TREES IN STL

The Standard Template Library does not provide any templates with `Tree` in their name. However, some of its containers — the `set<T>`, `map<T1, T2>`, `multiset<T>`, and `multmap<T1, T2>` templates — are generally built using a special kind of *self-balancing binary search tree* called a *red-black tree.* A self-balancing BST ensures that the tree is always as balanced as possible, so that searches take $O(\log_2 n)$ time. The study of these trees is beyond the level of this text and is left to the sequel.[1] In this section, we take a brief look at the `set<T>` and `map<T1, T2>` templates.

THE `set<T>` CONTAINER

The STL `set<T>` template is designed to model a mathematical set. As such, it supports the following operations (among others):

- `bool empty()` — return true if and only if there are no values in the set
- `int size()` — return the number of values in the set
- `pair<iterator, bool> insert(T aValue)` — insert *`aValue`* into the set
- `void erase(T aValue)` — remove *`aValue`* from the set
- `void clear()` — erase all values from the set
- `iterator find(T aValue)` — return an iterator to *`aValue`* (or `end()`, if *`aValue`* is not present in the set)
- `int count(T aValue)` — return the number of occurrences of *`aValue`* within the set
- `iterator begin()` — return an iterator to the first value in the set
- `iterator end()` — return an iterator pointing beyond the last value in the set

The assignment (`=`), equality (`==`), and less-than (`<`) operators are also defined for `set<T>`, as well as many others.[2]

In keeping with the mathematical notion of a set, a `set<T>` will never contain duplicate values. This is a primary difference between the `set<T>` and `multiset<T>` templates: a `set<T>` will ignore attempts to insert values already present in the `set<T>`; but a `multiset<T>` may contain duplicate values, and so attempts to insert values already present in a `multiset<T>` will succeed.

The following is a simple program that demonstrates using the `set<T>` and `multset<T>` containers. It reads a sentence from the keyboard `char`-by-`char`, and inserts each character read into a `set<char>` and into a `multiset<char>`. It then iterates through the `char` values in each container, displaying each one.

1. *C++: An Introduction to Data Structures* by Larry Nyhoff (Upper Saddle River, NJ: Prentice Hall, Inc. 1999)

2. For a detailed treatment of these STL containers, see *STL Tutorial And Reference Guide Second Edition,* by Musser, Derge, and Saini (Addison-Wesley, 2001).

Code15-1 The `set<T>` and `multiset<T>` Containers.

```
// setDriver.cpp demonstrates set<T> and multiset<T>.

#include <iostream>  // cin, cout, ...
#include <set>       // set<T>, multiset<T>, ...
using namespace std;

int main()
{
   cout << "To demonstrate set<T> vs. multiset<T>,\n"
        << " please enter a sentence: ";
   char ch;
   set<char> charSet;
   multiset<char> charMultiSet;
   for (;;)
   {
      cin.get(ch);
      if (ch == '\n') break;
      charSet.insert(ch);
      charMultiSet.insert(ch);
   }

   cout << "\nThe set<T>: ";
   for (set<char>::iterator it = charSet.begin();
        it != charSet.end(); it++)
      cout << *it << ',';
   cout << endl;

   cout << "\nThe multiset<T>: ";
   for (multiset<char>::iterator it = charMultiSet.begin();
         it != charMultiSet.end(); it++)
     cout << *it << ',';
   cout << endl;
}
```

Sample Run:

```
To demonstrate set<T> vs. multiset<T>
 please enter a sentence: A babboon blew up a balloon

The set<T>:  ,A,a,b,e,l,n,o,p,u,w,

The multiset<T>:  , , , , ,A,a,a,a,b,b,b,b,b,e,l,l,l,n,n,o,o,o,o,p,u,w,
```

Note that the `set<char>` contains just one instance of each character, but the `multiset<char>` contains each character (including the multiple spaces).

The STL `set<T>` template provides a convenient off-the-shelf container that can be used in any situation where the mathematical properties of a set are needed.

THE map<T1, T2> CONTAINER

The STL `map<T1, T2>` template is sometimes called an *associative array* and is designed to model a *dictionary*. Just as a dictionary maps a word to its definition, so a `map<T1, T2>` maps an arbitrary type `T1` to another arbitrary type `T2`.

A normal array or vector allows us to "map" integer values to the kind of value in the array. For example, if we declare:

```
string friend[3] = {"Anne", "Bob", "Chris"};
```

then the expression:

```
friend[0]
```

produces the string value `"Anne"`; the expression

```
friend[1]
```

produces the string value `"Bob"`; and so on. In essence, our array named `friend` is a container that maps the integers `0`, `1`, and `2` to the string values `"Anne"`, `"Bob"`, and `"Chris"`, respectively.

More generally, a declaration

```
SomeType anArray[SOME_SIZE];
```

declares `anArray` as a structure that can map the integers 0, 1, 2, ... to values of *`SomeType`*. Thus, an array or vector lets us map `int` values to any other type but will not let us map an arbitrary type to any other type. For example, how can we map the strings `"Anne"`, `"Bob"`, and `"Chris"` back into the integers `0`, `1`, and `2`?

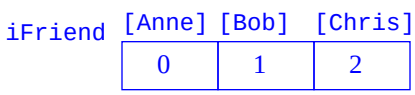
For situations where we need to map an arbitrary type `T1` to another arbitrary type `T2`, the Standard Template Library provides the `map<T1, T2>` container. For example, to map the strings `"Anne"`, `"Bob"`, and `"Chris"` back into the integers `0`, `1`, and `2`, we can declare:

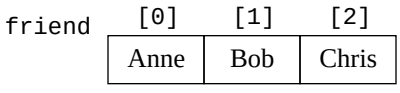
```
map<string, int> iFriend;
```

This in essence defines `iFriend` as an array that associates `string` values with `int` values. We can then "assign" mappings as follows:

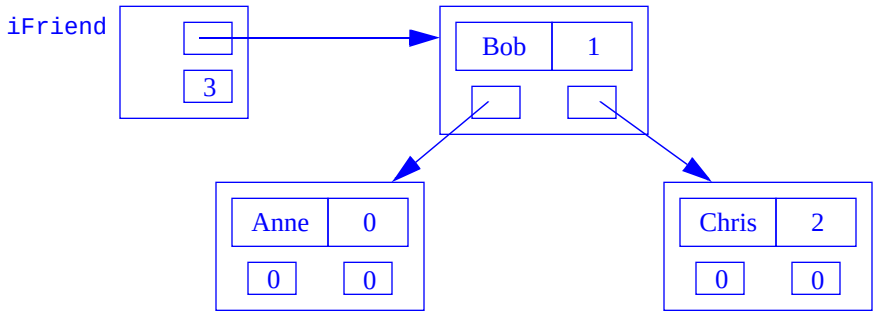
```
iFriend["Anne"] = 0;
iFriend["Bob"] = 1;
iFriend["Chris"] = 2;
```

Conceptually, this builds `iFriend` as an array that we might visualize as follows:





In actuality, `iFriend` is built as a self-balancing binary search tree that we can visualize as follows:



The `map<T1, T2>` container is thus useful in any situation in which we need to map values of some non-integer type to any other type.

Some of the operations we can perform on a `map<T1, T2>` include:

- `bool empty()` — return true if and only if the map contains no values
- `int size()` — return the number of values in the map
- `T2 operator[](T1 aValue)` — retrieve the `T2` value associated with *`aValue`* (when used on the left-side of an assignment, create a new entry for *`aValue`*)
- `int erase(T1 aValue)` — erase the entry for *`aValue`*
- `void clear()` — erase all entries
- `iterator find(T1 aValue)` — return an iterator to the pair<T1, T2> containing *`aValue`* (return `end()` if no such pair exists)
- `int count(T1 aValue)` — return the number of values whose T1 value is *`aValue`*
- `iterator begin()` — return an iterator to the first value in the map
- `iterator end()` — return an iterator pointing beyond the last value in the map

The assignment (`=`), equality (`==`), and less-than (`<`) operators are also defined for `map<T1, T2>`, as well as many others.[3]

To illustrate the use of these operations, the following program presents an alternative solution to our Big-10 Mascots program from Section 6.1. This version uses a `map<string, string>` to map the Big-10 universities to their mascots.

3. See Footnote 1.

Code15-2 The `map<T1, T2>` Container.

```
/* big10Mascots.cpp stores and looks up Big-10 mascots
   in map<T1, T2>. */

#include <iostream>  // cin, cout, ...
#include <map>       // map<T1, T2>, ...
#include <string>    // string
using namespace std;

int main()
{
   map<string, string> mascots;                 // build the map

   mascots["Illinois"]       = "Fighting Illini";
   mascots["Indiana"]        = "Hoosiers";
   mascots["Iowa"]           = "Hawkeyes";
   mascots["Michigan"]       = "Wolverines";
   mascots["Michigan State"] = "Spartans";
   mascots["Minnesota"]      = "Golden Gophers";
   mascots["Northwestern"]   = "Wildcats";
   mascots["Ohio State"]     = "Buckeyes";
   mascots["Penn State"]     = "Nittany Lions";
   mascots["Purdue"]         = "Boilermakers";
   mascots["Wisconsin"]      = "Badgers";

   for (;;)                                     // loop:
   {
      cout << "\nTo look up a Big-10 mascot, enter the name "
           << "\n of a Big-10 school ('q' to quit): ";
      string university;
      getline(cin, university);                 // get a school

      if (university == "q") break;             // quit?
                                                // look up the school
      map<string, string>::iterator it = mascots.find(university);

      if (it != mascots.end())                  // if found
      // cout << "--> " << it->second <<  endl;  // display it
         cout << "--> " << mascots[university] << endl;
      else
         cout << university << " is not a Big-10 school "
              << "(or is misspelled, not capitalized, etc?)" << endl;
}
```

Sample Runs:

```
To look up a Big-10 mascot, enter the name
 of a Big-10 school ('q' to quit): Michigan
--> Wolverines
```

```
To look up a Big-10 mascot, enter the name
 of a Big-10 school ('q' to quit): Ohio State
--> Buckeyes

To look up a Big-10 mascot, enter the name
 of a Big-10 school ('q' to quit): Missouri
Missouri is not a Big-10 school (or is misspelled, not capitalized, etc?)

To look up a Big-10 mascot, enter the name
 of a Big-10 school ('q' to quit): Minnesota
--> Golden Gophers

To look up a Big-10 mascot, enter the name
 of a Big-10 school ('q' to quit): q
```

Note that once we have determined that university is present in our mascots map, we can retrieve its mascot in either of two ways. The expression:

```
mascots[university]
```

uses the subscript operator to do the retrieval. While this is readable and intuitive, it is somewhat inefficient, because `operator[]` must re-search the map to find its value for university. The more efficient approach is to use the iterator that the `find()` method returns, which points to the `pair<T1, T2>` that each map element stores.[4] The two values stored in a `pair<T1, T2>` can be accessed via the public instance variables `first` and `second`, respectively. The expression:

```
it->second
```

thus retrieves the `second` value (i.e. the string representing the mascot) in the pair to which the iterator `it` returned by `find()` points. This approach is somewhat harder to read, but is much more efficient, since it uses the iterator returned by `find()`. Since this iterator already points to the map-element containing `university` and its mascot, this approach avoids the redundant search required by the use of `operator[]`.

SUMMARY

In summary, the Standard Template Library uses self-balancing binary search trees called red-black trees to implement its `set<T>`, `multiset<T>`, `map<T1, T2>`, and `multimap<T1, T2>` containers. While these container cover a broad range of scenarios, we still occasionally need to build our own tree, as we shall see in the next section.

4. `pair<T1, T2>` is yet another STL container.