developerWorks®

Ropes: Theory and practice

Why and when to use Ropes for Java for string manipulations

Amin Ahmad

February 12, 2008

Systems that manipulate large quantities of string data are poorly served by the Java™ language's default `String` and `StringBuilder` classes. A *rope* data structure can be a better alternative. This article introduces Ropes for Java, a rope implementation for the Java platform; explores performance issues; and provides pointers for effective use of the library.

A *rope data structure* represents an immutable sequence of characters, much like a Java `String`. But ropes' highly efficient mutations make ropes — unlike `String`s and their mutable `StringBuffer` and `StringBuilder` cousins — ideal for applications that do heavy string manipulation, especially in multithreaded environments.

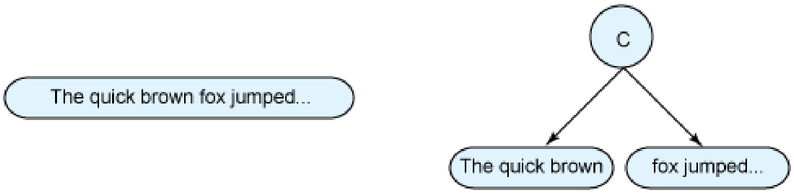
After briefly summarizing the rope data structure, this article introduces Ropes for Java, a rope implementation for the Java platform. Then it benchmarks `String`, `StringBuffer`, and the Ropes for Java `Rope` class on the Java platform; investigates some special performance issues; and concludes with a discussion about when (and when not) to use ropes in your applications.

Ropes: A brief overview

A *rope* represents an immutable character sequence. A rope's *length* is simply the number of characters in the sequence. Most string representations store all their characters contiguously in memory. The defining characteristic of a rope is that it does away with this restriction, instead allowing fragments of the rope to reside noncontiguously and joining them using *concatenation nodes*. This design allows concatenation to run asymptotically faster than for Java `String`s. The `String` version of concatenation requires strings you want to join to be copied to a new location, which is an O(*n*) operation. The rope counterpart simply creates a new concatenation node, which is an O(1) operation. (If you're unfamiliar with big-O notation, see Related topics for a link to explanatory material.)

Figure 1 illustrates two types of string representations. In the one on the left, the characters are located in contiguous memory locations. Java strings rely on this representation. In the representation on the right, a disjointed string is combined using a concatenation node.

Figure 1. Two representations of a string



Rope implementations also often defer evaluation of large substring operations by introducing a *substring node*. Use of a substring node reduces the time for extracting a substring of length *n* from O(*n*) to O(log *n*), and often to O(1). It is important to note that Java `String`s, being immutable, also have a constant-time substring operation, but `StringBuilder`s often do not.

A *flat rope* — a rope with no concatenation or substring nodes — has a *depth* of 1. The depth of concatenation and substring ropes is one more than the depth of the deepest node they enclose.

Ropes have two overheads that contiguous-character string representations do not. The first is that a superstructure of substring and concatenation nodes must be traversed to reach a specified character. Furthermore, this tree superstructure must be kept as balanced as possible to minimize traversal times, implying that ropes need occasional rebalancing to keep read performance good. Even when ropes are well balanced, obtaining the character at a specified position is an O(log *n*) operation, where *n* is the number of concatenation and substring nodes the rope comprises. (For convenience, the rope's length can be substituted for *n*, because the length is always greater than the number of substring and concatenation nodes in the rope.)

Luckily, rope rebalancing is fast, and the determination of when to rebalance can be made automatically, for example by comparing the rope's length and depth. And, in most data-processing routines, sequential access to a rope's characters is what's required, in which case a rope iterator can provide amortized O(1) access speed.

Table 1 compares the expected run times of some common string operations on both ropes and Java `String`s.

Table 1. Expected run times of common string operations

Operation	Rope	Java String
Concatenation	O(1)	O(*n*)
Substring	O(1)	O(1)
Retrieve character	O(log *n*)	O(1)
Retrieve all characters sequentially (cost per character)	O(1)	O(1)

Introducing Ropes for Java

Memory issues

Constant-time substring implementations in Java code can cause memory problems because the substring reference prevents the original string from being garbage collected. Both `Rope` and `String` suffer from this problem.

Ropes for Java is a high-quality implementation of the rope data structure for the Java platform (see Related topics). It implements a wide range of optimizations to help provide excellent all-around performance and memory usage. This section explains how to integrate ropes into a Java application and compares rope performance with that of `String` and `StringBuffer`.

The Ropes for Java implementation exposes a single class to clients: `Rope`. `Rope` instances are created from arbitrary `CharSequence`s using the `Rope.BUILDER` factory builder.

Listing 1 shows how to create a rope.

Listing 1. Creating a rope

```
Rope r = Rope.BUILDER.build("Hello World");
```

`Rope` exposes a standard battery of methods for manipulation, including methods to:

- Append another character sequence
- Delete a subsequence
- Insert another character sequence into a rope

Keep in mind that, as with `String`, each of these mutations returns a new rope; the original is left unmodified. Listing 2 illustrates some of these operations.

Listing 2. Rope mutations

```
Rope r = Rope.BUILDER.build("Hello World");
r = r.append("!");  // r is now "Hello World!"
r = r.delete(0,6);  // r is now "World!"
```

Efficient rope Iteration

Iterating over a rope requires some care. After examining the two code blocks in Listing 3, see if you can determine which one performs better.

Listing 3. Two techniques for iterating over a rope

```
//Technique 1
final Rope r=some initialization code;
for (int j=0; j<r.length(); ++j)
    result+=r.charAt(j);

//Technique 2
final Rope r=some initialization code;
for (final char c: r)
    result+=c;
```

Recall that returning a single character at an arbitrary position within a rope is an O(log n) operation. However, by using `charAt` for each character, the first code block in Listing 3 pays the O(log n) lookup time n times. The second block, which uses an `Iterator` instead, should perform faster than the first. Table 2 summarizes the performance of iterating over a rope of length 10,690,488 using both approaches. For comparison, Table 2 includes times for `String` and `StringBuffer`.

Table 2. Complex rope iteration performance

Technique	Time (ns)
`String`	69,809,420
`StringBuffer`	251,652,393
`Rope.charAt`	79,441,772,895
`Rope.iterator`	1,910,836,551

The rope used to obtain the results in Table 2, which are in line with expectations, was created by performing a complex series of mutations to an initial string. However, if the rope is created directly from a character sequence, without any subsequent mutations, the performance numbers take a surprising twist. Table 3 compares the performance of both approaches, this time by iterating over all 182,029 characters of a rope initialized directly from a character array containing the Project Gutenberg edition of Charles Dickens' *A Christmas Carol*.

Table 3. Rope iteration performance

Technique	Time (ns)
`String`	602,162
`StringBuffer`	2,259,917
`Rope.charAt`	462,904
`Rope.iterator`	3,466,047

How can this performance reversal be explained in light of my earlier theoretical discussion? The rope's construction is a key factor: When a rope is constructed directly from an underlying `CharacterSequence` or character array, it has a simple structure consisting of a single flat rope. Because the rope contains no concatenation or substring nodes, character lookup consists of direct delegation to the underlying sequence's `charAt` method (in the `CharacterSequence` case) or a direct array lookup (in the array case). The performance of `Rope.charAt` for a flat rope matches that of the underlying representation, which is most often O(1); hence the performance difference.

Astute readers might wonder why `charAt` is more than seven times faster than the iterator, given that both provide O(1) access time. This difference is due to the fact that in the Java language, `Iterator`s must return `Object`s. Although the `charAt` implementation directly returns `char` primitives, the iterator implementation must box each `char` into a `Character` object. Auto-boxing might sooth the syntactic pain of primitive boxing, but it can't eliminate the performance penalties.

Finally, it is remarkable that the performance of `Rope.charAt` is better than the performance of `String.charAt`. The reason is that `Rope` represents lazy substrings using a dedicated class, allowing the implementation of `charAt` to remain simple for regular ropes. In contrast, the Java SE implementation of `String` uses the same class to represent regular strings and lazy substrings, which somewhat complicates the logic of `charAt` and thereby adds a small performance penalty during iteration over regular strings.

Rope iteration will resurface when I discuss optimizing the performance of regular-expression searches on ropes.

Optimizing output with Rope.write

At some point, most rope instances must be written somewhere, usually to a stream. Unfortunately, writing an arbitrary object to a stream invokes the object's `toString` method. This approach to serialization forces a string representation of the entire object to be created in memory before a single character can be written, which is a big performance drag for large objects. Ropes for Java was designed with large string objects in mind, so it provides a better approach.

To improve performance, `Rope` introduces a write method that accepts a `Writer` and a range specification and writes the rope's specified range into the writer. This saves the time and memory cost of constructing a `String` from the rope, which, for large ropes, is enormous. Listing 4 shows both the standard and enhanced approaches to rope output.

Listing 4. Two approaches to `Rope` output

```
out.write(rope);
rope.write(out);
```

Table 4 contains the benchmark results of writing a rope with length 10,690,488 and depth 65 to a stream backed by an in-memory buffer. Note that only time savings are shown, but the savings to temporarily allocated heap space are much larger.

Table 4. Rope output performance

Technique	Time (ns)
`out.write`	75,136,884
`rope.write`	13,591,023

Benchmarking mutator performance

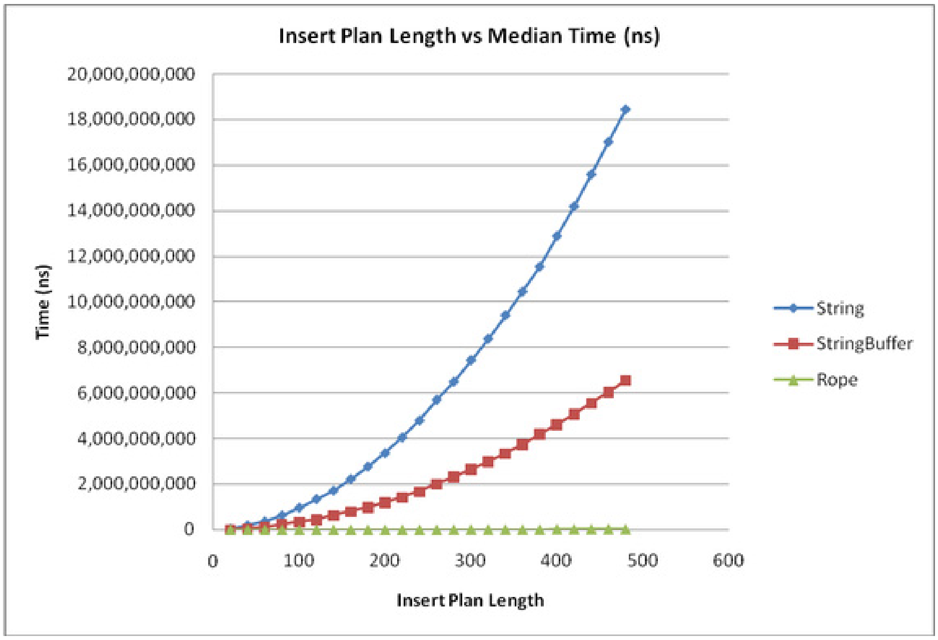
Rope mutations are, in theory, much faster than those of contiguous-character string representations. On the flip side, as you have seen, rope iteration can be slower as a result. In this section, you'll look at some benchmarks comparing the mutation performance of Ropes for Java with `String`s and `StringBuffer`s.

All the tests are initialized with the Project Gutenberg EBook of *A Christmas Carol* (182,029 bytes) and apply a successive series of mutations to it. In most cases, I varied the number of mutations from 20 to 480 to provide a picture of how well the data structures scale. (Figures 2, 3, and 4 refer to this as the *plan length*.) I performed each test seven times and used the median result.

Insert benchmark

In the insert benchmark, a substring was selected at random from the previous iteration's output and inserted at a random location in the string. Figure 2 shows the benchmark results.

Figure 2. Insert benchmark results

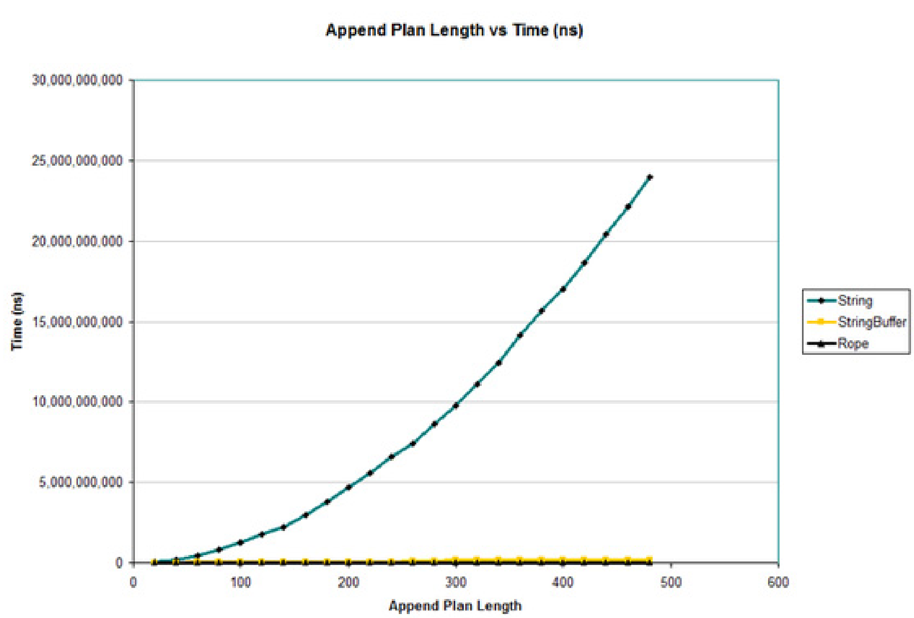


For both `String`s and `StringBuffer`s, the amount of time required to complete the benchmark rises exponentially as the plan length increases. Ropes, in contrast, perform extremely well.

Append benchmark

The append benchmark consists of appending a random range of the input string to itself. This test is interesting because `StringBuffer`s are optimized for performing fast appends. Figure 3 shows the benchmark results.

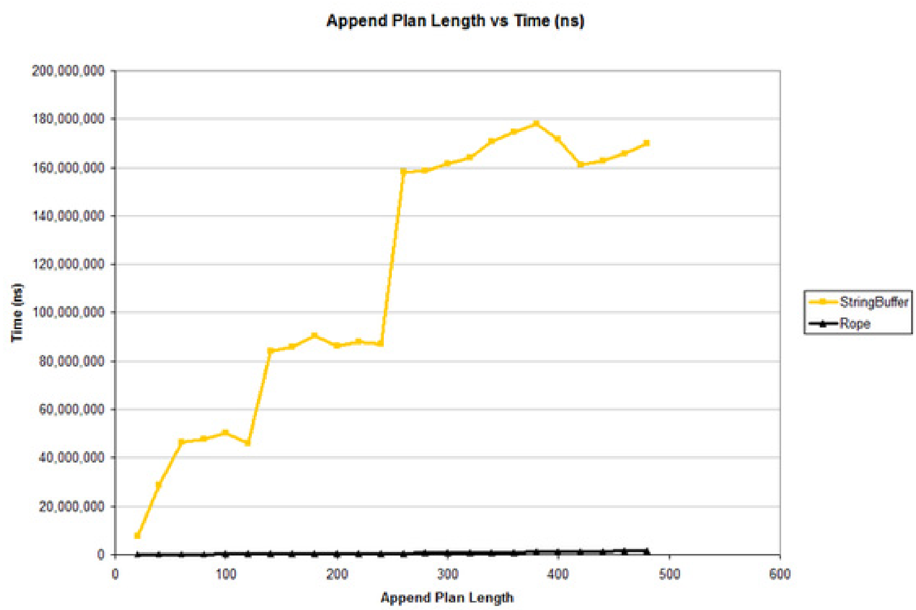
Figure 3. Append benchmark results



Unfortunately, the view in Figure 3 is obscured by the atrocious performance of `String`. However, as expected, `StringBuffer` performance is excellent.

Figure 4 takes `String` out of the equation and rescales the chart to show more clearly how `Rope` and `StringBuffer` performance compare.

Figure 4. Append benchmark results, excluding `String`



Rope immutability: A caveat

The `Rope` class is designed to be immutable, meaning that mutator functions never modify the original `Rope` instance, but, rather, return a modified copy. Immutability confers a number of benefits, the chief one being that `Rope` instances can be shared without precaution in a multithreaded environment, thereby greatly simplifying program logic.

Ropes for Java constructs ropes from an underlying character sequence. Because `CharSequence` is an interface, you gain a great deal of flexibility: a rope can be constructed from heterogeneous sources including files on disk, in-memory buffers, and documents stored on remote servers. However, unlike `String`, which is guaranteed to be immutable, `CharSequence` places no such restriction on its implementers. It is the application programmer's responsibility to ensure that the underlying character sequence used to build a rope is effectively immutable for the rope instance's lifespan.

Figure 4 shows a dramatic difference between `Rope` and `StringBuffer` that was not apparent in Figure 3. `Rope` barely manages to rise above the x-axis. However, what is really interesting is the graph of `StringBuffer` —it jumps and plateaus instead of rising smoothly. (As an exercise, try to explain why before reading the next paragraph.)

The reason has to do with how `StringBuffer`s allocate space. Recall that they allocate extra space at the end of their internal array to allow for efficient appending. But after that space is exhausted, a brand new array must be allocated and all the data copied to it. The new array is generally sized as some factor of the current length. As the plan length increases, so does the length of the resulting `StringBuffer`. And as resize thresholds are reached, the time taken jumps as an extra resize and copy occurs. Then performance plateaus (that is, time taken rises slowly) for the next several plan length increases. Because each plan item increases the total string length by roughly the same amount, the ratio of subsequent plateaus provides an indicator of the resize factor for the underlying array. Based on the results, an accurate estimate for this particular `StringBuffer` implementation is somewhere around 1.5.

Results summary

So far, I've relied on graphs to illustrate the performance differences among `Rope`, `String`, and `StringBuffer`. Table 5 provides the timing results for all mutation benchmarks, using 480-item plan lengths.

Table 5. Median mutation times with a 480-item plan, plus delete results

Mutation/data structure	Time (ns)
Insert	
`String`	18,447,562,195
`StringBuffer`	6,540,357,890
`Rope`	31,571,989
Prepend	
`String`	22,730,410,698
`StringBuffer`	6,251,045,63
`Rope`	57,748,641
Append	
`String`	23,984,100,264
`StringBuffer`	169,927,944
`Rope`	1,532,799
Delete (delete 230 random ranges from initial text)	
`String`	162,563,010
`StringBuffer`	10,422,938
`Rope`	865,154

See Related topics for a link to the benchmark program. I encourage you to run it on your platform and verify the results I'm presenting here.

Optimizing regular-expression performance

Regular expressions, introduced into the JDK in version 1.4, are a widely used feature in many applications. Therefore, it is critical that they perform well on ropes. In the Java language, regular expressions are represented as `Pattern`s. To match a `Pattern` against regions of an arbitrary `CharSequence`, you use the pattern instance to construct a `Matcher` object, passing the `CharSequence` as a parameter.

The ability to work with `CharSequence`s gives the Java regular-expression library excellent flexibility, but it also introduces one serious drawback. A `CharSequence` provides only a single method for accessing its characters: `charAt(x)`. As I mentioned in the overview section, random character access on a rope with many internal nodes is approximately O(log *n*), so traversal is O(*n* log *n*). To illustrate the problem that this causes, I benchmarked the time it takes to find all instances of the pattern "`Crachit*`" in a string with length 10,690,488. The rope used was constructed through the same series of mutations as the insert benchmark and has a depth of 65. Table 6 shows the results.

Table 6. Regular-expression search times

Technique	Time (ns)
`String`	75,286,078
`StringBuffer`	86,083,830
`Rope`	12,507,367,218
Rebalanced `Rope`	2,628,889,679

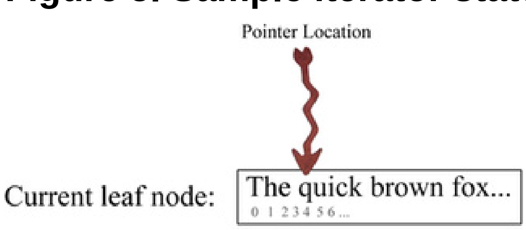
Clearly, the matching performance of a `Rope` is poor. (To be clear, this is true for a `Rope` with many internal nodes. For a flat rope, the performance is nearly identical to that of the underlying character representation.) Even explicitly rebalancing the `Rope`, which makes matching 3.5 times faster, doesn't put `Rope`s in the same league as either `String`s or `StringBuffer`s.

To improve this situation, `Matcher`s should and can harness the much faster O(1) access times that `Rope` iterators provide. To grasp how this works you first need to understand the access pattern of a `Matcher` to its `CharSequence`.

The most common scenario for matching regular expressions is to start at some point in a character sequence and move forward until all matches have been found and the end of the sequence is reached. In this scenario, the matcher mainly moves in a forward direction, often by more than one character at a time. Occasionally, however, the matcher is forced to backtrack.

You can easily modify the `Rope` iterator to accommodate skipping forward by more than one character at a time. But moving backward is more complicated, because internally the iterator performs a depth-first, preorder traversal of the `Rope`, visiting each of its leaf nodes. The traversal stack doesn't have enough information to move to the previous leaf, but if the amount to move back does not take the iterator off of the current leaf it is possible to service the request. To illustrate, Figure 5 shows the state of a hypothetical iterator that can move back one, two, three, or four places, but no more, because that would require accessing the previously visited leaf.

Figure 5. Sample iterator state



To enable this new capability, the `Rope`'s `charAt` method can be modified so that, on the first invocation, an iterator is constructed at the specified position. Subsequent invocations move the iterator backward or forward by the required distance. If the iterator can't move backward the required distance, then the default `charAt` routine is executed to obtain the character value — a scenario that one hopes will occur rarely.

Because this optimization is not generally applicable and because it requires the introduction of new member variables, it is best not to add it directly to the `Rope` class. Instead, you can decorate the rope with this optimization on demand. To let you accomplish this, Ropes for Java includes a method on the `Rope` class that produces an optimized matcher for a specified pattern. Listing 5 illustrates this approach:

Listing 5. Optimized regular-expression matching

```
Pattern p = ...
Matcher m = rope.matcher(p);
```

The call on the second line in Listing 5 decorates the rope to optimize regular-expression matching.

Table 7 provides benchmark results for this approach, with the previous results (from Table 6) included for clarity.

Table 7. Regular-expression search times with `rope.matcher()`

Technique	Time (ns)
`String`	75,286,078
`StringBuffer`	86,083,830
`Rope`	12,507,367,218
Rebalanced `Rope`	2,628,889,679
`Rope.matcher`	246,633,828

The optimization results in a significant 10.6x improvement over the rebalanced rope and brings the rope to within a factor of 3.2 times the `String`'s performance.

Applications

When not to use ropes

Often, enterprise Java applications contain code similar to the following:

```
String x =
"<input type='text'
name='name' value='"
+ escapePcData(bean.getName())
+ "'>";
```

x is subsequently sent as part of an HTML page to the client's browser. Does it make sense to use ropes to compute the value of *x*, rather than a `StringBuilder`, which is what the compiler generates by default?

The answer is no, for a variety of reasons. To begin with, the amount of data being concatenated is, presumably, small, so using a rope is unlikely to improve performance, although it could improve robustness and scalability. (Consider how both solutions would behave if `getName` unexpectedly returned a 50 MB string.)

But for the sake of argument, imagine that many more chunks of data are being concatenated. Given that `Rope`'s append performance is generally better than `StringBuffer`'s, would a rope make sense then? Again, no. Whenever input data is being combined to produce formatted output, the cleanest and most efficient approach is to use a template engine such as StringTemplate or FreeMarker. Not only does this approach cleanly separate presentation markup from code, but templates are compiled once (often to JVM bytecode) and then can be reused, giving them excellent performance characteristics.

A second benefit of using templates exposes a fundamental flaw common to output-building routines like those in the code above, including one written using ropes. This benefit is that templates can be incrementally evaluated, and output can be written to a `Writer` as soon as it is produced, rather than unnecessarily first being accumulated in memory. In a Java EE application, where the `Writer` is actually a buffered connection to the client's browser, this approach to output rendering uses constant memory, O(1), versus the other solutions' O(*n*) memory usage. This is a *huge* improvement to application scalability and robustness, even though it's not apparent for smaller inputs or at lower application loads. (See Related topics for links to two articles on streaming architecture that further explain and quantify this approach.)

Now that you have a good understanding of rope performance, it's time to consider some traditional uses for ropes as well as a tempting, but likely, inappropriate use in Java EE applications.

Although ropes can serve as a general-purpose replacement for contiguous-memory representations of strings, only applications that extensively modify large strings will see a significant performance improvement. Perhaps it is not surprising, then, that one of the earliest applications of ropes was to represent documents in a text editor. Not only can text insertions and deletions be performed in near-constant time for extremely large documents, but ropes' immutability makes implementation of an undo stack trivial: simply store a reference to the previous rope with every change.

Another, more esoteric, application of ropes is to represent state in a virtual machine. For example, the ICFP 2007 Programming Contest involved implementing a virtual machine that modified its state with every cycle and ran for millions of cycles for some inputs (see Related topics). In one Java implementation, the virtual machine's speed was improved by three orders of magnitude, from ~50 cycles/sec to more than 50,000/sec, by changing the state representation to use a `Rope` instead of a specialized `StringBuffer`.

Directions for future investigation

Although Ropes for Java is a new library, the underlying concepts are old, and the library appears to deliver on the performance promise of ropes. However, the project plans to improve some aspects of the library in future releases by:

- Providing high-performance implementations of other common string operations.
- Writing adapters to integrate ropes seamlessly into Scala (see Related topics) and other advanced languages for the Java platform.

- Enhancing quality through further automated testing. Ropes for Java is tested using both manually written automated JUnit tests, as well as automatically generated automated tests using JUnit Factory. Incorporating Java Modeling Language (JML) annotations checked by ESC/Java2 (see Related topics) might further improve quality.

Related topics

- Ropes for Java: Visit the project Web site.
- Big-O notation: Computational complexity theory uses big-O notation to describe how input-data size affects an algorithm's use of computational resources.
- The 10th ICFP Programming Contest: The author's entry in this contest demonstrated Ropes for Java's productivity.
- "Streaming Architecture" and "Streaming Presidents" (Amin Ahmad, Ahmadsoft.org): Detailed explanation and quantification of streaming architectures.
- The Scala programming language: Functional programming for the JVM.
- *The busy Java developer's guide to Scala* (Ted Neward, developerWorks): Learn Scala from the ground up.
- JUnit: and JUnitFactory: Ropes for Java is tested with JUnit and JUnitFactory.
- JML and ESC/Java2: ESC/Java2 is a programming tool that tries to find common run-time errors in JML-annotated Java programs.
- Ropes for Java: Download the latest release.
- PerformanceTest.java: The performance benchmark code used to test mutator performance for this article. You can download and run this code to obtain personalized results for your platform.
- Download IBM product evaluation versions and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.