(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.

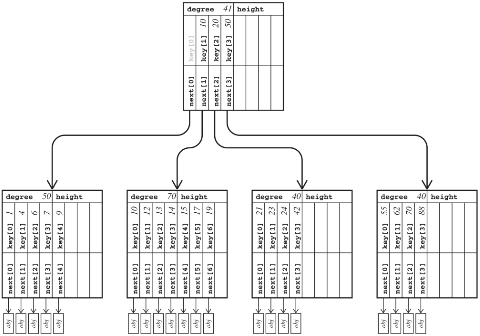


Figure: (a,b)-Tree

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.

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: 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

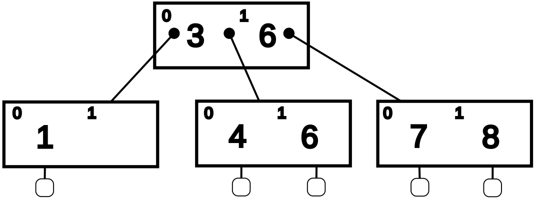


Figure: B-Tree, t(1, 2)

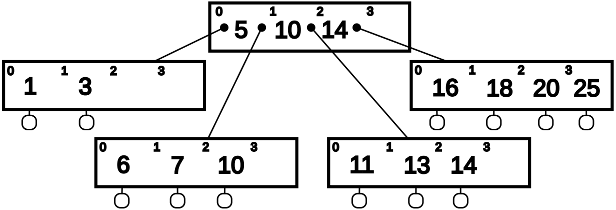


Figure: B-Tree, t(2, 2)

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 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

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

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 $l + 1$.

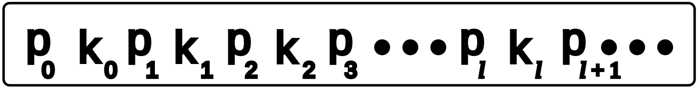


Figure: Order of Keys and Sub-trees in a B-Tree Node

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.

$$
\begin{aligned}
\forall y \in K(p_0); &\quad y < k_0 && \text{(Case 1)} \\
\forall y \in K(p_{i+1}); &\quad k_i < y < k_{i+1}; \quad 0 < i < l && \text{(Case 2)} \\
\forall y \in K(p_{l+1}); &\quad k_l < y && \text{(Case 3)}
\end{aligned}
$$

B-Tree Properties - Keys and Sub-trees III

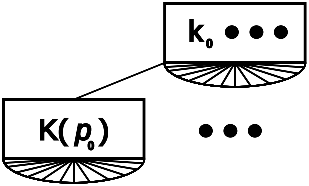


Figure: Sub-tree Keys (Case 1)

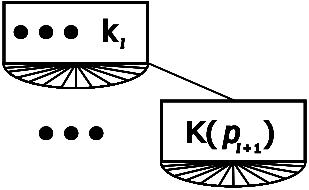


Figure: Sub-tree Keys (Case 3)

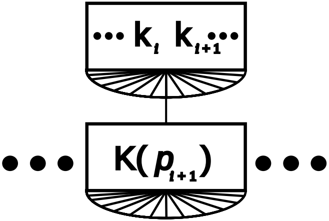


Figure: Sub-tree Keys (Case 2)

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):

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

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

B-Tree Properties - Height II

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

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

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

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

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 like:

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

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

B-Tree Properties - Height IV

Proof.

Let $T \in t(\alpha, h)$ where $h \geq 1$ and T is the *Root* of the B-Tree. By the Keys and Sub-Trees properties we will have that the *Root* node, T must have at least 1 key and 2 sub-trees. Also, the sub-trees have at least α keys and $\alpha + 1$ sub-sub-trees.

We can get the least number of nodes of this sub-sub-tree by a simple multiplication:

$$(\alpha + 1) \cdot (\alpha + 1) = (\alpha + 1)^2$$

Now, we can calculate the least number of nodes for any sub-tree using their depth,
being d the depth of a Node N:

$$(\alpha + 1)^d$$

B-Tree Properties - Height V

Then, we sum the least number of nodes of the *Root*, and the sub-trees with depth 1 to the leaves, or depth of h:

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

$\square$

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 btr_n_t {
    int isLeaf;
    int numKeys;
    int keys[2*alpha - 1];
    struct btr_n_t *kids[2*alpha - 1];
} tree_node_t;
```

B-Tree Operations

- For this operations, we will assume that the whole B-Tree isn't loaded into main memory, since the main usage of the B-Tree is oriented to secondary storage.
- But the *Root* and node to operate, if available, will be always available in memory.
- In order to read the nodes that aren't loaded into main memory we will define the functions:
- `disk_read(n *tree_node_t)`: Reads a node `n` from the secondary memory, and returns a pointer to it.
- `disk_write(n *tree_node_t)`: (Over)Writes into the node in the secondary memory, if no data was changed will skip the writing process, returns an error or finish code: `(0, -1)`.

B-Tree Operations - Creating an empty B-Tree

- We use `btCreate()` to create a empty B-Tree, and since we only need to use `malloc()`, this operation takes $O(1)$.

```
bTree btCreate(void) {
    bTree b;

    b = malloc(sizeof(*b));
    b->isLeaf = 1;
    b->numKeys = 0;

    return b;
}
```

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 `true` if a given key exists in the B-Tree.
- The operation is split between the search of the key in the B-Tree and the comparison.
- This first function compares recursively from the *Root* to a leaf, using `searchKey` to get the index of the comparison key.

B-Tree Operations - Search II

```c
int btSearch(bTree b, int key) {
  int pos;
  /* have to check for empty tree */
  if(b->numKeys == 0) {
    return 0;
  }
  pos = searchKey(b->numKeys, b->keys, key);
  if(pos < b->numKeys && b->keys[pos] == key)
    return 1;
  else {
    return(!b->isLeaf && btSearch(disk_read(b->kids
        [pos]), key));
  }
}
```

B-Tree Operations - Search III

- The `searchKey` function takes the length of the array of keys, the array itself and the key to search. Then, iters through the array of keys, searching the key by a bisection algorithm.
- Finally, if found, it returns the key, otherwise it returns the higher key in the last iteration of the bisection.
- Since it's only reading the keys in the current node, there's no need to use `disk_read()`
- The `search` opertaion takes $O\left(log_{\alpha}\frac{n+1}{/}2\right)$, since we only have to access nodes all the way down to the needed, or close enough, leaf with the given key.

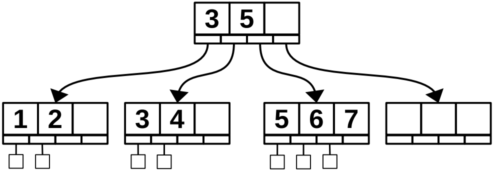
B-Tree Operations - Search IV

```
int searchKey(int n, const int *a, int key) {
   int lo = -1;
   int hi = n;
   int mid;
   while(lo + 1 < hi) {
      mid = (lo+hi)/2;
      if(a[mid] == key) {
         return mid;
      } else if(a[mid] < key) {
         lo = mid;
      } else {
         hi = mid;
      }
   }

   return hi;
}
```

B-Tree Operations - Search (Example)

- Enter into `searchKey` searching for 4 between indexes $(0, 2)$
- Since `a[mid] < key` for the next 2 iterations, then `lo = mid = 1`. And because `a[1] > key` then again `hi = mid = 1`, meaning that we stop the loop because `1 < 1` is `false`.
- Then we exit `searchKey` returning a position of 1.

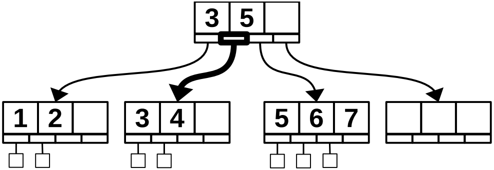



```
n = 2
a = {3, 5}
key = 4

/*
  Start / First / Second / Third
*/
lo = -1 ; 0 ; 1 ; 0 ;
hi = 2 ; 2 ; 1 ; 1 ;
mid = NULL ; 0 ; 1 ; 1 ;
```

B-Tree Operations - Search (Example)

- Due to `b->keys[pos] != key`, we have to search in the node with index `pos`. Where we repeat almost the same search process.
- Since `a[mid] < key` for the next 2 iterations, then `lo = mid = 1`. But this time `a[1] == key`, so we just return `mid`.
- Also, we have that `b->keys[pos] == key`, so we **finally return** 1.




```
key = 4 ; pos = 1

n = 2
a = {3, 4}

/*
  Start / First / Second
*/
lo = -1 ; 0 ; 1 ;
hi = 2 ; 2 ; 1 ;
mid = NULL ; 0 ; 1 ;
```

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

```
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 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

```
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 VI

- 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++;
    }
}
```

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

```c
    ...
    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-Tree Operation - Insert Node IX

```
        }

        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 $\Theta\left(\alpha logn\right)$, 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.

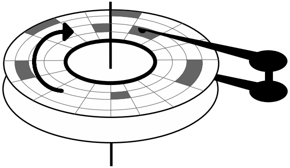


Figure: External storage with the sectors to access highlighted

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).