ハッシュテーブル（宿題 1）

横山奈穂

概要

このプログラムは、ハッシュテーブルを実装したものだ。キー（文字列型）と値（任意の型）のペアを格納し、ほぼ O(1)で追加、検索、削除、の 3 操作ができる。ハッシュ値が衝突した時には連結リストを使用して格納する。また、動的なサイズ変更ができるよう、(再ハッシュ)機能を持つ。

主な機能

1. データの追加（put）
   引数: キー（文字列型)、値（任意の型）
   戻り値: True（データの追加ができた時）
   　　False（既にキーが存在し、値を更新した時）

2. データの検索（get）
   引数: キー（文字列型）
   戻り値:（値, True）（キーがテーブル内に見つかった場合）
   　　（None, False）（キーがテーブル内に見つからなかった場合）

3. データの削除（delete）
   引数: キー（文字列型）
   戻り値: True（データの削除ができた時）
   　　False（削除するデータがテーブル内に見つからなかった場合）

アルゴリズムの説明

1. ハッシュ関数
   キーは文字列型のため、文字列から整数ハッシュ値を生成するためにハッシュ関数を定義している。このハッシュ値をハッシュテーブルのインデックスに変換することで、キーの位置を決定する。

   1.1 ハッシュ値の計算方法
   文字列のハッシュ値は以下の計算式で計算される。なお、N は文字列の長さで、文字数は左からカウントするものとする。

$$文字列のハッシュ値 = \sum_{i=1}^{N} i文字目のアルファベットのASCIIコード \times 2^{i-1}$$

2.　データの追加

データの追加は以下のように行われる。

1. キーが文字列型か確認
2. キーのハッシュ値を計算し、ハッシュテーブルのサイズ（k）で割ったあまりをインデックスとする
3. インデックスに対応する場所にある連結リストを確認し、重複するキーがないか確認
4. 連結リストの先頭にデータを追加（すでにそのキーが存在する場合は値を更新）

3. データの取得

データの取得は以下のように行われる。

1. キーが文字列型か確認
2. キーのハッシュ値を計算し、ハッシュテーブルのサイズ（k）で割ったあまりをインデックスとする
3. インデックスに対応する場所にある連結リストを確認し、見つけたいキーに対応する値を返す

4. データの削除

データの削除は以下のように行われる。

1. キーが文字列型か確認
2. キーのハッシュ値を計算し、ハッシュテーブルのサイズ（k）で割ったあまりをインデックスとする
3. インデックスに対応する場所にある連結リストを確認し、キーを元に削除したいデータを見つける
4. データを削除し、連結リストのリンクを更新する

5. 再ハッシュ機能

ハッシュテーブルの要素数がテーブルサイズの 30%以下、もしくは 70%以上になると、自動的にテーブルの大きさを変更し、再ハッシュを行う。

5.1 新しいテーブルサイズの計算方法

現在のテーブルの大きさを k とすると、新しいテーブルの大きさは以下のように計算される。

テーブルを拡張する場合: k × 2 + 1
テーブルを縮小する場合: k ÷ 2 (この値が偶数の場合は+1)

テーブルサイズを奇数にする理由は、その方がハッシュ値の衝突が少なくなるからだ。本来であれば素数にしたいが、今回はただ奇数であればよいとした。

工夫した点

1. ハッシュ関数の工夫

サンプルとして実装されていたハッシュ関数は、文字コードを足すのみだったが、これではハッシュ値の衝突が起きてしまう可能性が高い。例えば、文字を入れ替えても同じハッシュ値が生成されてしまう。そこで、ハッシュ関数を工夫し、前項 1.1 で説明した通り、以下のようにした (N は文字列の長さ)。

$$文字列のハッシュ値 = \sum_{i=1}^{N} i文字目のアルファベットのASCIIコード \times 2^{i-1}$$

これによるパフォーマンスの改善を、サンプルコードの performance_test()関数を使って測定した。比較の対象として、何も工夫をしないもの、再ハッシュを実装したもの、ハッシュ関数を工夫したもの、の 3 つを選んだ。図 1 に測定した実行時間をしめす。
図 1 を見ればわかるように、多少ばらつきがあるものの、再ハッシュを実装する前と後で、実行時間が半分程度改善している事が分かった。ただ、どちらもデータの量が増えるにつれ実行時間が増加しており、これでは O(1)で動いているとはいいがたい。逆に、ハッシュ関数を工夫したものは、データ数が多くなっても実行時間が 1 秒弱程度を保っており、ほぼ O(1)で動いている事が分かる。

図 1 ハッシュテーブルの実行時間

	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