探索の章で扱うアルゴリズムは以下と通りです。

- 線形探索
- 二分探索
- 二分探索木
- Treeq
- 平衡木 (AVL 木、B 木、赤黒木)
- ハッシュ法

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

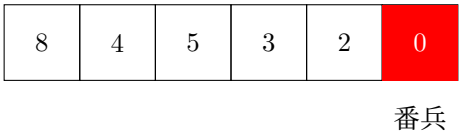
i.　線形探索の実装

コード 1　線形探索の実装

```python
def linear_search(array: list[int], value: int) -> int:
    """
    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
    """
    for i in range(len(array)):
        if array[i] == value:
            return i

    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。



コード 2　番兵の実装

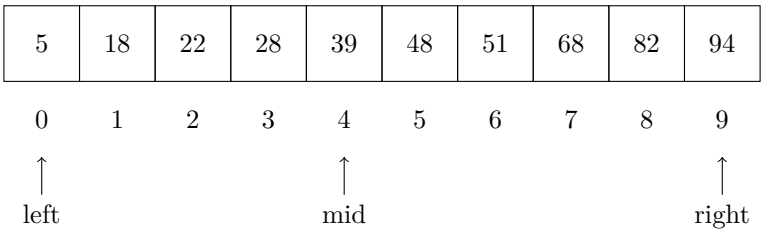
```python
def linear_search(array: list[int], value: int) -> int:
    """
    線形探索をして一致するならそのindexを返す．一致しないときは-1を返す
    """
    i = 0
    copied_array = array.copy()
    copied_array.append(value)
    while copied_array[i] != value: i += 1

    return i if i < len(array) else -1
```

II.　二分探索

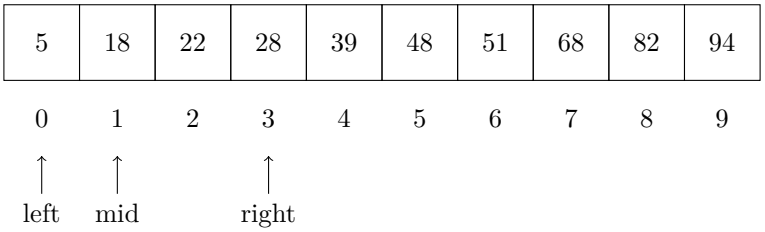
i.　配列を探索する二分探索

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。

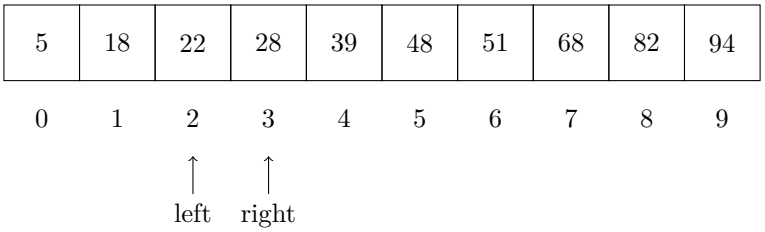
最初の区間は配列全体を取ります。0-indexed な配列を考えると、mid は 39 となり目的の 18 より大きいです。mid が目的の値より大きい場合、右の区間を狭めます。right = mid - 1 とします。



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。



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。



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

```python
def binary_search(A: list[int], value: int) -> int:
    left, right = 0, len(A) - 1

    while left <= right:
        mid = (left + right) // 2

        if A[mid] == value:
            return mid

        if A[mid] < value:
            left = mid + 1
        else:
            right = mid - 1

    return -1
```

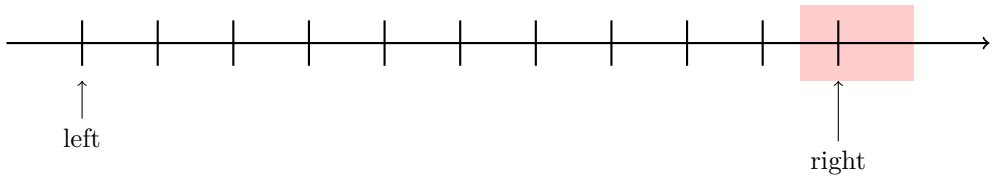
ii.　一般化した二分探索

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。

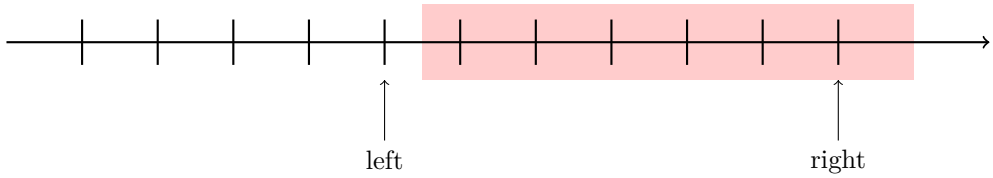
二分探索では以下の図のような条件を満たす赤い部分を条件を満たす境界ギリギリになるように更新していきます。

- 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。



初期状態



終了状態

bisect_left の実装は以下の通りです。

コード 4　bisect_left の実装

```
def bisect_left(array: list[int], key: int) -> int:
    left, right = -1, len(array)

    while right - left > 1:
        mid = (left + right) // 2

        if array[mid] < key:
```

```
            left = mid
        else:
            right = mid

    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

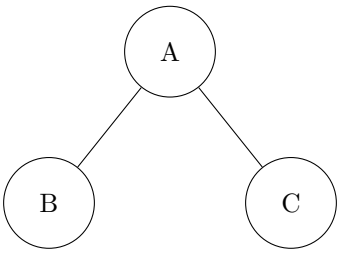
```
def binary_search(array: list[int], key: int):
    ng, ok = -1, len(array)

    while abs(ok - ng) > 1:
        mid = (ng + ok) // 2

        if is_ok(array, mid, key):
            ok = mid
        else:
            ng = mid

    return ok
```

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 **Treeq** 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。

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 $A < C$)



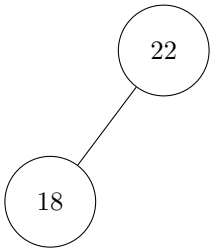
二分探索木で行う処理は以下の通りです。

- 木の作成
- 探索
- 削除

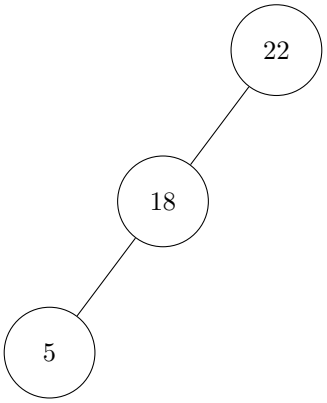
i.　二分探索木の作成

`[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。

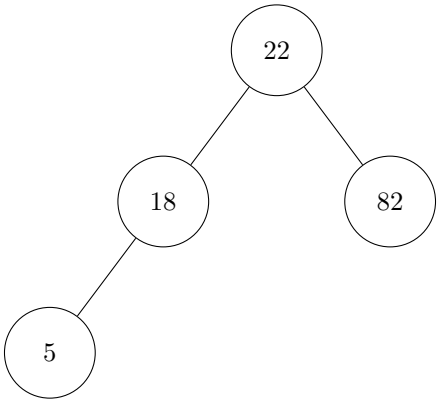
22 を挿入



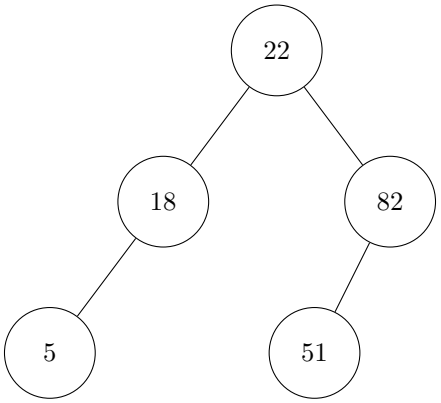
18 を挿入



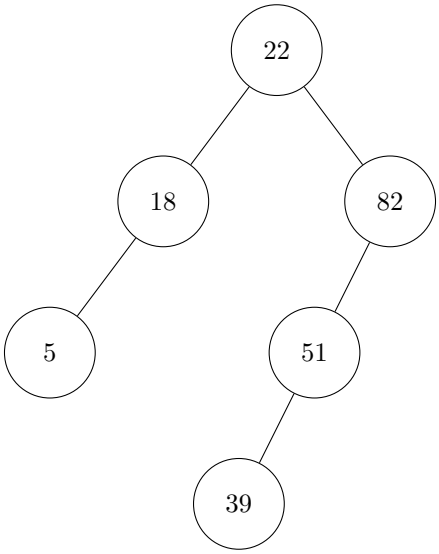
5 を挿入



82 を挿入



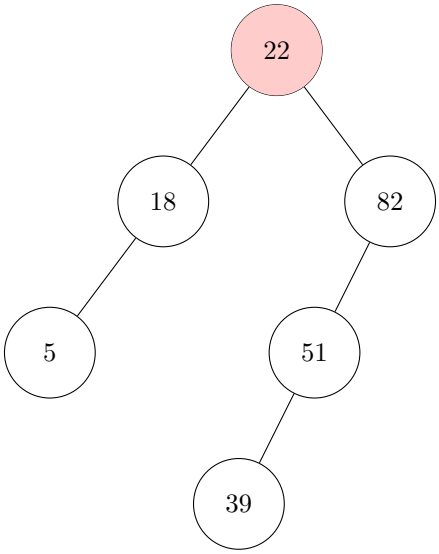
51 を挿入



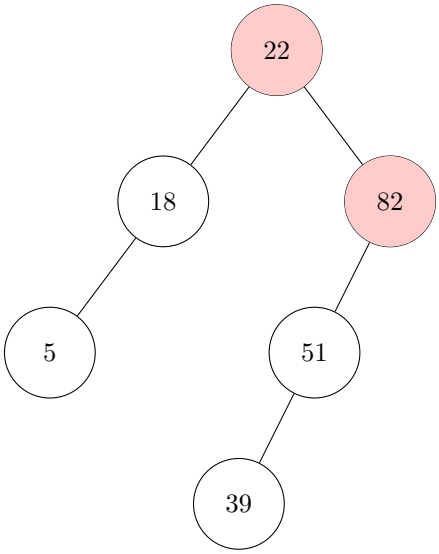
39 を挿入

ii.　二分探索木の探索

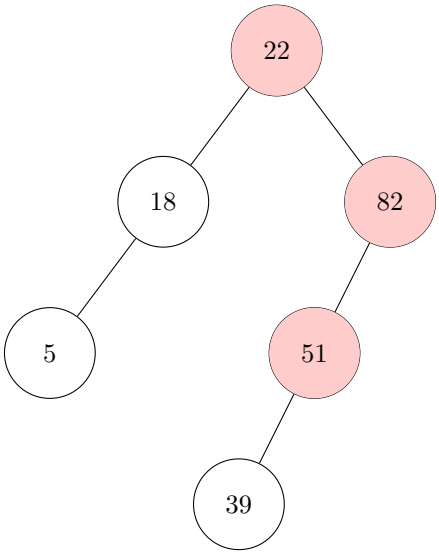
上の木を例に 51 を探してみましょう。根から順に探していきます。



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。



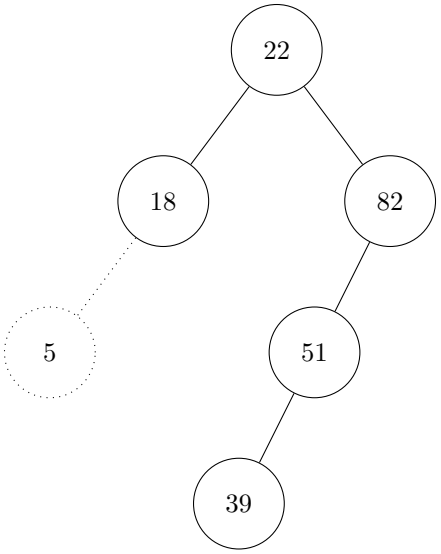
iii.　二分探索木の削除

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

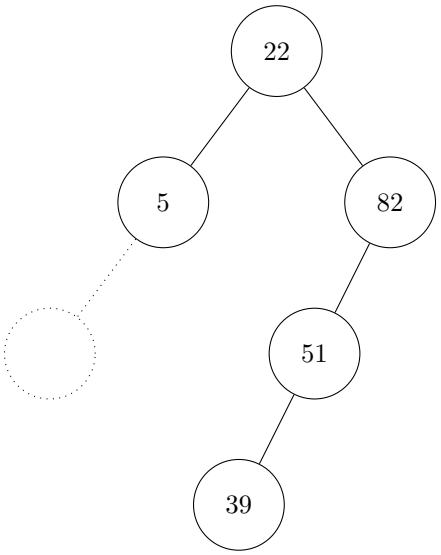
1. 子ノードがない場合

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。



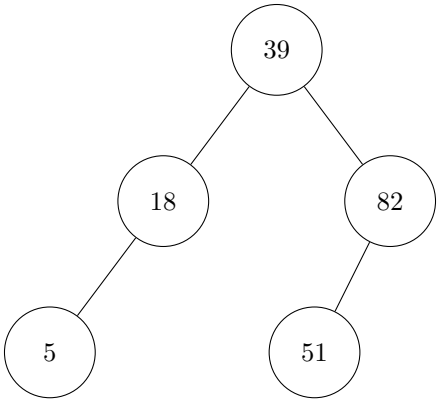
2. 子ノードが 1 つの場合

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。



3. 子ノードが 2 つの場合

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。



iv. 二分木の実装

コード 6　二分木の実装

```
class Node:
  def __init__(self, data: int) -> None:
      self.data: int = data
      self.right: Node | None = None
      self.left: Node | None = None

  def __str__(self) -> str:
      return str(self.data)

class BinarySearchTree:
  def __init__(self):
      self.root = None

  def create(self, array: list[int]) -> None:
      for i in range(len(array)):
          self.insert(array[i])

  def insert(self, data: int) -> None:
      if self.root is None:
```

```
            self.root = Node(data)
        else:
            self._insert_recursively(self.root, data)

    def _insert_recursively(self, node: Node, data: int) -> None:
        if data <= node.data:
            if node.left is None:
                node.left = Node(data)
            else:
                self._insert_recursively(node.left, data)
        else:
            if node.right is None:
                node.right = Node(data)
            else:
                self._insert_recursively(node.right, data)

    def search(self, value: int) -> Node | None:
        if self.root is None:
            return self.root

        return self._search_recursively(self.root, value)

    def _search_recursively(self, node: Node, value: int) -> Node | None:
        if node.data == value:
            return node

        if value < node.data:
            if node.left is None:
                return node.left
            else:
                return self._search_recursively(node.left, value)
        else:
            if node.right is None:
                return node.right
            else:
                return self._search_recursively(node.right, value)
```

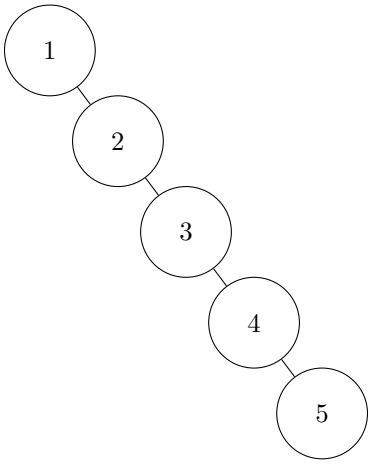
```
56
57    def delete(self, value: int) -> Node | None:
58        if self.root is None:
59            return self.root
60        else:
61            self.root = self._delete_recursively(self.root, value)
62
63    def _delete_recursively(self, node: Node, value: int) -> Node | None:
64        if value < 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

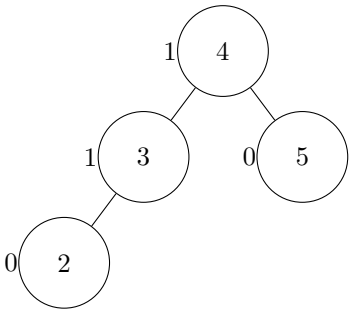
あるノードの左部分木の高さから右部分木の高さを引いた値を **balance factor** といいます。

- balance factor = 左部分木の高さ - 右部分木の高さ

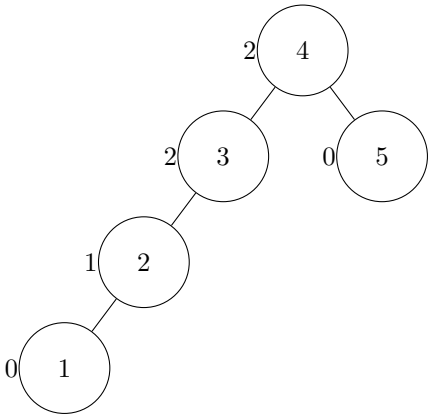
AVL 木の性質を balance factor を使って表すと以下のようになります。

- 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。

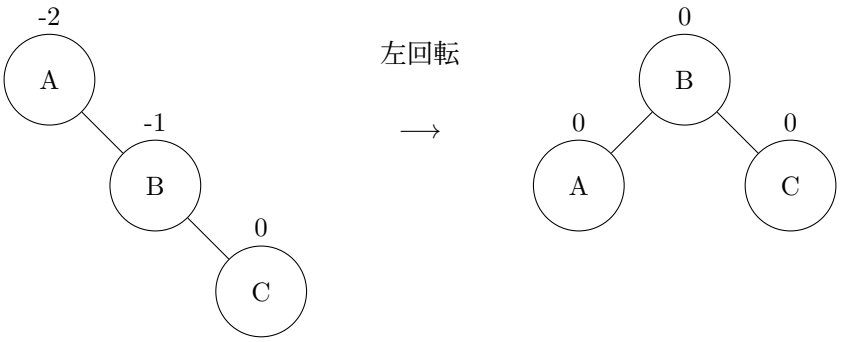


次は AVL 木の条件を満たしていない木です。

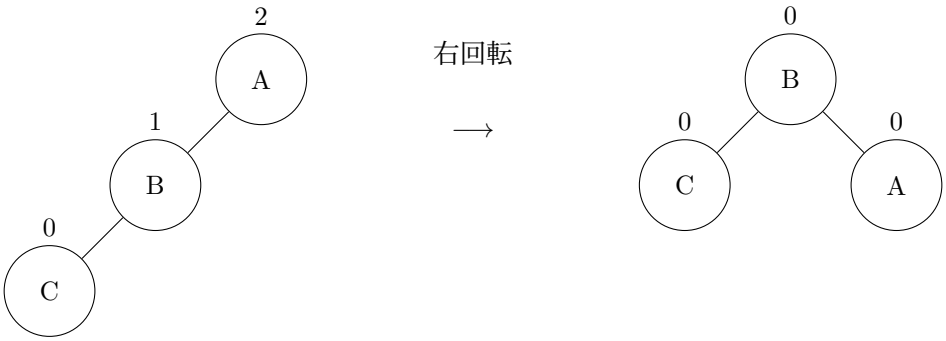


1.　回転

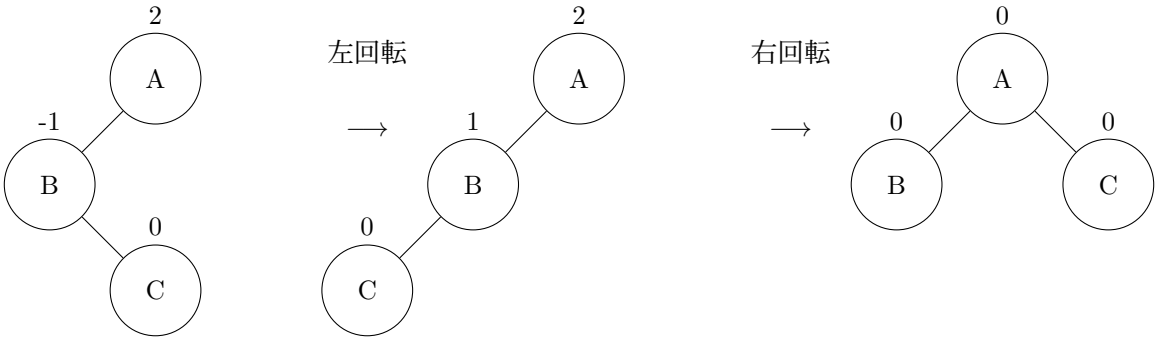
1. 左回転



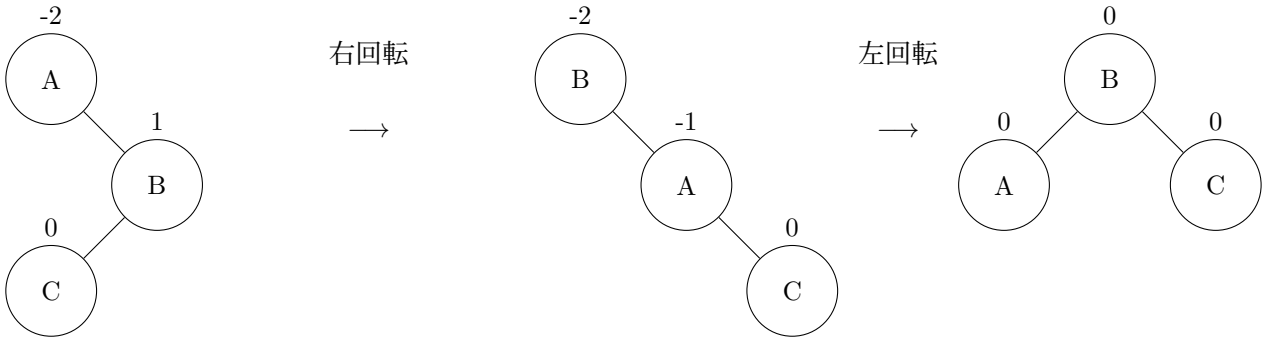
2. 右回転



3. 左右回転



4. 右左回転



iv. 赤黒木

v. B 木

vi. Treap

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/