CSE 100: RED-BLACK TREES (CONTD.)
B TREES
Which of the following are legal red-black trees?

A. A&B
B. A&B&C

Red Black Trees, review
Is this a legal Red-Black tree?

A. Yes
B. No
Insertions: Summary, so far

Case 0: The parent of the node you are inserting is black. Insert and you’re done

Case 1: the parent of the node is red, the sibling of the parent is black:

  Case 1a: P is left child of G, X is left child of P (single rotate then recolor)
  Case 1b: P is left child of G, X is right child of P (double rotate then recolor)

  Case 1c: P is right child of G, X is right child of P
  Case 1d: P is right child of G, X is left child of P

What if the sibling of the parent is red??
Insertions: Parent’s sibling is red

Insert 35?
Insertions: Parent’s sibling is red

30

15

10
5

20

60

50

40

55

35

70

80

85

90
Insertions: Parent’s sibling is red

In practice: fix the tree as you descend so you don’t run into this problem
Insertions: Parent’s sibling is red

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all it’s children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes

Insert 35?
Insertions: Parent’s sibling is red

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all its children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes
Insertions: Parent’s sibling is red

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all its children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes
Insertions: Parent’s sibling is red

Both children are red
So recolor

Insert 35?

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all its children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes
Insertions: Parent’s sibling is red

Now what?

Insert 35?

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all its children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes
Insertions: Parent’s sibling is red

Nodes are either red or black
2. Root is always black
3. If a node is red, all its children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes

Notice this is the “general case” we saw earlier!
Insertions: Parent’s sibling is red

Insert 35?

Was it possible for 85 to be red, so rotation + recoloring would not fix this problem?
A. Yes
B. No

Rotate!

Notice this is the “general case” we saw earlier!
If X’s Parent (P) is red, P is a left child of G, X is a left child of P, (and P’s sibling (S) is black), then Rotate P right, flip colors of P and G.
Insertions: Parent’s sibling is red

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all its children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes

Insert 35?
Insertions: Parent’s sibling is red

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all its children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes
Insertions: Parent’s sibling is red

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all its children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes
Insertions: Parent’s sibling is red

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all its children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes
1. Nodes are either red or black
2. Root is always black
3. If a node is red, all its children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes
Insertions: Parent’s sibling is red

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all its children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes
Can any node have 2 red children?

- As we descend the tree, we detect if a node X has 2 red children, and if so we do an operation to change the situation.

- Note that in doing so:
  - we may change things so that a node above X now has 2 red children, where it didn’t before! (example: node 60 after we insert 35)
  - if we have to do a double rotation, we will move X up and recolor it so that it becomes black, and has 2 red children itself! (example: work through inserting 64 in the tree on the following page)

- But neither of these is a problem, because:
  - it never violates any of the properties of red-black trees (those 2 red nodes will always have a black parent, for example),
  - and the 2 red siblings will be too “high” in the tree for either of them to be the sibling of the parent of any red node that we find or create when we continue this descent of the tree.
Exercise: Insert 64 into this tree
Exercise: Insert 64 into this tree

Recolor
Exercise: Insert 64 into this tree

Double rotation (rotation 1)
Exercise: Insert 64 into this tree

Double rotation (rotation 2)
Exercise: Insert 64 into this tree

Recolor
Exercise: Insert 64 into this tree
B- Trees

- Data structures for efficient search on secondary storage
Typical memory hierarchy: a picture

<table>
<thead>
<tr>
<th>AMOUNT OF STORAGE (approx!)</th>
<th>CPU</th>
<th>TIME TO ACCESS (approx!)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hundreds of bytes</td>
<td>CPU registers</td>
<td>1 nanosecond</td>
</tr>
<tr>
<td>hundreds of kilobytes</td>
<td>cache</td>
<td>10 nanoseconds</td>
</tr>
<tr>
<td>hundreds of megabytes</td>
<td>main memory</td>
<td>100 nanoseconds</td>
</tr>
<tr>
<td>hundreds of gigabytes</td>
<td>disk</td>
<td>10 milliseconds</td>
</tr>
</tbody>
</table>

1 sec = 1,000 millisec = 1,000,000 microsec = 1,000,000,000 nanosec
Consequences of the memory hierarchy

- Accessing a variable can be fast or slow, depending on various factors

- If a variable is in slow memory, accessing it will be slow

- However, when it is accessed, the operating system will typically move that variable to faster memory ("cache" or "buffer" it), along with some nearby variables
  - The idea is: if a variable is accessed once in a program, it (and nearby variables) is likely to be accessed again

- So it is possible for one access of a variable to be slow, and the next access to be faster; possibly orders of magnitude faster

  \[
  \begin{align*}
  x &= z[i]; & \text{// if } z[i] \text{ is on disk this takes a long time} \\
  z[i] &= 3; & \text{// now } z[i] \text{ is in cache, so this is very fast!} \\
  z[i+1] &= 9; & \text{// nearby variables also moved, so this is fast}
  \end{align*}
  \]

- The biggest speed difference is between disk access and semiconductor memory access, so that’s what we will pay most attention to
The disk drive

Fig. 2.1 Magnetic disk drive: (a) Data are stored on magnetized platters that rotate at a constant speed. Each platter surface is accessed by an arm that contains a read/write head, and data are stored on the platter in concentric circles called tracks. (b) The arms are physically connected so that they move in unison. The tracks (one per platter) that are addressable when the arms are in a fixed position are collectively referred to as a cylinder.
**Accessing data on disk**

- Data on disk is organized into *blocks*

- Typical block size: 1 kilobyte (1024 bytes), 4 kilobytes (4096 bytes), or more

- Because of the physical properties of disk drives (head seek time, platter rotation time), it is approximately as fast to read an entire disk block at once as it is to read any part of the block

- So, if you access a variable stored on disk, the operating system will read the entire disk block containing that variable into semiconductor memory

- While in semiconductor memory, accessing any item in that block will be fast

- Because disk accesses are many (thousands!) of times slower than semiconductor memory accesses, if a datastructure is going to reside on disk, it is important that it can be used with very few disk accesses

- The most commonly used data structure for large disk databases is a B-tree, which can be designed to use disk accesses very efficiently
## 2012 hardware Characteristics

<table>
<thead>
<tr>
<th></th>
<th>Cost</th>
<th>Power</th>
<th>Block Size</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1-L2</strong></td>
<td>On-Chip with CPU</td>
<td>low-end: iTouch: 1Watt high-end: Intel Core i7-950: 130Watt</td>
<td>L1: 32-64 Bytes  L2: 64-256 Bytes</td>
<td>100s of GB/sec</td>
</tr>
<tr>
<td><strong>DRAM</strong></td>
<td>8$-16$/GB</td>
<td>1-2 Watts/GB</td>
<td>max throughput at: 8-16KB</td>
<td>50-70 GB/sec</td>
</tr>
<tr>
<td><strong>SSD</strong></td>
<td>High end, $11/GB. Low end: $1-4/GB About $50,000 for ION</td>
<td>high end: 0.15W/GB Low End: 0.05 W/GB</td>
<td>4KB</td>
<td>high end: 1-3GB/sec low end: .1-.2GB/sec</td>
</tr>
<tr>
<td><strong>Disk</strong></td>
<td>0.03-0.1$/GB</td>
<td>0.01W/GB</td>
<td>4KB</td>
<td>100MB/sec sequential 0.4-2.0MB/sec random Getting to each block takes 2-10ms</td>
</tr>
</tbody>
</table>
Accessing data on disk

• Because disk accesses are many (thousands!) of times slower than semiconductor memory accesses, if a datastructure is going to reside on disk, it is important that it can be used with very few disk accesses

• The most commonly used data structure for large disk databases is a B-tree, which can be designed to use disk accesses very efficiently

• Operations that we are interested in:
  • Insert
  • Delete
  • Find

  All of them should be done with fewest disk accesses as possible
B-trees and storage

- Each node in a B-tree fits into a block (i.e., if you get part of the node, you get it all)
- Search tree property
- Keys in each node are sorted
The goal of B-Trees

- Always at least half full
- Perfectly Balanced
- Few levels
Properties of an m-order B trees

1. The root has at least 2 sub-trees, unless it is a leaf
2. All leaves are at the same level
3. Each node (leaf as well as non leaf) holds k-1 keys where \( \text{ceil}(m/2) \leq k \leq m \)
4. Each non-leaf node additionally holds k pointers to subtrees where \( \text{ceil}(m/2) \leq k \leq m \).
Order of B-trees

What order is this B-tree?
A. 2
B. 3
C. 4
D. 5
E. 6
Insertion and properties of B-trees

What is the minimum number of keys each non-root node in this B-tree is allowed to store?

A. 0
B. 1
C. 2
D. 3
E. 4

How can we guarantee this?
Insert 21 into this B-tree. Then insert 50
Insertion and properties of B-trees

Insert 15 into this B-tree
Insertion and properties of B-trees

Insert 22 and 23
Insertion and properties of B-trees

Insert 16
Insertion and properties of B-trees

Insert 16, after
Insertion and properties of B-trees

True or false: after the insert, the leaves will be at different levels
A. True
B. False
Insertion and properties of B-trees

Insert 62

Which key will be promoted up?
A. 61  B. 62  C. 68  D. 75  E. 80
Insertion and properties of B-trees

Insert 62

Which key will be promoted up?
A. 13  B. 25  C. 60  D. 85  E. 68
Insertion and properties of B-trees

Insert 62
Insertion and properties of B-trees

B-trees grow up! (Which is why all their leaves are always at the same level)
Insertion and properties of B-trees

Is this tree also a B+ tree?
A. Yes
B. No

Make it one (preserve only the information in the leaves)
A B+ tree
(2-3-4) B-trees
(2-3-4) B-trees == RBTs!
(2-3-4) B-trees == RBTs!

So which are asymptotically faster? RB-trees or 2-3-4 (B) trees?
A. RB trees
B. B-trees
C. They are the same
B-Tree performance

• The time savings in a B-Tree comes from \textit{efficiently reading lots of data from disk}
• When B-Trees are stored in memory they are typically comparable to other search trees
• When they have to access disk they are a big win

(For details see the slides here:
http://cseweb.ucsd.edu/users/kube/cls/100/Lectures/lec17/lec17.pdf

You will be responsible for the general ideas behind the tradeoffs of their design, but not the details. Example questions next class.)