode which starts with Key 100 and right BTreeNode which starts with 500, and next key aka ListNode in the current BTreeNode is 600

```
type #{
  ListNode* = object
    key                 *: int
    value               *: pointer
    next                *: ref ListNode
    prev                *: ref ListNode
    leftBTreeNode       *: pointer
    rightBTreeNode      *: pointer
    currentBTreeNode    *: pointer
    parentListNode      *: ref ListNode
```

#}

2.2 BTreeNode data structure.

This structure contains the start and end of ListNodes for faster searching and insertions.

```
type #{
  BTreeNode* = object
    bt_listhead      *: ref ListNode #Start node of keylist (ListNode)
    bt_listtail      *: ref ListNode #tail node of keylist (ListNode)
    bt_numkeys    *: int #No. of keys in current BTreeNode
    bt_id             *: string #Not mandatory, only for identification purpose
    bt_numchild    *: int #Not mandatory. Kept for debugging purpose
    bt_level         *: int #Not mandatory. Kept for debugging purpose

    # type of tree node
    # 1 = root
    # 2 = Non-Leaf and Non-Root
    # 3 = leaf
    bt_type         *: int
#}
```

2.3 SearchResult data structure

For searching purposes, a SearchResult type object is also defined. It is used to keep the return value of BTreeNode and the ListNode

```
type #{
  SearchResult* = object
    btreenode *: ref BTreeNode
    listnode     *: ref ListNode
#}
```

2.4 KeyValue data structure

This is an input data where key and value are encapsulated together in an object.

```
type
  KeyValue  * = object
       key  *: int
       value*: pointer
```

3 BTree Creation

Two APIs are provided for creating either an empty BTree or with an input list of keys.

proc initBTree* (order: int = BT_ORDER, logfile: string = BT_DEFAULT_LOG_FILE,): ref BTreeNode =
This function takes two parameters. First is the order of the tree and 2nd is the log file name. It returns a reference to the root of the empty BTree. Please note that there are no keys associated with this BTreeNode. Insertion will take place by invoking insertBTree function.

proc createBTree* (keys: seq[int]):ref BTreeNode =
This function takes the input keys as a sequence. It is basically a dynamic array of input keys. It returns a reference to the root of the BTree

4 BTree Insertion

At this point, the BTreeNode is created with zero or more keys (ListNodes). In order to insert keys at run time multiple variants of insertion function are provided

proc insertBTree*(key: int): int =
This is the basic one where an input key is pushed into the BTree and its return code is provided to the caller whether the insertion was successful or failed.

proc insertBTree*(key: int, value: pointer): int =
This function takes a key and its corresponding value to be stored in the ListNode. Note that the memory needs to be allocated first for the value provided.

proc insertBTree*(keyval: ref KeyValue): int =
Here a complete Key and its Value object are inserted into BTree

Here is the insertion logic in a normal layman term.

1. With the given input key k, first find the target leaf BTreeNode where the key is going to be inserted. A function InsertKey() function is written to insert the key in the current BTreeNode in sorted manner.
2. If the leaf is NOT full i.e. its no. of keys are less than BT_MAXKEYS then the key k is inserted in the linked list linearly.
3. If the leaf is FULL then split this target leaf into three parts.
    a. Left part is the original BTreeNode.
    b. Middle Node which is going to become a new Parent BTreeNode
    c. Right part (becomes another new BTreeNode) is basically the list of nodes which are next to Middle Node
    d. This process will continue repeatedly until the criteria of inserting the key in the node gets satisfied i.e. no. of keys are less than BT_MAXKEYS
    e. During this process, there could be a possibility that the insertion chain can go up to Root and define a new root node.

    InsertKeyNode() function is written to promote the middle Node as the new parent of the splitted overflowed current BTreeNode.

Two important points to be highlighted in the insertion logic:

1. To find the Target Leaf BTreeNode
   Here, the searching of BTreeNode starts from the root i.e. from the top, and goes down until the target leaf BTreeNode gets encountered. There is no way a target leaf BTreeNode can/will be NULL. The searching criteria are the same as we do in Binary Search Tree. A function findLeaf() is written to accomplish this logic.

2. To Insert the new Key
   Once a target leaf BtreeNode is found, the insertion process starts and typically goes from bottom to top. This is opposite to searching and insertion in Binary Search Tree. This whole process sometimes involves splitting the current BTreeNode and creating new BTreeNodes up in the ladder, so that it can maintain the logic of the maxkey and minkeys requirement of each intermediate BTreeNode. Due to this complex insertion logic, BTree is known as a Self Balancing Tree.

Normally two types of insertion happen in BTree:

1. Type-1: When the Root of BTree is not yet formulated
2. Type-2: When Root of BTree is already present

Let’s dwell deeper into both types of insertions.

Type-1: When the Root of BTree is not yet formulated

The logic of insertion is as follows

1. Insert a new key when the leaf node is empty. This is a normal linked list insertion.
2. Insert new key when leaf node is partially empty. Procedure is the same as above.
3. Insert a new key when the leaf node is already full. This requires a different approach
    3.1. Split the leaf BTreeNode and find the median or middle node.
    3.2. Create a new BTreeNode which as middle node as the only key. This node is the root of the BTree.
    3.3. The left side keys list becomes the new left BTreeNode. Its parent is set to root
    3.4. A new BTreeNode is created for right side keys also. Its parent is also set as root
4. After successful insertion, the root of BTree is formulated

Type-2: When Root of BTree is already present

Multiple cases exist in this type:
CASE-1: Insertion at Leaf Node.
CASE-2: Promotion of Leaf node due to overflow condition in CASE-1
CASE-3: Promotion of Internal Node due to overflow condition in CASE-2
CASE-4: Root Promotion

In all the above cases, first, the Target BTree LEAF node is identified. Identification of the target node requires a Binary Search traversal which starts from the ROOT Node and moves downwards i.e from top to bottom and ends up in the BTreeNode where the new key is placed for insertion.

4.1 CASE-1: Insertion at Leaf Node

Insertion in the target leaf node will always be in sorted order. This is the same as Type-1 insertion, until the target node is overflowed i.e., target node is having more than Tree Order - 1 keys. Once the new key is inserted, its prev and next pointers point to ListNode present in current BTreeNode and parent child relationship pointers are adjusted accordingly.

4.2 CASE-2: Promotion of Leaf Node

Any BTreeNode can be overflowed when the no. of keys crosses the BT_MAXKEYS limit. In the case of an overflowed target leaf node, a middle node is identified first by following the slow and fast pointer approach. This middle node is promoted to its parent node so that the maximum allowed keys of the target node are maintained. Overflow is a recursive process, i.e., once a middle node is promoted into its parent then the parent node can also be overflowed. The same process of finding the middle key and getting promoted is followed in the parent node also and the process goes on until the max allowed keys are maintained or a new root is formulated again.

Insertion is a complex process and it always happens from bottom to top. Due to this complexity, the root of the BTree keeps on changing. This is different from a normal Binary Tree or BST.

Let's dwell deeper into this case.

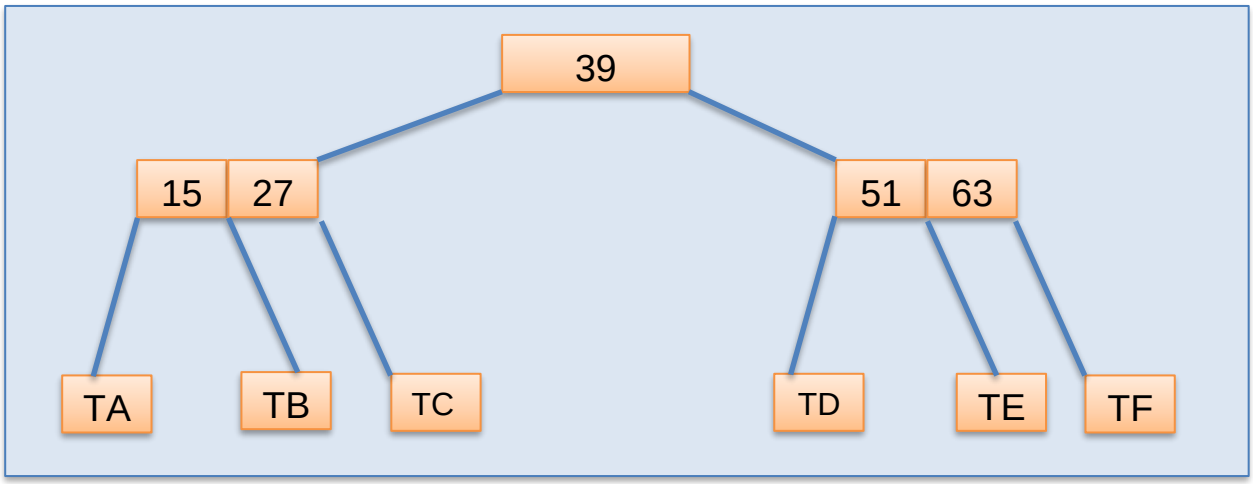


Figure 2

Figure 2 shows the initial state of BTree with root value as 39 and TA, TB…TF are the leaf nodes.

When a new key 3 (K3) is ready to be inserted, the findLeaf() function will return TA as the target leaf BtreeNode.
TA is a child BTreeNode of parent Key 15 ListNode. TA may have a list of keys (ListNodes) in the form of linked list but all the keys will be lesser than its parent ListNode 15.

If number of keys present in Leaf BTreeNode TA is less than the maximum no. of keys allowed then insertion of Key 3 will be like a normal linked list insertion in a sorted manner (CASE-1).

The situation gets complex when TA already has the maximum no. of keys. If new key K3 gets inserted into TA, an overflow condition occurs which results in splitting of BtreeNode TA. An overflow condition occurs when the number of keys present in a BTreeNode is greater than the maximum no. of keys (BT_MAXKEYS), a BTreeNode can accommodate.

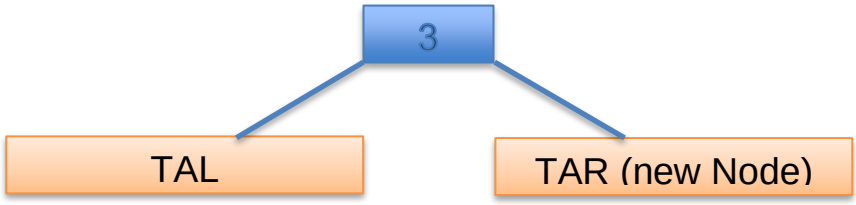
Therefore, in every overflow condition and in order to maintain the maximum no. of keys, the current BTreeNode is split into three parts: Left Key List, middle node, and Right key list.

During the insertion process, BTreeNode TA goes through multiple stages:

a) stage1: K3 is inserted into TA
b) stage2: TA is split into 3 parts.
c) stage3: Promotion of middle node created in stage2
d) stage4: Adjustment of Left and right part of TA as result of its split in stage2

Here is the stage1 representation. TA node is divided into 3 parts:
TAL i.e left side of TA
K3: A new intermediate ListNode
TAR i.e right side of TA

Here is the stage2 representation i.e. after splitting TA into 3 parts.



In stage2, a new BTreeNode, represented as TAR in the image above, is created with having all the keys (ListNodes) which are greater than 3.

The size of the existing TA node is reduced by 1 + no. of keys in TAR. The size of TAL and TAR can not go below a threshold of BT_MINKEYS.

Stage2 and Stage3 will lead to insertion of K3 into its parent node and TAL and TAR will become its left and right subtree.

Figure 3 below shows the final state of Parent and Child BTreeNodes after inserting the key K3.

Nodes which are yellow in color are newly created i.e. it will have a fresh memory allocation.
Nodes which are green in color are the ones which are modified due to this insertion logic.
BTreeNode 39 will now have its left subtree starting from 3, as opposed to 15 earlier.
Similarly, newly added ListNode 3 will have its Parent Node as 39.

Remaining BTreeNodes such as TB and TC and other BTreeNodes will maintain the Status Quo.

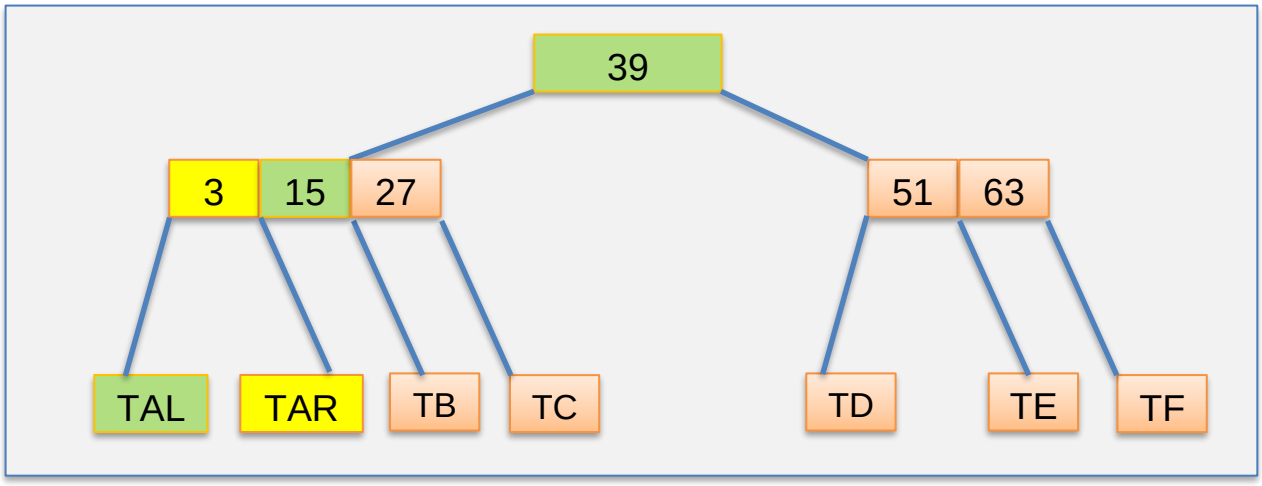


Figure 3

The example showed in Figure 2 and 3 describes the insertion process which results in the promotion of a leaf node which happens to be in the extreme left of its Parent BTreeNode.
Similarly, the promotion of a leaf node can happen either to be in the extreme right or at the intermediate of its Parent BTreeNode also. In both these cases, the next and prev pointers are adjusted for promoted node and its adjacent nodes. In addition to next and prev pointers, other pointer variables which maintains the parent child relationship between promoted node and its new parent are also adjusted.

Now we have come to a point where promotion of internal node happens. This condition arises when the promotion of leaf node in CASE-2 leads to another overflow situation in Parent BTreeNode which ultimately results into another split in Parent BTreeNode. CASE-3 deals with that situation.

4.3 CASE-3: Promotion of Internal Node

4.3.1 Promotion in Extreme Right Position

Consider the BTree in figure 4. This is the initial state of the tree before any sort of new insertion takes place. Here we have shown an Intermediate or Internal BTreeNode which is denoted by the key list [51 | 63]. This node is going to be split into futher pieces.

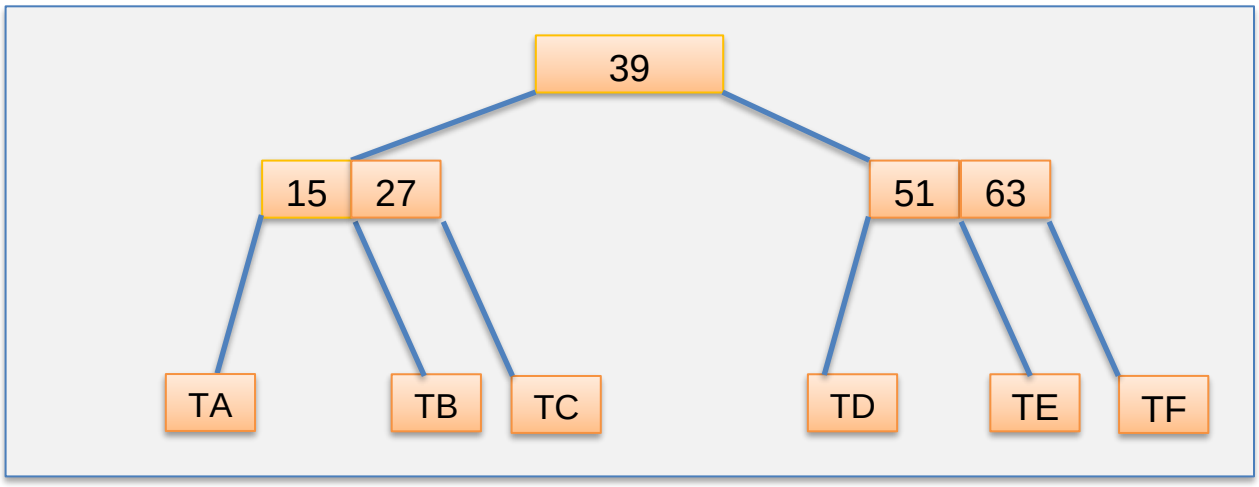


Figure 4

In the event of an insertion of a new key 45 i.e. K45, first findLeaf() function will be invoked and will return the target Leaf node as TD. Again, TD is a list of nodes (ListNode data type) which are connected through next and prev pointers.

For simplification purpose, lets assume that key 45 is inserted into the centre of TD list of nodes and it gets promoted to its parent node [51 | 63]
This process of promotion is same as we followed in CASE-2.

Here is the state of the tree after Key 45 is inserted and promoted. This is however an intermediate state and it is shown for understanding purpose.

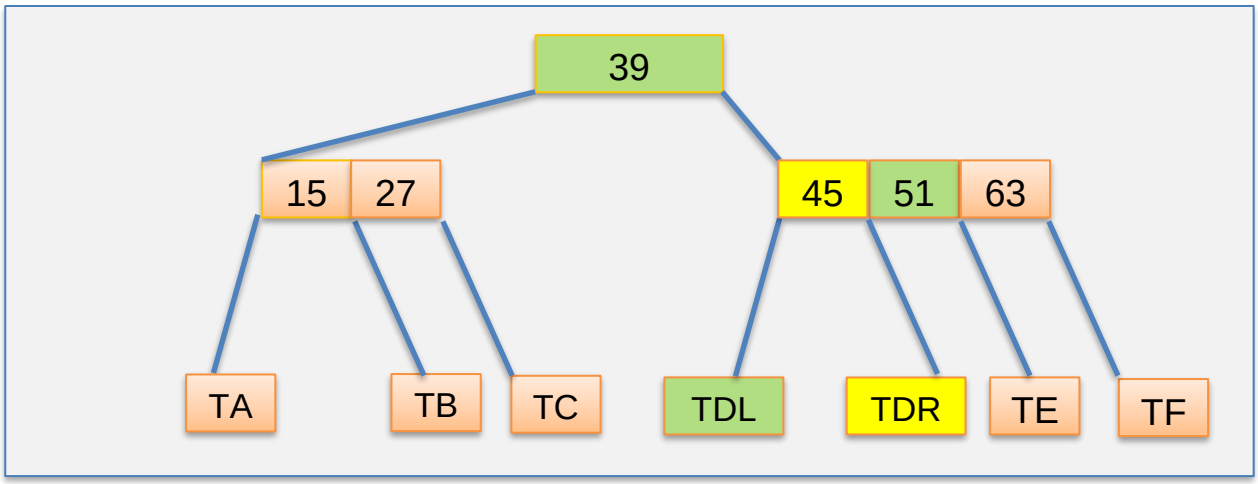


Figure 5

Nodes which are yellow in color are newly created i.e., it will have a fresh memory allocation.
Nodes which are green in color are the ones which are modified due to this insertion logic.
BTreeNode 39 will now have its right subtree starting from 45 as opposed to 51 earlier.
Similarly, newly added ListNode 45 will have its Parent Node as 39 and its left and right child BTreeNodes are TDL and TDR.

Remaining BTreeNodes such as TE and TF and other BTreeNodes will maintain the Status Quo.
Assume that the intermediate BTreeNode node [45 | 51 | 63] crosses it maxkeys limit and a split is required to promote the desired key 51 into its Parent Node.

This whole process goes through multiple stages:

a) stage1: Find the middle node and split the current internal BTreeNode
b) stage2: Promotion of middle node into its Parent Node.
c) stage3: Adjustment of Left and right pointers and parent-child relationship pointers.

Stage 1 and Stage 3 are having the same logic as we have seen in CASE-2 above i.e. during the promotion of a leaf node into its parent BTreeNode. Stage 2 is different here because now we have left and right BTreenode child pointers associated with each promoted node.
The important point to be noted here is that the right child (right subtree) of the promoted node will always become the left child of the newly created right keylist BTreeNode.

After successful insertion and promotion of key 45 and promotion of key 51, the BTree will finally look like this.

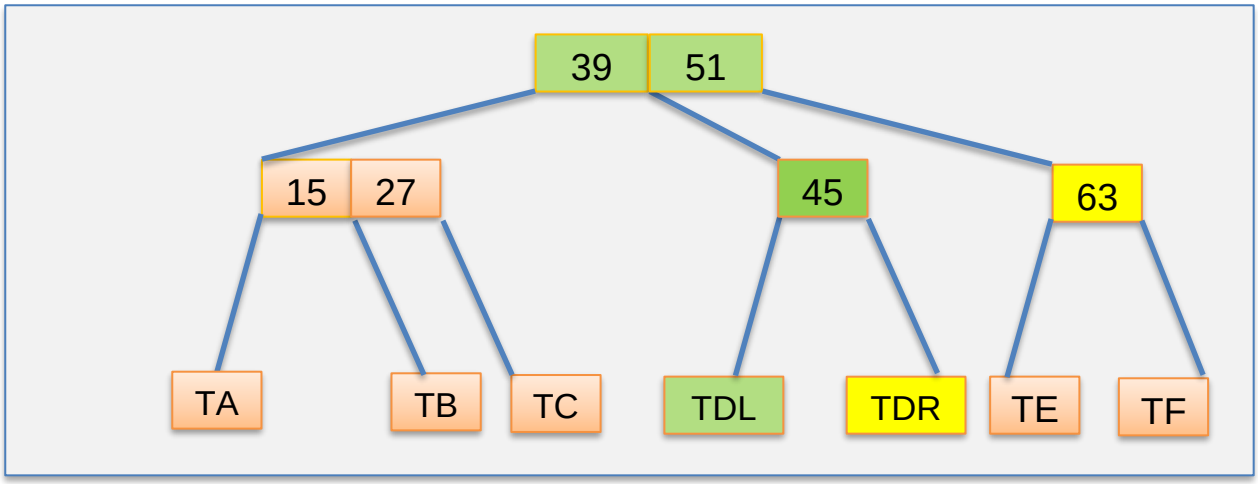


Figure 6

Here is the state change view of each affected node in the current BTree after promoting Key 51.
LST stands for Left SubTree
RST stands for Right SubTree

Before Promotion of 51	After Promotion of 51	State
39.LST = [15 | 27]	39.LST = [15 | 27]	SAME
39.RST = [51 | 63]	39.RST = 45	CHANGED
39.NEXT = NULL	39.NEXT = 51	CHANGED
45.NEXT = 51	45.NEXT = NULL	CHANGED
45.LST = TDL	45.LST = TDL	SAME
45.RST = TDR	45.RST = TDR	SAME
51.LST = TD	51.LST = NULL	CHANGED
51.RST = TE	51.RST = 63	CHANGED
51.PREV = 45	51.PREV = 39	CHANGED
51.NEXT = 63	51.NEXT = NULL	CHANGED
63.LST = NULL	63.LST = TE	CHANGED
63.RST = TF	63.RST = TF	SAME
63.PARENT = 39	63.PARENT = 51	CHANGED

4.3.2 Promotion in Extreme Left Position

Figure 6 considered a scenario where the internal node key promoted to its parent node and inserted into the extreme right location. E.g Key 51 was inserted right to key 39.

We now continue the same BTree as shown in Figure 7 and introduce another key 3 in the left subtree [15 | 27] to consider another scenario in which the middle key [15] will be promoted to its parent node [39 | 51] in its extreme left position

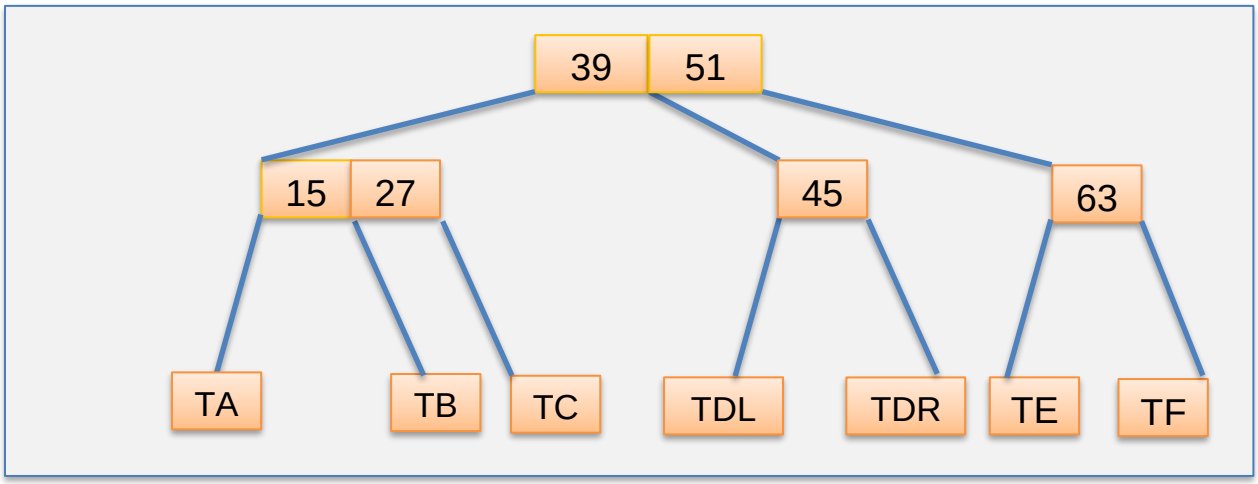


Figure 7

The intermediate state of BTree after inserting Key 3 in [15 ,27]

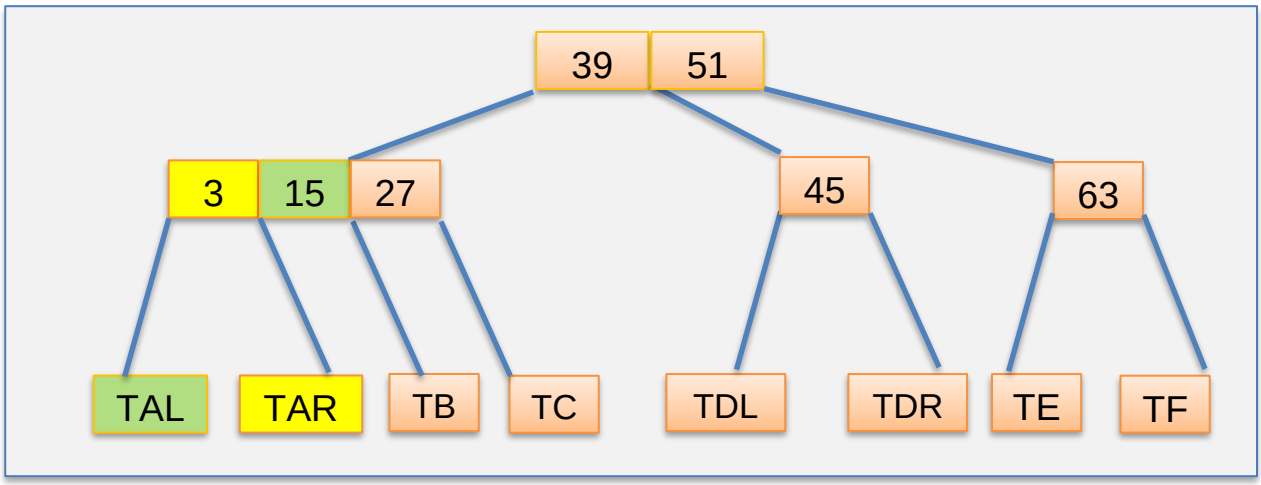


Figure 8 Intermediate state before promotion of key 15

As shown in Figure 8, Key 3 find its place in of BTreeNode [15 | 27]. It has now got two children which are denoted as left tree [TAL] and right tree [TAR]. The node is now in overflow condition which results in promoting its middle node i.e. key 15 to its parent [39 | 51]

Finally, after promotion of Key 15 takes place, the final BTree will look like this

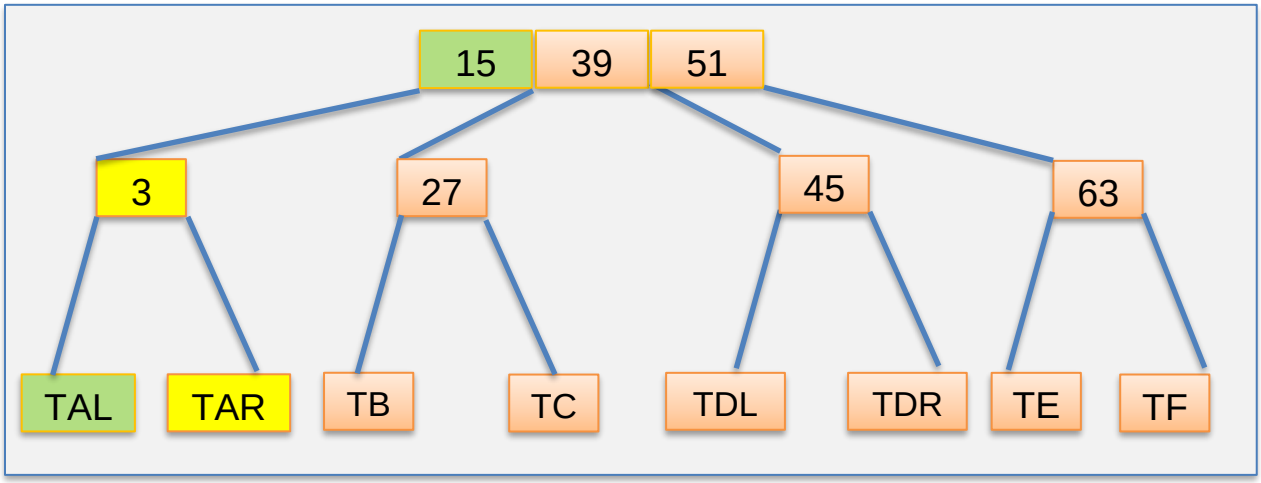


Figure 9 Promotion of key 15 in Extreme Left

Here is the state change view of each affected node in the current BTree after promoting Key 51.

Before Promotion of 51	After Promotion of 51	State
39.LST = [3 | 15 | 27]	39.LST = 27	CHANGED
39.RST = 45	39.RST = 45	SAME
39.NEXT = 51	39.NEXT = 51	SAME
39.PREV = NULL	39.PREV = 15	CHANGED
15.PARENT = 39	15.PARENT of [39 | 51]	CHANGED
15.NEXT = 27	15.NEXT = 39	CHANGED
15.PREV = NULL	15.PREV = NULL	SAME
15.LST = TA	15.LST = 3	CHANGED
15.RST = TB	15.RST = 27	CHANGED
27.LST = NULL	27.LST = TB	CHANGED
27.RST = TC	27.RST = TC	SAME
27.PREV = 15	27.PREV = NULL	CHANGED
3.LST (New Insertion)	3.LST = TAL	CHANGED

3.RST (New Insertion)	3.RST = TAR	CHANGED

In this whole promotion episode, subtrees belonging to 51, 45 and 63 will maintain its Status Quo.

4.3.3 Promotion at Intermediate Position

So far, we have covered promotion of internal nodes key into its parent in extreme left and right position.
There is however, one scenario pending in which the promotion of an internal node key goes into the intermediate position of its parent i.e., it neither go left nor go right.

In the figure 10 we are going to consider this pending scenario. A new key 35 will inserted first into target leaf node TC. From there it will be promoted to internal node [15 | 27]

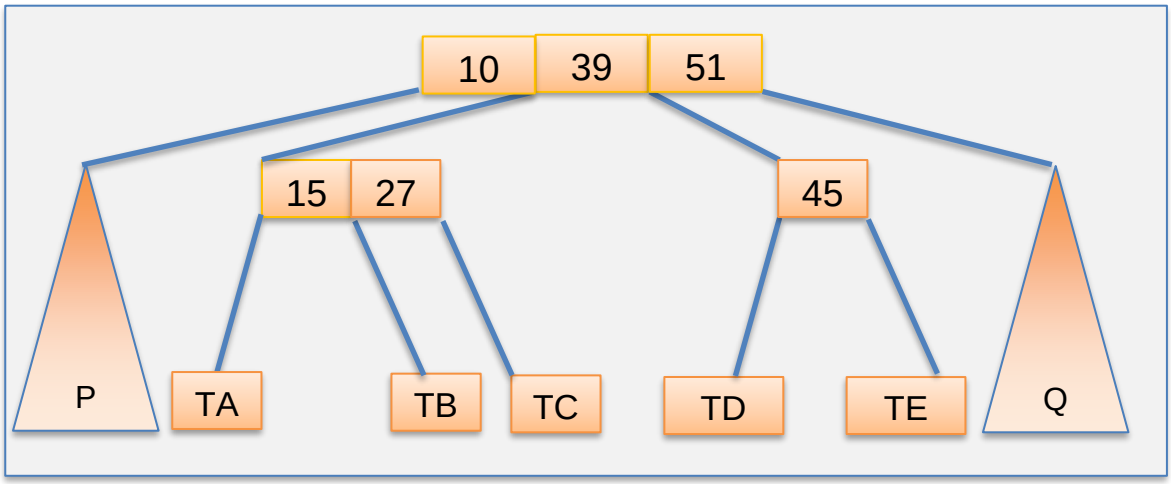


Figure 10

The intermediate state of BTree when the new key 35 is placed into the internal BTreeNode [15 | 27] is shown as below. Note that the extreme left and extreme right subtrees are denoted by P and Q respectively.

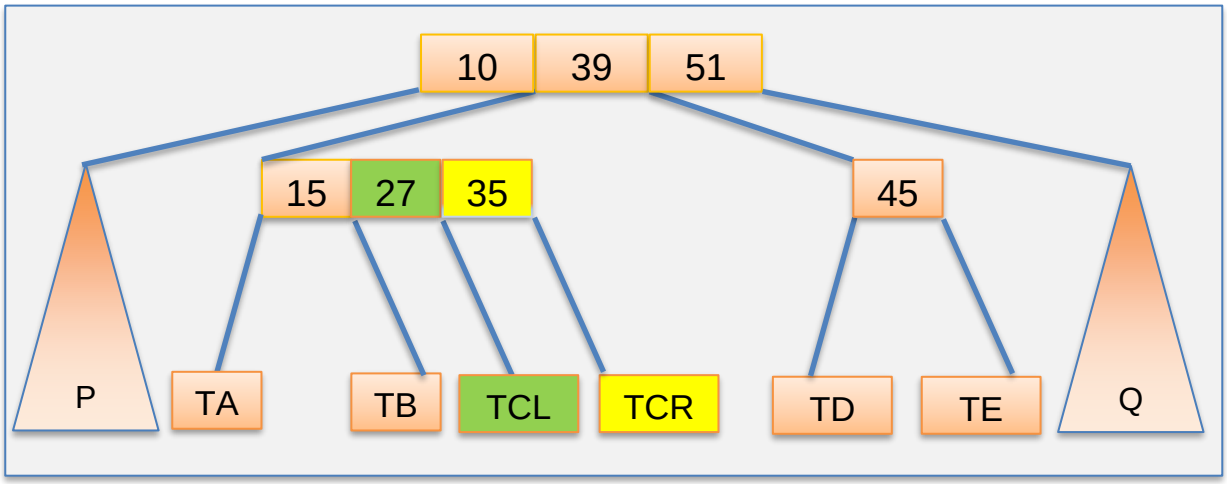


Figure 11 Before promotion of key 27

Nodes which are yellow in color are the ones for which the memory is freshly allocated.
Green color nodes are the ones which got impacted due to the promotion of key 35.
So, to summarize:

1. Inserted key 35 at leaf target node level
2. Key 35 promoted to internal node
3. Key 27 present in internal node promoted to its parent due to the overflow condition of promotion of key 35 in internal node.

After succesful insertion and promotions, BTree will finally look like this:

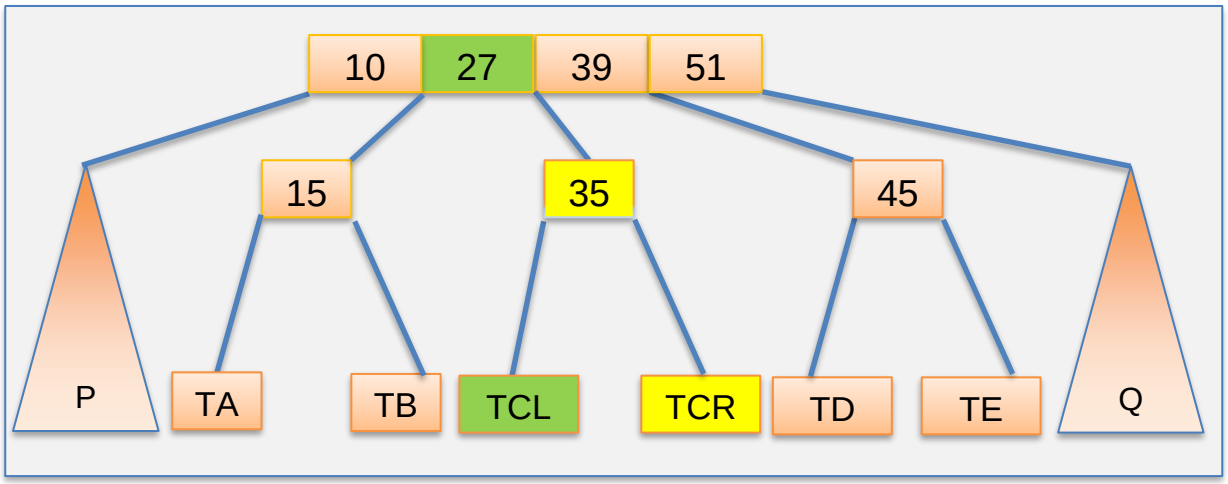


Figure 12 Promotion of Key 27 at Intermediate Position

Before Promotion of 51	After Promotion of 51	State
10.LST = P	10.LST = P	SAME
10.RST = [15 | 27]	10.RST = 15	CHANGED
10.NEXT = 39	10.NEXT = 27	CHANGED
39.PREV = 10	39.PREV = 27	CHANGED
15.NEXT = 27	15.NEXT = NULL	CHANGED
15.PARENT = 10	15.PARENT = 10	SAME
27.PREV = 15	27.PREV = 10	CHANGED
27.LST = NULL	27.LST = NULL	SAME
27.RST = TCL	27.RST = 35	CHANGED
27.NEXT = 35	27.NEXT =39	CHANGED

35.LST = NEW NODE	35.LST = TCL	CHANGED
35.RST = NEW NODE	35.RST = TCR	CHANGED

4.4 CASE-3: Root Promotion

The scenario in Figure 9 to 12 highlights one more promotion i.e., promotion of an intermediate node key to become the new root of the BTree.

E.g. the root node in figure 9 has got 3 keys [15 | 39 | 51] and figure 12 has got 4 keys in its root node [10 | 27 | 39 | 51]
For an order 3 BTree, the current root node [15 | 39 | 51] has overflowed. It essentially means that this node needs to be further divided into three parts i.e., root, left and right subtrees.
This promotion is called Root Node Promotion. In this scenario, the new root happens to be the middle node among the list of all nodes present in the current/old root. Root Node promotion increases the height of the BTree .

Here is the pictorial representation of before and after Root promotion

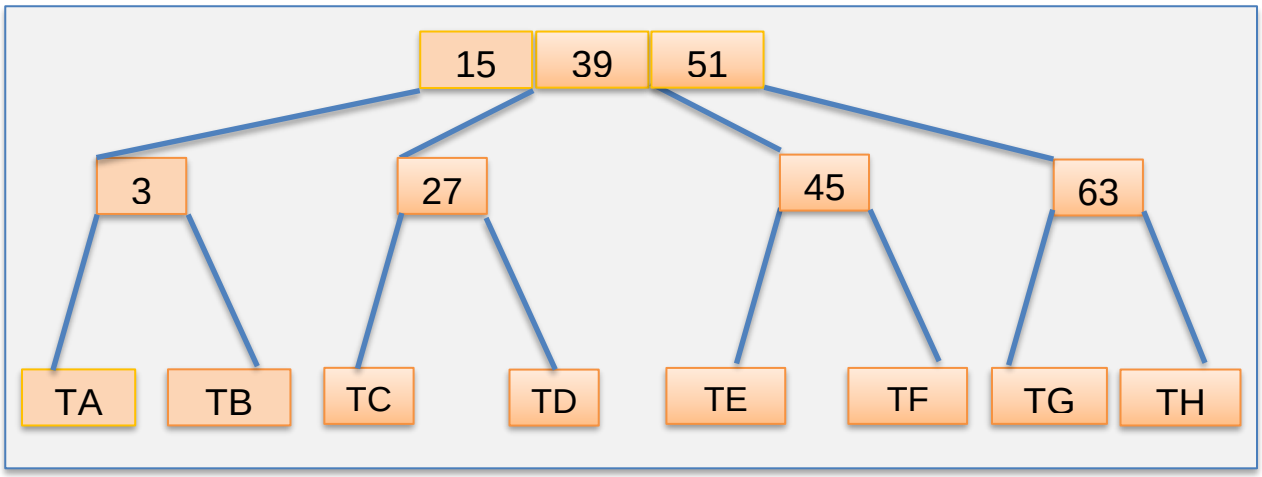


Figure 13 (Before Root Node Promotion)

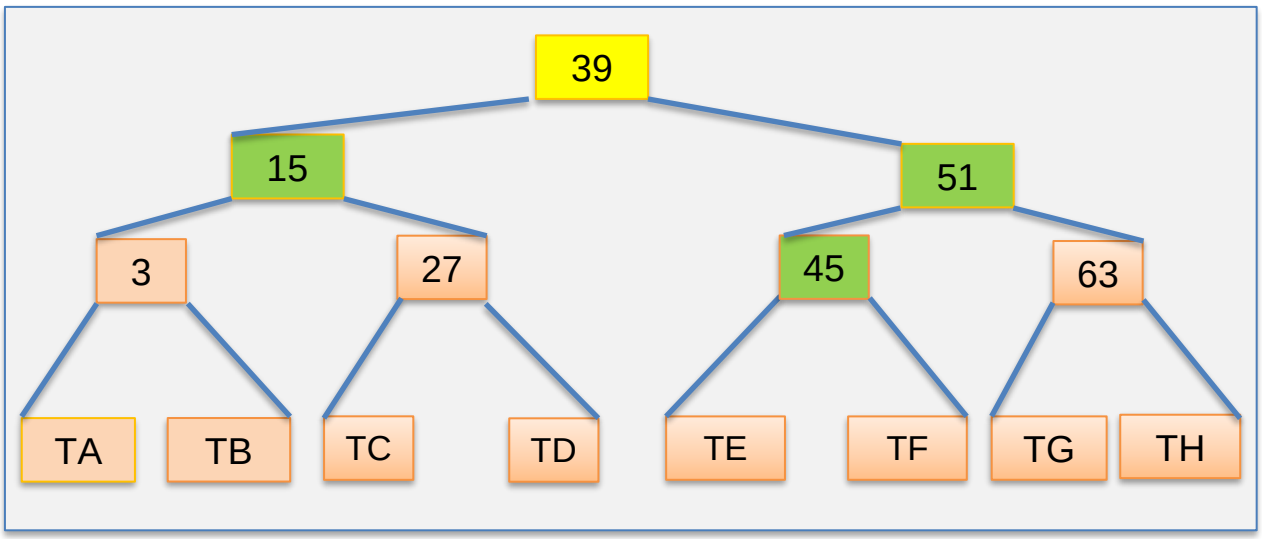


Figure 14 (Root Promotion)

5 BTree Searching

Searching process starts from the Root node of BTree and it follows the Binary Search Tree logic.
There are two types of searching functions defined.

1. findLeafBTreeNode()
2. searchBTreeNode()

The difference between findLeaf and search fucntion is that, findLeaf searches the BTreeNode in which new key is going to be inserted. searchBTreeNode function returns the exact match of key in the form of SearchResult Object.

5.1 findleafBTreeNode:

This function solves the purpose of finding the target leaf node which is going to be used for new key insertion. Here is the core logic of this function.

```
proc findLeafBTreeNode*(root: ref BTreeNode, key: int): ref BTreeNode =
#{
  if (root == nil or root.bt_listhead == nil or root.bt_listtail == nil):
    return nil
  var minkey = root.bt_listhead.key
  var maxkey = root.bt_listtail.key

  if (key > minkey and key < maxkey): #{
    var node = root.bt_listhead.next
    var prev = root.bt_listhead

    while(node != nil): #{
      if (key > prev.key and key < node.key):
        var right = getBTreeNodeRight(prev)
        if (right != nil): #{
         var bnode = findLeafBTreeNode(right, key)
          if (bnode != nil): #{
            return bnode
          #}
        #}
      #} End of first if block
      prev = node
      node = node.next
    #} End of while loop
```

```
    return root
  #} End of top level if condition

  if (key < minkey): #{
    var left = getBTreeNodeLeft(root)
    if (left == nil):
      return root
    else:
      return findLeafBTreeNode(left, key)
  #}

  if (key > maxkey): #{
    var right = getBTreeNodeRight(root)
    if (right == nil):
      return root
    else:
      return findLeafBTreeNode(right, key)
  #}
#}
```

5.2 searchBTreeNode:

```
proc searchBTree*(root: ref BTreeNode, key: int): ref SearchResult =
#{
  if (root == nil):
    return nil

  var head = root.bt_listhead
  var srobj: ref SearchResult = nil

  while(head != nil): #{

    if (key == head.key):

      srobj               = new SearchResult
      srobj.btreenode = root
      srobj.listnode     = head
      return srobj
```

```
    if (key < head.key):
      srobj = searchBTree(getBTreeNodeLeft(head), key)

    if (key > head.key):
      srobj = searchBTree(getBTreeNodeRight(head), key)

    if (srobj == nil):
      head = head.next
    else:
      return srobj
  #} End of while
#}
```

6 BTree Traversal

The inorder traversal of BTree keys will provide the output in an ascending order. getInorder() function implement this traversal in recursive manner.

```
proc getInorder*(root: ref BTreeNode): void =
#{
  if (root == nil):
    return

  var node: ref ListNode = root.bt_listhead
  if (node == nil):
    printf("node is null\n")

  while true:
    var left  = getBTreeNodeLeft(node)
    var right = getBTreeNodeRight(node)

    getInorder(left)
    inorderstr.add("" & $node.key)
    inorderstr.add(" ")
    inordercnt = inordercnt + 1
    getInorder(right)

    node = node.next
    if (node == nil):
```

```
    break
#}

proc inorderBTree*(root: ref BTreeNode): void =
#{
   inorderstr = ""
   inordercnt = 0
   echo ""
   inorderstr.add("Inorder string  :[ ")
   getInorder(root)
   inorderstr.add("] Inorder Count : " & $inordercnt)
   log(inorderstr)
   logf(inorderstr)
   echo ""
#}
```

7 BTree Deletion

Before going through the steps below, one must know these facts about a B tree of degree m.

A node can have a maximum of m children. (i.e. 3)
A node can contain a maximum of m - 1 keys. (i.e. 2)
A node should have a minimum of ⌈m/2⌉ children. (i.e. 2)
A node (except root node) should contain a minimum of ⌈m/2⌉ - 1 keys. (i.e. 1)

Deletion in a BTree can happen at three levels:

1. Leaf Node level
2. Internal node level
3. Root level.

We will examine all the cases for each level. Let's begin with Leaf Level

7.1 Leaf Level

Deletion at leaf level also comes with multipe scenarios. We need to identify those scenarios one by one

7.1.1 Leaf Node having keys > BT_MINKEYS

When a key-to-be-deleted is a part of leaf node and its numkeys is greater than minkeys. This is a normal node deletion from a Doubly Linked List (DLL). We have maintained a DLL of ListNodes as part of BTreeNode, so only next and prev pointers are adjusted during this scenario.

Before deletion of key a6
Inorder Traversal : [a1 a2 A a4 a5 **a6** B b1 b2]

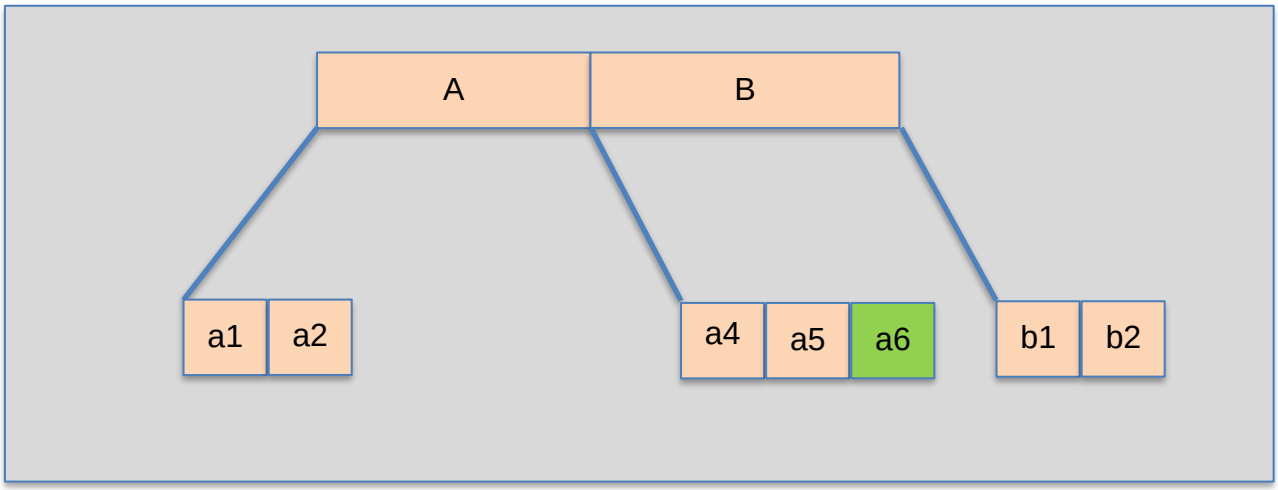


Figure 15 : Deletion of a Leaf Node a6

After deletion of key a6
Inorder Traversal : [a1 a2 A a4 a5 ~~a6~~ B b1 b2]

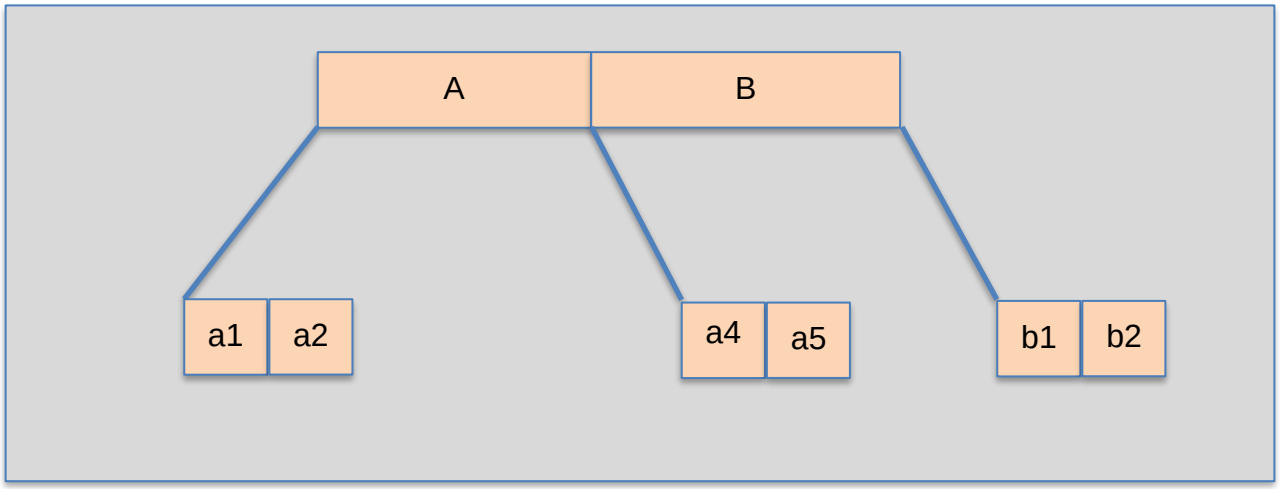


Figure 16 : Deletion of a Leaf Node a6

7.1.2 Leaf Node having keys equals to BT_MINKEYS

When a key-to-be-deleted is a part of leaf node and its numkeys is equal to minkeys but the left and/or right sibling's numkeys of leaf node are greater than minkeys.
In this case, borrowing of a key from left or right siblings takes place.
If borrowed from left sibling then a clockwise shift will happen from left to right
If borrowed from right sibling then an anti clockwise shift will happend from right to left.
Priority will be given to clockwise shift if number of keys in both the siblings are same and greater than leaf node numkeys.

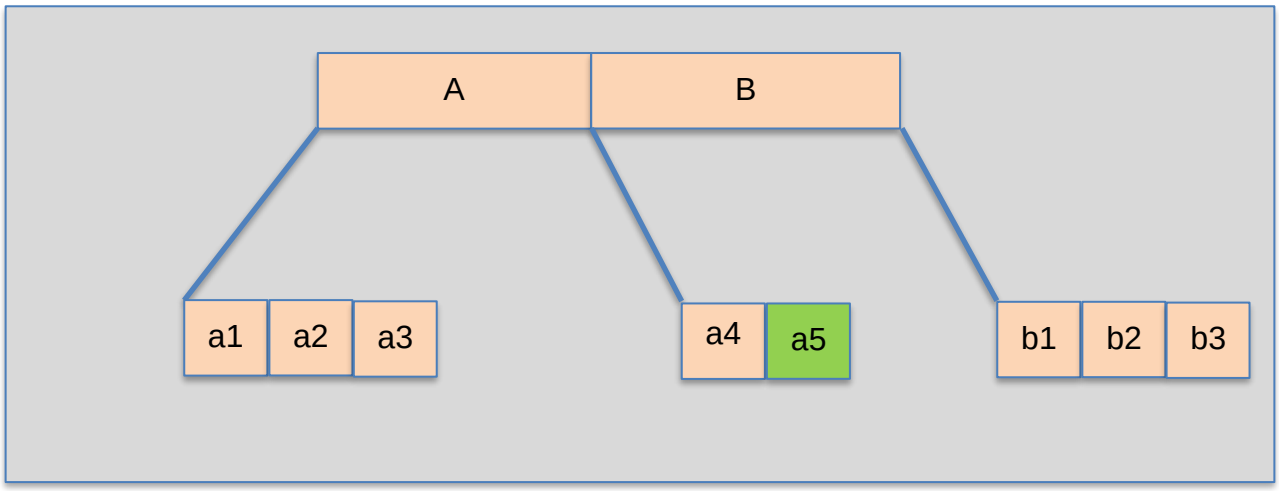


Figure 17 : Deletion of a Leaf Node a5

In Figure 17 key a5 is going to be deleted from the node [a2 | a5]. This node already has its BT_MINKEYS so this node as such can't be deleted. The concept of borrowing will take place here. Will borrow the key a3 from left sibling [a1 | a2 | a3] node. After borrowing Parent key A will be moved to its right child [a4 | a5] Here is the alog for borrowing.

Note: We could have borrowed a key b1 from right sibling [b1 | b2 | b3] also but that would have been happened only when the left sibling had not got extra keys.

Algorithm:
1. Disconnect key a3 from a2
2. Decrement numkeys count from a1 node
3. copy a3 key to parent A so that a3 will become new parent

4. copy parent A key to a4 so that A will become new right child
5. Now delete the memory of a3 node

We need to ensure that the inorder traversal should not be broken deletion.
Inorder traversal
Before deletion : a1 a2 a3 A a4 a5 B b1 b2
After deletion : a1 a2 a3 A a4 ~~a5~~ B b1 b2

Here is the pictorial represtation after deletion of a5 node. Here the parent key A is demoted to its child node.

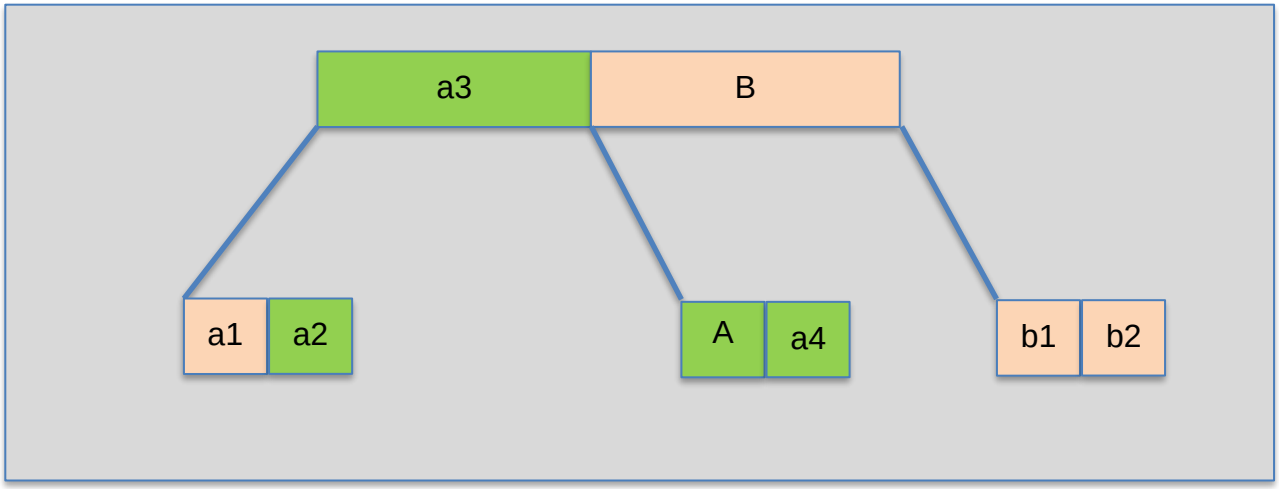


Figure 18 : After Deletion of key a5

Here is the code overview

proc deleteListNodeLeafCase2(btnode: var ref BTreeNode, delnode: var ref ListNode): void =

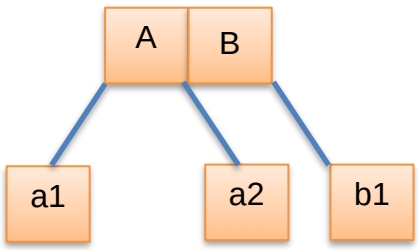
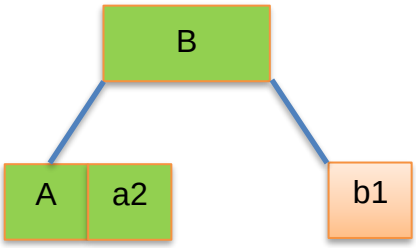
```
#{

    log("Will have to borrow the keys first from left or right siblings\n")

    # CASE-1: Borrow from Left Sibling
    var leftsib = getBTreeNodeLeftSibling(delnode.parentListNode)

    if ((leftsib != nil) and (getBTreeNodeNumKeys(leftsib) > BT_MINKEYS)):
```

```
    borrowFromLeftSibling(delnode, leftsib)
    return
  # CASE-2: Borrow from Right Sibling
  var rightsib = getBTreeNodeRightSibling(delnode.parentListNode)

  if ( (rightsib != nil) and (getBTreeNodeNumKeys(rightsib) > BT_MINKEYS)):
    borrowFromRightSibling(delnode, rightsib)
    return

  # CASE-3: Merge left and right siblings
  # we have come here because both left and right siblings have exactly
  # BT_MINKEYS,
#}
```

7.1.3 Leaf Node and its siblings having keys equals to BT_MINKEYS

When a key-to-be-deleted is a part of leaf node whose numkeys is equal to minkeys and the left and right sibling of the leaf node also have numkeys equal to minkeys
In this case merging of parent with child nodes takes place. There are 3 scenarios which can be considered here.

Scenario#1

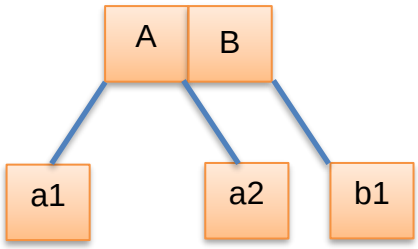
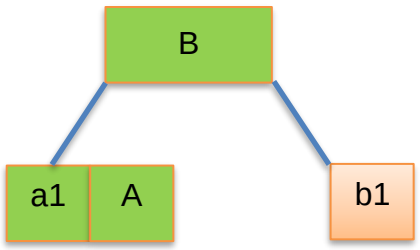
	
Before Deletion of key a1 Inorder : a1 A a2 B b1	After deletion of key a1 Inorder : A a2 B b1

Algorithm#1:

1. Remove Parent Child relationship between A and a1 and make them independent nodes

2. Merge 'A' with a2 i.e [A - a2] in sorted manner
3. Make Parent Child relation ship between B and new merger i.e [A - a2]
4. Delete a1 BTreeNode memory
5. Decrement numkeys of Parent BTreeNode B
6. Increment numkeys of new merged child [A - a2]

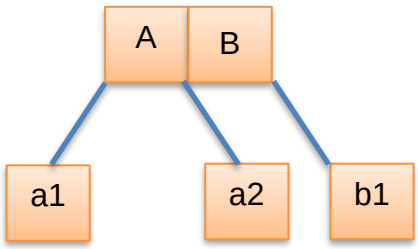
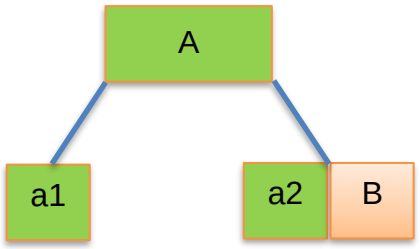
Scenario#2:

	
Before Deletion of key a2 Inorder : a1 A a2 B b1	After deletion of key a2 Inorder : a1 A B b1

Algorithm#2:

1. Remove Parent Child relationship between A and a2 and make them independent nodes
2. Merge 'a1' with A i.e. [a1 - A] in sorted manner
3. Make Parent Child relation ship between B and new merger i.e [a1 - A]
4. Delete a2 BTreeNode memory
5. Decrement numkeys of Parent BTreeNode B
6. Increment numkeys of new merged child [a1 - A]

Scenario#3:

	
Before Deletion of key b1 Inorder : a1 A a2 B b1	After deletion of key b1 Inorder : a1 A a2 B

Algorithm#3:

1. Remove Parent Child relationship between B and b1 and make them independent nodes
2. Merge 'B' with a2 i.e. [a2 - B] in sorted manner
3. Delete b1 BTreeNode memory
4. Decrement numkeys of Parent BTreeNode A
5. Increment numkeys of new merged child [a2 - B]

7.2 Intermediate node level

Inprogress, Stay Tuned !

7.3 Root level

Inprogress, Stay Tuned !

8 References

1. https://narimiran.github.io/nim-basics/
2. https://nim-lang.org/docs/lib.html
3. https://en.wikipedia.org/wiki/B-tree
4. https://www.cs.usfca.edu/~galles/visualization/BTree.html
5. https://nim-lang.org/docs/compiler/btrees.html
6.