Algorithms (2021 Summer)
#3：データ構造

矢谷 浩司

前回出た質問

「7の30乗をスライド通りと実装すると157445110と出力され、スライドの実行結果例と異なります。どちらか間違っていますでしょうか？」

→矢谷のミスです．申し訳ありません．．．
修正したものを講義のホームページにアップロード済みです.

前回出た質問

「しゃくとり法を使わなかったときにn(n+1)/2となる説明をもう一度説明していただけるとありがたいです。」

→以下，TAの鈴木くんによる説明.

|〇|〇|〇|〇|〇|

〇を数字だと思ってもらうと，区間の選び方は，N+1個の縦棒から右端と左端の2つを選ぶのと同じなのでbinomial(N+1, 2) = N*(N+1)/2になります

前回出た質問

「言語によって順番を考えないといけないというのはなぜですか？」

→以下，TAの役山くんによる説明.

言語によっては本来のa*b/cの値は32bitや64bitに収まっても，計算途中のa*bの時点でオーバーフローが発生し答えが正しくでない場合がある，ということです.

前回出た質問

「10^9+7 のような「大きな素数の剰余を取るべき大きい数」というのは具体的にどこを超えたらあてはまる、というのはありますか？」

→以下，TAの役山くんによる説明の要約.

競技プログラミングなどにおいては，64bit符号付整数の最大値を超えるあるような場合に，剰余を取って答えを出す指示が出ることが多いようです.

前回出た質問

「aで割ることはa^(m-2)をかけることに等しい理由をもう一度説明していただけませんか」

→フェルマーの小定理により，　a^(m-1)はmodの世界では1になります．直感的な説明をすると，これを分割してa×a^(m-2)と見ると，aにa^(m-2)を掛けると（modの世界では）1になる，と解釈できます．つまり，通常の数字の世界における1/aのような存在がa^(m-2) となります．

前回出た質問

「２種類コードを書き、１つ目は階乗を毎回計算するもの、２つ目は階乗のリスト(n!まで)をつくってから、そのリストを参照するものを書きました。結果は、前者は大きい入力だと実行時間が基準を超えてしまい、後者はokでした。

後者の方が速そうなのはなんとなくわかるのですが、テストの合格の是非を決めるほどの差があるとは思えないです。どちらも階乗の計算量はO(n)で、それを前者は３回(階乗計算関数を呼び出す回数)、後者は1回(階乗リスト作成)呼び出しており、後者の方が速いのはわかるものの、O(n)って数回呼び出しただけでそんなに遅くなるものでしょうか？」

前回出た質問

→TAさんの指摘通りでした！階乗計算で、計算途中で毎回ちゃんと剰余をとると、テスト通りました。ありがとうございます。

1000!

-> 毎回の剰余なし：309 usec / あり：202 usec

10000!

-> 毎回の剰余なし：19 msec / あり：2.1 msec

100000!

-> 毎回の剰余なし：2.3 sec / あり：21 msec

前回出た質問

「先週のチャンネルで基本課題の解答例のコードは公開されないというお話がありましたが、スライド通りに実装したと思っても実行時間が基準を超えてしまいます。実行時間が短くなるように自分で工夫した点が最適であったかを知りたいので、課題の考え方についてもっと詳しく書かれた資料などがあるとうれしいです。」

→今後もう少し細かく説明をするようにしたいと思います．一方で課題を出す側として，みなさんに単に写経するのではなく，少し頭を使ってほしいという意図があるので，そこはご理解いただけるとありがたいです．

前回出た質問

「基本課題でfactorialやpowを使っても大丈夫でしょうか？」

→基本課題においては基本的に自分で書かなければならないと考えてください.

前回出た質問

「extraでは特に制限なく、標準/外部ライブラリを使っても良いという認識であっておりますでしょうか?」

→構いませんが、使用したライブラリの効率性に頼った解答(つまりライブラリを使用しないように書き換えた時に不正解となる解答)については減点される可能性があることに注意してください.

前回出た質問

「この授業のextra課題は今後さらに時間がかかる問題が増えるのでしょうか？」

→こちらの想定としては長くても6時間程度としていますが，後半は少し内容が複雑になる（グラフの話など）ので，多少時間がかかるかもしれません．

前回出た質問

「課題の点数(/100)と成績(優、良)との関係は「80点以上なら優」のように、絶対評価で決まるのでしょうか？それとも、他の履修生との相対評価で決まるのでしょうか？」

→学部後期課程の授業では一部をのぞいて，全学の成績評価の取り決めに従うようになっていると理解しております．この授業でもこの取り決めにしたがい，3割程度が優，優上になるように評価されます.

前回出た質問

「レポートとextra課題の2択という形であるならば、extra課題に関しては提出期限をもう少し長くとって頂きたいです。周りの反応等を見ていても、挑戦はしたいが時間が無くて諦めざるを得ないという学生がある程度いるのではないかと思います。」

→Extra課題はあくまでExtraであり，選択課題として位置づけていることもありますので，受講者全員がすんなり解ける問題にはなりません．基本課題とのギャップがあることは理解していますが，だからこその選択課題になっています．

前回出た質問

「レポートとextra課題の2択という形であるならば、extra課題に関しては提出期限をもう少し長くとって頂きたいです。周りの反応等を見ていても、挑戦はしたいが時間が無くて諦めざるを得ないという学生がある程度いるのではないかと思います。」

→提出期限や難易度に関しては提出状況を見ながら調整する余地はあり，これはTAさんとも協議をしながら進めたいと思います．ただし，提出期限は伸びたとしても1日程度（月曜日の夜まで）とお考えください．

前回出た質問

「レポートとextra課題の2択という形であるならば、extra課題に関しては提出期限をもう少し長くとって頂きたいです。周りの反応等を見ていても、挑戦はしたいが時間が無くて諦めざるを得ないという学生がある程度いるのではないかと思います。」

→また，提出期限や難易度の調整に関しては，EEICの学生さん（特にUTAS登録者）の提出状況を基準に判断をしますので，こちらも合わせてご承知おきください．

前回出た質問

「この授業で学んだことを定着させるためにAtCoderのコンテストに出てみようと思っているのですが、基本課題とExtra課題はそれぞれ目安としてAtCoder ABCの問題番号(A~F)で言うとどれくらいの難易度を想定して作っているなどあれば教えていただけると幸いです」

→TAの役山さんのお答え.

ABCの問題番号の難易度は回によってばらつきがあるので一概には言いにくいですが、問題ごとのdifficulty(色)としてExtra課題(満点解法)は緑・水色から難しくても青色下位程度になることを想定しています

前回出た質問

「この授業で学んだことを定着させるためにAtCoderのコンテストに出てみようと思っているのですが、基本課題とExtra課題はそれぞれ目安としてAtCoder ABCの問題番号(A~F)で言うとどれくらいの難易度を想定して作っているなどあれば教えていただけると幸いです」

→TAの役山さんのお答え.

基本課題については、実際のコンテストだとライブラリを使用するだけになるので、どれぐらいの難易度かを表現するのは難しいですが、学習したアルゴリズム・データ構造を使用する問題は多く出題されます

前回出た質問

「編集途中で時間が終わってしまった場合、最終コードが提出されるということはテストケース0/nの状態で提出されるのでしょうか。」

→こちらが見えるものとしては，一番最後にテストケースを動かしたコードまでの，保存されている全てのコードで，点数は最高得点のものを自動的に採用する仕様になっています.

参加フォームで出た質問

「途中からになりますが，参加可能でしょうか？」

→もちろんです！今までの講義は配信の録画と
スライドにて自習していただければと思います．

ただし，コードチャレンジに関しては締切を
過ぎたものは遡って受験できないことをご承知
おきください．

参加フォームで出た質問

「アルゴリズムの原理や証明の解説を充実させて欲しいです。(競技で勝つために今まで多くのアルゴリズムを訳もわからないままコピペして使ってきたので)」

→この講義においても原理の直感的・簡単な説明はトピックに応じてしていきますが，全てのアルゴリズムに対してそれらが提供されるわけではありませんので，その点はご承知おきください．あくまでコードに落としこむことに重点を置いていきたいと思います.

参加フォームで出た質問

「正式にutasで履修登録はできないのですが、聴講はしたいです。そのため、utasで履修登録期間をすぎてもitclmsの履修登録はできる状態にしていただけるとありがたいです。」

→ITC-LMSの仕組みがよくわかっていませんが，今の状態を私の方で変更する予定はありません．また，お知らせ等は講義のslackスペース，およびホームページにて行うことを主とし，合わせてUTASやITC-LMSで行いますので，slackを随時チェックしていただければと思います．

単位取得を希望される方へ

現在，UTASには登録されているが，こちらの登録にはない方が4名います．課題を受け取れないと成績が付かないので，至急講義のページから登録をお願いします．
https://iis-lab.org/algorithms-entry

- 工学部電子情報工学科の4年生の方1名
- 工学部精密工学科の3年生の方2名
- 文学部人文学科心理学専修課程の4年生の方1名

課題未着手に関して

課題配信後，課題の開始をしないまま期限を迎えたものは「未着手」としてこちらに記録されます．

「未着手」が累積4回以上となった方は成績がつきませんので，ご注意ください．（未受験扱い）

途中からこの授業に参加された方で，配信を受け取っていない場合はこのカウントには含まれません．

Extra課題の解答例

以下にて公開しています（ECCSアカウント必要）.

https://drive.google.com/drive/folders/1YIBPZpjZfLe3kovlJSKx5fZM6ar5zB8S?usp=sharing

無断の転送や転載はお控えください.

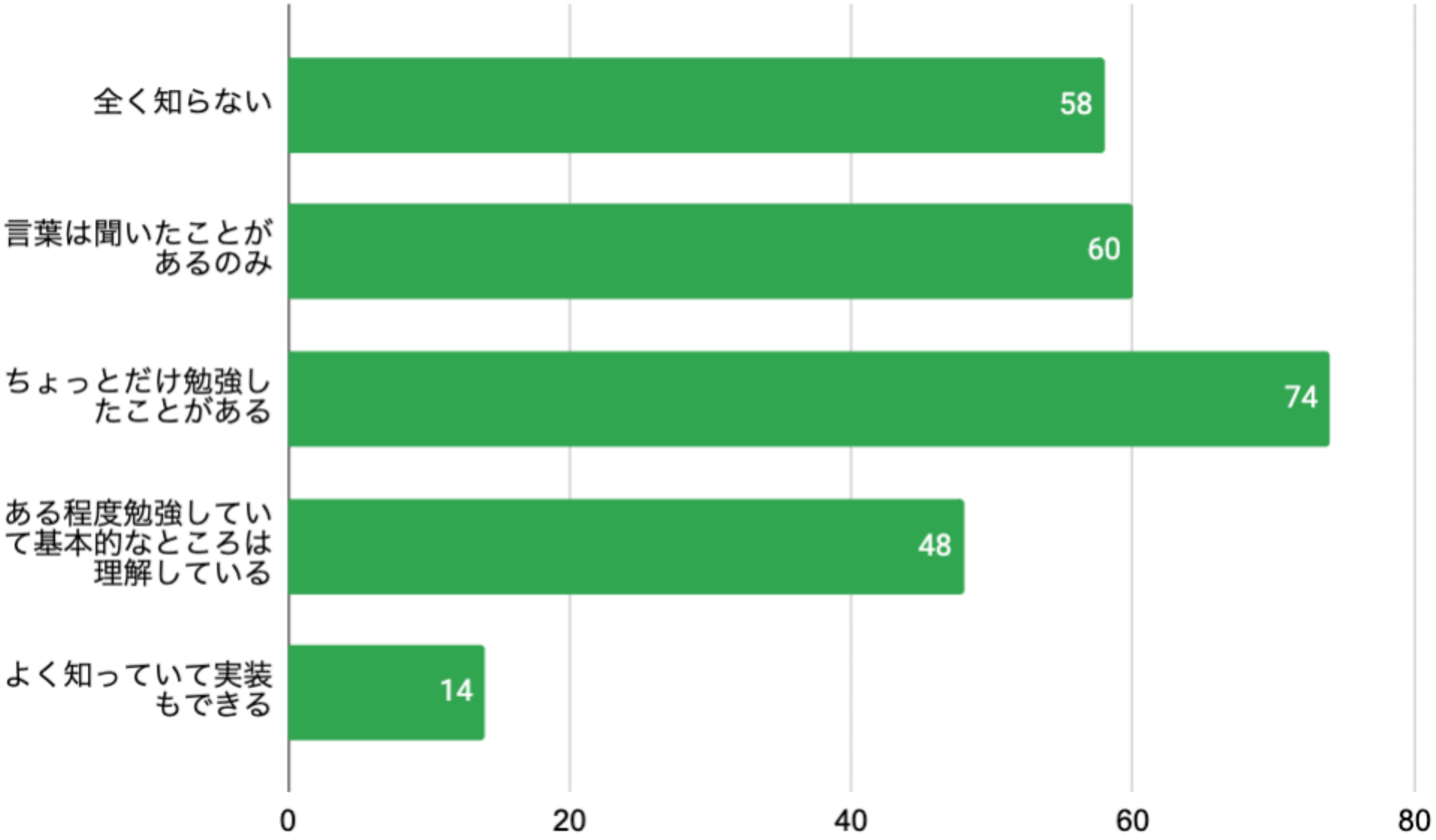
提出コードの返却

Google drive経由での返却を予定しております.

皆さん1人1人に対して共有フォルダを作り，そこに共有するというスタイルを予定.

今週，もしくは来週にテストを予定しています．Google driveより通知が来ると思いますので，ご確認をお願いします．Slackでもアナウンスいたします.

データ構造



全く知らない
58
言葉は聞いたことがあるのみ
60
ちょっとだけ勉強したことがある
74
ある程度勉強していて基本的なところは理解している
48
よく知っていて実装もできる
14
0
20
40
60
80

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・処理すればよくね？

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

セグメント木
BIT

スタック

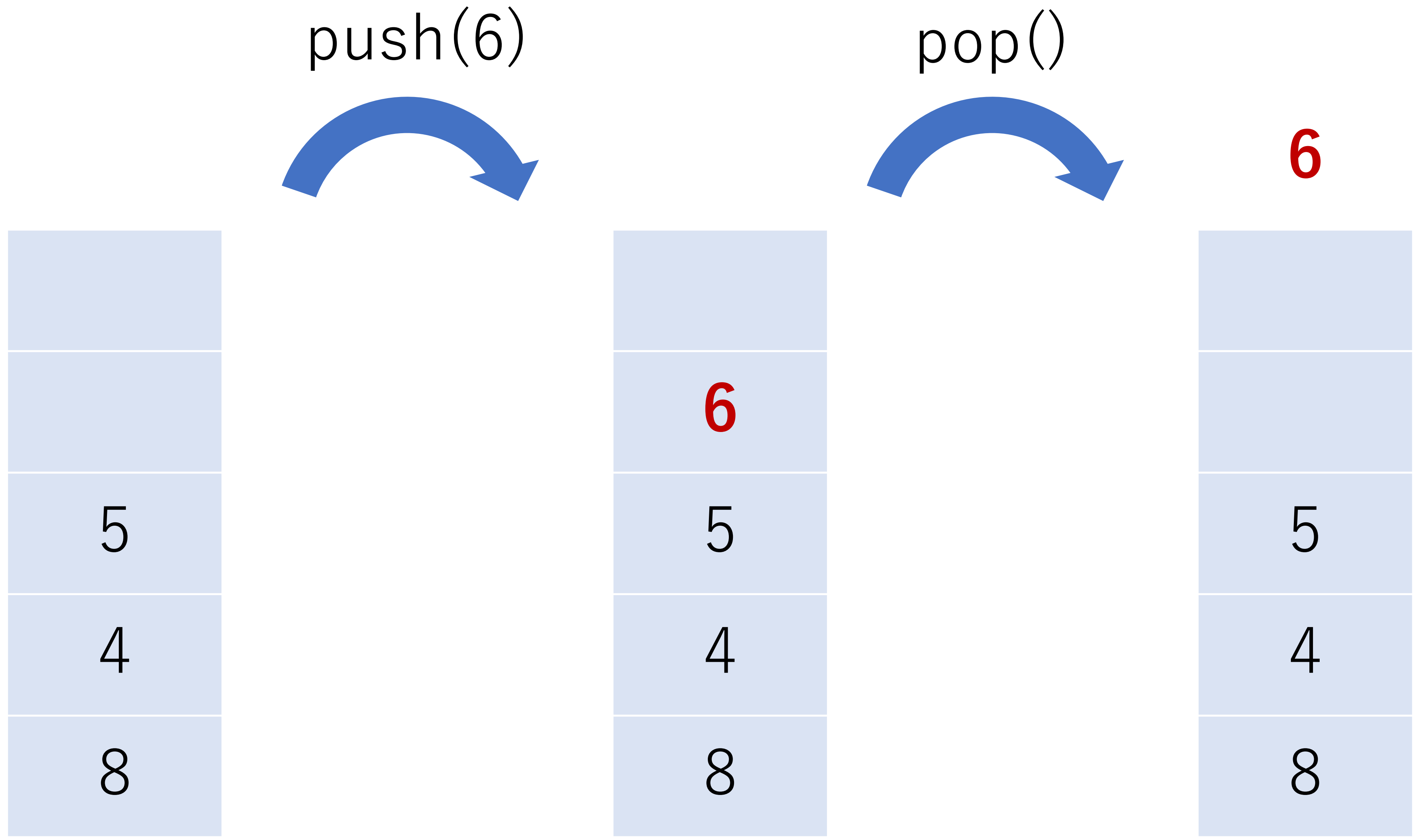
上にどんどん積み重ねていく形でデータを保持する構造.

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



スタック

```
class Stack:
    def __init__(self, size: int):
        # 値を格納しておくバッファ
        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

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

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

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

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.

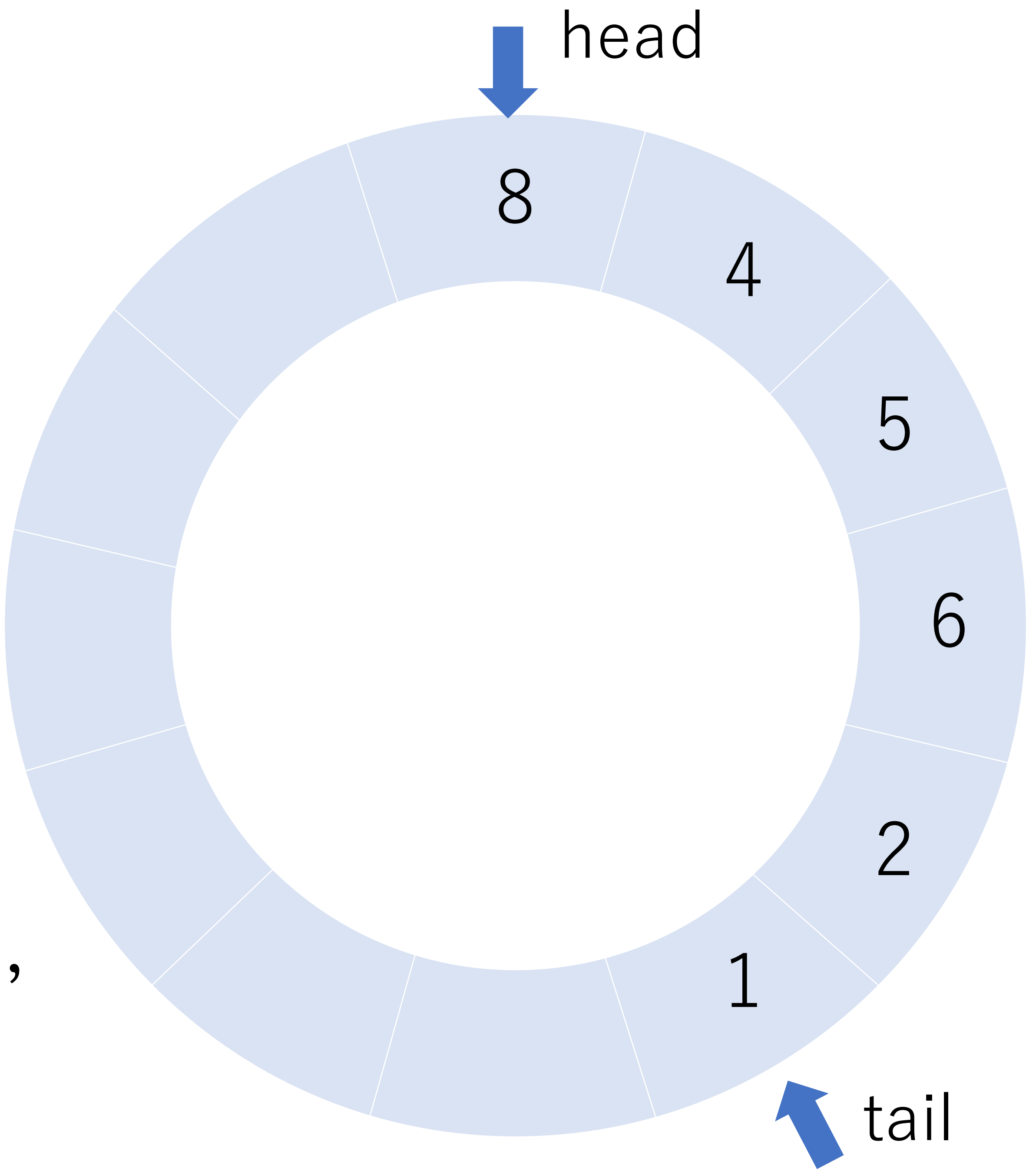
dequeueごとに全体をずらして先頭の位置をリセット
するのは非現実的.

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.

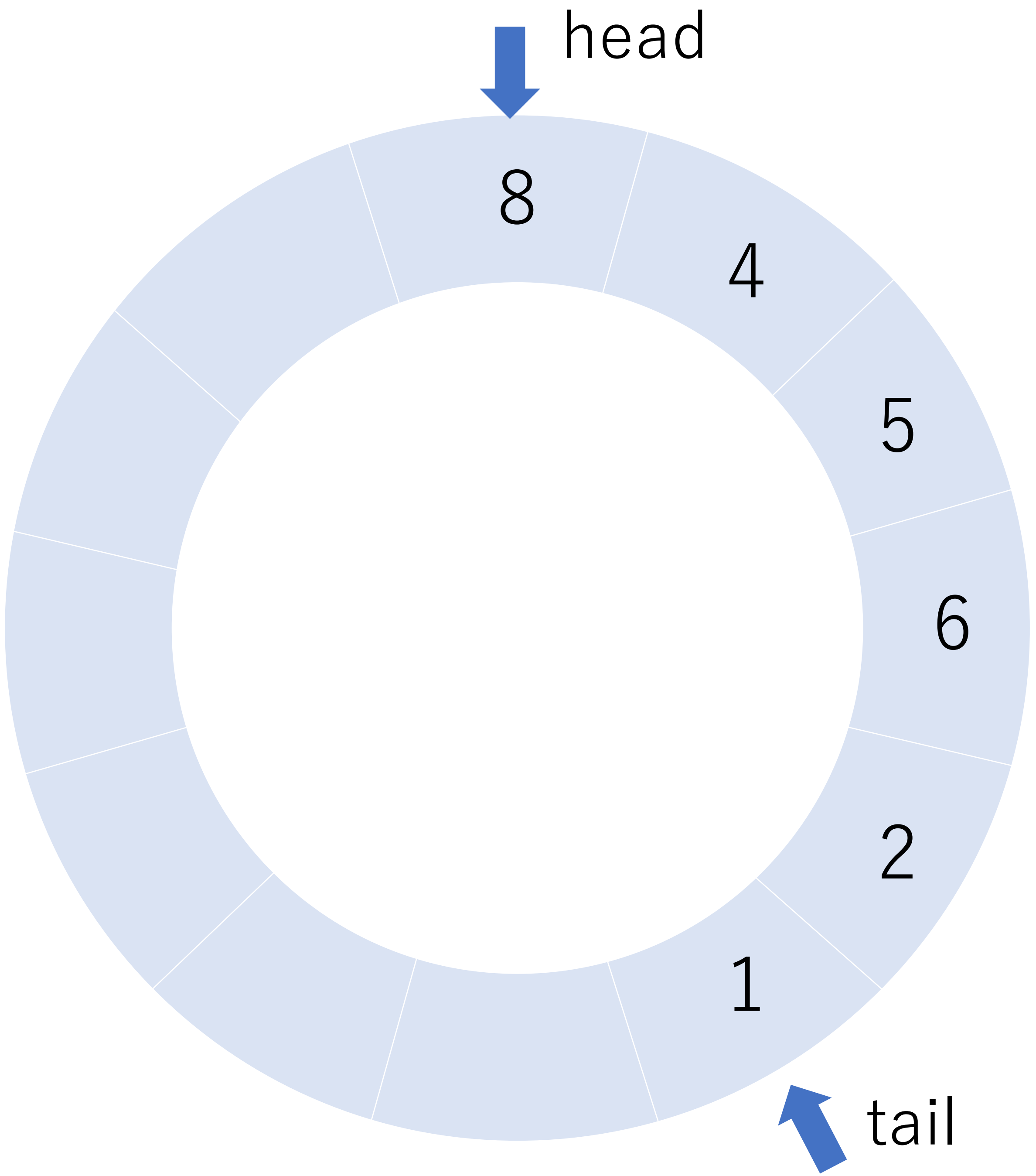
最新のX個の情報を覚えておく，みたいな使い方もできる.



リングバッファ

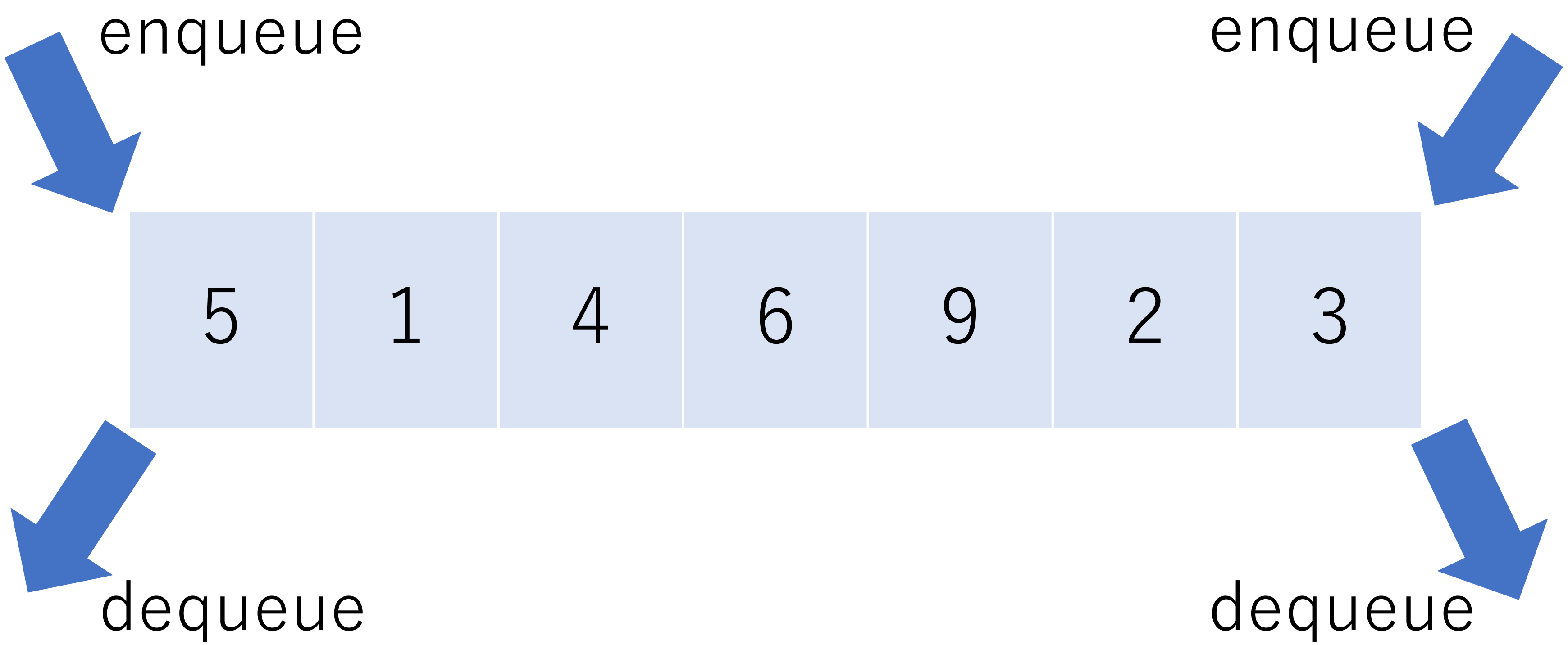
リングバッファを使うと,
バッファのサイズを固定に
できる.

空間計算量を固定にしながら,
未知のクエリ数にも対応する
ことができる実装になる.



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．

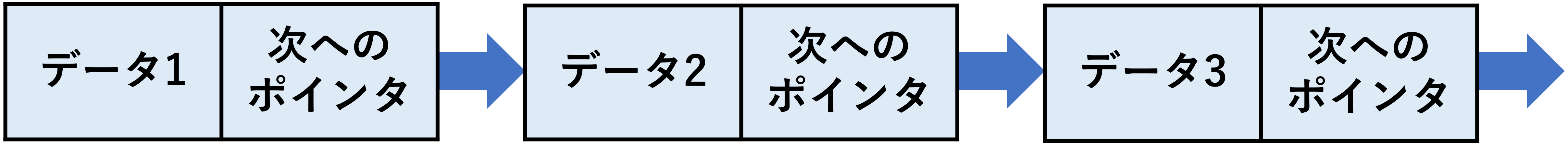


線形リスト

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．

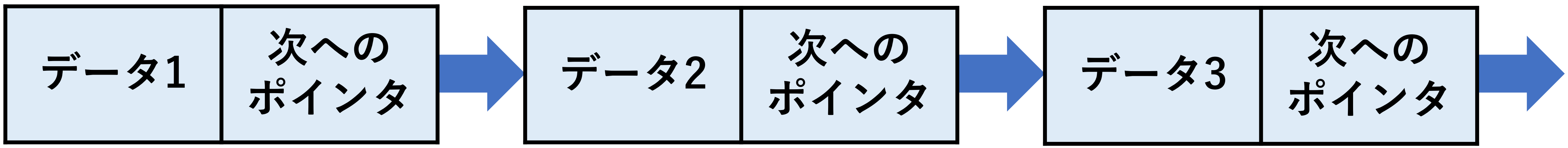
双方向や循環になっているものある．



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．

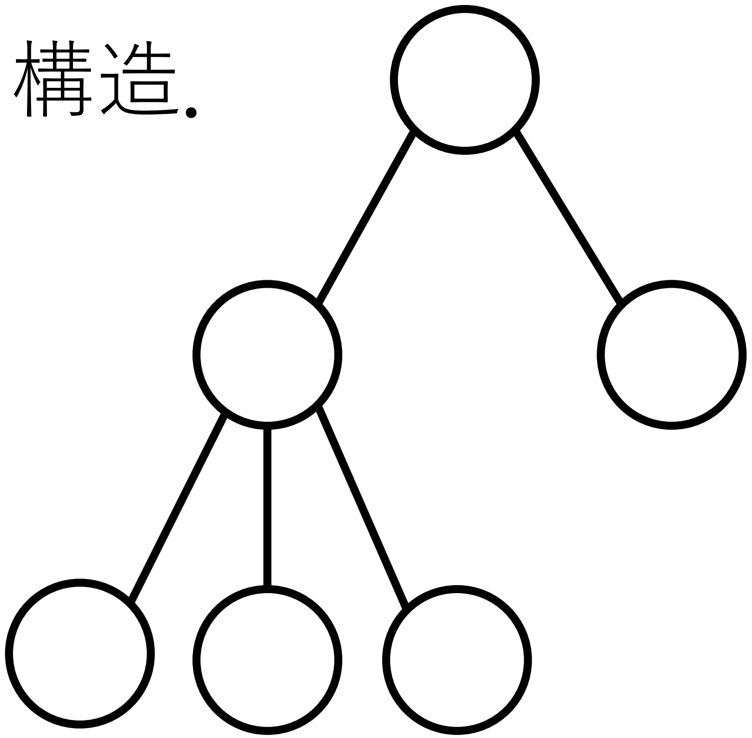


木構造（ツリー）

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.

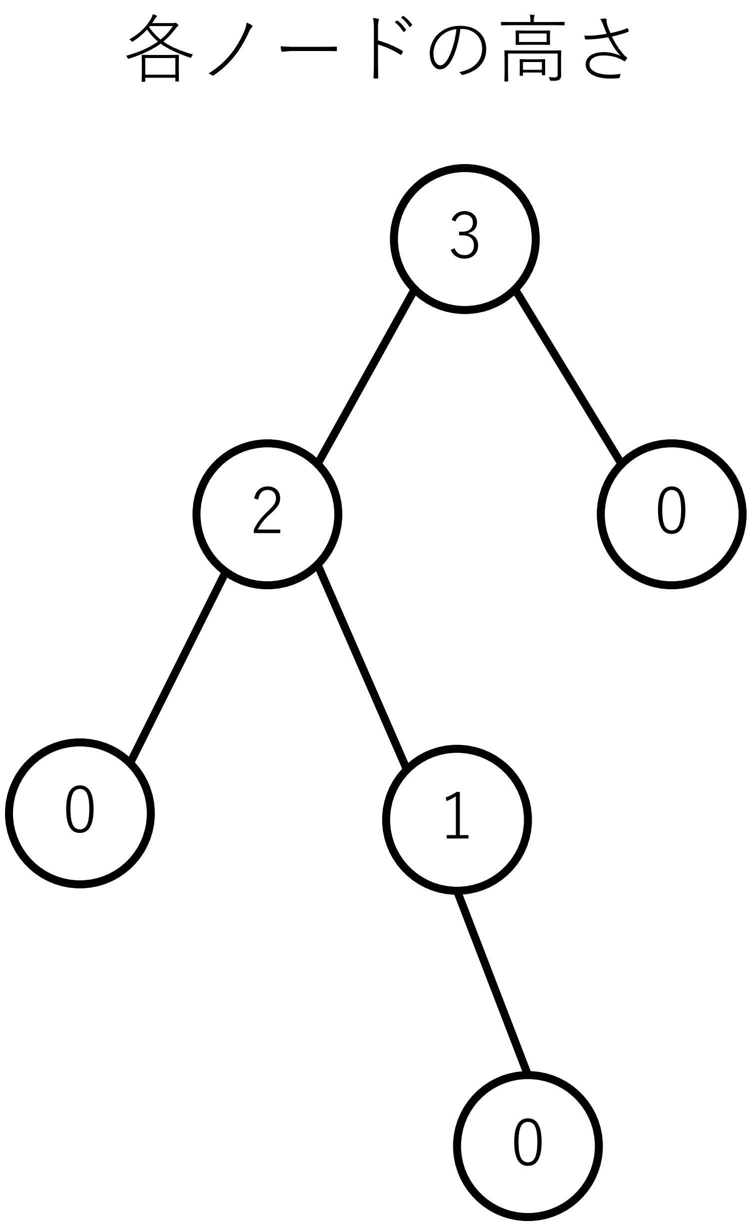
一番下（より子供のノードがない）のノードを葉（leaf）という.



木構造（ツリー）

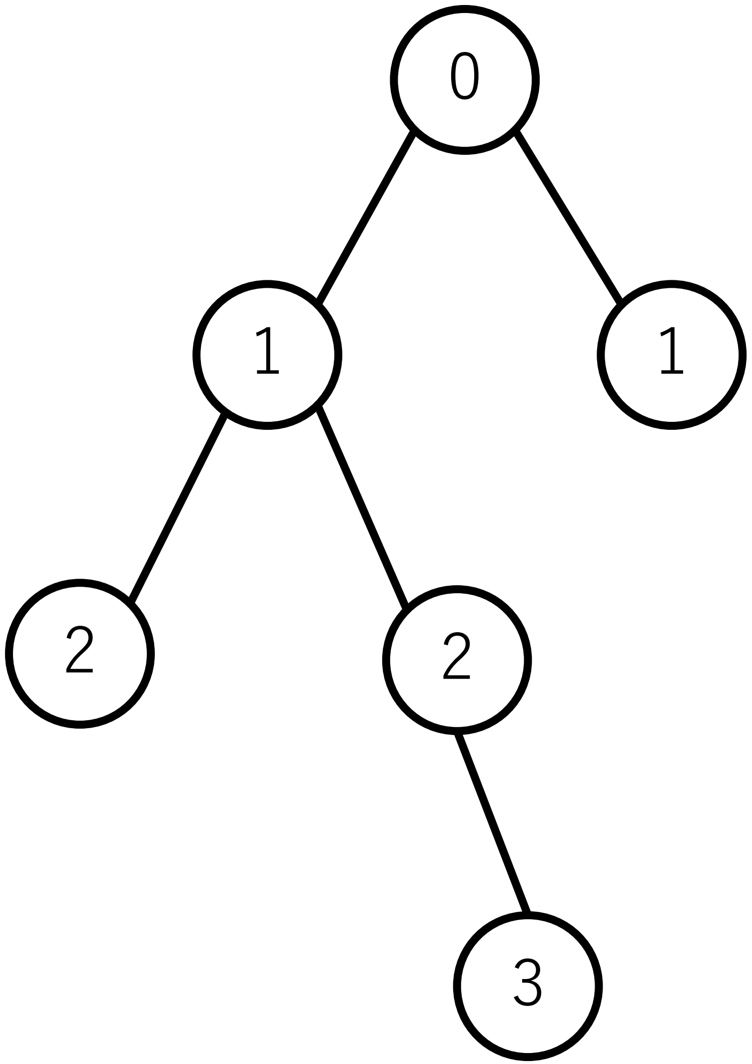
高さ

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）

各ノードの深さ

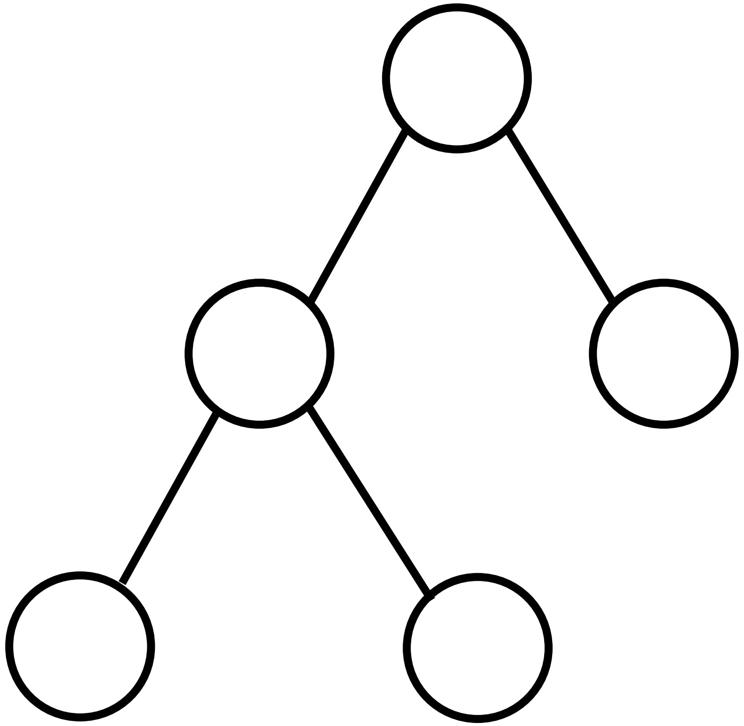


深さ

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.

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.

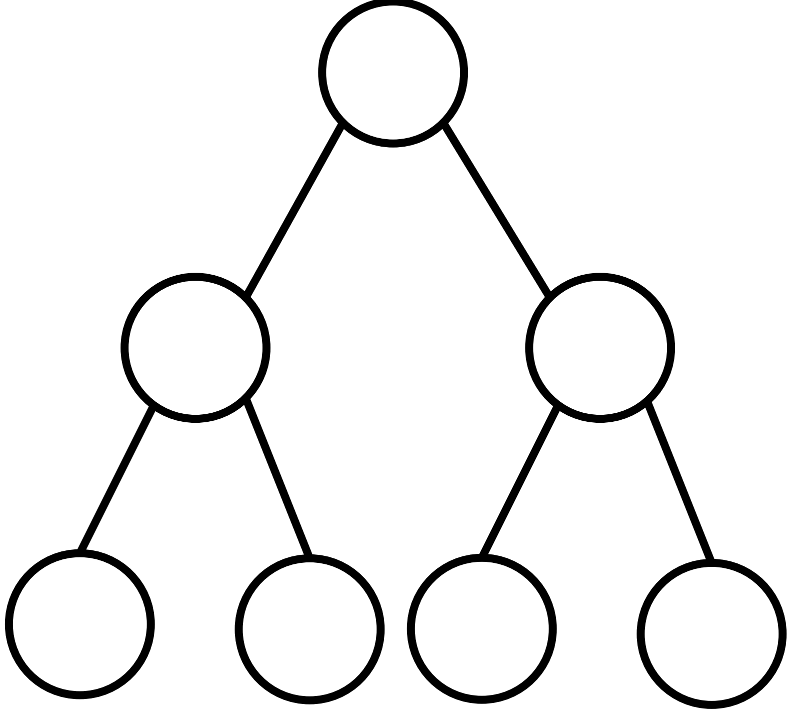


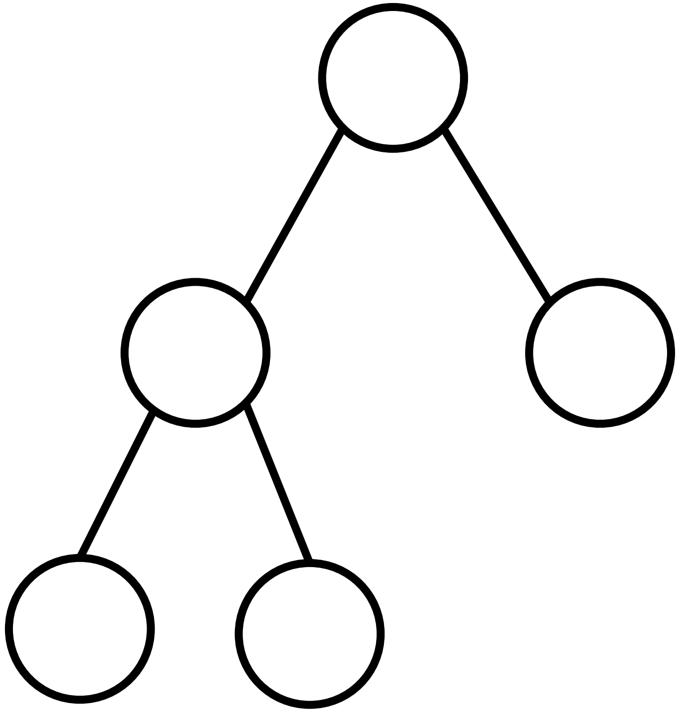
完全二分木

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.

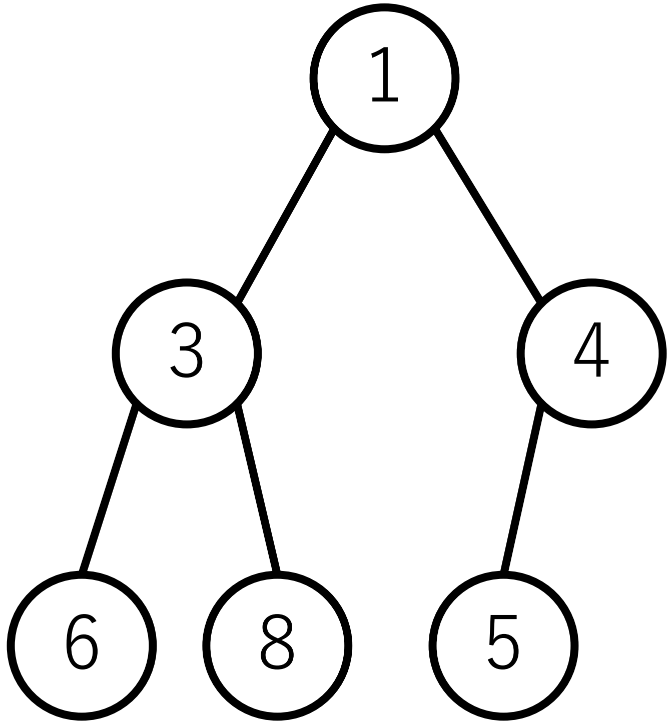
根ノードは常に最大値（最小値）になる.

二分ヒープ

以下の2つの制約を満たすヒープ.

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）.

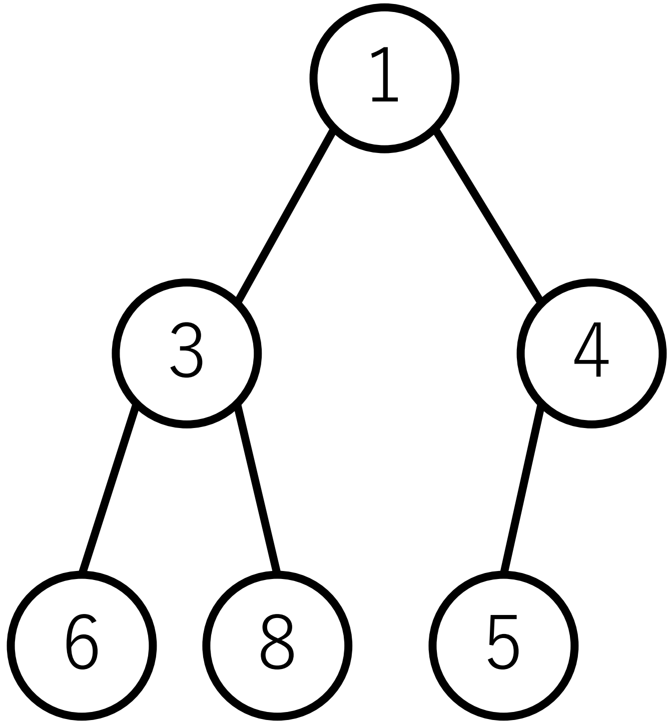
ツリーの形がcomplete binary tree.（たまたまfull binary treeになっていることもある）



二分ヒープ

いろんな形のヒープがあり得るが,
単に「ヒープ」と言った場合は,
二分ヒープを指すことが多い.

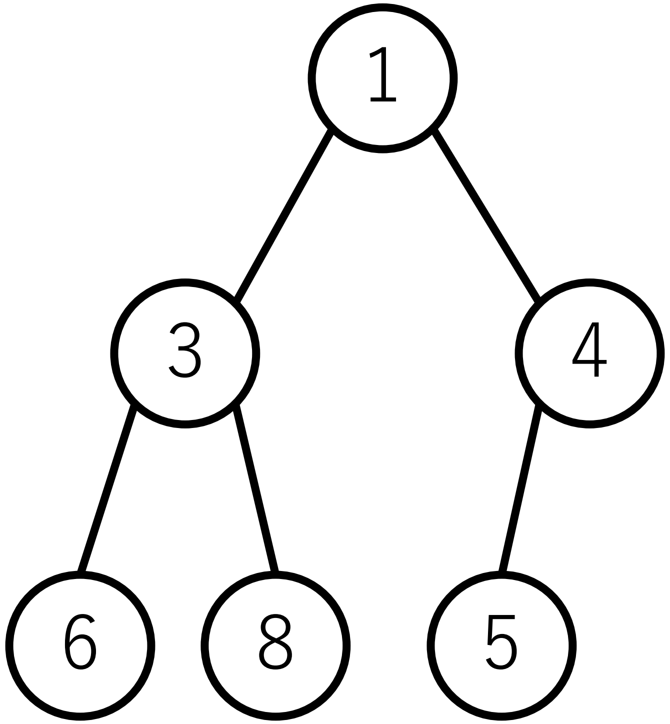
以降のスライドでも, 「ヒープ」
という表現は二分ヒープを意味
するものとしています.



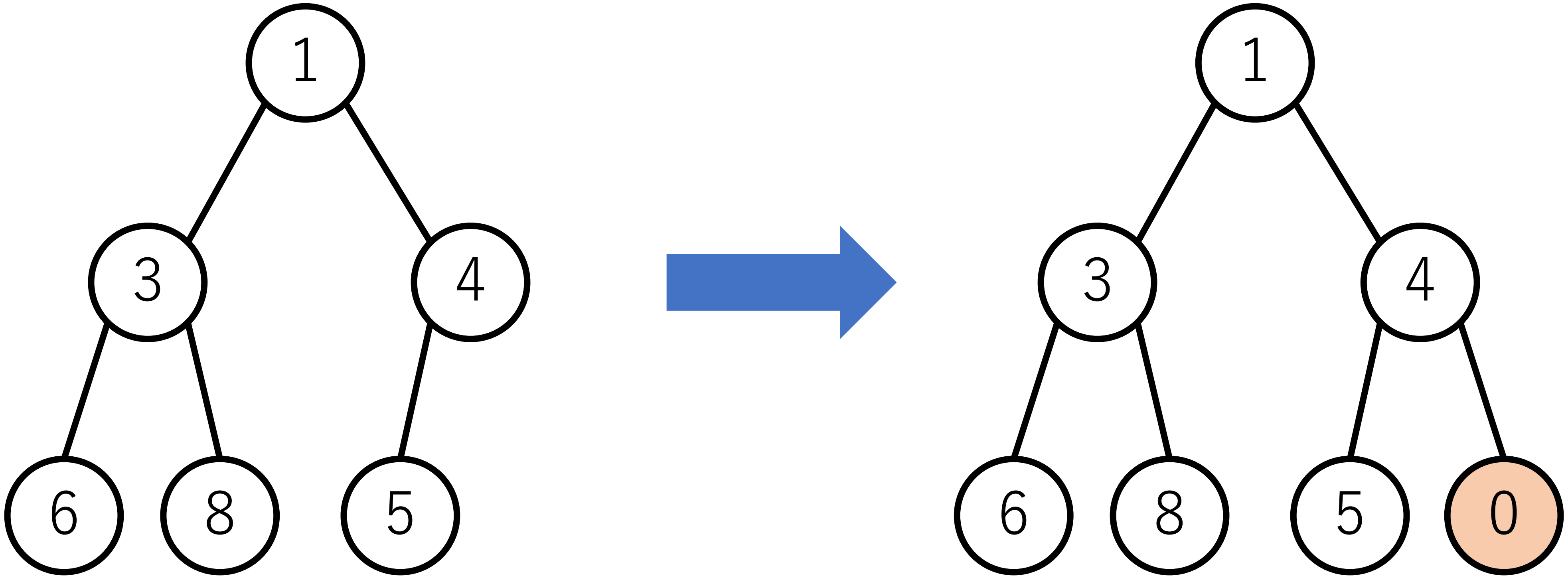
ヒープの操作：追加

#1 空いている一番左に葉として追加.

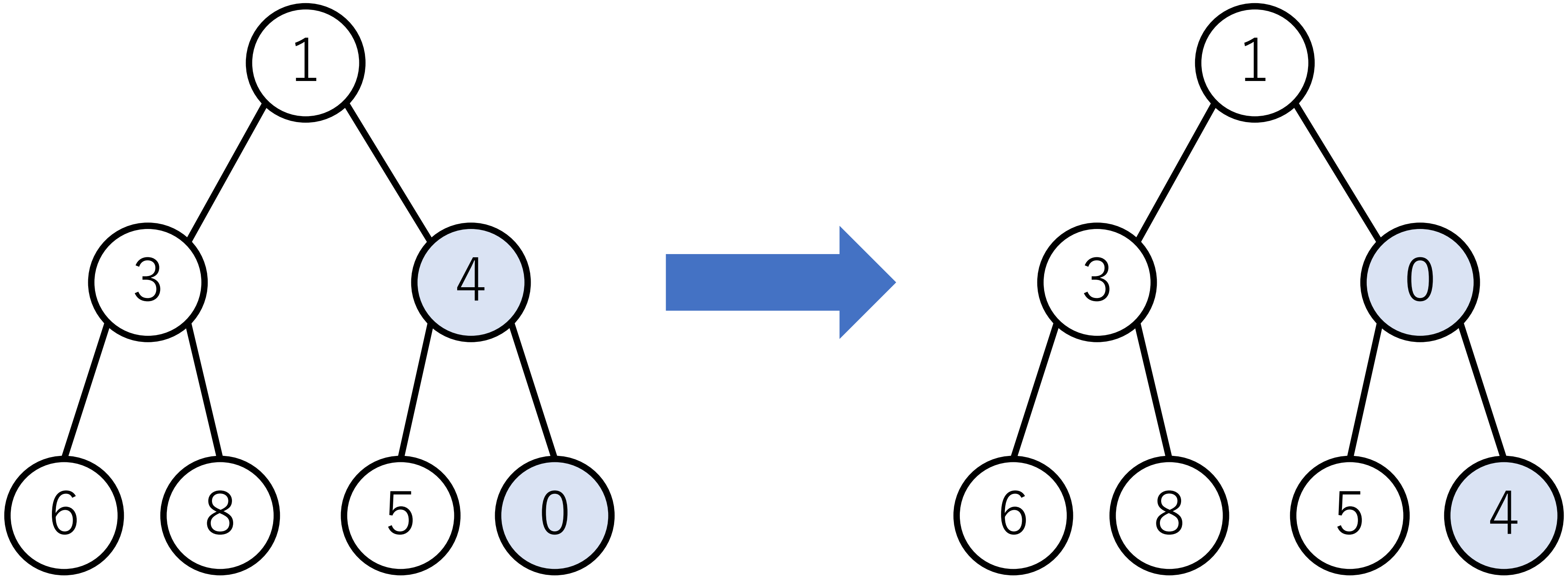
#2 親と比較．制約条件を満たすように順次入れ替える.



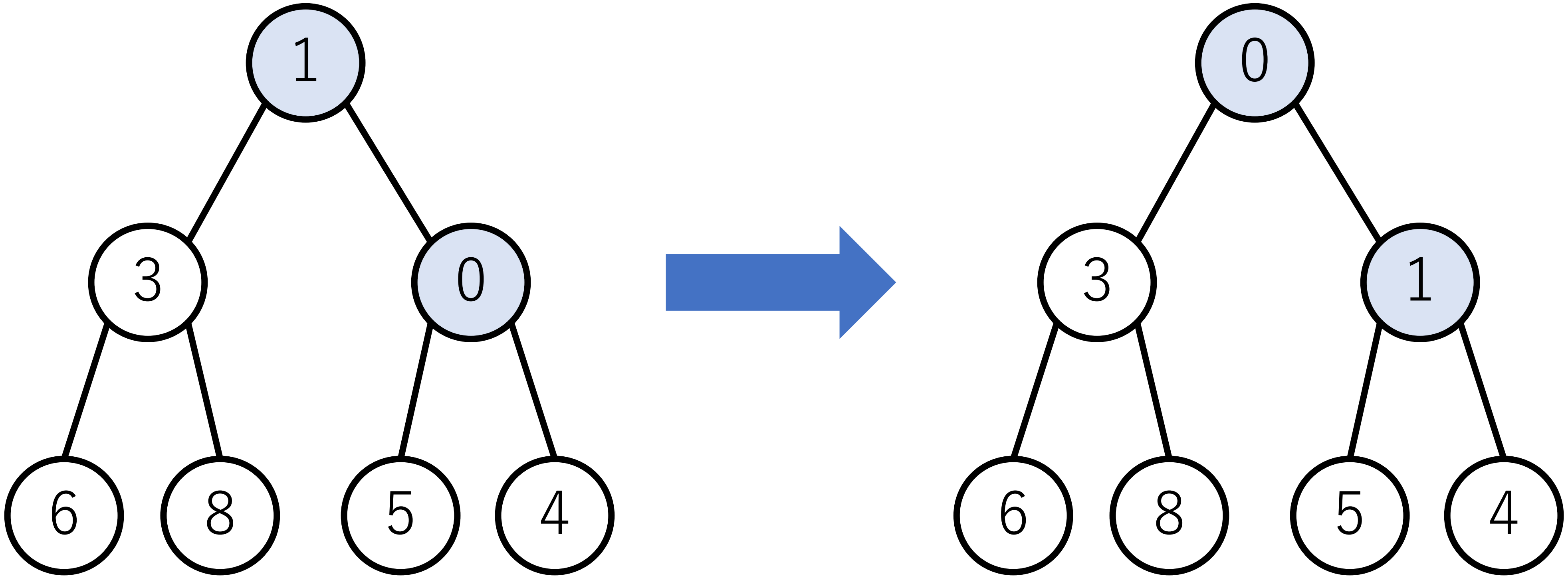
ヒープの操作：0を追加



ヒープの操作：0を追加



ヒープの操作：0を追加



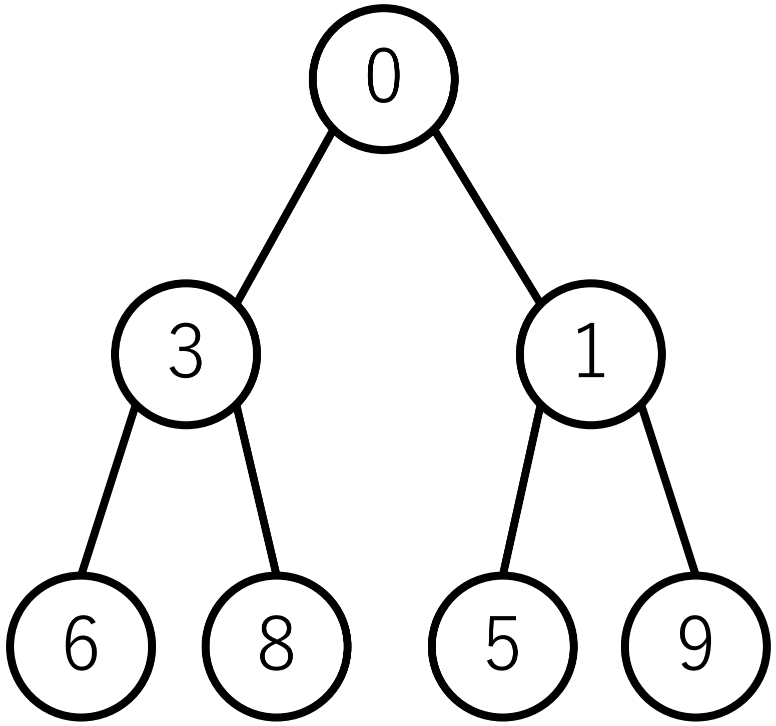
ヒープの操作：削除

#1 rootを取り除く.

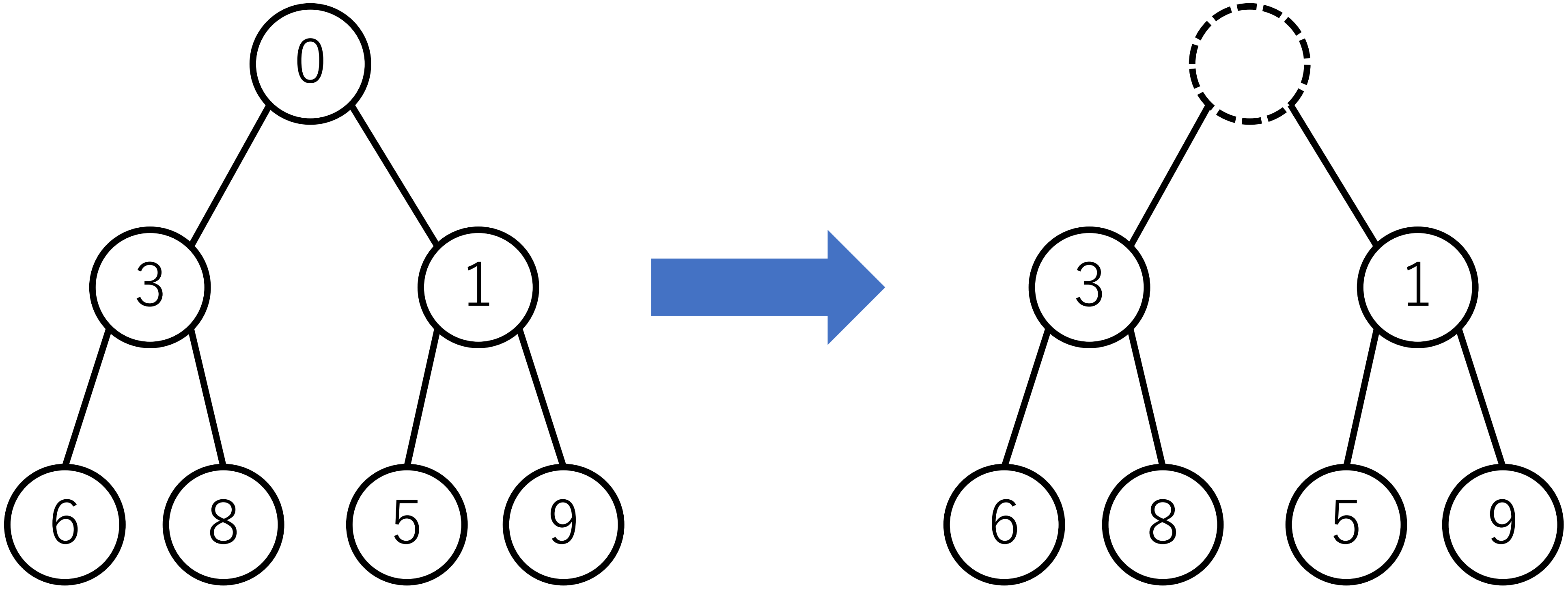
#2 一番右端にいる葉をrootにする.

#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.

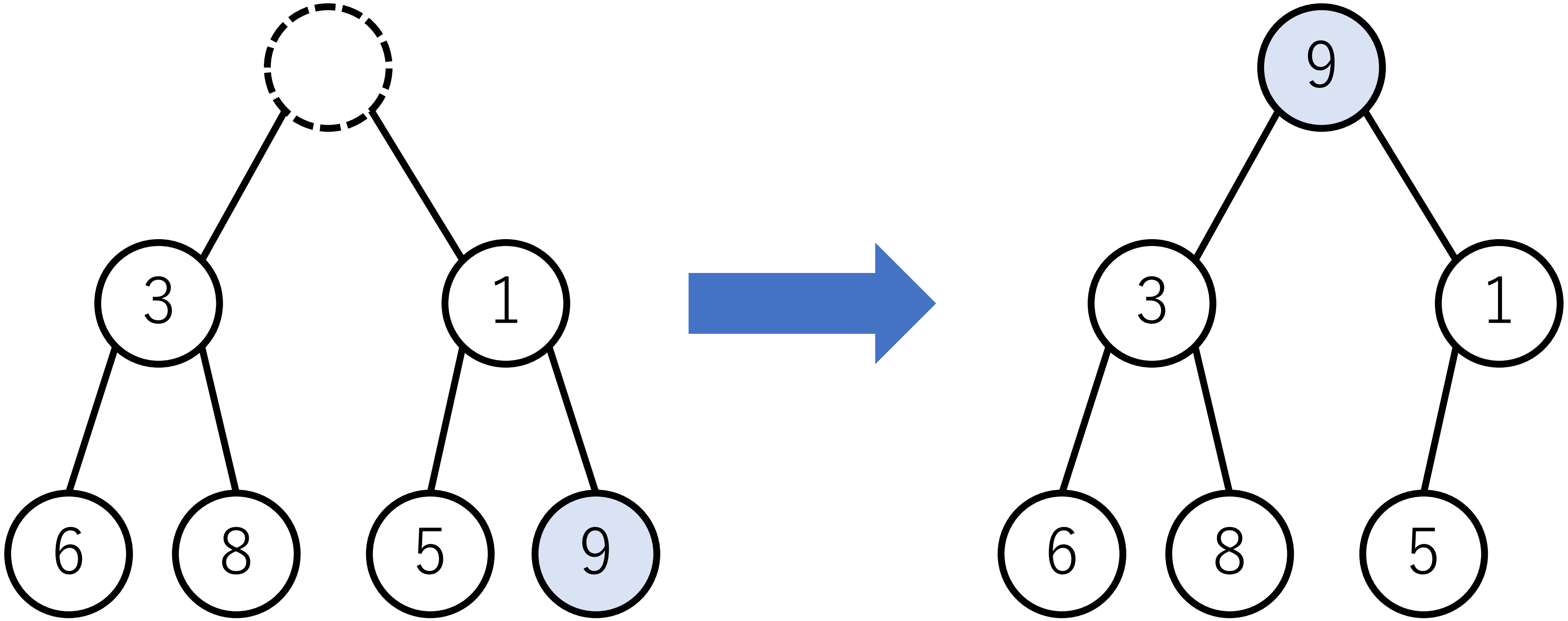
#4 制約条件を満たすまで入れ替えを続ける.



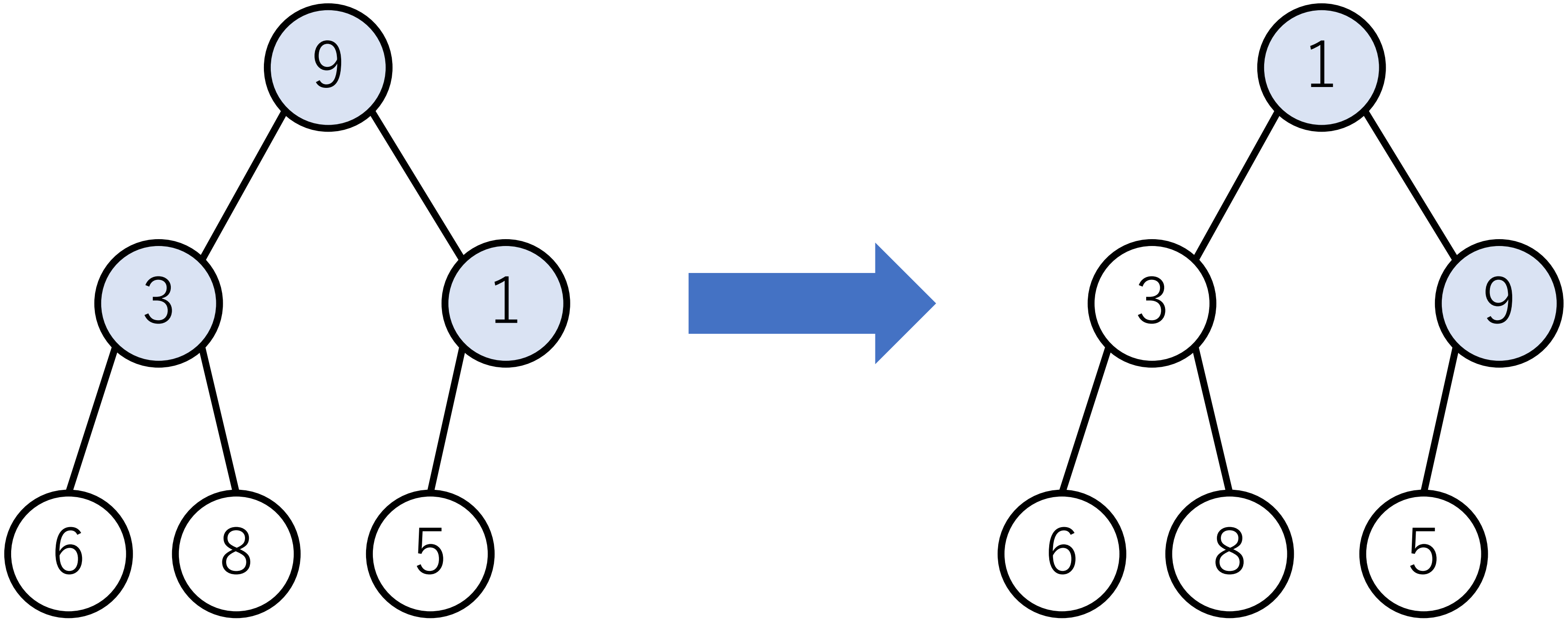
ヒープの操作：削除



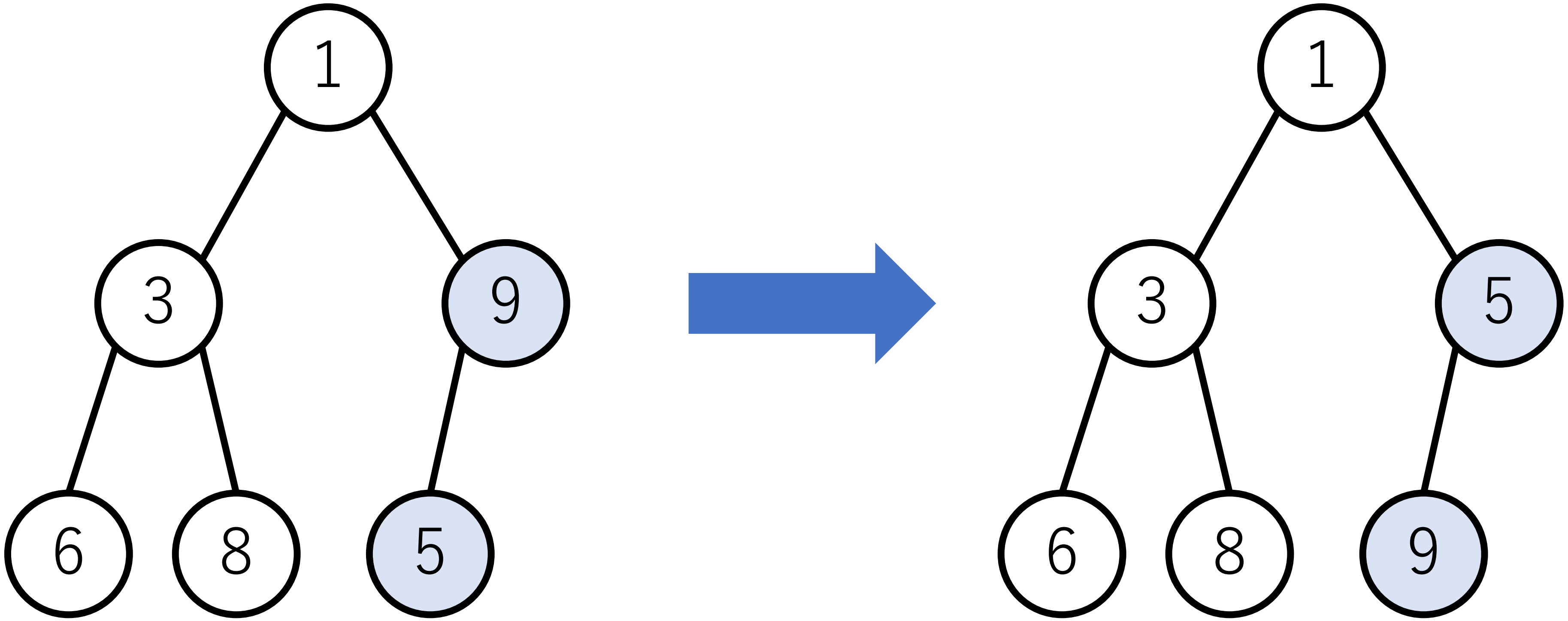
ヒープの操作：削除



ヒープの操作：削除



ヒープの操作：削除

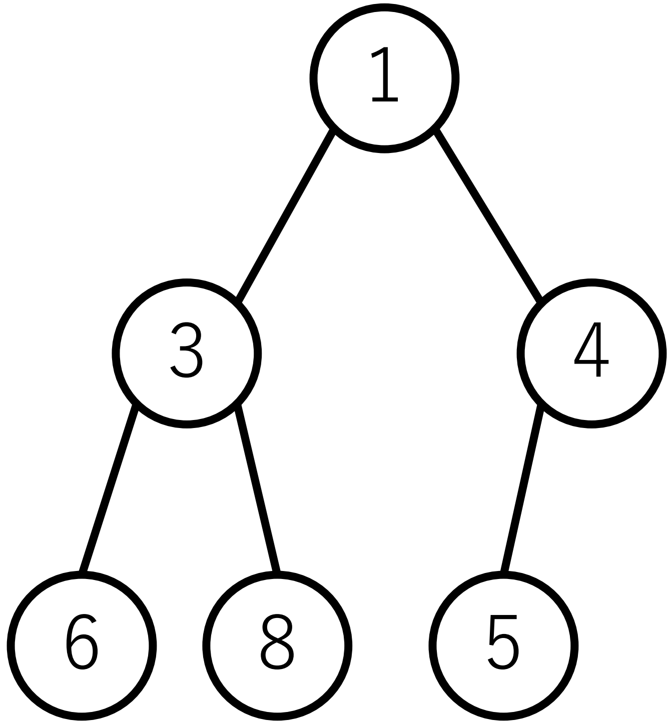


ヒープ

上下関係と木の形の制約が必ず守られていることが重要.

　最小ヒープなら親ノードは子ノードよりも必ず小さい.

左右の部分木の関する制約は特にない.



二分ヒープの配列での実装

ノードa[i]に対して,

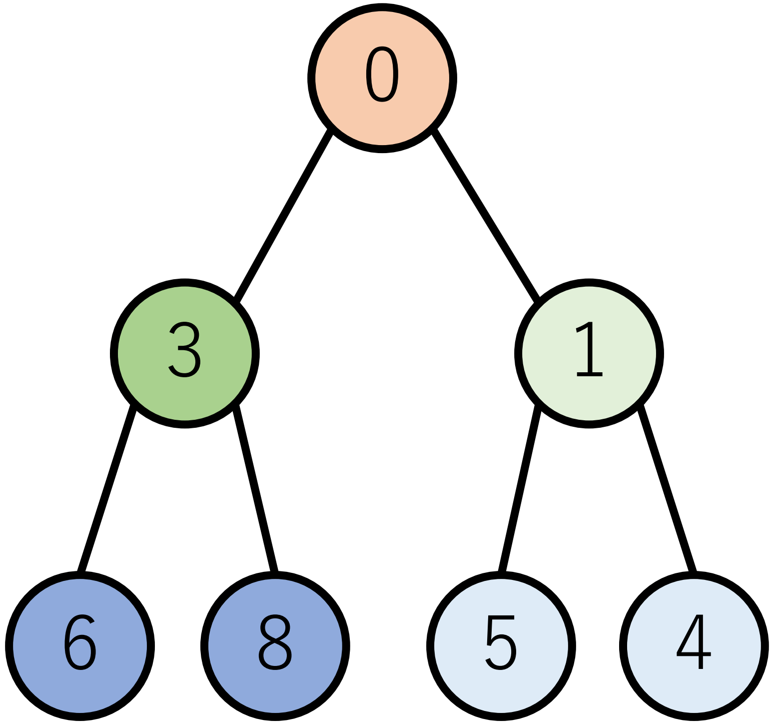
親: a[i//2]

左の子: a[2*i]

右の子: a[2*i+1]

となるようにデータを格納.

（a[0]は使わない，//は切り捨て除算）



構造体を使うことなく表現することが可能.

-	0	3	1	6	8	5	4

ヒープの実装例（最小ヒープ）

```
class MyHeap:
    def __init__(self, size):
        self.inf = 10**9          # 十分に大きい数
        self.size = size + 1
        # ヒープを構成する配列
        self.array = [self.inf]*self.size
        self.last = 0  # 現在までに入っているデータ数
```

ヒープの実装例（最小ヒープ）

```
class MyHeap:
    def add(self, value: int):
        if self.last != self.size:
            # 一番右の葉ノードとして追加
            self.last += 1
            self.array[self.last] = value
            # 制約を満たしているかチェックをする
            self.check_after_add(self.last)
```

ヒープの実装例（最小ヒープ）

```
class MyHeap:
    def remove(self):
        if self.last != 0:
            removed = self.array[1]
            # 一番右の葉ノードを根ノードに移動
            self.array[1] = self.array[self.last]
            self.array[self.last] = self.inf
            self.last -= 1
```

ヒープの実装例（最小ヒープ）

```
class MyHeap:
        def remove(self):
                if self.last != 0:
                        …
                        # 制約を満たしているかチェックをする
                        self.check_after_remove(self.last)
                        return removed
```

ヒープの実装例（最小ヒープ）

```
class MyHeap:
    def check_after_add(self, i):
        if i < 2: return # 根ノードまで行ったら終了

        if [ノードiとその親ノードを比較し，
        親ノードの方が大きい]:
            [ノードiとその親ノードをスワップ]
            [check_after_addを再帰で呼び出す
             （引数は？）]
```

ヒープの実装例

```
class MyHeap:
        def check_after_remove(self, i):
                # 子ノードのうち，より大きい方と比較をし，
                # 制約を満たすようにスワップ．

                # 根ノードから順に辿り，葉ノードまで行く．

                # check_after_addと同じく再帰で呼び出す．
                # （引数は？）また，再帰の終了条件は？
```

ヒープの計算量

追加の場合，高さ/2 回分の比較・入替が平均的には必要（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.

次はすこし発展的な内容

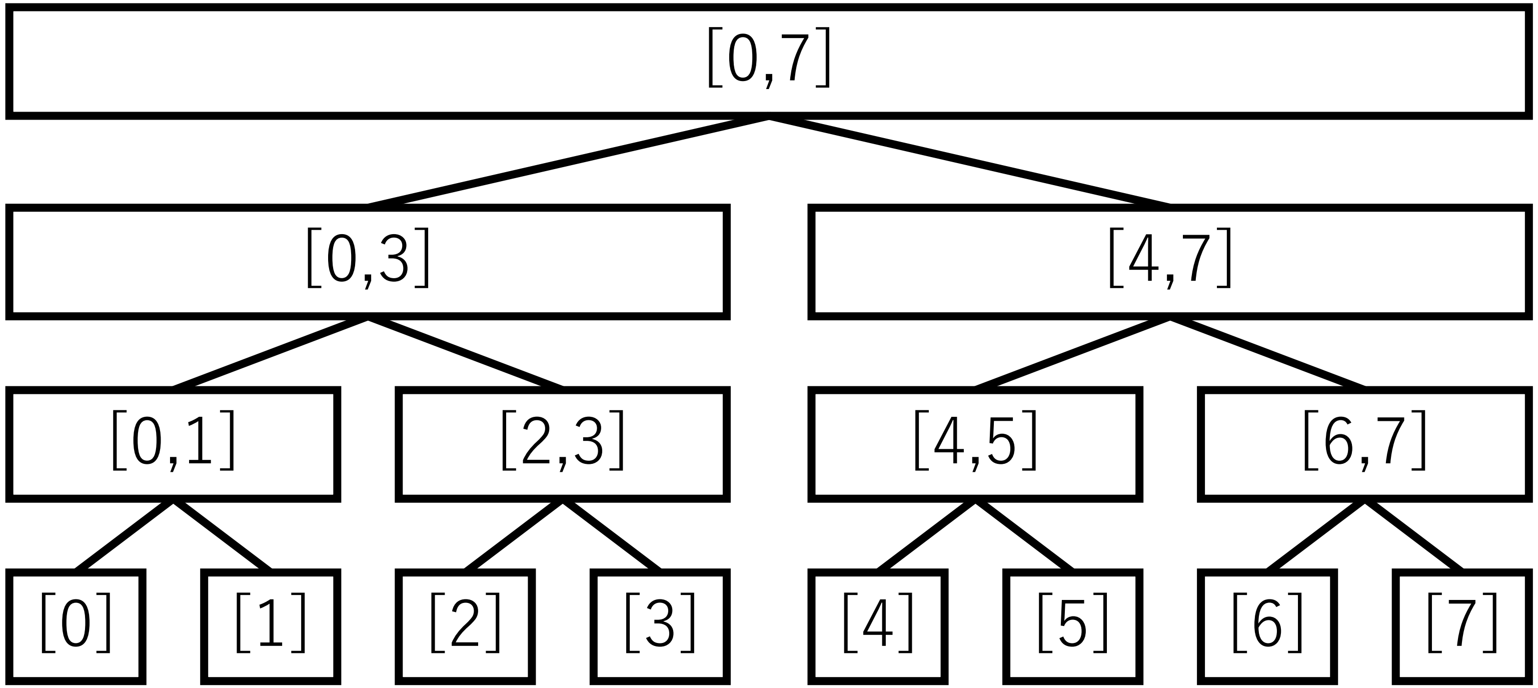
アルゴリズム初学者の人は上記内容がしっかりと
理解できていれば問題ありません. 😌

基本課題もここまでのスライドの内容ですので,
しっかりと復習していただければと思います.

以下はちょっと変わったデータ構造のご紹介.

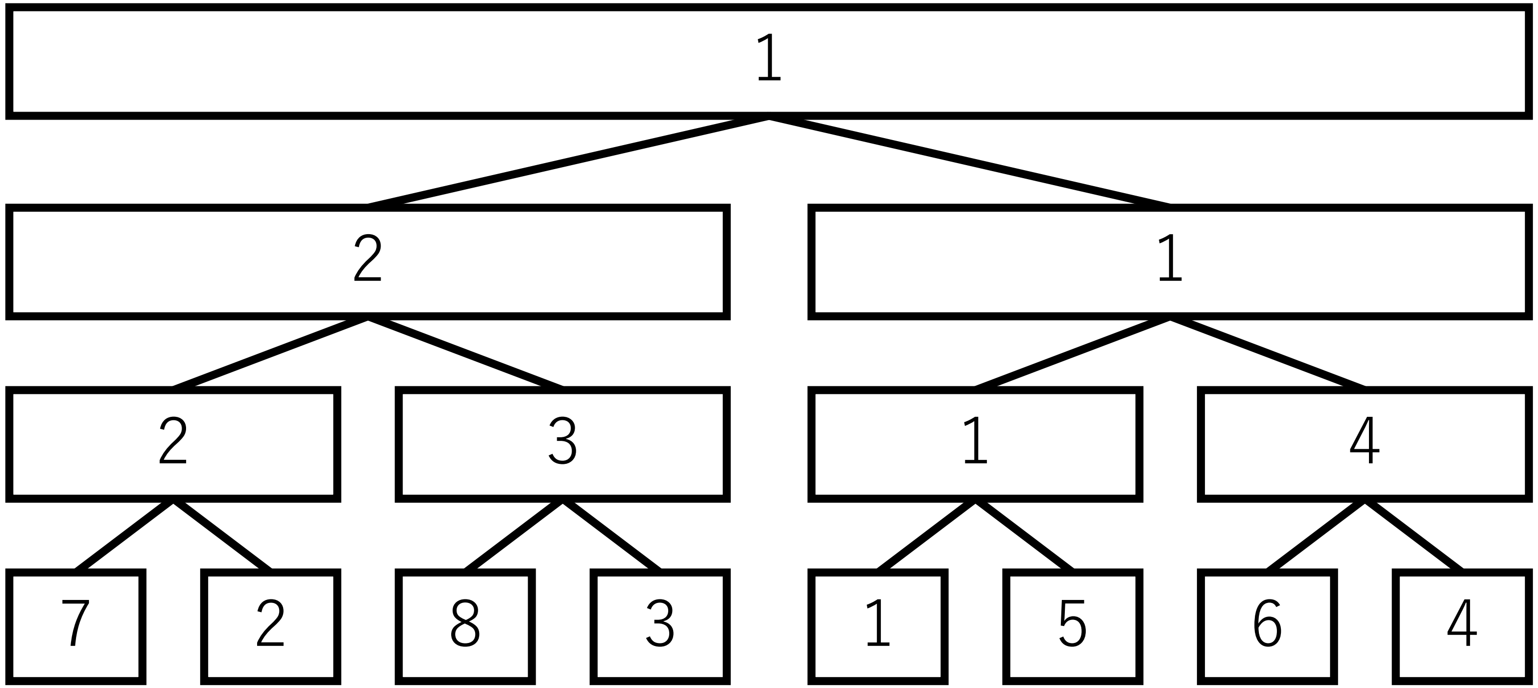
セグメント木

各ノードが子ノードの区間に関するある情報を保持する二分木．ある区間のクエリに対する応答を考える時に便利．二分ヒープのように配列で実装できる．



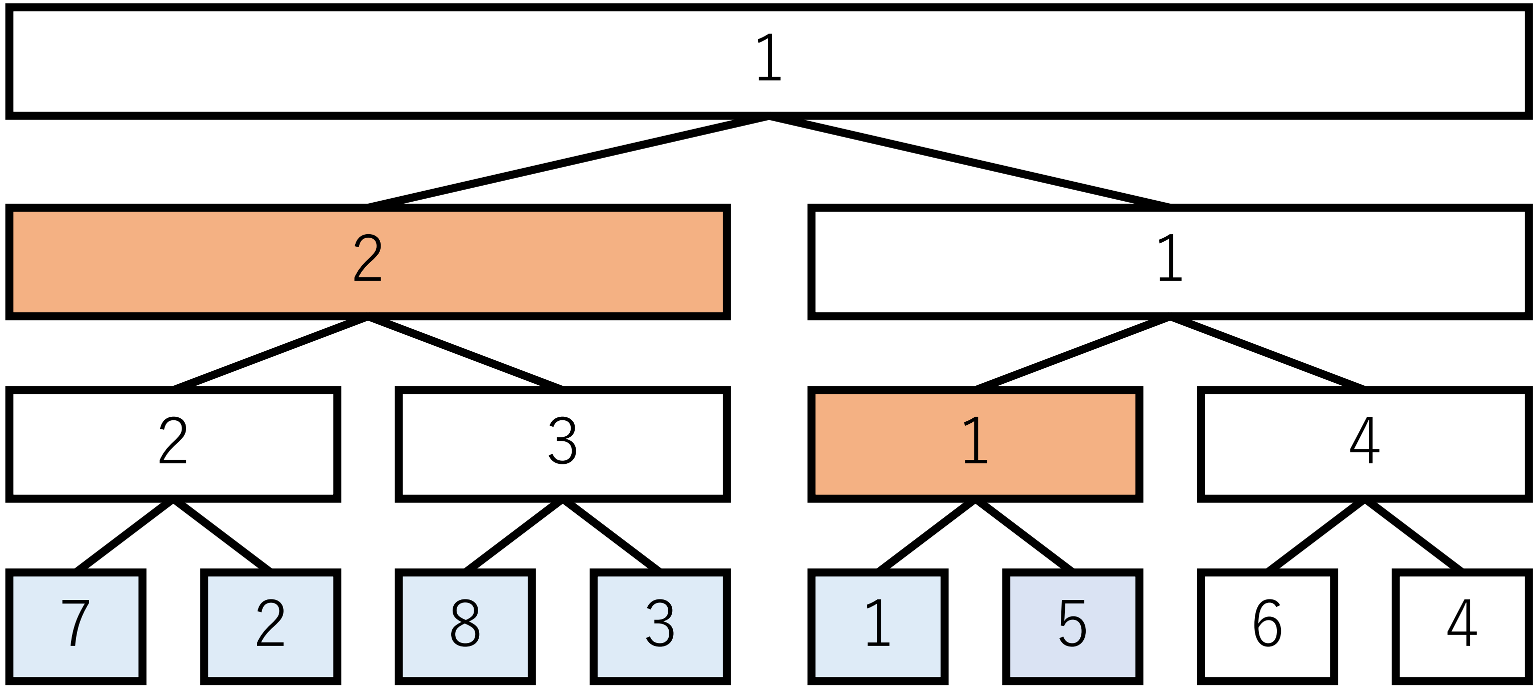
セグメント木

例）配列[7, 2, 8, 3, 1, 5, 6, 4]の最小値のセグメント木.



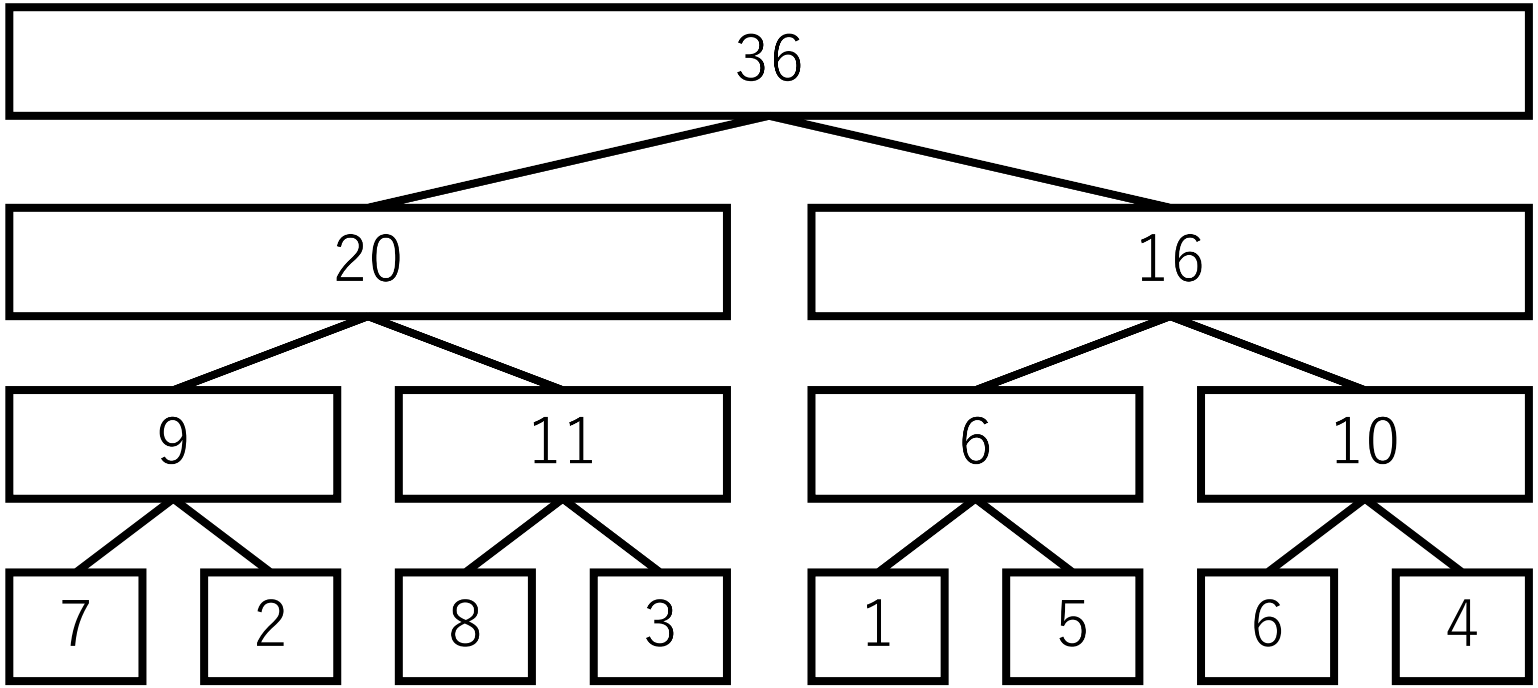
セグメント木

例）配列[7, 2, 8, 3, 1, 5, 6, 4]の最小値のセグメント木.
a[0]からa[5]の最小→オレンジのノードを見て1.



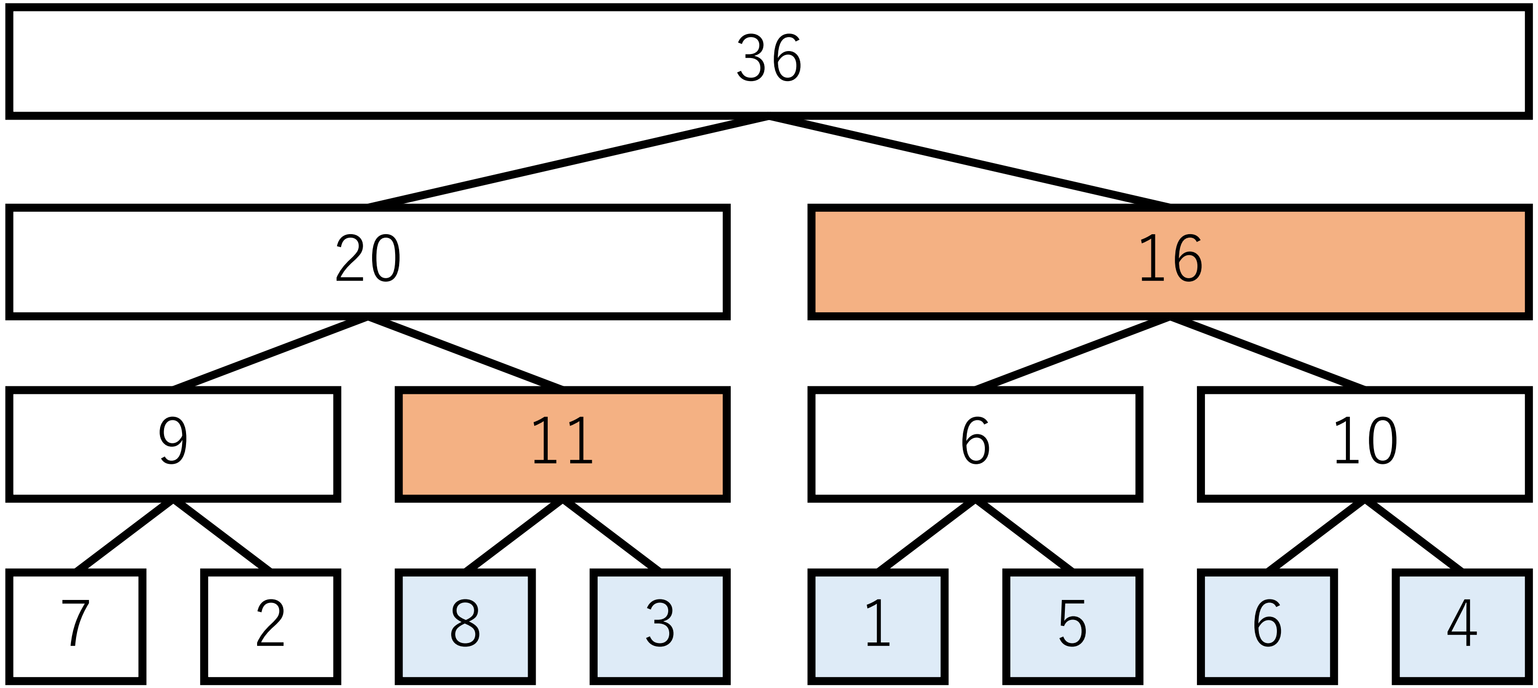
セグメント木

例）配列[7, 2, 8, 3, 1, 5, 6, 4]の区間和のセグメント木.



セグメント木

例）配列[7, 2, 8, 3, 1, 5, 6, 4]の区間和のセグメント木.
a[2]からa[7]の区間和→オレンジのノードを見て27.



セグメント木の実装

実装としては，二分ヒープを配列で実装する場合に
比較的よく似ている．

ノードa[i]に対して，

親: a[i//2]

左の子: a[2*i]

右の子: a[2*i+1]

となるようにデータを格納．（a[0]は使わない）

一番下の階層が与えられた配列そのままになる．

セグメント木の実装

完全二分木（full binary tree）を利用して実装されることが多い.

- セグメント木自体はfull binary treeやcomplete binary treeである必要はない.
- 平衡二分木（次回紹介）を使う実装もある.

以下の実装例でもfull binary treeを使ったものを紹介.

セグメント木の実装例（区間和）

```
class CSumSegTree:
    # 整数nよりも大きい最小の2のべき乗を求める.
    def get_2pow(self, n):
        ret = 1
        while ret < n:
            ret *= 2
        return ret
```

セグメント木の実装例（区間和）

```
class CSumSegTree:
    # seq: 一番最初に与えられる配列
    def __init__(self, seq):
        # 完全二分木として実装
        size = self.get_2pow(len(seq))
        # 0番目は使わない
        self.array = [0 for i in range(2*size)]
        self.leaf_start = size
```

セグメント木の実装例（区間和）

```
class CSumSegTree:
    def __init__(self, seq):
        ……
        # 葉ノードの位置に与えられた配列を格納
        for i in range(self.leaf_start, self.leaf_start + len(seq)):
            self.array[i] = seq[i - self.leaf_start]

        self.initialize()   # セグメント木の構築
```

セグメント木の実装例（区間和）

```
class CSumSegTree:
    def initialize(self):      # セグメント木の構築
        start_i = self.leaf_start
        # 下の層から順に値を計算し格納
        while start_i > 1:
            for i in range(start_i, start_i * 2, 2):
                parent_i = i // 2
                self.array[parent_i] = self.array[i] +
                                                    self.array[i+1]
            start_i = start_i // 2
```

セグメント木の実装例（区間和）

```
class CSumSegTree:
    def update(self, i, val):        # 配列の要素の更新
        node_i = i + self.leaf_start
        self.array[node_i] = val     # 葉ノードの更新

        while node_i > 1:       # 上に遡って更新
            parent_i = node_i // 2
            self.array[parent_i] = self.array[parent_i*2] +
                                   self.array[parent_i*2+1]
            node_i = parent_i
```

セグメント木の実装例（区間和）

```
class CSumSegTree:
    # 部分和を求める関数
    def findSum(self, l, r, k=1, le=0, re=-1):

    # l: 指定する区間の左端，r: 指定する区間の右端
    # 半開区間として指定（l番目からr-1番目の部分和）
    # k: self.arrayのインデックス
    # le: self.array[k]に保持されている部分和の区間の左端
    # re: self.array[k]に保持されている部分和の区間の右端
```

セグメント木の実装例（区間和）

```
class CSumSegTree:
        def findSum(self, l, r, k=1, le=0, re=-1):

                # 最初に呼び出した時は全区間（根ノード）
                if re==-1: re = self.leaf_start -1
```

セグメント木の実装例（区間和）

```
class CSumSegTree:
    def findSum(self, l, r, k=1, le=0, re=-1):
        if re==-1: re = self.leaf_start -1

        # 指定する範囲以外なら0を返す
        if (re < l) or (r <= le): return 0
```

セグメント木の実装例（区間和）

```
class CSumSegTree:
    def findSum(self, l, r, k=1, le=0, re=-1):
        if re==-1: re = self.leaf_start -1
        if (re < l) or (r <= le): return 0

        # 指定する範囲に完全に含まれているなら,
        # その値を利用する
        if (l <= le) and (re < r): return self.array[k]
```

セグメント木の実装例（区間和）

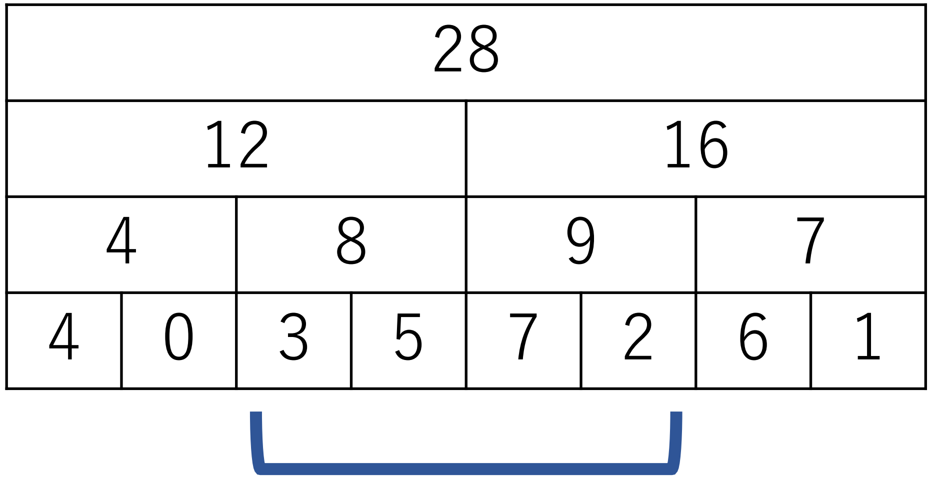
```
class CSumSegTree:
        def findSum(self, l, r, k=1, le=0, re=-1):
                ……
                # 子ノードに下がる．sum_lは左側，sum_rは
                # 右側の子ノードを見ていることになる．
                sum_l = self.findSum(l, r, 2*k, le, (le+re)//2)
                sum_r = self.findSum(l, r, 2*k+1, (le+re)//2+1, re)

                return sum_l + sum_r
```

セグメント木をつかった区間和計算例

```
seq = [4, 0, 3, 5, 7, 2, 6, 1]
segtree = CSumSegTree(seq)
print(segtree.findSum(2, 6))
---
```

実行結果

17



セグメント木をつかった区間和計算例

self.array：[0, 28, 12, 16, 4, 8, 9, 7, 4, 0, 3, 5, 7, 2, 6, 1]

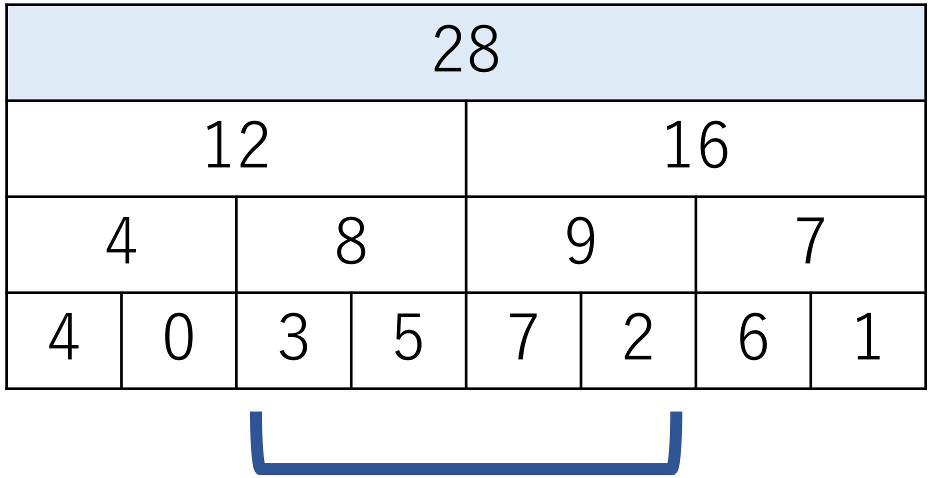
findSum(2, 6, 1, 0, -1)
findSum(2, 6, 2, 0, 3)
findSum(2, 6, 4, 0, 1)
findSum(2, 6, 5, 2, 3)
findSum(2, 6, 3, 4, 7)
findSum(2, 6, 6, 4, 5)
findSum(2, 6, 7, 6, 7)

<table>
<tr><td colspan="8">28</td></tr>
<tr><td colspan="4">12</td><td colspan="4">16</td></tr>
<tr><td colspan="2">4</td><td colspan="2">8</td><td colspan="2">9</td><td colspan="2">7</td></tr>
<tr><td>4</td><td>0</td><td>3</td><td>5</td><td>7</td><td>2</td><td>6</td><td>1</td></tr>
</table>

セグメント木をつかった区間和計算例

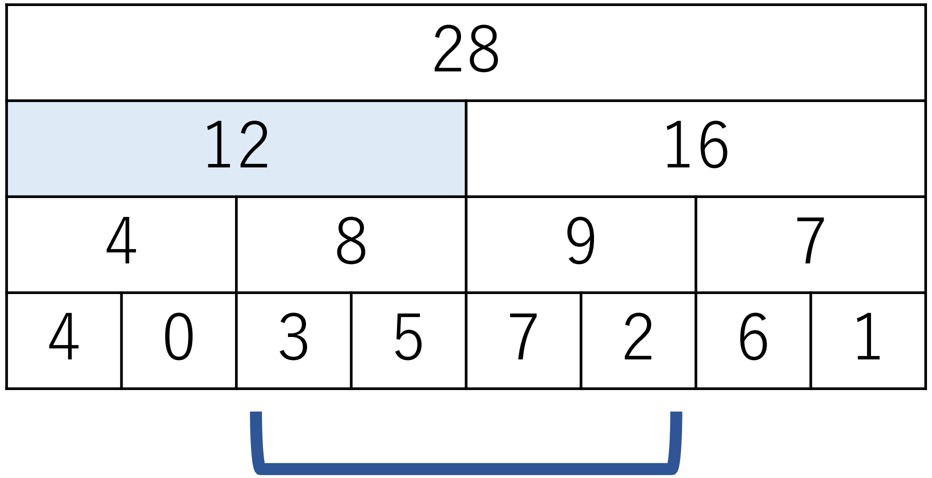
self.array：[0, 28, 12, 16, 4, 8, 9, 7, 4, 0, 3, 5, 7, 2, 6, 1]

findSum(2, 6, 1, 0, -1) # 一部含む
findSum(2, 6, 2, 0, 3)
findSum(2, 6, 4, 0, 1)
findSum(2, 6, 5, 2, 3)
findSum(2, 6, 3, 4, 7)
findSum(2, 6, 6, 4, 5)
findSum(2, 6, 7, 6, 7)



セグメント木をつかった区間和計算例

self.array：[0, 28, 12, 16, 4, 8, 9, 7, 4, 0, 3, 5, 7, 2, 6, 1]



findSum(2, 6, 1, 0, -1)

findSum(2, 6, 2, 0, 3) # 一部含む

findSum(2, 6, 4, 0, 1)

findSum(2, 6, 5, 2, 3)

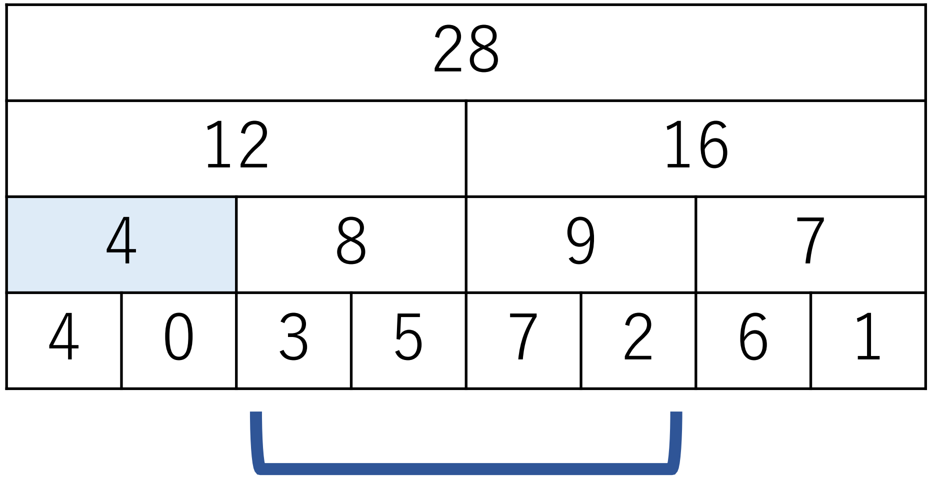
findSum(2, 6, 3, 4, 7)

findSum(2, 6, 6, 4, 5)

findSum(2, 6, 7, 6, 7)

セグメント木をつかった区間和計算例

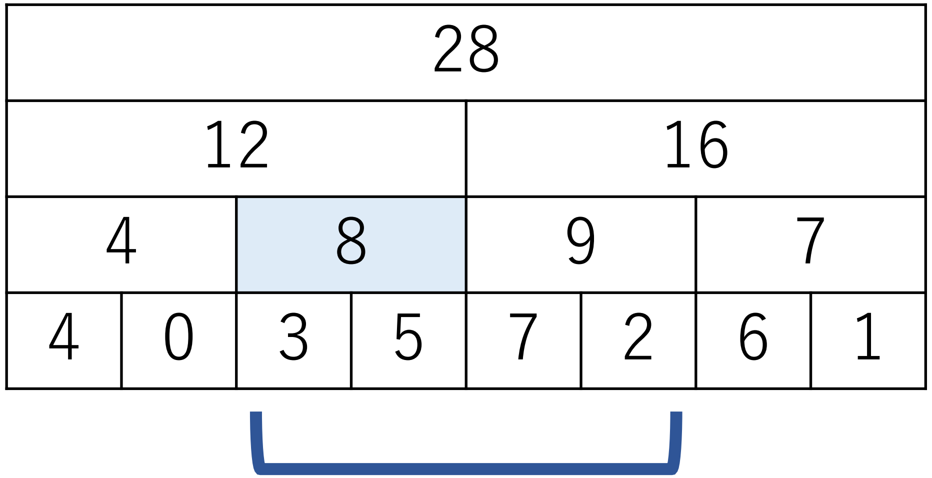
self.array：[0, 28, 12, 16, 4, 8, 9, 7, 4, 0, 3, 5, 7, 2, 6, 1]



findSum(2, 6, 1, 0, -1)
findSum(2, 6, 2, 0, 3)
findSum(2, 6, 4, 0, 1) # 全く含まない
findSum(2, 6, 5, 2, 3)
findSum(2, 6, 3, 4, 7)
findSum(2, 6, 6, 4, 5)
findSum(2, 6, 7, 6, 7)

セグメント木をつかった区間和計算例

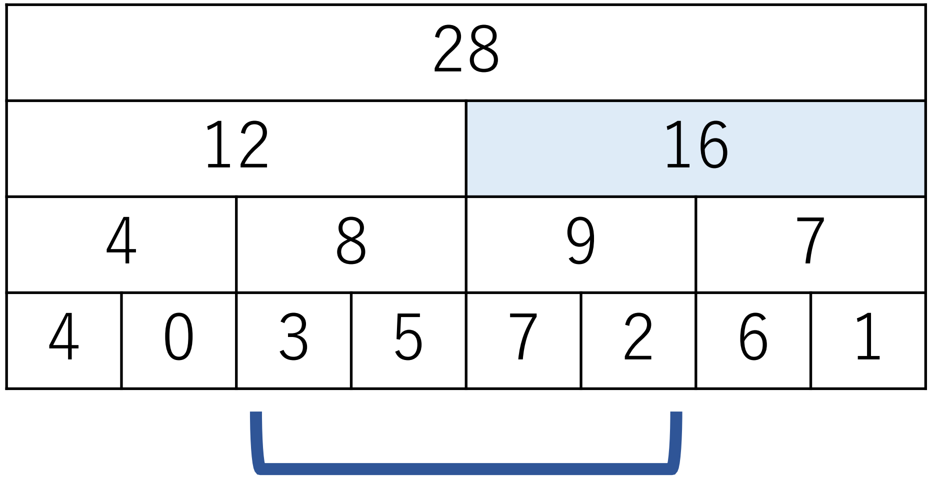
self.array：[0, 28, 12, 16, 4, 8, 9, 7, 4, 0, 3, 5, 7, 2, 6, 1]



findSum(2, 6, 1, 0, -1)
findSum(2, 6, 2, 0, 3)
findSum(2, 6, 4, 0, 1)
findSum(2, 6, 5, 2, 3) # 全部含む
findSum(2, 6, 3, 4, 7)
findSum(2, 6, 6, 4, 5)
findSum(2, 6, 7, 6, 7)

セグメント木をつかった区間和計算例

self.array：[0, 28, 12, 16, 4, 8, 9, 7, 4, 0, 3, 5, 7, 2, 6, 1]



findSum(2, 6, 1, 0, -1)

findSum(2, 6, 2, 0, 3)

findSum(2, 6, 4, 0, 1)

findSum(2, 6, 5, 2, 3)

findSum(2, 6, 3, 4, 7) # 一部含む

findSum(2, 6, 6, 4, 5)

findSum(2, 6, 7, 6, 7)

セグメント木をつかった区間和計算例

self.array：[0, 28, 12, 16, 4, 8, 9, 7, 4, 0, 3, 5, 7, 2, 6, 1]

<table>
<tr><td colspan="8">28</td></tr>
<tr><td colspan="4">12</td><td colspan="4">16</td></tr>
<tr><td colspan="2">4</td><td colspan="2">8</td><td colspan="2">9</td><td colspan="2">7</td></tr>
<tr><td>4</td><td>0</td><td>3</td><td>5</td><td>7</td><td>2</td><td>6</td><td>1</td></tr>
</table>

findSum(2, 6, 1, 0, -1)

findSum(2, 6, 2, 0, 3)

findSum(2, 6, 4, 0, 1)

findSum(2, 6, 5, 2, 3)

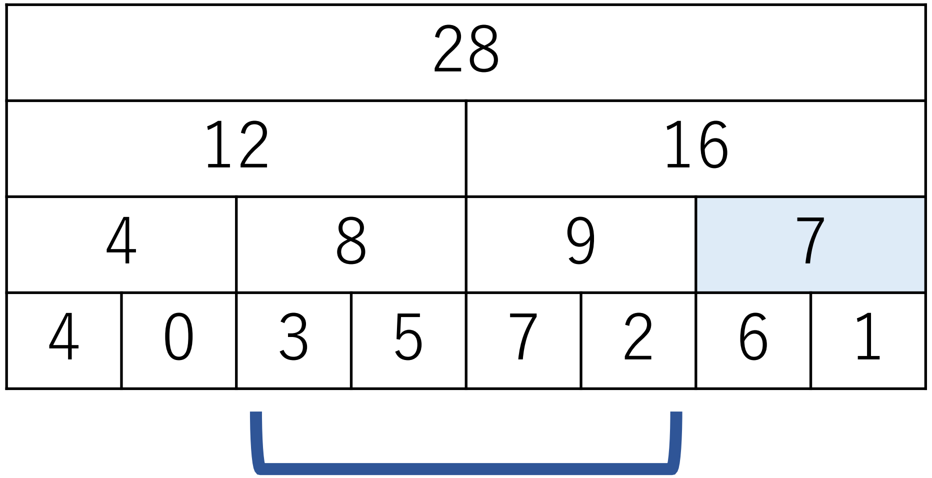
findSum(2, 6, 3, 4, 7)

findSum(2, 6, 6, 4, 5) # 全部含む

findSum(2, 6, 7, 6, 7)

セグメント木をつかった区間和計算例

self.array：[0, 28, 12, 16, 4, 8, 9, 7, 4, 0, 3, 5, 7, 2, 6, 1]



findSum(2, 6, 1, 0, -1)
findSum(2, 6, 2, 0, 3)
findSum(2, 6, 4, 0, 1)
findSum(2, 6, 5, 2, 3)
findSum(2, 6, 3, 4, 7)
findSum(2, 6, 6, 4, 5)
findSum(2, 6, 7, 6, 7) # 全く含まない

セグメント木をつかった区間和計算例

```
seq = [4, 6, 3, 5, 7, 2, 0, 1]
segtree = CSumSegTree(seq)
segtree.update(3, 50)        # 5 -> 50に変更
print(segtree.findSum(2, 6))
---
```

実行結果

62

self.array：

[0, 73, 57, 16, 4, 53, 9, 7, 4, 0, 3, 50, 7, 2, 6, 1]

<table>
<tr><td colspan="8">73</td></tr>
<tr><td colspan="4">57</td><td colspan="4">16</td></tr>
<tr><td colspan="2">4</td><td colspan="2">53</td><td colspan="2">9</td><td colspan="2">7</td></tr>
<tr><td>4</td><td>0</td><td>3</td><td>50</td><td>7</td><td>2</td><td>6</td><td>1</td></tr>
</table>

セグメント木の計算量：セグメント木の構築

セグメント木の一番下の目を埋める：n回の操作
セグメント木の下から2段目を埋める： $n/2$回の操作
セグメント木の下から3段目を埋める： $n/4$回の操作

$$\cdots$$

これが根ノードの計算が終わるまで続く.

セグメント木の計算量

すべての操作回数を足し合わせると,

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots < 2n$$

であるから，全体としての計算量は$O(n)$.

セグメント木の計算量：問い合わせ

セグメント木を使って計算結果を得る.

→最も深くまで辿っても高々$O(\log n)$.

セグメント木の計算量：更新

セグメント木の葉の要素の更新.

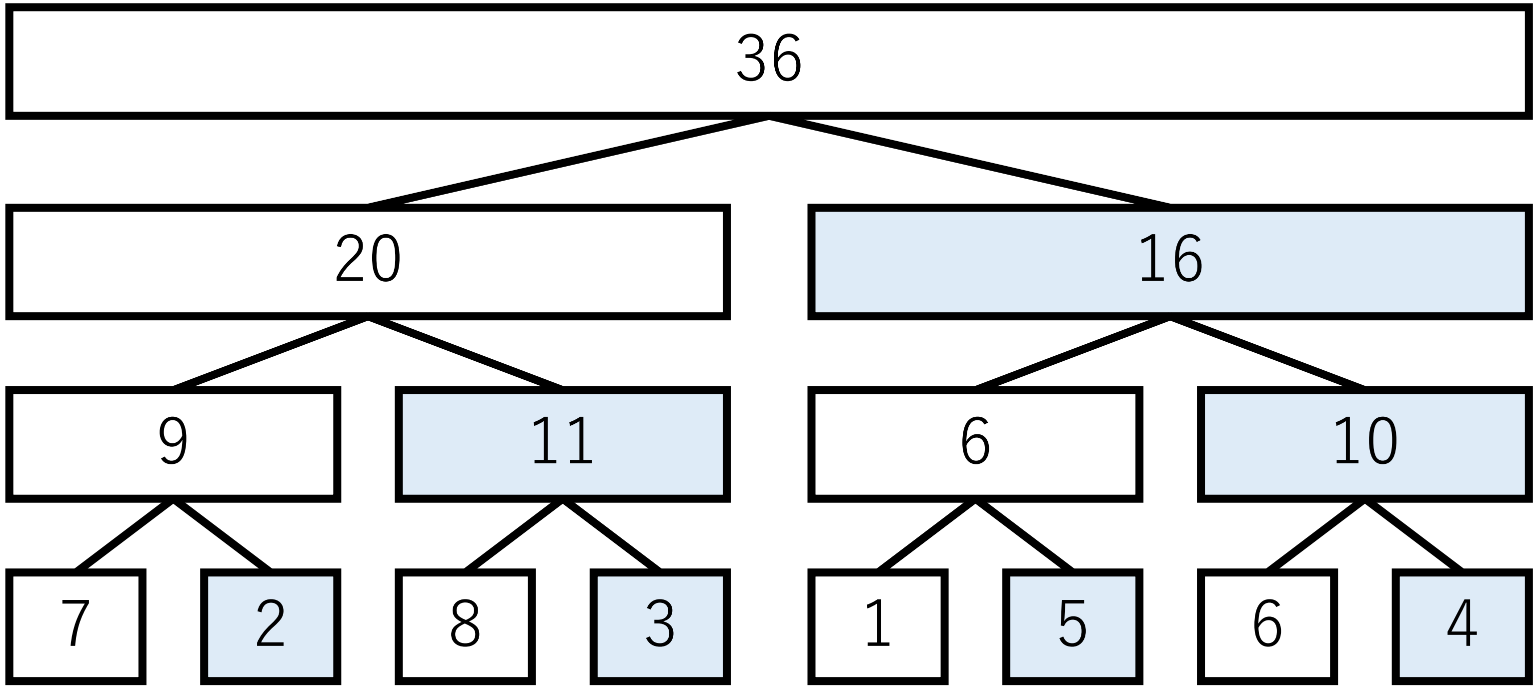
→葉ノードから順に親ノードを辿る．根（ルート）ノードのところまで辿っても高々$O(\log n)$.

単純な累積和による実装では，与えられる配列の要素が変更されると，再度累積和を計算し直す必要があり，平均的には$O(n)$かかってしまう.

要素の更新があるような場合の区間和のときに有利.

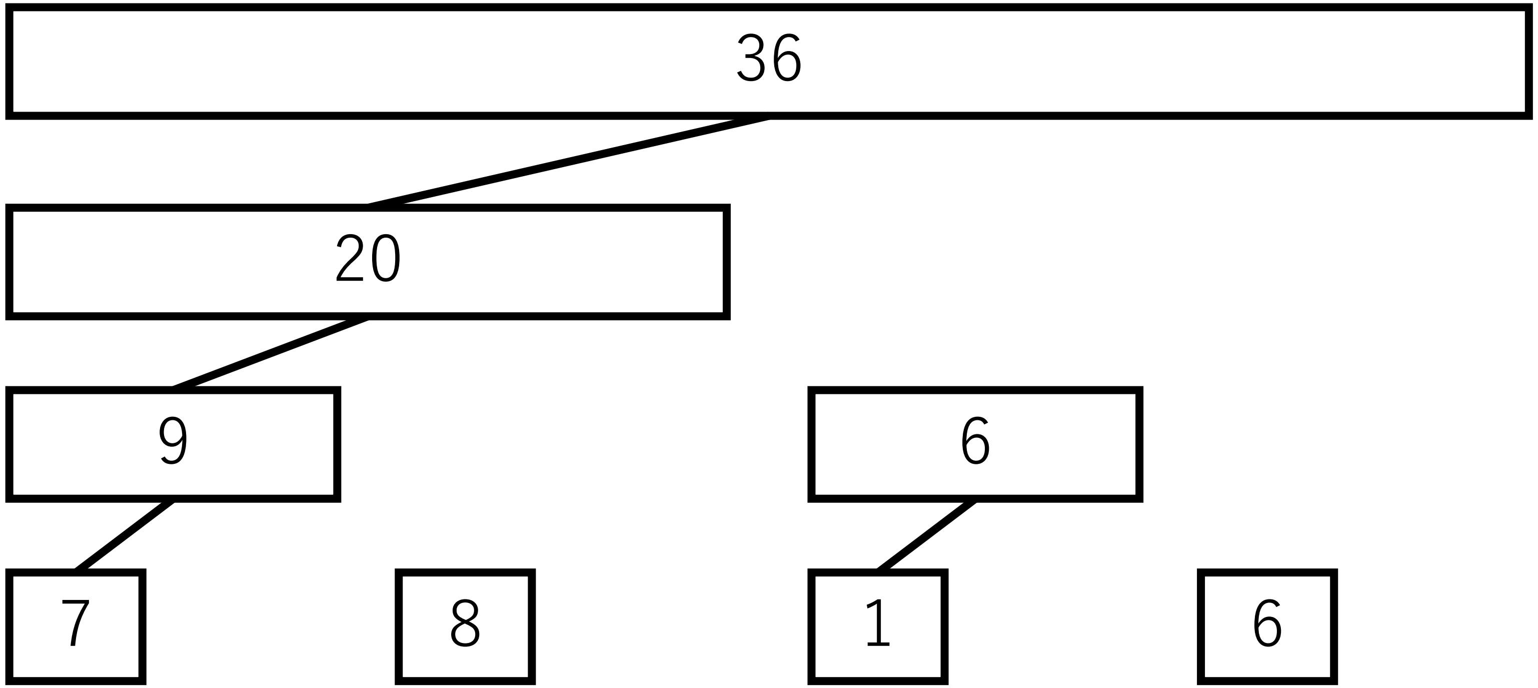
セグメント木

区間和のセグメント木の場合，青色部分の情報はなくても上位のノードの情報を使うことで計算できる．



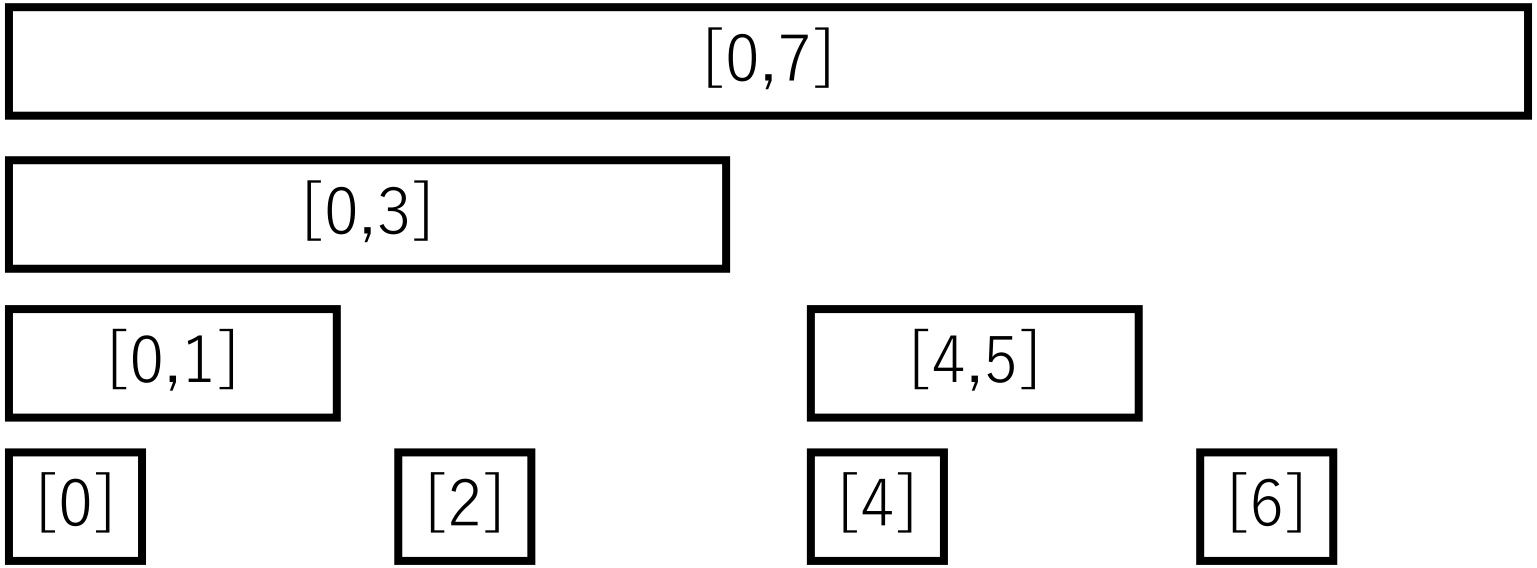
セグメント木

この復元可能部分を削ると，記憶領域をおよそ半分に減らすことができる．



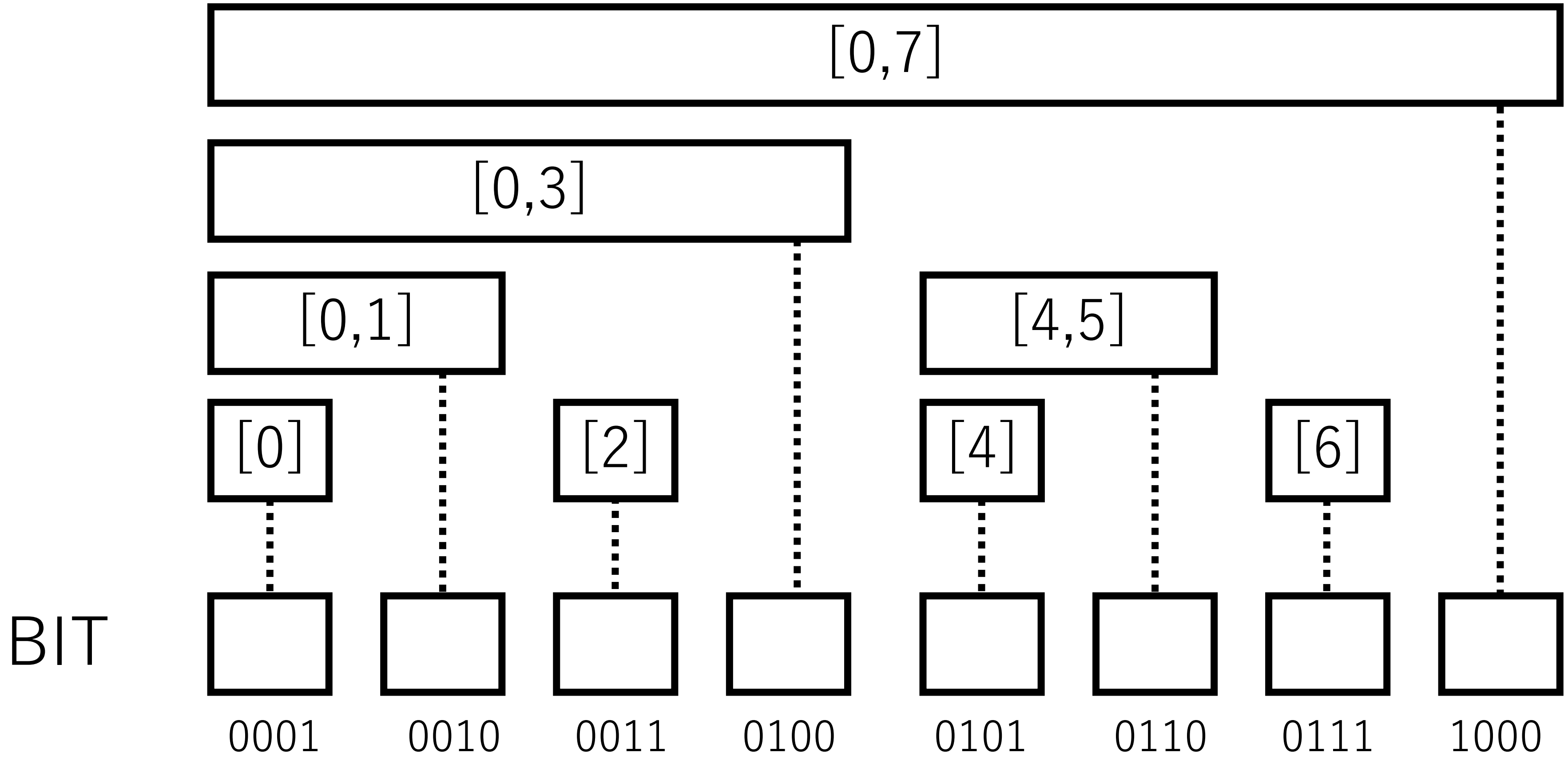
Binary Indexed Tree（BIT）

区間和の計算をビット演算で実行できるように設計した配列.
先頭からi番目までの部分和の計算に使える.



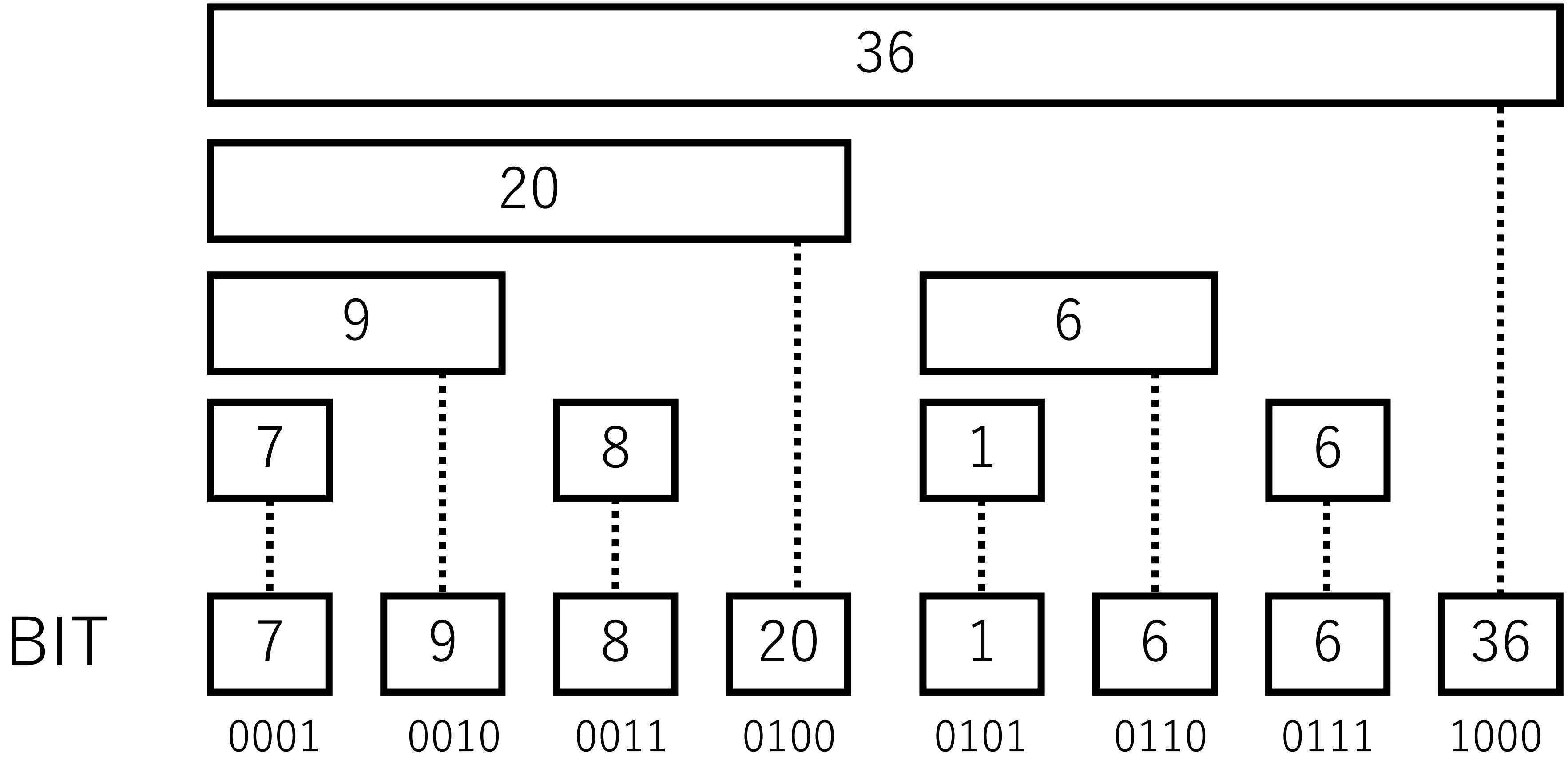
Binary Indexed Tree（BIT）

区間和の計算をビット演算で実行できるように設計した配列.
先頭からi番目までの部分和の計算に使える.



Binary Indexed Tree（BIT）

区間和の計算をビット演算で実行できるように設計した配列.
先頭からi番目までの部分和の計算に使える.



Binary Indexed Tree（BIT）

区間和の計算をビット演算で実行できるように設計した配列．
先頭からi番目までの部分和の計算に使える．

BIT	7	9	8	20	1	6	6	36
	0001	0010	0011	0100	0101	0110	0111	1000

Binary Indexed Tree（BIT）

a[0]からの区間和を求める
　→一番後ろの1のビットを減算しながら足していく.

BIT	7	9	8	20	1	6	6	36
	0001	0010	0011	0100	0101	0110	0111	1000

Binary Indexed Tree（BIT）

例）a[0]からa[5]の区間和.
a[5]は6番目の要素なので，6の2値表現0110からスタート.

BIT	7	9	8	20	1	6	6	36
	0001	0010	0011	0100	0101	0110	0111	1000

Binary Indexed Tree（BIT）

0110の要素を足して，一番後ろの1のビットを0にする．

→0100に移る．

BIT	7	9	8	20	1	6	6	36
	0001	0010	0011	0100	0101	0110	0111	1000

Binary Indexed Tree（BIT）

0100の要素を足して，一番後ろの1のビットを0にする．

→0000になるので終了．答えは26.

BIT	7	9	8	20	1	6	6	36
	0001	0010	0011	0100	0101	0110	0111	1000

Binary Indexed Tree（BIT）

一番後ろの1のビットはどうやって求めれば良い？

→自分自身の2の補数（ビットを反転して1足す）とのANDを取る.

例）6（0110）の場合，-6（1010）とのANDで，0010（2）.

→6番目の要素を見た後は，6-2=4番目の要素に移動，となる.

BITの計算量

時間計算量はセグメント木に同じ．ただし定数倍程度軽い．

ビット演算により必要な値を順に見ていくことができる．

空間使用量はセグメント木のおよそ半分．

ただ，セグメント木ではだめだけど，BITなら行ける，というケースは実際上は稀と思われる．

まとめ

データ構造

スタック，キュー，線形リスト，ツリー，ヒープ

セグメント木，BIT

コードチャレンジ：基本課題#3-a [1点]

スライドで説明されているキューのクラスに基づき，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．

enqueue，dequeueをどう実装すべきか，考えてみてください．

deque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#3-b [2点]

二分ヒープを<u>自分で</u>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.

heapq等を使用することは認めません.

コードチャレンジ：Extra課題#3 [3点]

データ構造に関係する課題．（セグメント木，BITは使いません）