オブジェクト指向哲学
～入門編～

第13回　　ハッシュテーブルを使ったプログラム

～高速に検索するには？～

学習目標

- ハッシュテーブルの概念を説明できる
  - ハッシュテーブルの利点･欠点を説明できる
  - 検索における key と value の関係を説明できる
  - 簡単なハッシュテーブルの仕組みを説明できる
- ハッシュテーブルを実装できる
  - 汎用的なハッシュテーブルを実装できる

13.1. ハッシュテーブルとは

13.1.1.　検索と効率

今までにやってきた検索方法は「リニアサーチ」、「バイナリサーチ」でした。その効率を下にまとめます。

リニアサーチ	O(N)
バイナリサーチ	O(logN)

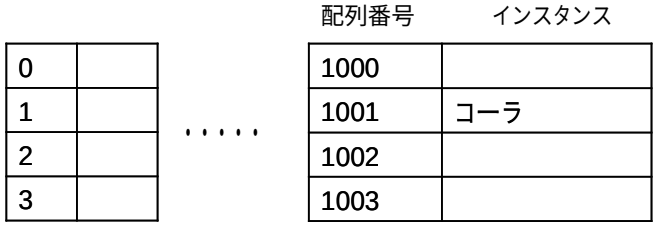
それに比べて、今回学習するハッシュテーブルは、検索の効率がほぼ O(1)のオーダーになるという、きわめて高速に検索できるデータ構造です。

ハッシュテーブル	O(1)

13.1.2.　ハッシュテーブル概要

ハッシュテーブルの基本的な考え方は、検索のキー（今までは商品番号を使ってきました）をそのまま配列番号にして入れておくというものです。

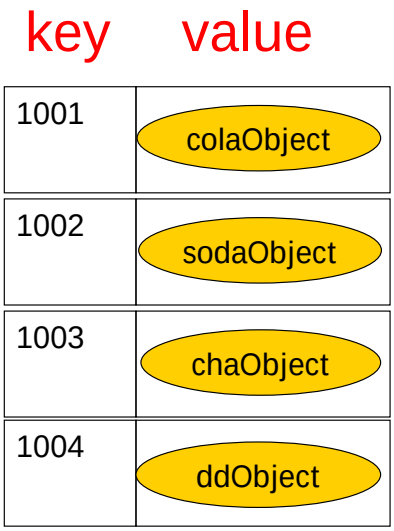
例えば、1001 という商品番号で後から商品種類インスタンスを検索したい場合、配列の 1001 番目にいれておけば、始めから順番に検索したりすることなく、ずばり 1 回で目標にたどり着けることになります。



しかし、皆さんお気づきのように、「そんな大きい配列を作ることができるか」などの問題があります。それらの問題を解決しながら、ハッシュテーブルの仕組みを学んでいきましょう。

13.1.3.　key と value

ここで、検索する時の key と value という概念を確認しておきます。key とは検索する時に使うキーのことで、value は検索される値のことです。例えば、今まで行ってきた検索は、商品番号をキーにして、商品種類インスタンスを値として取り出しました。ハッシュテーブルは、key と value の関係を保ったままデータを格納する方法なので、ハッシュテーブルを学ぶ際には、この key と value の関係を意識することが大切です。

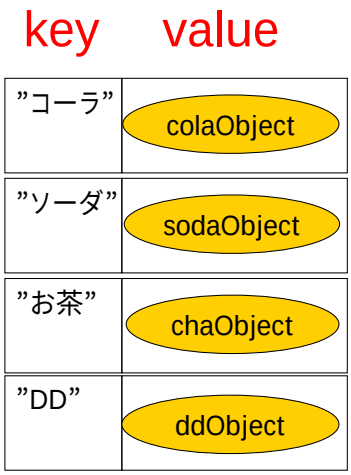


13.2. ハッシュテーブルの仕組み

13.2.1.　名前をキーに検索できるようにする

ところで、第 11 回で扱った自動販売機は、検索を商品番号で行っているため、非常に使いにくいという欠点がありました。

利用するユーザにとっても、管理者にとっても、「cola」は、1001 ではなくて、"cola"と扱いたいものですね。今まで名前ではなく商品番号で扱っていた理由は、バイナリサーチや並び替えなどを考える時には、番号のほうが扱いやすかったためです。しかし、ハッシュテーブルは、文字列のような数字ではない検索のキーに対しても検索が得意です。今回は、名前で検索できる商品種類管理プログラムを書いてみましょう。



13.2.2.　ハッシュ値を求める

ハッシュテーブルを作るためには、ハッシュ値を求める、つまり、キーを配列のインデクス番号（配列の番地）に変換する必要があります。ハッシュ値の求め方を説明します。

①.名前を数字に変換する

文字列のハッシュ値を求めるためには、まず、文字列を数字に変換する必要があります。これをインデックシングといいます。インデックシングには様々な方法が考えられます。例えば、次のような方法です。（この方法はかなりいい加減なので、詳しくは専門書を見てください。）

変換表

コ	98
ー	103
ラ	67

コーラ　→　9810367

文字列の変換表（文字列はコンピュータの内部では数字で扱われますから、変換表というよりもその文字の文字コードを扱えばよいのです）を用意して、それを単純につなげたり、足し算したりする方法が考えられます。

(1)Java でインデックシングしたハッシュ値を求める

Java では、インデックシングをするメソッドが標準装備されているので、今回はそれを利用することにします。

Object クラスに、hashCode()というメソッドがあります。Object クラスは、すべてのクラスのスーパークラスですから、String（文字列）クラスも当然このメソッドを利用することができます。

```
String cola = "コーラ";
int hashCode = cola.hashCode();
```

もちろんStringクラスもObjectクラスを継承しているよ

②.配列の番号と対応させる

次に、インデックシングした番号をそのまま、配列の番地として、配列に格納します。このように格納すれば、後で検索する時にも番地を求めて、1 回で検索が終了するはずです。

1	
2	
3	
4	

・・・・・

9810366	
9810367	コーラ
9810368	
9810369	

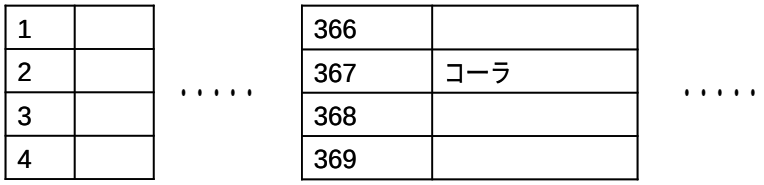
しかし、世の中はそううまくいきません。このような方法でうまくいくほど、番地が小さければ、問題ありませんが、上記の例で考えると、非常に大きな配列を作る必要があります。これはメモリに入りきるかどうか分からない数字です。そのため、小さい配列で済むような方法を考える必要があります。

③.小さい配列で済むためには

そこで、小さい配列で済む方法として、配列の大きさで割ったあまりを求めて、番地を小さくする方法を考えます。この数字を求めることをハッシングといいます。

例えば、先ほどの例を 1000 要素の配列で済ませることを考えると、9810367 を 1000 で割ったあまり 367 を求めて、配列の番地とします。

コーラ　→　9810367　→　367



※　Java プログラムで割ったあまりを求めるには「%」演算子を使います。

```
int hashCode = 9810367%1000;
```

13.2.3.　番地の衝突を回避する

①.番地の衝突

ハッシュ値を小さくするために､配列の要素数で割ったあまりを求めるのはすばらしい解決方法ですが、問題があります。番地の衝突が起こることです。

例えば、次のような例を考えましょう。キーである"コーラ"も"ソーダ"もハッシュ値を求めたところ、番地が同じになってしまいました。

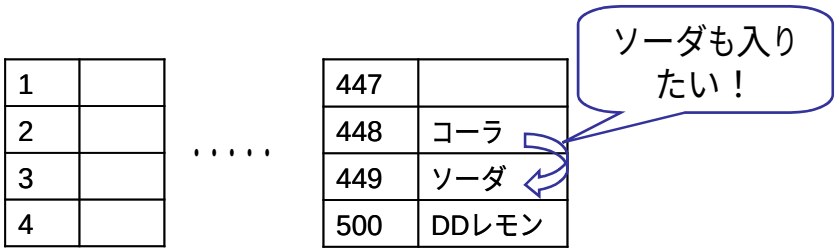
コーラ　→　31256448　→　448
ソーダ　→　587648448　→　448

これらの番地の衝突を回避する方法を考えましょう。

②.衝突回避①：空き番地法

衝突を回避する方法として、求めた番地に既に値が入っている場合、次の番地に入れておくというものです。次の番地も値が入っていた場合は、またその次の番地、とあいている番地を探していきます。

検索する時は、まず求めた番地を探し、ない場合は、リニアサーチで次、次と探します。これでも、ハッシュ値を使うことで、かなり近い番地まではたどり着くことができるので、検索は相当速くなるはずです。（O(1)とは行きませんが…）



空き番地法のリストを次に示します。

例題 13-1：空き番地法によるハッシュテーブル(ItemTypeList.java)

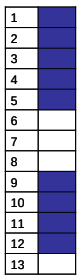
```
1:      /**
2:      * オブジェクト指向哲学　入門編
3:      * 例題 13-1：空き番地法によるハッシュテーブル
4:      * 商品種類を名前で検索するプログラム
5:      *
6:      * 商品種類リストクラス
7:      */
8:      public class ItemTypeList {
9:
10:       private int ARRAY_SIZE = 20;
11:       private ItemType[] itemTypeArray = new ItemType[ARRAY_SIZE];
12:       private ItemType nonItemType;//削除したデータを置き換えるための商品種類
13:       //データの削除は実際には削除されるわけではなくこのデータと置き換えられている
14:
15:       /**
16:       * コンストラクタ
17:       */
18:       public ItemTypeList() {
19:         nonItemType = new ItemType ("noItem",-1);//削除用のデータを初期化する。
20:       }
21:
22:       /**
```

```
23:       * 新たに商品種類を追加する
24:       */
25:      public void add(ItemType value){
26:        String key = value.getName();      //商品名をキーとする
27:        //ハッシュ値を求める
28:        int hashCode = key.hashCode();
29:        int arrayLoc = hashCode % ARRAY_SIZE;
30:
31:        //ハッシュした番地から、空のセルを探す
32:        while(true){
33:          if(itemTypeArray[arrayLoc] == null){//空のセルがあった
34:            itemTypeArray[arrayLoc] = value;//空のセルに商品種類を追加する
35:            return;
36:          }
37:
38:          //空のセルがなかったら
39:          arrayLoc++;//次の項目へ
40:          if(arrayLoc >= ARRAY_SIZE ){//必要ならラップアラウンド
41:            arrayLoc = 0;
42:          }
43:        }
44:      }
45:
46:      /**
47:      * 商品種類を商品名をキーに検索する
48:      */
49:      public ItemType search(String key){
50:        //ハッシュ値を求める
51:        int hashCode = key.hashCode();
52:        int arrayLoc = hashCode % ARRAY_SIZE;
53:
54:        //空のセルになるまで順番に探す
55:        while(itemTypeArray[arrayLoc] != null){
56:          if(itemTypeArray[arrayLoc].getName().equals(key)){
57:            //Key が等しい商品が見つかった
58:            return itemTypeArray[arrayLoc];
59:          }
60:
61:          arrayLoc++;//次の項目へ
62:          if(arrayLoc >= ARRAY_SIZE ){//必要ならラップアラウンド
63:            arrayLoc = 0;
64:          }
65:        }
66:
67:        return null;//見つからなかった
68:      }
69:
70:      /**
71:      * 指定されたキーを持つ商品種類をリストから削除する
72:      */
73:      public void remove(String key){
74:        //ハッシュ値を求める
75:        int hashCode = key.hashCode();
76:        int arrayLoc = hashCode %ARRAY_SIZE;
```

```
77:
78:         //空のセルになるまで順番に探す
79:         while(itemTypeArray[arrayLoc] != null){
80:           if(itemTypeArray[arrayLoc].getName().equals(key)){
81:             //削除する商品が見つかったら
82:             itemTypeArray[arrayLoc] = nonItemType;//削除対象を削除用データに置き換える
83:             return ;
84:           }
85:
86:           arrayLoc++;//次の項目へ
87:           if(arrayLoc >= ARRAY_SIZE ){//必要ならラップアラウンド
88:             arrayLoc = 0;
89:           }
90:         }
91:
92:       }
93:
94:       /**
95:       * 商品種類を表示する
96:       */
97:       public void display(){
98:         for(int i=0;i<ARRAY_SIZE;i++){
99:         //配列の要素に対して繰り返す
100:          if(itemTypeArray[i] != null && itemTypeArray[i] != nonItemType){//商品が入っ
ている
101:
System.out.println(itemTypeArray[i].getName()+":"+itemTypeArray[i].getPrice()+"円:");
102:          }
103:        }
104:      }
105:    }
```

③.空き番地法の問題点

しかし、空き番地法は、要素数が増えてくると極端に効率が悪くなります。局所的に要素が入っている範囲（クラスター）ができるからです。そのクラスター内にハッシュされた値は、自分を挿入するためにクラスターの端までリニアサーチで探していかなければないので、効率が悪くなり、そのクラスターは大きくなっていきます。

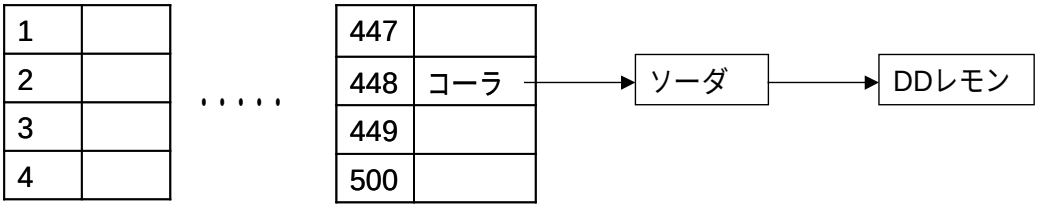


④.衝突回避②：分離連鎖法

今回は、クラスター化を防ぐ方法として、分離連鎖法を紹介します。

分離連鎖法は、番地が衝突したら、その番地から連結リストを使ってデータを格納する方法です。

検索する時は、ハッシュ値を求めた後、連結リストをたどって検索します。



次に、分離連鎖法で実装したハッシュテーブルのリストを示します。

例題 13-2：分離連鎖法によるハッシュテーブル(LinkObject.java)

```
1:      /**
2:      * オブジェクト指向哲学　入門編
3:      * 例題 13-2：分離連鎖法によるハッシュテーブル
4:      * 商品種類を名前で検索するプログラム
5:      *
6:      * 連結リストインスタンスクラス
7:      */
8:      public class LinkObject {
9:
10:       private ItemType value;     //商品種類
11:       private LinkObject next;   //次の要素への参照
12:
13:       /**
14:       * コンストラクタ
15:       */
16:       public LinkObject(ItemType newValue){
17:         value = newValue;
18:       }
19:
20:       /**
21:       * リンクインスタンスが保持している商品種類を取得する
22:       */
23:       public ItemType getValue(){
24:         return value;
25:       }
26:
27:       /**
28:       * リンクインスタンスの次の要素を取得する
```

```
29:       */
30:       public LinkObject getNext(){
31:         return next;
32:       }
33:
34:       /**
35:       * リンクインスタンスに次の要素を設定する
36:       */
37:       public void setNext(LinkObject newLink){
38:         next = newLink;
39:       }
40:
41:     }
```

ItemTypeList.java(例題 13-2：分離連鎖法によるハッシュテーブル)

```
1:      /**
2:      * オブジェクト指向哲学　入門編
3:      * 例題 13-2：分離連鎖法によるハッシュテーブル
4:      * 商品種類を名前で検索するプログラム
5:      *
6:      * 商品種類リストクラス
7:      */
8:      public class ItemTypeList {
9:
10:       private int ARRAY_SIZE =5;                                          //配列の最大
要素数
11:       private LinkObject[] itemTypeLinkArray = new LinkObject[ARRAY_SIZE];//連結リスト
の先頭となるリンクインスタンスの配列
12:
13:       /**
14:       * コンストラクタ
15:       */
16:       public ItemTypeList() {
17:       }
18:
19:       /**
20:       * 新たに商品種類を追加する
21:       */
22:       public void add(ItemType value){
23:         String key = value.getName();//商品名をキーに設定する
24:
25:         //ハッシュ値を求める
26:         int hashCode = key.hashCode();
27:         int arrayLoc = hashCode % ARRAY_SIZE;
28:
29:         LinkObject insertLink = new LinkObject(value);
30:         if (itemTypeLinkArray[arrayLoc] == null){
31:           //ハッシュした番地が未使用だったとき
32:           itemTypeLinkArray[arrayLoc] = insertLink;//追加
```

```
33:         }else{
34:           //ハッシュした番地が使用済みだったとき
35:           LinkObject current = itemTypeLinkArray[arrayLoc];//現在検索中のリンク
36:           while (true){
37:             if(current.getNext() == null){
38:               //連結リストの最後が見つかった
39:               current.setNext(insertLink);
40:               break;
41:             }else{
42:               //次のリンクを検索中のリンクに設定
43:               current = current.getNext();
44:             }
45:           }
46:         }
47:
48:       }
49:
50:       /**
51:       * 商品種類を商品名をキーにして検索する
52:       */
53:       public ItemType search(String key){
54:       //ハッシュ値を求める
55:         int hashCode = key.hashCode();
56:         int arrayLoc = hashCode % ARRAY_SIZE;
57:
58:         LinkObject current = itemTypeLinkArray[arrayLoc];//現在検索中のリンク
59:
60:         while (current != null){
61:         //ハッシュした番地に調査していないリンクインスタンスが残っている
62:           if(current.getValue().getName().equals(key)){
63:             //検索対象が見つかった
64:             return current.getValue();
65:           }else{
66:             //次のリンクを検索中のリンクに設定
67:             current = current.getNext();
68:           }
69:         }
70:
71:         //ハッシュした番地にリンクインスタンスがなかったとき
72:         return null;//見つからなかった
73:
74:       }
75:
76:       /**
77:       * 指定されたキーを持つ商品種類をリストから削除する
78:       */
79:       public void remove(String key){
80:         //ハッシュ値を求める
81:         int hashCode = key.hashCode();
82:         int arrayLoc = hashCode % ARRAY_SIZE;
83:
84:         LinkObject current = itemTypeLinkArray[arrayLoc];//現在検索中のリンクインスタ
ンス
85:         LinkObject prev = null;//現在検索中の一つ前のリンクインスタンス
```

```
86:
87:         //ハッシュした番地が使用されているとき
88:         while (current!=null){
89:           if(current.getValue().getName().equals(key)){
90:             //削除対象が見つかった
91:             if(prev==null){
92:               //連結リストの先頭だったとき
93:               if(current.getNext()== null){
94:                 //次の要素がないとき
95:                 itemTypeLinkArray[arrayLoc] = null;//対象の番地のリンクインスタンスを
削除する
96:                 break;
97:               }else{
98:                 //次の要素があるとき
99:                 itemTypeLinkArray[arrayLoc] = current.getNext();//次の要素へ配列からリ
ンクをつくる
100:                break;
101:              }
102:            }else if(current.getNext()==null){
103:              //連結リストの最後だったとき
104:              prev.setNext(null);//削除するリンクインスタンスへのリンクを消す
105:              break;
106:           }else{
107:             //連結リストの最初でも最後でもないとき
108:             //削除するリンクインスタンスの一つ前のリンクインスタンスから,削除するリ
ンクインスタンスの次のリンクインスタンスへ
109:             //リンクをはる
110:             prev.setNext(current.getNext());
111:             break;
112:           }
113:         }else{
114:           prev = current;//現在ののリンクインスタンスを一つ前のリンクインスタンスに
設定
115:           current = current.getNext();//次のリンクインスタンスを検索中のリンクインス
タンスに設定
116:         }
117:       }
118:     }
119:
120:     /**
121:     * 商品種類を表示する
122:     */
123:     public void display(){
124:       LinkObject current;//現在検索中のリンクインスタンス
125:       ItemType showItemType;//表示する商品種類
126:
127:       for(int i=0; i<ARRAY_SIZE; i++){
128:         current = itemTypeLinkArray[i];//検索中のリンクインスタンスを設定する
129:
130:         while(current != null){//リンクインスタンスがあるとき
131:           showItemType = current.getValue();
132:           System.out.println(showItemType.getName()+":"+showItemType.getPrice()+"
円:");
133:
```

```
134:             current = current.getNext();//リンクを次へ進める
135:           }
136:
137:         }
138:       }
139:     }
```

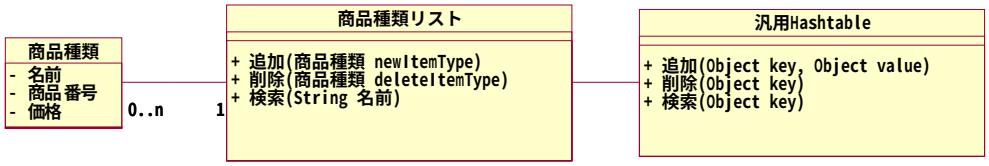
13.2.4.　ハッシュテーブルの利点･欠点

＜議論しよう！＞ハッシュテーブルの利点･欠点

- 利点
  - ✧
  - ✧
  - ✧
  - ✧
  - ✧
  - ✧
- 欠点
  - ✧
  - ✧
  - ✧
  - ✧
  - ✧
  - ✧

13.3. 汎用的なハッシュテーブル

前回作った汎用的なリストのように、汎用的なハッシュテーブルを作ることを考えてみましょう。商品種類リストでの商品種類の検索をモデルに、分離連鎖法を使って実装してみます。また、今回も継承ではなく委譲モデルで実装します。クラス図を以下に示します。



13.3.1.　汎用的なハッシュテーブルの実装

汎用的なハッシュテーブルの API は次のように設計します。

void put(Object key,Object value)	key と value の組を追加する
Object get(Object key)	与えられた key で value を検索する
void remove(Object key)	与えられた key により value の組を削除する
Object[] keys()	商品種類リストなどが display()メソッドを実装するために、格納するすべての key を取得する

汎用的なハッシュテーブルのリストを示します。

例題 13-3：汎用的なハッシュテーブル(LinkObject.java)

```
1:      /**
2:      * オブジェクト指向哲学　入門編
3:      * 例題 13-3：汎用的なハッシュテーブル
4:      * 商品種類を名前で検索するプログラム
5:      *
6:      * 連結リストインスタンスクラス
7:      */
8:      public class LinkObject {
9:
10:       private Object key;        //キー
11:       private Object value;      //値
```

```
12:     private LinkObject next;  //次の要素への参照
13:
14:     /**
15:     * コンストラクタ
16:     */
17:     public LinkObject(Object newKey,Object newValue){
18:       key = newKey;
19:       value = newValue;
20:     }
21:
22:     /**
23:     * キーを取得する
24:     */
25:     public Object getKey(){
26:       return key;
27:     }
28:
29:     /**
30:     * 値を取得する
31:     */
32:     public Object getValue(){
33:       return value;
34:     }
35:
36:     /**
37:     * 次の要素を取得する
38:     */
39:     public LinkObject getNext(){
40:       return next;
41:     }
42:
43:     /**
44:     * 次の要素を設定する
45:     */
46:     public void setNext(LinkObject newLink){
47:       next = newLink;
48:     }
49:
50:   }
```

例題 13-3：汎用的なハッシュテーブル(ObjectHashTable.java)

```
1:    /**
2:    * オブジェクト指向哲学　入門編
3:    * 例題 13-3：汎用的なハッシュテーブル
4:    * 商品種類を名前で検索するプログラム
5:    *
6:    * インスタンスハッシュテーブルクラス
7:    * Object を格納する汎用的なハッシュテーブル
8:    * 分離連鎖法により実装されている
```

```
9:     */
10:    public class ObjectHashTable {
11:
12:      private int ARRAY_SIZE = 5;                 //用意する配列の最大サイズ
13:      private LinkObject[] itemTypeLinkArray;    //連結リストの先頭になる配列を用意する
14:      private int size = 0;                       //HashTable に格納されたインスタンスの
数
15:
16:      /**
17:      * コンストラクタ
18:      */
19:      public ObjectHashTable() {
20:        itemTypeLinkArray = new LinkObject[ARRAY_SIZE];//配列の初期化
21:      }
22:
23:      /**
24:      * キーと値のペアをハッシュテーブルに格納する
25:      */
26:      public void put(Object key,Object value){
27:        //ハッシュ値を求める
28:        int hashCode = key.hashCode();
29:        int arrayLoc = hashCode % ARRAY_SIZE;
30:
31:        //追加する
32:        LinkObject insertLink = new LinkObject(key,value);
33:        if (itemTypeLinkArray[arrayLoc] == null){
34:          //ハッシュした番地が未使用だったとき->そのまま追加
35:          itemTypeLinkArray[arrayLoc] = insertLink;
36:          size++;//追加したので HashTable のサイズを一つ増やす
37:          return;
38:
39:        }else{
40:          //ハッシュした番地が使用済みだったとき->連結リストに追加
41:          LinkObject current = itemTypeLinkArray[arrayLoc];//現在検索中のリンクを設定
する
42:
43:          //連結リストの最後を探す
44:          while (true){
45:            if(current.getNext() == null){
46:              //連結リストの最後が見つかった
47:              current.setNext(insertLink);//最後に連結する
48:              size++;//追加したので HashTable のサイズを一つ増やす
49:              return;//連結完了
50:
51:            }else{
52:              //見つからない
53:              current = current.getNext();//次のリンクを検索中のリンクに設定
54:            }
55:          }
56:        }
57:
58:      }
59:
60:      /**
```

```
61:       * key に対応する Object を取得する
62:       *  （検索する）
63:       */
64:       public Object get(Object key){
65:        //実装するのが課題です
66:        return null;
67:       }
68:
69:       /**
70:       * key に対応する Object を削除する
71:       */
72:       public void remove(Object key){
73:         //ハッシュ値を求める
74:         int hashCode = key.hashCode();
75:         int arrayLoc = hashCode % ARRAY_SIZE;
76:
77:         LinkObject current = itemTypeLinkArray[arrayLoc];//現在検索中のリンクインスタ
ンス
78:         LinkObject prev = null;//現在検索中の一つ前のリンクインスタンス
79:
80:         //ハッシュした番地が使用されているとき
81:         while (current!=null){
82:           if(current.getKey().equals(key)){
83:             //削除対象が見つかった
84:             size--;
85:             if(prev==null){
86:               //連結リストの先頭だったとき
87:               if(current.getNext()== null){
88:                 //次の要素がないとき
89:                 itemTypeLinkArray[arrayLoc] = null;//対象の番地のリンクインスタンスを
削除する
90:                 break;
91:               }else{
92:                 //次の要素があるとき
93:                 itemTypeLinkArray[arrayLoc] = current.getNext();//次の要素へ配列からリ
ンクをつくる
94:                 break;
95:               }
96:             }else if(current.getNext()==null){
97:               //連結リストの最後だったとき
98:               prev.setNext(null);//削除するリンクインスタンスへのリンクを消す
99:               break;
100:            }else{
101:              //連結リストの最初でも最後でもないとき
102:              //削除するリンクインスタンスの一つ前のリンクインスタンスから,削除するリ
ンクインスタンスの次のリンクインスタンスへ
103:              //リンクをはる
104:              prev.setNext(current.getNext());
105:              break;
106:            }
107:          }else{
108:            prev = current;//現在ののリンクインスタンスを一つ前のリンクインスタンスに
設定
109:            current = current.getNext();//次のリンクインスタンスを検索中のリンクインス
```

```
タンスに設定
110:          }
111:        }
112:
113:      }
114:
115:      /**
116:      * HashTable に格納されている key を全て取り出す
117:      */
118:      public Object[] keys(){
119:        Object[] keys = new Object[size];//現在格納されている Key の数だけ配列を用意する
120:        int index = 0;                          //Key を配列に格納するためのインデックス
121:
122:        LinkObject current;//現在検索中のリンクインスタンス
123:
124:        for(int i=0; i<ARRAY_SIZE; i++){
125:          current = itemTypeLinkArray[i];//検索中のリンクインスタンスを連結リストの先頭に設定する
126:
127:          while(current != null){//リンクインスタンスがあるとき
128:            keys[index] = current.getKey();//key を配列に格納
129:            index++;                          //index を次に進める
130:            current = current.getNext();//リンクを次へ進める
131:          }
132:          if(size==index){
133:            break;
134:          }
135:        }
136:        return keys;//key の配列を返す
137:      }
138:
139:    }
```

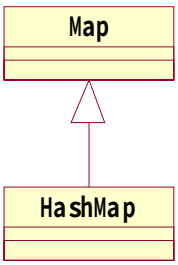
例題 13-3：汎用的なハッシュテーブル(ItemTypeList.java)

```
1:      /**
2:      * オブジェクト指向哲学　入門編
3:      * 例題 13-3：汎用的なハッシュテーブル
4:      * 商品種類を名前で検索するプログラム
5:      *
6:      * 商品種類リストクラス
7:      */
8:      public class ItemTypeList {
9:
10:       private ObjectHashTable itemTypeHashTable;//商品種類を格納するための HashTable
11:
12:       /**
13:       * コンストラクタ
14:       */
```

```
15:       public ItemTypeList() {
16:         itemTypeHashTable = new ObjectHashTable();//初期化
17:       }
18:
19:       /**
20:       * 新たに商品種類を追加する
21:       */
22:       public void add(ItemType addItemType){
23:         itemTypeHashTable.put(addItemType.getName(),addItemType);//商品名を Key にして
商品種類を格納する
24:       }
25:
26:       /**
27:       * 商品種類を商品名をキーにして検索する
28:       */
29:       public ItemType search(String key){
30:         return (ItemType)itemTypeHashTable.get(key);//与えられた Key に対応する商品種類
を返す
31:       }
32:
33:       /**
34:       * 指定されたキーを持つ商品種類をリストから削除する
35:       */
36:       public void remove(String key){
37:         itemTypeHashTable.remove(key);//Key に対応する商品種類を削除
38:       }
39:
40:       /**
41:       * 商品種類を表示する
42:       */
43:       public void display(){
44:         Object[] keys = itemTypeHashTable.keys();//HashTable の Key を全て取得する
45:         ItemType showItemType;                        //表示する商品種類
46:
47:         for(int i=0; i<keys.length; i++){
48:           showItemType = (ItemType)itemTypeHashTable.get(keys[i]);//i 番目の Key に対応
する商品種類を取り出す
49:           System.out.println(showItemType.getName()+":"+showItemType.getPrice()+"
円:");
50:         }
51:       }
52:     }
```

13.3.2.　JavaAPI を利用する

ハッシュテーブルもよく使われるデータ構造なので、JavaAPI が用意されています。名前は少し異なるので注意してください。key と value でデータを格納するクラスのインターフェイスを「Map」といいます。実は key と value でデータを格納する方法がハッシュテーブル以外にも在るためです。ハッシュテーブルで実装された「Map」クラスは「HashMap」というクラスです。以下にクラス図を示します。



①.Map クラスの API

void put(Object key,Object value)	key と value の組を追加する
Object get(Object key)	与えられた key で value を検索する
void remove(Object key)	与えられた key により value の組を削除する
Set keySet()	すべての key を Set というインターフェイス型で取得する

※　Set とは重複のないリストのことで、これも JavaAPI の一部です。この Set クラスの API いは、Object[] toArray()というメソッドがありますので、ObjectHashtable の keys()メソッドと同じように利用することができます。

練習問題

＜記述問題＞

☆記述問題 13-1

汎用的なハッシュテーブルを利用する時の注意点を挙げよ

＜プログラム問題＞

☆プログラム問題 13-1

例題 13-3 における汎用的なハッシュテーブル(ObjectHashtable.java)は、get メソッドが実装されていない。実装して完成させよ。

☆プログラム問題 13-2

プログラム問題 13-1 を JavaAPI のハッシュテーブル(HashMap)を利用して書き直せ。