(a, b) and 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

Similarities in (a,b)-Trees and B-Trees

- Both types of trees have higher degree than the previous trees.
- Meaning that, both have more than 1 key and 2 sub-trees in each node.
- Each type has a lower and upper limit of keys and sub-trees, which are defined by constants.
- Due to the higher degree, there's changes in the code of the `find`, `insert` and `delete` operations.

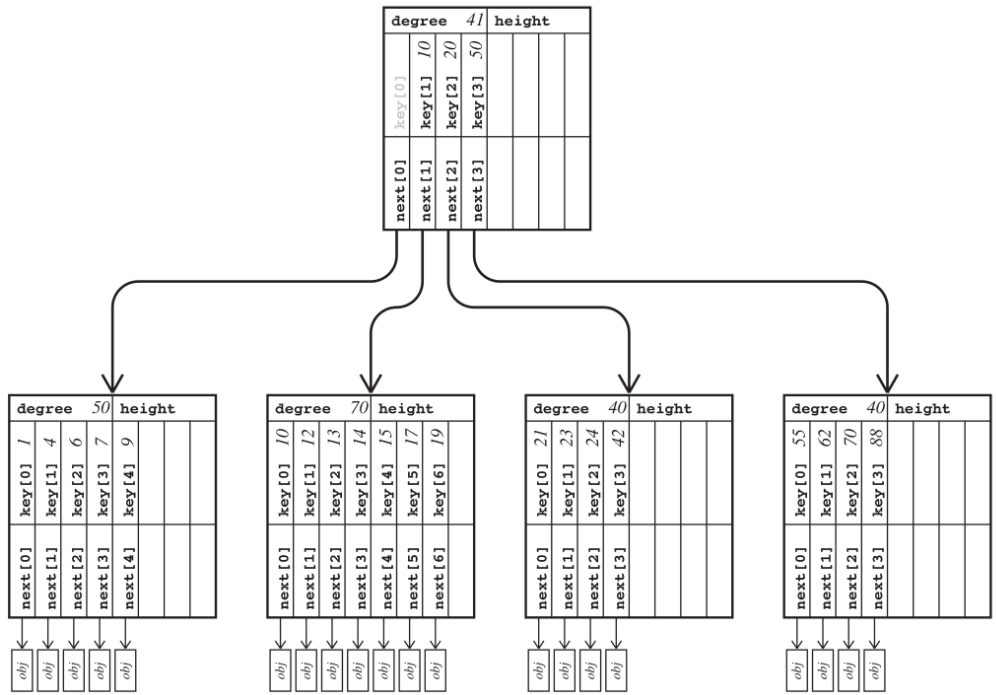


Figure: (a,b)-Tree

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.

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 360 / 44

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: IBM 360 / 44

B-Tree Definition

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

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

- Where h is the height of the B-Tree.
- And, α is a predefined constant.

B-Tree Properties - The α constant I

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

B-Tree Properties - The α constant II

- The α must be a Natural number, $\alpha \in \mathbb{N}$.
- Since, there's other definitions of B-Trees that define the bounds for the keys and sub-trees in a node differently, generally we choose an $\alpha \geq 2$.
- Also, the α is often the greatest number possible that the primary memory can handle the mentioned intervals.

B-Tree Properties - The α constant III

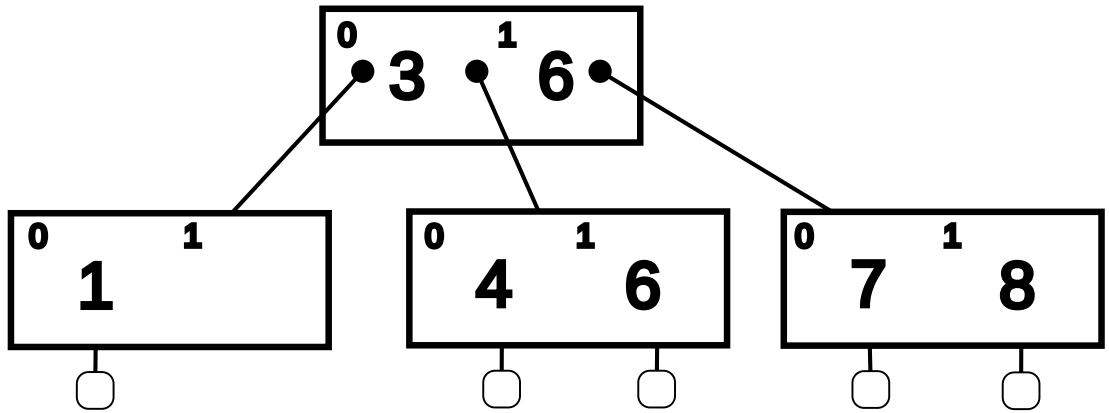


Figure: B-Tree, t(1, 2)

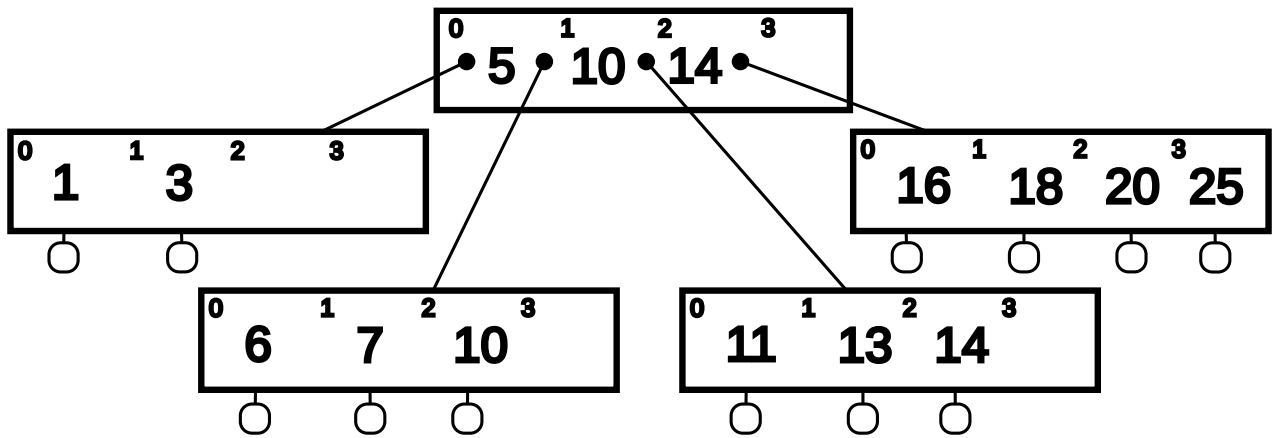


Figure: B-Tree, t(2, 2)

B-Tree Properties - The α constant IV

- 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

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

B-Tree Properties - The α constant V

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

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

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

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

(Nodes Bounds)

□

B-Tree Properties - Keys and Sub-trees I

- The keys and sub-trees are stored in a sequential increasing order.
- Each key has two sub-trees, one before and one after it. Like a mini-tree.
- We can define l as the number of keys in a node N, which isn't a leaf or *Root*.
- Such that for $t\,(\alpha, h)$, we have $\alpha \leq l \leq 2\alpha$.
- We can consider the sub-trees 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 $l + 1$.

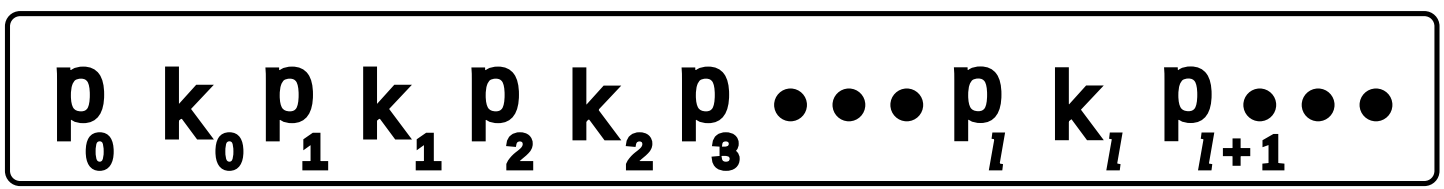


Figure: Order of Keys and Sub-trees in a B-Tree Node

B-Tree Properties - Keys and Sub-trees II

- 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.

$$\forall y \in K(p_0); \quad y < k_0 \qquad \text{(Case 1)}$$

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

$$\forall y \in K(p_{l+1}); \quad k_l < y \qquad \text{(Case 3)}$$

B-Tree Properties - Keys and Sub-trees III

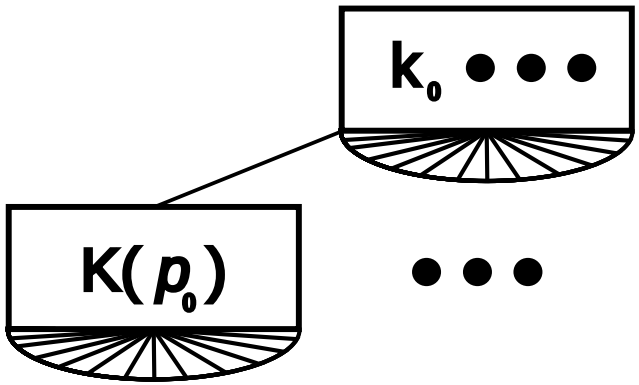


Figure: Sub-tree Keys (**??**)

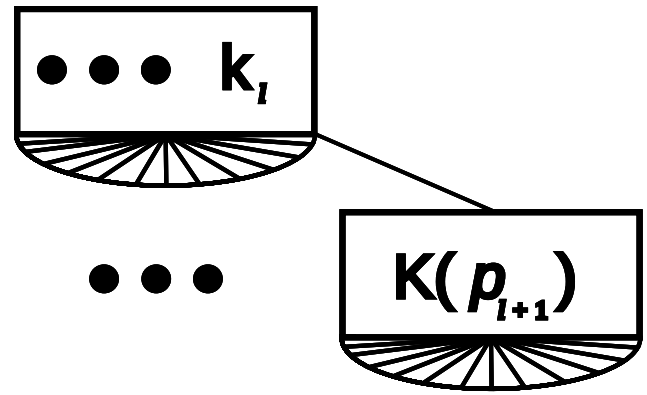


Figure: Sub-tree Keys (**??**)

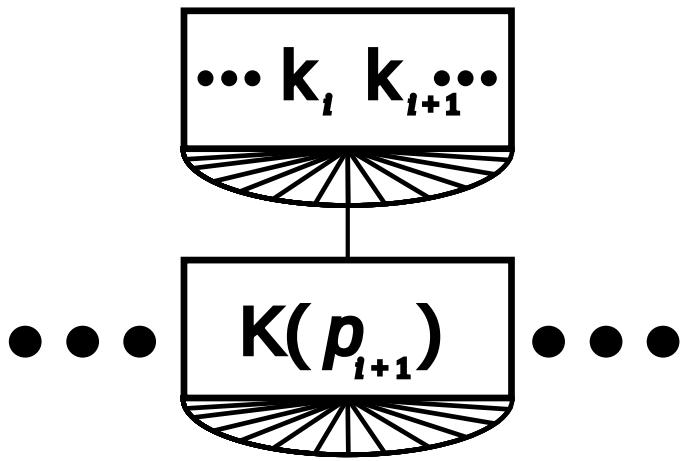


Figure: Sub-tree Keys (**??**)

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 (**??**):

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

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

B-Tree Properties - Height II

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

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

$$I_{\text{min}} = 2\,(\alpha+1)^{h-1} - 1$$
$$\frac{I_{\text{min}}}{2} + 1 = (\alpha+1)^{h-1}$$
$$\log_{\alpha+1} \frac{I_{\text{min}}+1}{2} = h$$

$$I_{\text{max}} = (2\alpha+1)^h - 1$$
$$I_{\text{max}} + 1 = (2\alpha+1)^h$$
$$\log_{2\alpha+1} I_{\text{max}} + 1 = h$$

B-Tree Properties - Height III

- Since, $2\alpha + 1 > \alpha + 1$, then $log_{2\alpha+1}x \leq log_{\alpha+1}x$, both in $[1, \infty)$.
- Hence, for $I \geq 1$, we will have the bounds for h:

$$\log_{2\alpha+1} I + 1 \leq h \leq \log_{\alpha+1} \frac{I+1}{2}$$

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

B Tree - Generic Example

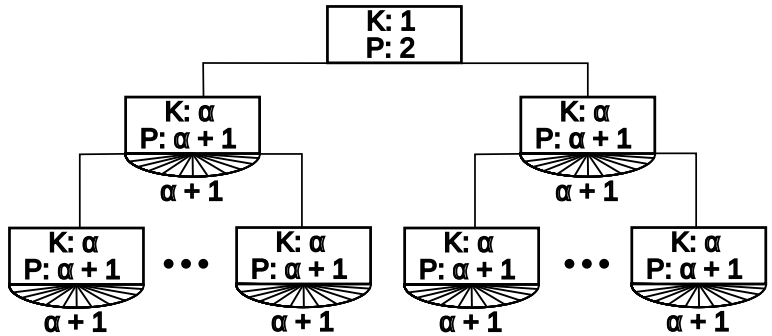


Figure: B-Tree w/ the least number of elements

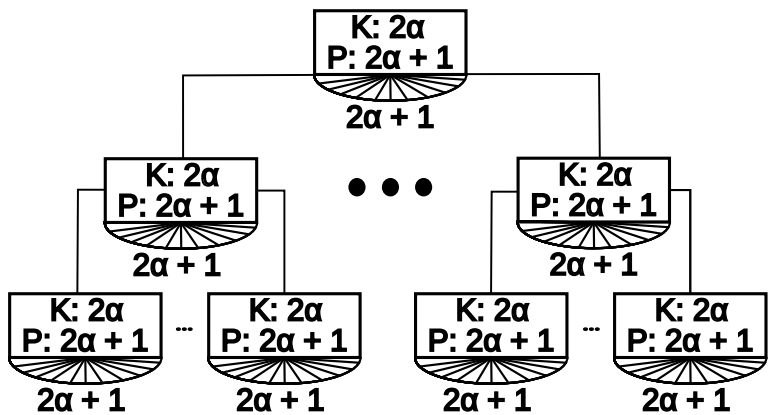


Figure: B-Tree w/ the most number of elements

B-Tree Structure

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

```
1  int alpha = 2 /* any int >= 2 */
2  typedef struct tr_n_t {
3    int degree;
4    int height;
5    key_t key[2 * alpha];
6    struct tr_n_t * next[(2 * alpha) + 1];
7    /* possibly other information */
8  } 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.

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 $O(1)$.

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

B-Tree Operations - Search I

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

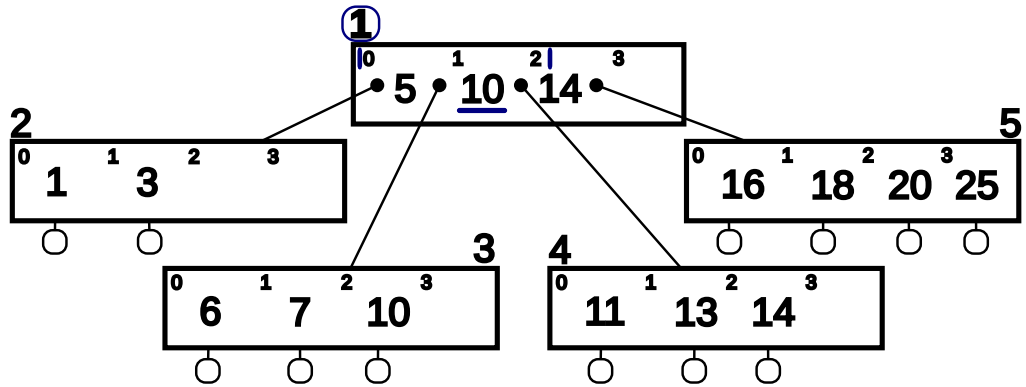
```
 1  object_t *find(tree_node_t *tree, key_t query_key) {
 2    tree_node_t *current_node;
 3    object_t *object;
 4    current_node = tree;
 5    while( current_node->height >= 0 ) { /* binary search
          among keys */
 6      int lower, upper;
 7      lower = 0;
 8      upper = current_node->degree;
 9      while( upper > lower +1 ) {
10        if( query_key < current_node->key[(upper+lower)/2
              ] )
```

B-Tree Operations - Search II

```
11          upper = (upper+lower)/2;
12        else
13          lower = (upper+lower)/2;
14      }
15      if( current_node->height > 0)
16        current_node = current_node->next[lower + 1]; /*
              Offset sub-tree ptrs def */
17      else { /* block of height 0, contains the object
            pointers */
18        if( current_node->key[lower] == query_key )
19          object = (object_t *)
20          current_node->next[lower];
21        else
22          object = NULL;
23        return( object );
24      }
25    }
26  }
```

B-Tree Operations - Search (Example) I

- We are going to search the key: 16.
- First we will do a *Binary Search* in the Keys of the node.
- We set the lower and upper as 0, 2.
- The median is 1, and the key[1] is lesser than the search key.
- Then, we update lower to 1.



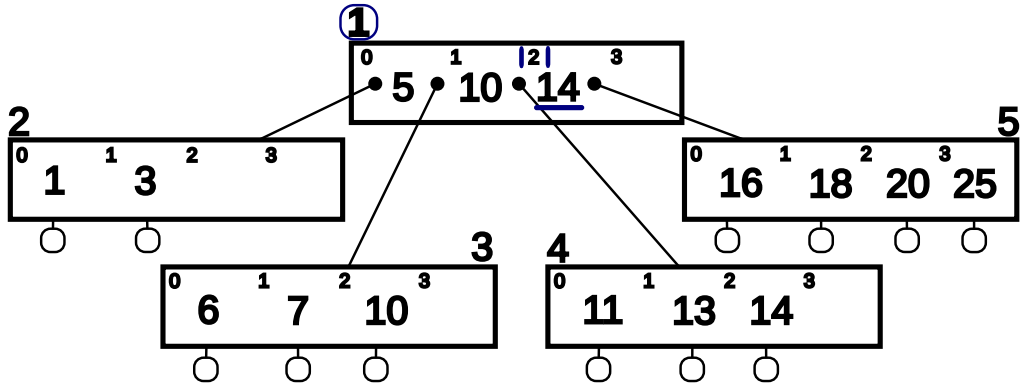

```
query_key = 16;
tree = *(node 1);

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

lower = 0;
upper = 2;
```

B-Tree Operations - Search (Example) II

- We repeat the median process and get 2.
- The key[2] is greater than the search key so we update the upper.
- Now that lower is 2 and upper is 2, we had reached the key and sub-tree to keep searching.
- We move `current_node` to `*(node 5)`.



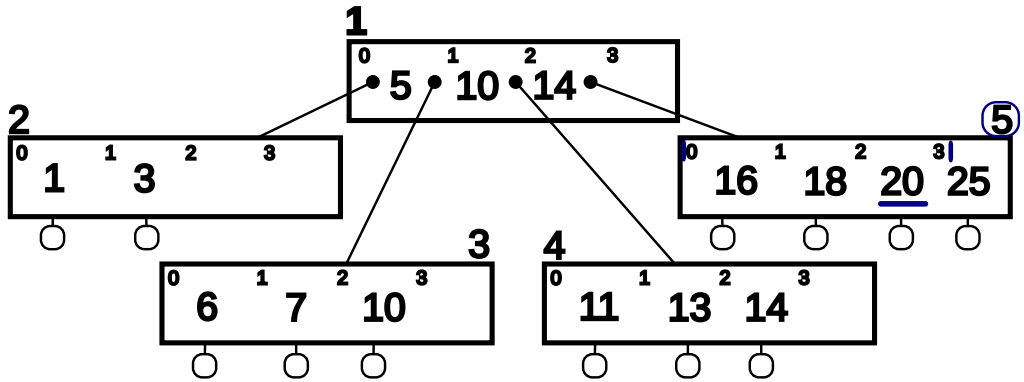

```
query_key = 16;
tree = *(node 1);

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

lower = 2;
upper = 2;
```

B-Tree Operations - Search (Example) III

- Now we repeat the *Binary search* process in the new node.
- Lower is 0, Upper is 3.
- Median is 2.
- Key[1] is greater than the search key.
- Update Upper.

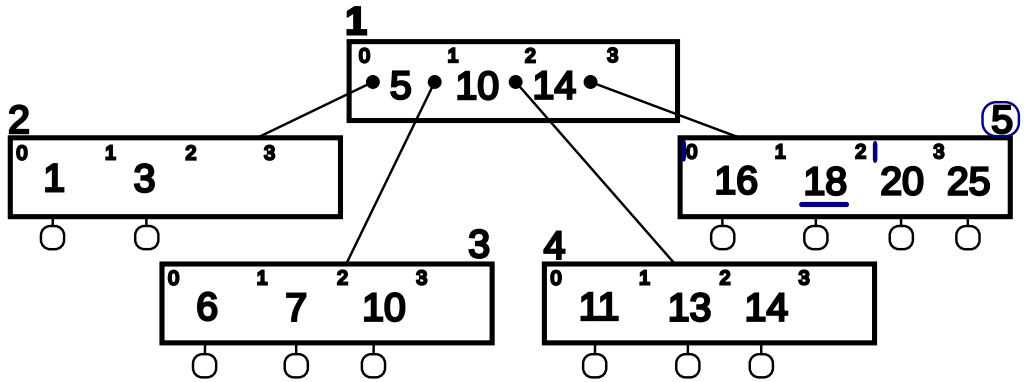



```
query_key = 16;
tree = *(node 1);

object = NULL;
current_node = *(node 5);
current_node->height = 0;
current_node->degree = 3;

lower = 0;
upper = 3;
```

B-Tree Operations - Search (Example) IV



- Lower is 0 and Upper is 2.
- Median is 1.
- Key[1] is still greater than the search key.
- Update Upper.

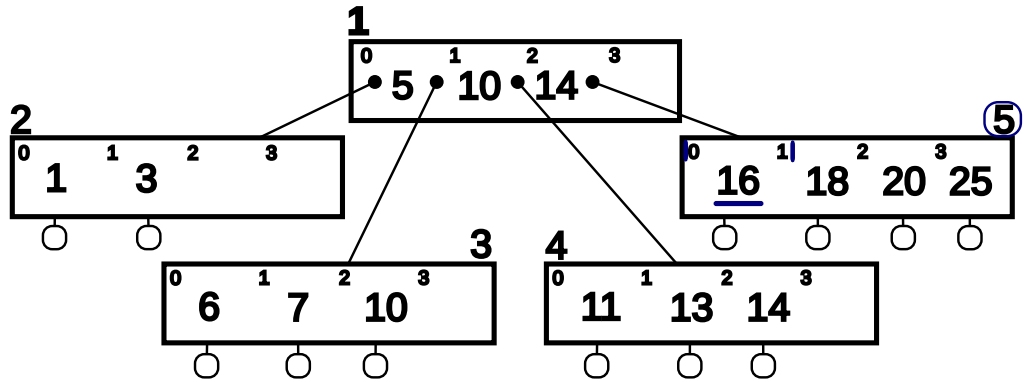
```
query_key = 16;
tree = *(node 1);

object = NULL;
current_node = *(node 5);
current_node->height = 0;
current_node->degree = 3;

lower = 0;
upper = 2;
```

B-Tree Operations - Search (Example) V

- Lower is 0 and Upper is 1.
- Median is 0.
- Key[0] is the key that we are searching, but we still need another step.
- Update Upper.



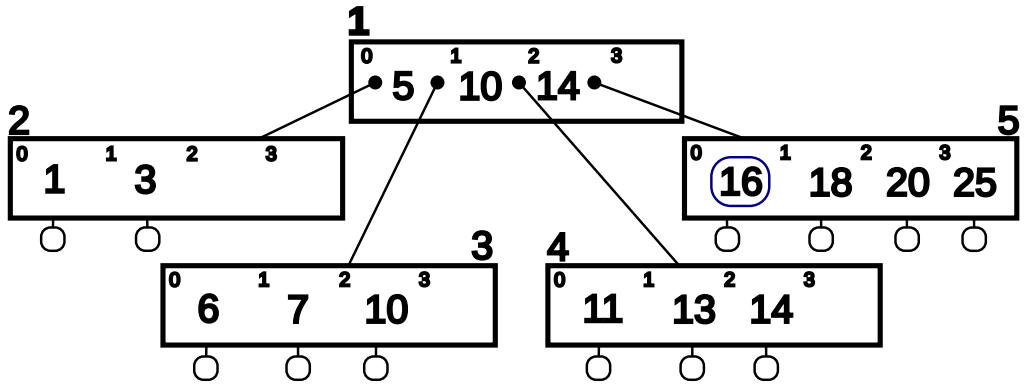

```
query_key = 16;
tree = *(node 1);

object = NULL;
current_node = *(node 5);
current_node->height = 0;
current_node->degree = 3;

lower = 0;
upper = 1;
```

B-Tree Operations - Search (Example) VI

- Lower is 0 and Upper is 0.
- We break from the *Binary Search* loop.
- And since we are on a leaf, we can just check if key[lower] is the search key.
- In this case, it is, so we return *(16).




```
query_key = 16;
tree = *(node 1);

object = *(16);
current_node = *(node 5);
current_node->height = 0;
current_node->degree = 3;
```

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**.

B-Tree Operation - Insert Node II

- 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 III

```
 1  void btInsert(bTree b, int key) {
 2    bTree b1, b2;
 3    int median;
 4
 5    b2 = btInsertInternal(b, key, &median);
 6    if(!b2) {
 7      return;
 8    }
 9
10    b1 = malloc(sizeof(*b1));
11    memmove(b1, b, sizeof(*b));
12    b->numKeys = 1;
13    b->isLeaf = 0;
14    b->keys[0] = median;
15    b->kids[0] = b1;
16    b->kids[1] = b2;
17  }
```

B-Tree Operation - Insert Node IV

- 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:

B-Tree Operation - Insert Node V

```
 1  bTree btInsertInternal(bTree b, int key, int *median) {
 2    int pos = searchKey(b->numKeys, b->keys, key);
 3    int mid;
 4    bTree b2;
 5
 6    if(pos < b->numKeys && b->keys[pos] == key)
 7      return 0; /* nothing to do */
 8
 9    if(b->isLeaf) {
10        memmove(&b->keys[pos+1], &b->keys[pos], sizeof(*(b
              ->keys)) * (b->numKeys - pos));
11        b->keys[pos] = key;
12        b->numKeys++;
13    } else {
14      ...
```

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

B-Tree Operation - Insert Node VI

```
12        ...
13      } else {
14        b2 = btInsertInternal(b->kids[pos], key, &mid);
15        if(b2) {
16          memmove(&b->keys[pos+1], &b->keys[pos], sizeof(*(b
                ->keys)) * (b->numKeys - pos));
17          memmove(&b->kids[pos+2], &b->kids[pos+1], sizeof
                (*(b->keys)) * (b->numKeys - pos));
18
19          b->keys[pos] = mid;
20          b->kids[pos+1] = b2;
21          b->numKeys++;
22        }
23      }
```

B-Tree Operation - Insert Node VII

- 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 VIII

```
24      ...
25      if(b->numKeys >= (2*alpha - 1)) {
26        mid = b->numKeys/2;
27
28        *median = b->keys[mid];
29
30        /* make a new node for keys > median */
31        b2 = malloc(sizeof(*b2));
32
33        b2->numKeys = b->numKeys - mid - 1;
34        b2->isLeaf = b->isLeaf;
35
36        memmove(b2->keys, &b->keys[mid+1], sizeof(*(b->keys)
              ) * b2->numKeys);
37        if(!b->isLeaf) {
38            memmove(b2->kids, &b->kids[mid+1], sizeof(*(b->
                  kids)) * (b2->numKeys + 1));
39        }
40
```

B-Tree Operation - Insert Node IX

```
41        b->numKeys = mid;
42        return b2;
43      } else {
44        return 0;
45      }
46    }
```

- 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.

```
1  void btDestroy(bTree b) {
2    if(!b->isLeaf) {
3      for (int i = 0; i < b->numKeys + 1; i++) {
4        btDestroy(b->kids[i]);
5      }
6    }
7    free(b);
8  }
```

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 $\Theta(\alpha log n)$, 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.

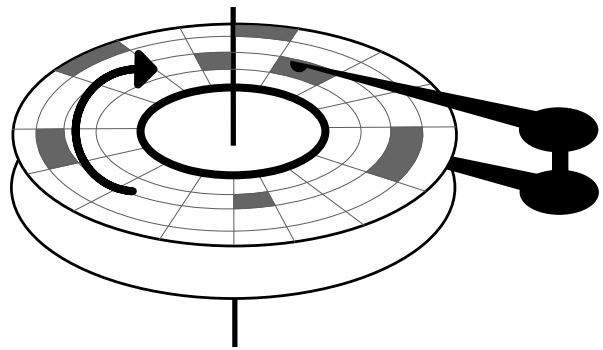


Figure: External storage with the sectors to access highlighted

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.

Bibliography II

[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).