データ構造とアルゴリズム　第１回

- 担当：増澤 利光　masuzawa@ist.osaka-u.ac.jp
- ＴＡ：原 悠樹（M2）hara.yuki@ist.osaka-u.ac.jp

  - 出席レポートチェック，ミニレポート採点，質問対応
- オフィスアワー：月曜16:10～17:10（G棟４階・教員控室Ｉ）
- 成績：試験（中間／期末）８割，
  ミニレポート １割，出席レポート １割
- ミニレポート：CLEの「コンテンツ／ミニレポート」
  - 課題提示，レポート提出，質問
  - 締切は次回の講義の前日
- 講義資料等の配布：CLEの「コンテンツ／講義資料」
- e-learning：CLEの「コンテンツ／講義映像」
  - e-learning で講義を実施することがあります
  - 好きな時間に学習できる．ミニレポート回答が必要（期限あり）

講義内容

- 内容：データ構造とアルゴリズムの基礎
- テキスト
  - 浅野，和田，増澤「アルゴリズム論 」オーム社(2003)
    - ８章まで（９章は「情報解析Ｂ」で学習）

◆ 効率よく問題を解く（計算する）手法を学ぶ
  - データ構造：計算機内でのデータの表現方法
  - アルゴリズム：データの処理方法（手順）
    - プログラミングＡ／Ｂと重複あり（二分探索，ソートなど）

◆ 何の役に立つのか
  - ソフトウェア開発
  - 問題の難しさの理解：この時間で解ける／解けない

データ構造とアルゴリズム　第１回

1. アルゴリズムの重要性
2. 探索問題
3. 基本的なデータ構造
4. 動的探索問題とデータ構造
5. データの整列
6. グラフのアルゴリズム
7. 文字列のアルゴリズム
8. アルゴリズム設計手法

第１章　アルゴリズムの重要性

1.1 アルゴリズムとは

1.2 アルゴリズムの記述

1.3 アルゴリズムの効率

1.4 アルゴリズムの最適性

今日の学習目標

- アルゴリズムとは何か説明できる
- アルゴリズムの重要性を説明できる
- アルゴリズムの記述を理解でき，アルゴリズムを記述できる
- 漸近的計算量とその重要性を説明できる
  - O記法，Ω記法
- アルゴリズムの最適性とは何か説明できる

1.1　アルゴリズムとは

- アルゴリズム（algorithm，算法）
  - 計算機で問題を解くための手順
  - 曖昧性があってはいけない
  - 正しくなければならない
  - プログラム ＝ アルゴリズムの計算機向け表現

アルゴリズムの計算量

- 優れたアルゴリズムとは
  - 理解しやすい
    - プログラム化しやすい
  - 計算時間が短い　　→ 時間計算量
  - 使用メモリが小さい → 領域計算量（空間計算量）
  - ・・・

アルゴリズムの計算量

- 計算量：問題のサイズの関数
  - 同じサイズの問題でも，入力によって異なる

    例：ソーティング（データを大きさで並び替える）

    挿入法（アルゴリズム）

    　入力データがソート済みに近いほど高速
- 最大時間計算量：最悪の場合の時間計算量
  - 実時間処理では重要（処理の遅れが許されない）
  - 評価は比較的容易
- 平均時間計算量：平均の場合の時間計算量
  - バッチ処理では重要
  - 評価は一般に難しい
  - 確率分布の仮定の妥当性？

第１章　アルゴリズムの重要性

1.1 アルゴリズムとは

1.2 アルゴリズムの記述

1.3 アルゴリズムの効率

1.4 アルゴリズムの最適性

１.２　アルゴリズムの記述

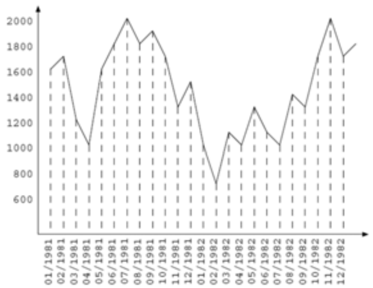
- テキストではＣ風の書き方
  - 分かりやすさのために，マクロ的な書き方もする
    例：株式投資における最大売却益

```
 入力：毎月の株価sp[0], ..., sp[n-1]
mxp = 0; // 利益の最大値 mxp を 0 に初期化
for i=0 to n-2
    for j=i+1 to n-1 {
        d = sp[j]-sp[i]; //売却益 = 売値 - 買値
        if d > mxp then mxp = d;
         // 今まで以上の売却益であればmxpの値を更新する
    }
mxpを答として返す;
```


最大売却益問題

問題：最大売却益問題

- 入力：$sp[0], sp[1], \ldots, sp[n-1]$（$SP[i]$：年月 i の株価）
- 出力：売却益が最大となる買う年月と売る年月を求める
  - 売却益：$sp[j] - sp[i]$
    年月 i に買って，年月 j に売る場合（$i < j$）



最大売却益アルゴリズム１.１

- アルゴリズム１.１：（Ａ）方式

```
mxp = 0; // 利益の最大値 mxp を 0 に初期化
for (i=0 to n-2)
    for (j=i+1 to n-1) {
        d = sp[j]-sp[i]; //売却益 = 売値 - 買値
        if (d > mxp) mxp = d;
         // 今まで以上の売却益であればmxpの値を更新する
    }
mxpを答として返す;
```

全 $i, j\ (0 \le i < j \le n-1)$ の組をチェック

計算時間：減算　$n(n-1)/2$ 回

比較　$n(n-1)/2$ 回

最大売却益アルゴリズム１.２

● アルゴリズム１.２：（Ａ）方式

```
mxp = 0; // 利益の最大値 mxp を 0 に初期化
for (i=0 to n-2) {
    mxsp = sp[i]; //株価の最高値 mxsp を sp[i] に初期化
    for (j=i+1 to n-1)
        if (sp[j] > mxsp) mxsp = sp[j];
    d = mxsp - sp[i]; //売却益=買値とそれ以降の最高値との差
    if (d > mxp) mxp = d;
    // 今まで以上の売却益であればmxpの値を更新する
    }
mxpを答として返す;
```

年月 i 以後の最高値を求める

計算時間：減算　$n-1$ 回（$n(n-1)/2$ から改善）

比較　$n(n-1)/2+n-1$ 回

最大売却益アルゴリズム１.３

● アルゴリズム１.３：（Ｂ）方式

```
mxp = 0; // 利益の最大値 mxp を 0 に初期化
for (j=1 to n-1) {
    mnsp = sp[j]; // 株価の最安値 mnsp を sp[j] に初期化
    for (i=0 to j-1)
        if (sp[i] < mnsp) mnsp = sp[i];
    d = sp[j] - mnsp; //売却益=売値とそれ以前の最安値との差
    if (d > mxp) mxp = d;
    // 今まで以上の売却益であればmxpの値を更新する
}
mxpを答として返す;
```

年月 j 以前の最安値を求める

計算時間：減算　$n-1$ 回

比較　$n(n-1)/2+n-1$ 回

実はバグが．．．

● 株価が下がり続けると売却益はマイナス
- アルゴリズムは，売却益$=0$ を出力

● 対処法
- 初期値として $mxp=sp[1]-sp[0]$ を設定
- より大きい売却益があれば，mxp を更新

● 大切なこと
- 起こりえる場合を網羅的に検討

第１章　アルゴリズムの重要性

1.1 アルゴリズムとは

1.2 アルゴリズムの記述


1.3 アルゴリズムの効率

1.4 アルゴリズムの最適性

1.3　アルゴリズムの効率

最大売却益アルゴリズム

- **アルゴリズム1.1**
  - 計算時間：減算　$n(n-1)/2$ 回
  - 比較　$n(n-1)/2$ 回
- **アルゴリズム1.2，1.3**
  - 計算時間：減算　$n-1$ 回
  - 比較　$n(n-1)/2\ +\ n-1$ 回
- **どれも，概ね $\boldsymbol{n^2}$ に比例する計算時間**
  - $\boldsymbol{O}(\boldsymbol{n^2})$ と表す
    - オーダ $\boldsymbol{n^2}$ またはビッグオー n^2と読む

漸近的計算量

- **O記法（オーダ記法）**
  - 漸近的計算量の表し方
  - 主要項以外の項を無視
  - 主要項の係数を無視

例：$\boldsymbol{an}+b\ \ (a>0)$　　　　→ $\boldsymbol{O}(\boldsymbol{n})$

$\boldsymbol{cn^2}+dn+e\ \ (c>0)$　→ $\boldsymbol{O}(\boldsymbol{n^2})$

⇧

主要項

O記法（オーダ記法）　p.29

$\boldsymbol{O}(\boldsymbol{f}(\boldsymbol{n}))$　　オーダ $f(n)$，ビッグオー $f(n)$

スモールオー $o(f(n))$ もある

ある正定数 n_0, c が存在し，

$n \geq n_0$ を満たすすべての n について，

$\boldsymbol{g}(\boldsymbol{n}) \leq \boldsymbol{c}\cdot\boldsymbol{f}(\boldsymbol{n})$ を満たす関数 $g(n)$ の集合

- $\boldsymbol{g}(\boldsymbol{n}) \in \boldsymbol{O}(\boldsymbol{f}(\boldsymbol{n}))$ を $\boldsymbol{g}(\boldsymbol{n}) = \boldsymbol{O}(\boldsymbol{f}(\boldsymbol{n}))$ と書くことも多い
- n が十分大きい場合の $g(n)$ の上界
- $an+b$ を表すのに正しいのはどれ？

$\cancel{O(\log n)}$　　$\cancel{O(\sqrt{n})}$　　$O(n)$

$O(5n)$　　$O(n^2)$　　$O(n+\sqrt{n})$

なるべく小さく簡単な関数で表す　$\boldsymbol{O}(\boldsymbol{n})$

O記法に関する演算

$f_1(n) \in O(g_1(n))$,　$f_2(n) = O(g_2(n))$

a.　$\boldsymbol{f_1(n)+f_2(n) = O(g_1(n)+g_2(n))}$

$f_1(n) \leq c_1\cdot g_1(n)$　　（$n \geq n_1$）

$f_2(n) \leq c_2\cdot g_2(n)$　　（$n \geq n_2$）

$\therefore\ f_1(n)+f_2(n)$

$\leq c_1\cdot g_1(n)+c_2\cdot g_2(n)$　　（$n \geq \max(n_1, n_2)$）

$\leq c(g_1(n)+g_2(n))$　　（$c = \max(c_1, c_2)$）

b.　$\boldsymbol{f_1(n)\cdot f_2(n) = O(g_1(n)\cdot g_2(n))}$

Ω記法（オメガ記法）　p.29

$\boldsymbol{\Omega(f(n))}$　　**オメガ** $f(n)$，**ビッグオメガ** $f(n)$

スモールオメガ $\omega(f(n))$ もある

ある正定数 n_0, c **が存在し，**
$n \geq n_0$ **を満たすすべての** n **について，**
$\boldsymbol{g(n) \geq c \cdot f(n)}$ **を満たす関数** $g(n)$ **の集合**

- $\boldsymbol{g(n) \in \Omega(f(n))}$ **を** $\boldsymbol{g(n) = \Omega(f(n))}$ **と書くことも多い**
- n **が十分大きい場合の** $g(n)$ **の下界**
- $an + b$ **を表すのに正しいのはどれ？**

$\Omega(\log n)$　　$\Omega(\sqrt{n})$　　$\Omega(n)$

$\Omega(5n)$　　$\cancel{\Omega(n^2)}$　　$\Omega(n + \sqrt{n})$

なるべく小さく簡単な関数で表す　$\Omega(\boldsymbol{n})$

Θ記法（シータ記法）　p.29

$\boldsymbol{f(n) \in \Theta(g(n))}$　　**シータ** $g(n)$

スモールシータはない

$f(n) \in O(g(n))$ **かつ** $f(n) \in \Omega(g(n))$

- $\boldsymbol{g(n) \in \Theta(f(n))}$ **を** $\boldsymbol{g(n) = \Theta(f(n))}$ **と書くことも多い**
- n **が十分大きい場合の** $g(n)$ **の上下界**
- $an + b$ **を表すのに正しいのはどれ？**

$\cancel{\Theta(\log n)}$　　$\cancel{\Theta(\sqrt{n})}$　　$\Theta(n)$

$\Theta(5n)$　　$\cancel{\Theta(n^2)}$　　$\Theta(n + \sqrt{n})$

なるべく小さく簡単な関数で表す　$\boldsymbol{\Theta(n)}$

他の漸近的計算量の記法（参考）

$\boldsymbol{o(f(n))}$　　**スモールオー** $f(n)$

任意の正定数 c **に対し，ある正定数** n_0 **が存在し，**
$n \geq n_0$ **を満たすすべての** n **について，**
$\boldsymbol{g(n) < c \cdot f(n)}$ **を満たす関数** $g(n)$ **の集合**

- O**記法との違いに注意**
  - ●**真に大きい上界**
- $an + b$ **を表すのに正しいのはどれ？**

$\cancel{o(\log n)}$　　$\cancel{o(\sqrt{n})}$　　$\cancel{o(n)}$

$\cancel{o(5n)}$　　$o(n^2)$　　$o(\cancel{n + \sqrt{n}})$

他の漸近的計算量の記法（参考）

$\boldsymbol{\omega(f(n))}$　　**スモールオメガ** $f(n)$

任意の正定数 c **に対し，ある正定数** n_0 **が存在し，**
$n \geq n_0$ **を満たすすべての** n **について，**
$\boldsymbol{g(n) > c \cdot f(n)}$ **を満たす関数** $g(n)$ **の集合**

- Ω**記法との違いに注意**
  - ●**真に小さい下界**
- $an + b$ **を表すのに正しいのはどれ？**

$\omega(\log n)$　　$\omega(\sqrt{n})$　　$\cancel{\omega(n)}$

$\cancel{\omega(5n)}$　　$\cancel{\omega(n^2)}$　　$\omega(\cancel{n + \sqrt{n}})$

最大売却益アルゴリズム１.４

- アルゴリズム１.４：（Ｂ）方式　改良版

```
mxp = 0;// 利益の最大値 mxp を 0 に初期化
msf = sp[0]; // これまでの最安値 msf を sp[0] に初期化
for (j=1 to n-1) {
    d = sp[j] - msf; // 売却益
    if (d > mxp) mxp = d;
    // 今まで以上の売却益であればmxpの値を更新する
    if (sp[j] < msf) msf = sp[j];
    // これまでの最安値の更新
}
mxpを答として返す;
```

計算時間：減算　$n-1$ 回

比較　$2(n-1)$ 回

$\boldsymbol{O(n)}$

漸近的計算量の重要性

- アルゴリズムのオーダーの改善 ⇒ 劇的な高速化
  - 例：$\boldsymbol{\Theta(n^2)}$ から $\boldsymbol{\Theta(n)}$ に改善
    - 係数が同じなら，

      $n = 100$ で 100 倍高速化

      $n = 1000$ で 1000 倍高速化
    - 計算機が 100 倍高速化したとき，

      同じ時間で解ける問題のサイズ

      $\Theta(n^2)$ なら 10 倍

      $\Theta(n)$ なら 100 倍

第１章　アルゴリズムの重要性

1.1 アルゴリズムとは

1.2 アルゴリズムの記述

1.3 アルゴリズムの効率

1.4 アルゴリズムの最適性

最大売却益アルゴリズム１.４

- アルゴリズム１.４：（Ｂ）方式　改良版
  - 計算時間　$\boldsymbol{O(n)}$

- より高速なアルゴリズムを考案できるか？　NO!
  - 株価を１つでも見落とすと正解を得られない可能性
  - n 個の株価をすべて見ないといけない
  - 絶対に n に比例した時間はかかる
- 最大売却益問題の計算時間の下界は $\Omega(\mathrm{n})$
- 最大売却益アルゴリズム１.４の計算時間は最適

今日のまとめ
第１章　アルゴリズムの重要性

1.1 アルゴリズムとは

1.2 アルゴリズムの記述

1.3 アルゴリズムの効率

1.4 アルゴリズムの最適性

今日の学習目標（振返り）

- アルゴリズムとは何か説明できる
- アルゴリズムの重要性を説明できる
- アルゴリズムの記述を理解でき，アルゴリズムを記述できる
- 漸近的計算量とその重要性を説明できる
  - O記法，Ω記法
- アルゴリズムの最適性とは何か説明できる

データ構造とアルゴリズム　第２回

1. アルゴリズムの重要性
2. 探索問題
3. 基本的なデータ構造
4. 動的探索問題とデータ構造
5. データの整列
6. グラフのアルゴリズム
7. 文字列のアルゴリズム
8. アルゴリズム設計手法

第２章　探索問題

2.1 探索問題とは

2.2 逐次探索の効率

2.3 順序関係を利用した探索

2.4 m-ブロック法

2.5 ２分探索法

2.6 ハッシュ法

今日の学習目標

- 探索問題とは何か、応用例を用いて説明できる
- 探索アルゴリズムを説明できる
  - 逐次探索、m-ブロック法、２分探索法、ハッシュ法
- 探索アルゴリズムの計算時間を説明できる
- 探索アルゴリズムを用いたプログラムを書ける

2.1 探索問題とは

- 探索問題
  - 数値データの集合 S が与えられているときに，S の中からデータ x を探す

    （x が S にない場合，ないことが分かる）
- 探索問題の例

  - 顧客リストから特定の顧客を探す

    氏名，住所，電話番号などから探索
  - ネットワークから特定の情報を探す

    Google, Yahoo などによる検索

    ＊ネットワークでの探索は講義の範囲外

第２章　探索問題

2.1 探索問題とは
2.2 逐次探索の効率
2.3 順序関係を利用した探索
2.4 m-ブロック法
2.5 ２分探索法
2.6 ハッシュ法

2.1 探索問題とは

- **数値データの集合 S が与えられているときに，S の中からデータ x を探す**
  - **（x が S にない場合，ないことが分かる）**
- **数値データの集合 S の保管方法**
  - **この章では配列に格納**
    - **要素の挿入・削除がないから**
      - **→ 挿入・削除は４章で扱う**

s
72
7
6
65
97
9
74
37

逐次探索法

- $s[0..n-1]$：**集合 S（n 個のデータ）**
- x：**探索するデータ**
- **逐次探索法**

```
xを入力する；
i = 0;
do {
    if (x == s[i]) i を返して終了;
    else i=i+1;
} while (i < n);
-1 を返して終了;
```

s
72
7
6
65
97
9
74
37

65

比較回数：最小　１回　　　最大 n 回
平均 $(n+1)/2$ 回
x **が見つかる場合**

逐次探索の平均比較回数

- **成功探索（x が見つかる場合）**
  - **S の全要素が同確率で探索されるなら**

    $(n+1)/2$ **回**

    $(= (1+2+3+\cdots+n)/n)$
- **失敗探索（x が見つからない場合）**

  n **回**
- **成功確率 p の場合**
  - **S の全要素が同確率で探索されるなら**

    $p(n+1)/2 + (1-p)n$ **回**

番兵：有名な高速化技法（テキスト外）

- **逐次探索の比較回数**
  - 最小 1 回　　　最大 n 回
  - 平均 $(n+1)/2$ 回
- **正確に表現すると，要素の比較回数**
  - ✓while の終了判定はカウントしていない

```
xを入力する；
i = 0;
do {
    if (x == s[i]) i を返して終了;
    else i=i+1;
} while (i < n);
-1 を返して終了;
```


番兵：有名な高速化技法（テキスト外）

- **配列を $s[0..n]$ とする（n 個のデータは $s[0..n-1]$ に格納）**
  - $s[n]$ に探索データ x を格納
  - x の探索は必ず成功（x を $s[i]$ で見つけたとする）
    - $0 \leq i \leq n-1$ → 探索データ発見（成功探索）
    - $i = n$ → 探索データ発見できず（失敗探索）

```
xを入力する；
s[n] = x;
i = 0;
while (x != s[i])
    i = i+1;
if (i < n) i を返して終了;
else -1 を返して終了;
```


第２章　探索問題

2.1 探索問題とは

2.2 逐次探索の効率

2.3 順序関係を利用した探索

2.4 m-ブロック法

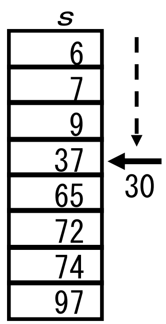
2.5 ２分探索法

2.6 ハッシュ法

2.3 順序関係を利用した探索

- $s[0..n-1]$：集合 S（n 個のデータ）
  - 昇順（小さい順）にソート済の場合
- x：探索するデータ
- ソート列上での逐次探索法

```
xを入力する；
i = 0;
do {
    if (s[i] ≧ x) ループから出る;
    else i=i+1;
} while (i < n);
if (s[i] == x) i を返して終了;
else -1 を返して終了;
```



2.3 順序関係を利用した探索

比較回数（繰り返し中の if 文）

- **成功探索（x が見つかる場合）**
  - S **の全要素が同確率で探索されるなら**

    (1) 回

- **失敗探索（x が見つからない場合）**
  - $x < s[0]$, $s[i] < x < s[i+1]$ $(0 \leq i \leq n-2)$，
    $s[n-1] < x$ **での失敗が同確率で起こるなら**

    (2) 回

- **成功確率 p の場合**

  (3) 回

第２章　探索問題

2.1 探索問題とは

2.2 逐次探索の効率

2.3 順序関係を利用した探索

2.4 m-ブロック法

2.5 ２分探索法

2.6 ハッシュ法

2.4 m-ブロック法

- $s[0..n-1]$：**集合** S（n **個のデータ）**
  - **昇順（小さい順）にソート済の場合**
- x：**探索するデータ**
- m**-ブロック法**
  - s **を** m **個のブロック** $B_0, \ldots, B_{m-1}$ **に分割**

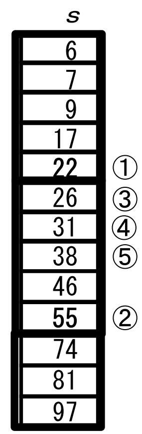
    B_j：$s[jk..(j+1)k-1]$ （$k = n/m$）
  - $B_0, \ldots, B_{m-1}$ **の最大値** $s[(j+1)k-1]$ **と** x **を順に比較（逐次探索）**
  - $x \leq s[(j+1)k-1]$ **なる最初のブロック** B_j **を逐次探索**

単純な逐次探索を２回繰返すだけで
計算時間は大きく短縮

2.4 m-ブロック法

- $s[0..n-1]$：**集合** S（n **個のデータ）**
  - **昇順（小さい順）にソート済の場合**

    38 の探索（$n = 13$）



- $m = 3$ **個のブロックに分割**
- **ブロックの最後の要素（ブロックの最大要素）と順に比較**
- **探すべきブロックがわかれば そのブロックを逐次探索**

m-ブロック法　アルゴリズム２.３

- アルゴリズム２．３：m-ブロック法

```
//ステップ１：xを含むブロックの逐次探索
  j = 0;
  while (j ≦ m-2)
      if (x ≦ s[(j+1)k-1]) ループから出る;
      else j = j+1;
//ステップ２：ブロック内での逐次探索
  i = jk; t = min {(j+1)k-1, n-1};
  while (i < t)
      if (x ≦ s[i]) ループから出る;
      else i = i+1;
  if (x == s[i]) i を返して終了;
  else -1を返して終了;
```


m-ブロック法の比較回数

- 最大比較回数

$$m-1+\left\lceil \frac{n}{m} \right\rceil < m+\frac{n}{m} \text{ 回}$$

$\lceil x \rceil$：x の切上げ（x 以上の最小の整数）

- 最適なブロックサイズ
  - $\boldsymbol{m=\sqrt{n}}$ のとき 最大比較回数は $2\sqrt{n}$

    厳密には，n が平方数の場合の評価

    n が平方数でない場合も同様

第２章　探索問題

2.1 探索問題とは

2.2 逐次探索の効率

2.3 順序関係を利用した探索

2.4 m-ブロック法

2.5 ２分探索法

2.6 ハッシュ法

２.５ ２分探索法

- $s[0..n-1]$：集合 S（n 個のデータ）
  - 昇順（小さい順）にソート済の場合
- x：探索するデータ
- ２分探索法の基本アイデア
  - 初期化：$left=0, right=n-1$

    探索範囲 $s[left..right]$
  - $s[left] \leq x \leq s[right]$ なら $mid=(left+right)/2$
    - x と $s[mid]$ を比較
      - $x < s[mid]$ なら $s[left..mid-1]$ を探索
      - $x = s[mid]$ なら 探索終了
      - $x > s[mid]$ なら $s[mid+1..right]$ を探索

２分探索法　アルゴリズム 2.7 (1)

- アルゴリズム 2.7：２分探索法（４）
  - 最終版：$left$, $right$ の扱いがトリッキーな方法

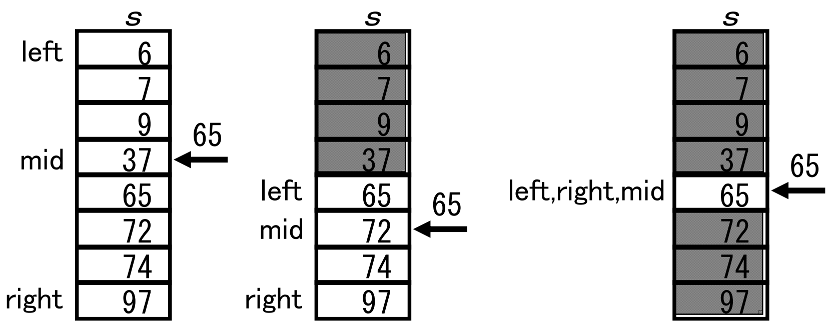
```
x を入力する;
if (x < s[0] または x > s[n-1]) -1を返して終了;
left = 0; right = n-1; //探索区間の設定
do {
   mid = (left + right)/2; //探索区間の中央
   if (x < s[mid]) right = mid-1;
   else left = mid+1;
   } while (left ≦ right);
if (x == s[right]) then right を返して終了;
else -1 を返して終了;
```


２分探索法　アルゴリズム 2.7 (2)

```
do {
   mid = (left + right)/2; //探索区間の中央
   if (x < s[mid]) right = mid-1;
   else left = mid+1;
   } while (left ≦ right);
```

- ループ不変式：ループ内で常に成立する性質
  - x が S に存在するなら，$\boldsymbol{x \leq s[right]}$
  - x が S に存在するなら，$\boldsymbol{x \geq s[left-1]}$
- ループ終了時：$\boldsymbol{right = left - 1}$
  - x が S に存在するなら，$\boldsymbol{x = s[right]}$

２分探索法　アルゴリズム 2.7　実行例



$x = s[mid]$ なので，
$left = mid + 1$ を実行して
while 文から脱出
$x = s[right]$ が成立

２分探索法　比較回数

- 探索範囲
  - $s[left\,..right]$ ：サイズ $right - left + 1$
  - 実行開始時

    $s[0\,..n-1]$：サイズ n
- 比較することに探索範囲のサイズは半減
  探索範囲のサイズが１になるまで
  - 比較回数：最大 $\log n + 1 = O(\log n)$ 回

mーブロック法と２分探索

- $m = 2$ の場合の m-ブロック法

```
//ステップ１：xを含むブロックの探索
  j = 0;
  if (x ≦ s[k-1]) ループから出る;
     else j = j+1;
//ステップ２：ブロック内での探索
  i = jk; t = min {(j+1)k-1, n-1};
  while (i < t)
     if (x ≦ s[i]) ループから出る;
     else i = i+1;
  if (x == s[i]) i を返して終了;
  else -1を返して終了;
```

ブロック B_j 内の選択

再帰的に ステップ１を実行

⇒ ２分探索

第２章　探索問題

2.1 探索問題とは
2.2 逐次探索の効率
2.3 順序関係を利用した探索
2.4 m-ブロック法
2.5 ２分探索法
2.6 ハッシュ法

2.6 ハッシュ法

- 逐次探索，m-ブロック法，２分探索
  - 大小比較を基礎とする方法（比較法）
  - x に近い値を求めることにも拡張可能
  - ２分探索：比較 $O(\log n)$ 回　（最適）
- ハッシュ法
  - 大小比較以外の方法を利用
  - x に近い値を求めることは一般に不可能
  - 比較：平均 $O(1)$ 回　（$O(1) =$ 定数）

ハッシュ法

- ハッシュ法：基本アイデア
  - サイズ m（n の1.5～2倍）の配列を利用
  - ハッシュ関数 h：
    データ定義域 → $\{0, 1, \ldots, m-1\}$
  - データ x は $S[h(x)]$ に格納
  - データ x の探索は $S[h(x)]$ を参照

ハッシュ法：データの衝突

- ハッシュ関数 h：データ定義域 → $\{0, 1, \ldots, m-1\}$
- データ x は $s\,[h(x)]$ に格納
- データ定義域のサイズ $> m$ なら
  - h(x)= h(y) なる x≠y が存在
  - x, y を同時に $s\,[h(x)]$ （$= s[h(y)]$ ）に格納できない

データの衝突

ハッシュ法：データ衝突時の処理

- データ x 格納時
  - $s[h(x)]$ に他データが存在（データの衝突）

    $\Rightarrow s[h(x)+1], s[h(x)+2], \ldots$ を順に調べ，

    最初の空の場所に格納

    $s[m-1]$ の次は $s[0]$ に戻る
- データ x 探索時
  - $s[h(x)]$ に他データが存在（データの衝突）

    $\Rightarrow s[h(x)+1], s[h(x)+2], \ldots$ を

    x か空の場所を見つけるまで探索

ハッシュ法：データ格納　アルゴリズム 2.8

- ハッシュ法でデータを配列に蓄える手続き

仮定
格納データは0でない

アルゴリズム 2.8

```
ハッシュ表 htb[0] ～ htb[m-1] の内容を 0 に初期化;
for (i=0 to n-1) do {
    i 番目のデータを x とする;
    j = hash(x); //ハッシュ値を計算
    while (htb[j] != 0) //ハッシュ表で空いている場所を探す
        j = (j+1) % m; //次の場所へ移動
    htb[j] = x; //最初の空き場所に x を格納
}
```


ハッシュ法：データ格納の例

h(x) =x mod 11

25 の挿入

0	
1	
2	
3	3
4	15
5	5
6	25
7	7
8	
9	20
10	

挿入：５　７　１５
２０　３　２５

ハッシュ法：データ探索　アルゴリズム 2.9

- 与えられたデータを探索する手続き

アルゴリズム 2.9

```
探索すべきデータ x を入力する;
j = hash(x);
while (htb[j] != 0 かつ htb[j] != x)
    j = (j+1) % m; //次の場所へ移動
if (htb[j] == x) j を返して終了;
else -1 を返して終了;
```


ハッシュ法：データ探索の例

h(x)＝x mod 11

② ２５の探索

③ ６の探索

① ２０の探索

0	
1	
2	
3	3
4	15
5	5
6	25
7	7
8	
9	20
10	

ハッシュ法の比較回数

- ハッシュ法の比較回数
  - 占有率$\alpha = n/m$ に依存
  - 平均成功探索回数

$$(1 + 1/(1-\alpha))/2$$

  - 平均失敗探索回数

$$(1 + 1/(1-\alpha)^2)/2$$

今日のまとめ　　第２章　探索問題

2.1 探索問題とは

2.2 逐次探索の効率

2.3 順序関係を利用した探索

2.4 m-ブロック法

2.5 ２分探索法

2.6 ハッシュ法

今日の学習目標（振返り）

- 探索問題とは何か、応用例を用いて説明できる
- 探索アルゴリズムを説明できる
  - 逐次探索、m-ブロック法、２分探索法、ハッシュ法
- 探索アルゴリズムの計算時間を説明できる
- 探索アルゴリズムを用いたプログラムを書ける

データ構造とアルゴリズム　第３回

1. アルゴリズムの重要性
2. 探索問題
3. 基本的なデータ構造
4. 動的探索問題とデータ構造
5. データの整列
6. グラフのアルゴリズム
7. 文字列のアルゴリズム
8. アルゴリズム設計手法

第３章　基本的なデータ構造

- 3.1 配列と連結リスト構造
- 3.2 連結リスト構造の利点
- 3.3 ２分探索法に対応するデータ構造
- 3.4 スタックとキューの概念
- 3.5 スタックの実現
- 3.6 キューの実現
- 3.7 ヒープ

今日の学習目標

- 基本的なデータ構造を説明できる
  - 配列，連結リスト，スタック，キュー，ヒープ
- データ構造に対する各種操作とその時間計算量を説明できる
  - 探索，挿入，削除

3.1 配列と連結リスト構造(1)

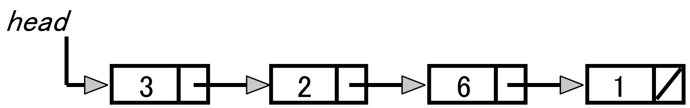
- 配列（array）
  - 要素は連続したメモリに配置
  - 参照（k 番目の要素） $O(1)$ 時間（ランダムアクセス）
  - 列の途中への挿入・削除　$O(n)$ 時間
    - n：配列内のデータ数
    - 挿入・削除データ以降のデータを１つずつ後・前にずらす
  - 宣言時に大きさを決める
    - 状況によっては困難
    - うまく決めれば，効率がよい
  - ２次元以上の多次元配列もある

	s
$s[0]$	72
$s[1]$	7
$s[2]$	6
$s[3]$	65
$s[4]$	97
$s[5]$	9
$s[6]$	74
$s[7]$	37

3.1 配列と連結リスト構造(2)

- 連結リスト

```
struct LIST {
    int data;
    struct LIST *next;
};
```

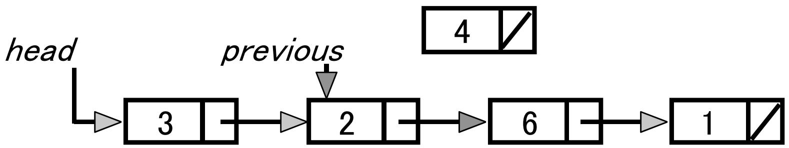


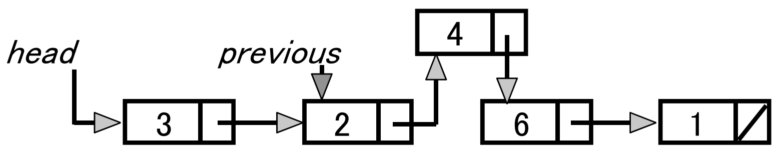
第３章　基本的なデータ構造

3.1 配列と連結リスト構造
3.2 連結リスト構造の利点
3.3 ２分探索法に対応するデータ構造
3.4 スタックとキューの概念
3.5 スタックの実現
3.6 キューの実現
3.7 ヒープ

3.2 連結リスト構造の利点

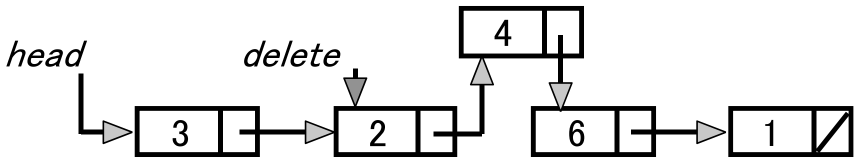
●挿入（指定セルの次へ挿入）：$O(1)$ 時間
- 最初のセルとしての挿入は特別な処理が必要
  ポインタheadの書換え

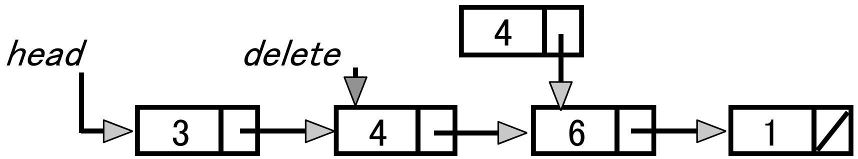




リストからの削除

- 挿入（指定セルの削除）：$O(1)$ 時間
  - 実際に削除できるのは指定セルの次のセル
  - データコピーによって指定セル削除と等価
  - 最後のセルの削除は特別な処理が必要：$O(n)$ 時間





配列とリストの比較

- 配列
  - 参照　$O(1)$ 時間　（ランダムアクセス）
  - 列の途中への挿入・削除　$O(n)$ 時間（n：データ数）
  - データ１個あたりの記憶領域：小
  - 宣言時に大きさを決定
- リスト
  - 参照　$O(n)$ 時間（n：列の長さ）
  - 挿入・削除　$O(1)$
  - データ１個あたりの記憶領域：大
  - 大きさは動的に変化

第３章　基本的なデータ構造

3.1 配列と連結リスト構造

3.2 連結リスト構造の利点

3.3 ２分探索法に対応するデータ構造

3.4 スタックとキューの概念

3.5 スタックの実現

3.6 キューの実現

3.7 ヒープ

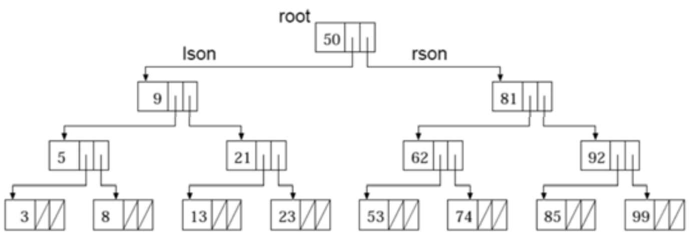
3.3 ２分探索法に対応するデータ構造

- ２分探索法
  - 探索データとの比較結果で次の比較位置を決定

```
struct BSTnode {
    int key;
    struct BSTnode *lson, *rson;
};
```

lson：探索キー＜比較キー　のときの次の比較位置

rson：探索キー＞比較キー　のときの次の比較位置



3.3 ２分探索法に対応するデータ構造

- 探索手続き　プログラム３.５

```
x を入力する;
v = root（根）とする.
while（v が NULL でない ）{
    if（x == 節点vのキー値(v->data)）v を出力して終了
    if（x < 節点vのキー値）v = v の左の子とする.
    else v = v の右の子とする
}
見つからなかったと報告して終了
```

探索時間：$O(\log n)$

第３章　基本的なデータ構造

3.1 配列と連結リスト構造

3.2 連結リスト構造の利点

3.3 ２分探索法に対応するデータ構造

3.4 スタックとキューの概念

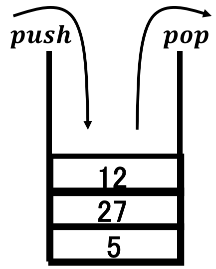
3.5 スタックの実現

3.6 キューの実現

3.7 ヒープ

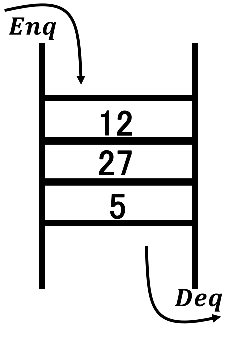
３.４ スタックとキューの概念

- スタック（プッシュダウンスタック）
  - 有用な抽象データ型
    - データの列
    - 操作：空スタック作成，挿入，削除
    - 具体的な実現方法を問わない
  - データの挿入・削除：列の同じ一方の端だけで行える
  - 操作
    - $push$：データの挿入
      $O(1)$ 時間
    - pop ：データの取出し（削除）
      $O(1)$ 時間
  - リストでも配列でも実現可能



３.４ スタックとキューの概念

- キュー（待ち行列，FIFOキュー（ﾌｧｲﾌｫｷｭｰ））
  - 有用な抽象データ型
  - データの挿入・削除
    - 挿入を列の一方の端だけ，
      削除を列の他方の端だけで行える
  - 操作
    - $Enqueue$：データの挿入
      $O(1)$ 時間
    - $Dequeue$ ：データの削除
      $O(1)$ 時間
  - リストでも配列でも実現可能



第３章　基本的なデータ構造

3.1 配列と連結リスト構造

3.2 連結リスト構造の利点

3.3 ２分探索法に対応するデータ構造

3.4 スタックとキューの概念

3.5 スタックの実現

3.6 キューの実現

3.7 ヒープ

3.5 スタックの実現

- 配列によるスタックの実現

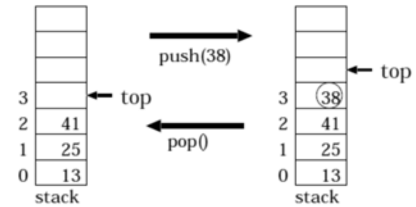
```
void push(int x)
{
    stack[top++] = x;
}
```

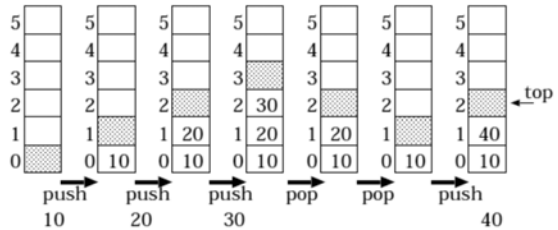
```
int pop(void)
{
    return stack[--top];
}
```

  - オーバーフロー（$push$ でデータ数が配列サイズを超える），アンダーフロー（データがないのに pop）も考慮する必要あり

3.5 スタックの実現

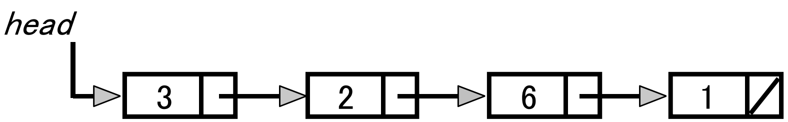
- 配列によるスタックの実現





3.5 スタックの実現

- 連結リストによるスタックの実現
  - 連結リストの先頭に挿入（$push$）
  - 連結リストの先頭から取出し（pop）



第３章　基本的なデータ構造

3.1 配列と連結リスト構造

3.2 連結リスト構造の利点

3.3 ２分探索法に対応するデータ構造

3.4 スタックとキューの概念

3.5 スタックの実現

3.6 キューの実現

3.7 ヒープ

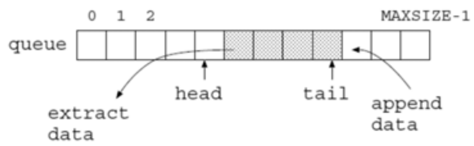
3.6 キューの実現

● 配列によるキューの実現

```
void Enqueue (int x)
{
    tail= (tail+1) %MAXSIZE; queue [tail]  = x;
}
```

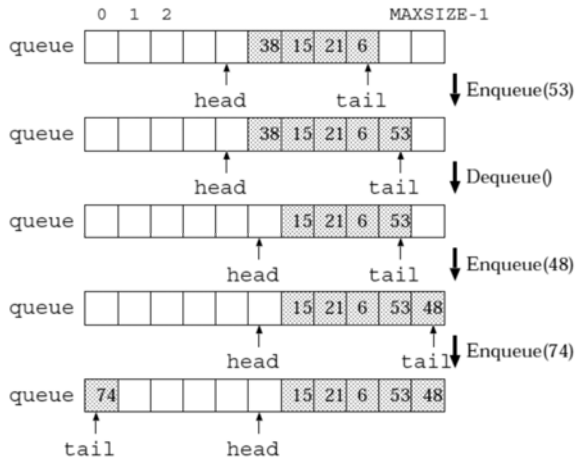
```
int Dequeue (void)
{
    head= (head+1) %MAXSIZE; return queue [head] ;
}
```

- 配列をリングバッファとして利用
- オーバーフロー，アンダーフローも考慮する必要あり



3.6 キューの実現

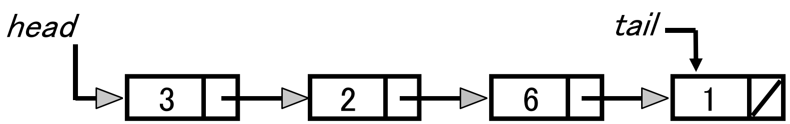
● 配列によるキューの実現



3.6　キューの実現

● 連結リストによるスタックの実現

- 連結リストの最後尾に挿入（Enqueue）
- 連結リストの先頭から取出し（Dequeue）



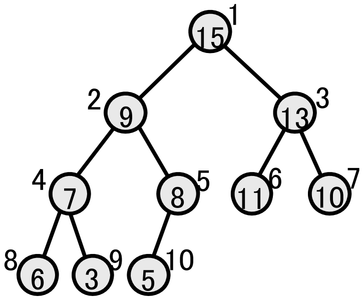
第３章　基本的なデータ構造

3.1 配列と連結リスト構造
3.2 連結リスト構造の利点
3.3 ２分探索法に対応するデータ構造
3.4 スタックとキューの概念
3.5 スタックの実現
3.6 キューの実現
3.7 ヒープ

3.7 ヒープ

- ヒープ
  - ２分木を配列 $heap[1..n]$ で実現（n：データ数）
    - ポインタを使用しない
    - $heap[1]$：根
    - $heap[k]$ の左の子：$heap[2k]$
    - $heap[k]$ の右の子：$heap[2k+1]$
  - 親のデータ $\geq$ 子のデータ
    - 根は最大のデータを持つ

3.7 ヒープ



1	15
2	9
3	13
4	7
5	8
6	11
7	10
8	6
9	3
10	5

3.7 ヒープ

- データの格納　プログラム３.１３

```
void PushHeap (int x)
{
   int i, j;
   if (++n >= MAXSIZE) stop ("Heap Overflow");
   else {
      heap [n] = x;
      i=n; j=i/2;
      while (j>0 && x > heap [j] ) {
         heap [i] = heap [j]; i=j; j=i/2;
      }
      heap [i] = x;
   }
}
```

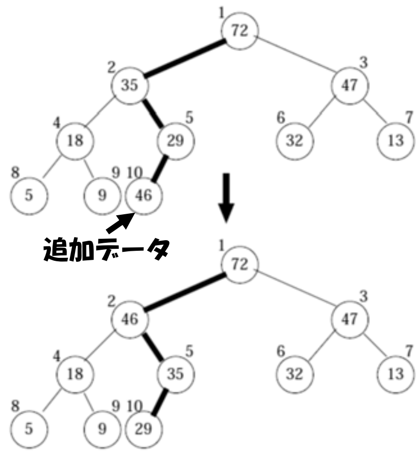
n：データ数

i：データ x の現在ノード
j：x の現在ノードの親

親ノード j のデータ $< x$
ノード j のデータを
ノード i に移動

3.7 ヒープ

- データの格納　プログラム３.１３



3.7 ヒープ

● 最大データの取出し　プログラム3.14

```
int DeleteMax(void) {
    int x, i, j, t;
    if (n == 0) stop("Heap Underflow");
    else {
        x=heap[1]; heap[1]=heap[n--]; i=1;
        while (i*2<=n) {
            j=i*2;
            if (i*2+1<=n && heap[i*2]<heap[i*2+1]) j=i*2+1;
            if (heap[i]>=heap[j]) break;
            else {t=heap[i]; heap[i]=heap[j]; heap[j]=t; }
            i=j;
        }
    }
    return x;
}
```

根に移動したデータの現在位置

根のデータが最大

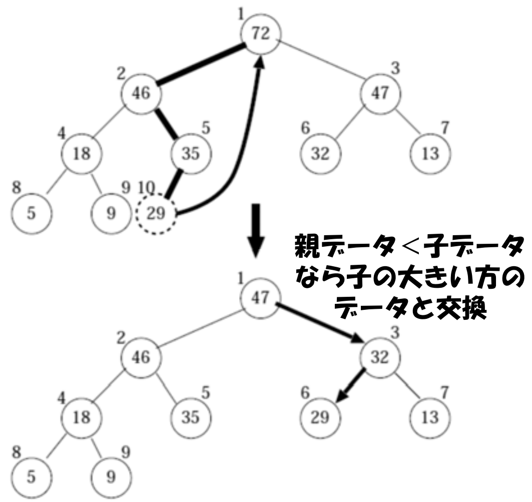
配列最後のデータを根に移動

i と j のデータを交換

i の子の内，大きいデータを持つ方が j

3.7 ヒープ

● 最大データの取出し　プログラム3.14



3.7 ヒープ

● データの格納

$O(log\,n)$ 時間　（n：データ数）

● 最大データの取出し

$O(\log n)$ 時間　（n：データ数）

今日のまとめ　第３章 基本的なデータ構造

3.1 配列と連結リスト構造

3.2 連結リスト構造の利点

3.3 ２分探索法に対応するデータ構造

3.4 スタックとキューの概念

3.5 スタックの実現

3.6 キューの実現

3.7 ヒープ

今日の学習目標（振返り）

- 基本的なデータ構造を説明できる
  - 配列，連結リスト，スタック，キュー，ヒープ
- データ構造に対する各種操作とその時間計算量を説明できる
  - 探索，挿入，削除

データ構造とアルゴリズム　第４，５回

1. アルゴリズムの重要性
2. 探索問題
3. 基本的なデータ構造
4. 動的探索問題とデータ構造
5. データの整列
6. グラフのアルゴリズム
7. 文字列のアルゴリズム
8. アルゴリズム設計手法

第４章　動的探索問題とデータ構造

4.1 線形リスト上での探索

4.2 ２分探索木

4.3 平衡２分探索木

4.4 動的ハッシュ法

「第2章 探索問題」との違い

新たなデータの挿入，
既にあるデータの削除
を考える

この章の学習目標

- 探索問題での挿入・削除とは何か，応用例を用いて説明できる
- 2分探索木，平衡2分探索木とは何か説明できる
- 挿入・削除のアルゴリズムを説明できる
  - 逐次探索，２分探索，2分探索木，2色木，ハッシュ法
- 挿入・削除アルゴリズムの計算時間を説明できる
  - 最悪時だけでなく，平均計算時間も
- 挿入・削除を含めた探索アルゴリズムのプログラムを書ける

４.１ 線形リスト上での探索

- ２分探索（配列で実現している場合）
  - データを小さい順に格納
    - 探索に必要な時間 $O(\log n)$
    - 挿入，削除に必要な時間 $O($？$)$
- 逐次探索
  - データの格納順は任意
    - 探索に必要な時間 $O(n)$
    - 挿入，削除に必要な時間 $O($？$)$

選択肢：$O(1)$，$O(\log n)$, $O(\sqrt{n})$，$O(n)$，$O(n^2)$

第４章　動的探索問題とデータ構造

4.1 線形リスト上での探索

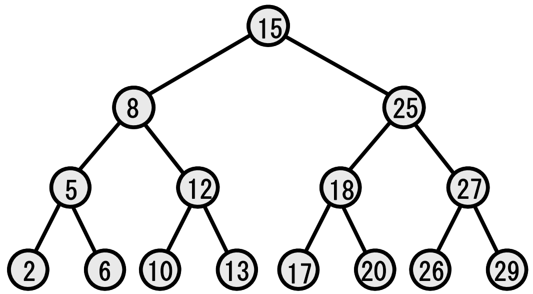
4.2 ２分探索木

4.3 平衡２分探索木

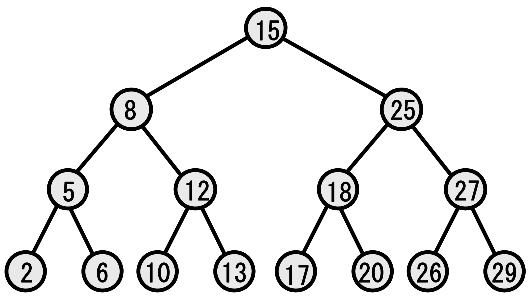
4.4 動的ハッシュ法

４.２ ２分探索木

- ３.３節 ２分探索法に対応するデータ構造
  - 図3.5（p.42）の木型データ構造（２分探索木）
  - データの挿入，削除は考えていない
    - ●完全２分木状



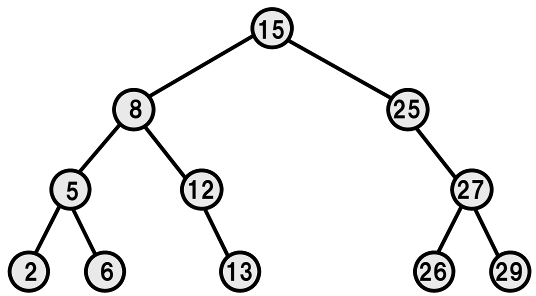
２分探索木と２分探索法の対応



2
5
6
8
10
12
13
15
17
18
20
25
26
27
29

２分探索木条件

- ２分探索木条件（完全２分木とは限らない）
  - ２分探索木の各節点 v に対し，
    - ●v の左部分木のデータ（キー）は v 以下
    - ●v の右部分木のデータ（キー）は v 以上



２分探索木の探索

- ２分探索木条件
  - ２分探索木の各節点 v に対し，
    - v の左部分木のデータ（キー）は v 以下
    - v の右部分木のデータ（キー）は v 以上
- 探索手続き

```
x を入力する;
v = root（根）とする.
while (v が NULL でない ) {
    if (x == 節点vのキー値(v->data)) v を出力して終了;
    if (x < 節点vのキー値) v = v の左の子とする;
    else v = v の右の子とする;
}
見つからなかったと報告して終了;
```

２分探索木へのデータの挿入

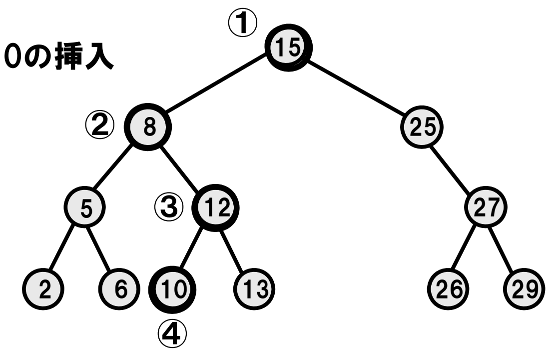
- 挿入手続き（アルゴリズム４.１）
  - キー x のレコードの挿入
  - 探索時間：根から挿入レコードまでの距離に比例

```
v = root;
while (v がNULLでない) {
    if (x < vのキー値) v = vの左の子;
    else v = vの右の子;
}
新しいレコードを作り，そこへのポインタをvとする;
vのキー値 = x とし，vの左右の子をNULLにする.
```


２分探索木へのデータの挿入

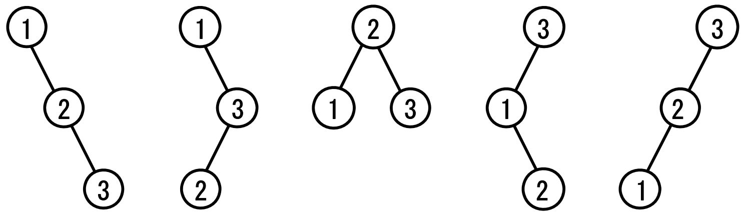
- 挿入手続き
  - キー x のレコードの挿入
  - 挿入時間：根から挿入レコードまでの距離に比例

10の挿入



４.２.２ ２分探索木のバランス

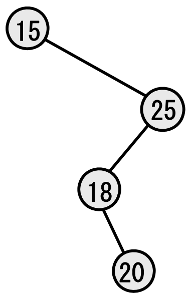
- レコードの挿入順により，異なる２分探索木を構成



(a) 1,2,3　(b) 1,3,2　(c) 2,1,3 2,3,1　(d) 3,1,2　(e) 3,2,1

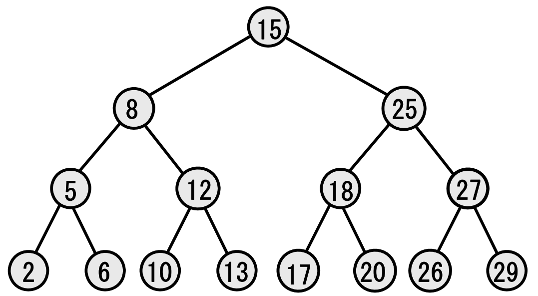
4.2.3 最悪の場合の探索時間

- 探索時間（n：２分探索木中のデータ数）
  - 比較回数：探索データの深さ（根からの距離）＋１
    - 最悪時　$O(n)$



4.2.3 最悪の場合の探索時間

- 探索時間（n：２分探索木中のデータ数）
  - 比較回数：探索データの深さ（根からの距離）＋１
    - 理想的な２分探索木
      - データをなるべく根の近くに配置
      - 最悪時　$O(\log n)$



4.2.4 平均的な探索時間

- 探索時間（n：２分探索木中のデータ数）
  - 比較回数：
    探索データの深さ（根からの距離）＋１
  - 平均探索時間
    - 各データの探索確率を $1/n$ と仮定
      - 探索の失敗は考えない
    - 鎖状の２分探索木　$O(n)$
    - 完全２分探索木　$O(\log n)$
    - 平均的な２分探索木では？

平均的な木での平均探索時間

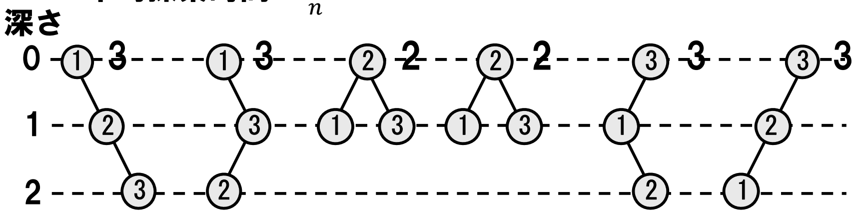
- 平均的な木

  データの挿入順序が決まれば2分探索木は決まる

  $n!$ 通りの挿入順序は等確率と仮定
- 平均探索時間評価のための仮定
  - （1）全てのキーの探索は等確率
  - （2）探索の失敗は考慮しない
- 平均探索時間：$O(\log n)$

平均的な木での平均探索時間

- 探索時間
  - 比較回数：探索データの深さ（根からの距離）＋１
- 平均探索時間：$O(\log n)$
- 平均的な木：$n!$ 通りの挿入順序が等確率
- D_n：n 頂点の２分探索木の全節点の深さの総和の平均
  - 平均探索時間：$\frac{D_n}{n}+1$



$$D_n = \frac{3+3+2+2+3+3}{3!} = \frac{8}{3}$$

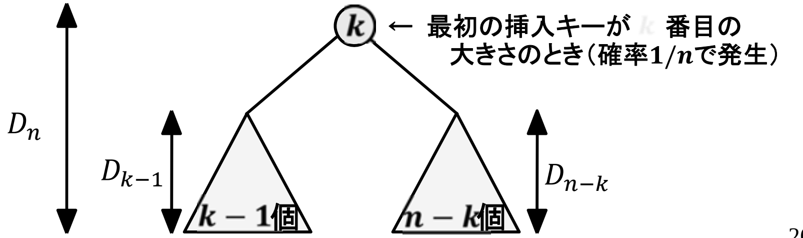
平均的な木での平均探索時間

- 平均探索時間：$\frac{D_n}{n}+1$
  - D_n：n 頂点の２分探索木の全節点の深さの総和の平均
    - $D_0 = D_1 = 0$
    - $D_n = \left(\sum_{k=1}^{n}(n-1+D_{k-1}+D_{n-k})\right)/n$

左右部分木の各節点の深さが 1 増加

左部分木

右部分木



平均的な木での平均探索時間

- 平均探索時間：$\frac{D_n}{n}+1$
  - D_n：n 頂点の２分探索木の全節点の深さの総和の平均
    - $D_0 = D_1 = 0$
    - $D_n = \left(\sum_{k=1}^{n}(n-1+D_{k-1}+D_{n-k})\right)/n$
  - この漸化式を解くと，$D_n = O(n\log n)$

- 平均探索時間： $O(\log n)$

漸化式を解いてみよう (1)

- $D_1 = 0$
- $D_n = \left(\sum_{k=1}^{n}(n-1+D_{k-1}+D_{n-k})\right)/n$
- この漸化式を解くと，$D_n = O(n\log n)$

$$D_n = \left(\sum_{k=1}^{n}(n-1+D_{k-1}+D_{n-k})\right)/n$$
$$= \left(\sum_{k=1}^{n}(n-1+2D_{k-1})\right)/n$$
$$= \mathrm{n}-1+\frac{2}{n}\cdot\sum_{k=1}^{n}D_{k-1}$$
$$= \mathrm{n}-1+\frac{2}{n}\cdot\sum_{k=0}^{n-1}D_{k}$$

$A_n = D_n - 2$ とおく

$$A_n + 2 = n-1+\frac{2}{n}\cdot\sum_{k=0}^{n-1}(A_k+2)$$
$$A_n = n-1+\frac{2}{n}\cdot\sum_{k=0}^{n-1}A_k + 4 - 2 = n+1+\frac{2}{n}\cdot\sum_{k=0}^{n-1}A_k$$

漸化式を解いてみよう(2)

- $A_n = n + 1 + \frac{2}{n} \cdot \sum_{k=0}^{n-1} A_k$

$n \cdot A_n = n(n+1) + 2 \cdot \sum_{k=0}^{n-1} A_k$ (1)

$(n-1)A_{n-1} = (n-1)n + 2 \cdot \sum_{k=0}^{n-2} A_k$ (2)

$(1) - (2)$

$n \cdot A_n - (n-1)A_{n-1} = 2n + 2A_{n-1}$

$n \cdot A_n = (n+1)A_{n-1} + 2n$

両辺を $n(n+1)$ **で割る**

$$\frac{A_n}{n+1} = \frac{A_{n-1}}{n} + \frac{2}{n+1} = \frac{A_{n-2}}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$= \frac{A_1}{2} + \frac{2}{3} + \frac{2}{4} + \cdots + \frac{2}{n} + \frac{2}{n+1}$$

漸化式を解いてみよう(3)

- $\frac{A_n}{n+1} = \frac{A_1}{2} + \frac{2}{3} + \frac{2}{4} + \cdots + \frac{2}{n} + \frac{2}{n+1}$

$A_1 = D_1 - 2$ より，$A_1 = -2$

$$\frac{A_n}{n+1} = -1 + \frac{2}{1} + \frac{2}{2} + \frac{2}{3} + \frac{2}{4} + \cdots + \frac{2}{n} + \frac{2}{n+1} - (\frac{2}{1} + \frac{2}{2})$$

$= 2H_{n+1} - 4$ $(H_n = O(\log n)$ 調和級数）

$= O(\log n)$

$A_n = O(n \log n)$

$A_n = D_n - 2$ より，$D_n = O(n \log n)$

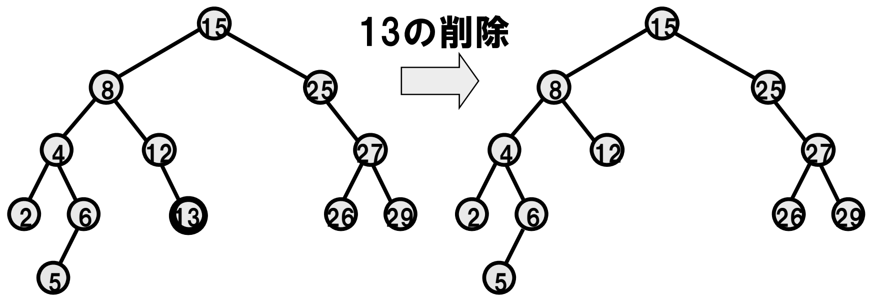
２分探索木からの削除

- **削除の方法**
  1. **削除するキーを探索**
  2. **そのキーを持つ節点を削除**
  3. **２分探索木に変形**
     - **(a) 削除する節点が葉の場合**
     - **(b) 削除する節点が子を１個もつ場合**
     - **(c) 削除する節点が子を２個もつ場合**

２分探索木からの削除

(a) 削除する節点が葉の場合

単純に節点を削除

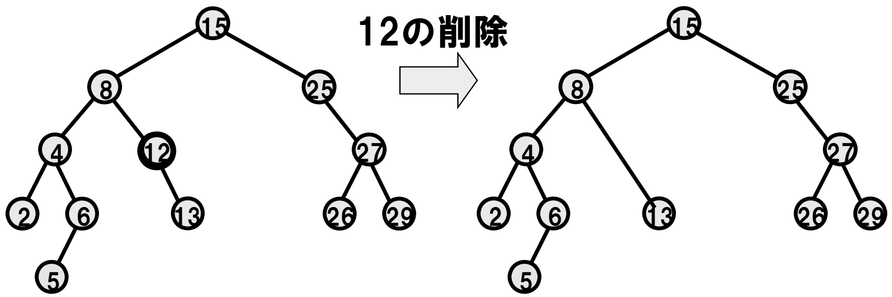


２分探索木からの削除

(b) 削除する節点が子を１個もつ場合

節点を削除し，親と子を接続する

２分探索木条件が成立することに注意

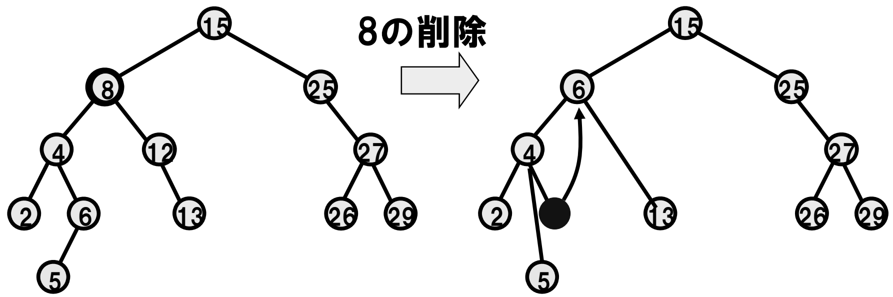


２分探索木からの削除

(c) 削除する節点が子を２個もつ場合

節点を削除し，その左部分木中の最大キーを移動

２分探索木条件が成立することに注意



２分探索木からの削除

(c) 削除する節点が子を２個もつ場合

節点を削除し，その左部分木中の最大キーを移動

２分探索木条件が成立することに注意

左部分木最大のキー：

探索が容易

削除が容易：右の子をもたず，子は高々１個

左部分木の根から探索開始
1. 右の子があれば，右部分木で探索
2. 右の子がなければ，そのノードが最大値

２分探索木からの削除

- 削除にかかる時間：最悪 $O(n)$　平均 $O(\log n)$
  1. 削除するキーを探索
     - 最悪 $O(n)$　平均 $O(\log n)$
  2. そのキーを持つ節点を削除
     - $O(1)$
  3. ２分探索木に変形
     - (a)，(b) $O(1)$
     - (c)　最悪 $O(n)$　平均 $O(\log n)$

直前キー，直後キー，最小キー，最大キーの探索

- 最小キー，最大キーの探索
  - ２分探索木中の最小キー，最大キーを探索する
  - $O(h)$時間　（h：２分探索木の高さ）
- 直前キー，直後キーの探索
  - ２分探索木中のキーを昇順にならべたときに，x の直前のキー，x の直後のキーを探索する
  - $O(h)$時間　（h：２分探索木の高さ）

第４章　動的探索問題とデータ構造

4.1 線形リスト上での探索

4.2 ２分探索木

4.3 平衡２分探索木

4.4 動的ハッシュ法

４.３ 平衡２分探索木

- ２分探索木
  - 探索，挿入，削除の平均時間の優れたデータ構造
    - 平均　$O(\log n)$
  - ただし最悪時は　$O(n)$
    - 木のバランスが崩れたとき
      - 木の形は挿入や削除の順で決まる
        例：データを昇順に挿入すると直線状の木

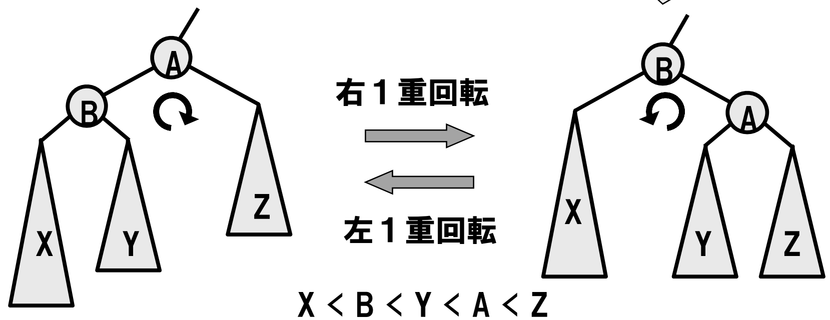
４.３ 平衡２分探索木

- 平衡２分探索木
  - 挿入，削除時の変形時に，木のバランスを適度に保つ
  - 探索，挿入，削除の最悪時間　$O(\log n)$
    - 挿入，削除時のバランス操作の時間を含む
  - ＡＶＬ木，２色木など

バランスを保つための変形操作

- 回転操作
  - ２分探索木条件を保持
  - 効率よく実行可能　$O(1)$時間
  - １重回転，２重回転

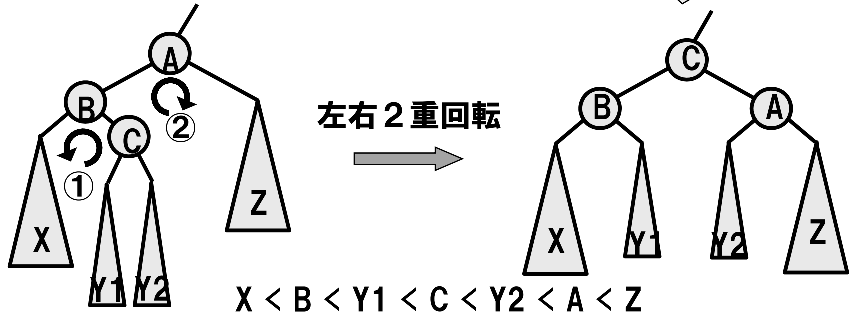
３つの親子関係の変更：$O(1)$時間



バランスを保つための変形操作

- 回転操作
  - ２分探索木条件を保持
  - 効率よく実行可能　$O(1)$時間
  - １重回転，２重回転

５つの親子関係の変更：$O(1)$時間

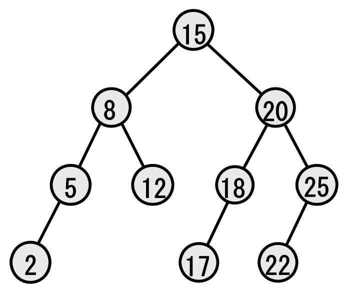


4.3.2 平衡条件

- 回転操作を用いた平衡木
  - ＡＶＬ木
  - ２色木

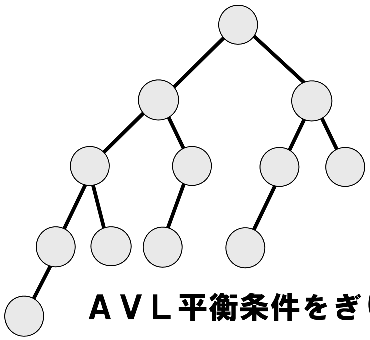
ＡＶＬ木

- ＡＶＬ木平衡条件（以下が葉以外の全頂点で成立）
  - ２つの子をもつ節点
    - 左右の部分木の高さの差≦１
  - １つの子しか持たない節点
    - その子は葉



ＡＶＬ木

- ＡＶＬ木
  - 木の高さは $O(\log n)$
    - ＡＶＬ木平衡条件から保証される
    - $O(\log n)$ 時間の探索，挿入，削除を可能にする



ＡＶＬ平衡条件をぎりぎり満たす例

２色木

- ２色木平衡条件

（RB0）どの内部頂点も２つの子をもつ

（RB1）各節点は，赤または黒のいずれかの色を持つ

（RB2）葉はすべて黒である

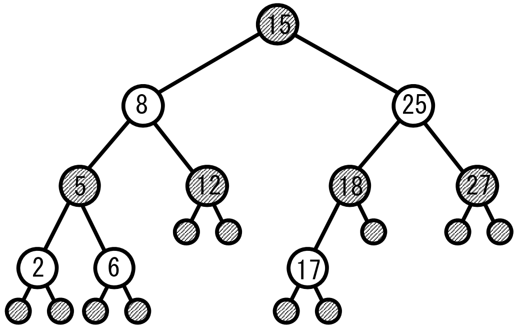
葉はデータ（キー）を保持しない

（RB3）赤節点の子は両方とも黒である

（RB4）根から葉までの全経路は同数の黒節点を含む

２色木

- ２色木の例



２色木平衡条件

（RB0）どの内部頂点も２つの子をもつ
（RB1）各節点は，赤または黒のいずれかの色を持つ
（RB2）葉はすべて黒である（葉はデータを保持しない）
（RB3）赤節点の子は両方とも黒である
（RB4）根から葉までの全経路は同数の黒節点を含む

4.3.3 ２色木の高さ

- n 個のキーを含む２色木の高さ：$O(\log n)$
  - h：根から葉までの各経路の黒節点数
    （根から葉までの各経路は同数の黒節点を含む）
  - 根から葉までの各経路の節点数
    h 以上　$2h$ 以下（∵赤節点の子は黒，葉は黒）
  - ２色木に含まれるキー数
    $2^{h-1}-1$ 以上　$2^{2h-1}-1$ 以下
  - $2^{h-1}-1 \le n \le 2^{2h-1}-1$ より，
    $h = O(\log n)$

4.3.4 ２色木の探索

- ２色木の探索
  - ２色木は２分探索木の一種
  - 探索は２分探索木と同じ方法
  - 最大探索時間：木の高さに比例　$O(\log n)$

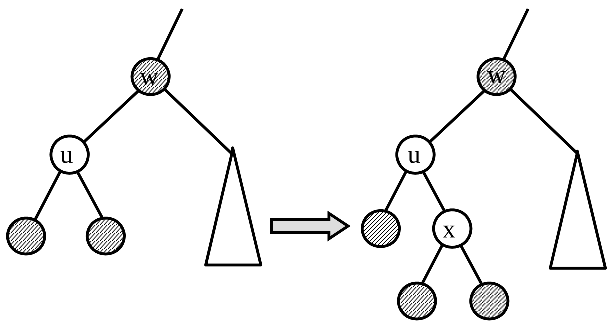
4.3.5 ２色木への挿入

- ２色木への挿入

1. ２分探索木と同様の方法で挿入
   - 空の葉（黒）にキー x を挿入
   - x の子として２つの空の葉（黒）を作成
   - x の色を赤に変色
     （根から葉への経路の黒節点数を保つため）
2. ２色木条件を満たすための変形・変色
   - x の親 u が赤のとき
     （「赤節点の子は黒」という条件を満たすため）

4.3.5 ２色木への挿入

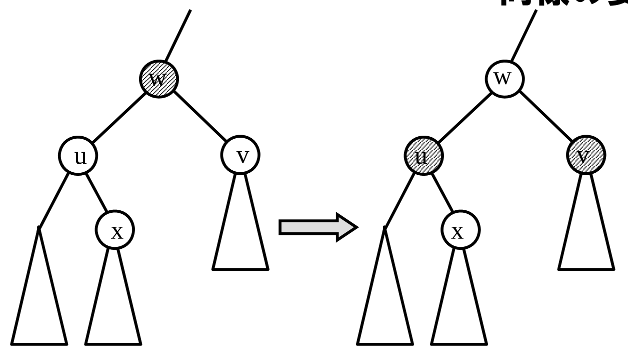
1. ２分探索木と同様の方法で挿入
   - 空の葉（黒）にキー x を挿入
   - x の子として２つの空の葉（黒）を作成
   - x の色を赤に変色



挿入：２色木の変形（１）

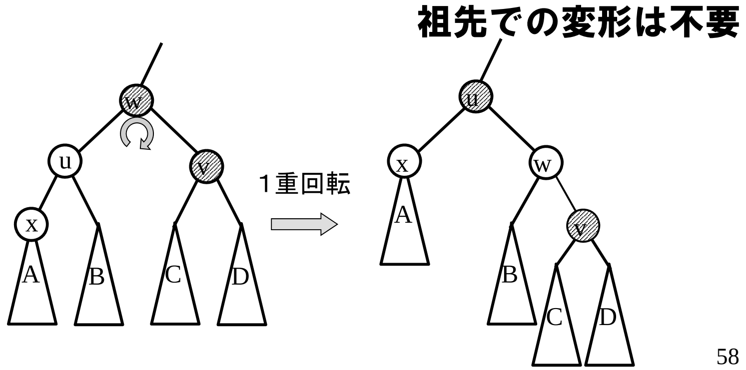
1. v が赤節点の場合
   x ：挿入された節点（赤）
   u：x の親（赤）
   v：u の兄弟

w の親が赤なら
同様の変形



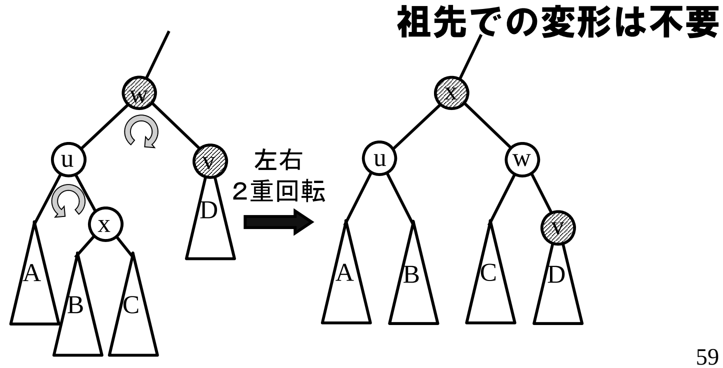
挿入：２色木の変形（２）

2. v が黒節点の場合
   a. x が u の左の子で u が w の左の子，または，
   x が u の右の子で u が w の右の子 の場合



挿入：２色木の変形（３）

2. v が黒節点の場合
   b. x が u の右の子で u が w の左の子，または，
   x が u の左の子で u が w の右の子 の場合

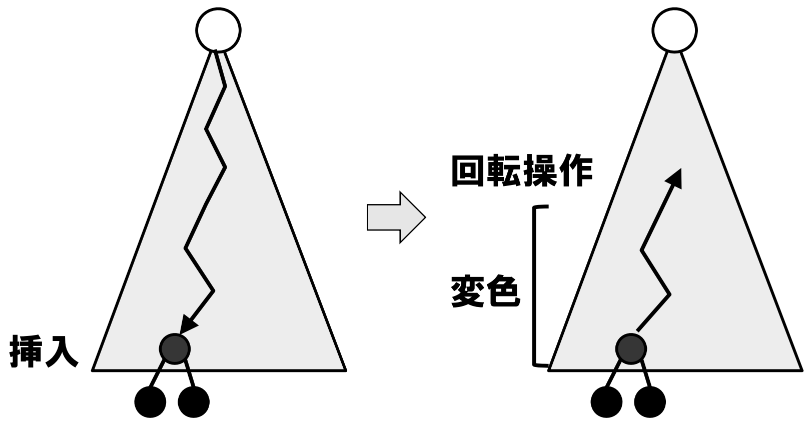


挿入の時間計算量

- 挿入の時間計算量：$O(\log n)$
  - 挿入場所の決定・挿入 $O(\log n)$
  - 変形・変色
    - 挿入した節点から根に向かって
    - 各変形・変色は $O(1)$ 時間
      - 回転操作は高々１回

挿入操作の概観

1. ２分探索木と同様に，空の葉に挿入
2. バランス条件回復のために，変色／変形

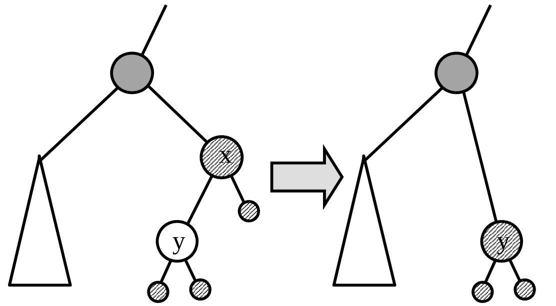


4.3.6 ２色木からの削除

- ２色木からのキーｘの削除
  1. ２分探索木と同様の方法で削除
     ① ｘの２つの子がともに空の葉のとき
        ｘを空の葉に置換
     ② ｘの１つの子のみが空の葉のとき
        ｘを抜き取り，ｘの親と葉でない子ｙを直結
     ③ ｘの２つの子がともに空の葉でないとき
        左部分木の最大キーを節点ｘに移動
        最大キーを保持していた節点ｚを削除
        ｚの削除は①，②いずれかの方法で
  2. ２色木条件を満たすための変形・変色

２色木からの削除（１）

- ２色木からのキー x の削除
  ② x の１つの子のみが空の葉のとき
     x を抜き取り，x の親と葉でない子ｙを直結
  - y は赤節点で２つの空の葉（黒）を持つ
  - x は黒節点

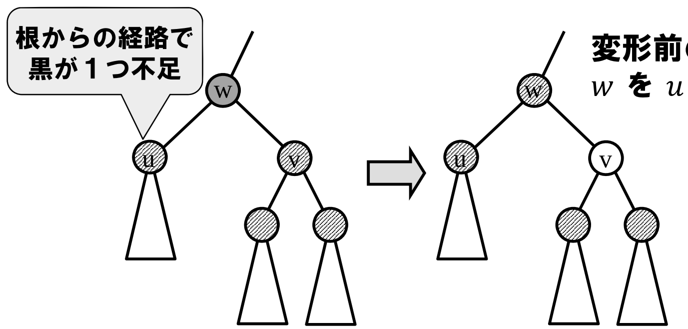


２色木からの削除（２）

- ２色木からのキー x の削除
  ① x の２つの子がともに空の葉のとき
     x を空の葉に置換
  - x が赤節点の場合，問題なし
  - x が黒節点の場合
    - 根から葉までの黒節点数を揃えるために
      木の変形・変色が必要

２色木からの削除（３）

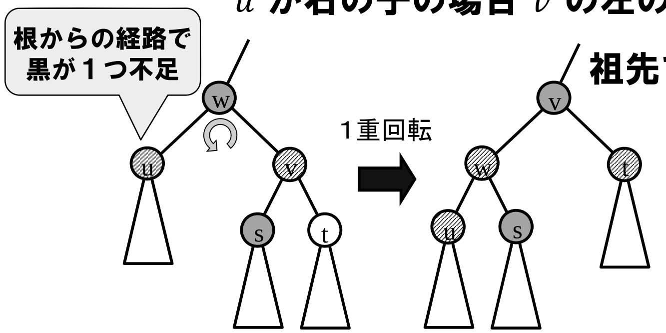
- ２色木からのキー x の削除
  ① x の２つの子がともに空の葉のとき
  - x が黒節点の場合
    u： x に置換された空の葉
    a. u の兄弟 v が黒，vの２つの子がともに黒



変形前の w が黒なら
w を u とみなして変形

２色木からの削除（４）

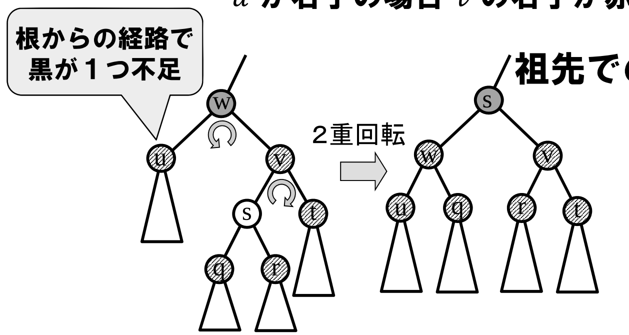
- **２色木からのキー x の削除**
  - **①　x の２つの子がともに空の葉のとき**
    - **–　x が黒節点の場合**
      - b.　**u の兄弟 v が黒,**
        u が左の子の場合 v の右の子が赤，または
        u が右の子の場合 v の左の子が赤



祖先での変形は不要

２色木からの削除（５）

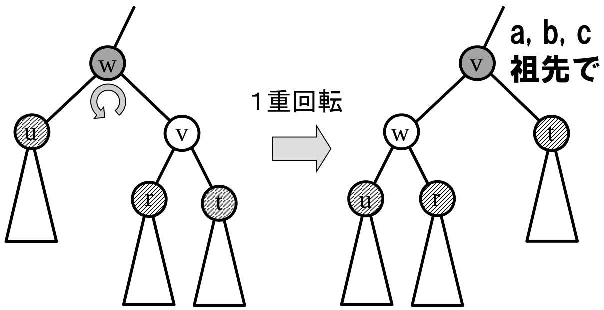
- **２色木からのキー x の削除**
  - **①　x の２つの子がともに空の葉のとき**
    - **–　x が黒節点の場合**
      - c.　**u の兄弟 v が黒,**
        u が左子の場合 vの左子が赤，右子が黒，または
        u が右子の場合 v の右子が赤，左子が黒



祖先での変形は不要

２色木からの削除（６）

- **２色木からのキー x の削除**
  - **①　x の２つの子がともに空の葉のとき**
    - **–　x が黒節点の場合**
      u：x に置換された空の葉
      - d.　**u の兄弟 v が赤**



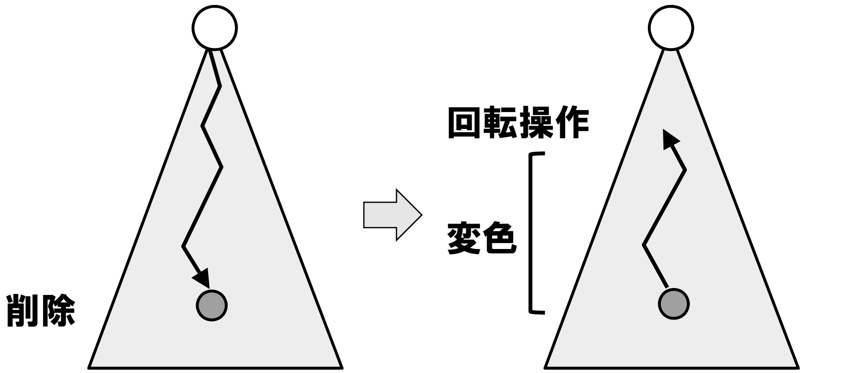
a, b, c の方法で変形
祖先での変形は不要

削除の時間計算量

- **削除の時間計算量：$O(\log n)$**
  - **削除キーの探索・削除 $O(\log n)$**
    - **左部分木中の最大キーの探索を含む**
  - **変形・変色**
    - **削除した節点から根に向かって**
    - **各変形・変色は $O(1)$ 時間**
      - **回転操作**
        - **１重回転：高々２回**
        - **２重回転：高々１回**

削除操作の概観

1. ２分探索木と同様に削除
2. バランス条件回復のために，変色／変形



第４章　動的探索問題とデータ構造

4.1 線形リスト上での探索
4.2 ２分探索木
4.3 平衡２分探索木
4.4 動的ハッシュ法

４.４ 動的ハッシュ法

- ハッシュ法を挿入，削除ができるように拡張
- ハッシュ表への挿入：キー x の挿入
  - データを配列に蓄える手続きと同様

```
j = hash(x); //ハッシュ値を計算
while (htb[j] ≠ 0) //ハッシュ表で空いている場所を探す
   j = (j+1) % m; //次の場所へ移動
htb[j] = x; //最初の空き場所に x を格納
```


ハッシュ表への挿入と探索：例

h(x) ＝x mod 11

0	
1	
2	
3	３
4	１５
5	５
6	２５
7	７
8	
9	２０
10	

挿入：５　７　１５
２０　３　２５

h(x) ＝x mod 11

0	
1	
2	
3	３
4	１５
5	５
6	２５
7	７
8	
9	２０
10	

② ２５の探索　②
③ ６の探索　③
① ２０の探索　①

動的ハッシュ法：削除

- ハッシュ表からのキー x の削除
  1. キー x の探索
  2. キー x の削除
     - キーが格納されていたことを示す印が必要

  （参考）キー x を探索する手続き

```
探索すべきデータ x を入力する;
j = hash(x);
while (htbl[j] ≠ 0 かつ htbl[j] ≠ x)
   j = (j+1) % m; //次の場所へ移動
if htbl[j] = x then j を返して終了;
else -1 を返して終了;
```


ハッシュ表への削除と探索：例

h(x) =x mod 11

0	
1	
2	
3	3
4	15
5	5
6	25
7	7
8	
9	20
10	

５の削除

h(x) =x mod 11

２５の探索

0	
1	
2	
3	3
4	15
5	＊
6	25
7	7
8	
9	20
10	

まとめ
第４章　動的探索問題とデータ構造

4.1 線形リスト上での探索

4.2 ２分探索木

4.3 平衡２分探索木

4.4 動的ハッシュ法

「第2章 探索問題」との違い

新たなデータの挿入，
既にあるデータの削除
を考える

この章の学習目標（振返り）

- 探索問題での挿入・削除とは何か，応用例を用いて説明できる
- 2分探索木，平衡2分探索木とは何か説明できる
- 挿入・削除のアルゴリズムを説明できる
  - 逐次探索，２分探索，2分探索木，2色木，ハッシュ法
- 挿入・削除アルゴリズムの計算時間を説明できる
  - 最悪時だけでなく，平均計算時間も
- 挿入・削除を含めた探索アルゴリズムのプログラムを書ける

データ構造とアルゴリズム　第6, 8, 9回

1. アルゴリズムの重要性
2. 探索問題
3. 基本的なデータ構造
4. 動的探索問題とデータ構造
5. データの整列
6. グラフのアルゴリズム
7. 文字列のアルゴリズム
8. アルゴリズム設計手法

第５章　データの整列

5.1 バブルソート

5.2 セレクションソート

5.3 インサーションソート

5.4 シェルソート

5.5 ヒープソート

5.6 クイックソート

＊　サンプルソート

5.7 マージソート

5.8 ソート問題の計算複雑度

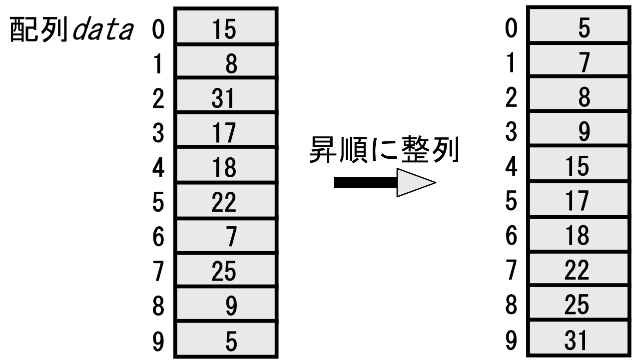
＊　ビンソート，基底法

この章の学習目標

- さまざまな整列アルゴリズムとその計算時間を例を用いて説明できる
  - バブルソート，セレクションソート，インサーションソート，シェルソート，ヒープソート，クイックソート，サンプルソート，マージソート，ビンソート，基底法
- 整列問題の時間計算量の下界について，理由とともに説明できる
- 整列アルゴリズムのプログラムを書ける

整列（ソート）問題の定義

- 整列（ソート）問題の定義
  - 与えられたデータを昇順（小さい順）に並べ替え
  - n：データ数
  - データ　$data[0 \, .. \, n-1]$



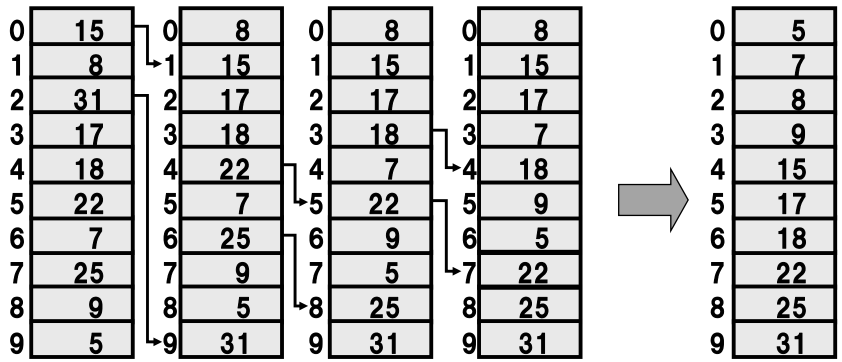
5.1 バブルソート

- バブルソート
  - 逆順ペアの交換
    - 逆順ペア　$data[i] > data[i+1]$

```
for (k=1 to n-1)
   for (i=0 to n-k-1)
      if (data[i]>data[i+1]) data[i] とdata[i+1] を交換
```


5.1 バブルソート

- 実行例



5.1 バブルソート

- 計算時間　常に $O(n^2)$
  - 比較回数：$n(n-1)/2$
  - 交換回数：0 〜 $n(n-1)/2$

    整列済みのとき 0回

    逆順のとき $n(n-1)/2$ 回

```
for (k=1 to n-1)
   for (i=0 to n-k-1)
      if (data[i]>data[i+1]) data[i] とdata[i+1] を交換
```


第５章　データの整列

5.1 バブルソート

5.2 セレクションソート

5.3 インサーションソート

5.4 シェルソート

5.5 ヒープソート

5.6 クイックソート

＊　サンプルソート

5.7 マージソート

5.8 ソート問題の計算複雑度

＊　ビンソート，基底法

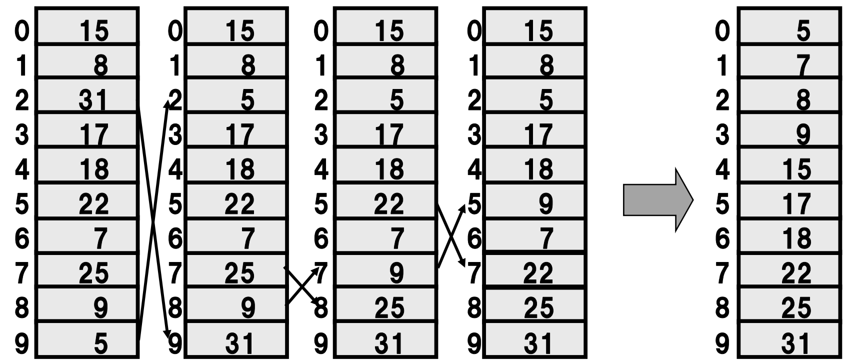
5.2 セレクションソート

● 最大値から大きい順に選択

```
for (k=n-1 to 1 step -1) {
    data[0]～data[k]の最大値data[m]を求める
    data[m]とdata[k]を交換;
}
```


5.2 セレクションソート

● 実行例



5.2 セレクションソート

● 計算時間　常に $O(n^2)$

- 比較回数：$n(n-1)/2$
- 交換回数：0 ～ $n-1$

整列済みのとき 0回

```
for (k=n-1 to 1 step -1) {
    data[0]～data[k]の最大値data[m]を求める
    data[m]とdata[k]を交換;
}
```


第５章　データの整列

5.1 バブルソート
5.2 セレクションソート
5.3 インサーションソート
5.4 シェルソート
5.5 ヒープソート
5.6 クイックソート
＊　サンプルソート
5.7 マージソート
5.8 ソート問題の計算複雑度
＊　ビンソート，基底法

5.3 インサーションソート

● 整列済みの範囲を配列の前から順に拡大

```
for (i=1 to n-1) {
    x=data[i]; j=i;
    while ((j>0) && (data[j-1]>x)) {
        data[j-1]をdata[j]に移動し, j=j-1とする
    }
    data[j]=x;
}
```

テキストの $(j-1)>0$ は誤り

5.3 インサーションソート

● 実行例

	入力	data[0..4]が整列済み	data[0..5]が整列済み
0	15	8	8
1	8	15	15
2	31	17	17
3	17	18	18
4	18	31	22
5	22	22	31
6	7	7	7
7	25	25	25
8	9	9	9
9	5	5	5

22を適当な位置に挿入

5.3 インサーションソート

● 計算時間　最大 $O(n^2)$

　　　　　　最小 $O(n)$

- 比較回数：$n-1$ ～ $n(n-1)/2$
  - 整列済みのとき $n-1$ 回
  - 逆順のとき $n(n-1)/2$ 回

● 重要な特長

- ほぼソート済の入力は $O(n)$ 時間で整列できる

第５章　データの整列

5.1 バブルソート

5.2 セレクションソート

5.3 インサーションソート

5.4 シェルソート

5.5 ヒープソート

5.6 クイックソート

＊　サンプルソート

5.7 マージソート

5.8 ソート問題の計算複雑度

＊　ビンソート，基底法

5.4 シェルソート

- これまでに紹介した整列アルゴリズムの計算時間
  - バブルソート，セレクションソート　常に $O(n^2)$
  - インサーションソート　最大 $O(n^2)$　　最小 $O(n)$
    - 入力がほぼ整列済みのとき高速
- シェルソートの計算時間
  - パラメタによって異なる．解析は難しい
  - あるパラメタでは
    - 最大 $O(n^{4/3})$，平均 $O(n^{5/4})$
    - ＊最大 $O(n \log^2 n)$ のパラメタもある

5.4 シェルソート

- 方針（インサーションソートの応用）

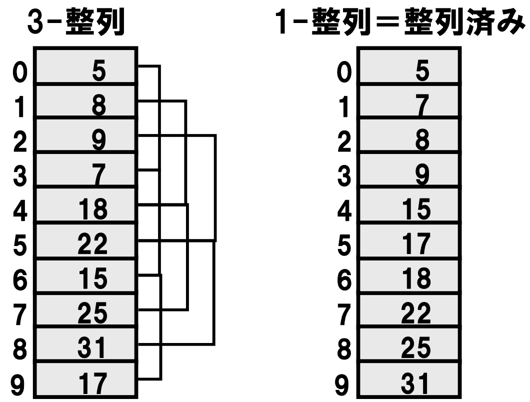
1. 大ざっぱにソート（インサーションソート）
   - 部分列に分割したソートを繰り返す
     - h 個の部分列に分割した場合
       - １つの部分列の長さ　n/h
       - １つの部分列の整列　$O((n/h)^2)$
       - h 個の部分列の整列　$O(n^2/h)$
2. インサーションソートでソート
   ほぼソート済みなので高速

シェルソート：h-整列

- 方針（インサーションソートの応用）

1. 大ざっぱにソート（インサーションソート）
   h-整列
   - 各 i について，$data[i] \le data[i+h]$
2. インサーションソートでソート

シェルソート：h-整列



5.4 シェルソート

- シェルソート
  - h_1-整列 → h_2-整列 → ・・・ → 1-整列
    （$h_1 > h_2 > \cdots > 1$）
  - 各整列は，インサーションソートを適用
  - h-整列
    - 各 i について，$data[i] \le data[i+h]$
    - 1-整列＝整列済み

5.4 シェルソート

- 実行例
  - $h_1 = 5,\ h_2 = 3,\ h_3 = 1$

	入力	5-整列	3-整列	1-整列
0	15	15	8	5
1	8	7	5	7
2	31	25	17	8
3	17	9	9	9
4	18	5	7	15
5	22	22	22	17
6	7	8	15	18
7	25	31	31	22
8	9	17	25	25
9	5	18	18	31

5.4 シェルソート

- 計算時間
  - $h_1,\ h_2, \dots, h_m$ の選び方に依存
    - うまく選べば
      - 最大　$O(n^{4/3})$　（$O(n \log^2 n)$ も可能）
      - 平均　$O(n^{5/4})$
    - よく利用される系列
      - $2^k - 1$　1, 3, 7, 15, 31, 63, …
      - $(3^k - 1)/2$　1, 4, 13, 40, 121, 364, …
    - プログラムが簡単で，かなり高速

第５章　データの整列

5.1 バブルソート

5.2 セレクションソート

5.3 インサーションソート

5.4 シェルソート

5.5 ヒープソート

5.6 クイックソート

＊　サンプルソート

5.7 マージソート

5.8 ソート問題の計算複雑度

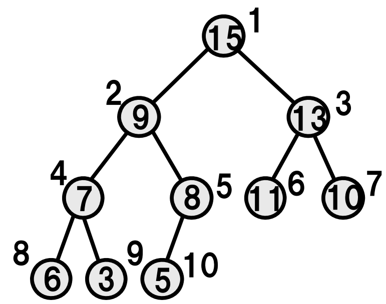
＊　ビンソート，基底法

5.5 ヒープソート

- ヒープを利用したソート
- ヒープ（第３章で学習済）
  - ２分木を配列 $data[1..n]$ で実現（n：データ数）
    - $data[1]$：根
    - $data[k]$ の左の子：$data[2k]$
    - $data[k]$ の右の子：$data[2k+1]$
  - 親のデータ $\geq$ 子のデータ
  - データの格納　$O(\log n)$ 時間
  - 最大データ（根のデータ）の取出し　$O(\log n)$時間

ヒープ

- ヒープの例



1	15
2	9
3	13
4	7
5	8
6	11
7	10
8	6
9	3
10	5

5.5 ヒープソート

- ヒープソート
  1. ヒープを構成;　　　　　　$O(n)$ 時間
  2. for（k=n−1; k>0; k−−）{
     data [1] と data [k+1] を交換;　　$O(1)$ 時間
     /* data [1] は data [1..k+1] の最大値 */
     data [1..k] でヒープを再構成　　$O(\log n)$ 時間
     }

まずはこちらから

- 計算時間　最大 $O(n \log n)$

5.5 ヒープソート

- ヒープ構成後の動作

アルゴリズム 5.5 の後半

```
for (k=n−1; k>0; k−−) {
    x=data [k+1] ; data [k+1] =data [1] ; i=1; j=2;
    while (j<=k) {
        if (j<k && data [j+1] >data [j] ) j=j+1;
        if (x<data [j] ) {
            data [i] =data [j] ; i=j; j=2*i;
        } else break;
    }
    data [i] =x;
}
```

テキスト $k-$ は誤り

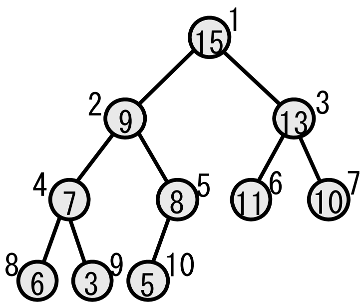
$data[1]$ と $data[k+1]$ を交換

左右の子の大きい方が j

親が子より小さければ入替

5.5 ヒープソート

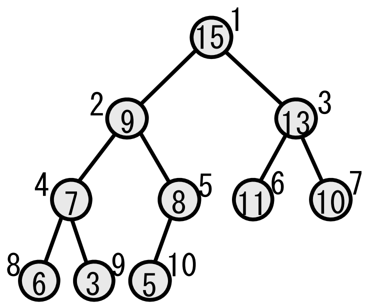
● ヒープ構成後の動作



1	15
2	9
3	13
4	7
5	8
6	11
7	10
8	6
9	3
10	5

5.5 ヒープソート

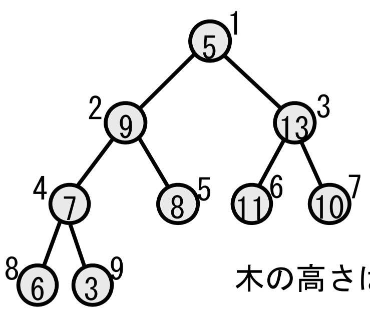
● ヒープ構成後の動作



1	15
2	9
3	13
4	7
5	8
6	11
7	10
8	6
9	3
10	5

5.5 ヒープソート

● ヒープ構成後の動作



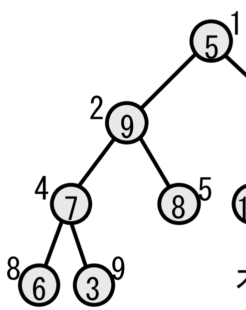
木の高さはlog *n*

1	5
2	9
3	13
4	7
5	8
6	11
7	10
8	6
9	3
10	15

5.5 ヒープソート

● ヒープ構成後の動作

親＜子なら，
大きい方の子と交換

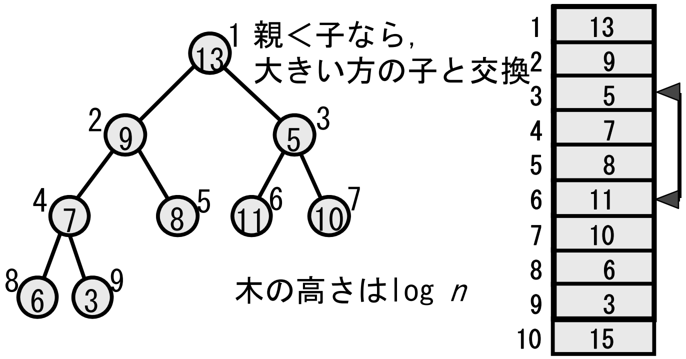


木の高さはlog *n*

1	5
2	9
3	13
4	7
5	8
6	11
7	10
8	6
9	3
10	15

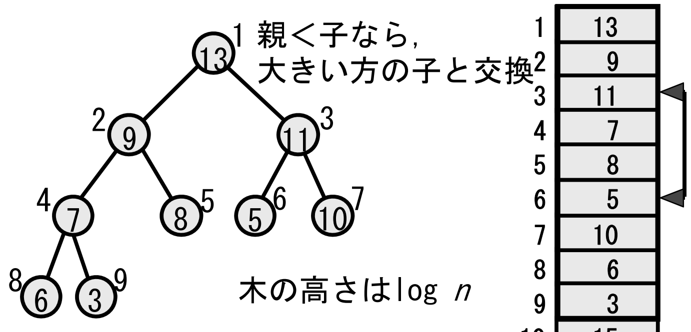
5.5 ヒープソート

- ヒープ構成後の動作



5.5 ヒープソート

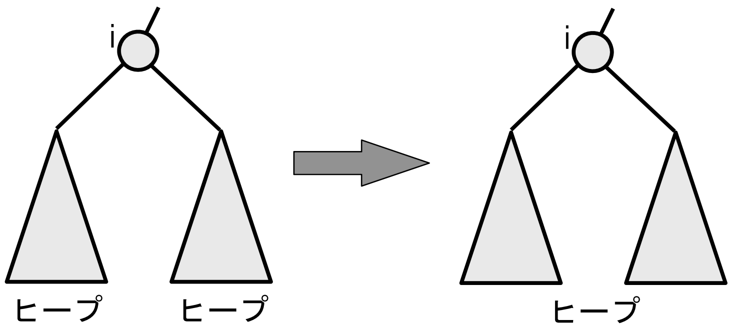
- ヒープ構成後の動作



ヒープの再構成 $O(\log n)$

5.5 ヒープソート

- ヒープの構成（初期化） $O(n)$ 時間
  - 葉に近い部分からヒープを構成



5.5 ヒープソート

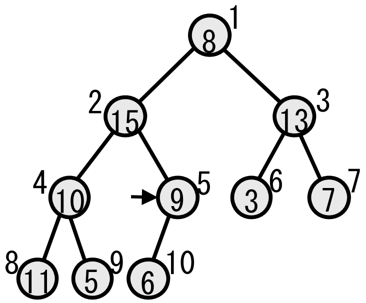
- ヒープの構成（初期化） $O(n)$ 時間
  - 葉に近い部分からヒープを構成

アルゴリズム 5.5 の前半

```
for (k=n/2; k>0; k=k-1) {          data[k] は子を持つ
    i=k; j=2*i; x=data[i];
    while (j<=i) {                 左右の子の大きい方が j
        if (j<n && data[j+1]>data[j]) j=j+1;
        if (x<data[j]) {           親が子より小さければ入替
            data[i]=data[j]; i=j; j=2*i;
        } else break;
    }
    data[i]=x;
}
```

5.5 ヒープソート

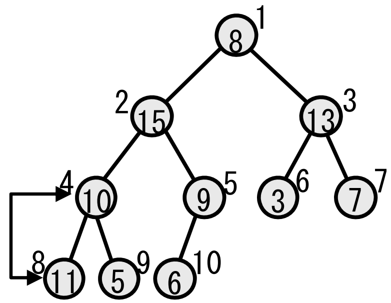
- ヒープの構成（初期化）　$O(n)$ 時間
  - 葉に近い部分からヒープを構成



1	8
2	15
3	13
4	10
5	9
6	3
7	7
8	11
9	5
10	6

5.5 ヒープソート

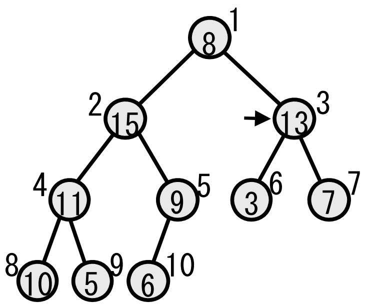
- ヒープの構成（初期化）　$O(n)$ 時間
  - 葉に近い部分からヒープを構成



1	8
2	15
3	13
4	10
5	9
6	3
7	7
8	11
9	5
10	6

5.5 ヒープソート

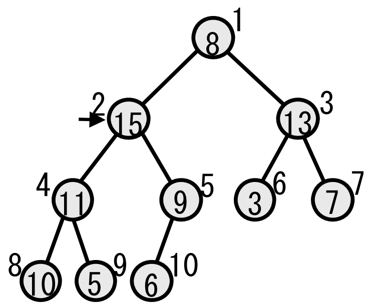
- ヒープの構成（初期化）　$O(n)$ 時間
  - 葉に近い部分からヒープを構成



1	8
2	15
3	13
4	11
5	9
6	3
7	7
8	10
9	5
10	6

5.5 ヒープソート

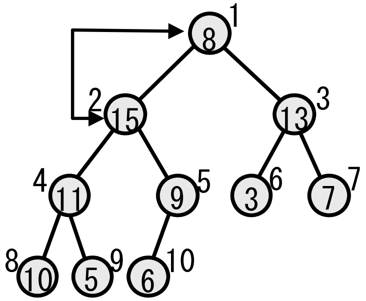
- ヒープの構成（初期化）　$O(n)$ 時間
  - 葉に近い部分からヒープを構成



1	8
2	15
3	13
4	11
5	9
6	3
7	7
8	10
9	5
10	6

5.5 ヒープソート

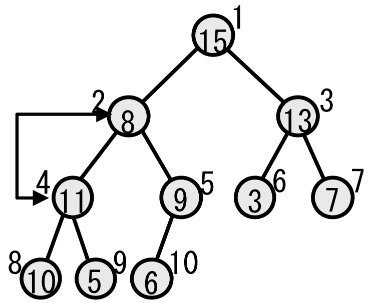
- ヒープの構成（初期化）　$O(n)$ 時間
  - 葉に近い部分からヒープを構成



1	8
2	15
3	13
4	11
5	9
6	3
7	7
8	10
9	5
10	6

5.5 ヒープソート

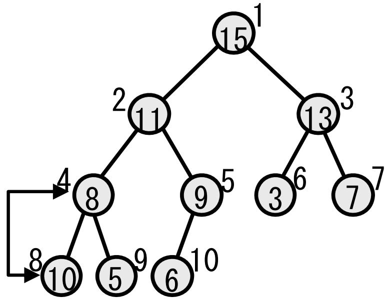
- ヒープの構成（初期化）　$O(n)$ 時間
  - 葉に近い部分からヒープを構成



1	15
2	8
3	13
4	11
5	9
6	3
7	7
8	10
9	5
10	6

5.5 ヒープソート

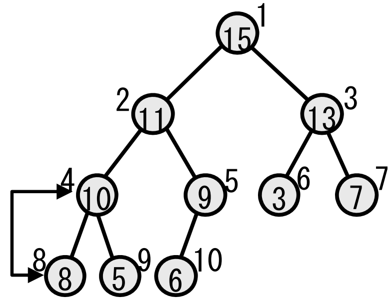
- ヒープの構成（初期化）　$O(n)$ 時間
  - 葉に近い部分からヒープを構成



1	15
2	11
3	13
4	8
5	9
6	3
7	7
8	10
9	5
10	6

5.5 ヒープソート

- ヒープの構成（初期化）　$O(n)$ 時間
  - 葉に近い部分からヒープを構成

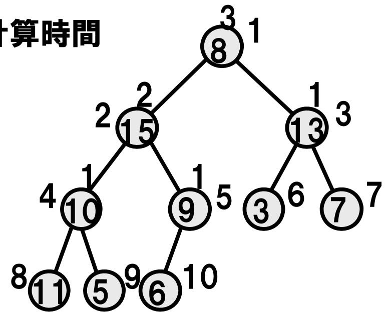


1	15
2	11
3	13
4	10
5	9
6	3
7	7
8	8
9	5
10	6

5.5 ヒープソート

- ヒープの構成（初期化）　$O(n)$ 時間
  - 葉に近い部分からヒープを構成

計算時間



$(\log \boldsymbol{n} - \boldsymbol{1}) +$

$\boldsymbol{2}(\log \boldsymbol{n} - \boldsymbol{2}) +$

$\boldsymbol{4}(\log \boldsymbol{n} - \boldsymbol{3}) +$

5.5 ヒープソート

- ヒープの構成（初期化）　$O(n)$ 時間

$$(\log n - 1) + (\log n - 2) \times 2 + (\log n - 3) \times 4 + \cdots$$
$$+ 2 \times (n/8) + 1 \times (n/4)$$

$k = \log n$　とすると，

$$(k-1) + (k-2) \times 2 + (k-3) \times 2^2 + \cdots$$
$$+\ 2 \times 2^{k-3} + 1 \times 2^{k-2}$$
$$= 2^k - (k+1)$$

従って，$O(n)$

5.5 ヒープソート

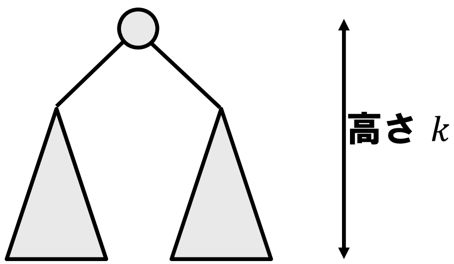
- ヒープの構成（初期化）　$O(n)$ 時間（別解）

$T(n)$ ：n 頂点の場合の計算時間

$T\left(2^k - 1\right) = 2T\left(2^{k-1} - 1\right) + (k-1)$

$T(1) = 0$　　（$k = 1$）

– $T\left(2^k - 1\right) = 2^k - (k+1)$ を帰納法で証明



5.5 ヒープソート

- ヒープソート
  1. ヒープを構成;　　$O(n)$ 時間
  2. for (k=n-1; k>0; k--) {
     data[1] と data[k+1] を交換;　　$O(1)$ 時間
     /* data[1] は data[1..k+1] の最大値 */
     data[1..k] でヒープを再構成　　$O(\log n)$ 時間
     }
- 計算時間　最大 $O(n \log n)$

第５章　データの整列

5.1 バブルソート

5.2 セレクションソート

5.3 インサーションソート

5.4 シェルソート

5.5 ヒープソート

5.6 クイックソート

＊　サンプルソート

5.7 マージソート

5.8 ソート問題の計算複雑度

＊　ビンソート，基底法

5.6 クイックソート

- 計算時間
  - 平均　$O(n \log n)$
  - 最大　$O(n^2)$
    - 基本操作が簡単
    - 実際に高速でもっともよく使用されている

クイックソートの考え方

- S:データ集合

1. $|S| \leq 1$ ならば，S を返して終了
2. S から任意に一つの要素 x を選ぶ（基準値）
3. S を，x より小さい要素の集合 S_1，
   x に等しい要素の集合 S_2，
   x より大きい要素の集合 S_3　　に分割
4. S_1，S_2 それぞれを再帰的にソート
5. S_1，S_2，S_3 のソート列を連結したものを返す

分割統治法（divide and conquer）

クイックソートの考え方

	基準値を選択		分割		独立に整列
0	15		15		5
1	8		8	基準値	7
2	31		5	より	8
3	17		17	小さい	9
4	18	←	9		15
5	22		7		17
6	7		18	基準値	18
7	25		25	基準値	22
8	9		22	より	25
9	5		31	大きい	31

クイックソートの計算時間

- S：データ集合
  1. $|S| \leq 1$ ならば，S を返して終了 $O(1)$
  2. S から任意に一つの要素 x を選ぶ（基準値） $O(1)$
  3. S を，x より小さい要素の集合 S_1，
     x に等しい要素の集合 S_2， $O(n)$
     x より大きい要素の集合 S_3 に分割
  4. S_1，S_2 それぞれを再帰的にソート 再帰の段数は基準値に依存
  5. S_1，S_2，S_3 のソート列を連結したものを返す

クイックソートの計算時間

- 再帰の段数は基準値に依存
  - 毎回、最小値（最大値）を選ぶと
    - 再帰の段数 $O(n)$
    - 計算時間 $O(n^2)$ 最大
  - 毎回、中央値を選ぶと
    - 再帰の段数 $O(\log n)$
    - 計算時間 $O(n \log n)$ 最小，平均
- 基準値の選び方（よく利用する方法）
  - $data[mid]$ ：$mid = (low + high)/2$
  - $data[low]$，$data[mid]$，$data[high]$ の中央値

分割の仕方

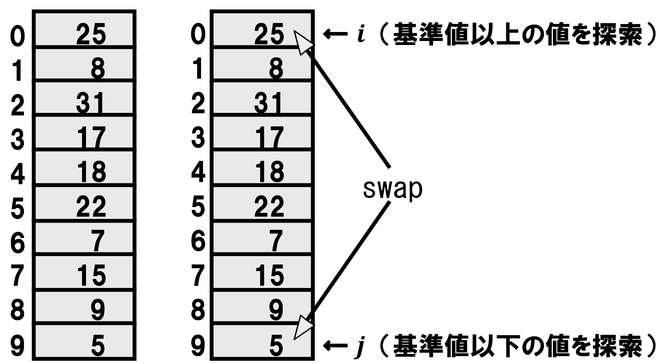
- S を，S_1，S_2，S_3 に分割

  ↓ 簡単化のため

  S を，S_1，S_2 に分割
  - S_1：基準値 x 以下のデータの集合
  - S_2：基準値 x 以上のデータの集合

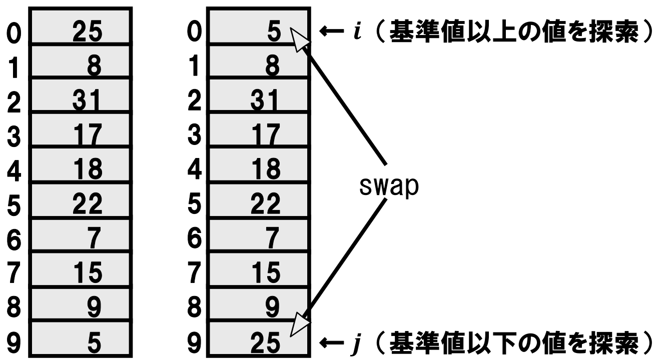
分割の仕方

- 計算時間 $O(n)$



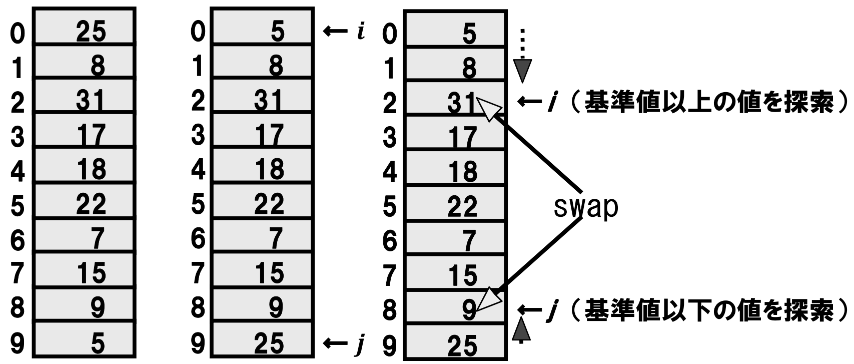
分割の仕方

● 計算時間　$O(n)$



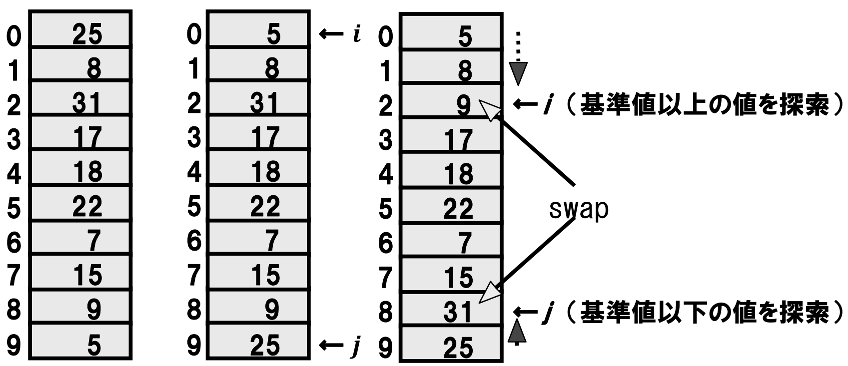
分割の仕方

● 計算時間　$O(n)$



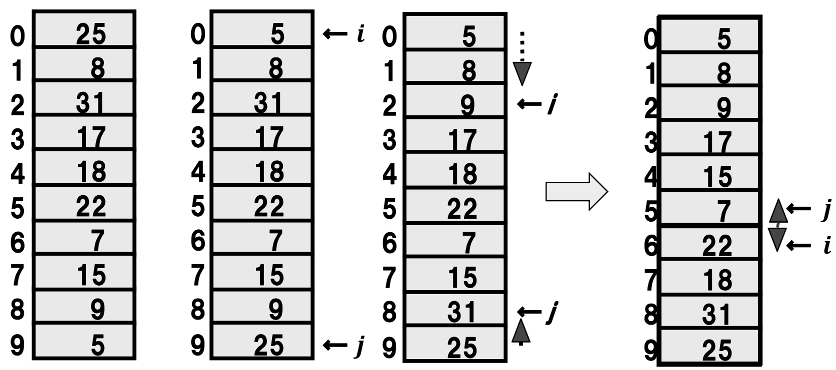
分割の仕方

● 計算時間　$O(n)$



分割の仕方

● 計算時間　$O(n)$



クイックソートの平均計算時間

- 最大計算時間：$O(n^2)$
- 平均計算時間：$O(n\ log\ n)$
  - 仮定：各要素を等確率で基準値とする
  - 平均比較回数 $C(n)$

$$C(n) = \Sigma_{k=1}^{n}(n+2+C(k-1)+C(n-k))/n$$
$$= n + 2 + \ 2\Sigma_{\mathrm{k=0}}^{\mathrm{n-1}}C(k)/n$$
$$= 1.38\, n \log_2 n + O(n)$$

第５章　データの整列

5.1 バブルソート
5.2 セレクションソート
5.3 インサーションソート
5.4 シェルソート
5.5 ヒープソート
5.6 クイックソート
＊　サンプルソート
5.7 マージソート
5.8 ソート問題の計算複雑度
＊　ビンソート，基底法

＊サンプルソート

- クイックソートの拡張
  - 実際的には，
    クイックソート（３値の中央値）と同程度の速度
    ３値の中央値：先頭，中央，末尾の中央値
  - より大きい n に対し，
    クイックソート（３値の中央値）より高速
- 平均比較回数（主要項）
  - 単純クイックソート $1.38n \log_2 n$
  - クイックソート（３値の中央値） $1.19n \log_2 n$
  - サンプルソート $1n \log_2 n$

＊サンプルソート

- クイックソート
  - １つの基準値を使って分割
- サンプルソート
  - 複数の基準値（サンプル）を使って分割

＊サンプルソート

1. $n/\log n$ 個のサンプルを抽出 $O(n/\log n)$
2. $n/\log n$ 個のサンプルを整列（クイックソート） 平均 $O(n)$
3. サンプルで，入力を $n/\log n+1$ 個のグループに分割
   比較回数 $n\log_2 n$
4. 各グループを整列 平均 $O(n\log\log n)$
   - グループ数 $n/\log n$
   - 各グループのデータ数 平均 $\log n$
   - 各グループの整列（クイックソート） 平均 $\log n\log\log n$

＊サンプルソート

● 実行例

入力

22	8	31	17	18	15	7	25	9	5	10	3	27	16	19	20

サンプル選択

22	8	31	17	18	15	7	25	9	5	10	3	27	16	19	20

サンプルでグループに分割

7	5	3	8	15	9	10	16	17	18	19	20	22	25	27	31

各グループを独立に整列

3	5	7	8	9	10	15	16	17	18	19	20	22	25	27	31

＊サンプルソート

● クイックソートの拡張
- 実際的には，
  クイックソート（３値の中央値）と同程度の速度
  ３値の中央値：先頭，中央，末尾の中央値
- より大きい n に対し，
  クイックソート（３値の中央値）より高速

● 平均比較回数（主要項）
- 単純クイックソート $1.38n\log_2 n$
- クイックソート（３値の中央値） $1.19n\log_2 n$
- サンプルソート $1n\log_2 n$

第５章　データの整列

5.1 バブルソート
5.2 セレクションソート
5.3 インサーションソート
5.4 シェルソート
5.5 ヒープソート
5.6 クイックソート
＊　サンプルソート
5.7 マージソート
5.8 ソート問題の計算複雑度
＊　ビンソート，基底法

5.7 マージソート

- 計算時間
  - 最大　$O(n \log n)$　最適
  - 実際的にはヒープソートより高速
  - 作業用記憶域：大きさ n の配列

5.7 マージソート

- マージ（併合）
  - 整列された２本の列（長さ：m, k）

　　↓　マージ　$O(m+k)$ 時間

  - 整列された１本の列（長さ：$m+k$）

16	23	29	33	41	57	77	91

11	31	55	63	66	71	82	93

マージ

11	16	23	29	31	33	41	55	57	63	66	71	77	82	91	93

5.7 マージソート

- ボトムアップ式マージソート：最大 $O(n \log n)$ 時間
  - （長さ１の整列された列）$\times n$　：入力

　　↓マージ　$O(n)$ 時間

  - （長さ２の整列された列）$\times n/2$

　　↓マージ　$O(n)$ 時間

  - （長さ４の整列された列）$\times n/4$

　　↓マージ　$O(n)$ 時間

　　・・・

　　↓マージ　$O(n)$ 時間

  - （長さ n の整列された列）$\times 1$

5.7 マージソート

- 実行例

23	57	41	91	33	16	29	77	82	63	93	11	55	71	31	66

（長さ１の整列された列）$\times n$ ：入力

マージ

23	57	41	91	16	33	29	77	63	82	11	93	55	71	31	66

（長さ２の整列された列）$\times n/2$

マージ

23	41	57	91	16	29	33	77	11	63	82	93	31	55	66	71

（長さ４の整列された列）$\times n/4$

5.7 マージソート

- 分割統治法に基づくマージソート
  - ソートすべき区間を $[low, high]$ とする
  1. 列の長さが十分に短いときは，別の方法で整列
  2. $mid = (low + high)/2$
  3. ２つの部分区間 $[low, mid]$ と $[mid+1, high]$ に分割
  4. 各部分区間を再帰的に整列し，２つの列をマージ
- 計算時間
  - $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$ より，$T(n) = O(n \log n)$

ソートアルゴリズムの時間計算量

- バブルソート，セレクションソート　　常に$O(n^2)$
- インサーションソート
  最大，平均 $O(n^2)$，最小 $O(n)$
- シェルソート　　最大 $O(n^{4/3})$，平均 $O(n^{5/4})$
- ヒープソート　　最大 $O(n \log n)$　最適
- クイックソート，サンプルソート
  最大 $O(n^2)$，平均 $O(n \log n)$
- マージソート　　最大 $O(n \log n)$　最適

第５章　データの整列

5.1 バブルソート
5.2 セレクションソート
5.3 インサーションソート
5.4 シェルソート
5.5 ヒープソート
5.6 クイックソート
＊　サンプルソート
5.7 マージソート
5.8 ソート問題の計算複雑度
＊　ビンソート，基底法

5.8 ソート問題の計算複雑度

かかい ⟷ 上界
げかい ⟷ 天界

- 最大計算時間の下界（かかい）　$\Omega(n \log n)$
  - 整列アルゴリズムの最大時間計算量は，
    少なくとも $n \log n$ に比例
  - 最大時間計算量 $O(n \log n)$ の整列アルゴリズム
    - 最大時間計算量に関して最適
    - ヒープソート，マージソート
  - アルゴリズムへの制限：比較アルゴリズム
    - データの大小比較により，次の動作を決定

Ω記法（オメガ記法）　復習 p.29

$\boldsymbol{\Omega(f(n))}$　　オメガ $f(n)$，ビッグオメガ $f(n)$

スモールオメガ $\omega(f(n))$ もある

ある正定数 n_0, c が存在し，
$n \geq n_0$ を満たすすべての n について，
$\boldsymbol{g(n) \geq c \cdot f(n)}$ を満たす関数 $g(n)$ の集合

- $\boldsymbol{g(n) \in \Omega(f(n))}$ を $\boldsymbol{g(n) = \Omega(f(n))}$ と書くことも多い
- n が十分大きい場合の $g(n)$ の下界
- $an + b$ を表すのに正しいのはどれ？

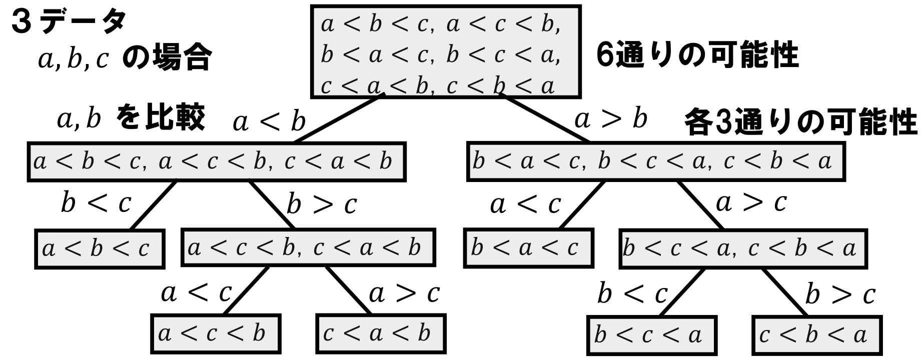
$\Omega(\log n)$　　$\Omega(\sqrt{n})$　　$\Omega(n)$

$\Omega(5n)$　　$\cancel{\Omega(n^2)}$　　$\Omega(n + \sqrt{n})$

なるべく小さく簡単な関数で表す　$\boldsymbol{\Omega(n)}$

5.8 ソート問題の計算複雑度

- 最大計算時間の下界　$\Omega(n \log n)$
  - 決定木による証明
    - ●決定木：整列の過程を表す２分木



5.8 ソート問題の計算複雑度

- 最大計算時間の下界　$\boldsymbol{\Omega(n \log n)}$
  - 決定木による証明
    - ●決定木：整列の過程を表す２分木
  - 葉の数 $n!$
    - → 木の高さ（計算時間）$\geq \log n! \approx n \log n$
    - ●Stirling の公式
      - $-n! = (2\pi n)^{1/2} n^n e^{-n}$

ソートアルゴリズムの時間計算量

- バブルソート，セレクションソート　　常に$O(n^2)$
- インサーションソート
  最大，平均 $O(n^2)$，最小 $O(n)$
- シェルソート　　最大 $O(n^{4/3})$，平均 $O(n^{5/4})$
- ヒープソート　　最大 $O(n \log n)$　最適
- クイックソート，サンプルソート
  最大 $O(n^2)$，平均 $O(n \log n)$
- マージソート　　最大 $O(n \log n)$　最適
- 最大計算時間の下界（比較ｱﾙｺﾞﾘｽﾞﾑ）　$\Omega(n \log n)$

第５章　データの整列

5.1 バブルソート

5.2 セレクションソート

5.3 インサーションソート

5.4 シェルソート

5.5 ヒープソート

5.6 クイックソート

＊　サンプルソート

5.7 マージソート

5.8 ソート問題の計算複雑度

＊　ビンソート，基底法

＊比較以外の方法による整列

- 整列の最大時間計算量の下界　$\Omega(n \log n)$
  - アルゴリズムへの制限：比較アルゴリズム
    - データの大小比較により，次の動作を決定
- この制限がなければ高速化可能か？
  - ビンソート，基底法　最大時間計算量　$O(n)$

＊比較以外の方法による整列

- ビンソート（バケツソート）
  - 最大時間計算量 $O(n)$
  - 基本的アイデア
    - データを 1〜m の整数に制限
    - 値 i 以下のデータの数を $count[i]$ に求める
    - 値 i のデータは
      $count[i-1]+1$〜$count[i]$ 番目のデータ

＊ビンソート（バケツソート）

データ：1〜8

	入力	データiの数	i以下の数	結果
1	3	2	2	1
2	7	1	3	1
3	1	3	6	2
4	1	0	6	3
5	3	0	6	3
6	6	1	7	3
7	8	2	9	6
8	7	1	10	7
9	2			7
10	3			8

＊ビンソート（バケツソート）

データ：1～8

	入力	データiの数	i 以下の数	結果
1	3	2	2	
2	7	1	3	
3	1	3	6	
4	1	0	6	
5	3	0	6	
6	6	1	7	3
7	8	2	9	
8	7	1	10	
9	2			
10	3			

＊ビンソート（バケツソート）

データ：1～8

	入力	データ iの数	i以下の数	結果
1	3	2	2	
2	7	1	3	
3	1	3	5	
4	1	0	6	
5	3	0	6	
6	6	1	7	3
7	8	2	9	
8	7	1	10	
9	2			
10	3			

＊ビンソート（バケツソート）

データ：1～8

	入力	データ iの数	i以下の数	結果
1	3	2	2	
2	7	1	3	
3	1	3	5	2
4	1	0	6	
5	3	0	6	
6	6	1	7	3
7	8	2	9	
8	7	1	10	
9	2			
10	3			

＊ビンソート（バケツソート）

データ：1～8

	入力	データ iの数	i以下の数	結果
1	3	2	2	
2	7	1	2	
3	1	3	5	2
4	1	0	6	
5	3	0	6	
6	6	1	7	3
7	8	2	9	
8	7	1	10	
9	2			
10	3			

＊ビンソート（バケツソート）

データ：1〜8

	入力	データ iの数	i以下の数	結果
1	3	2	2	
2	7	1	2	
3	1	3	5	2
4	1	0	6	
5	3	0	6	
6	6	1	7	3
7	8	2	9	
8	7	1	10	
9	2			7
10	3			

＊ビンソート（バケツソート）

データ：1〜8

	入力	データ iの数	i以下の数	結果
1	3	2	2	
2	7	1	2	
3	1	3	5	2
4	1	0	6	
5	3	0	6	
6	6	1	7	3
7	8	2	8	
8	7	1	10	
9	2			7
10	3			

＊比較以外の方法による整列

- ビンソート（バケツソート）
  - 最大時間計算量 $O(n)$
  - 基本的アイデア
    - データを 1〜m の整数に制限
    - 値 i 以下のデータの数を $count[i]$ に求める
    - 値 i のデータは
      $count[i-1]+1$〜$count[i]$ 番目のデータ
  - 時間計算量
    - $O(n+m)$
    - m が定数なら，$O(n)$

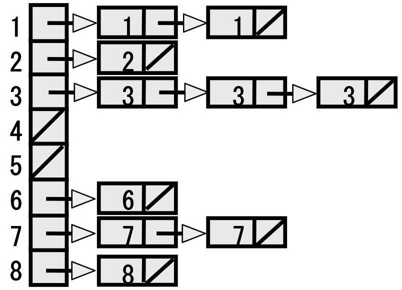
＊比較以外の方法による整列

- ビンソート（バケツソート）
  - 最大時間計算量 $O(n)$
  - 基本的アイデア
    - データを 1〜m の整数に制限
    - 値 i 以下のデータの数を $count[i]$ に求める
    - カウンタの代わりにリストを使用
  - 時間計算量
    - $O(n+m)$
    - m が定数なら，$O(n)$

＊ビンソート（バケツソート）

データ：1〜8

	入力	結果
1	3	1
2	7	1
3	1	2
4	1	3
5	3	3
6	6	3
7	8	6
8	7	7
9	2	7
10	3	8



＊比較以外の方法による整列

● 基底法
- ビンソート
  - ●データが1〜m の整数のとき，
    - 時間計算量 $O(n+m)$
    - サイズ m の配列を使用
      - → m が大きい（2^{32}）とき，非現実的

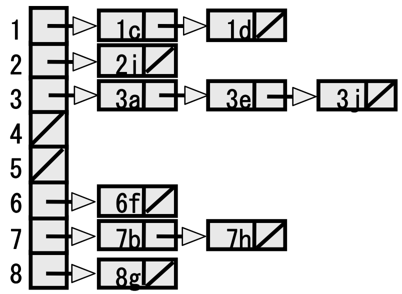
基底法

＊ビンソート（バケツソート）

● ビンソート　安定な整列
- 同じキーのデータは入力の順を保存

データ：1〜8

	入力	結果
1	3a	1c
2	7b	1d
3	1c	2i
4	1d	3a
5	3e	3e
6	6f	3j
7	8g	6f
8	7h	7b
9	2i	7h
10	3j	8g



＊比較以外の方法による整列

● 基底法
- ビンソートを利用
  - ●ビンソート　安定な（stable）整列
- データ　k 桁の m 進数（0〜m^k-1）
  - 32ビット正整数 ＝ 8桁の16進数（4ビット）
- 下位の桁から順にビンソートで整列

基底法

データ：0～999

	入力	第１桁	第２桁	第３桁
1	124	431	211	124
2	431	321	321	129
3	256	211	124	132
4	132	351	324	211
5	321	132	129	229
6	129	124	229	256
7	324	324	431	321
8	211	256	132	324
9	351	129	351	351
10	229	229	256	431

比較以外の方法による整列

- 基底法
  - ビンソートを利用
    - ビンソート　安定な（stable）整列
  - データ　k 桁の m 進数（0～m^k-1）

    32ビット正整数 ＝ 8桁の16進数（4ビット）
  - 下位の桁から順にビンソートで整列
  - 時間計算量

    $O((n+m)k)$

    $m,\ k$ が定数なら $O(n)$

ソートアルゴリズムの時間計算量

- バブルソート，セレクションソート　　常に$O(n^2)$
- インサーションソート

  最大，平均 $O(n^2)$，最小 $O(n)$
- シェルソート　　最大 $O(n^{4/3})$，平均 $O(n^{5/4})$
- ヒープソート　　最大 $O(n\log n)$　最適
- クイックソート，サンプルソート

  最大 $O(n^2)$，平均 $O(n\log n)$
- マージソート　　最大 $O(n\log n)$　最適
- 最大計算時間の下界（比較ｱﾙｺﾞﾘｽﾞﾑ）　$\Omega(n\log n)$
- ビンソート，基底法（比較以外の操作）　最大$O(n)$

まとめ　第５章　データの整列

5.1 バブルソート

5.2 セレクションソート

5.3 インサーションソート

5.4 シェルソート

5.5 ヒープソート

5.6 クイックソート

＊　サンプルソート

5.7 マージソート

5.8 ソート問題の計算複雑度

＊　ビンソート，基底法

この章の学習目標（振返り）

- さまざまな整列アルゴリズムとその計算時間を例を用いて説明できる
  - バブルソート，セレクションソート，インサーションソート，シェルソート，ヒープソート，クイックソート，サンプルソート，マージソート，ビンソート，基底法
- 整列問題の時間計算量の下界について，理由とともに説明できる
- 整列アルゴリズムのプログラムを書ける

データ構造とアルゴリズム　第10, 11回

1. アルゴリズムの重要性
2. 探索問題
3. 基本的なデータ構造
4. 動的探索問題とデータ構造
5. データの整列
6. グラフのアルゴリズム
7. 文字列のアルゴリズム
8. アルゴリズム設計手法

第６章　グラフアルゴリズム

6.1 グラフの利用

6.2 グラフの表現

6.3 用語の定義

6.4 グラフの探索

6.5 最短経路問題

6.6 ネットワークフロー

この章の学習目標

- グラフとは何か説明できる
- グラフを利用して問題を解ける例を示せる
- 隣接行列，隣接リストとその特徴を説明できる
- グラフアルゴリズムを実行例を示しながら説明できる
  - 幅優先探索，深さ優先探索，最短経路，最大フロー
- 上記アルゴリズムの計算時間を説明できる

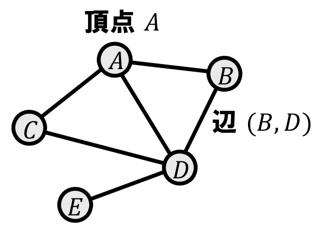
6.1 グラフの利用

- （無向）グラフ
  - 頂点，（無向）辺

    辺：頂点の順序のない組

    $\{u, v\}$：頂点 u, v 間の辺

    (u, v) または (v, u) と表すことも多い

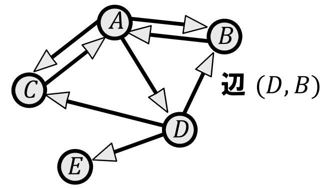


- 有向グラフ
  - 頂点，有向辺

    有向辺：頂点の順序のある組

    (u, v)：頂点 u から v への辺

    $(u, v) \neq (v, u)$



6.1 グラフの利用

- 計算機ネットワーク
  - 頂点：計算機　　辺：通信線
- 道路地図（最短経路）
  - 頂点：交差点　　辺：道路
- 鉄道地図（最短経路）
  - 頂点：駅　　　　辺：路線
- 研究室配属（マッチング）
  - 頂点：学生，研究室
  - 辺：配属希望研究室

第６章　グラフアルゴリズム

6.1 グラフの利用

6.2 グラフの表現

6.3 用語の定義

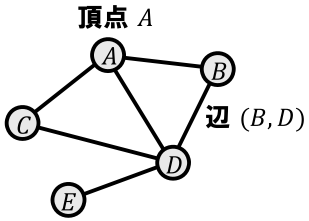
6.4 グラフの探索

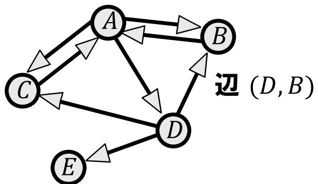
6.5 最短経路問題

6.6 ネットワークフロー

6.2 グラフの表現

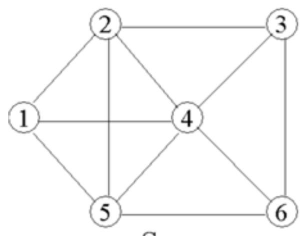
- グラフ G
  - $G=(V,E)$
    - V：頂点集合　　E：辺集合
- グラフ G の表現法
  - 隣接行列
    行列で辺集合を表現
  - 隣接リスト
    リストで辺集合を表現





6.2.1 隣接行列

- グラフ：$G=(V,E)$
  - 頂点集合：$V=\{1,2,\ldots,n\}$
- 隣接行列（adjacency matrix）
  - $M[1..n,1..n]$ of $bool$
  - $M[u,v]=true \Leftrightarrow$ 辺 $\{u,v\}$ が存在（無向グラフ）
    - 無向グラフの場合は対称行列



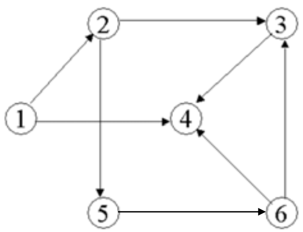
無向グラフの隣接行列

$$\begin{array}{c|c} & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{array}{c} 1\\2\\3\\4\\5\\6 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

対称行列

6.2.1 隣接行列

- **グラフ：**$G = (V, E)$
  - **頂点集合：**$V = \{1, 2, \ldots, n\}$
- **隣接行列（adjacency matrix）**
  - $M[1..n, 1..n]$ **of** $bool$
  - $M[u, v] = true$ **⇔ 辺** (u, v) **が存在（有向グラフ）**



有向グラフの隣接行列

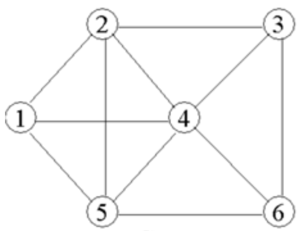
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

6.2.1 隣接行列

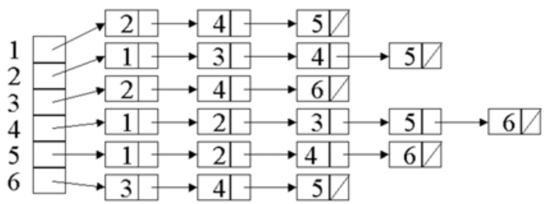
- **グラフ：**$G = (V, E)$
  - **頂点集合：**$V = \{1, 2, \ldots, n\}$
- **隣接行列（adjacency matrix）**
  - $M[1..n, 1..n]$ **of** $bool$
  - $M[u, v] = true$ **⇔ 辺** (u, v) **が存在**
  - **２頂点間の辺の有無の判定：定数時間**
  - **記憶領域：**$\Theta(n^2)$ **（必ず** n^2 **に比例）**
    辺は $O(n^2)$ **（高々** n^2 **に比例）**
    辺の多いグラフに適する
    辺の少ないグラフでは無駄が多い

6.2.2 隣接リスト

- **グラフ：**$G = (V, E)$
  - **頂点集合：**$V = \{1, 2, \ldots, n\}$
- **隣接リスト（adjacency list）**
  - $L[1..n]$
  - $L[u]$**：頂点** u **に隣接する頂点のリスト（無向グラフ）**
    - m **個の無向辺に対し，**$2m$ **個のセルを使用**



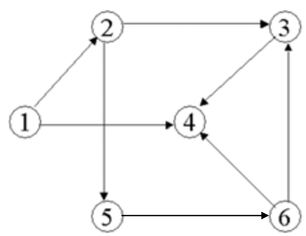
無向グラフの隣接リスト

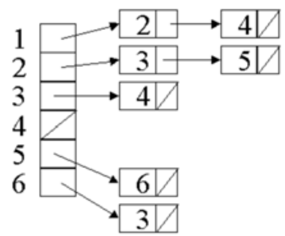


6.2.2 隣接リスト

- **グラフ：**$G = (V, E)$
  - **頂点集合：**$V = \{1, 2, \ldots, n\}$
- **隣接リスト（adjacency list）**
  - $L[1..n]$
  - $L[u]$**：頂点** u **から隣接する頂点のリスト（有向グラフ）**
    - m **個の有向辺に対し，**m **個のセルを使用**

有向グラフの隣接リスト



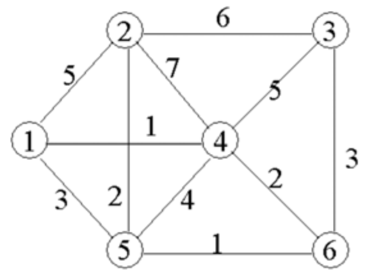


6.2.2 隣接リスト

- グラフ：$G = (V, E)$
  - 頂点集合：$V = \{1, 2, \ldots, n\}$
- 隣接リスト（adjacency list）
  - $L[1..n]$
  - $L[u]$：頂点 u に隣接する頂点のリスト（無向グラフ）
    頂点 u から隣接する頂点のリスト（有向グラフ）
  - ２頂点間の辺の有無の判定：$O(n)$ 時間
    → 隣接行列に劣る
  - 記憶領域：$\Theta(n+m)$　　m：辺数
    最適

6.2.3 重みつきグラフ

- 重みつきグラフ $G = (V, E, c)$
  - 辺が重みを有するグラフ
  - 辺の重み関数 $c: E \to R$　（R：実数の集合）
    - $c(e)$：辺 e の重み

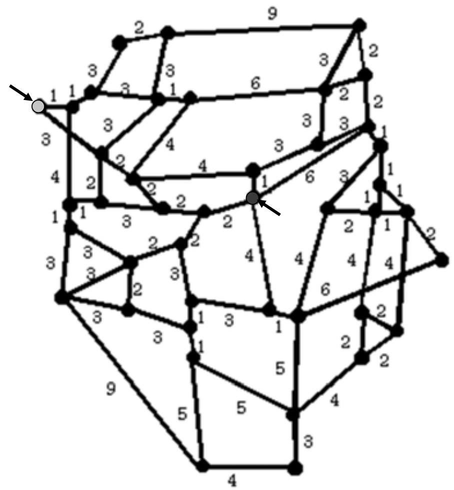


6.2.3 重みつきグラフ

- 重みつきグラフ $G = (V, E, c)$（p.106図6.1）
  - 辺が重みを有するグラフ
  - 辺の重み関数 $c: E \to R$　（R：実数の集合）
    - $c(e)$：辺 e の重み
- 例
  - 計算機ネットワーク
    - 頂点：計算機　　辺：通信線
    - 重み：通信遅延，帯域幅など
  - 道路地図
    - 頂点：交差点　　辺：道路
    - 重み：距離，走行時間，通行料など

最短経路問題の例

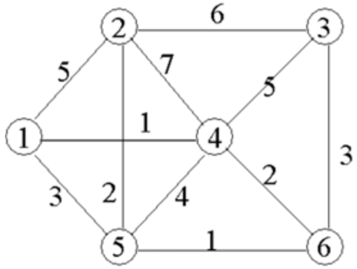
- ナビゲーションシステム



頂点：交差点
辺：道路
重み：距離

6.2.3 重みつきグラフ

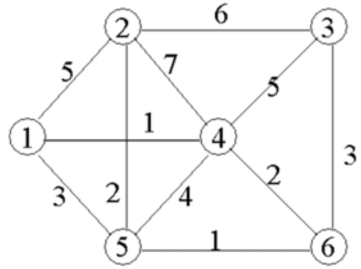
- 隣接行列
  - 論理値の代りに重みを使用
- 隣接リスト
  - セルに重みも格納

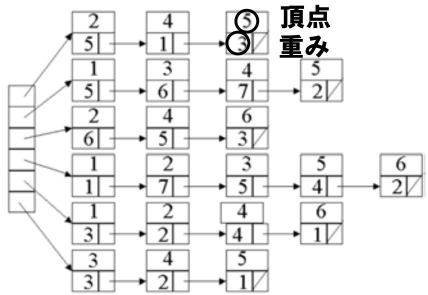


$$
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 5 & 0 & 1 & 3 & 0 \\
2 & 5 & 0 & 6 & 7 & 2 & 0 \\
3 & 0 & 6 & 0 & 5 & 0 & 3 \\
4 & 1 & 7 & 5 & 0 & 4 & 2 \\
5 & 3 & 2 & 0 & 4 & 0 & 1 \\
6 & 0 & 0 & 3 & 2 & 1 & 0
\end{array}
$$

6.2.3 重みつきグラフ

- 隣接行列
  - 論理値の代りに重みを使用
- 隣接リスト
  - セルに重みも格納



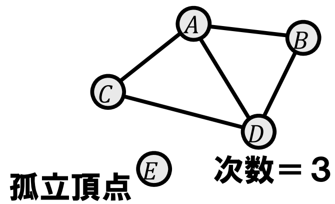


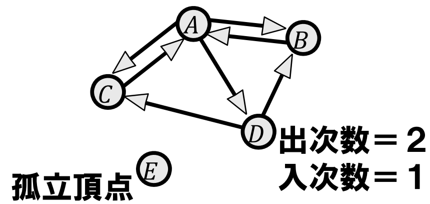
第６章　グラフアルゴリズム

6.1 グラフの利用
6.2 グラフの表現
6.3 用語の定義
6.4 グラフの探索
6.5 最短経路問題
6.6 ネットワークフロー

6.3 用語の定義（１）

- 次数（無向グラフ）：頂点に接続する辺数
- 出次数（有向グラフ）：頂点から出ている辺数
- 入次数（有向グラフ）：頂点に入っている辺数
- 隣接：頂点間に辺がある
- 孤立頂点：接続する辺を持たない頂点

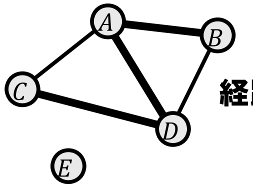




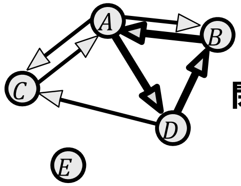
6.3 用語の定義（２）

- 経路
  - 頂点列 $v_0, v_1, \ldots, v_k$ （ただし，$(v_i, v_{i+1}) \in E$）
  - v_0 から v_k への長さ k の経路（長さは経路上の辺の数）
  - v_k は v_0 から到達可能
- 閉路：始点と終点が一致する経路
- 単純な経路／閉路：

途中で同じ頂点を２度以上通らない経路／閉路



経路

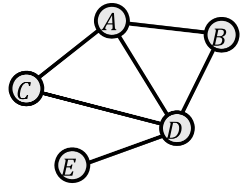


閉路

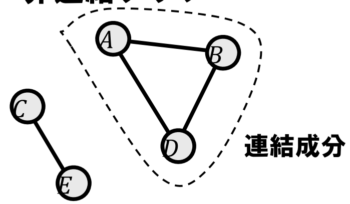
6.3 用語の定義（３）

- 連結：どの２頂点も互いに到達可能
  - 連結グラフ
  - 連結成分：極大な連結部分
- 木：閉路を含まない連結グラフ
- 根つき木：根（特別な頂点）が指定された木

連結グラフ



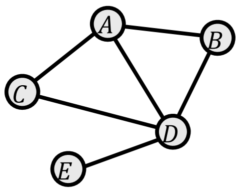
非連結グラフ



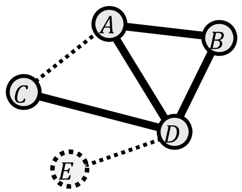
6.3 用語の定義（４）

- 部分グラフ：
  - $G' = (V',\ E')$ は $G = (V, E)$ の部分グラフ
  - $(V' \subseteq V) \land (E' \subseteq E)$
  - $G' \subseteq G$ と表す
- 全域木：$V' = V$ なる部分グラフで木のもの
- 最小全域木：重みの和が最小の全域木

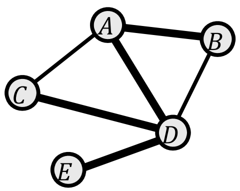
グラフ G



G の部分グラフ G'



全域木



第６章　グラフアルゴリズム

6.1 グラフの利用

6.2 グラフの表現

6.3 用語の定義

6.4 グラフの探索

6.5 最短経路問題

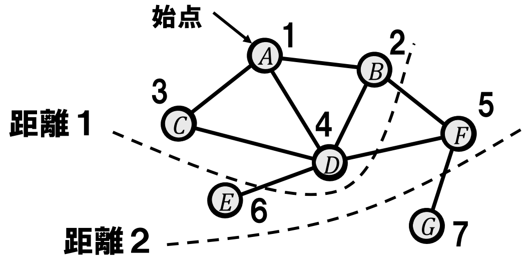
6.6 ネットワークフロー

6.4 グラフの探索

- グラフのすべての頂点，辺を１回ずつ処理
  - 隣接行列，隣接リストに現れる順に処理
    - グラフの構造に関係しない順
  - グラフの構造を反映した順に処理
    - 幅優先探索
      - キューを使用して簡単に実現できる
    - 深さ優先探索
      - スタックを使用して簡単に実現できる
  - 連結な無向グラフを対象に説明

6.4.1 幅優先探索

- 始点からの距離の順に頂点を訪問
  - 距離：最短経路の長さ（辺数）
  1. 始点（距離 0 の頂点）を訪問
  2. 距離 $k-1$ の頂点をすべて訪問したら，距離 k の頂点（距離 $k-1$ の頂点に隣接）のうち，未訪問のものを順に訪問（$k=1,2,\cdots$）



6.4.1 幅優先探索

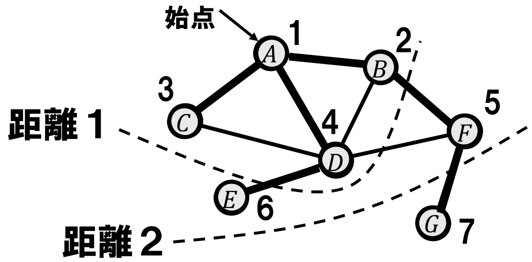
アルゴリズム 6.1 ― 幅優先探索アルゴリズム：BFS

- 入力：グラフ $G=(V,E)$，始点 $s \in V$
- for（各頂点 $v \in V$）{v に未訪問の印をつける}
  始点 s に訪問済の印をつける（s を訪問）
  キュー Q を $Q=[s]$ として初期化
  while（キュー Q が空でない）{
  　キュー Q から頂点 u を取り出す
  　for（u の各隣接頂点 $v \in adj(u)$）{
  　　if（v が未訪問）{
  　　　v に訪問済の印をつける（v を訪問）
  　　　v をキュー Q に入れる
  　　}
  　}
  }

キュー Q
訪問済の頂点を訪問順に挿入・取出し

幅優先木

- 幅優先探索で構成される木
  - 頂点 u から v を訪問（u の隣接頂点として v を訪問）
    $\Rightarrow$ u を v の親とする
- 始点 s を根とする全域木
  - s から各頂点 u への経路 $=$ G での最短 s–u 経路



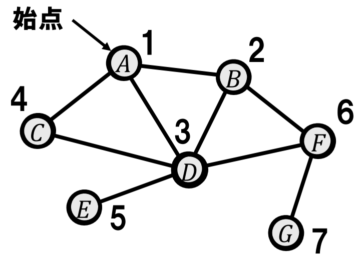
6.4.1 幅優先探索

- **計算時間**
  - 隣接リスト　$O(n+m)$
  - 隣接行列　　$O(n^2)$

    n：頂点数　　　m：辺数
- **非連結グラフ**
  - 連結成分ごとに幅優先探索を実行
  - キューが空になると，未訪問頂点を探す
  - 未訪問頂点があれば，そこから幅優先探索を開始
- **有向グラフ**
  - 無向グラフの場合と同様

6.4.2 深さ優先探索

- 始点を最初に訪問
- 未訪問の隣接頂点があれば，その頂点を訪問
  - 未訪問の隣接頂点がなければ，ひとつ前の頂点に戻る



6.4.2 深さ優先探索

アルゴリズム 6.2 ― 深さ優先探索アルゴリズム：DFS

> スタック S
> 訪問予定の辺を
> 挿入・取出し

- 入力：グラフ $G=(V,E)$，始点 $s\in V$
- スタック S を 空に初期化

  for（各頂点 $v\in V$）{v に未訪問の印をつける}
  始点 s に訪問済の印をつける（s を訪問）
  for（各辺 $(s,v)\in E$）{(s,v) をスタック S に push}
  while（スタック S が空でない）{
  　スタック S から辺 (x,y) を取り出す（pop）
  　if（y が未訪問）{
  　　y に訪問済の印をつける（y を訪問）
  　　for（各辺 $(y,z)\in E$）{(y,z) をスタック S に push
  　}
  }

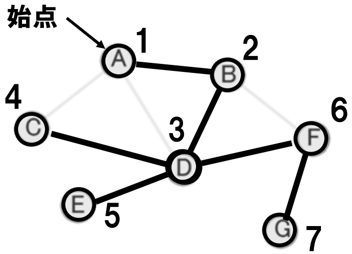
6.4.2 深さ優先探索（再帰版）

アルゴリズム 6.3 ― 深さ優先探索アルゴリズム：DFS

- 深さ優先探索は再帰を使えば簡単に書ける
  - 再帰とスタックは密接な関係
- 入力：グラフ $G=(V,E)$，始点 $s\in V$
- s に関する深さ優先探索手続き $dfs(s)$ を呼出す
- 手続き $dfs(u)$ {
  　頂点 u に訪問済の印をつける
  　for（各辺 $(u,v)\in E$）
  　　if（v が未訪問） $dfs(v)$
  }

深さ優先木

- 深さ優先探索で構成される木
  - 頂点 u から v を訪問（辺 (u,v) の処理で v を訪問）
    ⇒ u を v の親とする
- 始点 s を根とする全域木

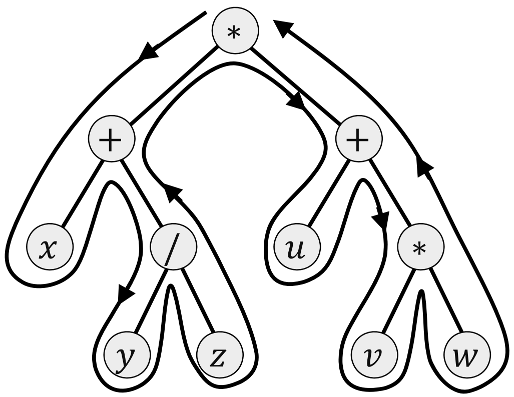


6.4.2 深さ優先探索

- 計算時間
  - 隣接リスト　$\boldsymbol{O(n+m)}$
  - 隣接行列　$\boldsymbol{O(n^2)}$
    n：頂点数　m：辺数
- 非連結グラフ
  - 連結成分ごとに深さ優先探索を実行
  - スタックが空になると，未訪問頂点を探す
  - 未訪問頂点があれば，そこから深さ優先探索を開始
- 有向グラフ
  - 無向グラフの場合と同様

6.4.3 深さ優先探索の応用

- 計算式を表現する根つき２分木



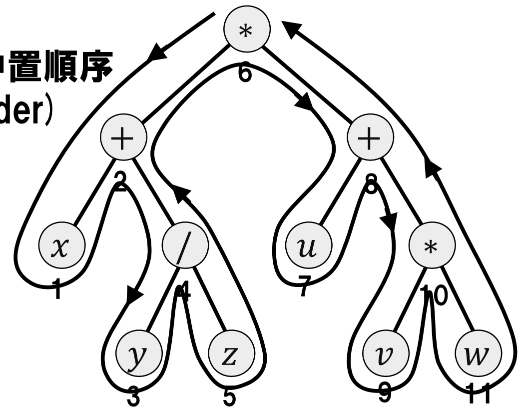
中置記法　$((x+(y/z))*(u+(v*w)))$

前置記法　$(*\,(+x(/yz)(+u(*\,vw)))$　ポーランド記法

後置記法　$((x(yz/)+)(u(vw*)+)*)$　逆ポーランド記法

中置記法

数字は中置順序
(inorder)



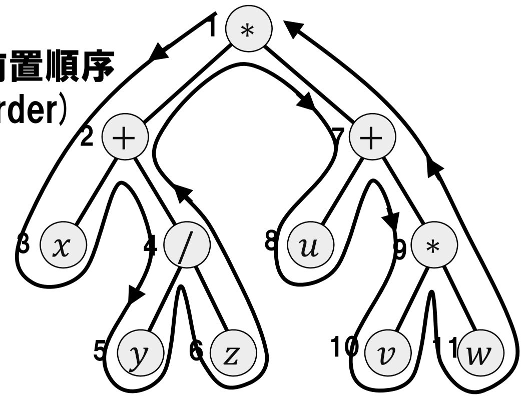
中置記法　$((x+(y/z))*(u+(v*w)))$　括弧は必要

✓ 葉は初めて訪問したときに出力

✓ 葉以外は，２回目に訪問したとき（左の子から戻ってきたとき）に出力

前置記法

数字は前置順序
(preorder)

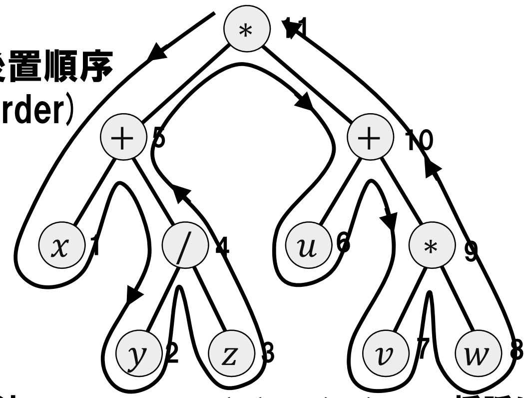


前置記法　$(*\ (+x(/yz)(+u(*\ vw))))$　括弧は不要

- ✓ ポーランド記法
- ✓ 初めて訪問したときに出力

後置記法

数字は後置順序
(postorder)

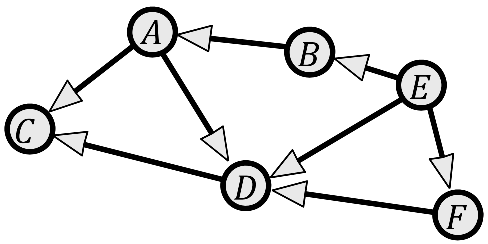


後置記法　$((x(yz/)+)(u(vw\ *)\ +)\ *)$　括弧は不要

- ✓ 逆ポーランド記法
- ✓ 葉は初めて訪問したときに出力
- ✓ 葉以外は，３回目に訪問したとき（右の子から戻ってきたとき）に出力に出力

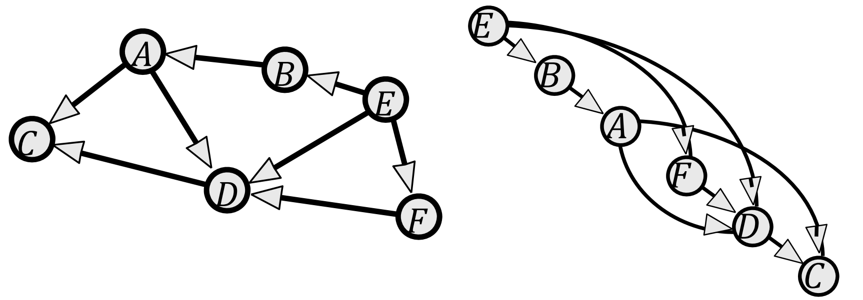
＊深さ優先探索の応用：トポロジカルソート

- DAG（Directed Acyclic Graph）
  - 閉路を含まない有向グラフ



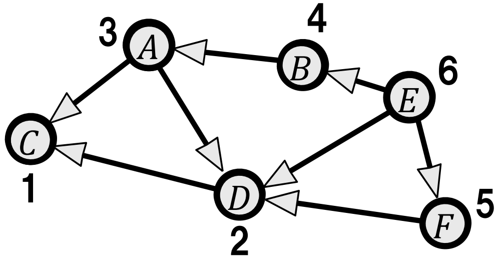
＊深さ優先探索の応用：トポロジカルソート

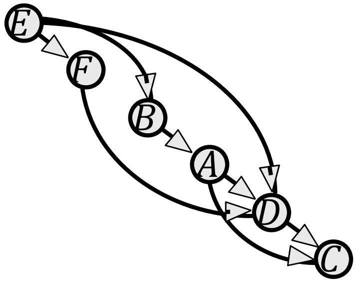
- トポロジカルソート
  - DAGの頂点を一直線上に配置
    - すべての有向辺の向きが同じになるように
  - 解は一意には定まらない



＊深さ優先探索の応用：トポロジカルソート

- トポロジカルソート
  1. 深さ優先探索を実行
  2. 手続き dfs 終了時に番号付け
     - 深さ優先探索木の後置順序
  3. この番号の逆順に並べる



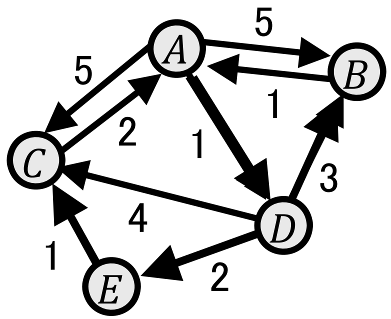


第６章　グラフアルゴリズム

6.1 グラフの利用
6.2 グラフの表現
6.3 用語の定義
6.4 グラフの探索
6.5 最短経路問題
6.6 ネットワークフロー

6.5 最短経路問題

- 最短経路問題
  - 入力：重みつき有向グラフ $\mathrm{G} = (\mathrm{V}, \mathrm{E}, \mathrm{c})$
    辺 (u, v) の重み $c(u, v)$ は非負
  - 出力：２頂点間の最短経路と距離



6.5 最短経路問題

- 最短経路問題
  - 入力：重みつき有向グラフ $G = (V, E, c)$
  - 出力：２頂点間の最短経路と距離
- ２頂点対最短経路問題
  - 指定された２頂点間の最短経路
- 単一始点最短経路問題
  - 指定された１頂点から全頂点への最短経路
- 全頂点対最短経路問題
  - 全頂点間の最短経路

6.5 最短経路問題

- 最短経路問題
  - 入力：重みつき有向グラフ $G=(V,E,c)$
  - 出力：２頂点間の最短経路と距離
- 辺の重みがすべて等しいとき
  - 幅優先探索で解ける
    - 計算時間（単一始点最短経路問題）
      - 隣接リスト　$\boldsymbol{O(n+m)}$
      - 隣接行列　$\boldsymbol{O(n^2)}$

n：頂点数　　m：辺数

6.5 最短経路問題

- 最短経路問題
  - 入力：重みつき有向グラフ $G=(V,E,c)$
  - 出力：２頂点間の最短経路と距離
- ２頂点対最短経路問題
  - 単一始点最短経路問題よりもオーダ的に高速なアルゴリズムは知られていない
- 全頂点対最短経路問題
  - 単一始点最短経路問題を n 回繰り返すよりもオーダ的に高速なアルゴリズムが存在

２頂点対最短経路問題

- ダイクストラのアルゴリズム
  - 代表的な貪欲法（グリーディ法，greedy）
  - 計算時間
    - 単純に配列で実現　$O(n^2)$ 時間
    - ヒープを利用　$O((n+m)\log n)$ 時間
    - フィボナッチヒープを利用
      $O(n\log n+m)$ 時間

ダイクストラの最短経路アルゴリズム

- 入力：重みつき無向グラフ $G=(V,E,c)$，始点 s
- for（各頂点 $v\in V$）$D[v]=\infty$

  $X=\emptyset$, $D[s]=0$

  while $(X\neq V)$ {

  　$V-X$ の中で D の値が最小の頂点を u とする

  　$X=X\cup\{u\}$ とする

  　for（頂点 u に接続する各辺 (u,v) s.t. $v\in V-X$）

  　　$D[v]=\min(D[v],\ D[u]+c(u,v))$

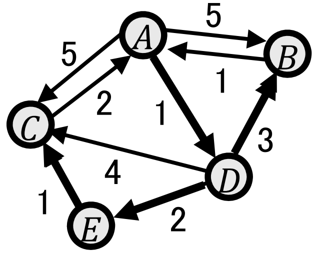
  }

配列 $D[v]$：始点から v までの仮の最短経路長
集合 X：最短経路が確定した頂点の集合

u の最短経路長を $D[u]$ に確定

確定した $D[u]$ による仮の最短経路長の更新

ダイクストラの最短経路アルゴリズム：適用例



A:始点

	D[A	B	C	D	E]	X
1	0	∞	∞	∞	∞	$\emptyset$
	0	5	5	1	∞	A
2	0	5	5	1	∞	A
	0	4	5	1	3	$A\ D$
3	0	4	5	1	3	$A\ D$
	0	4	4	1	3	$A\ D\ E$
4	0	4	4	1	3	$A\ D\ E$
	0	4	4	1	3	$A\ D\ E\ B$
5	0	4	4	1	3	$A\ D\ E\ B$
	0	4	4	1	3	$A\ D\ E\ B\ C$

ダイクストラの最短経路アルゴリズム：正当性

配列 $D[v]$：始点から v までの仮の最短経路長
集合 X：最短経路が確定した頂点の集合

【命題6.5.1】
v を X に属する頂点に隣接する頂点とする．
このとき，$D[v]$ には X の頂点のみを経由する最短 s-v 経路（X 経由最短 s-v 経路と表す）の長さが格納されている

✓ $|X|$ に関する帰納法で証明

✓ 終了時，$X = V$ （つまり，どの頂点も経由可能）
　→ $D[v]$ は最短 s-v 経路の長さを格納

ダイクストラの最短経路アルゴリズム：正当性

配列 $D[v]$：始点から v までの仮の最短経路長
集合 X：最短経路が確定した頂点の集合

【命題6.5.1】
v を X に属する頂点に隣接する頂点とする．
このとき，$D[v]$ には X の頂点のみを経由する最短 s-v 経路（X 経由最短 s-v 経路と表す）の長さが格納されている

証明の概略（$|X| = k$ のときの X の値を X_k と表す）

- $|X| = 1$（$X_1 = \{s\}$）のとき
  - $D[v] = c(s, v)$　　if v が s に隣接
    $= \infty$　　if v が s に隣接していない
  - $D[v]$ が $\{s\}$ 経由最短 s-v 経路の長さである（自明）
- $|X| = k$ のとき成立すると仮定（帰納仮定）

ダイクストラの最短経路アルゴリズム：正当性

配列 $D[v]$：始点から v までの仮の最短経路長
集合 X：最短経路が確定した頂点の集合

【命題6.5.1】
v を X に属する頂点に隣接する頂点とする．
このとき，$D[v]$ には X の頂点のみを経由する最短 s-v 経路（X 経由最短 s-v 経路と表す）の長さが格納されている

証明の概略（続き）

- $|X| = k + 1$ のとき（$\{u\} = X_{k+1} - X_k$ とする）
  - X_{k+1} 経由最短 s-v 経路が必ず u を通る場合を考えれば十分
    $\because$ u を通らない X_{k+1} 経由最短 s-v 経路が存在すれば自明
  - $v \in X_k$ なら u を通らない X_{k+1} 経由最短 s-v 経路が存在
    $\because$ $D[w] \leq D[u]$ が成立（D の値が小さい順に確定していく）
  - $v = u$ の場合は自明（X_k 経由でも X_{k+1} 経由でも同じ）
  - $v \in V - X_{k+1}$ の場合を考えればよい
    - X_{k+1} 経由最短 s-v 経路の v の直前の頂点は u
    - $D[u]$ が X_{k+1}経由最短 s-u 経路長であることから示せる

ダイクストラの最短経路アルゴリズム：計算時間 (1)

- **計算時間**

A) $V-X$ の中で D の値が最小の頂点 u を選択

B) 各辺 (u,v) に対し，$D[v] = min(D[v],\ D[u] + c(u,v))$

- 単純に配列で実現　$O(n^2)$ 時間

  A)　$O(n^2)$ 時間

  B)　$O(m)$ 時間　　辺数 $m \le \frac{n(n-1)}{2}$　　無向グラフ

  $m \le n(n-1)$　有向グラフ

ダイクストラの最短経路アルゴリズム：計算時間 (2)

- **計算時間**

A) $V-X$ の中で D の値が最小の頂点 u を選択

B) 各辺 (u,v) に対し，$D[v] = min(D[v],\ D[u] + c(u,v))$

- ヒープを利用　$O((n+m)\log n)$ 時間

  A)　$O(n \log n)$ 時間

  B)　$O(m \log n)$ 時間

- フィボナッチヒープを利用

  $O(n \log n + m)$ 時間　　実用性は低い

  A)　$O(n \log n)$ 時間

  B)　$O(m)$ 時間　ならし（amortized）計算量

ならし計算量

- **フィボナッチヒープを利用**

B) 各辺 (u,v) に対し，$D[v] = \min(D[v],\ D[u] + c(u,v))$

- $O(m)$ 時間　ならし（amortized）計算量
  - B) の計算は m 回
  - B) の各計算が $O(1)$ 時間ではない

    B) の m 回の計算全体で $O(m)$ 時間

＊全頂点対最短経路問題

- **２頂点対最短経路問題，単一始点最短経路問題**
  - ダイクストラのアルゴリズム
    - フィボナッチヒープを利用　$O(n \log n + m)$ 時間
- **全頂点対最短経路問題**

a. ダイクストラのアルゴリズムを各頂点に適用

- $\boldsymbol{O(n^2 \log n + nm)}$ 時間（$= \boldsymbol{O(n^3)}$）

b. フロイドのアルゴリズム

- $\boldsymbol{O(n^3)}$ 時間（a.よりよくない）
  - 操作が簡単なので高速
  - 負の重みの辺があっても適用可能

＊フロイドのアルゴリズム

$O(n^3)$ **時間**

【方針】

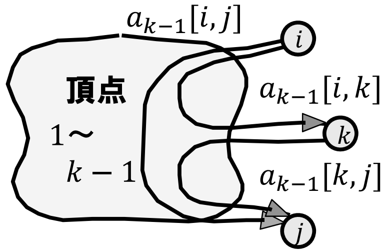
- **頂点：** $1,2,\ldots,n$
- $a_k[i,j]=$ **頂点** $1,2,\ldots,k$ **だけを経由する**
  頂点 i **から頂点** j **への最短路の長さ**
  - $a_0[i,j]=$ **辺** (i,j) **の重み**
  - $a_n[i,j]=$ **頂点** i **から頂点** j **への最短路の長さ**
- $a_0[i,j],\ a_1[i,j],\ldots,\ a_n[i,j]$ **を順に求める**

＊フロイドのアルゴリズム

$O(n^3)$ **時間**

【方針】

- $a_0[i,j],\ a_1[i,j],\ldots,\ a_n[i,j]$ **を順に求める**
  - $a_{k-1}[i,j]$ **から** $a_k[i,j]$ **を求める方法**
    - $a_k[i,j]=\min\{a_{k-1}[i,j],\ \ a_{k-1}[i,k]+a_{k-1}[k,j]\}$



＊フロイドのアルゴリズム

$O(n^3)$ **時間**

【アルゴリズム】

for $i:=1$ to n do
  for $j:=1$ to n do
    $a[i,j]:=c[i,j]$; /＊ **辺** (i,j) **がなければ** ∞ ＊/
for $k:=1$ to n do
  for $i:=1$ to n do
    for $j:=1$ to n do
      if $a[i,k]+a[k,j]<a\ [i,j]$ then
        $a[i,j]:=a[i,k]+a[k,j]$

＊フロイドのアルゴリズム

for $k:=1$ to n do
  for $i:=1$ to n do
    for $j:=1$ to n do
      if $a[i,k]+a[k,j]<a\ [i,j]$ then
        $a[i,j]:=a[i,k]+a[k,j]$

- **動的計画法（dynamic programming）**
  - **最適問題の解法の一つ**
  - **部分的な解を表に記録し，（再計算せず）再利用することにより，効率よく最適解を求める**

第６章　グラフアルゴリズム

6.1 グラフの利用

6.2 グラフの表現

6.3 用語の定義

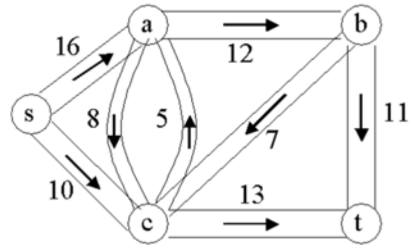
6.4 グラフの探索

6.5 最短経路問題

6.6 ネットワークフロー

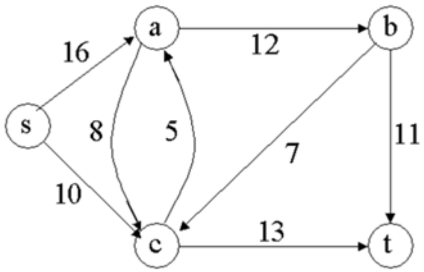
ネットワークフローとは

- 水道網でソース s からシンク t に最大量の水を流したい
  - 水道網：水道管で構成
  - 水道管
    - 流せる向きが決まっている（有向グラフ）
    - 容量（一定時間に流せる水量の上限）がある
- 送電，物流，交通など応用多数



ネットワークフロー問題

- ネットワークフロー
  - 入力：重みつき有向グラフ $G = (V, E, c)$，頂点 s, t
    - s：ソース（s に入る有向辺はないものとする）
    - t：シンク（t から出る有向辺はないものとする）
  - 出力：最大 s-t 流を実現するための各辺の流量
    辺 (u, v) の流量 $f(u, v)$　フロー

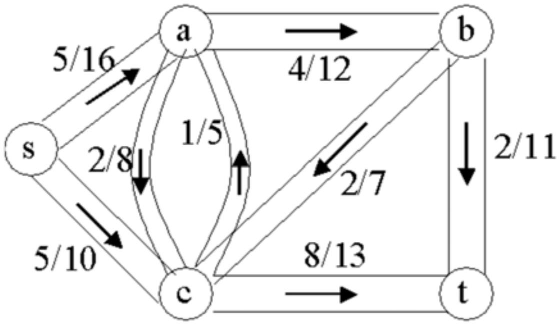


フロー f

- ネットワークフロー問題
  - 最大 s-t 流を実現する各辺のフローを求める
- フロー f が満たすべき条件　＊テキストを簡略化
  - フロー保存性：任意の $u \in V - \{s, t\}$ に対し，
    $\sum_{(v,u)\in E} f(v, u) = \sum_{(u,v)\in E} f(u, v)$
    （u の流入フローの和と流出フローの和が等しい）
  - 容量制約：任意の $(u, v) \in E$ に対し，$f(u, v) \le c(u, v)$
    （辺 (u, v) のフローは重み（容量）以下）
- 飽和辺：$f(u, v) = c(u, v)$ となる辺 (u, v)

ネットワークフロー

- **総フロー（s–t 流）$|f|$：s から t への流量**
  - $|f| = \sum_{(s,v)\in E} f(s,v)\ (= \sum_{(v,t)\in E} f(v,t))$
    - s **から流出するフローの総和**
    - **フロー保存性より，t に流入するフローの総和と同じ**



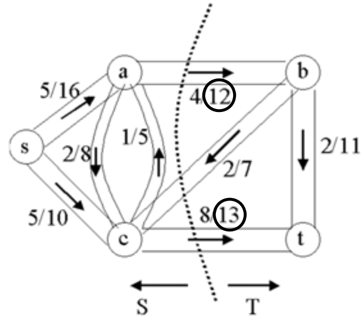
フロー／容量

$|f| = 10$

最大流最小カット定理：準備

- s,t**–カット**
  - **頂点集合 V の分割 (S,T)：$S \cup T = V,\ S \cap T = \emptyset$**
  - $s \in S,\ t \in T$
- s,t**–カット (S,T) の容量**
  - $\sum_{u\in S, v\in T} c(u,v)$

    （S の頂点から T の頂点に向う辺の容量の和）



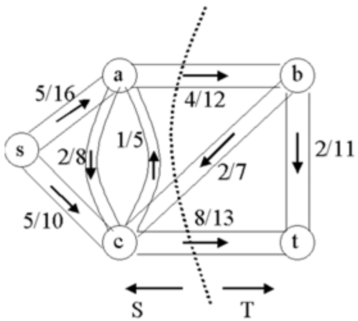
フロー／容量

(S,T) **の容量** $= 25$

最大流最小カット定理

- s,t**–カットの最大容量は総フロー $|f|$ に一致する**
  - **任意の s,t–カット (S,T) に対し，$|f| \le c(S,T)$ は自明**
    - $c(S,T)$**：S から T に流せるフローの上界**
  - T **から S へのフローがあると $|f| < c(S,T)$**
  - $|f| = c(S,T)$ **となる s,t–カット (S,T) が存在する**

この定理の本質

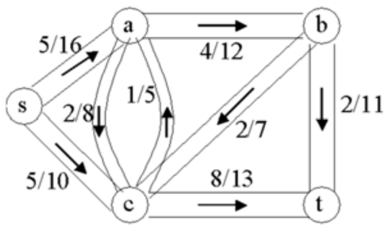


最大流アルゴリズムの概略

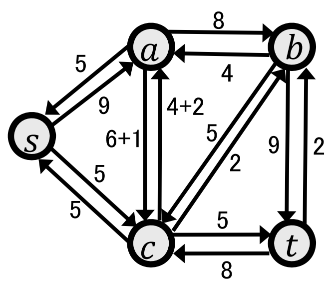
1. **総フロー $|f| = 0$ から開始**
2. **残余グラフを構成し，拡張可能経路を見つけて総フロー $|f|$ を増やす**
3. **2 を総フロー $|f|$ が増やせなくなるまで繰り返す**

残余グラフ

- フロー f に対し，各辺 (u,v) で増加可能な流量 $r(u,v)$
  - $u \to v$ 方向：$r(u,v) = c(u,v) - f(u,v)$　　容量の残余
  - $v \to u$ 方向：$r(v,u) = f(u,v)$　　フロー f を押し戻す
- 残余グラフ $G_f = (V, E_f)$
  - $E_f = \{(u,v) \mid r(u,v) > 0\}$　$r(u,v)$：辺 (u,v) の残余容量



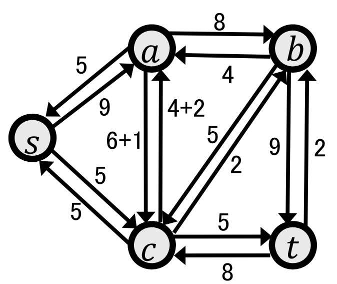
フロー f



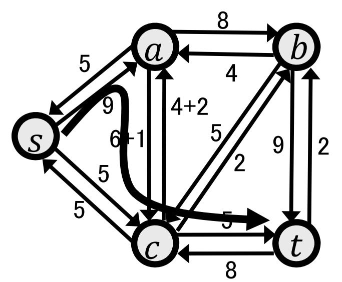
残余グラフ

拡張可能経路

- フロー f に対し，残余グラフ $G_f = (V, E_f)$ での s-t 経路 p
  - 各辺 (u,v) の残余容量 $r(u,v) > 0$
  - 拡張可能経路 p に沿って，フローを増加できる
    - フローの増加量：p に現れる辺の最小残余容量



残余グラフ



拡張可能経路（フロー 5）

最大流アルゴリズム

f をゼロフロー（全辺のフローが 0）とする

f に対する残余グラフを G_f とする

while（G_f に拡張可能経路が存在）{

　G_f の拡張可能経路 p を求める

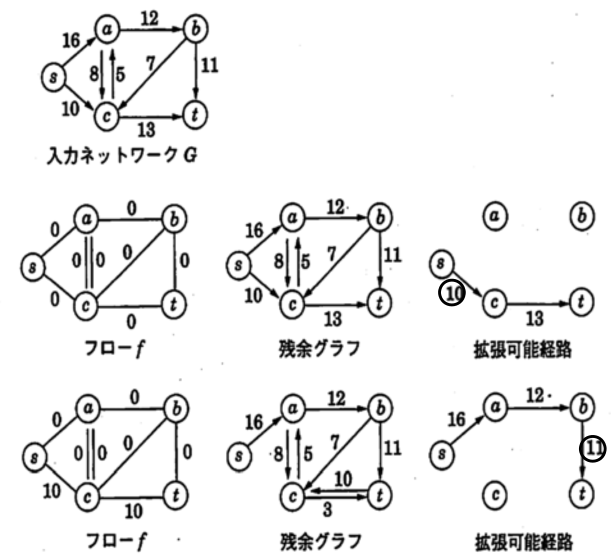
　p 上の辺の最小残余容量の分だけフロー f を増やす

　更新された f に対する残余グラフを G_f とする

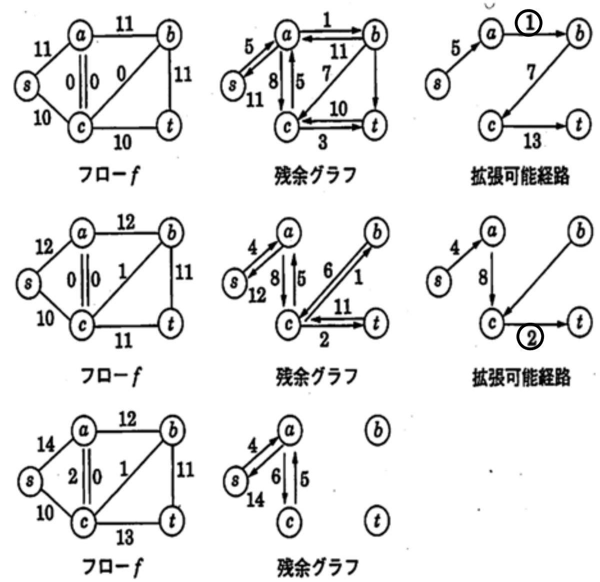
}

f を最大フローとして出力

最大流アルゴリズム：実行例 (1)



最大流アルゴリズム：実行例(2)



最大流アルゴリズム：時間計算量

- 容量はすべて正整数と仮定
- while 文の繰り返し回数は高々 $|f_{max}|$ （最大総フロー）
  - 拡張可能経路に沿ってフローは 1 以上増加
- 残余グラフ構築，拡張可能経路の発見：$O(m)$ 時間
  - m：変数
- 全体の時間計算量：$O(m|f_{max}|)$

- $O(|V|^3)$ 時間のアルゴリズムも知られている

まとめ　第６章　グラフアルゴリズム

6.1 グラフの利用

6.2 グラフの表現

6.3 用語の定義

6.4 グラフの探索

6.5 最短経路問題

6.6 ネットワークフロー

この章の学習目標（振返り）

- グラフとは何か説明できる
- グラフを利用して問題を解ける例を示せる
- 隣接行列，隣接リストとその特徴を説明できる
- グラフアルゴリズムを実行例を示しながら説明できる
  - 幅優先探索，深さ優先探索，最短経路，最大フロー
- 上記アルゴリズムの計算時間を説明できる

データ構造とアルゴリズム 第１2，14回（2020.1.20, 2.3）

1. アルゴリズムの重要性
2. 探索問題
3. 基本的なデータ構造
4. 動的探索問題とデータ構造
5. データの整列
6. グラフのアルゴリズム
7. 文字列のアルゴリズム
8. アルゴリズム設計手法

7　文字列のアルゴリズム

7.1 文字列照合問題とは

7.2 力まかせ法

7.3 ラビン-カープ法

7.4 クヌース-モリス-プラッツ法

7.5 ボイヤー-ムーア法

この章の学習目標

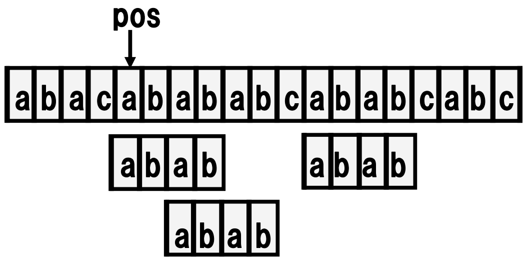
- 文字列照合問題とは何か，応用例を用いて説明できる
- 力まかせ法とは何か，また，その時間計算量を説明できる
- ハッシュを利用した文字列照合アルゴリズムのアイデアとその時間計算量を説明できる
- 線形時間の文字列照合アルゴリズムのアイデアとその時間計算量を説明できる
- 文字列照合アルゴリズムのプログラムを書ける

7.1 文字列照合問題とは

- 文字列照合
  - テキストから特定のパターンを探索する
  - テキスト：文字列（n 文字）
    - $text$: array $[1..n]$ of char ;
  - パターン：文字列（m 文字）
    - $pattern$: array $[1..m]$ of char ;
  - 一般に $n \gg m$
  - 講義で紹介するアルゴリズムは，
    $pattern[1..m] = text[pos..pos+m-1]$
    を満たす最小の pos を求める

7.1 文字列照合問題とは

- 文字列照合
  - テキスト：abacababcababcabc
  - パターン：abab



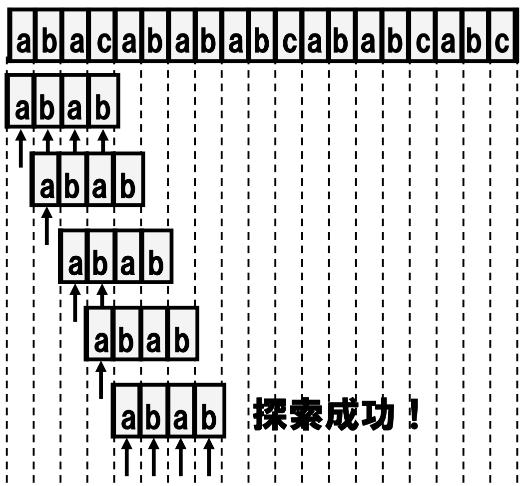
・パターンに一致する部分は３カ所
・どの位置を求めてもよい
・紹介するアルゴリズムはすべて最初の位置を求める

7　文字列のアルゴリズム

7.1 文字列照合問題とは
7.2 力まかせ法
7.3 ラビン-カープ法
7.4 クヌース-モリス-プラッツ法
7.5 ボイヤー-ムーア法

7.2 力まかせ法

- テキストでの位置を一つずつ右にずらしながらパターンを探索



7.2 力まかせ法

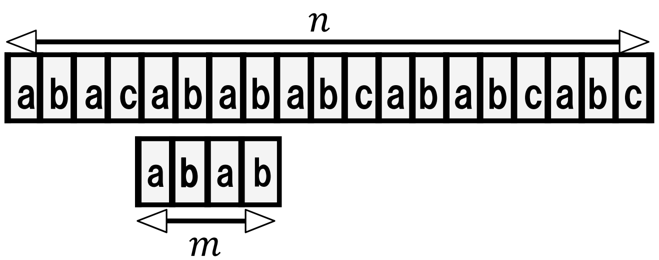
アルゴリズム7.1 ー 力まかせ法

// テキスト：$T[0]T[1]$ … $T[n-1]$
// パターン：$P[0]P[1]$ … $P[m-1]$ $(n \geq m)$

```
for (i=0; i<=n-m; i++) {
    for (j=0; j<m; j++)
        if (T[i+j] != P[j]) break;
    if (j==m) return i; // パターンを検出した
}
return -1; // パターンが出現しなかった
```

7.2 力まかせ法

- テキストでの位置を一つずつ右にずらしながらパターンを探索
- 最大計算時間
  - $O(nm)$　　n：テキストの長さ
    　　　　　m：パターンの長さ



7.2 力まかせ法

- テキストでの位置を一つずつ右にずらしながらパターンを探索
- 最大計算時間
  - $O(nm)$　　n：テキストの長さ
    　　　　　m：パターンの長さ
  - m が定数 $\Rightarrow O(n)$　　最適
  - $m = \frac{n}{2} \Rightarrow O(n^2)$　膨大な計算時間

7　文字列のアルゴリズム

7.1 文字列照合問題とは

7.2 力まかせ法

7.3 ラビン-カープ法

7.4 クヌース-モリス-プラッツ法

7.5 ボイヤー-ムーア法

7.3 ラビン-カープ法

- 力まかせ法をハッシュ法のアイデアで改良
- ハッシュ関数 h
  - 長さ m の文字列 → ハッシュ値（整数値）
  - $h^P = h(P[0]P[1] \ldots P[m-1])$
  - $h_i^T = h(T[i]T[i+1] \ldots P[i+m-1])$　$(0 \le i \le n-m)$
  - h^P を h_i^T $(0 \le i \le n-m)$ と順次比較
- $h^P = h_i^T$ のとき，
  $P[0]P[1] \cdots P[m-1]$ と $T[i]T[i+1] \cdots T[i+m-1]$ を比較
  - ハッシュ値の衝突の可能性があるため
    ハッシュ値の衝突：異なる文字列のハッシュ値が一致

7.3 ラビン-カープ法

アルゴリズム7．２　ラビン-カープ法

// テキスト：$T[0]T[1]\ \dots\ T[n-1]$

// パターン：$P[0]P[1] \dots P[m-1]$　$(n \le m)$

パターンのハッシュ関数 h^P を計算;

for（i=0; i<=n−m; i++）{

　テキストのハッシュ関数 h_i^T を計算;

　if（h^P == h_i^T）

　　for（j=0; j<m; j++）

　　　if（$T[i+j]$!= $P[j]$）break;

　if（j==m）return i; // パターンを検出した

}

return −1; // パターンが出現しなかった

ハッシュ関数の例

- a, b, c の3文字からなる長さ4の文字列 $uvwx$
- $f(a) = 0,\ f(b) = 1,\ f(c) = 2$ として，
  $h(uvwx)$＝$(f(u) \times 3^3$＋$f(v) \times 3^2 + f(w) \times 3$＋$f(x)$ mod 11
- 例：$h(abab) =\ (1 \times 3^2$＋$1)$ mod $11 = 10$ mod $11 = 10$
  $h(babc) =\ (1 \times 3^3$＋1×3＋$2)$ mod 11
  $= 32$ mod $11 = 10$

ラビン-カープ法の実行例

- $f(a) = 0,\ f(b) = 1,\ f(c) = 2$ として，
  $h(uvwx) = (f(u) \times 33$＋$f(v) \times 32$＋$f(w) \times 3$＋$f(x))$ mod 11

T																		
a	b	a	c	a	b	a	b	a	b	c	a	b	a	b	c	a	b	c

h_i^T															
0	0	8	2	10	8	10	10	4	2	2	10	10	4	2	4

P			
b	a	b	c

h^P＝10

7.3 ラビン-カープ法

- 力まかせ法をハッシュ法のアイデアで改良
- 計算時間
  - 各位置 $i\ (0 \le i \le n-m)$ での $h^P = h_i^T$ の判定
    定数時間
  - ハッシュ値の衝突が少なければ，
    全体で $O(n)$ 時間
    - ハッシュ関数の計算時間を除けば
  - 最大 $O(nm)$ 時間
    ハッシュ値の衝突が $O(n)$ 回生じた場合

ハッシュ関数

- 力まかせ法をハッシュ法のアイデアで改良
- ハッシュ関数 h
  - 長さ m の文字列 → ハッシュ値（整数値）
  - $h(a_1 a_2 \cdots a_m) = \underbrace{(a_1 d^{m-1} + a_2 d^{m-2} + \cdots + a_m)}_{d\ \text{進数として数値化}}$ mod q
  - $h_i^T = h(T[i]T[i+1] \cdots T[i+m-1])$
    $= (T[i]d^{m-1} + T[i+1]d^{m-2} + \cdots + T[i+m-1])$ mod q
  - $h_{i+1}^T = h\ (T[i+1]T[i+2] \ldots T[i+m])$
    $= (T[i+1]d^{m-1} + T[i+2]d^{m-2} + \cdots + T[i+m])$ mod q
    $= ((h_i^T - T[i]d^{m-1})d + T[i+m])$ mod q
  - h_{i+1}^T は h_i^T から定数時間で計算可能

d：文字種数
q：素数

ハッシュ関数

- 力まかせ法をハッシュ法のアイデアで改良
- ハッシュ関数 h
  - 長さ m の文字列 → ハッシュ値（整数値）
  - $h(a_1 a_2 \cdots a_m) = (a_1 d^{m-1} + a_2 d^{m-2} + \cdots + a_m)$ mod q
  - q が大きいほど，ハッシュ関数の衝突は少ない
    - ハッシュ表は不要なので，メモリ量を気にせずに q を大きくする（オーバーフローを生じない程度に）

7　文字列のアルゴリズム

7.1 文字列照合問題とは

7.2 力まかせ法

7.3 ラビン-カープ法

7.4 クヌース-モリス-プラッツ法

7.5 ボイヤー-ムーア法

7.4 クヌース-モリス-プラッツ法

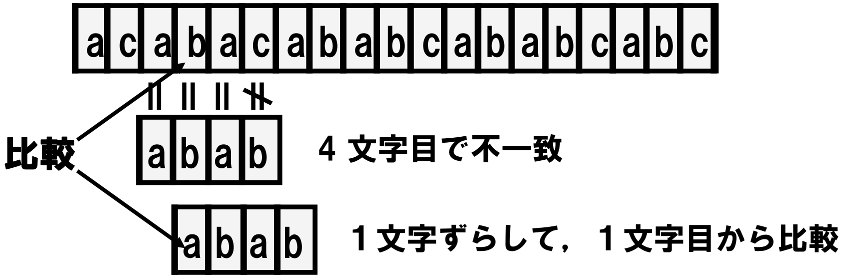
- 力まかせ法
  - 最大計算時間　$O(nm)$

n：テキストの長さ
m：パターンの長さ

- クヌース-モリス-プラッツ法
  - 最大計算時間　$O(n)$
    - 前処理（表作成）　$O(m)$ 時間
    - 探索　$O(n)$ 時間

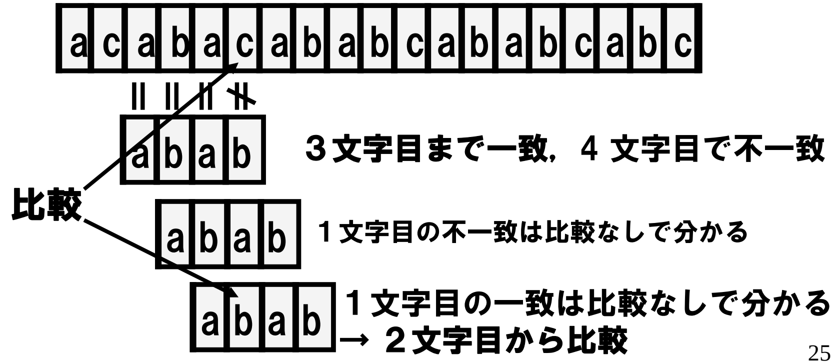
7.4 クヌース–モリス–プラッツ法

- 力まかせ法
  - 文字が不一致 →
    パターンのテキストでの位置を一つ進める



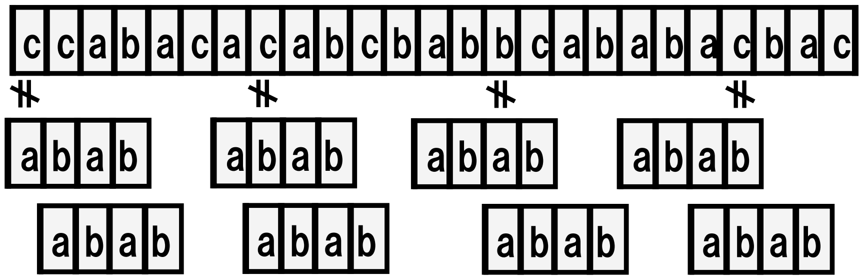
7.4 クヌース–モリス–プラッツ法

- クヌース–モリス - プラッツ法
  - 文字が不一致 →
    それまでの一致情報を利用して，
    パターンのテキストでの位置を一つ以上進



7.4 クヌース–モリス–プラッツ法

- パターンが $abab$ の場合



7.4 クヌース–モリス–プラッツ法

- 文字が不一致 →
  それまでの比較結果（一致情報）を利用して，
  パターンのテキストでの位置を一つ以上進める
- パターンが $abab$ の場合
  - 1 文字目で不一致 →1 ずらして，1 文字目から

  以下の場合は，テキストの不一致の文字から比較
  - 2 文字目で不一致 →（1 ずらして）1 文字目から
  - 3 文字目で不一致 →（2 ずらして）1 文字目から
  - 4 文字目で不一致 →（2 ずらして）2 文字目から
- 比較開始位置を前処理で表 $next$ に格納　$O(m)$ 時間
- 表 $next$ を利用して文字列照合　$O(n)$ 時間

7.4 クヌース–モリス–プラッツ法

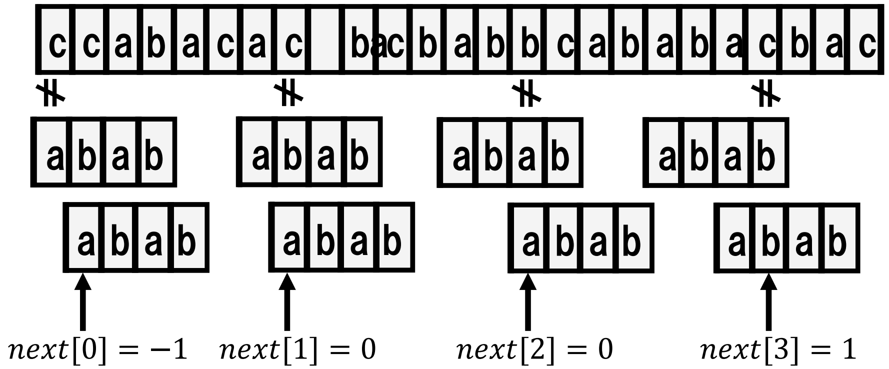
アルゴリズム７．３　クヌース-モリス-プラッツ法

// テキスト：$T[0]T[1]\ \ldots\ T[n-1]$
// パターン：$P[0]P[1]\ \ldots\ P[m-1]\quad(n \geq m)$
// 再開位置：$next[m]$　**（ただし，**$next[0]=-1$**）**

```
j=0;
for (i=0; i<n; i++) {
    while ((j>=0) && (T[i] != P[j]))
        j=next[j];
    if (j==m-1) return i-m+1; // パターンを検出した
    j++; };
return -1; // パターンが出現しなかった
```


7.4 クヌース–モリス–プラッツ法

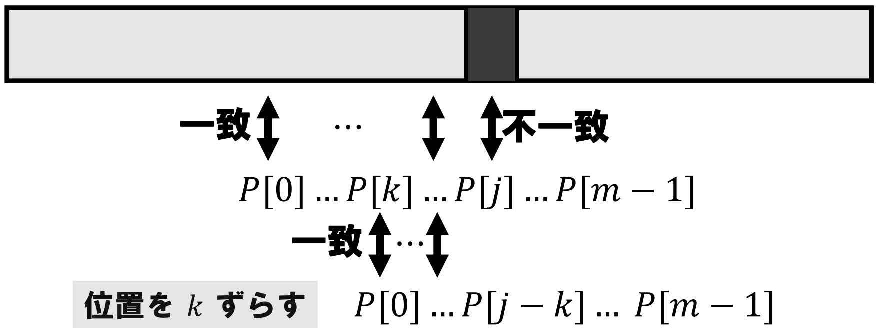
- **表** $next$ **：パターンが** $abab$ **の場合**



$next[0]=-1$ **： テキストの文字を１つ進め** $P[0]$ **と比較**
$next[i]=k$ **： テキストの同じ文字と** $P[k]$ **を比較**

7.4 クヌース–モリス–プラッツ法

- **表** $next$ **の意味**

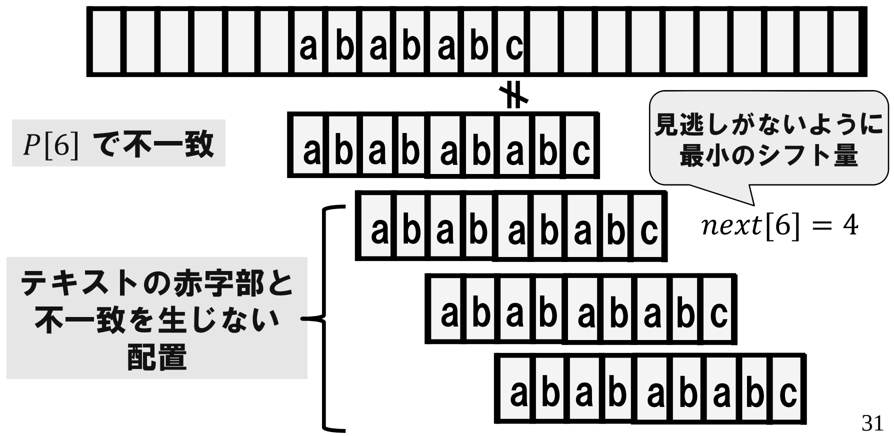


$next[j]=j-k$ **： テキストの同じ文字と** $P[j-k]$ **を比較**

$P[k]P[k+1]\ldots P[j-1]=P[0]P[1]\ldots P[j-k-1]$ **なる最小の** k

7.4 クヌース–モリス–プラッツ法

- **パターンが** $abababab c$ **の場合**
  - $next[6]$：$P[6]$ **で不一致が生じたとき，不一致が生じたテキストの文字と比較するパターンの文字**



7.4 クヌース-モリス-プラット法

アルゴリズム7．３　クヌース-モリス-プラット法

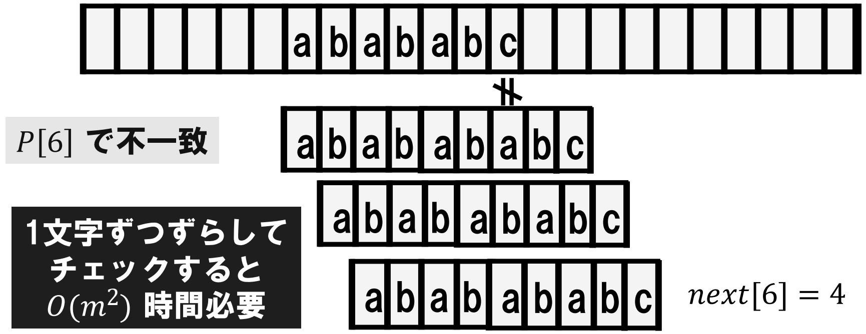
```
j=0;
for (i=0; i<n; i++) {
    while ((j>=0) && (T[i] != P[j]))
        j=next[j];
    if (j==m-1) return i-m+1; // パターンを検出した
    j++; };
return -1; // パターンが出現しなかった
```

時間計算量：$O(n)$

- while 条件不成立：i が増加（高々 n 回）
- while 条件成立：
  パターンをずらして，テキストの同じ文字と比較
  パターンは右にしか移動しないので，高々 n 回

7.4 クヌース-モリス-プラット法

- 表 $next$ の求め方
  - $next[i] = i - k$（k はシフト量）
    $P[k]P[k+1]\ldots P[i-1] = P[0]P[1]\ldots P[i-k-1]$
    なる最小の k



基本的には，パターンとパターンの文字列照合

7.4 クヌース-モリス-プラット法

- 前処理の表の計算　$O(m)$ 時間
  ＊単純な方法では $O(m^2)$ 時間

パターン：$P[0]P[1] \ldots P[m-1]$

```
// 再開位置：next[m] の計算
j=-1;
for (i=0; i<m; i++) {
    next[i]=j;
        while ((j>=0) && (P[i] != P[j]))
        j=next[j];
    j++;
};
```

パターンとパターンの文字列照合に
クヌース－モリス－プラット法を適用

$next[i]$ を i の昇順に求めているので可能

7.4 クヌース-モリス-プラット法

- 力まかせ法
  - 最大計算時間　$O(nm)$　　n：テキストの長さ
    m：パターンの長さ
- クヌース-モリス-プラット法
  - 最大計算時間　$O(n)$
    - 前処理（表作成）　$O(m)$ 時間
    - 探索　$O(n)$ 時間

7　文字列のアルゴリズム

7.1 文字列照合問題とは

7.2 力まかせ法

7.3 ラビン-カープ法

7.4 クヌース-モリス-プラッツ法

7.5 ボイヤー-ムーア法

7.5 ボイヤー-ムーア法

- 力まかせ法
  - 計算時間　$O(nm)$
- クヌース-モリス-プラッツ法
  - 計算時間　前処理　$O(m)$　探索　$O(n)$
- ボイヤー-ムーア法
  - 計算時間　前処理　$O(m)$　探索　$O(n)$
  - クヌース-モリス-プラッツ法より高速
  - 作戦１：パターンを右から照合，不一致情報を利用
  - 作戦２：パターンを右から照合，一致情報を利用

２種類の作戦の内，シフト量の大きい方を採用

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト	a	b	a	b	b	c	a	b	b	b	a	c	b	a	a	b	c
パターン	a	b	c														

比較回数：１

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト	a	b	a	b	b	c	a	b	b	b	a	c	b	a	a	b	c
パターン			a	b	c												

a に一致するように
パターンをずらす

比較回数：１

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

比較回数：２

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

b に一致するように
パターンをずらす

比較回数：２

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

比較回数：３

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

比較回数：４

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，
    パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

比較回数：５

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，
    パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

b に一致するように
パターンをずらす

比較回数：５

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，
    パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

比較回数：６

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，
    パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

a に一致するように
パターンをずらす

比較回数：６

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

比較回数：７

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

b に一致するように
パターンをずらす

比較回数：７

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

比較回数：８

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

b に一致するように
パターンをずらす

比較回数：８

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

比較回数：９

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

a に一致するように
パターンをずらす

比較回数：９

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

比較回数　10

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

b に一致するように
パターンをずらす

比較回数　10

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

比較回数　11

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

a に一致するように
パターンをずらす

比較回数　11

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

パターン　a b c

比較回数　12

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある

テキスト　a b a b b c a b b b a c b a a b c

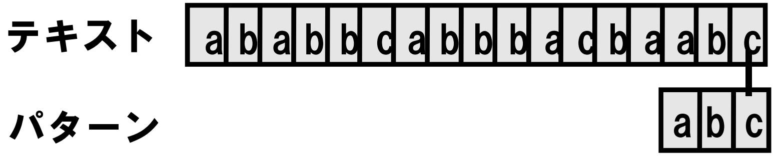
パターン　a b c

b に一致するように
パターンをずらす

比較回数　12

ボイヤー-ムーア法　作戦１

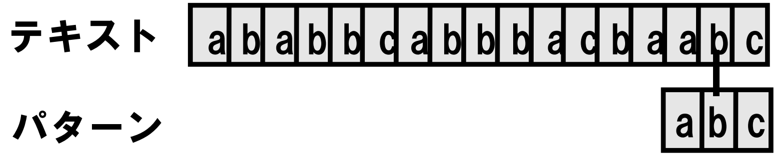
- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある



比較回数　13

ボイヤー-ムーア法　作戦１

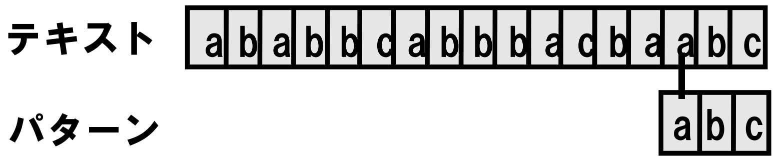
- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある



比較回数　14

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある



比較回数：15　テキスト長：17

ボイヤー-ムーア法　作戦１の効果

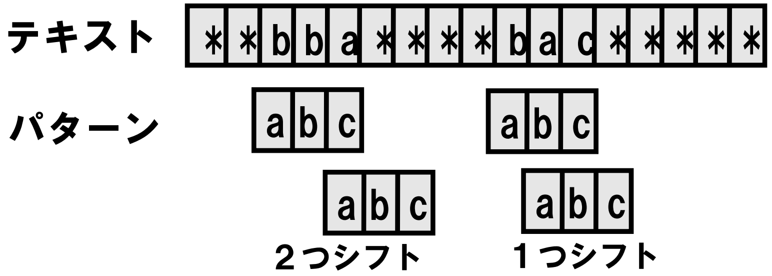
最も効果のある例

- テキスト：$aaaaaa \ldots a$　（n 文字）
- パターン：$bbb \ldots b$　（m 文字）

n/m 回の文字比較で終了

ボイヤー-ムーア法　作戦１

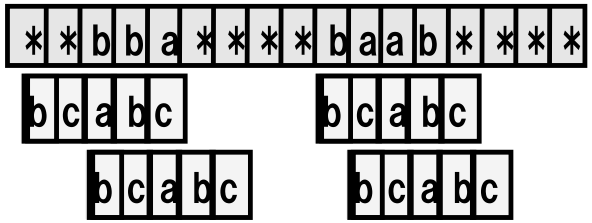
- 不一致情報を利用して，パターンの位置を進める
  - パターンの位置をいくつ進めるか



●パターンの比較位置の前で最後の不一致文字 a を
　テキストの不一致文字に合わせるようにシフト量を決定
●不一致のテキスト文字 a は同じでも，
　パターンの位置によってシフト量が異なる

ボイヤー-ムーア法　作戦１

- 不一致情報を利用して，パターンの位置を進める
  - パターンの比較位置の前で最後の不一致文字を
    テキストの不一致文字に合わせるようにシフト
    → サイズ σm のシフト表（σ：文字種類数）
  - サイズ σ の表を代用
    - ●パターン $P[0..m-2]$ で最後の不一致文字の位置



表を利用して２つシフト　表を利用できずに１つシフト

ボイヤー-ムーア法　作戦１

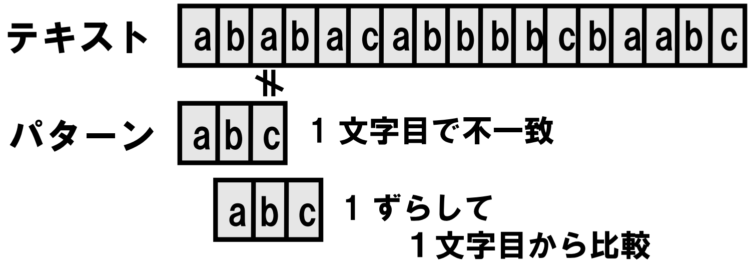
- 不一致情報を利用して，パターンの位置を進める
  - サイズ σ の表 $shift1$ を使用
    - ●$shift1[x]$　（x：文字）
      - パターン $P[0..m-2]$ に現れる最後の x の位置
        - ・パターン $bcabc$
          - $shift1[a]=2$，$shift1[b]=3$，$shift1[c]=1$，$shift1[d]=-1$
      - $P[j]$ とテキスト文字 x との比較で不一致
        ⇒ $shift1[x]<j$ なら，
        パターンを $j-shift1[x]$ 個シフトする

ボイヤー-ムーア法　作戦１

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，その情報（不一致情報）を利用して，パターンの位置を１つ以上進める
    ⇒ テキストのすべての文字を調べないことがある
- 早く不一致が見つかるほど有利！
- 計算時間
  - 前処理（表の作成）　$O(\sigma+m)$ 時間
    σ：文字種類数（定数）
  - 探索　最大 $O(mn)$ 時間
    最小 $O(\frac{n}{m})$ 時間

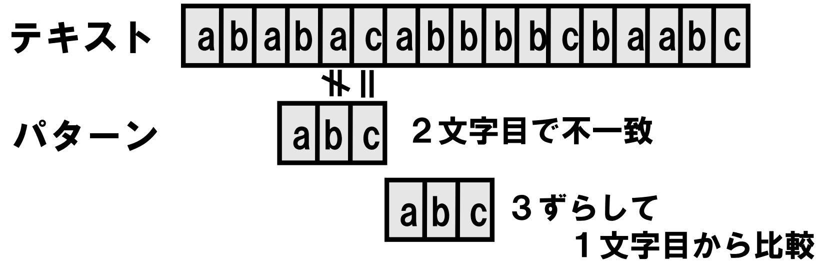
ボイヤー-ムーア法　作戦２

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，それまでの情報（一致情報）を利用して，パターンの位置を１つ以上進める（クヌース-モリス-プラッツ法と同じアイデア）



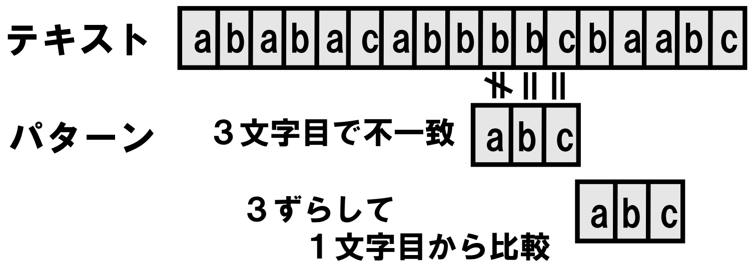
ボイヤー-ムーア法　作戦２

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，それまでの情報（一致情報）を利用して，パターンの位置を１つ以上進める（クヌース-モリス-プラッツ法と同じアイデア）



ボイヤー-ムーア法　作戦２

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，それまでの情報（一致情報）を利用して，パターンの位置を１つ以上進める（クヌース-モリス-プラッツ法と同じアイデア）



ボイヤー-ムーア法　作戦２

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，それまでの情報（一致情報）を利用して，パターンの位置を１つ以上進める（クヌース-モリス-プラッツ法と同じアイデア）
- パターンが abc の場合
  - 1 文字目で不一致 →1 ずらして， 1 文字目から
  - 2 文字目で不一致 →3 ずらして， 1 文字目から
  - 3 文字目で不一致 →3 ずらして， 1 文字目から

ボイヤー–ムーア法　作戦２

- テキストを左から探索
- パターンとの照合は右から
  - 不一致のときは，それまでの情報（一致情報）を利用して，パターンの位置を１つ以上進める（クヌース–モリス–プラッツ法と同じアイデア）
- 計算時間（クヌース–モリス–プラッツ法と同じ）
  - 前処理（表 $shift2$ の作成）　$O(m)$ 時間
  - 探索　$O(n)$ 時間
- 実際には，クヌース–モリス–プラッツ法より高速

文字列照合（まとめ）

- テキストからパターンを探索
  - テキスト：長さ n の文字列
  - パターン：長さ m の文字列（$n \gg m$）
- 力まかせ法
  - 計算時間　$O(nm)$
- クヌース–モリス–プラッツ法
  - 計算時間　　前処理　$O(m)$　　探索　$O(n)$
- ボイヤー–ムーア法
  - 計算時間　　前処理　$O(m)$　　探索　$O(n)$
  - クヌース–モリス–プラッツ法より高速

この章の学習目標（振返り）

- 文字列照合問題とは何か，応用例を用いて説明できる
- 力まかせ法とは何か，また，その時間計算量を説明できる
- ハッシュを利用した文字列照合アルゴリズムのアイデアとその時間計算量を説明できる
- 線形時間の文字列照合アルゴリズムのアイデアとその時間計算量を説明できる
- 文字列照合アルゴリズムのプログラムを書ける

データ構造とアルゴリズム
第13回（2020.1.27）

1. アルゴリズムの重要性
2. 探索問題
3. 基本的なデータ構造
4. 動的探索問題とデータ構造
5. データの整列
6. グラフのアルゴリズム
7. 文字列のアルゴリズム
8. アルゴリズム設計手法

8　アルゴリズム設計手法

8.1 再帰

8.2 分割統治法

8.3 動的計画法

8.4 グリーディ法

8.5 分枝限定法

8.6 線形計画法

この章の学習目標

- 学習したアルゴリズム設計法（定石）について，適用例を用いて説明できる
- 与えられた問題に対するアルゴリズムを設計するとき，どのアルゴリズム設計法を適用できるか判断できる
- 学習したアルゴリズム設計法を，自身がアルゴリズムを設計するときに活用できる

8.1 再帰

- 数学的帰納法と同じ原理
- 入力サイズ n の問題を解くアルゴリズム
  - 入力サイズ $n' \ (< n)$ の同じ問題を解くアルゴリズムを利用（数学的帰納法の帰納仮定と機能段階に相当）
  - 入力サイズが小さな定数のときのアルゴリズムは自明（数学的帰納法の基底に相当）
  - 漸化式を用いて計算時間を評価できる

ユークリッドの互除法

- 最古のアルゴリズムと言われている
  - 紀元前300年頃．ユークリッドの「原論」
- ２つの自然数 m と n の最大公約数 $\mathrm{gcd}(m,n)$ を求める
- $m \geq n > 0$ のとき， $\mathrm{gcd}(m,n) = \mathrm{gcd}(n,\ m \bmod n)$ を利用
  - $m \geq n,\ n > m \bmod n$ → 小さいサイズの問題を利用
- アルゴリズム
  - $\mathrm{r} = m \bmod n$ とする（$r < n$）
    - $r = 0$ ならば n が $\mathrm{gcd}(m,\ n)$
    - $r \neq 0$ ならば $\mathrm{gcd}(n,\ r)$ を求める（再帰）

ユークリッドの互除法：計算時間

- $T(m,n)$：$\mathrm{gcd}(m,n)$ を求めるのに要する時間
- $T(m,n) = T(n, m \bmod n) + O(1)$ （$m \geq n$ を仮定）
- $T(m,0) = O(1)$
  - ２回再帰を呼び出すと各引数の値は $1/2$ 以下になる
  - $T(m,n) \leq T(m/2, n/2) + O(1)$

    → $T(m,n) = O(\log n)$ （$m \geq n$ を仮定）

詳細はテキストで確認すること

８　アルゴリズム設計手法

8.1 再帰

8.2 分割統治法

8.3 動的計画法

8.4 グリーディ法

8.5 分枝限定法

8.6 線形計画法

8.2 分割統治法 − 原理

- 分割統治法の原理
  1. 分割：問題をいくつかの部分問題に分割
  2. 統治：各部分問題を再帰的に解く
     部分問題サイズが小さければ直接解く
  3. 統合：部分問題の解を統合して全体の解を構成
- 例：マージソート，クイックソート，サンプルソート
- 特長
  - 効率のよい分割法，合成法があれば有効
  - 計算量の解析が容易（漸化式）
  - プログラミング容易（再帰的構造）

分割統治法の例：充足可能性問題

- 準備
  - （論理）変数　$X = \{x_1, x_2, \ldots, x_n\}$
  - リテラル　$x_i, \neg x_i$（$1 \le i \le n$）
  - 節　リテラルを論理和 $+$ で結合したもの
    $x_2 + \neg x_4 + x_5$
  - 式　節を論理積 $\cdot$ で結合した論理式
    $f(x_1, x_2, \ldots, x_n)$
    $(\mathrm{x}_2 + \neg x_4 + x_5) \cdot (\neg x_1 + x_3) \cdot (x_1 + \neg x_5)$
  - k-式　各節のリテラルが k 個以下の式

SAT：充足可能性問題

- 入力
  - 式（k-SAT では k-式）
    $(\mathrm{x}_1 + x_2 + x_3) \cdot (\neg x_1 + \neg x_3) \cdot (\neg x_2 + x_3)$
- 出力
  - 式の値を真にするような，
    変数への真偽割当が存在するか？
    上の例では，YES
    （$x_1 = true$，$x_2 = false$，$x_3 = false$）

SAT：充足可能性問題

- SAT 問題は NP-完全
  - 多項式時間で解けそうにない
  - 3-SAT に限定しても NP-完全
  - 2-SAT は多項式時間で解ける
- 力まかせ法
  - 変数へのすべての真偽割当を試行
  - 3-SAT の場合　$O(m2^n)$ 時間
    n：変数の数　　m：節の数

3-SAT の分割統治法（1）

- 計算時間　$O(m(1.84)^n)$
  n：変数の数　　m：節の数
- 準備
  - f：3-式（入力）
  - ℓ：f に現れるリテラル
  - $f(\ell = 1)$：式 f に $\ell = 1\ (true)$ を代入して得られる式
    - $(\mathrm{x}_1 + x_2 + x_3) \cdot (\neg x_1 + \neg x_3) \cdot (\neg x_2 + x_3)$
    - $f(x_1 = 1) = (1 + x_2 + x_3) \cdot (0 + \neg x_3) \cdot (\neg x_2 + x_3)$
      $= 1 \cdot (\neg x_3) \cdot (\neg x_2 + x_3) = (\neg x_3) \cdot (\neg x_2 + x_3)$
  - $f(\ell = 0)$：式 f に $\ell = 0\ (false)$ を代入して得られる式
  - $f(\ell_1 = 0,\ \ell_2 = 1)$ なども同様に定義

3-SAT の分割統治法（2）

アルゴリズム 8.4 – 分割統治法を用いた 3-SAT

3-SAT-DC(f)

if （f の変数の数 ≦ 3）∨（f の節数 ≦ 2） ｛
すべての真偽割当を試行して判定
｝
C $= f$ 中でリテラル数最小の節；
if（$C == (\ell)$）　3-SAT-DC($f(\ell = 1)$)
else if（$C == (\ell_1 + \ell_2)$）
return 3-SAT-DC($f(\ell_1 = 1)$) + 3-SAT-DC($f(\ell_1 = 0, \ell_2 = 1)$)；
else /* $C == (\ell_1 + \ell_2 + \ell_3)$ */
return 3-SAT-DC($f(\ell_1 = 1)$) + 3-SAT-DC($f(\ell_1 = 0, \ell_2 = 1)$)
+ 3-SAT-DC($f(\ell_1 = 0, \ell_2 = 0, \ell_3 = 1)$)；

3-SAT の分割統治法 ：計算時間

- $T(n, m)$：n 変数，m 個の節の式 f に対する計算時間
- $T(n, m) \le T(n-1, m-1) + T(n-2, m-1)$
  $+T(n-3, m-1) + 54m$
  - $T(n-1, m-1)$： 3-SAT-DC($f(\ell_1 = 1)$) の計算時間
    - ℓ_1 の変数の値が決まる（変数の数が 1 減少）
    - 節 C が true になる（節の数が 1 減少）
  - $T(n-2, m-1)$： 3-SAT-DC($f(\ell_1 = 0, \ell_2 = 1)$) の計算時間
  - $T(n-3, m-1)$： 3-SAT-DC($f(\ell_1 = 0, \ell_2 = 0, \ell_3 = 1)$) の計算時間
  - $54m$：f から $f(\ell_1 = 1), f(\ell_1 = 0, \ell_2 = 1)$, $f(\ell_1 = 0, \ell_2 = 0, \ell_3 = 1)$ の構成に要する時間
- 上の漸化式から $T(n, m) = O(m(1.84)^n)$

詳細はテキストで確認すること

8　アルゴリズム設計手法

8.1 再帰
8.2 分割統治法
8.3 動的計画法
8.4 グリーディ法
8.5 分枝限定法
8.6 線形計画法

8.3 動的計画法

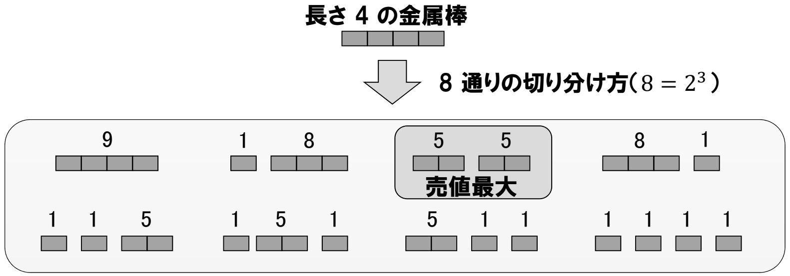
- 動的計画法の原理
  - 部分問題に分割して解く
  - サイズの小さい問題から順に解く
  - 結果を表に残して，再計算のムダを省く
- 動的計画法の例
  - 全頂点対最短路問題のフロイドのアルゴリズム
  - フィボナッチ数列（テキスト）
  - ナップサック問題（テキスト）

ミニレポート課題 (1)

- 長さ n の金属棒をロッドに切り分ける
  - ロッドの売値 $c(l)$ は長さ l によって異なる
- ロッドの売値の合計が最大となる切り分け方を求める

ロッドの売値

長さ l	1	2	3	4	5	6	7	8	9	10
売値 $c(l)$	1	5	8	9	10	17	17	20	24	30



8　アルゴリズム設計手法

8.1 再帰
8.2 分割統治法
8.3 動的計画法
8.4 グリーディ法
8.5 分枝限定法
8.6 線形計画法

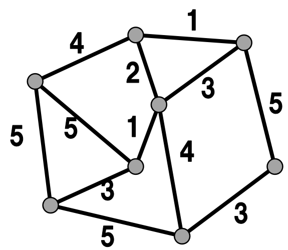
8.4 グリーディ法

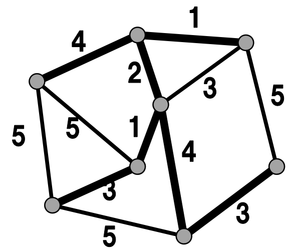
- グリーディ法の原理
  - 各時点で局所的に最良のものを選択
  - 最適解が得られるかどうかは問題による
    - マトロイドと密接に関連
- グリーディ法の例
  - 単一始点最短路問題のダイクストラのアルゴリズム
  - 硬貨の交換問題（最小の枚数の硬貨で支払いたい）

グリーディ法の例：最小全域木問題

最小全域木問題

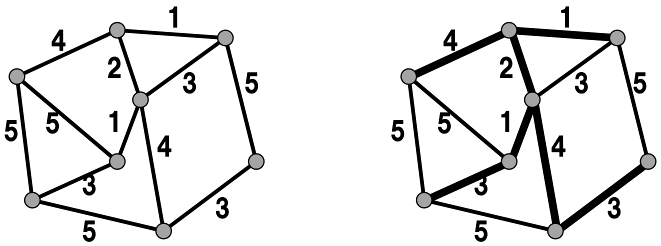
- 入力：重みつき無向グラフ　$G = (V, E, c)$
  - c：各辺に実数の重みを与える関数 $c : E \to \mathbb{R}$
- 出力：G の重みの和最小の全域木
  - 全域木：すべての頂点を含む木状の部分グラフ





最小全域木アルゴリズム

- グリーディ法（Kruskal のアルゴリズム）
  - 重みの小さい順に辺を木に追加
  - ただし，サイクルを生じないように



最小全域木アルゴリズム：計算時間 (1)

- グリーディ法
  - 重みの小さい順に辺を木に追加
  - ただし，サイクルを生じないように
- 計算時間　$O(m \log m)$　　m：辺数
  - 重みの小さい順に辺を選ぶ（全体で $O(m \log m)$ 時間）
    - ヒープ（プライオリティキュー）を使用（ソートでもよい）
      - ヒープへの挿入：各辺 $O(\log m)$ 時間
      - 重み最小の辺を抽出＋ヒープの再構成：$O(\log m)$ 時間

最小全域木アルゴリズム：計算時間 (2)

- グリーディ法
  - 重みの小さい順に辺を木に追加
  - ただし，サイクルを生じないように
- 計算時間　$O(m \log m)$　　m：辺数
  - サイクルを生じない辺のみ追加（全体で $O(m \log m)$）
    - union-find 操作を平衡２分木で実現
      - make-set (v)：v だけからなる集合を作る
        - ・最初に，各頂点 v に対し，集合 $\{v\}$ を作る
      - find (v)：v を含む集合（名）を返す
        - ・辺 (u, v) は，頂点 u, v が異なる集合（これまでの追加辺で連結している頂点の集合）に属するときのみ追加
      - union (u, v)：u を含む集合と v を含む集合の和集合を作る
        - ・辺 (u, v) の追加により，２つの頂点集合が連結される

８　アルゴリズム設計手法

8.1 再帰

8.2 分割統治法

8.3 動的計画法

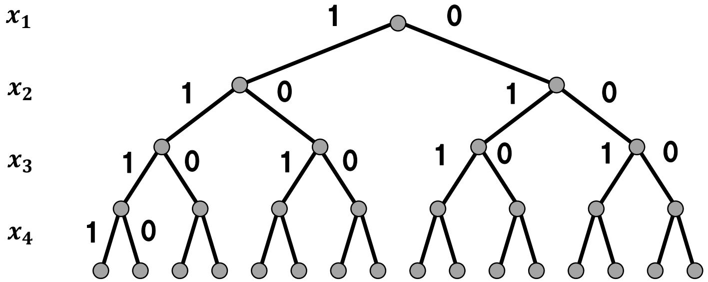
8.4 グリーディ法

8.5 分枝限定法

8.6 線形計画法

8.5 分枝限定法

- 分枝限定法の原理
  - 分枝：総当り（木の全探索）
  - 限定：解でないと分かった時点でそれ以降の探索を省略
- SAT の探索木：変数への割当を表す木



分枝限定法：MAX-SAT (1)

- 入力
  - 式：論理式（SAT と同じ）
- 出力
  - 真となる節の数が最大となる，変数への真偽割当
- 例：$f(x_1, x_2, x_3, x_4)$

$$= (x_1 + \neg x_3) \cdot (x_1 + x_2 + \neg x_4) \cdot (\neg x_1 + x_3) \cdot (x_1 + \neg x_2 + x_4) \cdot (x_2 + x_3 + \neg x_4) \cdot (x_1 + \neg x_2 + \neg x_4) \cdot (x_2) \cdot (x_1 + x_4) \cdot (\neg x_1 + \neg x_2) \cdot (x_1)$$

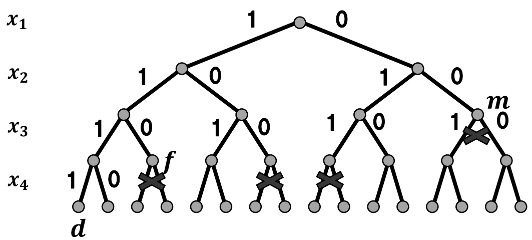
  - $x_1 = x_2 = x_3 = x_4 = false$： $(x_2), (x_1 + x_4), (x_1)$ 以外の７つの節が真
  - $x_1 = x_2 = true,\ x_3 = x_4 = false$：$(\neg x_1 + x_3), (\neg x_1 + \neg x_2)$ 以外の８つの節が真
  - $x_1 = x_2 = x_3 = x_4 = true$： $(\neg x_1 + \neg x_2)$ 以外の９つの節が真 ← 解

分枝限定法：MAX-SAT (2)

- $f(x_1, x_2, x_3, x_4)$

$$= (x_1 + \neg x_3) \cdot (x_1 + x_2 + \neg x_4) \cdot (\neg x_1 + x_3) \cdot (x_1 + \neg x_2 + x_4) \cdot (x_2 + x_3 + \neg x_4) \cdot (x_1 + \neg x_2 + \neg x_4) \cdot (x_2) \cdot (x_1 + x_4) \cdot (\neg x_1 + \neg x_2) \cdot (x_1)$$

- 探索木を深さ優先探索で探索
  - d $(x_1 = x_2 = x_3 = x_4 = 1)$ で９つの節が真（1つの節だけが偽）
  - f $(x_1 = x_2 = 1,\ x_3 = 0)$ で２つの節が偽 → これ以上探索しない
  - m $(x_1 = x_2 = 0)$ で２つの節が偽 → これ以上探索しない



8　アルゴリズム設計手法

8.1 再帰
8.2 分割統治法
8.3 動的計画法
8.4 グリーディ法
8.5 分枝限定法
8.6 線形計画法

8.6 線形計画法

- 線形計画法
  - 変数：$x_1, x_2, \ldots, x_n$
  - 制約：変数の線形不等式

$$x_1 + 2x_2 + 5x_3 \leq 5$$
$$2x_1 - 4x_2 + 3x_3 \leq 2$$

  - 目的：線形目的関数の最大化

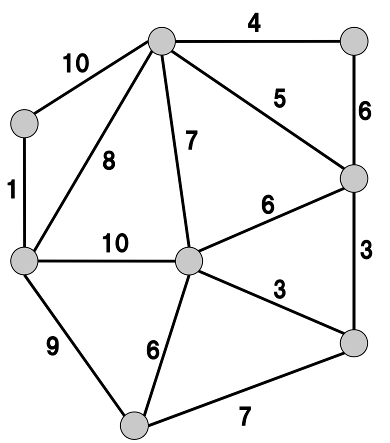
$x_1 + 2x_2 + 3x_3$ の最大化

線形計画法：シンプレックス法

- シンプレックス法
  - 多くの実際の問題を高速に解ける
  - 最悪の場合は指数時間
- 線形計画法を解く，多項式時間のアルゴリズムも存在
- 変数の値を整数に限定（整数計画法）は NP-困難
  - 多項式時間のアルゴリズムは知られていない
  - 変数の値を 0, 1 に限定（0, 1-計画法）も NP-困難

ミニレポート課題

下図のグラフの最小全域木を求めなさい



8　アルゴリズム設計手法

8.1 再帰

8.2 分割統治法

8.3 動的計画法

8.4 グリーディ法

8.5 分枝限定法

8.6 線形計画法