



Discussion 9

Disjoint Sets, (Balanced) Binary Search Trees, Hashing



Announcements

- Homework 2 due Wednesday, March 14
- Homework 3 due Monday, March 19
- Midterm 2 on Tuesday, March 20
 - Midterm review session Friday 16th 8-10 PM
 - Guerrilla section Saturday 17th 12-2 PM



Disjoint Sets

- Data structure that keeps track of whether or not elements are connected
- Two basic functions:
 - `connect(a, b)`
 - `isConnected(a, b)`

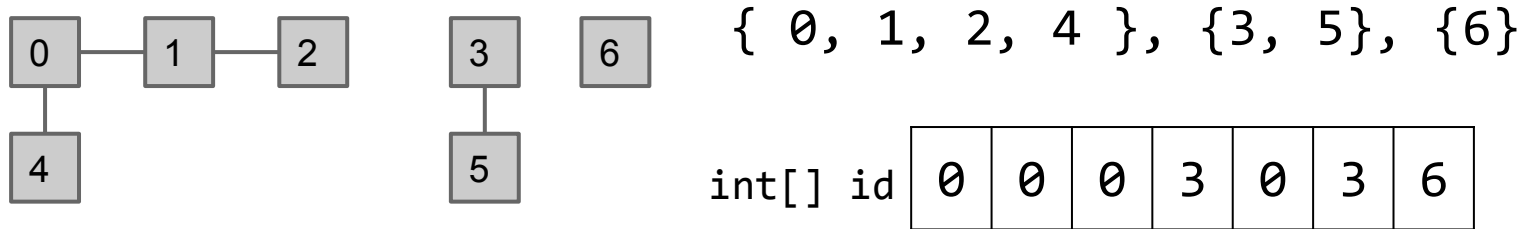


Implementations of Disjoint Sets

- Quick Find
- Quick Union
- Weighted Quick Union

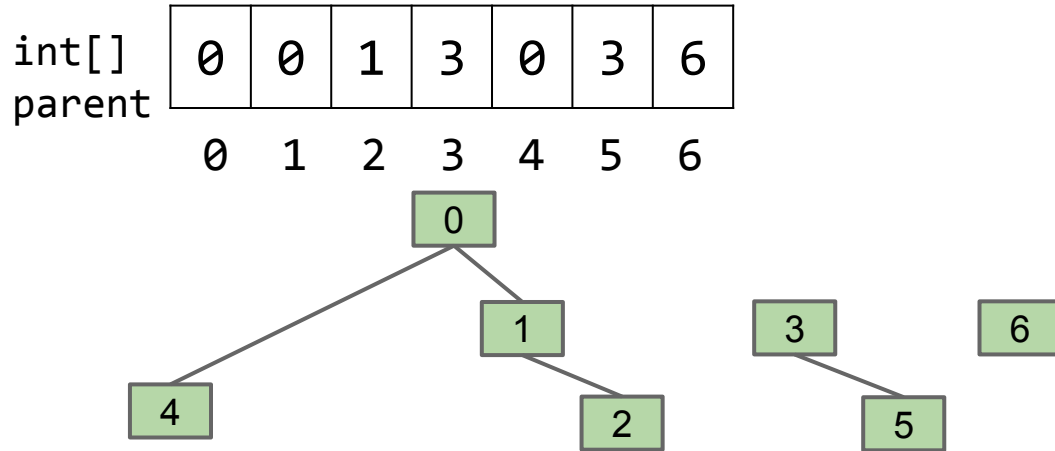
Quick Find

Use an array to store which set each element is in



Quick Union

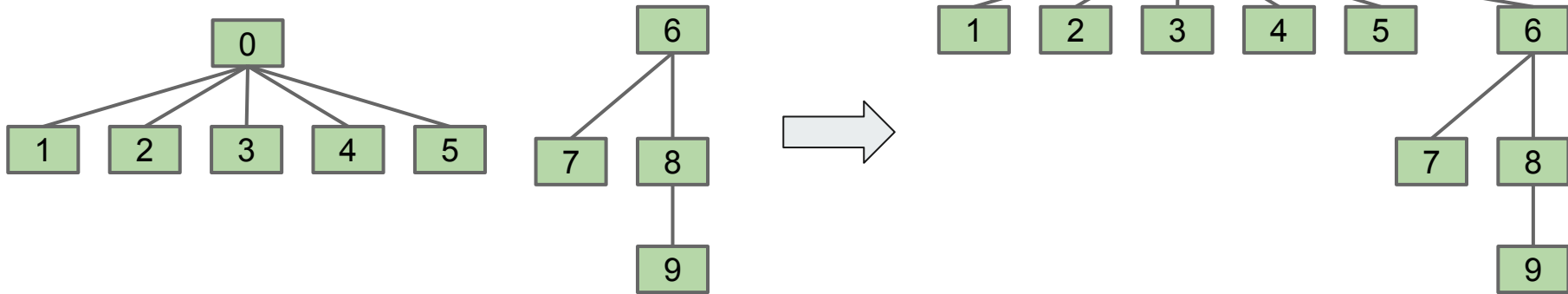
Use an array to store the parent of each node



Weighted Quick Union

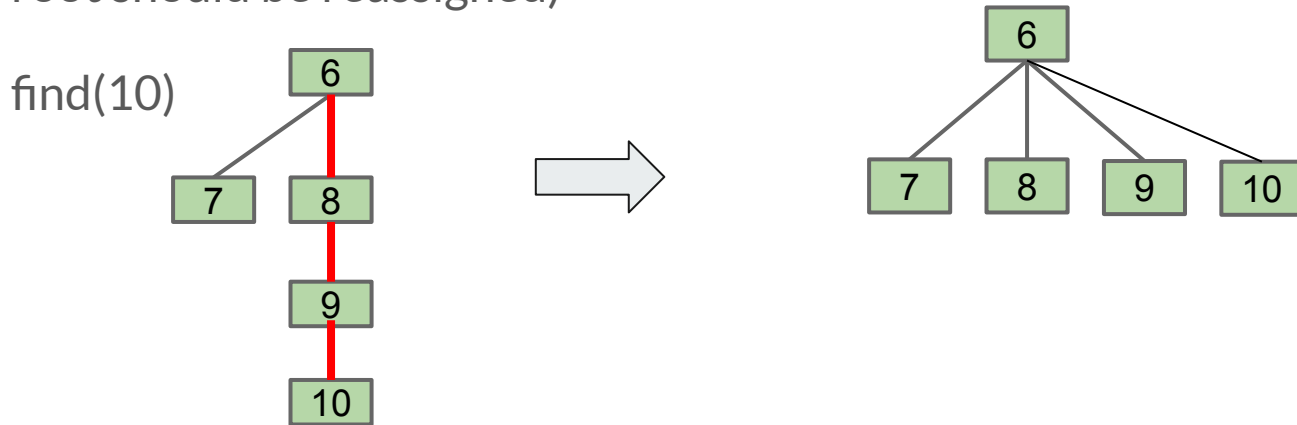
Quick Union, but always add smaller tree to larger tree

connect(3,8)



Weighted Quick Union with Path Compression

When you call find (recall that union calls find as well), as you traverse to the root, reassign every node's parent to the root. (Each node on path to the root should be reassigned)





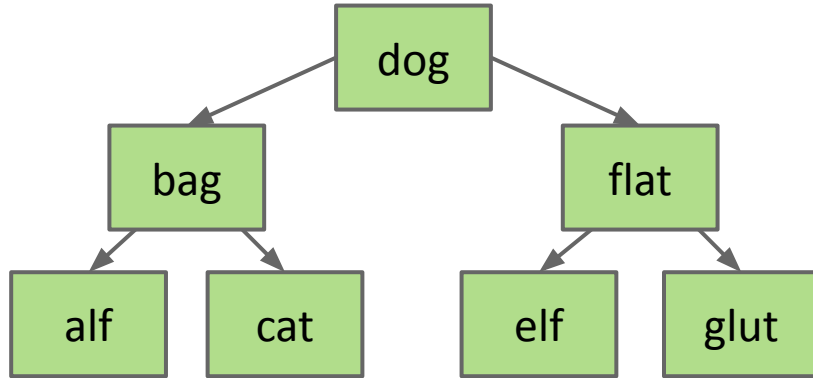
Disjoint Set Runtimes

Implementation	Constructor	connect	isConnected
QuickFindDS	$\Theta(N)$	$\Theta(N)$	$\Theta(1)$
QuickUnionDS	$\Theta(N)$	$O(N)$	$O(N)$
WeightedQuickUnionDS	$\Theta(N)$	$O(\log N)$	$O(\log N)$
^ with Path compress	$\Theta(N)$	$O(1)^*$	$O(1)^*$

*Technically inverse ackermann function (in lecture), but pretty much constant

Binary Search Tree

- Every key in the **left** subtree is **less** than X's key
- Every key in the **right** subtree is **greater** than X's key



Binary Search Tree



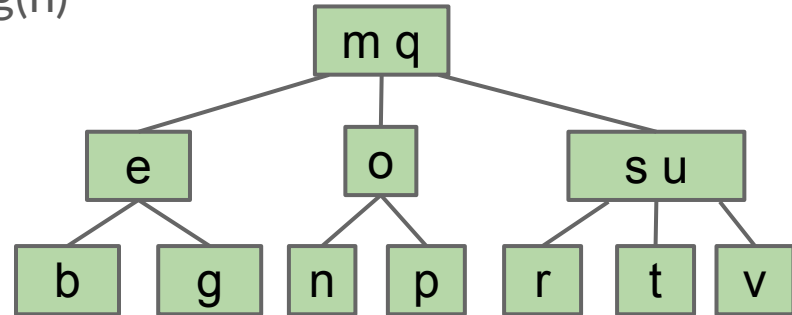
BST Runtimes

Shape of Tree	Height	add	search
“Bushy” (Best Case)	$\log N$	$\Theta(\log N)$	$\Theta(\log N)$
“Spindly” (Worst Case)	N	$\Theta(N)$	$\Theta(N)$

2-3 Tree

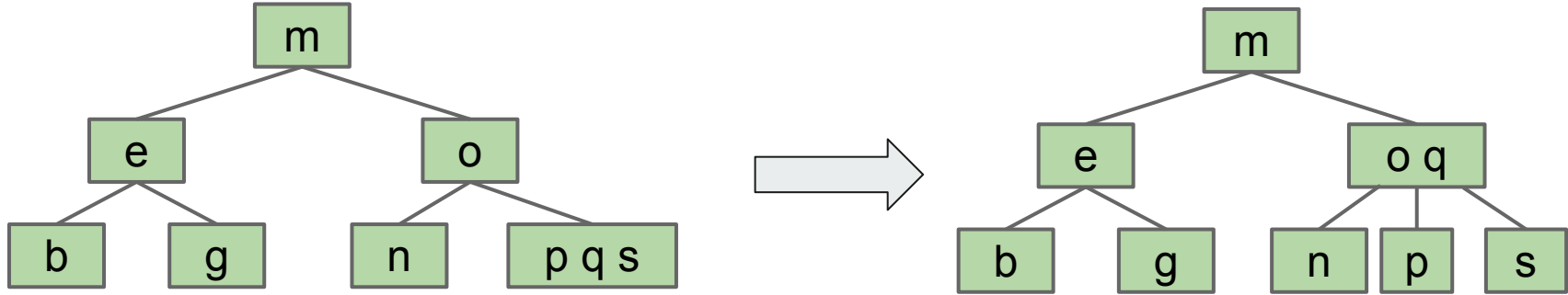
BST where a node can have 2,3 children

Ensures tree is “bushy” -> height is $\log(n)$



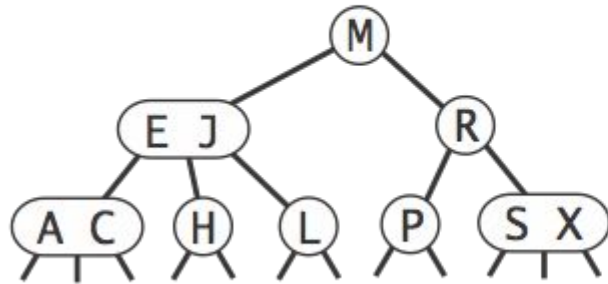
Insertion Example (2-3 Tree)

2-3 Tree after inserting "s"

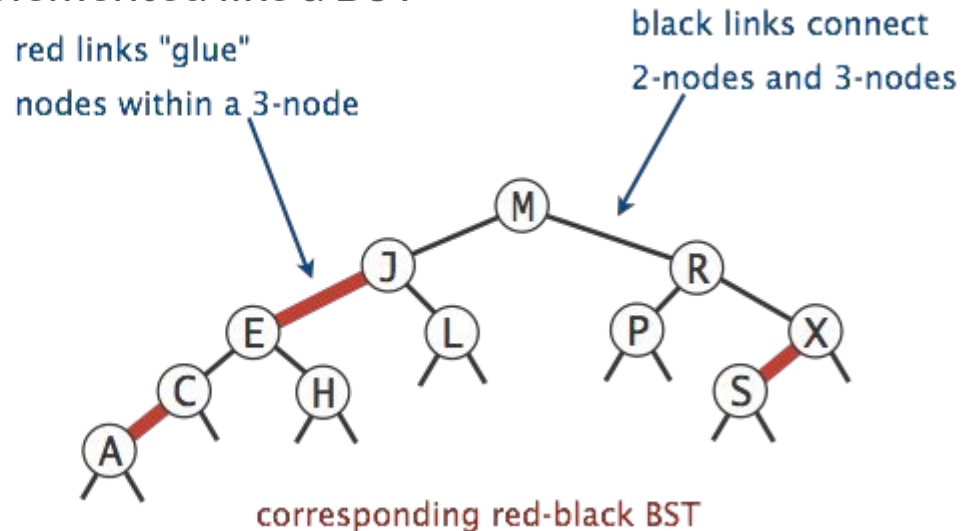


LLRB Tree

Properties of a 2-3 tree, but implemented like a BST



2-3 tree





Hashing

Function that produces an integer from a Java object

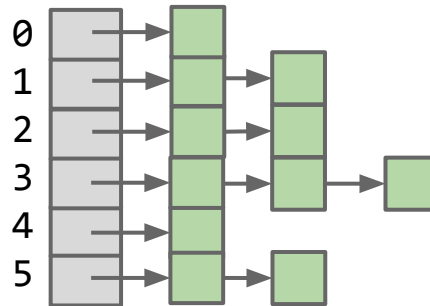
Properties:

1. The hashcode for the same object should always be the same
2. If two objects are “equal”, they have the same hashcode

Hashtable

Use the hashcode % array.length to index into an array

On collision, one resolution is external chaining





HashTable Runtimes

	add	get	remove
Best Case	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Worst Case	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$

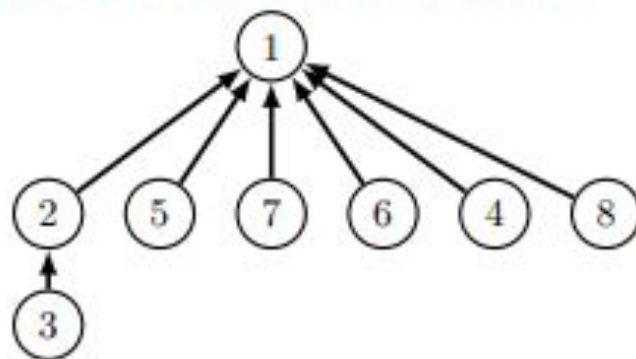



1 WQU and Path Compression

Assume we have eight sets, represented by integers 1 through 8, that start off as completely disjoint sets. Draw the WQU Tree after the series of `union()` and `find()` operations with path compression. Write down the result of `find()` operations. Break ties by choosing the smaller integer to be the root.

```
union(2, 3);  
union(1, 6);  
union(5, 7);  
union(8, 4);  
union(7, 2);  
find(3);  
union(6, 4);  
union(6, 3);  
find(7);  
find(8);
```

find() returns 2, 1, 1 respectively






2 Is This a BST?

The following method `isBSTBad` is supposed check if a given binary tree is a BST, though for some binary trees, it is returning the wrong answer. Think about an example of a binary tree for which `isBSTBad` fails. Then, write `isBSTGood` so that it returns the correct answer for any binary tree. The `TreeNode` class is defined as follows:



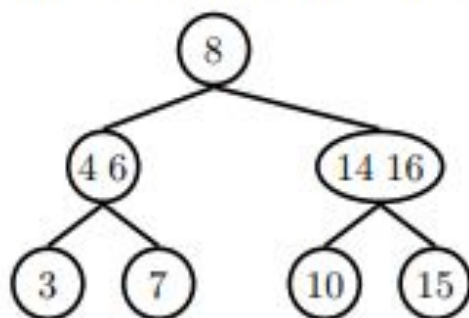
An example of a binary tree for which the method fails:

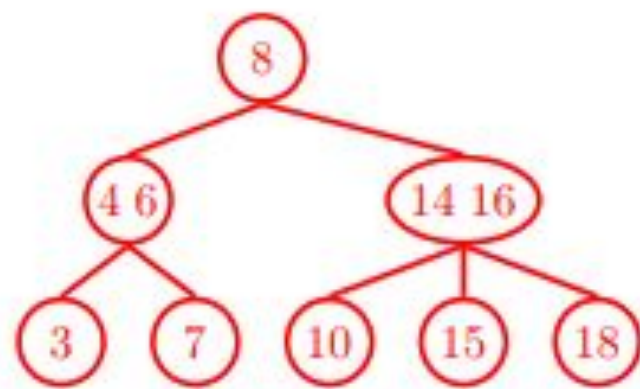


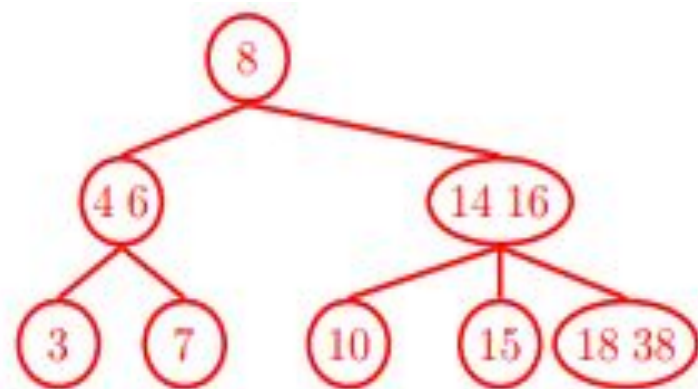


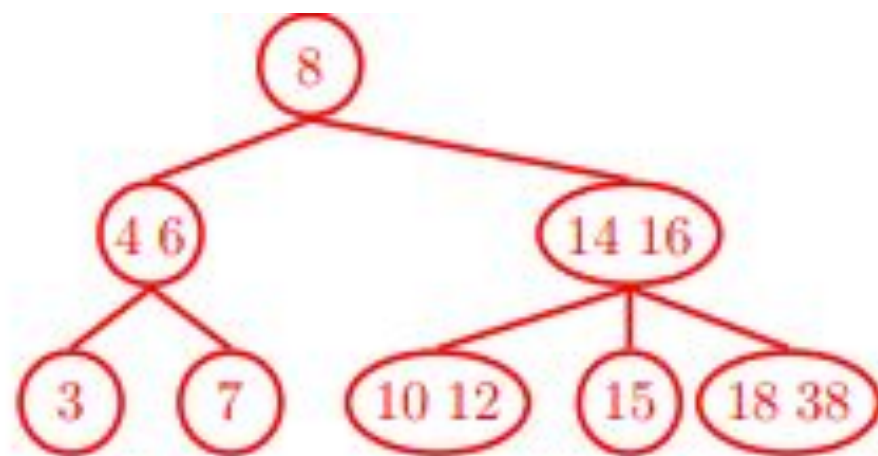
```
public static boolean isBSTGood(TreeNode T) {  
    return isBSTHelper(T, Integer.MIN_VALUE, Integer.MAX_VALUE);  
}  
  
public static boolean isBSTHelper(TreeNode T, int min, int max) {  
    if (T == null) {  
        return true;  
    } else if (T.val < min || T.val > max) {  
        return false;  
    } else {  
        return isBST(T.left, min, T.val) && isBST(T.right, T.val, max);  
    }  
}
```

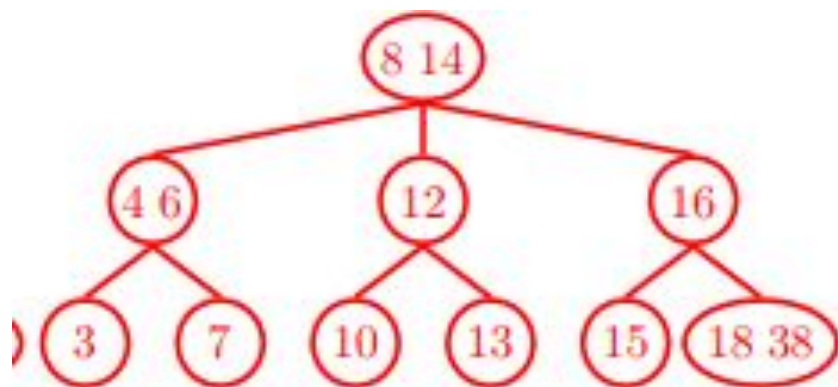
3.1 Draw what the following 2-3 tree would look like after inserting 18, 38, 12, and 13.




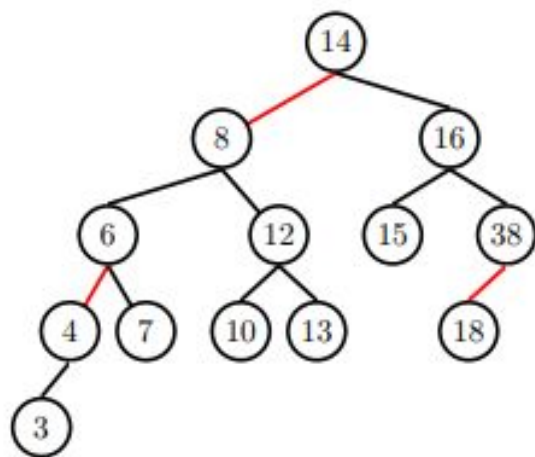









- 
- 3.2 Now, convert the resulting 2-3 tree to a left-leaning red-black tree.






3.3

Extra: If a 2-3 tree has depth H (that is, the leaves are at distance H from the root), what is the maximum number of comparisons done in the corresponding red-black tree to find whether a certain key is present in the tree?




$2h$ comparisons. The maximum number of comparisons occur from a root to leaf path with the most nodes. Because the height of the tree is h , we know that there is a path down the leaf-leaning red-black tree that consists of at most h black links, for black links in the left-leaning red-black tree are the links that add to the height of the corresponding 2-3 tree. In the worst case, in the 2-3 tree representation, this path can consist entirely of nodes with two items, meaning in the left-leaning red-black tree representation, each black link is followed by a red link. This doubles the amount of nodes on this path from the root to the leaf. This example will represent our longest path, which is $2h$ nodes long, meaning we make at most $2h$ comparisons in the left-leaning red-black tree.

- 
- 4.1 Here are three potential implementations of the `Integer`'s `hashCode()` function. Categorize each as either a valid or an invalid hash function. If it is invalid, explain why. If it is valid, point out a flaw or disadvantage.

```
1 public int hashCode() {  
2     return -1;  
3 }
```

```
1 public int hashCode() {  
2     return intValue() * intValue();  
3 }
```

```
1 public int hashCode() {  
2     return super.hashCode();  
3 }
```

```
1 public int hashCode() {  
2     return -1;  
3 }
```

Valid. As required, this hash function returns the same `hashCode` for Integers that are `equals()` to each other. However, this is a terrible hash code because collisions are extremely frequent (collisions occur 100% of the time).

```
1 public int hashCode() {  
2     return intValue() * intValue();  
3 }
```

Valid. Similar to (a), this hash function returns the same `hashCode` for integers that are `equals()`. However, integers that share the same absolute values will collide (for example, $x = 5$ and $x = -5$ will have the same hash code). A better hash function would be to just return the `intValue()` itself.

```
1 public int hashCode() {  
2     return super.hashCode();  
3 }
```

Invalid. This is not a valid hash function because integers that are `equals()` to each other will not have the same hash code. Instead, this hash function returns some integer corresponding to the integer object's location in memory.