Algorithms (2021 Summer)
#4：探索（サーチ）

矢谷 浩司

成績の扱いに関して

「未受験」となる条件を以下のように変更します．以下の条件に1つでも該当した場合，成績は「未受験」となります．

- コードチャレンジのうち，未着手のままの提出の回数が累積4回以上．
- Extra課題，レポート課題のどちらとも全て未提出．
- **期末試験を受験しなかった**

ただし，上記制約に引っかかる場合でも成績をつけてほしい場合，期末試験前日までに連絡をください．

前回出た質問

「「課題の模範解答を公開してほしい」というのは、課題を出された時に解答を公開するという形ではなく、課題が終わった次の時間の最初などに解答を見せてほしいということだと思います　私も解答が知れたら良いなと思うので、「写経になってしまう」以外に問題がなければ解答が知りたいと思います」

→基本課題のほとんどは，アルゴリズムの基礎的内容であり，他の教科書や講義，Webページ等でも非常に多く解説されていますので，もし授業での解説で不足であれば，それらをもとに自習していただく，という形で進めさせてください．

前回出た質問

「「課題の模範解答を公開してほしい」というのは、課題を出された時に解答を公開するという形ではなく、課題が終わった次の時間の最初などに解答を見せてほしいということだと思います　私も解答が知れたら良いなと思うので、「写経になってしまう」以外に問題がなければ解答が知りたいと思います」

→授業で聞くだけでなく，自分で試行錯誤し，調べてみて自分なりに咀嚼することが1番の勉強法だと思っております．答えを知る，ことではなく，自分なりに理解をするプロセスを大事にしてもらえると嬉しいです．😌

前回出た質問

「前回のextra課題で、締め切りを日曜24時だと思っていて間に合わなかったのですが、extra課題を何問か逃して、レポートを書かなくても単位取得上問題はないでしょうか」

→Extra課題に関しては未着手の制約はありません．
従って，提出がなければ0点という扱いで
進めていき，単位取得上は問題ありません．
（もちろん成績がつくかどうかは総合の点数によりますので，出さなくても必ず単位が来る，ということではありません．）

前回出た質問

「もし未受験となった場合は次からコードチャレンジを受けられなくなるのでしょうか？それとも、成績上だけの問題でしょうか？」

→配信自体は続ける予定ですが，場合によっては未受験該当者への配信を停止する可能性もあるので，受け取ったら少なくともアクセスし，着手しておいてもらえると良いのではないかと思います．

前回出た質問

「提出期限後もtrackなどでテストをできるようにして頂けませんでしょうか。それだと時間のある時期に取り組みやすいです。」

→現状のtrackではこれはサポートされていませんので，返却されたコードとテストケースを使って自習していただければと思います．

前回出た質問

「単位要件は、得点の絶対値が50点以上という認識であっていますか？」

→実際の成績は相対評価となりますので，50点あれば絶対に単位が来る，というわけでない点にはご注意ください．ただし，一定の努力が見られる方に関しては単位を出す方向で検討するようにします．

前回出た質問

「もし聴講という選択肢をとった場合、成績が関係なくなるので、extra課題が出されてから個別に教師の方々に質問することは可能でしょうか？」

→履修登録をしている人との公平性の観点，TAさんの負担の観点から，これは対応できないことをご理解ください．Extra課題はその次の週に配布される解答例を基に自分なりに復習していただければと思います．

前回出た質問

「講義開始時(13時)にコーディング課題が配信されるようにしていただくことは可能でしょうか」

→配信は授業パート終了直前に行います．あくまで授業の内容を受けて課題を行う，というスタイルですので，どうぞよろしくお願いします．

前回出た質問

「Pythonの基本を学ぶのにおすすめの教科書などありますか？」

→TAの鈴木さんより.

「Pythonプログラミング入門」という講義の教材が公開されていて、かなり分かりやすいと思います。 https://sites.google.com/view/ut-python/

→受講者の方より.

オライリーの本（PDFが無料でダウンロード可）
https://greenteapress.com/wp/think-python-2e/

前回出た質問

「今の質問に関して、アルゴリズムの原理や証明についてのおすすめの参考文献を教えてもらえませんか？」

→「アルゴリズムイントロダクション」が一番有名かなと思います．ただし高いので，まずは図書館等で見てみるのをお勧めします．もし図書館等になければ私が持ってきますので，DMください．

前回出た質問

「classを最初に明記しないと機能しないのでしょうか？」

→スライドではクラスとして実装していました．他の例でもクラスとして実装ことが多いと思います．それによりバッファを外から見えなくし，アクセス方法を定義したメソッドに限ることができるためです．

「バッファはclassの外部から参照できるものですか？」

→基本的には外部からは見えない形で実装することが多いと思います．（そうでなければキューやスタックである必然性も低いはず）．

前回出た質問

「キューの実装でリングバッファ以外の物はありますか？」

→TAの鈴木さんより.

例えばこの後紹介される線形リストで実装することもできます(その場合、先頭ノードと末尾ノードへのポインタを情報として持っておく必要があります)

「スタックは実装に特に必要な工夫はないということで大丈夫ですか？」

→TAの役山さんより.

queueにおけるリングバッファ、のような工夫は必要ありません

前回出た質問

「実用上、スタックとキューはPythonのリストが同等の機能を持っていると思ったのですが、これは正しいですか？」

→TAの役山さんより.

listをqueueやstackとして使うことは可能ですが、
別ものという認識を持っておいたほうがよいでしょう

https://qiita.com/hiroyuki_mrp/items/a300eb3ccce4cb31ec9e

前回出た質問

「pythonのようなポインタがない言語で線形リストは使いますか？（実際実装しようとした時、リストの入子構造で実装しなければならず、全く使いやすいものではなくなった記憶があります）」

→TAの役山さんより.

直接ポインタという表現ではないですが、似たものとして参照という概念があり、これを用いるとそれほど複雑にならずに実装することができます

http://bacspot.dip.jp/virtual_link/www/si.musashi-tech.ac.jp/www/Python_IntroProgramming/05/index-1d.html

前回出た質問

「次数・内部ノードとは何ですか」

→TAの鈴木さんより.

次数：子ノードの数
内部ノード：葉ノード以外のノード

なお，スライドの方は上記言葉を使わない表現に修正いたしました．失礼しました.

前回出た質問

「ヒープのメリットがわかりません。具体的にどんな使われかたをするのですか？」

→TAの役山さんより.

たくさんのデータがあって、追加・削除を行っていく場合に、そのデータの最大値または最小値を効率的に管理することができます

前回出た質問

「根から辿るのであればself.check_after_remove(self.last)ではなくself.check_after_remove(1)ではないですか？」

→ご指摘ありがとうございます．こちらも修正しておきました．失礼しました．

前回出た質問

「グラフ理論はこの授業では扱いませんか？」

→グラフそのものの話はあまり多くは扱いませんが，
代表的な話に関しては関連するアルゴリズムと
ともにご紹介する予定です．どちらかというと，
紹介したいアルゴリズムをもとに，それに関連する
グラフの話をする，という感じです．

前回出た質問

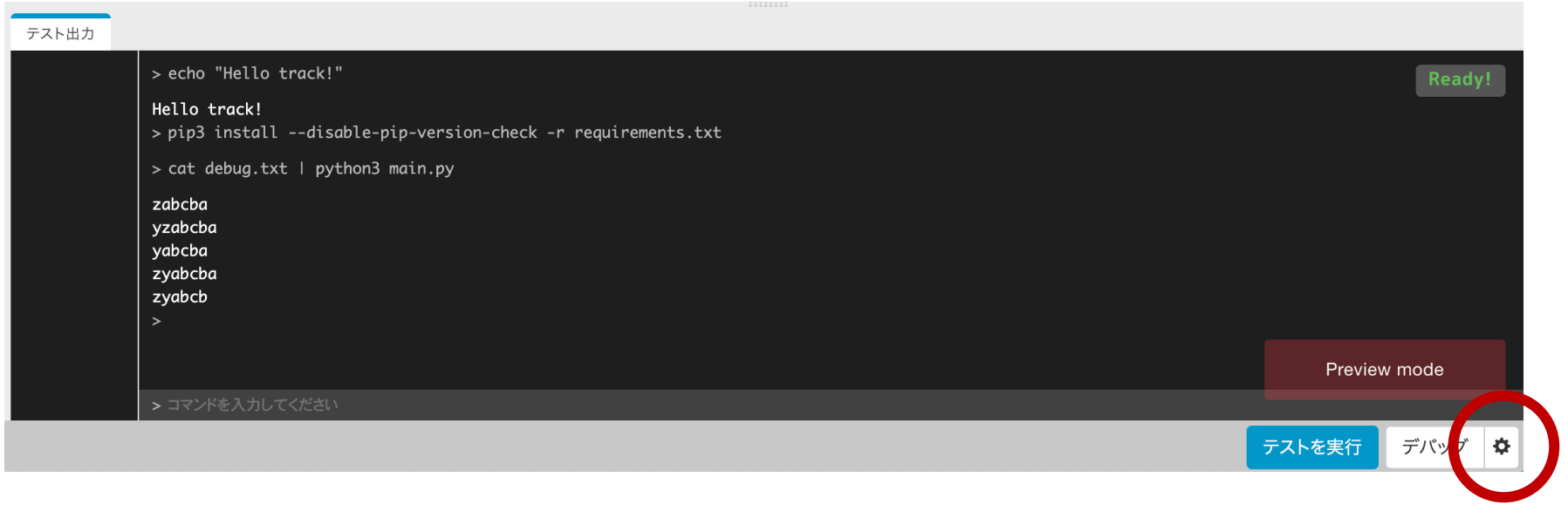
「課題をやっている途中でCommand terminated by signal 9と出たのですが、どのような内容のエラーなのでしょうか」

→多くの場合は，メモリ制限のエラー（こちらが指定しているメモリ量以上のメモリ消費が存在する）ときのエラーかと思います．不必要な変数のコピーなどがあると，この手のエラーになると思います．

前回出た質問

「標準エラー出力が全文見えずデバッグに悩んでいるのですが、全文表示させる方法はありますでしょうか？」

→テストケースを個別に動かすとよいです．画面右下の「デバッグ」横の歯車のボタンを押すと，個別のテストケースを選んで実行できます．



前回出た決意表明

「今日のAtCoderBeginnerContest199にチャレンジしてみようと思います」

→素晴らしい！😄 競技プログラミングは癖がある（パズル的要素が大きい問題もある）ので，慣れが結構必要ですが，ガンガン挑戦してみてください！

成績の扱いに関して

「未受験」となる条件を以下のように変更します．以下の条件に1つでも該当した場合，成績は「未受験」となります．

- コードチャレンジのうち，未着手のままの提出の回数が累積4回以上．
- Extra課題，レポート課題のどちらとも全て未提出．
- **期末試験を受験しなかった**

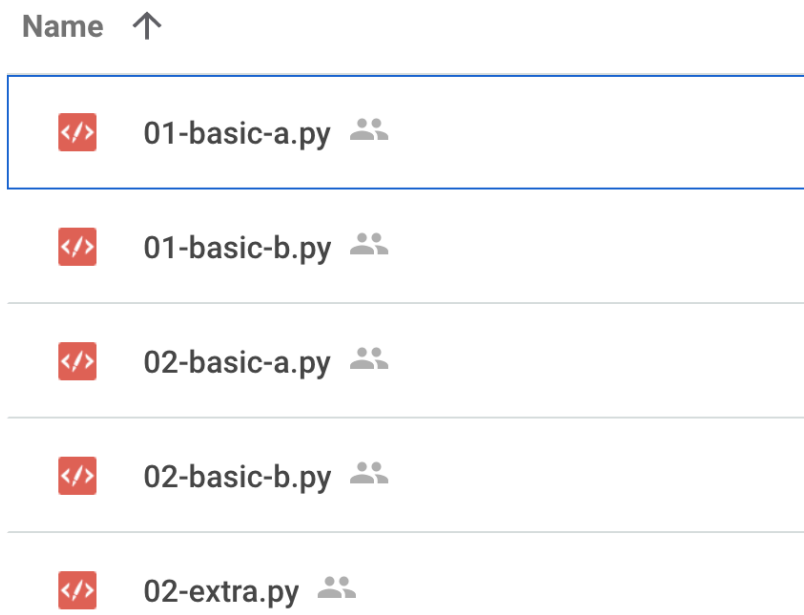
ただし，上記制約に引っかかる場合でも成績をつけてほしい場合，期末試験前日までに連絡をください．

提出されたコードの返却

先週末，コード返却の処理を行いましたので，ご確認ください．

皆さんのECCSアカウント1つ1つに共有フォルダを作成し，個別に招待をしてあります．（6桁の数字のフォルダ名になっています．）

ファイル名で中身が一目でわかるようにしてありますので，ご確認ください．



提出されたコードの返却

今後提出コードはすべてmain.pyの中に入れ込んでもらえる（別のファイルを作らず，すべてmain.pyの中に記述する）と嬉しいです.

返却用スクリプトはそれを前提に組み立てていますので，どうぞよろしくお願い致します.

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どちらも$O(n)$.

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？

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 = 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      # 見つかった
        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

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，$f(a,b,n) \geq K$を満たす最大の$n$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.

例えば，ある整数においてならば，以上の値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！

また，ならば，以下の値全てに対して
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，なぜ0？
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,

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

解法の方針

よって，あるKに対して，

$$K \leq \frac{R-K}{x-1} + \frac{B-K}{y-1}$$

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.

（ソートの種類によるが）これには一般的には
$O(n \log n)$かかる（6回目の授業で紹介予定）.

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.

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，左の子コードが存在する：

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，右の子コードが存在する：

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

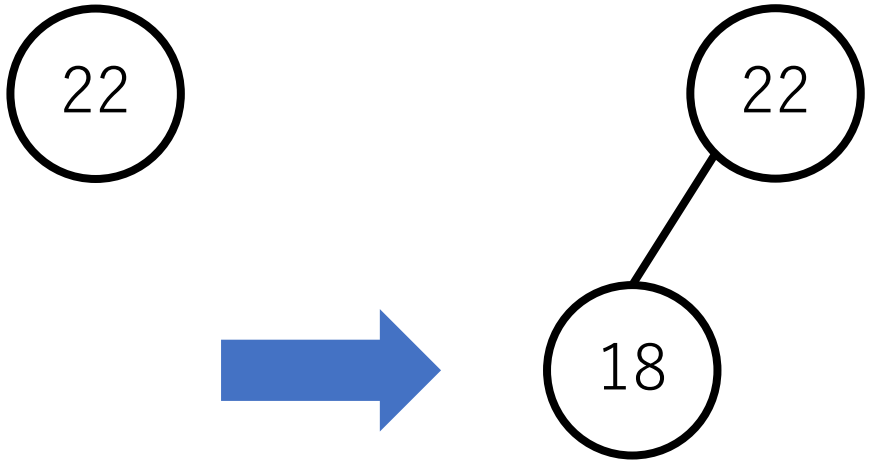
例：[22, 18, 5, 82, 51, 39, 48, 94, 68, 28]

二分木を作る

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

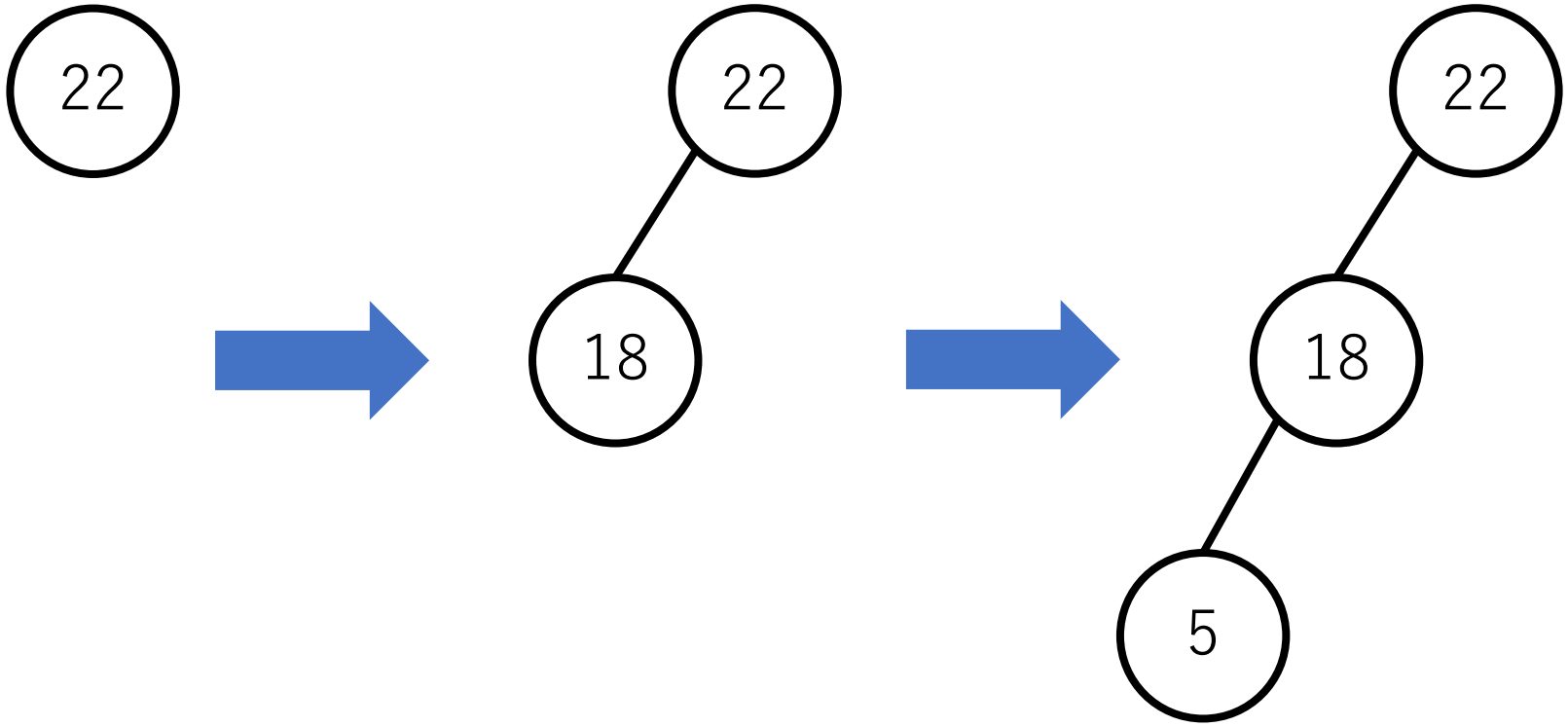
二分木を作る

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



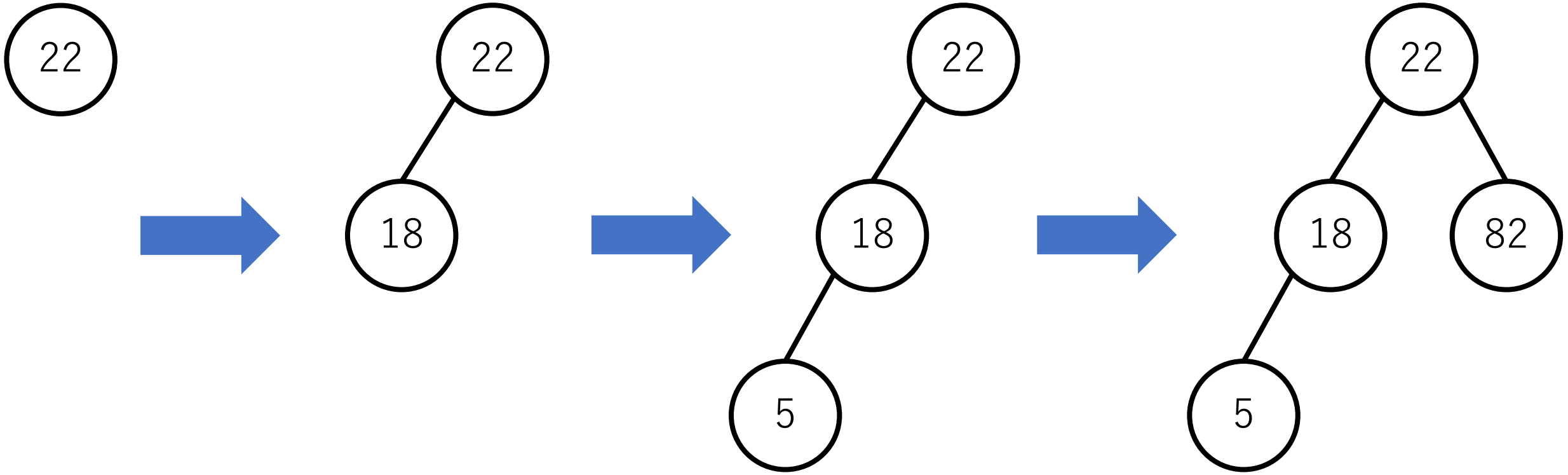
二分木を作る

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



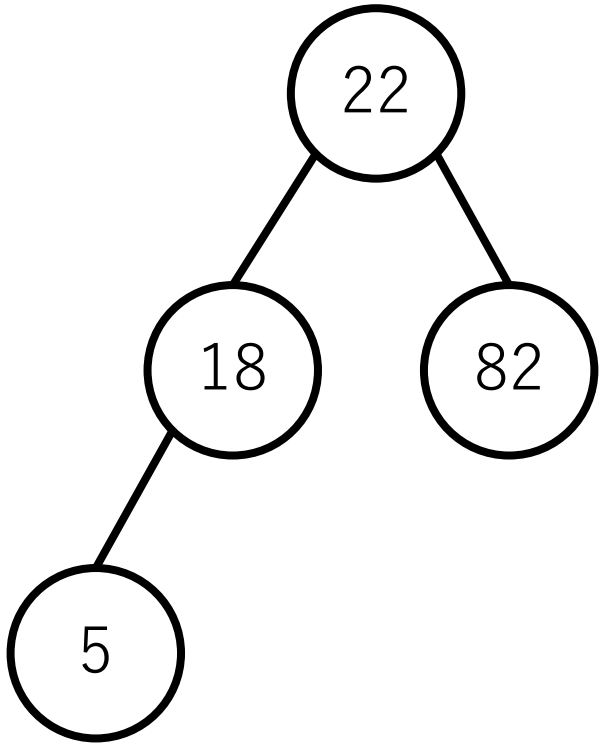
二分木を作る

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



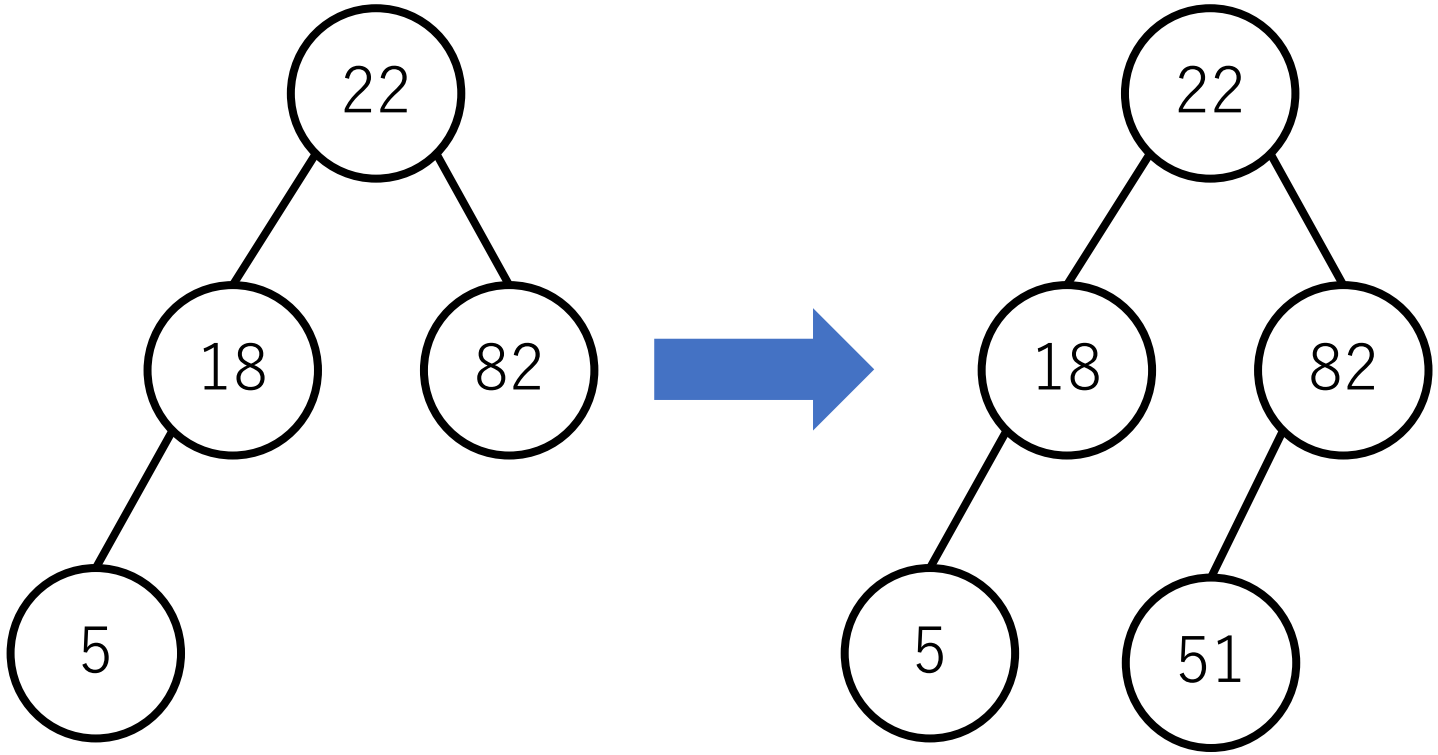
二分木を作る

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



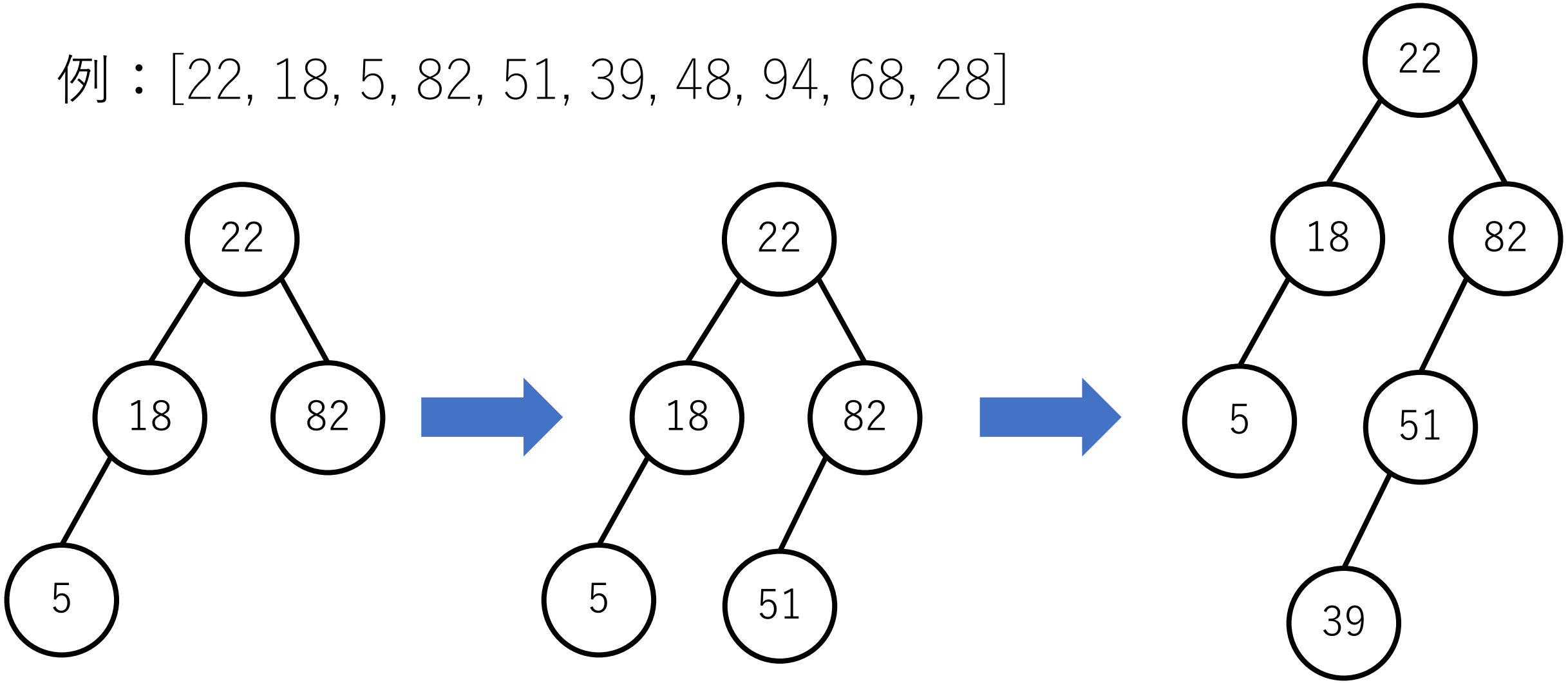
二分木を作る

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]

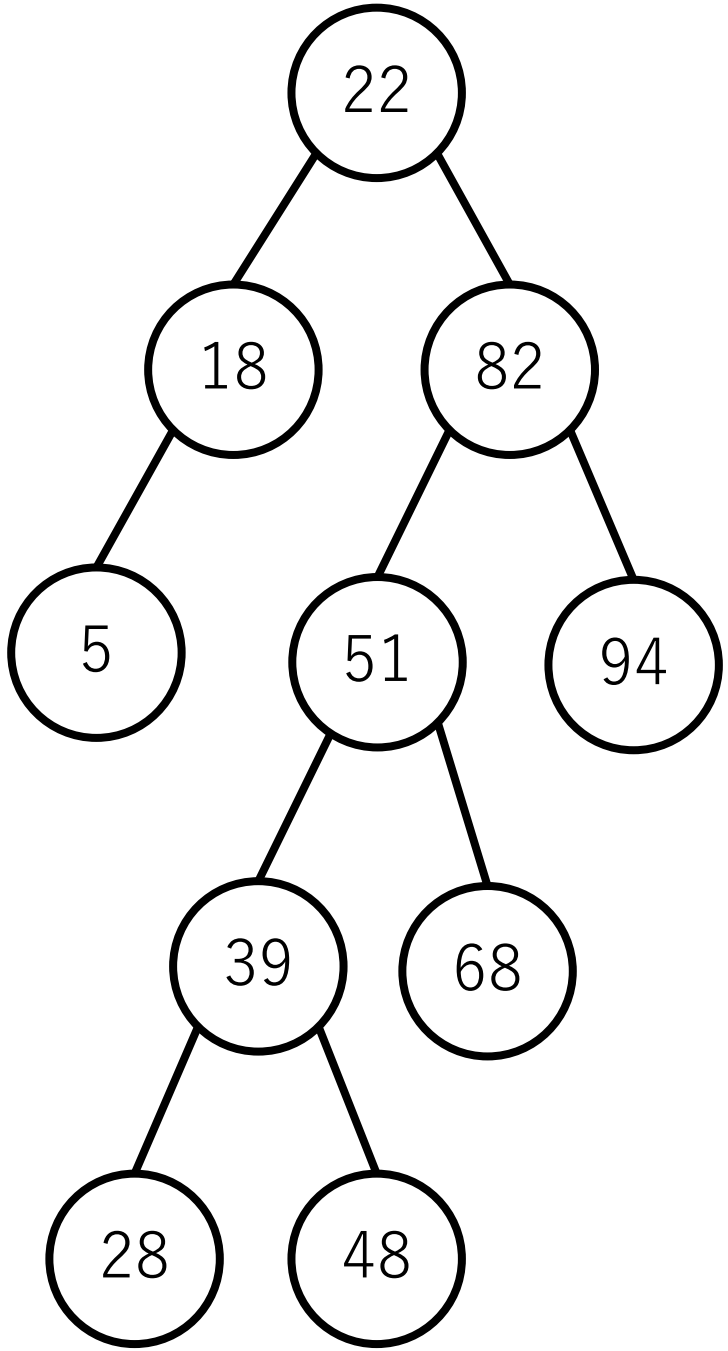


二分木を作る

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.

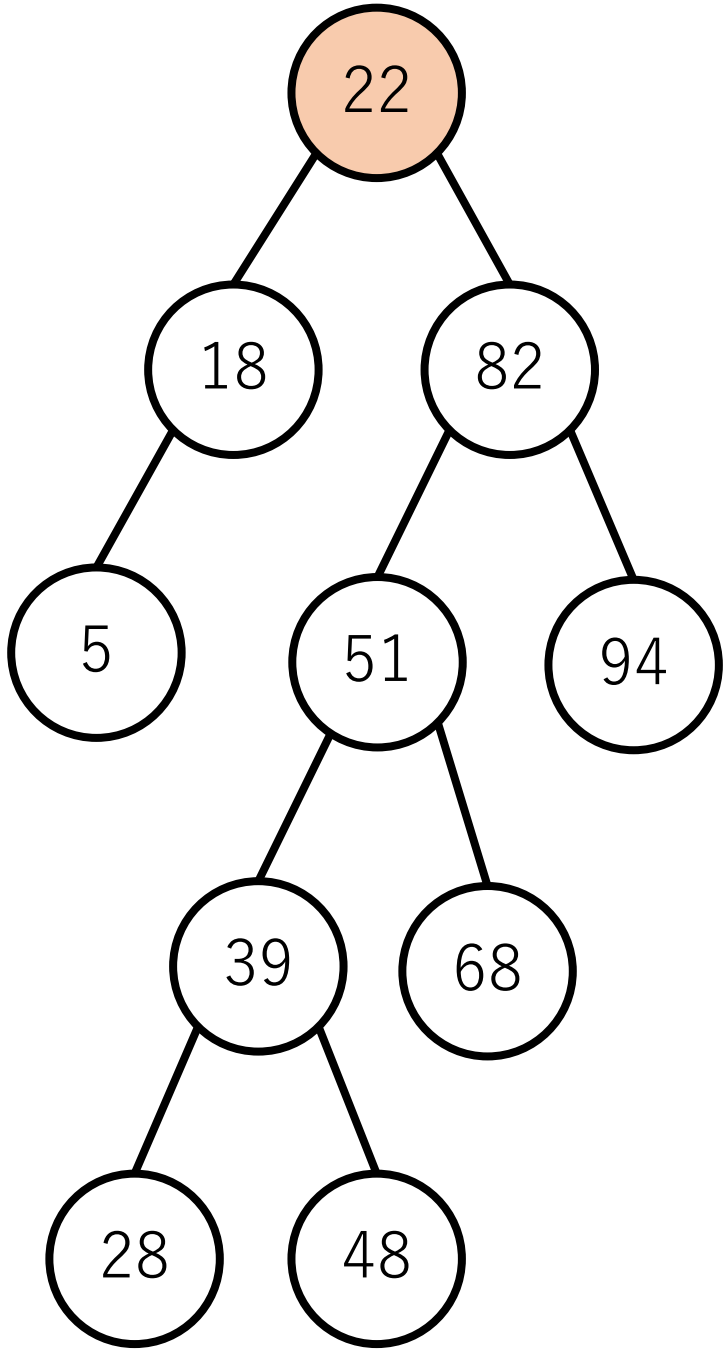
各ノードより小さな値は左側に，大きな値は右側に接続されるようになる.



二分木を使って探索

例：68を探す.

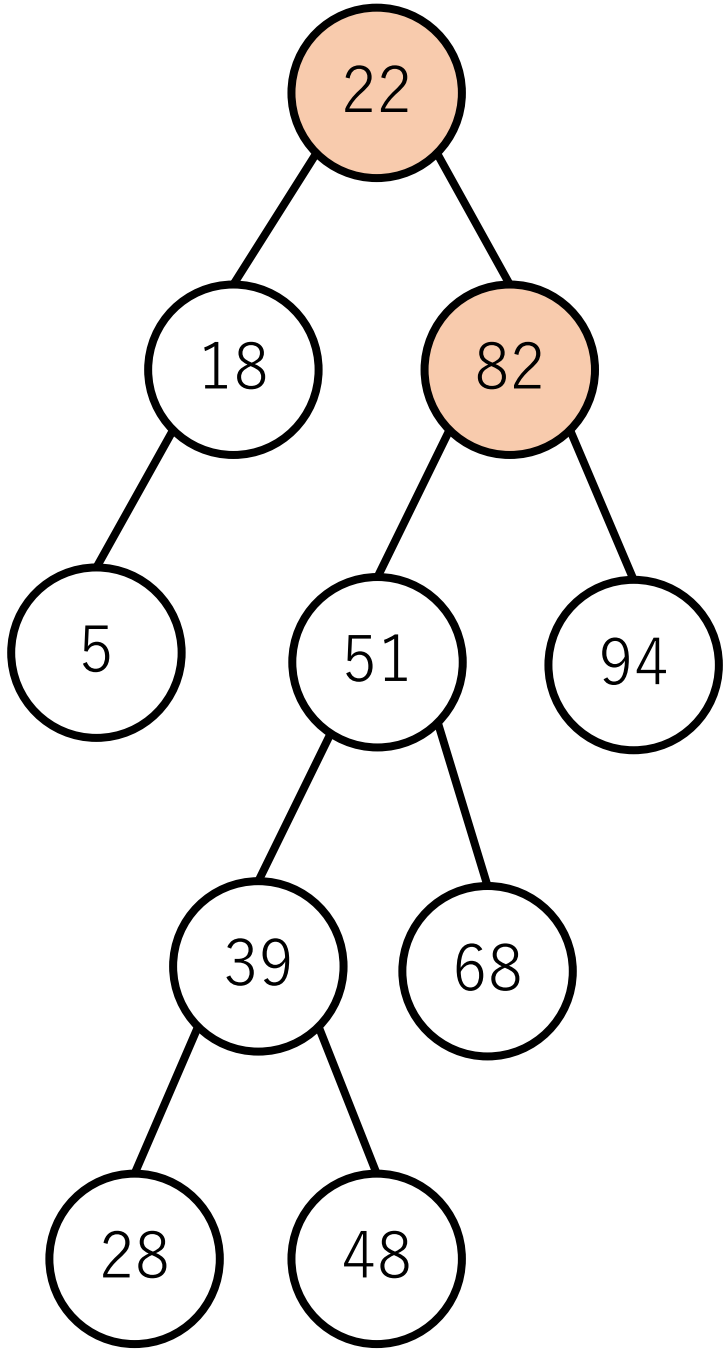
根ノードからスタート．68は22より
大きいので，右の子ノードに移動.



二分木を使って探索

例：68を探す.

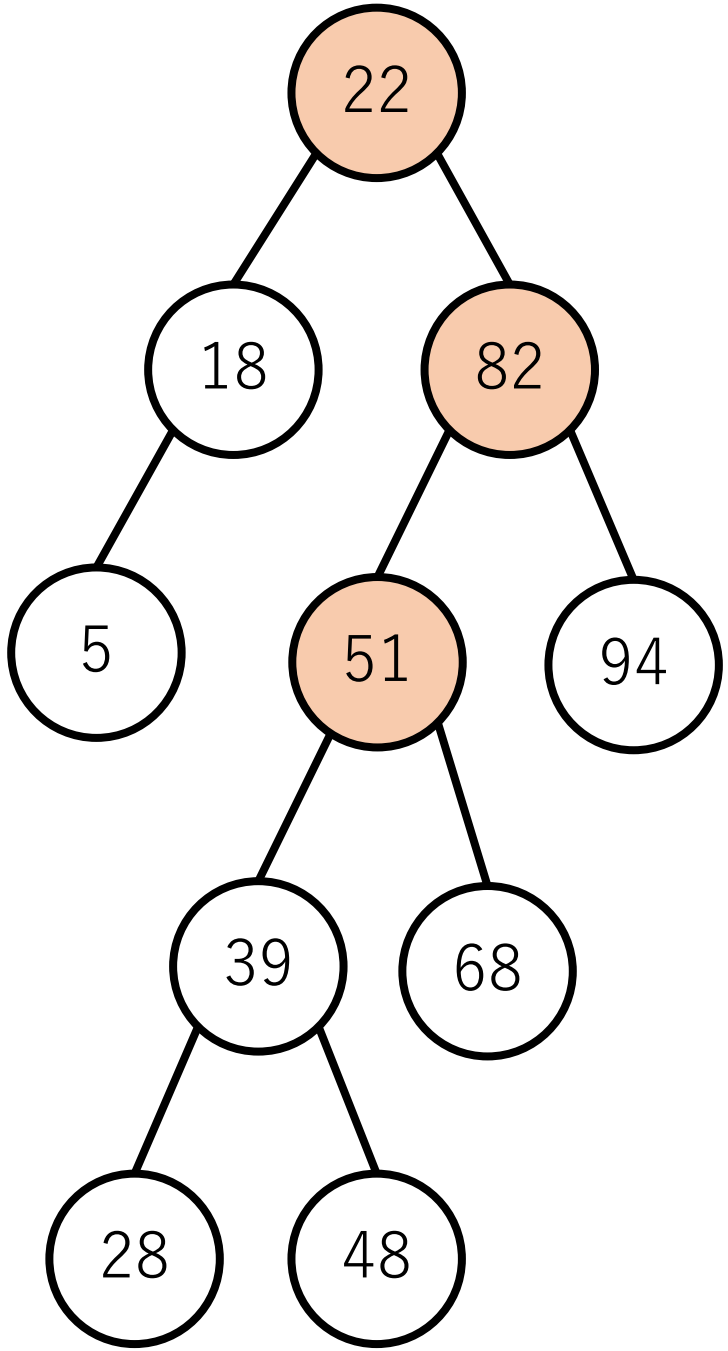
68は82より小さいので，左の子ノードに移動.



二分木を使って探索

例：68を探す.

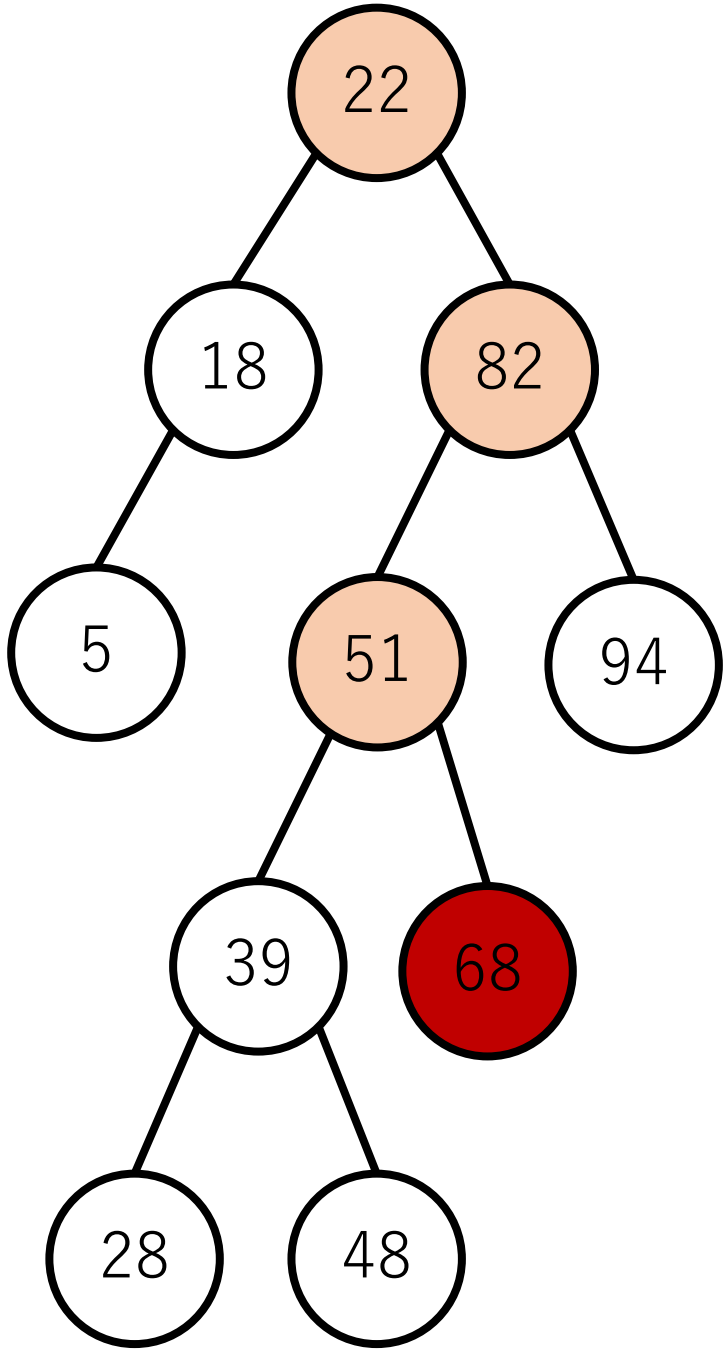
68は51より大きいので，今度は
右の子ノードに移動.



二分木を使って探索

例：68を探す.

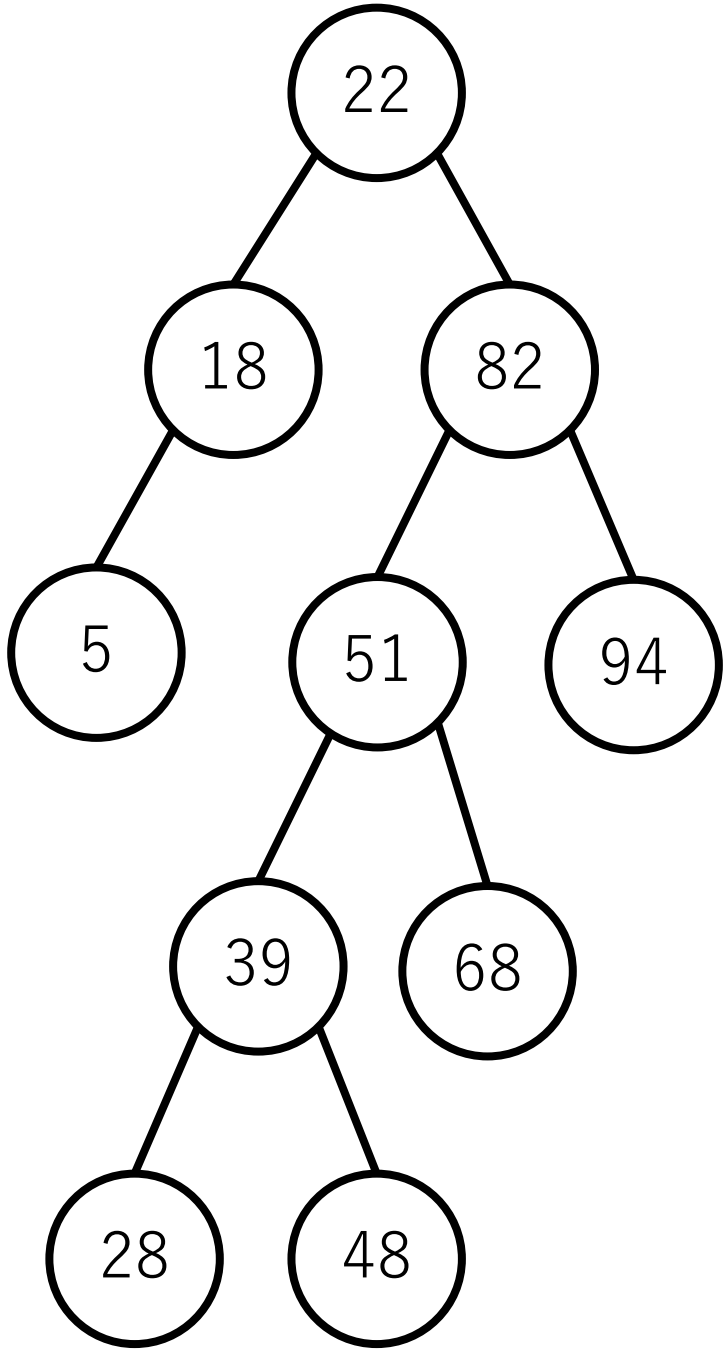
68を発見！



二分木を使った探索の計算量

挿入：

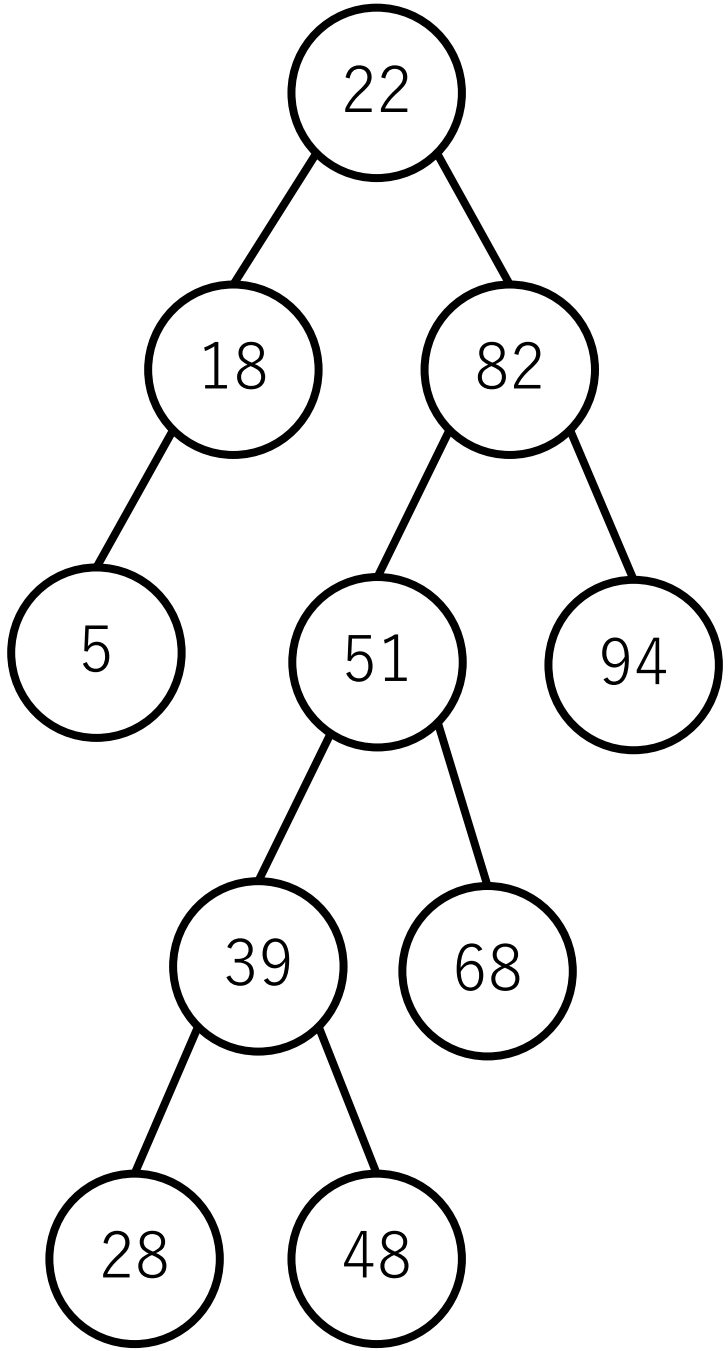
木が「普通の形」であれば，$O(\log n)$回比較をしたあとで追加できると期待できるので，$O(\log n)$.



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のときには，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，

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

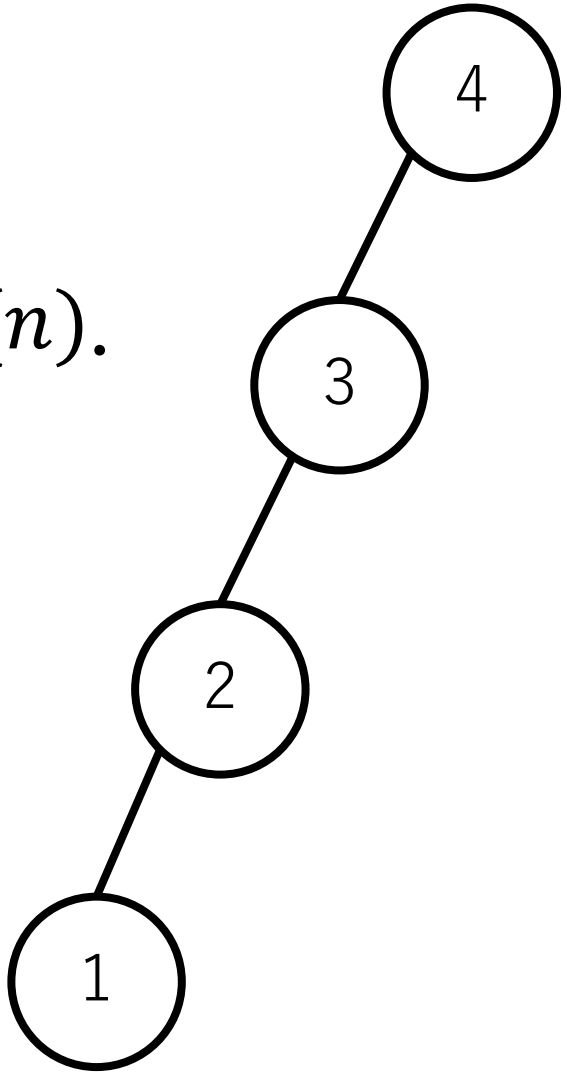
となる.

（大雑把に$O(n \times \log(n)) = O(\log(n^n))$でも問題なし）

最悪の二分木

一方に偏ってしまう.

こうなってしまうと木を作るのも探索も$O(n)$.
（つまり線形探索）



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なっている配列であれば，
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．

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．

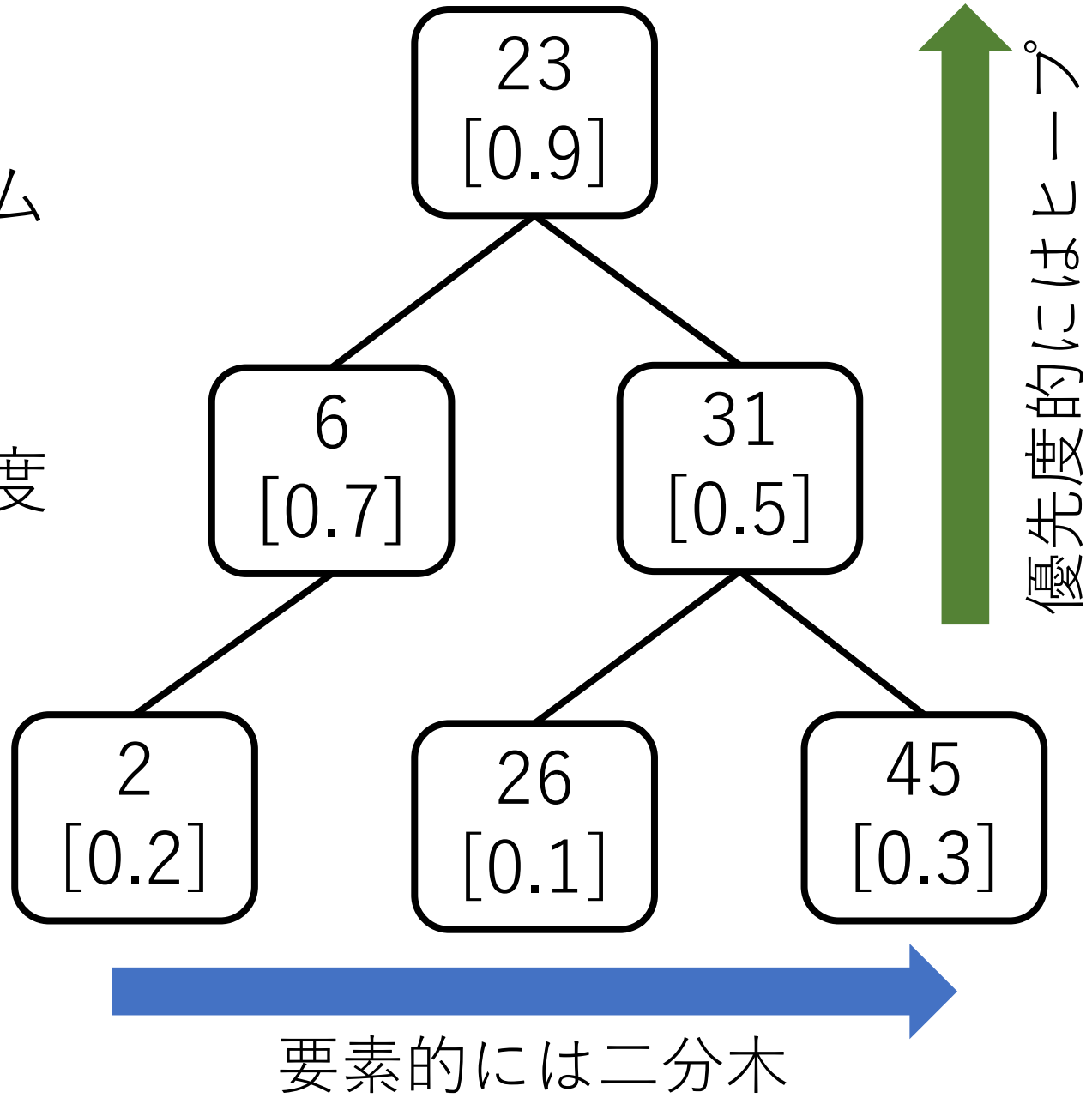
このランダム性を導入できないか？

Treap（ツリープ）

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.



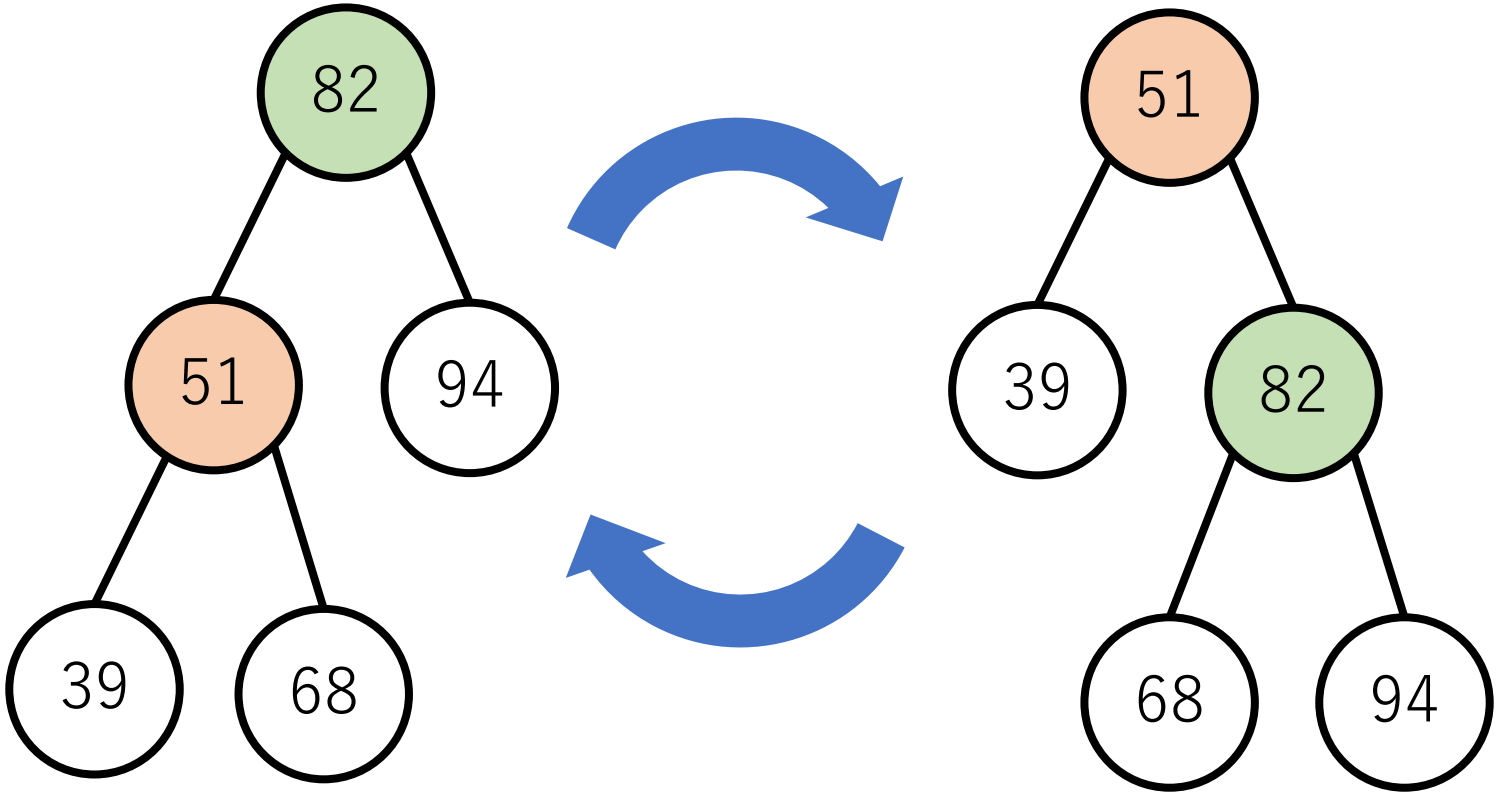
もう1つの方策

一方に偏った木になりそうな場合，修正できないか？

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）.

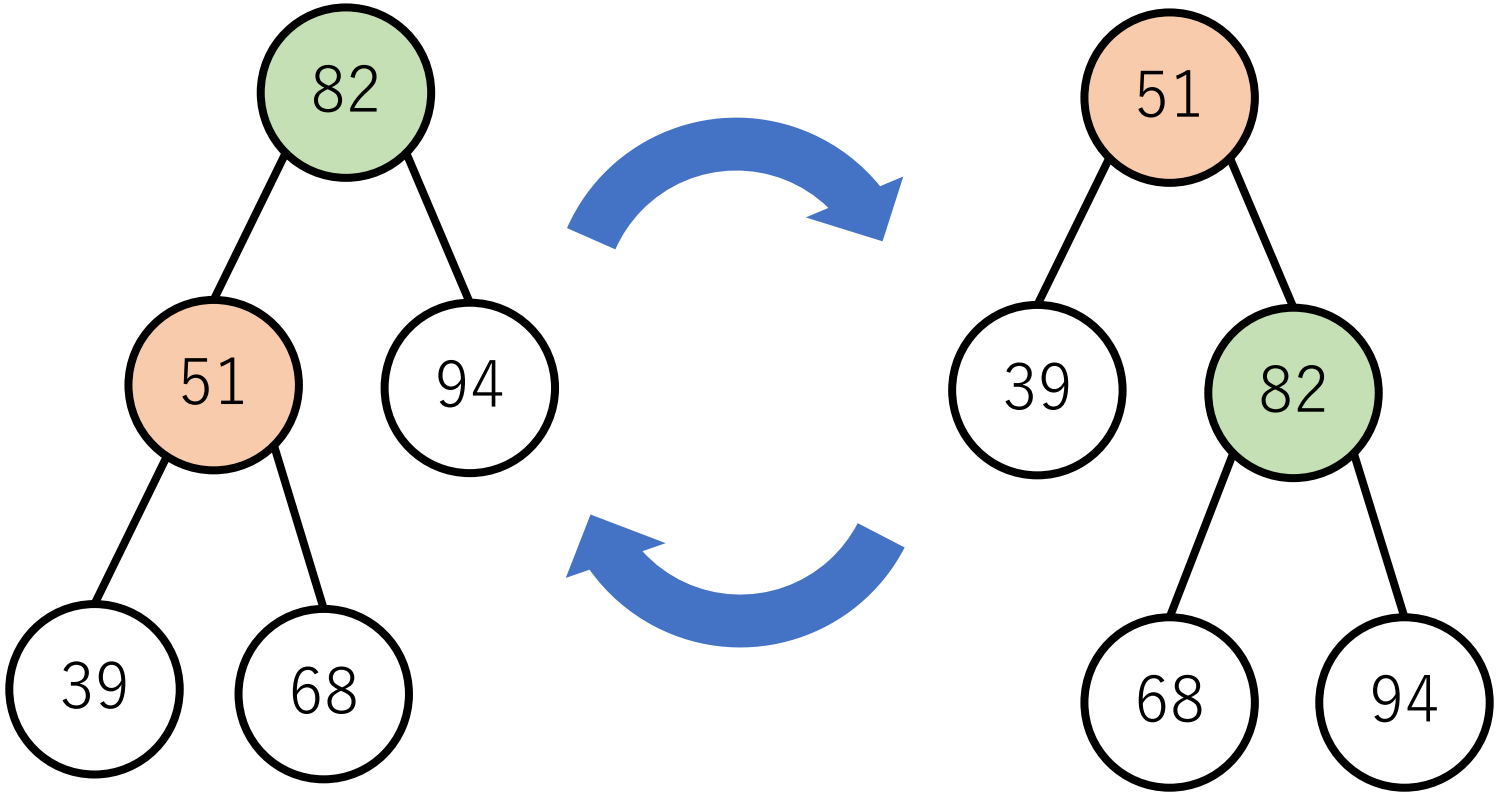
木の回転

要素の順序を崩さずに，あるノードを上にあげる．これによって高さの調整を行う．



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）

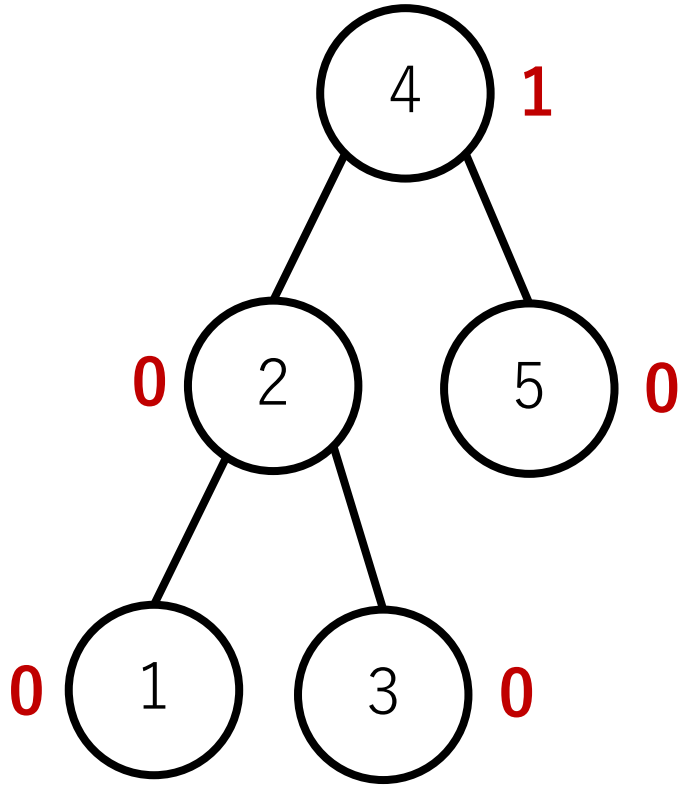
balance factorを各ノードに定義し，この値を見てどんな回転をするかを決める．

[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.

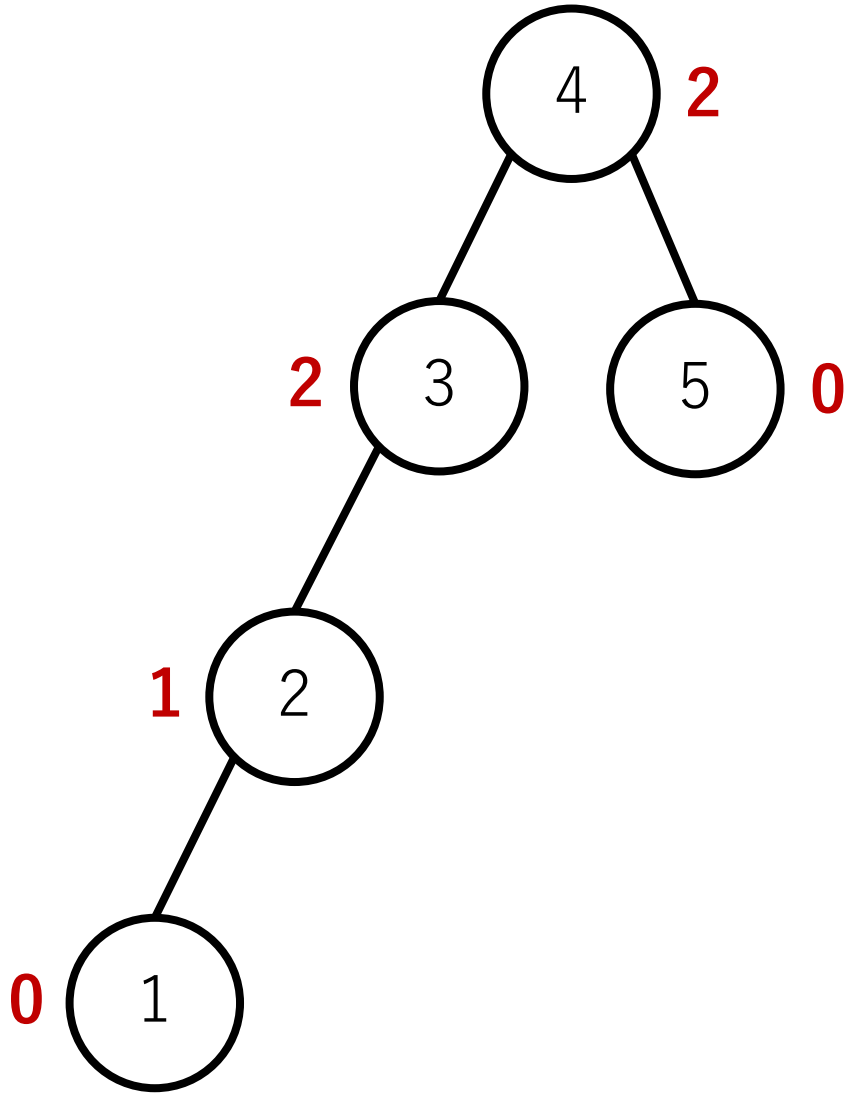
これは回転が必要ない．（全ての部分木の左右の高さの差が高々1）



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.

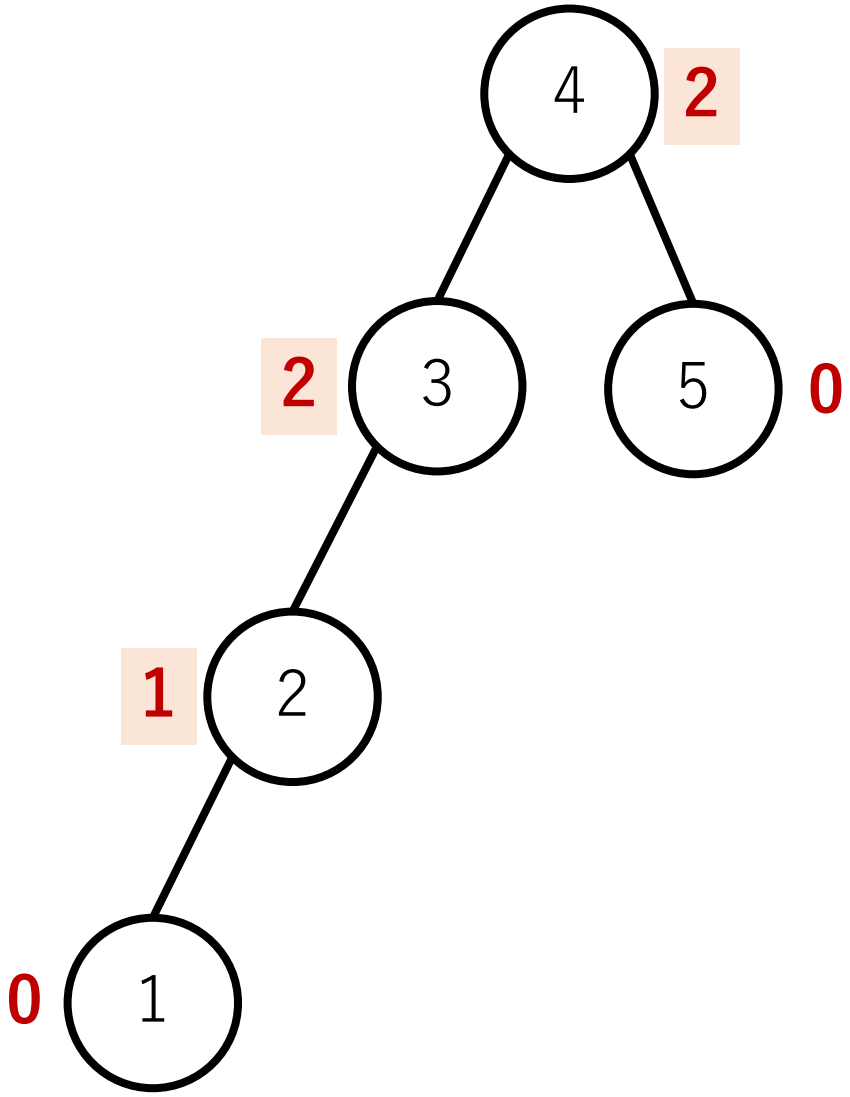
この木の場合は根ノードの下の左右の部分木の高さの差が2になっている.



AVL木の例

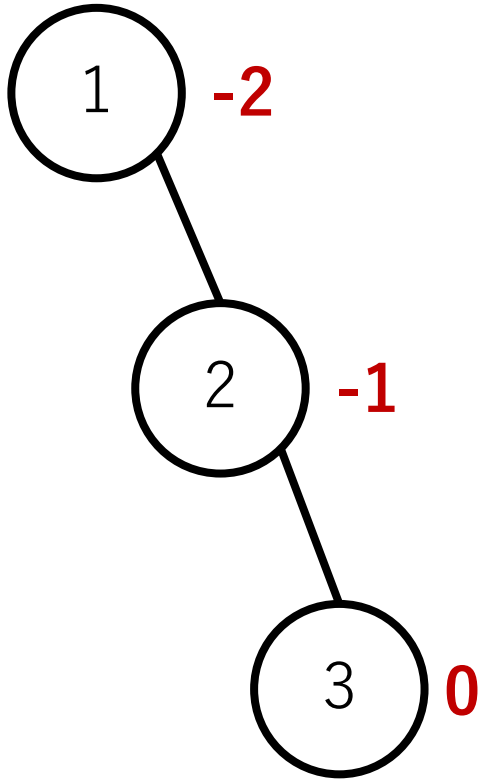
balance factorが各ノードに付与されている．このbalance factorの絶対値が2以上になったら回転が必要な合図．

よって，この場合は何かしらの回転を施したい．



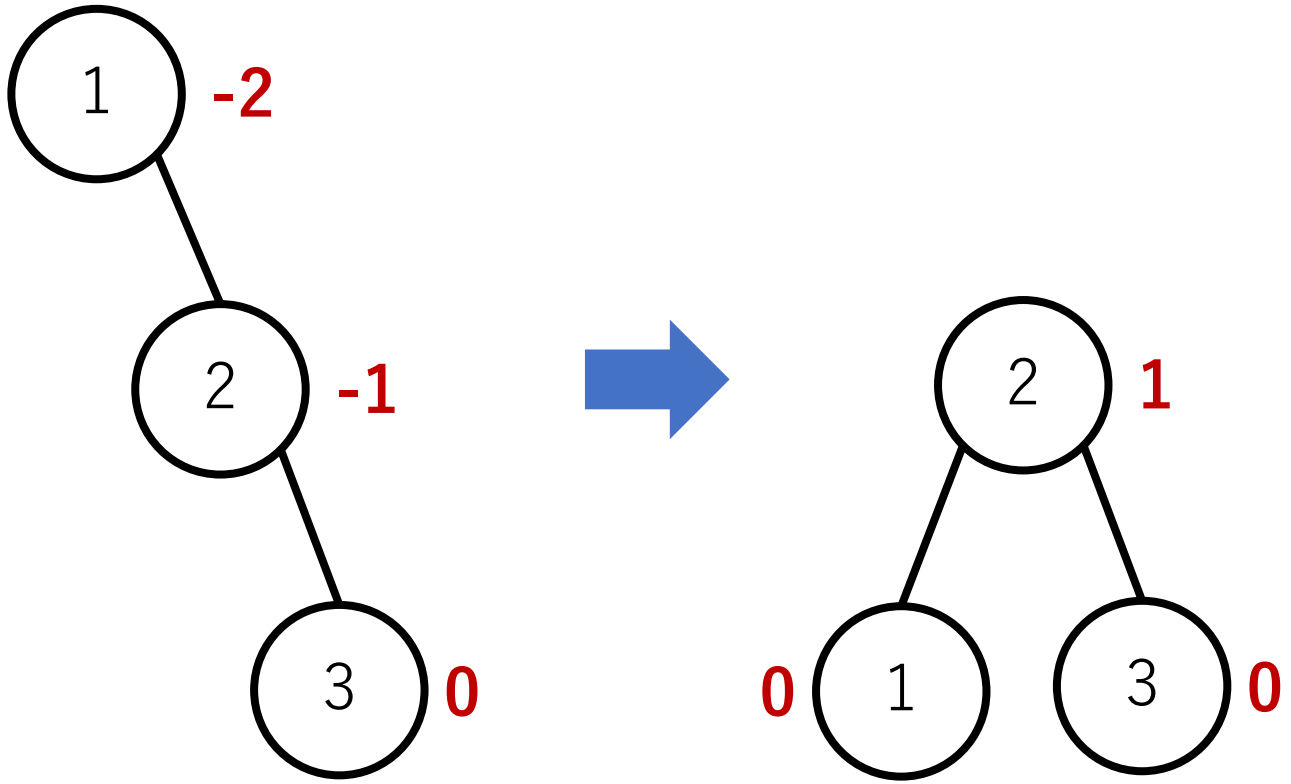
AVL木における回転

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



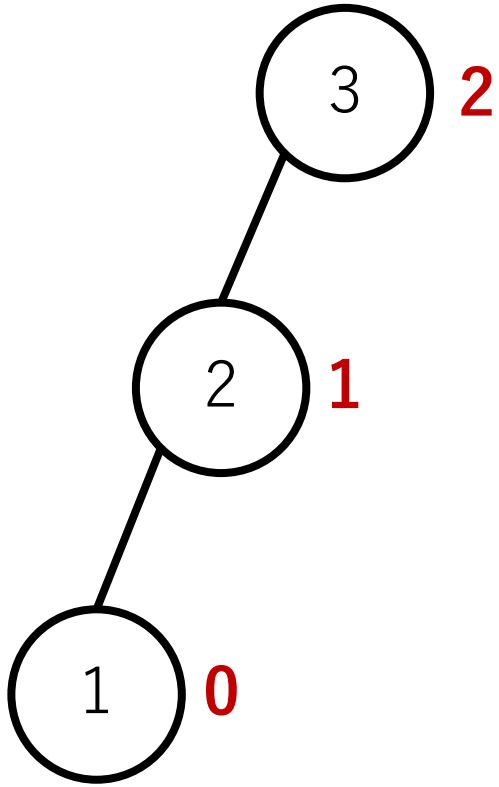
AVL木における回転

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



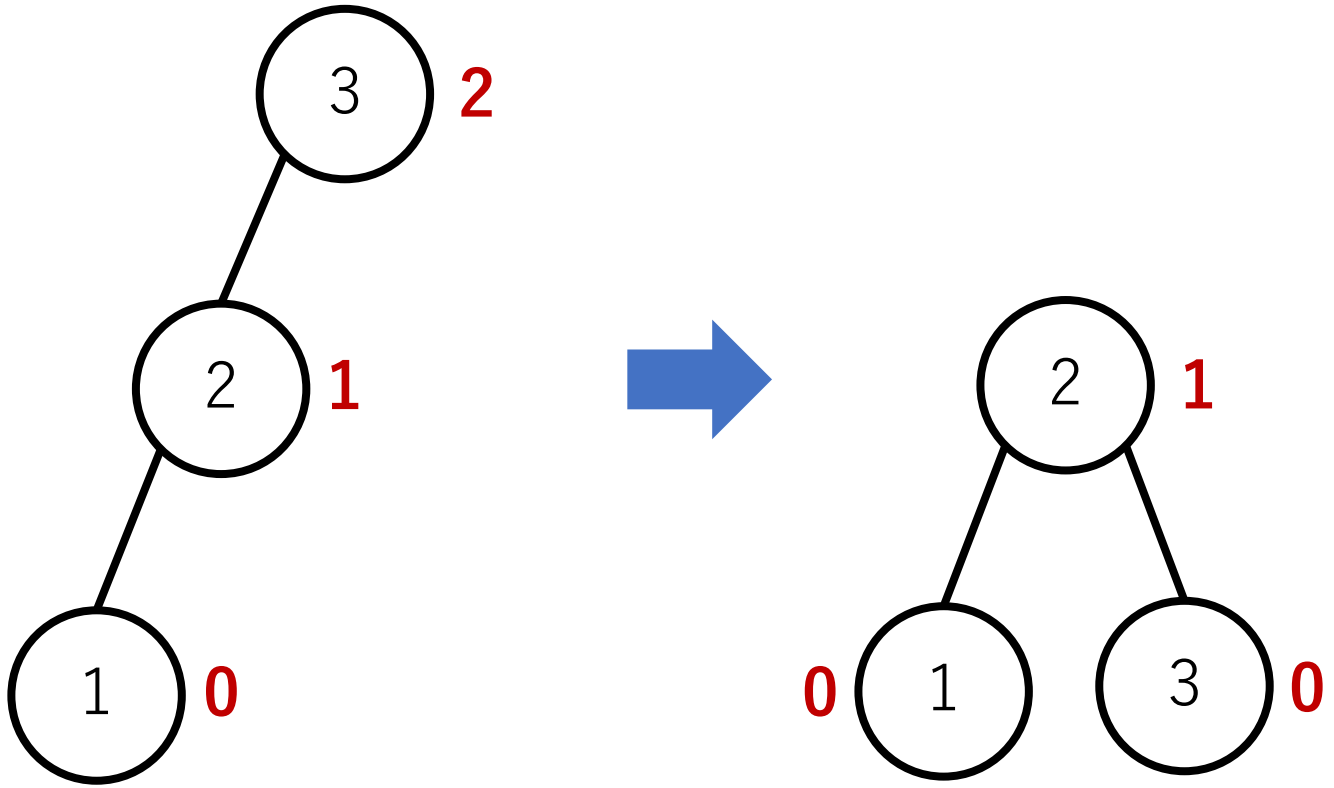
AVL木における回転

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



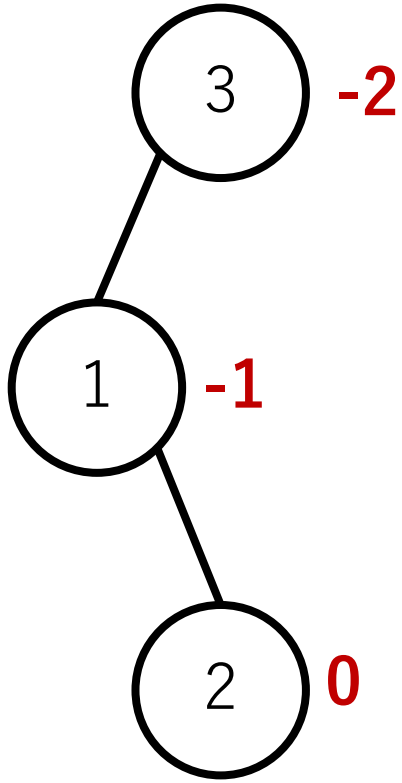
AVL木における回転

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



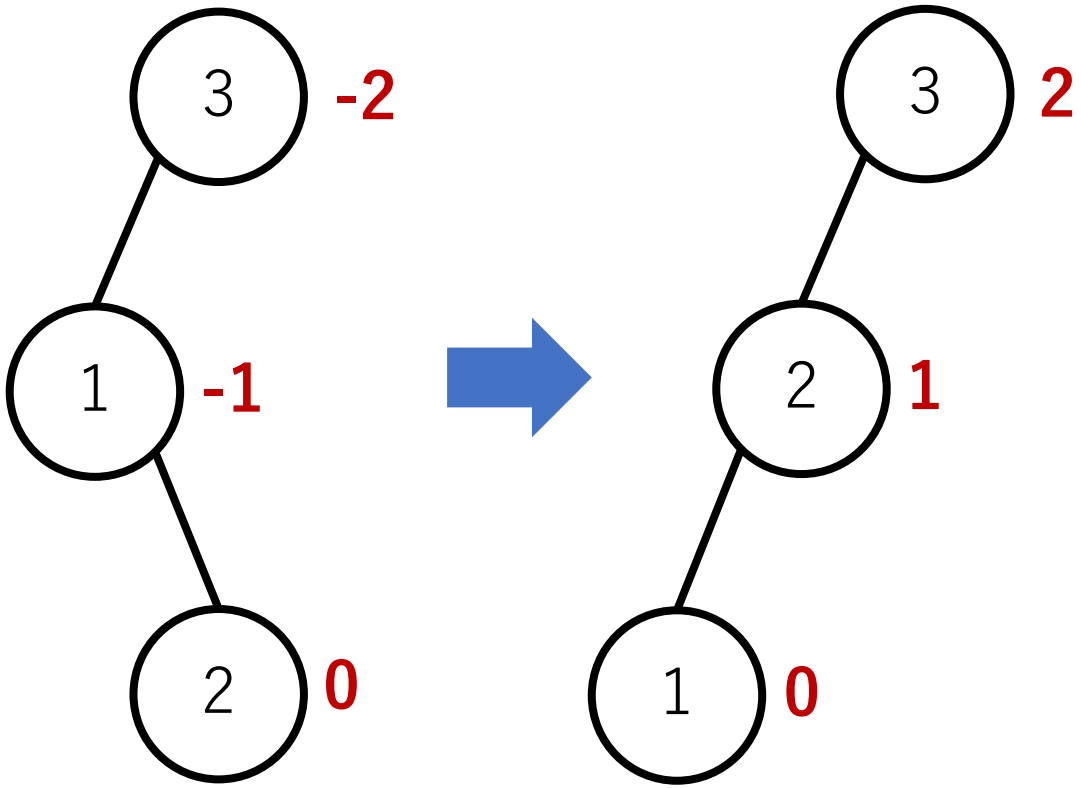
AVL木における回転

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



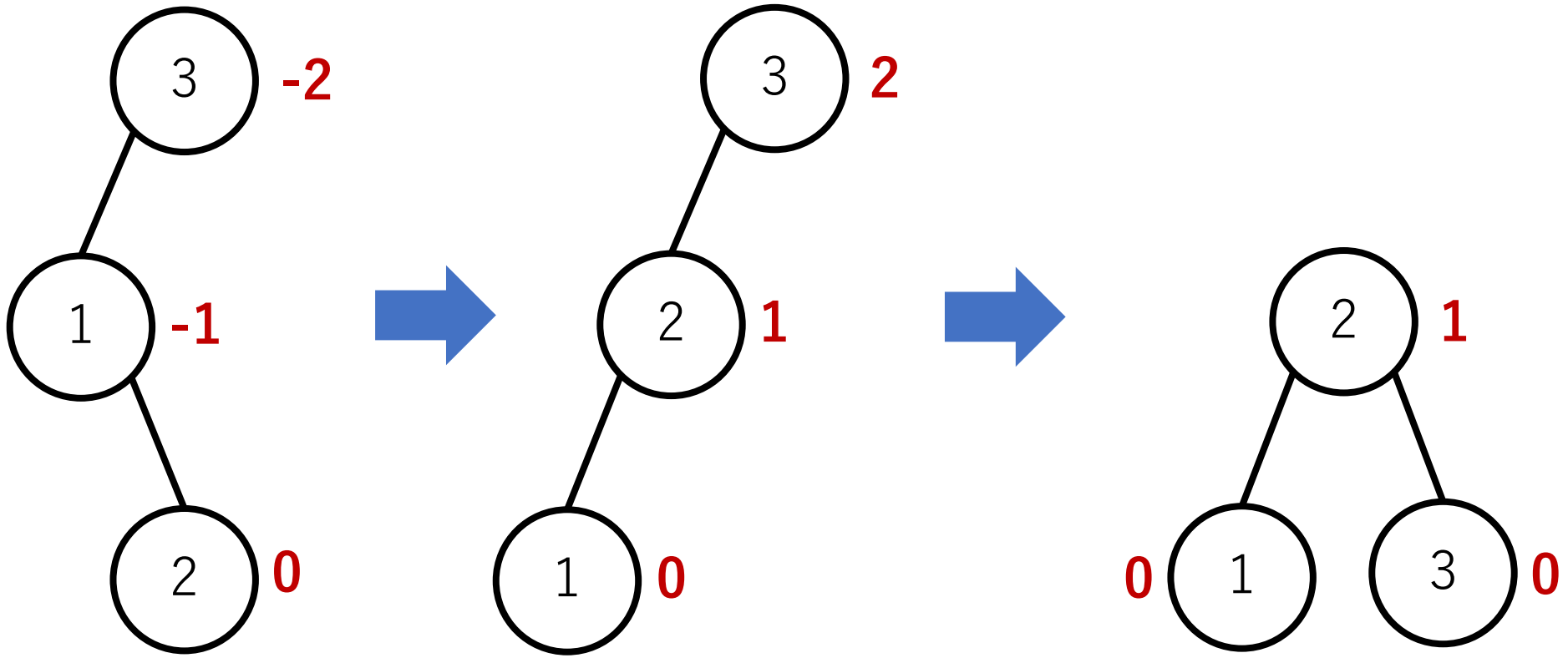
AVL木における回転

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



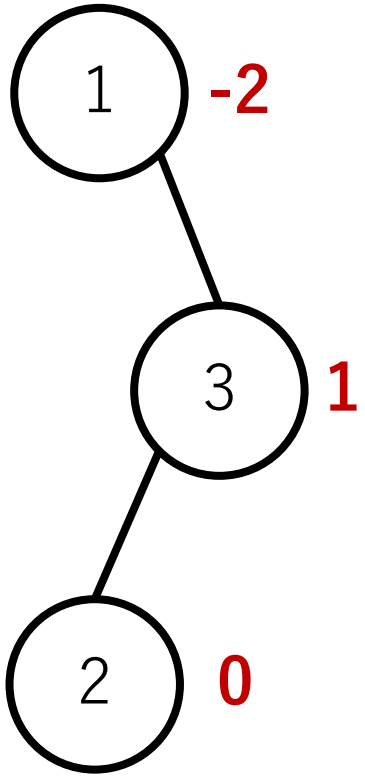
AVL木における回転

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



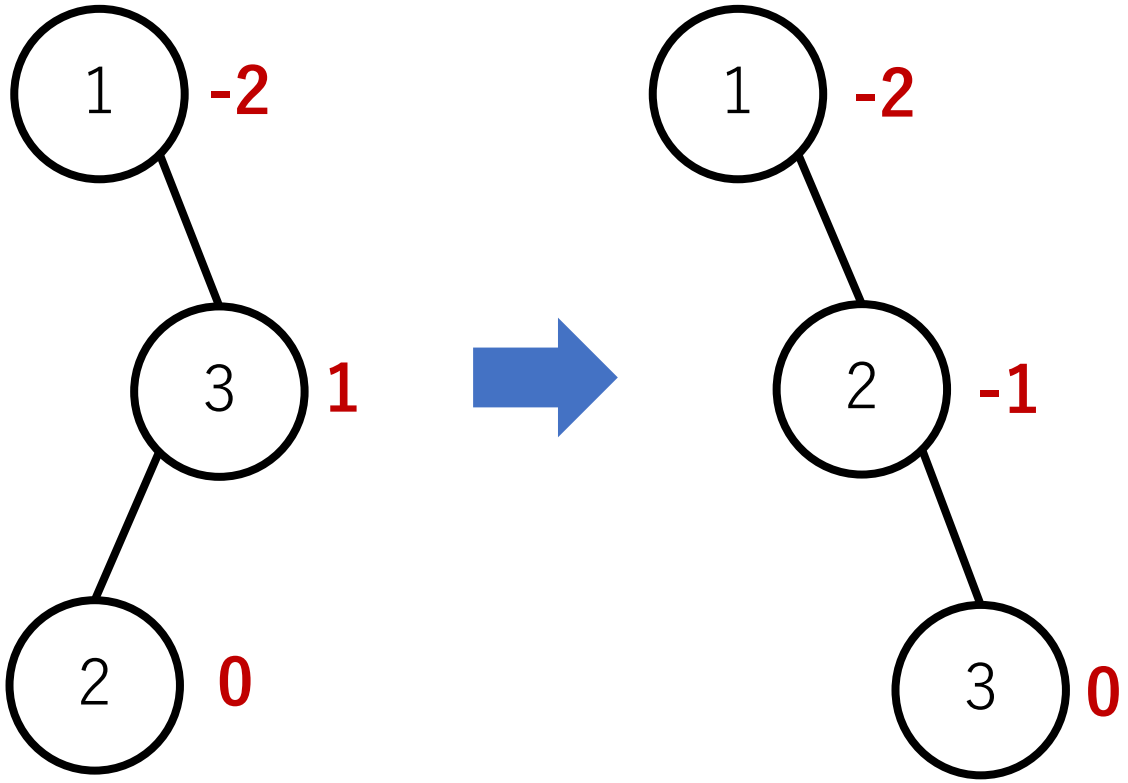
AVL木における回転

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



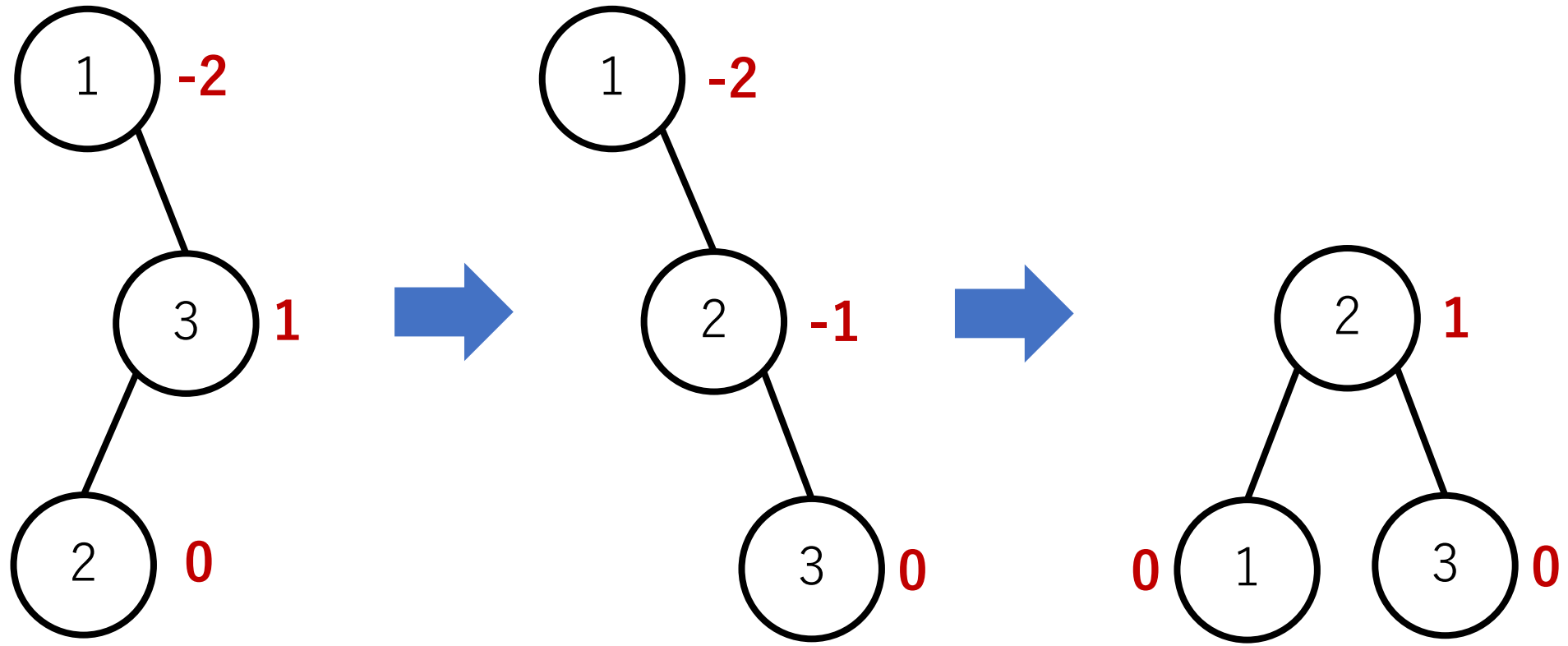
AVL木における回転

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.

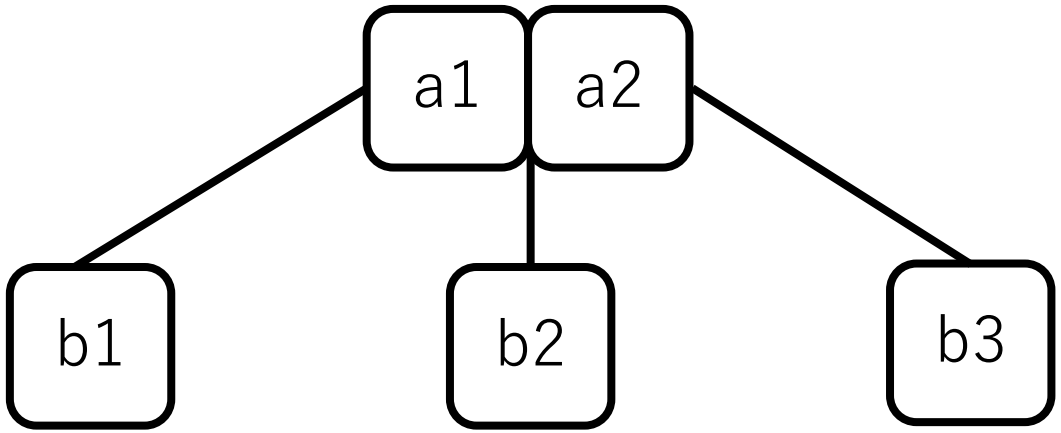
各ノードでの子ノードへの枝の数を最大mまで許す.
これをm次のB木（オーダーmのB木）という.

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

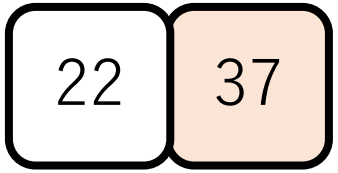
例：22, 37, 17, 9, 45, 34, 18

2-3木の例

例：22, 37, 17, 9, 45, 34, 18

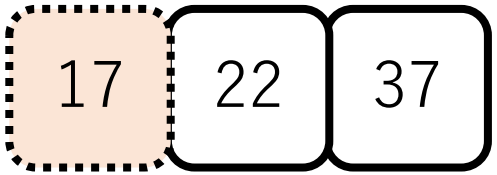
2-3木の例

例：22, 37, 17, 9, 45, 34, 18



2-3木の例

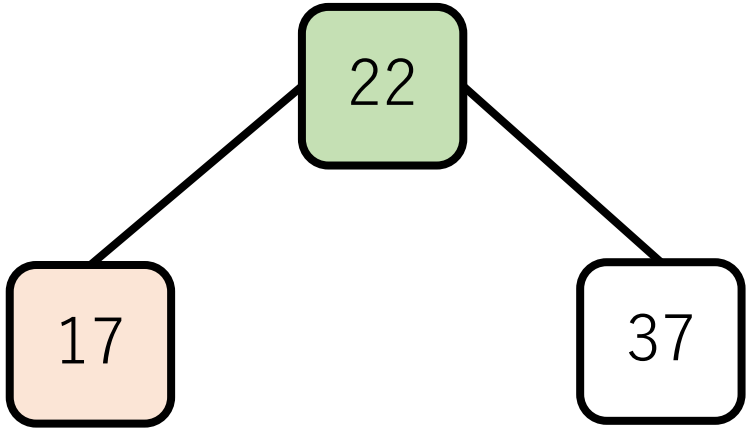
例：22, 37, 17, 9, 45, 34, 18



ここには入れない．

2-3木の例

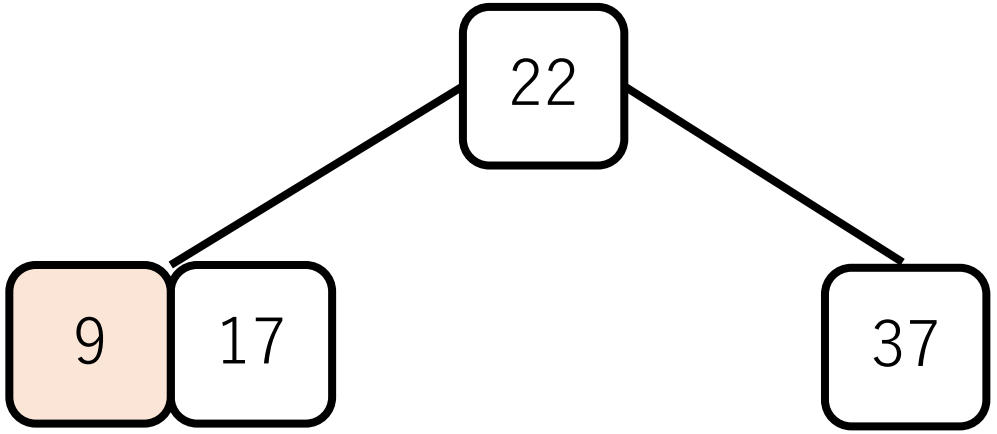
例：22, 37, 17, 9, 45, 34, 18



中央の値を親ノードにして，
残りの2つを分割する．

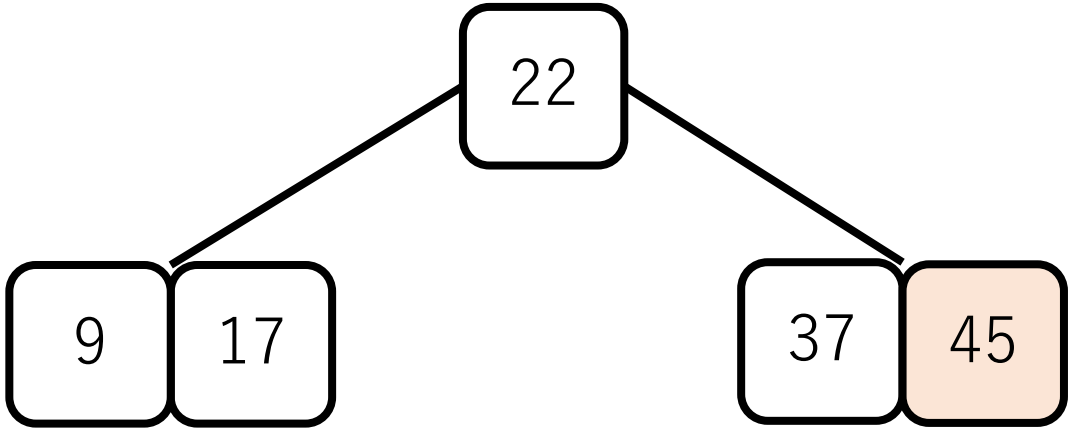
2-3木の例

例：22, 37, 17, 9, 45, 34, 18



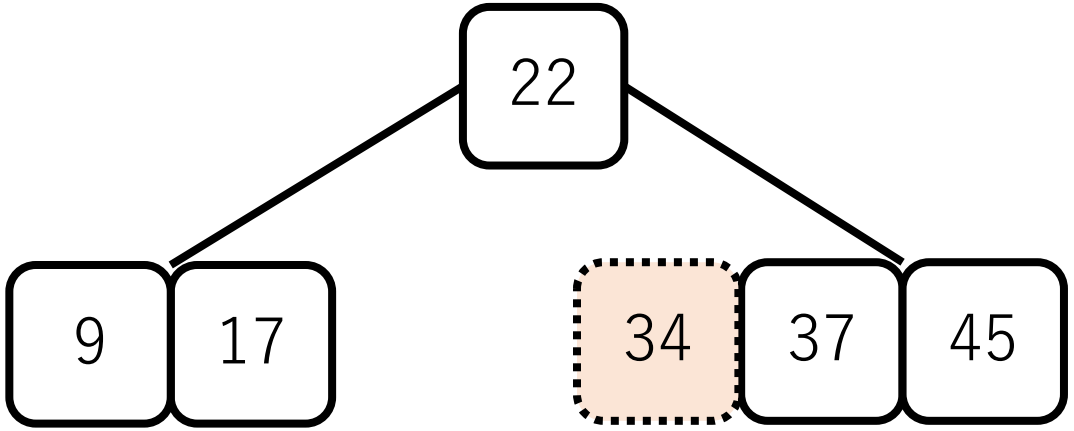
2-3木の例

例：22, 37, 17, 9, 45, 34, 18



2-3木の例

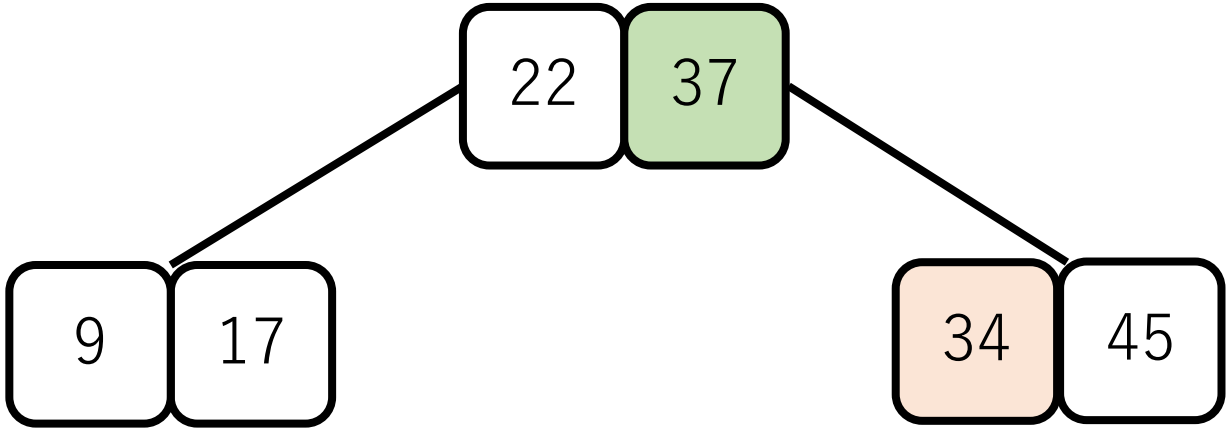
例：22, 37, 17, 9, 45, 34, 18



ここには入れない.

2-3木の例

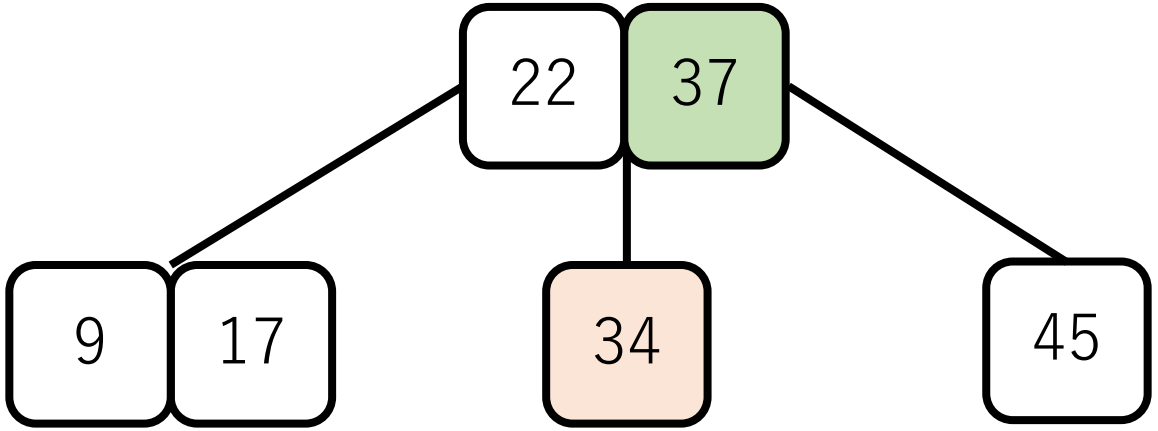
例：22, 37, 17, 9, 45, 34, 18



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

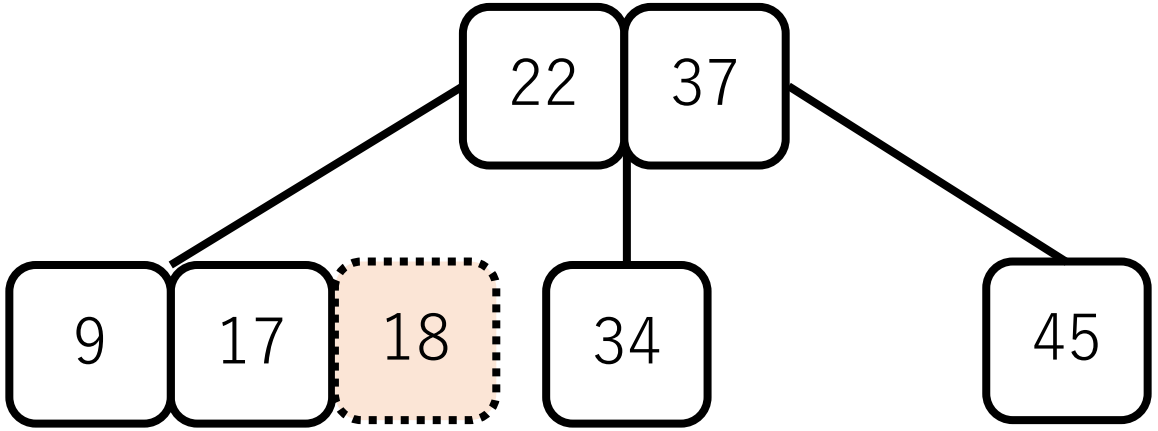
例：22, 37, 17, 9, 45, 34, 18



そこで，34と45を分割する．

2-3木の例

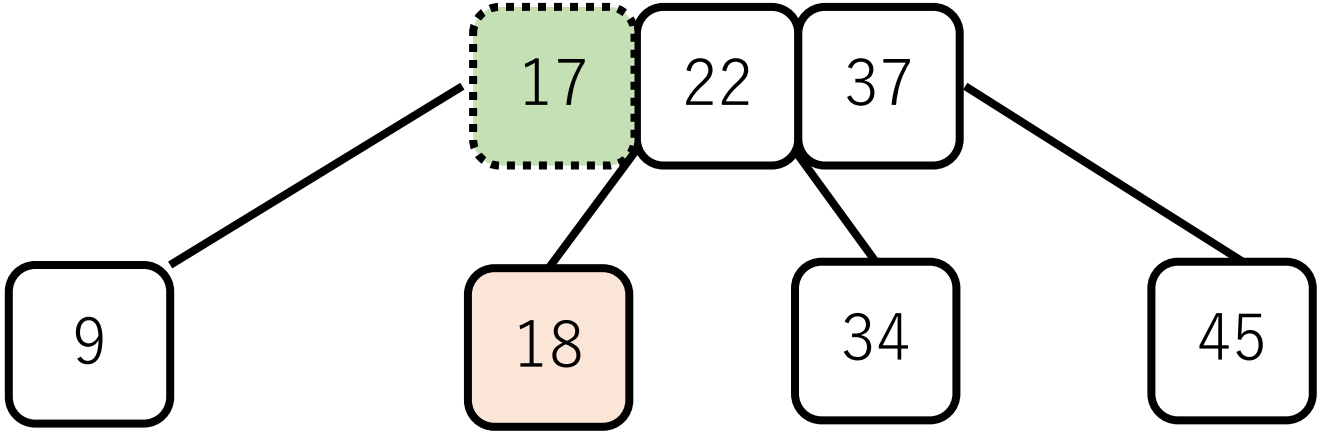
例：22, 37, 17, 9, 45, 34, 18



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

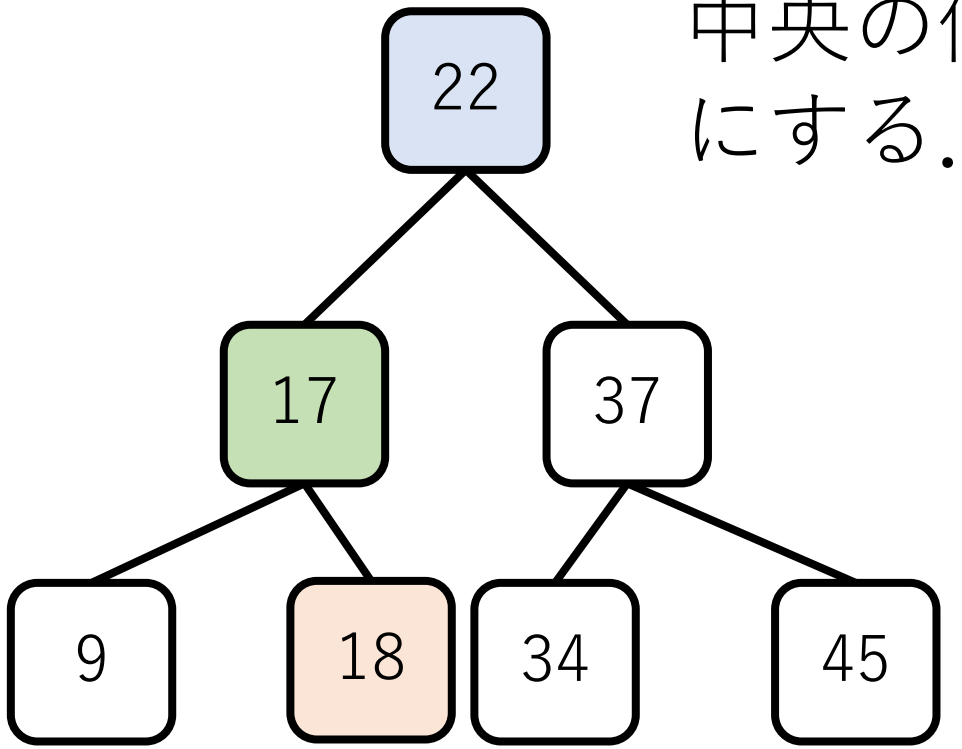
例：22, 37, 17, 9, 45, 34, 18



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．

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，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.

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 [1.5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.

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 [1.5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#3 [3点]

探索アルゴリズムを使用する問題.

問題文には「ソート」という言葉がありますが，明示的なソートを必要とせずに問題を解くことができます.
もし，どうしても必要であれば，sort関数等を使ってもらって構いません.