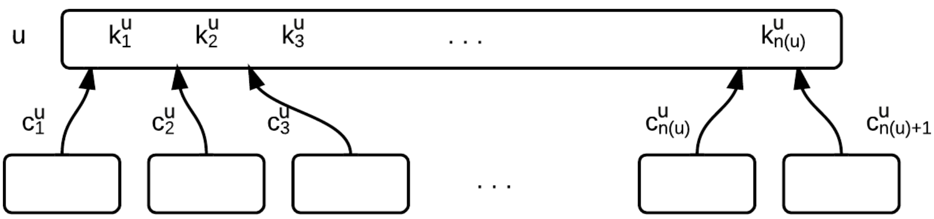
CSE 5095: Research Topics in Big Data Analytics
Lecture 06 (Feb 06, 2014)
Prof. Sanguthevar Rajasekaran
Prepared By: Abdullah-Al Mamun

Data structures play a key role in data processing and algorithms. Tree-based data structures exist in both in-core and out-of-core settings. 2-3 tree, B-tree, red-black tree, etc. are examples of widely used tree-based data structures. Popular operations such as insert, delete, search, etc. can be processed in these trees in time proportional to the heights of them. Since data are transferred in blocks of B items, these operations will take $\Omega(\log_B N)$ I/O operations in an external memory model.

One of the most widely used out-of-core memory data structures is a B-tree. A B-tree is a balanced search tree which has a height of $\mathcal{O}(\log_t n)$ where n is the number of keys and t is a parameter characterizing the B-tree. We'll choose t to be $\Theta(B)$. A B-tree has the following properties:

1. Any node u has the following information in it:
   n_u : the number of keys in u.
   $leaf_u$: a bit whose value is 1 if u is a leaf and 0 otherwise.
   Keys : $k_1^u, k_2^u, ..., k_{n_u}^u$ in non decreasing order.
   If u is not a leaf, it has pointers to $(n_u + 1)$ children, namely $c_1^u, c_2^u, ..., c_{n_u+1}^u$.



2. All the leaves are at the same level.

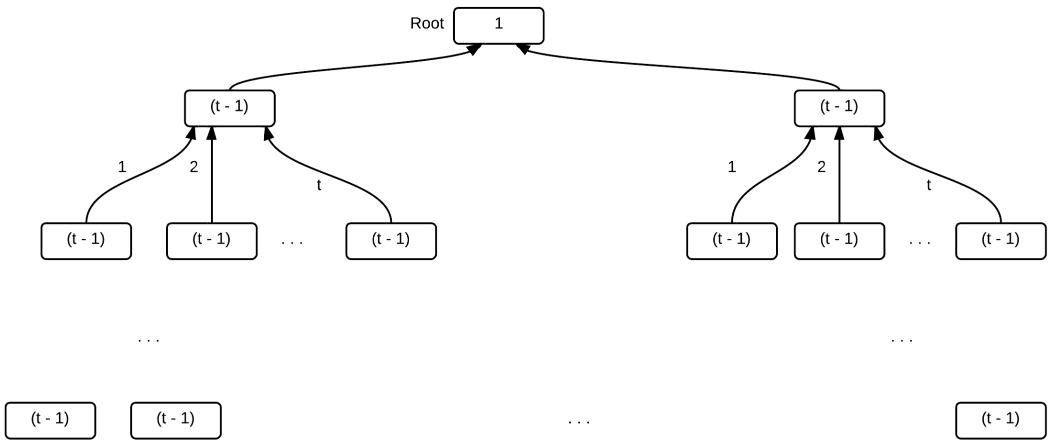
3. Let q_i be any key in c_i^u, $1 \leq i \leq (n_u + 1)$. Then $k_{i-1}^u \leq q_i \leq k_i^u$. Assume that $k_0^u = -\infty$ and $k_{n_u+1} = \infty$ for any node u in the tree.

4. Degree is defined with a parameter t. Any node other than the root has $\geq (t-1)$ and $\leq (2t-1)$ keys. The root has ≥ 1 key.

Definition: A node is *Full* if it has $(2t - 1)$ keys.
Note: Pick a value for t such that the size of a full node is the same as the block size B, i.e. $t = \Theta(B)$.

Lemma: If h is the height of a B-tree with n keys, then $h \leq \log_t \frac{n+1}{2}$.

Proof: To prove the upper bound, consider the case where the least number of keys are present in the nodes.



number of nodes	level number
1	0
2	1
$2t$	2
...	...
$2t^{h-1}$	h

The minimum number of keys in a B-tree with the parameter t is thus
$= 1 + 2(t-1) + 2t(t-1) + ... + 2t^{h-1}(t-1)$
$= 1 + 2(t-1)[1 + t + t^2 + ... + t^{h-1}]$
$= 1 + 2(t-1)\frac{t^h-1}{t-1}$
$= 2t^h - 1$

So
$n \geq 2t^h - 1$
$t^h \leq \frac{n+1}{2}$
$h \leq \log_t \frac{n+1}{2}$
[Proved]

In a similar manner we can prove that the height of a B-tree with the parameter t is $\Omega(\log_t n)$.

Search(u, k)**:** (look for k in the subtree rooted at u)
The algorithm used for this search operation is :

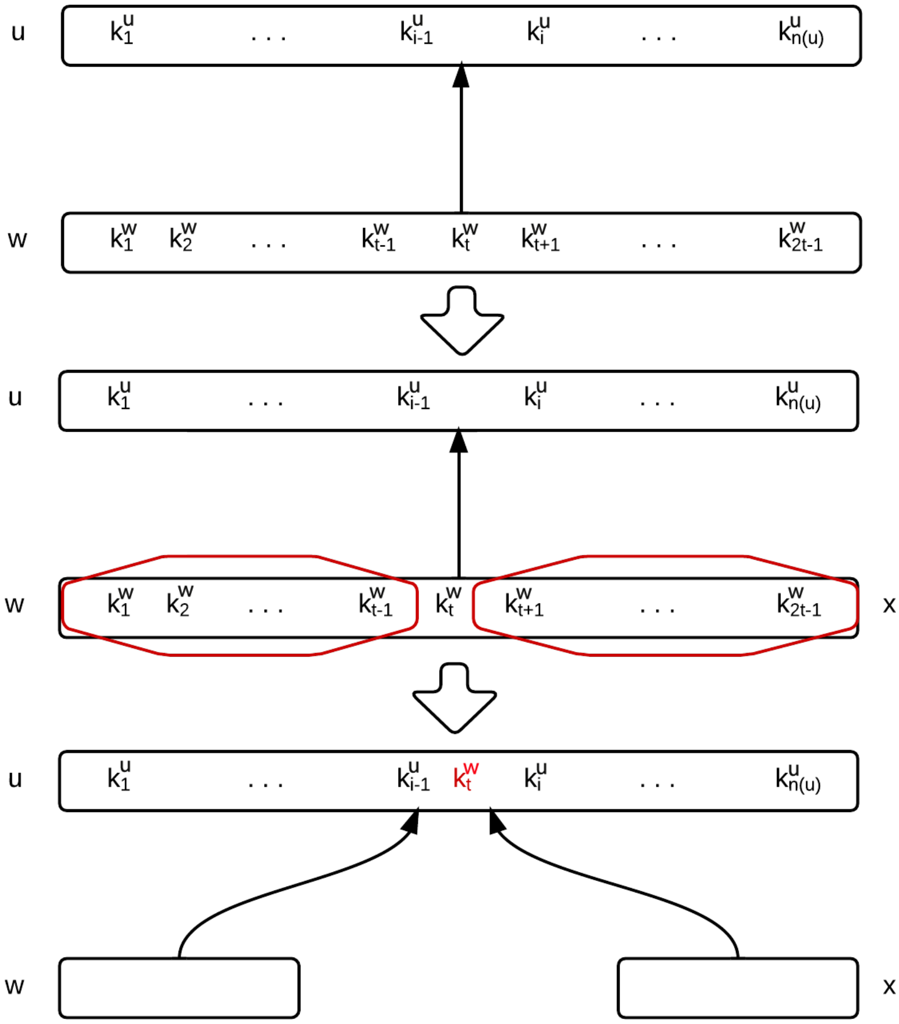
procedure SEARCH(u, k)
  if $k = k_i^u$ for some $i, 1 \le i \le n_u$ **then**
    Output (u, i)
  else if u is a leaf and $k \notin u$ **then**
    Output NIL
  else
    $k_0^u \leftarrow -\infty$ ▷ for all nodes u
    $k_{n_u+1}^u \leftarrow \infty$ ▷ for all nodes u
    Using a binary search identify i such that $k_{i-1}^u \le k \le k_i^u$
    DISK_READ(c_i^u)
    SEARCH(c_i^u, k)
  end if
end procedure

SplitNode(u, i, w)**:** An operation called *Split Node* will be used in *Insert* operations. This method is called when w is full. This method splits w into 2. u is the non-full parent of w and w is the i^{th} child of u.

procedure SPLITNODE(u, i, w)
  Create a new node x
  $leaf_x \leftarrow leaf_w$
  for $1 \le j \le (t-1)$ **do**
    $k_j^x \leftarrow k_{j+t}^w$
  end for
  for $j \leftarrow n_u$ down to i **do**
    $k_{j+1}^u \leftarrow k_j^u$; $c_{j+1}^u \leftarrow c_j^u$
  end for
  $k_i^u \leftarrow k_t^w$; $c_{i+1}^u \leftarrow x$
  if w is not a leaf **then**
    for $j \leftarrow 1$ to t **do**
      $c_j^x = c_{j+t}^w$
    end for
  end if
  $n_u \leftarrow n_u + 1; n_x \leftarrow t - 1; n_w \leftarrow t - 1$
end procedure

While inserting a key k into a B-tree we always make sure that the node that we recurse to is not full. This will help us in ensuring that the insert operation can be processed in one forward traversal through a path in the tree (starting from the root and ending in a leaf). We thus have two different procedures called **INSERT** and **INSERT_NONFULL**. The second procedure is called on a non full subtree (rooted at say u). The first procedure is called at the root. This procedure splits the root into two if the root is full, and invokes the second procedure. **INSERT_NONFULL** identifies the subtree w of the current node u that k belongs to and recursively calls itself on the

Figure 0.1: Split node into 2 nodes

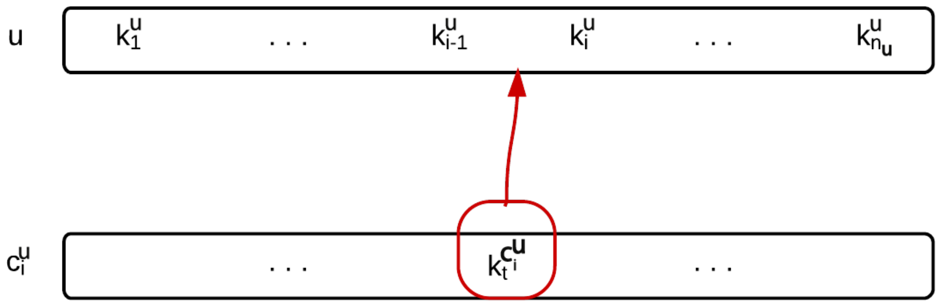


subtree w. Before making this recursive call, it splits w if it is full.

Insert(T, k)**:**

procedure INSERT(T, k)
    $r \leftarrow \text{root}(T)$
    if $n_r = (2t - 1)$ **then**
        Create a new node s
        $n_s \leftarrow 0$
        $leaf_s \leftarrow 0$

$c_1^s \leftarrow r$
SPLITNODE$(s, 1, r)$
root$(T) \leftarrow s; r \leftarrow s$
end if
INSERT_NONFULL(r, k)
end procedure
procedure INSERT_NONFULL(u, k) ▷ u has to be non-full
if u is a leaf **then**
Insert k at the right place
$n_u \leftarrow n_u + 1$
end if
if u is not a leaf **then**
Choose i such that $k_{i-1}^u \leq k < k_i^u$ ▷ k has to be inserted into c_i^u
end if
if $n_{c_i^u} = 2(t-1)$ **then**
SPLITNODE(u, i, c_i^u)
if $k \geq k_i^u$ **then**
$i \leftarrow i + 1$
end if
end if
INSERT_NONFULL(c_i^u, k)
end procedure



Delete(u, k)**:** This operation is analogous to insertion. Deletion works on nodes having at least t keys so that at least $(t-1)$ keys are present in a node after the operation.

more details on this topic will be discussed in the next lecture