Generic Programming Project - Skip List Container

Project ID: 24

Name	SRN
Madhav Jivrajani	PES1201800028
Samyak Sarnayak	PES1201801565
Rehan Vipin	PES1201801655

Introduction

A SkipList is a probabilistic data structure that achieves `O(log n)` complexity in operations such as insert, delete and find, with high probability. This structure typically has multiple levels, each level being a doubly linked list in itself and it *tries* to maintain the relation of each level having half the number of elements as the lower level, and this is where it gets the property of having `O(log n)` with high probability.

The skiplist is an *ordered* container, meaning that ordering of elements is according to a predicate, which by default is the *lesser than* relationship. It is this ordered nature that facilitates the mentioned complexity.

All operations start from the first node of the topmost level, and traversal happens left to right. When an element is encountered that violates the order according to the defined predicate, the traversal moves down one level, *skipping* all nodes in front of it in *that* level, because the desired node isn't going to be found on this level. By doing this, it trickles down to the lowermost level.

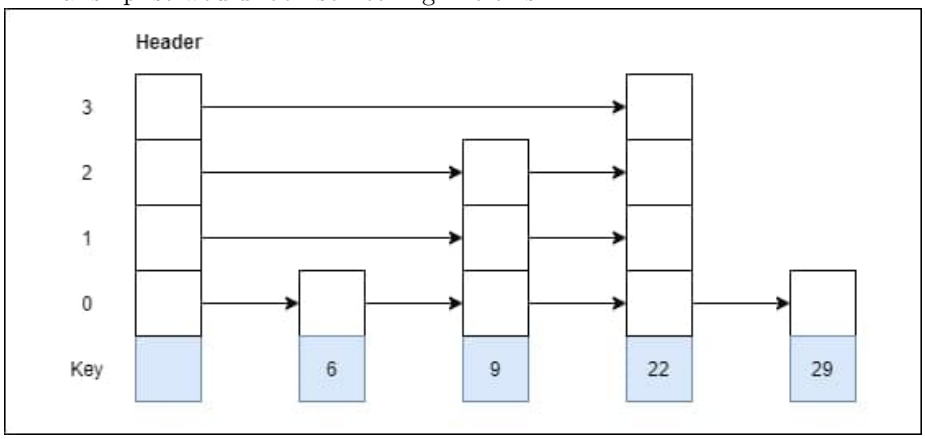
A skiplist typically handles only unique elements, similar to `std::set`, however it can be easily modified to handle non-unique elements as well. Our implementation handles non-unique elements by simply keeping a count of how many times that element has been encountered in total: +1 for `insert` and -1 for `delete`. Please see out `Algorithms` section for further information on this.

The skiplist container provides alternate implementations for two containers:

- Set variants – `skiplist.hpp`
- Map variants – `skiplist_map.hpp`

Use the map variant if you want to store key-value pairs, this is optimized for space as it stores only the keys at the upper levels and the value is only stored at the base level.

A final skiplist would look something like this:



Algorithms

`find`

- This finds an element and returns an `iterator` to it, if found.
- The search begins at the head of the topmost level of the skiplist, and traversals happen left to right.
- Whenever in a search on a level, the `next` element overshoots in terms of the ordering of elements, it moves `down` a level and search continues from there.
- This leveled approach helps skiplist achieve a O(log n) complexity with high probability.
- If an element is not found, an `iterator` to one beyond the last element of the bottom-most layer is returned.

`insert`

- Inserts an element into the container. The insert happens in an ordered way.
- First, a that element to be inserted is 'searched' (similar to `find`) for in the skiplist. It can already be present or be absent in the container.
  - If the element already exists in the container, then the `count` for that element is incremented, this is how non-unique elements are taken

care of.

  - If the element does not already exist in the container then the algorithm for insertion is applied. You first insert it in the list at the lowermost level, and then you "flip a coin" or generate a probability or a random number in the range `[0, 1]`. If this is *greater* than `0.5` then you keep inserting this element on upper levels with the element in the lowermost list as the *base* to all these elements.
    * See `visualize` for more clarity on this multi level insertion.

- If the container was empty to begin with, the element is inserted and the probabilistic/randomised inserts on multiple levels as mentioned above, is done again.

- The ordering of elements is by default based on the `less` relationship, but this can be a user defined predicate as well.

`erase`

- This removes an element by value. The iterator is assumed to be valid but cannot assume that the element itself exists.

- The general idea is to
  - Find the element
  - Decrement its `counter`
  - If the `counter` is now `0`, actually delete the element

- Finding the element uses an approach similar to `find`. If found, go up all levels and delete that element if present. If not found then stop there.

- This function is overloaded, elements can be deleted by value or by passing an `iterator` to the element that should be deleted.

Iterators

- Bidirectional iterator supported

- Iterator holds on to its location on de-referencing
- Can move forward and backwards

Forward iterators

- `begin` and `end`
  - These are *forward* iterators.
  - On de-referencing, they do not change their position.
  - `rvalue` de-reference is restricted because the container is ordered.
  - `begin` points to the head of the base list.
  - `end` points to one beyond the last element of the base list.
- `cbegin` and `cend`
  - These are constant forward iterators.
  - On de-referencing , they do not change their value.
  - On de-referencing, a constant reference is returned.
  - `cbegin` and `cend` are similar to `begin` and `end`.

Reverse iterators

- `rbegin` and `rend`
  - These are *backward* iterators.
  - They too do not change their position on de-referencing.
  - `rbegin` points to the last node of the base list and iterats in the reverse direction.
  - `rend` points to one before the head node of the base list.
- `crbegin` and `crend`
  - These are the constant versions of the the reverse iterators.

Benchmarks

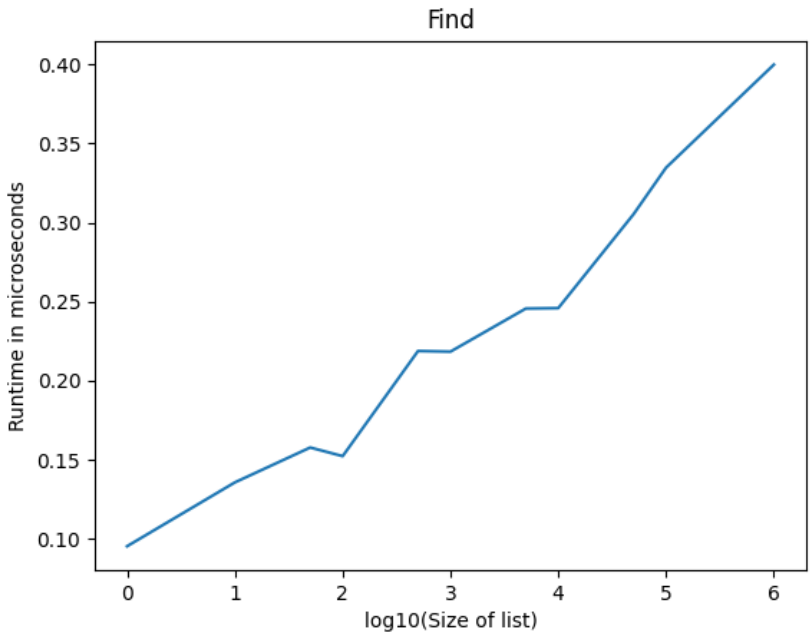
The operations were benchmarked with various input sizes to ensure that the expected performance characteristics (logarithmic scaling) was achieved.
The benchmarks were performed for the following sizes of skiplist: `1`, `10`, `50`, `100`, `500`, `1000`, `5000`, `10,000`, `50,000`, `100,000` and `1,000,000`. For each of the sizes, the operations were performed `100,000` times. The exection time of each operation was measured and averaged over all the `100,000` runs to get a consistent value.

A plot of the execution time vs `log10` of the size of the list was made. A straight line (close to `x = y`) in this plot means that the time complexity is close to `O(log n)` where `n` is the size of the list.

All of the operations were performed using random numbers from a discrete random distribution in the range `[0, n-1]`.

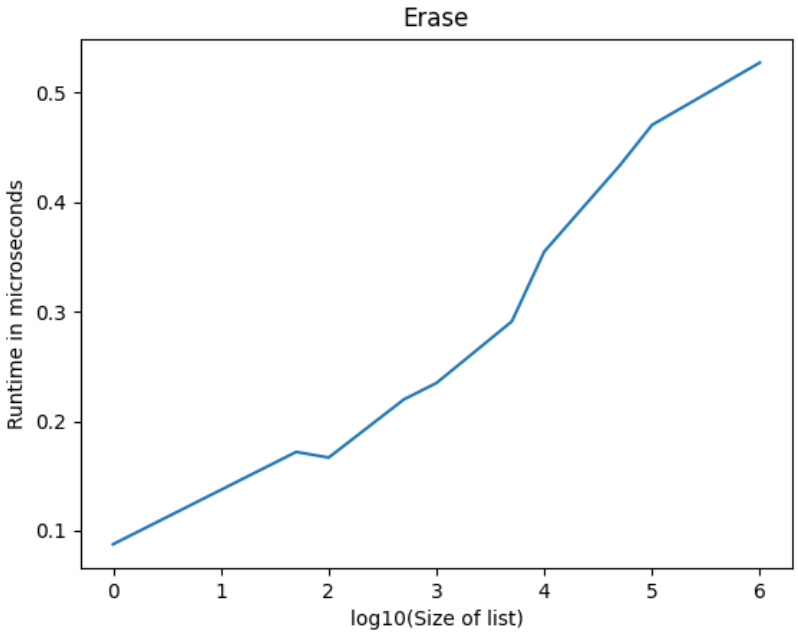
The plots were made using python's matplotlib library.

`find`



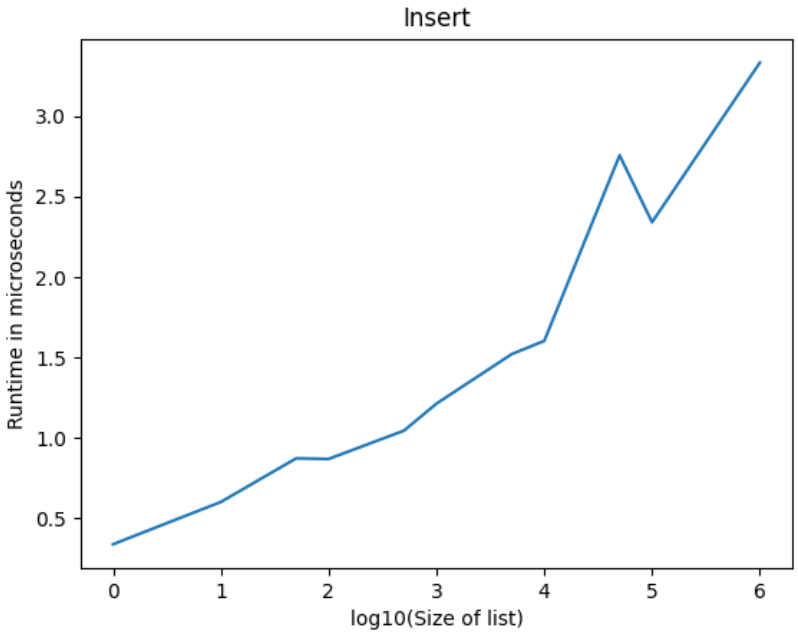
The above graph shows the `O(log n)` (empirical) time complexity of `find` (note that the x-axis is logarithmic). Further, all of the operations took less than `0.5` microseconds to execute even in a list with 1 million elements.

`erase`



The above graph shows the `O(log n)` (empirical) time complexity of `erase` (note that the x-axis is logarithmic). Further, all of the operations took less than `0.6` microseconds to execute even in a list with 1 million elements.

`insert`



The above graph shows the (close to) `O(log n)` (empirical) time complexity of `insert` (note that the x-axis is logarithmic). Although the execution time of insert is much higher than `find` and `erase`, it shows the same logarithmic scaling with input size. Further, all of the operations took less than `4` microseconds to execute even in a list with 1 million elements.

Examples and use cases

Skiplists can be used as an ordered collection where search, insert and delete need to be fast. Similar to a multi-set, skiplists can also have duplicate elements.

The element to be added to a skiplist must have the following interfaces

- Default constructor
- `operator<` or a functor that provides this functionality
- (optional, required for visualization) `operator<<` with `std::ostream`

Dictionary

In this example, we show how the skiplist can used to create a dictionary, similar to a multi-set.

The element for this dictionary must have:

- Default constructor
- `operator<` or a functor that provides this functionality

Optionally, the element can have these interfaces for ease of use:

- Single value constructor - this constructor must take just the key. This is used for finding an element based on key.
- `operator<<` with `std::ostream` - for visualization, not required.

With the above interfaces implemented on a class named `DictKey`, the skiplist dictionary can be used as follows.

```
#include <skiplist.hpp>
skiplist<DictKey> aDict;

// inserting new key-value pairs
aDict.insert({10, "hello"});
aDict.insert({20, "jello"});
aDict.insert({20, "yello"});
aDict.insert({40, "fello"});

// iterating through all elements
std::cout << "SKIPLIST:\n";
for(auto e: aDict)
std::cout << e << "\n";

// find
// note the usage of just the key `20`
// the single value constructor works here
auto iter = aDict.find(20);
std::cout << "Found: " << *iter << "\n";

// incrementing iterator
std::cout << "Next: " << *++iter << "\n";

// count
std::cout << "Count: " << aDict.count(20) << "\n";
```

A map type implementation of the same:

```
#include <skiplist_map.hpp>
skiplist<int, std::string> aDict;

aDict.insert(10, "hello");
aDict.insert(20, "jello");
aDict.insert(20, "yello");
aDict.insert(40, "fello");

std::cout << aDict;
```

Use with `<algorithm>`

The bidirectional iterators provided can be used with the `<algorithm>` library.

Example of using `equal` from `<algorithm>`:

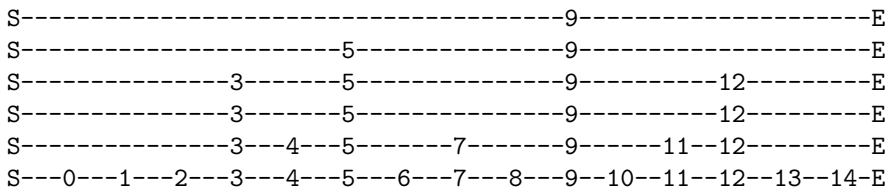
```
#include <skiplist.hpp>
skiplist<int> s1;
skiplist<int> s2;

for(int i = 0; i < 10; ++i) {
    s1.insert(i);
    s2.insert(i);
}

// equal
std::cout << std::boolalpha;
std::cout << std::equal(begin(s1), end(s1), begin(s2)) << "\n";

// for_each
std::for_each(begin(s1), end(s1), [](int e) { std::cout << e << "\t"; });
```

Visualizer

When the skiplist is printed using `cout`, the overloaded `operator<<` provides a visualization of all levels of the skiplist.

Since the levels are decided pseudo-randomly, the visualization will be different on each run even with the same elements.

Example of a visualization of a skiplist with 0 to 9 elements.


S----------9----------E
S----------5----------9----------E
S----------3-------5----------9----------12---------E
S----------3-------5----------9----------12---------E
S----------3---4---5-------7-------9------11--12---------E
S---0---1---2---3---4---5---6---7---8---9--10--11--12--13--14-E