

# Vector Implementation for the CPH STL

Mads D. Kristensen  
DIKU  
University of Copenhagen

CPH STL Report 2004-2, January 2004. Revised February 2004.

# Contents

<b>Contents</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
<b>2 The Datastructures in Theory</b>	<b>4</b>
2.1 Doubling Array . . . . .	4
2.2 Doubling Tree . . . . .	6
2.3 Hashed Array Tree . . . . .	8
2.3.1 Running Times . . . . .	8
2.3.2 Storage Efficiency . . . . .	9
2.4 Space Efficient Doubling Tree . . . . .	11
2.4.1 Storage Efficiency . . . . .	12
2.4.2 Running Times . . . . .	14
<b>3 Test of the Datastructures</b>	<b>16</b>
3.1 Test Setup . . . . .	16
3.2 The Individual Tests . . . . .	16
3.2.1 Lookup Performance . . . . .	17
3.2.2 <code>push_back</code> Performance . . . . .	19
3.2.3 <code>pop_back</code> Performance . . . . .	20
3.2.4 Scanning Performance . . . . .	22
<b>4 Conclusion</b>	<b>24</b>
<b>References</b>	<b>25</b>
<b>A Modifications to the CPH STL Benchmark Tool</b>	<b>26</b>
A.1 <code>measure_papi.c++</code> . . . . .	26
A.2 <code>measure_execution_time.c++</code> . . . . .	26
<b>B Source Code</b>	<b>26</b>
B.1 General Vector Source . . . . .	26
B.1.1 <code>vector.h</code> . . . . .	26
B.2 Doubling Array Source . . . . .	27
B.2.1 <code>doubling_array.h</code> . . . . .	27
B.3 Doubling Tree Source . . . . .	33
B.3.1 <code>doubling_tree.h</code> . . . . .	33
B.3.2 <code>dt_iterator.h</code> . . . . .	41
B.4 Hashed Array Tree Source . . . . .	43
B.4.1 <code>hashed_array_tree.h</code> . . . . .	43
B.4.2 <code>hat_iterator.h</code> . . . . .	56
B.5 Space Efficient Doubling Tree Source . . . . .	58

B.5.1	<code>se_doubling_array.h</code> . . . . .	58
B.5.2	<code>sedt_iterator.h</code> . . . . .	71
<b>C</b>	<b>Benchmark Scripts</b>	<b>73</b>
C.1	Lookup Tests . . . . .	73
C.1.1	<code>lookup_bench.cc</code> . . . . .	73
C.1.2	<code>lookup_benchmark.py</code> . . . . .	74
C.1.3	<code>lookup_benchmark_papi.py</code> . . . . .	75
C.2	<code>push_back</code> Test . . . . .	77
C.2.1	<code>push_back_bench.cc</code> . . . . .	77
C.2.2	<code>push_back_benchmark.py</code> . . . . .	78
C.3	<code>pop_back</code> Test . . . . .	79
C.3.1	<code>pop_back_bench.cc</code> . . . . .	79
C.3.2	<code>pop_back_benchmark.py</code> . . . . .	80
C.4	Scan Test . . . . .	81
C.4.1	<code>scan_bench.cc</code> . . . . .	81
C.4.2	<code>scan_benchmark_papi.py</code> . . . . .	82

# 1 Introduction

This report describes the implementation and test of a vector class in the *CPH STL* project. The vector class that is to be implemented will be a policy-based vector where it's possible to choose the storage model that should be used when instantiating a new vector.

The storage models for the vector that will be implemented and tested in this report are the following:

1. Doubling Array.
2. Doubling Tree.
3. Hashed Array Tree.
4. Space Efficient Doubling Tree.

The main goal of this report is to implement and benchmark these four different storage models to see how they perform in the three main vector operations which are lookups, insertion in the end of the vector and deletion of the last element in the vector.

The storage model used in SGI's STL is the doubling array and what I hope to show in this report is that there are other storage models that can perform just as well or even better than the doubling array. The incitement for this search for an alternative to the doubling array storage model is the fact that the doubling array has a storage efficiency of  $O(n)$  which is a big waste of space, in fact the worst-case storage usage in the doubling array is  $4n - 1$  where  $n$  is the number of element in the vector. There are other storage models, like the hashed array tree and the space efficient doubling tree, that have a storage efficiency of  $O(\sqrt{n})$  while still having the  $O(1)$  time bound on the three main vector operations. I would like to implement and test these storage models to see if they can perform as well as the doubling array in practice as well as in theory.

The different storage models will be presented in theory in section 2 and the benchmarks of them can be found in section 3.

## 2 The Datastructures in Theory

### 2.1 Doubling Array

The most commonly used method when implementing a vector is the doubling array. The technique is simple; you start of with an array of size  $n$  and when you insert the  $n + 1$ 'th element into the vector you simply allocate a new array of size  $2n$ , move the elements into this new array and deallocate the old array as can be seen in figure 1.

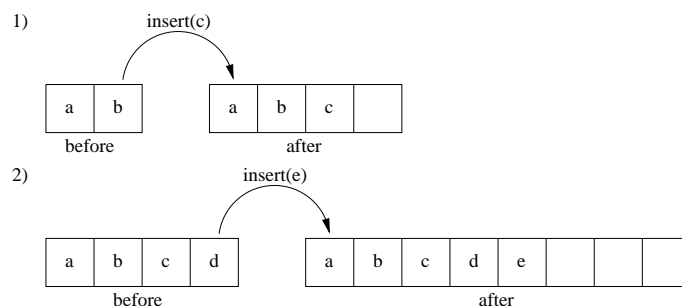


Figure 1: Two insertions into a vector using the doubling array technique.

It may seem like a terrible waste of space to double the size of the array every time more space is required but there is a good reason for choosing this method of resizing. If you choose to add a constant amount, say  $c$ , to the size of the array every time a resizing is needed you can make sure that only a constant amount of extra storage is used. But this upper bound on the amount of space used has dire consequences for the running time of the insert operations. If only a constant amount  $c$  of extra storage is allocated when resizing the array the running time of insert operations will be  $\frac{1}{c}n$ , which is  $O(n)$ , because for every  $c$  insertions the array needs to be resized and all  $n$  elements must be moved into the new array. If on the other hand you choose to use the doubling technique the running time of the insertion operations can be shown to run in  $O(1)$  amortized time. The few insert operations that are done on the boundaries of resizing will of course have to pay the  $O(n)$  time penalty for moving all the elements into the new array, but if you distribute the cost of these  $n$  element moves onto the  $\frac{n}{2}$  insertions that have to be done before a resizing is needed, you get an amortized running time of  $O(1)$  for the insertion operation. It's obvious that at least  $\frac{n}{2}$  insert operations will be run before a resize will be needed when the size of the array is doubled on every resize and the new array therefore has  $\frac{n}{2}$  free spaces that need to be filled before a resize is needed.

There is one other operation that needs to be efficient when designing a vector and that's the remove operation. This operation must remove the element in the tail of the array which of course can be done in  $O(1)$  time if no resizing of the array is necessary. In some implementations there are no

resizing at all when removing elements from the vector which means that there are no bound on the amount of space wasted when using such a vector. Like for insertions you have to chose the right time to resize the array when doing removals. If you make the array a constant amount  $c$  smaller every time you end up with a running time of  $\frac{1}{c}n$  which again is  $O(n)$ . This is of course not acceptable so you have to wait until the array is  $\frac{1}{4}$  full before you resize it. When the resizing is done the array is then resized to  $\frac{1}{2}$  of it's original size as is seen in figure 2.

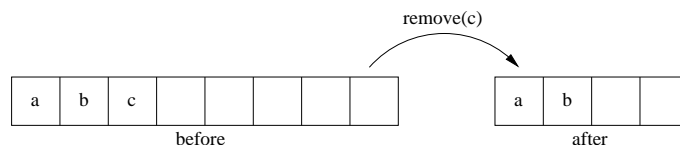


Figure 2: A removal from a vector using the doubling array technique.

This leads to a  $O(1)$  amortized running time because the cost of the boundary removals that have to pay for moving all  $n$  elements of the vector to a smaller array are paid for by all the other removals just like the case was for the insertions. If you choose to resize only when the array is  $\frac{1}{4}$  full and resize it to  $\frac{1}{2}$  of it's size there will always have been just as many removals before a resizing as there are elements left in vector that need to be moved.

To conclude on the doubling array technique the following specs can be listed, 1)  $O(1)$  amortized insertion and removal and 2) either unlimited or  $4n$  extra storage used depending on the resizing technique used when removing elements. In the rest of this paper whenever I refer to the doubling array technique the latter of the removal techniques will be used so that the doubling array as a whole has a  $O(n)$  storage usage.

The doubling array is very much used in practice and this is in fact the model that is implemented in SGI's STL version 3.3 but they use the model that doesn't resize when elements are removed from the vector. The doubling array model is used in many implementations for several reasons, 1) it's easy to implement and debug, 2) it features fast indexed access to the elements, 3) the fact that the storage for the vectors elements is allocated in one single block means that the elements are always lined up contiguous in memory which gives fast running times when scanning or sorting the vector because of the cache effect. The technique does however have some shortcomings like that fact that the time bound is  $O(1)$  amortized time and not just  $O(1)$  in general, which means that some insert or erase operations will take a long time to perform. There is also the problem with the large amount of memory that is potentially wasted when using this technique.

In the next subsections of this section some other methods of maintaining an efficient vector is presented which try to address these issues.

## 2.2 Doubling Tree

The doubling tree method<sup>1</sup> is very similar to the doubling array in the way that allocation of storage is handled. Again the amount of storage is doubled when new space is needed and halved when the vector becomes  $\frac{1}{4}$  full which means that  $4n$  or  $O(n)$  extra storage is still needed. The point that separates the doubling tree from the doubling array is the *way* that storage is allocated. Instead of allocating one single block of memory that has to contain all the elements of the vector the storage is allocated in blocks that are exponentially growing in size which can be seen as adding another level of leafs to a binary tree. The first level which is always allocated will be of size 1 and the next levels of sizes 2, 4, 8 and so on. This means that the amount of storage is  $2n + 1$  after a resizing. You will need to store pointers to these different layers of the tree but this will only require a constant amount of extra storage since only  $x$  pointers are needed on a  $x$ -bit architecture. This will allow  $2^x - 1$  elements to be inserted into the vector which also is the highest addressable index for an  $x$ -bit machine. The insertion of elements into a doubling tree can be seen in figure 3.

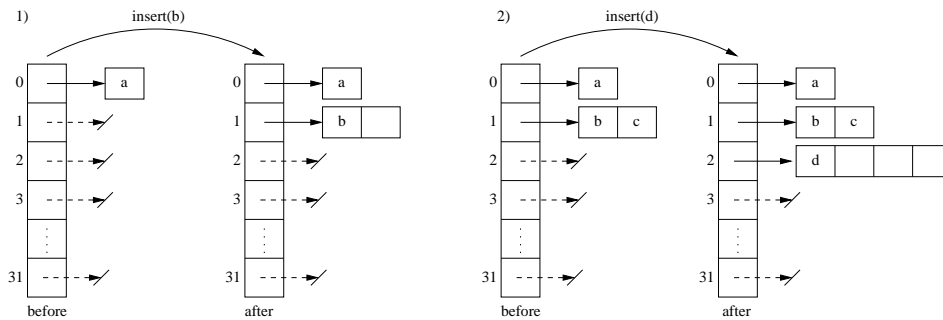


Figure 3: Two insertions into a vector using the doubling tree technique.

With this technique it's plain to see that insertions take  $O(1)$  time because no moving of elements is ever needed and we assume that memory allocation takes a constant amount of time no matter the size of the allocation<sup>2</sup>.

The resizing that is done when elements are removed from the vector is again done when the vector is  $\frac{1}{4}$  full. You could think that you might as well deallocate a block as soon as it's empty and in this way gain a  $2(n + 1)$  bound on the amount of storage. This would not affect the running time of the insertions or removals in theory since we assumed that memory allocations took constant time but in practice you could end up in cases where

<sup>1</sup>The name *doubling tree* is just one that I've made up myself since I don't know wether this technique has an *official* name.

<sup>2</sup>If memory allocation takes time linear with the size of the storage that is allocated insertion will still be amortized  $O(1)$ .

the same block of memory would continually be deallocated and allocated for every insert and remove operations executed. This will lead to terrible performance in practice and it is therefore decided to delete a block when the block behind it is empty. In this way only one empty block exists at any one time. The removal of elements is illustrated in figure 4.

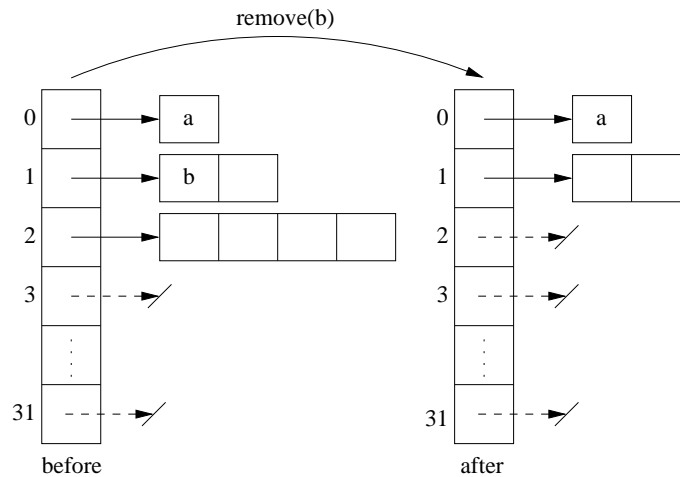


Figure 4: A removal from a vector using the doubling tree technique.

To sum up on the doubling tree technique we have the following properties, 1)  $O(1)$  insertion and removal of elements, 2)  $4n$  or  $O(n)$  extra storage needed and 3) the elements are never moved. Two desirable properties about the doubling array was the fast indexing times and the contiguous memory allocation. The indexing times are of course a bit slower when using a doubling tree but they are still a constant time operation. If you want to find element  $n$  in the vector the block that contains this element is number  $\lfloor \log_2(n+1) \rfloor$  (counting from 0) and the index into this block is  $n - (2^{\lfloor \log_2(n+1) \rfloor} - 1)$  (note that  $\lfloor \log_2(n+1) \rfloor$  is used twice and therefore only have to be calculated once). These operations can be implemented efficiently using simple shift operations and the indexing time is therefore still fast though of course not as fast as in the doubling array where no translation of indices are needed. The contiguous memory allocation property does not apply to the doubling tree as several blocks of memory are used. For a vector with 1.000.000 elements there are 20 of these blocks and within the blocks the elements are placed contiguous in memory. This means that an operation that run linearly through the entire vector might have 20 extra cache misses in a doubling tree than what would be the case for the doubling array. Wether this has any real impact on performance will be shown in the tests that I will conduct in section 3 where it will also be seen wether the doubling tree performs better than the doubling array for insertions and removals.

## 2.3 Hashed Array Tree

The *hashed array tree* (from now on referred to as *HAT*) datastructure is described in [4] by Edward Sitarski. As the name hints at the structure has some of the qualities of hashables, arrays and trees.

The basic idea of the HAT datastructure is the following: instead of just storing the elements in an array the elements are stored in a hashtable. This hashtable has a list of pointers to buckets called the *top-array*. This might sound like any other hashtable but that is where the tree properties comes into play with some rules defining which size the top-array and the buckets (which are called leafs in a hat) must have, and which sizes they should be resized to if an insert or erase operation requires a resizing. The sizing policy is the following:

1. The size of the top-array and the buckets must be the same and must be a power of 2, this is called the HAT's *leafsize*.
2. When a resizing is done the *leafsize* must be doubled or halved for growing or shrinking respectively.

The fact that the *leafsize* must always be a power of 2 makes it easy to find elements in a HAT through some simple operations. To find out which leaf contains a specific element with index  $n$  you just shift the index  $\log_2(\text{leafsize})$  positions to the right which gives the leafs index in the top-array, and to find the index of the element inside the leaf you can do a bitwise-AND of the index and a mask containing  $\log_2(\text{leafsize})$  1's in the rightmost bits. This will give the value of these rightmost bits which contain the index of the element inside the leaf. The bitmask that is needed for this operation and the information about which power of two the *leafsize* is, can be calculated once and for all when the HAT is resized. This means that you only need to perform one shift and one bitwise-AND operation to find an element in the HAT, which is obviously a constant time operation.

### 2.3.1 Running Times

The running time of insert operations is  $O(1)$  amortized time as was the case with the doubling array technique. When the HAT is resized the occupancy of the new HAT is only  $\frac{1}{4}$  which means that  $\frac{3}{4}$  elements will be inserted before a new resize is needed. This means that if you let the amortized cost of an insert operation be  $\frac{7}{3}$  the insert operations of the  $\frac{3}{4}$  elements will pay for the moving of the last  $\frac{1}{4}$  elements when a resizing is needed. The insertion of elements into a HAT can be seen in figure 5.

In [4] Sitarski describes the properties of a HAT only when insert operations are performed and the previous running time calculation was based on this perspective. If we need to allow erase operations in the structure we will see later that a HAT will be at  $\frac{1}{2}$  of it's maximal capacity after a shrink

operation, which means that we need to adjust our amortized costs a little. The new cost of an insert operation is now 3 which leaves 1 for performing the actual insert, 1 for moving the element when a resizing is needed and 1 for moving one of the elements that already reside in the HAT. This still leaves us with an amortized running time of  $O(1)$  though.

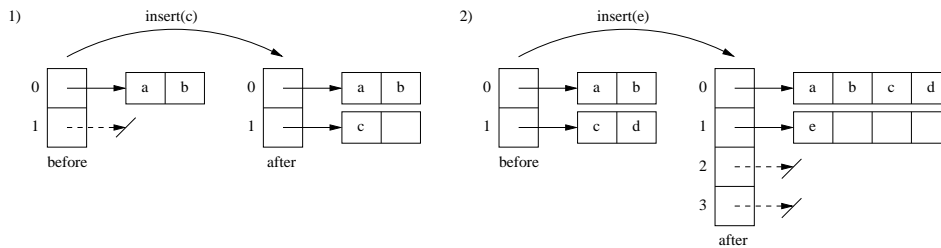


Figure 5: Two insertions into a vector using the HAT technique.

As was the case for the doubling array technique we also here need to determine when to resize the array when an erase operation is done, ie. when the HAT needs to shrink. If we want to preserve the  $O(1)$  amortized running time for erase operations we need to make sure that the resizing is done at the correct time, so that other erase operations have already paid for the moving of elements that is needed for the operation.

I have chosen to resize the HAT when it reaches  $\frac{1}{8}$  of its maximal capacity. Why this value has been chosen will be explained in the following. When a HAT grows in size the new capacity of the HAT is four times larger than the old capacity, which means that the HAT is  $\frac{1}{4}$  full after a resizing. If the only operations that are performed after the HAT has grown in size are erase operations, there will be performed  $\frac{1}{8}(\text{maximal capacity})$  erase operations before the HAT needs to shrink in size. By assigning an amortized cost of 2 to each erase operation this means that an erase pays 1 for erasing itself and 1 for moving one of the remaining  $\frac{1}{8}$  elements when the shrinking takes place. When the HAT shrinks its new maximal capacity is four times smaller than the old maximal capacity so the HAT will now be  $\frac{1}{2}$  of maximal capacity, as it was mentioned in the description of the running time for inserts. Erasing elements from a HAT can be seen in figure 6.

### 2.3.2 Storage Efficiency

In [4] Sitarski argues that the amount of extra storage needed in a HAT is  $O(\sqrt{n})$  provided that only insert operations are performed on the HAT. In the following argument I also take erase operations into consideration to compute the overall storage efficiency of the HAT structure.

At all times in the HAT structure the storage used for the top-array will be *wasted* storage, so *leafsize* pointers will always be wasted. When elements are added to a HAT new leaves are allocated one by one, which

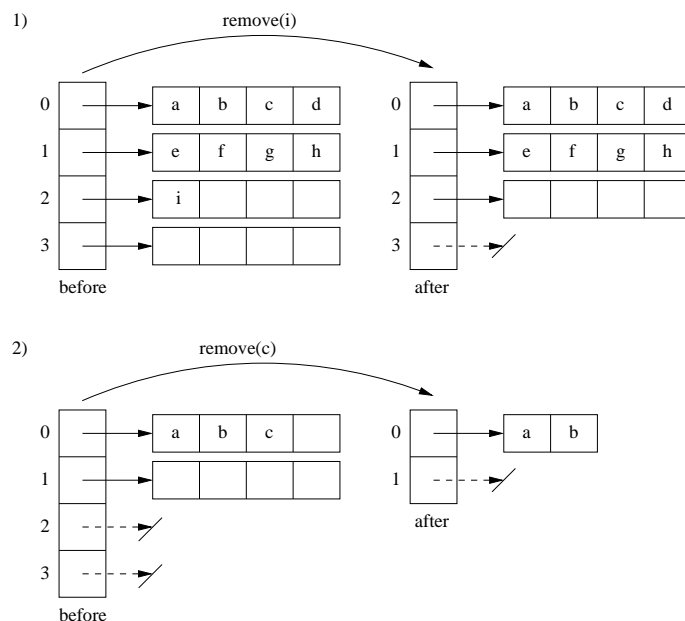


Figure 6: Two removals from a vector using the HAT technique.

means that at most  $leafsize - 1$  element positions are wasted at any one time when only inserts are done, which makes the total amount of wasted space  $2 \times leafsize - 1$ . To find the worst case for storage efficiency we need to look at the HAT right after a resizing has been done. After a resize the HAT contains  $\frac{1}{4} + 1$  elements and the last leaf only contains a single element<sup>3</sup> so the maximum of storage is wasted. The wasted storage in this scenario is the before mentioned  $2 \times leafsize - 1$  but we need a measure that is relative to the number of elements  $n$ .

Because both the top-array and the leafs must have the same size the maximum capacity of a HAT will always be  $leafsize^2$  which, because  $leafsize$  is a power of two, is a square of a power of two. We name this maximum capacity  $c$ . Expressed in terms of  $c$  the amount of waste is  $2\sqrt{c} - 1$ . As mentioned the HAT is  $\frac{1}{4}$  of full capacity when a resizing has been done, which means that  $n = \frac{1}{4}c + 1$  which in turn means that the waste expressed in terms of  $n$  is  $4\sqrt{n-1} - 1$ . The extra storage needed is thus  $O(\sqrt{n})$ .

When erase operations must be supported in the HAT structure the worst case for storage efficiency is just before the HAT shrinks in size. Just before this happens there are one empty leaf in the structure and one leaf that has only one element in it<sup>4</sup>. The storage waste is then  $3 \times \sqrt{c} - 1$ . Since  $n = \frac{1}{8}c + 1$  the total amount of waste expressed in terms of  $n$  is  $3\sqrt{8(n-1)} - 1$  which is still  $O(\sqrt{n})$ .

<sup>3</sup>This is true for all leafsizes greater than 2.

<sup>4</sup>Leafs are deallocated when two whole empty leafs exist in the structure.

In [4] Sitarski presents a table that shows the worst case amount of waste practically in a handful of different storage sizes. The table that is show in the article only take insert operations into account and as such does not match the results found in this section. Therefore I present another version of this table here:

Leafsize	Worst n	Worst waste	Percent of data
2	2	2	100
4	5	11	220
8	9	23	256
16	33	47	142
32	129	95	74
64	513	191	37
128	2049	383	19
256	8193	767	9

Table 1: Worst case storage overhead.

As can be seen from this table the amount of waste is negligible as the size of the  $n$  increases.

## 2.4 Space Efficient Doubling Tree

The space efficient doubling tree technique<sup>5</sup> is described by Brodnik et al. in [1]. The reason that I've chosen this name for it is because of the similarities of this datastructure and the doubling tree structure that I've described in section 2.2. In a doubling tree the space efficiency is  $O(n)$  because, as the name hints at, the storage is doubled every time a resizing is needed. In the space efficient version of a doubling tree the leafs are allocated in smaller pieces to make sure that the total amount of waste is within  $O(\sqrt{n})$ . That memory is allocated in smaller pieces means that the index table of the datastructure must contain more pointers to leafs, but as we will see shortly this can still be maintained in  $O(\sqrt{n})$  storage.

The main ideas of the datastructure are the same as for the doubling tree except for the allocation of memory. In this structure we have two types of leafs, *superleafs* and *subleafs*. Superleafs are the leafs that are allocated in the doubling tree datastructure. These leafs have sizes  $2^0, 2^1, 2^2, \dots, 2^i$  where  $i$  is  $\lfloor \log_2(\text{capacity}) \rfloor$ . In the doubling tree these superleafs are allocated in their entirety but this can not be done in the space efficient version, so instead of allocating the entire leaf we split the superleaf up into  $2^{\lfloor i/2 \rfloor}$  subleafs of size  $2^{\lfloor i/2 \rfloor}$  where  $i$  is the superleaf's number starting from 0. This means that a superleaf does not contain data but only pointers to the  $2^{\lfloor i/2 \rfloor}$

---

<sup>5</sup>This is not an official name but merely one that I use throughout this text.

subleaves that makes up this superleaf. This structure can be seen in the figure 7.

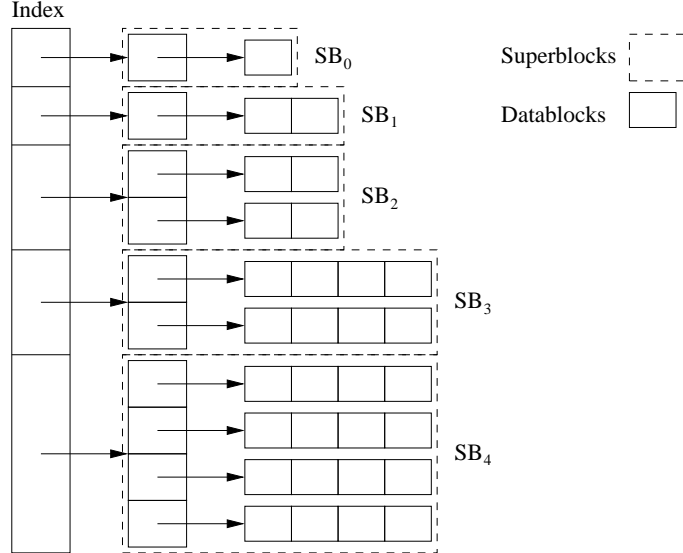


Figure 7: Overview of the space efficient doubling tree structure.

#### 2.4.1 Storage Efficiency

When discussing the storage efficiency of this datastructure we must take into account both the amount of wasted elementspace and the space used for the pointers to the superleaves and subleaves that are needed.

Like in the doubling tree datastructure the space required for the superleaf pointers is constant, since only  $x$  pointers will be needed in an  $x$ -bit architecture.

The overhead for keeping all the pointers to subleaves is still within the  $O(\sqrt{n})$  bound that we wish to hold for this structure. This is the case because at the time that the  $i$ 'th superleaf of size  $2^{\lfloor i/2 \rfloor}$  is allocated the structure contains  $n = 2^i - 1$  elements and the waste of  $2^{\lfloor i/2 \rfloor}$  pointers in the new superleaf is thus  $O(\sqrt{n})$ . Now we only need to account for the pointers to subleaves that already exist in the tree. Overall the number of pointers to subleaves in the tree before allocating the  $i$ 'th superleaf is

$$2^{\lfloor 0/2 \rfloor} + 2^{\lfloor 1/2 \rfloor} + 2^{\lfloor 2/2 \rfloor} + \dots + 2^{\lfloor (i-1)/2 \rfloor} = 2^{\lfloor (i-1)/2 \rfloor + 1} - 2$$

which is  $O(\sqrt{n})$ . The total amount of wasted space used on pointers is thus  $O(\sqrt{n})$ . The next thing that needs to be shown is that only  $O(\sqrt{n})$  elementspace is wasted when considering both insertions and removals.

When only considering insertions the worst case waste of elementspace is when allocating the first subleaf of a new superleaf. Before allocating the

first subleaf in a superleaf the size of the vector is as mentioned  $n = 2^i - 1$  which means that the amount of wasted storage of  $2^{\lceil i/2 \rceil} - 1$  elements in the subleaf after the insertion is  $O(\sqrt{n})$ . This means that the overall storage efficiency when inserting elements is  $O(\sqrt{n})$  since only one subleaf can be (almost) empty at any one time. The insertion of an element into the space efficient doubling array can be seen in figure 8.

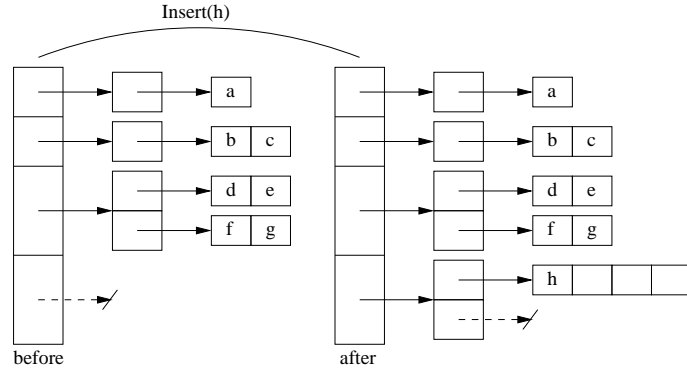


Figure 8: Insertion of an element into the space efficient doubling tree structure.

When removing elements from the vector we make sure to deallocate a subleaf every time two subleaves are empty and a superleaf as soon as it's empty. This means that a maximum of 2 (almost) empty subleaves of size  $O(\sqrt{n})$  can exist in the structure at any one time and yields storage efficiency of  $O(\sqrt{n})$  for removals in the tree. The removal of elements in the tree can be seen in figure 9.

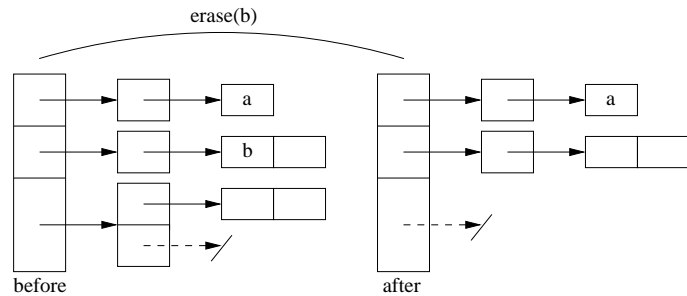


Figure 9: Removal of an element in the space efficient doubling tree structure.

As it has been show both the amount of storage used by pointers and the amount of wasted elementspace is within  $O(\sqrt{n})$  and this yields the final storage efficiency of  $O(\sqrt{n})$  for the space efficient doubling tree datastructure.

### 2.4.2 Running Times

The running time of lookup operations is  $O(1)$  as is demanded by the STL vector definition. Because of the 3-dimensional nature of the allocation scheme we need to find 3 indices into the structure to find a single element. These indices are found quite simply in the integer representation of the  $n + 1$  where  $n$  is the vector index that is requested:

1. The superleaf index is the position of the leading 1-bit which can be found in constant time with some simple bitwise shift operations. This can be done in constant time because an integer has a finite precision of  $x$  bits and thus a max of  $x$  shift operations will be needed.
2. The subleaf can be found in the  $\lfloor i/2 \rfloor$  bits of the integer representing  $(n + 1)$  where  $i$  is the superleaf index found in the previous step.
3. The index into the subleaf is the  $\lceil i/2 \rceil$  remaining bits of the integer representation of  $(n + 1)$ .

Step 1 is as mentioned a constant time operation and the remaining steps 2 and 3 are also constant time operations since they can be realized by a simple series of bitwise shift- and AND-operations. All in all this means that the lookup function in the tree can be done in  $O(1)$  as is required.

Insertion into the end of the tree is also a constant time operation which will be shown here. When you insert into the end of the tree there are only two cases to consider: 1) there is an empty space in the last subleaf and 2) a new subleaf and possibly a new superleaf must be allocated. In the first case the only thing we need to do is to find the index of the next free slot in the subleaf and copy the element into that slot. This is just a matter of doing a single lookup which has already been shown to run in  $O(1)$  time so this operation is clearly  $O(1)$ . In the other case we need to find the index of the new element and after that has been done we must allocate a new subleaf, and possibly a superleaf, both of size  $O(\sqrt{n})$ . If we ignore the cost of dynamic memory allocation this is still a constant time operation. If on the other hand we take the overhead of memory allocation into consideration the operation is still a constant time operation in amortized time, because since the last subleaf was allocated at least  $O(\sqrt{n})$  insert operations has been performed and the cost of  $O(\sqrt{n})$  to allocate the space has been paid for by these operations, which yields a  $O(1)$  amortized time bound.

Removing the last element from the tree must also be a constant time operation. When removing the last element from the tree you need to find this element which has been shown to be a constant time operation and then there are two cases: 1) the element can just be removed or 2) after removing the element we need to deallocate a subleaf and possibly a superleaf. The first case is obviously a constant time operation but the second case needs a little extra attention. If we, as in the insertion example, ignore

the cost of dynamic memory allocation the operation takes only the same amount of time as in the first case and is therefore  $O(1)$ . If we consider the memory deallocation overhead the operation is now an amortized constant time operation. This is because we only deallocate leafs when two subleafs are empty, which means that at least  $O(\sqrt{n})$  erase operations has been performed before the deallocation of leafs is done and these operations can thus pay for the deallocation yielding the final amortized  $O(1)$  time bound on removal of an element from the end of the tree.

## 3 Test of the Datastructures

### 3.1 Test Setup

All the tests were performed using a slightly modified version of the *CPH STL Benchmark Tool*[3]. The modifications to the tool was that I added some initialization and clean-up functions to the test that was not timed in the benchmark, a diff of the original files and the modified versions can be found in appendix A.

The tests were performed on a machine with the following specs:

CPU	AMD Athlon 700 MHz - 512 KB cache.
Memory	256 MB SD-RAM PC100
OS	Gentoo Linux system compiled with options “-O3 -march=athlon -funroll-loops -pipe”.
Kernel	2.4.19 with PAPI 2.4.5 extensions.

Table 2: Specs of the test machine.

The source code for the datastructures that are tested here can be seen in appendix B.

In the CPH STL Benchmark Tool the individual benchmarks are created as Python<sup>6</sup> scripts. The individual scripts used for the different tests can be seen in appendix C.

As it can be seen in the benchmark scripts I use both the execution time measurement capabilities and the cache-miss measurements done with the PAPI<sup>7</sup> interface.

### 3.2 The Individual Tests

The aspects of the different vectors performance that is of the greatest importance in this test is their performance in 1) inserting an element at the end of the vector (**push\_back**), 2) removing the element at the end of the vector (**pop\_back**) and 3) looking up an element at a specified index. Other operations as for example inserting or removing an element in the middle of the vector is of course supported but this is not what a vector is normally used for, and a structure based upon a double ended queue like the one described in [2] would perform better in this case. Therefore the test of these vectors will focus on only the before mentioned three operations.

As a special case of the indexing case it is also tested how the different datastructures perform when the vector is scanned linearly.

As we saw in section 2 the theoretical running time of **push\_back**, **pop\_back** and lookups is  $O(1)$  for all four datastructures. The questions that

---

<sup>6</sup><http://www.python.org>

<sup>7</sup><http://icl.cs.utk.edu/papi>

will be answered when testing these parts of the implementations is then 1) how big are the constants hidden in  $O(1)$  in the different datastructures, and 2) how is the cache performance of the different allocation schemes.

### 3.2.1 Lookup Performance

This benchmark was done with the scripts seen in appendix C.1. For lookups I have chosen to test both running time and how many cache misses was made when performing the operations. In both the running time and the cache miss test 10.000 lookups are performed in vectors of different size. Before each batch of 10.000 random lookup operations are performed a 1 MB array of integers are scanned to make sure that no elements of the vector are in the cache before the test is run.

The results of the running time test can be seen in figure 10.

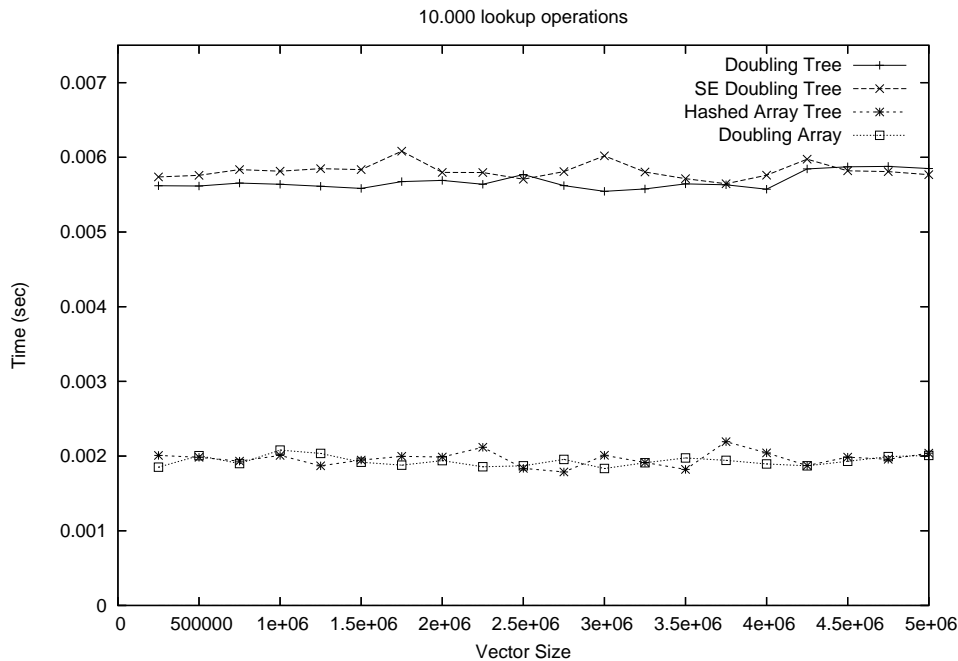


Figure 10: Lookup time measurement.

As could be expected the doubling array datastructure performed very well in this test, since no calculation of index is necessary with this datastructure and the elements are sequential in memory which gives a good cache performance.

It also comes as no surprise that the hashed array tree datastructure has a good performance in this test, although it's surprising to see exactly how well it performs. As can be seen in the figure the hashed array tree performs exactly as well as the doubling array. This is because the hashed array tree

has a good cache performance and the index computation consist of only a shift operation and a bitwise-AND operation.

The other two datastructures both perform about the same with the normal doubling tree being a small bit better that the space efficient variant most of the time. The reason that these two datastructures take almost three times longer to perform a lookup than the hashed array tree and the doubling array must be the cost of calculation the index. In both doubling trees the index calculation needs a  $\lfloor \log_2 \rfloor$  operation to find the index of the superleaf and this operation takes up most of the time spend calculating the index.

The results of the cache miss test can be seen in figure 11.

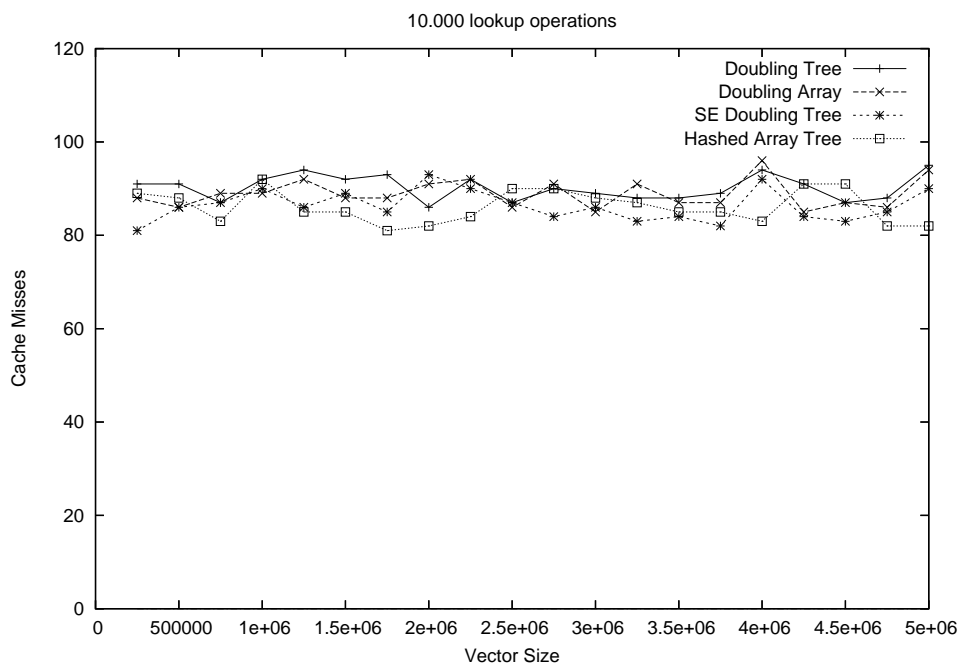


Figure 11: Lookup cache miss measurement.

This figure shows that all the four datastructures have a decent cache performance since all of them has less than 1% cache misses over the 10.000 lookup operations.

In a more realistic test environment the cache performance of the doubling array would probably be better than the other structures because the data is stored sequentially in memory. In the tests run here the vector is filled in one loop which means that even though the memory blocks are dynamically allocated in the datastructures chances are that the data will be sequential in memory. This will not affect the doubling array whilst giving a better cache performance to the three other datastructures.

### 3.2.2 `push_back` Performance

This benchmark was done with the scripts seen in appendix C.2.

There are two main aspects that play a part in the performance of `push_back` for the different datastructures, 1) the time spend finding the next available slot in the vector and 2) the time spend reorganizing the data if a resize of the internal structure is necessary.

For the doubling array datastructure part 1 is close to nothing since no recomputation of index is necessary but the resizing of the datastructure takes a lot of time since all elements must always be moved. For the hashed array tree part 1 is also negligible because it only consists of a shift and a bitwise-AND operation. The resizing of the datastructure on the other hand might take some time, but in the most cases resizing in the hashed array tree just means allocating another leaf and only once in a while the leafsize must change which means that all elements must be moved. This means that fewer elements are moved than in the doubling array and the cost of finding the next free space is very low so this datastructure might perform better for `push_back` than the doubling array.

The two doubling tree datastructures both use a  $\lfloor \log_2 \rfloor$  operation to determine the superleaf that the new element must reside in. Compared to the other two datastructures this is an expensive operation and it means that part 1 is much more expensive for these datastructures. The cost of resizing these two datastructures are on the other hand very small since no elements needs to be moved at any time. Whenever more storage is needed the doubling tree just allocates another superleaf and the space efficient version just allocates another subleaf.

The results of the `push_back` test can be seen in figure 12.

As can be seen from this figure the test winner is the hashed array tree. The fast lookup time and the fewer element moves of the hashed array tree makes it a lot faster than the simple doubling array.

The doubling tree also seems to be a bit faster than the doubling array. It is clear from the figure that the lookup time of the doubling tree is higher than that of the doubling array which is why the two takes turns in being the fastest one. The doubling tree overtakes the doubling array in the places where a resizing is needed, which is most evident in the figure around the 1.000.000'th operation. Here the doubling array has to move over one million elements while the doubling tree can just allocate another leaf and keep going. The doubling tree does show a small increase in running time when a doubling is done which is due to the construction of the empty elementspace in the new leaf. This increase is negligible though in comparison to the increase in cost experienced by the doubling array.

The test loser is definitely the space efficient variant of the doubling tree. This datastructure takes almost twice as long as any of the others to insert elements. This increase in running time compared to the normal

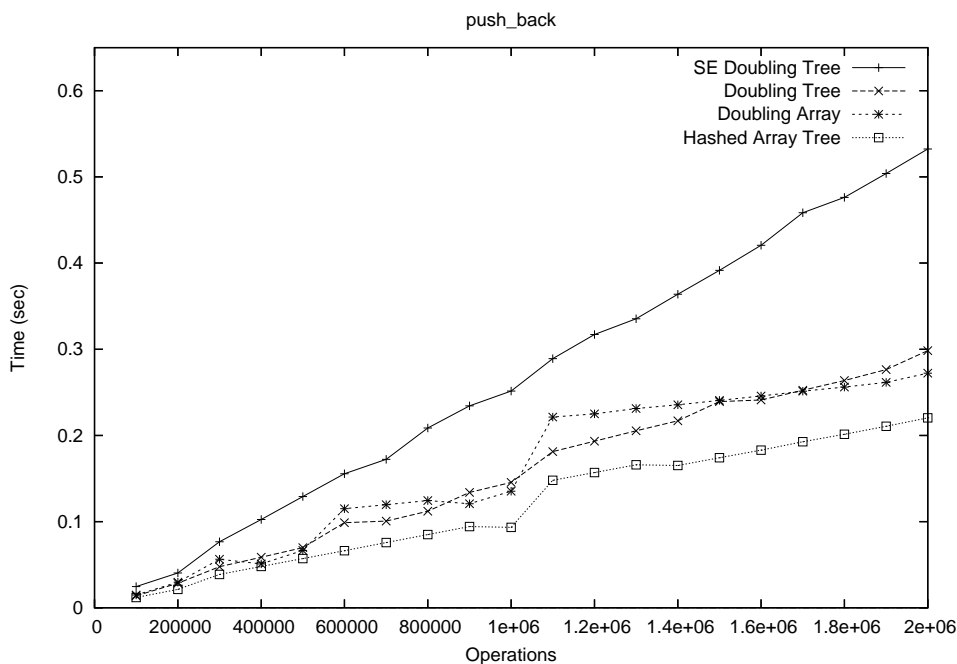


Figure 12: `push_back` time measurements.

doubling tree must be due to the many operations done when new memory must be allocated which is quite often considering the many small subleaves that the datastructure consists of. The cost to find the next free space in the datastructure is also a bit larger than for the doubling tree.

The running times of both the doubling trees could be improved upon by optimizing the  $\lfloor \log_2 \rfloor$  operation performed when finding the next free space<sup>8</sup>. Brodnik et al. mentions in [1] that newer Pentium processors can do this calculation in a single instruction which would greatly speed up the process. Another method would be to create a lookup function that found the value by a binary search in known values. This would however demand that the wordlength of the architecture be known. Both optimizations would destroy the portability of the code and has therefore not been implemented.

### 3.2.3 `pop_back` Performance

This benchmark was done with the scripts seen in appendix C.3.

For `pop_back` the elements of importance for the running time are the same as for `push_back` which were 1) the time spend finding the element to erase and 2) the time spend resizing the datastructure.

The times spend finding an element has already been discussed so the important thing here is how the individual datastructures perform when

<sup>8</sup>This would also improve the lookup times for the two datastructures.

performing shrinking. For shrinking the hashed array tree again has an advantage over the doubling array since it shrinks when it's at  $\frac{1}{8}$  of full capacity and not  $\frac{1}{4}$  like the doubling array. That, and the fact that the maximum capacity of the HAT grows faster than in the doubling array, means that fewer element moves are done in the hashed array tree and it can therefore be expected to perform better than the doubling array.

The performance of the two doubling trees can be expected to be the same as for `push_back` since the main part of the time will be spend finding the element to erase and, in the space efficient variants case, deallocating the many small subleafs.

The results of this test can be seen in figure 13.

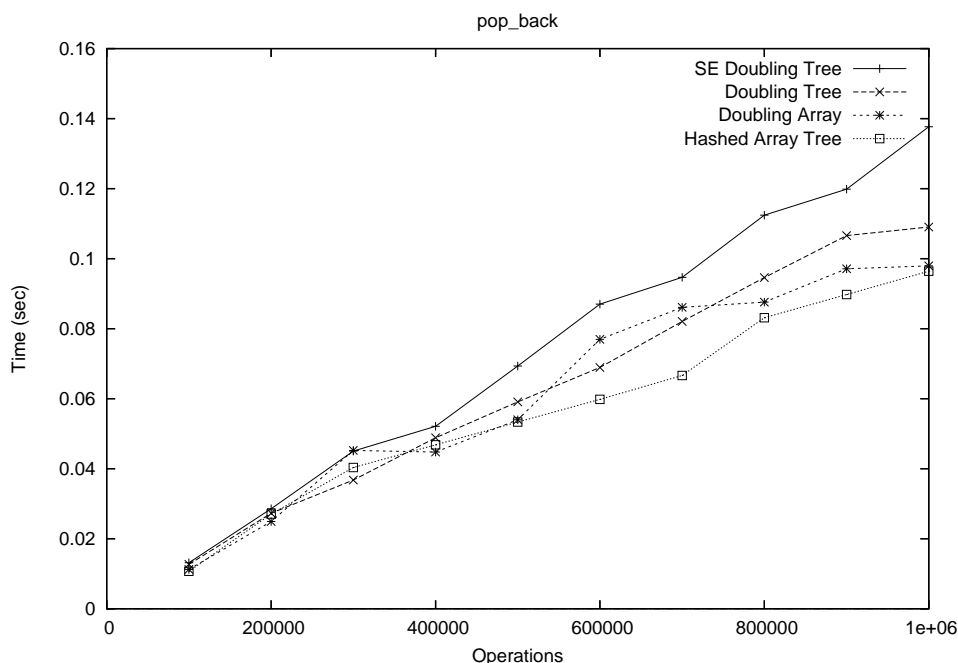


Figure 13: `pop_back` time measurements.

Considering the results of the `push_back` benchmark the results in this figure are what could be expected. The best performing datastructure is the hashed array tree because of the few element moves and the short lookup time.

Also like in the `push_back` test we see that the doubling array and the doubling tree take turns in being the fastest. This is for the same reason as mentioned before.

Again the space efficient doubling tree is the slowest of the four datastructures. Like in the `push_back` benchmark this must be accredited to the overhead in performing memory deallocation operations.

### 3.2.4 Scanning Performance

This benchmark was done with the scripts seen in appendix C.4.

The interesting thing to test when doing a full scan of a vector is to see how many cache misses this incurs in the different structures. For the doubling array that has all elements sequential in memory I would expect almost no cache misses. The other datastructures have their elements more or less spread out in memory and might therefore have a higher cache miss rate.

The results of this test can be seen in figure 14.

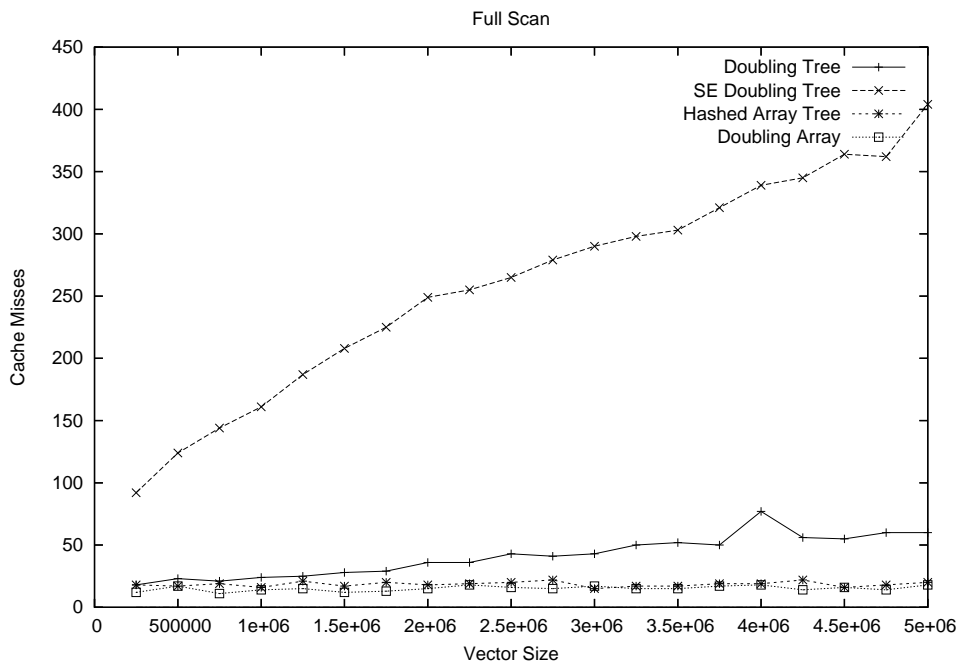


Figure 14: Full scan cache miss measurements.

As expected the doubling array has very few cache misses and is as such the best datastructure in this benchmark.

The hashed array tree also has a very good cache performance and is almost as good as the doubling array.

The doubling tree has slightly worse cache performance than the hashed array tree. This is probably the case because the doubling tree allocates larger and larger chunks of memory at the time as opposed to the hashed array tree which always allocates memory chunks of the same size. By allocating larger and larger blocks the doubling tree increases the odds that there won't be available space to allocate the individual leaves sequentially in memory and each leaf boundary thus gives cache misses.

For the space efficient doubling tree the cache performance is a lot worse

than for any of the other datastructures. This is because this datastructure allocates a lot of small chunks of memory and it can't be sure to get these chunks in a sequential order. This means that a lot of cache misses are incurred on the many subleaf boundaries and leads to the poor cache performance seen here.

In a more realistic test setup where data has not just been inserted into the vector in one go the advantage of the doubling array and the hashed array tree would probably be more visible. You could also suppose that the doubling array would perform a lot better than the hashed array tree in this case because the hashed array tree consists of several non-sequential chunks of memory. This might or might not be the case because every time the hashed array tree grows or shrinks all it's leafs will be allocated at once which gives it a good chance of having it's leafs placed sequentially in memory. This will keep the cache miss rate of the hashed array quite low but the doubling array will still be the best of the two.

## 4 Conclusion

Based upon the benchmarks done on the four different datastructures the choice of which should be the default in the CPH STL vector is easy to make. The hashed array tree performs just as well as the doubling array in the lookup benchmark and is clearly the best in the `push_back` and `pop_back` benchmarks. The hashed array table also has a very good cache performance as was shown in the scan benchmark. All this, and the fact that it has a storage efficiency of  $O(\sqrt{n})$  compared to the storage efficiency of  $O(n)$  for the doubling array, means that the hashed array tree has been picked as the default in the CPH STL vector, and I have thus succeeded in showing that there are better storage models that are more efficient both in time and space than the doubling array when implementing a vector.

There is one thing that hasn't been tested in the four datastructures and that is the general `insert` and `erase` functions that insert or delete an element in any position of the vector. These functions will perform better for the doubling array than any of the other datastructures because the moving of elements will be a lot easier in this simple structure. If you make extensive use of these functions and storage efficiency is not important to you your choice should be the doubling array. A more efficient datastructure when considering `insert` and `erase` operations is the tiered vector described by Goodrich and Kloss in [2]. How well this datastructure performs in lookup, `push_back` and `pop_back`, I don't know, but this would surely be an interesting benchmark to perform.

## References

- [1] Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Workshop on Algorithms and Data Structures*, pages 37–48, 1999.
- [2] Michael T. Goodrich and John G. Kloss II. Tiered vectors: Efficient dynamic arrays for rank-based sequences. In *Workshop on Algorithms and Data Structures*, pages 205–216, 1999.
- [3] Jyrki Katajainen. <http://www.cphstl.dk/www/benchmark.php>.
- [4] Edward Sitarski. Algorithm alley: Hats: Hashed array trees. *Dr. Dobb's Journal*, 1996.

## A Modifications to the CPH STL Benchmark Tool

### A.1 measure\_papi.cpp

```
31a32
> e.primal_init();
41c42
<
----
> e.primal_clean();
```

### A.2 measure\_execution\_time.cpp

```
79c79
< #define MEASURE(running_time, function) { \
----
> #define MEASURE(running_time, function, function_init, function_clean) { \
83a84
>     function_init(); \
92a94
>     function_clean(); \
114a117,122
>         std::clock_t init_start; \
>         std::clock_t init_stop; \
>         std::clock_t clean_start; \
>         std::clock_t clean_stop; \
>         std::clock_t init_time = 0; \
>         std::clock_t clean_time = 0; \
117c125,133
<         function(); \
----
>         init_start = std::clock(); \
>         function_init(); \
>         init_stop = std::clock(); \
>         init_time += (init_stop - init_start); \
>         function(); \
>         clean_start = std::clock(); \
>         function_clean(); \
>         clean_stop = std::clock(); \
>         clean_time += (clean_stop - clean_start); \
118a135
>         clock_calls += 4; \
128c145
<         (double(stop - start) - (double(clock_calls) * overhead)) / double(calls); \
----
>         (double(stop - start) - init_time - clean_time - \
>         (double(clock_calls) * overhead)) / double(calls); \
156c173
<     MEASURE(running_time, e.primal)
----
>     MEASURE(running_time, e.primal, e.primal_init, e.primal_clean)
163c180
<     MEASURE(running_time, e.dual)
----
>     MEASURE(running_time, e.dual, e.dual_init, e.dual_clean)
```

## B Source Code

### B.1 General Vector Source

#### B.1.1 vector.h

```

#ifndef VECTOR_H
#define VECTOR_H

namespace cphstl {
  template <class T, class A = std::allocator<T>,
           class S = hashed_array_tree<A> >
    class vector : public S {
    public:
      typedef typename A::size_type size_type;
      typedef typename A::difference_type difference_type;
      typedef typename A::value_type value_type;
      typedef typename A::pointer pointer;
      typedef typename A::const_pointer const_pointer;
      typedef typename A::reference reference;
      typedef typename A::const_reference const_reference;
      typedef typename S::iterator iterator;
      typedef typename S::const_iterator const_iterator;

      vector () : S() {};
      vector (size_type c) : S(c) {};
      vector (size_type c, const T val) : S(c, val) {};
      vector (const vector &v) : S(v) {};
      virtual ~vector () {};

      size_type size () const {return _size;}
      size_type maxsize () const {return UINT_MAX;}
      size_type capacity () const {return _capacity;}

      bool empty () const {return (_size == 0);}
    };
};
#endif

```

## B.2 Doubling Array Source

### B.2.1 doubling\_array.h

```

#ifndef DOUBLING_ARRAY_H
#define DOUBLING_ARRAY_H

namespace cphstl {
  template <class A>
  class doubling_array {
  public:
    typedef typename A::value_type value_type;
    typedef A allocator_type;
    typedef typename A::size_type size_type;
    typedef typename A::difference_type difference_type;

    typedef typename A::pointer iterator;
    typedef typename A::const_pointer const_iterator;
    typedef std::reverse_iterator <iterator> reverse_iterator;
    typedef std::reverse_iterator <const_iterator> const_reverse_iterator;

    typedef typename A::pointer pointer;
    typedef typename A::const_pointer const_pointer;
    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;

    doubling_array (const size_type c = 1,
                   const value_type& val = value_type(),

```

```

        const A& a = A()
: _alloc(a), _capacity(c), _size(0) {
_array = _alloc.allocate(_capacity);
iterator end = _array + _capacity;
for (iterator it = _array; it < end; it++)
    _alloc.construct(it, val);
}

doubling_array (const doubling_array& d, const A& a = A())
: _alloc(a), _capacity(d._capacity), _size(d._size) {
_array = _alloc.allocate(_capacity);
iterator target = _array;
const_iterator source_end = d._array + d._capacity;
for (const_iterator source = d.begin();
    source < source_end; source++)
    _alloc.construct(target++, *source);
}

~doubling_array () {
    iterator end = _array + _capacity;
    for (iterator it = _array; it < end; it++)
        _alloc.destroy(it);
    _alloc.deallocate(_array, _capacity);
}

iterator begin () {return iterator(_array);}
iterator end () {return iterator(&_array[_size]);}
const_iterator begin () const {return const_iterator(_array);}
const_iterator end () const {return const_iterator(&_array[_size]);}
reverse_iterator rbegin () {return reverse_iterator(&_array[_size]);}
reverse_iterator rend () {return reverse_iterator(_array);}

const_reverse_iterator rbegin () const {
    return const_reverse_iterator(&_array[_size]);
}

const_reverse_iterator rend () const {
    return const_reverse_iterator(_array);
}

reference operator [] (size_type n) {return _array[n];}
const_reference operator [] (size_type n) const {return _array[n];}
reference front () {if (_size == 0) return 0; return _array[0];}

const_reference front () const {
    if (_size == 0) return 0;
    return _array[0];
}

reference back () {
    if (_size == 0) return 0;
    return _array[_size - 1];
}

const_reference back () const {
    if (_size == 0) return 0;
    return _array[_size - 1];
}

doubling_array& operator = (const doubling_array& d) {
    // Deconstruct and deallocate the elements.
    iterator end = _array + _capacity;

```

```

for (iterator it = _array; it < end; it++)
    _alloc .destroy(it);
_alloc .deallocate(_array, _capacity);

// Construct the new elements.
_capacity = d._capacity;
_size = d._size;
_array = _alloc .allocate(_capacity);
const_iterator source_end = d._array + d._capacity;
iterator target = _array;
for (const_iterator source = d._array; source < source_end; source++)
    _alloc .construct(target++, *source);
return *this;
}

void reserve (size_type n) {
    if (n <= _capacity) return;

    // Allocate new space for the elements.
    iterator new_array = _alloc .allocate(n);

    // Move the objects from the old array into the new one. While we do
    // this we also destroy the old objects.
    iterator old_end = _array + _capacity;
    iterator it_new = new_array;
    iterator it_old = _array;
    while (it_old < old_end) {
        _alloc .construct(it_new++, *it_old);
        _alloc .destroy(it_old++);
    }

    // Construct the rest of the elements using the standard value val.
    value_type val = value_type();
    iterator new_end = new_array + n;
    while (it_new < new_end) {
        _alloc .construct(it_new++, val);
    }

    // Deallocate the old space and swap the pointers.
    _alloc .deallocate(_array, _capacity);
    _array = new_array;
    _capacity = n;
}

void push_back (const reference x) {
    // First we check whether a resize is necessary.
    if (_size == _capacity) {
        size_type new_capacity = _capacity * 2;
        pointer new_array = _alloc .allocate(new_capacity);
        value_type val = value_type();
        // Copy the old elements.
        for (size_type i = 0; i < _capacity; i++) {
            _alloc .construct(&new_array[i], _array[i]);
            _alloc .destroy(&_array[i]);
        }
        _alloc .deallocate(_array, _capacity);
        // And construct the rest of the elements.
        for (size_type i = _capacity + 1; i < new_capacity; i++)
            _alloc .construct(&new_array[i], val);

        _capacity = new_capacity;
        _array = new_array;
    }
}

```

```

    _alloc .construct(&_array[_size++], x);
}
else {
    // Else we just insert the new element.
    _alloc .destroy(&_array[_size]);
    _alloc .construct(&_array[_size++], x);
}
}

void pop_back () {
    _size -= 1;
    if (_size <= _capacity >> 2) {
        // Resize the array.
        size_type new_capacity = _capacity >> 1;
        pointer new_array = _alloc .allocate (new_capacity);
        value_type val = value_type ();
        // Copy the old elements into the new array. Only copy elements
        // until _size .
        for (size_type i = 0; i < _size; i++) {
            _alloc .construct(&new_array[i], _array[i]);
            _alloc .destroy(&_array[i]);
        }
        // Destroy the last elements – also empty spaces.
        for (size_type i = _size; i < _capacity; i++)
            _alloc .destroy(&_array[i]);
        _alloc .deallocate (_array, _capacity);
        // Construct empty spaces in the new array.
        for (size_type i = _size; i < new_capacity; i++)
            _alloc .construct(&new_array[i], val);
        _capacity = new_capacity;
        _array = new_array;
    }
    else {
        // Delete the element.
        _alloc .destroy(&_array[_size]);
        _alloc .construct(&_array[_size], value_type ());
    }
}

iterator insert ( iterator pos, const value_type& x) {
    // This has been split up into two parts, one where resizing is
    // needed and one that doesn't need resizing . This way less
    // element moving is done when a resize is needed.
    if (_size == _capacity) {
        // Increase the capacity of the array.
        iterator new_array = _alloc .allocate (_capacity * 2);

        // First we copy the elements in front of pos into the new array.
        iterator it_new = new_array, it_old = _array;
        while (it_old < pos) {
            _alloc .construct(it_new++, *it_old);
            _alloc .destroy(it_old++);
        }

        // Then we insert the new element and copy the rest of the
        // elements into the new array.
        pos = it_new;
        _alloc .construct(it_new++, x);
        iterator old_end = _array + _size;
        while (it_old < old_end) {
            _alloc .construct(it_new++, *it_old);
            _alloc .destroy(it_old++);
        }
    }
}

```

```

    }

    // Now we construct the rest of the elements using the
    // standard value val.
    value_type val = value_type();
    iterator new_end = new_array + _capacity * 2;
    while (it_new < new_end)
        _alloc.construct(it_new++, val);

    // The last thing to do is to deallocate the old array
    // and swap the pointers.
    _alloc.deallocate(_array, _capacity);
    _capacity *= 2;
    _array = new_array;
}
else {
    // In this case we move all the elements from pos and forward
    // one position to the right.
    iterator it = _array + _size;
    while (it > pos) {
        _alloc.destroy(it);
        _alloc.construct(it, *(it - 1));
        it--;
    }

    // And insert the element in the correct position.
    _alloc.destroy(pos);
    _alloc.construct(pos, x);
}

_size += 1;
return iterator(pos);
}

void swap (doubling_array& v) {
    pointer tmp_array = _array;
    size_type tmp_size = _size;
    size_type tmp_capacity = _capacity;
    _array = v._array;
    _size = v._size;
    _capacity = v._capacity;
    v._array = tmp_array;
    v._size = tmp_size;
    v._capacity = tmp_capacity;
}

iterator erase (iterator pos) {
    // Like the insert function erase has been split into a part that
    // resizes the array and one that doesn't to make fewer moves.
    if (_size - 1 <= _capacity >> 2) {
        // Allocate the new array.
        iterator new_array = _alloc.allocate(_capacity >> 1);

        // Copy the elements from the old array into the new one. In
        // this first copy-phase we only copy to pos (exclusive).
        iterator it_new = new_array, it_old = _array;
        while (it_old < pos) {
            _alloc.construct(it_new++, *it_old);
            _alloc.destroy(it_old++);
        }

        // Now we destroy the element at pos and continue copying the

```

```

    // rest of the elements from the old array.
    _alloc .destroy(it_old ++);
    pos = it_new;
    iterator old_end = _array + _size;
    while (it_old < old_end) {
        _alloc .construct(it_new++, *it_old);
        _alloc .destroy(it_old ++);
    }

    // Now we need to construct the rest of the elements using
    // the standard value val.
    value_type val = value_type();
    iterator new_end = new_array + (_capacity >> 1);
    while (it_new < new_end)
        _alloc .construct(it_new++, val);

    // Free the old array and swap the pointers.
    _alloc .deallocate(_array, _capacity);
    _array = new_array;
    _capacity >>= 1;
}
else {
    iterator end = _array + _size;
    iterator it = pos;
    while (it < end) {
        _alloc .destroy(it);
        _alloc .construct(it, *(it+1));
        it ++;
    }
}

_size -= 1;
return iterator(pos);
}

void clear () {
    // Destruct all the elements and deallocate the space.
    iterator end = _array + _capacity;
    for (iterator it = _array; it < end; it++)
        _alloc .destroy(it);
    _alloc .deallocate(_array, _capacity);

    // Create a new array with capacity 1.
    _size = 0;
    _capacity = 1;
    _array = _alloc .allocate(_capacity);
    _alloc .construct(_array, value_type());
}

friend bool operator == <> (const doubling_array& v1,
                             const doubling_array& v2);
friend bool operator < <> (const doubling_array& v1,
                             const doubling_array& v2);

private:
    allocator_type _alloc;

protected:
    size_type _capacity;
    size_type _size;
    pointer _array;
};

```

```

template <class A>
bool operator == (const doubling_array<A>& v1,
                 const doubling_array<A>& v2) {
    if (v1._size != v2._size)
        return false;
    else {
        for (unsigned int i = 0; i < v1._size; i++) {
            if (v1._array[i] != v2._array[i])
                return false;
        }
        return true;
    }
}

template <class A>
bool operator < (const doubling_array<A>& v1,
                const doubling_array<A>& v2) {
    unsigned int size = v1._size < v2._size ? v1._size : v2._size;
    for (unsigned int i = 0; i < size; i++) {
        if (v1._array[i] < v2._array[i])
            return true;
        else if (v1._array[i] > v2._array[i])
            return false;
    }

    return (v1._size < v2._size);
}
};
#endif

```

## B.3 Doubling Tree Source

### B.3.1 doubling\_tree.h

```

#ifndef DOUBLING_TREE_H
#define DOUBLING_TREE_H

#define TREESIZE 32

namespace cphstl {
    template <class A>
    class doubling_tree {
    public:
        typedef typename A::value_type value_type;
        typedef A allocator_type;
        typedef typename A::size_type size_type;
        typedef typename A::difference_type difference_type;

        typedef dt_iterator<A> iterator;
        typedef const dt_iterator<A> const_iterator;
        typedef std::reverse_iterator <iterator> reverse_iterator;
        typedef std::reverse_iterator <const_iterator> const_reverse_iterator;

        typedef typename A::pointer pointer;
        typedef typename A::const_pointer const_pointer;
        typedef typename A::reference reference;
        typedef typename A::const_reference const_reference;

        doubling_tree (size_type c = 1, const value_type val = value_type(),
                     const A& a = A())
            : _alloc(a), _size(0) {

```

```

int x = 1;
for (int i = 0; c; i++) {
    _tree[i] = _alloc.allocate(x);
    for (int j = 0; j < x; j++)
        _alloc.construct(&_tree[i][j], val);
    x <<= 1;
    c >>= 1;
}
_capacity = x - 1;
}

doubling_tree (const doubling_tree& d, const A& a = A())
: _alloc(a), _capacity(d._capacity), _size(d._size) {
    // Allocate space for the elements.
    int buckets = log_2(_capacity);
    for (int i = 0; i <= buckets; i++)
        _tree[i] = _alloc.allocate(pow_2(i));

    // Copy the elements from d into this doubling_tree.
    int bucket_size;
    // This for loop runs through all the buckets, even those without
    // data since they also need to be constructed.
    for (int i = 0; i <= buckets; i++) {
        bucket_size = pow_2(i);
        for (int j = 0; j < bucket_size; j++)
            _alloc.construct(&_tree[i][j], d._tree[i][j]);
    }
}

~doubling_tree () {
    int x = log_2(_capacity) + 1;
    int bsize;
    for (int i = 0; i < x; i++) {
        bsize = pow_2(i);
        for (int j = 0; j < bsize; j++)
            _alloc.destroy(&_tree[i][j]);
        _alloc.deallocate(_tree[i], bsize);
    }
}

iterator begin () {return iterator(0, _tree);}
iterator end () {return iterator(_size, _tree);}
const_iterator begin () const {return const_iterator(0, _tree);}
const_iterator end () const {return const_iterator(_size, _tree);}

reverse_iterator rbegin () {
    return reverse_iterator(iterator(_size, _tree));
}

reverse_iterator rend () {
    return reverse_iterator(iterator(0, _tree));
}

const_reverse_iterator rbegin () const {
    return const_reverse_iterator(const_iterator(_size, _tree));
}

const_reverse_iterator rend () const {
    return const_reverse_iterator(const_iterator(0, _tree));
}

reference operator [] (size_type n) {

```

```

    int bucket = log_2(n + 1);
    int bindex = n - (pow_2(bucket) - 1);
    return _tree[bucket][bindex];
}

const_reference operator [] (size_type n) const {
    int bucket = log_2(n + 1);
    int bindex = n - (pow_2(bucket) - 1);
    return _tree[bucket][bindex];
}

reference front () {return _size == 0 ? 0 : _tree [0][0];}
const_reference front () const {return _size == 0 ? 0 : _tree [0][0];}

reference back () {
    if (_size == 0) return 0;
    int bucket = log_2(_size);
    int bindex = (_size - 1) - (pow_2(bucket) - 1);
    return _tree[bucket][bindex];
}

const_reference back () const {
    if (_size == 0) return 0;
    int bucket = log_2(_size);
    int bindex = (_size - 1) - (pow_2(bucket) - 1);
    return _tree[bucket][bindex];
}

doubling_tree& operator = (const doubling_tree& v) {
    // Free the space that is no longer needed in the vector, or allocate
    // extra space if it's needed.
    int new_storage = 0; // This is an indicator that shows whether
                        // new storage was allocated.
    if (_capacity > v._capacity) {
        int start = log_2(v._capacity) + 1;
        int stop = log_2(_capacity) + 1;
        int bsize;
        while (start < stop) {
            bsize = pow_2(start);
            for (int i = 0; i < bsize; i++)
                _alloc.destroy(&_tree[start][i]);
            _alloc.deallocate(_tree[start], bsize);
        }
    }
    else if (_capacity < v._capacity) {
        int start = log_2(_capacity) + 1;
        int stop = log_2(v._capacity) + 1;
        while (start < stop) {
            _tree[start] = _alloc.allocate(pow_2(start));
            start++;
        }
        new_storage = start;
    }

    _size = v._size;
    _capacity = v._capacity;

    // Copy the elements from v into this vector.
    int buckets = log_2(_capacity);
    int bsize;
    // This loop runs through all the buckets even those without data
    // in them. There are two cases here depending on whether new storage

```

```

// was allocated earlier in the function. If new storage was
// allocated the elements in those buckets need not be destroyed.
if (new_storage) {
    // First we copy the elements into old storage where
    // destruction is needed.
    for (int i = 0; i < new_storage; i++) {
        bsize = pow_2(i);
        for (int j = 0; j < bsize; j++) {
            _alloc.destroy(&_tree[i][j]);
            _alloc.construct(&_tree[i][j], v._tree[i][j]);
        }
    }
    // Then we copy the rest of the element into the new storage.
    for (int i = new_storage; i <= buckets; i++) {
        bsize = pow_2(i);
        for (int j = 0; j < bsize; j++)
            _alloc.construct(&_tree[i][j], v._tree[i][j]);
    }
}
else {
    for (int i = 0; i <= buckets; i++) {
        bsize = pow_2(i);
        for (int j = 0; j < bsize; j++) {
            _alloc.destroy(&_tree[i][j]);
            _alloc.construct(&_tree[i][j], v._tree[i][j]);
        }
    }
}

return *this;
}

void reserve (size_type n) {
    if (n <= _capacity) return;
    int i = log_2(_capacity) + 1;
    size_type x = pow_2(i);
    value_type val = value_type();
    while (x < n) {
        _tree[i] = _alloc.allocate(x);
        for (unsigned int j = 0; j < x; j++)
            _alloc.construct(&_tree[i][j], val);
        i++;
        x <<= 1;
    }
    _capacity = x - 1;
}

void swap (doubling_tree& v) {
    // Swap the pointers to buckets.
    int min_cap = _capacity < v._capacity ? log_2(_capacity) :
        log_2(v._capacity);
    value_type *tmp;
    for (int i = 0; i <= min_cap; i++) {
        tmp = _tree[i];
        _tree[i] = v._tree[i];
        v._tree[i] = tmp;
    }

    if (_capacity < v._capacity) {
        int max_cap = log_2(v._capacity);
        for (int i = min_cap + 1; i <= max_cap; i++)
            _tree[i] = v._tree[i];
    }
}

```

```

    } else {
        int max_cap = log_2(_capacity);
        for (int i = min_cap + 1; i <= max_cap; i++)
            v._tree[i] = _tree[i];
    }

    // Swap the _size and _capacity information.
    size_type tmp_size = _size;
    size_type tmp_capacity = _capacity;
    _size = v._size;
    _capacity = v._capacity;
    v._size = tmp_size;
    v._capacity = tmp_capacity;
}

void push_back (const reference x) {
    size_type bucket = log_2(_size + 1);
    // Add an extra bucket if needed.
    if (_size == _capacity) {
        size_type bsize = _capacity + 1;
        value_type val = value_type();
        _tree[bucket] = _alloc.allocate(bsize);
        for (size_type i = 1; i < bsize; i++)
            _alloc.construct(&_tree[bucket][i], val);
        _capacity += bsize;
        // Insert the element in the first position in the new leaf.
        _alloc.construct(&_tree[bucket][0], x);
    }
    else {
        // Insert the new element.
        size_type bindex = _size - (pow_2(bucket) - 1);
        _alloc.destroy(&_tree[bucket][bindex]);
        _alloc.construct(&_tree[bucket][bindex], x);
    }
    _size += 1;
}

void pop_back () {
    _size -= 1;
    // Deallocate the last bucket if needed.
    if (_size <= _capacity >> 2) {
        size_type last_bucket = log_2(_capacity);
        size_type bsize = pow_2(last_bucket);
        for (size_type i = 0; i < bsize; i++)
            _alloc.destroy(&_tree[last_bucket][i]);
        _alloc.deallocate(_tree[last_bucket], bsize);
        _capacity >>= 1;
    }
    // Delete the last element.
    size_type bucket = log_2(_size + 1);
    size_type bindex = _size - (pow_2(bucket) - 1);
    _alloc.destroy(&_tree[bucket][bindex]);
    _alloc.construct(&_tree[bucket][bindex], value_type());
}

iterator insert (iterator pos, const value_type& x) {
    int n = pos - this->begin();
    int bucket = log_2(n+1);
    int bindex = n - (pow_2(bucket) - 1);

    // Allocate an extra bucket if the vector is already full.
    if (_size == _capacity) {

```

```

    int bsize = _capacity + 1;
    int new_idx = log_2(_capacity) + 1;
    value_type val = value_type();
    _tree[new_idx] = _alloc.allocate(bsize);
    for (int i = 0; i < bsize; i++)
        _alloc.construct(&_tree[new_idx][i], val);
    _capacity += bsize;
}

// Check if the insertion is to the last position in the tree. If
// so we insert it now and skip the whole "move" section below.
if (n != (int) _size) {
    // Move elements one position right.
    int last_bucket = log_2(_size);
    int cur_bucket = last_bucket;
    int move_from = _size - 1 - (pow_2(last_bucket) - 1);
    int bucket_size = pow_2(cur_bucket);
    while (cur_bucket >= bucket) {
        // If the first element to be moved is the last in this bucket
        // we copy it into the next bucket in the tree.
        if (move_from == bucket_size - 1) {
            _alloc.destroy(&_tree[cur_bucket+1][0]);
            _alloc.construct(&_tree[cur_bucket+1][0],
                _tree[cur_bucket][move_from--]);
        }

        // If this is the bucket that x is to be inserted into we
        // only move elements in the range [bindex:move_from]
        if (move_from != -1) {
            if (cur_bucket == bucket) {
                for (int i = move_from; i >= bindex; i--) {
                    _alloc.destroy(&_tree[bucket][i+1]);
                    _alloc.construct(&_tree[bucket][i+1], _tree[bucket][i]);
                }
            }
            else {
                for (int i = move_from; i >= 0; i--) {
                    _alloc.destroy(&_tree[cur_bucket][i+1]);
                    _alloc.construct(&_tree[cur_bucket][i+1],
                        _tree[cur_bucket][i]);
                }
            }
        }
    }

    bucket_size >>= 1;
    move_from = bucket_size - 1;
    cur_bucket--;
}

// Insert x into the bucket.
_alloc.destroy(&_tree[bucket][bindex]);
_alloc.construct(&_tree[bucket][bindex], x);

_size += 1;
return iterator(n, _tree);
}

iterator erase (iterator pos) {
    int n = pos - this->begin();
    value_type val = value_type();
    _size -= 1;
}

```

```

// Deallocate the last bucket if the number of elements are 4
// times lower than the capacity. This means that only one bucket
// can be empty at any one time.
if ( _size <= _capacity >> 2) {
    int idx = log_2(_capacity);
    int bsize = pow_2(idx);
    for (int i = 0; i < bsize; i++)
        _alloc .destroy(&_tree[idx][i]);
    _alloc .deallocate(_tree[idx], bsize);
    _capacity >>= 1;
}

// Move all the elements in front of pos one position to the left .
// bucket and move_from is set to point to the element after the one
// to be deleted. This is only done if n != _size .
if (n != (int) _size) {
    int bucket = log_2(n + 2);
    int move_from = n + 2 - pow_2(bucket);
    int last_bucket = log_2(_size + 1);
    int last_elem_bidx = _size - (pow_2(last_bucket) - 1);
    int cur_bucket = bucket;
    int bucket_size = pow_2(cur_bucket);

    while (cur_bucket <= last_bucket) {
        if (move_from == 0) {
            int i = cur_bucket - 1, j = (bucket_size >> 1) - 1;
            _alloc .destroy(&_tree[i][j]);
            _alloc .construct(&_tree[i][j], _tree[cur_bucket][0]);
        }

        // If this is the last bucket we only need to move until
        // last_elem_bidx.
        if (cur_bucket == last_bucket) {
            for (int i = 0; i < last_elem_bidx; i++) {
                _alloc .destroy(&_tree[cur_bucket][i]);
                _alloc .construct(&_tree[cur_bucket][i],
                    _tree[cur_bucket][i+1]);
            }
            _alloc .destroy(&_tree[cur_bucket][last_elem_bidx]);
            _alloc .construct(&_tree[cur_bucket][last_elem_bidx], val);
        }
        else {
            for (int i = 0; i < bucket_size; i++) {
                _alloc .destroy(&_tree[cur_bucket][i]);
                _alloc .construct(&_tree[cur_bucket][i],
                    _tree[cur_bucket][i+1]);
            }
        }
        move_from = 0;
        bucket_size <<= 1;
        cur_bucket++;
    }
}
else {
    int bucket = log_2(n + 1);
    int bindex = n - (pow_2(bucket) - 1);
    _alloc .destroy(&_tree[bucket][bindex]);
    _alloc .construct(&_tree[bucket][bindex], val);
}

return iterator(n, _tree);

```

```

}

void clear () {
    int buckets = log_2(_capacity);
    int bsize;
    for (int i = 1; i <= buckets; i++) {
        bsize = pow_2(i);
        for (int j = 0; j < bsize; j++)
            _alloc.destroy(&_tree[i][j]);
        _alloc.deallocate(_tree[i], bsize);
    }
    _size = 0;
    _capacity = 1;
    _alloc.destroy(&_tree[0][0]);
    _alloc.construct(&_tree[0][0], value_type());
}

friend bool operator == <> (const doubling_tree& v1,
                             const doubling_tree& v2);
friend bool operator < <> (const doubling_tree& v1,
                           const doubling_tree& v2);

private:
    allocator_type _alloc;

protected:
    size_type _capacity;
    size_type _size;
    pointer _tree[TREESIZE];
};

template <class A>
bool operator == (const doubling_tree<A>& v1,
                 const doubling_tree<A>& v2) {
    if (v1._size != v2._size)
        return false;
    else {
        int last_bucket = log_2(v1._size);
        int last_elem_bidx = v1._size - pow_2(last_bucket);
        int bucket_size = 1;
        int cur_bucket = 0;
        while (cur_bucket <= last_bucket){
            if (cur_bucket == last_bucket) {
                for (int i = 0; i <= last_elem_bidx; i++) {
                    if (v1._tree[cur_bucket][i] != v2._tree[cur_bucket][i])
                        return false;
                }
            }
            else {
                for (int i = 0; i < bucket_size; i++) {
                    if (v1._tree[cur_bucket][i] != v2._tree[cur_bucket][i])
                        return false;
                }
            }
            bucket_size <<= 1;
            cur_bucket++;
        }
        return true;
    }
}

template <class A>

```

```

bool operator < (const doubling_tree<A>& v1,
               const doubling_tree<A>& v2) {
    unsigned int size = v1._size < v2._size ? v1._size : v2._size;
    int last_bucket = log_2(size);
    int last_elem_bidx = size - pow_2(last_bucket);
    int bucket_size = 1;
    int cur_bucket = 0;
    while (cur_bucket <= last_bucket){
        if (cur_bucket == last_bucket) {
            for (int i = 0; i <= last_elem_bidx; i++) {
                if (v1._tree[cur_bucket][i] < v2._tree[cur_bucket][i])
                    return true;
                else if (v1._tree[cur_bucket][i] > v2._tree[cur_bucket][i])
                    return false;
            }
        }
        else {
            for (int i = 0; i < bucket_size; i++) {
                if (v1._tree[cur_bucket][i] < v2._tree[cur_bucket][i])
                    return true;
                else if (v1._tree[cur_bucket][i] > v2._tree[cur_bucket][i])
                    return false;
            }
        }
        bucket_size <<= 1;
        cur_bucket++;
    }

    return (v1._size < v2._size);
}
};
#endif

```

### B.3.2 dt\_iterator.h

```

#ifndef DT_ITERATOR_H
#define DT_ITERATOR_H

namespace cphstl {
    template <class A>
    class dt_iterator {
    private:
        typedef dt_iterator iterator;

    public:
        typedef std::random_access_iterator_tag iterator_category;
        typedef typename A::value_type value_type;
        typedef typename A::pointer pointer;
        typedef typename A::reference reference;
        typedef typename A::difference_type difference_type;
        typedef typename A::size_type size_type;

        dt_iterator () : _pos(0), _tree(0) {}
        dt_iterator (pointer *tree) : _pos(0), _tree(tree) {}
        dt_iterator (size_type pos, pointer *tree) : _pos(pos), _tree(tree) {}

        iterator operator ++ (int) {
            size_type x = _pos;
            _pos++;
            return iterator(x, _tree);
        }
    }
}

```

```

iterator operator -- (int) {
    size_type x = _pos;
    _pos--;
    return iterator(x, _tree);
}

iterator& operator ++ () {_pos++; return *this;}
iterator& operator -- () {_pos--; return *this;}

reference operator * () const {
    int bucket = log_2(_pos+1);
    int bindex = _pos - (pow_2(bucket) - 1);
    return _tree[bucket][bindex];
}

iterator& operator = (const iterator& it) {
    _pos = it._pos;
    _tree = it._tree;
    return *this;
}

iterator& operator += (const difference_type i) {
    _pos += i;
    return *this;
}

iterator& operator -= (const difference_type i) {
    _pos -= i;
    return *this;
}

// These comparison operators assumes that the iterators being compared
// are iterators into the same container.
bool operator == (const iterator& it) const {return _pos == it._pos;}
bool operator < (const iterator& it) const {return _pos < it._pos;}
bool operator <= (const iterator& it) const {return _pos <= it._pos;}
bool operator > (const iterator& it) const {return _pos > it._pos;}
bool operator >= (const iterator& it) const {return _pos >= it._pos;}
bool operator != (const iterator& it) const {return _pos != it._pos;}

// There are no bounds checking on these functions.
iterator operator + (const difference_type i) const {
    return iterator(_pos + i, _tree);
}

friend iterator operator + <> (difference_type i, const iterator& it);

iterator operator - (const difference_type i) const {
    return iterator(_pos - i, _tree);
}

difference_type operator - (const iterator& it) const {
    return _pos - it._pos;
}

private:
    size_type _pos;
    pointer * _tree;
};

template<class A>
dt_iterator<A> operator + (typename A::difference_type i,

```

```

        const dt_iterator<A>& it) {
    return dt_iterator<A>(it._pos + i, it._tree);
}
};
#endif

```

## B.4 Hashed Array Tree Source

### B.4.1 hashed\_array\_tree.h

```

#ifndef HASHED_ARRAY_TREE_H
#define HASHED_ARRAY_TREE_H

namespace cphstl {
    template <class A>
    class hashed_array_tree {
        friend class hat_iterator<A>;
    private:
        inline void find_index (unsigned int n, int& leaf, int& index) {
            leaf = n >> _power;
            index = n & _mask;
        }

        inline int find_leafsize (unsigned int n) {
            int leafsize = 1, capacity = 1;
            while (capacity < (int) n) {
                leafsize <<= 1;
                capacity <<= 2;
            }
            return leafsize;
        }
    public:
        typedef typename A::value_type value_type;
        typedef A allocator_type;
        typedef typename A::size_type size_type;
        typedef typename A::difference_type difference_type;

        typedef hat_iterator<A> iterator;
        typedef const hat_iterator<A> const_iterator;
        typedef std::reverse_iterator<iterator> reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

        typedef typename A::pointer pointer;
        typedef typename A::const_pointer const_pointer;
        typedef typename A::reference reference;
        typedef typename A::const_reference const_reference;

        hashed_array_tree (const size_type c = 1,
                          const value_type& val = value_type(),
                          const A& a = A())
            : _alloc(a), _size(0) {
            // Find the initial capacity as the square of a power of two
            // that corresponds best to the initial_capacity .
            _leafsize = find_leafsize(c);
            _top_array = _palloc.allocate(_leafsize);
            int leaves = c / _leafsize + (c % _leafsize ? 1 : 0);
            for (int i = 0; i < leaves; i++) {
                _top_array[i] = _alloc.allocate(_leafsize);
                for (int j = 0; j < _leafsize; j++)
                    _alloc.construct(&_top_array[i][j], val);
            }
        }
    };
}

```

```

        _capacity = _leafsize * leafs ;
        _power = log2_ceil( _leafsize );
        _mask = (1 << _power) - 1;
    }

    hashed_array_tree (const hashed_array_tree& v, const A& a = A())
        : _leafsize (v._leafsize), _mask(v._mask),
        _power(v._power), _alloc(a),
        _capacity(v._capacity), _size(v._size) {
        // Allocate the top array and the leafs.
        _top_array = _palloc.allocate( _leafsize );

        // Copy the elements.
        int leafs = _capacity / _leafsize ;
        for (int i = 0; i < leafs ; i++) {
            _top_array[i] = _alloc.allocate( _leafsize );
            for (int j = 0; j < _leafsize ; j++) {
                _alloc.construct(&_top_array[i][j], v._top_array[i][j]);
            }
        }
    }

    ~hashed_array_tree () {
        int leafs = _capacity / _leafsize ;
        for (int i = 0; i < leafs ; i++) {
            for (int j = 0; j < _leafsize ; j++)
                _alloc.destroy(&_top_array[i][j]);
            _alloc.deallocate(_top_array[i], _leafsize );
        }
        _palloc.deallocate(_top_array, _leafsize );
    }

    iterator begin () {return iterator(0, this);}
    iterator end () {return iterator(_size, this);}
    const_iterator begin () const {return const_iterator(0, this);}
    const_iterator end () const {return const_iterator(_size, this);}

    reverse_iterator rbegin () {
        return reverse_iterator(iterator(_size, this));
    }

    reverse_iterator rend () {
        return reverse_iterator(iterator(0, this));
    }

    const_reverse_iterator rbegin () const {
        return const_reverse_iterator(const_iterator(_size, this));
    }

    const_reverse_iterator rend () const {
        return const_reverse_iterator(const_iterator(0, this));
    }

    reference operator [] (size_type n) {
        return _top_array[n >> _power][n & _mask];
    }

    const_reference operator [] (size_type n) const {
        return _top_array[n >> _power][n & _mask];
    }

    reference front () {

```

```

    return _size == 0 ? 0 : _top_array [0][0];
}

const_reference front () const {
    return _size == 0 ? 0 : _top_array [0][0];
}

reference back () {
    if ( _size == 0) return 0;
    return _top_array[_size >> _power][_size & _mask];
}

const_reference back () const {
    if ( _size == 0) return 0;
    return _top_array[_size >> _power][_size & _mask];
}

hashed_array_tree& operator = (const hashed_array_tree& v) {
    // If the leafsize of the two trees are different we just deallocate
    // everything and allocate new leaves.
    if ( _leafsize != v. _leafsize ) {
        // Destroy and deallocate everything.
        int leafs = _capacity / _leafsize ;
        for (int i = 0; i < leafs; i++) {
            for (int j = 0; j < _leafsize ; j++)
                _alloc .destroy(&_top_array[i][j]);
            _alloc .deallocate(_top_array [i], _leafsize );
        }
        _palloc .deallocate(_top_array , _leafsize );

        // Allocate a new top_array and allocate the new elements. While
        // doing this we copy the values from v into the new leaves.
        _top_array = _palloc .allocate(v. _leafsize );
        int need = v. _capacity / v. _leafsize ;
        for (int i = 0; i < need; i++) {
            _top_array [i] = _alloc .allocate(v. _leafsize );
            for (int j = 0; j < v. _leafsize ; j++)
                _alloc .construct(&_top_array [i][j], v. _top_array [i][j]);
        }
        _leafsize = v. _leafsize ;
        _mask = v. _mask;
        _power = v. _power;
    }
    else {
        // The leafsizes are the same. Now we free the space that is no
        // longer needed or allocate extra leafs if they are needed.
        int leafs = _capacity / _leafsize ;
        int need = v. _capacity / _leafsize ;
        if ( leafs < need) {
            // We have to allocate extra leafs .
            for (int i = leafs ; i < need; i++)
                _top_array [i] = _alloc .allocate( _leafsize );
        }
        else {
            // We might have to deallocate some leafs.
            for (int i = need; i < leafs ; i++) {
                for (int j = 0; j < _leafsize ; j++)
                    _alloc .destroy(&_top_array [i][j]);
                _alloc .deallocate(_top_array [i], _leafsize );
            }
            leafs = need; // We set leafs to need to make sure that
            // element copying works.
        }
    }
}

```

```

    }

    // Copy the data from v into the new storage. First we copy into
    // the old leaves where destruction of old elements are needed.
    for (int i = 0; i < leafsize; i++) {
        for (int j = 0; j < _leafsize; j++) {
            _alloc.destroy(&_top_array[i][j]);
            _alloc.construct(&_top_array[i][j], v._top_array[i][j]);
        }
    }
    // Then we copy into the new leaves that doesn't need destruction
    // of elements.
    for (int i = leafsize; i < need; i++) {
        for (int j = 0; j < _leafsize; j++)
            _alloc.construct(&_top_array[i][j], v._top_array[i][j]);
    }
}
_size = v._size;
_capacity = v._capacity;
return *this;
}

void reserve (size_type n) {
    if (n <= _capacity) return;

    // There are two cases here, one where we have to change the
    // leafsize and one where we can reserve the space without
    // changing the leafsize.
    int new_leafsize = find_leafsize(n);
    int needed_leafs = n / new_leafsize + (n % new_leafsize ? 1 : 0);
    value_type val = value_type();

    if (_leafsize == new_leafsize) {
        for (int i = _capacity / _leafsize; i < needed_leafs; i++) {
            _top_array[i] = _alloc.allocate(_leafsize);
            for (int j = 0; j < _leafsize; j++)
                _alloc.construct(&_top_array[i][j], val);
            _capacity += _leafsize;
        }
    }
    else {
        pointer *new_top_array = _palloc.allocate(new_leafsize);

        // For each new leaf that is needed we allocate it and either
        // copy the old value into the position or construct it with
        // the default value val. First we allocate the new leaves.
        for (int i = 0; i < needed_leafs; i++) {
            new_top_array[i] = _alloc.allocate(new_leafsize);
        }

        // Then we copy the elements from the old top_array.
        int x = new_leafsize / _leafsize; // The number of old leaves that
        // fit into one new leaf.
        int leafsize = _capacity / _leafsize;
        for (int i = 0; i < leafsize; i++) {
            for (int j = 0; j < _leafsize; j++) {
                _alloc.construct(&new_top_array[i/x][(i%x) * _leafsize + j],
                    _top_array[i][j]);
                _alloc.destroy(&_top_array[i][j]);
            }
            _alloc.deallocate(_top_array[i], _leafsize);
        }
    }
}

```

```

// Now we have to check whether one of the leaves have not been
// fully constructed.
int idx = (_capacity / new_leafsize + (_capacity % new_leafsize ?
                                         1 : 0)) - 1;

int start;
if ((start = _capacity % new_leafsize) != 0) {
    for (int i = start; i < new_leafsize; i++)
        _alloc.construct(&new_top_array[idx][i], val);
    idx++;
}

// The rest of the needed leaves only needs to be constructed
// with the standard value val.
for (int i = idx; i < needed_leafs; i++) {
    for (int j = 0; j < new_leafsize; j++)
        _alloc.construct(&new_top_array[i][j], val);
}

_top_array = new_top_array;
_leafsize = new_leafsize;
_capacity = _leafsize * needed_leafs;
_power = log2_ceil(_leafsize);
_mask = (1 << _power) - 1;
}
}

void swap (hashed_array_tree& v) {
    int tmp_leafsize = _leafsize;
    unsigned int tmp_power = _power;
    unsigned int tmp_mask = _mask;
    unsigned int tmp_capacity = _capacity;
    unsigned int tmp_size = _size;
    value_type **tmp_top_array = _top_array;
    _leafsize = v._leafsize;
    _power = v._power;
    _mask = v._mask;
    _capacity = v._capacity;
    _size = v._size;
    _top_array = v._top_array;
    v._leafsize = tmp_leafsize;
    v._power = tmp_power;
    v._mask = tmp_mask;
    v._capacity = tmp_capacity;
    v._size = tmp_size;
    v._top_array = tmp_top_array;
}

void push_back (const reference x) {
    // Check to see if extra storage must be allocated.
    if (_size == _capacity) {
        // Check to see if we need to change leafsize.
        if (_size == (unsigned int)_leafsize * _leafsize) {
            int new_leafsize = _leafsize << 1;
            pointer* new_top_array = _palloc.allocate (new_leafsize);
            // This is one special case here and that's when the old
            // leafsize is 1. This is handled separately here. This is a
            // special case because the moving of old elements does not
            // completely fill the new leaves of the new top_array.
            if (_leafsize == 1) {
                new_top_array[0] = _alloc.allocate (2);
                _alloc.construct(&new_top_array[0][0], _top_array[0][0]);
            }
        }
    }
}

```

```

        _alloc .construct(&new_top_array[0][1], x);
        _alloc .destroy(&_top_array[0][0]);
        _alloc .deallocate(_top_array[0], _leafsize );
        _palloc .deallocate(_top_array, _leafsize );
        _top_array = new_top_array;
        _leafsize = 2;
        _capacity = 2;
        _power = 1;
        _mask = 1;
    }
    else {
        // This loops over the new leafs needed by the objects in
        // the old leafs .
        _power += 1;
        size_type last_leaf = _size >> _power;
        for (size_type i = 0; i < last_leaf ; i++) {
            new_top_array[i] = _alloc .allocate (new_leafsize );
            size_type pos = 0; // The starting position in this new leaf.
            // This loops over the next two old leafs that need to
            // be copied into new leafs.
            size_type stop_index = (i + 1) * 2;
            for (size_type j = i * 2; j < stop_index; j++) {
                // Copy the actual elements.
                for (int k = 0; k < _leafsize ; k++) {
                    _alloc .construct(&new_top_array[i][pos++],
                                     _top_array[j][k]);
                    _alloc .destroy(&_top_array[j][k]);
                }
                _alloc .deallocate(_top_array[j], _leafsize );
            }
        }
        _palloc .deallocate(_top_array, _leafsize );
        _top_array = new_top_array;

        // Allocate the extra leaf that we wanted to begin with.
        _top_array[ last_leaf ] = _alloc .allocate (new_leafsize );
        value_type val = value_type();
        for (int i = 1; i < new_leafsize ; i++)
            _alloc .construct(&_top_array[ last_leaf ][i], val);
        _leafsize = new_leafsize ;
        _capacity += _leafsize ;
        _mask = (_mask << 1) | 1;
        // Insert the element.
        _alloc .construct(&_top_array[ last_leaf ][0], x);
    }
}
// If no change in leafsize is needed we just allocate an
// extra leaf .
else {
    size_type new_leaf = _capacity >> _power;
    value_type val = value_type();
    _top_array [new_leaf] = _alloc .allocate ( _leafsize );
    for (int i = 1; i < _leafsize ; i++)
        _alloc .construct(&_top_array[new_leaf][i], val);
    _capacity += _leafsize ;
    // Insert the element.
    _alloc .construct(&_top_array[new_leaf][0], x);
}
}
else {
    // Insert the new element.
    int leaf = _size >> _power;

```

```

    int leaf_idx = _size & _mask;
    _alloc.destroy(&_top_array[leaf][leaf_idx]);
    _alloc.construct(&_top_array[leaf][leaf_idx], x);
}
_size += 1;
}

void pop_back () {
    value_type val = value_type();
    size_type max_capacity = _leafsize * _leafsize;
    _size -= 1;
    // Check if resizing is needed.
    if (_capacity != 1 && _size <= max_capacity / 8) {
        int new_leafsize = _leafsize >> 1;
        pointer *new_top_array = _palloc.allocate(new_leafsize);
        size_type last_leaf = _size >> _power;
        size_type last_idx = _size & _mask;
        size_type cur_new_leaf = 0;

        // Start by allocating the leaves that are needed in the new
        // top_array.
        size_type new_leafs = _size / new_leafsize +
            (_size % new_leafsize ? 1 : 0);
        if (new_leafs == 0) new_leafs = 1; // If _size is 0.
        for (size_type i = 0; i < new_leafs; i++)
            new_top_array[i] = _alloc.allocate(new_leafsize);

        // For every leaf containing data in the old top_array but
        // the last one.
        for (size_type i = 0; i < last_leaf; i++) {
            for (int j = 0; j < new_leafsize; j++)
                _alloc.construct(&new_top_array[cur_new_leaf][j],
                    _top_array[i][j]);
            cur_new_leaf++;
            for (int j = new_leafsize; j < _leafsize; j++)
                _alloc.construct(&new_top_array[cur_new_leaf][j%new_leafsize],
                    _top_array[i][j]);
            cur_new_leaf++;
        }

        // Now take care of the last leaf. First check to see if
        // index == 0 in which case no more elements must be moved.
        if (last_idx != 0) {
            // See if two new leaves are needed.
            if (((int)last_idx > new_leafsize) {
                // Fill the first leaf.
                for (int j = 0; j < new_leafsize; j++)
                    _alloc.construct(&new_top_array[cur_new_leaf][j],
                        _top_array[last_leaf][j]);
                cur_new_leaf++;
                // And copy the elements until n into the last leaf.
                for (size_type j = new_leafsize; j < last_idx; j++)
                    _alloc.construct(&new_top_array[cur_new_leaf][j%new_leafsize],
                        _top_array[last_leaf][j]);
                // Construct the empty spaces.
                for (int j = last_idx%new_leafsize; j < new_leafsize; j++)
                    _alloc.construct(&new_top_array[cur_new_leaf][j], val);
            }
            else {
                for (size_type j = 0; j < last_idx; j++)
                    _alloc.construct(&new_top_array[cur_new_leaf][j],
                        _top_array[last_leaf][j]);
            }
        }
    }
}

```

```

        // Construct the empty spaces.
        for (int j = last_idx; j < new_leafsize; j++)
            _alloc.construct(&new_top_array[cur_new_leaf][j], val);
    }
}

// Destroy the elements in the old top_array.
last_leaf = _capacity / _leafsize;
for (size_type i = 0; i < last_leaf; i++) {
    for (int j = 0; j < _leafsize; j++)
        _alloc.destroy(&_top_array[i][j]);
    _alloc.deallocate(_top_array[i], _leafsize);
}
_palloc.deallocate(_top_array, _leafsize);
_top_array = new_top_array;
_leafsize = new_leafsize;
_capacity = new_leafs * _leafsize;
_power -= 1;
_mask = _mask >> 1;
}
// No resizing is needed.
else {
    size_type last_leaf = _size >> _power;
    size_type last_idx = _size & _mask;
    _alloc.destroy(&_top_array[last_leaf][last_idx]);
    _alloc.construct(&_top_array[last_leaf][last_idx], val);

    // Check to see if there are two empty leaves. If so we
    // deallocate the last one.
    last_leaf = (_capacity - 1) >> _power;
    if (last_leaf >= ((size - 1) >> _power) + 2) {
        for (int i = 0; i < _leafsize; i++)
            _alloc.destroy(&_top_array[last_leaf][i]);
        _alloc.deallocate(_top_array[last_leaf], _leafsize);
        _capacity -= _leafsize;
    }
}
}

// This function could be optimized when a resize is done by moving
// elements while copying them from the old top_array. It has not
// been done because of 1) time, 2) it's not necessary for what
// I'm measuring in my report.
iterator insert (iterator pos, const value_type& x) {
    // Allocate an extra leaf if this is needed.
    if (_size == _capacity) {
        value_type val = value_type();
        // Check to see if we need to change leafsize.
        if (_size == (unsigned int)_leafsize * _leafsize) {
            int new_leafsize = _leafsize << 1;
            pointer* new_top_array = _palloc.allocate(new_leafsize);
            // The is one special case here and that's when the old
            // leafsize is 1. This is handled separately here. This is
            // a special case because the moving of old elements does
            // not completely fill the new leaves of the new top_array.
            if (_leafsize == 1) {
                new_top_array[0] = _alloc.allocate(2);
                _alloc.construct(&new_top_array[0][0], _top_array[0][0]);
                _alloc.construct(&new_top_array[0][1], val);
                _alloc.destroy(&_top_array[0][0]);
                _alloc.deallocate(_top_array[0], _leafsize);
                _palloc.deallocate(_top_array, _leafsize);
            }
        }
    }
}

```

```

_top_array = new_top_array;
_leafsize = 2;
_capacity = 2;
_power = 1;
_mask = 1;
}
else {
// This loops over the new leafs needed by the objects in
// the old leafs .
int last_leaf = _size / new_leafsize;
for (int i = 0; i < last_leaf ; i++) {
new_top_array[i] = _alloc .allocate (new_leafsize );
int pos = 0; // The starting position in this new leaf.
// This loops over the next two old leafs that need to
// be copied into
// new leafs .
for (int j = i * 2; j < (i + 1) * 2; j++) {
// Copy the actual elements.
for (int k = 0; k < _leafsize ; k++) {
_alloc .construct(&new_top_array[i][pos++],
_top_array [j][k]);
_alloc .destroy(&_top_array[j][k]);
}
_alloc .deallocate (_top_array[j] , _leafsize );
}
}
_palloc .deallocate (_top_array , _leafsize );
_top_array = new_top_array;

// Allocate the extra leaf that we wanted to begin with.
_top_array[ last_leaf ] = _alloc .allocate (new_leafsize );
for (int i = 0; i < new_leafsize ; i++)
_alloc .construct(&_top_array[ last_leaf ][i] , val);
_leafsize = new_leafsize ;
_capacity += _leafsize ;
_power = log_2_ceil( _leafsize );
_mask = (1 << _power) - 1;
}
}
// If no change in leafsize is needed we just allocate
// an extra leaf .
else {
int new_leaf = _capacity / _leafsize ;
_top_array [new_leaf] = _alloc .allocate ( _leafsize );
for (int i = 0; i < _leafsize ; i++)
_alloc .construct(&_top_array[new_leaf][i] , val);
_capacity += _leafsize ;
}
}

// Insert the element into position n.
int index = pos - this->begin();
int leaf = index / _leafsize ;
int n = index % _leafsize ; // The index in the leaf .
// Now move all the elements in front of position n one space up
// in the tree.
int last_leaf , move_from;
find_index( _size - 1, last_leaf , move_from);
// This loops over the leaf from the last one to the one that
// contains n.
for (int i = last_leaf ; i >= leaf; i--) {
// First we check to see if we need to move the last element

```

```

// in the leaf into the leaf in front of this one.
if (move_from == _leafsize - 1) {
    _alloc .destroy(&_top_array[i+1][0]);
    _alloc .construct(&_top_array[i+1][0],
                    _top_array[i][move_from--]);
}

// The we move the rest of the elements inside of the leaf . If
// this is the leaf that contains n we only move the elements
// until the n'th position.
if (i == leaf) {
    for (int j = move_from; j >= n; j--) {
        _alloc .destroy(&_top_array[i][j+1]);
        _alloc .construct(&_top_array[i][j+1], _top_array[i][j]);
    }
}
else {
    for (int j = move_from; j >= 0; j--) {
        _alloc .destroy(&_top_array[i][j+1]);
        _alloc .construct(&_top_array[i][j+1], _top_array[i][j]);
    }
}
move_from = _leafsize - 1;
}

// Insert x into the hole that has appeared.
_alloc .destroy(&_top_array[leaf][n]);
_alloc .construct(&_top_array[leaf][n], x);
_size += 1;

return iterator(n, this);
}

iterator erase (iterator pos) {
    int n = pos - this->begin();
    value_type val = value_type();

    // Check if we need to change to another leafsize after this
    // erase . We change leafsize if the size is 1/8 of the max
    // capacity with the current leafsize .
    unsigned int max_cap = _leafsize * _leafsize;
    if (_capacity != 1 && _size - 1 <= max_cap / 8) {
        int new_leafsize = _leafsize >> 1;
        pointer *new_top_array = _palloc .allocate (new_leafsize);
        int last_leaf = (_size - 1) / _leafsize ;
        int last_idx = (_size - 1) % _leafsize ;
        int n_leaf = n / _leafsize ;
        int n_idx = n % _leafsize ;
        int cur_new_leaf = 0;

        // Start by allocating the leaves that are needed in the
        // new top_array.
        int new_leafs = new_leafsize == 1 ? 1 : (_size - 1) /
            new_leafsize + (( _size - 1) % new_leafsize ? 1 : 0);
        for (int i = 0; i < new_leafs; i++)
            new_top_array[i] = _alloc .allocate (new_leafsize);

        // For every leaf containing data in the old top_array.
        for (int i = 0; i <= last_leaf; i++) {
            // If this is before the leaf that contain n we just
            // copy the elements.
            if (i < n_leaf) {

```

```

int split ;
for (int j = 0; j < _leafsize ; j++) {
    split = j >= new_leafsize ? 1 : 0;
    _alloc .construct(&new_top_array[cur_new_leaf+split]
                    [j-new_leafsize*split ], _top_array[i][j]);
}
}
// If this is exactly the leaf that contain n.
else if (i == n_leaf) {
    int split ;
    for (int j = 0; j < n_idx; j++) {
        split = j >= new_leafsize ? 1 : 0;
        _alloc .construct(&new_top_array[cur_new_leaf+split]
                        [j-new_leafsize*split ], _top_array[i][j]);
    }
    // Now we have to check if this is the last leaf in which
    // case we only have to copy the elements until last_idx .
    if (i == last_leaf) {
        for (int j = n_idx; j < last_idx ; j++) {
            split = j >= new_leafsize ? 1 : 0;
            _alloc .construct(&new_top_array[cur_new_leaf+split]
                            [j-new_leafsize*split ],
                            _top_array[i][j+1]);
        }

        // Now we need to construct the last elements in
        // new_top_array.
        int leaf , idx;
        if (last_idx >= new_leafsize) {
            leaf = cur_new_leaf + 1;
            idx = last_idx - new_leafsize;
            if (idx != 0) {
                for (int j = idx + 1; j < new_leafsize ; j++)
                    _alloc .construct(&new_top_array[leaf][j] , val);
            }
        }
        else {
            if (n_idx != 0) {
                for (int j = last_idx + 1; j < new_leafsize ; j++)
                    _alloc .construct(&new_top_array[cur_new_leaf][j] , val);
            }
        }
    }
    // If this is not the last leaf we copy all the elements
    // from this leaf (after the n'th position) into the new
    // leafs . When that is done we copy the first element
    // from the next leaf into the last position of the new leaf.
    else {
        for (int j = n_idx; j < _leafsize - 1; j++) {
            split = j >= new_leafsize ? 1 : 0;
            _alloc .construct(&new_top_array[cur_new_leaf+split]
                            [j-new_leafsize*split ],
                            _top_array[i][j+1]);
        }
        _alloc .construct(&new_top_array[cur_new_leaf+1]
                        [new_leafsize - 1], _top_array[i+1][0]);
    }
}
// If this is a leaf that comes after the leaf that contain n.
else {
    int split ;
    // If this is the last leaf we just copy over the elements

```

```

// from this leaf.
if (i == last_leaf) {
    for (int j = 0; j < last_idx; j++) {
        split = j >= new_leafsize ? 1 : 0;
        _alloc.construct(&new_top_array[cur_new_leaf+split]
                        [j-new_leafsize*split],
                        _top_array[i][j+1]);
    }

    // Now we construct the last elements of the new leaf.
    int leaf, idx;
    if (last_idx >= new_leafsize) {
        leaf = cur_new_leaf + 1;
        idx = last_idx - new_leafsize;
        if (idx != 0) {
            for (int j = idx + 1; j < new_leafsize; j++)
                _alloc.construct(&new_top_array[leaf][j], val);
        }
    }
    else {
        if (last_idx != 0) {
            for (int j = last_idx + 1; j < new_leafsize; j++)
                _alloc.construct(&new_top_array[cur_new_leaf][j], val);
        }
    }
}
else {
    for (int j = 0; j < _leafsize - 1; j++) {
        split = j >= new_leafsize ? 1 : 0;
        _alloc.construct(&new_top_array[cur_new_leaf+split]
                        [j-new_leafsize*split],
                        _top_array[i][j+1]);
    }
    _alloc.construct(&new_top_array[cur_new_leaf+1]
                    [new_leafsize-1], _top_array[i+1][0]);
}
}
cur_new_leaf += 2;
}
// Destroy the elements in the old top_array.
last_leaf = _capacity / _leafsize;
for (int i = 0; i < last_leaf; i++) {
    for (int j = 0; j < _leafsize; j++)
        _alloc.destroy(&_top_array[i][j]);
    _alloc.deallocate(_top_array[i], _leafsize);
}
_palloc.deallocate(_top_array, _leafsize);
_top_array = new_top_array;
_leafsize = new_leafsize;
_capacity = new_leafs * _leafsize;
_power -= 1;
_mask = _mask >> 1;
}
// If no resizing of leafs is necessary we just rearrange the
// elements in the existing structure. We just start from the
// n'th position and run forward through the leafs.
else {
    int last_leaf = (_size - 1) / _leafsize;
    int last_idx = (_size - 1) % _leafsize;
    int n_leaf = n / _leafsize;
    int start_idx = n % _leafsize;
    for (int i = n_leaf; i <= last_leaf; i++) {

```

```

// If this is the last leaf with data in it we only move
// elements until last_idx .
if (i == last_leaf) {
    for (int j = start_idx ; j < last_idx ; j++) {
        _alloc .destroy(&_top_array[i][j]);
        _alloc .construct(&_top_array[i][j], _top_array[i][j+1]);
    }
    _alloc .destroy(&_top_array[i][last_idx]);
    _alloc .construct(&_top_array[i][last_idx], val);
}
else {
    for (int j = start_idx ; j < _leafsize - 1 ; j++) {
        _alloc .destroy(&_top_array[i][j]);
        _alloc .construct(&_top_array[i][j], _top_array[i][j+1]);
    }
    // Move the first element from the next leaf into the last
    // position in this leaf . This makes the hole that is needed
    // to start rearranging the next leaf .
    _alloc .destroy(&_top_array[i][_leafsize - 1]);
    _alloc .construct(&_top_array[i][_leafsize - 1],
                    _top_array[i+1][0]);
}
start_idx = 0;
}

// Check to see if there are two empty leafs . If so we
// deallocate the last one .
last_leaf = _capacity / _leafsize - 1;
if (last_leaf >= (int)((_size - 2) / _leafsize + 2)) {
    for (int i = 0; i < _leafsize ; i++)
        _alloc .destroy(&_top_array[last_leaf][i]);
    _alloc .deallocate(_top_array[last_leaf], _leafsize);
    _capacity -= _leafsize;
}

_size -= 1;
return iterator(n, this);
}

void clear () {
    int last = _capacity / _leafsize ;
    for (int i = 0; i < last ; i++) {
        for (int j = 0; j < _leafsize ; j++)
            _alloc .destroy(&_top_array[i][j]);
        _alloc .deallocate(_top_array[i], _leafsize );
    }
    _palloc .deallocate(_top_array, _leafsize );
    _size = 0;
    _capacity = 1;
    _leafsize = 1;
    _power = 0;
    _mask = 0;
    _top_array = _palloc .allocate (1);
    _top_array[0] = _alloc .allocate (1);
    _alloc .construct (&_top_array [0][0], value_type ());
}

friend bool operator == <> (const hashed_array_tree& v1,
                             const hashed_array_tree& v2);
friend bool operator < <> (const hashed_array_tree& v1,
                             const hashed_array_tree& v2);

```

```

private:
    int _leafsize ;
    unsigned int _mask;
    unsigned int _power;
    typedef typename A::rebind<typename A::pointer>::other palloc;
    palloc _palloc ;
    allocator_type _alloc ;

protected:
    size_type _capacity ;
    size_type _size ;
    pointer *_top_array ;
};

template <class A>
bool operator == (const hashed_array_tree<A>& v1,
                 const hashed_array_tree<A>& v2) {
    if (v1._size != v2._size )
        return false;
    else {
        for (unsigned int i = 0; i < v1._size ; i++) {
            if (v1[i] != v2[i])
                return false;
        }
        return true;
    }
}

template <class A>
bool operator < (const hashed_array_tree<A>& v1,
               const hashed_array_tree<A>& v2) {
    unsigned int size = v1._size < v2._size ? v1._size : v2._size ;
    for (unsigned int i = 0; i < size; i++) {
        if (v1[i] < v2[i])
            return true;
        else if (v1[i] > v2[i])
            return false;
    }
    return (v1._size < v2._size );
}
};
#endif

```

## B.4.2 hat\_iterator.h

```

#ifndef HAT_ITERATOR_H
#define HAT_ITERATOR_H

namespace cphstl {
    template <class A> class hashed_array_tree;

    template <class A>
    class hat_iterator {
    private:
        typedef hat_iterator<A> iterator;

    public:
        typedef std::random_access_iterator_tag iterator_category ;
        typedef typename A::value_type value_type;
        typedef typename A::pointer pointer;
        typedef typename A::reference reference;

```

```

typedef typename A::difference_type difference_type;
typedef typename A::size_type size_type;

hat_iterator () : _pos(0), _hat(0) {}
hat_iterator (hashed_array_tree<A> *hat) : _pos(0), _hat(hat) {}
hat_iterator (size_type pos, hashed_array_tree<A> *hat) :
    _pos(pos), _hat(hat) {}

iterator operator ++ (int) {
    size_type x = _pos;
    _pos++;
    return iterator(x, _hat);
}

iterator operator -- (int) {
    size_type x = _pos;
    _pos--;
    return iterator(x, _hat);
}

iterator& operator ++ () {_pos++; return *this;}
iterator& operator -- () {_pos--; return *this;}

reference operator * () const {
    return _hat->_top_array[_pos >> _hat->_power][_pos & _hat->_mask];
}

iterator& operator = (const iterator& it) {
    _pos = it._pos;
    _hat = it._hat;
    return *this;
}

iterator& operator += (const difference_type i) {
    _pos += i;
    return *this;
}

iterator& operator -= (const difference_type i) {
    _pos -= i;
    return *this;
}

// These comparison operators assumes that the iterators being compared
// are iterators into the same container.
bool operator == (const iterator& it) const {return _pos == it._pos;}
bool operator < (const iterator& it) const {return _pos < it._pos;}
bool operator <= (const iterator& it) const {return _pos <= it._pos;}
bool operator > (const iterator& it) const {return _pos > it._pos;}
bool operator >= (const iterator& it) const {return _pos >= it._pos;}
bool operator != (const iterator& it) const {return _pos != it._pos;}

// There are no bounds checking on these functions.
iterator operator + (const difference_type i) const {
    return iterator(_pos + i, _hat);
}

friend iterator operator + <> (difference_type i, const iterator& it);

iterator operator - (const difference_type i) const {
    return iterator(_pos - i, _hat);
}

```

```

    difference_type operator - (const iterator& it) const {
        return _pos - it._pos;
    }

private:
    size_type _pos;
    hashed_array_tree<A> *_hat;
};

template<class A>
hat_iterator<A> operator + (typename A::difference_type i,
                           const hat_iterator<A>& it) {
    return hat_iterator<A>(it._pos + i, it._hat);
}
};
#endif

```

## B.5 Space Efficient Doubling Tree Source

### B.5.1 se\_doubling\_array.h

```

#ifndef se_doubling_tree_h
#define se_doubling_tree_h

#define TREESIZE 32

namespace cphstl {
    template <class A>
    class se_doubling_tree {
    private:
        inline static void find_index (unsigned int n, int &k, int &b, int &e) {
            unsigned int r = n + 1;
            k = log_2(r);
            int kdiv2ceil = k/2+k%2;
            unsigned int e_mask = (1<<kdiv2ceil) - 1;
            unsigned int b_mask = ((1<<k) - 1) - e_mask;
            b = (r & b_mask)>>kdiv2ceil;
            e = r & e_mask;
        }

    public:
        typedef typename A::value_type value_type;
        typedef A allocator_type;
        typedef typename A::size_type size_type;
        typedef typename A::difference_type difference_type;

        typedef sedt_iterator<A> iterator;
        typedef const sedt_iterator<A> const_iterator;
        typedef std::reverse_iterator<iterator> reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

        typedef typename A::pointer pointer;
        typedef typename A::const_pointer const_pointer;
        typedef typename A::reference reference;
        typedef typename A::const_reference const_reference;

        se_doubling_tree (const size_type c = 1, const value_type& val =
                          value_type(), const A& a = A())
            : _size (0), _alloc (a) {
            unsigned int superleafs = log_2(c); // This value is one smaller than

```

```

// the actual amount of superleafs.
unsigned int subleafs, subleaf_size, capacity;
// This loop runs through the initial (full) superleafs.
for (unsigned int i = 0; i < superleafs; i++) {
    subleafs = 1<<(i/2);
    subleaf_size = 1<<(i/2+i%2);
    _index[i] = _palloc . allocate (subleafs);
    for (unsigned int j = 0; j < subleafs; j++) {
        _index[i][j] = _alloc . allocate (subleaf_size);
        for (unsigned int k = 0; k < subleaf_size; k++)
            _alloc . construct (&_index[i][j][k], val);
    }
}

// Now need to allocate the last superleaf and as many subleafs as
// necessary to accommodate c elements.
subleaf_size = 1<<(superleafs/2+superleafs%2);
_index[superleafs] = _palloc . allocate (1<<(superleafs/2));
capacity = (1 << superleafs) - 1;

for (unsigned int i = 0; capacity < c; i++) {
    _index[superleafs][i] = _alloc . allocate (subleaf_size);
    for (unsigned int k = 0; k < subleaf_size; k++)
        _alloc . construct (&_index[superleafs][i][k], val);
    capacity += subleaf_size;
}

_capacity = capacity;
}

se_doubling_tree (const se_doubling_tree &v, const A& a = A())
: _size (v._size), _alloc (a) {
    unsigned int superleafs = log_2(_size);
    unsigned int subleafs, subleaf_size, capacity;
    for (unsigned int i = 0; i < superleafs; i++) {
        subleafs = 1<<(i/2);
        subleaf_size = 1<<(i/2+i%2);
        _index[i] = _palloc . allocate (subleafs);
        for (unsigned int j = 0; j < subleafs; j++) {
            _index[i][j] = _alloc . allocate (subleaf_size);
            for (unsigned int k = 0; k < subleaf_size; k++) {
                _alloc . construct (&_index[i][j][k], v._index[i][j][k]);
            }
        }
    }
}

// Now we only need the last superleaf. This leaf is not necessarily
// full so extra checking is needed.
subleafs = 1<<(superleafs/2);
subleaf_size = 1<<(superleafs/2+superleafs%2);
_index[superleafs] = _palloc . allocate (subleafs);
capacity = (1 << superleafs) - 1;
for (unsigned int i = 0; capacity < _size; i++) {
    _index[superleafs][i] = _alloc . allocate (subleaf_size);
    for (unsigned int j = 0; j < subleaf_size; j++)
        _alloc . construct (&_index[superleafs][i][j],
                           v._index[superleafs][i][j]);
    capacity += subleaf_size;
}
_capacity = capacity;
}

```

```

~se_doubling_tree () {
    unsigned int superleafs = log_2(_capacity);
    unsigned int subleafs, subleaf_size, capacity;
    // First we deallocate all the full superleafs.
    for (unsigned int i = 0; i < superleafs; i++) {
        subleafs = 1<<(i/2);
        subleaf_size = 1<<(i/2+i%2);
        for (unsigned int j = 0; j < subleafs; j++) {
            for (unsigned int k = 0; k < subleaf_size; k++)
                _alloc.destroy(&_index[i][j][k]);
            _alloc.deallocate(_index[i][j], subleaf_size);
        }
        _palloc.deallocate(_index[i], subleafs);
    }

    // Now we need to delete the last superleaf.
    capacity = (1<<superleafs) - 1;
    subleaf_size = 1<<(superleafs/2+superleafs%2);
    for (unsigned int i = 0; capacity < _capacity; i++) {
        for (unsigned int j = 0; j < subleaf_size; j++)
            _alloc.destroy(&_index[superleafs][i][j]);
        _alloc.deallocate(_index[superleafs][i], subleaf_size);
        capacity += subleaf_size;
    }
    _palloc.deallocate(_index[superleafs], 1<<(superleafs/2));
}

iterator begin () {return iterator(0, _index);}
iterator end () {return iterator(_size, _index);}

const_iterator begin () const {
    return const_iterator(0, (pointer**)_index);
}

const_iterator end () const {
    return const_iterator(_size, (pointer**)_index);
}

reverse_iterator rbegin () {
    return reverse_iterator(iterator(_size, _index));
}

reverse_iterator rend () {
    return reverse_iterator(iterator(0, _index));
}

const_reverse_iterator rbegin () const {
    return const_reverse_iterator(const_iterator(_size,
                                                (pointer**)_index));
}

const_reverse_iterator rend () const {
    return const_reverse_iterator(const_iterator(0, (pointer**)_index));
}

reference operator [] (size_type n) {
    int k, b, e;
    this->find_index(n,k,b,e);
    return _index[k][b][e];
}

const_reference operator [] (size_type n) const {

```

```

    int k, b, e;
    this->find_index(n,k,b,e);
    return _index[k][b][e];
}

reference front () {
    return _size == 0 ? 0 : _index [0][0];
}

const_reference front () const {
    return _size == 0 ? 0 : _index [0][0];
}

reference back () {
    if ( _size == 0) return 0;
    int k, b, e;
    this->find_index(_capacity - 1, k, b, e);
    return _index[k][b][e];
}

const_reference back () const {
    if ( _size == 0) return 0;
    int k, b, e;
    this->find_index(_capacity - 1, k, b, e);
    return _index[k][b][e];
}

se_doubling_tree& operator = (const se_doubling_tree &v) {
    // Deallocate the leaves .
    unsigned int superleafs = log_2(_capacity);
    unsigned int subleafs, subleaf_size , capacity;
    // First we deallocate all the full superleafs .
    for (unsigned int i = 0; i < superleafs; i++) {
        subleafs = 1<<(i/2);
        subleaf_size = 1<<(i/2+i%2);
        for (unsigned int j = 0; j < subleafs; j++) {
            for (unsigned int k = 0; k < subleaf_size; k++)
                _alloc .destroy(&_index[i][j][k]);
            _alloc .deallocate(_index[i][j], subleaf_size );
        }
        _palloc .deallocate (_index[i], subleafs);
    }

    // Now we need to delete the last superleaf .
    capacity = (1<<superleafs) - 1;
    subleafs = 1<<(superleafs/2);
    subleaf_size = 1<<(superleafs/2+superleafs%2);
    for (unsigned int i = 0; capacity < _capacity; i++) {
        for (unsigned int k = 0; k < subleaf_size; k++)
            _alloc .destroy(&_index[superleafs][i][k]);
        _alloc .deallocate(_index[superleafs][i], subleaf_size );
        capacity += subleaf_size;
    }
    _palloc .deallocate (_index [superleafs ], subleafs );

    // Allocate new leaves while copying the elements from v .
    _size = v . _size ;
    superleafs = log_2( _size );
    for (unsigned int i = 0; i < superleafs; i++) {
        subleafs = 1<<(i/2);
        subleaf_size = 1<<(i/2+i%2);
        _index[i] = _palloc . allocate (subleafs );
    }
}

```

```

    for (unsigned int j = 0; j < subleafs; j++) {
        _index[i][j] = _alloc . allocate ( subleaf_size );
        for (unsigned int k = 0; k < subleaf_size; k++) {
            _alloc . construct (&_index[i][j][k], v._index[i][j][k]);
        }
    }
}

// Now we only need the last superleaf. These leafs are not
// necessarily full so extra checking is needed.
subleafs = 1 << (superleafs/2);
subleaf_size = 1 << (superleafs/2 + superleafs%2);
_index[superleafs] = _palloc . allocate (subleafs);
capacity = (1 << superleafs) - 1;
for (unsigned int i = 0; capacity < _size; i++) {
    _index[superleafs][i] = _alloc . allocate ( subleaf_size );
    for (unsigned int j = 0; j < subleaf_size; j++)
        _alloc . construct (&_index[superleafs][i][j],
                            v._index[superleafs][i][j]);
    capacity += subleaf_size;
}
_capacity = capacity;

return *this;
}

// This function is not needed/supported for this vector type.
void reserve (size_type n) {
    if (n <= _capacity) return;
    int cap_superleaf, cap_subleaf, tmp, n_superleafs, n_subleafs,
        subleafs, subleaf_size;
    unsigned int capacity;
    value_type val = value_type();

    this->find_index(_capacity - 1, cap_superleaf, cap_subleaf, tmp);
    this->find_index(n - 1, n_superleafs, n_subleafs, tmp);

    // The special cases here are the first and last superleaf where
    // some of the subleafs might already be allocated or might not need
    // to be allocated. All leafs in between must be fully allocated.
    subleafs = 1 << (cap_superleaf/2);
    subleaf_size = 1 << (cap_superleaf/2 + cap_superleaf%2);

    // Handle the first case where the first leaf must be partially
    // allocated. In this case we need to take special care when the
    // first superleaf = the last superleaf.
    if (cap_superleaf == n_superleafs) {
        capacity = (1 << n_superleafs) - 1;
        for (int i = cap_subleaf + 1; i <= n_subleafs; i++) {
            _index[cap_superleaf][i] = _alloc . allocate ( subleaf_size );
            for (int j = 0; j < subleaf_size; j++)
                _alloc . construct (&_index[cap_superleaf][i][j], val);
            _capacity += subleaf_size;
        }
        return;
    }
    else {
        for (int i = cap_subleaf + 1; i < subleafs; i++) {
            _index[cap_superleaf][i] = _alloc . allocate ( subleaf_size );
            for (int j = 0; j < subleaf_size; j++)
                _alloc . construct (&_index[cap_superleaf][i][j], val);
        }
    }
}

```

```

}

// Now we allocate room for the full superleafs.
for (int i = cap_superleaf + 1; i < n_superleafs; i++) {
    subleafs = 1<<(i/2);
    subleaf_size = 1<<(i/2+i%2);
    _index[i] = _palloc . allocate (subleafs);
    for (int j = 0; j < subleafs; j++){
        _index[i][j] = _alloc . allocate (subleaf_size);
        for (int k = 0; k < subleaf_size; k++)
            _alloc . construct (&_index[i][j][k], val);
    }
}

// Now only the last superleaf is missing.
subleafs = 1<<(n_superleafs/2);
subleaf_size = 1<<(n_superleafs/2+n_superleafs%2);
_index[n_superleafs] = _palloc . allocate (subleafs);
capacity = (1<<n_superleafs)-1;
for (int i = 0; i <= n_subleafs; i++) {
    capacity += subleaf_size;
    _index[n_superleafs][i] = _alloc . allocate (subleaf_size);
    for (int j = 0; j < subleaf_size; j++)
        _alloc . construct (&_index[n_superleafs][i][j], val);
}

_capacity = capacity;
}

void swap (se_doubling_tree& v) {
    // Swap the pointers to buckets.
    int min_cap = _capacity < v._capacity ? log_2(_capacity) :
                                                log_2(v._capacity);

    pointer *tmp;
    for (int i = 0; i <= min_cap; i++) {
        tmp = _index[i];
        _index[i] = v._index[i];
        v._index[i] = tmp;
    }

    if (_capacity < v._capacity) {
        int max_cap = log_2(v._capacity);
        for (int i = min_cap + 1; i <= max_cap; i++)
            _index[i] = v._index[i];
    } else {
        int max_cap = log_2(_capacity);
        for (int i = min_cap + 1; i <= max_cap; i++)
            v._index[i] = _index[i];
    }

    // Swap the _size and _capacity information.
    size_type tmp_size = _size;
    size_type tmp_capacity = _capacity;
    _size = v._size;
    _capacity = v._capacity;
    v._size = tmp_size;
    v._capacity = tmp_capacity;
}

void push_back (const reference x) {
    // Find the position to insert x into.
    int superleaf, subleaf, index;

```

```

this->find_index(_size, superleaf, subleaf, index);

// Check if a new subleaf is needed.
if ( _size == _capacity) {
    value_type val = value_type();
    // Check to see if a new superleaf is needed. For push_back this
    // is the case when subleaf is 0 and _size == _capacity at the
    // same time.
    int subleaf_size = 1<<(superleaf/2+superleaf%2);
    if (subleaf == 0) {
        int subleafs = 1<<(superleaf/2);
        _index[superleaf] = _palloc.allocate(subleafs);
        _index[superleaf][0] = _alloc.allocate(subleaf_size);
        for (int i = 1; i < subleaf_size; i++)
            _alloc.construct(&_index[superleaf][0][i], val);
        _capacity += subleaf_size;
        _alloc.construct(&_index[superleaf][subleaf][index], x);
    }
    // Only a new subleaf is needed.
    else {
        _index[superleaf][subleaf] = _alloc.allocate(subleaf_size);
        for (int i = 1; i < subleaf_size; i++)
            _alloc.construct(&_index[superleaf][subleaf][i], val);
        _capacity += subleaf_size;
        _alloc.construct(&_index[superleaf][subleaf][index], x);
    }
}
else {
    // Insert the new element.
    _alloc.destroy(&_index[superleaf][subleaf][index]);
    _alloc.construct(&_index[superleaf][subleaf][index], x);
}
_size += 1;
}

void pop_back () {
    // Find the last elements position.
    int superleaf, subleaf, index;
    this->find_index(_size - 1, superleaf, subleaf, index);

    // Check if we need to deallocate a subleaf and possibly a superleaf.
    // For pop_back this can only be the case if index == 0 and there
    // is a subleaf with index subleaf + 1.
    if (index == 0) {
        int last_superleaf, last_subleaf, tmp;
        this->find_index(_capacity - 1, last_superleaf, last_subleaf, tmp);
        int subleafs = 1<<(last_superleaf/2);
        int subleaf_size = 1<<(last_superleaf/2+last_superleaf%2);
        if (superleaf < last_superleaf) {
            // we need to deallocate both last_superleaf and all
            // it's subleafs.
            for (int i = 0; i <= last_subleaf; i++) {
                for (int j = 0; j < subleaf_size; j++)
                    _alloc.destroy(&_index[last_superleaf][i][j]);
                _alloc.deallocate(_index[last_superleaf][i], subleaf_size);
                _capacity -= subleaf_size;
            }
            _palloc.deallocate(_index[last_superleaf], subleafs);
        }
        else if (subleaf < last_subleaf) {
            // We only need to deallocate cap_subleaf.
            for (int i = 0; i < subleaf_size; i++)

```

```

        _alloc .destroy(&_index[last_superleaf][last_subleaf][i]);
        _alloc .deallocate(_index[last_superleaf][last_subleaf],
                           subleaf_size);
        _capacity -= subleaf_size;
    }
}

// Remove the element.
_alloc .destroy(&_index[superleaf][subleaf][index]);
_alloc .construct(&_index[superleaf][subleaf][index], value_type());
_size -= 1;
}

iterator insert ( iterator pos, const value_type& x) {
    int n = pos - this->begin();
    int n_superleaf, n_subleaf, n_pos, last_superleaf, last_subleaf,
        last_pos, subleafs, subleaf_size;
    value_type val = value_type();

    // If the vector is empty we just insert the element.
    if ( _size == 0) {
        _alloc .construct(&_index [0][0][0], x);
        _size += 1;
        return iterator(0, _index);
    }

    this->find_index(n, n_superleaf, n_subleaf, n_pos);
    this->find_index(_size - 1, last_superleaf, last_subleaf, last_pos);

    // First we check if a resizing is necessary.
    if ( _size == _capacity) {
        // Check to see if a new superleaf is needed.
        subleafs = 1<<(last_superleaf/2);
        if ( last_subleaf == subleafs-1) {
            // A new superleaf is needed.
            int new_superleaf = last_superleaf + 1;
            subleafs = 1<<(new_superleaf/2);
            subleaf_size = 1<<(new_superleaf/2+new_superleaf%2);
            _index[new_superleaf] = _alloc .allocate (subleafs);
            _index[new_superleaf][0] = _alloc .allocate ( subleaf_size );
            for (int i = 0; i < subleaf_size; i++)
                _alloc .construct(&_index[new_superleaf][0][i], val);
            _capacity += subleaf_size;
        }
        else {
            // We only need to allocate a new subleaf.
            subleaf_size = 1<<(last_superleaf/2+last_superleaf%2);
            int leaf = last_subleaf+1;
            _index[ last_superleaf ][ leaf ] = _alloc .allocate ( subleaf_size );
            for (int i = 0; i < subleaf_size; i++)
                _alloc .construct(&_index[ last_superleaf ][ leaf ][i], val);
            _capacity += subleaf_size;
        }
    }

    // If the element inserted is in the last position of the vector
    // we can just insert it now. So we only start the move-part of
    // the code if this if not the case. This checks wether the
    // insertion is in another subleaf of the same superleaf. If the
    // insertion is in another (higher-indexed) superleaf it will be
    // dropped because of the condition in the first for-loop.
    if (!( ( last_superleaf == n_superleaf && last_subleaf < n_subleaf) ||

```





```

        last_subleaf < cap_subleaf)) {
    // We only need to deallocate cap_subleaf.
    for (int i = 0; i < subleaf_size; i++)
        _alloc .destroy(&_index[cap_superleaf][cap_subleaf][i]);
    _alloc .deallocate(_index[cap_superleaf][cap_subleaf],
                      subleaf_size);
    _capacity -= subleaf_size;
}
}

subleafs = 1<<(n_superleaf/2);
subleaf_size = 1<<(n_superleaf/2+n_superleaf%2);

// Now we move all elements in front of pos one position to the left.
// We move the elements in the first subleaf separately because
// in this case the moving does not start from 0. We have to
// check if the first subleaf is also the last one and act
// accordingly if this is the case.
if (last_superleaf == n_superleaf && last_subleaf == n_subleaf) {
    for (int i = n_pos; i < last_pos; i++) {
        _alloc .destroy(&_index[n_superleaf][n_subleaf][i]);
        _alloc .construct(&_index[n_superleaf][n_subleaf][i],
                        _index[n_superleaf][n_subleaf][i+1]);
    }
    _alloc .destroy(&_index[last_superleaf][last_subleaf][last_pos]);
    _alloc .construct(&_index[last_superleaf][last_subleaf][last_pos],
                    val);
    _size -= 1;
    return iterator(n, _index);
}
else {
    for (int i = n_pos; i < subleaf_size - 1; i++) {
        _alloc .destroy(&_index[n_superleaf][n_subleaf][i]);
        _alloc .construct(&_index[n_superleaf][n_subleaf][i],
                        _index[n_superleaf][n_subleaf][i+1]);
    }
    // The last element must be copied from the next subleaf.
    if (n_subleaf == subleafs - 1) {
        // The element can be found in the next superleaf.
        _alloc .destroy(&_index[n_superleaf][n_subleaf][subleaf_size-1]);
        _alloc .construct(&_index[n_superleaf][n_subleaf][subleaf_size-1],
                        _index[n_superleaf+1][0][0]);
        n_subleaf = 0;
        n_superleaf++;
    }
    else {
        // The element resides in the subleaf at position n_subleaf+1.
        _alloc .destroy(&_index[n_superleaf][n_subleaf][subleaf_size-1]);
        _alloc .construct(&_index[n_superleaf][n_subleaf][subleaf_size-1],
                        _index[n_superleaf][n_subleaf+1][0]);
        n_subleaf++;
    }
}
}

// Now we can move the rest of the element until last_pos.
for (int i = n_superleaf; i <= last_superleaf; i++) {
    subleafs = 1<<(i/2);
    subleaf_size = 1<<(i/2+i%2);
    if (i == last_superleaf) {
        // Move the elements until the last_subleaf.
        for (int j = n_subleaf; j < last_subleaf; j++) {
            for (int k = 0; k < subleaf_size - 1; k++) {

```

```

        _alloc .destroy(&_index[i][j][k]);
        _alloc .construct(&_index[i][j][k], _index[i][j][k+1]);
    }
    if (j == subleafs - 1) {
        _alloc .destroy(&_index[i][j][subleaf_size - 1]);
        _alloc .construct(&_index[i][j][subleaf_size - 1],
            _index[i + 1][0][0]);
    }
    else {
        _alloc .destroy(&_index[i][j][subleaf_size - 1]);
        _alloc .construct(&_index[i][j][subleaf_size - 1],
            _index[i][j + 1][0]);
    }
}

// Move the elements in the last_subleaf .
for (int j = 0; j < last_pos; j++) {
    _alloc .destroy(&_index[i][last_subleaf][j]);
    _alloc .construct(&_index[i][last_subleaf][j],
        _index[i][last_subleaf][j + 1]);
}
}
else {
    for (int j = n_subleaf; j < subleafs; j++) {
        for (int k = 0; k < subleaf_size - 1; k++) {
            _alloc .destroy(&_index[i][j][k]);
            _alloc .construct(&_index[i][j][k], _index[i][j][k + 1]);
        }
        if (j == subleafs - 1) {
            _alloc .destroy(&_index[i][j][subleaf_size - 1]);
            _alloc .construct(&_index[i][j][subleaf_size - 1],
                _index[i + 1][0][0]);
        }
        else {
            _alloc .destroy(&_index[i][j][subleaf_size - 1]);
            _alloc .construct(&_index[i][j][subleaf_size - 1],
                _index[i][j + 1][0]);
        }
    }
}
n_subleaf = 0;
}

_alloc .destroy(&_index[last_superleaf][last_subleaf][last_pos]);
_alloc .construct(&_index[last_superleaf][last_subleaf][last_pos],
    val);
_size -= 1;
return iterator(n, _index);
}

void clear () {
    unsigned int superleafs = log_2(capacity);
    unsigned int subleafs, subleaf_size, capacity;
    // First we deallocate all the full superleafs .
    for (unsigned int i = 1; i < superleafs; i++) {
        subleafs = 1 << (i/2);
        subleaf_size = 1 << (i/2 + i%2);
        for (unsigned int j = 0; j < subleafs; j++) {
            for (unsigned int k = 0; k < subleaf_size; k++)
                _alloc .destroy(&_index[i][j][k]);
            _alloc .deallocate(_index[i][j], subleaf_size);
        }
    }
}

```

```

    _palloc . deallocate ( _index [ i ], subleafs );
}

// Now we need to delete the last superleaf.
capacity = ( 1 << superleafs ) - 1;
subleafs = 1 << ( superleafs / 2 );
subleaf_size = 1 << ( superleafs / 2 + superleafs % 2 );
for ( unsigned int i = 0; capacity < _capacity; i++ ) {
    for ( unsigned int j = 0; j < subleaf_size; j++ )
        _alloc . destroy ( & _index [ superleafs ] [ i ] [ j ] );
    _alloc . deallocate ( _index [ superleafs ] [ i ], subleaf_size );
    capacity += subleaf_size;
}
_palloc . deallocate ( _index [ superleafs ], subleafs );

_alloc . destroy ( & _index [ 0 ] [ 0 ] [ 0 ] );
_alloc . construct ( & _index [ 0 ] [ 0 ] [ 0 ], value_type ( ) );
_capacity = 1;
_size = 0;
}

friend bool operator == < A > ( const se_doubling_tree & v1,
                                const se_doubling_tree & v2 );
friend bool operator < < A > ( const se_doubling_tree & v1,
                               const se_doubling_tree & v2 );

protected:
    size_type _capacity;
    size_type _size;

private:
    pointer *_index [ TREESIZE ];
    typedef typename A::rebind< typename A::pointer >::other palloc;
    palloc _palloc;
    allocator_type _alloc;
};

template < class A >
bool operator == ( const se_doubling_tree < A > & v1,
                  const se_doubling_tree < A > & v2 ) {
    if ( v1 . _size != v2 . _size )
        return false;
    else {
        int last_superleaf , last_subleaf , last_elem , subleafs , subleaf_size ;
        v1 . find_index ( v1 . _size - 1 , last_superleaf , last_subleaf , last_elem );
        for ( int i = 0; i < last_superleaf; i++ ) {
            subleafs = 1 << ( i / 2 );
            subleaf_size = 1 << ( i / 2 + i % 2 );
            for ( int j = 0; j < subleafs; j++ ) {
                for ( int k = 0; k < subleaf_size; k++ ) {
                    if ( v1 . _index [ i ] [ j ] [ k ] != v2 . _index [ i ] [ j ] [ k ] ) return false;
                }
            }
        }
        subleaf_size = 1 << ( last_superleaf / 2 + last_superleaf % 2 );
        for ( int i = 0; i <= last_subleaf; i++ ) {
            if ( i == last_subleaf ) {
                for ( int j = 0; j <= last_elem; j++ ) {
                    if ( v1 . _index [ last_superleaf ] [ i ] [ j ] !=
                        v2 . _index [ last_superleaf ] [ i ] [ j ] ) return false;
                }
            }
        }
    }
}

```

```

        else {
            for (int j = 0; j < subleaf_size ; j++){
                if (v1._index[ last_superleaf ][i][j] !=
                    v2._index[ last_superleaf ][i][j]) return false;
            }
        }
    }
    return true;
}

template <class A>
bool operator < (const se.doubling_tree<A>& v1,
                const se.doubling_tree<A>& v2) {
    int last_superleaf , last_subleaf , last_elem , subleafs , subleaf_size ;
    v1.find_index(v1._size - 1, last_superleaf , last_subleaf , last_elem );
    for (int i = 0; i < last_superleaf ; i++) {
        subleafs = 1<<(i/2);
        subleaf_size = 1<<(i/2+i%2);
        for (int j = 0; j < subleafs; j++) {
            for (int k = 0; k < subleaf_size ; k++) {
                if (v1._index[i][j][k] < v2._index[i][j][k])
                    return true;
                else if (v1._index[i][j][k] > v2._index[i][j][k])
                    return false;
            }
        }
    }
    subleaf_size = 1<<(last_superleaf/2+last_superleaf%2);
    for (int i = 0; i <= last_subleaf; i++) {
        if (i == last_subleaf){
            for (int j = 0; j <= last_elem; j++){
                if (v1._index[ last_superleaf ][i][j] <
                    v2._index[ last_superleaf ][i][j])
                    return true;
                else if (v1._index[ last_superleaf ][i][j] >
                    v2._index[ last_superleaf ][i][j])
                    return false;
            }
        }
        else {
            for (int j = 0; j < subleaf_size ; j++){
                if (v1._index[ last_superleaf ][i][j] <
                    v2._index[ last_superleaf ][i][j])
                    return true;
                else if (v1._index[ last_superleaf ][i][j] >
                    v2._index[ last_superleaf ][i][j])
                    return false;
            }
        }
    }
    return (v1._size < v2._size );
}
};
#endif

```

## B.5.2 sedt\_iterator.h

```

#ifndef SEDT_ITERATOR_H
#define SEDT_ITERATOR_H

```

```

namespace cphstl {
template <class A>
class sedt_iterator {
private:
    typedef sedt_iterator<A> iterator;

public:
    typedef std::random_access_iterator_tag iterator_category ;
    typedef typename A::value_type value_type;
    typedef typename A::pointer pointer;
    typedef typename A::reference reference;
    typedef typename A::difference_type difference_type;
    typedef typename A::size_type size_type;

    sedt_iterator () : _pos(0), _index(0) {}
    sedt_iterator (pointer **index) : _pos(0), _index(index) {}
    sedt_iterator (size_type pos, pointer **index) :
        _pos(pos), _index(index) {}

    iterator operator ++ (int) {
        size_type x = _pos;
        _pos++; return iterator(x, _index);
    }

    iterator operator -- (int) {
        size_type x = _pos;
        _pos--; return iterator(x, _index);
    }

    iterator & operator ++ () {_pos++; return *this;}
    iterator & operator -- () {_pos--; return *this;}

    reference operator * () const {
        int r = _pos + 1;
        int k = log_2(r);
        int kdiv2ceil = k/2+k%2;
        unsigned int e_mask = (1<<kdiv2ceil) - 1;
        unsigned int b_mask = ((1<<k) - 1) - e_mask;
        int b = (r & b_mask)>>kdiv2ceil;
        int e = r & e_mask;

        return _index[k][b][e];
    }

    iterator & operator = (const iterator& it) {
        _pos = it._pos;
        _index = it._index;
        return *this;
    }

    iterator & operator += (const difference_type i) {
        _pos += i;
        return *this;
    }

    iterator & operator -= (const difference_type i) {
        _pos -= i;
        return *this;
    }

    // These comparison operators assumes that the iterators being compared

```

```

// are iterators into the same container.
bool operator == (const iterator& it) const {return _pos == it._pos;}
bool operator < (const iterator& it) const {return _pos < it._pos;}
bool operator <= (const iterator& it) const {return _pos <= it._pos;}
bool operator > (const iterator& it) const {return _pos > it._pos;}
bool operator >= (const iterator& it) const {return _pos >= it._pos;}
bool operator != (const iterator& it) const {return _pos != it._pos;}

// There are no bounds checking on these functions.
iterator operator + (const difference_type i) const {
    return iterator(_pos + i, _index);
}

friend iterator operator + <A> (difference_type i, const iterator& it);

iterator operator - (const difference_type i) const {
    return iterator(_pos - i, _index);
}

difference_type operator - (const iterator& it) const {
    return _pos - it._pos;
}

private:
    size_type _pos;
    pointer **_index;
};

template<class A>
sedt_iterator<A>
operator + (typename A::difference_type i, const sedt_iterator<A>& it) {
    return *new sedt_iterator<A>(it._pos + i, it._index);
}
};

#endif

```

## C Benchmark Scripts

### C.1 Lookup Tests

#### C.1.1 lookup\_bench.cc

```

#include <cstdlib>

template <class V>
class lookup_bench {
public:
    lookup_bench (int n, int m) {
        this->n = n;
        this->m = m;
        for (int i = 0; i < n; i++)
            v.push_back(i);
    }

    void primal_init () {
        int nonsense[250000];
        for (int i = 0; i < 250000; i++)
            nonsense[i] = i;
    }
}

```

```

void primal () {
    for (int i = 0; i < m; i++)
        v[unsigned(std::rand()) % n];
}

void primal_clean () {
}

private:
    int n, m;
    V v;
};

```

### C.1.2 lookup\_benchmark.py

```

#!/usr/bin/python
import benchmark
import os

VECTOR = os.environ['CPHSTL'] + '/Program/Vector/vector'
LOOKUPS = 10000

class superclass(benchmark.case):
    def __init__(self):
        benchmark.case.__init__(self)
        self.n = 0
        self.compiler_options.append("-O3")
        self.include_files.extend([VECTOR, 'lookup_bench.cc'])
        self.time_unit = 's'
        self.dual_exists = 0

    def output(self):
        return (self.n, self.driver_output)

class lookup_da(superclass):
    def __init__(self, n, m):
        superclass.__init__(self)
        self.n = n
        self.constructor_call = 'lookup_bench<cphstl::vector<int, \
std::allocator<int>, cphstl::doubling_array<std::allocator<int> \
>>>(<'+str(n)+'+', '<'+str(m)+'+')'
        self.driver_file = self.generate_cpu_time_driver()

class lookup_dt(superclass):
    def __init__(self, n, m):
        superclass.__init__(self)
        self.n = n
        self.constructor_call = 'lookup_bench<cphstl::vector<int, \
std::allocator<int>, cphstl::doubling_tree<std::allocator<int> \
>>>(<'+str(n)+'+', '<'+str(m)+'+')'
        self.driver_file = self.generate_cpu_time_driver()

class lookup_hat(superclass):
    def __init__(self, n, m):
        superclass.__init__(self)
        self.n = n
        self.constructor_call = 'lookup_bench<cphstl::vector<int, \
std::allocator<int>, cphstl::hashed_array_tree<std::allocator<int> \
>>>(<'+str(n)+'+', '<'+str(m)+'+')'
        self.driver_file = self.generate_cpu_time_driver()

```

```

class lookup_sedt(superclass):
    def __init__(self, n, m):
        superclass.__init__(self)
        self.n = n
        self.constructor_call = 'lookup_bench<cphstl::vector<int,\
std::allocator<int>, cphstl::se_doubling_tree<std::allocator<int> \
>>>(<'+str(n)+'>,<'+str(m)+'>)'
        self.driver_file = self.generate_cpu_time_driver()

class da_curve(benchmark.curve_suite):
    def __init__(self):
        benchmark.curve_suite.__init__(self)
        self.title = 'Doubling_Array'
        for x in range(250000, 5250000, 250000):
            self.add(lookup_da(x, LOOKUPS))

class dt_curve(benchmark.curve_suite):
    def __init__(self):
        benchmark.curve_suite.__init__(self)
        self.title = 'Doubling_Tree'
        for x in range(250000, 5250000, 250000):
            self.add(lookup_dt(x, LOOKUPS))

class hat_curve(benchmark.curve_suite):
    def __init__(self):
        benchmark.curve_suite.__init__(self)
        self.title = 'Hashed_Array_Tree'
        for x in range(250000, 5250000, 250000):
            self.add(lookup_hat(x, LOOKUPS))

class sedt_curve(benchmark.curve_suite):
    def __init__(self):
        benchmark.curve_suite.__init__(self)
        self.title = 'SE_Doubling_Tree'
        for x in range(250000, 5250000, 250000):
            self.add(lookup_sedt(x, LOOKUPS))

class myplot(benchmark.plot_suite):
    def __init__(self):
        benchmark.plot_suite.__init__(self)
        self.output = 'lookup_bench.ps'
        self.title = '10.000_lookup_operations'
        self.xlabel = 'Vector_Size'
        self.ylabel = 'Time(sec)'
        self.add(da_curve())
        self.add(dt_curve())
        self.add(hat_curve())
        self.add(sedt_curve())
        self.set_commands()

if __name__ == '__main__':
    benchmark.main(task = myplot(), runner = benchmark.gnuplot_runner)

```

### C.1.3 lookup\_benchmark\_papi.py

```

#!/usr/bin/python
import benchmark
import os
import string

VECTOR = os.environ['CPHSTL'] + '/Program/Vector/vector'
LOOKUPS = 10000

```

```

class superclass(benchmark.case):
    def __init__( self ):
        benchmark.case.__init__( self )
        self .n = 0
        self .compiler_options.append( "-O3" )
        self .include_files.extend( [VECTOR, 'lookup_bench.cc'] )
        self .time_unit = 's'
        self .dual_exists = 0
        self .papi_path = '/usr/local'
        self .papi_type = 'single'
        self .time_unit = 's'
        self .dual_exists = 0

    def output( self ):
        x = self .driver_output.split( ";" ) [2]
        return ( self .n, x )

class lookup_da( superclass ):
    def __init__( self , n, m ):
        superclass.__init__( self )
        self .n = n
        self .constructor_call = 'lookup_bench<cphstl::vector<int, \
std::allocator<int>, cphstl::doubling_array<std::allocator<int> \
>>>( '+str(n)+' , '+str(m)+' )'
        self .driver_file = self .generate_execution_mem_driver()

class lookup_dt( superclass ):
    def __init__( self , n, m ):
        superclass.__init__( self )
        self .n = n
        self .constructor_call = 'lookup_bench<cphstl::vector<int, \
std::allocator<int>, cphstl::doubling_tree<std::allocator<int> \
>>>( '+str(n)+' , '+str(m)+' )'
        self .driver_file = self .generate_execution_mem_driver()

class lookup_hat( superclass ):
    def __init__( self , n, m ):
        superclass.__init__( self )
        self .n = n
        self .constructor_call = 'lookup_bench<cphstl::vector<int, \
std::allocator<int>, cphstl::hashed_array_tree<std::allocator<int> \
>>>( '+str(n)+' , '+str(m)+' )'
        self .driver_file = self .generate_execution_mem_driver()

class lookup_sedt( superclass ):
    def __init__( self , n, m ):
        superclass.__init__( self )
        self .n = n
        self .constructor_call = 'lookup_bench<cphstl::vector<int, \
std::allocator<int>, cphstl::se_doubling_tree<std::allocator<int> \
>>>( '+str(n)+' , '+str(m)+' )'
        self .driver_file = self .generate_execution_mem_driver()

class da_curve( benchmark.curve_suite ):
    def __init__( self ):
        benchmark.curve_suite.__init__( self )
        self .title = 'Doubling_Array'
        for x in range( 250000, 5250000, 250000 ):
            self .add( lookup_da( x, LOOKUPS ) )

class dt_curve( benchmark.curve_suite ):

```

```

def __init__( self ):
    benchmark.curve_suite.__init__( self )
    self . title = 'Doubling_Tree'
    for x in range(250000, 5250000, 250000):
        self .add(lookup_dt(x, LOOKUPS))

class hat_curve(benchmark.curve_suite):
    def __init__( self ):
        benchmark.curve_suite.__init__( self )
        self . title = 'Hashed_Array_Tree'
        for x in range(250000, 5250000, 250000):
            self .add(lookup_hat(x, LOOKUPS))

class sedt_curve(benchmark.curve_suite):
    def __init__( self ):
        benchmark.curve_suite.__init__( self )
        self . title = 'SE_Doubling_Tree'
        for x in range(250000, 5250000, 250000):
            self .add(lookup_sedt(x, LOOKUPS))

class myplot(benchmark.plot_suite):
    def __init__( self ):
        benchmark.plot_suite.__init__( self )
        self . output = 'lookup_bench_papi.ps'
        self . title = '10.000_lookup_operations'
        self . xlabel = 'Vector_Size'
        self . ylabel = 'Cache_Misses'
        self .add(da_curve())
        self .add(dt_curve())
        self .add(hat_curve())
        self .add(sedt_curve())
        self .set_commands()

if __name__=='__main__':
    benchmark.main(task = myplot(), runner = benchmark.gnuplot_runner)

```

## C.2 push\_back Test

### C.2.1 push\_back\_bench.cc

```

template <class V>
class push_back_bench {
public:
    push_back_bench (int n) {
        this->n = n;
    }

    void primal_init () {
    }

    void primal () {
        for (int i = 0; i < n; i++)
            v.push_back(i);
    }

    void primal_clean () {
        v.clear ();
    }

private:
    int n;
    V v;

```

```
};
```

### C.2.2 push\_back\_benchmark.py

```
#!/usr/bin/python
import benchmark
import os

VECTOR = os.environ['CPHSTL'] + '/Program/Vector/vector'

class superclass(benchmark.case):
    def __init__(self):
        benchmark.case.__init__(self)
        self.n = 0;
        self.compiler_options.append("-O3")
        self.include_files.extend([VECTOR,'push_back_bench.cc'])
        self.time_unit = 's'
        self.dual_exists = 0

    def output(self):
        return (self.n, self.driver_output)

class push_back_da(superclass):
    def __init__(self, n):
        superclass.__init__(self)
        self.n = n
        self.constructor_call = 'push_back_bench<cphstl::vector<int,\
std::allocator<int>, cphstl::doubling_array<std::allocator<int> \
>>>(' + str(n) + ')''
        self.driver_file = self.generate_cpu_time_driver()

class push_back_dt(superclass):
    def __init__(self, n):
        superclass.__init__(self)
        self.n = n
        self.constructor_call = 'push_back_bench<cphstl::vector<int,\
std::allocator<int>, cphstl::doubling_tree<std::allocator<int> \
>>>(' + str(n) + ')''
        self.driver_file = self.generate_cpu_time_driver()

class push_back_hat(superclass):
    def __init__(self, n):
        superclass.__init__(self)
        self.n = n
        self.constructor_call = 'push_back_bench<cphstl::vector<int,\
std::allocator<int>, cphstl::hashed_array_tree<std::allocator<int> \
>>>(' + str(n) + ')''
        self.driver_file = self.generate_cpu_time_driver()

class push_back_sedt(superclass):
    def __init__(self, n):
        superclass.__init__(self)
        self.n = n
        self.constructor_call = 'push_back_bench<cphstl::vector<int,\
std::allocator<int>, cphstl::se_doubling_tree<std::allocator<int> \
>>>(' + str(n) + ')''
        self.driver_file = self.generate_cpu_time_driver()

class da_curve(benchmark.curve_suite):
    def __init__(self):
        benchmark.curve_suite.__init__(self)
        self.title = 'Doubling_Array'
```

```

        for x in range(100000, 2100000, 100000):
            self.add(push_back_da(x))

class dt_curve(benchmark.curve_suite):
    def __init__( self ):
        benchmark.curve_suite.__init__( self )
        self.title = 'Doubling_Tree'
        for x in range(100000, 2100000, 100000):
            self.add(push_back_dt(x))

class hat_curve(benchmark.curve_suite):
    def __init__( self ):
        benchmark.curve_suite.__init__( self )
        self.title = 'Hashed_Array_Tree'
        for x in range(100000, 2100000, 100000):
            self.add(push_back_hat(x))

class sedt_curve(benchmark.curve_suite):
    def __init__( self ):
        benchmark.curve_suite.__init__( self )
        self.title = 'SE_Doubling_Tree'
        for x in range(100000, 2100000, 100000):
            self.add(push_back_sedt(x))

class myplot(benchmark.plot_suite):
    def __init__( self ):
        benchmark.plot_suite.__init__( self )
        self.output = 'push_back_bench.ps'
        self.title = 'push_back'
        self.xlabel = 'Operations'
        self.ylabel = 'Time_(sec)'
        self.add(da_curve())
        self.add(dt_curve())
        self.add(hat_curve())
        self.add(sedt_curve())
        self.set_commands()

if __name__=='__main__':
    benchmark.main(task = myplot(), runner = benchmark.gnuplot_runner)

```

## C.3 pop\_back Test

### C.3.1 pop\_back\_bench.cc

```

template <class V>
class pop_back_bench {
public:
    pop_back_bench (int n) {
        this->n = n;
    }

    void primal_init () {
        for (int i = 0; i < n; i++)
            v.push_back(i);
    }

    void primal () {
        for (int i = 0; i < n; i++)
            v.pop_back();
    }

    void primal_clean () {

```

```

    }

private:
    int n;
    V v;
};

```

### C.3.2 pop\_back\_benchmark.py

```

#!/usr/bin/python
import benchmark
import os

VECTOR = os.environ['CPHSTL'] + '/Program/Vector/vector'

class superclass(benchmark.case):
    def __init__(self):
        benchmark.case.__init__(self)
        self.n = 0;
        self.compiler_options.append("-O3")
        self.include_files.extend([VECTOR,'pop_back_bench.cc'])
        self.time_unit = 's'
        self.dual_exists = 0

    def output(self):
        return (self.n, self.driver_output)

class pop_back_da(superclass):
    def __init__(self, n):
        superclass.__init__(self)
        self.n = n
        self.constructor_call = 'pop_back_bench<cphstl::vector<int, \
std::allocator<int>, cphstl::doubling_array<std::allocator<int> \
>>>(' + str(n) + ')''
        self.driver_file = self.generate_cpu_time_driver()

class pop_back_dt(superclass):
    def __init__(self, n):
        superclass.__init__(self)
        self.n = n
        self.constructor_call = 'pop_back_bench<cphstl::vector<int, \
std::allocator<int>, cphstl::doubling_tree<std::allocator<int> \
>>>(' + str(n) + ')''
        self.driver_file = self.generate_cpu_time_driver()

class pop_back_hat(superclass):
    def __init__(self, n):
        superclass.__init__(self)
        self.n = n
        self.constructor_call = 'pop_back_bench<cphstl::vector<int, \
std::allocator<int>, cphstl::hashed_array_tree<std::allocator<int> \
>>>(' + str(n) + ')''
        self.driver_file = self.generate_cpu_time_driver()

class pop_back_sedt(superclass):
    def __init__(self, n):
        superclass.__init__(self)
        self.n = n
        self.constructor_call = 'pop_back_bench<cphstl::vector<int, \
std::allocator<int>, cphstl::se_doubling_tree<std::allocator<int> \
>>>(' + str(n) + ')''
        self.driver_file = self.generate_cpu_time_driver()

```

```

class da_curve(benchmark.curve_suite):
    def __init__( self ):
        benchmark.curve_suite.__init__( self )
        self . title = 'Doubling_Array'
        for x in range(100000, 1100000, 100000):
            self . add(pop_back_da(x))

class dt_curve(benchmark.curve_suite):
    def __init__( self ):
        benchmark.curve_suite.__init__( self )
        self . title = 'Doubling_Tree'
        for x in range(100000, 1100000, 100000):
            self . add(pop_back_dt(x))

class hat_curve(benchmark.curve_suite):
    def __init__( self ):
        benchmark.curve_suite.__init__( self )
        self . title = 'Hashed_Array_Tree'
        for x in range(100000, 1100000, 100000):
            self . add(pop_back_hat(x))

class sedt_curve(benchmark.curve_suite):
    def __init__( self ):
        benchmark.curve_suite.__init__( self )
        self . title = 'SE_Doubling_Tree'
        for x in range(100000, 1100000, 100000):
            self . add(pop_back_sedt(x))

class myplot(benchmark.plot_suite):
    def __init__( self ):
        benchmark.plot_suite.__init__( self )
        self . output = 'pop_back_bench.ps'
        self . title = 'pop_back'
        self . xlabel = 'Operations'
        self . ylabel = 'Time_(sec)'
        self . add(da_curve())
        self . add(dt_curve())
        self . add(hat_curve())
        self . add(sedt_curve())
        self . set_commands()

if __name__=='__main__':
    benchmark.main(task = myplot(), runner = benchmark.gnuplot_runner)

```

## C.4 Scan Test

### C.4.1 scan\_bench.cc

```

#include <cstdlib>

template <class V>
class scan_bench {
public:
    scan_bench (int n) {
        this->n = n;
    }

    void primal_init () {
        for (int i = 0; i < n; i++)
            v.push_back(i);
    }

```

```

    // Fill the cache with nonsense.
    int nonsense[250000];
    int x;
    for (int i = 0; i < 250000; i++)
        x = nonsense[i];
}

void primal () {
    value_type x;
    iterator it = v.begin(), end = v.end();
    while (it < end)
        x = *it++;
}

void primal_clean () {
    v.clear ();
}

private:
    int n;
    V v;
    typedef typename V::value_type value_type;
    typedef typename V::iterator iterator;
};

```

#### C.4.2 scan\_benchmark\_papi.py

```

#!/usr/bin/python
import benchmark
import os
import string

VECTOR = os.environ['CPHSTL'] + '/Program/Vector/vector'

class superclass(benchmark.case):
    def __init__( self ):
        benchmark.case.__init__( self )
        self.n = 0
        self.compiler_options.append("-O3")
        self.include_files.extend([VECTOR,'scan_bench.cc'])
        self.time_unit = 's'
        self.dual_exists = 0
        self.papi_path = '/usr/local'
        self.papitype = 'single'
        self.time_unit = 's'
        self.dual_exists = 0

    def output(self):
        x = self.driver_output.split(";")[2]
        return (self.n, x)

class scan_da(superclass):
    def __init__( self , n):
        superclass.__init__( self )
        self.n = n
        self.constructor_call = 'scan_bench<cphstl::vector<int,\
std::allocator<int>, cphstl::doubling_array<std::allocator<int> \
>>>(<'+str(n)+'>)'
        self.driver_file = self.generate_execution_mem_driver()

class scan_dt(superclass):
    def __init__( self , n):

```

```

    superclass . _init_ ( self )
    self . n = n
    self . constructor_call = 'scan_bench<cphstl::vector<int,\
std::allocator<int>, cphstl::doubling_tree<std::allocator<int> \
>>>(' + str(n) + ')'
```

```

    self . driver_file = self . generate_execution_mem_driver()

class scan_hat(superclass):
    def _init_ ( self , n):
        superclass . _init_ ( self )
        self . n = n
        self . constructor_call = 'scan_bench<cphstl::vector<int,\
std::allocator<int>, cphstl::hashed_array_tree<std::allocator<int> \
>>>(' + str(n) + ')'
```

```

    self . driver_file = self . generate_execution_mem_driver()

class scan_sedt(superclass):
    def _init_ ( self , n):
        superclass . _init_ ( self )
        self . n = n
        self . constructor_call = 'scan_bench<cphstl::vector<int,\
std::allocator<int>, cphstl::se_doubling_tree<std::allocator<int> \
>>>(' + str(n) + ')'
```

```

    self . driver_file = self . generate_execution_mem_driver()

class da_curve(benchmark.curve_suite):
    def _init_ ( self ):
        benchmark.curve_suite . _init_ ( self )
        self . title = 'Doubling_Array'
        for x in range(250000, 5250000, 250000):
            self . add(scan_da(x))

class dt_curve(benchmark.curve_suite):
    def _init_ ( self ):
        benchmark.curve_suite . _init_ ( self )
        self . title = 'Doubling_Tree'
        for x in range(250000, 5250000, 250000):
            self . add(scan_dt(x))

class hat_curve(benchmark.curve_suite):
    def _init_ ( self ):
        benchmark.curve_suite . _init_ ( self )
        self . title = 'Hashed_Array_Tree'
        for x in range(250000, 5250000, 250000):
            self . add(scan_hat(x))

class sedt_curve(benchmark.curve_suite):
    def _init_ ( self ):
        benchmark.curve_suite . _init_ ( self )
        self . title = 'SE_Doubling_Tree'
        for x in range(250000, 5250000, 250000):
            self . add(scan_sedt(x))

class myplot(benchmark.plot_suite):
    def _init_ ( self ):
        benchmark.plot_suite . _init_ ( self )
        self . output = 'scan_bench_papi.ps'
        self . title = 'Full_Scan'
        self . xlabel = 'Vector_Size'
        self . ylabel = 'Cache_Misses'
        self . add(da_curve())
        self . add(dt_curve())

```

```
        self .add(hat_curve())
        self .add(sedt_curve())
        self .set_commands()

if __name__ == '__main__':
    benchmark.main(task = myplot(), runner = benchmark.gnuplot_runner)
```