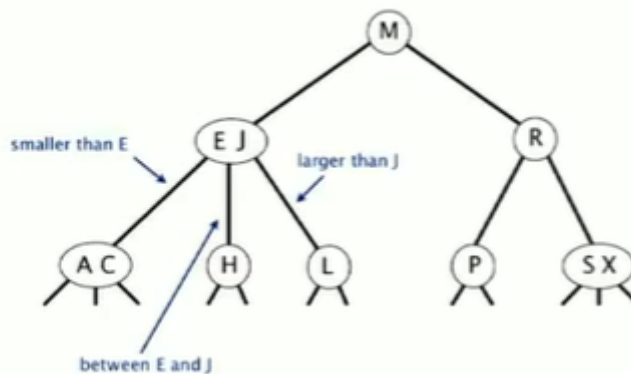


Balanced Search Trees

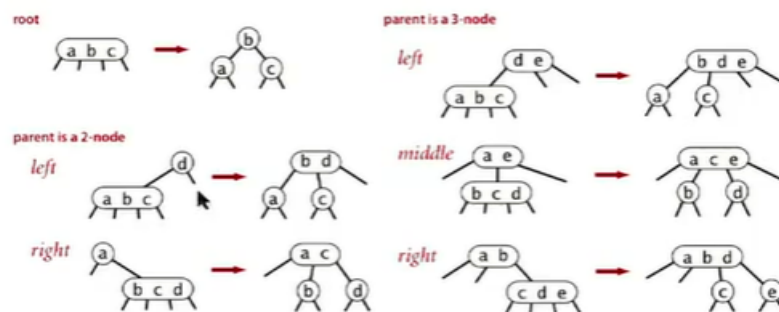
Search Trees

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--- $\log N$

- 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()

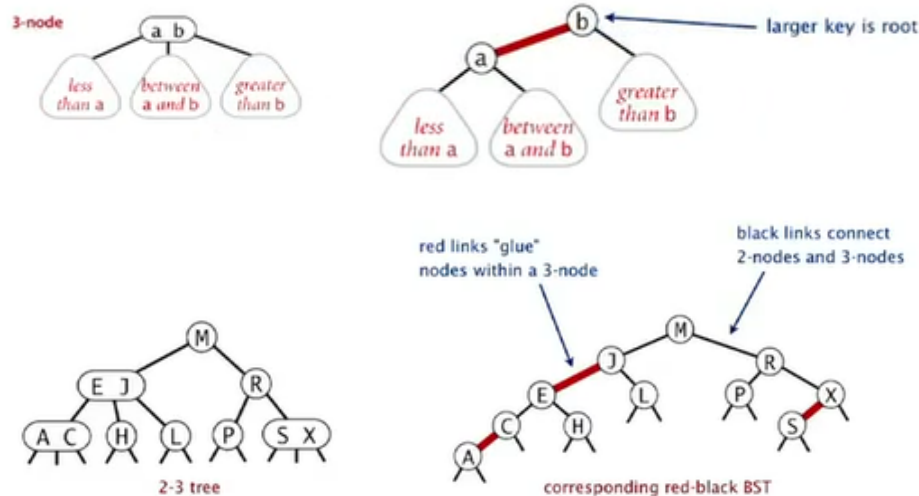
Red-black BSTs

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

```

1  import java.util.NoSuchElementException;
2
3  public class RedBlackBST<Key extends Comparable<Key>, Value> {
4
5      private static final boolean RED    = true;
6      private static final boolean BLACK = false;
7
8      private Node root;    // root of the BST
9
10     // BST helper node data type
11     private class Node {
12         private Key key;    // key
13         private Value val;  // associated data
14         private Node left, right; // links to left and right subtrees
15         private boolean color; // color of parent link
16         private int size;    // subtree count
17
18         public Node(Key key, Value val, boolean color, int size) {
19             this.key = key;
20             this.val = val;
21             this.color = color;
22             this.size = size;
23         }
24     }
25
26     public RedBlackBST() {
27     }
28
29     private boolean isRed(Node x) {
30         if (x == null) return false;
31         return x.color == RED;
32     }
33
34     private int size(Node x) {
35         if (x == null) return 0;
36         return x.size;
37     }
38

```

```

39     public int size() {
40         return size(root);
41     }
42
43     public boolean isEmpty() {
44         return root == null;
45     }
46
47     public Value get(Key key) {
48         if (key == null) throw new IllegalArgumentException("argument to get()
is null");
49         return get(root, key);
50     }
51
52     private Value get(Node x, Key key) {
53         while (x != null) {
54             int cmp = key.compareTo(x.key);
55             if (cmp < 0) x = x.left;
56             else if (cmp > 0) x = x.right;
57             else return x.val;
58         }
59         return null;
60     }
61
62     public boolean contains(Key key) {
63         return get(key) != null;
64     }
65
66     public void put(Key key, Value val) {
67         if (key == null) throw new IllegalArgumentException("first argument to
put() is null");
68         if (val == null) {
69             delete(key);
70             return;
71         }
72
73         root = put(root, key, val);
74         root.color = BLACK;
75     }
76
77     private Node put(Node h, Key key, Value val) {
78         if (h == null) return new Node(key, val, RED, 1);
79
80         int cmp = key.compareTo(h.key);
81         if (cmp < 0) h.left = put(h.left, key, val);
82         else if (cmp > 0) h.right = put(h.right, key, val);
83         else h.val = val;
84
85         // fix-up any right-leaning links
86         if (isRed(h.right) && !isRed(h.left)) h = rotateLeft(h);
87         if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
88         if (isRed(h.left) && isRed(h.right)) flipColors(h);

```

```

89         h.size = size(h.left) + size(h.right) + 1;
90
91         return h;
92     }
93
94     public void deleteMin() {
95         if (isEmpty()) throw new NoSuchElementException("BST underflow");
96
97         // if both children of root are black, set root to red
98         if (!isRed(root.left) && !isRed(root.right))
99             root.color = RED;
100
101         root = deleteMin(root);
102         if (!isEmpty()) root.color = BLACK;
103         // assert check();
104     }
105
106     // delete the key-value pair with the minimum key rooted at h
107     private Node deleteMin(Node h) {
108         if (h.left == null)
109             return null;
110
111         if (!isRed(h.left) && !isRed(h.left.left))
112             h = moveRedLeft(h);
113
114         h.left = deleteMin(h.left);
115         return balance(h);
116
117     public void deleteMax() {
118         if (isEmpty()) throw new NoSuchElementException("BST underflow");
119
120         // if both children of root are black, set root to red
121         if (!isRed(root.left) && !isRed(root.right))
122             root.color = RED;
123
124         root = deleteMax(root);
125         if (!isEmpty()) root.color = BLACK;
126     }
127
128     // delete the key-value pair with the maximum key rooted at h
129     private Node deleteMax(Node h) {
130         if (isRed(h.left))
131             h = rotateRight(h);
132
133         if (h.right == null)
134             return null;
135
136         if (!isRed(h.right) && !isRed(h.right.left))
137             h = moveRedRight(h);
138
139         h.right = deleteMax(h.right);
140

```

```

141         return balance(h);
142     }
143
144     public void delete(Key key) {
145         if (key == null) throw new IllegalArgumentException("argument to
delete() is null");
146         if (!contains(key)) return;
147
148         // if both children of root are black, set root to red
149         if (!isRed(root.left) && !isRed(root.right))
150             root.color = RED;
151
152         root = delete(root, key);
153         if (!isEmpty()) root.color = BLACK;
154         // assert check();
155     }
156
157     private Node delete(Node h, Key key) {
158         // assert get(h, key) != null;
159
160         if (key.compareTo(h.key) < 0) {
161             if (!isRed(h.left) && !isRed(h.left.left))
162                 h = moveRedLeft(h);
163             h.left = delete(h.left, key);
164         }
165         else {
166             if (isRed(h.left))
167                 h = rotateRight(h);
168             if (key.compareTo(h.key) == 0 && (h.right == null))
169                 return null;
170             if (!isRed(h.right) && !isRed(h.right.left))
171                 h = moveRedRight(h);
172             if (key.compareTo(h.key) == 0) {
173                 Node x = min(h.right);
174                 h.key = x.key;
175                 h.val = x.val;
176                 h.right = deleteMin(h.right);
177             }
178             else h.right = delete(h.right, key);
179         }
180         return balance(h);
181     }
182
183     private Node rotateRight(Node h) {
184         assert (h != null) && isRed(h.left);
185         Node x = h.left;
186         h.left = x.right;
187         x.right = h;
188         x.color = h.color;
189         h.color = RED;
190         x.size = h.size;
191         h.size = size(h.left) + size(h.right) + 1;

```

```

192     return x;
193 }
194
195 // make a right-leaning link lean to the left
196 private Node rotateLeft(Node h) {
197     assert (h != null) && isRed(h.right);
198     // assert (h != null) && isRed(h.right) && !isRed(h.left); // for
insertion only
199     Node x = h.right;
200     h.right = x.left;
201     x.left = h;
202     x.color = h.color;
203     h.color = RED;
204     x.size = h.size;
205     h.size = size(h.left) + size(h.right) + 1;
206     return x;
207 }
208
209 // flip the colors of a node and its two children
210 private void flipColors(Node h) {
211
212     h.color = !h.color;
213     h.left.color = !h.left.color;
214     h.right.color = !h.right.color;
215 }
216
217 // Assuming that h is red and both h.left and h.left.left
218 // are black, make h.left or one of its children red.
219 private Node moveRedLeft(Node h) {
220
221     flipColors(h);
222     if (isRed(h.right.left)) {
223         h.right = rotateRight(h.right);
224         h = rotateLeft(h);
225         flipColors(h);
226     }
227     return h;
228 }
229
230 private Node moveRedRight(Node h) {
231     // assert (h != null);
232     // assert isRed(h) && !isRed(h.right) && !isRed(h.right.left);
233     flipColors(h);
234     if (isRed(h.left.left)) {
235         h = rotateRight(h);
236         flipColors(h);
237     }
238     return h;
239 }
240
241 // restore red-black tree invariant
242 private Node balance(Node h) {

```

```

243         // assert (h != null);
244
245         if (isRed(h.right) && !isRed(h.left))    h = rotateLeft(h);
246         if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
247         if (isRed(h.left) && isRed(h.right))     flipColors(h);
248
249         h.size = size(h.left) + size(h.right) + 1;
250         return h;
251     }
252
253     public int height() {
254         return height(root);
255     }
256     private int height(Node x) {
257         if (x == null) return -1;
258         return 1 + Math.max(height(x.left), height(x.right));
259     }
260
261     public Key min() {
262         if (isEmpty()) throw new NoSuchElementException("calls min() with
empty symbol table");
263         return min(root).key;
264     }
265
266     // the smallest key in subtree rooted at x; null if no such key
267     private Node min(Node x) {
268         // assert x != null;
269         if (x.left == null) return x;
270         else                 return min(x.left);
271     }
272
273     /**
274      * Returns the largest key in the symbol table.
275      * @return the largest key in the symbol table
276      * @throws NoSuchElementException if the symbol table is empty
277      */
278     public Key max() {
279         if (isEmpty()) throw new NoSuchElementException("calls max() with
empty symbol table");
280         return max(root).key;
281     }
282
283     // the largest key in the subtree rooted at x; null if no such key
284     private Node max(Node x) {
285         // assert x != null;
286         if (x.right == null) return x;
287         else                 return max(x.right);
288     }
289
290     public Key floor(Key key) {
291         if (key == null) throw new IllegalArgumentException("argument to
floor() is null");

```



```

292         if (isEmpty()) throw new NoSuchElementException("calls floor() with
empty symbol table");
293         Node x = floor(root, key);
294         if (x == null) throw new NoSuchElementException("argument to floor()
is too small");
295         else return x.key;
296     }
297
298     // the largest key in the subtree rooted at x less than or equal to the
given key
299     private Node floor(Node x, Key key) {
300         if (x == null) return null;
301         int cmp = key.compareTo(x.key);
302         if (cmp == 0) return x;
303         if (cmp < 0) return floor(x.left, key);
304         Node t = floor(x.right, key);
305         if (t != null) return t;
306         else return x;
307     }
308
309     public Key ceiling(Key key) {
310         if (key == null) throw new IllegalArgumentException("argument to
ceiling() is null");
311         if (isEmpty()) throw new NoSuchElementException("calls ceiling() with
empty symbol table");
312         Node x = ceiling(root, key);
313         if (x == null) throw new NoSuchElementException("argument to ceiling()
is too large");
314         else return x.key;
315     }
316
317     // the smallest key in the subtree rooted at x greater than or equal to
the given key
318     private Node ceiling(Node x, Key key) {
319         if (x == null) return null;
320         int cmp = key.compareTo(x.key);
321         if (cmp == 0) return x;
322         if (cmp > 0) return ceiling(x.right, key);
323         Node t = ceiling(x.left, key);
324         if (t != null) return t;
325         else return x;
326     }
327
328     public Key select(int rank) {
329         if (rank < 0 || rank >= size()) {
330             throw new IllegalArgumentException("argument to select() is
invalid: " + rank);
331         }
332         return select(root, rank);
333     }
334
335     // Return key in BST rooted at x of given rank.

```

```

336 // Precondition: rank is in legal range.
337 private Key select(Node x, int rank) {
338     if (x == null) return null;
339     int leftSize = size(x.left);
340     if (leftSize > rank) return select(x.left, rank);
341     else if (leftSize < rank) return select(x.right, rank - leftSize - 1);
342     else return x.key;
343 }
344
345 public int rank(Key key) {
346     if (key == null) throw new IllegalArgumentException("argument to
rank() is null");
347     return rank(key, root);
348 }
349
350 // number of keys less than key in the subtree rooted at x
351 private int rank(Key key, Node x) {
352     if (x == null) return 0;
353     int cmp = key.compareTo(x.key);
354     if (cmp < 0) return rank(key, x.left);
355     else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
356     else return size(x.left);
357 }
358
359 public Iterable<Key> keys() {
360     if (isEmpty()) return new Queue<Key>();
361     return keys(min(), max());
362 }
363
364 public Iterable<Key> keys(Key lo, Key hi) {
365     if (lo == null) throw new IllegalArgumentException("first argument to
keys() is null");
366     if (hi == null) throw new IllegalArgumentException("second argument to
keys() is null");
367
368     Queue<Key> queue = new Queue<Key>();
369     // if (isEmpty() || lo.compareTo(hi) > 0) return queue;
370     keys(root, queue, lo, hi);
371     return queue;
372 }
373
374 // add the keys between lo and hi in the subtree rooted at x
375 // to the queue
376 private void keys(Node x, Queue<Key> queue, Key lo, Key hi) {
377     if (x == null) return;
378     int cmplo = lo.compareTo(x.key);
379     int cmphi = hi.compareTo(x.key);
380     if (cmplo < 0) keys(x.left, queue, lo, hi);
381     if (cmplo <= 0 && cmphi >= 0) queue.enqueue(x.key);
382     if (cmphi > 0) keys(x.right, queue, lo, hi);
383 }
384

```

```

385     public int size(Key lo, Key hi) {
386         if (lo == null) throw new IllegalArgumentException("first argument to
size() is null");
387         if (hi == null) throw new IllegalArgumentException("second argument to
size() is null");
388
389         if (lo.compareTo(hi) > 0) return 0;
390         if (contains(hi)) return rank(hi) - rank(lo) + 1;
391         else return rank(hi) - rank(lo);
392     }
393
394     private boolean check() {
395         if (!isBST()) StdOut.println("Not in symmetric order");
396         if (!isSizeConsistent()) StdOut.println("Subtree counts not
consistent");
397         if (!isRankConsistent()) StdOut.println("Ranks not consistent");
398         if (!is23()) StdOut.println("Not a 2-3 tree");
399         if (!isBalanced()) StdOut.println("Not balanced");
400         return isBST() && isSizeConsistent() && isRankConsistent() && is23()
&& isBalanced();
401     }
402
403     // does this binary tree satisfy symmetric order?
404     // Note: this test also ensures that data structure is a binary tree since
order is strict
405     private boolean isBST() {
406         return isBST(root, null, null);
407     }
408
409     // is the tree rooted at x a BST with all keys strictly between min and
max
410     // (if min or max is null, treat as empty constraint)
411     // Credit: elegant solution due to Bob Dondero
412     private boolean isBST(Node x, Key min, Key max) {
413         if (x == null) return true;
414         if (min != null && x.key.compareTo(min) <= 0) return false;
415         if (max != null && x.key.compareTo(max) >= 0) return false;
416         return isBST(x.left, min, x.key) && isBST(x.right, x.key, max);
417     }
418
419     // are the size fields correct?
420     private boolean isSizeConsistent() { return isSizeConsistent(root); }
421     private boolean isSizeConsistent(Node x) {
422         if (x == null) return true;
423         if (x.size != size(x.left) + size(x.right) + 1) return false;
424         return isSizeConsistent(x.left) && isSizeConsistent(x.right);
425     }
426
427     // check that ranks are consistent
428     private boolean isRankConsistent() {
429         for (int i = 0; i < size(); i++)
430             if (i != rank(select(i))) return false;

```

```

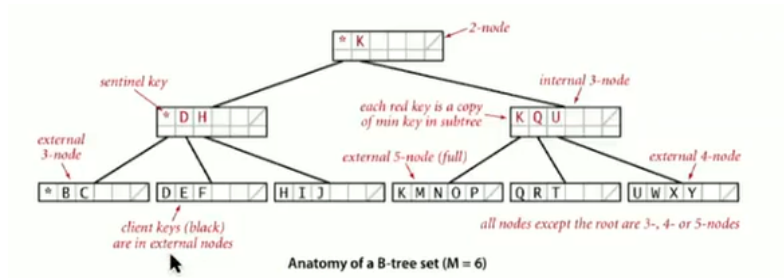
431         for (Key key : keys())
432             if (key.compareTo(select(rank(key))) != 0) return false;
433         return true;
434     }
435
436     // Does the tree have no red right links, and at most one (left)
437     // red links in a row on any path?
438     private boolean is23() { return is23(root); }
439     private boolean is23(Node x) {
440         if (x == null) return true;
441         if (isRed(x.right)) return false;
442         if (x != root && isRed(x) && isRed(x.left))
443             return false;
444         return is23(x.left) && is23(x.right);
445     }
446
447     // do all paths from root to leaf have same number of black edges?
448     private boolean isBalanced() {
449         int black = 0;        // number of black links on path from root to min
450         Node x = root;
451         while (x != null) {
452             if (!isRed(x)) black++;
453             x = x.left;
454         }
455         return isBalanced(root, black);
456     }
457
458     // does every path from the root to a leaf have the given number of black
459     links?
460     private boolean isBalanced(Node x, int black) {
461         if (x == null) return black == 0;
462         if (!isRed(x)) black--;
463         return isBalanced(x.left, black) && isBalanced(x.right, black);
464     }
465
466     public static void main(String[] args) {
467         RedBlackBST<String, Integer> st = new RedBlackBST<String, Integer>();
468         for (int i = 0; !StdIn.isEmpty(); i++) {
469             String key = StdIn.readString();
470             st.put(key, i);
471         }
472         StdOut.println();
473         for (String s : st.keys())
474             StdOut.println(s + " " + st.get(s));
475         StdOut.println();
476     }

```

B-trees

B-tree

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



File system model