Abstract. This paper has been produced as a solution to the mini-project Associate containers with strong guarantees in the course Generic Programming and Library Development quarter 4 2006/07 at the Institute of Computer Science, University of Copenhagen. A datastructure called Deterministic Skip List has been implemented to be used in associative containers, such as sets and maps. Functionality to insert and erase using iterators have not been implemented.

1. Introduction

In this assignment, the task is to write an associative container along with the associated data structure. We are given a list of possible data structures and a number of conflicting requirements for the finished product. Our job is to make the hard choices regarding these requirements and end up with a subset that we would like to focus on. Several of the requirements are absolute; they are given by the C++ standard and must be met. Others depend on the choice of data structures and are mutually exclusive.

2. Goals

The choices we have made are motivated by the idea that trees and variations on them are well known. We have chosen a relatively new data structure, namely the skip list (first described in [4] in 1990). The idea of the skip list is that we introduce a kind of “fast lane” in an otherwise standard linked list to let us progress through the list at an adaptable speed. It is described more thoroughly in section 3.

The specific goal that this data structure can achieve is referred to in the assignment text as $R_5$. Aside from this goal, we also provide solutions for most of $R_1-R_4$, as required. The linear time construction requirement in $R_4$ has not been met, but it is clear that we could do so given a suitable algorithm.

The project formulation is vague in the sense that it is not entirely clear whether we need to implement both a container class and a data structure. We have chosen to rely on the distributed code for a container class, thinking that rewriting it would only be relevant to some of the goals that we have

Written 18 June 2007.
chosen not to pursue. We have attempted to avoid making any changes to the set class. As such, our deliverable code consists of source and header files for the skip list class. To the extent that any changes are required in the set class, we simply describe those.

3. Deterministic Skip Lists

Skip Lists were introduced by Pugh [4] as a probabilistic alternative to binary search trees. Instead of explicitly maintaining a balanced search tree, Pugh proposed using a probabilistic approach where it is likely that balance will be maintained: When inserting a new key into the skip list, the height (see figure 1) of the new node is randomly chosen. This gives a simple algorithm for inserting and deleting, as it’s just updating the pointers, and no re-balancing is needed.

The drawback of skip lists is the worst case time complexity of the insert and delete methods. The time complexity of these methods is linear in the height of the list. As the height of each node is decided by probabilities, the height is unbounded in the worst case. Assuming a maximum height of nodes the time complexity could be linear with the number of nodes in the list [2]. However, the more nodes in the skip list, the smaller is the probability that we get a linear time complexity. On average the time complexity is logarithmic in the number of nodes.

To get around this horrible worst case time complexity, Munro, Papadakis and Sedgewick proposed using Deterministic Skip Lists [2]. The basic data structure is the same as in the probabilistic skip lists, but the insertion and deletion algorithms are altered so that they maintain a balanced skip list. This complicates these algorithms a little bit, as each insertion/deletion requires an update of the skip list to conserve balance, but the algorithms are still significantly simpler than those used in red-black trees.
4. Analysis of Deterministic Skip Lists

In [2] the authors analyze two different general ways of implementing Deterministic Skip Lists: The one uses an array for each key (array implementation), and the other uses a linked list (linked list implementation) to keep track of which key follows which at a given height. The authors also define an invariant used to maintain balance.

A 1-2 skip list is a skip list in which there exist either one or two nodes of height \( h - 1 \) between any two nodes of height \( h \) (these nodes are called the gap, see figure 1). A 1-2-3 skip list (or simply 1-3 skip list) extends this concept to gaps of max. length 3. We use this variant because working code for it is given in [3].

The gap size is defined as the number of nodes of height \( h - 1 \) between two nodes of height \( h \). Figure 1 illustrates the notions of height, gaps and gapsizes. We follow standard notation by starting count of height at 0, counting from the bottom and up (see figure 1).

4.1 Search

The search algorithm is fairly simple:

1. Starting at the header, at the highest layer of the list:
2. Look at the key in the node following the pointer at the current height from the current node.
3. If the key of the node is larger than the key we’re looking for, go down a level, else follow the pointer to the next node.
4. Repeat until the key is found or the bottom is reached.

A search is illustrated in figure 2.

4.2 Insertion

Insertion must maintain the invariant of the skip list. Using a 1–3 skip list, a way of inserting into the list while maintaining the invariant could be to split any gap of size 3 encountered during the search for the correct place to insert the new element. The splitting is done by raising the middle element, yielding two gaps of size 1, and, thus, maintaining the invariant [3].
1. Start search at the header, at a level higher than the height of the list.
2. When dropping into a gap do one of the following:
   – If the gap has size 1 or 2, drop into the gap.
   – If the gap has size 3, raise middle element in this gap and then drop
3. Continue doing this until the bottom is reached.
4. At the bottom insert the new element with height 1.

As an example, we insert 42 into the skip list in figure 1. Starting at height 2, we see that the gap at height 1 has size 2, and thus we can drop to height 1. At height 1 we follow the pointer to the right (key with value 30), and there we stop as the next node has key 51. The gap between these nodes has size 3, and thus we raise the node with key 41 one height. As 41 < 42, we follow the new pointer to 41, drop into the gap and insert 42 between 41 and 48. The updated skip list is shown in figure 3.

4.3 Deletion

Deleting an element from a deterministic skip list is somewhat more complicated than adding a new element. In [2] no implementation is given, and most "implementations" of deterministic skip list have not implemented the delete operation (the only exception is in [3]). The best description of a delete algorithm we could find was in [3].
1. Start search at header, at a level higher than the height of the list.
2. If gap has size 2 or 3 drop into gap
3. If gap has size 1, do the following:
4. If the gap is not the last gap in its height, and the following gap has size 1, we merge the gaps (by lowering the height of the separating element). If the following gap has size 2 or 3, we borrow an element from it by lowering the height of the separating element and increasing the height of the first element in the gap.
5. If the gap is the last gap in its height, and the preceding gap has size 1, we merge the gaps (by lowering the height of the separating element). If the preceding gap has size 2 or 3, we borrow an element from it by lowering the height of the separating element and increasing the height of the last element in the gap.
6. After this we drop, and redo the preceding until we reach the bottom.
7. When we find the element to delete, and it has height 1, we delete it
   (and update pointers). If its height is greater than 1, we swap it with
   its predecessor that has height 1 before deleting it\(^1\).

As an example, we delete 53 from the skip list in figure 3. Starting at
height 2, we note that the gap size is 3, and no changes are needed, and
we drop to height 1. As 30 < 53, 41 < 53, and 51 < 53 we go right to the
node with key 51. This gap has size 1, and must be expanded to secure the
invariant. As this is the last element in the height before the tail, we have
to look at the previous gap. It has size 2. Therefore we “borrow” an element
by raising 48 and decreasing 51. Then we drop to height 0. The node to
the right of 51 is 53, which has height 0 and can be deleted without further
ado. The updated skip list is shown in figure 4.

4.4 Array implementation

The array implementation uses an array in each node to store the point-
ers to the following nodes. Expansion of these arrays gives problems, as
the naive method (expand array-size by one every time) gives a worst case
\(\Theta(\log^2 n)\) cost. By choosing the arrays to have exponentially increasing phys-
ical heights, the worst case cost can be reduced to to \(O(\log n)\), as "the logical
heights of the nodes will never exceed their physical heights" \[2\]. This generates
an increase in memory usage, but it is shown that the memory usage
of the array never exceeds \(2.282n\) pointers on average.

The array implementation also has the advantage of better localization in
memory as the elements of the array lie in a contiguous chunk in memory.
Therefore, it’s highly unlikely that decreasing or increasing height will result
in a memory-miss.

4.5 Linked List implementation

Using linked lists in each node makes it very easy to increase or decrease
the height of a node. Just add or remove an element from the list. It

\(^1\) This is where iterator validity can be destroyed
does, however, require more space: The pointer to the right in the array-implementation must be substituted with a pointer to the right, a pointer to the next element in the linked list, and a pointer to the key. In the worst case the list implementation will require $6n$ pointers [3] – more than twice as much as the array implementation.

As these pointers can be allocated at different positions in memory, the probability of a memory-miss is greater than that of the array-implementation. This is aggravated by the fact that the linked list uses more memory. On the other hand, the spatial locality problem could be remedied using a custom allocator for list nodes.

This implementation has two advantages over the array implementation:
1. It is simpler to implement.
2. An implementation of 1-3 skip lists already exists [3].

The implementations presented in [2] and [3] use a modified version of the linked list implementation, noting that the use of linked lists makes central nodes containing the linked list unnecessary.

5. Construction of the solution

From the preceding discussion (section 4) it is clear that the array implementation is the most efficient: The time complexity is the same, but it uses significantly less memory. This is why we will implement the array solution using a 1-3 skip list. In the following sections we will discuss and present a sketch of the solution, aimed at a C++ implementation, including pseudocode for important algorithms. A graphical overview of our design can be seen in figure 5.

5.1 Representation of the skip list

At the core of the implementation of deterministic skip lists lies the Node. In the Node we will need an array of Node pointers, right, and a pointer to a key or value of some sort. This will make it easy to give strong iterator guarantees. We will also need two integers to keep track of the logical height (number of elements) and physical height (max. number of elements) of the array. The alternative – calculating them as needed – will significantly hurt performance, and for large lists, the increased memory usage will be negligible. To allow iterator-operations ++ and -- in constant time, we add a Node pointer, left to the previous element of the list. The time and memory overhead of this addition is negligible. We have chosen to implement the node size controller as a separate class, such that the height can be controlled using the “pretty” pseudocode-like statement node_size++ and equivalently for reducing the node size, node_size--. These operations automatically take care of resizing the underlying pointer array.

The SkipList class will need a Node pointer to the head and tail. The key of the head and tail were assumed to be max and max + 1 in [2]. For a generic SkipList this will not do. Instead we propose to let head have
key null and tail contain the maximum key inserted. Additionally, we will let the array of tail contain only null pointers. To ensure correctness of insert and erase these two nodes must be initialized with height 2, and must always be at least one level higher than the list. As we have constant time access to array elements, operations begin and end can easily be implemented to run in constant time.

5.2 Datahandlers

A map establishes a relation between two data values, called the key and the satellite, respectively. Sets only contain keys. We will need some way of handling this data, and we would also like to ensure iterator validity (that the iterator always points to the correct data, unless the data is deleted).

One option could be to let the nodes contain the key and the satellite (or a pointer to these). As the delete-algorithm can move nodes around, this will not ensure iterator validity.

An alternative is to use a class, DataHandler, to contain the data. The Node should have a pointer to the DataHandler, and the DataHandler should have a pointer to the Node.

This will ensure iterator validity, but leaves us with a decision to be made. Either the DataHandler should contain the data explicitly, or it should contain pointers to the data. Considerations in this matter include the size of objects and the complexity of their copy constructors, as well as delegation of new/delete responsibilities in the pointer case. The exchange trick used in deletion (section 4.3) makes the exact boundaries of responsibility unclear.

We have chosen the pointer method due to its generality. As a consequence, storage of small objects is expensive time- and memory-wise (in a set of ints, each node will need memory for 4 pointers on top of the key). On the other hand it is easier to create non-pointer specializations for small data types (POD items in particular). If given time, it would be prudent to make some specializations of the template to decrease memory usage.

5.3 Comparators

The C++ standard mandates only one comparison operator for the interface we need to use (namely the compare function which works as a less-than operator). This is very awkward, and since the skip list implementation depends on performing a variety of comparisons, we have been forced to implement a class that wraps this function into a more convenient set of operators. Not all compilers are able to compile this to efficient code, however. Profiling our code reveals that a lot of time is spent in the comparator, even when optimization is turned on with the -O2 option. This is another obvious opportunity for improvement which we have not pursued.

---

2 Section 25.3 clause 2
5.4 Iterators

The basic concept of iterators is a kind of element pointer that can be moved back and forth in the list. We want to be able to do this in constant time, and this is easy using skip lists. Since one layer of the skip list always has links to all elements, we can simply use that layer.

To recover the data element that an iterator points to, we use the \* (dereferentiation) operator. The exact return type of this operator depends on the container it is attached to. If the container is a set or multiset, the return value has type Key. If the container is a map or multimap, the return value is a pair consisting of Key and Satellite. Templates are ideal for this purpose, because they allow us to have just one code base for both cases. In the first iteration of the code, we used the \texttt{IfThenElse} template presented in [5] which allowed us to select a return type appropriately. However, we also needed this sort of control over the prototype of e.g. the \texttt{insert} method and its dependent routines. This is because we signal the lack of a satellite in the template arguments by setting the type of Satellite to \texttt{void}, but \texttt{void} is not really suited in general to indicate the absence of something.

In the end, we made a combined traits/policy class that embodies the logic involved in this, in conjunction with a base class and two specialized subclasses that expose exactly the prototypes we need. A motivating factor in this choice was the desire to duplicate as little algorithm-related code as possible. We can not entirely avoid code duplication as part of template specialization, but the \texttt{insert} and \texttt{erase} implementations, in particular,
ASSOCIATIVE CONTAINERS WITH STRONG GUARANTEES

5.5 Pseudocode for lower_bound

As we should implement functions find, lower_bound and upper_bound, and these functions obviously have a lot of code in common, it would be wise to implement one function to do the work, and let the others call this function. We note that lower_bound should return either the key or the first element with a larger key. This function can be called by find and if it returns key, find can return key. If called by upper_bound and key is returned, upper_bound can return the value to the right of key. Also note, that we define the invariant that height = −1 is the “bottom” discussed in section 4.1 – 4.3.

This pseudocode can be used to search a skip list for a key. It is at the core of the insert and erase operations, to be presented later. It is written in the style of [1].

5.6 Pseudocode for insert

In the pseudocode for insert it will be assumed that we have some functions whose pseudocode is not shown here. These are CalcGapSize which calculates the size of a gap to the right of a node x at a specified height; SplitGap which splits a gap of size 3 into two gaps of size 1 by raising the middle element; UpdatePointers that takes care of updating pointers on insertion; and NewNode that builds a new node. In general, the helper routines work on the gap to the right of the current position because the implementations in [3] and [2] are based on singly linked lists and therefore assume that we can’t move backwards in the list. Additionally we must have some function to increase the height of a Node. Also note that the gap size at height −1 is undefined, and the function should return 0 or a similar sentinel value.
INSERT(key)
1 \( x \leftarrow head \)
2 \( height = x.\log\text{height} - 1 \)
3 \( \text{if} \ tail.key = \text{null} \text{ or} \ tail.key < key \)
4 \( \text{then} \ tail.key = key \)
5 \( \text{while} \ (x = head \text{ or} \ x.key < key) \text{ and} \ x \neq tail \text{ and} \ height \neq -1 \triangleright \text{search for insert position} \)
6 \( \text{do while} \ x.\text{right}(height).key < key \text{ and} \ x.\text{right}(height) \neq tail \triangleright \text{go right} \)
7 \( \text{do} \ x \leftarrow x.\text{right}(height) \)
8 \( \text{if} \ x.\text{right}(height) = tail \text{ or} \ x.\text{right}(height).key \geq key \triangleright \text{go down into gap} \)
9 \( \text{then if} \ \text{CALCGAPSIZE}(x, height - 1) = 3 \)
10 \( \text{then} \ \text{SPLITGAP}(x, height - 1) \)
11 \( \text{else} \ height \leftarrow height - 1 \)
12 \( y \leftarrow \text{NEWNODE}(key) \triangleright y \text{ is the node to insert} \)
13 \( z \leftarrow x.\text{right}(0) \triangleright z \text{ is the node after the one to insert} \)
14 \( \text{UPDATEPOINTERS}(x, y, z) \)

Next, we illustrate with pseudocode how insertion is performed. As we will need to support insertion for sets, multisets, maps and multmaps, our actual implementation uses a general insert helper function that handles all cases and can be called by the other inserts. These wrappers are then exposed as needed.

5.7 Pseudocode for erase

This algorithm is somewhat more complicated than insert, and the helper functions are a bit more complex. The functions BorrowLeft and BorrowRight borrows a Node from a gap to the left or right, respectively. The functions MergeLeft and MergeRight merges a gap of size 1 with another gap of size 1 to the left or right of the first gap, respectively. erase start at a level lower than in insert and lower_bound as it can’t borrow or merge in either direction. Finally, we need a function, DeleteNode to delete the specified Node and update pointers. This function will also need to ensure that a head or a tail isn’t deleted.

6. Tests

The purpose of this section is to document which parts of the implemented skip list that work, and which parts that don’t. As we haven’t implemented the full interface, we only test the parts that have been implemented. This is done by unit testing. The classes SkipList, iterator and UsableComparator each has a function, UnitTest, that performs the unit test. The function returns a bool, true if the unit test was successful and false otherwise. In the latter case, a message specifying the error is printed to the screen. Unit testing has been a useful tool in finding and re-
Erasure(key)

1. \( x \leftarrow \text{head} \)
2. \( \text{height} = x.\log\text{height} - 2 \)
3. \( \text{left} \leftarrow \text{null} \)
4. \( \text{while} \ (x = \text{head} \lor x.\text{key} < \text{key}) \land x \neq \text{tail} \land \text{height} \neq -1 \Rightarrow \text{search for delete node} \)
5. \( \text{do while} \ x.\text{right}(\text{height}) \neq \text{tail} \land (x.\text{right}(\text{height}).\text{key} < \text{key}) \Rightarrow \text{go right} \)
6. \( \text{do left} \leftarrow x \)
7. \( x \leftarrow x.\text{right}(\text{height}) \)
8. \( \text{if} \ x.\text{right}(\text{height}) = \text{tail} \lor (x.\text{right}(\text{height}).\text{key} \geq \text{key}) \Rightarrow \text{go down into gap} \)
9. \( \text{then if} \ \text{CalcGapSize}(x, \text{height} - 1) = 1 \Rightarrow \text{we must fix gap} \)
10. \( \text{then if} \ x.\text{right}(\text{height}).\log\text{height}! = \text{height} + 1 \Rightarrow x \text{ last of height} \)
11. \( \text{then if} \ \text{CalcGapSize}(\text{left}, \text{height} - 1) \neq 1 \)
12. \( \text{then BorrowLeft(\text{left}, x, \text{height} - 1)} \)
13. \( \) else
14. \( \text{MergeLeft(left, x, height - 1)} \)
15. \( \) else \( x \) not last of height
16. \( \text{if} \ \text{CalcGapSize}(x.\text{right}(\text{height}), \text{height} - 1) \neq 1 \)
17. \( \text{then BorrowRight(x.right(height), height - 1)} \)
18. \( \) else
19. \( \text{MergeRight(x, x.right(\text{height}), \text{height} - 1)} \)
20. \( \text{height} \leftarrow \text{height} - 1 \)
21. \( \text{if} \ x.\text{key} = \text{key} \)
22. \( \text{then DeleteNode}(x) \Rightarrow x \text{ contains the value to be deleted} \)

The unit test functions are implemented in unittest.cpp.

The unit test of UsableComparator defines two comparators and tests if the UsableComparators functions return correctly.

In the unit test of iterator a skip list is constructed and two iterators are declared. These two iterators are used to test all of the iterator methods.

The unit test of SkipList is somewhat more complex. This is because of the large numbers of private methods. All the private methods are tested with legal inputs – it is the responsibility of other functions to make sure these functions are used correctly. It has been attempted to test for all cases, as described in UnitTest – many are also mentioned in section 4 and 5. As the functions are private, the test method must be a class method and it must operate on the calling skip list. The testing of the private functions also allows for a very thorough investigation of the results, and this has made this part of the unittest fairly long and cumbersome (but it gives a good indication of correctness).

The public functions are tested by using them on different types of skip lists. This makes the tests less thorough, but a bit simpler to read. The public functions are functions that control the logic, so the tests have focused on both making sure that all cases are called (the logic), as well as correctness.
Morten Poulsen and Lars Skovlund

Figure 6. Time pr. operation for a skip list with unique keys (y-axis), given the number of operations (x-axis)

6.1 Benchmarking

The central operations of the deterministic skip list should run in logarithmic time. An interesting question is whether or not this implementation of skip lists is able to do this. A simple program has been written to execute a lot of inserts, erases and finds on skip lists with unique and nonunique keys. The benchmark program was run on a computer with a Pentium 1.86 GHz processor, 2.0 GB ram, using Windows XP. The program were compiled with Visual Studio 2005 using the setting release. The results are summed up in figures 6 and 7.

From the figures it can be seen that operations run faster on unique-key skip lists, which is not surprising, as they (on average) will include more keys. Second, it is seen that insert and find seems to run in logarithmic time.

Erase, however, acts a bit strangely, as the run time per operation seem to decrease as the length of the skip list increases. One reason for this may the resizing of arrays: During insertion an array is resized by doubling the size. During erasing an array is resized when the size reaches one quarter of the functions.

All the unit tests return true, and thus it is likely that the implementation is correct. Note, however, that a unit test is no proof of correctness: There may still be bugs in the implementation. Also note that the Node and DataHandler classes haven’t been unit tested separately, but as they are used heavily in SkipList, it’s likely that they are correct as well.
Figure 7. Time pr. operation for a skip list with duplicate keys (y-axis), given the number of operations (x-axis)

<table>
<thead>
<tr>
<th>Function</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructor</td>
<td>done</td>
</tr>
<tr>
<td>++</td>
<td>Prefix done, postfix missing</td>
</tr>
<tr>
<td>-=</td>
<td>Prefix done, postfix missing</td>
</tr>
<tr>
<td>=</td>
<td>done</td>
</tr>
<tr>
<td>==</td>
<td>done</td>
</tr>
<tr>
<td>!=</td>
<td>done</td>
</tr>
</tbody>
</table>

Figure 8. Implementation of iterator

the allocated size, and it is less expensive as fewer elements are copied. So, as the length of the skip list is increased, the amount of time erase uses on average to resize arrays is reduced.

7. Conclusion

We started this project by considering each data structure in detail. Once the final data structure was chosen, we spent a lot of time planning the code. As a consequence we have not achieved working versions of all the constructs required in the C++ standard (see figures 8 and 9). On the other hand, we have spent a lot of time verifying the code we do have with unit tests.

As can be seen, a functional skip list has been implemented. It is possible to do inserts, erases and searches, and it supports both sets, maps, multisets
and multimaps. Most notably missing are the iterator based insert and erase operations included in the standard. These were not covered in [2] or [3], and would require development of two new algorithms. The basic skip list operations are done and were coded from scratch. Various advanced OOP techniques were used in the process. The code was plagued with memory leaks in earlier versions, but we believe we have managed to find them all in the version presented here.

<table>
<thead>
<tr>
<th>Function</th>
<th>Status</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructor</td>
<td>done</td>
<td>construction from sorted list</td>
</tr>
<tr>
<td>requires insert with iterators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>begin()</td>
<td>done</td>
<td></td>
</tr>
<tr>
<td>end()</td>
<td>done</td>
<td></td>
</tr>
<tr>
<td>insert(Key)</td>
<td>done</td>
<td></td>
</tr>
<tr>
<td>insert(Key, Value)</td>
<td>done</td>
<td></td>
</tr>
<tr>
<td>insert(iterator, Key)</td>
<td>not done</td>
<td></td>
</tr>
<tr>
<td>insert(iterator, Key, Value)</td>
<td>not done</td>
<td></td>
</tr>
<tr>
<td>insert(iterator, iterator)</td>
<td>not done</td>
<td>requires insert with iterators</td>
</tr>
<tr>
<td>erase(Key)</td>
<td>done</td>
<td>can be improved by erase(iterator)</td>
</tr>
<tr>
<td>erase(iterator)</td>
<td>not done</td>
<td></td>
</tr>
<tr>
<td>erase(iterator, iterator)</td>
<td>not done</td>
<td>requires erase(iterator)</td>
</tr>
<tr>
<td>clear()</td>
<td>not done</td>
<td>requires erase(iterator, iterator)</td>
</tr>
<tr>
<td>find(Key)</td>
<td>done</td>
<td></td>
</tr>
<tr>
<td>lower_bound(Key)</td>
<td>done</td>
<td></td>
</tr>
<tr>
<td>upper_bound(Key)</td>
<td>done</td>
<td></td>
</tr>
<tr>
<td>count(Key)</td>
<td>done</td>
<td></td>
</tr>
<tr>
<td>equal_range(Key)</td>
<td>done</td>
<td></td>
</tr>
<tr>
<td>size()</td>
<td>done</td>
<td></td>
</tr>
<tr>
<td>max_size()</td>
<td>not done</td>
<td>should be implemented in container</td>
</tr>
<tr>
<td>empty()</td>
<td>done</td>
<td></td>
</tr>
<tr>
<td>boolean operators</td>
<td>not done</td>
<td>should be implemented in container</td>
</tr>
</tbody>
</table>

Figure 9. Implementation of skip list
Appendix A. Source

The source code of our implementation of deterministic skip lists follows below. Note that the unit test is included in the test section.

Appendix A.1 skiplist.h

```cpp
/**
 * SKIPLIST.H
 * by Lars Skovlund and Morten Poulsen
 * this file contains headers and in some instances implementation
 * for the following classes:
 * - UsableComparator
 * - IsVoid_T
 * - SkipList (and it's iterator)
 * - Node
 * - NodeSize
 * - DataHandler
 *
 * **************************************************************/

#include <stdlib.h>
#include <utility>
#include <iostream>

#include "ifthenelse.hpp"

/* Forward declare some classes */
template<typename Key , typename Satellite , typename Comparator>
class Node;

template<typename Key , typename Satellite , typename Comparator , typename unique>
class SkipList;

template<typename Key , typename Satellite , typename Comparator , typename unique>
class BaseSkipList;

template<typename Key , typename Satellite , typename Comparator ,
         typename unique>
class DataHandler;

#include "traits.hpp"

/* the comparison function compare in the c++ standard is very heavy to use,
 * therefore an alternative is defined here
 * the standard defines (25.3) compare(a,b) as a>b */
template<typename Key , typename Comparator>
class UsableComparator
{
  public:
    // constructors
    UsableComparator(): comp(){};
    ~UsableComparator() {};
    // methods
    // Note: These is not optimized
    bool Equals (Key const* a, Key const* b)
    { return !(comp.Compare(*a, *b) || comp.Compare(*b, *a)); }
```

class Comparator comp;

// testing
bool UnitTest();

// unit test function to test less than, less than equals, greater than, greater than equals
bool HelpUnitTest(Key *a, Key *b, Key *c);

// field
template<typename Key, typename Satellite, typename Comparator, typename unique>
class BaseSkipList
{
    public:
    class iterator
    {
        public:
    } // fields

    // testing
    bool UnitTest();

    // unit test function to test less than, less than equals, greater than, greater than equals
    bool HelpUnitTest(Key* a, Key* b, Key* c);
};

template<typename Key, typename Satellite, typename Comparator, typename unique>

    class iterator
    {
        public:
    } // fields

    // testing
    bool UnitTest();

    // unit test function to test less than, less than equals, greater than, greater than equals
    bool HelpUnitTest(Key* a, Key* b, Key* c);
};
DataHandler< Key, Satellite, Comparator > *position;
/* Internal constructor for iterator for use by parent class */
iterator(DataHandler< Key, Satellite, Comparator > *d) : position(d) {
};
/* Default constructor */
BaseSkipList() : comp(),
    head(unique::CreateSentinel()),
    tail(unique::CreateSentinel())
{
    head->right[1] = tail;
    head->right[0] = tail;
    tail->left = head;
}
BaseSkipList(BaseSkipList const &a) {} /* Copy constructor */
~BaseSkipList();
iterator begin() { return iterator(this->head->right[0]->data); }
iterator end() { return iterator(this->tail->data); }
void insert(iterator p, Key t); // not implemented
int erase(Key e);
void erase(iterator p); // not implemented
void erase(iterator p, iterator q); // not implemented
iterator lower_bound(Key e);
iterator upper_bound(Key e);
int count(Key e);
int size();
bool empty() { return (size() == 0); }
std::pair<iterator,iterator> equal_range(Key k)
{
    return std::make_pair(lower_bound(k),upper_bound(k));
}
BaseSkipList &operator=(BaseSkipList const &a) {};
Node< Key, Satellite, Comparator >*head, *tail;

/* Test function*/
bool UnitTest();
/* help in unittest*/
bool HelpUnitTest(
    Node<Key,Satellite,Comparator>* node,
    Key expectedKey, int expectedHeight, Key expectedLeft,
    Key expectedRight0, Key expectedRight1, Key expectedRight2);
/* prints node to screen*/
void print();

private:
/* Helper functions*/
/* Insert is the function to do the actual insertion. */
std::pair<iterator, bool> Insert(typename unique::key_value_type k);
/* helper function to calculate size of gap to the right of node */
int CalcGapSize(Node<Key, Satellite, Comparator> const* node, int height);
/* decreases the height of node */
void DecreaseHeight(Node<Key, Satellite, Comparator>* node);
/* deletes node and updates pointers around it*/
void DeleteNode( Node< Key, Satellite, Comparator > * node );
Morten Poulsen and Lars Skovlund

```c++
/* increases the height of node */
void IncreaseHeight(Node<Key, Satellite, Comparator>* node);
/* Splits the gap to the right of node with height */
void SplitGap(Node<Key, Satellite, Comparator> const* node,
             unsigned int height);
/* updates pointers after insertion*/
void UpdatePointers(Node<Key, Satellite, Comparator>* left,
                    Node<Key, Satellite, Comparator>* newcenter,
                    Node<Key, Satellite, Comparator>* right);
/* borrows a node from the gap to the left of separator (starting at gapStarter)
   * Basically it looks like this (where g is gapStarter and s is separator):
   *           # # # # # # # height+1
   *      # # # # # # # height
   */
void BorrowLeft(Node<Key, Satellite, Comparator>* gapStarter,
                Node<Key, Satellite, Comparator>* separator,
                int height);
/* borrows a node from the gap to the right of separator
   * (the right gap starts at gapStarter)
   * Basically it looks like this (where g is gapStarter and s is separator):
   *           # # # # # height+1
   *      # # # # # height
   */
void BorrowRight(Node<Key, Satellite, Comparator>* gapStarter,
                 Node<Key, Satellite, Comparator>* separator,
                 int height);
/* merges two gaps of size 1, the left gap starting at gapStarter
   * Basically it looks like this (where g is gapStarter and s is separator):
   *           # # # # height+1
   *      # # # # height
   */
void MergeLeft(Node<Key, Satellite, Comparator>* gapStarter,
               Node<Key, Satellite, Comparator>* separator,
               int height);
/* merges two gaps of size 1, the right gap starting at gapStarter
   * Basically it looks like this (where g is gapStarter and s is separator):
   *           # # height+1
   *      # # height
   */
void MergeRight(Node<Key, Satellite, Comparator>* gapStarter,
                Node<Key, Satellite, Comparator>* separator,
                int height);
/* updates the height of head and tail on delete and decreaseheight*/
void UpdateHeadAndTail();
/* Fields*/
/* usablecomparator defines usable comparison operators */
UsableComparator<Key, Comparator> comp;
friend class SkipList<Key, void, Comparator, unique>;
friend class SkipList<Key, Satellite, Comparator, unique>;
};

template<typename Key, typename Satellite, typename Comparator, typename unique>
class SkipList : public BaseSkipList<Key, Satellite, Comparator, unique>
{    
};
```
public:
    std::pair<typename BaseSkipList<Key, Satellite, Comparator, unique>::iterator, bool>
    insert(Key k, Satellite sat)
    { return Insert(std::make_pair(k, sat)); }

private:

template<typename Key, typename Comparator, typename unique>
class SkipList<Key, void, Comparator, unique>
    : public BaseSkipList<Key, void, Comparator, unique>
{

    template<typename Key, typename Satellite, typename Comparator>
    class Node
    {
    
    
    
    public:

    NodeSize(N ***array) : physical(1), logical(1)
    {
        right = static_cast<N *> (malloc(sizeof(N *) * physical));
        memset(*right, 0, sizeof(N *) * physical);
    }

    NodeSize(N ***array, int initial_height) :
        physical(initial_height),
        logical(initial_height)
    {
        right = array;
        *right = static_cast<N *> (malloc(sizeof(N *) * physical));
        memset(*right, 0, sizeof(N *) * physical);
    }

    NodeSize &operator ++()
    {
        logical++;
        if (logical > physical)
            *right = static_cast<N *> (realloc(*right, sizeof(N *) * physical * 2));
        memset(&(*right)[physical], 0, sizeof(N *) * physical);
        physical += 2;
    }

    return *this;

    NodeSize &operator --()
    {
        logical --;
    }
if (logical <= physical/4) {
    physical /= 4;
    *right = static_cast<N **>(realloc(*right, sizeof(N *) * physical));
} return *this;

N *operator[](int n) const {
    return right[n];
}

unsigned int logical_height() const {
    return logical;
}

unsigned int physical_height() const {
    return physical;
}

/*! Q: Why can’t we just write N here?
   A: Because it results in a compiler error, that’s why! :-)/

friend class Node<Key, Satellite, Comparator>;
};

public:
    Node() : right_size(&right), left(NULL) {} /* Default constructor */
    Node(Key *key, Satellite *sat = NULL, unsigned int initial_height = 1) :
        right_size(NodeSize(&right, initial_height)),
        data(new DataHandler<Key, Satellite, Comparator>(this, key, sat)),
        left(NULL) {
    }
    Node(Node const &a) {} /* Copy constructor */
    ~Node() {}
    const Key *get_key() const { return data->key; }
    void set_key(Key k) {
        delete data->key;
        data->key = new Key(k);
    }
    void set_data(DataHandler<Key, Satellite, Comparator> *newData) {
        data = newData;
    }
    void print_node(Node<Key, Satellite, Comparator> *tail);
    DataHandler<Key, Satellite, Comparator> *data;
    Node<Key, Satellite, Comparator> *left;
    NodeSize right_size;
    Node<Key, Satellite, Comparator> **right;

friend class SkipList<Key, Satellite, Comparator, UniqueTraits<Key, Satellite, Comparator>> :: iterator;
friend class SkipList<Key, Satellite, Comparator, NonUniqueTraits<Key, Satellite, Comparator>>::iterator;

template<typename Key, typename Comparator>
class Node<Key, void, Comparator> {
    class NodeSize {
        typedef Node<Key, void, Comparator> N;

        private:
        unsigned int physical;
        unsigned int logical;
        N ***right;

        public:
        NodeSize(N ***array) : physical(1), logical(1) {
            right = array;
            *right = static_cast<N **>(malloc(sizeof(N *) * physical));
            memset(*right, 0, sizeof(N *) * physical);
        }

        NodeSize(N ***array, int initial_height) :
            physical(initial_height),
            logical(initial_height) {
            right = array;
            *right = static_cast<N **>(malloc(sizeof(N *) * physical));
            memset(*right, 0, sizeof(N *) * physical);
        }

        NodeSize &operator++() {
            logical++;
            if (logical > physical) {
                *right = static_cast<N **>(realloc(*right, sizeof(N *) * physical * 2));
                memset(&(*right)[physical], 0, sizeof(N *) * physical);
                physical *= 2;
            }
            return *this;
        }

        NodeSize() { free(*right); }

        NodeSize &operator--() {
            logical--;
            if (logical <= physical / 4) {
                physical /= 4;
                *right = static_cast<N **>(realloc(*right, sizeof(N *) * physical));
            }
            return *this;
        }

    } &operator[] (int n) const {
};

N &operator[] (int n) const {
```cpp
unsigned int logical_height()
{
    return logical;
}

unsigned int physical_height()
{
    return physical;
}

/* Q: Why can't we just write N here? */
A: Because it results in a compiler error, that's why! */
friend class Node<Key, void, Comparator>;

public:
    Node() : right_size(&right), left(NULL) { /* Default constructor */
    right_size(NodeSize(&right, initial_height));
    data(new DataHandler<Key, void, Comparator>(this, key)),
    left(NULL)
    { };

    Node(Node const &a) { } /* Copy constructor */
    ~Node() { delete data; }
    Key *get_key() { return data->key; }
    void set_key(Key k) {
        delete data->key;
        data->key = new Key(k);
    }
    void set_data(DataHandler<Key, void, Comparator> *newData) { data = newData; }
    void print_node(Node<Key, void, Comparator> *tail);

DataHandler<Key, void, Comparator> *data;
Node<Key, void, Comparator> *left;
NodeSize right_size;
Node<Key, void, Comparator> **right;
friend class SkipList<Key, void, Comparator, UniqueTraits<Key, void, Comparator>> ::iterator;
friend class SkipList<Key, void, Comparator, NonUniqueTraits<Key, void, Comparator>> ::iterator;
};

template<typename Key, typename Satellite, typename Comparator>
class DataHandler
{
public:
    DataHandler() { } /* Default constructor */
    DataHandler(DataHandler const &a) { } /* Copy constructor */
    DataHandler(Node<Key, Satellite, Comparator> *owner, Key *k, Satellite *s) :
        key(k), sat(s), node(owner) {};
    //DataHandler(Node<Key, Satellite, Comparator> *owner, Key *k)
    // : key(k), sat(NULL), node(owner) {};
    ~DataHandler() { delete key; delete sat; }
};
```
const Key &get_key() { return *key; }
std::pair<Key, Satellite> get_keyval() { return std::make_pair(*key, *sat); }
void set_node(Node<Key, Satellite, Comparator> *newNode) { node = newNode; }

private:
Node<Key, Satellite, Comparator> *node;
Key *key;
Satellite *sat;

friend class SkipList<Key, Satellite, Comparator, UniqueTraits<Key, Satellite, Comparator>, NonUniqueTraits<Key, Satellite, Comparator>> :: iterator;
friend class SkipList<Key, Satellite, Comparator, NonUniqueTraits<Key, Satellite, Comparator>> :: iterator;
friend class Node<Key, Satellite, Comparator>;

};

};

}

#include "DeterministicSkipList.cpp"
#include "unittest.cpp"

 AppendixA.2 deterministicskiplist.cpp

/*****************************/
2  * DETERMINISTICSKIPLIST.CPP
3  *
4  * by Lars Skovlund and Morten Poulsen
5  *
6  * this file implements the methods and functions of the
7  * DeterministicSkipList class.
8  *
9  **************************************************************/

using namespace std;
template<typename Key, typename Satellite, typename Comparator, typename unique>
BaseSkipList<Key, Satellite, Comparator, unique>::BorrowLeft(Node<Key, Satellite, Comparator> * gapStarter, Node<Key, Satellite, Comparator> * seperator, int height)
{
    Node<Key, Satellite, Comparator> * borrow = gapStarter;
    // find the rightmost node in left gap
    while( (borrow->right[height])->right_size.logical_height() - 1 == height )
    {
        borrow = borrow->right[height];
    }
    if (borrow != gapStarter)
    {
        IncreaseHeight(borrow);
        borrow->right[height+1] = seperator->right[height+1];
        gapStarter->right[height+1] = borrow;
        DecreaseHeight(seperator);
    }
    else
    {
        // std::cout << "Error in BorrowLeft!" << std::endl;
    }
}

template<typename Key, typename Satellite, typename Comparator, typename unique>
BaseSkipList<Key, Satellite, Comparator, unique>::~BaseSkipList()
{
    Node<Key, Satellite, Comparator> * deleter = tail->left;
    Node<Key, Satellite, Comparator> * save;

    while (deleter != head)
    {
        save = deleter->left;
        delete deleter;
        deleter = save;
    }

    delete head;
    delete tail;
}

template<typename Key, typename Satellite, typename Comparator, typename unique>
void BaseSkipList<Key, Satellite, Comparator, unique>::BorrowRight(Node<Key, Satellite, Comparator> * gapStarter, Node<Key, Satellite, Comparator> * seperator, int height)
{
Associative containers with strong guarantees

// the leftmost node in gap
Node<Key, Satellite, Comparator>* borrow = gapStarter->right[height];
if (borrow != gapStarter)
{
    IncreaseHeight(borrow);
    separator->right[height+1] = gapStarter->right[height+1];

    DecreaseHeight(gapStarter);
}
else
{
    // std::cout << "Error in BorrowRight!" << std::endl;
}

/* helper function to calculate size of gap to the right of node at height*/
/* it should return 0 at height -1; and 1, 2 or 3 at heights larger than -1*/
template<typename Key, typename Satellite, typename Comparator, typename unique>
int BaseSkipList<Key, Satellite, Comparator, unique>::CalcGapSize(
    Node<Key, Satellite, Comparator>* node, int height)
{
    if (height != -1)
    {
        int i = 0;
        while ( (node->right[height])->right_size.logical_height() - 1 == height )
        {
            i++;
            node = node->right[height];
        }
        return i;
    }
    return 0;
}

/* decreases the height of node,
* testing if height of head and tail should be decreased as well */
template<typename Key, typename Satellite, typename Comparator, typename unique>
void BaseSkipList<Key, Satellite, Comparator, unique>::DecreaseHeight(
    Node<Key, Satellite, Comparator>* node)
{
    if (node->right_size.logical_height() > 1)
    {
        node->right[node->right_size.logical_height()-1] = NULL;
        --node->right_size;
        // update height of head and tail, if needed
        UpdateHeadAndTail();
    }
    else
    {
        // std::cout << "Tried to decrease height of node below 1!" << std::endl;
    }
}
template <typename Key, typename Satellite, typename Comparator, typename unique>
void BaseSkipList<Key, Satellite, Comparator, unique>::DeleteNode(Node<Key, Satellite, Comparator>* node)
{
    if (node != head && node != tail)
    {
        Node<Key, Satellite, Comparator>* previous = node->left;
        Node<Key, Satellite, Comparator>* next = node->right[0];
        // if height of node is 1, we can just delete it
        if (node->right_size.logical_height() == 1)
        {
            previous->right[0] = next;
            next->left = previous;
            delete node;
        }
        else
        {
            // height is larger than 1, so we must swap and then delete
            // we delete the old data in node, inserts data from previous
            // and delete previous
            delete node->data;
            node->set_data(previous->data);
            node->data->set_node(node);
            // int* prevKey_ptr = previous->get_key();
            // int prevKey = *prevKey_ptr;
            // node->set_key(prevKey);
            node->left = previous->left;
            previous->left->right[0] = node;
            previous->data = NULL; // we do not want to delete it's datahandler
            delete previous;
        }
    }

    // updating head and tail
    UpdateHeadAndTail();
}

// increases the height of node,
// testing if height of head and tail should be increased as well */
template <typename Key, typename Satellite, typename Comparator, typename unique>
void BaseSkipList<Key, Satellite, Comparator, unique>::IncreaseHeight(
    Node<Key, Satellite, Comparator>* node)
{
    if (node->right_size.logical_height() == head->right_size.logical_height() - 1)
    {
        ++head->right_size;
        ++tail->right_size;
        head->right[ head->right_size.logical_height() - 1] = tail;
    }
    ++node->right_size;
}

// this function inserts the key k into the skiplist. */
// note that it assumes that the skiplist is a correct 1–3 skiplist */
template<typename Key, typename Satellite, typename Comparator, typename unique>

typename BaseSkipList<Key, Satellite, Comparator, unique>::iterator,
bool>

BaseSkipList<Key, Satellite, Comparator, unique>::

Insert(typename unique::key_value_type k)

{ 
    Node<Key, Satellite, Comparator>* x = this->head;
    int height = x->right_size.logical_height() - 1;
    Key const* key = new Key(unique::get_key(k));

    if (tail->get_key() == NULL || comp.LessThan(tail->get_key(), key))
        { tail->set_key(unique::get_key(k)); }
    while( (x == head || comp.LessThan(x->get_key(), key))
        && x != tail && height != -1)
    {
        // go right
        while( comp.LessThan(x->right[height]->get_key(), key)
            && x->right[height] != tail)
            { x = x->right[height]; }
        // go into gap
        if( x->right[height] == tail
            || comp.GreaterThanEquals(x->right[height]->get_key(), key))
            { if( CalcGapSize(x, height -1) == 3 )
                { SplitGap(x, height -1); }
                else
                    { --height; }
            }
    }
    // done searching, ready for insertion
    Node<Key, Satellite, Comparator>* y = unique::createNode(k);
    Node<Key, Satellite, Comparator>* z = x->right[0];
    bool inserted = false;
    DataHandler<Key, Satellite, Comparator>* satPtr = x->data;
    if( unique::isUnique )
        { if( (x == head || *(x->get_key()) != unique::get_key(k))
                && (z == tail || *(z->get_key()) != unique::get_key(k)) )
            { UpdatePointers(x,y,z);
                satPtr = y->data;
                inserted = true;
            }
            else if( *(z->get_key()) == unique::get_key(k) )
            { satPtr = z->data;
            } }
else {
    UpdatePointers(x, y, z);
    satPtr = y > data;
    inserted = true;
}

delete key;
return std::make_pair<typename BaseSkipList<Key, Satellite, Comparator, unique>::iterator(satPtr), inserted);

/\* merges two gaps of size 1, the left gap starting at gapStarter */
template<typename Key, typename Satellite, typename Comparator, typename unique>
void BaseSkipList<Key, Satellite, Comparator, unique>::MergeLeft(
    Node<Key, Satellite, Comparator>** gapStarter,
    Node<Key, Satellite, Comparator>** seperator, int height)
{
    gapStarter->right[height +1] = seperator->right[height +1];
    DecreaseHeight(seperator);
}

/\* merges two gaps of size 1, the right gap starting at gapStarter */
template<typename Key, typename Satellite, typename Comparator, typename unique>
void BaseSkipList<Key, Satellite, Comparator, unique>::MergeRight(
    Node<Key, Satellite, Comparator>** gapStarter,
    Node<Key, Satellite, Comparator>** seperator, int height)
{
    seperator->right[height +1] = gapStarter->right[height +1];
    DecreaseHeight(gapStarter);
}

/\* Splits the gap to the right of node with height */
/\* assumes a gap size of 3 */
template<typename Key, typename Satellite, typename Comparator, typename unique>
void BaseSkipList<Key, Satellite, Comparator, unique>::SplitGap(
    Node<Key, Satellite, Comparator> const* node, unsigned int height)
{
    Node<Key, Satellite, Comparator>** nodeToIncrease =
        node->right[height]->right[height];
    IncreaseHeight(nodeToIncrease);
    nodeToIncrease->right[height +1] = node->right[height +1];
    node->right[height +1] = nodeToIncrease;
}

/\* update head and tail so that height is right after deletes and decreaseheights */
template<typename Key, typename Satellite, typename Comparator, typename unique>
void BaseSkipList<Key, Satellite, Comparator, unique>::UpdateHeadAndTail()
{
    int head_height = head->right_size.logical_height() -1;
if ( head->right[head_height] == tail
    && head->right[head_height - 1] == tail
    && head_height > 1 )
{
    // Updates pointers after insertion
    if ( head->right_size ;
        tail->right_size ;
    }
}

// Updates pointers after insertion

template<typename Key, typename Satellite, typename Comparator, typename unique>
void BaseSkipList<Key, Satellite, Comparator, unique>::UpdatePointers(Node<Key, Satellite, Comparator>* leftnode,
    Node<Key, Satellite, Comparator>* newcenter,
    Node<Key, Satellite, Comparator>* rightnode)
{
    leftnode->right[0] = newcenter;
    newcenter->right[0] = rightnode;
    newcenter->left = leftnode;
    rightnode->left = newcenter;
}

/*********************************************************
** Returns the number of elements containing key **
***********************************************************/
template<typename Key, typename Satellite, typename Comparator, typename unique>
int BaseSkipList<Key, Satellite, Comparator, unique>::count(Key k)
{
    int count = 0;
    typename BaseSkipList<Key, Satellite, Comparator, unique>::iterator iter = find(k);
    while( iter.position != tail->data
        && comp.Equals(&(iter.position->get_key()), &k) )
    {
        ++iter;
        count++;
    }
    return count;
}

/**********************************************************
** Returns the number of elements in skip list **
***********************************************************/
template<typename Key, typename Satellite, typename Comparator, typename unique>
int BaseSkipList<Key, Satellite, Comparator, unique>::size()
{
    int count = 0;
    typename BaseSkipList<Key, Satellite, Comparator, unique>::iterator iter = begin();
    while( iter != end() )
    {
        ++iter;
        count++;
    }
    return count;
}
Morten Poulsen and Lars Skovlund

```cpp
++iter;
    count++;
}
return count;

/* this function deletes the key k from the skip list. */
/* note that it assumes that the skip list is a correct 1-3 skip list */

template<typename Key, typename Satellite, typename Comparator, typename unique>
int BaseSkipList<Key, Satellite, Comparator, unique>::erase(Key k)
{
    Node<Key, Satellite, Comparator>* x = this->head;
    // we'll never have to do anything in the first iteration
    int height = x->right_size.logical_height() - 2;
    Key const* key = new Key(k);
    Node<Key, Satellite, Comparator>* left = NULL;

    // search for node to delete
    while((x == head || comp.LessThan(x->get_key(), key))
          && x != tail && height != -1)
    {
        // go right
        while(comp.LessThan(x->right[height]->get_key(), key)
            && x->right[height] != tail)
        {
            left = x;
            x = x->right[height];
        }
        // go into gap
        if(x->right[height] == tail
            || comp.GreaterThanEquals(x->right[height]->get_key(), key))
        {
            if(CalcGapSize(x, height-1) == 1) // need to fix gap
                {
                    if (height+1 == x->right[height]->right_size.logical_height())
                        && x != head) // x last of height
                        {
                            if(CalcGapSize(left, height-1) != 1)
                                {
                                    BorrowLeft(left, x, height-1);
                                }
                        }
                    else
                        {
                            MergeLeft(left, x, height-1);
                        }
                }
            else // x not last of height
                {
                    if (CalcGapSize(x->right[height], height-1) != 1)
                        {
                            BorrowRight(x->right[height], x, height-1);
                        }
                }
        }
    }
}
```
Associative containers with strong guarantees

```cpp
404     } } --height;
405     } } // done searching, ready for deleting
406     int noOfErasedKeys = 0;
407     Node<Key, Satellite, Comparator>* y = NULL;
408     if ( x != head && comp.Equals(x->get_key(), key) )
409     { y = x->right[0];
410     DeleteNode(x);
411     noOfErasedKeys++;
412     } else if ( comp.Equals(x->right[0]->get_key(), key) )
413     { y = x->right[0]->right[0];
414     DeleteNode(x->right[0]);
415     noOfErasedKeys++;
416     } // we must delete all keys equal to k and return number of keys deleted
417     // this is a dirty and inefficient way of doing it.
418     if ( y != NULL && comp.Equals(y->get_key(), key) && y != tail )
419     { return noOfErasedKeys + erase(k); }
420     delete key;
421     return noOfErasedKeys;
422     } } /* finds and returns an iterator to lower bound, that is the first element
423     * that is either key or greater than key */
424     /* note that it assumes that the skiplist is a correct 1–3 skiplist */
425     template<typename Key, typename Satellite, typename Comparator, typename unique>
426     typename BaseSkipList<Key, Satellite, Comparator, unique>::iterator
427     BaseSkipList<Key, Satellite, Comparator, unique>::lower_bound(Key k)
428     { Node<Key, Satellite, Comparator>* x = this->head;
429     int height = x->right.size.logical_height() - 1;
430     Key const* key = &k;
431     while ( x == head || comp.LessThan( x->get_key(), key) && x != tail
432     && height != -1)
433     { // go right
434     while ( comp.LessThan(x->right[height]->get_key(), key)
435     && x->right[height] != tail )
436     { x = x->right[height];
437     } // go into gap
438     if ( x->right[height] == tail
439     | comp.GreaterThanEquals(x->right[height]->get_key(), key) )
440     { --height;
441     } */
Morten Poulsen and Lars Skovlund

```cpp
460 } } iterator i(x->right[0]->data); return i; } */
465 /*Returns an iterator to the element if it exists
466 * if key doesn’t exist, return end*/
467 template<typename Key, typename Satellite, typename Comparator, typename unique>
468 typename BaseSkipList<Key, Satellite, Comparator, unique>::iterator
469 BaseSkipList<Key, Satellite, Comparator, unique>::find(Key k) {
470   typename BaseSkipList<Key, Satellite, Comparator, unique>::iterator iter = lower_bound(k);
471   if( iter.position != tail->data && comp.NotEquals(&(iter.position->get_key()), &k) )
472     iter = end();
473   return iter;
474 }
477 /*Returns an iterator to first element that is larger than the key (but not the key)*/
478 template<typename Key, typename Satellite, typename Comparator, typename unique>
479 typename BaseSkipList<Key, Satellite, Comparator, unique>::iterator
480 BaseSkipList<Key, Satellite, Comparator, unique>::upper_bound(Key k) {
481   typename BaseSkipList<Key, Satellite, Comparator, unique>::iterator iter = lower_bound(k);
482   while( iter.position != tail->data && comp.LessThanEquals( &(iter.position->get_key()), &k ) )
483     ++iter;
484   return iter;
486 }
489 //these implement both functions from Node and BasicSkipList class
492 template<typename Key, typename Satellite, typename Comparator>
493 void Node<Key, Satellite, Comparator>::print_node(Node<Key, Satellite, Comparator>* tail) {
495   if (data->key == NULL) std::cout << "Head: \n"; else
496   if (this == tail) std::cout << "Tail: \n"; else
497     std::cout << *data->key << ": \n";
498   for (unsigned int i = 0; i < right.size.logical_height(); i++)
499     std::cout << '"' << this->data[i] << '"' << data[i] << '"' << left[i] << '"';
502   std::cout << std::endl;
503   if (this != tail)
Associative containers with strong guarantees

```cpp
516 right[0]->print node(tail);
517 }
518
519 template<typename Key, typename Comparator>
520 void Node<Key,void,Comparator>::print node(Node<Key,void,Comparator>* tail) {
521 if (data->key == NULL)
522 std::cout << "Head:\n"; else
523 if (this == tail)
524 std::cout << "Tail:\n"; else
525 std::cout << *(data->key) << ":\n";
526 for (unsigned int i = 0; i < right_size.logical_height(); i++)
527 std::cout << ",\n";
528 std::cout << ",this:\n" << this << ",data:\n" << data << ",left:\n" << left;
529 std::cout << ",right:\n";
530 for (unsigned int i = 0; i < right_size.logical_height(); i++)
531 std::cout << this->right[i] << ",\n";
532 std::cout << std::endl;
533 if (this != tail)
534 right[0]->print node(tail);
535 }
536
537 template<typename Key, typename Satellite, typename Comparator, typename unique>
538 void BaseSkipList<Key,Satellite,Comparator,unique>::print() {
539 head->print node(tail);
540 }
```

Appendix A.3 ifthenelse.hpp

```cpp
/*
 * IFTHENELSE.HPP
 * Implemented from example in C++Templates by Vandevoorde and Josuttis
 */

#ifndef IFTHENELSE_HPP
#define IFTHENELSE_HPP

template<bool C, typename Ta, typename Tb>
class IfThenElse;

template<typename Ta, typename Tb>
class IfThenElse<true, Ta, Tb> {
    public:
    typedef Ta ResultT;
};

template<typename Ta, typename Tb>
class IfThenElse>false, Ta, Tb> {
    public:
    typedef Tb ResultT;
};
```
This can be used to define a general comparator:

```cpp
template<typename T>
class comparator{
  public:
    bool Compare(T a, T b) {
      return a < b;
    }
};
```

```cpp
template<typename T>
class IsVoid_T {
  public:
    enum { No = 1, Yes = 0 }; }
```

```cpp
template<> class IsVoid_T<void> {
  public:
    enum { Yes = 1, No = 0 }; }
```

```cpp
template<typename Key, typename Satellite, typename Comparator = comparator<Key>>
class UniqueTraits {
  public:
    enum { IsUnique = true }; typedef std::pair<Key, Satellite> key_value_type;
    typedef DataHandler<Key, Satellite, Comparator> compatible_datahandler;
    typedef Node<Key, Satellite, Comparator> compatible_node;

    static inline compatible_datahandler *CreateDataHandler(key_value_type k) {
      return new compatible_datahandler(k.first, k.second);
    }

    static inline compatible_node *CreateNode(key_value_type k) {
      return new compatible_node(new Key(k.first), new Satellite(k.second));
    }

    static inline const Key get_key(key_value_type k) {
      return k.first;
    }

    static inline compatible_node *CreateSentinel() {
      return new compatible_node(NULL, NULL, 2);
    }
};
```
Associative containers with strong guarantees

52
53
54
55 template<
56 typename Key, typename Comparator >
57 class UniqueTraits<
58 Key, void, Comparator >
59 {
60 public:
61 enum { IsUnique = true };
62
typedef Key key_value_type;
63
typedef DataHandler< Key, void, Comparator > compatible_datahandler;
64
typedef Node< Key, void, Comparator > compatible_node;
65
66 static inline compatible_datahandler *CreateDataHandler ( key_value_type k)
67 {
68 return new compatible_datahandler(k);
69 }
70
71 static inline const Key get_key ( key_value_type k)
72 {
73 return k;
74 }
75
76 static inline compatible_node *CreateSentinel ()
77 {
78 return new compatible_node(NULL, 2);
79 }
80
81 static inline compatible_node *CreateNode ( key_value_type k)
82 {
83 return new compatible_node(new Key(k));
84 }
85
86 template<
87 typename Key, typename Satellite, typename Comparator = comparator<
88 Key > >
89 class NonUniqueTraits
90 {
91 public:
92 enum { IsUnique = false };
93
typedef typename
94 IfThenElse<
95 IsVoid, T<Satellite >::No,
96 std::pair<
97 Key>::ResultT key_value_type;
98
typedef DataHandler< Key, Satellite , Comparator > compatible_datahandler;
99
typedef Node< Key, Satellite , Comparator > compatible_node;
100
101 static inline compatible_datahandler *CreateDataHandler ( key_value_type k)
102 {
103 return new compatible_datahandler(k.first, k.second);
104 }
105
106 static inline compatible_node *CreateNode ( key_value_type k)
107 {
108 return new compatible_node (new Key(k.first), new Satellite(k.second));
109 }
110
111 static inline Key get_key ( key_value_type k)
112 {
113 return k.first;
static inline compatible_node *CreateSentinel()
{
    return new compatible_node(NULL, NULL, 2);
}

};

template<
typename Key, typename Comparator>
class NonUniqueTraits<Key, void, Comparator>
{
public:

    enum {
        IsUnique = false
    }

typedef typename
    IfThenElse&lt;IsVoid_T&lt;void&gt;::No,
    std::pair&lt;Key, void&gt;,
    Key&gt;::ResultT key_value_type;

typedef DataHandler&lt;Key, void, Comparator &gt; compatible_datahandler;

typedef Node&lt;Key, void, Comparator &gt; compatible_node;

static inline compatible_datahandler *CreateDataHandler(key_value_type k)
{
    return new compatible_datahandler(k);
}

static inline Key get_key(key_value_type k)
{
    return k;
}

static inline compatible_node *CreateSentinel()
{
    return new compatible_node(NULL, 2);
}

static inline compatible_node *CreateNode(key_value_type k)
{
    return new compatible_node(new Key(k));
}

};

AppendixB. Testprograms

AppendixB.1 unittest.cpp
#ifndef __GNUC__
#define BREAKPOINT
{ __asm { int 0x03 } }
#else
#define BREAKPOINT
#endif

template <typename Key, typename Comparator>
bool UsableComparator<Key, Comparator>::UnitTest()
{
  UsableComparator<int, comparator<int>> intComparator;
  UsableComparator<float, comparator<float>> floatComparator;
  int a = 1, b = 2, c = 2;
  float f = 5.0, g = 4.0, h = 5.0;
  bool success = true;
  if ( !(intComparator.HelpUnitTest(&a,&b,&c) ) )
  {
    std::cout << "Above errors found in int" << std::endl;
    success = false;
  }
  if ( !(floatComparator.HelpUnitTest(&f,&g,&h) ) )
  {
    std::cout << "Above errors found in float" << std::endl;
    success = false;
  }
  return success;
}

template <typename Key, typename Comparator>
bool UsableComparator<Key, Comparator>::HelpUnitTest(Key *a, Key *b, Key *c)
{
  bool success = true;
  if( Equals(a,b) != (*a == *b) || Equals(c,b) != (*c == *b) )
  {
    std::cout << "Error in Equals!" << std::endl;
    success = false;
  }
  if( NotEquals(a,b) != (*a != *b) || NotEquals(c,b) != (*c != *b) )
  {
    std::cout << "Error in NotEquals!" << std::endl;
    success = false;
  }
  if( LessThan(a,b) != (*a < *b) || LessThan(b,a) != (*b < *a) || LessThan(c,b) != (*c < *b) )
  {
    std::cout << "Error in LessThan!" << std::endl;
    success = false;
  }
  if( LessThanEquals(a,b) != (*a <= *b) || LessThanEquals(b,a) != (*b <= *a) || LessThanEquals(c,b) != (*c <= *b) )
  {  
    
  }
```cpp
{  
  std::cout << "Error in LessThanEquals!" << std::endl;  
  success = false;  
}
if( GreaterThan(a,b) != (*a > *b)  
  || GreaterThan(b,a) != (*b > *a)  
  || GreaterThan(c,b) != (*c > *b) )  
{  
  std::cout << "Error in GreaterThan!" << std::endl;  
  success = false;  
}
if( GreaterThanEquals(a,b) != (*a >= *b)  
  || GreaterThanEquals(b,a) != (*b >= *a)  
  || GreaterThanEquals(c,b) != (*c >= *b) )  
{  
  std::cout << "Error in GreaterThanEquals!" << std::endl;  
  success = false;  
}
return success;
}
}                                 
/--------------------------------------------------------------------------  
***********Unit test of SkipList-Iterator*******************************  
***************************************************************************
template<typename Key, typename Satellite, typename Comparator, typename unique>  
bool BaseSkipList<Key, Satellite, Comparator, unique>::iterator::UnitTest()  
{  
  //success is true if no errors were found, set to false if an error is found  
  bool success = true;  
  // a few convinient typedefs  
typedef SkipList<int, void, comparator<int>, UniqueTraits<int, void, comparator<int>, slUniqueSet;  
typedef SkipList<int, void, comparator<int>, UniqueTraits<int, void, comparator<int>, > > slUniqueSet;  
// initialzing a few skip lists  
slUniqueSet *sll = new slUniqueSet();  
// initialzing a few iterators  
slUniqueIteratorSet iterla;  
slUniqueIteratorSet iterlb;  
// test constructor  
if( iterla.position != NULL )  
{  
  std::cout << "Error in default constructor of iterator!" << std::endl;  
  success = false;  
}
int v[7] = {4,5,8,10,42,42};
// build a sample skip list
s11->insert(5);
s11->insert(10);
s11->insert(4);
s11->insert(8);
s11->insert(42);

// test copy constructor and = with begin and end for s11
iter1a = s11->begin();
if( iter1a->position->get_key() != 4 )
{
    std::cout << "Error in copy constructor of iterator case 1!" << std::endl;
    success = false;
}
iter1a = s11->end();
if( iter1a->position->node != s11->tail )
{
    s11->print();
    std::cout << "Error in copy constructor of iterator case 2!" << std::endl;
    success = false;
}

// testing -- and ++
int positionInV = 5;
while( iter1a != s11->begin() )
{
    --iter1a;
    --positionInV;
    if( iter1a->position->get_key() != v[positionInV] )
    {
        std::cout << "Error in -- at key " << v[positionInV] << std::endl;
        success = false;
    }
}
while( iter1a != s11->end() )
{
    ++iter1a;
    ++positionInV;
    if( iter1a->position->get_key() != v[positionInV] )
    {
        std::cout << "Error in ++ at key " << v[positionInV] << std::endl;
        success = false;
    }
}

// testing == and !=
iter1b = s11->end();
if( !(iter1a == iter1b) )
{
    std::cout << "Error in ==" << std::endl;
    success = false;
}
if( iter1a != iter1b )
{
    std::cout << "Error in !=" << std::endl;
    success = false;
Morten Poulsen and Lars Skovlund

```cpp
183   iter1b = sl1->begin();
184   if( iter1a == iter1b )
185   {
186     std::cout << "Error \in \notin\" << std::endl;
187     success = false;
188   }
189   if( !(iter1a != iter1b) )
190   {
191     std::cout << "Error \notin\in\" << std::endl;
192     success = false;
193   }
194   return success;
195 }
196
197 /****************************************************************************
198 */
199 /*********************************************************/
200 /*********************************************************/
201 /*********************************************************/
202 template <typename Key, typename Satellite, typename Comparator, typename unique>
203 bool BaseSkipList<Key, Satellite, Comparator, unique>::UnitTest()
204 {
205   /* We will want to test constructors, then private helper functions,
206    * and then the rest. The constructor and the helper functions will be
207    * tested by the calling skip list (which is expected to be empty when
208    * UnitTest() is called and will be overwritten).
209    */
210
211   //success is true if no errors were found, set to false if an error is found
212   bool success = true;
213
214   // a few convinient typedefs
215   //for maps
216   typedef SkipList<int, int, comparator<int>, UniqueTraits<int, int, comparator<int>>, iterator>
217     sUniquerIterator; //allow only unique keys
218   typedef SkipList<int, int, comparator<int>, NonUniqueTraits<int, int, comparator<int>>, iterator>
219     sMultipleIterator; // allow multiple equal keys
220   typedef SkipList<int, int, comparator<int>, UniqueTraits<int, int, comparator<int>>, iterator>
221     sUnique; //allow only unique keys
222   typedef SkipList<int, int, comparator<int>, NonUniqueTraits<int, int, comparator<int>>, iterator>
223     sUniqueSet; // allow multiple equal keys
224   //for sets
225   typedef SkipList<int, void, comparator<int>, UniqueTraits<int, void, comparator<int>>, iterator>
226     sUniquerIteratorSet; //allow only unique keys
227   typedef SkipList<int, void, comparator<int>, NonUniqueTraits<int, void, comparator<int>>, iterator>
228     sMultipleIteratorSet; // allow multiple equal keys
229   typedef SkipList<int, void, comparator<int>, UniqueTraits<int, void, comparator<int>>, iterator>
230     sUniqueSet; // allow only unique keys
```
typedef SkipList< int, void, comparator<int> > slMultipleSet; // allow multiple equal keys
typedef Node<int, int, comparator<int> > slNode;

// initializing a few skip lists
slUnique *sl1 = new slUnique();
slMultiple *sl2 = new slMultiple();
slUniqueSet *sl3 = new slUniqueSet();
slMultipleSet *sl4 = new slMultipleSet();

// a lot of nodes that we will use later on
int i1=1, i2=2, i3=3, i4=4, i5=5, i6=6, i7=7,
    i8=8, i9=9, i10=10, i11=11, i12=12;
slNode* n1 = new slNode(new Key(i1), NULL, 1);
slNode* n2 = new slNode(new Key(i2), NULL, 1);
slNode* n3 = new slNode(new Key(i3), NULL, 2);
slNode* n4 = new slNode(new Key(i4), NULL, 1);
slNode* n5 = new slNode(new Key(i5), NULL, 1);
slNode* n6 = new slNode(new Key(i6), NULL, 1);
slNode* n7 = new slNode(new Key(i7), NULL, 1);
slNode* n8 = new slNode(new Key(i8), NULL, 2);
slNode* n9 = new slNode(new Key(i9), NULL, 1);
slNode* n10 = new slNode(new Key(i10), NULL, 1);
slNode* n11 = new slNode(new Key(i11), NULL, 2);
slNode* n12 = new slNode(new Key(i12), NULL, 1);

/******************** Test of constructor**************/
if( head->right_size.logical_height() != 2
    || head->right_size.logical_height() != 2
    || head->right[0] != tail || head->right[1] != tail
    || head->get_key() != NULL || head->left != NULL
    || tail->get_key() != NULL || tail->left != head )
    { print();
      std::cout << "Error in SkipList constructor!" << std::endl;
      success = false;
    }

/******************** Test of private helper functions**************/
/* make the following skip lists to test CalcGapSize */
++(head)->right_size;
++(tail)->right_size;
head->right[2] = tail;
head->right[1] = n3;
head->right[0] = n1;
n1->right[0] = n2;
n1->left = head;
n2->right[0] = n3;
n2->left = n1;
n3->right[1] = tail;
n3->right[0] = n4;
n3->left = n2;
n4->right[0] = tail;
n4->left = n3;
tail->left = n4;
tail->set_key(4);

if( CalcGapSize(head, 0) != 2
    || CalcGapSize(n3, 0) != 1
    || CalcGapSize(n3, -1) != 0 )
{
    std::cout << "CalcGapSize(head,0)\=i" << CalcGapSize(head, 0)
    << ",CalcGapSize(n3,0)\=i" << CalcGapSize(n3, 0)
    << ",CalcGapSize(n3,-1)\=i" << CalcGapSize(n3, -1) << std::endl;
    std::cout << "Error\in\_\_\_CalcGapSize\_\_\_first\_\_\_list!" << std::endl;
    success = false;
}

/*testing BorrowLeft (case 1)
 * 
 * BorrowLeft(head, n3, 0);
 */
if( head->right[2] != tail || head->right[1] != n2 || head->right[0] != n1
    || head->right_size.logical_height() != 3 || head->get_key() != NULL
    || n1->right[0] != n2 || n1->left != head
    || n1->right_size.logical_height() != 1 || *(n1->get_key()) != i1
    || n2->right[1] != tail || n2->right[0] != n3 || n2->left != n1
    || n2->right_size.logical_height() != 2 || *(n2->get_key()) != i2
    || n3->right[0] != n4 || n3->left != n2
    || n3->right_size.logical_height() != 1 || *(n3->get_key()) != i3
    || n4->right[0] != tail || n4->left != n3
    || n4->right_size.logical_height() != 1 || *(n4->get_key()) != i4
    || tail->left != n4 || tail->right_size.logical_height() != 3
    || *(tail->get_key()) != i4 )
{
    print();
    std::cout << "Error\in\_\_\_BorrowLeft\_\_\_case\_\_1!" << std::endl;
    success = false;
}

/*testing BorrowRight (case 1)
 * 
 * BorrowRight(n2, head, 0);
 */
if( head->right[2] != tail || head->right[1] != n3 || head->right[0] != n1
    || head->right_size.logical_height() != 3 || head->get_key() != NULL
    || n1->right[0] != n2 || n1->left != head
    || n1->right_size.logical_height() != 1 || *(n1->get_key()) != i1
    || n2->right[0] != n3 || n2->left != n1 || n2->right_size.logical_height() != 1
    || *(n2->get_key()) != i2
    || n3->right[1] != tail || n3->right[0] != n4 || n3->left != n2
    || n3->right_size.logical_height() != 2 || *(n3->get_key()) != i3
    || n4->right[0] != tail || n4->left != n3
    || n4->right_size.logical_height() != 1 || *(n4->get_key()) != i4
Associative containers with strong guarantees

43

|| tail->left != n4 || tail->right_size.logical_height() != 3
44|| *(tail->get_key()) != i4)
45{
46    print();
47    std::cout << "Error in BorrowRight case 1!" << std::endl;
48    success = false;
49}
50// Testing delete node when deleting a node of height 2
51* # # # # # # # # # # # # # # # # # # # # # # # # # # #
52DeleteNode(n3);
53* #note the peculiarity that n2 has now become n3
54* and that it's n2 that has been deleted*/
55if( head->right[2] != tail || head->right[1] != n3 || head->right[0] != n1
56    || head->right_size.logical_height() != 3 || head->get_key() != NULL
57    || n1->right[0] != n3 || n1->left != head
58    || n1->right_size.logical_height() != 1 || *(n1->get_key()) != i1
59    || n3->right[1] != tail || n3->right[0] != n4 || n3->left != n1
60    || n3->right_size.logical_height() != 2 || *(n3->get_key()) != i2
61    || n4->right[0] != tail || n4->left != n3
62    || n4->right_size.logical_height() != 1 || *(n4->get_key()) != i4
63    || tail->left != n4 || tail->right_size.logical_height() != 3
64    || *(tail->get_key()) != i4)
65{
66    print();
67    std::cout << "Error in DeleteNode case 2!" << std::endl;
68    success = false;
69}
70// now we change the list and continue the unit test
71head->right[2] = tail;
72head->right[1] = n8;
73head->right[0] = n5;
74n5->right[0] = n6;
75n5->left = head;
76n6->right[0] = n7;
77n6->left = n5;
78n7->right[0] = n8;
79n7->left = n6;
80n8->right[0] = tail;
81n8->right[0] = n9;
82n8->left = n7;
83n9->right[0] = tail;
84n9->left = n8;
85tail->left = n9;
86tail->set_key(9);
87// the last test of CalcGapSize
88if( CalcGapSize(head, 0) != 3 )
89{
90    std::cout << "Error in CalcGapSize second list! CalcGapSize returned: "
91    << CalcGapSize(head, 0) << ", expected 3" << std::endl;
92    success = false;
93}
94BorrowLeft(head, n8, 0);
if ( head->right[2] != tail || head->right[1] != n7 || head->right[0] != n5
    || head->right[1].logical_height() != 3 || head->get_key() != NULL
    || n5->right[0] != n6 || n5->left != head
    || n5->right[1].logical_height() != 1 || *(n5->get_key()) != i5
    || n6->right[0] != n7 || n6->left != n5
    || n6->right[1].logical_height() != 1 || *(n6->get_key()) != i6
    || n7->right[1] != tail || n7->right[0] != n8 || n7->left != n6
    || n7->right[1].logical_height() != 2 || *(n7->get_key()) != i7
    || n8->right[0] != n9 || n8->left != n7
    || n8->right[1].logical_height() != 1 || *(n8->get_key()) != i8
    || n9->right[0] != tail || n9->left != n8
    || n9->right[1].logical_height() != 1 || *(n9->get_key()) != i9
    || tail->left != n9 || tail->right->logical_height() != 3
    || *(tail->get_key()) != i9)
{
    print();
    std::cout << "Error in BorrowLeft case 2!" << std::endl;
    success = false;
}

// Testing BorrowRight (case 2)
* s g s g
* # # # # # # 1
* # # # # # # # # # # # # # # # # # # 0
*/

// increase height of n6 and decrease that of n7
head->right[1] = n6;
++n6->right->size;
n5->right[1] = tail;
n7->right[1] = NULL;
-n7->right->size;

BorrowRight(n6,head->0);

if ( head->right[2] != tail || head->right[1] != n7 || head->right[0] != n5
    || head->right[1].logical_height() != 3 || head->get_key() != NULL
    || n5->right[0] != n6 || n5->left != head || *(n5->get_key()) != i5
    || n5->right[1].logical_height() != 1
    || n6->right[0] != n7 || n6->left != n5
    || *(n6->get_key()) != i6
    || n6->right[1].logical_height() != 1
    || n7->right[1] != tail || n7->right[0] != n8
    || n7->right[1].logical_height() != 2 || *(n7->get_key()) != i7
    || n8->right[0] != n9
    || n8->right[1].logical_height() != 1
    || *(n8->get_key()) != i8
    || n9->right[0] != tail || n9->left != n8
    || n9->right[1].logical_height() != 1 || *(n9->get_key()) != i9
    || tail->left != n9
    || tail->right->logical_height() != 3
    || *(tail->get_key()) != i9)
{
    print();
    std::cout << "Error in BorrowRight case 2!" << std::endl;
    success = false;
}

*/ Testing IncreaseHeight and DecreaseHeight
* # # # 1
First, we decrease the height of the middle node, expecting a return value of true. Second, we try to decrease the height of the middle node, which should be impossible. Third, we try to increase the height of the middle node again, which should be impossible.

To ensure that the DecreaseHeight method correctly updates the height of the node, we perform the following checks:

- If the right subtree of the middle node is not equal to the tail, and the right subtree of the right child of the middle node is not equal to the middle node, and the left subtree of the right child of the middle node is not equal to the middle node, then we increase the height of the middle node.

- If the right subtree of the middle node is not equal to the tail, and the right subtree of the right child of the middle node is not equal to the middle node, and the left subtree of the right child of the middle node is not equal to the middle node, then we decrease the height of the middle node.

- If the right subtree of the middle node is not equal to the tail, and the right subtree of the right child of the middle node is not equal to the middle node, and the left subtree of the right child of the middle node is not equal to the middle node, then we print an error message and return false.

- If the right subtree of the middle node is not equal to the tail, and the right subtree of the right child of the middle node is not equal to the middle node, and the left subtree of the right child of the middle node is not equal to the middle node, then we print an error message and return false.

- If the right subtree of the middle node is not equal to the tail, and the right subtree of the right child of the middle node is not equal to the middle node, and the left subtree of the right child of the middle node is not equal to the middle node, then we print an error message and return false.

- If the right subtree of the middle node is not equal to the tail, and the right subtree of the right child of the middle node is not equal to the middle node, and the left subtree of the right child of the middle node is not equal to the middle node, then we print an error message and return false.

- If the right subtree of the middle node is not equal to the tail, and the right subtree of the right child of the middle node is not equal to the middle node, and the left subtree of the right child of the middle node is not equal to the middle node, then we print an error message and return false.

- If the right subtree of the middle node is not equal to the tail, and the right subtree of the right child of the middle node is not equal to the middle node, and the left subtree of the right child of the middle node is not equal to the middle node, then we print an error message and return false.

- If the right subtree of the middle node is not equal to the tail, and the right subtree of the right child of the middle node is not equal to the middle node, and the left subtree of the right child of the middle node is not equal to the middle node, then we print an error message and return false.

- If the right subtree of the middle node is not equal to the tail, and the right subtree of the right child of the middle node is not equal to the middle node, and the left subtree of the right child of the middle node is not equal to the middle node, then we print an error message and return false.
\begin{verbatim}
519  || head->right.size.logical_height() != 3 || head->get_key() != NULL
520  || n5->right[0] != n6 || n5->left != head
521  || n5->right.size.logical_height() != 1 || *(n5->get_key()) != i5
522  || n6->right[0] != n7 || n6->left != n5
523  || n6->right.size.logical_height() != 1 || *(n6->get_key()) != i6
524  || n7->right[1] != NULL || n7->right[0] != n8 || n7->left != n6
525  || n7->right.size.logical_height() != 2 || *(n7->get_key()) != i7
526  || n8->right[0] != n9 || n8->left != n7
527  || n8->right.size.logical_height() != 1 || *(n8->get_key()) != i8
528  || n9->right[0] != tail || n9->left != n8
529  || n9->right.size.logical_height() != 1 || *(n9->get_key()) != i9
530  || tail->left != n9 || tail->right.size.logical_height() != 3
531  || *(tail->get_key()) != i9)
532  
533  { print();
534  std::cout << "Error in MergeHeight!" << std::endl;
535  success = false;
536  }
537  
538  /* MergeLeft: */
539  
540  * g s g s
541  * ####### -> #######
542  * We build a new skip list, sL3 to use for this.
543  *
544  head->right[2] = tail;
545  head->right[1] = n11;
546  head->right[0] = n10;
547  n10->right[0] = n11;
548  n10->left = head;
549  n11->right[1] = tail;
550  n11->right[0] = n12;
551  n11->left = n10;
552  n12->right[0] = tail;
553  n12->left = n11;
554  tail->left = n12;
555  tail->set_key(12);
556  
557  MergeLeft(head,n11,0);
558  if( head->right[1] != tail || head->right[0] != n10
559  || head->right.size.logical_height() != 2 || head->get_key() != NULL
560  || n10->right[0] != n11 || n10->left != head
561  || n10->right.size.logical_height() != 1 || *(n10->get_key()) != i10
562  || n11->right[0] != n12 || n11->left != n10
563  || n11->right.size.logical_height() != 1 || *(n11->get_key()) != i11
564  || n12->right[0] != tail || n12->left != n11
565  || n12->right.size.logical_height() != 1 || *(n12->get_key()) != i12
566  || tail->left != n12 || tail->right.size.logical_height() != 2
567  || *(tail->get_key()) != i12)
568  
569  { print();
570  std::cout << "Error in MergeLeft!" << std::endl;
571  success = false;
572  }
573  
574  /* MergeRight:
\end{verbatim}
Associative containers with strong guarantees

We edit our skip list

MergeRight(n11, head, 0);

if (head->right[1] != tail || head->right[0] != n10
|| head->right_size.logical_height() != 2 || head->get_key() != NULL
|| n10->right[0] != n11 || n10->left != head
|| n10->right_size.logical_height() != 1 || *(n10->get_key()) != i10
|| n11->right[0] != n12 || n11->left != n10
|| n11->right_size.logical_height() != 1 || *(n11->get_key()) != i11
|| n12->right[0] != tail || n12->left != n11
|| n12->right_size.logical_height() != 1 || *(n12->get_key()) != i12
|| tail->left != n12 || tail->right_size.logical_height() != 2
|| *(tail->get_key()) != i12)
{
  print();
  std::cout << "Error in MergeRight!" << std::endl;
  success = false;
}

DeleteNode and UpdatePointers (and SplitGap):

First, we delete n11 from sl3, using DeleteNode
* # # # # #
Second, we re-insert the node with UpdatePointers
* # # # #
* # # # # #
Third, we raise the height of n11 with SplitGap
* # # # # #
* # # # # #
* # # # # # # # # # # # # # #
Fourth, we delete n11 with DeleteNode. As sl3 is not ready for deletion
* (invariant not maintained), we use sl1 in stead
* # # # # # # # # # # # # # #
* # # # # # # # # # # # # # #
*

DeleteNode(n11);

if (head->right[1] != tail || head->right[0] != n10
|| head->right_size.logical_height() != 2 || head->get_key() != NULL
|| n10->right[0] != n12 || n10->left != head
|| n10->right_size.logical_height() != 1 || *(n10->get_key()) != i10
|| n12->right[0] != tail || n12->left != n10
|| n12->right_size.logical_height() != 1 || *(n12->get_key()) != i12
|| tail->left != n12 || tail->right_size.logical_height() != 2
|| *(tail->get_key()) != i12)
{
  print();
  std::cout << "Error in DeleteNode case 1!" << std::endl;
success = false;
}
n11 = new sllNode(new Key(i11), NULL, 1);
UpdatePointers(n10, n11, n12);
if (head->right[1] != tail || head->right[0] != n10
         || head->right.size.logical_height() != 2 || head->get_key() != NULL
         || n10->right[0] != n11 || n10->left != head
         || n10->right.size.logical_height() != 1 || *(n10->get_key()) != i10
         || n11->right[0] != n12 || n11->left != n10
         || n11->right.size.logical_height() != 1 || *(n11->get_key()) != i11
         || n12->right[0] != tail || n12->left != n11
         || n12->right.size.logical_height() != 1 || *(n12->get_key()) != i12
         || tail->left != n12 || tail->right.size.logical_height() != 2
         || *(tail->get_key()) != i12)
{
    print();
    std::cout << "Error in UpdatePointers!" << std::endl;
    success = false;
}

// increasing height of n11 - this is an excellent test of SplitGap
SplitGap(head, 0);
if (head->right[2] != tail || head->right[1] != n11 || head->right[0] != n10
         || head->right.size.logical_height() != 3 || head->get_key() != NULL
         || n10->right[0] != n11 || n10->left != head
         || n10->right.size.logical_height() != 1 || *(n10->get_key()) != i10
         || n11->right[1] != tail || n11->right[0] != n12
         || n11->left != n10 || n11->right.size.logical_height() != 2
         || *(n11->get_key()) != i11 || n12->right[0] != tail || n12->left != n11
         || n12->right.size.logical_height() != 1 || *(n12->get_key()) != i12
         || tail->left != n12 || tail->right.size.logical_height() != 3
         || *(tail->get_key()) != i12)
{
    print();
    std::cout << "Error in SplitGap!" << std::endl;
    success = false;
}

******** Test of public work functions************

******** Test of insert, when key must be unique************
/*Note that the private function Insert is tested by using the public methods
* in the first two insertions, we will also check the return values from insert*/

/* Testing insertion of first element into an empty skip list*/
std::pair< sllUniqueIterator , bool> pair1 = sll1->insert(5,1);
if (sll1->head->right.size.logical_height() != 2 || sll1->head->get_key() != NULL
         || sll1->head->left != NULL
         || sll1->head->right[1] != sll1->tail || *(sll1->head->right[0]->get_key()) != 5
         || sll1->head->right[0]->right[0] != sll1->tail
         || sll1->head->right.size.logical_height() != 1
         || sll1->tail->right.size.logical_height() != 2
         || *(sll1->tail->get_key()) != 5 || (sll1->tail->left != sll1->head->right[0])
         || (*(pair1.first)).first != 5 || (*(pair1.first)).second != 1
         || (pair1.second != true)
#include <iostream>
#include <string>

class DataHandler {
public:
    DataHandler() : head(10), tail(NULL) {}

    bool insert(int key, int value) {
        std::cout << "Error in SkipList unique insert case 1. List print below: " << std::endl;
        std::cout << std::endl;
        std::cout << std::endl;
        return false;
    }

private:
    int head;
    int tail;
};

// Insertion of 3rd element that is smaller than the original elements
if ( !sl1->HelpUnitTest(sl1->head,NULL,2,NULL,5,10,NULL) ) {
    std::cout << "Error in SkipList unique insert case 2. List print below: " << std::endl;
    std::cout << std::endl;
    return false;
}

// Insertion of 2nd element that is larger than the original elements
if ( !sl1->HelpUnitTest(sl1->head,NULL,2,NULL,5,10,NULL) ) {
    std::cout << "Error in SkipList unique insert case 3. List print below: " << std::endl;
    std::cout << std::endl;
    return false;
}

// Insertion of 1st element that is different than the original elements
if ( !sl1->HelpUnitTest(sl1->head,NULL,2,NULL,5,10,NULL) ) {
    std::cout << "Error in SkipList unique insert case 4. List print below: " << std::endl;
    std::cout << std::endl;
    return false;
}

// Testing multiple insertion of key in unique-key skip list
* note that the pointer to the data must be unchanged!/
if ( !sl1->insert(5,2) ) {
    std::cout << "Error in SkipList unique insert case 5. List print below: " << std::endl;
    std::cout << std::endl;
    return false;
}

int main() {
    return 0;
}
743 }
744 // insertion of 4th element, should call a SplitGap*/
745 pair2 = sl1->insert(8, 4);
746 if( !sl1->HelpUnitTest(sl1->head, NULL, 3, NULL, 4, 5, 10)  
747     || !sl1->HelpUnitTest(sl1->head->right[0], 4, 1, NULL, 5, NULL, NULL)  
748     || !sl1->HelpUnitTest(sl1->head->right[0]->right[0], 5, 2, 4, 8, 10, NULL)  
749     || !sl1->HelpUnitTest(sl1->head->right[0]->right[0]->right[0], 8, 1, 5, 10,  
750         NULL, NULL)  
751     || !sl1->HelpUnitTest(sl1->head->right[0]->right[0]->right[0]->right[0],  
752         10, 8, 10, NULL, NULL)  
753     || !sl1->HelpUnitTest(sl1->tail, 10, 3, 10, NULL, NULL, NULL))  
754 {
755     std::cout << "Error in SkipList insert unique case 5. List print below: "  
756     << std::endl;
757     sl1->print();
758     success = false;
759 }
760 // insertion of 5th element, just before tail*/
761 pair2 = sl1->insert(42, 5);
762 if( !sl1->HelpUnitTest(sl1->head, NULL, 3, NULL, 4, 5, 42)  
763     || !sl1->HelpUnitTest(sl1->head->right[0], 4, 1, NULL, 5, NULL, NULL)  
764     || !sl1->HelpUnitTest(sl1->head->right[0]->right[0], 5, 2, 4, 8, 42, NULL)  
765     || !sl1->HelpUnitTest(sl1->head->right[0]->right[0]->right[0], 8, 1, 5, 10,  
766         NULL, NULL)  
767     || !sl1->HelpUnitTest(sl1->head->right[0]->right[0]->right[0]->right[0],  
768         10, 1, 8, 42, NULL, NULL)  
769     || !sl1->HelpUnitTest(sl1->head->right[0]->right[0]->right[0]->right[0]-  
770         right[0], 42, 10, 42, NULL, NULL)  
771     || !sl1->HelpUnitTest(sl1->tail, 42, 3, 42, NULL, NULL, NULL))  
772 {
773     std::cout << "Error in SkipList insert unique case 6. List print below: "  
774     << std::endl;
775     sl1->print();
776     success = false;
777 }
778 // trying to insert 42 again - 42 will not be inserted,  
779 * but last gap should be split*/
780 pair2 = sl1->insert(42, 5);
781 if( !sl1->HelpUnitTest(sl1->head, NULL, 3, NULL, 4, 5, 42)  
782     || !sl1->HelpUnitTest(sl1->head->right[0], 4, 1, NULL, 5, NULL, NULL)  
783     || !sl1->HelpUnitTest(sl1->head->right[0]->right[0], 5, 2, 4, 8, 10, NULL)  
784     || !sl1->HelpUnitTest(sl1->head->right[0]->right[0]->right[0], 8, 1, 5, 10,  
785         NULL, NULL)  
786     || !sl1->HelpUnitTest(sl1->head->right[0]->right[0]->right[0]->right[0]-  
787         right[0], 10, 2, 8, 42, NULL)  
788     || !sl1->HelpUnitTest(sl1->head->right[0]->right[0]->right[0]->right[0]-  
789         right[0], 42, 1, 10, 42, NULL, NULL)  
790     || !sl1->HelpUnitTest(sl1->tail, 42, 3, 42, NULL, NULL, NULL))  
791 {
792     std::cout << "Error in SkipList insert unique case 7. List print below: "  
793     << std::endl;
794     sl1->print();
795     success = false;
796 }
797
798 /************ Test of erase, when key must be unique************/
*/ delete 42 to test if we can reach MergeLeft
* note that the list has history: tails key is the maximum key inserted*/
sl1->erase (42);
if ( !sl1->HelpUnitTest (sl1->head,NULL,3,NULL,4,5,42)
  || !sl1->HelpUnitTest (sl1->head->->right[0],4,1,NULL,5,NULL,NULL)
  || !sl1->HelpUnitTest (sl1->head->->right[0]->->right[0],5,2,NULL,8,42,NULL)
  || !sl1->HelpUnitTest (sl1->head->->right[0]->->right[0]->->right[0],8,1,NULL
      ,8,1,5,10,42,NULL,NULL)
  || !sl1->HelpUnitTest (sl1->head->->right[0]->->right[0]->->right[0]->->right[0
      ,10,1,8,42,NULL,NULL)
  [!sl1->HelpUnitTest (sl1->tail,42,3,10,NULL,NULL,NULL) )
  { std::cout << "Error in SkipList delete unique case 1. List print below: ":
    << std::endl;
    sl1->print ( );
    success = false;
  }
*/ delete 4, expect a BorrowRight*/
sl1->erase (4);
if ( !sl1->HelpUnitTest (sl1->head,NULL,3,NULL,5,8,42)
  || !sl1->HelpUnitTest (sl1->head->->right[0],5,1,NULL,8,NULL,NULL)
  || !sl1->HelpUnitTest (sl1->head->->right[0]->->right[0],8,2,5,10,42,NULL)
  || !sl1->HelpUnitTest (sl1->head->->right[0]->->right[0]->->right[0],10,1,8,42
      ,NULL,NULL)
  || !sl1->HelpUnitTest (sl1->tail,42,3,10,NULL,NULL,NULL) )
  { std::cout << "Error in SkipList delete unique case 2. List print below: ":
    << std::endl;
    sl1->print ( );
    success = false;
  }
*/ delete 5, expect a MergeRight*/
sl1->erase (5);
if ( !sl1->HelpUnitTest (sl1->head,NULL,2,NULL,8,42,NULL)
  || !sl1->HelpUnitTest (sl1->head->->right[0],8,1,NULL,10,NULL,NULL)
  || !sl1->HelpUnitTest (sl1->head->->right[0]->->right[0],10,1,8,42,NULL,NULL)
  || !sl1->HelpUnitTest (sl1->tail,42,2,10,NULL,NULL,NULL) )
  { std::cout << "Error in SkipList delete unique case 3. List print below: ":
    << std::endl;
    sl1->print ( );
    success = false;
  }
*/ delete 8 and 10 to get the empty skip list*/
sl1->erase (8);
sl1->erase (10);
if ( !sl1->HelpUnitTest (sl1->head,NULL,2,NULL,42,42,NULL)
  || !sl1->HelpUnitTest (sl1->tail,42,2,NULL,NULL,NULL,NULL) )
  { std::cout << "Error in SkipList delete unique case 4. List print below: ":
    << std::endl;
    sl1->print ( );
    success = false;
  }
*/now, we'll test for errors on the top height in the skip list
We'll try to delete the first, second and third node to see if the result is invariant

```
sl1->insert(2,1);
sl1->insert(4,1);
sl1->insert(6,1); // raise middle element
// deleting 2
if( !sl1->HelpUnitTest(sl1->head, NULL, 2,NULL, 4, 42,NULL)
  | !sl1->HelpUnitTest(sl1->head->right[0]->right[0], 6, 1, 4, 42,NULL, NULL)
  | !sl1->HelpUnitTest(sl1->tail, 42,2,6,NULL, NULL, NULL) )
{
    std::cout << "Error in SkipList delete unique case 5. List print below:"
    << std::endl;
    sl1->print();
    success = false;
}
```

```
sl1->erase(2);
```

```
sl1->insert(2,1);
sl1->insert(2,1);
// deleting 4
if( !sl1->HelpUnitTest(sl1->head, NULL, 2,NULL, 2, 42,NULL)
  | !sl1->HelpUnitTest(sl1->head->right[0], 2, 1,NULL, 6,NULL, NULL)
  | !sl1->HelpUnitTest(sl1->head->right[0], 6, 1, 2, 42,NULL, NULL)
  | !sl1->HelpUnitTest(sl1->tail, 42,2,6,NULL, NULL, NULL) )
{
    std::cout << "Error in SkipList delete unique case 6. List print below:"
    << std::endl;
    sl1->print();
    success = false;
}
```

```
sl1->erase(4);
```

```
sl1->insert(4,1);
sl1->insert(4,1);
// deleting 6
if( !sl1->HelpUnitTest(sl1->head, NULL, 2,NULL, 2, 42,NULL)
  | !sl1->HelpUnitTest(sl1->head->right[0], 2, 1,NULL, 4,NULL, NULL)
  | !sl1->HelpUnitTest(sl1->head->right[0], 4, 1, 2, 42,NULL, NULL)
  | !sl1->HelpUnitTest(sl1->tail, 42,2,4,NULL, NULL, NULL) )
{
    std::cout << "Error in SkipList delete unique case 7. List print below:"
    << std::endl;
    success = false;
}
```

```
/******* Test of insert, when key isn't unique***********/
/* We make the same inserts as before, but the resulting skip list will
* be somewhat different, as we now allow duplicate keys*/
std::pair<slMultipleIterator, bool> pairMap1 = sl2->insert(5,1);
if( !sl2->HelpUnitTest(sl2->head, NULL, 2,NULL, 5, 5,NULL)
  | !sl2->HelpUnitTest(sl2->head->right[0], 5, 1,NULL, 5,NULL, NULL)
  | !sl2->HelpUnitTest(sl2->tail, 5,2,5,NULL, NULL, NULL) )
```
Associative containers with strong guarantees

911  || (pairMap1.first).first != 5 || *(pairMap1.first).second != 1
912  || (pairMap1.second != true)
913 {
914     std::cout << "Error in SkipList insert nonunique case 1. List print below:
915     << std::endl;
916     sl2->print();
917     success = false;
918 }
919     std::pair<slMultipleIterator, bool> pairMap2 = sl2->insert(5, 2);
920     if( !sl2->helpUnitTest(sl2->head, NULL, 2, NULL, 5, 5, NULL)
921         || !sl2->helpUnitTest(sl2->head->right[0], 5, 1, NULL, 5, NULL, NULL)
922         || !sl2->helpUnitTest(sl2->head->right[0]->right[0], 5, 1, 5, 5, NULL, NULL)
923         || !sl2->helpUnitTest(sl2->head->right[0]->right[0]->right[0], 5, 1, 5, 10, NULL, NULL)
924         || *(pairMap2.first).first != 5 || *(pairMap2.first).second != 2
925         || (pairMap2.second != true)
926 {
927     std::cout << "Error in SkipList insert nonunique case 2. List print below:
928     << std::endl;
929     sl2->print();
930     success = false;
931 }
932     pairMap1 = sl2->insert(10, 1);
933     if( !sl2->helpUnitTest(sl2->head, NULL, 2, NULL, 5, 10, NULL)
934         || !sl2->helpUnitTest(sl2->head->right[0], 5, 1, NULL, 5, NULL, NULL)
935         || !sl2->helpUnitTest(sl2->head->right[0]->right[0], 5, 1, 5, 10, NULL, NULL)
936         || !sl2->helpUnitTest(sl2->head->right[0]->right[0]->right[0], 5, 1, 5, 10, 10, NULL, NULL)
937         || !sl2->helpUnitTest(sl2->tail, 10, 2, 10, NULL, NULL, NULL) )
938 {
939     std::cout << "Error in SkipList insert nonunique case 3. List print below:
940     << std::endl;
941     sl2->print();
942     success = false;
943 }
944     pairMap2 = sl2->insert(4, 2);
945     if( !sl2->helpUnitTest(sl2->head, NULL, 3, NULL, 4, 5, 10)
946         || !sl2->helpUnitTest(sl2->head->right[0], 4, 1, NULL, 5, NULL, NULL)
947         || !sl2->helpUnitTest(sl2->head->right[0]->right[0], 5, 1, 4, 5, NULL, NULL)
948         || !sl2->helpUnitTest(sl2->head->right[0]->right[0]->right[0], 5, 1, 4, 5, 10, NULL, NULL)
949         || !sl2->helpUnitTest(sl2->head->right[0]->right[0]->right[0]->right[0], 5, 1, 4, 5, 10, 10, NULL, NULL)
950         || !sl2->helpUnitTest(sl2->head->right[0]->right[0]->right[0]->right[0], 5, 2, 5, 10, NULL)
951         || !sl2->helpUnitTest(sl2->head->right[0]->right[0]->right[0]->right[0], 5, 2, 5, 10, NULL)
952         || !sl2->helpUnitTest(sl2->tail, 10, 3, 10, NULL, NULL, NULL) )
953 {
954     std::cout << "Error in SkipList insert nonunique case 4. List print below:
955     << std::endl;
956     sl2->print();
957     success = false;
958 }
959     pairMap2 = sl2->insert(8, 2);
960     if( !sl2->helpUnitTest(sl2->head, NULL, 3, NULL, 4, 5, 10)
961         || !sl2->helpUnitTest(sl2->head->right[0], 4, 1, NULL, 5, NULL, NULL)
962         || !sl2->helpUnitTest(sl2->head->right[0]->right[0], 5, 1, 4, 5, NULL, NULL)
963         || !sl2->helpUnitTest(sl2->head->right[0]->right[0]->right[0], 5, 1, 4, 5, 10, NULL)
964         || !sl2->helpUnitTest(sl2->head->right[0]->right[0]->right[0]->right[0], 5, 1, 4, 5, 10, NULL)
965         || !sl2->helpUnitTest(sl2->head->right[0]->right[0]->right[0]->right[0], 5, 2, 5, 8, 10, NULL)
966         || !sl2->helpUnitTest(sl2->head->right[0]->right[0]->right[0]->right[0], 5, 2, 5, 8, 10, NULL)
967         || !sl2->helpUnitTest(sl2->head->right[0]->right[0]->right[0]->right[0], 5, 2, 5, 8, 10, NULL)
968         || !sl2->helpUnitTest(sl2->head->right[0]->right[0]->right[0]->right[0], 5, 2, 5, 8, 10, NULL)
969         || !sl2->helpUnitTest(sl2->head->right[0]->right[0]->right[0]->right[0], 5, 2, 5, 8, 10, NULL)
8, 1, 5, 10, NULL, NULL)
968 | ! sl2->HelpUnitTest(sl2->head->right[0] -> right[0] -> right[0] -> right[0] -> right[0]
969 -> right[0], 10, 1, 8, 10, NULL, NULL)
970 | ! sl2->HelpUnitTest(sl2->tail, 10, 3, 10, NULL, NULL, NULL)
971 { std::cout << "Error in SkipList insert nonunique case 5. List print below:"
972 << std::endl;
973 sl2->print();
974 success = false;
975 }
976 pairMap2 = sl2->insert(42, 2);
977 if( ! sl2->HelpUnitTest(sl2->head, NULL, 3, NULL, 4, 5, 42)
978 | ! sl2->HelpUnitTest(sl2->head->right[0], 4, 1, NULL, 5, NULL, NULL)
979 | ! sl2->HelpUnitTest(sl2->head->right[0] -> right[0] -> right[0] -> right[0] -> right[0] ->
980 right[0], 5, 1, 4, 5, NULL, NULL)
981 | ! sl2->HelpUnitTest(sl2->head->right[0] -> right[0] -> right[0] -> right[0] -> right[0] ->
982 right[0] -> right[0], 8, 1, 5, 10, NULL, NULL)
983 | ! sl2->HelpUnitTest(sl2->head->right[0] -> right[0] -> right[0] -> right[0] -> right[0] ->
984 right[0] -> right[0] -> right[0] -> right[0] -> right[0] ->
985 right[0] -> right[0] -> right[0] -> right[0] -> right[0] ->
986 right[0] -> right[0] -> right[0] -> right[0] -> right[0] ->
987 right[0], 42, 1, 10, NULL, NULL, NULL)
988 | ! sl2->HelpUnitTest(sl2->tail, 42, 3, 42, NULL, NULL, NULL) )
989 {
990 std::cout << "Error in SkipList insert nonunique case 6. List print below:"
991 << std::endl;
992 sl2->print();
993 success = false;
994 }
995 pairMap2 = sl2->insert(42, 2);
996 if( ! sl2->HelpUnitTest(sl2->head, NULL, 3, NULL, 4, 5, 42)
997 | ! sl2->HelpUnitTest(sl2->head->right[0], 4, 1, NULL, 5, NULL, NULL)
998 | ! sl2->HelpUnitTest(sl2->head->right[0] -> right[0] -> right[0] -> right[0] -> right[0] ->
999 right[0], 5, 1, 4, 5, NULL, NULL)
1000 | ! sl2->HelpUnitTest(sl2->head->right[0] -> right[0] -> right[0] -> right[0] -> right[0] ->
1001 right[0] -> right[0], 8, 1, 5, 10, NULL, NULL)
1002 | ! sl2->HelpUnitTest(sl2->head->right[0] -> right[0] -> right[0] -> right[0] -> right[0] ->
1003 right[0] -> right[0] -> right[0] -> right[0] -> right[0] ->
1004 right[0] -> right[0] -> right[0] -> right[0] -> right[0] ->
1005 right[0] -> right[0] -> right[0] -> right[0] -> right[0] ->
1006 right[0] -> right[0] -> right[0] -> right[0] -> right[0] ->
1007 right[0] -> right[0] -> right[0] -> right[0] -> right[0] ->
1008 right[0] -> right[0] -> right[0] -> right[0], 42, 1, 42, NULL, NULL, NULL)
1009 | ! sl2->HelpUnitTest(sl2->tail, 42, 3, 42, NULL, NULL, NULL) )
1010 {
1011 std::cout << "Error in SkipList insert nonunique case 7. List print below:"
1012 << std::endl;
1013 sl2->print();
1014 success = false;
1015 }
1016 /* we delete 42, 8 and 10 from sl2, as this will
1017 * fire a BorrowLeft on the last erase */
1018 sl2->erase(42);
1019 sl2->erase(8);
1020 if( ! sl2->HelpUnitTest(sl2->head, NULL, 3, NULL, 4, 5, 42)
1021 | ! sl2->HelpUnitTest(sl2->head->right[0], 4, 1, NULL, 5, NULL, NULL)
1022 | ! sl2->HelpUnitTest(sl2->head->right[0] -> right[0] -> right[0] -> right[0] ->
1023 right[0] -> right[0] -> right[0] -> right[0] -> right[0] ->
1024 right[0] -> right[0] -> right[0] -> right[0] -> right[0] ->
1025 right[0] -> right[0] -> right[0] -> right[0], 5, 2, 4, 5, 42, NULL)
Associative containers with strong guarantees

1023  // Setup HelpUnitTest sl2
1024  HelpUnitTest sl2;
1025  HelpUnitTest sl2;
1026  {
1027    std::cout << "Error in SkipList erase nonunique. List print below:
1028    std::endl;
1029    sl2->print();
1030    success = false;
1031  }
1032
1033  /* Testing set unique insertion — this will be done quickly*/
1034  sl3->insert(5);
1035  sl3->insert(5);
1036  sl3->insert(10);
1037  sl3->insert(4);
1038  sl3->insert(8);
1039  sl3->insert(42);
1040  sl3->insert(42);
1041  if( !sl3->HelpUnitTest(sl3->head,NULL,3,NULL,42,NULL))
1042  {
1043    std::cout << "Error in SkipList set insert unique. List print below:
1044    std::endl;
1045    sl3->print();
1046    success = false;
1047  }
1048
1049  /* Testing set unique erase — this will be done quickly*/
1050  sl3->erase(42);
1051  sl3->erase(4);
1052  sl3->erase(5);
1053  sl3->erase(8);
1054  sl3->erase(10);
1055  if( !sl3->HelpUnitTest(sl3->head,NULL,2,NULL,42,NULL))
1056  {
1057    std::cout << "Error in SkipList set erase unique. List print below:
1058    std::endl;
1059    sl3->print();
1060    success = false;
1061  }
1062
1063  /* Testing set non unique insertion — this will be done quickly*/
1064  sl4->insert(5);
1065  sl4->insert(5);
1066  sl4->insert(10);
1067  sl4->insert(4);
1068  sl4->insert(8);
Morten Poulsen and Lars Skovlund

```cpp
sl4->insert(42);
sl4->insert(42);
if (!sl4->HelpUnitTest(sl4->head, NULL, 3, NULL, 4, 5, 42)
    || !sl4->HelpUnitTest(sl4->head->right[0], 4, 1, NULL, 5, NULL, NULL)
    || !sl4->HelpUnitTest(sl4->head->right[0]->right[0], 5, 1, 4, 5, NULL, NULL)
    || !sl4->HelpUnitTest(sl4->head->right[0]->right[0]->right[0], 5, 2, 5, 8, 10, NULL)
    || !sl4->HelpUnitTest(sl4->head->right[0]->right[0]->right[0]->right[0], 8, 1, 5, 10, NULL, NULL)
    || !sl4->HelpUnitTest(sl4->head->right[0]->right[0]->right[0]->right[0]->right[0], 42, 1, 7, 10, 10, NULL, NULL)
    || !sl4->HelpUnitTest(sl4->head->right[0]->right[0]->right[0]->right[0]->right[0]->right[0], 42, 2, 42, 42, NULL)
    || !sl4->HelpUnitTest(sl4->head->right[0]->right[0]->right[0]->right[0]->right[0]->right[0]->right[0], 42, 1, 42, NULL, NULL, NULL)
    || !sl4->HelpUnitTest(sl4->head->right[0]->right[0]->right[0]->right[0]->right[0]->right[0]->right[0]->right[0], 42, 2, 42, NULL, NULL)
    || !sl4->HelpUnitTest(sl4->head->right[0]->right[0]->right[0]->right[0]->right[0]->right[0]->right[0]->right[0]->right[0], 42, 3, 42, NULL, NULL, NULL)
    || !sl4->HelpUnitTest(sl4->tail, 42, 3, 42, NULL, NULL, NULL))
{
    std::cout << "Error in SkipList insert nonunique case 7. List print below: \n"
        << std::endl;
    sl4->print();
    success = false;
}

/* Testing lower bound on sl4, we search for 2, 5, 8, 43 */
sl4->lower_bound(2);
if (mit1->position->get_key() != 4)
{
    std::cout << "Error in SkipList lower_bound case 1. List print below: \n"
        << std::endl;
    sl4->print();
    success = false;
}
mit1 = sl4->lower_bound(5);
if (mit1->position->get_key() != 5)
{
    std::cout << "Error in SkipList lower_bound case 2. List print below: \n"
        << std::endl;
    sl4->print();
    success = false;
}
mit1 = sl4->lower_bound(8);
if (mit1->position->get_key() != 8)
{
    std::cout << "Error in SkipList lower_bound case 3. List print below: \n"
        << std::endl;
    sl4->print();
    success = false;
}
mit1 = sl4->lower_bound(43);
if (mit1->position != sl4->tail->data)
{
    std::cout << "Error in SkipList lower_bound case 4. List print below: \n"
        << std::endl;
    sl4->print();
    success = false;
}
```
/* Testing upper_bound on sl4, we search for 2, 5, 8, 43 */
mit1 = sl4->upper_bound(2);
if ( mit1.position->get_key() != 4 )
{
    std::cout << "Error in SkipList upper_bound case 1. List print below:
" << std::endl;
    sl4->print();
    success = false;
}
mit1 = sl4->upper_bound(5);
if ( mit1.position->get_key() != 8 )
{
    std::cout << "Error in SkipList upper_bound case 2. List print below:
" << std::endl;
    sl4->print();
    success = false;
}
mit1 = sl4->upper_bound(8);
if ( mit1.position->get_key() != 10 )
{
    std::cout << "Error in SkipList upper_bound case 3. List print below:
" << std::endl;
    sl4->print();
    success = false;
}
mit1 = sl4->upper_bound(43);
if ( mit1.position != sl4->tail->data )
{
    std::cout << "Error in SkipList upper_bound case 4. List print below:
" << std::endl;
    sl4->print();
    success = false;
}

/* Testing find on sl4, we search for 2, 5, 8, 43 */
mit1 = sl4->find(2);
if ( mit1.position != sl4->tail->data )
{
    std::cout << "Error in SkipList find case 1. List print below:
" << std::endl;
    sl4->print();
    success = false;
}
mit1 = sl4->find(5);
if ( mit1.position->get_key() != 5 )
{
    std::cout << "Error in SkipList find case 2. List print below:
" << std::endl;
    sl4->print();
    success = false;
}
mit1 = sl4->find(8);
if ( mit1.position->get_key() != 8 )
{
    std::cout << "Error in SkipList find case 3. List print below:
" << std::endl;
1191     sl4->print();
1192     success = false;
1193 }
1194 mit1 = sl4->find(43);
1195 if ( mit1.position != sl4->tail->data )
1196 {
1197     std::cout << "Error in SkipList find case 4. List print below:
1198       " << std::endl;
1199     sl4->print();
1200     success = false;
1201 }
1202
1203 /* Testing set non unique erase*/
1204 int erase42 = sl4->erase(42); // return 2
1205 int erase5 = sl4->erase(5); // return 2
1206 int erase10 = sl4->erase(10); // return 1
1207 int erase1 = sl4->erase(1); // return 0
1208 if ( erase42 != 2 || erase10 != 1 || erase5 != 2 || erase1 != 0
1209  || !sl4->HelpUnitTest(sl4->head,NULL,2,NULL,4,42,NULL)
1210  || !sl4->HelpUnitTest(sl4->head->right[0],4,1,NULL,8,NULL,NULL)
1211  || !sl4->HelpUnitTest(sl4->head->right[0]->right[0],8,1,4,42,
1212    NULL,NULL)
1213  || !sl4->HelpUnitTest(sl4->tail,42,2,8,NULL,NULL,NULL) )
1214 {
1215     std::cout << "Error in SkipList delete nonunique. List print below:
1216       " << std::endl;
1217     sl4->print();
1218     success = false;
1219 }
1220
1221 /* Testing count with sl2 (tail contains 42)*/
1222 */
1223 if ( sl2->count(1) != 0 )
1224 {
1225     std::cout << "Error in SkipList count case 1. List print below:
1226       " << std::endl;
1227     sl2->print();
1228     success = false;
1229 }
1230 if ( sl2->count(4) != 1 )
1231 {
1232     std::cout << "Error in SkipList count case 2. List print below:
1233       " << std::endl;
1234     sl2->print();
1235     success = false;
1236 }
1237 if ( sl2->count(5) != 2 )
1238 {
1239     std::cout << "Error in SkipList count case 3. List print below:
1240       " << std::endl;
1241     sl2->print();
1242     success = false;
1243 }
Associative containers with strong guarantees

```cpp
#testing equal_range with sl2*/
std::pair<slMultipleIterator, slMultipleIterator> pairSlMult1 = sl2->equal_range(1);
if ( (pairSlMult1.first).position->get_key() != 4 |
    (pairSlMult1.second).position->get_key() != 4 )
{
    std::cout << "Error in SkipList equal_range case 1. List print below:
    " << std::endl;
    sl2->print();
success = false;
}
pairSlMult1 = sl2->equal_range(4);
if ( (pairSlMult1.first).position->get_key() != 4 |
    (pairSlMult1.second).position->get_key() != 5 )
{
    std::cout << "Error in SkipList equal_range case 2. List print below:
    " << std::endl;
    sl2->print();
success = false;
}
pairSlMult1 = sl2->equal_range(5);
if ( (pairSlMult1.first).position->get_key() != 5 |
    (pairSlMult1.second).position->get_key() != 42 )
{
    std::cout << "Error in SkipList equal_range case 3. List print below:
    " << std::endl;
    sl2->print();
success = false;
}
/* Testing size ()*/
if (sl2->size() != 3)
{
    std::cout << "Error in SkipList size case 1. List print below:
    " << std::endl;
    sl2->print();
success = false;
}
if (sl3->size() != 0)
{
    std::cout << "Error in SkipList size case 2. List print below:
    " << std::endl;
    sl3->print();
success = false;
}
if (sl4->size() != 2)
{
    std::cout << "Error in SkipList size case 3. List print below:
    " << std::endl;
    sl4->print();
success = false;
}
/* Testing empty*/
if (sl2->empty())
{
```
```
```cpp
Morten Poulsen and Lars Skovlund

1303 std::cout << "Error in SkipList empty case 1. List print below:");
1304 << std::endl;
1305 sl2->print();
1306 success = false;
1307 }
1308 if( !(sl3->empty()) )
1309 {
1310 std::cout << "Error in SkipList empty case 2. List print below:");
1311 << std::endl;
1312 sl3->print();
1313 success = false;
1314 }
1315 return success;
1316 }
1317 }
1318 template <typename Key, typename Satellite, typename Comparator, typename unique>
1319 bool BaseSkipList<Key, Satellite, Comparator, unique>::HelpUnitTest(Node<Key, Satellite, Comparator> * node, Key expectedKey,
1320 int expectedHeight, Key expectedLeft, Key expectedRight0,
1321 Key expectedRight1, Key expectedRight2)
1322 {
1323 bool success = true;
1324 if( node->right.size.logical_height() != expectedHeight )
1325 {
1326 BREAKPOINT;
1327 success = false;
1328 }
1329 if( expectedKey != NULL && *(node->get_key()) != expectedKey )
1330 {
1331 BREAKPOINT;
1332 success = false;
1333 }
1334 if( expectedLeft != NULL && *(node->left->get_key()) != expectedLeft )
1335 {
1336 BREAKPOINT;
1337 success = false;
1338 }
1339 if( expectedRight0 != NULL && *(node->right[0]->get_key()) != expectedRight0 )
1340 {
1341 BREAKPOINT;
1342 success = false;
1343 }
1344 if( expectedRight1 != NULL && *(node->right[1]->get_key()) != expectedRight1 )
1345 {
1346 BREAKPOINT;
1347 success = false;
1348 }
1349 if( expectedRight2 != NULL && *(node->right[2]->get_key()) != expectedRight2 )
1350 {
1351 BREAKPOINT;
1352 success = false;
1353 }
1354 if( !success && node != head )
```
Associative containers with strong guarantees

```cpp
std::cout << "Key:" << *(node->get_key()) << "\n";
return success;
}

AppendixB.2 benchmark.cpp

/*********************/
/* BENCHMARK.CPP */
/* by Lars Skovlund and Morten Poulsen */
/* this file makes a simple benchmark test of skip lists */
/* *********************/
#include "skiplist.h"
#include <ctime>

typedef SkipList<int, void, comparator<int>>, NonUniqueTraits<int, void, comparator<int>> slNonUnique; // allow duplicate keys
// takes a skiplist and a number and performs n insertions
void benchInsert(slNonUnique* sl, const int N)
{
    for(int i = 0; i < N; ++i)
    { sl->insert((int)5*N*rand()); }
}

void benchFind(slNonUnique* sl, const int N)
{
    for(int i = 0; i < N; ++i)
    { sl->find((int)5*N*rand()); }
}

void benchErase(slNonUnique* sl, const int N)
{
    for(int i = 0; i < N; ++i)
    { sl->erase((int)5*N*rand()); }
}

int main()
{
    const int NOOFELEM = 8;
    const int RUNS = 5;
    int noOfElements[NOOFELEM] = 
    { 25000, 50000, 75000, 100000, 250000, 500000, 750000, 1000000};
    double insertTotals[NOOFELEM];
    memset(insertTotals, 0, sizeof(insertTotals));
    double findTotals[NOOFELEM];
    memset(findTotals, 0, sizeof(findTotals));
```
double eraseTotals[NOOFELEM];
memset(eraseTotals, 0, sizeof(eraseTotals));
std::cout << endl;
std::cout << "Benchmarking" << std::endl;
for( int j = 0; j < RUNS; j++ )
{
    slNonUnique* sl = new slNonUnique();

clock_t primal_start = clock();
benchInsert(sl, noOfElements[i]);

clock_t primal_ticks = clock() - primal_start;
insertTotals[i] =
    insertTotals[i] + double(primal_ticks) / double(CLOCKS_PER_SEC);

primal_start = clock();
benchFind(sl, noOfElements[i]);
primal_ticks = clock() - primal_start;
findTotals[i] =
    findTotals[i] + double(primal_ticks) / double(CLOCKS_PER_SEC);

primal_start = clock();
benchErase(sl, noOfElements[i]);
primal_ticks = clock() - primal_start;
eraseTotals[i] =
    eraseTotals[i] + double(primal_ticks) / double(CLOCKS_PER_SEC);
}

delete sl;

std::cout << ".";
}
std::cout << std::endl;

// writing to screen in format usable by gnuplot and readable by people
std::cout << 
#insert
\n#elems \n#t\n(elem)\n#t/elem\n(elem)

for( int i = 0; i < NOOFELEM; ++i )
{
    std::cout << noOfElements[i] << " #insert\n\n#elems\n" << insertTotals[i] / double(RUNS)
<< " #t\n(elem)\n#t/elem\n(elem)" << std::endl;
}

for( int i = 0; i < NOOFELEM; ++i )
{
    std::cout << noOfElements[i] << " #find\n\n#elems\n" << findTotals[i] / double(RUNS)
<< " #t\n(elem)\n#t/elem\n(elem)" << std::endl;
}

for( int i = 0; i < NOOFELEM; ++i )
{
    std::cout << noOfElements[i] << " #erase\n\n#elems\n" << eraseTotals[i] / double(RUNS)
<< " #t\n(elem)\n#t/elem\n(elem)" << std::endl;
}
return 0;

Appendix B.3 Output from benchmark

The output is generated by changing the typedef in benchmark.cpp so it uses first Unique traits and then NonUnique traits.

C:\Documents and Settings\mop\Desktop\GenericProgramming\miniproject2\DeterministicSkipList\release>DeterministicSkipList.exe

Benchmarking

#insert
#elems t(s) t/elem (ns)
25000 0.0748 2992
50000 0.1562 3124
75000 0.2564 3418.67
100000 0.3438 3438
125000 0.4352 3588.8
150000 0.5266 3700.4
175000 0.6180 3750.13
200000 0.7094 3784.4

#find
#elems t(s) t/elem (ns)
25000 0.0408 1632
50000 0.0908 1816
75000 0.1434 1912
100000 0.1938 1938
125000 0.2452 2048.8
150000 0.2966 2093.2
175000 0.3480 2116.53
200000 0.3984 2156.2

#erase
#elems t(s) t/elem (ns)
25000 0.0686 2744
50000 0.1344 2688
75000 0.1784 2378.67
100000 0.2156 2156
125000 0.2528 2156
150000 0.2892 1237.6
175000 0.3256 750.4
200000 0.3620 595.73

C:\Documents and Settings\mop\Desktop\GenericProgramming\miniproject2\DeterministicSkipList\release>DeterministicSkipList.exe

Benchmarking

#insert
#elems t(s) t/elem (ns)
25000 0.0842 3368
47 50000 0.2062 4124
48 75000 0.3344 458.67
49 100000 0.4784 4784
50 250000 1.3656 554.24
51 500000 2.9688 5937.6
52 750000 4.6688 6225.07
53 1000000 6.5126 6512.6
54
55
56 25000 0.0534 2136
57 50000 0.1280 2560
58 75000 0.2160 2880
59 100000 0.3062 3062
60 250000 0.8940 3576
61 500000 1.8936 3787.2
62 750000 2.9374 3916.53
63 1000000 4.1374 4137.4
64
65 25000 0.0936 3744
66 50000 0.2284 4568
67 75000 0.3624 4832
68 100000 0.4874 4874
69 250000 1.0310 4124
70 500000 1.6844 3368.8
71 750000 2.3188 3091.73
72 1000000 2.9376 2937.6

References


