CSE 5095: Research Topics in Big Data Analytics
Lecture 07 (Feb 11, 2014)
Prof. Sanguthevar Rajasekaran

Notes prepared By: Priya Periaswamy

In the last lecture, we looked into basic operations on B-trees: Searching and insertion. In this lecture we will look into the operation of deleting a key from the B-tree.

Delete(x, k):
→ Similar to search and insert operations, when we say **Delete**(x, k), it implies deleting a key k from the subtree rooted at x.
→ While inserting a key k into a B-tree, when a recursive call is made to any node x, we always make sure that the node is not full. In the delete process, when a recursive call is made to any node x, we should ensure that the node has at least t keys, to ensure that the deletion can be performed in one forward pass through a path in the tree starting from the root.

Various cases of deleting keys from a B-tree:

Case 1a:

If x is a leaf, and $k \in x$, then delete k from x.

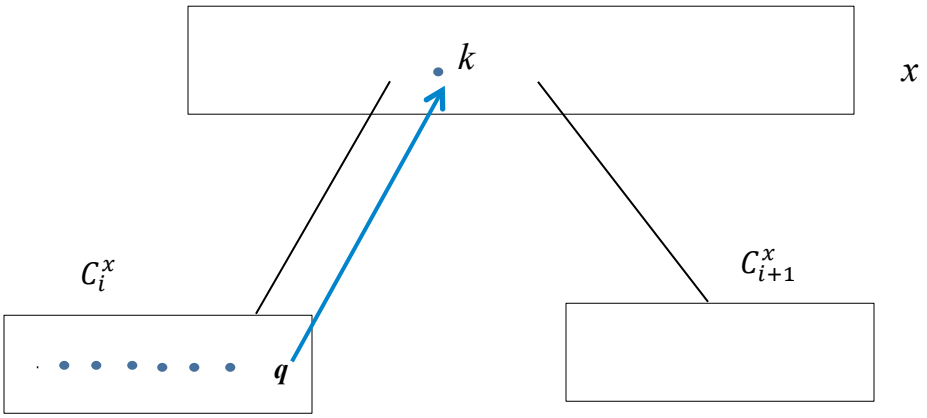
Let n_x is the total number of keys in x, then

$n_x = n_x - 1$; /*that is the number of keys in x is decreased by 1*/

Case 1b: If x is a leaf and x does not contain k, then report error and quit.

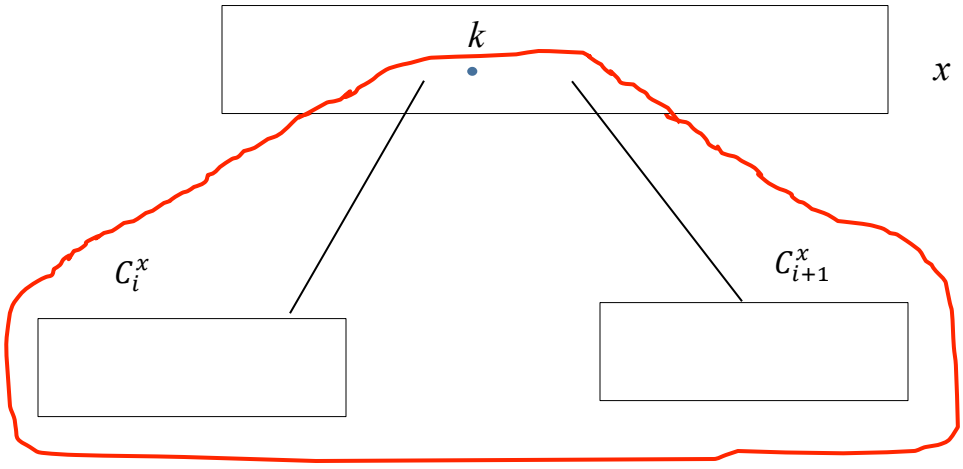
Case 2:
$k \in x$ and x is not a leaf.
Let $k = k_i^x$

Case 2a:
Check if the left sub tree C_i^x has atleast t keys. If this is the case, let q be the largest key in C_i^x. Replace k with q and recursively delete q from C_i^x (i.e., delete(C_i^x, q)).



Case 2b:
Check if the right sub tree C_{i+1}^x has atleast t keys. This case is symmetric to *case 2a*. Here we look for the smallest key in C_{i+1}^x to replace k.

Case 2c:
Both subtrees C_i^x and C_{i+1}^x have (t-1) keys each.



In this case,

i) Merge C_i^x and C_{i+1}^x together with k to form a new node. This node will have keys of C_i^x, followed by k, followed by keys of C_{i+1}^x.

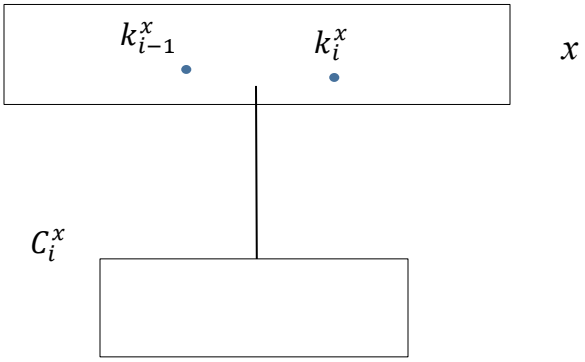
ii) The node x loses k and the pointer to C_{i+1}^x.

iii) Recursively delete (C_i^x, k)

Case 3:

$k \notin x$ and x is not a leaf.

Identify i such that $k_{i-1}^x \leq k \leq k_i^x$

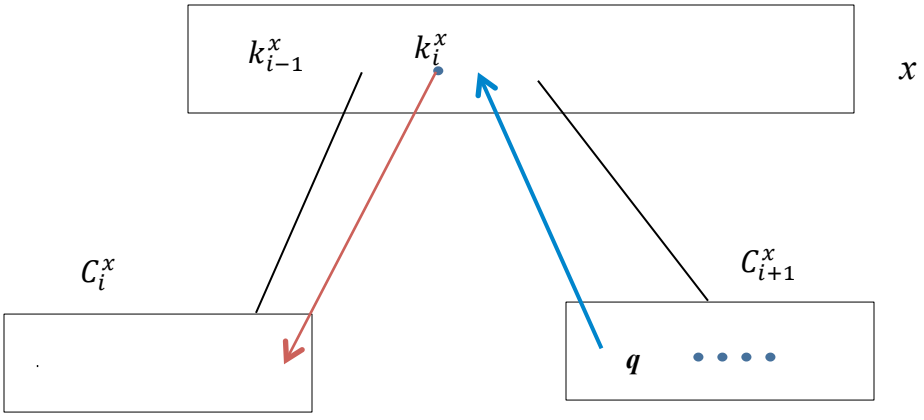


Case 3a:

Check if the left sub tree C_i^x has atleast t keys. In this case, call delete recursively on C_i^x. (ie., delete(C_i^x, k)).

Case 3b:

If C_i^x has (t-1) keys and C_{i+1}^x has $\geq t$ keys:



Let q be the smallest key of C_{i+1}^x. In this case,

i) Replace k_i^x with q

ii) Move k_i^x as the largest key of C_i^x; also move the first (i.e., leftmost) subtree of C_{i+1}^x as the last (ie., rightmost) subtree of C_i^x

iii) Recursively delete (C_i^x, k)

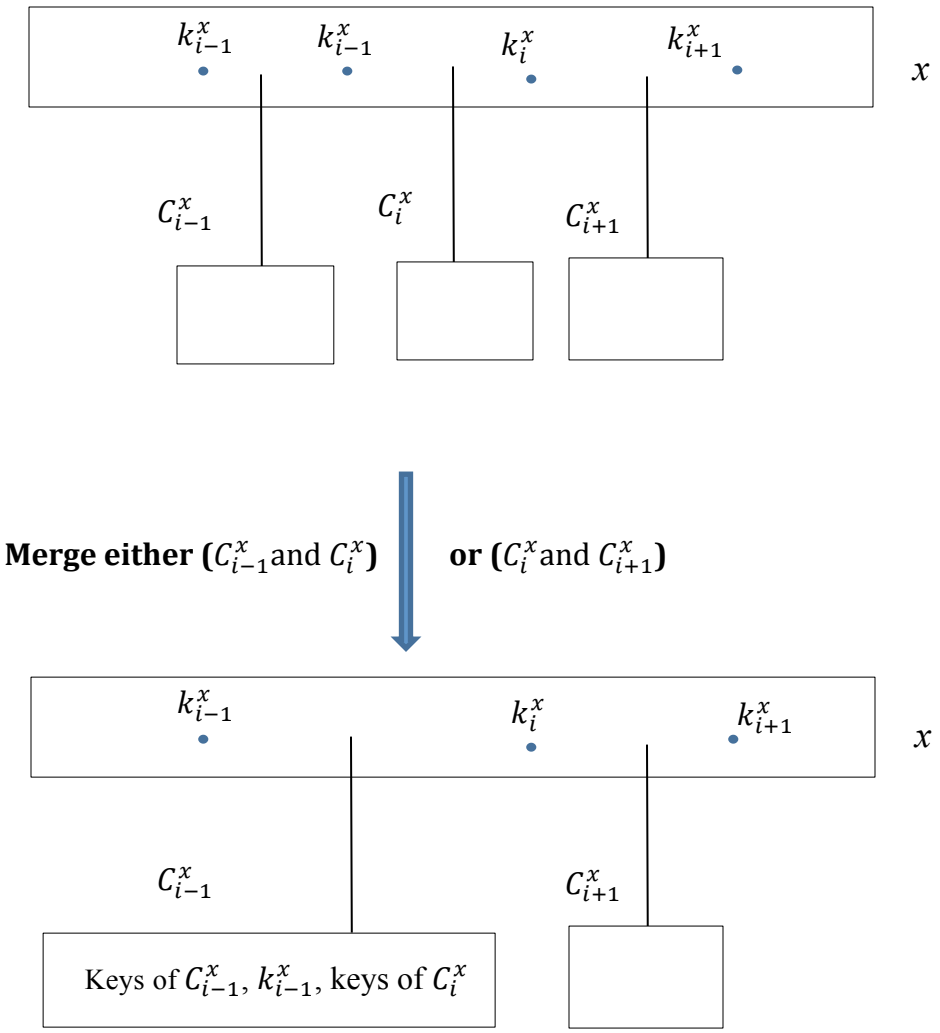
Thus the ordering of keys is preserved and also we ensure that the nodes have at least t keys.

Case 3c:

If C_i^x has (t-1) keys and C_{i-1}^x has $\geq t$ keys:
This case is symmetric to case 3b

Case 3d:

C_{i-1}^x, C_i^x, and C_{i+1}^x have exactly (t-1) keys each:



After merging, we can recursively delete(C_{i-1}^{x}, k).

Note: If at any time, x has 0 keys, then x is deleted and its only child becomes the root. This is an exception and happens only at the root decreasing the height of the tree by one and preserving the property that the root of the tree contains at least one key.

Theorem:
In a B-Tree, the operations Insert, delete, search, FindMin (where we can get the leftmost key) and FindMax (we can access the rightmost key in the tree) take $\Theta\left(\frac{\log n}{\log B}\right)$ I/O operations each.

The Minimum Spanning Tree (MST) Problem
PRIM's Algorithm:
Next we will move on to the problem of finding a minimum spanning tree (MST) of a weighted graph G in the out of core computing setting.

Input: A weighted undirected graph $G(V,E)$
Output: A minimum spanning tree of G

Prim's algorithm always has a single subtree which is grown one node at a time. The tree starts with the lightest edge. At any time we add the lightest edge going out of the tree (i.e., the lightest edge that goes from a tree node to a non-tree node).

This strategy is greedy since, in each iteration, the tree gains an edge that adds the minimum amount possible to the tree's weight. During execution of the algorithm, all vertices that are *not* in the tree reside in a min-priority queue Q. For each vertex v, $key[v]$ is the minimum weight of any edge connecting v to a vertex in the tree.

Also, we define a data structure *Near* as follows. For any non-tree node *u*, *Near[u]* is following:
Near[u] = the nearest tree neighbor of the vertex u;

PRIM's Algorithm to find MST for the graph G(V,E):

Procedure PRIM's_MST(G)
Step 1: Find the lightest edge e = (a,b) in the graph. Add it to the tree T.
T={a,b}
Step 2:
For every u∈V-{a,b} do
if cost(u,a) < cost(u,b) then
Near[u] = a;
else
Near[u] = b;
endIf
endFor

Step 3:

For every u∈V−{a,b} do

Insert u into a priority queue Q with a key of cost (u, Near[u]);

endFor

Step 4:

For i = 1 to |V| −2 do

Find the smallest key in Q;

Let u be the corresponding node;

Insert u into T;

Step 5: For w ∈ Adj(u) do

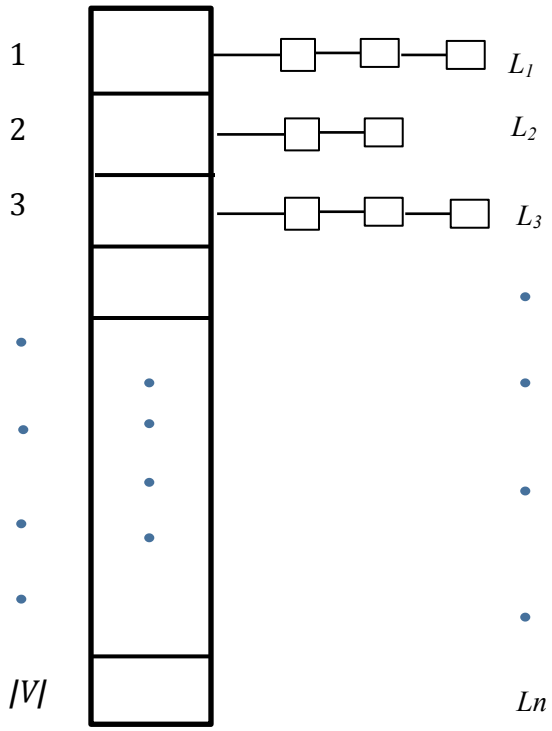
if cost(w, Near[w]) > cost(w, u) then

Near[w] = u;

endFor

end procedure

Now let's think about the representation of a graph in the disk. Let the graph be represented as the adjacency lists of all nodes as in the following figure.



Here $n = |V|$

Please note that the size each of the lists L_1, L_2…L_n need not be an integral multiple of the block size *B.*

Assumption:
Assume that there is space in core memory for the priority queue *Q.* Note that the size of Q is $O(n)$.
Let's analyze the time taken for each of the steps in the Prim's algorithm according to the assumption we made above.

1. Step *1* takes $O\left(\frac{|E|}{B}\right)$ I/O operations.
2. Step *2* and *3* takes $O\left(\frac{|V|}{B}\right)$ I/O operations.
3. Let d_u be the degree of u, $\forall u \in V$.
   In Step *5*, we spend $\lceil \frac{du}{B} \rceil$ I/O operations

   Total number of I/O's in step *4*

$$= \sum_{u \in V} \lceil \frac{du}{B} \rceil$$
$$\leq \sum_{u \in V} (\frac{du}{B} + 1)$$
$$\leq \frac{|E|}{B} + |V|$$

   If $M \neq \Omega(n)$, we can use a B-tree to implement priority queue. In this case, figuring out the I/O complexity is left as an exercise.

Kruskal's Algorithm:

We will next look at the Kruskal's algorithm which is widely used to find the MST. Before we look at the algorithm, let's revisit the union-find data structure.

As we know, Kruskal's algorithm uses union-find data structure.

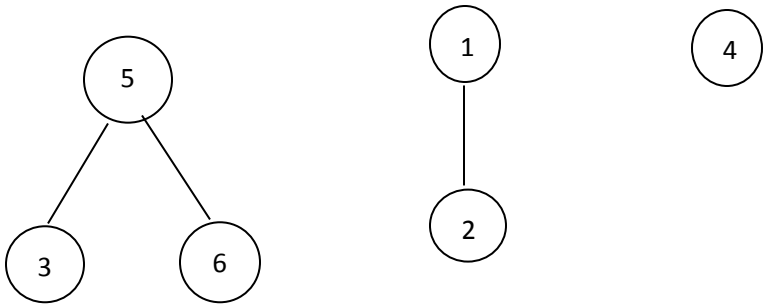
Union-Find data structure:
We have n-sets $\{1\}, \{2\}, \ldots, \{n\}$
We want to perform a sequence of Union-find operations

Union(A, B) → performs a destructive union of the two sets named A and B, i.e., after the union of A and B is computed, the sets A and B will not exist any more.

Find(x) → returns the name of the set x belongs to.

We can represent the sets as trees.

$A = \{3, 5, 6\}$; $B = \{1, 2\}$; $C = \{4\}$



For simplicity we can use the root as the name of the set represented by any tree. For example, *Find*(*2*) returns *1*.

Union(*5*,*1*): One root is made a child of the other root. Specifically, the root of the tree with the smaller number of elements is made a child of the other root. In this case we can shown that Union takes O(1) time and Find takes O(log *n*) time where *n* is the number of elements in the tree.