Skip List Documentation

Release 0.3.13

Paul Ross

Apr 01, 2025

CONTENTS

CHAPTER

ONE

INTRODUCTION

A SkipList behaves as a sorted list with, typically, O(log(n)) cost for insertion, look-up and removal. This makes it ideal for such operations as computing the rolling median of a large dataset.

A SkipList is implemented as a singly linked list of ordered nodes where each node participates in a subset of, sparser, linked lists. These additional 'sparse' linked lists provide rapid indexing and mutation of the underlying linked list. It is a probabilistic data structure using a random function to determine how many 'sparse' linked lists any particular node participates in. As such SkipList is an alternative to binary tree, Wikipedia has an introductory page on SkipLists .

An advantage claimed for SkipLists are that the insert and remove logic is simpler (however I do not subscribe to this). The drawbacks of a SkipList include its larger space requirements and its O(log(N)) lookup behaviour compared to other, more restricted and specialised, data structures that may have either faster runtime behaviour or lower space requirements or both.

This project contains a SkipList implementation in C++, with Python bindings, which has these characteristics:

- No capacity restrictions apart from available memory.
- Works with any C++ type <T> that has meaningful comparison operators.
- The C++ SkipList can be compiled as thread safe.
- The Python SkipList is thread safe.
- The SkipList has exhaustive internal integrity checks.
- Python SkipLists can be long/float/bytes/object types, the latter can have user defined comparison functions.
- With Python 3.8+ SkipLists can be combined with the multiprocessing.shared_memory module for concurrent operation on large arrays. For example see *Rolling Median in Python with multiprocessing.shared_memory* which speeds up a rolling median near linearly with the number of cores.
- This implementation is extensively performance tested in C++ and Python, see *Skip List Performance*.

There are a some novel features to this implementation:

- A SkipList is a probabilistic data structure but we have deterministic tests that work for any (sane) random number generator, see: *Testing a Probabilistic Structure*
- This SkipList can dynamically generate visualisations of its current internal state, see: *Visualising a Skip List*

CHAPTER TWO

PROJECT

This project is hosted in a number of places:

- Source (the source of truth): GitHub
- Builds on PyPi
- Documentation: Read The Docs

CHAPTER THREE

CREDITS

Originally written by Paul Ross with credits to: Wilfred Hughes (AHL), Luke Sewell (AHL) and Terry Tsantagoeds (AHL).

CHAPTER

FOUR

INSTALLATION

4.1 C++

This SkipList requires:

- A C++17 compiler.
- `-I<skiplist>/src/cpp` as an include path.
- `<skiplist>/src/cpp/SkipList.cpp` to be compiled/linked.
- The macro `SKIPLIST_THREAD_SUPPORT` set if you want a thread safe SkipList

4.2 Python

This SkipList version supports Python 3.7, 3.8, 3.9, 3.10, 3.11 (and, probably, some earlier Python 3 versions) [older versions, now abandoned, supported Python 2.7].

4.2.1 From PyPi

The project is on PyPi

```
$ pip install orderedstructs
```

4.2.2 Development

Assuming you are in a virtual environment:

```
$ cd <skiplist>
$ pip install -r requirements.txt
$ python setup.py develop
```

If you are on a platform that uses GCC you will need to set the environment variable `USING_GCC` before invoking `setup.py` (this just fixes some warnings) thus:

```
$ USING_GCC=1 python setup.py develop
```

CHAPTER

FIVE

EXAMPLES

Here are some examples of using a SkipList in your code:

5.1 C++

```cpp
#include "SkipList.h"

// Declare with any type that has sane comparison.
OrderedStructs::SkipList::HeadNode<double> sl;

sl.insert(42.0);
sl.insert(21.0);
sl.insert(84.0);

sl.has(42.0) // true
sl.size()    // 3
sl.at(1)     // 42.0, throws OrderedStructs::SkipList::IndexError if index out of␣
↪range

sl.remove(21.0); // throws OrderedStructs::SkipList::ValueError if value not present

sl.size()    // 2
sl.at(1)     // 84.0
```

The C++ SkipList is thread safe when compiled with the macro SKIPLIST_THREAD_SUPPORT, then a SkipList can then be shared across threads, such as:

```cpp
#include <thread>
#include <vector>

#include "SkipList.h"

void do_something(OrderedStructs::SkipList::HeadNode<double> *pSkipList) {
    // Insert/remove items into *pSkipList
    // Read items inserted by other threads.
}

OrderedStructs::SkipList::HeadNode<double> sl;
std::vector<std::thread> threads;

for (size_t i = 0; i < thread_count; ++i) {
    threads.push_back(std::thread(do_something, &sl));
}
for (auto &t: threads) {
```

(continues on next page)

(continued from previous page)

```
    t.join();
}
// The SkipList now contains the totality of the thread actions.
```

See src/cpp/test/test_concurrent.cpp for more examples.

5.2 Python

An example of using a SkipList of always ordered floats:

```python
import orderedstructs

# Declare with a type. Supported types are long/float/bytes/object.
sl = orderedstructs.SkipList(float)

sl.has(42.0) # True
sl.size()    # 3
sl.at(1)     # 42.0, raises IndexError if index out of range

sl.remove(21.0) # raises ValueError if value not present

sl.size()    # 2
sl.at(1)     # 84.0
```

The Python SkipList can be used with user defined objects with a user defined sort order. In this example the last name of the person takes precedence over the first name:

```python
import functools

import orderedstructs

@functools.total_ordering
class Person:
    """Simple example of ordering based on last name/first name."""
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def __eq__(self, other):
        try:
            return self.last_name == other.last_name and self.first_name == other.
↪first_name
        except AttributeError:
            return NotImplemented

    def __lt__(self, other):
        try:
            return self.last_name < other.last_name or self.first_name < other.first_
↪name
        except AttributeError:
            return NotImplemented

    def __str__(self):
        return '{}, {}'.format(self.last_name, self.first_name)

```

(continues on next page)

(continued from previous page)

```python
sl = orderedstructs.SkipList(object)

sl.insert(Person('Peter', 'Pan'))
sl.insert(Person('Alan', 'Pan'))
assert sl.size() == 2
assert str(sl.at(0)) == 'Pan, Alan'
assert str(sl.at(1)) == 'Pan, Peter'
```

CHAPTER

SIX

TESTING

This SkipList has extensive tests for correctness and performance.

6.1 C++

6.1.1 Using a Makefile

To run all the C++ functional and performance tests:

```
$ cd <skiplist>/src/cpp
$ make release
$ ./SkipList_R.exe
```

To run the C++ functional tests with agressive internal integrity checks but excluding the performance checks:

```
$ cd <skiplist>/src/cpp
$ make debug
$ ./SkipList_D.exe
```

To run all the C++ functional and performance tests for a thread safe SkipList:

```
$ cd <skiplist>/src/cpp
$ make release CXXFLAGS=-DSKIPLIST_THREAD_SUPPORT
$ ./SkipList_R.exe
```

6.1.2 Using CMake

To build the binary:

```
$ cd <skiplist>
$ cmake --build cmake-build-release --target SkipList -- -j 6
```

And to run the tests:

```
$ cd <skiplist>
$ ./cmake-build-release/SkipList
```

This takes 60s or so.

6.2 Python

Testing requires `pytest` and `hypothesis`:

To run basic tests:

```
$ cd <skiplist>
$ pytest tests/
```

This takes typically 30s.

To run more extensive (slow) tests and generate benchmarks:

```
$ cd <skiplist>
$ pytest tests/ --runslow --benchmark-sort=name --benchmark-autosave --benchmark-
↪histogram
```

6.3 Both C++ and Python

Running all tests (C++ and Python) and building all currently supported Python versions is done with a build script:

```
$ cd <skiplist>
$ ./build_all.sh
```

Note: Naming conventions

The project on PyPi is called `orderedstructs` as it was originally intended as a repository for multiple ordered structures. Because of changing requirements the sole ordered structure in this project currently is a Skip List. Thus `SkipList` and `orderedstructs` are used (sort of) interchangeably.

CHAPTER

SEVEN

DESIGN AND IMPLEMENTATION OF THE C++ SKIP LIST

7.1 Design of the C++ Skip List

A skip list is a a singly linked list with additional, coarser, linked lists. These additional lists allow rapid location, insertion and removal of nodes. Values in a skip list are maintained in order at all times.

Skip lists are alternatives to balanced trees for operations such as a rolling median. The disadvantages of skip lists are:

- Less space efficient than balanced trees, see: *Space Complexity*.
- Performance is similar to balanced trees except finding the mid-point which is O(log(N)) for a skip list compared with O(1) for a balanced tree.

The advantages claimed for skip lists are:

- The insert() and remove() logic is simpler (I do not subscribe to this).

This skip list design has the coarser lists implemented as optional additional links between the nodes themselves. The illustration below shows a well formed skip list with a head node (`HED`) linked to the ordered nodes `A` to `H`:

```
| 5 E |------------------------------------->| 4 0 |---------------------------->|␣
↪NULL |
| 1 A |->| 2 C |---------->| 2 E |---------->| 2 G |---------->| 2 0 |---------->|␣
↪NULL |
| 1 A |->| 1 B |->| 1 C |->| 1 D |->| 1 E |->| 1 F |->| 1 G |->| 1 H |->| 1 0 |->|␣
↪NULL |
| HED |  |  A  |  |  B  |  |  C  |  |  D  |  |  E  |  |  F  |  |  G  |  |  H  |
```

Each node has a stack of values that consist of a 'width' and a reference to another node (or NULL). At the lowest level is a singly linked list and all widths are 1. They are shown in the diagram as a number for the width and the node name for the reference (0 for NULL). The 'widths' at each node/level specify how many level 0 nodes the node reference skips over. These widths are used to rapidly index into the skip list starting from the highest level and working down.

At level 1 the links are (ideally) to every other node, at level 2 the links are (ideally) to every fourth node and at level 3 the links are (ideally) to every eighth node and so on.

7.1.1 Search Algorithms

Presence

This search returns true/false as to whether a value is present. The search is recursive starting at the coarsest list dropping down a level when the search overshoots.

Take, for example, searching for the presence of `D`.

- The search starts as `HED[2]` this leads to `E[2]` and `E > D` so that search is rejected.
- The search then evolves to: `HED[2] -> HED[1]` which leads to `A[1] -> C[1] -> E[1]`, again that search is rejected as `E > D`.
- The search finally succeeds following the path `HED[2] -> HED[1] -> A[1] -> C[1] -> C[0] -> D[0]` so returns true.

It is not necessary for the search to reach level 0 if it can be terminated earlier. For example the search for `C` would be achieved by `HED[2] -> HED[1] -> A[1] -> C[1]`.

Index

Searching for a particular index value is accomplished in a similar way using the widths to calculate the position in the list. For example to find the 6th node (`F`):

- The search starts at `HED[2]` and moves to `E[2]`. Any further move to the right is not possible because 5 + 4 > 6.
- Dropping down to `E[1]` no further progress can be made to the right as 5 + 2 > 6.
- Dropping down to `E[0]` finds node `F` as 5 + 1 == 6.

It is not necessary for the search to reach level 0 if it can be terminated earlier. For example the search for the fifth member would be achieved by `HED[2] -> E[2]`.

7.1.2 Insertion Algorithm

To understand how the skip list is maintained, consider insertion. Before inserting node `E` the skip list would look like this:

```
| 1 A |->| 2 C |---------->| 3 G |------------------->| 2 0 |---------->| NULL |
| 1 A |->| 1 B |->| 1 C |->| 1 D |->| 1 F |->| 1 G |->| 1 H |->| 1 0 |->| NULL |
| HED |  |  A  |  |  B  |  |  C  |  |  D  |  |  F  |  |  G  |  |  H  |
```

Inserting `E` means:

- Finding where `E` should be inserted (after `D`).
- Creating node `E` with a random height (tossing a coin gives heads/heads/tails so 3 high).
- Updating `D` to refer to `E` at level 0.
- Updating `C` to refer to `E` at level 1 and decreasing `C`'s width to 2, increasing `E` width at level 1 to 2.
- Expanding `HED` to level 2 with a reference to `E` and a width of 5.
- Updating `E` with a reference to NULL and a width of 4.

After inserting node E the skip list looks like this:

```
| 5 E |------------------------------------->| 4 0 |----------------------------->|␣
↪NULL |
| 1 A |->| 2 C |---------->| 2 E |---------->| 2 G |---------->| 2 0 |---------->|␣
↪NULL |
```

(continues on next page)

(continued from previous page)

```
| 1 A |->| 1 B |->| 1 C |->| 1 D |->| 1 E |->| 1 F |->| 1 G |->| 1 H |->| 1 0 |->|␣
↪NULL |
| HED |   |  A  |   |  B  |   |  C  |   |  D  |   |  E  |   |  F  |   |  G  |   |  H  |
```

Insertion Position

The first two operations are done by a recursive search. This creates the call chain that passes through these nodes:

`HED[1] -> A[1] -> C[1] -> C[0] -> D[0]`.

Thus `E` will be created at level 0 and inserted after `D`. All node creation is done from level 0 so that when the recursive search unwinds only the appropriate nodes and levels will be updated.

Node Creation

Node `E` is created with a stack containing a single pointer to the next node `F`. Then a virtual coin is tossed, for each 'head' a pair of values `{this, 1}` is added to this stack. These values will be swapped with other nodes when the recursive calls unwind. If a 'tail' is thrown the stack is complete. In the example above when creating `E` we have encountered tosses of 'head', 'head', 'tail'.

Recursive Unwinding

The remaining operations are done as recursion unwinds. `D[0]` and `C[0]` update `E[1]` with their cumulative width (2). `C[1]` adds 1 to width (a new node is inserted) then subtracts `E[1]`. Then `C[1]` / `E[1]` are swapped so that the pointers and widths are correct. And so on until `HED` is reached, in this case a new level is added and `HED[2]` swapped with `E[2]`.

There are some visualisations of this behaviour here: *Visualising a Skip List*

7.1.3 Removal Algorithm

This is essentially the insertion algorithm in reverse. Before removing node E the skip list looks like this:

```
| 5 E |------------------------------------->| 4 0 |---------------------------->|␣
↪NULL |
| 1 A |->| 2 C |---------->| 2 E |---------->| 2 G |---------->| 2 0 |---------->|␣
↪NULL |
| 1 A |->| 1 B |->| 1 C |->| 1 D |->| 1 E |->| 1 F |->| 1 G |->| 1 H |->| 1 0 |->|␣
↪NULL |
| HED |   |  A  |   |  B  |   |  C  |   |  D  |   |  E  |   |  F  |   |  G  |   |  H  |
```

After removing `E` the skip list looks like this:

```
| 1 A |->| 2 C |---------->| 3 G |------------------->| 2 0 |---------->| NULL |
| 1 A |->| 1 B |->| 1 C |->| 1 D |->| 1 F |->| 1 G |->| 1 H |->| 1 0 |->| NULL |
| HED |   |  A  |   |  B  |   |  C  |   |  D  |   |  F  |   |  G  |   |  H  |
```

To remove `E` a recursive search for `E[0]` is made. Starting at the top (level = 2) of the head node `HED[2]` then `E[2]` is found immediately however the level is not 0 so the search is discarded. A search through `HED[2] -> HED[1]` follows the path `HED[2] -> HED[1] -> A[1] -> C[1]` to `E[1]`, again this search is discarded as the level is not 0.

The final search succeeds by setting up the call chain `HED[2] -> HED[1] -> A[1] -> C[1] -> C[0] -> D[0] -> E[0]`.

As the recursion unwinds node that data at each level of `E` is used to swap node pointers and adjust the widths of node C. Finally `HED` can drop one level before deleting `E`.

A more detailed description of the algorithm used for each method is in the *C++ API*

7.2 Implementation Notes

This skip list is implemented in C++ with templates so works for any type <T> that can be compared deterministically.

Node membership of the additional lists is done in a probabilistic manner. This is achieved at node creation time by tossing a virtual coin. These lists are not explicit, they are implied by the references between Nodes at a particular level.

'Level' is an `size_t` type that specifies the linked list, level 0 is the linked list to every node. The list at level 1 links (ideally) to every other node. The list at level 2 links (ideally) to every fourth node and so on. In general the list at level n links (ideally) to every 2**n node.

7.2.1 Memory Management

Memory management is pretty simple in a skip list. Essentially it is no more complicated than a singly linked list as every node is created and deleted at level 0. The duplicate pointers at higher levels can be ignored.

7.2.2 Thread Safety

The C++ SkipList can be compiled with the macro `SKIPLIST_THREAD_SUPPORT` set and this will introduce a mutex on the `HeadNode` that makes the SkipList thread safe. There are illustrations of the use of this in *skiplist/src/cpp/test/test_concurrent.cpp*

The Python SkipList is thread safe. There are tests in *skiplist/tests/unit/test_SkipList_PyObject_threaded.py* that illustrate this.

7.3 Code Layout

There are three template classes defined in their own .h files and these are all included into the *SkipList.h* file.

The classes are:

Class	Description
`Swappab`	A simple bookkeeping class that has a vector of `[{skip_width, Node<T>*}, ...]` This vector can be expanded or contracted at will. Both HeadNode and Node classes have one of these to manage their references. This is defined in *NodeRefs.h*
`Node<T>`	This represents a single value in the skip list. The height of a Node is determined at construction by tossing a virtual coin, this determines how many coarser lists this node participates in. A Node has a `SwappableNodeRefStack` object and a value of type `T`. This is defined in *Node.h*
`HeadNod`	There is one of these per skip list and this provides the API to the entire skip list. The height of the HeadNode expands and contracts as required when Nodes are inserted or removed (it is the height of the highest Node). A HeadNode has a `SwappableNodeRefStack` object and an independently maintained count of the number of Node objects in the skip list. This is defined in *HeadNode.h*

A `Node` and `HeadNode` have specialised methods such as `has()`, `at()`, `insert()`, `remove()` that traverse the skip list recursively. These are described in the *C++ API*

7.3.1 Other Files of Significance

SkipList.cpp exposes the random number generator (`rand()`) and seeder (`srand()`) so that they can be accessed by CPython and others for deterministic testing.

IntegrityEnums.h has definitions of error codes that can be created by the skip list integrity checking functions.

cSkipList.h, *cSkipList.cpp* contains a CPython module with a SkipList implementation for a number of builtin Python types. This has the usual *setup.py* and *setup.cfg* files.

7.4 Code Idioms

7.4.1 Prevent Copying

Copying operations are (mostly) prohibited for performance reasons. The only class that allows copying is `struct NodeRef` that contains fundamental types (int and pointer). All other classes declare their copying operation private and unimplemented (rather than using C++11 delete) for compatibility with older compilers.

7.4.2 Reverse Loop of `unsigned int`

In a lot of the code we have to count down from some value to 0 with a `size_t` (an `unsigned int` type), the idiom used is this:

```
for (size_t l = height(); l-- > 0;) {
```

The `l-- > 0` will test `l` against 0 then decrement it. `l` will thus start at the value `height() - 1` down to 0 then exit the loop.

Note: If `l` is declared before the loop it will have the maximum value of a size_t on exit from this loop unless a break statement is encountered.

7.5 Worst Case Behaviour

It is possible that the skip list can get mal-formed with an unfortunate set of coin tosses (or a knowledgeable and adversarial user). In which case the search time complexity can become O(n) rather than O(log(n)). Here is a well-formed skip list:

```
| 5 E |-------------------------------------->| 4 0 |----------------------------->|␣
↪NULL |
| 1 A |->| 2 C |---------->| 2 E |---------->| 2 G |---------->| 2 0 |---------->|␣
↪NULL |
| 1 A |->| 1 B |->| 1 C |->| 1 D |->| 1 E |->| 1 F |->| 1 G |->| 1 H |->| 1 0 |->|␣
↪NULL |
| HED |  |  A  |  |  B  |  |  C  |  |  D  |  |  E  |  |  F  |  |  G  |  |  H  |
```

The same frequency of coin tosses but in worst case order might produce this:

```
| 5 E |--| 8 0 |------------------------------------------------------------------>|␣
↪NULL |
| 1 A |->| 1 B |--| 1 C |->| 1 D |->| 5 0 |-------------------------------------->|␣
↪NULL |
| 1 A |->| 1 B |->| 1 C |->| 1 D |->| 1 E |->| 1 F |->| 1 G |->| 1 H |->| 1 0 |->|␣
```

(continues on next page)

(continued from previous page)

```
↪NULL |
| HED |   | A  |   | B  |   | C  |   | D  |   | E  |   | F  |   | G  |   | H  |
```

Any search on the latter skip list rapidly degrades into a simple linear search.

There is no particular protection against this in this implementation of a skip list.

7.6 Roads not Travelled

Certain design and implementations were not explored, here they are and why.

7.6.1 Key/Value Implementation

Skip lists are commonly used for key/value dictionaries. Given things like `std::map` or `std::unorderedmap` etc. in C++ I see no reason why a SkipList should be used as an alternative.

7.6.2 Adversarial Users

If the user knows the behaviour of the random number generator it is possible that they can change the order of insertion to create a poor distribution of nodes which will make operations tend to O(N) rather than O(log(N)).

7.6.3 Optimisation: Re-index Nodes on Complete Traversal

To correct the worst case behaviour above it would be possible to re-index the skip list from time to time, for example when a full traversal is made. This is not included in this implementation.

7.6.4 Optimisation: Reuse removed nodes

If node construction is expensive then a `remove()` + `insert()` operation could re-use the removed node rather than deleting and creating a new one. This is not included in this implementation.

7.6.5 Reference Counting

Some time (and particularly space) improvement could be obtained by reference counting nodes so that duplicate values could be eliminated at some additional code/space cost. Since the primary use case for this skip list is for computing the rolling median of doubles the chances of duplicates are slim and no advantage would be gained. For int, long and string there is a higher probability of sameness so reference counting might be implemented in the future if these types become commonly used.

7.6.6 Use an Array of `{skip_width, Node<T>*}` rather than a vector

Less space would be used for each Node if the `SwappableNodeRefStack` used a dynamically allocated array of `[{skip_width, Node<T>*}, ...]` rather than a vector.

CHAPTER

EIGHT

SKIP LIST PERFORMANCE

8.1 C++ Performance Tests

The time performance tests are run as follows:

```
$ cd src/cpp
$ make release
$ ./SkipList_R.exe
                 test_very_simple_insert(): PASS
                                                    ...
                 test_roll_med_even_mean(): PASS
               perf_single_insert_remove(): 451.554 (ms) rate x.xe+06 /s
                                                    ...
            perf_roll_med_odd_index_wins(): vectors length:  1000000 window width: 524288 time: x.x (ms)
                            perf_size_of(): size_of(       1):      216 bytes ratio:     216 /sizeof(T):       x
                                                    ...
Final result: PASS
Exec time: x.x (s)
Bye, bye!
```

The ouptut is a combination of test results, performance results and dot visualisations

If multi-threaded support is enabled (see *Multi-threaded C++ Performance* below) the execution time including the multi-threaded tests takes about three minutes. With single threaded support the tests take around two minutes. If the debug version is built the performance tests are omitted as the cost of integrity checking at every step is very high which would make the performance test data irrelevant.

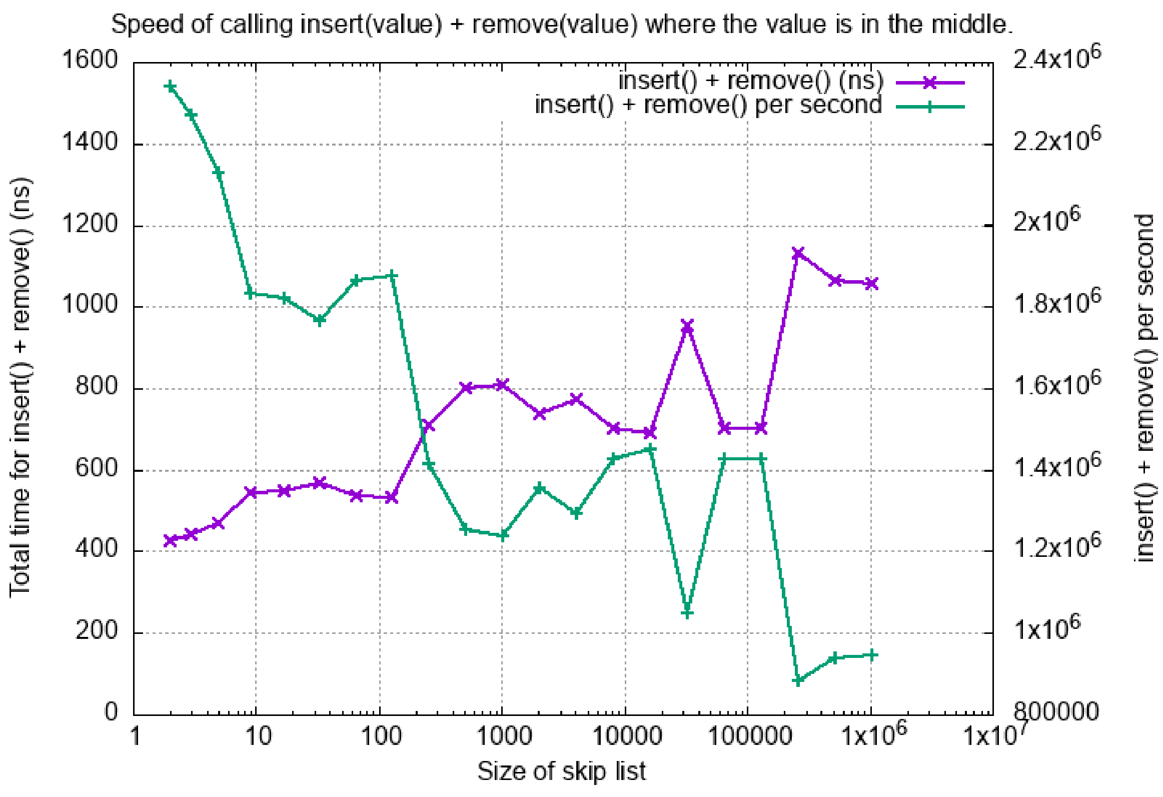
8.2 Time Performance

The performance test mostly work on a SkipList of type `double` that has 1 million values. Test on a couple of modern 64 bit OS's [Linux, Mac OS X] show that the cost of SkipList operations is typically as follows.

8.2.1 Mutating operations: `insert()`, `remove()`

These operations depend on the size of the SkipList. For one containing 1 million doubles each operation is typically 450 ns (2.2 million operations per second).

Here is a graph showing the cost of the *combined* `insert()` plus `remove()` of a value in the middle of the list, both as time in (ns) and rate per second. The test function is `perf_single_ins_rem_middle_vary_length()`.



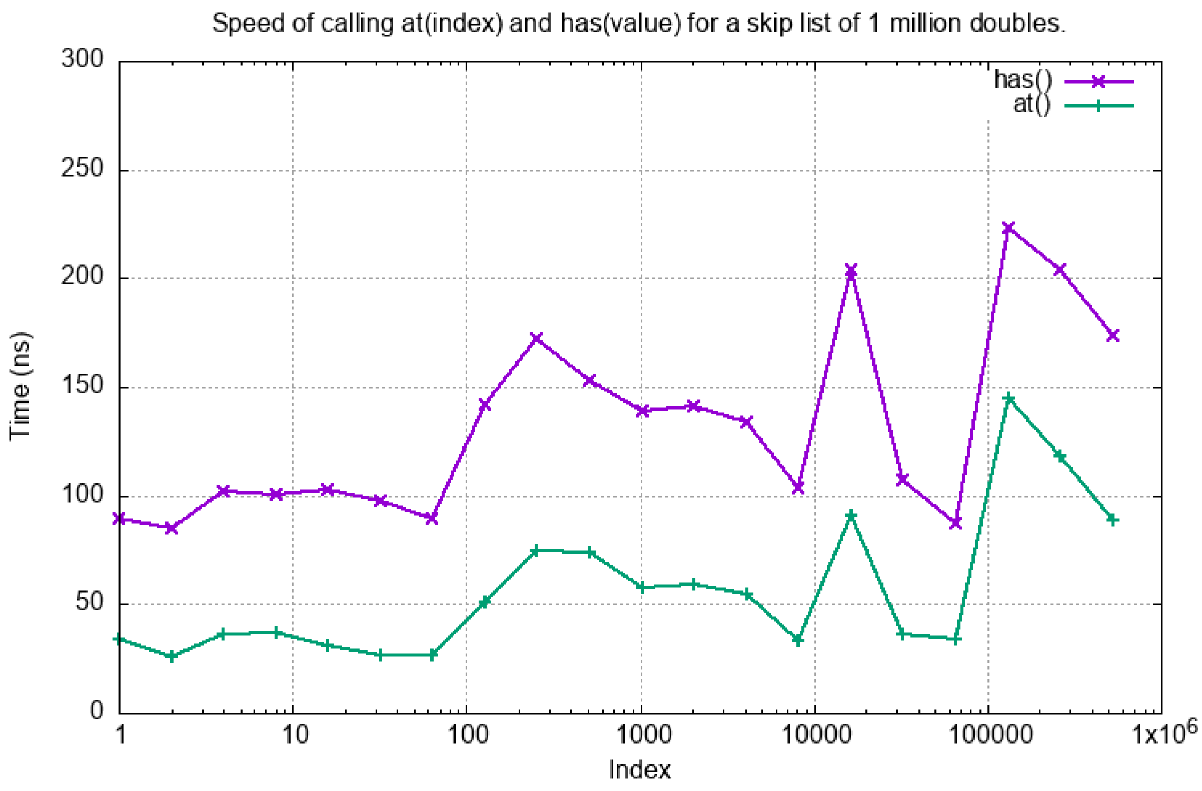
This shows good O(log(n)) behaviour where n is the SkipList size.

8.2.2 Indexing operations: `at()`, `has()` `index()`

These operations on a SkipList containing 1 million doubles is typically 220 ns (4.6 million operations per second).

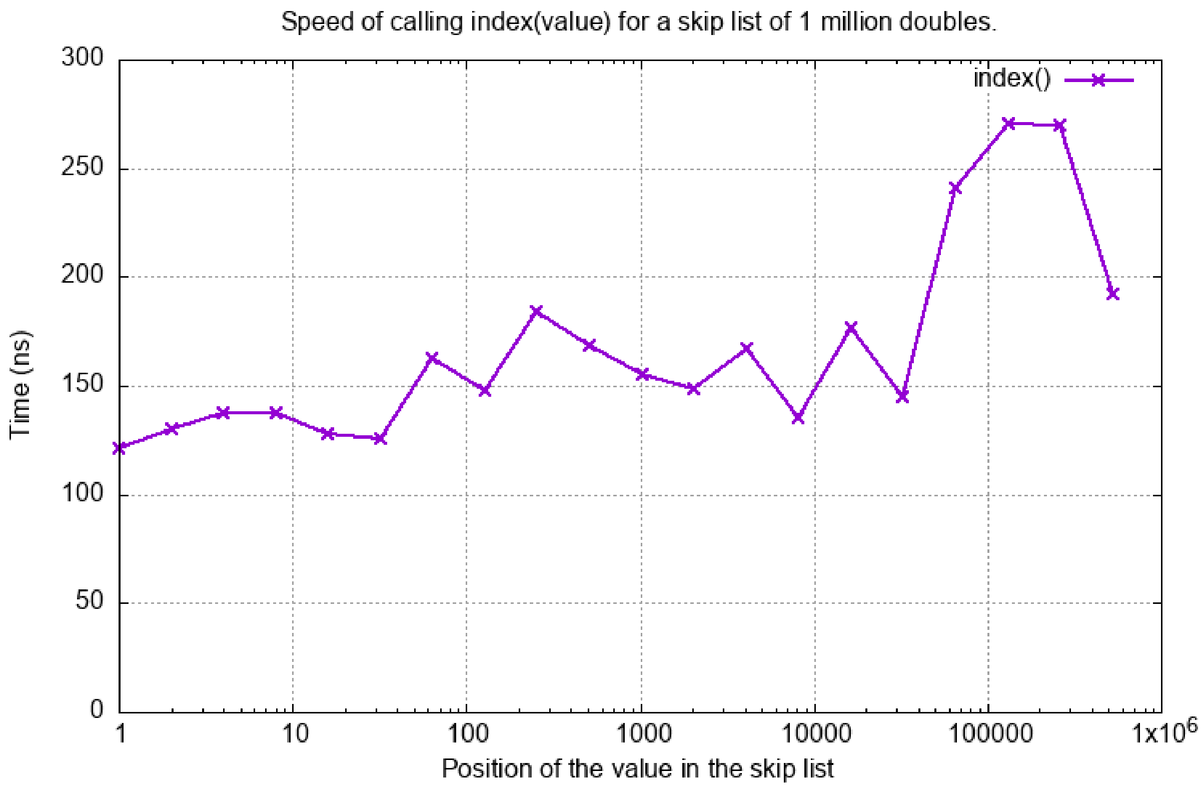
vs Location

Here is plot of the time taken to execute `at()` or `has()` on a SkipList of 1 million doubles where the X-axis is the position in the SkipList of the found double. The test functions are respectively `perf_at_in_one_million()` and `perf_has_in_one_million()`.



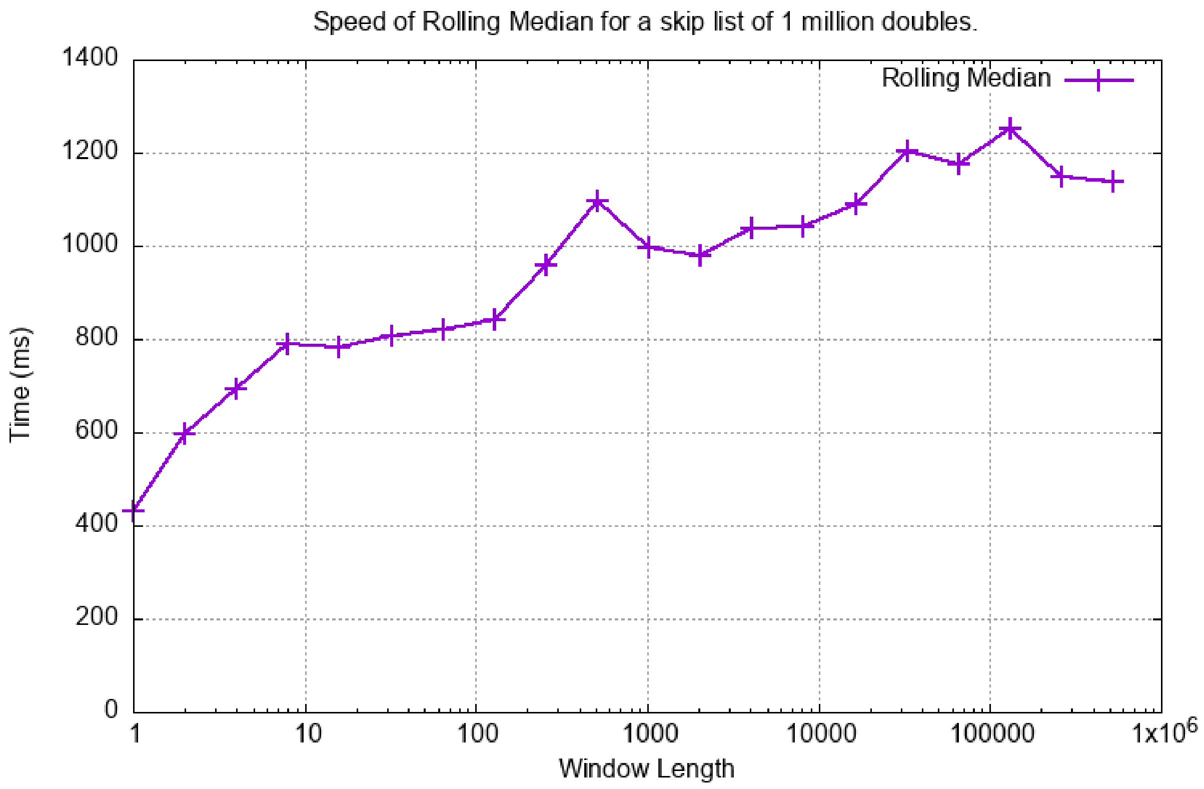
This shows fairly decent O(log(n))'ish type behaviour.

The `index(value)` method has similar behaviour:



Rolling Median

Here is a plot of the time taken to compute a rolling median on one million values using different window sizes. The number of results is 1e6 - window size. This needs to `insert(new_value)` then `at(middle)` then `remove(old_value)`. A window size of 1000 and 1m values (the size of the SkipList) takes around 1 second or 1000 ns /value.



The test function is `perf_roll_med_odd_index_wins()`.

8.3 Space Complexity

Space usage is a weakness of SkipLists. There is a large amount of bookkeeping involved with multiple node pointers plus the width values for each node for an indexed SkipList.

8.3.1 Theoretical Memory Usage for `double`

The space requirements for a SkipList of doubles can be estimated as follows.

`t = sizeof(T)` ~ typ. 8 bytes for a double. `v = sizeof(std::vector<struct NodeRef<T>>)` ~ typ. 32 bytes. `p = sizeof(Node<T>*)` ~ typ. 8 bytes. `e = sizeof(struct NodeRef<T>)` ~ typ. 8 + p = 16 bytes. Then each node: is `t + v` bytes.

Linked list at level 0 is `e` bytes per node and at level 1 is, typically, `e / 2` bytes per node (given `p()` as a fair coin) and so on. So the totality of linked lists is about `2 * e` bytes per node.

Then the total is `N (t + v + 2 e)` which for `T` as a double is typically 72 bytes per item.

Memory usage can be gauged by any of the following methods:

- Theoretical calculation such as above which gives ~72 bytes per node for doubles.
- Observing a process that creates a SkipList using OS tools, this typically gives ~86 bytes per node for doubles.

- Calling the `size_of()` method that can make use of its knowledge of the internal structure of a SkipList to estimate memory usage. For `double` this is shown to be about 76 bytes per node. Any `size_of()` estimate will be an underestimate if the SkipList <T> uses dynamic memory allocation such as `std::string`.

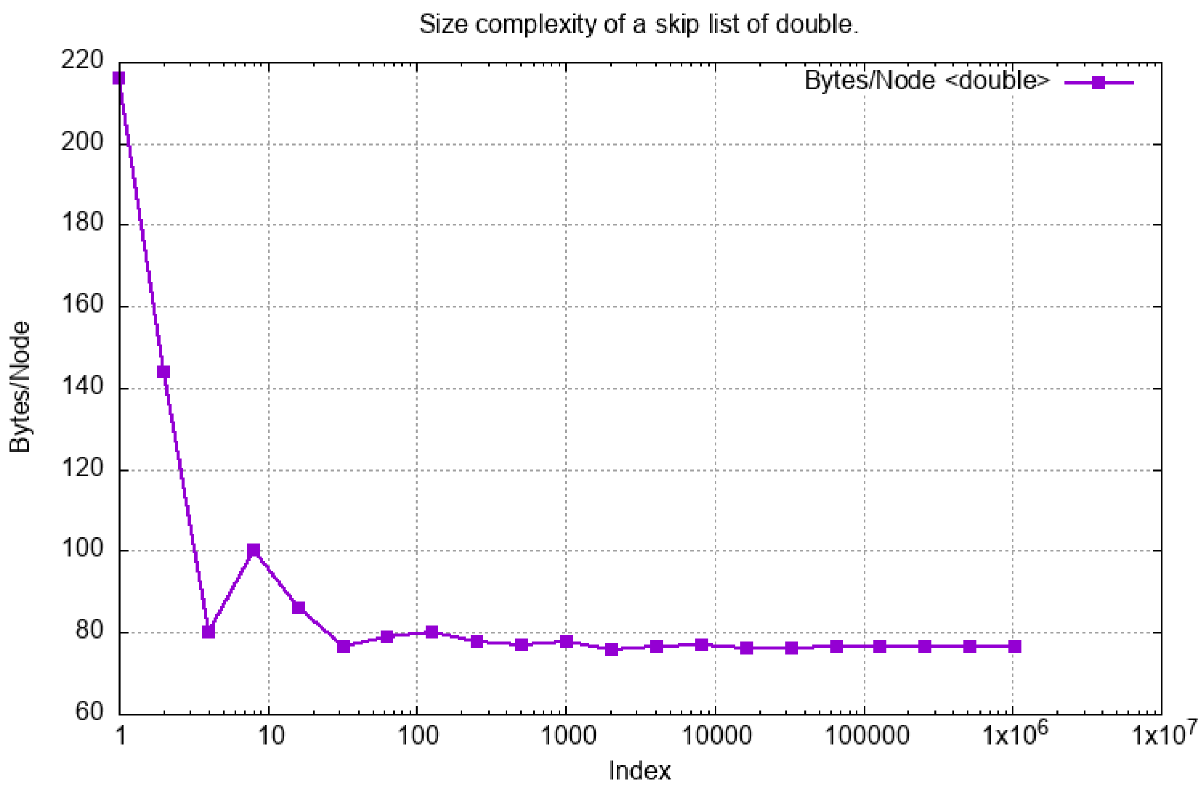
8.3.2 Estimate Memory Usage With `size_of()`

This implementation of a SkipList has a `size_of()` function that estimates the current memory usage of the SkipList. This function uses `sizeof(T)` which will not account for any dynamically allocated content, for example if `T` was a `std::string`.

Total memory allocation is a function of a number of factors:

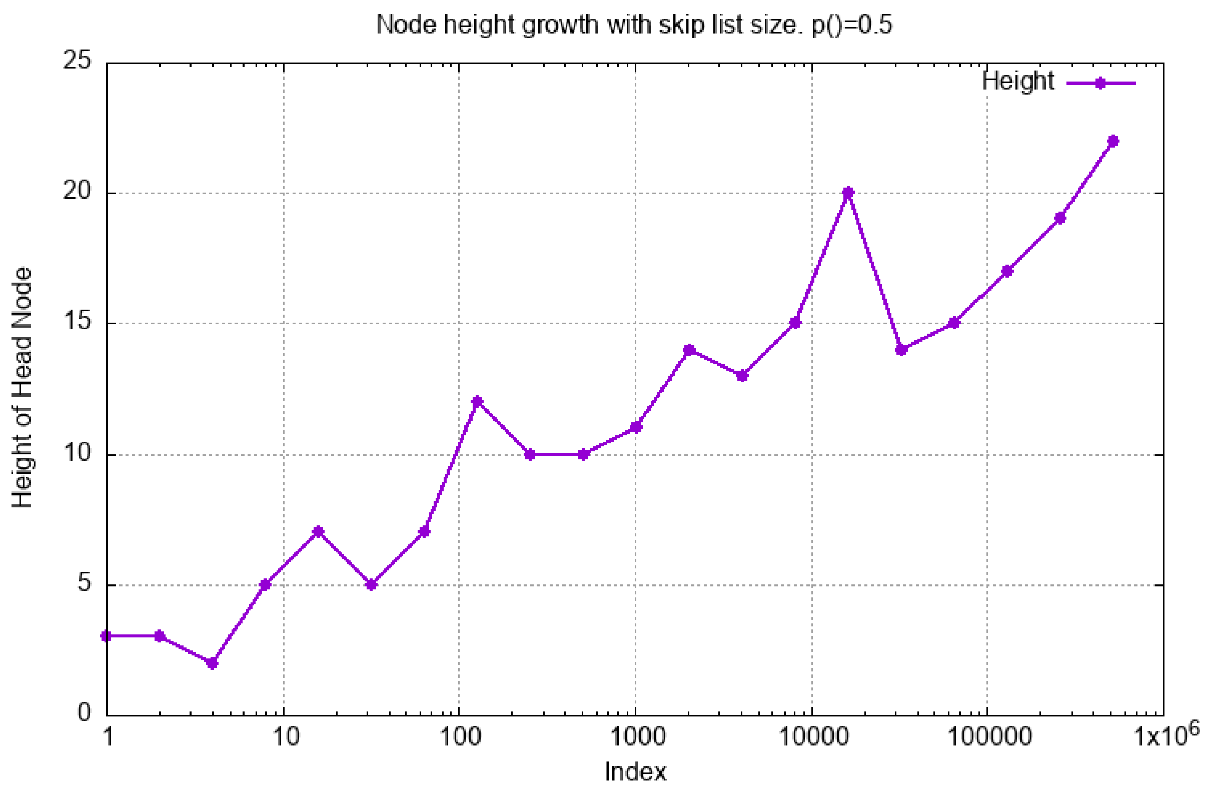
- Alignment issues with the members of `class Node` which has members `T _value;` and `SwappableNodeRefStack<T> _nodeRefs;`. If `T` was a `char` type then alignment issues on 64 bit machines may mean the `char` takes eight bytes, not one.
- The size of the SkipLists, very small SkipLists carry the overhead of the `HeadNode`.
- The coin probability `p()`. Unfair coins can change the overhead of the additional coarser linked lists. More about this later.

The following graph shows the `size_of()` a SkipList of doubles of varying lengths with a fair coin. The Y axis is the `size_of()` divided by the length of the SkipList in bytes per node. Fairly quickly this settles down to around 80 bytes a node or around 10 times the size of a single double. The test name is `perf_size_of()`.



8.3.3 Height Distribution

This graph shows the height growth of the SkipList where the height is the number of additional coarse linked lists. It should grow in a log(n) fashion and it does. It is not monotonic as this SkipList is a probabilistic data structure.



8.4 Effect of a Biased Coin

The default compilation of the SkipList uses a fair coin. The coin toss is determined by `tossCoin()` in *SkipList.cpp* which has the following implementation:

```
bool tossCoin() {
    return rand() < RAND_MAX / 2;
}
```

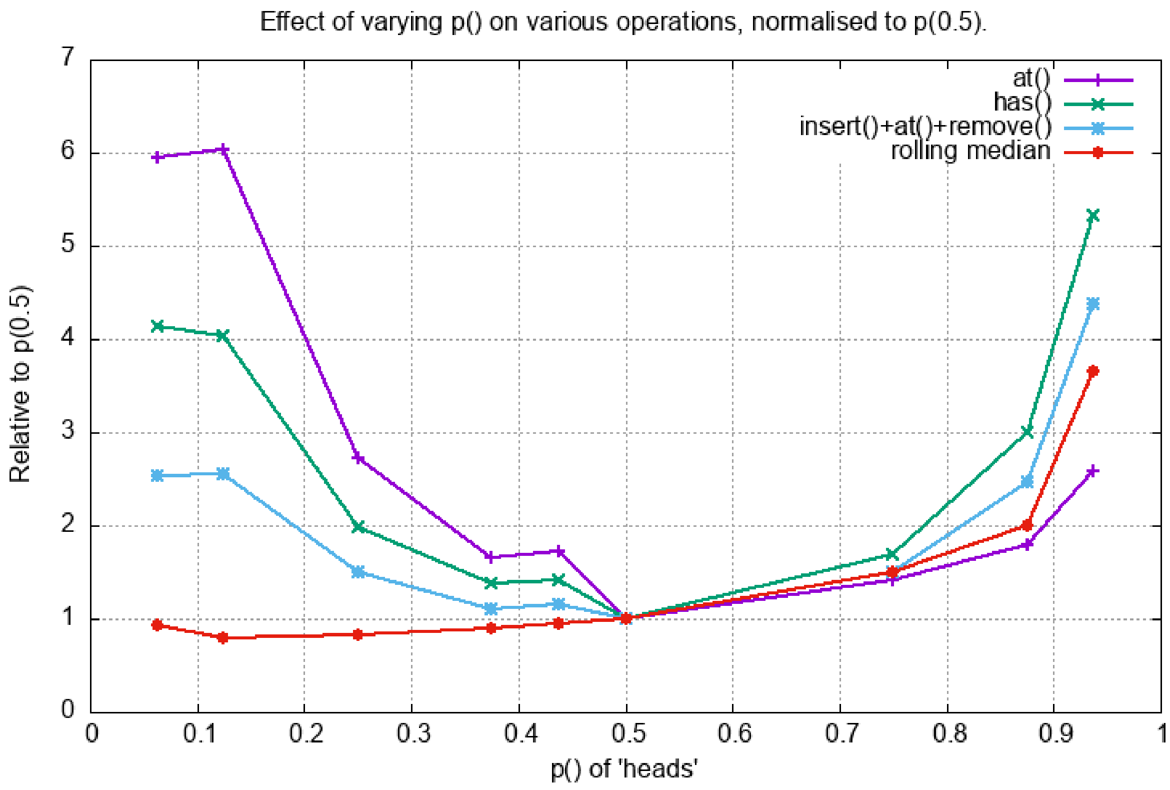
The following biases can be introduced with these return statements:

p()	Return statement
6.25%	`return rand() < RAND_MAX / 16;`
12.5%	`return rand() < RAND_MAX / 8;`
25%	`return rand() < RAND_MAX / 4;`
50%	`return rand() < RAND_MAX / 2;`
75%	`return rand() < RAND_MAX - RAND_MAX / 4;`
87.5%	`return rand() < RAND_MAX - RAND_MAX / 8;`

For visualising what a SkipList looks like with a biased coin see *Biased Coin Skip List Visualisations*

8.4.1 Time Performance

The following graph plots the time cost of `at(middle)`, `has(middle_value)`, `insert()`, `at()`, `remove()` and the rolling median (window size 101) all on a 1 million long SkipList of doubles against `p()` the probability of the coin toss being heads. The time cost is normalised to `p(0.5)`.



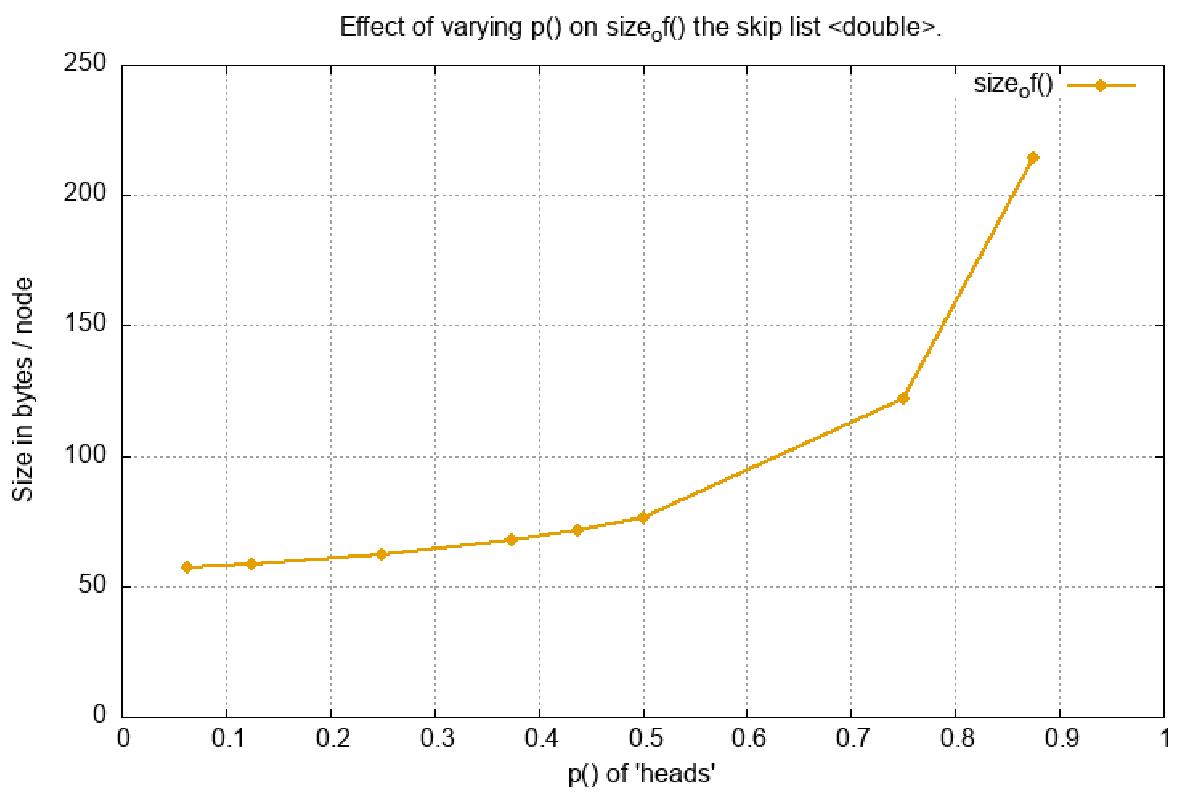
Reducing `p()` reduces the number of coarser linked lists that help speed up the search so it is expected that the performance would deteriorate. If `p()` was zero the SkipList would be, effectively, a singly linked list with O(n) search performance. I do not understand why the rolling median performance appears to improve slightly when the rolling median is really just an `insert()`, `at()`, `remove()` operation.

Increasing `p()` increases the number of coarser linked lists that might be expected to speed up the search. This does not do so in practice, possible explanations are:

- The increased cost of creating a node
- The increased memory usage (see next section)
- Poor locality of reference of the nodes.

8.4.2 Space Performance

Different values of `p()` greatly influences the space used as it directly affects the number of coarser linked lists created. In practice a reduction of `p()` provides some small space improvement.



If the SkipList was highly optimised for rolling median operations it might be worth experimenting with `p(0.25)` or even `p(0.125)` and evaluate the time/space requirements but otherwise there seems no reason, in the general case, to use anything but `p(0.5)`.

8.5 Multi-threaded C++ Performance

The C++ code is capable of multi-threading support where a single SkipList can be mutated by multiple threads. The code must be compiled with the macro `SKIPLIST_THREAD_SUPPORT` defined.

Test C++ execution code can be run by invoking the the makefile thus:

```
$ cd src/cpp
$ make release CXXFLAGS=-DSKIPLIST_THREAD_SUPPORT
$ ./SkipList_R.exe
              ...
              test_single_thread_insert(): PASS
      test_two_thread_insert_has_remove(): PASS
     test_two_thread_insert_count_has_remove_count(): PASS
    test_perf_insert_count_has_remove_count_multi_threads_vary_length(): threads: ␣
↪1 SkiplistSize:   131072 time:       180145 (us) rate       727592 /s
                                                        ...
   test_perf_single_thread_fixed_length(): PASS

Final result: PASS
Exec time: x.x (s)
Bye, bye!
```

Note: If you omit CXXFLAGS=-DSKIPLIST_THREAD_SUPPORT then the threaded tests will be omitted:

```
$ make release
$ ./SkipList_R.exe
               ...
               test_single_thread_insert(): N/A
       test_two_thread_insert_has_remove(): N/A
      test_two_thread_insert_count_has_remove_count(): N/A
```

8.5.1 Effect of SKIPLIST_THREAD_SUPPORT

Here are several performance measurements when SKIPLIST_THREAD_SUPPORT is defined:

- A SkipList in a single threaded environment.
- A SkipList in a multi threaded environment where threads vie for the same SkipList.

To explore this we create a task that is to insert a unique double into an empty SkipList 2**14 (16384) times and then remove that number one by one to empty the SkipList. This task typically takes 18 ms (around 1 us per insert+remove).

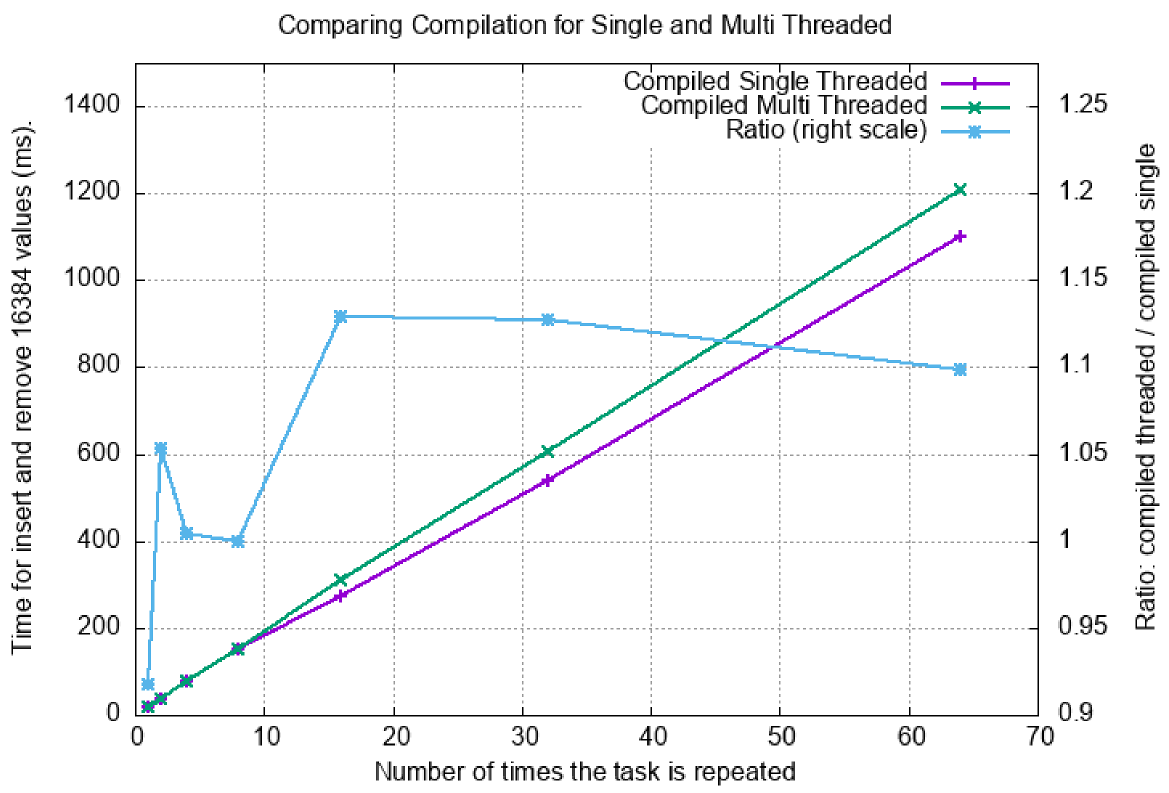
This task will be repeated 1, 2, 4, . . . 64 times using single and multiple threads. The single threaded version is sequential, the multithreaded version creates simultaneous operations on the same SkipList.

The code for these tests is in `test/test_concurrent.cpp`.

A Single Threaded Environment

The sheer act of using -DSKIPLIST_THREAD_SUPPORT will introduce a mutex into the head node. This will increase the time of any operation on the SkipList even when run in the single thread as there is a cost of acquiring the mutex even in the absence of contention. The test function is `test_perf_single_thread_fixed_length()`.

In the graph below the X axis is the number of times the task is repeated (sequentially). The left Y axis is the total execution time with the SkipList in the main thread. The right Y axis is the ratio: time with -DSKIPLIST_THREAD_SUPPORT / time without -DSKIPLIST_THREAD_SUPPORT



The overhead of using `-DSKIPLIST_THREAD_SUPPORT` is about 0% to 15%.

A Multi Threaded Environment

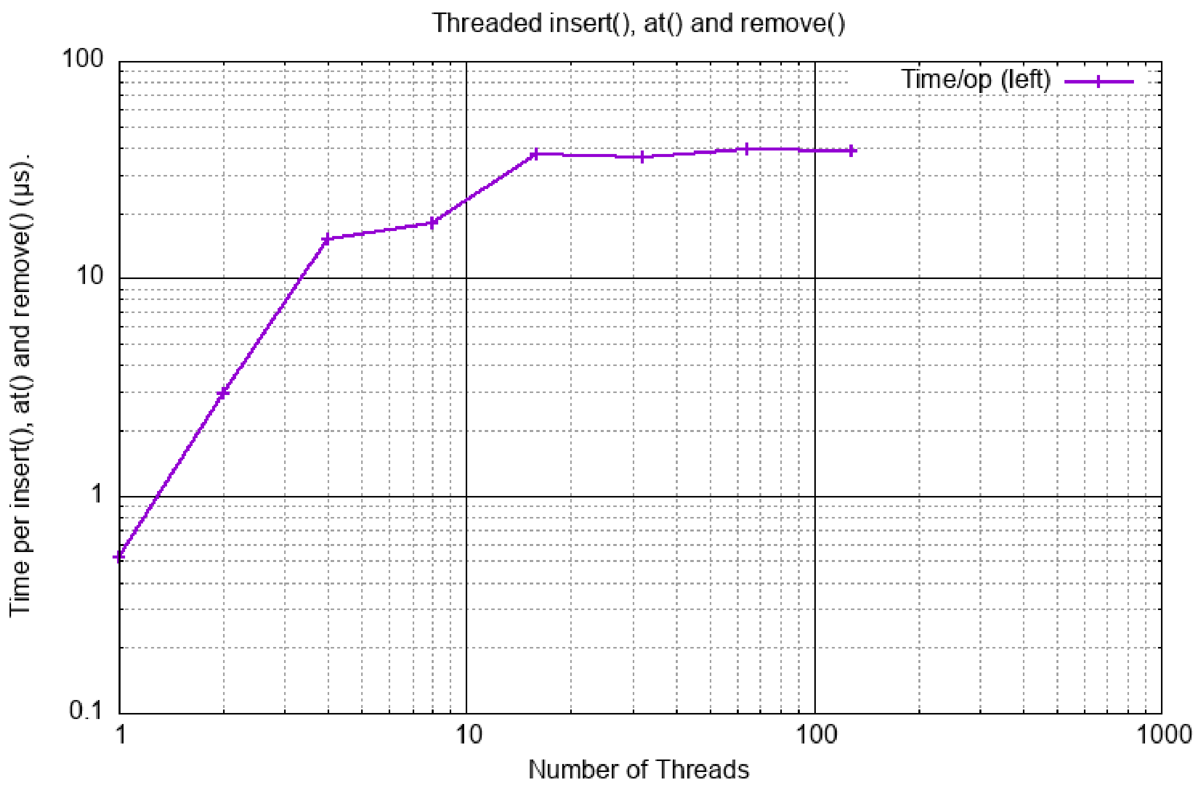
Here is a multi-threaded simulation of the operations typical of rolling median on a shared Skip List. Each thread:

- Inserts a value at the mid-point of the same Skip List using `insert()`.
- Checks it has that value using `has()`.
- Removes that value using `remove()`

Each of these operations obtains a lock on the Skip List.

Then repeating the task in 1, 2, 4, … 128 new threads simultaneously where they share the same Skip List. The test function is `_test_perf_sim_rolling_median_multi_thread()`.

In the graph below the Y-axis shows the cost for each insert/has/remove operation in microseconds against the number of threads contending for the Skip List.



8.6 Detailed Performance

The performance test function names all start with `perf_...` and are as follows. The SkipList type is `<double>`. In the table below 1M means mega, i.e. 2**20 or 1024*1024 or 1048576:

Test Name	Measure	Time	Rate
`perf_single`	With an empty SkipList add one item and remove it.	240 ns	4.1 M/s
`perf_large_`	Starting with an empty SkipList append 1 million values.	740 ns	1.3 M/s
`perf_large_`	Starting with an empty SkipList append 1 million values then remove the first (lowest) value until the SkipList is empty.	900 ns	1.1 M/s
`perf_single`	With a SkipList of 1 million values insert the middle value (i.e. 500,000.0) and remove it.	1200 ns	0.85 M/s
`perf_single`	With a SkipList of 1 million values find the middle value.	220 ns	4.6 M/s
`perf_single`	With a SkipList of 1 million values test for the middle value.	210 ns	4.8 M/s
`perf_single`	With a SkipList of 1 million values call `insert(v)`, `at(500000)` and `remove(v)` where `v` corresponds to the middle value. This simulates the actions of a rolling median.	1400 ns	0.7 M/s
`perf_mediar`	Simulate a rolling median of 100 values. Create an initially empty SkipList. For each of 10,000 random values insert the value into the SkipList. For indicies > 100 extract the middle value from the SkipList as the median then remove the i-100 value from the SkipList.	800 ns	1.3 M/s
`perf_1m_mec`	Simulate a rolling median of 101 values. Similar to `perf_median_sliding_window()` but uses 1 million values.	720 ns	1.4 M/s
`perf_1m_mec`	Simulate a rolling median of 101 values. Similar to `perf_1m_median_values()` but uses 1000 values repeated 1000 times.	690 ns	1.4 M/s
`perf_simula`	Simulate a rolling median of 200 values. Similar to `perf_1m_medians_1000_vectors()` but uses 8000 values repeated 8000 times i.e. the rolling median of 8000x8000 array.	760 ns	1.3 M/s
`perf_roll_m`	Tests the time cost of `OrderedStructs::RollingMedian::odd_index` for 1 million values and a window size of 101.	830 ns	1.2 M/s
`perf_index(`	Tests the time cost of `index()` half way through 1m doubles.	200 ns	5 M/s

8.6.1 Time Complexity

There are a number of tests that check the execution time of operations with varying sizes of SkipLists. The expectation is that the time complexity is O(log(n)).

Test Name	Description
`perf_at_in_one_millio`	For 1M values call `at(i)` where i ranges from 2**1 to 2**19. This explores the time complexity of `at()`.
`perf_has_in_one_milli`	For 1M values call `has(i)` where i ranges from 2**1 to 2**19. This explores the time complexity of `has()`.
`perf_roll_med_odd_ind`	As `perf_roll_med_odd_index()` but explores various window sizes from 1 to 524288.
`perf_index_vary_lengt`	For 1M values call `index(value)` where value ranges from 2**1 to 2**19. This explores the time complexity of `index()`.

8.7 Python Performance Tests

Some informal testing of the Python wrapper around the C++ SkipList was done using `timeit` in *tests/perf/test_perf_cSkipList.py*. The SkipList has 1m items. The performance is comparable to the C++ tests.

Test	Time per operation (ns)	Factor over C++ time
`test_at_integer()`	217	
`test_at_float()`	242	x2.7
`test_has_integer()`	234	
`test_has_float()`	238	x1.4
`test_insert_remove_mid_integer()`	1312	
`test_insert_remove_mid_float()`	1497	x1.4
`test_index_mid_int()`	400	
`test_index_mid_float()`	356	x1.9

It is rather surprising, and satisfying, that the Python overhead is so small considering the boxing/unboxing that is going on. The test methodology is different in the Python/C++ cases which might skew the figures.

CHAPTER

NINE

COMPUTING A ROLLING MEDIAN

9.1 Rolling Median in C++

A powerful use case for a skip list is in the computation of a rolling fraction, for example a rolling median.

Here is a reasonable C++ attempt at doing that with the arguments:

- `data` - A vector of data of type `T` of length `L`.
- `win_length` - a 'window' size. The median is computed over this number of values.
- `result` - a destination vector for the result. This will either end up with `L - win_length` values, alternatively is will be `L` long and start with `win_length` `NaN` s.

Rolling median code using a skip list might look like this, error checking is omitted:

```cpp
#include "SkipList.h"

template <typename T>
void rolling_median(const std::vector<T> data,
                    size_t win_length,
                    std::vector<T> &result) {

    OrderedStructs::SkipList::HeadNode<T> sl;

    result.clear();
    for (size_t i = 0; i < data.size(); ++i) {
        sl.insert(data[i]);
        if (i  >= win_length) {
            result.push_back(sl.at(win_length / 2));
            sl.remove(data[i - win_length]);
        }
    }
}
```

If you are working with C arrays (such as Numpy arrays) then this C'ish approach might be better, again error checking omitted:

```cpp
#include "SkipList.h"

template <typename T>
void rolling_median(const T *src, size_t count, size_t win_length, T *dest) {

    OrderedStructs::SkipList::HeadNode<T> sl;
    const T *tail = src;

    for (size_t i = 0; i < count; ++i) {
```

(continues on next page)

(continued from previous page)

```
        sl.insert(*src);
        if (i + 1 >= win_length) {
            *dest = sl.at(win_length / 2);
            ++dest;
            sl.remove(*tail);
            ++tail;
        }
        ++src;
    }
}
```

Multidimensional Numpy arrays have a stride value which is omitted in the above code but is simple to add. See *RollingMedian.h* and *test/test_rolling_median.cpp* for further examples.

Rolling percentiles require a argument that says what fraction of the window the required value lies. Again, this is easy to add.

9.1.1 Even Window Length

The above code assumes that if the window length is even that the median is at `(window length - 1) / 2`. A more plausible median for even sized window lengths is the mean of `(window length - 1) / 2` and `window length / 2`. This requires that the mean of two types is meaningful which it will not be for strings.

9.2 Rolling Median in Python

Here is an example of computing a rolling median of a `numpy` 1D array. This creates an array with the same length as the input starting with `window_length` `NaN` s:

```python
import numpy as np

import orderedstructs

def simple_python_rolling_median(vector: np.ndarray,
                                 window_length: int) -> np.ndarray:
    """Computes a rolling median of a numpy vector returning a new numpy
    vector of the same length.
    NaNs in the input are not handled but a ValueError will be raised."""
    if vector.ndim != 1:
        raise ValueError(
            f'vector must be one dimensional not shape {vector.shape}'
        )
    skip_list = orderedstructs.SkipList(float)
    ret = np.empty_like(vector)
    for i in range(len(vector)):
        value = vector[i]
        skip_list.insert(value)
        if i >= window_length - 1:
            # // 4 for lower quartile
            # * 3 // 4 for upper quartile etc.
            median = skip_list.at(window_length // 2)
            skip_list.remove(vector[i - window_length + 1])
        else:
            median = np.nan
        ret[i] = median
    return ret
```

This can be called thus:

```python
np_array = np.arange(10.0)
print('Original:', np_array)
result = simple_python_rolling_median(np_array, 3)
print('  Result:', result)
```

And the result will be:

```
Original: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
  Result: [nan nan  1.  2.  3.  4.  5.  6.  7.  8.]
```

9.3 Rolling Median in Python with `multiprocessing.shared_memory`

An exiting development in Python 3.8+ is multiprocessing.shared_memory This allows a parent process to share memory with its child processes.

In this example we are going to compute a rolling median on a 2D numpy array where each child process works on a single column of the same array and writes the result to a shared output array. There will be two shared memory areas; a read one with the input data and a write one with the result from all the child processes There will be two corresponding numpy arrays the input that we are given and the output numpy array that we create.

The only copying going on here is the initial copy of the input array into shared memory and then the final copy, when all child processes have completed of *that* shared memory to a single numpy array.

Pictorially:

```
Parent                                          Children
======                                          ========
Copies the array to the input SharedMemory
Creates the output SharedMemory
Launches n child processes...
\>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\
                                                Work on part of the input SharedMemory
                                                Write to the output SharedMemory
                                                ...
/<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<</
When all child processes complete...
Copies output SharedMemory to a new numpy array
Releases both SharedMemory resources.
```

Note: This solution assumes that you are given a numpy array that you need to process. An alternative solution is to create a shared memory object, create an empty numpy array that uses the shared memory buffer, populate that buffer and pass the buffer to the child processes. This would save the cost, in time and memory, of the first copy operation.

9.3.1 Code

First let's write some code that wraps the low level `multiprocessing.shared_memory.SharedMemory` class that can be used by the parent process. This is a dataclass that records essential information about the array and includes the `SharedMemory` object itself. We will call it an `SharedMemoryArraySpecification`, it is pretty simple, just a named tuple:

```python
import multiprocessing
import typing
from multiprocessing import shared_memory

import numpy as np

import orderedstructs

class SharedMemoryArraySpecification(typing.NamedTuple):
    shape: typing.Tuple[int, ...]
    dtype: np.dtype
    nbytes: int
    shm: shared_memory.SharedMemory

    @property
    def name(self) -> str:
        return self.shm.name

    def __str__(self):
        return (
            f'<SharedMemoryArraySpecification shape {self.shape} dtype {self.dtype} name "{self.name}"'
            f' nbytes {self.nbytes} buffer id 0x{id(self.shm)}>'
        )

    def close(self) -> None:
        self.shm.close()

    def close_and_unlink(self) -> None:
        self.close()
        self.shm.unlink()
```

Here is the rolling median function used by the child processes:

```python
def rolling_median_of_column(read_array: np.ndarray, window_length: int, column_index: int,
                             write_array: np.ndarray) -> int:
    """Computes a rolling median of given column and writes out the results to the write array.
    Called by a child process."""
    assert read_array.ndim == 2
    assert write_array.ndim == 2
    assert read_array.shape == write_array.shape
    skip_list = orderedstructs.SkipList(float)
    write_count = 0
    for i in range(len(read_array)):
        value = read_array[i, column_index]
        skip_list.insert(value)
        if i >= window_length:
            median = skip_list.at(window_length // 2)
            skip_list.remove(read_array[i - window_length, column_index])
```

(continues on next page)

(continued from previous page)

```
            write_count += 1
        else:
            median = np.nan
        write_array[i, column_index] = median
    return write_count
```

Now create a function that takes a numpy array, creates the `SharedMemory` and returns an `ArraySpecification`:

```
# NOTE: This code used by the parent process.

def shared_memory_and_array_spec(arr: np.ndarray) -> ArraySpecification:
    """Given a numpy array create a SharedMemory object and encapsulate it in an
    ArraySpecification."""
    shm = shared_memory.SharedMemory(create=True, size=arr.nbytes)
    array_spec = ArraySpecification(arr.shape, arr.dtype, arr.nbytes, shm)
    return array_spec
```

Now create a context manager that will wrap a numpy array around a `SharedMemoryArraySpecification` On exit this automatically releases the reference to the shared memory from the child process.

```
@contextlib.contextmanager
def recover_array_from_shared_memory_and_close(array_spec:␣
↪SharedMemoryArraySpecification) -> np.ndarray:
    array_shm = shared_memory.SharedMemory(name=array_spec.name)
    array_view = np.ndarray(array_spec.shape, array_spec.dtype, buffer=array_shm.buf)
    try:
        yield array_view
    finally:
        array_shm.close()
```

And use it in the child process:

```
def compute_rolling_median_2d_from_index(read_array_spec:␣
↪SharedMemoryArraySpecification, window_length: int,
                                         column_index: int,
                                         write_array_spec:␣
↪SharedMemoryArraySpecification) -> int:
    """Computes a rolling median of the 2D read array and window length and writes it␣
↪to the 2D write array.
    This is invoked by a child process."""
    with recover_array_from_shared_memory_and_close(read_array_spec) as read_array:
        with recover_array_from_shared_memory_and_close(write_array_spec) as write_
↪array:
            write_count = rolling_median_of_column(read_array, window_length, column_
↪index, write_array)
            return write_count
```

Create two more context managers, one to copy the input numpy array to shared memory:

```
@contextlib.contextmanager
def create_read_shared_memory_array_spec_close_unlink(arr: np.ndarray) ->␣
↪SharedMemoryArraySpecification:
    """Context manager that creates a Shared Memory instance and copies the numpy␣
↪array into it.
    The Shared Memory instance is closed and unlinked on exit."""
    shm = shared_memory.SharedMemory(create=True, size=arr.nbytes)
    array_spec = SharedMemoryArraySpecification(arr.shape, arr.dtype, arr.nbytes, shm)
```

(continues on next page)

(continued from previous page)

```
    logger.info('Created shared memory %s ', array_spec)
    try:
        # Copy the numpy array into shared memory.
        array_view = np.ndarray(array_spec.shape, dtype=array_spec.dtype,␣
↪buffer=array_spec.shm.buf)
        array_view[:] = arr[:]
        yield array_spec
    finally:
        array_spec.close_and_unlink()
```

And another to wrap around the output numpy array:

```
@contextlib.contextmanager
def create_write_shared_memory_array_spec_close_unlink(arr: np.ndarray) ->␣
↪SharedMemoryArraySpecification:
    """Context manager that creates a Shared Memory instance for writing to a numpy␣
↪array.
    The numpy array can be recovered with copy_shared_memory_into_new_numpy_array().
    The Shared Memory instance is closed and unlinked on exit."""
    shm = shared_memory.SharedMemory(create=True, size=arr.nbytes)
    array_spec = SharedMemoryArraySpecification(arr.shape, arr.dtype, arr.nbytes, shm)
    logger.info('Created shared memory %s ', array_spec)
    try:
        yield array_spec
    finally:
        array_spec.close_and_unlink()
```

Finally a function to copy the output shared memory to a numpy array:

```
def copy_shared_memory_into_new_numpy_array(write_array_spec:␣
↪SharedMemoryArraySpecification) -> np.ndarray:
    """With the output SharedMemoryArraySpecification this creates a new numpy array␣
↪and copies the shared memory into it."""
    temp_write = np.ndarray(write_array_spec.shape, dtype=write_array_spec.dtype,␣
↪buffer=write_array_spec.shm.buf)
    write_array = np.empty(write_array_spec.shape, dtype=write_array_spec.dtype)
    write_array[:] = temp_write[:]
    return write_array
```

Finally here is the code for the parent process that puts this all together:

```
def compute_rolling_median_2d_mp(read_array: np.ndarray, window_length: int, num_
↪processes: int) -> np.ndarray:
    """Compute a rolling median of a numpy 2D array using multiprocessing and shared␣
↪memory.
    This is run as the parent process."""
    if read_array.ndim != 2:
        raise ValueError(f'Array  must be 2D not {read_array.shape}')
    # Limit number of processes if the number of columns is small.
    if read_array.shape[1] < num_processes:
        num_processes = read_array.shape[1]
    # Create the read and write shared memory context managers
    with create_read_shared_memory_array_spec_close_unlink(read_array) as read_array_
↪spec:
        with create_write_shared_memory_array_spec_close_unlink(read_array) as write_
↪array_spec:
            # Set up the multiprocessing pool.
```

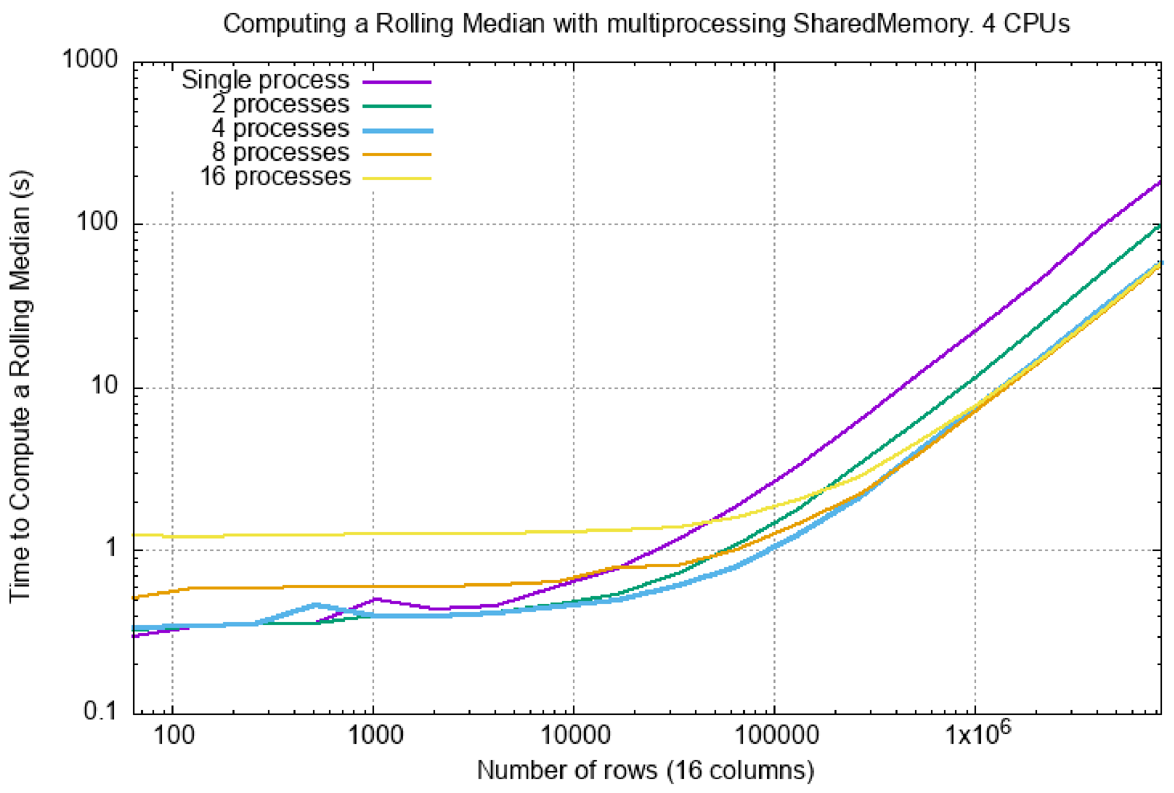
(continues on next page)

(continued from previous page)

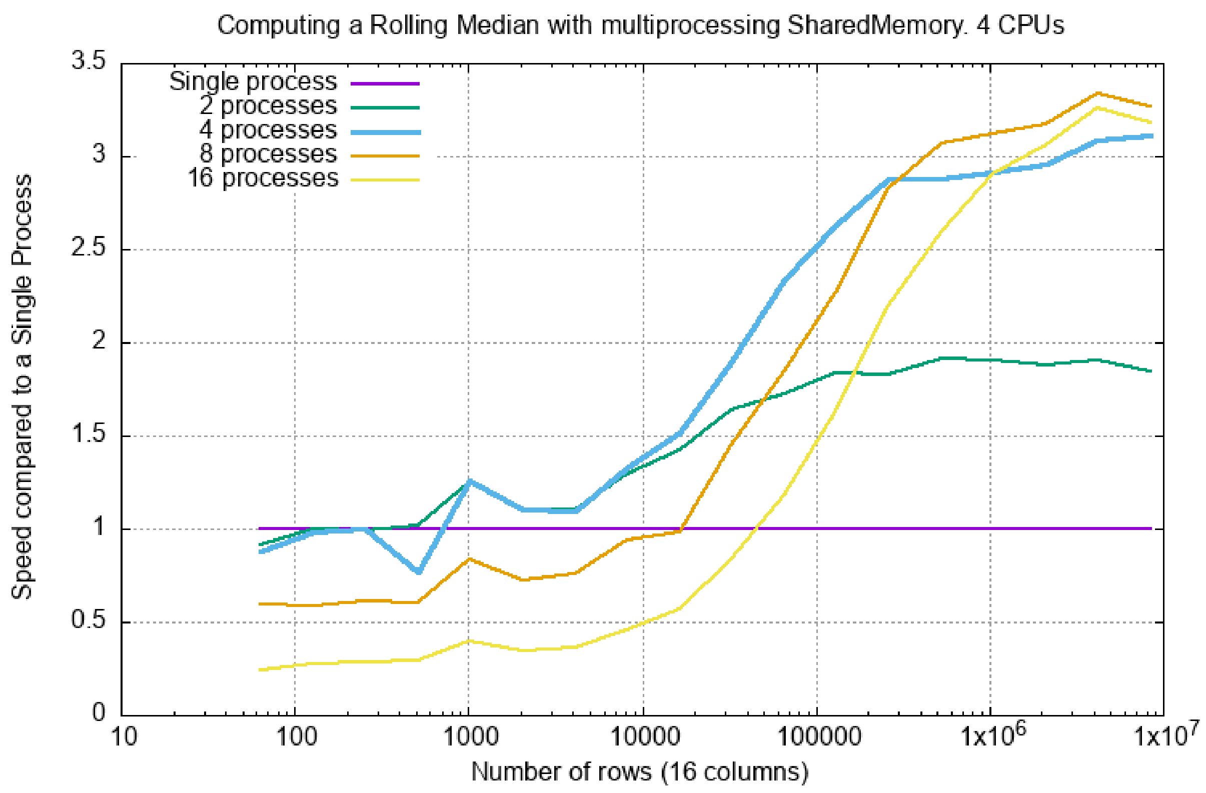
```
            mp_pool = multiprocessing.Pool(processes=num_processes)
            tasks = []
            for column_index in range(read_array.shape[1]):
                tasks.append((read_array_spec, window_length, column_index, write_
↪array_spec))
            # Run compute_rolling_median_2d_from_index() on the pool
            pool_apply = [mp_pool.apply_async(compute_rolling_median_2d_from_index,␣
↪t) for t in tasks]
            results = [r.get() for r in pool_apply]
            # Extract the result as a numpy array.
            write_array = copy_shared_memory_into_new_numpy_array(write_array_spec)
    return write_array
```

9.3.2 Performance

Running this on 16 column arrays with up to 1m rows with processes from 1 to 16 gives the following execution times. Mac OS X with 4 cores and hyper-threading:



Comparing the **speed** of execution compared to a single process gives:



Clearly there is some overhead so it is not really worth doing this for less that 10,000 rows. The number of processes equal to the number of CPUs is optimum, twice that *might* give a *small* advantage.

9.3.3 Memory Usage

What I would expect in processing a 100Mb numpy array. Values are in Mb.

Table 1: Expected `multiprocessing.shared_memory` Memory Usage With 100 Mb numpy array

Action	Memory Delta (Mb)	Total Memory (Mb)
Create a 'read' numpy array.	+100	100
Create a 'read' shared memory object.	+100	200
Copy the 'read' array into 'read' shared memory.	0 or very small.	200
Create a 'write' shared memory object.	+100	300
Calculate the rolling median and write into the 'write' shared memory object.	0 or very small.	300
Create an empty 'write' numpy array.	+100	400
Copy the 'write' shared memory into the 'write' numpy array.	0 or very small.	400
Unlink the 'read' shared memory	-100	300
Unlink the 'write' shared memory.	-100	200
Delete the 'read' numpy array when de-referenced.	-100	100
Delete the 'write' numpy array when de-referenced.	-100	Nominal.

Here are the actual results running on Mac OS X. Two things are noticeable:

- Creating the shared memory object has no memory cost. It is only when data is copied into it that the memory is allocated and that is incremental.

- The RSS shown here is collected from `psutil` and it looks like this is including shared memory so there may be double counting here. `psutil` can not identify shared memory on Mac OS X, it can on Linux.

Here is the breakdown of the RSS memory profile of processing a numpy array with 6m rows with 2 columns (100Mb) with a parent [P] and two child processes [0] and [1]. The change in RSS is indicated by "d" (if non-zero). Values are in Mb.

Table 2: `multiprocessing.shared_memory` Memory Usage With Two Processes

Action	P	dP	0	d0	1	d1	Notes
Parent start	30	+30					Normal Python executable.
Create numpy array	130	+100					Cost of creating a 100Mb numpy array.
Create read shared memory	130						No immediate memory cost.
Copy numpy array into shared memory	225	+95					
Create write shared memory	225						No immediate memory cost.
Child start			23	+23	23	+23	Normal Python executable.
Rolling median start			23		23		
Rolling median 25%			71	+48	71	+48	Incremental memory increase, similar to copy on write.
Rolling median 50%			119	+48	119	+48	
Rolling median 75%			166	+47	166	+47	
Rolling median complete			214	+48	214	+48	Peak figure, it looks like the RSS for the child processes is including both shared memory areas (192Mb).
Close write shared memory			119	-95	119	-95	
Close read shared memory			23	-96	23	-96	Child process now using the normal memory for a Python process.
After child processes complete.	227	+2					Children have written to write shared memory which is now included in the parent memory RSS.
After creating empty numpy write array.	227						NOTE: Buffer is lazily allocated.
After writing write shared memory to numpy write array.	419	+192					Not sure why this twice what is expected (100Mb).
After unlink write array spec.	321	-98					
After unlink read array spec.	226	-95					Discard read array shared memory. Numpy read and write arrays still exist, 100Mb each.
Parent process ends.	227						Read array and write array discarded. See note below.

Note: There is an interesting quirk here, the array is 6m rows with 2 columns and has a residual memory of 227Mb. This is not reduced by a `gc.collect()`. This does not increase if the same function calls are repeated. If the array is changed to *16m* rows, 2 columns (260Mb) the residual memory is 35Mb, typical for a minimal Python process.

9.4 Handling NaNs

Not-a-number (NaN) values can not be inserted into a Skip List as they are not comparable to anything (including themselves). An attempt to call `insert()`, `index()`, `has()`, `remove()` with a NaN will raise an error. In C++ this will throw a `OrderedStructs::SkipList::FailedComparison`. In Python it will raise a `ValueError`. This section looks at handling NaNs in Python.

Here are several ways of handling NaNs:

- Propogate the Exception.
- Make the Median NaN.
- Forward Filling.

9.4.1 Propogate the Exception

Here is a rolling median that will raise a `ValueError` if there is a NaN in the input.

```python
def rolling_median_no_nan(vector: typing.List[float],
                          window_length: int) -> typing.List[float]:
    """Computes a rolling median of a vector of floats and returns the results.
    NaNs will throw an exception."""
    skip_list = orderedstructs.SkipList(float)
    ret: typing.List[float] = []
    for i in range(len(vector)):
        value = vector[i]
        skip_list.insert(float(value))
        if i >= window_length:
            median = skip_list.at(window_length // 2)
            skip_list.remove(vector[i - window_length])
        else:
            median = math.nan
        ret.append(median)
    return ret
```

9.4.2 Make the Median NaN

Here is a rolling median that will make the median NaN if there is a NaN in the input. Incidentally this is the approach that numpy takes.

```python
def rolling_median_with_nan(vector: typing.List[float],
                            window_length: int) -> typing.List[float]:
    """Computes a rolling median of a vector of floats and returns the results.
    NaNs will be consumed."""
    skip_list = orderedstructs.SkipList(float)
    ret: typing.List[float] = []
    for i in range(len(vector)):
        value = vector[i]
        if math.isnan(value):
            median = math.nan
        else:
            skip_list.insert(float(value))
            if i >= window_length:
                median = skip_list.at(window_length // 2)
                remove_value = vector[i - window_length]
                if not math.isnan(remove_value):
```

(continues on next page)

(continued from previous page)

```
                    skip_list.remove(remove_value)
            else:
                median = math.nan
        ret.append(median)
    return ret
```

The first row is the input, the second the output. Window length is 5:

```
[0.0,      math.nan,      2.0,      3.0,      4.0, 5.0, 6.0, math.nan, 8.0, 9.0],
[math.nan, math.nan, math.nan, math.nan, math.nan, 3.0, 4.0, math.nan, 4.0, 5.0],
```

9.4.3 Forward Filling

Another approach is to replace the NaN with the previous value. This is *very popular* in FinTech and is commonly know as *Forward Filling*. Here is an implementation:

```
def forward_fill(vector: typing.List[float]) -> None:
    """Forward fills NaNs in-place."""
    previous_value = math.nan
    for i in range(len(vector)):
        value = vector[i]
        if math.isnan(value):
            vector[i] = previous_value
        if not math.isnan(value):
            previous_value = value

def rolling_median_with_nan_forward_fill(vector: typing.List[float],
                                         window_length: int) -> typing.List[float]:
    """Computes a rolling median of a vector of floats and returns the results.
    NaNs will be forward filled."""
    forward_fill(vector)
    return rolling_median_no_nan(vector, window_length)
```

The first row is the input, the second the output. Window length is 5:

```
[0.0,      math.nan,      2.0,      3.0,      4.0, 5.0, 6.0, math.nan, 8.0, 9.0],
[math.nan, math.nan, math.nan, math.nan, math.nan, 2.0, 3.0,      4.0, 5.0, 6.0],
```

Another example, the first row is the input, the second the output. Window length is 5:

```
[0.0,      math.nan,      2.0, math.nan,      4.0, 5.0, 6.0, math.nan, 8.0, 9.0],
[math.nan, math.nan, math.nan, math.nan, math.nan, 2.0, 2.0,      4.0, 5.0, 6.0],
```

There is no ‘right way’ to handle NaNs. They are always problematic. For example what is the ‘right way’ to sort a sequence of values that may include NaNs?

CHAPTER

TEN

VISUALISING A SKIP LIST

If the skip list is compiled with `INCLUDE_METHODS_THAT_USE_STREAMS` defined both the `HeadNode` and `Node` gain the methods `dotFile` and `dotFileFinalise`. These can be used to write out the current state of the skip list to sdtout in DOT (GraphViz) format. These can be converted to SVG or PNG using dot.

Multiple snapshots of the same skiplist can be taken and plotted in the same graph. Each snapshot is denoted by a number.

Function	Description
`dotFile()`	Arguments are: an output stream.
`dotFileFinalise()`	Arguments are: an output stream.

10.1 Simple Example in C++

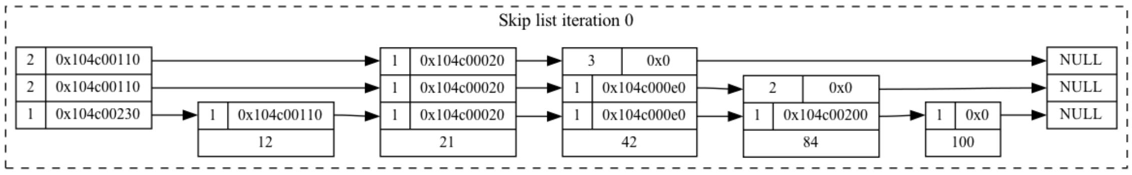
This takes a single snapshot of the skip list. Create a skip list and insert 5 values into it. Then call `dotFile()` to write out the state of the skip list stdout and `dotFinalise()` to complete the graph.

```
#include "SkipList.h"

void doc_insert_simple() {
    OrderedStructs::SkipList::HeadNode<int> sl;

    sl.insert(42);
    sl.insert(84);
    sl.insert(21);
    sl.insert(100);
    sl.insert(12);

    sl.dotFile(std::cout);
    sl.dotFileFinalise(std::cout);
}
```

Saving `stdout` to a text file, say *doc_simple.dot* then running this on the command line:

```
dot -odoc_simple.png -Tpng doc_simple.dot
```

Will produce something like this SVG diagram:



SkipList.

10.1.1 Inserting the Values 0 to 7

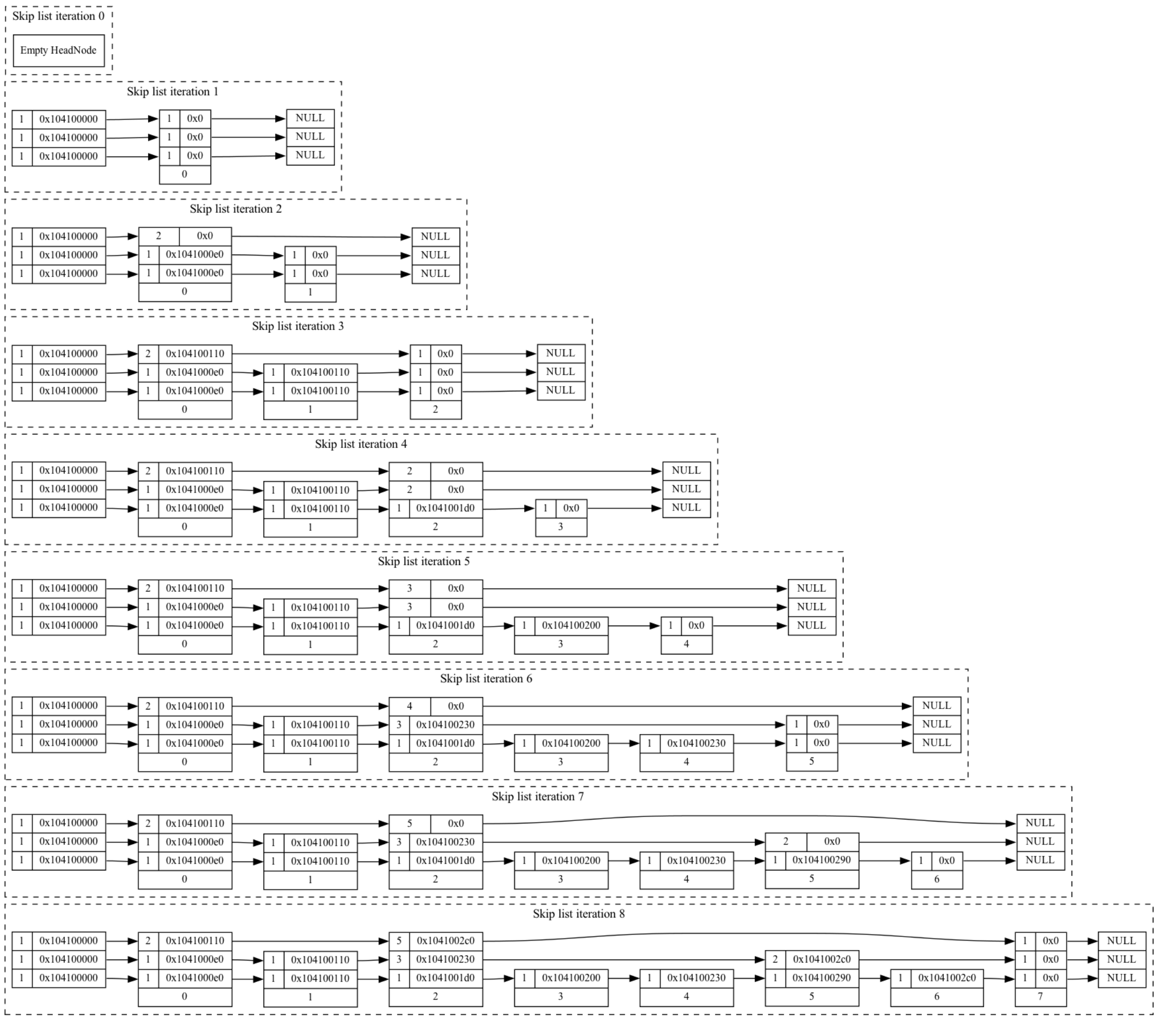
Multiple snapshots can also be created showing how the skiplist grows and shrinks. This diagram was created with the following C++ code:

```
#include "SkipList.h"

void doc_insert() {
    OrderedStructs::SkipList::HeadNode<int> sl;

    // Write out the empty head node
    sl.dotFile(std::cout);
    // Now insert a value and add the current representation to the DOT file
    for (int i = 0; i < 8; ++i) {
        sl.insert(i);
        sl.dotFile(std::cout);
    }
    // Finalise the dot file with the number of snapshots, this updates internal␣
↪references
    sl.dotFileFinalise(std::cout);
}
```

Saving this output to a text file, say *doc_insert.dot* then running this on the command line:

```
dot -odoc_insert.png -Tpng doc_insert.dot
```

Will produce this [*doc_insert.png*]:



SkipList.

10.1.2 Inserting the Values 0 to 3 and Removing them Multiple Times

This visualisation is produced by this code:

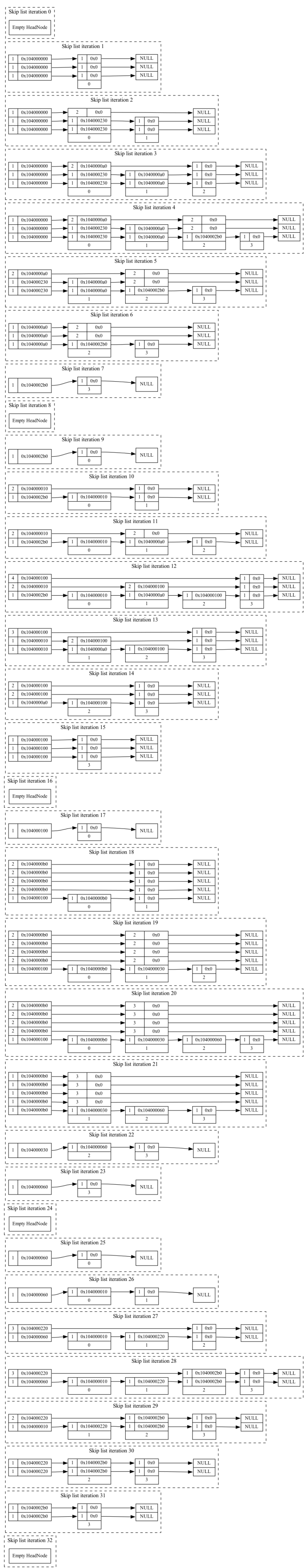
```cpp
#include "SkipList.h"

void doc_insert_remove_repeat() {
    int NUM = 4;
    int REPEAT_COUNT = 4;

    OrderedStructs::SkipList::HeadNode<int> sl;

    sl.dotFile(std::cout);
    for (int c = 0; c < REPEAT_COUNT; ++c) {
        for (int i = 0; i < NUM; ++i) {
            sl.insert(i);
            sl.dotFile(std::cout);
        }
        for (int i = 0; i < NUM; ++i) {
            sl.remove(i);
            sl.dotFile(std::cout);
        }
    }
```

(continues on next page)

(continued from previous page)

```
    sl.dotFileFinalise(std::cout);
}
```

Produces this image, note how the shape of the skip list nodes changes with repeated inserts.


Skip list iteration 0
Empty HeadNode
Skip list iteration 1
Skip list iteration 2
Skip list iteration 3
Skip list iteration 4
Skip list iteration 5
Skip list iteration 6
Skip list iteration 7
Skip list iteration 8
Empty HeadNode
Skip list iteration 9
Skip list iteration 10
Skip list iteration 11
Skip list iteration 12
Skip list iteration 13
Skip list iteration 14
Skip list iteration 15
Skip list iteration 16
Empty HeadNode
Skip list iteration 17
Skip list iteration 18
Skip list iteration 19
Skip list iteration 20
Skip list iteration 21
Skip list iteration 22
Skip list iteration 23
Skip list iteration 24
Empty HeadNode
Skip list iteration 25
Skip list iteration 26
Skip list iteration 27
Skip list iteration 28
Skip list iteration 29
Skip list iteration 30
Skip list iteration 31
Skip list iteration 32
Empty HeadNode
SkipList.

10.2 Simple Example in Python

The Python interface is via a single function `.dot_file()` that returns a bytes object that is suitable for saving as a .dot file.

```python
import cSkipList

sl = cSkipList.PySkipList(float)

sl.insert(42.0)
sl.insert(21.0)
sl.insert(84.0)

dot_bytes = sl.dot_file()

with open('doc_simple_py.dot', 'w') as dot_file:
    dot_file.write(dot_bytes.decode('ascii'))
```

doc_simple_py.dot will look something like:

```
digraph SkipList {
label = "SkipList."
graph [rankdir = "LR"];
node [fontsize = "12" shape = "ellipse"];
edge [];

subgraph cluster0 {
style=dashed
label="Skip list iteration 0"

"HeadNode0" [
label = "{ 1 | <f2> 0x7f8a68d86280} | { 1 | <f1> 0x7f8a68d86280}"
shape = "record"
];
"HeadNode0":f1 -> "node00x7f8a68d86280":w1 [];
"HeadNode0":f2 -> "node00x7f8a68d86280":w2 [];

"node00x7f8a68d86280" [
label = " { <w2> 2 | <f2> 0x7f8a68d9dcc0 } | { <w1> 1 | <f1> 0x7f8a68d44ec0 } | <f0>␣
↪21"
shape = "record"
];
"node00x7f8a68d86280":f1 -> "node00x7f8a68d44ec0":w1 [];
"node00x7f8a68d86280":f2 -> "node00x7f8a68d9dcc0":w2 [];
"node00x7f8a68d44ec0" [
label = " { <w1> 1 | <f1> 0x7f8a68d9dcc0 } | <f0> 42"
shape = "record"
];
"node00x7f8a68d44ec0":f1 -> "node00x7f8a68d9dcc0":w1 [];
"node00x7f8a68d9dcc0" [
label = " { <w2> 1 | <f2> 0x0 } | { <w1> 1 | <f1> 0x0 } | <f0> 84"
shape = "record"
];
"node00x7f8a68d9dcc0":f1 -> "node00x0":w1 [];
"node00x7f8a68d9dcc0":f2 -> "node00x0":w2 [];

"node00x0" [label = "<w2> NULL | <w1> NULL" shape = "record"];
```

(continues on next page)

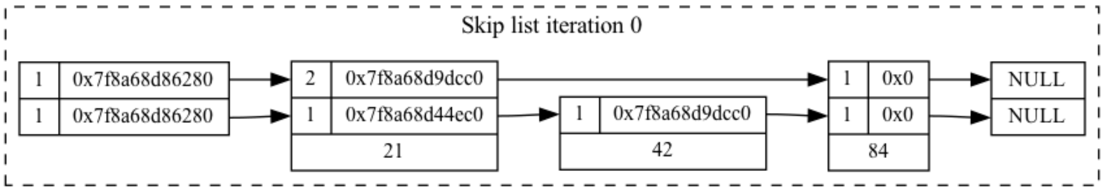
(continued from previous page)

```
}

node0 [shape=record, label = "<f0> | ", style=invis, width=0.01];
node0:f0 -> HeadNode0 [style=invis];
}
```

Running:

```
dot -odoc_simple_py.png -Tpng doc_simple_py.dot
```

Will produce something like this:



SkipList.

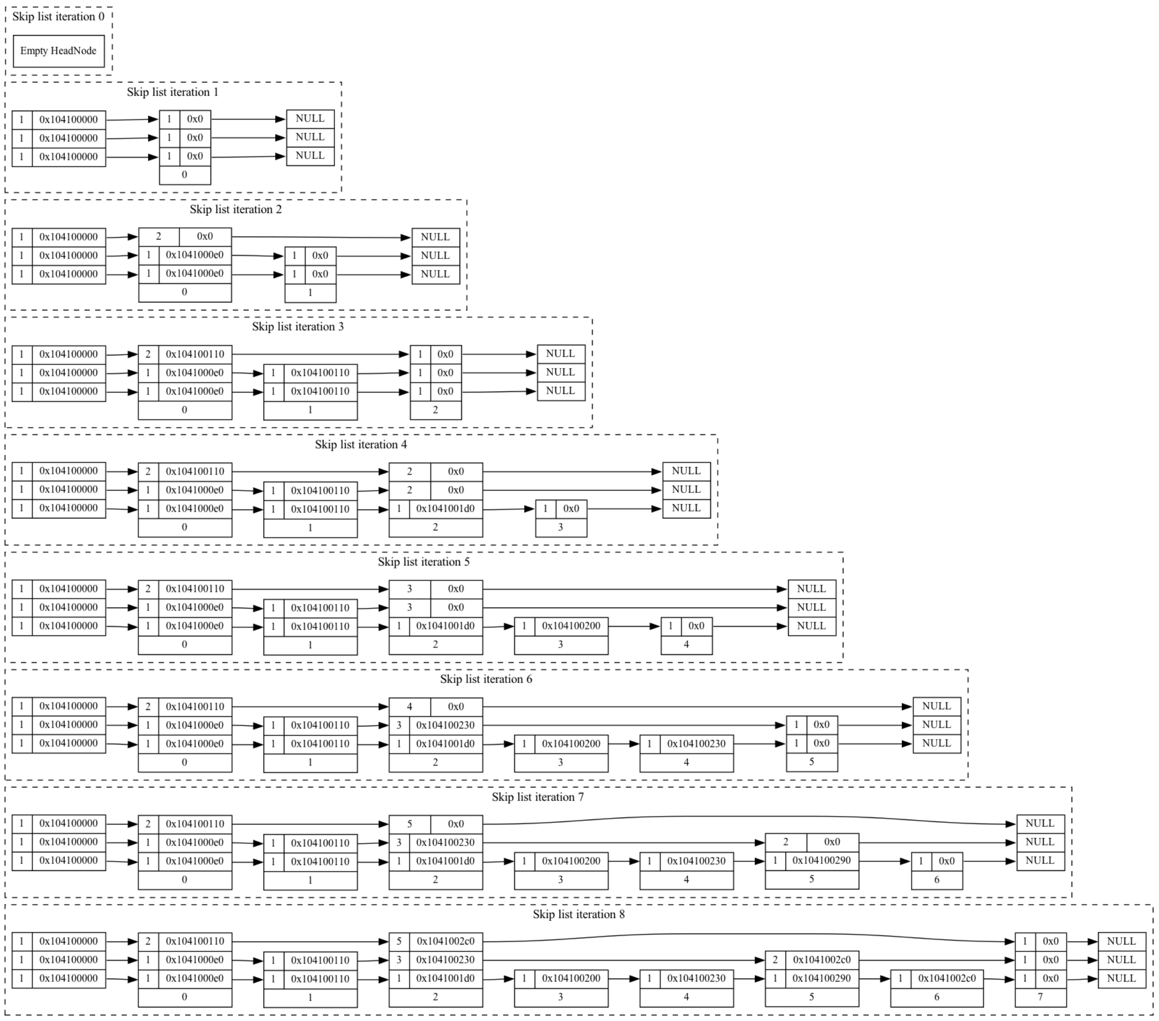
CHAPTER

ELEVEN

BIASED COIN SKIP LIST VISUALISATIONS

11.1 Fair coin, `p(0.5)`

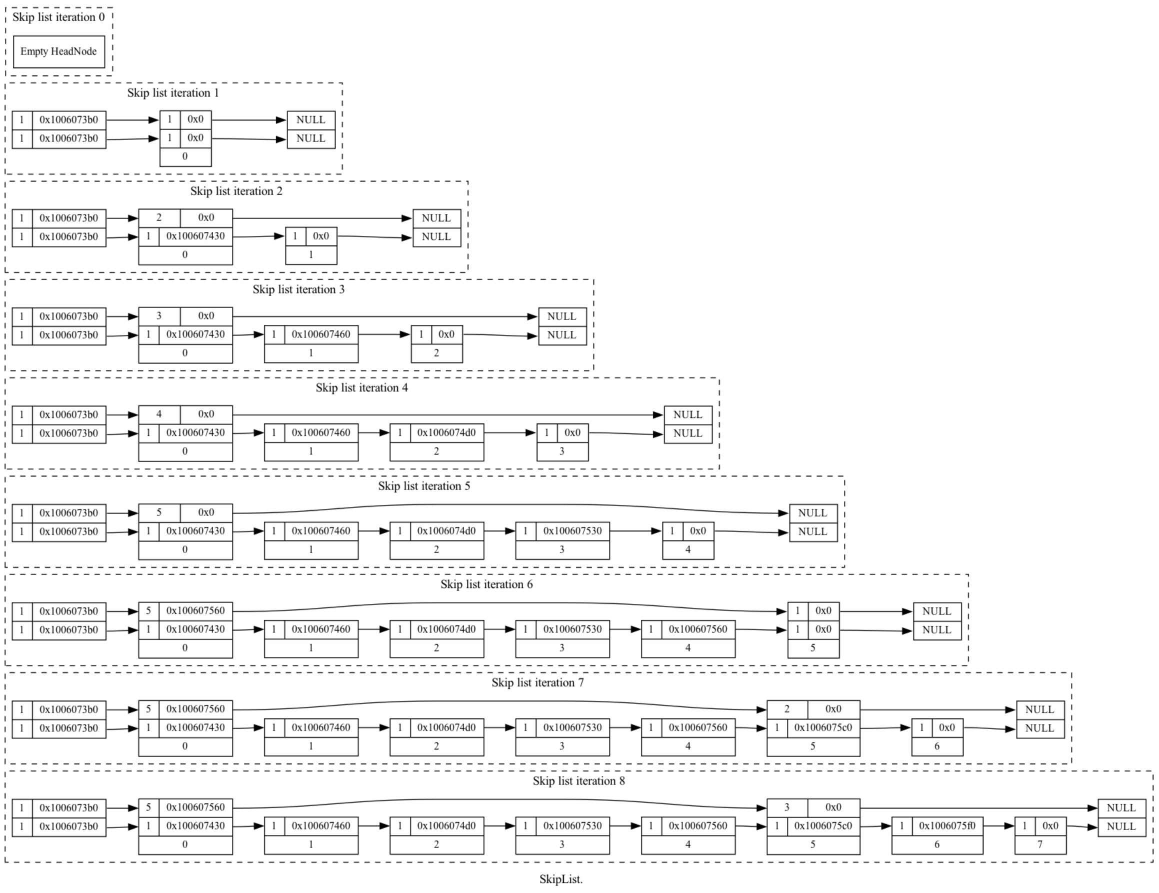
This is the default implementation of the skip list. Each coarser linked list (should) have half the references.



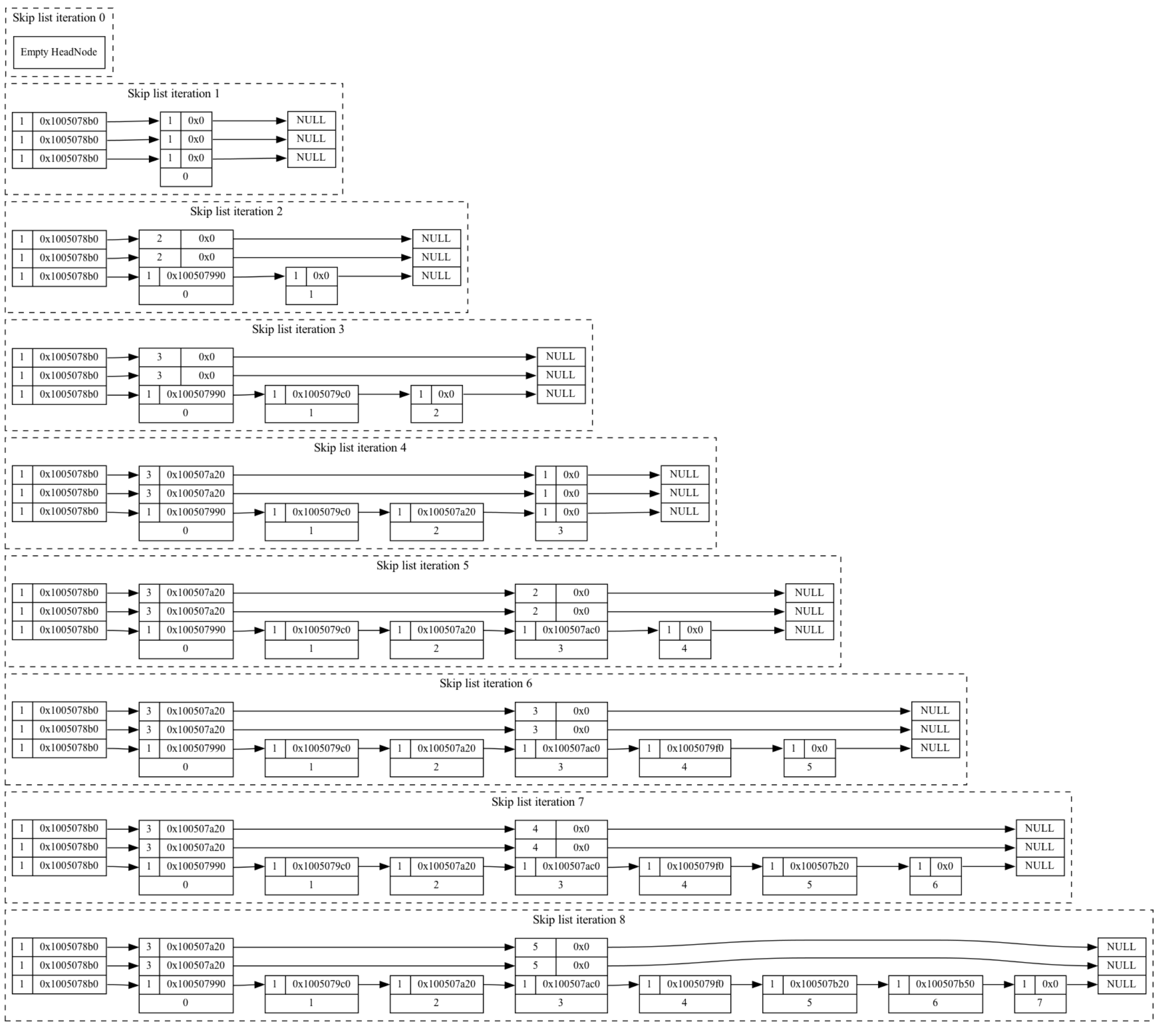
SkipList.

11.2 1:8, `p(0.125)`

There are very few coarser linked lists here. This is getting closer to a normal linked list with O(n) search behaviour.

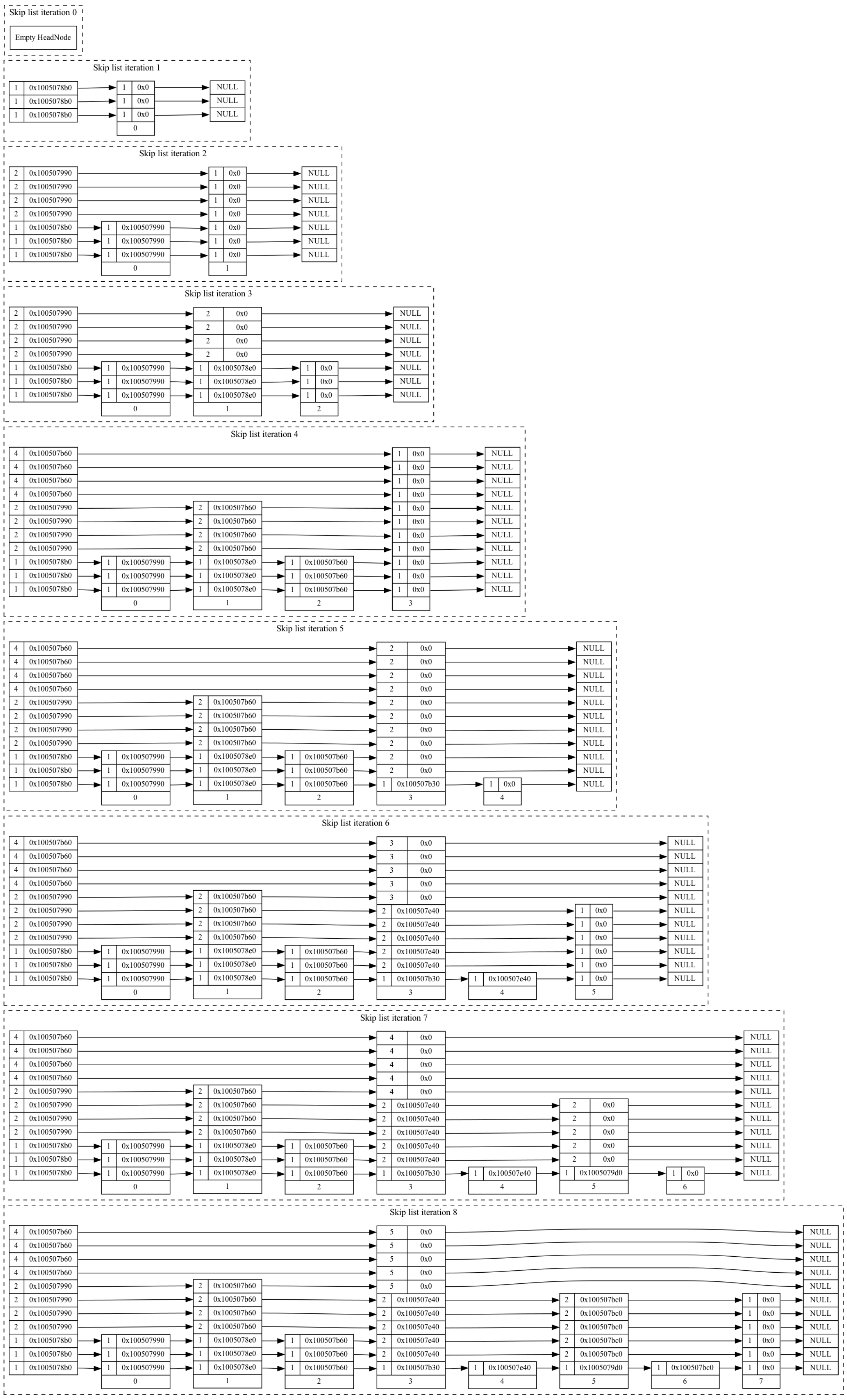


11.3 1:4, `p(0.25)`


SkipList.

11.4 3:4, `p(0.75)`

Each node now participates in many more linked lists. This may/may not improve search performance but it will certainly increase size requirements.


SkipList.

For the effect of a biased coin on time/space performance see *Effect of a Biased Coin*

CHAPTER

TWELVE

SOME NOTES ON TESTING

12.1 Test Code

There are several areas of test code:

- C++ functional tests.
- C++ performance tests.
- Python functional tests.

12.1.1 C++ Functional Tests.

The functional tests are run with a debug build of the C++ binary. In a debug build every operation on a skip list is checked before and afterwards with an integrity check. These integrity checks are thorough but expensive, see `template <typename T> IntegrityCheck HeadNode<T>::lacksIntegrity() const` in *HeadNode.h* for what checks are done.

Running these tests is as follows:

```
$ cd src/cpp
$ make debug
$ ./SkipList_D.exe
Running skip list tests...
                 test_very_simple_insert(): PASS
                      test_simple_insert(): PASS
             test_insert_and_remove_same(): PASS
             test_insert_remove_multiple(): PASS
                       test_ins_rem_rand(): PASS
              test_insert_n_numbers_same(): PASS
                                 test_at(): PASS
...
                doc_insert_remove_repeat(): PASS
Final result: PASS
Exec time: 19.5811 (s)
Bye, bye!
```

12.1.2 C++ Performance Tests.

Running the release version of the C++ code introduces some performance tests where the results are sent to stdout.

```
$ cd src/cpp
$ make release
$ ./SkipList_R.exe
Running skip list tests...
                 test_very_simple_insert(): PASS
                      test_simple_insert(): PASS
             test_insert_and_remove_same(): PASS
             test_insert_remove_multiple(): PASS
                       test_ins_rem_rand(): PASS
              test_insert_n_numbers_same(): PASS
                                 test_at(): PASS
               perf_single_insert_remove(): 451.554 (ms) rate x.xe+06 /s
                                                    ...
            perf_roll_med_odd_index_wins(): vectors length:  1000000 window width: 524288 time: x.x (ms)
                            perf_size_of(): size_of(      1):      216 bytes ratio:     216 /sizeof(T):       x
...
                doc_insert_remove_repeat(): PASS
Final result: PASS
Exec time: 127.5 (s)
Bye, bye!
```

This data is scraped from stdout to describe the *Skip List Performance*

12.1.3 Python Functional Tests.

There are some tests for the Python API in *tests/*. One of the reasons for creating a Python API was to use the most excellent Hypothesis to test the underlying C++ code.

12.2 Testing a Probabilistic Structure

One problem with probabilistic data structures like a skip list is deterministic testing. Along the way you discover particular corner cases that are worthy of testing the implementation against. Each case is completely prescribed by the order of insertion and the order of head/tail results generated by the virtual coin toss function. Whilst the former can be controlled by the test case the latter depends on the state of the pseudo random number generator and there lies the problem.

In this project this is solved in a novel way by seeding the random number generator with a value and seeing what head/tail sequence results. The random number generator is then brute forced with different seeds until every possible head/tail sequence of a certain length is obtained. A map is constructed that has `{sequence : seed, ...}` which can be use by any test to find the seed that will create the desired sequence. The random number generator is seeded with that value and the test is now deterministic.

The function that creates a dictionary is `find_seeds_for_sequences()` in *tests/unit/seed_tree.py*. It is tested in *tests/unit/test_seed_tree.py*. You can see it at work in *tests/unit/test_cSkipList.py* where `SEED_DICT` is created using the C `rand()` function exposed through `cSkipList.seed_rand()` and `cSkipList.toss_coin()`. There are various tests in there that specify a sequence of coin tosses and inject nodes that require that sequence.

CHAPTER

THIRTEEN

SKIP LIST API

13.1 C++ API

This describes the C++ API to a skip list with some algorithmic details of their implementation. Here is a visualisation of a skip list:

```
| 5 E |-------------------------------------->| 4 0 |----------------------------->|␣
↪NULL |
| 1 A |->| 2 C |---------->| 2 E |---------->| 2 G |---------->| 2 0 |---------->|␣
↪NULL |
| 1 A |->| 1 B |->| 1 C |->| 1 D |->| 1 E |->| 1 F |->| 1 G |->| 1 H |->| 1 0 |->|␣
↪NULL |
| HED |  |  A  |  |  B  |  |  C  |  |  D  |  |  E  |  |  F  |  |  G  |  |  H  |
```

In these descriptions:

- 'right' and 'left' is used to mean move to a higher/lower ordinal node.
- 'up' and 'down' means to move to a coarser/finer grained list, 'top' is the highest, 'bottom' is the level 0.

13.1.1 Example in C++

```cpp
#include "SkipList.h"

OrderedStructs::SkipList::HeadNode<double> sl;

sl.insert(42.0);
sl.insert(21.0);
sl.insert(84.0);

sl.has(42.0) // true
sl.size()    // 3
sl.at(1)     // 42.0, throws SkipList::IndexError if index out of range

sl.remove(21.0); // throws SkipList::ValueError if value not present

sl.size()    // 2
sl.at(1)     // 84.0
```

13.1.2 Constructors

There is only one constructor to a `HeadNode` and that takes no arguments.

13.1.3 `HeadNode::insert(const T &val)`

Declaration: `void HeadNode::insert(const T &value);`

Inserts a copy of `value` such that the previous value, if present, is <= `value` and the next value, if present, is > `value`.

Algorithm

Finding the place to insert a node first follows the `has(T &val)` algorithm to find the place in the skip list to create a new node. Inserts of duplicate values are made after any existing duplicate values. All nodes are inserted at level 0 even if the insertion point can be seen at a higher level. The search for an insertion location creates a recursion stack that, when unwound, updates the traversed nodes `{width, Node<T>*}` data.

Once an insert position is found a Node is created whose height is determined by repeatedly tossing a virtual coin until 'tails' is found. This node initially has all node references to be to itself (this), and the widths set to 1 for level 0 and 0 for the remaining levels, they will be used to sum the widths at one level down. On recursion ('left') each node adds its width to the new node at the level above the current level. On moving up a level the current node swaps its width and node pointer with the new node at that new level.

13.1.4 `HeadNode::remove(const T &val)`

Declaration: `void HeadNode::remove(const T &value);`

Removes the value from the skip list. This will throw an `ValueError` if the value is not present.

If there are duplicate values the last one is removed first, this is for symmetry with `insert()`. Essentially this is the same as `insert()` but once the node is found the `insert()` updating algorithm is reversed and the node deleted.

13.1.5 `HeadNode::has(const T &val) const;`

Declaration: `bool HeadNode::has(const T &value) const;`

This returns `true` or `false` if the skip list has the value val. This is O(log(n)) for well formed skip lists.

Algorithm

Starting at the highest possible level search rightwards until a larger value is encountered, then drop down. At level 0 return true if the Node value is the supplied value.

13.1.6 `HeadNode::at(size_t index) const;`

Declaration: `const T& HeadNode::at(size_t index) const;`

This returns the value of type `T` at the given index. This will throw an `IndexError` if the index is >= size of the skip list. This is O(log(n)) for well formed skip lists.

Algorithm

The algorithm is similar to `has(T &val)` but the search moves rightwards if the width is less than the index and decrementing the index by the width. If progress can not be made to the right, drop down a level. If the index is 0 return the node value.

13.1.7 `HeadNode::at(size_t index, size_t count, std::vector<T> &dest) const;`

Declaration: `void HeadNode::at(size_t index, size_t count, std::vector<T> &dest) const;`

This loads a vector of type `T` with `count` values starting at the given index. This will throw an `IndexError` if the index + count is >= size of the skip list. This is O(count * log(n)) for well formed skip lists.

Algorithm

The inital location follows the algorithm of `at(size_t index) const;` then sequential nodes are included.

13.1.8 `HeadNode::index(const T &value) const;`

Declaration: `size_t HeadNode<T>::index(const T& value) const`

Returns the index of the first occurence of the value. This will throw a `ValueError` if not found. This will throw a `FailedComparison` if the value is not comparable. This is O(log(n)) for well formed skip lists.

`at(index(value))` is always true if `value` is in the skip list. If there are no duplicate values `index(at(i))` is true for all indices.

13.1.9 `HeadNode::size() const`

Declaration: `size_t HeadNode::size() const;`

Returns the number of items in the skip list.

13.1.10 Specialised APIs

`HeadNode::height() const`

Declaration: `size_t HeadNode::height() const;`

Returns the number of linked lists that make up the skip list (minimum 1).

`HeadNode::height(size_t idx) const`

Declaration: `size_t HeadNode::height(size_t idx) const;`

Returns the number of linked lists that a particular node at index `idx` participates in. Will throw a IndexError if idx is >= size of the skip list (minimum 1).

`HeadNode::width(size_t idx, size_t level) const`

Declaration: `size_t HeadNode::width(size_t idx, size_t level) const;`

Returns the number of nodes a node reference skips for the node at `idx` and the level at `level`. Will throw a IndexError if idx is >= size of the skip list or `level` >= the node height (minimum 1).

`HeadNode::dotFile(std::ostream &os) const`

Declaration: `void HeadNode::dotFile(std::ostream &os) const;`

Writes a representation of the current state of the skip is as a fragment of a DOT file to the stream `os` as a `subgraph`.

Internally the `HeadNode` keeps a count of how many times this has been called. The first time this function is called writes the preamble of DOT file as well as the subgraph.

Writing to `os` before or between `dotFile()` calls may produce undefined DOT format files.

Here is an example of using this method:

```cpp
#include "SkipList.h"

OrderedStructs::SkipList::HeadNode<double> sl;
sl.insert(42.0);
sl.insert(84.0);
sl.insert(21.0);
sl.insert(100.0);
sl.insert(12.0);
sl.dotFile(std::cout);
sl.dotFileFinalise(std::cout);
```

There are examples of code that performs this in *src/cpp/main.cpp*

There are examples of the result in *Visualising a Skip List*

This method requires the macro `INCLUDE_METHODS_THAT_USE_STREAMS` to be defined.

`HeadNode::dotFileFinalise(std::ostream &os) const`

Declaration: `void HeadNode::dotFileFinalise(std::ostream &os) const;`

Writing to `os` after `dotFileFinalise()` produces may produce undefined DOT format files.

This method requires the macro `INCLUDE_METHODS_THAT_USE_STREAMS` to be defined.

`HeadNode::lacksIntegrity() const;`

Declaration: `IntegrityCheck HeadNode::lacksIntegrity() const;`

Returns non-zero if the integrity of this data structure is compromised. The return values are defined in *src/cpp/IntegrityEnums.h* This is a thorough but expensive check!

`HeadNode::size_of() const`

Declaration: `size_t HeadNode::size_of() const;`

Estimate of the number of bytes used by the skip list. This will not include any dynamically allocated memory such as used by `std::string`.

13.2 Python API

The Python API closely follows the C++ API.

13.2.1 Example in Python

```python
import cSkipList

sl = cSkipList.PySkipList(float)

sl.insert(42.0)
sl.insert(21.0)
sl.insert(84.0)

sl.has(42.0) # True
sl.size()    # 3
sl.at(1)     # 42.0

sl.has(42.0) # True
sl.size()    # 3
sl.at(1)     # 42.0, raises IndexError if index out of range

sl.remove(21.0); # raises ValueError if value not present

sl.size()    # 2
sl.at(1)     # 84.0
```

13.2.2 Module `cSkipList`

The module contains the following attributes:

Attribute	Description
`cSkipList.__build_target__`	The Python version this module is built for. Example: `'3.5.1'`
`cSkipList.__build_time__`	The date and time the module was built. Example: `'Jul 14 2016 11:35:03'`
`cSkipList.__build_type__`	The build type, either `'debug'` or `'release'`.
`cSkipList.__version__`	The version of the build. Example `'0.1.0'`.

The module contains the following module level functions:

Function	Description
`toss_coin()`	Returns the result of a virtual coin toss.
`seed_rand(long)`	Seeds the random number generator with a long integer.
`min_long()`	The minimum storable value of a `PySkipList(long)`.
`max_long()`	The maximum storable value of a `PySkipList(long)`.

13.2.3 Class `cSkipList.PySkipList`

13.2.4 Constructor

The constructor takes a Python type. The following are valid:

```
import cSkipList

sl = cSkipList.PySkipList(int) # Python 3, for Python 2 use PySkipList(long)
sl = cSkipList.PySkipList(float)
sl = cSkipList.PySkipList(bytes) # In Python 2 PySkipList(str) also works
```

13.2.5 `PySkipList.has(val)`

This returns `True` or `False` if the skip list has the value val. Will raise a `TypeError` if `val` is not the same type as the skip list was constructed with. This is O(log(n)) for well formed skip lists.

13.2.6 `PySkipList.at(index)`

This returns the value at the given index which must be of type long. Negative values of the index are dealt with Pythonically. This will raise an `IndexError` if the index is >= size of the skip list. This is O(log(n)) for well formed skip lists.

13.2.7 `PySkipList.at_seq(index, count)`

This returns a tuple of `count` values starting at the given `index` which must be of type long. Negative values of the index are dealt with Pythonically. `count` must be positive. This tuple contains a copy of the data in the skip list. This will raise an `IndexError` if the `index` + `count` is >= size of the skip list. This is O(count * log(n)) for well formed skip lists.

13.2.8 `PySkipList.index(value)`

Returns the index of the first occurence of the value. This will throw a `ValueError` if not found or the value is not comparable. This is O(log(n)) for well formed skip lists.

13.2.9 `PySkipList.size()`

Returns the number of items in the skip list.

13.2.10 `PySkipList.insert(value)`

Inserts a copy of `value` such that the previous value, if present, is <= `value` and the next value, if present, is > `value`. Will raise a `TypeError` if `val` is not the same type as the skip list was constructed with.

In the case of a `PySkipList(long)` if the value < `min_long()` or > `max_long()` an `OverflowError` will be raised.

13.2.11 `PySkipList.remove(value)`

Removes the value from the skip list. This will raise an `ValueError` if the value is not present. Will raise a `TypeError` if `val` is not the same type as the skip list was constructed with.

If there are duplicate values the last one is removed first, this is for symmetry with `insert()`. Essentially this is the same as `insert()` but once the node is found the `insert()` updating algorithm is reversed and the node deleted.

In the case of a `PySkipList(long)` if the value < `min_long()` or > `max_long()` an `OverflowError` will be raised.

13.2.12 Specialised APIs

`PySkipList.height()`

Returns the number of linked lists that make up the skip list (minimum 1).

`PySkipList.node_height(index)`

Returns the number of linked lists that a particular node at index `index` participates in (minimum 1). Will raise a `IndexError` if idx is >= size of the skip list.

`PySkipList.width(index, level)`

Returns the number of nodes a node reference skips for the node at `index` and the level at `level`. Will raise an `IndexError` if `index` is >= size of the skip list or `level` >= the node height (minimum 1).

`PySkipList.dot_file()`

Returns a representation of the current state of the skip is as a bytes object that represents a DOT file.

This method requires the macro `INCLUDE_METHODS_THAT_USE_STREAMS` to be defined.

There are examples of the result in *Visualising a Skip List*

`PySkipList.lacks_integrity()`

Returns non-zero if the integrity of this data structure is compromised. The return values are defined in *src/cpp/IntegrityEnums.h* This is a thorough but expensive check!

CHAPTER

FOURTEEN

ORDEREDSTRUCTS PYTHON REFERENCE

14.1 orderedstructs.SkipList

SkipList - An implementation of a skip list for floats, longs, bytes and Python objects.

`orderedstructs.SkipList.`**`__delattr__`**(*self*, *name*, */*)

Implement delattr(self, name).

`orderedstructs.SkipList.`**`__dir__`**(*self*, */*)

Default dir() implementation.

`orderedstructs.SkipList.`**`__eq__`**(*self*, *value*, */*)

Return self==value.

`orderedstructs.SkipList.`**`__format__`**(*self*, *format_spec*, */*)

Default object formatter.

`orderedstructs.SkipList.`**`__ge__`**(*self*, *value*, */*)

Return self>=value.

`orderedstructs.SkipList.`**`__getattribute__`**(*self*, *name*, */*)

Return getattr(self, name).

`orderedstructs.SkipList.`**`__getstate__`**(*self*, */*)

Helper for pickle.

`orderedstructs.SkipList.`**`__gt__`**(*self*, *value*, */*)

Return self>value.

`orderedstructs.SkipList.`**`__hash__`**(*self*, */*)

Return hash(self).

`orderedstructs.SkipList.`**`__init__`**(*self*, */*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

`orderedstructs.SkipList.`**`__init_subclass__`**()

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

`orderedstructs.SkipList.`**`__le__`**(*self*, *value*, */*)

Return self<=value.

`orderedstructs.SkipList.`**`__len__`**(*self*, */*)

Return len(self).

`orderedstructs.SkipList.`**`__lt__`**(*self*, *value*, */*)

Return self<value.

`orderedstructs.SkipList.`**`__ne__`**`(`*self*, *value*, */*`)`

Return self!=value.

`orderedstructs.SkipList.`**`__new__`**`(`**args*, ***kwargs*`)`

Create and return a new object. See help(type) for accurate signature.

`orderedstructs.SkipList.`**`__reduce__`**`(`*self*, */*`)`

Helper for pickle.

`orderedstructs.SkipList.`**`__reduce_ex__`**`(`*self*, *protocol*, */*`)`

Helper for pickle.

`orderedstructs.SkipList.`**`__repr__`**`(`*self*, */*`)`

Return repr(self).

`orderedstructs.SkipList.`**`__setattr__`**`(`*self*, *name*, *value*, */*`)`

Implement setattr(self, name, value).

`orderedstructs.SkipList.`**`__sizeof__`**`()`

Return the size of the skiplist in bytes.

`orderedstructs.SkipList.`**`__str__`**`(`*self*, */*`)`

Return str(self).

`orderedstructs.SkipList.`**`__subclasshook__`**`()`

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

`orderedstructs.SkipList.`**`at`**`()`

Return the value at the given index.

`orderedstructs.SkipList.`**`at_seq`**`()`

Return the sequence of count values at the given index.

`orderedstructs.SkipList.`**`dot_file`**`()`

Returns a bytes object suitable for Graphviz processing of thecurrent state of the skip list. Use open(<name>, ‘wb’).write(self.dotFile()) for Graphviz.

`orderedstructs.SkipList.`**`has`**`()`

Return True if the value is in the skip list, False otherwise.

`orderedstructs.SkipList.`**`height`**`()`

Return the height of the skip list head node.

`orderedstructs.SkipList.`**`index`**`()`

Return the index of the given value. Will raise a ValueError if not found.

`orderedstructs.SkipList.`**`insert`**`()`

Insert the value into the skip list.

`orderedstructs.SkipList.`**`lacks_integrity`**`()`

Returns non-zero is the skip list faulty, zero if OK.

`orderedstructs.SkipList.`**`node_height`**`()`

Return the height of node at the given index.

`orderedstructs.SkipList.`**`node_width`**`()`

Return the width of node at the given index and level.

`orderedstructs.SkipList.`**`remove`**`()`

Remove the value from the skip list.

`orderedstructs.SkipList.`**`size`**`()`

Return the number of elements in the skip list.

`orderedstructs.SkipList.`**`size_of`**`()`

Return an estimate of the number of bytes of memory used by the skip list.

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

INDEX