プログラム設計とアルゴリズム
第7回 (11/8)

早稲田大学高等研究所　講師
福永津嵩

(前回の復習) 木とは

図10.14

(前回の復習)二分ヒープ

- データから最大値(最小値)を取得するのに適したデータ構造

- 各頂点xが値key[x]を持つ二分木であり、次の条件を満たす

  1. xの親頂点をpとしたとき、key[p] >= key[x]が成立する。
     (不等号を逆にすると最小値の取得になる)
  2. 木の高さをhとすると、h-1以下の部分は完全二分木である。
  3. 高さhの部分は、頂点が左詰されている。

- 兄弟姉妹の順序などはkey[x]に依存せずどうでも良いことに注意

(前回の復習)二分ヒープの例

図10.19上部

- 定義から明らかに、根が最大値になるので、
  O(1)で最大値を取得することが可能である。
- 要素の検索のような操作には向いていない。

(前回の復習)ヒープソート

- ヒープを利用した次のソートを、ヒープソートと呼ぶ。

  1. 与えられた配列からヒープを構築する。(O(N)の計算量)
  2. 最大値を順に取り出して配列の後ろから詰めていく
     (O(NlogN)の計算量)

- 全体の計算量はO(NlogN)となる。

- O(NlogN)のソートは他にもあり、ヒープソート自体は平均的に遅いため全体をソートしたい時にはあまり使われない。

- ただし、大きい方から上位K個を取り出してソートしたいという時には、O(KlogN)でソートが可能という特徴を持つ。

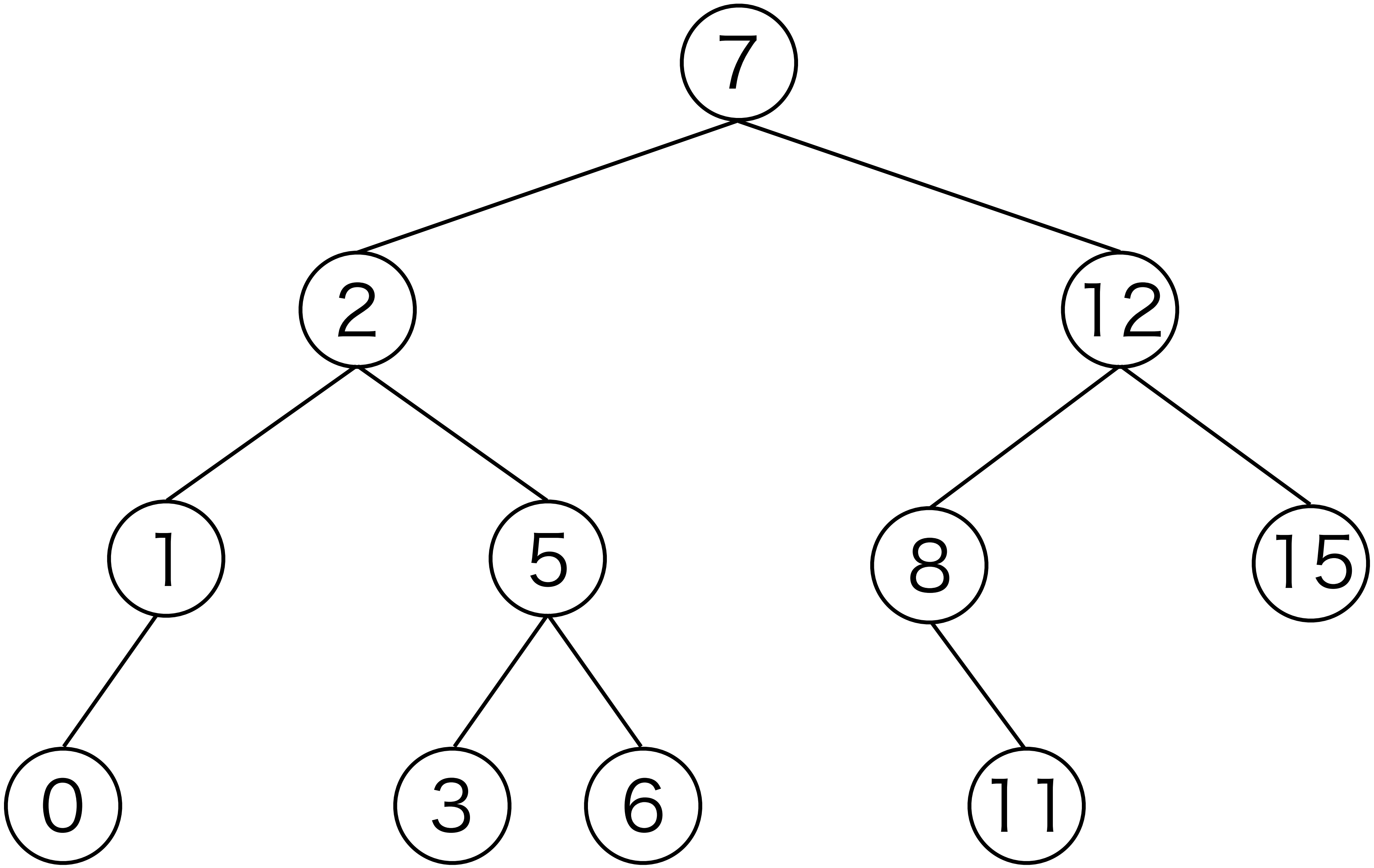
(前回の復習)二分探索木

- ハッシュテーブルや連結リストと同様に、要素の挿入・削除・検索をサポートするデータ構造

- 二分探索木は、各頂点vが値key[v]を持つ二分木であり、次の条件を満たすもののことを言う

二分探索木の条件

(前回の復習) 二分探索木の例



- 二分探索木が平衡である場合には、探索・挿入・削除の操作が O(log N) で可能。

(前回の復習) Union-Find

- グループ分けを管理するデータ構造であり、次の処理を行う事が出来る。

  ・issame(x, y): x, yが同じグループに属するかどうかを調べる
  ・unite(x, y): xが属するグループと、yが属するグループを併合する。

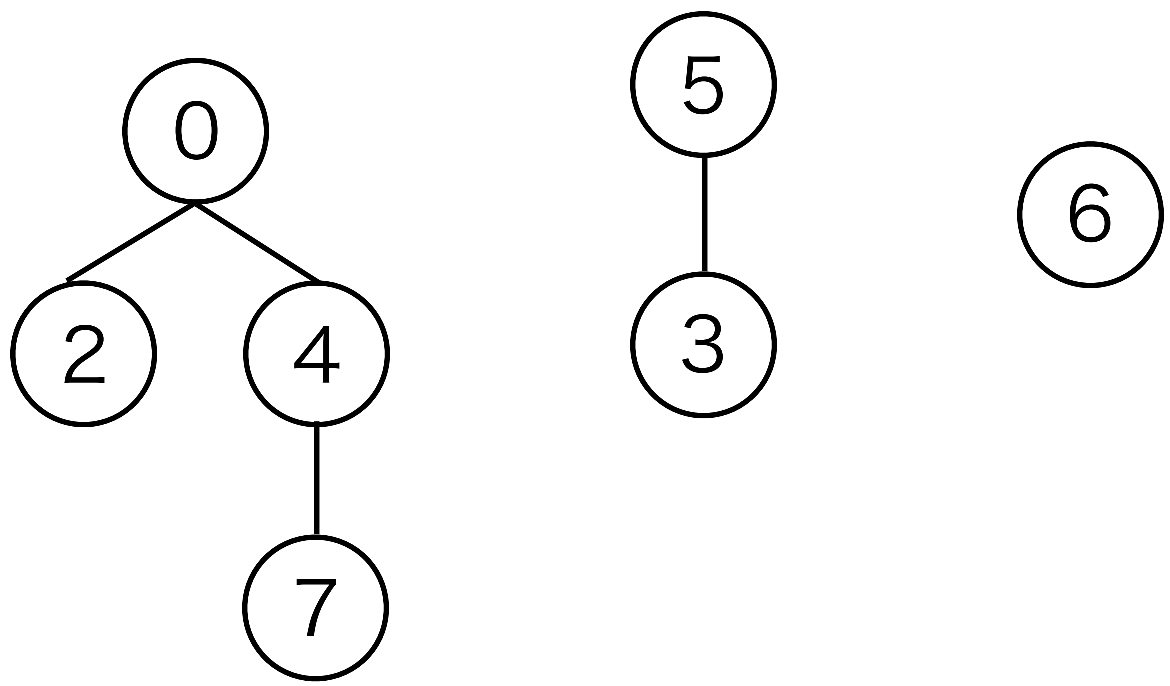
- 右の例に対しては、

  issame(0, 4) = true
  issame(3, 5) = true
  issame(2, 6) = false

図11.1左部

(前回の復習) Union-Find

- Union-Findでは、1つのグループが1つの木で表現される。
  なので、グループの集合は森となる。



- 同一グループに属する様子が1つの木としてまとまっていれば、
  木の形や親子の関係は何でも良い。

第十二章
ソート

ソートとは

- これまでに何度か登場したが、
  与えられたデータを、順序に従って並び替えることをソートという。

- 6, 1, 2, 8, 9 ,2, 5 というデータを小さい順にソートすると、
  1, 2, 2, 5, 6, 8, 9 となる。

- 先ほど紹介したヒープソートはソートアルゴリズムの一例である。

ソートアルゴリズムの良し悪し

- まず、アルゴリズムの実行時間に大きく影響を与えるため、
計算量はとても重要。

- また、アルゴリズムのメモリ使用量も重要。特に、与えられたデータ以外に追加でメモリの使用がほとんど必要ないアルゴリズムを
in-placeであるという。

- 最後に、ソートの安定性が評価されることがある。安定とは、
ソートアルゴリズムを行なった際に、同一の値を持つ要素の間で
順序が入れ替わらないことを意味する。

ソートの安定性が破壊される例

図12.1

ソート(1): ボゴソート

- 1. 与えられた配列がソートされているかどうかをチェックする。
  ソートされていたら終了。ソートされていない場合は2へ。
  2. 配列をシャッフルする。1に戻る。

- どう考えても、非常に効率が悪い。
  最悪計算時間はO(∞)、平均計算時間はO(n・n!)となる。

- もちろん、実用性は全くない。

ソート(2): 挿入ソート

- 左からx枚の要素がソートされている時、x+1枚目の要素を適切な位置に格納する。

図12.2

挿入ソートの性質

- 1個の要素を適切な位置に持っていく計算量はO(N)、それをN個行うので最悪計算量はO(N^2)となる。
- ただし、ほとんどソートされている配列では高速であり、また要素数が非常に少ない(10以下とか)場合にもかなり高速である。
- in-placeなソートであり、また安定なソートである。

ソート(3): マージソート

図12.3上部

ソート(3): マージソート

図12. 3下部

ソート(3): マージソート

- マージソートでは、まず配列を半分ずつに分割していき、要素一つまで分割し切ったら、再帰的にソートを行なって併合していく。

- 講義の第2回で紹介した、分割統治法を活用したソートアルゴリズムである。

マージソートにおける併合

図12.4

マージソートの性質

- マージソートの最悪計算量はO(NlogN)となる(証明は次スライド以降)
- データ以外に外部メモリを必要とし、すなわちin-placeではない。
- また、マージソートは安定ソートであり、C++の標準ライブラリのstable_sort()はマージソートであることが多い。

マージソートの計算量

- マージソートの計算量をT(N)とすると、

  $T(1) = O(1)$
  $T(N) = 2T(N/2) + O(N)$

  という漸化式で書くことができる。

- より一般に、

  $T(1) = c$
  $T(N) = aT(N/b) + dN$

  という漸化式で書ける時の計算量を考える。(マージソートは$a = b = 2$)

マージソートの計算量

- 簡単のため$N = b^k$とする。

$$
\begin{aligned}
&T(N)\\
&= aT\left(\frac{N}{b}\right) + dN\\
&= a\left(aT\left(\frac{N}{b^2}\right) + d\frac{N}{b}\right) + dN\\
&= \dots\\
&= a\left(a\left(\dots a\left(aT\left(\frac{N}{b^k}\right) + d\frac{N}{b^{k-1}}\right) + d\frac{N}{b^{k-2}} + \dots\right) + d\frac{N}{b}\right) + dN\\
&= ca^k + dN\left(1 + \frac{a}{b} + \left(\frac{a}{b}\right)^2 + \dots + \left(\frac{a}{b}\right)^{k-1}\right)\\
&= cN^{\log_b a} + dN\left(1 + \frac{a}{b} + \left(\frac{a}{b}\right)^2 + \dots + \left(\frac{a}{b}\right)^{k-1}\right)
\end{aligned}
$$

マージソートの計算量

$$cN^{\log_b a} + dN\left(1 + \frac{a}{b} + \left(\frac{a}{b}\right)^2 + \cdots + \left(\frac{a}{b}\right)^{k-1}\right)$$

- この式変形より、
  $a < b \rightarrow T(N) = O(N)$
  $a = b \rightarrow T(N) = O(N\log N)$
  $a > b \rightarrow T(N) = O(N^{\log_b a})$

- よって、マージソートの計算量はO(NlogN)となる。

ソート(4): クイックソート

図12.6

クイックソートの性質

- クイックソートの最悪計算量は$O(N^2)$となる。
  これは、要素m個の配列を分割する時に、pivotとして最も小さい値が選ばれたなら、分割が1:m-1になるためである。
  ただし平均的には$O(N \log N)$であり実用上は最も高速である。

- (クイックソートは一見in-placeアルゴリズムのように見えるが、
  関数再帰呼び出しに必要なスタック領域を考慮する必要がある。
  呼び出しごとにスタック領域が必要なため、最悪$O(N)$の追加領域が必要になるが、実装上の工夫により$O(\log N)$となる。
  これをin-placeと呼ぶかどうかは定義による。)

- pivotの選び方で順番が変わり得るので、安定ソートではない。

イントロソート(Introspective sort)

- クイックソート、ヒープソート、挿入ソートを組み合わせたハイブリッドなソートアルゴリズム。
- 基本的にはクイックソートでソートされ、再帰の数が深くなりすぎている時にはヒープソートに切り替える。また、要素数が少なくなった場合には挿入ソートに切り替える、とするアルゴリズム。
- イントロソートの最悪計算量はO(NlogN)であり、C++のsort()ではイントロソートが利用されている事も良くある。
- なおPythonではマージソートと挿入ソート(+実装上の様々な工夫)のハイブリッド法であるティムソートが採用されている。

乱択クイックソート

- クイックソートのpivotの選び方に乱数を考慮することで、
偏りのあるデータのソートに対しても高速にソートできる手法。
乱択アルゴリズムの一種

- leftとrightの中点をpivotとしていた部分を、
leftからrightの中で1点ランダムに選びそれをpivotにするよう変更する

- その平均的な計算量はO(NlogN)となる(次スライド以降で証明)

乱択クイックソートの計算量

- ソートの計算量は、要素の比較を行う回数である。
- ある乱択クイックソートが行われた時、
  配列のi番目に小さい要素とj番目に小さい要素の比較が行われた時には1を取り、行われなかった時には0を取る確率変数X_{ij}を考える。
- よって、その平均計算量は

$$E[\sum_{0 \le i < j \le N-1} X_{ij}] = \sum_{0 \le i < j \le N-1} E[X_{ij}]$$

乱択クイックソートの計算量

- クイックソートでは、pivotに選ばれない限り比較は行われない。

- i番目とj番目が比較が行われなかったとは、i番目かj番目がpivotに選ばれる前に、i+1~j-1番目のいずれかがpivotに選ばれてしまったことを意味する。

- 逆に、比較が行われたとは、i+1~j-1番目がpivotに選ばれる前に、i番目かj番目がpivotに選ばれてしまったことを意味する。

- よって$E[X_{ij}]$は、i~j番目のうちiかjが先に選ばれる確率となるので

$$E[X_{ij}] = \frac{2}{j-i+1}$$

乱択クイックソートの計算量

$$
\begin{aligned}
E[\sum_{0\leq i<j\leq N-1} X_{ij}] &= \sum_{0\leq i<j\leq N-1} \frac{2}{j-i+1} \\
&< \sum_{0\leq i\leq N-1, 0\leq j-i\leq N-1} \frac{2}{j-i+1} \\
&= \sum_{0\leq i\leq N-1} \sum_{0\leq k\leq N-1} \frac{2}{k+1} \\
&= 2N \sum_{1\leq k\leq N} \frac{1}{k} \\
&= O(N\log N)
\end{aligned}
$$

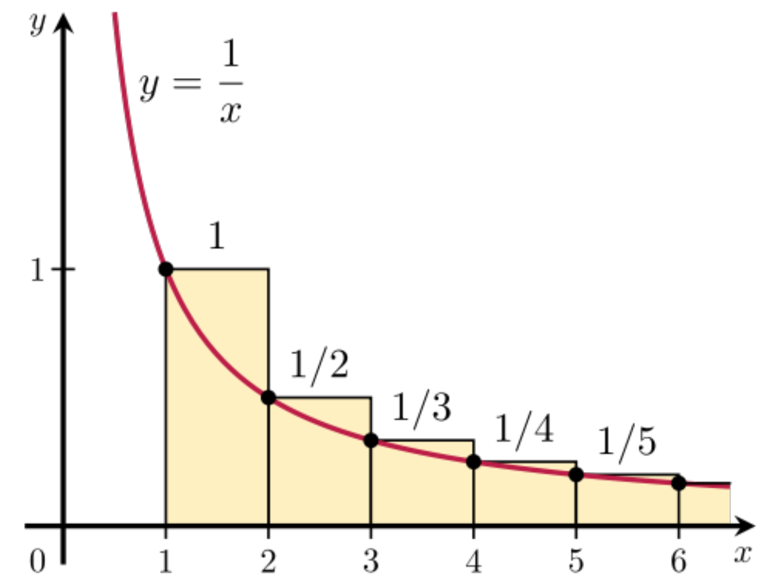
乱択クイックソートの計算量

- $1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{N}=O(\log N)$ の証明について

- 積分で上下から挟むのが良くある証明

- 図より、色付きの部分の面積は

$$\sum_{k=1}^{N}\frac{1}{k} > \int_{1}^{N+1}\frac{1}{x}dx = log(N+1)$$



Wikipedia「調和級数」より

- また、

$$1+\sum_{k=2}^{N}\frac{1}{k} < 1+\int_{1}^{N}\frac{1}{x}dx = 1+log(N)$$

ソートの計算量の下界

- これまで紹介したソートアルゴリズムは(ボゴソートを除き)
要素の比較によってソートを行うアルゴリズムであった。

- このような比較ソートでは、計算量の下界がΩ(NlogN)であることが
知られている。

- N個の要素があってそれを並び替えるとすると、並び替え方は
全部でN!通り存在する。このうちどれかがソートの解として正しい。

- 大小比較をh回行うと、最大で2^h種類の異なる解を識別することが
出来る。逆に言えば、$2^h < N!$ であれば、その比較回数では識別出来ない
並び替えが存在する。

ソートの計算量の下界

図12.8

- この事を二分木上で表現したのが上図である。
- すなわち、

$$h \geq \log N! = \sum_{k=1}^{N} \log k > \int_{1}^{N} \log x dx > N \log N$$

- よって、比較ソートアルゴリズムの計算量の下界はΩ(NlogN)である。

ソート(5):バケットソート

- 要素の比較によらないソート法であれば、O(NlogN)を下回る計算量でソートを行うことが可能であり、その一つがバケットソートである。

- バケットソートでは、ソートしたい配列aの各要素が、0以上A未満の整数値であるという仮定を用いる。(そのため、実数値や文字列のソートを行うことは難しい。)

- バケットソートでは、配列numを用意する。num[i]は配列a中に含まれる値iの要素数を意味する。

- この時、バケットソートはO(N+A)でソートを行うことが可能である。

ソート(5):バケットソート

- 例)
a = {0, 2, 0, 1, 1, 1, 2, 0, 2, 0, 1}
であるとして、全ての要素が3未満であることがわかっているとする。

- 配列numに、各要素の出現回数を記録する。この操作はO(N)
すなわち、num[0] = 4, num[1] = 4, num[2] = 3 となる。
num[i]のことをバケット(バケツ)と呼ぶことがある。

- num[i]に入っている数分iを小さい方から順番に並べていく。よって、
まず0を4つ並べ、次に1を4つ並べ、最後に2を3つ並べればよい。
この操作はO(A)
結果、a = {0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2}となる。

ソート(6):基数ソート

- バケットソートはAが非常に大きい時には現実的ではない。
  そこで、低い桁から順番に、桁ごとに区切ってバケットソートを行うことで、効率的にソートを行う手法が提案されている。
  (ただし、バケットソートを安定ソートとして実装しなければならない)

  例)

373		251		443		171
663		171		251		251
251		373		363		273
273	一桁目で	663	二桁目で	171	三桁目で	363
171	ソート	273	ソート	373	ソート	373
443		443		273		443

ソート(6):基数ソート

- 基数ソートは、文字列の辞書順へのソートにも応用可能である。
(英単語の場合、26進数とみなせる)

- A進数でL桁の要素を基数ソートで並び替える際には、
バケットをA個用意したバケットソートをL回行うことに
なるので、その計算量はO(L(N+A))となる。

まとめ

表12.1