データ構造とアルゴリズム
（第10回）

モバイルコンピューティング研究室
柴田史久

1

本日の講義内容

- 整列（３）
  - マージソートの原理
  - 配列によるマージソート
  - 連結リストによるマージソート
  - 外部整列（概要）

2

教科書 第15章（pp.332～355）

マージソート

3

マージソート(Merge Sort)

- 計算量 $O(n \log n)$ の高速な整列アルゴリズム
- 定数係数が大きいのでクイックソートには負ける
- 特徴は要素をシーケンシャルにアクセスすること
- 連結リストや外部記憶上のデータの整列に利用

4

マージソートの原理

- マージ（merge）
  - 整列済みの2つのデータ列を1つにまとめる操作
  - 結果として得られる列も値の順序通りに並ぶ

5

マージの概略

詳細は教科書 p.333 List 15.1

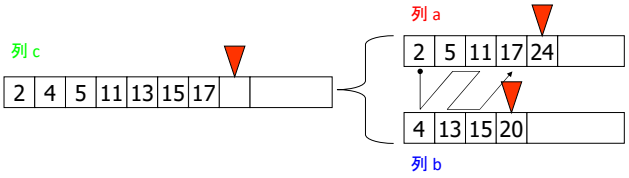
```
入力:マージ対象となる2つの列 a, 列 b
出力:列 a, 列 b の要素をマージして得られた列 c

static void merge()
{
  列 c を空にしておく
  while (列 a が空でない && 列 b が空でない) {
    if (列 a の先頭の要素 <= 列 b の先頭の要素) {
      列 a の先頭の要素を取り除いて，列 c の最後尾に追加する
    } else {
      列 b の先頭の要素を取り除いて，列 c の最後尾に追加する
    }
  }
  if (列 a が空である) {
    列 b に残っている要素を，そのままの順番で列 c の最後尾に追加する
  } else {
    列 a に残っている要素を，そのままの順番で列 c の最後尾に追加する
  }
}
```

6

マージソートの原理

- マージ（merge）
  - 整列済みの2つのデータ列を1つにまとめる操作
  - 結果として得られる列も値の順序通りに並ぶ



7

マージソート(1)

- 手順
  1. データ列を真ん中で2つの部分列 a と b に分割
  2. 部分列 a と b を，それぞれ（再帰的に）整列
  3. 整列済みになった部分列 a と b をマージ
- ステップ2での整列に再帰呼び出しを利用
- 再帰呼び出し毎に部分列の長さは半分に
- 部分列の長さが1なら再帰終了（自明なケース）
- 分割統治法

8

マージソート(2)

詳細は教科書 p.335 List 15.2

```
入力:列 a
出力:列 a を整列したもの

static void mergeSort()
{
  if (列 a はただ1つの要素からなる) {  // 再帰の終了条件
    return ;  // 当然ながら整列済みなので何もせずに戻る
  }

  列 a を2つの列 x, y に分割する
  mergeSort() を再帰的に呼び出して，列 x を整列する
  mergeSort() を再帰的に呼び出して，列 y を整列する
  列 x, y をマージして，その結果を列 a に戻す
}
```

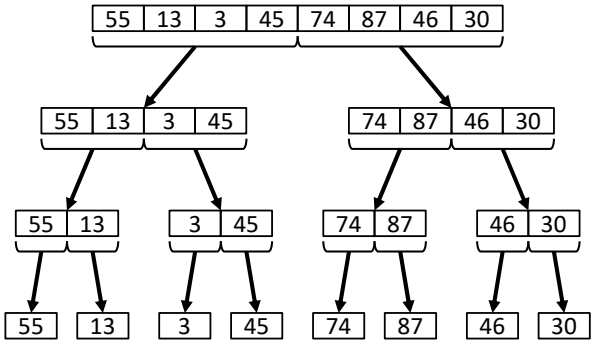
9

再帰処理の違い

- クイックソート
  - 仕事をする（分割処理）
  - 分割した部分に対して，再帰的に処理を行う
- マージソート
  - 分割した部分に対して，再帰的に処理を行う
  - 仕事をする（マージ処理）

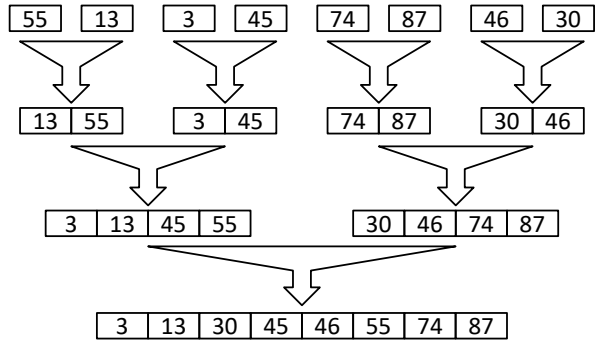
10

マージソートの過程(1) -分割-



11

マージソートの過程(2) -マージ-



12

マージソートの実装(1)

- 配列によるマージソート
- 教科書 List 15.3 は少しトリッキー
  - マージの処理をマージソートのメソッド内部で実装
  - 作業配列へのコピーがトリッキー
  - 詳しくは pp.338～342の説明を読むこと
- 以降では素直な実装例を紹介

13

マージソートの実装(1) マージの概略

List 15.1を素直に実装

練習課題12-1のヒント

```
入力:int[] arrayA, int[] arrayB
出力:int[] arrayC

static void merge(int[] arrayA, int[] arrayB, int[] arrayC) {
  int aIdx = 0, bIdx = 0, cIdx = 0 ;
  while (aIdx < arrayA.length && bIdx < arrayB.length) {
    if (arrayA[aIdx] < arrayB[bIdx]) {
      arrayC[cIdx++] = arrayA[aIdx++] ;
    } else {
      arrayC[cIdx++] = arrayB[bIdx++] ;
    }
  }
  while (aIdx < arrayA.length) {
    arrayC[cIdx++] = arrayA[aIdx++] ;
  }
  while (bIdx < arrayB.length) {
    arrayC[cIdx++] = arrayB[bIdx++] ;
  }
}
```

小さいほうからマージ

残りの部分を書き出す

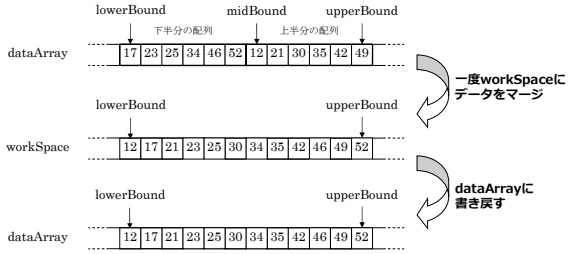
残りの部分を書き出す

if 文は不要．どちらかの while は無視される

14

マージソートの実装(2) mergeの実装

- ３つの配列を使うとメモリの確保などで効率が悪い
- 整列対象の配列と同じ長さの作業配列を用いて処理
  - 整列対象配列：dataArray
  - 作業配列：workSpace



15

マージソートの実装(3) mergeの実装

- merge()は１つの配列中の部分配列をマージ
- 引数
  - int[] dataArray：整列対象のデータ
  - int[] workSpace：作業配列（事前に領域確保）
  - int lowBound：下半分の配列の開始位置
  - int midBound：上半分の配列の開始位置
    （＝下半分の配列の上限位置＋１）
  - int upperBound：上半分の配列の上限位置
- マージ後， workSpace からものと配列にデータを戻す
- 使用する workSpace の添字の範囲に注意

16

マージソートの実装(4) 再帰呼び出し

List 15.2を素直に実装

```
private static void recMergeSort(int[] dataArray,
  int[] workSpace, int lowerBound, int upperBound) {
  if (lowerBound == upperBound) {  /* 再帰の終了条件 */
    return ;
  } else {
    int mid = (lowerBound + upperBound) / 2 ; // 中間点を決めて

    recMergeSort(dataArray, workSpace, lowerBound, mid) ;
                                        // 下半分をソート
    recMergeSort(dataArray, workSpace, mid+1, upperBound) ;
                                        // 上半分をソート
    merge(dataArray, workspace, lowerBound, mid+1, upperBound);
                                        // 両者をマージ
  }
}
```

作業用配列（workSpace）の領域はrecMergeSortでは確保しない

17

マージソートの実装(5) 整列の呼び出し

```
public static void mergeSort(int[] dataArray) {

  // 整列対象と同じ大きさの作業用配列を確保
  int[] workSpace = new int[dataArray.length];

  // 配列全体を対象として再帰処理に入る
  recMergeSort(dataArray, workSpace, 0, dataArray.length - 1);
}
```

作業用配列（workSpace）の領域を確保し，引数として渡していく

18

マージソートの性質(1)

- 分割の手間
  - 配列を利用する場合（p は部分列長）：$O(1) \times n / p$
  - 連結リストを利用する場合：$O(n)$
    - 要素数を数えながら分割場所を探すため
- マージの手間：$O(n \log n)$
  - n個の要素のマージに $O(n)$
  - マージ回数は $\log_2 n$
- マージソートの計算量：$O(n \log n)$
- 分割は要素の値に非依存なので計算量は常に同じ
  - マージソートのメリット

マージソートの性質(2)

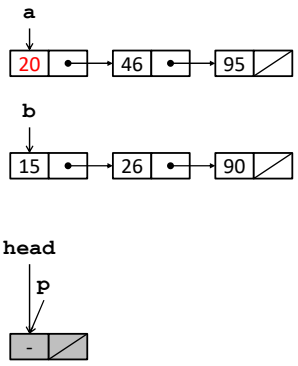
- 配列の整列にベストなのはクイックソート
- 定数項部分がクイックソートのほうがよい
- マージソートは配列のコピーが手間
- 整列に必要な作業領域が $O(n)$ なのも不利な点
  - クイックソートは $O(\log n)$
- マージの際に位置関係を保てば安定な整列が可能

連結リストによるマージ

- 実はマージソートは配列が不得意
  - 作業用配列へのコピーが高コスト
  - 作業用配列として元の配列と同容量のメモリが必要
- 連結リストを使うと上記の問題が解決可能
  - リンクの書き換えで要素の移動が可能
  - リンクの容量は $O(n)$ だが，本来必要なもの
  - マージ操作は列へのシーケンシャルアクセスで，単方向の連結リストで実現可能

連結リストのマージの流れ(1)

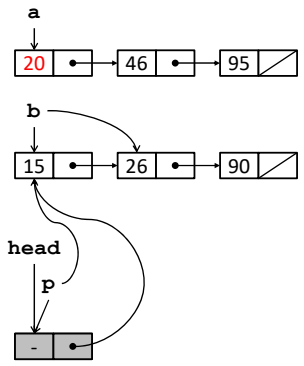
詳細は教科書 pp.346～348 List 15.5




```
private static Cell mergeList
  (Cell a, Cell b) {
  Cell head = new Cell(0);
  Cell p = head;
  while (a != null && b != null) {
    if (a.data <= b.data) {
      p.next = a;
      p = a;
      a = a.next;
    } else {
      p.next = b;
      p = b;
      b = b.next;
    }
  }
  if (a == null) {
    p.next = b;
  } else {
    p.next = a;
  }
  return head.next;
}
```


連結リストのマージの流れ(2)

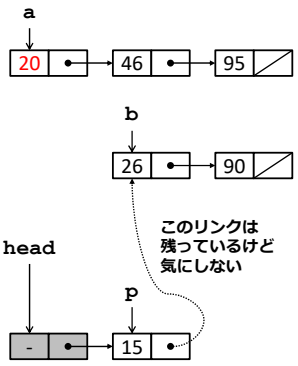
詳細は教科書 pp.346～348 List 15.5




```
private static Cell mergeList
  (Cell a, Cell b) {
  Cell head = new Cell(0);
  Cell p = head;
  while (a != null && b != null) {
    if (a.data <= b.data) {
      p.next = a;
      p = a;
      a = a.next;
    } else {
      p.next = b;
      p = b;
      b = b.next;
    }
  }
  if (a == null) {
    p.next = b;
  } else {
    p.next = a;
  }
  return head.next;
}
```


連結リストのマージの流れ(3)

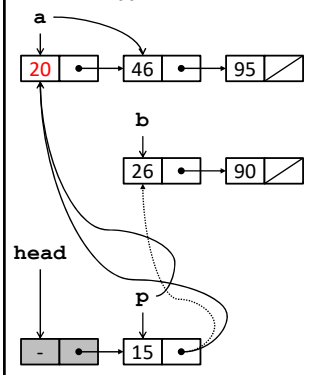
詳細は教科書 pp.346～348 List 15.5




```
private static Cell mergeList
  (Cell a, Cell b) {
  Cell head = new Cell(0);
  Cell p = head;
  while (a != null && b != null) {
    if (a.data <= b.data) {
      p.next = a;
      p = a;
      a = a.next;
    } else {
      p.next = b;
      p = b;
      b = b.next;
    }
  }
  if (a == null) {
    p.next = b;
  } else {
    p.next = a;
  }
  return head.next;
}
```

連結リストのマージの流れ(4)

詳細は教科書 pp.346～348 List 15.5

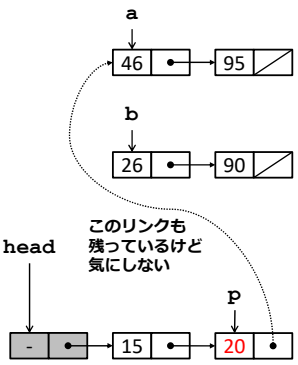



```
private static Cell mergeList
  (Cell a, Cell b) {
  Cell head = new Cell(0);
  Cell p = head;
  while (a != null && b != null) {
    if (a.data <= b.data) {
      p.next = a;
      p = a;
      a = a.next;
    } else {
      p.next = b;
      p = b;
      b = b.next;
    }
  }
  if (a == null) {
    p.next = b;
  } else {
    p.next = a;
  }
  return head.next;
}
```

25

連結リストのマージの流れ(5)

詳細は教科書 pp.346～348 List 15.5

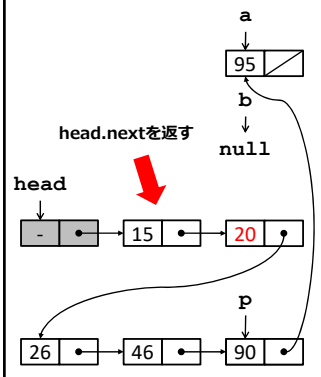



```
private static Cell mergeList
  (Cell a, Cell b) {
  Cell head = new Cell(0);
  Cell p = head;
  while (a != null && b != null) {
    if (a.data <= b.data) {
      p.next = a;
      p = a;
      a = a.next;
    } else {
      p.next = b;
      p = b;
      b = b.next;
    }
  }
  if (a == null) {
    p.next = b;
  } else {
    p.next = a;
  }
  return head.next;
}
```

26

連結リストのマージの流れ(6)

詳細は教科書 pp.346～348 List 15.5



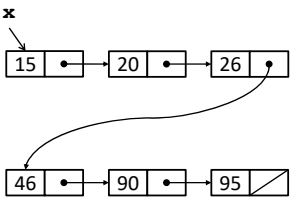

```
private static Cell mergeList
  (Cell a, Cell b) {
  Cell head = new Cell(0);
  Cell p = head;
  while (a != null && b != null) {
    if (a.data <= b.data) {
      p.next = a;
      p = a;
      a = a.next;
    } else {
      p.next = b;
      p = b;
      b = b.next;
    }
  }
  if (a == null) {
    p.next = b;
  } else {
    p.next = a;
  }
  return head.next;
}
```

27

連結リストのマージソート(1)

詳細は教科書 pp.346～348 List 15.5

連結リストの要素が 0 か 1 つなら
そのまま戻る（再帰の終了条件）

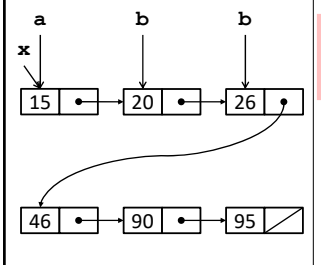



```
public static Cell mergeSortList
  (Cell x) {
  if (x == null || x.next == null) {
    return x;
  }
  Cell a = x;
  Cell b = x.next;
  if (b != null) {
    b = b.next;
  }
  while (b != null) {
    a = a.next;
    b = b.next;
    if (b != null) {
      b = b.next;
    }
  }
  Cell p = a.next;
  a.next = null;
  return mergeList(mergeSortList(x),
                   mergeSortList(p));
}
```

28

連結リストのマージソート(2)

詳細は教科書 pp.346～348 List 15.5

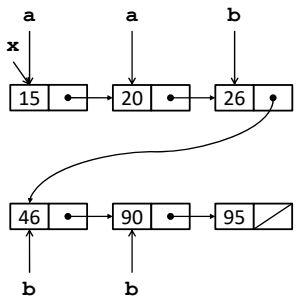



```
public static Cell mergeSortList
  (Cell x) {
  if (x == null || x.next == null) {
    return x;
  }
  Cell a = x;
  Cell b = x.next;
  if (b != null) {
    b = b.next;
  }
  while (b != null) {
    a = a.next;
    b = b.next;
    if (b != null) {
      b = b.next;
    }
  }
  Cell p = a.next;
  a.next = null;
  return mergeList(mergeSortList(x),
                   mergeSortList(p));
}
```

29

連結リストのマージソート(3)

詳細は教科書 pp.346～348 List 15.5

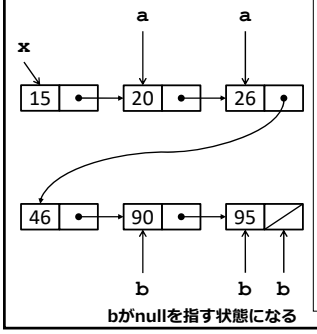



```
public static Cell mergeSortList
  (Cell x) {
  if (x == null || x.next == null) {
    return x;
  }
  Cell a = x;
  Cell b = x.next;
  if (b != null) {
    b = b.next;
  }
  while (b != null) {
    a = a.next;
    b = b.next;
    if (b != null) {
      b = b.next;
    }
  }
  Cell p = a.next;
  a.next = null;
  return mergeList(mergeSortList(x),
                   mergeSortList(p));
}
```

30

連結リストのマージソート(4)

詳細は教科書 pp.346～348 List 15.5



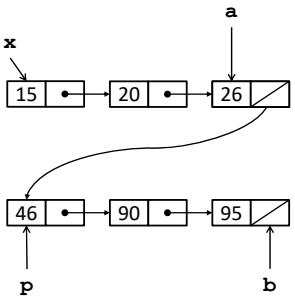
bがnullを指す状態になる

```
public static Cell mergeSortList
  (Cell x) {
  if (x == null || x.next == null) {
    return x;
  }
  Cell a = x;
  Cell b = x.next;
  if (b != null) {
    b = b.next;
  }
  while (b != null) {
    a = a.next;
    b = b.next;
    if (b != null) {
      b = b.next;
    }
  }
  Cell p = a.next;
  a.next = null;
  return mergeList(mergeSortList(x),
                   mergeSortList(p));
}
```

31

連結リストのマージソート(5)

詳細は教科書 pp.346～348 List 15.5




```
public static Cell mergeSortList
  (Cell x) {
  if (x == null || x.next == null) {
    return x;
  }
  Cell a = x;
  Cell b = x.next;
  if (b != null) {
    b = b.next;
  }
  while (b != null) {
    a = a.next;
    b = b.next;
    if (b != null) {
      b = b.next;
    }
  }
  Cell p = a.next;
  a.next = null;
  return mergeList(mergeSortList(x),
                   mergeSortList(p));
}
```

分割したリスト x, p に対して再帰呼び出し

32

外部整列

- 外部記憶上のデータを整列すること
- 注意事項
  - アクセスに時間がかかる
  - 入出力はバイト単位ではなくブロック単位
- 入出力の回数を減らすのがコツ
- アクセス方式の違い
  - ランダムアクセス（ディスク）
  - シーケンシャルアクセス（テープ）

33

マージソートを利用した外部整列

- 基本的な考え方
  - データを主記憶に入る分量だけ読み込む
  - 読み込んだデータを内部整列してからファイルに書き出す
    - 書き出した塊を連（run）と呼ぶ
  - マージアルゴリズムを使って連を整列された列にまとめる
- 詳細は教科書 pp.351～355 を読むこと

34

まとめ

- マージソートの原理
- 配列によるマージソート
- 連結リストによるマージソート
- 外部整列（概要）

35

参考文献

- 定本 Javaプログラマのための アルゴリズムとデータ構造（近藤嘉雪）
- 新・明解 Javaで学ぶ アルゴリズムとデータ構造（柴田望洋）
- 岩波講座ソフトウェア科学 3 アルゴリズムとデータ構造（石畑清）
- Javaで学ぶアルゴリズムとデータ構造 Robert Lafore（著）・岩谷 宏（翻訳）
- Java アルゴリズム+データ構造完全制覇 オングス (著)・杉山 貴章・後藤 大地 (監修)

36