B Trees and B+ Trees

Storing large amounts of Data

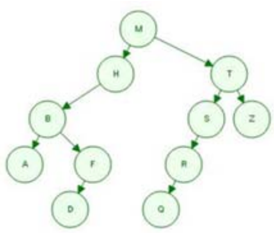
- B Trees, B+ Trees: data structures and algorithms for
    - large data
    - stored on disk (ie, slow access)
- File systems:
    - Lots of files, each file stored as lots of blocks.
- Databases:
    - large tables of data
    - each indexed by a key (or perhaps multiple alternative keys)

Problem:

- How do we access the data efficiently:
    - individual items (given key)
    - sequence of all items (preferably in order)
    - assume data is stored in files on hard drives (slow access time)
- Use some kind of index structure
    - assume the index is also stored in a file

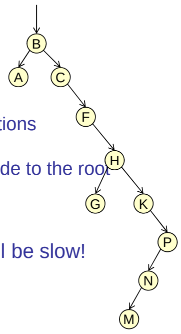
COMP103 approaches:

- Efficient Set and Map implementations:
    - Hash Tables.
    - Binary Search Tree
- http://www.cs.usfca.edu/~galles/visualization
- Binary search tree
    - Add M H T S R Q B A F D Z



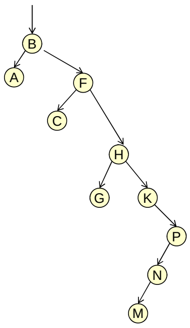
- Search: Log(n)

Problems with Binary Search



- Unbalanced trees are inefficient
  - ⇒ must keep the tree balanced
    - AVL trees: self-balanced by tree rotations
    - red-black trees: node with a color bit
    - splay trees: move recently access node to the root
    - B trees
- lots of pointer following
  - ⇒ if each node is stored in a file, this will be slow!

Balanced trees

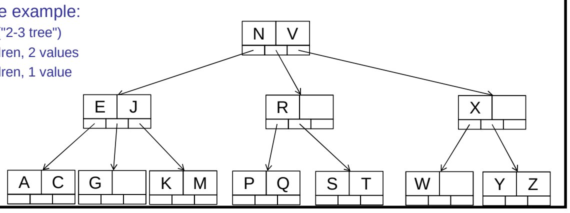


- Balancing:
  - (a) "rotate" nodes in tree if unbalanced.
    - AVL trees
    - red-black trees
    - splay trees
    - (B trees)
  - (b) always add levels at the top!
    - B trees (and B+ trees,…)
- Too many Pointers
  - More data in each node
  - More children of each node
    - ⇒ "bushier" trees, fewer steps to leaves

B Trees

- Like binary search trees, but
  - non-leaf nodes have up to *m* children, and *m*-1 data values
  - non-leaf, non-root nodes always have at least $\lceil m/2 \rceil$ children and $\lceil m/2 \rceil$-1 data values (ie, at least half full)
  - leaf nodes contain up to m-1 data values and no children
  - adding is done "at the top" rather than "at the bottom"

B tree example:
m=3, ("2-3 tree")
3 children, 2 values
2 children, 1 value



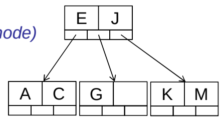
2-3 trees

- Every internal node is a 3-node or a 2-node
  - 3-node: 3 children, 2 values
  - 2-node: 2 children, 1 value
- All leaves are at the same level
  - Leaf node has 1 or 2 values
- All data is kept in sorted order

B Trees: Search

- Data values might be
  - single items (set of values)
  - key:value pairs (map)
- Search(key, node):
  - Just like binary search, but more comparisons at each node:

```
if node is null
    return fail
for i = 0 to k-1    (k is number of keys in node)
    if key < keys[i]
        return search(key, children[i])
    if key == keys[i]
        return value[i]
return search(key, children[k])
```



B Trees: Insert

To insert item (eg key:value pair):

Search for leaf where the item should be.

If leaf is not full, add item to the leaf.

If leaf is full:

Identify the middle item (existing item or the new item)

Create a new leaf node:

retain items before middle key in original node

put items after middle key in new leaf node,

push item up to parent, along with pointer to new node

To add new item to parent:

if parent is not full:

add new child pointer just right of new item

add new item to parent, and

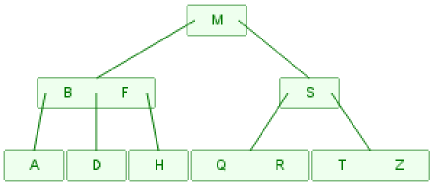
else: split parent node into two nodes (like leaf)

push middle item up to grandparent.

add pointer to new child just right of pushed up item

2-3 B Tree: Inserting values

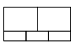
- Add M H T S R Q B A F D Z



- http://www.cs.usfca.edu/~galles/visualization/BTree.html

2-3 B Tree: Inserting values

- Add 8, 5, 1, 7, 3,12, 9, 6



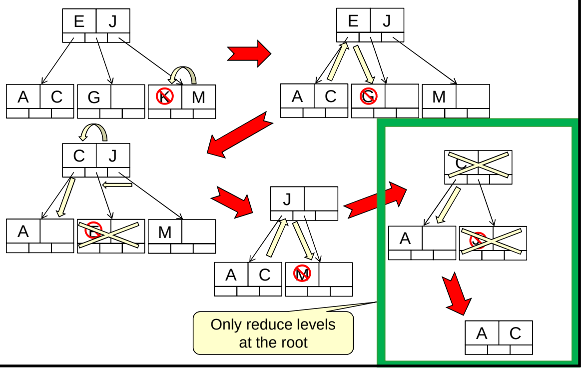
- http://www.cs.usfca.edu/~galles/visualization/BTree.html

2-3 BTree: Deletion

- Opposite of inserting:
  if a node becomes empty,
    if possible, rotate a value from a sibling through the parent
    to ensure minimum number of values per node.
      if two siblings, require >= 5 keys in parent and siblings
      if one sibling, require >=3 keys in parent and siblings
    - else merge nodes:
      - if two siblings, merge

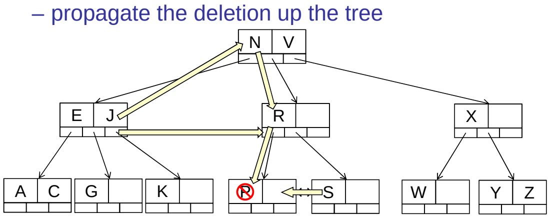
  - Easier at a leaf.
  - Harder if at an internal node.

Deleting from leaves



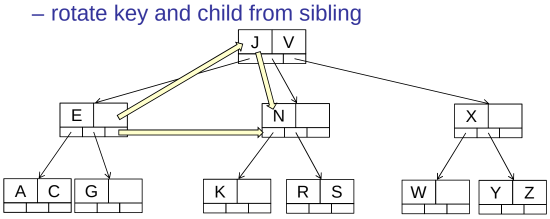
More deletions:

- When internal node becomes too empty
  - propagate the deletion up the tree



More deletions:

- Deletion from internal node ⇒
  - rotate key and child from sibling



Analysis

- B trees are balanced:
  - A new level is introduced only at the top
  - A level is removed only from the top

  Therefore:
  - all leaves are at the same level.
- Cost of search/add/delete:
  - $O(\log_{\lceil m/2 \rceil}(n))$ (at worst) = depth of tree with all half full nodes
  - $O(\log_m(n))$ (at best) = depth of tree with full nodes
  - if 100 million items in a B tree with m = 20,
    $\log_{10}(100{,}000{,}000)$ = ? 8
    $\log_{20}(100{,}000{,}000)$ = ? 6.14
  - if billion items in a B tree with m = 100,
    $\log_{50}(1{,}000{,}000{,}000)$ = ? 5.3
    $\log_{100}(1{,}000{,}000{,}000)$ = ? 4.5