

# B-Trees

Martín Hernández  
Juan Mendivelso

# Contents I

## B-Tree

History

Definition

Properties

- The  $\alpha$  Constant

- Keys and Sub-trees

- Height

Structure

Operations

- Creating an empty B-Tree

- Search value

- Insert value

- Delete value

Secondary Memory Access

## Bibliography

# B-Tree History I

B-Trees were firstly studied, defined and implemented by R. Bayer and E. McCreight in 1972, using an IBM 360 series model 44 with an 2311 disk drive.



**Figure:** IBM 360 / 44

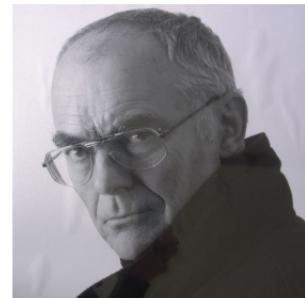
An IBM 360 series model 44 had from 32 to 256 *KB* of Random Access Memory, and weighed from 1,315 to 1,905 kg.



**Figure:** IBM 2311 disk drive

## B-Tree History II

"(...) actual experiments show that it is possible to maintain an index of size 15.000 with an average of 9 retrievals, insertions, and deletions per second in real time on an IBM 360/44 with a 2311 disc as backup store. (...) it should be possible to main tain all index of size 1'500.000 with at least two transactions per second." (Bayer and McCreight)



**Figure:** Rudolf Bayer



**Figure:** Edward McCreight

## B-Tree Definition I

- We will define that  $T$ , an object, is a B-Tree if they are an instance of the class.

$$T \in t(\alpha, h)$$

- Where  $h$  is the height of the B-Tree.
- And,  $\alpha$  is a predefined constant.
- This type of balanced tree have a higher degree than the previous trees.
- Or in simple words, they have more than 1 key and 2 sub-trees in each node.
- Keep in mind that in B-Trees, **leafs are not nodes**.
- This higher degree have a couple of properties added to it, which we need to check and prove
- Also, due to the higher degree of the nodes, we will have to change the `find`, `insert` and `delete` operations of the B-Tree.

## B-Tree Definition II

d	n	p	n		h	n
0		1		...	k	
		20		...	n	
.	.	.	.	.	.	.

Figure: Node of a B-Tree

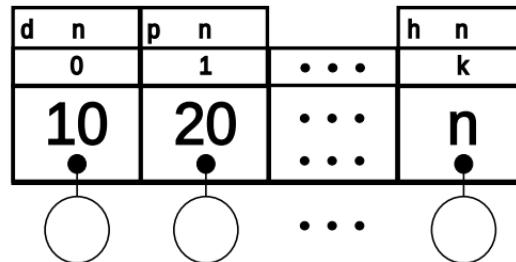


Figure: Leaf of a B-Tree

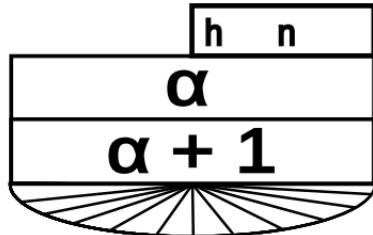
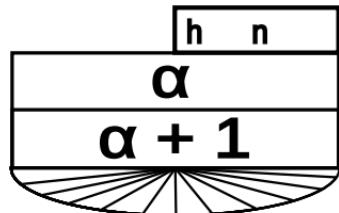


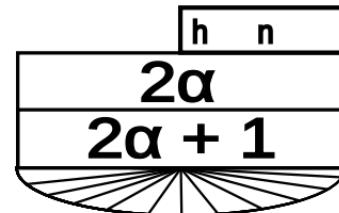
Figure: Generic Node of a B-Tree

## B-Tree Properties - The $\alpha$ constant I

- ▶ The main property of the B-Trees is the  $\alpha$ , a predefined constant.
- ▶ The  $\alpha$  must be a Natural number,  $\alpha \in \mathbb{N}$  and  $\alpha \geq 2$ .
- ▶ This constant will determine the interval of keys and sub-trees, in a balanced node. This is called the *Branching factor* of the tree.
- ▶ The tree is balanced if they have from  $\alpha + 1$  to  $2\alpha + 1$  sub-trees in a single node.
- ▶ Also, each balanced node have from  $\alpha$  to  $2\alpha$  keys.
- ▶ The only node that can have less than  $\alpha + 1$  sub-trees and only 1 key is the *Root* of the tree.
- ▶ But, the *Root* still have the upper bounds of sub-trees and keys.



**Figure:** Minimum Keys and Sub-Trees on a Node



**Figure:** Maximum Keys and Sub-Trees on a Node

## B-Tree Properties - The $\alpha$ constant II

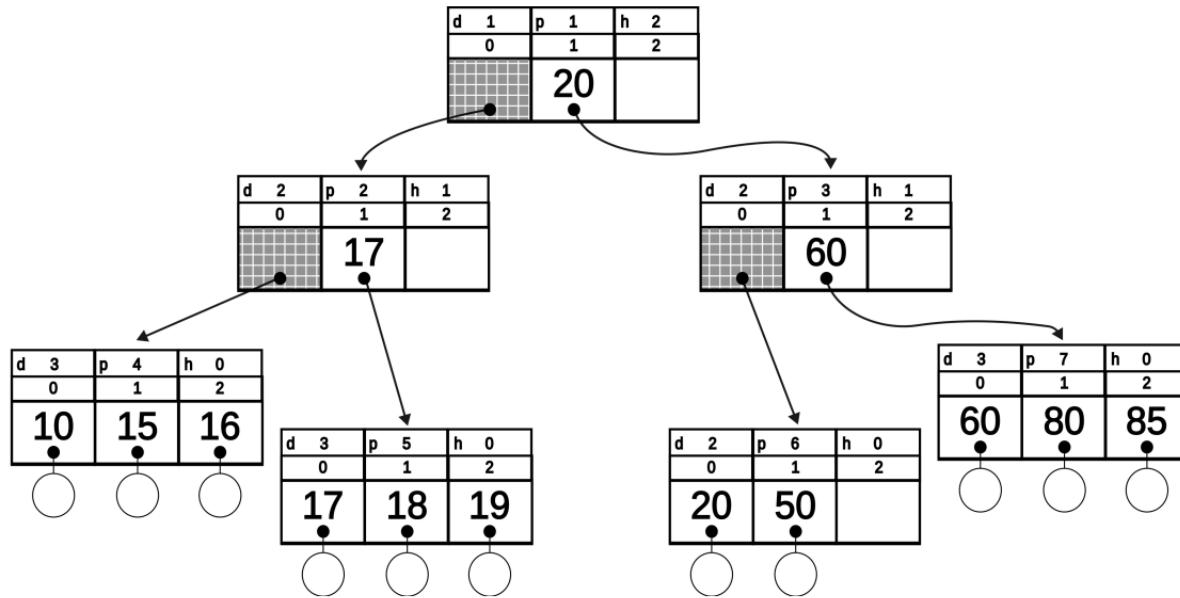


Figure: B-Tree,  $t(2, 2)$

## B-Tree Properties - The $\alpha$ constant III

- We can prove the bounds of the number of sub-trees in a node, and define a function that let us get the number of sub-trees in a node.

### Proof.

Let  $T \in t(\alpha, h)$ , and  $N(T)$  be a function that returns the number of nodes in  $T$ . Let  $N_{\min}$  and  $N_{\max}$  the minimum and maximal number of nodes in  $T$ . Then

$$\begin{aligned}N_{\min} &= 1 + 2 \left( (\alpha + 1)^0 + (\alpha + 1)^1 + \dots + (\alpha + 1)^{h-2} \right) \\&= 1 + 2 \left( \sum_{i=0}^{h-2} (\alpha + 1)^i \right) \\&= 1 + \frac{2}{\alpha} \left( (\alpha + 1)^{h-1} - 1 \right)\end{aligned}$$

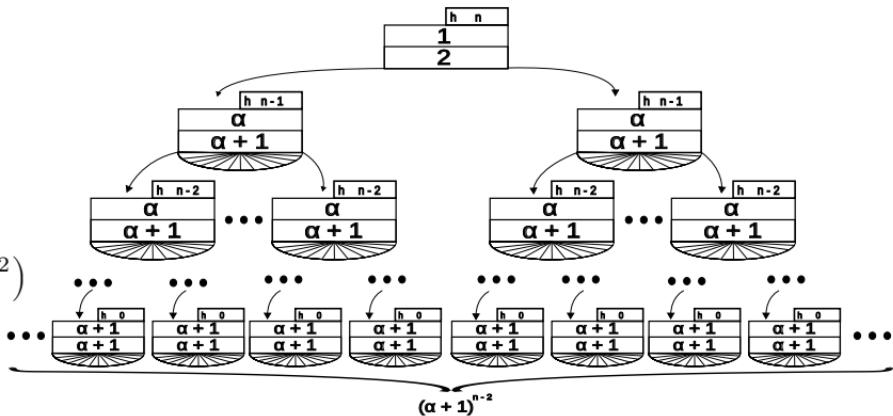
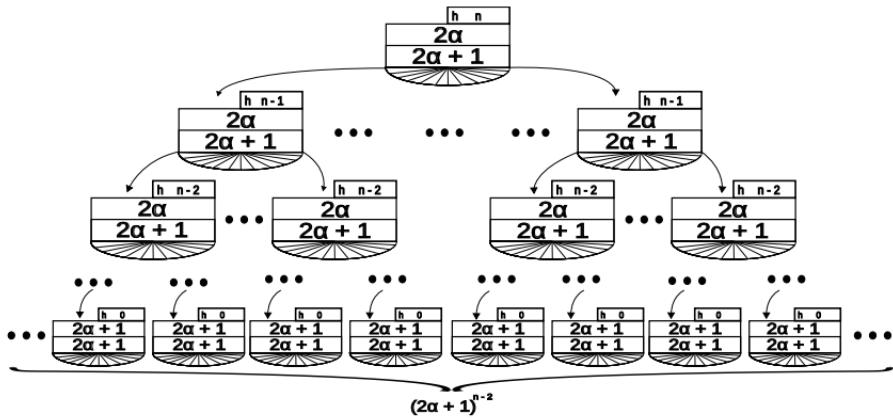


Figure: B-Tree w/ the least number of nodes

## B-Tree Properties - The $\alpha$ constant IV

For  $h \geq 1$ , we also have that

$$N_{\max} = 2 \left( \sum_{i=0}^{h-1} (2\alpha + 1)^i \right) \\ = \frac{1}{2\alpha} \left( (2\alpha + 1)^h - 1 \right)$$



**Figure:** B-Tree w/ the most number of nodes

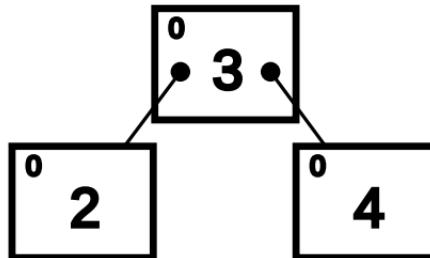
Then, if  $h = 0$ , we have that  $N(T) = 0$ . Else, if  $h \geq 1$

$$1 + \frac{2}{\alpha} \left( (\alpha + 1)^{h-1} - 1 \right) \leq N(T) \leq \frac{1}{2\alpha} \left( (2\alpha + 1)^h - 1 \right) \quad (\text{Nodes Bounds})$$

□

## B-Tree Properties - Keys and Sub-trees I

- ▶ Each key has two sub-trees, one before and one after it. Like a normal tree.
- ▶ First, let's define  $N$ , a Node which isn't a leaf or *Root*, from a B-Tree.
- ▶ Then, we can define the set of the keys on a B-Tree Node  $N$  as  $\{k_1, k_2, \dots, k_j\}$ .
- ▶ Leaving the index 0 for a placeholder, which is going to be used later.
- ▶ Also, defining  $l$  as the number of keys in  $N$ .
- ▶ Such that for  $t(\alpha, h)$ , we have  $\alpha \leq l \leq 2\alpha$ .



**Figure:** Simple node of a Normal Binary Tree

## B-Tree Properties - Keys and Sub-trees II

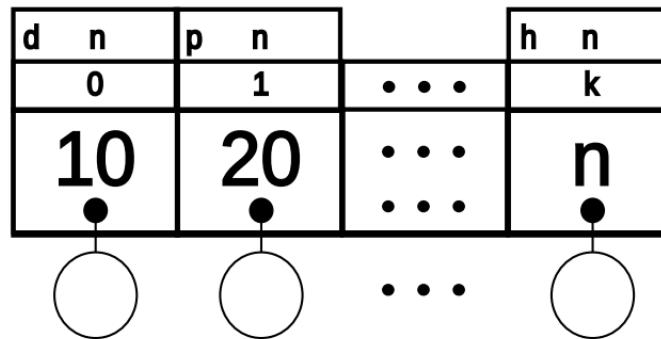
- ▶ Now, we also define the set of sub-trees of  $N$  as  $\{p_0, p_1, \dots, p_j\}$ .
- ▶ Where  $j$  is the number of sub-trees in  $N$ .
- ▶ Since there's a sub-tree before and after each key in  $N$ .
- ▶ Then,  $j$  must be equal to  $l + 1$ .
- ▶ The keys and sub-trees are stored in a sequential increasing order.

The diagram shows a sequence of alternating elements: Subtree Pointers ( $p$ ) and Keys ( $k$ ). The sequence starts with  $p_0$ , followed by  $k_1$ , then  $p_1$ , then  $k_2$ , then  $p_2$ , then  $k_3$ , then  $p_3$ , then three dots ( $\dots$ ), then  $p_{i-1}$ , then  $k_i$ , then  $p_i$ , and finally another three dots ( $\dots$ ). The pointers are in bold, and the keys are in regular font.

**Figure:** Order of the Subtree Pointers and Keys.

## B-Tree Properties - Keys and Sub-trees III

- ▶ In the case that  $N$  is the *Root* of the tree, the only change is the minimum number of keys and sub-trees.
- ▶ With  $l$ , already defined, *Root* will have  $1 \leq l \leq 2\alpha$  keys.
- ▶ And  $2 \leq l + 1 \leq 2\alpha + 1$  sub-trees.
- ▶ If  $N$  is a leaf of the tree, we are going to give the  $k_0$  a simple use.
- ▶ The  $k_0$  will store a key value for an object.
- ▶ This simple usage on a leaf is just one usage of the  $k_0$  on the nodes.



**Figure:** Leaf of a B-Tree

## B-Tree Properties - Keys and Sub-trees IV

- ▶ Going back where  $N$  is a node on the B-Tree, but now this time  $N$  can be the tree *Root*.
- ▶ The order of the keys of  $p_i$ , a subtree of  $N$ ; where  $0 \leq i \leq l$ , in comparison to the keys of  $N$  can be defined by 3 cases.
- ▶ But first, we need to define  $K(T)$ , where  $T \in t(\alpha, h)$ , which is the set of keys inside the Node  $T$ .
- ▶ And,  $k_j \in K(N)$ , where  $j$  is the index or position of the key in  $N$ .

$$\forall y \in K(p_0); \quad y < k_1 \quad (\text{Case 1})$$

$$\forall y \in K(p_i); \quad k_i \leq y < k_{i+1}; \quad 0 < i < l \wedge i \in \mathbb{N} \quad (\text{Case 2})$$

$$\forall y \in K(p_l); \quad k_l \leq y \quad (\text{Case 3})$$

## B-Tree Properties - Keys and Sub-trees V

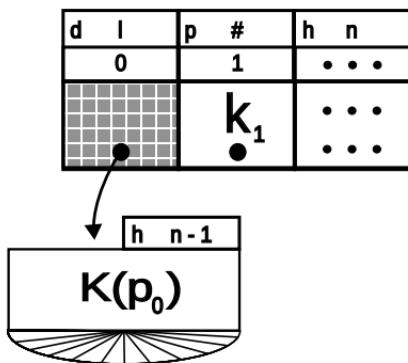


Figure: Sub-tree Keys (Case 1)

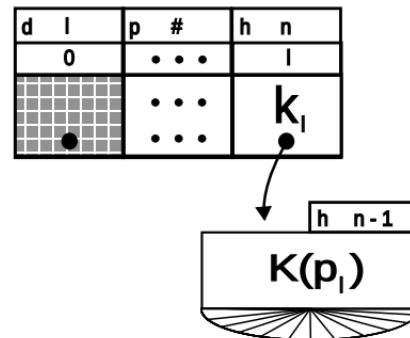


Figure: Sub-tree Keys (Case 3)

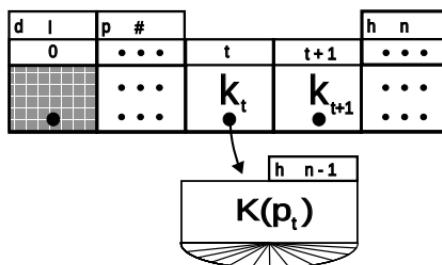


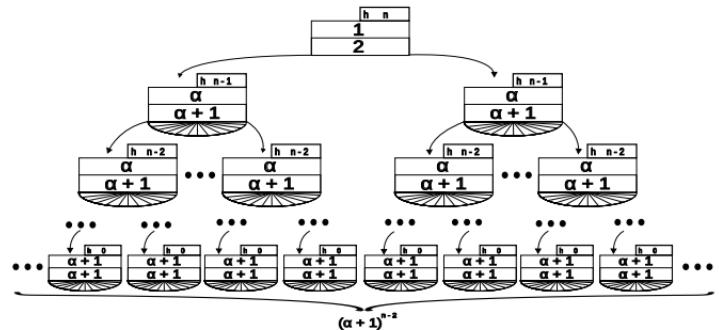
Figure: Sub-tree Keys (Case 2)

## B-Tree Properties - Height I

- Before we can define and prove the height of a B-Tree we need to define some things.
- First, The set of the keys in  $T \in t(\alpha, h)$  will be defined as  $I$ .
- Now, The  $I_{\min}$  and  $I_{\max}$  of  $T$  can be easily defined by (Nodes Bounds):

$$1 + 2 \frac{((\alpha + 1)^{h-1} - 1)}{\alpha} \leq N(T) \leq \frac{((2\alpha + 1)^h - 1)}{2\alpha}$$

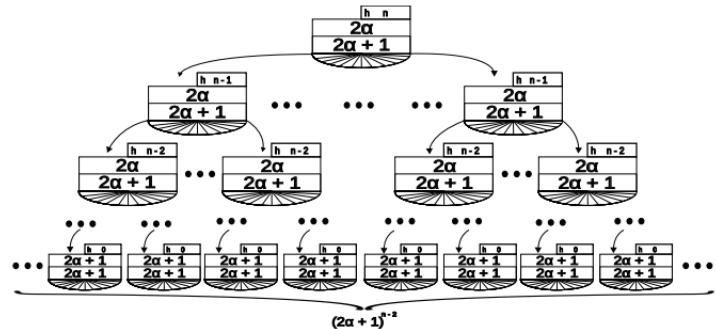
$$\begin{aligned}I_{\min} &= 1 + \alpha (N_{\min}(T) - 1) \\&= 1 + \alpha \left( \frac{2(\alpha + 1)^{h-1} - 2}{\alpha} \right) \\&= 2(\alpha + 1)^{h-1} - 1\end{aligned}$$



**Figure:** B-Tree w/ the least number of nodes

## B-Tree Properties - Height II

$$\begin{aligned} I_{\max} &= 2\alpha (N_{\max}(T)) \\ &= 2\alpha \left( \frac{(2\alpha + 1)^h - 1}{2\alpha} \right) \\ &= (2\alpha + 1)^h - 1 \end{aligned}$$



**Figure:** B-Tree w/ the most number of nodes

► Now, we can solve for  $h$  with each bound of  $I$  and define an bound of  $h$  with them.

$$\begin{aligned} I_{\min} &= 2(\alpha + 1)^{h-1} - 1 \\ \frac{I_{\min+1}}{2} &= (\alpha + 1)^{h-1} \\ \log_{\alpha+1} \left( \frac{I_{\min} + 1}{2} + 1 \right) &= h_{\min} \end{aligned}$$

$$\begin{aligned} I_{\max} &= (2\alpha + 1)^h - 1 \\ I_{\max} + 1 &= (2\alpha + 1)^h \\ \log_{2\alpha+1} (I_{\max} + 1) &= h_{\max} \end{aligned}$$

## B-Tree Properties - Height III

- ▶ Since,  $2\alpha + 1 > \alpha + 1$ , then  $\log_{2\alpha+1}x \leq \log_{\alpha+1}x$ , both in  $[1, \infty)$ .
- ▶ Or also, if we have more nodes in a B-Tree, the height of the Tree will be less than if we have less nodes in the B-Tree.
- ▶ Hence, for  $I \geq 1$ , we will have the bounds for  $h$ :

$$\log_{2\alpha+1}(I+1) \leq h \leq \log_{\alpha+1}\left(\frac{I+1}{2} + 1\right)$$

- ▶ And if,  $I = 0$  then,  $h = 0$ .

## B-Tree Properties - Summary

- ▶ A B-Tree is defined as:  $T \in t(\alpha, h)$
- ▶ A B-Tree has a predefined constant  $\alpha$ .
- ▶ Node can have  $\alpha \leq I \leq 2\alpha$  keys.
- ▶ Also, it has  $\alpha + 1 \leq I + 1 \leq 2\alpha + 1$  sub-trees.
- ▶ Except the *Root* node, which can have at least 1 key and 2 sub-trees.
- ▶ The leafs use the  $k_0$  space to store object key information.
- ▶ For each key on sub-tree of a Node, there's 3 cases:

$$\forall y \in K(p_0); \quad y < k_1$$

$$\forall y \in K(p_i); \quad k_i \leq y < k_{i+1}; \quad 0 < i < l \wedge i \in \mathbb{N}$$

$$\forall y \in K(p_l); \quad k_l \leq y$$

- ▶ The number of nodes of a B-Tree is bounded by:  $1 + \frac{2}{\alpha} ((\alpha + 1)^{h-1} - 1) \leq N(T) \leq \frac{1}{2\alpha} ((2\alpha + 1)^h - 1)$
- ▶ The number of Keys in a B-Tree is bounded by:  $2(\alpha + 1)^{h-1} - 1 \leq I \leq (2\alpha + 1)^h - 1$
- ▶ The height of a B-Tree is bounded by:

$$\log_{2\alpha+1}(I + 1) \leq h \leq \log_{\alpha+1}\left(\frac{I + 1}{2} + 1\right)$$

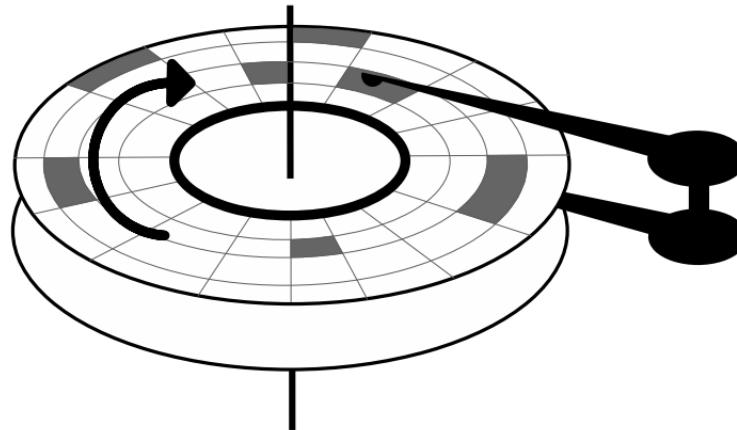
# B-Tree Structure

- The structure of the B-Tree's node adds two arrays where the keys and sub-trees' pointers will be stored:

```
1 int ALPHA = 2; /* any int >= 2 */
2 typedef struct tr_n_t {
3     int degree;
4     int height;
5     key_t key[(2 * ALPHA) - 1];
6     struct tr_n_t *next[(2 * ALPHA) - 1];
7     /* ... */
8 } tree_node_t;
```

## B-Tree Operations

- ▶ For these operations, we will assume that the whole B-Tree is loaded into main memory.
- ▶ We have to assume this since the main usage of the B-Tree is oriented to secondary storage.
- ▶ Generally, only the *Root* and node to operate, if available, will be always available in memory.
- ▶ But if we need any other node, we will have to read into our secondary memory and fetch its data.
- ▶ This process takes more time than the general data fetch from main memory.
- ▶ So, the fewer times we do this process the better.



**Figure:** External storage with the sectors to access highlighted

## B-Tree Operations - Creating an empty B-Tree

- We use `create_tree()` to create an empty B-Tree, and since we only need to use `get_node()`, this operation takes  $\Theta(1)$ .

```
1 tree_node_t *create_tree() {  
2     tree_node_t *tmp;  
3     tmp = get_node();  
4     tmp->height = 0;  
5     tmp->degree = 0;  
6     return( tmp );  
7 }
```

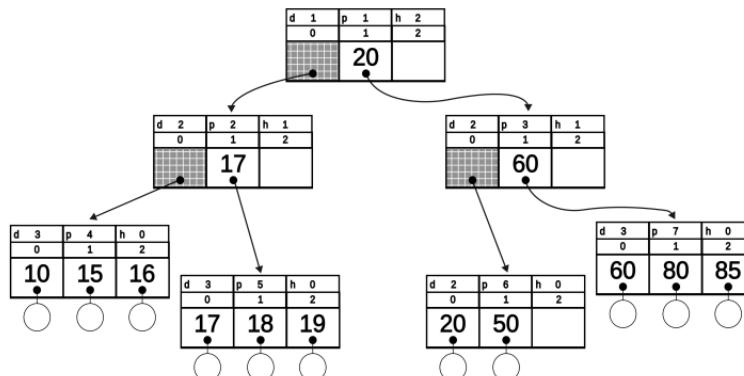
## B-Tree Operations - Search I

- The changes of this operations are mainly focused on the search part, since we have to compare to an array of keys and not only the node key.
- This operation returns the object in the B-Tree if a given key exists.

```
1 object_t *find(tree_node_t *tree, key_t query_key) {
2     tree_node_t *current_node;
3     object_t *object;
4     current_node = tree;
5
6     while( current_node->height >= 0 ) {
7         /* binary search among keys */
8         int lower, upper;
9         lower = 0;
10        upper = current_node->degree;
11
12        while( upper > lower +1 ) {
13            int med = (upper+lower)/2;
14            if( query_key < current_node->key[med] )
15                upper = med;
16            else
17                lower = med;
18        }
19        if( current_node->height > 0)
20            current_node = current_node->next[lower];
21    }
```

## B-Tree Operations - Search II

```
21
22     else {
23         if( current_node->key[lower] == query_key )
24             object = (object_t *) current_node->next[lower];
25         else
26             object = NULL;
27         return( object );
28     }
29 }
30 }
```



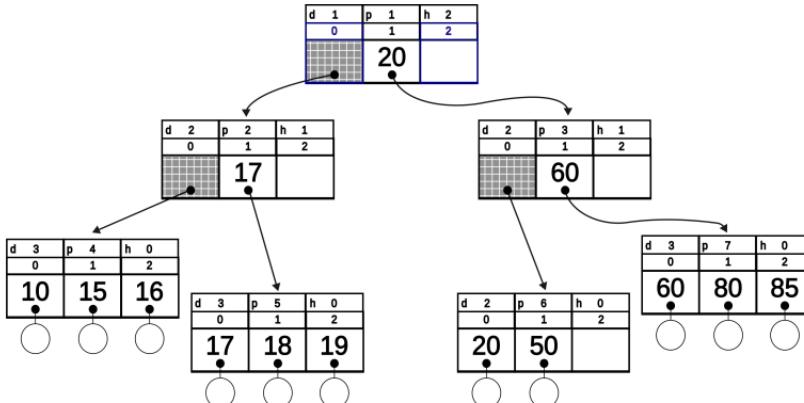
- Let's search for 19 in this  $t(2, 2)$  B-Tree.

## B-Tree Operations - Search (Example) I

```
2     tree_node_t *current_node;  
3     object_t *object;  
4     current_node = tree;  
5  
6     while( current_node->height >= 0 ) {  
7         /* binary search among keys */  
8         int lower, upper;  
9         lower = 0;  
10        upper = current_node->degree;
```

```
// Step 0  
query_key = 19;  
tree = *(node 1);
```

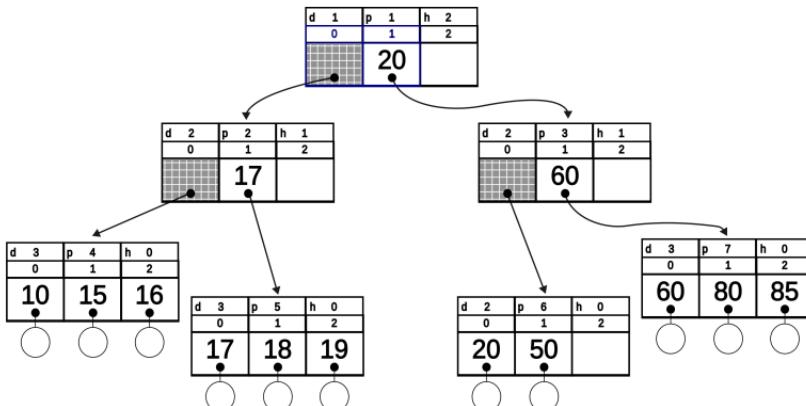
```
current_node = *(node 1);  
current_node->height = 2;  
current_node->degree = 2;  
  
lower = 0;  
upper = 2;
```



## B-Tree Operations - Search (Example) II

```
12 while( upper > lower +1 ) {  
13     int med = (upper+lower)/2;  
14     if( query_key < current_node->key[med] )  
15         upper = med;  
16     else  
17         lower = med;  
18 }  
19 if( current_node->height > 0)  
20     current_node = current_node->next[lower];  
21 }
```

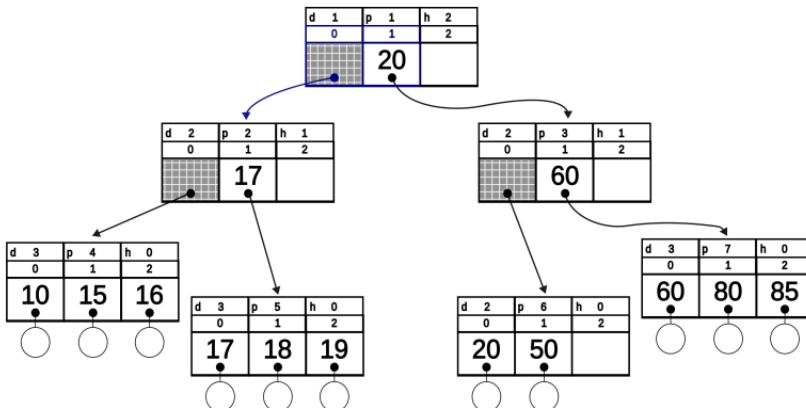
```
// Step 1  
query_key = 19;  
tree = *(node 1);  
  
current_node = *(node 1);  
current_node->height = 2;  
current_node->degree = 2;  
  
lower = 0;  
upper = 1;  
med = 1;
```



## B-Tree Operations - Search (Example) III

```
12 while( upper > lower +1 ) {  
13     int med = (upper+lower)/2;  
14     if( query_key < current_node->key[med] )  
15         upper = med;  
16     else  
17         lower = med;  
18 }  
19 if( current_node->height > 0)  
20     current_node = current_node->next[lower];  
21 }
```

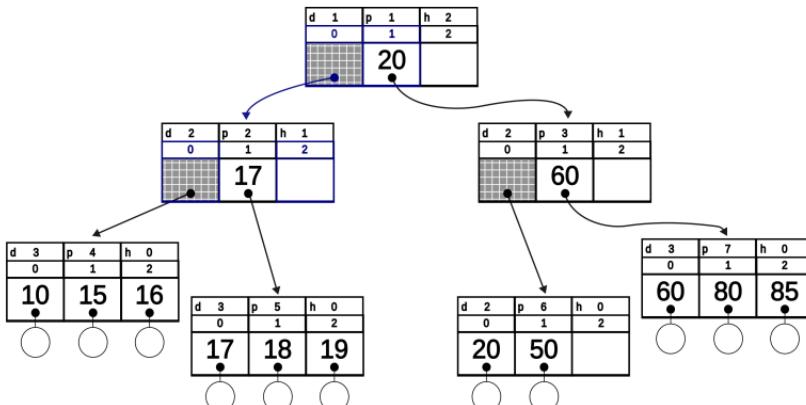
```
// Step 2  
query_key = 19;  
tree = *(node 1);  
  
current_node = *(node 1);  
current_node->height = 2;  
current_node->degree = 2;  
  
lower = 0;  
upper = 1;  
med = 1;
```



## B-Tree Operations - Search (Example) IV

```
6 while( current_node->height >= 0 ) {  
7     /* binary search among keys */  
8     int lower, upper;  
9     lower = 0;  
10    upper = current_node->degree;  
11  
12    while( upper > lower + 1 ) {
```

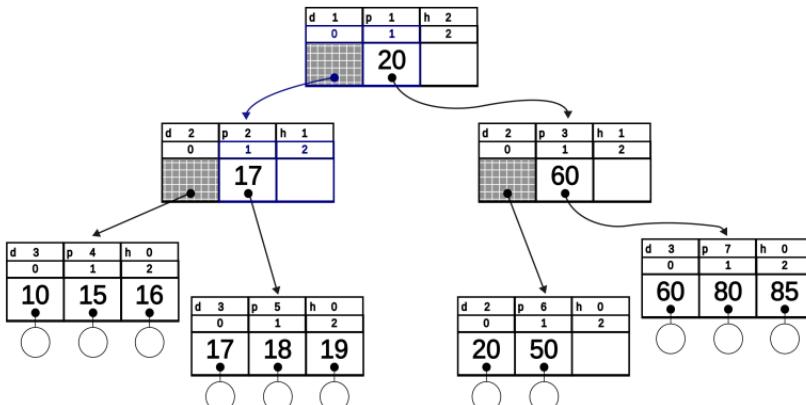
```
// Step 3  
query_key = 19;  
tree = *(node 1);  
  
current_node = *(node 2);  
current_node->height = 1;  
current_node->degree = 2;  
  
lower = 0;  
upper = 2;  
med = 1; // Not changed yet
```



## B-Tree Operations - Search (Example) V

```
12 while( upper > lower +1 ) {  
13     int med = (upper+lower)/2;  
14     if( query_key < current_node->key[med] )  
15         upper = med;  
16     else  
17         lower = med;  
18 }  
19 if( current_node->height > 0)  
20     current_node = current_node->next[lower];  
21 }
```

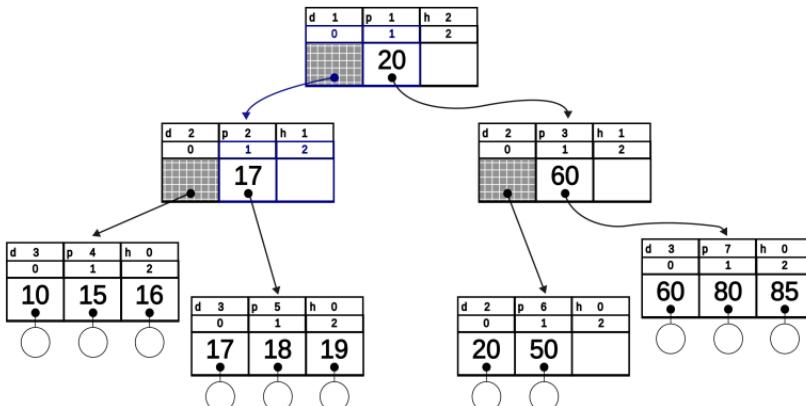
```
// Step 4  
query_key = 19;  
tree = *(node 1);  
  
current_node = *(node 2);  
current_node->height = 1;  
current_node->degree = 2;  
  
lower = 1;  
upper = 2;  
med = 1;
```



## B-Tree Operations - Search (Example) VI

```
12 while( upper > lower +1 ) {  
13     int med = (upper+lower)/2;  
14     if( query_key < current_node->key[med] )  
15         upper = med;  
16     else  
17         lower = med;  
18 }  
19 if( current_node->height > 0)  
20     current_node = current_node->next[lower];  
21 }
```

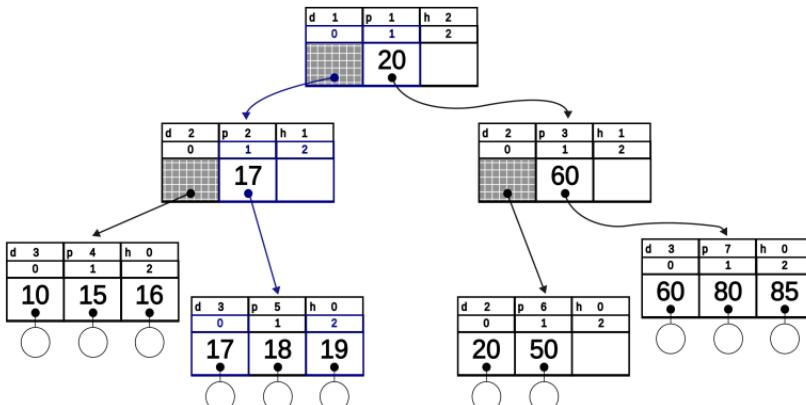
```
// Step 5  
query_key = 19;  
tree = *(node 1);  
  
current_node = *(node 2);  
current_node->height = 1;  
current_node->degree = 2;  
  
lower = 1;  
upper = 2;  
med = 1;
```



## B-Tree Operations - Search (Example) VII

```
6 while( current_node->height >= 0 ) {  
7     /* binary search among keys */  
8     int lower, upper;  
9     lower = 0;  
10    upper = current_node->degree;  
11  
12    while( upper > lower +1 ) {
```

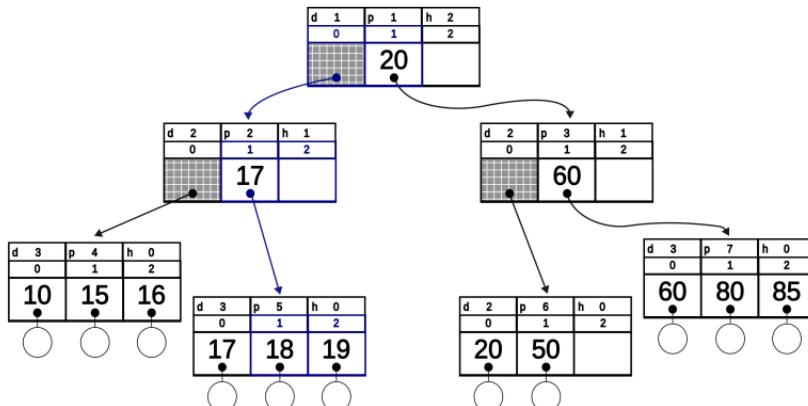
```
// Step 6  
query_key = 19;  
tree = *(node 1);  
  
current_node = *(node 6);  
current_node->height = 0;  
current_node->degree = 3;  
  
lower = 0;  
upper = 3;  
med = 1; // Not changed yet
```



## B-Tree Operations - Search (Example) VIII

```
12 while( upper > lower +1 ) {  
13     int med = (upper+lower)/2;  
14     if( query_key < current_node->key[med] )  
15         upper = med;  
16     else  
17         lower = med;  
18 }  
19 if( current_node->height > 0)  
20     current_node = current_node->next[lower];  
21 }
```

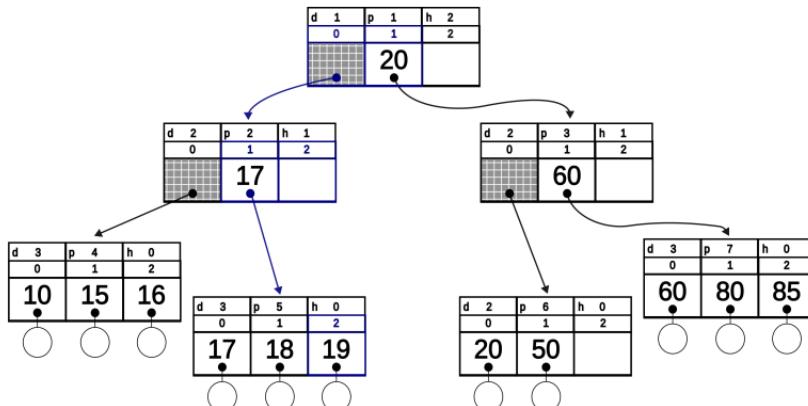
```
// Step 7  
query_key = 19;  
tree = *(node 1);  
  
current_node = *(node 6);  
current_node->height = 0;  
current_node->degree = 3;  
  
lower = 1;  
upper = 3;  
med = 1;
```



## B-Tree Operations - Search (Example) IX

```
12 while( upper > lower +1 ) {  
13     int med = (upper+lower)/2;  
14     if( query_key < current_node->key[med] )  
15         upper = med;  
16     else  
17         lower = med;  
18 }  
19 if( current_node->height > 0)  
20     current_node = current_node->next[lower];  
21 }
```

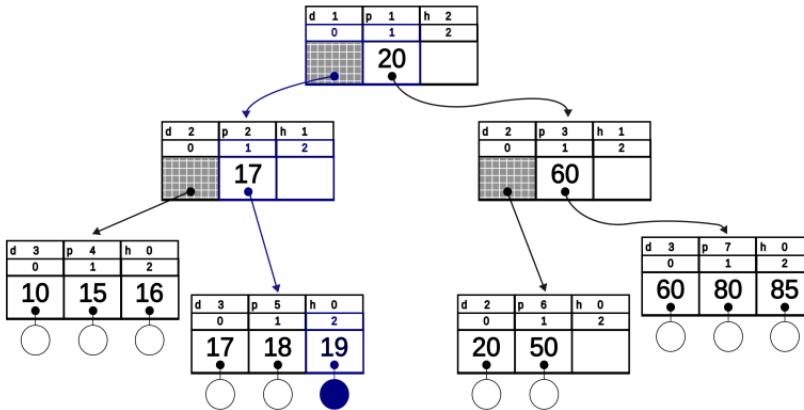
```
// Step 8  
query_key = 19;  
tree = *(node 1);  
  
current_node = *(node 6);  
current_node->height = 0;  
current_node->degree = 3;  
  
lower = 2;  
upper = 3;  
med = 2;
```



## B-Tree Operations - Search (Example) X

```
12 while( upper > lower +1 ) {  
13  
14 }  
15 if( current_node->height > 0)  
16   current_node = current_node->next[lower];  
17  
18 else {  
19   if( current_node->key[lower] == query_key )  
20     object = (object_t *) current_node->next[lower];  
21   else  
22     object = NULL;  
23   return( object );  
24 }  
25  
26 }
```

```
// Step 9  
query_key = 19;  
tree = *(node 1);  
  
object = *(19)  
  
lower = 2;  
upper = 3;  
med = 2;
```



## B-Tree Operation - Insert Value I

- ▶ The insertion algorithm in the B-Tree almost has nothing to share with any tree insertion algorithm.
- ▶ The first section of the code is the same `find` algorithm so we can see if the value to add is already stored in the B-Tree and where could it be stored, also storing in a stack the nodes that we are going to access.
- ▶ Then, if the node isn't full yet, we are just going to move everything by an index until the current elements are less than the key that we are going to insert.
- ▶ But, if the node is full, we will get a new node for the B-Tree and split in half the full node.
- ▶ Then, insert the new key into one of those of the splitted nodes.
- ▶ Then, the median key of the splitted node will be taken from the nodes and will be inserted on the upper node.
- ▶ In the new insertion of the median key and new node, will be repeated until we have a non-full node which can take another element, or if we reach the root node we will have to do a extra process.
- ▶ This extra process is that we have to split the root node, create a new node and increase the height of the B-Tree by inserting the new node with keys, pointers and such to the rest of the B-Tree above everything.
- ▶ **This is one of the only ways that the B-Tree can change its height.**

## B-Tree Operation - Insert Value II

```
1 int insert(tree_node_t *tree, key_t new_key, object_t *new_object) {
2     tree_node_t *current_node, *insert_pt;
3     key_t insert_key;
4     int finished;
5     current_node = tree;
6     if( tree->height == 0 && tree->degree == 0 ) {
7         tree->key[0] = new_key;
8         tree->next[0] = (tree_node_t *) new_object;
9         tree->degree = 1;
10        return(0); /* insert in empty tree */
11    }
12
13    create_stack();
14    while( current_node->height > 0 ) {
15        int lower, upper;
16        /* binary search among keys */
17        push( current_node );
18        lower = 0;
19        upper = current_node->degree;
20        while( upper > lower +1 ) {
21            if( new_key < current_node->key[(upper+lower)/2] )
22                upper = (upper+lower)/2;
23            else
24                lower = (upper+lower)/2;
25        }
26        current_node = current_node->next[lower];
```

## B-Tree Operation - Insert Value III

```
27     }
28     /* now current_node is leaf node in which we insert */
29
30     insert_pt = (tree_node_t *) new_object;
31     insert_key = new_key;
32     finished = 0;
33     while( !finished ){
34         int i, start;
35         if( current_node->height > 0 )
36             start = 1;
37             /* insertion in non-leaf starts at 1 */
38         else
39             start = 0;
40             /* insertion in non-leaf starts at 0 */
41         /* node still has room */
42         if( current_node->degree < (2 * ALPHA) - 1 ) {
43             /* move everything up to create the insertion gap */
44             i = current_node->degree;
45             while( (i > start) && (current_node->key[i-1] > insert_key)) {
46                 current_node->key[i] =
47                     current_node->key[i-1];
48                 current_node->next[i] =
49                     current_node->next[i-1];
50                 i -= 1;
51             }
52 }
```

## B-Tree Operation - Insert Value IV

```
53     current_node->key[i] = insert_key;
54     current_node->next[i] = insert_pt;
55     current_node->degree +=1;
56     finished = 1;
57 }
58
59 /* end insert in non-full node */
60 else {
61     /* node is full, have to split the node*/
62     tree_node_t *new_node;
63     int j, insert_done = 0;
64     new_node = get_node();
65     i = ((2 * ALPHA) - 1)-1;
66     j = (((2 * ALPHA) - 1)-1)/2;
67     while( j >= 0 ) {
68         /* copy upper half to new node */
69         if( insert_done || insert_key < current_node->key[i] ) {
70             new_node->next[j] =
71                 current_node->next[i];
72             new_node->key[j--] =
73                 current_node->key[i--];
74         } else {
75             new_node->next[j] = insert_pt;
76             new_node->key[j--] = insert_key;
77             insert_done = 1;
78     }
```

## B-Tree Operation - Insert Value V

```
79     }
80     /* upper half done, insert in lower half, if necessary*/
81     while( !insert_done) {
82         if( insert_key < current_node->key[i] && i >= start ) {
83             current_node->next[i+1] =
84                 current_node->next[i];
85             current_node->key[i+1] =
86                 current_node->key[i];
87             i -=1;
88         } else {
89             current_node->next[i+1] =
90                 insert_pt;
91             current_node->key[i+1] =
92                 insert_key;
93             insert_done = 1;
94         }
95     }
```

## B-Tree Operation - Insert Value VI

```
96     /*finished insertion */
97
98     current_node->degree = ((2 * ALPHA) - 1)+1 - (((2 * ALPHA) - 1)+1)/2;
99     new_node->degree = (((2 * ALPHA) - 1)+1)/2;
100    new_node->height = current_node->height;
101   /* split nodes complete, now insert the new node above */
102   insert_pt = new_node;
103   insert_key = new_node->key[0];
104   if( ! stack_empty() ) {
105       /* not at root; move one level up*/
106       current_node = pop();
107   }
108   else {
109       /* splitting root: needs copy to keep root address*/
110       new_node = get_node();
111       for( i=0; i < current_node->degree; i++ ) {
112           new_node->next[i] =
113               current_node->next[i];
114           new_node->key[i] =
115               current_node->key[i];
116       }
117       new_node->height =
118           current_node->height;
119       new_node->degree =
120           current_node->degree;
121       current_node->height += 1;
```

## B-Tree Operation - Insert Value VII

```
122     current_node->degree = 2;
123     current_node->next[0] = new_node;
124     current_node->next[1] = insert_pt;
125     current_node->key[1] = insert_key;
126     finished =1;
127 } /* end splitting root */
128 } /* end node splitting */
129 } /* end of rebalancing */
130 remove_stack();
131 return( 0 );
132 }
```

d	0	p	1	h	0
0		1		2	

- Now, lets create a new empty tree and insert a lot of elements in a  $t(2,0)$  B-Tree.

## B-Tree Operations - Search (Example) I

```
1 int insert(tree_node_t *tree, key_t new_key, object_t
2     *new_object) {
3     tree_node_t *current_node, *insert_pt;
4     key_t insert_key;
5     int finished;
6     current_node = tree;
7     if( tree->height == 0 && tree->degree == 0 ) {
8         tree->key[0] = new_key;
9         tree->next[0] = (tree_node_t *) new_object;
10        tree->degree = 1;
11        return(0); /* insert in empty tree */
12    }
```

```
// Step 0
tree = (*p 1);
new_key = 10; new_object = (*10);

current_node = tree;
current_node->height = 0;
current_node->degree = 0;

insert_pt = NULL;
insert_key = NULL;
finished = NULL;
```

d	0	p	1	h	0
0		1		2	

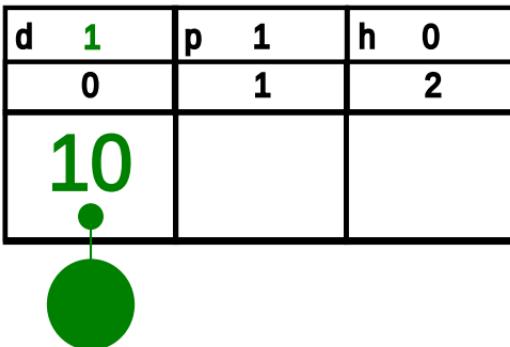
## B-Tree Operations - Search (Example) II

```
1 int insert(tree_node_t *tree, key_t new_key, object_t
2     *new_object) {
3     tree_node_t *current_node, *insert_pt;
4     key_t insert_key;
5     int finished;
6     current_node = tree;
7     if( tree->height == 0 && tree->degree == 0 ) {
8         tree->key[0] = new_key;
9         tree->next[0] = (tree_node_t *) new_object;
10        tree->degree = 1;
11        return(0); /* insert in empty tree */
12    }
```

```
// Step 1
tree = (*p 1);
new_key = 10; new_object = (*10);

current_node = tree;
current_node->height = 0;
current_node->degree = 1;

insert_pt = NULL;
insert_key = NULL;
finished = NULL;
```



## B-Tree Operations - Search (Example) III

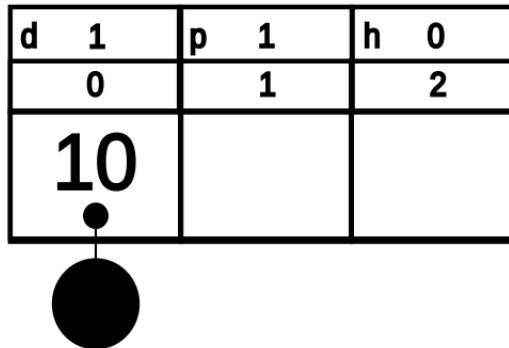
```
13 create_stack();
14 while( current_node->height > 0 ) {
15
16     insert_pt = (tree_node_t *) new_object;
17     insert_key = new_key;
18     finished = 0;
19
20     while( !finished ){
21         int i, start;
22         if( current_node->height > 0 )
23             start = 1;
24             /* insertion in non-leaf starts at 1 */
25         else
26             start = 0;
27             /* insertion in non-leaf starts at 0 */
28
29
30 }
```

```
// Step 2 - 1
tree = (*p 1);
new_key = 20; new_object = (*20);

current_node = tree;
current_node->height = 0; current_node->degree = 1;

insert_pt = (*20); insert_key = 20;
finished = 0;

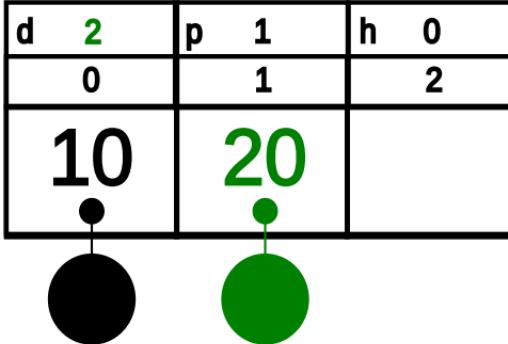
stack = [];
start = 0; i = NULL;
```



## B-Tree Operations - Search (Example) IV

```
41 /* node still has room */  
42 if( current_node->degree < (2 * ALPHA) - 1 ) {  
    /* move everything up to create the insertion gap  
     * /  
43     i = current_node->degree;  
44     while( (i > start) && (current_node->key[i-1] >  
45         insert_key)) {  
  
46         current_node->key[i] = insert_key;  
47         current_node->next[i] = insert_pt;  
48         current_node->degree +=1;  
49         finished = 1;  
50     }  
51 }
```

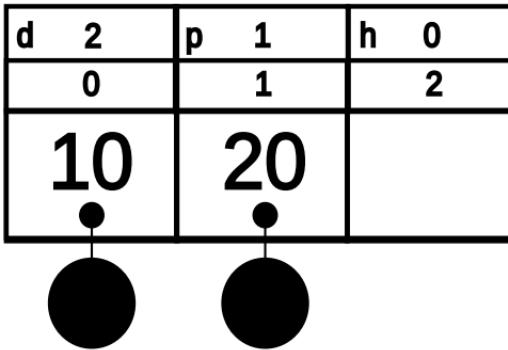
```
// Step 2 - 2  
tree = (*p 1);  
new_key = 20; new_object = (*20);  
  
current_node = tree;  
current_node->height = 0; current_node->degree = 2;  
  
insert_pt = (*20); insert_key = 20;  
finished = 1;  
  
stack = [];  
start = 0; i = 1;
```



## B-Tree Operations - Search (Example) V

```
41 /* node still has room */  
42 if( current_node->degree < (2 * ALPHA) - 1 ) {  
43     /* move everything up to create the insertion gap  
44     ↵     */  
45     i = current_node->degree;  
46     while( (i > start) && (current_node->key[i-1] >  
47         ↵     insert_key)) {  
48         current_node->key[i] =  
49             ↵     current_node->key[i-1];  
50         current_node->next[i] =  
51             ↵     current_node->next[i-1];  
52         i -= 1;  
53     }  
54 }
```

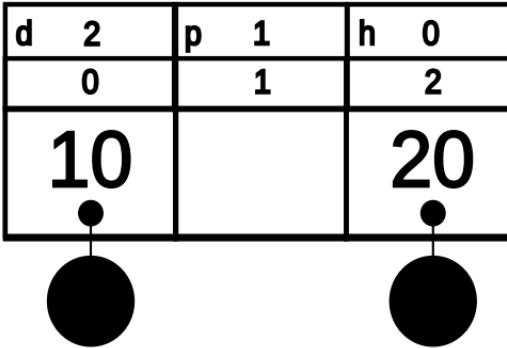
```
// Step 3 - 1  
tree = (*p 1);  
new_key = 15; new_object = (*15);  
  
current_node = tree;  
current_node->height = 0; current_node->degree = 2;  
  
insert_pt = (*15); insert_key = 15;  
finished = 0;  
  
stack = [];  
start = 0; i = 2;
```



## B-Tree Operations - Search (Example) VI

```
41 /* node still has room */  
42 if( current_node->degree < (2 * ALPHA) - 1 ) {  
43     /* move everything up to create the insertion gap  
44     ↵     */  
45     i = current_node->degree;  
46     while( (i > start) && (current_node->key[i-1] >  
47         insert_key)) {  
48         current_node->key[i] =  
49             current_node->key[i-1];  
50         current_node->next[i] =  
51             current_node->next[i-1];  
52         i -= 1;  
53     }  
54 }
```

```
// Step 3 - 2  
tree = (*p 1);  
new_key = 15; new_object = (*15);  
  
current_node = tree;  
current_node->height = 0; current_node->degree = 2;  
  
insert_pt = (*15); insert_key = 15;  
finished = 0;  
  
stack = [];  
start = 0; i = 1;
```



## B-Tree Operations - Search (Example) VII

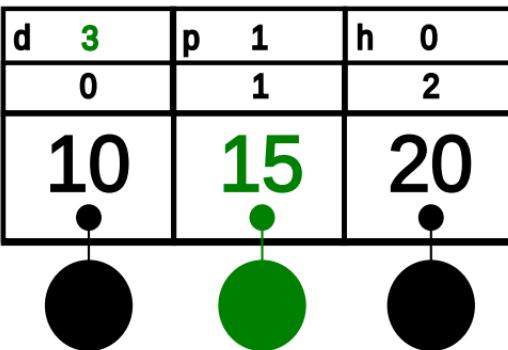
```
53     current_node->key[i] = insert_key;
54     current_node->next[i] = insert_pt;
55     current_node->degree +=1;
56     finished = 1;
57 }
```

```
// Step 3 - 3
tree = (*p 1);
new_key = 15; new_object = (*15);

current_node = tree;
current_node->height = 0; current_node->degree = 3;

insert_pt = (*15); insert_key = 15;
finished = 1;

stack = [];
start = 0; i = 1;
```



## B-Tree Operations - Search (Example) VIII

42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66

```
if( current_node->degree < (2 * ALPHA) - 1) {  
  
/* end insert in non-full node */  
else {  
    /* node is full, have to split the node*/  
    tree_node_t *new_node;  
    int j, insert_done = 0;  
    new_node = get_node();  
    i = ((2 * ALPHA) - 1)-1;  
    j = (((2 * ALPHA) - 1)-1)/2;
```

```
// Step 4 - 1  
tree = (*p 1);  
new_key = 17; new_object = (*17);  
  
current_node = tree;  
current_node->height = 0; current_node->degree = 3;  
new_node = NULL;  
  
insert_pt = (*17); insert_key = 17;  
finished = 0; insert_done = 0;  
  
stack = [];  
start = 0; i = 2; j = 1;
```

d	3	p	1	h	0
0		1		2	
10		15		20	

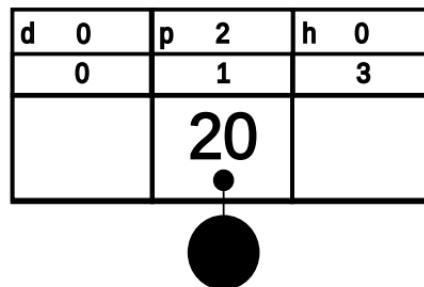
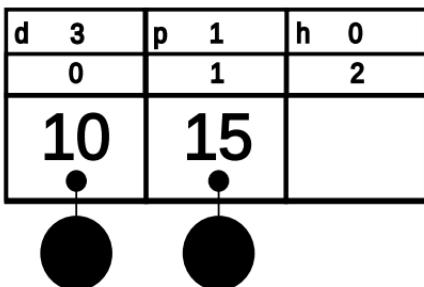
Diagram illustrating a B-tree node structure. The node has degree 3, represented by the value 3 in the d column. It contains three keys: 10, 15, and 20, represented by the values 1, 2, and 3 in the p column respectively. The height of the node is 0, represented by the value 0 in the h column. Below the node, three solid black circles represent child pointers, one for each key.

d	0	p	2	h	0
0		1		3	

## B-Tree Operations - Search (Example) IX

```
67 while( j >= 0 ) {  
68     /* copy upper half to new node */  
69     if( insert_done || insert_key <  
70         current_node->key[i] ) {  
71         new_node->next[j] =  
72             current_node->next[i];  
73         new_node->key[j--] =  
74             current_node->key[i--];  
75     } else {  
76         new_node->next[j] = insert_pt;  
77         new_node->key[j--] = insert_key;  
78         insert_done = 1;  
79     }  
}
```

```
// Step 4 - 2  
tree = (*p 1);  
new_key = 17; new_object = (*17);  
  
current_node = tree; ->height = 0; ->degree = 3;  
new_node = (*p 2); ->height = 0; ->degree = 0;  
  
insert_pt = (*17); insert_key = 17;  
finished = 0; insert_done = 0;  
  
stack = [];  
start = 0; i = 1; j = 0;
```



## B-Tree Operations - Search (Example) X

```
67 while( j >= 0 ) {  
68     /* copy upper half to new node */  
69     if( insert_done || insert_key <  
70         current_node->key[i] ) {  
71         new_node->next[j] =  
72             current_node->next[i];  
73         new_node->key[j--] =  
74             current_node->key[i--];  
75     } else {  
76         new_node->next[j] = insert_pt;  
77         new_node->key[j--] = insert_key;  
78         insert_done = 1;  
79     }  
}
```

```
// Step 4 - 3  
tree = (*p 1);  
new_key = 17; new_object = (*17);  
  
current_node = tree; ->height = 0; ->degree = 3;  
new_node = (*p 2); ->height = 0; ->degree = 0;  
  
insert_pt = (*17); insert_key = 17;  
finished = 0; insert_done = 1;  
  
stack = [];  
start = 0; i = 0; j = -1;
```

d	3	p	1	h	0
0		1		2	
10		15			



d	0	p	2	h	0
0		1		3	
17		20			



## B-Tree Operations - Search (Example) XI

```
96 /*finished insertion */  
97  
98 current_node->degree = ((2 * ALPHA) - 1)+1 - (((((2 *  
99    ↵ ALPHA) - 1)+1)/2);  
00 new_node->degree = (((2 * ALPHA) - 1)+1)/2;  
01 new_node->height = current_node->height;  
02 /* split nodes complete, now insert the new node  
03    ↵ above */  
04 insert_pt = new_node;  
05 insert_key = new_node->key[0];  
06 if( ! stack_empty() ) {  
07 }
```

```
// Step 4 - 5  
tree = (*p 1);  
new_key = 17; new_object = (*17);  
  
current_node = tree; ->height = 0; ->degree = 2;  
new_node = (*p 2); ->height = 0; ->degree = 2;  
insert_pt = (*p 2);  
  
insert_key = 17;  
finished = 0; insert_done = 0;  
  
stack = [];  
start = 0; i = 0; j = -1;
```

d	2	p	1	h	0
0		1		2	
10		15			



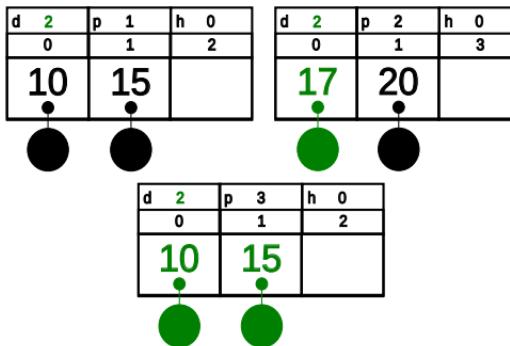
d	2	p	2	h	0
0		1		3	
17		20			



## B-Tree Operations - Search (Example) XII

```
08  
09 else {  
10     /* splitting root: needs copy to keep root  
11     ↵ address*/  
12     new_node = get_node();  
13     for( i=0; i < current_node->degree; i++ ) {  
14         new_node->next[i] =  
15             current_node->next[i];  
16         new_node->key[i] =  
17             current_node->key[i];  
18     }  
19     new_node->height =  
20         current_node->height;  
21     new_node->degree =  
22         current_node->degree;
```

```
// Step 4 - 5  
tree = (*p 1);  
new_key = 17; new_object = (*17);  
  
current_node = tree; ->height = 0; ->degree = 2;  
new_node = (*p 2); ->height = 0; ->degree = 2;  
insert_pt = (*p 2);  
  
insert_key = 17;  
finished = 0; insert_done = 0;  
  
stack = [];
```



## B-Tree Operations - Search (Example) XIII

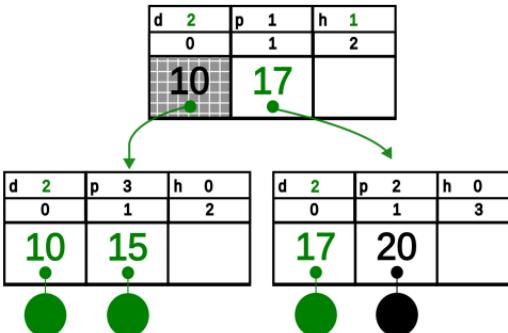
```
21     current_node->height += 1;
22     current_node->degree = 2;
23     current_node->next[0] = new_node;
24     current_node->next[1] = insert_pt;
25     current_node->key[1] = insert_key;
26     finished = 1;
27 } /* end splitting root */
28 } /* end node splitting */
29 } /* end of rebalancing */
30 remove_stack();
31 return( 0 );
32 }
```

```
// Step 4 - 6
tree = (*p 1);
new_key = 17; new_object = (*17);

current_node = tree; ->height = 1; ->degree = 2;
new_node = (*p 3); ->height = 0; ->degree = 2;
insert_pt = (*p 2); ->height = 0; ->degree = 2;

insert_key = 17;
finished = 1; insert_done = 0;

stack = [];
start = 0; i = 0; j = -1;
```



## B-Tree Operations - Search (Example) XIV

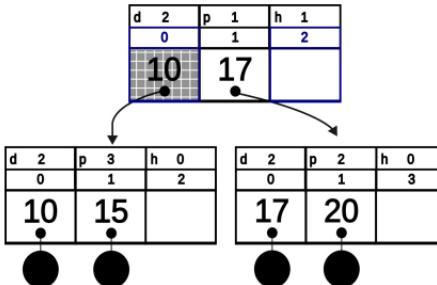
```
13 create_stack();
14 while( current_node->height > 0 ) {
15     int lower, upper;
16     /* binary search among keys */
17     push( current_node );
18     lower = 0;
19     upper = current_node->degree;
20     while( upper > lower +1 ) {
21         if( new_key <
22             ↳ current_node->key[(upper+lower)/2] )
23             upper = (upper+lower)/2;
24         else
25             lower = (upper+lower)/2;
26     }
27     current_node = current_node->next[lower];
}
```

```
// Step 5 - 1
tree = (*p 1);
new_key = 25; new_object = (*25);

current_node = tree; ->height = 1; ->degree = 2;

insert_pt = NULL; insert_key = NULL;
finished = NULL;

stack = [(*p 1)];
lower = 0; upper = 2;
```



## B-Tree Operations - Search (Example) XV

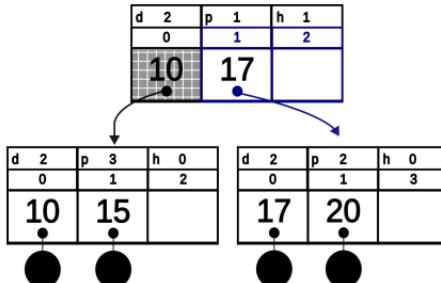
```
13 create_stack();
14 while( current_node->height > 0 ) {
15     int lower, upper;
16     /* binary search among keys */
17     push( current_node );
18     lower = 0;
19     upper = current_node->degree;
20     while( upper > lower +1 ) {
21         if( new_key <
22             ↳ current_node->key[(upper+lower)/2] )
23             upper = (upper+lower)/2;
24         else
25             lower = (upper+lower)/2;
26     }
27     current_node = current_node->next[lower];
}
```

```
// Step 5 - 2
tree = (*p 1);
new_key = 25; new_object = (*25);

current_node = (*p 2); ->height = 0; ->degree = 2;

insert_pt = NULL; insert_key = NULL;
finished = NULL;

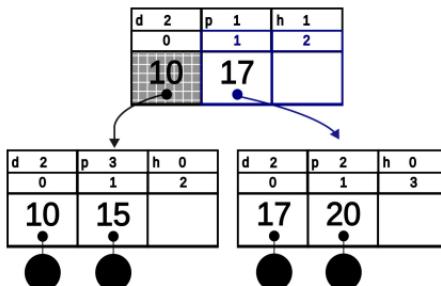
stack = [(*p 1)];
lowwer = 1; upper = 2;
```



## B-Tree Operations - Search (Example) XVI

```
33 while( !finished ){  
34     int i, start;  
35     if( current_node->height > 0 )  
36         start = 1;  
37         /* insertion in non-leaf starts at 1 */  
38     else  
39         start = 0;  
40         /* insertion in non-leaf starts at 0 */  
41     /* node still has room */  
42     if( current_node->degree < (2 * ALPHA) - 1) {
```

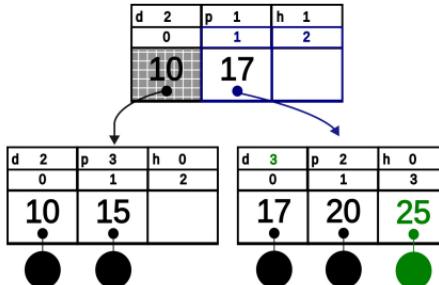
```
// Step 5 - 3  
tree = (*p 1);  
new_key = 25; new_object = (*25);  
  
current_node = (*p 2); ->height = 0; ->degree = 2;  
  
insert_pt = (* 25); insert_key = 25;  
finished = 0;  
  
stack = [(*p 1)];  
start = 0; i = NULL;
```



## B-Tree Operations - Search (Example) XVII

```
42 if( current_node->degree < (2 * ALPHA) - 1) {  
43     /* move everything up to create the insertion gap  
44     ↵ */  
45     i = current_node->degree;  
46     while( (i > start) && (current_node->key[i-1] >  
47         insert_key)) {  
48         }  
49         current_node->key[i] = insert_key;  
50         current_node->next[i] = insert_pt;  
51         current_node->degree +=1;  
52         finished = 1;  
53     }  
54 }
```

```
// Step 5 - 4  
tree = (*p 1);  
new_key = 25; new_object = (*25);  
  
current_node = (*p 2); ->height = 0; ->degree = 3;  
  
insert_pt = (* 25); insert_key = 25;  
finished = 1;  
  
stack = [(*p 1)];  
start = 0; i = 2;
```



## B-Tree Operations - Search (Example) XVIII

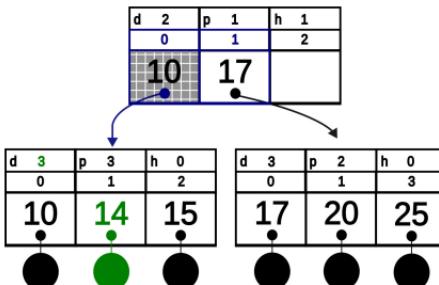
```
53     current_node->key[i] = insert_key;
54     current_node->next[i] = insert_pt;
55     current_node->degree +=1;
56     finished = 1;
57 }
```

```
// Step 6 - 1 - ... - 5
tree = (*p 1);
new_key = 14; new_object = (*14);

current_node = (*p 3); ->height = 0; ->degree = 3;

insert_pt = (* 14); insert_key = 14;
finished = 1;

stack = [(*p 1)];
start = 0; i = 1;
```



## B-Tree Operations - Search (Example) XIX

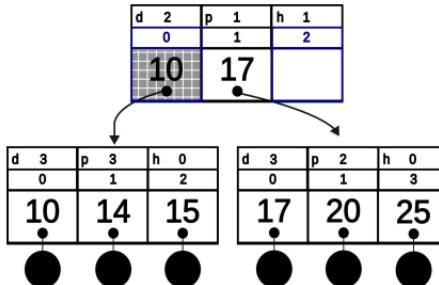
```
13 create_stack();
14 while( current_node->height > 0 ) {
15     int lower, upper;
16     /* binary search among keys */
17     push( current_node );
18     lower = 0;
19     upper = current_node->degree;
20     while( upper > lower +1 ) {
21         if( new_key <
22             ↳ current_node->key[(upper+lower)/2] )
23             upper = (upper+lower)/2;
24         else
25             lower = (upper+lower)/2;
26     }
27     current_node = current_node->next[lower];
}
```

```
// Step 7 - 1
tree = (*p 1);
new_key = 30; new_object = (*30);

current_node = tree; ->height = 1; ->degree = 2;

insert_pt = NULL; insert_key = NULL;
finished = NULL;

stack = [(*p 1)];
lowwer = 0; upper = 2;
```



## B-Tree Operations - Search (Example) XX

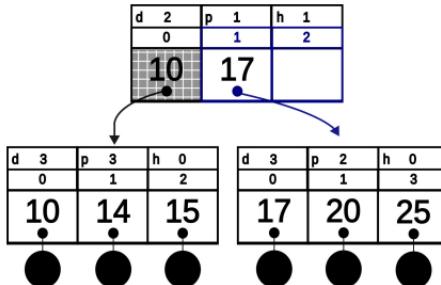
```
13 create_stack();
14 while( current_node->height > 0 ) {
15     int lower, upper;
16     /* binary search among keys */
17     push( current_node );
18     lower = 0;
19     upper = current_node->degree;
20     while( upper > lower +1 ) {
21         if( new_key <
22             ↳ current_node->key[(upper+lower)/2] )
23             upper = (upper+lower)/2;
24         else
25             lower = (upper+lower)/2;
26     }
27     current_node = current_node->next[lower];
}
```

```
// Step 7 - 2
tree = (*p 1);
new_key = 30; new_object = (*30);

current_node = (*p 2); ->height = 0; ->degree = 3;

insert_pt = NULL; insert_key = NULL;
finished = NULL;

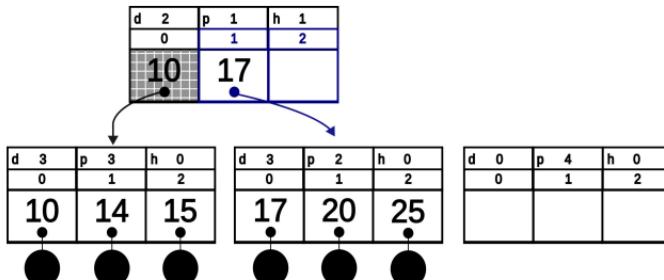
stack = [(*p 1)];
lower = 1; upper = 2;
```



## B-Tree Operations - Search (Example) XXI

```
42 if( current_node->degree < (2 * ALPHA) - 1) {  
  
60 else {  
61     /* node is full, have to split the node*/  
62     tree_node_t *new_node;  
63     int j, insert_done = 0;  
64     new_node = get_node();  
65     i = ((2 * ALPHA) - 1)-1;  
66     j = (((2 * ALPHA) - 1)-1)/2;
```

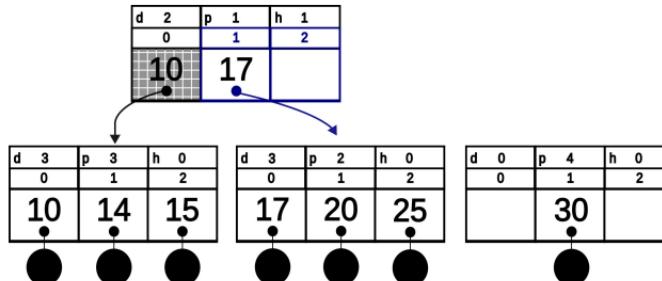
```
// Step 7 - 3  
tree = (*p 1);  
new_key = 30; new_object = (*30);  
  
current_node = (*p 2); ->height = 0; ->degree = 3;  
new_node = (*p 4); -> height = 0; ->degree 0;  
  
insert_pt = (*30); insert_key = 30;  
finished = 0; insert_done = 0;  
  
stack = [(*p 1)];  
start = 0; i = 2; j = 1;
```



## B-Tree Operations - Search (Example) XXII

```
69  
70 if( insert_done || insert_key <  
71     ↵ current_node->key[i] ) {  
72     new_node->next[j] =  
73         ↵ current_node->next[i];  
74     new_node->key[j--] =  
75         ↵ current_node->key[i--];  
76 } else {  
77     new_node->next[j] = insert_pt;  
78     new_node->key[j--] = insert_key;  
79     insert_done = 1;  
 }
```

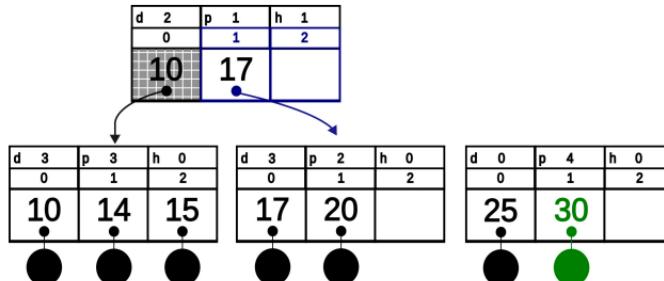
```
// Step 7 - 4  
tree = (*p 1);  
new_key = 30; new_object = (*30);  
  
current_node = (*p 2); ->height = 0; ->degree = 3;  
new_node = (*p 4); -> height = 0; ->degree 0;  
  
insert_pt = (*30); insert_key = 30;  
finished = 0; insert_done = 1;  
  
stack = [(*p 1)];  
start = 0; i = 2; j = 0;
```



## B-Tree Operations - Search (Example) XXIII

```
69  
70 if( insert_done || insert_key <  
71     ↵ current_node->key[i] ) {  
72     new_node->next[j] =  
73         ↵ current_node->next[i];  
74     new_node->key[j--] =  
75         ↵ current_node->key[i--];  
76 } else {  
77     new_node->next[j] = insert_pt;  
78     new_node->key[j--] = insert_key;  
79     insert_done = 1;  
 }
```

```
// Step 7 - 5  
tree = (*p 1);  
new_key = 30; new_object = (*30);  
  
current_node = (*p 2); ->height = 0; ->degree = 3;  
new_node = (*p 4); -> height = 0; ->degree 0;  
  
insert_pt = (*30); insert_key = 30;  
finished = 0; insert_done = 1;  
  
stack = [(*p 1)];  
start = 0; i = 1; j = -1;
```



## B-Tree Operations - Search (Example) XXIV

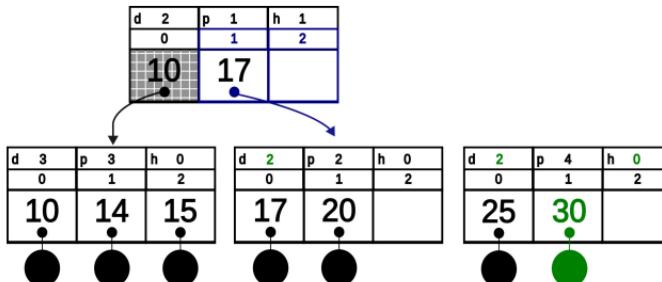
```
98 current_node->degree = (((2 * ALPHA) - 1)+1 - (((2 *
99    ↵ ALPHA) - 1)+1)/2);
00 new_node->degree = (((2 * ALPHA) - 1)+1)/2;
01 new_node->height = current_node->height;
02 /* split nodes complete, now insert the new node
03    ↵ above */
04 insert_pt = new_node;
05 insert_key = new_node->key[0];
06 if( ! stack_empty() ) {
07     /* not at root; move one level up*/
08     current_node = pop();
09 }
```

```
// Step 7 - 6
tree = (*p 1);
new_key = 30; new_object = (*30);

//current_node = (*p 2); ->height = 0; ->degree = 2;
current_node = (*p 1); ->height = 1; ->degree = 2;
new_node = (*p 4); -> height = 0; ->degree 2;

insert_pt = (*p 4); insert_key = 30;
finished = 0; insert_done = 1;

stack = [];
```



## B-Tree Operations - Search (Example) XXV

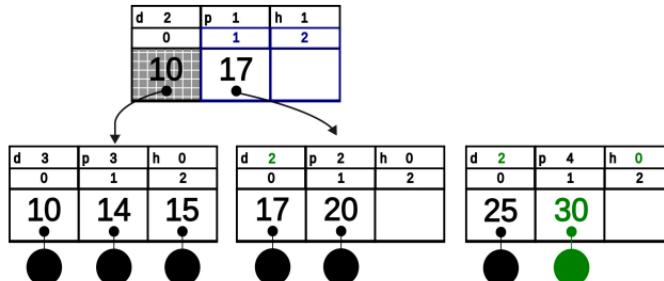
```
33 while( !finished ){
34     int i, start;
35     if( current_node->height > 0 )
36         start = 1;
37         /* insertion in non-leaf starts at 1 */
38     else
39         start = 0;
40         /* insertion in non-leaf starts at 0 */
```

```
// Step 7 - 7
tree = (*p 1);
new_key = 30; new_object = (*30);

current_node = (*p 1); ->height = 1; ->degree = 2;
new_node = (*p 4); -> height = 0; ->degree 2;

insert_pt = (*p 4); insert_key = 30;
finished = 0; insert_done = 1;

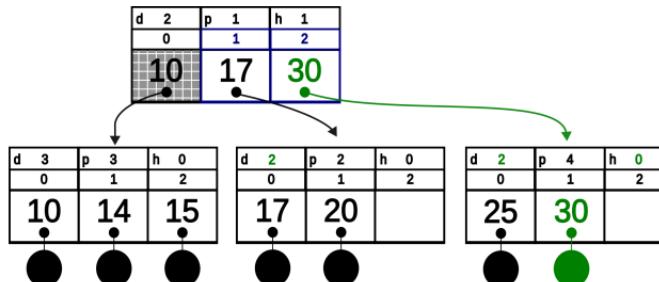
stack = [];
start = 1; i = NULL;
```



## B-Tree Operations - Search (Example) XXVI

```
42 if( current_node->degree < (2 * ALPHA) - 1) {  
43     /* move everything up to create the insertion gap  
44     ↵ */  
45     i = current_node->degree;  
46     while( (i > start) && (current_node->key[i-1] >  
47         insert_key)) {  
48         ...  
49     }  
50     current_node->key[i] = insert_key;  
51     current_node->next[i] = insert_pt;  
52     current_node->degree +=1;  
53     finished = 1;  
54 }  
55  
56 }
```

```
// Step 7 - 8  
tree = (*p 1);  
new_key = 30; new_object = (*30);  
  
current_node = (*p 1); ->height = 1; ->degree = 3;  
new_node = (*p 4); -> height = 0; ->degree 2;  
  
insert_pt = (*p 4); insert_key = 30;  
finished = 1; insert_done = 1;  
  
stack = [];  
start = 1; i = 2;
```



## B-Tree Operations - Search (Example) XXVII

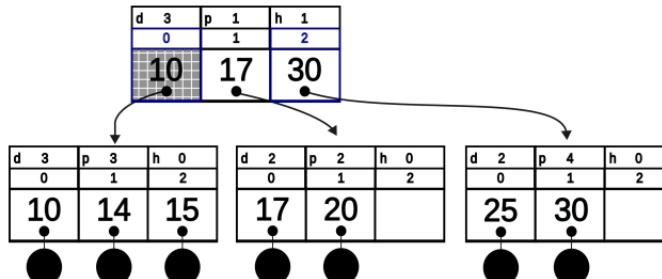
```
13 create_stack();
14 while( current_node->height > 0 ) {
15     int lower, upper;
16     /* binary search among keys */
17     push( current_node );
18     lower = 0;
19     upper = current_node->degree;
20     while( upper > lower +1 ) {
21         if( new_key <
22             ↳ current_node->key[(upper+lower)/2] )
23             upper = (upper+lower)/2;
24         else
25             lower = (upper+lower)/2;
26     }
27     current_node = current_node->next[lower];
}
```

```
// Step 8 - 1
tree = (*p 1);
new_key = 5; new_object = (*5);

current_node = tree; ->height = 1; ->degree = 3;

insert_pt = NULL; insert_key = NULL;
finished = NULL;

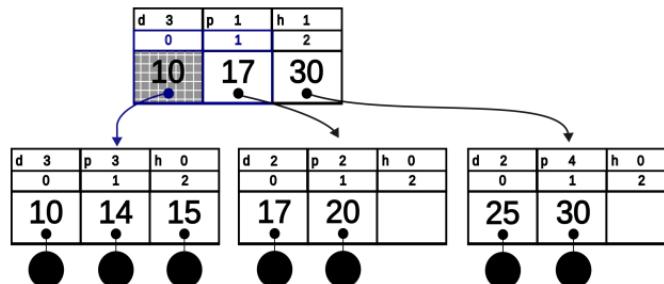
stack = []
lower = 0; upper = 3;
```



## B-Tree Operations - Search (Example) XXVIII

```
14 while( current_node->height > 0 ) {  
15     int lower, upper;  
16     /* binary search among keys */  
17     push( current_node );  
18     lower = 0;  
19     upper = current_node->degree;  
20     while( upper > lower +1 ) {  
21         if( new_key <  
22             ↳ current_node->key[(upper+lower)/2] )  
23             upper = (upper+lower)/2;  
24         else  
25             lower = (upper+lower)/2;  
26     }  
27     current_node = current_node->next[lower];  
}
```

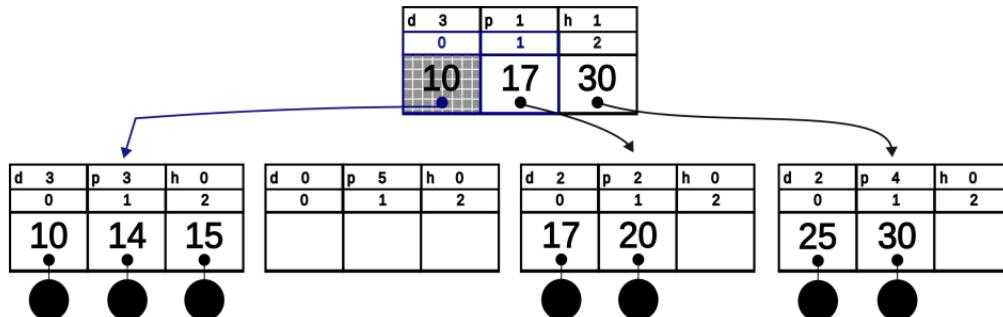
```
// Step 8 - 2  
tree = (*p 1);  
new_key = 5; new_object = (*5);  
  
current_node = (*p 3); ->height = 0; ->degree = 3;  
  
insert_pt = NULL; insert_key = NULL;  
finished = NULL;  
  
stack = [(*p 1)]  
lower = 0; upper = 1;
```



## B-Tree Operations - Search (Example) XXIX

```
33 while( !finished ){  
42 if( current_node->degree < (2 * ALPHA) - 1) {  
  
60 else {  
61     /* node is full, have to split the node*/  
62     tree_node_t *new_node;  
63     int j, insert_done = 0;  
64     new_node = get_node();  
65     i = ((2 * ALPHA) - 1)-1;  
66     j = (((2 * ALPHA) - 1)-1)/2;
```

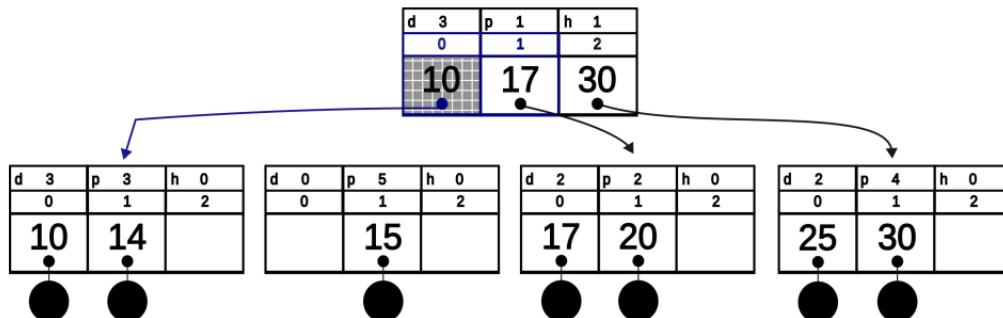
```
// Step 8 - 3  
tree = (*p 1);  
new_key = 5; new_object = (*5);  
  
current_node = (*p 3); ->height = 0; ->degree = 3;  
new_node = (*p 5); ->height = 0; -> degree = 0;  
  
insert_pt = (* 9); insert_key = 9;  
finished = 0; insert_done = 0;  
  
stack = [(*p 1)]  
start = 0; i = 2; j = 1;
```



# B-Tree Operations - Search (Example) XXX

```
67 while( j >= 0 ) {  
68     /* copy upper half to new node */  
69     if( insert_done || insert_key <  
70         current_node->key[i] ) {  
71         new_node->next[j] =  
72             current_node->next[i];  
73         new_node->key[j--] =  
74             current_node->key[i--];  
75     } else {  
76         new_node->next[j] = insert_pt;  
77         new_node->key[j--] = insert_key;  
78         insert_done = 1;  
79     }  
}
```

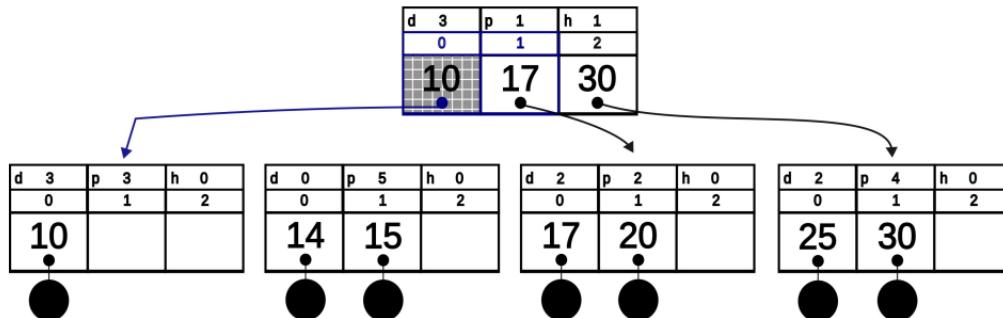
```
// Step 8 - 4  
tree = (*p 1);  
new_key = 5; new_object = (*5);  
  
current_node = (*p 3); ->height = 0; ->degree = 3;  
new_node = (*p 5); ->height = 0; -> degree = 0;  
  
insert_pt = (* 9); insert_key = 9;  
finished = 0; insert_done = 0;  
  
stack = [(*p 1)]  
start = 0; i = 1; j = 0;
```



## B-Tree Operations - Search (Example) XXXI

```
67 while( j >= 0 ) {  
68     /* copy upper half to new node */  
69     if( insert_done || insert_key <  
70         current_node->key[i] ) {  
71         new_node->next[j] =  
72             current_node->next[i];  
73         new_node->key[j--] =  
74             current_node->key[i--];  
75     } else {  
76         new_node->next[j] = insert_pt;  
77         new_node->key[j--] = insert_key;  
78         insert_done = 1;  
79     }  
}
```

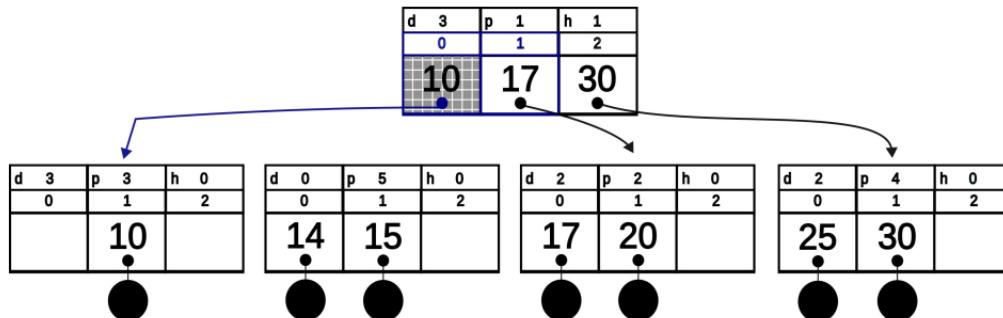
```
// Step 8 - 5  
tree = (*p 1);  
new_key = 5; new_object = (*5);  
  
current_node = (*p 3); ->height = 0; ->degree = 3;  
new_node = (*p 5); ->height = 0; -> degree = 0;  
  
insert_pt = (* 9); insert_key = 9;  
finished = 0; insert_done = 0;  
  
stack = [(*p 1)]  
start = 0; i = 0; j = -1;
```



## B-Tree Operations - Search (Example) XXXII

```
81 while( !insert_done) {  
82     if( insert_key < current_node->key[i] && i >=  
83         start ) {  
84         current_node->next[i+1] =  
85             current_node->next[i];  
86         current_node->key[i+1] =  
87             current_node->key[i];  
88         i -=1;  
89     } else {  
90         current_node->next[i+1] =  
91             insert_pt;  
92         current_node->key[i+1] =  
93             insert_key;  
94         insert_done = 1;  
95     }  
}
```

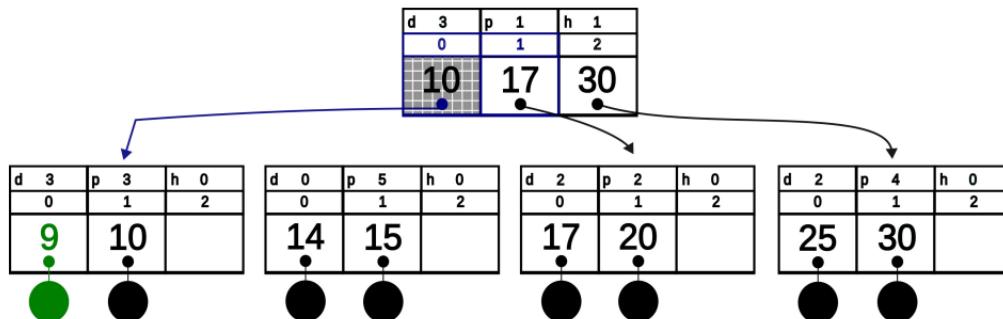
```
// Step 8 - 5  
tree = (*p 1);  
new_key = 5; new_object = (*5);  
  
current_node = (*p 3); ->height = 0; ->degree = 3;  
new_node = (*p 5); ->height = 0; -> degree = 0;  
  
insert_pt = (* 9); insert_key = 9;  
finished = 0; insert_done = 0;  
  
stack = [(*p 1)]  
start = 0; i = 0; j = -1;
```



## B-Tree Operations - Search (Example) XXXIII

```
81 while( !insert_done) {  
82     if( insert_key < current_node->key[i] && i >=  
83         start ) {  
84         current_node->next[i+1] =  
85             current_node->next[i];  
86         current_node->key[i+1] =  
87             current_node->key[i];  
88         i -=1;  
89     } else {  
90         current_node->next[i+1] =  
91             insert_pt;  
92         current_node->key[i+1] =  
93             insert_key;  
94         insert_done = 1;  
95     }  
}
```

```
// Step 8 - 7  
tree = (*p 1);  
new_key = 5; new_object = (*5);  
  
current_node = (*p 3); ->height = 0; ->degree = 3;  
new_node = (*p 5); ->height = 0; -> degree = 0;  
  
insert_pt = (* 9); insert_key = 9;  
finished = 0; insert_done = 1;  
  
stack = [(*p 1)]  
start = 0; i = -1; j = -1;
```



## B-Tree Operations - Search (Example) XXXIV

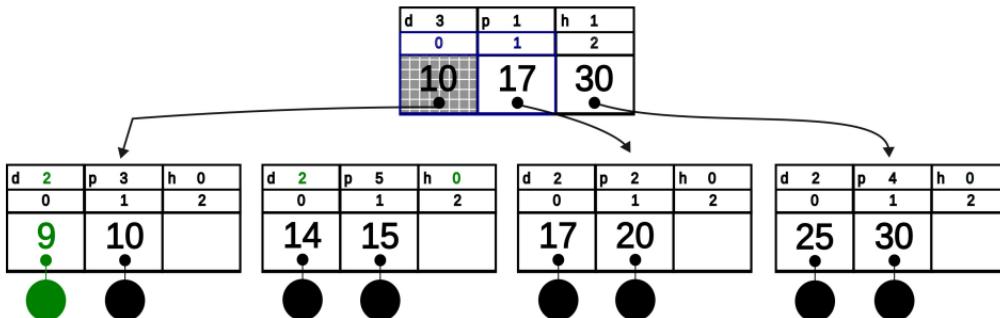
```
98 current_node->degree = (((2 * ALPHA) - 1)+1 - (((2 *
99    ↵ ALPHA) - 1)+1)/2);
00 new_node->degree = (((2 * ALPHA) - 1)+1)/2;
01 new_node->height = current_node->height;
02 /* split nodes complete, now insert the new node
03    ↵ above */
04 insert_pt = new_node;
05 insert_key = new_node->key[0];
06 if( ! stack_empty() ) {
07     /* not at root; move one level up*/
08     current_node = pop();
09 }
```

```
// Step 8 - 8
tree = (*p 1);
new_key = 5; new_object = (*5);

//current_node = (*p 3); ->height = 0; ->degree = 2;
current_node = (*p 1); ->height = 1; ->degree = 3;
new_node = (*p 5); ->height = 0; -> degree = 2;

insert_pt = (*p 5); insert_key = 14;
finished = 0;

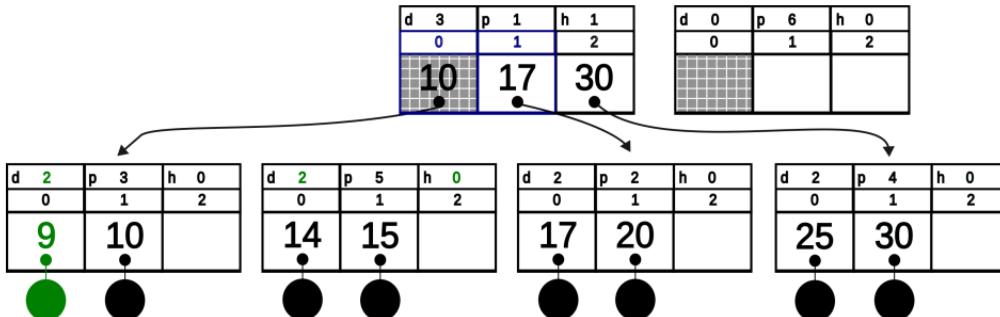
// Step 8 - 9
tree = (*p 1);
new_key = 5; new_object = (*5);
```



## B-Tree Operations - Search (Example) XXXV

```
33 while( !finished ){  
42     if( current_node->degree < (2 * ALPHA) - 1 ) {  
60         else {  
61             /* node is full, have to split the node*/  
62             tree_node_t *new_node;  
63             int j, insert_done = 0;  
64             new_node = get_node();  
65             i = ((2 * ALPHA) - 1)-1;  
66             j = (((2 * ALPHA) - 1)-1)/2;
```

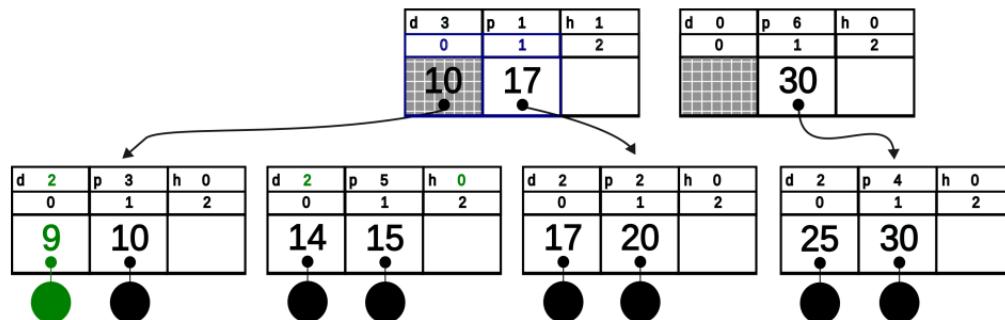
```
// Step 8 - 9  
tree = (*p 1);  
new_key = 5; new_object = (*5);  
  
current_node = (*p 1); ->height = 1; ->degree = 3;  
//new_node = (*p 5); ->height = 0; -> degree = 2;  
new_node = (*p 6); ->height = 0; -> degree = 2;  
  
insert_pt = (*p 5); insert_key = 14;  
finished = 0; insert_done = 0;  
  
stack = [];  
start = 1; i = 2; j = 1;
```



## B-Tree Operations - Search (Example) XXXVI

```
67 while( j >= 0 ) {  
68     /* copy upper half to new node */  
69     if( insert_done || insert_key <  
70         current_node->key[i] ) {  
71         new_node->next[j] =  
72             current_node->next[i];  
73         new_node->key[j--] =  
74             current_node->key[i--];  
75     } else {  
76         new_node->next[j] = insert_pt;  
77         new_node->key[j--] = insert_key;  
78         insert_done = 1;  
79     }  
}
```

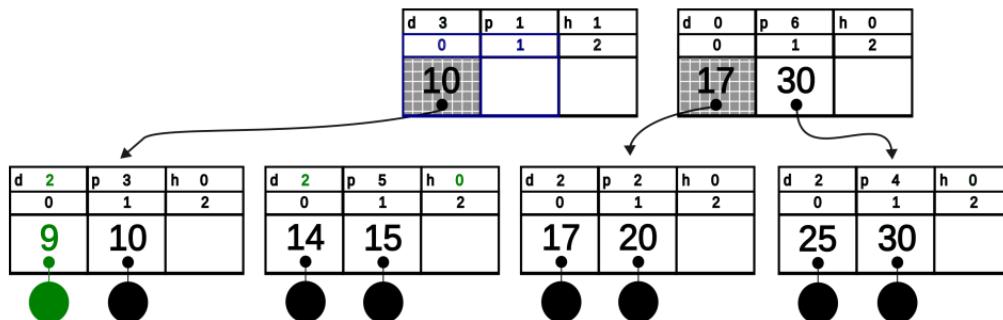
```
// Step 8 - 10  
tree = (*p 1);  
new_key = 5; new_object = (*5);  
  
current_node = (*p 1); ->height = 1; ->degree = 3;  
new_node = (*p 6); ->height = 0; -> degree = 2;  
  
insert_pt = (*p 5); insert_key = 14;  
finished = 0; insert_done = 0;  
  
stack = [];  
start = 1; i = 1; j = 0;
```



## B-Tree Operations - Search (Example) XXXVII

```
67 while( j >= 0 ) {  
68     /* copy upper half to new node */  
69     if( insert_done || insert_key <  
70         current_node->key[i] ) {  
71         new_node->next[j] =  
72             current_node->next[i];  
73         new_node->key[j--] =  
74             current_node->key[i--];  
75     } else {  
76         new_node->next[j] = insert_pt;  
77         new_node->key[j--] = insert_key;  
78         insert_done = 1;  
79     }  
}
```

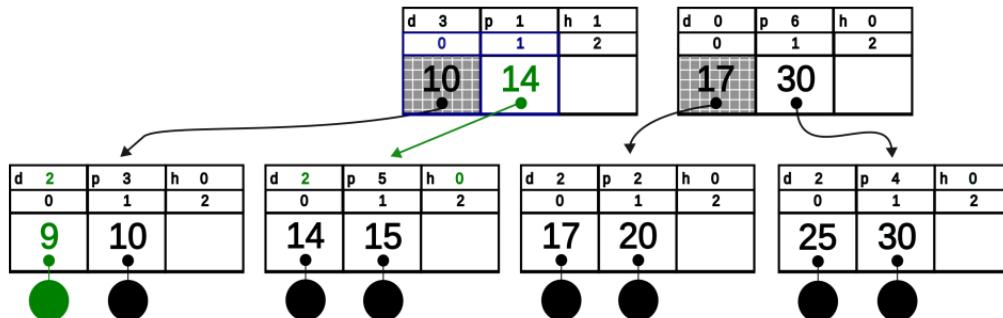
```
// Step 8 - 11  
tree = (*p 1);  
new_key = 5; new_object = (*5);  
  
current_node = (*p 1); ->height = 1; ->degree = 3;  
new_node = (*p 6); ->height = 0; -> degree = 2;  
  
insert_pt = (*p 5); insert_key = 14;  
finished = 0; insert_done = 0;  
  
stack = [];  
start = 1; i = 0; j = -1;
```



## B-Tree Operations - Search (Example) XXXVIII

```
81 while( !insert_done) {  
82     if( insert_key < current_node->key[i] && i >=  
83         start ) {  
84         current_node->next[i+1] =  
85             current_node->next[i];  
86         current_node->key[i+1] =  
87             current_node->key[i];  
88         i -=1;  
89     } else {  
90         current_node->next[i+1] =  
91             insert_pt;  
92         current_node->key[i+1] =  
93             insert_key;  
94         insert_done = 1;  
95     }  
}
```

```
// Step 8 - 12  
tree = (*p 1);  
new_key = 5; new_object = (*5);  
  
current_node = (*p 1); ->height = 1; ->degree = 3;  
new_node = (*p 6); ->height = 0; -> degree = 2;  
  
insert_pt = (*p 5); insert_key = 14;  
finished = 0; insert_done = 1;  
  
stack = [];  
start = 1; i = 0; j = -1;
```



## B-Tree Operations - Search (Example) XXXIX

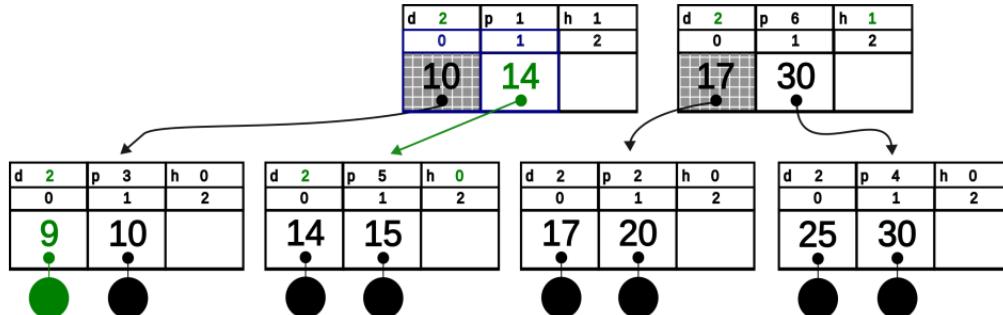
```
98 current_node->degree = ((2 * ALPHA) - 1)+1 - (((2 *
99   ↵ ALPHA) - 1)+1)/2);
00 new_node->degree = (((2 * ALPHA) - 1)+1)/2;
01 new_node->height = current_node->height;
02 /* split nodes complete, now insert the new node
03   ↵ above */
04 insert_pt = new_node;
05 insert_key = new_node->key[0];
06 if( ! stack_empty() ) {
07   /* not at root; move one level up*/
08   current_node = pop();
09 }
```

```
// Step 8 - 13
tree = (*p 1);
new_key = 5; new_object = (*5);

current_node = (*p 1); ->height = 1; ->degree = 2;
new_node = (*p 6); ->height = 1; -> degree = 2;

insert_pt = (*p 6); insert_key = 17;
finished = 0; insert_done = 1;

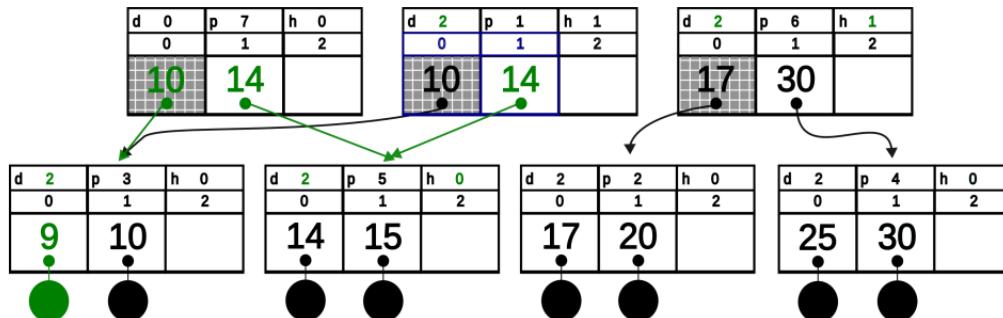
stack = [];
start = 1; i = 0; j = -1;
```



## B-Tree Operations - Search (Example) XL

```
08  
09 else {  
10     /* splitting root: needs copy to keep root  
11     ↵ address*/  
12     new_node = get_node();  
13     for( i=0; i < current_node->degree; i++ ) {  
14         new_node->next[i] =  
15             current_node->next[i];  
16         new_node->key[i] =  
17             current_node->key[i];  
18     }  
19 }
```

```
// Step 8 - 14  
tree = (*p 1);  
new_key = 5; new_object = (*5);  
  
current_node = (*p 1); ->height = 1; ->degree = 2;  
new_node = (*p 7); ->height = 0; -> degree = 0;  
insert_pt = (*p 6); insert_key = 17;  
  
finished = 0;
```



## B-Tree Operations - Search (Example) XLI

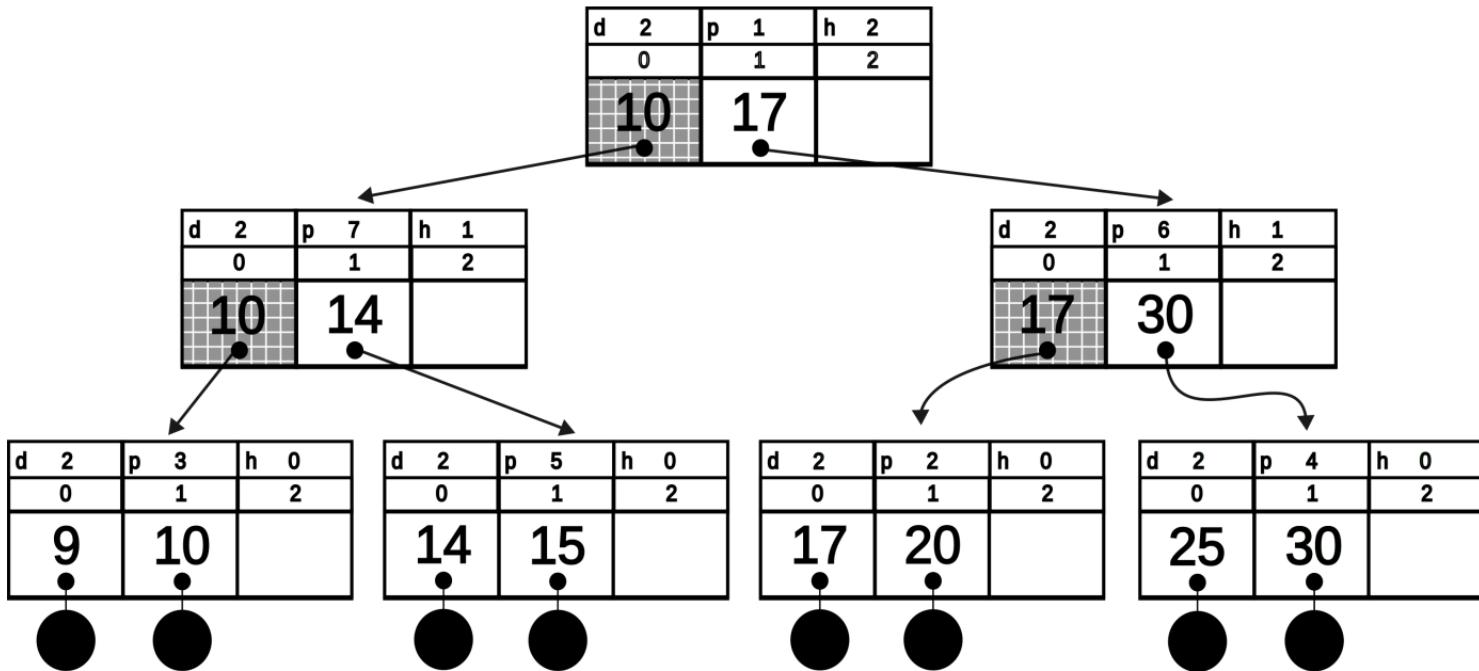
```
17 new_node->height =
18     current_node->height;
19 new_node->degree =
20     current_node->degree;
21 current_node->height += 1;
22 current_node->degree = 2;
23 current_node->next[0] = new_node;
24 current_node->next[1] = insert_pt;
25 current_node->key[1] = insert_key;
26 finished =1;
```

```
// Step 8 - 15
tree = (*p 1);
new_key = 5; new_object = (*5);

current_node = (*p 1); ->height = 2; ->degree = 2;
new_node = (*p 7); ->height = 1; -> degree = 2;
insert_pt = (*p 6); insert_key = 17;

finished = 1;
```

## B-Tree Operations - Search (Example) XLII



## B-Tree Operation - Delete Value I

- ▶ The deletion algorithm, just like the insert or find, in the B-Tree almost has nothing to share with any tree deletion algorithm.
- ▶ Also, the first part is a find algorithm where we are going to search if the key to delete exists and if it does and its position, and we store the nodes that we access and their pointer index on separated stacks.
- ▶ Then, when reached a leaf with the value to delete, we just delete it. But now, we have to check for all the rebalancing cases.
- ▶ If the current balancing node has a degree greater than  $\alpha$  we can stop the rebalancing process.
- ▶ Then, if we are not on the root, we will check if our current node is not the last sub-tree on the parent node.
- ▶ If the node isn't, we will check if the next neighbor node can share a key, or if it has more than  $\alpha$  keys.
- ▶ In the case that the neighbor doesn't have  $\alpha$  elements we are going to join both nodes.
- ▶ Then, we are going to check if the parent node needs some rebalancing and restart the rebalancing process.
- ▶ Now, in the case that we are the the last sub-tree of the parent node we can't just chare elements with the next neighbor.
- ▶ So we are just going to do the same thing but with the previous neighbor. Both process, the sharing or the join.
- ▶ Also, if we reach the root on the rebalancing process, we check if the root has at least one key, and isn't a leaf at the same time.
- ▶ But if the root doesn't have any element, we just return the root memory.
- ▶ When we finally exit the rebalancing loop, we just return the object that we deleted.

## B-Tree Operation - Delete Value II

```
1 object_t *delete(tree_node_t *tree, key_t delete_key) {
2     tree_node_t *current, *tmp_node;
3     int finished, i, j;
4     current = tree;
5     create_node_stack();
6     create_index_stack();
7     while( current->height > 0 ) {
8         /* not at leaf level */
9         int lower, upper;
10        /* binary search among keys */
11        lower = 0;
12        upper = current->degree;
13        while( upper > lower +1 ) {
14            if( delete_key < current->key[ (upper+lower)/2 ] )
15                upper = (upper+lower)/2;
16            else
17                lower = (upper+lower)/2;
18        }
19        push_index_stack( lower );
20        push_node_stack( current );
21        current = current->next[lower];
22    }
23    /* now current is leaf node from which we delete */
24    for ( i=0; i < current->degree ; i++ )
25        if( current->key[i] == delete_key )
```

## B-Tree Operation - Delete Value III

```
27     break;
28
29     if( i == current->degree ) {
30         /* delete failed; key does not exist */
31         return( NULL );
32     } else {
33         /* key exists, now delete from leaf node */
34         object_t *del_object;
35         del_object = (object_t *) current->next[i];
36         current->degree -=1;
37         while( i < current->degree ) {
38             current->next[i] = current->next[i+1];
39             current->key[i] = current->key[i+1];
40             i+=1;
41         }
42         /* deleted from node, now rebalance */
43         finished = 0;
44         while( ! finished ) {
45             if(current->degree >= ALPHA ) {
46                 finished = 1;
47                 /* node still full enough, can stop */
48             }
49             else {
50                 /* node became underfull */
51                 if( stack_empty() ) {
52                     /* current is root */
53                     if(current->degree >= 2 )
54                         /* root still necessary */
```

## B-Tree Operation - Delete Value IV

```
54                     finished = 1;
55             else if ( current->height == 0 )
56                 /* deleting last keys from root */
57                 finished = 1;
58             else {
59                 /* delete root, copy to keep address */
60                 tmp_node = current->next[0];
61                 for( i=0; i< tmp_node->degree; i++ ) {
62                     current->next[i] = tmp_node->next[i];
63                     current->key[i] = tmp_node->key[i];
64                 }
65                 current->degree =
66                     tmp_node->degree;
67                 current->height =
68                     tmp_node->height;
69                 return_node( tmp_node );
70                 finished = 1;
71             }
72             /* done with root */
73         } else {
74             /* delete from non-root node */
75             tree_node_t *upper, *neighbor;
76             int curr;
77             upper = pop_node_stack();
78             curr = pop_index_stack();
79             if( curr < upper->degree -1 ) {
80                 /* not last*/
```

## B-Tree Operation - Delete Value V

```
81         neighbor = upper->next[curr+1];
82         if( neighbor->degree > ALPHA ) {
83             /* sharing possible */
84             i = current->degree;
85             if( current->height > 0 )
86                 current->key[i] =
87                     upper->key[curr+1];
88             else {
89                 /* on leaf level, take leaf key */
90                 current->key[i] =
91                     neighbor->key[0];
92                 neighbor->key[0] =
93                     neighbor->key[1];
94             }
95             current->next[i] =
96                 neighbor->next[0];
97             upper->key[curr+1] =
98                 neighbor->key[1];
99             neighbor->next[0] =
100                neighbor->next[1];
101            for( j = 2; j < neighbor->degree; j++ ) {
102                neighbor->next[j-1] =
103                    neighbor->next[j];
104                neighbor->key[j-1] =
105                    neighbor->key[j];
106            }
107            neighbor->degree -=1;
```

## B-Tree Operation - Delete Value VI

```
108                     current->degree+=1;
109                     finished =1;
110 } /* sharing complete */
111 else {
112     /* must join */
113     i = current->degree;
114     if( current->height > 0 )
115         current->key[i] =
116             upper->key[curr+1];
117     else /* on leaf level, take leaf key */
118         current->key[i] =
119             neighbor->key[0];
120     current->next[i] =
121         neighbor->next[0];
122     for( j = 1; j < neighbor->degree; j++ ) {
123         current->next[++i] =
124             neighbor->next[j];
125         current->key[i] =
126             neighbor->key[j];
127     }
128     current->degree = i+1;
129     return_node( neighbor );
130     upper->degree -=1;
131     i = curr+1;
132     while( i < upper->degree ) {
133         upper->next[i] =
134             upper->next[i+1];
```

## B-Tree Operation - Delete Value VII

```
135                     upper->key[i] =
136                     upper->key[i+1];
137                     i +=1;
138                 }
139                 /* deleted from upper, now propagate up */
140                 current = upper;
141             } /* end of share/joining if-else*/
142         }
143     else {
144         /* current is last entry in upper */
145         neighbor = upper->next[curr-1]
146         if( neighbor->degree > ALPHA ) {
147             /* sharing possible */
148             for( j = current->degree; j > 1; j-- ) {
149                 current->next[j] =
150                     current->next[j-1];
151                 current->key[j] =
152                     current->key[j-1];
153             }
154             current->next[1] =
155                 current->next[0];
156             i = neighbor->degree;
157             current->next[0] =
158                 neighbor->next[i-1];
159             if( current->height > 0 ) {
160                 current->key[1] =
161                     upper->key[curr];
```

## B-Tree Operation - Delete Value VIII

```
162 }  
163 else {  
164     /* on leaf level, take leaf key */  
165     current->key[1] =  
166         current->key[0];  
167     current->key[0] =  
168         neighbor->key[i-1];  
169 }  
170 upper->key[curr] =  
171     neighbor->key[i-1];  
172 neighbor->degree -=1;  
173 current->degree+=1;  
174 finished =1;  
175 } /* sharing complete */  
176 else {  
177     /* must join */  
178     i = neighbor->degree;  
179     if( current->height > 0 )  
180         neighbor->key[i] =  
181             upper->key[curr];  
182     else /* on leaf level, take leaf key */  
183         neighbor->key[i] =  
184             current->key[0];  
185     neighbor->next[i] =  
186         current->next[0];  
187     for( j = 1; j < current->degree; j++ ) {  
188         neighbor->next[++i] =
```

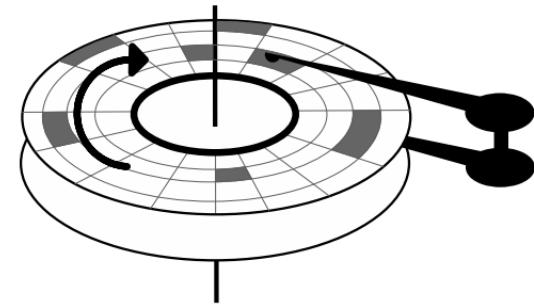
## B-Tree Operation - Delete Value IX

```
189                     current->next[j];
190                     neighbor->key[i] =
191                         current->key[j];
192                 }
193                 neighbor->degree = i+1;
194                 return_node( current );
195                 upper->degree -=1;
196                 /* deleted from upper, now propagate up */
197                 current = upper;
198             } /* end of share/joining if-else */
199         } /* end of current is (not) last in upper if-else*/
200     } /* end of delete root/non-root if-else */
201     } /* end of full/underfull if-else */
202 } /* end of while not finished */

203
204     return( del_object );
205
206 } /* end of delete object exists if-else */
207 }
```

# B-Tree Secondary Memory Access I

- ▶ The B-Tree is fairly good for storing data in external memory in comparison to height, weight or search trees.
- ▶ The limit of  $2\alpha$  keys help us by having a balance availability and fragmentation of the data.
- ▶ But, this limit also make that if we need to re-balance the tree the operation will take  $\Theta(\alpha \log n)$ , updating all the split nodes.
- ▶ This operation doesn't affect much in main memory, but in secondary memory where the access time isn't always constant
- ▶ Each read on the secondary memory can make a lot of problems in the execution of the code.



**Figure:** External storage with the sectors to access highlighted

## B-Tree Secondary Memory Access II

Retrieval	Insertion w/ overflow	Deletion w/ underfull
$\Omega t = 1 \ w = 0$	$t = h \ w = 1$	$t = h \ w = 1$
$\Theta t \leq h \ w = 0$	$t \leq h + 2 + \frac{2}{\alpha} \ w \leq 3 + \frac{2}{\alpha}$	$t \leq 3h - 2 \ w \leq 2h + 1$
$O t = h \ w = 0$	$t = 3h - 2 \ w = 2h + 1$	$t = 3h - 2 \ w = 2h + 1$

- Where  $t$  is the number of fetch and readings of nodes on the secondary memory.
- And  $w$  is the number of writings of nodes on the secondary memory.

[1]

# Bibliography I

- [1] R. Bayer and E. M. McCreight. "Organization and maintenance of large ordered indexes". In: *Acta Informatica* 1.3 (1972), pp. 173–189. ISSN: 0001-5903, 1432-0525. DOI: 10.1007/BF00288683. URL: <http://link.springer.com/10.1007/BF00288683> (visited on 10/17/2024).
- [2] Peter Brass. *Advanced Data Structures*. Google-Books-ID: g8rZoSKLbhwC. Cambridge University Press, Sept. 8, 2008. 456 pp. ISBN: 978-0-521-88037-4.
- [3] Thomas H. Cormen et al. *Introduction To Algorithms*. Google-Books-ID: NLngYyWFI\_YC. MIT Press, 2001. 1216 pp. ISBN: 978-0-262-03293-3.
- [4] Scott Huddleston and Kurt Mehlhorn. "A new data structure for representing sorted lists". In: *Acta Inf.* 17.2 (June 1, 1982), pp. 157–184. ISSN: 0001-5903. DOI: 10.1007/BF00288968. URL: <https://doi.org/10.1007/BF00288968> (visited on 10/17/2024).