Assignment 3 *Rutha Prasad*

UE15CS311 *01FB15ECS367*

<Page Number 1>

Question Statement:

Implement a B tree with the following operations:

1. create
2. insert
3. delete
4. search

Experiment with different values of t - the minimum degree of each node:

a) Implement the B-tree as an array **b)** Implement the B-tree as a file.

Approach/Heuristics:

- Deletion policy: (Logical deletion)
  Mark the node as deleted, and gets a new node for insertion: from the end of the array. The deleted nodes become garbage.

- Records policy: Records are of fixed size.

- Key policy: Key is a single field.

- # of passes
  - A single pass policy for insertion.
  - A single pass/a two pass policy deletion depending on the key being in a leaf or internal node.

- # of nodes
  - In file implementation, a file stores all the nodes.
  - # of nodes depends on the maximum file size allowed by the system.
  - In array implementation, an array of nodes is created at the time of tree creation.
  - The size of the array depends on the approximate # of records inserted into the tree and the degree t of the tree. (Provided by the client during compilation).
  - The degree of the tree decides the maximum # of records a node can hold.
  - # of nodes = approximate # of records /max # of records stored in a node.

- Space complexity: θ(n)→θ(# of keys)

- Testing Efficiency
  - # of records : 1 Million
  - T values : 3,5,25,50

Algorithms and Complexity:

Best case height of the b-tree is : $\log_t(n+1)$→max # of keys in each node (2t-1)

Worst case height of the b-tree is : $\log_t(n+1/2)$→min # of keys in each node (t-1)

Therefore ,Height →O(logn)

(Assume updating ,reading, writing overheads into the file or array structures are ignored)

Algo INSERT(tree,record)

Worst case = O(logn)

θ (traverse total height to insert + split child (O(t)) +insert_non_full O(2t))

→θ(logn+t+2t)→O(logn)

Best case =O(logn)

O(find height at which to insert + no split child + insert_non_full O(1))

→O(logn)

Average Case = O(logn)

Algo SEARCH(tree,key)

Worst case= O(logn)

θ (not found but traverse total height)→θ(logn)

Best case=O(logn)

O(traverse some height to internal node)→O(logn)

Average Case=O(logn)

Algo DELETE(tree,key)

Worst case = O(logn)

θ (traverse total height + merge children(==t-1) at each level (O(2t)) + compact each node(O(2t)))

→θ(logn+logn*2t +2t)→O(logn)

Best case = O(logn)

O(find height at which to delete without merge/compacting node)

→O(logn)

Average Case = O(logn)

Algo CREATE(tree)

Time complexity: O(initialize rootO(t))→O(t)

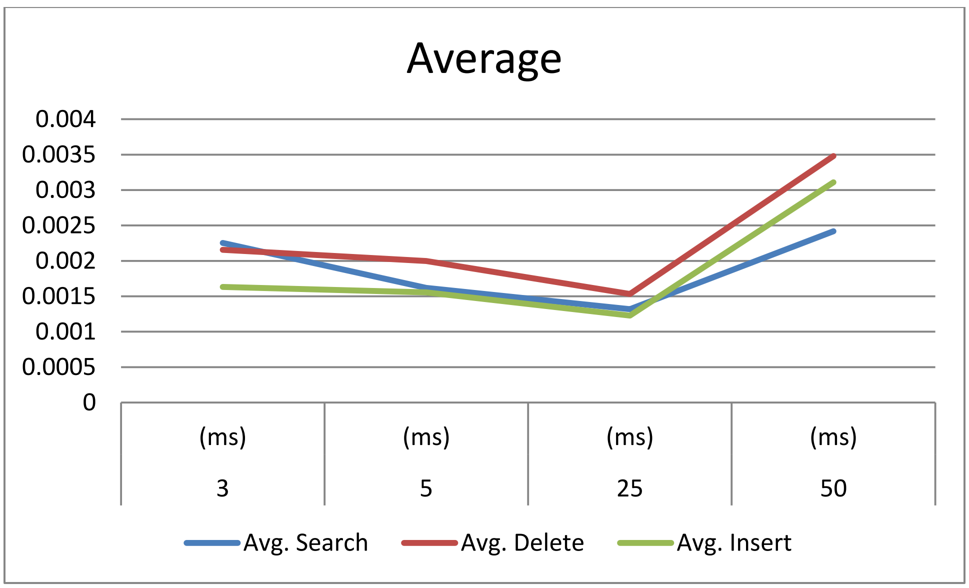
Average Case = O(t)

Testing Efficiency:

- # of records = 1M
- T values
  - t=3
    - Max # of nodes = 200k
    - Max. Height = 13
  - t=5
    - Max # of nodes = 120k
    - Max. Height = 9
  - t=25
    - Max # of nodes = 30k
    - Max. Height = 5
  - t=50
    - Max # of nodes = 10k
    - Max. Height = 3

Array Implementation:

T	3	5	25	50
ARRAY	(ms)	(ms)	(ms)	(ms)
Insert 1M records	1590.253158	1445.2069	2048.001343	8891.539347
Avg. Search	0.002254	0.001618	0.001319	0.002418
Avg. Delete	0.002155	0.001999	0.001533	0.003481
Avg. Insert	0.001631	0.001555	0.001227	0.00311

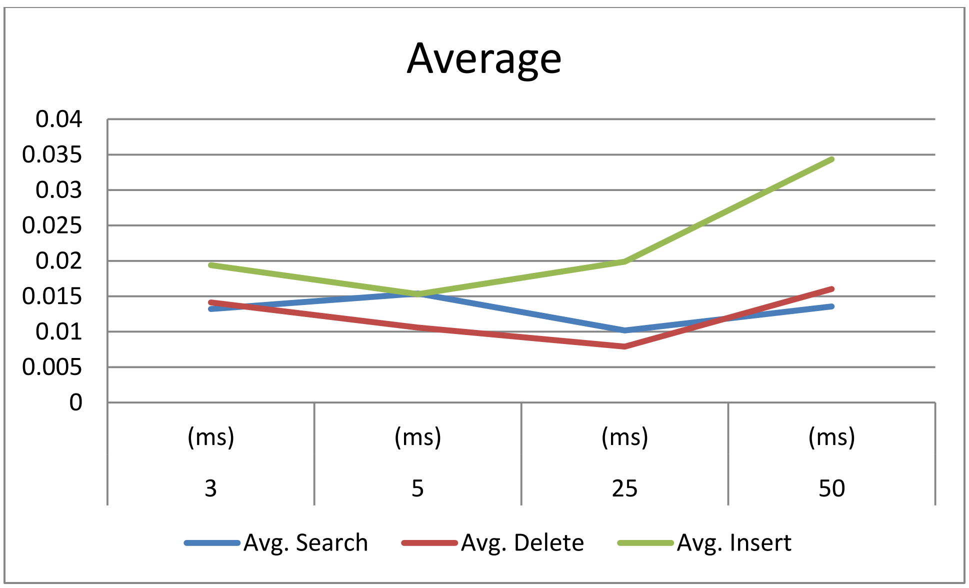


Conclusion:

- For larger values of t, the number of keys in each node increases.
- Therefore the time increases since linear search in each node starts becoming a big overhead.
- The height decreases, but the overhead of sequential search is large.
- Optimum values such as t=25 , give the least average times for all operations.

File Implementation:

T	3	5	25	50
FILE	(ms)	(ms)	(ms)	(ms)
Insert 1M records	265.496831	216.158886	252.630793	333.93839
Avg. Search	0.013197	0.015383	0.010185	0.013562
Avg. Delete	0.014124	0.010588	0.007898	0.016019
Avg. Insert	0.019386	0.015298	0.019881	0.034347



Conclusion:

- For larger values of t, the number of keys in each node increases.
- Therefore the time increases since linear search in each node starts becoming a big overhead.
- The height decreases, but the overhead of sequential search is large.
- Optimum values such as t=25 , give the least average times for most operations.
- File input-output operations take more time, since the file is in the disk. Therefore the avg insert and avg delete times are higher than that of Array implementation.
- Avg.Insert shows a steep increase when t increase, as memory read overhead is high.