Algorithms (2020 Summer)
#3：探索（サーチ）

矢谷 浩司

前回の質問

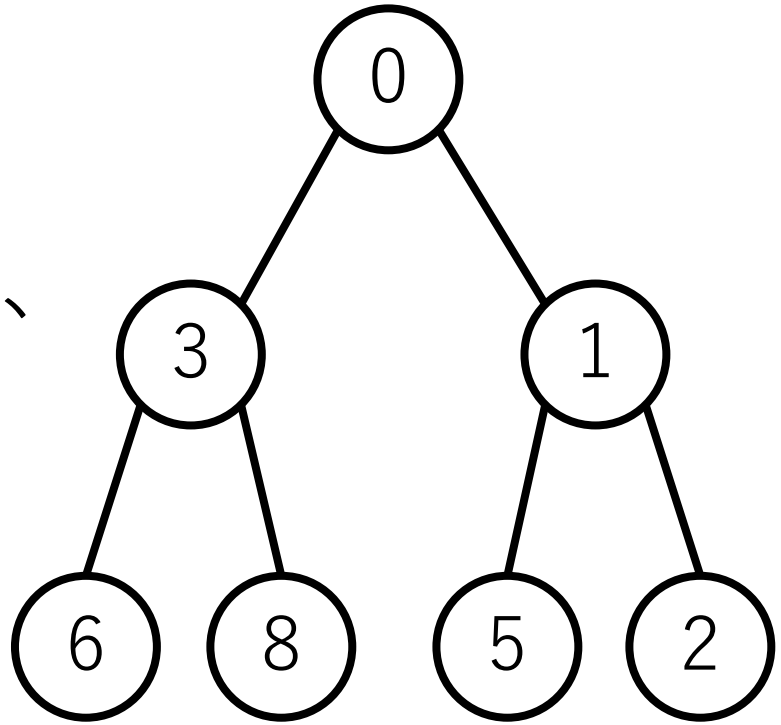
「（基本課題において）この条件でリングバッファにする理由やメリットがわからない」

→リングバッファを使うと，バッファのサイズを固定にできます．よって，空間計算量を固定にしながら，未知のクエリ数にも対応できるように実装できることになります．

TAさんより「テストケースがオープンなのでどのくらいまで（バッファのサイズ）を小さくしても壊れないかテスト実行で確認してみると良いかも．」

前回の質問

「スライドにあったヒープの操作:0の追加を、元々のヒープの4が2であった場合に行うと、2の深さ＞3の深さとなりますが問題ないのでしょうか。(ヒープにおいて、a＞b であるとき、aの深さ＞bの深さ、は必ずしも成り立たないのですか)」



→問題ありません．ヒープにおいては制約条件さえ満たしていれば（親ノードは子ノードよりも小さい），あとは問われないことになります．

前回の質問

「課題提出期限も最後まで解きたいので，採点に反映されなくても再チャレンジできますか？」

→残念ながら，今のシステムはサポートできないので，予めローカルにコピーを取っておいて，提出後に改めて自習していただければと思います．（Giveryさんへのフィードバックとさせてもらいます．）

前回の質問

「今回提出期限が前回より短いですが、今後はこの期限になるのでしょうか」

→はい．3日後の20:00:00が締切となります．

「講義参加者のZoom飲み会やってみたいですね」

→😆🍻お手伝いしてくれる方がいらっしゃったらぜひご連絡を！

提出物の返却に関して

TAさんがいろいろと方法を検討してくださったのですが,
提出物を皆さん1人1人のもとに返すうまい方法が現状
ありません．申し訳ありません．．．

したがって，もしご自身のコードを取っておきたい,
という方は提出前にローカルに保存してください.

TAさんの負荷を増大させないため，提出後の対応はしない,
ということで統一いたしますので，どうぞよろしくお願い
いたします.

Extra課題の解説

Extra課題の解説をTAさんが用意してくださいました.

授業のzoom linkを掲載しているスプレッドシートに書かれているリンクを辿って，ダウンロードしてください.
（ECCSアカウント必要）

基本課題はスライドをご覧いただければ，きっと出来ると思いますので，解説はありません.

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．

「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）」というもの．

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
```

ちょっとだけ効率化：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!

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.

二分探索の計算量

1段階経ると，探索範囲はおよそ半分になる．

よって最悪の場合（最後まで見つからなかった場合）で$\log n + 1$回の操作が必要となる．

よって，$O(\log n)$．（ただし，ソートにかかる時間は除く．）

二分探索の前提条件

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$O(n \log n)$の計算量がかかる（5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

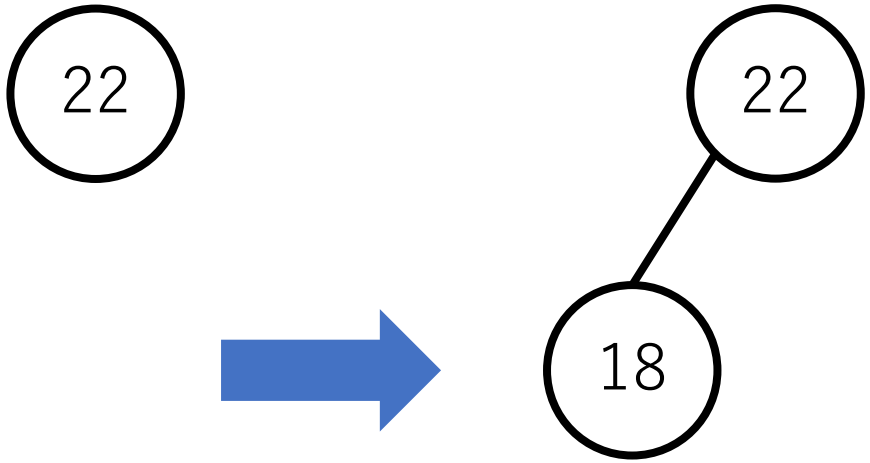
例：[22, 18, 5, 82, 51, 39, 48, 94, 68, 28]

二分木を作る

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

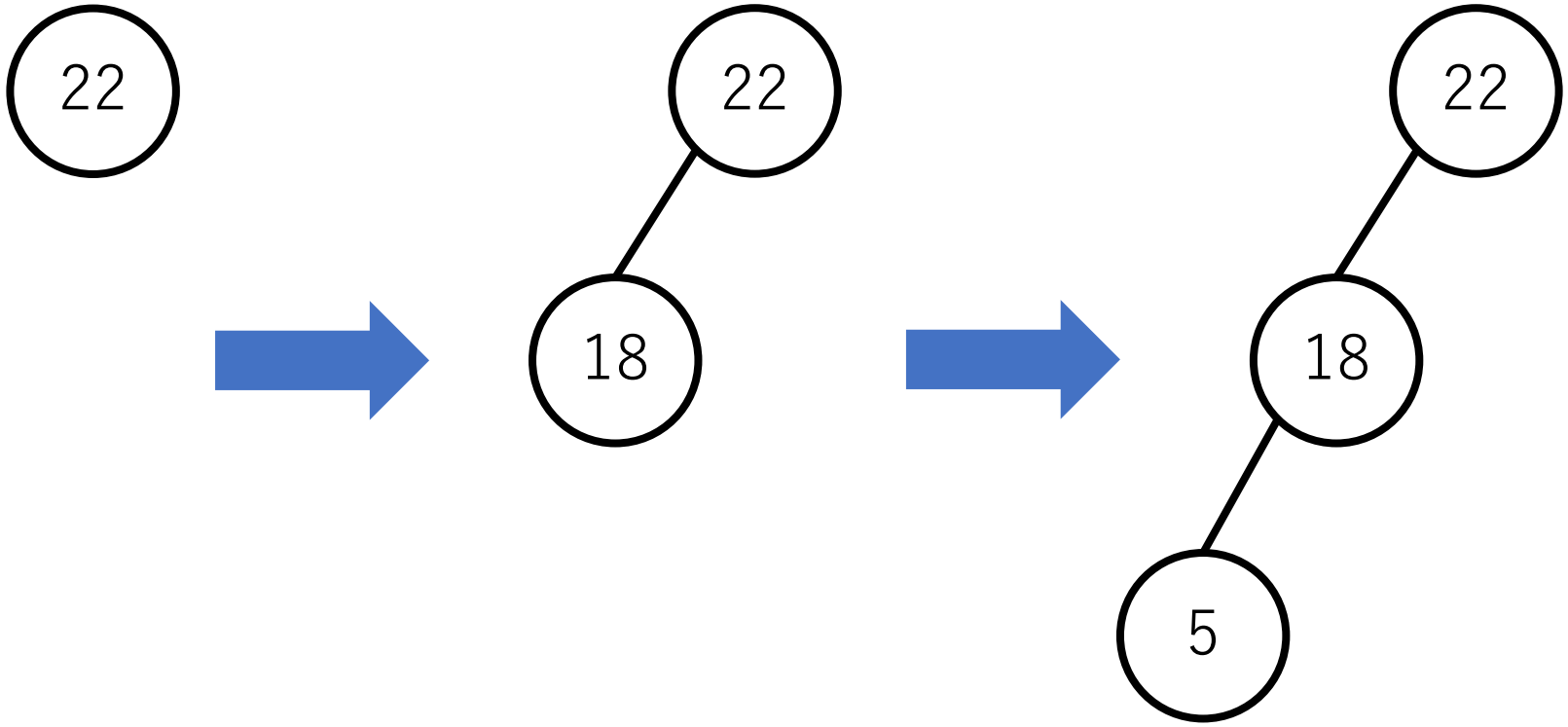
二分木を作る

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



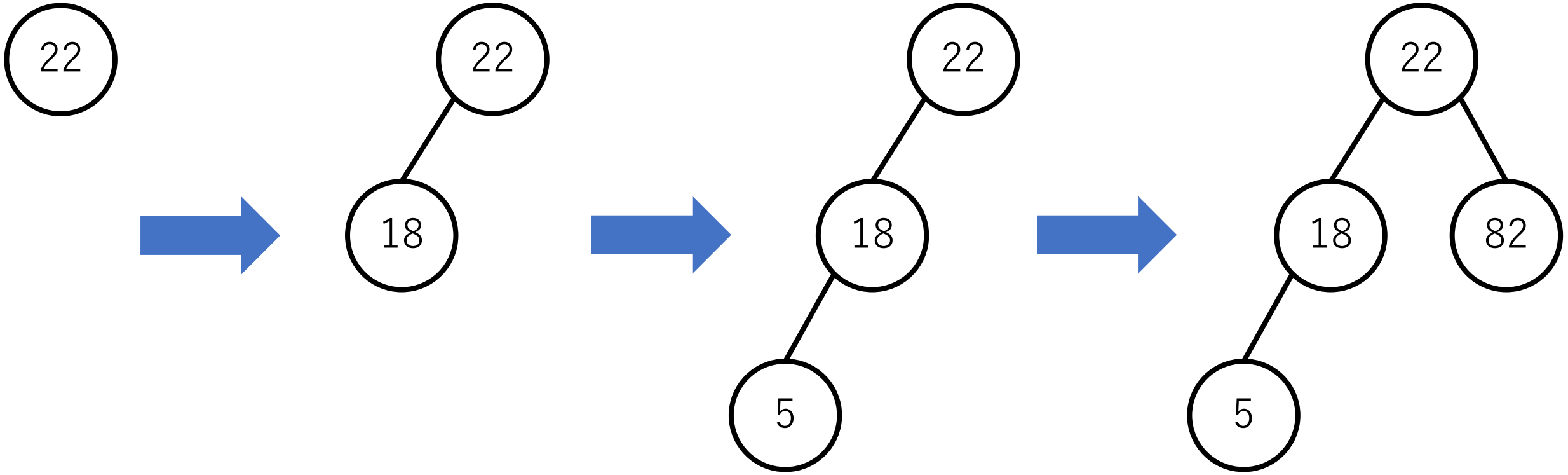
二分木を作る

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



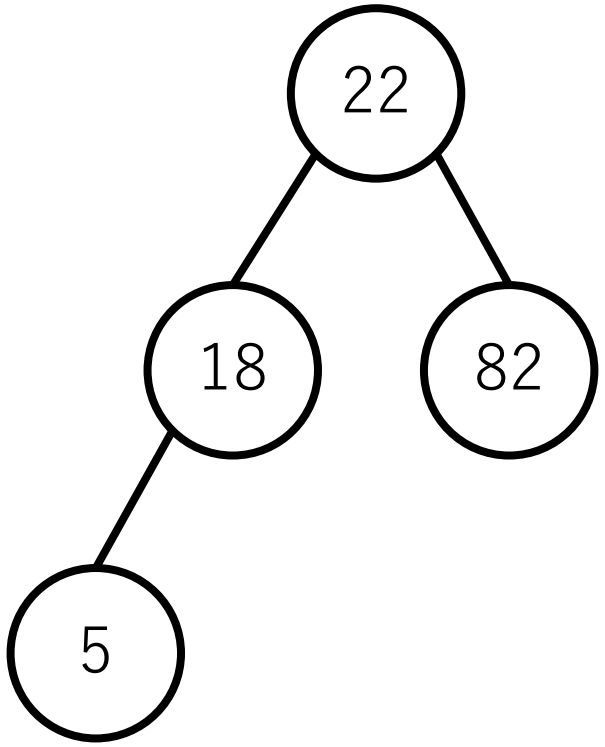
二分木を作る

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



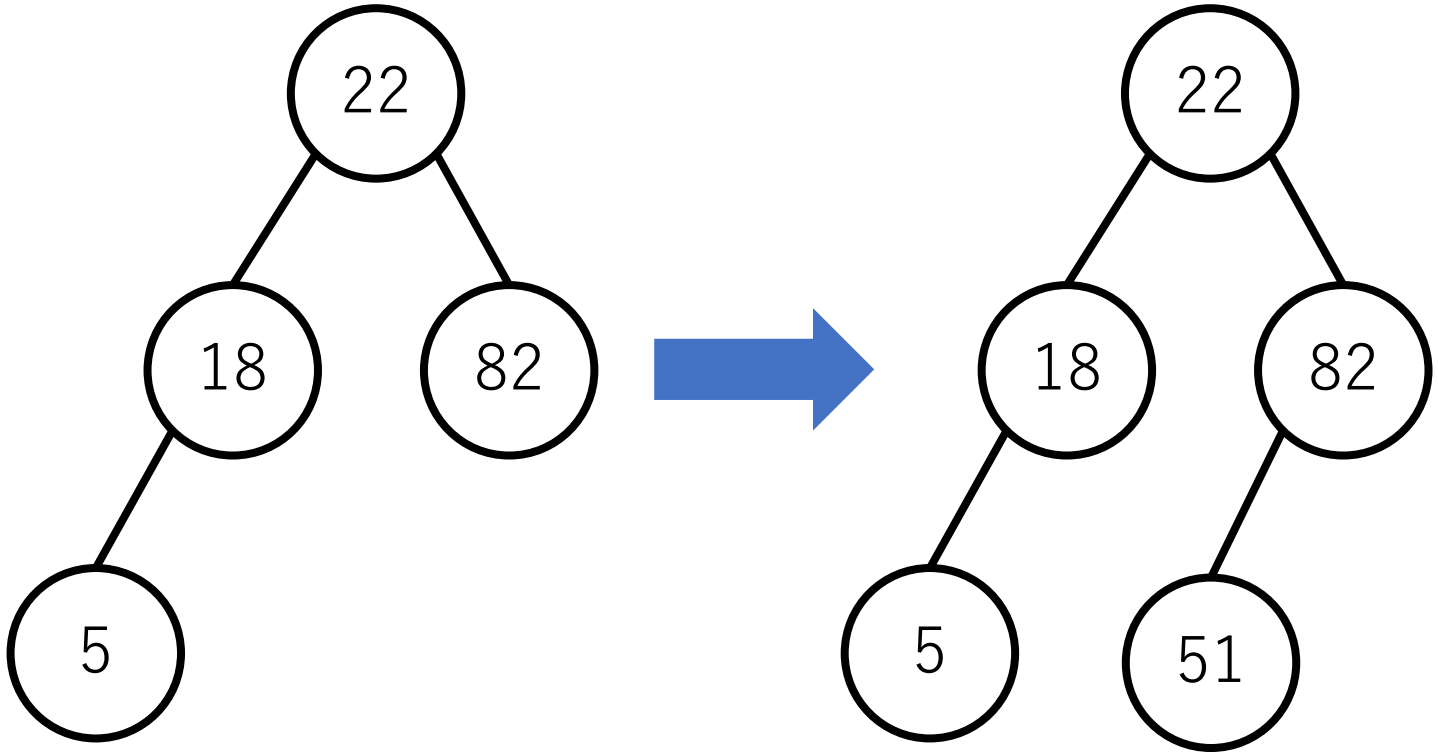
二分木を作る

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



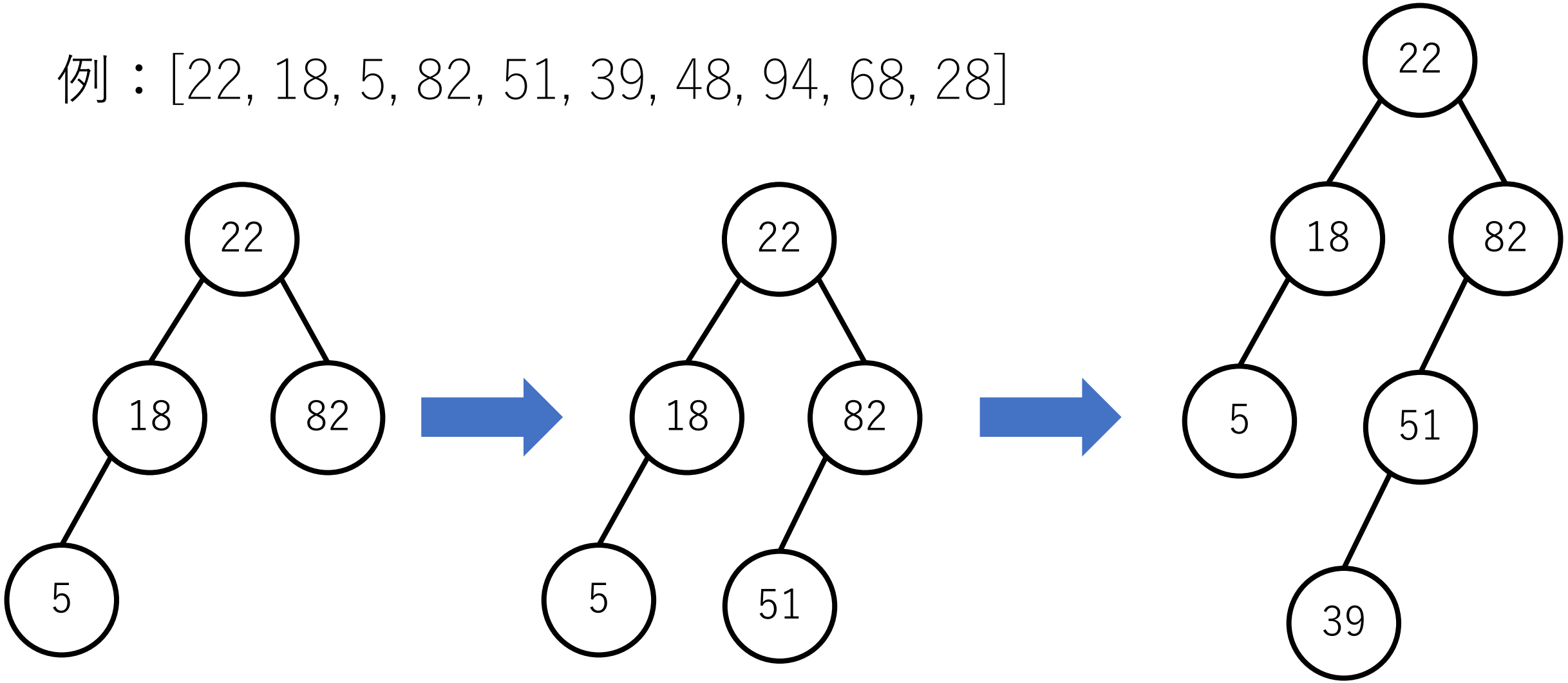
二分木を作る

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]

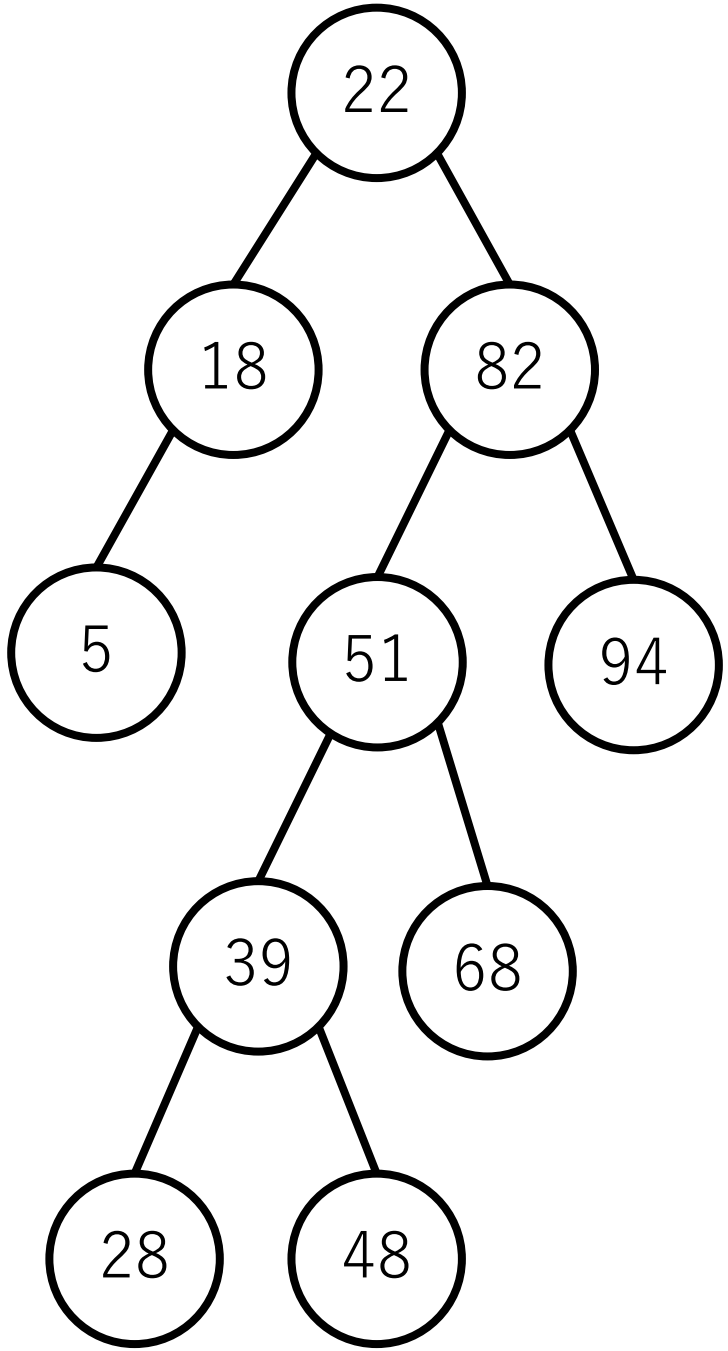


二分木を作る

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.

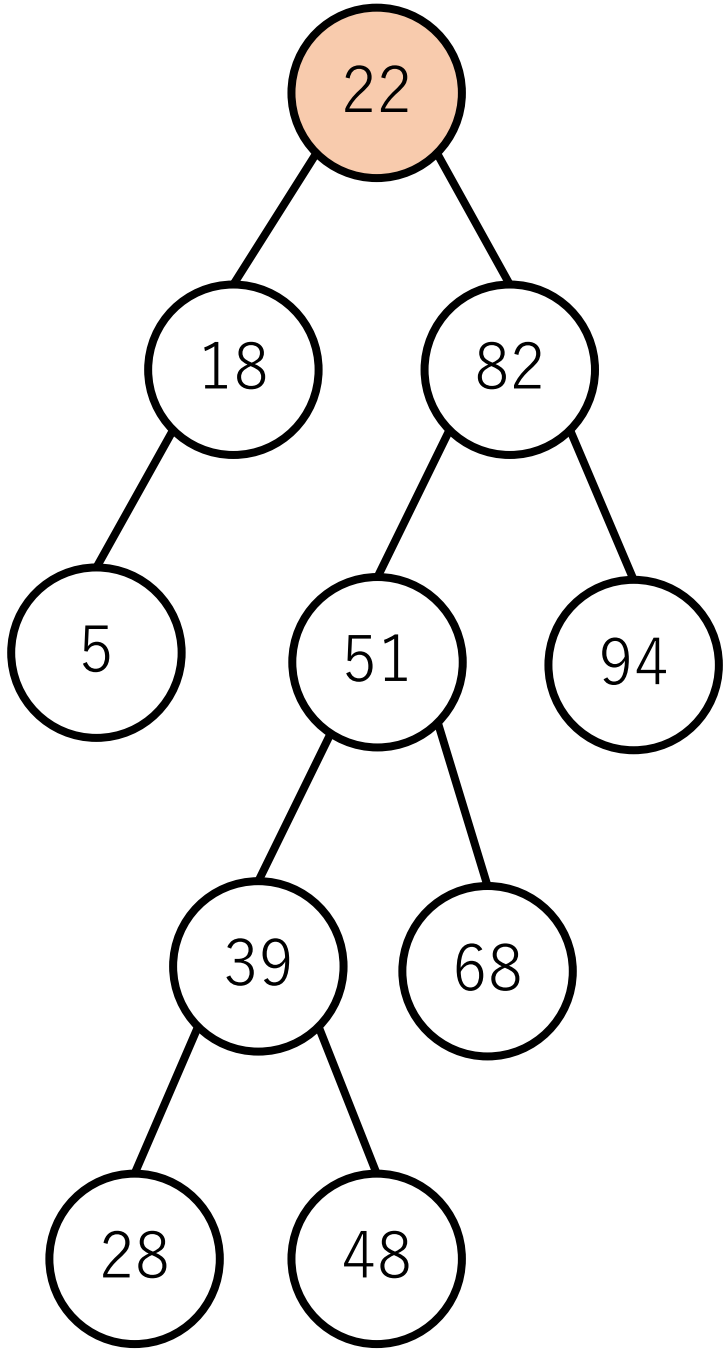
各ノードより小さな値は左側に，大きな値は右側に接続されるようになる.



二分木を使って探索

例：68を探す.

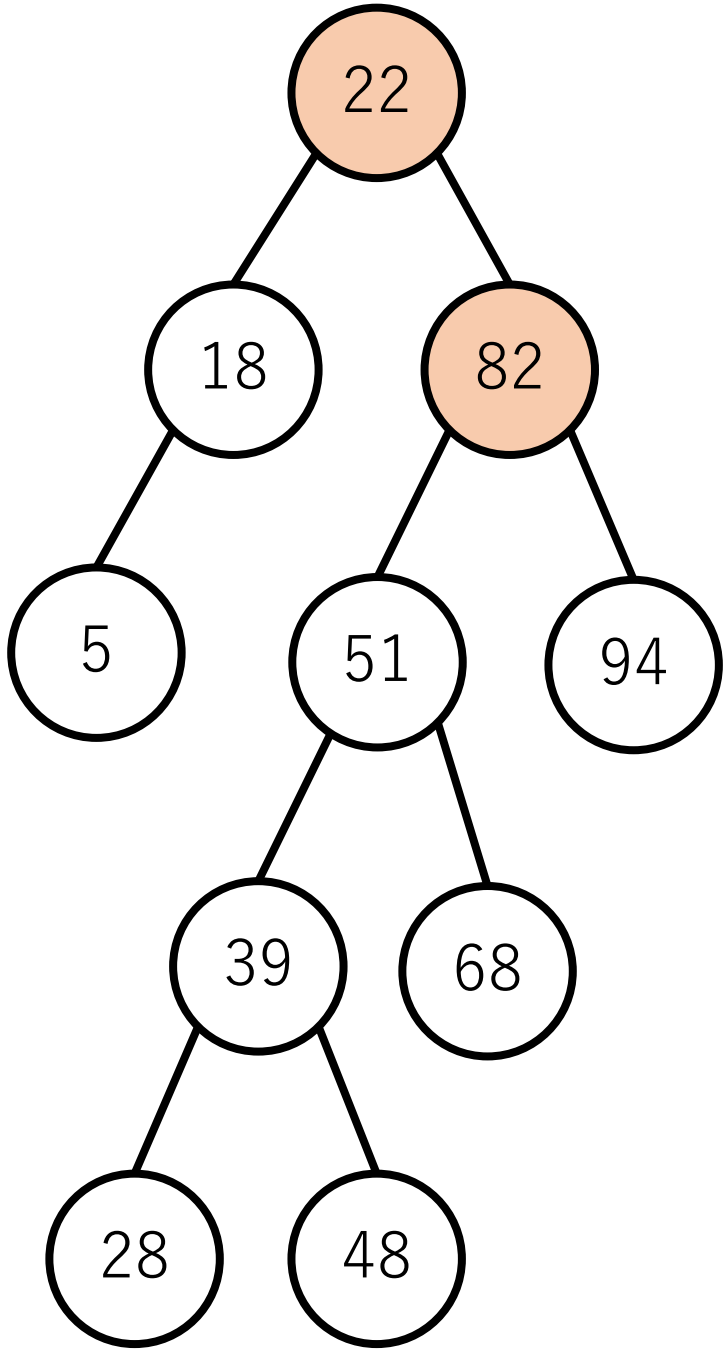
根ノードからスタート．68は22より
大きいので，右の子ノードに移動.



二分木を使って探索

例：68を探す.

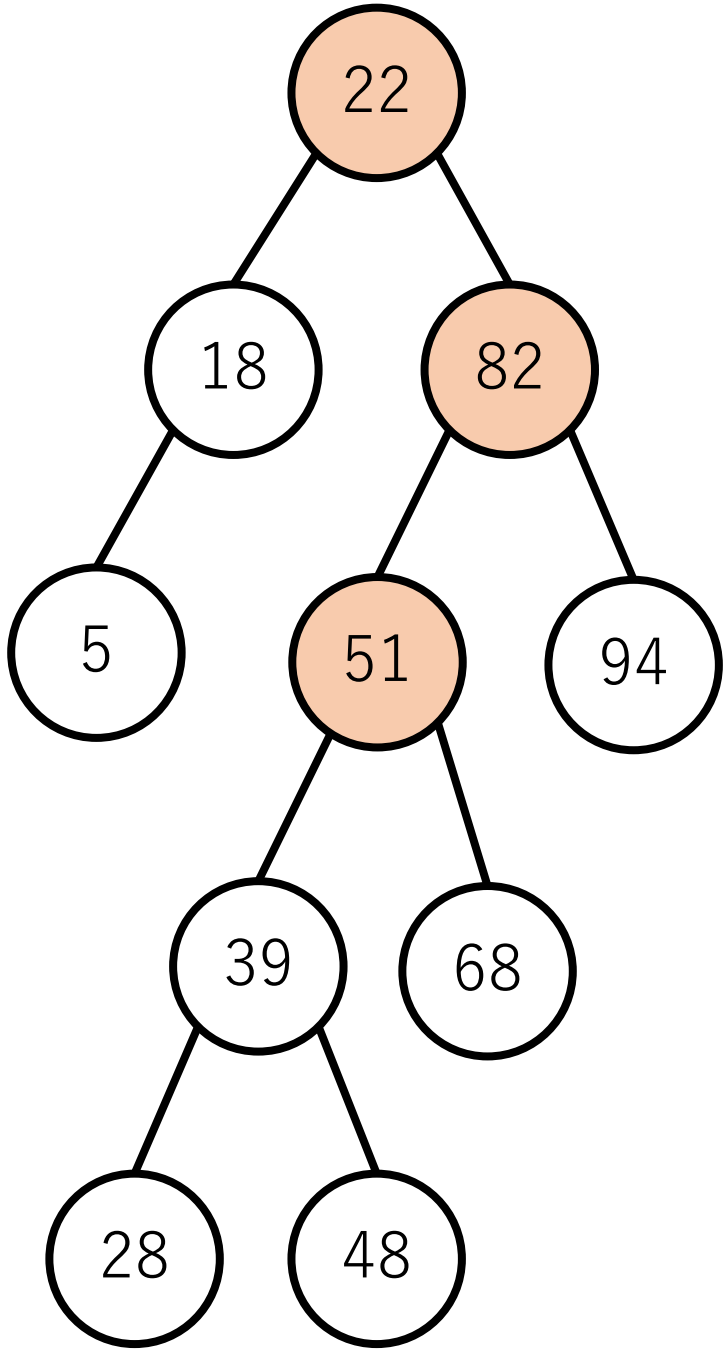
68は82より小さいので，左の子ノードに移動.



二分木を使って探索

例：68を探す.

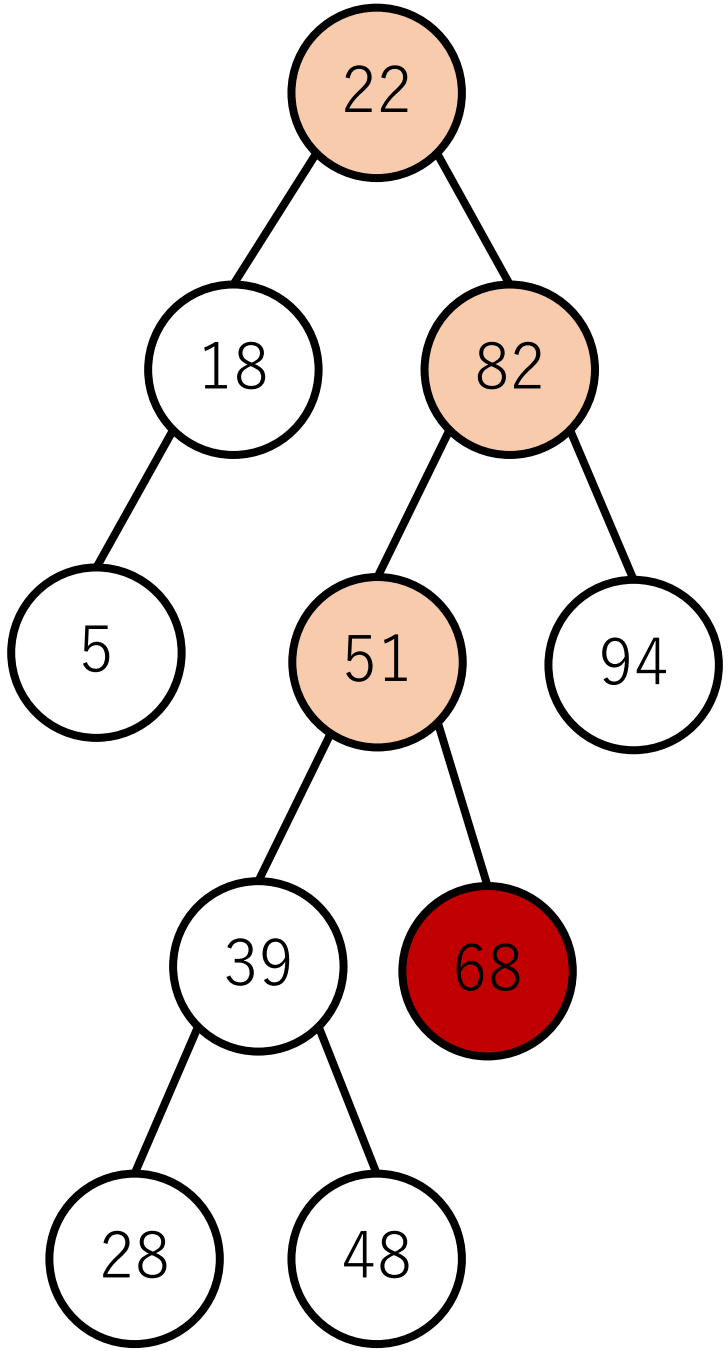
68は51より大きいので，今度は
右の子ノードに移動.



二分木を使って探索

例：68を探す.

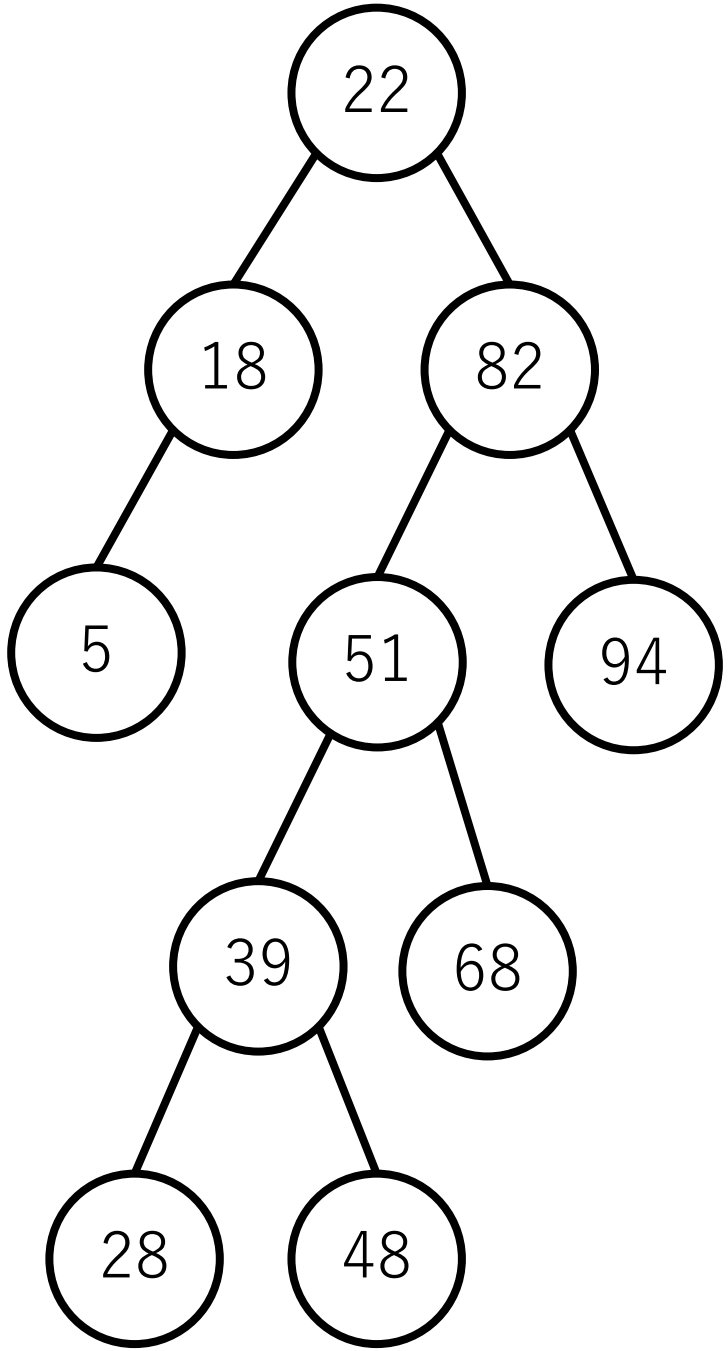
68を発見！



二分木を使った探索の計算量

挿入：

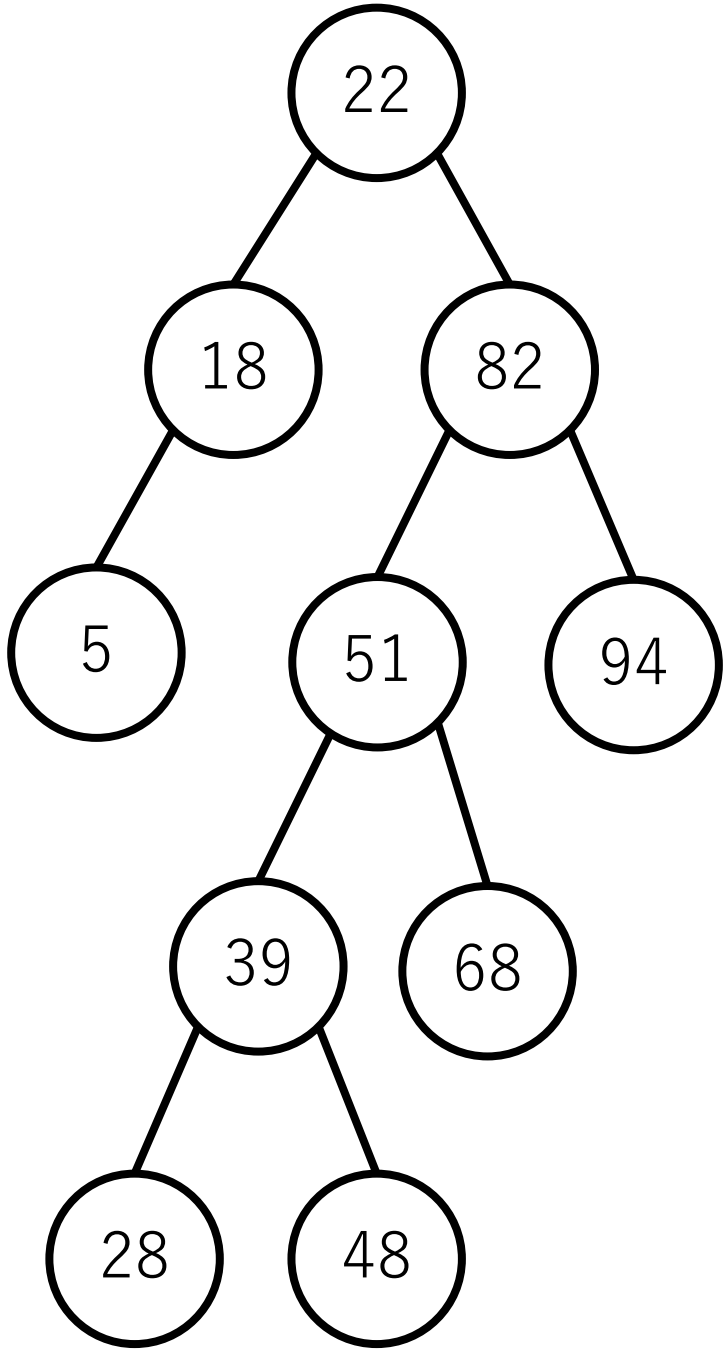
木が「普通の形」であれば，高さ$\log n$回比較をしたあとで追加できると期待できるので，$O(\log n)$



二分木を使った探索の計算量

探索：

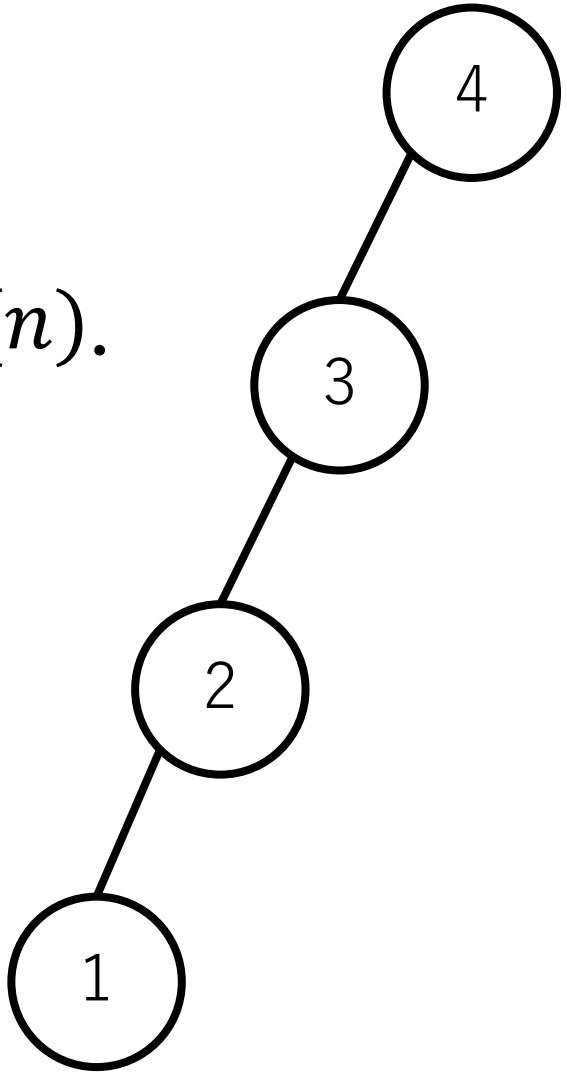
木が「普通の形」であれば，高さ$\log n$回辿れば結果がでると期待できるので，
$O(\log n)$.



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．

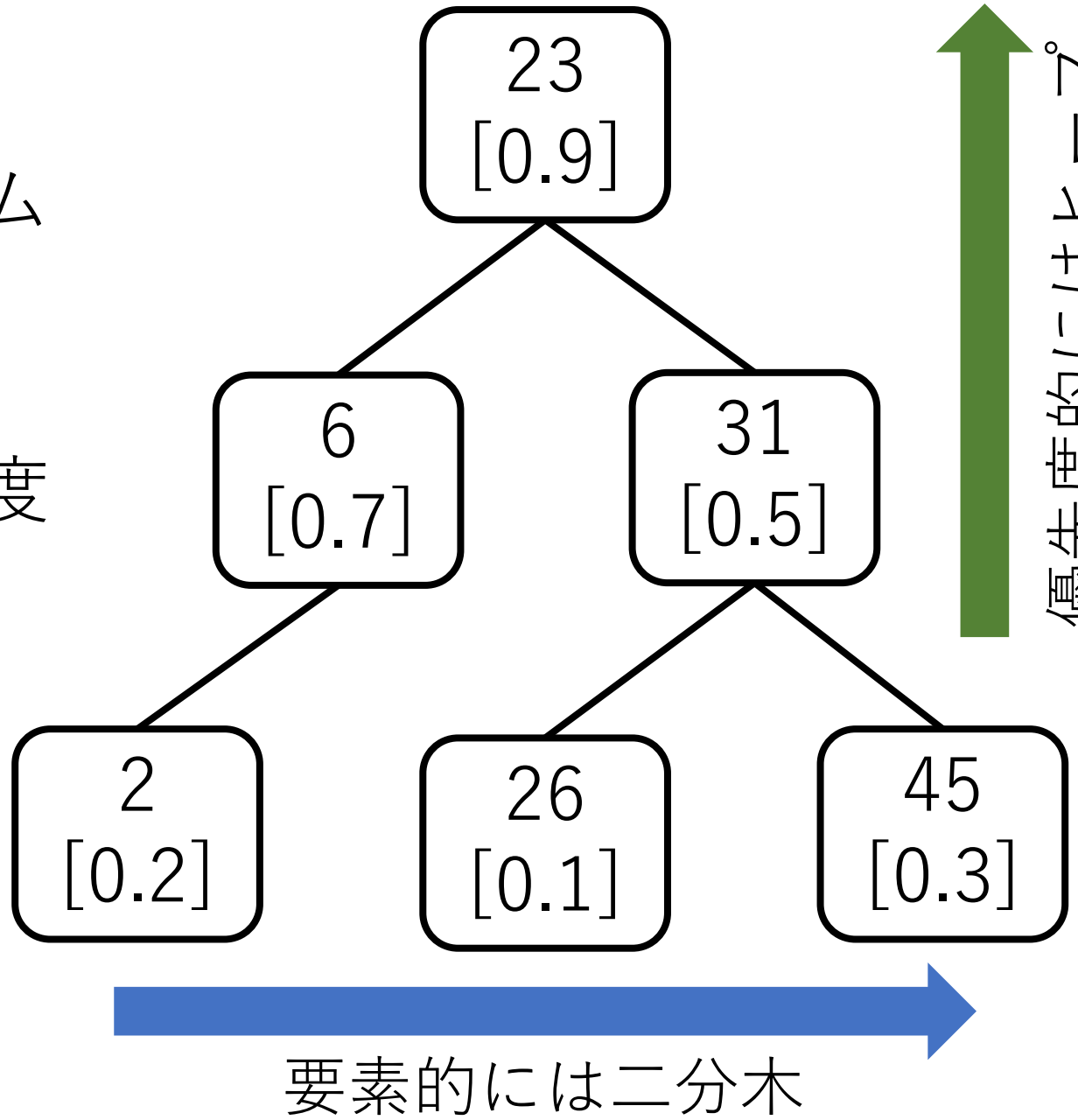
このランダム性を導入できないか？

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.



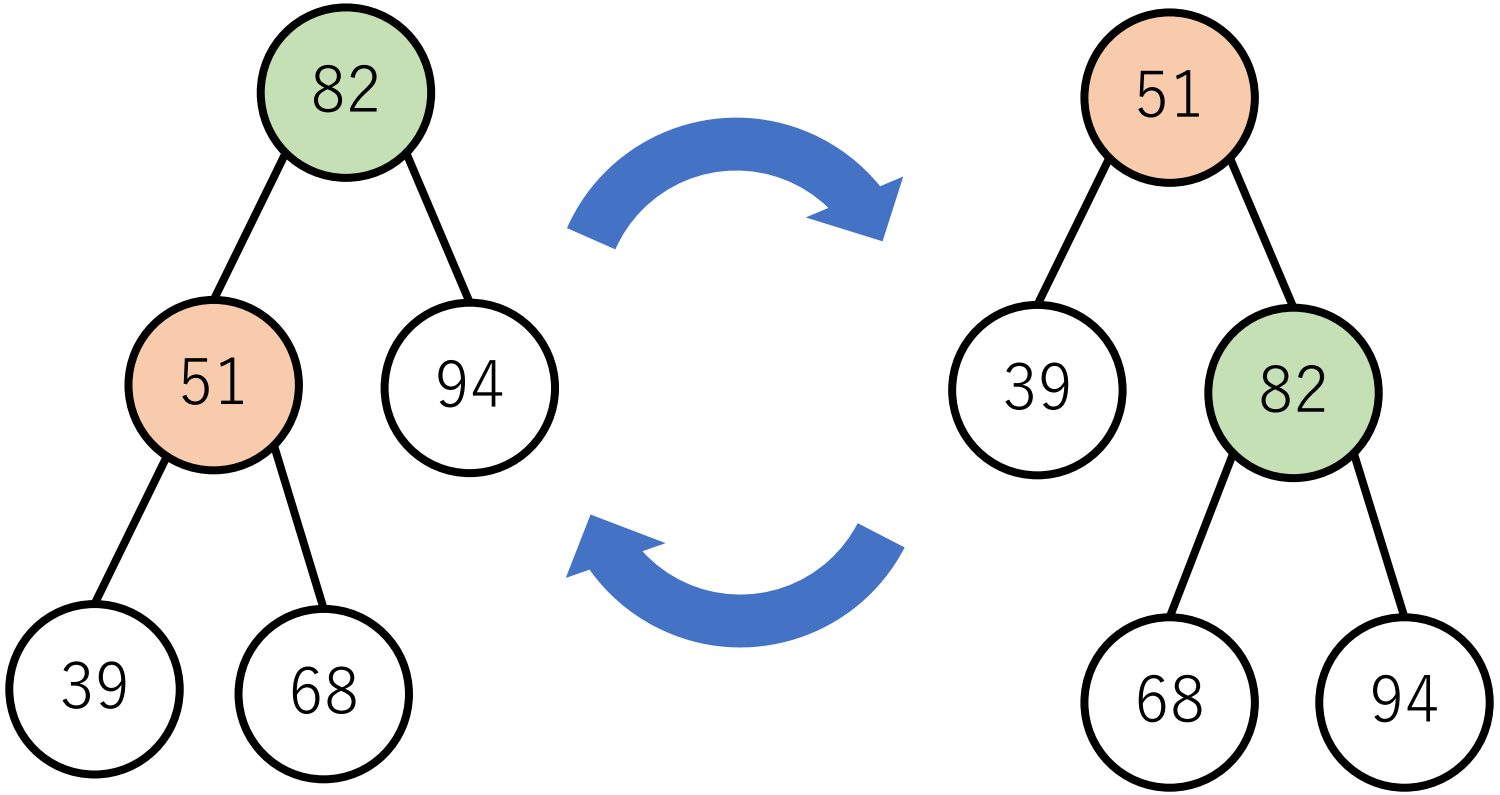
もう1つの方策

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）.

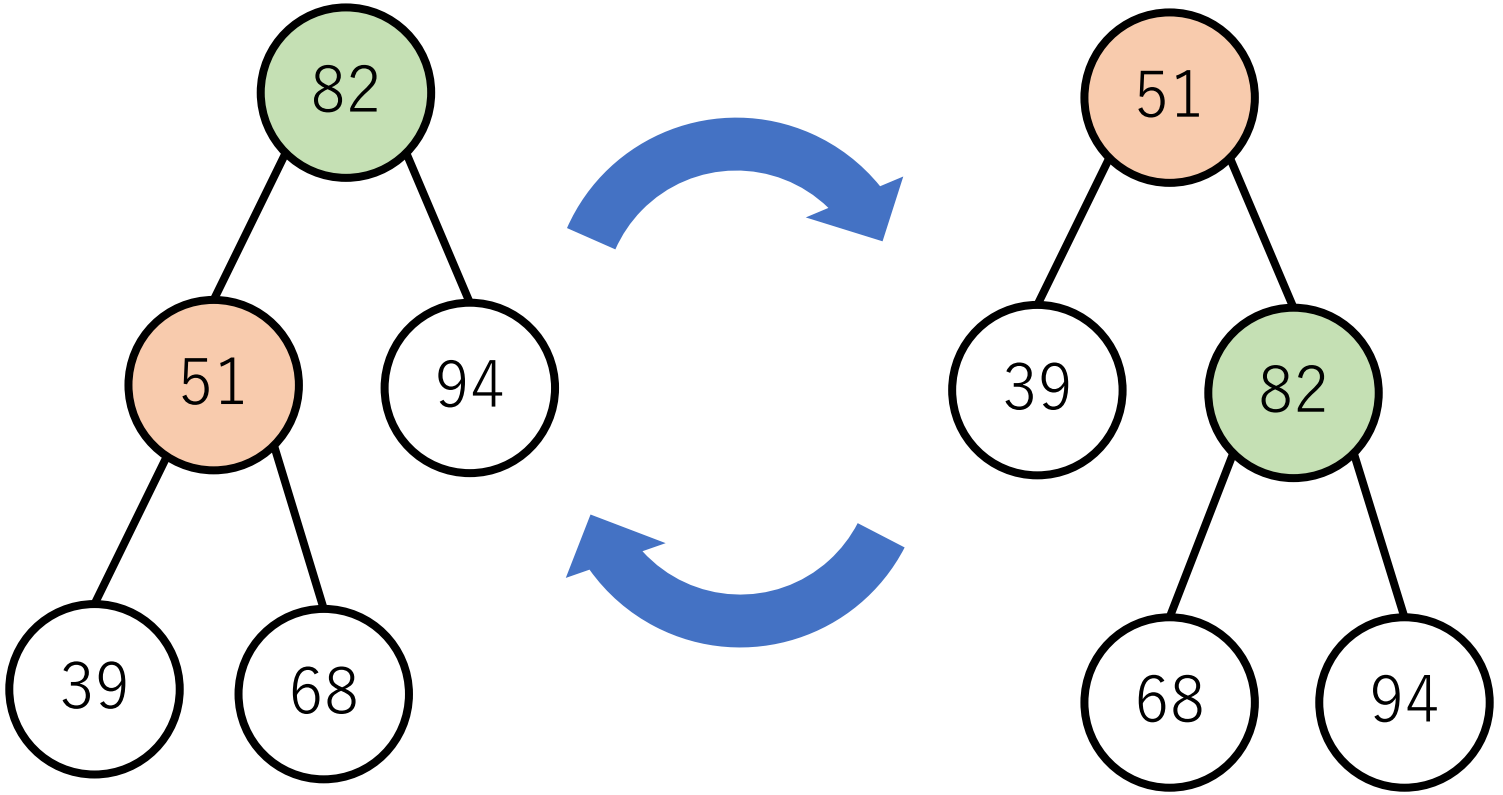
木の回転

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）

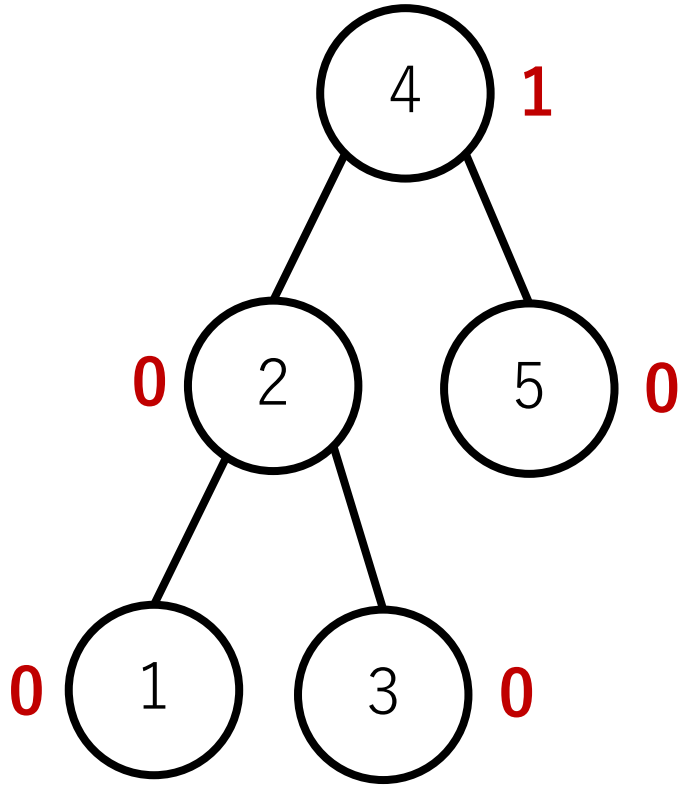
balance factorを各ノードに定義し，この値を見てどんな回転をするかを決める．

[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.

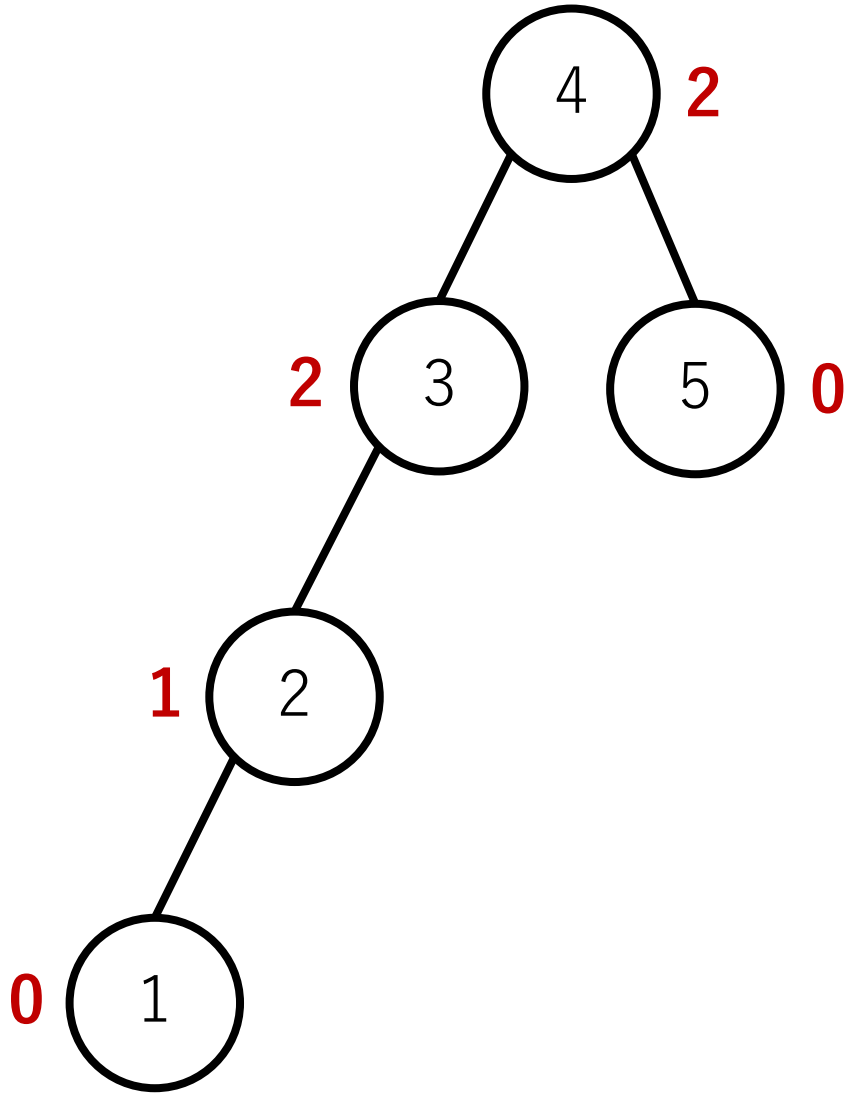
これは回転が必要ない．（全ての部分木の左右の高さの差が高々1）



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.

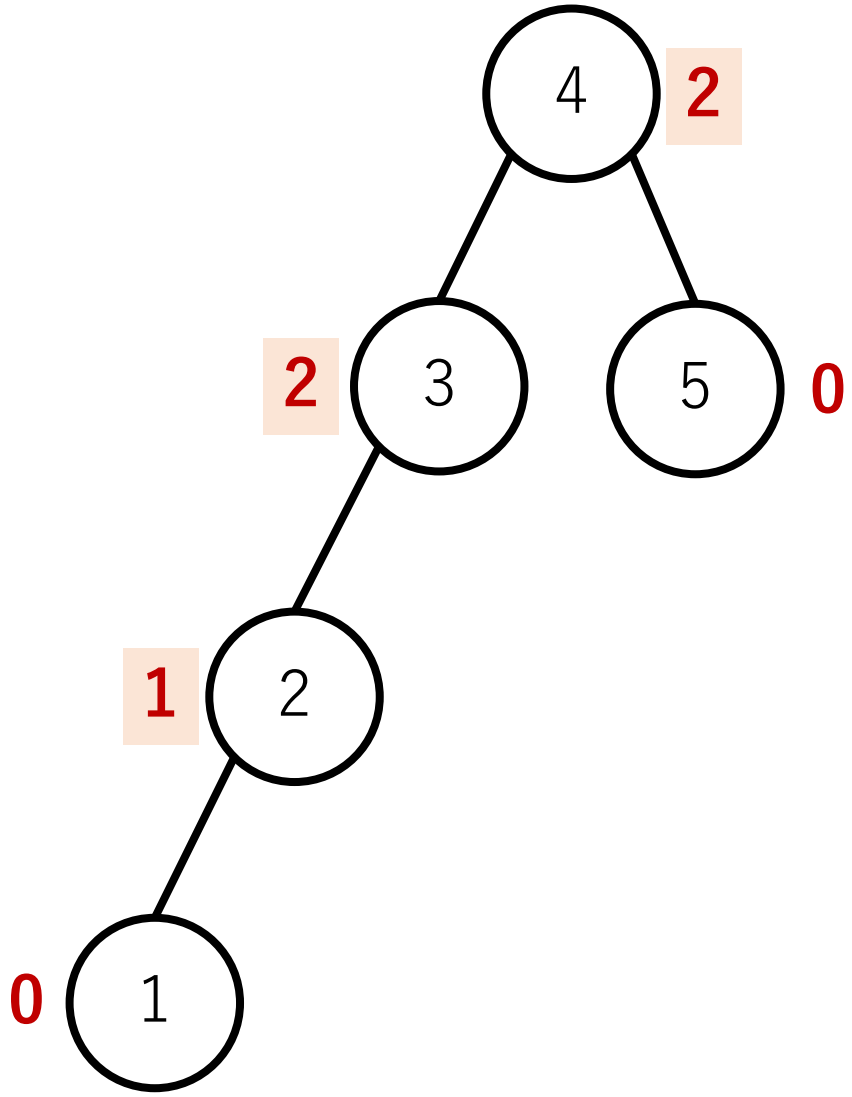
この木の場合は根ノードの下の左右の部分木の高さの差が2になっている.



AVL木の例

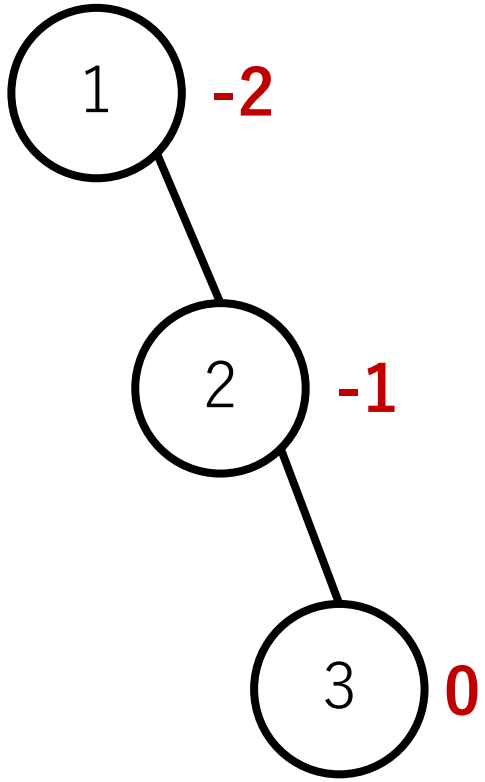
balance factorが各ノードに付与されている．このbalance factorの絶対値が2以上になったら回転が必要な合図.

よって，この場合は何かしらの回転を施したい.



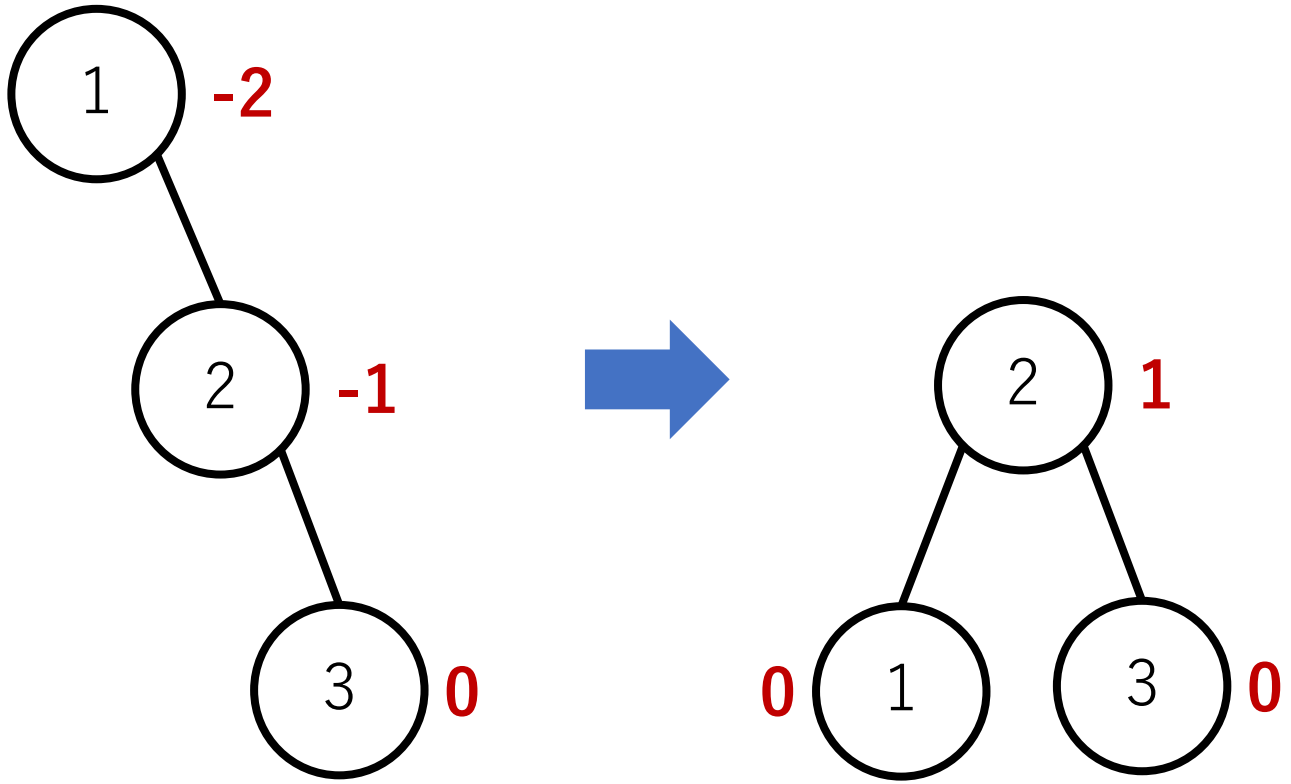
AVL木における回転

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



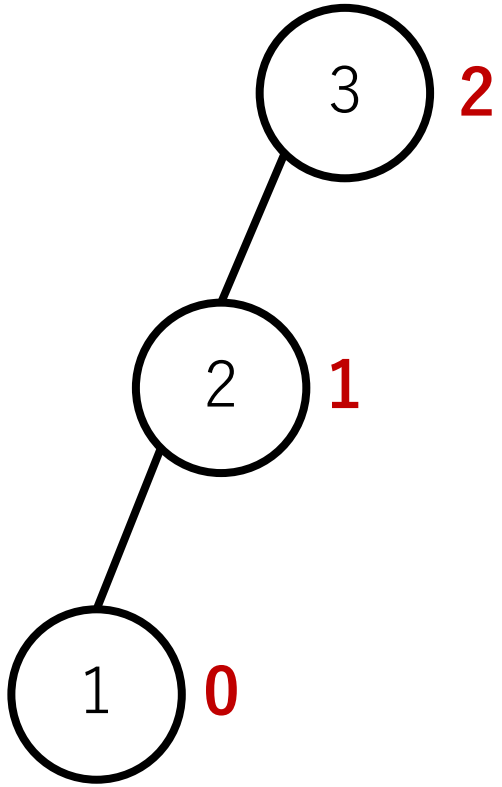
AVL木における回転

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



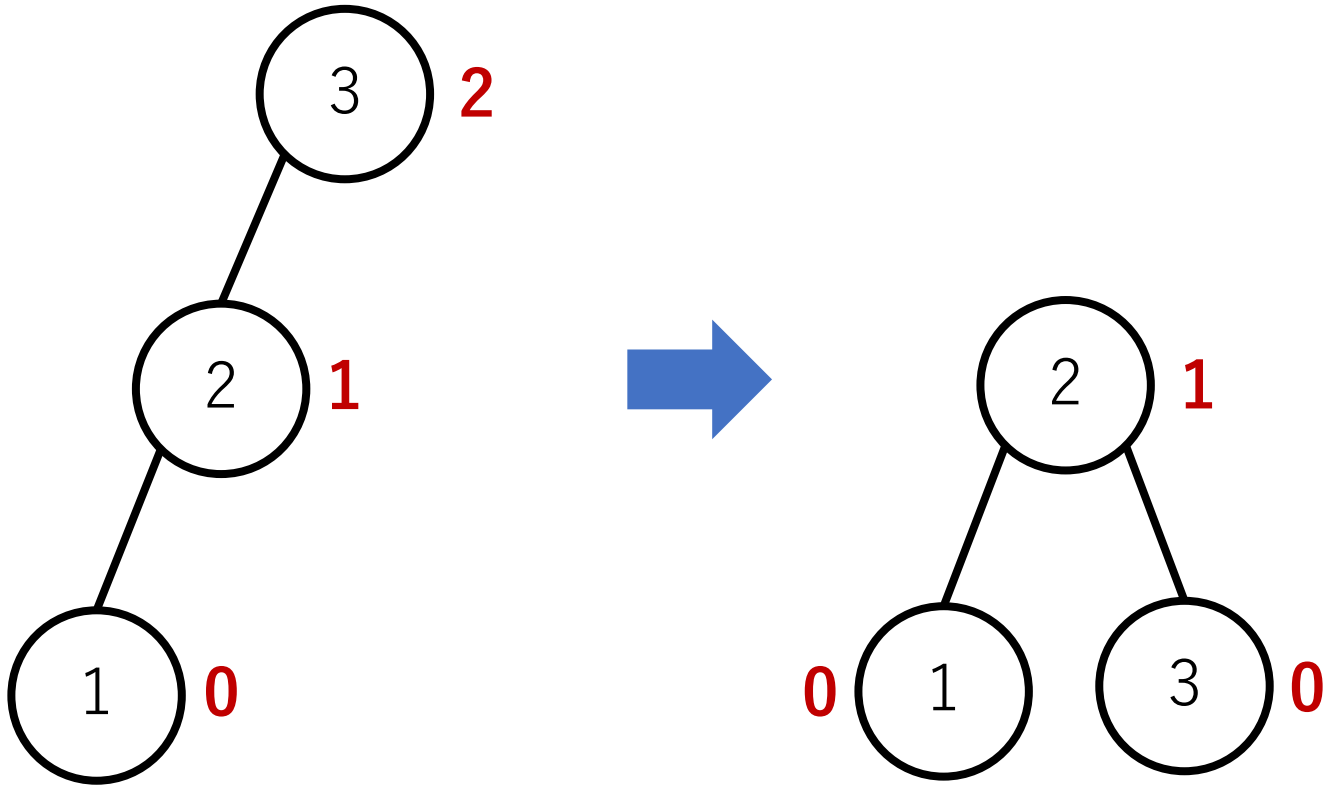
AVL木における回転

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



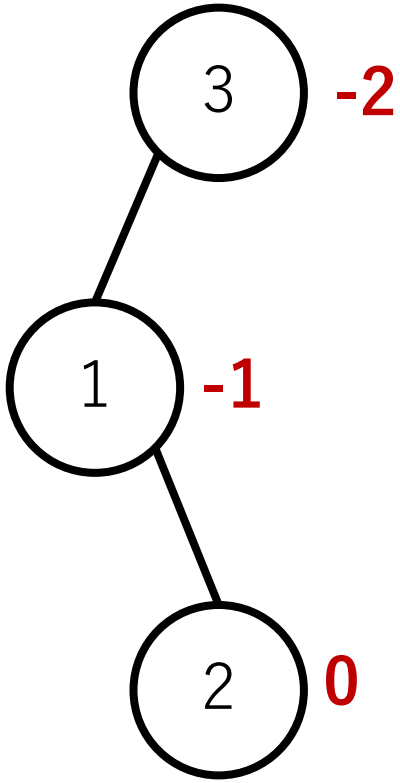
AVL木における回転

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



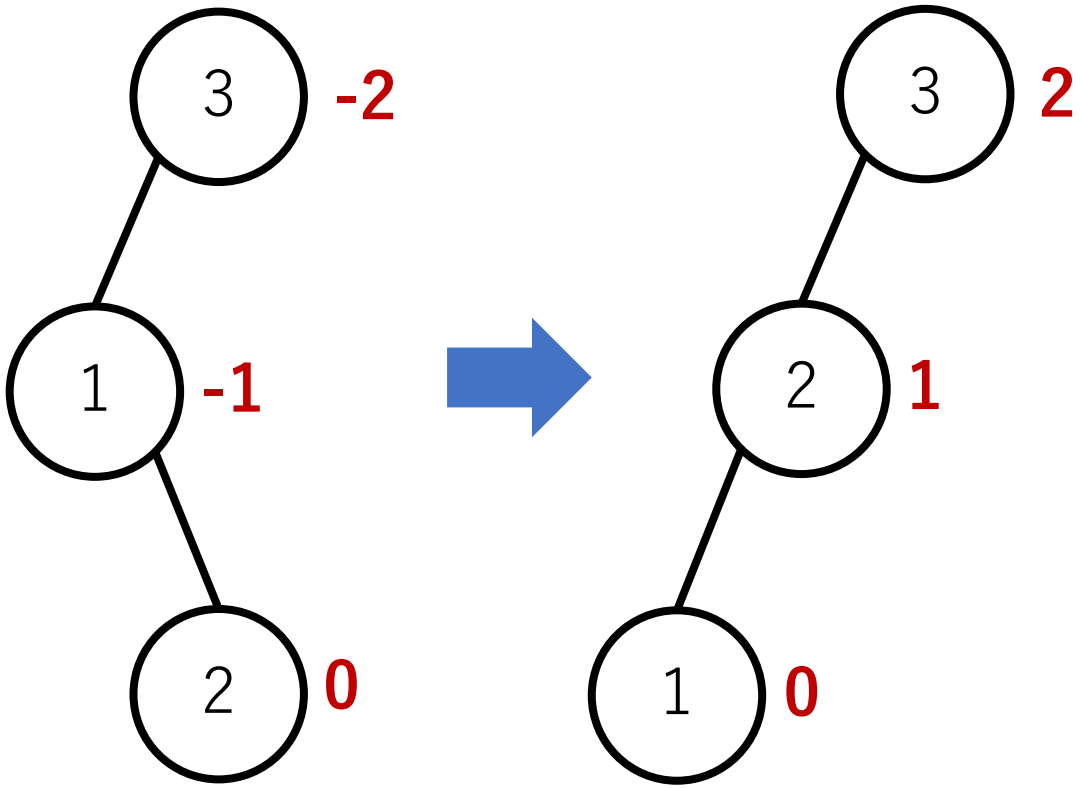
AVL木における回転

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



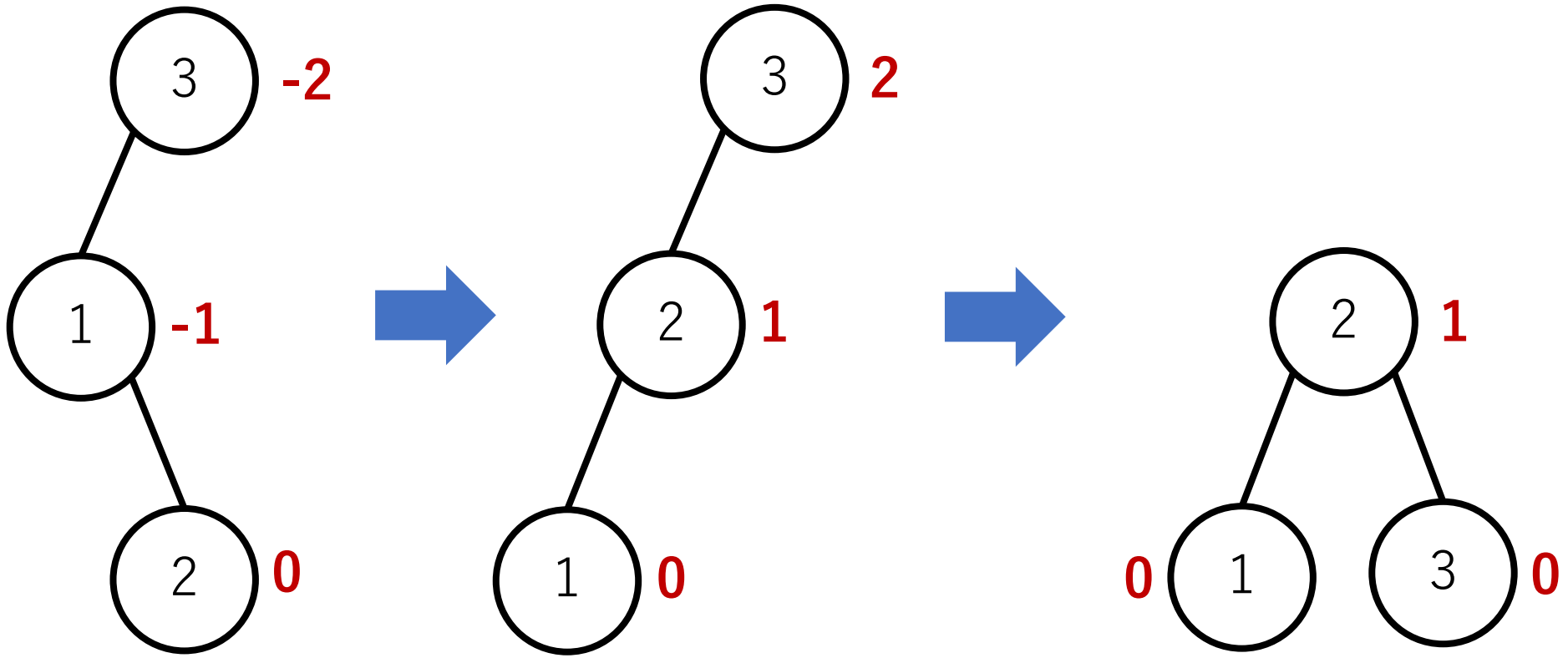
AVL木における回転

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



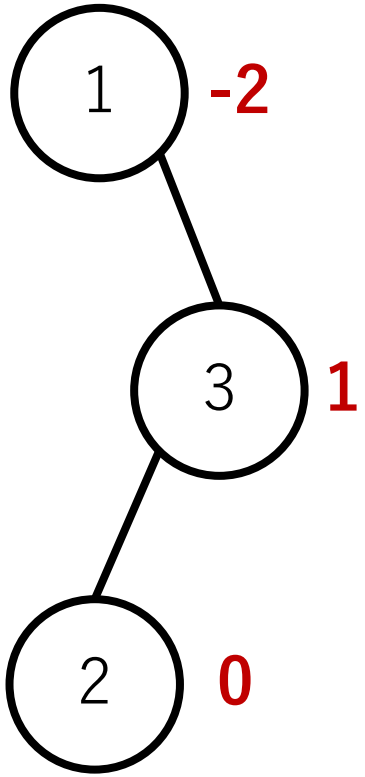
AVL木における回転

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



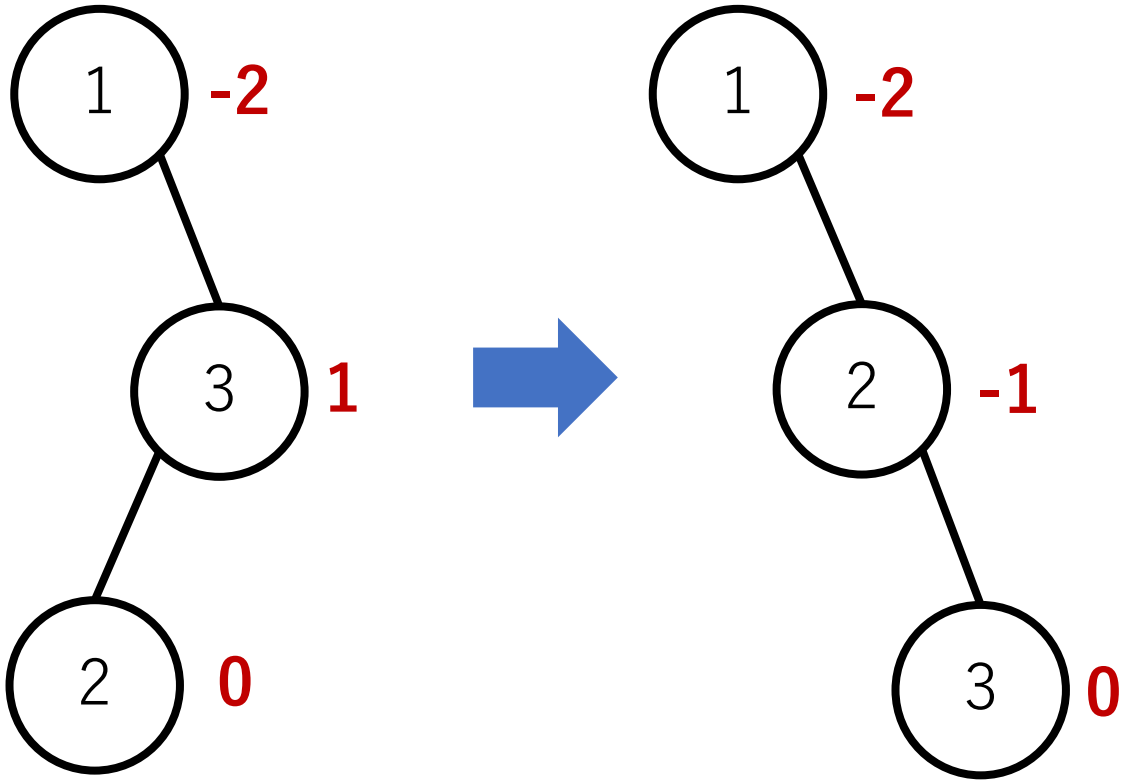
AVL木における回転

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



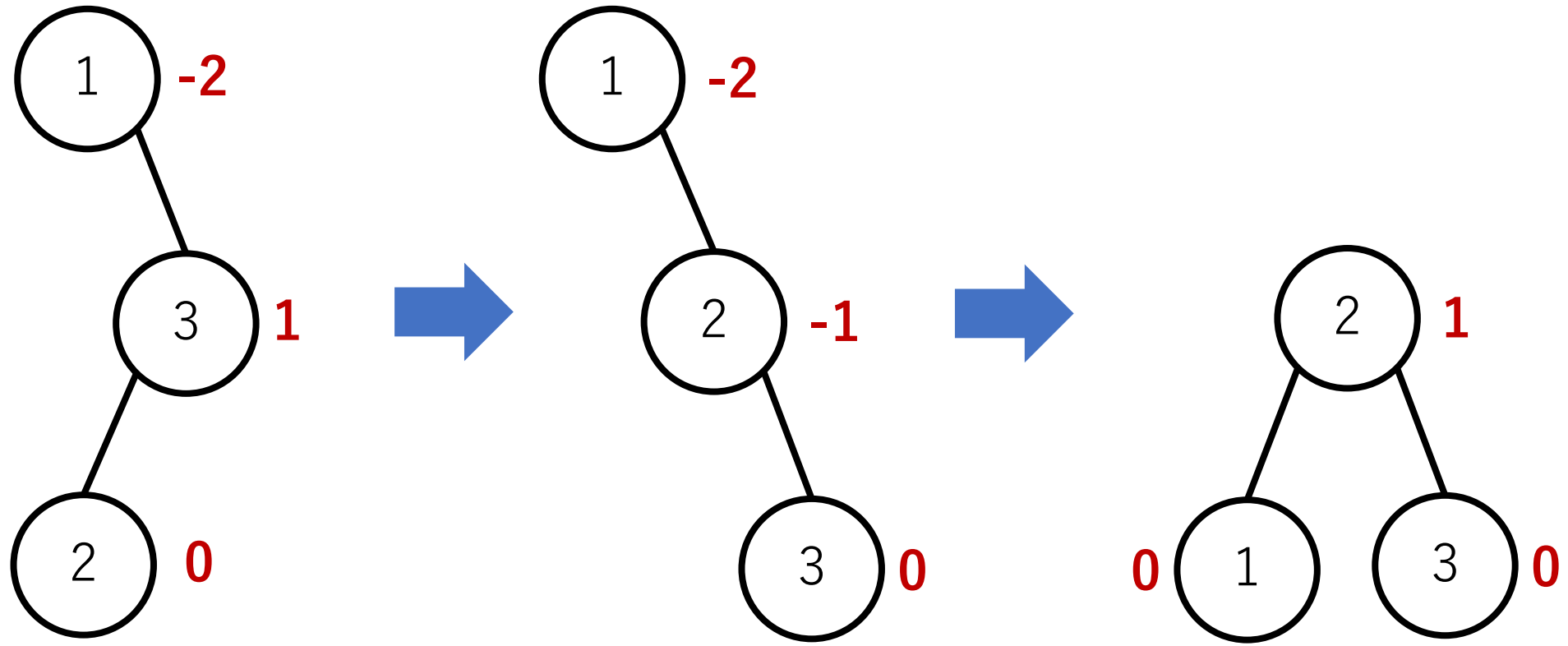
AVL木における回転

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



B木

多分木化を取り入れてバランス化をはかる.

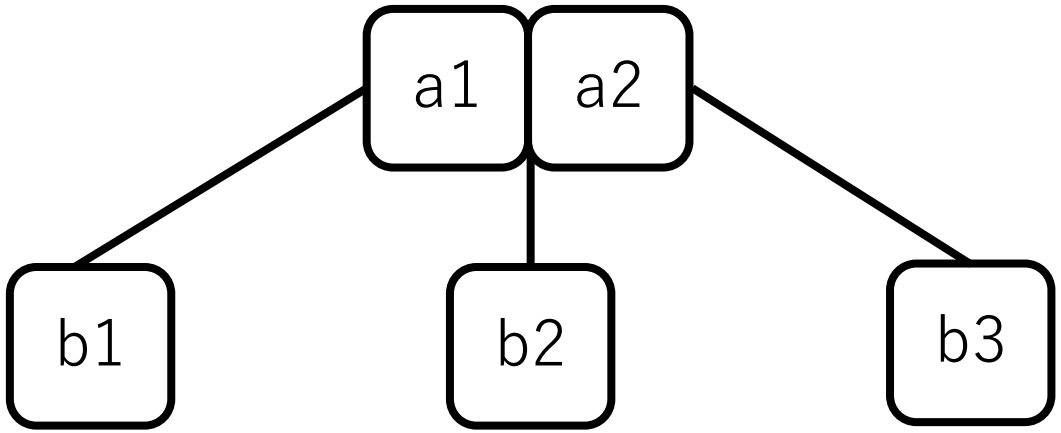
各ノードでの子ノードへの枝の数を最大mまで許す.
これをm次のB木（オーダーmのB木）という.

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

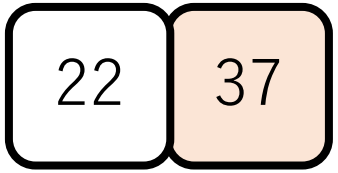
例：22, 37, 17, 9, 45, 34, 18

2-3木の例

例：22, 37, 17, 9, 45, 34, 18

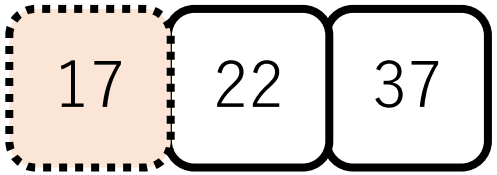
2-3木の例

例：22, 37, 17, 9, 45, 34, 18



2-3木の例

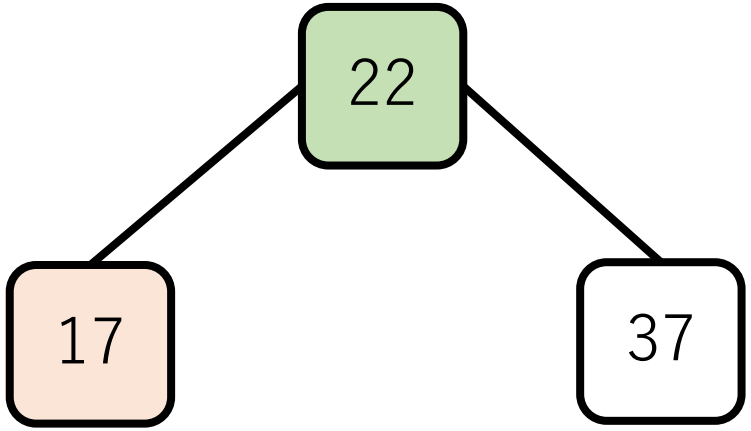
例：22, 37, 17, 9, 45, 34, 18



ここには入れない．

2-3木の例

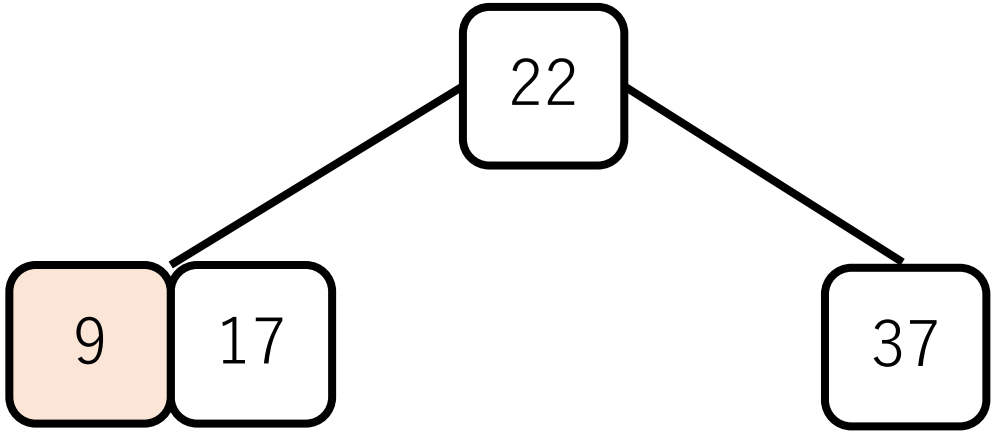
例：22, 37, 17, 9, 45, 34, 18



中央の値を親ノードにして，
残りの2つを分割する．

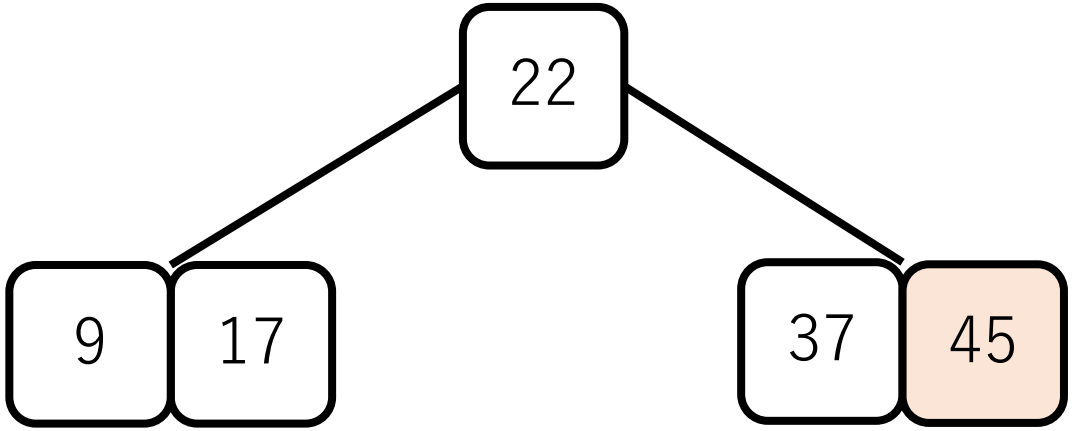
2-3木の例

例：22, 37, 17, 9, 45, 34, 18



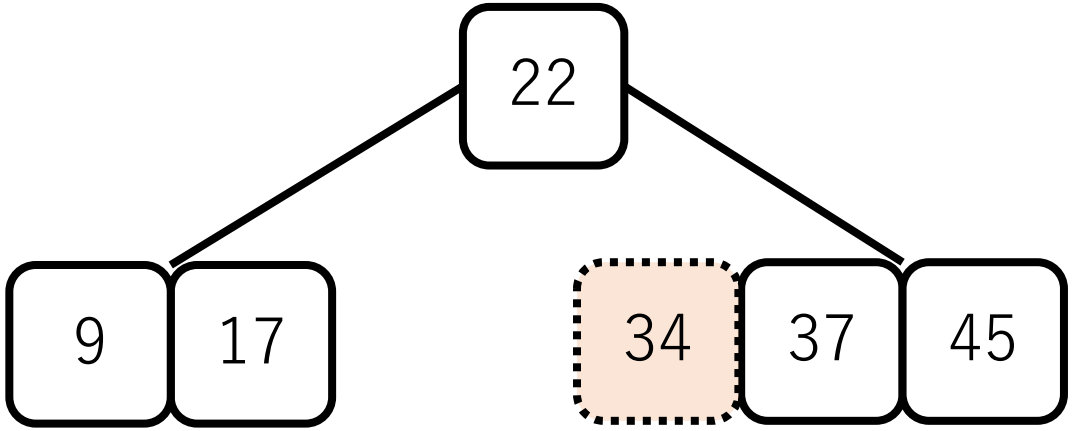
2-3木の例

例：22, 37, 17, 9, 45, 34, 18



2-3木の例

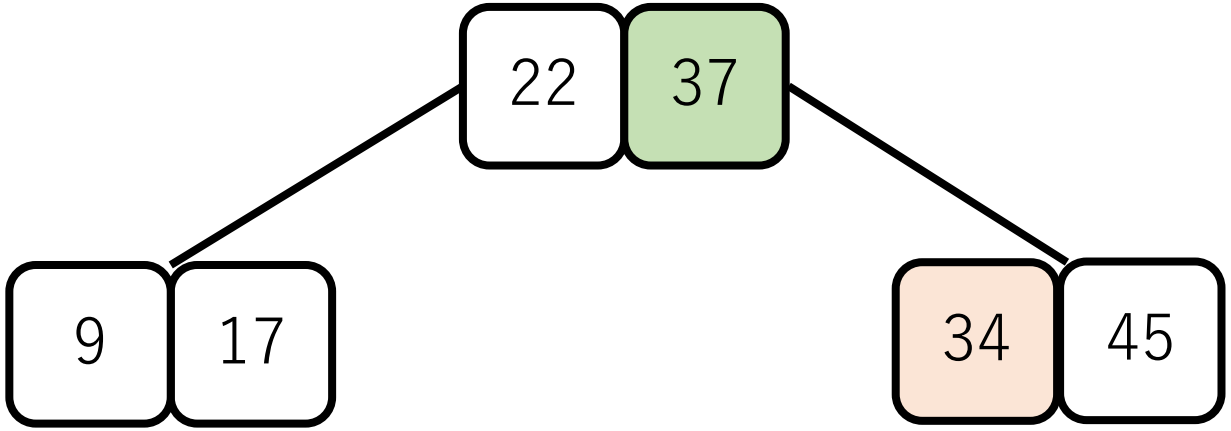
例：22, 37, 17, 9, 45, 34, 18



ここには入れない.

2-3木の例

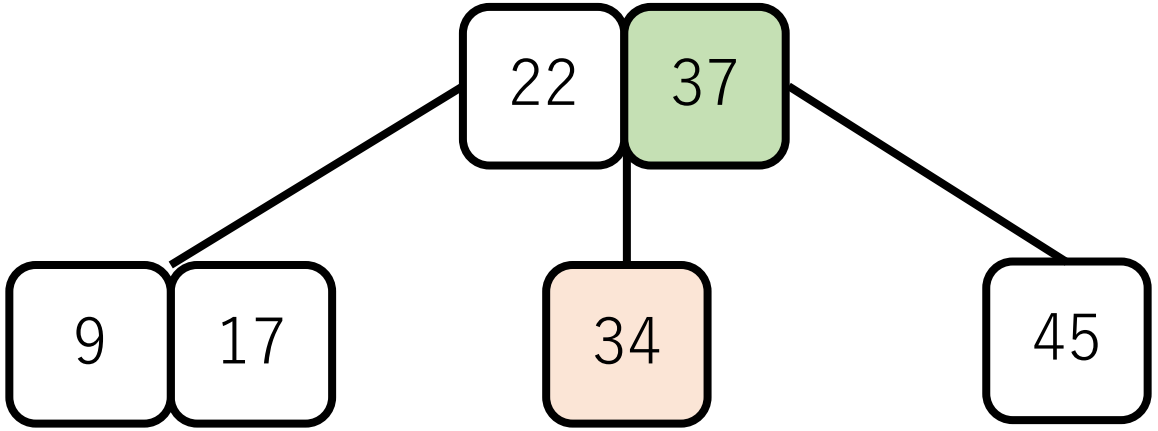
例：22, 37, 17, 9, 45, 34, 18



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

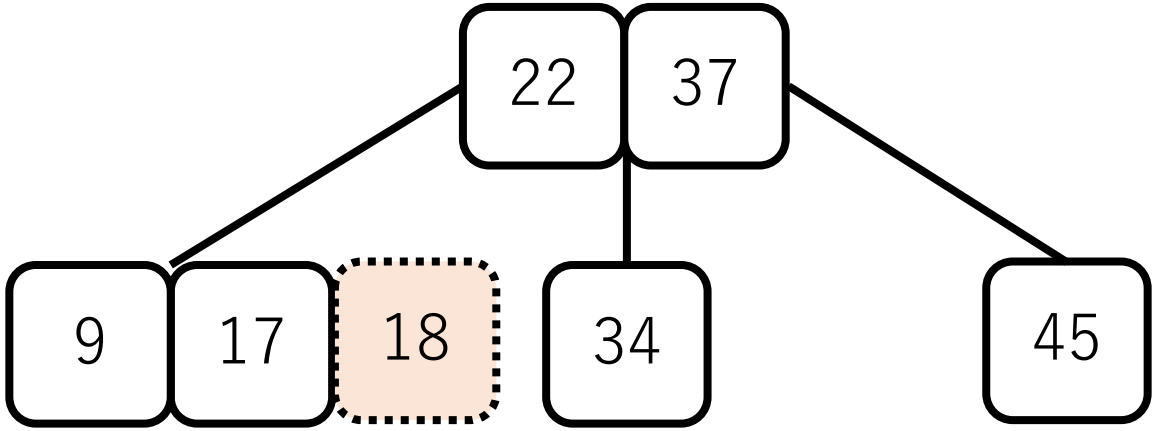
例：22, 37, 17, 9, 45, 34, 18



そこで，34と45を分割する．

2-3木の例

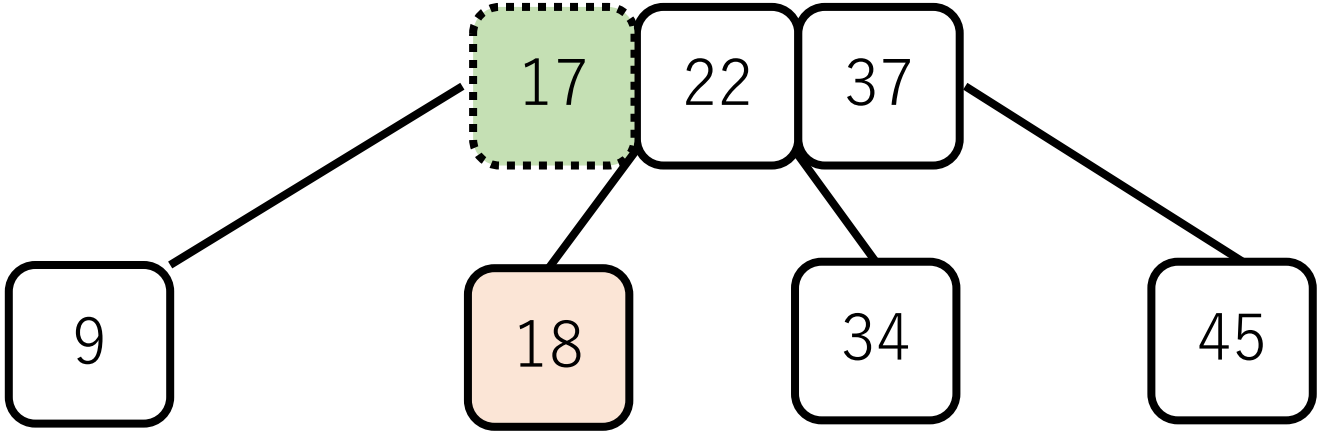
例：22, 37, 17, 9, 45, 34, 18



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

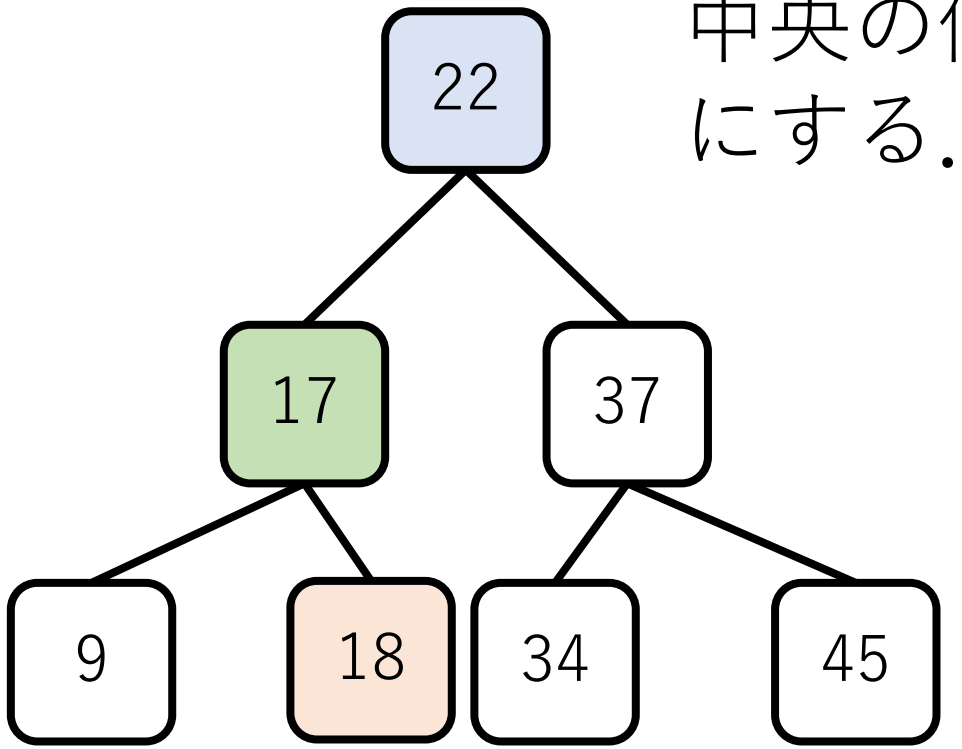
例：22, 37, 17, 9, 45, 34, 18



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

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．

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#3-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.

コードチャレンジ：基本課題#3-b [2点]

昇順にソートされている配列とキーが与えられた時,
二分探索を行うコードを書いてください.

コードチャレンジ：Extra課題#3 [3点]

目標とするクイズ正解率をピッタリ達成するために,
必要な最小の問題数を求める問題.