B-Trees & AB-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)

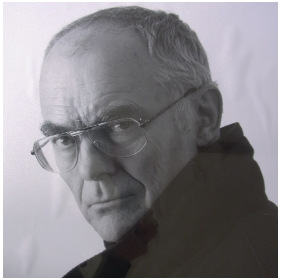

Figure: Rudolf Bayer

"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

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

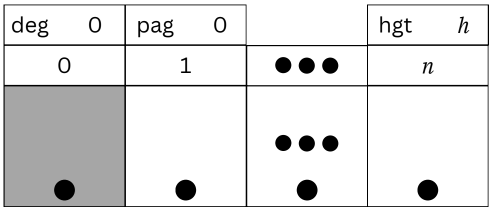

Figure: Node of a B-Tree

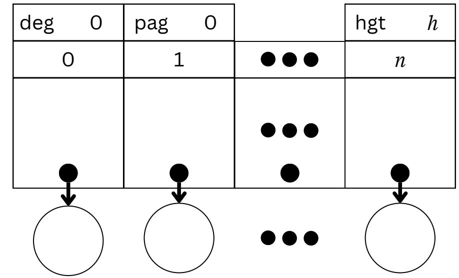

Figure: Leaf of a B-Tree

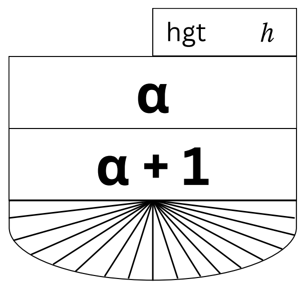

Figure: Generic Node of a B-Tree

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.

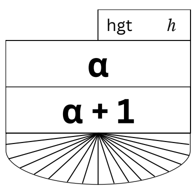

Figure: Miminum Keys and Sub-Trees on a Node

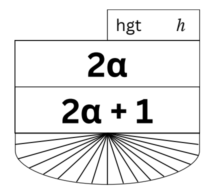

Figure: Maximun Keys and Sub-Trees on a Node

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

B-Tree Properties—The α constant II

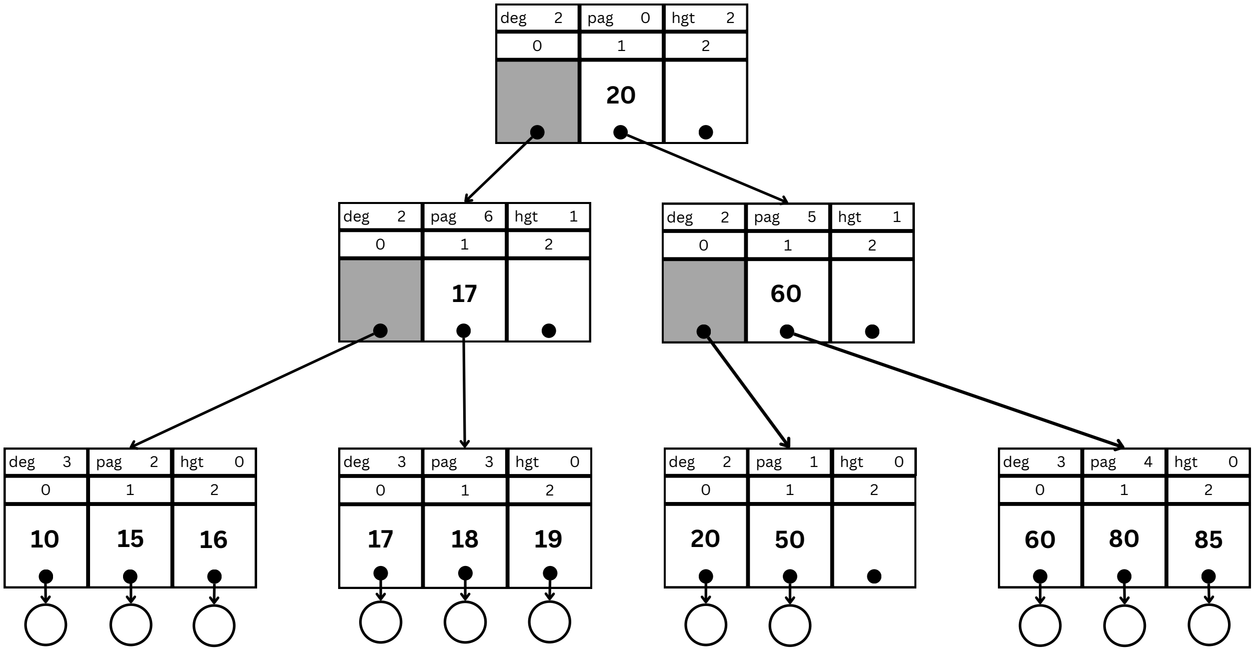

Figure: B-Tree, $t\,(2,2)$

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

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

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

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

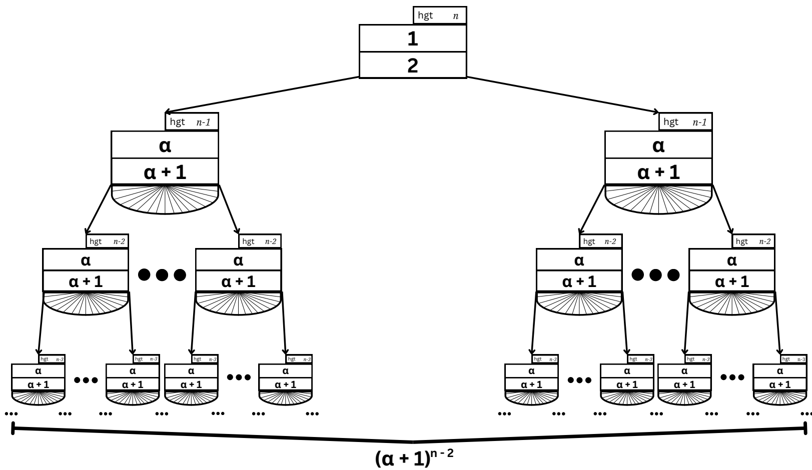

Figure: B-Tree w/ the least number of nodes

B-Tree Properties—The α constant IV

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

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

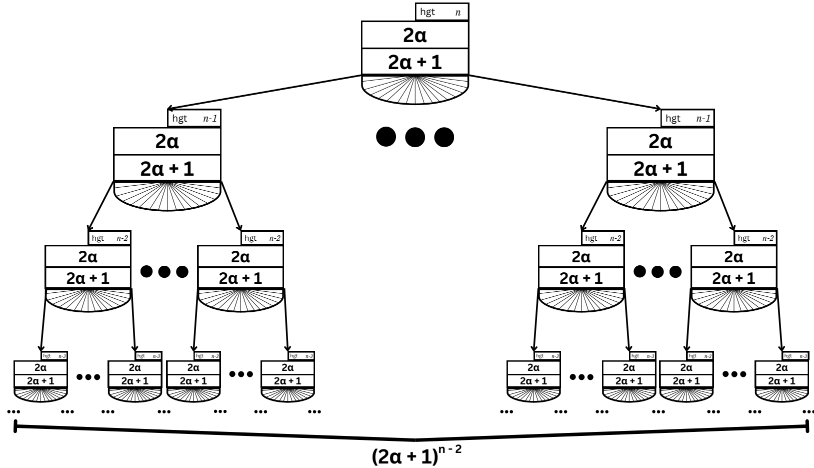

Figure: B-Tree w/ the most number of nodes

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

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

$\square$

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

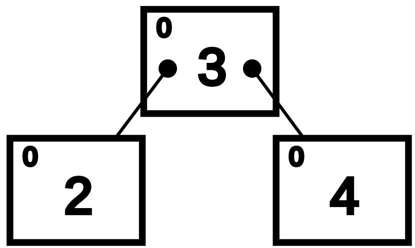

Figure: Simple node of a Normal Binary Tree

B-Tree Properties—Keys and Sub-trees 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.

$$\boxed{p_0\ k_1\ p_1\ k_2\ p_2\ k_3\ p_3 \bullet\bullet\bullet\ p_{i-1}\ k_i\ p_i \bullet\bullet\bullet}$$

Figure: Order of the Subtree Pointers and Keys.

B-Tree Properties—Keys and Sub-trees III

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

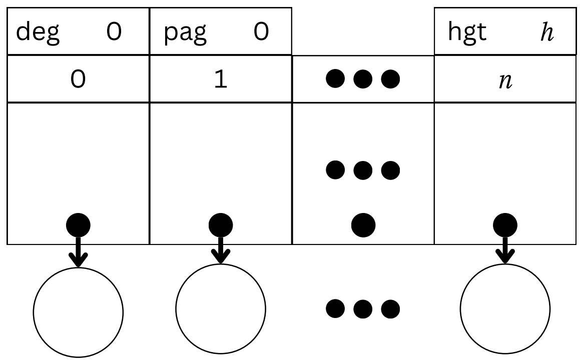

Figure: Leaf of a B-Tree

B-Tree Properties—Keys and Sub-trees IV

> Going back where N is a node on the B-Tree, but now this time N can be the tree *Root*.

> The order of the keys of p_i, a subtree of N; where $0 \leq i \leq l$, in comparison to the keys of N can be defined by 3 cases.

> But first, we need to define $K(T)$, where $T \in t(\alpha, h)$, which is the set of keys inside the Node T.

> And, $k_j \in K(N)$, where j is the index or position of the key in N.

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

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

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

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

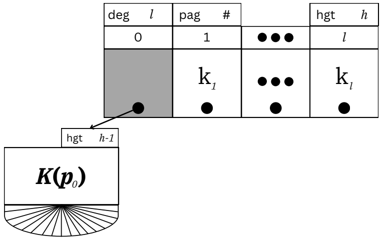

Figure: Sub-tree Keys (Case 1)

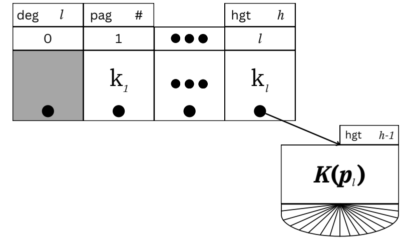

Figure: Sub-tree Keys (Case 3)

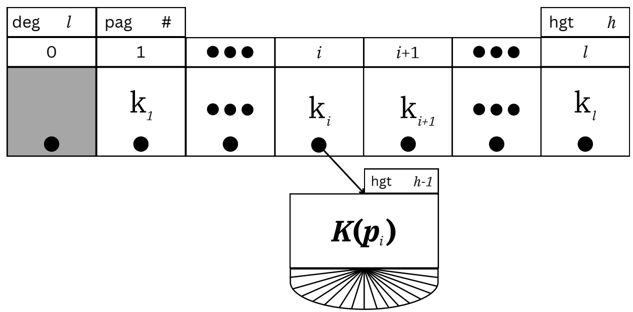

Figure: Sub-tree Keys (Case 2)

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):

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

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

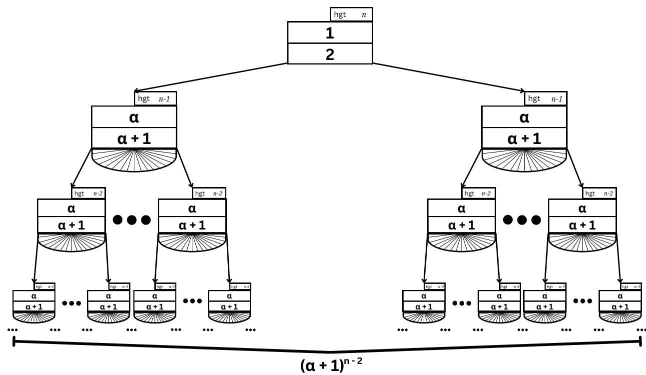

Figure: B-Tree w/ the least number of nodes

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

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

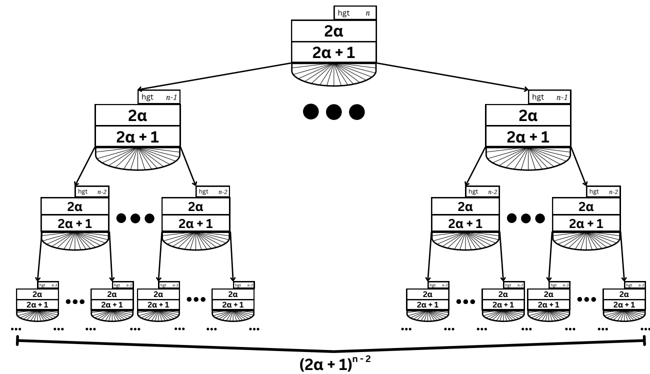

Figure: B-Tree w/ the most number of nodes

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

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

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

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 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:

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

> 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 Operations—Insert Value I

- The insertion algorithm in the B-Tree almost has nothing to share with any tree insertion algorithm.
- The first section of the code is the same `find` algorithm so we can see if the value to add is already stored in the B-Tree and where could it be stored, also storing in a stack the nodes that we are going to access.
- Then, if the node isn't full yet, we are just going to move everything by an index until the current elements are less than the key that we are going to insert.
- But, if the node is full, we will get a new node for the B-Tree and split in half the full node.
- Then, insert the new key into one of 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.**

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

B-Tree Operations—Insert Value II

```
int insert(tree_node_t *tree, key_t new_key, object_t *new_object) {
  tree_node_t *current_node, *insert_pt;
  key_t insert_key;
  int finished;
  current_node = tree;
  if( tree->height == 0 && tree->degree == 0 ) {
    tree->key[0] = new_key;
    tree->next[0] = (tree_node_t *) new_object;
    tree->degree = 1;
    return(0); /* insert in empty tree */
  }

  create_stack();
  while( current_node->height > 0 ) {
    int lower, upper;
    /* binary search among keys */
    push( current_node );
    lower = 0;
    upper = current_node->degree;
    while( upper > lower +1 ) {
      if( new_key < current_node->key[(upper+lower)/2 ] )
        upper = (upper+lower)/2;
      else
        lower = (upper+lower)/2;
    }
    current_node = current_node->next[lower];
```

B-Tree Operations—Insert Value III

```
  }
  /* now current_node is leaf node in which we insert */

  insert_pt = (tree_node_t *) new_object;
  insert_key = new_key;
  finished = 0;
  while( !finished ){
    int i, start;
    if( current_node->height > 0 )
      start = 1;
    /* insertion in non-leaf starts at 1 */
    else
      start = 0;
    /* insertion in leaf starts at 0 */
    /* node still has room */
    if( current_node->degree < (2 * ALPHA) - 1) {
      /* move everything up to create the insertion gap */
      i = current_node->degree;
      while( (i > start) && (current_node->key[i-1] > insert_key)) {
        current_node->key[i] = current_node->key[i-1];
        current_node->next[i] = current_node->next[i-1];
        i -= 1;
      }

      current_node->key[i] = insert_key;
      current_node->next[i] = insert_pt;
      current_node->degree +=1;
```

B-Tree Operations—Insert Value IV

```
        finished = 1;
      } /* end insert in non-full node */
      else {
        /* node is full, have to split the node*/
        tree_node_t *new_node;
        int j, insert_done = 0;
        new_node = get_node();
        i = ((2 * ALPHA) - 1)-1;
        j = (((2 * ALPHA) - 1)-1)/2;
        while( j >= 0 ) {
          /* copy upper half to new node */
          if( insert_done || insert_key < current_node->key[i] ) {
            new_node->next[j] =
              current_node->next[i];
            new_node->key[j--] =
              current_node->key[i--];
          } else {
            new_node->next[j] = insert_pt;
            new_node->key[j--] = insert_key;
            insert_done = 1;
          }
        }
        /* upper half done, insert in lower half, if necessary*/
        while( !insert_done) {
          if( insert_key < current_node->key[i] && i >= start ) {
            current_node->next[i+1] =
              current_node->next[i];
```


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

B-Tree Operations—Insert Value V

```
            current_node->key[i+1] =
              current_node->key[i];
            i -=1;
          } else {
            current_node->next[i+1] =
              insert_pt;
            current_node->key[i+1] =
              insert_key;
            insert_done = 1;
          }
        }
        /*finished insertion */

        current_node->degree = ((2 * ALPHA) - 1)+1 - ((((2 * ALPHA) - 1)+1)/2);
        new_node->degree = (((2 * ALPHA) - 1)+1)/2;
        new_node->height = current_node->height;
        /* split nodes complete, now insert the new node above */
        insert_pt = new_node;
        insert_key = new_node->key[0];
        if( ! stack_empty() ) {
          /* not at root; move one level up*/
          current_node = pop();
        } else {
          /* splitting root: needs copy to keep root address*/
          new_node = get_node();
          for(i = 0; i < current_node->degree; i++) {
            new_node->next[i] =
```


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

B-Tree Operations—Insert Value VI

```
              current_node->next[i];
            new_node->key[i] =
              current_node->key[i];
          }
          new_node->height =
            current_node->height;
          new_node->degree =
            current_node->degree;
          current_node->height += 1;
          current_node->degree = 2;
          current_node->next[0] = new_node;
          current_node->next[1] = insert_pt;
          current_node->key[1] = insert_key;
          finished =1;
        } /* end splitting root */
      } /* end node splitting */
    } /* end of rebalancing */
    remove_stack();
    return( 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—Insert Value VII

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—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.

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 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 )
            break;
```

B-Tree Operations—Delete III

```
    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 */
                        finished = 1;
                    else if ( current->height == 0 )
```

B-Tree Operations—Delete IV

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

B-Tree Operations—Delete V

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

B-Tree Operations—Delete VI

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


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 VII

```
                            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];
                                }
                                else {
                                    /* on leaf level, take leaf key */
                                    current->key[1] =
                                        current->key[0];
                                    current->key[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—Delete VIII

```
                        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] =
                        current->next[j];
                    neighbor->key[i] =
                        current->key[j];
                }
                neighbor->degree = i+1;
                return_node( current );
                upper->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 IX

```
                                        /* 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 */
}
```

AB-Tree Definition I

> 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.

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

AB-Tree The α & β constants I

> 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.

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

AB-Tree The α & β constants II

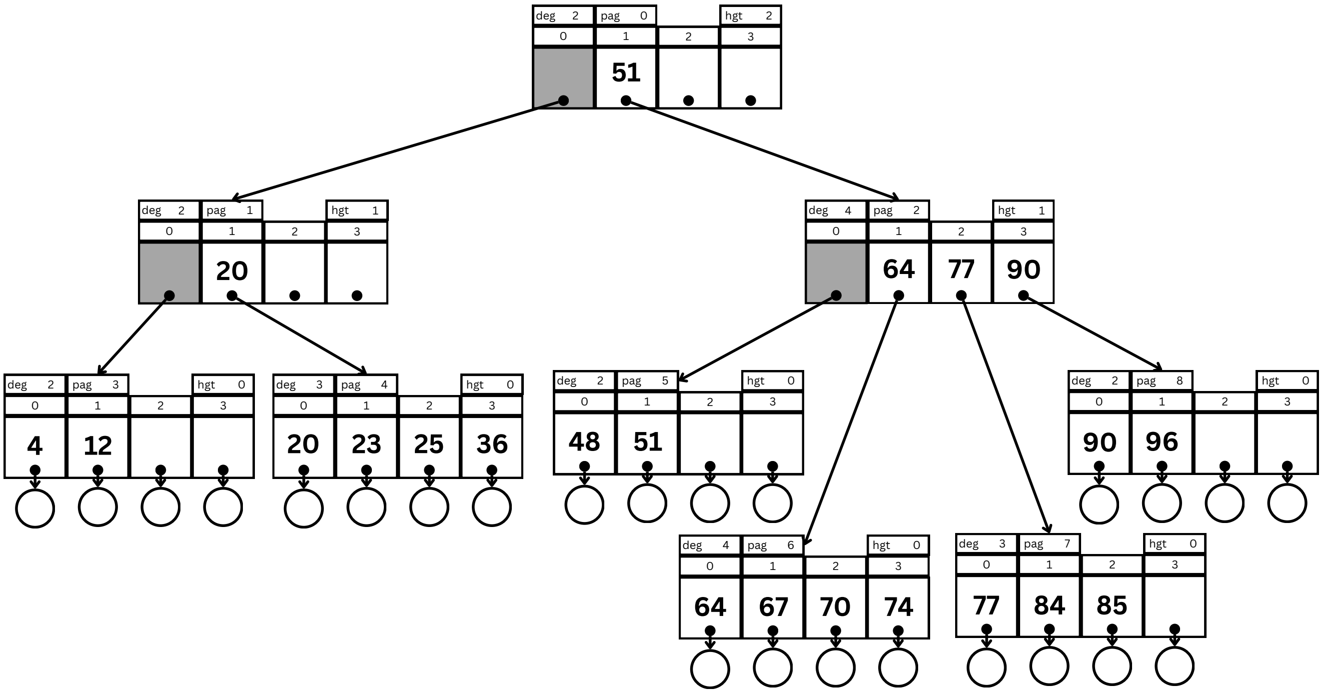

Figure: AB-Tree, $\tau(2,4,2)$

AB-Tree differences with B-Trees—Examples I

> 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

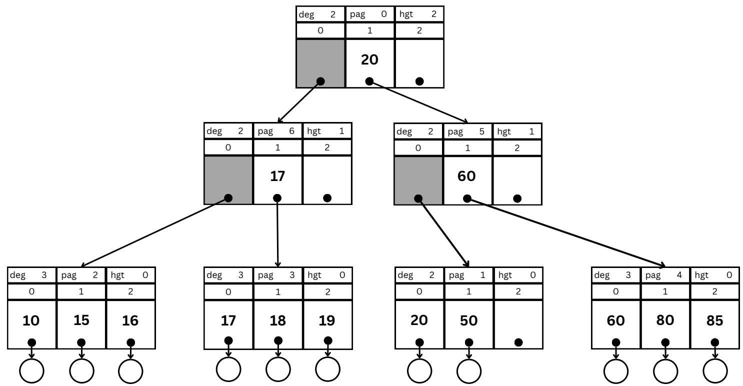

Figure: B-Tree, $t\,(2,2)$

> 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 II

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

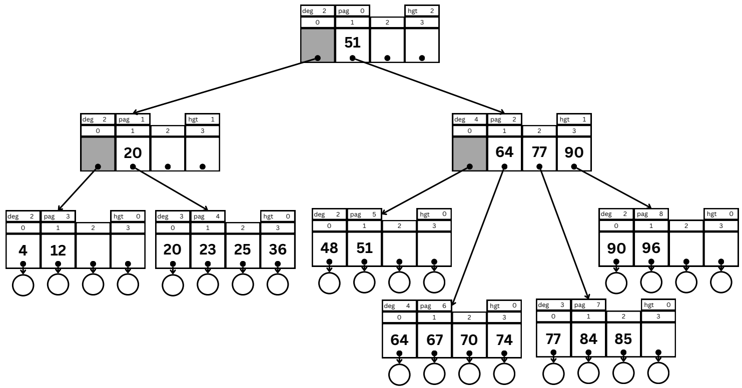

Figure: $(2, 4)$-Tree or $\tau(2, 4, 2)$ AB-Tree

> 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 I

- 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.

```c
#define ALPHA 2
#define BETA 4 /* any int >= (2 * ALPHA) - 1*/
typedef struct tr_n_t {
  int degree;
  int height;
  key_t key[BETA];
  struct tr_n_t *next[BETA];
  /* ... */
} 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 II

- 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 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).