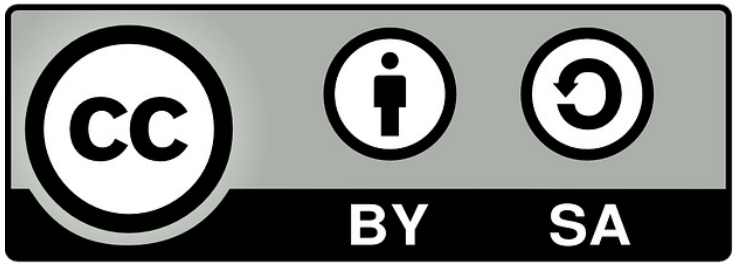
Algorithms (2024 Summer)
＃４：探索（サーチ）

矢谷 浩司
東京大学工学部電子情報工学科



探索（サーチ）とは

あるデータ集合（例えば配列）から，目的とする値を持った要素を探し出す．幅広い意味を持つ．

「配列の中で値が0のものを取り出す．」

「登録者の中で所属が東京大学の人を探す．」

「価格が1,000～1,500円の商品を取り出す．」

「『探索』に文字が似ている熟語を取り出す（例えば，探検，探究，検索，など）．」

探索（サーチ）とは

今日扱うのは，「配列からキーと完全一致する要素を見つけ出す（ない場合は「見つからなかった」と返す）」という，狭い意味での探索がメイン．

例）[9, 4, 2, 1, 8, 7, 6, 3, 5]から7を探す．

線形探索（1回目の計算量の所で紹介）

単純に頭からチェックしていく方法．$O(n)$

```
def linear_search(sequence, key):
    i = 0
    while i < len(sequence):
        if sequence[i] == key:
            return i
        i += 1
    return -1
```

定数倍効率化：番兵

キーと同じ値の要素を配列の最後に付け加える．これを「番兵（sentinel）」と呼ぶ．

先頭から順にキーに一致するかどうかをチェック．

一番最後まで一致していたら，「見つからなかった」として返す．それ以外の場合は，その時のindexを返す．

番兵があるので，必ず一致する場所があって終了する．

番兵付き線形探索

```
def linear_search2(sequence, key):
    i = 0
    sequence.append(key)   # 番兵をつける
    while sequence[i] != key:
        i += 1

    if i == len(sequence) - 1:
        return -1
    return i
```

何が違う？

linear_search

```
i = 0
while i < len(sequence):
        if sequence[i] == key:
                return i
        i += 1
return -1
```

linear_search2

```
i = 0
sequence.append(key)
while sequence[i] != key:
        i += 1

if i == len(sequence) - 1:
        return -1
return i
```

何が違う？

linear_search

```
i = 0
while i < len(sequence):
    if sequence[i] == key:
        return i
    i += 1
return -1
```

比較が2回

linear_search2

```
i = 0
sequence.append(key)
while sequence[i] != key:
    i += 1
if i == len(sequence) - 1:
    return -1
return i
```

比較が1回!

パフォーマンス比較例

与えられた配列の一番最後の要素と同じ値をキーとして線形探索を行う.

配列の長さ	番兵なし	番兵あり
1,000	169 usec	91.1 usec
10,000	1.81 msec	981 usec
100,000	23.5 msec	13.4 msec
1,000,000	238 msec	159 msec

改良版線形探索

ビッグオー記法では$O(n)$.

ただし，ループ内における処理の回数を半分にすることができるので，配列が大きくなれば差が出てくる.

とはいえ，本質的には変わらないので，もっと早くできないだろうか？

なお，

Pythonの場合，whileループをforループで置き換えると，比較をしない（range型のオブジェクトから順に要素を取得して処理をする）処理になり，比較の回数は減ります．

なので上記スライドでは意図的にwhileを使っています．ただし置き換えても計算量としては$O(n)$のままです．

forループの実装の中に比較がある言語の場合は，whileループをforループに置き換えても比較の回数は減りません．

例えば，for (int i = 0; i < n; i++) みたいな．

二分探索

配列がソートされているという前提.　（昇順に並んでいるなど）

非常に高速に探索できる手法.

二分探索

昇順に並んでいる配列の中央に位置する値とキーを比較.

キーの方が小さい：
　　配列の左側に探索範囲を絞る.
キーの方が大きい：
　　配列の右側に探索範囲を絞る.

絞った範囲の中央に位置する値と比較し，一致が見つかるか，絞った範囲が1になっても一致しないかまで続ける.

二分探索の例

[5, 18, 22, 28, 39, 48, 51, 68, 82, 94]から51を探す.

二分探索の例

[5, 18, 22, 28, 39, 48, 51, 68, 82, 94]から51を探す.

#1：[5, 18, 22, 28, 39, 48, 51, 68, 82, 94]と51を比較. キーの方が大きいので右側に検索範囲を絞る.

二分探索の例

[5, 18, 22, 28, 39, 48, 51, 68, 82, 94]から51を探す.

#1：[5, 18, 22, 28, 39, 48, 51, 68, 82, 94]と51を比較. キーの方が大きいので右側に検索範囲を絞る.

#2：[48, 51, 68, 82, 94]と51を比較. キーの方が小さいので左側に検索範囲を絞る.

二分探索の実装例

[5, 18, 22, 28, 39, 48, 51, 68, 82, 94]から51を探す.

#1：[5, 18, 22, 28, 39, 48, 51, 68, 82, 94]と51を比較. キーの方が大きいので右側に検索範囲を絞る.

#2：[48, 51, 68, 82, 94]と51を比較. キーの方が小さいので左側に検索範囲を絞る.

#3：[48, 51]と51を比較. キーの方が大きいので右側に検索範囲を絞り，残った1つの要素と比較すると一致.

二分探索の実装例

```
def binary_search(seq, key):
    # left, rightにそれぞれ最左端, 最右端を指定
    left = 0; right = len(seq) - 1
```

二分探索の実装例

```
def binary_search(seq, key):
    left = 0; right = len(seq) - 1

    while right >= left: # leftとrightが交差しない限り実行
        # leftとrightの真ん中を取ってくる
        pivot = (left + right) // 2
```

闭区间模型

二分探索の実装例

```
def binary_search(seq, key):
    left = 0; right = len(seq) - 1

    while right >= left:
        pivot = (left + right) // 2
        if seq[pivot] == key: return pivot      # 見つかった
```

二分探索の実装例

```
def binary_search(seq, key):
    left = 0; right = len(seq) - 1

    while right >= left:
        pivot = (left + right) // 2
        if seq[pivot] == key: return pivot
        elif seq[pivot] < key: left = pivot+1   # 右側に絞る
        else: right = pivot-1   # 左側に絞る

    return -1          # 見つからなかったら-1を返す
```

二分探索の計算量

1段階経ると，探索範囲はほぼ半分になる．

よって最悪の場合（最後まで見つからなかった場合）で
$O(\log n)$回の操作が必要となる．

よって，$O(\log n)$.

ただし配列が事前にソートされていることが前提で，
そのソートにかかる時間は除く．

パフォーマンス比較例

与えられた配列（整列済み）の一番最後の要素と同じ値をキーとして線形探索を行う.

配列の長さ	線形探索（番兵なし）	線形探索（番兵あり）	二分探索
1,000	172 usec	96.1 usec	11.2 usec
10,000	1.64 msec	817 usec	12.2 usec
100,000	16.4 msec	8.34 msec	17.9 usec
1,000,000	156 msec	81.6 msec	20.0 usec

\mu

二分探索のちょっとした拡張

先程のコードを少しいじると，「ある値より大きい要素の中で最も小さい値」や「ある値より小さい要素の中で最も大きい値」を求めることができる．

例）配列 [5, 18, 22, 28, 39, 48, 51, 68, 82, 94]において50より大きい最小の値とそのindexを求める．

→51で，indexは6.

二分探索のちょっとした拡張

さらにこれを応用すると，「ある条件を満たす最大（最小）の値を求める」といった類の問題に応用できる．

例）「$f(a,b,x) = ax^2 + b\log x, a > 0, b > 0$において，1から$N$までの整数$n$のうち，$f(a,b,n) \geq K$を満たす最小の$n$を求めよ．」

→まともにやると$O(N)$.

二分探索のちょっとした拡張

このは単調に増加する.

ある整数において，ならば，以上の値の全てに対してが成立する.

→つまり，以下の値に候補が絞られる！

もし，ならば，以下の値全てのに対してが成立する.

→つまり，以上の値に候補が絞られる！

二分探索をうまく使おう

```
left = 0     # 探索範囲の左端
right = 10**7    # 探索範囲の右端，十分大きな値にする

# 真ん中の値から探索をスタート
```

二分探索をうまく使おう

…

while [区間の幅が1になるまで続ける]:

    m = (left + right) // 2

    if $[f(a, b, m) \geq K]$:

        [探索範囲の右端を更新]

    else:

        [探索範囲の左端を更新]

[何を返せば良い？]

二分探索をうまく使おう

この場合，計算量は$O(\log N)$となり，劇的に向上！😆

可能性のある値の絞り込みを効率的にできる！

二分探索を使う問題に直接は見えなくても，「〜〜の条件を満たす最大値（最小値）」というタイプの問題は，二分探索を使える可能性あり．

B - 花束 Editorial

Time Limit: 2 sec / Memory Limit: 256 MB

問題文

高橋君は赤い花を R 本、青い花を B 本持っています。高橋君は次の 2 種類の花束を作ることができます。

- x 本の赤い花と 1 本の青い花からなる花束
- 1 本の赤い花と y 本の青い花からなる花束

高橋君が作ることのできる花束の個数の最大値を求めてください。すべての花を使い切る必要はありません。

制約

- $1 \leqq R$, $B \leqq 10^{18}$
- $2 \leqq x$, $y \leqq 10^{9}$

入力

入力は以下の形式で標準入力から与えられる。

```
R B
x y
```

出力

高橋君が作ることのできる花束の個数の最大値を出力せよ。

https://atcoder.jp/contests/arc050/tasks/arc050_b

解法の方針

「～～の条件を満たす最大値（最小値）」というタイプの問題になっている.

問題を以下のように読み替えてみよう.

「x，y，赤の花の総数，青の花の総数が与えられた時，x本の赤い花と1本の青い花の花束と，1本の赤い花とy本の青い花の花束をあわせてK個作ることができるかを判定せよ．」

解法の方針

あるKに対して，作ることができる．

→K+1から2つの花の総数の値の間に求める値が存在する．

あるKに対して，作ることができない．

→1からK-1の間に求める値が存在する．

このKの値の範囲を二分探索で絞っていけば良い！

解法の方針

x本の赤い花と1本の青い花の花束の個数：m
1本の赤い花とy本の青い花の花束の個数：n
赤い花の総数：R，青い花の総数：B，花束の総数：K

とすると，与えられた条件は，

$$m + n = K, \qquad mx + n \leq R, \qquad m + ny \leq B$$

と表される．

解法の方針

これらの式を変形すると，

$$mx + n \leq R \qquad m + ny \leq B$$

$$mx + (K - m) \leq R \qquad (K - n) + ny \leq B$$

$$m \leq \frac{R - K}{x - 1} \qquad n \leq \frac{B - K}{y - 1}$$

となる．

解法の方針

さらに，$m+n=K$なので，

$$K=m+n\leq\left\lfloor\frac{R-K}{x-1}\right\rfloor+\left\lfloor\frac{B-K}{y-1}\right\rfloor$$

（割り算は切り捨て）

解法の方針

よって，あるKに対して，

$$K \leq \left\lfloor \frac{R-K}{x-1} \right\rfloor + \left\lfloor \frac{B-K}{y-1} \right\rfloor$$

を満たすかどうかが分かれば，そのKの数だけ花束を作ることができるかどうかがわかる！

この判定をKを二分探索で範囲を絞りながら繰り返す.

二分探索の使い方

可能性のある値の範囲を毎回半分に絞っていき，その結果として条件を満たす最大値や最小値を効率的に求めることができる！

与えられた配列からある値を探す方法，と考えるのではなく，**可能性のある値の範囲を効率良く絞る方法**と理解すると応用の幅が広がる．

書籍やWeb上にもいろんな解説がありますので，ぜひ参考にし，自分のものにしてみてください．😉

さて，二分探索の嬉しくないところは？

配列のまま扱うならば，ソートが必須．

ソートには一般的には$O(n \log n)$かかる（次回の授業で紹介予定）ので，もし探索を1回しか行わないなら，線形探索の方が時間計算量が小さい．

データが出たり入ったりする場合には，毎回ソートするのは手間．😩

他のやり方は？

データ構造で解決：二分木を作る

二分探索木という．

BST　不一定是complete binary tree

左の子ノードは親ノードよりも小さく，右の子ノードは親よりも大きい．

つまり，[左] < [親] < [右]．片方だけなら等号を入れることも出来る（重複はそもそも考えないことが多いが）．

各ノードの子ノードは最大でも2つ．

二分木を作る

根ノードから比較をスタート．根ノードの値と追加すべき値を比較する．

追加する値のほうが小さい：

今比較しているノードに，左の子ノードが存在する：

左の子ノードに移動して，比較をする．

存在しない：

左の子ノードとして追加

二分木を作る

追加する値のほうが大きい：

今比較しているノードに，右の子ノードが存在する：

右の子ノードに移動して，比較をする．

存在しない：

右の子ノードとして追加

以上，追加が行われるまで繰り返す．

二分木を作る

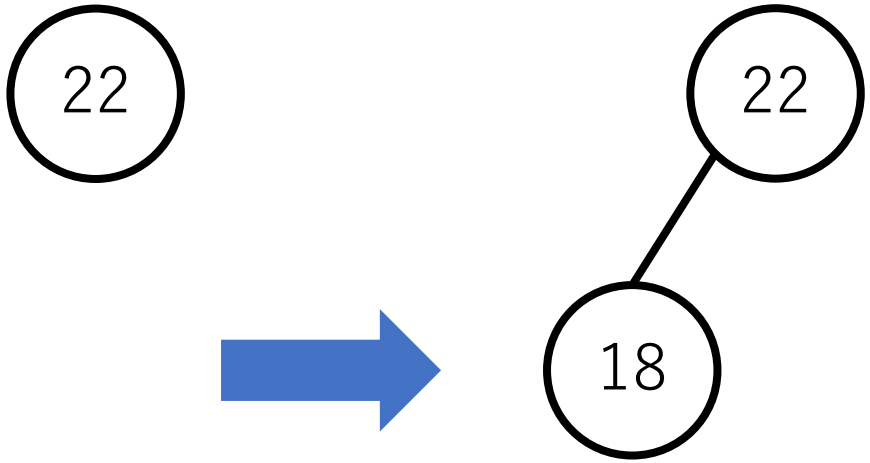
例：[22, 18, 5, 82, 51, 39, 48, 94, 68, 28]

二分木を作る

例：[22, 18, 5, 82, 51, 39, 48, 94, 68, 28]

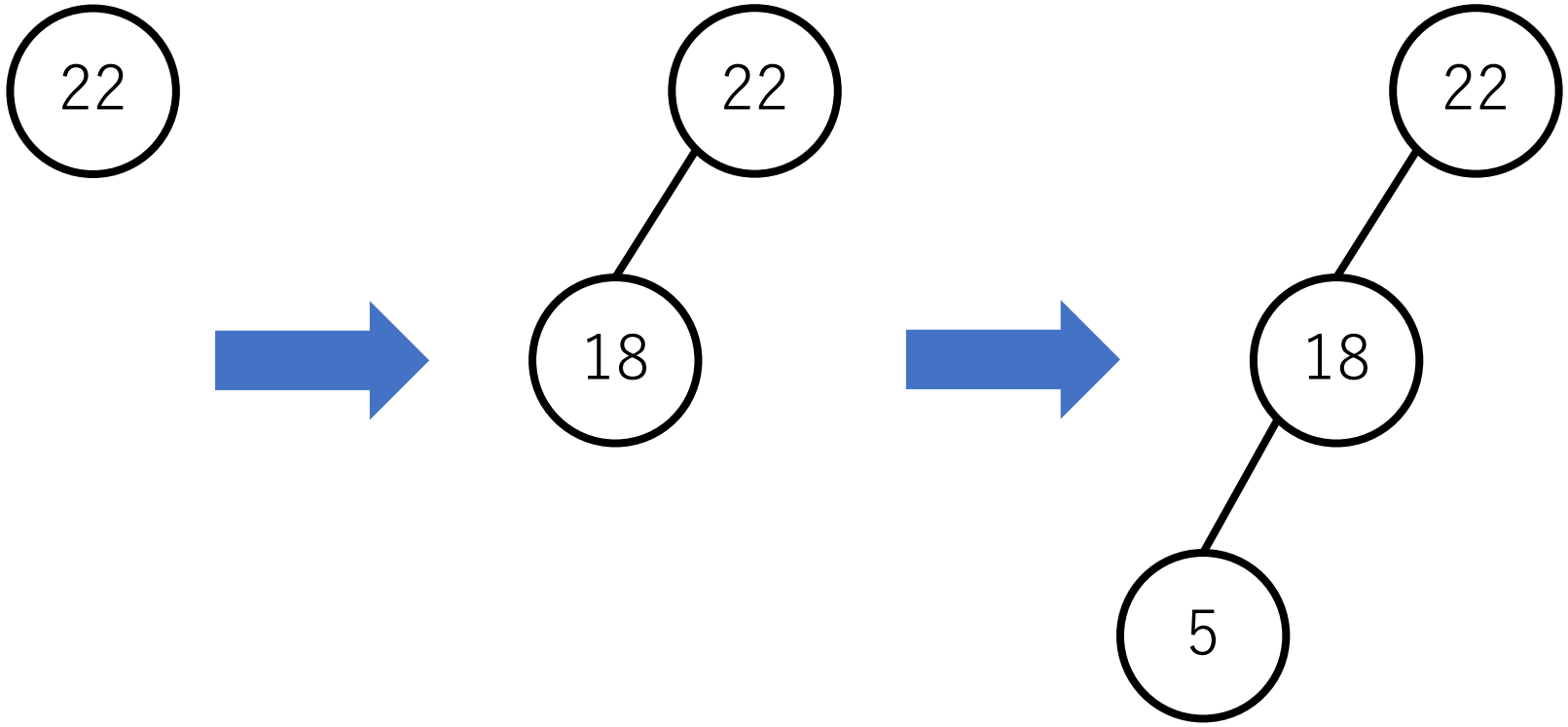
二分木を作る

例：[22, 18, 5, 82, 51, 39, 48, 94, 68, 28]



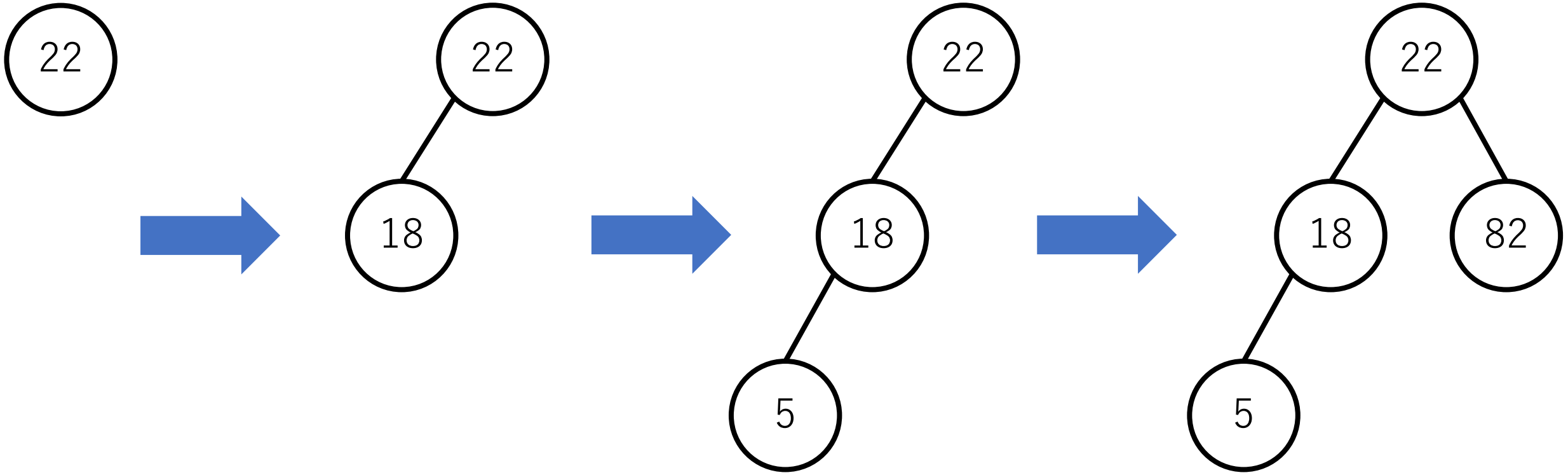
二分木を作る

例：[22, 18, 5, 82, 51, 39, 48, 94, 68, 28]



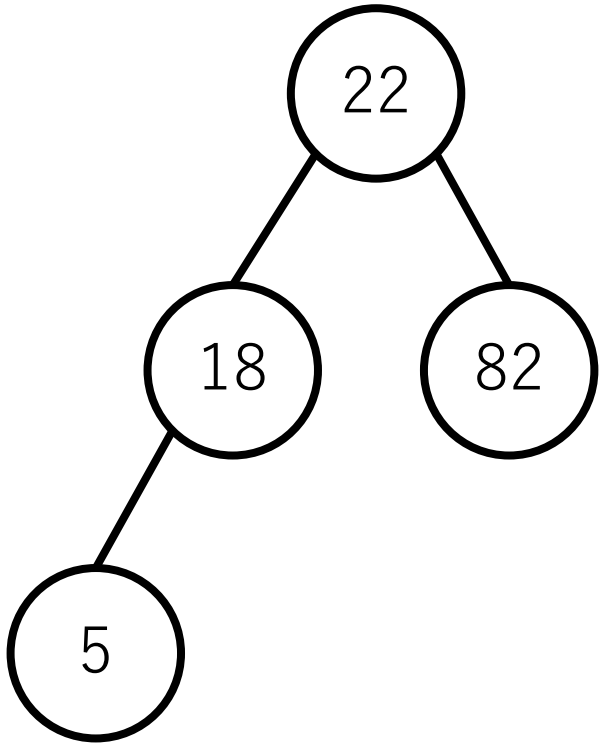
二分木を作る

例：[22, 18, 5, 82, 51, 39, 48, 94, 68, 28]



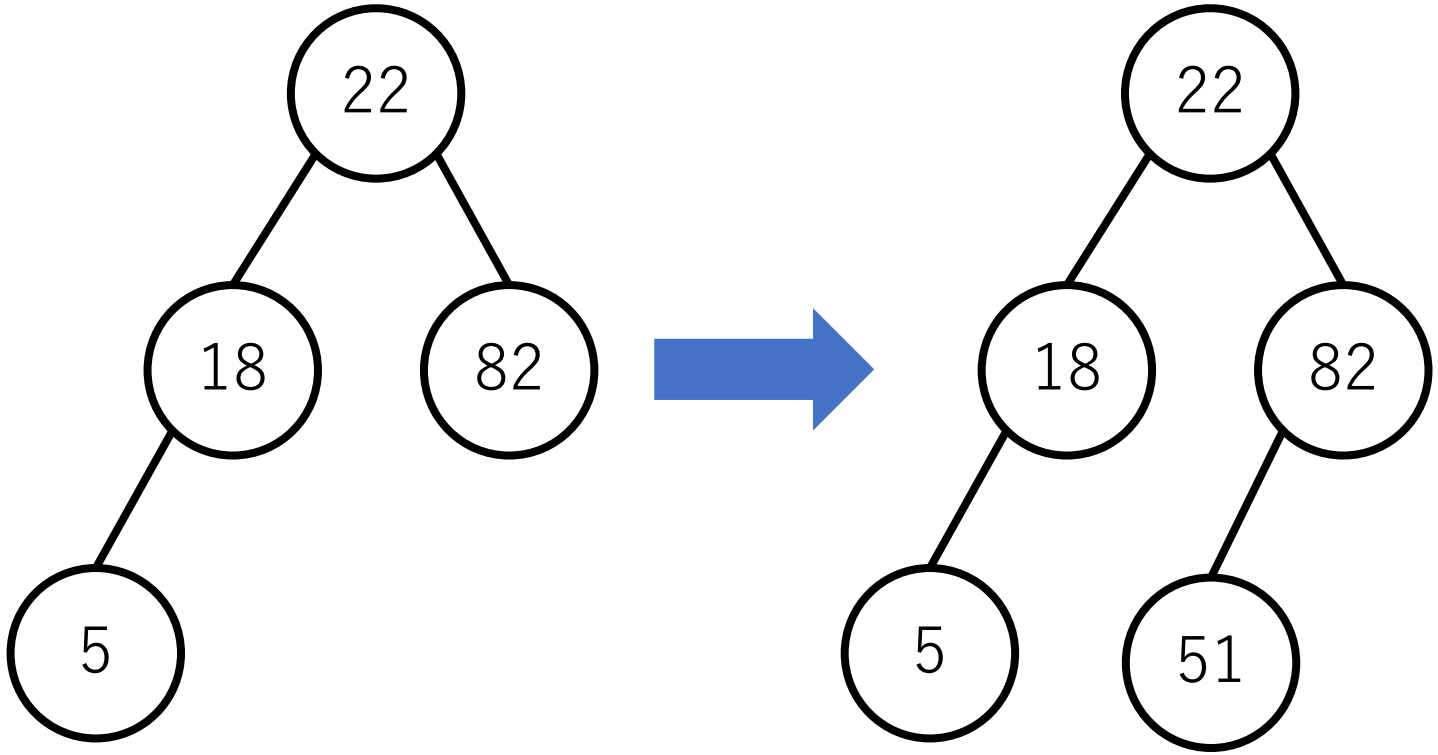
二分木を作る

例：[22, 18, 5, 82, 51, 39, 48, 94, 68, 28]



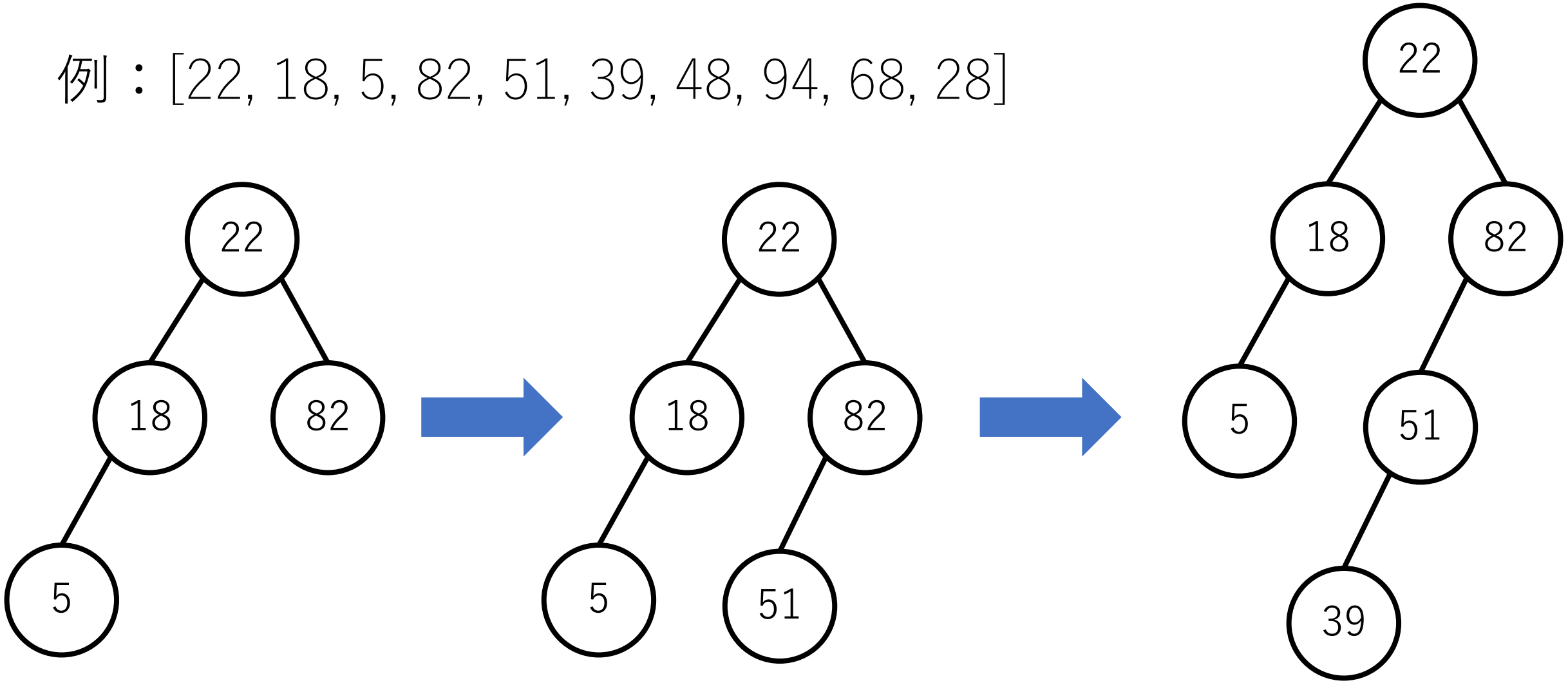
二分木を作る

例：[22, 18, 5, 82, 51, 39, 48, 94, 68, 28]



二分木を作る

例：[22, 18, 5, 82, 51, 39, 48, 94, 68, 28]

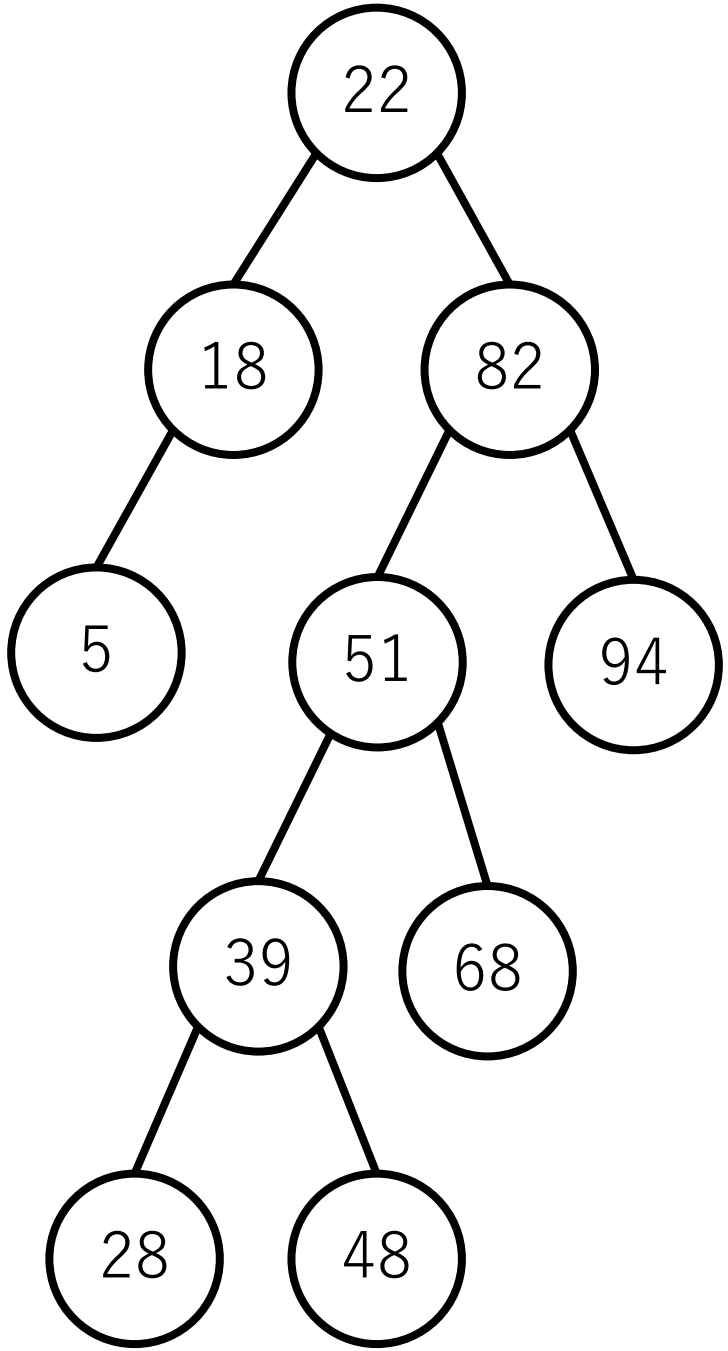


二分木を作る

例：[22, 18, 5, 82, 51, 39, 48, 94, 68, 28]

最終的には右のような木になる.

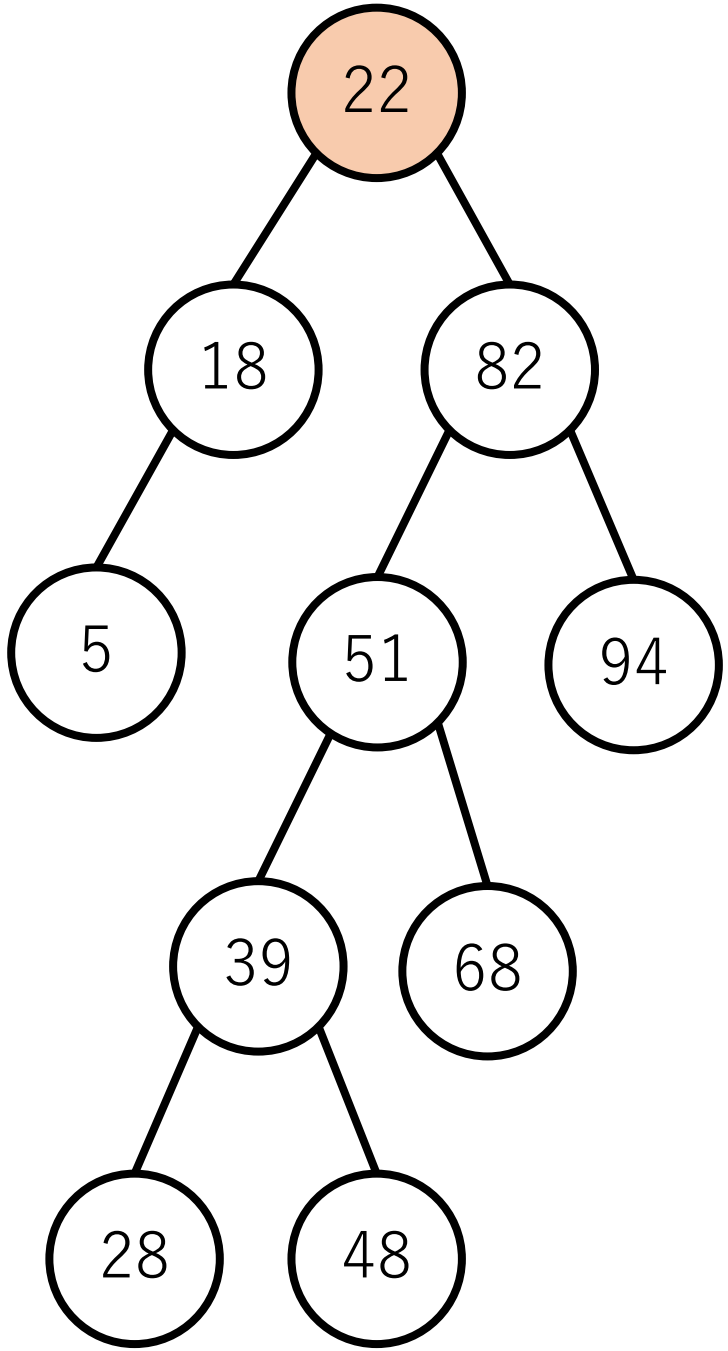
各ノードより小さな値は左側に，大きな値は右側に接続されるようになる.



二分木を使って探索

例：68を探す.

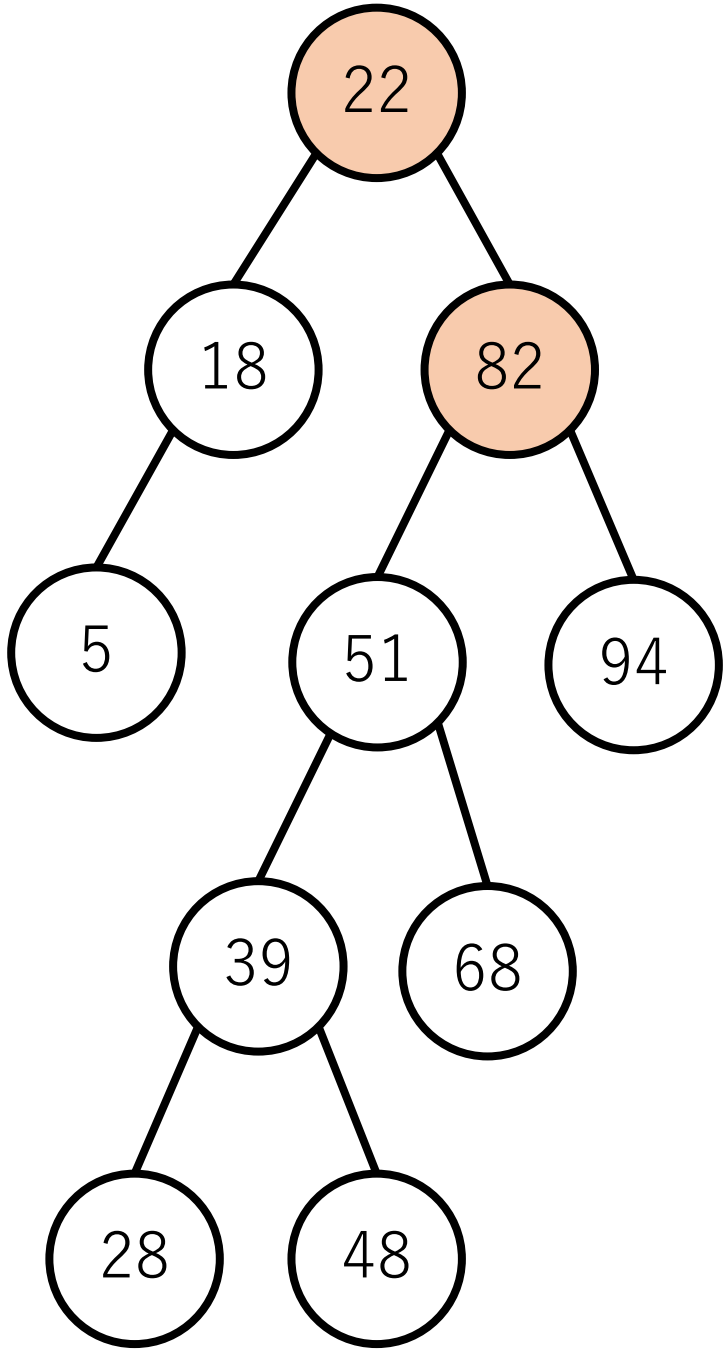
根ノードからスタート．68は22より
大きいので，右の子ノードに移動.



二分木を使って探索

例：68を探す.

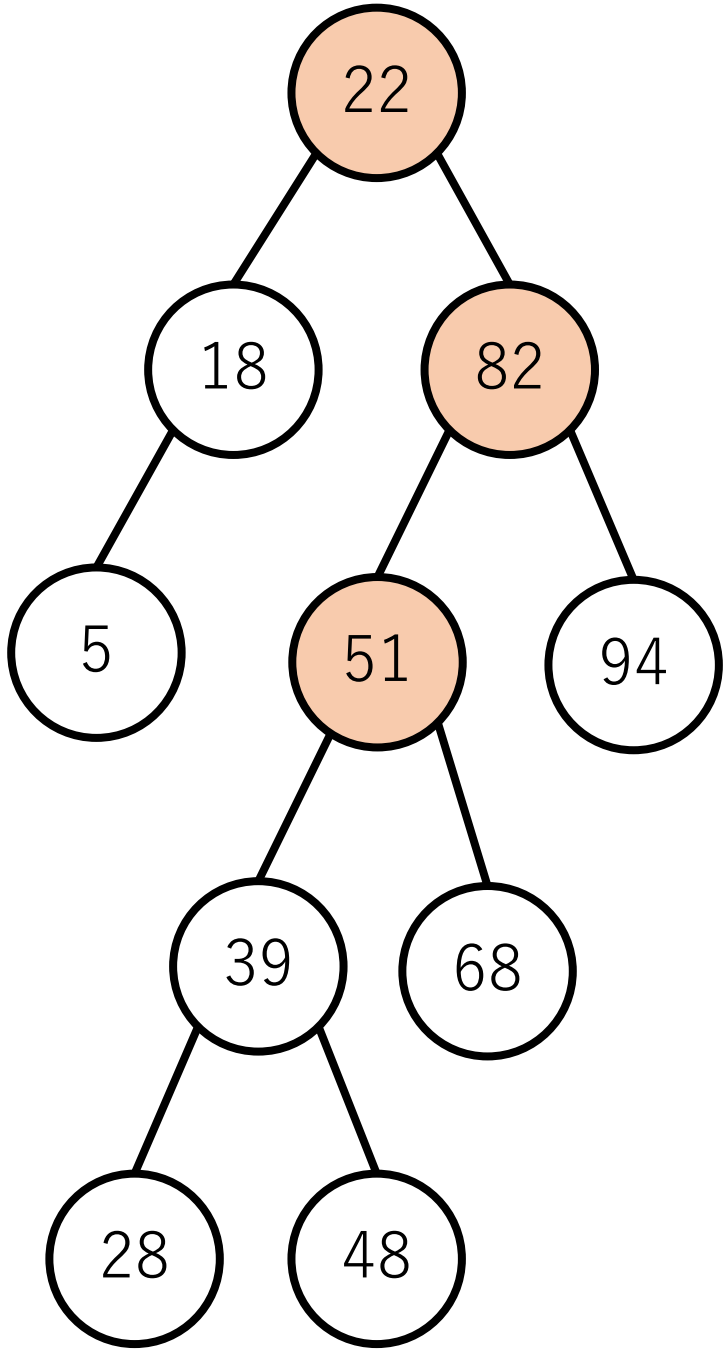
68は82より小さいので，左の子ノードに移動.



二分木を使って探索

例：68を探す.

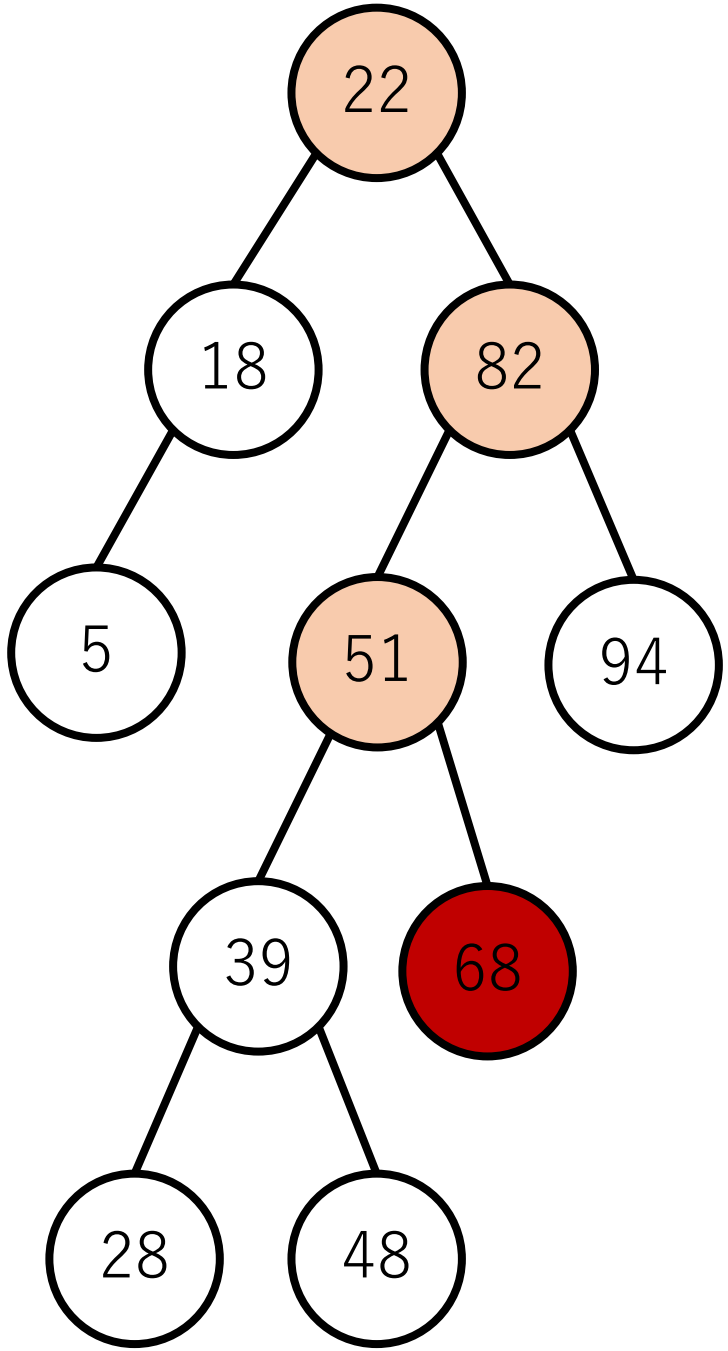
68は51より大きいので，今度は
右の子ノードに移動.



二分木を使って探索

例：68を探す.

68を発見！

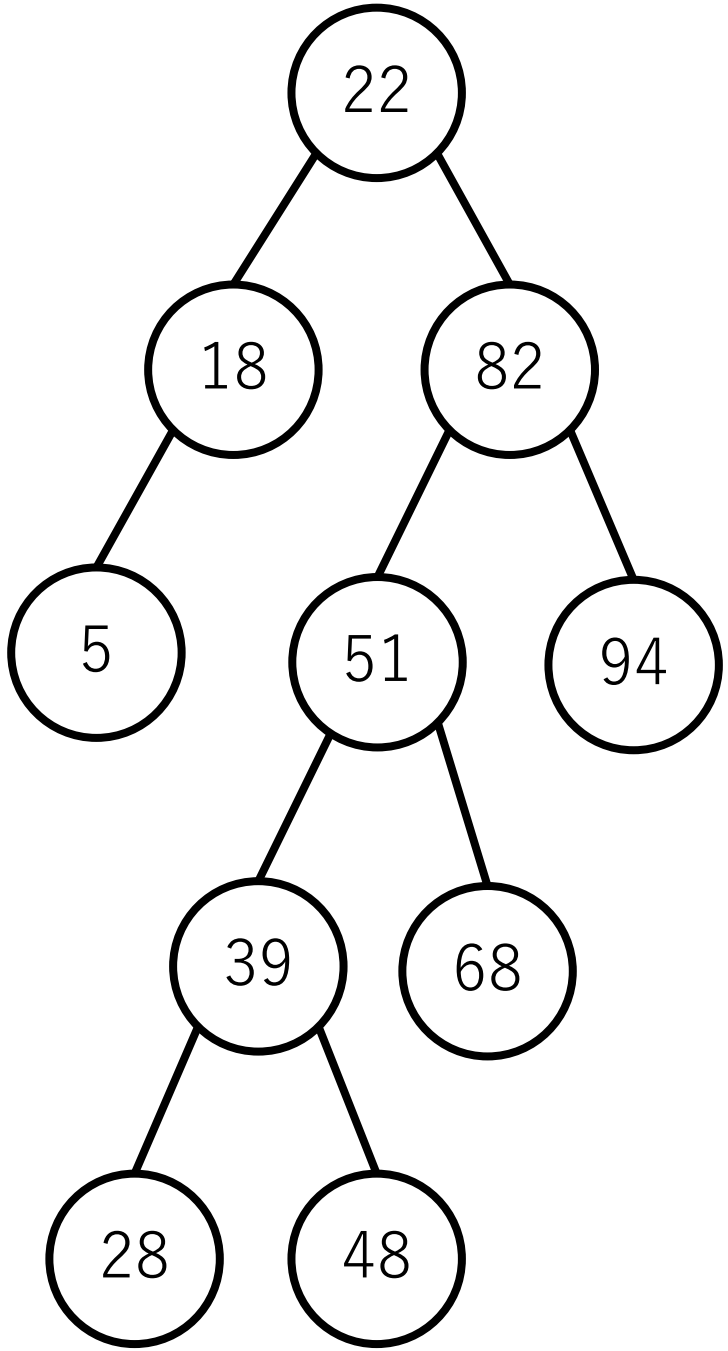


ノードの削除

3パターンありえる.

削除対象ノードの子ノードが：

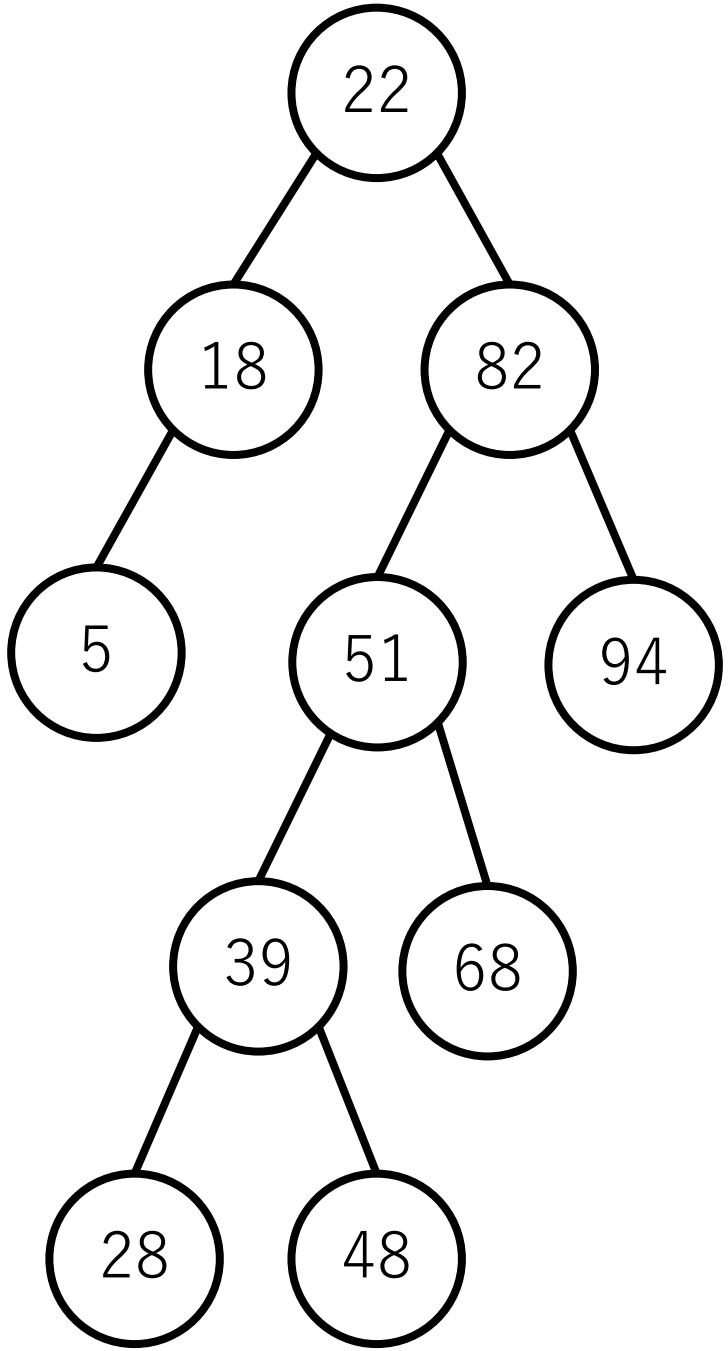
- 0個の場合
- 1個の場合
- 2個の場合



ノードの削除

削除対象ノードの子ノードが0個の場合

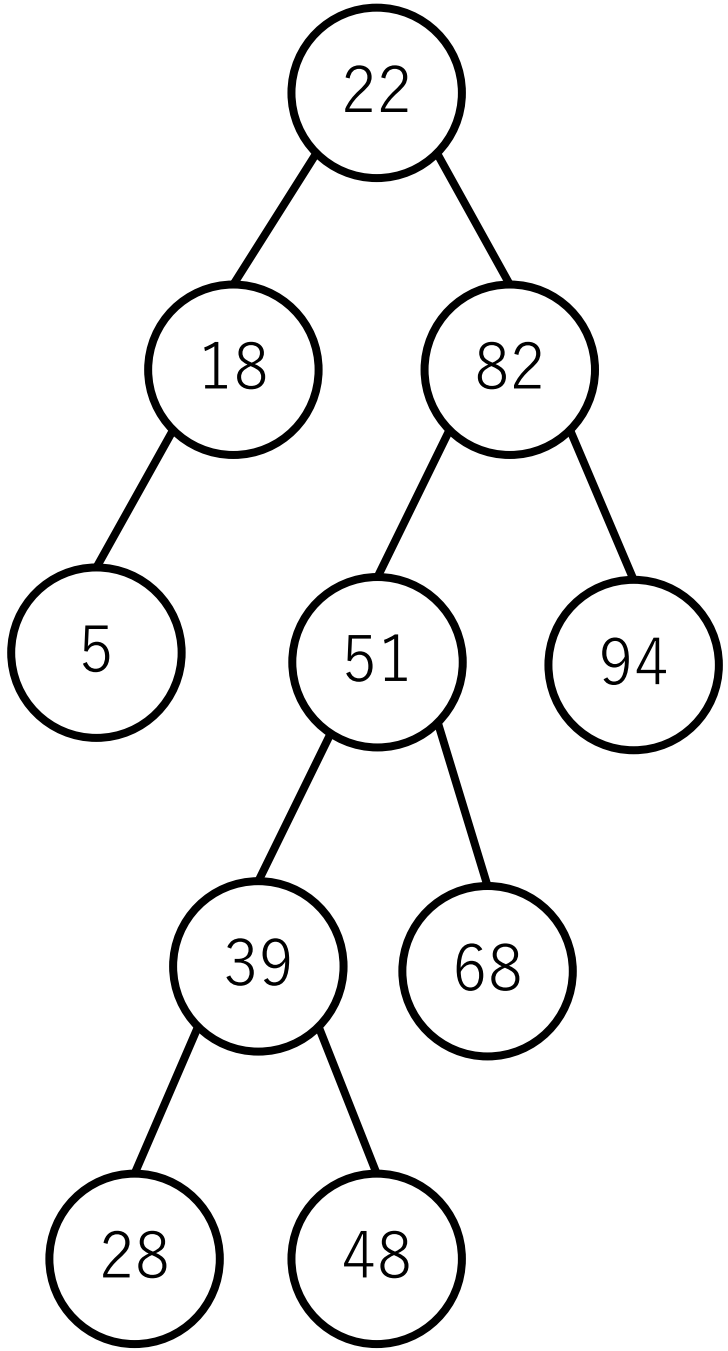
単に削除対象のノードを削除すればよい.



ノードの削除

削除対象ノードの子ノードが1個の場合

子ノードを削除したノードの場所に
引き上げてくる.

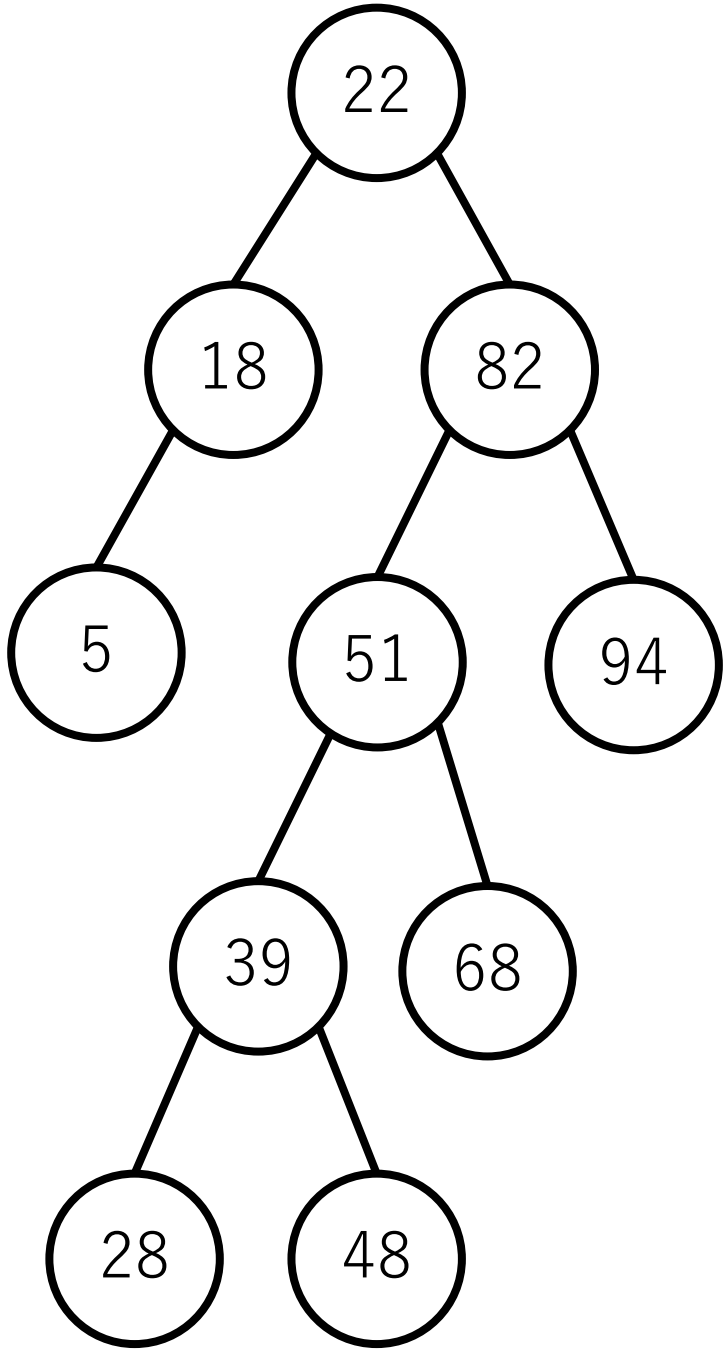


ノードの削除

削除対象ノードの子ノードが2個の場合

右の部分木で最も小さい値のノードか，
左の部分木で最も大きい値のノードを
削除したノードの場所に引き上げてくる．

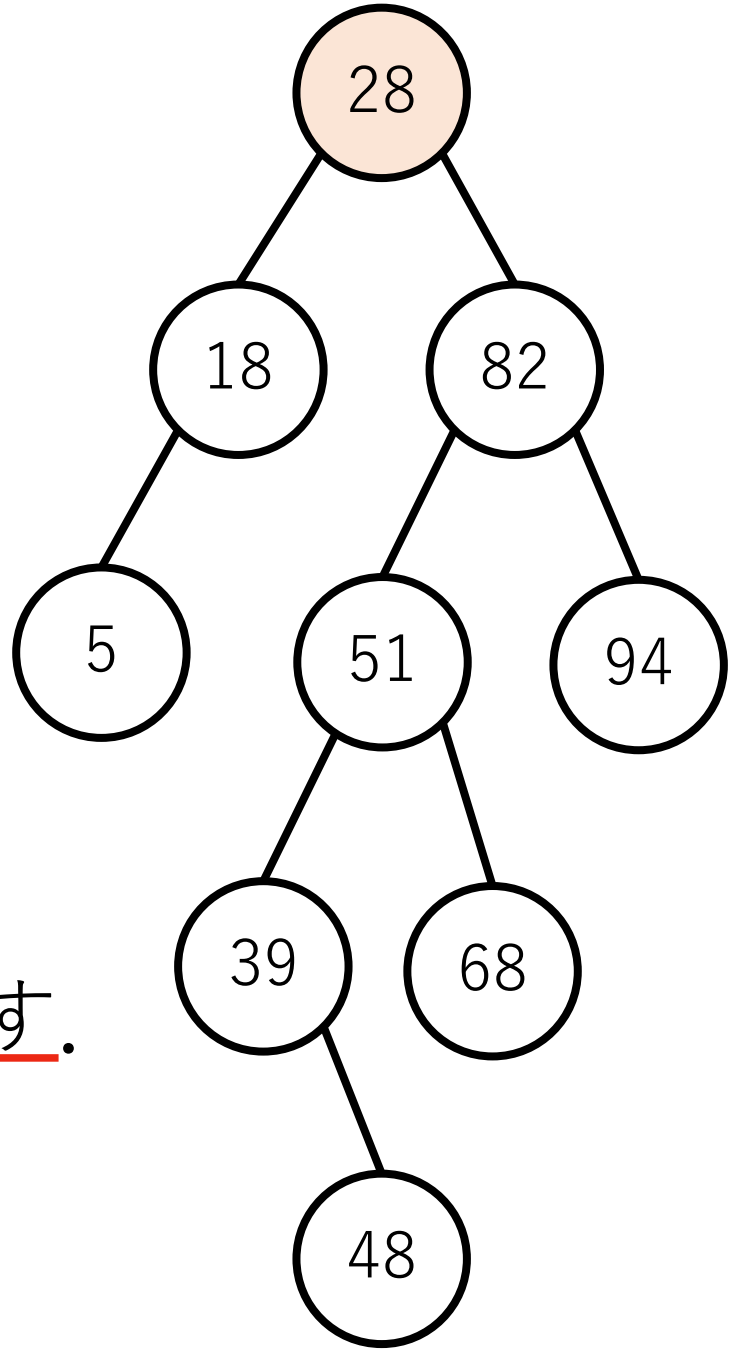
前者を次節点（successor），後者を
前節点（predecessor）と呼ぶ．



ノードの削除

右の部分木で最も小さい値のノード
（次節点）を上げる場合を考えてみる.

この例では28.



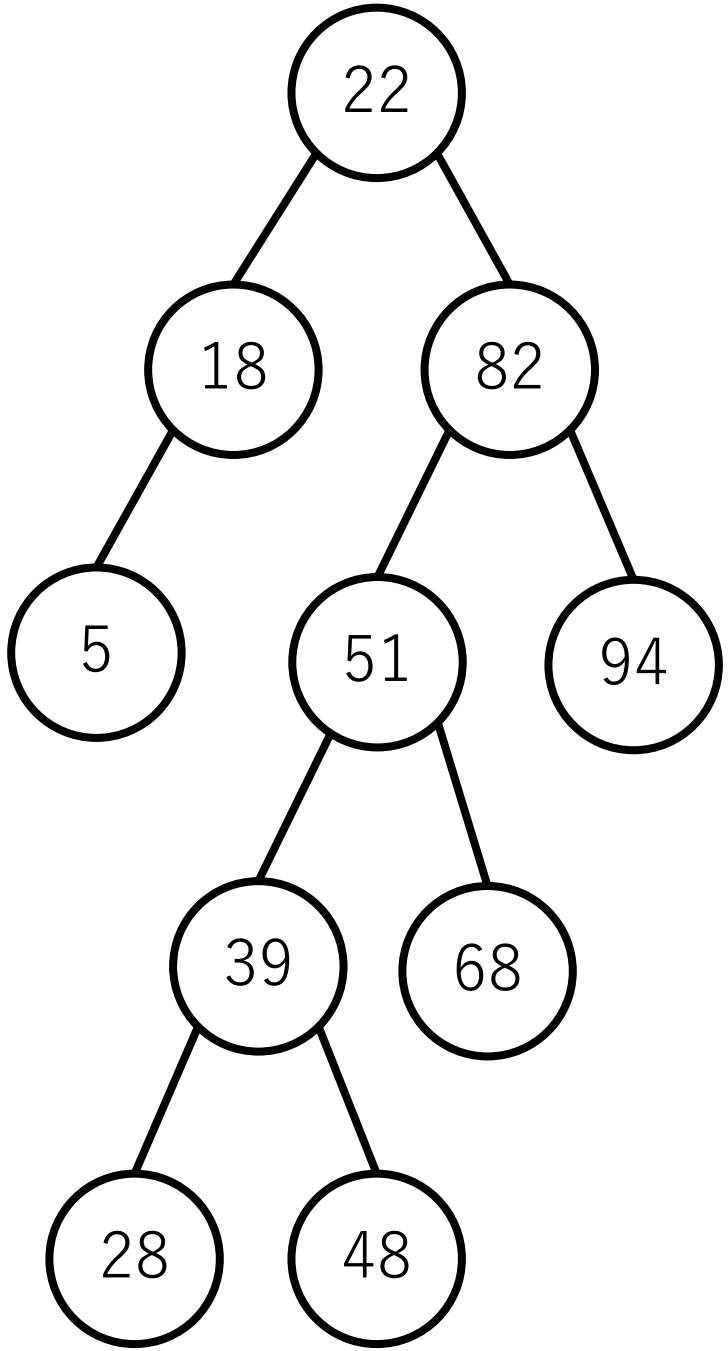
28は左の部分木の最大の値よりも大きく,
さらに右の部分木のどの値よりも小さい.

→削除後も二分探索木の制約を満たす.

ノードの削除

次節点はどうやったら求められる？

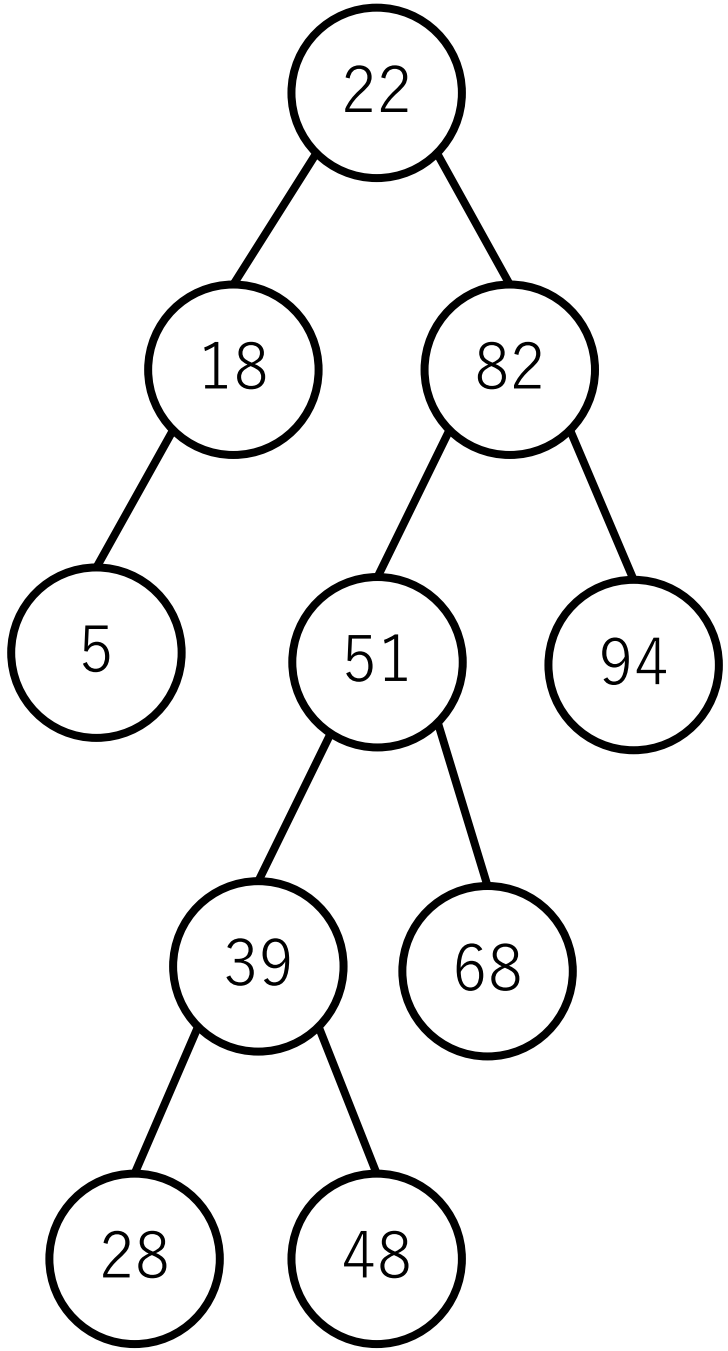
自ノードの右の部分木において，左子ノードがないノードまで左子ノードを辿れるだけ辿っていけばよい．



ノードの削除

具体的な実装としては,

- 削除対象ノードを探す.
- そのノードの次節点を探し，削除対象のノードの値を次節点の値で置き換える.
- 次節点について削除を行う．つまり，上2つの処理を再帰的に行なっていく.



二分木の実装例

```
# ノードの構造体
class Node:
        def __init__(self, value):
                self.data = value# 自分自身の値
                self.left = None          # 左ノードのNode
                self.right = None# 右ノードのNode

        def print(self):
                print((str(self.left.data) if self.left!=None else "") +
                " / " + str(self.data) + " / " + (str(self.right.data)
                if self.right!=None else ""))
```

二分木の実装例

```
# 二分木のクラス
class BinarySearchTree:
        def __init__(self):
                self.root = None        # 根ノード

        # リストから木を一気に作るメソッド
        def createTree(self, array):
                for i in range(len(array)):
                        self.insert(array[i])
```

二分木の実装例

```
class BinarySearchTree:
    def insert(self, value):
        # 再帰を使ってノードを挿入する場所を探す.
        if not self.root: self.root = Node(value)
        else: self._insertRec(self.root, value)
```

self.root == None

从某个节点开始递归插入

二分木の実装例

```
class BinarySearchTree:
    def _insertRec(self, node, value):
        # valueが今のノードの値より小さい場合,
        # 左子ノードを辿る. なければ今のノードの
        # 左子ノードとして追加.
        if value < node.data:
            if node.left: self._insertRec(node.left, value)
            else: node.left = Node(value)
```

二分木の実装例

```
class BinarySearchTree:
    def _insertRec(self, node, value):
        ...

        # valueが今のノードの値より大きい場合
        # 右子ノードを辿る．なければ今のノードの
        # 右子ノードとして追加.
        else:
            if node.right: self._insertRec(node.right, value)
            else: node.right = Node(value)
```

二分木の実装例

```
class BinarySearchTree:
    def search(self, value): #探索を実行するメソッド
        node = self.root
        while node:
            node.print()
            if node.data == value: print("found!"); return
            elif value < node.data: #左子ノードへ行く
                print("go left"); node = node.left
            else: #右子ノードへ行く
                print(“go right”); node = node.right
        print("not found")
```

二分木の実装例

```
class BinarySearchTree:
    def delete(self, value): #削除を再帰で実行するメソッド
        self.root = self._deleteRec(self.root, value)

    def _findSuccessor(self, node): #次節点を探すメソッド
        while node.left:
            node = node.left
        return node
```

右子树中最小的节点

二分木の実装例

```
class BinarySearchTree:
        def _deleteRec(self, node, value):
                # 子ノードがこれ以上ない場合，再帰を終了.
                if not node: return node
                # 所望のノードをsearch同様に探す.
                if value < node.data:
                        node.left = self._deleteRec(node.left, value)
                elif value > node.data:
                        node.right = self._deleteRec(node.right, value)
```

二分木の実装例

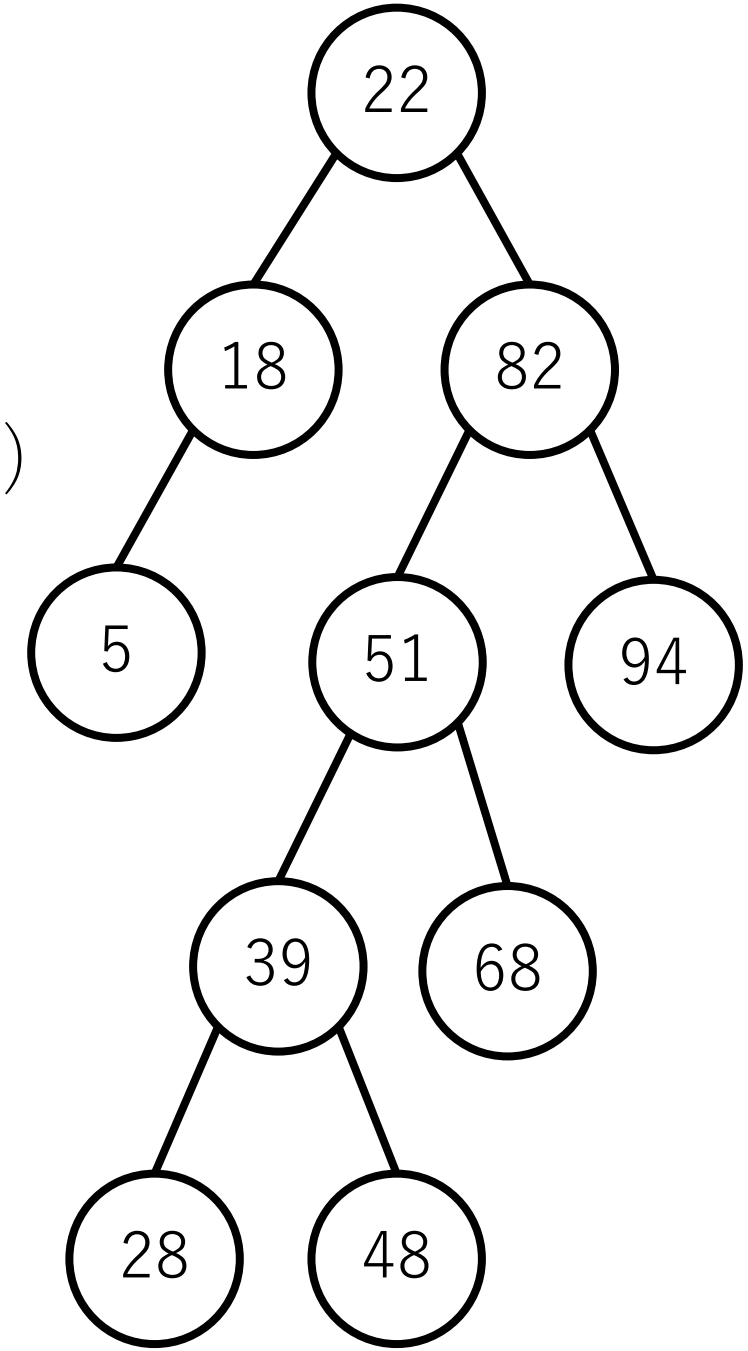
```
class BinarySearchTree:
    ...
        else:  value == node.data
            # 子ノードが0か1つの場合
            # （子ノードが0なら最初の行が実行され,
            # node.rightに入っているNoneが返る．）
            if not node.left: return node.right
            elif not node.right: return node.left
```

二分木の実装例

```
class BinarySearchTree:
        …
                else:
                        …
                        # 子ノードが2つの場合
                        sucNode = self._findSuccessor(node.right)
                        node.data = sucNode.data
                        node.right = self._deleteRec(node.right,
                                                     sucNode.data)
                return node
```

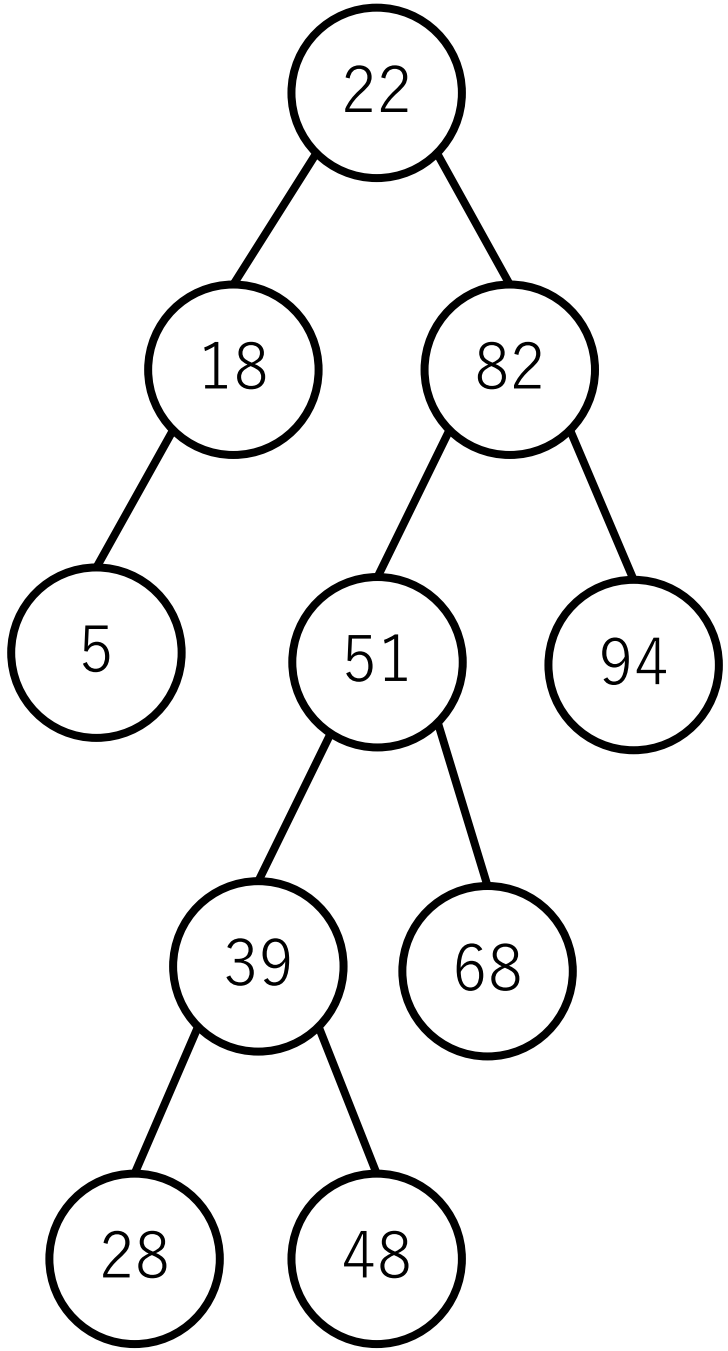
二分木の実行例

```
a = BinarySearchTree()
a.createTree([22, 18, 5, 82, 51, 39, 48, 94, 68, 28])
```



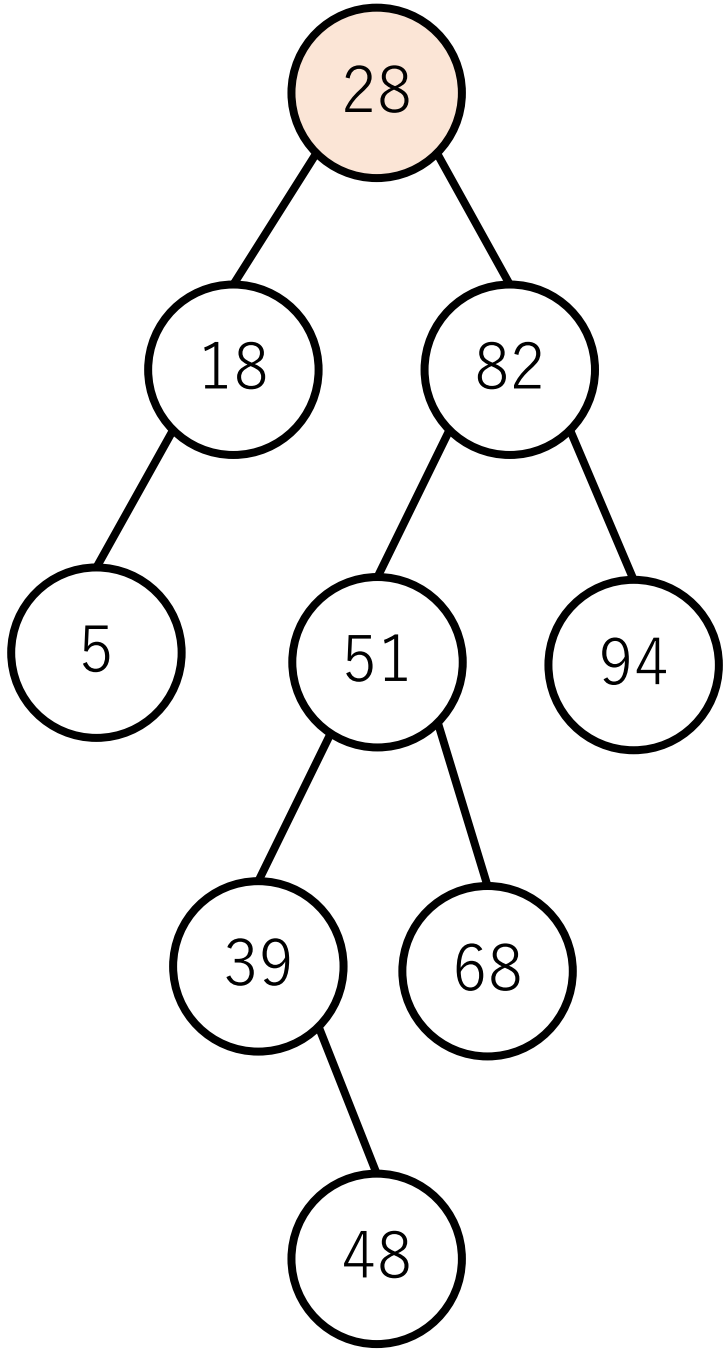
二分木の実行例

a.search(28)
---- 実行結果 ----
18 / 22 / 82
go right
51 / 82 / 94
go left
39 / 51 / 68
go left
28 / 39 / 48
go left
/ 28 /
found!



二分木の実行例

a.delete(22)

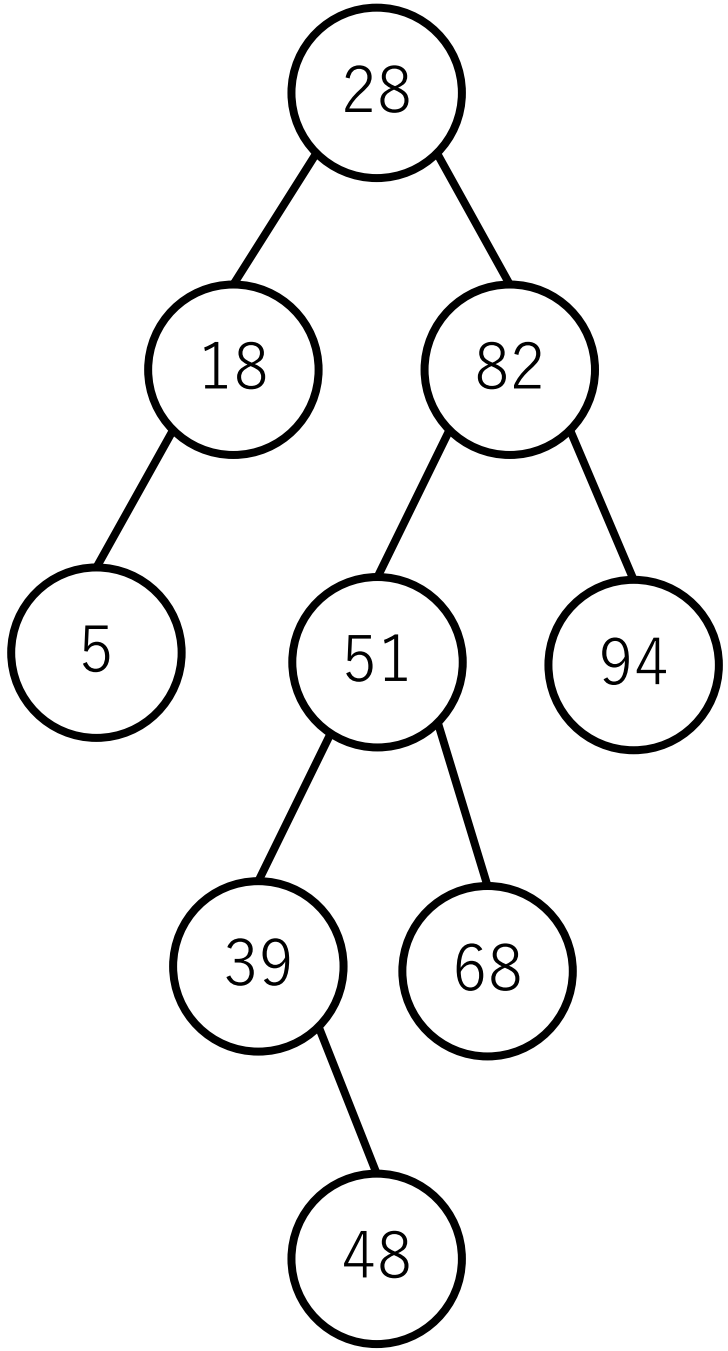


二分木の実行例

a.search(28)

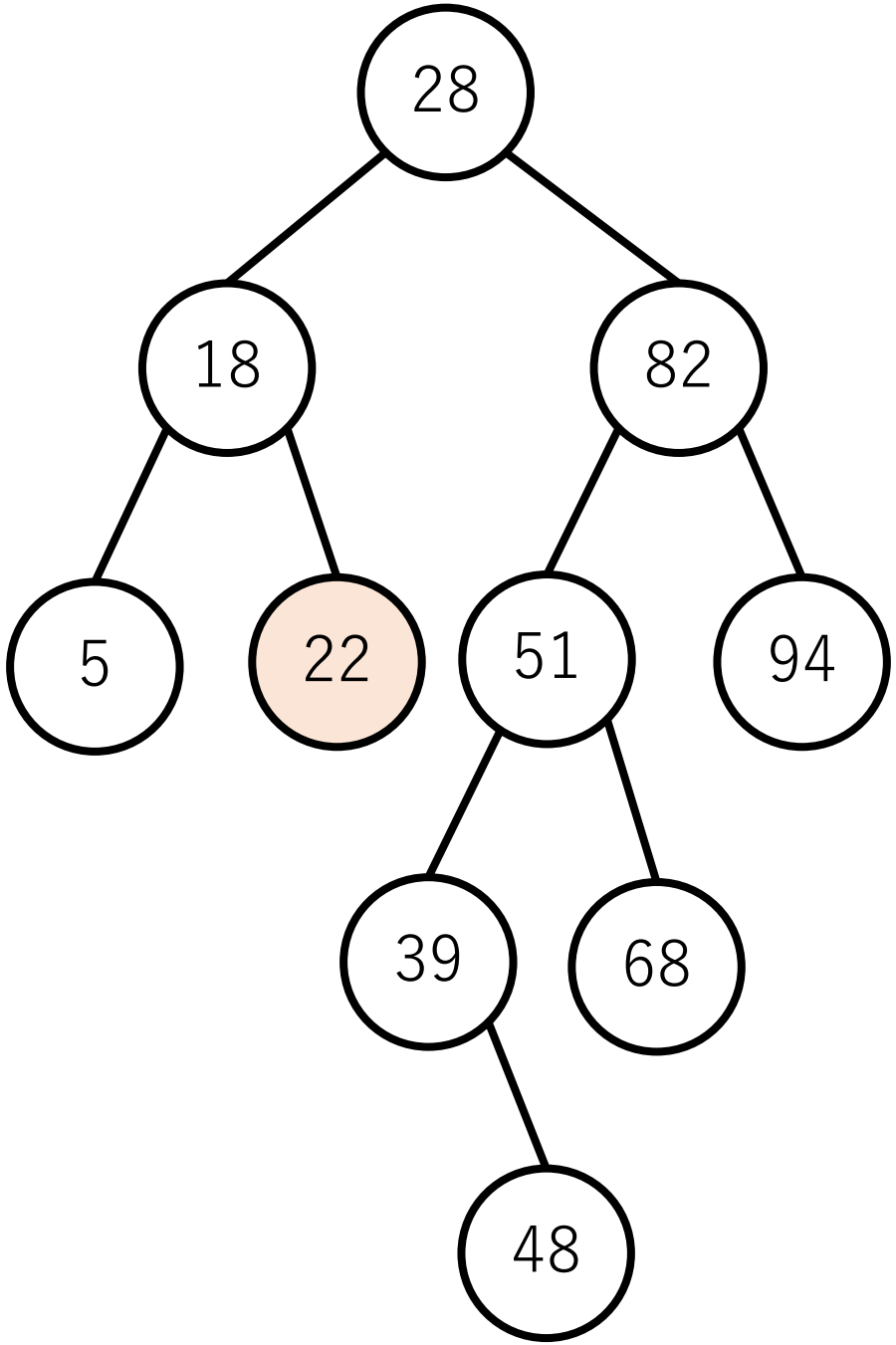
---- 実行結果 ----

18 / 28 / 82 found!



二分木の実行例

a.insert(22)
a.search(22)
---- 実行結果 ----
18 / 28 / 82
go left
5 / 18 / 22
go right
/ 22 /
found!

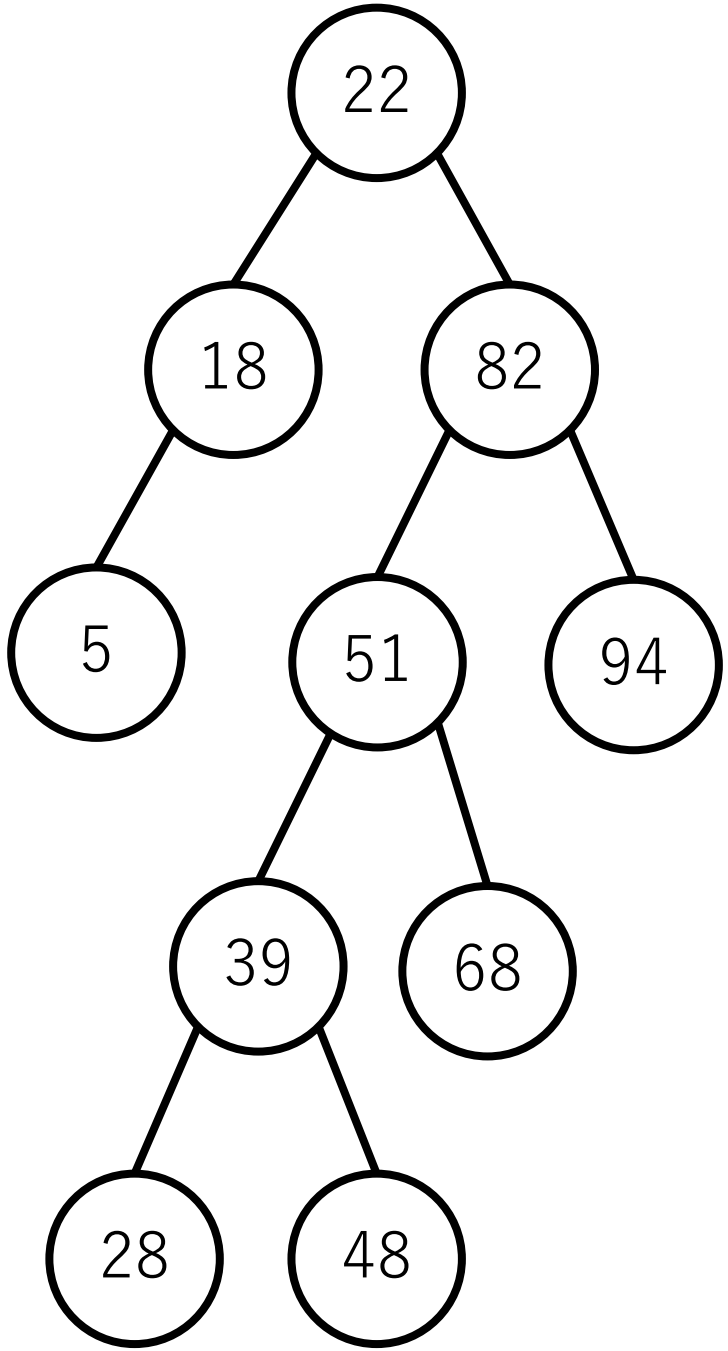


二分木を使った探索の計算量

挿入：

木が「普通の形」であれば，$O(\log n)$ 回比較をしたあとで追加できると期待できるので，$O(\log n)$.

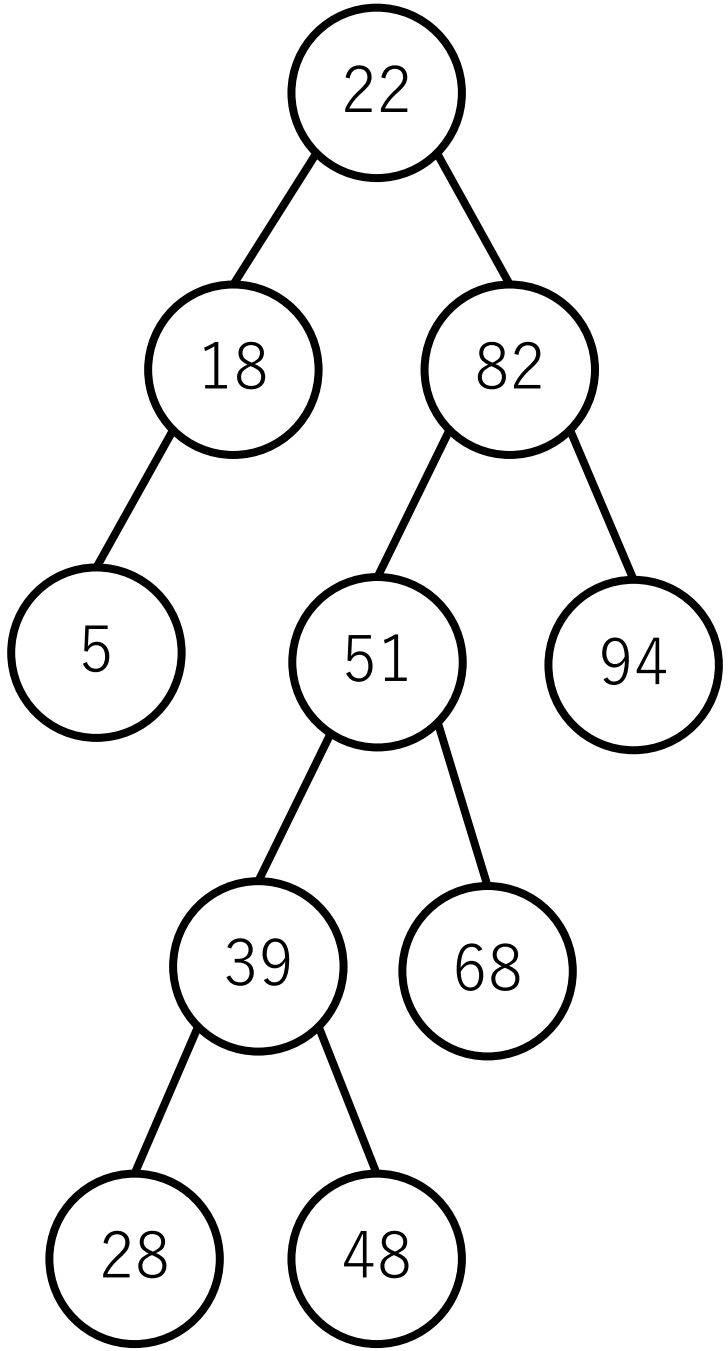
削除も同様に考えて$O(\log n)$.



二分木を使った探索の計算量

探索：

木が「普通の形」であれば，$O(\log n)$回子ノード辿れば結果がでると期待できるので，$O(\log n)$.



二分木を使った探索の計算量

木を1から作る：

二分探索木の要素の数がiの時，次の要素の挿入に$O(\log(i))$回の比較が必要．よって木全部を構成するためには，

$$\sum_{i=1}^{n} \log(i) \to O(\log(n!))$$

大雑把に$O(\log n)$がn回あると思って，$O(n \log n)$でもよい．

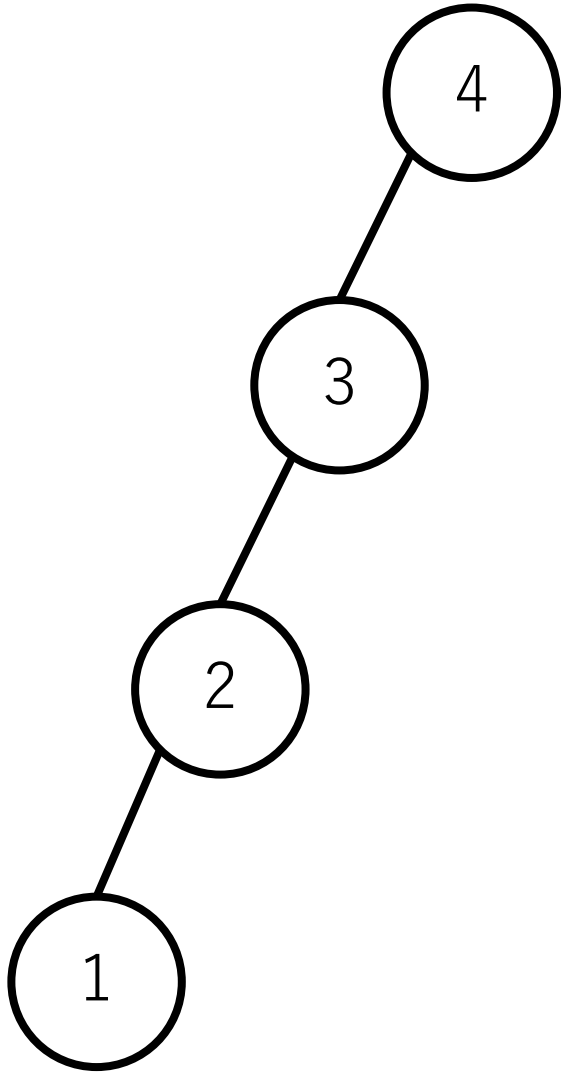
$$O(n \log n) = O(\log(n^n)) > O(\log(n!))$$

最悪の二分木

一方に偏ってしまう.

こうなってしまうと挿入・探索・削除
すべて$O(n)$.

線形探索，もしくは線形リスト上で
探索を行っているのと同じ.



何が問題？

先に挿入された要素ほど木の根に近い部分に配置される.

この順番を入れ替える方法がないため，配列の要素がほぼ整列されている状態である場合などには，非効率な木の形になってしまう.

何が問題？

もし要素が順当にランダムに並んでいる配列であれば，
何も工夫せずに二分探索木を作っても，その高さは平均的には$O(\log n)$になると期待できる．

このランダム性を導入できないか？

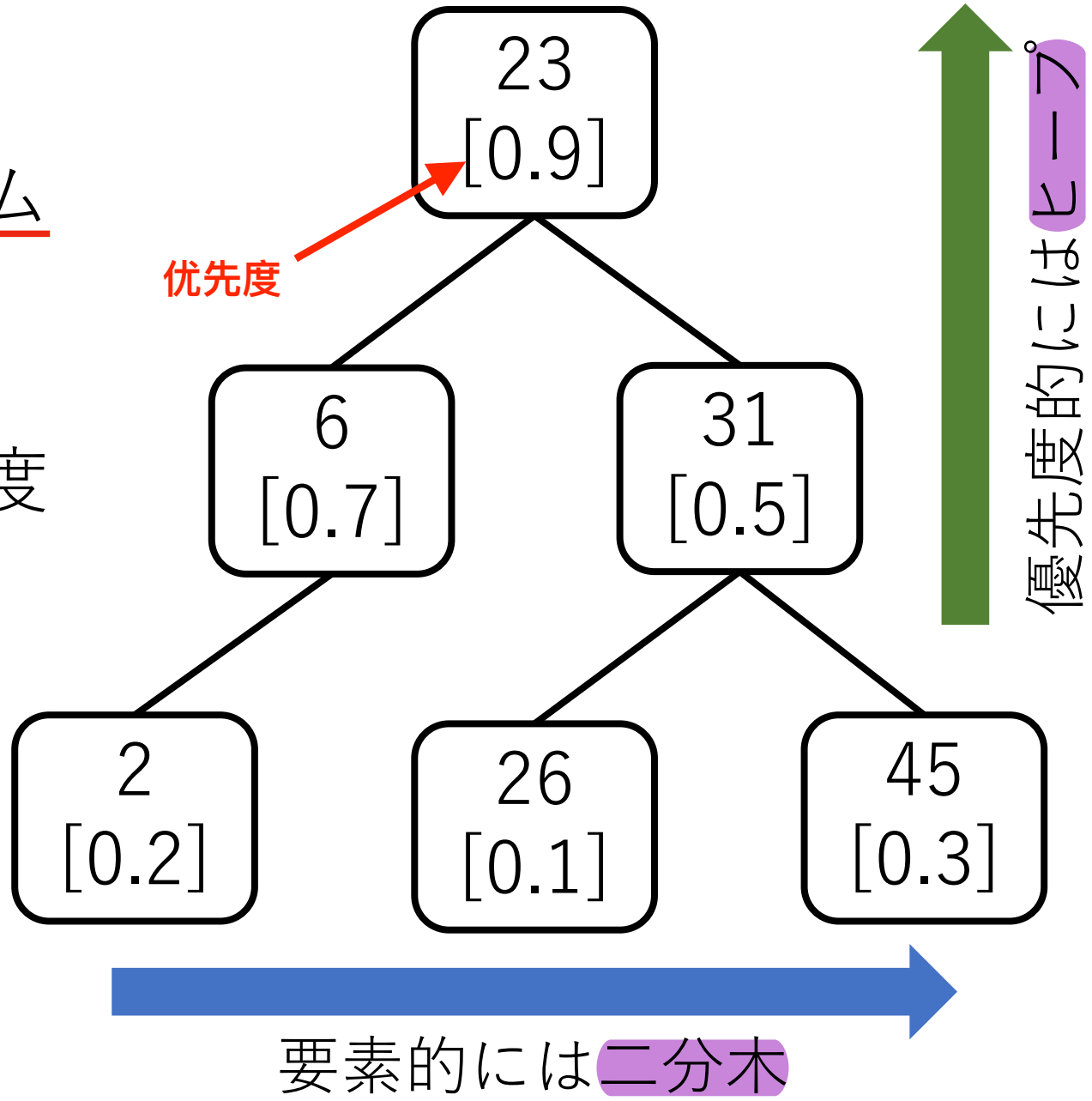
Treap（ツリープ）

结合了BST的有序性和heap的随机性

挿入の順番とは別にランダムに「優先度」を付与.

要素としては二分木，優先度としては二分ヒープになるような構造を作る.

優先度はランダムに付与されるので平衡に近くなる.



Treap（ツリープ）

追加：

まず要素に対して通常の二分探索木のように追加.

そして，回転（このあとすぐ出てきます）を
使ってヒープの制約を満たすように修正.

探索：

通常の二分探索木に同じ.

もう1つの方策

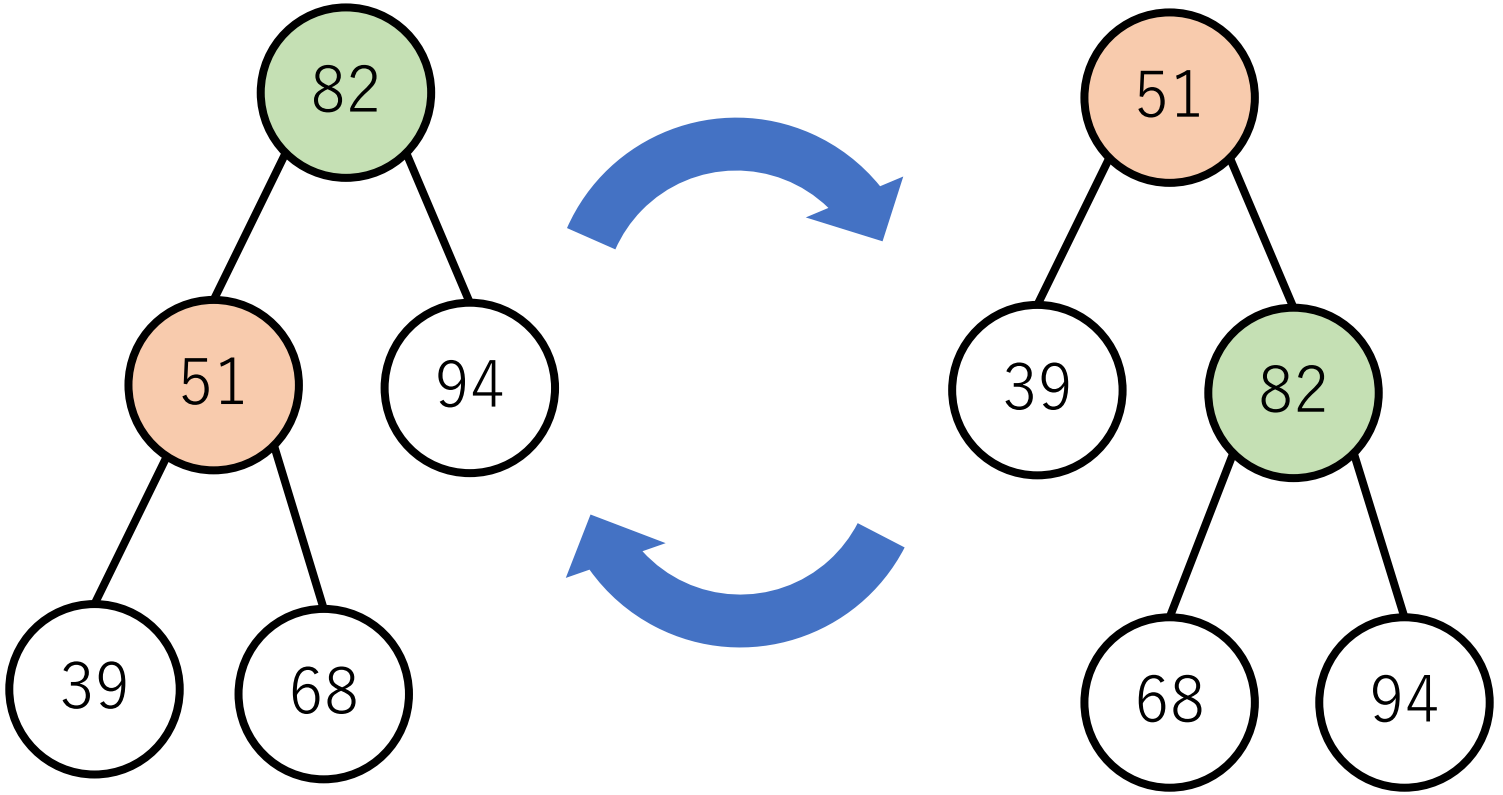
一方に偏った木になりそうな場合，修正できないか？

使いにくくなったお家をリフォームするイメージ.

ただし，修正が$O(\log n)$で終わらないと意味ない.
（じゃないと$O(n)$の線形探索の方がまし）.

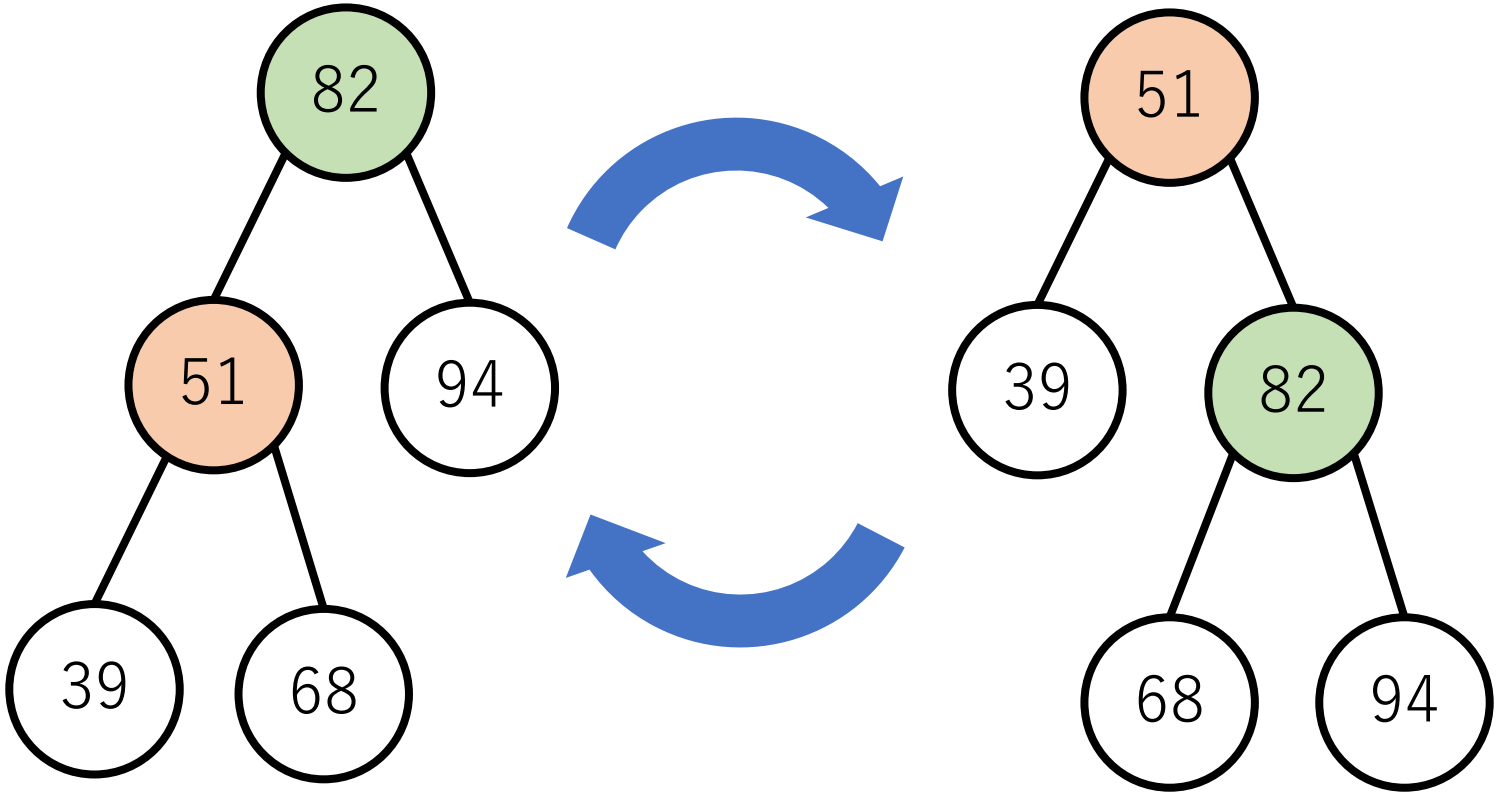
木の回転

要素の順序を崩さずに（[左の子] < [親] < [右の子]の大小関係を崩さずに），あるノードを上にあげる．これにより全体の高さの調整を行う．



木の回転

木の回転にはポインタの付け替えが必要．ただし，これは定数回で終わる．（下の場合には，51と82の子ノードの情報の更新が必要）．



平衡木

木の形のバランスが取れている木.

AVL木，B木，赤黒木，スプレー木などなど.

回転や多分木化，ノードに特別なラベルなどを導入してバランスを取ることを試みる.

実装はそこそこ大変なので，紹介だけ（興味ある人はぜひご自身で調べてください）.

AVL木（Adelson-Velskii and Landis’ tree）

平衡二分木の一種.

ノードの挿入が行われるとき，必要に応じて1回，もしくは2回の回転を行い，「**全ての部分木の左右の高さの差が1以下**」になるようする.

AVL木（Adelson-Velskii and Landis’ tree）

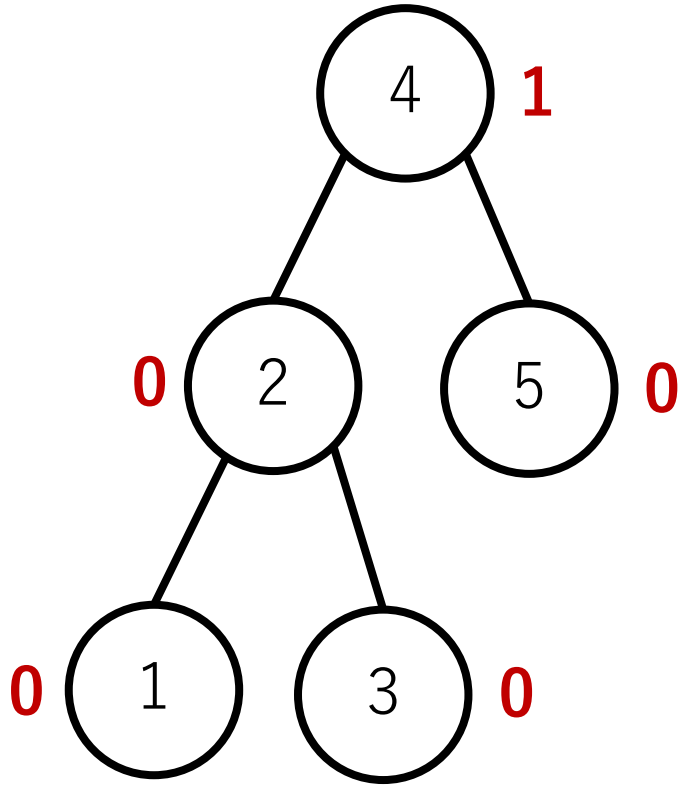
balance factorを各ノードに定義し，この値を見てどんな回転をするかを決める．

[balance factor] =
[**左の部分木の高さ**] - [**右の部分木の高さ**]

AVL木の例

balance factorが各ノードに付与されている．このbalance factorの絶対値が2以上になったら回転が必要な合図.

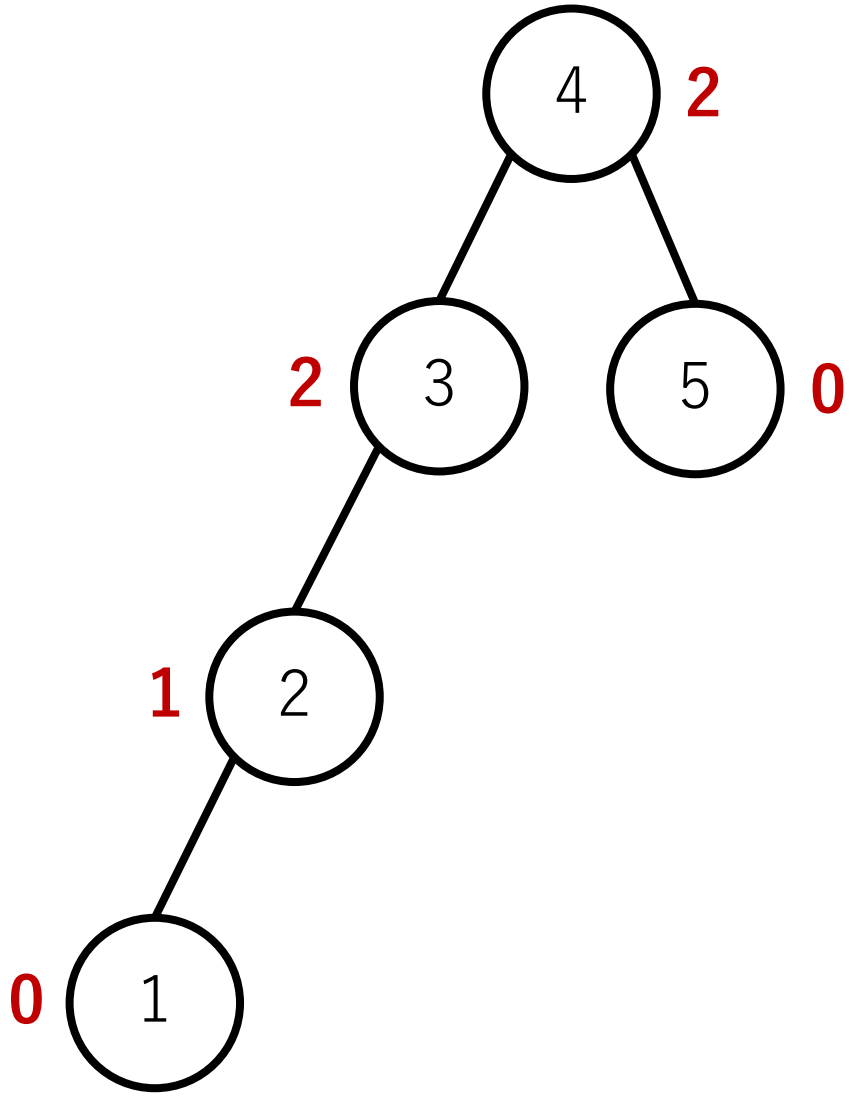
これは回転が必要ない．（全ての部分木の左右の高さの差が高々1）



AVL木の例

balance factorが各ノードに付与されている．このbalance factorの絶対値が2以上になったら回転が必要な合図.

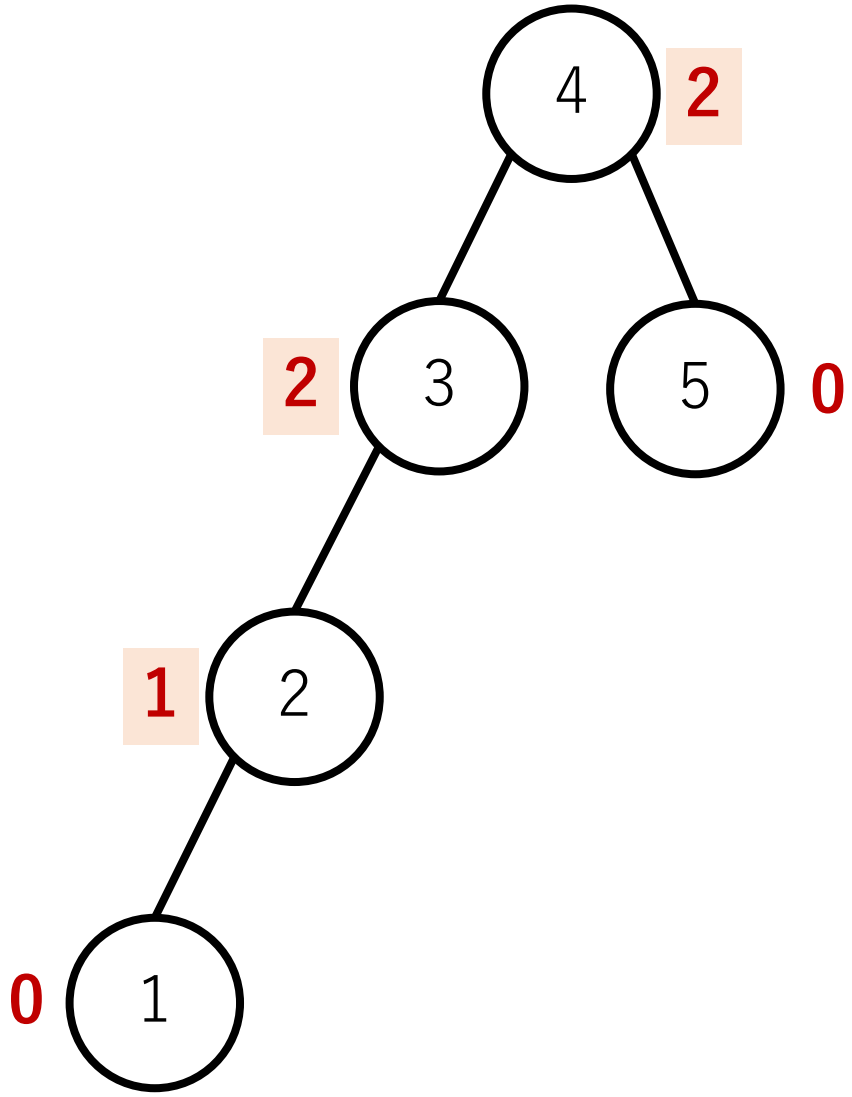
この木の場合は根ノードの下の左右の部分木の高さの差が2になっている.



AVL木の例

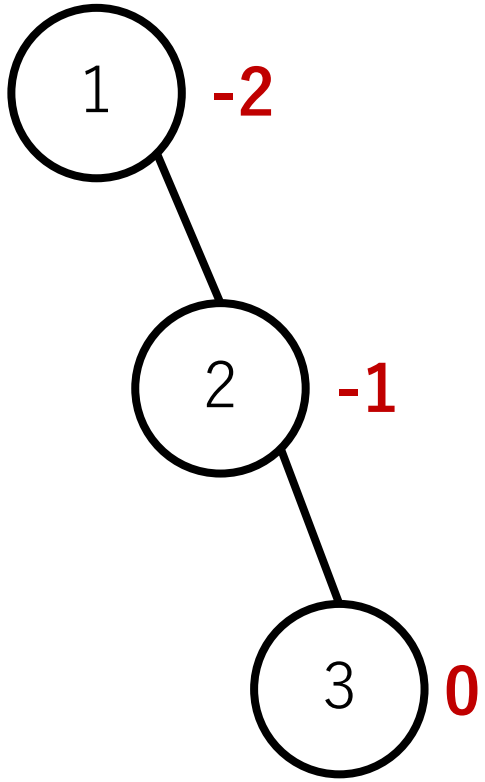
balance factorが各ノードに付与されている．このbalance factorの絶対値が2以上になったら回転が必要な合図.

よって，この場合は何かしらの回転を施したい.



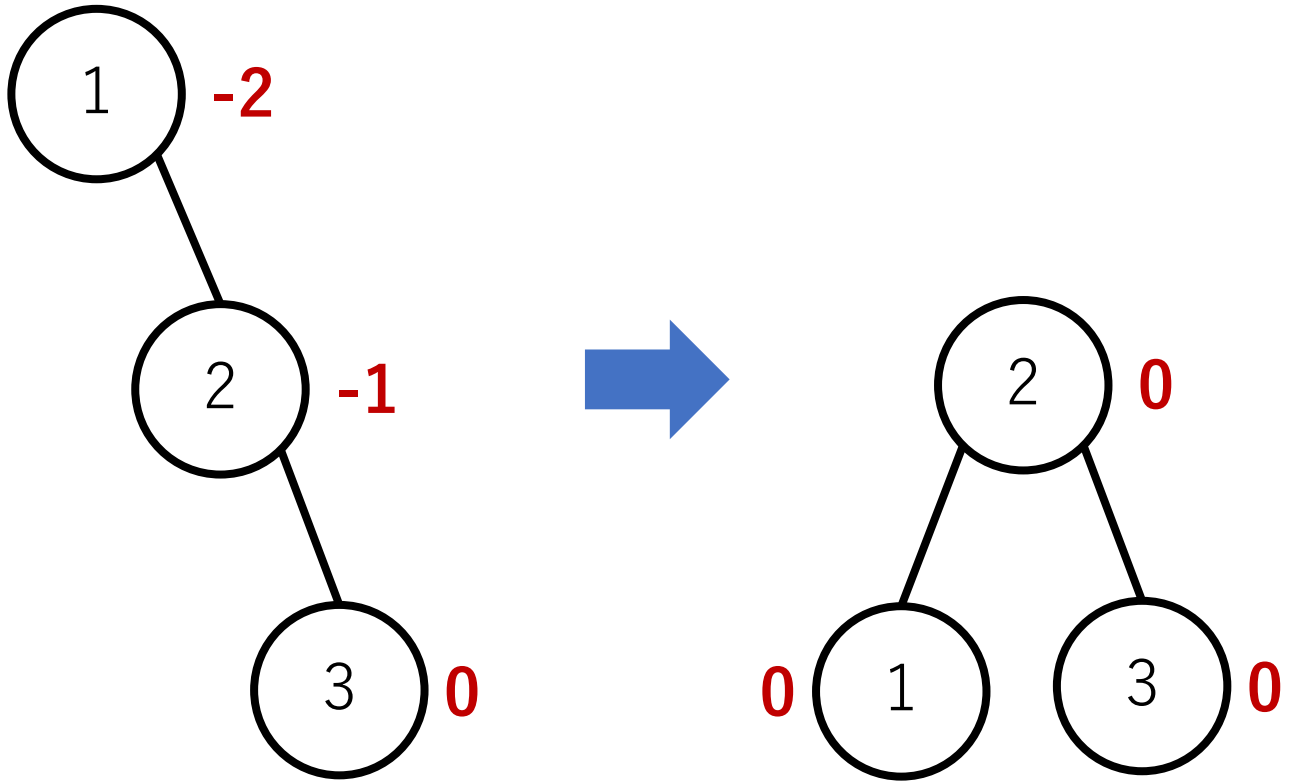
AVL木における回転

Single left rotation：balance factorが[-2, -1]という組み合わせ



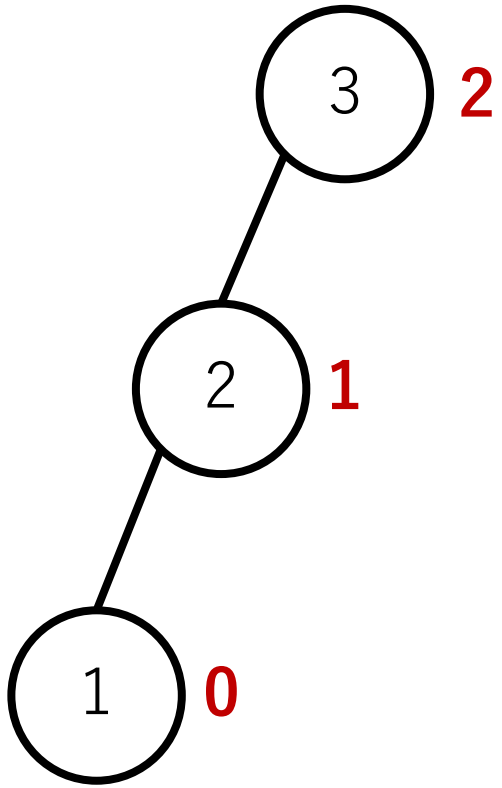
AVL木における回転

Single left rotation：balance factorが[-2, -1]という組み合わせ



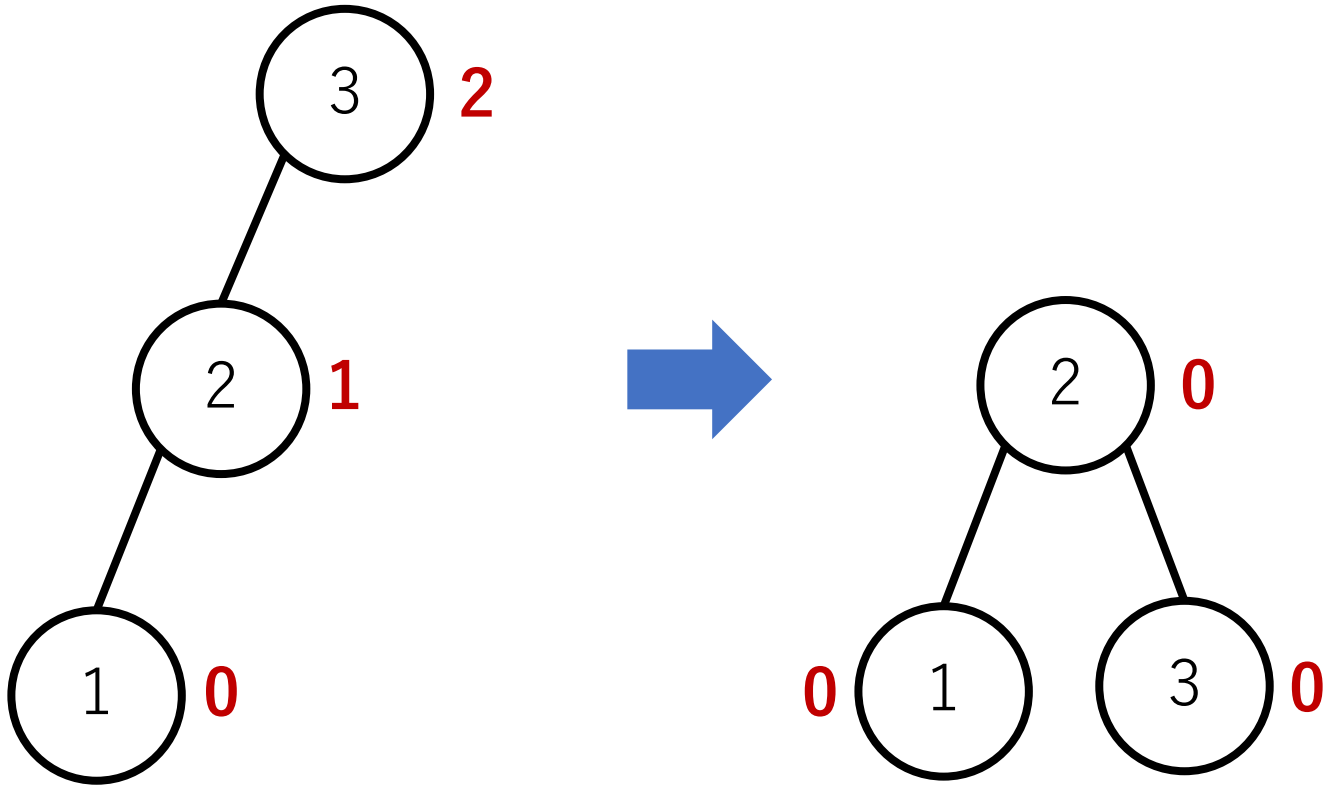
AVL木における回転

Single right rotation：balance factorが[2, 1]という組み合わせ



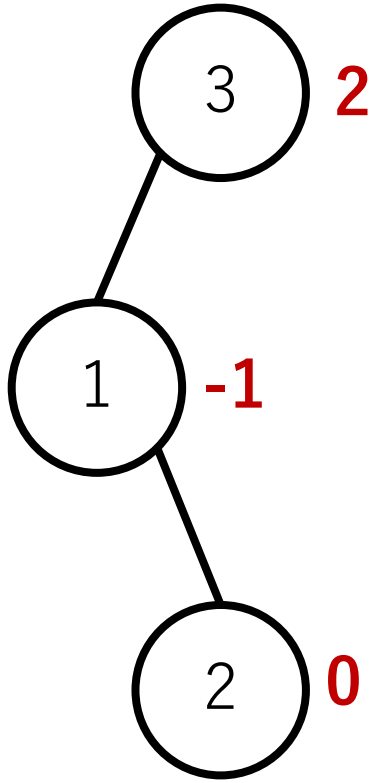
AVL木における回転

Single right rotation：balance factorが[2, 1]という組み合わせ



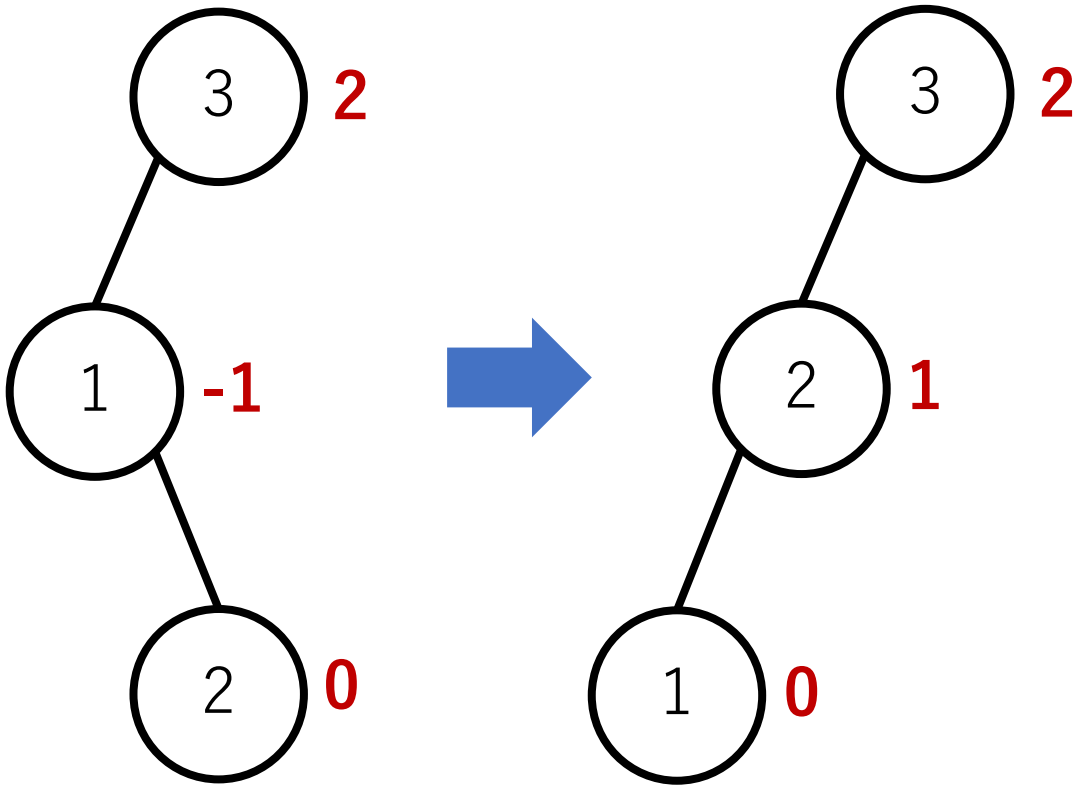
AVL木における回転

Left right rotation：balance factorが[2, -1]という組み合わせ



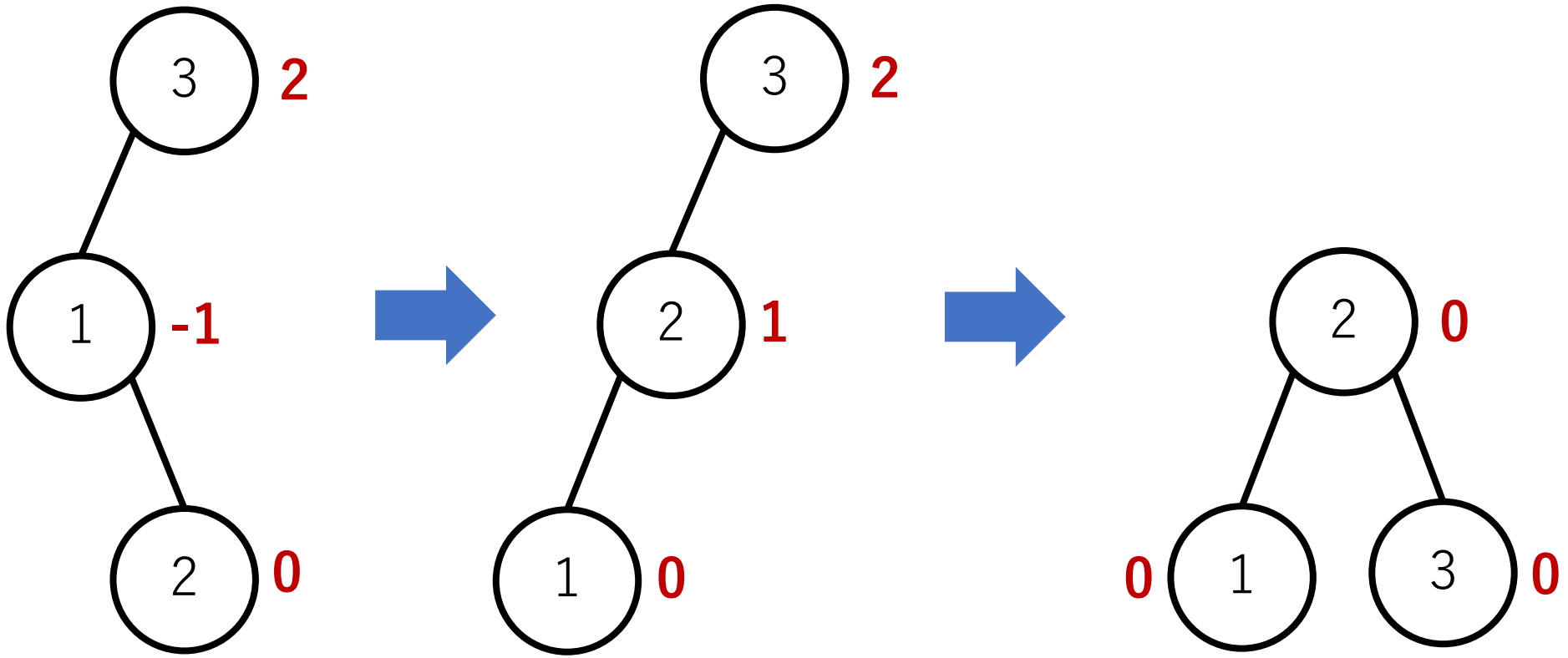
AVL木における回転

Left right rotation：balance factorが[2, -1]という組み合わせ



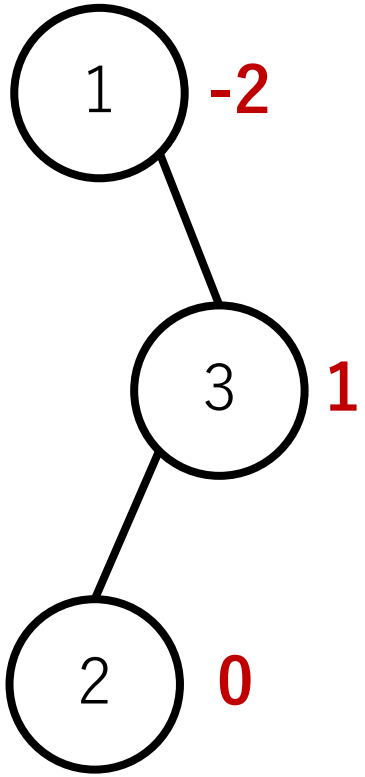
AVL木における回転

Left right rotation：balance factorが[2, -1]という組み合わせ



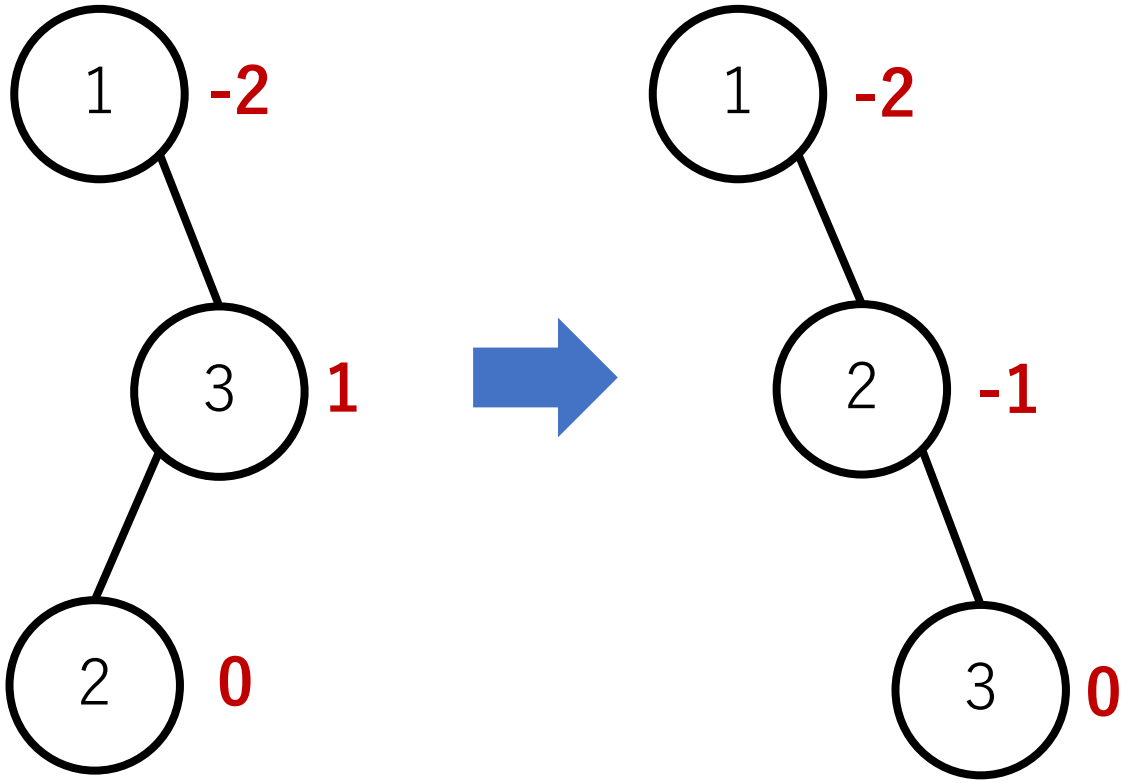
AVL木における回転

Right left rotation：balance factorが[-2, 1]という組み合わせ



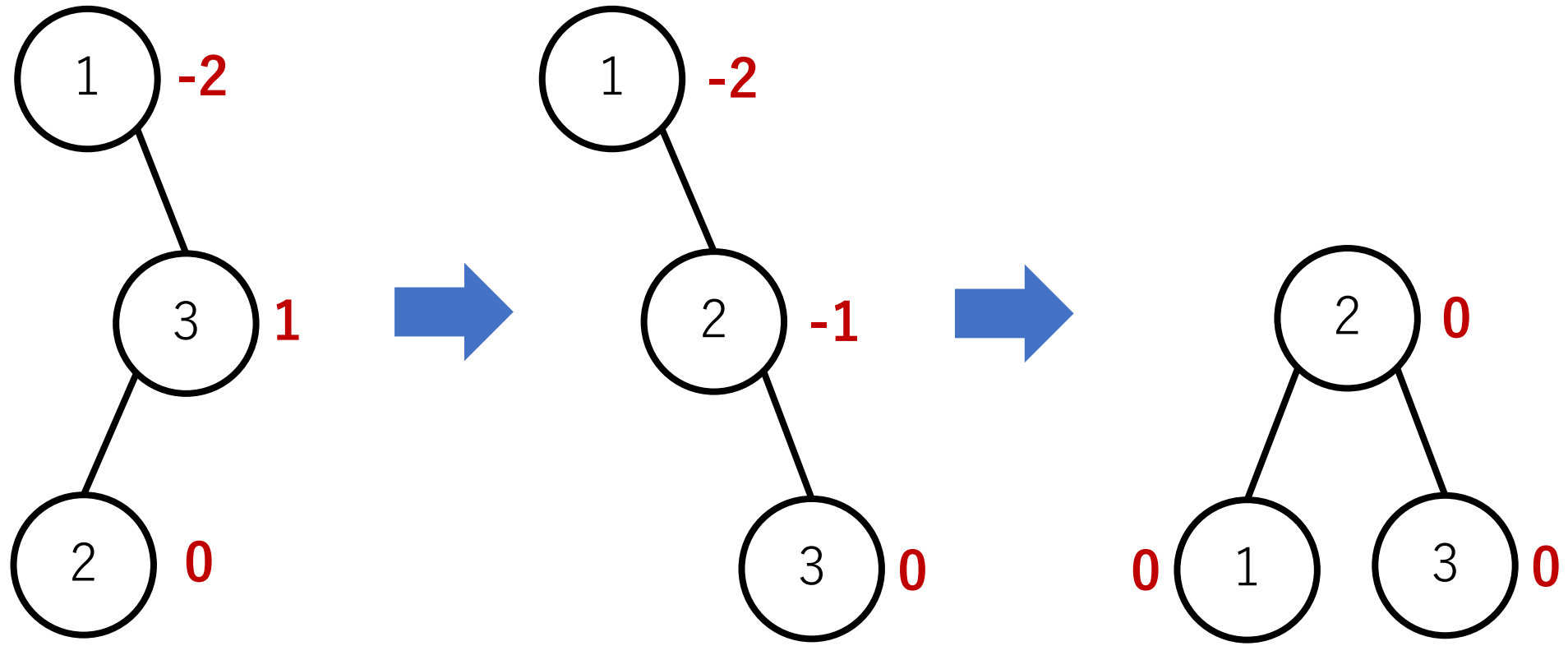
AVL木における回転

Right left rotation：balance factorが[-2, 1]という組み合わせ



AVL木における回転

Right left rotation：balance factorが[-2, 1]という組み合わせ



バランスファクターの更新

挿入した要素はとりあえず葉ノードになる.

その後，その葉ノードに到達するまでの根ノードからの経路上において，バランスが取れているかを確認する.

そのために，追加で新しくできた葉ノードから順にバランスファクターを更新する.

バランスファクターの更新

最悪の場合は根ノードまで辿る必要があるが，根ノードに到達する前までにどこかで回転をしたり（すなわちその部分木の高さを-1する），バランスファクターが0のノードにぶつかれば（挿入したことでその部分木の左右バランスが完全に取れた），バランスのチェックを終了することが出来る．

したがって，最悪でも木の高さ回のチェックと更新で済み，$O(\log n)$となる．また，チェックと更新は葉ノードから辿ることのできる部分木のみで行われるので，$O(\log n)$で収まる．

B木

多分木化を取り入れてバランス化をはかる.

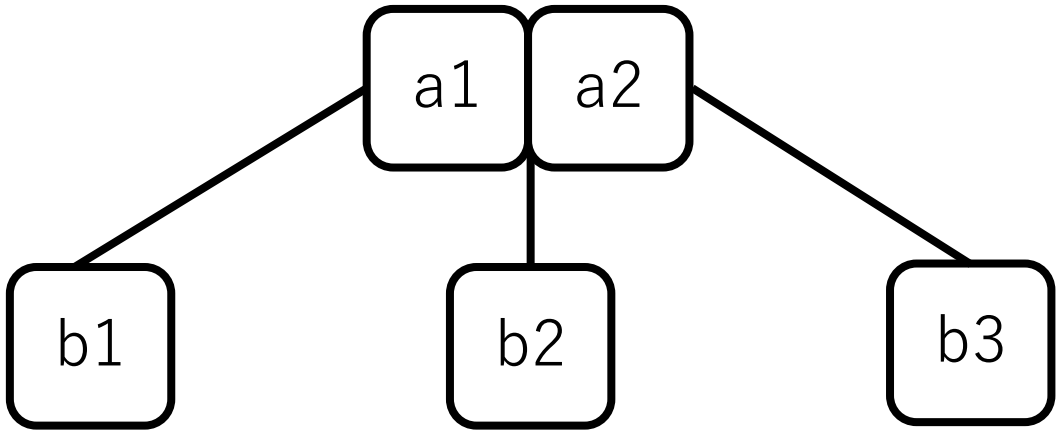
各ノードでの子ノードへの枝の数を最大mまで許す.
これをm次のB木（オーダーmのB木）という.

3次のB木：2-3木, 4次のB木：2-3-4木, とも呼ぶ.

2-3木のノード

1つのノードには最大で2つまで値を格納することができる.

ただし，探索ができるために，$b1 < a1 < b2 < a2 < b3$ という制約を守る必要がある.



2-3木におけるノードの追加

根ノードから探索し，新しい要素を追加するべき葉ノードを特定する．

そのノードが1つしか値を保持していない場合は，そこに追加する．

2-3木におけるノードの追加

追加したいノードがすでに2つの値を保持している場合，新しく加える値を合わせた3つの値のうち，真ん中の値を親ノードに送り，残りを2分割する．

送った先の親ノードにもスペースがない場合は，さらに親ノードに送る．

2-3木の例

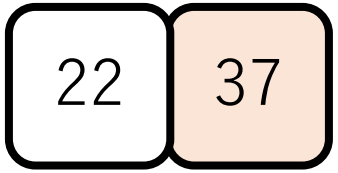
例：22, 37, 17, 9, 45, 34, 18

2-3木の例

例：22, 37, 17, 9, 45, 34, 18

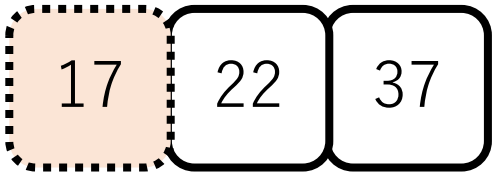
2-3木の例

例：22, 37, 17, 9, 45, 34, 18



2-3木の例

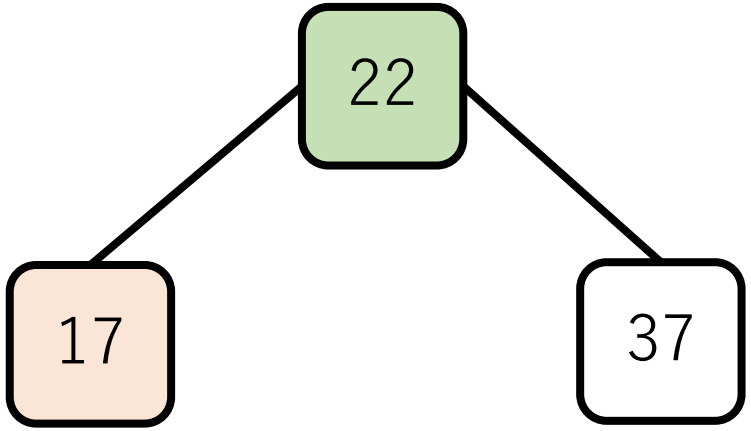
例：22, 37, 17, 9, 45, 34, 18



ここには入れない．

2-3木の例

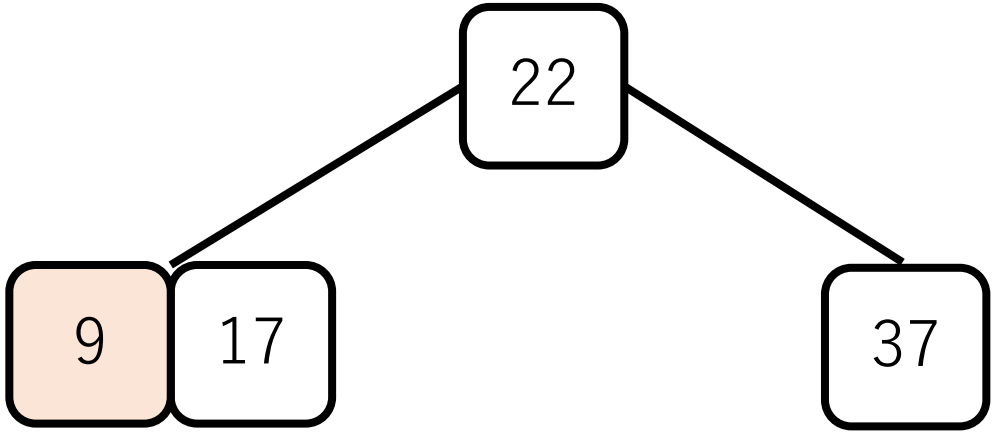
例：22, 37, 17, 9, 45, 34, 18



中央の値を親ノードにして，
残りの2つを分割する．

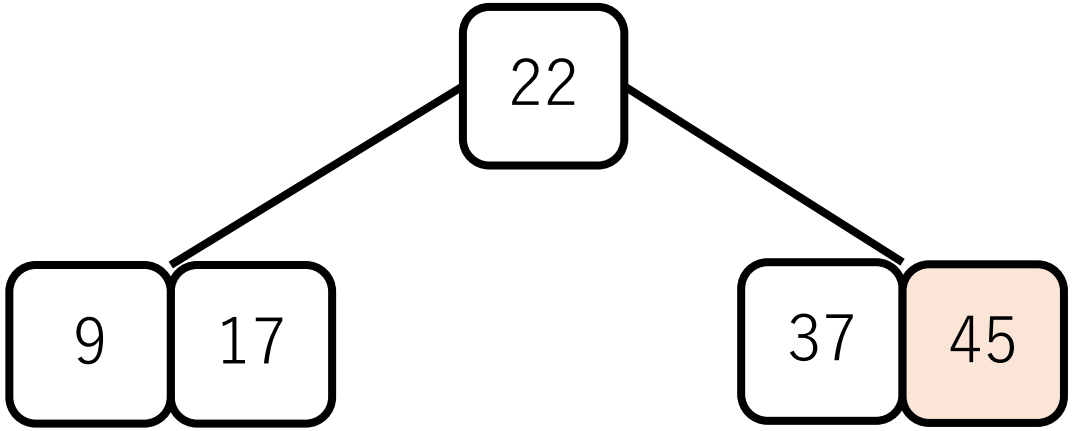
2-3木の例

例：22, 37, 17, 9, 45, 34, 18



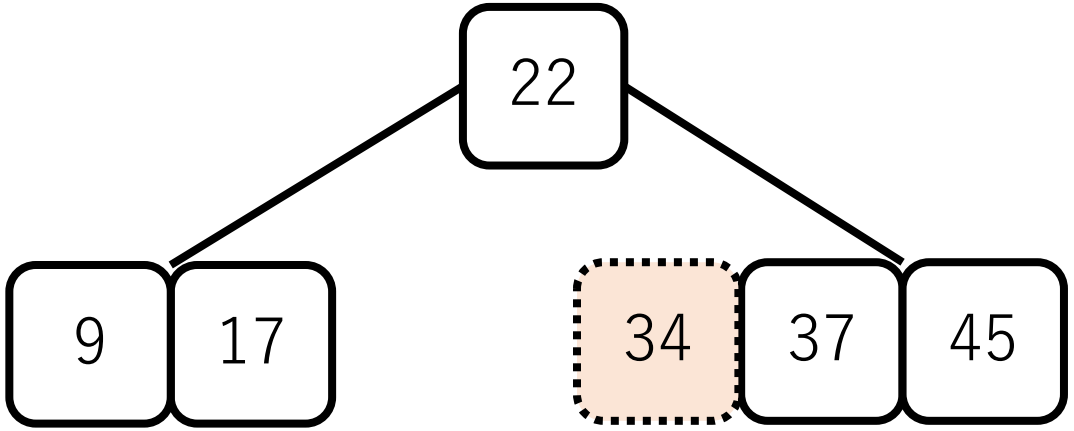
2-3木の例

例：22, 37, 17, 9, 45, 34, 18



2-3木の例

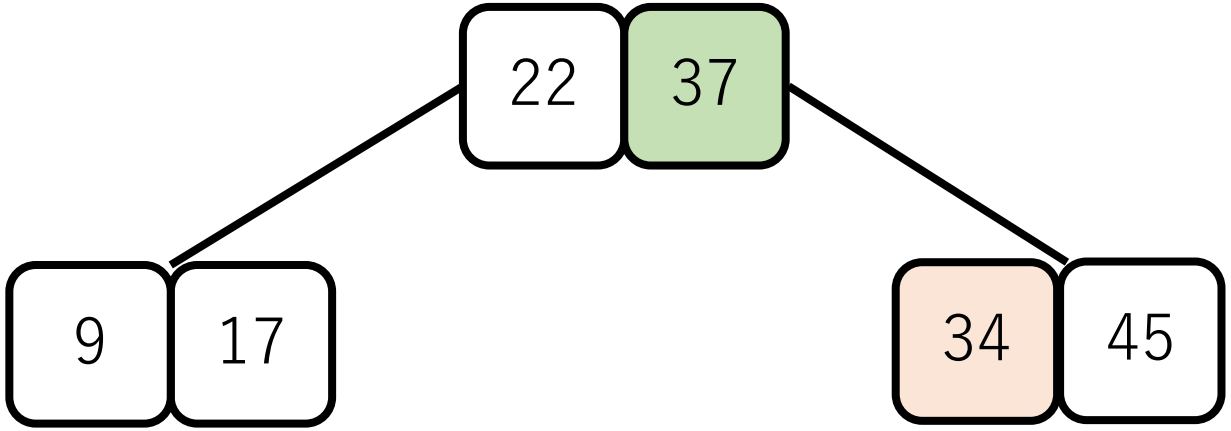
例：22, 37, 17, 9, 45, 34, 18



ここには入れない.

2-3木の例

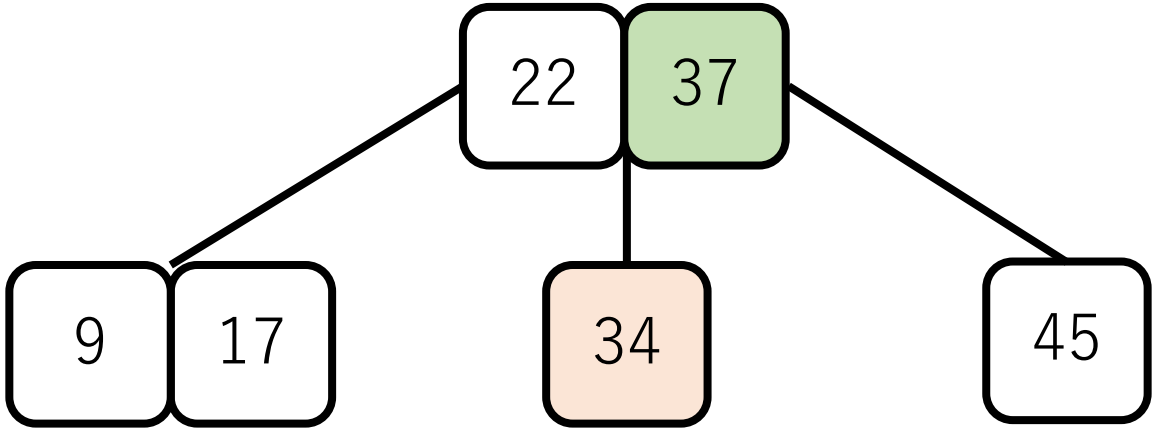
例：22, 37, 17, 9, 45, 34, 18



中央の値を親ノードに送る.
ただし，このままでは34は
制約を満たさない.

2-3木の例

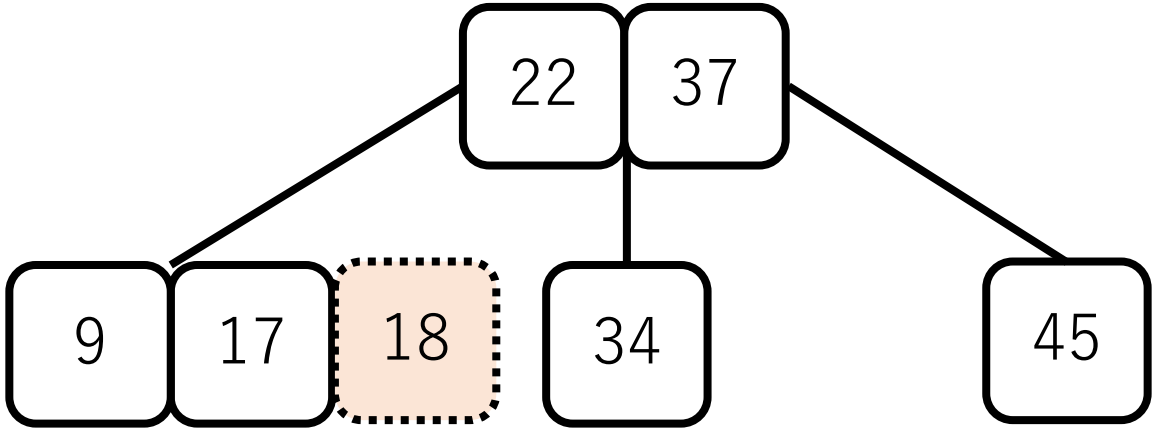
例：22, 37, 17, 9, 45, 34, 18



そこで，34と45を分割する．

2-3木の例

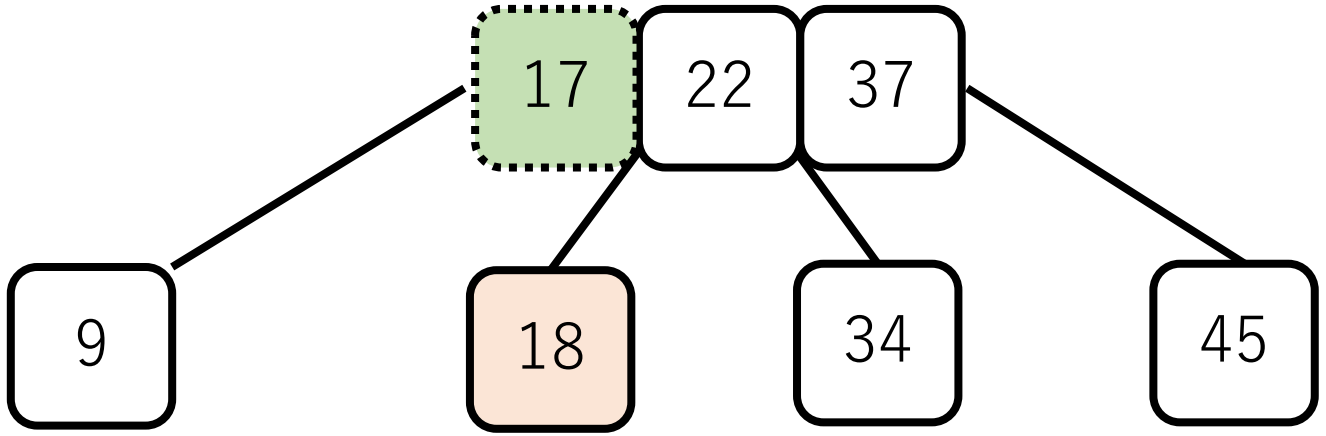
例：22, 37, 17, 9, 45, 34, 18



ここにはそのままでは入れないので,
真ん中の値17を親ノードに送り,
9と18は分割する.

2-3木の例

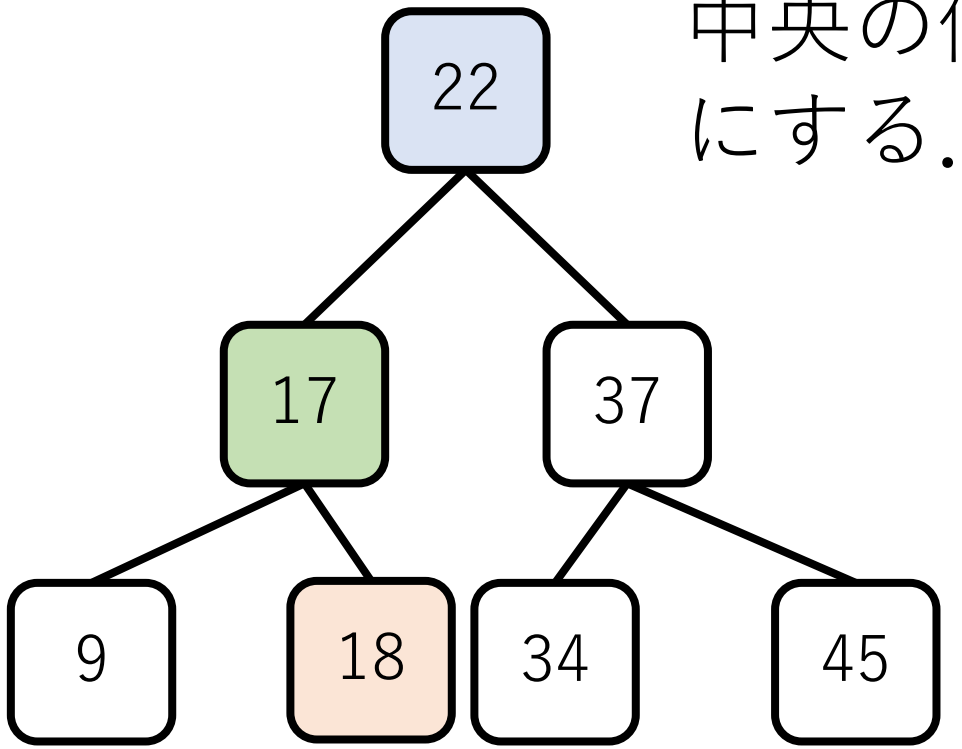
例：22, 37, 17, 9, 45, 34, 18



親ノードもめいっぱいなので入れない.
よって，中央の値22を更に上に送る.

2-3木の例

例：22, 37, 17, 9, 45, 34, 18



中央の値を親ノードにする.

B木の高さ

直感的には，中央にある値がどんどん根ノードの方に吸い寄せられるようになるため，バランスのとれた木になる．

よって，木の高さは $O(\log n)$ になると期待できる．

B木の計算量

最悪のケース：すでに埋まっている葉ノードに対して新しい要素をくっつけようとして，かつ，すでに埋まっている親ノードへの追加が順次行われ，根ノードまで行ってしまう．（先ほどの例で言えば18を追加するケース）

それでも木の高さは $O(\log n)$ なので，操作も $O(\log n)$.
ノードの付け替えは定数回のポインタ更新で可能.

したがって，最悪のケースでの要素追加でも $O(\log n)$.

発想の転換

今までの探索法はデータの数が増えれば探索時間も増大する.

データ構造をうまく使って大小比較の回数を減らしているものの，結局比較はしないといけないのが原因.

探索の前処理も必要（ソートなど）.

ハッシュ法

空間計算量を犠牲にして，時間計算量を稼ぐ．

探索時間は驚きの$O(1)$！！

ただし，空間計算量は$O(n)$．

データをメモリ上に全部置いておける場合などには有効．

ハッシュ

与えられた値に対してある変換を行う（例，剰余を計算）ことで，その値を格納する場所を決定する．

探索時にも同じ変換を利用して，その場所にある値と比較．

ハッシュの構築

例：[23, 36, 97, 4, 51, 11] で9の剰余でハッシュを作る.

ハッシュの構築

例）[23, 36, 97, 4, 51, 11] で9の剰余でハッシュを作る.

mod 9	0	1	2	3	4	5	6	7	8
	36		11		4	23	51	97	

ハッシュの構築

例）11を検索．mod 9を計算すると2.

mod 9	0	1	2	3	4	5	6	7	8
	36		11		4	23	51	97	

ハッシュの構築

例）12を検索．mod 9を計算すると3.

mod 9	0	1	2	3	4	5	6	7	8
	36		11		4	23	51	97	

空いているということは
そもそも存在しない．
→見つからなかったとして返す．

ハッシュの問題点：衝突

例）[23, 36, 97, 4, 51, 11, 20] で9の剰余でハッシュを作る.

mod 9	0	1	2	3	4	5	6	7	8
	36		11		4	23	51	97	

すでに11がはいっているので,
20をそのままでは挿入できない.

チェイン法

連結リストなどで追加する.

mod 9	0	1	2	3	4	5	6	7	8
	36		11		4	23	51	97	

オープンアドレス法

計算し直して開いているところを探す.

mod 9	0	1	2	3	4	5	6	7	8
	36		11	20	4	23	51	97	

その次の格納位置に移動し,
空いていればそこに格納する.

オープンアドレス法

例）20を検索.

mod 9	0	1	2	3	4	5	6	7	8
	36		11	20	4	23	51	97	

まずはハッシュで変換した場所（この場合2）を探索．でも11なのでマッチしない.

オープンアドレス法

例）20を検索.

mod 9	0	1	2	3	4	5	6	7	8
	36		11	20	4	23	51	97	

1つずらした場所をチェック.
この場合，ここで見つかる.

オープンアドレス法：挿入

もし空いていなかったら，更に次の格納場所へ移動し，空いているかどうかを確認する．

ハッシュ表の末尾まで来たら，先頭に移動して同じ処理を繰り返す．

もしどこも空いていなければ，最初の場所まで戻ってくるので，その場合はエラーを返す．

オープンアドレス法：探索

ハッシュで変換した場所をチェック.

その場所が空いているなら,
　　値は存在しないとして返す.

その場所に所望の値が入っていたなら,
　　値が見つかったとして返す.

オープンアドレス法：探索

その場所に値はあるが，所望の値でないならば，

オープンアドレス法で別の場所に格納されている
ことがありえる．

この場合は線形探索しないといけない．

線形探索を終了するタイミングは，3パターンある．

所望の値が見つかった．

途中で空のセルにぶつかった．

全てのセルをチェックしたが見つからなかった．

オープンアドレス法：探索

ただし削除の操作を許す場合には，話は少し厄介．

「ハッシュで見つけた場所に値がない」が，
「もともと値が存在していない」なのか，
「値はあったけど，すでに消してしまった」
かを区別しないといけない．

この場合，削除をしたというフラグを別に保存する，
削除した箇所に別の場所に保存されている値を移動
させてくるなどの実装が必要になる．

ハッシュの計算量

衝突がなければ，挿入，削除，探索全て$O(1)$.

ハッシュを一番最初に構築するのに必要な時間計算量は$O(n)$．空間計算量も同じ.

ハッシュの衝突

鳩の巣原理（ディリクレの箱入れ原理）

「a個のものをb個の箱に入れるとき，$a > b$ならば，
少なくとも1つの箱には2つ以上のものが入っている．」

「1つの箱に1つしか入れないとするならば，b個の
箱には最大b個しか入れることができない．」

ハッシュの衝突

a種類の値がある場合で，それをb種類のハッシュ値に変換し格納しようとする場合，$a > b$ならばどうやっても衝突（元々は違う値でも変換したハッシュ値が同じになってしまう）が起こる.

$\mathrm{mod}\ 9$の例ならば，先ほどの11と20とか.

普通は$a > b$となるようにハッシュ値が設計される.
取りうる値の空間がより小さくなるように設計される.

よって衝突に対してちゃんと処理できないと，ハッシュ法はうまく行かない.

辞書

キーと値を保持するデータ構造．pythonに限らず多くのプログラミング言語で実装されている．

```
dict = {}
dict['coffee_small'] = 200
dict['coffee_medium'] = 300
dict['coffee_large'] = 400

print('コーヒーSの値段：{}'.format(dict['coffee_small']))
```

辞書

キーは一意であれば数値でも文字列でも良い.

ハッシュなので検索は非常に早い.

簡易的な検索機能を備えたプロトタイプを作る上では,
とても便利.

まとめ

アルゴリズムで問題を解決

線形探索，二分探索

データ構造で問題を解決

二分探索木，ハッシュ

コードチャレンジ：基本課題#4-a [1点]

授業中に説明したハッシュをオープンアドレス法で実装してください.

値の削除はないものとします.

ハッシュテーブルの大きさはご自身で試行して,適宜調整してください.

Pythonの辞書等を使用することは認めません.

コードチャレンジ：基本課題#4-b [1点]

昇順にソートされている整数を格納している配列とキーが与えられた時，キーよりも大きい整数のうち，最も小さい整数の値を返す二分探索を行うコードを書いてください．

二分探索を自分で実装してください．bisect等を使用して実現することは認めません．

a = [i for i in array if i>K]; print(a[0])とかもダメです．😅

コードチャレンジ：Extra課題#4 [3点]

探索アルゴリズムを使用する問題.