Algorithms (2024 Summer)
#3：データ構造

矢谷 浩司
東京大学工学部電子情報工学科

なぜデータ構造を考えるか？

単に変数やリストで保存しておいて，必要な度に適当に引っ張り出して，計算・処理すればよいのではは？

拿出来

なぜデータ構造を考えるか？

単に変数やリストで保存しておいて，必要な度に適当に引っ張り出して，計算・処理すればよいのでは？

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.

ないほう

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.

包含

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

（少し発展的な内容）

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.

堆起来

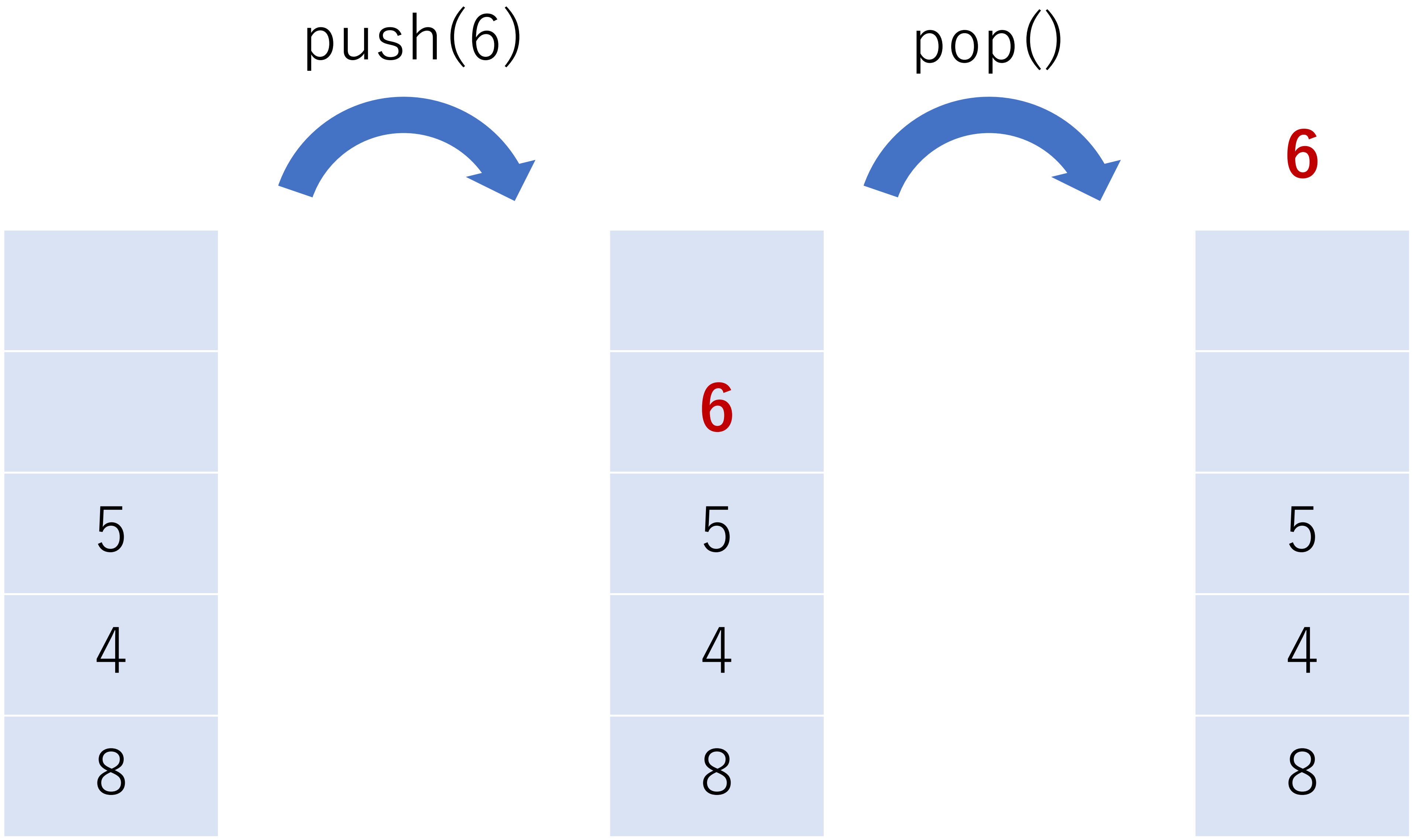
積む「つむ」

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  用list实现
        self.top = 0  stack顶位置
```

スタック

```
class Stack:
    def push(self, a: int):
        if self.top < len(self.stack):  确认不会overflow
            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:            避免underflow
            self.top -= 1           self.top是待存入/取出数据的位置
            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．

使...靠拢

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:
    入队
    def enqueue(self, a: int):
        if self.tail < len(self.queue):
            self.queue[self.tail] = a    self.tail是待插入元素的位置
            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.')
```

出队

self.head是队头元素的位置

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.

挪动

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.

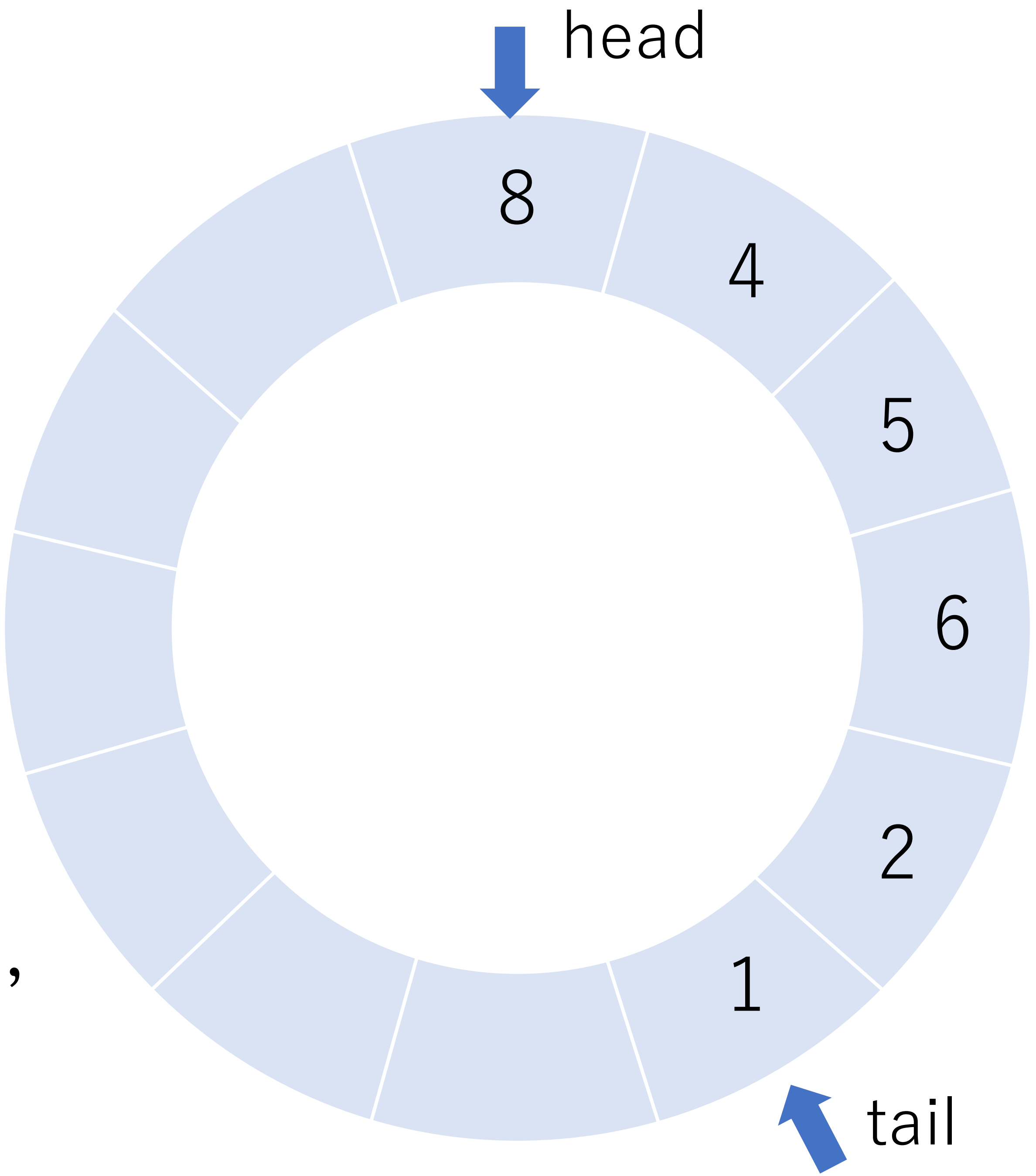
=> 一定会造成空间浪费

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.

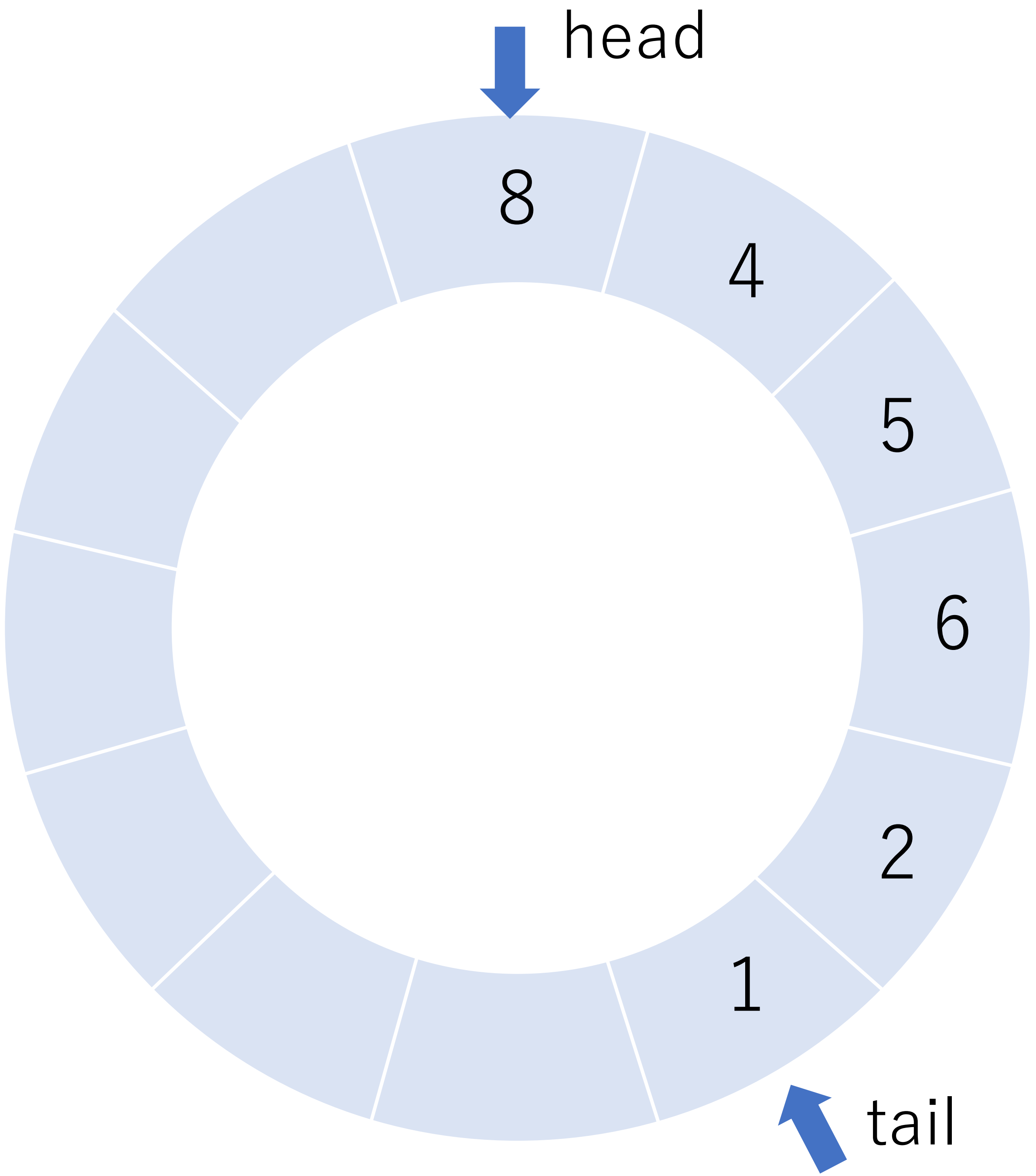
最新のX個の情報を覚えておく，みたいな使い方もできる.



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．



リングバッファの実装

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.

例えば，バッファサイズを10に設定したとする.

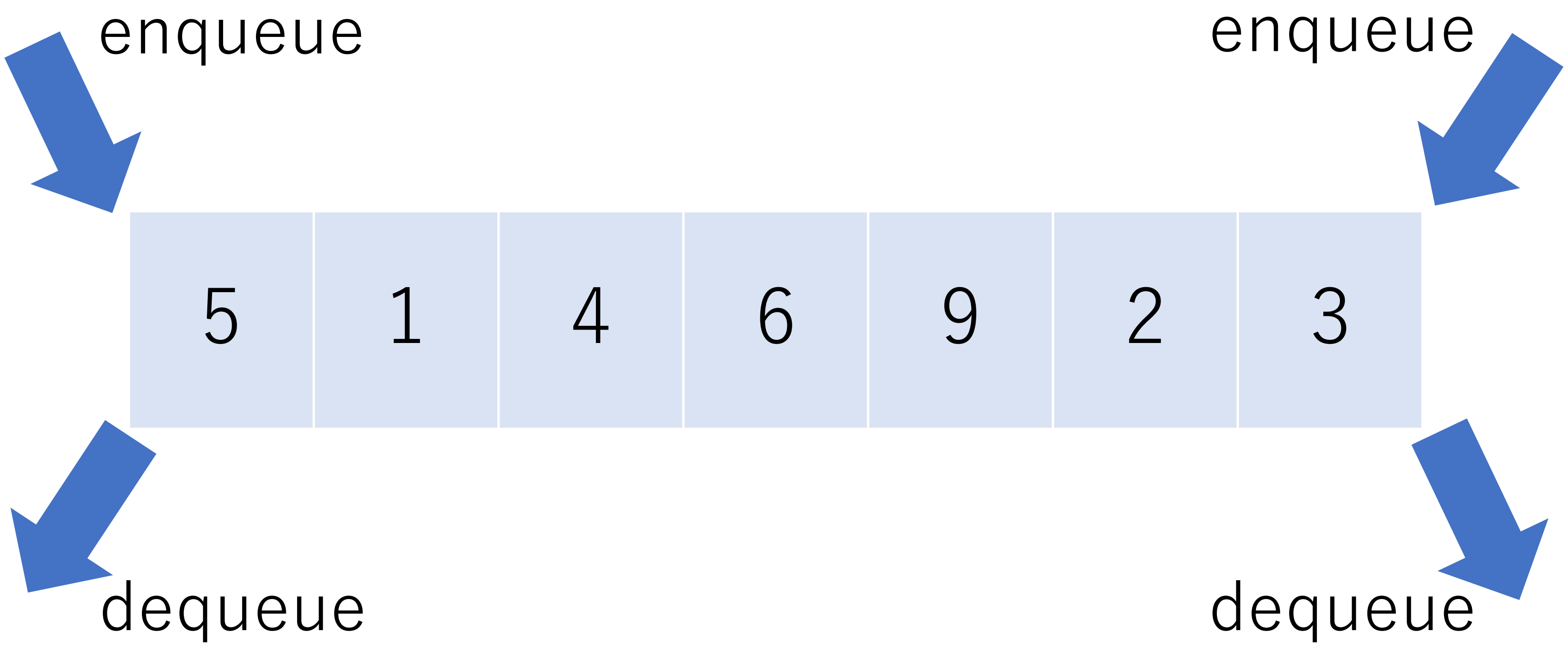
先頭（head）のindexが9である時，さらに先頭を進めることになったら，indexを0にする.

末尾の方も同様に処理すれば良い.

ただし，tailはheadを超えないようにする.

デック（deque）

キューの先頭と末尾のどちらからでもenqueue，dequeueできる．



デック（deque）

pythonではcollectionsモジュールにおいてdequeという型が提供されており，便利に使うことができる．

```
from collections import deque
d = deque([1])          # [1]
d.append(2)             # 右側に追加，[1, 2]
d.appendleft(3)         # 左側に追加，[3, 1, 2]
a = d.pop()             # 右側から取り出す，a = 2
b = d.popleft()         # 左側から取り出す，b = 3
```

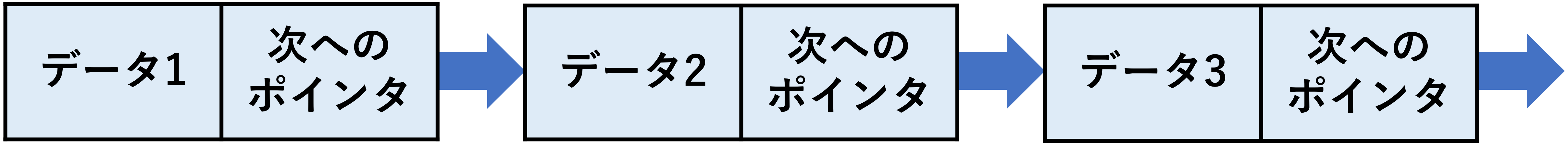
連結リスト（線形リスト）

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．

念珠
じゅず0

末尾のリストの次へのポインタはNULLになる．

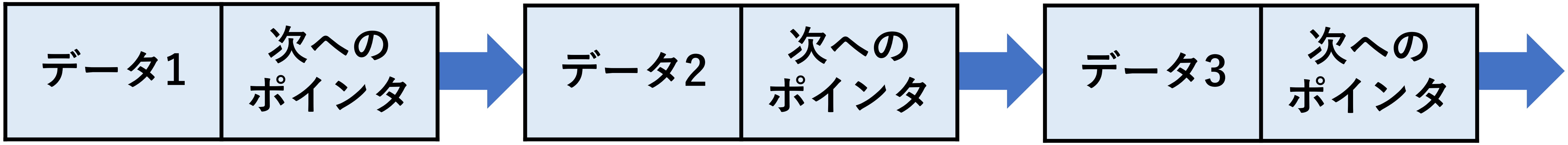
双方向や循環になっているものある．



連結リスト（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．



連結リストの実装例

以下では簡易的な連結リストの実装を行う.

片方向のリスト.

新しい要素は常に末尾に追加.

要素を見つけたら同時にそれを取り出す.

双方向や循環するリスト，ソート順を確保しながら要素を挿入する方法なども似たような形で実装できる.

連結リストの実装例

```
# 連結リストのセルを表すクラス
class LinkedListCell:
    def __init__(self, value):
        self.value = value
        self.next = None
```

而不是什么”NULL”

連結リストの実装例

```
# 連結リストを表すクラス
class LinkedList:
    def __init__(self):
                            sentinal node
        # 先頭にダミーデータをいれておく
        self.head = LinkedListCell(None)
```

連結リストの実装例

```
class LinkedList:
    def append(self, value):     # 要素の追加
        cur_cell = self.head
        while cur_cell.next != None:      # 最後尾まで移動
            cur_cell = cur_cell.next

        new_cell = LinkedListCell(value)
        cur_cell.next = new_cell
        print('Added')
```

連結リストの実装例

注意这里是dummy节点

```
class LinkedList:
        def pop(self, value):
                prev_cell = None; cur_cell = self.head
                while cur_cell != None:
                        if cur_cell.value == value:
                                prev_cell.next = cur_cell.next; cur_cell = None
                                print('Found and deleted')
                                return value
                        prev_cell = cur_cell; cur_cell = cur_cell.next

                print('Not found') # 要素が見つからなかった
```

連結リストの実装例

```
class LinkedList:
        def show(self):
                output_str = ''
                cur_cell = self.head.next
                while cur_cell != None:
                        output_str += str(cur_cell.value) + ' '
                        cur_cell = cur_cell.next

                print(output_str)
```

self.head是sentinal node

連結リストの実行例

```
LL = LinkedList()
LL.append(1)
LL.append(2)
LL.append(3)
LL.show()
LL.pop(2)
LL.show()
LL.pop(4)
```

```
Added
Added
Added
1 2 3
Found and deleted
1 3
Not found
```

連結リストの計算量例（配列と比較）

現在格納されている要素の数をn，とりうる最大の要素数Mをとすれば，

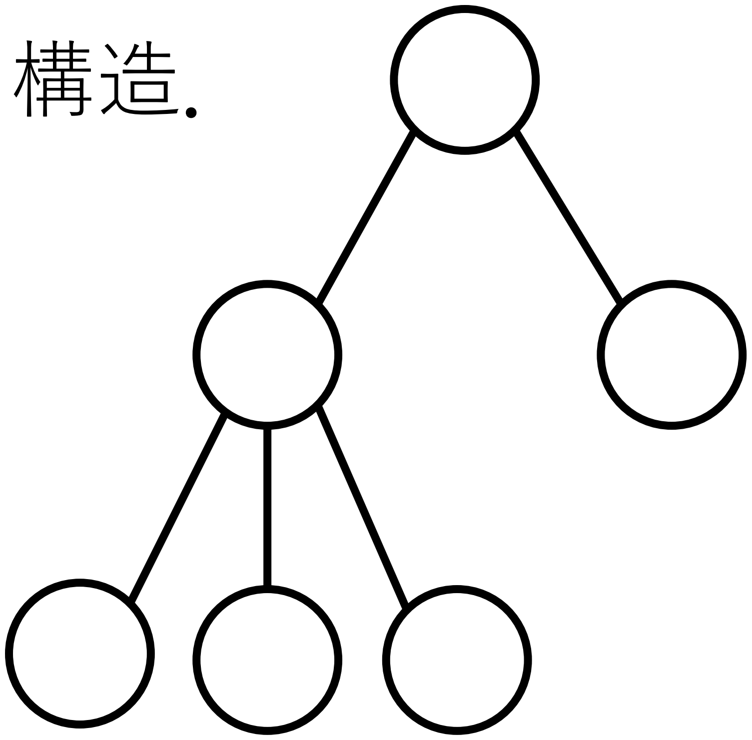
	連結リスト（上の実装例の場合）	配列（動的に長さを変えない）
要素の追加	$O(n)$	$O(1)$
i番目の要素の取り出し	$O(n)$	$O(1)$
ある値の要素の探索	$O(n)$	$O(n)$
空間計算量	$O(n)$ 変動する	$O(M)$ 変動しない

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.
翻过来

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.

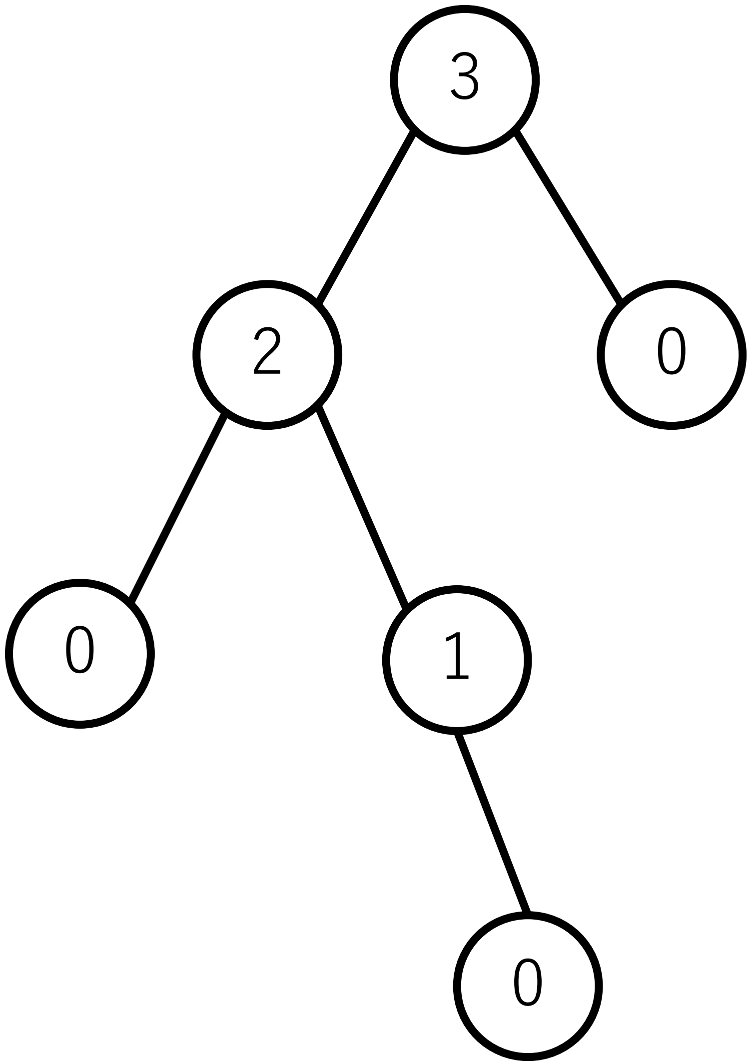


木構造（ツリー）

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.

各ノードの高さ

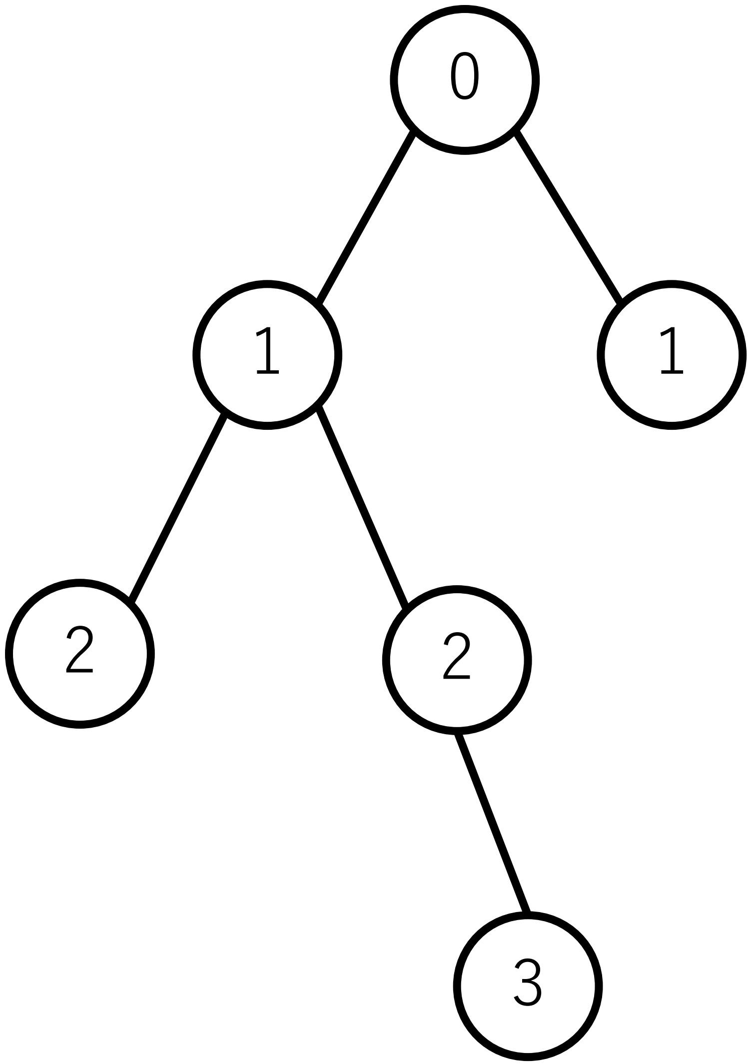


木構造（ツリー）

深さ

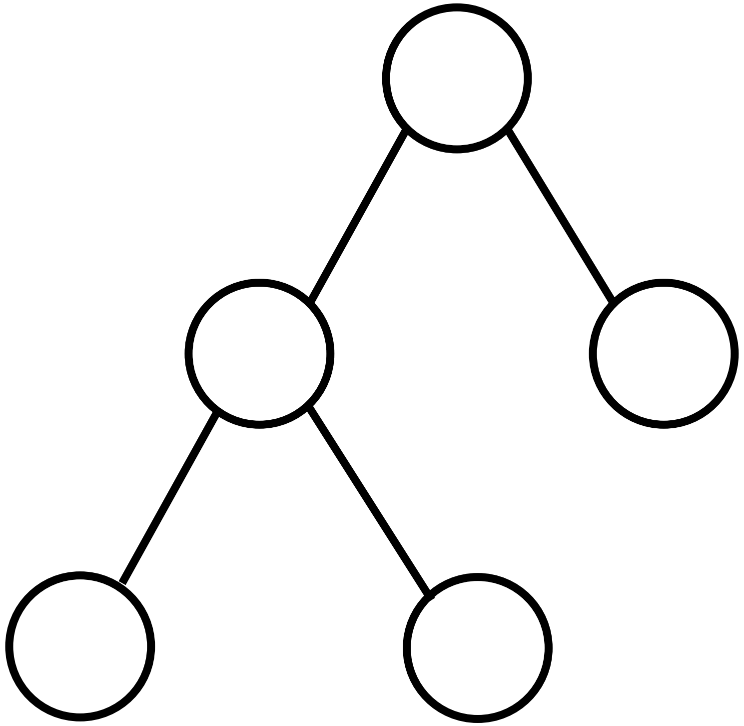
根ノードから，あるノードに至るまでに辿る必要があるエッジの数.
根ノード自体の深さは0.

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.

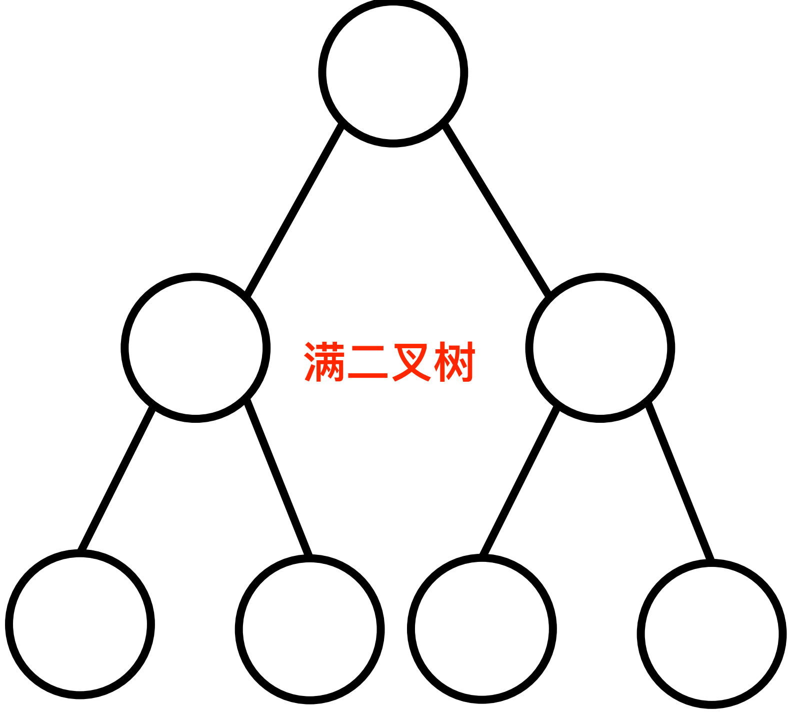


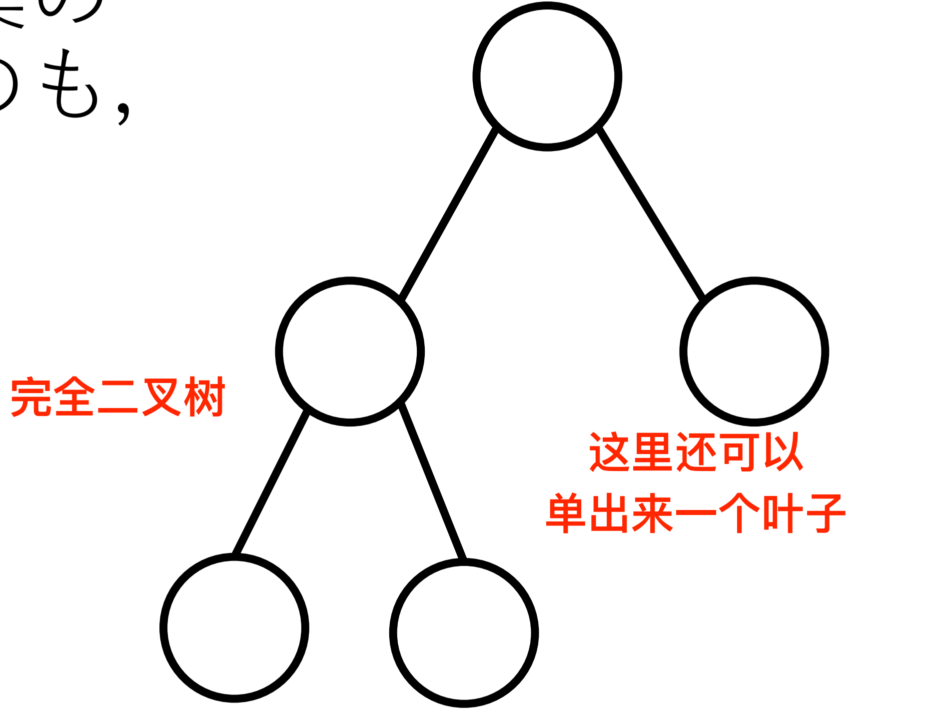
完全二分木

全ての葉が同じ深さ＆葉ノード以外の
全てのノードの子ノードの数が2.

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で
違いがわかるのだけど. . .





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:  只是Node的定义
    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. 也是完全二叉树

根ノードは常に最大値（最小値）になる.

max(min) heap

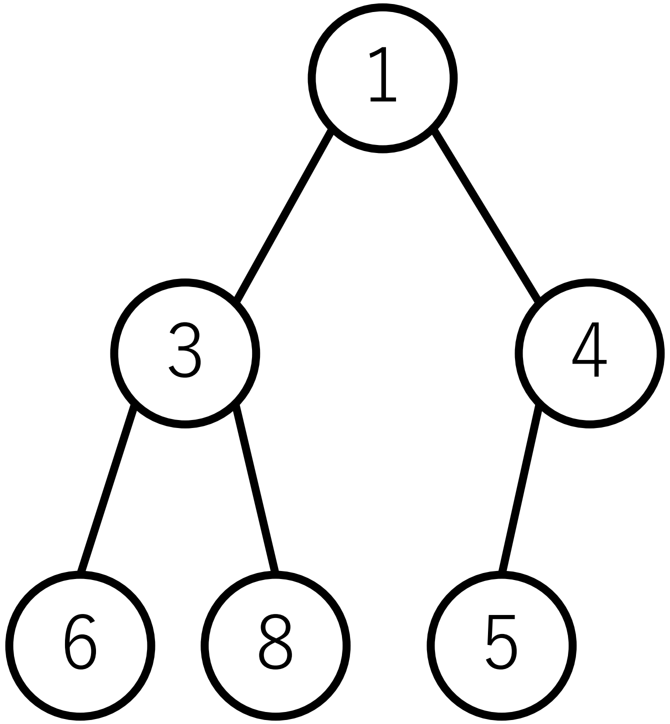
たくさんのデータがあり，かつ追加，削除が頻繁に行われる場合において，最大値や最小値に効率的に管理できる.

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）.

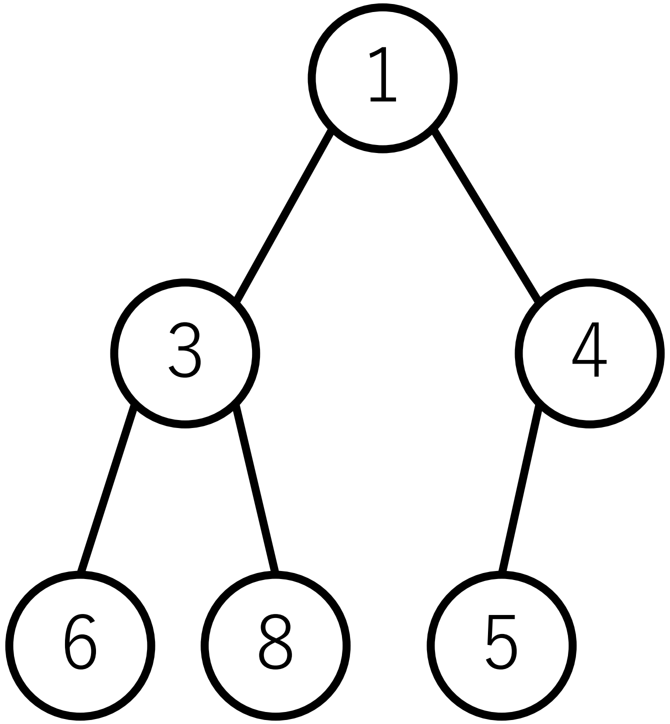
ツリーの形がcomplete binary tree.（たまたまfull binary treeになっていることもある）



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.

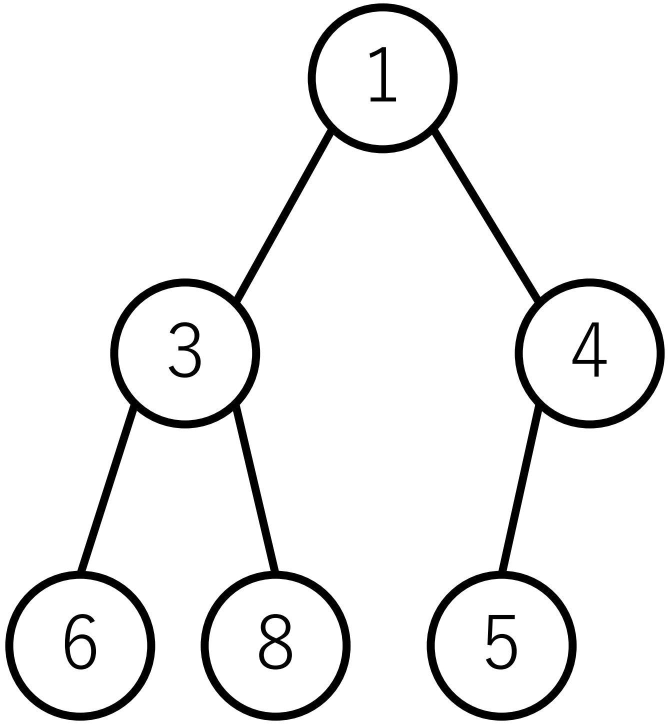
以降のスライドでも,「ヒープ」
という表現は二分ヒープを意味
するものとしています.



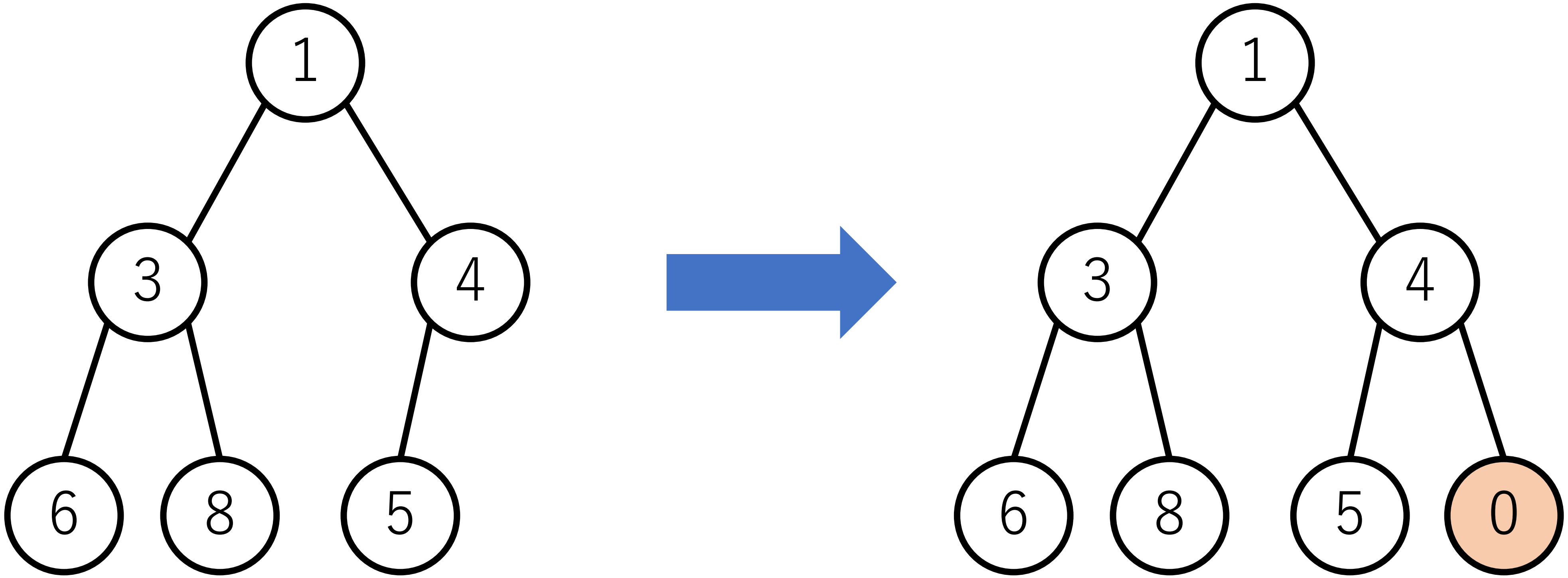
ヒープの操作：追加

#1 空いている一番左に葉として追加.

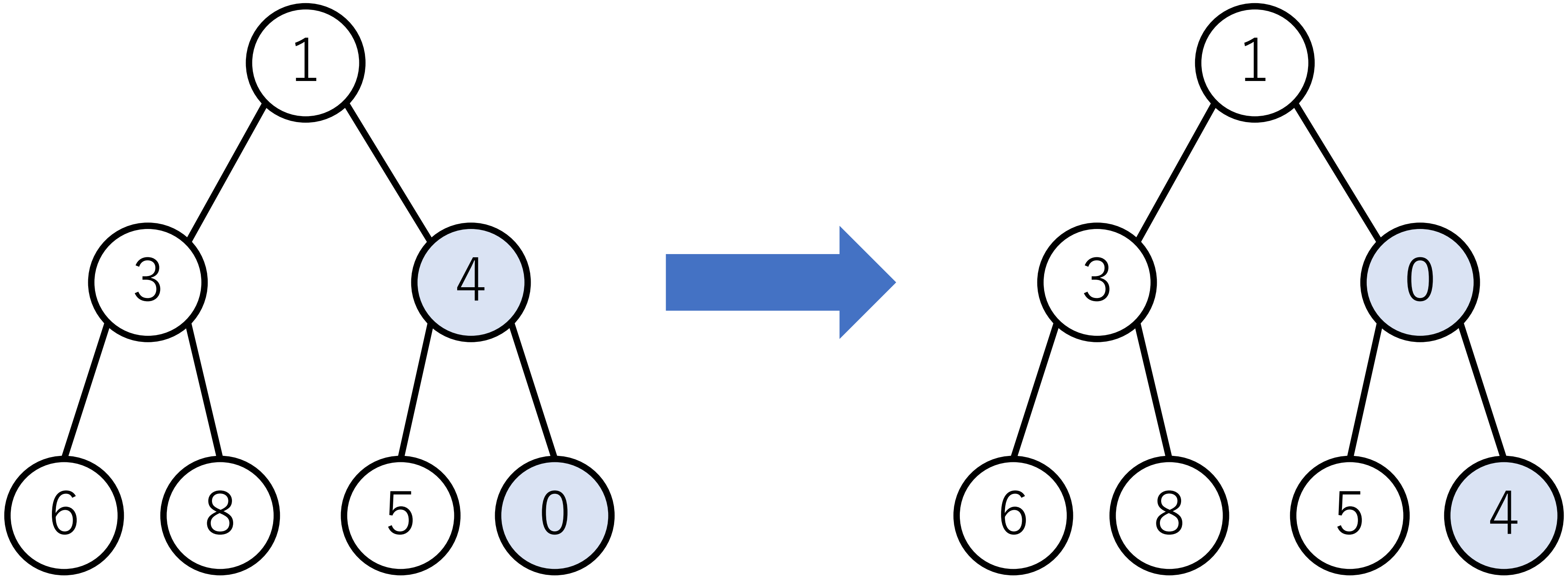
#2 親と比較. 制約条件を満たすように順次入れ替える.



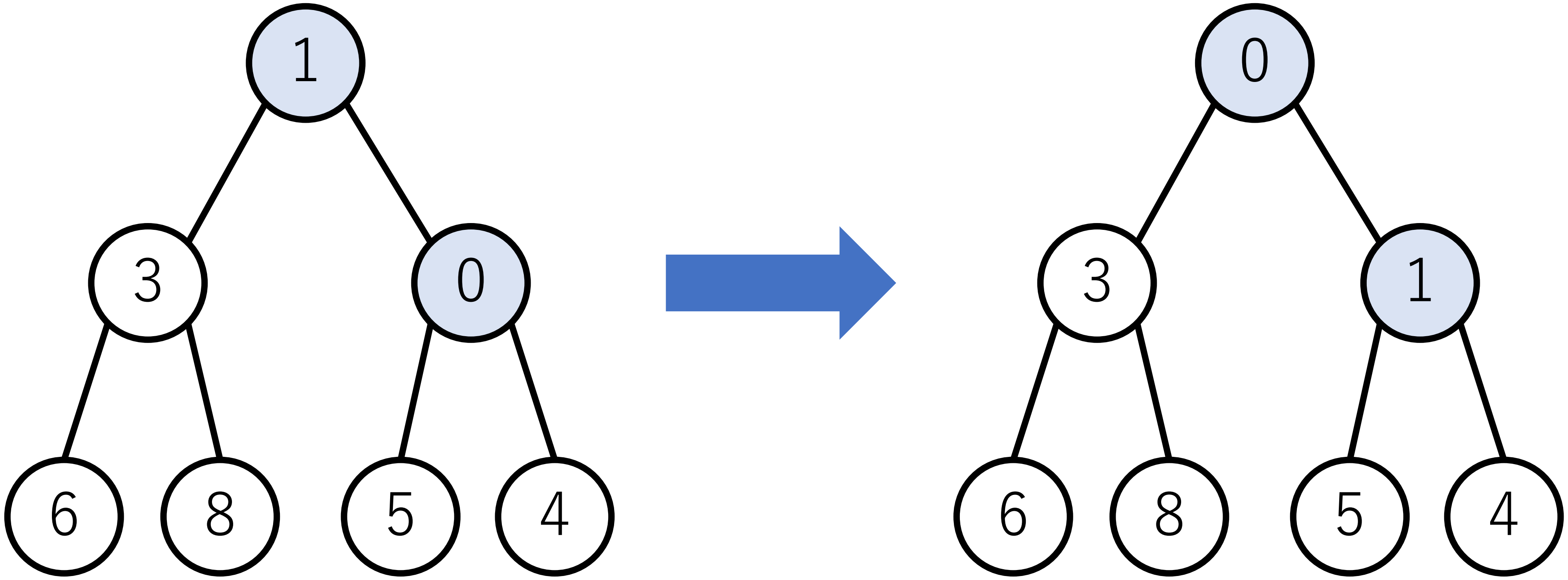
ヒープの操作：0を追加



ヒープの操作：0を追加



ヒープの操作：0を追加



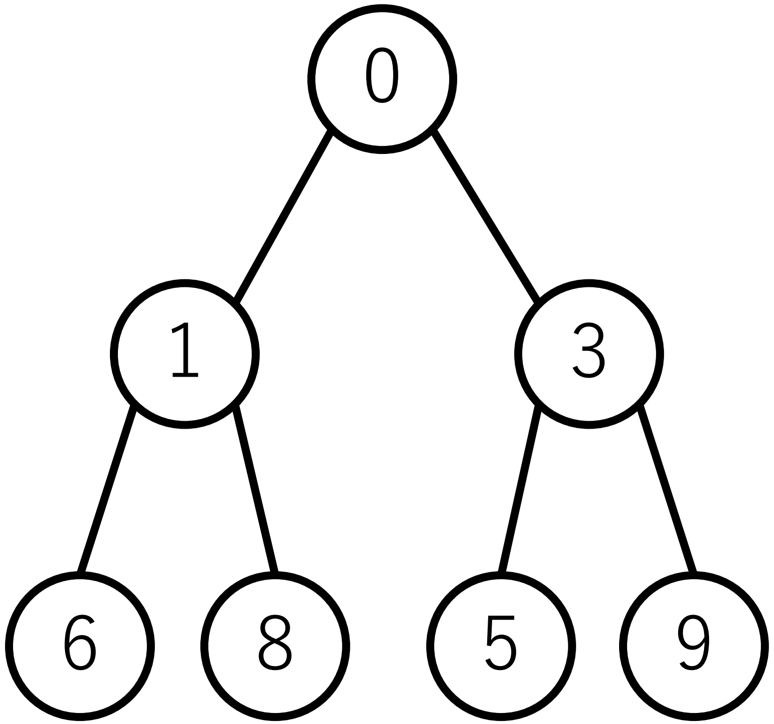
ヒープの操作：削除（値の取り出し）

#1 rootを取り除く.

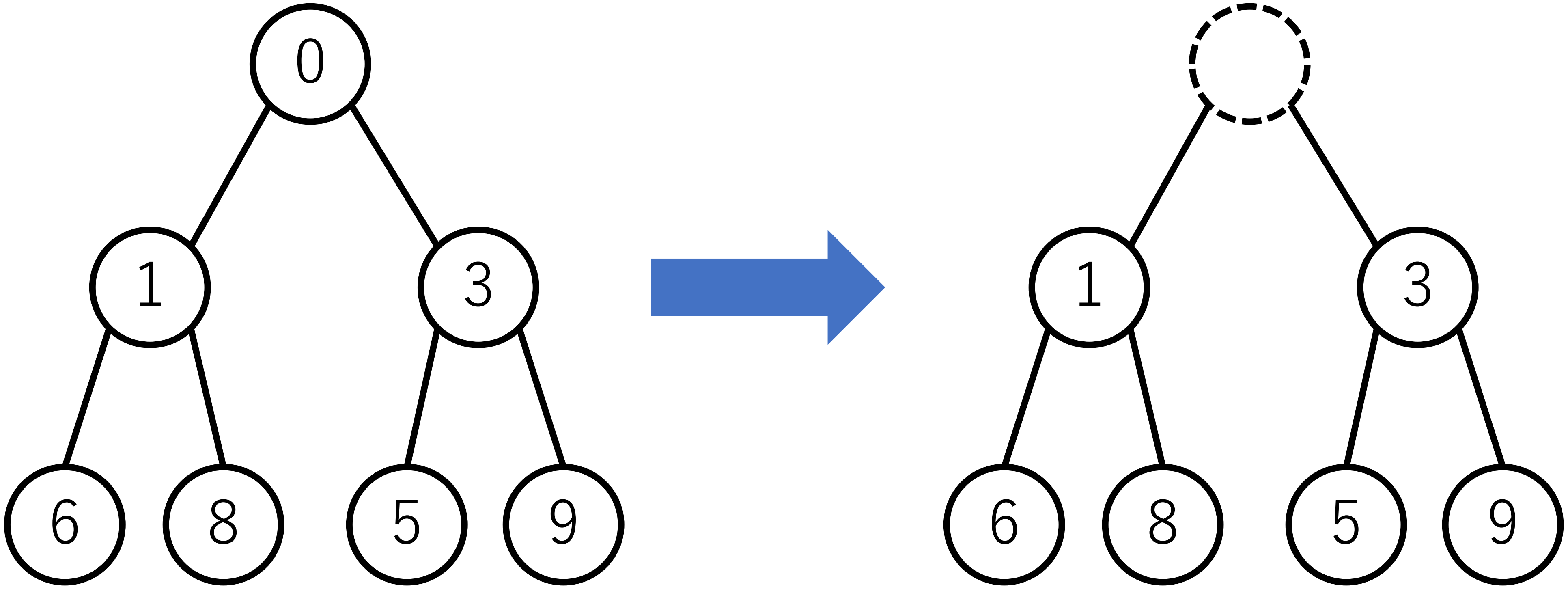
#2 一番右端にいる葉をrootにする.

#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.

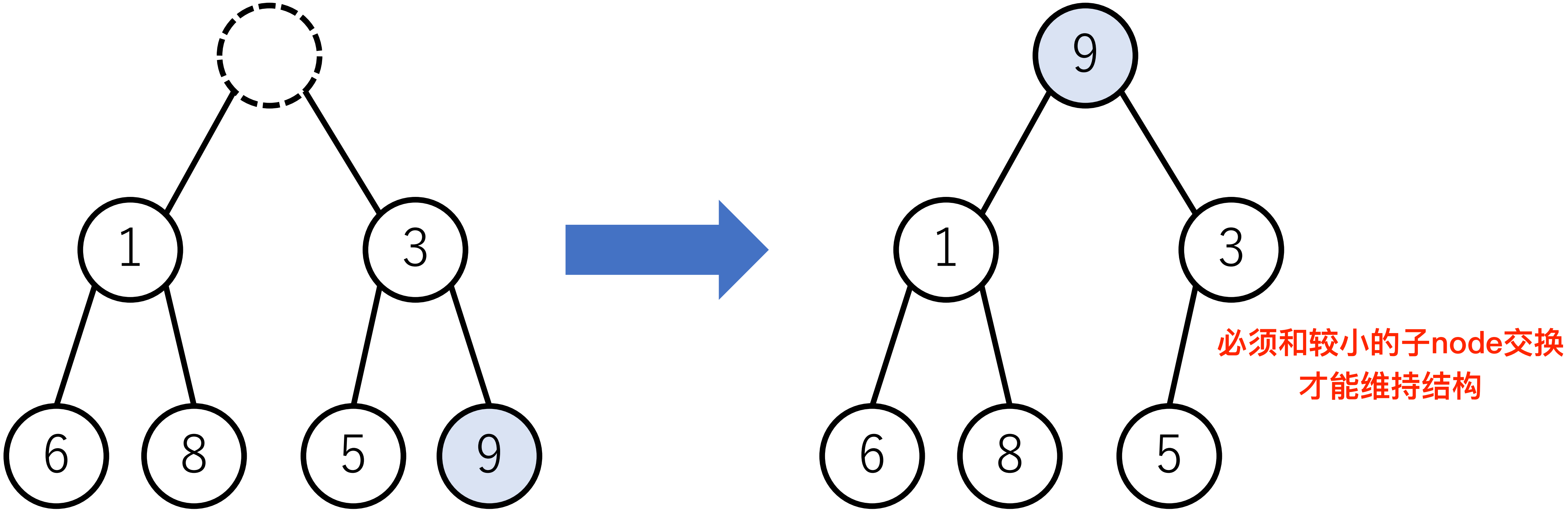
#4 制約条件を満たすまで入れ替えを続ける.



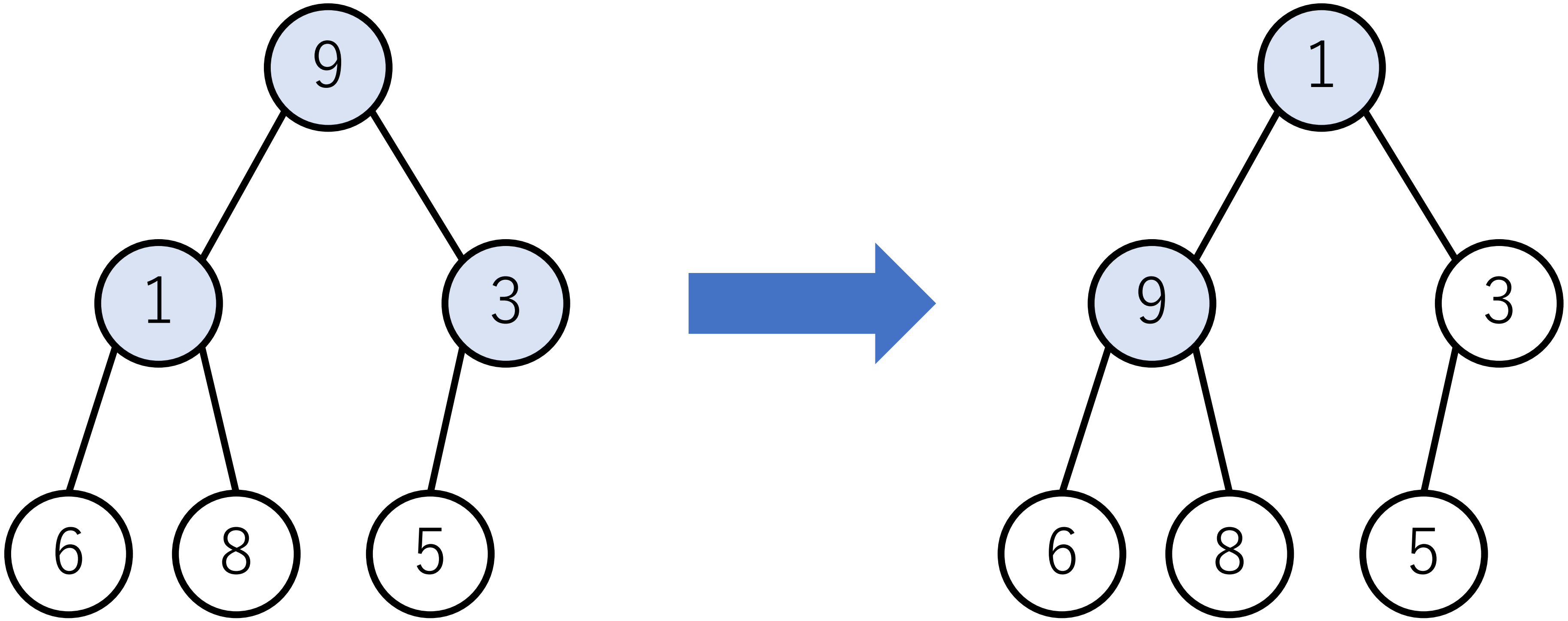
ヒープの操作：削除



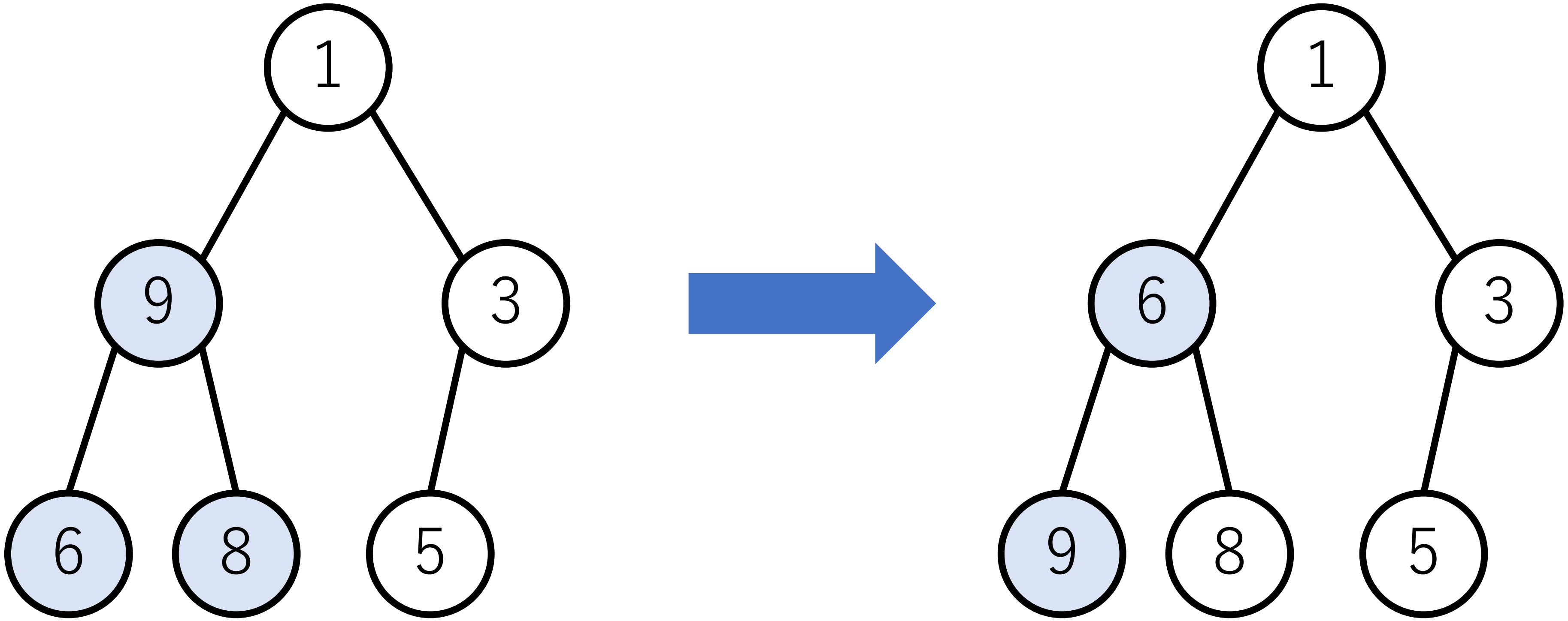
ヒープの操作：削除



ヒープの操作：削除



ヒープの操作：削除

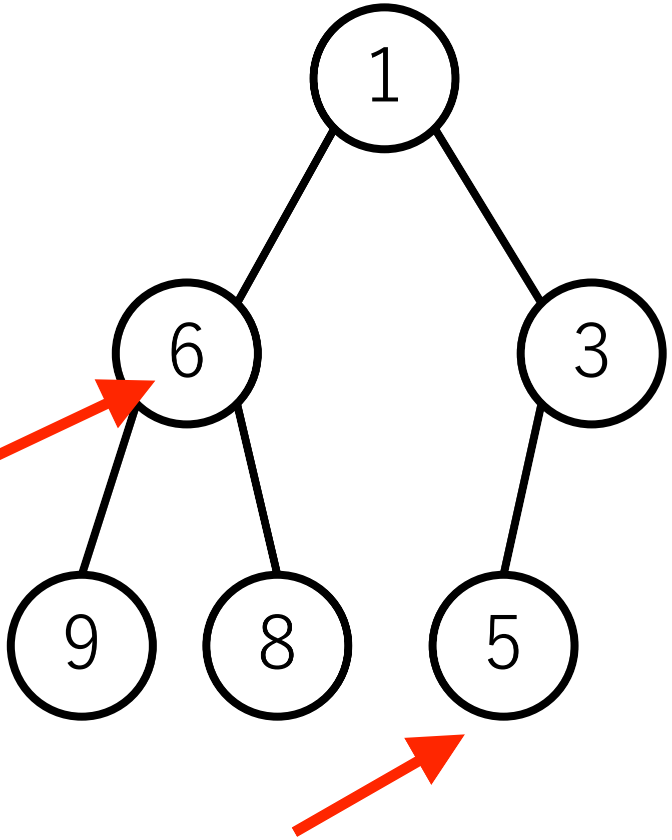


ヒープ

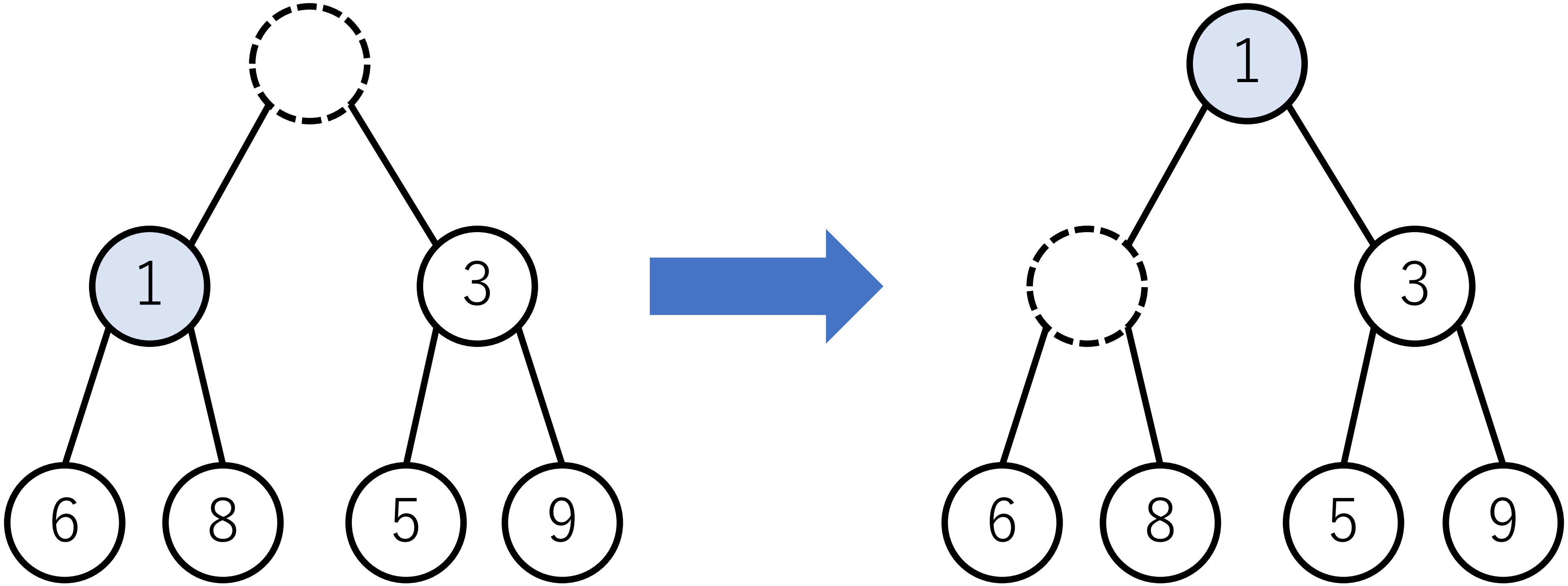
上下関係と木の形の制約が必ず守られていることが重要.

最小ヒープなら親ノードは子ノードよりも必ず小さい.

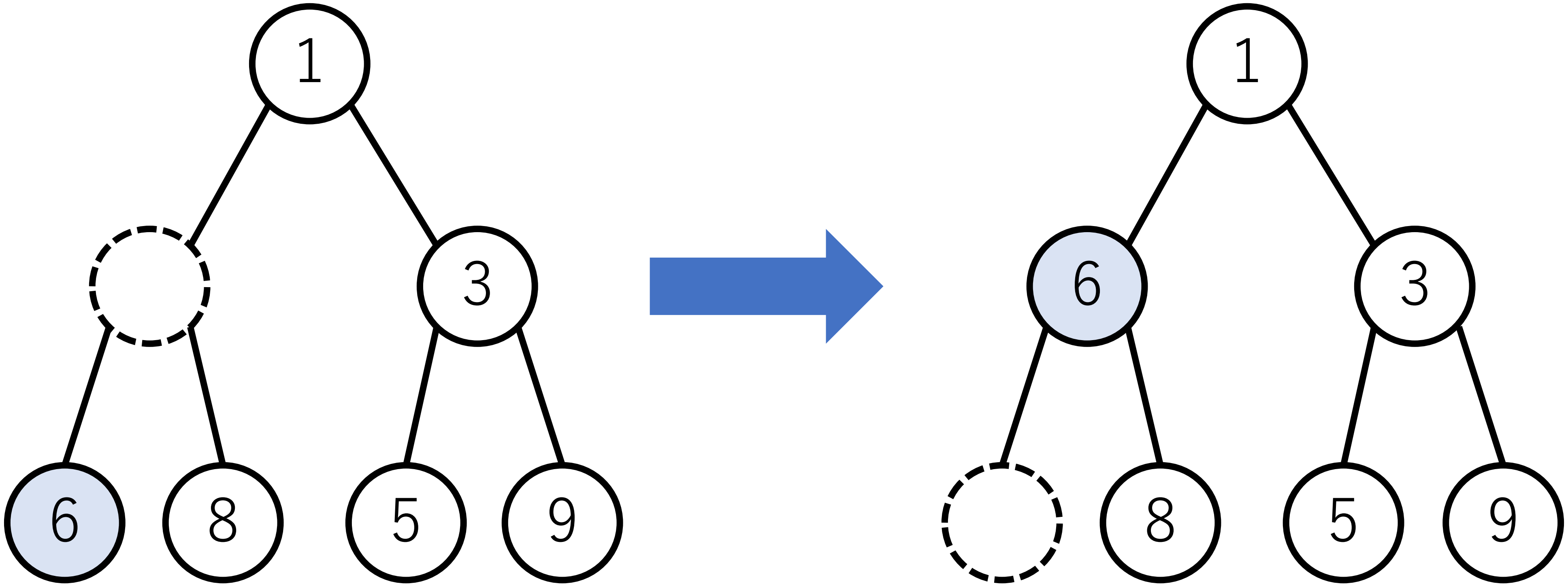
左右の部分木の関する制約は特にない.



もし，直接子ノードをあげてしまうと？

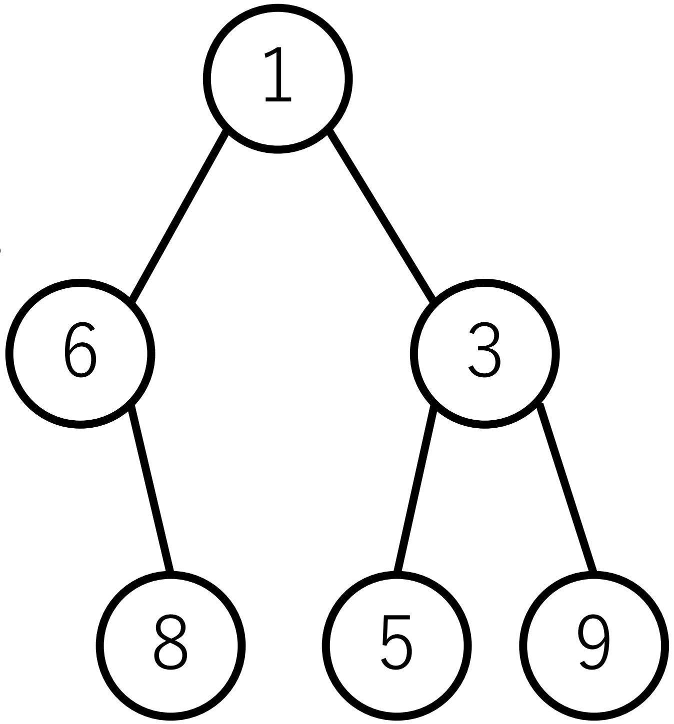


もし，直接子ノードをあげてしまうと？



もし，直接子ノードをあげてしまうと？

上下関係は守られているが，必ずしもcomplete binary treeにならず，次に値を挿入するときなど，実装上いろいろと面倒なことが起こる．．．



二分ヒープの配列での実装

ノードa[i]に対して,

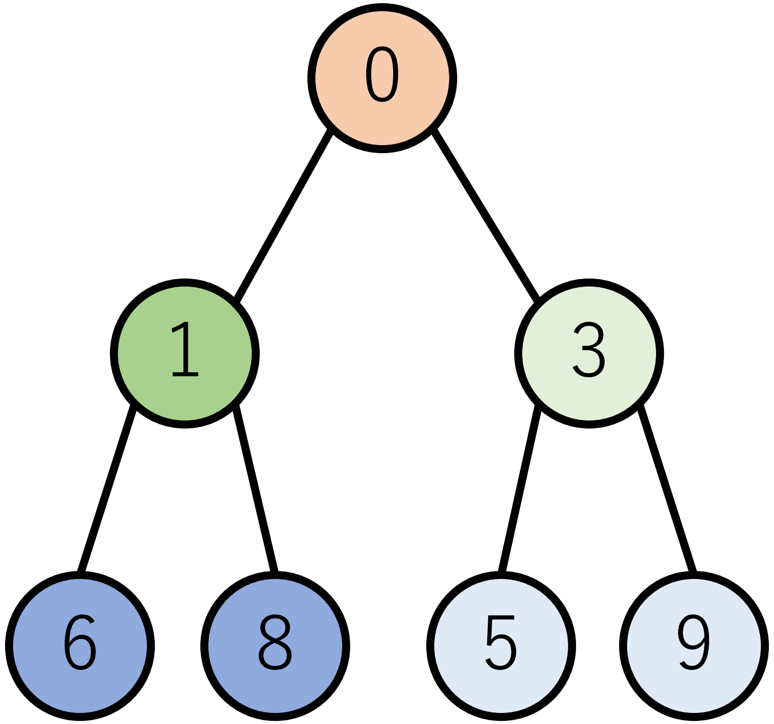
親: a[i//2]

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	1	3	6	8	5	9

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  因为舍弃array[0]不用
        # ヒープを構成する配列
        self.array = [self.inf]*self.size  但其实不会涉及到与self.inf的比较?
        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)
```

我觉得应该-1

self.last指向最后一个有数据的节点

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
```

此处remove指remove最小的结点

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(1)
                        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
```

此时只有根结点

ヒープの実装例（最小ヒープ）

```
class MyHeap:
    def check_after_add(self, i):
        if i < 2: return

                    self.array[i//2] > self.array[i]
        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？）]   check_after_add(self, i//2)
```

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，$O(\log n)$.

よって，追加にかかる計算量も，$O(\log n)$.

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．😌

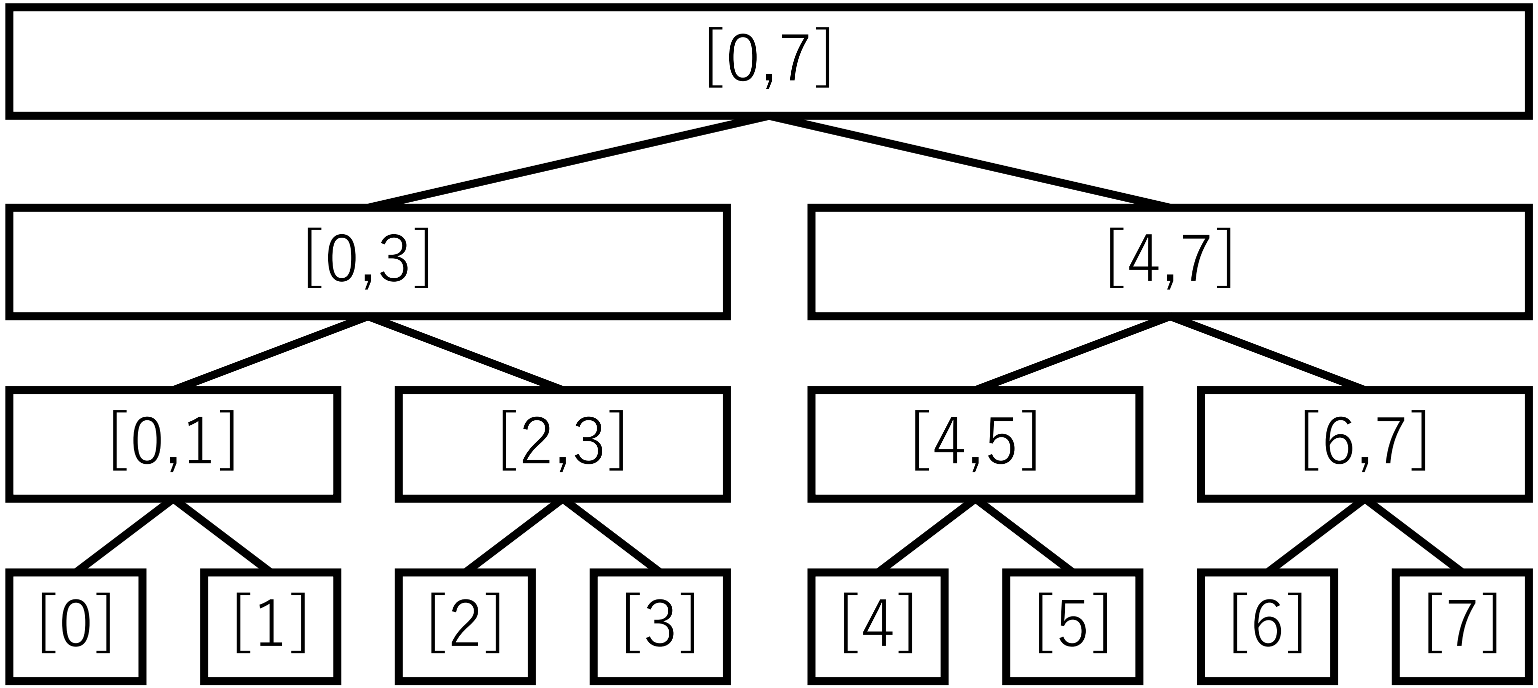
基本課題もここまでのスライドの内容ですので，
しっかりと復習していただければと思います．

以下は普通のアルゴリズムの講義では紹介されない
データ構造のご紹介．

セグメント木

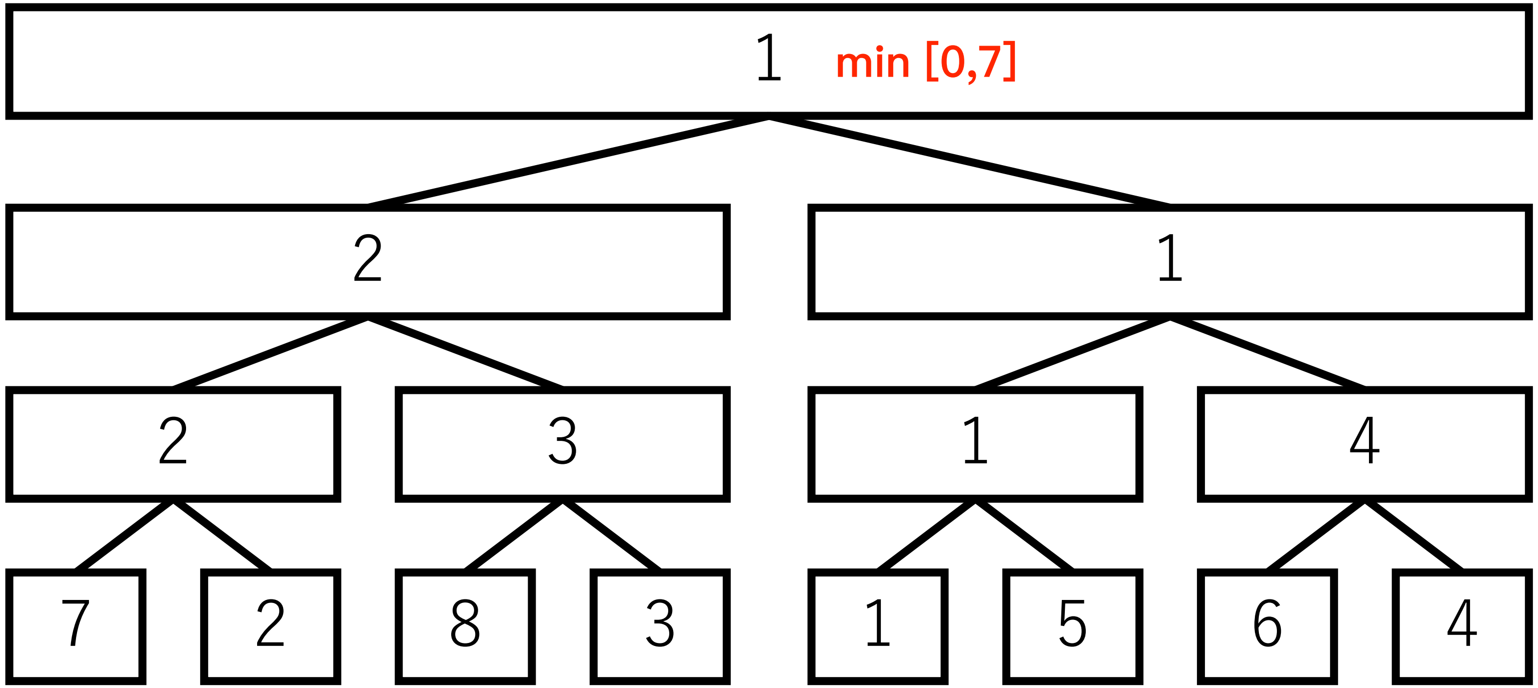
各ノードが子ノードの区間に関するある情報を保持する二分木．ある区間のクエリに対する応答を考える時に便利．二分ヒープのように配列で実装できる．

最大値/最小値



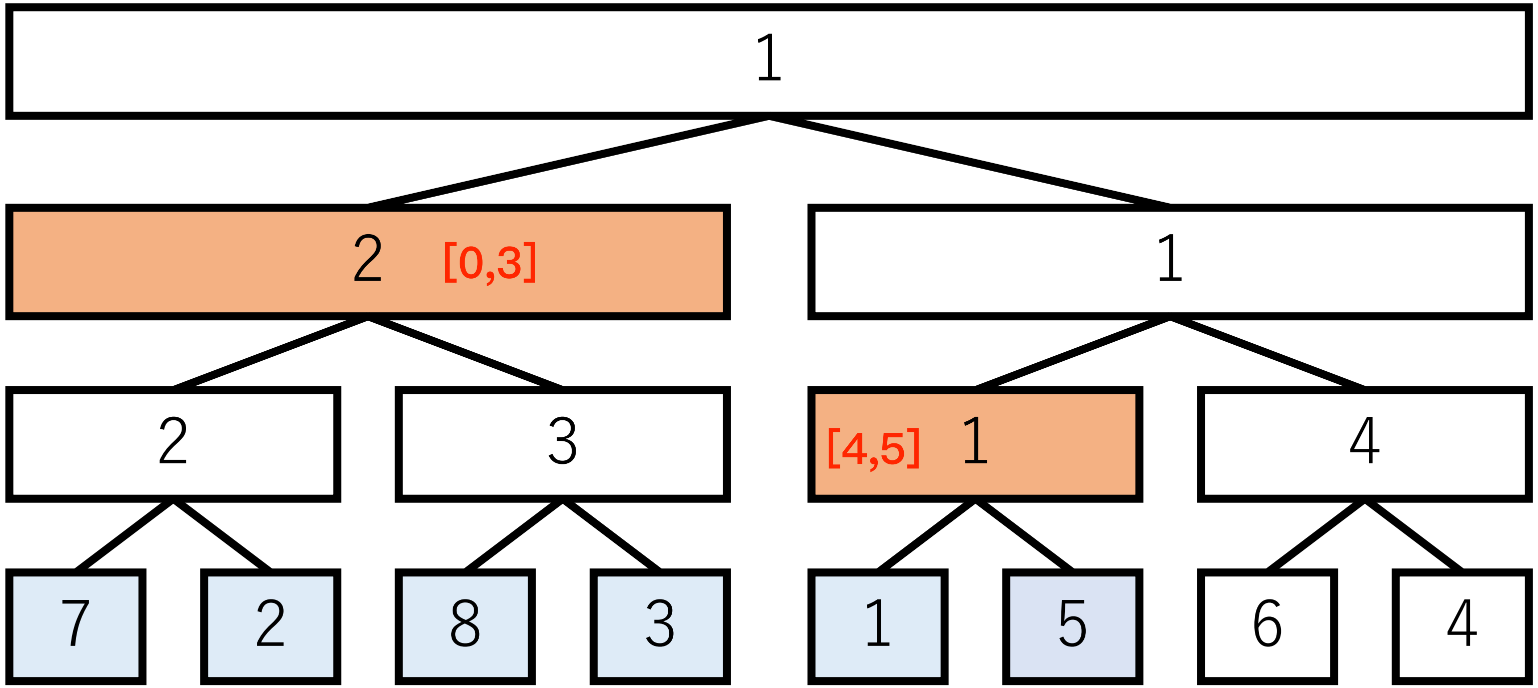
セグメント木

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



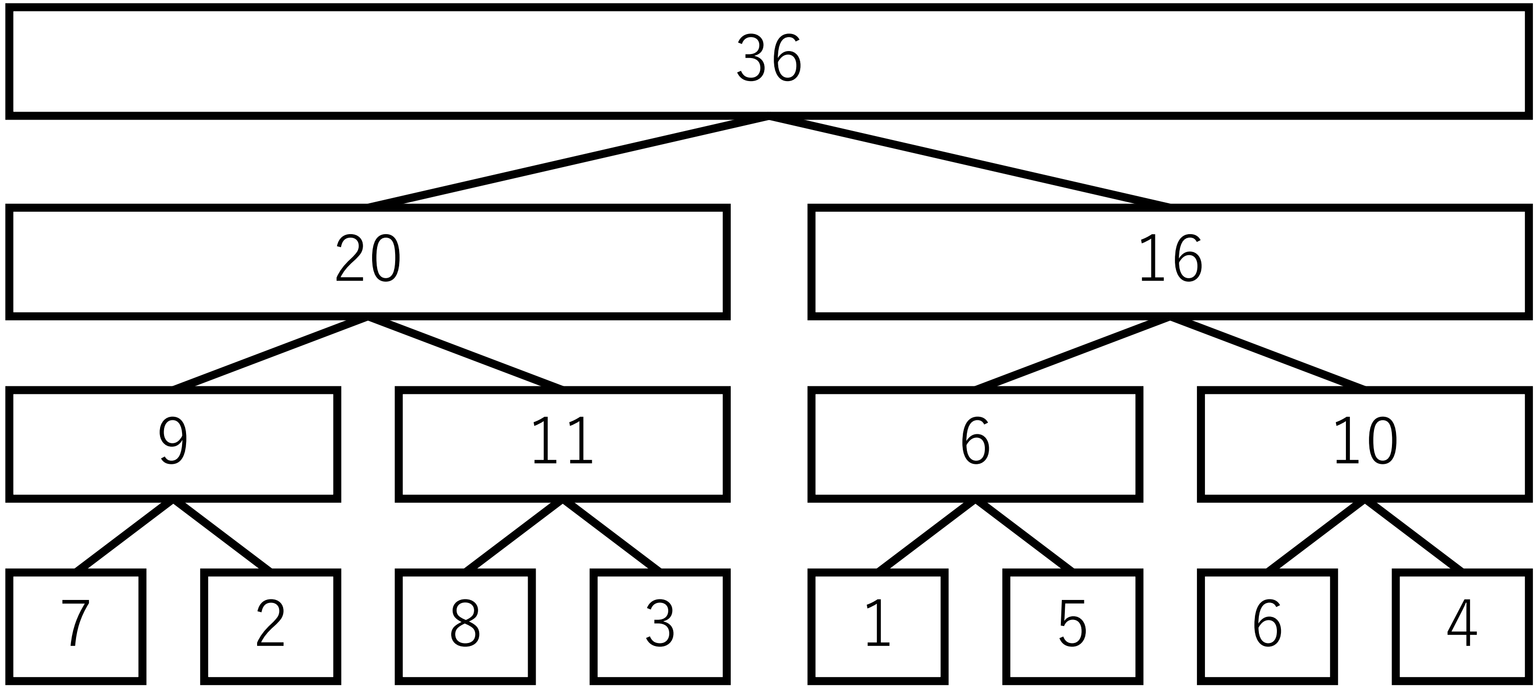
セグメント木

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.



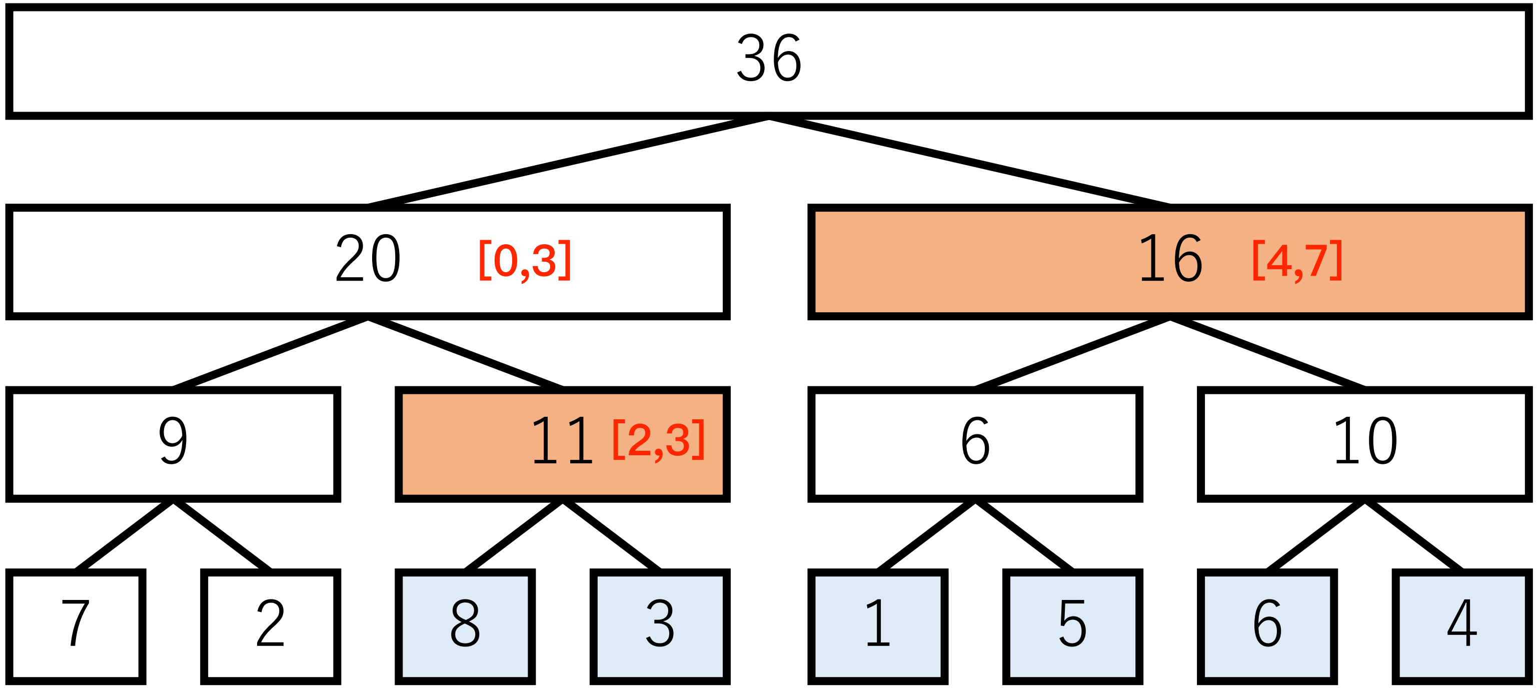
セグメント木

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． 满二叉树

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
```

冪乗

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
```

问题提供的数列

初始元素为0

为了给最下面一行分配原始数据

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
```

跨步走

セグメント木の実装例（区間和）

更改了数组中某个元素的值，更新线段树

只更新包含新节点的子树

不能直接用node_i+1

```
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):
```

私以为直接用 累積和 就能做区间和

```
    # 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
```

array[k]存着[le, re)的区间和

```
    # 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
    # （le， reも半開区間として指定される．）
```

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
```

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

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

✅左闭 右开

[le, re), [l, r)   [l, r), [le, re)

为什么这里没等号?

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
```

递归出口

```
        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
```

递归出口

```
        if (l <= le) and (re < r): return self.array[k]
```

[l [le, re) r)

セグメント木の実装例（区間和）

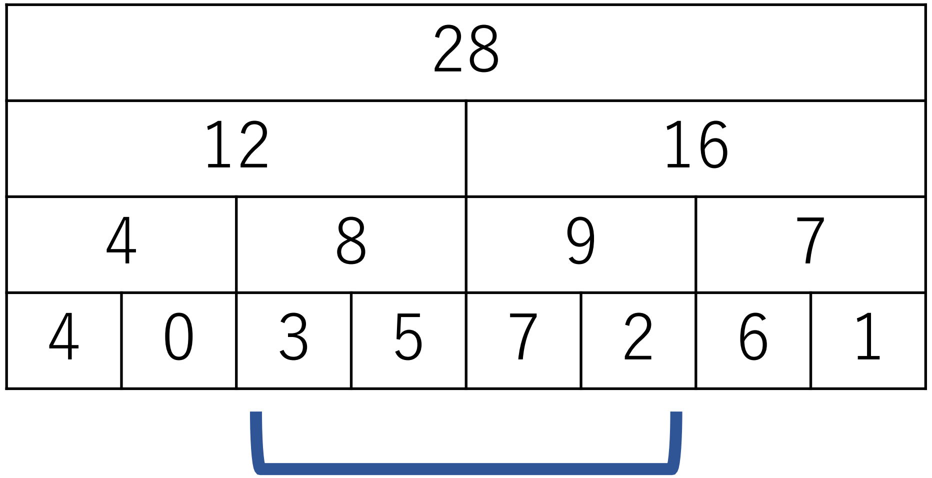
```
class CSumSegTree:
    def findSum(self, l, r, k=1, le=0, re=-1):
        ……
        # 子ノードに下がる．sum_lは左側，sum_rは
        # 右側の子ノードを見ていることになる．
        sum_l = self.findSum(l, r, 2*k, le, (le+re-1)//2)
        sum_r = self.findSum(l, r, 2*k+1, (le+re-1)//2+1, re)

        return sum_l + sum_r
```

array[2*k]的区间和代表的区间

array[2*k+1]的区间和代表的区间

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) from root

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

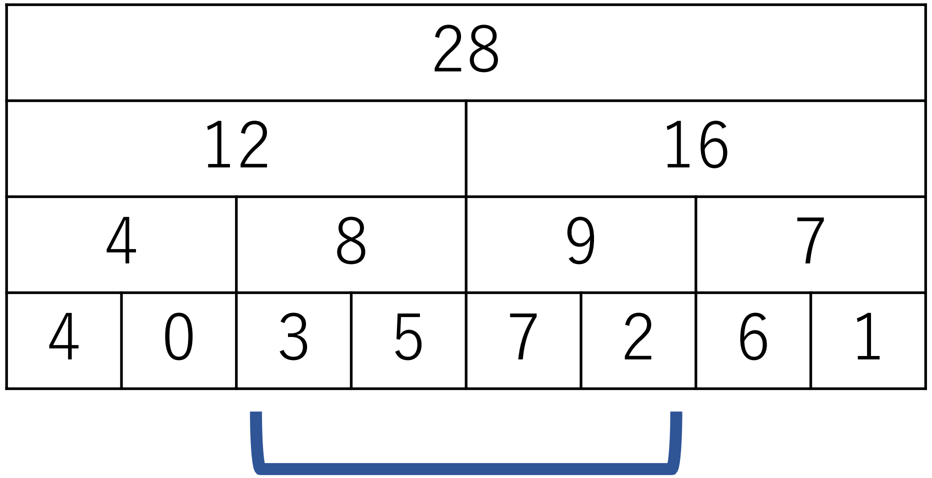
findSum(2, 6, 4, 0, 1)

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

findSum(2, 6, 3, 4, 8)

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

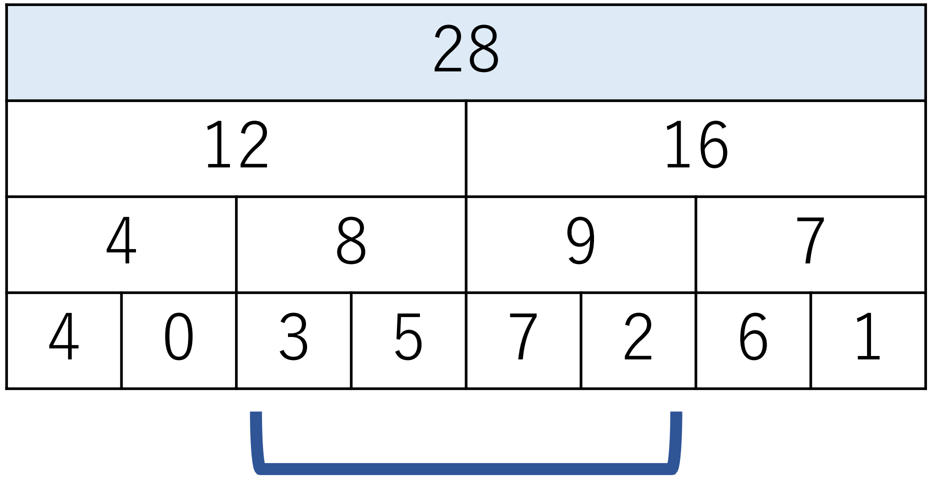
findSum(2, 6, 7, 6, 8)



セグメント木をつかった区間和計算例

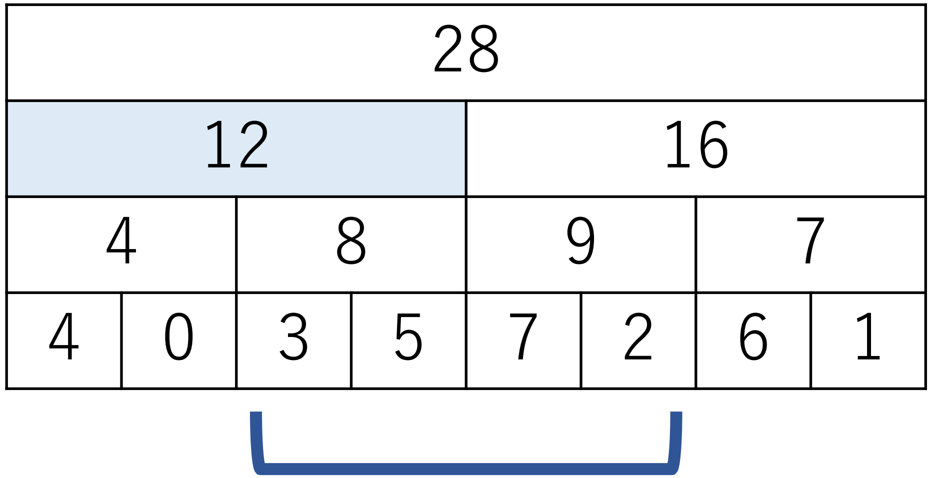
self.array：[0, 28, 12, 16, 4, 8, 9, 7, 4, 0, 3, 5, 7, 2, 6, 1]

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, 8)
findSum(2, 6, 6, 4, 5)
findSum(2, 6, 7, 6, 8)



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, 8)

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

findSum(2, 6, 7, 6, 8)

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)

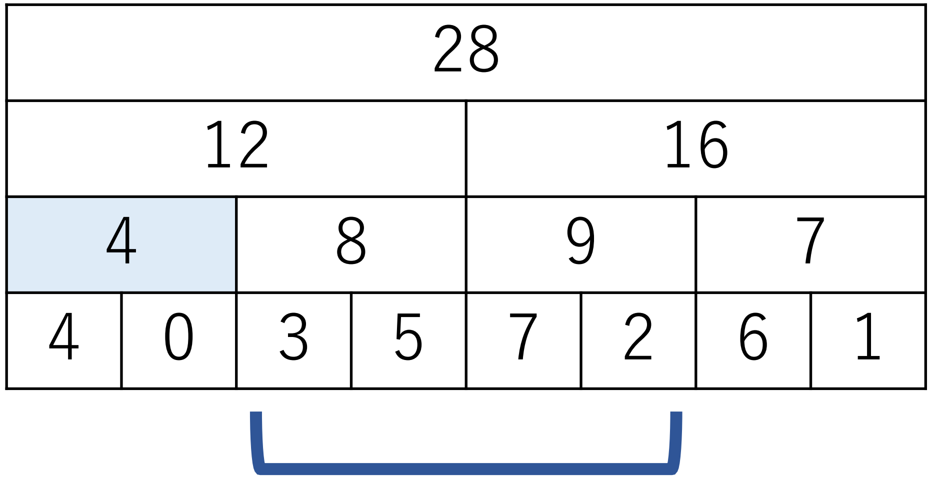
findSum(2, 6, 4, 0, 1) # 全く含まない

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

findSum(2, 6, 3, 4, 8)

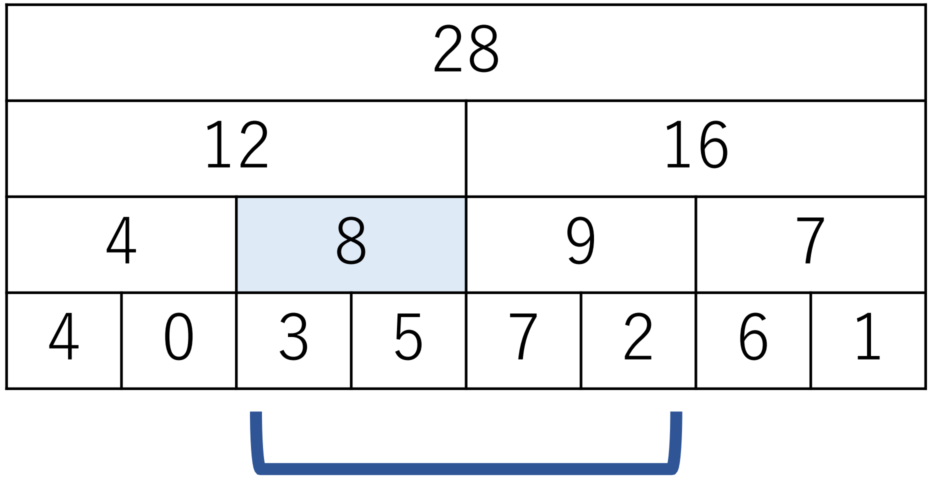
findSum(2, 6, 6, 4, 5)

findSum(2, 6, 7, 6, 8)



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, 8)
findSum(2, 6, 6, 4, 5)
findSum(2, 6, 7, 6, 8)

セグメント木をつかった区間和計算例

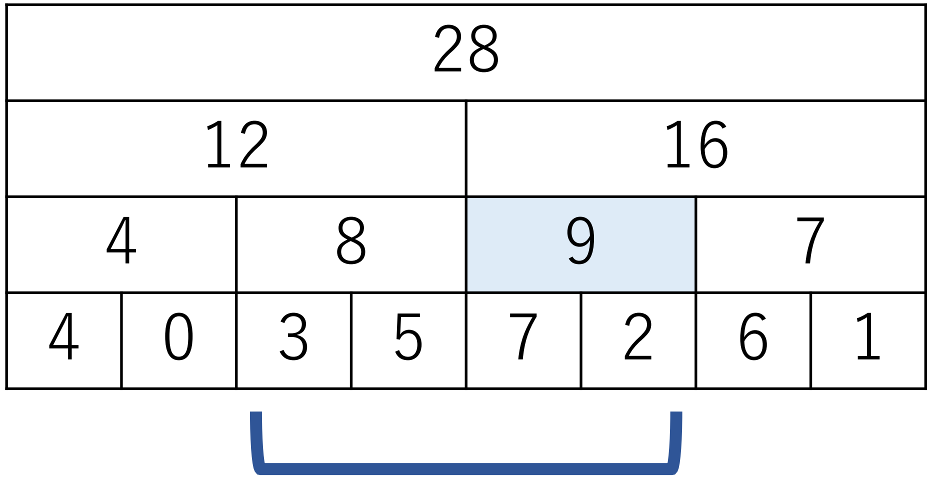
self.array：[0, 28, 12, 16, 4, 8, 9, 7, 4, 0, 3, 5, 7, 2, 6, 1]

<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, 8) # 一部含む
findSum(2, 6, 6, 4, 5)
findSum(2, 6, 7, 6, 8)

セグメント木をつかった区間和計算例

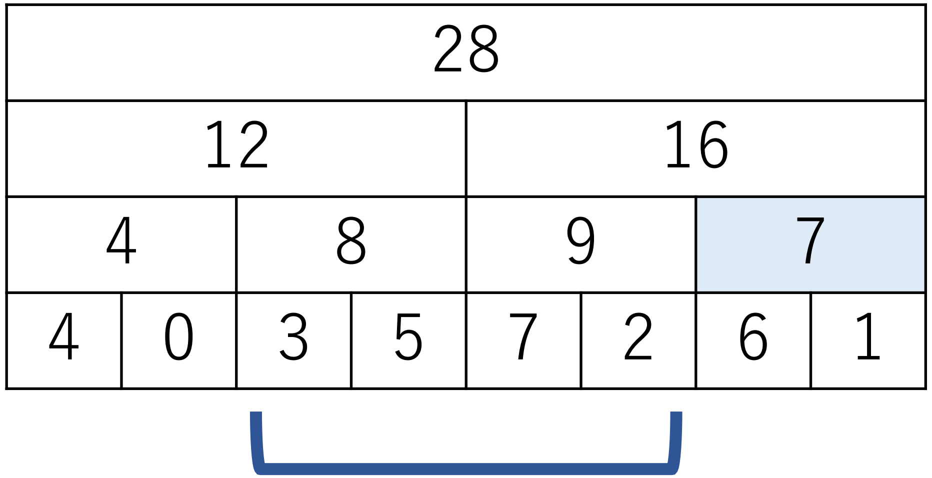
self.array：[0, 28, 12, 16, 4, 8, 9, 7, 4, 0, 3, 5, 7, 2, 6, 1]



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, 8)
findSum(2, 6, 6, 4, 5) # 全部含む
findSum(2, 6, 7, 6, 8)

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, 8)
findSum(2, 6, 6, 4, 5)
findSum(2, 6, 7, 6, 8) # **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" align="center">73</td></tr>
<tr><td colspan="4" align="center">57</td><td colspan="4" align="center">16</td></tr>
<tr><td colspan="2" align="center">4</td><td colspan="2" align="center">53</td><td colspan="2" align="center">9</td><td colspan="2" align="center">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，その計算し直しに平均的には$O(n)$かかってしまう.

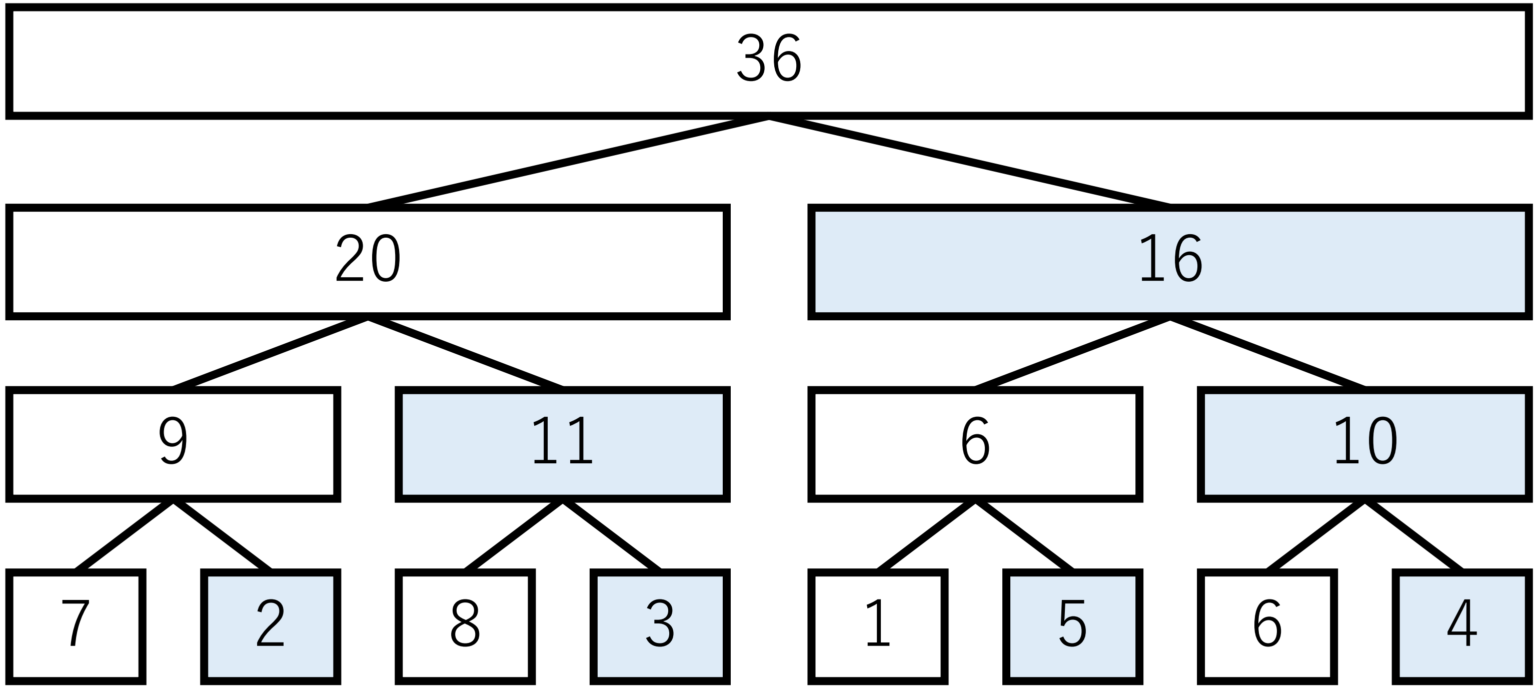
比累加法好的地方

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

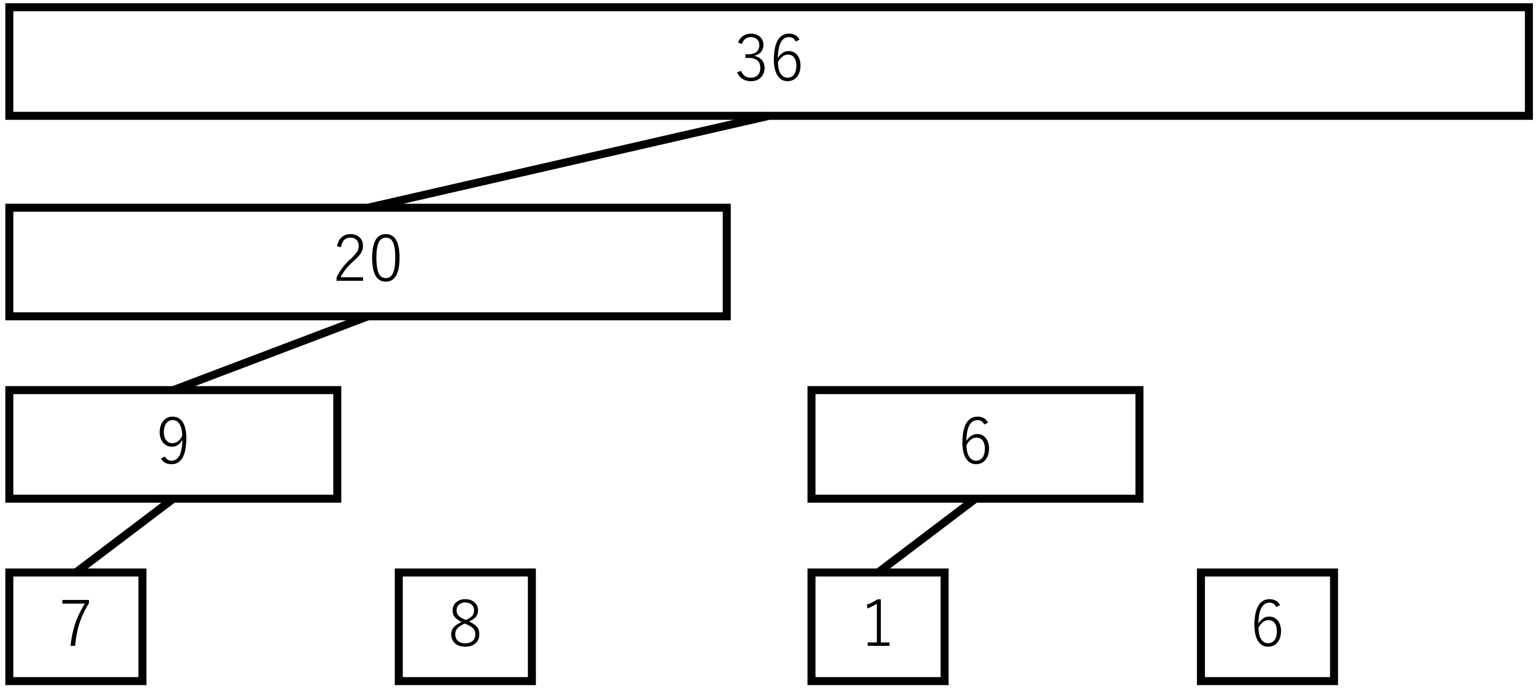
区間和のセグメント木の場合，青色部分の情報はなくても上位のノードの情報を使うことで計算できる．

如何减少空间复杂度



セグメント木

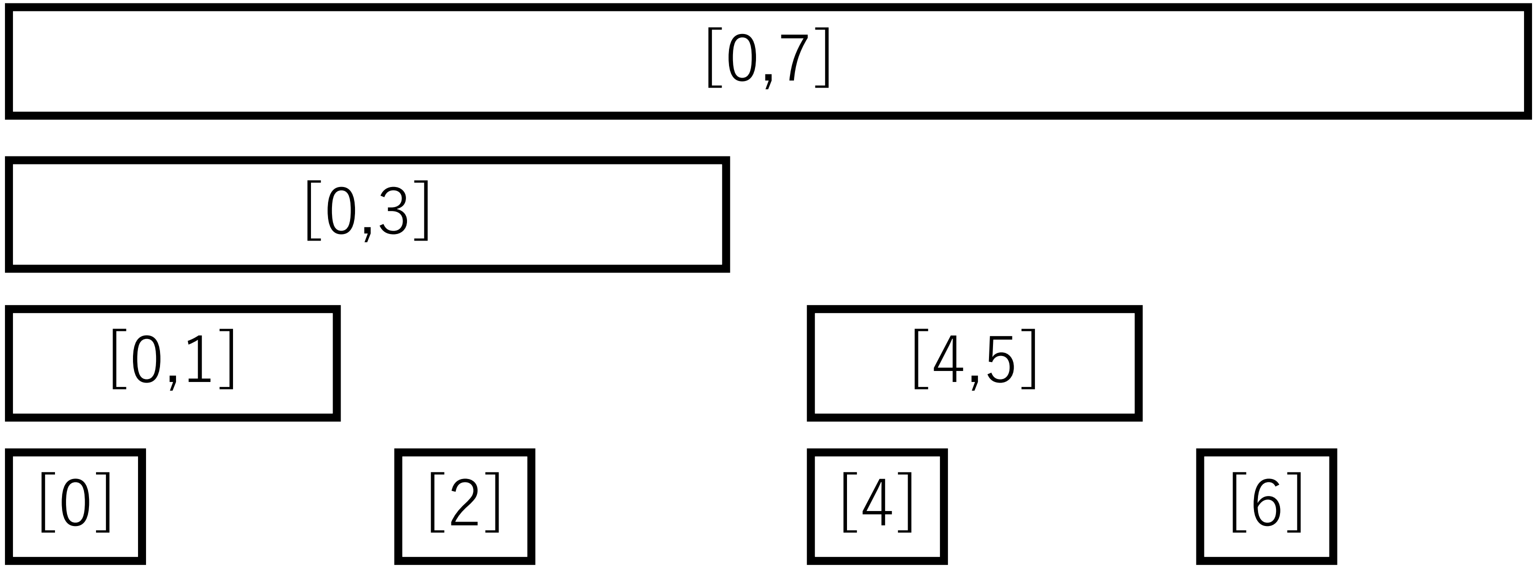
この復元可能部分を削ると，記憶領域をおよそ半分に減らすことができる．



Binary Indexed Tree（BIT）

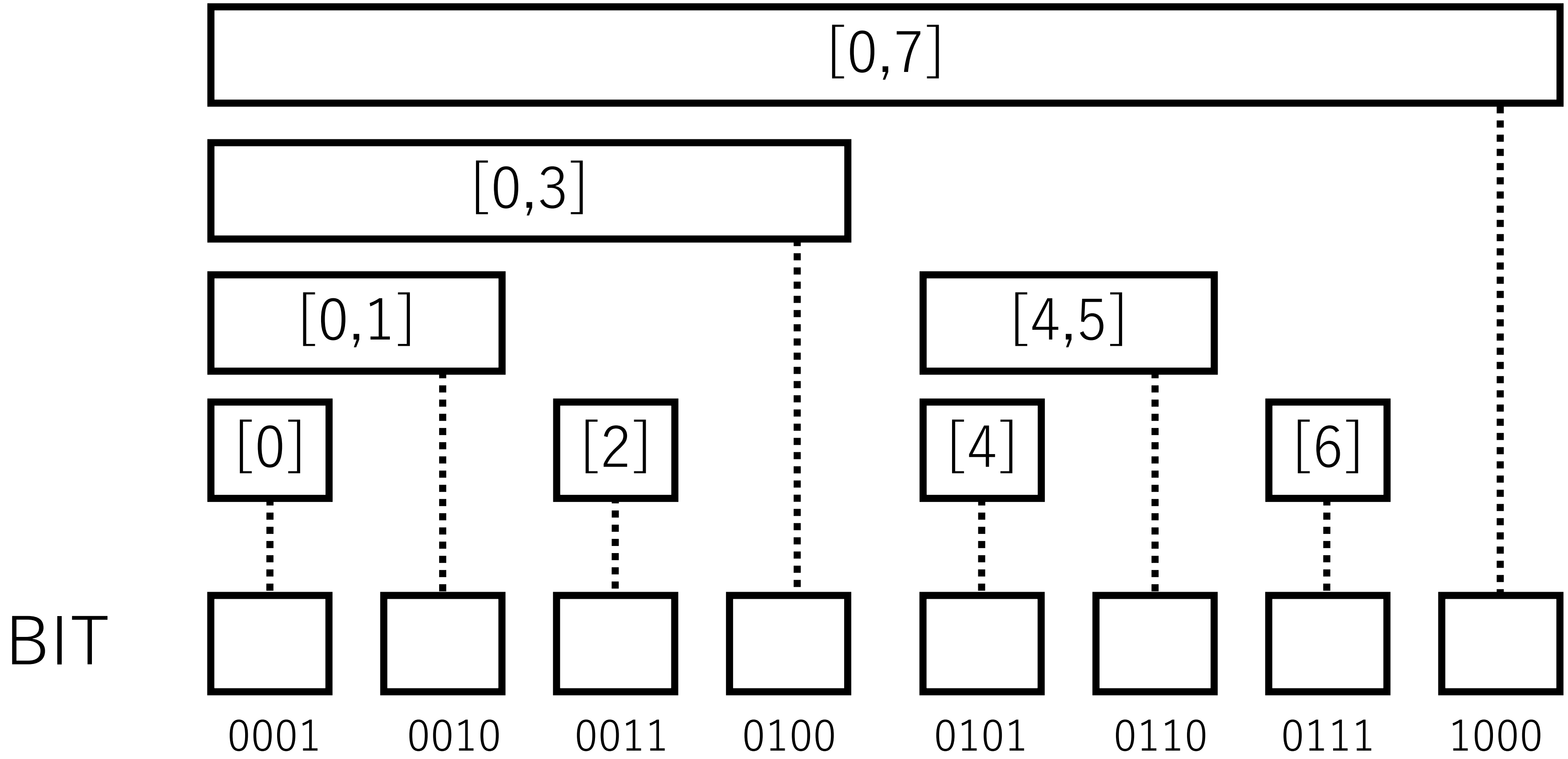
树状数组/二叉索引树

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.



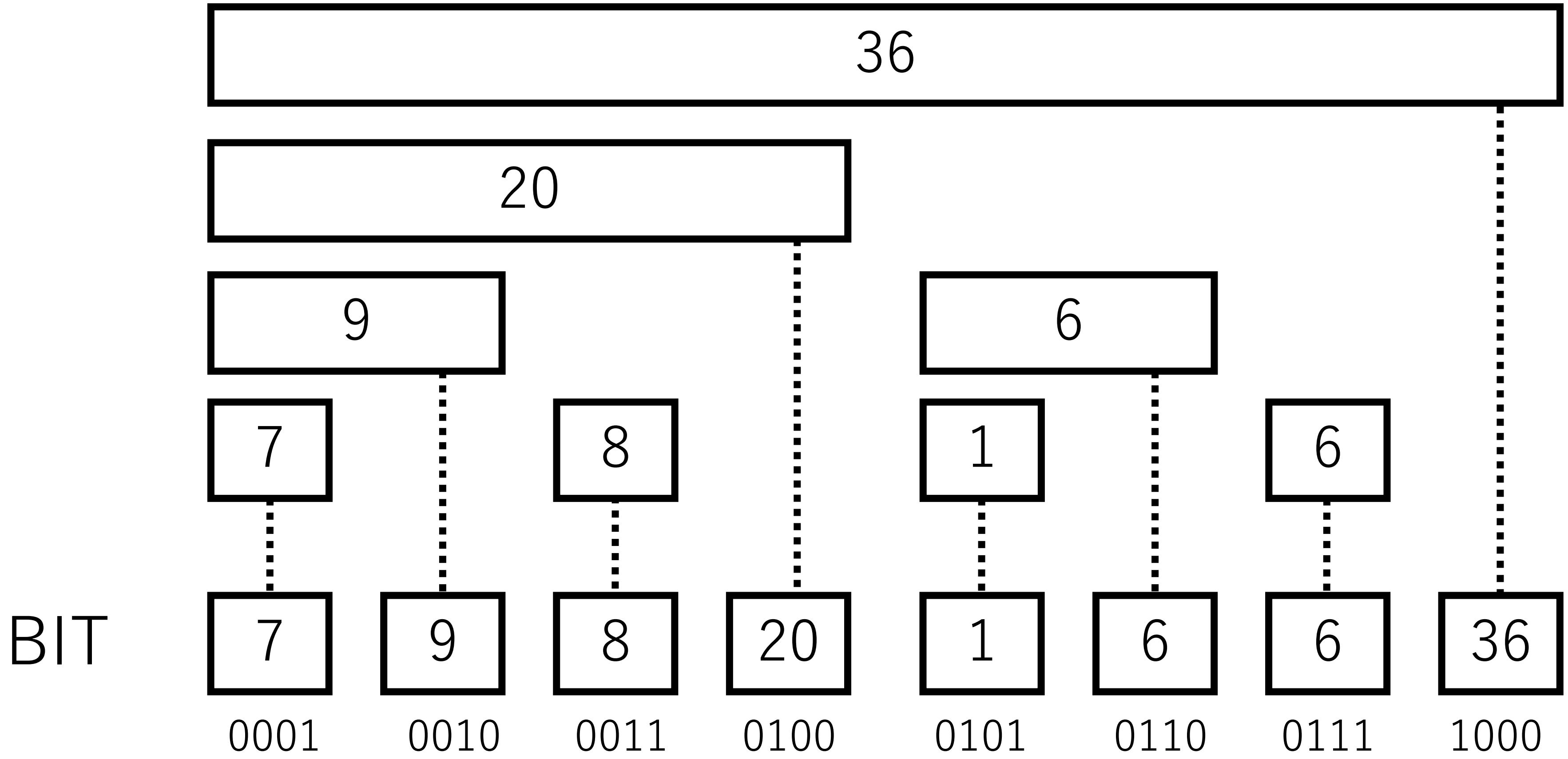
Binary Indexed Tree（BIT）

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．

基本上都可以用线段树处理

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 [1.5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．

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）