Theory and Code Task 4

Russell Cannon, Ian Mooney, Patrick Murphy

May 6, 2025

Abstract

citations
www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/
www.geeksforgeeks.org/string-hashing-using-polynomial-rolling-hash-function/
"Introduction to C++ Programming and Data Structures" Y. Daniel Yiang

1 Experimentation with different hash functions

1.1 Simple Hash

```
static int hash(const std::string& word, int size) {
    long int hashValue = 0;

    for (char c : word) {
        hashValue += c;
    }

    return hashValue % size;
}
```

Results

Hash Type	Time(ms)	λ	Chain Length/ Cluster Size		
			Max	Min	Mean (non-zero)
Open	31	0.705688	33	0	5.74652
Linear	283	0.340332	5481	1	242.435

The simple hash function suggested works but causes considerable collisions.

1.2 Rabin-Karp Hash

```
static int hash(const std::string& word, int size) {
    int n = 0; // Hash value
    int d = 256; // number of characters in the input alphabet
    for (int i = 0; i < (int)word.size(); i++)
        n = (d * n + word[i]) % size;

    return abs(n);
}
```

Results

Hash Type	Time(ms)	λ	Chain Length/ Cluster Size		
			Max	Min	Mean (non-zero)
Open	40	0.705688	740	0	20.573
Linear	80	0.340332	4879	1	242.435

This hash function modified from the suggested hash function used in the Rabin-Karp pattern matching algorithm turned out to be worse than the most simple hash function when it comes to separate chaining, but better for linear probing. Chain lengths are out of control in this hash function and the times are marginally worse for open hashing. The linear probing times are considerably better, but the size of clusters is about the same.

1.3 Polynomial Rolling Hash

Our second attempt: (Polynomial Rolling Hash from Geeks4Geeks)

```
static int hash(const std::string& word, int size) {
    int p = 31, m = 1e9 + 7, hashValue = 0, pPow = 1;

    for (char c : word) {
        hashValue = (hashValue + (c - '-' + 1) * pPow) % m;
        pPow = (pPow * p) % m;
    }

    return abs(hashValue % size);
}
```

Results

Hash Type	Time(ms)	λ	Chain Length/ Cluster Size		
			Max	Min	Mean (non-zero)
Open	46	0.705688	6	0	1.40044
Linear	24	0.340332	32	1	1.79491

The runtime for open chaining is actually worse for this hash function every other statistic is improved exponentially.

1.4 Polynomial Rolling Hash With Reduced Alphabet

third attempt: (maps only the characters we allow)

```
static int charToIndex(const char c) {
    //-, a, b, ..., z, 0, 1, ... 9
    if (c == '-') return 1;
    if (c == '\'') return 2;
    if (std::isalpha(c)) return c - 'a' + 3;
    if (std::isdigit(c)) return c - '0' + 26 + 4;
    return 0;
}

static int hash(const std::string& word, int size) {
    int p = 31;
    long int m = 1e9 + 7, hashValue = 0, pPow = 1;

    for (char c : word) {
        hashValue = (hashValue + charToIndex(c) * pPow) % m;
        pPow = (pPow * p) % m;
    }

    return hashValue % size;
```

}

Results

Hash Type	Time(ms)	λ	Chain Length/ Cluster Size		
			Max	Min	Mean (non-zero)
Open	26	0.705688	5	0	1.389
Linear	21	0.340332	14	1	1.80804

The max chain length and the runtime for linear probing are marginally improved but the runtime for open chaining is finally down. Due to continually reduced gains, this hashing method was decided to be best for our use case.

2 Experimenting with occupancy ratios in linear probing

Exp. #	Cutoff	Time(ns)	λ
1.	0.5	22045420	0.340332
2.	...	22854454	...
3.	...	27654239	...
Mean	0.5	24184704	0.340332
1.	0.9	21338314	0.754639
2.	...	22113859	...
3.	...	22115551	...
Mean	0.9	21855908	0.754639
1.	0.75	22713966	0.680664
2.	...	20728820	...
3.	...	21238333	...
Mean	0.75	21560373	0.680664

Although runtimes did improve, the results could likely be attributed to random error.

3 Optimizing chain length in open hashing

Exp. #	Cutoff	Time(ns)	λ	Chain Length	
				Max	Mean (non-zero)
1.	1.0	28200083	0.705688	5	1.389
2.	...	28867946	...		...
3.	...	27633391	...		...
Mean	1.0	28233807	0.705688	5	1.389
1.	0.9	25575107	0.705688	5	1.389
2.	...	26536237	...		...
3.	...	25898983	...		...
Mean	0.9	26003442	0.705688	5	1.389
1.	0.5	29691677	0.352844	4	1.18755
2.	...	27364069	...		...
3.	...	27763650	...		...
Mean	0.5	28273132	0.352844	4	1.18755
1.	0.25	41166534	0.176422	3	1.08952
2.	...	30426945	...		...
3.	...	32216903	...		...
Mean	0.25	34603460	0.176422	3	1.08952

Keeping lambda under 0.9 provided the best runtimes in separate chaining. Notably, Y. Daniel Yiang in "Introduction to C++ Programming and Data Structures" recommends a cutoff of 0.9. But to optimize chain length against time, a cutoff of 0.5 works best.

4 Handling collisions in the table for linear probing

Handling collisions in the table for linear probing. The collision resolution method implemented must be described, with research and inclusion of a method described in the lecture.

4.1 Linear

```
int index = hash(pair.word, size);
while (!arr[index].empty && arr[index].word != pair.word) {
    index = (index + 1) % size;
}
```

4.2 Quadratic Probing

```
int index = hash(pair.word, size);
for (int i = 1; !arr[index].empty && arr[index].word != pair.word; i++) {
    index = (index + i*i) % size;
}
```

Results

Probe type	Time(ms)	λ	Cluster Size		
			Max	Min	Mean (non-zero)
Linear	21	0.340332	14	1	1.80804
Quadratic	32	0.340332	13	1	1.77799

Despite marginal improvements in custer size, the quadratic probing adds considerable runtime to our program. It was decided we would stick with linear probing.

5 80 Least and most frequent words

Linear probing creates a heap and pushes all the words in the hash table onto it. Then, it pops the top of the heap 80 times for the most frequently occurring words. The same method must be repeated for the least frequent words.

Separate chaining creates a new array the size of the occupancy in the hash table, sets everything in the hash table to an index in the array, and sorts the array. It reads the top 80 words in the sorted array and the bottom 80 words in the array for the most and least frequent words.

After both hashing methods are complete, we create another hash table (using linear probing) to combine the most frequent words in works I through VI (the separate chaining hash table) and works VII through XII (the linear probing hash table). Then the same is repeated for the least frequent words.

5.1 Most Frequent Words

1. the: 2855
2. i: 1602
3. and: 1485
4. to: 1392
5. a: 1353
6. of: 1311
7. that: 923
8. in: 894
9. it: 889
10. you: 803
11. was: 696
12. he: 661
13. is: 581
14. my: 580
15. his: 514
16. have: 474
17. had: 434
18. with: 419
19. at: 418
20. which: 404
21. as: 403
22. for: 374
23. me: 358
24. but: 338
25. be: 335
26. not: 308

27. we: 287
28. this: 278
29. there: 265
30. her: 262
31. said: 259
32. she: 252
33. so: 250
34. from: 236
35. your: 218
36. holmes: 213
37. upon: 212
38. been: 207
39. very: 207
40. no: 206
41. all: 204
42. one: 202
43. what: 196
44. him: 195
45. then: 194
46. on: 188
47. were: 185
48. are: 182
49. by: 173
50. into: 164
51. out: 163
52. when: 162
53. an: 161

54. would: 161
55. has: 160
56. up: 155
57. little: 151
58. could: 146
59. who: 142
60. man: 135
61. do: 134
62. now: 133
63. will: 133
64. our: 131
65. should: 130
66. if: 128
67. see: 118
68. mr: 114
69. some: 112
70. down: 110
71. think: 106
72. over: 101
73. may: 99
74. they: 95
75. shall: 93
76. before: 91
77. door: 91
78. only: 91
79. more: 90
80. can: 89

5.2 Least Frequent Words

Because there are any number of words that are only used once, the sorting method we used defaults to sorting words by their alphabetic position if their counts are the same.

1. zigzag: 1
2. zest: 1
3. zero-point: 1
4. youngster: 1
5. you've: 1
6. yonder: 1
7. yellow-backed: 1
8. yell: 1
9. yawning: 1
10. xii: 1
11. xi: 1
12. x: 1
13. wrote: 1
14. writhing: 1
15. wrists: 1
16. wrinkles: 1
17. wrenching: 1
18. wreaths: 1
19. wreath: 1
20. wounded: 1
21. worst: 1
22. worry: 1
23. worm-eaten: 1
24. worlds: 1

25. world-wide: 1
26. workmen: 1
27. worked: 1
28. wooing: 1
29. wooden-legged: 1
30. wooden-leg: 1
31. wonderfully: 1
32. wonderful: 1
33. wondered: 1
34. womanly: 1
35. womanhood: 1
36. woke: 1
37. withdraw: 1
38. wishing: 1
39. wisely: 1
40. wiry: 1
41. wintry: 1
42. winter: 1
43. wink: 1
44. wine-cellar: 1
45. window-sill: 1
46. windfall: 1
47. wind-swept: 1
48. wincing: 1
49. wimpole: 1
50. wilton: 1
51. will-o'-the-wisp: 1

52. wigmore: 1
53. widow: 1
54. wicket: 1
55. wicker-work: 1
56. wholesome: 1
57. whoever: 1
58. whittington: 1
59. whiter: 1
60. whiten: 1
61. white-counterpaned: 1
62. white-aproned: 1
63. whisky: 1
64. whishing: 1
65. whined: 1
66. whine: 1
67. whims: 1
68. wherever: 1
69. where's: 1
70. whenever: 1
71. wheels: 1
72. westphail: 1
73. western: 1
74. westbury: 1
75. westaway's: 1
76. westaway: 1
77. well-to-do: 1
78. well-opened: 1
79. well-cut: 1
80. weighed: 1