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)

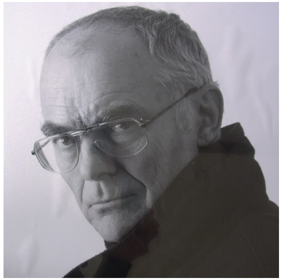


Figure: Rudolf Bayer

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.

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

- 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 II

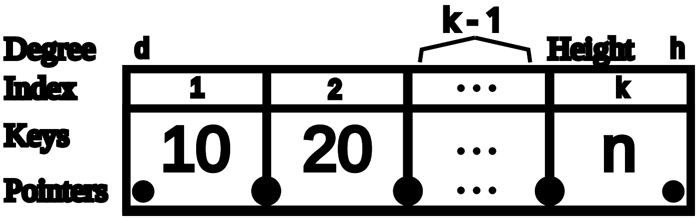


Figure: Node of a B-Tree

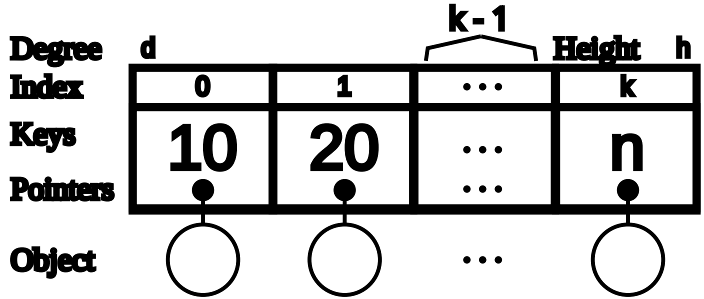


Figure: Leaf of a B-Tree

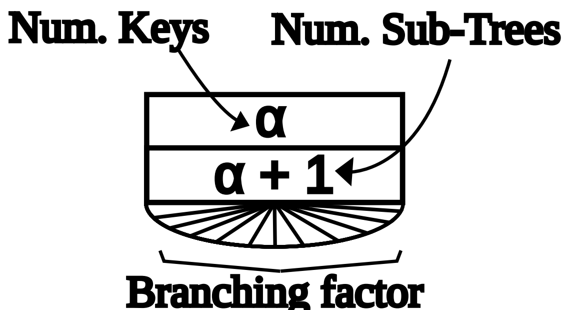


Figure: Generic Node of a B-Tree

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 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.

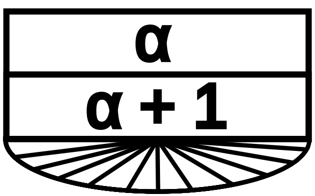


Figure: Miminum Keys and Sub-Trees on a Node

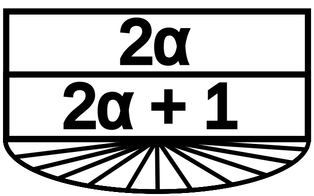


Figure: Maximun Keys and Sub-Trees on a Node

B-Tree Properties - The α constant II

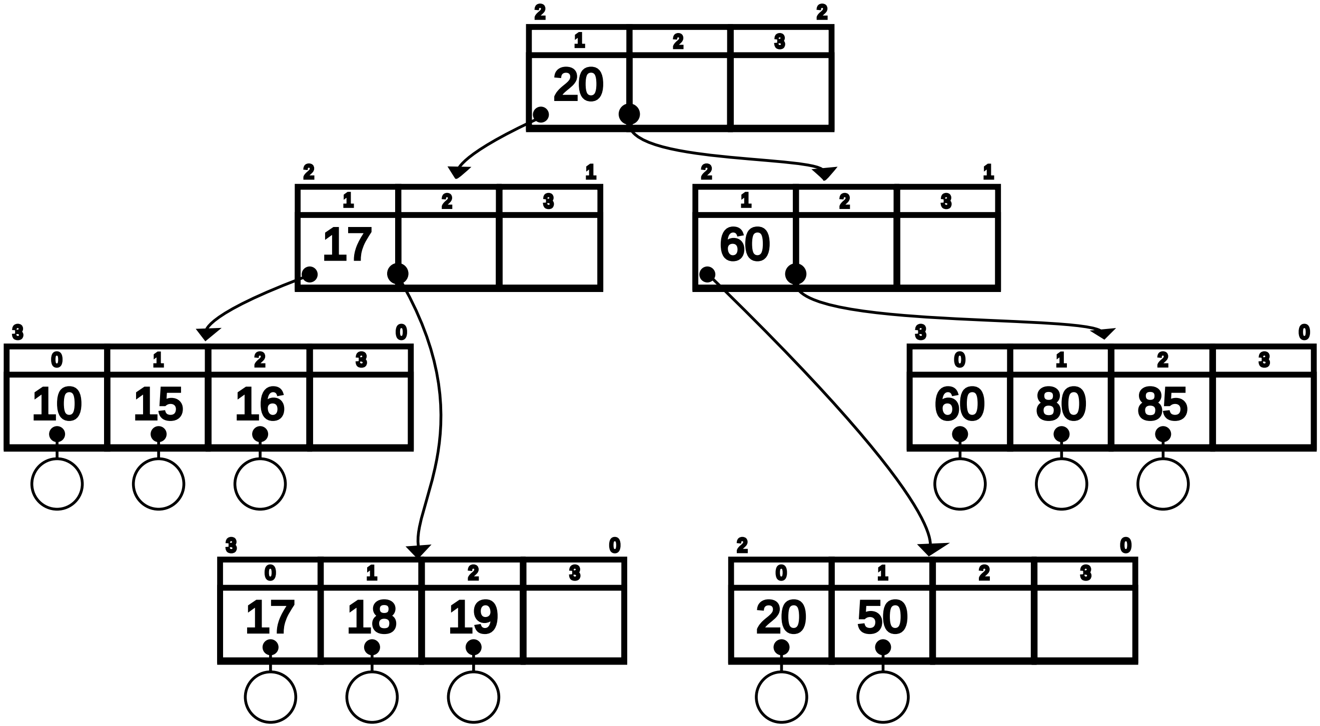


Figure: B-Tree, t(2, 2)

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

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

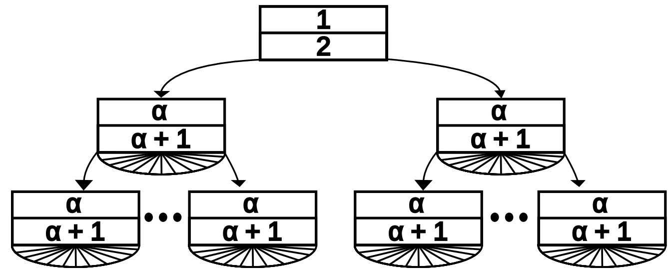


Figure: B-Tree w/ the least number of nodes

B-Tree Properties - The α constant IV

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

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

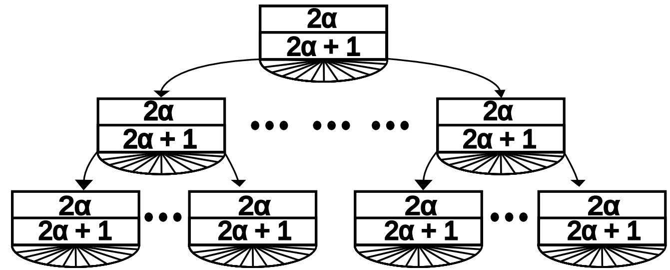


Figure: B-Tree w/ the most number of nodes

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 - 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 $\alpha \leq l \leq 2\alpha$.

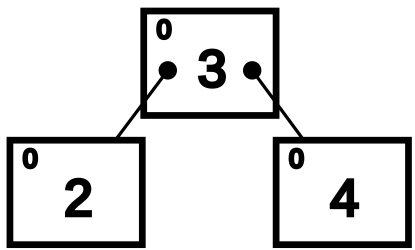


Figure: Simple node of a Normal Binary Tree

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 II

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

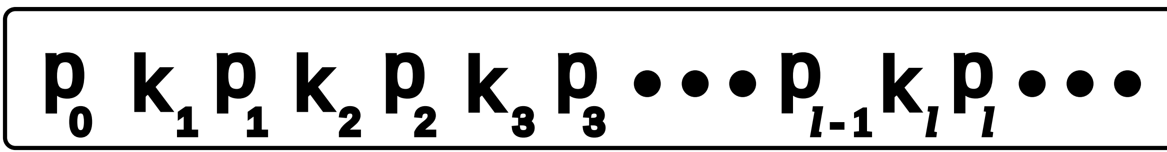


Figure: Order of the Subtree Pointers and Keys.

B-Tree Properties - Keys and Sub-trees 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.

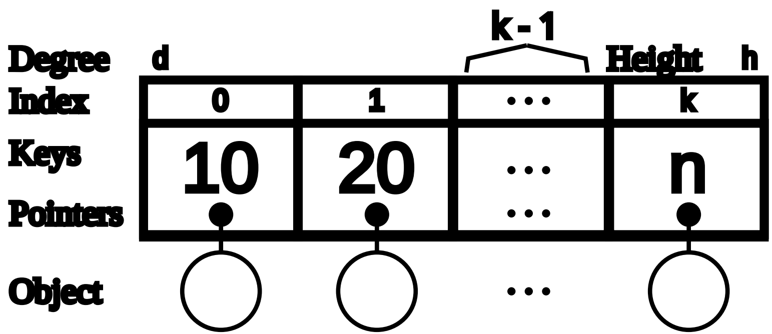


Figure: Leaf of a B-Tree

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 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.

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

B-Tree Properties - Keys and Sub-trees V

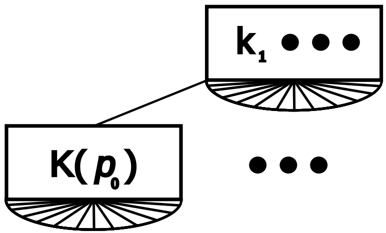


Figure: Sub-tree Keys (Case 1)

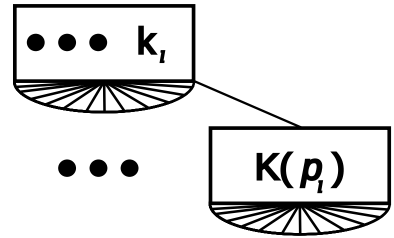


Figure: Sub-tree Keys (Case 3)

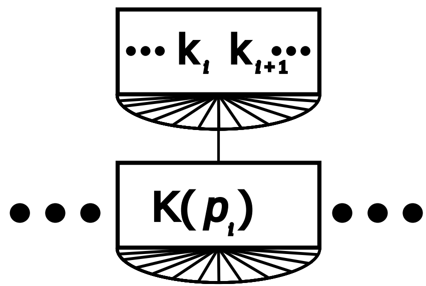


Figure: Sub-tree Keys (Case 2)

B-Tree Properties - Height I

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

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

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

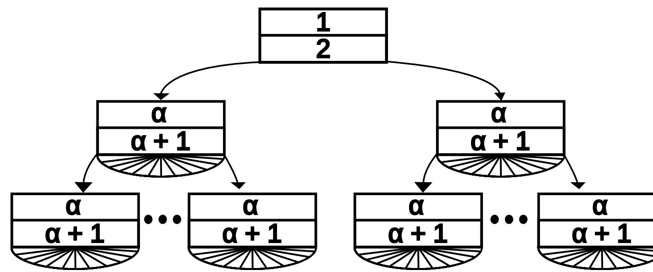


Figure: B-Tree w/ the least number of nodes

B-Tree Properties - Height II

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

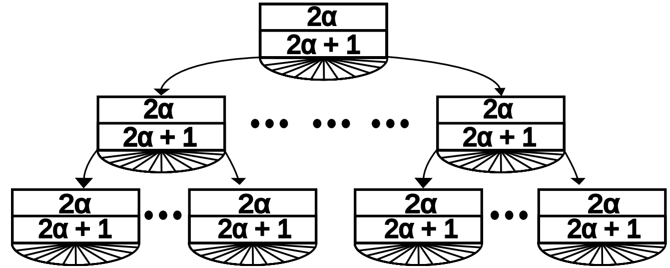


Figure: B-Tree w/ the most number of nodes

- 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:

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

- 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:

```
int alpha = 2; /* any int >= 2 */
typedef struct tr_n_t {
  int degree;
  int height;
  key_t key[2 * alpha];
  struct tr_n_t *next[(2 * alpha) + 1];
  /* ... */
} tree_node_t;
```

B-Tree Operations

- For this 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.

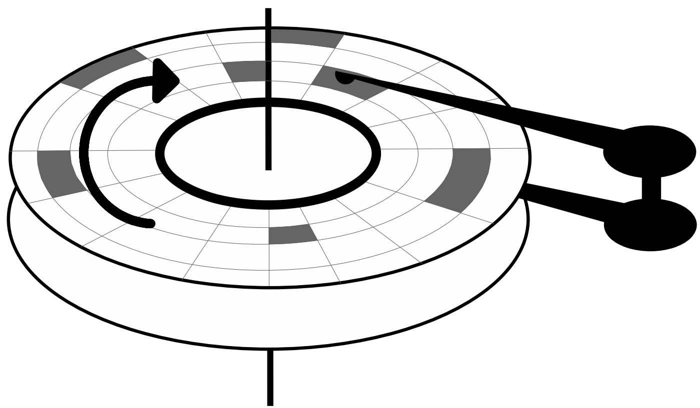


Figure: External storage with the sectors to access highlighted

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

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 );
}
```

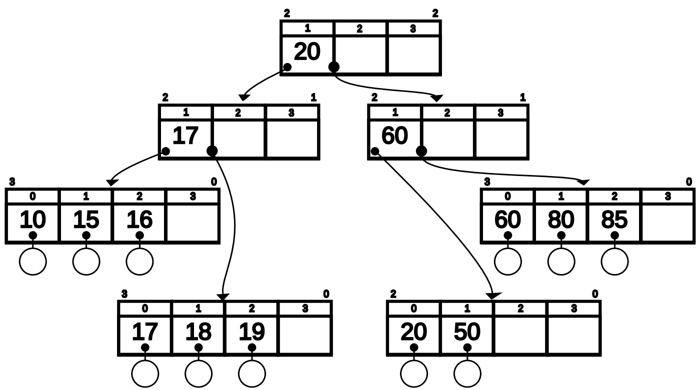
B-Tree Operations - Search I

- 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];

```

B-Tree Operations - Search II

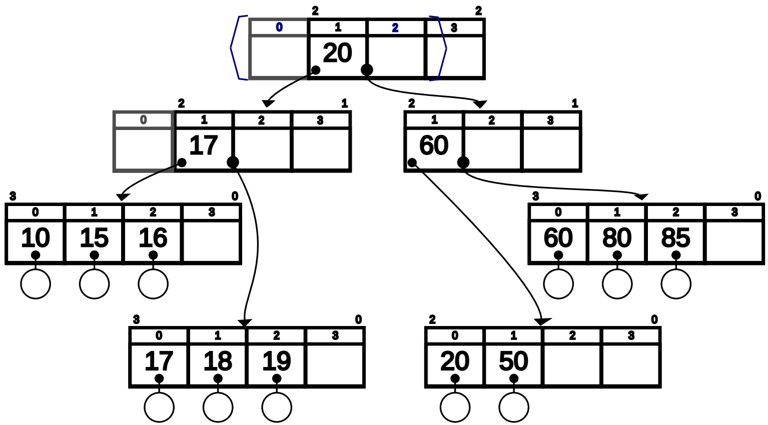
```

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



B-Tree Operations - Search (Example) I

```
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;
```

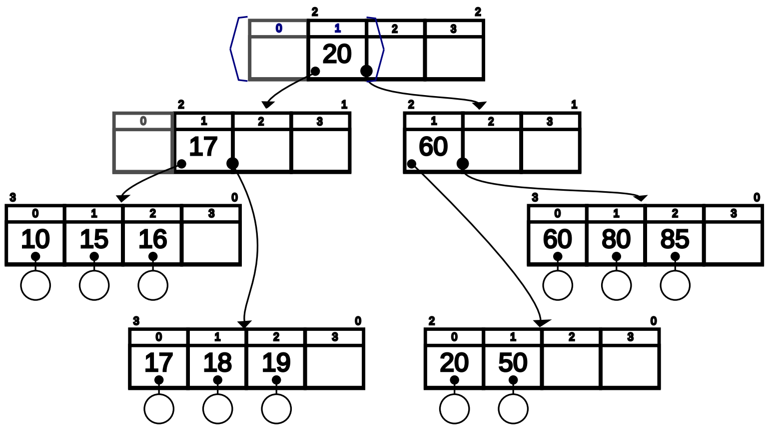
```
// Step 0
query_key = 19;
tree = *(node 1);

current_node = *(node 1);
current_node->height = 2;
current_node->degree = 2;

lower = 0;
upper = 2;
```



B-Tree Operations - Search (Example) II

```
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];
```

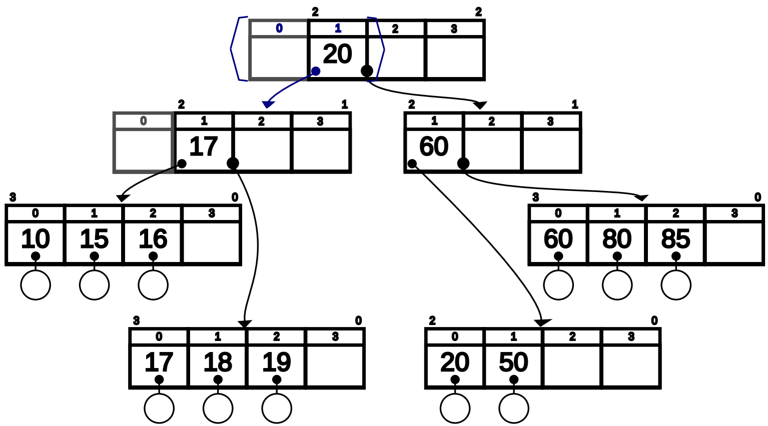
```
// Step 1
query_key = 19;
tree = *(node 1);

current_node = *(node 1);
current_node->height = 2;
current_node->degree = 2;

lower = 0;
upper = 1;
med = 1;
```



B-Tree Operations - Search (Example) III

```
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];

```

```
// Step 2
query_key = 19;
tree = *(node 1);

current_node = *(node 1);
current_node->height = 2;
current_node->degree = 2;

lower = 0;
upper = 1;
med = 1;
```

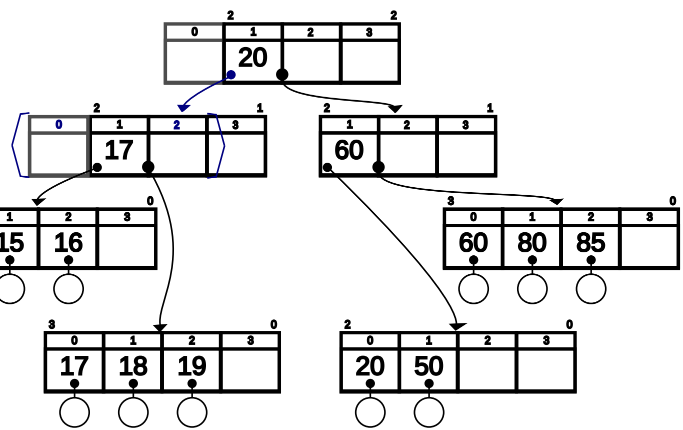


Presentation made by Hernández, Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

B-Tree Operations - Search (Example) IV

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

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

```
// Step 3
query_key = 19;
tree = *(node 1);

current_node = *(node 2);
current_node->height = 1;
current_node->degree = 2;

lower = 0;
upper = 2;
med = 1; // Not changed yet
```

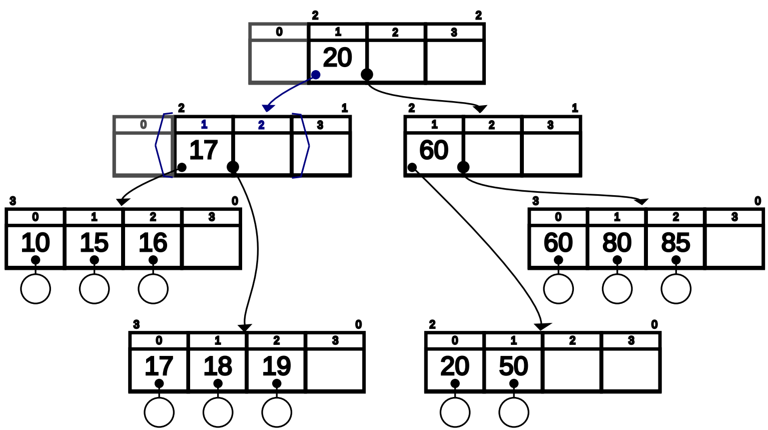


Presentation made by Hernández, Mendivelso. Contents and figures extracted from the book: Advanced Data Structures. Peter Brass. Cambridge University Press. 2008.

B-Tree Operations - Search (Example) V

```
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];
```

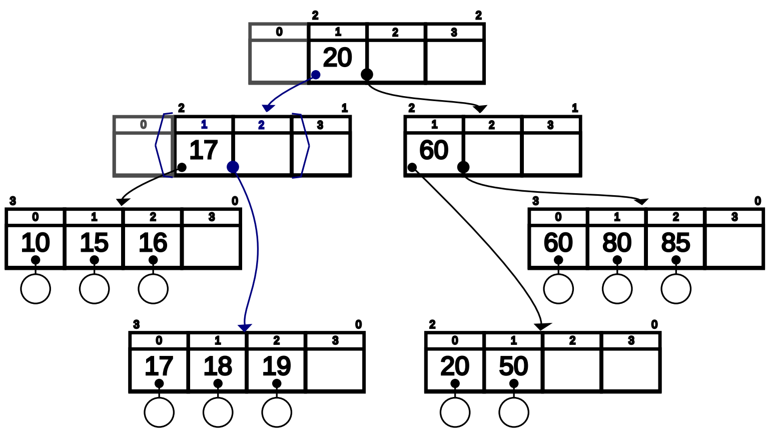
```
// Step 4
query_key = 19;
tree = *(node 1);

current_node = *(node 2);
current_node->height = 1;
current_node->degree = 2;

lower = 1;
upper = 2;
med = 1;
```



B-Tree Operations - Search (Example) VI

```
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];
```

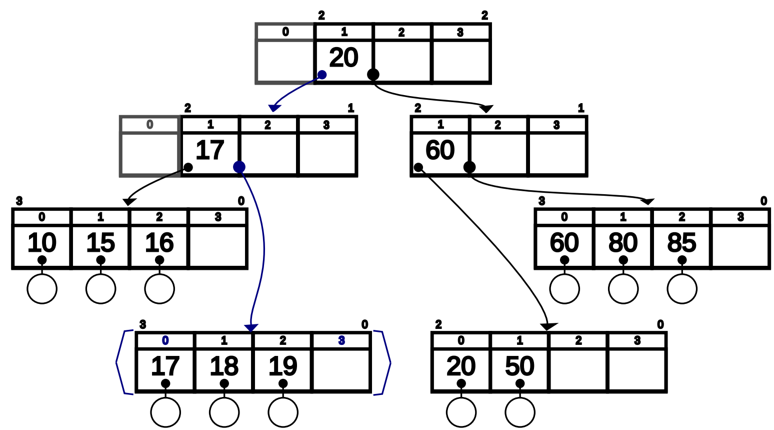
```
// Step 5
query_key = 19;
tree = *(node 1);

current_node = *(node 2);
current_node->height = 1;
current_node->degree = 2;

lower = 1;
upper = 2;
med = 1;
```



B-Tree Operations - Search (Example) VII

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

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

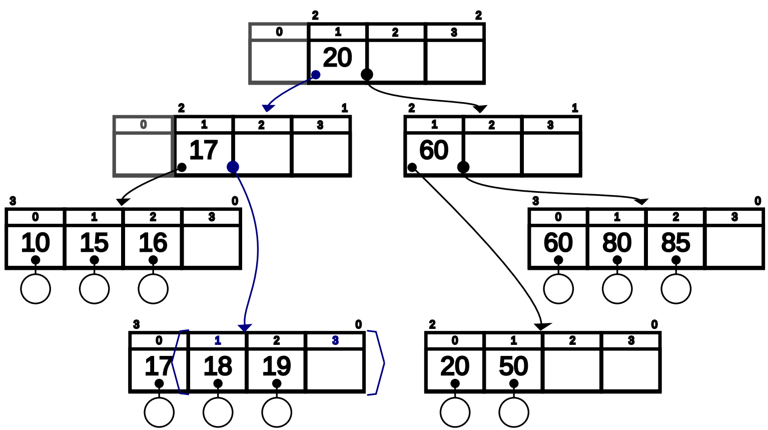
```
// Step 6
query_key = 19;
tree = *(node 1);

current_node = *(node 6);
current_node->height = 0;
current_node->degree = 3;

lower = 0;
upper = 3;
med = 1; // Not changed yet
```



B-Tree Operations - Search (Example) VIII

```
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];

```

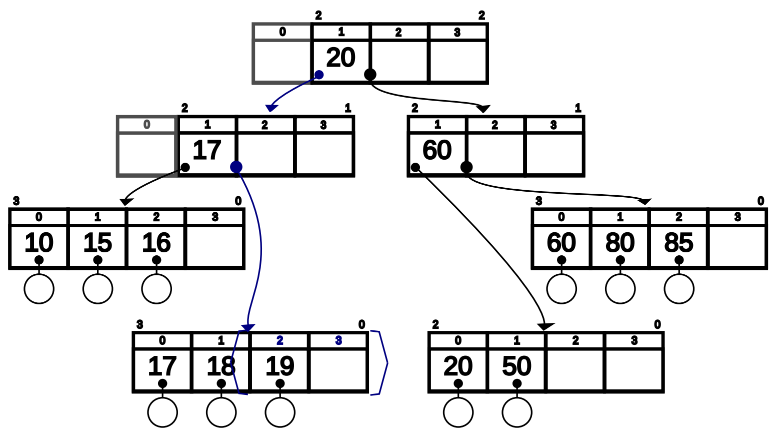
```
// Step 7
query_key = 19;
tree = *(node 1);

current_node = *(node 6);
current_node->height = 0;
current_node->degree = 3;

lower = 1;
upper = 3;
med = 1;
```



B-Tree Operations - Search (Example) IX

```
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];
```

```
// Step 8
query_key = 19;
tree = *(node 1);

current_node = *(node 6);
current_node->height = 0;
current_node->degree = 3;

lower = 2;
upper = 3;
med = 2;
```

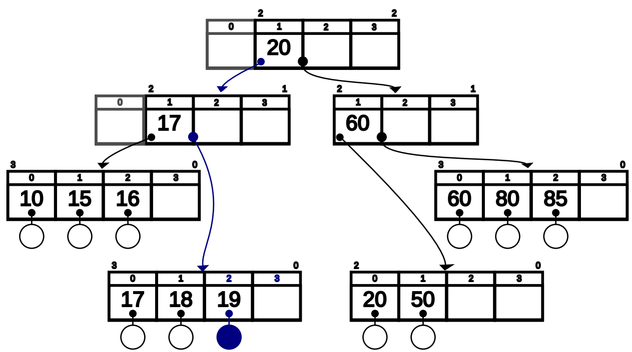


B-Tree Operations - Search (Example) X

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

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

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

```
// Step 9
query_key = 19;
tree = *(node 1);

object = *(19)

lower = 2;
upper = 3;
med = 2;
```



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

B-Tree Operation - Insert Node I

- Unlike all the types of trees that we've seen, in order to insert an element with it's key and value, we can't create a leaf node and insert it besides the other leaves of the tree, and if needed do some rotations in order to keep balance.
- We can only insert the key into a existing Node, and keeping in mind the *Branching factor* of the tree at all times by avoiding filling up the node elements to the upper bound.
- Keeping the *Branching factor* right is made by splitting of the nodes and then rotating around their **median key**.

- The `insert` operation mainly depends on the function `insertInternal` which takes handles almost all of the important logic when we are inserting a new key in the tree.
- The only case handled in the `insert` operation if when we have split a node and need to create a new root to it points to the old and new nodes.

B-Tree Operation - Insert Node II

```
void btInsert(bTree b, int key) {
  bTree b1, b2;
  int median;

  b2 = btInsertInternal(b, key, &median);
  if(!b2) {
    return;
  }

  b1 = malloc(sizeof(*b1));
  memmove(b1, b, sizeof(*b));
  b->numKeys = 1;
  b->isLeaf = 0;
  b->keys[0] = median;
  b->kids[0] = b1;
  b->kids[1] = b2;
}
```

B-Tree Operation - Insert Node III

- The `insertInternal` function starts by getting the position of the key in the node by using `searchKey`, the same function that in the `search` operation.
- This to first check if the key is already in the node.
- And since the `searchKey` function gives us the smaller index i such that for a node n and key k to insert: $k \leq n.keys\,[i]$.
- If we are in a leaf we can insert the key directly by moving the memory of the keys in the array of the node by 1 position:

```
bTree btInsertInternal(bTree b, int key, int *median) {
  int pos = searchKey(b->numKeys, b->keys, key);
  int mid;
  bTree b2;

  if(pos < b->numKeys && b->keys[pos] == key)
    return 0; /* nothing to do */

  if(b->isLeaf) {
      memmove(&b->keys[pos+1], &b->keys[pos], sizeof(*(b->keys)) * (b->numKeys - pos));
      b->keys[pos] = key;
      b->numKeys++;
  } else {
```

B-Tree Operation - Insert Node IV

```
    ...
```

- Otherwise we will call recursively `insertInternal` until we reach a leaf that we can insert the key.

```
    ...
  } else {
    b2 = btInsertInternal(b->kids[pos], key, &mid);
    if(b2) {
      memmove(&b->keys[pos+1], &b->keys[pos], sizeof(*(b->keys)) * (b->numKeys - pos));
      memmove(&b->kids[pos+2], &b->kids[pos+1], sizeof(*(b->keys)) * (b->numKeys - pos));

      b->keys[pos] = mid;
      b->kids[pos+1] = b2;
      b->numKeys++;
    }
  }
```

- Then, we will check if the number of keys in the node doesn't overflow the $2\alpha - 1$ upper limit.
- In case of overflow, we will calculate the median key of the node, and pass it to `insert` via an argument by reference.
- Then, it'll create a new node with the elements, keys and sub-trees to the right of the median key.
- Also setting node information like the number of keys, and is it's a leaf.
- Otherwise, if there's no overflow, just return 0.

B-Tree Operation - Insert Node V

```
    ...
    if(b->numKeys >= (2*alpha - 1)) {
      mid = b->numKeys/2;

      *median = b->keys[mid];

      /* make a new node for keys > median */
      b2 = malloc(sizeof(*b2));

      b2->numKeys = b->numKeys - mid - 1;
      b2->isLeaf = b->isLeaf;

      memmove(b2->keys, &b->keys[mid+1], sizeof(*(b->keys)) * b2->numKeys);
      if(!b->isLeaf) {
          memmove(b2->kids, &b->kids[mid+1], sizeof(*(b->kids)) * (b2->numKeys + 1));
      }

      b->numKeys = mid;
      return b2;
    } else {
      return 0;
    }
  }
```

- And thus, completing the `insert` operation.
- Since we had to access nodes all the way until a leaf, and re-balance a bunch of the nodes in the worst scenario.
- The operation will take only $O\left(log_{\alpha} \frac{n+1}{2}\right)$ of CPU processing and $O\left(h\right)$ of disk accesses.
- Which is fast, but there's tree that are faster in this process, mainly in the disk access.

B-Tree Operation - Destroying a B-Tree

- We just iter through each node recursively, freeing the leaves first and then the inner nodes all the way to the *Root* of the tree.

```
void btDestroy(bTree b) {
  if(!b->isLeaf) {
    for (int i = 0; i < b->numKeys + 1; i++) {
      btDestroy(b->kids[i]);
    }
  }
  free(b);
}
```

B-Tree Secondary Memory Access

- Fairly good for storing data in external memory in comparison to height, weight or search trees.
- The limit of $2\alpha - 1$ help us by forcing that each size node will be optimized.
- But, this limit also make that if we need to re-balance the tree the operation will take $\Theta(\alpha logn)$, updating all the split nodes.
- This operation doesn't affect much in main memory, but in secondary memory where each reading can take longer time due to the technology available we might run in multiple problems of efficiency.

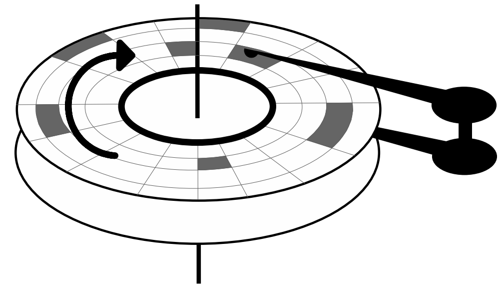


Figure: External storage with the sectors to access highlighted

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

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).