Problem 7: Skip Lists.

There are two subtasks here.

(a) Let's implement the skip list and compare it against any reputable built-in ordered map. Please design your own experiments. What are some questions you would like to know and how would you experimentally answer them?

Answer to 7.a:

1 Introduction:

This report examines the performance of a C++ skip list implementation through a series of experiments conducted on a MacOS system (Apple M3 chip). The experiments cover:

- The comparison of two methods for simulating coin tosses (used in random level generation).
- An investigation into how varying the maximum height (or maximum level) affects the performance of skip lists.
- Performance benchmarks comparing skip list operations (insertion and search) with the standard C++ ordered set.

The following sections describe the experimental setup, present detailed results, and discuss the observed trends.

2 Experimental Setup

System Configuration:

- **Processor:** Apple M3 (8-core: 4 performance cores up to 4.05 GHz and 4 efficiency cores up to 2.75 GHz).
- **Cache Hierarchy:**
  - **L1 Cache:** Performance cores: 192 KB (instruction) and 128 KB (data); Efficiency cores: 128 KB (instruction) and 64 KB (data).
  - **L2 Cache:** 16 MiB (shared among performance cores for the M3/M3 Pro) and 32 MiB for the M3 Max.
  - **System Level Cache (SLC):** 8 MiB integrated; extrapolated values for M3 Pro and M3 Max are approximately 12 MiB and 48 MiB, respectively.
- **Memory:** LPDDR5-6400 with a bandwidth of 100 GB/s.

Compilation: Benchmarks were compiled with `-O3` optimization using `clang 19.1.6`.

3 Coin Toss Methods for Random Level Generation:

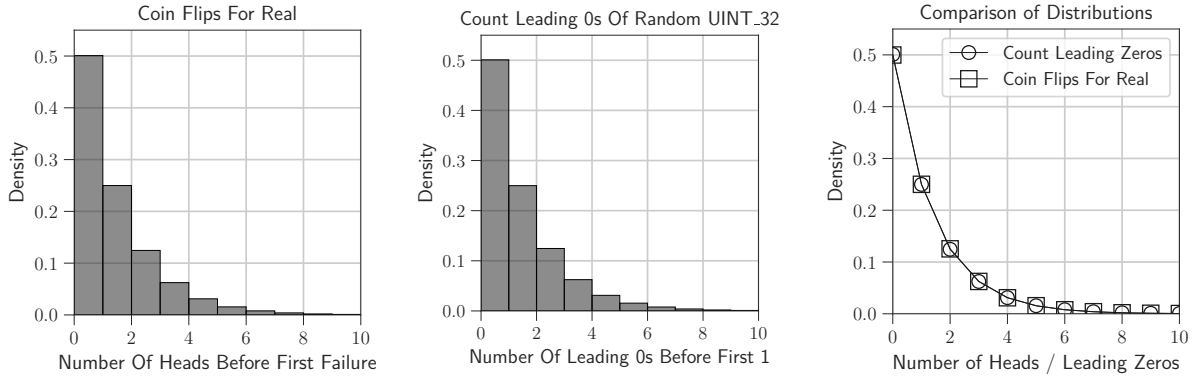
The skip list requires random decisions (coin tosses) to determine the height of each node. Two methods were benchmarked:

1. **Coin Flip For Real:** Simulates each coin toss individually.
2. **Count Leading 0s:** Uses a fast bit-level method by counting the number of leading zeros in a random 32-bit integer.

3.1 Benchmark Results

Benchmark	Time (ns)	Iterations
Coin Flip For Real	25.7	26848418
Coin Flip Count Leading 0s	2.45	292785351

Table 1: The *Count Leading 0s* method outperforms the standard coin-flip simulation by a large margin.



Coin Flip For Real | Count Leading 0s | Distribution Comparison

Figure 1: Visual comparison of the two coin toss methodologies.

From Figure 1 we can see that the simulation of coin flips by counting leading zeros is comparable to an actual coin flip simulation, but with significantly better performance.

4 Impact of Varying Maximum Height

The maximum level (height) of a skip list affects the balance between search speed and memory overhead. In the following experiments, the maximum level was varied while measuring CPU time and memory usage for both insertion and search operations.

4.1 Experimental Data

Skip List Insertion		
Max Level	CPU Time (ms)	Memory Usage (KB)
2	11000.61	4224
4	2363.20	688
6	676.20	2368
8	117.08	496
10	33.78	3040
12	25.01	2832
14	24.41	2832
16	22.37	2816
18	23.28	2032
20	23.87	3136
22	23.38	3424
24	23.86	2816
26	22.76	3056
28	24.80	2896
30	24.81	2816
32	25.10	3248

Table 2: Skip List Insertion Performance vs. Maximum Level

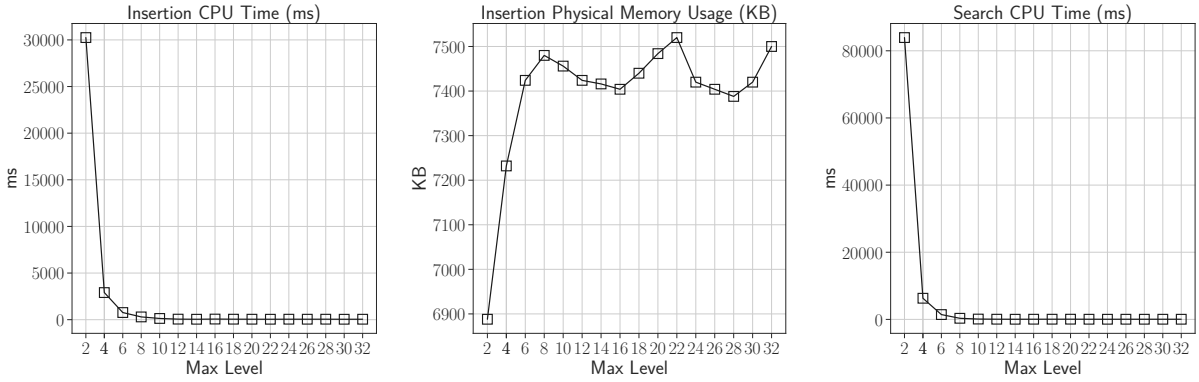
Skip List Search		
Max Level	CPU Time (ms)	Memory Usage (KB)
2	23331.85	448
4	5396.31	352
6	914.18	2592
8	109.63	2864
10	39.30	2800
12	26.00	2048
14	24.83	3056
16	23.67	3056
18	22.98	2032
20	22.92	3296
22	22.60	3056
24	25.35	3040
26	22.53	3056
28	24.45	3040
30	23.37	2816
32	23.22	2752

Table 3: Skip List Search Performance vs. Maximum Level

4.2 Observations

- Lower maximum levels (e.g., 2 or 4) lead to significantly higher CPU times.
- The memory usage seems to have outliers for insertion.
- Increasing the maximum level reduces CPU time until the benefit plateaus (diminishing returns).
- Both insertion and search operations follow similar trends with respect to the maximum level.

It is evident that the benefit in performance stops approximately around max level 12-14. Therefore for brevity, the next benchmarks all use a skip list of max level 20.



Insertion CPU Time | Insertion Memory Usage | Search CPU Time

Figure 2: Performance of skip list operations as the maximum level varies.

5 Skip List vs. Ordered Map Performance

This section presents the performance comparison between Skip List (Max Level 20) and Ordered Map in terms of insertion and search operations.

Table 4: Skip List Insertion Performance

n	CPU Time (ms)
8	1.14E-04
16	1.76E-04
32	5.78E-04
64	1.20E-03
128	3.25E-03
256	6.63E-03
512	1.60E-02
1024	6.98E-02
2048	2.00E-01
4096	4.39E-01

Table 5: Ordered Map Insertion Performance

n	CPU Time (ms)
8	0.00E+00
16	0.00E+00
32	0.00E+00
64	1.32E+01
128	4.73E+00
256	1.25E+01
512	0.00E+00
1024	5.78E+00
2048	8.72E+00
4096	1.45E+01

Table 6: Skip List Search Performance

n	CPU Time (ms)
8	2.90E-05
16	8.50E-05
32	3.49E-04
64	6.10E-04
128	1.22E-03
256	2.70E-03
512	7.12E-03
1024	6.00E-02
2048	1.57E-01
4096	3.99E-01

Table 7: Ordered Map Search Performance

n	CPU Time (ms)
8	3.20E-05
16	9.40E-05
32	2.00E-04
64	4.30E-04
128	9.84E-04
256	2.25E-03
512	6.70E-03
1024	3.84E-02
2048	1.09E-01
4096	2.51E-01

6 Conclusion

The experiments demonstrate that:

- The **Count Leading 0s** method greatly improves the performance of random level generation compared to simulating individual coin flips.
- Skip lists exhibit competitive performance in both insertion and search operations when compared to an ordered map, although the performance trade-offs differ by operation.
- Adjusting the maximum level of the skip list can optimize performance—lower levels incur higher costs, while higher levels improve speed up to a point.

■

(b) Searching in a skip list typically starts from the top-left corner. In this way, a search takes $O(\log n)$ whp., where n is the number of keys stored in the skip list. If we keep bi-directional pointers for each vertical column, we will be able to go up and down any column freely. This allows us to start from the smallest key and work our way to the desired key.

You'll extend the skip list in the following way. Maintain a "pointer" to the smallest key. Make up/down pointers bi-directional. Design and implement a search algorithm that runs in $O(\log d)$ time, where d is the number of elements smaller than the key you're searching for.

Answer to 7.b:

■

■