Project 8: Skip List and B+ Tree Node Implementations

Group: P8G03

Authors

Canyu Liu
Xuanting Liu
Sihan Chen

Abstract

This report studies the skip list, a randomized layered data structure that supports search, insertion, and deletion efficiently in expectation. We first describe how a skip list organizes elements across multiple levels using forward pointers and randomized node heights. We then provide a formal expected-time analysis showing $O(\log n)$ expected running time for search, insertion, and deletion, and $O(n)$ expected space. Next, we integrate different node data structures (array, linked list, and a skip-list-based store) into a B+ tree implementation through a unified `NodeStoreOps` interface. We prove that when node-local search is $O(\log M)$, the overall B+ tree search time on n keys is $O(\log n)$, independent of the tree order M. Finally, we use our benchmark results and the generated test data to provide empirical evidence supporting the key steps of the proof.

---

1. Skip List Description

1.1 Data Structure Overview

A **skip list** maintains keys in sorted order using multiple levels of forward pointers:

- **Level 0** is a standard sorted linked list containing all keys.
- Each higher level $i > 0$ is a subsequence of level $i - 1$, providing "express lanes" that allow search to skip over many nodes.
- A **header/sentinel** node exists at all levels and serves as the entry point.

Each inserted key is assigned a random height (level count). A typical construction promotes a node from level i to $i + 1$ independently with probability p (commonly $p = 0.5$), so higher levels contain exponentially fewer nodes.

1.2 Node and List Representation

- A `SkipListNode` stores an integer `key` and a flexible array `forward[]`.
- The node's `level` is the length of `forward[]`. If a node has height h, it appears on levels $0..h - 1$.
- A `SkipList` stores:
  - `max_level` (upper bound on node height),
  - `p` (promotion probability),
  - `level` (current highest non-empty level),
  - `size` (number of keys),
  - `header` (sentinel node with height `max_level`).

1.3 Search Operation

Search starts from the highest active level. At each level:

1. **Move right** while the next node exists and its key is smaller than the target.
2. When moving right would overshoot (next key $\geq$ target), **drop down** one level.

After reaching level 0, the algorithm checks whether the next node equals the target.

1.3.1 Pseudocode — `SEARCH`

```
SEARCH(SL, key):
    x <- SL.header
    for i from SL.level-1 down to 0:
        while x.forward[i] != NIL and x.forward[i].key < key:
            x <- x.forward[i]
    x <- x.forward[0]
    if x != NIL and x.key == key:
        return FOUND
    else:
        return NOT_FOUND
```

1.4 Insertion Operation

Insertion performs a traversal like search but records predecessors `update[]`. If the key does not already exist, we generate a random node height and insert the new node by rewiring forward pointers.

1.4.1 Pseudocode — `RANDOM_LEVEL`

```
RANDOM_LEVEL(p, max_level):
    lvl <- 1
    while Random(0,1) < p and lvl < max_level:
        lvl <- lvl + 1
    return lvl
```

1.4.2 Pseudocode — `INSERT`

```
INSERT(SL, key):
    update[0..SL.max_level-1]
    x <- SL.header

    for i from SL.level-1 down to 0:
```

```
    while x.forward[i] != NIL and x.forward[i].key < key:
        x <- x.forward[i]
    update[i] <- x

x <- x.forward[0]
if x != NIL and x.key == key:
    return FAIL // duplicate key

lvl <- RANDOM_LEVEL(SL.p, SL.max_level)

if lvl > SL.level:
    for i from SL.level to lvl-1:
        update[i] <- SL.header
    SL.level <- lvl

new <- newNode(key, lvl)
for i from 0 to lvl-1:
    new.forward[i] <- update[i].forward[i]
    update[i].forward[i] <- new

SL.size <- SL.size + 1
return OK
```

1.5 Deletion Operation

Deletion records predecessors `update[]`. If the key exists, it is removed by bypassing it on each level where it appears. If the highest level becomes empty, the skip list's current height decreases.

1.5.1 Pseudocode — DELETE

```
DELETE(SL, key):
    update[0..SL.max_level-1]
    x <- SL.header

    for i from SL.level-1 down to 0:
        while x.forward[i] != NIL and x.forward[i].key < key:
            x <- x.forward[i]
        update[i] <- x

    x <- x.forward[0]
    if x == NIL or x.key != key:
        return FAIL

    for i from 0 to SL.level-1:
```

```
        if update[i].forward[i] == x:
            update[i].forward[i] <- x.forward[i]

    free(x)
    SL.size <- SL.size - 1

    while SL.level > 1 and SL.header.forward[SL.level-1] == NIL:
        SL.level <- SL.level - 1

    return OK
```

---

2. Formal Analysis of Skip List Operations

2.1 Expected Height

A node appears on level i with probability p^i. Therefore, the expected number of nodes on level i is:

$$\mathbb{E}[N_i] = np^i$$

The maximum non-empty level h is near the point where $np^h \approx 1$, implying:

$$h = O(\log_{1/p} n)$$

For a constant p (e.g., $p = 0.5$), $h = O(\log n)$.

2.2 Expected Search Time

A standard skip-list argument yields expected $O(1/p)$ horizontal steps per level. With $O(\log_{1/p} n)$ levels, expected search time is:

$$\mathbb{E}[T_{\text{search}}] = O\left(\frac{1}{p}\log_{1/p} n\right) = O(\log n) \quad \text{(constant } p\text{)}$$

2.3 Expected Insertion and Deletion Time

Insertion/deletion perform one search (to compute `update[]`) plus pointer updates on the node height. The expected node height is:

$$\mathbb{E}[\text{height}] = \sum_{k\geq 1} \Pr(\text{height} \geq k) = \sum_{k\geq 1} p^{k-1} = \frac{1}{1-p}$$

Thus insertion and deletion are expected $O(\log n)$.

2.4 Space Complexity

Expected total pointers are $n/(1-p)$, so expected space is $O(n)$.

---

3. B+ Tree with Skip-List Nodes

3.1 Node Storage Abstraction (`NodeStoreOps`)

We keep B+ tree logic unchanged and swap node implementations via `NodeStoreOps`:

- `lower_bound(store, key)` → first index with key ≥ target
- `key_at(store, idx)`, `val_at(store, idx)`
- `insert_at(store, idx, key, val)`, `erase_at(store, idx)`
- `split(left, right)` → move half entries to right and return separator key

We compare three node stores:

- **array**: sorted arrays + binary search (fast search, shifts on updates)
- **list**: sorted linked list (linear scan)
- **skip**: skip-list-based store (designed to mimic node-level $O(\log M)$ search)

3.2 Claim to Prove (from project hint)

> Using skip list to implement B+ tree nodes: prove the running time of finding a node in a B+ tree of n keys is $O(\log n)$, independent of its order M, and analyze insertion/deletion.

3.3 Formal Proof: Search is $O(\log n)$ Independent of M

A B+ tree of order M has height $H = O(\log_M n)$. If **search inside a node** costs $O(\log M)$ (e.g., binary search in an array, or expected $O(\log M)$ in a skip list), then total search time is:

$$T(n) = O(\log_M n) \cdot O(\log M) = O\left(\frac{\log n}{\log M} \cdot \log M\right) = O(\log n)$$

Therefore, the asymptotic search time is $O(\log n)$ and does **not** depend on M.

3.4 Insertion and Deletion (analysis sketch)

Insertion performs:

- one root-to-leaf descent: $O(\log_M n)$ node visits,
- node-local insert: depends on node store (array shifts $O(M)$, list scan $O(M)$, ideal skip-list $O(\log M)$),
- occasional splits that may propagate upward for at most $O(\log_M n)$ levels.

Deletion performs:

- one descent: $O(\log_M n)$,
- node-local erase: again depends on node store,
- possible rebalancing via **borrow** or **merge** up the path (at most $O(\log_M n)$ levels).

Thus, if node-local update is $U(M)$, insertion/deletion are roughly:

$$O\Big(\log_M n \cdot (\log M + U(M))\Big)$$

(amortized / expected depending on the node store).

---

4. Test Data (Generator) and Experimental Setup

4.1 Benchmark Input Format

Our benchmark reads three files:

- `-insert <file>`: keys to insert
- `-search <file>`: keys to query
- `-delete <file>`: keys to delete

File format:

- integers separated by whitespace
- optional comment lines starting with `#`

This matches our generator output exactly.

4.2 Test Data Generator (Python)

We use a Python generator program to create **reproducible** datasets. It writes:

- `<prefix>_insert.txt`
- `<prefix>_search.txt`
- `<prefix>_delete.txt`

4.2.1 Insert keys: `-insert-dist`

Main choices:

- `unique_uniform`: generate n **unique** keys uniformly in `[key_min, key_max]` (recommended).
- `sorted_unique`: generate unique keys then sort ascending.
- `reverse_unique`: generate unique keys then sort descending.
- `nearly_sorted_unique`: generate unique keys, sort, then do about `swap_frac * n` random swaps.
- `uniform`, `normal`, `exp`, `pareto`, `clusters`: other distributions (may include duplicates).

4.2.2 Search keys: hit/miss mixture

Search queries are generated with a target hit ratio `-hit-ratio`:

- hits: sampled from `insert_keys`
- misses: generated to avoid the inserted set, via:
  - `offset` (add a large offset),
  - `uniform_retry`,
  - `normal_retry`

  Queries are shuffled to avoid patterns.

4.2.3 Delete keys: `-delete-mode`

- `shuffle_all`: delete all inserted keys (random order) or the first `n_delete`.
- `in_order`: delete in insertion order.
- `random_subset`: delete a random subset of size `n_delete`.

4.2.4 Reproducibility

We fix `-seed`. The script uses deterministic derived seeds for search/delete streams, so the same command reproduces exactly the same three files.

4.3 Our Experimental Parameters (from uploaded result)

From `test.md`:

- order $M = 64$
- $n_{\text{insert}} = 200000$
- $n_{\text{search}} = 200000$
- $n_{\text{delete}} = 200000$
- 4 cases, each repeated 5 rounds, for each implementation (`array`, `list`, `skip`).

Hit ratio evidence from results.
Case 1 has `found_count = 140000` out of `200000` searches (≈ 0.7 hit ratio). Cases 2–4 have `found_count = 100000` (≈ 0.5 hit ratio). This is consistent with our generator's `-hit-ratio` feature.

---

5. Results and Evidence Supporting the Claim

5.1 Average Time per Case

Times below are averaged over 5 rounds per case (ms). `found` is average `found_count`.

case	impl	rounds	insert_ms	search_ms	delete_ms	found	height
1	array	5	19.51	13.89	18.90	140000	1
1	list	5	83.91	63.40	76.37	140000	1
1	skip	5	817.66	14.09	853.11	140000	1
2	array	5	26.95	9.66	38.61	100000	1
2	list	5	119.98	52.15	182.88	100000	1
2	skip	5	2331.71	10.14	2978.99	100000	1
3	array	5	22.85	9.13	37.64	100000	1
3	list	5	172.83	42.74	173.29	100000	1
3	skip	5	2028.60	10.49	2960.64	100000	1
4	array	5	32.87	9.66	37.75	100000	1
4	list	5	186.73	60.74	185.43	100000	1
4	skip	5	2414.19	10.25	2992.05	100000	1

5.2 Overall Average Across All Cases

impl	rounds	insert_ms	search_ms	delete_ms	found	height
array	20	25.55	10.58	33.22	110000	1
list	20	140.86	54.76	154.49	110000	1
skip	20	1898.04	11.24	2446.20	110000	1

5.3 Using the Results to Support the "$O(\log n)$ independent of M" Viewpoint

The formal proof in Section 3.3 relies on a key decomposition:

1. B+ tree search visits $O(\log_M n)$ nodes (tree height).
2. Each node visit needs an **in-node search** costing $O(\log M)$ (array binary search or skip-list expected search).
3. Multiply them to get $O(\log n)$.

Our experiment provides empirical evidence for step (2), i.e., **node-local search efficiency matters**:

- Under the same $M = 64$ and the same query workload, the **linked-list** node store (linear scan inside node) has much larger search time than the **array** and **skip** stores.
- Overall average search time (ms) from Section 5.2:
  - array: 10.58
  - skip: 11.24
  - list: 54.76

The list is about $\approx 5.17\times$ slower than array, and $\approx 4.87\times$ slower than skip in search time (overall averages). This aligns with the theoretical expectation that list-based node search is $\Theta(M)$ while array/skip are closer to $\Theta(\log M)$ for node-local search.

5.3.1 About `height_after_insert = 1` in this run

In this benchmark output, `height_after_insert = 1` in all rounds. This implies that the measured `search_ms` is dominated by **single-node lookup cost** (effectively isolating the in-node search). That is precisely the sub-problem needed in the proof: demonstrating that using better node structures reduces in-node search from linear to logarithmic.

5.3.2 Recommended follow-up to fully validate independence from M

To empirically validate the complete statement "$O(\log n)$ independent of M" end-to-end, we recommend:

- ensuring the B+ tree grows to height > 1 (so $\log_M n$ is observable),
- running multiple values of M (e.g., 32, 64, 128, 256) with the same n,
- plotting search time vs. M. Theory predicts the asymptotic trend stays $O(\log n)$; varying M changes constants but not the overall order.

---

6. Discussion and Conclusion

6.1 Discussion

Array node store. Best overall. Search is efficient (binary search), and updates shift $O(M)$ elements but remain fast in practice due to locality.

Linked-list node store. Search is slow because each in-node lookup requires a linear scan. The experimental results show a consistent multiplicative slowdown for searches.

Skip-list-based node store. Search time is close to array, but insertion/deletion are much slower. A major cause is **interface mismatch**: our `skiplist.c` is key-only, while internal B+ nodes require storing `(key, child_ptr)` pairs and supporting index-based operations. Without an indexable, value-carrying skip list (e.g., with span/width augmentation), bridging can introduce extra overhead.

6.2 Conclusion

- Skip lists support expected $O(\log n)$ search/insert/delete with expected $O(n)$ space.
- If B+ node-local search is $O(\log M)$, B+ tree search is $O(\log n)$ independent of M.
- Our benchmark results support the key step that node-local search should be logarithmic: array/skip are much faster than linked list on the same dataset.

---

7. References

- W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," 1990.
- Wikipedia: Skip list, B+ tree.