

# アルゴリズム 計算量

森下

教科書 Robert Sedgewick: “Algorithms in C” Addison Wesley  
(邦訳 近代科学社)

Homepage 講義資料と講義で使う Java program

<http://www.gi.k.u-tokyo.ac.jp/~moris/lecture/upbsb/2004-system/>

# メニュー

生物情報 情報システム概論 森下

## 基礎

- 1 はじめに
- 2 C
- 3 基本データ構造
- 4 木
- 5 再帰呼出し
- 6 アルゴリズムの解析
- 7 アルゴリズムの実現

## 整列

- 8 初等的な整列法
- 9 クイックソート
- 10 基数整列法
- 11 順序キュー
- 12 マージソート

## 探索

- 16 ハッシュ表
- 文字列処理
- 19 文字列探索

## 生物情報科学実験2の課題との関係について触れる

- shotgun sequencing: 生物情報科学実験I で解読された配列についてソフトウェアを利用してつなげ評価したのちに、大規模塩基配列決定の仕組みを理解するためのプログラム製作を行う。
- タンパク質発現プロファイル: プロテインチップで収集した断片からデータベースを利用して本来のタンパク質を同定する作業を行った後に、データベースを検索するプログラムを作成する。
- DNA チップデータ解析: DNA チップを用いて観測された遺伝子発現量データを、機械学習手法(クラスタリング、クラス分類など)を用いて解析する。さらに初歩的な機械学習のプログラムを作成する。

# 2章 C言語⇒Java言語

# 最大公約数の計算 ユークリッドの互助法

生物情報 情報システム概論 森下

$u > v$  ならば  $(u \text{ と } v \text{ の最大公約数}) = (v \text{ と } u - v \text{ の最大公約数})$

```
public class Gcd {
    public static int gcd(int u, int v) {
        int t;
        while (u > 0) {
            if (u < v) {
                t = u;
                u = v;
                v = t;
            }
            u = u - v;
        }
        return v;
    }

    public static void main(String[] args) {
        System.out.println(gcd(36, 24));
        System.out.println(gcd(36, 45));
    }
}
```

# 3章 基本データ構造

# 配列 エラトステネスのふるい

|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| a | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|   | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
|   | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |
|   | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  |

```
public class Eratosthenes {
    private static final int N=50;    // N=50 までの素数を出力

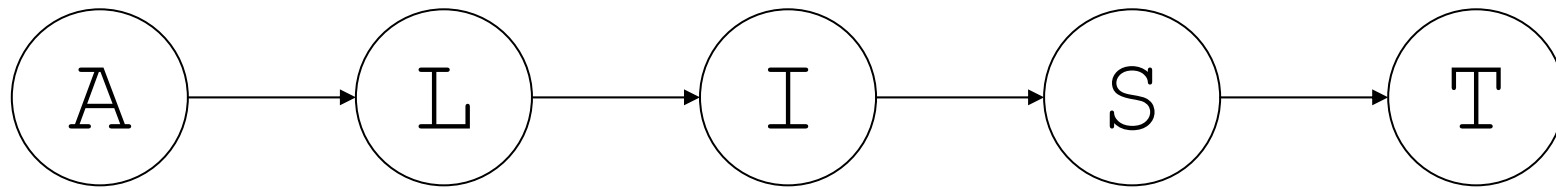
    public static void main(String[] args) {
        int[] a = new int [N+1];    // N+1の長さの配列を定義
        for(int i=1; i<=N; i++){    // i++ は i=i+1;
            a[i]=1;                    // 1 は素数であることを意味
        }                            // 最初は全て素数として初期化
        for(int i=2; i<=N/2; i++){
            for(int j=2; j<=N/i; j++){
                a[i*j]=0;            // 合成数であることがわかると 0
            }
        }
        for(int i=1; i<=N; i++){
            if(a[i]>0){
                System.out.print(i+" ");
            }
        }
    }
}
```

# 配列のインデックスの範囲に注意

```
public class Eratosthenes {  
    private static final int N=50;           // N=50 までの素数を出力  
  
    public static void main(String[] args) {  
        int[] a = new int [N+1];           // N+1の長さの配列を定義  
        for(int i=1; i<=N; i++){  
            a[i]=1;                         // 1 は素数であることを意味  
        }                                  // 最初は全て素数として初期化  
        for(int i=2; i<=N/2; i++){  
            for(int j=2; j<=N; j++){        // バグ  
                a[i*j]=0;                   // N+1 < N/2 * N  
            }  
        }  
        for(int i=1; i<=N; i++){  
            if(a[i]>0){  
                System.out.print(i+" ");  
            }  
        }  
    }  
}
```



# リンクを使ったリスト表現

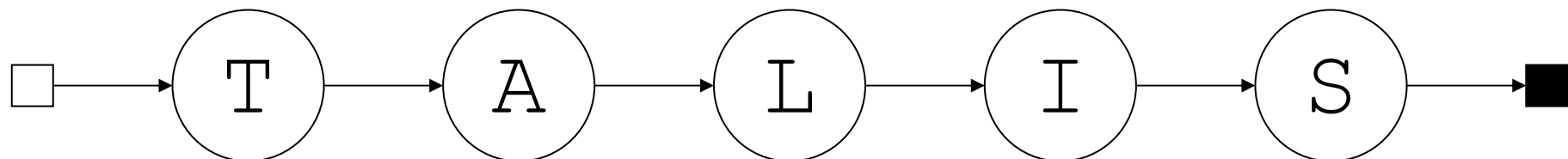
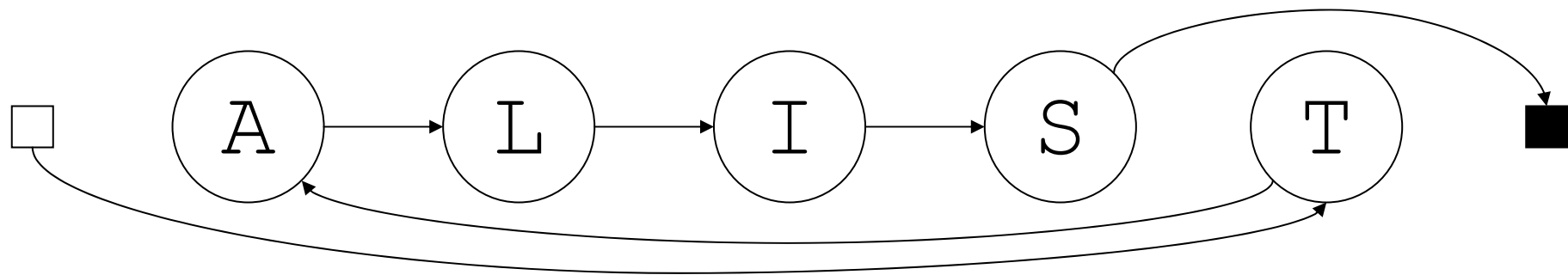
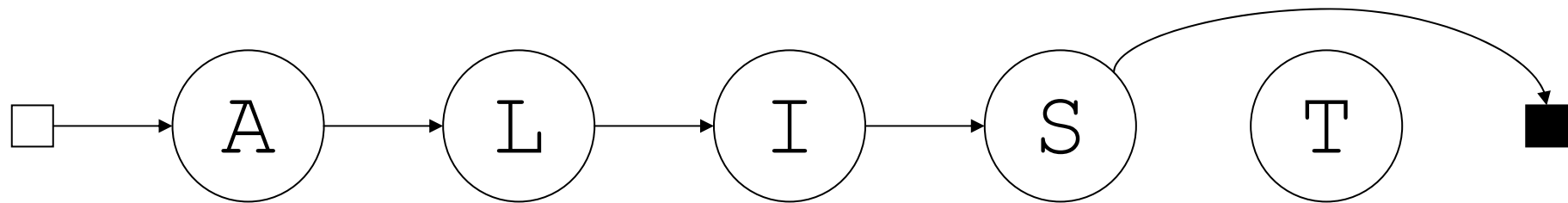
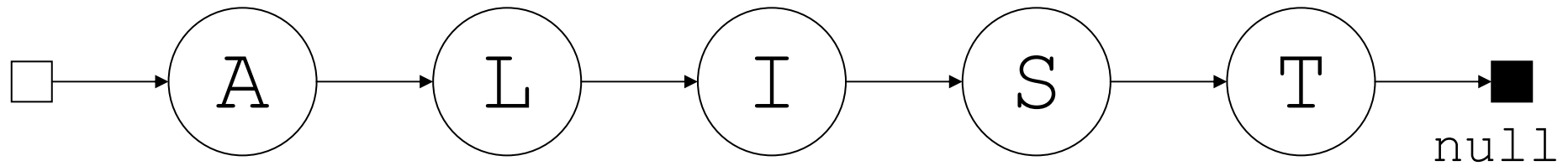


利点 実行中にデータ量に応じて伸縮できる  
前もってデータ量を知らなくてよい

項目の並べ替えが楽

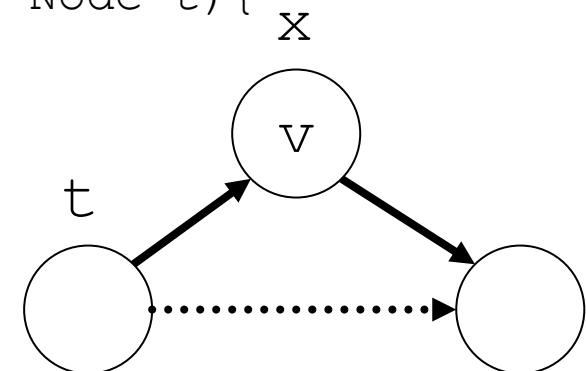
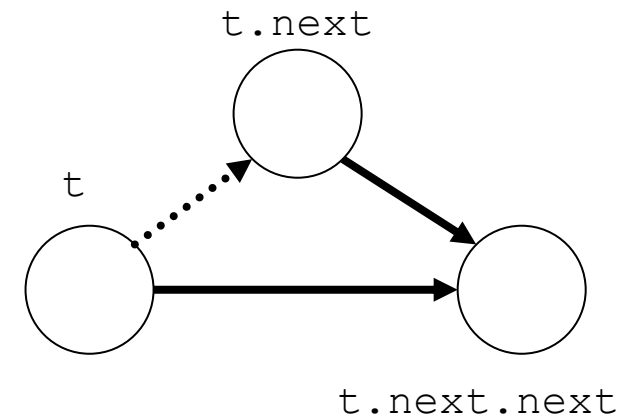
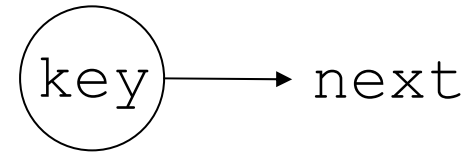
欠点 k 番目の項目を見つける操作  
直前の項目を見つける操作

# リンクを使ったリスト表現



# リスト処理の実装

```
public class Node {  
    public int key;  
    public Node next;  
  
    public Node(int n){  
        key = n;  
        next = null;  
    }  
  
    public void deleteNext(Node t){  
        if(t.next != null){  
            t.next = t.next.next;  
        }  
    }  
  
    public static void insertAfter(int v, Node t){  
        Node x = new Node(v);  
        x.next = t.next;  
        t.next = x;  
    }  
:  
}
```

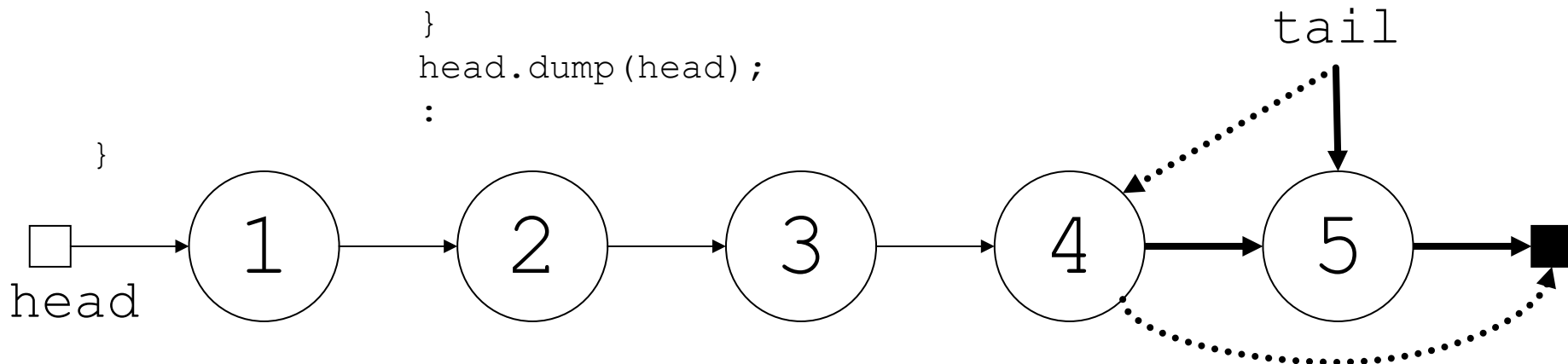


# リスト処理の実装

生物情報 情報システム概論 森下

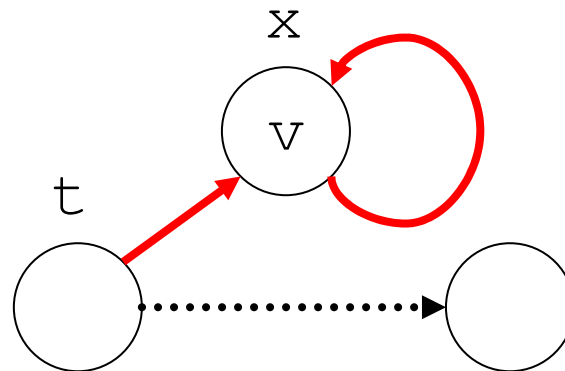
```
:  
public void dump(Node t){  
    System.out.println("Dumping");  
    for(Node x = t; x != null; x = x.next){  
        System.out.print(x.key+" ");  
    }  
    System.out.println();  
}
```

```
public static void main(String[] args) {  
    Node head = new Node(0);  
    Node tail = head;  
    for(int i = 1; i < 20; i++){  
        insertAfter(i, tail);  
        tail = tail.next;  
    }  
    head.dump(head);  
    :  
}
```



# ポインターのつけかえ順序に注意

```
public Node insertAfter(int v, Node t) {  
    Node x = new Node(v);  
    t.next = x;  
    x.next = t.next;  
    return x;  
}
```

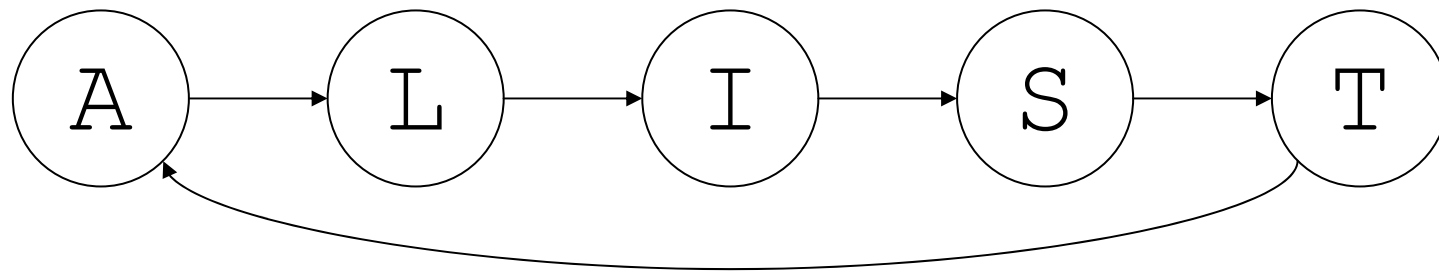


# リストの拡張

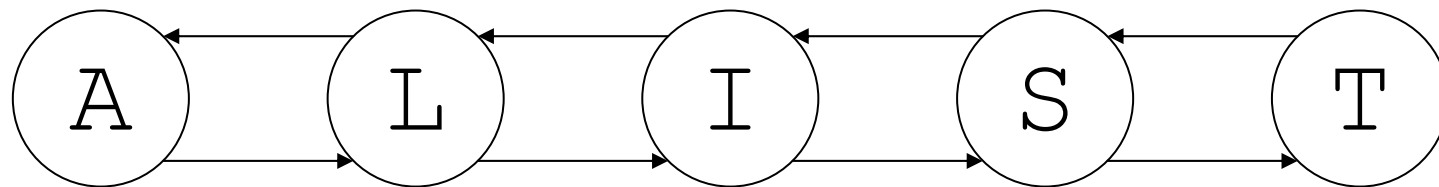
最初と最後に特殊なノードは必要か？

直前の項目を見つける操作が困難

循環リスト



両方向リスト

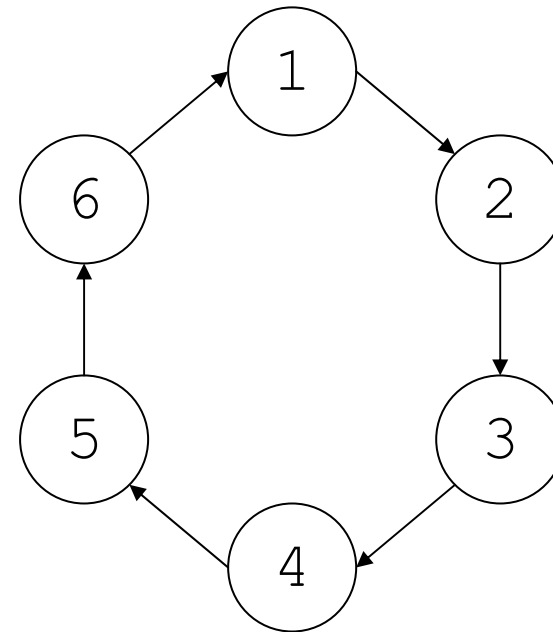


# ジョセファスの問題

N個のノードを円周上に配置

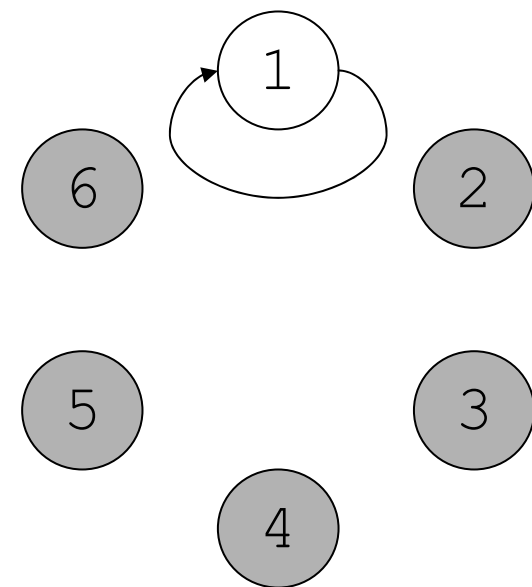
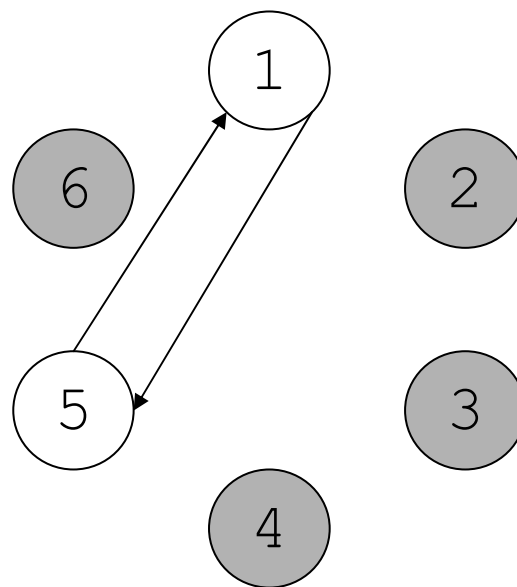
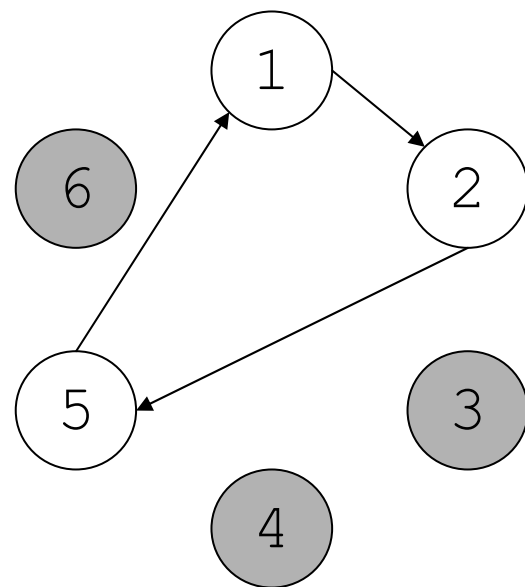
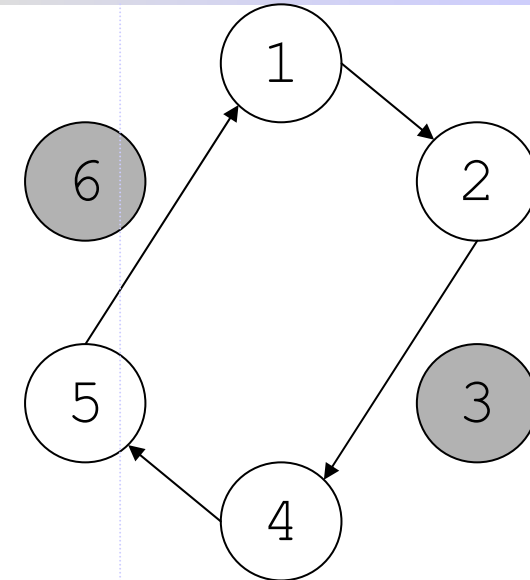
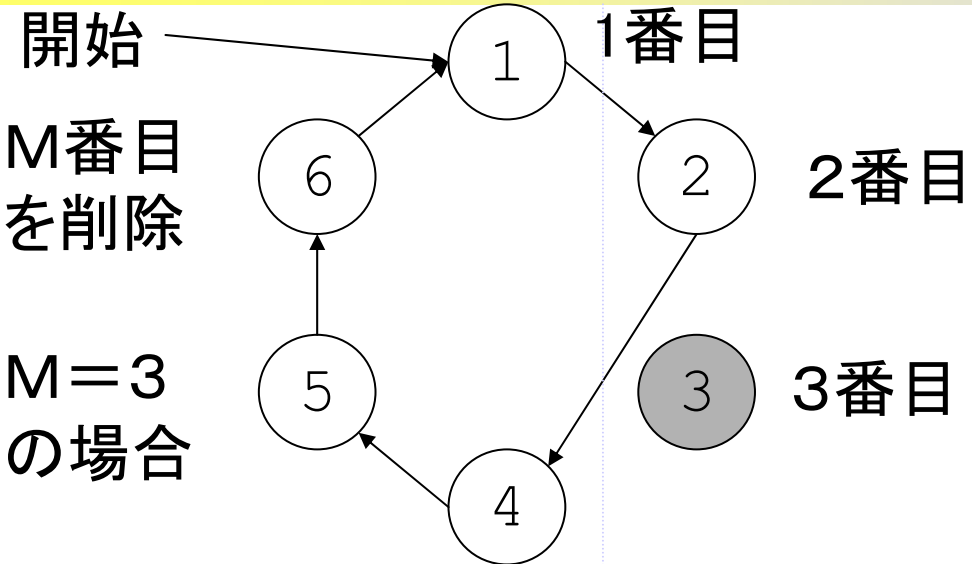
M番目ごとにノードを除去

最後に残るノードは何か？



# ジョセファスの問題

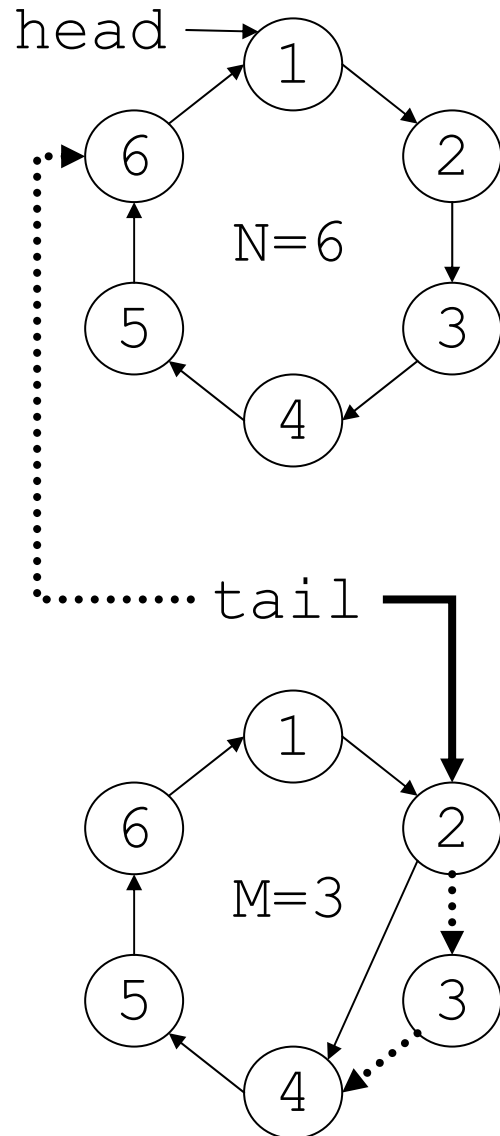
生物情報 情報システム概論 森下





# 循環リストとジョセファスの問題

生物情報 情報システム概論 森下



tail の次を削除

```
int n = 6;  
int m = 3;
```

```
head = new Node(1);  
tail = head;  
for(int i = 2; i <= n; i++){  
    insertAfter(i, tail);  
    tail = tail.next;  
}  
tail.next = head;  
  
while(tail != tail.next){  
    for(int i = 1; i < m; i++){  
        tail = tail.next;  
    }  
    System.out.println("Delete " + tail.next.key);  
    tail.next = tail.next.next;  
}  
System.out.println("Answer = " + tail.key);
```

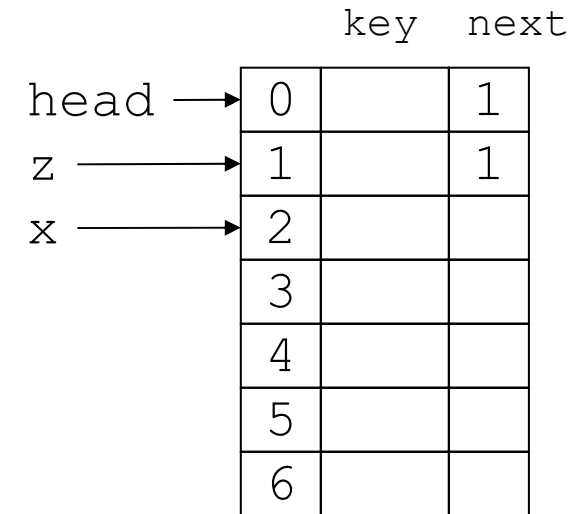
# 配列を使ったリストの実装

```
public class Node {
    private char[] key;
    private int[] next;
    private int head, z, x; // z は null に相当

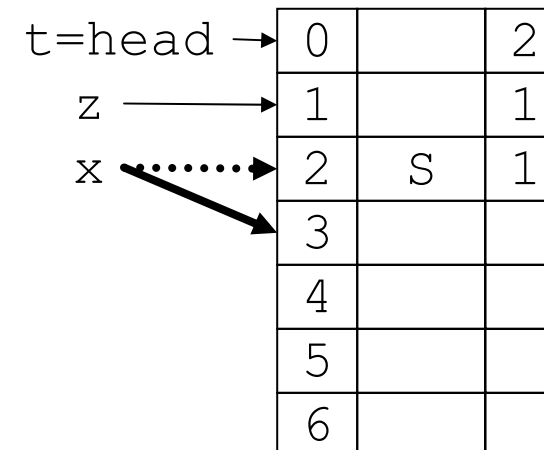
    public Node(int size) {
        key = new char[size + 2];
        next = new int[size + 2];
        head = 0;
        z = 1;
        x = 2;
        next[head] = z;
        next[z] = z;
    }

    public void deleteNext(int t) {
        next[t] = next[next[t]];
    }

    public int insertAfter(char v, int t) {
        key[x] = v;
        next[x] = next[t];
        next[t] = x;
        return x++;
    }
}
```



insertAfter('S', head)



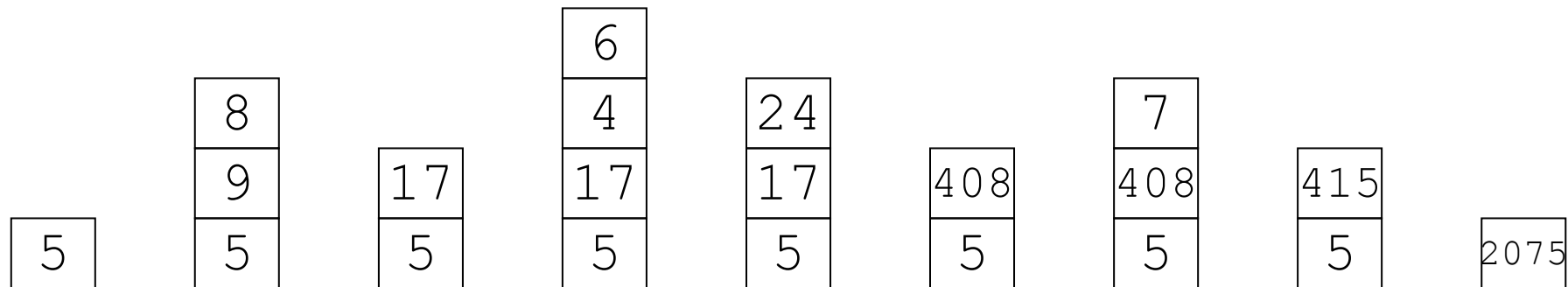
# 配列を使ったリストの実装

```
public static void main(String[] args) {  
    Node l = new Node(5);  
    int t1 = l.insertAfter('S', l.head);  
    int t2 = l.insertAfter('L', l.head);  
    l.insertAfter('A', l.head);  
    l.insertAfter('I', t2);  
    l.insertAfter('T', t1);  
    l.printList();  
}
```

|      |   |   |   |    |   |   |    |   |   |    |   |   |
|------|---|---|---|----|---|---|----|---|---|----|---|---|
| head | 0 |   | 2 | t2 |   | 3 | t2 |   | 4 | t1 |   | 4 |
|      | 1 |   | 1 |    |   | 1 |    |   | 1 |    |   | 1 |
|      | 2 | S | 1 |    | S | 1 |    | S | 1 |    | S | 6 |
|      | 3 |   |   |    | L | 2 |    | L | 5 |    | L | 5 |
|      | 4 |   |   |    |   |   |    | A | 3 |    | A | 3 |
|      | 5 |   |   |    |   |   |    |   |   |    | I | 2 |
|      | 6 |   |   |    |   |   |    |   |   |    | T | 1 |

# スタック

$$5 * ( ( (9 + 8) * (4 * 6) ) + 7 )$$



```
push (5) ;  
push (9) ;  
push (8) ;  
push (pop () + pop ()) ;  
push (4) ;  
push (6) ;  
push (pop () * pop ()) ;  
push (pop () * pop ()) ;  
push (7) ;  
push (pop () + pop ()) ;  
push (pop () * pop ()) ;
```

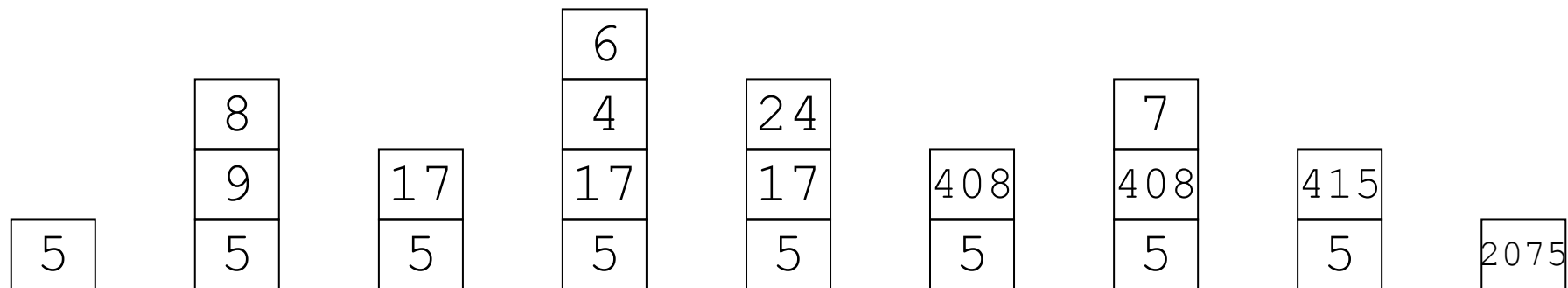
計算の中間結果を貯える

# 逆ポーランド記法 演算子を後置する記法

生物情報 情報システム概論 森下

$5 * ((9 + 8) * (4 * 6)) + 7$  中置記法

$5\ 9\ 8 +\ 4\ 6 * * 7 + *$  逆ポーランド記法

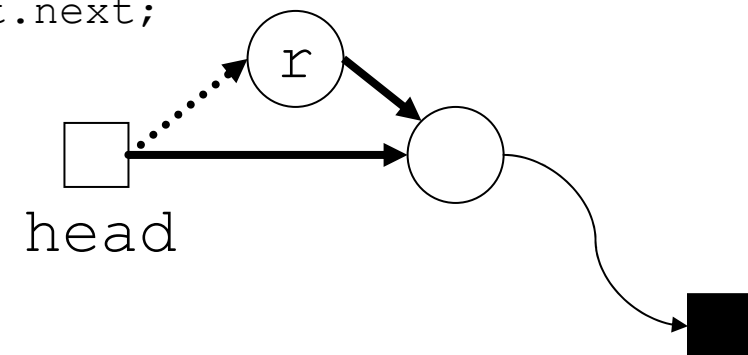
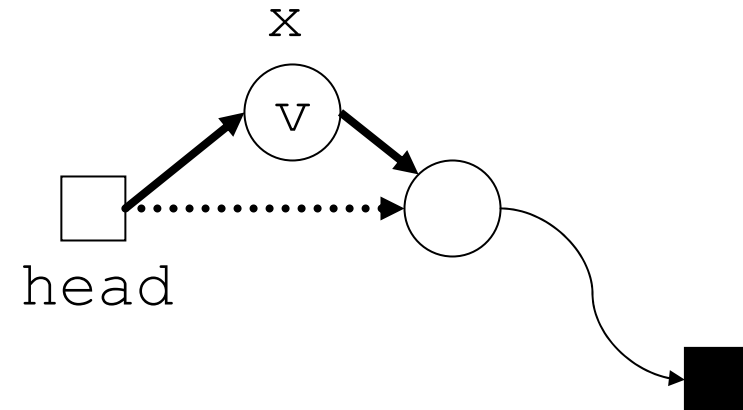
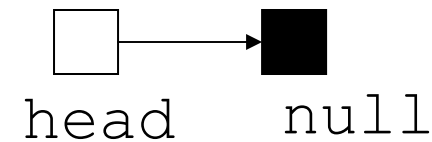


```
push (5) ;  
push (9) ;  
push (8) ;  
push (pop () + pop ()) ;  
push (4) ;  
push (6) ;  
push (pop () * pop ()) ;  
push (pop () * pop ()) ;  
push (7) ;  
push (pop () + pop ()) ;  
push (pop () * pop ()) ;
```

逆ポーランド記法はスタックでの計算を意識すれば書き下しやすい

# リンクをつかったスタックの実装

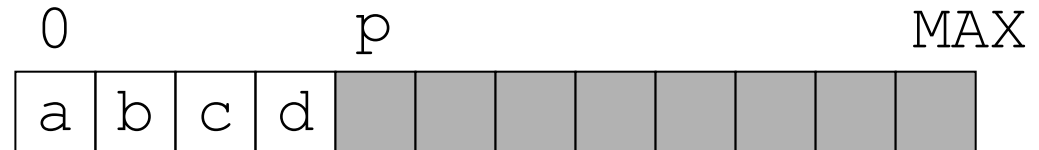
```
public class Stack {
    private Node head;
    public Stack(){
        head = new Node(-1);
        head.next = null;
    }
    public void push(int v){
        Node x = new Node(v);
        x.next = head.next;
        head.next = x;
    }
    public int pop(){
        int r = -1;
        if( !stackEmpty() ){
            r = head.next.key;
            head.next = head.next.next;
        }
        return r;
    }
    public boolean stackEmpty(){
        return (head.next == null);
    }
    :
}
```



# 配列を使ったスタックの実装

生物情報 情報システム概論 森下

```
public class Stack {
    private int MAX;
    private int[] stack;
    private int p;
    public Stack(int size) {
        MAX = size;
        stack = new int[MAX + 1];
        p = 0;
    }
    public void push(int v) {
        if (p < MAX) {
            stack[p++] = v; // stack[p]=v; p++;
        }
    }
    public int pop() {
        if ( !stackEmpty() ) {
            return stack[--p]; // p=p-1; return stack[p];
        }else {
            return -1;
        }
    }
    public boolean stackEmpty(){
        return (p == 0);
    }
    :
}
```



**スタック** (LIFO)

Last-In-First-Out

**最後に入ったものを最初に処理**  
**キュー**

(FIFO) First-In-First-Out

**最初に入ったものを最初に処理**

# キュー

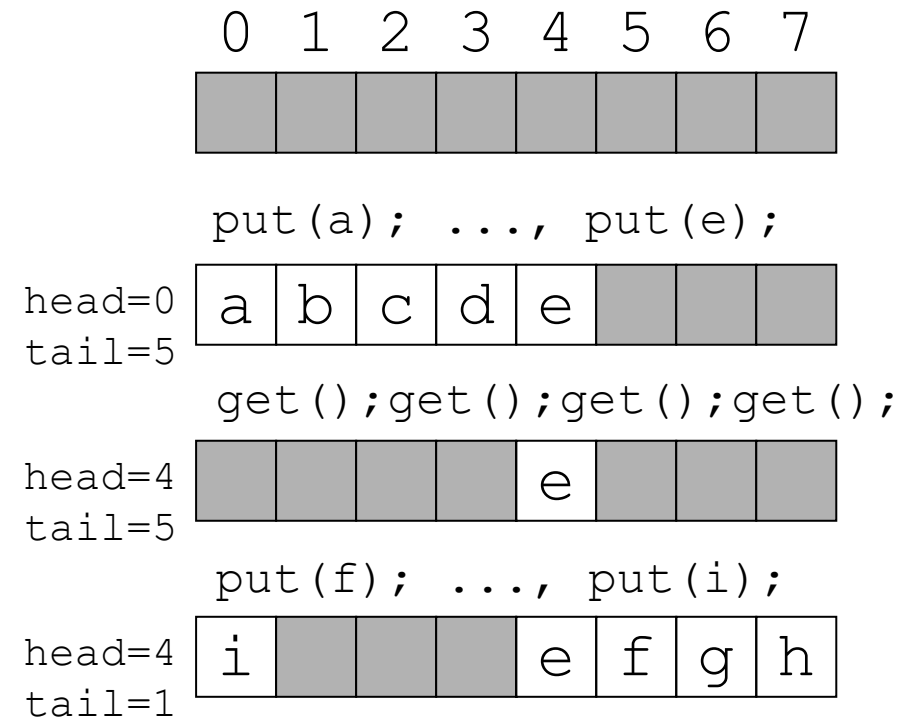
```
public class Queue {
    private int MAX;
    private char[] queue;
    private int head, tail;

    public Queue(int size){
        MAX = size;
        queue = new char[MAX+1];
        head = 0;
        tail = 0;
    }

    public void put(char v){
        queue[tail++] = v;
        if(tail > MAX){
            tail = 0;
        }
    }

    public char get(){
        char t = queue[head++];
        if(head > MAX){
            head = 0;
        }
        return t;
    }
}
```

MAX = 7



バグはどこ？



# キュー

head=4  
tail=1

|   |  |  |  |   |   |   |   |
|---|--|--|--|---|---|---|---|
| i |  |  |  | e | f | g | h |
|---|--|--|--|---|---|---|---|

## キューがあふれる場合

`put(j); put(k); put(l); put(m);`

head=4  
tail=5

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| i | j | k | l | m | f | g | h |
|---|---|---|---|---|---|---|---|

## 空の場合

`get(); get(); get();`  
`get(); get(); get();`

head=2  
tail=1

|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|

`put(j); get();`

head=2  
tail=2

|  |   |  |  |  |  |  |  |
|--|---|--|--|--|--|--|--|
|  | j |  |  |  |  |  |  |
|--|---|--|--|--|--|--|--|

入れたはずのjを取り出せない...

# キュー

max を超えたとき 0 に戻さない方針

head=4  
tail=9

|   |  |  |  |   |   |   |   |
|---|--|--|--|---|---|---|---|
| i |  |  |  | e | f | g | h |
|---|--|--|--|---|---|---|---|

キューがあふれる場合

put(j);put(k);put(l);put(m);

head=4  
tail=13

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| i | j | k | l | m | f | g | h |
|---|---|---|---|---|---|---|---|

空の場合

get();get();get();  
get();get();get();

head=10  
tail=9

|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|

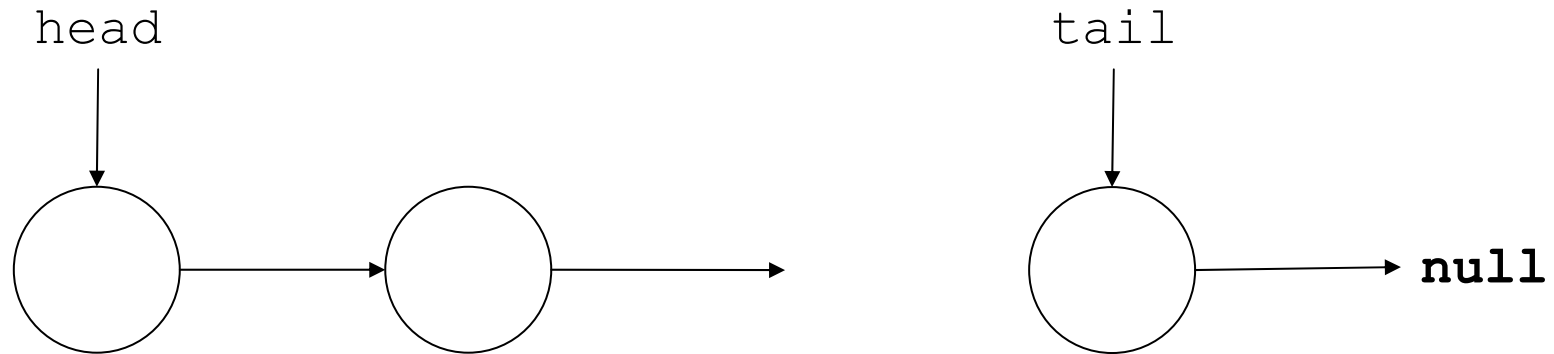
修正案 配列のインデックスを mod で計算

```
public void put(char v){
    if( tail < head + MAX ){
        // MAX 個より多く入れない
        queue[tail%(MAX+1)] = v;
        tail++;
    }
}
```

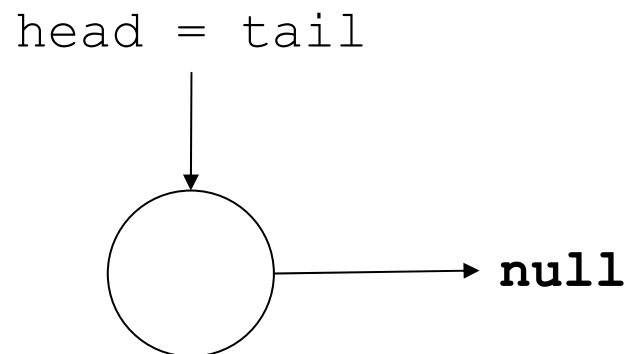
```
public char get(){
    char t = -1;
    if( head < tail ){
        t = queue[head%(MAX+1)];
        head++;
    }
    return t;
}
```

MAX = 7

## リンクを使ってキューを実装するには？



空の場合 `head = tail = null`



## リンクを使ってキューを実装するには？

```
public class Queue {
    private Node head, tail;          // Node は以前の定義を借用
    public Queue(){ head = null; tail = null; }
    public void put(char c){
        Node x = new Node(c);
        if(queueEmpty()){ tail = x; head = x;}
        // キューが空のとき head, tail を新ノードに
        else{ tail.next = x; tail = x; } // 一番最後に x を付加
    }
    public char get(){
        char r = ' ';                  // キューが空の時には空白をかえすように
        if(!queueEmpty()){
            r = (char) head.key; // 空でないなら先頭の値を r へ
            if(head == tail){ head = null; tail = null; }
            // 1つしか元がない場合
            else{ head = head.next; } // 元が複数存在する場合
        }
        return r;
    }
    public boolean queueEmpty(){
        return (head == null);
    }
}
```

# 抽象データ型 (abstract data type)

スタック、キューはリストや配列で実装できる

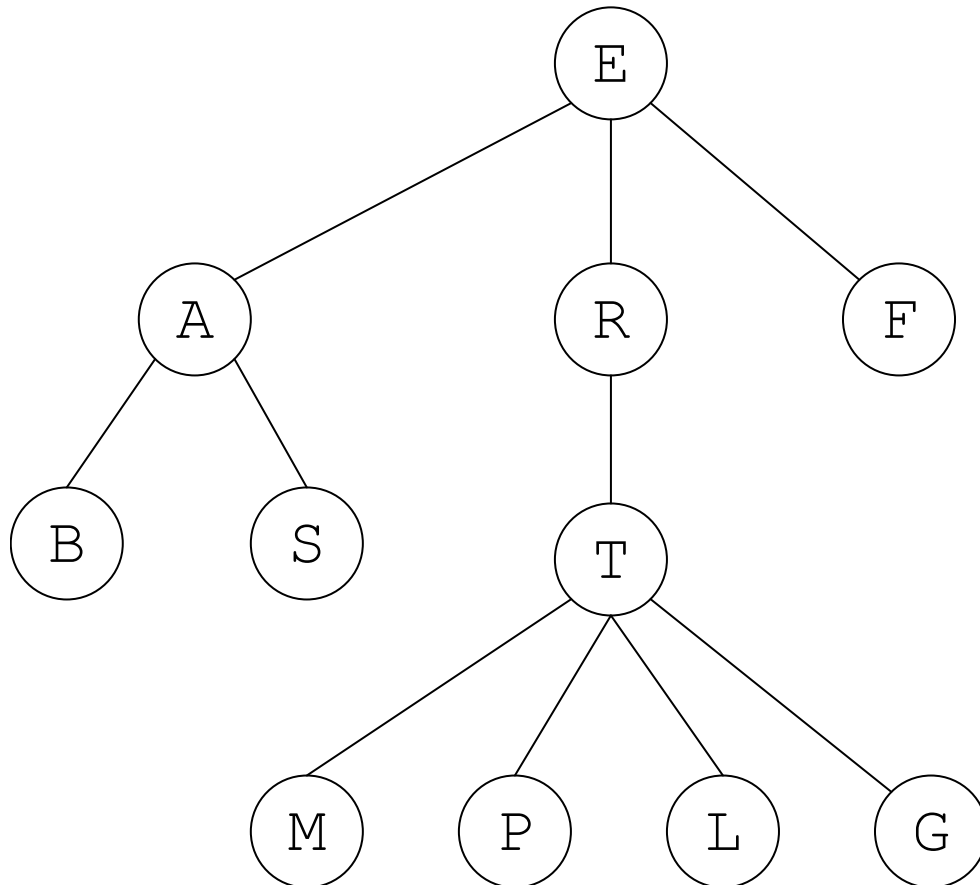
外部からみると、キューにおいては、  
実装方式の詳細を知らなくても  
`put`, `get` を使うことができれば便利

`put`, `get` のような操作だけを公開して、  
内部の実装を参照させない

実装を変更しても外部への影響がない  
大規模なソフトウェア開発では、むしろ好都合

# 4章 木構造

# 木



節点(ノード)／頂点

根ノード E

道(パス)

$E \rightarrow R \rightarrow T \rightarrow P$

親

M に対する T

子

T に対する M, P, L, G

葉／終端点／外部節点

B, S, M, P, L, G, F

内部節点(葉でない)

E, A, R, T

部分木

T の部分木

T, M, P, L, G

レベル

根からの節点数

R=1, T=2, M=3

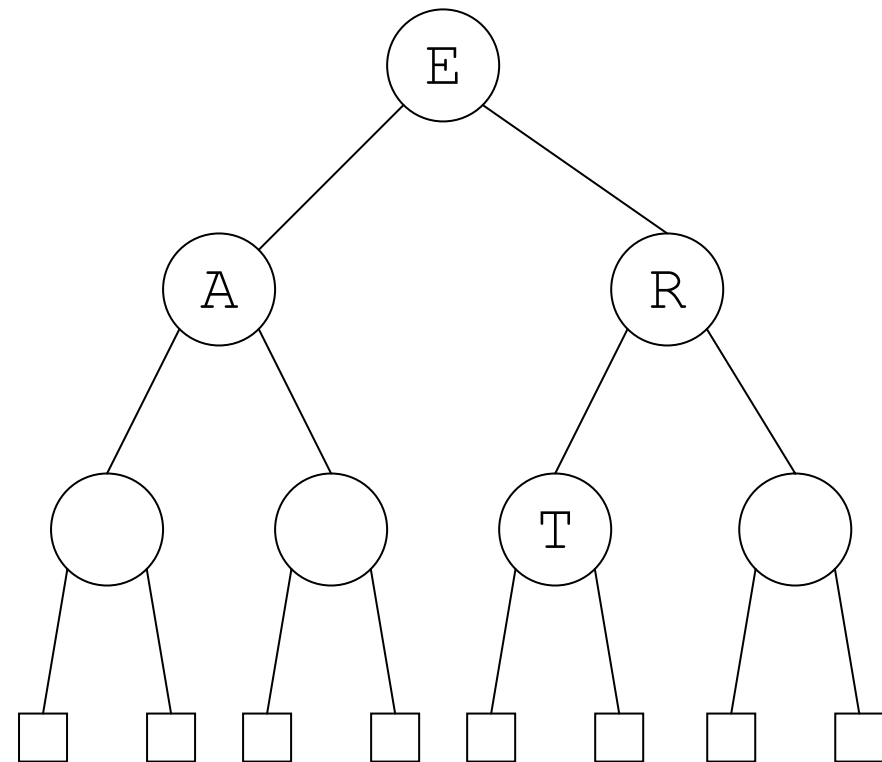
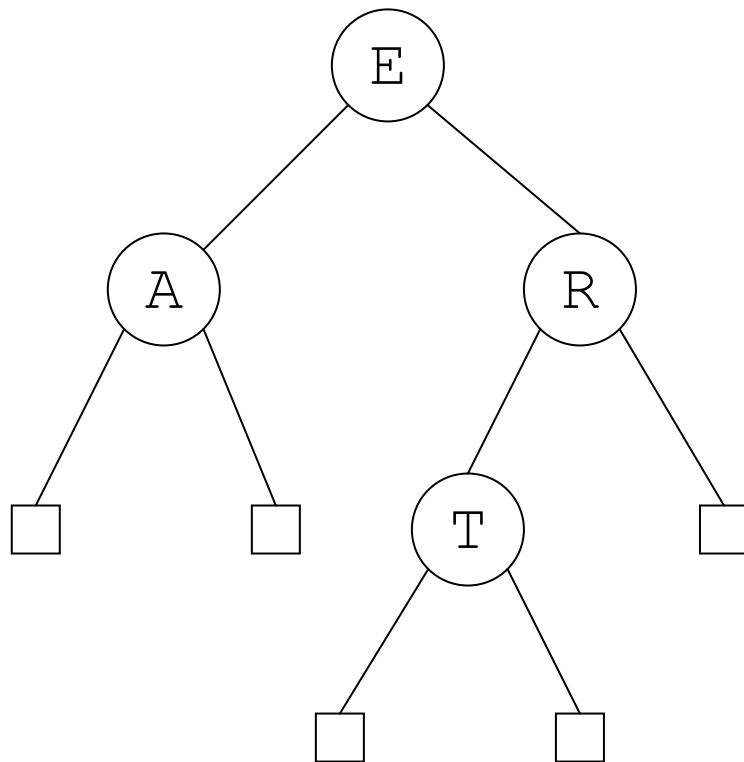
高さ

最大のレベル

# 2分木

2分木  
完全2分木

内部節点は2つの子をもつ  
一番下のレベル以外のレベルは内部節点で詰まっている





# 木の性質

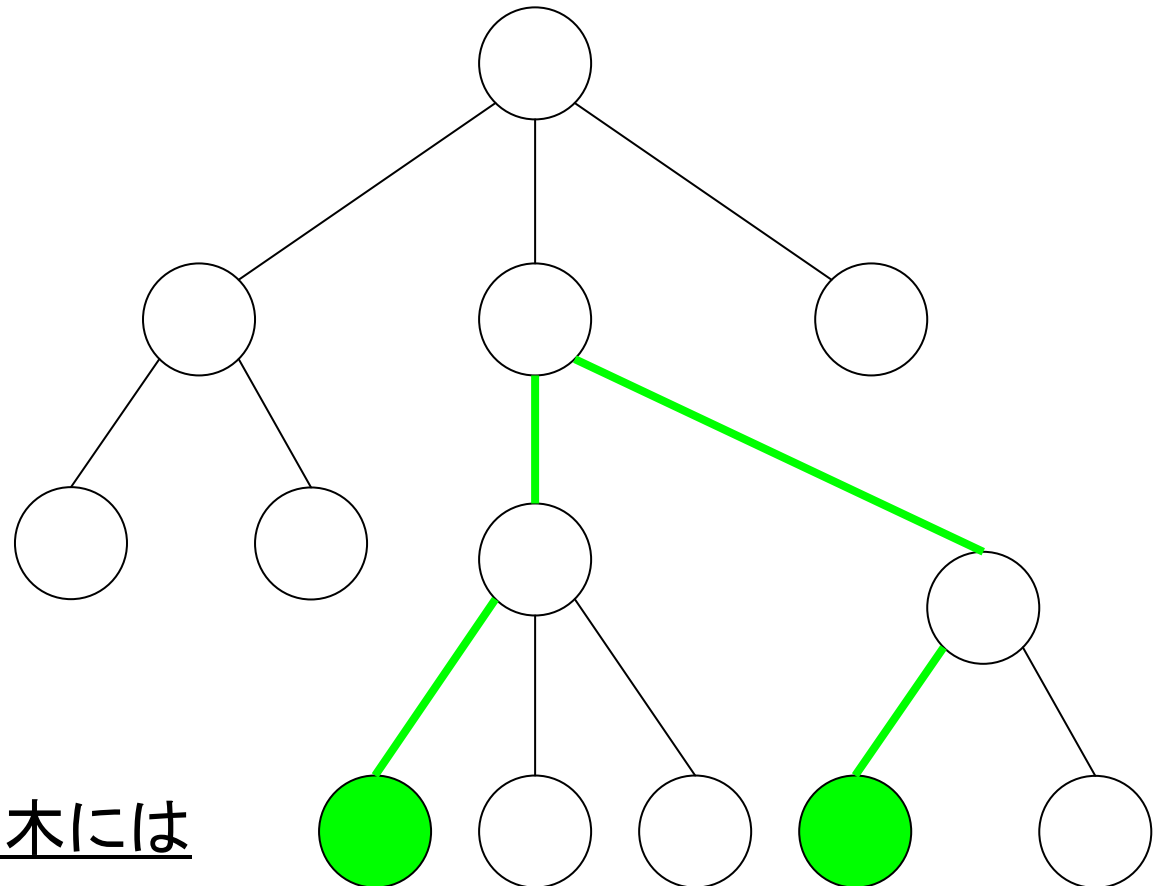
任意の2つの節点を結ぶ道はただ1つである

最小共通祖先:  
2つの頂点と根を結ぶ  
2つの道に共通して  
現れる節点で  
最も葉に近い節点

最小共通節点を  
経由して唯一の道が  
みつかる

節点の個数が  $N$  である木には  
辺が  $N-1$  個

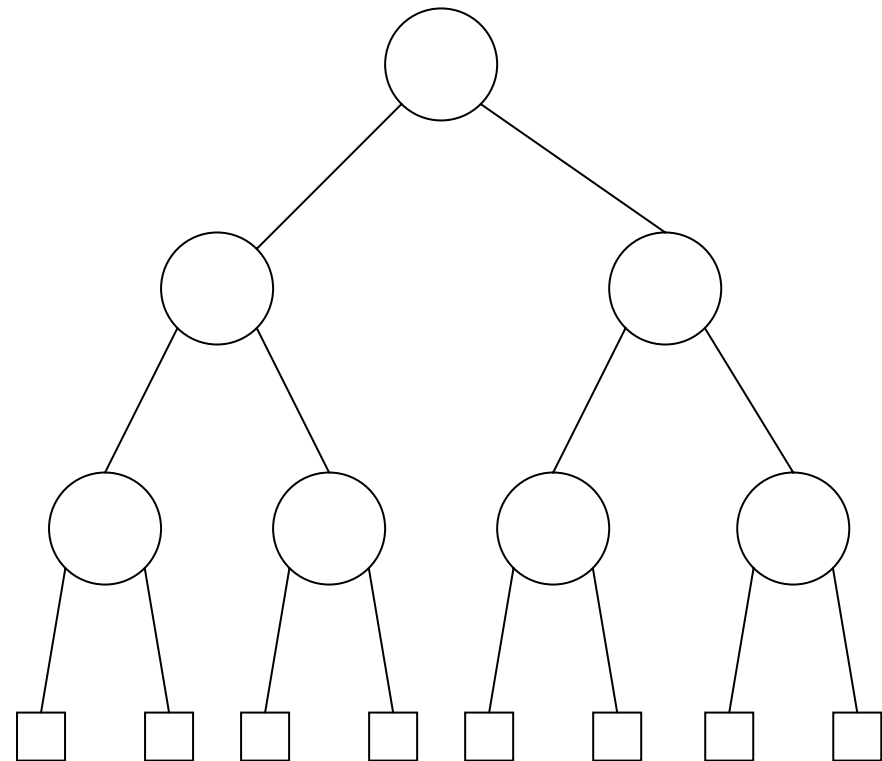
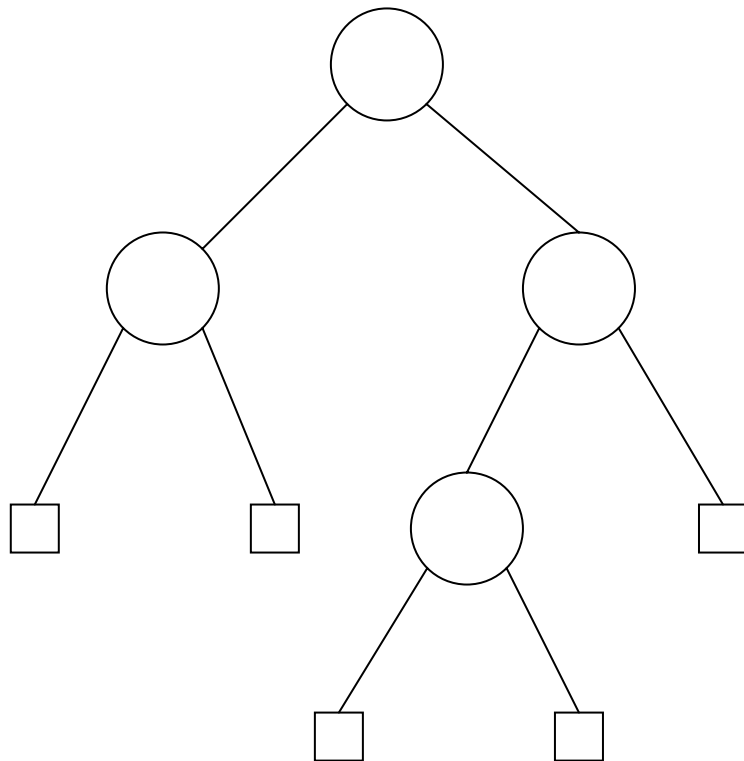
各節点には親への辺がただ1つ      根には親がない



# 木の性質

内部節点数が  $N$  の2分木には、 $N+1$  個の外部節点

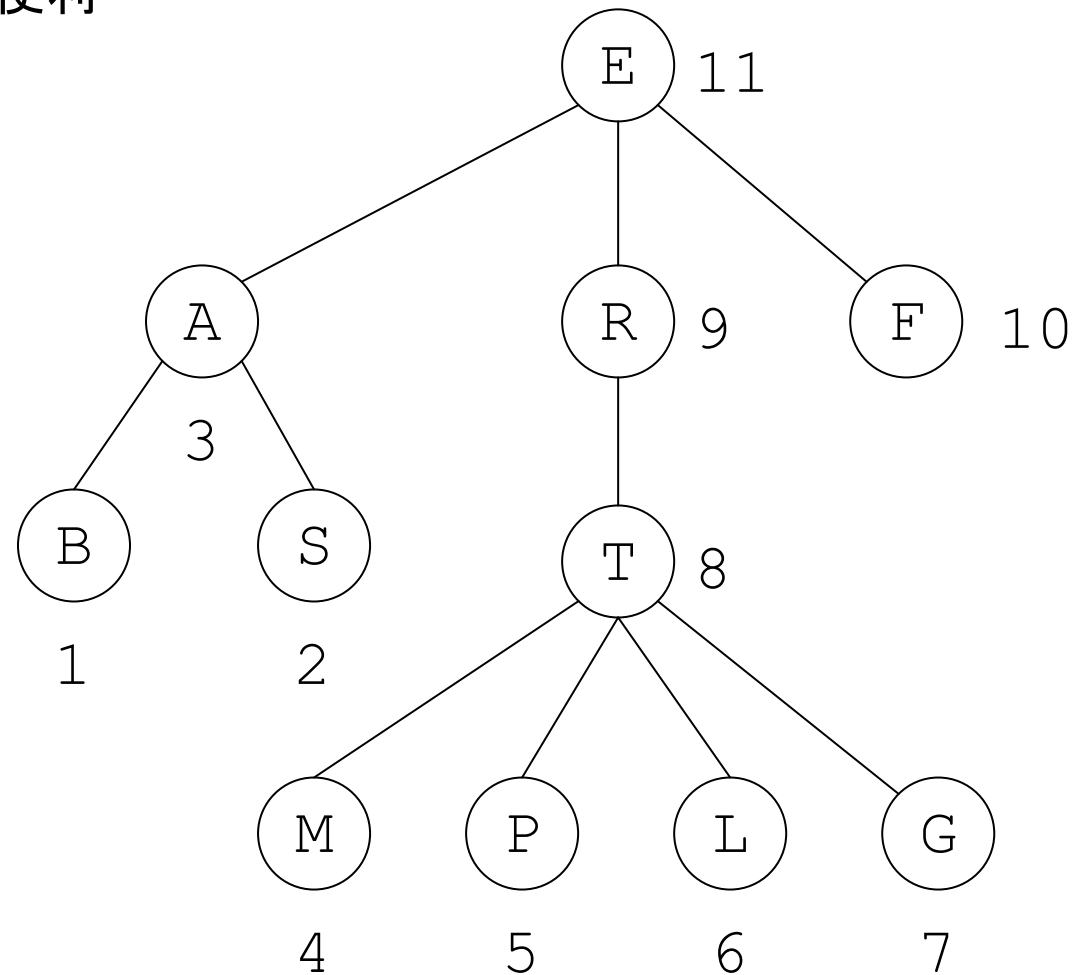
内部節点数が  $N$  の完全2分木の高さは、約  $\log_2 N$



# 木の表現

親だけ覚えておく方式  
下から上への移動に便利

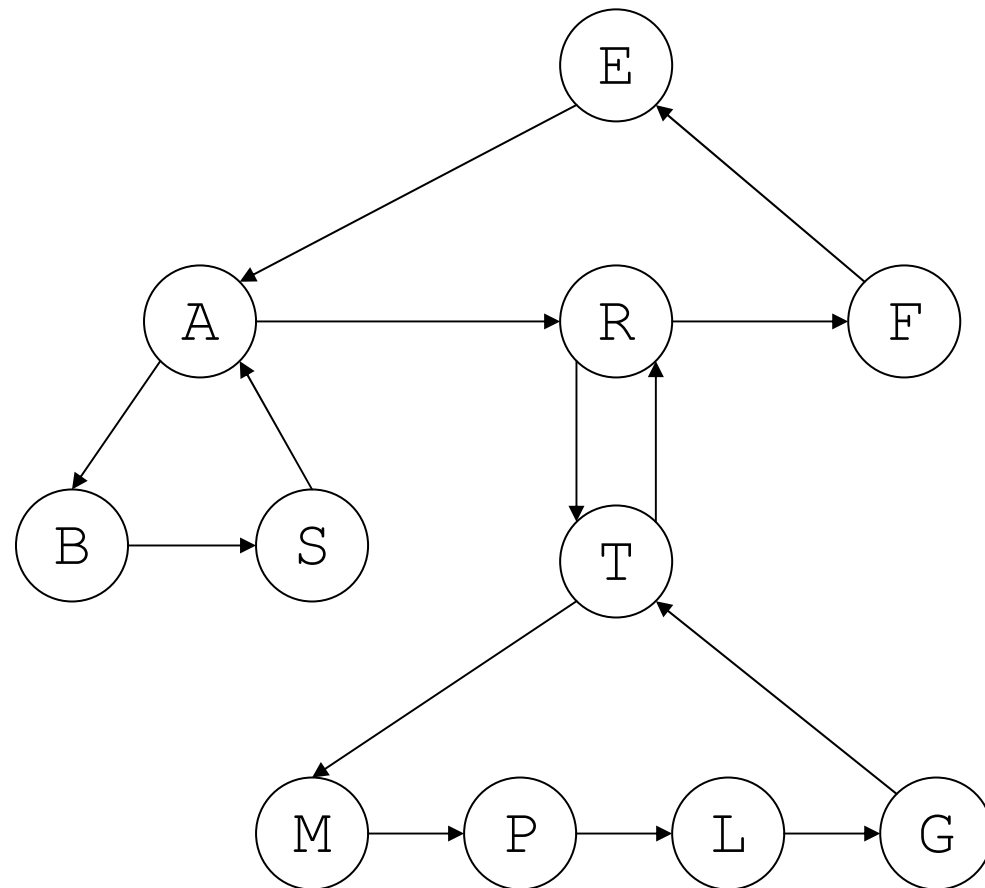
| k  | a[k] | dad[k] |
|----|------|--------|
| 1  | B    | 3      |
| 2  | S    | 3      |
| 3  | A    | 11     |
| 4  | M    | 8      |
| 5  | P    | 8      |
| 6  | L    | 8      |
| 7  | G    | 8      |
| 8  | T    | 9      |
| 9  | R    | 11     |
| 10 | E    | 11     |
| 11 | F    | 11     |



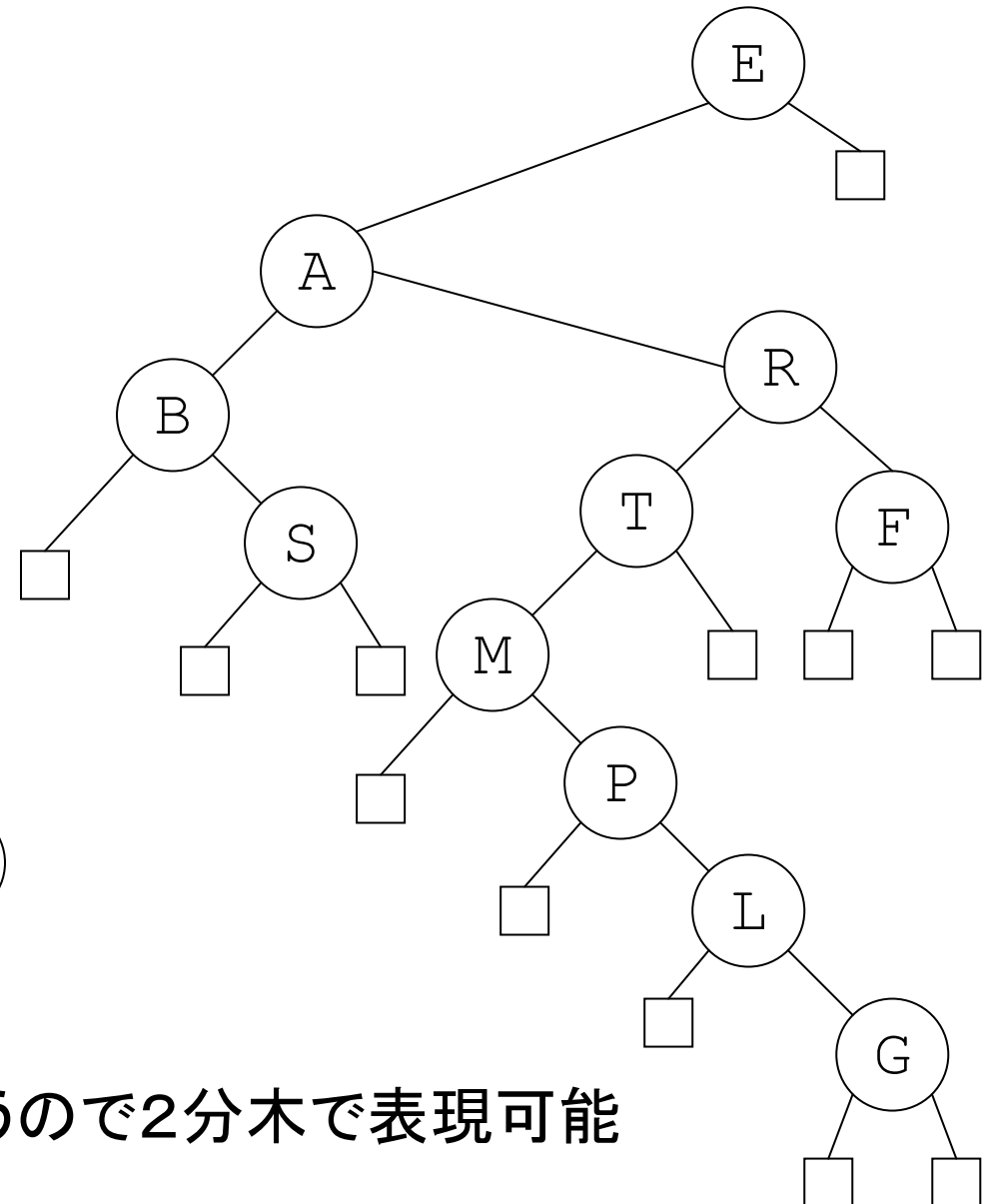
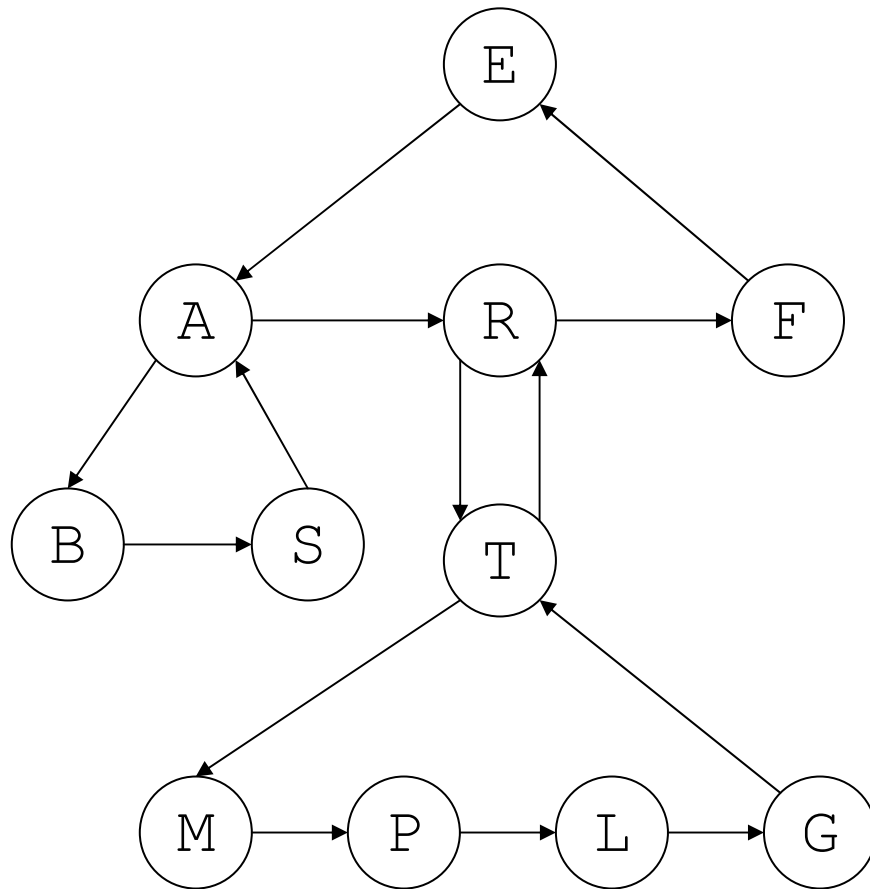
# 木の表現

各ノードで  
子へのリンクと  
兄弟へのリンク  
(末っ子は親へのリンク)  
を使う方式

上から下への移動も容易

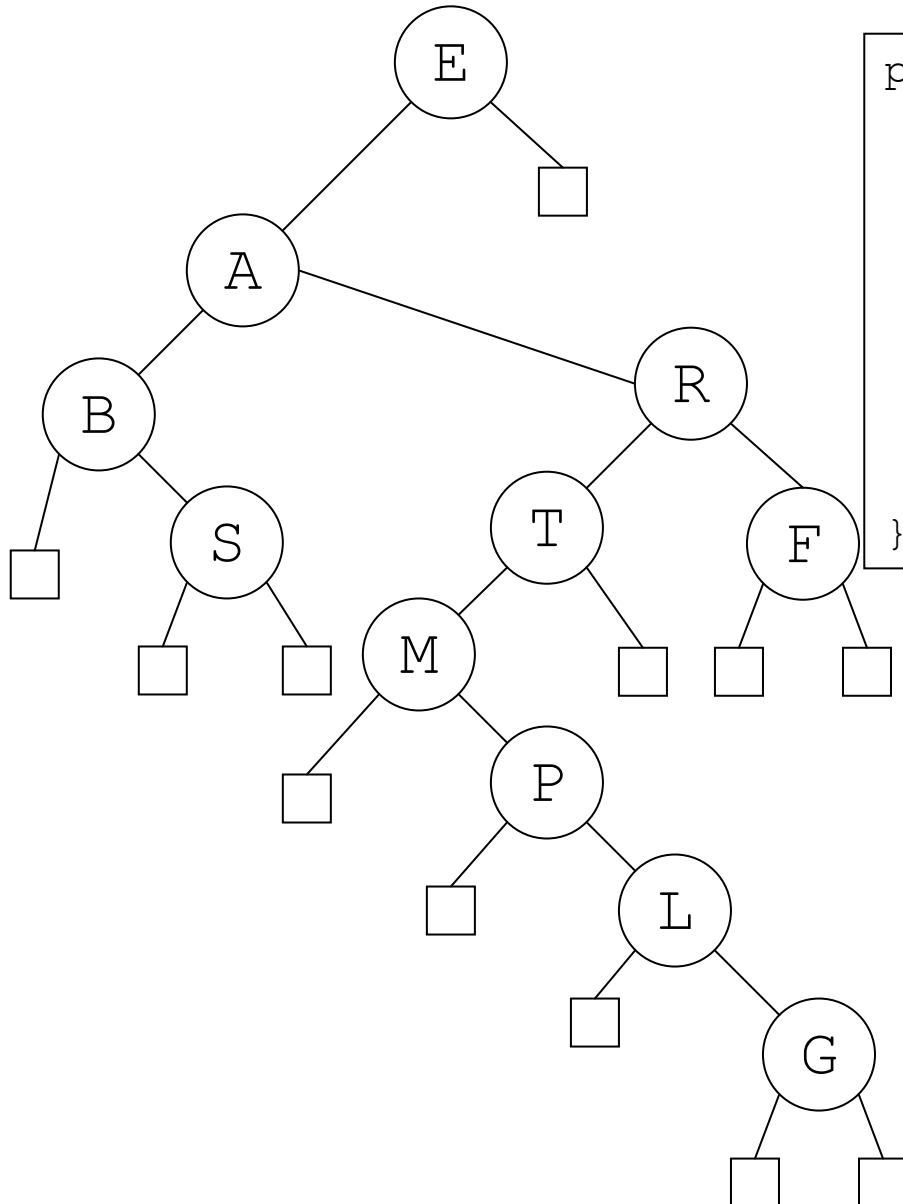


## 2分木による一般の木の表現



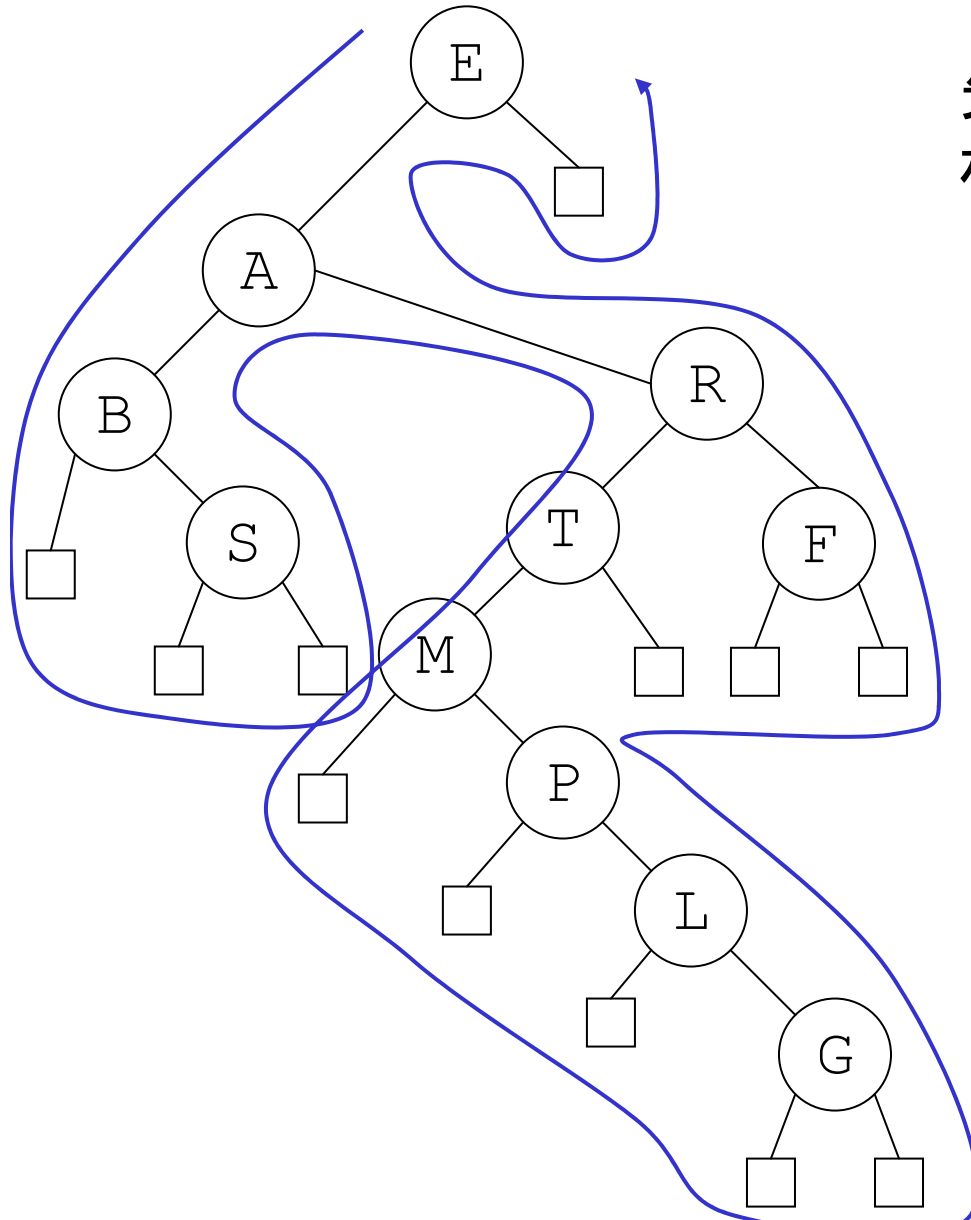
2つのリンクを使うので2分木で表現可能

# 2分木のデータ構造



```
public class Node {  
    public char info;  
    public Node left, right;  
  
    public Node(char c, Node x, Node y) {  
        info = c;  
        left = x;  
        right = y;  
    }  
}
```

# 木の走査



先行順  
根→左部分木→右部分木

```
public class Traverse {  
  
    private static void traverse(Node r) {  
        if (r != null) {  
            System.out.print(r.info + " ");  
            traverse(r.left);  
            traverse(r.right);  
        }  
    }  
}
```

# 木の再帰的走査 スタックによる実装

生物情報 情報システム概論 森下

```
public class Stack {
    final int MAX = 100;
    private Node[] stack;
    private int p;
    public Stack() {
        stack = new Node[MAX + 1];
        p = 0;
    }
    public void push(Node v) {
        if (p < MAX) {
            stack[p++] = v;
        }
    }
    public Node pop() {
        if ( !stackEmpty() ) {
            return stack[--p];
        }else {
            return null;
        }
    }
    public boolean stackEmpty(){
        return (p == 0);
    }
}
```

先行順 根→左部分木→右部分木

```
public void traverse(Node r){
    push(r);

    while( !stackEmpty() ){
        Node t = pop();
        System.out.print(t.info);

        if(t.right != null){
            push(t.right);
        }

        if(t.left != null){
            push(t.left);
        }
    }
}
```



# 5章 再帰呼び出し

# 漸化式と再帰的呼出し

## フィボナッチ数列

$\text{fib}(0) = \text{fib}(1) = 1$

$\text{fib}(N) = \text{fib}(N-2) + \text{fib}(N-1) \quad N \geq 2$

1 1 2 3 5 8 13 21 34 55 89 144 ...

```
private static int fib(int n){
    if(n <= 1){
        return 1;           // 負の数も 1
    }
    count++;
    return fib(n-2) + fib(n-1);
}
```

## 無駄な計算の重複

$\text{fib}(4)$

$\text{fib}(2) + \text{fib}(3)$

$(\text{fib}(0) + \text{fib}(1)) + (\text{fib}(1) + \text{fib}(2))$

$(\text{fib}(0) + \text{fib}(1)) + (\text{fib}(1) + (\text{fib}(0) + \text{fib}(1)))$

$\text{fib}(N)$  を計算するのに

$\text{fib}(N)-1$  回  $\text{fib}$  を呼び出す無駄

## for ループでの書き換え

```
private static int iterFib(int n) {  
    if(n <= 1) {  
        return 1;           // 負の数も 1  
    }  
    int p2 = 1;              // 2つ前  
    int p1 = 1;              // 1つ前  
    int tmp;  
    for(int i = 2; i <= n; i++) {  
        tmp = p1;  
        p1 = p2 + p1;  
        p2 = tmp;  
        count++;  
    }  
    return p1;  
}
```

1 1 2 3 5 8 13 21 34 55 89 144 ...

$$\begin{aligned} \text{pn}(2) &= 2 && 2, && 1+1 \\ \text{pn}(3) &= 3 && 3, && 2+1, && 1+1+1 \\ \text{pn}(4) &= 5 && 4, && 2+2, && 3+1, && 2+1+1, && 1+1+1+1 \\ \text{pn}(5) &= 7 && 5, && 3+2, && 4+1, && 2+2+1, && 3+1+1, && 2+1+1+1, && 1+1+1+1+1 \end{aligned}$$

## 再帰的呼び出しの例 分割数

$p_n(n, i)$  :  $n$  を  $i$  個の自然数の和で表現する場合の数

$$p_n(n, 1) = 1 \quad n$$

$$p_n(n, n) = 1 \quad 1 + 1 + \dots + 1$$

$1 < i < n$  のとき

□ 分割された自然数のうち少なくとも一つは1である場合

( $n-1$  を  $i-1$  個に分割した和) + 1 の形なので分割数は  $p_n(n-1, i-1)$

□ 分割された自然数がすべて 2 以上の場合

$n = x_1 + x_2 + \dots + x_i$  ならば、

$n-i = (x_1 - 1) + (x_2 - 1) + \dots + (x_i - 1)$  へ変形可能

分割数は  $p_n(n-i, i)$

$n-i < i$  となる可能性があるので  $n < i$  ならば  $p_n(n, i) = 0$  と約束

$$p_n(n, 1) = 1, p_n(n, n) = 1$$

$$p_n(n, i) = p_n(n-1, i-1) + p_n(n-i, i) \quad \text{ただし } 1 < i < n \text{ のとき}$$

$$p_n(n, i) = 0 \quad \text{ただし } n < i \text{ のとき}$$

$$p_n(n) = p_n(n, 1) + p_n(n, 2) + \dots + p_n(n, n)$$

## 再帰的呼び出しの例 分割数

```
public static int pn(int n){
    int sum = 0;
    if(n > 0){
        for(int i=1; i<=n; i++){ sum += pn(n,i); }
    }
    return sum;
}

public static int pn(int n, int i){
    if(n < i){ return 0; }
    if(n == i || i == 1){ return 1; }
    return pn(n-i,i)+pn(n-1,i-1);
}
```

# 分割統治 (divide-and-conquer)の再帰的呼出しによる記述

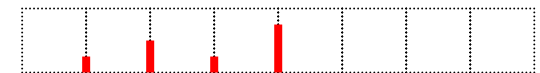
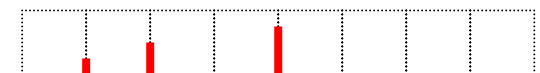
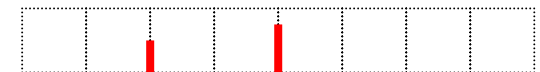
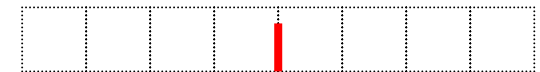
分割統治: 小さな問題に分割(通常は2分割)して解き、あとでまとめる

```
static void rule
(int l, int r, int h){

    int m = (l+r)/2;

    if(h > 0){
        mark(m, h);
        rule(l, m, h-1);
        rule(m, r, h-1);
    }
}
```

```
rule(0, 8, 3)
mark(4, 3)
rule(0, 4, 2)
mark(2, 2)
rule(0, 2, 1)
mark(1, 1)
rule(0, 1, 0)
rule(1, 2, 0)
rule(2, 4, 1)
mark(3, 1)
rule(2, 3, 0)
rule(3, 4, 0)
rule(4, 8, 2)
```



# 分割統治

rule(0, 8, 3)

:  
:  
:  
:  
:  
:  
:

rule(4, 8, 2)

mark(6, 2)

rule(4, 6, 1)

mark(5, 1)

rule(4, 5, 0)

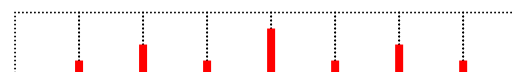
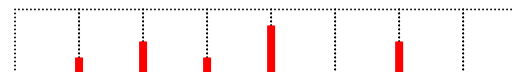
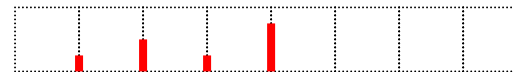
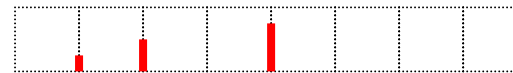
rule(5, 6, 0)

rule(6, 8, 1)

mark(7, 1)

rule(6, 7, 0)

rule(7, 8, 0)





# 分割統治

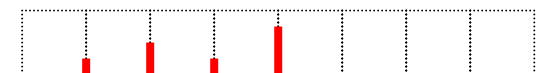
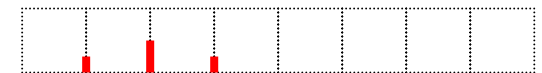
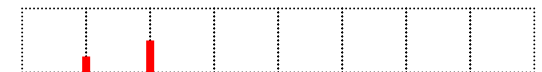
目盛を書く順番を変更しても結果に変化なし

```
static void rule
(int l, int r, int h){

    int m = (l+r)/2;

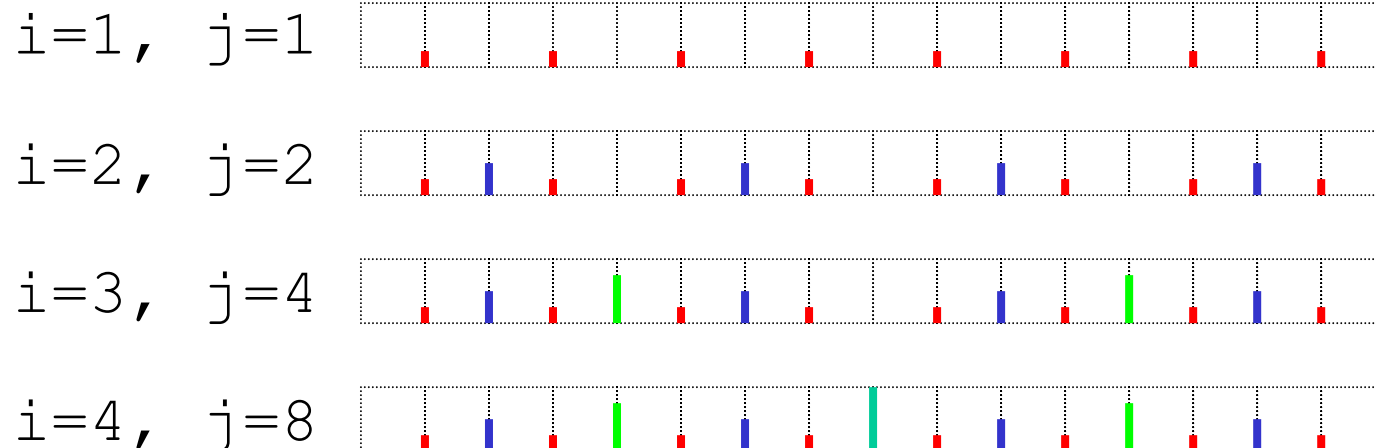
    if(h > 0){
        rule(l, m, h-1);
        mark(m, h);
        rule(m, r, h-1);
    }
}
```

```
rule(0, 8, 3)
    rule(0, 4, 2)
        rule(0, 2, 1)
            rule(0, 1, 0)
                mark(1, 1)
            rule(1, 2, 0)
                mark(2, 2)
            rule(2, 4, 1)
                rule(2, 3, 0)
                    mark(3, 1)
                rule(3, 4, 0)
                    mark(4, 3)
                rule(4, 8, 2)
```



# for ループでの書き換え

```
int l = 0;
int r = 16;
int h = 4;
int i, j;
for(i=1, j=1; i <= h; i++, j+=j){
    // i=1,2, ..., h   j=1,2,4,8, ...
    for(int t = 0; t < (l+r)/(2*j); t++){
        mark(l+j+2*j*t, i);
    }
}
```



# 6章 アルゴリズムの解析

# 計算時間の評価

平均の場合

典型的なデータに対して期待される計算時間

最悪の場合

最もたちの悪い種類の入力に対して必要な計算時間

# アルゴリズムの分類

生物情報 情報システム概論 森下

$N$  をデータ数    関数  $f(N)$  に比例する計算時間

$f(N)$      $\lg N, \lg^2 N, N^{1/2}, N, N \lg N, N \lg^2 N, N^2, N^3, 2^N$

ただし  $\log_2 N = \lg N, \log_e N = \ln N$

「比例する」 計算時間    定数倍は気にしない  
 $\log$  の底は 2 でも自然対数の底  $e$  でもよい

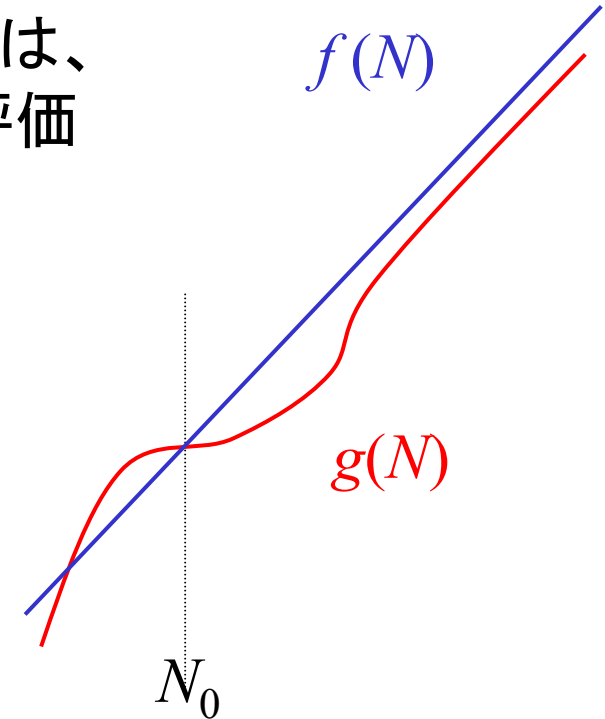
# 最悪計算量

アルゴリズム性能を理論的に解析するさいには、  
定数係数を無視して、最悪の場合の性能を評価

厳密に数学的結果を証明できる良さ

「比例する」という言葉の定義

O-notation (計算時間の漸近的上界を示す)



ある定数  $c_0$  と  $N_0$  が存在して、 $N > N_0$  である任意の  $N$  に対して  $g(N) < c_0 f(N)$  ならば、関数  $g(N)$  は  $O(f(N))$  (オーダー  $f(N)$  と読む) であるという。

# 最悪計算量

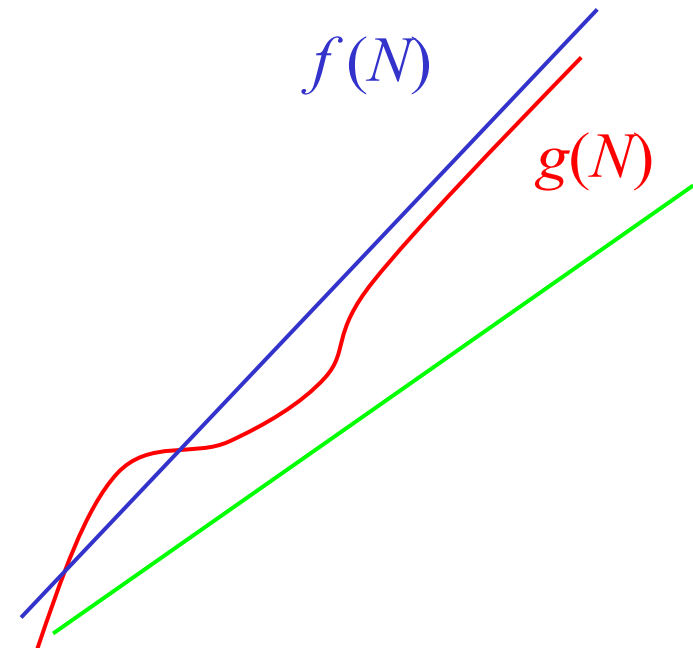
$O(f(N))$  は上界を示したに過ぎない

$g(N)$  が  $f(N)$  より小さいとは

$$\lim_{N \rightarrow \infty} g(N) / f(N) = 0$$

$f(N)$  より小さい  $h(N)$  が存在して  
 $g(N)$  は  $O(h(N))$  であると、  
 $f(N)$  はギリギリの上界でない

しかし、このような  $h(N)$  が存在しない  
ことを証明するのは通常難しい



良質な上界を  
得るのは難しい

# 計算量の下界

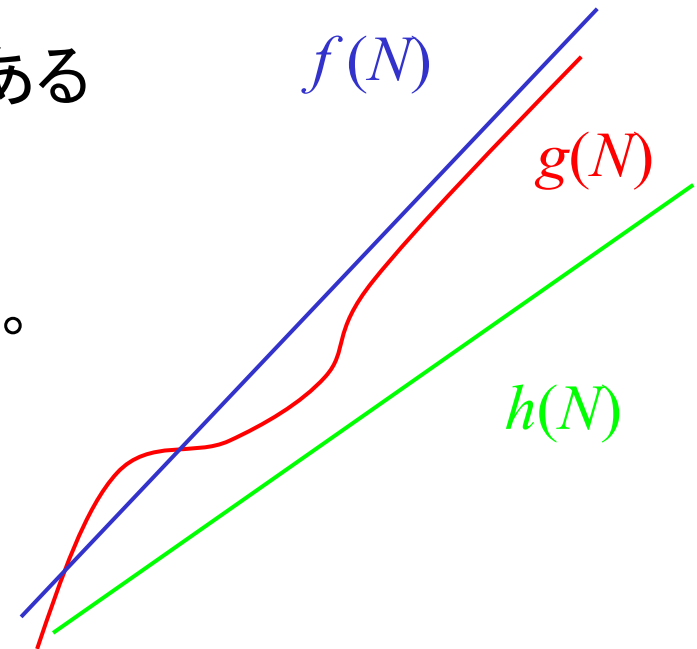
$\Omega$ -notation (計算時間の漸近的下界を示す)

ある定数  $c_0$  と  $N_0$  が存在して、 $N > N_0$  である  
任意の  $N$  に対して

$$0 \leq c_0 h(N) \leq g(N)$$

ならば、関数  $g(N)$  は  $\Omega(h(N))$  であるという。

良質な下界を得るのも難しい





# 漸近的な差と実用性

| $N$       | $(1/4)Mg^2N$ | $(1/2)Mg^2N$ | $Mg^2N$     | $N^{3/2}$     |
|-----------|--------------|--------------|-------------|---------------|
| 10        | 22           | 45           | 90          | 30            |
| 100       | 900          | 1,800        | 3,600       | 1,000         |
| 1,000     | 20,250       | 40,500       | 81,000      | 31,000        |
| 10,000    | 422,500      | 845,000      | 1,690,000   | 1,000,000     |
| 100,000   | 6,400,000    | 12,800,000   | 25,600,000  | 31,600,000    |
| 1,000,000 | 90,250,000   | 180,500,000  | 361,000,000 | 1,000,000,000 |

# 平均の場合の解析

命令時間の正確な見積りが難しい

コンパイラ、ハードウェアによる命令の並列実行など

解析が複雑になりがち 定数の正確な見積り

漸近的計算量は定数を見捨てることで問題を簡単にしている

入力モデルが現実を反映していない場合が多い

ランダムに生成されたデータはプログラムに容易な例となりがち  
現実にはデータは偏っている

## O 記法(O-notation) による近似

$$a_0 N \lg N + a_1 N + a_2 \qquad a_0 N \lg N + O(N)$$

$$f(N) + O(g(N))$$

$f(N)$  が漸近的に  $g(N)$  より大きくなるとき、  
 $f(N) + O(g(N))$  は約  $f(N)$  という

# 基本漸化式

$$C(N) = C(N-1) + N \quad N \geq 2$$

$$C(1) = 1$$

$$C(N) = \sum_{i=1}^N i = \frac{N(N+1)}{2}$$

$$C(N) = C(N/2) + 1 \quad N \geq 2$$

$$C(1) = 0$$

$N = 2^n$  の場合

$$C(2^n) = C(2^{n-1}) + 1 = C(2^0) + n = n$$

$2^n \leq N < 2^{n+1}$  の場合

$$n \leq C(N) < n+1$$

$$C(N) = C(N/2) + N \quad N \geq 2$$

$$C(1) = 0$$

$$C(N) = N + N/2 + N/4 + \dots \approx 2N$$

$$C(N) = 2C(N/2) + N \quad N \geq 2$$

$$C(1) = 0$$

$N = 2^n$  のとき  $N$  で両辺を割る

$$\frac{C(2^n)}{2^n} = \frac{C(2^{n-1})}{2^{n-1}} + 1$$

$$= \frac{C(2^{n-2})}{2^{n-2}} + 1 + 1 = \dots$$

$$= n = \lg N$$

$$C(N) = N \lg N$$

# 8章 初等的な整列法

# Selection Sort

## 最小の元を選択

BIOINFORM**A**TICS  
AIOINFORM**B**TICS  
ABOINFORMIT**I**C  
ABCIN**F**ORMITIOS  
ABCFN**I**ORMITIOS  
ABCFINORM**I**TIOS  
ABCFIIORMNT**I**OS  
ABCFIIIR**M**NTOOS  
ABCFIIIMR**N**TOOS  
ABCFIIIMN**R**T**O**OS  
ABCFIIIMNO**T**R**O**S  
ABCFIIIMNOO**R**TS  
ABCFIIIMNOORT**S**  
ABCFIIIMNOOR**S**T

## 約 $N^2/2$ 回の比較と最悪約 $N$ 回の交換

```
public void selection() {
    for (int i = 0; i < N - 1; i++) {
        int min = i;
        for (int j = i + 1; j < N; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }
        // a[i], a[min] を交換
        int tmp = a[min];
        a[min] = a[i];
        a[i] = tmp;
    }
}
```

# Bubble Sort

隣り合う元の大小関係を修正 最小の元は左側に

BIOINFORMATIC  
ABIOINFORMCTIS  
ABCIOINFORMITS  
ABCFIOINIORMST  
ABCFIIOINMORST  
ABCFIIIO MNORST  
ABCFIIIMONORST  
ABCFIIIMNOORST  
ABCFIIIMNOORST  
ABCFIIIMNOORST  
ABCFIIIMNOORST  
ABCFIIIMNOORST  
ABCFIIIMNOORST  
ABCFIIIMNOORST  
ABCFIIIMNOORST

```
public void bubble() {  
    for (int i = 0; i < N; i++) {  
        for (int j = N - 1; i < j; j--) {  
            if (a[j - 1] > a[j]) {  
                // a[j-1] と a[j] を交換  
                int tmp = a[j - 1];  
                a[j - 1] = a[j];  
                a[j] = tmp;  
            }  
        }  
    }  
}
```

約 $N^2/2$ 回の比較と  
最悪約 $N^2/2$ 回の交換

# Insertion Sort

整列された列に新たな元を挿入

最悪約 $N^2/2$ 回の比較と $N^2/4$ 回の交換  
計算コストが最悪の例と最小の例は？

BI OINFORMATICS  
BI OINFORMATICS  
BI O I NFORMATICS  
BI I O NFORMATICS  
BI I N OFORMATICS  
BI I N O OFORMATICS  
BI I N O O RFORMATICS  
BI I N O O R MFORMATICS  
BI I M N O O R A TICS  
ABFI I M N O O R TICS  
ABFI I M N O O R T ICS  
ABFI I M N O O R T C S  
ABCFI I I M N O O R T S  
ABCFI I I M N O O R S T

```
public void insertion() {
    for (int i = 1; i < N; i++) {
        int v = a[i];
        int j;
        for (j = i;
             0 < j && v < a[j - 1];
             j--) {
            a[j] = a[j - 1];
        }
        a[j] = v;
    }
}
```

insertion sort

要素の交換数が多くなる傾向

shellsort

離れた要素間での交換を予備的に実行



# Shellsort    h要素分はなれた要素の集まりを insertion sort

生物情報 情報システム概論 森下

h=13    **B**IOINFORMATICS**S**

h=4

**B**IOIN**N**FORMATICS  
B**I**OIN**F**ORMATICS  
B**F**OINI**O**RMATICS  
B**F**OIN**I**OR**R**MATICS  
**B**FOIN**N**IOR**M**ATICS  
B**F**OIM**I**ORN**A**TICS  
BAOIM**F**ORNIT**T**ICS  
BAO**I**M**F**OR**N**IT**I**CS  
**B**AOIM**F**OIN**I**TR**C**S  
BAOICFOIM**I**TRNS**S**

h=1

**B****A**OICFOIMITRNS  
A**B**OICFOIMITRNS  
A**B**O**I**CFOIMITRNS  
A**B**I**O****C**FOIMITRNS  
A**B****C**I**O****F**OIMITRNS  
A**B**C**F**I**O****O**IMITRNS  
A**B**C**F**I**O****O****I**MITRNS  
A**B**C**F**I**I****O****O****M**ITRNS  
A**B**C**F**I**I****M****O****O****I**TRNS  
A**B**C**F**I**I**I**M****O****O****T**RNS  
A**B**C**F**I**I**I**M****O****O****T****R**NS  
A**B**C**F**I**I**I**M****O****O****R****T****N**S  
A**B**C**F**I**I**I**M****N****O****O****R****T****S**  
A**B**C**F**I**I**I**M****N****O****O****R****S****T**

# Shellsort

```
public void shell() {  
    int h;  
    int f = 3;  
    for (h = 1; h <= N / f; h = f * h + 1);  
    // h = 1, 4, 13, 40, 121, ... < N  
  
    for (; 0 < h; h /= f) {  
        // h を f で割った商(整数部分)を h に代入.  
        // h = ..., 121, 40, 13, 4, 1  
  
        for (int i = h; i < N; i++) {  
            int v = a[i];  
            int j;  
            for (j = i; h <= j && v < a[j - h]; j = j - h) {  
                a[j] = a[j - h];  
            }  
            a[j] = v;  
        }  
    }  
}
```

# 9章 Quick Sort

# 分割統治法によるソート

```
public void quick() {
    quick(0, N-1);
}

public void quick(int l, int r) {
    if (l < r) {
        int i = partition(l, r);
        quick(l, i - 1);
        quick(i + 1, r);
    }
}

i = partition(l, r)
```

a[]を以下のように分割

- a[l], ..., a[i-1] は  
a[i] を超えない
- a[i+1], ..., a[r] は  
a[i] を下回らない

**S** ≤ **S**

BIOINFORMAT**I**C**S**

BIOINFORMA**C**I**T**S

BIOINFORMA**C**I**S**T

**I** ≤ **I**

B**I**OINFORMA**C****I**

B**C**OINFORMA**I**I

B**C**OINFORMA**A**I

B**C**A**I**NFORMO**I**I

B**C**A**I**N**F**ORMO**I**I

B**C**A**F****N****I**ORMO**I**I

B**C**A**F****I**IORMO**I****N**

## partition の実装

生物情報 情報システム概論 森下

```
public int partition(int l, int r) {
    int v = a[r];
    int i = l - 1; // i は左側から
    int j = r; // j は右側から
    int tmp; // 値の退避用
    for (;;) {
        // v より小さい左の要素はとばす
        while (a[++i] < v) {}
        // v より大きい右の要素はとばす
        while (v < a[--j] && i < j) {}
        // 交差したら終了
        if (i >= j) break;
        // a[i] と a[j] を交換
        tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
    }
    tmp = a[i];
    a[i] = a[r];
    a[r] = tmp;
    return i;
}
```

**S** ≤  $\xrightarrow{\quad}$   $\xleftarrow{\quad}$  ≤ **S**

BIOINFORMAT**I**C**S**

BIOINFORMAC**I**T**S**

BIOINFORMACI**S**T

**I** ≤  $\xrightarrow{\quad}$   $\xleftarrow{\quad}$  ≤ **I**

B**I**OINFORMAC**I**

BC**O**INFORMA**I**I

BC**O**INFORM**A**I I

BC**A**INFORM**O** I I

BCA**I**N**F**ORMO I I

BCA**F**N**I**ORMO I I

BCAF**I**IORMOIN**N**

## partition の実装

生物情報 情報システム概論 森下

```
public int partition(int l, int r) {
    int v = a[r];
    int i = l - 1; // i は左側から
    int j = r; // j は右側から
    int tmp; // 値の退避用
    for (;;) {
        // v より小さい左の要素はとばす
        while (a[++i] < v) {}
        // v より大きい右の要素はとばす
        while (v < a[--j] && i < j) {}
        // 交差したら終了
        if (i >= j) break;
        // a[i] と a[j] を交換
        tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
    }
    tmp = a[i];
    a[i] = a[r];
    a[r] = tmp;
    return i;
}
```

$S \leq$   $\leq S$

BIOINFORMAC**ITS**

i

j

BIOINFORMAC**I****ST**

$I \leq$   $\leq I$

BCA**F****I**ORMOII

i

j

BCAF**I**IORMOIN**N**

## quick sort の動作例

生物情報 情報システム概論 森下

BCA**F**

**B**CA **A** F

**A**CB F

A **C**B F

A **B**C F

II**O**RM**O**I**N**

II**I**RM**O**ON

III**R**M**O**ON

III**M**R**O**ON

IIIM**N**O**O**R

IIIM **N**O**O**R

A BC F IIIM NOOR ST

# 性能

最悪の場合

$a[]$  が既にソートされている

$N^2/2$  に比例する計算時間

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I | J |
| A | B | C | D | E | F | G | H | I |   |
| A | B | C | D | E | F | G | H |   |   |
| : |   |   |   |   |   |   |   |   |   |

配列が常に丁度約半分分割される場合： 比較回数  $C_N$

$$C_N = 2C_{N/2} + N \approx N \lg N$$

$$\lg = \log_2 \quad \ln = \log_e$$



ランダムな配列では 平均  $2N \ln N$  回の比較

$$C_1 = C_0 = 0$$

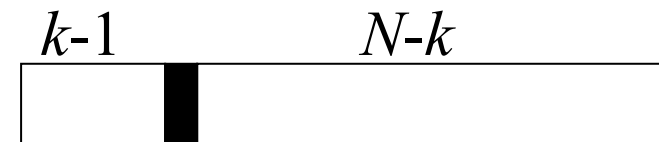
$$C_N = N + 1 + \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k})$$

$$C_0 + \dots + C_{N-1} = C_{N-1} + \dots + C_0 \text{ なので}$$

$$C_N = N + 1 + \frac{2}{N} \sum_{1 \leq k \leq N} C_{k-1}$$

両辺に  $N$  を掛ける

$$NC_N = N(N + 1) + 2 \sum_{1 \leq k \leq N} C_{k-1}$$



$k-1$  と  $N-k$  に分割される  
確率は  $1/N$

# 性能

$$NC_N = N(N+1) + 2 \sum_{1 \leq k \leq N} C_{k-1}$$

$N-1$ の場合

$$(N-1)C_{N-1} = (N-1)N + 2 \sum_{1 \leq k \leq N-1} C_{k-1}$$

引く

$$NC_N - (N-1)C_{N-1} = 2N + 2C_{N-1}$$

$$NC_N = (N+1)C_{N-1} + 2N$$

$N(N+1)$ で割る

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1} = \dots = \frac{C_2}{3} + \sum_{3 \leq k \leq N} \frac{2}{k+1}$$

$$\approx 2 \sum_{1 \leq k \leq N} \frac{1}{k} \approx 2 \int_1^N \frac{1}{x} dx = 2 \ln N = 1.38 \lg N$$

$$C_N \approx 2N \ln N$$

# 再帰呼出しの除去 スタックを使った実装

生物情報 情報システム概論 森下

```
public void iterQuick(){    // iteration で quick sort を実現
    int l = 0;           // 左端は 0
    int r = N-1;         // 右端は N-1
    Stack s = new Stack(2*N);
    for( ; ; ){
        while(l < r){
            int i = partition(l, r);
            if(i-l > r-i){    // 大きい方の分割を push
                s.push(l);    // 大きい分割がスタックに積まれるのは高々 log N
                s.push(i-1);
                l = i+1;
            }else{
                s.push(i+1);
                s.push(r);
                r = i-1;
            }
        }
        if(s.stackEmpty())
            break;
        r = s.pop();        // 積んだ順番の逆順に取り出す
        l = s.pop();
    }
}
```

# なぜ大きい方の分割をスタックに push するのか？

## スタックの節約

大きい分割がスタックに積まれる回数は高々約  $\lg N$

スタックに積まれる最初の分割の大きさは  $N/2$  以上

次は  $N/4$  以上

次は  $N/8$  以上

:

小さい分割をスタックに積むと、最悪約  $N$

# 小さい部分配列の処理

- 短い部分配列については  
insertion sort の方が実質的には速い
- quicksort と insertion sort の組合せが  
経験的には高速

# 分割用要素の選択

分割が偏る

BIOINFORMAT**I****C****S**

配列の最後の要素？

BIOINFORMAC**I****T****S**

BIOINFORMAC**I****S****T**

要素をランダムに選択  
偏りのない選択  
(乱数生成ルーチン)

B**I**OINFORMAC**I**

B**C**OINFORMA**I**I

BC**O**INFORM**A**I I

BC**A**INFORM**O**I I

BCA**I**N**F**ORMO I I

BCA**F**N**I**ORMO I I

BCAF**I**IORMOIN**N**

3つの要素を選び  
中央値を利用

# k番目 (k=0,1,...) に小さい要素を計算する方法

生物情報 情報システム概論 森下

配列をソートせずに計算

選択整列法の改良  
最初の  $k$  回目までを利用  
 $Nk$  に比例する時間

平均で  $N$  に比例する時間  
で計算する方法

quick sort の変更

観察:  $i = \text{partition}(l, r)$  で  
 $i$  が  $l$  から  $k$  番目の要素 ( $i=l+k-1$ ) ならば終了

$k=5$

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| B | I | O | I | N | F | O | R | M | A | T | I | C | S |
| A | I | O | I | N | F | O | R | M | B | T | I | C | S |
| A | B | O | I | N | F | O | R | M | I | T | I | C | S |
| A | B | C | I | N | F | O | R | M | I | T | I | O | S |
| A | B | C | F | N | I | O | R | M | I | T | I | O | S |
| A | B | C | F | I | N | O | R | M | I | T | I | O | S |
| A | B | C | F | I | I | O | R | M | N | T | I | O | S |
| A | B | C | F | I | I | I | R | M | N | T | O | O | S |
| A | B | C | F | I | I | I | M | R | N | T | O | O | S |
| A | B | C | F | I | I | I | M | N | R | T | O | O | S |
| A | B | C | F | I | I | I | M | N | O | T | R | O | S |
| A | B | C | F | I | I | I | M | N | O | O | R | T | S |
| A | B | C | F | I | I | I | M | N | O | O | R | T | S |
| A | B | C | F | I | I | I | M | N | O | O | R | S | T |

## k番目に小さい要素を計算する方法

生物情報 情報システム概論 森下

$kMin(0, 13, 8)$     <sup>0</sup>                      <sup>8</sup>                      <sup>13</sup>  
BIOINFORMATICS**S**  
:  
BIOINFORMACI**ST**  
 $kMin(0, 11, 8)$     BIOINFORMAC**I**  
:  
BCAF**I**IORMOIN  
 $kMin(4, 11, 4)$                       IIORMOIN**N**  
:  
IIIM**N**OOR

$a[]$  のソートは未完成

k番目に小さい要素は求まる

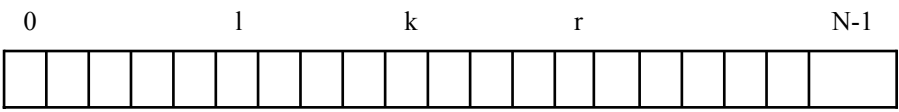


# k番目に小さい要素を計算する方法

生物情報 情報システム概論 森下

```
public int kMin(int k){
    if(k <= N){
        return kMin(0, N-1, k);
    }else{
        return -1;
    }
}

public int kMin(int l, int r, int k) {    // k 番目に小さい元を l から r の範囲で探索
    if (l < r) {
        int i = partition(l, r);
        int j = i-l+1;    // l, ..., i までには j=i-l+1 個の元が存在
        if(j == k){        // i が丁度 k 個目
            return a[i];
        }
        if(j > k){        // i より左に k 個以上
            return kMin(l, i - 1, k);
        }
        if(j < k){        // i より右に k-j 個以上
            return kMin(i + 1, r, k-j);
        }
    }
    if(l == r && k == 1){    // l が丁度 1 個目
        return a[l];
    }else{
        return -1;
    }
}
```



# 遺伝子発現解析

生物情報 情報システム概論 森下

- 生物情報実験2
- 遺伝子発現量パターンが近い順序でソート
- 1からk番目まで近いパターンを列挙

# 10章 Radix Sort

# ビット演算

B 00010

I 01001

O 01111

I 01001

N 01110

F 00110

O 01111

C 00011

M 01101

A 00001

I 01001

T 10100

R 10010

S 10011

c の右から b ビット目を値として返す

```
public int bits(int c, int b) {  
    int x = c;  
    int y = 0;  
    for(int i = 0; i < b; i++) {  
        y = x%2;  
        x = x/2;  
    }  
    return y;  
}
```

87654321

c = 00**1**01001      bits(c, 6)

# 基数交換法 (radix exchange sort)

生物情報 情報システム概論 森下

## ビットを左から右へと比較

|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
| B00010 | B00010 | B00010 | B00010 | A00001 | A00001 |
| I01001 | I01001 | A00001 | A00001 | B00010 | B00010 |
| O01111 | O01111 | C00011 | C00011 | C00011 | C00011 |
| I01001 | I01001 | F00110 | F00110 | F00110 | F00110 |
| N01110 | N01110 | N01110 | I01001 | I01001 | I01001 |
| F00110 | F00110 | I01001 | I01001 | I01001 | I01001 |
| O01111 | O01111 | O01111 | I01001 | I01001 | I01001 |
| R10010 | C00011 | O01111 | O01111 | M01101 | M01101 |
| M01101 | M01101 | M01101 | M01101 | O01111 | N01110 |
| A00001 | A00001 | I01001 | O01111 | O01111 | O01111 |
| T10100 | I01001 | I01001 | N01110 | N01110 | O01111 |
| I01001 | T10100 | T10100 | S10011 | S10011 | S10011 |
| C00011 | R10010 | R10010 | R10010 | R10010 | R10010 |
| S10011 | S10011 | S10011 | T10100 | T10100 | T10100 |

# radix exchange sort の実装

```
public void radixExchange() {
    radixExchange(0, N-1, 5);
}

public void radixExchange(int l, int r, int b) {
    if(l < r && b > 0) {
        int i = l;                // i は左側から
        int j = r;                // j は右側から
        while(i < j) {
            while( bits(a[i], b) == 0 && i < j ) {
                i++;                // 0 のうちは i は右へ
            }
            while( bits(a[j], b) == 1 && i < j ) {
                j--;                // 1 のうちは j は左へ
            }
            int tmp = a[i];        // a[i] == 1 と a[j] == 0 を交換
            a[i] = a[j];
            a[j] = tmp;
        }
        if( bits(a[r], b) == 0 ) { j++; }
        radixExchange(l, j-1, b-1);
        radixExchange(j, r, b-1);
    }
}
```

## radix exchange sort の弱点

最初の何ビットかが、どの文字も同じだと無駄

|      |      |      |      |      |
|------|------|------|------|------|
| 0000 | 0000 | 0011 | 1010 | 1010 |
| 0000 | 0000 | 0010 | 0010 | 1011 |
| 0000 | 0000 | 0011 | 1110 | 1010 |
| 0000 | 0000 | 0001 | 1110 | 1011 |
| ⋮    |      |      |      |      |

比較するビットをランダムに選ぶことができれば  
理想的には  $\lg N$  ビット調べたところで終了  
( $N$  は配列の要素数)

# 直接基数法 (straight radix sort) ビットを右から左へと比較

生物情報 情報システム概論 森下

|        |   |               |               |               |               |               |
|--------|---|---------------|---------------|---------------|---------------|---------------|
| B00010 | — | B00010        | T10100        | I01001        | A00001        | A00001        |
| I01001 |   | N01110        | I01001        | I01001        | B00010        | B00010        |
| O01111 |   | F00110        | I01001        | A00001        | R10010        | C00011        |
| I01001 |   | R10010        | M01101        | I01001        | C00011        | F00110        |
| N01110 |   | <u>T10100</u> | A00001        | B00010        | S10011        | I01001        |
| F00110 |   | I01001        | <u>I01001</u> | R10010        | T10100        | I01001        |
| O01111 |   | O01111        | B00010        | C00011        | <u>F00110</u> | I01001        |
| R10010 |   | I01001        | N01110        | <u>S10011</u> | I01001        | M01101        |
| M01101 |   | O01111        | F00110        | T10100        | I01001        | N01110        |
| A00001 |   | M01101        | R10010        | M01101        | I01001        | O01111        |
| T10100 |   | A00001        | O01111        | N01110        | M01101        | <u>O01111</u> |
| I01001 | — | I01001        | O01111        | F00110        | N01110        | R10010        |
| C00011 | — | C00011        | —             | C00011        | O01111        | S10011        |
| S10011 | — | S10011        | —             | S10011        | O01111        | T10100        |

注目するビットの値が同じ文字は、既存の順番を保存して移動



# straight radix sort の実装方針

生物情報 情報システム概論 森下

| a [ ]   |   | b [ ]          |
|---------|---|----------------|
| B 00010 |   | B 00010        |
| I 01001 |   | N 01110        |
| O 01111 |   | F 00110        |
| I 01001 |   | R 10010        |
| N 01110 |   | <u>T 10100</u> |
| F 00110 |   | I 01001        |
| O 01111 |   | O 01111        |
| R 10010 |   | I 01001        |
| M 01101 |   | O 01111        |
| A 00001 |   | M 01101        |
| T 10100 | ↗ | A 00001        |
| I 01001 | → | I 01001        |
| C 00011 | → | C 00011        |
| S 10011 | → | S 10011        |

## 分配計数法

対象ビットの値ごとにヒストグラムを生成

右端ビットの場合、各 a[i] に対して

```
count[bits(a[i], 1)]++;  
0 か 1
```

ヒストグラム → 累積をとる

```
count[0]=5    5  
count[1]=9    14
```

- a[13]=S の右端は 1  
count[1]=14 なので b[14-1]=a[14];  
count[1]--;
- a[12]=C の右端は 1  
count[1]=13 なので b[13-1]=a[13];  
count[1]--;
- a[10]=T の右端は 0  
count[0]=5 なので b[5-1]=a[10]  
count[0]--;

# straight radix sort の実装

```
public void straightRadix(){
    int nBits = 5;
    int[] b = new int[N];
    int[] count = new int[2];

    for(int pass = 1; pass <= nBits; pass++){
        count[0] = 0;
        count[1] = 0;
        for(int i = 0; i < N; i++){
            count[bits(a[i], pass)]++;
        }
        count[1] += count[0];
        for(int i = N-1; 0 <= i; i--){
            b[ count[bits(a[i], pass)]-1 ] = a[i];
            // count[bits(a[i], pass)] より 1 小さくないと
            // 正確なインデックスにならない
            count[bits(a[i], pass)]--;
        }
        for(int i = 0; i < N; i++){
            a[i] = b[i];
        }
    }
}
```

# 基数整列法 (radix sort) の性能

## 基数整列法 (radix sort)

- 基数交換法 (radix exchange sort)
- 直接基数法 (straight radix sort)

基数交換法では平均約  $N \lg N$  ビットの比較  
(quick sort と同様の解析)

どちらの基数整列法も  $b$  ビットのキー  $N$  個の整列に  
 $Nb$  回以下のビットしか調べない

# radix sort はゲノム配列のソートに向いている

## ゲノム配列のソート

ACGTCATCGTCGATCGTACG ...

A 00

T 01

G 10

C 11

|   |   |   |   |   |   |   |   |   |   |     |   |   |   |   |   |   |   |   |   |     |
|---|---|---|---|---|---|---|---|---|---|-----|---|---|---|---|---|---|---|---|---|-----|
| A | C | G | T | C | A | T | C | G | T | ... |   |   |   |   |   |   |   |   |   |     |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0   | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | ... |

ヒトゲノムの部分配列(長さ50)のソートをするなら

- 最初の長さ10塩基前後で radix sort
- 最初の長さ10塩基前後が共通の部分配列を quick sort

# 11章 順序キュー

# 順序キューとは

キューは先着順に並んでおり、最初に入ったデータを最初に取り出す  
(FIFO) First-In-First-Out

順序キューはデータが降順(もしくは昇順)で並んでいるキュー

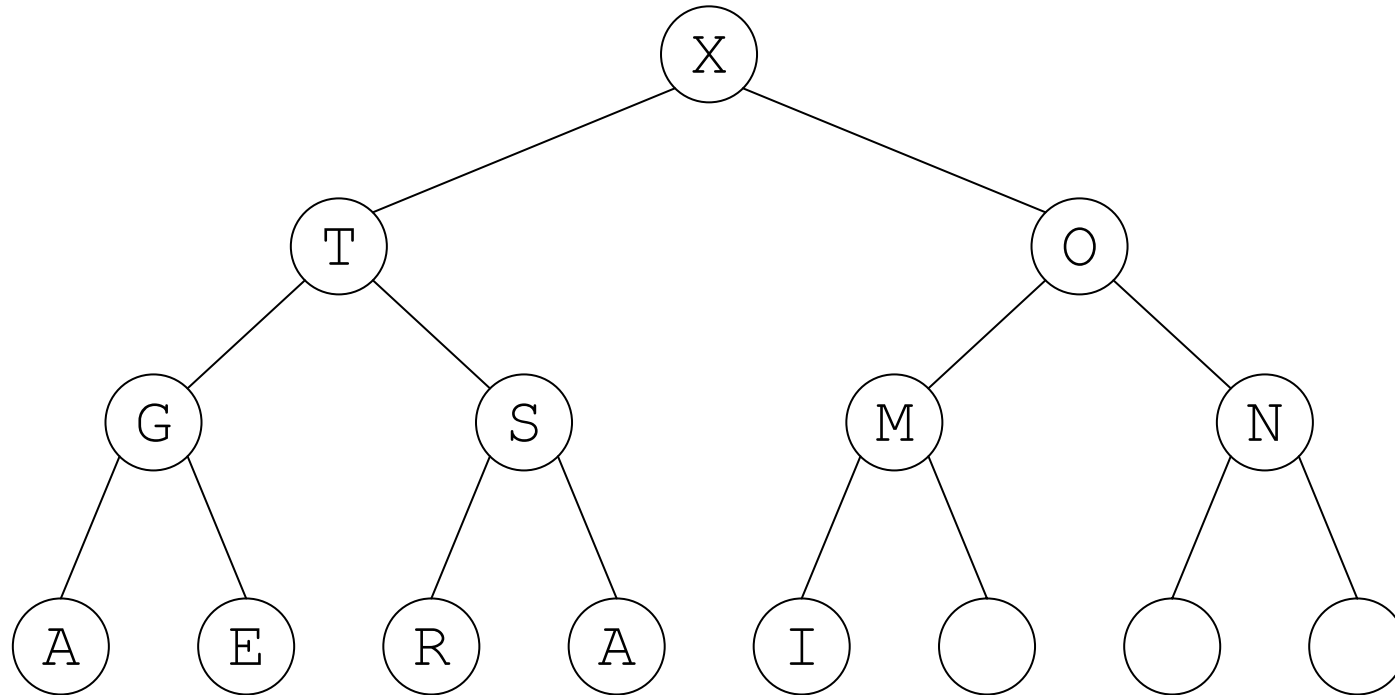
最大値を直ぐに取り出せる

順序キューに入っているデータ総数を  $N$  とすれば、  
データの追加と最大値の取出しが  $O(\lg N)$  時間で実行可能

オンライン処理向き

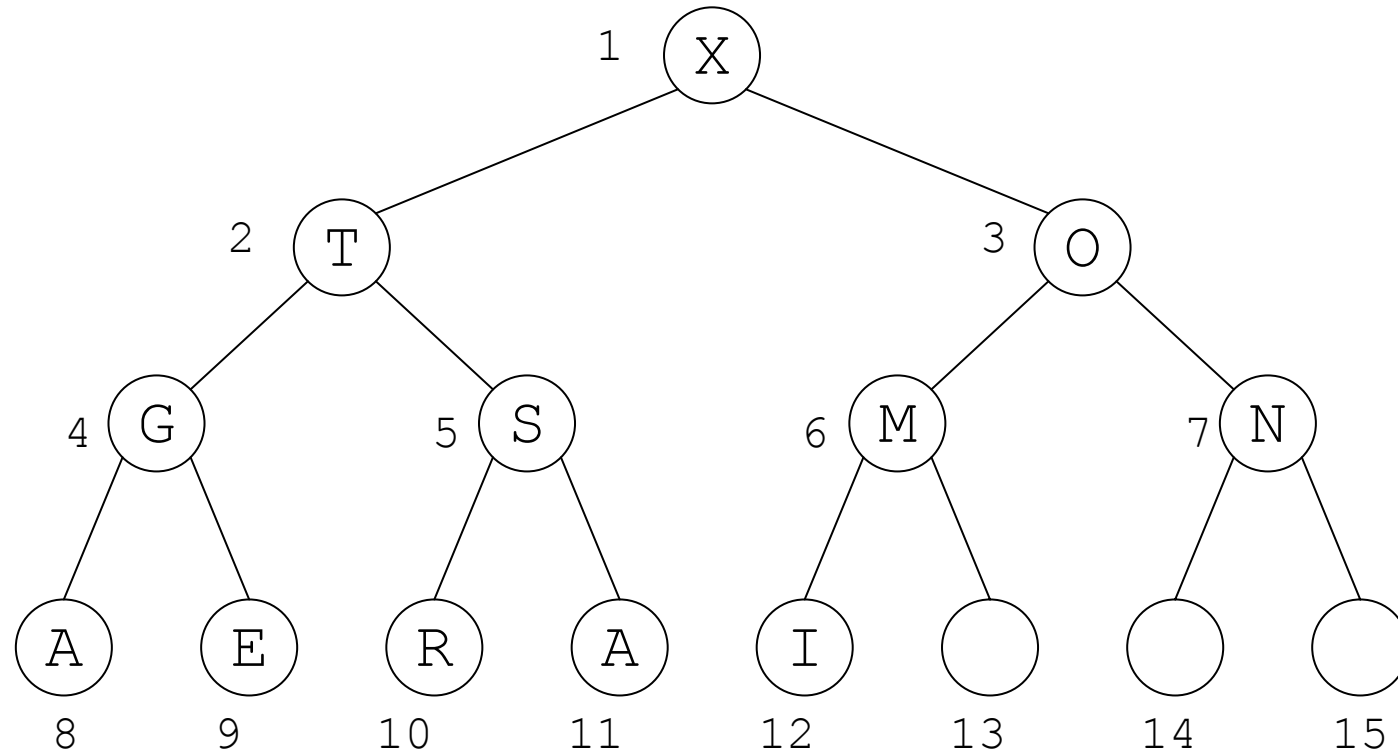
順序キューに  $N$  個のデータを追加すれば自然にソートでき  
 $O(N \lg N)$  時間で実行可能

# ヒープ



- 完全2分木の各ノードを、上から下へ、左から右へデータを埋める
- 各ノードの値は、子ノードの値(存在すれば)より大きいか等しい
- 根ノードの値が最大
- たとえば G より大きい R がレベルが低くても問題にしない

# ヒープの実装



|      |                     |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|------|---------------------|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| k    | 1                   | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| a[k] | X                   | T | O | G | S | M | N | A | E | R  | A  | I  |    |    |    |
| N=12 | N は値のある最後の場所のインデックス |   |   |   |   |   |   |   |   |    |    |    |    |    |    |

親ノードへのアクセス  $k/2$  (商 小数点以下切捨て)

$k$  の子ノードは  $k*2$ ,  $k*2+1$

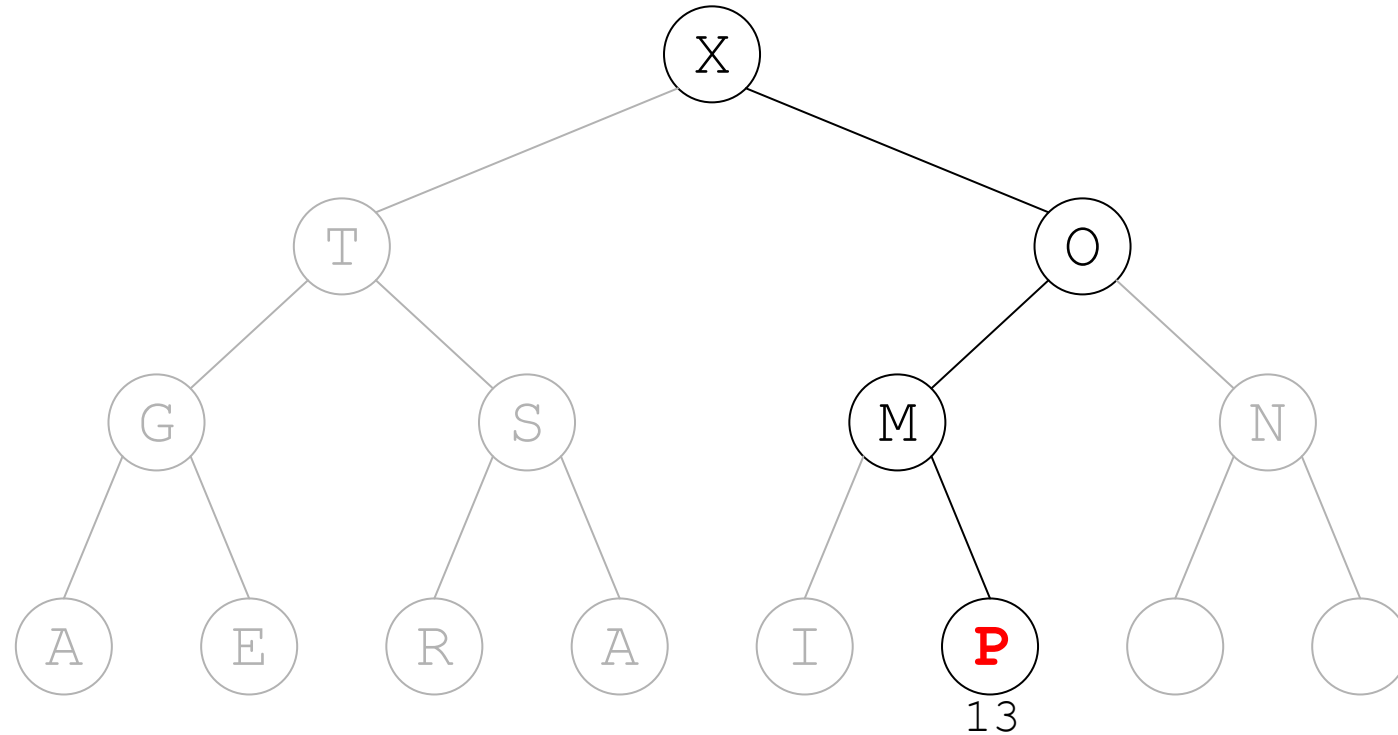


# Heap のデータ構造

生物情報 情報システム概論 森下

```
public class Heap {  
    private int N, size;          // 1 ≤ N ≤ size  
    private int[] a;  
  
    public Heap(int k) {  
        size = k;                // heap の大きさ  
        a = new int[size+1];  
        N = 0;  
        // N = 0 は空 N+1の場所  
        // (1, 2, 3, ...) に新しい元を入れる  
    }
```

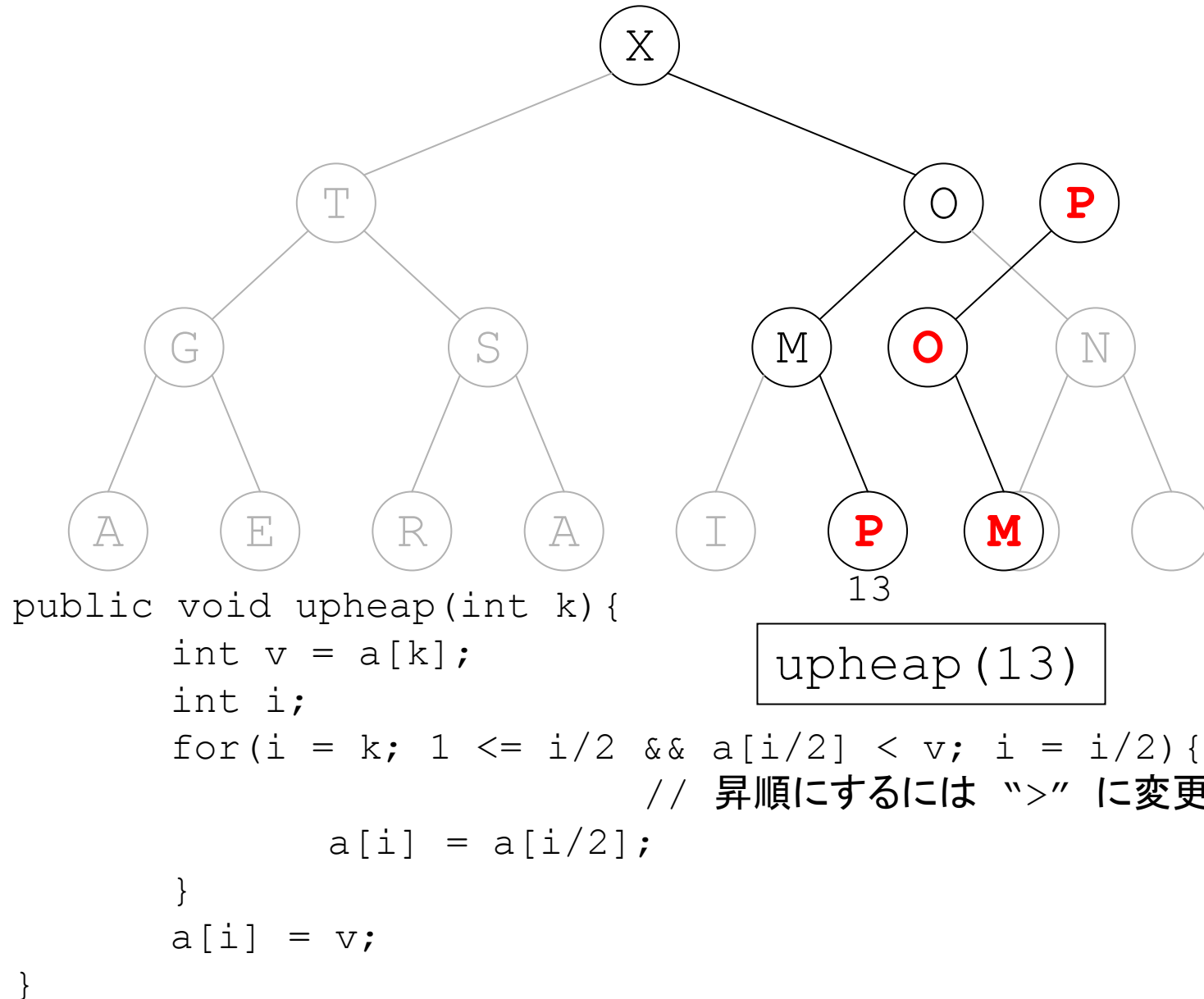
# ヒープへの挿入



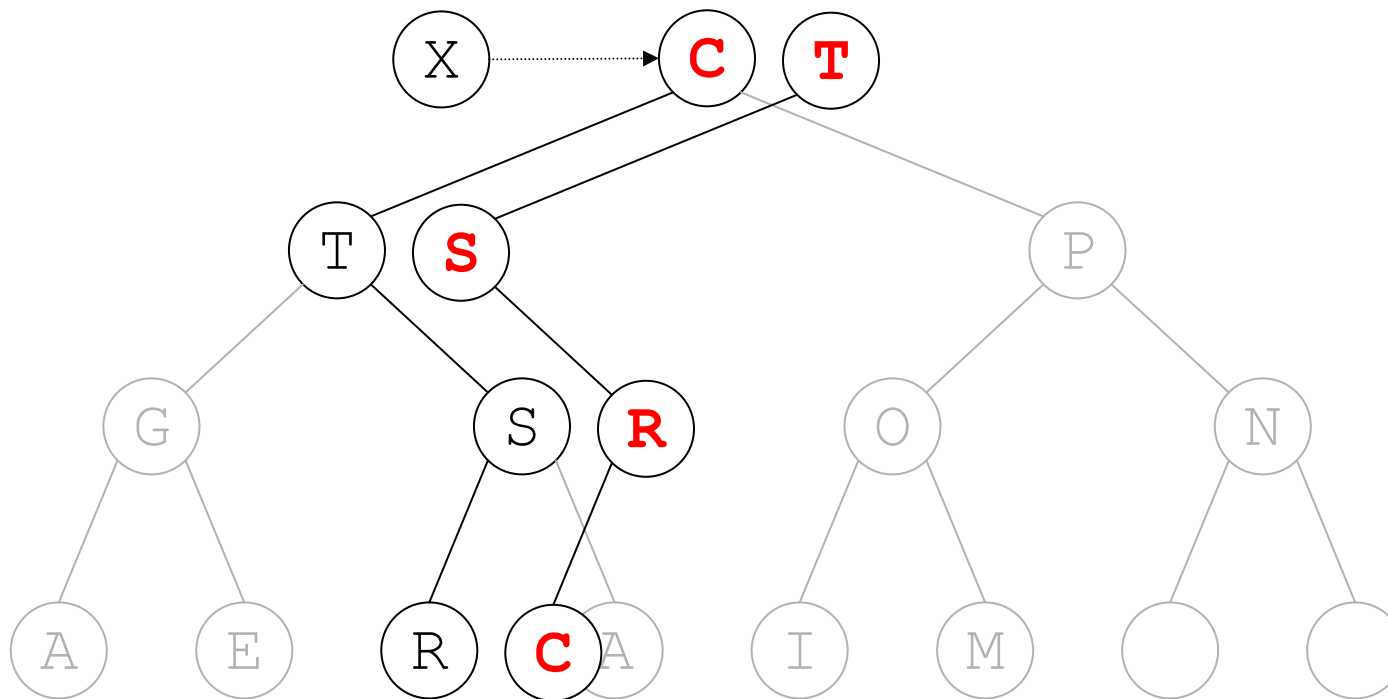
$N = 12, \text{insert}('P')$

```
public void insert(int v){  
    if(N < size){           // 入れる余裕があるか?  
        a[++N] = v;        // N を増やして空の場所をみつけて挿入  
        upheap(N);  
    }  
}
```

# ヒープへの挿入



# downheap



## downheap の実装

```
public void downheap(int k){
    int v = a[k];
    int i = k;
    while(i <= N/2){          // i には少なくとも左の子は存在
        int j = i*2;         // j は i の左の子

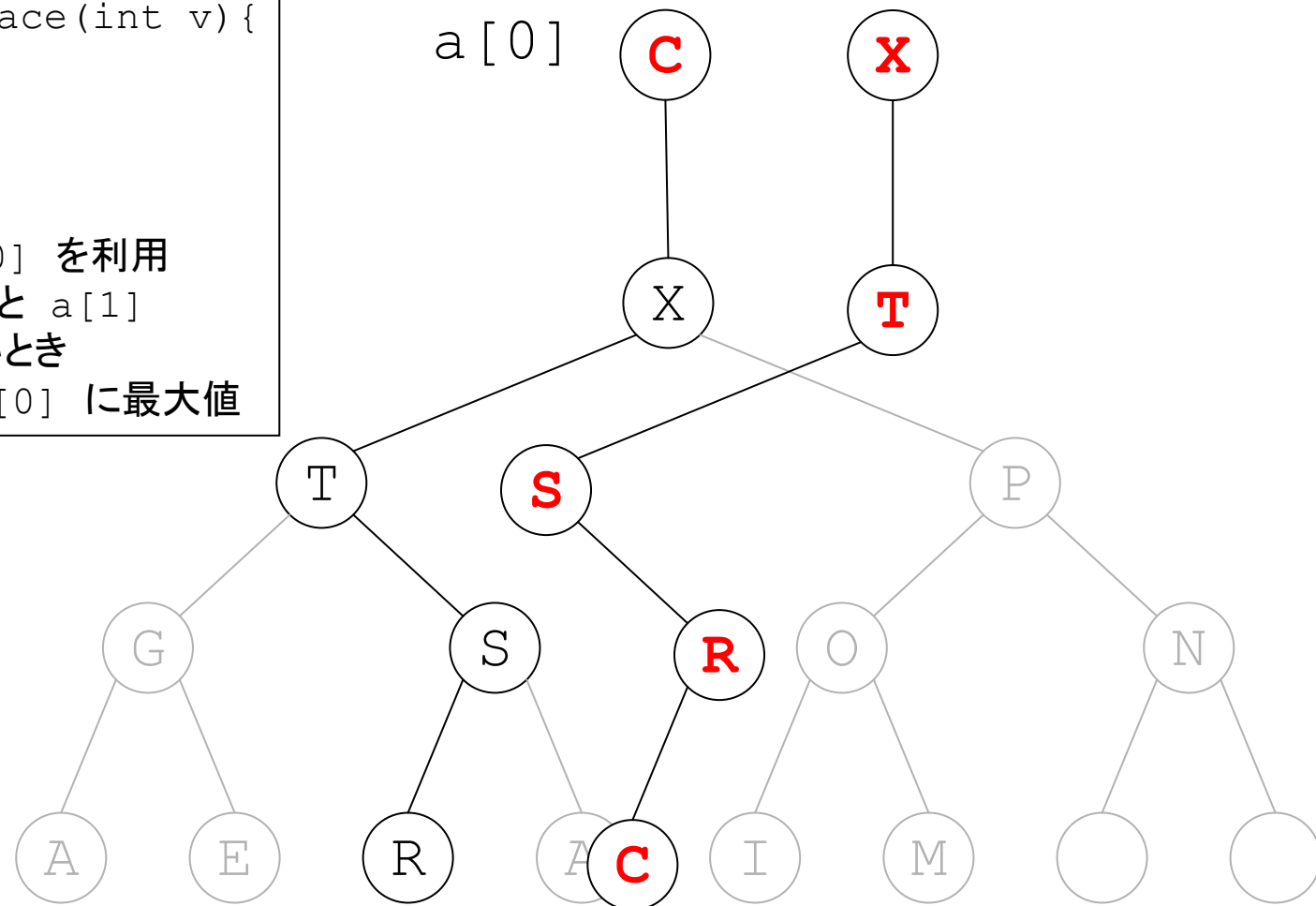
        if(j < N && a[j] < a[j+1]){
            // 昇順にするには a[j] > a[j+1]
            // i の右の子 j+1 の方が左の子 j より大きい
            j++;              // 大きい方が j になるように変更
        }
        if(v >= a[j]){ // v の downheap は終了 昇順にするには "<="
            break;
        }else{
            a[i] = a[j]; // 子 j を親にあげ、子 j の下へすすむ
            i = j;
        }
    }
    a[i] = v;              // v の落ち着き先 i を見つけたので置き換える
}
```

# 置き換え

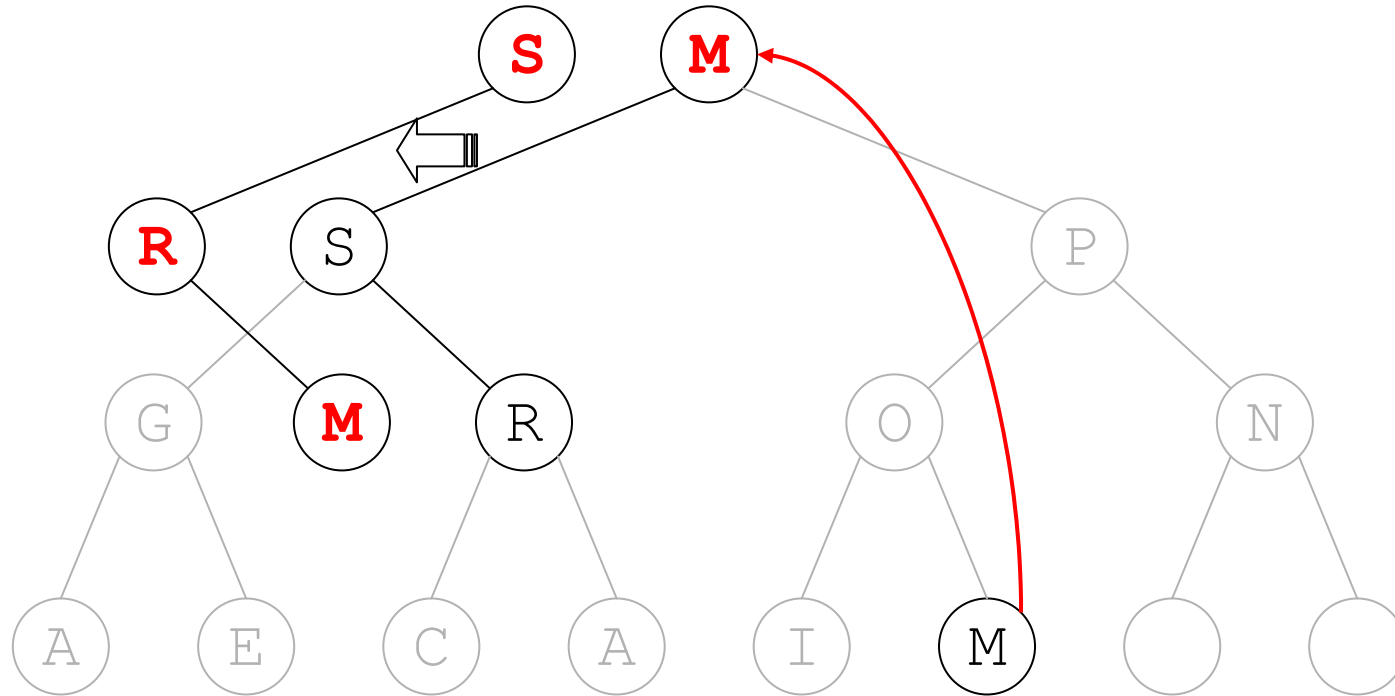
最大値を、それを超えない  
新しい値  $v$  で置き換える操作

```
public void replace(int v){  
    a[0] = v;  
    downheap(0);  
}
```

値の入っていない  $a[0]$  を利用  
 $a[0]$  の子は  $a[0]$  と  $a[1]$   
 $v$  が最大値を超えないとき  
 $\text{downheap}(0)$  は  $a[0]$  に最大値



# 最大値の削除



```
public int remove() {  
    int v = a[1];  
    a[1] = a[N--];  
    downheap(1);  
    return v;  
}
```

# ヒープへの操作の性能

insert, remove, replace は  $N$  要素のヒープに対して  
( $2 \lg N$ ) 回より少ない比較操作で実行可能

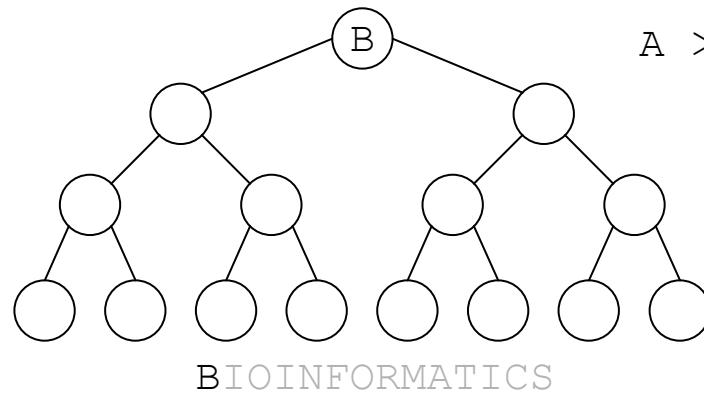
ヒープの高さは  $\lg N$   
downheap での比較回数は高々この2倍  
(2つの子との比較)



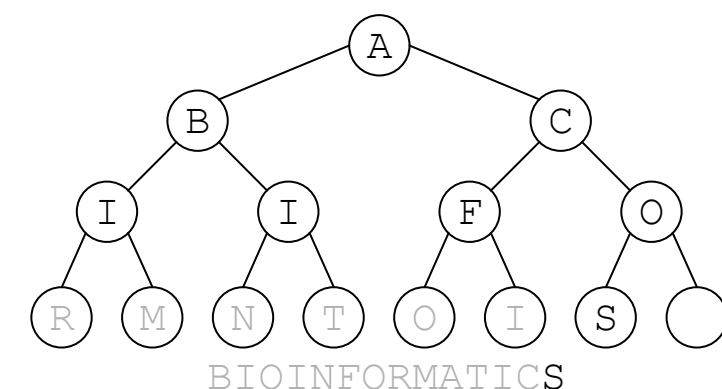
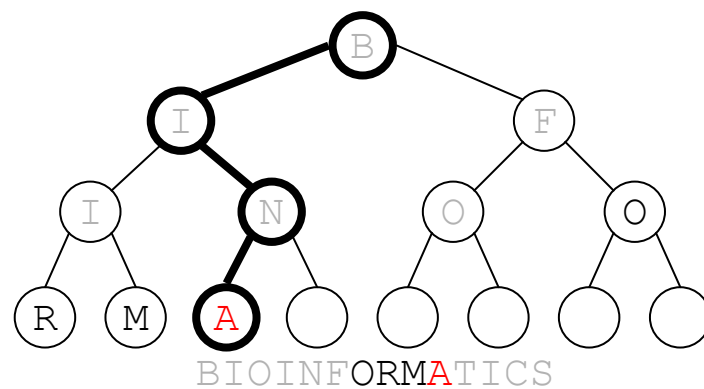
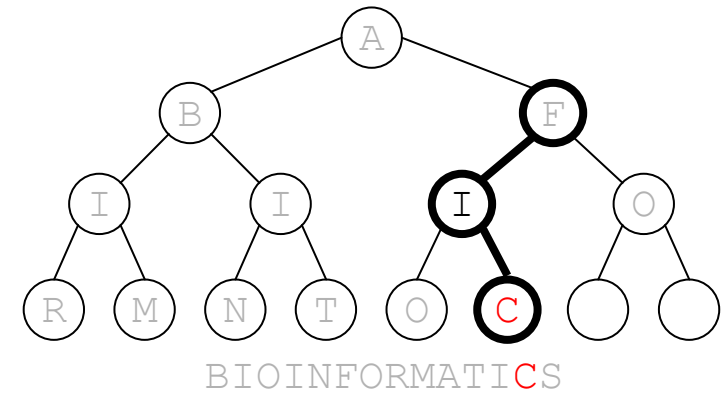
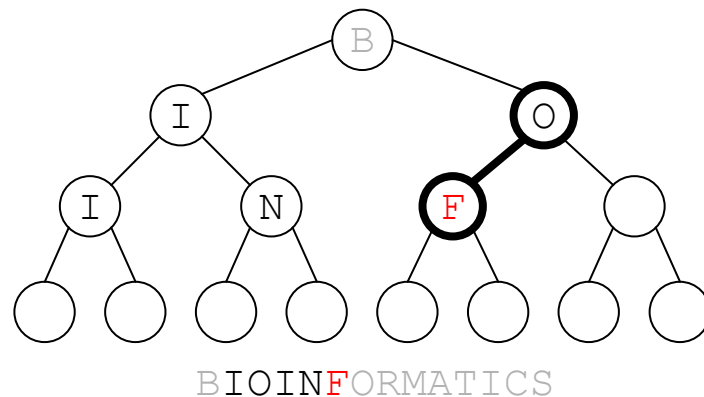
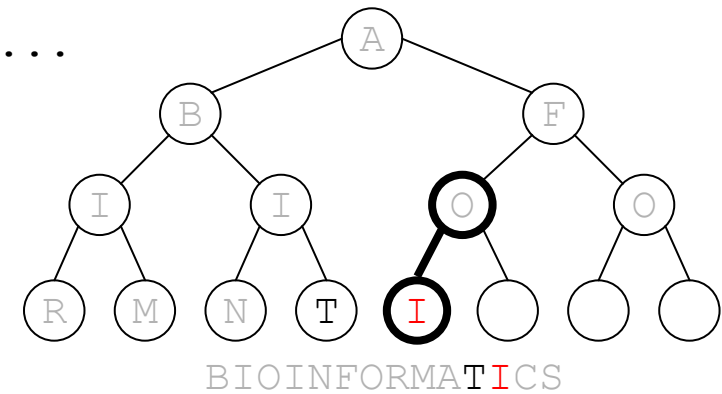
# ヒープソート

## 空のヒープに N 個の要素を挿入

生物情報 情報システム概論 森下

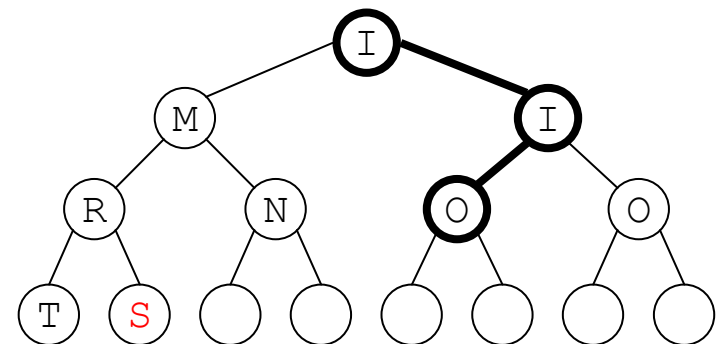
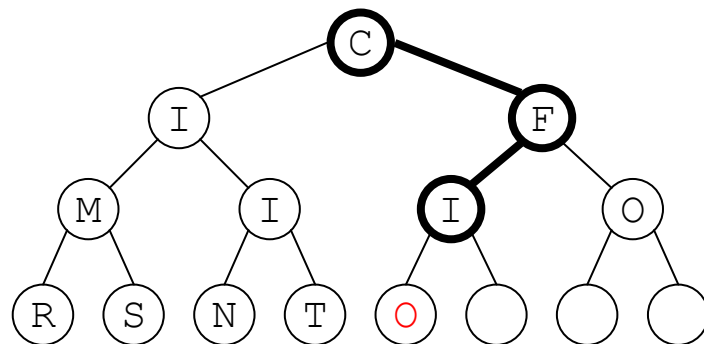
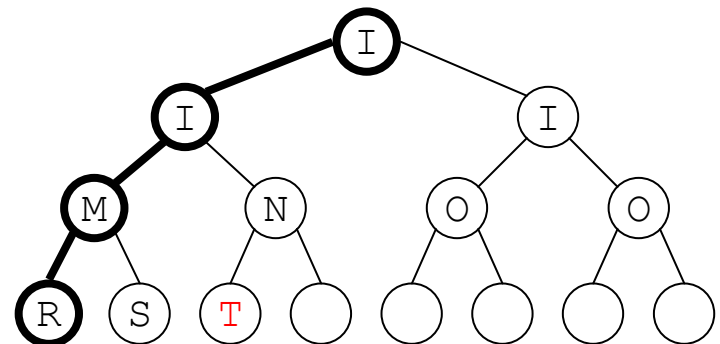
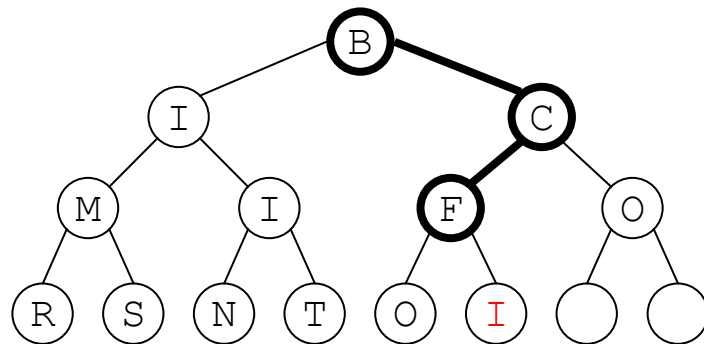
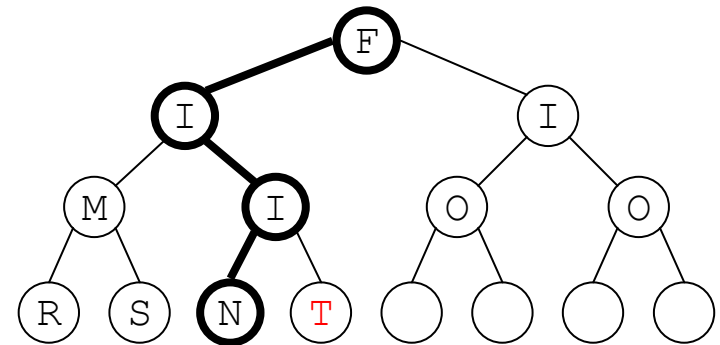
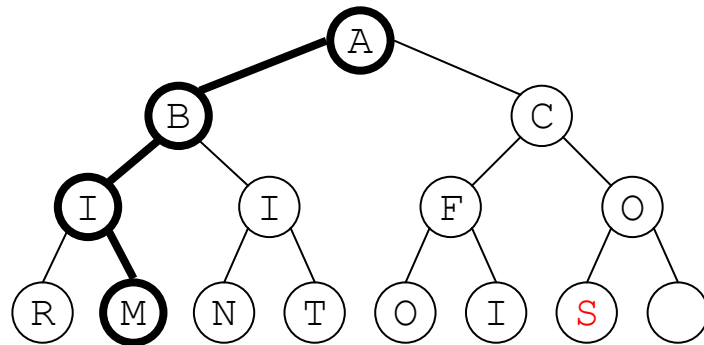


$A > B > C > \dots$



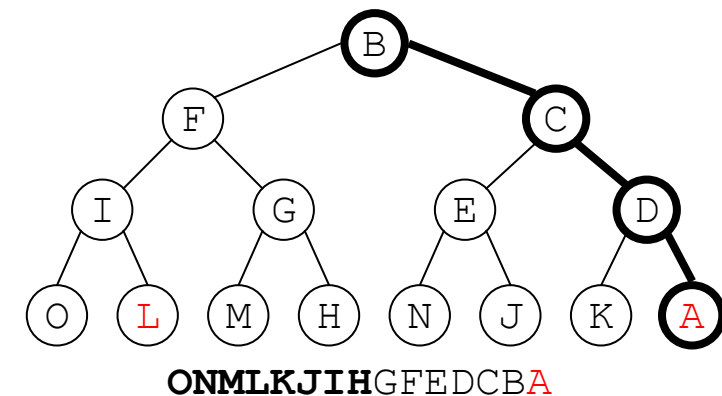
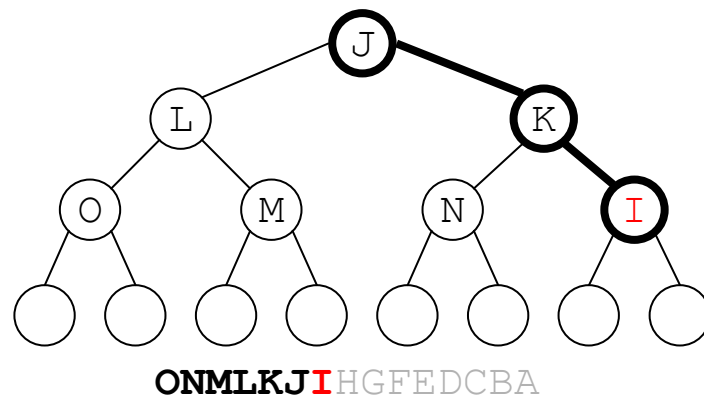
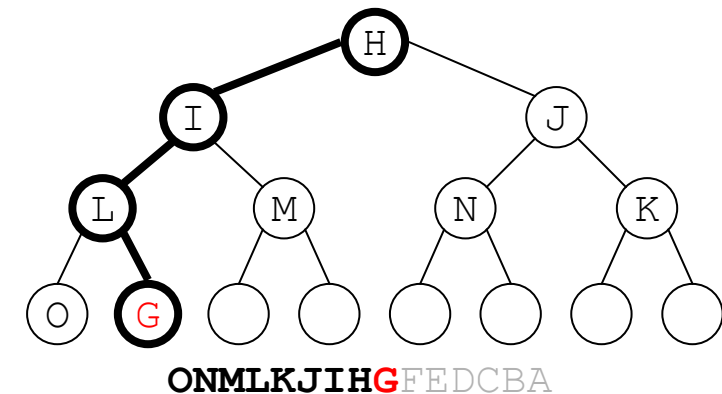
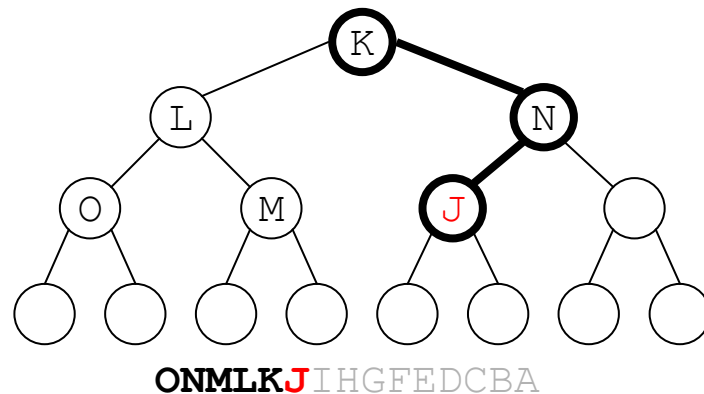
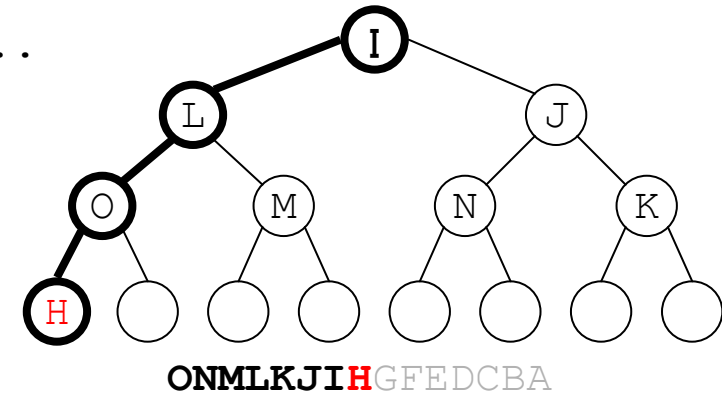
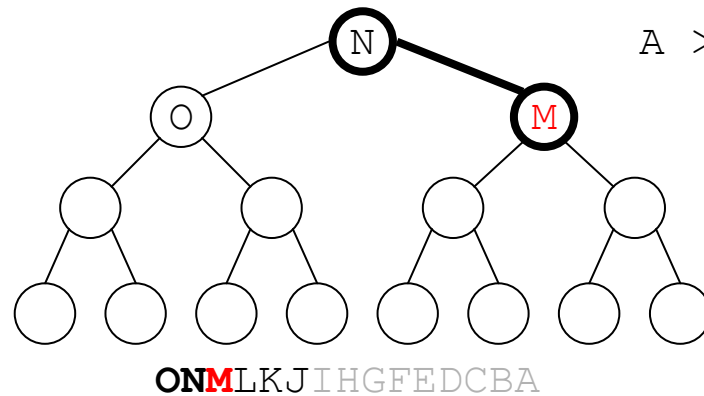
# ヒープソート 最大値の削除を N 回くりかえす

生物情報 情報システム概論 森下



# ヒープ作成で最も upheap が発生する場合

生物情報 情報システム概論 森下



## ヒープ作成で最も upheap が発生する場合の計算コスト

$$N = 2^d$$

$$C(N) = (d-1)2^{d-1} + (d-2)2^{d-2} + \dots + 3 \cdot 2^3 + 2 \cdot 2^2 + 1 \cdot 2^1$$

$$C(N) = 2C(N) - C(N)$$

$$= (d-1)2^d - (2^{d-1} + 2^{d-2} + \dots + 2^3 + 2^2 + 2^1)$$

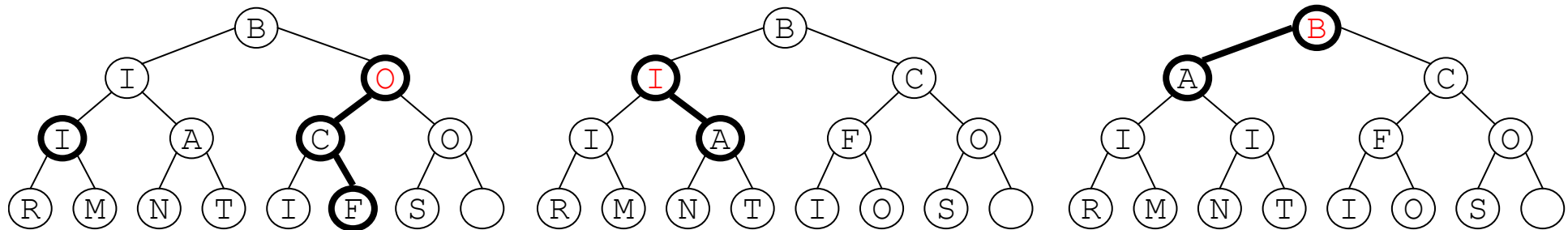
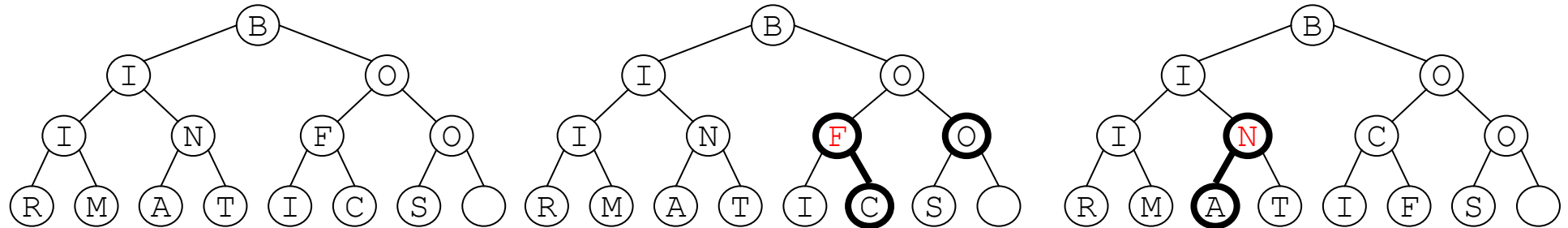
$$= (d-1)2^d - (2^d - 2)$$

$$= (d-2)2^d + 2$$

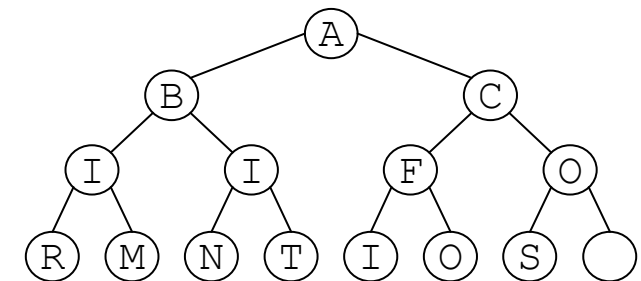
$$\cong N \lg N$$

# ボトムアップによるヒープの生成

生物情報 情報システム概論 森下



1. 入力を完全2分木の上から下へ、左から右へ格納
2. 内部ノードを右から左、下から上に巡り  
downheapを実行
3. この巡回により、内部ノード以下の部分木は  
常にヒープ(downheapの安全な適用が可能)



# ボトムアップによるヒープソートの計算コスト

ボトムアップによるヒープの生成は最悪でも要素数  $N$  の線形時間  
upheap を使う方法より減らせた！

$$N = 2^4 - 1 \quad C(N) = 3 \cdot 2^0 + 2 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3$$

$$N = 2^n - 1 \quad C(N) = \sum_{d=0}^{n-1} (n-1-d)2^d$$

$$C(N) = 2C(N) - C(N) = -(n-1) + 2^1 + \dots + 2^{n-1} = 2^n - n - 1 < N$$

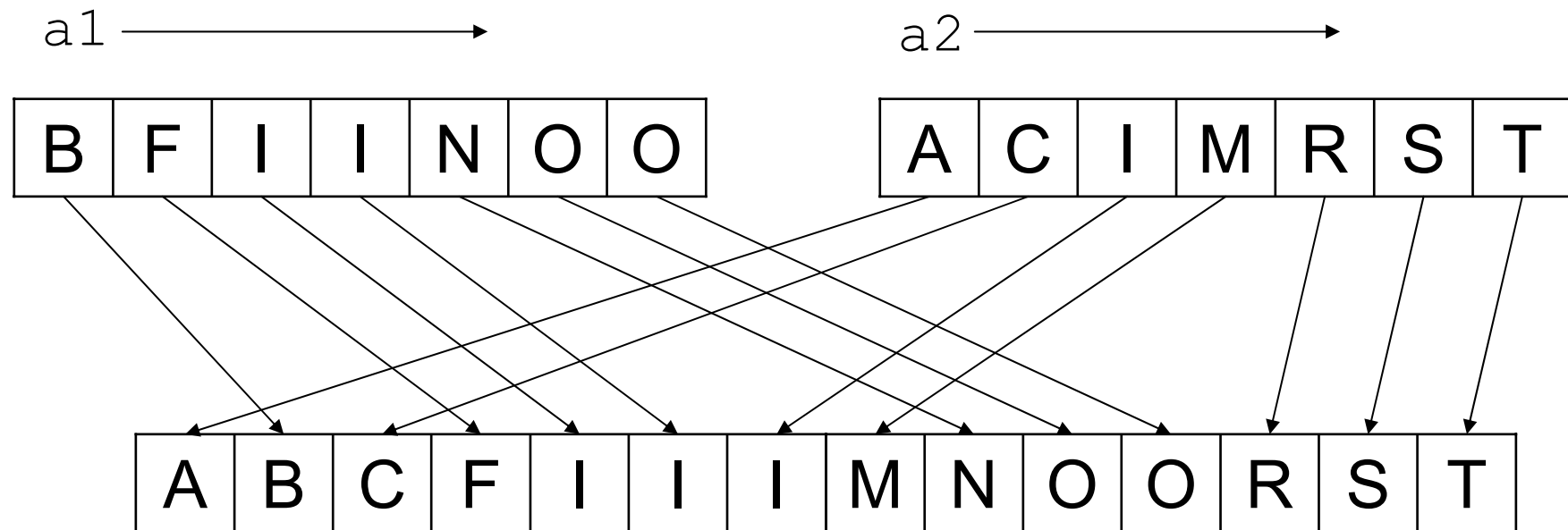
ヒープソート全体で、 $N$  個の要素の整列に、  
最悪でも  $2N \lg N$  回以下の比較しか実行しない

selection, insertion, bubble, quick sort  
はどれも最悪の場合  $N^2$  に比例する計算時間がかかる

# 12章 Merge Sort

## 着想 整列済み配列の併合

- 配列  $a_1$ ,  $a_2$  はソート済みであることを仮定
- 残っている先頭の2つの値で小さいほうを配列  $b$  に代入



- 配列ではなく、リンクをつかった実装方法も容易



# マージソート

**B**IOINFORMATICS

BI**I**ONFORMATICS

**BI**IOINFORMATICS

BIIO**F**NFORMATICS

BIIOF**N**FORMATICS

BIIO**FNO**FORMATICS

**BFI**INOOFORMATICS

BFIINOO**MR**ATICS

BFIINOOMR**AT**ICS

BFIINOO**AMRT**ICS

BFIINOOAMRT**CI**S

BFIINOOAMRTC**I**S

BFIINOOAMRT**CIS**

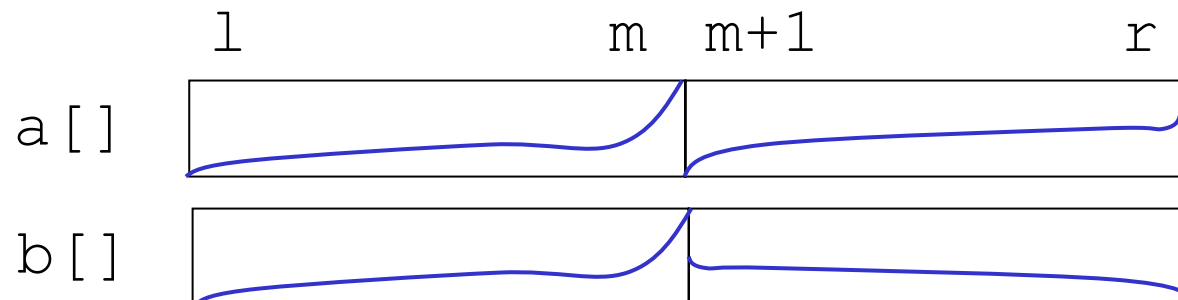
BFIINOO**ACIMRST**

**ABCFIIMNOORST**

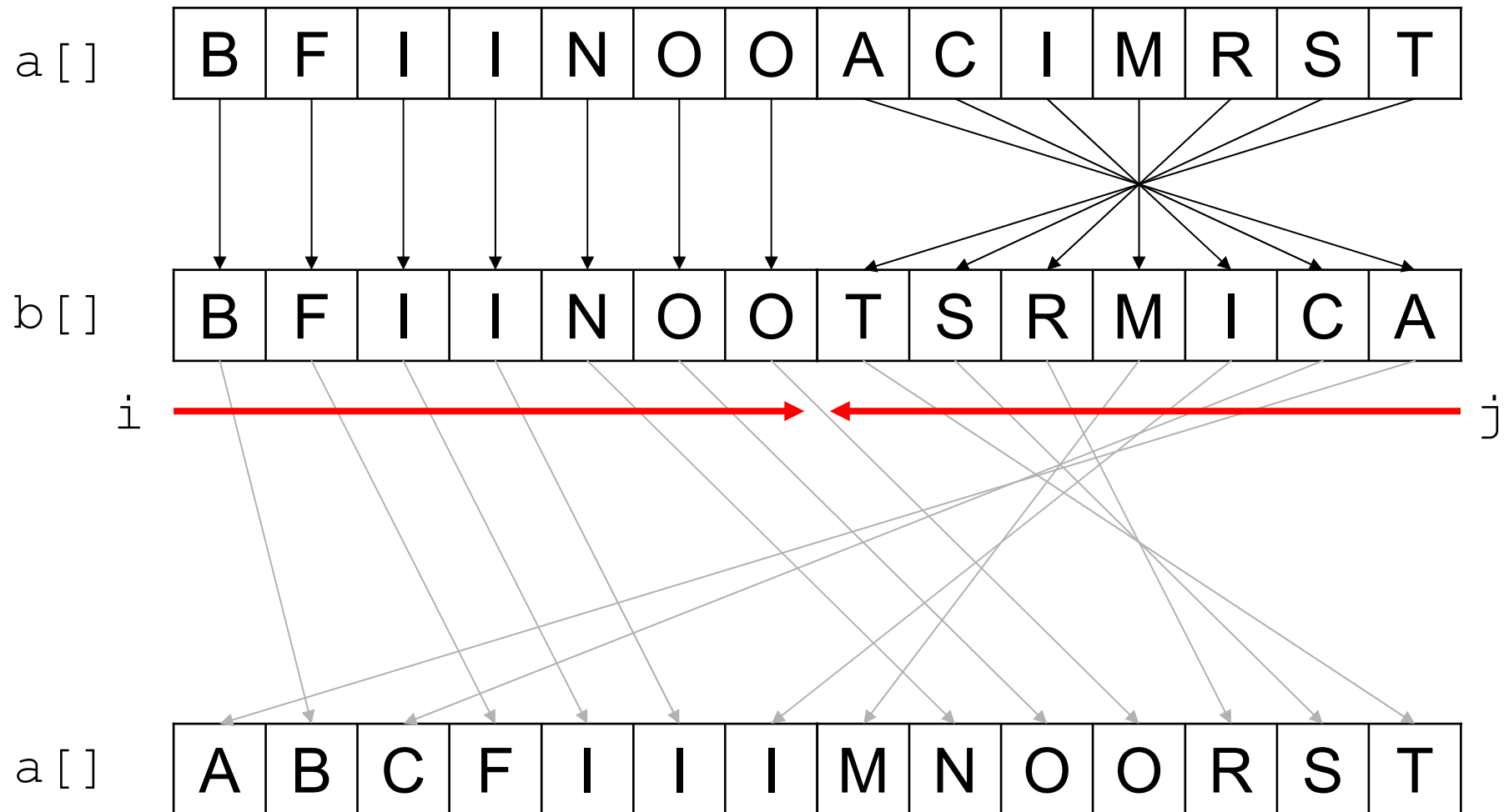
# マージソートの実装

生物情報 情報システム概論 森下

```
public void merge(int l, int r){
    if(l < r){
        int m = (r+1)/2;
        merge(l, m);           // 左半分を merge sort
        merge(m+1, r);         // 右半分を merge sort
        // 左半分と右半분을背中合わせにコピー
        for(int i = m+1; l < i; i--){           // 左半分はそのまま a から b へ
            b[i-1] = a[i-1];           }
        for(int j = m; j < r; j++){           // 右半分は逆順にコピー
            b[r+m-j] = a[j+1]; }
        // ソート
        int i = l;           // 左端から
        int j = r;           // 右端から
        for(int k = l; k <= r; k++){
            if(b[i] < b[j]){
                a[k] = b[i++];
            }else{
                a[k] = b[j--];
            } // 右半分もしくは左半分を超えても正常に動作
        }
    }
}
```



# 動作例



# ボトムアップ型マージソート

生物情報 情報システム概論 森下

BIOINFORMATICS  
BIIONFORMATICS  
BIIO**FN**ORMATICS  
BIIOFN**O**RMATICS  
BIIOFNOR**AM**TICS  
BIIOFNORAM**IT**CS  
BIIOFNORAMIT**C**S

BIIOFNORAMITCS  
BIIOFNORAMITCS  
BIIOFNOR**AIM**TCS  
BIIOFNORAIMTCS  
**BFI**INOORAIMTCS  
BFIINOOR**AC**IMST  
**ABC**FIIIMNOORST

# マージソートの性能

マージソートは  $N$  個の要素を最悪  $N \lg N$  回の比較で整列

ボトムアップ型では約  $N$  回の比較が  $\lg N$  回行われる

トップダウン型では  $N$  個の要素の比較回数を  $M_N$  とすれば  
分割統治の漸化式  $M_N = 2M_{N/2} + N$  より  $M_N$  は約  $N \lg N$

# 16章 ハッシュ法

# ハッシュ関数

レコードを識別するキーに算術演算（ハッシュ関数）をほどこし  
表（ハッシュ表）のアドレスに変換する

ハッシュ関数の利点

異なるアドレスに変換された 2つのキーは異なる

衝突処理

同じアドレスに変換された 2つのキーが異なる可能性が残る

ハッシュ表を大きく取ることによって衝突処理を回避

記憶領域と衝突頻度のトレードオフ

# ハッシュ関数

もっとも伝統的な方法

$$h(k) = k \bmod M$$

$M$  はハッシュ表のサイズで大きな素数であること

$i$  番目のアルファベットは、 $i$  を 5 ビットのコードで符号化

|   |       |   |       |   |       |
|---|-------|---|-------|---|-------|
| A | 00001 | K | 01011 | Y | 11001 |
| E | 00101 | L | 01100 | Z | 11010 |

$M=101$       キー "AKEY"

00001010110010111001 = 44217 (10進数)

$44217 \bmod M = 80$

$M=32$  だと  $44217 \bmod M = 25$

最後の文字の数になる "-----Y" の形は 25



# ハッシュ関数の高速計算

## mod の性質とホーナー法

生物情報 情報システム概論 森下

整数へ変換  
した値

下位5ビット  
の整数表現

|   |    |    |                  |
|---|----|----|------------------|
| V | 86 | 22 | $22 * 32^{10} +$ |
| E | 69 | 5  | $5 * 32^9 +$     |
| R | 82 | 18 | $18 * 32^8 +$    |
| Y | 89 | 25 | $25 * 32^7 +$    |
| L | 76 | 12 | $12 * 32^6 +$    |
| O | 79 | 15 | $15 * 32^5 +$    |
| N | 78 | 14 | $14 * 32^4 +$    |
| G | 71 | 7  | $7 * 32^3 +$     |
| K | 75 | 11 | $11 * 32^2 +$    |
| E | 69 | 5  | $5 * 32^1 +$     |
| Y | 89 | 25 | $25 \mod M$      |

大きな数の生成を避けたい

$$(( (22 * 32 + 5) * 32 + 18) * 32 + 25) \dots \mod M$$

$$= (( (22 * 32 + 5 \mod M) * 32 + 18 \mod M) * 32 + 25 \mod M) \dots$$

mod を加算、乗算前に実行しても同じ

# ハッシュ関数の実装

生物情報 情報システム概論 森下

```
public int hashFun(String s, int size){
    int answer = 0;
    for(int i = 0; i < s.length(); i++){
        answer =
            (answer*32+((int)s.charAt(i))%32)%size;
    }

    // ホーナー法で計算
    // 大文字アルファベットを 1-26 へ写像するため
    // 32 の余りを使っている
    return answer;
}
```

V E R Y L O N G K E Y

86 69 82 ...

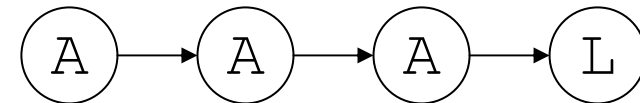
22 5 18 ...

# 衝突処理 分離連鎖法

key i hash(i mod M) M=11

|   |    |   |
|---|----|---|
| A | 1  | 1 |
| S | 19 | 8 |
| E | 5  | 5 |
| A | 1  | 1 |
| R | 18 | 7 |
| C | 3  | 3 |
| H | 8  | 8 |
| I | 9  | 9 |
| N | 14 | 3 |
| G | 7  | 7 |
| E | 5  | 5 |
| X | 24 | 2 |
| A | 1  | 1 |
| M | 13 | 2 |
| P | 16 | 5 |
| L | 12 | 1 |
| E | 5  | 5 |

| hash |   |   |   |   |
|------|---|---|---|---|
| 0    |   |   |   |   |
| 1    | A | A | A | L |
| 2    | M | X |   |   |
| 3    | N | C |   |   |
| 4    |   |   |   |   |
| 5    | E | E | E | P |
| 6    |   |   |   |   |
| 7    | G | R |   |   |
| 8    | H | S |   |   |
| 9    | I |   |   |   |
| 10   |   |   |   |   |



各 hash に key が  
いくつ挿入されるか  
わからない場合

各 key をリストで繋ぐ

# 衝突処理 線形探索法

生物情報 情報システム概論 森下

開番地法 (open addressing)

N 個のレコードを  $M (> N)$  個の hash 表に格納し、  
衝突処理に  $M - N$  個の空き領域を利用

線形探索法 (linear probing)

hash した値が、異なるキーで使われている場合、  
空いた場所が見つかるまで、ハッシュ表を順番に線形に探索

# 衝突処理 線形探索法

生物情報 情報システム概論 森下

| key | i  | hash(i mod M) |
|-----|----|---------------|
| A   | 1  | 1             |
| S   | 19 | 0             |
| E   | 5  | 5             |
| A   | 1  | 1             |
| R   | 18 | 18            |
| C   | 3  | 3             |
| H   | 8  | 8             |
| I   | 9  | 9             |
| N   | 14 | 14            |
| G   | 7  | 7             |
| E   | 5  | 5             |
| X   | 24 | 5             |
| A   | 1  | 1             |
| M   | 13 | 13            |
| P   | 16 | 16            |
| L   | 12 | 12            |
| E   | 5  | 5             |

N=17  
M=19

|    |   |    |
|----|---|----|
| 0  | S | 1  |
| 1  | A | 0  |
| 2  | A | 3  |
| 3  | C | 5  |
| 4  | A | 12 |
| 5  | E | 2  |
| 6  | E | 10 |
| 7  | G | 9  |
| 8  | H | 6  |
| 9  | I | 7  |
| 10 | X | 11 |
| 11 | E | 16 |
| 12 | L | 15 |
| 13 | M | 13 |
| 14 | N | 8  |
| 15 |   |    |
| 16 | P | 14 |
| 17 |   |    |
| 18 | R | 4  |

hash が衝突  
した場合、  
空いた場所まで  
移動する

T を探す場合  
 $i=20$   
 $i \bmod 19=1$

hash値 1~14  
まで探索して  
無いことを確認

# 線形探索法の実装

生物情報 情報システム概論 森下

```
public class Hash {
    private int M;           // ハッシュ用素数
    private int N;           // 入力ストリングの長さ
    private int L;           // ハッシュに使うストリングの長さ
    private String[] a;      // ハッシュ表 長さ L の部分列
    private int[] p;         // ハッシュ表 部分列が出現する位置
    public Hash(int size, int width, String s){
        M = size;
        N = s.length();
        L = width;
        a = new String[M];
        p = new int[M];
        for(int i = 0; i < M; i++){           // a, pos を空で初期化
            a[i] = "";
            p[i] = -1;
        }
        int i;
        for(i = 0; i+L-1 < N; i++){
            String t = s.substring(i,i+L); // i 番目から長さL の部分列を切り出す
            int k = hashFun(t, M);          // 切り出した配列が格納できる場所を探す
            while(0 < a[k].length()){       // a[k] が空でないならば先に進む
                k = (k+1) % M;
            }
            a[k] = t;
            p[k] = i;
        }
    }
} ...
```

# 線形探索法で目的の文字をサーチ

```
public void search(String t) {  
    int k = hashFun(t, M);  
  
    while (0 < a[k].length()) { // 空になるまで探す  
        if (t.equals(a[k])) { // 同一か否か？  
            System.out.print(" " + p[k]);  
        }  
        k = (k+1) % M;  
    }  
}
```

## 2重ハッシュ法

### 線形探索法の探索スピードを向上

hash が衝突した場合、空いた場所まで 1 つずつ移動するのではなく  $u$  個おきに移動する

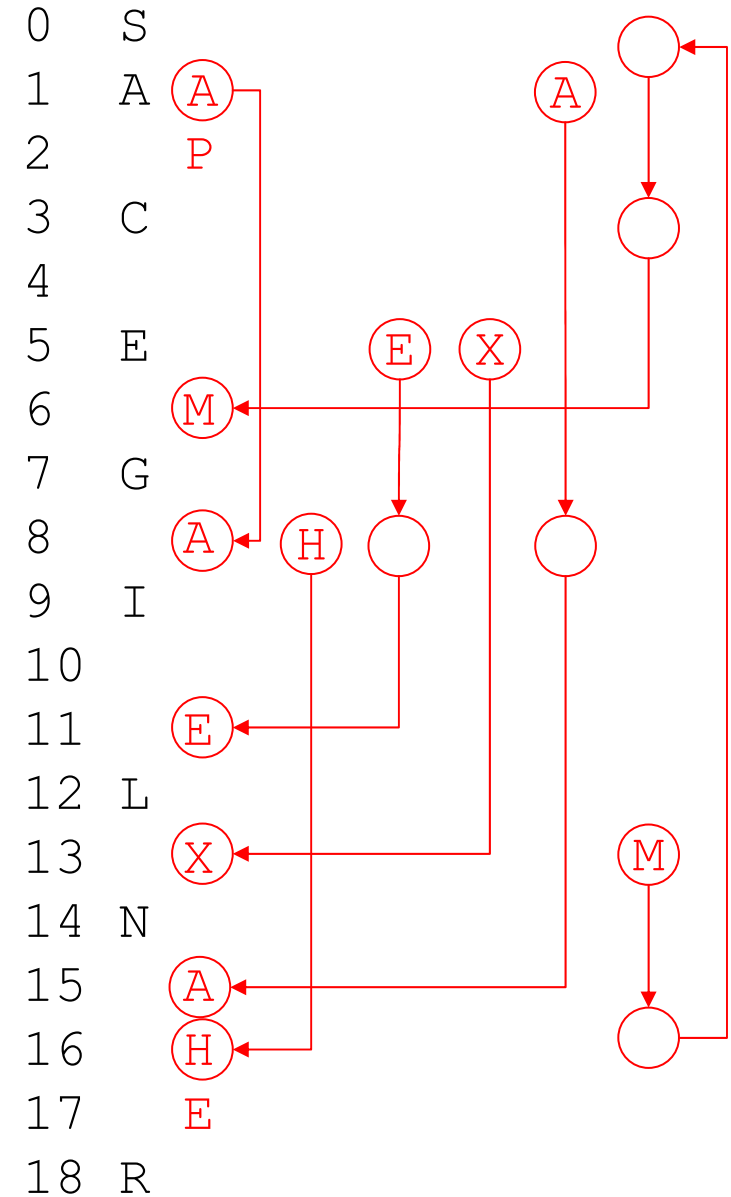
$u$  と  $M$  は互いに素であるように選択して、最終的にはすべての空いた場所を巡ることができるように配慮

実際は  $u = 8 - (k \bmod 8)$  のような簡単な関数で十分



# 2重ハッシュ法

| key | i  | hash1 | hash2 | hash1<br>$k \bmod 19$ | hash2<br>$8 - (k \bmod 8)$ |
|-----|----|-------|-------|-----------------------|----------------------------|
| A   | 1  | 1     | 7     |                       |                            |
| S   | 19 | 0     | 5     |                       |                            |
| E   | 5  | 5     | 3     |                       |                            |
| A   | 1  | 1     | 7     |                       |                            |
| R   | 18 | 18    | 6     |                       |                            |
| C   | 3  | 3     | 5     |                       |                            |
| H   | 8  | 8     | 8     |                       |                            |
| I   | 9  | 9     | 7     |                       |                            |
| N   | 14 | 14    | 2     |                       |                            |
| G   | 7  | 7     | 1     |                       |                            |
| E   | 5  | 5     | 3     |                       |                            |
| X   | 24 | 5     | 8     |                       |                            |
| A   | 1  | 1     | 7     |                       |                            |
| M   | 13 | 13    | 3     |                       |                            |
| P   | 16 | 16    | 8     |                       |                            |
| L   | 12 | 12    | 4     |                       |                            |
| E   | 5  | 5     | 3     |                       |                            |



# 索引

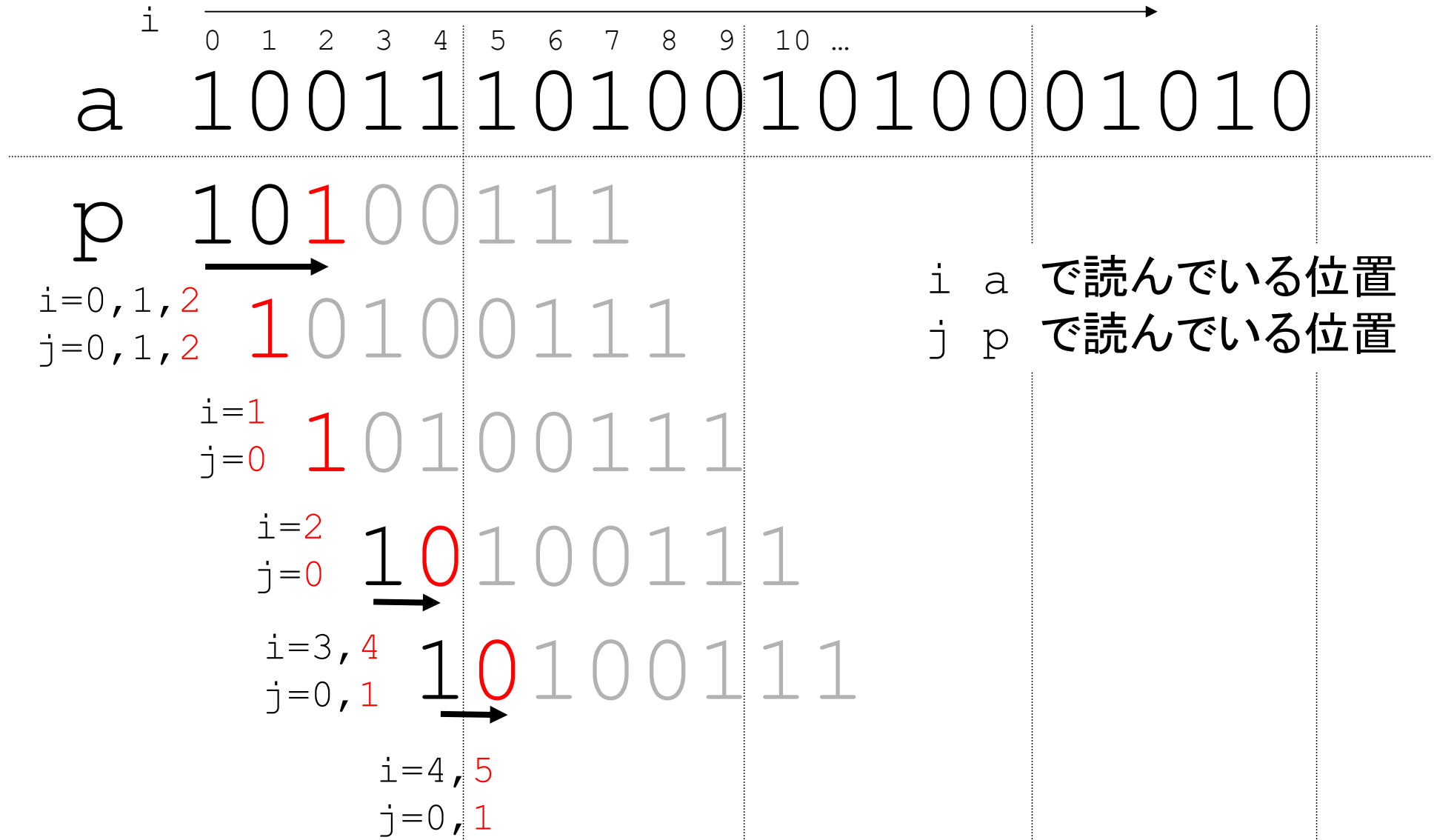
0                      5                      10                      15                      20  
A T G C G A A C T A T G C C T C T **A T** **G A** C G C T

|   |    |    |    |    |    |                 |
|---|----|----|----|----|----|-----------------|
| 0 | CT | 7  | 10 | TG | 10 | 20              |
| 1 | CT | 13 | 11 | TG | 18 | 21              |
| 2 | CT | 15 | 12 | CC | 12 | 22 <b>GA</b> 4  |
| 3 | TA | 8  | 13 | GC | 22 | 23 <b>AT</b> 0  |
| 4 | AA | 5  | 14 | CT | 23 | 24 GC 2         |
| 5 | TC | 14 | 15 |    |    | 25 <b>AT</b> 9  |
| 6 | AC | 6  | 16 | CG | 3  | 26 GC 11        |
| 7 | TA | 16 | 17 | CG | 21 | 27 <b>AT</b> 17 |
| 8 | AC | 20 | 18 |    |    | 28 <b>GA</b> 19 |
| 9 | TG | 1  | 19 |    |    |                 |

|    |           |                                 |
|----|-----------|---------------------------------|
| AT | A=1, T=20 | $1 \times 32 + 20 \mod 29 = 23$ |
| CT | C=3, T=20 | $3 \times 32 + 20 \mod 29 = 0$  |
| GA | G=7, A=1  | $7 \times 32 + 1 \mod 29 = 22$  |

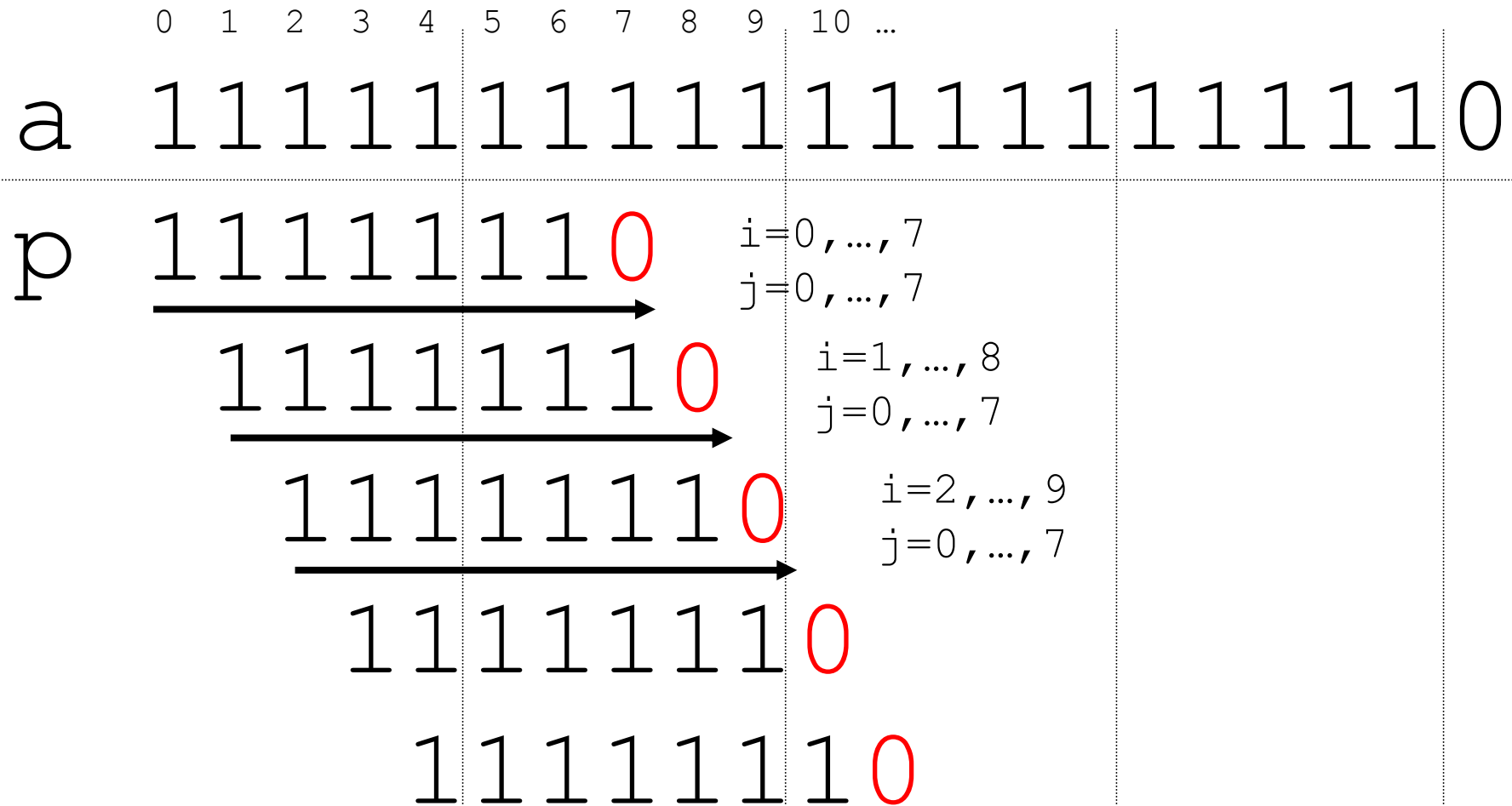
# 19章 文字列探索 (完全マッチ)

# brute-force algorithm



# brute-force algorithm

生物情報 情報システム概論 森下



最悪約 MN 回比較

# brute-force algorithm

```
public static int bruteSearch(String a, String p) {
    int N = a.length();
    int M = p.length();
    if(M > N) {                // パターン pの方が長いので完全にマッチするはずない
        return -1;
    }
    int i, j;
    for (i = 0, j = 0; i < N && j < M; ) {
        if (a.charAt(i) != p.charAt(j)) {
            i -= j - 1;        // マッチしなかったので前回の次の位置 (i-j)+1 から再開
            j = 0;              // パターンは、ふりだしに戻る
        } else {               // 1文字マッチしたら i, j とともに1つ進める
            i++;
            j++;
        }
    }
    if (j == M) {               // パターン p が最後まで完全にマッチした
        return i - M;          // パターン p のマッチが開始した位置を返す
    } else {
        return -1;             // p は完全にはマッチしなかった
    }
}
```

# Knuth – Morris – Pratt

生物情報 情報システム概論 森下

a 1 0 1 0 1 0 0 1 1 1

---

p 1 0 1 0 0 1 1 1

---

再実行の  
場所は？

1 0 1 0 0 1 1 1

brute-force

むだ

見落とし

1 0 1 0 0 1 1 1

1 0 1 0 0 1 1 1

むだなく

見落とし無く

# Knuth – Morris – Pratt

a    1 0 1 0 | 1 0 0 1 1 1  
                   $i$

p    1 0 1 0 | 0 1 1 1

$j = 4$

      1 0 | 1 0 0 1 1 1

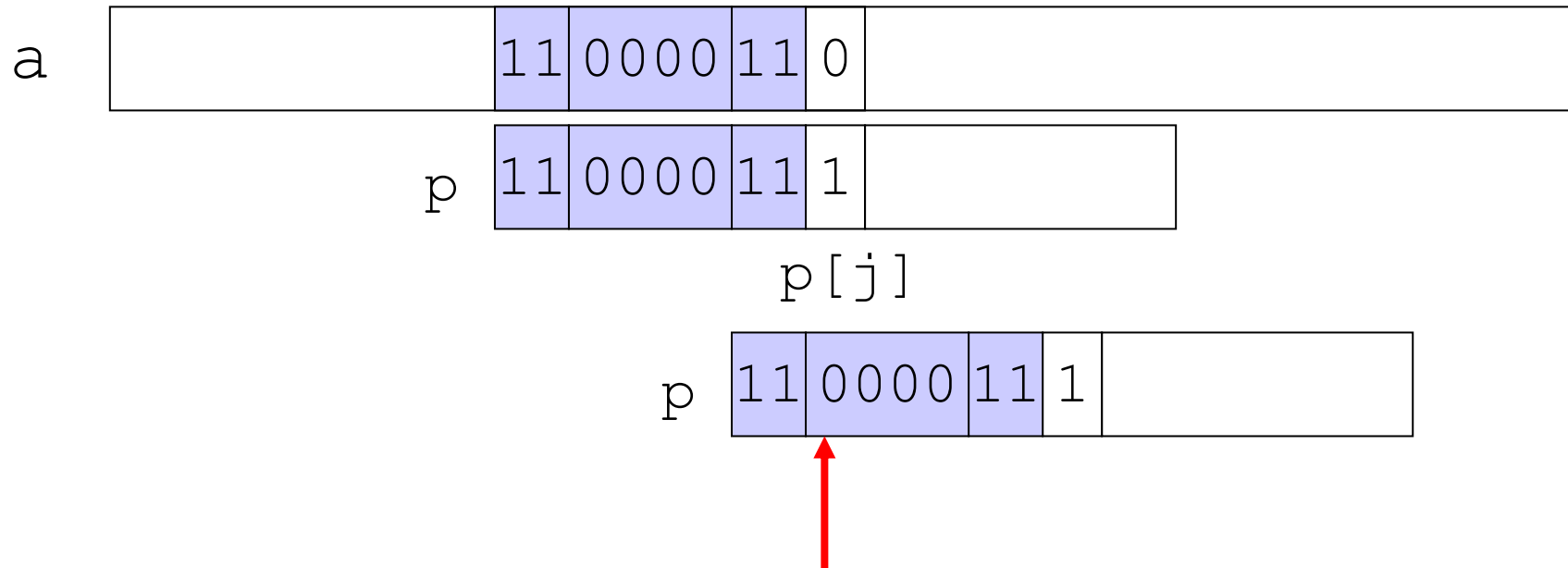
$j = 2$

$i$  を逆行させず固定し、  
パターンだけを横滑り  
( $j$  を巻き戻す)



# Knuth – Morris – Pratt

一致している部分列  $a[i]$



$p[j]$

$p$  11 0000 11 1

next[j]  
 $p[j]$  で一致しないとき  
検索を再実行する位置

$a$  を見ないで  $p$  から計算できる

# Knuth – Morris – Pratt

10100111

10100111

next[0] = -1

10100111

10100111

next[1] = 0

10100111

10100111

next[2] = 0

10100111

10100111

next[3] = 1

青 一致 赤 不一致 灰 未走

# Knuth – Morris – Pratt

1 0 1 0 0 1 1 1

1 0 1 0 0 1 1 1

next[4] = 2

1 0 1 0 0 1 1 1

1 0 1 0 0 1 1 1

next[5] = 0

1 0 1 0 0 1 1 1

1 0 1 0 0 1 1 1

next[6] = 1

1 0 1 0 0 1 1 1

1 0 1 0 0 1 1 1

next[7] = 1

|      |    |   |   |   |   |   |   |   |
|------|----|---|---|---|---|---|---|---|
|      | 0  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| next | -1 | 0 | 0 | 1 | 2 | 0 | 1 | 1 |

# Knuth – Morris – Pratt

|                 | $i$ | $j$ | $\text{next}[i]$ |
|-----------------|-----|-----|------------------|
| 1 0 1 0 0 1 1 1 | 0   |     |                  |
| 1 0 1 0 0 1 1 1 |     | -1  | -1               |
| 1 0 1 0 0 1 1 1 | 1   |     |                  |
| 1 0 1 0 0 1 1 1 |     | 0   | 0                |
|                 |     | -1  |                  |
| 1 0 1 0 0 1 1 1 | 2   |     |                  |
| 1 0 1 0 0 1 1 1 |     | 0   | 0                |
| 1 0 1 0 0 1 1 1 | 3   |     |                  |
| 1 0 1 0 0 1 1 1 |     | 1   | 1                |

# Knuth – Morris – Pratt

|          | $i$ | $j$ | $\text{next}[i]$ |
|----------|-----|-----|------------------|
| 10100111 | 4   |     |                  |
| 10100111 |     | 2   | 2                |
| 10100-   |     | 0   |                  |
|          |     | -1  |                  |
| 10100111 | 5   |     |                  |
| 1010-    |     | 0   | 0                |
| 1010011  | 6   |     |                  |
| 1010     |     | 1   | 1                |
| 1010-    |     | 0   |                  |

# Knuth – Morris – Pratt

```
public static int kmpSearch(String a, String p){
    int N = a.length();
    int M = p.length();
    if(M > N){ return -1;}
    int i, j;
    // next を初期化
    int[] next = new int[M+1];
    next[0] = -1;
    for(i = 0, j = -1; i < M; i++, j++, next[i]=j){
        while( (0 <= j) && (p.charAt(i) != p.charAt(j)) ){
            j = next[j];
        }
    }
    // マッチするか否かを探索
    for(i = 0, j = 0; (i < N) && (j < M); i++, j++){
        while( (0 <= j) && (a.charAt(i) != p.charAt(j)) ){
            j = next[j];
        }
    }
    if(j == M)
        return i - M;
    else
        return -1;
}
```

# Knuth – Morris – Pratt

生物情報 情報システム概論 森下

10011101001010000101000111  
10**1**00111  
  **1**0100111  
    **1**0100111  
      **1**0100111  
        10100**1**1  
          10100**1**11

|      |    |   |   |   |   |   |   |   |
|------|----|---|---|---|---|---|---|---|
| j    | 0  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| next | -1 | 0 | 0 | 1 | 2 | 0 | 1 | 1 |

Diagram illustrating the iterative step of the Longest Common Subsequence (LCS) algorithm. The strings are:

- $a = 11111111111111111111$
- $p = 11111111111111111111$

The diagram shows the iterative step where the LCS is updated for each character in  $p$  based on the previous state. The current state is shown with the last character of  $p$  (1) and the last character of  $a$  (1) highlighted in red. The previous state is shown with the last character of  $p$  (1) and the last character of  $a$  (1) highlighted in red.



# Knuth – Morris – Pratt

性質 文字比較  $a[i] \neq p[j]$  の  
実行回数は  $2N$  回以下

$i++$ ,  $j++$  の実行回数(高々  $N$ )と  $j = \text{next}[j]$  の実行回数の和

$j++$  は 1 ふえる

$j = \text{next}[j]$  は 1 以上減り、 $j \geq -1$

$j = \text{next}[j]$  は  $j++$  より実行回数が少ない

$j++$  は高々  $N$  回実行される．  $j = \text{next}[j]$  の実行回数は高々  $N$  回．

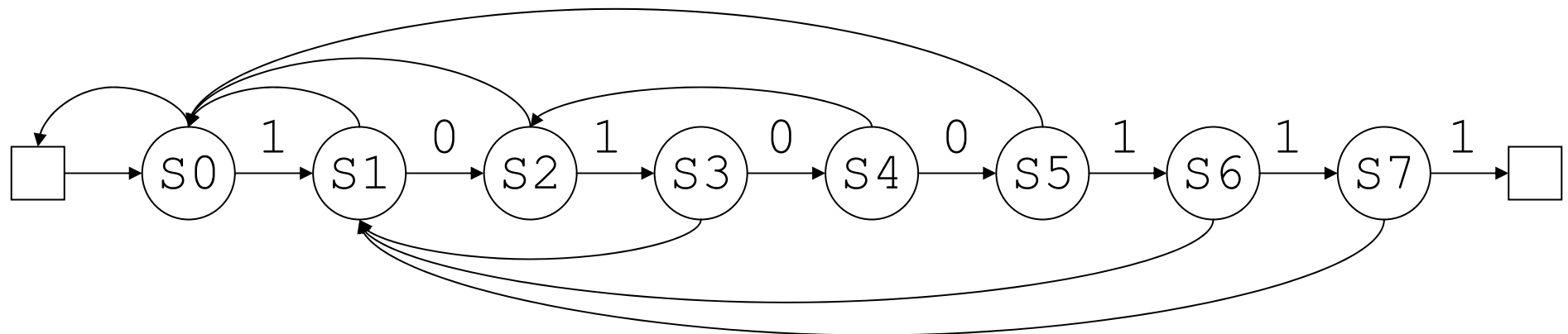
性質  $\text{initnext}(p)$  の文字比較  
回数は  $2M$  回以下

証明は同様

# Knuth – Morris – Pratt

|      |    |   |   |   |   |   |   |   |
|------|----|---|---|---|---|---|---|---|
|      | 0  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| next | -1 | 0 | 0 | 1 | 2 | 0 | 1 | 1 |

## 有限状態機械として表現



S3 は  $p[3]$  を  $a[i]$  と比較しようとしている状態  
もどる辺が  $next[j]$  を表現

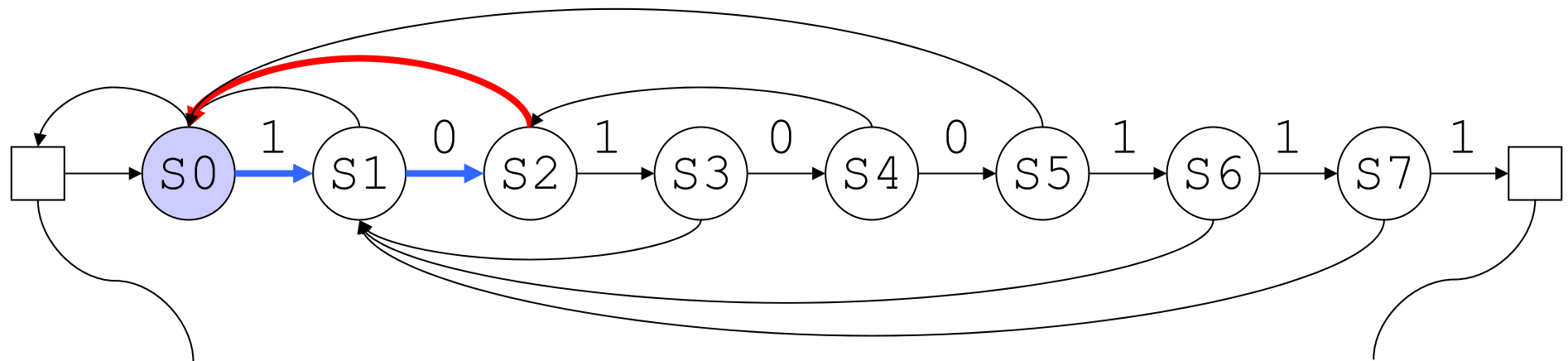
# Knuth – Morris – Pratt

生物情報 情報システム概論 森下

100111010010100010100111

10100111

10100111



初期状態  
常に右側に遷移

停止状態  
パターンを検出

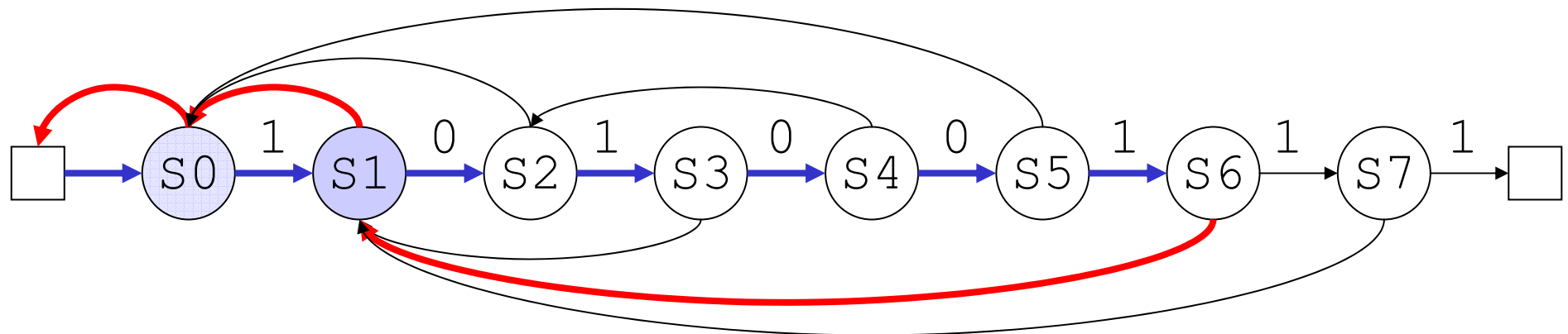
# Knuth – Morris – Pratt

100111010010100010100111

10100111

10100111

10100111

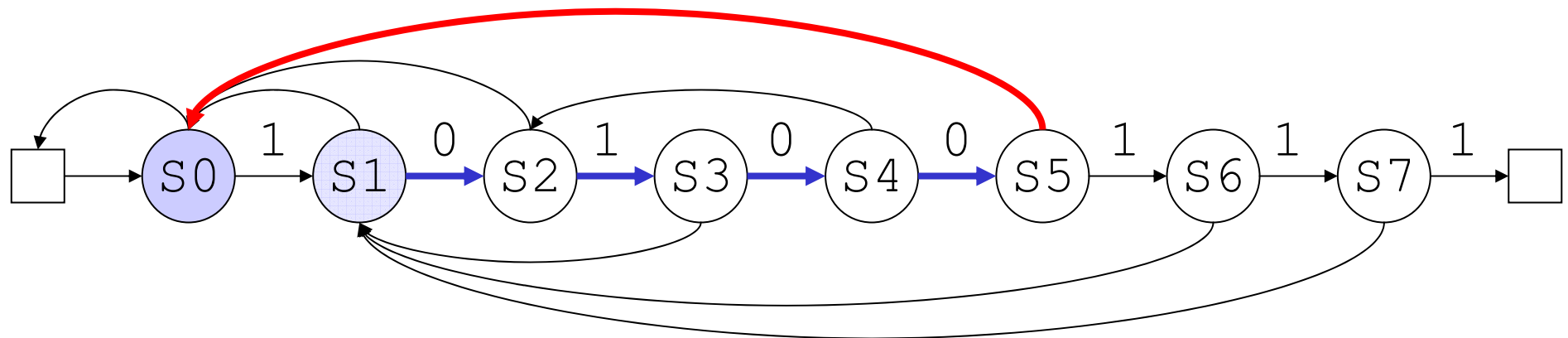


# Knuth – Morris – Pratt

100111010010100010100111

10100111

10100111

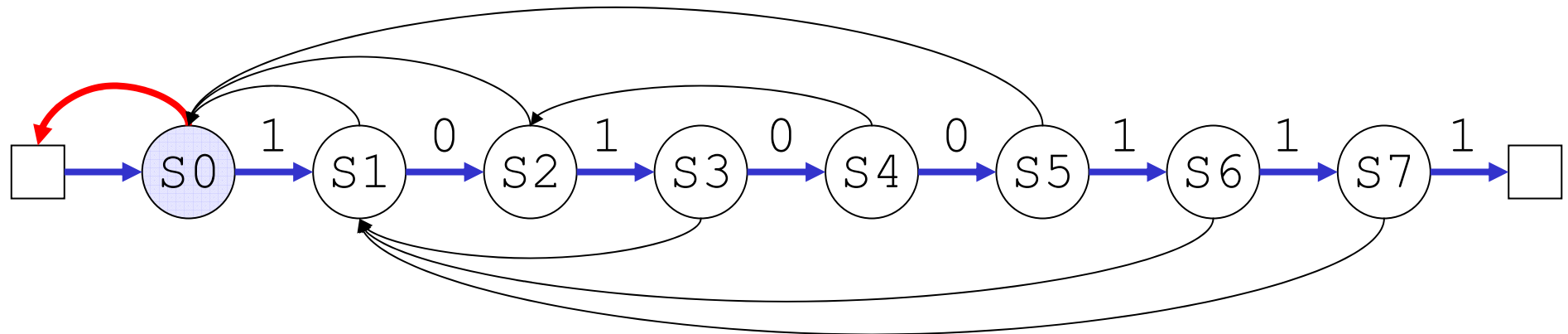


# Knuth – Morris – Pratt

100111010010100010100111

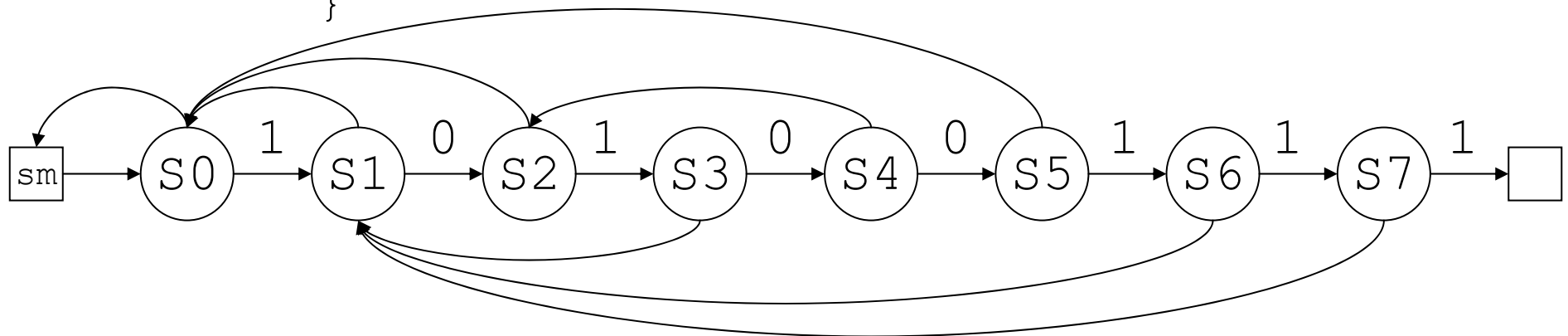
10100111

10100111



# Knuth – Morris – Pratt

```
int kmpsearch(char *a) {  
    int i=-1;  
sm:   i++;  
s0:   if(a[i] != '1') goto sm; i++;  
s1:   if(a[i] != '0') goto s0; i++;  
s2:   if(a[i] != '1') goto s0; i++;  
s3:   if(a[i] != '0') goto s1; i++;  
s4:   if(a[i] != '0') goto s2; i++;  
s5:   if(a[i] != '1') goto s0; i++;  
s6:   if(a[i] != '1') goto s1; i++;  
s7:   if(a[i] != '1') goto s1; i++;  
    return i-8;  
}
```



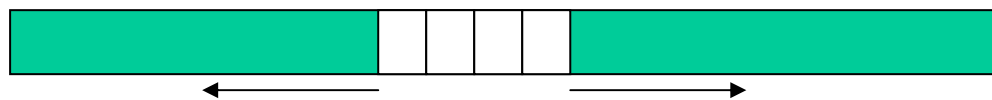
# BLAST

## 記号列

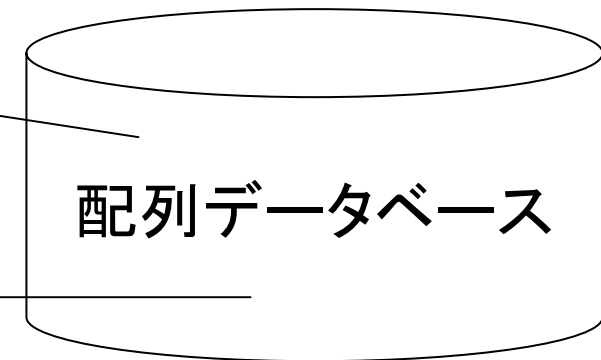


長さ  $l$  の部分文字列の集合  $W$  を生成

$W$  の元を部分にもつ配列を検索  
(Knuth-Morris-Pratt, Aho-Corasick)



マッチした部分を伸ばす





## Kunth-Morris-Pratt の逆走査版

0

最後から 1 文字目が不一致  
ならばパターン全体を 1 ずらす

10110101

10110101

$\text{next}[1]=1$

11

最後から 2 文字目が初めて不一致  
ならばパターン全体を 4 ずらす

10110101

10110101

$\text{next}[2]=4$

001

最後から 3 文字目が初めて不一致  
ならばパターン全体を 7 ずらす

10110101

10110101

$\text{next}[3]=7$

# Boyer – Moore

1101

10110101

10110101

next[4]=2

00101

10110101

10110101 next[5]=5

10110101

10110101 next[6]=5

10110101

10110101 next[7]=5

# Boyer – Moore

逆走査版と不一致文字法 (パターンに出現しない文字の場合は大幅にスキップ) を組合せる

文字列 a

A STRING SEARCHING EXAMPLE CONSISTING OF

STRING

STRING

R は p に  
現れない  
p を 5 移動

STRING

STRING

T は p に  
現れる  
p を 3 移動

S は p に  
現れる  
p を 4 移動

STRING

STRING

STRING

STRING

パターン p

[illegible]

# Boyer – Moore

不一致文字法は  $0, 1$  の2進列に不向き

解決案

$b$ 個のビットでブロックとしてまとめ、ブロック化文字として扱う

長さ  $M$  のパターン  $p$  に  $(M-b+1)$  個のブロック化文字

長大なテキスト  $a$  に  $2^b$  個のブロック化文字があるとき

$$M-b+1 \ll 2^b$$

であれば不一致文字が増えて都合がよい

# Rabin – Karp

a

p

長さ  $M$  のパターン  $p$  にたいして  
テキスト  $a$  のすべての  $M$  文字列  $x$  が  
 $p$  と一致するかをハッシュ関数  $h$  で判定

$$h(p) = h(x) ?$$

$$h(k) = k \bmod q \quad (q \text{ は大きな素数})$$

1つのパターン  $p$  をキーとして探索できればよい  
ハッシュ表を保持しなくてよいメリット

ハッシュ関数の計算が重たい？

# Rabin – Karp

文字は  $d$  種類、各文字  $a[i]$  は  $0 \leq a[i] < d$  をみたす整数

$$x = a[i]d^{M-1} + a[i+1]d^{M-2} + \dots + a[i+M-1]$$

$$a[i], a[i+1], \dots, a[i+M-1], a[i+M]$$

$$(x - a[i]d^{M-1})d + a[i+M]$$

- 差分だけを計算
- $\text{mod } q$  は算術演算より先に実行して  
大きな数の計算を未然に防ぐ

# Rabin – Karp

```
public static int rkSearch(String a, String p){
    int N = a.length();
    int M = p.length();
    if(M > N){ return -1;}
    int q = 33554393;        // 大きな素数 4バイトで表現可能
    int d = 32;              // 文字の種類は高々 D 個
    int dM = 1;              // d の M-1 乗 (M 乗でない) を dM に計算
    for(int i = 1; i < M; i++){ dM = (d * dM) % q; }
    int ha = 0;              // a のハッシュ値
    int hp = 0;              // p のハッシュ値
    for(int i = 0; i < M; i++){
        ha = ( ha * d + ((int)a.charAt(i) % d) ) % q;
        hp = ( hp * d + ((int)p.charAt(i) % d) ) % q;
    }
    int i;
    for(i = 0; hp != ha && i+M < N; i++){
        int tmp = (ha - ((int)a.charAt(i) % d) * dM + d * q) % q;
        // d * q は (...) が正になるように余分に追加
        ha = (tmp * d + ((int)a.charAt(i+M) % d)) % q;
    }
    if(N <= i+M){ return -1;}    // マッチしなかった
    else{ return i; }
}
```



# Rabin – Karp

異なる文字列のハッシュ値が一致する可能性あり

$h(k) = k \bmod q$  の素数  $q$  の値を非常に大きくすれば  
この可能性を限りなく 0 にできる

Rabin-Karp の方法は  $M+N$  にほぼ比例する時間で動作

異なる文字列のハッシュ値が一致する可能性が高い場合、

最悪  $O(MN)$  の計算時間

# 文字列探索 (不完全マッチ／ アラインメント)

# 不完全なマッチ

- ゲノム、遺伝子配列には読み間違いがある
- Genome Shotgun Assembly では read を繋げてゆく

生物情報科学実験2では、  
このソフトウェアを作成

- EST を収集したら類似性の高い配列をグループ化

# 配列アラインメント

## 配列アラインメント

AT-CGAT

| |   | |   |

ATGCG-T

全域的

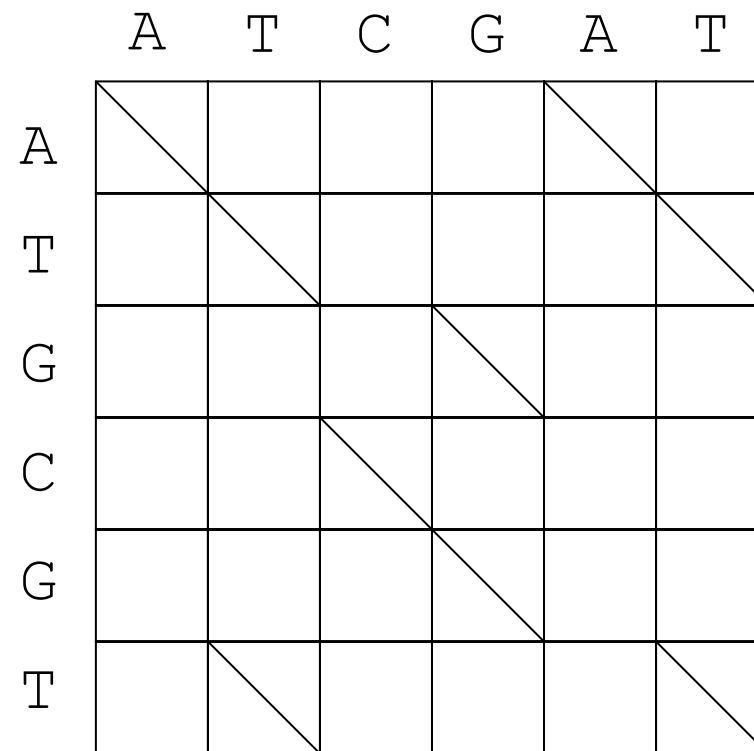
ATGCGATTAG

| |   | |

CG-TT

局所的

# 全域的アラインメント

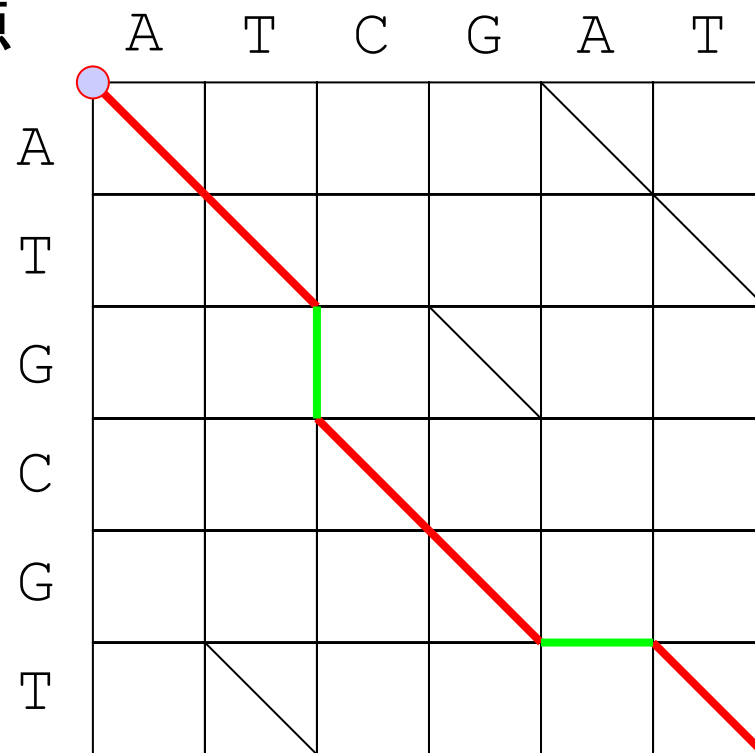


Edit graph

# 全域的アラインメント

生物情報 情報システム概論 森下

出発点



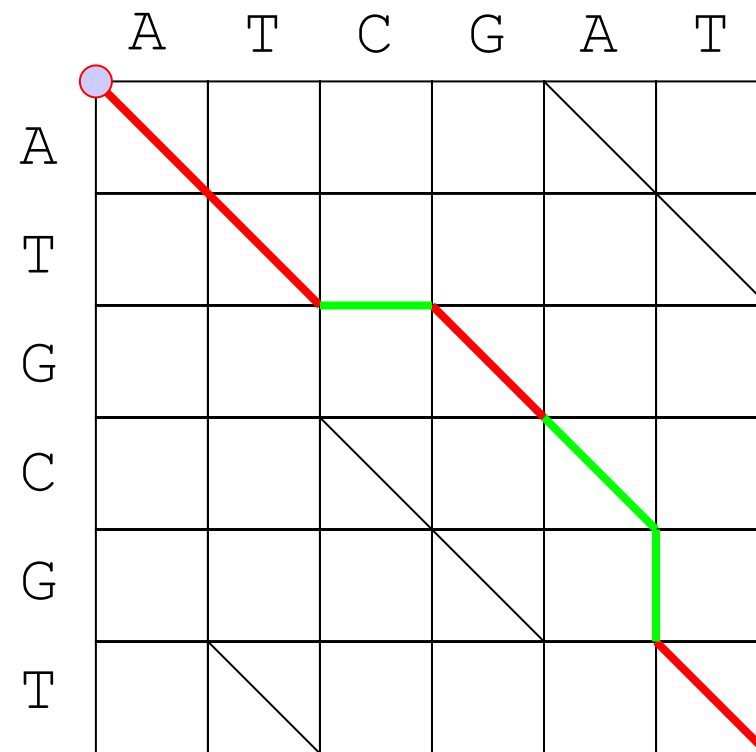
AT-CGAT

|| || |

ATGCG-T

# 全域的アラインメント

生物情報 情報システム概論 森下



ATCGA-T

|| | |

AT-GCGT

# 全域的アラインメント

|                |         | $\xrightarrow{j}$ |       |       |       |       |       |
|----------------|---------|-------------------|-------|-------|-------|-------|-------|
|                |         | $y_1$             | $y_2$ | $y_3$ | $y_4$ | $y_5$ | $y_6$ |
|                |         | A                 | T     | C     | G     | A     | T     |
| $\downarrow i$ | $x_1$ A | 1                 | 0.5   | 0     | -0.5  | -1    | -1.5  |
|                | $x_2$ T | 0.5               | 2     | 1.5   | 1     | 0.5   | 0     |
|                | $x_3$ G | 0                 | 1.5   | 1.5   | 2.5   | 2     | 1.5   |
|                | $x_4$ C | -0.5              | 1     | 2.5   | 2     | 2     | 1.5   |
|                | $x_5$ G | -1                | 0.5   | 2     | 3.5   | 3     | 2.5   |
|                | $x_6$ T | -1.5              | 0     | 1.5   | 3     | 3     | 4     |

$i = 0$  or  $j = 0$ :  $Score(i, j) = -\infty$

ただし例外として  $Score(0, 0) = 0$

$i > 0$  and  $j > 0$ :

$$Score(i, j) = \max \begin{cases} Score(i-1, j-1) + match(x_i, y_j) \\ Score(i-1, j) - gap \\ Score(i, j-1) - gap \end{cases}$$

$gap = 0.5$

$$match(x, y) = \begin{cases} 1 & x = y \\ -0.5 & x \neq y \end{cases}$$

|     |     |
|-----|-----|
| 1   | 0.5 |
| 0.5 |     |



# 全域的アラインメントの復元

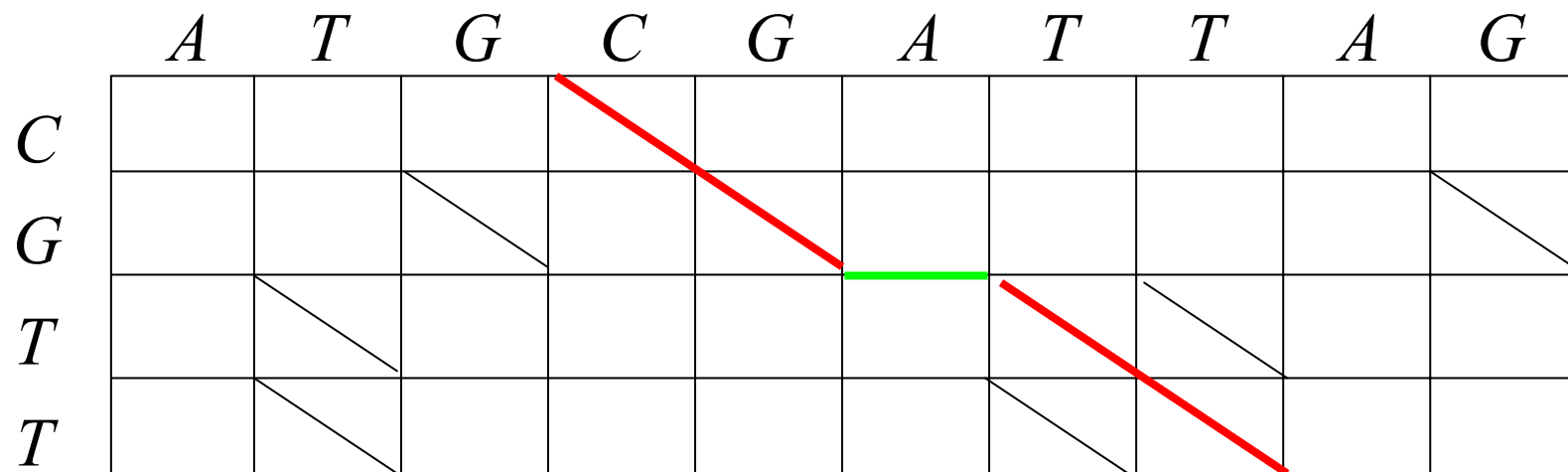
|   | A    | T   | C   | G    | A   | T    |
|---|------|-----|-----|------|-----|------|
| A | 1    | 0.5 | 0   | -0.5 | -1  | -1.5 |
| T | 0.5  | 2   | 1.5 | 1    | 0.5 | 0    |
| G | 0    | 1.5 | 1.5 | 2.5  | 2   | 1.5  |
| C | -0.5 | 1   | 2.5 | 2    | 2   | 1.5  |
| G | -1   | 0.5 | 2   | 3.5  | 3   | 2.5  |
| T | -1.5 | 0   | 1.5 | 3    | 3   | 4    |

AT-CGAT

|| || |

ATGCG-T

# 局所的アラインメント



ATGCGATTAG

|||

CG-TT

# 局所的アラインメント

|          | <i>A</i> | <i>T</i> | <i>G</i> | <i>C</i> | <i>G</i> | <i>A</i> | <i>T</i> | <i>T</i> | <i>A</i> | <i>G</i> |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| <i>C</i> |          |          |          |          |          |          |          |          |          |          |
| <i>G</i> |          |          |          |          |          |          |          |          |          |          |
| <i>T</i> |          |          |          |          |          |          |          |          |          |          |
| <i>T</i> |          |          |          |          |          |          |          |          |          |          |

ATGCGATTAG

|||

CGTT

## 局所的アラインメント

|       |     | $y_1$ | $y_2$ | $y_3$ | $y_4$ | $y_5$ | $y_6$ | $y_7$ | $y_8$ | $y_9$ | $y_{10}$ |
|-------|-----|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|
|       |     | $A$   | $T$   | $G$   | $C$   | $G$   | $A$   | $T$   | $T$   | $A$   | $G$      |
| $x_1$ | $C$ | 0     | 0     | 0     | 1     | 0.5   | 0     | 0     | 0     | 0     | 0        |
| $x_2$ | $G$ | 0     | 0     | 1     | 0.5   | 2     | 1.5   | 1     | 0.5   | 0     | 1        |
| $x_3$ | $T$ | 0     | 1     | 0.5   | 0.5   | 1.5   | 1.5   | 2.5   | 2     | 1.5   | 1        |
| $x_4$ | $T$ | 0     | 1     | 0.5   | 0     | 1     | 1     | 2.5   | 3.5   | 2.5   | 2        |

$i = 0$  or  $j = 0$ :  $Score(i, j) = 0$

$i > 0$  and  $j > 0$ :

$$Score(i, j) = \max \begin{cases} 0 \\ Score(i-1, j-1) + match(x_i, y_j) \\ Score(i-1, j) - gap \\ Score(i, j-1) - gap \end{cases}$$

[Smith, Waterman 1981]

$gap = 0.5$

$$match(x, y) = \begin{cases} 1 & x = y \\ -0.5 & x \neq y \end{cases}$$

# 局所的アラインメントの復元

|          | <i>A</i> | <i>T</i> | <i>G</i> | <i>C</i> | <i>G</i> | <i>A</i> | <i>T</i> | <i>T</i> | <i>A</i> | <i>G</i> |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| <i>C</i> | 0        | 0        | 0        | 1        | 0.5      | 0        | 0        | 0        | 0        | 0        |
| <i>G</i> | 0        | 0        | 1        | 0.5      | 2        | 1.5      | 1        | 0.5      | 0        | 1        |
| <i>T</i> | 0        | 1        | 0.5      | 0.5      | 1.5      | 1.5      | 2.5      | 2        | 1.5      | 1        |
| <i>T</i> | 0        | 1        | 0.5      | 0        | 1        | 1        | 2.5      | 3.5      | 2.5      | 2        |

ATGCGATTAG

|| ||

CG-TT

# 計算量

# 計算量

↑ 決定不能問題 (ゲーデル、チューリング 1930年代)  
プログラムの停止性の判定 定理の自動証明

$$\Omega\left(2^{2^{\cdots 2^N}}\right)$$

$O(2^N)$  NP困難問題 NP完全問題 (クック、カープ 1972年)  
最適なマルチプルアラインメント (近似解CLUSTAL W)  
最適なクラスタリング 最適なクラス分類(決定木など)  
近似解法 探索法が試みられている

..... ??

$$O(N^k)$$

$O(N^2)$  2つの $N$ 塩基(残基)配列のアラインメント (Smith-Waterman)  
最短経路

$O(N \lg N)$   $N$ 個の数のソーティング (merge sort, heap sort)

$O(N)$  文字列の探索、 $N$  個の数の最大値

# プロテイン・チップ 実験2



# プロテインチップの生データの補正

生物情報 情報システム概論 森下

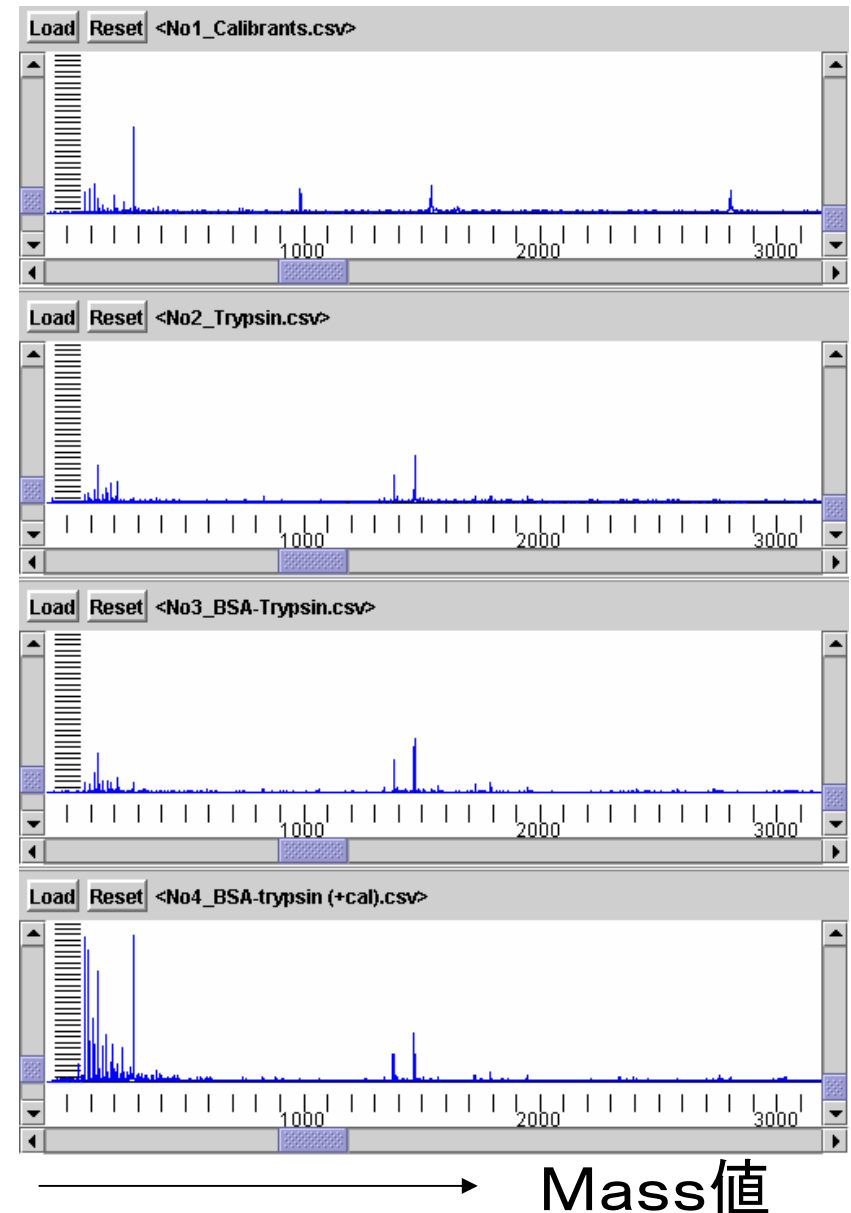
Intensity値 ↑

ピーク

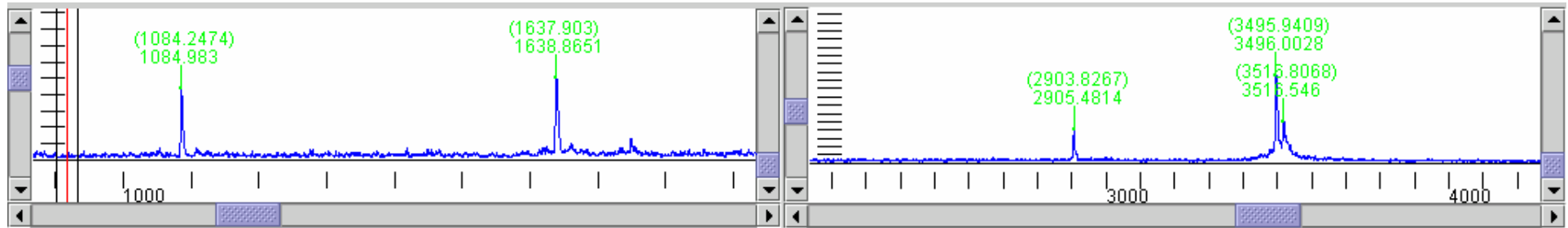
1. Calibration(複数タンパク質)基準ピーク
2. トリプシン(消化剤) ノイズ  
自己消化があるので複数のピークが誤差として出現
3. タンパク質X + トリプシン
4. タンパク質X + トリプシン + Calibration + Cal. の消化ピーク(ノイズ)

手続き

- ・ 1のCalピークから補正関数  $F$  を計算
- ・  $F$  を使って1から3を補正
- ・ 4のCalピークから補正関数  $F'$  を計算  
 $F$  と  $F'$  は違うことに注意
- ・  $F'$  を4に適用
- ・ 差分  $4 \cap 3 \cap \neg 2 = \text{タンパク質X}$



# 補正関数の設計



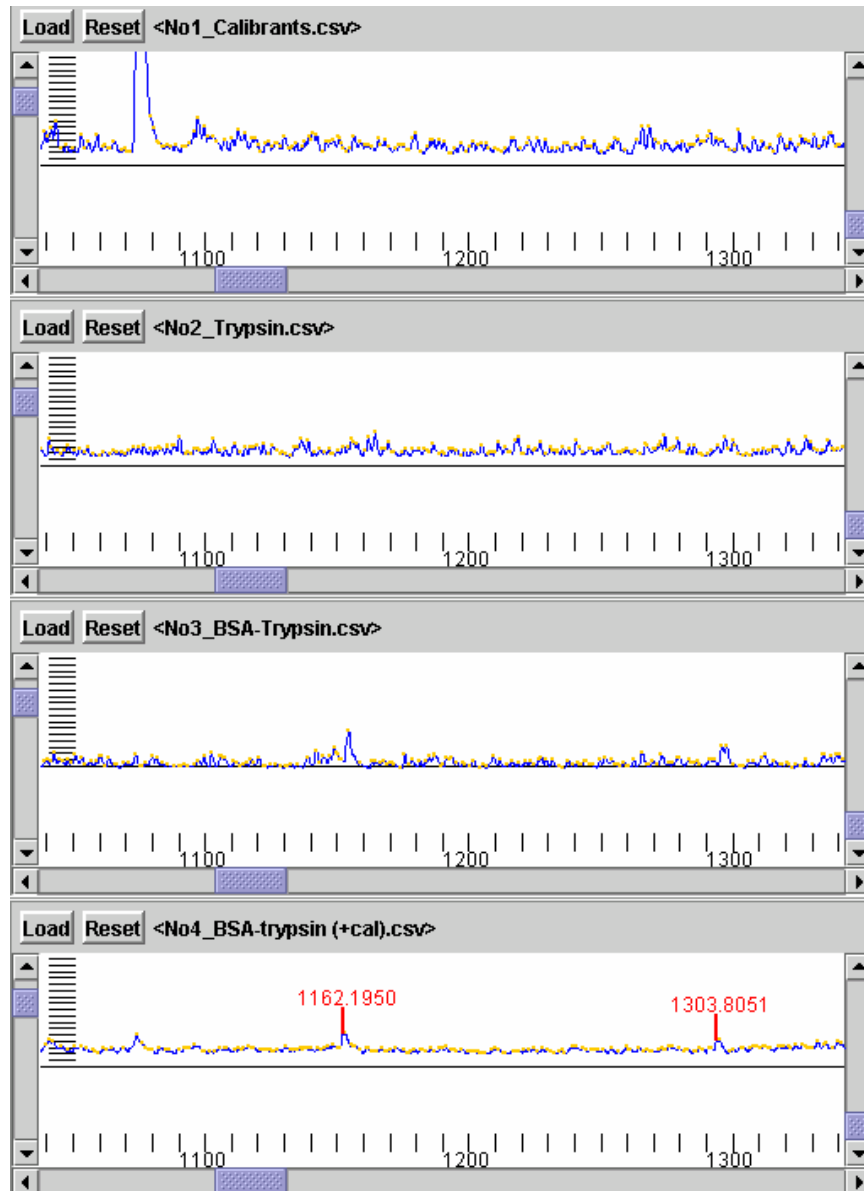
|                  | $i$ | 観測値 $A_i$ | 真値 $B_i$  |
|------------------|-----|-----------|-----------|
| Arg8-Vasopressin | 1   | 1084.983  | 1084.2474 |
| Somatostatin     | 2   | 1638.8651 | 1637.903  |
| Human Insulin    | 3   | 2905.4814 | 2903.8267 |
| Bovine Insulin   | 4   | 3496.0028 | 3495.9409 |
| Hirudin BHVK     | 5   | 3516.546  | 3516.8068 |

$x = \text{観測値}, y = \text{補正值}$

補正関数 
$$y = B_i + \frac{B_{i+1} - B_i}{A_{i+1} - A_i} (x - A_i) \quad (A_i \leq x \leq A_{i+1})$$

# 補正後のピークを選択とタンパク質Xの推定

生物情報 情報システム概論 森下



ピークを選択

タンパク質 X の推定

# タンパク質Xの推定 Profound の出力

SB882A6FF-0860-1C2F768C - Netscape

http://129.85.19.192/profound/results/SB882A6FF-0860-1C2F768C.html

Reprints of our posters at the 48th ASMS Conference are available in PDF format (ThPB096, ThPB097).

## ProFound - Search Result Summary

Version 4.10.5  
The Rockefeller University Edition

Protein Candidates for search B882A6FF-0860-1C2F768C [225935 sequences searched]

| Rank | Probability | Est'd Z | Protein Information and Sequence Analyse Tools (T)   | %  | pI  | kDa    | @ |
|------|-------------|---------|--|----|-----|--------|---|
| +1   | 1.0e+000    | 1.32    | <a href="#">gi 418694 ref AB005</a> serum albumin precursor [validated] - bovine   | 22 | 5.8 | 71.74  | @ |
| +2   | 4.1e-004    | -       | <a href="#">gi 14786208 ref XP_030024.1</a> (XM_030024) similar to protein kinase C binding protein 1 (H. sapiens) [Homo sapiens]                  | 17 | 8.8 | 105.77 | @ |
| 3    | 2.4e-004    | -       | <a href="#">gi 1345943 sp P49457 DAF_PONPY</a> Complement decay-accelerating factor (CD55)   | 23 | 9.3 | 38.30  | @ |
| 4    | 2.4e-004    | -       | <a href="#">gi 2494652 sp Q29492 G6PD_MACRO</a> Glucose-6-phosphate 1-dehydrogenase (G6PD)   | 23 | 6.2 | 59.87  | @ |
| 5    | 1.7e-004    | -       | <a href="#">gi 1470066 eb AAC52678.1</a> nebulin [Mus musculus]  | 12 | 9.3 | 158.84 | @ |
| +6   | 1.0e-004    | -       | <a href="#">gi 15079366 eb AAH11529.1 AAH11529</a> (BC011529) Unknown (protein for MGC-17365) [Homo sapiens]                                       | 11 | 9.8 | 116.80 | @ |
| 7    | 1.0e-004    | -       | <a href="#">gi 130201 sp P15083 PIGR_RAT</a> Polymeric-immunoglobulin receptor precursor (Poly-Ig receptor) (PIGR) [Contains: Secretory component] | 12 | 5.1 | 86.42  | @ |
| 8    | 8.5e-005    | -       | <a href="#">gi 20826641 ref XP_145533.1</a> similar to zinc finger protein 91 (HPF7, HTF10) [Homo sapiens] [Mus musculus]                          | 17 | 9.8 | 111.52 | @ |
| +9   | 5.4e-005    | -       | <a href="#">gi 186396 eb AAA59163.1</a> (M94131) mucin [Homo sapiens]  | 6  | 5.5 | 141.68 | @ |
| 10   | 2.8e-005    | -       | <a href="#">gi 20849462 ref XP_136730.1</a> similar to dysferlin [Mus musculus]  | 7  | 8.2 | 237.27 | @ |

NOTE:  
1. To search again using unmatched masses, click the symbol @.  
2. Highly similar protein sequences were given the same rank (IE user: click "+" to expand/contract).

### Input Summary

|                                   |  |
|-----------------------------------|--|
| Date & Time                       | Tue Jul 22 14:56:39 2003 UTC (Search Time: 6.95 sec.)  |
| Sample ID                         |  |
| Database                          | NCBI nr (2002/11/27)   |
| Taxonomy Category                 | Mammalia (mammals)   |
| Protein Mass Range                | 0 - 3000 kDa   |
| Protein pI Range                  | 0.0 - 14.0   |
| Search for                        | Single protein only  |
| Digest Chemistry                  | Trypsin  |
| Max Missed Cut                    | 2  |
| Modifications                     | +C3H5ON@C (Complete);  |
| Charge State                      | M  |
| Peptide Masses (Da, Average)      | 927.361 1163.476 1305.482 1440.081 1480.213 1492.157 1568.158 1606.445 1824.724 1889.905 1964.135 2045.968 2315.622 2437.224 2677.524 3137.067 3729.231 3830.625 |
| Tolerance (AVG)                   | 1.00 Da  |
| Peptide Masses (Da, Monoisotopic) |  |
| Tolerance (MON)                   | 0.10 Da  |
| Number of Peptides                | 18   |

This ProFound is made available to public through the courtesy of [ProteoMetrics](#) [search + transmission time: >=7.22 sec]

ドキュメント完了 (9.644 秒)

DB882A6FF-0860-1C2F768C - Netscape

http://129.85.19.192/profound/results/DB882A6FF-0860-1C2F768C.html

## ProFound - Search Result Details

Version 4.10.5  
The Rockefeller University Edition

### Details for rank 1 candidate in search B882A6FF-0860-1C2F768C

[gi|418694|ref|AB005](#) serum albumin precursor [validated] - bovine

Sample ID : [Pass:0]  
Measured peptides : 18  
Matched peptides : 11  
Min. sequence coverage: 22%

COVERAGE MAP AND ERROR MAP

Measured Avg/ Computed Error Residues Missed  
Mass(M) Mono Mass (Da) Start To Cut Peptide sequence

|          |   |          |        |     |     |   |                                  |
|----------|---|----------|--------|-----|-----|---|----------------------------------|
| 927.361  | A | 927.068  | 0.293  | 161 | 167 | 0 | YLYEIAIR                         |
| 1163.476 | A | 1163.336 | 0.140  | 66  | 75  | 0 | LYNELTEFAK                       |
| 1305.481 | A | 1305.497 | -0.015 | 402 | 412 | 0 | HLVDEPQNLIK                      |
| 1440.081 | A | 1439.681 | 0.400  | 360 | 371 | 1 | RHPEYAVSVLLR                     |
| 1480.213 | A | 1479.699 | 0.514  | 421 | 433 | 0 | LGEYGFQNALIVR                    |
| 1492.157 | A | 1491.623 | 0.534  | 76  | 88  | 0 | TCVADESHAGCEK                    |
| 1568.157 | A | 1567.718 | 0.440  | 347 | 359 | 0 | DAFLGSFLYEYSR                    |
| 1568.157 | A | 1568.684 | -0.526 | 387 | 399 | 0 | DDPHACYSTVFQK                    |
| 1889.905 | A | 1889.140 | 0.765  | 169 | 183 | 0 | HPYFYAPELLYYANK                  |
| 2045.968 | A | 2045.328 | 0.641  | 168 | 183 | 1 | RHPYFYAPELLYYANK                 |
| 3729.231 | A | 3729.271 | -0.040 | 35  | 65  | 2 | FKDLGEEQFKGLVLIATSGYLQQCPFDEHVK  |
| 3729.231 | A | 3730.106 | -0.875 | 267 | 297 | 2 | ECCHGDLLECAADDRADLAKYICDNGDTISSK |
| 3830.625 | A | 3831.324 | -0.699 | 402 | 433 | 2 | HLVDEPQNLIKQNCQGFELKGEYGFQNALIVR |

### Unmatched Average Masses:

1606.445 1824.724 1964.135 2315.622 2437.224 2677.524 3137.067

### Search again using unmatched masses:

Search for single protein only in All taxa

ドキュメント完了 (2.334 秒)

# 実験2(プロテインチップ)の目標

生物情報 情報システム概論 森下

- ピークからタンパク質 X を推定する  
Profound のようなソフトウェアを作成
- タンパク質データベースから分子量を  
たよりにして類似したペプチド配列を検索
- どのようにすればソフトウェアを使って  
観測の精度を上げることができるかも  
考えてみる