

## 1 線形探索

検索 (サーチ) とは、データ集合 (配列など) から、目的とする値を持った要素を探し出すことを意味する。

### I. 線形探索

線形探索は、目的とする要素が見つかるまで先頭から順に要素を見ていく探索方法です。例えば、以下の配列で 0 を探すなら前から順番に、8, 4, 5, 0, 2 の順に要素を見ていきます。

8	4	5	0	2
---	---	---	---	---

#### i. 線形探索の実装

コード 1 線形探索の実装

```
1 def linear_search(array: list[int], value: int) -> int:
2     """
3     線形探索をして一致するならその index を返す. 一致しないときは -1 を返す
4     """
5     for i in range(len(array)):
6         if array[i] == value:
7             return i
8
9     return -1
```

#### ii. 番兵

番兵法では探索するデータ集合の最後に目的とする数を追加します。最後に番兵を追加することで、より効率的に探索を行うことができます。実際には番兵を追加してもそこまで効率が良くなるわけではないですが、番兵を追加することで、ループの条件判定を省略することができます。

8	4	5	3	2	0
---	---	---	---	---	---

番兵

コード 2 番兵の実装

```
1 def linear_search(array: list[int], value: int) -> int:
2     """
3     線形探索をして一致するならその index を返す. 一致しないときは -1 を返す
4     """
```

## 2 二分探索

```
5     i = 0
6     copied_array = array.copy()
7     copied_array.append(value)
8     while copied_array[i] != value: i += 1
9
10    return i if i < len(array) else -1
```

## II. 二分探索

### i. 配列を探索する二分探索

二分探索はソートされた配列に対して高速に探索を行うアルゴリズムの一つです。二分探索では探索する区間が条件に応じてどんどん半分になっていくため、計算量は  $O(\log n)$  となります。以下の配列に対して、二分探索で 18 を探す場合を考えましょう。

最初の区間は配列全体を取ります。0-indexed な配列を考えると、mid は 39 となり目的の 18 より大きいです。mid が目的の値より大きい場合、右の区間を狭めます。right = mid - 1 とします。

5	18	22	28	39	48	51	68	82	94
0	1	2	3	4	5	6	7	8	9
↑				↑					↑
left				mid					right

right を 3 に更新しました。mid = (0 + 3) // 2 = 1 となります。mid の値は 18 で目的の値と一致します。

5	18	22	28	39	48	51	68	82	94
0	1	2	3	4	5	6	7	8	9
↑	↑		↑						
left	mid		right						

次に、目的とする値を 19 として配列に存在しない場合を考えましょう。mid = 1 の値は 18 で目的の値よりも小さいので、下の図のように left を mid + 1 に更新します。mid = (2 + 3) // 2 = 2 となります。mid の値は 22 で目的の値より大きいです。よって、right を mid - 1 に更新します。すると、right = 1, left = 2 となり、left > right となるので探索を終了します。

5	18	22	28	39	48	51	68	82	94
0	1	2	3	4	5	6	7	8	9
		↑	↑						
		left	right						

二分探索の実装は以下の通りです。

コード 3 二分探索の実装

```

1 def binary_search(A: list[int], value: int) -> int:
2     left, right = 0, len(A) - 1
3
4     while left <= right:
5         mid = (left + right) // 2
6
7         if A[mid] == value:
8             return mid
9
10        if A[mid] < value:
11            left = mid + 1
12        else:
13            right = mid - 1
14
15    return -1

```

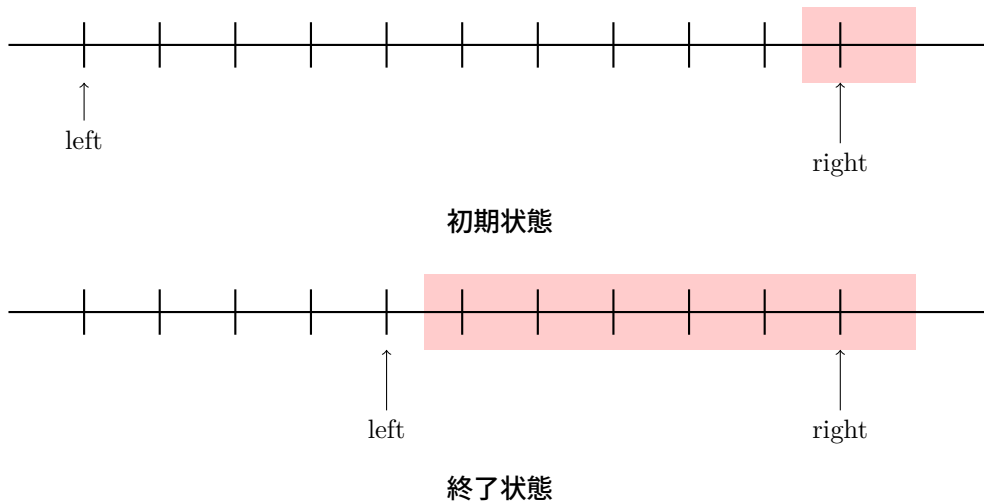
## ii. 一般化した二分探索

Python の標準ライブラリである `bisect` の `bisect_left` の実装をしましょう。`bisect_left` はソートされた配列に対して、与えられた値以上の最小の index を返す関数です。`bisect_left` は二分探索を用いて実装されています。先ほどの配列から要素を探す二分探索では、要素一致するか否かを判定していましたが、今回はもっと一般化して「mid がある条件を満たすか否か」を判定して範囲を狭めていきます。このとき、left より右側の区間は条件を満たさず、right より左側の区間は条件を満たすようにします。

二分探索では以下の図のような条件を満たす赤い部分を条件を満たす境界ギリギリになるように更新していきます。

- left は常に条件を満たさない
- right は常に条件を満たす

という条件を満たすように処理を進めていき、最終的に `left` が条件を満たさない最大の index、`right` が条件を満たす最小の index を返します。`bisect_left` では `left` が与えられた値よりも小さく、`right` が与えられた値以上の最小の index を返す関数です。



`bisect_left` の実装は以下の通りです。

コード 4 `bisect_left` の実装

```

1 def bisect_left(array: list[int], key: int) -> int:
2     left, right = -1, len(array)
3
4     while right - left > 1:
5         mid = (left + right) // 2
6
7         if array[mid] < key:
8             left = mid
9         else:
10            right = mid
11
12    return right

```

`bisect_left` の実装では、左側が条件を満たさない、右側が条件を満たすといった実装になっています。これだとまだ条件次第で `left` が条件を満たす、`right` が条件を満たさないという実装もあり得ます。そこで以下ではめぐる式二分探索のさらに一般化した実装を紹介します。

### iii. めぐる式二分探索

めぐる式二分探索では、

- ng は常に条件を満たさない
- ok は常に条件を満たす

のように数直線上の右や左という概念を持たせずに実装します。めぐる式二分探索は以下のように実装されます。is\_ok は条件を満たすかどうかを判定する関数ですので、条件に合わせて実装します。

コード 5 めぐる式二分探索

```
1 def binary_search(array: list[int], key: int):
2     ng, ok = -1, len(array)
3
4     while abs(ok - ng) > 1:
5         mid = (ng + ok) // 2
6
7         if is_ok(array, mid, key):
8             ok = mid
9         else:
10            ng = mid
11
12    return ok
```

#### iv. 問題

**問題 1** AtCoder 典型アルゴリズム問題集 A - 二分探索の練習問題 bisect.bisect\_left の実装問題です。自分で実装して AC しましょう。また、bisect.bisect\_left の使い方も同時に理解しましょう。

解答は以下の通りです。

コード 6 問題 1

```
1 def is_ok(A: list[int], key: int, index: int) -> bool:
2     return A[index] >= key
3
4 def my_bisect_left(A: list[int], key) -> int:
5     ok, ng = len(A), -1
6     while abs(ng - ok) > 1:
7         mid = (ng + ok) // 2
8
9         if is_ok(A, key, mid):
```

```
10         ok = mid
11     else:
12         ng = mid
13
14     return ok
15
16 def main():
17     n, k = map(int, input().split())
18     A = list(map(int, input().split()))
19
20     index = my_bisect_left(A, k)
21
22     print(index if index != n else -1)
23
24 if __name__ == "__main__":
25     main()
```

**問題 2** AtCoder Beginner Contest ARC 050 B 花束

参考文献 2. にある通り、「答えを決めつタイプ」の二分探索です。詳しくは参考文献に譲りますが、二分探索の応用性の高さを感じる問題です。

**v. 参考**

1. <https://qiita.com/drken/items/97e37dd6143e33a64c8c>
2. <https://bettrue12.hateblo.jp/entry/2019/05/11/013403>

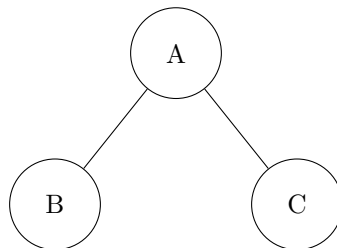
### III. 二分探索木

二分探索は探索自体は  $O(\log n)$  で行うことができますが、配列がソートされているひつようがあり、データの挿入や削除がある場合は毎回ソートがあり非効率ではあります。解決策としては、**データ構造で解決**や **Treepq** などがあります。データ構造で解決する方法の一つが**二分探索木**です。

#### i. 二分探索木の概要と実装

二分探索木は以下の性質を持った木構造です。

- 左子ノードは親ノードよりも小さい (または等しい) (下の図では  $B \leq A$ )
- 右子ノードは親ノードよりも大きい (下の図では  $A < C$ )



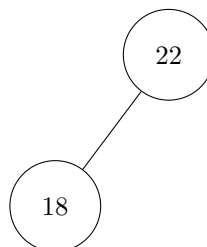
二分探索木で行う処理は以下の通りです。

- 木の作成
- 探索
- 削除

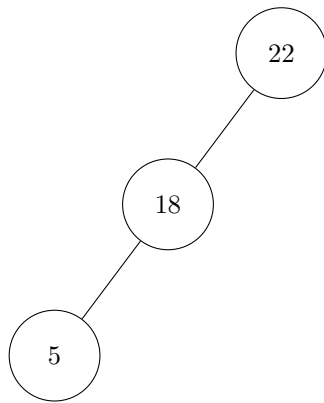
二分探索木を作成する手順を見てみましょう。[22, 18, 5, 82, 51, 39] の配列を二分探索木に変換してみましょう。



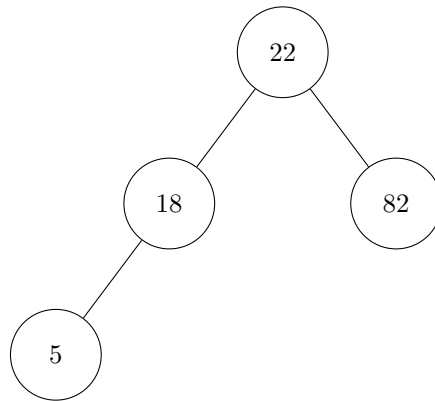
22 を挿入



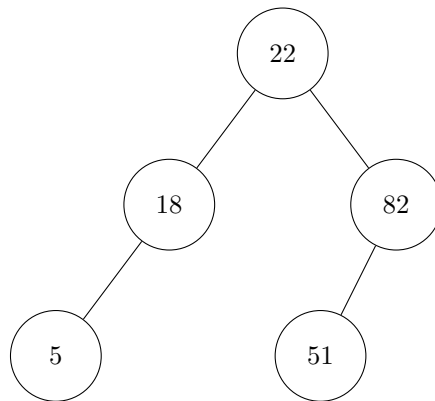
18 を挿入



5 を挿入

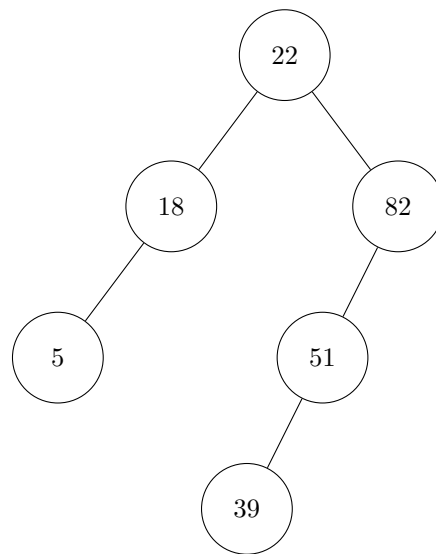


82 を挿入

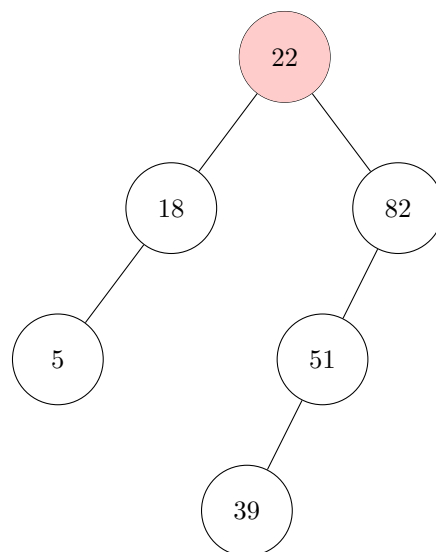


51 を挿入

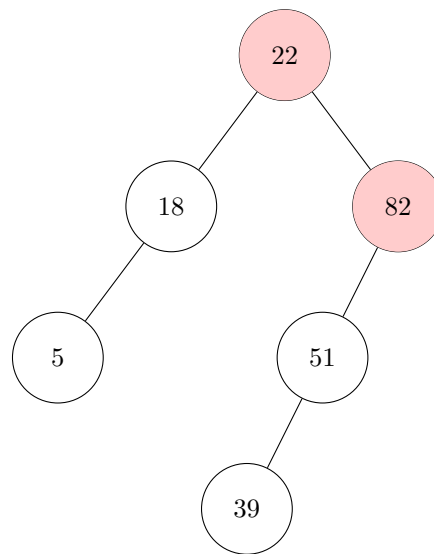


**39 を挿入**

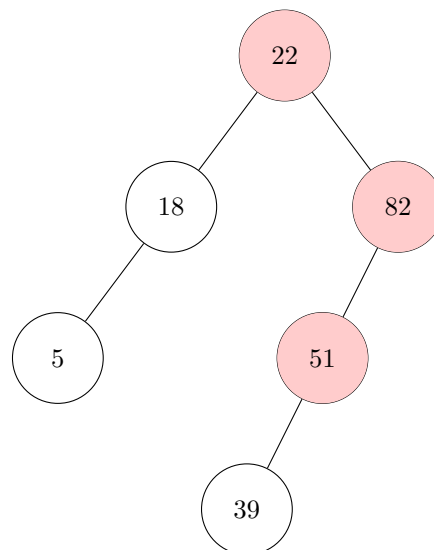
二分探索木での探索を考えます。上の木を例に 51 を探してみましょう。根から順に探していきます。



51 は 22 よりも大きいので右の子ノードに進みます。



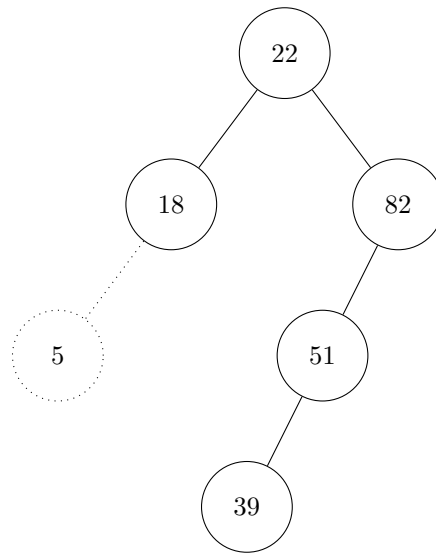
51 は 82 よりも小さいので左の子ノードに進みます。見つかりました。



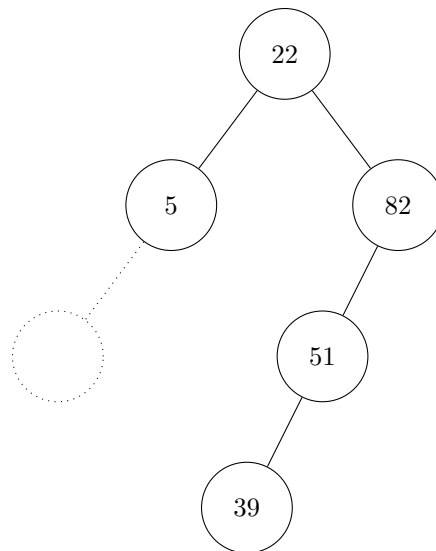
二分探索木からノードを削除する場合は以下の 3 つのケースがあります。

- 子ノードがない場合
- 子ノードが 1 つの場合
- 子ノードが 2 つの場合

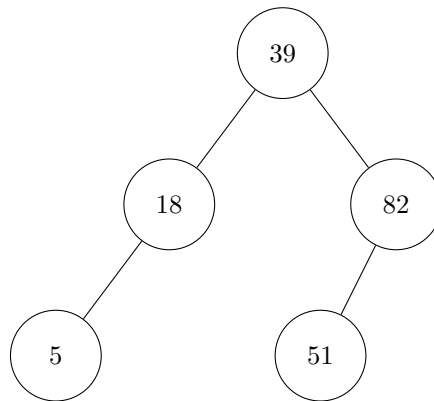
子ノードがない場合をばあいを扱います。例えば 5 を削除する場合を考えます。5 は左右の子ノードがないので、他のノードに影響がないためそのまま削除します。



子ノードが 1 つの場合を扱います。次に 18 を削除する場合を考えます。18 は左の子ノードが 5 しか持っていないので、5 を 18 の位置に移動させます。18 は親ノードから見て左の子ノードですが、右ノードの場合も同様に処理します。



子ノードが 2 つの場合を扱います。最後に 22 を削除する場合を考えます。削除するノードが 2 つの子ノードを持っている場合は、削除するノードの左の子ノードの最大値か、右の子ノードの最小値を持ってきて、削除するノードに移動させます。ここでは、22 の右の子ノードの最小値 39 を持ってきて、22 の位置に移動させます。



最後に二分探索木の実装例を紹介します。

コード 7 二分木の実装

```
1 class Node:
2     def __init__(self, data: int) -> None:
3         self.data: int = data
4         self.right: Node | None = None
5         self.left: Node | None = None
6
7     def __str__(self) -> str:
8         return str(self.data)
9
10 class BinarySearchTree:
11     def __init__(self):
12         self.root = None
13
14     def create(self, array: list[int]) -> None:
15         for i in range(len(array)):
16             self.insert(array[i])
17
18     def insert(self, data: int) -> None:
19         if self.root is None:
20             self.root = Node(data)
21         else:
22             self._insert_recursively(self.root, data)
23
24     def _insert_recursively(self, node: Node, data: int) -> None:
25         if data <= node.data:
```

```
26         if node.left is None:
27             node.left = Node(data)
28         else:
29             self._insert_recursively(node.left, data)
30     else:
31         if node.right is None:
32             node.right = Node(data)
33         else:
34             self._insert_recursively(node.right, data)
35
36     def search(self, value: int) -> Node | None:
37         if self.root is None:
38             return self.root
39
40         return self._search_recursively(self.root, value)
41
42     def _search_recursively(self, node: Node, value: int) -> Node | None:
43         if node.data == value:
44             return node
45
46         if value < node.data:
47             if node.left is None:
48                 return node.left
49             else:
50                 return self._search_recursively(node.left, value)
51         else:
52             if node.right is None:
53                 return node.right
54             else:
55                 return self._search_recursively(node.right, value)
56
57     def delete(self, value: int) -> Node | None:
58         if self.root is None:
59             return self.root
60         else:
61             self.root = self._delete_recursively(self.root, value)
```

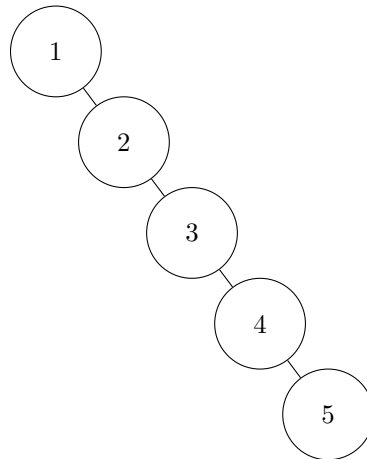
```
62
63 def _delete_recursively(self, node: Node, value: int) -> Node | None:
64     if value < node.data:
65         if node.left is None:
66             return None
67         else:
68             node.left = self._delete_recursively(node.left, value)
69     elif value > node.data:
70         if node.right is None:
71             return None
72         else:
73             node.right = self._delete_recursively(node.right, value)
74     else:
75         if node.left is None and node.right is None:
76             return None
77         elif node.left is None:
78             return node.right
79         elif node.right is None:
80             return node.left
81         else:
82             successor = self._successor()
83             node.data = successor.data
84             self._delete_recursively(node.right, successor.data)
85
86     return node
87
88 def _successor(self) -> Node | None:
89     if self.root is None:
90         return self.root
91
92     node = self.root
93
94     while node.left is not None:
95         node = node.left
96
97     return node
```

## IV. 平衡木

---

### i. 二分木の問題点

二分木は最悪の場合木が以下のように片方に伸びてしまった場合、計算量が  $O(n)$  になってしまいます。



### ii. 平衡木

平衡木は**回転**や乱択アルゴリズムを用いて木の高さを自動的に小さくする方向に調整できる木構造です。

扱う平衡木の代表的なものには以下のものがあります。

- AVL 木
- 赤黒木
- B 木
- Treap

### iii. AVL 木

AVL 木とは、**Adelson-Velsky and Landis** の名前から取られた木構造で、以下の性質を持ちます。

- 任意のノードにおいて、左右の部分木の高さの差が 1 以下である

以降の考察のために **balance factor** という言葉を導入します。

**balance factor**

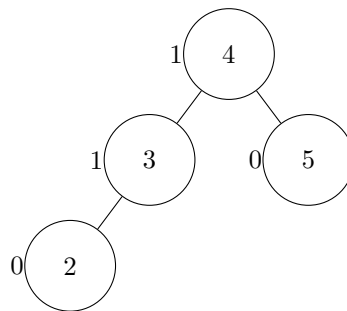
あるノードの左部分木の高さから右部分木の高さを引いた値を **balance factor** といいます。

- $\text{balance factor} = \text{左部分木の高さ} - \text{右部分木の高さ}$

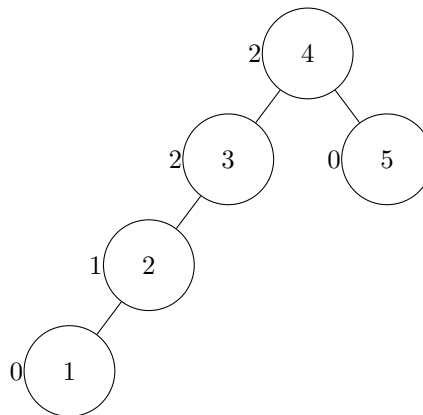
AVL 木の性質を balance factor を使って表すと以下のようになります。

- 任意のノードにおいて、balance factor は-1, 0, 1 のいずれかである

AVL 木の例を以下に示します。ノードの横に書いてあるのが balance factor です。

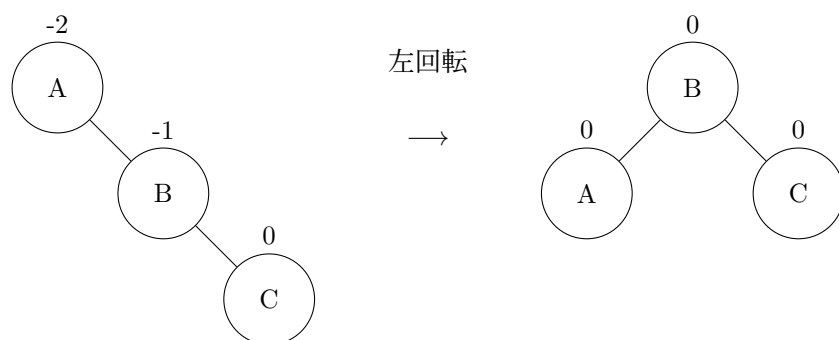


次は AVL 木の条件を満たしていない木です。

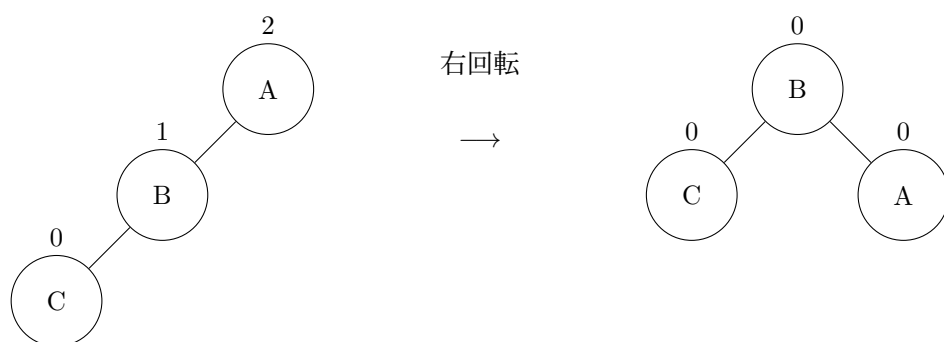




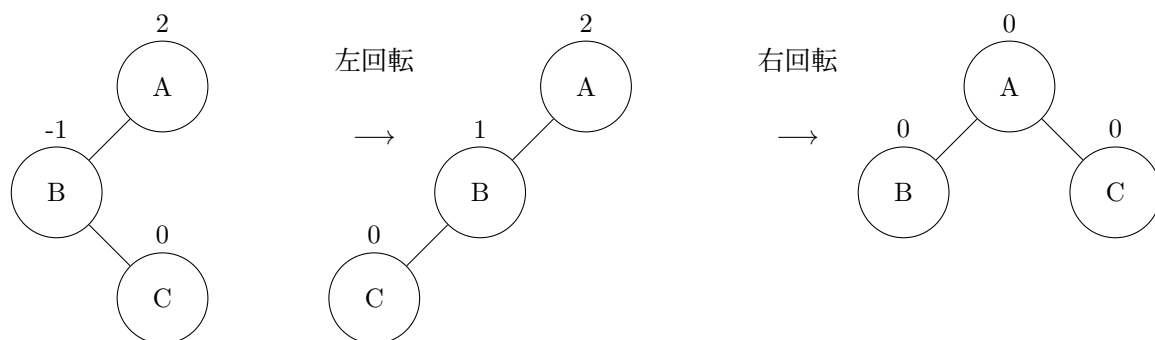
## 1. 左回転



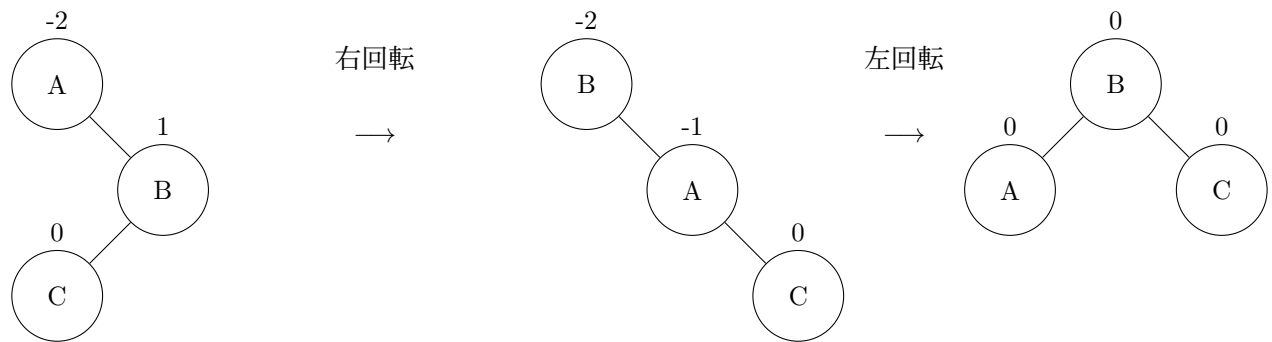
## 2. 右回転



## 3. 左右回転



## 4. 右左回転



コード 8 二分ヒープの実装

```

1  class Node:
2      def \_\_init\_\_(self, value: int) -> None:
3          self.data: int = value
4          self.left: Node | None = None
5          self.right: Node | None = None
6          self.height: int = 0
7
8      def \_\_str\_\_(self) -> str:
9          return f"Node: data = {self.data} height = {self.height}"
10
11 class AVLTree:
12     def \_\_init\_\_(self):
13         self.root = None
14
15     def \_height(self, node: Node | None) -> int:
16         if node is None:
17             return -1
18
19         return node.height
20
21     def \_update\_height(self, node:
22         Node) -> None:
23         node.height = 1 + max(self.\_height(node.left), self.\_height(
24             node.right))
25
26     def \_balance\_factor(self, node: Node) -> int:
27         return self.\_height(node.left) - self.\_height(node.right)

```

```
27
28     def \_right\_rotate(self, node: Node) -> Node:
29         new\_root = node.left
30         node.left = new\_root.right
31         new\_root.right = node
32
33         self.\_update\_height(node)
34         self.\_update\_height(new\_root)
35
36         return new\_root
37
38     def \_left\_rotate(self, node: Node) -> Node:
39         new\_root = node.right
40         node.right = new\_root.left
41         new\_root.left = node
42
43         self.\_update\_height(node)
44         self.\_update\_height(new\_root)
45
46         return new\_root
47
48     def create(self, values: list[int]) -> None:
49         for value in values:
50             self.insert(value)
51
52     def insert(self, value: int) -> None:
53         if self.root is None:
54             self.root = Node(value)
55             return self.root
56         else:
57             self.root = self.\_insert\_recursively(self.root, value)
58             return self.root
59
60     def \_insert\_recursively(self, node: Node, value: int) -> Node:
61         if value <= node.data:
62             if node.left is None:
```

```
63         node.left = Node(value)
64     else:
65         node.left = self._insert_recursively(node.left, value
66         )
67     else:
68         if node.right is None:
69             node.right = Node(value)
70         else:
71             node.right = self._insert_recursively(node.right,
72             value)
73
74     self._update_height(node)
75
76     balance_factor = self._balance_factor(node)
77
78     if balance_factor > 1 and value < node.left.data:
79         return self._right_rotate(node)
80
81     if balance_factor > 1 and value > node.left.data:
82         node.left = self._left_rotate(node.left)
83         return self._right_rotate(node)
84
85     if balance_factor < -1 and value > node.right.data:
86         return self._left_rotate(node)
87
88     if balance_factor < -1 and value < node.right.data:
89         node.right = self._right_rotate(node.right)
90         return self._left_rotate(node)
91
92     return node
93
94     def search(self, value: int) -> Node | None:
95         if self.root is None:
96             return None
```

```
97         return self.\_search\_recursively(self.root, value)
98
99     def \_search\_recursively(self, node: Node, value: int) -> Node |
None:
100         if value == node.data:
101             return node
102
103         if value < node.data:
104             if node.left is None:
105                 return None
106             else:
107                 return self.\_search\_recursively(node.left, value)
108         else:
109             if node.right is None:
110                 return None
111             else:
112                 return self.\_search\_recursively(node.right, value)
113
114
115     def delete(self, value: int) -> Node | None:
116         if self.root is None:
117             return self.root
118         else:
119             self.root = self.\_delete\_recursively(self.root, value)
120             return self.root
121
122     def \_delete\_recursively(self, node: Node, value: int) -> Node |
None:
123         if value < node.data:
124             node.left = self.\_delete\_recursively(node.left, value)
125         elif value > node.data:
126             node.right = self.\_delete\_recursively(node.right, value)
127         else:
128             if node.left is None and node.right is None:
129                 return None
130             elif node.left is None:
```

```
131         return node.right
132     elif node.right is None:
133         return node.left
134     else:
135         min\_node = self.\_find\_min(node.right)
136         node.data = min\_node.data
137         node.right = self.\_delete\_recursively(node.right, min\_node.data)
138
139     self.\_update\_height(node)
140
141     balance\_factor = self.\_balance\_factor(node)
142
143     if balance\_factor > 1 and self.\_balance\_factor(node.left) >=
144         0:
145         return self.\_right\_rotate(node)
146
147     if balance\_factor < -1 and self.\_balance\_factor(node.right)
148         <= 0:
149         return self.\_left\_rotate(node)
150
151     if balance\_factor > 1 and self.\_balance\_factor(node.left) <
152         0:
153         node.left = self.\_left\_rotate(node.left)
154         return self.\_right\_rotate(node)
155
156     if balance\_factor < -1 and self.\_balance\_factor(node.right)
157         > 0:
158         node.right = self.\_right\_rotate(node.right)
159         return self.\_left\_rotate(node)
160
161     return node
```

## iv. B 木

AVL 木は二分木であるため、データの数が多い場合に気が高くなり検索や挿入に時間がかかってしまう問題があります。そこで枝の数が 2 本よりも多く取る B 木というデータ構造を紹介します。B 木は応用範囲が広く以下の様な様々なものに使われています。

- データベース
- ファイルシステム

B 木はノードが持てる最大の枝の本数 *order* によって定義されます。二分木ではノードは 1 つの値を持っていてその値との大小比較で左右の子ノードに振り分けていましたが、B 木ではノードが複数の値を持ち、その値の範囲で子ノードに振り分けます。order=3 の B 木の例を以下に示します。[10, 5, 3, 2, 8, 4, 1, 6] のデータを order=3 の B 木に入れていきます。



コード 9 B 木の実装

```

1 from bisect import bisect_left, bisect_right
2
3 class Node:
4     def __init__(self):
5         self.keys: list[int] = []
6         self.children: list[Node] = []
7
8 class BTree:
9     def __init__(self, order: int) -> None:
10         self.order = order
11         self.root = Node()
12
13     def _median(self, array: list[int]) -> int:
14         return array[len(array) // 2]
15
16     def insert(self, value: int) -> None:
17         node = self._insert_recursively(self.root, value)
18         if node:
19             new_root = Node()
20             new_root.keys = [node[0]]
21             new_root.children = [node[1], node[2]]

```

```
22         self.root = new_root
23
24     def _insert_recursively(self, node: Node, value: int):
25         # 葉ノード
26         if not node.children:
27             node.keys.append(value)
28             node.keys.sort()
29             if len(node.keys) < self.order:
30                 return None
31             else:
32                 return self._split(node)
33         else:
34             index = bisect_right(node.keys, value)
35             result = self._insert_recursively(node.children[index], value
36                                             )
37
38             if result is not None:
39                 median, left_node, right_node = result
40                 node.keys.insert(index, median)
41
42                 # 修正：左ノードを適切に設定し、右ノードを挿入
43                 node.children[index] = left_node # 左の子ノードを置き換
44                 # える
45                 node.children.insert(index + 1, right_node) # 右の子ノー
46                 # ドを挿入
47
48             if len(node.keys) < self.order:
49                 return None
50             else:
51                 return self._split(node)
52         return None
53
54     def _split(self, node: Node):
55         median_index = len(node.keys) // 2
56         median = node.keys[median_index]
```



```
55     left_node = Node()
56     right_node = Node()
57
58     left_node.keys = node.keys[:median_index]
59     right_node.keys = node.keys[median_index + 1:]
60
61     if node.children:
62         left_node.children = node.children[:median_index + 1]
63         right_node.children = node.children[median_index + 1:]
64
65     return median, left_node, right_node
66
67     def search(self, value: int) -> bool:
68         node = self.root
69         if value in node.keys:
70             return True
71         else:
72             return self._search_recursively(self.root, value)
73
74     def _search_recursively(self, node: Node, value: int) -> bool:
75         if len(node.children) == 0:
76             return False
77
78         index = bisect_left(node.keys, value)
79         next_node = node.children[index]
80
81         if value in next_node.keys:
82             return True
83         else:
84             return self._search_recursively(node.children[index], value)
85
86 b_tree = BTree(order=3)
87 b_tree.insert(5)
88 b_tree.insert(10)
89 b_tree.insert(15)
90 b_tree.insert(20)
```

```
91 b_tree.insert(30)
92
93 b_tree.insert(21)
94 b_tree.insert(22)
```

## v. 赤黒木

赤黒木は AVL 木同様平衡二分探索木の 1 つで、以下の特徴を持っています。

- 各ノードは赤か黒の色を持つ
- 根ノードは黒である
- 赤のノードの子ノードは黒である
- あるノードからその子孫の葉ノードまでの黒の数は同じである
- 葉ノードは黒である

## V. 参考

---

### 二分探索

- <https://qiita.com/drken/items/97e37dd6143e33a64c8c>

### 二分探索木

- <https://www.geeksforgeeks.org/introduction-to-avl-tree/>

### B 木

- <https://www.geeksforgeeks.org/b-tree-in-python/>
- <https://www.javatpoint.com/b-tree>
- <https://wqwq3215.medium.com/b-tree%E3%82%92%E7%90%86%E8%A7%A3%E3%81%97%E3%81%A6%E3%81%84%E3%81%8F-142f93fc3c6c>

### 赤黒木

- <https://qiita.com/kgoto/items/b15b9a494deae010d660>
- <http://wwa.pikara.ne.jp/okojisan/rb-tree/index.html>