データ構造とアルゴリズム実験レポート
課題２：<Java によるプログラミングの復讐>

<201811320><1 クラス><江頭輝>

締切日：<2019 年 4 月 22 日>
提出日：<2019 年 4 月 27 日>

1　必須課題 2-1

この課題では，セルに整数値を格納する連結リストを実現するクラス List を Java 言語で実装する。

1.1　List.java の作成

1.1.1　実装の方針

まず，`List` クラスを定義する。

1. コンストラクタ

セルの直後（ポインタは）‘p.next‘のような形で表現し、セルの値は ‘p.data‘のように表記する。そして、リストの最初を表すリストを head として定義した。

2. `insert_cell` メゾッド

リスト p の次に d を値として持つリストを挿入するので、まずデータが d のリストを作成する。その後、p の次にそのリストが来て、その次に p.next が来るような実装を行う。

3. `delete_cell` メゾッド

引数として指定したリスト p のポインタを表すコピーを作成して、p のポインタは p のポインタのポインタになるようにリスト関係を実装する。

4. `display`

まず head のコピーを作成する必要がある。

5. 実装コードおよびコードの説明

1.1.2　実装の方針

insert_cell と insert_cell_top では即席変数である new_cell を作成しそのデータに引数を渡すことで新たなセルを作成する。top の場合はのポインタにこれまでの `head` をよび、前者では引数として渡したのポインタがのポインタになるようにする。delete_cell ではのポインタに `p` のポインタのポインタを代入することでセルの削除を行うようにしている。delete_cell_top ではではなく texttthead になった。display ではのポインタが `null` になるまで値をループで出力する。

```java
// List.java
public class List {
    static List head;
    List next;
    int data;

    // 新たなリストを生成 p => new_cell => p.next
    static void insert_cell(List p, int d) {
        List new_cell = new List();
        new_cell.data = d;
        new_cell.next = p.next;
        p.next = new_cell;
    }

    // head = new_cellデータは(d) => old_head
    static void insert_cell_top(int d) {
        List new_cell = new List();
        new_cell.data = d;
        new_cell.next = head;
        head = new_cell;
    }

    // p.next を p.next.next で上書きしてp.を削除next
    static void delete_cell(List p) {
        List q = p.next;
        p.next = q.next;
    }

    // head を head.next で上書き
    static void delete_cell_top() {
        List q = head;
        head = q.next;
    }

    // head のコピーを.で終わりまでループしてそれぞれのデータを表示next
    static void display() {
        List tmp = head;
        while (tmp != null) {
            System.out.print(tmp.data);
            tmp = tmp.next;
        }
        // 改行
        System.out.println();
    }

    // List クラスの動作確認
    public static void main(String[] args) {
        insert_cell_top(1);
        insert_cell(head, 3);
        insert_cell(head, 2);
        display();

        delete_cell(head);
        display();

        delete_cell_top();
        display();
    }
}
```

図 1　List.java のソースコード

1.1.3　実行結果

まず，List.java を以下のコマンドでコンパイルする．

```
--------------------------------------
$ javac List.java
--------------------------------------
```

コンパイル後に以下のコマンドで実行を行った。

```
-------------------------------------
$ java List
123
13
3
-------------------------------------
```

main メゾッドでまず、1 を先頭に追加、先頭の直後に 3, 2 を順番に追加していった。なので、43 行目の display 関数で出力っされる値は 123 である。次に head セルの次のセルを削除したので 46 行目の display 関数では 13 が出力される。最後に delete_cell_top 関数を呼び出し、先頭のセルを削除したため、49 行目での display 関数では 3 のみが出力された。

1.1.4 考察

このコードだと、head の先頭にセルを追加する場合新しいリストを作る必要があります。また、head のみ仕様が異なるので、余分に insert_cell_top メゾッドと delete_cell_top メゾッドを呼ぶ必要がありました。それを踏まえると、ダミーのデータ部を持つ head を作成するほうが実装が単純になると考えられます。

2 必須課題 2-2

2.1 QueueArray.java の作成

2.1.1 実装の方針

QueueArray クラスにはエンキューを行うための enqueue メゾッドとデキューを行うための dequeue。キュー全体を表示する display メゾッドを作成した。そしてその動作を確認するための main メゾッドを作成する。

1. QueueArray クラスのコンストラクタ

キューの作成に必要なものとしてコンストラクタで整数の配列を表す queue、格納できるメモリ量を表す長さを表す length、キューの先頭を front 末尾を rear と定義した。

2. enqueue メゾッド

まずオーバーフローの処理を行う。もともとの配列 Queue を超えるような値の挿入が起こってしまう可能性があるからだ。

オーバーフローしないことがわかったら要素の末尾に引数で指定した値を代入して rear に適切な処理を行う。

3. dequeue メゾッド

まずアンダーフローの処理を行う。それはキューの要素が存在しない場合にデキューを行わないようにするためである。

アンダーフローの処理後に要素の先頭の処理をしてその値を返すようにする。

4. display メゾッド

要素の位置はコンストラクタで指定した front と rear をつかって表現することができるので、その間の値を返すように実装を行う。

```java
public class QueueArray {
    int length, front, rear;
    int[] queue;

    // 指定された長さの配列を生成するコンストラクタ
    QueueArray(int len) {
        queue = new int[len];
        length = len;
        front = 0;
        rear = 0;
    }

    // queue の後ろに引数を代入する
    void enqueue(int val) {
        if (rear > length-1) {
            System.err.println("Queue␣Overflow␣!!");
            System.exit(1);
        }
        queue[rear] = val;
        rear++;
    }

    // queue の先頭を返し、番号を一つずらす
    int dequeue() {
        if (front == rear) {
            System.err.println("Queue␣Underflow␣!!");
            System.exit(1);
        }
        int x = queue[front];
        front++;
        return x;
    }

    // queue の先頭から末尾までを出力
    void display() {
        for (int i = front; i < rear; i++) {
            System.out.print(queue[i]);
        }
        System.out.println();
    }

    // QueueArray クラスの動作確認
    public static void main(String[] args) {
          QueueArray queue = new QueueArray(10);

          queue.enqueue(1);
          queue.enqueue(2);
          queue.display();                          // 12

          System.out.println(queue.dequeue());     // 1
          System.out.println(queue.dequeue());     // 2
    }
}
```

図 2　QueueArray.java のソースコード

2.1.2　実装コードおよびコードの説明

図 2QueueArray.java のソースコードを示す.

4.1.1 節で述べた，enqueue メソッドと dequeue メソッドは，それぞれ 14~21 行目，24~32 行の部分に相当する.

enqueue は末尾が配列の長さより大きくなる場合はオーバーフローとなり終了する。15~18 行目でその実装をしている。

オーバーフローを起こす場合、System.exit(1) を書いて終了することができる。

オーバーフローを起こさない場合、queue[rear] が要素の末尾にあたるのでそこに引数として与えた int 型

の変数 val を代入する。
キューのサイズを変えるために rear の値を一つ上げる。そうすることで要素の末尾は右に一つずれたような挙動になる.

dequeue は要素の中身がない場合はアンダーフローとなり終了する。25~28 行目でその実装をしている。front と rear が等しいときに dequeue を行うと、アンダーフローになるので、その条件のときに Queue Underflow!!と出力して先程と同様に System.exit(1) によりプログラムを終了させる。デキューされるのは queue[front] であるのでその値を返す値である x に代入する。引数の値を代入した後に front を一つ動かし要素を拡大する。

2.1.3 実行結果

1.1.3 節で述べたコンパイル方法と同様に，QueueArray.java を javac コマンドでコンパイルし，QueueArray.class を生成する．その後，以下のコマンドを入力することによって，QueueArray.java のプログラムを実行する.

```
-------------------------------------
$ java QueueArray
12
1
2
-------------------------------------
```

配列の長さが 10 の QUEUE を作成し、そこに順番に 1, 2 をエンキューする。display すると 12 が出力される。

デキューをすると 1 次に 2 が出力される。この状態でデキューを行うと underflow!が出力されて終了することも確認できた。

2.1.4 考察

enqueue メゾッドの計算量は O(1) であり、dequeue メゾッドの計算量も O(1) である。これはあらかじめ先頭と末尾の位置を変数で格納していることにより実現できた。

3 発展課題 2-3

3.1 双方向循環リストの作成

3.1.1 実装の方針

ListDL クラスには次のポインタを表す next と前のポインタを表す prev 値を表す val を定義する。それらに対して以下の処理を行い実装をする。

1. コンストラクタ

はじめは循環するように next と prev の関係を作る。そのために、this の次も前も指すポインタ部は this になるように書く。引数が与えられたときはその値を格納する。

2. insertNext, insertPrev メゾッド

引数となるリストを cell とおくと前者は this -¿ cell -¿ this.next
後者は this.prev -¿ cell -¿ this
という位置関係になっている。この 2 つの関係は 3 つのリストの位置関係というコードで共通するため、__insertNext で共通化してコードを実装する。

3. delete メゾッド

this.prev -¿ this.next の結びつけをおこない、その後 this を初期化する。それぞれ__Delete メゾッドと initLinks メゾッドを作成する。

4. search メゾッド

コピーを作成し、ループさせる。データが引数と同じ場合にそのリストを返すように実装します。またリストのループはポインタを移動していき、リストを一周して this になった場合に終了する。終了したら null を返すように実装する。

5. display メゾッド

4 と同様のループ機構を書き、データ部をコマンドラインへ出力するようにする。

3.1.2 実装コードおよびコードの説明

図 4ListDL.java のソースコードを示す．4.1.1 節で述べた，

コンストラクタは 6∼13 行目の部分に相当する．head セルはデータ値を持たないようにするため、引数がない場合にはデータ値である val は定義しない。そして、引数がある場合のコンストラクタにはデータ値を定義する。またどちらにも循環するようにリンクの初期化メゾッドを使っている。

insertNext, insertPrev メソッドは 36∼43 行目の部分に相当する．セルの前後にそれぞれ双方向にリストを結びつけるための 4 つコードがそれぞれ書かれている。

delete メソッドは 46∼49 行目の部分に相当する．セルの前後をリンク付ける。その後削除されたセルから前後のセルが呼び出せられるのを防ぐため、リンクの初期化を行う。

search メソッドは 52∼61 行目の部分に相当する．コピーを一周ループさせ、データの評価を行う。このときにコピーをポインタ部のセルにさせようにして終了条件をそのコピーが this になったときにする。そうすることでループの開始では while の条件に含まれないようにしている。データ値と引数が等しいときにそのリストを返す。このループ中でこの条件を満たすものがない場合、null を返す。

display メソッドは 63∼70 行目の部分に相当する．ループ機構は同じである。while ループの中でデータ値を出力している。一周ループした後の 69 行目で改行をした。

このコードの確認のため、このクラス内に main メゾッドを実装した。head をダミーセルとして 1-¿2-¿5 と表示できるようにリストを挿入する。

elem は search メゾッドを使うことで 2-¿5-¿1 というリストになる。3 を挿入して 1-¿2-¿3-¿5 のリストに、最後に 5 を削除して 1-¿2-¿3 というリストになることを想定している。

3.1.3 実行結果

1.1.3 節で述べたコンパイル方法と同様に，ListDL.java を javac コマンドでコンパイルし，ListDL.class を生成する．その後，以下のコマンドを入力することによって，ListDL.java のプログラムを実行する．

```
-------------------------------------
$ java ListDL
```

```
12
1
2
-------------------------------------
```

3.1.4 考察

search メゾッドの計算量は単純な線形探索を採用しているので O(n) である。insert, delete メゾッドの計算量は O(1) である。

4 必須課題 2-4

4.1 双方向循環リストの改良

4.1.1 実装の方針

ListDLInt はデータを int 型にする。ListDL のデータの型は Object 型にした。これはどのような型に対しても代入可能にするためである。その後に配列や外部ファイルからのリスト作成用メゾッドを実装する。

1. readFromArray メゾッド

引数として受けとった配列を順に insert する。

2. writeToArray メゾッド

ループするごとにそのインデックスに応じた配列に格納する。このときリスト数分の配列をさくせいするためにリストの長さを返すメゾッド length も作成する。

3. readFromFil, writeToFile、メゾッド

ファイルから一行ずつ読み込むコードを実装する。

4.1.2 実装コードおよびコードの説明

図 4ListDL.java のソースコードを示す．追加コードは 75~143 行目である。リストをループするコードを少々改良した。for 文で実装することでコード行数を減らしている。readFromFile, writeToFile メゾッドは BufferedWriter でファイルを開き、その後終了した後にメモリ管理のためにファイルを閉じる必要がある。そのときに try_finally 文を使った。これにより、エラーなどにも対応できる。

4.1.3 実行結果

1.1.3 節で述べたコンパイル方法と同様に，ListDL.java を javac コマンドでコンパイルし，ListDL.class を生成する．その後，以下のコマンドを入力することによって，ListDL.java のプログラムを実行する．

```
-------------------------------------
$ java ListDLmain
12
1
2
-------------------------------------
```

4.1.4　考察

```java
// ListDL.java
public class ListDL {
    private ListDL prev, next;
    int val;

    ListDL() {
        initLinks();
    }

    ListDL(int val) {
        this.val = val;
        initLinks();
    }

    // prev <=> cell <=> next
    private static void __Insert(ListDL cell, ListDL prev, ListDL next) {
        prev.next = cell;
        next.prev = cell;
        cell.next = next;
        cell.prev = prev;
    }

    // prev と next の結びつけ
    private static void __Delete(ListDL prev, ListDL next) {
        // 条件を満たすときはもでないnextnull
        prev.next = next;
        next.prev = prev;
    }

    // 結びつきをなくす
    private void initLinks() {
        this.prev = this.next = this;
    }

    // this => cell => this.next
    void insertNext(ListDL cell) {
        __Insert(cell, this, this.next);
    }

    // this.prev => cell => this
    void insertPrev(ListDL cell) {
        __Insert(cell, this.prev, this);
    }

    // this.prev => this.next (no this)
    void delete() {
        __Delete(this.prev, this.next);
        initLinks();
    }

    // リストのコピーを走査して、データ値が引数と同じものを返す
    ListDL search(int i) {
        ListDL tmp = this.next;
        while (tmp != this) {
            if (tmp.val == i) {
                return tmp;
            }
            tmp = tmp.next;
        }
        return null;
    }

    void display() {
        ListDL tmp = this.next;
        while (tmp != this) {
            System.out.print(tmp.val);
            tmp = tmp.next;
        }
        System.out.println();
    }

    // リストの実装をテストするためのクラス
    public static void main(String[] args) {
        ListDL head = new ListDL(); // ダミーセルの生成
        ListDL elem;

        head.insertNext(new ListDL(2)); // セルの先頭への追加
        head.insertNext(new ListDL(1));
        head.insertPrev(new ListDL(5)); // セルの末尾への追加
        head.display(); // リストの表示

        elem = head.search(2); // セルを探す elem.val == 5
        elem.insertNext(new ListDL(3)); // 探したセルの直後にセルを追加
        head.display();

        elem = head.search(5); // セルを探す elem.val == 5
        elem.delete(); // 探したセルを削除
        head.display();
    }
}
```

```java
import java.io.*;

// ListDLInt.java
public class ListDL {
    private ListDL next, prev;
    Object val;

    ListDL() {
        // 循環する
        initLinks();
    }

    ListDL(Object val) {
        initLinks();
        this.val = val;
    }

    // prev <=> cell <=> next
    private static void __Insert(ListDL cell, ListDL prev, ListDL next) {
        prev.next = cell;       // prev => cell
        next.prev = cell;       // next => cell
        cell.next = next;       // cell => next
        cell.prev = prev;       // cell => prev
    }

    // prev と next の結びつけ
    private static void __Delete(ListDL prev, ListDL next) {
        prev.next = next;
        next.prev = prev;
    }

    // 初期化
    private void initLinks() {
        this.next = this;
        this.prev = this;
    }

    // this => cell => this.next
    void insertNext(ListDL cell) {
        __Insert(cell, this, this.next);
        // this.next = cell;
    }

    // this.prev => cell => this
    void insertPrev(ListDL cell) {
        __Insert(cell, this.prev, this);
        // this.prev = cell;
    }

    // this.prev => this.next (no this)
    void delete() {
        __Delete(this.prev, this.next);
        initLinks();
    }

    // リストのコピーを走査して、データ値が引数と同じものを返す
    ListDL search(Object i) {
        for (ListDL j = this.next; j != this; j = j.next) {
            if (i.equals(j.val)) {
                return j;
            }
        }
        return null;
    }

    void display() {
        for (ListDL j = this.next; j != this; j = j.next) {
            if (j.val != null) {
                System.out.print(j.val + "␣->␣");
            }
        }
        System.out.println();
    }

    // 配列からリストの要素を読み込むメソッド readFromArray()
    ListDL readFromArray(Object[] Array) {
        for (Object val : Array) {
            this.insertPrev(new ListDL(val));
        }
        return this;
    }

    // リストの要素数を返す
    int length() {
        int Count = 0;
        for (ListDL i = this.next; i != this; i = i.next) {
            Count++;
        }
        return Count;
    }
```