CSE 100: RED-BLACK TREES (CONTD.)
B TREES
True or false: Red-Black trees are an example of a self-balancing binary search tree. That is, they reorganize themselves when you insert nodes, so no global rebalancing is ever necessary.

A. True
B. False
Q2: True or false: Nodes in a red-black tree are assigned a color once and may never be recolored.

A. True  
B. False
Q3: When a node is first inserted in a red-black tree, it is placed according to the insert procedure in a binary search tree. What color is this newly inserted node (initially) if it is NOT the root?

A. Red
B. Black
C. Either Red or Black
D. It has no color
Q4: Which of the following facts did we use to prove that the height of a red-black tree is $O(\log_2 N)$ (where $N$ is the number of nodes in the tree)?

A. The black height of the tree is bounded by $\log_2(N+1)$
B. The number of black nodes is less than or equal to $N$
C. Both A and B
Which of the following are legal red-black trees?

A. 12
B. 82
C. 81
D. A&B
E. A&B&C
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

Insert 35?
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

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

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

![Red-Black Tree Diagram]

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

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 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
Insertions: Parent’s sibling is red

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

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

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

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

Insert 35?

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
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 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
Insertions: Parent’s sibling is red

30

15

10

20

5

30

60

50

40

55

65

80

90

70

85

35

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
Exercise: Insert 64 into this tree

At this point we find that both children of node 65 are red. We have to fix this before proceeding further down the tree. We will keep a pointer to 65, so that we can continue down the tree after fixing the problem of 2 red children.
Exercise: Insert 64 into this tree

If a node has two red children (in this case 65):
- Recolor children black
- Recolor the node as red
- Check the parent of the node (65). If it is red, we have a problem. Need to fix this. We can be sure that the sibling of 60 is not red (if 60 is red). Think on way?
Exercise: Insert 64 into this tree

Double rotation (rotation 1)

Since the sibling of 65 is black, this defaults to case 1.
We can fix this problem in one or two rotations and a color flip (see rotation shown above).
Exercise: Insert 64 into this tree

Double rotation (rotation 2)

Now flip the color of 65.
Exercise: Insert 64 into this tree

We will now continue from where we left off. (Node 65)

We encounter the condition that 65 has two red children (again).

This would not be a problem in this tree if we go to insert 64 because the children of 60 are black & they have no red children. It would be a problem if 63 had two red children.

This is where we left
Fix this condition
Exercise: Insert 64 into this tree

The parent of 65 is black. (Phew!) Proceed to insert 64 from this point.

Recolor
Exercise: Insert 64 into this tree
Experiment with red-Black trees

The link below will help you interactively visualize insertions in a red-black tree
https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
B- Trees

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

AMOUNT OF STORAGE (approx!)

- hundreds of bytes
- hundreds of kilobytes
- hundreds of megabytes
- hundreds of gigabytes

CPU

- CPU registers
- cache
- main memory
- disk

TIME TO ACCESS (approx!)

- 1 nanosecond
- 10 nanoseconds
- 100 nanoseconds
- 10 milliseconds ~ 10 nanoseconds

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

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

- 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 data structure 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 \( \text{ceil}(m/2) \leq k \leq m \).

If \( m \) is the order of the tree: each non-leaf node can have a maximum of \( m \) children.
What order is this B-tree?

A. 2
B. 3
C. 4
D. 5
E. 6

It is important to know how the order of the B-tree is selected when designing the B-tree. The order of the B-tree is selected solely based on the block size of the disk read/write. In a way that the size of each node is a block size (see Paul Kubr's slides on the details).
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?
B-Tree performance

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

For details see Paul Kube’s slides here:

http://cseweb.ucsd.edu/users/kube/cls/100/Lectures/lec17/lec17.pdf