Hash Table (Homework 1)

Naho Yokoyama

Overview

This program implements a hash table. It stores key-value pairs, where the keys are of string type and the values can be of any type, and allows for the three operations of addition, search, and deletion of data to be performed in approximately O(1) time. When hash collisions occur, it uses linked lists to store the collided elements. Additionally, it has a rehashing function to dynamically resize the table.

Important Functions

1. Addition of data (put)
   Argument(s): key (String), value (any type)
   Return value(s): True (When it successfully adds the data)
   False (When the key is already exists; it updates the value)

2. Search data (get)
   Argument(s): key (String)
   Return value(s): (value, True) (When it finds the key in the table)
   (None, False) (When it cannot find the key in the table)

3. Deletion of data (delete)
   Argument(s): key (String)
   Return value(s): True (When it successfully deletes the data)
   False (When it cannot find the key to be deleted)

Explanation of the algorithm

1. Hash function
   Since the keys are of string type, a hash function is defined to generate an integer hash value from the strings. By converting this hash value to an index in the hash table, it can determine the position of the key to be stored.

   1.1 How to calculate a hash value

   The hash value of a string is calculated using the following formula. Note that N is the length of the string, and the characters are counted from the left:。

$$Hash\ value\ of\ the\ string = \sum_{i=1}^{N} ASCII\ value\ of\ the\ i^{th} alphabet \times 2^{i-1}$$

2. Addition of data

The addition of data is performed as follows:

1. Check if the key is of string type
2. Calculate the hash value of the key and take the remainder when divided by the size of the hash table (k) to get the index.
3. Check the linked list at the corresponding index to ensure there are no duplicate keys
4. Add the data to the head of the linked list (if the key already exists, update the value)

3. Getting data

Getting data is done based on the algorithm below:

1. Check if the key is of string type
2. Calculate the hash value of the key and take the remainder when divided by the size of the hash table (k) to get the index
3. Check the linked list at the corresponding index and return the value associated with the key if found

4. Deletion of data

Deletion of data is done as follows:

1. Check if the key is of string type.
2. Calculate the hash value of the key and take the remainder when divided by the size of the hash table (k) to get the index.
3. Check the linked list at the corresponding index and locate the data to be deleted using the key.
4. Delete the data and update the links in the linked list accordingly.

5. Rehash

If the number of elements in the hash table becomes less than 30% or more than 70% of the table size, the table size is automatically adjusted, and rehashing is performed.

5.1 How to calculate the size of the new hash table

The new size of the table is calculated as follows, given the current table size k:

When expanding the table: k × 2 + 1

When shrinking the table: k ÷ 2 (if this value is even, add 1 to make it odd)

The reason for making the table size odd is to reduce hash collisions. Ideally, the size would be a prime number, but this time, I just ensured that it is odd.

Key points that were considered

1. Hash function

The sample hash function implemented initially just summed the character codes, which led to a high possibility of hash collisions. For example, swapping characters would generate the same hash value. Therefore, the hash function was improved as described in section 1.1, using the following approach (where N is the length of the string):

$$Hash\ value\ of\ the\ string = \sum_{i=1}^{N} ASCII\ value\ of\ the\ i^{th} alphabet \times 2^{i-1}$$

The performance improvements were measured using the performance_test() function in the sample code. We compared three versions: no optimizations as a baseline, with rehashing, and optimized hash function

Figure 1 shows the measured execution times and although there is some variations, it is evident that implementing rehashing improved execution time by approximately half compared to before rehashing was implemented. However, both versions showed an increase in the execution time as the data volume increased, indicating they do not operate in O(1) as expected. In contrast, the version with the optimized hash function maintained an execution time of around one second, even as the data volume increased, demonstrating that it operates in nearly O(1) time. This shows the significant performance benefit of using a well-designed hash function in the hash table implementation.

Figure 1. Execution time of the hash table

	Execution Time (s)		
# of repetitions	First Attempt	Implemented Rehash	Changed Hash Function
0	0.593371	0.239938	0.218420
1	1.303396	0.708123	0.339913
2	2.726115	0.765740	0.182390
3	4.032746	1.694646	0.410773
4	5.717483	1.903952	0.231887
5	8.058983	4.562716	0.266127
6	9.898893	6.431457	0.417388
7	11.916055	6.391488	0.652678
8	14.081833	7.586612	0.321649
9	15.546169	8.912681	0.504272
10	17.020840	11.059493	0.379099
11	18.778003	13.125428	0.408289
12	23.124359	18.225282	0.470781
13	30.497726	15.283265	0.476793
14	24.826802	13.929575	1.705123
15	27.319383	13.721803	0.450372
16	28.730599	14.863264	0.466214
17	30.674497	16.870347	0.493391
18	32.946332	20.481150	0.523585
19	40.789052	23.237447	0.562112
20	49.705205	21.204512	0.594223
21	330.141636	25.257546	0.622680
22	41.247047	25.584767	0.652242
23	41.092272	28.479581	1.056136
24	42.980959	37.800339	0.744399
25	45.976101	40.970640	0.801615
26	48.157236	56.599727	0.883740
27	48.297085	43.700714	0.901758
28	49.693645	35.337984	3.415403
29	52.641038	25.302304	0.704611
30	53.606064	27.358870	0.751730