二分木ヒープ

1. はじめに

様々なアルゴリズムにおいて、ある要素の集合またはリストから、「最小」な要素を取り出す必要がある。そのような場合に使われる標準的データ構造が二分木ヒープ(binary heap)である。

あるオブジェクトOを考える。そのオブジェクトは、ラベル$O.label$と値$O.value$を持つとする。このようなオブジェクトを保存する二分木ヒープについて考える。二分木ヒープは以下の二つの制約のある二分木である。

1. ある頂点のオブジェクトの値は、その子の頂点の値よりも小さいか等しい。
2. 完全二分木であること。つまり、最下層以外の第k層には、2^{k-1}個の頂点があり、最下層のみ、左から詰めてあること。

上記の 1 より、この二分木は半順序木となる。

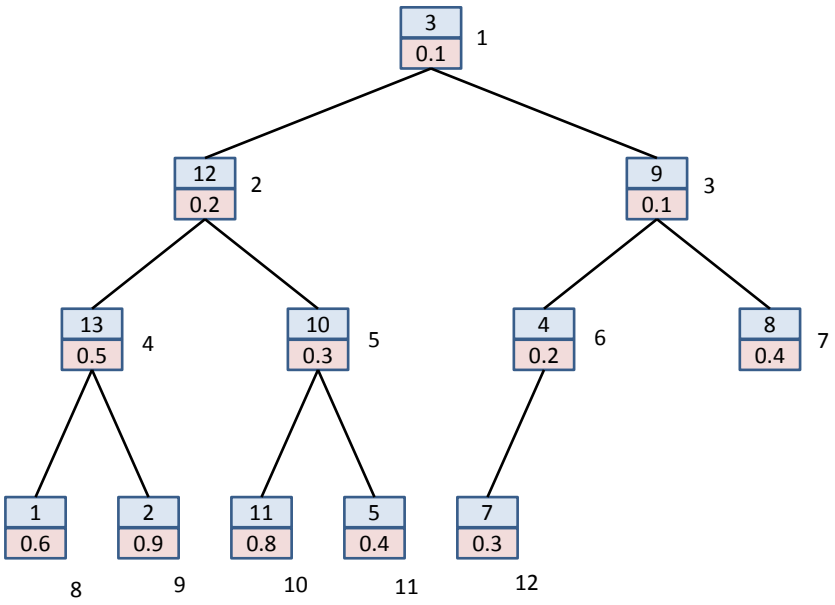

Figure 1　二分木ヒープの例

例の中で、各頂点の右の番号を位置と呼ぶことにする。根を 1 番とし、第k層の頂点を左から2^k番から順番に番号を付けることとする。このような番号を付けることで、ある頂点iの親の番号は$\lfloor i/2 \rfloor$であること、その子の番号は$2i$及び$2i+1$であることが分かる。

2. データ構造

ヒープに保存するデータは、番号付けられて保存される。従って、リストLとして保存することとする。

3. アルゴリズム

3.1. 要素の追加

新しい要素の追加は、リストの終端に置くことで開始する。つまり、最下層の一番右、または新たに最下層を生成してその一番左となる。この後、この要素を正しい位置に移動させる。この操作をシフトアップと呼ぶことにする。

```
void add(O o){
    int n = |L|;
    L.append(o);
    n++;
    shiftUp(n);
}
```

ある位置kに居る要素が、親の位置$\lfloor k/2 \rfloor$にある要素よりも小さいならば、二つの要素を入れ替える必要がある。この操作を繰り返すことで、追加した要素を正しい位置に配置する。シフトアップのアルゴリズムは、以下のようになる。

```
void shiftUp(int k){
    if(k > 1 && isLess(k, ⌊k/2⌋)){
        swap(k, ⌊k/2⌋);
        k = ⌊k/2⌋;
        shiftUp(k);
    }
}
```

ここでisLess(i, j)は、$o_i.value < o_j.value$のとき真となるメソッド、swap(i, j)は、o_iとo_jの位置を入れ替えるメソッドである。

3.2. 最小要素の取出し

ヒープの目的は、最小要素を見つけることである。最小要素は、根として保存されている。この根を取り除くと、ヒープとしての制限を満たさない。そこで、最小要素を取り除

いた後、以下のような手順でヒープを再構築する。

```
O poll(){
    O  t = L.get(1);
    O  x = L.removeLast();
    L.set(1, x);
    shiftDown(1);
    return t;
}
```

最小要素、つまり木の根を取り除いた後、リストの最後の要素を仮に根の位置に置く。こうすることで、木が完全であることが復元される。その後、この要素を下向きに移動させ、適切な位置に配置する。この操作をシフトダウンと呼ぶことにする。

ある位置kに居る要素の値が、その子の位置$2k$または$2k+1$に居る要素の小さいほうよりも大きい場合に、その小さいほうの要素と入れ替える操作をシフトダウンと呼ぶ。これにより、ヒープであることが復元される。シフトダウンのアルゴリズムは以下のようになる。

```
void shiftDown(int k){

    int n = |L|;

    if(2×k ≤ n){
        int j = 2×k;
        if( j < n  && isLess( j+1, j)) j++;
        if(isLess(k, j))return;
        swap(k, j);
        shiftDown( j);
        }
    }
}
```

貪欲アルゴリズム (GREEDY ALGORITHM, KRUSKAL ALGORITHM)

$T = \emptyset$
H : G の弧の重みに関するヒープ
while (T は G の極大木ではない) {
    $a = H$.poll() ヒープから最小要素を取得
    $a_{new} = \text{null}$
    while($a_{new} == \text{null}$){
        if ($T \cup \{a\}$ は閉路を持たない) {
            $a_{new} = a$
        }
    }
    $T = T \cup \{a_{new}\}$
}

BinaryHeap.java

```java
package utils;

import java.text.NumberFormat;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

/**
 *
 * @author tadaki
 */
public class BinaryHeap<T> {

    /** データを保持するリスト */
    private List<T> list;
    /** 要素を比較する方法 */
    private Comparator<T> comparator = null;
    /** 要素数 */
    private int n;

    /**
     * コンストラクタ：比較方法を指定する場合
     * 比較方法を指定しない場合は、
     *     要素はインターフェイスComparableを実装していること
     * @param comparator
     */
    public BinaryHeap(Comparator<T> comparator) {
        this();
        this.comparator = comparator;
    }

    public BinaryHeap() {
        list = Collections.synchronizedList(new ArrayList<T>());
        list.add(null);
        n = 0;
    }

    /**
     * 新しい要素を追加する
     * @param t
     * @return
     */
    public boolean add(T t) {
```

```
        boolean b = list.add(t);
        if (b) {
            n++;
            shiftUp(n);
        }
        return b;
    }

    /**
     * 最小の要素を得る：削除しない
     * @return
     */
    public T peek() {
        if (n == 0) {
            return null;
        }
        return list.get(1);
    }

    /**
     * 最小の要素を取り出し、削除する
     * @return
     */
    public T poll() {
        T t = null;
        if (n == 0) {
            return null;
        }
        if (n == 1) {//残りの要素が一つ
            t = list.remove(n);
            n--;
        } else {
            T x = list.remove(n);
            t = list.get(1);
            n--;
            list.set(1, x);
            shiftDown(1);
        }
        return t;
    }

    /**
     * 特定の要素の値を小さくした場合の再配置
     * @param t
```

```
     */
    public void reduceValue(T t) {
        int k = list.indexOf(t);
        shiftUp(k);
    }

    /**
     * リストの取得
     * @return
     */
    public List<T> getList() {
        return list;
    }

    public boolean isEmpty() {
        return (n == 0);
    }


/**************************************************************************/
    /**
     * あるk にあるobjectを上位の適切な位置に置く
     * @param k
     */
    private void shiftUp(int k) {
        if (k > 1 && isLess(k, (int) (k / 2))) {
            int j = (int) (k / 2);
            swap(k, j);
            shiftUp(j);
        }
    }

    /**
     * あるk にあるobjectを下位の適切な位置に置く
     * @param k
     */
    private void shiftDown(int k) {
        if (2 * k <= n) {
            int j = 2 * k;
            if (j < n && isLess(j + 1, j)) {
                j++;
            }
            if (isLess(k, j)) {
                return;
```

```
            }
            swap(k, j);
            shiftDown(j);
        }
    }

    private boolean isLess(int i, int j) {
        int a = 0;
        if (comparator == null
                && list.get(i) instanceof Comparable) {
            Comparable x = (Comparable) list.get(i);
            Comparable y = (Comparable) list.get(j);
            a = x.compareTo(y);
        } else {
            a = comparator.compare(list.get(i), list.get(j));
        }
        return (a < 0);
    }

    private void swap(int i, int j) {
        T o = list.get(i);
        list.set(i, list.get(j));
        list.set(j, o);
    }


/*************************************************************************/
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        /** サンプルとなるデータオブジェクト */
        class Data {

            int label;
            double value;

            public Data(int label, double value) {
                this.label = label;
                this.value = value;
            }
        }
        /** サンプルとなるデータの比較方法 */
        class CompareData implements Comparator<Data> {
```

```
        public int compare(Data o1, Data o2) {
            int a = 0;
            if (o1.value > o2.value) {
                a = 1;
            }
            if (o1.value < o2.value) {
                a = -1;
            }
            return a;
        }
    }
    BinaryHeap<Data> h = new BinaryHeap(new CompareData());
    /** データ追加 */
    int n = 20;
    for (int i = 0; i < n; i++) {
        Data d = new Data(i + 1, Math.random());
        h.add(d);
    }
    /** 結果取得 */
    List<Data> list = h.getList();

    /** 最小値を削除 */
    for (int i = 0; i < 2; i++) {
        Data d = h.poll();
        n--;
    }
    /** 出力 */
    NumberFormat format = NumberFormat.getInstance();
    format.setMaximumFractionDigits(4);

    int l = (int) (Math.log((double) n) / Math.log(2.) + 0.1);
    for (int i = 0; i <= l; i++) {
        int m = (int) (Math.pow(2., (double) i) + 0.1);
        System.out.println();
        for (int j = 0; j < m; j++) {
            int k = m + j;
            if (k != 0 && k <= n) {
                System.out.print("(");
                System.out.print(list.get(k).label);
                System.out.print(",");
                System.out.print(format.format(list.get(k).value));
                System.out.print(") ");
            }
```

```
                }
            }
            System.out.println();
        }
    }
```