

Designing a Generic String Class for the CPH STL

Filip Bruman

Abstract. The C++ `basic_string` is closely tied to its historical roots of the C-string. Since the introduction of the STL, the `basic_string` has evolved into a template class, but it still utilizes an array as its data structure. It offers, due to backwards compatibility, functionality that—in the advent of the STL—has become superfluous. These and other aspects of the `basic_string` class are not easily susceptible to change. The goal of the CPH STL project is to study, analyse and design enhanced versions of STL components. In the view of this, it is the aim of this report to analyse, design and to some degree implement a `basic_string` class for CPH STL using the STL counterpart as a reference. In an attempt to make the CPH STL string allow for more flexibility and variety in use, the design employ the ability of an optional inherent data structure. To accomplish this the string class is split into two separate classes; a core and a utility, attaining a modularity of the string, not only in the respect to the underlying data structure but also regarding implementation of functionality. In this report the core class utilizes a red-black tree as a data structure—recognized for its sub-linear time on selected operations. As a specialization the design lets strings share references and only to create new copies when written to, as it is seen in other object-oriented languages (e.g. Java and C#), thus rendering the string fully immutable—contrary to that of the C++ standard library string. A rudimentary implementation of the string class designed is trying to form a seminal basis for future projects.

1. Background

One of the research proposals posed in [12] was a string container and in addition, to consider generic, *alphabet independent* (conform to no alphabet in particular) versions of traditional string manipulation operations and algorithms to be effective on both large and small alphabets (e.g. for Unicode and DNA sequences).

Implementing the CPH STL string class as a container makes it, to a greater extent than the C++ standard library string, in accordance with the C++ STL.

In Section 1 of this report the various concepts used throughout the report are introduced. This is followed by a description of the ideas behind the design of the CPH STL string class (henceforth abbreviated as CSS). Design decisions made in relation to the C++ standard library string (henceforth abbreviated as SLS), and the outcome and effects are presented in Section 2. A discussion of the end result and some thoughts of applicability is found in Section 3.

1.1 *Generic programming*

When writing software code a common technique for avoiding redundant work is to reuse code through software libraries. When writing components for a software library it is desirable to design the components of that library to be applicable to an entire class or group so as to be easily reusable without loosing efficiency. The reuse of components can be achieved by means of using a programming language that allows for *polymorphism*, meaning that some values and variables may assume more than one type. Polymorphism can be sub-divided into different categories depending on how it is applied. The type of polymorphism that is treated here is *parametric polymorphism*.

Cardelli and Wegner [7] refines the term parametric polymorphism and restricts it to mean a polymorphic function that has “an implicit or explicit type parameter, which determines the type of the argument for each application of that function.” Today, parametric polymorphism is called generic programming. Generic programming is thus the adherents use of this technique. Hence a function is called *generic* if it, independently of the type of an argument, performs generally the same kind of work. Some of the important advantages of this is the ability to reuse code and still maintain strong type-checking.

1.2 *CPH STL*

This section shortly describes the mission of the CPH STL project. The CPH STL [8] project is an initiative of the Performance Engineering Laboratory at DIKU. Its purpose is to “study and analyse existing specifications” of STL implementations (e.g. HP and SGI). Although many components of these STL implementations displays good performance there is still room for improvement on both an algorithmic and implementation level. The CPH STL tries to “determine the best approaches to optimization” on the existing specifications, “design alternative/enhanced versions of individual STL components using standard algorithmic and performance engineering techniques, and to implement and document the new versions in C++.”

1.3 *String*

“A string is a sequence of characters.”
— Bjarne Stroustrup [16, §20.1]

The goal of the CSS class is for it to conform to the C++ standard library. However, it cannot be said to be a sequence of characters, but a sequence of symbols—not excluding characters. While being C++ library compliant, the aim is to be able to represent a sequence of symbols from an arbitrary alphabet—of any type—and still keep the same functionality found in the `<string>` standard header file without any severe reduction in performance.

1.3.1 A perspective on the string type

Looking at the string type from an abstract point of view one gets the impression that it is a structure that constitutes a series of consecutive symbols of an alphabet that are in some cases ordered according to a grammar connected to a language. The fact is that strings often accommodate symbols that comprise of groups of symbols or *words* and *sentences*, all according to the current language and grammar.

Humans, which are frequent users of a multitude of distinct languages (natural, formal etc.) often wish to apply a distinct set of algorithms upon strings of symbols—string functions.

The arrangement of symbols in a language into groupings of *words* etc. carry data and data, when interpreted or treated by string functions, gives rise to information. One might argue that such a set of string functions logically belonged together.

The set of string functions in the C programming language is placed into a number of header files. In C there are no strings. Strings are achieved by using common arrays, containing atomic characters. In this way they are in a sense neutral, since an array is not a dedicated type or structure reserved for strings of characters alone. The string functions in C require that the symbols of a string are of type `char`. Yet, there are exceptions; for example, `strncmp` and `memcmp` provide the same functionality. The latter function compares a specified number of bytes of unknown type (`void*`) whereas the former compares a specified number of characters (`char`).

The C++ programming language contains a string class (the standard library string) where most of the string functions are internal members of the class. The class data structure is also an internal member of the class and it is—like in C—an array. Why create a dedicated type (i.e. a string class) that accommodates all this? Is this not possible to attain an order of symbols in any of the existing structures such as a vector or a list? Indeed so. In C++ there is a dissimilarity between arrays, vectors, and lists. To classify as a container class, a data structure needs to satisfy a number of requirements e.g. be able to contain information to whereas the container is empty etc. This is something that both vectors and lists do, but arrays do not. The STL includes, apart from container classes, algorithms that can be used in connection with container classes. The algorithms are all generic and allow for any container class to utilize, as they are decoupled from the container classes.

The SLS class that contain both string functions and a data structure causes it to be a semi-container. At one end making container like information accessible and at the other enclosing string functions. Having non-generic string functions enclosed inside the class and generic counterparts in the STL does render some of the algorithms built-in more or less redundant.

1.3.2 The `std::string` — an overview

This section contains the issues that has come up while studying the standard library string and are concisely considered below.

In addition to the support of its own type (the C++ string), the string class also supports C-type strings which make each function family in the class explicitly accept C-strings as function arguments. Keeping a class implementation lean and simple it would of course be preferable to accept only one type of string; keeping the class size to a minimum and function families small, making usage somewhat simpler and any maintenance, it would seem, more manageable.

The string class offers member functions that are redundant. Functions located in the STL `<algorithm>` header file yield the same functionality, while at the same time being generic and can thus substitute these “older”, type specific member functions [16, §20.3.2]. Reducing class size through code reuse via STL functions i.e. bringing down the number of member functions of the class, simplifying usage and maintenance should become more tractable.

The SLS constitutes a set of basic and fundamental functions. Any potential for specialising a structure such as the standard string is limited; such as optimization for small strings or specifying the exact storage implementation for symbols [16, §20.3]. This is due to the fact that the SLS is a *concrete type* (the *exact* class of which an object is an instance) and contains no virtual functions which can be overridden by a derived class, and it is thus only usable as a member of a more sophisticated text manipulation class in case of a wish for extending the set of string functions.

1.3.3 One size does not fit all

There are plenty of different string implementations to be found [4, 9, 14, 15]. The reason for the great diversity among string implementations seems to be the need for special features, optimized functions i.e. speed/memory utilization, cache-obliviousness/awareness, thread safety, compatibility towards another library, specialization towards a specific platform or architecture and portability.

Each of these implementations solve problems specific to their own problem space. Some of the features, extensions, functionalities that these and others make available are often mutually exclusive, resulting in programmers reinventing the wheel. Acknowledging a need for such diverse requirements, a library design could use some degree of modularity that allowed easy specialization of the class; taking into account that some areas of functionality simply are incompatible. Such an overwhelming design goal is out of the scope of this project.

1.4 Templates

Templates are a means of providing classes and functions the ability to be written without consideration of the data type with which it will eventually be used. Something that will be a key issue in the design of the CSS. In C++ there are two categories of template type parameters, *function templates* and *class templates*.

A function template provide a mechanism by which the semantics of a function definition can be preserved, independent of the type of the argument for the function. The function thus behaves in the same way independent of the type of the data treated. Hence a single function specifies a family of functions; each family member representing a specific type.

Class templates are similar to function templates, in that they allow for any type to be passed to the class. Class templates can affect the type for member functions and static data members. In the STL class templates are used to implement *container classes*, classes of objects that are intended to contain other objects i.e. list, vector, set etc. are all container classes.

1.4.1 Traits & policy classes

Two types of template parameters have in the literature received their own names. However the literature and community is not unified as to what the precise definitions of the terms *traits class* and *policy class* are [17, §15.1.5].

A policy is, according to the dictionary, “a course or principle of action adopted or proposed”. A more concrete explanation of the term policy class states:

“A policy class is a template parameter used to transmit behaviour. An example from the standard library is `std::allocator`, which supplies memory management behaviours to standard containers.” [5]

About the term traits class the same source states:

“A traits class provides a way of associating information with a compile-time entity (a type, integral constant, or address). For example, the class template `std::iterator_traits<T>` looks something like this:

```
template <class Iterator>
struct iterator_traits {
    typedef ... iterator_category;
    typedef ... value_type;
    typedef ... difference_type;
    typedef ... pointer;
    typedef ... reference;
};” [5]
```

Concerning the topic of trait and policy classes and the specifics of the CPH STL will be treated in §2.1.

1.5 Containers & iterators

A *container* is a representation of a set of objects. The representation contain, manage, and provide access to these objects; the elements of the container. Access to the elements inside the container is provided through *iterators*. An iterator is a handle to an element inside of a container. The iterator gives the ability to iterate through a range of elements. A pair of iterators can thus represent a selected range of elements within the container. All STL containers provide iterators and in this way let algorithms use these to access elements in the container instead of working directly with the container. Hence providing an interface between the algorithm and the container. Various kinds of iterators exists and each container have a type that is suitable to its data structure. A list, for example, provides *bidirectional* iterators meaning that the iterator can be both incremented and decremented, providing access to elements located both before and after the current element [3, §7.5]. The SLS will provide *random-access* iterators which means that it can move bidirectionally in arbitrary-sized steps hence giving access to any symbol without having to iterate through others, and giving (if at all possible) constant-time access to symbols.

1.5.1 Iterator guarantees

The C++ standard states that iterators should guarantee upper time limits on iterator operations. The random-access iterator provided by the SLS guarantees constant amortized time for all operations [3, §7.6]. The CPH STL requires that all iterator operations take $O(1)$ worst-case time [11]. Furthermore, it also requires that all container classes guarantees *iterator validity* for its iterators, contrary to the C++ standard [11]. Iterator validity is defined as:

“[...] A data structure is said to provide **iterator validity** if the iterators to its elements are kept valid at all times independent of the element moves.” [11]

2. Design

This section goes through the general layout of the design of the CSS class and accompanies the various decisions with a brief comment.

2.1 The `cphstl::basic_string` — an overview

The central idea of the design behind the CSS class is to split the class up in two—a core an a utility class. The inspiration for this comes from the article [1]. The result of employing the techniques presented therein provides a string class that can be changed according the the demands of the task at hand.

Splitting the string up into two classes has the advantage that the data structure the string is utilizing, to represent how the symbols are stored,

can be exchanged without the need to rewrite or modify any code. The basic idea of the *core class* is that this class implements a few, functionally independent, functions that the *utility class* can then use; combining the functions to produce the remaining functions. Resulting in the utility class remaining unaffected by the choice of any storage policy class, while the core class may be changed.

The CSS will have three template parameters which are all policy classes. The first template parameter, which is a mix between a policy and a traits class, indicates the symbols comprising the string. The second template parameter is the C++ standard library memory allocator (`std::allocator`) that is required for standard container types. The third template parameter gives the storage policy used for storing the symbols.

A difference between the CSS and the SLS is the fact that the CSS does not require a specific traits class [16, §20.2.1]. It is the responsibility of the symbol class to implement any comparisons operators etc. that should be available. Thus, if the functionality required for the symbol excludes comparison operators then there is no need to implement them in the symbol class.

The C++ SLS is an “almost container”. What it lacks to fully qualify as a standard container is the ability to accommodate a wide selection of types as elements [16, §17.1]. This is one of the things that the CSS aims at obliging.

The CSS though complying with the functionality of the SLS, will not supply all functions or all members of each function family that one can find in the standard library. The endorsement of C-strings is all together removed from the class. This is to streamline the implementation and keeping the functions families as small as possible. However, to still be of any use as a string in a C/C++ environment, functions are provided that convert C/C++ strings to CSS strings and the other way around.

Use of STL functions will be favoured in preference to implementing functionality that otherwise would be redundant in the class.

The data structure of the core class in this report will be based on a balanced search tree; a red-black tree. The idea of using a balanced search tree as a data structure for internal representation of the symbols in a string is not new [13, 4].

The red-black tree implementation used for the core class is a simplified version of the `rb_tree` class originating from [6].

Based on this, a number of design decisions has been made that allow flexibility and variety in use and hopefully will make the CSS a “fitter” string not burdened by superfluous functions and large function families and still able to handle C/C++-type strings and thus be of use in an existing code-base.

2.2 Design difficulties

The article of Hans-J. Boehm and Russ Atkinson, and Michael Plass [4] started out as a great inspiration for the project. The desired characteristics of rope strings were all agreeable and the distinguishing features promised were just what the CPH STL was looking for; immutability, commonly occurring operations should be efficient, operations should scale as to string length and operation performance and functions on strings should be reusable.

However, it turned out that the design of ropes crossed some of the ideas of the CSS and made others impossible or inefficient:

- The tree representing a rope is rebalanced (rebalancing is implicitly invoked) only if its height exceeds the bound (word size of the machine).
- A balanced rope can have unbalanced sub-ropes.
- The result of a concatenation must be able to share much of the data structure with its argument (see §2.6.1).

2.3 The core class and its functions

Since the choice of data structure for the string in this report is a red-black tree, two private functions will constitute some of the public utility functions; these are join and split [13, §6.2.3]. The joining of two trees would form the body of the concatenate algorithm. The C++ standard does not specify the complexity of SLS operations. However, according to [13, §6.2.3] the cost of the split algorithm is $\mathcal{O}(\lg n)$ units of time and $\mathcal{O}(k)$ steps for join, where n denotes the number of symbols stored in the tree and k is the difference in heights between the trees being concatenated.

“Of the operations that modify the value of a string, one of the most common is appending to it” [16, §20.3.9]

The picking of core functions is by no means finished as the implementation of the string class is unfinished and therefore lacks thorough examination of varying possibilities.

2.4 The utility class and its functions

Append, concatenate, erase, insert and replace: all of which it is possible to implement via the two private core functions join and split.

Concatenate, as has been seen, is comprised by an invocation of the join algorithm. Running time: $\mathcal{O}(k)$.

Append is similar to concatenate and would also use join. Running time: $\mathcal{O}(k)$.

Insert split at the symbol where the new string is inserted and join at both ends of the new. Running time: $\mathcal{O}(\lg n + 2k)$.

Erase split at both ends of the substring that is erased and join the two remaining strings. Running time: $\mathcal{O}(2\lg n + k)$.

Replace can be achieved by erasing the substring that is going to be replaced followed by joining the new string at each end. Running time: $\mathcal{O}(2\lg n + 2k)$.

The functions for converting between CSS, C, and SLS strings are part of the utility class.

2.5 Non-implemented standard string functionality

In particular the different find functions of the `<string>` standard header can be achieved through the use of the generic functions declared in the STL. Due to the generic nature of these functions they can also be used for CSS strings.

In this section and its subsections the omitted functions and operators from the CSS implementation are listed.

Some of the functions have been selected not to be implemented due to the possibility to accomplish the same functionality through the use of generic functions in the STL, others due to the requirements of the CPH STL, and yet others because they were unwanted as a part of the CSS. Whenever suitable, the function or operator in question is either accompanied with a proposal on how to accomplish the same functionality through the use of generic functions in the STL. Furthermore, a short comment on why the function or operator was left out of the implementation will accompany it.

As noted in §1.3.2 there were some issues about the SLS that were found when studying it. Sections §2.5.1, §2.5.3 and §2.5.2 will suggest what can be done in the CSS to solve these issues.

2.5.1 Backwards compatibility with C

All backwards compatibility with C has been eliminated. Keeping the backwards compatibility would have made the number of functions in each function family supported large. This make it a huge task just to provide all of the different family members.

Not accepting C-type strings also has the advantage of not having to convert to and from each type internally.

There should be no problem in converting a CSS to a C-string with the `'\0'`-termination of the character array. The symbol type of the CSS is without any effect on the array in that the array is filled with the symbols of the CSS and is then terminated with the `'\0'` symbol.

2.5.2 Possibility for specialization

There is really no limit to specialization in that a core class is open to the user and both data structure and any core functions are customizable.

2.5.3 Alphabet independence

Since the functionality of finding substrings in the generic string is moved outside the CSS relying totally upon the functions already present in the C++ STL `<algorithm>` standard header, there is no need for applying a traits class as is the custom for the SLS, for specifying how to compare two symbols. These operations should instead be present in the symbol class, making the CSS string smaller and the responsibility laid out to the user of the class.

2.5.4 Find function family

“There is a bewildering variety of functions for finding substrings”
 – Bjarne Stroustrup [16, §20.3.11]

The `<string>` standard header file has a number of functions for finding substrings. The advent of the STL makes these functions superfluous since it provides functions with that same functionality. A good guess is that, for backwards compatibility reasons, these functions cannot just be removed from the `<string>` header file. However, the CSS has no obligations and is therefore not bound by this. Operators as *equal* are specified in the symbol class provided by the user. The result is a smaller CSS.

Actually [16, §20.3.2] states that the operations provided by the `<string>` header file will—hopefully—be further optimized than is “easy to do for general algorithms”.

The C++ standard library functions for finding substrings all use a common reoccurring pattern in the function families. There is the version that takes a C++ string (`const basic_string& s`) as its argument, two that take a C-style strings (`const char* c`) and finally one that takes a single character (`char c`). All the find functions return an index (`size_type`). The types of the parameters taken by the different members in the families look like this:

1. `const string&, size_type`
2. `const char*, size_type`
3. `const char*, size_type, size_type`
4. `char, size_type`

Table 1.

One could roughly divide a user scenario into two cases for the use of the find functions which can be used for finding a substring in a given string:

- i. A user has an instance of a CSS upon which a find function is to be applied.
- ii. A user has a C or a C++ string that (s)he later wants the functionality of the CSS.

In the former case the generic STL find substitute described in this section can be used accordingly. The latter case first requires the user to convert the C/C++ string to a CSS. This is achieved by calling any of the conversion functions. When describing the family members of the various find function families that take pointers to C-style strings (cf. case ii above) it is presumed that they are first converted to a CSS.

When referring to the different family member types the following pattern is used:

function_family_name **re.** [1–4]

where “re.” is latin and means “in reference to”, 1–4 refer to the items listed in Table 1.

- find re. 1** Using the appropriate conversion function turning a C++ string into a CSS. The `algorithm::search` function takes two pairs of iterators one being the CSS stored in range `[first1, last1)`. Along with this the other set is for that of the pattern being searched `[first2, last2)`. The `first2`-iterator is incremented to specify the symbol (`index`) of which the pattern is to start from. Thus searching the pattern in the interval `[first2+n, last2)` with that of the CSS `[first1, last1)`.
- find re. 2** The only difference from that of the above is that the argument `char* str` will need to use a different conversion function.
- find re. 3** Similar to **find re. 2**, but by decrementing the `last2`-iterator specifying the number of symbols (`length`) in the range `[first2, last2)` that should make out the pattern that is searched in the interval `[first1, last1)`.
- find re. 4** The `first`-iterator, and a `last`-iterator to indicate the interval of the CSS in which to find the symbol (`ch`). Incrementing the `first`-iterator determines at which symbol (`index`) of the CSS to searching.

To avoid repetition of similarities with the above, the following function families are treated together.

find_first_of re. 1–4 This function is very similar to the `string::find` function. Actually, it is the “same” only when `index = 0`. To accomplish an alternative using the `<algorithm>` standard header file the same functions are used as in the case of **find re. 1**.

The other function family members are handled just as their `find` equivalents.

find_first_not_of re. 1–4 The generic STL function `find_if` could replace this function using the generic predicate `not_equal_to<T>`.

The other function family members of the `find_first_not_of` can be obtained in a similar manner as family members.

find_last_of re. 1–4 Using the generic STL function `find_end` accomplishes the same as this function.

The function family members are handled as previously.

`find_last_not_of` **re. 1–4** The `find_end` function used for `find_last_of` can also be used here. It has a second, overloaded version, that takes a binary predicate that can be used; `not_equal_to<T>`. As before, function family members are handled in the same manner as previously.

`rfind` **re. 1–4** A generic substitute for this function would be to use the `search` function used for `find` **re. 1**, giving it reverse iterators instead of forward iterators. And yet again, as before, function family members are handled in the same manner as previously.

2.5.5 Replace function family

The effect of the C++ STL `algorithm::replace` function is not similar to that of `string::replace`. The STL version replaces all occurrences of a specified symbol with a new symbol in an interval specified by iterators. Whereas the `string::replace` replaces an interval or symbol with one or more new symbols. The `replace` function, which is one of the core functions in the CSS, provides the same functionality as `string::replace` does.

2.5.6 Swap function family

Just like the SLS find function family are substituted with those found in the STL (§2.5.4), the SLS swap family can be replaced by the STL `algorithm::swap_ranges` (ASR). However, both the SLS swap and the `std::swap` are guaranteed an $O(1)$ time constraint whereas the complexity of ASR is linear $O(n)$ [10].

2.5.7 Operator[]

The CSS class is a random access container, though it lacks the implementation of the `operator[]`. It makes the container cleaner to exclusively use iterators for accessing symbols. Furthermore it lacks bounds checking in the SLS i.e. subscript accesses can result in segmentation faults. The `at()` function, which does the same thing, uses bounds checking. The influence on performance should be minimal.

2.5.8 Operator= and operator()

Assigning a value to a CSS through the `operator()` has some similarity with that of manipulating the string, in that any iterators to the string becomes invalid. To solve this the `operator=`, which supposedly does the same as `operator()`, instead changes the *owner* to the object giving it a new handle, thus preserving the validity of any iterators.

2.5.9 copy(), c_str(), and data()

The three functions `copy()`, `c_str()` and `data()` are replaced with functions to convert between C/C++ and CSS strings. The reason is that none of the

three functions meet the immutability requirement of the CSS, while the conversion function makes a copy of each of the symbols in the CSS, putting them into an array which is then returned.

2.5.10 Stream operator<< and operator>>

The design of the CSS requires the creator of the symbols comprising the string to supply operators for comparison. Likewise should the input/output stream operators be supplied by the user of the class. As a symbol is a generic entity, accurate reading and writing of such user defined symbol types cannot be incorporated into the CSS class.

2.6 Mutable, immutable, reference counting and COW

In this section we describe an optimization techniques that will be part of the CSS design.

In programming languages such as Java and C#, the default string types are *immutable*. Immutable objects possess the attribute of not being susceptible to change or variation once instantiated. The C++ standard library string is mutable and can thus be changed at any time.

Immutable objects have favourable characteristics such as:

- being automatically thread-safe and have no synchronization issues
- the copy constructor need only to return a reference to the object, attaining constant instead of linear time complexity as required by the standard
- does not need to be copied defensively when used as class members
- the class invariant is established once upon construction, and never needs to be checked again

An optimization strategy, called *copy-on-write* (COW), allows multiple users of an object to share the same instance by handing, each subsequent user asking for the object, a reference to the initial instance of the object. Creating a semblance that each holder of a reference possesses his/her very own copy of the object. A technique, called *reference counting*, counts the handed out references with the intention of deallocating the object as soon as the counter reaches zero; hereby reclaiming resources immediately. Supplementing immutability with COW, the requiring for additional locking, when objects in different threads need to safely share the same representation, is hence unnecessary, making the combination of the two, a good choice for the CSS.

2.6.1 The effects of the optimizations

How should the CSS handle string function calls such as concatenation and append. The principle of immutability dictates that the string cannot be altered and should return a reference to the string and COW prescribe that

the string should return a copy whenever written to. Assume the following example of the use of `append`:

```
a = ('Hello ');
b = ('World!');
a += b;
```

The “Hello” string representation can be seen in Figure 1. The tree structure for the “World!” string would be similar.

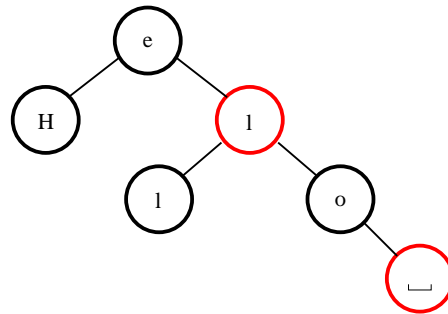


Figure 1. The red-black tree containing the string “Hello ”

A solution to the `append` operation could be to just add the iterator returned from `b.begin()` of the appended string to the tree, as shown in Figure 2. There are some problems with this solution:

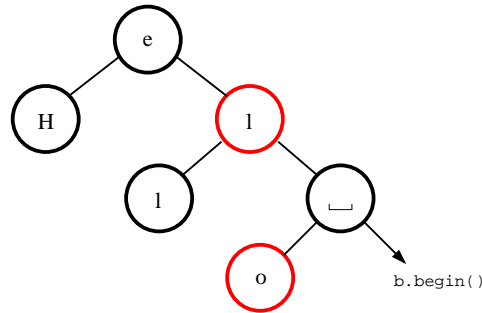


Figure 2. Appending of immutable COW string

- i. Had the iterator to the appended string been a single symbol instead, had its node colour been red after the insertion. Recall that the tree structure for the “World!” string was identical to that of Figure 1. This means that the iterator points to a root node of a red-black tree where the node colour is black. Recolouring the nodes in the appended tree is not an option since it is an immutable object.
- ii. Suppose that the appended string consisted of a greater number of symbols than the string it was appended to. Using a balanced binary search

tree as a data structure this would immediately cause the tree to become unbalanced. Searching such a tree could by no means guarantee $\mathcal{O}(\lg n)$ time. Rebalancing the tree would be quite complex and is not an option since the string is an immutable object.

All of mentioned issues are the result of the data structure being a red-black tree. If the data structure was a vector neither **i** or **ii** above would constitute a problem. Returning a substring of a binary tree is, in the best case, a question of returning the iterator of a root node of a subtree. Worst case; returning both the begin and end iterators of the substring. The same is the situation for a vector. Although the vector data structure would not suffer in the same sense as the red-black tree, using iterators when appending strings, there is still a risk of cluttering the string. Working with a string for an extended period of time could issuing a vast number of append and concatenate operations and could result in a “descendant”-string that, for a major part, is comprised of iterators; perhaps even in multiple, nested levels. While resulting in great utilization of resources it might have a negative impact when applying functions such as string compare, retrieving substrings, pattern matching and the likes. To be able to support many types of data structures, using iterators to replace strings of symbols cannot be allowed. Instead a copy of the original string, or a part of it, have to be made and used in the new string. Keeping in mind that the CSS uses reference counting this decision should not have any catastrophic influences in terms of never deallocating resources.

Intensive use of string functions that modify the string e.g. append, replace, erase etc. includes copying the original string or parts of it. In situations like these it is wiser to have a mutable CSS variant. The immutable aspect of the CSS is a key characteristic of the design and incompatible with a mutable version. It would be impractical to pass a characteristic such as (im)mutability as a policy. All functions that made any kind of modification to the string would have to be placed into the policy, requiring the user to supply extensive amount of code just to be able to use the class. Hence a mutable version should be designed as a wholly separate class.

2.6.2 Compatibility

Suppose both a mutable and an immutable variant of the CSS were implemented. Would an assignment from a mutable to an immutable or the other way around be a legal operation? Allowing such conversations requires conversations between policies i.e. something that has to be implemented into each one of the classes. Such conversations should only occur when each one of the strings have a reference counter equal to 1 [2]. This is something that the CSS class will not support at this time.

3. Discussion

A great deal of the report has been design issues of the CSS. Which functions to include and why some other are avoided or skipped.

The immutable/mutable question has been a though one, since the SLS class and its functions are mutable, it has been difficult to comply with the iterator validity of the CPH STL not to mention the fact that it should at the same time be immutable.

Using the CSS as a basis it would be possible to make a number of specialized string classes, thus solving problems in specific domains. One such domain could be word processing. However dealing with editing issues, such as inserting larger text blocks and moving them around, would be interesting to observe if one chooses to employ a red-black tree as a data structure. And would the choice land on a mutable or immutable string?

Another specialization that might be of interest to implement is regular expressions.

3.1 Conclusion

Concerning analysing of what functions to include in the core class and what functions, by that choice, to be able to create in the utility class has regrettably not been completed. The structure of the core and utility classes has been laid. The developed source code can be found in AppendixA and AppendixB.

Acknowledgements

I would like to thank my supervisor Jyrki Katajainen for being a great source of inspiration.

References

- [1] A. Alexandrescu, Generic <Programming>: A policy-based basic_string implementation, *C/C++ Users Journal*, June (2001).
- [2] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley Professional (2001).
- [3] M. H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*, Addison-Wesley (1999).
- [4] H. J. Boehm, R. Atkinson, and M. Plass, Ropes: an alternative to strings, *Software—Practice and Experience* **25**,12 (1995), 1315–1330.
- [5] Boost, Generic programming techniques, Website accessible at http://www.boost.org/more/generic_programming.html (2004-2006).
- [6] H. Brönnimann and J. Katajainen, *Efficiency of various forms of red-black trees*, CPH STL Report 2006-2 (2006).
- [7] L. Cardelli and P. Wegner, On understanding types, data abstraction, and polymorphism, *Computing Surveys* **17**,4 (1985), 471–522.
- [8] Performance Engineering Laboratory, The CPH STL, Website accessible at www.cphstl.dk (2000–2006).

- [9] P. Hsieh, The better string library, Worldwide Web Document (2006). Available at <http://bstring.sourceforge.net/>.
- [10] N.M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley (1999).
- [11] J. Katajainen, *The cost of iterator validity*, Lecture slides from www.cphstl.dk 12.05.2004 (2004).
- [12] J. Katajainen, Research proposal: Generic programming—algorithms and tools, Technical Report CPH STL Report 2005-5, Department of Computing, University of Copenhagen (2005).
- [13] D.E. Knuth, *The Art of Computer Programming, Searching and Sorting*, Third Edition, Addison-Wesley (1997).
- [14] Trolltech, Qstring class reference, Website accessible at <http://doc.trolltech.com/4.1/qstring.html> (2006).
- [15] Silicon Graphics, Inc., rope<T, Alloc>, Website accessible at www.sgi.com (2003).
- [16] B. Stroustrup, *The C++ Programming Language, Special Edition*, Addison-Wesley (2000).
- [17] D. Vandevorde and N.M. Josuttis, *C++ Templates, The Complete Guide*, Addison-Wesley (2003).

Appendix: Source code listings

A. CPH STL Basic String	17
A.1 basic_string.hpp	17
A.2 basic_string.cpp	20
B. Red-black tree string	27
B.1 rb_tree_string.hpp	27
B.2 rb_tree_string.cpp	28

AppendixA. CPH STL Basic String

AppendixA.1 basic_string.hpp

```

/*
 *
 * Copyright (c) 2006
 * Filip Bruman, University of Copenhagen, Denmark
5  *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear in
10 * supporting documentation. Neither the author nor University of
 * Copenhagen, Polytechnic University, Silicon Graphics, or
 * Hewlett Packard, makes any representations about the suitability of
 * this software for any purpose. It is provided "as is" without
 * express or implied warranty.
15 *
 */

#include "rb_tree_string.hpp"

20 #ifndef CPHSTL_BASIC_STRING_HPP

```

```

#define CPHSTL_BASIC_STRING_HPP

    // from set_adapter.hpp
    namespace detail {
25
        template <class T>
        struct Identity {
            T operator()(T const& t) const { return t; }
            T& operator()(T& t) const { return t; }
30
        };

        } // namespace detail

    namespace cphstl
35 {

        template <
            typename Symbol,
            typename Alloc = std::allocator<Symbol>,
40
            /* Key, Value, KeyOfValue, Compare, Alloc */
            typename Storage = cphstl::rb_tree_string
            <Symbol,
            Symbol/*void*/,
            detail::Identity<Symbol>,
45
            std::less<Symbol>,
            Alloc> >
        class basic_string
        {
            // types (required)
50
        public:
            typedef Alloc allocator_type;
            typedef Symbol value_type;
            typedef value_type& reference;
            typedef const value_type& const_reference;
55
            typedef typename Alloc::difference_type difference_type;
            typedef typename Alloc::size_type size_type;
            typedef typename Alloc::pointer pointer;
            typedef typename Alloc::const_pointer const_pointer;

60
            typedef typename Storage::iterator iterator;
            typedef typename
            Storage::const_iterator const_iterator;
            typedef typename
            Storage::reverse_iterator reverse_iterator;
65
            typedef typename
            Storage::const_reverse_iterator const_reverse_iterator;

            // types (other)
            typedef typename Storage::value_type storage_value_type;
70

        protected:
            Storage storage;
            static const size_type npos = -1;

75
            // 21.3.1 construct/copy/destroy
        public:

```

```

    basic_string();
    basic_string(const basic_string& str);
    ~basic_string();
80    basic_string& operator=(const basic_string& str);

    // 21.3.2 iterators
public:
    iterator      begin();
85    const_iterator begin() const;
    iterator      end();
    const_iterator end()   const;
    reverse_iterator      rbegin();
    const_reverse_iterator rbegin() const;
90    reverse_iterator      rend();
    const_reverse_iterator rend()   const;

    // 21.3.3 capacity
public:
95    bool empty() const;
    size_type size() const;
    size_type max_size() const;
    void clear();

100    // 21.3.4 element access
public:
    reference at(size_type n);
    const_reference at(size_type n) const;

105    // 21.3.5 modifiers
public:
    basic_string& operator+=(const basic_string& str);
    basic_string& append(const basic_string& str);
    basic_string& append(iterator begin, iterator end);
110    iterator insert(iterator p, const storage_value_type& t);
    iterator insert(const storage_value_type& t); // set insert
    void insert(iterator p, int n, iterator t);
    void erase(iterator begin);
    void erase(iterator begin, iterator end);
115    basic_string& replace(iterator s1, iterator s2
                           , iterator t1, iterator t2);
    void swap(basic_string& str);

    // 21.3.6 string operations
120 public:
    allocator_type get_allocator() const;
    const char* cph_str2c_str(const basic_string& str);
    basic_string& c_str2cph_str(const char& c_str);
    std::string& cph_str2std_str(const basic_string& str);
125    basic_string& std_str2cph_str(const std::string& str);
    basic_string& substr(size_type pos = 0, size_type n = npos) const;

    // 21.3.7.2-7 non-member functions
public:
130    bool operator==(const basic_string& str);
    bool operator!=(const basic_string& str);
    bool operator< (const basic_string& str);

```

```

    bool operator> (const basic_string& str);
    bool operator<=(const basic_string& str);
135  bool operator>=(const basic_string& str);

    // 21.3.7.9 inserters and extractors
    public:
        istream& operator>>(basic_string& str);
140  ostream& operator<<(const basic_string& str);

    }; // class
} // namespace cphstl

145 #include "basic_string.cpp"

#ifdef CPHSTL_BASIC_STRING_HPP

Appendix A.2 basic_string.cpp

/*
 *
 * Copyright (c) 2006
 * Filip Bruman, University of Copenhagen, Denmark
5  *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear in
10 * supporting documentation. Neither the author nor University of
 * Copenhagen, Polytechnic University, Silicon Graphics, or
 * Hewlett Packard, makes any representations about the suitability of
 * this software for any purpose. It is provided "as is" without
 * express or implied warranty.
15 *
 */

#include <functional> // std::pair
#include "rb_tree_string.hpp"
20 #ifndef CPHSTL_RB_TREE_STRING_CPP
#define CPHSTL_RB_TREE_STRING_CPP

namespace cphstl
25 {
    //
    // 21.3.1 construct/copy/destroy
    //
    template <typename Symbol, typename Alloc, typename Storage>
30  basic_string<Symbol, Alloc, Storage>::basic_string()
    {
        // not implemented
    }

35  template <typename Symbol, typename Alloc, typename Storage>
    basic_string<Symbol, Alloc, Storage>::~~basic_string()
    {
        // not implemented
    }
}

```

```

}
40
template <typename Symbol,typename Alloc,typename Storage>
basic_string<Symbol,Alloc,Storage>
    ::basic_string(const basic_string& str)
{
45     (*this).storage(str);
}

template <typename Symbol,typename Alloc,typename Storage>
basic_string<Symbol,Alloc,Storage>&
50 basic_string<Symbol,Alloc,Storage>
    ::operator=(const basic_string& str)
{
    return (*this).storage.operator=(str);
}

55
//
// 21.3.2 iterators
//
template <typename Symbol,typename Alloc,typename Storage>
60 typename basic_string<Symbol,Alloc,Storage>::iterator
basic_string<Symbol,Alloc,Storage>::begin()
{
    return (*this).storage.begin();
}

65
template <typename Symbol,typename Alloc,typename Storage>
typename basic_string<Symbol,Alloc,Storage>::const_iterator
basic_string<Symbol,Alloc,Storage>::begin() const
{
70     return (*this).storage.begin();
}

template <typename Symbol,typename Alloc,typename Storage>
typename basic_string<Symbol,Alloc,Storage>::iterator
75 basic_string<Symbol,Alloc,Storage>::end()
{
    return (*this).storage.end();
}

80
template <typename Symbol,typename Alloc,typename Storage>
typename basic_string<Symbol,Alloc,Storage>::const_iterator
basic_string<Symbol,Alloc,Storage>::end() const
{
85     return (*this).storage.end();
}

template <typename Symbol,typename Alloc,typename Storage>
typename basic_string<Symbol,Alloc,Storage>::reverse_iterator
basic_string<Symbol,Alloc,Storage>::rbegin()
90 {
    return (*this).storage.rbegin();
}

template <typename Symbol,typename Alloc,typename Storage>

```

```

95  typename basic_string<Symbol, Alloc, Storage>::const_reverse_iterator
    basic_string<Symbol, Alloc, Storage>::rbegin() const
    {
        return (*this).storage.rbegin();
    }
100
    template <typename Symbol, typename Alloc, typename Storage>
    typename basic_string<Symbol, Alloc, Storage>::reverse_iterator
    basic_string<Symbol, Alloc, Storage>::rend()
    {
105     return (*this).storage.rend();
    }

    template <typename Symbol, typename Alloc, typename Storage>
    typename basic_string<Symbol, Alloc, Storage>::const_reverse_iterator
110  basic_string<Symbol, Alloc, Storage>::rend() const
    {
        return (*this).storage.rend();
    }

115  //
    // 21.3.3 capacity
    //
    template <typename Symbol, typename Alloc, typename Storage>
    bool
120  basic_string<Symbol, Alloc, Storage>::empty() const
    {
        return (*this).storage.empty();
    }

125  template <typename Symbol, typename Alloc, typename Storage>
    typename basic_string<Symbol, Alloc, Storage>::size_type
    basic_string<Symbol, Alloc, Storage>::size() const
    {
130     return (*this).storage.size();
    }

    template <typename Symbol, typename Alloc, typename Storage>
    typename basic_string<Symbol, Alloc, Storage>::size_type
    basic_string<Symbol, Alloc, Storage>::max_size() const
135  {
        return (*this).storage.max_size();
    }

    template <typename Symbol, typename Alloc, typename Storage>
140  void basic_string<Symbol, Alloc, Storage>::clear()
    {
        (*this).storage.clear();
    }

145  //
    // 21.3.4 element access
    //
    template <typename Symbol, typename Alloc, typename Storage>
    typename basic_string<Symbol, Alloc, Storage>::reference
150  basic_string<Symbol, Alloc, Storage>::at(size_type n)

```

```

{
  Symbol s;

  if( n > 0 && n < (*this).size() )
155   {
      // not implemented
    }

  return s;
160 }

template <typename Symbol,typename Alloc,typename Storage>
typename basic_string<Symbol, Alloc, Storage>::const_reference
basic_string<Symbol, Alloc, Storage>::at(size_type n) const
165 {
  Symbol s;

  if( n > 0 && n < (*this).size() )
170   {
      // not implemented
    }

  return s;
175 }

//
// 21.3.5 modifiers
//
template <typename Symbol,typename Alloc,typename Storage>
180 typename basic_string<Symbol, Alloc, Storage>::basic_string&
basic_string<Symbol, Alloc, Storage>
  ::operator+=(const basic_string& str)
{
  // not implemented
185   return str;
}

template <typename Symbol,typename Alloc,typename Storage>
typename basic_string<Symbol, Alloc, Storage>::basic_string&
190 basic_string<Symbol, Alloc, Storage>::append(const basic_string& str)
{
  // not implemented
  return (*this);
195 }

template <typename Symbol,typename Alloc,typename Storage>
typename basic_string<Symbol, Alloc, Storage>::basic_string&
basic_string<Symbol, Alloc, Storage>
  ::append(iterator begin, iterator end)
200 {
  // not implemented
  return (*this);
}

205 template <typename Symbol,typename Alloc,typename Storage>
void
```

```

basic_string<Symbol, Alloc, Storage>::erase(iterator begin)
{
  (*this).storage.erase(begin);
210 }

template <typename Symbol, typename Alloc, typename Storage>
void
basic_string<Symbol, Alloc, Storage>
215   ::erase(iterator begin, iterator end)
{
  // not implemented
}

220 template <typename Symbol, typename Alloc, typename Storage>
typename basic_string<Symbol, Alloc, Storage>::iterator
basic_string<Symbol, Alloc, Storage>
  ::insert(iterator p, const storage_value_type& t)
{
225   return (*this).storage.insert_equal(p, &t);
}

template <typename Symbol, typename Alloc, typename Storage>
void
230 basic_string<Symbol, Alloc, Storage>
  ::insert(iterator p, int n, iterator t)
{
  // not implemented
}
235

template <typename Symbol, typename Alloc, typename Storage>
typename basic_string<Symbol, Alloc, Storage>::basic_string&
basic_string<Symbol, Alloc, Storage>
  ::replace(iterator s1, iterator s2, iterator t1, iterator t2)
240 {
  // not implemented
  return (*this);
}

245 template <typename Symbol, typename Alloc, typename Storage>
void
basic_string<Symbol, Alloc, Storage>::swap(basic_string& str)
{
  // not implemented
250 }

//
// 21.3.6 string operations
//
255 template <typename Symbol, typename Alloc, typename Storage>
typename basic_string<Symbol, Alloc, Storage>::allocator_type
basic_string<Symbol, Alloc, Storage>::get_allocator() const
{
  return Alloc::get_allocator();
260 }

// make friend

```

```

template <typename Symbol,typename Alloc,typename Storage>
const char*
265 basic_string<Symbol,Alloc,Storage>
    ::cph_str2c_str(const basic_string& str)
    {
        // not implemented
        char *cp;
270     return cp;
    }

    // make friend
template <typename Symbol,typename Alloc,typename Storage>
275 typename basic_string<Symbol,Alloc,Storage>::basic_string&
basic_string<Symbol,Alloc,Storage>::c_str2cph_str(const char& c_str)
    {
        // not implemented
        return (*this);
280     }

    // make friend
template <typename Symbol,typename Alloc,typename Storage>
std::string&
285 basic_string<Symbol,Alloc,Storage>
    ::cph_str2std_str(const basic_string& str)
    {
        // not implemented
        std::string s("");
290     return &s;
    }

    // make friend
template <typename Symbol,typename Alloc,typename Storage>
295 typename basic_string<Symbol,Alloc,Storage>::basic_string&
basic_string<Symbol,Alloc,Storage>
    ::std_str2cph_str(const std::string& str)
    {
        // not implemented
300     return (*this);
    }

template <typename Symbol,typename Alloc,typename Storage>
typename basic_string<Symbol,Alloc,Storage>::basic_string&
305 basic_string<Symbol,Alloc,Storage>
    ::substr(size_type pos, size_type n) const
    {
        // not implemented
        if( pos == 0 && n > npos )
310     {
            while( pos != npos )
                {}
        }
        return (*this);
315     }

//
// 21.3.7.2-7 non-member functions

```

```

320 //
    template <typename Symbol,typename Alloc ,typename Storage>
    bool
    basic_string<Symbol, Alloc ,Storage>
        ::operator==(const basic_string& str)
    {
325     return Symbol::operator==(str);
    }

    template <typename Symbol,typename Alloc ,typename Storage>
    bool
330 basic_string<Symbol, Alloc ,Storage>
        ::operator!=(const basic_string& str)
    {
        return Symbol::operator!=(str);
    }
335

    template <typename Symbol,typename Alloc ,typename Storage>
    bool
    basic_string<Symbol, Alloc ,Storage>
        ::operator<(const basic_string& str)
340 {
        return Symbol::operator<(str);
    }

    template <typename Symbol,typename Alloc ,typename Storage>
345 bool
    basic_string<Symbol, Alloc ,Storage>
        ::operator>(const basic_string& str)
    {
        return Symbol::operator>(str);
350 }

    template <typename Symbol,typename Alloc ,typename Storage>
    bool
    basic_string<Symbol, Alloc ,Storage>
355     ::operator<=(const basic_string& str)
    {
        return Symbol::operator<=(str);
    }

360 template <typename Symbol,typename Alloc ,typename Storage>
    bool
    basic_string<Symbol, Alloc ,Storage>
        ::operator>=(const basic_string& str)
    {
365     return Symbol::operator>=(str);
    }

    //
    // 21.3.7.9 inserters and extractors
370 //
    template <typename Symbol,typename Alloc ,typename Storage>
    istream&
    basic_string<Symbol, Alloc ,Storage>
        ::operator>>(basic_string& str)

```

```

375  {
      return Symbol::operator>>(str);
    }

    template <typename Symbol,typename Alloc ,typename Storage>
380  ostream&
    basic_string<Symbol, Alloc ,Storage>
      ::operator<<(const basic_string& str)
    {
      return Symbol::operator<<(str);
385  }
    }

#endif // CPHSTL_RB_TREE_STRING_CPP

```

AppendixB. Red-black tree string

AppendixB.1 rb_tree_string.hpp

```

/*
 *
 * Copyright (c) 2006
 * Filip Bruman, University of Copenhagen, Denmark
5  *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear in
10 * supporting documentation. Neither the author nor University of
 * Copenhagen, Polytechnic University, Silicon Graphics, or
 * Hewlett Packard, makes any representations about the suitability of
 * this software for any purpose. It is provided "as is" without
 * express or implied warranty.
15 *
 */
#include <functional> // std::pair
#include "rb_tree.hpp"

20 #ifndef CPHSTL_RB_TREE_STRING_HPP
#define CPHSTL_RB_TREE_STRING_HPP

    namespace cphstl
    {
25
        template <
            class Key,
            class Value,
            class KeyOfValue,
30            class Compare,
            class Alloc
        >
        class rb_tree_string
        : public boost::tree::rb_tree <Key,
35            Value,
            KeyOfValue,

```

```

Compare,
Alloc>
{
40 // types
protected:
    typedef typename
        boost::tree::rb_tree<Key, Value, KeyOfValue, Compare, Alloc>
        ::const_base_ptr const_base_ptr;
45 public:
    typedef typename
        boost::tree::rb_tree<Key, Value, KeyOfValue, Compare, Alloc>
        ::value_type value_type;
    typedef typename
50 boost::tree::rb_tree<Key, Value, KeyOfValue, Compare, Alloc>
        ::iterator iterator;

    // internal modifiers
protected:
55 std::pair<rb_tree_string*, rb_tree_string*>
    split(const_base_ptr p) const;
    rb_tree_string& join(const rb_tree_string& t) const;

    // for test purposes
60 public:
    void print_rb_tree_string(const_base_ptr p);

    // 21.3.5 modifiers
    rb_tree_string& append(const rb_tree_string& str);
65 iterator insert(iterator p, const value_type& t);
    iterator insert(const value_type& t); // set insert
    void erase(iterator begin);
    rb_tree_string& replace(iterator s1, iterator s2
        , iterator t1, iterator t2);
70

    // 21.3.7.1 non-member function
    rb_tree_string& operator+(const rb_tree_string& t);

}; // class
75 } // namespace cphstl

#include "rb_tree.cpp"

#endif // CPHSTL_RB_TREE_STRING_HPP

Appendix B.2 rb_tree_string.cpp

/*
 *
 * Copyright (c) 2006
 * Filip Bruman, University of Copenhagen, Denmark
5 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear in
10 * supporting documentation. Neither the author nor University of

```

```

* Copenhagen, Polytechnic University, Silicon Graphics, or
* Hewlett Packard, makes any representations about the suitability of
* this software for any purpose. It is provided "as is" without
* express or implied warranty.
15 *
*/

#include <functional> // std::pair
#include "rb_tree.hpp"
20
#ifndef CPHSTL_RB_TREE_STRING_CPP
#define CPHSTL_RB_TREE_STRING_CPP

namespace cphstl
25 {
    template <class Key, class Value, class KeyOfValue
                , class Compare, class Alloc>
    inline typename rb_tree_string<Key, Value, KeyOfValue
                , Compare, Alloc >::rb_tree_string&
30 rb_tree_string<Key, Value, KeyOfValue, Compare, Alloc>
        ::join(const rb_tree& t) const
    {
        RB_TREE_USE_CONST_PROPERTY_MAPS

35 // to comply with immutability principle copy the two trees
        rb_tree l1 = rb_tree(*this);
        rb_tree l2 = rb_tree(t);

        size_type
40     l2_black_height = l2.m_black_count(l2.m_leftmost()
                , l2.m_root()),
        l1_black_height = l1.m_black_count(l1.m_rightmost()
                , l1.m_root());

45     if( l1_black_height >= l2_black_height )
        {
            const_base_ptr x = l1.m_root();
            base_ptr& p;

50     if( l1_black_height == l2_black_height )
            p = l1.m_leftmost();
            else
            {
                const_base_ptr r = l1.m_rightmost();
55     while( l1_black_height > l2_black_height )
                {
                    x = right[x];
                    l1_black_height = l1.m_black_count(r, x);
                }
60     p = x;
            }

            parent[p] = 0;
            // p's parent
65     base_ptr& pp = parent[p];
            right[pp] = 0;

```

```

// juncture node in l2
base_ptr& j = l2.m_leftmost();
70 // its parent
base_ptr& jp = parent[j];

left[jp] = 0;
parent[j] = 0;
75

// insert j at p's place and let
// p and subtree be the left subtree
// of j; set (the remaining) l2 as
80 // j's right subtree
right[pp] = j;
left[j] = p;
right[j] = l2;

85 return *l1;
}
else
return l2.join(*l1);

90 return l1;
}

// Splitting tree at node i
// returning two trees where i
95 // belongs to the left tree
template <class Key, class Value, class KeyOfValue
, class Compare, class Alloc>
std::pair<typename rb_tree_string<Key, Value, KeyOfValue
, Compare, Alloc>::rb_tree_string*,
100 typename rb_tree_string<Key, Value, KeyOfValue
, Compare, Alloc>::rb_tree_string*>
rb_tree_string<Key, Value, KeyOfValue, Compare, Alloc>
::split(const_base_ptr y) const
{
105 RB_TREE_USE_CONST_PROPERTY_MAPS

std::pair<rb_tree_string*, rb_tree_string*> p;
rb_tree_string l = rb_tree_string(*this);
// need to find 'y' in l to
110 // comply with immutability

// y's left and right subtrees
base_ptr s1 = right[y];
base_ptr s2 = left[y];
115 base_ptr x = parent[y];

while(x)
{
120 if(y==left[x])
{
left[x] = s2;
s2 = x;

```

```

    }
    else // y==right[x]
125     {
        right[x] = s1;
        s1 = x;
    }
    y = x;
130     x = parent[x];
}

rb_tree_string* t1 = s1;
rb_tree_string* t2 = s2;
135
return p(t1, t2);
}

template <class Key, class Value, class KeyOfValue
140         , class Compare, class Alloc>
void
rb_tree_string<Key, Value, KeyOfValue, Compare, Alloc>
:: print_rb_tree_inorder(const_base_ptr p)
{
145     RB_TREE_USE_CONST_PROPERTY_MAPS

    if(p != 0)
    {
        if(left[p] != 0)
150         print_rb_tree_inorder(left[p]); // print left subtree
        std::cout << color[p] ? "(black)_" : "(red)_"
            << value[p] << "_ " << endl; // print this node
        if(right[p] != 0)
            print_rb_tree_inorder(right[p]); // print right subtree
155     }
}

// 21.3.5 modifiers
// set insert
160 template <class Key, class Value, class KeyOfValue
        , class Compare, class Alloc>
typename rb_tree_string<Key, Value, KeyOfValue
        , Compare, Alloc>::iterator
rb_tree_string<Key, Value, KeyOfValue
165         , Compare, Alloc>::insert(const value_type& x)
{
    return (*this).insert_equal(x); // call to rb_tree insert function
}

// 21.3.7.1 non-member function
170 template <class Key, class Value, class KeyOfValue
        , class Compare, class Alloc>
typename rb_tree_string<Key, Value, KeyOfValue
        , Compare, Alloc>::rb_tree_string&
175 rb_tree_string<Key, Value, KeyOfValue, Compare, Alloc>
::operator+(const rb_tree_string& t)
{
    return (*this).join(t);
}

```

```
    }  
180 }  
  
#endif // CPHSTL_RB_TREE_STRING_CPP
```