Hash Table Performance Analysis

Student ID: 2205040

Introduction

Hash tables are advanced data structures that use hashing algorithms to insert data into specific locations (called **buckets**). When searching, the hash function computes the index of the bucket where the data is expected. This drastically reduces search and deletion time compared to simple data structures like arrays, where a full traversal is required.

Hash Functions Used

sdbm Hash

The **sdbm** hash is simple and fast, especially effective for short strings. It is efficient for search operations due to its quick computation.

```
function SDBM_Hash(string):
    hash ← 0
    for each character c in string:
        hash ← ASCII(c) + (hash << 6) + (hash << 16) − hash
    return hash
```

Polynomial Rolling Hash

The **Polynomial Rolling Hash** is ideal for double hashing. It has a low collision rate and distributes values well, even for similar strings.

```
function PolynomialRollingHash(string, base, mod):
    hash ← 0
    power ← 1
    for each character c in string:
        value ← (ASCII(c) − ASCII('a') + 1)
        hash ← (hash + value × power) mod mod
        power ← (power × base) mod mod
    return hash
```

Procedure

To perform the experiment, a dataset of 10,000 words was generated using

```
generateData.cpp
```

and stored in

```
alldata.txt
```

. From this dataset, 1,000 random words were selected for testing. The same data was used across all hashing methods to ensure a fair comparison.

The experiment was conducted using varying load factors, starting from **0.4** to **0.9**. Performance metrics were recorded including collisions, average probes, and average search times before and after deletions.

Hash Table Performance Analysis

Load Factor: 0.4

Method	SDBM Hash Function				Polynomial Rolling Hash Function			
	# of Collisions during insertion	Before Deletion : Avg Search Time	After Deletion : Avg Search Time	Avg Probes Before → After	# of Collisions during insertion	Before Deletion : Avg Search Time	After Deletion : Avg Search Time	Avg Probes Before → After
Separate Chaining with balanced BST	1724	102 ns	190 ns	*N/A*	1788	245 ns	230 ns	*N/A*
Linear Probing with step Adjustment	3382	112 ns	98 ns	1→2	3402	287 ns	130 ns	1→2
Double Hashing	2811	169 ns	178 ns	1→2	2887	276 ns	187 ns	1→2

Load Factor: 0.5

Method	SDBM Hash Function				Polynomial Rolling Hash Function			
	# of Collisions during insertion	Before Deletion : Avg Search Time	After Deletion : Avg Search Time	Avg Probes Before → After	# of Collisions during insertion	Before Deletion : Avg Search Time	After Deletion : Avg Search Time	Avg Probes Before → After
Separate Chaining with balanced BST	2101	99 ns	182 ns	*N/A*	2143	133 ns	167 ns	*N/A*
Linear Probing with step Adjustment	4982	207 ns	122 ns	1→3	4888	302 ns	149 ns	2→3

Method	SDBM Hash Function				Polynomial Rolling Hash Function			
	# of Collisions during insertion	Before Deletion : Avg Search Time	After Deletion : Avg Search Time	Avg Probes Before → After	# of Collisions during insertion	Before Deletion : Avg Search Time	After Deletion : Avg Search Time	Avg Probes Before → After
Double Hashing	3884	211 ns	191 ns	1→2	3893	184 ns	192 ns	1→2

Load Factor: 0.6

Method	SDBM Hash Function				Polynomial Rolling Hash Function			
	# of Collisions during insertion	Before Deletion : Avg Search Time	After Deletion : Avg Search Time	Avg Probes Before → After	# of Collisions during insertion	Before Deletion : Avg Search Time	After Deletion : Avg Search Time	Avg Probes Before → After
Separate Chaining with balanced BST	2447	110 ns	265 ns	*N/A*	2415	142 ns	393 ns	*N/A*
Linear Probing with step Adjustment	7655	231 ns	141 ns	2→4	7441	207 ns	177 ns	2→4
Double Hashing	5145	173 ns	174 ns	2→3	5068	321 ns	224 ns	1→3

Load Factor: 0.7

Method	SDBM Hash Function				Polynomial Rolling Hash Function			
	# of Collisions during insertion	Before Deletion : Avg Search Time	After Deletion : Avg Search Time	Avg Probes Before → After	# of Collisions during insertion	Before Deletion : Avg Search Time	After Deletion : Avg Search Time	Avg Probes Before → After
Separate Chaining with balanced BST	2854	114 ns	254 ns	*N/A*	2820	148 ns	147 ns	*N/A*

Method	SDBM Hash Function				Polynomial Rolling Hash Function			
	# of Collisions during insertion	Before Deletion : Avg Search Time	After Deletion : Avg Search Time	Avg Probes Before → After	# of Collisions during insertion	Before Deletion : Avg Search Time	After Deletion : Avg Search Time	Avg Probes Before → After
Linear Probing with step Adjustment	11288	219 ns	174 ns	2→5	11616	220 ns	194 ns	2→5
Double Hashing	7125	174 ns	273 ns	2→3	7175	190 ns	232 ns	2→4

Load Factor: 0.8

Method	SDBM Hash Function				Polynomial Rolling Hash Function			
	# of Collisions during insertion	Before Deletion : Avg Search Time	After Deletion : Avg Search Time	Avg Probes Before → After	# of Collisions during insertion	Before Deletion : Avg Search Time	After Deletion : Avg Search Time	Avg Probes Before → After
Separate Chaining with balanced BST	3098	103 ns	319 ns	*N/A*	3105	142 ns	154 ns	*N/A*
Linear Probing with step Adjustment	18731	224 ns	257 ns	2→8	22627	189 ns	363 ns	3→11
Double Hashing	10136	274 ns	260 ns	2→5	9970	215 ns	309 ns	2→5

Load Factor: 0.9

Method	SDBM Hash Function				Polynomial Rolling Hash Function			
	# of Collisions during insertion	Before Deletion : Avg Search Time	After Deletion : Avg Search Time	Avg Probes Before → After	# of Collisions during insertion	Before Deletion : Avg Search Time	After Deletion : Avg Search Time	Avg Probes Before → After
Separate Chaining with balanced BST	3369	140 ns	264 ns	*N/A*	3434	186 ns	242 ns	*N/A*
Linear Probing with step Adjustment	49218	224 ns	844 ns	6→31	43475	374 ns	930 ns	5→29
Double Hashing	15242	182 ns	373 ns	2→8	15886	217 ns	391 ns	2→9

Key Observations:

- **Collision Patterns:** Linear probing shows the highest collision counts, especially at higher load factors
- **Performance Degradation:** Linear probing performance dramatically degrades at load factors 0.8 and 0.9 (highlighted rows)
- **Hash Function Comparison:** Both SDBM and Polynomial Rolling hash functions show similar collision patterns
- **Probe Counts:** Average probes increase significantly with load factor, particularly for linear probing
- **Deletion Impact:** Performance after deletion varies significantly across different methods and load factors

Impact of Load Factor on Hash Table Performance

As the load factor increases (i.e., the table becomes fuller), the performance of different hashing techniques is affected differently:

1. Chaining (Separate Chaining)

- Collisions increase linearly with load factor.
- Average search time increases moderately.
- Deletion has little impact since it only affects linked lists.
- Remains stable even at high load factors (e.g., 0.9).
- **Best suited for high load factors and dynamic deletions.**

2. Linear Probing (Open Addressing)

- Collisions rise sharply with load factor due to clustering.
- Search time and average probes increase drastically, especially after deletion.
- Performance collapses at high load factors (e.g., 31 probes at 0.9).
- **Very efficient at low load factors but unsuitable for high load or frequent deletions.**

3. Double Hashing (Open Addressing)

- Handles higher load better than linear probing.
- Collisions and search time increase, but more gradually.
- Deletion has less severe effects compared to linear probing.
- **Offers a balance between speed and resilience at medium-to-high load.**

Summary Table

Load Factor	Chaining	Linear Probing	Double Hashing
0.4 - 0.6	Fast, stable	Fast, few probes	Fast, few probes
0.7 - 0.8	Moderate load	High collisions, slower	Moderate slowdown
0.9	Acceptable	Very slow, many probes	Still usable, moderate

Conclusion

- Chaining is most robust under all load conditions.
- Double hashing is a good compromise between performance and memory use.
- Linear probing should be avoided under high load or deletion-heavy scenarios.