B-TREES

Vibha Satyanarayana, 01FB15ECS346

Abstract—Description of B-trees and their importance. Implementation of a B-tree as an array with each reference as an index of the array and implementation of a B-tree as a file with each reference made using the file pointer. Basic B-tree operations - create,insert,delete and search.

Index Terms— Storage,memory,btrees,files.

1 INTRODUCTION:

A B-tree is a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. Unlike self-balancing binary search trees, it is optimized for systems that read and write large blocks of data. B-trees are a good example of a data structure for external memory. They are designed to work well on disks or other direct access secondary storage devices.

Applications:

B-trees are used to create indexes to huge volumes of data. They are very useful in filesystems where optimal access to files are required by the file server and in databases as they help in faster disk operations.

2 DEFINITION:

A B-tree T is a rooted tree (whose root is T:*root*) having the following properties:

1. Every node x has the following attributes:
   - x.*n*, the number of keys currently stored in node x.
   - b. the x.*n* keys themselves, x.key_1,x.key_2,...,x.$key_{x.n}$, stored in nondecreasing order, so that x.*key*1 ≤ x:*key*2 ≤.....x.$key_{x.n}$.
   - x.*leaf* , a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.
2. Each internal node x also contains x.*n*+1 pointers x.c_1, x.c_2... .x.$c_{x.\,n+1}$ to its children. Leaf nodes have no children, and so their c_i attributes are undefined.
3. The keys x.key_i separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root x.c_i, then $k_1 \le x.key_1 \le k_2 \le x.key_2 \le .....\le x.key_{x.\,n} \le k_{x.\,n+1}$.
4. All leaves have the same depth, which is the tree's height h.
5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \ge 2$ called the ***minimum degree*** of the B-tree.
   - Every node other than the root must have at least t - 1 keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.
   - Every node may contain at most 2t - 1 keys. Therefore, an internal node may have at most 2t children. We say that a node is ***full*** if it contains exactly 2t -1 key

3 IMPLEMENTATION:

The following B-tree operations were implemented:

1. Create
2. Insert
3. Delete
4. Search

The B-tree was implemented in two ways:

1. **As an array:**
   Each node is stored in an array and is referenced by the index of the array.
2. **As a file:**
   Each node is stored in a data file and is referenced by moving the file pointer to the position where it is stored.

NODE STRUCTURE:

Every node in the B-tree stores:

- pos- position of the node in the array/file
- n - the number of keys currently present in the node
- keys - array that stores data (which includes key value and other attributes of a record)
- children - array that stores the position of every child in the array/file
- leaf - stores 0 (FALSE) if the node is not a leaf node and 1 (TRUE) if it is a leaf node.

Every key has an element called valid. This stores 0 if the key is invalid (if it has been deleted) and 1 if the key is valid.

B-TREE AS AN ARRAY:

An array is allocated to store all nodes. The index of the most recently added node is stored and updated. Every new insertion is made at the end of the filled array. Rootindex stores the index of the root of the B-tree and is updated each time the tree grows upwards. A variable t stores the degree of the B-tree.

A. CREATION:

This function is called when the first record has to be inserted. It creates the rootnode and sets leaf to 1. The index of the rootnode in the array is stored.

B. INSERTION:

This operation requires three functions- splitChild, insert and inset_nonfull.

SplitChild - this function takes two parameters - node x and index i. x is the node whose child (y) has 2t children. Node y originally has 2t -1 keys but is reduced to t - 1 keys by this operation. Node z takes the t largest children from y, and becomes a new child of x, positioned just after y in x's table of children. The median key of y moves up to become the key in x that separates y and z.

Insert- the record that has to be inserted is sent to this function. In case root node is full: the root splits and a new node s (having two children) becomes the root. Splitting the root is the only way to increase the height of a B-tree. The procedure finishes by calling insert_nonfull to insert key k into the tree rooted at the nonfull root node. Insert_nonfull recurses as necessary down the tree, at all times guaranteeing that the node to which it recurses is not full by calling splitChild as necessary. The function in sert_nonfull inserts key k into node x, which is assumed to be nonfull when the procedure is called. The operation of insert and the recursive operation of insert_nonfull guarantee that this assumption is true.

Insert_nonfull- takes node x and key k as parameters. If x is a leaf node, k is inserted into x. If x is not a leaf node, then k must be inserted into the appropriate leaf node in the subtree rooted at x. The procedure calls itself recursively to descend to the correct child. In case the child is full, splitChild is called to split it into two nonfull children. Then, the correct child out of the two is determined in order to find the descending path. The procedure recurses till it finds the appropriate subtree to insert k into.

C. DELETION:

Each key stores a variable to store the validity of the key. The valid variable stores 0 if the key was deleted and 1 if it exists. The delete function calls search to get the position of the node and the key in the node that stores the record. The valid variable of this key is set to 0 to indicate that it has been deleted.

D. SEARCH:

Using a linear-search procedure, the smallest index i such that $k \leq x.key_i$ is found by traversing through the node. If the key is found, the record is returned. If x is the leaf node and the key isnt found NIL is returned. If not, the recursion descends down to the ith child and repeats the same procedure.

B-TREE AS A FILE:

All nodes are stored in a file. The position of the most recently added node is stored and updated. Every new insertion is made by appending to the file. Each key stores its position in the file. If any changes are made to the key, its written back into the file by moving the file pointer to the position stored in the key and re-writing it in its position. Similarly, a key is read from the file by moving the file pointer to its position and reading one node at that position.

Rootindex stores the index of the root of the B-tree and is updated each time the tree grows upwards. A variable t stores the degree of the B-tree. Create, insert, delete and search functions work the same way they do in case of array implementation.

4 THEORETICAL ANALYSIS:

TIME COMPLEXITY OF EVERY B-TREE OPERATION:

Search	$O(t \log_t n)$
Create	O(1)
SplitChild	O(1)
Insert	$O(t \log_t n)$
Insert_nonfull	$O(t \log_t n)$
Delete (by setting valid bit - no rearrangement of tree)	$O(t \log_t n)$

5 RESULT:

The following table gives time (in milliseconds) required to do basic B-tree operations in case of array implementation. The code was run for degree t=4 and t=3.

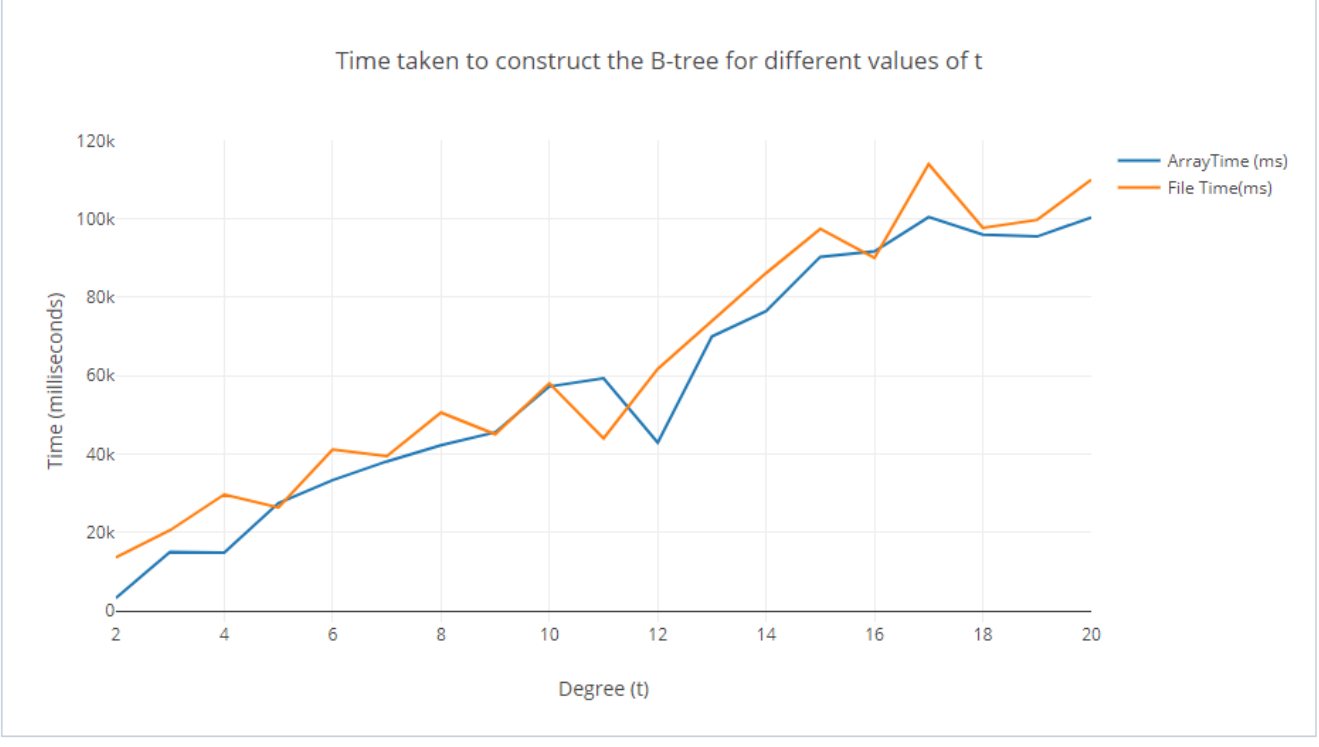
DEGREE (T)	NUMBER OF RECORDS	CONSTRUCTION	INSERTION	DELETION	SEARCH
4	20	59	1	8	0
4	100	169	2	8	5
4	1000	2747	7	7	7
4	100000	15 SECONDS	585	12	10
3	20	63	1	9	6
3	100	172	1	9	7
3	1000	3475	9	8	7
3	100000	15 SECONDS	796	14	10

The following table gives time (in milliseconds) required to do basic B-tree operations in case of file implementation. The code was run for degree t=4 and t=3.

DEGREE (T)	NUMBER OF RECORDS	CONSTRUCTION	INSERTION	DELETION	SEARCH
4	20	193	39	271455	5
4	100	591	38	444762	7
4	1000	7545	44	157933	6
4	100000	18	836	-	8
3	20	32130	34	393366	6
3	100	588	39	634324	6
3	1000	7522	46	74156	6
3	100000	22 SECONDS	847	-	8

*(deletion could be more due to the time required to read from the file and update the file after search)

THE FOLLOWING GRAPH SHOWS THE TIME REQUIRED FOR DIFFERENT VALUES OF DEGREE (T) FOR BOTH IMPLEMENTATIONS.



6 REFERENCES:

https://en.wikipedia.org/wiki/B-tree
http://www.geeksforgeeks.org
Introduction to Algorithms- Thomas H. Cormen, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein