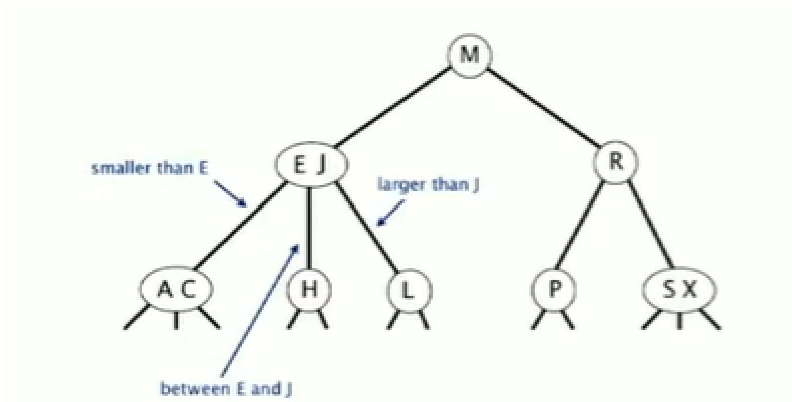
Balanced Search Trees

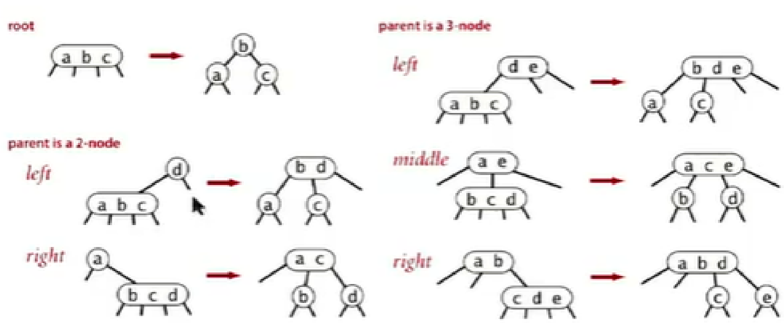
Search Trees

2-3 tree

- Allow 1 or 2 keys per node
  - 2-node: one key, two children
  - 3-node: two keys, three children
- Perfect balance---every path from root to null link has same length



- insert
  - insert into a 2-node at bottom
  - insert into a 3-node at bottom
    - add new key to 3-node to create temporary 4-node
    - move middle key in 4-node into parent
    - repeat up the tree, as necessary
- Properties
  - Invariants---each transformation maintains symmetric order and perfect balance



  - Perfect balance--every path from root to null link has same length, guaranteed log performance
    - tree height
      - worst case---$logN$

- best case---$log_3 N$

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	equals()
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	compareTo()
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	compareTo()
2-3 tree	c lg N	c lg N	c lg N	c lg N	c lg N	c lg N	yes	compareTo()

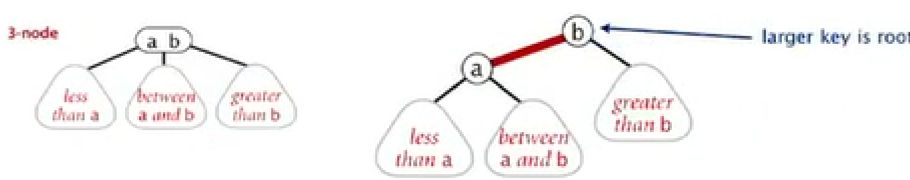
Red-black BSTs

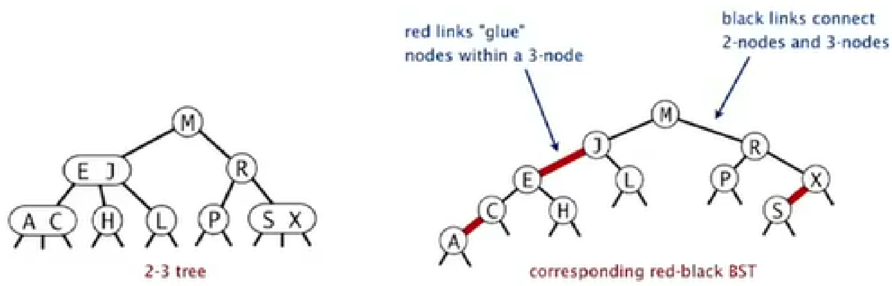
Left-leaning red-black BSTs

> represent 2-3 tree as a BST

> use "internal" left-leaning links as "glue" for 3 nodes

- A BST such that
  - no node has two red links connected to it
  - every path from root to null link has the same number of black links
  - red links lean left





- Search
  - the same as for elementary BST (ignore color)
- Insert

Complexity

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	equals()
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	compareTo()
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	compareTo()
2-3 tree	c lg N	c lg N	c lg N	c lg N	c lg N	c lg N	yes	compareTo()
red-black BST	2 lg N	2 lg N	2 lg N	1.00 lg N *	1.00 lg N *	1.00 lg N *	yes	compareTo()

Implementation

```
import java.util.NoSuchElementException;

public class RedBlackBST<Key extends Comparable<Key>, Value> {

    private static final boolean RED   = true;
    private static final boolean BLACK = false;

    private Node root;     // root of the BST

    // BST helper node data type
    private class Node {
        private Key key;           // key
        private Value val;         // associated data
        private Node left, right;  // links to left and right subtrees
        private boolean color;     // color of parent link
        private int size;          // subtree count

        public Node(Key key, Value val, boolean color, int size) {
            this.key = key;
            this.val = val;
            this.color = color;
            this.size = size;
        }
    }

    public RedBlackBST() {
    }

    private boolean isRed(Node x) {
        if (x == null) return false;
        return x.color == RED;
    }

    private int size(Node x) {
        if (x == null) return 0;
        return x.size;
    }

```

```
    public int size() {
        return size(root);
    }

    public boolean isEmpty() {
        return root == null;
    }

    public Value get(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to get() is null");
        return get(root, key);
    }

    private Value get(Node x, Key key) {
        while (x != null) {
            int cmp = key.compareTo(x.key);
            if      (cmp < 0) x = x.left;
            else if (cmp > 0) x = x.right;
            else              return x.val;
        }
        return null;
    }

    public boolean contains(Key key) {
        return get(key) != null;
    }

    public void put(Key key, Value val) {
        if (key == null) throw new IllegalArgumentException("first argument to put() is null");
        if (val == null) {
            delete(key);
            return;
        }

        root = put(root, key, val);
        root.color = BLACK;
    }

    private Node put(Node h, Key key, Value val) {
        if (h == null) return new Node(key, val, RED, 1);

        int cmp = key.compareTo(h.key);
        if      (cmp < 0) h.left  = put(h.left,  key, val);
        else if (cmp > 0) h.right = put(h.right, key, val);
        else              h.val   = val;

        // fix-up any right-leaning links
        if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h);
        if (isRed(h.left)  &&  isRed(h.left.left)) h = rotateRight(h);
        if (isRed(h.left)  &&  isRed(h.right))     flipColors(h);
```

```
        h.size = size(h.left) + size(h.right) + 1;

        return h;
    }

    public void deleteMin() {
        if (isEmpty()) throw new NoSuchElementException("BST underflow");

        // if both children of root are black, set root to red
        if (!isRed(root.left) && !isRed(root.right))
            root.color = RED;

        root = deleteMin(root);
        if (!isEmpty()) root.color = BLACK;
        // assert check();
    }

    // delete the key-value pair with the minimum key rooted at h
    private Node deleteMin(Node h) {
        if (h.left == null)
            return null;

        if (!isRed(h.left) && !isRed(h.left.left))
            h = moveRedLeft(h);

        h.left = deleteMin(h.left);
        return balance(h);

    public void deleteMax() {
        if (isEmpty()) throw new NoSuchElementException("BST underflow");

        // if both children of root are black, set root to red
        if (!isRed(root.left) && !isRed(root.right))
            root.color = RED;

        root = deleteMax(root);
        if (!isEmpty()) root.color = BLACK;
    }

    // delete the key-value pair with the maximum key rooted at h
    private Node deleteMax(Node h) {
        if (isRed(h.left))
            h = rotateRight(h);

        if (h.right == null)
            return null;

        if (!isRed(h.right) && !isRed(h.right.left))
            h = moveRedRight(h);

        h.right = deleteMax(h.right);

```

```java
        return balance(h);
    }

    public void delete(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to delete() is null");
        if (!contains(key)) return;

        // if both children of root are black, set root to red
        if (!isRed(root.left) && !isRed(root.right))
            root.color = RED;

        root = delete(root, key);
        if (!isEmpty()) root.color = BLACK;
        // assert check();
    }

    private Node delete(Node h, Key key) {
        // assert get(h, key) != null;

        if (key.compareTo(h.key) < 0)  {
            if (!isRed(h.left) && !isRed(h.left.left))
                h = moveRedLeft(h);
            h.left = delete(h.left, key);
        }
        else {
            if (isRed(h.left))
                h = rotateRight(h);
            if (key.compareTo(h.key) == 0 && (h.right == null))
                return null;
            if (!isRed(h.right) && !isRed(h.right.left))
                h = moveRedRight(h);
            if (key.compareTo(h.key) == 0) {
                Node x = min(h.right);
                h.key = x.key;
                h.val = x.val;
                h.right = deleteMin(h.right);
            }
            else h.right = delete(h.right, key);
        }
        return balance(h);
    }

    private Node rotateRight(Node h) {
        assert (h != null) && isRed(h.left);
        Node x = h.left;
        h.left = x.right;
        x.right = h;
        x.color = h.color;
        h.color = RED;
        x.size = h.size;
        h.size = size(h.left) + size(h.right) + 1;
```

```
            return x;
    }

    // make a right-leaning link lean to the left
    private Node rotateLeft(Node h) {
        assert (h != null) && isRed(h.right);
        // assert (h != null) && isRed(h.right) && !isRed(h.left);  // for insertion only
        Node x = h.right;
        h.right = x.left;
        x.left = h;
        x.color = h.color;
        h.color = RED;
        x.size = h.size;
        h.size = size(h.left) + size(h.right) + 1;
        return x;
    }

    // flip the colors of a node and its two children
    private void flipColors(Node h) {

        h.color = !h.color;
        h.left.color = !h.left.color;
        h.right.color = !h.right.color;
    }

    // Assuming that h is red and both h.left and h.left.left
    // are black, make h.left or one of its children red.
    private Node moveRedLeft(Node h) {

        flipColors(h);
        if (isRed(h.right.left)) {
            h.right = rotateRight(h.right);
            h = rotateLeft(h);
            flipColors(h);
        }
        return h;
    }

    private Node moveRedRight(Node h) {
        // assert (h != null);
        // assert isRed(h) && !isRed(h.right) && !isRed(h.right.left);
        flipColors(h);
        if (isRed(h.left.left)) {
            h = rotateRight(h);
            flipColors(h);
        }
        return h;
    }

    // restore red-black tree invariant
    private Node balance(Node h) {
```

```
        // assert (h != null);

        if (isRed(h.right) && !isRed(h.left))    h = rotateLeft(h);
        if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
        if (isRed(h.left) && isRed(h.right))     flipColors(h);

        h.size = size(h.left) + size(h.right) + 1;
        return h;
    }

    public int height() {
        return height(root);
    }
    private int height(Node x) {
        if (x == null) return -1;
        return 1 + Math.max(height(x.left), height(x.right));
    }

    public Key min() {
        if (isEmpty()) throw new NoSuchElementException("calls min() with empty symbol table");
        return min(root).key;
    }

    // the smallest key in subtree rooted at x; null if no such key
    private Node min(Node x) {
        // assert x != null;
        if (x.left == null) return x;
        else                return min(x.left);
    }

    /**
     * Returns the largest key in the symbol table.
     * @return the largest key in the symbol table
     * @throws NoSuchElementException if the symbol table is empty
     */
    public Key max() {
        if (isEmpty()) throw new NoSuchElementException("calls max() with empty symbol table");
        return max(root).key;
    }

    // the largest key in the subtree rooted at x; null if no such key
    private Node max(Node x) {
        // assert x != null;
        if (x.right == null) return x;
        else                 return max(x.right);
    }

    public Key floor(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to floor() is null");
```

```java
        if (isEmpty()) throw new NoSuchElementException("calls floor() with empty symbol table");
        Node x = floor(root, key);
        if (x == null) throw new NoSuchElementException("argument to floor() is too small");
        else           return x.key;
    }

    // the largest key in the subtree rooted at x less than or equal to the given key
    private Node floor(Node x, Key key) {
        if (x == null) return null;
        int cmp = key.compareTo(x.key);
        if (cmp == 0) return x;
        if (cmp < 0)  return floor(x.left, key);
        Node t = floor(x.right, key);
        if (t != null) return t;
        else           return x;
    }

    public Key ceiling(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to ceiling() is null");
        if (isEmpty()) throw new NoSuchElementException("calls ceiling() with empty symbol table");
        Node x = ceiling(root, key);
        if (x == null) throw new NoSuchElementException("argument to ceiling() is too large");
        else           return x.key;
    }

    // the smallest key in the subtree rooted at x greater than or equal to the given key
    private Node ceiling(Node x, Key key) {
        if (x == null) return null;
        int cmp = key.compareTo(x.key);
        if (cmp == 0) return x;
        if (cmp > 0)  return ceiling(x.right, key);
        Node t = ceiling(x.left, key);
        if (t != null) return t;
        else           return x;
    }

    public Key select(int rank) {
        if (rank < 0 || rank >= size()) {
            throw new IllegalArgumentException("argument to select() is invalid: " + rank);
        }
        return select(root, rank);
    }

    // Return key in BST rooted at x of given rank.
```

```java
    // Precondition: rank is in legal range.
    private Key select(Node x, int rank) {
        if (x == null) return null;
        int leftSize = size(x.left);
        if      (leftSize > rank) return select(x.left,  rank);
        else if (leftSize < rank) return select(x.right, rank - leftSize - 1);
        else                      return x.key;
    }

    public int rank(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to rank() is null");
        return rank(key, root);
    }

    // number of keys less than key in the subtree rooted at x
    private int rank(Key key, Node x) {
        if (x == null) return 0;
        int cmp = key.compareTo(x.key);
        if      (cmp < 0) return rank(key, x.left);
        else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
        else              return size(x.left);
    }

    public Iterable<Key> keys() {
        if (isEmpty()) return new Queue<Key>();
        return keys(min(), max());
    }

    public Iterable<Key> keys(Key lo, Key hi) {
        if (lo == null) throw new IllegalArgumentException("first argument to keys() is null");
        if (hi == null) throw new IllegalArgumentException("second argument to keys() is null");

        Queue<Key> queue = new Queue<Key>();
        // if (isEmpty() || lo.compareTo(hi) > 0) return queue;
        keys(root, queue, lo, hi);
        return queue;
    }

    // add the keys between lo and hi in the subtree rooted at x
    // to the queue
    private void keys(Node x, Queue<Key> queue, Key lo, Key hi) {
        if (x == null) return;
        int cmplo = lo.compareTo(x.key);
        int cmphi = hi.compareTo(x.key);
        if (cmplo < 0) keys(x.left, queue, lo, hi);
        if (cmplo <= 0 && cmphi >= 0) queue.enqueue(x.key);
        if (cmphi > 0) keys(x.right, queue, lo, hi);
    }
```

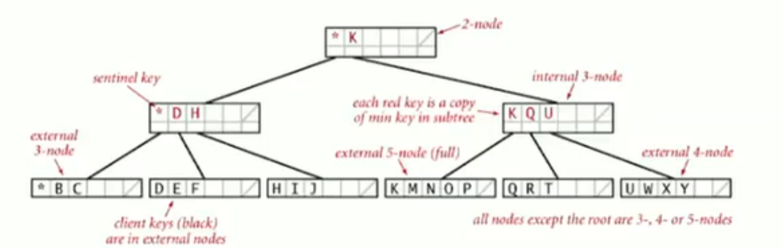
```java
    public int size(Key lo, Key hi) {
        if (lo == null) throw new IllegalArgumentException("first argument to size() is null");
        if (hi == null) throw new IllegalArgumentException("second argument to size() is null");

        if (lo.compareTo(hi) > 0) return 0;
        if (contains(hi)) return rank(hi) - rank(lo) + 1;
        else              return rank(hi) - rank(lo);
    }

    private boolean check() {
        if (!isBST())            StdOut.println("Not in symmetric order");
        if (!isSizeConsistent()) StdOut.println("Subtree counts not consistent");
        if (!isRankConsistent()) StdOut.println("Ranks not consistent");
        if (!is23())             StdOut.println("Not a 2-3 tree");
        if (!isBalanced())       StdOut.println("Not balanced");
        return isBST() && isSizeConsistent() && isRankConsistent() && is23() && isBalanced();
    }

    // does this binary tree satisfy symmetric order?
    // Note: this test also ensures that data structure is a binary tree since order is strict
    private boolean isBST() {
        return isBST(root, null, null);
    }

    // is the tree rooted at x a BST with all keys strictly between min and max
    // (if min or max is null, treat as empty constraint)
    // Credit: elegant solution due to Bob Dondero
    private boolean isBST(Node x, Key min, Key max) {
        if (x == null) return true;
        if (min != null && x.key.compareTo(min) <= 0) return false;
        if (max != null && x.key.compareTo(max) >= 0) return false;
        return isBST(x.left, min, x.key) && isBST(x.right, x.key, max);
    }

    // are the size fields correct?
    private boolean isSizeConsistent() { return isSizeConsistent(root); }
    private boolean isSizeConsistent(Node x) {
        if (x == null) return true;
        if (x.size != size(x.left) + size(x.right) + 1) return false;
        return isSizeConsistent(x.left) && isSizeConsistent(x.right);
    }

    // check that ranks are consistent
    private boolean isRankConsistent() {
        for (int i = 0; i < size(); i++)
            if (i != rank(select(i))) return false;
```

```java
        for (Key key : keys())
            if (key.compareTo(select(rank(key))) != 0) return false;
        return true;
    }

    // Does the tree have no red right links, and at most one (left)
    // red links in a row on any path?
    private boolean is23() { return is23(root); }
    private boolean is23(Node x) {
        if (x == null) return true;
        if (isRed(x.right)) return false;
        if (x != root && isRed(x) && isRed(x.left))
            return false;
        return is23(x.left) && is23(x.right);
    }

    // do all paths from root to leaf have same number of black edges?
    private boolean isBalanced() {
        int black = 0;     // number of black links on path from root to min
        Node x = root;
        while (x != null) {
            if (!isRed(x)) black++;
            x = x.left;
        }
        return isBalanced(root, black);
    }

    // does every path from the root to a leaf have the given number of black links?
    private boolean isBalanced(Node x, int black) {
        if (x == null) return black == 0;
        if (!isRed(x)) black--;
        return isBalanced(x.left, black) && isBalanced(x.right, black);
    }

    public static void main(String[] args) {
        RedBlackBST<String, Integer> st = new RedBlackBST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }
        StdOut.println();
        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));
        StdOut.println();
    }
}
```

B-trees

B-tree

- generalize 2-3 trees by allowing up to M-1 ley-link pairs per node
- at least 2 key-link pairs at root
- as least M/2 key-link pairs in other nodes
- external nodes contain client keys
- internal nodes contain copies of keys to guide search



Anatomy of a B-tree set (M = 6)

File system model