Insertion

Inserting a new node into a red-black tree starts off with a typical binary search tree insertion.

- New nodes are inserted at a leaf, which means that they replace a black node.
- New nodes are always colored red and are given two black leaf (NULL) nodes.

Once we've done that, we fix any resulting red-black property violations. We have two possible violations:

- Red violations: A red node has a red child (or the root is red).
- Black violations: One path has more blacks than another path.

The node inserted is red. We didn't change the number of black nodes on any path to a leaf, so we know that we won't have a black violation. However, we might have a red violation.

In the special case that where the root is red, we can always just turn it black to satisfy property 2, without violating the other constraints.

Otherwise, if there's a red violation, then this means that we have a red node under another red node. Oops!

Let's call N the current node. P is N's parent. G is N's grandparent. U is N's uncle and P's sibling. We know that:

- N is red and P is red, since we have a red violation.
- G is definitely black, since we didn't *previously* have a red violation.

The unknown parts are:

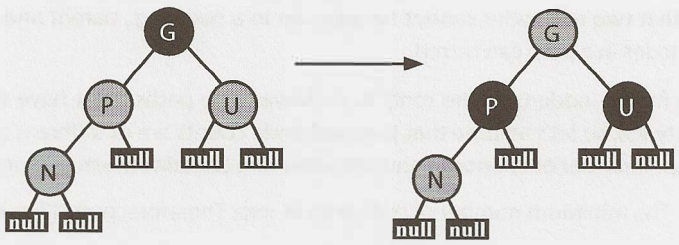
- U could be either red or black.
- U could be either a left or right child.
- N could be either a left or right child.

By simple combinatorics, that's eight cases to consider. Fortunately some of these cases will be equivalent.

- **Case 1: U is red.**

  It doesn't matter whether U is a left or right child, nor whether P is a left or right child. We can merge four of our eight cases into one.

  If U is red, we can just toggle the colors of P, U, and G. Flip G from black to red. Flip P and U from red to black. We haven't changed the number of black nodes in any path.



  However, by making G red, we might have created a red violation with G's parent. If so, we recursively apply the full logic to handle a red violation, where this G becomes the new N.

  Note that in the general recursive case, N, P, and U may also have subtrees in place of each black NULL (the leaves shown). In Case 1, these subtrees stay attached to the same parents, as the tree structure remains unchanged.

- **Case 2: U is black.**

  We'll need to consider the configurations (left vs. right child) of N and U. In each case, our goal is to fix up the red violation (red on top of red) without:

  » Messing up the ordering of the binary search tree.

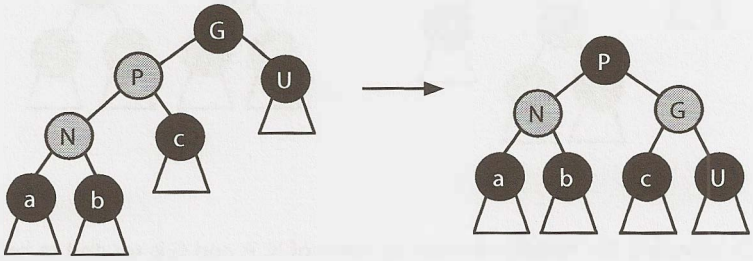
  » *Introducing a black violation (more black nodes on one path than another).*

  If we can do this, we're good. In each of the cases below, the red violation is fixed with rotations that maintain the node ordering.

  Further, the below rotations maintain the exact number of black nodes in each path through the affected portion of the tree that were in place beforehand. The children of the rotating section are either NULL leaves or subtrees that remain internally unchanged.

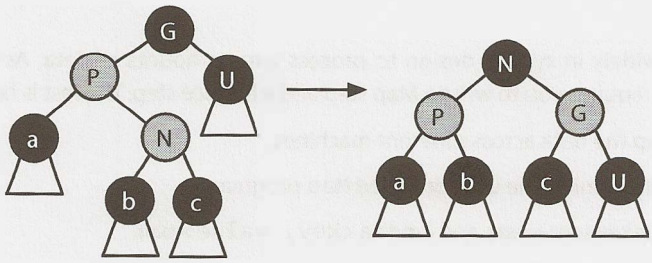
  Case A: N and P are both left children.

  We resolve the red violation with the rotation of N, P, and G and the associated recoloring shown below. If you picture the in-order traversal, you can see the rotation maintains the node ordering (a <= N <= b <= P <= c <= G <= U). The tree maintains the same, equal number of black nodes in the path down to each subtree a, b, c, and U (which may all be NULL) .



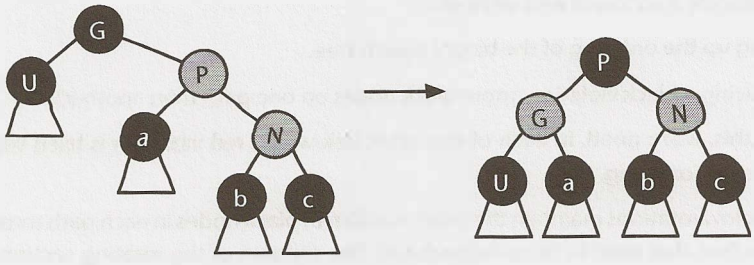
  Case B: P is a left child, and N is a right child.

  The rotations in Case B resolve the red violation and maintain the in-order property: a <= P <= b <= N <= c <= G <= U. Again, the count of the black nodes remains constant in each path down to the leaves (or subtrees).



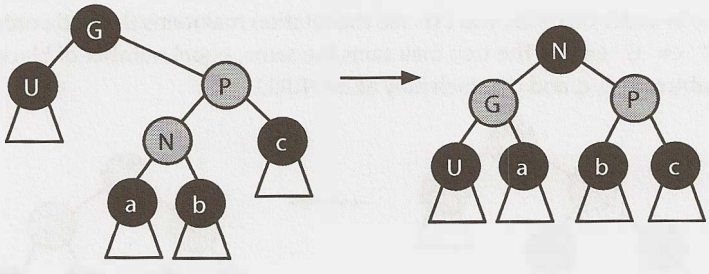
Case C: N and P are both right children.

This is a mirror image of case A.



Case D: N is a left child, and P is a right child.

This is a mirror image of case B.



In each of Case 2's subcases, the middle element by value of N, P, and G is rotated to become the root of what was G's subtree, and that element and G swap colors.

That said, do not try to just memorize these cases. Rather, study why they work. How does each one ensure no red violations, no black violations, and no violations of the binary search tree property?

▸ MapReduce

MapReduce is used widely in system design to process large amounts of data. As its name suggests, a MapReduce program requires you to write a Map step and a Reduce step. The rest is handled by the system.

1. The system splits up the data across different machines.
2. Each machine starts running the user-provided `Map` program.
3. The `Map` program takes some data and emits a `<key, value>` pair.
4. The system-provided `Shuffle` process reorganizes the data so that all <key, value> pairs associated with a given key go to the same machine, to be processed by `Reduce`.
5. The user-provided `Reduce` program takes a key and a set of associated values and "reduces" them in some way, emitting a new key and value. The results of this might be fed back into the `Reduce` program for more reducing.

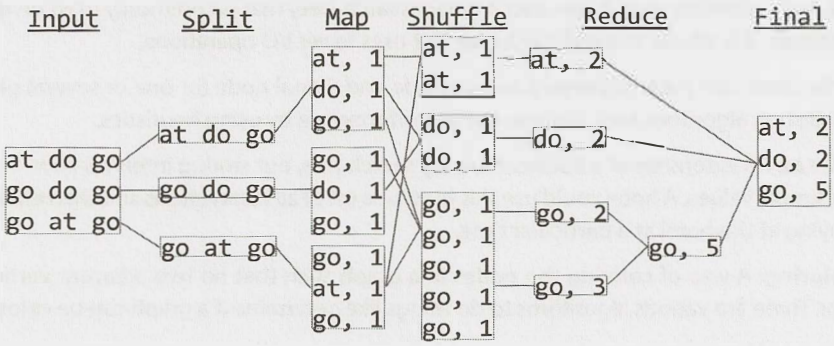
The typical example of using MapReduce—basically the "Hello World" of MapReduce—is counting the frequency of words within a set of documents.

Of course, you could write this as a single function that reads in all the data, counts the number of times each word appears via a hash table, and then outputs the result.

MapReduce allows you to process the document in parallel. The `Map` function reads in a document and emits just each individual word and the count (which is always 1). The `Reduce` function reads in keys (words) and associated values (counts). It emits the sum of the counts. This sum could possibly wind up as input for another call to `Reduce` on the same key (as shown in the diagram).

```
1  void map(String name, String document):
2    for each word w in document:
3      emit(w, 1)
4
5  void reduce(String word, Iterator partialCounts):
6    int sum = 0
7    for each count in partialCounts:
8      sum += count
9    emit(word, sum)
```

The diagram below shows how this might work on this example.



Here's another example: You have a list of data in the form {City, Temperature, Date}. Calculate the average temperature in each city every year. For example {(2012, Philadelphia, 58.2), (2011, Philadelphia, 56.6), (2012, Seattle, 45.1)}.

- **Map**: The Map step outputs a key value pair where the key is `City_Year` and the value is `(Temperature, 1)`. The '1' reflects that this is the average temperature out of one data point. This will be important for the `Reduce` step.
- **Reduce**: The Reduce step will be given a list of temperatures that correspond with a particular city and year. It must use these to compute the average temperature for this input. You cannot simply add up the temperatures and divide by the number of values.

  To see this, imagine we have five data points for a particular city and year: 25, 100, 75, 85, 50. The Reduce step might only get some of this data at once. If you averaged {75, 85} you would get 80. This might end up being input for another Reduce step with 50, and it would be a mistake to just naively average 80 and 50. The 80 has more weight.

  Therefore, our Reduce step instead takes in {(80, 2), (50, 1)}, then sums the *weighted* temperatures. So it does 80 * 2 + 50 * 1 and then divides by (2 + 1) to get an average temperature of 70. It then emits (70, 3).

  Another Reduce step might reduce {(25, 1), (100, 1)} to get (62.5, 2). If we reduce this with (70, 3) we get the final answer: (67, 5). In other words, the average temperature in this city for this year was 67 degrees.

We could do this in other ways, too. We could have just the city as the key, and the value be (Year, Temperature, Count). The Reduce step would do essentially the same thing, but would have to group by Year itself.

In many cases, it's useful to think about what the `Reduce` step should do first, and then design the `Map` step around that. What data does `Reduce` need to have to do its job?

Additional Studying

So, you've mastered this material and you want to learn even more? Okay. Here are some topics to get you started:

- **Bellman-Ford Algorithm**: Finds the shortest paths from a single node in a weighted directed graph with positive and negative edges.
- **Floyd-Warshall Algorithm**: Finds the shortest paths in a weighted graph with positive or negative weight edges (but no negative weight cycles).
- **Minimum Spanning Trees**: In a weighted, connected, undirected graph, a spanning tree is a tree that connects all the vertices. The minimum spanning tree is the spanning tree with minimum weight. There are various algorithms to do this.
- **B-Trees:** A self-balancing search tree (not a binary search tree) that is commonly used on disks or other storage devices. It is similar to a red-black tree, but uses fewer I/O operations.
- **A***: Find the least-cost path between a source node and a goal node (or one of several goal nodes). It extends Dijkstra's algorithm and achieves better performance by using heuristics.
- **Interval Trees**: An extension of a balanced binary search tree, but storing intervals (low -> high ranges) instead of simple values. A hotel could use this to store a list of all reservations and then efficiently detect who is staying at the hotel at a particular time.
- **Graph coloring**: A way of coloring the nodes in a graph such that no two adjacent vertices have the same color. There are various algorithms to do things like determine if a graph can be colored with only K colors.
- **P, NP, and NP-Complete**: P, NP, and NP-Complete refer to classes of problems. P problems are problems that can be quickly solved (where "quickly" means polynomial time). NP problems are those where, given a solution, the solution can be quickly verified. NP-Complete problems are a subset of NP problems that can all be reduced to each other (that is, if you found a solution to one problem, you could tweak the solution to solve other problems in the set in polynomial time).

  It is an open (and very famous) question whether P = NP, but the answer is generally believed to be no.
- **Combinatorics and Probability**: There are various things you can learn about here, such as random variables, expected value, and n-choose-k.
- **Bipartite Graph**: A bipartite graph is a graph where you can divide its nodes into two sets such that every edge stretches across the two sets (that is, there is never an edge between two nodes in the same set). There is an algorithm to check if a graph is a bipartite graph. Note that a bipartite graph is equivalent to a graph that can be colored with two colors.
- **Regular Expressions**: You should know that regular expressions exist and what they can be used for (roughly). You can also learn about how an algorithm to match regular expressions would work. Some of the basic syntax behind regular expressions could be useful as well.

There is of course a great deal more to data structures and algorithms. If you're interested in exploring these topics more deeply, I recommend picking up the hefty *Introduction to Algorithms* ("CLRS" by Cormen, Leiserson, Rivest and Stein) or *The Algorithm Design Manual* (by Steven Skiena).

XII

Code Library

Certain patterns came up while implementing the code for this book. We've tried to generally include the full code for a solution with the solution, but in some cases it got quite redundant.

This appendix provides the code for a few of the most useful chunks of code.

The complete compilable solutions can be downloaded from CrackingTheCodingInterview.com.

XI

Code Library

Certain patterns came up while implementing the code for this book. We've tried to generally include the full code for a solution with the solution, but in some cases it got quite redundant.

This appendix provides the code for a few of the most useful chunks of code.

All code for the book can be downloaded from CrackingTheCodingInterview.com.

▸ HashMapList<T, E>

The `HashMapList` class is essentially shorthand for `HashMap<T, ArrayList<E>>`. It allows us to map from an item of type of T to an `ArrayList` of type E.

For example, we might want a data structure that maps from an integer to a list of strings. Ordinarily, we'd have to write something like this:

```
1  HashMap<Integer, ArrayList<String>> maplist =
2    new HashMap<Integer, ArrayList<String>>();
3  for (String s : strings) {
4    int key = computeValue(s);
5    if (!maplist.containsKey(key)) {
6      maplist.put(key, new ArrayList<String>());
7    }
8    maplist.get(key).add(s);
9  }
```

Now, we can just write this:

```
1  HashMapList<Integer, String> maplist = new HashMapList<Integer, String>();
2  for (String s : strings) {
3    int key = computeValue(s);
4    maplist.put(key, s);
5  }
```

It's not a big change, but it makes our code a bit simpler.

```
1  public class HashMapList<T, E> {
2    private HashMap<T, ArrayList<E>> map = new HashMap<T, ArrayList<E>>();
3
4    /* Insert item into list at key. */
5    public void put(T key, E item) {
6      if (!map.containsKey(key)) {
7        map.put(key, new ArrayList<E>());
8      }
9      map.get(key).add(item);
```

```
    }

    /* Insert list of items at key. */
    public void put(T key, ArrayList<E> items) {
        map.put(key, items);
    }

    /* Get list of items at key. */
    public ArrayList<E> get(T key) {
        return map.get(key);
    }

    /* Check if hashmaplist contains key. */
    public boolean containsKey(T key) {
        return map.containsKey(key);
    }

    /* Check if list at key contains value. */
    public boolean containsKeyValue(T key, E value) {
        ArrayList<E> list = get(key);
        if (list == null) return false;
        return list.contains(value);
    }

    /* Get the list of keys. */
    public Set<T> keySet() {
        return map.keySet();
    }

    @Override
    public String toString() {
        return map.toString();
    }
}
```

▸ TreeNode (Binary Search Tree)

While it's perfectly fine—even good—to use the built-in binary tree class when possible, it's not always possible. In many questions, we needed access to the internals of the node or tree class (or needed to tweak these) and thus couldn't use the built-in libraries.

The `TreeNode` class supports a variety of functionality, much of which we wouldn't necessarily want for every question/solution. For example, the TreeNode class tracks the parent of the node, even though we often don't use it (or specifically ban using it).

For simplicity, we'd implemented this tree as storing integers for data.

```
public class TreeNode {
    public int data;
    public TreeNode left, right, parent;
    private int size = 0;

    public TreeNode(int d) {
        data = d;
        size = 1;
    }
```

```
10
11    public void insertInOrder(int d) {
12       if (d <= data) {
13          if (left == null) {
14             setLeftChild(new TreeNode(d));
15          } else {
16             left.insertInOrder(d);
17          }
18       } else {
19          if (right == null) {
20             setRightChild(new TreeNode(d));
21          } else {
22             right.insertInOrder(d);
23          }
24       }
25       size++;
26    }
27
28    public int size() {
29       return size;
30    }
31
32    public TreeNode find(int d) {
33       if (d == data) {
34          return this;
35       } else if (d <= data) {
36          return left != null ? left.find(d) : null;
37       } else if (d > data) {
38          return right != null ? right.find(d) : null;
39       }
40       return null;
41    }
42
43    public void setLeftChild(TreeNode left) {
44       this.left = left;
45       if (left != null) {
46          left.parent = this;
47       }
48    }
49
50    public void setRightChild(TreeNode right) {
51       this.right = right;
52       if (right != null) {
53          right.parent = this;
54       }
55    }
56
57 }
```

This tree is implemented to be a binary search tree. However, you can use it for other purposes. You would just need to use the `setLeftChild`/`setRightChild` methods, or the `left` and `right` child variables. For this reason, we have kept these methods and variables `public`. We need this sort of access for many problems.

LinkedListNode (Linked List)

Like the TreeNode class, we often needed access to the internals of a linked list in a way that the built-in linked list class wouldn't support. For this reason, we implemented our own class and used it for many problems.

```
1  public class LinkedListNode {
2    public LinkedListNode next, prev, last;
3    public int data;
4    public LinkedListNode(int d, LinkedListNode n, LinkedListNode p){
5      data = d;
6      setNext(n);
7      setPrevious(p);
8    }
9
10   public LinkedListNode(int d) {
11     data = d;
12   }
13
14   public LinkedListNode() { }
15
16   public void setNext(LinkedListNode n) {
17     next = n;
18     if (this == last) {
19       last = n;
20     }
21     if (n != null && n.prev != this) {
22       n.setPrevious(this);
23     }
24   }
25
26   public void setPrevious(LinkedListNode p) {
27     prev = p;
28     if (p != null && p.next != this) {
29       p.setNext(this);
30     }
31   }
32
33   public LinkedListNode clone() {
34     LinkedListNode next2 = null;
35     if (next != null) {
36       next2 = next.clone();
37     }
38     LinkedListNode head2 = new LinkedListNode(data, next2, null);
39     return head2;
40   }
41 }
```

Again, we've kept the methods and variables public because we often needed this access. This would allow the user to "destroy" the linked list, but we actually needed this sort of functionality for our purposes.

Trie & TrieNode

The trie data structure is used in a few problems to make it easier to look up if a word is a prefix of any other words in a dictionary (or list of valid words). This is often used when we're recursively building words so that we can short circuit when the word is not valid.