Alternative Implementations of the SortedTable Interface

Allan Heydon

DRAFT — January 26, 1998 — DRAFT

1 Introduction

The standard Modula-3 library includes two generic interfaces named `Table` and `SortedTable` [3, Sections 3.5 and 3.6]. A `Table.T` is a table of key-value pairs. A `SortedTable.T` is a subtype of `Table.T` that also provides methods for iterating over the elements of the table in increasing or decreasing key order and for seeking to an arbitrary key in the iteration sequence.

The default sorted table implementation, `SortedTable.Default`, is implemented using a randomized data structure called a heap-ordered binary tree, or "treap" [1]. This paper describes the implementations of two other well-known data structures, one based on red-black trees, and one based on skip lists. These implementations are provided by the `sortedtableextras` package, which also includes templates for instantiating the generics as described below. The paper concludes by comparing the performance and memory usage of the four implementations.

2 RedBlackTbl

A `RedBlackTbl.T` is a subtype of a `SortedTable.T`, but it is implemented using red-black trees. Red-black trees are self-balancing binary search trees.

```
GENERIC INTERFACE RedBlackTbl(Key, Value, SortedTbl);
```

Where the same requirments exist on the `Key` and `Value` interfaces as those described in the generic `SortedTable` interface and where `SortedTbl` is the generic instance `SortedTable(Key, Value)`.

```
CONST Brand = "(RedBlackTbl " & Key.Brand & " " & Value.Brand & ")";
```

The type `T` is revealed to have brand `Brand`.

```
TYPE
  T <: Public;
  Public = SortedTbl.T OBJECT METHODS
    init(): T;
    keyCompare(READONLY k1, k2: Key.T): [-1..1];
  END;
```

```
  Iterator <: IteratorPublic;
  IteratorPublic = SortedTbl.Iterator OBJECT METHODS
    reset();
  END;

END RedBlackTbl.
```

2.1 Method Specifications

The expression `NEW(T).init()` evaluates to a new table with no elements. The `init` method may also be invoked on an existing table to delete all of its entries.

The implementation calls the `keyCompare` method to compare two keys. The default `keyCompare` method simply returns `Key.Compare(k1, k2)`. However, subtypes may wish to override the `keyCompare` method to effect a new key ordering. `keyCompare` is required to implement a total order.

The `iterate` method returns an iterator of type `Iterator`, a subtype of `SortedTbl.Iterator`. Its `reset` method resets the iterator. This allows clients to iterate over a table multiple times without having to allocate a new `Iterator` object on each pass.

2.2 Synchronization

For efficiency, red-black tables and their iterators are not monitored, so a client accessing a table from multiple threads must ensure that if two operations are active concurrently, then neither of them has side-effects on the same table or iterator. The `init`, `put`, and `delete` methods are the only ones with side-effects on the table. All three of an iterator's `reset`, `next`, and `seek` methods have side-effects on the iterator.

2.3 Quake Instantiation Procedures

The `sortedtableextras` package includes a quake template that defines quake procedures for instantiating instances of the `RedBlackTbl` generic interface and implemenation. The two procedures are:

```
redblack_table (nm, key, value)
RedBlack_table (nm, key, value)
```

The only difference between these two procedures is that tables instantiated by the former are private to the package in which they are built, while those instantiated by the latter are exported.

These procedures create and include the two generic instantiation files `RedBlack<nm>Tbl.i3` and `RedBlack<nm>Tbl.m3`. The generic interface and implementation are instantiated with the interfaces named `key` and `value`. `nm` should be a string representing the concatenation of the names `key` and `value`, possibly in abbreviated form; it must be the same name that is used to instantiate the generic `Table` and `SortedTable` interfaces. Here are some examples: uses

```
redblack_table ("IntInt", "Integer", "Integer")
redblack_table ("IntText", "Integer", "Text")
redblack_table ("RealRef", "RealType", "Refany")
```

For example, the last procedure call would create the two derived files `RedBlackRealRefTbl.i3` and `RedBlackRealRefTbl.m3`.

In order for a program that includes a `RedBlackTbl` instantiation to link successfully, it must also instantiate the generic `Table` and `SortedTable` interfaces with the same `nm`, `key`, and `value` arguments.

2.4 Performance and Implementation

A red-black table's `get`, `put`, and `delete` methods take `O(log n)` time in the worst case, where `n` is the number of elements in the table. The other table methods take constant time. An iterator's `reset`, `next`, and `seek` methods also take `O(log n)` time in the worst case. As opposed to seeking on a `SortedTbl.Default`, seeking in a red-black table has the same cost whether seeking forward or backward.

This implementation is based on the description of red-black trees in a well-known algorithms text [2, Chapter 14]. In this implementation, the tree is only rebalanced on insertions and deletions, not on searches or iterations.

The space requirements of a red-black table are dominated by the space costs for each of its entries. The space required for each entry is the space required for the key and the value plus the space for three `REF`s and the space for the color bit. Stricly speaking, the color bit should require only 1 bit. However, due to alignment restrictions, it probably requires `Word.Size` bits in practice.

3 SkipListTbl

A `SkipListTbl.T` is a subtype of a `SortedTable.T`, but it is implemented using skip lists. Skip lists are randomized data structures that have logarithmic expected-time performance.

```
GENERIC INTERFACE SkipListTbl(Key, Value, SortedTbl);
```

Where the same requirments exist on the `Key` and `Value` interfaces as those described in the generic `SortedTable` interface and where `SortedTbl` is the generic instance `SortedTable(Key, Value)`.

```
CONST Brand = "(SkipListTbl " & Key.Brand & " " & Value.Brand & ")";
```

The type `T` is revealed to have brand `Brand`.

```
TYPE
  T <: Public;
  Public = SortedTbl.T OBJECT METHODS
    init(maxSizeHint: CARDINAL := 10000; fixedSeed := FALSE): T;
    keyCompare(READONLY k1, k2: Key.T): [-1..1];
  END;

  Iterator <: IteratorPublic;
  IteratorPublic = SortedTbl.Iterator OBJECT METHODS
    reset();
  END;
```

```
END SkipListTbl.
```

3.1 Method Specifications

The expression `NEW(T).init(maxSizeHint, fixedSeed)` evaluates to a new table with no elements. The `init` method may also be invoked on an existing table to delete all of its entries.

The `maxSizeHint` parameter should be an estimate of the table's maximum size. If the estimate is too small, the table will perform poorly, so it is better to over-estimate. The cost of over-estimating is that the table will consume more space than necessary.

Each `SkipListTbl.T` uses its own random number generator. The generator is initialized with a fixed seed if and only if the `fixedSeed` parameter is `TRUE`. Use of a fixed seed is only recommended for testing purposes.

The implementation calls the `keyCompare` method to compare two keys. The default `keyCompare` method simply returns `Key.Compare(k1, k2)`. However, subtypes may wish to override the `keyCompare` method to effect a new key ordering. `keyCompare` is required to implement a total order.

The `iterate` method returns an iterator of type `Iterator`, a subtype of `SortedTbl.Iterator`. Its `reset` method resets the iterator. This allows clients to iterate over a table multiple times without having to allocate a new `Iterator` object on each pass.

3.2 Synchronization

For efficiency, skip list tables and their iterators are not monitored, so a client accessing a table from multiple threads must ensure that if two operations are active concurrently, then neither of them has side-effects on the same table or iterator. The `init`, `put`, and `delete` methods are the only ones with side-effects on the table. All three of an iterator's `reset`, `next`, and `seek` methods have side-effects on the iterator.

3.3 Quake Instantiation Procedures

The `sortedtableextras` package includes a quake template that defines quake procedures for instantiating instances of the `SkipListTbl` generic interface and implemenation. The two procedures are:

```
skiplist_table (nm, key, value)
SkipList_table (nm, key, value)
```

The only difference between these two procedures is that tables instantiated by the former are private to the package in which they are built, while those instantiated by the latter are exported.

These procedures create and include the two generic instantiation files `SkipList<nm>Tbl.i3` and `SkipList<nm>Tbl.m3`. The generic interface and implementation are instantiated with the interfaces named `key` and `value`. `nm` should be a string representing the concatenation of the names `key` and `value`, possibly in abbreviated form; it must be the same name that is used to instantiate the generic `Table` and `SortedTable` interfaces. Here are some examples:

```
skiplist_table ("IntInt", "Integer", "Integer")
skiplist_table ("IntText", "Integer", "Text")
```

```
skiplist_table ("RealRef", "RealType", "Refany")
```

For example, the last procedure call would create the two derived files `SkipListRealRefTbl.i3` and `SkipListRealRefTbl.m3`.

In order for a program that includes a `SkipListTbl` instantiation to link successfully, it must also instantiate the generic `Table` and `SortedTable` interfaces with the same `nm`, `key`, and `value` arguments.

3.4 Performance and Implementation

A skip list table's `get`, `put`, and `delete` methods take `O(log n)` expected time, where `n` is the number of elements in the table. The other table methods take constant time. An iterator's `reset`, `next`, and `seek` methods also take `O(log n)` expected time.

Skip lists were invented by William Pugh [5, 4]. This implementation of skip lists uses:

- A `p` value of 1/4 as recommended in Pugh's papers.
- An extra back-pointer per node to allow downward iterations.
- The extra test desribed in section 3.5 of Pugh's "Cookbook" paper [4] for minimizing the number of key comparisons. If key comparisons are cheap, including this test is unnecessary and hurts performance slightly, but in a generic implementation where the cost of key comparisons is potentially unbounded, including the test seems prudent.

There is no well-defined value of type `Key.T` that exceeds all other keys. Hence, the use of a "nil" sentinel as described in Pugh's papers could not be used. Instead, some extra tests against `NIL` are required.

The space requirements of a skip list table are dominated by the space costs for each of its entries. The space required for each entry is the space for the key and the value plus the space for the foward and backward `REF`s. According to Pugh, the expected number of forward `REF`s per entry with a value for `p` of 1/4 is 1.333. However, since the number of forward `REF`s per entry may vary, and since this is a safe implementation, the forward `REF`'s are represented as a `REF ARRAY` of `REF`s. Hence, each node requires an extra `REF` for the `REF ARRAY` plus the runtime's space overhead for the `REF ARRAY` itself, which includes its typecode and its size. All told then, the expected number of `REF`s per entry is 3.333, and there is an additional space cost per entry of the runtime `REF ARRAY` space overhead.

4 Performance

This section describes the performance of the `Table.Default`, `SortedTable.Default`, `RedBlackTbl.T`, and `SkipListTbl.T` implementations. All performance experiments were made on a Digital AlphaStation 400 workstation equipped with a 233 megahertz DECchip 21064 processor running Digital Unix version 4.0. The tests were performed on an unloaded machine, and all code was compiled with optimization.

We instantiated each kind of table with integer keys and integer values. Table 1 shows the average elapsed times in microseconds of various operations performed on these tables. Each experiment was performed 10 times; the table shows the mean elapsed time for each operation.

Three kinds of experiments were performed on each type of table, denoted by the labels *Random*, *Increasing*, and *Decreasing*. As described in more detail below, the only difference between the three kinds of experiments is the keys that were used, and the order in which elements were inserted into and deleted from the tables.

Experiment	Insert	Search	Iter Up	Iter Down	Seek Up	Seek Down	Delete
`Table.Default`							
Random	12	2	2		N/A	N/A	3
Increasing	20	6	1		N/A	N/A	2
Decreasing	21	6	1		N/A	N/A	2
`SortedTable.Default`							
Random	40	15	1	1	10	10	14
Increasing	22	18	1	1	10	10	6
Decreasing	24	24	1	1	11	11	6
`RedBlackTbl.T`							
Random	30	5	1	1	8	9	10
Increasing	34	11	1	1	8	9	6
Decreasing	35	11	1	1	8	9	6
`SkipListTbl.T`							
Random	74	14	1	1	23	23	24
Increasing	58	42	1	1	23	24	6
Decreasing	54	24	1	1	22	23	15

Table 1: Average elapsed times in microseconds of various operations on each of the four table implementations.

In the Random experiment, 100,000 keys were chosen at random from the interval [1, 100000] and inserted into the table. Due to collisions, the resulting table almost certainly contained fewer than 100,000 elements. In the Increasing experiment, the keys 1 through 100,000 were inserted into the table in increasing order; in the Decreasing experiment, the same set of keys were inserted in decreasing order.

After the keys were inserted, several more tests were performed on each table:

- A *search* was done on the keys 1 through 100,000 in order.
- Upward and downward *iterations* were done on all of the elements of each table. The single time shown for the `Table.Default` implementation is the time required for unordered iteration, since the type `Table.T` does not support ordered iteration.
- 100,000 upward and downward *seeks* were done on each table. The `i`th key to seek for was calculated as `(i * 23) MOD 100000`. This test has the effect of striding over the keys of the table in order, but wrapping back to the start (or end if seeking downward) of the table roughly 23 times. These times are not reported for the `Table.Default` implementation because the type `Table.Iterator` does not provide a `seek` method.

- 100,000 *deletions* were performed using the same keys and key order as were used in the insertion test. In the case of the Random experiment, some of those deletions were undoubtably no-ops, but 100,000 deletion operations were performed nonetheless.

The Random test is probably the most important of the three. The Random results show that, as expected, the simple hash tables are significantly faster than the tables supporting ordered iteration. The Random results also show that the red-black tree implementation is somewhat faster than the treap implementation, and significantly faster than the skip list implementation.

We ran the Increasing and Decreasing tests to measure the performance of the red-black implementation on its worst-case input. Even when given its worst-case input, the red-black implementation does not perform much worse than the treap implementation, and it is still substantially faster than the skip list implementation.

Table Type	Interface	Implementation
Table.Default	143	287
SortedTable.Default	109	433
RedBlackTbl.T	118	487
SkipListTbl.T	172	311

Table 2: The size of each generic interface and implementation module in lines of code.

What price is there to be paid in code complexity for the increased performance offered by red-black trees? As shown in Table 2, red-black trees do have the longest implementation when measured in lines of code. However, the implementation of red-black trees is only 12% longer than that of treaps and 57% longer than that of skip lists. The red-black implementation is indeed a bit more subtle than the skip list implementation, but the increased code size is due partly to the replicated code necessary to handle left and right cases. We spent only an hour or so longer debugging the red-black code than we did debugging the skip list code.

5 Memory Use

This section describes the memory use of the `SortedTable.Default`, `RedBlackTbl.T`, and `SkipListTbl.T` implementations. The results are summarized in Table 3. For each implementation, we distinguish the per-element memory size from that of the object itself. These values do not include standard heap-allocated object overheads such as typecodes and method tables, but they do include an extra 1-word cost for storing the number of elements in each heap-allocated array.

The exact sizes of the data structures used in each implementation are architecture dependent. For example, the size of a reference (pointer) is architecture dependent, as are any alignment constraints imposed on the way records are layed out in memory. Another factor that contributes to each table's memory usage is the variable size of the key and value types with which the table is parameterized.

The data in Table 3 has thus been parameterized by the declared maximum number of elements N in the table, the sizes K and V of the key and value types, the size R of a reference, and the size

Implementation	Per-Element Size	Object Size
`SortedTable.Default`	$K+V+2R+W$	$5R+W+(55W+1)$
`RedBlackTbl.T`	$K+V+3R+1$	$2R+W$
`SkipListTbl.T`	$K+V+3.33R+W$	$(3+\lceil\log_4 N\rceil)R+6W+(55W+1)$

Table 3: Memory usage of the various implementations in bytes. In this table, K and V denote the space required to store the table key and value, respectively, R denotes the size of a reference, W denotes the word size `BYTESIZE(Word.T)`, and N denotes the declared maximum number of nodes in the table.

W of a machine word. For example, on a 64-bit Digital Alpha machine, R and W are both 8, while on a DECStation, they are both 4.

A couple of points about this data are worth mentioning. In the `RedBlackTbl.T` node size, the $+1$ term is the cost of recording the color (red or black) of the node. In the `SkipListTbl.T` node size, the $3.33R$ term is the *expected* number of references per node: one for a back pointer, one for the reference to an array of forward pointers, and 1.33 for the expected size of the forward pointer array. The W term in that expression is the cost of storing the number of elements in the dynamic forward pointer array. In the `SkipListTbl.T` object size, the $\lceil\log_4 N\rceil R$ term is the size of the object's main forward pointer array; again, one word of cost has been added to account for the dynamic size of the array. Finally, the $55W+1$ term in the sizes of the `SortedTable.Default` and `SkipListTbl.T` objects is the size of a nested `Random.Default` object required by those implementations.

	Per-Node Size		Object Size	
Implementation	Theory	Actual	Theory	Actual
`SortedTable.Default`	40	40	489	496
`RedBlackTbl.T`	41	48	24	24
`SkipListTbl.T`	50.6	50.6	553	560

Table 4: Theoretical and actual sizes of the three implementations in bytes on a Digital Alpha workstation for sorted tables mapping integers to integers.

Table 3 reports the minimum number of bytes required for each data structure, ignoring alignment constraints. However, due to alignment constraints, the actual memory requirements may be somewhat higher. Table 4 shows the theoretical and real sizes of the three implementations on a Digital Alpha workstation assuming a maximum table size of $N = 1000$ and using integer keys and values (so $K = V = 8$).

6 Conclusions

Of the three table implementations we measured that support ordered iteration, red black trees outperformed both treaps and skip lists. Moreover, the red black implementation guarantees logarithmic *worst case* performance, while the other two implementations have only *expected* logarithmic performance.

Although the red black implementation is slightly longer and more complicated than the other implementations, the implementation described here took only a day to implement and test. Because all three sorted tables were implemented using Modula-3's generic interfaces and modules, their implementations can be easily reused.

References

[1] Cecilia Aragon and Raimund Seidel. Randomized Search Trees. In *Proceedings of 30th Foundations of Computer Science (FOCS)*, pages 540–545, October 1989.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[3] Jim Horning, Bill Kalsow, Paul McJones, and Greg Nelson. Some Useful Modula-3 Interfaces. SRC Research Report 113, Digital Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, December 1993. Also available at `http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-113.html`.

[4] William Pugh. A Skip List Cookbook. Technical Report CS-TR-2286.1, Department of Computer Science, University of Maryland, College Park, April 1989. Also available in file `ftp://ftp.cs.umd.edu/pub/skipLists/cookbook.ps`.

[5] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, June 1990. Also available in file `ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.ps`.