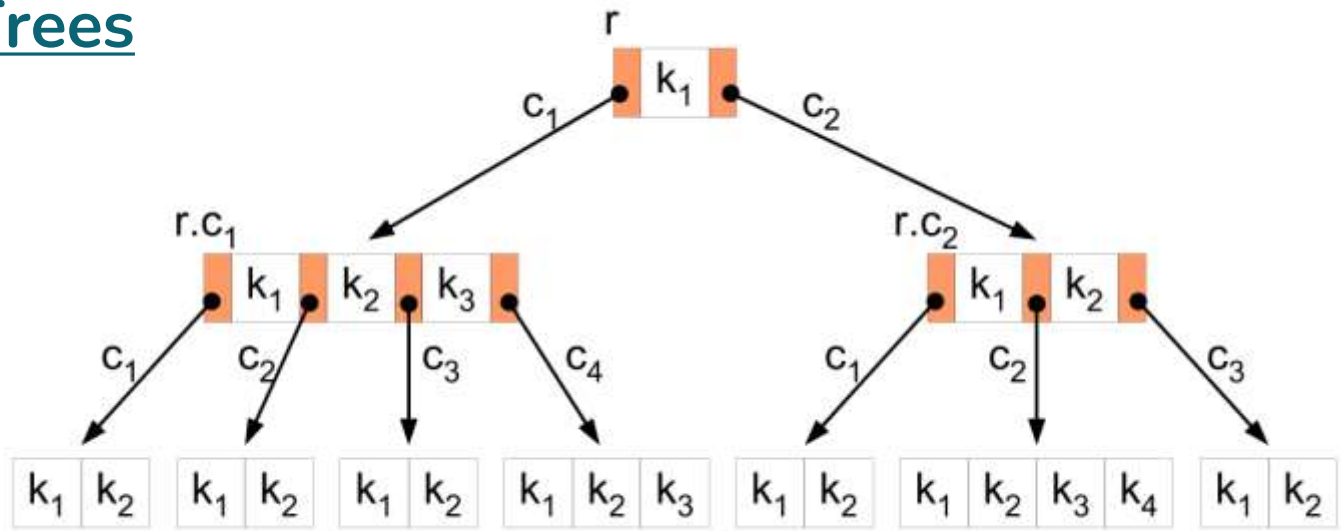
Recap

- Binomial Heaps
- Fibonacci Heaps

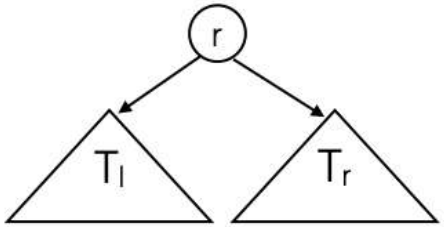
Onto…

B-Trees


r
k1
c1
c2
r.c1
k1
k2
k3
r.c2
k1
k2
c1
c2
c3
c4
c1
c2
c3
k1
k2
k1
k2
k1
k2
k1
k2
k3
k1
k2
k1
k2
k3
k4
k1
k2

Review: Binary Search Trees

- BST property:
  - For all nodes s in T_l, $s_{item} < r_{item}$.
  - *F*or all nodes t in T_l, $t_{item} > r_{item}$.



- To keep BST operations (search/insert/delete/findMin/ findMax) efficient, we need to maintain a **balanced tree:**
  - height of the tree should be close to log(N).
  - Example: AVL balancing condition, height difference between left and right subtree is at most 1.

M-way trees

Columbia University Lectures | Slides: 3-30

B-Tree: Intro, Insert, Delete

IIT-D Lectures | Slides: 2-22

B-Tree: Deletion

Case 1: *The search arrives at a leaf node* x. If x contains key k, then delete k from x. If x does not contain key k, then k was not in the B-tree and nothing else needs to be done.

Case 2: *The search arrives at an internal node* x *that contains key* k. Let $k = x.key_i$. One of the following three cases applies, depending on the number of keys in $x.c_i$ (the child of x that precedes k) and $x.c_{i+1}$ (the child of x that follows k).

Case 2a: $x.c_i$ *has at least* t *keys*. Find the predecessor k' of k in the subtree rooted at $x.c_i$. Recursively delete k' from $x.c_i$, and replace k by k' in x. (Key k' can be found and deleted in a single downward pass.)

Case 2b: $x.c_i$ *has* $t - 1$ *keys and* $x.c_{i+1}$ *has at least* t *keys*. This case is symmetric to case 2a. Find the successor k' of k in the subtree rooted at $x.c_{i+1}$.

B-Tree: Deletion

Recursively delete k' from $x.c_{i+1}$, and replace k by k' in x. (Again, finding and deleting k' can be done in a single downward pass.)

Case 2c: *Both $x.c_i$ and $x.c_{i+1}$ have $t-1$ keys.* Merge k and all of $x.c_{i+1}$ into $x.c_i$, so that x loses both k and the pointer to $x.c_{i+1}$, and $x.c_i$ now contains $2t-1$ keys. Then free $x.c_{i+1}$ and recursively delete k from $x.c_i$.

Case 3: *The search arrives at an internal node x that does not contain key k.* Continue searching down the tree while ensuring that each node visited has at least t keys. To do so, determine the root $x.c_i$ of the appropriate subtree that must contain k, if k is in the tree at all. If $x.c_i$ has only $t-1$ keys, execute case 3a or 3b as necessary to guarantee descending to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

Case 3a: *$x.c_i$ has only $t-1$ keys but has an immediate sibling with at least t keys.* Give $x.c_i$ an extra key by moving a key from x down into $x.c_i$, moving a key from $x.c_i$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $x.c_i$.

Case 3b: *$x.c_i$ and each of $x.c_i$'s immediate siblings have $t-1$ keys.* (It is possible for $x.c_i$ to have either one or two siblings.) Merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

B-Tree: Deletion

In cases 2c and 3b, if node x is the root, it could end up having no keys. When this situation occurs, then x is deleted, and x's only child $x.c_1$ becomes the new root of the tree. This action decreases the height of the tree by one and preserves the property that the root of the tree contains at least one key (unless the tree is empty).

B-Tree: Deletion

Since most of the keys in a B-tree are in the leaves, deletion operations often end up deleting keys from leaves. The B-TREE-DELETE procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node x, however, the procedure might make a downward pass through the tree to find the key's predecessor or successor and then return to node x to replace the key with its predecessor or successor (cases 2a and 2b). Returning to node x does not require a traversal through all the levels between x and the node containing the predecessor or successor, however, since the procedure can just keep a pointer to x and the key position within x and put the predecessor or successor key directly there.

B-Tree: Deletion

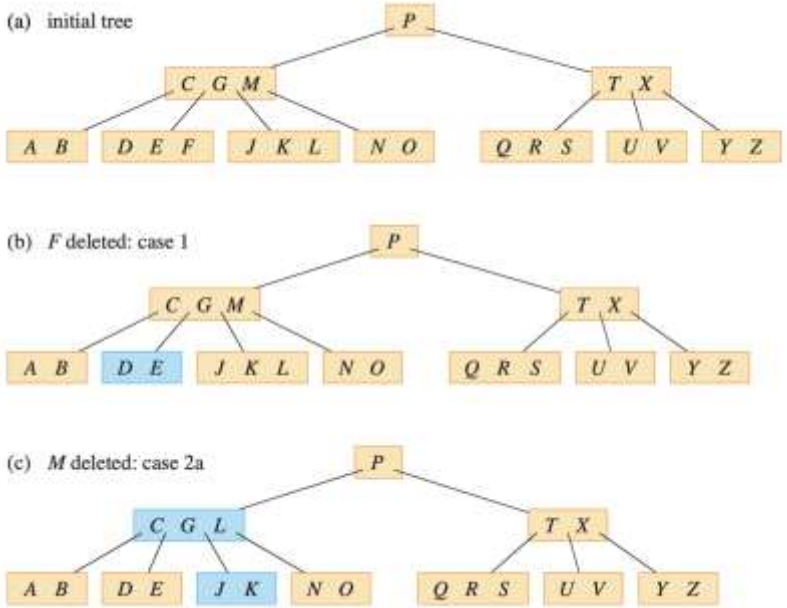


Figure 18.8 Deleting keys from a B-tree. The minimum degree for this B-tree is $t = 3$, so that, other than the root, every node must have at least 2 keys. Blue nodes are those that are modified by the deletion process. **(a)** The B-tree of Figure 18.7(e). **(b)** Deletion of F, which is case 1: simple deletion from a leaf when all nodes visited during the search (other than the root) have at least $t = 3$ keys. **(c)** Deletion of M, which is case 2a: the predecessor L of M moves up to take M's position.

B-Tree: Deletion

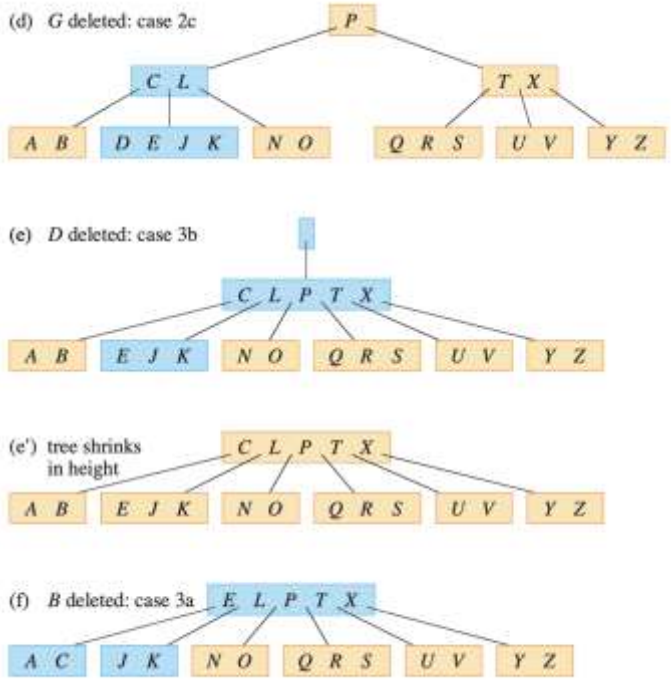


Figure 18.8, continued **(d)** Deletion of G, which is case 2c: push G down to make node $DEGJK$ and then delete G from this leaf (case 1). **(e)** Deletion of D, which is case 3b: since the recursion cannot descend to node CL because it has only 2 keys, push P down and merge it with CL and TX to form $CLPTX$. Then delete D from a leaf (case 1). **(e′)** After (e), delete the empty root. The tree shrinks in height by 1. **(f)** Deletion of B, which is case 3a: C moves to fill B's position and E moves to fill C's position.

B-Tree: Intro, Insert, Delete

UoW Lectures | Another Example

B+Tree: <Bonus>

Intro, Operations

Recap

- M-way/M-ary tree
- B-Tree
  - Introduction
  - Insertion
  - Deletion
- B-Tree Examples
- B^+ Tree