B-Trees & AB-Trees

Martín Hernández
Juan Mendivelso

Contents

B-Tree History

B-Trees where 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

"(...) 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)

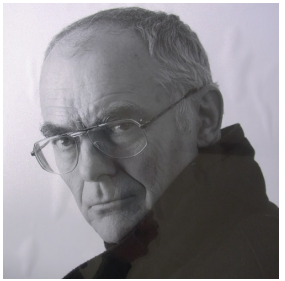


Figure: Rudolf Bayer

"I am occasionally asked what the B in B-Tree means. (...) We wanted the name to be short, quick to type, easy to remember. It honored our employer, Boeing Airplane Company, but we wouldn't have to request permission to use the name. It suggested Balance. Rudolf Bayer was the senior researcher of the two of us. (...) I don't recall one meaning standing out above the others that day. Rudolf is fond of saying that the more you think about what the B could mean, the more you learn about B-Trees, and that is good. " (Bayer)

Figure: Edward McCreight

B-Tree Definition

> 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, α 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 cuple 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

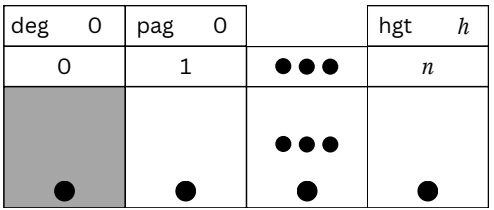


Figure: Node of a B-Tree

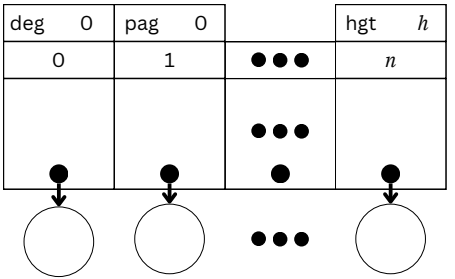


Figure: Leaf of a B-Tree

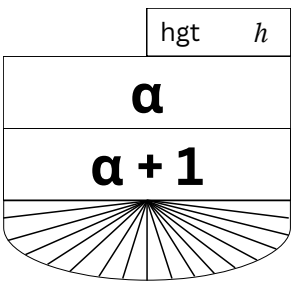


Figure: Generic Node of a B-Tree

B-Tree Properties—The α constant

- The main property of the B-Trees is the α, a predefined constant.
- The α 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 α to 2α 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.

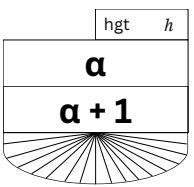


Figure: Miminum Keys and Sub-Trees on a Node

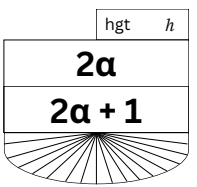


Figure: Maximun Keys and Sub-Trees on a Node

B-Tree Properties—The α constant

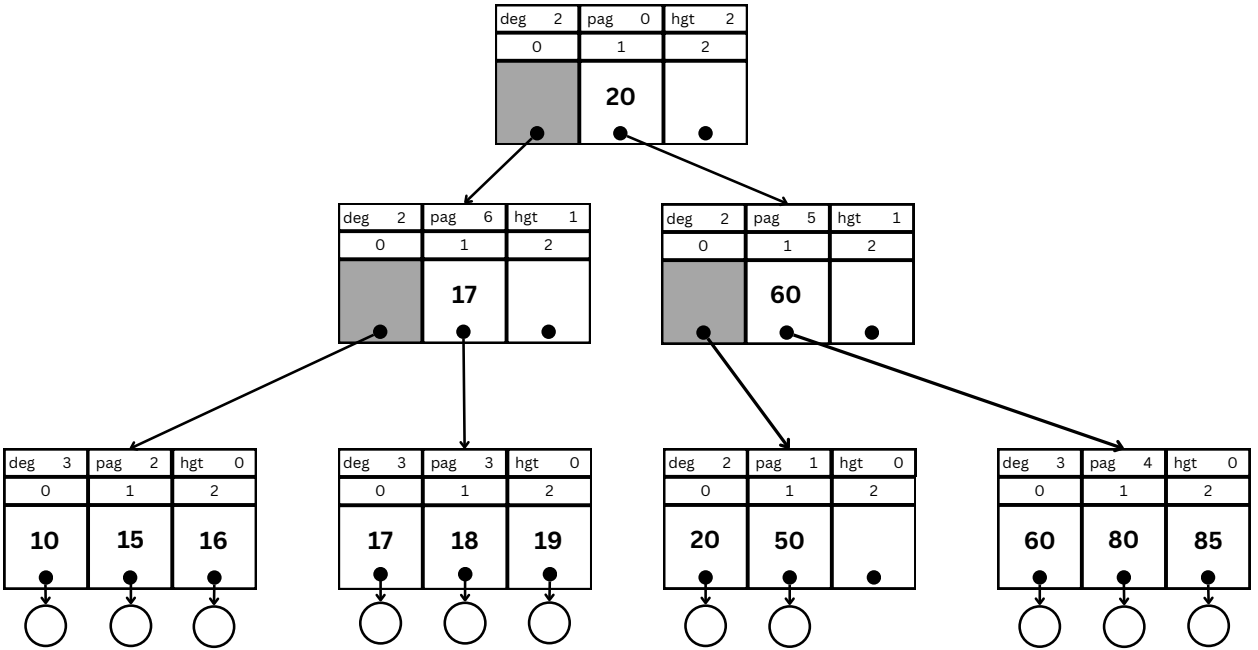


Figure: B-Tree, $t\,(2,2)$

B-Tree Properties—The α constant

> 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

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

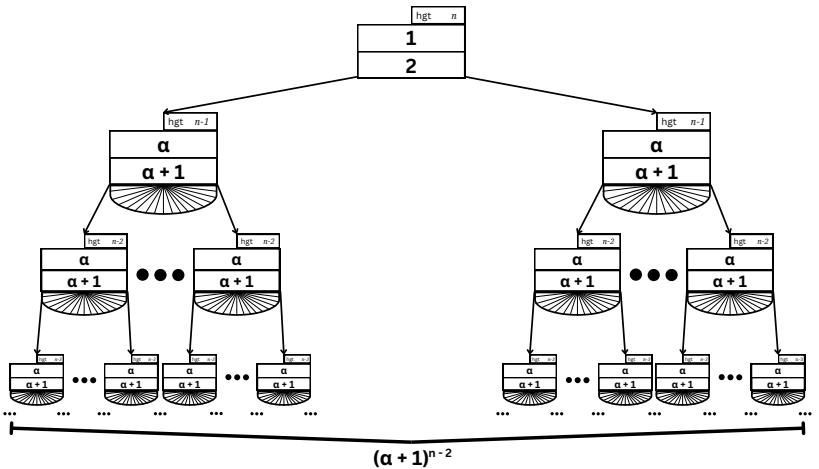


Figure: B-Tree w/ the least number of nodes

B-Tree Properties—The α constant

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

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

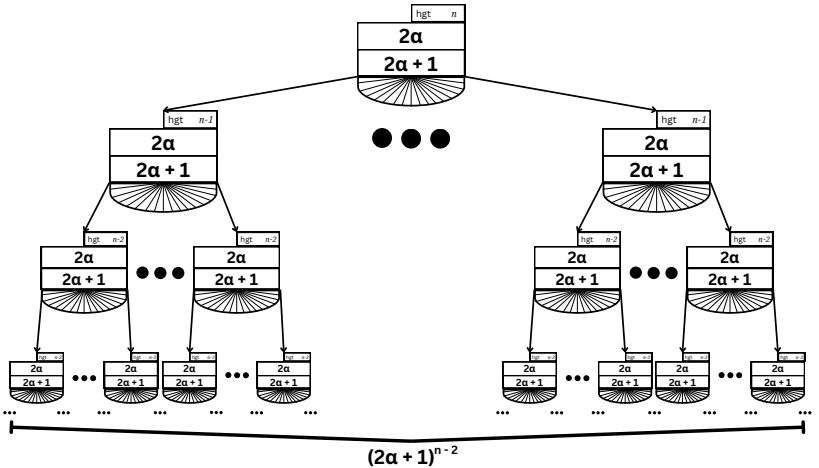


Figure: B-Tree w/ the most number of nodes

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

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

□

B-Tree Properties—The α constant

- Keep in mind that the *Branching Factor* of a B-Tree might change from each implementation, mostly in papers and books.
- For example, on the original paper by Bayer and McCreight of B-Trees[1], the *Branching Factor* goes from $\alpha + 1$ to $2\alpha + 1$ subtrees and from α to 2α keys on a node.
- But in the book made by Brass[2], the *Branching Factor* goes from α to $2\alpha - 1$ for both, subtrees and keys in a node.
- And on the original paper by Huddleston and Mehlhorn of AB-Trees[4] keeps the same *Branching Factor* as Brass.
- But, we will see later that by limiting the upper bound of the *Branching Factor* to something greater than 2α we will reach a even greater performance from this type of data structure.

B-Tree Properties—Keys and Sub-trees

> 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 $\{k_1, k_2, \ldots, k_j\}$.
> 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 $\alpha \leq l \leq 2\alpha$.

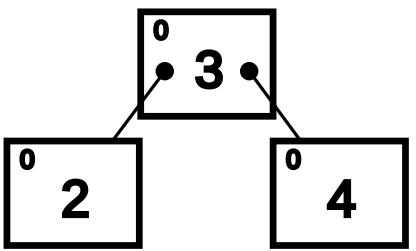


Figure: Simple node of a Normal Binary Tree

B-Tree Properties—Keys and Sub-trees

> Now, we also define the set of sub-trees of N as $\{p_0, p_1, \ldots, p_j\}$.
> 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.

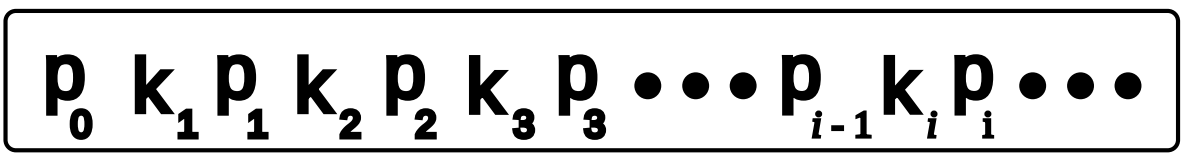


Figure: Order of the Subtree Pointers and Keys.

B-Tree Properties—Keys and Sub-trees

> 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.

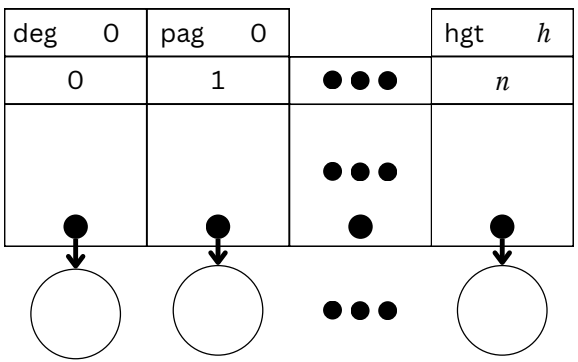


Figure: Leaf of a B-Tree

B-Tree Properties—Keys and Sub-trees

> 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.

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

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

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

B-Tree Properties—Keys and Sub-trees

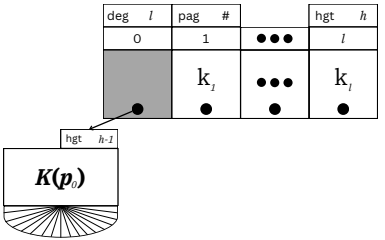


Figure: Sub-tree Keys (Case 1)

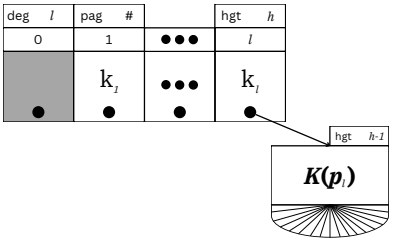


Figure: Sub-tree Keys (Case 3)

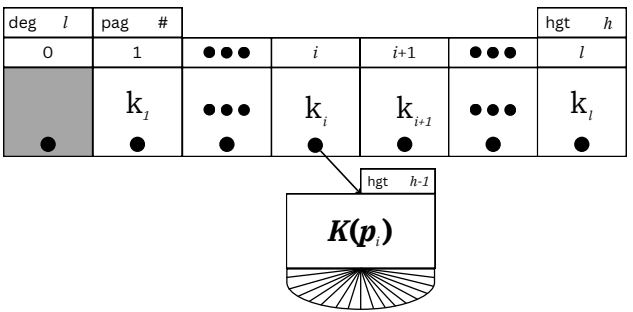


Figure: Sub-tree Keys (Case 2)

B-Tree Properties—Height

> 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):

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

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

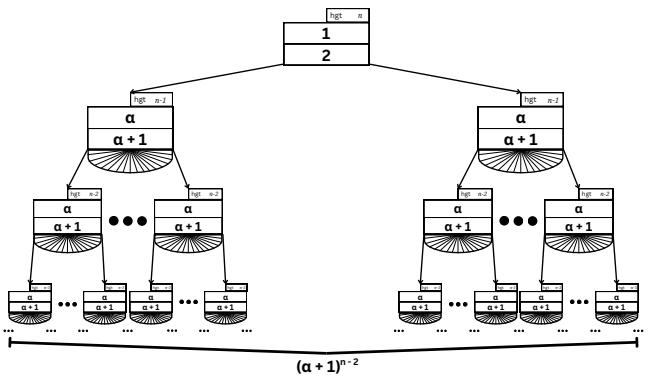


Figure: B-Tree w/ the least number of nodes

B-Tree Properties—Height

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

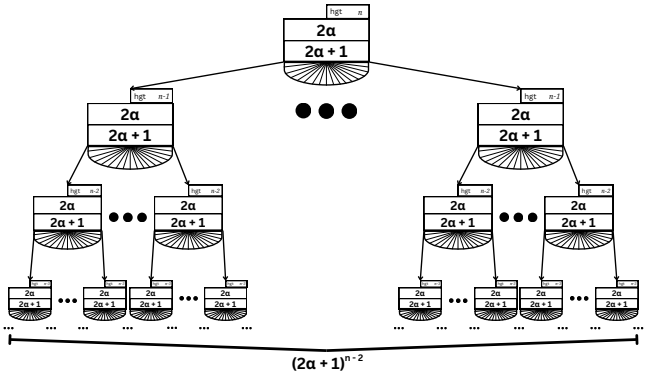


Figure: B-Tree w/ the most number of nodes

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

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

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

B-Tree Properties—Height

- 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 α.
> 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:

$$\begin{aligned} &\forall y \in K(p_0); \quad y < k_1 \\ &\forall y \in K(p_i); \quad k_i \leq y < k_{i+1}; \quad 0 < i < l \wedge i \in \mathbb{N} \\ &\forall y \in K(p_l); \quad k_l \leq y \end{aligned}$$

> The number of nodes of a B-Tree is bounded by: $1 + \frac{2}{\alpha}\left((\alpha+1)^{h-1} - 1\right) \leq N(T) \leq \frac{1}{2\alpha}\left((2\alpha+1)^{h} - 1\right)$
> 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  #define 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 asume 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 it's data.
> This process takes more time than the general data fetch from main memory.
> So, the fewer times we do this process the better.

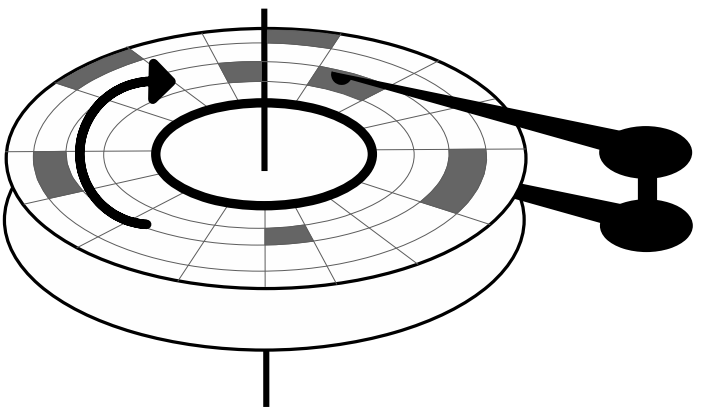


Figure: External storage with the sectors to access highlighted

B-Tree Operations—Creating an empty B-Tree

> We use `create_tree()` to create a 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

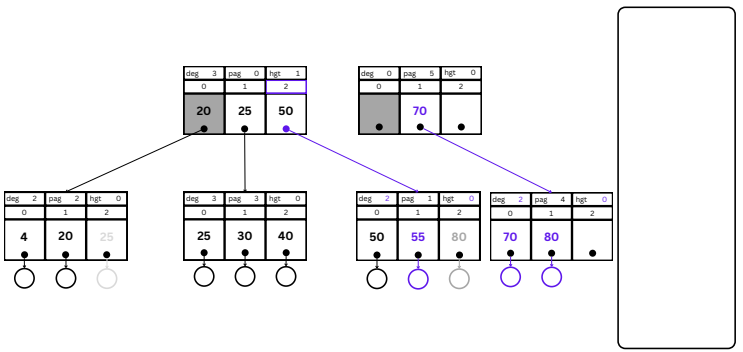
> The changes of this operations are mainly focused on the search part, since we have to compare to an array of keys and not only the node key.

> 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
22     else {
23       if( current_node->key[lower] == query_key )
```

B-Tree Operations—Search

```
24          object = (object_t *) current_node->next[lower];
25        else
26          object = NULL;
27        return( object );
28      }
29    }
30  }
```

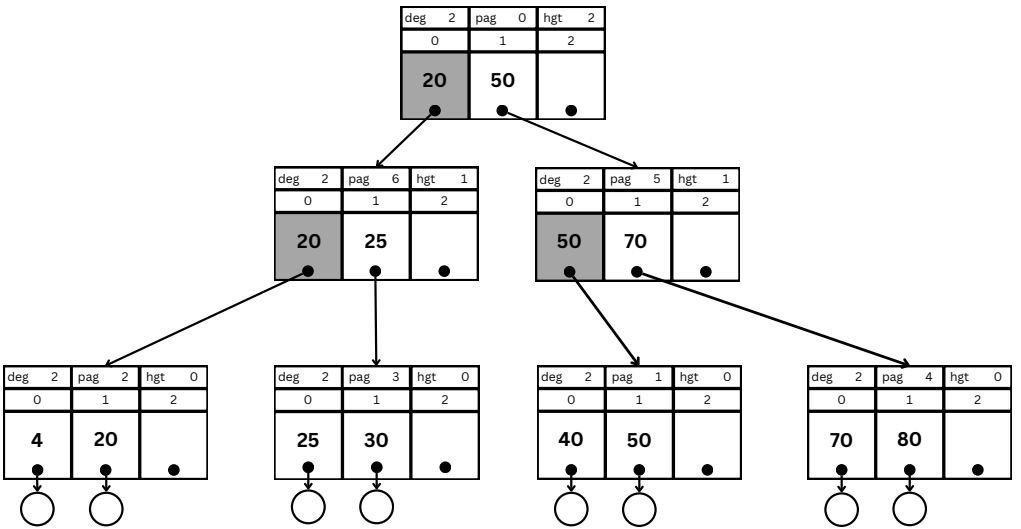


> Let's search for 70 in this $t\,(2,2)$ B-Tree.

B-Tree Operations—Search (Example)

```
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
```

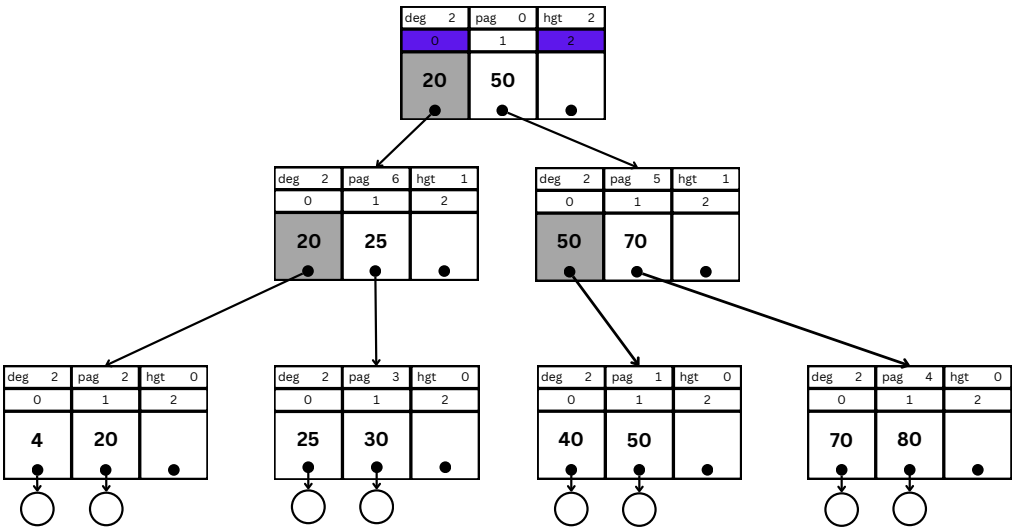
> Search 1; Step 1;
> tree=(*pag 0); query_key=70;
> object;
> current_node=(*pag 0);



B-Tree Operations—Search (Example)

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

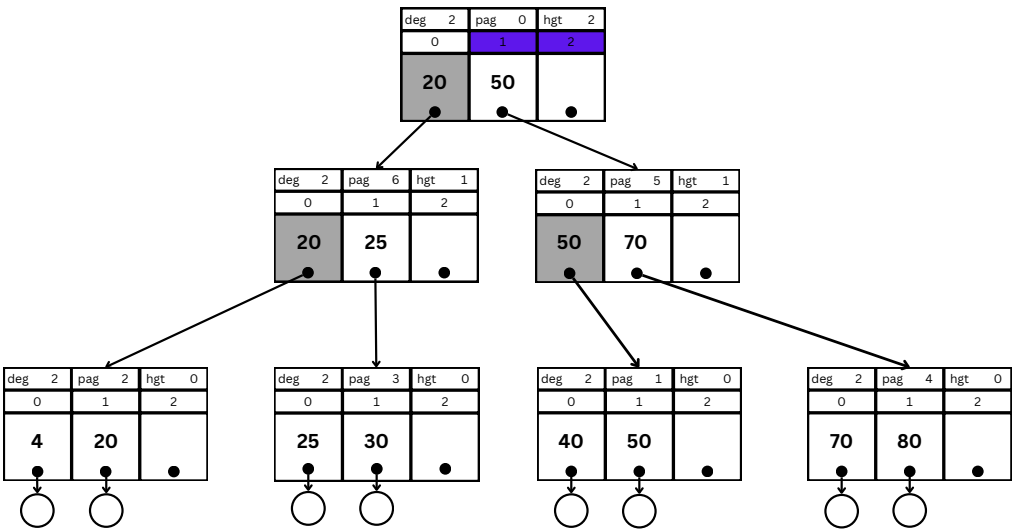
> Search 1; Step 2;
> tree=(*pag 0); query_key=70;
> object; lower=0; upper=2;
> current_node=(*pag 0);



B-Tree Operations—Search (Example)

```
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      }
```

> Search 1; Step 3;
> tree=(*pag 0); query_key=70;
> object; lower=0 → 1; upper=2; med=1;
> current_node=(*pag 0);

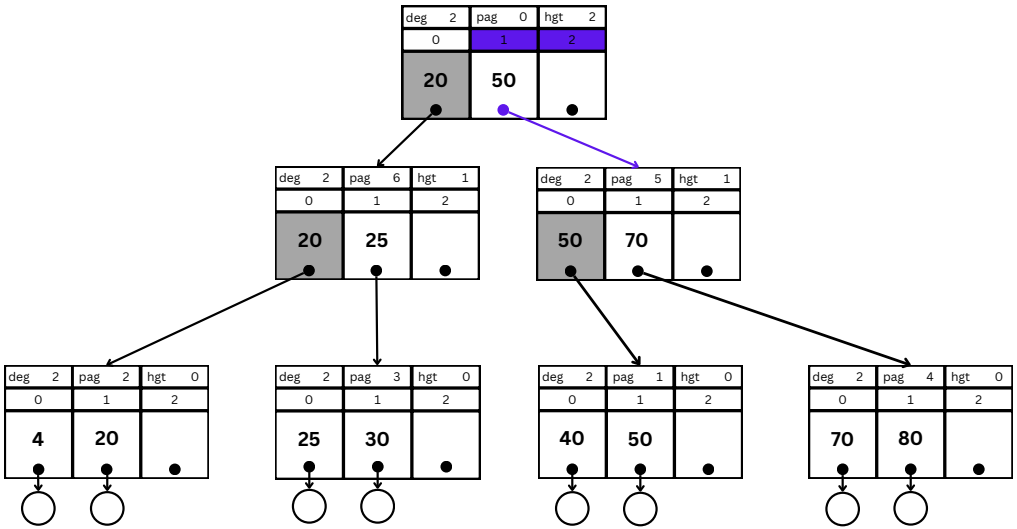


B-Tree Operations—Search (Example)

```
12      while( upper > lower +1 ) {
```

```
18      }
19      if( current_node->height > 0)
20        current_node = current_node->next[lower];
21
```

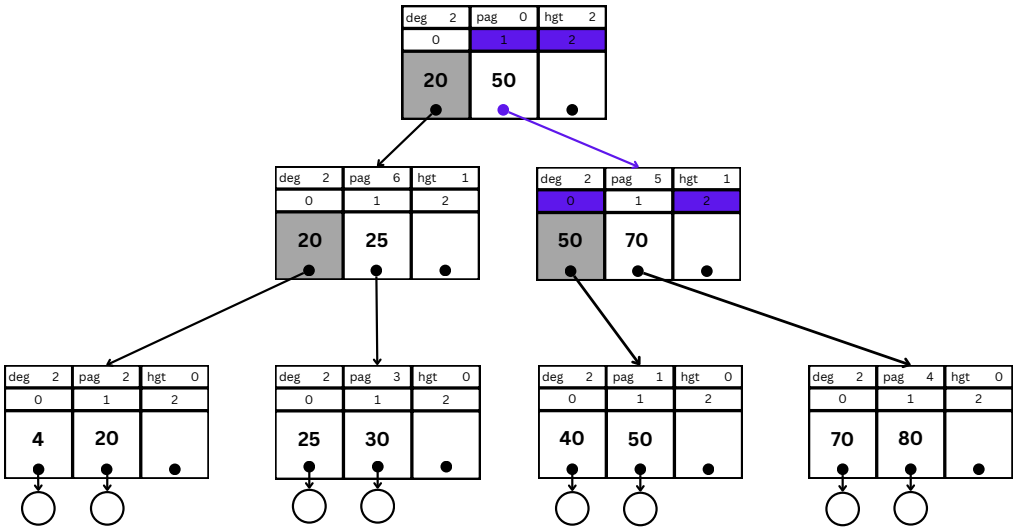
> Search 1; Step 4;
> tree=(*pag 0); query_key=70;
> object; lower=1; upper=2;
> current_node=(*pag 0) → (*pag 5);



B-Tree Operations—Search (Example)

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

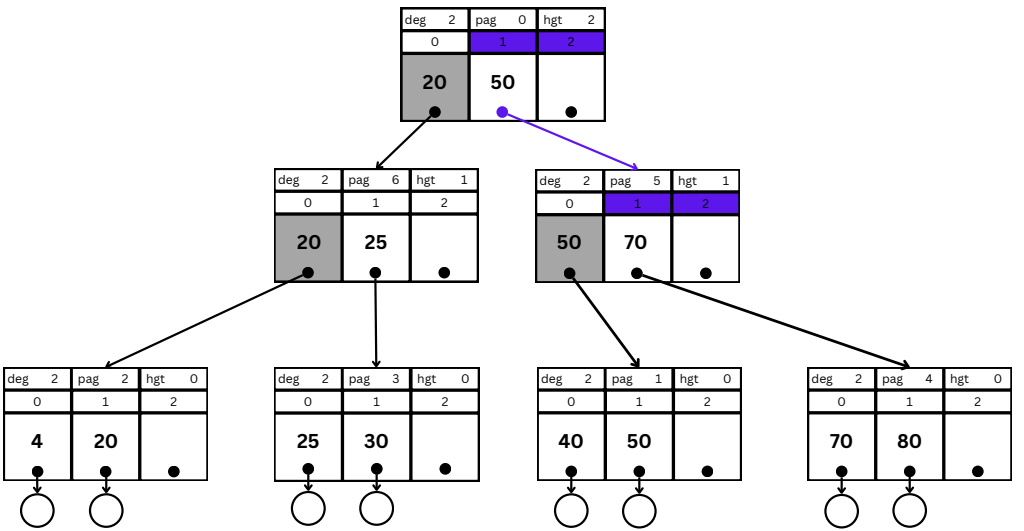
> Search 1; Step 5;
> tree=(*pag 0); query_key=70;
> object; lower=0; upper=2;
> current_node=(*pag 5);



B-Tree Operations—Search (Example)

```
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      }
```

> Search 1; Step 6;
> tree=(*pag 0); query_key=70;
> object; lower=0 → 1; upper=2; med=1;
> current_node=(*pag 5);

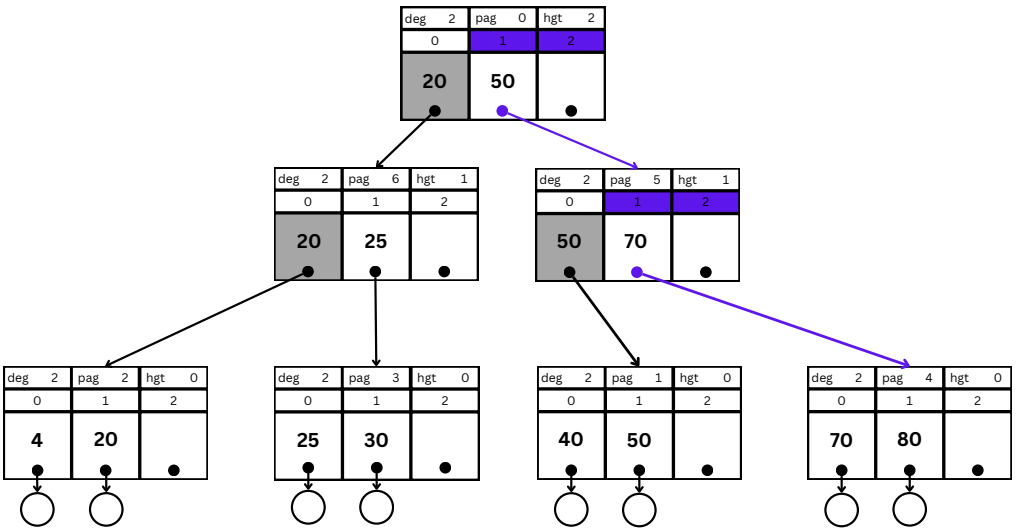


B-Tree Operations—Search (Example)

```
12      while( upper > lower +1 ) {
```

```
18      }
19      if( current_node->height > 0)
20        current_node = current_node->next[lower];
21
```

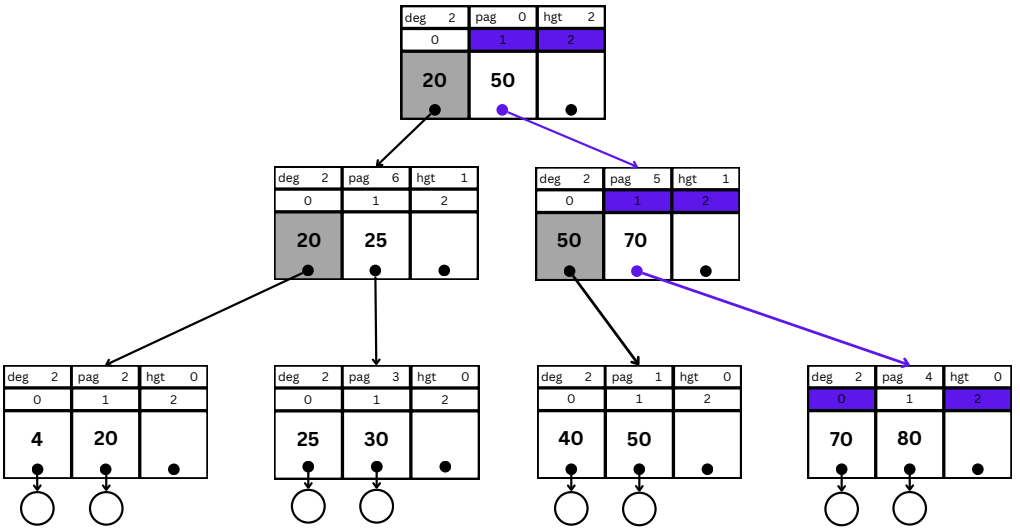
> Search 1; Step 7;
> tree=(*pag 0); query_key=70;
> object; lower=1; upper=2;
> current_node=(*pag 5) → (*pag 4);



B-Tree Operations—Search (Example)

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

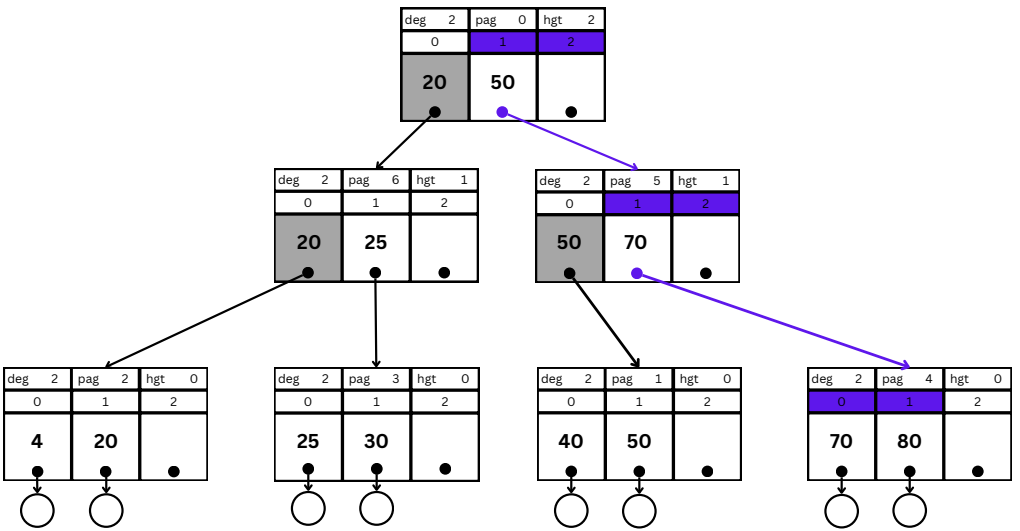
> Search 1; Step 8;
> tree=(*pag 0); query_key=70;
> object; lower=0; upper=2;
> current_node=(*pag 4);



B-Tree Operations—Search (Example)

```
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      }
```

> Search 1; Step 9;
> tree=(*pag 0); query_key=70;
> object; lower=0; upper=2 → 1; med=1;
> current_node=(*pag 4);



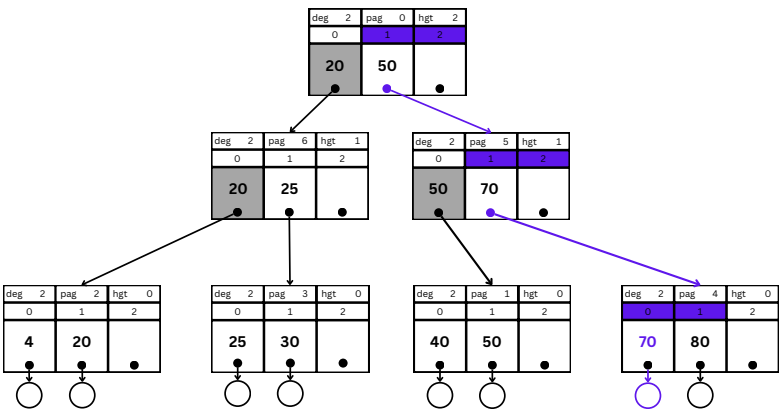
B-Tree Operations—Search (Example)

```
12    while( upper > lower +1 ) {
```

```
19    if( current_node->height > 0)
```

```
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    }
```

> Search 1; Step 10;
> tree=(*pag 0); query_key=70;
> lower=0; upper=1;
> current_node=(*pag 4);
> object=(*70);



B-Tree Operations—Insert Value

- 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 thoose of the splited nodes.
- Then, the median key of the splited 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 it's height.**

B-Tree Operations—Insert Value

```
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 Operations—Insert Value

```
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 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] = current_node->key[i-1];
47          current_node->next[i] = current_node->next[i-1];
48          i -= 1;
49        }
50
51        current_node->key[i] = insert_key;
52        current_node->next[i] = insert_pt;
53        current_node->degree +=1;
```

B-Tree Operations—Insert Value

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

B-Tree Operations—Insert Value

```
81            current_node->key[i+1] =
82              current_node->key[i];
83            i -=1;
84          } else {
85            current_node->next[i+1] =
86              insert_pt;
87            current_node->key[i+1] =
88              insert_key;
89            insert_done = 1;
90          }
91        }
92        /*finished insertion */
93
94        current_node->degree = ((2 * ALPHA) - 1)+1 - ((((2 * ALPHA) - 1)+1)/2);
95        new_node->degree = (((2 * ALPHA) - 1)+1)/2;
96        new_node->height = current_node->height;
97        /* split nodes complete, now insert the new node above */
98        insert_pt = new_node;
99        insert_key = new_node->key[0];
100       if( ! stack_empty() ) {
101         /* not at root; move one level up*/
102         current_node = pop();
103       } else {
104         /* splitting root: needs copy to keep root address*/
105         new_node = get_node();
106         for(i = 0; i < current_node->degree; i++) {
107           new_node->next[i] =
```

B-Tree Operations—Insert Value

```
108             current_node->next[i];
109           new_node->key[i] =
110             current_node->key[i];
111         }
112         new_node->height =
113           current_node->height;
114         new_node->degree =
115           current_node->degree;
116         current_node->height += 1;
117         current_node->degree = 2;
118         current_node->next[0] = new_node;
119         current_node->next[1] = insert_pt;
120         current_node->key[1] = insert_key;
121         finished =1;
122       } /* end splitting root */
123     } /* end node splitting */
124   } /* end of rebalancing */
125   remove_stack();
126   return( 0 );
127 }
```

B-Tree Operations—Insert Value

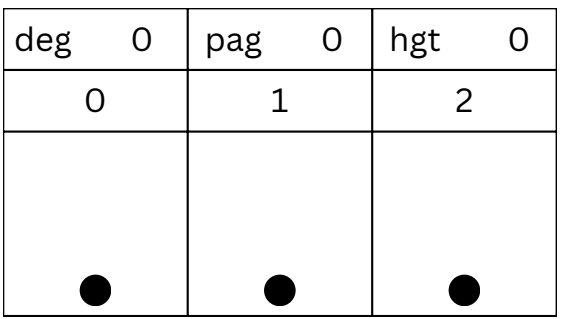
deg 0	pag 0	hgt 0
0	1	2
●	●	●

> Now, lets create a new empty tree and insert a lot of elements in a $t\,(2,0)$ B-Tree. With the bounds α and $2\alpha - 1$.

B-Tree Operations—Insert (Example)

```
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   }
```

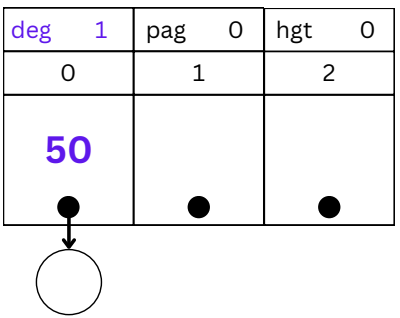
> Insert 1; Step 1;
> tree=(*pag 0); new_key=50; new_object=(*50);
> finished; insert_key;
> current_node=(*pag 0);



B-Tree Operations—Insert (Example)

```
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   }
```

> Insert 1; Step 2;
> tree=(*pag 0); new_key=50; new_object=(*50);
> finished; insert_key;
> current_node=(*pag 0);

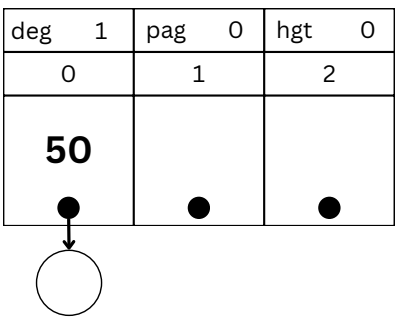


B-Tree Operations—Insert (Example)

```
6   if( tree->height == 0 && tree->degree == 0 ) {
```

```
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];
27    }
```

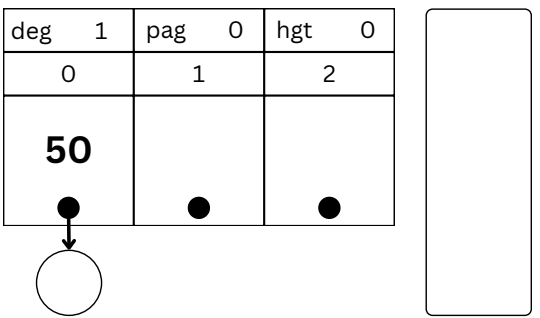
> Insert 2; Step 1;
> tree=(*pag 0); new_key=20; new_object=(*20);
> finished; insert_key;
> current_node=(*pag 0);



B-Tree Operations—Insert (Example)

```
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 leaf starts at 0 */
```

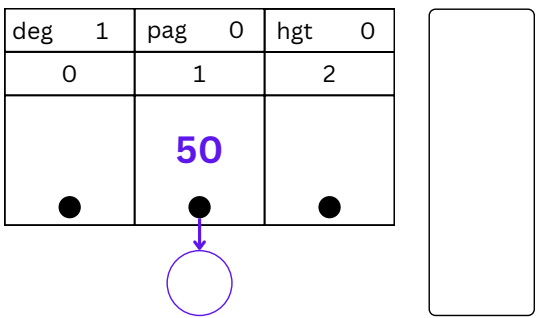
> Insert 2; Step 2;
> tree=(*pag 0); new_key=20; new_object=(*20);
> finished=0; insert_key=20; insert_pt=(*20);
> current_node=(*pag 0);
> i; start=0;



B-Tree Operations—Insert (Example)

```
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] = current_node->key[i-1];
47          current_node->next[i] = current_node->next[i-1];
48          i -= 1;
49        }
```

> Insert 2; Step 3;
> tree=(*pag 0); new_key=20; new_object=(*20);
> finished=0; insert_key=20; insert_pt=(*20);
> current_node=(*pag 0);
> i=1 → 0; start=0;

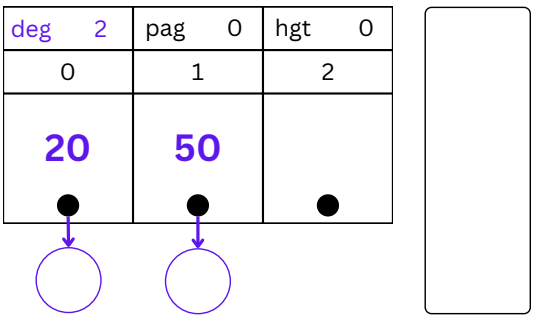


B-Tree Operations—Insert (Example)

```
51        current_node->key[i] = insert_key;
52        current_node->next[i] = insert_pt;
53        current_node->degree +=1;
54        finished = 1;
55      } /* end insert in non-full node */
```

```
125   remove_stack();
126   return( 0 );
127 }
```

> Insert 2; Step 4;
> tree=(*pag 0); new_key=20; new_object=(*20);
> finished=1; insert_key=20; insert_pt=(*20);
> current_node=(*pag 0);
> i=0; start=0;



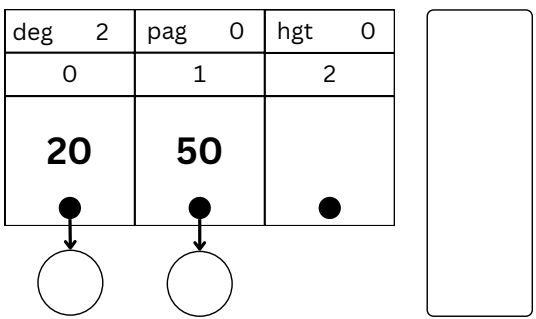
B-Tree Operations—Insert (Example)

```
6   if( tree->height == 0 && tree->degree == 0 ) {
```

```
13  create_stack();
14  while( current_node->height > 0 ) {
```

```
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 leaf starts at 0 */
```

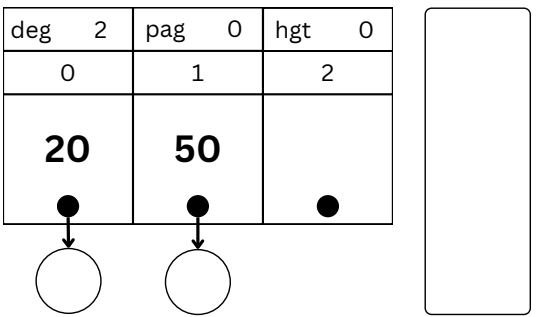
> Insert 3; Step 1;
> tree=(*pag 0); new_key=70; new_object=(*70);
> finished=0; insert_key=70; insert_pt=(*70);
> current_node=(*pag 0);
> i; start=0;



B-Tree Operations—Insert (Example)

```
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] = current_node->key[i-1];
47          current_node->next[i] = current_node->next[i-1];
48          i -= 1;
49        }
```

> Insert 3; Step 2;
> tree=(*pag 0); new_key=70; new_object=(*70);
> finished=0; insert_key=70; insert_pt=(*70);
> current_node=(*pag 0);
> i=2; start=0;

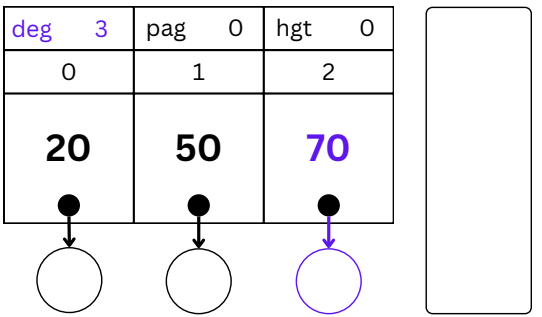


B-Tree Operations—Insert (Example)

```
51        current_node->key[i] = insert_key;
52        current_node->next[i] = insert_pt;
53        current_node->degree +=1;
54        finished = 1;
55      } /* end insert in non-full node */
```

```
125   remove_stack();
126   return( 0 );
127 }
```

> Insert 3; Step 3;
> tree=(*pag 0); new_key=70; new_object=(*70);
> finished=1; insert_key=70; insert_pt=(*70);
> current_node=(*pag 0);
> i=2; start=0;



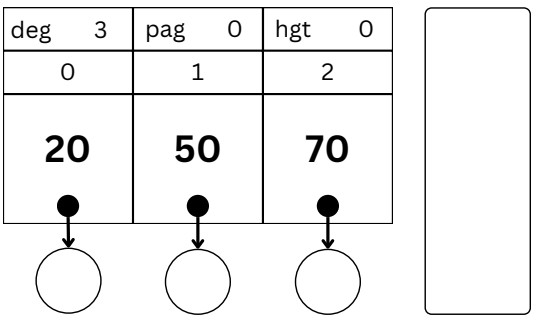
B-Tree Operations—Insert (Example)

```
6   if( tree->height == 0 && tree->degree == 0 ) {
```

```
13  create_stack();
14  while( current_node->height > 0 ) {
```

```
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 leaf starts at 0 */
```

> Insert 4; Step 1;
> tree=(*pag 0); new_key=25; new_object=(*25);
> finished=0; insert_key=25; insert_pt=(*25);
> current_node=(*pag 0);
> i; start=0;

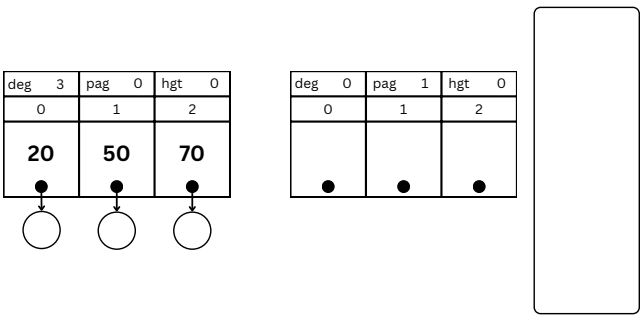


B-Tree Operations—Insert (Example)

```
42    if( current_node->degree < (2 * ALPHA) - 1) {
```

```
56    else {
57      /* node is full, have to split the node*/
58      tree_node_t *new_node;
59      int j, insert_done = 0;
60      new_node = get_node();
61      i = ((2 * ALPHA) - 1)-1;
62      j = (((2 * ALPHA) - 1)-1)/2;
```

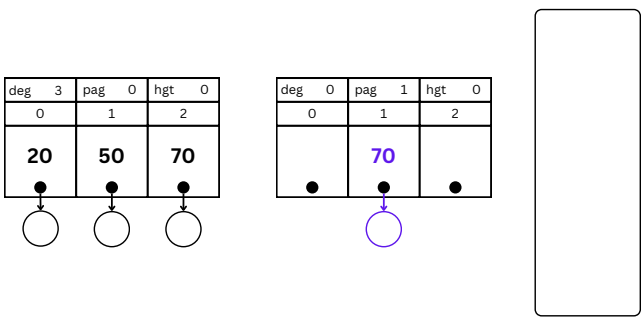
> Insert 4; Step 2;
> tree=(*pag 0); new_key=25; new_object=(*25);
> insert_key=25; insert_pt=(*25);
> finished=0; insert_done=0;
> current_node=(*pag 0); new_node=(*pag 1);
> start=0;
> i=2; j=1;



B-Tree Operations—Insert (Example)

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

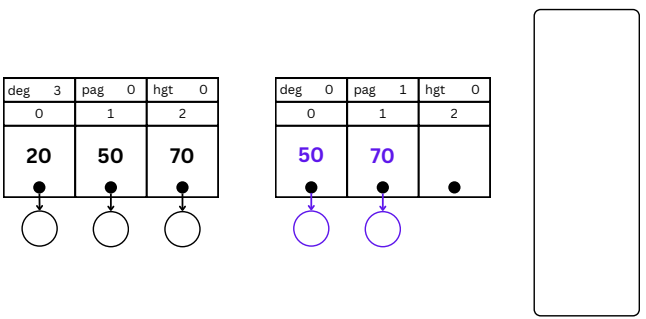
> Insert 4; Step 3;
> tree=(*pag 0); new_key=25; new_object=(*25);
> insert_key=25; insert_pt=(*25);
> finished=0; insert_done=0;
> current_node=(*pag 0); new_node=(*pag 1);
> start=0;
> i=2 → 1; j=1 → 0;



B-Tree Operations—Insert (Example)

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

> Insert 4; Step 4;
> tree=(*pag 0); new_key=25; new_object=(*25);
> insert_key=25; insert_pt=(*25);
> finished=0; insert_done=0;
> current_node=(*pag 0); new_node=(*pag 1);
> start=0;
> i=1 → 0; j=0 → -1;



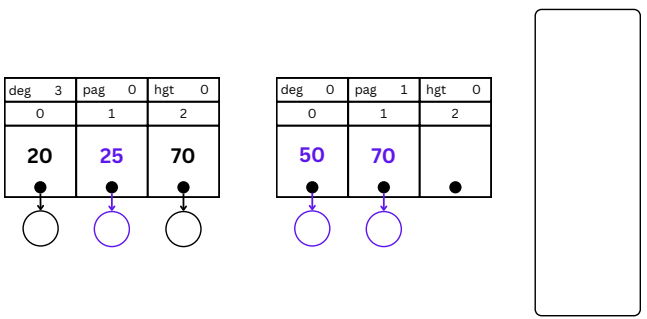
B-Tree Operations—Insert (Example)

```
63      while( j >= 0 ) {
```

```
77      while( !insert_done) {
78        if( insert_key < current_node->key[i] && i >= start
            ↪  ) {
```

```
84        } else {
85          current_node->next[i+1] =
86            insert_pt;
87          current_node->key[i+1] =
88            insert_key;
89          insert_done = 1;
90        }
91      }
```

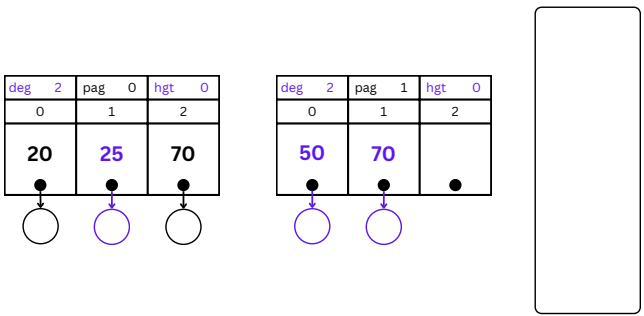
> Insert 4; Step 5;
> tree=(*pag 0); new_key=25; new_object=(*25);
> insert_key=25; insert_pt=(*25);
> finished=0; insert_done=0 →1;
> current_node=(*pag 0); new_node=(*pag 1);
> start=0;
> i=0; j=-1;



B-Tree Operations—Insert (Example)

```
94      current_node->degree = ((2 * ALPHA) - 1)+1 - ((((2 *
        ↪  ALPHA) - 1)+1)/2);
95      new_node->degree = (((2 * ALPHA) - 1)+1)/2;
96      new_node->height = current_node->height;
97      /* split nodes complete, now insert the new node above
        ↪  */
98      insert_pt = new_node;
99      insert_key = new_node->key[0];
```

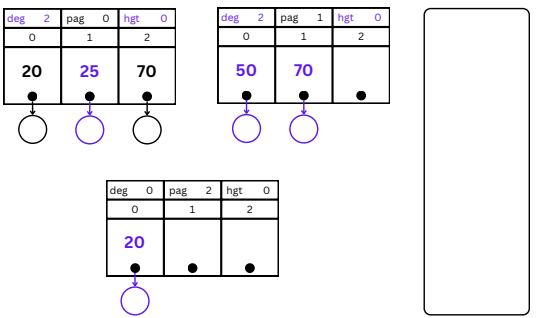
> Insert 4; Step 6;
> tree=(*pag 0); new_key=25; new_object=(*25);
> insert_key=50; insert_pt=(*pag 1);
> finished=0; insert_done=1;
> current_node=(*pag 0); new_node=(*pag 1);
> start=0;
> i=0; j=-1;



B-Tree Operations—Insert (Example)

```
100       if( ! stack_empty() ) {

103       } else {
104         /* splitting root: needs copy to keep root address*/
105         new_node = get_node();
106         for(i = 0; i < current_node->degree; i++) {
107           new_node->next[i] =
108             current_node->next[i];
109           new_node->key[i] =
110             current_node->key[i];
111         }
```

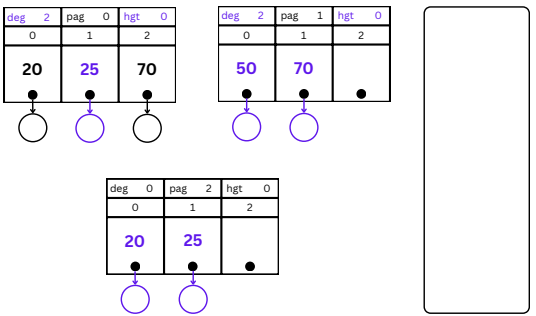
> Insert 4; Step 7;
> tree=(*pag 0); new_key=25; new_object=(*25);
> insert_key=50; insert_pt=(*pag 1);
> finished=0; insert_done=1;
> current_node=(*pag 0); new_node=(*pag 2);
> start=0;
> i=0 → 1; j=-1;



B-Tree Operations—Insert (Example)

```
103       } else {
104         /* splitting root: needs copy to keep root address*/
105         new_node = get_node();
106         for(i = 0; i < current_node->degree; i++) {
107           new_node->next[i] =
108             current_node->next[i];
109           new_node->key[i] =
110             current_node->key[i];
111         }
```

> Insert 4; Step 8;
> tree=(*pag 0); new_key=25; new_object=(*25);
> insert_key=50; insert_pt=(*pag 1);
> finished=0; insert_done=1;
> current_node=(*pag 0); new_node=(*pag 2);
> start=0;
> i=1 → 2; j=-1;

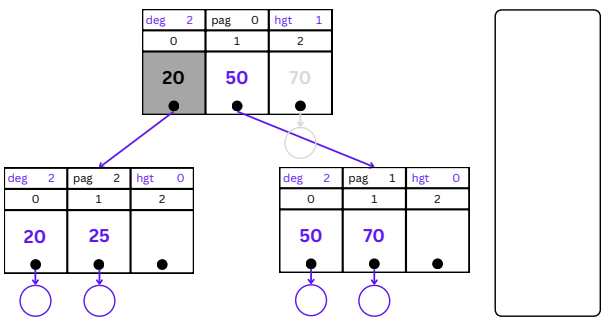


B-Tree Operations—Insert (Example)

```
112         new_node->height =
113           current_node->height;
114         new_node->degree =
115           current_node->degree;
116         current_node->height += 1;
117         current_node->degree = 2;
118         current_node->next[0] = new_node;
119         current_node->next[1] = insert_pt;
120         current_node->key[1] = insert_key;
121         finished =1;
122       } /* end splitting root */
```

```
125   remove_stack();
126   return( 0 );
127 }
```

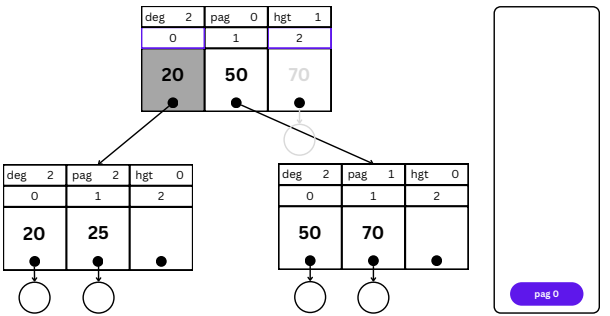
> Insert 4; Step 9;
> tree=(*pag 0); new_key=25; new_object=(*25);
> insert_key=50; insert_pt=(*pag 1);
> finished=1; insert_done=1;
> current_node=(*pag 0); new_node=(*pag 2);
> start=0;
> i=2; j=-1;



B-Tree Operations—Insert (Example)

```
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;
```

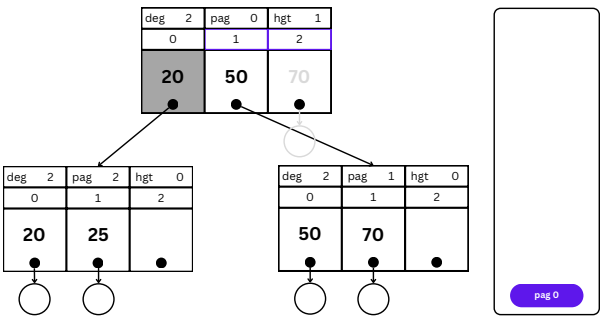
> Insert 5; Step 1;
> tree=(*pag 0); new_key=80; new_object=(*80);
> finished; lower=0; uppper=2;
> current_node=(*pag 0);



B-Tree Operations—Insert (Example)

```
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];
27    }
```

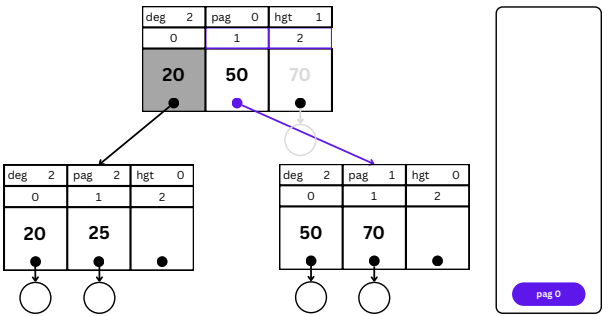
- Insert 5; Step 2;
- tree=(*pag 0); new_key=80; new_object=(*80);
- finished; lower=0 →1; uppper=2;
- current_node=(*pag 0);



B-Tree Operations—Insert (Example)

```
14  while( current_node->height > 0 ) {

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];
27  }
```

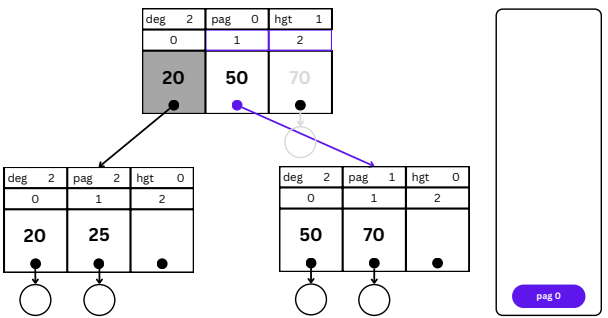
> Insert 5; Step 3;
> tree=(*pag 0); new_key=80; new_object=(*80);
> finished; lower=1; uppper=2;
> current_node=(*pag 0) → (*pag 1);



B-Tree Operations—Insert (Example)

```
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 leaf starts at 0 */
```

> Insert 5; Step 4;
> tree=(*pag 0); new_key=80; new_object=(*80);
> insert_key=80; insert_pt=(*80);
> finished=0; lower=1; uppper= 2;
> current_node=(*pag 1);
> i; start=0;



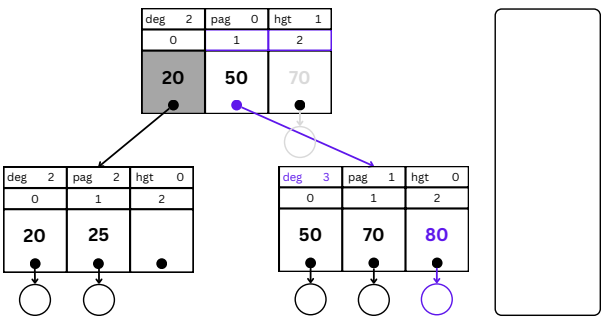
B-Tree Operations—Insert (Example)

```
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)) {
```

```
51        current_node->key[i] = insert_key;
52        current_node->next[i] = insert_pt;
53        current_node->degree +=1;
54        finished = 1;
55      } /* end insert in non-full node */
```

```
125   remove_stack();
126   return( 0 );
127 }
```

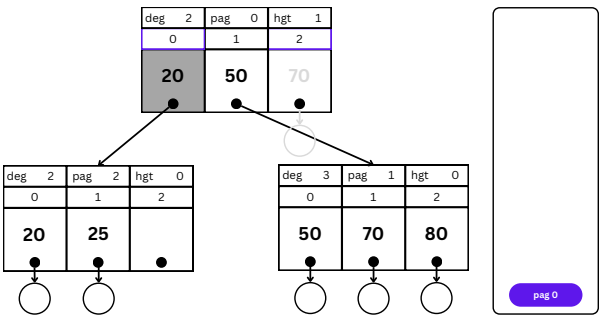
> Insert 5; Step 5;
> tree=(*pag 0); new_key=80; new_object=(*80);
> insert_key=80; insert_pt=(*80);
> finished=1; lower=1; uppper= 2;
> current_node=(*pag 1);
> i=2; start=0;



B-Tree Operations—Insert (Example)

```
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;
```

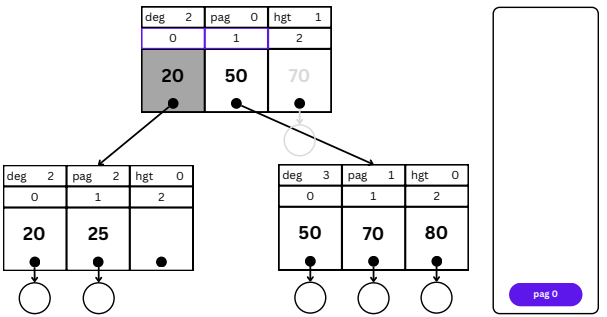
> Insert 6; Step 1;
> tree=(*pag 0); new_key=4; new_object=(*4);
> finished; lower=0; uppper=2;
> current_node=(*pag 0);



B-Tree Operations—Insert (Example)

```
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];
27    }
```

- Insert 6; Step 2;
- tree=(*pag 0); new_key=4; new_object=(*4);
- finished; lower=0; uppper= 2 → 1;
- current_node=(*pag 0);

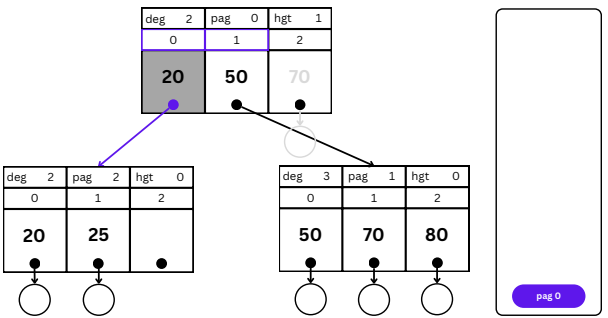


B-Tree Operations—Insert (Example)

```
14  while( current_node->height > 0 ) {
```

```
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];
27  }
```

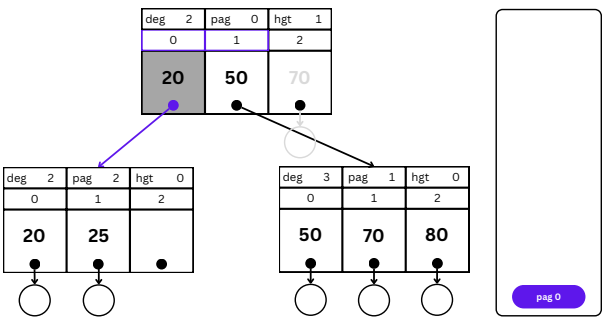
> Insert 6; Step 3;
> tree=(*pag 0); new_key=4; new_object=(*4);
> finished; lower=0; uppper=1;
> current_node=(*pag 0) → (*pag 2);



B-Tree Operations—Insert (Example)

```
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;
```

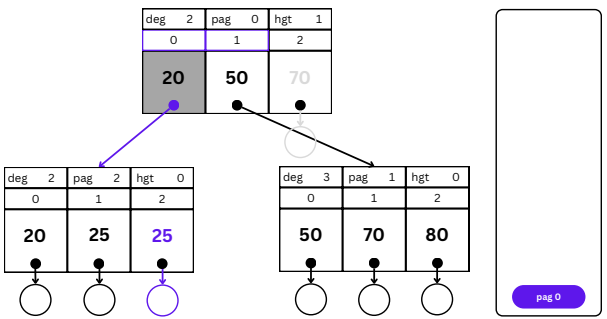
> Insert 6; Step 4;
> tree=(*pag 0); new_key=4; new_object=(*4);
> insert_key=4; insert_pt=(*4);
> finished=0; lower=0; uppper= 1;
> current_node=(*pag 2);
> i; start=0;



B-Tree Operations—Insert (Example)

```
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] = current_node->key[i-1];
47          current_node->next[i] = current_node->next[i-1];
48          i -= 1;
49        }
```

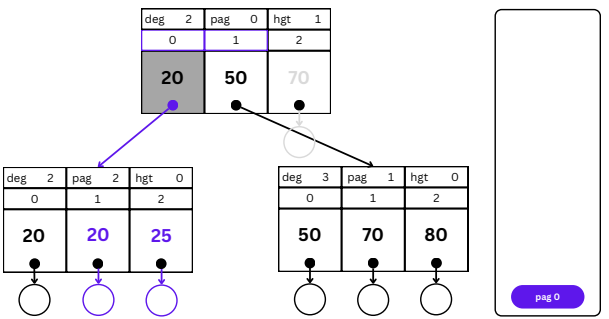
> Insert 6; Step 5;
> tree=(*pag 0); new_key=4; new_object=(*4);
> insert_key=4; insert_pt=(*4);
> finished=0; lower=0; uppper= 1;
> current_node=(*pag 2);
> i=2 →1; start=0;



B-Tree Operations—Insert (Example)

```
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] = current_node->key[i-1];
47          current_node->next[i] = current_node->next[i-1];
48          i -= 1;
49        }
```

> Insert 6; Step 6;
> tree=(*pag 0); new_key=4; new_object=(*4);
> insert_key=4; insert_pt=(*4);
> finished=0; lower=0; uppper= 1;
> current_node=(*pag 2);
> i=1 →0; start=0;

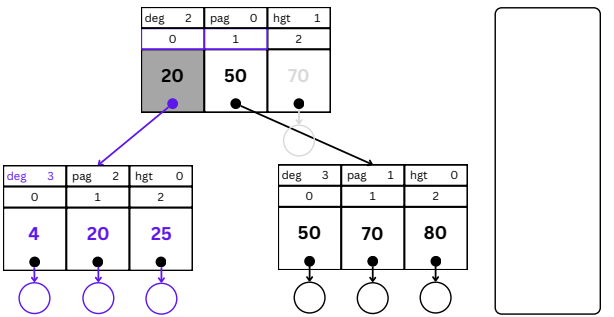


B-Tree Operations—Insert (Example)

```
51        current_node->key[i] = insert_key;
52        current_node->next[i] = insert_pt;
53        current_node->degree +=1;
54        finished = 1;
```

```
125     remove_stack();
126     return( 0 );
127   }
```

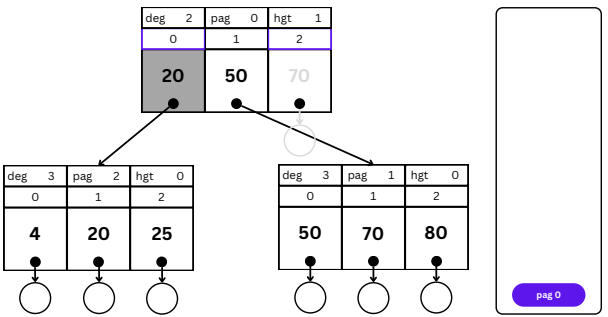
> Insert 6; Step 7;
> tree=(*pag 0); new_key=4; new_object=(*4);
> insert_key=4; insert_pt=(*4);
> finished=1; lower=0; uppper= 1;
> current_node=(*pag 2);
> i=0; start=0;



B-Tree Operations—Insert (Example)

```
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;
```

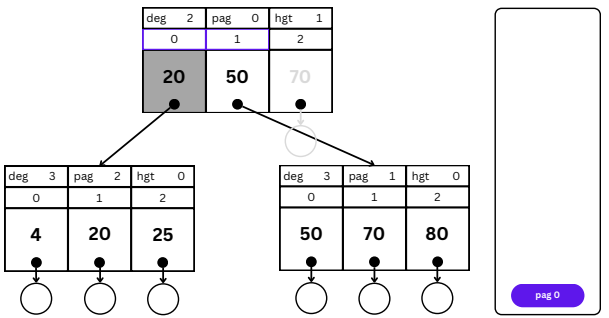
> Insert 7; Step 1;
> tree=(*pag 0); new_key=30; new_object=(*30);
> finished; lower=0; uppper=2;
> current_node=(*pag 0);



B-Tree Operations—Insert (Example)

```
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];
27    }
```

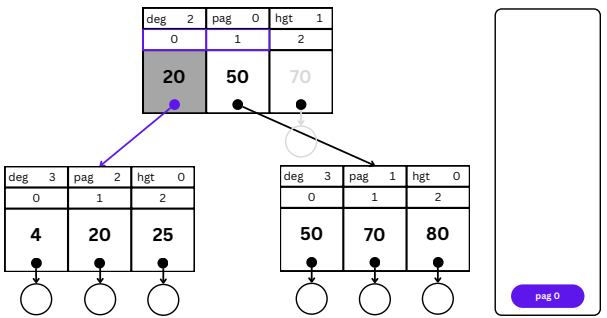
- Insert 7; Step 2;
- tree=(*pag 0); new_key=30; new_object=(*30);
- finished; lower=0; uppper=2→1;
- current_node=(*pag 0);



B-Tree Operations—Insert (Example)

```
14  while( current_node->height > 0 ) {

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];
27  }
```

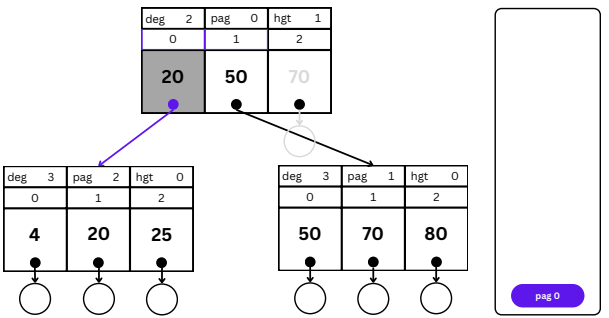
> Insert 7; Step 3;
> tree=(*pag 0); new_key=30; new_object=(*30);
> finished; lower=0; uppper=1;
> current_node=(*pag 0)→(*pag 2);



B-Tree Operations—Insert (Example)

```
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;
```

> Insert 7; Step 4;
> tree=(*pag 0); new_key=30; new_object=(*30);
> insert_key=30; insert_pt=(*30);
> finished=0; lower=0; uppper=1;
> current_node=(*pag 2);
> start=0;
> i;

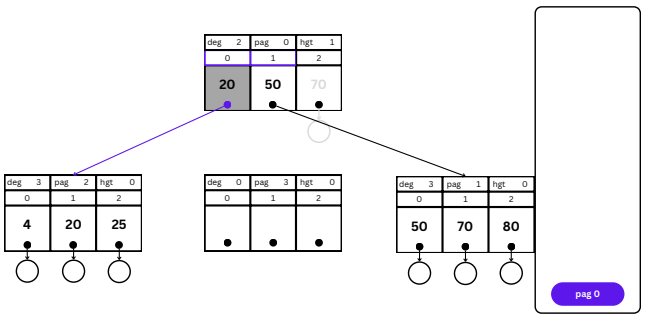


B-Tree Operations—Insert (Example)

```
42    if( current_node->degree < (2 * ALPHA) - 1) {
```

```
56    else {
57      /* node is full, have to split the node*/
58      tree_node_t *new_node;
59      int j, insert_done = 0;
60      new_node = get_node();
61      i = ((2 * ALPHA) - 1)-1;
62      j = (((2 * ALPHA) - 1)-1)/2;
```

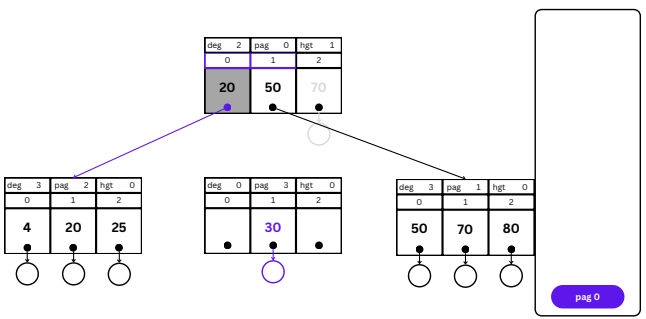
> Insert 7; Step 5;
> tree=(*pag 0); new_key=30; new_object=(*30);
> insert_key=30; insert_pt=(*30);
> finished=0; lower=0; uppper=1;
> current_node=(*pag 2); new_node=(*pag 3);
> start=0; insert_done=0;
> i=2; j=1;



B-Tree Operations—Insert (Example)

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

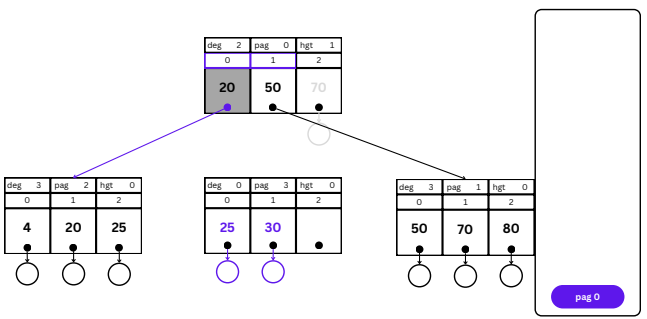
> Insert 7; Step 6;
> tree=(*pag 0); new_key=30; new_object=(*30);
> insert_key=30; insert_pt=(*30);
> finished=0; lower=0; uppper=1;
> current_node=(*pag 2); new_node=(*pag 3);
> start=0; insert_done=1;
> i=2; j=1 → 0;



B-Tree Operations—Insert (Example)

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

> Insert 7; Step 7;
> tree=(*pag 0); new_key=30; new_object=(*30);
> insert_key=30; insert_pt=(*30);
> finished=0; lower=0; uppper=1;
> current_node=(*pag 2); new_node=(*pag 3);
> start=0; insert_done=1;
> i=2 → 1; j=0 → -1;



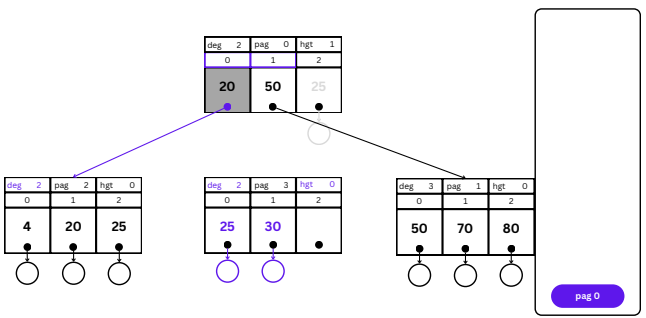
B-Tree Operations—Insert (Example)

```
63        while( j >= 0 ) {
```

```
77        while( !insert_done) {
```

```
94        current_node->degree = ((2 * ALPHA) - 1)+1 - ((((2 *
          ↪  ALPHA) - 1)+1)/2);
95        new_node->degree = (((2 * ALPHA) - 1)+1)/2;
96        new_node->height = current_node->height;
97        /* split nodes complete, now insert the new node above
          ↪  */
98        insert_pt = new_node;
99        insert_key = new_node->key[0];
```

> Insert 7; Step 8;
> tree=(*pag 0); new_key=30; new_object=(*30);
> insert_key=25; insert_pt=(*pag 3);
> finished=0; lower=0; uppper=1;
> current_node=(*pag 2); new_node=(*pag 3);
> start=0; insert_done=1;
> i=1; j=-1;

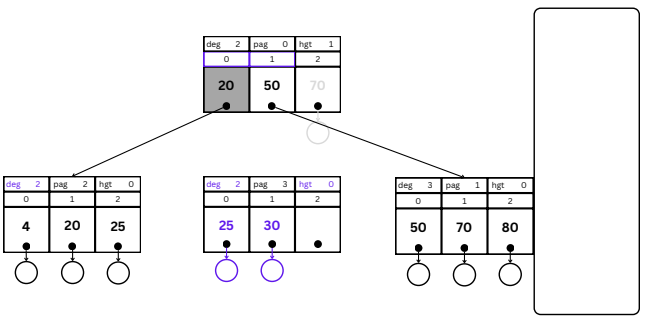


B-Tree Operations—Insert (Example)

```
100       if( ! stack_empty() ) {
101         /* not at root; move one level up*/
102         current_node = pop();
103       } else {
```

```
122       } /* end splitting root */
123     } /* end node splitting */
124   } /* end of rebalancing */
```

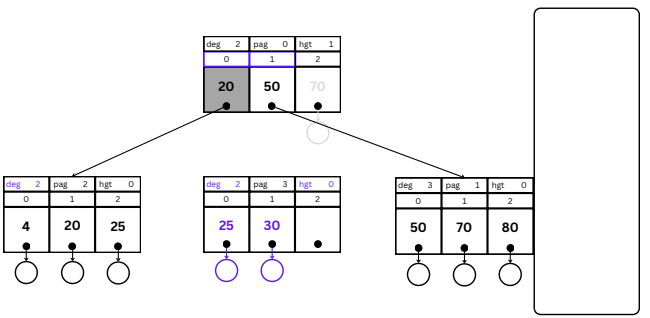
> Insert 7; Step 9;
> tree=(*pag 0); new_key=30; new_object=(*30);
> insert_key=25; insert_pt=(*pag 3);
> finished=0; lower=0; uppper=1;
> current_node=(*pag 0); new_node=(*pag 3);
> start=0; insert_done=1;
> i=1; j=-1;



B-Tree Operations—Insert (Example)

```
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 leaf starts at 0 */
```

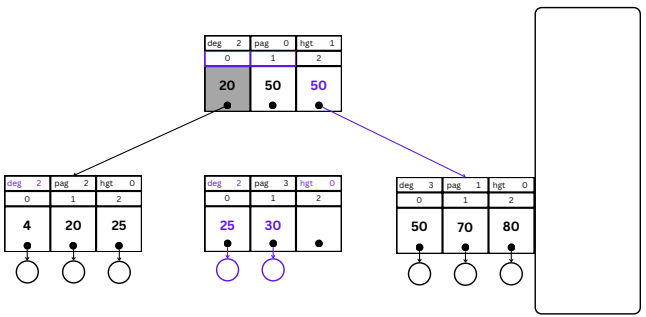
> Insert 7; Step 10;
> tree=(*pag 0); new_key=30; new_object=(*30);
> insert_key=25; insert_pt=(*pag 3);
> finished=0; lower=0; uppper=1;
> current_node=(*pag 0);
> start=1;
> i;



B-Tree Operations—Insert (Example)

```
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] = current_node->key[i-1];
47          current_node->next[i] = current_node->next[i-1];
48          i -= 1;
49        }
```

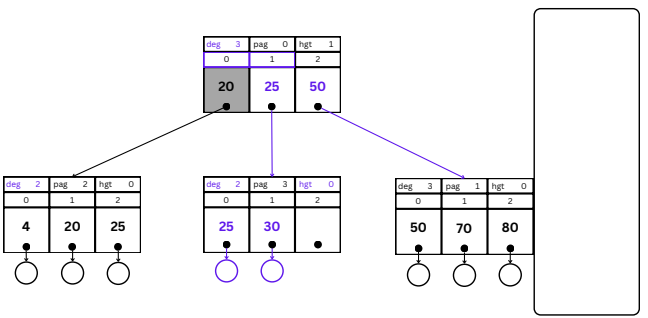
> Insert 7; Step 11;
> tree=(*pag 0); new_key=30; new_object=(*30);
> insert_key=25; insert_pt=(*pag 3);
> finished=0; lower=0; uppper=1;
> current_node=(*pag 0);
> start=1;
> i=2 → 1;



B-Tree Operations—Insert (Example)

```
45      while( (i > start) && (current_node->key[i-1] >
        ↪  insert_key)) {

51      current_node->key[i] = insert_key;
52      current_node->next[i] = insert_pt;
53      current_node->degree +=1;
54      finished = 1;

125   remove_stack();
126   return( 0 );
127 }
```

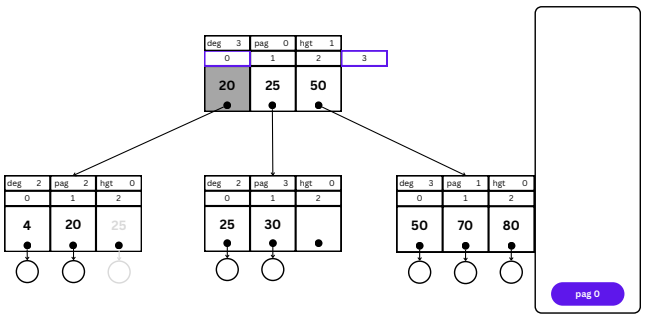
> Insert 7; Step 12;
> tree=(*pag 0); new_key=30; new_object=(*30);
> insert_key=25; insert_pt=(*pag 3);
> finished=1; lower=0; uppper=1;
> current_node=(*pag 0);
> start=1;
> i=1;



B-Tree Operations—Insert (Example)

```
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;
```

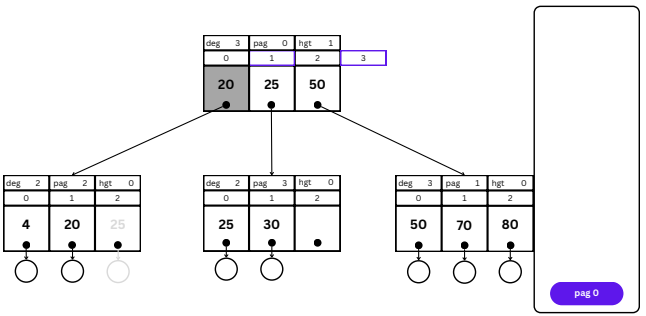
> Insert 8; Step 1;
> tree=(*pag 0); new_key=40; new_object=(*40);
> finished; lower=0; uppper=3;
> current_node=(*pag 0);



B-Tree Operations—Insert (Example)

```
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];
27    }
```

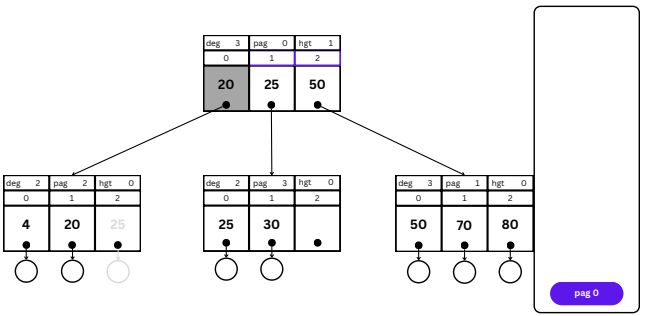
> Insert 8; Step 2;
> tree=(*pag 0); new_key=40; new_object=(*40);
> finished; lower=1; uppper=3;
> current_node=(*pag 0);



B-Tree Operations—Insert (Example)

```
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];
27    }
```

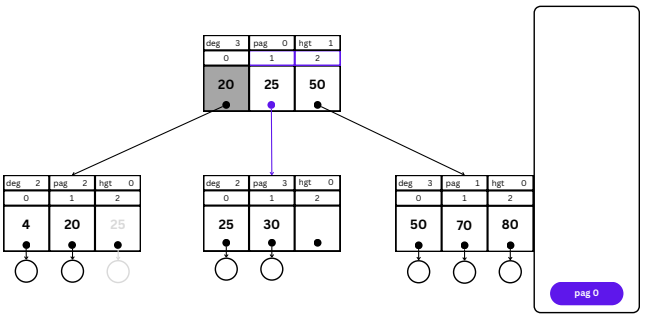
> Insert 8; Step 3;
> tree=(*pag 0); new_key=40; new_object=(*40);
> finished; lower=1; uppper=2;
> current_node=(*pag 0);



B-Tree Operations—Insert (Example)

```
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];
27    }
```

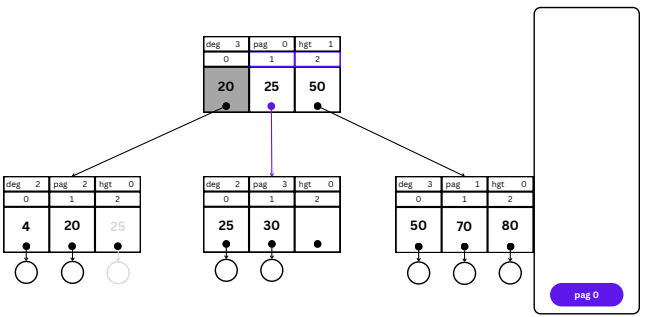
> Insert 8; Step 4;
> tree=(*pag 0); new_key=40; new_object=(*40);
> finished; lower=1; uppper=2;
> current_node=(*pag 0) → (*pag 3);



B-Tree Operations—Insert (Example)

```
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;
```

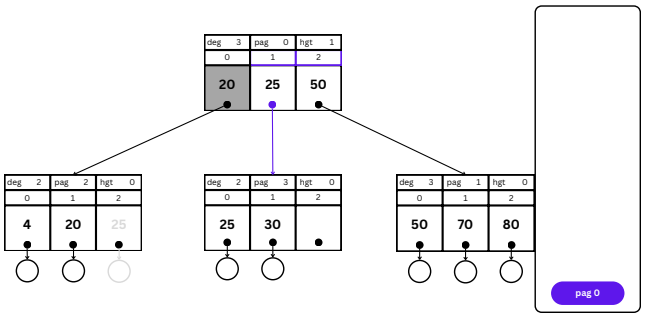
> Insert 8; Step 5;
> tree=(*pag 0); new_key=40; new_object=(*40);
> finished=0; insert_key=40; insert_pt=(*40);
> current_node=(*pag 3);
> start=0;



B-Tree Operations—Insert (Example)

```
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] = current_node->key[i-1];
47          current_node->next[i] = current_node->next[i-1];
48          i -= 1;
```

> Insert 8; Step 6;
> tree=(*pag 0); new_key=40; new_object=(*40);
> finished=0; insert_key=40; insert_pt=(*40);
> current_node=(*pag 3);
> start=0; i=2;



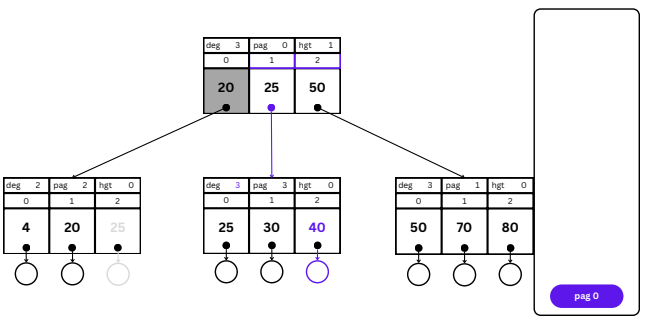
B-Tree Operations—Insert (Example)

```
33    while( !finished ){
```

```
51        current_node->key[i] = insert_key;
52        current_node->next[i] = insert_pt;
53        current_node->degree +=1;
54        finished = 1;
```

```
124     } /* end of rebalancing */
125     remove_stack();
126     return( 0 );
127   }
```

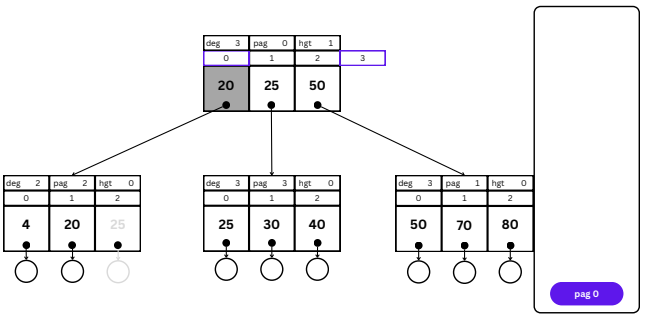
> Insert 8; Step 7;
> tree=(*pag 0); new_key=40; new_object=(*40);
> finished=0; insert_key=40; insert_pt=(*40);
> current_node=(*pag 3);
> start=0; i=2;



B-Tree Operations—Insert (Example)

```
1  int insert(tree_node_t *tree, key_t new_key, object_t
   ↪  *new_object) {

13   create_stack();
14   while( current_node->height > 0 ) {

17     push( current_node );
18     lower = 0;
19     upper = current_node->degree;
```

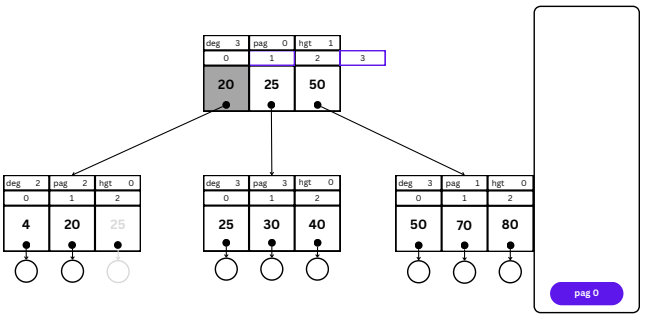
> Insert 9; Step 1;
> tree=(*pag 0); new_key=55; new_object=(*55);
> finished;
> current_node=(*pag 0);
> lower=0; upper=3;



B-Tree Operations—Insert (Example)

```
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      }
```

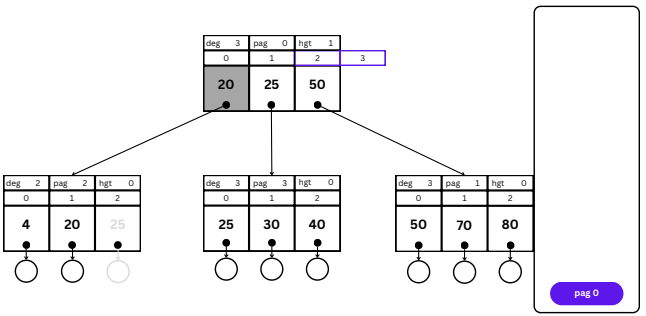
> Insert 9; Step 2;
> tree=(*pag 0); new_key=55; new_object=(*55);
> finished;
> current_node=(*pag 0);
> lower=0 → 1; upper=3;



B-Tree Operations—Insert (Example)

```
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      }
```

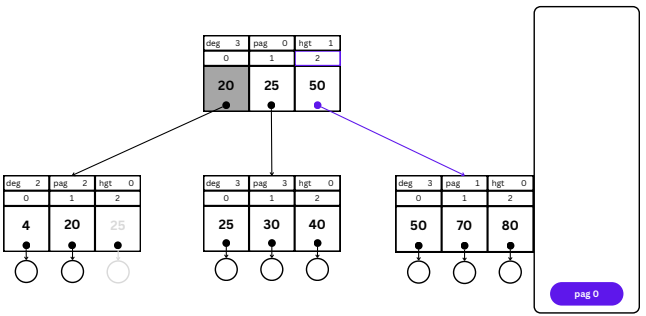
> Insert 9; Step 3;
> tree=(*pag 0); new_key=55; new_object=(*55);
> finished;
> current_node=(*pag 0);
> lower=1 → 2; upper=3;



B-Tree Operations—Insert (Example)

```
14  while( current_node->height > 0 ) {

20    while( upper > lower +1 ) {

25    }
26    current_node = current_node->next[lower];
27  }
```

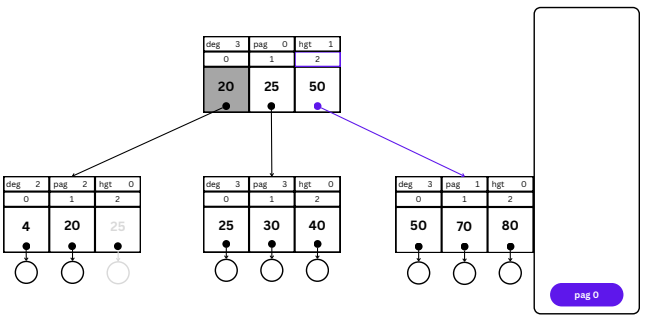
> Insert 9; Step 4;
> tree=(*pag 0); new_key=55; new_object=(*55);
> finished;
> current_node=(*pag 0) → (*pag 1);
> lower=2; upper=3;



B-Tree Operations—Insert (Example)

```
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;
```

> Insert 9; Step 5;
> tree=(*pag 0); new_key=55; new_object=(*55);
> finished=0; insert_pt=(*55); insert_key=55;
> current_node=(*pag 1);
> start=0;
> i;

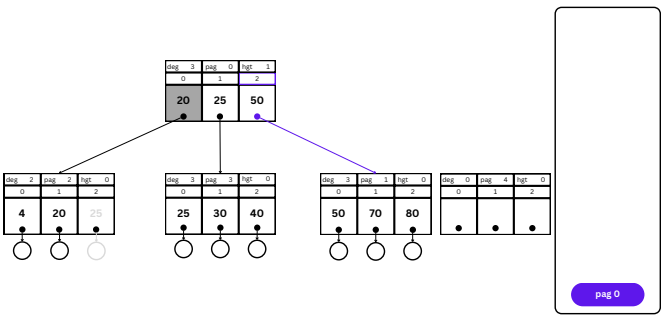


B-Tree Operations—Insert (Example)

```
42    if( current_node->degree < (2 * ALPHA) - 1) {
```

```
56    else {
57      /* node is full, have to split the node*/
58      tree_node_t *new_node;
59      int j, insert_done = 0;
60      new_node = get_node();
61      i = ((2 * ALPHA) - 1)-1;
62      j = (((2 * ALPHA) - 1)-1)/2;
```

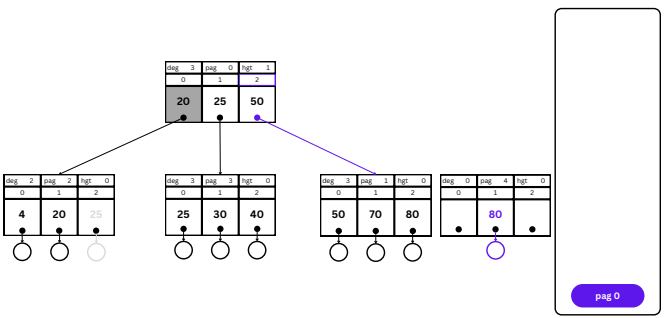
> Insert 9; Step 6;
> tree=(*pag 0); new_key=55; new_object=(*55);
> finished=0; insert_pt=(*55); insert_key=55;
> current_node=(*pag 1); start=0; insert_done=0;
> i=2; j=1;



B-Tree Operations—Insert (Example)

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

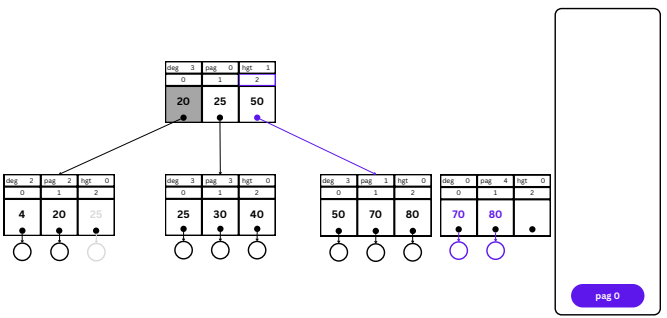
> Insert 9; Step 7;
> tree=(*pag 0); new_key=55; new_object=(*55);
> finished=0; insert_pt=(*55); insert_key=55;
> current_node=(*pag 1); start=0; insert_done=0;
> i=2 → 1; j=1 → 0;



B-Tree Operations—Insert (Example)

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

> Insert 9; Step 8;
> tree=(*pag 0); new_key=55; new_object=(*55);
> finished=0; insert_pt=(*55); insert_key=55;
> current_node=(*pag 1); start=0; insert_done=0;
> i=1 → 0; j=0 → -1;

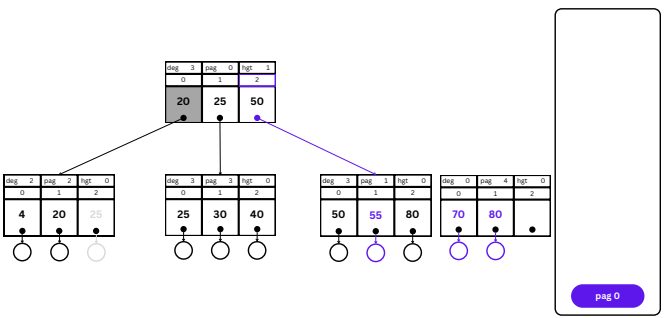


B-Tree Operations—Insert (Example)

```
77        while( !insert_done) {
78          if( insert_key < current_node->key[i] && i >= start
              ↪  ) {
```

```
84          } else {
85            current_node->next[i+1] =
86              insert_pt;
87            current_node->key[i+1] =
88              insert_key;
89            insert_done = 1;
90          }
91        }
```

> Insert 9; Step 9;
> tree=(*pag 0); new_key=55; new_object=(*55);
> finished=0; insert_pt=(*55); insert_key=55;
> current_node=(*pag 1); start=0; insert_done=0 → 1;
> i=0; j=-1;

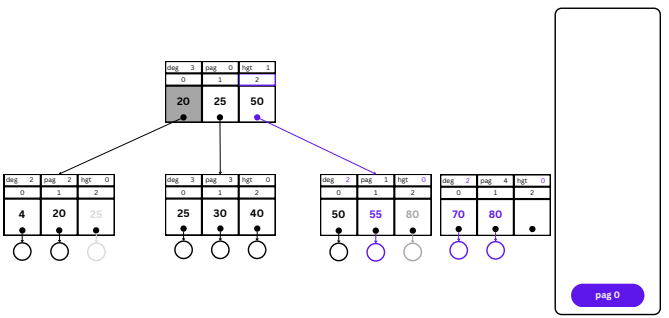


B-Tree Operations—Insert (Example)

```
94        current_node->degree = ((2 * ALPHA) - 1)+1 - ((((2 *
          ↪  ALPHA) - 1)+1)/2);
95        new_node->degree = (((2 * ALPHA) - 1)+1)/2;
96        new_node->height = current_node->height;
97        /* split nodes complete, now insert the new node above
          ↪  */
98        insert_pt = new_node;
99        insert_key = new_node->key[0];
100       if( ! stack_empty() ) {
101         /* not at root; move one level up*/
102         current_node = pop();
```

```
122       } /* end splitting root */
123     } /* end node splitting */
124   } /* end of rebalancing */
```

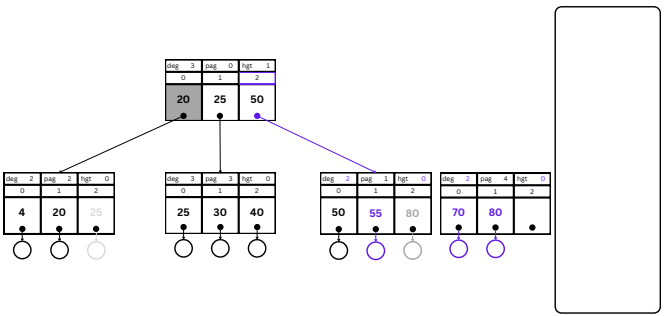
> Insert 9; Step 10;
> tree=(*pag 0); new_key=55; new_object=(*55);
> finished=0; insert_pt=(*pag 4); insert_key=70;
> current_node=(*pag 1) → (*pag 0); start=0; insert_done=1;
> i=0; j=-1;



B-Tree Operations—Insert (Example)

```
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;
```

> Insert 9; Step 11;
> tree=(*pag 0); new_key=55; new_object=(*55);
> finished=0; insert_pt=(*pag 4); insert_key=70;
> current_node=(*pag 0); start=1; insert_done=1;
> i;

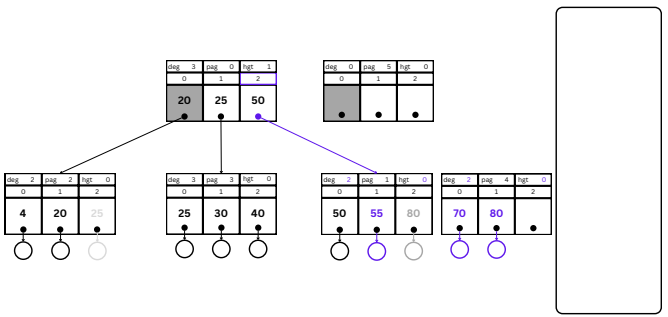


B-Tree Operations—Insert (Example)

```
42    if( current_node->degree < (2 * ALPHA) - 1) {
```

```
56    else {
57      /* node is full, have to split the node*/
58      tree_node_t *new_node;
59      int j, insert_done = 0;
60      new_node = get_node();
61      i = ((2 * ALPHA) - 1)-1;
62      j = (((2 * ALPHA) - 1)-1)/2;
```

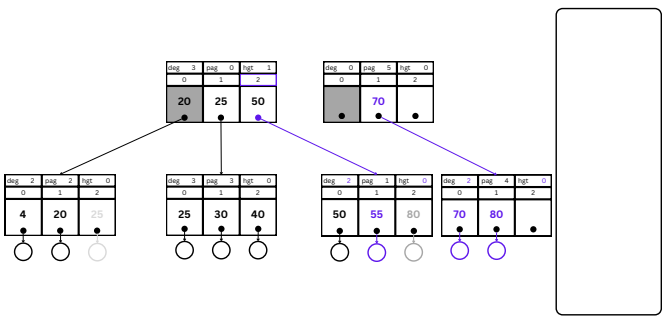
> Insert 9; Step 12;
> tree=(*pag 0); new_key=55; new_object=(*55);
> finished=0; insert_pt=(*pag 4); insert_key=70;
> current_node=(*pag 0); start=1; insert_done=0;
> i=2; j=1;



B-Tree Operations—Insert (Example)

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

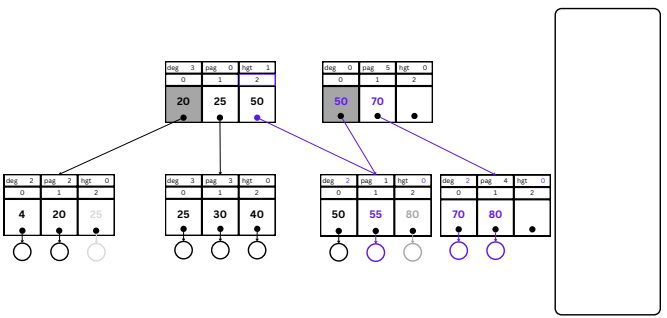
> Insert 9; Step 13;
> tree=(*pag 0); new_key=55; new_object=(*55);
> finished=0; insert_pt=(*pag 4); insert_key=70;
> current_node=(*pag 0); start=1; insert_done=0 → 1;
> i=2; j=1 → 0;



B-Tree Operations—Insert (Example)

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

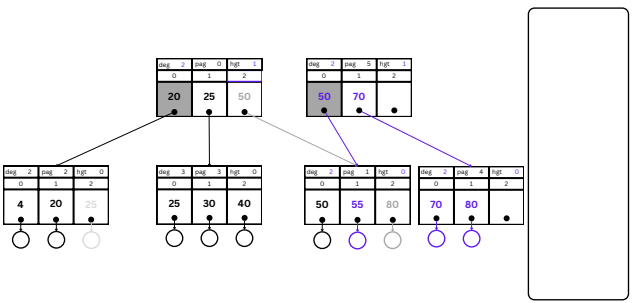
> Insert 9; Step 14;
> tree=(*pag 0); new_key=55; new_object=(*55);
> finished=0; insert_pt=(*pag 4); insert_key=70;
> current_node=(*pag 0); start=1; insert_done=1;
> i=2 → 1; j=0 → -1;



B-Tree Operations—Insert (Example)

```
77      while( !insert_done) {

94      current_node->degree = ((2 * ALPHA) - 1)+1 - ((((2 *
          ↪  ALPHA) - 1)+1)/2);
95      new_node->degree = (((2 * ALPHA) - 1)+1)/2;
96      new_node->height = current_node->height;
97      /* split nodes complete, now insert the new node above
          ↪  */
98      insert_pt = new_node;
99      insert_key = new_node->key[0];
```

> Insert 9; Step 15;
> tree=(*pag 0); new_key=55; new_object=(*55);
> finished=0; insert_pt=(*pag 5); insert_key=50;
> current_node=(*pag 0); start=1; insert_done=1;
> i=1; j=-1;

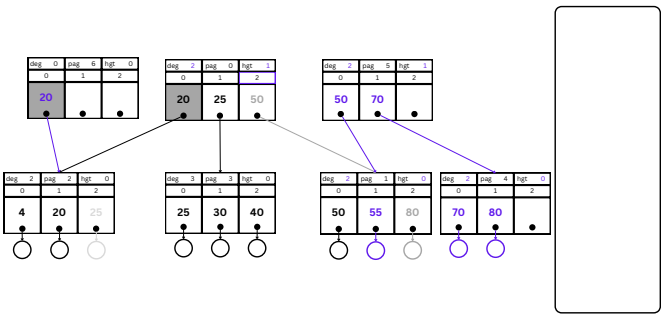


B-Tree Operations—Insert (Example)

```
100       if( ! stack_empty() ) {
```

```
103       } else {
104         /* splitting root: needs copy to keep root address*/
105         new_node = get_node();
106         for(i = 0; i < current_node->degree; i++) {
107           new_node->next[i] =
108             current_node->next[i];
109           new_node->key[i] =
110             current_node->key[i];
111         }
```

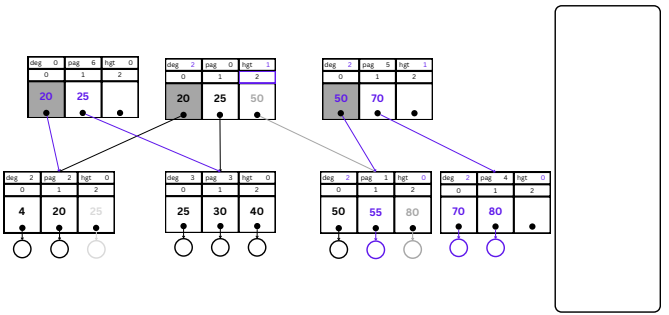
> Insert 9; Step 16;
> tree=(*pag 0); new_key=55; new_object=(*55);
> finished=0; insert_pt=(*pag 5); insert_key=50;
> current_node=(*pag 0); start=1; insert_done=1;
> i=1; j=-1;



B-Tree Operations—Insert (Example)

```
106         for(i = 0; i < current_node->degree; i++) {
107           new_node->next[i] =
108             current_node->next[i];
109           new_node->key[i] =
110             current_node->key[i];
111         }
```

> Insert 9; Step 17;
> tree=(*pag 0); new_key=55; new_object=(*55);
> finished=0; insert_pt=(*pag 5); insert_key=50;
> current_node=(*pag 0); start=1; insert_done=1;
> i=1; j=-1;

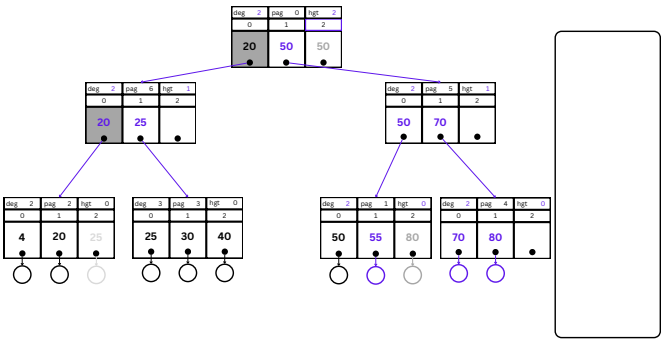


B-Tree Operations—Insert (Example)

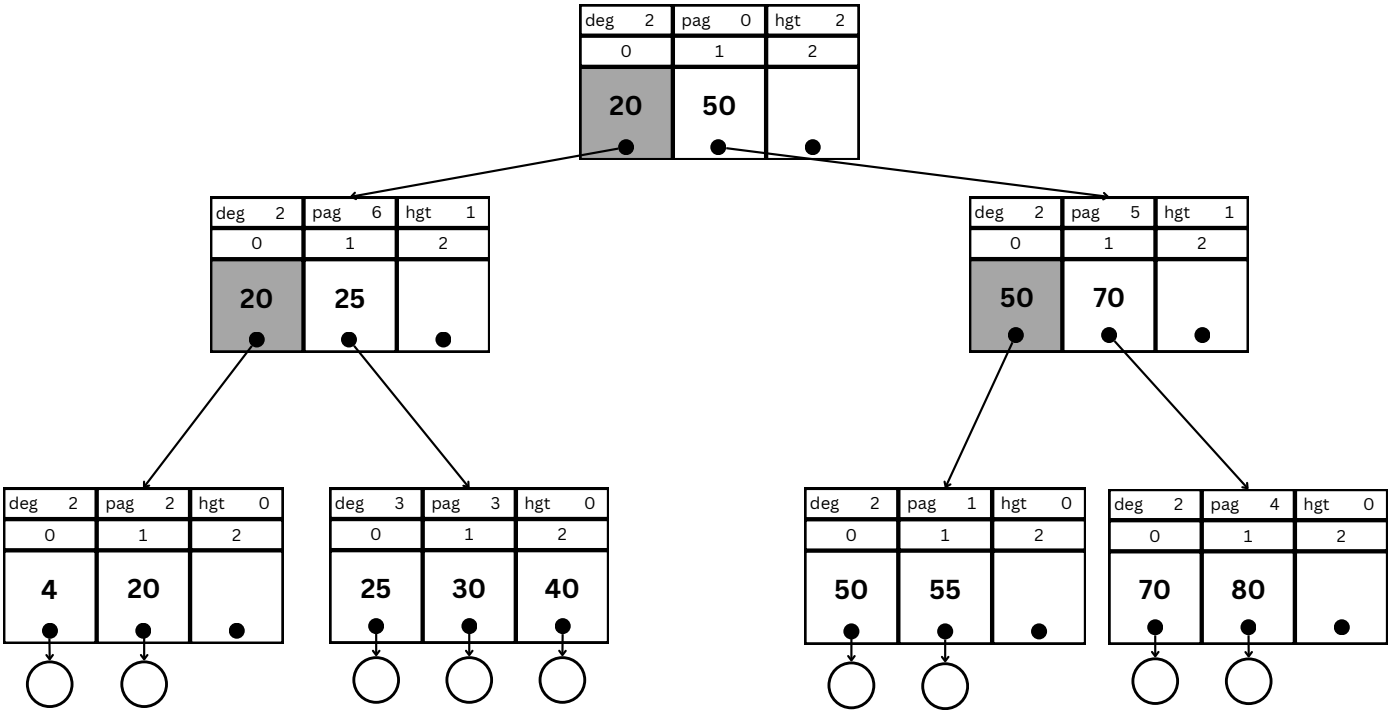
```
112         new_node->height =
113           current_node->height;
114         new_node->degree =
115           current_node->degree;
116         current_node->height += 1;
117         current_node->degree = 2;
118         current_node->next[0] = new_node;
119         current_node->next[1] = insert_pt;
120         current_node->key[1] = insert_key;
121         finished =1;
```

```
125   remove_stack();
126   return( 0 );
127 }
```

> Insert 9; Step 18;
> tree=(*pag 0); new_key=55; new_object=(*55);
> finished=0 → 1; insert_pt=(*pag 5); insert_key=50;
> current_node=(*pag 0); start=1; insert_done=1;
> i=1; j=-1;



B-Tree Operations—Insert (Example)



B-Tree Operations—Delete

> 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 it's 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 α 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 α keys.

> In the case that the neighbor doesn't have α 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 leaft 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 Operations—Delete

```
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
20         push_index_stack( lower );
21         push_node_stack( current );
22         current = current->next[lower];
23     }
24     /* now current is leaf node from which we delete */
25     for ( i=0; i < current->degree ; i++ )
26         if( current->key[i] == delete_key )
27             break;
```

B-Tree Operations—Delete

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

B-Tree Operations—Delete

```
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*/
81                          neighbor = upper->next[curr+1];
82                          if( neighbor->degree > ALPHA ) {
83                              /* sharing possible */
```

B-Tree Operations—Delete

```
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;
108                         current->degree += 1;
109                         finished = 1;
110                     } /* sharing complete */
111                     else {
```

B-Tree Operations—Delete

```
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];
135                             upper->key[i] =
136                                 upper->key[i+1];
137                             i +=1;
138                         }
139                         /* deleted from upper, now propagate up */
```

B-Tree Operations—Delete

```
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];
162                                 }
163                                 else {
164                                     /* on leaf level, take leaf key */
165                                     current->key[1] =
166                                         current->key[0];
167                                     current->key[0] =
```

B-Tree Operations—Delete

```
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] =
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;
```

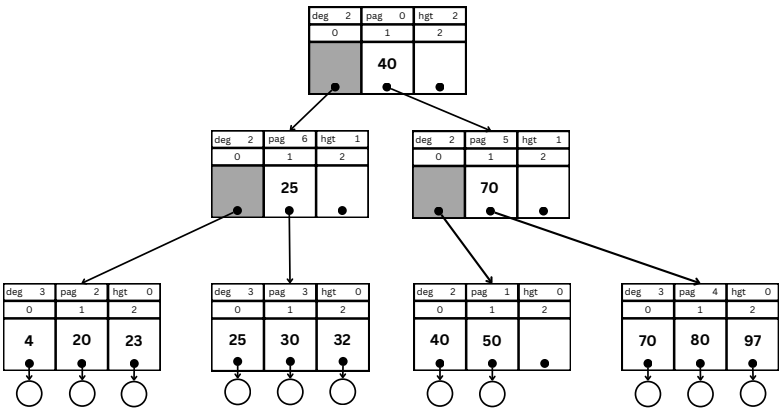
B-Tree Operations—Delete

```
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 Operations—Delete (Example)

```
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();
```

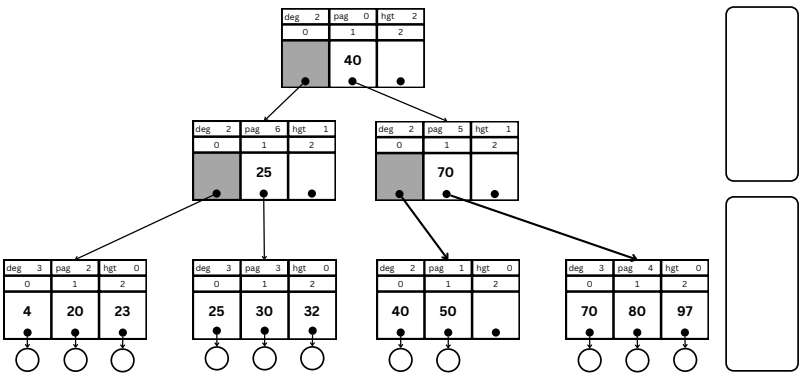
> Delete 1; Step 1;
> tree=(*pag 0); delete_key=32;
> finished;
> i; j;
> current=(*pag 0); tmp_node;



B-Tree Operations—Delete (Example)

```
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;
```

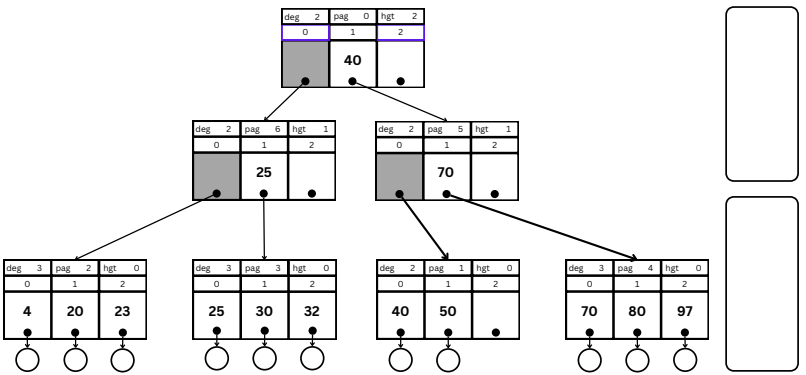
> Delete 1; Step 2;
> tree=(*pag 0); delete_key=32;
> finished;
> i; j;
> current=(*pag 0); tmp_node;
> lower=0; upper=2;



B-Tree Operations—Delete (Example)

```
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          }
```

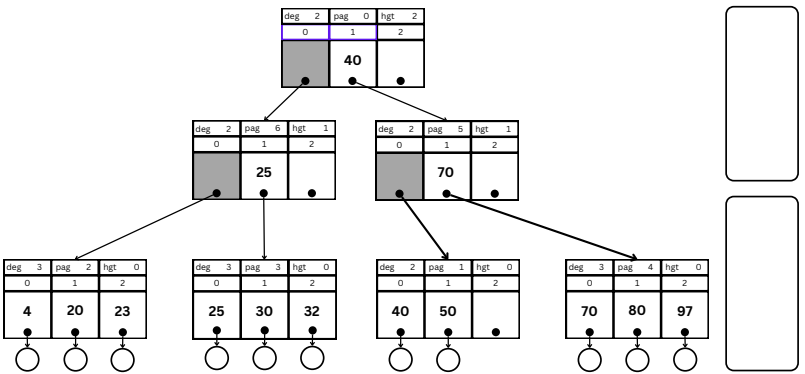
> Delete 1; Step 3;
> tree=(*pag 0); delete_key=32;
> finished;
> i; j;
> current=(*pag 0); tmp_node;
> lower=0; upper=2 → 1;



B-Tree Operations—Delete (Example)

```
7     while( current->height > 0 ) {

13        while( upper > lower +1 ) {

20            push_index_stack( lower );
21            push_node_stack( current );
22            current = current->next[lower];
23        }
```

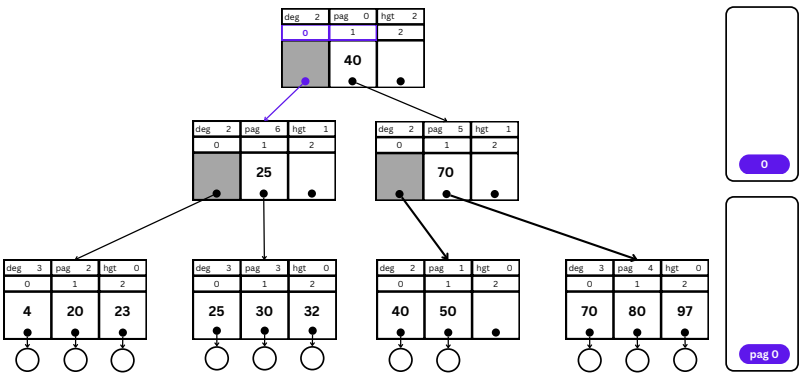
> Delete 1; Step 4;
> tree=(*pag 0); delete_key=32;
> finished;
> i; j;
> current=(*pag 0) → (*pag 6); tmp_node;
> lower=0; upper=1;



B-Tree Operations—Delete (Example)

```
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;
```

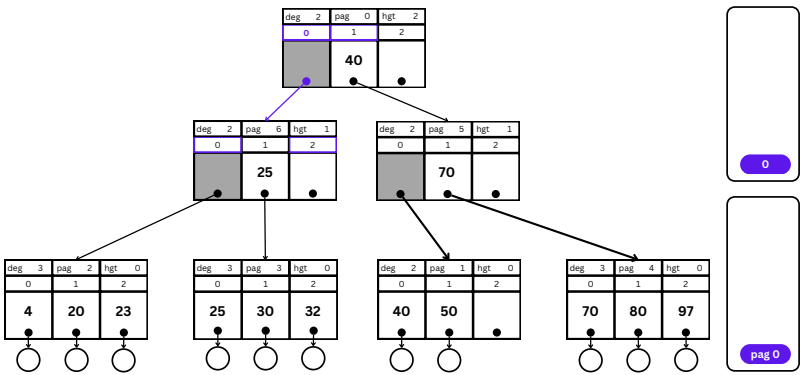
> Delete 1; Step 5;
> tree=(*pag 0); delete_key=32;
> finished;
> i; j;
> current=(*pag 6); tmp_node;
> lower=0; upper=2;



B-Tree Operations—Delete (Example)

```
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          }
```

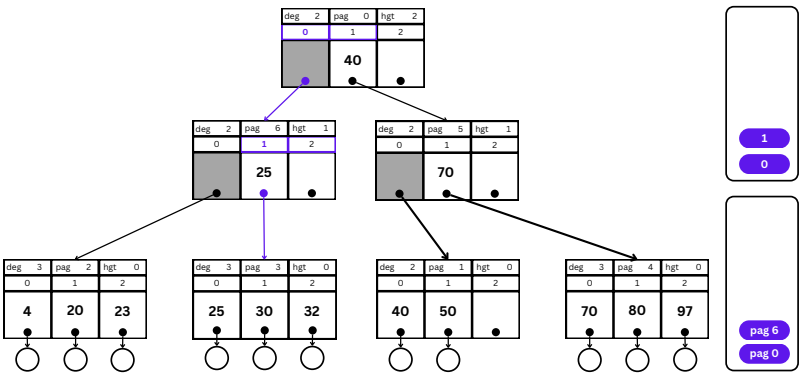
> Delete 1; Step 6;
> tree=(*pag 0); delete_key=32;
> finished;
> i; j;
> current=(*pag 6); tmp_node;
> lower=0 → 1; upper=2;



B-Tree Operations—Delete (Example)

```
7     while( current->height > 0 ) {

13        while( upper > lower +1 ) {

20        push_index_stack( lower );
21        push_node_stack( current );
22        current = current->next[lower];
23    }
```

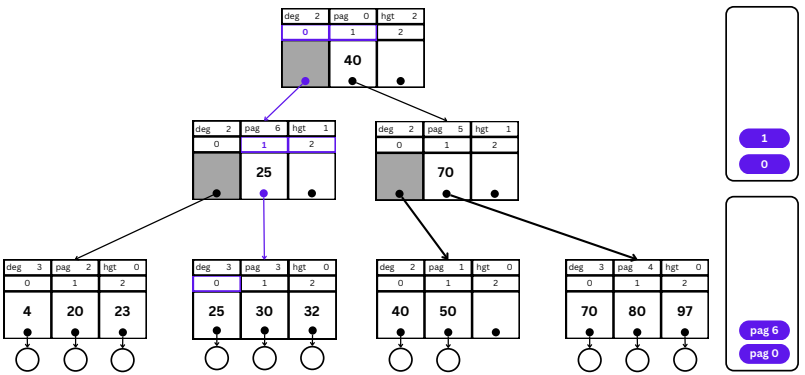
> Delete 1; Step 7;
> tree=(*pag 0); delete_key=32;
> finished;
> i; j;
> current=(*pag 6) → (*pag 3); tmp_node;
> lower=1; upper=2;



B-Tree Operations—Delete (Example)

```
24      /* now current is leaf node from which we delete */
25      for ( i=0; i < current->degree ; i++ )
26          if( current->key[i] == delete_key )
27              break;
28      if( i == current->degree ) {
29          /* delete failed; key does not exist */
30          return( NULL );
31      } else {
```

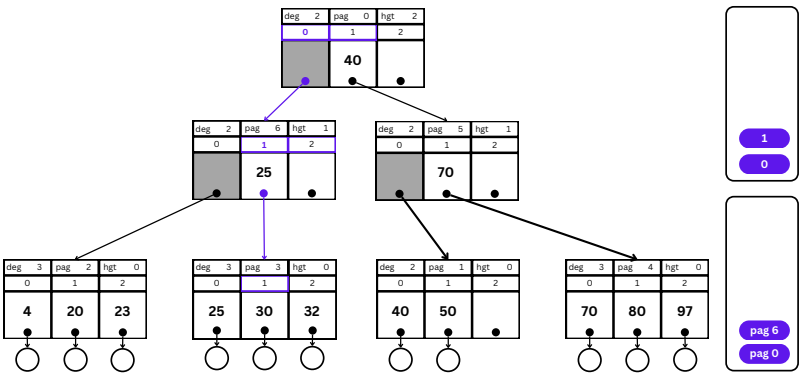
> Delete 1; Step 8;
> tree=(*pag 0); delete_key=32;
> finished;
> i=0; j;
> current=(*pag 3); tmp_node;



B-Tree Operations—Delete (Example)

```
24      /* now current is leaf node from which we delete */
25      for ( i=0; i < current->degree ; i++ )
26          if( current->key[i] == delete_key )
27              break;
28      if( i == current->degree ) {
29          /* delete failed; key does not exist */
30          return( NULL );
31      } else {
```

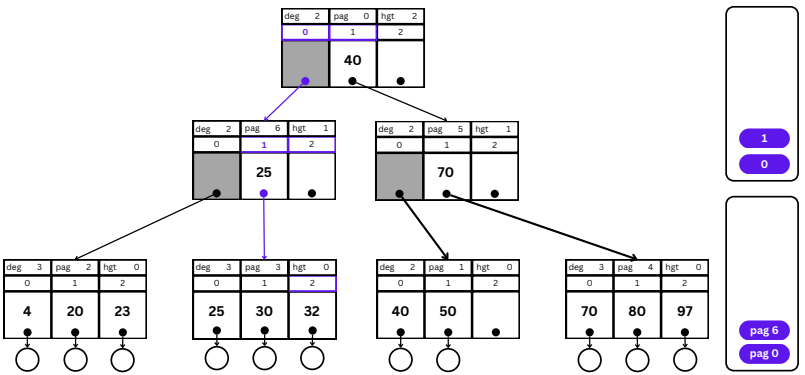
> Delete 1; Step 9;
> tree=(*pag 0); delete_key=32;
> finished;
> i=1; j;
> current=(*pag 3); tmp_node;



B-Tree Operations—Delete (Example)

```
24      /* now current is leaf node from which we delete */
25      for ( i=0; i < current->degree ; i++ )
26          if( current->key[i] == delete_key )
27              break;
28      if( i == current->degree ) {
29          /* delete failed; key does not exist */
30          return( NULL );
31      } else {
```

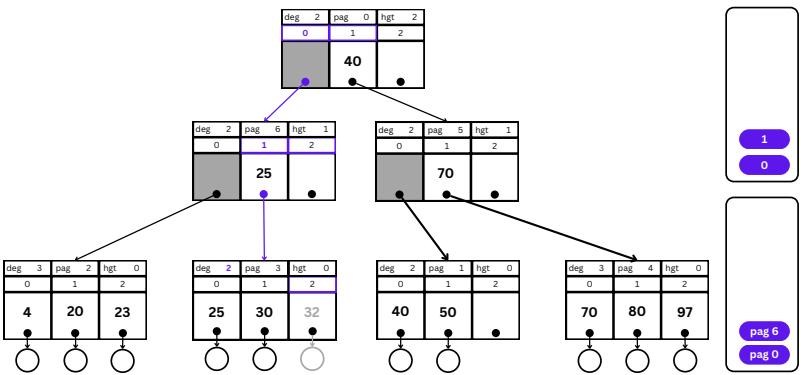
> Delete 1; Step 10;
> tree=(*pag 0); delete_key=32;
> finished;
> i=2; j;
> current=(*pag 3); tmp_node;



B-Tree Operations—Delete (Example)

```
32          /* key exists, now delete from leaf node */
33          object_t *del_object;
34          del_object = (object_t *) current->next[i];
35          current->degree -=1;
36          while( i < current->degree ) {
37              current->next[i] = current->next[i+1];
38              current->key[i] = current->key[i+1];
39              i+=1;
40          }
41          /* deleted from node, now rebalance */
42          finished = 0;
```

> Delete 1; Step 11;
> tree=(*pag 0); delete_key=32;
> del_object=(*32);
> finished=0;
> i=2; j;
> current=(*pag 3); tmp_node;

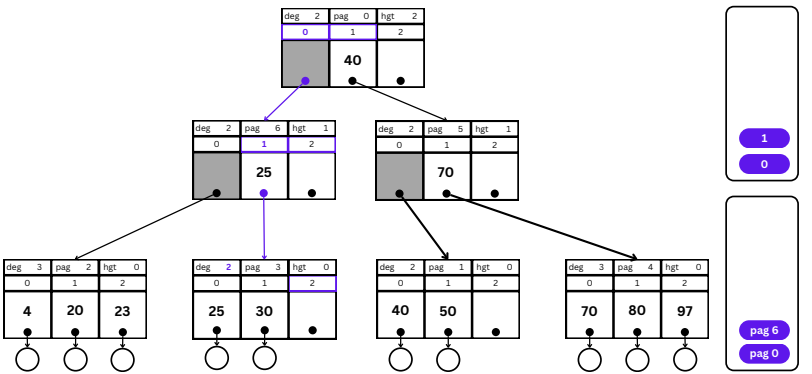


B-Tree Operations—Delete (Example)

```
43          while( ! finished ) {
44              if(current->degree >= ALPHA ) {
45                  finished = 1;
46                  /* node still full enough, can stop */
47              }
```

```
202         } /* end of while not finished */
203
204         return( del_object );
205
206     } /* end of delete object exists if-else */
207 }
```

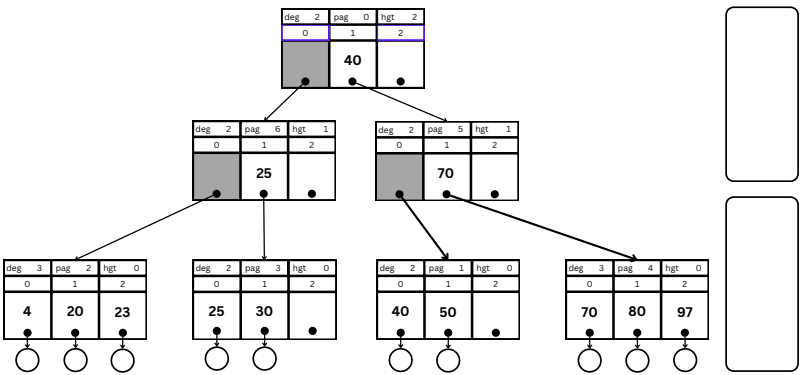
> Delete 1; Step 12;
> tree=(*pag 0); delete_key=32;
> del_object=(*32);
> finished=1;
> i=2; j;
> current=(*pag 3); tmp_node;



B-Tree Operations—Delete (Example)

```
1   object_t *delete(tree_node_t *tree, key_t delete_key) {
...
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;
```

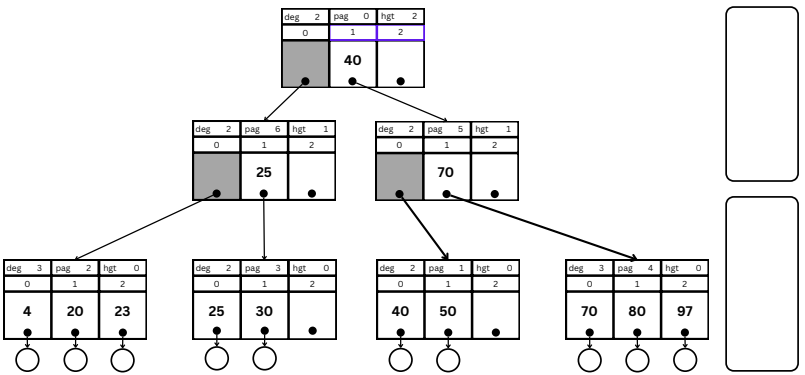
> Delete 2; Step 1;
> tree=(*pag 0); delete_key=40;
> finished;
> i; j;
> current=(*pag 0); tmp_node;
> lower=0; upper=2



B-Tree Operations—Delete (Example)

```
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          }
```

> Delete 2; Step 2;
> tree=(*pag 0); delete_key=40;
> finished;
> i; j;
> current=(*pag 0); tmp_node;
> lower=0 → 1; upper=2

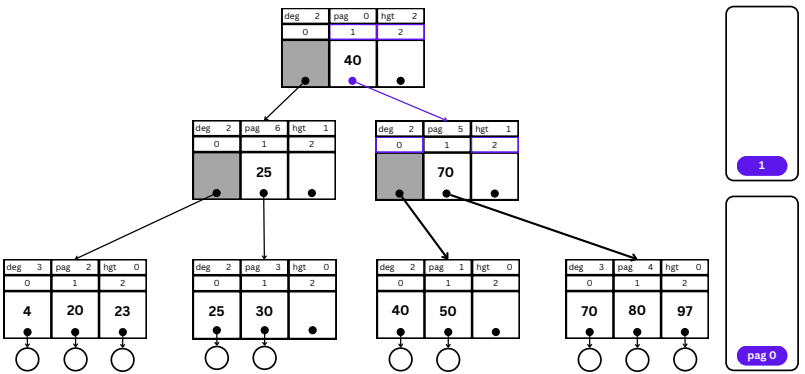


B-Tree Operations—Delete (Example)

```
13        while( upper > lower +1 ) {
```

```
20        push_index_stack( lower );
21        push_node_stack( current );
22        current = current->next[lower];
```

- Delete 2; Step 3;
- tree=(*pag 0); delete_key=40;
- finished;
- i; j;
- current=(*pag 0) → (*pag 5); tmp_node;
- lower=1; upper=2

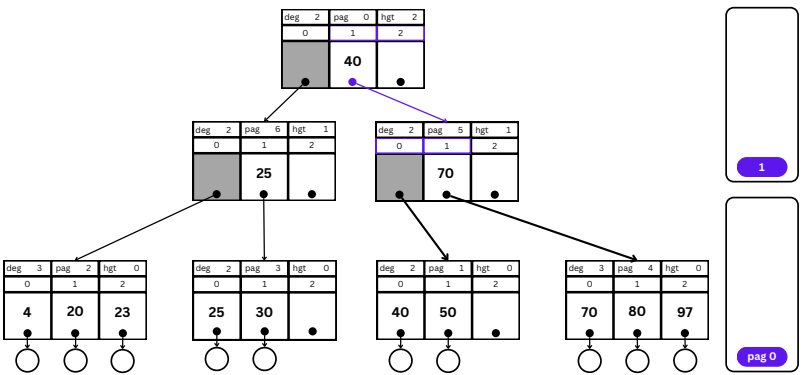


B-Tree Operations—Delete (Example)

```
7     while( current->height > 0 ) {
```

```
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        }
```

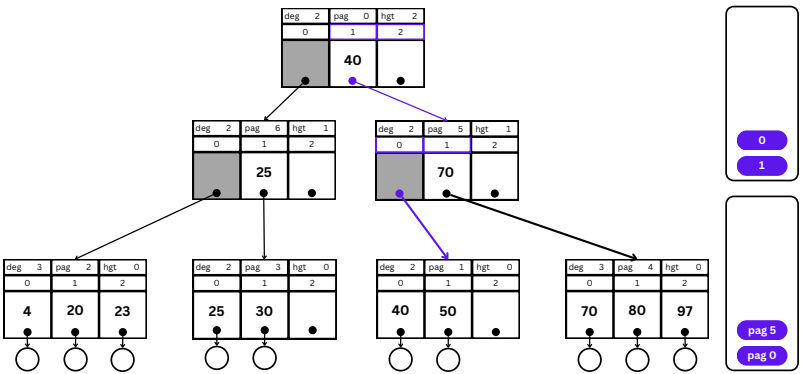
> Delete 2; Step 4;
> tree=(*pag 0); delete_key=40;
> finished;
> i; j;
> current=(*pag 5); tmp_node;
> lower=0; upper=2 → 1



B-Tree Operations—Delete (Example)

```
7     while( current->height > 0 ) {

13        while( upper > lower +1 ) {

20        push_index_stack( lower );
21        push_node_stack( current );
22        current = current->next[lower];
```

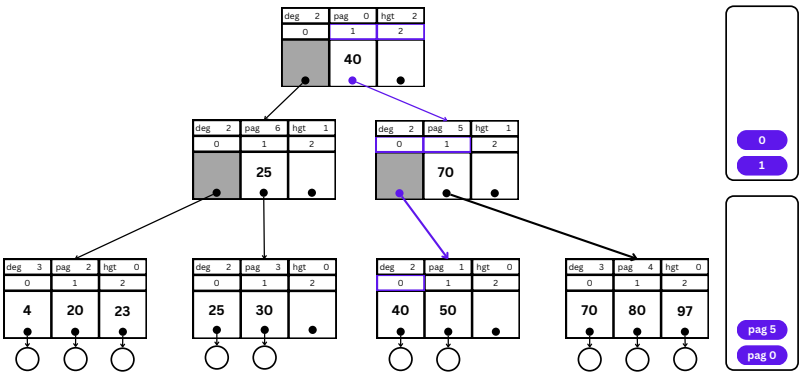
> Delete 2; Step 5;
> tree=(*pag 0); delete_key=40;
> finished;
> i; j;
> current=(*pag 5) → (*pag 1); tmp_node;
> lower=0; upper=1



B-Tree Operations—Delete (Example)

```
25      for ( i=0; i < current->degree ; i++ )
26          if( current->key[i] == delete_key )
27              break;
28      if( i == current->degree ) {
29          /* delete failed; key does not exist */
30          return( NULL );
31      } else {
```

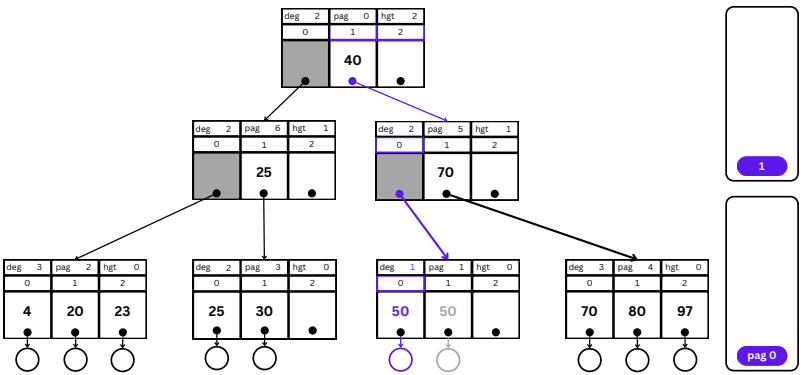
> Delete 2; Step 6;
> tree=(*pag 0); delete_key=40;
> finished;
> i=0; j;
> current=(*pag 1); tmp_node;



B-Tree Operations—Delete (Example)

```
33          object_t *del_object;
34          del_object = (object_t *) current->next[i];
35          current->degree -=1;
36          while( i < current->degree ) {
37              current->next[i] = current->next[i+1];
38              current->key[i] = current->key[i+1];
39              i+=1;
40          }
```

- Delete 2; Step 7;
- tree=(*pag 0); delete_key=40;
- finished; del_object=(*40);
- i=0 → 1; j;
- current=(*pag 1); tmp_node;



B-Tree Operations—Delete (Example)

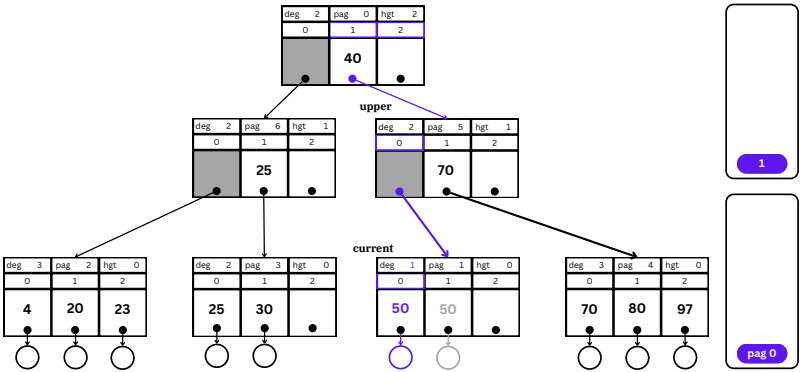
```
42          finished = 0;
43          while( ! finished ) {
44              if(current->degree >= ALPHA ) {
```

```
48              else {
49                  /* node became underfull */
50                  if( stack_empty() ) {
```

```
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();
```

> Delete 2; Step 8;
> tree=(*pag 0); delete_key=40;
> finished; del_object=(*40);
> i=1; j;
> current=(*pag 1); curr=0; tmp_node;
> upper=(*pag 5); neighbor;

B-Tree Operations—Delete (Example)

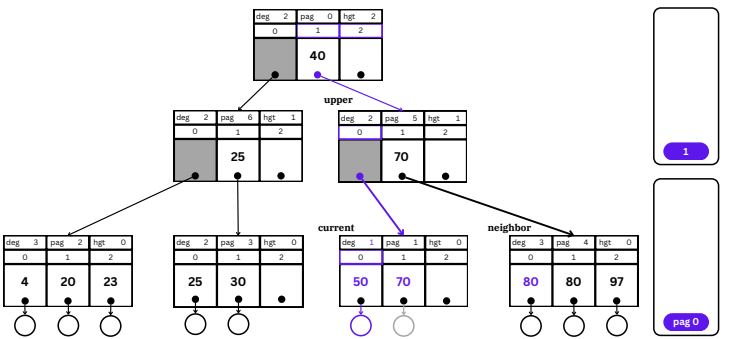


B-Tree Operations—Delete (Example)

```
79                      if( curr < upper->degree -1 ) {
80                          /* not last*/
81                          neighbor = upper->next[curr+1];
82                          if( neighbor->degree > ALPHA ) {
83                              /* sharing possible */
84                              i = current->degree;
85                              if( current->height > 0 )
```

```
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                              }
```

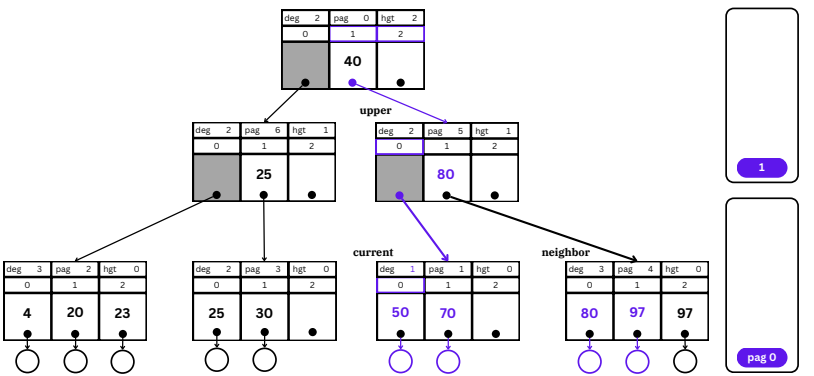
> Delete 2; Step 9;
> tree=(*pag 0); delete_key=40;
> finished; del_object=(*40);
> i=1; j;
> current=(*pag 1); curr=0; tmp_node;
> upper=(*pag 5); neighbor=(*pag 4);



B-Tree Operations—Delete (Example)

```
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                         }
```

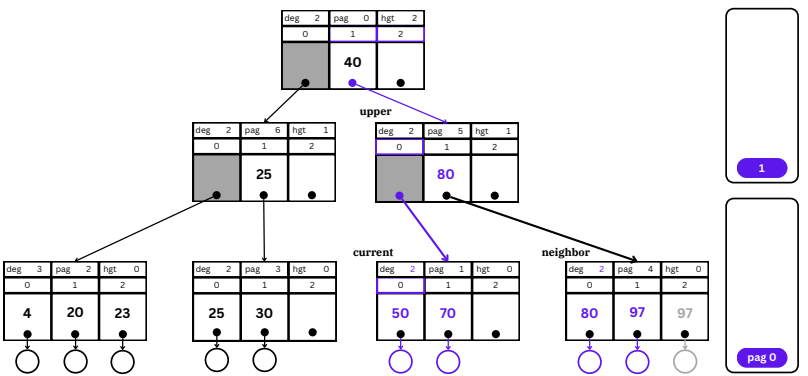
- Delete 2; Step 10;
- tree=(*pag 0); delete_key=40;
- finished; del_object=(*40);
- i=1; j=2 → 3;
- current=(*pag 1); curr=0; tmp_node;
- upper=(*pag 5); neighbor=(*pag 4);



B-Tree Operations—Delete (Example)

```
43          while( ! finished ) {

107                             neighbor->degree -= 1;
108                             current->degree += 1;
109                             finished = 1;
110                         } /* sharing complete */

202         } /* end of while not finished */
203
204         return( del_object );
205
206     } /* end of delete object exists if-else */
207 }
```

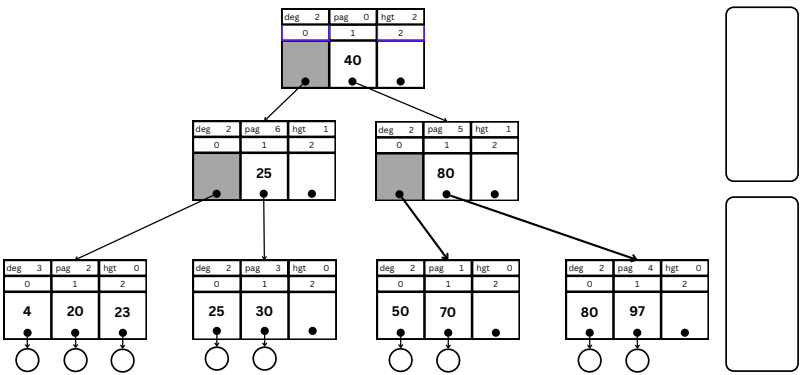
- Delete 2; Step 11;
- tree=(*pag 0); delete_key=40;
- finished=1; del_object=(*40);
- i=1; j=3;
- current=(*pag 1); curr=0; tmp_node;
- upper=(*pag 5); neighbor=(*pag 4);



B-Tree Operations—Delete (Example)

```
1   object_t *delete(tree_node_t *tree, key_t delete_key) {
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;
```

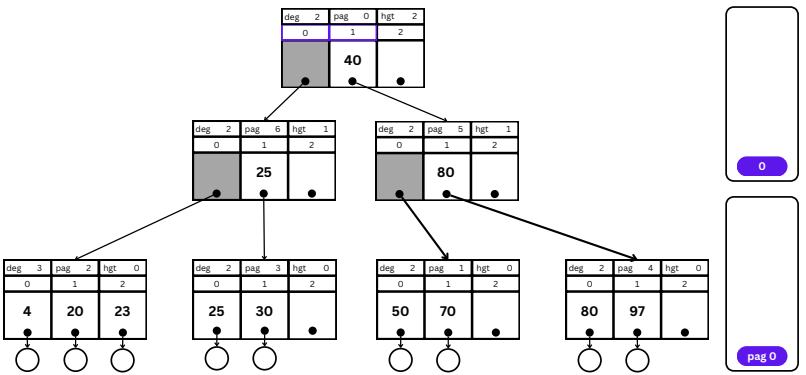
> Delete 3; Step 1;
> tree=(*pag 0); delete_key=30;
> finished;
> i; j;
> current=(*pag 0);
> lower=0; upper=2;



B-Tree Operations—Delete (Example)

```
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          }
```

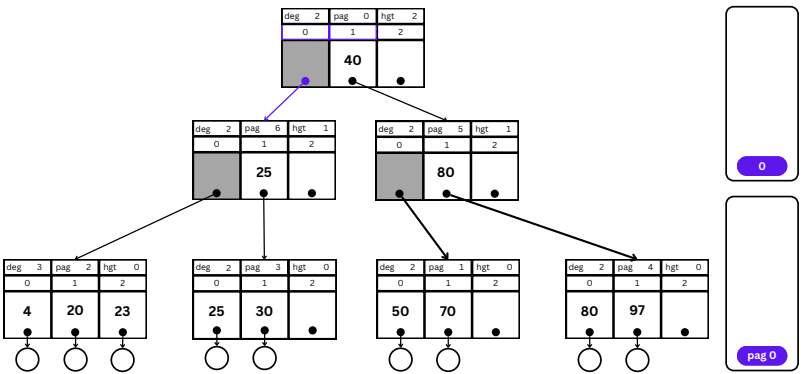
> Delete 3; Step 2;
> tree=(*pag 0); delete_key=30;
> finished;
> i; j;
> current=(*pag 0);
> lower=0; upper=2 → 1;



B-Tree Operations—Delete (Example)

```
7     while( current->height > 0 ) {

13        while( upper > lower +1 ) {

20        push_index_stack( lower );
21        push_node_stack( current );
22        current = current->next[lower];
23    }
```

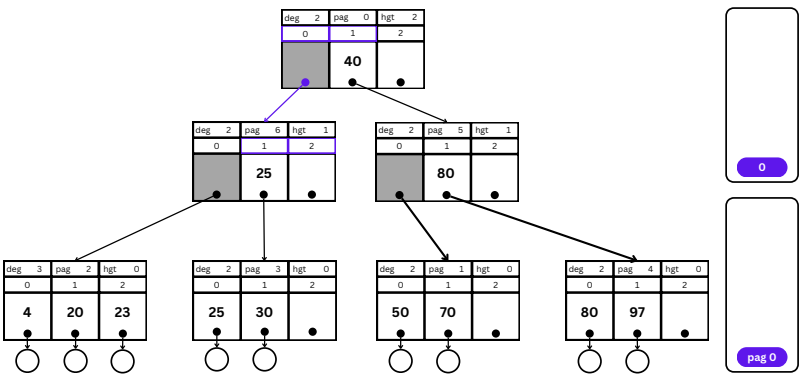
> Delete 3; Step 3;
> tree=(*pag 0); delete_key=30;
> finished;
> i; j;
> current=(*pag 0) → (*pag 6);
> lower=0; upper=1;



B-Tree Operations—Delete (Example)

```
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          }
```

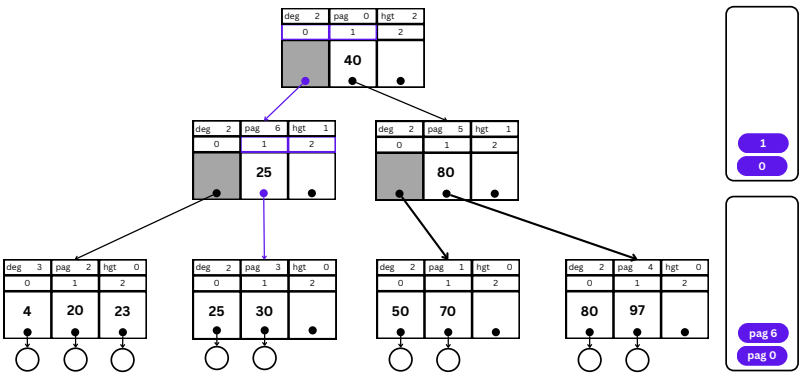
> Delete 3; Step 4;
> tree=(*pag 0); delete_key=30;
> finished;
> i; j;
> current=(*pag 6);
> lower=0 → 1; upper=2;



B-Tree Operations—Delete (Example)

```
7     while( current->height > 0 ) {

13        while( upper > lower +1 ) {

20        push_index_stack( lower );
21        push_node_stack( current );
22        current = current->next[lower];
23    }
```

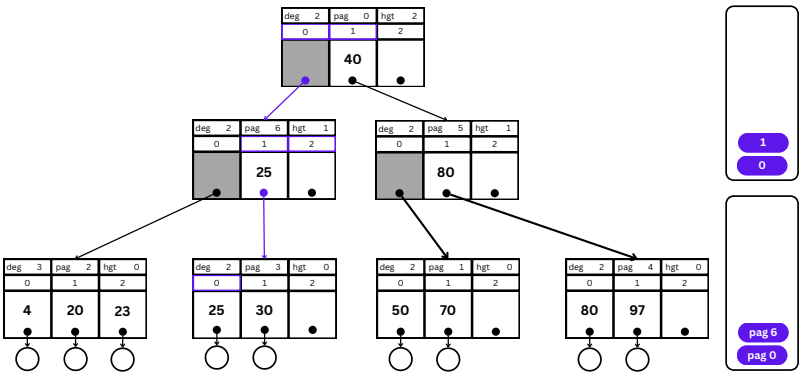
> Delete 3; Step 5;
> tree=(*pag 0); delete_key=30;
> finished;
> i; j;
> current=(*pag 6) → (*pag 3);
> lower=1; upper=2;



B-Tree Operations—Delete (Example)

```
25      for ( i=0; i < current->degree ; i++ )
26          if( current->key[i] == delete_key )
27              break;
28      if( i == current->degree ) {
29          /* delete failed; key does not exist */
30          return( NULL );
31      } else {
```

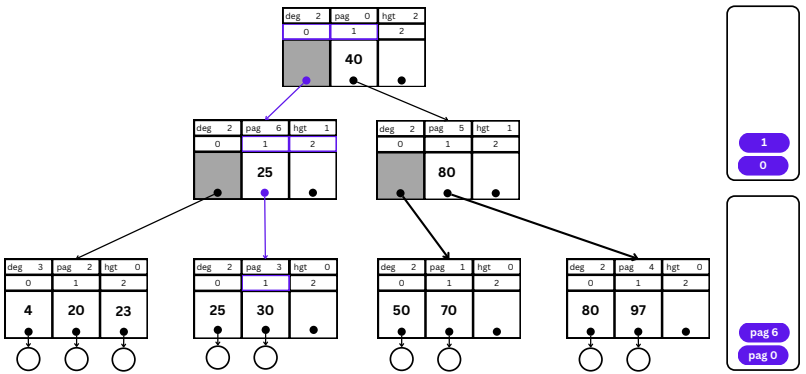
> Delete 3; Step 6;
> tree=(*pag 0); delete_key=30;
> finished;
> i=0; j;
> current=(*pag 3);



B-Tree Operations—Delete (Example)

```
25      for ( i=0; i < current->degree ; i++ )
26          if( current->key[i] == delete_key )
27              break;
28      if( i == current->degree ) {
29          /* delete failed; key does not exist */
30          return( NULL );
31      } else {
```

> Delete 3; Step 7;
> tree=(*pag 0); delete_key=30;
> finished;
> i=1; j;
> current=(*pag 3);

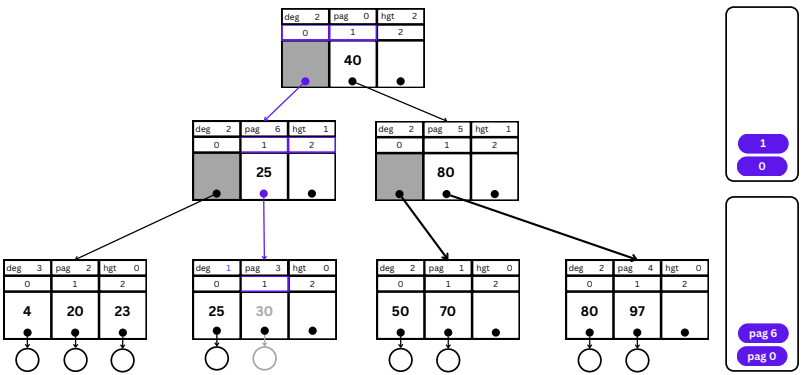


B-Tree Operations—Delete (Example)

```
33          object_t *del_object;
34          del_object = (object_t *) current->next[i];
35          current->degree -=1;
36          while( i < current->degree ) {
```

```
40          }
```

> Delete 3; Step 8;
> tree=(*pag 0); delete_key=30;
> finished;
> i=1; j;
> current=(*pag 3);



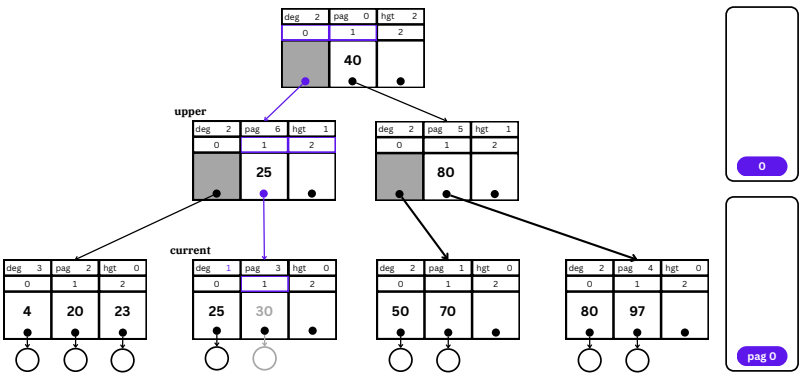
B-Tree Operations—Delete (Example)

```
42          finished = 0;
43          while( ! finished ) {
44              if(current->degree >= ALPHA ) {
```

```
48              else {
```

```
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();
```

> Delete 3; Step 9;
> tree=(*pag 0); delete_key=30;
> finished=0; del_object=(*30);
> i=1; j; curr=1;
> current=(*pag 3); upper=(*pag 6);

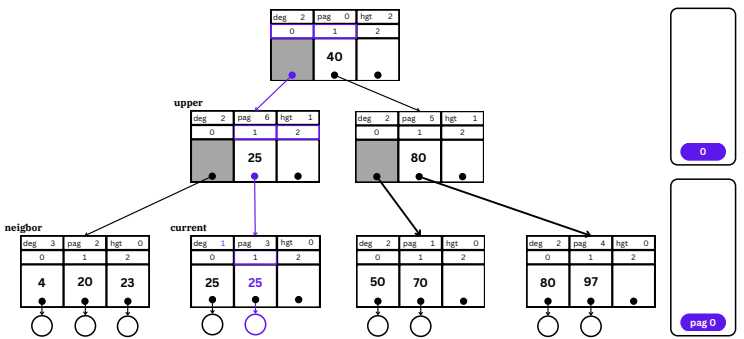


B-Tree Operations—Delete (Example)

```
79                  if( curr < upper->degree -1 ) {
```

```
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                             }
```

- Delete 3; Step 10;
- tree=(*pag 0); delete_key=30;
- finished=0; del_object=(*30);
- i=1; j=1; curr=1;
- current=(*pag 3); upper=(*pag 6); neighbor=(*pag 2);

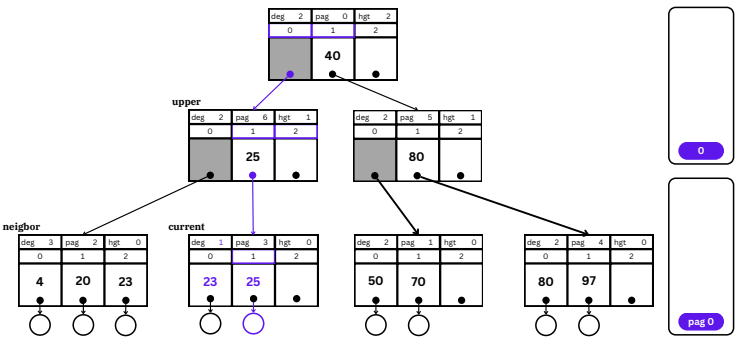


B-Tree Operations—Delete (Example)

```
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 ) {
```

```
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                             }
```

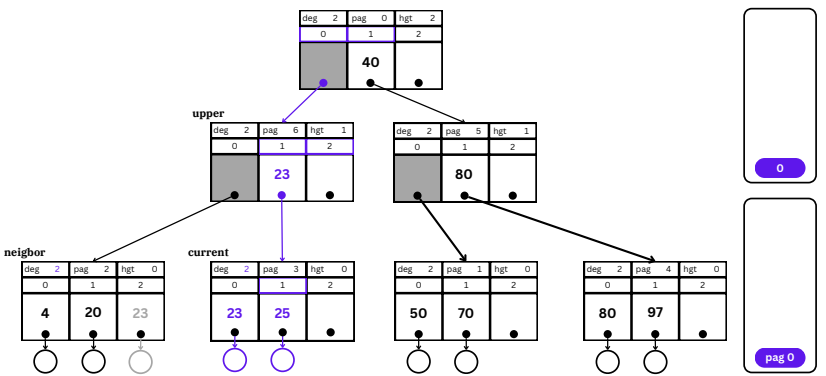
- Delete 3; Step 11;
- tree=(*pag 0); delete_key=30;
- finished=0; del_object=(*30);
- i=3; j=1; curr=1;
- current=(*pag 3); upper=(*pag 6); neighbor=(*pag 2);



B-Tree Operations—Delete (Example)

```
43          while( ! finished ) {

170                             upper->key[curr] =
171                                 neighbor->key[i-1];
172                             neighbor->degree -= 1;
173                             current->degree += 1;
174                             finished = 1;

202         } /* end of while not finished */
203
204         return( del_object );
205
206     } /* end of delete object exists if-else */
```

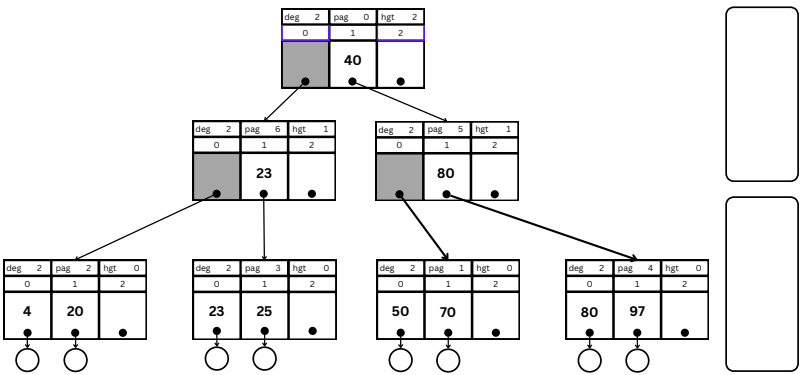
> Delete 3; Step 12;
> tree=(*pag 0); delete_key=30;
> finished=1; del_object=(*30);
> i=3; j=1; curr=1;
> current=(*pag 3); upper=(*pag 6); neighbor=(*pag 2);



B-Tree Operations—Delete (Example)

```
1   object_t *delete(tree_node_t *tree, key_t delete_key) {
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;
```

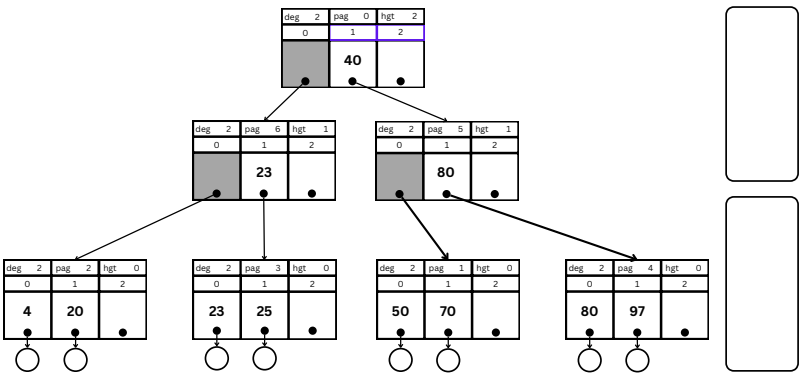
> Delete 4; Step 1;
> tree=(*pag 0); delete_key=50;
> finished;
> i; j;
> current=(*pag 0);
> lower=0; upper=2;



B-Tree Operations—Delete (Example)

```
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          }
```

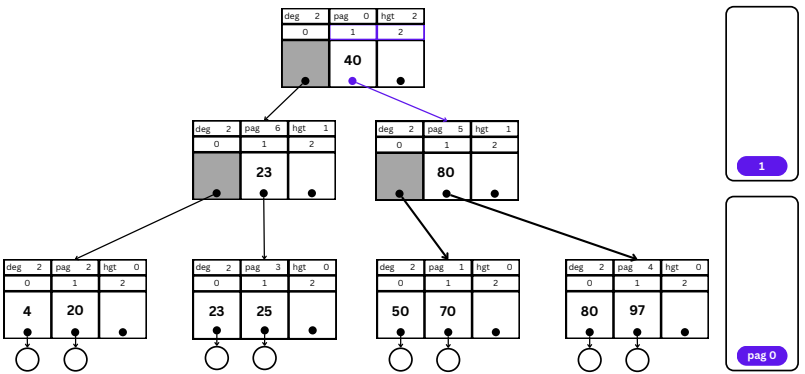
> Delete 4; Step 2;
> tree=(*pag 0); delete_key=50;
> finished;
> i; j;
> current=(*pag 0);
> lower=0 → 1; upper=2;



B-Tree Operations—Delete (Example)

```
7     while( current->height > 0 ) {

13        while( upper > lower +1 ) {

20        push_index_stack( lower );
21        push_node_stack( current );
22        current = current->next[lower];
23    }
```

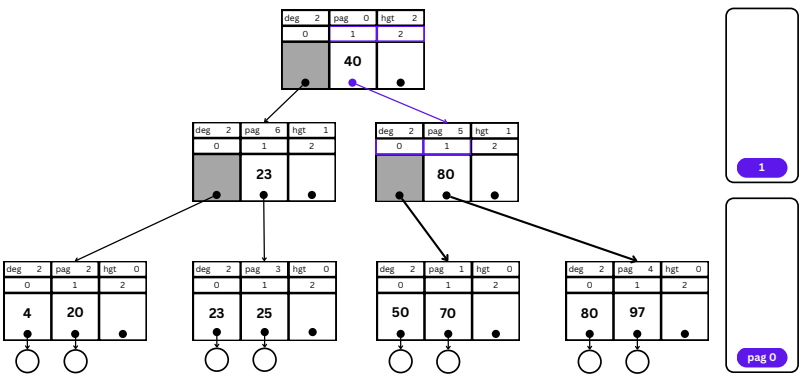
> Delete 4; Step 3;
> tree=(*pag 0); delete_key=50;
> finished;
> i; j;
> current=(*pag 0) → (*pag 5);
> lower=1; upper=2;



B-Tree Operations—Delete (Example)

```
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          }
```

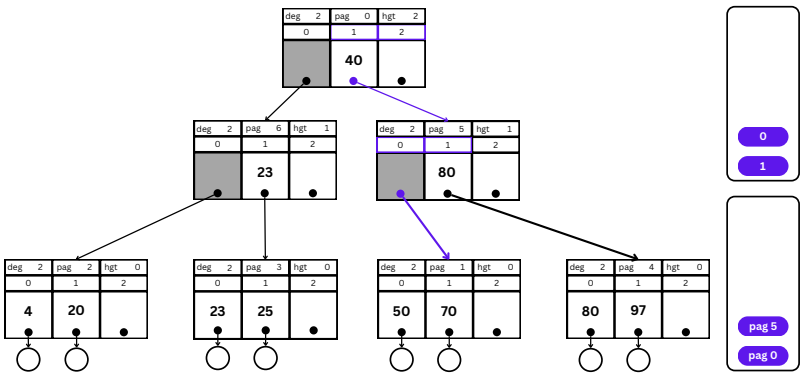
> Delete 4; Step 4;
> tree=(*pag 0); delete_key=50;
> finished;
> i; j;
> current=(*pag 5);
> lower=0; upper=2 → 1;



B-Tree Operations—Delete (Example)

```
7     while( current->height > 0 ) {

13        while( upper > lower +1 ) {

20        push_index_stack( lower );
21        push_node_stack( current );
22        current = current->next[lower];
23    }
```

> Delete 4; Step 5;
> tree=(*pag 0); delete_key=50;
> finished;
> i; j;
> current=(*pag 5) → (*pag 1);
> lower=0; upper=1;

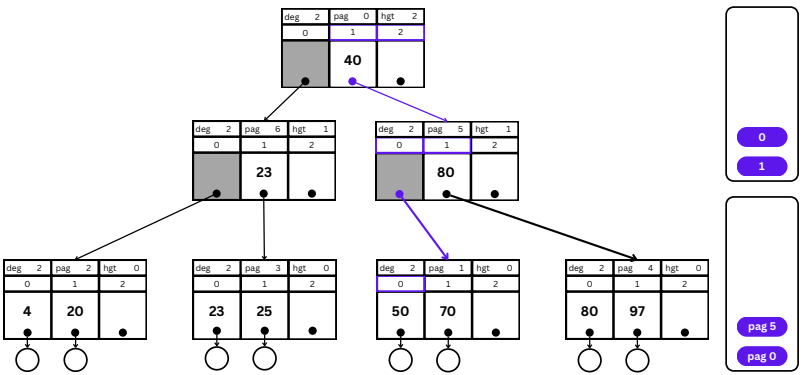


B-Tree Operations—Delete (Example)

```
25      for ( i=0; i < current->degree ; i++ )
26          if( current->key[i] == delete_key )
27              break;
28      if( i == current->degree ) {
```

```
31      } else {
```

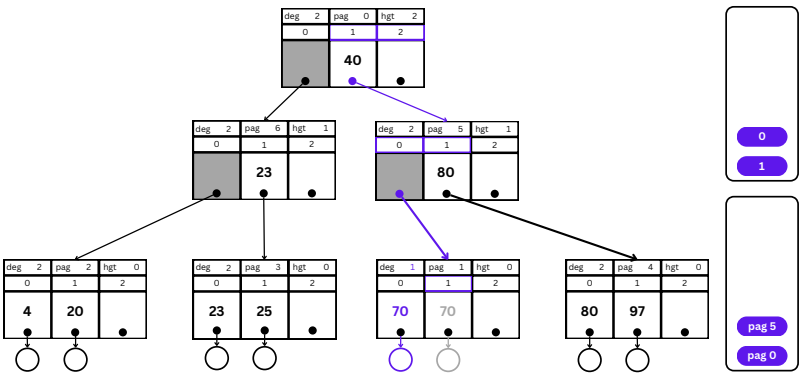
> Delete 4; Step 6;
> tree=(*pag 0); delete_key=50;
> finished;
> i=0; j;
> current=(*pag 1);



B-Tree Operations—Delete (Example)

```
33          object_t *del_object;
34          del_object = (object_t *) current->next[i];
35          current->degree -=1;
36          while( i < current->degree ) {
37              current->next[i] = current->next[i+1];
38              current->key[i] = current->key[i+1];
39              i+=1;
40          }
```

- Delete 4; Step 7;
- tree=(*pag 0); delete_key=50;
- finished; del_object=(*50);
- i=0 → 1; j;
- current=(*pag 1);



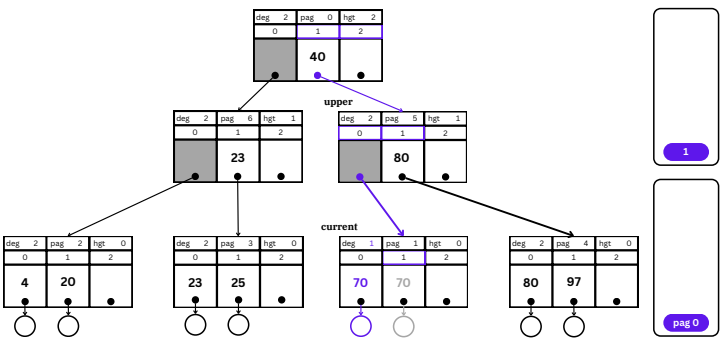
B-Tree Operations—Delete (Example)

```
42          finished = 0;
43          while( ! finished ) {
44              if(current->degree >= ALPHA ) {
```

```
48              else {
49                  /* node became underfull */
50                  if( stack_empty() ) {
```

```
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();
```

- Delete 4; Step 8;
- tree=(*pag 0); delete_key=50;
- finished=0; del_object=(*50);
- i=1; j; curr=0;
- current=(*pag 1); upper=(*pag 5);

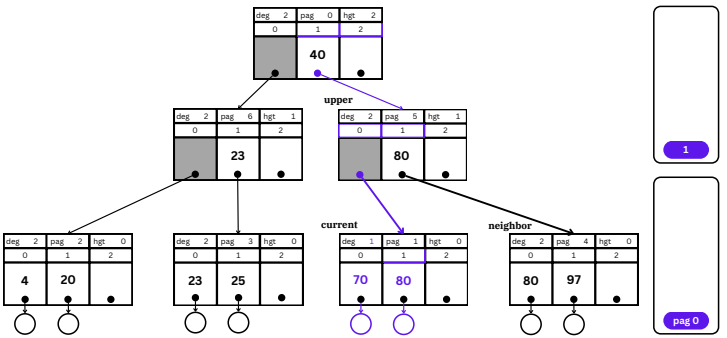


B-Tree Operations—Delete (Example)

```
79                      if( curr < upper->degree -1 ) {
80                          /* not last*/
81                          neighbor = upper->next[curr+1];
82                          if( neighbor->degree > ALPHA ) {
```

```
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];
```

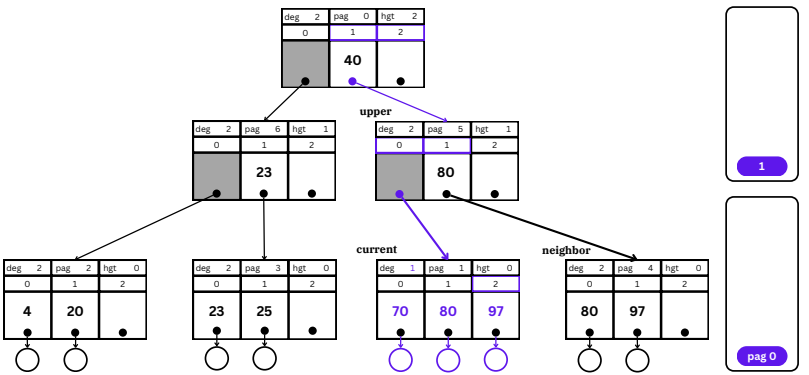
> Delete 4; Step 9;
> tree=(*pag 0); delete_key=50;
> finished=0; del_object=(*50);
> i=1; j; curr=0;
> current=(*pag 1); upper=(*pag 5); neighbor=(*pag 4);



B-Tree Operations—Delete (Example)

```
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                             }
```

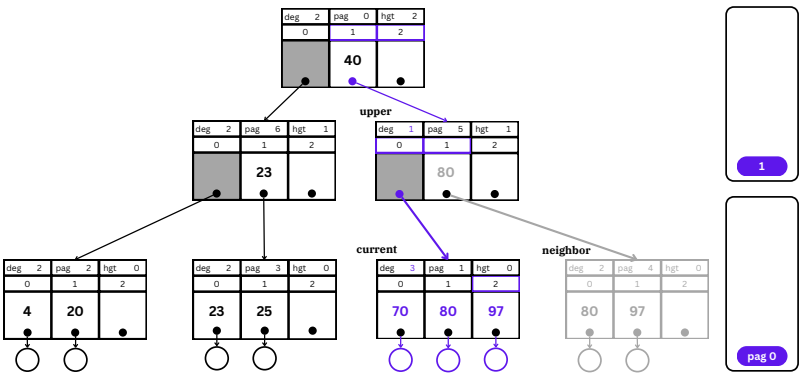
- Delete 4; Step 10;
- tree=(*pag 0); delete_key=50;
- finished=0; del_object=(*50);
- i=1 → 2; j=1 → 2; curr=0;
- current=(*pag 1); upper=(*pag 5); neighbor=(*pag 4);



B-Tree Operations—Delete (Example)

```
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];
135                             upper->key[i] =
136                                 upper->key[i+1];
137                             i +=1;
138                         }
139                         /* deleted from upper, now
                            ↪  propagate up */
140                         current = upper;
```

- Delete 4; Step 11;
- tree=(*pag 0); delete_key=50;
- finished=0; del_object=(*50);
- i=2 → 1 → 2; j=2; curr=0;
- current=(*pag 1) → (*pag 5); upper=(*pag 5); neighbor=(*pag 4);



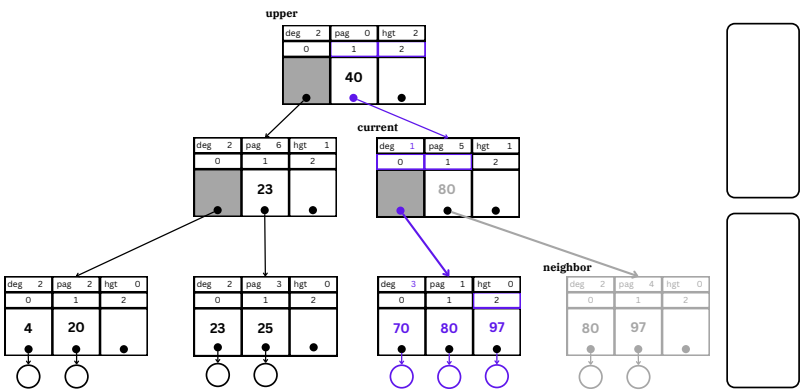
B-Tree Operations—Delete (Example)

```
43          while( ! finished ) {
44              if(current->degree >= ALPHA ) {
```

```
48              else {
49                  /* node became underfull */
50                  if( stack_empty() ) {
```

```
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();
```

> Delete 4; Step 12;
> tree=(*pag 0); delete_key=50;
> finished=0; del_object=(*50);
> i=2; j=2; curr=0 → 1;
> current=(*pag 5); upper=(*pag 5) → (*pag 0); neighbor=(*pag 4);



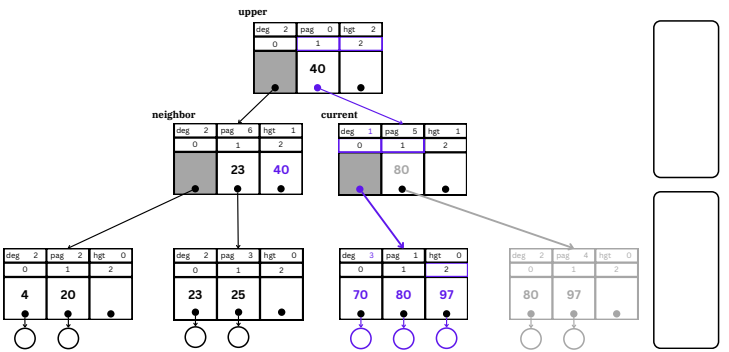
B-Tree Operations—Delete (Example)

```
79                      if( curr < upper->degree -1 ) {
```

```
143                     else {
144                         /* current is last entry in upper
                            ↪  */
145                         neighbor = upper->next[curr-1]
146                             if( neighbor->degree > ALPHA )
                                ↪  {
```

```
176                             else {
177                                 /* must join */
178                                 i = neighbor->degree;
179                                 if( current->height > 0 )
180                                     neighbor->key[i] =
```

> Delete 4; Step 13;
> tree=(*pag 0); delete_key=50;
> finished=0; del_object=(*50);
> i=2; j=2; curr=1;
> current=(*pag 5); upper=(*pag 0); neighbor=(*pag 4) → (*pag 6);

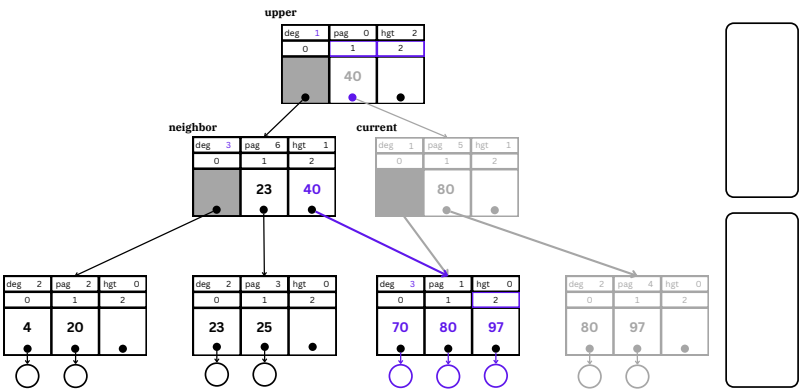


B-Tree Operations—Delete (Example)

```
185                                 neighbor->next[i] =
186                                     current->next[0];
187                                 for( j = 1; j <
                                    ↪  current->degree; 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;
```

> Delete 4; Step 14;
> tree=(*pag 0); delete_key=50;
> finished=0; del_object=(*50);
> i=2; j=1; curr=1;
> current=(*pag 5) → (*pag 0); upper=(*pag 0); neighbor=(*pag 6);



B-Tree Operations—Delete (Example)

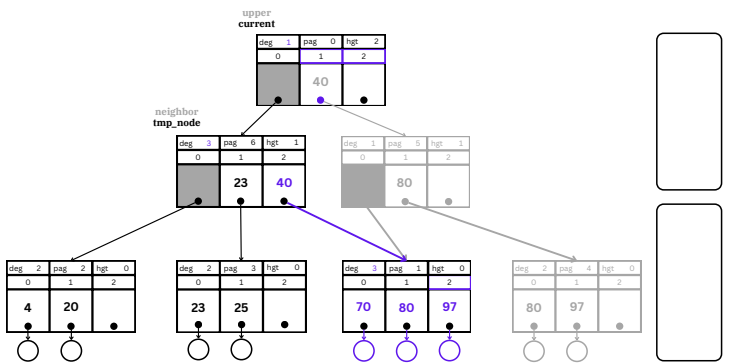
```
43          while( ! finished ) {
44              if(current->degree >= ALPHA ) {
```

```
48              else {
49                  /* node became underfull */
50                  if( stack_empty() ) {
51                      /* current is root */
52                      if(current->degree >= 2 )
```

```
55                      else if ( current->height == 0 )
```

```
58                      else {
59                          /* delete root, copy to keep address
                            ↪  */
60                          tmp_node = current->next[0];
```

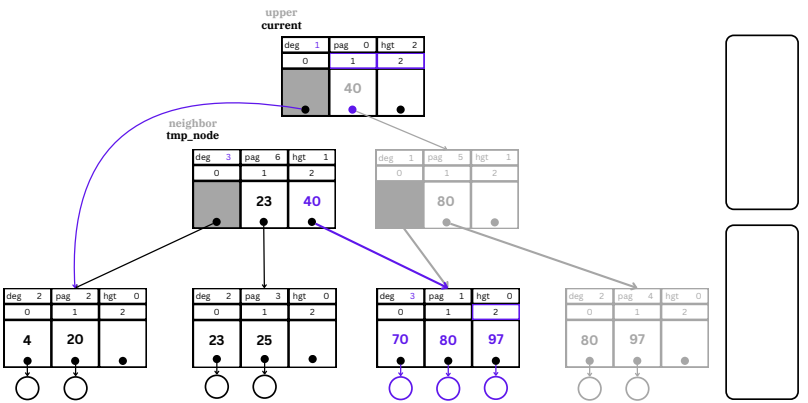
> Delete 4; Step 15;
> tree=(*pag 0); delete_key=50;
> finished=0; del_object=(*50);
> i=2; j=1; curr=1;
> current=(*pag 0); tmp_node=(*pag 6);
> upper=(*pag 0); neighbor=(*pag 6);



B-Tree Operations—Delete (Example)

```
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                      }
```

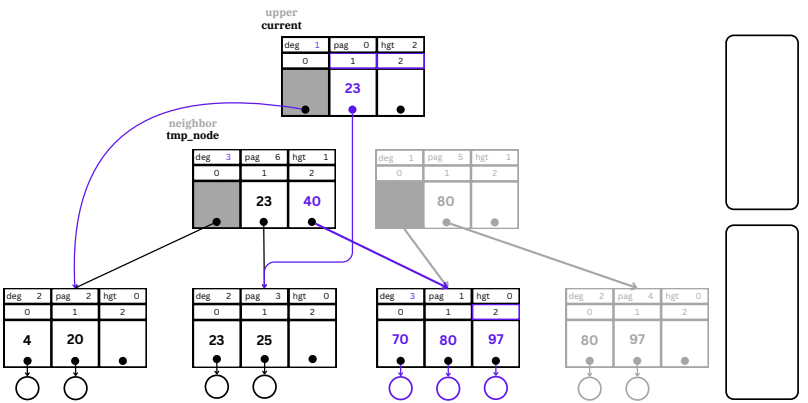
> Delete 4; Step 16;
> tree=(*pag 0); delete_key=50;
> finished=0; del_object=(*50);
> i=0 → 1; j=1; curr=1;
> current=(*pag 0); tmp_node=(*pag 6);
> upper=(*pag 0); neighbor=(*pag 6);



B-Tree Operations—Delete (Example)

```
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                          }
```

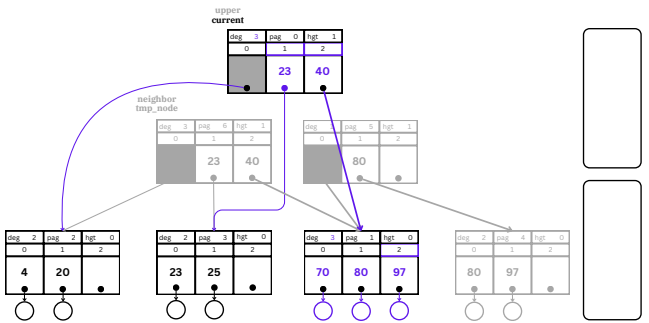
> Delete 4; Step 17;
> tree=(*pag 0); delete_key=50;
> finished=0; del_object=(*50);
> i=1 → 2; j=1; curr=1;
> current=(*pag 0); tmp_node=(*pag 6);
> upper=(*pag 0); neighbor=(*pag 6);



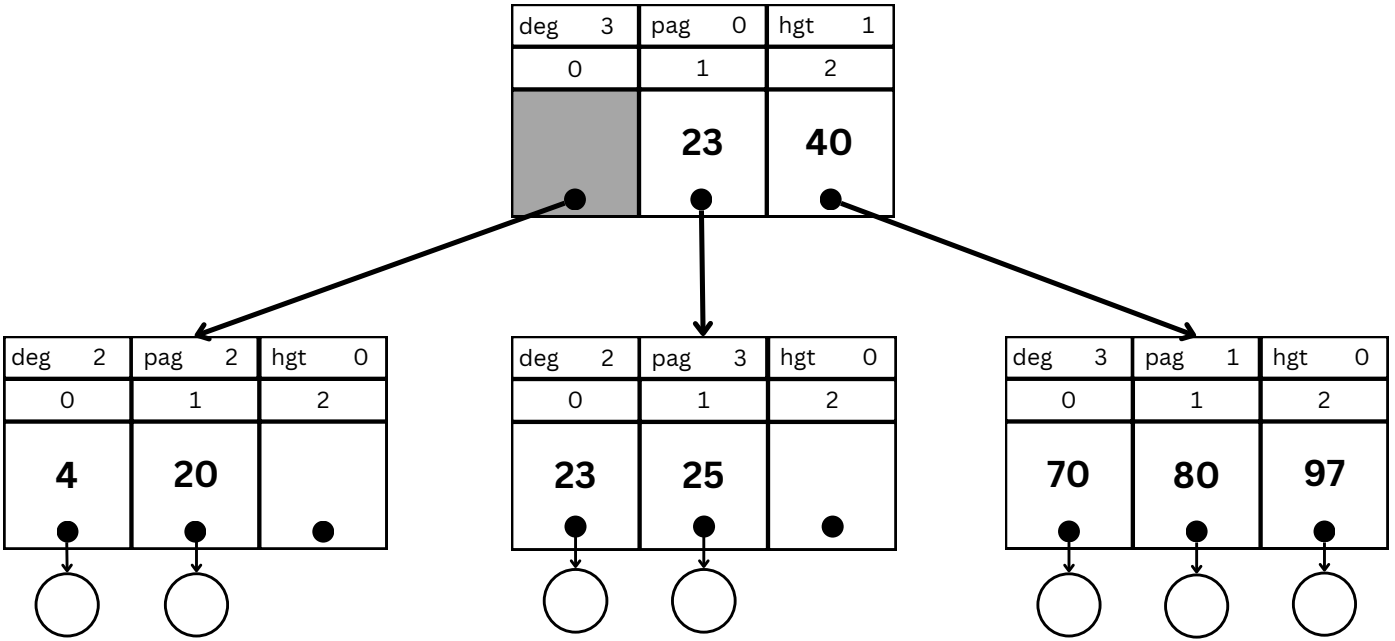
B-Tree Operations—Delete (Example)

```
43        while( ! finished ) {

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;

204       return( del_object );
```

> Delete 4; Step 18;
> tree=(*pag 0); delete_key=50;
> finished=0 → 1; del_object=(*50);
> i=2 → 3; j=1; curr=1;
> current=(*pag 0); tmp_node=(*pag 6);
> upper=(*pag 0); neighbor=(*pag 6);



B-Tree Operations—Delete (Example)



B-Tree Secondary Memory Access

- The B-Tree is fairly good for storing data in external memory in comparison to height, weight or search trees.
- The limit of 2α 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.

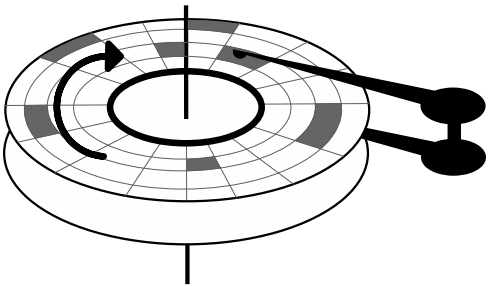


Figure: External storage with the sectors to access highlighted

B-Tree Secondary Memory Access

> In summary we can see the minimun, average and maximun time and write cost of each operation in a large B-Tree, using Big-O notation.

	Retrival	Insertion without overflow	Insertion with overflow	Deletion with or without underfull
Ω	$t = 1$ $w = 0$	$t = h$ $w = 1$	$t = h$ $w = 1$	$t = h$ $w = 1$
Θ	$t \leq h$ $w = 0$	$t = h$ $w < 1 + \frac{2}{\alpha}$	$t \leq h + 2 + \frac{2}{\alpha}$ $w \leq 3 + \frac{2}{\alpha}$	$t \leq 3h - 2$ $w \leq 2h + 1$
O	$t = h$ $w = 0$	$t = h$ $w = 2h + 1$	$t = 3h - 2$ $w = 2h + 1$	$t = 3h - 2$ $w = 2h + 1$

> 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].

AB-Tree Definition

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

$$T \in \tau\,(\alpha, \beta, h)$$

> Where h is the height of the AB-Tree.

> And, α and β are predefined constants.

> This is a tree based on B-Trees, which modifies the bounds of the B-Tree's α constant.

> The AB-Trees shares the height, keys and sub-trees properties with the B-Tree.

> It mostly shares the operations with the B-Tree, such as `find`, `insert` and `delete`, only having slight changes in some implementations.

> With this we can say that, **every B-Tree is a AB-Tree but not every AB-Tree is a B-Tree**.

> Also, an AB-Tree can be defined as

$$T \text{ is a } (\alpha, \beta)\text{–Tree}$$

which is the most popular notation.

> And we will keep using the same notation for a leaf, node and generic page from the B-Trees.

AB-Tree The α & β constants

> As stated the α in a B-Tree, a predefined constant, defines the *Branching factor* of the tree.

> Likewise the α and β of a AB-Tree, both predefined constants, will define the *Branching factor* of the tree.

> We define α and β as natural numbers such that

$$\alpha \geq 2 \qquad \beta \geq 2\alpha - 1$$

Which, as stated before, are the bounds of the α constant in some definitions of the B-Tree.

> Since the AB-Tree and B-Tree shares the same minimun number of keys and subtrees on a page, we can use the same lower bound for the height of a AB-Tree.

> But they don't share the maximun number of keys and subtrees on a page, then the upper bound of the height of the AB-Tree will be different.

AB-Tree The α & β constants

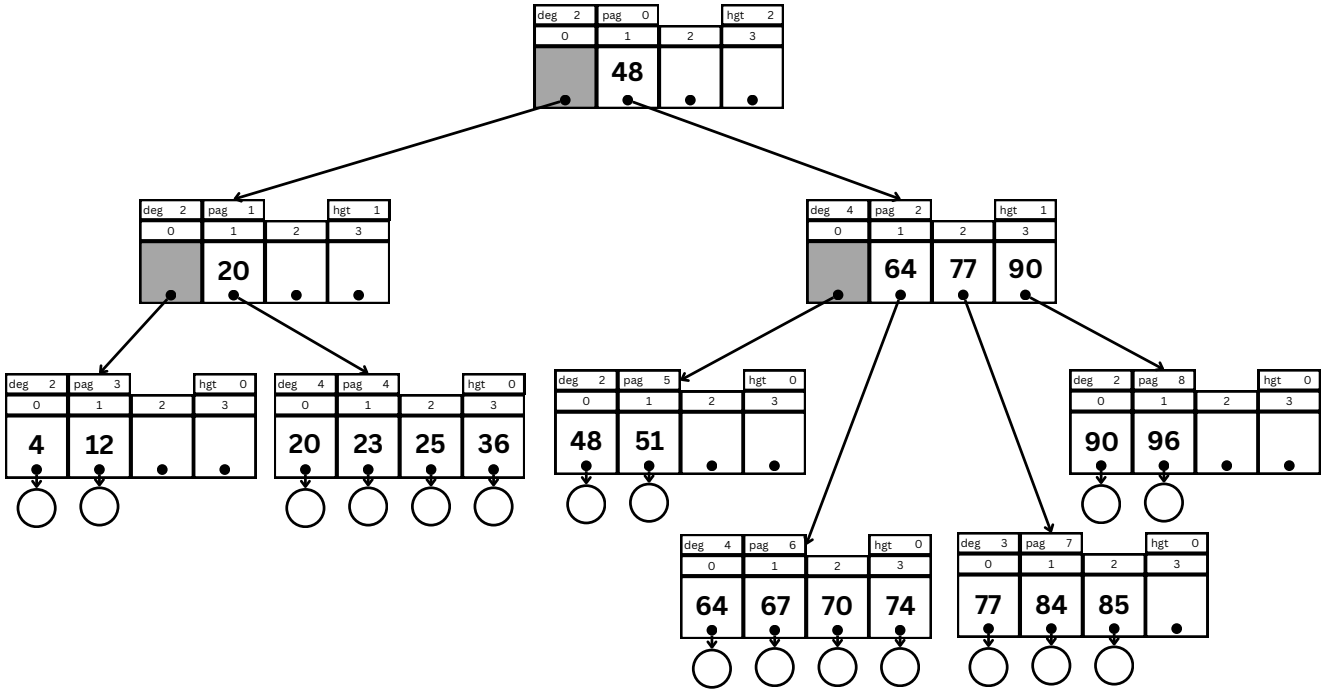


Figure: AB-Tree, $\tau\,(2, 4, 2)$

AB-Tree differences with B-Trees—Examples

> The main difference was already discussed, the α and β constants which define a different *Branching factor* in the AB-Trees

> And, as stated before, every B-Tree is a type of AB-Tree, which for example we can see that

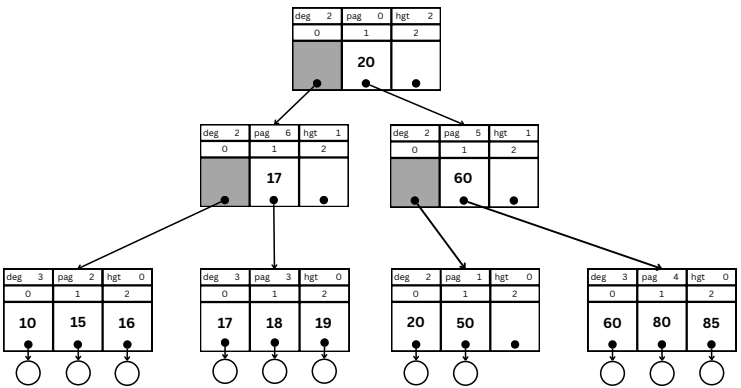


Figure: B-Tree, $t\,(2, 2)$

> Is a B-Tree with α equal to 2, but it's *Branching factor* falls under the definition of the minimun β value for a (α, β)–Tree.

> Which, for this tree, β would be 3, since $2\alpha - 1 = 3$.

> Then, this tree is also an $(2, 3)$–Tree.

AB-Tree differences with B-Trees—Examples

> But, also, we can see that not every AB-Tree is a B-Tree, for example

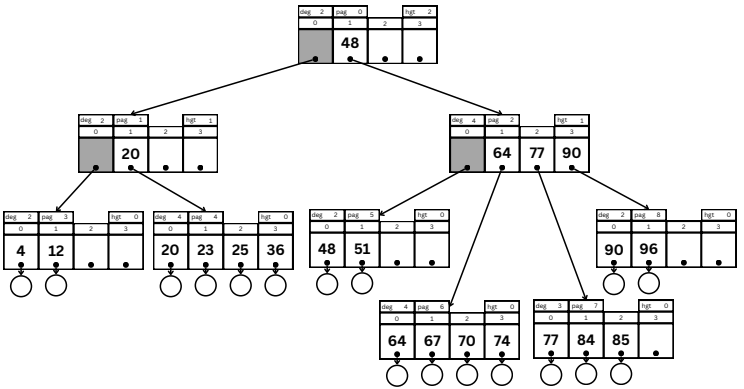


Figure: $(2,4)$-Tree or $\tau\,(2,4,2)$ AB-Tree

> Is a $(2,4)$-Tree with α and β equal to 2 and 4.

> But, as seen before, if the α of a B-Tree is equal to 2, then it's *Branching factor* would be different to the $(2,4)$–Tree.

> Hence, this AB-Tree is not a B-Tree.

AB-Tree differences with B-Trees—Code implementation

> In the implementation of a AB-Tree we will only replace the upper bound of the *Branching factor* by a `BETA` constant. For example, in the AB-Tree structure.

> It's important to define that this change of the upper bound won't change the rebalancing algorithms in a great way.

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

> In the `insert` operation, we can have some changes in some implementations.

> Theese changes happen mainly on the *Splitting*, of non-root pages, process in the rebalancing of the B-Tree.

AB-Tree differences with B-Trees—Code implementation

- In the *Splitting* process on a B-Tree, since we overflow the node, we will have 2α elements in the node, which we will split on the current node and a new node, resulting in two nodes with the minimun bound of α elements.
- But in AB-Trees, since for a node to overflow we could have the same or more elements, 2α, in the overflowing node we have to decide which node will have to take the extra elements after each node gets the minimun elements.
- In the current implementation, the balancing algorithm is just spliting the elements in half for each node, ending with two nodes with $\frac{\beta}{2}$ elements.
- In the current implementation there isn't any simple change to the `delete` operation.
- Since it depends mainly on the lower bound of the *Branching factor*, α, which is the same for both types of tree.

Bibliography

[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: NLngYyWFl_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).