Student name: Quan Hong Doan

Student ID: 986956

Assignment: Lab 9, & 10

Week: 10

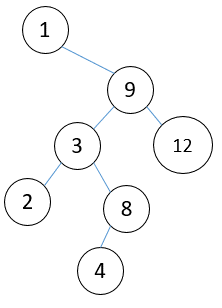
Due: Dec 23, 2019

**Lab 9**

**Problem 1**:

Draw the BST obtained by inserting the following integers (using successive calls to insert): 1, 9, 3, 8, 12, 4, 2

**Answer**:



**Problem 2**:

Create a sorting routine based on a BST and place it in the sorting environment, distributed earlier. For this, your new class, BSTSort, should be a subclass of Sorter. Your BSTSort class can be essentially the same as the BST class given in the slides (see the folder in your labs directory for this lab), except that you will need to modify the printTree method so that it outputs values to an array (rather than printing to console).

After you have implemented, discuss the asymptotic running time of your new sorting algorithm. Run an empirical test in the sorting environment and explain where BSTSort fits in with the other sorting routines (which algorithms is it faster than? which is it slower than?).

**Answer**:

For the insertion into BST, the average case time complexity is O (n log n), because each time we add an item to BST, it takes O (log n) on average. So, adding n items will take O (n log n).

In worst case, that would be O (n^2). If we use self-balancing BST like AVL Tree or Red Black Tree, then it will take O (n log n) time to sort the array.

Following is the implementation.

**public** **class** MyBSTSort {

Node root;

MyBSTSort() {

root = **null**;

}

**class** Node {

**int** key;

Node left, right;

**public** Node(**int** key) {

**this**.key = key;

left = right = **null**;

}

}

**void** insert(**int** key) {

root = insertRecursive(root, key);

}

Node insertRecursive(Node root, **int** key) {

**if** (**null** == root) {

root = **new** Node(key);

**return** root;

}

**if** (key < root.key)

root.left = insertRecursive(root.left, key);

**else**

root.right = insertRecursive(root.right, key);

**return** root;

}

**void** inOrderTraversal(Node root) { // Left-Visit-Right

**if** (**null** != root) {

inOrderTraversal(root.left);

System.***out***.print(root.key + " ");

inOrderTraversal(root.right);

}

}

**public** **static** **void** main(String[] args) {

MyBSTSort my = **new** MyBSTSort();

**int** arr[] = {5, 4, 7, 2, 11};

System.***out***.println("In array: " + Arrays.*toString*(arr));

**for** (**int** i : arr) {

my.insert(i);

}

System.***out***.print("Sorted array: ");

my.inOrderTraversal(my.root);

}

}

Console logs

In array: [5, 4, 7, 2, 11]

Sorted array: 2 4 5 7 11

**Problem 3:**

For each integer n = 1, 2, 3,…, 7, determine whether there exists a red-black tree

having exactly n nodes, with all of them black. Fill out the chart below to tabulate the

results:

Num nodes n Does there exist a red-black tree with n nodes, all of which are black?

1 Yes

2

…

**Answer**:

|  |  |
| --- | --- |
| Num nodes n | Does there exist a red-black tree with n nodes, all of which are black? |
| 1 | Yes |
| 2 | No |
| 3 | Yes |
| 4 | No |
| 5 | No |
| 6 | No |
| 7 | No |

As we know the black height <= log (n + 1), so if we want all nodes are black, meaning black height = log (n+1). From our example, there are only 1 and 3 will meet the requirement, which result the black height to 1 and 2.

**Problem 4**:

For each integer n = 1,2,3,…, 7, determine whether there exists a red-black tree

having exactly n nodes, where exactly one of the nodes is red. Fill out the chart below

to tabulate the results:

Num nodes n Does there exist a red-black tree with n nodes, where exactly one of the nodes is red?

1 No

2

…

**Answer**:

|  |  |
| --- | --- |
| Num nodes n | Does there exist a red-black tree with n nodes, where exactly one of the nodes is red? |
| 1 | No |
| 2 | Yes |
| 3 | No |
| 4 | Yes |
| 5 | No |
| 6 | No |
| 7 | No |

Similar to the problem 3, when we want all red, that means the red-height = log (n), where n = 2 or 4, because the height should be integer.

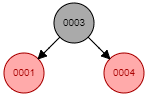
**Lab 10**

**Problem 1**:

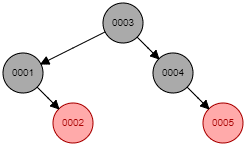
A red-black tree is said to be derivable if it is obtained from an insertion sequence of nodes, using the rules for insertions starting from an empty tree. Give an example to show that not every red-black tree is derivable. (In other words, you can build a BST that satisfies the four conditions for a red-black tree, and yet there is no way to obtain this tree by successively inserting nodes using the insertion algorithm rules.)

**Answer**:

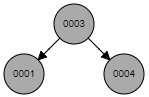
If we create a RB tree from {1, 3, 4}, then we would have as below.



However, if we add 2 more items like {2, 5}, then our tree becomes.



In this case, we remove {2, 5}, then the tree is not derivable anymore.



Console logs:

Insert node=1

Insert node=3

Fix tree for node=3

Insert node=4

Fix tree for node=4

node.parent == node.parent.parent.right

Rotate left node=1

Color: Red Key: 1 Parent: 3

Color: Black Key: 3 Parent: -1

Color: Red Key: 4 Parent: 3

Insert node=2

Fix tree for node=2

node.parent == node.parent.parent.left

Insert node=5

Fix tree for node=5

Color: Black Key: 1 Parent: 3

Color: Red Key: 2 Parent: 1

Color: Black Key: 3 Parent: -1

Color: Black Key: 4 Parent: 3

Color: Red Key: 5 Parent: 4

Delete node=5

Delete node=2

Color: Black Key: 1 Parent: 3

Color: Black Key: 3 Parent: -1

Color: Black Key: 4 Parent: 3

**Problem 2**:

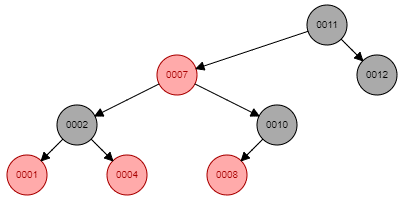
An AVL Tree is a BST that satisfies a different balance condition, namely:

The AVL Balance Condition For each internal node x, the height of the left child of x differs from the height of the right child of x by at most 1. (Equivalently, the heights of the left and right subtrees of x differ by at most 1.)

Create a red-black tree that does not satisfy the AVL Balance Condition.

**Answer**:

Array = { 10, 11, 12, 7, 2, 4, 8, 1 }



Console logs

Insert node=10

Insert node=11

Fix tree for node=11

Insert node=12

Fix tree for node=12

node.parent == node.parent.parent.right

Rotate left node=10

Insert node=7

Fix tree for node=7

node.parent == node.parent.parent.left

Insert node=2

Fix tree for node=2

node.parent == node.parent.parent.left

Rotate right node=10

Insert node=4

Fix tree for node=4

node.parent == node.parent.parent.left

Insert node=8

Fix tree for node=8

Insert node=1

Fix tree for node=1

Color: Red Key: 1 Parent: 2

Color: Black Key: 2 Parent: 7

Color: Red Key: 4 Parent: 2

Color: Red Key: 7 Parent: 11

Color: Red Key: 8 Parent: 10

Color: Black Key: 10 Parent: 7

Color: Black Key: 11 Parent: -1

Color: Black Key: 12 Parent: 11

**Problem 3:**

Use the insertion algorithm for red-black trees to successively insert the following nodes, starting with an empty tree.

a. 1, 2, 3, 4, 5, 6, 7, 8

b. 3, 2, 1, 4, 5, 6

Note on Part (a): Recall that an already sorted insertion sequence is a worst case for an ordinary BST. Notice how the red-black balancing operations handle this to remain balanced.

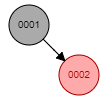
**Answer**:

**For the {1, 2, 3, 4, 5, 6, 7, 8}**

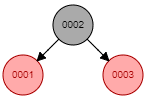
* Insert {1}



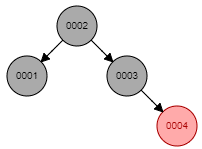
* Insert {2}



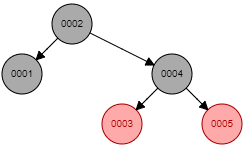
* Insert {3}



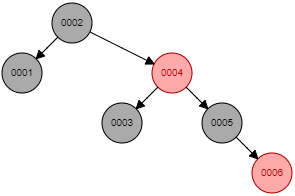
* Insert {4}



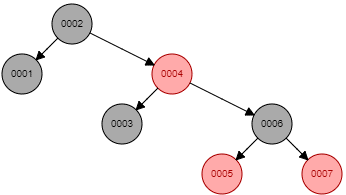
* Insert {5}



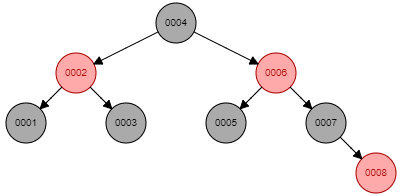
* Insert {6}



* Insert {7}



* Insert {8}



Console logs:

Insert node=1

Insert node=2

Fix tree for node=2

Insert node=3

Fix tree for node=3

node.parent == node.parent.parent.right

Rotate left node=1

Insert node=4

Fix tree for node=4

node.parent == node.parent.parent.right

Insert node=5

Fix tree for node=5

node.parent == node.parent.parent.right

Rotate left node=3

Insert node=6

Fix tree for node=6

node.parent == node.parent.parent.right

Insert node=7

Fix tree for node=7

node.parent == node.parent.parent.right

Rotate left node=5

Insert node=8

Fix tree for node=8

node.parent == node.parent.parent.right

node.parent == node.parent.parent.right

Rotate left node=2

Color: Black Key: 1 Parent: 2

Color: Red Key: 2 Parent: 4

Color: Black Key: 3 Parent: 2

Color: Black Key: 4 Parent: -1

Color: Black Key: 5 Parent: 6

Color: Red Key: 6 Parent: 4

Color: Black Key: 7 Parent: 6

Color: Red Key: 8 Parent: 7

**For the {3, 2, 1, 4, 5, 6}**

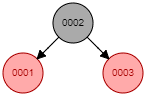
* Insert {3}



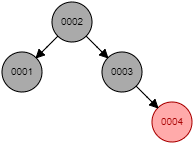
* Insert {2}



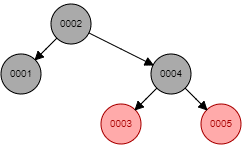
* Insert {1}



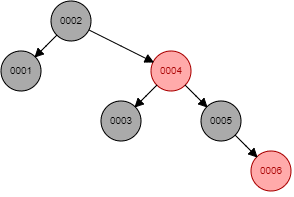
* Insert {4}



* Insert {5}



* Insert {6}



Console logs:

Insert node=3

Insert node=2

Fix tree for node=2

Insert node=1

Fix tree for node=1

node.parent == node.parent.parent.left

Rotate right node=3

Insert node=4

Fix tree for node=4

node.parent == node.parent.parent.right

Insert node=5

Fix tree for node=5

node.parent == node.parent.parent.right

Rotate left node=3

Insert node=6

Fix tree for node=6

node.parent == node.parent.parent.right

Color: Black Key: 1 Parent: 2

Color: Black Key: 2 Parent: -1

Color: Black Key: 3 Parent: 4

Color: Red Key: 4 Parent: 2

Color: Black Key: 5 Parent: 4

Color: Red Key: 6 Parent: 5

**Problem 4**:

Devise an algorithm IsPrime(n) which outputs TRUE if n is prime, FALSE otherwise. Then implement as a Java method. What is the asymptotic running time of IsPrime? Explain.

**Answer**:

To check a number is prime or not we can approach in few ways.

The simplest method is using trial division, we will check the number against its square root, and even better, we can ensure that all primes are in form of 6k ± 1 (not including 2 or 3). The reference is <https://en.wikipedia.org/wiki/Primality_test>

The time complexity for this is O ( )

We also have some other approaches like applying Fermat Little’s Theorem or an advanced version with Miller-Rabin test. Both have some advantages. For the Fermat’s, the time complexity is O (k log n), and the Miller-Rabin’s would take [O](https://en.wikipedia.org/wiki/Big_O_notation) (*k* log3*n*).

Reference:

* Fermat Little’s Theorem: <https://en.wikipedia.org/wiki/Proofs_of_Fermat's_little_theorem>
* Miller-Rabin test: <https://en.wikipedia.org/wiki/Miller-Rabin_primality_test>

Implementations below:

**public** **class** CheckingPrime {

/\*\*

\* Instead of checking until n, we check until square of n

\* Also, all primes are of the form 6k +/- 1 (not including 2, 3).

\* https://en.wikipedia.org/wiki/Primality\_test

\* **@return**

\*/

**static** **boolean** isPrimeSimple(**int** n) {

**if** (n <= 1) **return** **false**;

**if** (n <= 3) **return** **true**; // n = 2 or 3

**if** (n % 2 == 0 || n % 3 == 0) **return** **false**;

**for** (**int** i = 5; i \* i <= n; i = i + 6)

**if** (n % i == 0 || n % (i + 2) == 0)

**return** **false**;

**return** **true**;

}

/\*\*

\* Fermat's Little Theorem If n is a prime,

\* then for every a, 1 < a < n - 1,

\* a^(n-1) = 1 (mod n) OR a^(n-1) % n = 1

\*

\* https://en.wikipedia.org/wiki/Proofs\_of\_Fermat's\_little\_theorem

\* If n is prime, then always returns true.

\* If n is composite than returns false with high probability.

\* Higher value of k increases probability of correct result.

\*/

**static** **boolean** isPrimeFermat(**int** n) {

**int** k = 3;

**if** (n <= 1 || n == 4)

**return** **false**;

**if** (n <= 3)

**return** **true**;

// Try k times

**while** (k > 0) {

// Pick a random number in [2..n-2]

// Above corner cases make sure that n > 4

**int** a = 2 + (**int**) (Math.*random*() % (n - 4));

// Fermat's little theorem

**if** (*powerFermat*(a, n - 1, n) != 1)

**return** **false**;

k--;

}

**return** **true**;

}

/\*\*

\* Iterative Function to calculate (a^n)%p in O(logy)

\*/

**static** **int** powerFermat(**int** a,**int** n, **int** p) {

**int** res = 1;

// if a >= p

a = a % p;

**while** (n > 0) {

// If n is odd, multiply a with result

**if** ((n & 1) == 1)

res = (res \* a) % p;

// n must be even now

n = n >> 1; // n = n/2

a = (a \* a) % p;

}

**return** res;

}

/\*\*

\* It returns false if n is composite and returns true if n is probably prime.

\* k is an input parameter that determines accuracy level.

\* Higher value of k indicates more accuracy.

\* https://en.wikipedia.org/wiki/Miller–Rabin\_primality\_test

\*/

**static** **boolean** isPrimeMillerRabin(**int** n, **int** k) {

**if** (n <= 1 || n == 4)

**return** **false**;

**if** (n <= 3)

**return** **true**;

// Find r such that n = 2^d \* r + 1

// for some r >= 1

**int** d = n - 1;

**while** (d % 2 == 0)

d /= 2;

// Iterate given number of 'k' times

**for** (**int** i = 0; i < k; i++)

**if** (!*miillerTest*(d, n))

**return** **false**;

**return** **true**;

}

/\*\*

\* This function is called for all k trials.

\* It returns false if n is composite and returns false if n is probably prime.

\* d is an odd number such that d\*2^r = n-1 for some r >= 1.

\*/

**static** **boolean** miillerTest(**int** d, **int** n) {

// Pick a random number in [2..n-2]

// Corner cases make sure that n > 4

**int** a = 2 + (**int**)(Math.*random*() % (n - 4));

// Compute a^d % n

**int** x = *powerMillerRabin*(a, d, n);

**if** (x == 1 || x == n - 1)

**return** **true**;

// Keep squaring x while one of the following doesn't happen

// - d does not reach n-1

// - (x^2) % n is not 1

// - (x^2) % n is not n-1

**while** (d != n - 1) {

x = (x \* x) % n;

d \*= 2;

**if** (x == 1)

**return** **false**;

**if** (x == n - 1)

**return** **true**;

}

// Return composite

**return** **false**;

}

/\*\*

\* Function to do modular exponentiation.

\* It returns (x^y) % p.

\*/

**static** **int** powerMillerRabin(**int** x, **int** y, **int** p) {

**int** res = 1;

//Update x if it is more than or equal to p

x = x % p;

**while** (y > 0) {

// If y is odd, multiply x with result

**if** ((y & 1) == 1)

res = (res \* x) % p;

// y must be even now

y = y >> 1; // y = y/2

x = (x \* x) % p;

}

**return** res;

}

}

Console logs

Simple Prime checking solution

isPrimeSimple: 97 is prime? => true

isPrimeSimple: 63 is prime? => false

isPrimeSimple: 101 is prime? => true

Prime checking solution with Fermat Little's Theorem

isPrimeFermat: 97 is prime? => true

isPrimeFermat: 63 is prime? => false

isPrimeFermat: 101 is prime? => true

Prime checking solution with Miller-Rabin

isPrimeMillerRabin: 97 is prime? => true

isPrimeMillerRabin: 63 is prime? => false

isPrimeMillerRabin: 101 is prime? => true