データ構造

データ構造とアルゴリズム

- 少ないメモリで速い処理を行いたい
- データ構造を工夫することでアルゴリズムを高速化
- データ構造とアルゴリズムは密接に関係

データ集合

- データの集まり（データセット）
- 種類
  - 同じ属性のデータの集まり ➡️ 配列，リスト
    - 属性が単一（Numpy配列）
    - 属性が複数（Pythonリスト）
  - 違う属性のデータの集まり ➡️ タプル

データ集合への典型的な操作

- クエリ（問い合わせ）
  - 先頭から順番にN個のデータを読み取る（シーケンシャルアクセス）
  - 先頭からi番目のデータを一つ読み取る（ランダムアクセス）
  - 新しいデータxを追加する
  - データxを削除する

典型的なデータ構造

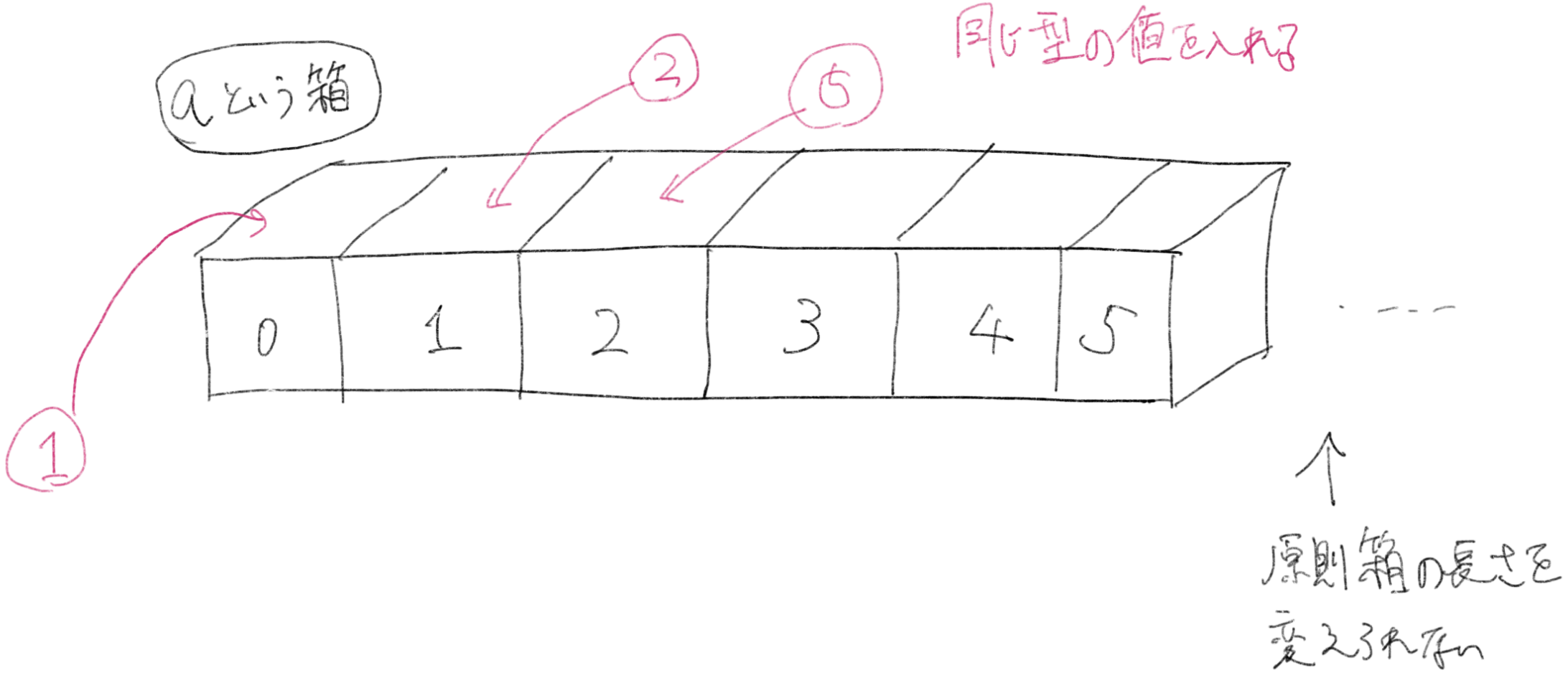
（同じ属性データの集まり）

- 配列
  - 同じメモリ上に連続的に配置
- 線形リスト
  - 次の要素を示すポインタを持つ
- ハッシュテーブル
  - キーの値を数値化した場所にデータを格納

用途によって，時間計算量，空間計算量が変わるため適切な使い分けが重要

配列

- 大きな箱に仕切りをつける（同じ型の変数が一列にならぶ）
- 固定長配列：最初に準備した箱の大きさを変えられない（Numpy配列）
- 可変長配列：箱の大きさを変えられる（Pythonリスト）


同じ型の値を入れる
aという箱
2
6
1
0
1
2
3
4
5
原則箱の長さを
変えられない

計算量

クエリ	計算量
シーケンシャルアクセス	$O(N)$
ランダムアクセス	$O(1)$
データ挿入（最後尾に追加）	可変長配列 $O(1)$
データ削除	可変長配列 $O(N)$

用途

- あらゆる集合型データの基本
- 非常に汎用的に使えるデータ構造
- データの途中挿入や削除が頻繁に発生する場合には別のデータ構造を検討すべき

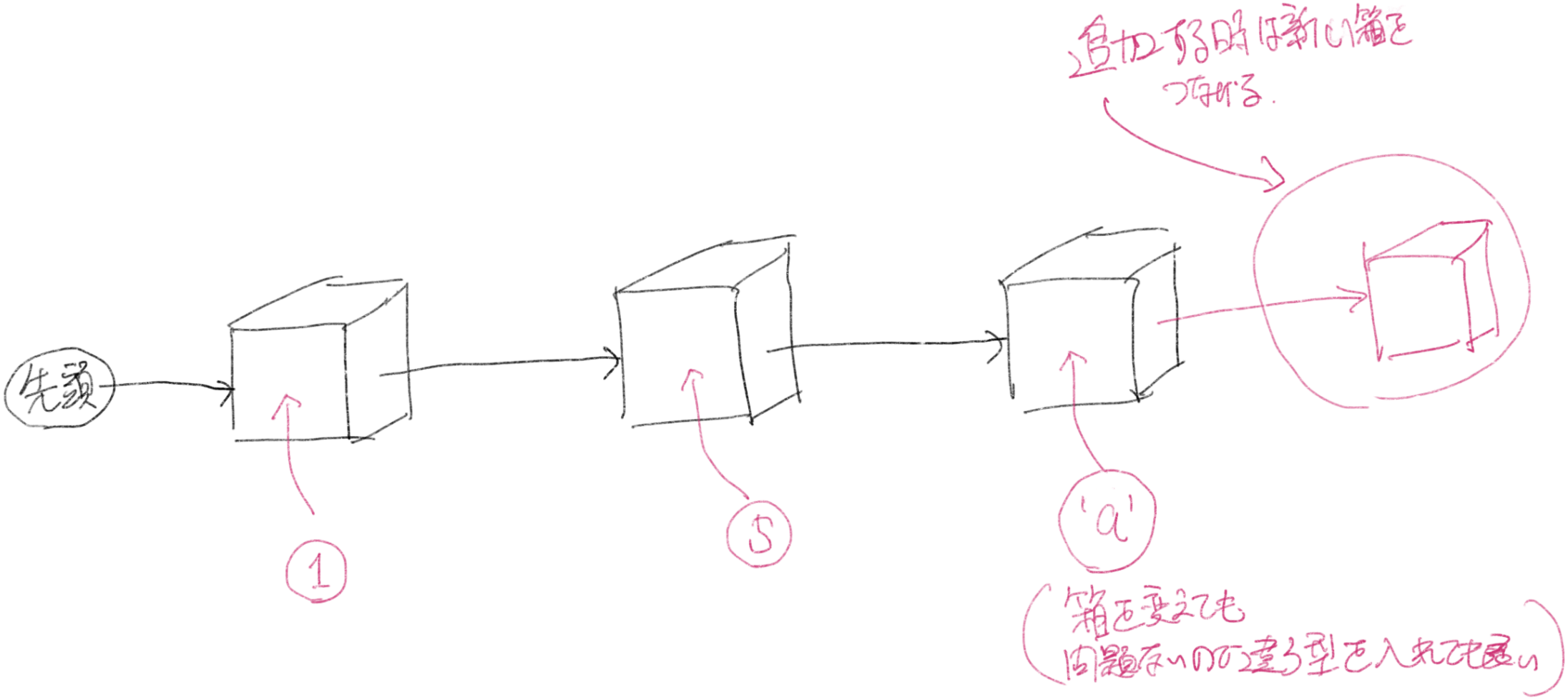
```python
import numpy as np

x = [4, 2, 3, 7, 2] # pythonリスト（可変長配列）
x[3] # ランダムアクセス
x.append(10) # Pythonリストへ要素の追加
x.extend([1, 2, 3]) # 複数要素の追加
x.insert(0, 100) # 0要素に100を挿入（以降はインデックスがずれる）

y = np.array(x) # numpy配列（固定長配列）
y[3] # ランダムアクセス
# numpy配列はデータ挿入・削除は得意ではない
```

線形リスト

- 一つの要素毎に箱を準備
- 箱に格納する情報
  - データ
  - 次の箱がどこにあるかを示す情報
- Pythonリストは線形リストではない
- Pythonでは deque を使う


追加する時は新しい箱を
つなげる.
先頭
1
S
'a'
(箱を変えても
問題ないので違う型を入れても良い

計算量

クエリ	計算量
シーケンシャルアクセス	$O(N)$
ランダムアクセス	$O(N)$
データ挿入（最後尾に追加）	$O(1)$
データ削除	$O(1)$

用途

- データの挿入・削除に強い
- ランダムアクセスに弱いため，データの挿入・削除があるがランダムアクセスが必要なデータにはハッシュテーブルの利用を検討する
- スタック，キュー，ハッシュテーブルなどの他のより抽象的なデータ構造に利用されている

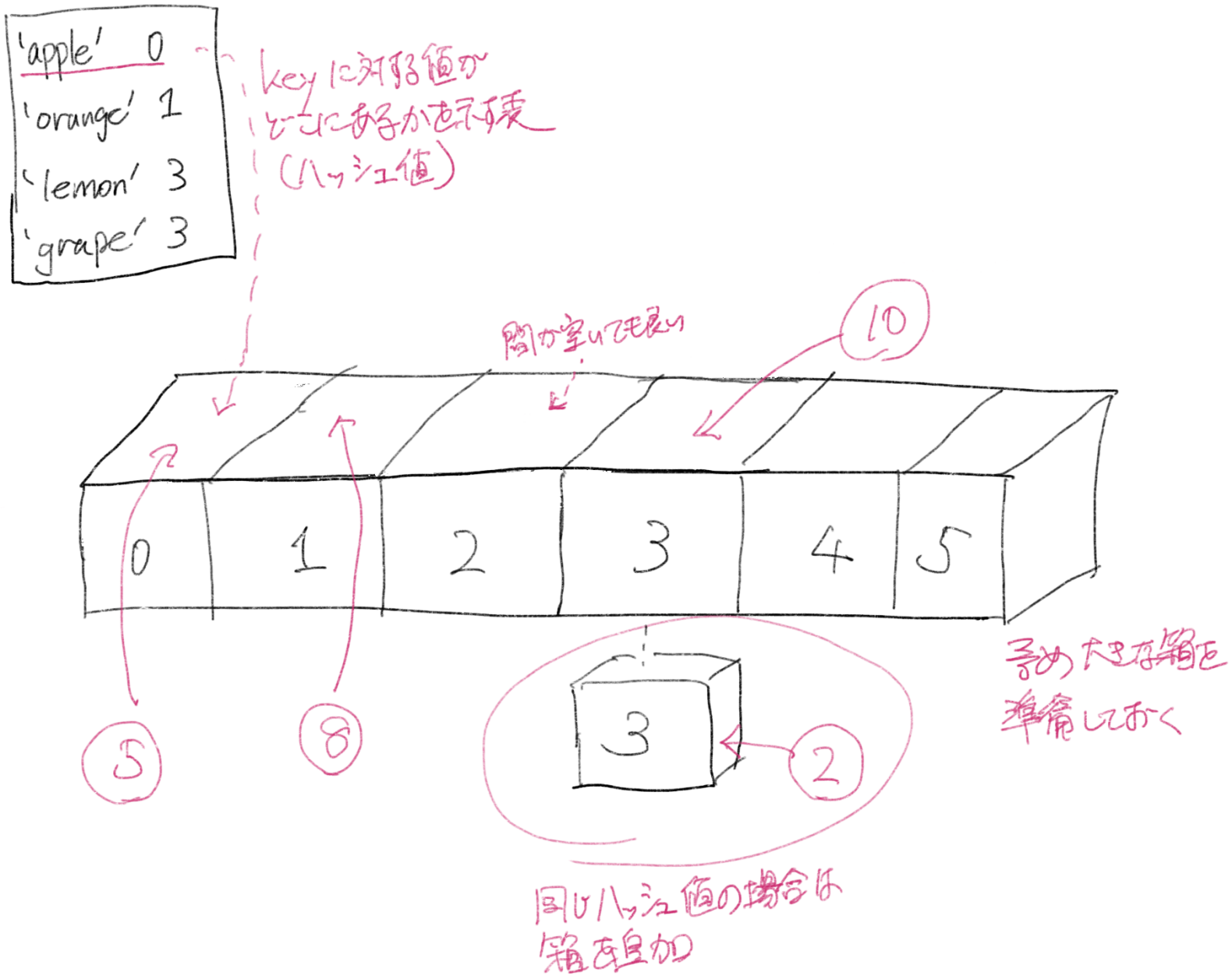
```python
from collections import deque

x = deque([4, 2, 3, 7, 2]) # 両端queue
x[3] # ランダムアクセスは苦手
x.append(10) # 要素の追加
x.extend([1, 2, 3]) # 複数要素の追加
a = x.pop() # 一番後ろの要素を取得・リストから削除
b = x.popleft() # 一番前の要素を取得・リストから削除

# シーケンシャルアクセス
for v in x:
  print(v)
```

ハッシュテーブル

- 大きな箱に仕切りをつける（同じ型の変数が一列にならぶ）
- 箱にラベルがついている
- 箱に複数の要素を入れることも可能
- PythonのDict


'apple' 0
'orange' 1
'lemon' 3
'grape' 3
keyに対する値が
どこにあるかを示す
(ハッシュ値)
間が空いてもOK
10
0
1
2
3
4
5
5
8
3
2
多めに大きな箱を
準備しておく
同じハッシュ値の場合は
箱を追加

計算量

クエリ	計算量
シーケンシャルアクセス	$O(N)$
ランダムアクセス	$O(1)$
データ挿入（最後尾に追加）	$O(1)$
データ削除	$O(1)$

用途

- ランダムアクセス・データの挿入・削除のいずれにも強い
- あらかじめデータを格納する領域を多く必要する
- キーから計算されるラベル付け（ハッシュ値）により性能が異なる

```python
x = {'apple': 4, 'banana': 2, 'chocolate': 3, 'dessert': 7, 'eat': 2}
x['banana'] # ランダムアクセス
x['fresh'] = 10 # 要素の追加
x.pop('apple') # 要素の削除. del でも削除できる

# シーケンシャルアクセス（定義した順番に並んでいるとは限らない）
for k in x:
  print(k, x[k])

for k,v in x.items():
  print(k,v)
```