Algorithms (2020 Summer)
#2：累積和，データ構造

矢谷 浩司

前回のQ&A

「「解答を完了」したあと、「試験を提出する」を押さないと、採点されなかったりますか？前回は「試験を提出する」を押し忘れたようなきがするのですが」

→自動で提出はされますが，押してもらうほうが確実です．

「デフォルトで書いてあるコードは、特に気にしなくていいものですか？それとも理解しないといけない内容が書かれていますか？」

→気にしなくて良いです．

前回のQ&A

「Python入門を復習しようと考えています。無料で利用できるコンテンツでオススメのものがあればご教示ください。」

→いろんなオンラインチュートリアルがありますので，試してみてください.

https://www.python.ambitious-engineer.com/introduction-index#i-4

https://www.python-izm.com/introduction/

https://docs.python.org/3/

https://www.learnpython.org/

前回のQ&A

「聞き逃したかもしれないのですが、採点結果はITC-LMSなどで見れますか？」

→採点は[配点]*[テストケース合格率]が原則ですので，それで自分の点数がわかります．採点を取り消すケースなどが発生する場合は個別に連絡します．

前回のQ&A

「受講者フォームの単位習得の項目で「聴講のみ」を選択したのですが，この回答を撤回して単位習得をするために行わなければならないことはありますか？(UTASでの履修登録をするだけで大丈夫でしょうか)」

→今後のコードチャレンジで「単位取得」の方を選んでおいてください.

「課題を終えるまでにかかった時間は評価に入りますか？」

→考慮しません.

前回のQ&A

「採点のフィードバックはありますか」

→システム上，私達からの個別のフィードバックが難しいので，テストケースに通ったかどうかが一応のフィードバックになるようにしてます．

「前回以前に自分が提出したコードを確認することはできますか?」

→「試験を提出する」を押してしまうとあとからは見れませんので，予めローカルにコピーしておくとよいかと思います．

前回のQ&A

「後から復習できるようにtrack上の問題文をローカルに保存する方法はありますか？」

→あとから提出したものを見ることが出来る画面のURLを配布予定です.

「翌週に模範解答的なものは公開されますか」

→上と合わせて，その予定です.

前回のQ&A

「スライドとは違うアルゴリズムで提出してしまったのですが減点等はありますか？」

→スライドや問題文の指定と違うアルゴリズムで提出した場合は，テストケースに通っていても今後は採点されないことがありますので，問題文の指示をよくお読みください．

「コードを初めからやり直したいときにリセットする方法はありますか？」

→右上の歯車のところにテスト環境を初期化というボタンがあります．

前回のQ&A

「課題についてWEB上のコードを参照して、そっくりじゃなくても似ているコードはいけませんか？」

→自分自身で咀嚼した上で書いたコードだと判断してよいと第三者が見て言えるレベルにしてもらえればと思います.

「dequeやheapqなどは使用しない方が良いということでしょうか？」

→基本課題ではその類のものは使用できません.

前回のQ&A

「数学2Dの演習の授業も同じ曜限に開講されていて、
どちらも履修したいのですがそれは可能でしょうか？」

→残念ながら認められません．EEICでは情報よりの
興味をお持ちであれば「数学2G＋アルゴリズム」を,
物理よりの興味であれば「数学2D (講義+演習)」を
取るように授業が設計されていますので，そのよう
にしてください.

Crush Your Coding Interview with Facebook

エンジニアの採用試験で取り入れられている，コーディング面接がどんなものか，またそれを乗り越えるためにはどんなことを知っておくべきか，facebookのエンジニアが直接皆さんに紹介してくださいます！

海外の大学では多く行われているこのイベントを，今回はこの授業の中で特別に行っていただけることになりました！

Crush Your Coding Interview

Tips on the other side

FACEBOOK

JOIN US!

Never experienced a technical interview before? Want to extra-prepare for your next one? Facebook is excited to be partnering with University of Tokyo to get students up to speed about interviewing tips and the best way to show your strengths on a whiteboard.

We'll see you virtually on 15 July at 1.00pm!

Technical interviews are one of the major hurdles of landing an awesome internship or full time career. Come learn about the common pitfalls and tips to help you crush your next coding interview! You will also get a chance to ask our engineers questions and learn more about their experience.

Computer Science/ Engineering major focused. Open to all students.

All class levels are welcome to attend! Be sure to come with lots of questions.

Limited seats available!

Crush Your Coding Interview with Facebook

7/15（水）13:00～14:45

このワークショップはzoom上にて英語で開催されます.

ワークショップへの参加はこの講義の成績には関係しません.
また，この講義を履修していない方でも参加可能ですので,
ぜひご友人をお誘いの上，お越しください！

累積和とは？

前回の基本課題: 累積和の典型問題

ランダムな整数が格納されている長さNの配列の中で,
m個の隣接する要素の和が最大となる部分を1つ求めよ.

最もナイーブな解き方で求めた.

前回の基本課題でやったこと

[m: 今までの部分和の最大値を入れる]
[m_index: 今までの部分和で最大になる最初のインデックス]

```
for i: 0からN-mまで:
    tmp = 0
    for j: 0からm-1まで:
        tmp += sequence[i+j]
    if tmp > m:
        [mとm_indexを更新]
```

前回の基本課題でやったこと

```
for i: 0からN-mまで:
    tmp = 0
    for j: 0からm-1まで:
        tmp += sequence[i+j]
    if tmp > m:
        [mとm_indexを更新]
```

$N \gg m \gg 1$なら，$O(Nm)$.　（より厳密には$O(m(N-m))$）
N, mがそれなりに大きいと結構大変.

ナイーブな方法

毎回 sequence[i] から sequence[i+m-1] まで足し合わせているのが無駄.

その次sequence[i+1] から sequence[i+m]まで計算することになるが，変更があるのは最初と最後だけ.

よってその差分だけ計算するようにすれば無駄を大きく削減できる！

改良版

```
tmp = [0からmまでのsequenceの部分和]
m = tmp
m_index = 0

for i: 1からN-mまで:
	tmp = tmp - sequence[i] + sequence[m+i]
	if tmp > m:
		[mとm_indexを更新]
```

改良版

新しい部分和を計算するところが$O(m)$から$O(1)$に.

よって，全体の計算量も$O(N)$！

パフォーマンスの比較

sequence=10,000,　m=100
（表の単位はmsec）

ナイーブ方式： $O(Nm)$
改良版： $O(N)$

ナイーブ方式	改良版
206	3.3
163	4.4
191	4.3
158	3.3
167	3.4
170	3.4
158	3.5
170	3.4
163	3.5
170	3.7

パフォーマンスの比較

sequence=100,000,　m=100
Nを10倍

ナイーブ方式：$O(Nm)$
改良版：$O(N)$

どちらもほぼ10倍になる．

ナイーブ方式	改良版
1,638	37
1,616	37
1,614	37
1,624	38
1,642	38
1,624	36
1,617	38
1,662	36
1,662	42
1,643	36

パフォーマンスの比較

sequence=100,000,　m=1,000
さらにmを10倍.

ナイーブ方式：$O(Nm)$
改良版：$O(N)$

ナイーブ方式には影響するが,
改良版には影響しない.

ナイーブ方式	改良版
17,676	37
19,122	51
20,810	61
19,720	41
20,130	40
20,147	41
18,543	46
20,981	38
20,167	40
19,770	38

累積和のその他の例

あるデータのうちAからBまでで該当するデータの数や和を問い合わせるクエリが大量に発生する，みたいなシナリオのときに有効.

例）「A月からB月までの総売上を問い合わせるクエリが大量に発生する．」

似たアルゴリズム：しゃくとり法

問題例「ランダムな整数が格納されている長さNの配列の中で，部分和がmになる連続した要素のうち，その長さが最小になるものを求めよ．」

先ほどの問題と違い，長さが可変長.

しゃくとり法の考え方

部分和を計算する左端，右端のindexを保持する変数を定義．それぞれ0からスタート．

右端のindexを1つずつ増やしながら部分和を計算．

決められた値（例の場合はm）を部分和が超えたら，
右端のindexを増やすのをやめる．

しゃくとり法の考え方

次に，左端のindexを進めながら部分和を更新（つまり最初の要素から順に部分和から引いていく）.

決められた値（例の場合はm）を部分和を下回ったら超えたら，左端のindexを増やすのをやめる.

また右端のindexを右に動かしていき，以降同様に繰り返す．ぴったりmになった時には現在の部分和を構成する連続するようその長さを比較し，最短なら記録しておく.

しゃくとり法

[3, 4, 9, 5, 1, 4, 6] から部分和が14になるものを1つ探す.

しゃくとり法

[3, 4, 9, 5, 1, 4, 6] から部分和が14になるものを1つ探す.

[3, 4, 9, 5, 1, 4, 6]: 部分和は3，13より下 -> 右端+1

しゃくとり法

[3, 4, 9, 5, 1, 4, 6] から部分和が14になるものを1つ探す.

[3, 4, 9, 5, 1, 4, 6]: 部分和は3,　13より下 -> 右端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は7,　13より下-> 右端+1

しゃくとり法

[3, 4, 9, 5, 1, 4, 6] から部分和が14になるものを1つ探す.

[3, 4, 9, 5, 1, 4, 6]: 部分和は3，13より下 -> 右端+1

[3, 4, 9, 5, 1, 4, 6]: 部分和は7，13より下-> 右端+1

[3, 4, 9, 5, 1, 4, 6]: 部分和は16，13より上-> 左端+1

しゃくとり法

[3, 4, 9, 5, 1, 4, 6] から部分和が14になるものを1つ探す.

[3, 4, 9, 5, 1, 4, 6]: 部分和は3，13より下 -> 右端+1

[3, 4, 9, 5, 1, 4, 6]: 部分和は7，13より下-> 右端+1

[3, 4, 9, 5, 1, 4, 6]: 部分和は16，13より上-> 左端+1

[3, 4, 9, 5, 1, 4, 6]: 部分和は13，14より下-> 右端+1

しゃくとり法

[3, 4, 9, 5, 1, 4, 6] から部分和が14になるものを1つ探す.

[3, 4, 9, 5, 1, 4, 6]: 部分和は3，13より下 -> 右端+1

[3, 4, 9, 5, 1, 4, 6]: 部分和は7，13より下-> 右端+1

[3, 4, 9, 5, 1, 4, 6]: 部分和は16，13より上-> 左端+1

[3, 4, 9, 5, 1, 4, 6]: 部分和は13，14より下-> 右端+1

[3, 4, 9, 5, 1, 4, 6]: 部分和は18，14より上-> 左端+1

しゃくとり法

[3, 4, 9, 5, 1, 4, 6] から部分和が14になるものを1つ探す.

[3, 4, 9, 5, 1, 4, 6]: 部分和は3, 13より下 -> 右端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は7, 13より下-> 右端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は16, 13より上-> 左端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は13, 14より下-> 右端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は18, 14より上-> 左端+1
[3, 4, 9, 5, 1, 4, 6]: 部分和は14, ぴったり！

しゃくとり法の前提

条件を満たしている区間の部分区間も条件を満たす.

例えば，[3, 4, 9, 5, 1, 4, 6]の場合，4も9も14以下なので，右端を拡張して，部分和を増やすしか探索の可能性がない.

単純に言えば，単調増加（減少）するようなデータ列である必要がある.

しゃくとり法の計算量

区間の選び方の総数は，$n(n+1)/2$.　（左端と右端が同じ場所であるものも含める）

よって，計算量は$O(n^2)$.

しゃくとり法では，左端も右端もn回しか動かず，かつ，部分和の更新は定数回.

よって，計算量は$O(n)$.　右端が後戻りしない分効率化できている.

データ構造

なぜデータ構造を考えるか？

単に保存しておいて，必要な度に計算すれば？

効率的にデータを取り出せることは計算量を削減する上で重要.

データ構造自体がある種の処理を内包できるので，追加の処理がいらない.

今日紹介するデータ構造

スタック

キュー

線形リスト

ツリー

ヒープ

スタック

上にどんどん積み重ねていく形でデータを保持する構造.

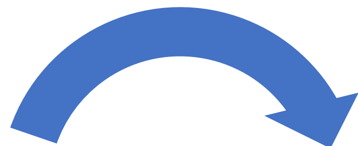
読むつもりの本を机の上に積んでいくような感じ.

上に積む（pushする）か，一番上を取る（popする）という操作でデータの出し入れをする．（スタックの世界では横から取り出す，ということはしない．）

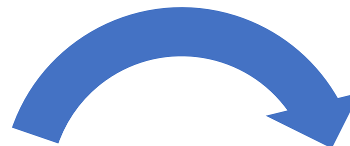
LIFO（last in, first out)

スタック

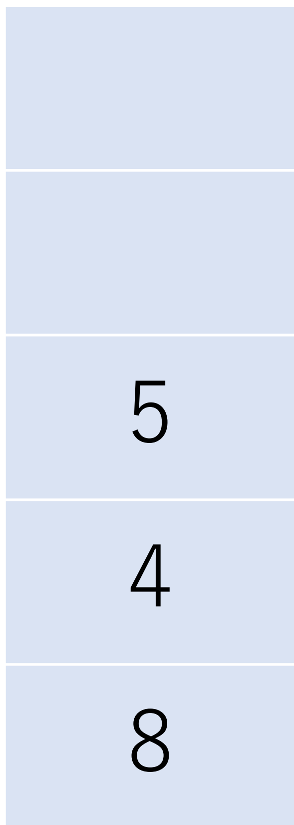
push(6)

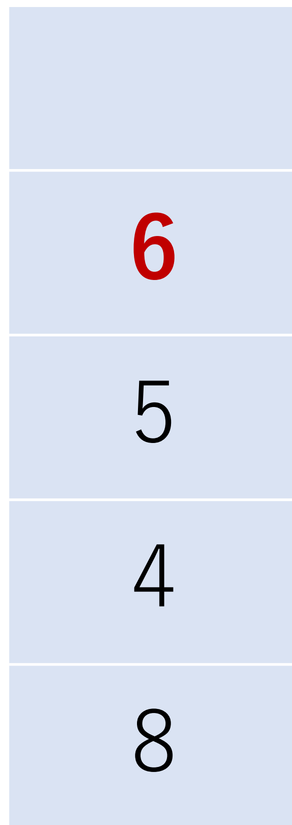


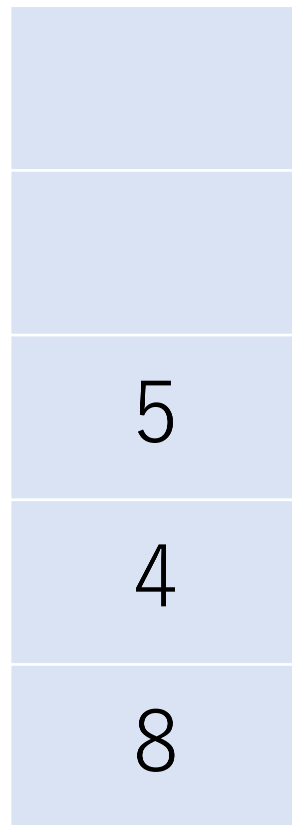
pop()



6







スタック

```
class Stack:
    def __init__(self, size: int):
        self.stack = [None]*size
        self.top = 0
```

スタック

```
class Stack:
    def push(self, a: int):
        if self.top < len(self.stack):
            self.stack[self.top] = a
            self.top += 1
            print(self.stack)
        else:
            print('This stack is full.')
```

スタック

```
class Stack:
    def pop(self):
        if self.top > 0:
            self.top -= 1
            tmp = self.stack[self.top]
            self.stack[self.top] = None  # なくてもよい.
            print(self.stack)
            return tmp
        else:
            print('This stack is empty.')
```

スタック

Code	Result
a = Stack(5)	
a.push(8)	[8, None, None, None, None]
a.push(4)	[8, 4, None, None, None]
a.push(5)	[8, 4, 5, None, None]
a.push(6)	[8, 4, 5, 6, None]
b = a.pop()	[8, 4, 5, None, None]
a.push(2)	[8, 4, 5, 2, None]
b	6

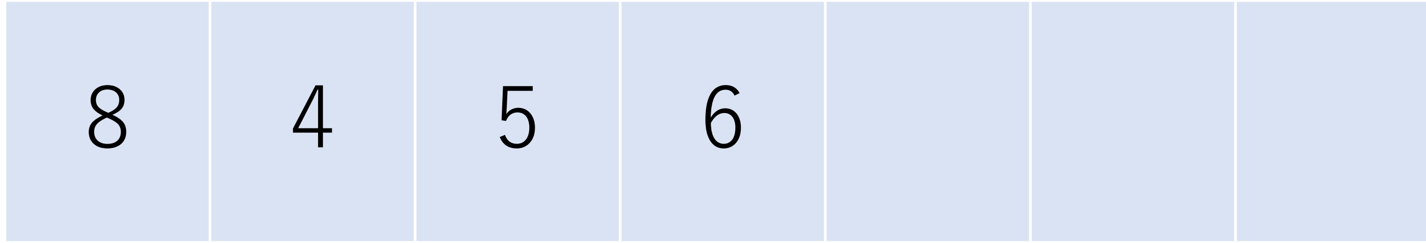
キュー

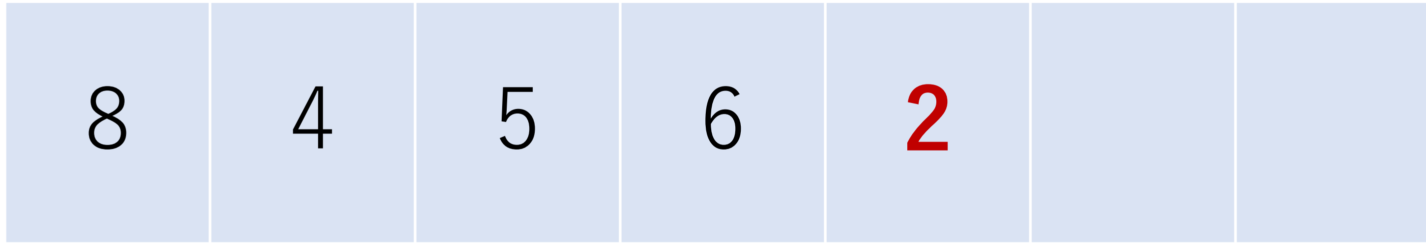
いわゆる「（ラーメン屋さんとかの）行列」．入った順に出ていく.

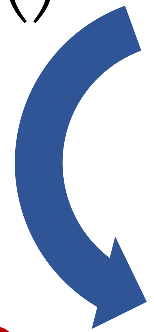
データを入れるenqueue は一番最後にくっつける，データを出すdequeueは一番先頭にあるデータを取り出す,という操作になる.

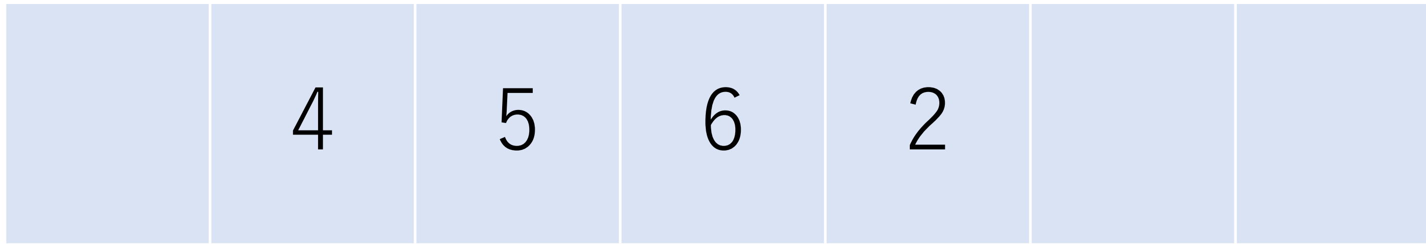
FIFO (first in, first out)

キュー









キュー

```
class Queue:
    def __init__(self, size: int):
        self.queue = [None]*size
        self.head = 0
        self.tail = 0
```

キュー

```
class Queue:
    def enqueue(self, a: int):
        if self.tail < len(self.queue):
            self.queue[self.tail] = a
            self.tail += 1
            print(self.queue)
        else:
            print('This queue is full.')
```

キュー

```
class Queue:
    def dequeue(self):
        if self.head <= self.tail:
            tmp = self.queue[self.head]
            self.queue[self.head] = None # なくてもよい.
            self.head += 1
            print(self.queue)
            return tmp
        else:
            print('This queue is empty.')
```

キュー

b = Queue(5)	
b.enqueue(8)	[8, None, None, None, None]
b.enqueue(4)	[8, 4, None, None, None]
b.enqueue(5)	[8, 4, 5, None, None]
b.enqueue(6)	[8, 4, 5, 6, None]
c = b.dequeue()	[None, 4, 5, 6, None]
b.enqueue(2)	[None, 4, 5, 6, 2]
c	8

キューの実装

単純に実装するとdequeueするたびにデータ領域が動いてしまう.

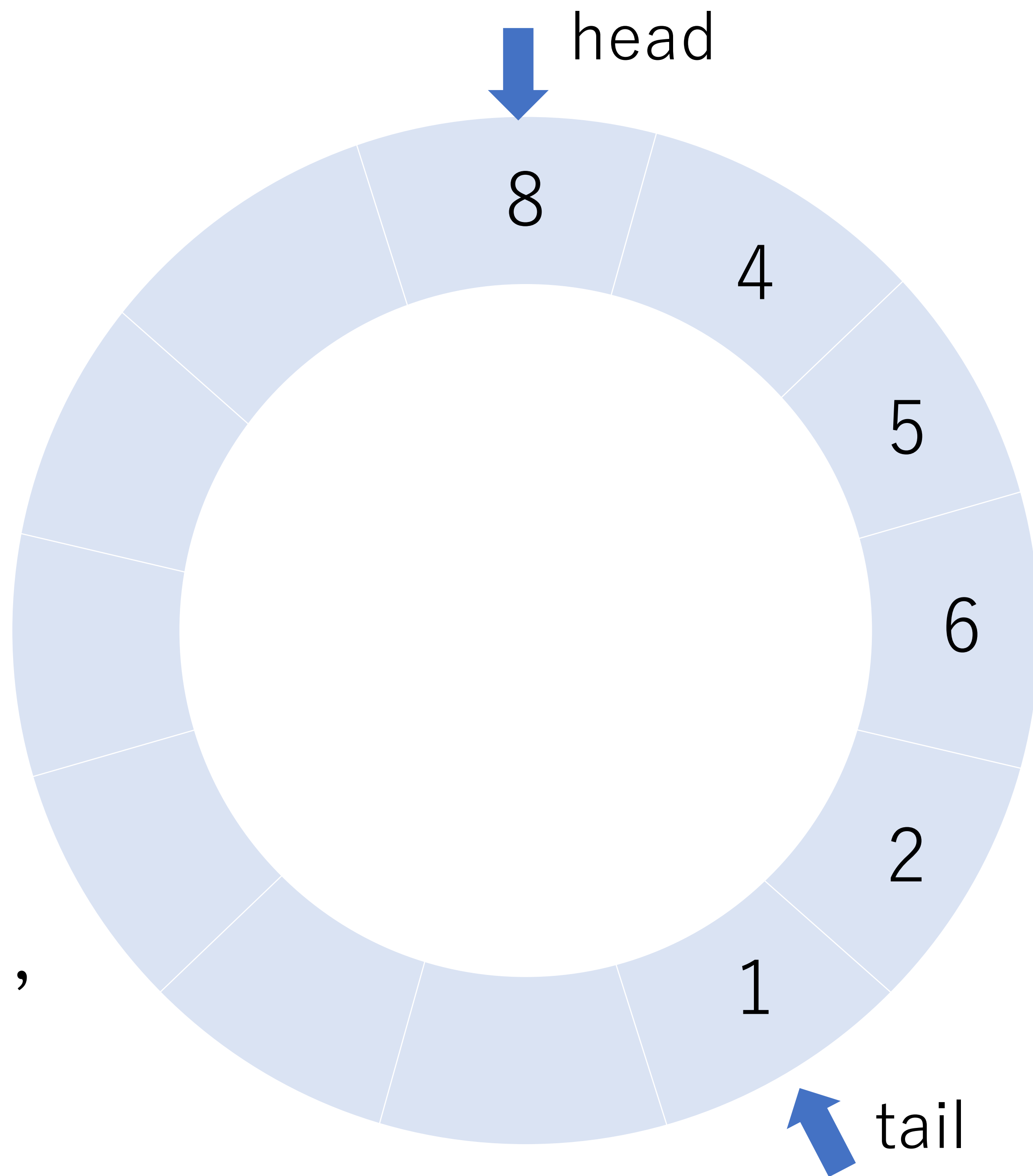
dequeueごとに全体をずらして先頭の位置をリセットするのは非現実的.

リングバッファ

円環状に見立てたバッファ．

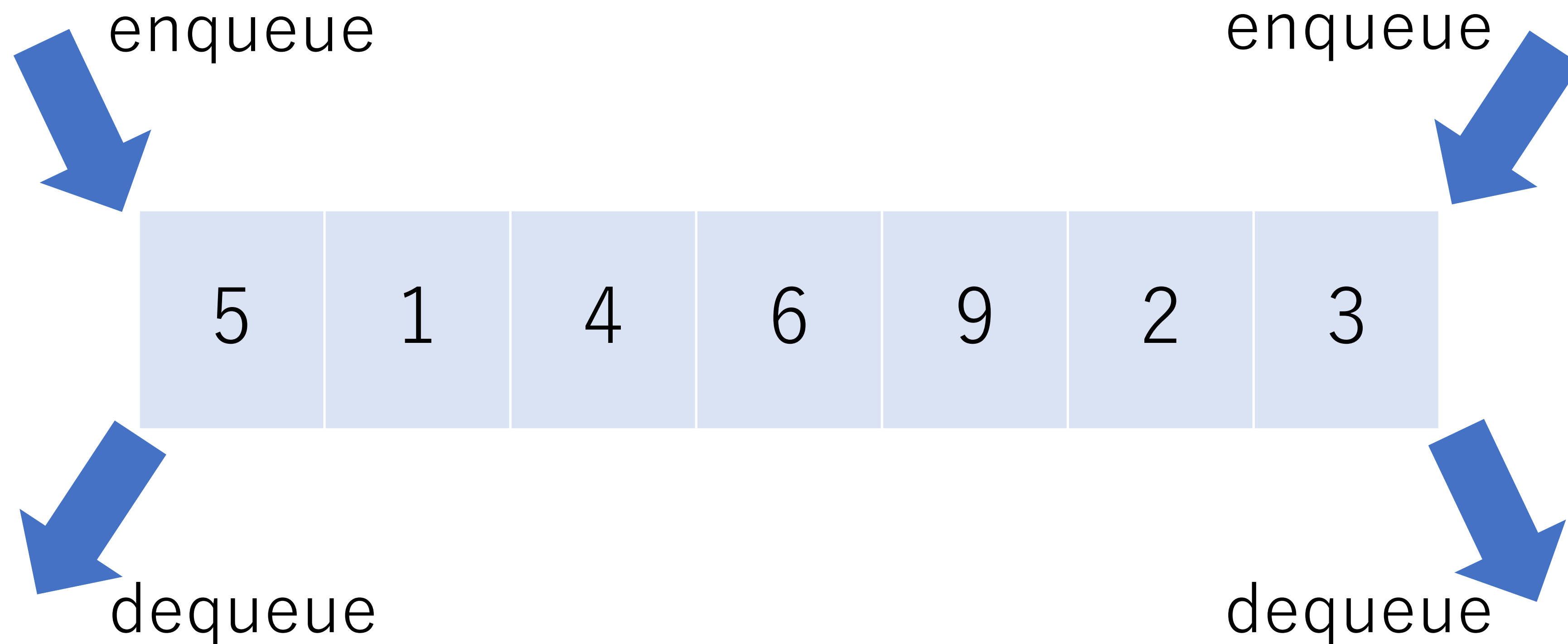
実装上はあらかじめ確保している領域の最後までいったらまた先頭に戻るように，先頭と末尾のindexを変更する．

最新のX個の情報を覚えておく，みたいな使い方もできる．



デック

キューの先頭と末尾のどちらからでもenqueue，dequeueできる．

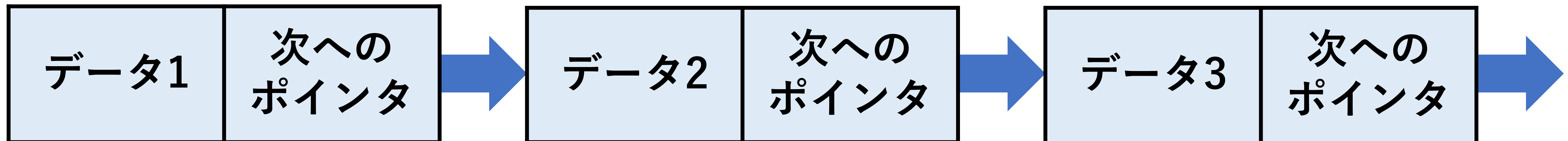


線形リスト

データと次のデータへのポインタ（あるいは次の要素・ノードの情報）を格納して，数珠つなぎにできるようにしたデータ構造.

末尾のリストの次へのポインタはNULLになる.

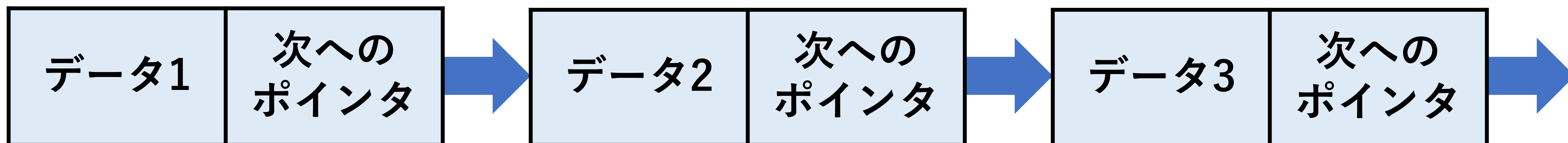
双方向や循環になっているものある.



線形リスト

要素の数の増減に応じて，必要な分だけのメモリ量のみを消費するので，空間計算量でメリットがある．

リストを順に辿らないといけないので，データのアクセスに時間がかかる場合あり．

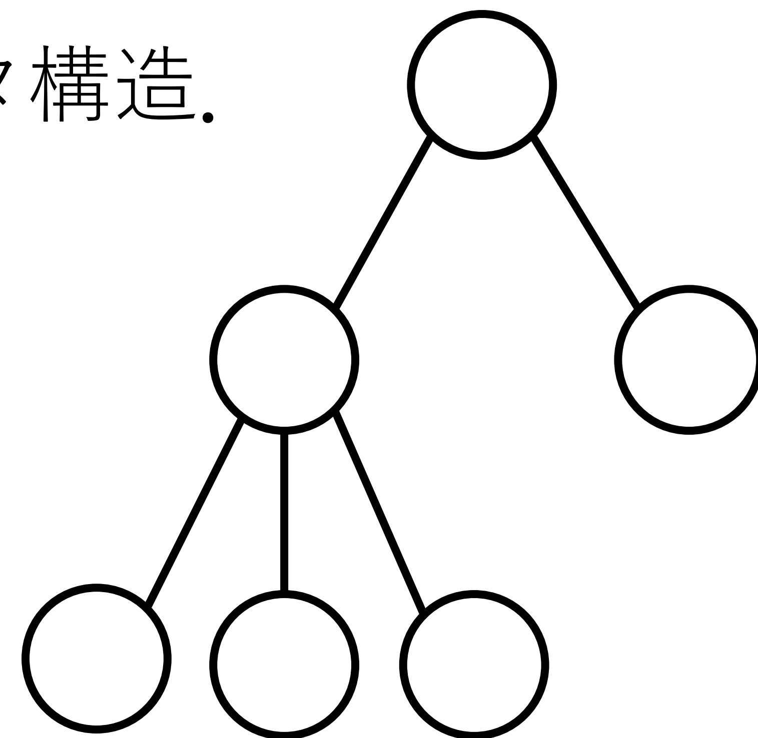


木構造（ツリー）

木をひっくり返してみたような形のデータ構造.

根ノード（root）と呼ばれる一番上の節点（node）から枝分かれしていく.

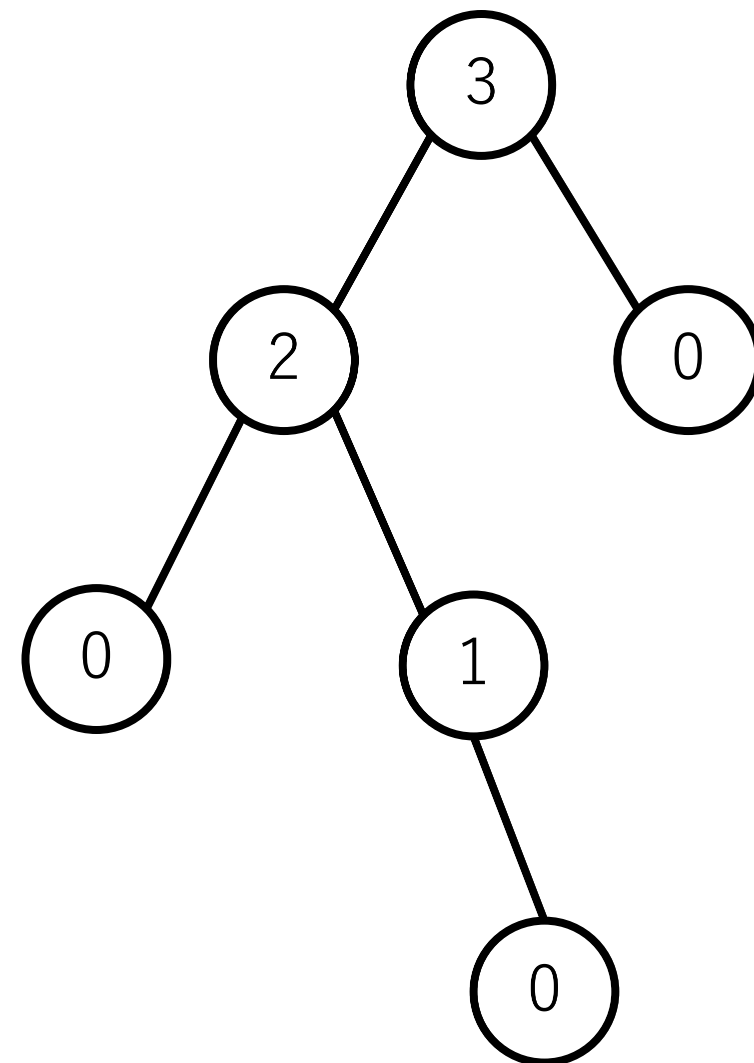
一番下（より子供のノードがない）ものを葉（leaf）という.



木構造（ツリー）

高さ

あるノードから，つながっている
葉ノードに至るまでの最大のエッジ
の数．葉ノード自体の高さは0.

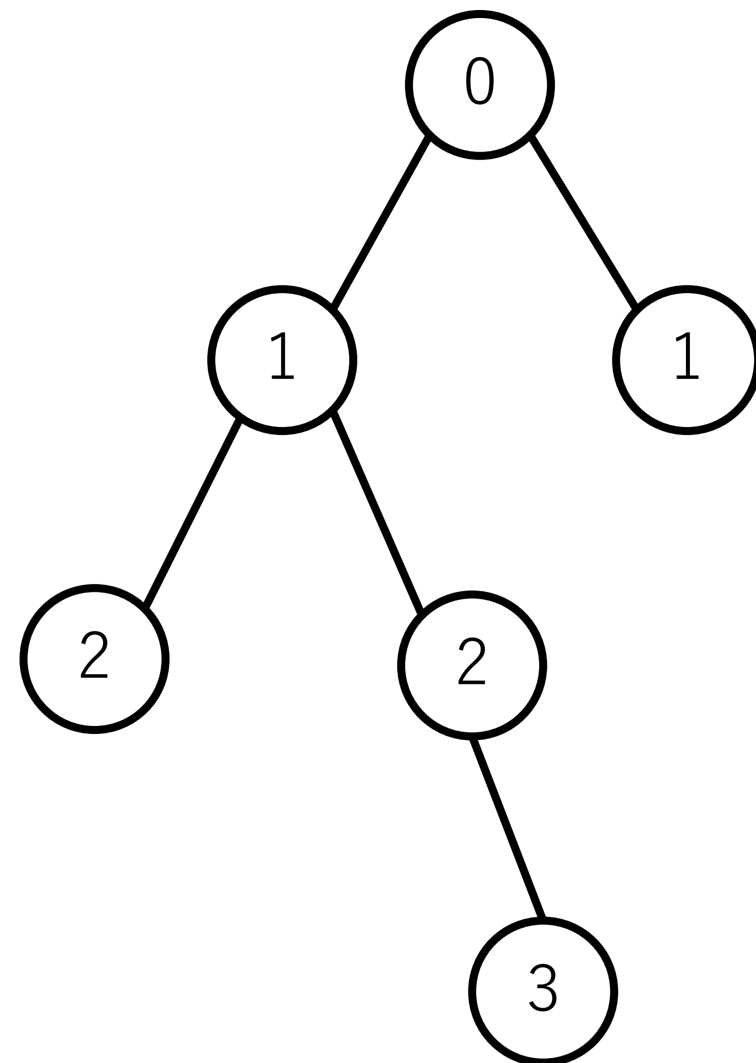


木構造（ツリー）

深さ

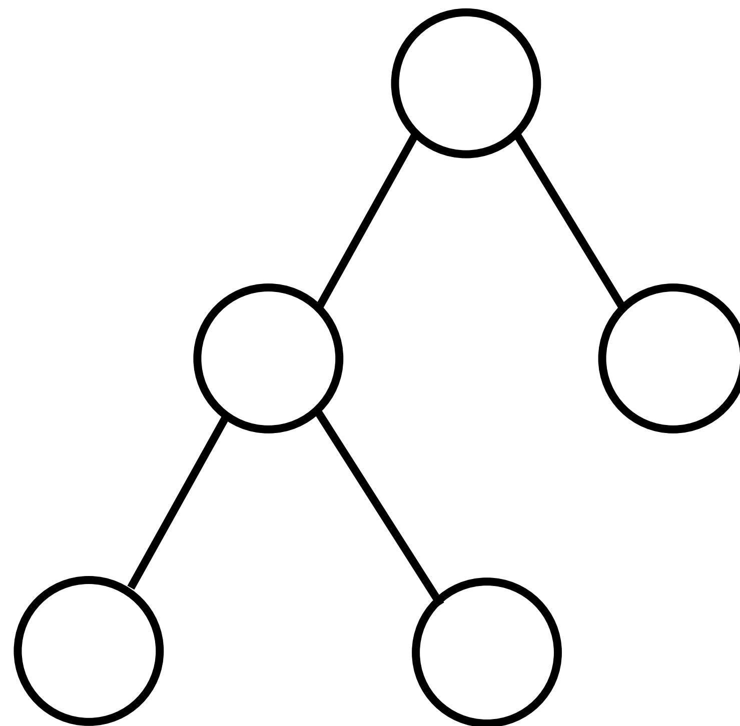
根ノードから，あるノードに至るまでに辿る必要があるエッジの数.
根ノード自体の深さは0.

各ノードの深さ



二分木（binary tree）

各ノードが持つ子ノードの数
が最大でも2つである木.

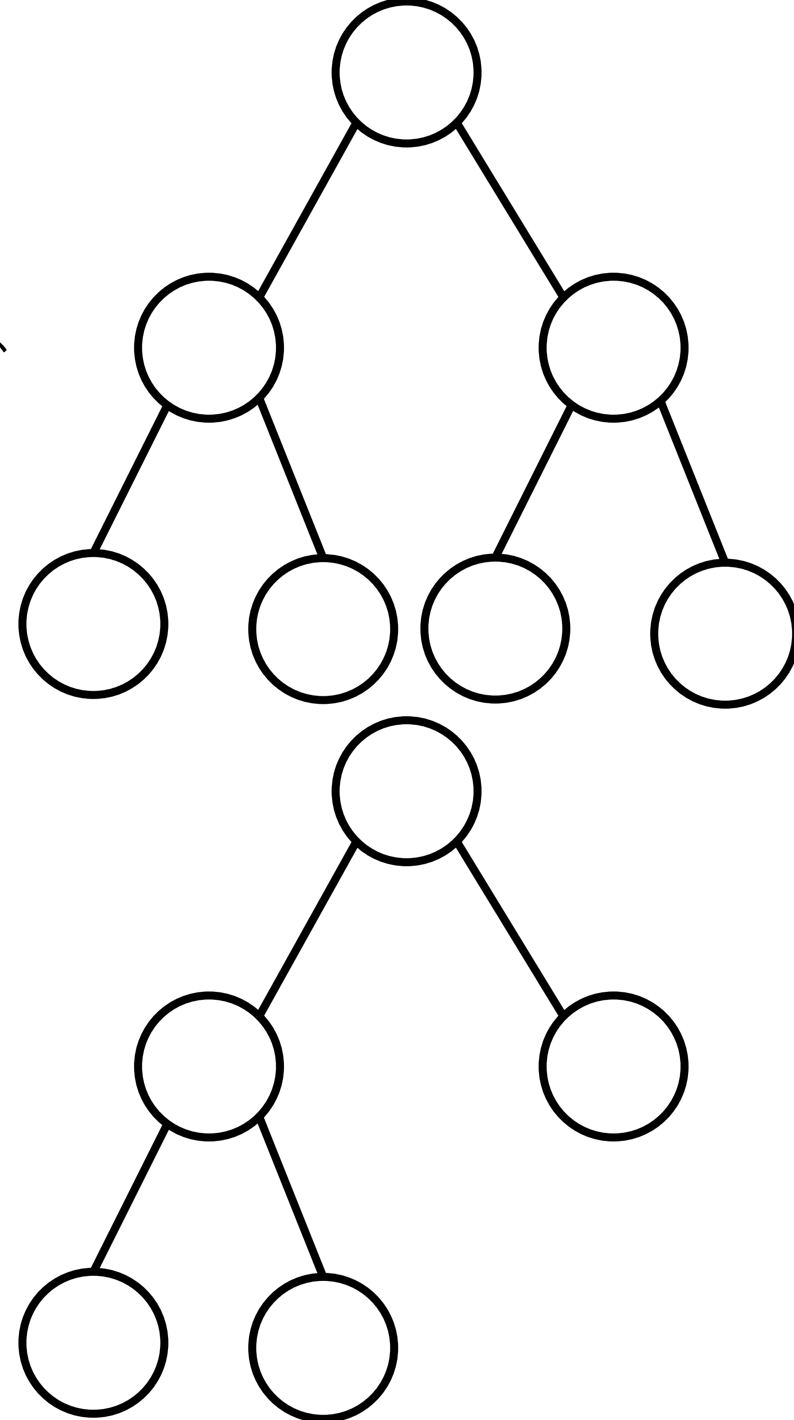


完全二分木

全ての葉が同じ深さ＆全ての内部ノードの次数が2である二分木.

一番下以外は完全に埋まっていて,
葉の部分のみ左から順に埋まっているものも，完全二分木と呼ぶ（ことが多い）.

英語だと，上がfull binary tree
で下がcomplete binary tree.



二分木

構造が簡単なので実装も難しくない．ノードは構造体で定義.

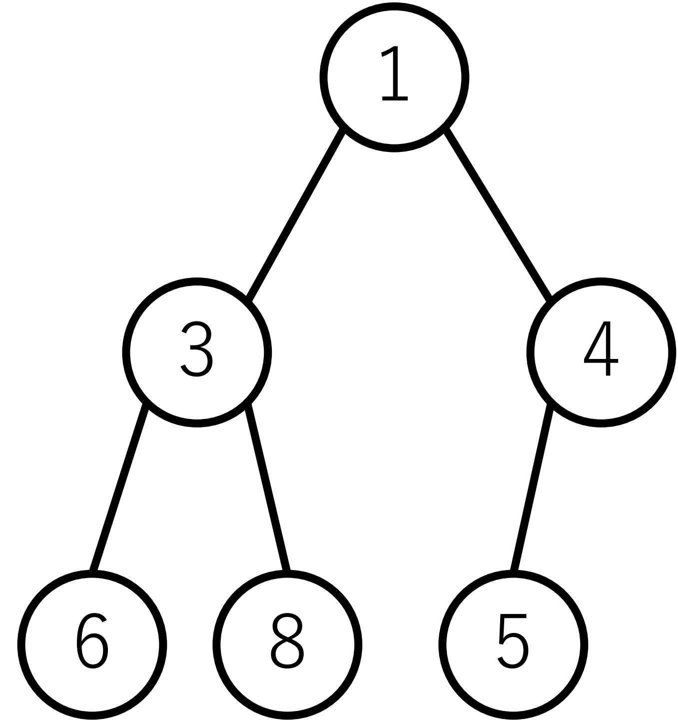
```
class Node:
    def __init__(self, data):
        self.data = data #このノードの値
        self.parent = None #親ノード
        self.left = None #左子ノード
        self.right = None #右子ノード
```

ヒープ

「親ノードは子ノードよりも常に同じか小さい（またはその逆）」という制約があるツリー.

根ノードは常に最大値（最小値）になる.

二分木を使った二分ヒープを指すことが多い.

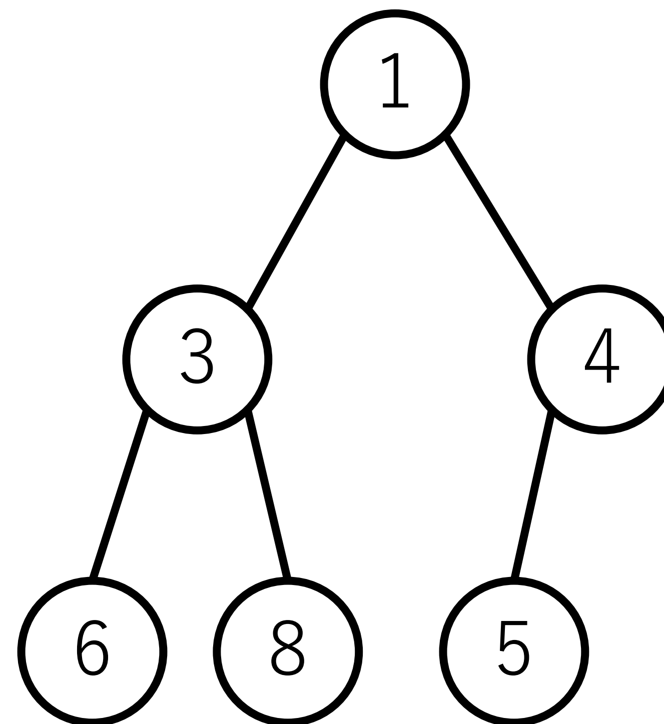


ヒープの操作：追加

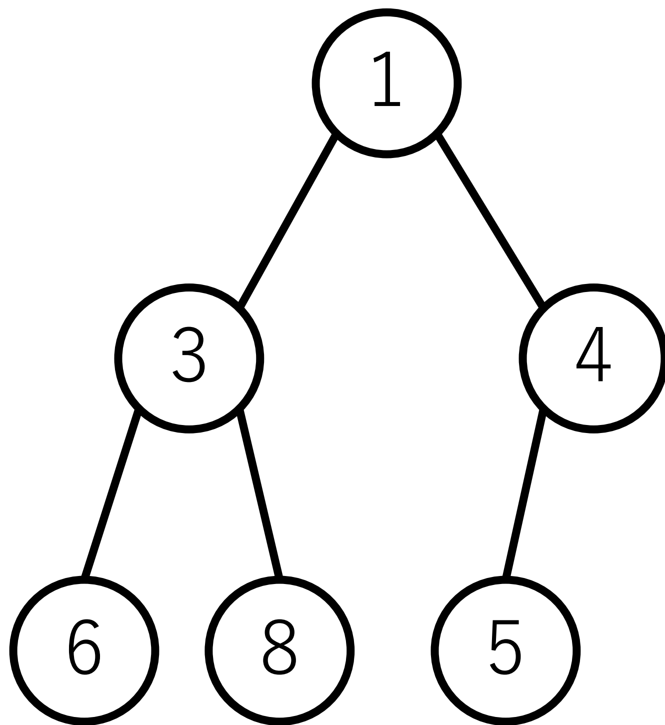
（以下，二分ヒープで説明）

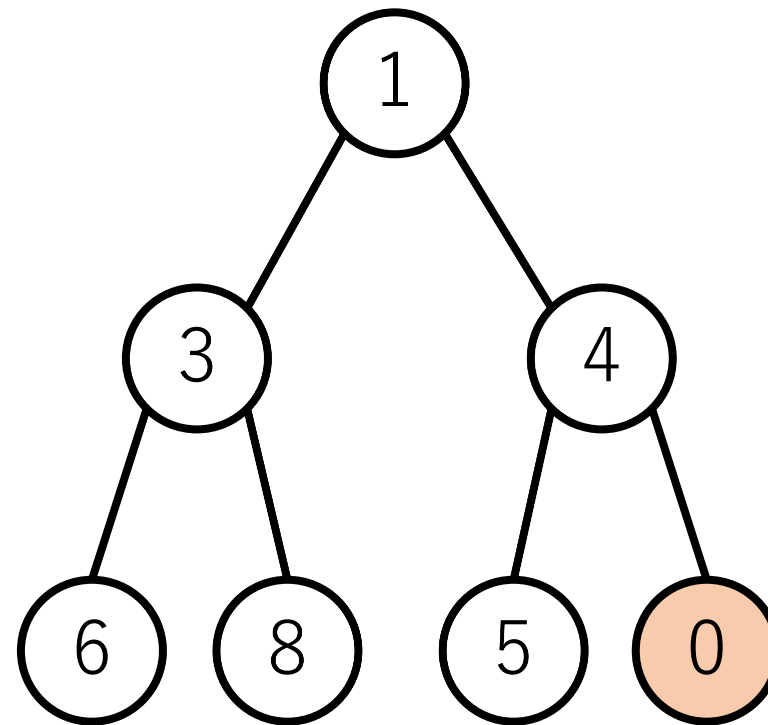
#1 空いている一番左に葉として追加.

#2 親と比較．制約条件を満たすように順次入れ替える.

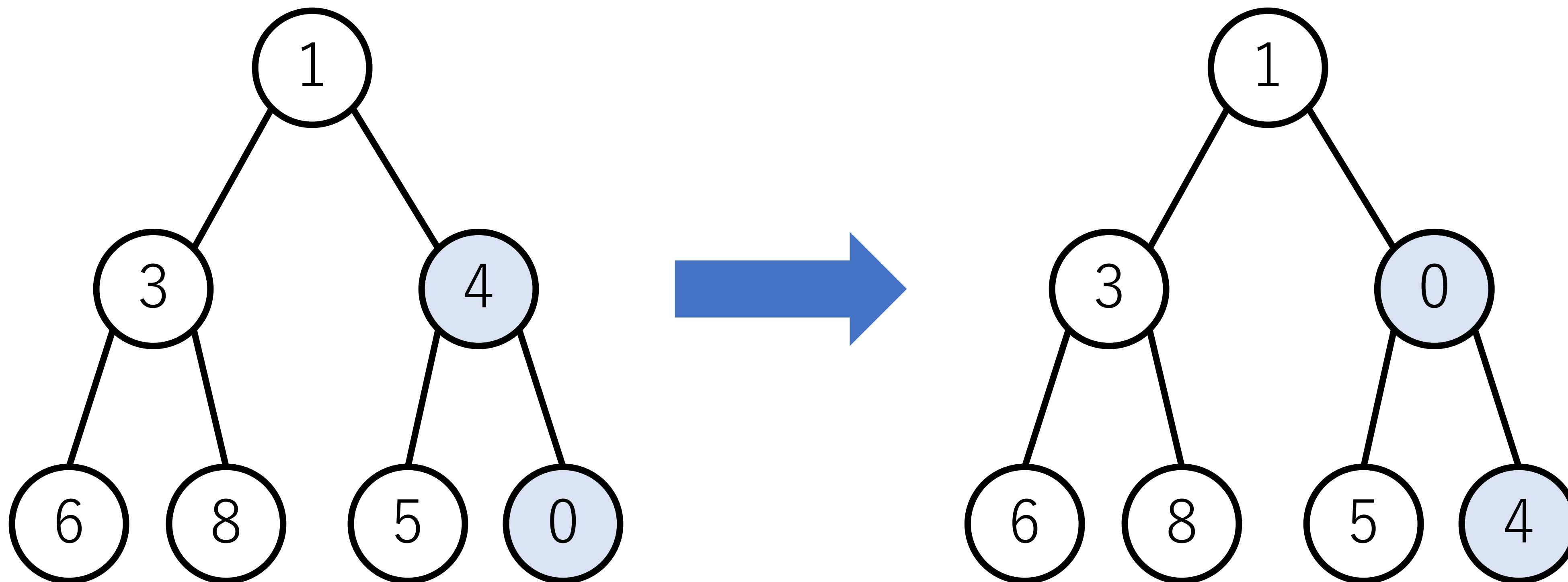


ヒープの操作：0を追加

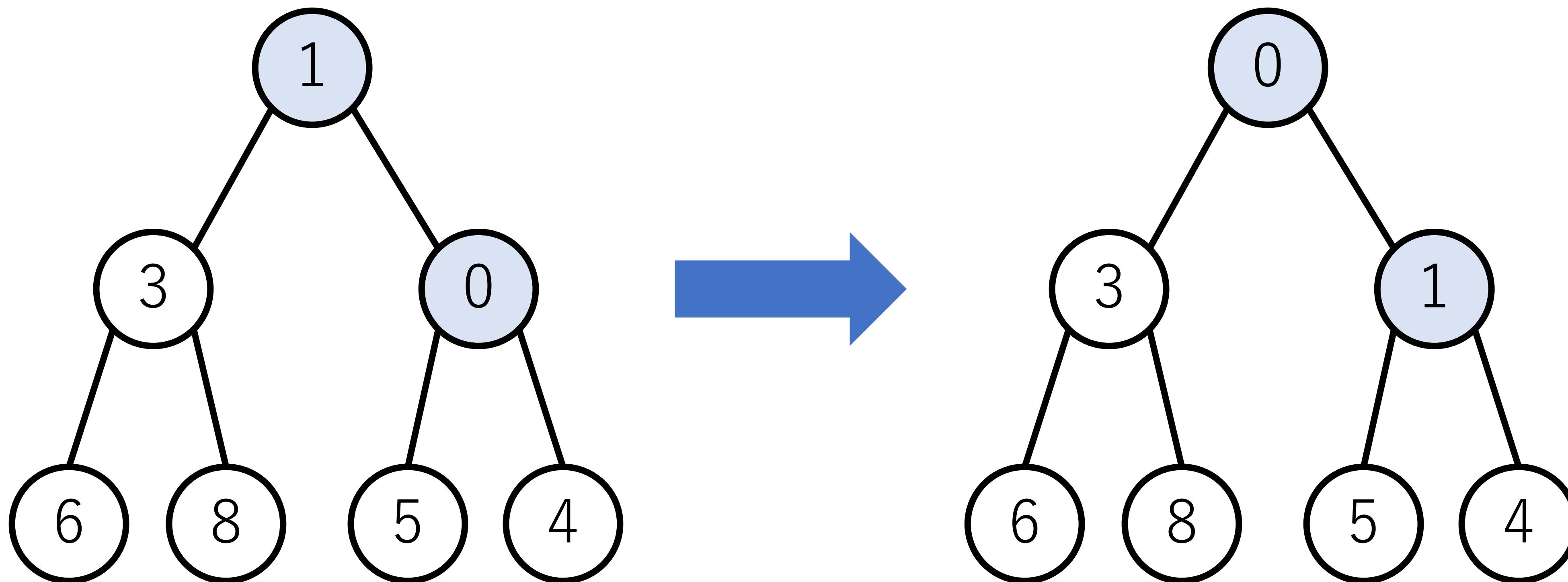




ヒープの操作：0を追加



ヒープの操作：0を追加



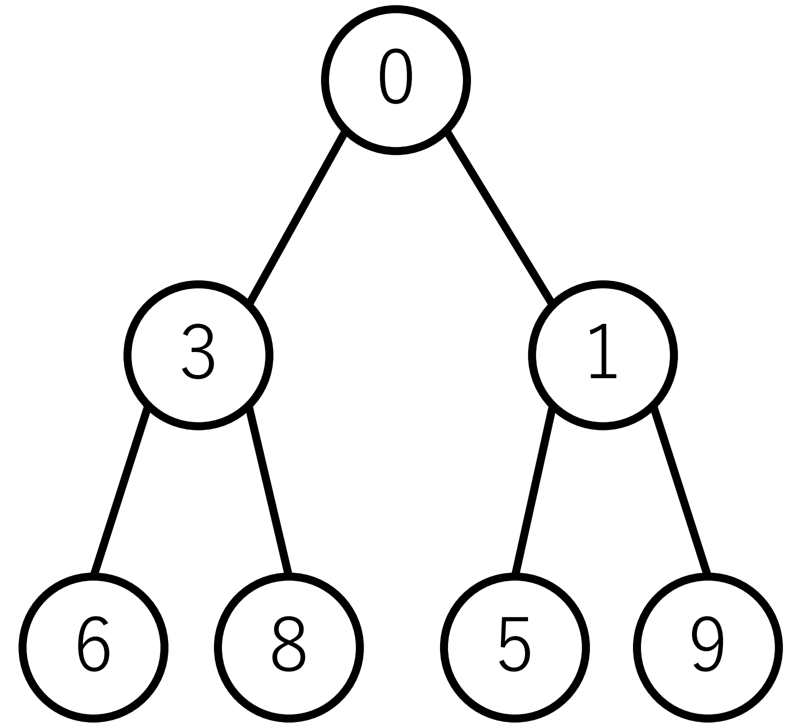
ヒープの操作：削除

#1 rootをとる.

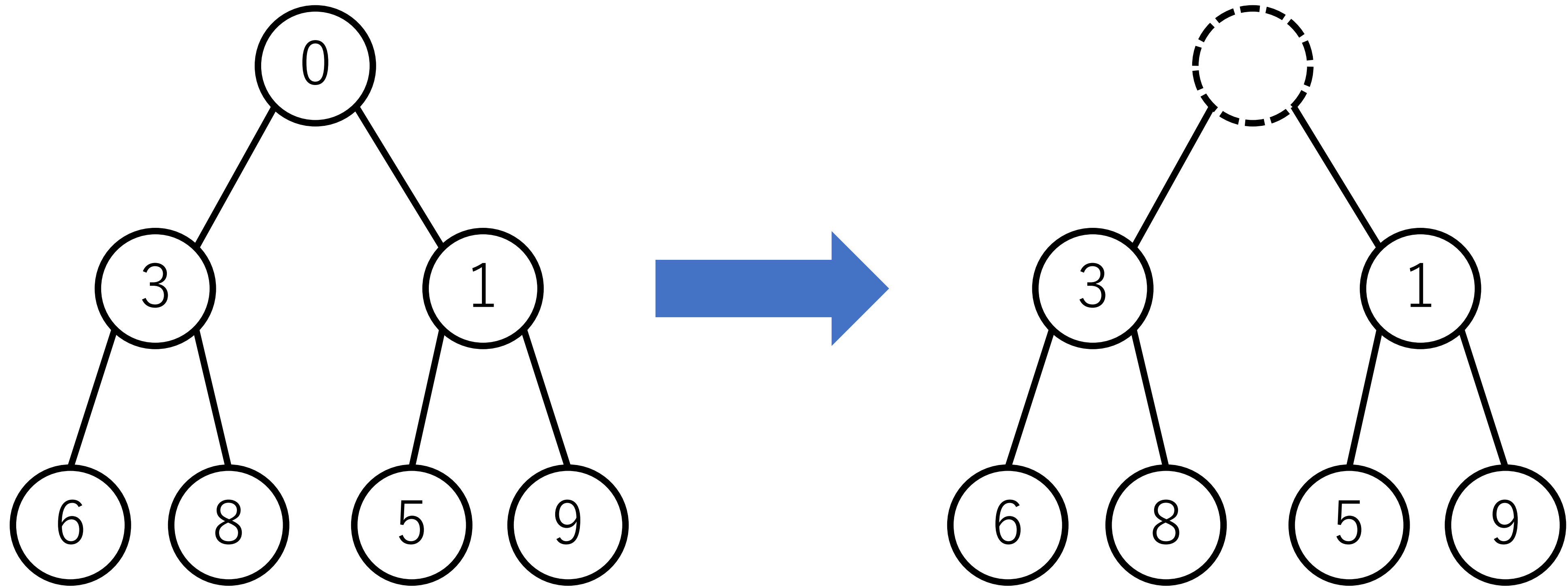
#2 一番右端にいる葉をrootにする.

#3 子ノードと比較し，子ノードのほうが小さい場合，より小さい方の子ノードと入れ替える.

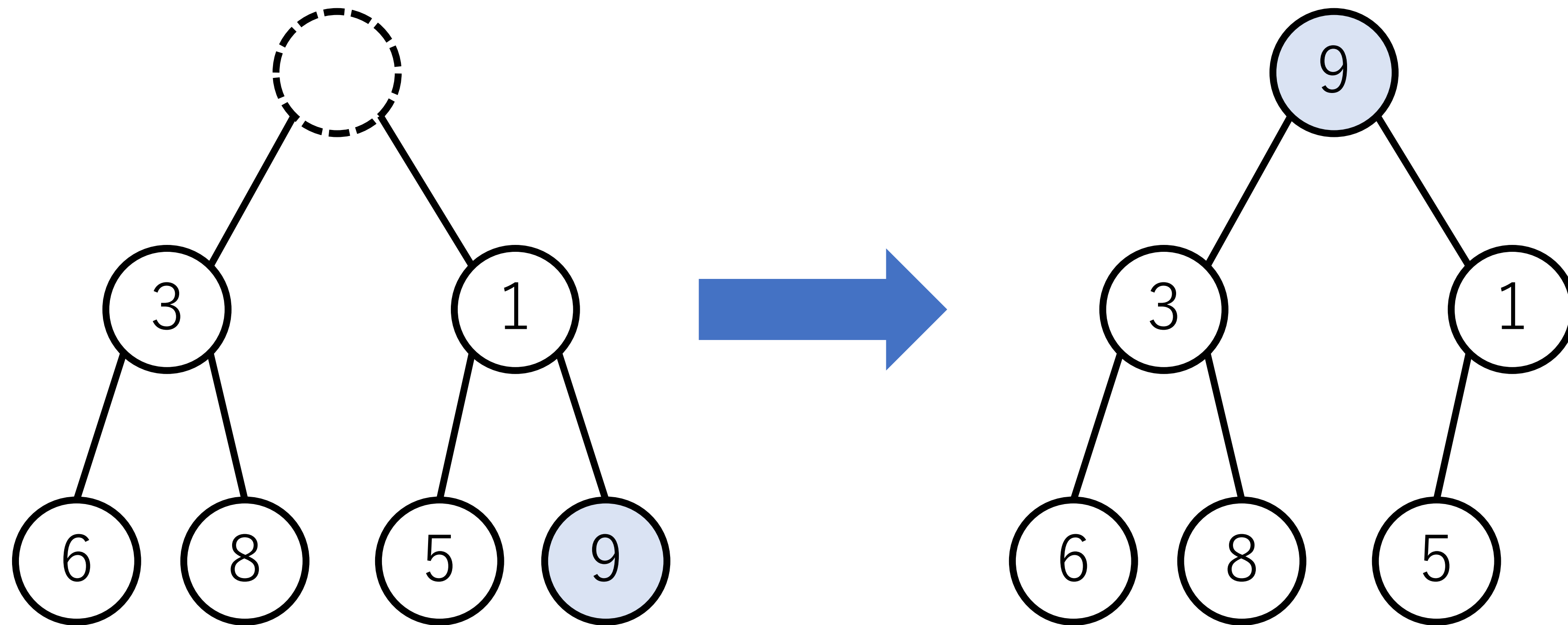
#4 制約条件を満たすまで入れ替えを続ける.



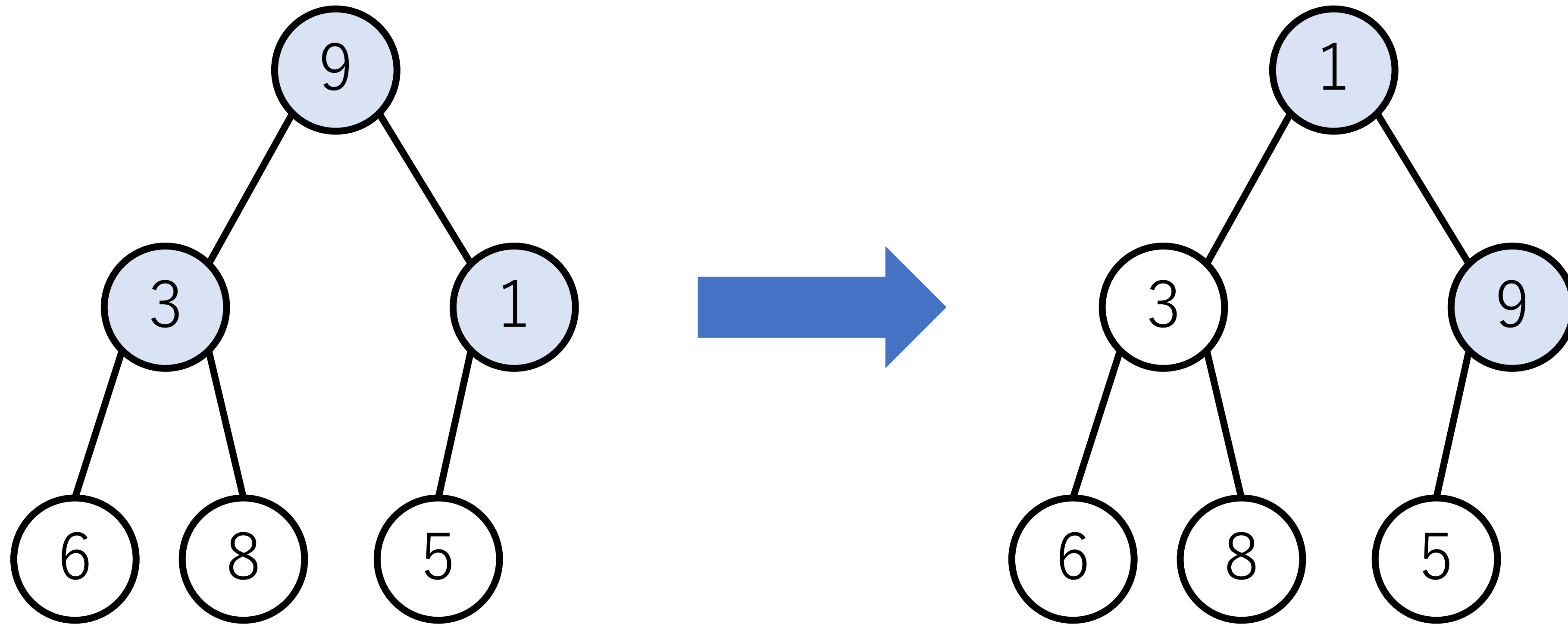
ヒープの操作：削除



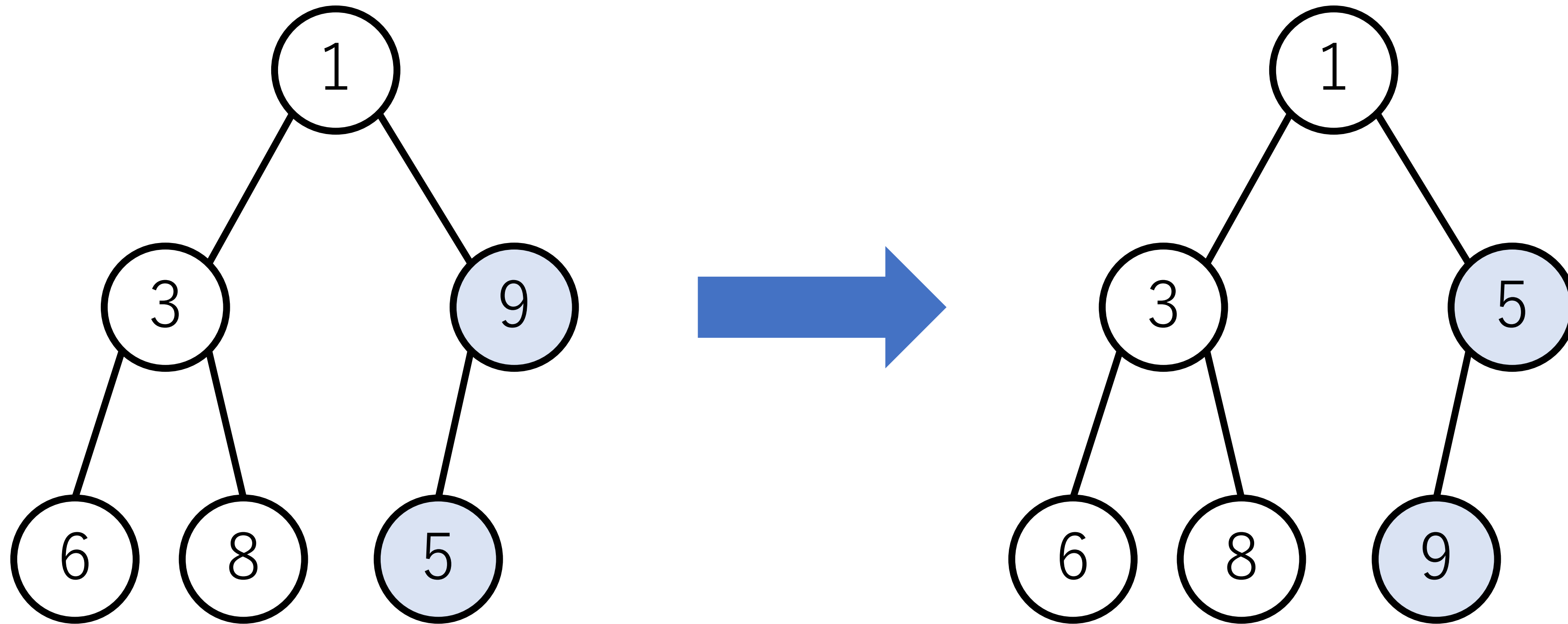
ヒープの操作：削除



ヒープの操作：削除



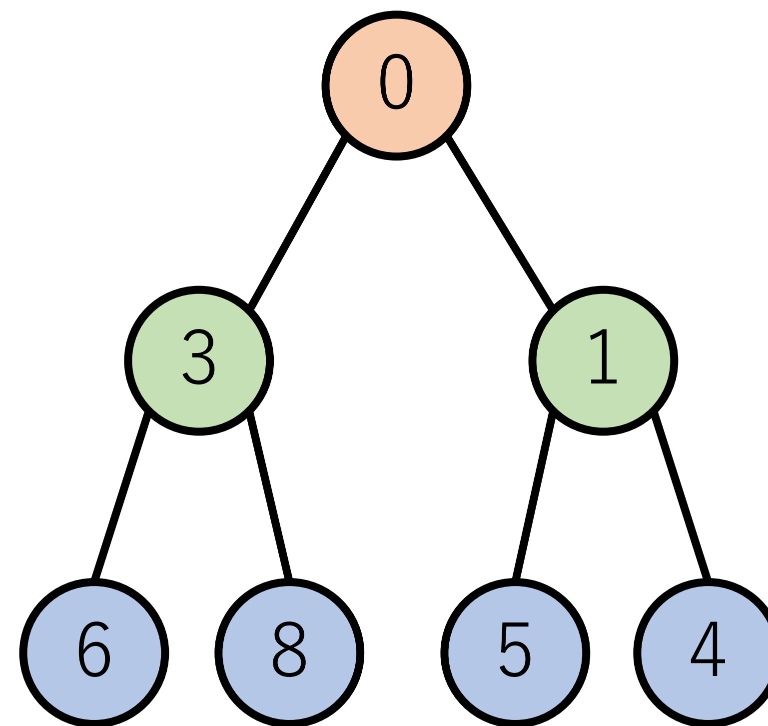
ヒープの操作：削除



二分ヒープの配列での実装

ノードa[i]に対して,
親: a[(i - 1)//2]
左の子: a[2*i + 1]
右の子: a[2*i + 2]
となるようにデータを格納.

構造体を使うことなく表現することが可能.



ヒープの計算量

追加の場合，平均的には，高さ/2 回分の比較・入替が必要．（最悪の場合は高さ分）．

木の高さはノードの数nに対して，$\log_2(n+1)$.

よって，$O(\log n)$.

削除も同じく，$O(\log n)$.

優先度付きキュー（プライオリティキュー）

dequeue時に優先度の高いものから順に出すキュー.

優先度は要素自体の値で決めていることが多い（例えば,
要素の値が大きければ優先度が高い，とするなど）.

ヒープを使って実装することも多い.

まとめ

累積和，しゃくとり法

みなさんもぜひスライドを見ながら追実装し，
性能比較してみてください．

累積和的な考え方は今後ちらほら出てきます．

データ構造

スタック，キュー，線形リスト，ツリー，ヒープ

コードチャレンジ：基本課題#2-a [1点]

スライドで説明されているキューのクラスを改良して, リングバッファを使ったキューを自分で実装してください.

キューにおける追加，削除を自分のコードで実装してください.

コードチャレンジ：基本課題#2-b [2点]

二分ヒープを自分で実装してください．

スライドで紹介したものは最小ヒープ（根が最小値になるもの）ですが，課題では最大ヒープ（根が最大値になるもの）である点に注意してください．

ヒープにおける追加，削除を自分のコードで実装してください．

コードチャレンジ：Extra課題#2 [3点]

括弧の対応が取れている文字列と，それに対する操作の回数の関係を問う問題.

詳細はtrack上の問題文をよくお読みください.