CS163: Data structures

Week 5: Trees (cont.)

6/2016

CS163: today?

- **Continue Discussing Trees**
- Examine more advanced trees
  - 2-3 (evaluate what we learned)
  - B-Trees
  - AVL
  - 2-3-4
  - red-black trees

Discuss 2-3 Trees

- **A 2-3 tree is always balanced**
- Therefore, you can search it in all situations with logarithmic efficiency of the binary search
- You might be concerned about the extra work in the insertion/deletion algorithms to split and merge the nodes...

Discuss 2-3 Trees

- **But,** rigorous mathematical analysis has proved that this extra work to maintain structure is <u>not</u> significant
- It is sufficient to consider only the time required to locate an item (or a position to insert)

Discuss 2-3 Trees

- **So, if 2-3 trees are so good, why not have nodes that can have more data items and more than 3 children?**
- Well, remember why 2-3 trees are great?
  - because they are <u>balanced</u> and that balanced structure is pretty easy to maintain

Discuss 2-3 Trees

- The advantage is <u>not</u> that the tree is shorter than a balanced binary search tree
  - the reduction in height is actually offset by the extra comparisons that have to be made to find out which branch to take
  - actually a binary search tree that is balanced minimizes the amount of work required to support ADT table operations

Discuss 2-3 Trees

- But, with binary search trees balance is hard to maintain
  - A 2-3 tree is really a compromise
  - Searching may not be quite as efficient as a binary tree of minimum height
  - but, it is relatively simple to maintain

Discuss 2-3 Trees

- Allowing nodes to have more than 3 children would require more comparisons and would therefore be counter productive
  - unless you are working with external storage and each node requires a disk access, then we use b-trees which have the minimum height possible

Discuss B-Trees

- Tables stored externally can be searched with B-Trees.
  - B-Trees are a more generalized approach than the 2-3 Tree
  - With externally stored tables, we want to keep the search tree as short as possible; it is much faster to do extra comparisons at a particular node than try to find the next node.

Discuss B-Trees

- Every time we want to get another node,
  - we have to access the external file and read in the appropriate information.
  - It takes far less time to operate on a particular node (i.e., doing comparisons) once it has been read in.
  - This means that for externally stored tables we should try to reduce the height of the tree...even if it means doing more comparisons at every node.

Discuss B-Trees

- Therefore, with an external search tree,
  - we allow each node to have as many children as possible.
  - If a node is to have m children, then you must be able to allocate enough memory for that node to contain the data and m pointers to the node.
  - The data such a node must have m-1 key values.

Discuss B-Trees

- Remember in a binary search tree,
  - if a node has 2 children then it contains one data value (i.e., one value).
  - You can think of the data value at a node as separating the data values in the two child subtrees.
  - All keys to the left are less than the node's data value and all key values to the right are greater than or equal.
  - The value of the data at a particular node tells you which branch to take.

Discuss B-Trees

- In a 2-3 tree,
  - if a node has 3 children then it must contain two key values.
  - These two values separate the key values in the node's three child subtrees.
  - All of the key values in the left subtree are less than the node's smaller key value;
  - all of the key values in the middle subtree are between the node's two key values;
  - all of the key values in the right subtree are greater than or equal to the node's larger key

Discuss B-Trees

- Ideally, you should structure these types of trees such that every internal node has m children and all leaves are at the same level.
- For example, if m is 5 -- then every node should have 5 children and 4 data values.
  - But, this is too difficult to maintain when you are doing a variety of insertions and deletions.

Discuss B-Trees

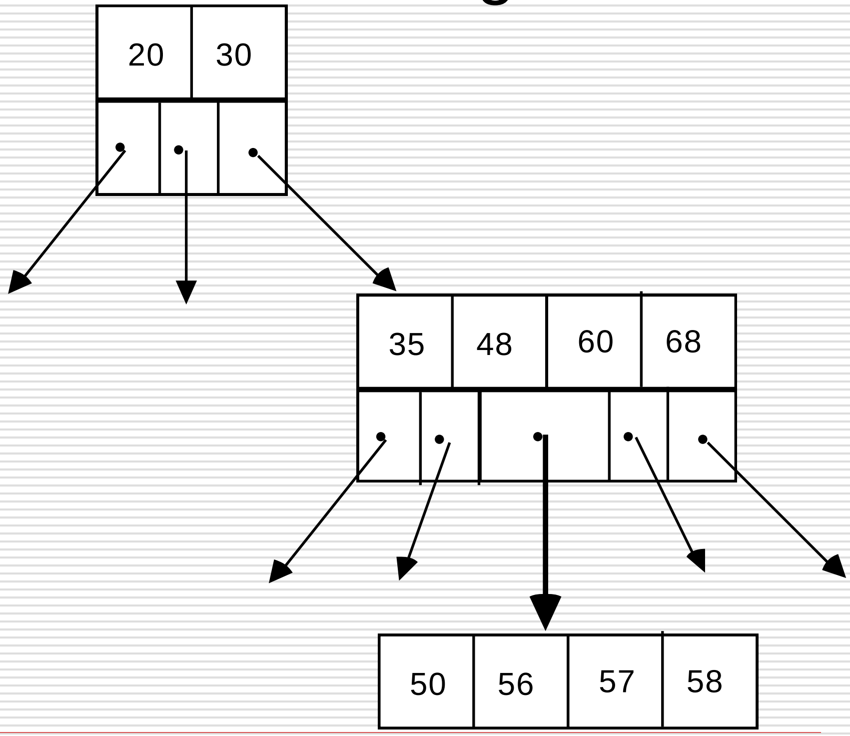
- So, we can require that B-trees be balanced (as we saw with 2-3 trees)...
  - but the number of children for any internal node should be able to be somewhere between m and (m div 2)+1.
- We call this a B-Tree of degree m
- This requires that all leaves be at the same level (balanced).

Discuss B-Trees

- Each node contains between m-1 and (m div 2) values.
- Each internal node has one more child than it has values.
- There is one exception;
  - the root of the tree can contain as few as 1 value and can have as few as two children (or none -- if the tree consists of only a root!).

Discuss B-Trees

- Notice, a 2-3 tree is a B-tree of degree 3.
- Data can be inserted into a B-tree using the same strategy of splitting and merging nodes that we discussed
- Here is a B-tree of degree 5:

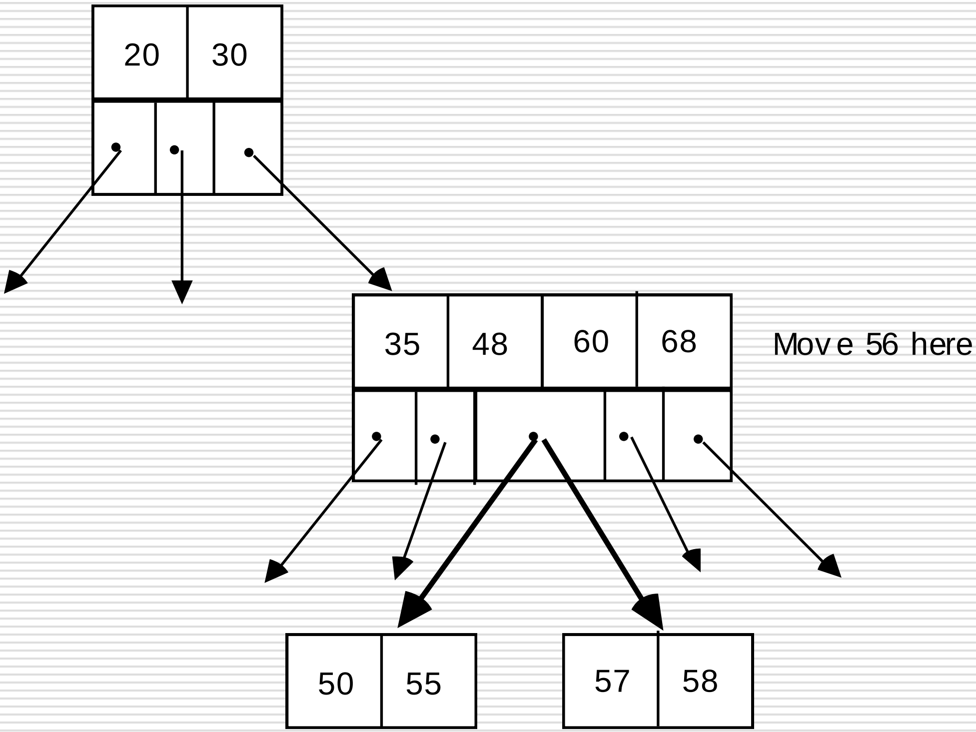


Discuss B-Trees

- Then, insert 55.
  - The first step is to locate the leaf of the tree in which this index belongs by determining where the search for 55 would terminate.
- We would find that we would want to insert 55 in the node containing 50,56,57, 58.
  - But, that would cause this node to contain 5 records. Since a node can contain only 4 records, you must split this node into two...the new left node gets the two smaller values and the new right node gets the two larger values.

Discuss B-Trees

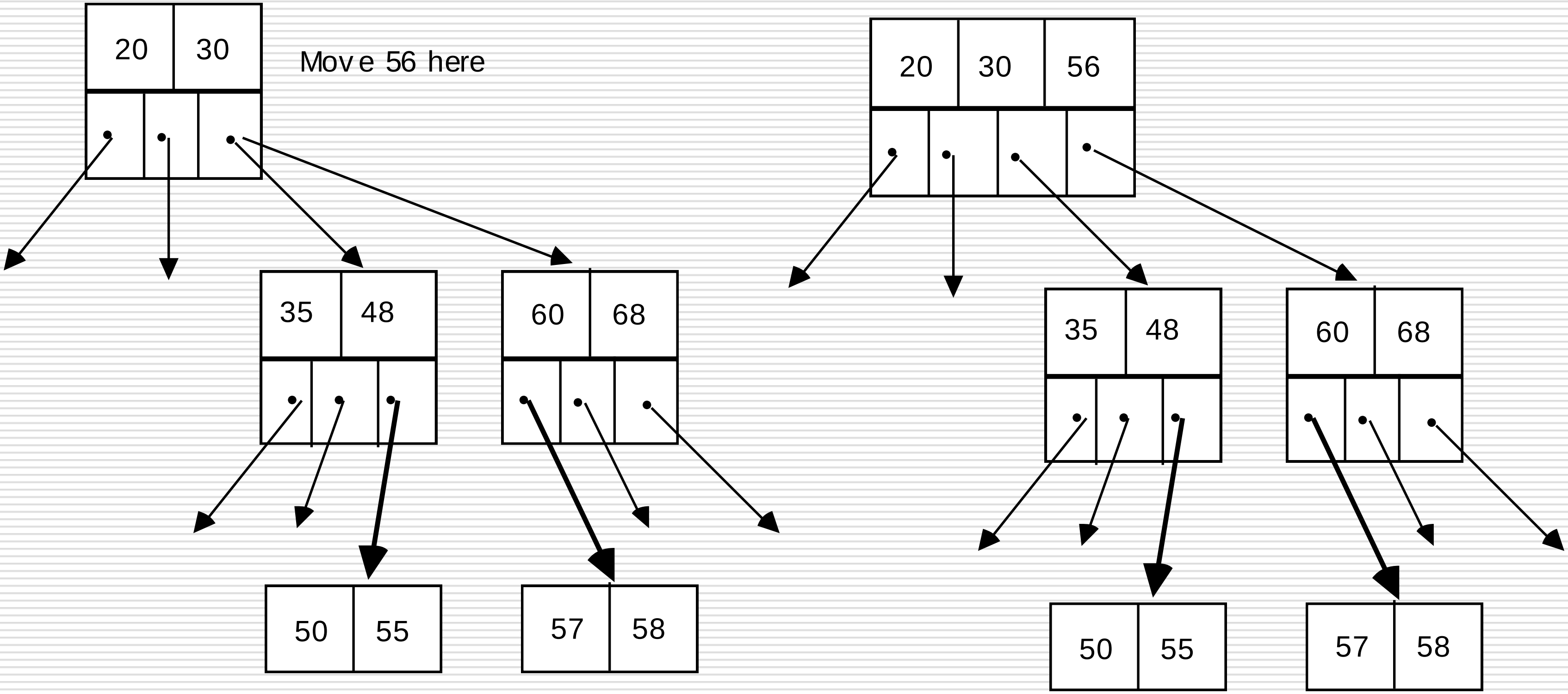
- The record with the middle key value (56) is moved up to the parent:



Discuss B-Trees

- This causes two problems,
  - the parent now has six children and five records!!
  - So, we must split the parent into two nodes and move the middle data value up to its parent.
  - Remember, when we split an internal node, we need to also move that node's children too
  - Since the root only has 2 data items, we can simply add 56 there.
  - The solution is on the next slide...

Discuss B-Trees



Discuss B-Trees

- Notice, that if the root had needed to be spit,
  - the new root will contain only one value and have only 2 children (that is why we have the exception to the B-Tree definition stated earlier).
- To traverse a B-Tree in sorted order, all we need to do is visit the search keys in sorted order by using an inorder traversal of the B-Tree.

Balancing Algorithms

- But, are there other alternatives?
- Remember the advantage of trees is that they are well suited for problems that are hierarchical in nature and they are much faster than linked lists
  - but, this is not valid if the tree in not balanced
  - luckily, there are a number of techniques to balance a binary tree

Balancing Algorithms

- Some of the balancing techniques require constant restructuring of the tree as data is inserted
  - the AVL algorithm uses this approach
- Some algorithms consist of build an unbalanced tree and then reordering the data once the tree is generated
  - this can be simple but depending on the frequency of data being inserted, it may not be realistic

Balancing Algorithms

- The “brute force” technique is to create an array of pointers to your data by traversing an unbalanced BST using “inorder” traversal
  - then re-build the tree by splitting the array in the middle for each subarray (much like what we have seen with the binary search algorithm used with arrays)
  - the middle data item <u>should</u> be the root, as it splits what is less than it, and what is greater!

Balancing Algorithms

- The algorithm for the "brute force" approach is:

```
balance(data_type data [], int
   first, int last)
   if (first <= last) {
      int middle = (first + last)/2;
      insert(data[middle]);
      balance(data, first, middle-1);
      balance(data, middle+1, last);
```

Balancing Algorithms

- The "brute force" technique has a serious drawback
  - all of the data must be put in an array before a balanced tree can be created
  - what would happen if you weren't using pointers to the data but instances of the data?
  - if an unbalanced tree is not used (i.e., the data is directly inserted into the array from the client), then a sorting algorithm must be used and fixed size issues arise

AVL Trees

- The AVL tree is a classical method proposed by Adelson-Velskii and Landis
  - creates an “admissible tree” (its original name!)
  - focuses on rebalancing the tree locally to the portion of the tree affected by insertion and deletions
  - it allows the height of the left and right subtrees of every node to differ by at most one

AVL Trees

- With AVL trees
  - each node must now keep track of the “balance factors” which records the differences between the heights of the left and right subtrees
  - the balance factor is the height of the right subtree minus the height of the left subtree
  - all balance factors must be +1, 0, or -1
  - notice, this <u>does</u> meet the definition we learned about for a balanced tree

AVL Trees

- However, the concept of AVL trees always includes implicitly the techniques for balancing trees
  - and does not guarantee that the resulting tree is perfectly balanced (unlike all of the other techniques we have seen so far)
  - but, an AVL tree can be searched almost as efficiently as a minimum height binary search tree
  - but insert and removal are not as efficient

AVL Trees

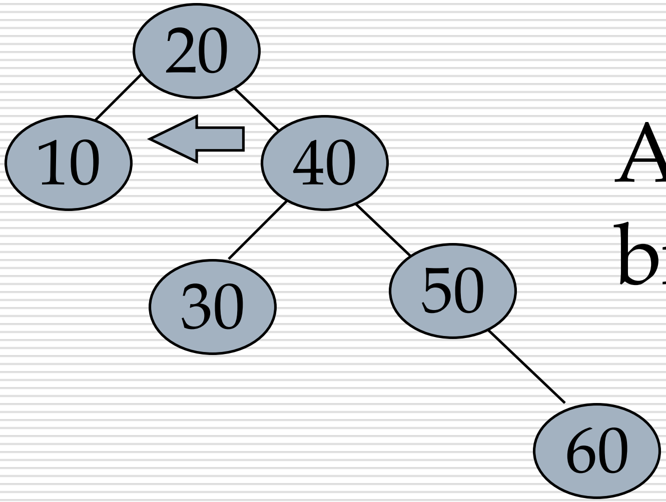
- AVL trees actually maintains the height close to minimum by monitoring the shape of the tree as you insert and delete
- After you insert/delete
  - the tree is checked to see if any node differs by more than 1 in height
  - if it does, you rearrange the nodes to restore balance
  - But, as you can guess, we can’t arbitrarily rearrange nodes....we must keep proper order

AVL Trees

- What we do is <u>rotate</u> the tree to make it balanced
- Rotations are <u>not</u> necessary after every insertion & deletion (it is only needed when the height differs by more than 1)
  - experiments indicate that deletions in 78% of the cases require no rebalancing
  - and only 53% of the insertions do not bring the tree out of balance

AVL Trees

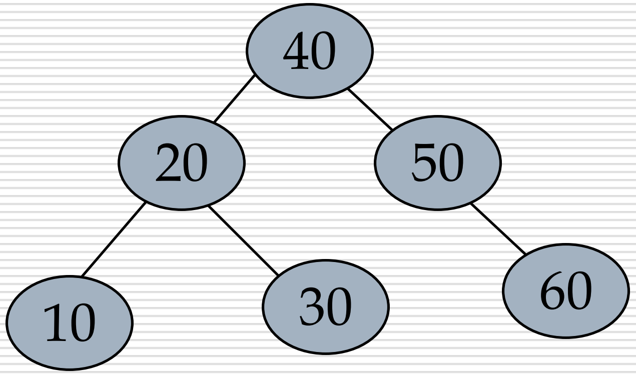
- Single rotation is one type of rotation:
  - In the following, the tree was fine after inserting 20, 10, 40, 30, 50...but when 60 is inserted...



An unbalanced binary search tree

AVL Trees

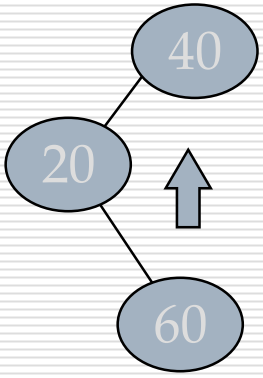
- Start at the node inserted...move up the tree (recursively return)
  - examining the balancing factor
  - stop when it is not +1, 0, -1 and rotate from the "heavy" side to the "light"



40 rotates up, 20 inherits 40's left child

AVL Trees

- If a single rotation does not create a balanced tree
  - then a double rotation is required
  - first rotate the subtree at the root where the problem occurred
  - <u>and</u> then rotate the tree's root
  - there is, however, on special case:



AVL Trees

- In class, walk through a few examples on your own (and then on the board) building AVL trees
  - so you can understand the process of rotations
  - insert: 50,60,30,70,55,20,52,65,40
  - or, insert: 10, 20, 30, 40, 50, 60, 70, 80
  - what would the corresponding BST and 2-3 tree looked like?

AVL Trees

- The main question you should be facing with an AVL tree is
  - whether or not such restructuring is always necessary
  - binary search trees are used to insert, retrieve, and delete elements quickly and the speed of performing these operations is the issue, not the shape of the tree
  - performance can be improved by balancing the tree but luckily this is not the only method available

2-3-4 and red-black Trees

- Now let’s go back to rethinking about how we organize our nodes
  - maybe instead of trying to balance the tree we keep the tree balancing at all times (perfectly balanced)
  - but the 2-3 tree had a flaw in that there may be situations where each node is “full” requiring a rippling effect of nodes being split as you recursively return back to the root

2-3-4 and red-black Trees

- A 2-3-4 tree solves this problem
  - which allows **4-nodes which are nodes that have 4 pieces of data and 3 children**
  - each insertion and deletion can have fewer steps than are required by a 2-3 tree (when looking at the insertions/deletions in isolation)
  - but does this by using more memory
  - essentially, each node can have 1,2, or 3 pieces of data, and 4 child pointers!!!!!

2-3-4 and red-black Trees

- A 2-3-4 tree solves this problem
  - a node can either be a leaf or,
  - if it has 1 data item there are 2 children,
  - 2 data items has 3 children, and
  - 3 data items has 4 children
- A 2-3-4 tree remains perfectly balanced
  - but its insertion algorithm splits the nodes as it traverses down the tree toward a leaf, rather than upon the return to the root

2-3-4 and red-black Trees

- As you travel down the tree to insert a data item,
  - if you encounter a node with 3 pieces of data you immediate split the node at that time (just as we did with a 2-3 tree...but now we don't use the new data we are trying to insert...because we haven't inserted it yet!)
  - then, you continue traveling towards a leaf to insert the data

2-3-4 and red-black Trees

- What this means is that the tree cannot contain all nodes with 3 pieces of data. Impossible.
- In fact, on insert, once you insert data at a leaf it is guaranteed that the leaf's parent will <u>not</u> have 3 pieces of data...
  - because if it did, it would have split on the way to find the leaf!

2-3-4 and red-black Trees

- The advantage of both the 2-3 and 2-3-4 trees
  - is that they are easy to maintain balance (not that their height is shorter due to the extra comparisons required)
  - where the 2-3-4 tree has an advantage is that the insertion/deletion algs require only one pass through the tree so they are simpler than those for a 2-3 tree
  - decrease in effort makes them attractive………

2-3-4 and red-black Trees

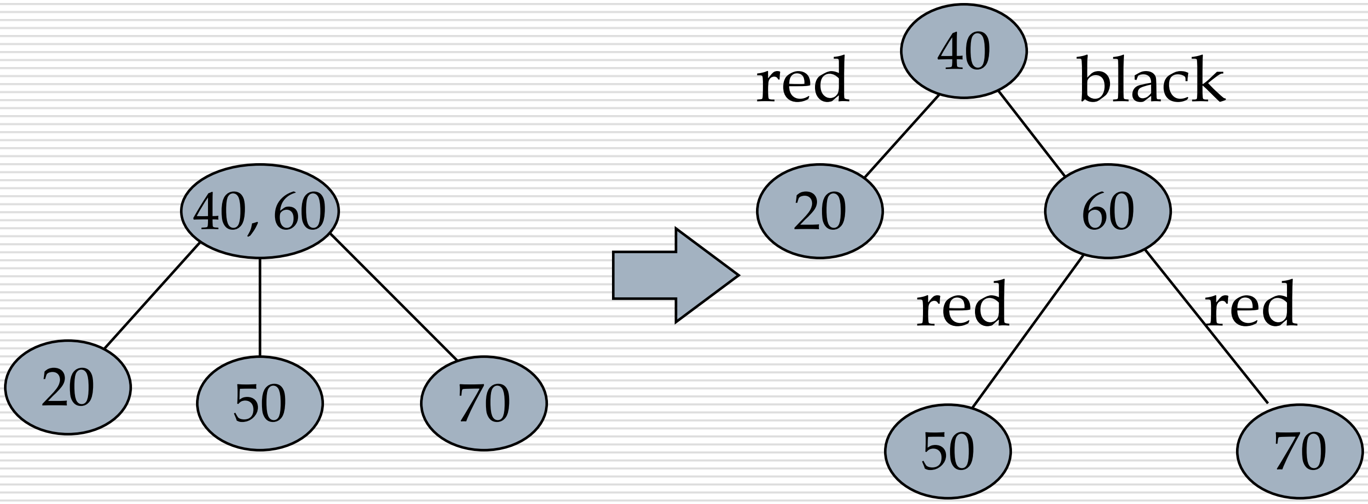
- On the other hand, 2-3-4 trees require more storage than a binary search tree
  - <u>and</u> more storage (and less efficiently used storage) than a 2-3 tree
- But, a binary search tree may be inappropriate
  - because it may not be balanced
  - so we use a red-black tree which is a special binary search tree

2-3-4 and red-black Trees

- A red-black tree is a BST representation of a 2-3-4 tree with 2 extra fields in the node to represent whether the connection is within the current node or a child
  - it retains the advantages of a 2-3-4 tree without the storage overhead!
  - with all of the benefits of a binary search tree and none of the drawbacks!

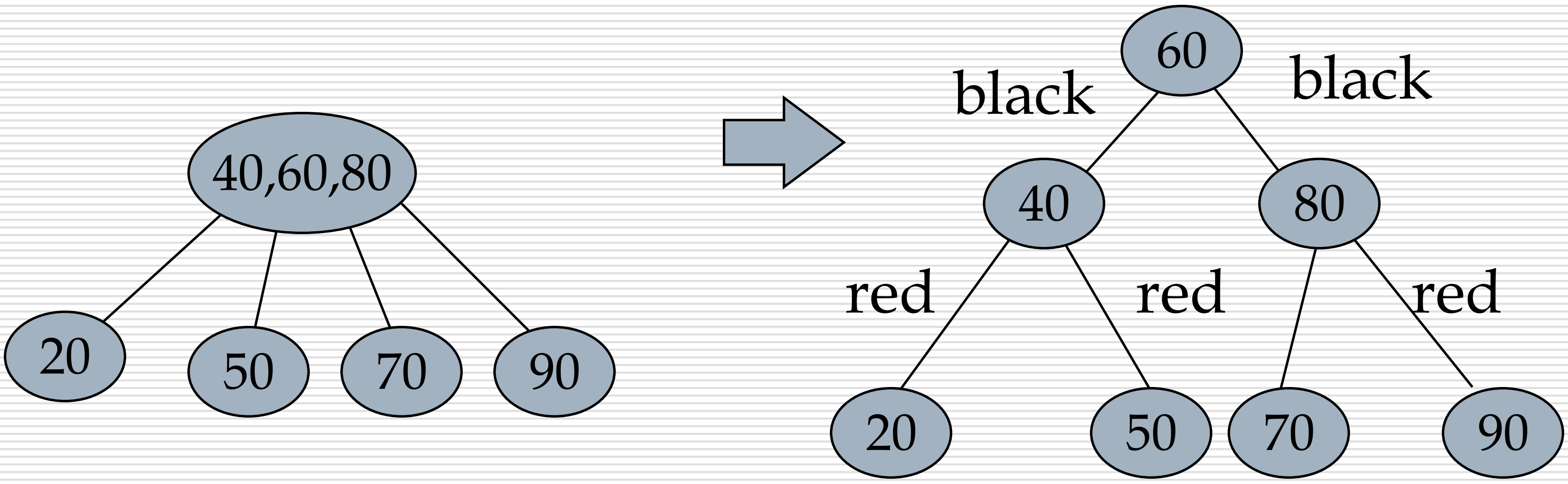
2-3-4 and red-black Trees

- The idea is to represent a node with 2 pieces of data and 3 children as a binary search tree with <u>red and black child pointers</u>



2-3-4 and red-black Trees

- And, we represent a node with 3 pieces of data and 4 children as a binary search tree with <u>red and black child pointers</u>



2-3-4 and red-black Trees

- In class, walk through examples of
  - 2-3
  - 2-3-4
  - AVL
  - BST
  - and see how you can take a 2-3-4 and turn it into a red black tree

2-3-4 and red-black Trees

- For next time,
  - practice creating each of these trees on your own so that you understand the insertion algorithms
  - think about what would be needed to remove nodes from these trees
  - try deleting a leaf and an internal node from you 2-3, AVL, and 2-3-4 trees