

アルゴリズム基礎

《学修項目》

- アルゴリズムと表現方法
- データの並べ替え（ソートアルゴリズム）
- データの探索（サーチアルゴリズム）

《キーワード》

アルゴリズム、擬似コード、表現方法、フローチャート、計算量、ソートアルゴリズム、バブルソート、挿入ソート、マージソート、選択ソート、クイックソート、探索アルゴリズム、リスト探索、線形探索、木構造、二分探索

《参考文献、参考書籍》

- [1] 東京大学MIセンター公開教材 「1-7 アルゴリズム基礎」 《利用条件CC BY-NC-SA》
- [2] 応用基礎としてのデータサイエンス（講談社 データサイエンス入門シリーズ）
- [3] Pythonで学ぶアルゴリズムとデータ構造（講談社 データサイエンス入門シリーズ）
- [4] Pythonによるあたらしいデータ分析の教科書 第2版（翔泳社）
- [5] 数理・データサイエンス・AI公開講座（放送大学）

1. アルゴリズムと表現方法

1.1 アルゴリズム [1][2]

コンピュータで何らかの問題を解きたいと思ったら、プログラムを作る必要がある。現在広く使われている構造化プログラミング言語は、順次実行、条件分岐、繰り返しを基本的な構成要素としている。これらの構成要素の有限個の集まりを使って、課題となっている問題を有限ステップ以内で解き、値を出力する手順を考える必要がある。アルゴリズム(algorithm)とは、ある問題を解くことができる一連の手順のことを指す[2]。

1.1.1 ユークリッドの互除法

2つの数の最大公約数を求めるアルゴリズムとして、ユークリッドの互除法が知られている。

自然数 a と b について、最大公約数 $\gcd(a, b)$ を求めるための手順は以下の通り：

1. a を b で割った余りを r とする
2. r が 0 ならば、 b を 最大公約数として出力して終了する
3. $a = b$, $b = r$ を代入して、1.に戻る

ユークリッドの互除法の計算量は、 $a \geq b$ としたとき、 $O(\log b)$ である。対数オーダーであり、非常に高速である。

1.2 アルゴリズムの表現方法 [2]

1.2.1 擬似コード

疑似コードとは、アルゴリズムや関数などのコードを自然言語とプログラミング言語の要素を組み合わせたものである。実際には実行できないことから、「疑似」コードと呼ばれる。

コーディングのロジックを説明したり、他のメンバーも交えて計画を立てる際に有用である。誰もが理解しやすい方法でプログラムの手順を記述しつつ、特定のプログラミング言語に後で変換できるよう、ある程度まで詳細化する。

疑似コードの簡単な例として、サイトやアプリの訪問者に対して、名前付きのメッセージを表示する基本的なロジックを以下に示す[*]。

```
PROCESS GreetUser
    INPUT userName
    DISPLAY "Hello, " + userName + "!"
END
```

PROCESS（処理）、DISPLAY（表示）、+のような、簡単な言葉やプログラミング要素で構成し、プラットフォームに独立した形で自然語として誰でも理解できるようになっている。

- [Kinsta | 疑似コードとは](#)

1.2.2 フローチャート

自然語や擬似コードではなく、アルゴリズムを図で表現する方法がある。下図はユークリッドの互除法をフローチャート (flowchart) で表現したものである。

フローチャートは、長方形が処理、ひし形が条件判定を表現する。簡単なアルゴリズムをフローチャートで記述できると視覚的にわかりやすいが、アルゴリズムが複雑になるとフローチャートも複雑になるため、最近ではこれを用いて設計する機会は非常に少ない。それに代わって、UMLふるまい図（アクティビティ図、シーケンス図、あるいはステートマシン図）を用いて上位設計を行うことがある。

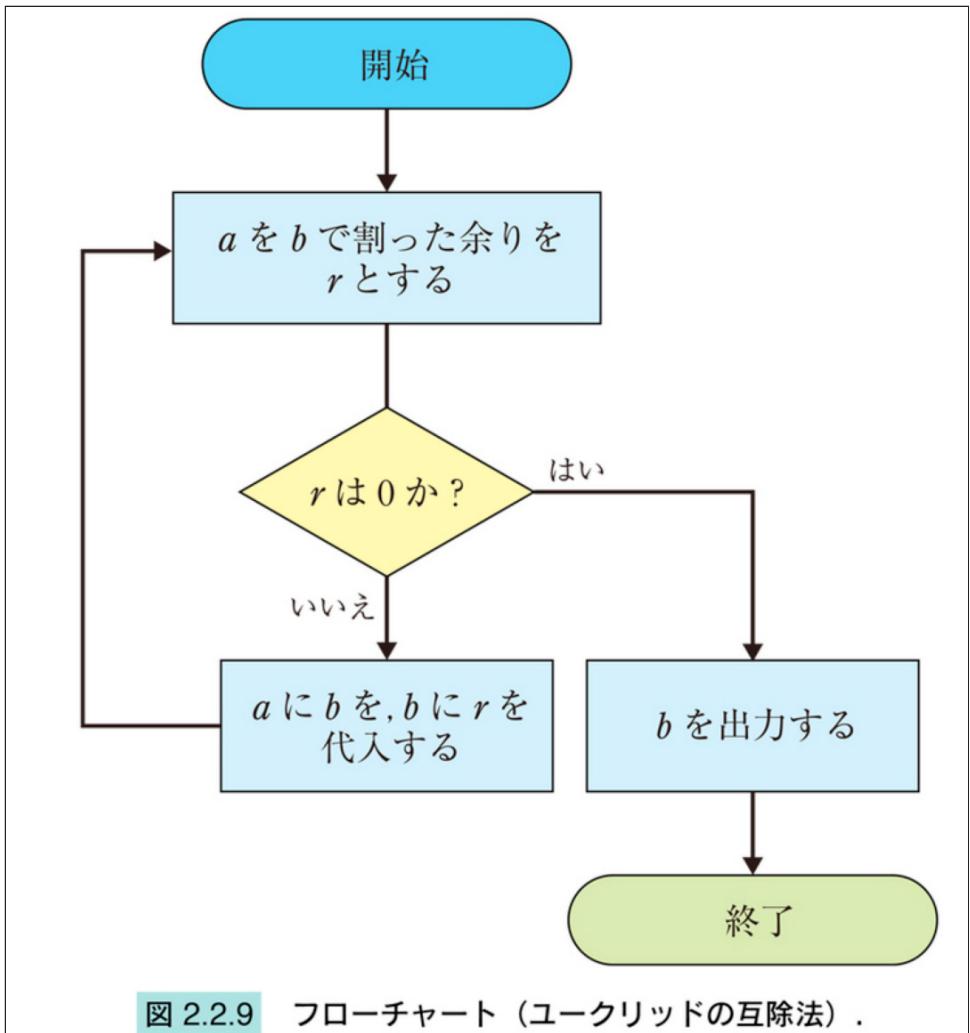


図 2.2.9 フローチャート（ユークリッドの互除法）.

ユークリッドの互除法 フローチャート ([2]より引用)

2. データの並べ替え（ソートアルゴリズム）

2.1 Pythonのソート関数

数値、文字列、日付や日時でデータを並べ替えることはよくある。これはソート(sort)と呼ばれる。多くのプログラミング言語では、sorted関数, sortメソッドでソートの機能が提供されている[*]。現在のPython実装のsort機能は TimSortで大規模データにおいて高速である。

- [Python Documentation - sort HOW TO](#)

ソートを目的とした様々なアルゴリズムが提案してきた。バブルソート、挿入ソート、選択ソート、マージソート、クイックソートなどがある。

```
In [1]: # Python ソート関数によるインプレース操作
sorted([5, 2, 3, 1, 4])
```

```
Out[1]: [1, 2, 3, 4, 5]
```

```
In [2]: # operator モジュール関数を使った操作(安定ソート)
from operator import itemgetter, attrgetter
data = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4)]
sorted(data, key=itemgetter(0))
```

```
Out[2]: [('blue', 2), ('blue', 4), ('red', 1), ('red', 3)]
```

```
In [3]: # numpyによる整数配列の生成とソート(ソート手法はデフォルトでquick sort)
import numpy as np
a = np.random.randint(0, 100, size=20)
print(a)
print(np.sort(a))
```

```
[26 60 7 97 28 97 42 23 18 71 76 15 89 68 3 35 30 5 43 63]
[ 3  5  7 15 18 23 26 28 30 35 42 43 60 63 68 71 76 89 97 97]
```

2.2 バブルソート (Bubble Sort)

バブルソートは隣り合う要素で大小が逆になっているものを交換していく。列の先頭（あるいは末尾）から要素の交換を行っていく場合、列の先頭から順にソートされていくイメージ。平均計算時間・最悪計算時間はともに $O(n^2)$ 。安定ソートアルゴリズム[*]。

- コード引用 Qiita 基本的なソートアルゴリズムとPythonによる実装例

```
In [4]: # バブルソート(Bubble Sort)
def bubble_sort(a):
    # (要素数-1)回繰り返す
    for i in range(len(a) - 1):
        # 列の先頭2要素を最初の比較対象に選ぶ
        l = 0
        r = 1
        # rが末尾に達するまで、隣り合う要素の大小を比較する
        while r < len(a):
            # 隣り合う要素の大小が逆であれば交換する
            if a[l] > a[r]:
                a[l], a[r] = a[r], a[l]
            # 比較対象のindexをインクリメントする
            l += 1
            r += 1
```

2.3 挿入ソート (Insertion Sort)

挿入ソートは手に持ったトランプの並べ替え方に例えられる手法。配列をソート済みの部分と未ソートの部分に分けて考え、未ソート部分の要素をソート部分の然るべき位置に挿入していく。平均計算時間・最悪計算時間はともに $O(n^2)$ であるが、ある程度整列されたデータに対しては高速に動作する。安定ソートアルゴリズム[*]。

- コード引用 Qiita 基本的なソートアルゴリズムとPythonによる実装例

```
In [5]: # 挿入ソート(Insertion Sort)
def insertion_sort(a):
    for i in range(len(a) - 1):
        # a[i]の一つ右の要素: a[j]をソート対象に定める
        j = i + 1

        # a[j]がソート済みの位置に収まるまで繰り返す
        while i >= 0:
            # a[j]を一つ左の要素と比較し、a[j]の方が小さければ交換する
            if a[j] < a[i]:
                a[i], a[j] = a[j], a[i]
            # 比較対象が左に移動しているので、indexを1ずつ減らす
            j -= 1
```

```
i -= 1
j -= 1
else:
    break
```

2.4 クイックソート (Quick Sort)

一般的に最も高速なソートアルゴリズムである。最悪計算量 $O(n^2)$ から最良計算量 $O(n \log n)$ まで幅がある。

配列全体に対して以下の動作を繰り返す。部分配列に対する操作の繰り返しであるので、部分配列長が急速に短縮されるので一般的には高速になる[*]。

- ピボットを選ぶ(中央値がいいが今回は部分配列の最後の要素とする)
- パーティションを用いて部分配列をさらに二つの部分配列にする

1. パーティションの基準点を x として配列の要素を i までの範囲に x 以下の要素を $i + 1$ から j までの範囲に x 以上の要素を移動させる。
2. i までの要素の配列と $i + 1$ から j までの要素の配列の二つに分割する

- コード引用 pythonでクイックソートを実装してみた

```
In [6]: # パーティションの実装
def partition(A, start, end):
    pivot = A[end]
    i = start - 1
    for j in range(start, end):
        if A[j] <= pivot:
            i += 1
            A[i], A[j] = A[j], A[i]
    A[i+1], A[end] = A[end], A[i+1]
    print(A)
    return i+1

# クイックソートの実装
def quicksort(A, start, end):
    if start < end:
        pivot_position = partition(A, start, end)
        quicksort(A, start, pivot_position -1)
        quicksort(A,pivot_position + 1, end)

a = np.random.randint(0, 100, size=16)
print(a)
quicksort(a, 0, len(a)-1)
print(a)
```

```
[74 27 27 18 55 12 74 96 27 38 78 81 86 29 16 27]
[27 27 18 12 27 16 27 96 55 38 78 81 86 29 74 74]
[12 16 18 27 27 27 27 96 55 38 78 81 86 29 74 74]
[12 16 18 27 27 27 27 96 55 38 78 81 86 29 74 74]
[12 16 18 27 27 27 27 96 55 38 78 81 86 29 74 74]
[12 16 18 27 27 27 27 96 55 38 78 81 86 29 74 74]
[12 16 18 27 27 27 27 55 38 29 74 74 86 96 78 81]
[12 16 18 27 27 27 27 55 38 29 74 74 86 96 78 81]
[12 16 18 27 27 27 27 29 38 55 74 74 86 96 78 81]
[12 16 18 27 27 27 27 29 38 55 74 74 78 81 86 96]
[12 16 18 27 27 27 27 29 38 55 74 74 78 81 86 96]
[12 16 18 27 27 27 27 29 38 55 74 74 78 81 86 96]
```

3. データの探索（サーチアルゴリズム）

3.1 配列とデータの探索[3]

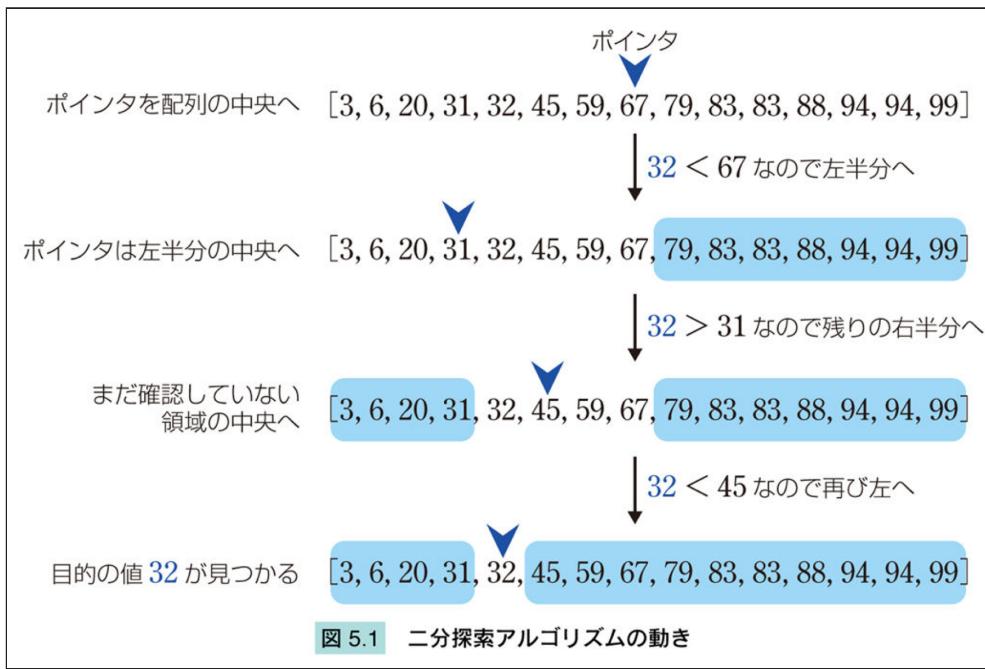
3.1.1 概要

配列におけるデータの探索に関しては、非ソート配列に対する線形探索の計算量は $O(n)$ である。配列がソートされていれば、もっと早く目的の値を見つける方法がある。

二分探索（binary search）アルゴリズムについて説明する。ソート済配列に対して二分探索で目的の値（例えば32）を探す様子を図5.1に示す。

探索は配列の中央から開始する。現在注目している場所をポインタ（青）で示す。ポインタの場所と場所の値（図では67）と目的の値32を比較する。目的の値のほうが小さいので、32があるとすればソートされている配列の左半分にあるということになる。他方、目的の値のほうが大きければ、配列の右半分にあるということになる。

これを繰り返す。最終的に、可能性がある場所が1箇所に絞られるが、これが目的の値なら探索成功なのでTrueを返し、そうでなければFalseを返してアルゴリズムが終了する。ソート済配列に対する二分探索の計算量は $O(\log n)$ である。二分探索結果により、任意の数の挿入位置を求めることができる（後述）。



二分探索アルゴリズムの動き ([3]より引用)

3.1.2 挿入位置の探索

参考：Pythonライブラリ bisect.py

<https://github.com/python/cpython/blob/main/Lib/bisect.py>

```
In [7]: # 亂数を用いて0から100までの非ソート配列(size=15)を生成後、ソートする(seed値=5であれば、図5.1)
import random
random.seed(5)
my_array = [random.randint(0, 100) for i in range(15)]
my_array.sort()
my_array
```

```
Out[7]: [3, 6, 20, 31, 32, 45, 59, 67, 79, 83, 83, 88, 94, 94, 99]
```

```
In [8]: # Pythonの二分探索ライブラリ bisect を使ってデータ挿入位置を求める
# bisectは、ソートされたリストにソートされた状態を保ちながら挿入、挿入する場所を求めることができるラ
import bisect
bisect.bisect(my_array, 40)
```

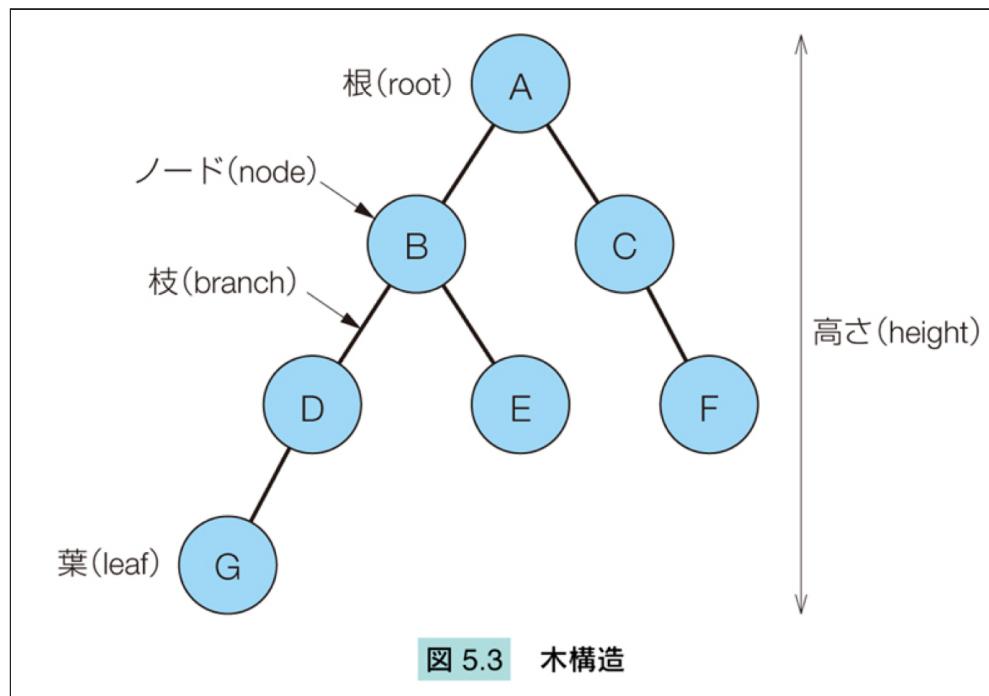
```
Out[8]: 5
```

3.2 探索のためのデータ構造[3]

3.2.1 概要

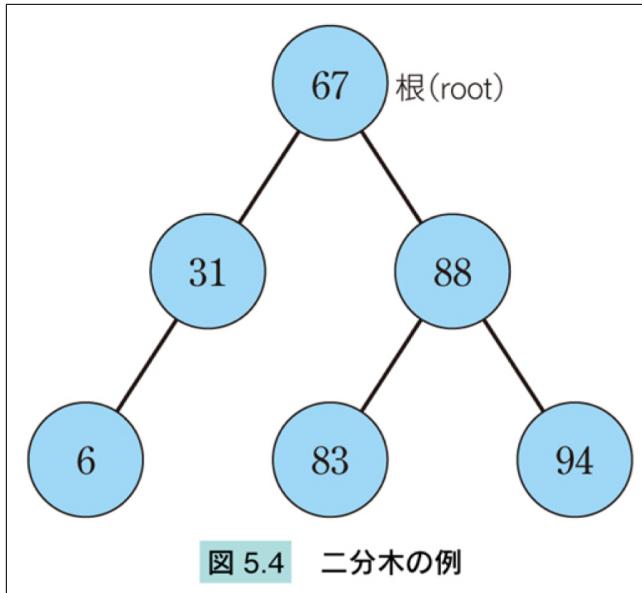
ソートされた配列に対するデータの探索は、二分探索を使えば効率よく実現できる。しかし元のデータが配列のままだと、データの挿入や全体の並び替えに時間がかかる。

データ構造として木構造で保持すると、探索と挿入が高速に行える。各ノードに値を保持した二分木を作る。図5.3に木構造を示す。



木構造 ([3]より引用)

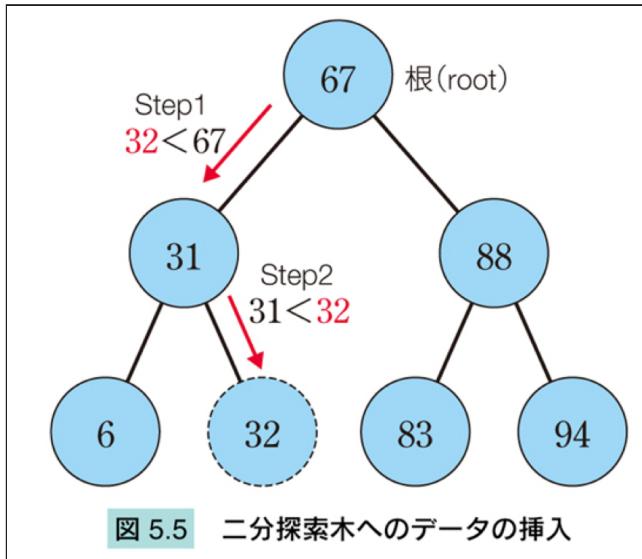
図5.4に二分木の例を示す。ノードに数字を書く。根には67が格納されている。その子ノードとして左側に31、右側に88がある。二分木のノードの親子関係と値の大小に注目すると、各親子関係では必ず左側の子く親く右側の子 という順序関係がある。この順序関係を全てのノードに対して満たしている二分木を、二分探索木 (binary search tree) と言う。



二分探索木の例 ([3]より引用)

図5.4の二分探索木に対して、データの挿入を考える。二分探索で見たように、データ探索の結果として挿入の位置がわかる。いま図5.5のように32の新たなノードを挿入する場合を見ていく。

根から開始する。67との比較で32は左側に位置する。次のノード31より大きいので右側へ位置する。そこには子ノードがないから、ここが32を追加する場所となる。



二分探索木へのデータの挿入例 ([3]より引用)

3.2.2 二分探索木のPythonによる実装

コード引用 : [3] [Pythonで学ぶアルゴリズムとデータ構造](#) コード5.3

実装の方針としては、まずNodeクラスを作る。これは1つのノードを表現している。ノードは値をもち、左側の子ノードと右側の子ノードへの参照を保

持する。二分探索木を表現するクラスBinarySearchTreeは、いくつかのNodeを保持する。順次Nodeを追加して木を成長させられるようとする。簡単にするために、すでに木に含まれる値は追加できないものとする。また、できあがった二分探索木の中から、新たなデータの挿入点を探すメソッドも実装する。これは、自分の親が誰になるかを返すメソッドである。

```
In [9]: class Node:

    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

    def __str__(self):
        # Nodeクラスのインスタンスを文字列表現にする
        left = f'[{self.left.value}]' if self.left else '[]'
        right = f'[{self.right.value}]' if self.right else '[]'
        return f'{left} <- {self.value} -> {right}'


class BinarySearchTree:

    def __init__(self):
        self.nodes = []

    def add_node(self, value):
        node = Node(value)
        if self.nodes:
            # 自分の親ノードを探す
            parent, direction = self.find_parent(value)
            if direction == 'left':
                parent.left = node
            else:
                parent.right = node
        # この木のノードとして格納
        self.nodes.append(node)

    def find_parent(self, value):
        node = self.nodes[0]
        # nodeがNoneになるまでループを回す
        while node:
            p = node # 戻り値の候補(親かもしれない)としてとておく。
            if p.value == value:
                raise ValueError('すでにある値と同じ値を格納することはできません。')
            if p.value > value:
                direction = 'left'
                node = p.left
            else:
                direction = 'right'
                node = p.right
        return p, direction
```

BinarySearchTreeのインスタンスを作って値を追加し木を成長させてみる。例えば次のようなコードを実行してみる[3]。

```
In [10]: btree = BinarySearchTree()
for v in [10, 20, 12, 4, 3, 9, 30]:
    btree.add_node(v)
```

```
# 1つ1つのノードを文字列にする
for node in btree.nodes:
    print(node)
```

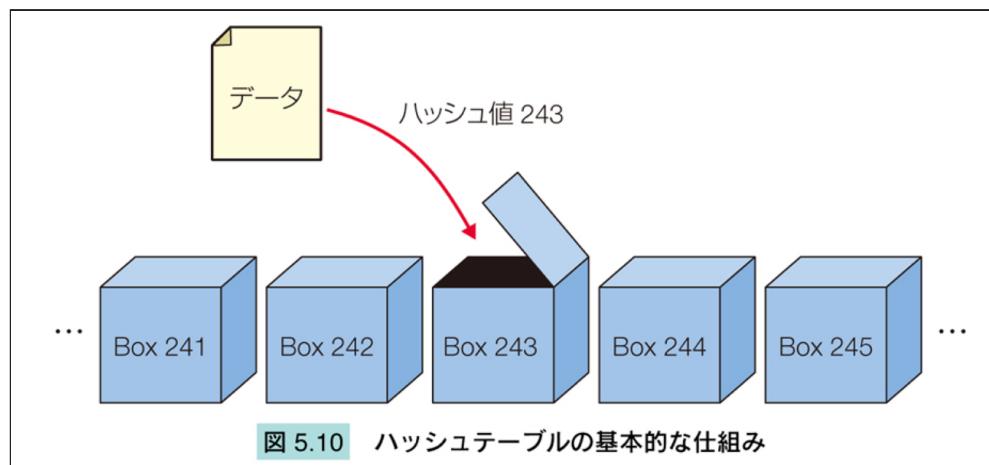
```
[4] <- 10 -> [20]
[12] <- 20 -> [30]
[] <- 12 -> []
[3] <- 4 -> [9]
[] <- 3 -> []
[] <- 9 -> []
[] <- 30 -> []
```

3.3 ハッシュを使った探索[3]

3.3.1 ハッシュ関数とハッシュテーブル

hash関数は引数にとったオブジェクトのハッシュ値（整数）を返す関数である。hash関数の特徴は、オブジェクトが同じであれば同一のハッシュ値を返すが、逆は成立しないことがあげられる。つまり、違うオブジェクトで同一のハッシュ値を返すことがある（ハッシュの衝突）。

データのハッシュ値をデータ探索に利用するのがハッシュテーブルである。オブジェクトが与えられるごとにハッシュ値を計算し、その値に応じた場所にデータを格納する。ハッシュ値が分かれればデータの位置が素早くわかる。このためハッシュテーブル探索にかかる計算量はデータサイズに依らず $O(1)$ の定数時間となる（重要）。



ハッシュテーブルの基本的なしくみ ([3]より引用)

3.3.2 ハッシュテーブルのPythonによる実装（チェーン法）

コード引用： [3] [Pythonで学ぶアルゴリズムとデータ構造](#) コード5.6

チェーン法

実装の方針としては、まず十分な長さのリストを用意しておき、ハッシュ値の剩余をとった値をインデックスとする。作ろうとしているものはキーと値のペアを保存でき、後からキーを指定すると値を取得できるものである。ここで1つのハッシュテーブルの同一キーには複数の値が格納できない。ハッシュ・キーの衝突が起こった場合には、同一キーを持つ値を更にリストを作

って並べることとする。このときのキーもまとめて保存しておき、このリストは線形探索で後から探せるようにしておく。

```
In [11]: # ハッシュテーブルのPythonによる実装(チェーン法)
class HashTable_C:

    def __init__(self, table_size):
        # テーブルのサイズを引数で変更できるようにしてある
        self.data = [[] for i in range(table_size)]
        self.n = table_size

    def get_hash(self, v):
        # オブジェクトのハッシュ値を計算する
        return hash(v) % self.n

    def search(self, key):
        # keyを使って値を探す
        i = self.get_hash(key)
        for j, v in enumerate(self.data[i]):
            if v[0] == key:
                return (i, j)
        return (i, -1)

    def set(self, key, value):
        # データを格納するべき場所を探す
        i, j = self.search(key)
        if j != -1:
            # すでにある値を書き換える
            self.data[i][j][1] = value
        else:
            # 新たなデータとして付け加える
            self.data[i].append([key, value])

    def get(self, key):
        i, j = self.search(key)
        if j != -1:
            return self.data[i][j][1]
        # キーが見付からない場合はエラーを返す
        raise KeyError(f'{key} was not found in this HashTable!')
```

```
In [12]: # ハッシュテーブルのテスト
my_hash_table = HashTable_C(100)
my_hash_table.set('taro', 10) # キー taro 値 10 のペアを格納
my_hash_table.get('taro') # キー taro の格納値を検索(存在しないキーが指定されたらエラー)
```

Out [12]: 10

3.3.3 ハッシュテーブルのPythonによる実装（オープンアドレス法）

コード引用：chatGPT - プロンプト [Python ハッシュ 探索 オープンアドレス法 サンプル コード]

参考：<https://zenn.dev/fikastudio/articles/efcfe246642553>

オープンアドレス法は、同じバケットに衝突したときに再ハッシュを行うことで、空いているバケットを探し出す手法である。クローズドハッシュ法とも呼ばれる。再ハッシュ関数は自由に設定することができる。

オープンアドレス法での、データの検索と削除は少し面倒である。単に、挿入データのハッシュ値を検索するだけでなく、データの再ハッシュについても検索する必要があるからである。

```
In [13]: # ハッシュテーブルのPythonによる実装(オープンアドレス法)
class HashTable_OA:
    def __init__(self, size):
        self.size = size
        self.keys = [None] * self.size
        self.values = [None] * self.size

    def hash_function(self, key):
        # キーのハッシュ値を計算する関数
        return hash(key) % self.size

    def rehash(self, old_hash):
        # 再ハッシュ関数
        return (old_hash + 1) % self.size

    def put(self, key, value):
        hash_value = self.hash_function(key)

        # ハッシュ値が未使用または削除済みの場合、キーと値を格納する
        if self.keys[hash_value] is None or self.keys[hash_value] == "delete":
            self.keys[hash_value] = key
            self.values[hash_value] = value
        else:
            # 再ハッシュして空きスロットを探す
            next_slot = self.rehash(hash_value)
            while self.keys[next_slot] is not None and self.keys[next_slot] != "delete":
                next_slot = self.rehash(next_slot)

            # キーと値を格納する
            if self.keys[next_slot] is None or self.keys[next_slot] == "delete":
                self.keys[next_slot] = key
                self.values[next_slot] = value
            else:
                # キーが既に存在する場合は値を更新する
                self.values[next_slot] = value

    def get(self, key):
        hash_value = self.hash_function(key)
        slot = hash_value

        while self.keys[slot] is not None:
            if self.keys[slot] == key:
                return self.values[slot]
            else:
                slot = self.rehash(slot)
                if slot == hash_value:
                    break

        # キーが見つからない場合はNoneを返す
        return None

    def remove(self, key):
        hash_value = self.hash_function(key)
        slot = hash_value

        while self.keys[slot] is not None:
```

```
    if self.keys[slot] == key:
        # キーと値を削除する
        self.keys[slot] = "deleted"
        self.values[slot] = None
        return
    else:
        slot = self.rehash(slot)
        if slot == hash_value:
            break
    # キーが見つからない場合は何もしない
```

```
In [14]: # ハッシュテーブルのテスト
hash_table = HashTable_0A(100)

hash_table.put("apple", 5)
hash_table.put("banana", 10)
hash_table.put("orange", 15)

print(hash_table.get("apple")) # 5
print(hash_table.get("banana")) # 10
print(hash_table.get("orange")) # 15
print(hash_table.get("grape")) # None

hash_table.remove("banana")
print(hash_table.get("banana")) # None
```

5
10
15
None
None

memo