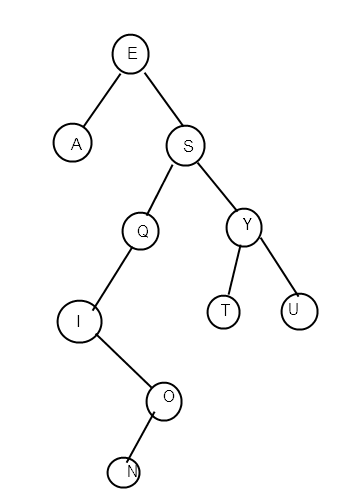
**3.2.1** Draw the BST that results when you insert the keys E A S Y Q U E S T I O N, in that order (associating the value i with the ith key, as per the convention in the text) into an initially empty tree. How many compares are needed to build the tree?



Total compares = 24.

**3.2.2** Inserting the keys in the order A X C S E R H into an initially empty BST gives a worst-case tree where every node has one null link, except one at the bottom, which has two null links. Give five other orderings of these keys that produce worst-case trees.

0. A C E H R S X

1. X S R H E C A

2. X S R H E A C

3. X S R H A E C

4. X S R H A C E

5 X S R A H C E

**3.2.3** Give five orderings of the keys A X C S E R H that, when inserted into an initially empty BST, produce the *best-case* tree.

H C A E S R X

**3.2.4** Suppose that a certain BST has keys that are integers between 1 and 10, and we search for 5. Which sequence below *cannot* be the sequence of keys examined?

*a.* 10, 9, 8, 7, 6, 5

*b.* 4, 10, 8, 7, 5

*c.* 1, 10, 2, 9, 3, 8, 4, 7, 6, 5

*d.* 2, 7, 3, 8, 4, 5 impossible for <7, 3, 8>

*e.* 1, 2, 10, 4, 8, 5

**3.2.5** Suppose that we have an estimate ahead of time of how often search keys are to be accessed in a BST, and the freedom to insert them in any order that we desire. Should the keys be inserted into the tree in increasing order, decreasing order of likely frequency of access, or some other order? Explain your answer.

Decreasing order is much better and reasonable than increasing order for those keys with high frequency of access will have shorter distance to root which means shorter search length or less compares needed. Though this may not be the optimal solution

Consider this:

Key Frequency

A 1000

B 999

C 998

Inserting these keys with decreasing frequency (A, B, C) will get a worst case binary tree and the total compares will be 1 \* 1000 + 2 \* 999 + 3 \* 998 = 5992 whereas insert them in (B, A, C) order will get 2 \* 1000 + 1 \* 999 + 2 \* 998 = 4995 < 5992

**3.2.6** Add to BST a method height() that computes the height of the tree. Develop two implementations: a recursive method (which takes linear time and space proportional to the height), and a method like size() that adds a field to each node in the tree (and takes linear space and constant time per query).

//Definition. The size of a tree is its number of nodes. The depth of a node in a tree

//is the number of links on the path from it to the root. The height of a tree is the

//maximum depth among its nodes.

**public** **int** height()

{

**return** height(root) - 1;

}

**private** **int** height(Node n)

{

**if**(n == **null**) **return** 0;

**return** Math.*max*(height(n.Left), height(n.Right)) + 1;

}

**private** **class** Node

{

**public** Node Left;

**public** Node Right;

**public** Value Value;

**public** Key Key;

**public** **int** Size;

**public** **int** Height;

**public** Node(Key key, Value value)

{

**this**.Key = key;

**this**.Value = value;

**this**.Size = 1;

**this**.Height = 1;

}

}

**private** Node put(Node n, Key key, Value value)

{

**if**(n == **null**) **return** **new** Node(key, value);

**int** cmp = n.Key.compareTo(key);

**if**(cmp > 0) n.Left = put(n.Left, key, value);

**else** **if** (cmp < 0) n.Right = put(n.Right, key, value);

**else**

{

n.Key = key;

n.Value = value;

}

n.Size = size(n.Left) + size(n.Right) + 1;

**n.Height = Math.*max*(height(n.Left), height(n.Right)) + 1;**

**return** n;

}

**private** Node delete(Node n, Key k)

{

**if**(n == **null**) **return** **null**;

**int** cmp = n.Key.compareTo(k);

**if**(cmp > 0) n.Left = delete(n.Left, k);

**else** **if**(cmp < 0) n.Right = delete(n.Right, k);

**else**

{

**if**(n.Left == **null**) **return** n.Right;

**if**(n.Right == **null**) **return** n.Left;

Node t = n;

n = minNode(t.Right);

n.Right = delMin(t.Right);

n.Left = n.Left;

}

n.Size = size(n.Left) + size(n.Right) + 1;

**n.Height = Math.*max*(height(n.Left), height(n.Right)) + 1;**

**return** n;

}

**private** **int** height(Node n)

{

**if**(n == **null**) **return** 0;

**return** n.Height;

}

**3.2.7** Add to BST a recursive method avgCompares() that computes the average number of compares required by a random search hit in a given BST (the internal path length of the tree divided by its size, plus one). Develop two implementations: a recursive method (which takes linear time and space proportional to the height), and a method like size() that adds a field to each node in the tree (and takes linear space and constant time per query).

**public** **double** avgCompares()

{

**return** root == **null** ? 0.0 : internalPathLength(root, 0) \* 1.0 / root.Size + 1;

}

**private** **int** internalPathLength(Node n, **int** depth)

{

**if**(n == **null**) **return** 0;

**return** depth + internalPathLength(n.Left, depth + 1) + internalPathLength(n.Right, depth + 1);

}

**private** **void** fixUpSize(Node startNode, Node destNode,**int** diff)

{

//fix down the size;

Node p = root;

**while**(p != destNode)

{

p.Size += diff;

**if**(p.Key.compareTo(destNode.Key) > 0)

p = p.Left;

**else**

p = p.Right;

}

}

**public** **void** putNonRec(Key key, Value value)

{

**if**(key == **null**) **return**;

Node p = **null**;

Node c = root;

**int** depth = 0;

**while**(c != **null**)

{

**int** cmp = key.compareTo(c.Key);

**if**(cmp > 0)

{

p = c;

c = c.Right;

depth++;

}

**else** **if**(cmp < 0)

{

p = c;

c = c.Left;

depth++;

}

**else**

{

c.Value = value;

**return**;

}

}

Node newNode = **new** Node(key, value);

**if**(p == **null**)

root = newNode;

**else**

{

**if**(key.compareTo(p.Key) > 0)

p.Right = newNode;

**else**

p.Left = newNode;

}

fixUpSize(root, newNode, 1);

**this**.internalPathLength += depth;

}

**public** **void** deleteNonRec(Key k)

{

**if**(root == **null**)//no bother delete anything in an empty tree;

**return**;

Node p = **null**;

Node c = root;

**int** depth = 0;

//look for node with k as its key and its parent.

**while**(c != **null** && k.compareTo(c.Key) != 0)

{

**int** cmp = k.compareTo(c.Key);

**if**(cmp > 0)

{

p = c;

c = c.Right;

depth++;

}

**else** **if**(cmp < 0)

{

p = c;

c = c.Left;

depth++;

}

}

**if**(c == **null**) **return**; //not found;

//till now, c is node to be deleted and p is its parent;

fixUpSize(root, c, -1);

//if c is a node with 0 to 1 child.

**if**(c.Left == **null** || c.Right == **null**)

{

**if**(p.Left == c)

{

**if**(c.Left == **null**) p.Left = c.Right;

**else** p.Left = c.Left;

}

**else**

{

**if**(c.Left == **null**) p.Right = c.Right;

**else** p.Right = c.Left;

}

}

**else** // 2 children, look for its successor

{

Node succParent = **null**;

Node succ = c.Right;

c.Size -= 1;

depth++;

**while**(succ.Left != **null**)

{

succ.Size -= 1;

succParent = succ;

succ = succ.Left;

depth++;

}

**if**(succParent == **null**) //just one node as its right child and its c's successor

c.Right = succ.Right;

**else**

succParent.Left = succ.Right;

c.Key = succ.Key;

c.Value = succ.Value;

}

**this**.internalPathLength -= depth;

}

**3.2.8** Write a static method optCompares() that takes an integer argument N and computes the number of compares required by a random search hit in an optimal (perfectly balanced) BST, where all the null links are on the same level if the number of links is a power of 2 or on one of two levels otherwise.

**public** **static** **double** optCompares(**int** n)

{

/\*

\* Internal path length of a perfect balancing binary search tree:

\*

\* h = floor(lgN)

\*

\* 2 items

\*

\* 1. root to the next to the last level that contains full nodes

\*

\* Sum (k from 0 to h - 1) k \* 2^k

\*

\* 2. remaining nodes

\*

\* h \* ( n - 2^h + 1)

\*

\* By calculation Sum ( k from 0 to m ) k \* 2^k = 2^(m+1) \* (m - 1) + 2

\*

\* So finally we can get the result

\*

\* h \* (n + 1) - 2^(h+1) + 2

\*

\*/

**int** h = (**int**)Math.*floor*(Math.*log*(n) / Math.*log*(2));

**return** (h \* (n+1) - (1 << (h+1)) + 2) \* 1.0 / n + 1;

}

**3.2.9** Draw all the different BST shapes that can result when N keys are inserted into an initially empty tree, for N = 2, 3, 4, 5, and 6.

Let S(n) denotes number different shapes of N nodes BST

S(0) = S(1) = 1

S(n) = Sum (k from 0 to n-1) S(k) \* S(n-1 –k) where n >= 2

S(2) = S(0)S(1) + S(1)S(0) = 2

S(3) = S(0)S(2) + S(1)S(1) + S(2)S(0) = 2 + 1 + 2 = 5

S(4) = 14 S(5) = 46 S(6) = 140

**3.2.10** Write a test client TestBST.java for use in testing the implementations of min(), max(), floor(), ceiling(), select(), rank(), delete(), deleteMin(), deleteMax(), and keys() that are given in the text. Start with the standard indexing client given on page 370. Add code to take additional command-line arguments, as appropriate.

**3.2.11** How many binary tree shapes of *N* nodes are there with height *N*? How many different ways are there to insert *N* distinct keys into an initially empty BST that result in a tree of height *N*? (See Exercise 3.2.2.)

Let Sn denotes the number of N nodes tree shapes with height N nodes then we have

S1 = 1

Sn = 2 \* Sn-1 for a height N tree with N nodes can be made by adding a height N-1 tree with N-1 nodes as its left child tree or right child tree.

Then Sn = 2^(n-1);

Exactly 2^(n-1) ways. Because to get the N node tree with height N, every time the inserted key must be the largest or smallest key in the remaining keys. The total ways to do so is

2 \* 2 \* 2 \* 2…\*2 \* 1 = 2^(n-1)

**3.2.12** Develop a BST implementation that omits rank() and select() and does not use a count field in Node.

**3.2.13** Give nonrecursive implementations of get() and put() for BST.

See 3.2.7

**3.2.14** Give nonrecursive implementations of min(), max(), floor(), ceiling(), rank(), and select().

**public** Key minNonRec()

{

**if**(root == **null**) **return** **null**;

Node n = root;

**while**(n.Left != **null**)

n = n.Left;

**return** n.Key;

}

**public** Key maxNonRec()

{

**if**(root == **null**) **return** **null**;

Node n = root;

**while**(n.Right != **null**)

n = n.Right;

**return** n.Key;

}

**public** Key floorNonRec(Key k)

{

**if**(root == **null** || k == **null**) **return** **null**;

Node n = root;

Key ret = **null**;

**while**(n != **null**)

{

**int** cmp = k.compareTo(n.Key);

**if**(cmp == 0)

**return** n.Key;

**else** **if** (cmp < 0)

n = n.Left;

**else**

{

ret = n.Key;

n = n.Right;

}

}

**return** ret;

}

**public** Key ceilingNonRec(Key k)

{

**if**(root == **null** || k == **null**) **return** **null**;

Node n = root;

Key ret = **null**;

**while**(n != **null**)

{

**int** cmp = k.compareTo(n.Key);

**if**(cmp == 0)

**return** n.Key;

**else** **if** (cmp > 0)

n = n.Right;

**else**

{

ret = n.Key;

n = n.Left;

}

}

**return** ret;

}

**public** **int** rankNonRec(Key k)

{

**if**(root == **null** || k == **null**) **return** -1;

**int** r = 0;

Node n = root;

**while**( n != **null** )

{

**int** cmp = k.compareTo(n.Key);

**if**(cmp < 0)

n = n.Left;

**else** **if**(cmp > 0)

{

r += size(n.Left) + 1;

n = n.Right;

}

**else**

{

r += size(n.Left);

**return** r;

}

}

**return** r;

}

**public** Key selectNonRec(**int** k)

{

**if**(root == **null** || k < 0) **return** **null**;

Node n = root;

**while**(n != **null**)

{

**int** leftSize = size(n.Left);

**if**(leftSize == k)

**return** n.Key;

**else** **if**(leftSize > k)

n = n.Left;

**else**

{

k -= leftSize+1;

n = n.Right;

}

}

**return** **null**;

}

**3.2.15** Give the sequences of nodes examined when the methods in BST are used to compute each of the following quantities for the tree drawn at right.

1. E, Q.

2. E, Q.

3. E, Q.

4. E, Q, J

5, 6. E, Q, J, M, T, S

**3.2.16** Define the number of nodes on the paths from the root to all null links. Prove that the difference between the external and internal path lengths in any binary tree with *N* nodes is 2*N* (see Proposition C).

This can be proved by induction.

First for a tree with N nodes, we know it has N + 1 null links. Because all nodes has totally 2 \* N links and to link N nodes we need N – 1 links, so 2 \* N – N + 1 = N + 1 null links.

Let I denotes internal length of a binary tree, E denotes external length of a binary tree.

Base case, I (1) = 0, E(1) = 2, E(1) = I(1) + 2\* 1;

Suppose this holds true for all trees with k nodes;

For the tree with n = k+1 nodes.

2 cases:

1. The root has no left (or right) child. Then it’s child tree c has k nodes

Let I(c)=m, E(c)=m + 2k.

I (t) = 0\*I(1) + I(c) + k = m + k E(t) = m + 2k + k +1 +1 = m + k + 2k + 2 = I(n) + 2n;

1. The root has 2 children. Let’s say its left child l has p nodes while right child r has q nodes.

E(l) = I(l) + 2p, E(r) =I(r) + 2q

I(t) = I(l)+ p + I(r) + q E(t) = E(l) + p + 1 + E(r) + q + 1= I(l) + 3p + I(r) + 3 q + 2 =I(t) + 2 ( p + q + 1) = I(t) + 2n.

**3.2.17** Draw the sequence of BSTs that results when you delete the keys from the tree of Exercise 3.2.1, one by one, in the order they were inserted.

**3.2.18** Draw the sequence of BSTs that results when you delete the keys from the tree of Exercise 3.2.1, one by one, in alphabetical order.

**3.2.19** Draw the sequence of BSTs that results when you delete the keys from the tree of Exercise 3.2.1, one by one, by successively deleting the key at the root.

**3.2.20** Prove that the running time of the two-argument keys() in a BST with *N* nodes is at most proportional to the tree height plus the number of keys in the range.

The keys(Key lo, Key hi) operation goes down 1 or 2 paths each time, plus 1 operation to add the node in the queue if it is in the range. So we can expect the running time is proportional to the height plus the number of keys in the range.

The worst case comes from all the nodes of the BST are in the range where the running time is ~N

**3.2.21** Add a BST method randomKey() that returns a random key from the symbol table in time proportional to the tree height, in the worst case.

**public** Key randomKey()

{

**if**(**this**.size() == 0) **return** **null**;

java.util.Random rd= **new** java.util.Random();

**return** select(rd.nextInt(**this**.size()));

}

**3.2.22** Prove that if a node in a BST has two children, its successor has no left child and its predecessor has no right child.

The successor of a node A is defined as the node whose key is smallest one in A’s right child tree that is, the smallest key that is larger than A’s key (and in A’s sub tree). If this node has a left child, then the left child will be smaller than it but also larger than A’s key. This is a contradiction.

**3.2.23** Is delete() commutative? (Does deleting x, then y give the same result as deleting y, then x?)

No. Consider tree 3, 2, 5, 4, 8, 7. Delete 3 then 5 will get a different tree with deleting 5 then 3.

**3.2.24** Prove that no compare-based algorithm can build a BST using fewer than lg(*N* !) ~ *N* lg *N* compares.

First, to build a BST with minimum compares, we can choose this method to insert nodes: for every input node, insert it with minimum possible height. This results in a perfect balanced BST because the height of nodes at leaves cannot differentiate greater than 1.

Second, we calculate compares needed to build this BST: inserting a node to a perfect balanced BST with K nodes requires floor(lg(K + 1)) + 1 compares (including compare to null link), that is ceiling(log(K + 1)) >= log(K+1). So minimum compares to insert N nodes >= Sum (k from 0 to N - 1) log(k+1) = logN! ~ NlogN.

**3.2.25** *Perfect balance.* Write a program that inserts a set of keys into an initially empty BST such that the tree produced is equivalent to binary search, in the sense that the sequence of compares done in the search for any key in the BST is the same as the sequence of compares used by binary search for the same set of keys.

**public** **static** <TKey **extends** Comparable<TKey>, TValue> BinarySearchTreeST<TKey, TValue>

buildFromArray(KeyValuePair<TKey, TValue>[] pairs)

{

BinarySearchTreeST<TKey, TValue> tree = **new** BinarySearchTreeST<TKey, TValue>();

Arrays.*sort*(pairs);

tree.root = tree.buildFromArray(pairs, 0, pairs.length - 1);

**return** tree;

}

**private** Node buildFromArray(KeyValuePair<Key, Value>[] pairs, **int** lo, **int** hi)

{

Node n = **null**;

**if**(lo <= hi)

{

**int** mid = lo + (hi - lo) / 2;

n = **new** Node(pairs[mid].Key, pairs[mid].Value);

n.Left = buildFromArray(pairs, lo, mid - 1);

n.Right = buildFromArray(pairs, mid + 1, hi);

}

**return** n;

}

**public** **static** <TKey **extends** Comparable<TKey>, TValue> BinarySearchTreeST<TKey, TValue>

buildFromArray2(KeyValuePair<TKey, TValue>[] pairs)

{

BinarySearchTreeST<TKey, TValue> tree = **new** BinarySearchTreeST<TKey, TValue>();

KeyValuePair<TKey, TValue>[] aux = (KeyValuePair<TKey, TValue>[])**new** KeyValuePair[pairs.length];

Arrays.*sort*(pairs);

*permuteToBalancedArray*(pairs, 0, pairs.length - 1, aux, **new** **int**[]{0});

**for**(KeyValuePair<TKey, TValue> pair : aux)

{

tree.put(pair.Key, pair.Value);

}

**return** tree;

}

**private** **static** <TKey **extends** Comparable<TKey>, TValue>

**void** permuteToBalancedArray(KeyValuePair<TKey, TValue>[] pairs, **int** lo, **int** hi, KeyValuePair<TKey, TValue>[] aux, **int**[] auxIndex)

{

**if**(lo <= hi)

{

**int** mid = lo + (hi - lo) / 2;

aux[auxIndex[0]++] = pairs[mid];

*permuteToBalancedArray*(pairs, lo, mid - 1, aux, auxIndex);

*permuteToBalancedArray*(pairs, mid + 1, hi, aux, auxIndex);

}

}

**3.2.28** *Sofware caching.* Modify BST to keep the most recently accessed Node in an instance variable so that it can be accessed in constant time if the next put() or get() uses the same key (see Exercise 3.1.25).

**public** **void** put(Key key, Value value)

{

**if**(cachedVisitedNode != **null** && key.equals(cachedVisitedNode.Key))

cachedVisitedNode.Value = value;

root = put(root, key, value, 0);

}

**private** Node put(Node n, Key key, Value value, **int** depth)

{

**if**(n == **null**)

{

//save it to last visited node for caching.

cachedVisitedNode = **new** Node(key, value);

**return** cachedVisitedNode;

}

**int** cmp = n.Key.compareTo(key);

**if**(cmp > 0)

{

n.Left = put(n.Left, key, value, depth + 1);

}

**else** **if** (cmp < 0)

{

n.Right = put(n.Right, key, value, depth + 1);

}

**else**

{

n.Key = key;

n.Value = value;

}

n.Size = size(n.Left) + size(n.Right) + 1;

n.Height = Math.*max*(height(n.Left), height(n.Right)) + 1;

**return** n;

}

**public** Value get(Key key)

{

**if**(cachedVisitedNode != **null** && key.equals(cachedVisitedNode.Key))

**return** cachedVisitedNode.Value;

**return** get(root, key);

}

**private** Value get(Node n, Key key)

{

**if**(n == **null**) **return** **null**;

**int** cmp = n.Key.compareTo(key);

**if**(cmp > 0) **return** get(n.Left, key);

**else** **if**(cmp < 0) **return** get(n.Right, key);

**else**

{

**this**.cachedVisitedNode = n;

**return** n.Value;

}

}

**3.2.29** *Binary tree check.* Write a recursive method isBinaryTree() that takes a Node as argument and returns true if the subtree count field N is consistent in the data structure rooted at that node, false otherwise. *Note* : This check also ensures that the data structure has no cycles and is therefore a binary tree (!).

**3.2.30** *Order check.* Write a recursive method isOrdered() that takes a Node and two keys min and max as arguments and returns true if all the keys in the tree are between min and max; min and max are indeed the smallest and largest keys in the tree, respectively; and the BST ordering property holds for all keys in the tree; false otherwise.

**3.2.31** *Equal key check.* Write a method hasNoDuplicates() that takes a Node as argument and returns true if there are no equal keys in the binary tree rooted at the argument node, false otherwise. Assume that the test of the previous exercise has passed.

**3.2.32** *Certification.* Write a method isBST() that takes a Node as argument and returns true if the argument node is the root of a binary search tree, false otherwise. *Hint*: This task is also more difficult than it might seem, because the order in which you call the methods in the previous three exercises is important.

**public** **class** BinaryNode <TKey **extends** Comparable<TKey>, TValue> {

**public** BinaryNode<TKey, TValue> Left;

**public** BinaryNode<TKey, TValue> Right;

**public** TKey Key;

**public** TValue Value;

**private** **int** size;

**public** BinaryNode(TKey key, TValue value)

{

**this**.Key = key;

**this**.Value = value;

size = 1;

}

**public** **static** <TKey **extends** Comparable<TKey>, TValue>

**boolean** isBinaryTree(BinaryNode<TKey, TValue> n)

{

**return** n == **null**

||

(

n.size == 1 + (n.Left == **null** ? 0 : n.Left.size) + (n.Right == **null** ? 0 : n.Right.size)

&& *isBinaryTree*(n.Left)

&& *isBinaryTree*(n.Right)

);

}

**private** **static** <TKey **extends** Comparable<TKey>, TValue>

BinaryNode<TKey, TValue> maxNode(

BinaryNode<TKey, TValue> n

) {

**if**(n == **null**) **return** **null**;

BinaryNode<TKey, TValue> p = n;

**while**(p. Right != **null**)

p = p.Right;

**return** p;

}

**private** **static** <TKey **extends** Comparable<TKey>, TValue>

BinaryNode<TKey, TValue> minNode(

BinaryNode<TKey, TValue> n) {

**if**(n == **null**) **return** **null**;

BinaryNode<TKey, TValue> p = n;

**while**(p.Left != **null**)

p = p.Left;

**return** p;

}

**public** **static** <TKey **extends** Comparable<TKey>, TValue>

**boolean** isOrdered(BinaryNode<TKey, TValue> n, TKey min, TKey max)

{

**if**(n == **null**) **return** **true**;

**int** minCmp = n.Key.compareTo(min);

**int** maxCmp = n.Key.compareTo(max);

**return**

minCmp >= 0 && maxCmp <= 0

&& *isOrdered*(n.Left, min, n.Key)

&& *isOrdered*(n.Right, n.Key, max);

}

**public** **static** <TKey **extends** Comparable<TKey>, TValue>

**boolean** hasNoDuplicate(BinaryNode<TKey, TValue> n)

{

**if**(n == **null**) **return** **true**;

**for**(**int** i = 1 ; i < n.size ; i++)

**if**(n.select(n, i).equals(n.select(n, i - 1)))

**return** **false**;

**return** **true**;

}

**public** **static** <TKey **extends** Comparable<TKey>, TValue>

**boolean** isBST(BinaryNode<TKey, TValue> n)

{

**return** *isBinaryTree*(n)

&&

*isOrdered*(n, *minNode*(n).Key, *maxNode*(n).Key)

&&

*hasNoDuplicate*(n);

}

**private** TKey select(BinaryNode<TKey, TValue> n, **int** i)

{

**if**(n == **null**) **return** **null**;

**int** leftSize = n.Left == **null** ? 0 : n.Left.size;

**if**(leftSize == i) **return** n.Key;

**if**(leftSize > i) **return** select(n.Left, i);

**return** select(n.Right, i - leftSize - 1);

}

}

**3.2.33** *Select/rank check.* Write a method that checks, for all i from 0 to size()-1,whether i is equal to rank(select(i)) and, for all keys in the BST, whether key is equal to select(rank(key)).

**public** **boolean** isRankConsistent()

{

**if**(root == **null**) **return** **true**;

**for**(**int** i = 0; i < root.Size ; i++)

{

**if**(i != rank(select(i)))

**return** **false**;

}

**for**(Key key : keys())

{

**if**(!key.equals(select(rank(key))))

**return** **false**;

}

**return** **true**;

}

**3.2.34** *Threading.* Your goal is to support an extended API ThreadedST that supports the following additional operations in constant time:

Key next(Key key) *key that follows* key *(*null *if* key *is the maximum)*

Key prev(Key key) *key that precedes* key *(*null *if* key *is the minimum)*

To do so, add fields pred and succ to Node that contain links to the predecessor and successor nodes, and modify put(), deleteMin(), deleteMax(), and delete() to maintain these fields.

**public** **void** putWithThreading(Key key, Value value)

{

**if**(key == **null**) **return**;

Node p = **null**;

Node c = root;

**int** depth = 0;

**while**(c != **null**)

{

**int** cmp = key.compareTo(c.Key);

**if**(cmp > 0)

{

p = c;

c = c.Right;

depth++;

}

**else** **if**(cmp < 0)

{

p = c;

c = c.Left;

depth++;

}

**else**

{

c.Value = value;

**return**;

}

}

Node newNode = **new** Node(key, value);

**if**(p == **null**)

root = newNode;

**else**

{

**if**(key.compareTo(p.Key) > 0)

{

p.Right = newNode;

//fix threading

newNode.Succ = p.Succ;

p.Succ = newNode;

newNode.Pred = p;

**if**(newNode.Succ != **null**)

newNode.Succ.Pred = newNode;

}

**else**

{

p.Left = newNode;

//fix threading

newNode.Pred = p.Pred;

p.Pred = newNode;

newNode.Succ = p;

**if**(newNode.Pred != **null**)

newNode.Pred.Succ = newNode;

}

}

fixUpSize(root, newNode, 1);

**this**.internalPathLength += depth;

}

**public** **void** deleteWithThreading(Key k)

{

**if**(root == **null**)//no bother delete anything in an empty tree;

**return**;

Node p = **null**;

Node c = root;

**int** depth = 0;

//look for node with k as its key and its parent.

**while**(c != **null** && k.compareTo(c.Key) != 0)

{

**int** cmp = k.compareTo(c.Key);

**if**(cmp > 0)

{

p = c;

c = c.Right;

depth++;

}

**else** **if**(cmp < 0)

{

p = c;

c = c.Left;

depth++;

}

}

**if**(c == **null**) **return**; //not found;

//till now, c is node to be deleted and p is its parent;

fixUpSize(root, c, -1);

//if c is a node with 0 to 1 child.

**if**(c.Left == **null** || c.Right == **null**)

{

**if**(p.Left == c)

{

**if**(c.Left == **null**) p.Left = c.Right;

**else** p.Left = c.Left;

}

**else**

{

**if**(c.Left == **null**) p.Right = c.Right;

**else** p.Right = c.Left;

}

//fix threading.

**if**(c.Pred != **null**)

{

c.Pred.Succ = c.Succ;

c.Succ = **null**;

}

**if**(c.Succ != **null**)

{

c.Succ.Pred = c.Pred;

c.Pred = **null**;

}

}

**else** // 2 children, look for its successor

{

Node succParent = **null**;

Node succ = c.Right;

c.Size -= 1;

depth++;

**while**(succ.Left != **null**)

{

succ.Size -= 1;

succParent = succ;

succ = succ.Left;

depth++;

}

**if**(succParent == **null**) //just one node as its right child and its c's successor

c.Right = succ.Right;

**else**

succParent.Left = succ.Right;

c.Key = succ.Key;

c.Value = succ.Value;

//fix threading

c.Succ = succ.Succ;

succ.Succ = **null**;

**if**(c.Succ != **null**)

c.Succ.Pred = c;

}

**this**.internalPathLength -= depth;

}

**3.2.35** *Refined analysis.* Refine the mathematical model to better explain the experimental results in the table given in the text. Specifically, show that the average number of compares for a successful search in a tree built from random keys approaches the limit 2 ln *N* \_ 2 *–* 3 \_ 1.39 lg *N* ***–*** 1.85 as *N* increases, where \_ .57721... is *Euler’s constant*.

Let Cn be the internal path length of all nodes of a tree random built by n nodes.

Then the average compares needed = 1 + Cn/n;

Cn = n – 1 + (C0 + Cn-1) / n + (C1 + Cn-2) / n + … + (Cn-1 + C0) / n

Cn = n – 1+ 2 \* Sum(k from 0 to n-1) (Ck) / n

* nCn = n(n-1) + 2 \* Sum(k from 0 to n-1) (Ck) ----------- (1)
* (n-1)Cn-1 = (n-1)(n-2) + 2 \* Sum(k from 0 to n-2) (Ck) --(2)
* (1) – (2)
* nCn – (n-1)Cn-1 = 2(n-1) + 2Cn-1
* nCn = (n+1)Cn-1 + 2(n-1)
* Cn/(n+1) = Cn-1/n + 2(n-1)/n(n+1) --- (3)
* Let Fn = Cn/(n+1)
* From (3)
* Fn = Fn-1 + 4/(n+1) – 2/n

= F1 + 4 \* Sum (k from 3 to n+1) (1/k) – 2 \* Sum (k from 2 to n) (1/k)

= -2 + 4 \* Sum(k from 2 to n)(1/k) + 4/(n+1) – 2 \* Sum (k from 2 to n) (1/k)

= 2 \* Sum(k from 2 to n) (1/k) – 2 + 4/(n+1)

= 2 \* Sum(k from 1 to n) (1/k) – 4 + 4/(n+1)

* average compares Cn/n + 1 = (n+1)/n \* (2 \* Sum(k from 1 to n) (1/k) – 4 + 4/(n+1)) + 1 --(4)
* When n is efficiently large, Cn/n + 1 = 2 \* ln(n) + 2 \* gamma – 4 + 1 = 2ln(n) + 2 \* gamma – 3
* Where gamma is Euler’s constant = 0.5772.
* 1.39 lg(n) - 1.85

**3.2.36** *Iterator.* Is it possible to write a nonrecursive version of keys() that uses space proportional to the tree height (independent of the number of keys in the range)?

Answer is “no” for general cases. We need either function call stack (for recursive traverse) or our own stack (for non-recursive traverse) to trace the node to be visited. In the worst case this stack length is proportional to n.

However, the answer is “yes” for tree with threading, because

**3.2.37** *Level-order traversal.* Write a method printLevel() that takes a Node as argument and prints the keys in the subtree rooted at that node in level order (in order of their distance from the root, with nodes on each level in order from left to right). *Hint* : Use a Queue.

**public** **void** levelOrderTraverse()

{

**if**(root == **null**) **return**;

Queue<Node> q = **new** LinkedList<Node>();

q.add(root);

**while**(!q.isEmpty())

{

Node n = q.remove();

System.*out*.println("LOT: " + n.Value);

**if**(n.Left != **null**) q.add(n.Left);

**if**(n.Right != **null**) q.add(n.Right);

}

}