# Balanced Search Trees

Computer Science E-22
Harvard Extension School

David G. Sullivan, Ph.D.
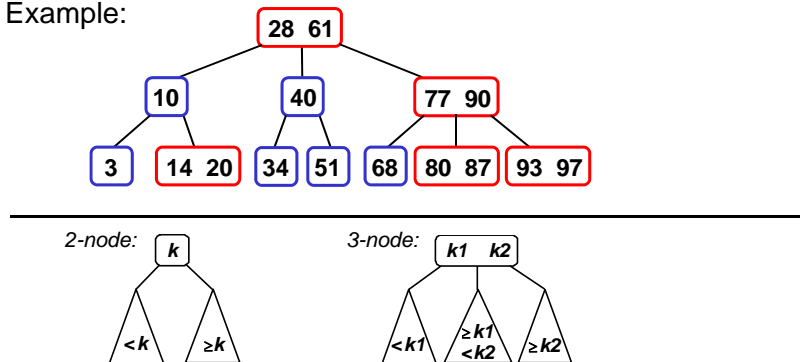

Review: Balanced Trees

- A tree is *balanced* if, for each node, the node's subtrees have the same height or have heights that differ by 1.

- For a balanced tree with n nodes:
    - height = $O(\log_2 n)$.

    - gives a worst-case time complexity that is logarithmic ($O(\log_2 n)$)
        - the best worst-case time complexity for a binary search tree

- With a binary search tree, there's no way to ensure that the tree remains balanced.
    - can degenerate to $O(n)$ time
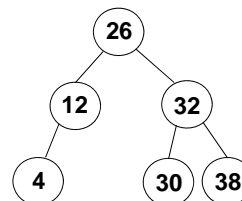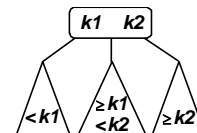
2-3 Trees

- A 2-3 tree is a balanced tree in which:
  - *all* nodes have equal-height subtrees (perfect balance)
  - each node is either
    - a **2-node**, which contains one data item and 0 or 2 children
    - a **3-node**, which contains two data items and 0 or 3 children
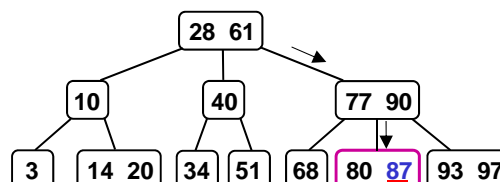 - the keys form a search tree

- Example:

```
                        28  61
            ┌─────────────┼──────────────┐
          10            40             77  90
        ┌──┴──┐       ┌──┴──┐      ┌─────┼─────┐
       3   14 20    34    51     68   80 87   93 97
```

2-node: ┌─┐
         │k│
         └─┘
        /   \
     <k   ≥k

3-node: ┌──┬──┐
         │k1│k2│
         └──┴──┘
        / | \
     <k1 ≥k1   ≥k2
          <k2

Search in 2-3 Trees

- Algorithm for searching for an item with a key k:

  if $k ==$ one of the root node's keys, you're done
  else if $k <$ the root node's first key
      search the left subtree
  else if the root is a 3-node and k < its second key
   search the middle subtree
  else
      search the right subtree

  ┌──┬──┐
  │k1│k2│
  └──┴──┘
 / | \
<k1 ≥k1 ≥k2
    <k2

- Example: search for 87

```
                        28  61
            ┌─────────────┼──────────────┐
          10            40             77  90
        ┌──┴──┐       ┌──┴──┐      ┌─────┼─────┐
       3   14 20    34    51     68   80 87   93 97
```

Insertion in 2-3 Trees

- Algorithm for inserting an item with a key *k*:

    search for *k,* but don't stop until you hit a leaf node

    let L be the leaf node at the end of the search

    if L is a 2-node

    add *k* to L, making it a 3-node

    else if L is a 3-node

    split L into two 2-nodes containing the items with the smallest and largest of: *k,* L's 1st key, L's 2nd key

    the middle item is "sent up" and inserted in L's parent
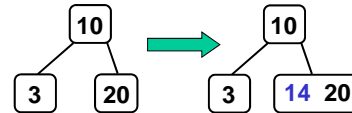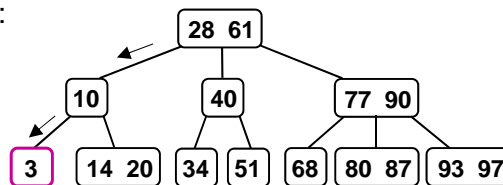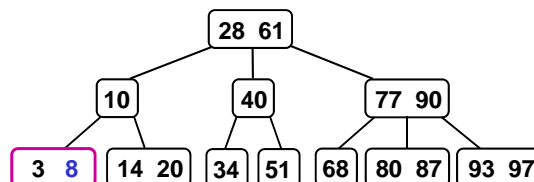
    example: add 52
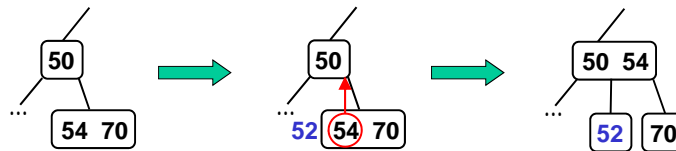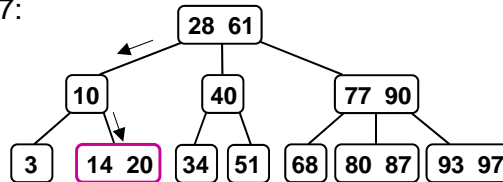
Example 1: Insert 8

- Search for 8:

- Add 8 to the leaf node, making it a 3-node:

Example 2: Insert 17

- Search for 17:

```
                    28 61
          10        40        77 90
      3   14 20   34  51   68  80 87  93 97
```
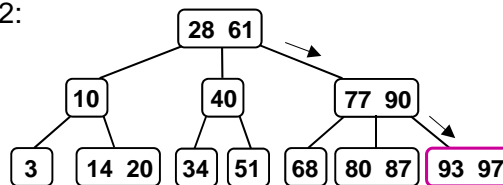
- Split the leaf node, and send up the middle of 14, 17, 20 and insert it the leaf node's parent:

```
        28 61                              28 61
   10        40      ...              10 17      40      ...
            17
  3  14 20  34 51                    3  14  20  34 51
```
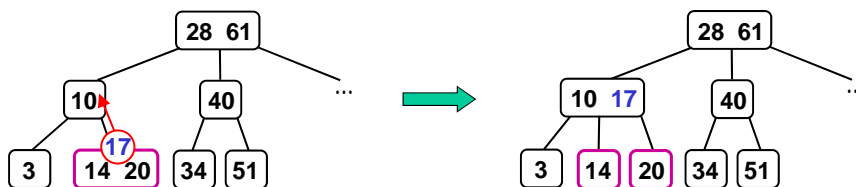
Example 3: Insert 92

- Search for 92:

```
                    28 61
          10        40        77 90
      3   14 20   34  51   68  80 87  93 97
```

- Split the leaf node, and send up the middle of 92, 93, 97 and insert it the leaf node's parent:

```
      28 61                              28 61
  ...     40      77 90              ...     40      77 90 93
       34 51   68  80 87  92 93 97         34 51  68  80 87  92  97
```
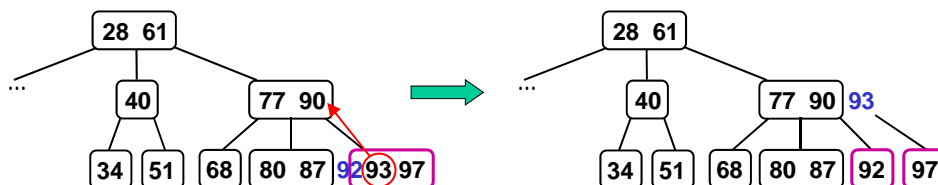
- In this case, the leaf node's parent is also a 3-node, so we need to split is as well…

Splitting the Root Node

- If an item propagates up to the root node, and the root is a 3-node, we split the root node and create a new, 2-node root containing the middle of the three items.

- Continuing our example, we split the root's right child:
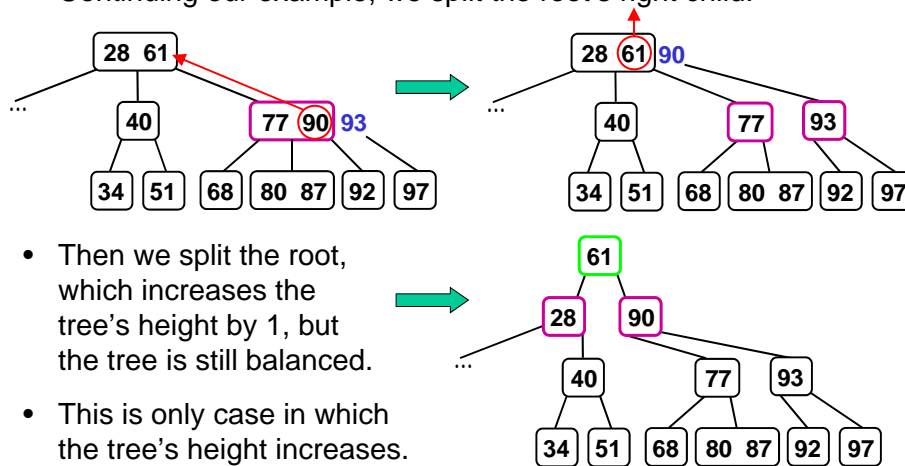
- Then we split the root, which increases the tree's height by 1, but the tree is still balanced.

- This is only case in which the tree's height increases.

Efficiency of 2-3 Trees
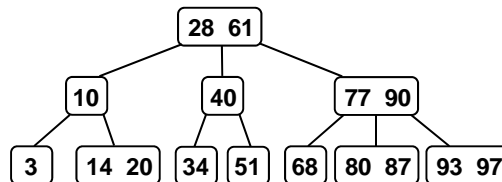
- A 2-3 tree containing n items has a height <= $\log_2 n$.

- Thus, search and insertion are both *O*(log n).
    - a search visits at most $\log_2 n$ nodes
    - an insertion begins with a search; in the worst case, it goes all the way back up to the root performing splits, so it visits at most $2\log_2 n$ nodes

- Deletion is tricky – you may need to coalesce nodes! However, it also has a time complexity of *O*(log n).

- Thus, we can use 2-3 trees for a *O*(log n)-time data dictionary.

External Storage

- The balanced trees that we've covered don't work well if you want to store the data dictionary externally – i.e., on disk.

- Key facts about disks:
  - data is transferred to and from disk in units called *blocks,* which are typically 4 or 8 KB in size
  - disk accesses are slow!
    - reading a block takes ~10 milliseconds (10^{-3} sec)
    - vs. reading from memory, which takes ~10 nanoseconds
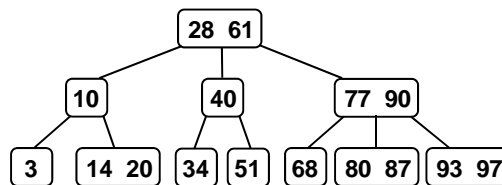   - in 10 ms, a modern CPU can perform millions of operations!

B-Trees

- A B-tree of order m is a tree in which each node has:
  - at most $2m$ entries (and, for internal nodes, $2m + 1$ children)
  - at least m entries (and, for internal nodes, $m + 1$ children)
  - exception: the root node may have as few as 1 entry
  - a 2-3 tree is essentially a B-tree of order 1

- To minimize the number of disk accesses, we make m as large as possible.
  - each disk read brings in more items
  - the tree will be shorter (each level has more nodes), and thus searching for an item requires fewer disk reads

- A large value of m doesn't make sense for a memory-only tree, because it leads to many key comparisons per node.

- These comparisons are less expensive than accessing the disk, so large values of m make sense for on-disk trees.

Example: a B-Tree of Order 2

```
                  20  40  68  90
        ┌─────────┬────┴────┬─────────┐
   3 10 14  28 34  51 61  77 80 87  93 97
```
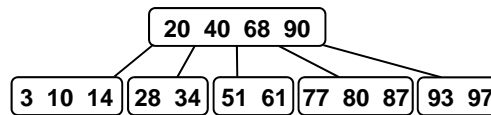
- Order 2: at most 4 data items per node (and at most 5 children)

- The above tree holds the same keys as one of our earlier 2-3 trees, which is shown again below:

```
                    28  61
          ┌───────────┼───────────┐
         10          40         77  90
       ┌──┴──┐      ┌─┴─┐      ┌──┬──┴──┐
      3   14 20   34   51   68   80 87  93 97
```

- We used the same order of insertion to create both trees:

  51, 3, 40, 77, 20, 10, 34, 28, 61, 80, 68, 93, 90, 97, 87, 14

- For extra practice, see if you can reproduce the trees!

Search in B-Trees

- Similar to search in a 2-3 tree.

- Example: search for 87

```
                  20  40  68  90
        ┌─────────┬────┴────┬─────────┐
   3 10 14  28 34  51 61  77 80 87  93 97
```
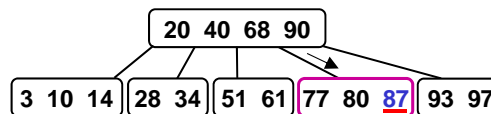
Insertion in B-Trees

- Similar to insertion in a 2-3 tree:

  search for the key until you reach a leaf node

  if a leaf node has fewer than $2m$ items, add the item to the leaf node

  else split the node, dividing up the $2m + 1$ items:

  the smallest m items remain in the original node

  the largest m items go in a new node

  send the middle entry up and insert it (and a pointer to the new node) in the parent

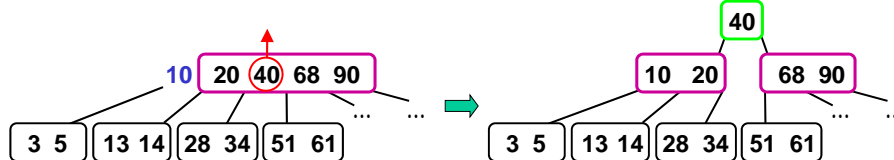- Example of an insertion without a split: insert 13
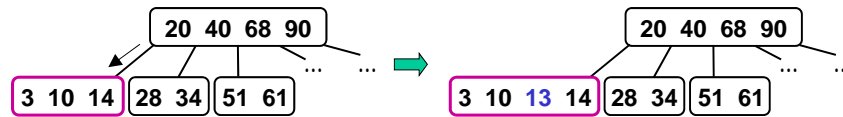
Splits in B-Trees

- Insert 5 into the result of the previous insertion:
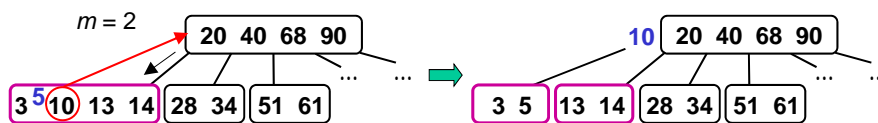
- The middle item (the 10) was sent up to the root.
  It has no room, so it is split as well, and a new root is formed:

- Splitting the root increases the tree's height by 1, but the tree is still balanced. This is only way that the tree's height increases.

- When an internal node is split, its $2m + 2$ pointers are split evenly between the original node and the new node.

Analysis of B-Trees

```
                    20  40  68  90
              ┌────┬────┬────┬────┴──────┐
        3 10 14  28 34  51 61  77 80 87  93 97
```
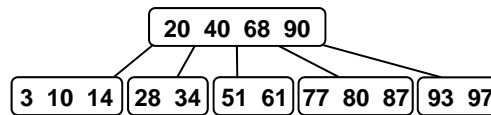
- All internal nodes have at least *m* children (actually, at least *m*+1).

- Thus, a B-tree with n items has a height $\leq \log_m n$, and search and insertion are both $O(\log_m n)$.

- As with 2-3 trees, deletion is tricky, but it's still logarithmic.

Search Trees: Conclusions

- Binary search trees can be $O(\log n)$, but they can degenerate to $O(n)$ running time if they are out of balance.

- 2-3 trees and B-trees are *balanced* search trees that guarantee $O(\log n)$ performance.

- When data is stored on disk, the most important performance consideration is reducing the number of disk accesses.

- B-trees offer improved performance for on-disk data dictionaries.