Chasing the STL: Implementing a Trie in C++

Honors Project

George Hilliard — gh403

November 14, 2013

Contents

"On my honor, as a Mississippi State University student, I have neither given nor received unauthorized assistance on this academic work."

CSE2383
Class Section H02
Rikk Anderson

1. Introduction

For my honors project, I decided to implement a trie data structure in C++. Having recently become familiar with C++11, I recognized the power of the STL, so I wanted to make the container act as much like a C++11 STL container as possible. In particular, my trie is much like the `std::set`[1] data structure, because it cannot store duplicates, automatically sorts its elements, and uses a tree internally to store data. (Some tries store mappings, where each key leaf maps to a value, but mine does not.) The interface to my trie implementation, `trie`, is very similar to that of `set`. The class possesses iterators and overloaded `insert`, `find`, and `erase` methods whose signatures are nearly identical to `set`'s. `trie` also includes a few functions from the array-like class `vector`.

Perhaps more importantly, `trie` is fully templatized (as is `set`), allowing one to create a `trie` that holds any other container that provides a forward iterator and a few other methods. This means that one can instantiate a `trie<basic_string<char>>` (a trie of strings), a `trie<vector<int>>` (a `trie` of arrays of `int`s), and even a `trie<trie<vector<int>>>` (a `trie` of `trie`s of arrays of `int`s)!

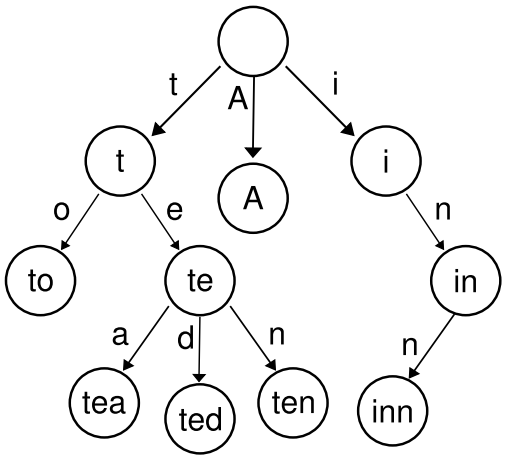
2. Internal Structure

The similarities to `set` end once the actual `trie` data structure is examined. This section will discuss the functionality of "`trie<T>`" in a *general*, templatized way. Note that `trie<T>` (like other STL containers) uses the declarations

```
typedef T key_type;
typedef T value_type;
```

The types `T` and `value_type` will be used interchangably here.

`trie<T>` separates `T::value_type`s and stores those individually in a tree that describes how the `value_type`s are arranged into `T`s. (The example trie, to the right, holds the strings `"A"`, `"to"`, `"tea"`, `"ted"`, `"ten"`, and `"inn"`.) This is radically different from a `set<T>`, which stores `T`s intact in a binary tree describing less-than or greater-than relationships between `T`s.



A `trie` has just two members. The first is merely a boolean flag indicating whether or not the node is a leaf. The other is a mapping of `T::value_type` to pointers to other `trie`s. Nodes in a `trie` do not store any information about their own meaning—they only store information about what `T::value_type`s their *children* represent.

This particular structure was chosen because it saves space: if a node's child has no children, then the null pointer can be interpreted as "this child should exist, but it has no children." This saves the space the extra `trie` node would have occupied. It does substantially complicate the insertion and iteration code, however.

In sum, it is convenient to think of a `trie` as the following structure:

[1] All STL containers mentioned hereafter will be without the `std::` namespace delimiter for brevity.

```
struct trie {
    std::map<typename T::value_type, trie<T>*> children;
    bool is_leaf;
};
```

The actual details of the implementation are discussed later.

3. Benchmarking

3.1. Method

The class was tested alongside several of the most common STL containers, including of course `set`, `vector`, `unordered_set`, and `list`. The idea was to time a function that performs a specific operation (such as insertion) many times. In these tests, about 12MiB of words from UNIX's `/usr/share/dict` dictionaries were used as input.

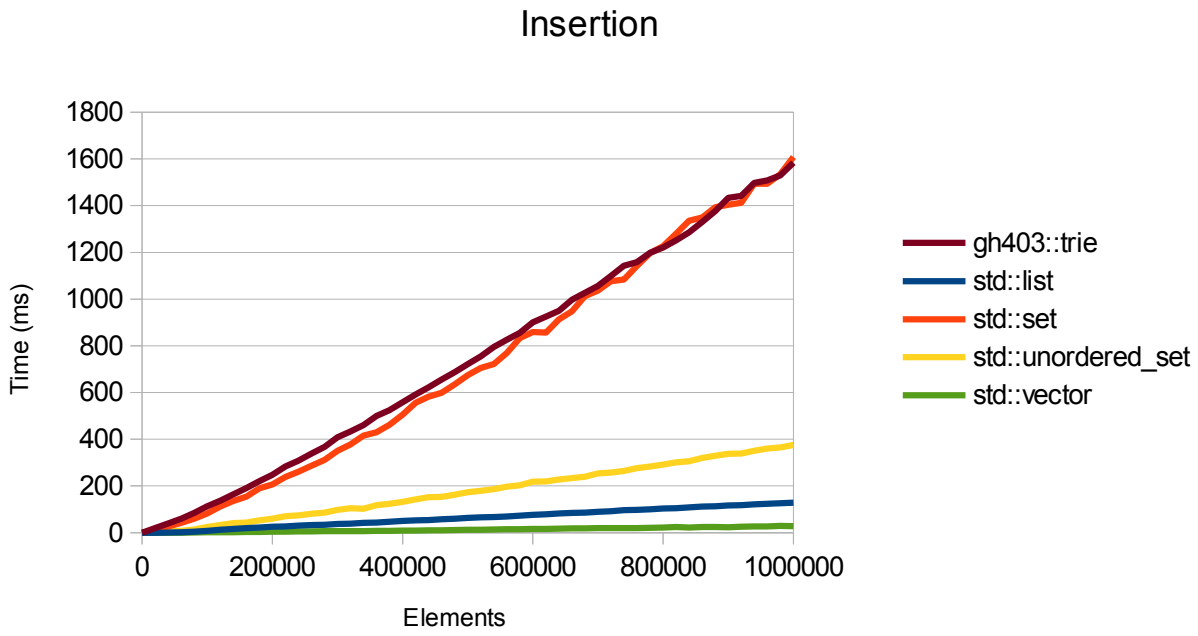
Care had to be taken that the tested code remained as tight as possible—for example, reading from the dictionary file should not be timed! The following example is a slightly-simplified example of the code used to time the insertion operation for each of the five data types used. The words have already been read into a shuffled `vector<string>` called `source`. The `timeFunctionCall` function just times the function that it is passed (the benchmark code always passes lambdas) and returns a result in milliseconds.

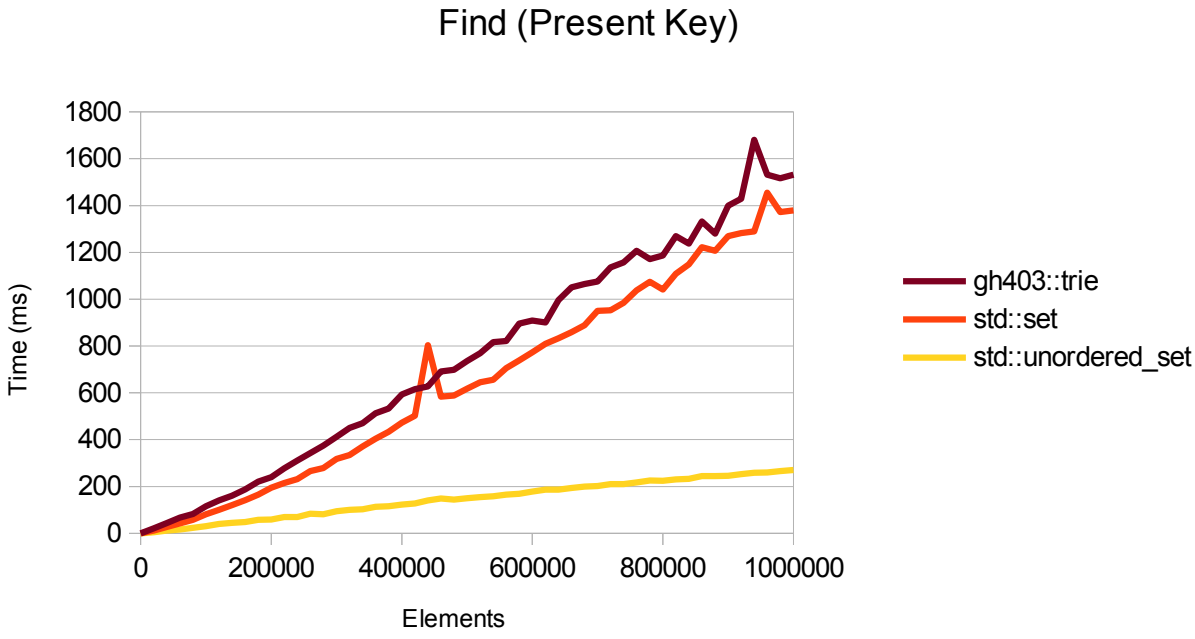
```
timeFunctionCall( [&]{ l->insert(l->begin(), source.begin(), source.end()); } );
timeFunctionCall( [&]{ s->insert(source.begin(), source.end()); } );
timeFunctionCall( [&]{ u->insert(source.begin(), source.end()); } );
timeFunctionCall( [&]{ v->insert(v->begin(), source.begin(), source.end()); } );
timeFunctionCall( [&]{ t->insert(source.begin(), source.end()); } );
```

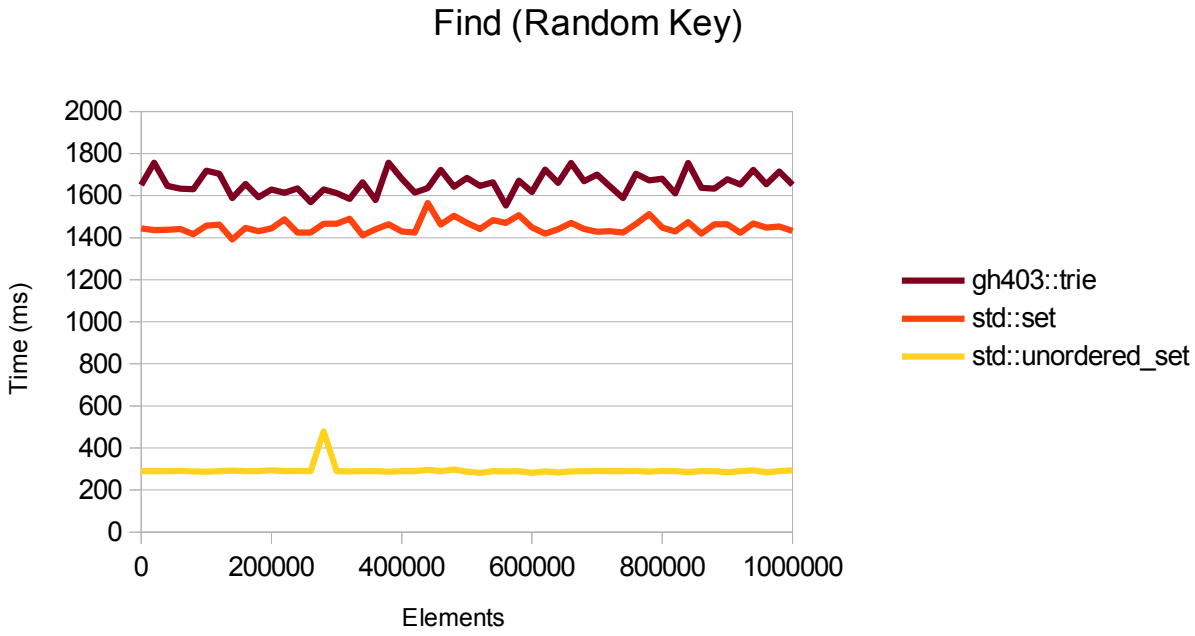
(It would be difficult to get a tighter loop than this!) These functions are repeated many times for different sizes of `source`.

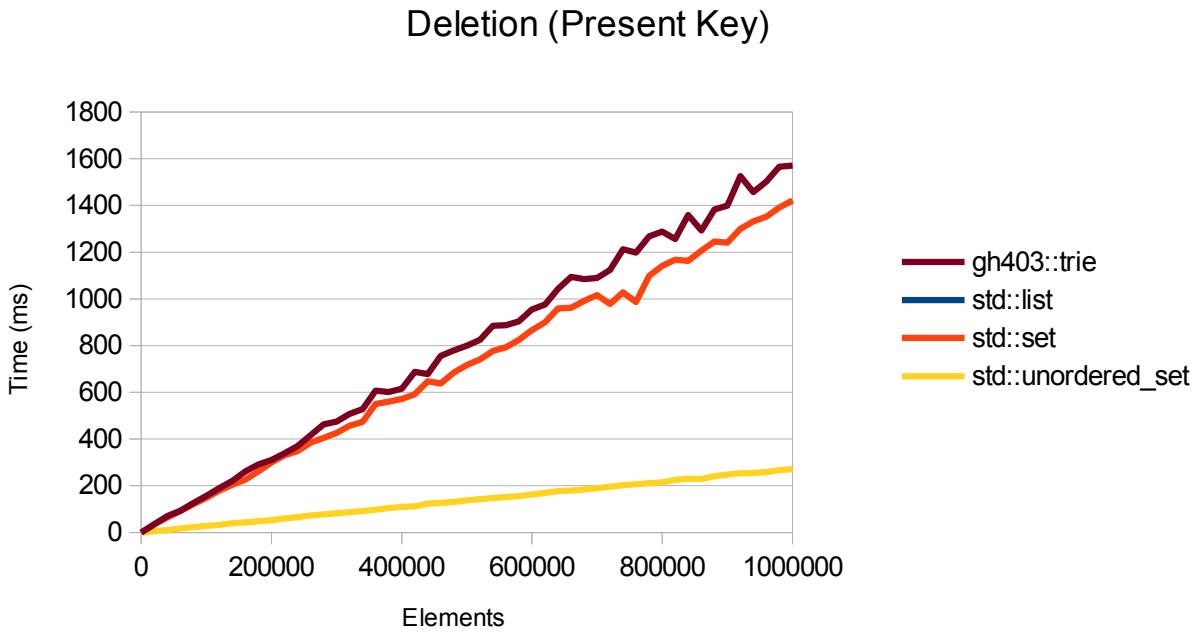
Some of the benchmark terminology may need to be described. A *present key* search or deletion means that the key is known to exist in the data structure. Inversely, a *random key* means that the key was selected out of the entire set of dictionary words—not just those in the data structure. On all tests except insertion, the data structure was filled with a copy of the entire set of words. The benchmark source code will clarify any remaining ambiguities.

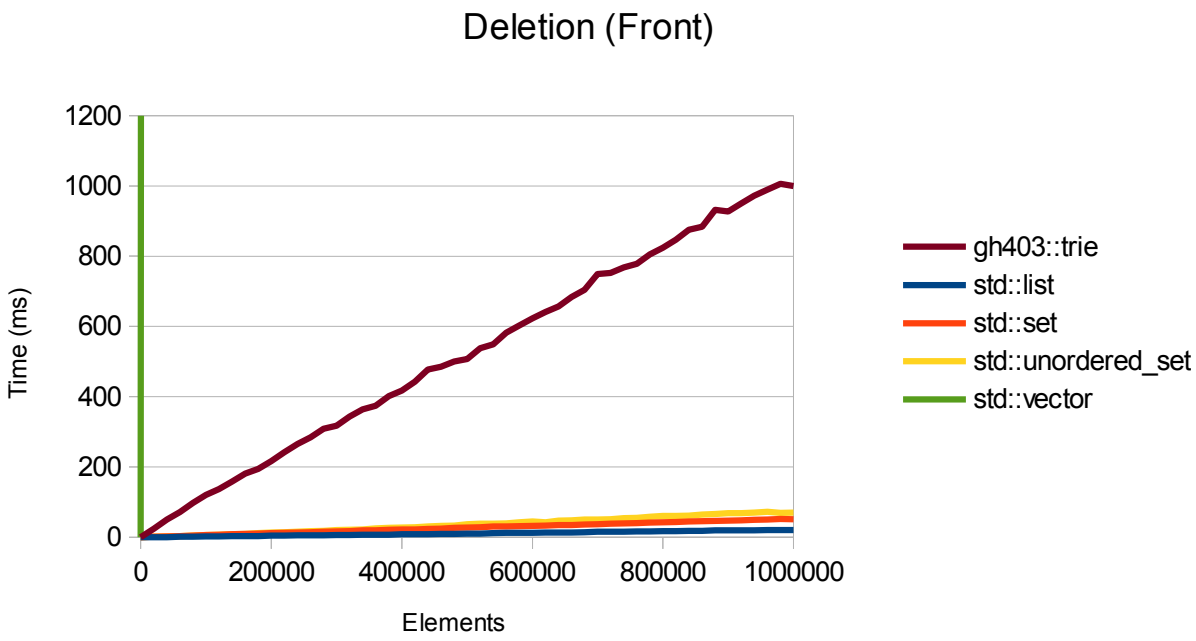
3.2. Data

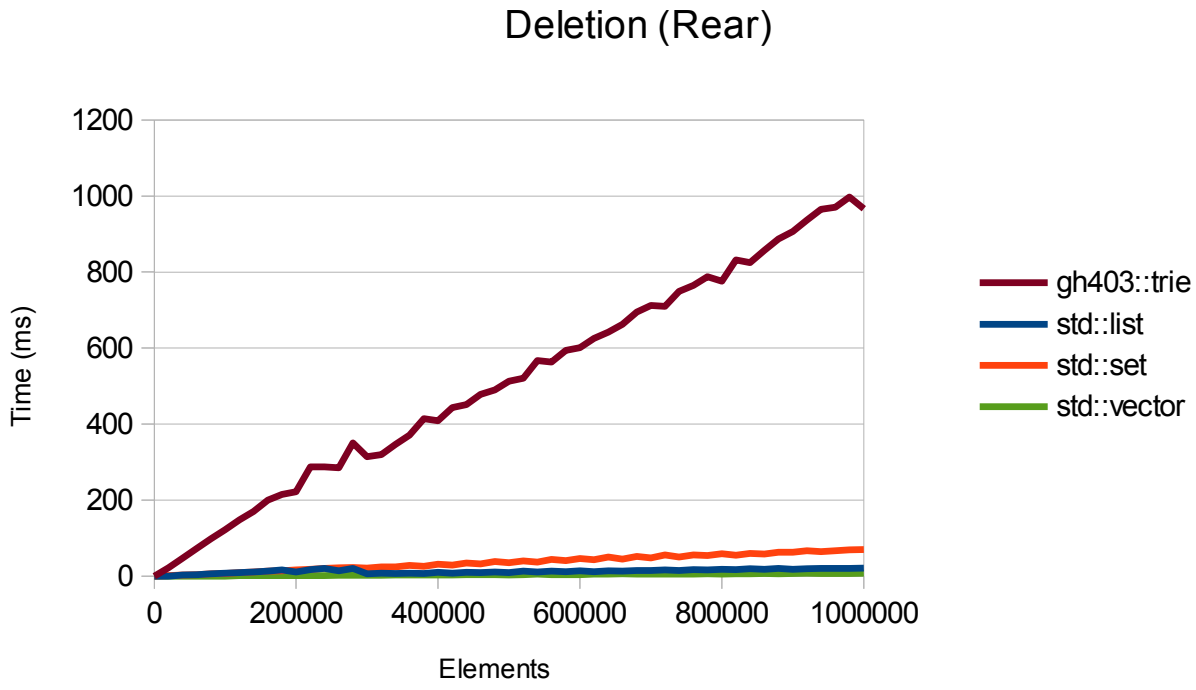

Insertion
Time (ms)
1800
1600
1400
1200
1000
800
600
400
200
0
0
200000
400000
600000
800000
1000000
Elements
gh403::trie
std::list
std::set
std::unordered_set
std::vector


Find (Present Key)
Time (ms)
1800
1600
1400
1200
1000
800
600
400
200
0
0
200000
400000
600000
800000
1000000
Elements
gh403::trie
std::set
std::unordered_set


Find (Random Key)
Time (ms)
2000
1800
1600
1400
1200
1000
800
600
400
200
0
0
200000
400000
600000
800000
1000000
Elements
gh403::trie
std::set
std::unordered_set


Deletion (Present Key)
Time (ms)
1800
1600
1400
1200
1000
800
600
400
200
0
0
200000
400000
600000
800000
1000000
Elements
gh403::trie
std::list
std::set
std::unordered_set





All this raw data is included in the file `bench.csv`. The benchmarking was performed on an Intel Core i7 3612-QM at 2.4GHz with 8GiB RAM under Arch Linux x86_64, compiled with G++ 4.8.2.

3.3. Reaction

The ultimate goal with `trie` was to compete with the STL, in terms of both size and speed. Space used is comparable with `set`; `trie` uses almost a negligable amount more. In terms of speed, though, the STL libraries clearly have a decisive advantage. Although `trie` manages

to compete with `set` for insertion, other operations on `trie` are generally slightly slower than those on `set`.

The outstanding exceptions are of course the front and rear deletion operations. `trie` gets its clock cleaned by every STL container (except, obviously, `vector` for front deletion). My theory is that `trie`'s iterator implementation is still vastly suboptimal. Iterators are used and passed around extensively by the STL libraries and `trie`, both as return values and as arguments; if the iterator is slow, the library will be slow. The `trie<T>::iterator` implementation constantly updates its assembled return value for `operator*()` so that that operation is fast; however, this comes at a great cost to `find` and `delete` speed.

My takeaway from benchmarking is the following. First, the STL implementation is already nearly as fast and well-written as one could hope for; at least, the GNU implementation is solid. It is extremely unlikely that a competing data structure will be able to outperform one of the STL containers in every way. In fact, my structure seems to be "chasing the STL" in nearly every aspect. Second, the difference in time required by `set` and `unordered_set` is noteworthy. The latter consumes more memory, but if the memory can be spared, an `unordered_set` is very preferable. Finally, fast iterators and well-chosen low-level data structures are keys to fast STL operations.

And of course, don't perform front deletions on a `vector`.

4. Implementation Notes

There are many interesting aspects of `trie`; this section talks about some of the highlights.

4.1. Feature Matrix

Following is a comparison of some of the most commonly used functions provided by the STL and by `trie`. Notice that the `trie`'s set of provided functions generally matches those that `set` provides. There are a few omissions, most notably the reverse iterators and the emplace family. Reverse iterators were omitted simply due to a lack of time to get them working to my satisfaction; there is not a technical reason they could not be implemented. Emplace operations were omitted purposefully; it is not meaningful to emplace an object that will immediately be cut up into its constituent `value_type`s!

	trie	std::list	std::vector	std::set	std::unordered_set
(constructor)	•	•	•	•	•
(destructor)	•	•	•	•	•
operator=	•	•	•	•	•
empty	•	•	•	•	•
size	•	•	•	•	•
max_size	•	•	•	•	•
resize		•	•		
begin	•	•	•	•	•
end	•	•	•	•	•
rbegin	•	•	•	•	
rend	•	•	•	•	
cbegin	•	•	•	•	•
cend	•	•	•	•	•
crbegin	•	•	•	•	
crend	•	•	•	•	
front		•	•		
back		•	•		
find	•			•	•
count	•			•	•
lower_bound				•	
upper_bound				•	
equal_range				•	•
assign		•	•		
emplace_front		•			
push_front		•			
pop_front		•			
emplace_back		•	•		
push_back		•	•		
pop_back		•	•		
emplace		•	•	•	•
emplace_hint				•	•
insert	•	•	•	•	•
erase	•	•	•	•	•
clear	•	•	•	•	•
swap	•	•	•	•	•

4.2. Child Array

The `trie` uses a vector of pairs as its internal list of children, as shown here:

```
// data members and types
typedef std::vector<std::pair<typename T::value_type, std::unique_ptr<trie<T>>>> child_map_type;
child_map_type children;
```

The `child_map_type` was carefully chosen. Initially it was a `std::map<T::value_type, std::unique_ptr<trie<T>>>`. Although this worked, it consumed an inordinate amount of memory because of the overhead for small `map`s. Replacing the `map` with a `vector<pair>` helped tremendously. Of course, this created a new problem—`vector` does not sort its members! This limitation can be overcome by sorting the elements as they are

inserted. The following code is used in `trie<T>::insert` to find either the correct child node, or the place where the new one should go:

```
auto childIt = std::upper_bound(currentNode->children.begin(), currentNode->children.end(), *inputIt,
                                [&inputIt](const typename T::value_type& x,
                                           const std::pair<typename T::value_type,std::unique_ptr<trie<T>>>& y)
                                          { return x <= y.first; });
```

Notice the lambda—it performs an unusual comparison (`upper_bound` expects the comparison functor to perform a `<` comparison). If the node we're looking for is not present, this comparison allows us to stop on the element *before* the one `upper_bound` would normally return. `upper_bound` has a runtime of $O(\log n)$, so it's nice and efficient.

The other interesting thing about the child array is its use of `unique_ptr`s to simplify destruction of nodes. These "smart pointers" automatically destruct the object they point to when they are reassigned or themselves destructed. This has two effects. First, it greatly simplifies algorithms, as the compiler can be allowed to write the `trie` destructor and must automatically destroy subtries that get replaced. Second, it makes it nigh on impossible to create a memory leak! Smart pointers are a simple but great addition to C++.

4.3. Iterators

Iterators are probably the most complex part of this entire project. They are not trivial to write for a trie, because a trie has several properties that exclude most of the usual iterator optimizations such as tree threading. My implementation uses a stack of parent nodes and parent node iterators to provide the same functionality that the hardware stack would when walking the tree recursively. Here is the declaration of a `trie<T>::iterator`'s member variables:

```
// data members and types
struct state {
    const trie<T>* node;
    typename trie<T>::child_map_type::const_iterator node_map_it;
};

std::stack<state> parents;
T built;
bool at_end;
bool at_leaf;
```

Whenever an iterator descends to a new `trie` node, it pushes to its stack a) the node's address and b) a `begin` iterator to the node's child list. Then, as the iterator walks around that node, the iterator in the iterator's stack is modified.

A `trie` also has two flags to signify special conditions. The `at_end` flag just indicates that the iterator has walked "past the end" of the root `trie`. Incrementing such an iterator is undefined and will probably cause a memory access error. The `at_leaf` flag indicates that the node in the top of the stack is also a leaf; such a node is visited before any of its children.

`const_cast<>` ing Pointers

When implementing the erase operation, I ran into a nasty issue. `erase` is required by C++ to accept an `iterator`. Note that in the above code snippet from the iterator's declaration that the node pointer is declared const! This makes sense for an iterator; after all, it is an error if the iterator's code modifies the trie it is supposed to just walk over. However, it is obviously *not* a (logical) error for erase to modify the object pointed to by that pointer. However, C++ balks when trying to modify a const object!

In this case, it is actually correct to cast const-ness away. The reason for this is that the iterator passed to `erase` must be an iterator of `*this`, or behavior is undefined. Because this non-const function is executing, it means that the pointer *must* be to a non-const `trie`!

After resolving this issue, I immediately ran into another issue, namely that GNU's STL implementation won't fully support C++11 until version 4.9 is released in 2014. Until then, passing a `const_iterator` to `erase` functions in STL is not implemented. This lead to what is in my opinion the ugliest workaround in the project.

```
// As soon as GCC 4.9 is released, this whole thing should be replaced with:
//const_cast<trie<T>*&>(it.parents.top().node)->children.erase(it.parents.top().node_map_it);

auto nonconst_it = const_cast<trie<T>*&>(it.parents.top().node)->children.begin();
std::advance(nonconst_it, std::distance(it.parents.top().node->children.cbegin(),
            it.parents.top().node_map_it));
const_cast<trie<T>*&>(it.parents.top().node)->children.erase(nonconst_it);
if(it.parents.top().node->children.size() == 0 && it.parents.size() > 1) {
    it.parents.pop();
    const_cast<std::unique_ptr<trie<T>>&>(it.parents.top().node_map_it->second).reset(nullptr);
}
```

This code creates a second, non-const iterator and increments it to the same position as the const iterator we could directly pass if GNU supported it. This probably creates a minor performance hit and should certainly be changed when the compiler can support it.

5. Conclusion

Two things could be done to improve the efficiency of `trie`. First, it could be modified to store data in a PATRICIA trie. Essentially, this structure is a trie, but with the added feature that it collapses nodes with single children into single nodes. However, this would not be a trivial modification to make.

The second thing that could be done is much simpler. Switching the iterator around to only update its `built` member when `operator*()` is called would speed up walking around, although this would, as stated above, lead to slower access times. My tests do not measure access time, but it is likely negligable for all data structures.

Overall, my `trie` is feature-complete and can pass a set of tests. It stores information in a manner very different from that of `set`, but externally appears very similar, including being fully templatized. On several tests, it is nearly as fast as `set`. Overall, I am generally happy with the way it works, although there is certainly still room for improvement.

A. Declaration of `trie`

The following is the complete declaration of the implemented `trie`. Of course, it is also present in the file `trie.h`, but it may be handy to have here while reading.

```
template<typename T>
class trie {
    // data members and types
    typedef std::vector<std::pair<typename T::value_type, std::unique_ptr<trie<T>>>> child_map_type;
    child_map_type children;
    bool is_leaf = false;

public:
    // misc. declarations
    class iterator;
    typedef T key_type;
    typedef T value_type;
    typedef size_t size_type;
    typedef iterator const_iterator;

    // constructors
    trie(bool = false);
    trie(const trie<T>&);
    trie(trie<T>&&);
    template<typename InputIt> trie(InputIt, InputIt, bool = false);
    trie(std::initializer_list<T>);

    // destructor, auto-generated one is fine
    ~trie() =default;

    // operators
    trie<T>& operator=(trie<T>);

    // iterators and related
    iterator begin() const;
    iterator end() const;
    const_iterator cbegin() const { return begin(); }
    const_iterator cend() const { return end(); }

    // other members
    std::pair<iterator,bool> insert(const value_type&);
    template<typename InputIt> void insert(InputIt, const InputIt&);

    iterator erase(const_iterator);
    size_type erase(const key_type&);
    iterator erase(const_iterator, const_iterator);
    void clear();

    bool empty() const { return children.empty() && !is_leaf; }
    size_type size() const;
    constexpr size_type max_size() const;

    const_iterator find(const key_type&) const;
    size_type count(const key_type&) const;

    void swap(trie<T>&);
    static void swap(trie<T>& a, trie<T>& b) { a.swap(b); }
};
```

B. References

This is a list of resources that have been invaluable in researching and writing this project.

1. **StackOverflow** I cannot recommend this website enough. Intelligent questions are greeted warmly with incredibly detailed and thoughtful answers. This usually happens in under an hour, and the best answers trickle to the top within a day. Posting here helped me with some of the gnarliest parts of this project:
   - Const-correctness semantics in C++
   - `std::vector::erase()` and `const_iterators` with g++
   - Performance cost of comparing two C++ iterators
   - STL iterator before `std::map::begin()`
   - Adding element to back of STL container
   - Efficient algorithm to step through all of linked list (in C++)
2. `cppreference.com` is an excellent wiki-style website documenting C++. They have excellent C++11 support, and plenty of good examples.
3. Bjarne Stroustrup's **C++11 FAQ** is a very good overview of C++11 as well as a good repository of tips and syntax.