Efficient Implementation of Red-Black Trees with Split and Catenate Operations

Ron Wein
Tel-Aviv University
`wein@post.tau.ac.il`

Abstract

We describe the implementation of the `Multiset` class-template within the support library of Cgal, the Computational Geometry Algorithms' Library. The interface of this class is inspired by the `multiset` class-template included in the standard template library (Stl). Both class templates are implemented in terms of red-black trees, yet they differ from one another quite substantially. In this report we highlight the differences between the two class templates and show the advantages of our implementation.

1 Introduction

We have recently developed a multiset class-template[1] based on a proprietary red-black tree implementation. Our class template has a similar interface to the Stl `multiset` class, which is also implemented using red-black trees, but is significantly different from it. Not only it strives to minimize the number of comparison operations invoked by its interface functions, it also introduces some new functionality which is non-existent in the standard `multiset` class. In this report we highlight the differences between the two classes and give the implementation details of the additional operations we support.

As our class is part of the support library of Cgal, the Computational Geometry Algorithms' Library.[2] It is mainly intended for storing ordered collection of geometric objects that are represented using a number type that guarantees the robustness of the geometric operation (e.g., in case of linear objects we use a rational number type whose numerator and denominator are unbounded integers). In such scenarios, comparing two geometric entities is by no means a constant time operation, and usually depends on the bit-length of the input. We therefore keep track of the number of comparison operations each tree operation invokes and analyze its running time accordingly, where we use C to denote the average time consumed by a single comparison operation. Following this analysis, we explain how our implementation of red-black trees can help programmers achieve faster running times.

We have recently re-implemented the sweep-line algorithm in the arrangement package of Cgal [FWH04]. This algorithm heavily relies on red-black trees to efficiently maintain its auxiliary data structures; see, e.g., [dBvKOS00, Chapter 2]. Switching to `CGAL::Multiset` class instead of using the `std::multiset` helped reducing the running times by almost a factor of 2. Moreover, as

[1]The term *multiset* refers to a collection of elements, where as opposed to a regular set, equivalent elements (or even identical elements) may occur in the collection.

[2]The Cgal Project Homepage, `http://www.cgal.org/`.

our multiset class introduces extra functionality, it was recently used for implementing the snap-rounding algorithm of de Berg *et al.* [dBHO05], which requires efficient split and catenate operations on ordered sets.

The rest of this report is organized as follows. In Section 2 we review the red-black tree data-structure. In Section 3 we emphasize all the differences between our `CGAL::Multiset` class and the `std::multiset` class. Finally, in Section 4 we give the details of our implementation, concentrating on the additional set operations we support.

2 Preliminaries

A *red-black tree* is a balanced binary tree, where each tree node stores an extra *color* attribute, and such that the following invariants, known as the *red-black properties*, are always maintained:

- Every node is either *red* or *black*.
- If a node is red, then both its children are black.
- Every simple path from a node to a descendant leaf contains the same number of black nodes.

From these properties, it follows that the height of a tree containing n elements is bounded by $2\log_2(n+1)$. Thus, finding a node that contains an element that is equivalent to a query element takes $O(C\log n)$ time, as it involves $O(\log n)$ comparison operations. The elementary insertion and deletion operations of a single element also take $O(C\log n)$ time, for the same reasons.

It should be noted that it is possible to have red leaves in the tree, or red nodes that have only one valid child. In order to satisfy property (2), we treat null nodes as if they are black. The *black height* of a node ν is defined recursively:

$$\mathrm{black\_height}(\nu) = \begin{cases} 0 & \nu \text{ is null} \\ \mathrm{black\_height}(\mathrm{right}(\nu)) & \nu \text{ is red} \\ 1 + \mathrm{black\_height}(\mathrm{right}(\nu)) & \text{otherwise} \end{cases}$$

The black-height of the tree is the black height of the root node (for example, the black height of the tree shown in Figure 1 is 2). We always keep the node black, so any non-empty tree has a black-height of at least 1.

When we insert a node into a red-black tree (or when deleting a node from a tree), we may temporarily violate the red-black properties. To remedy the situation, we apply an *insertion fix-up* (or *deletion fix-up*) procedure, which restores the red-black properties by re-coloring nodes and applying local rotation operations, where each such elementary operation takes constant time. It is important to observe that the fix-up procedures do not invoke any comparison operations, and it can be shown that they perform an amortized constant number of re-coloring and rotation operations [Tar83b], thus they do not add to the asymptotic time complexity of the insertion and deletion operations. More important, if one knows the exact location for the element one wishes to insert (or the exact location of the element one wishes to delete), the insertion (deletion) operation can be carried out in $O(1)$ amortized time.

The red-black data-structure was designed by Bayer [Bay72] under the name "symmetric binary B-tree". Guibas and Sedgewick [GS78] gave a detailed analysis of the properties and performance of the data structure and introduced the red/black color convention. Cormen *et al.* [CLRS01, Chapter 14] give an excellent introduction on red-black trees and bring detailed pseudo-code for the elementary tree operations. Tarjan [Tar83a] describes the supplementary *split* and *catenate* operations, which we review in the next section.

3 Multiset Interface

In this section we describe the design of the CGAL multiset class, which is based on an efficient implementation of a red-black tree data structure. The interface of the `CGAL::Multiset<Type, Compare, Allocator>` class-template has the look-and-feel of the STL multiset class-template (i.e., `std::multiset<Type, Less, Allocator>` — see, e.g., [MS96, Section 19.9]). However, there are some fundamental differences between the two class templates, which we highlight in this section.

3.1 Three-Valued Comparison Functor

The main difference between the two class templates is that `std::multiset` is templated by a less-than functor with a Boolean return type, while `CGAL::Multiset` is parameterized by a comparison functor `Compare` that returns the three-valued `Comparison_result` enumeration type (namely it returns either `SMALLER`, `EQUAL`, or `LARGER`). It is thus sufficient to test equality between two elements using one comparison operation, instead of checking whether `(! Less()(x, y) && ! Less()(y, x))`. We are therefore able to maintain the underlying red-black tree with less invocations of the comparison functor, which can considerably decrease running times, especially when comparing elements of type `Type` is an expensive operation.

The usage of the three-valued comparison functor also enables us to guarantee that the order of elements sent to the comparison functor is fixed. For example, if we insert a new element `x` into the set (or look-up an element in the set), then we always invoke `Compare()(x, y)` (and never `Compare()(y, x)`), where `y` is an element already stored in the set. This behavior, which clearly cannot by supported by `std::multiset`, is sometimes crucial for designing more efficient comparison predicates.

3.2 In-Place Insertion

The red-black tree representation is encapsulated in the multiset class, where tree nodes can be implicitly accessed via *iterators* (see, e.g., [MS96, Chapter 18]). As we mentioned in the previous section, it is more efficient to insert a new element into a set if we know the place for it. The function `insert(pos, x)` inserts a new element `x` into the set, where `pos` is a multiset iterator pointing to some existing tree node, serving as an insertion *hint*. This function operates in $O(C \min\{d+1, \log n\})$ time, where d is the distance between the hint node and the true position of `x` in the tree.[3] Note that in any case, this insertion function uses at least two comparison operations, as it has to make sure that the new element lies strictly between the element stored at `pos` and its successor.

In addition to the standard insertion functions taken from the `std::multiset` interface, our `CGAL::Multiset` class-template provides two supplementary insertion functions, named `insert_before(pos, x)` and `insert_after(pos, x)`. These functions accept an iterator `pos` that specifies the immediate successor node (or immediate predecessor node) of `x` in the set, namely the *exact* insertion position is given. These functions do not perform any comparison operation and their amortized running time is therefore constant. In case the comparison operation is time

[3]Note that to bound the running time, we start proceeding from the hint node, but if after black_height$(T) = O(\log n)$ steps we come to the conclusion hint is not useful, we and switch to a normal $O(C \log n)$ insertion process.

consuming, users can considerably reduce the running time of their application by using these specialized insertion functions, whenever possible.

3.3 Key Look-Ups

The interface of the `std::multiset` class includes several *look-up* functions, such as `count(x)`, which returns the number of elements in the set that are equal to `x` (the other look-up functions are `find(x)`, `lower_bound(x)`, `upper_bound(x)`, and `equal_range(x)` — see [MS96] for the exact specification of their interface). The running time of all the look-up functions is $O(C \log n)$ (or, in the case of `count(x)` it is $O(C(\log n + m))$, where m is the number of elements that equal `x`).

Our `CGAL::Multiset` class supports all these functions, but it also introduces a complementary set of look-up functions that accept a *key* instead of a concrete element. Namely, these look-up functions accept a key instance of type `Key`, which may differ from `Type` (the type of elements stored in the set), as long as users supply a comparison functor `CompareKey`, where `CompareKey()(key, y)` returns a `Comparison_result` value (`key` is the look-up key and `y` is an element of type `Type`). It is essential of course that the `CompareKey` operates in a way that is consistent with the tree order.

Indeed, it is very convenient to look-up equivalent objects in the set given just by their key. For example, consider the following structure that we use to represent employee records:

```
struct Employee {
  std::string      name;
  std::string      title;
  unsigned int     salary;
  // Other data fields ....
};
```

We also define the following set of employee records, sorted alphabetically:

```
struct Employee_compare {
  CGAL::Comparison_result operator() (const Employee& emp1,
                                      const Employee& emp2) const
  {
    int    res = strcmp (emp1.name.c_str(), emp2.name.c_str());

    if (res == 0)
      return (CGAL::EQUAL);
    else if (res < 0)
      return (CGAL::SMALLER);
    else
      return (CGAL::LARGER);
  }
};

CGAL::Multiset<Employee, Employee_compare>   employees;
```

Suppose that we wish to look for an employee record given its name. Had we used `std::multiset`, we should have defined an `Employee` instance, fill in the name field to be the query string (the content of the rest of the fields is of course unimportant) and use one of the standard look-up functions. However, in our case we should simply define the following functor:

```
struct Employee_name_compare {
  CGAL::Comparison_result operator() (const std::string& name,
                                          const Employee& emp) const
  {
    int    res = strcmp (name.c_str(), emp.name.c_str());

    if (res == 0)
      return (CGAL::EQUAL);
    else if (res < 0)
      return (CGAL::SMALLER);
    else
      return (CGAL::LARGER);
  }
};
```

We then use this functor for performing simple queries on our set, such as `employees.find ("John Smith", Employee_name_compare())`.

We should also mention that for some applications it is necessary, not just convenient, to use the key look-up functions. Consider for example a set of planar non-vertical line segments, all defined over some x-coordinate x_0 and sorted by their ascending y-order over x_0. We now wish to locate a point $q = (x_0, y_0)$ with respect to this set, namely to identify the line segments lying immediately above or below it. It is clear that since a point and a line segment are represented using different classes, this operation is only possible by a key look-up of q in the set.

3.4 Additional Multiset Operations

While the STL multiset does not supply efficient operations between different sets, our class offers to additional operations:

Catenate: Given two sets S_1 and S_2 such that for each $x \in S_1$ and $y \in S_2$ we have $x \leq y$, it is possible to "glue" S_2 after S_1 and obtain a consolidated set. Such an operation is known as *catenation*, and can be carried out in amortized $O(\log \frac{n_2}{n_1})$ time, where n_2 is the size of the set having a larger (or equal) number of elements and n_1 is the size of the other set.

Split: Given a set S and an element x (it is possible that $x \notin S$), split the set to S_1 and S_2, where all elements in S_1 are less than x and all elements in S_2 are greater or equal to it. This operation can be performed in $O(C \log n)$ time, where $n = |S|$.

We give the implementation details of these two functions in Sections 4.2 and 4.3.

3.5 Storing the Black Height

In order to conveniently implement the `catenate()` and `split()` functions, it is useful to store the black height of the tree (see Section 2) as a data member in the `Multiset` class. The value of this data member can obviously change only when we insert or delete elements, and can be maintained within the insertion fix-up and the deletion fix-up procedures using a constant number of operations.

Having the black_height(T) property of a tree T available also help us to bound the running time of some of the tree operations so it is $O(C \log n)$. For example, the `equal_range(x)` function returns a pair of iterators which define a range equivalent to `[lower_bound(x),upper_bound(x))`. We implement this function as follows: We locate `lower_bound(x)` and proceed from there as long as

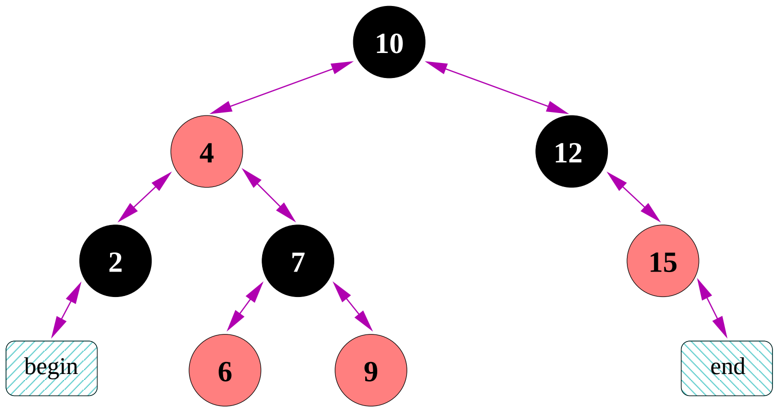

Figure 1: A red-black tree representing a set of eight integers. The tree consists of eight valid nodes and two fictitious *begin* and *end* nodes.

we encounter elements equivalent to x. If, however, we find more than black_height$(T) = O(\log n)$ elements, we stop and compute `upper_bound(x)` directly. This implementation is more efficient when the number of elements equivalent to x in the tree is small.

4 Implementation Details

We should first mention that in some of the related literature, the data structure is defined such that the actual data elements are stored just in the leaf nodes, where the internal tree nodes contain only *keys* that enable the location of elements by simple comparison operations. For example, if our elements are employee records that contain the ID (say, social security number), name, title and salary of each employee, the keys stored in an internal node may be the minimal ID in the sub-tree rooted at the right child of this node. Indeed, such a representation is more convenient for the description and implementation of some algorithms while it does not incur any asymptotic run-time or memory consumption. However, it may increase the memory occupied by the tree by a constant factor of 2. For practical consideration, this overhead is a good enough reason to prefer the red-black tree representation given in [CLRS01], where each valid tree nodes stores a multiset element.

4.1 Null Nodes and Fictitious Nodes

Cormen *et al.* [CLRS01] recommend having a sentinel null node. Thus, each tree instance has its own null sentinel, which is of course colored black, and all valid nodes that have a null child store a pointer to this sentinel. However, as in some cases we need to move nodes from one tree instance to another (see the next subsections), such a representation will spoil the nice asymptotic running-time bounds these operations have. We therefore allow `NULL` node pointers in the tree, but we define an internal `_is_black(Node *ptr)` predicate, which returns `true` if `ptr == NULL` or if it points to a valid tree node.

As we mentioned before, tree nodes can be implicitly accessed via iterators. For example, the `begin()` function returns an iterator pointing to the leftmost tree node, while `end()` returns an iterator pointing past the rightmost tree node. It is possible to increment (or decrement) an iterator in amortized constant time and reach its successor (or predecessor) in the tree. It is not difficult to see how to implement `begin()` in $O(\log n)$ time, but according to the STL requirements we need a constant-time operation.

To this end, we introduce two extra node members called `beginNode` and `endNode` in each tree instance. The parent nodes of these fictitious tree nodes are the leftmost and the rightmost tree nodes, respectively (or `NULL` if the tree is empty). In addition, the left child of the leftmost tree node points to `beginNode` while the right child of the rightmost node points to `endNode` (see Figure 1 for an illustration). To designate the fictitious begin and end roots, we abuse the `Color` enumeration by introducing two dummy colors on top of `RED` and `BLACK`. The auxiliary `Node` structure nested in the CGAL multiset class therefore has the following structure:

```
struct Node
{
  enum Color {RED, BLACK, DUMMY_BEGIN, DUMMY_END};

  Type         object;       // The stored element.
  Color        color;        // The color.
  Node        *parentP;      // Points to the parent node.
  Node        *rightP;       // Points to the right child of the node.
  Node        *leftP;        // Points to the left child of the node.

  // Check if the node is valid (not a dummy node).
  bool is_valid () const { return (color == BLACK || color == RED); }

  // Check if the node is red.
  bool is_red () const {return (color == RED); }

  // Check if the node is black.
  // Note that invalid nodes are also regarded as if they are black.
  bool is_black () const {return (color != RED); }

  // Get the previous node in the tree (according to the tree in-order).
  Node* predecessor () const
  {
    if (leftP != NULL)
      // If there is a left child, the predecessor is the rightmost node in
      // the sub-tree rooted at this child.
    else
      // Otherwise, go up the tree until reaching the parent from a right
      // child (this also covers the case of the DUMMY_END node).
  }

  // Get the next node in the tree (according to the tree in-order).
  Node* successor () const
  {
    if (rightP != NULL)
      // If there is a right child, the successor is the leftmost node in
      // the sub-tree rooted at this child.
    else
      // Otherwise, go up the tree until reaching the parent from a left
      // child (this also covers the case of the DUMMY_BEGIN node).
  }
};
```

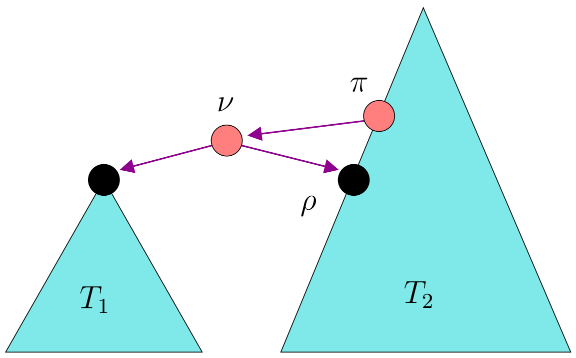

Figure 2: Catenating two red-black trees T_1 and T_2, such that $\text{black\_height}(T_1) < \text{black\_height}(T_2)$.

Note that the `predecessor()` and `successor()` methods work seamlessly for the fictitious node. Thus, `begin()` simply returns the parent of the fictitious begin node, while `end()` returns the fictitious end node. When we decrement the past-the-end iterator, we get to the rightmost tree node, just as the STL standard specifies.

4.2 Implementing the Catenation Operation

Tarjan [Tar83a], who gives the details of the catenate and split procedures for red-black trees, uses a variant of the data structure that stores data elements only in the tree leaves, where internal nodes only contain keys. Using this representation, it is possible to give a simple representation of the catenate operation.

We are given two sets S_1 and S_2, represented as red-black trees T_1 and T_2, respectively, such that for each $x \in S_1$ and $y \in S_2$ we have $x \leq y$. Assume, without loss of generality, that $\text{black\_height}(T_1) \leq \text{black\_height}(T_2)$. We perform the following procedure (see Figure 2 for an illustration).

1. Go down from $\text{root}(T_2)$ along the leftmost path in the tree, until reaching a black node ρ such that $\text{black\_height}(\rho) = \text{black\_height}(T_1)$.

2. Let $\pi \longleftarrow \text{parent}(\rho)$.

3. Make $\text{root}(T_1)$ and ρ siblings by creating a new node ν, and setting $\text{left}(\nu) \longleftarrow \text{root}(T_1)$, $\text{right}(\nu) \longleftarrow \rho$ and $\text{color}(\nu) \longleftarrow \text{RED}$.

4. If $\pi = \text{NULL}$, set $\text{root}(T_1) \longleftarrow \nu$.
   Otherwise, set $\text{parent}(\nu) \longleftarrow \pi$ and $\text{root}(T_1) \longleftarrow \text{root}(T_2)$.

5. Perform the insertion fix-up procedure from the node ν.

6. Mark T_2 as empty by setting $\text{root}(T_2) \longleftarrow \text{NULL}$.

The result of this procedure is a consolidated set S which is represented by the catenated red-black tree T_1. The running time of the catenation procedure is clearly dominated by Step (1) and is therefore $O(\log n_2 - \log n_1) = O(\log \frac{n_2}{n_2})$, where $n_1 = |S_1|$ and $n_2 = |S_2|$. Note however that if the node ρ is known in advance, the rest of the steps can be carried out in constant amortized time.

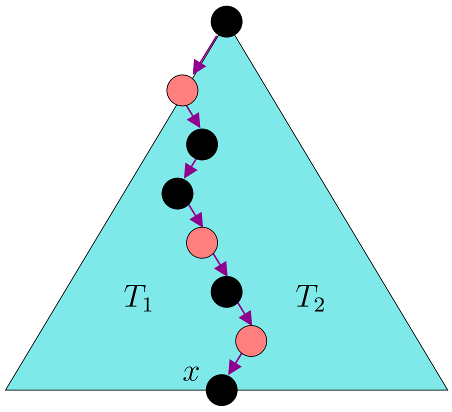

Figure 3: Splitting a red-black tree into two sets T_1 and T_2, such that all elements of T_1 are smaller than x and all elements of T_2 are equal or greater than x. We catenate subtrees along the marked path from the tree root to the leaf node containing x.

Since in our representation all valid tree nodes store valid elements, it seems problematic, at first glance, to introduce a new node ν "out of the blue". We therefore detach the leftmost node of T_2 from its position,[4] color it red and use it as our *pivot node* (the node ν in the pseudo-code above). Note that the element stored in the pivot node is clearly larger than every element in T_1 (which now becomes its left subtree) and smaller than every element in the subtree rooted at ρ. The rest of the procedure follows the pseudocode given above.

4.3 Implementing the Split Operation

As in the previous section, we begin by describing the split algorithm as it appears in [Tar83a]. We are given a set S represented by a red-black tree T and an element x (which is not necessarily contained in the set), and construct two red-black trees T_1 and T_2 as follows:

```
1. Let $T_1 \longleftarrow \emptyset$, and $T_2 \longleftarrow \emptyset$.
2. Let $\kappa \longleftarrow$ root($T$).
3. While $\kappa$ is not a leaf node do:
     • If $x <$ key($\kappa$), then:
       Catenate the subtree rooted at right($\kappa$) to $T_2$.
       Let $\kappa \longleftarrow$ left($\kappa$).
     • Otherwise (if $x \geq$ key($\kappa$)):
       Catenate the subtree rooted at left($\kappa$) to $T_1$.
       Let $\kappa \longleftarrow$ right($\kappa$)
4. Insert the element stored at $\kappa$ into $T_1$ if it is less than $x$,
   otherwise insert it into $T_2$.
```

[4]Note that we can reach and detach this node in constant time, and apply a deletion fix-up procedure in constant amortized time to make sure that T_2 maintains the red-black properties.

The resulting trees represent two sets S_1 and S_2, such as all S_1's elements are smaller than x and all elements of S_2 are equal or greater than x. The main loop (Step 3 of the algorithm) performs $O(\log n)$ comparison operations and catenation operations. However, as we go down the tree we can easily go simultaneously down the rightmost path of T_1 and the leftmost path of T_2 and locate nodes that have the same black height as κ. Thus, each catenation operation is carried out in constant amortized time, and the total running time of the entire procedure is $O(C \log n)$.

Note that the procedure above discards all the internal nodes that lie on the path it traverses (see Figure 3 for an illustration), as they are not inserted to either of the two resulting trees. This is of course acceptable as there are no data elements in these internal node. However, since in our implementation these nodes contain concrete data elements that are an integral part of the set S, we have to take care of them as well. Luckily, a simple augmentation of the split algorithm can help us dealing with the nodes, as well as keeping the catenation process simpler.

The main idea is to introduce two auxiliary pivot nodes — a left pivot ν_1 and a right pivot ν_2. Initially, both these nodes are null. We now modify Steps 3 and 4 in the split procedure above, resulting in the following procedure:

```
1. Let $\langle T_1, \nu_1 \rangle \longleftarrow \langle \emptyset, \textsc{Null} \rangle$, and $\langle T_2, \nu_2 \rangle \longleftarrow \langle \emptyset, \textsc{Null} \rangle$.
2. Let $\kappa \longleftarrow \text{root}(T)$.
3. While $\kappa \neq \textsc{Null}$, do:
     • If $x < \text{element}(\kappa)$, then:
       Catenate the subtree rooted at right($\kappa$) to $T_2$ using the pivot $\nu_2$.
       Let $\nu_2 \longleftarrow \kappa$.
       Let $\kappa \longleftarrow \text{left}(\kappa)$.
     • Else if $x > \text{element}(\kappa)$, then:
       Catenate the subtree rooted at left($\kappa$) to $T_1$ using the pivot $\nu_1$.
       Let $\nu_1 \longleftarrow \kappa$.
       Let $\kappa \longleftarrow \text{right}(\kappa)$
     • Otherwise ($x = \text{element}(\kappa)$):
       Catenate the subtree rooted at left($\kappa$) to $T_1$ using the pivot $\nu_1$, and set $\nu_1 \longleftarrow \textsc{Null}$
       Catenate the subtree rooted at right($\kappa$) to $T_2$ using the pivot $\nu_2$, and set $\nu_2 \longleftarrow \textsc{Null}$.
       Insert $\kappa$ as the minimum node of $T_2$.
       End the split procedure.
4. If $\nu_1 \neq \textsc{Null}$, insert it as the maximum node of $T_1$.
   If $\nu_2 \neq \textsc{Null}$, insert it as the minimum node of $T_2$.
```

Note that as we have our catenation pivots available, we do not have to splice out nodes from T_1 or T_2 to perform the catenation operations, as we did in the previous section. The reader can easily verify that using the pivots does not violate the tree-order properties.

It is of course possible to send the `split()` function a key object, along with a proper comparison functor, instead of sending it an element (see Section 3.3). In addition, users may send the function an iterator pointing to the tree node that contains the element x. In this case, it is possible to go up to the tree root and apply the split algorithm from there without invoking any comparison operations. The running time of the split operation in this case is $O(\log n)$.

In this context, we should mention that we keep an integer field with each tree instance indicating the tree size (the number of elements it contains), so the size of a multiset can be queried at $O(1)$

time. Updating the value of this field in constant time after insertion, deletion of catenation operations is trivial. However, when we split a set into two, we cannot set the sizes of S_1 and S_2 unless we traverse both sets and count the number of elements in each one, which requires a time linear in the size of the original set. In order to keep the logarithmic running time, we use a "lazy evaluation" technique: we set an invalid size value for S_1 and S_2, and only if the user invokes the `size()` function on one of these multisets, we go over its tree, count the number of nodes and update the size field.

References

[Bay72] R. Bayer. Symmetric binary B-trees: Data structures and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 2nd edition, 2001.

[dBHO05] M. de Berg, D. Halperin, and M. Overmars. An intersection-sensitive algorithm for snap rounding. Manuscript (avaialble on `http://www.cs.tau.ac.il/~danha/papers/issr.pdf`), 2005.

[dBvKOS00] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin, Germany, 2nd edition, 2000.

[FWH04] E. Fogel, R. Wein, and D. Halperin. Code flexibility and program efficiency by genericity: Improving CGAL's arrangements. In *Proc. 12th European Symposium on Algorithms*, pages 664–676. Springer-Verlag, 2004.

[GS78] L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symp. Foundations of Computer Science*, pages 8–21. IEEE Computer Society, 1978.

[MS96] D. R. Musser and A. Saini. *STL Totutial and Reference Guide: C++ Programming with the Standard Template Library.* Addison-Wesley, Reading, MA, 1996.

[Tar83a] R. E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[Tar83b] R. E. Tarjan. Updating balanced search tree in $O(1)$ rotations. *Information Processing Letters*, 16, 1983.