Implementation of a circular singly linked list: slist

Henrik Skovby
skovby@diku.dk

Abstract. I have implemented, documented and tested an implementation of a circular singly linked list with a tail pointer. In the light of the tests I conclude that the implementation is working correctly, and that this implementation is slightly slower than SGI's implementation of `slist`, due to my inclusion of a tail pointer.

1. Research

1.1 Introduction

This report describe the development of a singly linked list (`slist`) for the Copenhagen STL project at DIKU, in connection with the course Performance Engineering 2001. The `slist` is not as opposed to `list` part of the C++ standard. This means, that I in principle can choose the interface of `slist` as I want. As I starting point I have chosen to follow the specification and interface of SGI's version of `slist`. The advantage of this choice is that programs using SGI's `slist` can easily switch implementation to use `slist` of the CPH STL.

1.2 Data structure

When implementing a linked list you first has to decide on a data structure. There are quite a few different data structures to choose from [Preiss 1999] (see Figure 1.1), and the best choice depend on the semantics you want for the singly linked list. As I mentioned in section 1.1 the semantics I want is as a starting point constrained by the semantics of SGI's `slist`. However, in my research I found some special situations using `slist`, which I was not quiet satisfied with. I found that the following examples gave a segmentation fault in the SGI implementation.

```
//Ex. 1
    slist<int> sl1;
    sl1.insert_after(sl1.begin(), 1);
//Ex. 2
    slist<int> sl2(10, 1);
    sl2.insert_after(sl2.end(), 1);
    sl2.erase_after(sl12.end());
```

CPH STL Report 2001-10, April 2001.

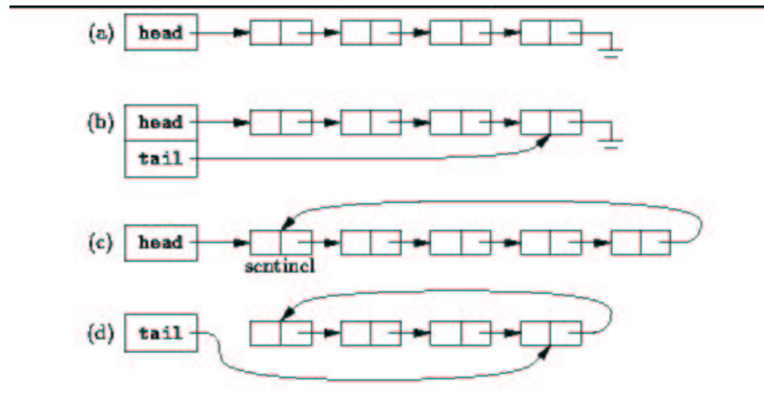

Figure 1.1. Different possible data structures.

In the first example an element `1` is inserted into an empty list after the beginning and in the second example an element `1` is inserted after the `end()` marker of the list. The concrete reason why these operations gives segmentation fault is because in the SGI implementation the `end()` marker is actually the `null` pointer.

A way to avoid segmentation fault is to actually give a meaning to these operations, so that the are legal. A way to do that and also keep a consistent semantic for the other member functions is to make the list circular. The first example will then succeed because `sl1.begin()++` points to it self and the element is correctly inserted at the beginning. The second example succeed because `sl1.end()++` points to the beginning of the list and the element is correctly inserted or erased respectively at the beginning of the list.

Based on this discussing I have chosen a circular list and as we shall see later there are actually also some performance advantages to gain when choosing a circular list.

1.3 Node Representation

There are at least to different node representations when implementing a singly linked list. The two versions I considered was.

```
template <class T>
struct Node1 {
    Node1* next;
    T data;
}

template <class T>
struct Node2 {
    Node2* next;
```

```
        T* data;
    }
```

The differences is in the member variable `data`. In the first example memory for element data is allocated together with the node, which gives a simpler implementation and also lesser calls to the allocator. A lot of the lists member functions ex. `previous, insert() and erase()` traverses the list looking only on the next pointer. This means that with the second structure you do not actually have to fetched the data element into the cache for all the bypassed elements, which can be an performance advantages for large elements.

I have chosen the simpler scheme for this `slist` implementation.

1.4 Sentinel

Another choice you have to make is whether we want to use an sentinel, as we can see from Figure 1.1. In general a sentinel element removes some special cases that check if the list is empty (see [Cormen et al. 1997, pp. 206-208]). Furthermore, search through the list can be improved with a sentinel value as mentioned in [Bentley 2000, p. 90]. The following code piece uses this idea.

```
    search(const T val) {
        sentinel.data = val;
        node = head;
        while( node->next != val);
        return node->next;
    }
```

As you can see you save one comparison in each iteration, that checks if you have reach the end. The drawback of this technique is that it is not thread safe, because it actually changes the list. This price is that you get a member function that to the user it does not change the state of the list, but in reality is it does.

I have chosen to use a sentinel, because it removes the special cases for when the list is empty, and despite the drawbacks I plan to explore the sentinel search technique by adding a `find()` member function to the interface.

1.5 Tail Pointer

As you can see from Figure 1.1 it is also possible to add a tail pointer to the list. This is useful when you want to support constant time insertion at the end of the list and when you want to append two lists together. The drawback is that you have to maintain the tail pointer in all other member functions.

I plan to extend the SGI `slist` in this case have therefore chosen to add a tail pointer to the implementation to allow insertion at the back.

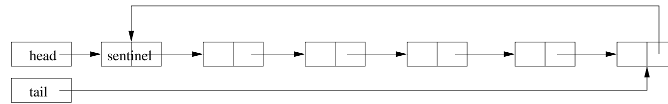

Figure 1.2. The final choice of a data structure for slist

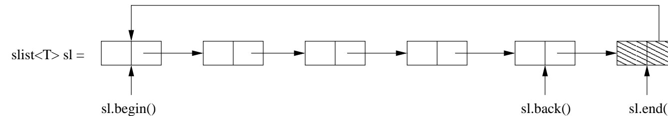

Figure 1.3. The slist from the users point of view. When operating on the slist the user can think of list as pictured here.

1.6 Block Allocation

It was noted in [Bentley 2000, pp. 136-138] that block allocation of nodes can bust the performance of list. This technique requires that you are allowed to rearrange the elements after they are inserted. Otherwise you cannot use a non-full block, when you want to insert an element between two consecutive elements in a block. This results in a huge waste of space. Like `list` SGI's `slist` is not allowed to invalidate existing iterators to data elements when inserting or deleting elements [1], which means that member functions cannot move data elements after insertion. Therefore block allocation is not useful in this container.

1.7 Finally Choice of Data structure

The data structure I end up with after this discussion is the one depicted in Figure 1.2. As you can see it is version (c) from Figure 1.1 with a tail pointer. The reason I chose (c) and not (d) was because in choosing (c) I am able to allocate space for the sentinel without calling `T's` constructor and at the same time I am able to use the sentinel search technique.

As mentioned in 1.1 I also changed the semantics of some of the member functions compared to the SGI implementation. From a users point of view the semantics of this `slist` implementation is expressed in Figure 1.3, where also the `back()` iterator is introduced.

2. Results

In this section I describe the results and the experience I gained when implementing and testing `slist`.

[1] In the last case only iterators to the deleted element is invalidated

2.1 Status on implementation

I have implemented, documented and thoroughly tested all the member functions from the SGI interface of `slist`. Still to be implemented is the member functions `back(), push_back() and find()` that I planned to implement. Furthermore, I have made performance tests of `push_front()`, `sort()`, `pop_front()`, `insert_after()`, `insert()`, `erase_after()` and `erase()`, which I compared with the SGI implementation.

2.2 Documentation

The documentation is made with Doxygen. The documentation is located in:

```
cphstl/Slist/Documentation
```

It is generated by typing:

```
cd /cphstl/Slist
make doc
```

2.3 Correctness Test

To test my implementation I have used the testing framework CppUnit ver. 3.5., which I have copied to CVS. I test all member functions with both integer lists and string lists. This test consist of the files

```
cphstl/Slist/Benchmark/slistTest.h/slistTest.cpp
```

Running the test

```
cd cphstl/Slist/Benchmark
make all
./CPH/correctnessTest
./SGI/correctnessTest
```

2.4 Performance advantages of circular lists

In implementing the member function `previous(iterator i)` which return an iterator to the previous element, I discovered that I actually saves one comparison per iteration because the list is circular. To find the previous element you have to traverse the list from the beginning, because you only have iterators in one direction. In SGI's implementation they make two checks in there stop condition. One to see if the `current` pointer is `null` and one to see if `current->next` is `i`. In my version I only have to check the last conditions because `current` never becomes `null`.

2.5 Disadvantages of Tail Pointer

When implementing the `slist` I realized that it was quiet difficult to update the tail pointer. It gives some not so nice special cases in many member functions, and it is only used in a few. Furthermore, you still cannot make

constant time removal at the end, because you need previous of `back()` (see Figure 1.3) to remove `back()`.

In this implementation I check the tail pointer on every insertion and removal. Even if more elements are inserted or erased I check if the tail pointer should be updated for each element inserted. This check is required if you want to respect the basic guarantee of exceptions safety defined in [Stroustrup 2000, Appendix E]. If you do not make the check for each element when for example inserting 10 elements, `T's` constructor might throw an exception after insertion of 5 elements and because tail pointer is incorrectly updated the list is in an inconsistent state.

2.6 Performance Tests

In the performance test I use the timer made by Steffen Nissen to measure the performance of the following member functions.

- `push_front()` see Figure 2.1
- `sort()` see Figure 2.2
- `pop_front()` see Figure 2.3
- `insert_after()` see Figure 2.4
- `insert()` see Figure 2.5
- `erase_after()` see Figure 2.6
- `erase()` see Figure 2.7

The performance test file is:

```
cphstl/Slist/Benchmark/UnitTest/performanceTest.cpp
```

Running the tests and plotting data

```
cd cphstl/Slist/Benchmark
make all
make plots
```

Apart from the test of `insert_after()` and `insert()`, which are inconclusive, it seems that SGI's implementation is a little bit faster. This result was expected, because my implementation carry around a tail pointer that have to be updated. The exception is `pop_front()` where mine implementation is a little faster. When I noticed this I discovered that in my implementation of `pop_front()`, I do not check if the tail pointer should be updated[2] The biggest difference between SGI's and mine implementation as far as performance is concerned is, that I have a tail pointer and SGI's implementation does not. So, in the light of these tests I can conclude that having a tail pointer in a singly linked list incur a slight, but noticeable performance penalty.

[2] I have fixed this bug in the code, but I have not run the performance test again, because of limited time.

3. Conclusion

I have implemented, documented and thoroughly tested an implementation of a circular singly linked list with a tail pointer. Apart from some minor changes and additions my implementation is compatible with the interface and specification of SGI's `slist`. I have made both a correctness and a performance test with to different element types. One element type was an built-in integral type and one with the larger string type. The correctness test was made using the testing framework CppUnit ver. 3.5 and this test runs without any failures. In the performance test I have measured some of the key member functions. In this test I conclude as expected, that having a tail pointer in a singly linked list incurs a slight but noticeable performance penalty.

This concludes my work on `slist` for now, which have taking about the 90 hours allocated for the development exercise in Performance Engineering. As mentioned in 2.1 there are still some more member functions to implement according to my preliminary plan. Furthermore, it could be useful to compare this version of `cphstl::slist` with a version where the tail pointer is removed. Maybe it is then possible to beat the implementation of SGI's `slist`.

References

Bentley, J. 2000. *Programming Pearls*, 2nd edition. Addision-Wesley, New Jersey.

Cormen, Thomas H., Leiserson, Charles E., and Rivest, Ronald L. 1997. *Introduction to algorithms*. The MIT Press, Cambridge, Massachusetts.

Preiss, Bruno R. 1999. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. John Wiley & Sons, Waterloo, Canada.

Stroustrup, Bjarne. 2000. *The C++ Programming Language, Special Edition*. Addison Wesley.

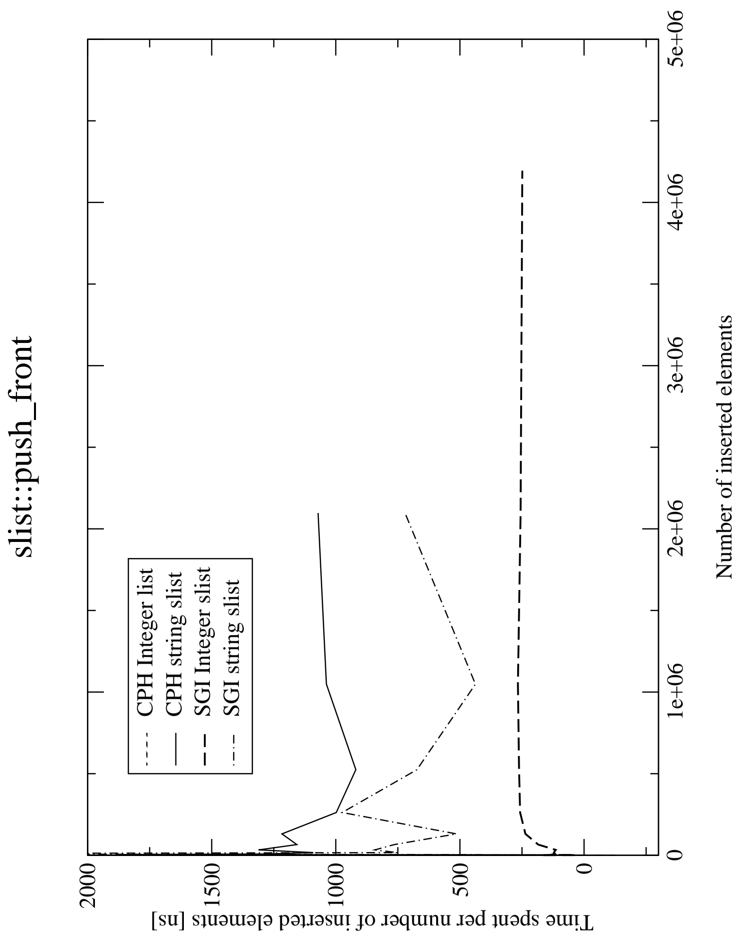

Figure 2.1. Performance test of slist::push_front

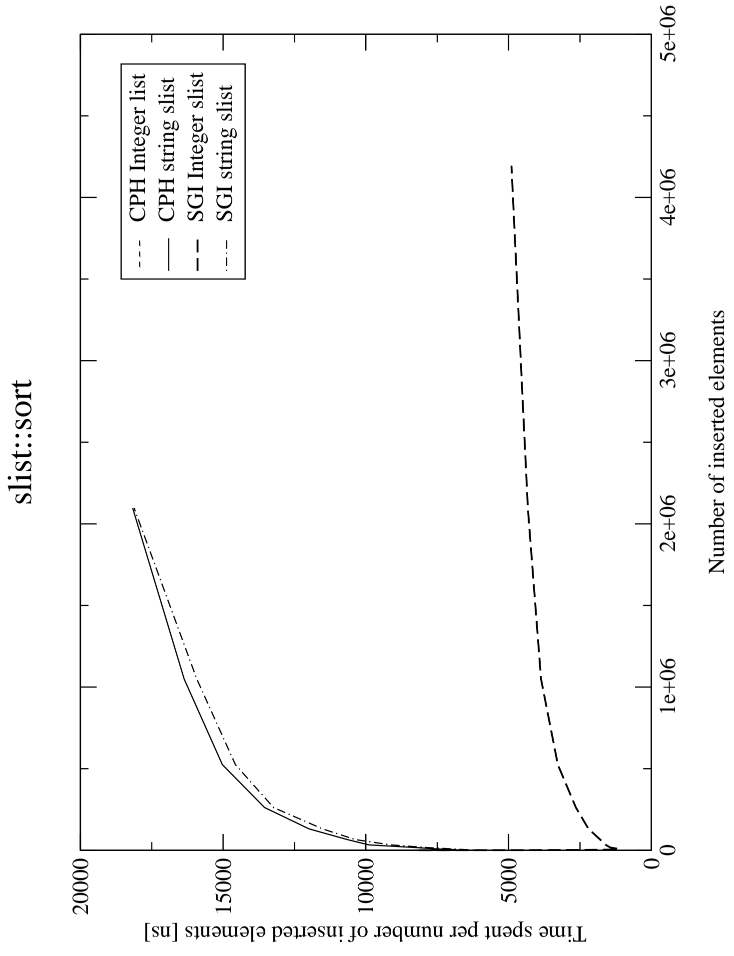

Figure 2.2. Performance test of slist::sort

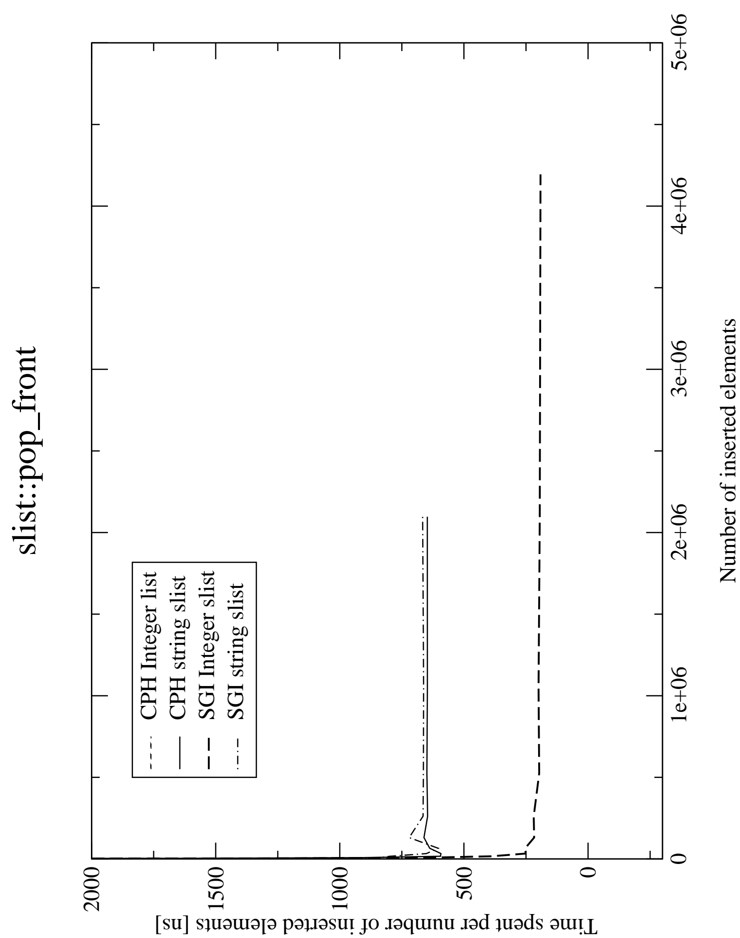

Figure 2.3. Performance test of slist::pop_front. Note, the data set for this plot was computed with an older version of pop_front() that does not check the tail pointer.

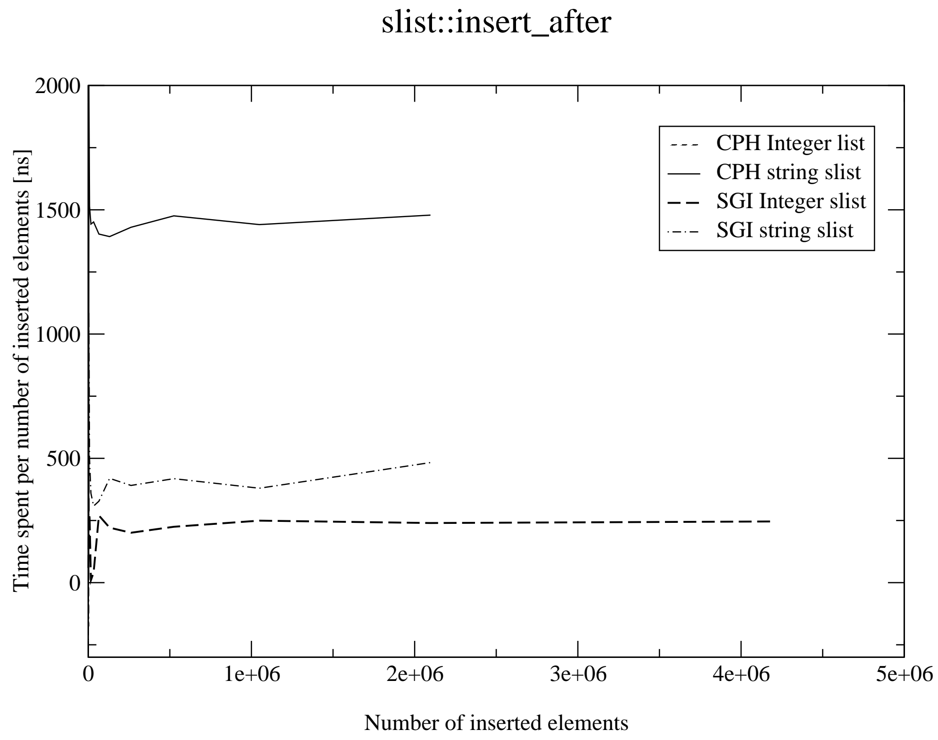

Figure 2.4. Performance test of slist::insert_after

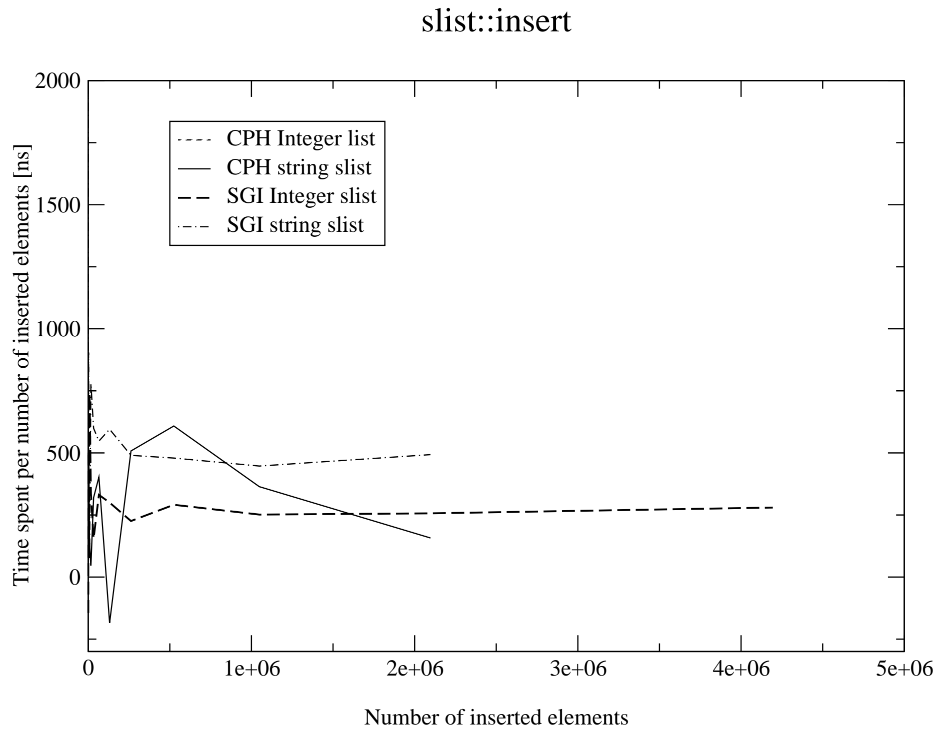

Figure 2.5. Performance test of slist::insert

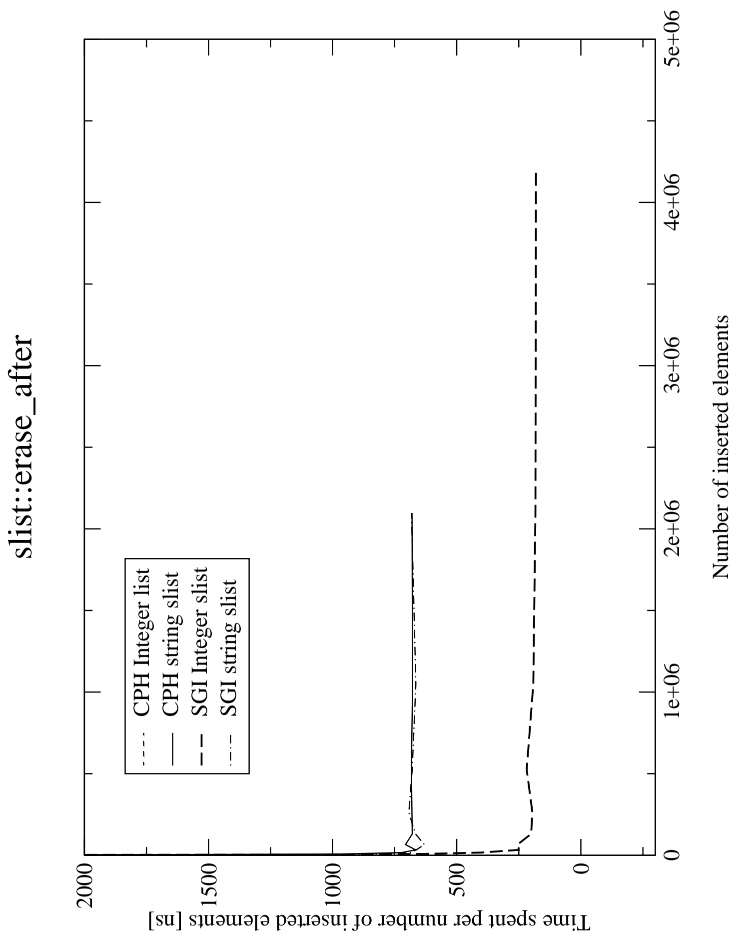

Figure 2.6. Performance test of slist::erase_after

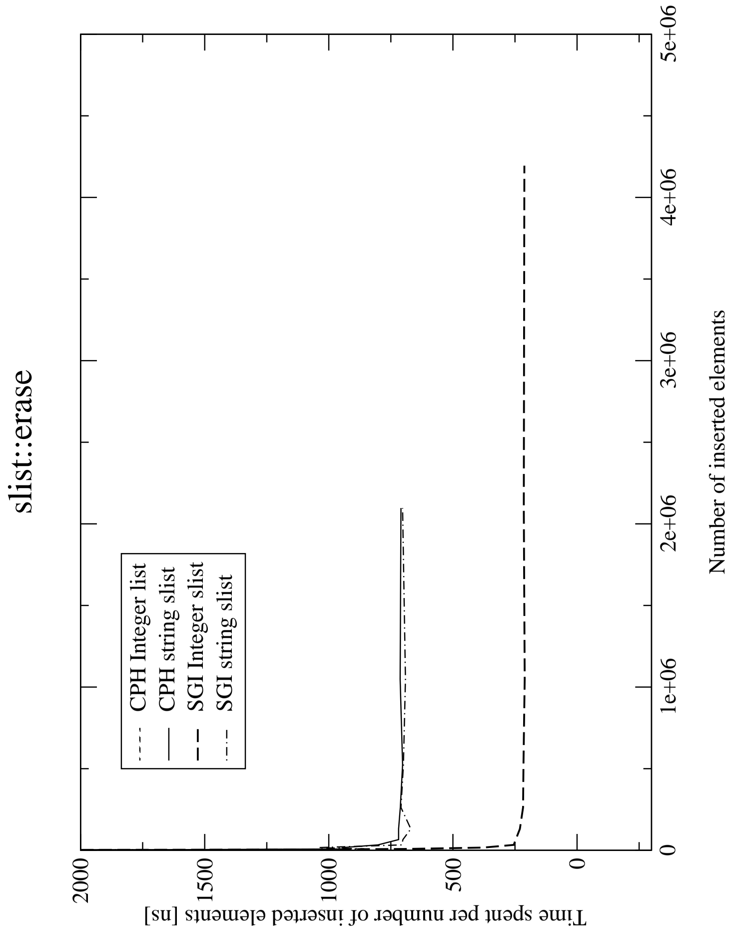

Figure 2.7. Performance test of slist::erase