Today’s Outline

- Admin:
  - Midterm #2 – Friday Nov 18th, topic list has been posted
  - HW #5 – Graphs, partners due Wed 23 at 11pm, due Thurs Dec 1 at 11pm

- Graphs
  - Minimum Spanning Trees

- Dictionaries
  - B-Trees

Trees so far

- BST
- AVL

$M$-ary Search Tree

- Maximum branching factor of $M$
- Complete tree has height $\approx$

# disk accesses for find:

Runtime of find:

Solution: B-Trees

- specialized $M$-ary search trees
- Each node has (up to) $M-1$ keys:
  - subtree between two keys $x$ and $y$ contains leaves with values $v$ such that $x \leq v < y$
- Pick branching factor $M$ such that each node takes one full page, block of memory
B-Trees

What makes them disk-friendly?

1. Many keys stored in a node
   • All brought to memory/cache in one access!

2. Internal nodes contain only keys;
   Only leaf nodes contain keys and actual data
   • The tree structure can be loaded into memory irrespective of data object size
   • Data actually resides in disk

B-Tree: Example

B-Tree with \( M = 4 \) (# pointers in internal node) and \( L = 4 \) (# data items in leaf)

Data objects, that I’ll ignore in slides

Note: All leaves at the same depth!

B-Tree Properties

– Data is stored at the leaves
– All leaves are at the same depth and contain between \( \lceil L/2 \rceil \) and \( L \) data items
– Internal nodes store up to \( M-1 \) keys
– Internal nodes have between \( \lceil M/2 \rceil \) and \( M \) children
– Root (special case) has between 2 and \( M \) children (or root could be a leaf)

B-trees vs. AVL trees

Suppose we have 100 million items (100,000,000):

• Depth of AVL Tree
• Depth of B+ Tree with \( M = 128, L = 64 \)
Insert(1)

And create a new root

Splitting the Root

Too many keys in a leaf!

Insert(59)

Overflowing leaves

So, split the leaf.

Insert(26)

Insert(5)

Propagating Splits

Add new child

Split the leaf, but no space in parent!

Split the leaf.

Create a new root

So, split the node.

Insertion Algorithm

1. Insert the key in its leaf
2. If the leaf ends up with L+1 items, overflow!
   - Split the leaf into two nodes:
     - original with \( \lceil (L+1)/2 \rceil \) items
     - new one with \( \lfloor (L+1)/2 \rfloor \) items
   - Add the new child to the parent
   - If the parent ends up with \( M+1 \) items, overflow!
3. If an internal node ends up with \( M+1 \) items, overflow!
   - Split the node into two nodes:
     - original with \( \lceil (M+1)/2 \rceil \) items
     - new one with \( \lfloor (M+1)/2 \rfloor \) items
   - Add the new child to the parent
   - If the parent ends up with \( M+1 \) items, overflow!
4. Split an overflowed root in two and hang the new nodes under a new root

This makes the tree deeper!

Insertion Algorithm

1. Insert the key in its leaf
2. If the leaf ends up with L+1 items, overflow!
   - Split the leaf into two nodes:
     - original with \( \lceil (L+1)/2 \rceil \) items
     - new one with \( \lfloor (L+1)/2 \rfloor \) items
   - Add the new child to the parent
   - If the parent ends up with \( M+1 \) items, overflow!
3. If an internal node ends up with \( M+1 \) items, overflow!
   - Split the node into two nodes:
     - original with \( \lceil (M+1)/2 \rceil \) items
     - new one with \( \lfloor (M+1)/2 \rfloor \) items
   - Add the new child to the parent
   - If the parent ends up with \( M+1 \) items, overflow!
4. Split an overflowed root in two and hang the new nodes under a new root

This makes the tree deeper!

Insertion Algorithm

1. Insert the key in its leaf
2. If the leaf ends up with L+1 items, overflow!
   - Split the leaf into two nodes:
     - original with \( \lceil (L+1)/2 \rceil \) items
     - new one with \( \lfloor (L+1)/2 \rfloor \) items
   - Add the new child to the parent
   - If the parent ends up with \( M+1 \) items, overflow!
3. If an internal node ends up with \( M+1 \) items, overflow!
   - Split the node into two nodes:
     - original with \( \lceil (M+1)/2 \rceil \) items
     - new one with \( \lfloor (M+1)/2 \rfloor \) items
   - Add the new child to the parent
   - If the parent ends up with \( M+1 \) items, overflow!
4. Split an overflowed root in two and hang the new nodes under a new root

This makes the tree deeper!

Insertion Algorithm

1. Insert the key in its leaf
2. If the leaf ends up with L+1 items, overflow!
   - Split the leaf into two nodes:
     - original with \( \lceil (L+1)/2 \rceil \) items
     - new one with \( \lfloor (L+1)/2 \rfloor \) items
   - Add the new child to the parent
   - If the parent ends up with \( M+1 \) items, overflow!
3. If an internal node ends up with \( M+1 \) items, overflow!
   - Split the node into two nodes:
     - original with \( \lceil (M+1)/2 \rceil \) items
     - new one with \( \lfloor (M+1)/2 \rfloor \) items
   - Add the new child to the parent
   - If the parent ends up with \( M+1 \) items, overflow!
4. Split an overflowed root in two and hang the new nodes under a new root

This makes the tree deeper!

Deletion

1. Delete item from leaf
2. Update keys of ancestors if necessary
3. If an internal node ends up with M-1 items, underflow!
   - Split the node into two nodes:
     - original with \( \lceil (M-1)/2 \rceil \) items
     - new one with \( \lfloor (M-1)/2 \rfloor \) items
   - Add the new child to the parent
   - If the parent ends up with \( M-1 \) items, underflow!
4. Split an underflowed root in two and hang the new nodes under a new root

What could go wrong?
Deletion and Adoption

A leaf has too few keys!

Delete(5)

So, borrow from a sibling

Delete(1)

Does Adoption Always Work?

- What if the sibling doesn’t have enough for you to borrow from?
- e.g. you have \([L/2]-1\) and sibling has \([L/2]\)?

Deletion and Merging

A leaf has too few keys!

Delete(3)

And no sibling with surplus!

But now an internal node has too few subtrees!

Deletion with Propagation

(More Adoption)

Delete(26)

Pulling out the Root

A leaf has too few keys!
And no sibling with surplus!

So, delete the leaf; merge

But now the root has just one subtree!

A node has too few subtrees and no neighbor with surplus!

So, delete the node
### Pulling out the Root (continued)

The root has just one subtree!

Simply make the one child the new root!

### Deletion Algorithm

1. Remove the key from its leaf

2. If the leaf ends up with fewer than \( \lceil L/2 \rceil \) items, **underflow**!
   - Adopt data from a sibling; update the parent
   - If adopting won’t work, delete node and merge with neighbor
   - If the parent ends up with fewer than \( \lceil M/2 \rceil \) items, **underflow**!

### Deletion Slide Two

3. If an *internal* node ends up with fewer than \( \lceil M/2 \rceil \) items, **underflow**!
   - Adopt from a neighbor; update the parent
   - If adoption won’t work, merge with neighbor
   - If the parent ends up with fewer than \( \lceil M/2 \rceil \) items, **underflow**!

4. If the root ends up with only one child, make the child the new root of the tree

   ![This reduces the height of the tree!]

### Thinking about B-Trees

- B-Tree insertion can cause (expensive) splitting and propagation
- B-Tree deletion can cause (cheap) adoption or (expensive) deletion, merging and propagation
- Propagation is rare if \( M \) and \( L \) are large
  - Why?
- If \( M = L = 128 \), then a B-Tree of height 4 will store at least 30,000,000 items

### Tree Names You Might Encounter

FYI:
- B-Trees with \( M = 3, L = x \) are called **2-3 trees**
  - Nodes can have 2 or 3 pointers
- B-Trees with \( M = 4, L = x \) are called **2-3-4 trees**
  - Nodes can have 2, 3, or 4 pointers

### Determining M and L for a B-Tree

1. Page on disk = 1 KByte
2. Key = 8 bytes, Pointer = 4 bytes
3. Data = 256 bytes per record (includes key)

\[ M = \]
\[ L = \]