Tirgul 6

B-Trees – Another kind of balanced trees

Some notes regarding Home Work

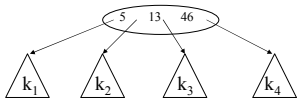
Motivation

- Primary memory (RAM) : very fast, but costly
  Secondary storage (disk) : very cheap, but slow
- Problem: a large D.B. must reside partially on disk. But disk operations are very slow.
- Solution: take advantage of important disk property -Basic read/write unit is a page (2-4 Kb) - can't read/write less.
- Thus when analyzing D.B. performance, we consider two different measures: CPU time and number of times we need to access the disk.
- Besides, B-trees are an interesting type of balanced trees...

B-Trees

B-Tree: a balanced search tree whose nodes can have many children:

- A node x contains $n[x]$ keys, and has $n[x]+1$ children ($c_1[x]$, $c_2[x]$, … , $c_{n[x]+1}[x]$).

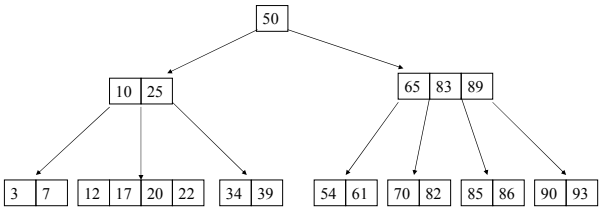


- The keys in each node are ordered, and relate to their left and right sub-trees like regular search trees: if k_i is any key stored in the sub-tree rooted at $c_i[x]$, then:

$$k_1 \le key_1[x] \le k_2 \le key_2[x] \le \ldots \le key_{n[x]}[x] \le k_{n[x]+1}$$

- All leaves have the same depth h (the tree's height)
- There is a parameter t (an integer) such that:
  - Every node (besides the root) has at least t-1 keys (i.e. t children)
  - Every node can contain at most $2t$-1 keys ($2t$ children).

Example



t=3

B-Trees and disk access (last time...)

- Each node contains as many keys as possible without being larger than a single page on disk.
- Whenever we need to access a node – load it from the disk (one read operation), after changing a node – rewrite it to the disk.
- (The root is always in memory.)
- For example, say each node contains 1000 keys – and the root has 1001 children, each of which also has 1001 children. Thus with just 2 disk accesses we are able to access ~1000^3 records.
- Operations are designed to work in one pass from the root to the leaves – we do not need to backtrack our steps. This further reduces the number of disk accesses we make.

The height of a B-Tree

Theorem: If $n \ge 1$, then for any B-tree of height h with n keys and minimum degree $t \ge 2$:

$$h \le \log_t((n+1)/2)$$

Proof: Each child of the root has at least t children, each of them also has at least t children, and so on. Thus in every sub-tree of the root there are at least $\sum_{i=1}^{h} t^{i-1}$ nodes. Each of them contains at least t-1 keys. The root contains at least one key and has at least two children, so we have:

$$n \ge 1 + 2(t-1)\sum_{i=1}^{h} t^{i-1}$$

$$= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) = 2t^h - 1$$

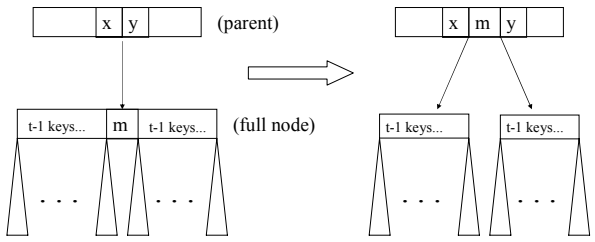
B-Tree Search

- Search is done in the regular way: In each node, we find the sub-tree in which our value might be, and recursively find it there.
- Performance:
  $O(t*h) = O(t \log_t n)$ - total run-time, out of which:
  $O(h) = O(\log_t n)$ - disk access operations

B-Tree Insert

- Since every node contains many keys, we simply insert a key to the appropriate leaf in a natural order. (Not creating a new leaf)
- What might be the problem?
- If the leaf if full (i.e. contains already contains 2t-1 keys before the insert). What do you suggest?

B-tree split

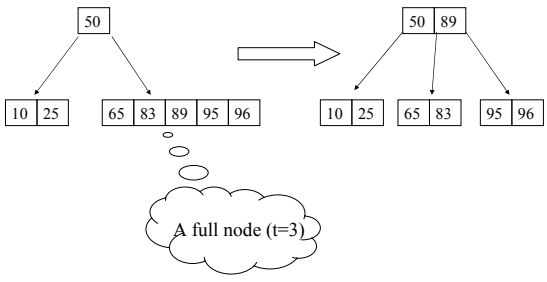


Notice that the parent has many other sub-trees that don't change.

B-Tree Split

- Used for insertion. This operation verifies that a node will have less than *2t-1* keys.
- What we do is *split* the node into two nodes, each with *t*-1 keys. The extra key goes into the node's parent (We assume the parent is not full)
- To split a node *x* (look at the previous slide for illustration), take $key_t[x]$ (notice it is the *median* key). All smaller keys (exactly *t-1* of them) form one new (legal) node, the same with all larger keys. $key_t[x]$ goes into *x*'s parent.
- If the node we split is the root, then a new root is created. This new root contains only one key.

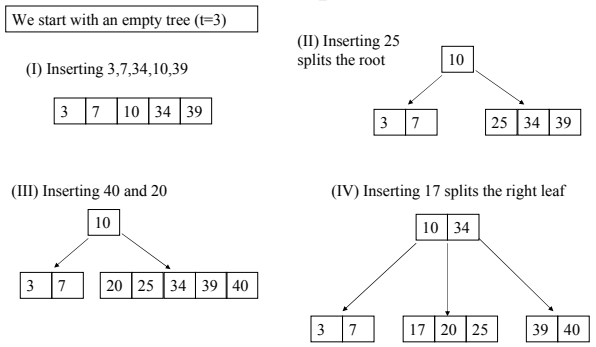
Example



B-Tree Insert

- We insert a key only to a leaf. We start from the root and go down to the appropriate leaf.
- On the way, before going down to the next node, we check if it is full. If so, we split it (its father is non-full because we checked this before going down to the father).
- When we reach the correct leaf, we know that the leaf is not full, so we can simply insert the new value to the leaf.
- Notice that we may need to split the root, if it is full. In this case, the tree's height increases (but the tree remains completely balanced!). That's why we say that a B-tree grows from the root, in contrast to most of the trees, who grow from the leaves...

Example



B-Tree Insert (cont.)

- Performance:
  - Split:
    - three disk accesses (to write the 2 new nodes, and the parent)
    - $O(t)$ - total run time
  - Insert:
    - $O(h)$ - disk accesses
    - $O(t\log_t n)$ - total run time
    - Requires $O(1)$ pages in main memory.

B-Tree delete

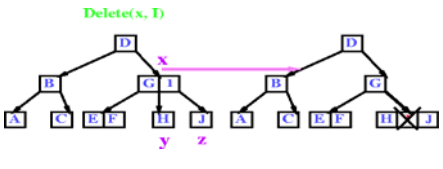
- Once again we'd like to do one recursive pass (almost true).
- For that purpose, we keep an invariant, that except in the root, a node we deal with contains always t (rather then $t-1$) keys. To keep the invariant we might need to "push down" a key to the node we are about to enter. (why can we do that?)

B-Tree delete (cont')

- Many cases of deleting k
- 1. k is in a leaf – simply delete it (why no problem?)
- 2. k is in internal node x
  - a. the child y that precedes k has t or more keys,
    Find the predecessor k' of k in the sub tree of y. Delete k' and replace k by k'.
  - *b.* similar for the child z the predecessor of k.
  - c. Both y and z have $t-1$ keys, merge y,k,z into one node of size $2t-1$, then delete k.

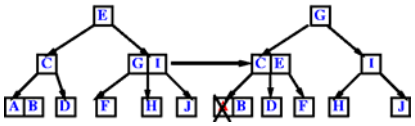
B-Tree delete (cont')

- 2. k is in internal node x
  - c. Both y and z have $t-1$ keys, merge y,k,z into one node of size $2t-1$, then delete k.



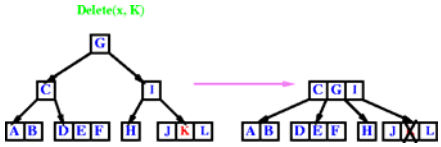
B-Tree delete (cont')

- 3. k is not in node x (how to keep the minimum t keys invariant). Determine the relevant child $c_i[x]$, if it has t or more keys cool, otherwise:
  - a. $c_i[x]$ immediate left or right sibling has t or more keys . Shift a key from sibling to $c_i[x]$ through x,



B-Tree delete (cont')

- 3. *k* is not in node *x* (how to keep the minimum *t* keys invariant). Determine the relevant child $c_i[x]$, if it has t or more keys cool, otherwise:
  - b. $c_i[x]$ immediate left and right sibling have t-1 keys . Merge with father.



Programming note

- Some of you made SortedMap a derived class of LinkedList.
- This is mistake.
- When do we use inheritance?
- The rule of thumb is the "is-a" relationship.
- Is it true that: "A SortedMap is-a Map"?
  - Naturally every method that Map implements SortedMap implements as well.
- Is it true that "A SortedMap is-a LinkedList"?
  - No, A sortedMap might be implemented as a linked list or by other means (such as?).
  - Indeed some of the methods that LinkedList implements are not implemented by SortedMap.
- A good reference "Thinking in Java/Bruce Eckel"
  - Link from the course homepage.

Theoretical mistakes

- $n \geq log(n)$ implies that for every c_1, c_2 there exists an n_0 from which $c_1 n > c_2\ log(n)$ – this is a mistake hidden as not being formal enough. Why?
- Lots of people mistake having to prove for all *c* with proving for a specific *c*.
- n goes to infinity faster than *log(n)* therefore … This is not a proof (at most it can serve as an intuition).
- From Infi' we know… Please quote exactly the statement taught in Infi' you are using. In most cases the statement will have a name.

Login problems

- You should state your login on the ex' (theoretical as well)
- Use only login from cs.
- Some people didn't even write a name.