Let's briefly review insert and delete methods.

Then we'll do some example insertions and deletions to help you get a better feel for how it works.
Insertion Overview

First do standard binary search tree insertion, making new node red, having extraRed reference this new node.

```java
def insertFixUp(BSTNode extraRed):
    while (extraRed != root && ((RBNode) extraRed.parent).isRed()):
        if (((RBNode) extraRed.grandparent()).recolorRed()):
            extraRed = extraRed.grandparent();
        else:
            if (extraRed.isLeftChild() != extraRed.parent.isLeftChild()):
                extraRed = liftUp(extraRed);
                ((RBNode) extraRed.parent).setBlack();
                ((RBNode) extraRed.grandparent()).setRed();
                extraRed = liftUp(extraRed.parent);
    ((RBNode) root).setBlack();
```
Overview deletion

Recall that y is node being deleted (successor of original node to delete had 2 children).

doubleBlack is initialized to reference child of y (or Frontier/null if y is leaf)

```java
void deleteFixUp(RBNode doubleBlack) {
    // called on a node that has an "double" black
    while (doubleBlack != root && doubleBlack.isBlack()) {
        // stop if at root or red node
        USE rotations/recoloring to move doubleBlack up towards root
    }
    doubleBlack.setBlack(); // used when loop terminates with a red node as doubleBlack
}
```
red (or root) when loop exits
Analysis of time complexity for Red-Black tree methods

\[ h \leq 2 \log_2 (n+1) = O(\log n) \]

- add, delete, search, succ, pred,
- max, min take \( O(\log n) \)
- in-order traversal \( O(n) \)
- pred/succ/search followed by iterating \( k \) times \( O(\log n + k) \)
We now move to the next Ordered Collection ADT that we'll be studying. Start with motivation.
B-tree Design

Data Structure for Ordered Collection

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>approximate access time (ns)</th>
<th>cost per megabyte ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache</td>
<td>5-20</td>
<td>10-75</td>
</tr>
<tr>
<td>main memory</td>
<td>60-120</td>
<td>0.50-5.00</td>
</tr>
<tr>
<td>secondary storage</td>
<td>20,000,000</td>
<td>.001-0.10</td>
</tr>
</tbody>
</table>

Table 29.1 Approximate access times in nanoseconds (ns) as compared with the cost for cache, main memory, and secondary storage (disk).

Designed for situation in which n (# elements or tags in a tagged collection) is so large that B-tree + n references to elements cannot fit in main memory.

Data is moved from disk to RAM in chunk called page.
How to best group portions of tree into pages to minimize the # pages that are fetched in worst case?

What if we could fit 7 nodes per disk page?

not very good it's possible only

since the root is looked at
Page Fault

Virtual memory lets you store data on secondary storage (hard drive) and act as if it's in main memory.

If a program accesses data not in main memory, it's a page fault.
Binary Search Tree

B-tree

Max of \(2t-1\) tags

Abstract Search Tree

Internally

Sorted array of \(2t-1\) tags

Array of \(2t-1\) data references (disk page id)

Array of \(2t\) children (disk page id)
B-tree Properties

order of B-tree $t$  

1) Completely balanced (every path from root to a leaf has the same # of nodes)

2) Every node (except root) has $\geq t$ children (means $\geq t-1$ keys)

3) Every node has $\leq 2t$ children (means $\leq 2t-1$ keys)
Split

2t-1 element  median  Full node

2t children

Join the parent

\( t \)
$t=2$ in this example
2, 3 or 4 children
1, 2 or 3 elements

Full node

Top-down insertion

ABC

Split

A

CDE

B

A

C

EFG

BDF

ACE

GHI
Insert J

Step 1

Step 2

height \approx \log_{10} N