**3.1.1** Write a client that creates a symbol table mapping letter grades to numerical scores, as in the table below, then reads from standard input a list of letter grades and computes and prints the GPA (the average of the numbers corresponding to the grades).

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

BinarySearchST<String, Double> st = **new** BinarySearchST<String, Double>();

st.put("A+", 4.33);

st.put("A", 4.00);

st.put("A-", 3.67);

st.put("B+", 3.33);

st.put("B", 3.00);

st.put("B-", 2.67);

st.put("C+", 2.33);

st.put("C", 2.00);

st.put("C-", 1.67);

st.put("D", 1.00);

st.put("F", 0.00);

}

**3.1.2** Develop a symbol-table implementation ArrayST that uses an (unordered) array as the underlying data structure to implement our basic symbol-table API.

**public** **class** ArrayST <TKey, TValue> **implements** SymbolTable<TKey, TValue> {

**private** TKey[] keys;

**private** TValue[] values;

**private** **int** N = 0;

**private** **int** capicity = 16;

**public** ArrayST()

{

keys = (TKey[])**new** Object[capicity];

values = (TValue[])**new** Object[capicity];

}

**public** ArrayST(**int** capicity)

{

**this**.capicity = capicity;

keys = (TKey[])**new** Object[capicity];

values = (TValue[])**new** Object[capicity];

}

**public** **void** put(TKey key, TValue value) {

keys[N] = key;

values[N] = value;

N++;

**if**(N == capicity)

resize(2 \* capicity);

}

**protected** **void** resize(**int** newSize)

{

TKey[] newKeys = (TKey[])**new** Object[newSize];

TValue[] newValues = (TValue[])**new** Object[newSize];

**for**(**int** i = 0 ; i < N ; i++){

newKeys[i] = keys[i];

newValues[i] = values[i];

}

keys = newKeys;

values = newValues;

capicity = newSize;

}

**public** TValue get(TKey key) {

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

**if**(keys[i].equals(key))

**return** values[i];

**return** **null**;

}

**public** **void** delete(TKey key) {

**int** i = 0;

**for**( ; i < N && !keys[i].equals(key); i++)

;

**if**(i == N) **return**;

**while**(i < N)

{

keys[i] = keys[i + 1];

values[i] = values[i + 1];

i++;

}

**if**(N < capicity / 4)

resize(capicity / 2);

}

**public** **boolean** contains(TKey key) {

**return** get(key) != **null**;

}

**public** **int** size() {

**return** N;

}

/\*\*

\* **@param** args

\*/

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

// **TODO** Auto-generated method stub

ArrayST<String, Integer> at = **new** ArrayST<String, Integer>(1);

at.put("1", 1);

at.put("2", 1);

at.delete("2");

System.*out*.println(at.get("1"));

}

}

**3.1.3** Develop a symbol-table implementation OrderedSequentialSearchST that uses an ordered linked list as the underlying data structure to implement our ordered symbol-table API.

**package** symbolTables;

**import** java.util.LinkedList;

**import** java.util.Queue;

**public** **class** OrderedSequentialSearchST<TKey **extends** Comparable<TKey>, TValue> **implements** OrderedSymbolTable<TKey, TValue> {

**private** **class** Node

{

TKey key;

TValue value;

Node next;

**public** Node(TKey key, TValue value, Node next)

{

**this**.key = key;

**this**.value = value;

**this**.next = next;

}

}

**private** Node first;

**private** **int** N = 0;

/\*\*

\* **@param** args

\*/

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

// **TODO** Auto-generated method stub

}

**public** Iterable<TKey> keys()

{

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

Node p = first;

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

{

q.add(p.key);

p = p.next;

}

**return** q;

}

**public** **void** put(TKey key, TValue value) {

Node prev = **null**;

Node cur = first;

**while**(cur != **null** && cur.key.compareTo(key) < 0)

{

prev = cur;

cur = cur.next;

}

**if**(N == 0 || cur == first)

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

**else** **if** (cur == **null** || cur.key.compareTo(key) > 0)

prev.next = **new** Node(key, value, prev.next);

**else** //cur.key == key

{

cur.value = value;

**return**;

}

N++;

}

**public** TValue get(TKey key) {

Node n = first;

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

**if**(n.equals(key))

**return** n.value;

**return** **null**;

}

**public** **void** delete(TKey key) {

**if**(N == 0) **return**;

Node prev = **null**;

Node cur = first;

**while**(cur != **null** && cur.key.compareTo(key) < 0)

{

prev = cur;

cur = cur.next;

}

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

**if**(cur.key.compareTo(key) == 0)

{

prev = prev.next;

N--;

}

}

**public** **boolean** contains(TKey key) {

**return** get(key) != **null**;

}

**public** **int** size() {

**return** N;

}

**public** TKey min() {

**return** first != **null** ? first.key : **null**;

}

**public** TKey max() {

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

Node p = first;

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

p = p.next;

**return** p.key;

}

**public** TKey floor(TKey key) {

**if**(N == 0 || first.key.compareTo(key) > 0) **return** **null**;

Node p = first;

**if**(p.next != **null** && p.next.key.compareTo(key) < 0)

p = p.next;

**return** p.key;

}

**public** TKey ceiling(TKey key) {

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

**if**(first.key.compareTo(key) > 0) **return** first.key;

Node p = first ;

**while**(p != **null** && p.key.compareTo(key) < 0)

p = p.next;

**return** p == **null** ? **null** : p.key;

}

**public** TKey select(**int** i) {

**if**(N == 0|| i >= N) **return** **null**;

Node p = first;

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

p = p.next;

**return** p.key;

}

**public** **int** rank(TKey key) {

**if**(N == 0) **return** 0;

**int** rank = 0;

**for**(Node p = first; rank < N && key.compareTo(p.key) > 0; rank++)

p = p.next;

**return** rank;

}

**public** **void** deleteMin() {

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

**else** first = first.next;

N--;

}

**public** **void** deleteMax() {

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

**if**(N == 1) first = **null**;

**else**

{

Node p = first;

**for**(**int** i = 0 ; i < N - 1 ; i++)

p = p.next;

**if**(p == first) first.next = **null**;

**else** p.next = **null**;

N--;

}

}

}

**3.1.4** Develop Time and Event ADTs that allow processing of data as in the example illustrated on page 367.

**3.1.5** Implement size(), delete(), and keys() for SequentialSearchST.

**public** **class** SequentialSearchST<Key, Value> **implements** SymbolTable<Key, Value> {

**private** **class** Node

{

**public** Key Key;

**public** Value Value;

**public** Node Next;

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

{

Key = key;

Value = value;

Next = next;

}

}

**private** Node first;

**private** **int** N = 0;

**public** SequentialSearchST()

{

}

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

{

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

Node node = getNode(key);

**if**(node == **null**)

{

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

N++;

}

**else**

node.Value = value;

}

**public** Value get(Key key)

{

Node n = getNode(key);

**return** n == **null** ? **null** : n.Value;

}

**public** **void** delete(Key key)

{

**if**(first.Key.equals(key))

{

first = first.Next;

}

**else**

{

Node f = first;

**while**(f.Next != **null** && !f.Next.Key.equals(key))

f = f.Next;

**if**(f.Next == **null**)

**return**;

f.Next = f.Next.Next;

}

N--;

}

**public** **int** size()

{

**return** N;

}

**public** **boolean** isEmpty()

{

**return** N == 0;

}

**private** Node getNode(Key key)

{

Node t = first;

**while**(t != **null** && !t.Key.equals(key))

t = t.Next;

**return** t;

}

**public** Iterable<Key> keys()

{

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

Node n = first;

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

{

q.add(n.Key);

n = n.Next;

}

**return** q;

}

**public** **boolean** contains(Key key) {

**return** get(key) != **null**;

}

/\*\*

\* **@param** args

\*/

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

// **TODO** Auto-generated method stub

SequentialSearchST<String, Integer> st = **new** SequentialSearchST<String, Integer>();

st.put("Felix Wang", 100);

st.put("Neil", 55);

st.put("Zirl", 77);

System.*out*.println(st.get("Zirl"));

System.*out*.println(st.get("Neil"));

st.delete("Felix Wang");

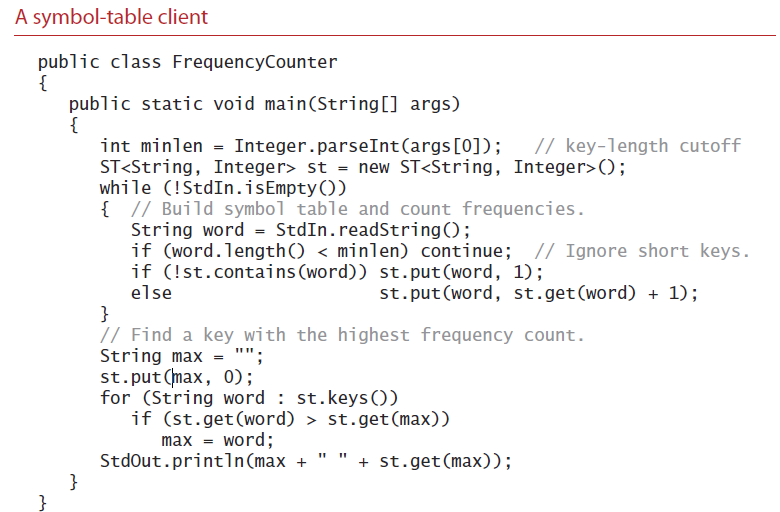
System.*out*.println(st.get("Felix Wang"));

st.delete("Zirl");

System.*out*.println(st.get("Zirl"));

}

**3.1.6** Give the number of calls to put() and get() issued by FrequencyCounter, as a function of the number *W* of words and the number *D* of distinct words in the input.



put() = D + 1

get() = W + 2 \* D + 1

**3.1.7** What is the average number of distinct keys that FrequencyCounter will find among *N* random nonnegative integers less than 1,000, for *N*=10, 102, 103, 104, 105, and 106?

**3.1.8** What is the most frequently used word of ten letters or more in *Tale of Two Cities*?

Monseigneur

**3.1.9** Add code to FrequencyCounter to keep track of the *last* call to put(). Print the last word inserted and the number of words that were processed in the input stream prior to this insertion. Run your program for tale.txt with length cutoffs 1, 8, and 10.

**3.1.10** Give a trace of the process of inserting the keys E A S Y Q U E S T I O N into an initially empty table using SequentialSearchST. How many compares are involved?

E

E A + 1

E A S + 2

E A S Y +3

E A S Y Q +4

E A S Y Q U +5

E A S Y Q U + 1

E A S Y Q U + 3

E A S Y Q U T + 6

E A S Y Q U T I + 7

E A S Y Q U T I O +8

E A S Y Q U T I O N +9

Total 49.

**3.1.11** Give a trace of the process of inserting the keys E A S Y Q U E S T I O N into an initially empty table using BinarySearchST. How many compares are involved?

E

A E + 1

A E S + 2

A E S Y + 2

A E Q S Y + 3

A E Q S U Y + 3

A E Q S U Y + 3

A E Q S U Y + 3

A E Q S T U Y + 3

A E I Q S T U Y + 3

A E I O Q S T U Y + 3

A E I N O Q S T U Y + 4

Total: 30.

**3.1.12** Modify BinarySearchST to maintain one array of Item objects that contain keys and values, rather than two parallel arrays. Add a constructor that takes an array of Item values as argument and uses mergesort to sort the array.

**3.1.13** Which of the symbol-table implementations in this section would you use for an application that does 10 3 put() operations and 10 6 get() operations, randomly intermixed? Justify your answer.

BinarySearchST would be the choice because its search operation takes less than lgN + 1 compares to get the result.

At the worst case the totally compare used 5 \* 10^5 + 6 \* lg10 + 1

**3.1.14** Which of the symbol-table implementations in this section would you use for an application that does 10 6 put() operations and 10 3 get() operations, randomly intermixed? Justify your answer.

SequentialSearchST would be better.

At the worst case, total compares = 10^6 + 10^9

Or suppose there is 1 get() every 1000 put() and the key to search is around half way of the array.

Then the totally compares = 10^6 + 1/2 \* (1000 + 2000 +… + 10^6) = 10^6 + 500 \* 500 \* 1001 ~= 10^6 + 2.5 \* 10^8

While if it is BinarySearchST, total compares = 1/2 \* (1 + 2 + 3 + …10^6) + (lg1000 + lg2000 + ... + lg10^6) + 10^3 ~= 1/4 \* (10^6 + 1) \* 10 ^ 6 + (3\* lg10 \* 10^3 + 3000 lg10) + 10^3 ~= 2.5 \*10^11 + 6000 lg10 + 1000 >> 10^9

**3.1.15** Assume that searches are 1,000 times more frequent than insertions for a BinarySearchST client. Estimate the percentage of the total time that is devoted to insertions, when the number of searches is 103, 10 6, and 10 9.

**3.1.16** Implement the delete() method for BinarySearchST.

**public** **void** delete(Key key)

{

**int** k = rank(key);

**if**(keys[k].equals(key))

{

**for**(**int** i = k ; i < N - 1 ; i++)

{

keys[i] = keys[i + 1];

values[i] = values[i+1];

}

N--;

**if**(N < capicity / 4)

resize(capicity / 2);

}

}

**3.1.17** Implement the floor() method for BinarySearchST.

**public** Key floor(Key key)

{

**int** k = rank(key);

**if**(keys[k].equals(key)) **return** keys[k];

**else** **if**(k > 0) **return** keys[k - 1];

**else** **return** **null**;

}

**3.1.18** Prove that the rank() method in BinarySearchST is correct.

The basic observation is that for a sorted array, the index of array, say i, accidentally represents the items in array that less than or equal to a[i]. This can be proved by simple induction.

So when search hit by rank(), the correctness is straight forward, we returned the index of the found key which accidentally equal to the number of items in the array less than or equal to given key.

When search is missed, we need to prove it is correct to return ‘low’ index.

Consider the second last search of a search miss when low == high (and mid == low). If the given key is larger than mid (== low) eventually we will return low + 1 which is correct. Otherwise the low is returned which is also correct.

Either way rank() is correct.

**3.1.19** Modify FrequencyCounter to print all of the values having the highest frequency of occurrence, not just one of them. *Hint* : Use a Queue.

PriorityQueue<Stat> pq = **new** PriorityQueue<Stat>();

pq.add(**new** Stat("",0));

**for**(String word : st.keys())

{

pq.add(**new** Stat(word, st.get(word)));

**if**(pq.size() > 10)

pq.remove();

}

**for**(**int** i = 0 ; i < 10 ; i++)

{

Stat s = pq.remove();

StdOut.*println*(s.word + " " + s.freq);

}

**class** Stat **implements** Comparable<Stat>

{

String word;

Integer freq;

**public** **int** compareTo(Stat rhs) {

**return** **this**.freq - rhs.freq;

}

**public** Stat(String s, Integer freq)

{

word = s;

**this**.freq = freq;

}

}

**3.1.20** Complete the proof of Proposition B (show that it holds for all values of *N*). *Hint* : Start by showing that *C*(*N*) is monotonic: *C*(*N*) \_ *C*(*N*+1) for all *N* > 0.

First we prove for any N > 0 we have C(N) <= C(N+1).

Prove by induction, base case:

C(1) = 1, C(2) = 2

The inequality holds.

Suppose it holds for any N <= m. Consider N > m but < 2m.

If N is an even number

C(N) = C(N/2) + 1 or C(N/2 – 1) + 1)

C(N + 1) = C (N / 2) + 1

Because N/2 >= N/2 or N/2 -1 we have C(N/2) >= C(N/2) or C(N/2-1) (by induction).

In either case C(N+1) >= C(N)

If N is an odd number

C(N) = C((N – 1) / 2) + 1

C(N+1) = C((N+1)/2) + 1 or C((N-1)/2) + 1.

Both (N+1)/2 or (N-1)/2 >= (N – 1) / 2.

In either case C(N+1) >= C(N).

So we have C(N+1) >= C(N) for any positive number N.

We need also prove another result: when N = 2^m where m >= 1, C(N) > C(N-1).

Base case is C(1) < C(2).

Suppose for m <= k this inequality holds true.

See m = k + 1, C(N) = C(2^(m+1)) = C(2^m) + 1, C(N-1) = C((N-2)/2) + 1 = C(N/2 – 1) + 1 = C(2^m – 1) + 1. By induction hypothesis, C(2^m) > C(2^m -1) we have C(N) > C(N-1)

So we have C(N) > C(N-1) when N = 2^m where m >= 1

Consider 2^m <= N < 2^(m+1) => m <= lgN < m + 1 => floor(lgN) + 1 = m + 1

Then m + 1<= C(N) < m + 2 => C(N) = m + 1.

Finally we have for any N > 0, C(N) = floor(lgN) + 1

**3.1.21** *Memory usage.* Compare the memory usage of BinarySearchST with that of SequentialSearchST for *N* key-value pairs, under the assumptions described in Section 1.4. Do not count the memory for the keys and values themselves, but do count references to them. For BinarySearchST, assume that array resizing is used, so that the array is between 25 percent and 100 percent full.

BinarySearchST with N items:

Min: 16 (object overhead) + 4 (int size) + 24 (Keys array overhead) + 8 \* N (Keys array references) + 24 (Values array overhead) + 8 \* N (Values array references) = 58 + 16N = 64 + 16N (padding to 8)

Max = 58 + 64N = 64 + 64N (padding to 8)

SequentialSearchST: 16 (object overhead) + (8 (Key) + 8(Value) + 8 (Next) + 8 (outer ref) + 16) (Node inner class) \* N + 4 (int size) = 20 + 48N = 24 + 48N (padding to 8).

**3.1.22** *Self-organizing search.* A self-organizing search algorithm is one that rearranges items to make those that are accessed frequently likely to be found early in the search. Modify your search implementation for Exercise 3.1.2 to perform the following action on every search hit: move the key-value pair found to the beginning of the list, moving all pairs between the beginning of the list and the vacated position to the right one position. This procedure is called the *move-to-front* heuristic.

**public** **void** put(TKey key, TValue value) {

//double the size

**if**(N == capicity)

resize(2 \* capicity);

//found.

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

{

**if**(keys[i].equals(key))

{

values[i] = value;

moveToFront(i);

**return**;

}

}

//not found, insert;

keys[N] = key;

values[N] = value;

N++;

}

**private** **void** moveToFront(**int** pos)

{

TKey key = keys[pos];

TValue value = values[pos];

**for**(**int** k = pos - 1 ; k >= 0 ; k--)

{

keys[k + 1] = keys[k];

values[k + 1] = values[k];

}

keys[0] = key;

values[0] = value;

}

**public** **void** delete(TKey key) {

**int** i = 0;

**for**( ; i < N && !keys[i].equals(key); i++)

;

**if**(i == N) **return**;

**while**(i + 1 < N)

{

keys[i] = keys[i + 1];

values[i] = values[i + 1];

i++;

}

N--;

**if**(N < capicity / 4)

resize(capicity / 2);

}

**3.1.23** *Analysis of binary search.* Prove that the maximum number of compares used for a binary search in a table of size *N* is precisely the number of bits in the binary representation of *N*, because the operation of shifting 1 bit to the right converts the binary representation of *N* into the binary representation of ⎣*N*/2⎦.

All we need to prove is the number of bits in the binary representation of N just equals to floor(lgN) + 1

Let B(N) represents the number of bits in binary representation.

This can be done by induction. N = 1 is a trivial case. Suppose for N <= k the equation holds true.

B(n) = floor(lg(n)) + 1 where n <= k

Consider N = n > k (but < 2 \* k)

B(n) = 1 + B(floor(n/2)) = 1 + floor(log(floor(n/2)) + 1 = floor(log(2\*floor(n/2))) + 1

If n is even 2\* floor(n/2) = n; so the equation holds true.

If n is odd 2\* floor(n/2) = n - 1. So B(n) = floor(log(n-1)) + 1

Let’s say 2^m < n < 2^(m+1) => m < log(n) < m + 1

2^m <= n – 1 < 2^(m+1) => m <= log(n-1) < m +1

When floor is applied, they both equal to m.

That is to say anyway, B(n) = floor(log(n)) + 1 for n > k.

That is to say B(n) = floor(log(n)) + 1 for any positive number n.

**3.1.24** *Interpolation search.* Suppose that arithmetic operations are allowed on keys (for example, they may be Double or Integer values). Write a version of binary search that mimics the process of looking near the beginning of a dictionary when the word begins with a letter near the beginning of the alphabet. Specifically, if *kx* is the key value sought, *klo* is the key value of the first key in the table, and *khi* is the key value of the last key in the table, look first ⎣(*kx* \_ *klo*)/(*khi* \_ *klo*)⎦-way through the table, not halfway. Test your implementation against BinarySearchST for FrequencyCounter using SearchCompare.

**private** **int** rank(Double d, **int** low, **int** high)

{

**if**(low <= high)

{

**double** r = 0;

**if**(low < high)

r = (d - keys[low])/(keys[high] - keys[low]);

**if**(r > 1) **return** high + 1;

**if**(r < 0) **return** low;

**int** p = low + (**int**)((high - low) \* r);

**int** cmp = d.compareTo(keys[p]);

**if**(cmp < 0)

**return** rank(d, low, p - 1);

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

**return** rank(d, p + 1, high);

**else**

**return** p;

}

**return** low;

}

**3.1.25** *Software caching.* Since the default implementation of contains() calls get(), the inner loop of FrequencyCounter

if (!st.contains(word)) st.put(word, 1);

else st.put(word, st.get(word) + 1);

leads to two or three searches for the same key. To enable clear client code like this without sacrificing efficiency, we can use a technique known as *software caching*, where we save the location of the most recently accessed key in an instance variable. Modify SequentialSearchST and BinarySearchST to take advantage of this idea.

BinarySearchST

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

{

**if**(N == capicity)

resize(2 \* capicity);

**if**(N == 0 || key.compareTo(keys[N - 1]) > 0)

{

keys[N] = key;

values[N] = value;

cachedIndex = N;

N++;

**return**;

}

**int** k = rank(key);

cachedIndex = k;

**if**(k < N && keys[k].equals(key))

{

values[k] = value;

}

**else**

{

**for**(**int** i = N - 1 ; i >= k ; i--)

{

keys[i + 1] = keys[i];

values[i + 1] = values[i];

}

keys[k] = key;

values[k] = value;

N++;

}

}

**public** Value get(Key key)

{

**if**(cachedIndex != -1 && key.equals(keys[cachedIndex]))

**return** values[cachedIndex];

**int** k = rank(key);

**if**(k < N && keys[k].equals(key))

{

cachedIndex = k;

**return** values[k];

}

**else**

**return** **null**;

}

**public** **void** delete(Key key)

{

**int** k = 0;

**if**(cachedIndex != -1 && key.equals(keys[cachedIndex]))

{

k = cachedIndex;

cachedIndex = -1;

}

**else**

k = rank(key);

**if**(keys[k].equals(key))

{

**for**(**int** i = k ; i < N - 1 ; i++)

{

keys[i] = keys[i + 1];

values[i] = values[i+1];

}

N--;

**if**(N < capicity / 4)

resize(capicity / 2);

}

}

**3.1.26** *Frequency count from a dictionary.* Modify FrequencyCounter to take the name of a dictionary file as its argument, count frequencies of the words from standard input that are also in that file, and print two tables of the words with their frequencies, one sorted by frequency, the other sorted in the order found in the dictionary file.

**3.1.27** *Small tables.* Suppose that a BinarySearchST client has *S* search operations and *N* distinct keys. Give the order of growth of *S* such that the cost of building the table is the same as the cost of all the searches.

At the worst case, building a BinarySearchST with N distinct keys needs N(N-1)/2 compares while every search takes floor(lgN) + 1 compares

So S = N\*(N-1) / 2\*(floor(lgN) + 1) ~= N^2/lgN^2

**3.1.28** *Ordered insertions.* Modify BinarySearchST so that inserting a key that is larger than all keys in the table takes constant time (so that building a table by calling put() for keys that are in order takes linear time).

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

{

**if**(N == 0 || key.compareTo(keys[N - 1]) > 0)

{

keys[N] = key;

values[N] = value;

N++;

**return**;

}

**int** k = rank(key);

**if**(k < N && keys[k].equals(key))

{

values[k] = value;

}

**else**

{

**for**(**int** i = N - 1 ; i >= k ; i--)

{

keys[i + 1] = keys[i];

values[i + 1] = values[i];

}

keys[k] = key;

values[k] = value;

N++;

**if**(N == capicity)

resize(2 \* capicity);

}

}

**3.1.29** *Test client.* Write a test client TestBinarySearch.java for use in testing the implementations of min(), max(), floor(), ceiling(), select(), rank(), 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.

**3.1.30** *Certification.* Add assert statements to BinarySearchST to check algorithm invariants and data structure integrity after every insertion and deletion. For example, every index i should always be equal to rank(select(i)) and the array should always be in order.

**public** **static** <TKey **extends** Comparable<TKey>, TValue> **boolean** isValid(BinarySearchST<TKey, TValue> bsst)

{

**for**(**int** i = 0 ; i < bsst.N ; i++)

{

**if**( i < bsst.N - 1)

**if**( bsst.keys[i].compareTo(bsst.keys[i+1]) > 0)

**return** **false**;

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

**return** **false**;

}

**return** **true**;

}