Skip Lists

Daniel F. Savarese

Last Updated: 2012-08-10

Note

This article was published originally in the April 2001 issue of *Java Pro* with the unenlightening title "The Sort-ed Details." It was written when JDK 1.3 was the latest available Java release, years before `ConcurrentSkipListMap` appeared in JDK 1.6. The example code is instructional and does not implement all of the Java Collections interfaces required for a production-use implementation. Even though you will want to use `ConcurrentSkipListMap` and `ConcurrentSkipListSet` instead of implementing your own skip list, the article remains useful for those who would like an overview of how a skip list works.

Changes. The code examples have been updated to use generics, an exercise of little value that makes the code harder to read. I have also added timings for `ConcurrentSkipListMap` to the driver program so you can see that the performance properties described in the article are implementation-independent. Depending on the hardware and JVM used—and the order in which the tests are run—`ConcurrentSkipListMap` executes `put`, `get`, and `remove` slower or faster than the article's code. In all cases, the execution times are the same base-2 order of magnitude (i.e., less than a factor of two difference) as the example code, which is two to three times slower than `TreeMap` for the tests performed.

Sorting and Searching

Sorting continues to be one of the most common operations performed by computer programs. Java programs are no exception. The Collections Framework recognizes the fundamental need for ordering computer data by providing the `Comparable` and `Comparator` interfaces. If you cannot determine the natural ordering between two objects, you cannot sort a collection of objects. The `Comparable` interface allows an object to control its ordering by implementing the `compareTo(Object)` method. The `Comparator` interface allows you to implement a `compare(Object,Object)` method that returns the ordering of an arbitrary pair of objects that may not have been designed with ordering in mind.

A common approach to ordering data is to store the data in a container, be it an array or a list of some kind, and sort it as needed. Arrays are often difficult to work with when you don't know how many array elements you will need to store all of your data. The `Vector` and `ArrayList` classes—each of which implements the `java.util.List` interface—resolve this difficulty by providing dynamically growing array-based storage. Array data can be sorted with `java.util.Arrays.sort()` and `List` data can be sorted with `java.util.Collections.sort()`. Once the data is sorted, you can search it relatively efficiently in $O(\log_2(n))$ time using a binary search algorithm, implemented by `Arrays.binarySearch` and `Collections.binarySearch`.

Searching for data in linear storage containers, such as arrays, is inefficient. In the worst case it requires the examination of every item of data. If the data is sorted, you can take advantage of a binary search, but you still have to pay the cost of sorting the data. For dynamically changing collections of data, that cost is not acceptable. An alternative is to use a data structure that maintains the ordering between its elements as those elements are inserted and removed.

Mappings, such as those represented by the `java.util.Map` interface, associate a set of keys with a set of values. Sometimes all we care about is the ability to quickly retrieve a value associated with a given key. Should we want to access the mapped values based on the natural ordering of their keys, or define a new submapping over a range of key values, we have to sort the keys. In these cases, an ordered mapping, represented by the `java.util.SortedMap` interface, is more appropriate. The `java.util.TreeMap` class implements the `SortedMap` interface using a red-black tree, a type of balanced binary search tree that allows both efficient searching and ordered traversal. There are many alternative data structures that could have been used for

[1] https://www.savarese.org/

a default `SortedMap` implementation in the Collections Framework. Implementing one of them may shed some light on why the red-black tree was chosen over the available alternatives.

Enter the Skip List

Figure 1 contains the source for an implementation of a probabilistically balanced container called a skip list. Skip lists were invented by Bill Pugh, a professor at the University of Maryland, in 1987 (see "Skip lists: a probablistic alternative to balanced trees," *Communications of the ACM*, Vol. 33, No. 6, June 1990), who observed that a hierarchy of linked lists is equivalent to a binary tree. You can think of a skip list as a set of linked lists stacked one on top of the other.

List Arrays

It's easier to get a handle on skip lists by studying a special case called a list array. In a list array, the linked list at the lowest level (level 0) contains all the elements of the list, like a normal linear linked list. The list at the second level contains pointers to every other element. The third level contains pointers to every fourth element, and so on. Each level contains pointers to $^{n}/_{2^{l}}$ elements, where *n* is the number of elements in the list and *l* is the level number starting from 0.

Searching for a key in a list array can be done in $O(\log_2(n))$ time. Starting at the topmost level, traverse the list until you encounter an element greater than or equal to the search key. If the element is equal to the search key, you're done. If not, go down a level from the previous element and repeat the process. If you reach level 0 without finding the key, it's not in the list and you're done. Assuming a list of *n* elements, the search requires no more than $2\log_2(n)$ comparisons because there are $\log_2(n)$ lists and you compare no more than two elements before moving to a lower level. Given the structure of the list array, the search is analogous to a binary search, although it incurs twice as many comparisons.

The list array search algorithm is implemented in the `get()` method in Figure 1. Notice that a dummy list header and tail are used to avoid treating boundary conditions as special cases. The use of the `MinimumKey` and `MaximumKey` classes ensures that the head of the list will always appear to be the lowest-valued element and the tail of the list will appear to be the highest-valued element.

Although searching exhibits a running time competitive with tree-based techniques, insertions and deletions do not perform as well. If you were to insert a key at the beginning of the list, you would need to adjust the level of every other element in the list because a list array restricts pointers at a given level to skip exactly one element in the list at the next lowest level.

Probabilistic Balancing

It turns out that you can achieve similar search performance and much better insertion and deletion performance if you can guarantee that the *average* number of elements skipped is one, instead of the exact number. A skip list is a list array that provides such a guarantee by randomly generating the skip increment for each level when you insert a key. To insert a key, use the search algorithm to locate where to perform the insertion. If the search succeeds and duplicates are disallowed, simply replace the value associated with the key. If the search fails, you should insert the key immediately after the last node visited whose value was less than the value to be inserted.

While you are searching, keep track of the last node visited at each level so that you can update their links after the insertion. Before you insert the key, you have to decide at which levels to insert it (note that every key must be inserted at level 0). You do this by generating a random number between 0 and 1. If the value is less than 0.5, stop. If the value is equal or greater, go up an additional level and insert the key. This generates another random number and repeats the process. The random number generation is implemented in `__randomLevel()` (but uses a probability of 0.25 instead of 0.5) in Figure 1, and the insertion process is implemented in `put()`.

Skip lists can become relatively deep, but the probablistic nature of the balancing makes it unlikely. You can set a cap on the depth to keep small lists shallow. The maximum level should be set so that the expected number of elements at the maximum level is 1. If the probability for level skipping is *p*, then the maximum level is $\log_{1/p}(n)$. The example driver in Figure 2 sets *n* to 2^{20} and *p* to $^{1}/_{4}$, making the maximum level equal to 10. Deletions work in much the same way as insertions, as shown in the `remove()` method from Figure 1.

Performance

Figure 2 contains a driver program that tests the `SkipList` class from Figure 1, comparing its performance to `TreeMap`. The test is unscientific and should really explore a range of values for both `NUM_LEVELS` and `NUM_ELEMENTS`. Nonetheless, it is good enough to get a rough picture of skip list behavior.

If you run the driver, you will find that for large numbers of elements, the `SkipList` class takes roughly twice as long as `TreeMap` to perform insertions, deletions, and searches. Insertions take longer because of the time spent generating random numbers. Both insertions and deletions suffer because they have to maintain the `__update[]` array and index into an array on every call to `getNext()`. Array indexing incurs a higher cost than accessing a left or right child in a tree because of run-time array bounds checking. All of the operations suffer from the extra comparisons required when traversing a skip list. Even though both skip list and tree searches are $O(\log_2(n))$, the constant factors associated with skip list searches in Java are greater.

Straight-up in-order list traversal—the primary reason for using an ordered mapping—is comparable in both situations. You would expect it to be more efficient in a skip list because all you have to do is follow all of the links at the lowest level, whereas a tree requires you to perform potentially costly comparisons. Array indexing again appears to be the culprit. For smaller numbers of elements, `SkipList` is more competitive.

One of the reasons advocated for using skip lists is that they are easier to implement. I find that they are no easier to implement than red-black trees. They also require more storage, which is generally undesirable. Even so, their worst-case performance is independent of the data they store. If you enter sorted data in sequence to a binary tree (not a red-black tree), it degenerates to a linear linked list. With a skip list, your performance is protected by the probabilistic nature of the link construction. Some of the alternatives to skip lists and red-black trees are AVL trees, splay trees, and 2-3 trees. The red-black tree was perhaps the best choice for implementing a general-purpose ordered mapping for the core APIs, but Java makes a capable playground for exploring the alternatives.

Code Listings

Figure 1. SkipList Class

```
package example;

import java.util.Random;
import java.util.Iterator;
import java.util.NoSuchElementException;

// SkipListNode is a container for a key to value mapping in a SkipList.
final class SkipListNode<Key extends Comparable<Key>,Value> {
  Comparable<Key> _key;
  Value _value;
  SkipListNode<Key,Value>[] _next;

  @SuppressWarnings({"rawtypes","unchecked"})
  SkipListNode(Comparable<Key> key, Value value, int level) {
    _next = (SkipListNode<Key,Value>[])new SkipListNode[level+1];
    _key = key;
    _value = value;
  }

  public int getLevel() { return (_next.length - 1); }

  public Comparable<Key> getKey() { return _key; }

  public void setValue(Value value) {
    _value = value;
  }

  public Value getValue() { return _value; }

  public void setNext(int level, SkipListNode<Key,Value> node) {
    _next[level] = node;
  }

  public SkipListNode<Key,Value> getNext(int level) {
    return _next[level];
  }
}

public class SkipList<Key extends Comparable<Key>, Value> {
  private int __level, __numLevels;
  private SkipListNode<Key,Value> __head, __tail;
  private SkipListNode<Key,Value>[] __update;
  private Random __random;

  // A class that represents a key with a value of negative infinity.
  private final class MinimumKey implements Comparable<Key> {
    public int compareTo(Key obj) {

      if(obj == MinimumKey.this) {
        return 0;
      }

      return -1;
    }
  }

  // A class that represents a key with a value of positive infinity.
  private final class MaximumKey implements Comparable<Key> {
    public int compareTo(Key obj) {

      if(obj == MaximumKey.this) {
        return 0;
      }
```

```
    return 1;
  }
}

private final class SkipListIterator implements Iterator<Value> {
  private SkipListNode<Key,Value> __next;

  SkipListIterator() {
    __next = __head.getNext(0);
  }

  public boolean hasNext() {
    return (__next != __tail);
  }

  public Value next() {
    Value result;

    if(__next == __tail) {
      throw new NoSuchElementException();
    }

    result = __next.getValue();
    __next = __next.getNext(0);

    return result;
  }

  public void remove() {
    throw new UnsupportedOperationException();
  }
}

private int __randomLevel() {
  int level = 0;

  // Hardcoded probabilty of 1/4.  Returns at most __level + 1.
  while(__random.nextInt(4) == 0 && level <= __level) {
    ++level;
  }

  if(level >= __numLevels) {
    level = __numLevels - 1;
  }

  return level;
}

/**
 * Creates a SkipList with a given maximum number of levels.  This
 * number should be lg(n) - 1, where n is the number of nodes you
 * expect the list to contain.
 */
@SuppressWarnings({"rawtypes","unchecked"})
public SkipList(int numLevels) {
  __level  = 0;
  __numLevels = numLevels;
  __random = new Random(System.currentTimeMillis());
  __head = new SkipListNode<Key,Value>(new MinimumKey(), null, numLevels - 1);
  __tail = new SkipListNode<Key,Value>(new MaximumKey(), null, numLevels - 1);
  __update = (SkipListNode<Key,Value>[])new SkipListNode[numLevels];

  for(int i = 0; i < numLevels; ++i) {
    __head.setNext(i, __tail);
```

```
    __tail.setNext(i, null);
    __update[i] = __head;
  }
}


public Iterator<Value> iterator() {
  return new SkipListIterator();
}

/**
 * Returns the value associated with the key, null if the list
 * doesn't contain the key.
 */
public Value get(Key key) {
  int level, comparison;
  SkipListNode<Key,Value> node, nextNode;
  Comparable<Key> nextKey;

  node = __head;

  for(level = __level; level >= 0; --level) {
    do {
      nextNode   = node.getNext(level);
      nextKey    = nextNode.getKey();
      comparison = nextKey.compareTo(key);

      if(comparison >= 0) {
        break;
      }

      node = nextNode;
    } while(true);

    if(comparison == 0) {
      return node.getValue();
    }
  }

  return null;
}

/**
 * Adds a key/value pair to the list.  If the key is already in the
 * list, the existing value is replaced with the new value and the
 * old value is returned.  Returns null if the key wasn't already
 * present (this is a problem if you store null values in the list).
 */
public Value put(Key key, Value value) {
  int level, newLevel, comparison;
  SkipListNode<Key,Value> node, nextNode;
  Comparable<Key> nextKey;
  Value result = null;

  node = __head;

  for(level = __level; level >= 0; --level) {
    do {
      nextNode   = node.getNext(level);
      nextKey    = nextNode.getKey();
      comparison = nextKey.compareTo(key);

      if(comparison >= 0) {
        break;
      }
```

```
      node = nextNode;
    } while(true);

    if(comparison == 0) {
      result = nextNode.getValue();
      nextNode.setValue(value);

      return result;
    }

    __update[level] = node;
  }

  // The key isn't in the list.
  newLevel = __randomLevel();
  if(newLevel > __level) {
    // newLevel is always __level + 1 at this point so
    // we don't have to update levels in between.
    __update[newLevel] = __head;
    __level = newLevel;
  }

  node = new SkipListNode<Key,Value>(key, value, newLevel);
  for(level = 0; level <= newLevel; ++level) {
    node.setNext(level, __update[level].getNext(level));
    __update[level].setNext(level, node);
  }

  return result;
}

/**
 * Removes the key an associated value from the list and returns the
 * value that was removed.  Returns null if the key wasn't found.
 */
public Value remove(Key key) {
  int level, comparison = -1;
  SkipListNode<Key,Value> nextNode = null, node;
  Comparable<Key> nextKey;
  Value result = null;

  node = __head;

  for(level = __level; level >= 0; --level) {
    do {
      nextNode   = node.getNext(level);
      nextKey    = nextNode.getKey();
      comparison = nextKey.compareTo(key);

      if(comparison >= 0) {
        break;
      }

      node = nextNode;
    } while(true);

    __update[level] = node;
  }

  if(comparison == 0) {
    int maxLevel = nextNode.getLevel();
    result = nextNode.getValue();

    for(level = 0; level <= maxLevel; ++level) {
```

```
        __update[level].setNext(level, nextNode.getNext(level));
      }

      // Reset uppermost level if there are no more items
      if(maxLevel == __level && __update[maxLevel] == __head &&
         __head.getNext(maxLevel) == __tail)
      {
        --__level;
      }
    }

    return result;
  }
}
```

Figure 2. SkipSearch Class

```
package example;

import java.util.Iterator;
import java.util.Random;
import java.util.TreeMap;
import java.util.concurrent.ConcurrentSkipListMap;

public class SkipSearch {

  // An interface to make generic timings possible given that SkipList
  // does not implement java.util.Map.
  interface SimpleMap<Key extends Comparable<Key>, Value> {
    public Value put(Key key, Value value);
    public Value get(Key key);
    public Value remove(Key key);
    public Iterator<Value> iterator();
  }

  public static class MySkipList<Key extends Comparable<Key>, Value>
    extends SkipList<Key,Value> implements SimpleMap<Key, Value>
  {
    public MySkipList(int numLevels) {
      super(numLevels);
    }
    public Value put(Key key, Value value) {
      return super.put(key, value);
    }
    public Value get(Key key) {
      return super.get(key);
    }
    public Value remove(Key key) {
      return super.remove(key);
    }
    public Iterator<Value> iterator() {
      return super.iterator();
    }
  }

  @SuppressWarnings("serial")
  public static class MyTreeMap<Key extends Comparable<Key>, Value>
    extends TreeMap<Comparable<Key>, Value> implements SimpleMap<Key, Value>
  {
    public Value put(Key key, Value value) {
      return super.put(key, value);
    }
    public Value get(Key key) {
      return super.get(key);
    }
    public Value remove(Key key) {
      return super.remove(key);
    }
    public Iterator<Value> iterator() {
      return super.values().iterator();
    }
  }

  @SuppressWarnings("serial")
  public static class MySkipListMap<Key extends Comparable<Key>, Value>
    extends ConcurrentSkipListMap<Comparable<Key>, Value> implements SimpleMap<Key, Value>
  {
    public Value put(Key key, Value value) {
      return super.put(key, value);
    }
```

```
  public Value get(Key key) {
    return super.get(key);
  }
  public Value remove(Key key) {
    return super.remove(key);
  }
  public Iterator<Value> iterator() {
    return super.values().iterator();
  }
}

public static final <K extends Comparable<K>>
long timePuts(SimpleMap<K,K> map, K[] keys)
{
  long start, finish;

  start = System.currentTimeMillis();

  for(int i = 0; i < keys.length; ++i) {
    map.put(keys[i], keys[i]);
  }

  finish = System.currentTimeMillis();

  return (finish - start);
}

public static final <K extends Comparable<K>,V>
long timeGets(SimpleMap<K,V> map, K[] keys, Random random)
{
  long start, finish;
  // Only get a fraction of the keys.
  int max = keys.length >> 4;

  if(max <= 0) {
    max = keys.length;
  }

  start = System.currentTimeMillis();

  for(int i = 0; i < max; ++i) {
    map.get(keys[random.nextInt(keys.length)]);
  }

  finish = System.currentTimeMillis();

  return (finish - start);
}

public static final <K extends Comparable<K>, V>
long timeRemoves(SimpleMap<K,V> map, K[] keys, Random random)
{
  long start, finish;
  // Only remove a fraction of the keys.
  int max = keys.length >> 4;

  if(max <= 0) {
    max = keys.length;
  }

  start = System.currentTimeMillis();

  for(int i = 0; i < max; ++i) {
    map.remove(keys[random.nextInt(keys.length)]);
  }
```

```
    finish = System.currentTimeMillis();

    return (finish - start);
  }

  public static final <K extends Comparable<K>,V>
  long timeIteration(SimpleMap<K,V> map)
  {
    long start, finish;
    Iterator<V> iterator = map.iterator();

    start = System.currentTimeMillis();

    while(iterator.hasNext()) {
      iterator.next();
    }

    finish = System.currentTimeMillis();

    return (finish - start);
  }

  public static final Integer[] makeRandomKeys(int iterations) {
    Integer[] keys = new Integer[iterations];
    Random random = new Random(System.currentTimeMillis());

    for(int i = 0; i < iterations; ++i) {
      keys[i] = new Integer(random.nextInt(iterations));
    }

    return keys;
  }

  public static final <Key extends Comparable<Key>>
  void doTimings(String name, SimpleMap<Key,Key> map, Key[] keys, Random random)
  {
    long time;

    System.out.println();
    System.out.println(name);
    System.out.println("\n  puts (ms)");
    time = timePuts(map, keys);
    System.out.println("  total  : " + time);
    System.out.println("  average: " +
                       (float)time/(float)keys.length);
    System.out.println("\n  gets (ms)");
    time = timeGets(map, keys, random);
    System.out.println("  total  : " + time);
    System.out.println("  average: " +
                       (float)time/(float)keys.length);
    System.out.println("\n  iteration (ms)");
    time = timeIteration(map);
    System.out.println("  total  : " + time);
    System.out.println("  average: " +
                       (float)time/(float)keys.length);
    System.out.println("\n  removes (ms)");
    time = timeRemoves(map, keys, random);
    System.out.println("  total  : " + time);
    System.out.println("  average: " +
                       (float)time/(float)keys.length);
  }

  public static final int NUM_ELEMENTS = 1024 * 1024;
  // Expected number of elements is 4^10
```

```
  public static final int NUM_LEVELS = 10;

  public static final void main(String[] args) {
    MySkipList<Integer,Integer> skipList =
      new MySkipList<Integer,Integer>(NUM_LEVELS);
    MyTreeMap<Integer,Integer> treeMap = new MyTreeMap<Integer,Integer>();
    MySkipListMap<Integer,Integer> skipListMap =
      new MySkipListMap<Integer,Integer>();
    Integer[] keys = makeRandomKeys(NUM_ELEMENTS);
    long time = System.currentTimeMillis();
    // Create identical pseudo-random number sequences
    Random skipRand = new Random(time);
    Random treeRand = new Random(time);
    Random slmapRand = new Random(time);

    doTimings("SkipList", skipList, keys, skipRand);
    doTimings("TreeMap", treeMap, keys, treeRand);
    doTimings("ConcurrentSkipListMap", skipListMap, keys, slmapRand);
  }
}
```