第8章 データ構造(1)

配列、連結リスト、ハッシュテーブル

畠野 和裕
PAS社 車載システムズ事業部 SSBU6-2

目次

8.1 データ構造を学ぶ意義

- データ構造とは？
  - データの持ち方のこと
  - 読み込んだ値や計算中に求めた値をデータ構造という形で保持して，必要に応じてデータ構造から所望の値を取り出すことは多々ある
  - Pythonなら、
    - list
    - dict
    - set
    - …

8.1 データ構造を学ぶ意義

- クエリ（query）とは？
  - データ構造への要求
  - 3つのタイプ
    - 要素xをデータ構造に挿入する
    - 要素xをデータ構造から削除する
    - 要素xがデータ構造に含まれるかを判定する

表 8.1　各データ構造の各クエリに対する計算量

	配列	連結リスト	ハッシュテーブル
C++ でのライブラリ	vector	list	unordered_set
Python でのライブラリ	list	-	set
i 番目の要素へのアクセス	$O(1)$	$O(N)$	-
要素 x を挿入	$O(1)$	$O(1)$	$O(1)$
要素 x を特定の要素の直後に挿入	$O(N)$	$O(1)$	$O(1)$
要素 x を削除	$O(N)$	$O(1)$	$O(1)$
要素 x を検索	$O(N)$	$O(N)$	$O(1)$

各データ構造で得意/不得意なクエリがあるので適切に使い分けることが大切

8.2 配列

- 配列
  - 要素を１列に並べて各要素に容易にアクセスできるようにしたデータ構造
    - C++：std::vector
    - Python：list
  - 得意なこと
    - データa[i]にアクセス（O(1)）
  - 苦手なこと
    - 要素xを要素yの直後に挿入（O(N)）
    - 要素xを削除（O(N)）

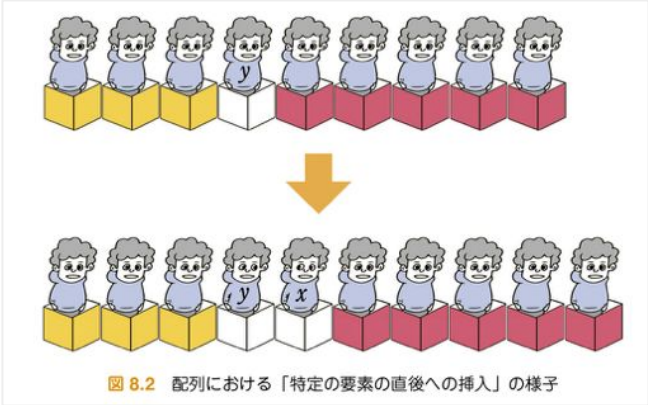


図 8.2　配列における「特定の要素の直後への挿入」の様子

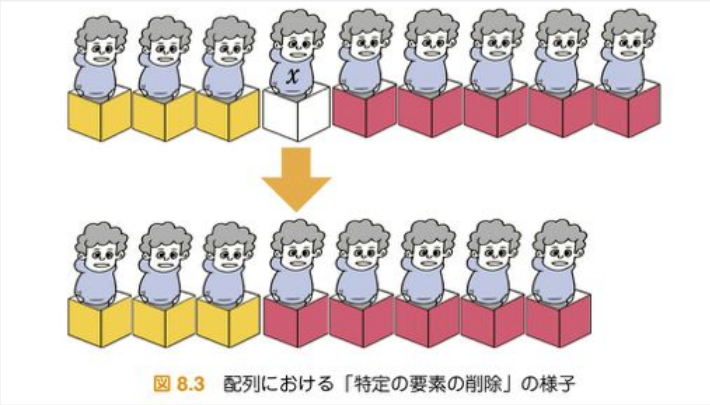


図 8.3　配列における「特定の要素の削除」の様子

8.3 連結リスト

- 連結リスト
  - 各要素（ノード）をポインタ（矢印）で1列に繋いだもの
  - 得意なこと：配列が苦手なこと
    - 要素xを要素yの直後に挿入（O(1)）
    - 要素xを削除（O(1)）

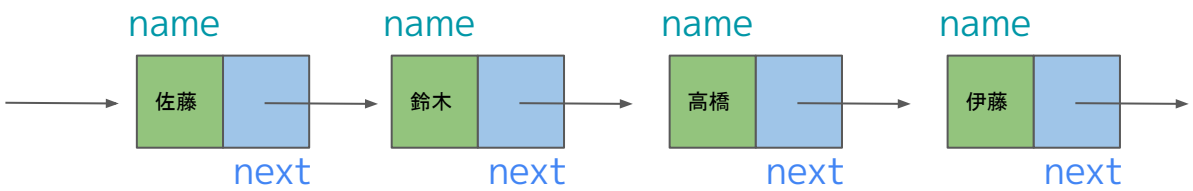
図 8.4 連結リストの様子

8.3 連結リスト

- 連結リスト
  - 各要素（ノード）をポインタ（矢印）で1列に繋いだもの
  - 自己参照構造体
    - 自分自身の型へのポインタをメンバにもつ構造体
    - 連結リストの１つ１つのノードを，自己参照構造体のインスタンスで表現

code 8.2　自己参照構造体

```
1  struct Node {
2      Node* next; // 次がどのノードを指すか
3      string name; // ノードに付随している値
4
5      Node(string name_ = "") : next(NULL), name(name_) { }
6  };
```

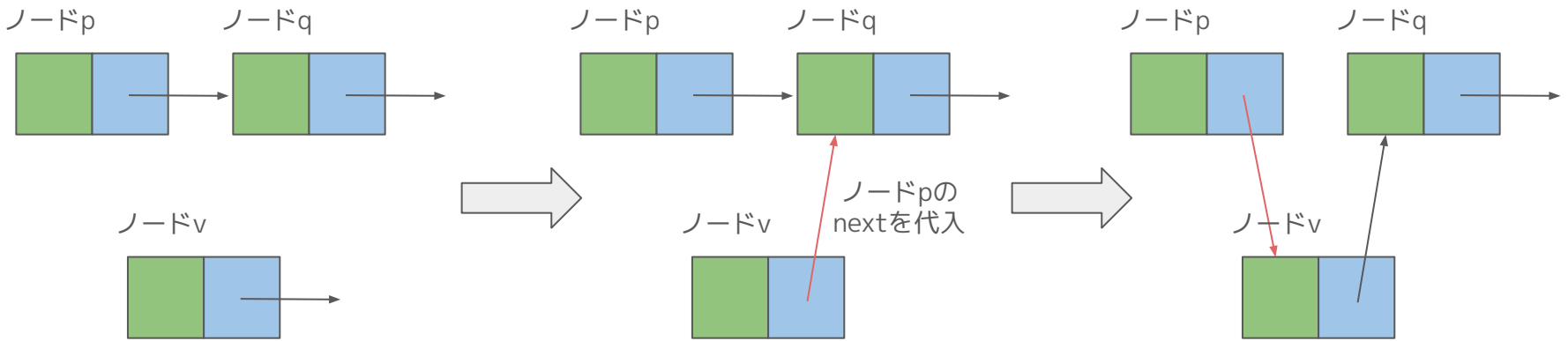


8.4 連結リストの挿入操作と削除操作

- 8.4.1 連結リストの挿入操作
  - ある特定の要素の直後に他の要素を挿入するには？
  - ポインタを繋ぎかえる

code 8.3　連結リストの挿入操作

```
1  // ノード p の直後にノード v を挿入する
2  void insert(Node* v, Node* p) {
3      v->next = p->next;
4      p->next = v;
5  }
```



8.4 連結リストの挿入操作と削除操作

- 8.4.2 連結リストの削除操作
  - ある特定の要素を削除するには？

図 8.6　リストにおける「所定の要素の削除」の様子

例：渡辺ノードを消したい
→渡辺ノードの1つ前の、
　伊藤ノードを山本ノードに繋ぎかえればOK

つまり、ある特定のノードを削除したいときは、
削除したいノードの前のノードを取得できるようにする必要がある

しかし、現状の方法だと、
削除したい（渡辺）ノードから
1つ前の（伊藤）ノードを直接参照できない

8.4 連結リストの挿入操作と削除操作

- 8.4.2 連結リストの削除操作
  - ある特定の要素を削除するには？

- 単方向連結リスト
  - 次のノードをつなぐポインタのみもつ
    - 1つ前のノードに直接アクセスできない
  - 挿入操作：○
  - 削除操作：△

図 8.4 連結リストの様子

- 双方向連結リスト
  - 各ノードをつなぐポインタを双方向に
    - 1つ前のノードにも直接アクセス可能
  - 挿入操作：○
  - 削除操作：○

図 8.7 双方向連結リストの概念図

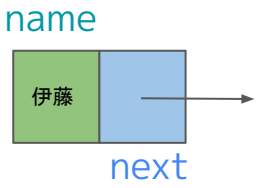
8.4 連結リストの挿入操作と削除操作

- 8.4.2 連結リストの削除操作
  - ある特定の要素を削除するには？

- 単方向連結リスト

code 8.2　自己参照構造体

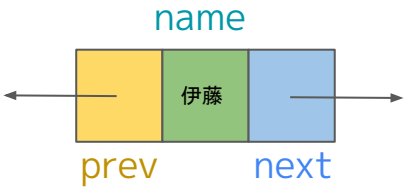
```
1  struct Node {
2      Node* next; // 次がどのノードを指すか
3      string name; // ノードに付随している値
4
5      Node(string name_ = "") : next(NULL), name(name_) { }
6  };
```



- 双方向連結リスト

code 8.5　双方向への自己参照構造体

```
1  struct Node {
2      Node *prev, *next;
3      string name; // ノードに付随している値
4
5      Node(string name_ = "") :
6      prev(NULL), next(NULL), name(name_) { }
7  };
```



8.4 連結リストの挿入操作と削除操作

- 8.4.2 連結リストの削除操作
  - ある特定の要素を削除するには？

図 8.6　リストにおける「所定の要素の削除」の様子

図 8.9　双方向連結リストにおける削除操作

8.5 配列と連結リストの比較

- 配列
  - i 番目の要素にアクセスするのが得意
- 連結リスト
  - 要素を挿入したり、削除するのが得意

表 8.2　配列と連結リストの比較

クエリ	配列	連結リスト	備考
i 番目の要素へのアクセス	$O(1)$	$O(N)$	
要素 x を最後尾へ挿入	$O(1)$	$O(1)$	
要素 x を特定の要素の直後に挿入	$O(N)$	$O(1)$	連結リストでは，特定のノード p を指定すれば，p の直後への挿入処理を $O(1)$ の計算量で実現できます．
要素 x を削除	$O(N)$	$O(1)$	ただし連結リストにおいて，特定の要素 x 自体を探索する必要がある場合には，その探索に $O(N)$ の計算量がかかります．
要素 x を検索	$O(N)$	$O(N)$	3 章で解説した線形探索法を適用します．

8.5 配列と連結リストの比較

- 配列
  - i 番目の要素にアクセスするのが得意
- 連結リスト
  - 要素を挿入したり、削除するのが得意

表 8.2　配列と連結リストの比較

クエリ	配列	連結リスト	備考
i 番目の要素へのアクセス	$O(1)$	$O(N)$	
要素 x を最後尾へ挿入	$O(1)$	$O(1)$	
要素 x を特定の要素の直後に挿入	$O(N)$	$O(1)$	連結リストでは，特定のノード p を指定すれば，p の直後への挿入処理を $O(1)$ の計算量で実現できます．
要素 x を削除	$O(N)$	$O(1)$	ただし連結リストにおいて，特定の要素 x 自体を探索する必要がある場合には，その探索に $O(N)$ の計算量がかかります．
要素 x を検索	$O(N)$	$O(N)$	3 章で解説した線形探索法を適用します．

どちらもO(N)

```cpp
// C++
if (find(a.begin(), a.end(), x) != a.end()) {
    (処理)
}
```

```python
# Python
if x in a:
    (処理)
```

安易に使うのは注意

8.6 ハッシュテーブル

- 8.6.1 ハッシュテーブルの考え方
  - データ集合S の要素x に対し、0 ≦ h(x) < M を満たす整数 h(x) に対応させたもの
    - ある整数nと要素xを対応づけるとき、n = h(x)
    - x：キー
    - h(x)：ハッシュ関数
    - n：ハッシュ値

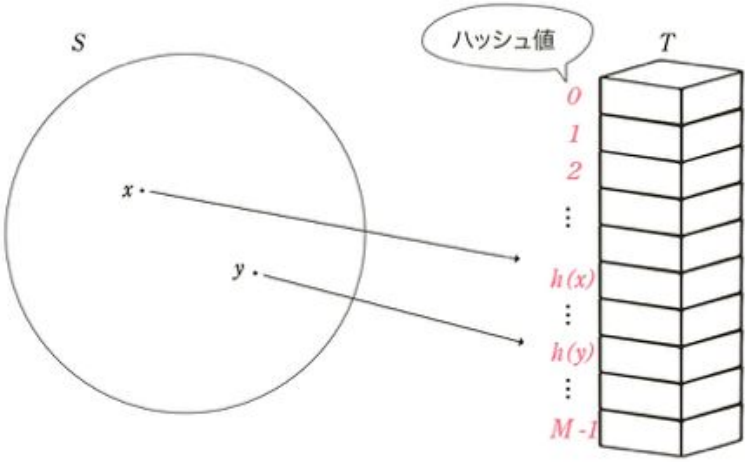


図 8.10　ハッシュテーブルの考え方

8.6 ハッシュテーブル

- 8.6.1 ハッシュテーブルの考え方
  - 完全ハッシュ関数
    - どのキーに対しても、ハッシュ値が異なるハッシュ関数
      - n = h(x)、n = h(y) のようなハッシュ値の衝突が起きない
    - 挿入、削除、検索全ての処理が、O(1)で可能

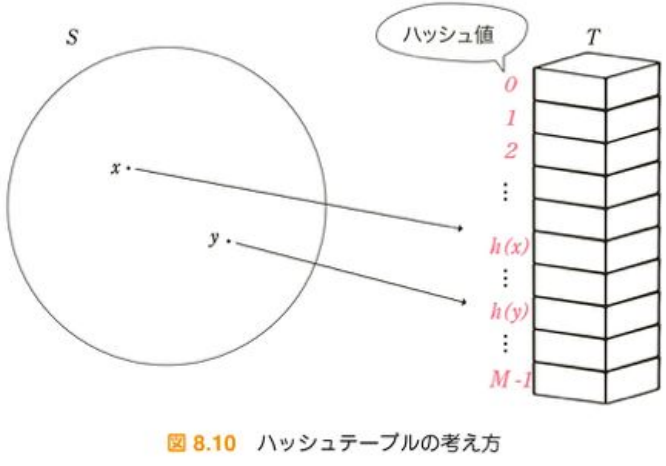


図 8.10 ハッシュテーブルの考え方

表 8.4 完全ハッシュ関数が設計できた場合の，ハッシュテーブルにおける挿入・削除・検索クエリ処理

クエリ	計算量	実装
要素 x の挿入	$O(1)$	$T[h(x)] \leftarrow$ true
要素 x の削除	$O(1)$	$T[h(x)] \leftarrow$ false
要素 x の検索	$O(1)$	$T[h(x)]$ が true かどうか

この例では、
- キーが存在していれば、ハッシュ値がTrue、
- キーが存在していなければ、ハッシュ値がFalse、

として定義

8.6 ハッシュテーブル

- 8.6.2 ハッシュの衝突対策
  - 現実的には、完全ハッシュ関数を設計することは困難
  - 衝突
    - 異なるキーに対してハッシュ値が等しくなること
  - 対策例
    - ハッシュ値ごとに連結リストを構築する方法

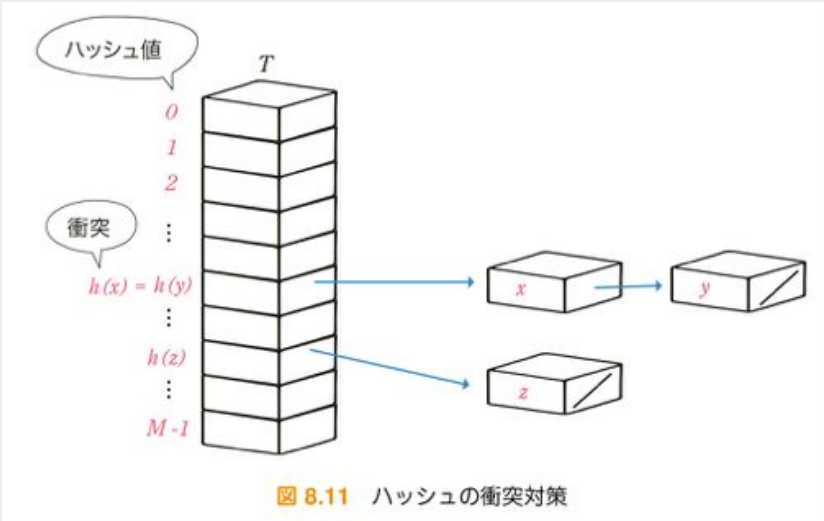


図 8.11 ハッシュの衝突対策

あるハッシュ値ごとに
連結リストを構築する

8.6 ハッシュテーブル

- 8.6.3 ハッシュテーブルの計算量
  - 最悪のケース
    - O(N)
    - N個のキー全てで衝突が起きるケース
  - ハッシュ関数が十分な性能を持つとき
    - O(1+N/M)
      - M：ハッシュ値の種類
    - N/M=1/2程度であれば、O(1)の計算量を達成可能

8.6 ハッシュテーブル

- 8.6.4 C++やPythonにおけるハッシュテーブル
  - C++
    - std::unordered_set
    - std::set
  - Python
    - 集合型set

code 8.7 C++ におけるハッシュテーブルの挿入・削除・検索クエリ処理

```
// 要素 x の挿入
a.insert(x);

// 要素 x の削除
a.erase(x);

// 要素 x の検索
if (a.count(x)) {
    (処理)
}
```

code 8.8 Python におけるハッシュテーブルの挿入・削除・検索クエリ処理

```
# 要素 x の挿入
a.add(x);

# 要素 x の削除
a.remove(x)

# 要素 x の検索
if x in a:
    (処理)
```

8.6 ハッシュテーブル

- 8.6.5 連想配列
  - 通常の配列
    - 非負の整数値のみ添字に取れる
    - a[i], i=0~n
  - 連想配列
    - 非負の整数値以外も含めて、添字に取れるようにしたもの
    - a[1], a[-1], a[“cat”]
  - 使い方
    - C++
      - std::unordered_map
      - std::map
    - Python
      - 辞書型dict

8.7 まとめ

- 基本的なデータ構造
  - 配列
  - 連結リスト
  - ハッシュテーブル
- 処理したいクエリ内容に応じて，適切なデータ構造を用いることが大切