BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Data Structures and Algorithms
CS F211

Vishal Gupta
Department of Computer Science and Information Systems
Birla Institute of Technology and Science
Pilani Campus, Pilani

Agenda: B Trees

B Tree: Motivation

Motivation for B-Trees

- Index structures for large datasets cannot be stored in main memory
- Storing it on disk requires different approach to efficiency

- Assuming that a disk spins at 3600 RPM, one revolution occurs in 1/60 of a second, or 16.7ms
- Crudely speaking, one disk access takes about the same time as 200,000 instructions

Motivation (cont.)

- Assume that we use an AVL tree to store about 20 million records
- We end up with a **very** deep binary tree with lots of different disk accesses; $\log_2 20{,}000{,}000$ is about 24, so this takes about 0.2 seconds
- We know we can't improve on the log *n* lower bound on search for a binary tree
- But, the solution is to use more branches and thus reduce the height of the tree!
  – As branching increases, depth decreases

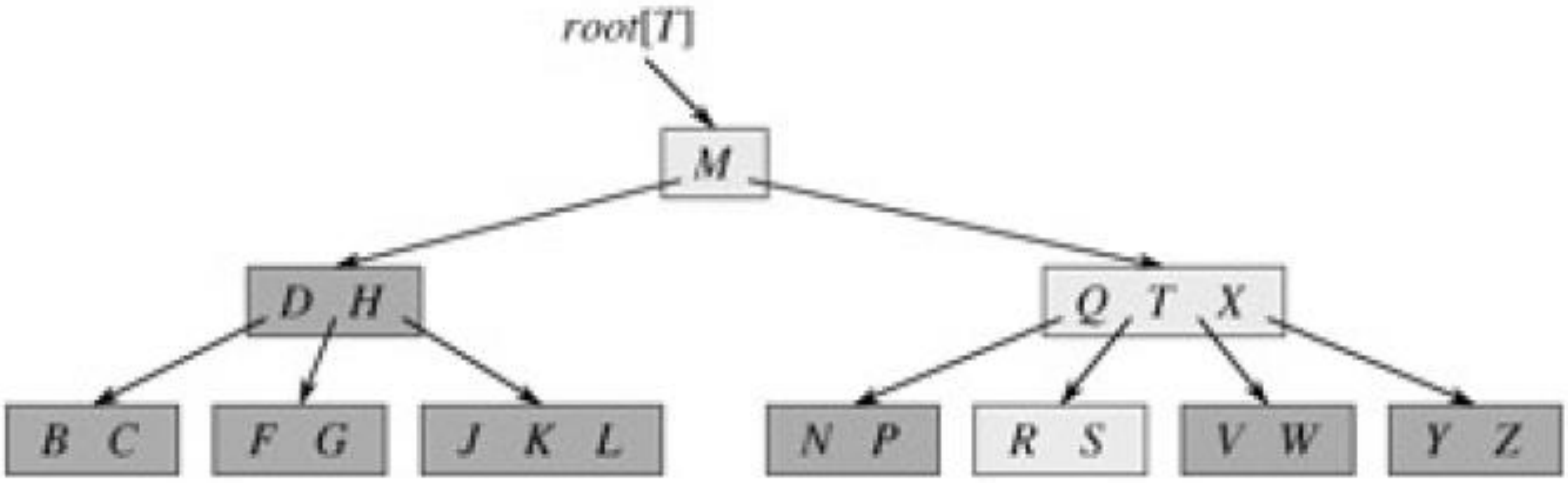
B-Trees

B-Trees are useful in the following cases:

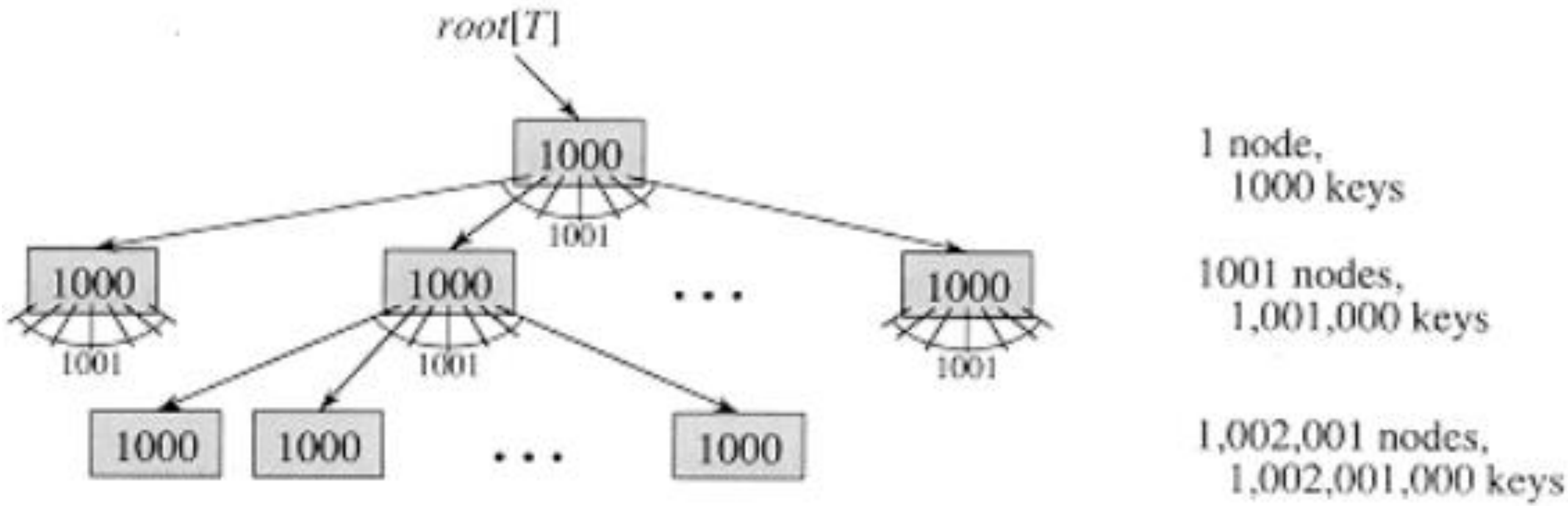
- The number of objects is too large to fit in memory.
- Need external storage.
- Disk accesses are slow, thus need to minimize the number of disk accesses

Is RB Tree good in these situations?

B Tree Example

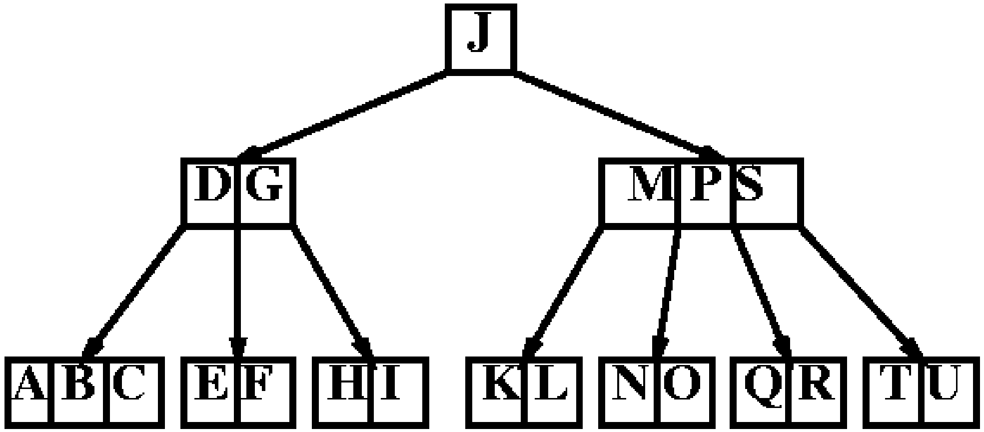


A B-tree of height 2 containing over one billion keys.



B-Trees

- B-Trees are balanced, like RB trees.
- They have a large number of children (large branching factor), unlike RB trees.
- The branching factor is determined by the size of disk transfers (page size).
- Each object (node) referenced requires a DiskRead.
- Each object modified requires a DiskWrite.
- The root of the tree is kept in memory at all times.
- Insert, Delete, Search = O(h), where h is the height of the tree.
  O(lgn), though much less in reality ($log_{BF}\, n$).



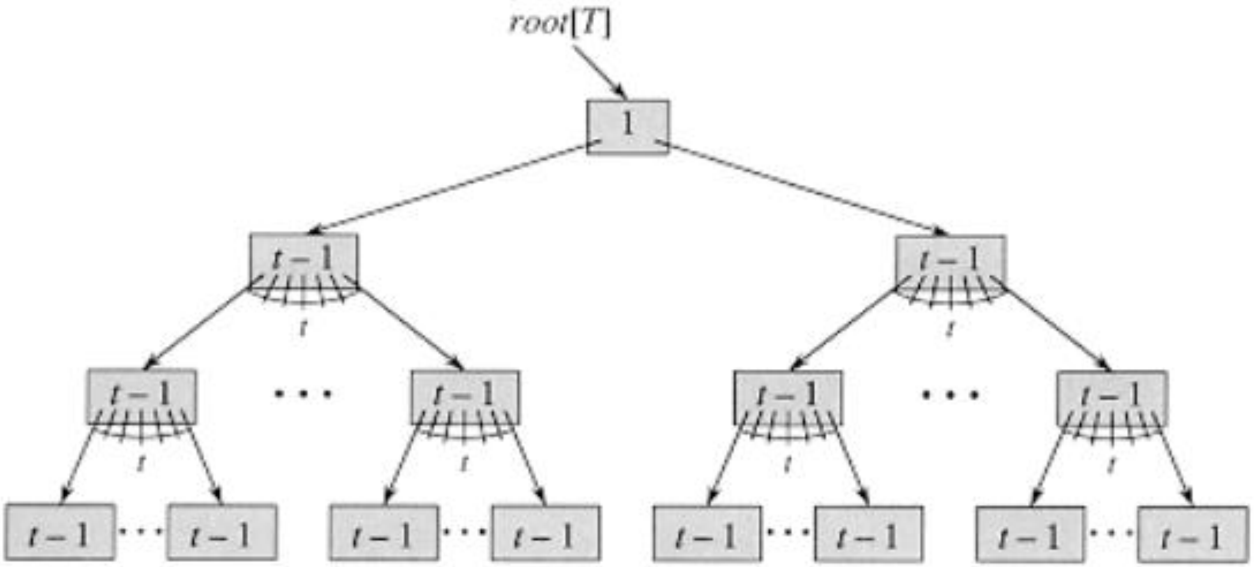
Properties of B-Trees

- Every leaf has the same depth equal to the height of the tree.
- The number of keys is bounded in terms of the minimum degree $t \geq 2$. (WHY?)
- $n(x) \geq t-1$ (except root ≥ 1)
- #children(x) $\geq t$ (except root ≥ 0), leaves = 0
- $n(x) \leq 2t - 1$
- #children $\leq 2t$ (except leaves which = 0)
- If $n(x) = 2t - 1$ then n is a full node.

What is the height of B-Tree

- Given a B-Tree of height h, minimum degree t, and number of keys n, prove that the height of the B-tree is:

$$h \leq \log_t \frac{n+1}{2}$$



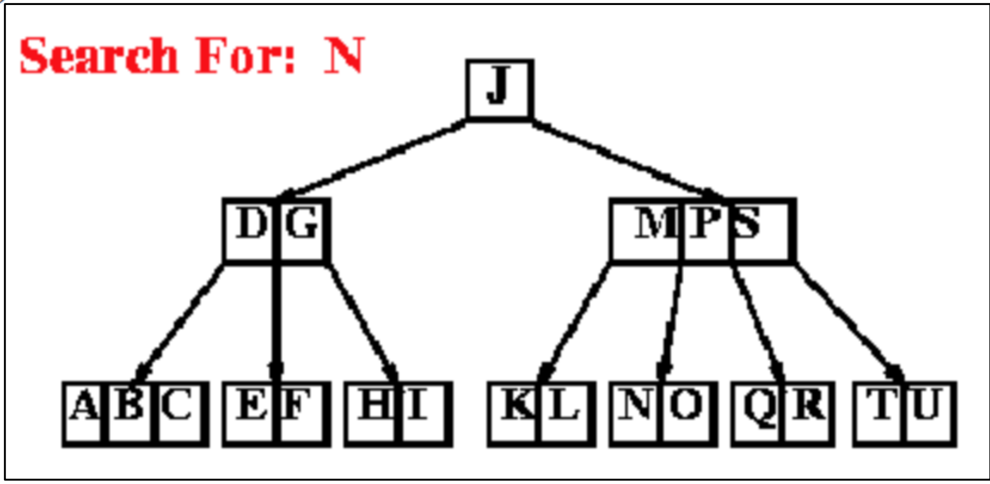
What is the height of B-Tree

- Given a B-Tree of height h, minimum degree t, and number of keys n, prove that the height of the B-tree is:

$$h \le \log_t \frac{n+1}{2}$$

Searching a key

```
Search(x, k)
    if k in node x
    then return x and i such that key_i(x) = k
    else if x is a leaf
        then return NIL
        else find i such that key_{i-1}(x) < k < key_i(x)
            DiskRead(child_i(x))
            return Search(child_i(x), k)
```



What is the Time-Complexity ??

Inserting a Node

Overview

- If node x is a non-full (< 2t-1 keys) leaf, then insert new key k in node x
- If node x is non-full but not a leaf, then recurse to appropriate child of x
- If node x is full (2t-1 keys), then ``split'' the node into x1 and x2, and recurse to appropriate node x1 or x2.

Splitting a Node

B-Tree-Split-Child(x, i, y) ; x is parent, y is child in ith subtree

Allocate(z) ; n(z)=t-1, leaf(z) = leaf(y)

Copy y's second half keys and children to z

n(y) = t-1

Shift x's keys and children one to the right from i

$child_{i+1}(x) = z$

$key_i(x) = key_t(y)$

n(x) = n(x) + 1

Write(x)

DiskWrite(y)

DiskWrite(z)

Inserting a Node

Insert: B-Tree-Insert(T, k)

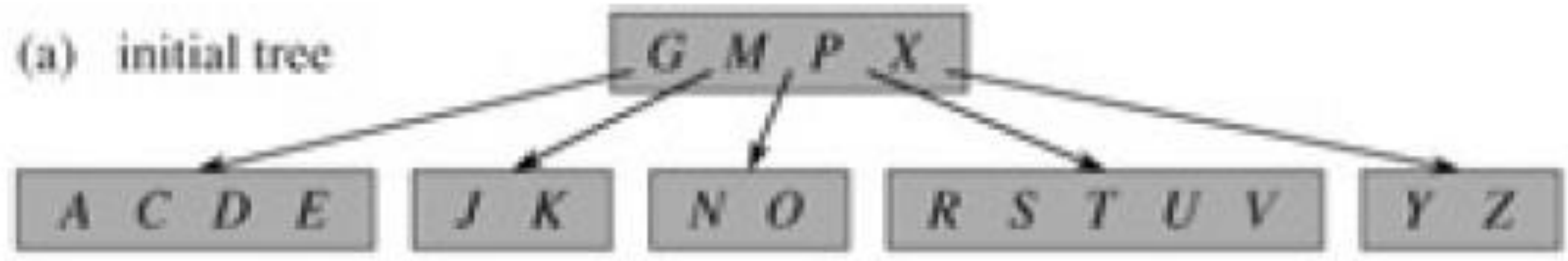
- Start at root(T) moving down the tree looking for the proper leaf to put k
- Split all full nodes along the way

Inserting a Node

```
B-Tree-Insert(T, k)
    r = root(T)
    if n(r) = 2t-1                        ; full
    then allocate empty node s pointing to r
        B-Tree-Split-Child(s, 1, r)
        B-Tree-Insert-Nonfull(s, k)
    else B-Tree-Insert-Nonfull(r, k)
```

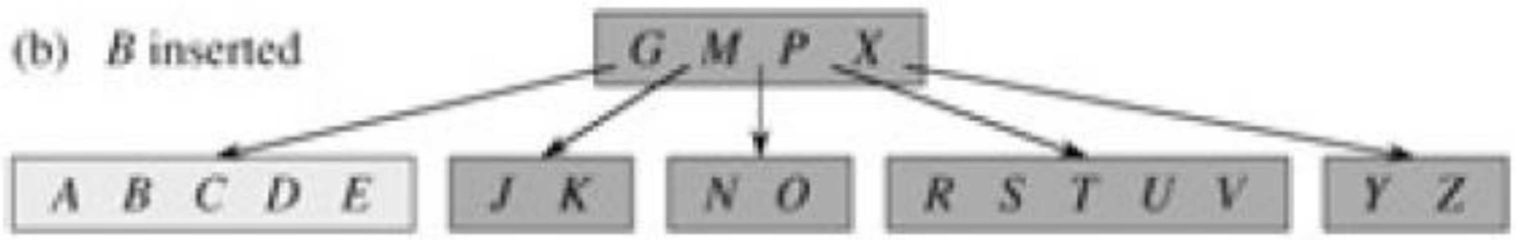
```
B-Tree-Insert-Nonfull(x, k)
    if leaf(x)
    then shift keys of x higher than k one to the right
        put k in appropriate spot
        n(x) = n(x) + 1
        DiskWrite(x)
    else find smallest i such that k < key_i(x)
        DiskRead(child_i(x))
        if n(child_i(x)) = 2t - 1 ; full
        then B-Tree-Split-Child(x, i, child_i(x))
```

Insertion Example



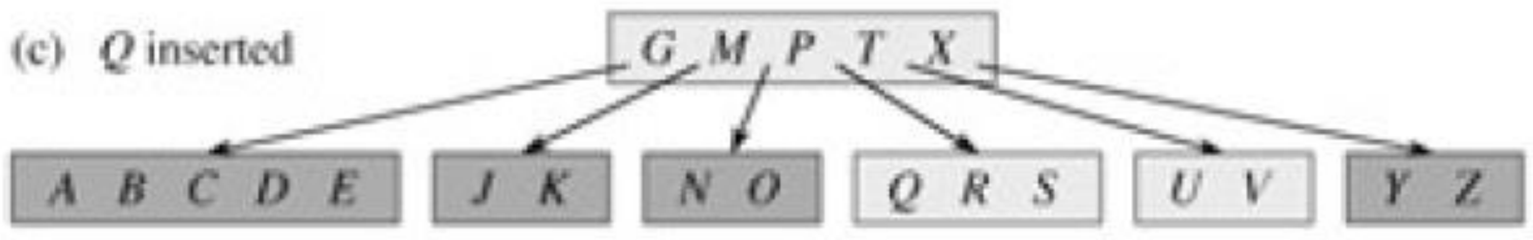
Insert B:

Insertion Example



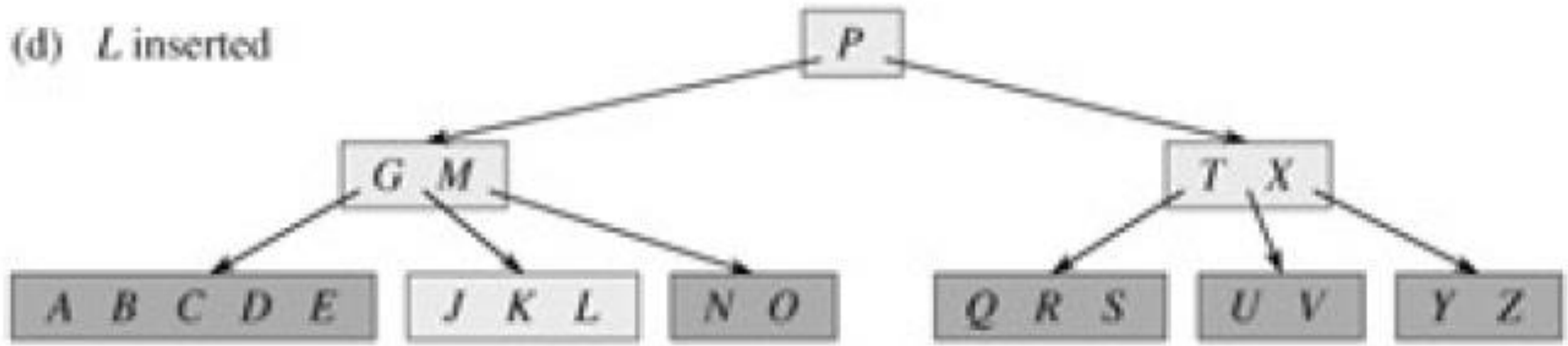
Insert Q:

Insertion Example



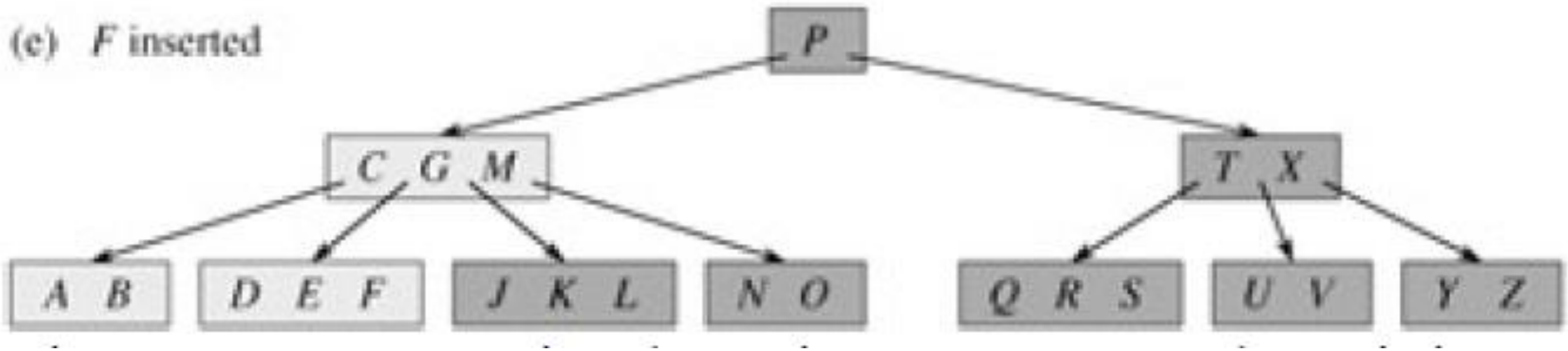
Insert L:

Insertion Example



Insert F:

Insertion Example



Insert F:

Deleting a Node

Overview

Deletion: B-Tree-Delete(x, k)

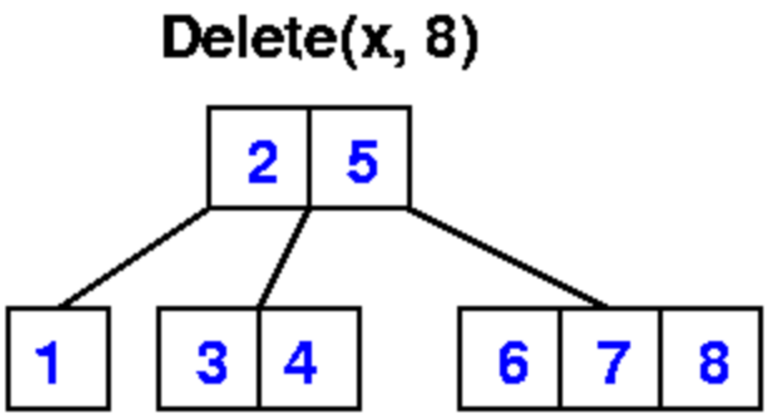
1) Search down the tree for node containing k
2) When B-Tree-Delete is called recursively, the number of keys in x must be at least the minimum degree t (the root can have < t keys)
1) If x is a leaf, just remove key k and still have at least t-1 keys in x
2) If there are not ≥ t keys in x, then borrow keys from other nodes.

Deleting a Node

Deletion

There are three general cases:

[Case 1:] If key k in node x and x is a leaf, then remove k from x.



[Case 2:] If k is in x and x is an internal node.

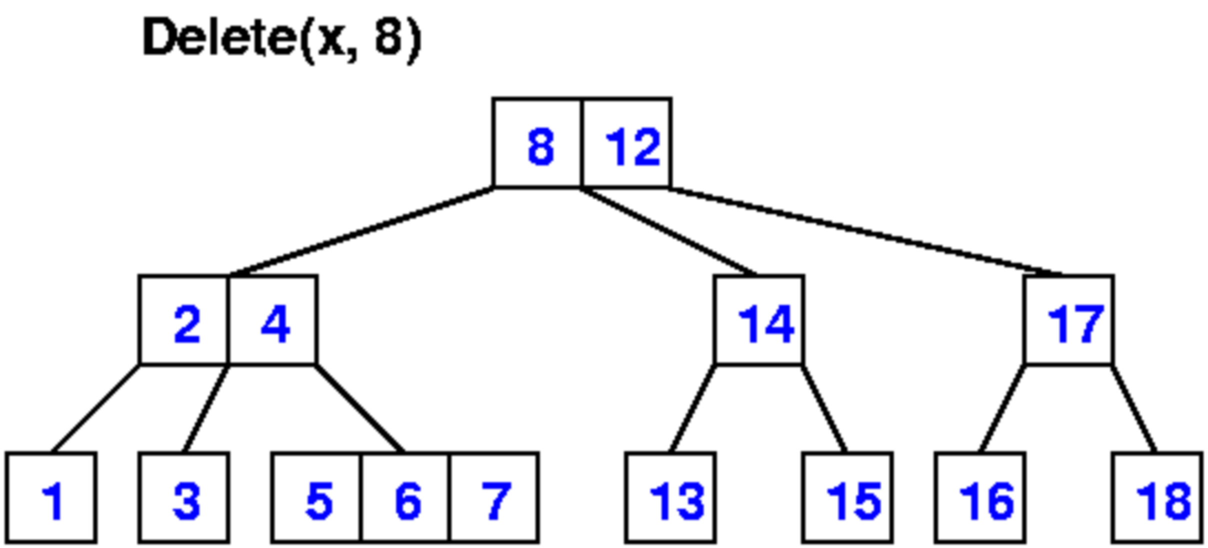
One of three subcases:

Deleting a Node

Case 2a

If child y preceding k in x has $\geq$ t keys:

- Find predecessor k' of k in subtree rooted at y
- Recursively delete k' (first two steps can be performed in one pass down the tree)
- Replace k by k' in x

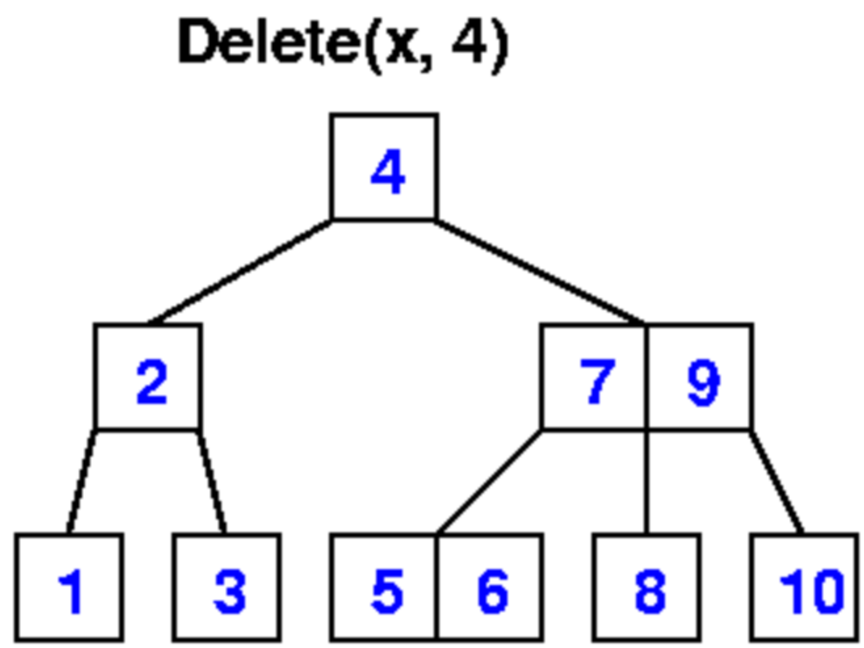


Deleting a Node

Case 2b

If child z following k in x has $\geq$ t keys:

- Find successor k' of k in subtree y
- Recursively delete k'
- Replace k by k' in x



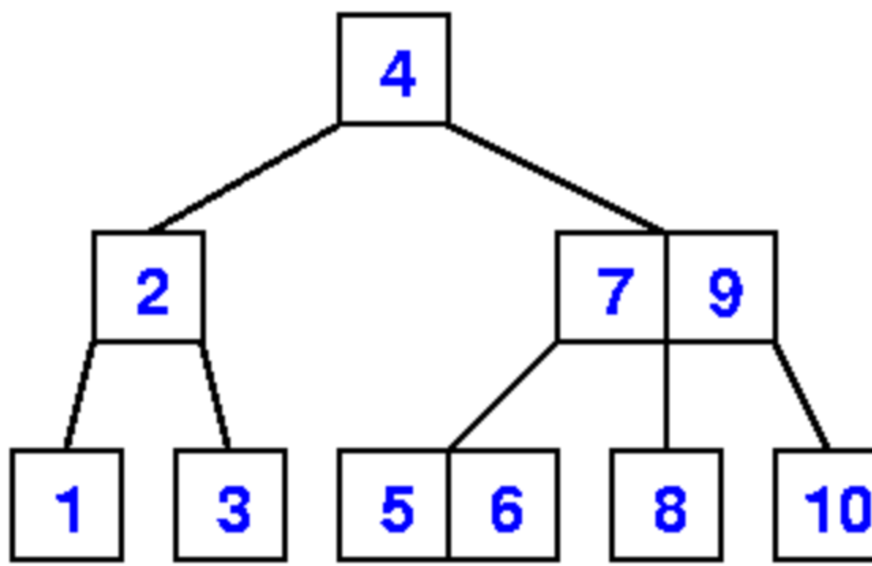
Deleting a Node

Case 2c

If both y and z have t-1 keys:

- Merge k and all of z into y
- Free z
- Recursively delete k from y

Delete(x, 9)



Deleting a Node

innovate achieve lead

Case 3

if k not in internal node x
then determine subtree $child_i(x)$ containing k
    if $child_i(x)$ has $\geq$ t keys

    then B-Tree-Delete($child_i(x)$, k)
    else execute Case 3a or 3b until can descend to node having $\geq$ t keys

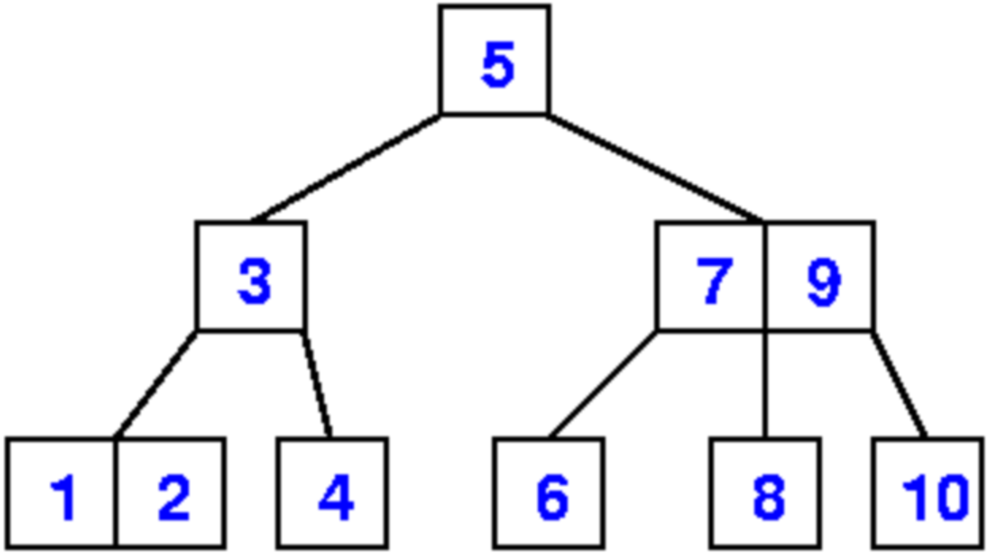
Deleting a Node

Case 3a

If $child_i(x)$ has t-1 keys but has a left or right sibling with $\geq$ t keys, then borrow one from sibling

- move key from x to $child_i(x)$
- move key from sibling to x
- move child from sibling to $child_i(x)$

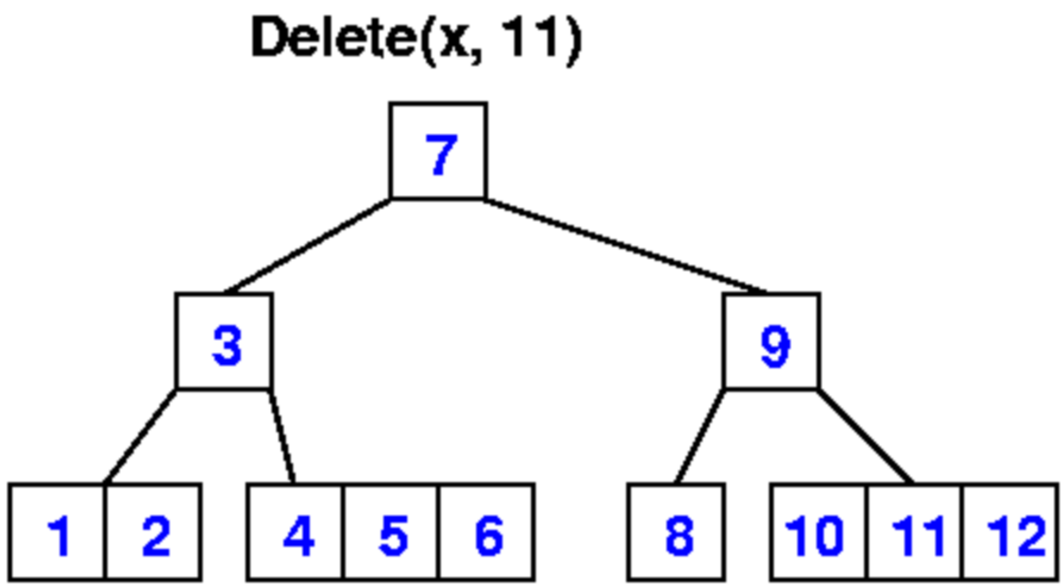
Delete(x, 1)



Deleting a Node

Case 3b

If $child_i(x)$ and its left and right siblings have t-1 keys then merge $child_i(x)$ with one sibling using median key from x.



Deleting a Node

Deletion Example

1. If the key k is in node x and x is a leaf, delete the key k from x.

2. If the key k is in node x and x is an internal node, do the following.

a)If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x. (Finding k' and deleting it can be performed in a single downward pass.)

b)Symmetrically, if the child z that follows k in node x has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x. (Finding k' and deleting it can be performed in a single downward pass.)

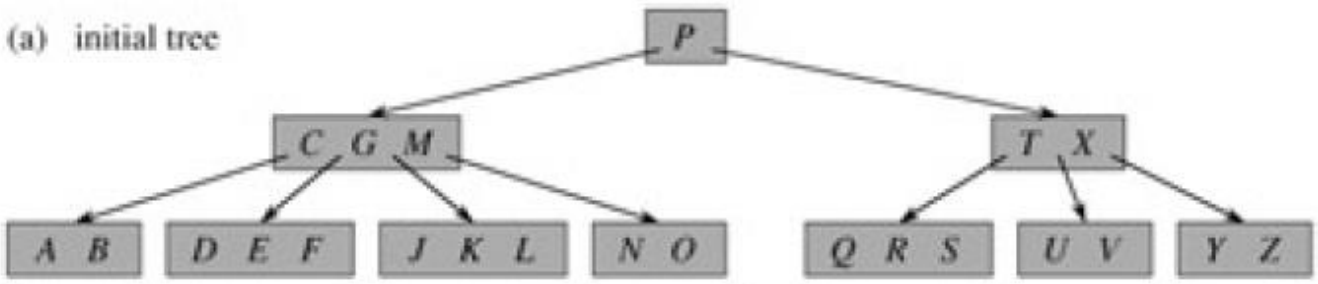
a)Otherwise, if both y and z have only t- 1 keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains $2t$ - 1 keys. Then, free z and recursively delete k from y.

3. If the key k is not present in internal node x, determine the root $c_i[x]$ of the appropriate subtree that must contain k, if k is in the tree at all. If $c_i[x]$ has only t - 1 keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then, finish by recursing on the appropriate child of x.

a) If $c_i[x]$ has only t - 1 keys but has a sibling with t keys, give $c_i[x]$ an extra key by moving a key from x down into $c_i[x]$, moving a key from $c_i[x]$'s immediate left or right sibling up into x, and moving the appropriate child from the sibling into $c_i[x]$.

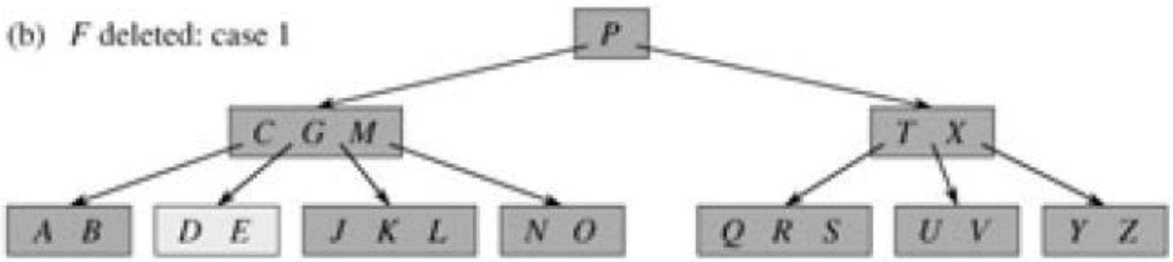
a) If $c_i[x]$ and all of $c_i[x]$'s siblings have t - 1 keys, merge c_i with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

Deletion Example



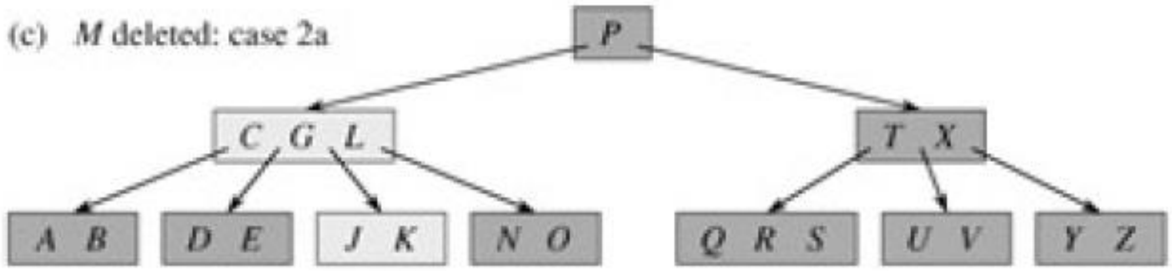
Delete F

Deletion Example



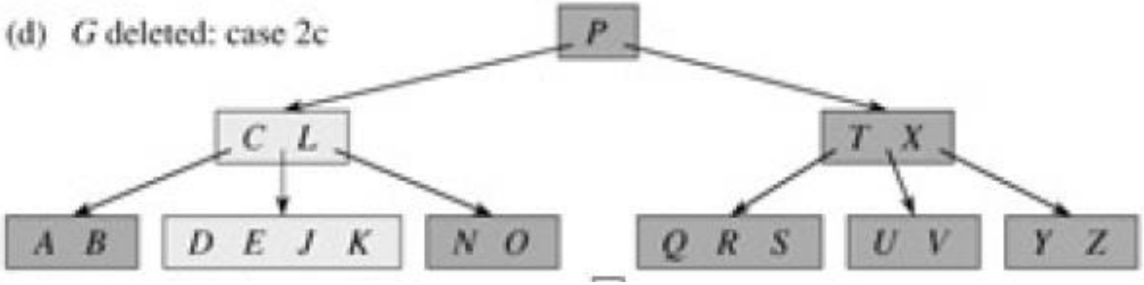
Delete M

Deletion Example



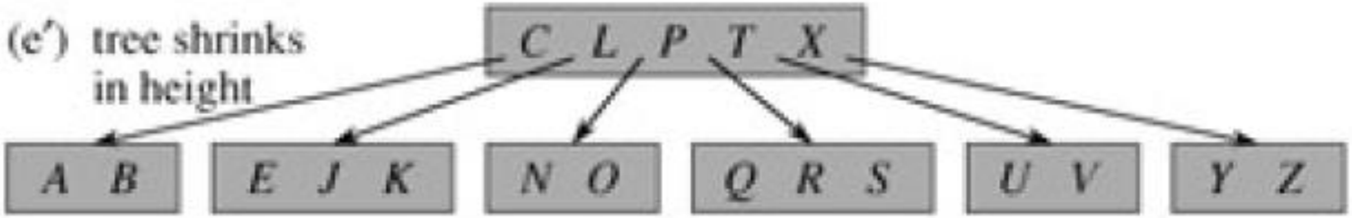
Delete G

Deletion Example



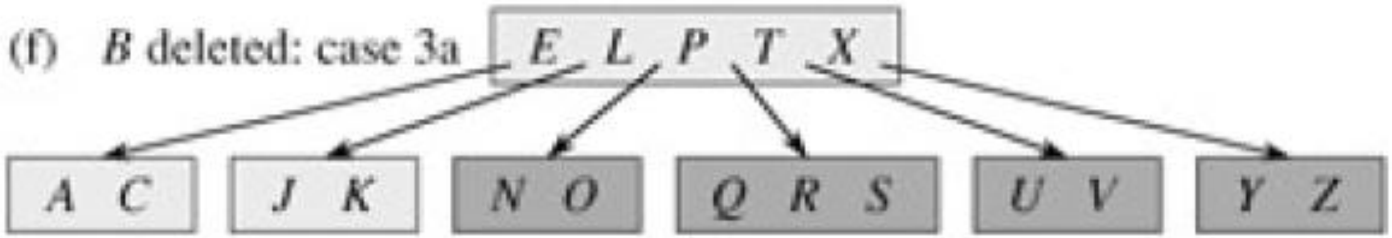
Delete D

Deletion Example



Delete B

Deletion Example



Reasons for using B-Trees

- When searching tables held on disc, the cost of each disc transfer is high but doesn't depend much on the amount of data transferred, especially if consecutive items are transferred
  - If we use a B-tree of order 101, say, we can transfer each node in one disc read operation
  - A B-tree of order 101 and height 3 can hold $101^4 – 1$ items (approximately 100 million) and any item can be accessed with 3 disc reads (assuming we hold the root in memory)
- If we take *m* = 3, we get a **2-3 tree**, in which non-leaf nodes have two or three children (i.e., one or two keys)
  - B-Trees are always balanced (since the leaves are all at the same level), so 2-3 trees make a good type of balanced tree

Comparing Trees

- Binary trees
  - Can become *unbalanced* and *lose* their good time complexity (big O)
  - AVL trees are strict binary trees that *overcome the balance problem*
  - Heaps remain balanced but only *prioritise* (not order) the keys

- Multi-way trees
  - B-Trees can be *m*-way, they can have any (odd) number of children
  - One B-Tree, the 2-3 (or 3-way) B-Tree, *approximates* a permanently balanced binary tree, exchanging the AVL tree's balancing operations for insertion and (more complex) deletion operations

Thank You