

Implementation of a Cache-Oblivious Search Tree for the CPH STL*

Andrei Josephsen

August 2003

*CPH STL Report 2003-3, August 2003

Contents

1	Introduction	2
2	Design	2
3	The size of the static tree	3
3.1	Organizing keys and values	3
3.2	Navigating the main structure	3
3.3	Accessing the data	3
3.4	The address calculations	4
3.4.1	definitions	4
3.4.2	Navigation in a VEB layout	5
3.4.3	Observations	5
3.4.4	Locating the left subtree	5
3.4.5	Neccessary data	6
3.5	The Iterator	6
4	Implementation	7
4.1	Differences from <code>stl_map</code>	7
5	Benchmarks	7
5.1	Benchmark 1 and 2	7
5.2	Comparison with <code>rb_tree</code>	8
5.3	Benchmark 3	8
6	Conclusion	8
A	profile phase 1	10
B	profile phase 2	12
C	profile phase 3	14
D	profile of <code>stl_tree</code>	15
E	the source of the program	16

Abstract

In this report we describe an implementation of a cache-oblivious search tree as described by Brodal et al. in [1]. The goal was to benchmark this data structure against the red-black tree available at the SGI STL. The implementation part was successful, but the structure performs badly compared to red-black trees. Especially, the insertion and deletion operations are slow.

1 Introduction

Yet another goal was to make the interface close to that of the STL map. I do not think this can get much closer. Due to the nature of the data structure there are some details that are not perfect. For instance, the iterator class implemented is complex and it does not guarantee iterator validity.

Modern computers contain not just memory but hierarchies of memory each working as a cache of another. The hierarchy could be registers, L1 cache, L2 cache, main memory and disk. It is known that cache has a large impact on performance on various algorithms and often big speedups can be made by optimizing to a certain cache level.

Cache-oblivious algorithms are algorithms optimized to cache but without the algorithm knowing the size of the cache. This leads to efficient utilization of cache on any level and size and not just one. But why use a cache-oblivious algorithm instead of a cache-aware algorithm? In general, it is probably easier to optimize an algorithm by just looking at one or two levels of cache. Moreover, performance will probably be better by optimizing exactly for the size of a cache. However, implementing an algorithm as cache oblivious gives some advantages. It is a nice idea that it adapts to any cache size. It could be easy to implement. It is also interesting in its own right to study cache-oblivious algorithms because there might be a need for such algorithms whatever the reason might be. Even though cache-oblivious algorithms probably will not perform as well as cache-aware algorithms, but they might be comparable in speed and maybe worth an effort.

The data structure I am going to implement is the one described in [1]. In the article the structure is described in two parts. The first part is a static binary tree which is laid out in a cache-oblivious way. The way the nodes are located in memory is the so called Van Emde Boas layout (VEB). The layout is described in the design section. The second part is about how to embed a dynamic tree into a static tree and support insert, delete, find, and scan in an efficient way. Since it is embedded in a tree, there are no pointer updates and all rebalancing has to be done by moving elements around.

Other studies on cache-oblivious algorithms have been made and some show pessimistic results about their performance and others have some more positive results. The article considered here [1] shows some positive performance results, and the simplicity of the basic algorithms made me believe that it might be possible to create an efficient and general implementation. As we shall see, this was not the case for my implementation, but it might be possible to implement the structure more efficiently. With further optimizations in some cases this structure might be comparable in speed and maybe even outperform red-black trees. Since red-black trees are not cache optimized, the goal should really be to outperform them, but as we shall see, the dynamic tree embedded in the static tree slows down the insert operations.

As to the performance of the cache-oblivious search tree, Brodal et al. promise the following [1]: “For storing n elements, our proposal uses $(1 + \varepsilon)n$ times the element size of memory, and performs searches in worst case $O(\log_B n)$ memory transfers, updates in amortized $O(((\log_2 n)^2)/(\varepsilon B))$ memory transfers, and range queries in worst case $O(\log_B n + k/B)$ memory transfers, where k is the size of the output”.

2 Design

Since this is going to be implemented as part of CPHSTL it should meet some criteria from the C++ standard. Both on the interface and on performance. In some way I have ignored those

criteria and just focused on implementing the data structure and not changing on anything that would ruin the bounds shown in [1]. So maybe insert aren't as fast they should according to the ISO C++ standard but they are asymptotically within the bounds of [1].

One important goal of this report was to follow the interface of `stl_map`. But this was mainly due to unknown facts of `stl_map` and the implementation. Behind `stl_map` and some other structures is a red-black tree implementation `stl_tree`. I have chosen to implement a structure that mimics the `stl_tree` and thus will present the same functionality to `stl_map`. I have focused on implementing the functionality needed by `stl_map`, and not the complete `stl_tree` interface.

3 The size of the static tree

The article describes how to create static trees of any size and not just $2^n - 1$. But this step is not important for the speed of this algorithm but rather an optimization on memory utilisation and it will rather cost some speed. If the implementation proves to be fast enough i will consider implementing it. But this part is not important to prove the concept.

3.1 Organizing keys and values

Originally I wanted to separate keys and values. This would lead to better cache performance. But `stl_map` inserts (key,value) pairs and at first it seemed so much easier to just keep that idea. I do not think it would require that much work to change that later on. All the addressing and searching would be on the tree with keys and the values could just be stored in a regular array. It would only cost one extra lookup in the array to get the value. So I will keep in mind that changing this in the future could be a good idea.

3.2 Navigating the main structure

The original data structure proposes two data structures. One embedded into the other. The first data structure is a complete static tree in the Van Emde Boas layout and the second a general binary search tree. This lead to my choice of designing a general iterator for the Van Emde Boas layout and another one for the search tree. Since it's not really a iterator because it does not point to anything but just calculates addresses i call it a Navigator.

Splitting the Navigator out as a separate concept was also to make possible to experiment with just the navigation times. And maybe other experiments with the Van Emde Boas layout could be done on other structures. To access the embedded search tree and to fulfill the requirements of the map interface a general iterator is needed. So I have created this one. It is pretty straight forward. The main thing to keep in mind is how to identify empty elements and how to clear a node.

3.3 Accessing the data

Accessing data should be done through iterators. It is needed to conform to the `stl_tree` interface. To iterate through a search tree it is necessary to break up an inorder traversal in some way. The increment on the iterator is implemented in a rule based way. If a node `e` has a right child then

the next element is the leftmost node of that child. Otherwise it is located further up amongst the ancestors. The order of a binary tree implies that in this case the next key would be the first ancestor where e is located in the ancestors left subtree. This should work in general for any binary search tree.

Empty and Clear Since every node in the static tree is present we somehow has to mark those belonging to the embedded tree and those who are not in the tree. This is not discussed in [1]. Several possibilities exist and to be flexible i choose to implement two functions. `isEmpty()` to check if a node is marked empty and `clear()` to mark nodes as being empty(). Those thwo methods should be template arguments so that they could easily be changed but for a start i will just implement them as members.

One way to mark nodes as empty could be through an extra bit of information for each node. It could be through zero keys or zero values. By this i mean that there exist a key or value that is reserved to mark empty elements. This could be null pointers if the values are pointers. It could be the lowest possible key if in some way we could restrict the keys. If we had unique keys (like in `map`) then we could use duplicate keys to mark empty elements. In [1] elements are unique but i find no reason to keep this restriction. An empty node should have the same key as its parent.

I have chosen another general way that also works with non unique keys. I call it out of order elements. That is at heart, that if an element breaks the ordering of the tree then its should be considered as belonging to an empty node. There are some special cases for the maximum and minimum keys. I ignore those special cases which leads to a tree with unique maximum and minimum keys. This is a minor restriction I hope that most people can live with. It is possible to remove this restriction, but I will not spend any time on that. And for the `std::map` this does not matter since keys are unique. Now to identify if a node is a leaf we have to visit both of its children. This could lead to extra cache misses. Since identifying leaves is a important condition when traversing a tree, and identyfing existing branches we could choose to supply each node with information about its children. I have chosen not to do that because it could lead to unaligned keys and if i start to extend the nodes with extra information we could go on. Why not add pointers to the children. Maybe this could lead to better performance but thats not the structure proposed by [1].

3.4 The address calculations

To be able to navigate around in the static tree we have to understand the Van Emde Boas (VEB) layout. The purpose of the static binary tree is to provide a cache oblivious data structure in which we can embed another structure. We could look at this structure as being a mapping from breath first (BF) numbers to memory locations. A BF number for a node is also a binary version of its path and since every node is uniquely described by its path the BF number is in some way the natural number. The problem with BF numbers if we use them for addressing is that they grow exponentially as we travel down a tree. This give very bad locality for nodes on a path. Also it almost guarantees a cache miss for every level in the tree beyond a certain level. The idea of VEB layout is to improve locality of subtrees and from this follows a fair chance that a child of a node would be in cache already. Lets go into the details

3.4.1 definitions

We need some basic definitions.

- A tree is rooted and ordered.
- The depth of a node is the number of nodes on the simple path from the root to that node. The depth of the root is 1.
- The size of a tree is the number of nodes in the tree.

- A leaf is a node with no subtrees.
- The height of a tree is the depth of the deepest node.
- A complete tree is a tree containing $2^h - 1$ nodes, where h is the height.
- An embedded tree is a tree that can be transformed from another tree by pruning subtrees.
- A search tree is a tree containing keys from an ordered universe. Its ordered in such a way that every key in a left subtree is less than or equal to the key in the node and the keys in the right subtree are greater than. This definition is bit different from the definition in [1] to allow duplicate keys.

A tree T can be placed in an array in the Van Emde Boas Layout. The VEB layout of T is given by: If T is a leaf we just store T . Otherwise we cut the tree at height $h/2$ into the VEB subtrees T_0, \dots, T_n where T_0 is the tree that were located over the cut. The VEB layout for each tree T_0, \dots, T_n is stored after each other in the array. I will call the first upper VEB subtree T_0 a top tree.

This definition leads to a layout where a VEB sub tree of size K is located in a part of the array with the same size K . In other words it leads to locality of VEB subtrees.

3.4.2 Navigation in a VEB layout

For a binary tree T , it is essential to support the following operations.

- $\text{parent}(e)$, locate the parent of a node e
- $\text{left}(e)$, locate the left subtree from the node e
- $\text{right}(e)$, locate the right subtree from the node e

To locate a subtree from a node e in a VEB tree it is necessary to find the VEB sub tree where e is just above the next cut. In this tree we can calculate the position of the subtrees from e . First the subtrees are VEB subtrees. If we call the VEB subtree containing e above the cut-line T and the VEB subtrees that will be created by the cut T_0, \dots, T_n we just have to calculate the number of the subtree to find. Then the location of that subtree will be at the location of the root of T plus n times the size of the VEB subtrees $|T_1|$. The depth of a cut is defined as the height of the toptree.

3.4.3 Observations

To understand the calculations it is necessary with some observations.

1. If e is an element in a VEB tree then there is exactly one largest VEB subtree T where e is a leaf. This is quite natural since there exist a VEB subtree containing e since e is in fact a VEB subtree. The uniqueness follows by the fact that VEB sub trees at a certain level are disjoint and on different levels of very different sizes.
2. If e is an element in a VEB tree then there is a largest VEB subtree containing e as the root element. The argument is something similar to the previous.

3.4.4 Locating the left subtree

The location of the left VEB subtree L from e can be found by locating the largest VEB subtree T_{top} containing e as a leaf. Let n be the leaf number of e in T_{top} where the leafs are numbered from $-1, \dots, n$. Let $|T_{lower}|$ be the size of the largest VEB subtrees that follows after T_{top} . The position of L is then $T_{top} + |T_{top}| + 2 * n * |T_{lower}|$, where T_{top} is the position of that tree.

3.4.5 Necessary data

The following informations need to be known to locate the children of e .

- $\text{rootOfLeaf}(e)$ the position of the root of the largest VEB subtree containing e as a leaf.
- $\text{sizeOfTreeWithLeaf}(e)$ the size of the largest VEB subtree containing e as a leaf.
- $\text{sizeOfSubTreeFromLeaf}(e)$ the size of the largest VEB subtrees that are located below e .
- $\text{leafNumberOf}(e)$ The leaf number of e in the largest VEB subtree where e is a leaf.
- $\text{levelOf}(e)$ the depth of e . Not strictly necessary.

How do we calculate those numbers? There are some different possibilities.

1. If we presume that we would need an iterator this iterator could contain different parts of the data needed.
2. Another possibility would to recalculate all the data needed every time.
3. Maybe there exists a labeling scheme so that the addresses of the children could be extracted from the labels. It is not very likely that this would be much better than just pointers to the children.
4. Finally we could just store the positions of the children in the nodes.

I have chosen to use iterators containing as much data as necessary.

3.5 The Iterator

[1] suggest a strategy for finding the successor to a node. I could have followed this straight forward. But by analyzing on the structure i came up with a different strategy. It is basically the same strategy but with a another formulation. Early implementation experiments showed that there was not room for mistakes in these calculations. So i carefully thought this out once more to get better insights on the calculations. In this section the calculations leads to a design of the iterator.

To find the successor of a element we start by finding the location of the toptree T_{top} containing v as a leaf. We should be able to lookup this tree since we have already visited it on the path from the root to v . Now we lookup the size of that top-tree and the corresponding lower VEB subtrees. Those sizes only depend on the depth of v . Finally we need to know the position of v amongst the leaves in T_{top} . If we know the BFS number of v this number can be extracted from this by looking at the last bits of the BFS number. The location is: $\text{Location}[\text{Top}[v]] + \text{size}[\text{Top}[v]] + n * \text{size}[\text{Bottom}[v]]$. As i said this is basically the same calculation as in BRICS but a different explanation. There is a possibility that the last multiplication could be done by some clever shifting but i wont look into that. This leads us to save the following information:

- i , the BFS number.
- $\text{Pos}[d]$, the position of the node at depth d on the path.
- $\text{TopSize}[d]$ the size of the toptree that was created by a cut at depth d .
- $\text{BottomSize}[d]$, the size of the VEB subtrees that were created by a cut on depth d .
- $\text{D}[d]$ depth of the root of the top-tree created from a cut on depth d .
- $\text{TopSize}, \text{BottomSize}, \text{D}$ are static data structures that only needs to be calculated when we create the static tree.

- Pos[d] need to be build up during each traversal of a path.

The number n can be calculated by extracting the last d bits of i . This leads to some information stored in the iterators and some other information stored globally to minimize the size of the iterators. But still those iterators come with a price. Therefore unneccesarry copying of iterators should be avoided.

4 Implementation

The implementation was done in several stages. The first stage was to get a working none template version of a binary tree. This was to get some hands on the structure and to check that no logical laws were broken.

In second stage this was translated into a template structure. In the third stage the working template were transformed to obey the `sgi-stl-tree` interface. The first two steps were much easier two do than the third and final step. A lot of methods are required by the `stl_map` and many of those required big rewrites of the code. I think a fourth rewrite of the code would be a good idea to clean it up and optimize it a bit.

In the end i succeeded with the third step all though i did ignore some cases. I do not use the comparisson function so every key has to be ordered by the `j` operator. And i have not carefully checked that its the keys that are compared and not the (key,value) pair. But it works for the `stl_map` and thats enough for testing the performance.

A fourth cleanup stage is missing. But the code could use a careful cleanup and minor optimization phase. And even a fifth major optimization phase.

4.1 Differences from `stl_map`

Still some parts are missing. The iterator class should be rewritten to take the whole list of template arguments. The base class for `StaticTree` could be removed. `rightmost()` and `leftmost()` element should just be looked up. To make this implementation i had to change the keys to be non constant to allow copying and moving of keys.

The way the structure is build up leads to some differences from `stl_map`. The iterators are volatile. Otherwise i would have to track how keys are moved around in the tree for each iterator. Or other clever and probably time consuming solutions. I have just kept them as volatile.

The keys need to support some sort of increment and decrement operation. Right now its `+1` and `-1` but i think it should take an increment class as an argument so this could be generalized. The increment is needed to create out of order elements and the method should satisfy that $k < k + 1$ and $x > k - 1$ for every key maybe except for the maximum and minimum key.

All though there are differences i think i have proven it possible to implement the structure as a general template.

5 Benchmarks

I wanted to compare the structure with other structures. But the data structure is very slow. So rather than comparing operations on different sizes of trees i have chosen to profile the code to extract information on where all that time is spent. This leads to a few and very time consuming methods that should be optimized further if possible.

5.1 Benchmark 1 and 2

In my first finished phase implementation about 50 percent of the time is spent on calculating the size of a subtree. The `size` method is especially important during inserts and removes. The method is very simple and there is not much to do on that part. But the number of calls it very large. So I rewrote the `insert` method to avoid unnessecary size calculations. But still as we can

see in the next profile a lot of time is still spent on size calculations. Now we also see that the rightmost element is search for a lot. This could be precalculated at insert and removal time and should be.

5.2 Comparison with rb_tree

If we compare the profiles to the profiles generated using rb_trees we notice that only a very few methods are executed. Most of the time spend is on searches using lower_bound() and almost nothing on inserts and removes. The running time is extremely fast compared to my implementation. In less than a second it does what takes more than 200 seconds in my version. So at this point my implementation do not stand that strong.

Another thing I have noticed is than the time spend in benchmark one and two is on inserts and not on search. So next step is to isolate the running time for searches.

5.3 Benchmark 3

This time we look at 50.000 inserts and 50.000.000 searches. Now this looks much more promising. We cant compare this directly to the other benchmarks. But if we factor out that this one is doing five hundreds as many operations as the other benchmarks we look at a running time at about 8 seconds if we were only doing 100.000 searches. And still only about 20% of the running time was spend on doing actual searches. About 50% of the time is spend on finding the rightmost element and this should easily be optimized away. Now this result should get even better if we had a much higher number of keys. But unfortunately the slow speed of inserts makes it hard to test with for example 200.000.000 keys.

6 Conclusion

As we have seen the performance is rather poor. Especially if we look at inserts. I think one way to speed inserts up would be implement a fast insertion of already sorted lists. Then cache inserts in another tree the whole list at one time. But there is no way we are going to avoid the movements of elements around. If we look at the profiling information there is a lot of time spent on calculating the size of a subtree. This is necessary for the re-balancing scheme to work. But so much time is spent that i think that maybe sizes should be stored as extra information in the nodes.

If we look at searches they perform a lot better. And there are room for improvements. Unfortunately the speed of inserts makes it almost impossible to experiment with searches with very large sizes. Of course it could be done by creating a very big tree and then inserting in a way so that no movements are nessecary. But I still think that more work should be done on the inserts if this data structure should be useful in practice.

The iterators for this structure are rather big in size. If we somehow could avoid iterators I think we could gain a serious speedup. But for scans we really need those iterators. So we could reserve iterators for the scan operation. And maybe we could create a minimal but slower iterator that only contains information of the current element. This would probably be much slower on increment or decrement but maybe that is payed of by a general speedup.

Since we only have trees of 32 different sizes on a 32-bit machine we could easily write different code to manage each case. And for each tree three is a maximum of 32 levels so we could also write separate code to manage each level. This would lead to no more than 1024 cases. I think a lot of speedup could be gained from this approach but I have not had time to pursue it any further. Just a weak idea about creating tree templates for each height. Of course this does not lead to any bigger improvements on size calculations. But they should not be of that big importance if we have an insertion buffer. Second this will not help on the number of elements to be moved around. But we could hope that we would gain a speed up on moving the individual elements.

In this implementation there is a lot of room for improvement. I am pretty sure that it would be possible to equal performance of `stl_tree` by optimizing a lot and using insert buffers. I did not have the time to do all those improvements. There are still missing parts of the code that should be worked on. But all in all I think this still shows that this data structure could be of practical use. Especially if the majority of operations are searches.

Andrei Josephsen

References

- [1] Gert Stølting Brodal, Rolf Fagerberg, Riko Jacob. Cache Oblivious Search Tree. BRICS REPORT SERIES RS-01-36. October 2001
- [2] Jyrki Katajainen. Project proposal: the Copenhagen STL. `cphstl-report-2000-1`.
- [3] Silicon Graphics, Inc. Standard Template Library Programmer's Guide
<http://www.sgi.com/tech/stl/>

A profile phase 1

50.000 inserts and searches

Flat profile:

Each sample counts as 0.01 seconds.

%	cumulative	self	self	self	total	
time	seconds	seconds	calls	us/call	us/call	name
42.09	89.74	89.74	1197517	74.94	91.82	StaticTree::size()
19.70	131.73	41.99	1000609	41.96	49.56	StaticTree::remove()
8.54	149.93	18.20	346716913	0.05	0.07	StaticTree::isEmpty()
8.08	167.15	17.22	196865	87.47	156.18	StaticTree::insert()
4.83	177.44	10.29	26066444	0.39	0.52	StaticTree::setleaf()
3.31	184.50	7.06	2000000	3.53	4.81	StaticTree::rightmost()
2.57	189.98	5.48	397707876	0.01	0.01	STIterator::parentVal()
2.27	194.83	4.85	1000000	4.85	11.30	StaticTree::lower_bound()
1.67	198.39	3.56	93786843	0.04	0.04	Navigator::left()
1.49	201.57	3.18	93786843	0.03	0.03	Navigator::right()
1.08	203.88	2.31	500000	4.62	392.58	StaticTree::realInsert()
0.99	206.00	2.12	2394738	0.89	0.89	pair *__uninitialized_copy_aux ?
0.85	207.82	1.82	52187960	0.03	0.06	StaticTree::clear()
0.46	208.80	0.98	93786843	0.01	0.01	Navigator::up()
0.43	209.72	0.92	196865	4.67	258.93	StaticTree::remove()
0.30	210.37	0.65	196865	3.30	4.52	StaticTree::find()
0.30	211.00	0.63	2000000	0.32	5.13	StaticTree::rightmost()
0.22	211.47	0.47				main

% the percentage of the total running time of the program used by this function.

cumulative a running sum of the number of seconds accounted seconds for by this function and those listed above it.

self the number of seconds accounted for by this seconds function alone. This is the major sort for this listing.

calls the number of times this function was invoked, if this function is profiled, else blank.

self the average number of milliseconds spent in this ms/call function per call, if this function is profiled, else blank.

total the average number of milliseconds spent in this ms/call function and its descendents per call, if this function is profiled, else blank.

name the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

B profile phase 2

50.000 inserts and searches Insertion and search on 50000 elements

Flat profile:

Each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	us/call	us/call	name
28.43	42.28	42.28	1000609	42.25	49.63	StaticTree::remove()
27.04	82.50	40.22	1197473	33.59	38.12	StaticTree::size()
12.14	100.56	18.06	196865	91.74	159.82	StaticTree::insert()
8.44	113.11	12.55	242932119	0.05	0.06	StaticTree::isEmpty()
6.57	122.88	9.77	26066444	0.37	0.51	StaticTree::setleaf()
4.42	129.46	6.58	2000000	3.29	4.53	StaticTree::rightmost()
3.07	134.03	4.57	1000000	4.57	10.72	StaticTree::lower_bound()
2.41	137.62	3.59	293923170	0.01	0.01	STIterator::parentVal()
1.69	140.13	2.51	500000	5.02	265.16	StaticTree::realInsert()
1.55	142.44	2.31	52187960	0.04	0.07	StaticTree::clear()
1.39	144.50	2.06	2394738	0.86	0.86	pair* __uninzed_copy_aux?
0.69	145.52	1.02	196865	5.18	259.76	StaticTree::remove()
0.48	146.23	0.71	196865	3.61	4.78	StaticTree::find()
0.42	146.86	0.63	2000000	0.32	4.86	StaticTree::rightmost()
0.38	147.42	0.56				main

% the percentage of the total running time of the program used by this function.

cumulative a running sum of the number of seconds accounted seconds for by this function and those listed above it.

self the number of seconds accounted for by this seconds function alone. This is the major sort for this listing.

calls the number of times this function was invoked, if this function is profiled, else blank.

self the average number of milliseconds spent in this ms/call function per call, if this function is profiled, else blank.

total the average number of milliseconds spent in this ms/call function and its descendents per call, if this function is profiled, else blank.

name the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

Call graph (explanation follows)

C profile phase 3

50.000,000 searches

Flat profile:

Each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	us/call	us/call	name
35.33	1550.16	1550.16	1001000000	1.55	32.56	StaticTree::rightmost()
24.82	2639.29	1089.13	132213011	8.24	10.94	StaticTree::isEmpty()
18.31	3442.85	803.56	500500000	1.61	65.18	StaticTree::lower_bound()
8.24	3804.23	361.38	3978671358	0.09	0.09	STIterator::parentVal()
7.41	4129.32	325.09	1001000000	0.32	32.89	StaticTree::rightmost()
2.94	4258.36	129.04				main
0.95	4299.98	41.62	1000609	41.59	968.22	StaticTree::remove()
0.91	4339.94	39.96	1197473	33.37	809.45	StaticTree::size()
0.41	4357.99	18.05	196865	91.69	179.36	StaticTree::insert()
0.28	4370.49	12.50	1501999999	0.01	0.01	STIterator::STIterator()
0.22	4380.18	9.69	26066444	0.37	0.66	StaticTree::setleaf()
0.07	4383.05	2.87	500000	5.74	4275.60	StaticTree::realInsert()
0.05	4385.25	2.20	52187960	0.04	0.15	StaticTree::clear()
0.02	4386.20	0.95	196865	4.83	4992.51	StaticTree::remove()
0.02	4386.92	0.72	196865	3.66	204.88	StaticTree::find()

% the percentage of the total running time of the program used by this function.

cumulative a running sum of the number of seconds accounted for by this function and those listed above it.

self the number of seconds accounted for by this function alone. This is the major sort for this listing.

calls the number of times this function was invoked, if this function is profiled, else blank.

self the average number of milliseconds spent in this function per call, if this function is profiled, else blank.

total the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.

name the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

Call graph (explanation follows)

D profile of stl_tree

50.000 inserts and searches

Flat profile:

Each sample counts as 0.01 seconds.

%	cumulative	self	self	total		
time	seconds	seconds	calls	us/call	us/call	name
58.62	0.34	0.34	1000000	0.34	0.34	_Rb_tree::lower_bound()
15.52	0.43	0.09	1	90000.00	90000.00	_Rb_tree::_M_erase()
10.34	0.49	0.06	500000	0.12	0.12	_Rb_tree::_M_insert()
8.62	0.54	0.05	500000	0.10	0.22	_Rb_tree::insert_unique()
6.90	0.58	0.04				main

% the percentage of the total running time of the program used by this function.

cumulative a running sum of the number of seconds accounted for by this function and those listed above it.

self the number of seconds accounted for by this function alone. This is the major sort for this listing.

calls the number of times this function was invoked, if this function is profiled, else blank.

self the average number of milliseconds spent in this function per call, if this function is profiled, else blank.

total the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.

name the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

Call graph (explanation follows)

E the source of the program

```
/*
 * CPHSTL implementation of Cache Oblivious Search tree
 * by Andrei Josephsen, DIKU 2003
 *
 */

/*
 *
 * Copyright (c) 1996,1997
 * Silicon Graphics Computer Systems, Inc.
 *
 * 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 supporting documentation. Silicon Graphics makes no
 * representations about the suitability of this software for any
 * purpose. It is provided "as is" without express or implied warranty.
 *
 *
 * Copyright (c) 1994
 * Hewlett-Packard Company
 *
 * 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 supporting documentation. Hewlett-Packard Company makes no
 * representations about the suitability of this software for any
 * purpose. It is provided "as is" without express or implied warranty.
 *
 *
 */

/* NOTE: This is an internal header file, included by other STL headers.
 * You should not attempt to use it directly.
 */

#ifndef __SGI_STL_INTERNAL_STATIC_TREE_H
#define __SGI_STL_INTERNAL_STATIC_TREE_H

#include <stl_algobase.h>
#include <stl_alloc.h>
#include <stl_construct.h>
#include <stl_function.h>
#include <vector>
__STL_BEGIN_NAMESPACE

template <class _Value>
_Value operator+(_Value v,int i){
```

```

    return _Value(_Select1st<_Value>()(v)+i,_Select2nd<_Value>()(v));
}

template <class _Value>
_Value operator-(const _Value &v,int i){
    i=-i;
    return v+i;
}

#include "veb_tree.h"
#include "navigator.h"
#include "stl_tree.h"

template <class _Value, class _Ref, class _Ptr>
class STIterator; // StaticTreeIterator

template <class _Value>
class StaticTreeBase{
public:
    typedef _Value value_type;
    int H; // current max height;
    double d; // max density at root
    double ld; // min density at root
    double ldh; // min density at depth h
    value_type *data; //The static search tree
    int maxsize; //the maximum size of data
    int n; // number of elements in tree;
    // Iterator<T> it; // The navigator to navigate around the data;
    double density[32]; // density tresholds for each depth
    double ldensity[32]; //low density tresholds for each depth

    //precalculate density tresholds for each depth/level
    void initDensity(int H,double d,double ld,double ldh);
public:
    // StaticTree(int Height=1,double d=0.5,double ld=0.2);

    StaticTreeBase(int Height=1,double d=0.5,double ld=0.2)
        :H(Height),d(d),ld(ld),ldh(0),n(0)
    {
        maxsize=(1<<Height)+100;
        data=new value_type[maxsize];
        initDensity(H,d,ld,ldh);
    }
    void init(int Height=1,double d=0.5,double ld=0.2)
    {
        delete[] data;
        H=Height;
        this->d=d;
        this->ld=ld;
        ldh=0;
    }
}

```

```

        n=0;
        maxsize=(1<<Height)+100;
        data=new value_type[maxsize];
        initDensity(H,d,ld,ldh);
    }
    virtual bool isEmpty(STIterator<_Value,_Value&,_Value*> it);

};

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
          class _Alloc = __STL_DEFAULT_ALLOCATOR(_Value) >
class StaticTree : public StaticTreeBase<_Value>{
protected:
    _Compare key_compare;
public:
    _Compare key_comp() const { return key_compare;}
protected:
    _Alloc allocator;
public:
    typedef _Key key_type;
    typedef _Value value_type;
    typedef value_type* pointer;
    typedef const value_type* const_pointer;
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

    typedef STIterator<value_type, reference, pointer> iterator;
    typedef STIterator<value_type, const_reference, const_pointer>
        const_iterator;

    typedef reverse_iterator<const_iterator> const_reverse_iterator;
    typedef reverse_iterator<iterator> reverse_iterator;

    typedef _Alloc allocator_type;
    allocator_type get_allocator() const { return allocator_type(); }

    // _Compare key_compare;

private:
    iterator maxIt;
    key_type maxKey;
public:
    StaticTree(int Height=1,double d=0.5,double ld=0.2)
        :StaticTreeBase<value_type>(Height,d,ld)
    {

    }

    StaticTree(const _Compare& __comp, const allocator_type& __a)
        : key_compare(__comp),allocator(__a)

```

```

    { }

iterator end(){
    return rightmost().right();
}

int size();
int s();
double p();
friend class iterator;

iterator realInsert(value_type c);
iterator leftmost(iterator it);
iterator rightmost(iterator it);
iterator rightmost();
iterator root(){iterator start(this,H);return start.root();}
void checkLowerBalance(iterator it);

private:
public:
    iterator expand(value_type extra); // expand and insert extra
    void shrink(); // shrink the tree
    iterator find(const key_type& c);
    iterator findv(const value_type&v){
        return find(_KeyOfValue()(v));
    }
    iterator lower_bound(const key_type& c);
    void remove(iterator root,vector<value_type> &vect); //remove subtree into vect
    void remove(iterator root,vector<value_type> &vect,value_type extra);
    void insert(iterator emptyRoot,vector<value_type> &vect,int start,int n);
    void insert(vector<value_type> &vect);
    void rebalance(iterator it);
    iterator rebalance(iterator it,value_type c);
    void setleaf(iterator it,value_type c); // asuming it is empty
    int size(iterator r); // size of subtree rooted at r
    void remove(iterator it);
    bool isEmpty(iterator it);
    bool isLeaf(iterator it1);
    void clear(iterator it);
    double p(iterator v);
    int s(iterator v){ // the size of the complete subtree rooted at v
        return 1<<(H-v.getDepth()+1)-1;
    }

    void space(int n)
    {
        while(n-->0)
            cout << " ";
    }
}

```

```

void show(value_type x,int width){
    space(width/2);
    cout << x<<" ";
    space(width/2);
}

public:
void showTree(){
    cout << "Tree n:"<<n<<" H:"<<H<<endl;
    int width=4<<H;

    iterator it=root();

    vector<iterator > q;

    q.push_back(it);
    int i=0;
    int depth=0;
    while(i<q.size()&& depth<=H){

        iterator cur=q[i];
        if(depth<cur.getDepth()){
depth++;
width/=2;
cout<<endl;
space(H-cur.getDepth());
        }

        if(0 && isEmpty(cur)){
show(*cur,width);

        }
        else
        {
value_type stopval=*cur;
show(*cur,width);

if(cur.getDepth()<H){
    cur.left();
    q.push_back(cur);
    cur.up();
    cur.right();
    q.push_back(cur);
}

        }
        i++;
    }
    cout << endl;
}

```

```

public:
    pair<iterator,bool> insert_unique(const value_type& x){
        return pair<iterator,bool>(realInsert(x),true);
    }

    iterator insert_unique(iterator position,const value_type& x)
    {
        return realInsert(x);
    }

};

template <class _Key, class _Value, class _KeyOfValue,
          class _Compare, class _Alloc>
typename StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::iterator
StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::find(const _Key& k){
    iterator it=root();
    while(!isEmpty(it) && it.depth()<=H){
        if(k<_KeyOfValue>(*it))
            it.left();
        else if(k>_KeyOfValue(*it))
            it.right();
        else
            return it;
    }
    return end();
}

template <class _Key, class _Value, class _KeyOfValue,
          class _Compare, class _Alloc>
typename StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::iterator
StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::lower_bound(const _Key& k){

    if(n==0)
        return end();

    iterator guess=end();
    iterator it=root();

    while(it.depth()<=H &&!isEmpty(it) ){
        if(k<=_KeyOfValue(*it)){
            guess=it;
            it.left();
        }
        else{
            it.right();
        }
    }
    return guess;
}

```

```

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
         class _Alloc >
int StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::size(iterator r){
    if(r.depth()>H)
        return 0;
    if(isEmpty(r))
        return 0;
    if(r.getDepth()<H)
        return 1+size(r.left()+size(r.up().right());
    else
        return 1;
}

```

```

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
         class _Alloc >
void StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::
insert(vector<value_type> &vect){

    insert(root(),vect,0,vect.size());
}

```

```

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
         class _Alloc >
void StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::
insert(iterator emptyRoot,vector<value_type> &vect,int start,int n){
    if(n<3){
        setleaf(emptyRoot,vect[start+n/2]);
        if(n==2){
            emptyRoot.left();
            setleaf(emptyRoot,vect[start]);
        }
    }
    else{
        *emptyRoot=vect[start+n/2];

        insert(emptyRoot.left(),vect,start,n/2);
        emptyRoot.up();
        insert(emptyRoot.right(),vect,start+n/2+1,n-n/2-1);
        this->n++;
    }
}

```

```

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
         class _Alloc >

```

```

STIterator<_Value,_Value&,_Value*> StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::
rebalance(iterator it,value_type c){
    vector<value_type> vect; // a good guess of size would be the complete subtree size
    vect.reserve(1<<(H-it.depth()));
    remove(it,vect,c); // removed because we need to track the location of c
    // cout << "removed " <<vect.size()<<" elements ";

    // cout << "done removing... H is " <<H<<endl;
    insert(it,vect,0,vect.size());
    return findv(c);//realInsert(c);
}

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
          class _Alloc >
double StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::p(iterator v){
    return (double)size(v)/s(v);
}

template < class _Value >
bool StaticTreeBase<_Value>::isEmpty(STIterator<_Value,_Value&,_Value*> it)
{
    /*
    if(it.depth()==1)
        return !n;
    if(it.depth()>H)
        cout << "MUJAJA!!!"<<endl;
    if(it.navigate.isLeft()){
        // cout <<it.navigate<<" is left and "<<*it<<" > "<<it.parentVal()<<endl;
        return _KeyOfValue>(*it)>_KeyOfValue()(it.parentVal()); // check if its out of order
    }
    else{
        // cout <<it.navigate<<" is right and "<<*it<<" < "<<it.parentVal()<<endl;
        return _KeyOfValue>(*it)<_KeyOfValue()(it.parentVal());
    }
    */
}

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
          class _Alloc >
bool StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::isEmpty(iterator it)
{
    if(it.depth()==1)
        return !n;
    if(it.depth()>H)
        cout << "MUJAJA!!!"<<endl;
    if(it.navigate.isLeft()){
        // cout <<it.navigate<<" is left and "<<*it<<" > "<<it.parentVal()<<endl;
        return *it>it.parentVal(); // check if its out of order
    }
    else{
        // cout <<it.navigate<<" is right and "<<*it<<" < "<<it.parentVal()<<endl;

```

```

        return *it<it.parentVal();
    }
}

template <class _Value, class _Ref, class _Ptr>
class STIterator {
public:
    typedef _Value value_type;
    typedef _Ref reference;
    typedef _Ptr pointer;
    typedef STIterator<_Value, _Value&, _Value*> iterator;
    typedef STIterator<_Value, const _Value&, const _Value*> const_iterator;
    typedef STIterator<_Value, _Ref, _Ptr> _Self;

public:
    StaticTreeBase<value_type> *owner;
    Navigator navigate;
    STIterator(StaticTreeBase<value_type> *owner,int height);
    STIterator():navigate(1){}
public:

    value_type& parentVal();
    friend class StaticTreeBase<value_type>;
    value_type& operator*(){ return owner->data[navigate];}
    iterator& root(){navigate.root();return *this;}
    iterator& up(){navigate.up();return *this;}
    iterator& parent(){navigate.parent();return *this;}
    iterator& left(){navigate.left();return *this;}
    iterator& right(){navigate.right();return *this;}

    int getDepth(){return navigate.depth();}
    int depth(){return navigate.depth();}

    void increment();
    void decrement();

    _Self & operator++ ()
    {
        increment();
        return *this;
    }
    _Self operator++ (int)
    {
        _Self __tmp = *this;
        increment ();
        return __tmp;
    }

    _Self & operator-- ()
    {
        decrement ();
        return *this;
    }
}

```

```

_Self operator-- (int)
{
    _Self __tmp = *this;
    decrement ();
    return __tmp;
}

};

template <class _Value, class _Ref, class _Ptr>
inline bool operator==(const STIterator<_Value,_Ref,_Ptr>& __x,
                      const STIterator<_Value,_Ref,_Ptr>& __y) {
    return __x.navigate.getBFS() == __y.navigate.getBFS();
}

template <class _Value, class _Ref, class _Ptr>
inline bool operator!=(const STIterator<_Value,_Ref,_Ptr>& __x,
                      const STIterator<_Value,_Ref,_Ptr>& __y) {
    return __x.navigate.getBFS() != __y.navigate.getBFS();
}

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
          class _Alloc >
STIterator<_Value,_Value&,_Value*> StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::
realInsert(value_type c){//insert with rebalance if needed
    //cout << "trying to insert " << c << endl;
    //find pos for c
    //if depth is too deep... find tree to rebalance.
    //
    key_type k=_KeyOfValue()(c);

    if(H==1 && n){ //case tree with one occupied position
        //    cout << "expanding from H==1"<<endl;
        if(k>maxKey){
maxKey=k;
return maxIt=expand(c);
        }

        return expand(c);

        //cout << "done expanding and insert of c "<<<<" n "<<n<<endl;
        //return;
    }

    iterator it=root();

```

```

if(!n){ // realinsert at start.
    value_type v=c;
    (*it)=c;
    n++;
    if(H==1)
        return it;
    it.left();
    clear(it);
    it.up();
    it.right();
    clear(it);
    maxKey=k;
    return maxIt=it.parent();
}

int depth=1;
//start searching for empty pos. Return if not found

while(depth<H){
    if(c<*it){
        it.left();
    }
    else if(c>*it){
        it.right();
    }
    depth++;

    if(isEmpty(it)){//found empty position
        break;
    }
}

if(depth==H){
    if(isEmpty(it)){//empty leaf
        *it=c;
        n++;
        if(k>maxKey){
maxKey=k;
return maxIt=it;
        }
    }
    else{
        //find rebalance point and insert
        // p(it)=(double)size(it)/s(it);

        int size_it=size(it);
        double pit=double(size_it)/s(it);
        while(pit>=density[it.getDepth()]&& it.getDepth(>1){

```

```

if(it.navigate.isLeft()){// new size is size of left right sibling + size +1
    it.parent();
    size_it=size_it+1+size(it.right());
    it.parent();
}
else{
    it.parent();
    size_it=size_it+1+size(it.left());
    it.parent();
}
pit=double(size_it)/s(it);
    }
    //&& p(it)<=ldensity[it.getDepth()]);
    //check if resizing is needed
    if(it.getDepth()==1 && pit>=density[it.getDepth()]){
if(k>maxKey){
    maxKey=k;
    return maxIt=expand(c);
}
return expand(c);

    }
if(k>maxKey){
    maxKey=k;
    return maxIt=rebalance(it,c);
}
    return rebalance(it,c);

    }
}

else{//found an empty position

    *it=c;
    it.left();
    clear(it);
    it.up();
    it.right();
    clear(it);
    n++;
    if(k>maxKey){
maxKey=k;
maxIt=it.parent();
    }

    return it.parent();
}

}

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
        class _Alloc >

```

```

STIterator<_Value,_Value&,_Value*> StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::
expand(value_type elem){
    iterator start=root();
    vector<value_type> temp;
    temp.reserve(1<<H);
    remove(start,temp,elem);
    //StaticTree<value_type,_Value,_KeyOfValue,_Compare,_Alloc> newTree(H+1,d,ld);
    init(H+1,d,ld);
    // cout << "initialized new tree ... height is "<< H << endl;
    insert(temp);
    return findv(elem);;//realInsert(elem);
}

int f(){
    return 42;
}

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
          class _Alloc >
void StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::shrink(){
    //create new static search tree and insert everything from the old one.
    //first extract all elements into temporary array in sorted order
    //then insert all elements balanced into new array
    //the temporary could possibly be avoided by using empty positions
    //in new tree or by avoiding temp array and copying directly
    //this is almost the same as expand
    if(H==1) // dont shrink if height is one.
        return;
    // cout << "starting shrink operation"<<endl;
    //cout << "shrinking from height "<<H<<endl;
    vector<value_type> temp;
    remove(root(),temp);
    StaticTree newTree(H-1,d,ld);
    newTree.insert(temp);
    delete data;
    *this=newTree;
}

/*
removes subtree rooted at root into vect in inorder (sorted)
precondition: the subtree should contain atleast 1 element

final:
dosent actually change tree. ?
*/
template <class _Key, class _Value, class _KeyOfValue, class _Compare,
          class _Alloc >
void StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::
remove(iterator root,vector<value_type> &vect)
{
    if(isEmpty(root))
        return;

    if(root.getDepth()<H){ // if possible child

```

```

        remove(root.left(),vect);
        root.up();
        vect.push_back(*root);

        remove(root.right(),vect);
        n--;

    }
    else{
        vect.push_back(*root); // just remove key if at last level
        n--;
    }
}

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
          class _Alloc >
void StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::clear(iterator it){
    if(it.navigate.isLeft()){
        (*it)=(it.parentVal()+1); // create out of order value
    }
    else{
        (*it)=(it.parentVal()-1);
    }
}

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
          class _Alloc >
void StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::
setleaf(iterator it,value_type val){
    if(it.depth()>H)
        cout << "MUJAJFA"<<endl;
    *it=val;
    //cout <<"seting leaf to "<<val<<endl;
    if(it.getDepth()<H){
        it.left();
        clear(it);
        it.up();
        it.right();
        clear(it);
    }
    n++;
}

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
          class _Alloc >
void StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::
remove(iterator root,vector<value_type> &vect,
        value_type extra){

    if(isEmpty(root)){
        vect.push_back(extra);
        return;
    }
}

```

```

if(root.getDepth()<H){
    if(extra<*root){
        remove(root.left(),vect,extra);
        root.up();
        vect.push_back(*root);
        remove(root.right(),vect);
        n--;
    }else{
        remove(root.left(),vect);
        root.up();
        vect.push_back(*root);
        remove(root.right(),vect,extra);
        n--;
    }
}
else{

    if(extra<*root){
        vect.push_back(extra);
        vect.push_back(*root);

    }
    else
    {
vect.push_back(*root);
vect.push_back(extra);
    }
    n--;
}
}

template <class _Value, class _Ref, class _Ptr>
STIterator<_Value,_Ref,_Ptr>::STIterator<_Value,_Ref,_Ptr>( StaticTreeBase<value_type> *owner,in
:owner(owner),navigate(height)
{

}

template <class _Value, class _Ref, class _Ptr>
_Value& STIterator<_Value,_Ref,_Ptr>::parentVal(){
    return owner-> data[navigate.ppos(1)];
}

template <class _Value, class _Ref, class _Ptr>
void STIterator<_Value,_Ref,_Ptr>::decrement(){
    //check if it should go right
    if(navigate.depth()<owner->H){
        navigate.left();

```

```

        if(!owner->isEmpty(*this)){
            //find rightmost child
            while(navigate.depth()<owner->H){
navigate.right();
if(!owner->isEmpty(*this)){
    up();
    return;
}
        }
        return;
    }
    else
        navigate.up();
}
//otherwise... go up right and down right... and find leftmost
while(navigate.isLeft()){
    navigate.up();
}
navigate.up();
}

```

```

template <class _Value, class _Ref, class _Ptr>
void STIterator<_Value,_Ref,_Ptr>::increment(){
    //check if it should go right
    if(navigate.depth()<owner->H){
        navigate.right();
        if(!owner->isEmpty(*this)){
            //find leftmost child
            while(navigate.depth()<owner->H){
navigate.left();
if(owner->isEmpty(*this)){
    up();
    return;
}
        }
        return;
    }
    else
        navigate.up();
}
//otherwise... go up right and down right... and find leftmost
while(navigate.isRight()){
    navigate.up();
}
navigate.up();
}

```

```

template <class _Value>
void StaticTreeBase<_Value>::initDensity(int H,double d,double ld,double ldh){
    double delta=(1-d)/(H-1);
    for(int i=1;i<=H;i++){
        density[i]=d+(i-1)*delta;
    }
}

```

```

}

double delta=(ld-ldh)/(H-1);
for(int i=1;i<=H;i++)
    ldensity[i]=ld-(i-1)*delta;
}

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
          class _Alloc>
void StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::remove(iterator it){
    if(n==1){ // only root is present
        n--;
        return;
    }

    if(isLeaf(it)){
        //    cout << "clearing leaf "<<*it<<endl;
        clear(it);
        n--;
        checkLowerBalance(it.parent());
//    n--;
        return;
    }
    //    cout << "not leaf"<<endl;

    it.left();
    if(isEmpty(it)){ // no left subtree
        it.parent().right();
        iterator target=leftmost(it);
        it.parent();
        *it=*target;
        *it.left(); // remember to mark next node
        clear(it);
        remove(target);
    }
    else{
        //    cout << "removing from left subtree"<<endl;
        iterator target=rightmost(it);
        it.parent();
        if(isEmpty(it.right())){// no right subtree
            it.parent(); // double calculation of right !!!
            *it=*target;
            it.right();
            clear(it);
            remove(target);
        }
        else{
            it.parent(); // double calculation of right !!!
            *it=*target;
            it.right();
            remove(target);
        }
    }
}

```

```

    }
    //check if rebalancing or resizing is needed
}

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
          class _Alloc>
STIterator<_Value,_Value&,_Value*> StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::
leftmost(iterator it){
    while(it.depth()<H){
        it.left();
        if(isEmpty(it))
            return it.parent();
    }
    return it;
}

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
          class _Alloc >
STIterator<_Value,_Value&,_Value*> StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::
rightmost(iterator it){

    while(it.getDepth()<H){
        it.right();
        if(isEmpty(it))
            return it.parent();

    }

    return it;
}

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
          class _Alloc >
STIterator<_Value,_Value&,_Value*> StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::
rightmost(){
    iterator it=root();
    return rightmost(it);
}

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
          class _Alloc >
bool StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::isLeaf(iterator it1){
    if(it1.getDepth()==H)
        return true;
    iterator it2=it1; // optimize it2 away ? !!!
    return isEmpty(it1.left()) && isEmpty(it2.right());
}

```

```

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
         class _Alloc >
void StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::checkLowerBalance(iterator it){
    if(n<2)
        return;

    while(p(it)<=ldensity[it.getDepth()]&& it.getDepth(>1){
        it.parent();
    }

    if(it.getDepth()==1 && p(it)<=ldensity[it.getDepth()]&&H>1)
        shrink();
    else
        rebalance(it);

}

template <class _Key, class _Value, class _KeyOfValue, class _Compare,
         class _Alloc>
void StaticTree<_Key,_Value,_KeyOfValue,_Compare,_Alloc>::rebalance(iterator it){
    vector<value_type> vect; // a good guess of size would be the complete subtree size
    // cout << "rebalancing tree"<<endl;
    //showTree();
    remove(it,vect);
    insert(it,vect,0,vect.size());
    //cout << "done rebalancing tree"<<endl<<endl;
}

__STL_END_NAMESPACE

#endif /* __SGI_STL_INTERNAL_STATIC_TREE_H */

// Local Variables:
// mode:C++
// End:

```