
続・プログラミングゼミ
(第6回スタートアップゼミ？)
2012年9月11日(火)
M1 伊藤 創太

データ構造とアルゴリズム

情報科学？

駒場教養の「情報科学」(理系準必修？)のカリキュラム

- 数の計算と関数
- 条件分岐と繰り返し
- 関数から計算へ
- アルゴリズムと計算量
- 数値計算
- パターン認識
- レコードとオブジェクト
- 再帰データ構造
- いろいろなプログラミング言語

データ構造とアルゴリズム

プログラムの実装

データ構造

データをどのようにコンピュータのメモリに並べるかという方式・形式

アルゴリズム

データを操作する手続き

アルゴリズムの理解

→プログラミング

適切なアルゴリズム・データ構造の選択

→計算の高速化・汎用化

探索

探索：配列データからの検索

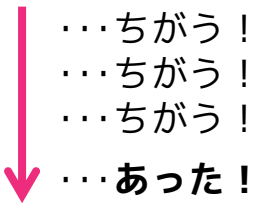
(例) 郵便番号リスト（ソート済み）からの住所の検索

郵便番号1130033の住所は？

線形探索

配列の順番に見つかるまで調べていく

0010000	北海道札幌市北区
:	:
1130032	東京都文京区弥生
1130033	東京都文京区本郷
1130034	東京都文京区湯島
:	:
9998531	山形県飽海郡遊佐町菅里



・・・ちがう！
・・・ちがう！
・・・ちがう！
・・・**あった！**

知りたい答えが見つかったら、探索終了。

線形探索　コード例：

```
import java.io.*;                                   //java.ioパッケージを使う

public class Main {
  public static void main(String[] args) {
    try {                                           //データをうまく読み込めるとき
      String inputfile = "./input/input-sep1.csv";     //インプットファイル名
      int dataN = 123325;                           //データ数は123325（今回は固定）
      BufferedReader br = new BufferedReader(new FileReader(inputfile));
      String line = null;                           //1行ごとに読み込む変数を用意
      String[][] yubin = new String[dataN][2];      //郵便番号表の格納配列
      int i = 0;                                    //郵便番号データ格納に必要なポインタ
      while ((line = br.readLine()) != null) {      //最終行になるまで読み込む
        String[] splitline = line.split(",");       //カンマ区切りで分割
        yubin[i][0] = splitline[0];                 //1列目に郵便番号
        yubin[i][1] = splitline[1];                 //2列目に住所
        i++;
      }
      br.close();

      System.out.println( addressmatch("1130033", yubin) );
```

郵便番号データの格納

探索実行結果表示

parseInt

探索

線形探索　コード例：

```
    }
    catch( IOException e ) {                    //データをうまく読み込めないとき
      System.out.println("データ入力失敗");
    }
  }

  static String addressmatch(String num, String[][] yubin) {
    for (int i = 0; i < yubin.length; i++ ){           //順番に検索対象と一致するか調べる
      if(Integer.parseInt(yubin[i][0]) == Integer.parseInt(num)){
        return yubin[i][1];                            //一致したら住所を返す
      }
    }
    return "なし";                              //一致するものがない場合は「なし」を返す
  }
}
```

例外処理

線形探索アルゴリズム

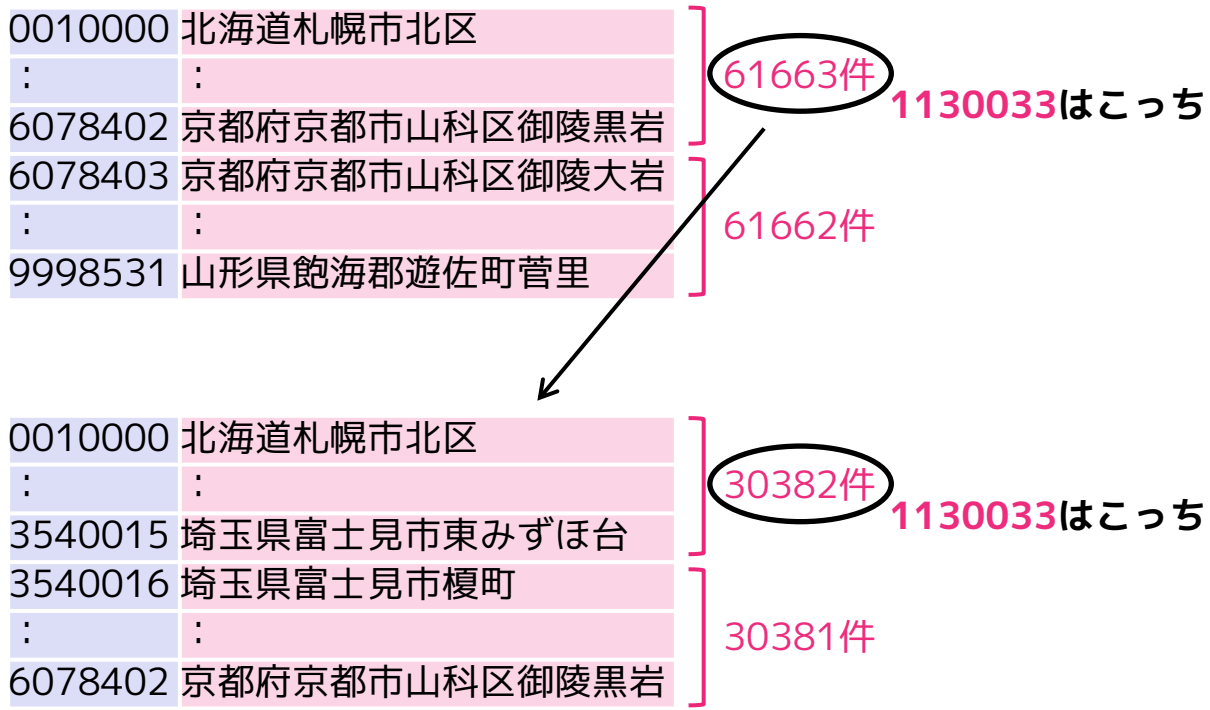
実行結果

```
東京都文京区本郷
```

探索

二分探索

前半と後半にわけて絞っていき、検索対象を見つけ出す



1つに絞られるまで繰り返す。

データは昇順にソートされている必要がある。

課題1

二分探索アルゴリズムの実装

・二分探索法で郵便番号リストから住所を検索する

・郵便番号がString型になっていることに注意（0から始める番号がある都合上。purseIntやvalueOfでint型に直せる）

（ヒント）

・探索範囲の上限と下限の変数を導入することとして、upper=yubin.length、lower=0のようにして始めて、upperとlowerが一致したらそれが答え・・・みたいな感じ。

```
static String addressmatch2(String num, String[][] yubin) {

        ？？？？？

}
```

計算量

2つの探索方法の比較

計算時間（同一の1000件の検索を行った所要時間）

線形探索・・・4.674秒

二分探索・・・0.087秒

→二分探索の方が速い

アルゴリズムの計算量をどう評価するか

計算時間はコンピュータやコンパイラの性能によって違う

使うデータによっても計算時間は違う

→計算の繰り返し処理の回数の、最大回数で比較

→データ数による繰り返し回数の増加でだいたい計算量の差がわかる

O表記法

計算量の評価 – O表記法

あるアルゴリズムについてn個のデータに対して必要な
計算量のオーダーを示す。

線形探索の場合・・・

最悪の場合、1番目からn番目まで全て比較する → **O(n)**

（最良の場合は1回で済むが、最大計算量で比較する）

二分探索の場合・・・

比較を行う回数は、n個のデータのとき$\log_2 n$回 → **$O(\log_2 n)$**

O表記法

計算量の評価 – O表記法

データ数nが増えると、オーダーの違いによる差は大きくなる

n:	2	4	8	16	100	1万	100万	1億
1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
$\log_2 n$	0.7	1.4	2.1	2.8	4.6	9.2	13.8	18.4
logn	1.0	2.0	3.0	4.0	6.6	13.3	19.9	26.6
n	2.0	4.0	8.0	16.0	100.0	10000.0	1.0×10^{6}	1.0×10^{8}
nlogn	1.4	5.6	16.6	44.4	460.5	92103.4	1.4×10^{7}	1.8×10^{10}
n^2	4.0	16.0	64.0	256.0	10000.0	1.0×10^{8}	1.0×10^{12}	1.0×10^{16}
n!	2.0	24.0	40320.0	2.0×10^{13}	#	#	#	#

データ構造

おもなデータ構造

・配列　　・・・int[]など

・ハッシュテーブル　　・・・Hashmap、Hashtableクラス
（前回プログラミングゼミ参照）

・連結リスト

・二分木

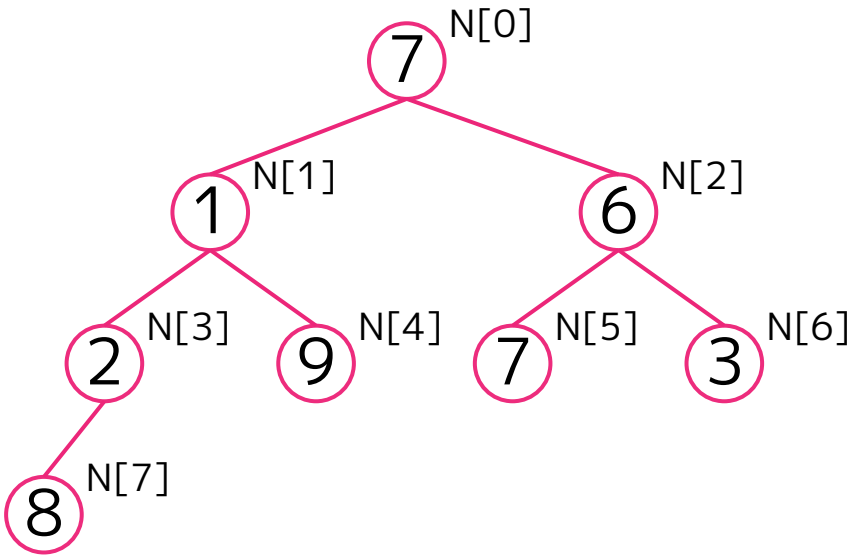
・グラフ

データ構造により計算量が大きく変わることも多い

二分木

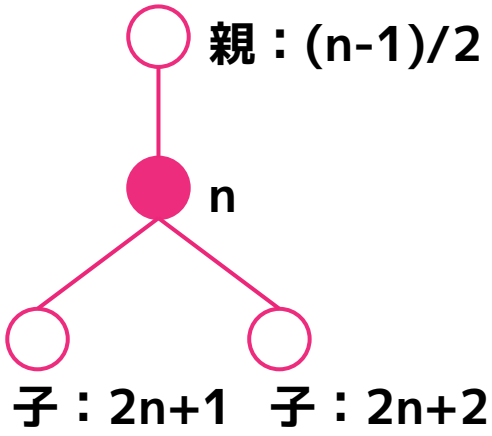
配列で表す二分木

木の構造と配列の要素を
対応させてデータを格納する



何が嬉しいか

親子関係を配列で表せる
どの部分でも配列添え字の関係が同じ

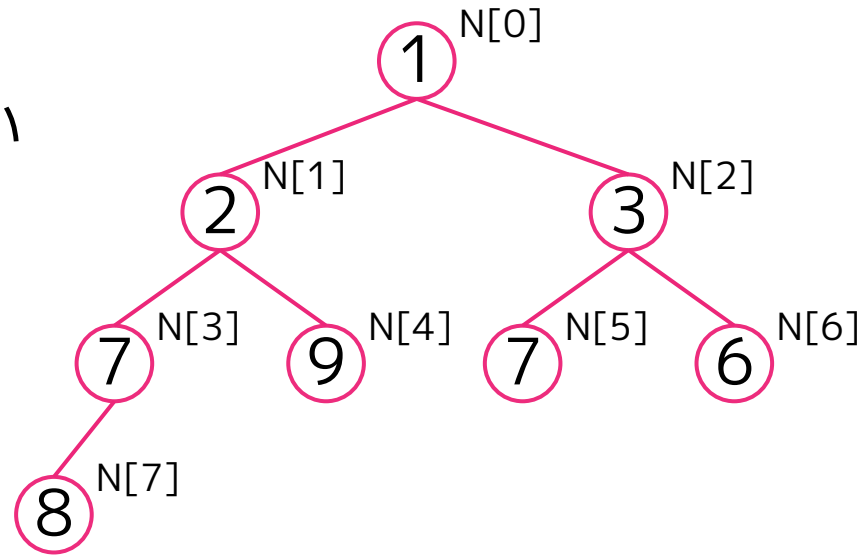


ヒープ木

ヒープ構造

親の要素は子の要素より小さい

木の根の要素は最小の要素



何が嬉しいか

常に根がその木の中の

最小値を格納している

→最小値取り出しの計算量は**O(1)**

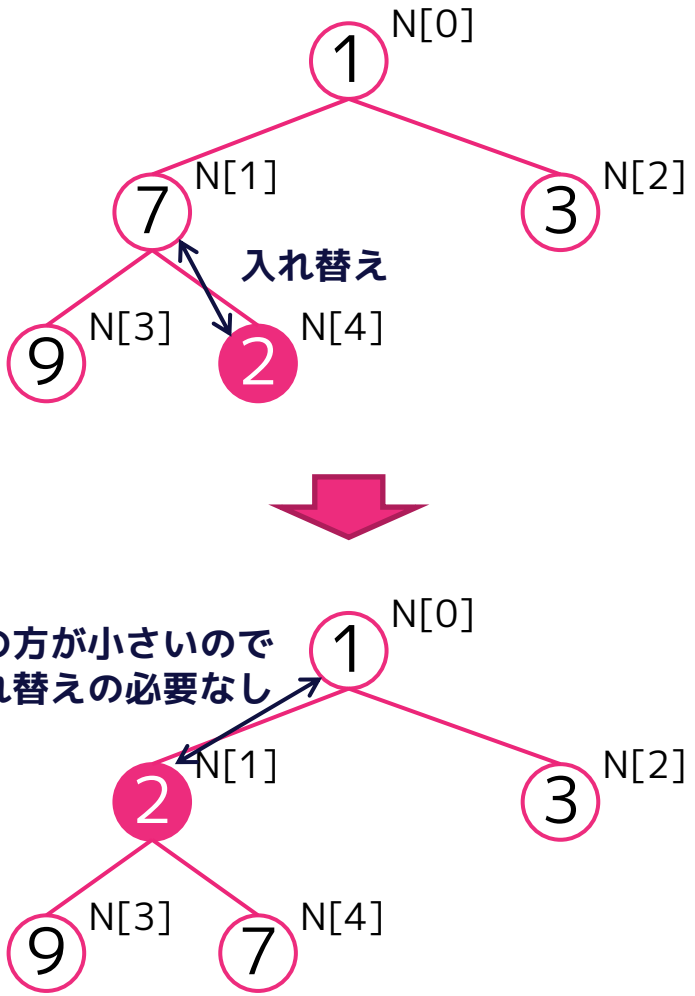
ヒープ木

要素の挿入

木の最後（配列の最後に追加）

親と比較して親が大きければ

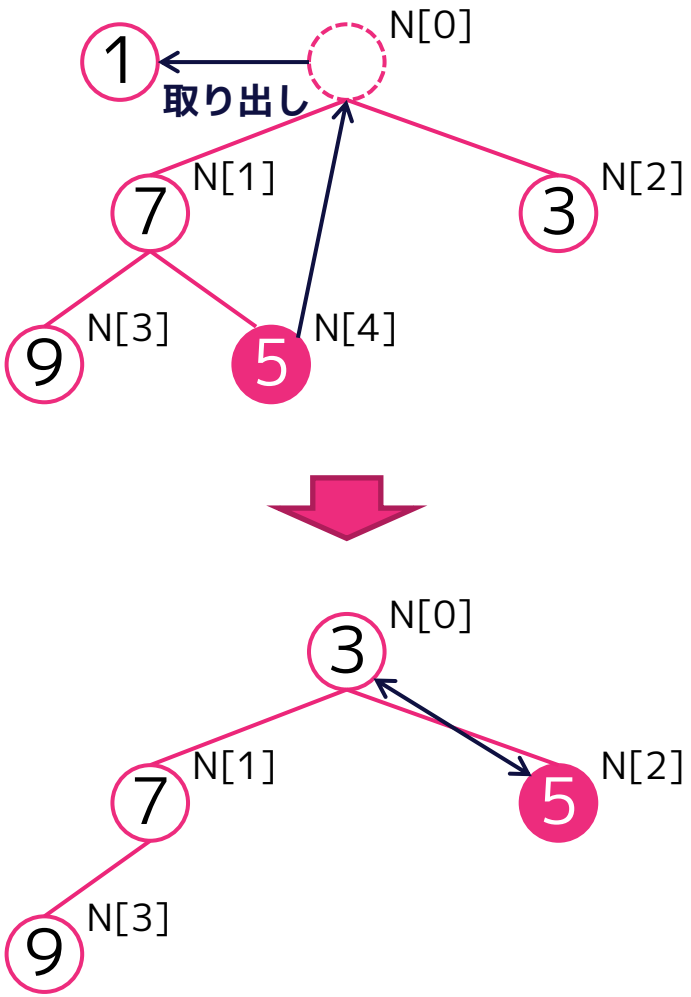
入れ替え



ヒープ木

要素の取り出し

最後の要素を先頭に入れ、
子の方が小さければ入れ替え



ヒープ木

javaでの実装例

```
import java.io.*;                                   //java.ioパッケージを使う
import java.util.ArrayList;                         //ArrayListを使う

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> heapnode = new ArrayList<Integer>();
    insert(3,heapnode);                             //ここでは3,2,1を挿入してみている
    insert(2,heapnode);
    insert(1,heapnode);
    System.out.println(draw(heapnode));             //根の要素を取り出して結果を表示
  }

  static void insert(int x, ArrayList<Integer> heap){  //新しい要素の挿入の関数
    heap.add(x);                                        //配列の最後に入れる
    int n = heap.size() - 1;
    while(heap.get((n-1) / 2) > heap.get(n)){           //親ノード((n-1)/2)との比較
      int a = heap.get((n-1) / 2);                      //要素の入れ替え
      heap.set((n-1) / 2, heap.get(n));
      heap.set(n, a);
      n = (n-1) / 2;
    }
  }
```

ヒープ木

javaでの実装例

```
  static int draw(ArrayList<Integer> heap){            //根の要素の取り出しの関数
    int minimum = heap.get(0);                         //いったん根の要素を保管
    heap.set(0, heap.get(heap.size() - 1));            //最後尾の要素を根に持ってくる
    heap.remove(heap.size() - 1);                      //最後尾要素を消去
    int n = 0;
    while(2*n+1 < heap.size() && heap.get(n) > heap.get(2*n+1)){ //子(左側)との比較
      int a = heap.get(n);
      heap.set(n, heap.get(2*n+1));
      heap.set(2*n+1, a);
      n = 2*n+1;
    }
    while(2*n+2 < heap.size() && heap.get(n) > heap.get(2*n+2)){ //子(右側)との比較
      int a = heap.get(n);
      heap.set(n, heap.get(2*n+2));
      heap.set(2*n+2, a);
      n = 2*n+2;
    }
    return minimum;
  }
}
```

ダイクストラ法

javaでの実装例

```
  static int draw(ArrayList<Integer> heap){            //根の要素の取り出しの関数
    int minimum = heap.get(0);                         //いったん根の要素を保管
    heap.set(0, heap.get(heap.size() - 1));            //最後尾の要素を根に持ってくる
    heap.remove(heap.size() - 1);                      //最後尾要素を消去
    int n = 0;
    while(2*n+1 < heap.size() && heap.get(n) > heap.get(2*n+1)){ //子(左側)との比較
      int a = heap.get(n);
      heap.set(n, heap.get(2*n+1));
      heap.set(2*n+1, a);
      n = 2*n+1;
    }
    while(2*n+2 < heap.size() && heap.get(n) > heap.get(2*n+2)){ //子(右側)との比較
      int a = heap.get(n);
      heap.set(n, heap.get(2*n+2));
      heap.set(2*n+2, a);
      n = 2*n+2;
    }
    return minimum;
  }
}
```

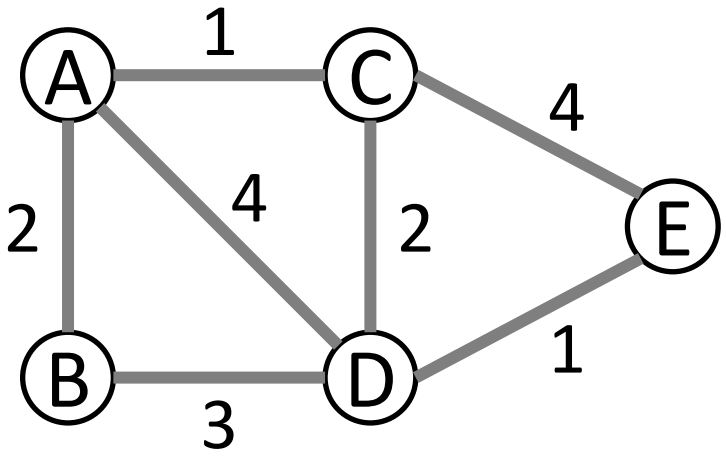
○ダイクストラ法

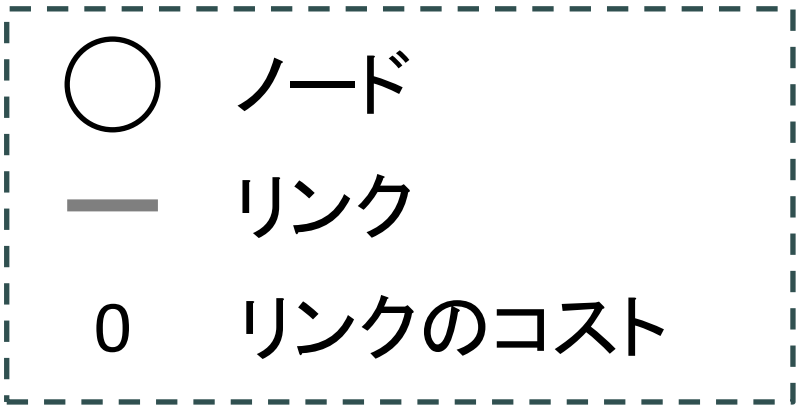
▼ダイクストラ法

グラフの最短距離・経路を求めるアルゴリズム

ダイクストラ（蘭）が1958年に考案

（例） AからEまで行く最短距離は？
そしてその経路は？





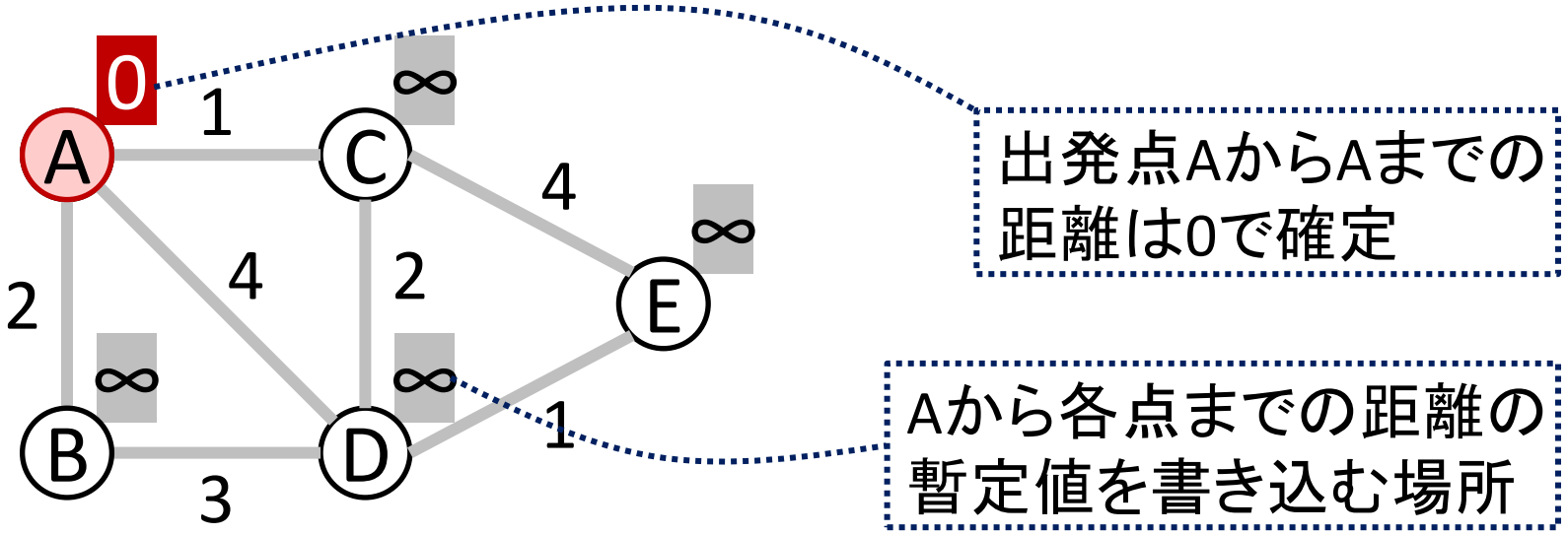
○ダイクストラ法

▼ダイクストラ法の考え方

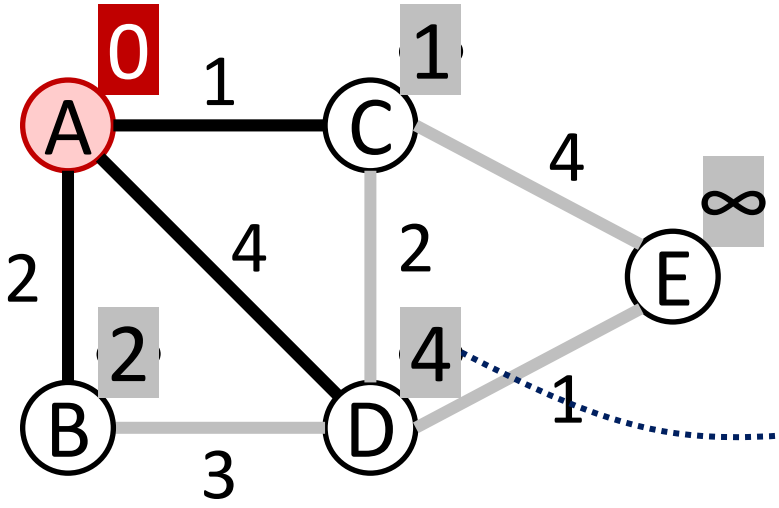
＜わかった最短経路から確定させていく＞

【Step1】 出発点から各点の距離の暫定値を設定

（赤色：確定済のノード・値、灰色：未確定）



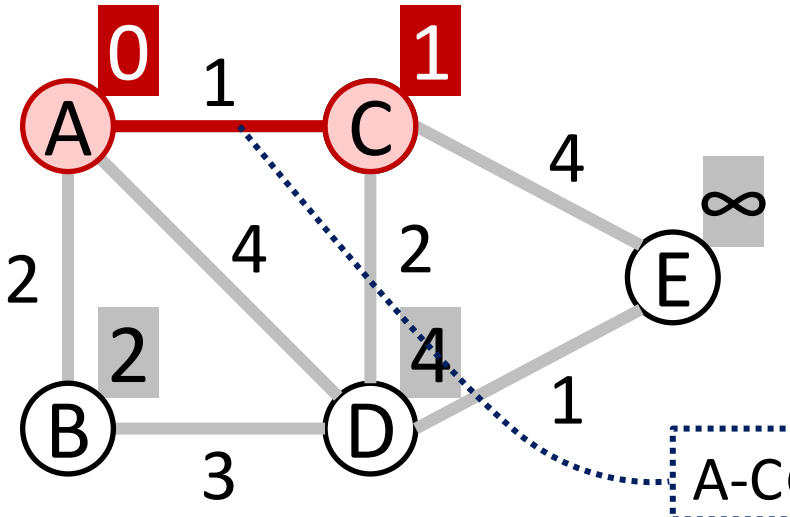
○ダイクストラ法



【Step2】

確定済の点から経路をたどり暫定値を更新

例えば現時点でのAからDの最短距離は4

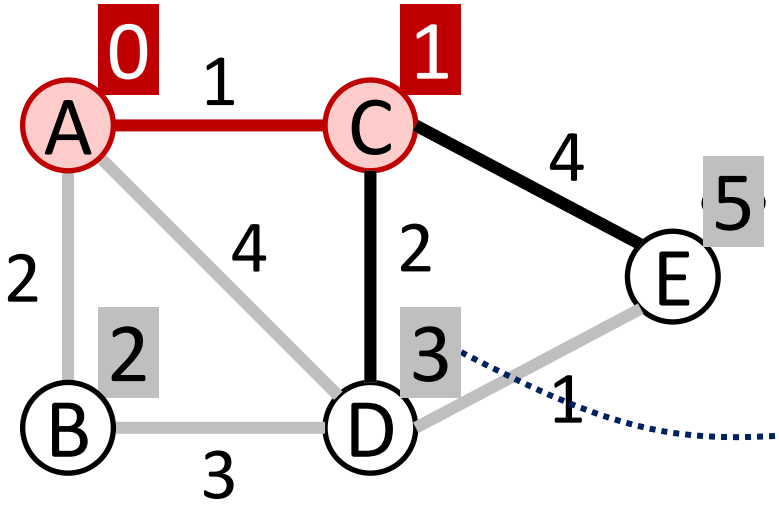


【Step3】

未確定の中で最小距離の点を確定させる

A-Cの最短経路も確定

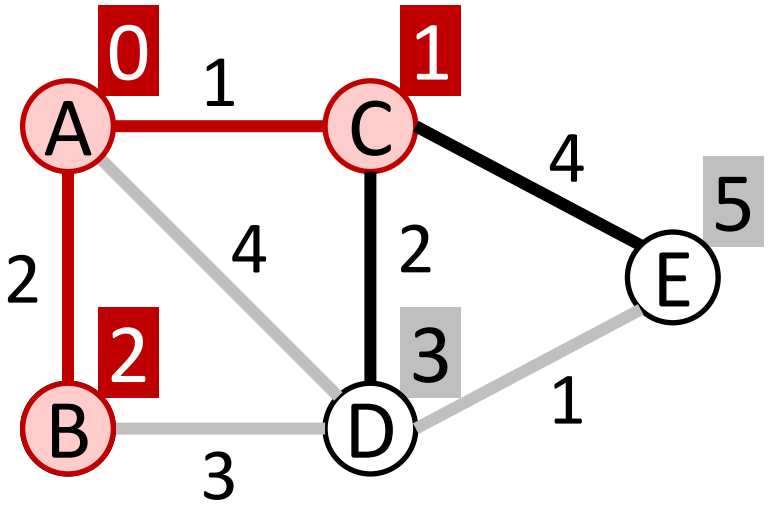
○ダイクストラ法



【Step4】

確定済の点から経路をたどり暫定値を更新

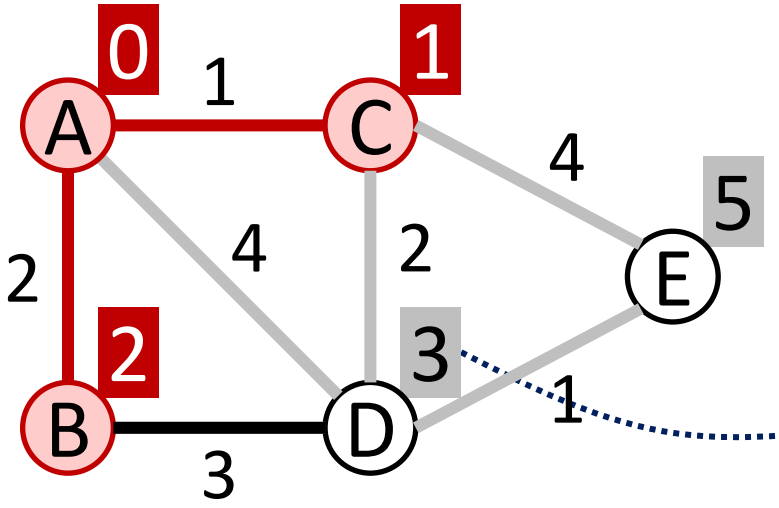
A-Dの4よりA-C-Dの3の方が小さいため、3に更新



【Step5】

未確定の中で最小の点を確定させる

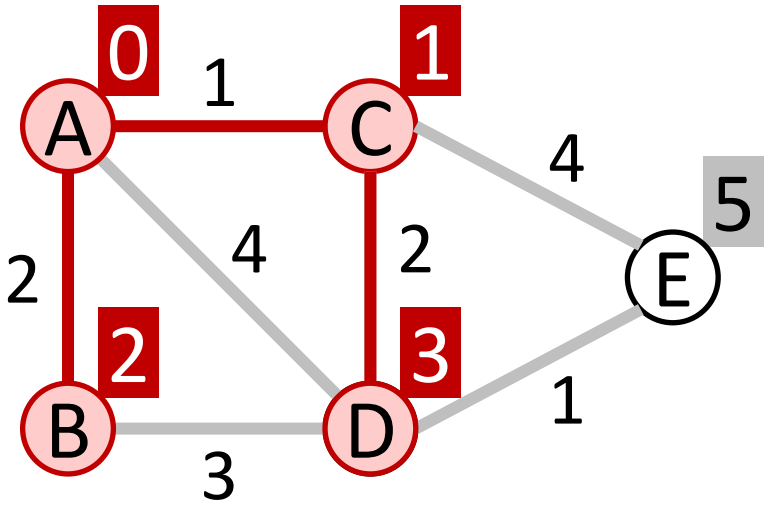
○ダイクストラ法



【Step6】

確定済の点から経路をたどり暫定値を更新

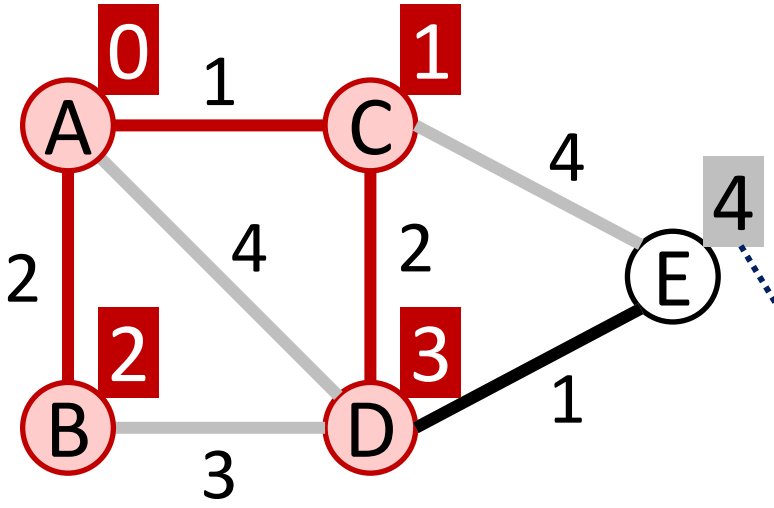
A-C-Dの3よりA-B-Dの5の方が大きいため更新されず



【Step7】

未確定の中で最小の点を確定させる

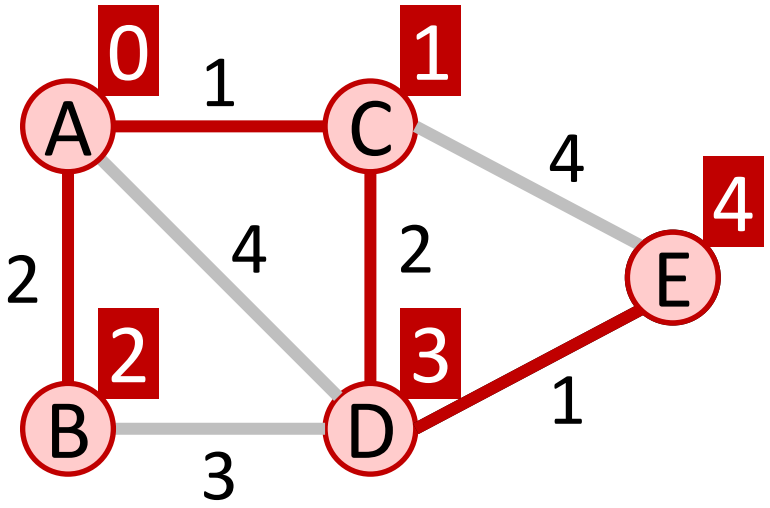
○ダイクストラ法



【Step8】

確定済の点から経路をたどり暫定値を更新

A-C-Eの5よりA-C-D-Eの4の方が小さいため更新



【Step9】

未確定の中で最小の点を確定させる

完成

AからEの最短距離は4で経路はA-C-D-E

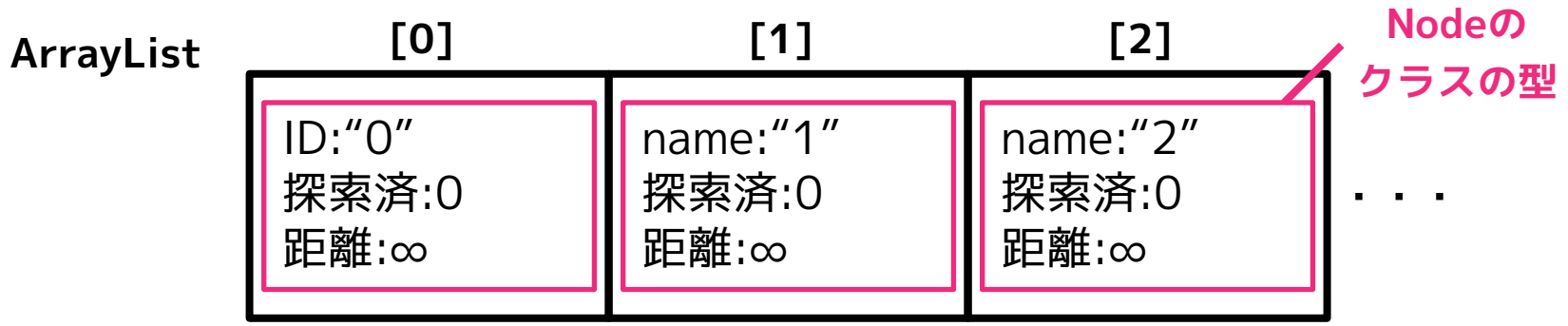
課題2

ダイクストラ法アルゴリズムの実装

・ダイクストラ法をjavaで実装

・まずはやってみる、可能であればデータの構造やアルゴリズムの工夫で高速化

基本的には、nodeとlinkについてそれぞれクラスの型を作って配列で管理、逐次探索する

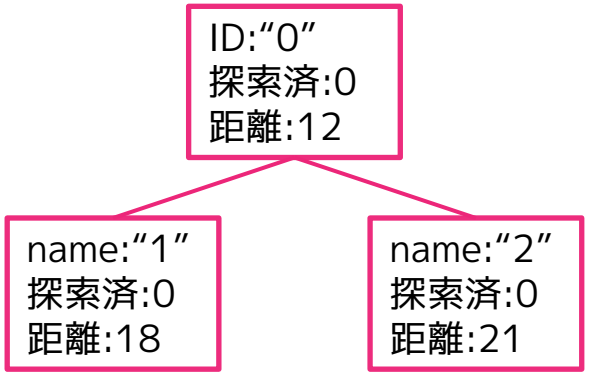


課題2

ダイクストラ法アルゴリズムの実装

高速化のヒントとして・・・

・次に探索する可能性のあるノードをヒープ木にするとか



・Nodeのデータの中に次に接続可能なNodeを格納して、リンクの長さは行列で管理するとか・・・