B-Trees

Martín Hernández
Juan Mendivelso

Presentation made by Hernández, Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

Contents I

B-Tree History I

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 II

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

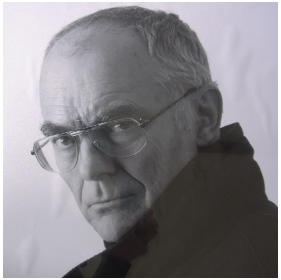


Figure: Rudolf Bayer

Figure: Edward McCreight

Presentation made by Hernández, Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

B-Tree Definition I

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

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

> 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 II

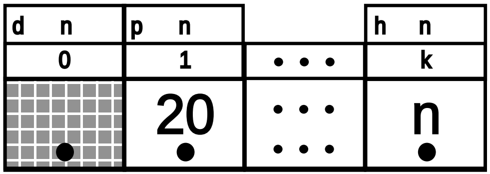


Figure: Node of a B-Tree

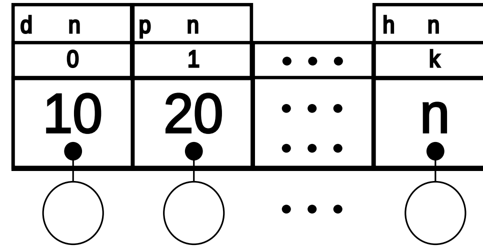


Figure: Leaf of a B-Tree

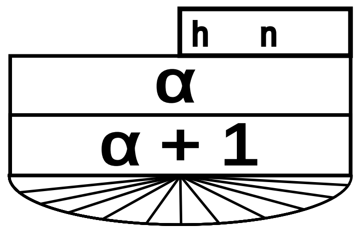


Figure: Generic Node of a B-Tree

B-Tree Properties—The α constant I

- 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.

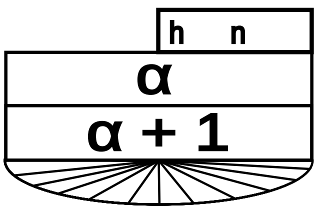


Figure: Miminum Keys and Sub-Trees on a Node

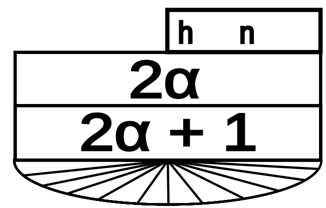


Figure: Maximun Keys and Sub-Trees on a Node

B-Tree Properties—The α constant II

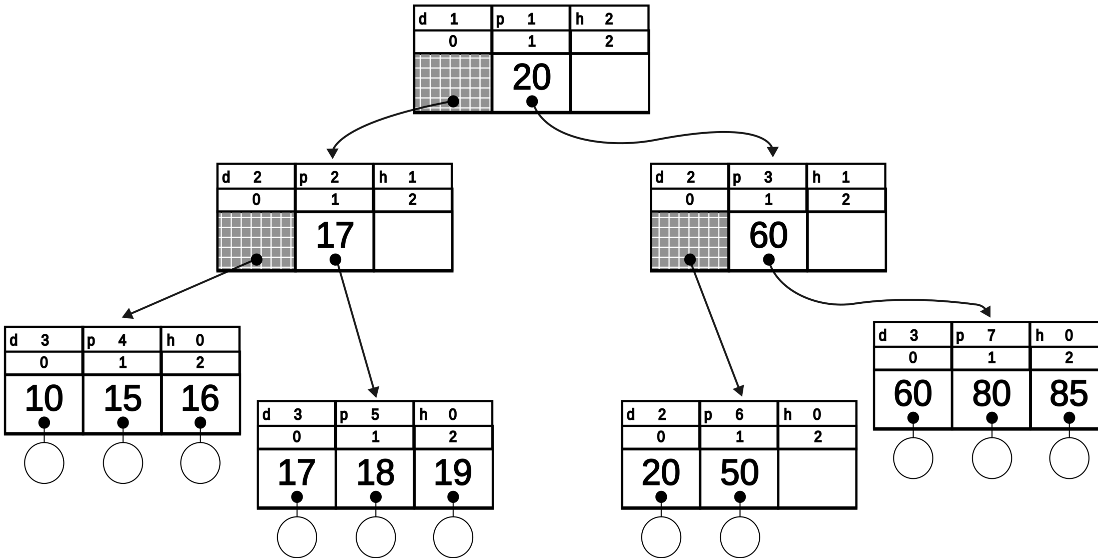


Figure: B-Tree, t (2, 2)

B-Tree Properties—The α constant III

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

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

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

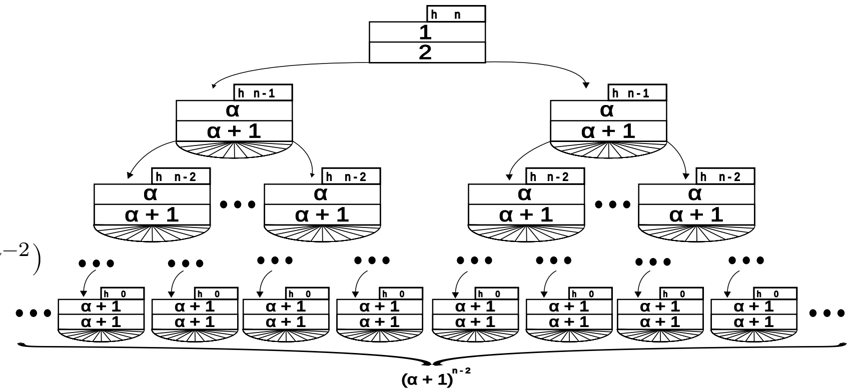


Figure: B-Tree w/ the least number of nodes

B-Tree Properties—The α constant IV

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

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

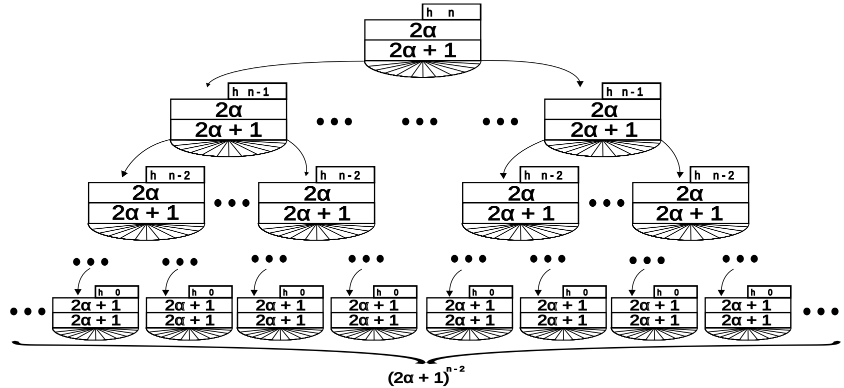


Figure: B-Tree w/ the most number of nodes

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

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

$\square$

B-Tree Properties—The α constant V

- 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 I

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

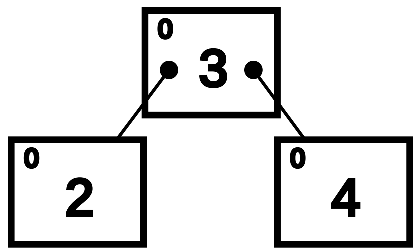


Figure: Simple node of a Normal Binary Tree

B-Tree Properties—Keys and Sub-trees II

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

> 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.

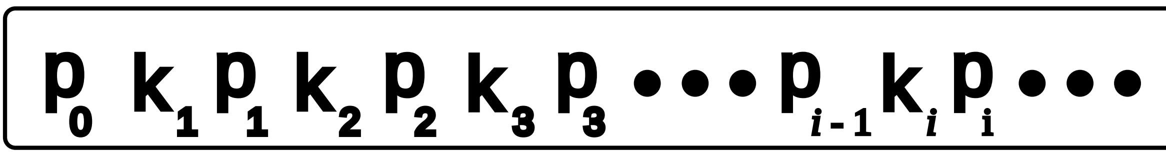


Figure: Order of the Subtree Pointers and Keys.

Presentation made by Hernández, Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

B-Tree Properties—Keys and Sub-trees III

> In the case that N is the *Root* of the tree, the only change is the minimum number of keys and sub-trees.

> With l, already defined, *Root* will have $1 \leq l \leq 2\alpha$ keys.

> And $2 \leq l + 1 \leq 2\alpha + 1$ sub-trees.

> If N is a leaf of the tree, we are going to give the k_0 a simple use.

> The k_0 will store a key value for an object.

> This simple usage on a leaf is just one usage of the k_0 on the nodes.

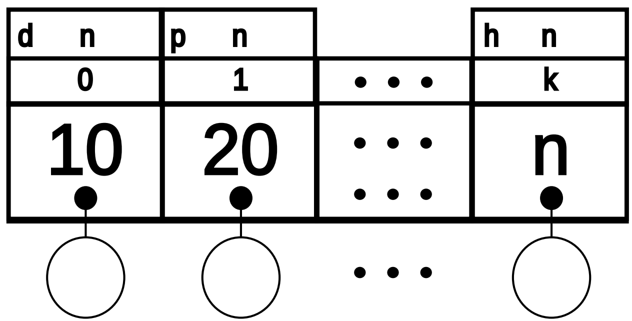


Figure: Leaf of a B-Tree

B-Tree Properties—Keys and Sub-trees IV

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

$$\forall y \in K(p_0); \quad y < k_1 \qquad \text{(Case 1)}$$
$$\forall y \in K(p_i); \quad k_i \leq y < k_{i+1}; \quad 0 < i < l \wedge i \in \mathbb{N} \qquad \text{(Case 2)}$$
$$\forall y \in K(p_l); \quad k_l \leq y \qquad \text{(Case 3)}$$

Presentation made by Hernández, Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

B-Tree Properties—Keys and Sub-trees V

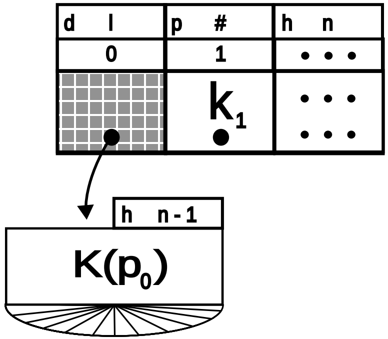


Figure: Sub-tree Keys (Case 1)

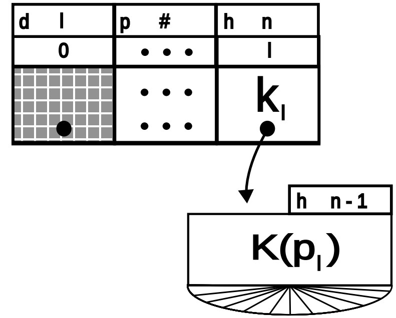


Figure: Sub-tree Keys (Case 3)

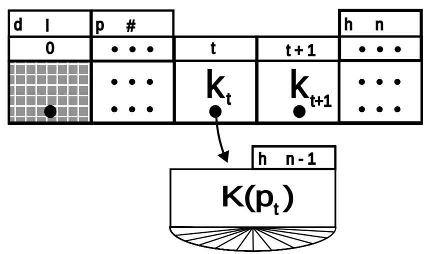


Figure: Sub-tree Keys (Case 2)

B-Tree Properties—Height I

> Before we can define and prove the height of a B-Tree we need to define some things.

> First, The set of the keys in $T \in t(\alpha, h)$ will be defined as I.

> Now, The I_{min} and I_{max} of T can be easily defined by (Nodes Bounds):

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

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

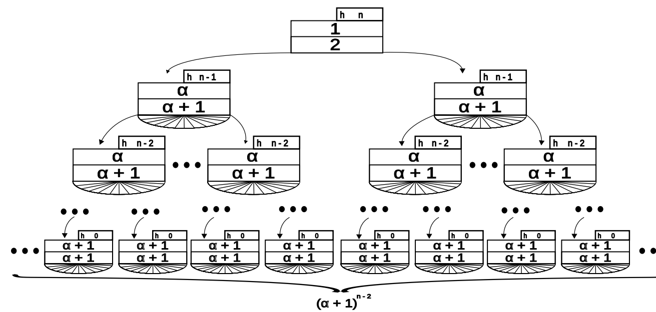


Figure: B-Tree w/ the least number of nodes

Presentation made by Hernández, Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

B-Tree Properties—Height II

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

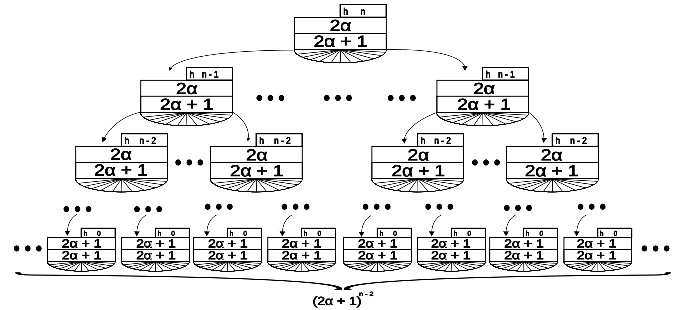


Figure: B-Tree w/ the most number of nodes

> 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 III

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

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

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

B-Tree Properties—Summary

> A B-Tree is defined as: $T \in t(\alpha, h)$
> A B-Tree has a predefined constant 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:

```
#define ALPHA 2 /* any int >= 2 */
typedef struct tr_n_t {
  int degree;
  int height;
  key_t key[(2 * ALPHA) - 1];
  struct tr_n_t *next[(2 * ALPHA) - 1];
  /* ... */
} 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.

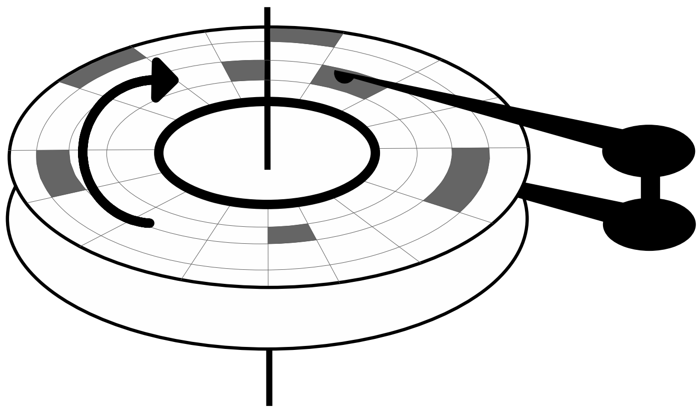


Figure: External storage with the sectors to access highlighted

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 $\Theta(1)$.

```
tree_node_t *create_tree() {
  tree_node_t *tmp;
  tmp = get_node();
  tmp->height = 0;
  tmp->degree = 0;
  return( tmp );
}
```


Presentation made by Hernández, Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

B-Tree Operations—Search I

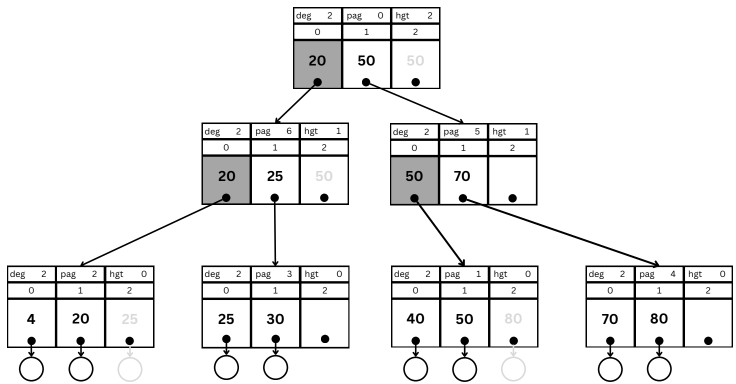
> The changes of this operations are mainly focused on the search part, since we have to compare to an array of keys and not only the node key.

> This operation returns the object in the B-Tree if a given key exists.

```
object_t *find(tree_node_t *tree, key_t query_key) {
  tree_node_t *current_node;
  object_t *object;
  current_node = tree;

  while( current_node->height >= 0 ) {
    /* binary search among keys */
    int lower, upper;
    lower = 0;
    upper = current_node->degree;

    while( upper > lower +1 ) {
      int med = (upper+lower)/2;
      if( query_key < current_node->key[med] )
        upper = med;
      else
        lower = med;
    }
    if( current_node->height > 0)
      current_node = current_node->next[lower];

    else {
```

B-Tree Operations—Search II

```
      if( current_node->key[lower] == query_key )
        object = (object_t *) current_node->next[lower];
      else
        object = NULL;
      return( object );
    }
  }
}
```

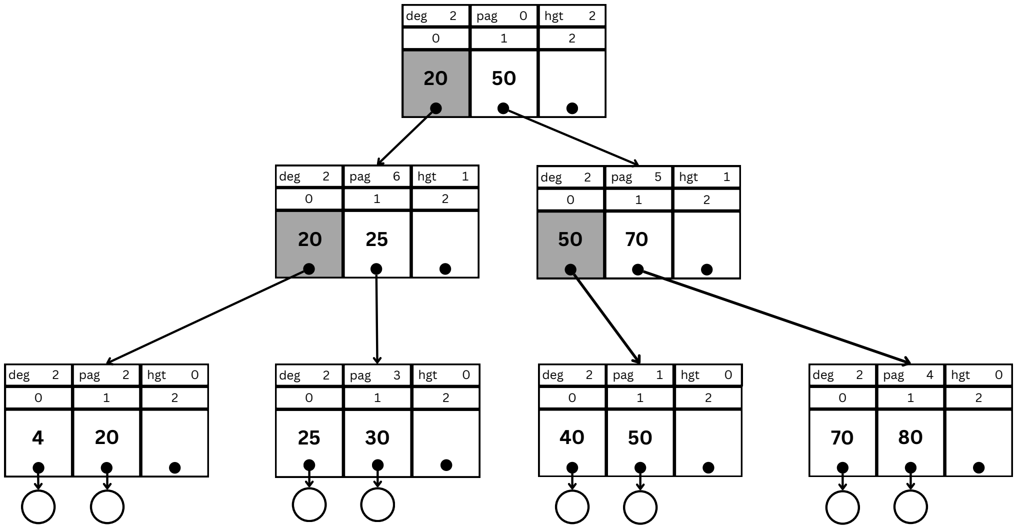


> Let's search for 70 in this $t\,(2,2)$ B-Tree.

B-Tree Operations—Search (Example) I

```
object_t *find(tree_node_t *tree, key_t query_key) {
  tree_node_t *current_node;
  object_t *object;
  current_node = tree;

```

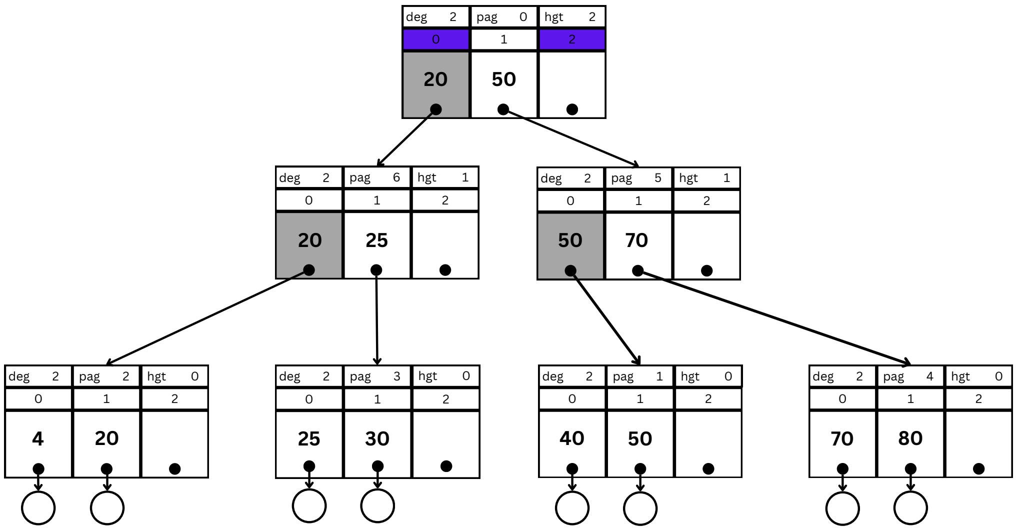
> Search 1; Step 1;
> tree=(*pag 0); query_key=70;
> object;
> current_node=(*pag 0);



B-Tree Operations—Search (Example) II

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

```

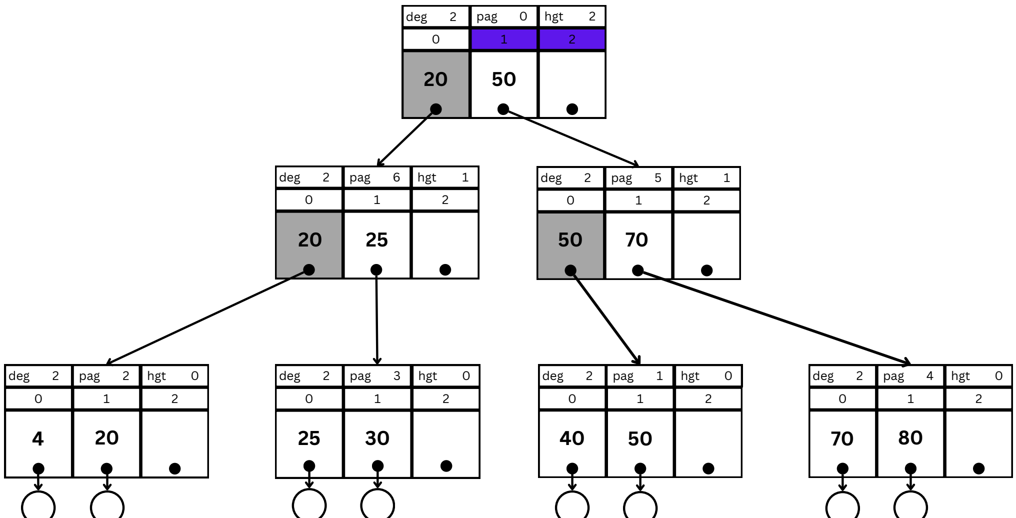
> Search 1; Step 2;
> tree=(*pag 0); query_key=70;
> object; lower=0; upper=2;
> current_node=(*pag 0);



B-Tree Operations—Search (Example) III

```
while( upper > lower +1 ) {
  int med = (upper+lower)/2;
  if( query_key < current_node->key[med] )
    upper = med;
  else
    lower = med;
}
```

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



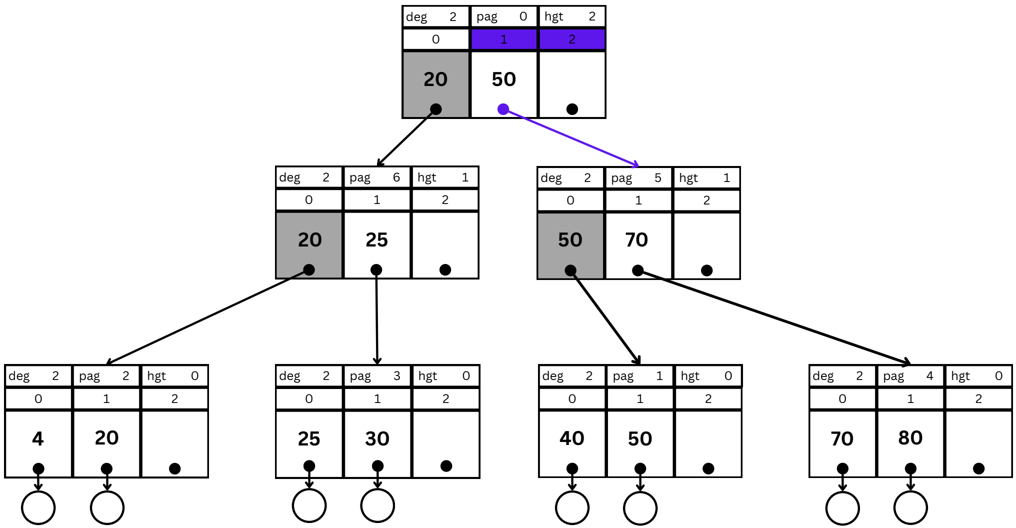
Presentation made by Hernández, Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

B-Tree Operations—Search (Example) IV

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

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

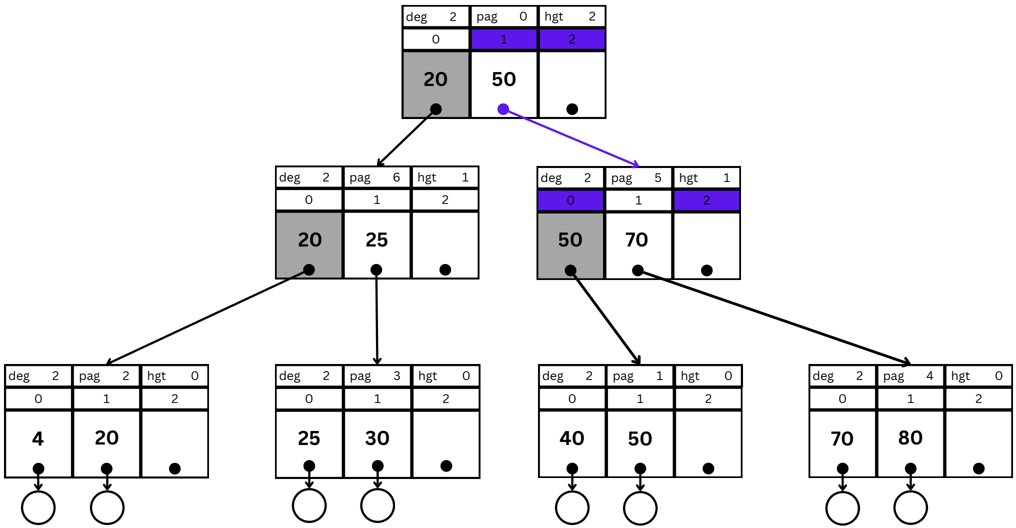
> Search 1; Step 4;
> tree=(*pag 0); query_key=70;
> object; lower=1; upper=2;
> current_node=(*pag 0) → (*pag 5);



B-Tree Operations—Search (Example) V

```c
while( current_node->height >= 0 ) {
  /* binary search among keys */
  int lower, upper;
  lower = 0;
  upper = current_node->degree;
```

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

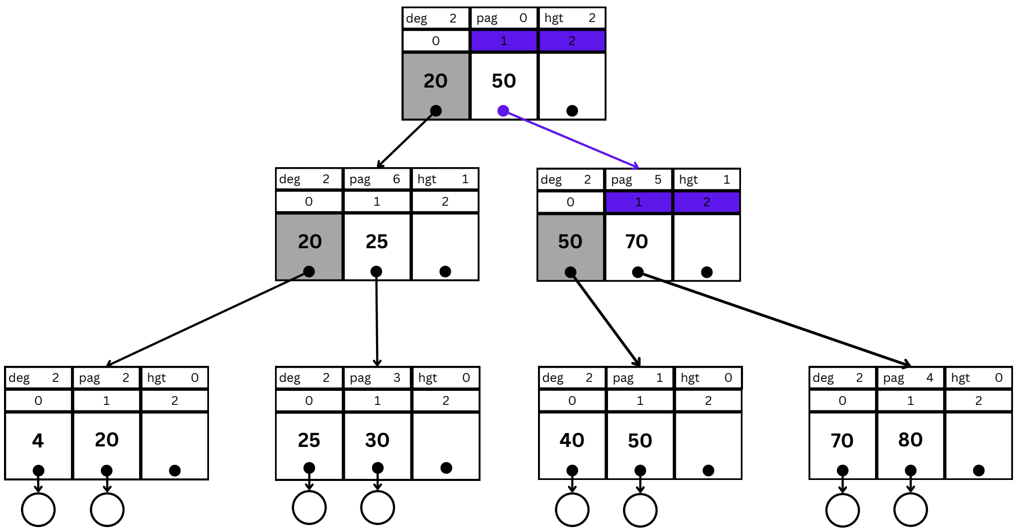


Presentation made by Hernández, Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

B-Tree Operations—Search (Example) VI

```
while( upper > lower +1 ) {
  int med = (upper+lower)/2;
  if( query_key < current_node->key[med] )
    upper = med;
  else
    lower = med;
}
```

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

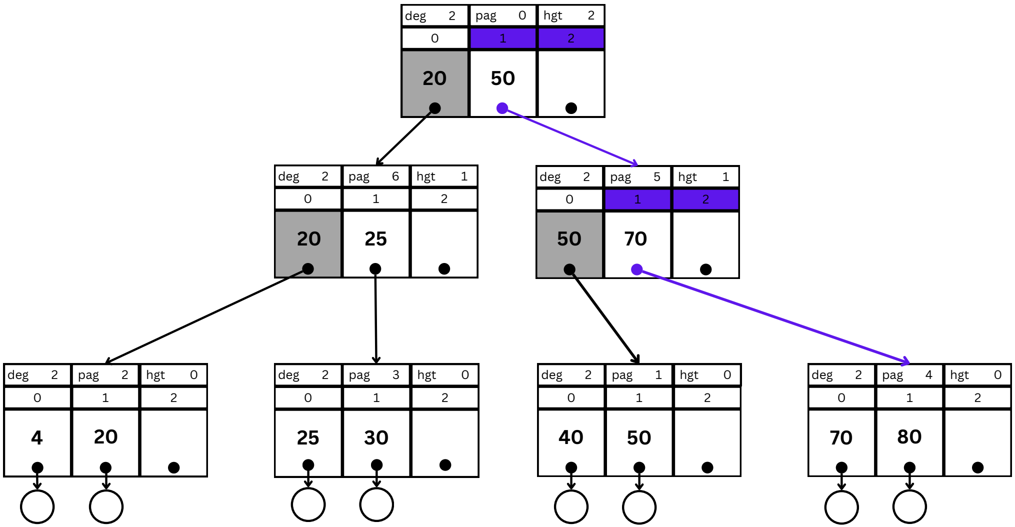


B-Tree Operations—Search (Example) VII

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

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

```

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

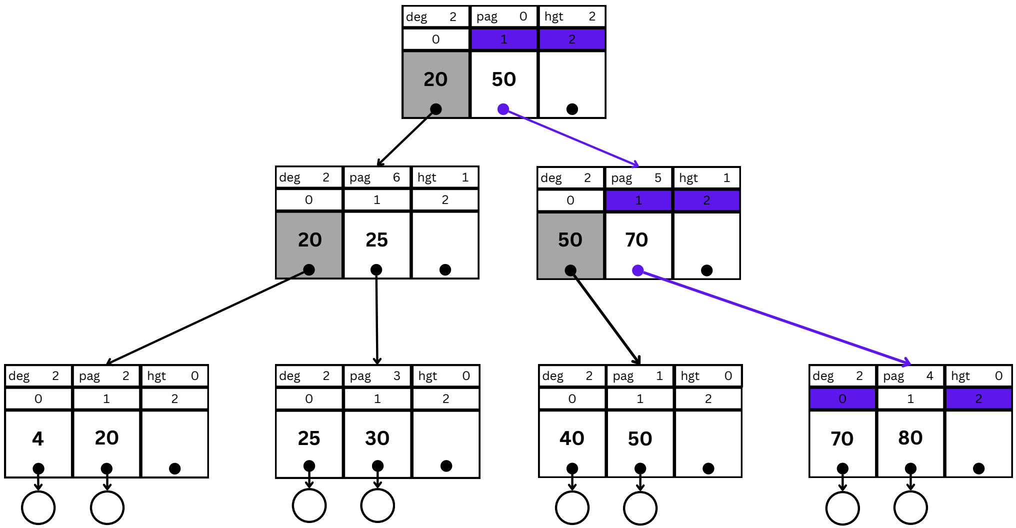


Presentation made by Hernández, Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

B-Tree Operations—Search (Example) VIII

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

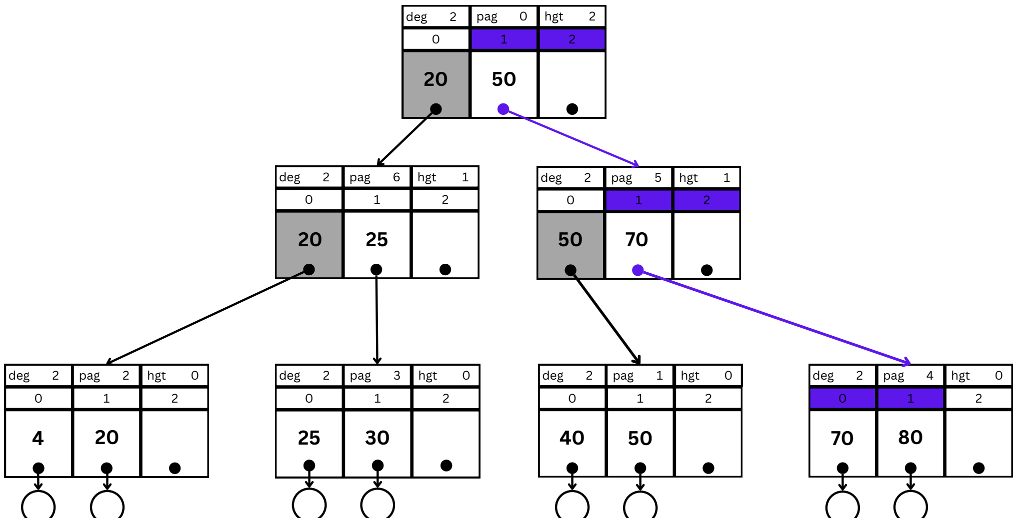
> Search 1; Step 8;
> tree=(*pag 0); query_key=70;
> object; lower=0; upper=2;
> current_node=(*pag 4);



B-Tree Operations—Search (Example) IX

```
while( upper > lower +1 ) {
  int med = (upper+lower)/2;
  if( query_key < current_node->key[med] )
    upper = med;
  else
    lower = med;
}
```

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



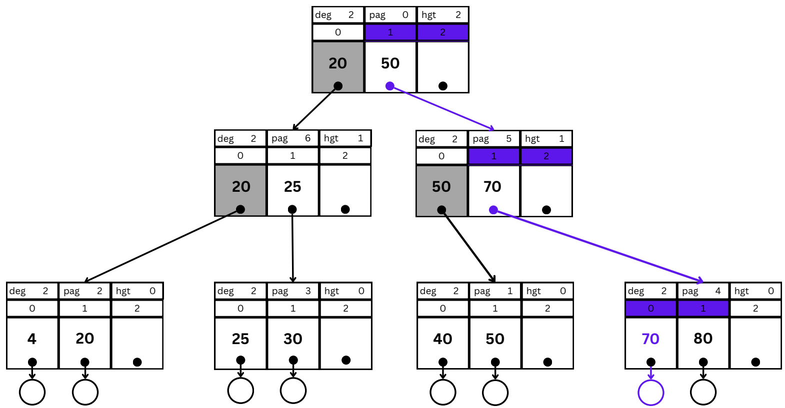
B-Tree Operations—Search (Example) X

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

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

```
else {
  if( current_node->key[lower] == query_key )
    object = (object_t *) current_node->next[lower];
  else
    object = NULL;
  return( object );
}
```

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



Presentation made by Hernández, Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

B-Tree Operations—Delete I

- The deletion algorithm, just like the insert or find, in the B-Tree almost has nothing to share with any tree deletion algorithm.
- Also, the first part is a `find` algorithm where we are going to search if the key to delete exists and if it does and 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 II

```
object_t *delete(tree_node_t *tree, key_t delete_key) {
    tree_node_t *current, *tmp_node;
    int finished, i, j;
    current = tree;
    create_node_stack();
    create_index_stack();
    while( current->height > 0 ) {
        /* not at leaf level */
        int lower, upper;
        /* binary search among keys */
        lower = 0;
        upper = current->degree;
        while( upper > lower +1 ) {
            if( delete_key < current->key[ (upper+lower)/2 ] )
                upper = (upper+lower)/2;
            else
                lower = (upper+lower)/2;
        }

        push_index_stack( lower );
        push_node_stack( current );
        current = current->next[lower];
    }
    /* now current is leaf node from which we delete */
    for ( i=0; i < current->degree ; i++ )
        if( current->key[i] == delete_key )
```


Presentation made by Hernández, Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

B-Tree Operations—Delete III

```
            break;
    if( i == current->degree ) {
        /* delete failed; key does not exist */
        return( NULL );
    } else {
        /* key exists, now delete from leaf node */
        object_t *del_object;
        del_object = (object_t *) current->next[i];
        current->degree -=1;
        while( i < current->degree ) {
            current->next[i] = current->next[i+1];
            current->key[i] = current->key[i+1];
            i+=1;
        }
        /* deleted from node, now rebalance */
        finished = 0;
        while( ! finished ) {
            if(current->degree >= ALPHA ) {
                finished = 1;
                /* node still full enough, can stop */
            }
            else {
                /* node became underfull */
                if( stack_empty() ) {
                    /* current is root */
                    if(current->degree >= 2 )
                        /* root still necessary */
```

B-Tree Operations—Delete IV

```c
                        finished = 1;
                    else if ( current->height == 0 )
                        /* deleting last keys from root */
                        finished = 1;
                    else {
                        /* delete root, copy to keep address */
                        tmp_node = current->next[0];
                        for( i=0; i< tmp_node->degree; i++ ) {
                            current->next[i] = tmp_node->next[i];
                            current->key[i] = tmp_node->key[i];
                        }
                        current->degree =
                            tmp_node->degree;
                        current->height =
                            tmp_node->height;
                        return_node( tmp_node );
                        finished = 1;
                    }
                    /* done with root */
                } else {
                    /* delete from non-root node */
                    tree_node_t *upper, *neighbor;
                    int curr;
                    upper = pop_node_stack();
                    curr = pop_index_stack();
                    if( curr < upper->degree -1 ) {
                        /* not last*/
```

B-Tree Operations—Delete V

```
                neighbor = upper->next[curr+1];
                if( neighbor->degree > ALPHA ) {
                    /* sharing possible */
                    i = current->degree;
                    if( current->height > 0 )
                        current->key[i] =
                            upper->key[curr+1];
                    else {
                        /* on leaf level, take leaf key */
                        current->key[i] =
                            neighbor->key[0];
                        neighbor->key[0] =
                            neighbor->key[1];
                    }
                    current->next[i] =
                        neighbor->next[0];
                    upper->key[curr+1] =
                        neighbor->key[1];
                    neighbor->next[0] =
                        neighbor->next[1];
                    for( j = 2; j < neighbor->degree; j++) {
                        neighbor->next[j-1] =
                            neighbor->next[j];
                        neighbor->key[j-1] =
                            neighbor->key[j];
                    }
                    neighbor->degree -=1;
```


Presentation made by Hernández, Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

B-Tree Operations—Delete VI

```
                current->degree+=1;
                finished =1;
            } /* sharing complete */
            else {
                /* must join */
                i = current->degree;
                if( current->height > 0 )
                    current->key[i] =
                        upper->key[curr+1];
                else /* on leaf level, take leaf key */
                    current->key[i] =
                        neighbor->key[0];
                current->next[i] =
                    neighbor->next[0];
                for( j = 1; j < neighbor->degree; j++) {
                    current->next[++i] =
                        neighbor->next[j];
                    current->key[i] =
                        neighbor->key[j];
                }
                current->degree = i+1;
                return_node( neighbor );
                upper->degree -=1;
                i = curr+1;
                while( i < upper->degree ) {
                    upper->next[i] =
                        upper->next[i+1];
```

B-Tree Operations—Delete VII

```
                        upper->key[i] =
                            upper->key[i+1];
                        i +=1;
                    }
                    /* deleted from upper, now propagate up */
                    current = upper;
                } /* end of share/joining if-else*/
            }
            else {
                /* current is last entry in upper */
                neighbor = upper->next[curr-1]
                    if( neighbor->degree > ALPHA ) {
                        /* sharing possible */
                        for( j = current->degree; j > 1; j--) {
                            current->next[j] =
                                current->next[j-1];
                            current->key[j] =
                                current->key[j-1];
                        }
                        current->next[1] =
                            current->next[0];
                        i = neighbor->degree;
                        current->next[0] =
                            neighbor->next[i-1];
                        if( current->height > 0 ) {
                            current->key[1] =
                                upper->key[curr];
```


Presentation made by Hernández, Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

B-Tree Operations—Delete VIII

```
    }
    else {
        /* on leaf level, take leaf key */
        current->key[1] =
            current->key[0];
        current->key[0] =
            neighbor->key[i-1];
    }
    upper->key[curr] =
        neighbor->key[i-1];
    neighbor->degree -=1;
    current->degree+=1;
    finished =1;
} /* sharing complete */
else {
    /* must join */
    i = neighbor->degree;
    if( current->height > 0 )
        neighbor->key[i] =
            upper->key[curr];
    else /* on leaf level, take leaf key */
        neighbor->key[i] =
            current->key[0];
    neighbor->next[i] =
        current->next[0];
    for( j = 1; j < current->degree; j++) {
        neighbor->next[++i] =
```

B-Tree Operations—Delete IX

```
                                          current->next[j];
                                      neighbor->key[i] =
                                          current->key[j];
                                  }
                                  neighbor->degree = i+1;
                                  return_node( current );
                                  upper->degree -=1;
                                  /* deleted from upper, now propagate up */
                                  current = upper;
                              } /* end of share/joining if-else */
                      } /* end of current is (not) last in upper if-else*/
                  } /* end of delete root/non-root if-else */
              } /* end of full/underfull if-else */
          } /* end of while not finished */

          return( del_object );

      } /* end of delete object exists if-else */
}
```

B-Tree Operations—Delete (Example) I

B-Tree Secondary Memory Access I

- The B-Tree is fairly good for storing data in external memory in comparison to height, weight or search trees.
- The limit of 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 $\Theta\left(\alpha \log n\right)$, 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.

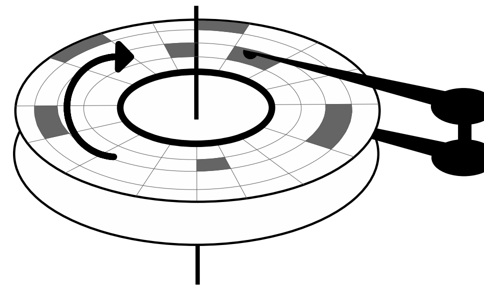


Figure: External storage with the sectors to access highlighted

Presentation made by Hernández, Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

B-Tree Secondary Memory Access II

|	Retrival	Insertion w/ overflow	Deletion w/ underfull
Ω|	$t = 1\ w = 0$	$t = h\ w = 1$	$t = h\ w = 1$
Θ|	$t \leq h\ w = 0$	$t \leq h + 2 + \frac{2}{\alpha}\ w \leq 3 + \frac{2}{\alpha}$	$t \leq 3h - 2\ w \leq 2h + 1$
O|	$t = h\ w = 0$	$t = 3h - 2\ w = 2h + 1$	$t = 3h - 2\ w = 2h + 1$

$>$ Where t is the number of fetch and readings of nodes on the secondary memory.

$>$ And w is the number of writings of nodes on the secondary memory.

[1]

Bibliography I

[1] R. Bayer and E. M. McCreight. “Organization and maintenance of large ordered indexes”. In: *Acta Informatica* 1.3 (1972), pp. 173–189. ISSN: 0001-5903, 1432-0525. DOI: 10.1007/BF00288683. URL: http://link.springer.com/10.1007/BF00288683 (visited on 10/17/2024).

[2] Peter Brass. *Advanced Data Structures*. Google-Books-ID: g8rZoSKLbhwC. Cambridge University Press, Sept. 8, 2008. 456 pp. ISBN: 978-0-521-88037-4.

[3] Thomas H. Cormen et al. *Introduction To Algorithms*. Google-Books-ID: 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).