プログラミング応用 第5回

河瀬 康志

2016年7月11日

授業スケジュール

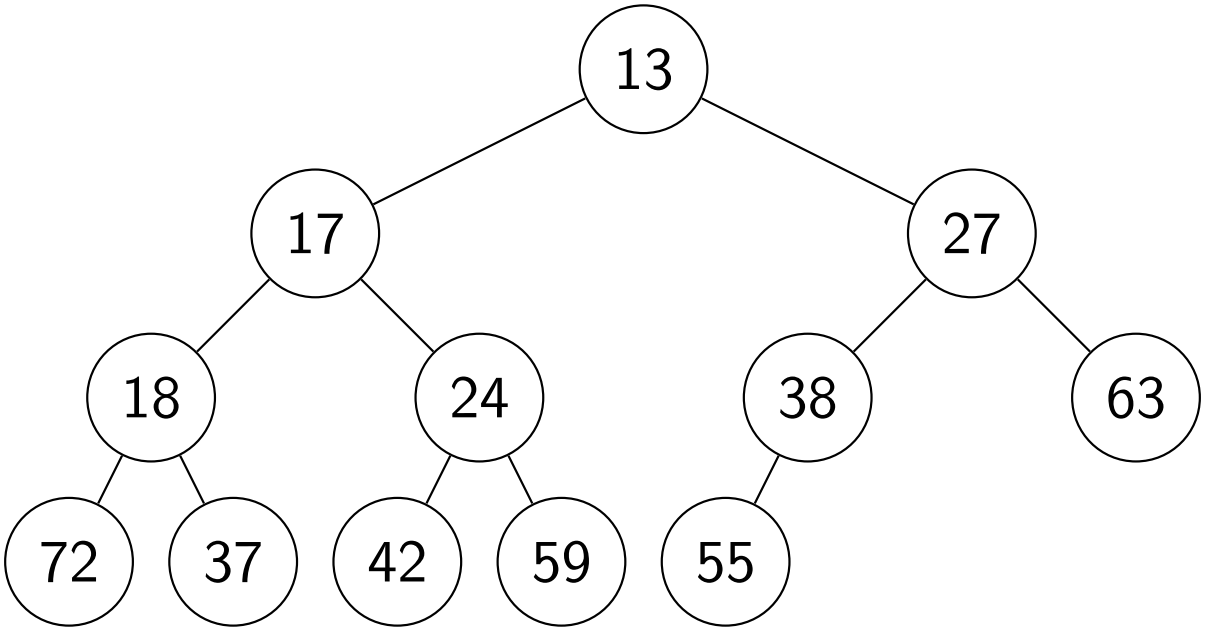
	日程	内容
第 1 回	6/13	ガイダンス・復習
第 2 回	6/20	文字列操作（文字列整形，パターンマッチ，正規表現） 平面幾何（線分の交差判定，点と直線の距離，凸包）
第 3 回	6/27	乱数（一様分布，正規分布への変換，乱数生成） 統計（データ処理，フィッティング）
第 4 回	7/4	計算量（オーダー表記） スタックとキュー（幅優先探索，深さ優先探索）
第 5 回	7/11	二分木（ヒープ，二分探索木） ソートアルゴリズム
第 6 回	7/18	バックトラック（N クイーン問題，数独） 動的計画法（ナップサック問題）
第 7 回	7/25	最短経路探索（Warshall-Floyd, Bellman-Ford, Dijkstra） TSP（全探索，ビット DP, ヒューリスティックス）
期末試験	8/1	—

アウトライン

1. 二分木
2. ソートアルゴリズム
3. 演習

ヒープ

- どのノードの値も，その全ての子の値以下であるような完全二分木
  - 二分木: 各ノードの子は高々2 つ
  - 完全: 高さは完全にバランスしていて，ノードは左から右に埋まる



ヒープ操作の計算量

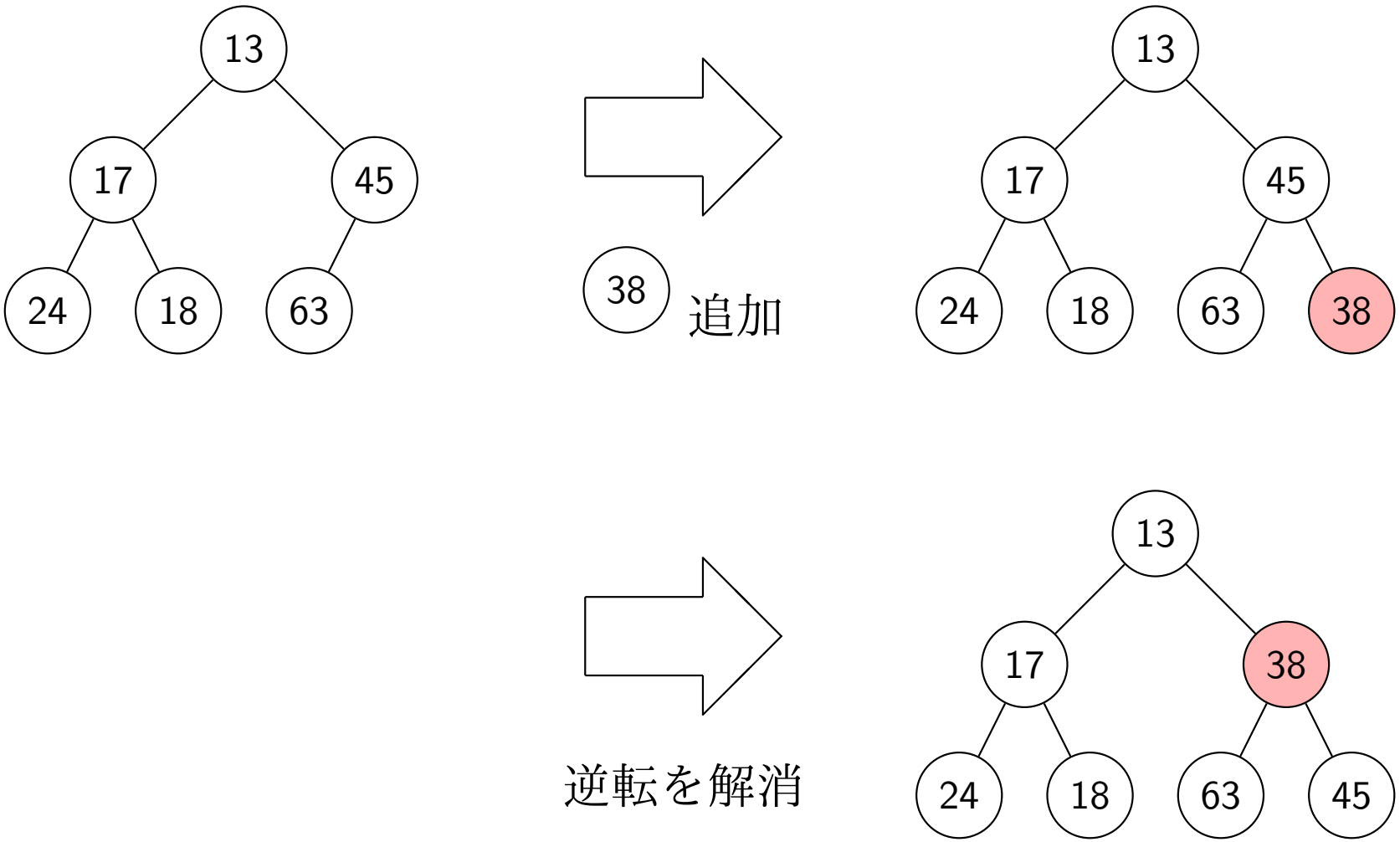
計算量 (要素数 n)

- 最小値を取得する $O(1)$
- 要素数を取得する $O(1)$
- 最小値の要素を削除する $O(\log n)$
- 要素を挿入する $O(\log n)$

動的な最小値の更新を効率よく行える

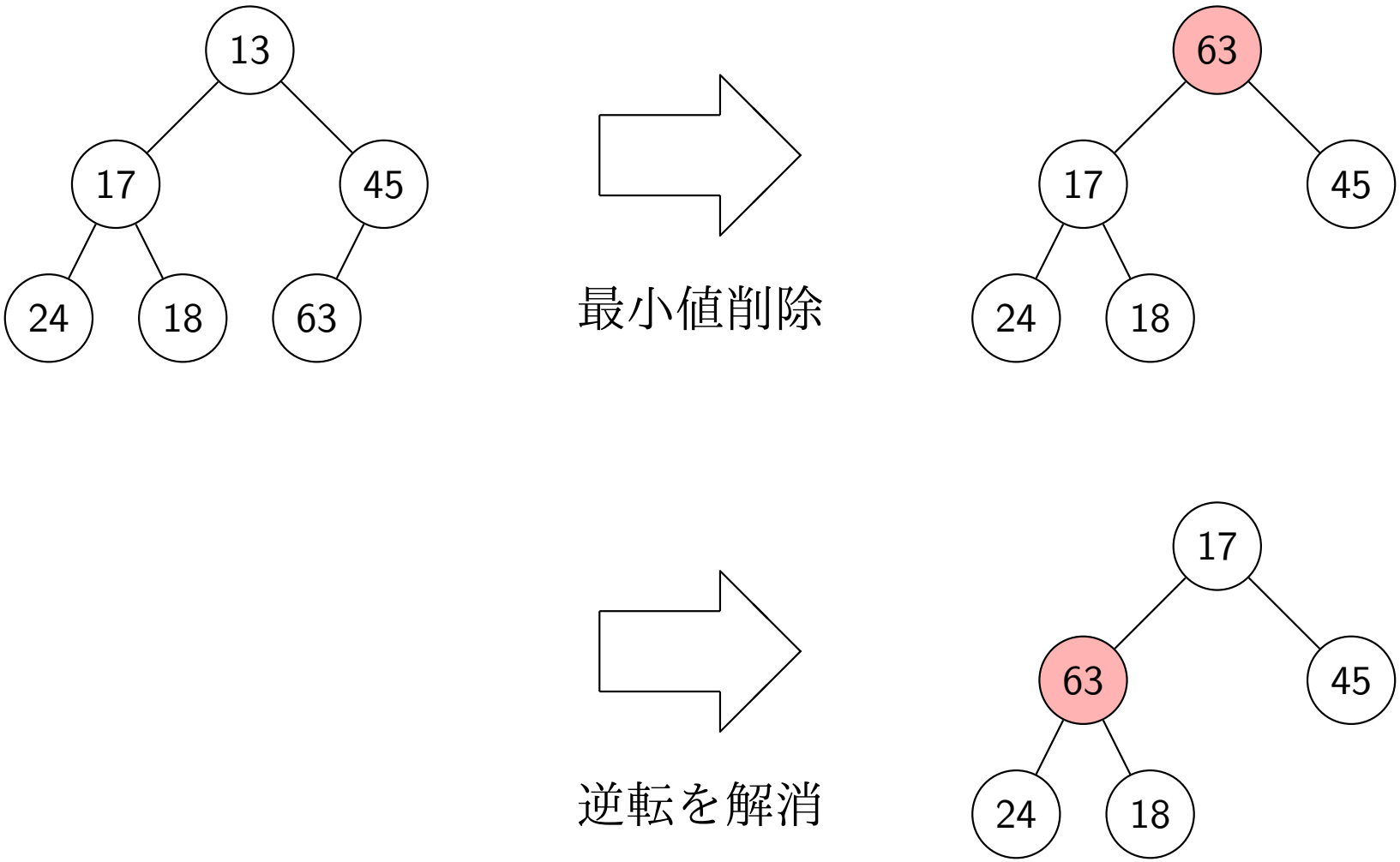
ヒープへの要素の挿入

新たな要素は末尾に追加し，逆転がなくなるまで上げていく



ヒープの最小値の削除

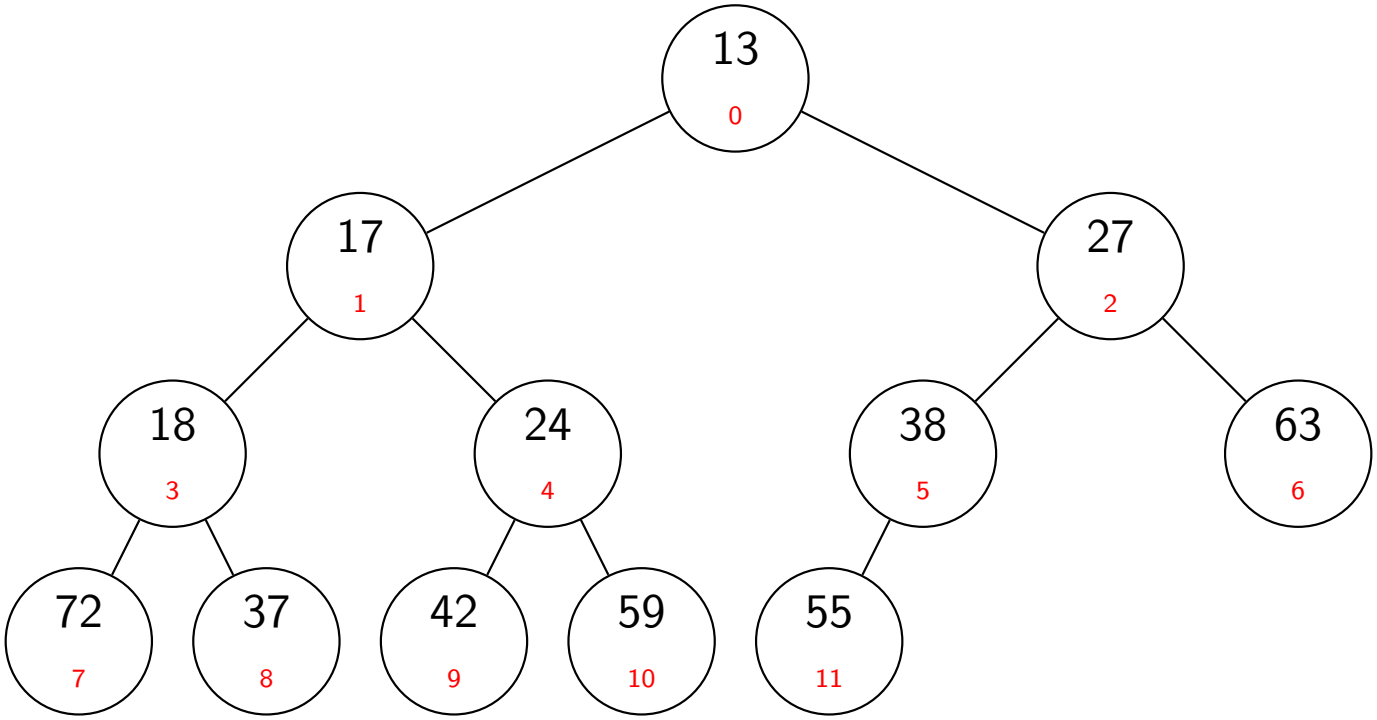
ルートに末尾の要素を移動し，逆転がなくなるまで下げていく



ヒープの実装

リストで表現できる

k 番目の要素の子は $2k+1$ と $2k+2$



Python では

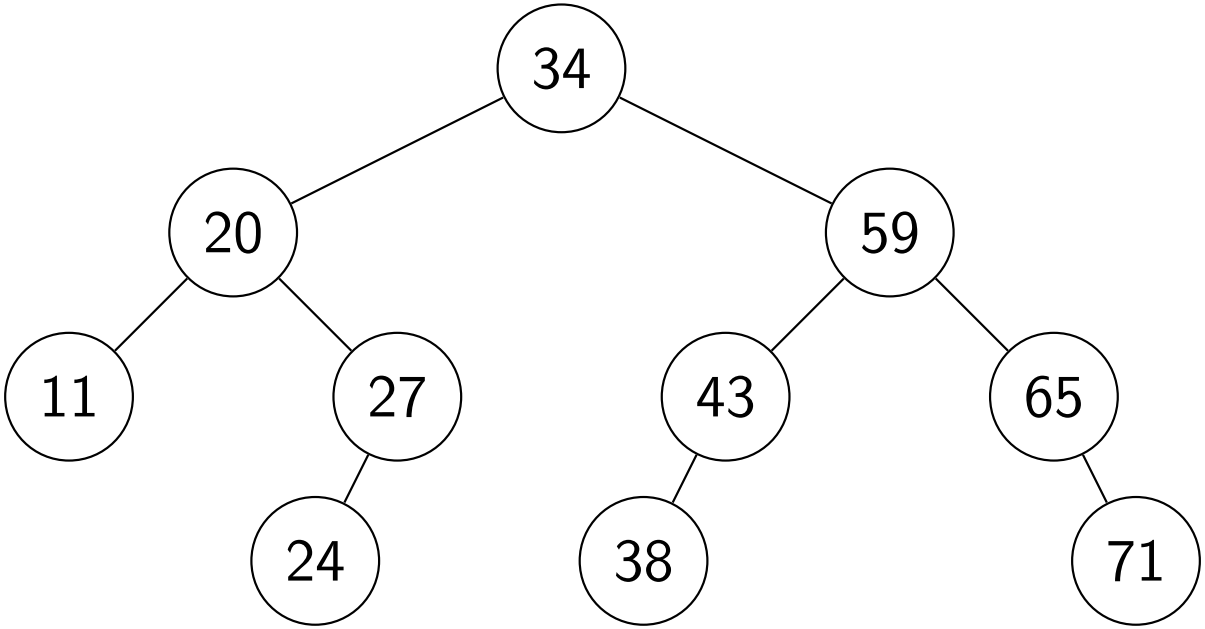
- import heapq でヒープ操作が使えるようになります
- リスト上の操作として定義されています

```
>>> import heapq
>>> h = [20,30,10]
>>> heapq.heapify(h) # リストをヒープに変換します
>>> h
[10, 30, 20]
>>> heapq.heappush(h,5) # h に 5 を追加
>>> heapq.heappush(h,12) # h に 12 を追加
>>> h
[5, 10, 20, 30, 12]
>>> h[0] # 最小値
5
>>> heapq.heappop(h)
5
>>> h
[10, 12, 20, 30]
```

二分探索木

どのノードの値も以下の性質を満たすような二分木

- 左の子以下のどのノードの値より大きい
- 右の子以下のどのノードの値より小さい



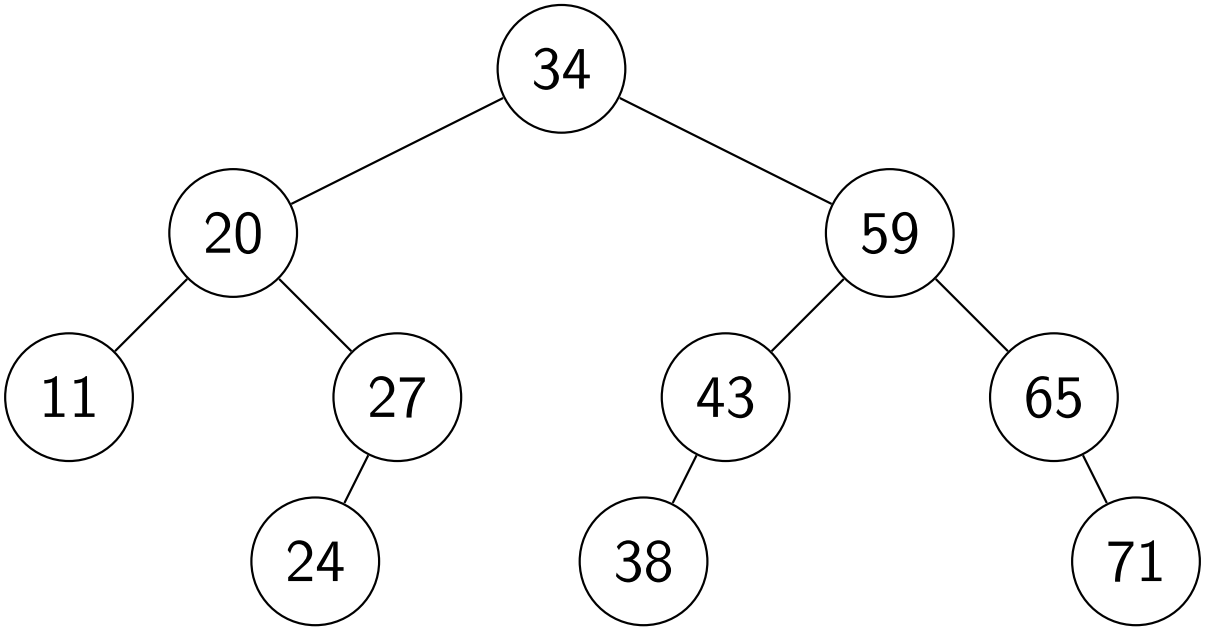
二分探索木操作の計算量

平均計算量 (要素数 n)

- 指定した要素が含まれるか判定 $O(\log n)$
- 指定した要素を削除 $O(\log n)$
- 要素を追加する $O(\log n)$
- 最小値，最大値の取得 $O(\log n)$
- 要素数を取得する $O(1)$

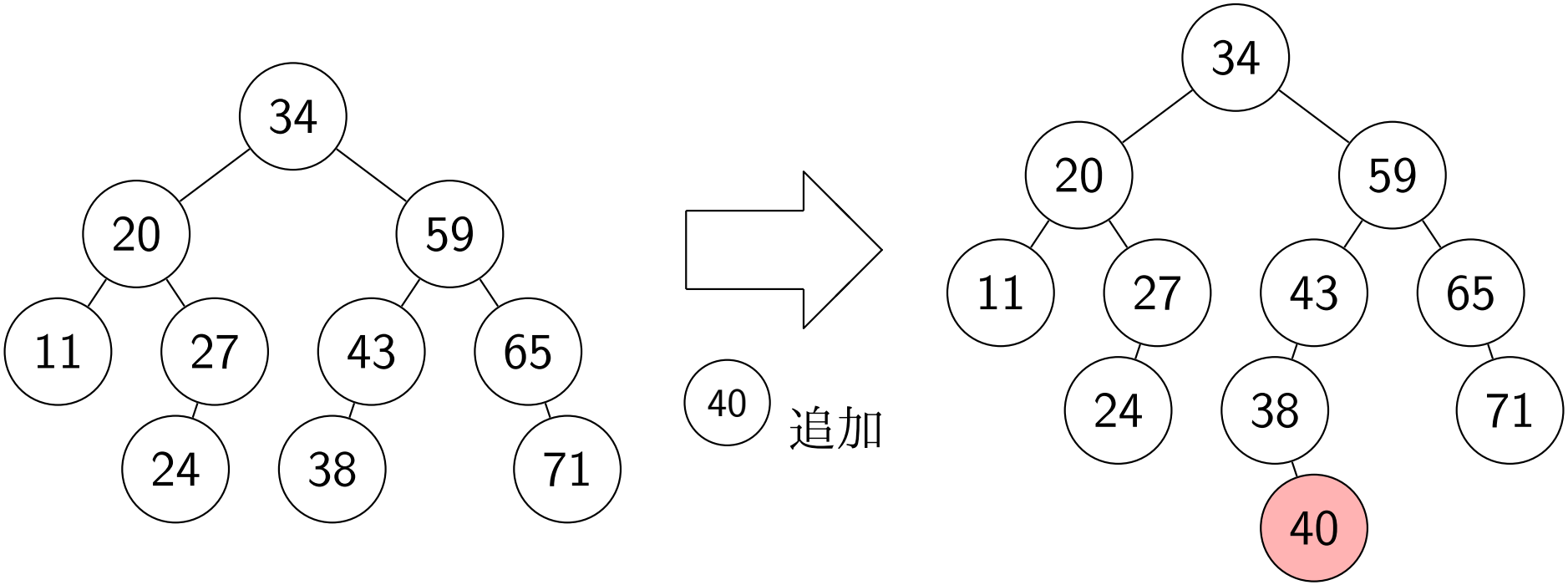
数の集合を効率よく管理できる

指定した要素の探索



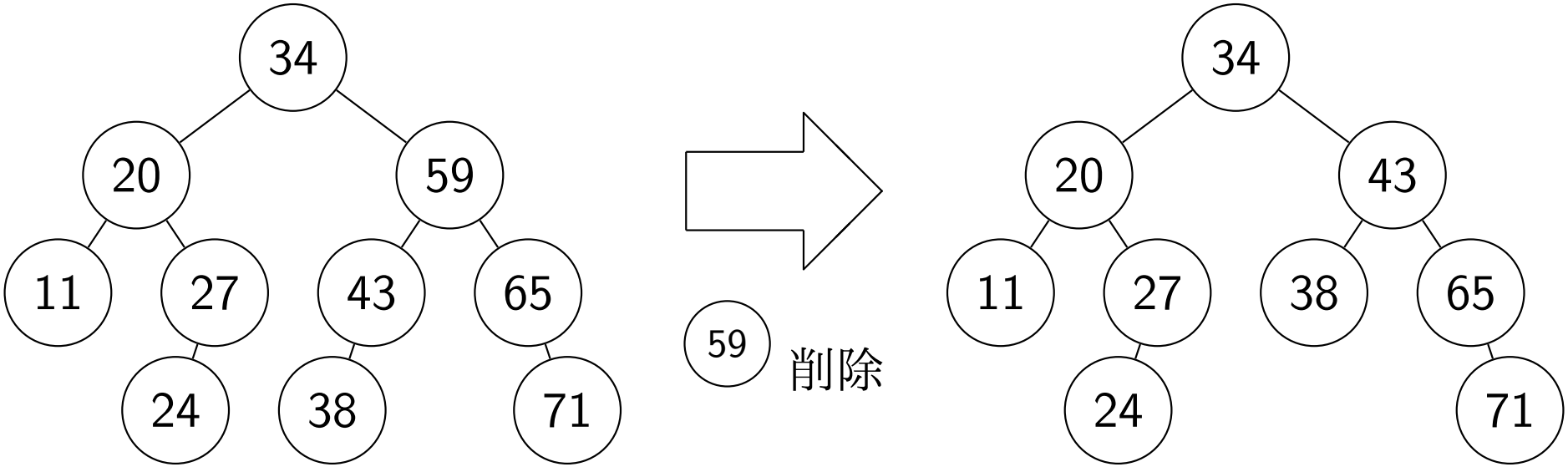
- 根のノードから探索
  - 指定された要素の方が大きければ右の子から下を探索
  - 指定された要素の方が小さければ左の子から下を探索
  - 同じ値ならば発見!
- 一番下 (葉) まで指定された要素が見つからなければ，存在しない

要素の挿入



根から要素の探索を行い，葉として取り付ける

指定した要素の削除



- 削除したいノードが左の子を持っていなければ，右の子をもってくる
- 左の子が右の子をもっていなければ，左の子をもってくる
- 左の子以下で最大のノードをもってきて，そのノードの下を付け直す

二分探索木操作の実装例

```
class Node:
    def __init__(self,value):
        self.left = None
        self.right = None
        self.value = value

class Tree:
    def __init__(self):
        self.root = None
        self.size = 0
```

Tree クラスのメソッドとして insert, find, remove を実装する

二分探索木操作の実装例 2

```
def insert(self,value):
  self.size += 1
  if self.root == None:
    self.root = Node(value)
  else:
    node = self.root
    while True:
      if node.value < value:
        if node.right == None:
          node.right = Node(value)
          break
        else:
          node = node.right
      elif node.value > value:
        if node.left == None:
          node.left = Node(value)
          break
        else:
          node = node.left
      else: # node.value == value
        break
```

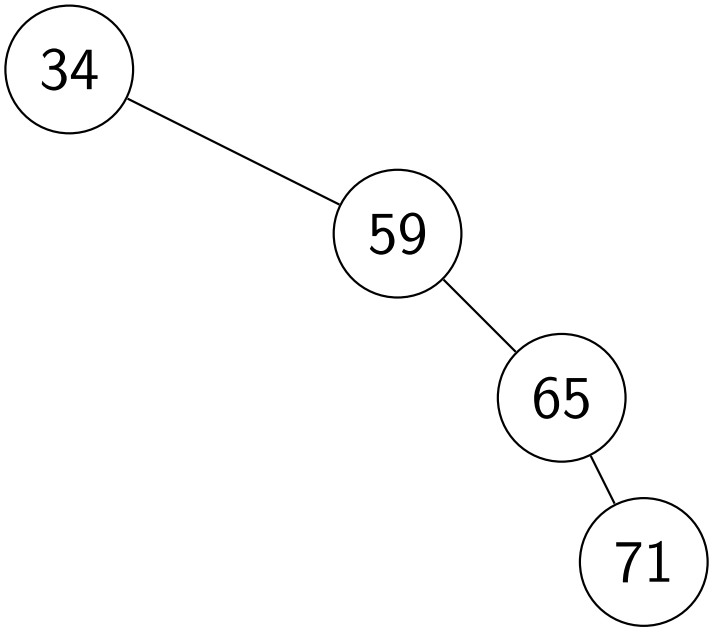
```
def find(self,value):
    node = self.root
    while node:
        if node.value < value:
            node=node.right
        elif node.value > value:
            node=node.left
        else:
            return True
    return False
```

```
t=Tree()
t.insert(3)
t.insert(7)
t.insert(5)
t.insert(4)
print t.find(3) # True
print t.find(6) # False
```

remove の実装（の一部）は演習課題

実際に利用するときには

- 木の高さがバランスを保たないと効率が悪くなる
- 平衡化の仕組みが必要 ⇒ 平衡二分探索木
  - AVL 木，赤黒木，スプレー木，Treap など
- 自分で実装するのは大変



ハッシュテーブル

- ハッシュ関数： 入力データに対し，固定長の数値を返す関数
- ハッシュ値： ハッシュ関数が返す値
- ハッシュテーブル： データをハッシュ値に従ってリストに格納したテーブル
  - 違うデータでも同じ場所に格納されるかも
  - 同じ場所に格納されると処理が遅くなる

ハッシュテーブルの性質

- 要素の追加，削除，存在判定が定数時間でできる
- データが大きいとハッシュ値の計算が大変
- リストのサイズは，データ数に応じ大きくしなければならない

Python では

- 辞書, set の正体はハッシュテーブルである
- ハッシュ値は hash の値を元に計算される

```
>>> hash('a')
12416037344
>>> hash(0.1)
2576882278
```

データ構造の比較

	ヒープ	(平衡) 二分探索木	ハッシュテーブル
挿入	$O(\log n)$	$O(\log n)$	$O(1)$
指定要素検索	$O(n)$	$O(\log n)$	$O(1)$
指定要素削除	$O(n)$	$O(\log n)$	$O(1)$
最小値取得	$O(1)$	$O(\log n)$	$O(n)$
最小値削除	$O(\log n)$	$O(\log n)$	$O(n)$
k 番目の要素取得	$O(k \log n)$	$O(\log n)$	—
要素数	$O(1)$	$O(1)$	$O(1)$
メモリ使用量	小	中	大

アウトライン

1. 二分木
2. ソートアルゴリズム
3. 演習

ソートアルゴリズム

- 与えられた数の列を小さい順に並べ替える
  - ソート前：$[8, 4, 5, 6, 1, 2, 7]$，ソート後：$[1, 2, 4, 5, 6, 7, 8]$
- 目標
  - sorted をメソッド使わずにソートをできるようになる
  - 効率的にソートをするためにはどのようにすればよいかを学ぶ

```
>>> sorted([8,4,5,6,1,2,7])
[1, 2, 4, 5, 6, 7, 8]
```

バブルソート

最も単純な方法の 1 つ

- 隣合う要素を比較し，順序が逆になっていれば入れ替える
- 一度リストを捜査すると，最大の要素が最後に移動する
- 計算量は $O(n^2)$ (n はリストの長さ)

```
def bubble_sort(l):
    for i in range(len(l)):
        for j in range(len(l)-i-1):
            if l[j]>l[j+1]:
                l[j],l[j+1]=l[j+1],l[j]
```

ヒープソート

- ヒープを用いたソート方法
  - リストの要素を全てヒープに挿入する
  - ヒープから 1 つずつ最小の要素を取り出す
- 計算量は $O(n \log n)$ (n はリストの長さ)

```
def heap_sort(l):
    h = []
    for i in range(len(l)): heapq.heappush(h,l.pop())
    for i in range(len(h)): l.append(heapq.heappop(h))
```

マージソート

- 分割統治法を用いてソートする
  - 前半分と後ろ半分をそれぞれソート (サイズ $n/2$ の問題 2 つ)
  - 2 つの結果をくっつける ($O(n)$ 時間)
- 計算量は $O(n \log n)$

```
def merge_sort(l):
    _merge_sort(l,0,len(l))
def _merge_sort(l,s,t):
    if t-s<=1: return
    m=(s+t)/2
    _merge_sort(l,s,m) # 前半をソート
    _merge_sort(l,m,t) # 後半をソート
    a,j,k = [],s,m
    # merge
    for i in range(s,t):
        if k==t or (j<m and l[j]<l[k]):
            a.append(l[j])
            j+=1
        else:
            a.append(l[k])
            k+=1
    l[s:t]=a
```

マージソート

くっつけかた

- l: $\underset{j}{4}, 5, 8,$ $\underset{k}{1}, 2, 6, 7,$
- a:

```
# merge
for i in range(s,t):
    if k==t or (j<m and l[j]<l[k]):
        a.append(l[j])
        j+=1
    else:
        a.append(l[k])
        k+=1
l[s:t]=a
```

マージソート

くっつけかた

- l: $\underset{j}{4}, 5, 8,$ $\quad 1, \underset{k}{2}, 6, 7,$
- a: 1,

```
# merge
for i in range(s,t):
    if k==t or (j<m and l[j]<l[k]):
        a.append(l[j])
        j+=1
    else:
        a.append(l[k])
        k+=1
l[s:t]=a
```

マージソート

くっつけかた

- l: $\underset{j}{4}, 5, 8,$ $\quad 1, 2, \underset{k}{6}, 7,$
- a: 1, 2,

```
# merge
for i in range(s,t):
    if k==t or (j<m and l[j]<l[k]):
        a.append(l[j])
        j+=1
    else:
        a.append(l[k])
        k+=1
l[s:t]=a
```

マージソート

くっつけかた

- l: $4, \underset{j}{5}, 8, \qquad 1, 2, \underset{k}{6}, 7,$
- a: 1, 2, 4,

```
# merge
for i in range(s,t):
    if k==t or (j<m and l[j]<l[k]):
        a.append(l[j])
        j+=1
    else:
        a.append(l[k])
        k+=1
l[s:t]=a
```

マージソート

くっつけかた

- l: $4, 5, \underset{j}{8}, \qquad 1, 2, \underset{k}{6}, 7,$
- a: 1, 2, 4, 5,

```
# merge
for i in range(s,t):
    if k==t or (j<m and l[j]<l[k]):
        a.append(l[j])
        j+=1
    else:
        a.append(l[k])
        k+=1
l[s:t]=a
```

マージソート

くっつけかた

- l: $4, 5, \underset{j}{8}, \qquad 1, 2, 6, \underset{k}{7},$
- a: 1, 2, 4, 5, 6,

```
# merge
for i in range(s,t):
    if k==t or (j<m and l[j]<l[k]):
        a.append(l[j])
        j+=1
    else:
        a.append(l[k])
        k+=1
l[s:t]=a
```

マージソート

くっつけかた

- l: $4, 5, \underset{j}{8}, \qquad 1, 2, 6, 7, \underset{k}{}$
- a: 1, 2, 4, 5, 6, 7,

```
# merge
for i in range(s,t):
    if k==t or (j<m and l[j]<l[k]):
        a.append(l[j])
        j+=1
    else:
        a.append(l[k])
        k+=1
l[s:t]=a
```

マージソート

くっつけかた

- l: $4, 5, 8, \underset{j}{} \quad 1, 2, 6, 7, \underset{k}{}$
- a: 1, 2, 4, 5, 6, 7, 8

```
# merge
for i in range(s,t):
    if k==t or (j<m and l[j]<l[k]):
        a.append(l[j])
        j+=1
    else:
        a.append(l[k])
        k+=1
l[s:t]=a
```

クイックソート

- 実用的にはとても高速な方法
  - 適当な pivot を選び、それより大きいものと小さいものに分割
  - それぞれをクイックソート
- 計算量は $O(n \log n)$ (最悪の場合は $O(n^2)$)

```
# あまり早くない実装
def quick_sort(l):
    if len(l)<=1: return
    pivot = random.choice(l)
    le = [i for i in l if i<pivot]
    eq = [i for i in l if i==pivot]
    gr = [i for i in l if i>pivot]
    quick_sort(le)
    quick_sort(gr)
    l[:]=(le+eq+gr)
```

おまけ：線形探索と二分探索

リストに入った指定データがどこにあるか検索する方法

- 線形探索：
  - 先頭から順に調べていく方法
  - 計算量は $O(n)$
- 二分探索：
  - 昇順にソート済みのリストに対し，中央の値を見て，指定データの方が大きければ後半を，小さければ前半を再帰的に調べる．
  - 計算量は $O(\log n)$

```
def binary_search(l,x):
    lo, hi = 0, len(l)
    while hi>lo:
        mid = (lo+hi)//2
        if l[mid]<x: lo=mid+1
        else: hi=mid
    return lo
```

おまけ2：k 番目に小さい値

ソートされていないデータにおいて k 番目に小さい値を見つける方法

- 昇順にソートして k 番目の値を見る $\Rightarrow$ $O(n \log n)$ 時間
- クイックソートに似た分割統治法 $\Rightarrow$ $O(n)$ 時間

```
def find(l,k):
    n = len(l)
    if n==1: return l[0]
    pivot = random.choice(l)
    le = [i for i in l if i<pivot]
    gr = [i for i in l if i>pivot]
    if len(le)>=k: return find(le,k)
    if n-len(gr)<k: return find(gr,k-(n-len(gr)))
    return pivot
```

計算量は $T(n) = T(n/2) + O(n)$

アウトライン

1. 二分木
2. ソートアルゴリズム
3. 演習

演習問題提出方法

演習問題は，解答プログラムをまとめたテキストファイル (***.txt) を作成して，OCW-i で提出して下さい.

- 提出期限は次回の授業開始時間です.
- どの演習問題のプログラムかわかるように記述してください.
- 出力結果もつけてください.
- 途中まででもできたところまで提出してください.

演習問題

問 1

二分探索木
(`http://yambi.jp/lecture/advanced_programming2016/ex5-1.py`)
を完成させ，正しく動いていることを確認せよ．

問 2

bubble_sort, heap_sort, merge_sort, quick_sort 以外のソート方法を調べ，どのようなアルゴリズムであるか説明し，実装せよ．

問 3

ランダムなリストに対して各ソート関数 (bubble_sort, heap_sort, merge_sort, quick_sort + 問 2 で実装したもの) の実行時間を比較せよ．

問 4 （おまけ）

リストを用いてヒープを実装せよ．