Assignment 1 — Applied Algorithms, T. II/2024–25

Austin Jetrin Maddison 6481268

February 8, 2025

Problem 1. Las Vegas and Monte Carlo

a.i) We want to show the probablity of running time of Monte Carlo is atleast the worst running time which is $4f(n)$. We can use markov inqualities to bound it..

$$\mathbf{P}(X \leq \lambda) \leq \frac{E[X]}{\lambda}$$

$$\begin{aligned}\mathbf{P}(T(n) \leq 4f(n)) &\leq \frac{f(n)}{4f(n)} \\ &\leq \frac{1}{4}\end{aligned}$$

a.ii) The worse-case running time happens at most 1/4 which produces incorrect answers. We can get the complement of the last answer...

$$1 - \mathbf{P}(T(n) \leq 4f(n)) \leq 1 - \frac{1}{4} = \frac{3}{4}$$

b.i) The LV algorithm running time is described as the follwing. Each iteration requires running A to produce an answer then run C to check the answer. So the running time for each trial is...

$$\text{1 iteration running time of LV} = f(n) + g(n)$$

So the question is what is the expected iterations needed to run LV to get a correct answer. If p is the probablity of success then the expected $1/p$.

$$\text{Running time of LV} = \frac{1}{p}(f(n) + g(n))$$

Problem 2. Chernoff-Hoeffding With Bounds

2.1)

$$\Pr[X > (1+\beta)\mu] \leq \exp\left(-\frac{\beta^2}{2+\beta}\mu H\right)$$
$$\Pr[X > (1+\varepsilon)\mu H] \leq \exp\left(-\frac{\varepsilon^2}{2+\varepsilon}\mu H\right)$$

$$(1+\epsilon)\mu_H = (1+\beta)\mu$$
$$\frac{\mu_H}{\mu} = \frac{1+\epsilon}{1+\beta}$$

2.2)

$$\Pr[X > (1+\beta)\mu] \leq \exp\left(-\frac{\beta^2}{2+\beta}\mu H\right)$$
$$\Pr[X > (1+\varepsilon)\mu H] \leq \exp\left(-\frac{\varepsilon^2}{2+\varepsilon}\mu H\right)$$

$$(1+\epsilon)\mu_H = (1+\beta)\mu$$
$$\frac{\mu_H}{\mu} = \frac{1+\epsilon}{1+\beta}$$

2.3)

$$\Pr[X > (1+\beta)\mu] \leq \exp\left(-\frac{\beta^2}{2+\beta}\mu H\right)$$
$$\Pr[X > (1+\varepsilon)\mu H] \leq \exp\left(-\frac{\varepsilon^2}{2+\varepsilon}\mu H\right)$$

$$(1+\epsilon)\mu_H = (1+\beta)\mu$$
$$\frac{\mu_H}{\mu} = \frac{1+\epsilon}{1+\beta}$$

2.4)

$$\Pr[X > (1+\beta)\mu] \leq \exp\left(-\frac{\beta^2}{2+\beta}\mu H\right)$$
$$\Pr[X > (1+\varepsilon)\mu H] \leq \exp\left(-\frac{\varepsilon^2}{2+\varepsilon}\mu H\right)$$

$$(1+\epsilon)\mu_H = (1+\beta)\mu$$
$$\frac{\mu_H}{\mu} = \frac{1+\epsilon}{1+\beta}$$

Problem 3. Rescaling Trick

(Statement of problem goes here.)

Proof. (Type your proof here.) □

Problem 4. x^2 With π Degrees of Freedom

(Statement of problem goes here.)

Proof. (Type your proof here.) □

Problem 5. Simple Samplers.

(Statement of problem goes here.)

Proof. (Type your proof here.) □

Problem 6. Median of Means

(Statement of problem goes here.)

Proof. (Type your proof here.) □

Problem 7. Skip List

Experimental Setup

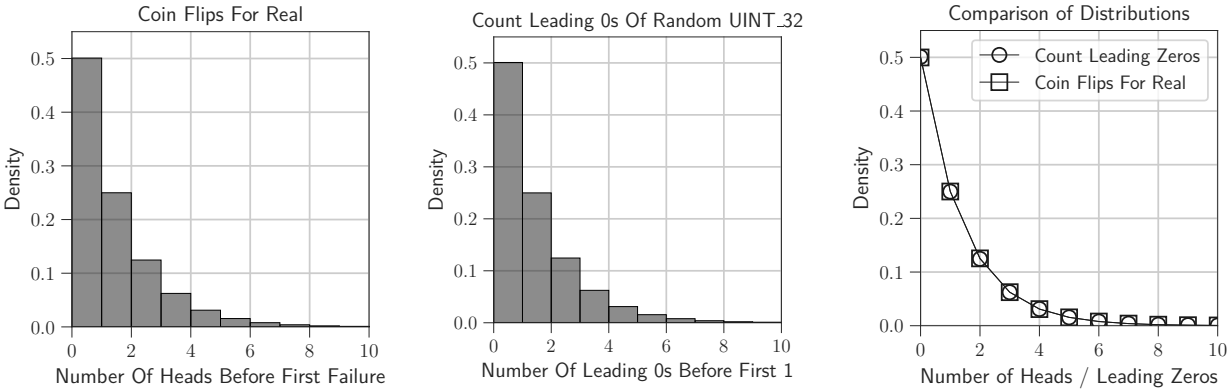
Conducted our benchmarks using Google Benchmark on a system with the following hardware specifications:

- **Processor**: 11th Gen Intel(R) Core(TM) i7-11370H 8-core CPU @ 3.3 GHz
- **Cache Hierarchy**:
  - L1 Data: 48 KiB per core (×4)
  - L1 Instruction: 32 KiB per core (×4)
  - L2 Unified: 1280 KiB per core (×4)
  - L3 Unified: 12,288 KiB (shared)

All benchmarks were compiled using `-Ox` optimization level with `MSVC` (version 19.42.34436) and executed in a single-threaded environment to minimize external interference. Memory usage was measured using the Windows API `GetProcessMemory()`.

Questions And Experiments

Q1.) Can we perform count coin tosses differently and is the alternative better?



Benchmark	Time (ns)	CPU (ns)	Iterations
Coin Flip For Real	29.8	29.3	22,400,000
Coin Flip Count Leading 0s	5.11	4.87	144,516,129

Table 1: Benchmark results comparing different coin flip implementations.

Q2.) How does varying max height change performance?
Q3.) Linked lists are known to be cache unfriendly, is there a way we can modify
Q4.) How does it perform against a reputable ordered map?

b.) Search algorithm of Skip List when start is at the bottom left corner in $O(\log(d))$ where d is the number of elements smaller than the key?

Problem 8. (a, b) tree. $(2, 3)$ tree.

a.)

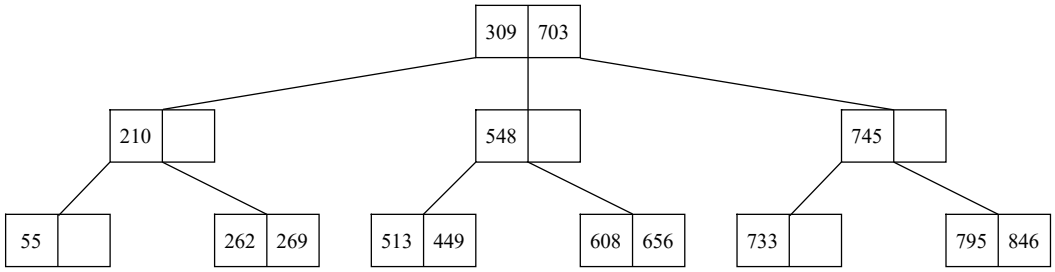


Figure 1: Keys $733, 703, 608, 846, 309, 269, 55, 745, 548, 449, 513, 210, 795, 656, 262$ inserted into a $(2, 3)$ tree.

b.)

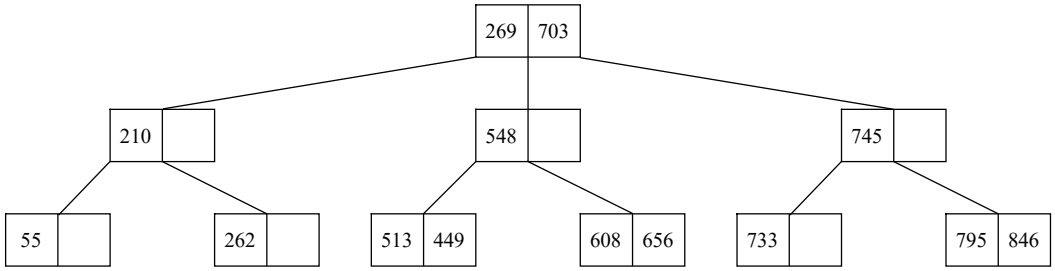


Figure 2: Key 309 removed from Figure 1 tree.

Problem 9. B-Tree Speed

Experimental Setup

Conducted our benchmarks using Google Benchmark on a system with the following hardware specifications:

- **Processor**: 11th Gen Intel(R) Core(TM) i7-11370H 8-core CPU @ 3.3 GHz
- **Cache Hierarchy**:
  - L1 Data: 48 KiB per core (×4)
  - L1 Instruction: 32 KiB per core (×4)
  - L2 Unified: 1280 KiB per core (×4)
  - L3 Unified: 12,288 KiB (shared)

All benchmarks were compiled using `-Ox` optimization level with `MSVC` (version 19.42.34436) and executed in a single-threaded environment to minimize external interference. Memory usage was measured using the Windows API `GetProcessMemory()`.

The B-Tree implementation utilized in this experiment is sourced from the repository by frozenca on GitHub [1].

Finding Optimal Parameter b Of B-Tree

First I carried out benchmarks to measure performance of tens of millions of random unique keys being inserted into the B-Tree while varying b parameter.

b	CPU Time (ms)	Virtual Memory Usage (KB)	Physical Memory Usage (KB)
2	59359.375	1294456	2544788
4	86968.750	539168	1063908
6	106484.375	375208	740108
8	123265.625	302968	596900
16	135156.250	202252	400248
32	144640.625	156320	309792
64	153750.000	135372	268680
128	163312.500	129468	256996
256	174062.500	135368	267676
512	186078.125	142356	280852
1024	200796.875	145848	285412
2048	221015.625	122476	246636

Table 2: Benchmark results for B-Tree insertion with 20×10^6 inserts and varying b from 2 to 2048.

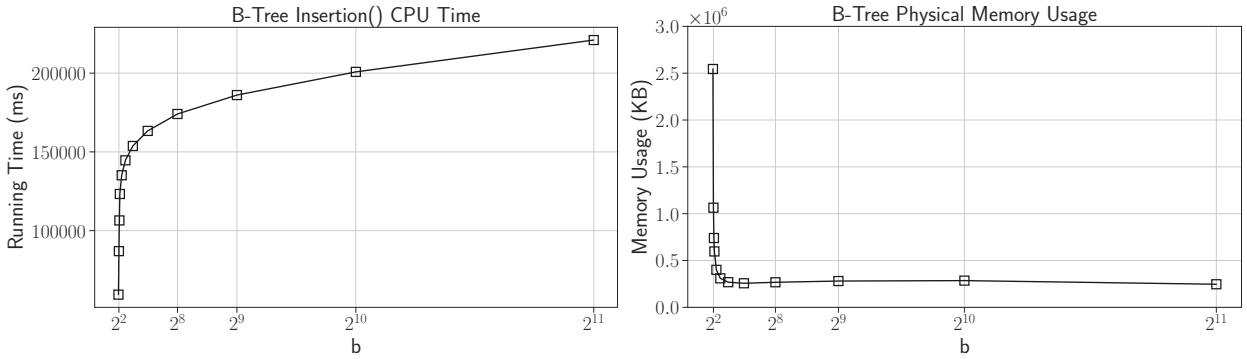


Figure 3: CPU Time and Physcal Memory Usage for B-Tree insertion with varying b values.

Notice that as the parameter of b the B-Tree increases, the physical memory usage decreases. This is strange, even when considering virtual memory (page file) usage. My theory is that increasing b makes the tree shallower and more compact. Since a shallower tree reduces the number of pointers and improves spatial locality, nodes are more likely to fit within a cache line, leading to better memory efficiency?

Anywhow... we want the b such that the it minimizes CPU time and memory used. They way I did it is to simply normalize CPU time and memory used and get closest b with to the smallest differnce.

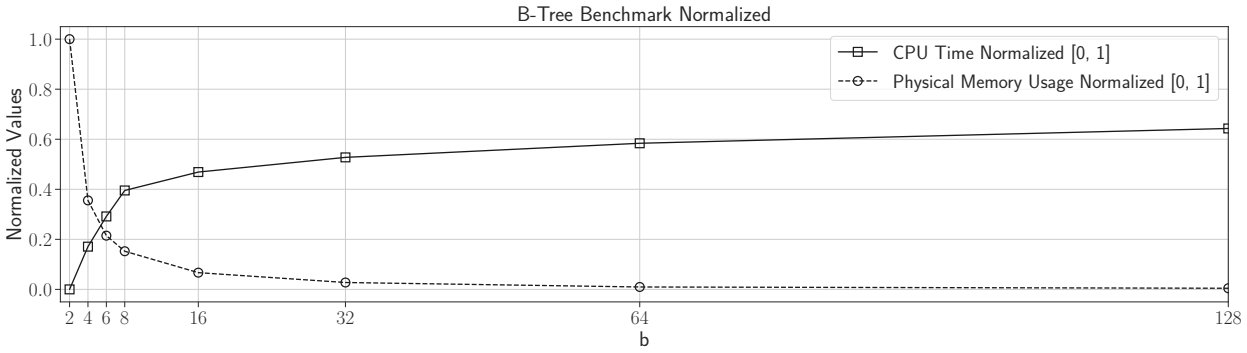


Figure 4: Normalized CPU Time and Physcal Memory Usage comparison for B-Tree insertion with varying b values.

Setting $b = 6$ minimizes both performance overhead and memory usage the most. However, for applications that prioritize memory efficiency over execution time, a larger value, such as $b = 16$, may be more optimal, as the performance trade-off becomes less significant.This version clarifies that $b = 6$ minimizes both performance overhead and memory usage while improving the flow of the second sentence. Let me know if you need further adjustments!

Comparing B-Tree and Builtin Ordered-Map Performance

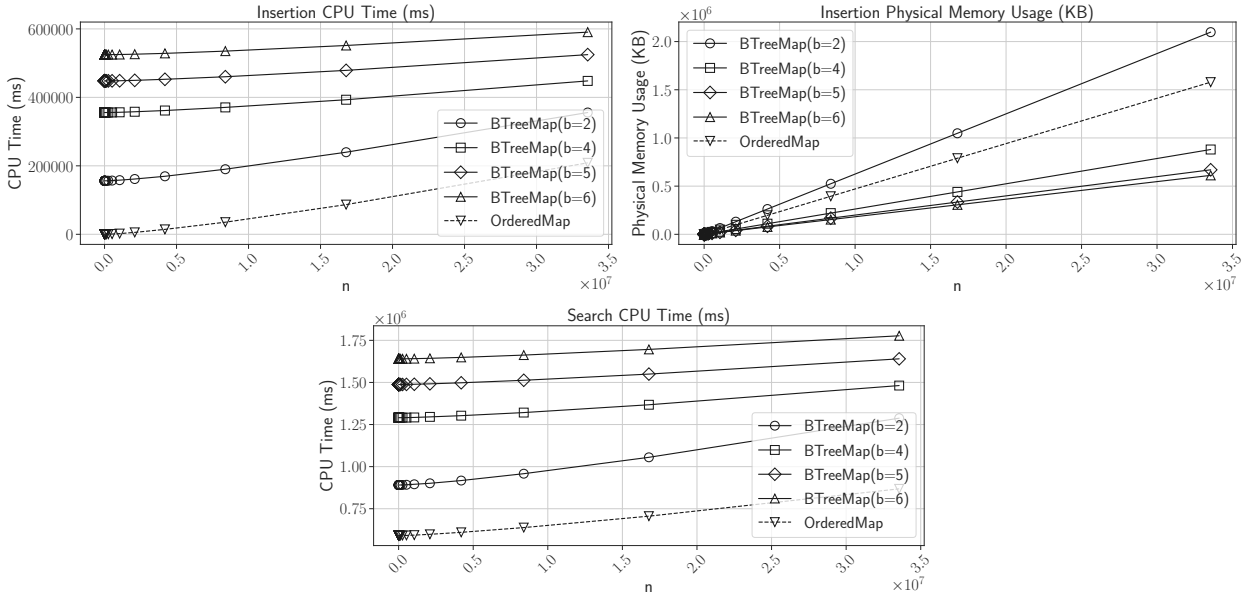


Figure 5: CPU Time and Physcal Memory Usage of B-Tree and C++ Builtin Ordered Map.

References

[1] B-Tree Implementation on GitHub. *GitHub Repository*. Available at: `https://github.com/frozenca/BTree`