# 目錄


Introduction 1.1

1. Data Structure 1.2

1.1 Stack 1.3

1.1.1 Stack: Revert String 1.4

1.1.2 Stack: Brackets Matching 1.5

1.1.3 Stack: Reverse Polish Notation 1.6

1.1.4 Stack: Calculattion for Reverse Polish Notation 1.7

1.2 Queue 1.8

1.2.1 Priority Queue 1.9

1.3 Linked List 1.10

1.3.1 Linked List - Reorder Function 1.11

1.3.2 Ordered Linked List 1.12

1.4 Tree 1.13

1.4.1 Binary Search Tree 1.14

1.4.2 Heap Tree 1.15

1.4.3 Red-Black Tree 1.16

1.4.3.1 RB Tree Insertion 1.17

1.4.3.2 RB Tree Insertion Implementation 1.18

1.4.3.3 RB Tree Delete 1.19

1.4.3.4 RB Tree Delete Implementation 1.20

1.4.4 B-Tree 1.21

2. Algorithm 1.22

2.1 Sort 1.23

2.1.1 Bubble Sort 1.24

2.1.2 Selection Sort 1.25

2.1.3 Insertion Sort 1.26

2.1.4 Merge Sort 1.27

2.1.5 Quick Sort 1.28
2.1.6 Merge Sort v.s. Quick Sort 1.29
2.2 Search 1.30
2.2.1 Binary Search 1.31
2.3 Dynamic Programming 1.32
2.3.1 Fibonacci Series 1.33
2.3.2 Find Longest Common Suffix 1.34
X. Time Complexity Cheat Sheet 1.35

# 資料結構&演算法筆記 (Datastructures & Algorithms)

Hope this note helps you well.

# Data Structure

好的老師帶你上天堂 , 爛的老師讓你住套房. 資料結構亦是如此, 好的資料結構讓你事半功倍, 壞的資料結構讓你...嗯???

# Stack(堆疊)

線性表:

又稱爲順序表, 是一個線性的序列結構, 是一個含有n≥0個節點的有線序列, 對於其中的節點:

1. 有且僅有一個開始節點, 沒有前驅, 但有後繼節點
2. 有且僅有一個終端節點, 沒有後繼, 但有前驅節點
3. 其他的節點都有且僅有一個前驅和後繼節點
4. 維基百科

線性表跟陣列的比較:

1. 是兩種不同的資料結構, 陣列有維度的概念, 線性表沒有; 線性表有前驅/後繼節點的概念, 且線性表的資料是相互有關聯的, 但陣列並沒有這些概念.
2. 線性表可以使用陣列來實作, 通常用一維陣列來作爲其資料的存儲結構.

什麼是Stack(堆疊):

1. Stack是一種特殊的線性表(Linear List), 限定只能在表的一端進行插入和刪除操作, 俗稱後進先出(LIFO).
2. 操作資料的這一端就稱爲表頭, 或top, 相對地, 另一端叫bottom, 不含任何元素的時候叫做empty stack.

Stack的基本操作:

1. push: 塞東西到stack
2. pop: 把最上面的東西彈出來
3. peek: 只觀看最上面的東西, 不要彈出來

```
package idv.carl.datastructures.stack;

/**
 * @author Carl Lu
 */
public class Stack {

    private int[] data;
```

```
    private int top = -1;

    public Stack(int length) {
        data = new int[length];
    }

    public void push(int element) {
        if (top < this.data.length - 1) {
            top++;
            this.data[top] = element;
        }
    }

    public int pop() {
        return data[top--];
    }

    public int peek() {
        return data[top];
    }

    public boolean isEmpty() {
        return top == -1;
    }

    public boolean isFull() {
        return top == data.length - 1;
    }

    public int size() {
        return top + 1;
    }

}
```

## Stack的效率

Stack的pop跟push之時間複雜度皆爲常數, 即**O(1)**, 不涉及複製和移動操作

# Stack: Revert String

產生反序的字串
接續前一章中的自製Stack資料結構, 我們可用此資料結構來做出顛倒字串的動作:

原始碼點我

```
package idv.carl.datastructures.stack;

/**
 * @author Carl Lu
 */
public class ReverseString {

    public String doRevert(String input) {
        StringBuffer result = new StringBuffer();
        Stack stack = new Stack(100);

        // Step1. Read the string as chars one by one
        char[] chars = input.toCharArray();

        // Step2. Push those chars into the stack sequentially
        for (char c : chars) {
            stack.push(c);
        }

        // Step3. Pop all chars to form a new string
        while (!stack.isEmpty()) {
            result.append((char) stack.pop());
        }

        return result.toString();
    }

}
```

# Stack: Brackets Matching

當想要檢驗一個輸入當中的括號順序跟種類是否正確, 可以透過stack來做檢查.

原始碼點我

```
package idv.carl.datastructures.stack;

/**
 * @author Carl Lu
 */
public class CheckBrackets {

    private final static char FORMER_PARENTHESES = '(';
    private final static char LATTER_PARENTHESES = ')';
    private final static char FORMER_BRACKET = '[';
    private final static char LATTER_BRACKET = ']';
    private final static char FORMER_BRACE = '{';
    private final static char LATTER_BRACE = '}';

    public boolean check(String input) {
        boolean match = true;
        Stack stack = new Stack(50);
        // Step1. Transform the string into char array
        char[] chars = input.toCharArray();

        for (int i = 0; i < chars.length; i++) {
            char c = chars[i];

            if (isFormerPart(c)) {
                /*
                 * Step2. Read chars sequentially, push the char
 into stack
                 * if it's the former part of the brackets
                 */
                stack.push(c);
            } else if (isLatterPart(c)) {
                /*
```

```
                 * Step3. If encounter the latter part of the br
ackets,
                 * pop a value from stack and try to match
                 */
                char former = (char) stack.pop();
                if (misMatch(former, c)) {
                    System.out.println("Char mismatch, index at:
 " + (i + 1));
                    match = false;
                } else {
                    System.out.println("Char match.");
                    match = true;
                }
            }

        }
        return match;
    }

    private boolean isFormerPart(char c) {
        return (c == FORMER_PARENTHESES || c == FORMER_BRACKET |
| c == FORMER_BRACE);
    }

    private boolean isLatterPart(char c) {
        return (c == LATTER_PARENTHESES || c == LATTER_BRACKET |
| c == LATTER_BRACE);
    }

    private boolean misMatch(char former, char latter) {
        return ((former == FORMER_PARENTHESES && latter != LATTE
R_PARENTHESES)
                || (former == FORMER_BRACKET && latter != LATTER
_BRACKET) || (former == FORMER_BRACE && latter != LATTER_BRACE))
;
    }

}
```

# Stack: Reverse Polish Notation

計算算式表達式
後綴表示法: 又稱爲逆波蘭表示法, 此種表示法將運算子寫在運算物件的後面, 例如, 把a+b寫成ab+. 此種表示法的優點是根據運算元和運算子的出現次序進行運算, 不需要使用括號, 也便於用程式來實作求值.

如何把中綴表達式轉換成後綴表達式:
把中綴表達式換成後綴表達式不用做算術運算, 只是把運算子和運算元重新按照後綴表達式的方式進行排列而已.

1. 由左至右讀取表達式中的字元
2. 若爲運算元, 則複製到後綴表達式字串中
3. 若爲左括號, 則push至stack中
4. 若爲右括號, 從stack中pop字元至後綴表達式, 直到遇到左括號, 然後把左括號pop出來
5. 若是運算子, 且若此時stack top的運算子優先級≥此運算子, 彈出stack top的運算子到後綴表達式, 直到發現優先級更低的元素位置, 把運算子push至stack
6. 讀到輸入的尾端, 將stack元素pop出來直到該stack爲empty, 將符號寫入後綴表達式中

原始碼點我

```
package idv.carl.datastructures.stack;

/**
 * @author Carl Lu
 */
public class ReversePolishNotation {

    private final static int PRIORITY_LEVEL_1 = 1;
    private final static int PRIORITY_LEVEL_2 = 2;
    private final static char ADD = '+';
    private final static char MINUS = '-';
    private final static char MULTIPLY = '*';
    private final static char DIVIDE = '/';
    private final static char LEFT_PARENTHESES = '(';
```

```
    private final static char RIGHT_PARENTHESES = ')';

    public String doTransfer(String input) {
        StringBuffer result = new StringBuffer();
        Stack stack = new Stack(50);
        // Step1. Transform the input into char array
        char[] chars = input.toCharArray();
        // Step2. Apply related operation rule on each char
        for (int i = 0; i < chars.length; i++) {
            char c = chars[i];

            // 2.1 If it's operator, operate it according to the
 priority
            if (isAddOrMinus(c)) {
                doOperation(stack, result, c, PRIORITY_LEVEL_1);
            } else if (isMultiplyOrDivide(c)) {
                doOperation(stack, result, c, PRIORITY_LEVEL_2);
            }
            // 2.2 If it's left bracket, push to stack
            else if (c == LEFT_PARENTHESES) {
                stack.push(c);
            }
            // 2.3 If it's right bracket, pop from stack, stop w
hen encounter the left bracket
            else if (c == RIGHT_PARENTHESES) {
                handleForRightBracket(stack, result);
            }
            // 2.4 If it's operand, add to output directly
            else {
                result.append(c);
            }
        }

        // Step3. If iterate to the end, pop up the operator int
o the output
        while (!stack.isEmpty()) {
            result.append((char) stack.pop());
        }
        return result.toString();
    }
```

```
    private void handleForRightBracket(Stack stack, StringBuffer
 result) {
        // Step1. Pop up value from stack into result
        while (!stack.isEmpty()) {
            char top = (char) stack.pop();
            // Step2. Stop until meet the left bracket
            if (top == LEFT_PARENTHESES) {
                break;
            } else {
                result.append(top);
            }
        }
    }

    private boolean isAddOrMinus(char c) {
        return (c == ADD || c == MINUS);
    }

    private boolean isMultiplyOrDivide(char c) {
        return (c == MULTIPLY || c == DIVIDE);
    }

    private void doOperation(Stack stack, StringBuffer result, c
har c, int priority) {
        // Step1. Obtain value from the top of stack
        while (!stack.isEmpty()) {
            char top = (char) stack.pop();
            // Step2. Compare with input char according to prior
ity
            // 2.1 If top is left bracket, do nothing (need to p
ush back the value)
            if (top == LEFT_PARENTHESES) {
                stack.push(top);
                break;
            } else {
                // Determine the priority of the top element
                int pTop;
                if (isAddOrMinus(c)) {
                    pTop = PRIORITY_LEVEL_1;
```

```
                } else {
                    pTop = PRIORITY_LEVEL_2;
                }

                if (pTop >= priority) {
                    // 2.2 If p(top) ≥ p(value), add top to resu
lt
                    result.append(top);
                } else {
                    // 2.3 If p(top) < p(value), do nothing (nee
d to push back the value)
                    stack.push(c);
                    break;
                }
            }
        }
        // Step3. After found the lower priority element, push i
nto the stack
        stack.push(c);
    }

}
```

# Stack: Calculation for Reverse Polish Notation

計算後綴表達式求值

1. 從左到右依序讀取表達式中的字元
2. 若是運算元, push to stack
3. 若是運算子, 從stack中pop出兩個資料進行運算, 並把結果push入stack中
4. 直到表達式結束

原始碼點我

```
package idv.carl.datastructures.stack;

/**
 * @author Carl Lu
 */
public class ReversePolishNotationCalculator {

    private final static char ADD = '+';
    private final static char MINUS = '-';
    private final static char MULTIPLY = '*';
    private final static char DIVIDE = '/';

    public int calculate(String rpnExpression) {
        Stack stack = new Stack(20);
        char[] chars = rpnExpression.toCharArray();

        // Step1. Obtain char from input sequentially
        for (char c : chars) {
            /*
             * Step2. Push the element into stack if it's a oper
and
             * (here we only assume the operand is n, and 0 ≤ n
≤ 9)
             */
            if (c >= '0' && c <= '9') {
```

```
                stack.push((int) c - '0');
            }
            /*
             * Step3. If it's a operator, pop two value from sta
ck for calculation, then put the
             * result back to the stack
             */
            else {
                int latter = stack.pop();
                int former = stack.pop();
                int tmp = 0;
                if (c == ADD) {
                    tmp = former + latter;
                } else if (c == MINUS) {
                    tmp = former - latter;
                } else if (c == MULTIPLY) {
                    tmp = former * latter;
                } else if (c == DIVIDE) {
                    tmp = former / latter;
                }
                // Push back to the stack
                stack.push(tmp);
            }
        }
        return stack.pop();
    }

}
```

# Queue(佇列)

什麼是Queue: Queue是一種特殊的線性表, 限定只能在表的一端進行插入(隊尾), 而在另一端進行刪除操作(隊頭), 特點是"先進先出"(FIFO).

Queue的基本操作:

1. insert:在隊尾插入資料
2. remove: 從隊頭移走資料
3. peek: 查看隊頭的資料

Circular Queue: 爲了避免queue未滿, 卻不能插入新資料項的問題, 可以讓隊頭隊尾的指標繞回陣列開始的位置, 這就是circular queue, 又稱作ring buffer.

Queue的效能: insert和remove的時間複雜度均爲**O(1)**

一個簡單的circular queue的實作(原始碼點我):

```
package idv.carl.datastructures.queue;

/**
 * @author Carl Lu
 */
public class CircularQueue {
    private int[] queue;
    private int head;
    private int tail;

    // Number of elements in this queue
    private int elementCount;

    public CircularQueue(int length) {
        queue = new int[length];
        head = 0;
        tail = -1;
        elementCount = 0;
    }

    public void insert(int element) {
```

```
        // Check the tail index already exceed the max length or
 not
        if (tail == queue.length - 1) {
            tail = -1;
        }
        tail++;
        queue[tail] = element;
        elementCount++;

        if (elementCount > queue.length) {
            elementCount = queue.length;
        }
    }

    public int remove() {
        if (elementCount == 0) {
            return 0;
        }
        int temp = queue[head];
        queue[head] = 0;
        // Check the head index already exceed the max length or
 not
        if (head == queue.length - 1) {
            /*
             * If the removed node is tail, it means that the ne
xt node will be removed must be the
             * head node since this is a circular queue, so rese
t head index to 0.
             */
            head = 0;
        } else {
            head++;
        }
        elementCount--;
        return temp;
    }

    public int peek() {
        return queue[head];
    }
```

```
    public boolean isEmpty() {
        return elementCount == 0;
    }

    public boolean isFull() {
        return elementCount == queue.length;
    }

    public int getElementCount() {
        return elementCount;
    }

}
```

# Priority Queue

Priority Queue: 優先佇列, 即資料按照關鍵字排好的佇列, 詳細可見維基百科

Priority Queue的效率: 下方的是比較粗糙的實作, insert需要**O(n)**, 而remove是O(1), 通常這邊要改進insert的效能可以用**heap tree**來實作內部的資料結構, 這樣可以令insert的效能提升至**O(logn),** 所以也有人將改進後的priority queue稱爲是一種**complete binary tree.**

原始碼點我

```
package idv.carl.datastructures.queue;

/**
 * @author Carl Lu
 */
public class PriorityQueue {

    private int[] queue;

    // Number of elements in this queue
    private int elementCount;

    public PriorityQueue(int length) {
        queue = new int[length];
        elementCount = 0;
    }

    public void insert(int element) {
        if (elementCount == queue.length) {
            return;
        } else if (elementCount == 0) {
            queue[elementCount] = element;
        } else {
            /*
             * If the queue is not empty, execute sorting before
 insert the element
             */
```

```
            int i;
            for (i = elementCount - 1; i >= 0; i--) {
                if (element > queue[i]) {
                    queue[i + 1] = queue[i];
                } else {
                    break;
                }
            }
            /*
             * Since we use i-- in the for loop on line 28,
             * so here we need to add 1 for the index i.
             */
            queue[i + 1] = element;
        }
        elementCount++;
    }

    public int remove() {
        if (elementCount == 0) {
            return 0;
        }
        // Decrease the elementCount because it already be incre
ased at the end of insert.
        elementCount--;
        // Remove the last element
        int removed = queue[elementCount];
        // Assume that 0 means the data was removed
        queue[elementCount] = 0;
        return removed;
    }

    public int peek() {
        return queue[elementCount - 1];
    }

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

    public boolean isFull() {
```

```
        return elementCount == queue.length;
    }

    public int getElementCount() {
        return elementCount;
    }

}
```

# Linked List(連結串列)

Linked List也是一種特殊的線性表, 其由一系列的節點組成, 節點的順序是通過節點元素中的指標連接次序來確定的. Linked List中的節點包含兩個部分, 一個是其自身需存放的資料, 另一個是指向下一個節點的參照(reference).

Linked List v.s. Array

1. 都可作爲資料的儲存結構
2. Array: 固定長度, 依序存放
3. Linked List: 無容量限制, 非連續和非順序的儲存結構
4. 從效率上來說, linked list基本上是優於array的
5. 基本上, 只要是能用array的地方, 都可以用linked list來代替array. Linked list的缺點是引入了複雜度

Linked List的基本操作

1. 向list中插入資料
2. 從list中移除資料
3. 查看list中所有的資料
4. 查詢指定的節點
5. 刪除指定的節點

Linked List的效能

1. 在表頭插入和刪除非常快, 基本就是修改一下參照值, 時間大約爲常量, 即**O(1)**.
2. 若爲查詢/刪除特定節點, 大約需要**O(n)**次比較, 跟陣列差不多, 但仍然比陣列快, 因爲它不需要移動或複製資料.

```
package idv.carl.datastructures.list;

/**
 * @author Carl Lu
 */
public class LinkedList {

    private LinkedNode head;
    private int size = 0;
```

```
    public void insertHead(int id) {
        LinkedNode newNode = new LinkedNode(id);
        newNode.setNext(head);
        head = newNode;
        size++;
    }

    public void insertTail(int id) {
        LinkedNode newNode = new LinkedNode(id);

        if (head == null) {
            head = newNode;
            size++;
            return;
        }

        LinkedNode tail = head;

        while (tail.getNext() != null) {
            tail = tail.getNext();
        }
        tail.setNext(newNode);
        size++;
    }

    public LinkedNode removeHead() {
        if (size == 0) {
            return null;
        }

        LinkedNode tmp = head;
        head = head.getNext();
        size--;
        return tmp;
    }

    public LinkedNode find(int id) {
        LinkedNode node = head;
        while (node.getId() != id) {
```

```
            if (node.getNext() == null) {
                return null;
            } else {
                node = node.getNext();
            }
        }
        return node;
    }

    public LinkedNode remove(int id) {
        if (size == 0) {
            return null;
        }

        LinkedNode deleted = head;
        LinkedNode previous = head;

        // To search the deleted node and it's previous node
        while (deleted.getId() != id) {
            if (deleted.getNext() == null) {
                return null;
            } else {
                previous = deleted;
                deleted = deleted.getNext();
            }
        }

        // Reset the relationship of nodes
        if (deleted.equals(head)) {
            head = head.getNext();
        } else {
            previous.setNext(deleted.getNext());
        }

        size--;
        return deleted;
    }

    public void displayList() {
        LinkedNode tmp = head;
```

```
        while (tmp != null) {
            System.out.println(tmp.toString());
            tmp = tmp.getNext();
        }
    }

    public int getSize() {
        return size;
    }
}
```

原始碼點我

# Linked List - Reorder Function

若給定一個單向的連結串列:

N(1), N(2), N(3), N(4), ..., N(m-1), N(m)

請實作一個演算法可以產出如下的reorder list:

N(1), N(m), N(2), N(m-1), N(3)...

e.g.:

Input: 1,2,3,4,5

Output: 1,5,2,4,3

限制: 盡可能追求時間最佳化與空間最佳化

這裡只討論最佳解, 時間複雜度爲**O(n)**

思路:

1. 透過龜兔賽跑演算法求出中間節點, 即要對半切的那個點
2. 用剛才找到的中間點去把串列分成兩半
3. 用反序的方式重排後半段
4. 把兩半串列合起來

原始碼點我, 詳閱reorderList function.

```
package idv.carl.datastructures.list;

/**
 * @author Carl Lu
 */
public class LinkedList {

    private LinkedNode head;
    private int size = 0;

    public void insertHead(int id) {
        LinkedNode newNode = new LinkedNode(id);
```

```
        newNode.setNext(head);
        head = newNode;
        size++;
    }

    public void insertTail(int id) {
        LinkedNode newNode = new LinkedNode(id);

        if (head == null) {
            head = newNode;
            size++;
            return;
        }

        LinkedNode tail = head;

        while (tail.getNext() != null) {
            tail = tail.getNext();
        }
        tail.setNext(newNode);
        size++;
    }

    public LinkedNode removeHead() {
        if (size == 0) {
            return null;
        }

        LinkedNode tmp = head;
        head = head.getNext();
        size--;
        return tmp;
    }

    public LinkedNode find(int id) {
        LinkedNode node = head;
        while (node.getId() != id) {
            if (node.getNext() == null) {
                return null;
            } else {
```

```
                node = node.getNext();
            }
        }
        return node;
    }

    public LinkedNode remove(int id) {
        if (size == 0) {
            return null;
        }

        LinkedNode deleted = head;
        LinkedNode previous = head;

        // To search the deleted node and it's previous node
        while (deleted.getId() != id) {
            if (deleted.getNext() == null) {
                return null;
            } else {
                previous = deleted;
                deleted = deleted.getNext();
            }
        }

        // Reset the relationship of nodes
        if (deleted.equals(head)) {
            head = head.getNext();
        } else {
            previous.setNext(deleted.getNext());
        }

        size--;
        return deleted;
    }

    private LinkedNode reverseList(LinkedNode node) {
        LinkedNode previous = null;
        LinkedNode current = node;
        LinkedNode next;
```

```
        while (current != null) {
            next = current.getNext();
            current.setNext(previous);
            previous = current;
            current = next;
        }

        node = previous;
        return node;
    }

    public LinkedNode reorderList(LinkedNode node) {
        // Step1. Find the middle node by using tortoise and har
e algorithm
        LinkedNode tortoise = node;
        LinkedNode hare = tortoise.getNext();
        while (hare != null && hare.getNext() != null) {
            tortoise = tortoise.getNext();
            hare = hare.getNext().getNext();
        }

        // Step2. Split the list into two parts
        LinkedNode firstHead = node;
        LinkedNode secondHead = tortoise.getNext();
        tortoise.setNext(null);

        secondHead = reverseList(secondHead);

        // Just see it as a dummy node
        node = new LinkedNode(0);

        LinkedNode current = node;
        while (firstHead != null || secondHead != null) {
            // First, add one node from the first part into the
list
            if (firstHead != null) {
                current.setNext(firstHead);
                current = current.getNext();
                firstHead = firstHead.getNext();
            }
```

```
            // Then, add one node from the second part into the
list
            if (secondHead != null) {
                current.setNext(secondHead);
                current = current.getNext();
                secondHead = secondHead.getNext();
            }
        }

        // Since the node is dummy node, need to take the next n
ode as head
        node = node.getNext();
        return node;
    }

    public void displayList() {
        LinkedNode tmp = head;
        while (tmp != null) {
            System.out.println(tmp.toString());
            tmp = tmp.getNext();
        }
    }

    public int getSize() {
        return size;
    }
}
```

```
@Test
    public void testReorderFunction() {
        linkedList.insertTail(1);
        linkedList.insertTail(2);
        linkedList.insertTail(3);
        linkedList.insertTail(4);
        linkedList.insertTail(5);
        linkedList.reorderList(linkedList.find(1));
        assertEquals(1, linkedList.removeHead().getId());
        assertEquals(5, linkedList.removeHead().getId());
        assertEquals(2, linkedList.removeHead().getId());
        assertEquals(4, linkedList.removeHead().getId());
        assertEquals(3, linkedList.removeHead().getId());
    }
```

# Ordered Linked List(有序連結串列)

就是list中的資料是排好順序的喇

效能: 插入和刪除都要**O(n)**的時間複雜度

以下是比較粗糙的實作(升冪排列), 只使用有單向連結串列, 所以省略了從尾部新增節點的操作

```
package idv.carl.datastructures.list;

/**
 * @author Carl Lu
 */
public class OrderedLinkedList {
    private LinkedNode head;
    private int size = 0;

    public void insertHead(int id) {
        LinkedNode newNode = new LinkedNode(id);

        // Need to find the correct location for insert operation

        LinkedNode previous = null;
        LinkedNode current = head;

        // Order the list ascending by id
        while (current != null && id > current.getId()) {
            previous = current;
            current = current.getNext();
        }

        if (previous == null) {
            head = newNode;
        } else {
            previous.setNext(newNode);
        }
```

```
            newNode.setNext(current);
            size++;
        }

        public LinkedNode removeHead() {
            if (size == 0) {
                return null;
            }

            LinkedNode tmp = head;
            head = head.getNext();
            size--;
            return tmp;
        }

        public LinkedNode find(int id) {
            LinkedNode node = head;
            while (node.getId() != id) {
                if (node.getNext() == null) {
                    return null;
                } else {
                    node = node.getNext();
                }
            }
            return node;
        }

        public LinkedNode remove(int id) {
            if (size == 0) {
                return null;
            }

            LinkedNode deleted = head;
            LinkedNode previous = head;

            // To search the deleted node and it's previous node
            while (deleted.getId() != id) {
                if (deleted.getNext() == null) {
                    return null;
                } else {
```

```
                previous = deleted;
                deleted = deleted.getNext();
            }
        }

        // Reset the relationship of nodes
        if (deleted.equals(head)) {
            head = head.getNext();
        } else {
            previous.setNext(deleted.getNext());
        }

        size--;
        return deleted;
    }

    public void displayList() {
        LinkedNode tmp = head;
        while (tmp != null) {
            System.out.println(tmp.toString());
            tmp = tmp.getNext();
        }
    }

    public int getSize() {
        return size;
    }
}
```

原始碼點我

使用有序連結串列來實作插入排序

思路: 把資料依序插入到有序連結串列, 然後再依次讀取出來, 這樣就排好惹.

效率: 比陣列插入法還高, 因爲在這種方式下, 資料的複製次數比較少一些, 每個節點只要2n次複製, 但在陣列中約需要n^2次的複製

```
package idv.carl.datastructures.list;

import java.util.Arrays;

/**
 * @author Carl Lu
 */
public class SortByOrderedLinkedList {

    public int[] sort(int[] data) {
        int[] result = new int[data.length];
        OrderedLinkedList orderedLinkedList = new OrderedLinkedL
ist();
        // Step1. Add the data into list sequentially
        Arrays.stream(data).forEach(d -> orderedLinkedList.inser
tHead(d));
        // Step2. Obtain all the data from list sequentially
        int index = 0;
        while (!orderedLinkedList.isEmpty()) {
            result[index++] = orderedLinkedList.removeHead().get
Id();
        }
        return result;
    }

}
```

原始碼點我

# 樹(Tree)

由邊和節點構成的資料結構, 節點通常就是儲存資料的實體.

```
     o  -> root node
    / \
   o   o
  / \   \
 o   o   o -> 這層都是葉節點
```

## 常見術語

根: 樹的頂端節點, 一棵樹只有一個根

邊: 節點到節點的連接

路徑: 沿著邊, 從一個節點走到另一個節點, 所經過的節點順序稱爲路徑

父節點, 子節點

節點, 葉節點(沒有子節點的節點)

度: 一個節點所包含的子節點數

子樹

層: 從根開始到指定節點的層數, 也稱爲高度或深度

走訪: 按照某個特定的順序存取節點

關鍵字: 節點物件域中的某個屬性, 用來識別節點物件

## 二元樹

### 定義: 樹中的每個節點, 最多只能有兩個子節點

對二元樹的理解

1. 二元樹與樹的區別爲:

a. 樹中節點的子節點樹沒有限制, 而二元樹中限制節點數爲不超過兩個

b. 樹的節點沒有左右之分, 但二元樹的節點是分左右的

2. 二元樹有五種基本型態

a. 空二元樹, 連根節點都沒有

b. 只有一個根節點的二元樹

c. 只有左樹

d. 只有右樹

e. 完全二元樹(complete binary tree): 若設二元樹的高度爲h, 除了第h層外, 其它各層的節點樹都達到最大個數,

第h層有葉節點, 並且葉節點都是從左到右依次排序, 此即完全二元樹

3. 滿二元樹(full binary tree): 除了葉節點外, 每個節點都有左右子節點, 且葉節點都處在最底層

4. 滿二元樹必爲完全二元樹, 完全二元樹不必然爲滿二元樹

5. 二元樹常被用作二元搜尋樹(Binary Search Tree, a.k.a BST), 二元排序樹, 二元堆(Binary Heap, a.k.a Heap Tree)

## 性質

性質1. 在二元樹的第i層上至多有2^(i-1)個節點

性質2. 深度爲k的二元樹至多有(2^k)-1個節點

性質3. 對任何一顆二元樹T, 若其終端節點數量爲n0, 度爲2的節點數爲n2, 則 n0=n2+1

性質4. 具有N個節點的完全二元樹的深度爲[log2以N爲底]+1

性質5. 如果對一顆有n個節點的完全二元樹的節點按層序編號, 即從第1層到第[log2以n爲底]+1層,

每層從左到右, 對任一節點i(1 <= i <= n)有:

a. 若i = 1, 則節點i是二元樹的根; 若i > 1, 則其父節點是[i/2]

b. 若2i > n, 則節點i無左子節點; 否則其左子節點是2i. 即該節點是葉節點

c. 若2i+1 > n, 則節點i無右子節點; 否則其右子節點是2i+1

d. 若i爲奇數且不小於1, 則Ki的左兄弟編號是i-1; 否則Ki無左兄弟

d. 若i爲偶數且小於n, 則Ki的右兄弟編號是i+1; 否則Ki無右兄弟

# 二元搜尋樹(Binary Search Tree, BST)

## 定義: 若一顆二元樹滿足以下兩點:

a. 左子節點的值小於節點的值

b. 右子節點的值大於節點的值

即可稱爲二元搜尋樹

```
    2             6
   / \           / \
  1   3         5   8      <-----像這樣, 這兩棵都是BST
               /   / \
              3   7   9
```

二元搜尋樹的查詢, 插入, 走訪, 查詢最大最小值和刪除操作

提示: 刪除節點有兩個子節點的時候, 要用它的中序後繼來代替該節點,

演算法爲: 找到被刪除節點的右子節點, 然後查詢此右子節點下的最後一個左子節點, 即此顆子樹的最小值節點, 這就是被刪除節點的中序後繼節點.

何謂前序, 中序, 後序?

以上圖右邊的三層樹爲例, 看1~2層的子樹:

前序: 6 -> 5 -> 8

中序: 5 -> 6 -> 8

後序: 5 -> 8 -> 6

若看整顆:

中序: 3 -> 5 -> 6 -> 7 -> 8 -> 9

所以若要刪除8, 要找它的“中序後繼節點(in-order successor)”的話, 就是9了

# 二元搜尋樹操作的效率

常見的時間複雜度爲: **O(logN),** 是以**2**爲底的

## 用陣列來表示樹

把樹的節點打上編號, 按順序放到陣列中. 如此一來, 查找節點就變成了查找相應的索引了. 這種方法不常用, 了解即可.

此種方式效率不高, 因爲不滿的節點還有刪除的節點, 在陣列中留下了多餘的空間, 這是一種記憶體上的浪費, 更糟糕的是要刪除節點時, 若要移動子樹的話, 就更浪費時間了.

關於BST的實作, 可以看這裡(點我)

以下是針對實作中的getSuccessor()的部分做圖示說明

初始化:

```
            6

          / \

         5   8 deletedNode = successor = successorParent

        /   / \

       3   7   10 current = deletedNode.getRight()

              / \

             9   11
```

```
// Find the node

            6

          / \

         5   8 successorParent = successor
```

```
          /   / \

         3   7   10 successor = current

                / \

               9   11

               ^---------- current = current.getLeft()




             6

            / \

           5   8 deletedNode

          /   / \

         3   7   10 successorParent = successor

                / \

               9   11

               ^------- successor = current

                        current = current.getLeft() = null


// Set related value

             6

            / \
```

```
            5   9 successor.setRight(deletedNode.getRight())

          /   / \

         3   7   10 successorParent.setLeft(successor.getRight()
)

               / \

              null 11
```

# 堆積樹(Heap Tree)

## 定義:

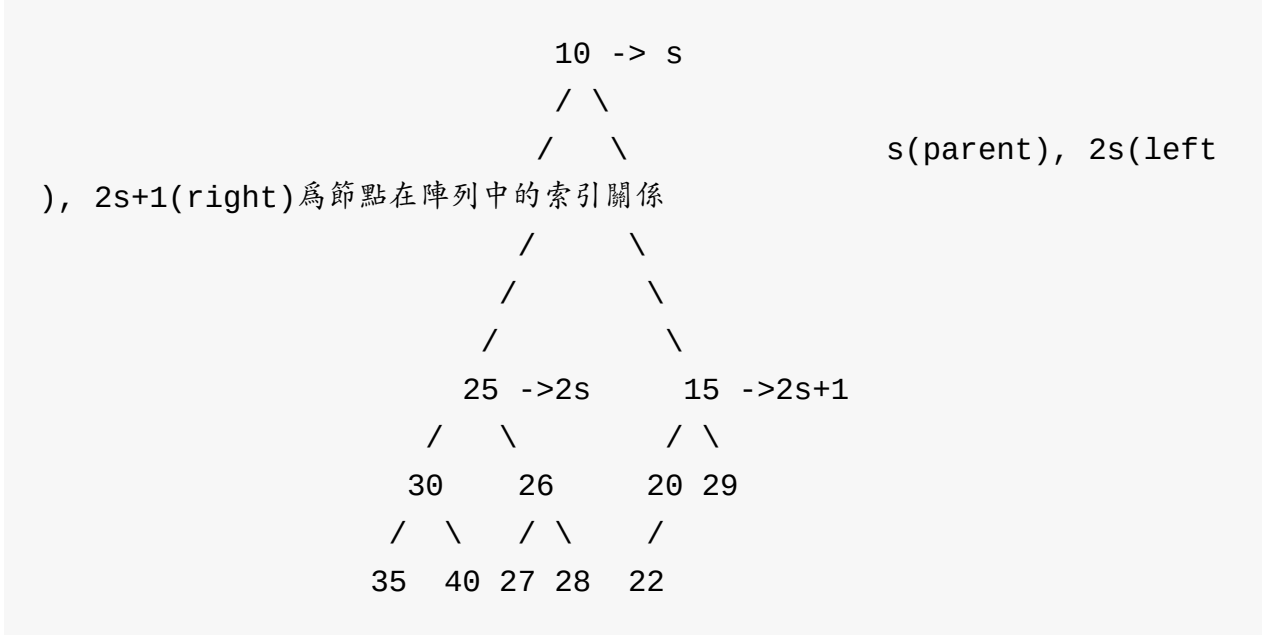
## 最小堆積(Min heap):父節點若小於子節點, 則稱之.

## 最大堆積(Max heap):父節點若大於子節點, 則稱之.

(然而, 同一層的子節點則無須理會其大小關係)

一個堆積樹必定爲完整二元樹(complete binary tree), 且通常會用陣列來實作.

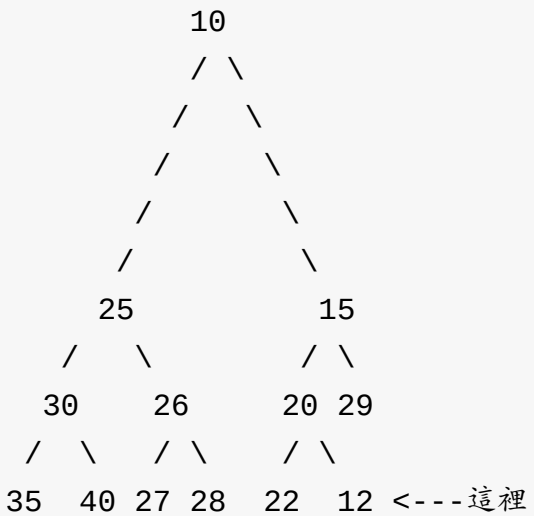
所以大概長得像這樣(Min heap):



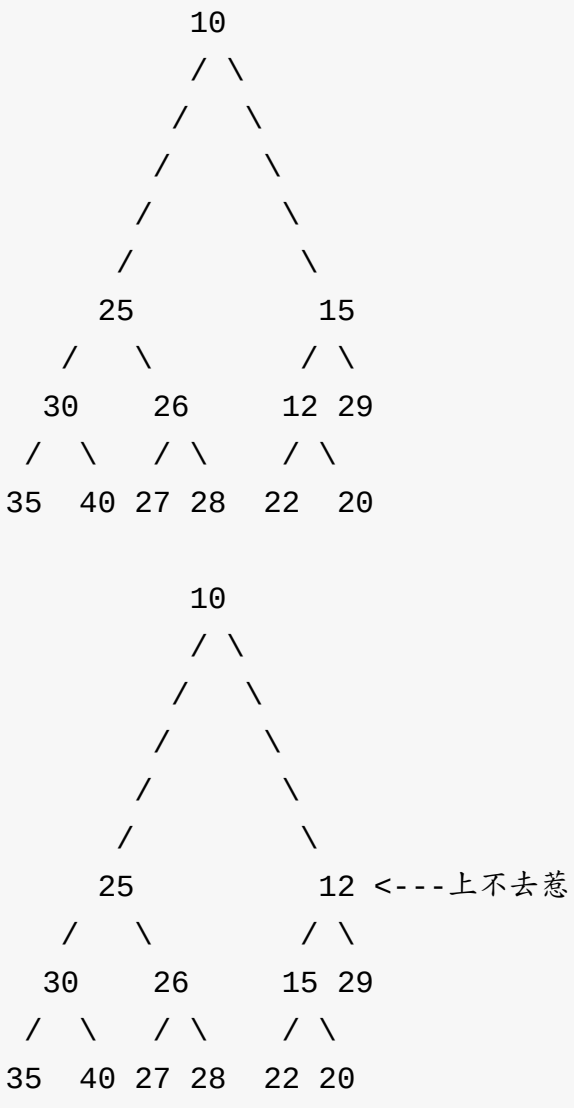

再來要知道怎麼建立堆積樹, 此處以上圖的min heap爲例:

假設此處以一維陣列來儲存這棵樹, 現在要加入一個元素(12)

1. 首先, 新加入的元素會被放到最後的葉節點




2. 然後新加入的節點會跟父節點做比對, 若小於父節點則往上升, 直至滿足堆積樹的條件爲止才停下來

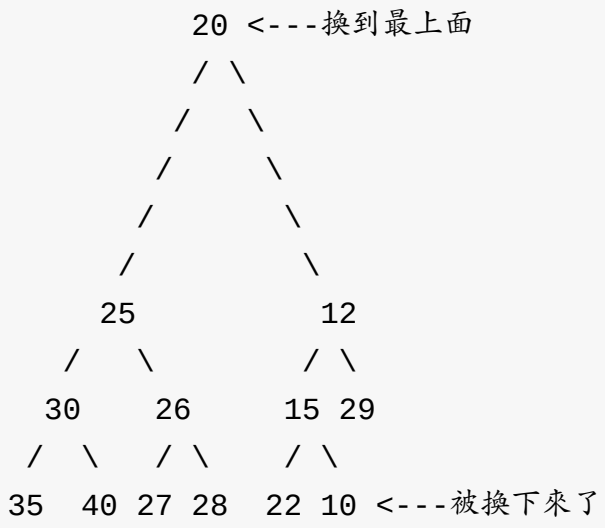


建立好堆積樹之後, 樹根一定是所有元素的最小值, 排序應用時:

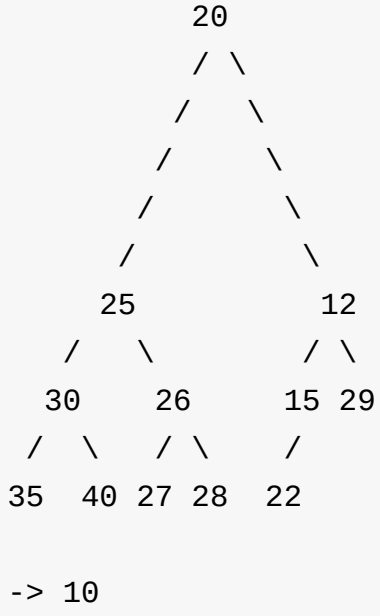
1. 將最小值取出
2. 調整樹爲最小堆積樹

不斷重複以上步驟就可達到排序的效果了, 其中, 取出最小值的做法是將樹根與最後一個葉節點做交換,然後切下葉節點, 並重新調整爲堆積樹, 在這段過程中, 找出父節點跟兩個子節點中較小的一個做交換:

1. 將最小值取出(跟最後一個葉節點交換)




2.將最小值取出(拿出來)



```
3. 開始調整直到變成堆積樹

                  12
                  / \
                 /   \
                /     \
               /       \
              /         \
             /           \
            25           15
           /   \         / \
          30    26      20 29
         /  \   / \     / \_________>它往下換兩次後就停住了
        35  40 27 28   22

        -> 10
```

因爲此處假設使用一維陣列來儲存堆積樹, 所以一直重複此步驟的話, 每次將葉節點跟最小值交換的動作就是把最小值放到陣列的後端, 所以到最後陣列就變成排序好的狀態了.

## 整理:

堆積樹建立的過程(insert): 氣泡排序(樹葉到樹根), O(logn)

調整過程: 氣泡排序(樹根到樹葉), O(logn)

取出最大/小值: 直接拿根節點, Θ(1)

堆積排序法(Heap Sort)又被稱爲改良的選擇排序法

・Best Case: O(nlogn)

・Worst Case: O(nlogn)

・Average Case: O(nlogn)

說明：

建立MaxHeap： O(n)

執行n-1次Delete Max：(n-1) × O(log n) = O( n log n)

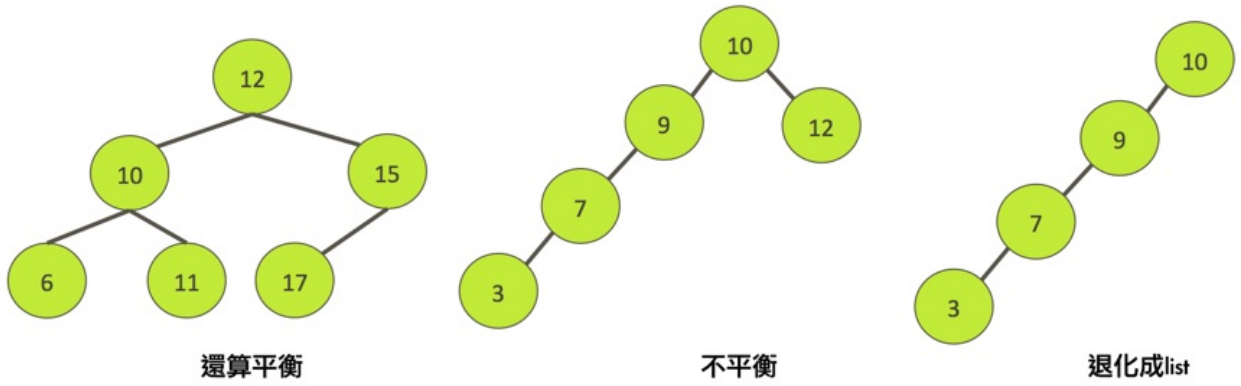
・合併兩個Heap Tree: 最優方法是把兩個heap tree首尾相連放在一個陣列中, 然後構造新的heap tree. 時間複雜度爲O(logn*logk), 其中n、k爲兩個heap tree的元素數目.

- 範例程式: 點我

# Red-Black Tree (紅黑樹)

樹的平衡: 所謂樹的平衡指的是, 樹中每個節點的左邊後代的數目應該與其右邊後代的數目大致相等(不用完全一樣多).

對於用隨機數構成的二元樹, 一般來說是大致平衡的, 但是對於有順序的資料, 就可能導致二元樹極度的不平衡了. 在最極端的情況下, 甚至會退化成**list**, 此時的時間複雜度會退化到O(n), 而不再是平衡樹的O(logn)了.




紅黑樹(R-B Tree): 紅黑樹是增加了某些特性的二元搜尋樹, 它可以保持樹的大致平衡. 主要的思路爲: 在插入或刪除節點的時候, 檢查是否破壞了樹的某些特徵, 若破壞了, 則進行糾正, 進而保持樹的平衡.

在JDK中, 以紅黑樹爲實作基礎的資料結構如: TreeSet, TreeMap以及最新的HashTable

紅黑樹的特徵:

1. 節點都有顏色(紅/黑)
2. 在插入和刪除的過程中, 要遵循保持這些顏色的不同排列的規則

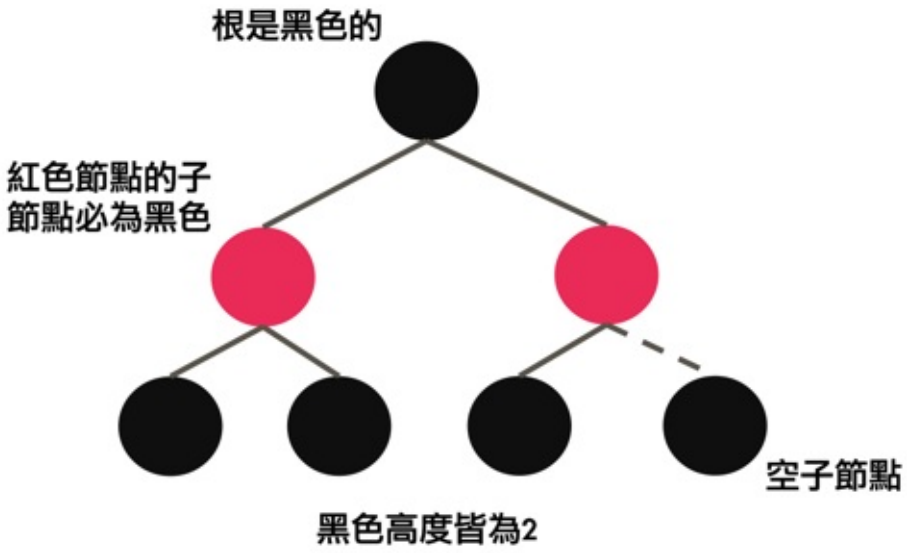
紅黑樹的規則(紅黑規則):

1. 每個節點不是紅就是黑的(廢話)
2. 根總是黑色的
3. 若節點是紅色的, 則其子節點必爲黑色, 反之不必爲眞(亦即若節點是黑色, 則其子節點可爲紅也可爲黑), 這條規則其實也是在說明就垂直方向來看, 紅色節點不可以相連
4. 每個空子節點都是黑色的: 所謂的空子節點指的是, 對非葉節點而言, 本可能有,

但實際沒有的那個子節點. 譬如一個節點只有右子節點, 那麼其空缺的左子節點就是空子節點

5. 從根節點到葉節點或空子節點的每條路徑(簡單路徑), 必須包含相同數目的黑色節點(這些黑色節點的數目也稱爲黑色高度)

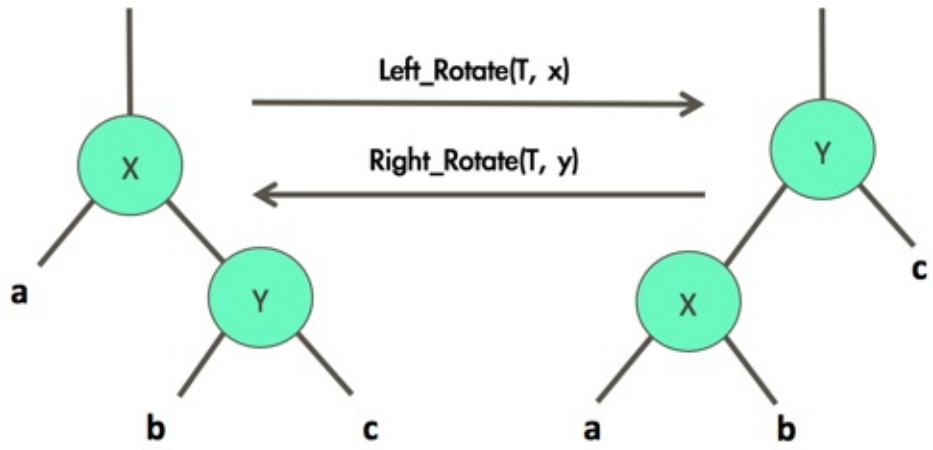
一個滿足紅黑樹規則的二元搜尋樹大概長得像這樣:




紅黑樹的修正手段:

1. 改變節點顏色
2. 執行旋轉操作

旋轉操作示意圖:




以下是紅黑樹的節點程式碼:

```
package idv.carl.datastructures.rbtree;
```

```
/**
 * @author Carl Lu
 */
public class RbNode {

    private int id;
    private int data;
    private boolean red = true;
    private RbNode parent;
    private RbNode left;
    private RbNode right;

    public RbNode(int id, int data) {
        super();
        this.id = id;
        this.data = data;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getData() {
        return data;
    }

    public void setData(int data) {
        this.data = data;
    }

    public boolean isRed() {
        return red;
    }

    public void setRed(boolean red) {
```

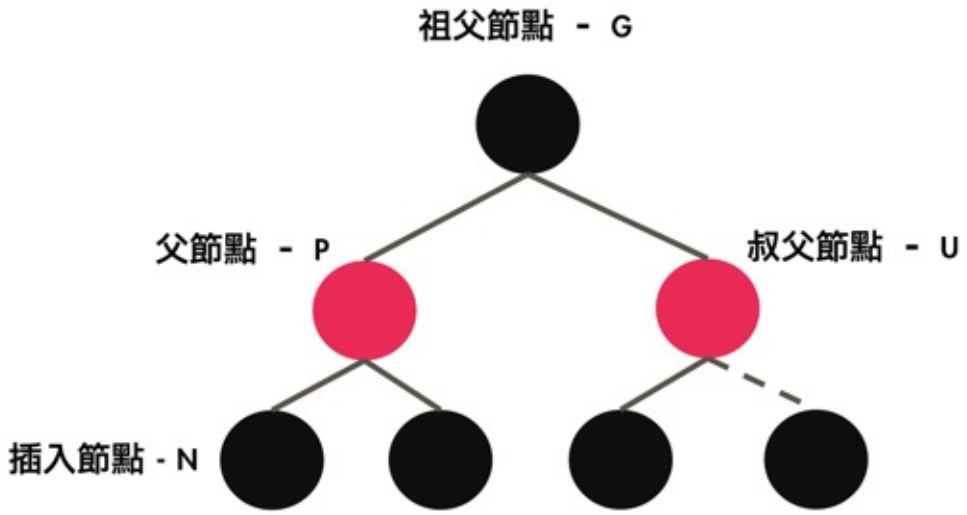
```
        this.red = red;
    }

    public RbNode getParent() {
        return parent;
    }

    public void setParent(RbNode parent) {
        this.parent = parent;
    }

    public RbNode getLeft() {
        return left;
    }

    public void setLeft(RbNode left) {
        this.left = left;
    }

    public RbNode getRight() {
        return right;
    }

    public void setRight(RbNode right) {
        this.right = right;
    }

    @Override
    public String toString() {
        return "RbNode{" + "id=" + id + ", data=" + data + ", re
d=" + red + '}';
    }
}
```

# RB Tree Insertion (紅黑樹的插入)

紅黑樹的插入演算法

紅黑樹的插入, 前面的部分跟二元搜尋樹一樣, 就是從根節點向下尋找節點要插入的位置, 找到後插入節點; 插入節點後, 多了這樣的一個操作: 檢查樹是否平衡, 如果不平衡, 就要做修復, 使樹重新變得平衡.

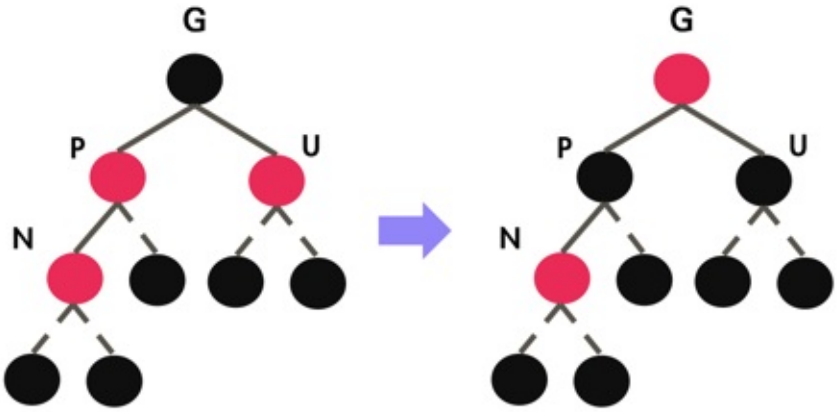
在開始討論插入演算法之前, 先釐清節點之間的關係與簡稱:



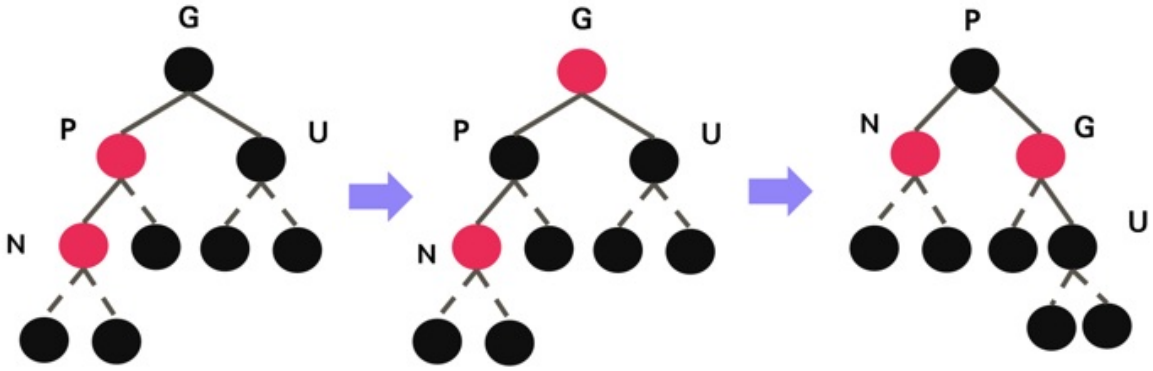

插入新的節點, 通常會設置這個節點爲紅色, 因爲這樣可以降低違反紅黑規則的機率, 而插入節點後, 會有以下幾點的情境出現:

1. 若插入的是根節點, 那麼違反規則**2**(根總是黑色的), 那就直接把節點改成黑色的
2. 若插入節點的P節點是黑色的, 即符合規則, 就什麼都不做

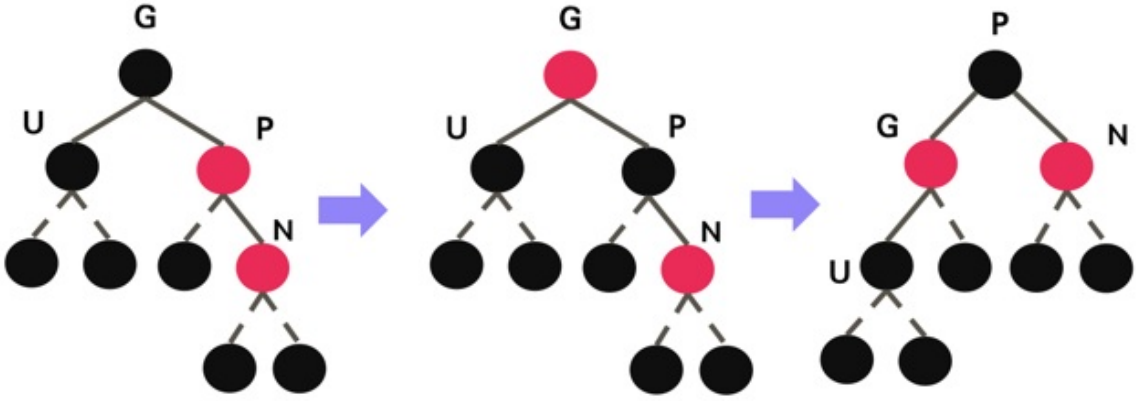
3. 若插入節點的P節點是紅色的, 且U節點也是紅色的, 那麼: 將G節點變紅, 而P與U節點變黑(遵守規則**5**), 然後設置G節點爲current node, 並且重新開始調整



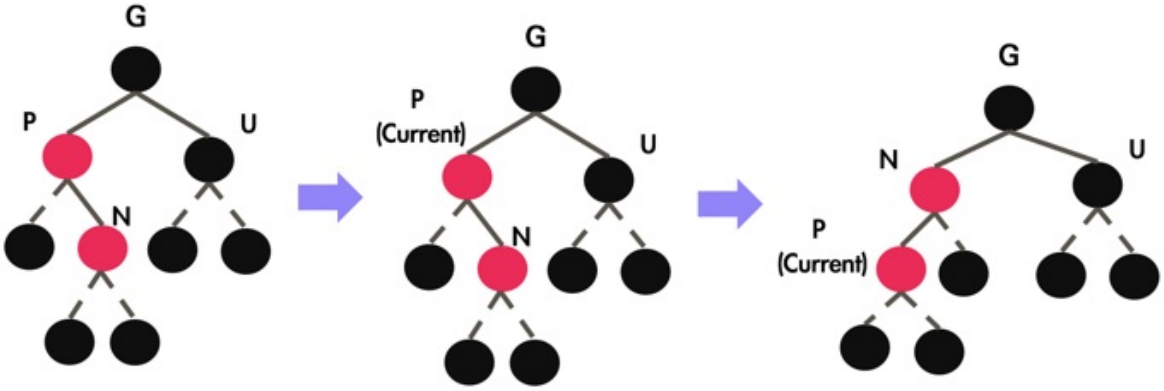

4. 若插入節點的P節點是紅色的, 而U節點是黑色或缺少, 且插入節點是其P節點的左子節點, 而P節點是G節點的左子節點, 那麼: 把P節點變成黑色, G節點變爲紅色, 然後對G節點做右旋的動作, 最後重新開始調整



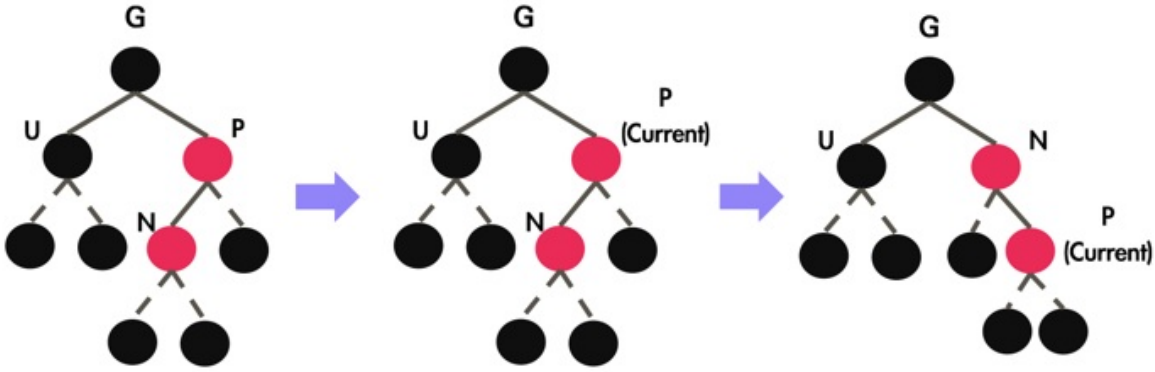

5. 若插入節點的P節點是紅色的, 而U節點是黑色或缺少, 且插入節點是其P節點的右子節點, 而P節點是G節點的右子節點, 那麼: 把P節點變成黑色, G節點變爲紅色, 然後對G節點做左旋的動作, 最後重新開始調整



6. 若插入節點的P節點是紅色的, 而U節點是黑色或缺少, 且插入節點是其P節點的右子節點, 而P節點是G節點的左子節點, 那麼: 把當前節點的P節點作爲新的當前節點, 對新的當前節點進行左旋, 並且重新開始調整




7. 若插入節點的P節點是紅色的, 而U節點是黑色或缺少, 且插入節點是其P節點的左子節點, 而P節點是G節點的右子節點, 那麼: 把當前節點的P節點作爲新的當前節點, 對新的當前節點進行右旋, 最後重新開始調整



# RB Tree Insertion Implementation

完整的原始碼點我

這邊是insert的部分:

```
public void insert(int id, int data) {
        // Create a new node
        RbNode newNode = new RbNode(id, data, true);

        if (root == null) {
            root = newNode;
        } else {
            // Find the insert location
            RbNode current = root;
            RbNode parent = null;

            while (true) {
                parent = current;
                if (id < current.getId()) {
                    current = current.getLeft();
                    // If no left
                    if (current == null) {
                        // Modify related node's attribute
                        parent.setLeft(newNode);
                        newNode.setParent(parent);
                        break;
                    }
                } else {
                    current = current.getRight();
                    // If no right
                    if (current == null) {
                        // Modify related node's attribute
                        parent.setRight(newNode);
                        newNode.setParent(parent);
                        break;
                    }
                }
            }
        }

        balanceAfterInsert(newNode);
    }
```

這邊是平衡的部分:

```
private void balanceAfterInsert(RbNode currentNode) {
        /*
         * Case 1: If the node is root, this will violate the ru
le
         * that the root should be black, so change the node to
black.
         */
        if (currentNode.getParent() == null) {
            currentNode.setRed(false);
            root = currentNode;
        } else
        /*
         * Case 2: If node P is black, do nothing since it alrea
dy match the rule.
         */
        if (!currentNode.getParent().isRed()) {
            // Do nothing.
        } else if (currentNode.getParent().isRed()) {
            RbNode gNode = currentNode.getParent().getParent();
            RbNode uNode = null;
            if (gNode != null) {
                uNode = (gNode.getLeft() == currentNode.getParen
t()) ? gNode.getRight() : gNode.getLeft();
            }
            /*
             * Case 3: If node P and node U are all red, modify
node G to red and modify both node P
             * and node U to black, then set the node G as curre
nt node and perform
             * balanceAfterInsert
             * operation on it.
             */
            if (uNode != null && uNode.isRed()) {
                gNode.setRed(true);
                uNode.setRed(false);
                currentNode.getParent().setRed(false);
                currentNode = gNode;
                balanceAfterInsert(currentNode);
            } else if (uNodeIsBlackOrNull(uNode)) {
```

```
                /*
                 * Case 4: If node P is red and node U is black,
 and the inserted node is the left
                 * of node P, and the node P is the left of node
 G, then modify the node P to black,
                 * and modify the node G to red. Finally, perfor
m right rotate on node G and
                 * balanceAfterInsert
                 * on current node.
                 */
                if (currentNode == currentNode.getParent().getLe
ft()
                        && (gNode != null && currentNode.getPare
nt() == gNode.getLeft())) {
                    currentNode.getParent().setRed(false);
                    gNode.setRed(true);
                    rightRotate(gNode);
                    balanceAfterInsert(currentNode);
                } else
                /*
                 * Case 5: If node P is red and node U is black,
 and the inserted node is the right
                 * of node P, and the node P is the right of nod
e G, then modify the node P to
                 * black, and modify the node G to red. Finally,
 perform left rotate on node G and
                 * balanceAfterInsert on current node.
                 */
                if (currentNode == currentNode.getParent().getRi
ght()
                        && (gNode != null && currentNode.getPare
nt() == gNode.getRight())) {
                    currentNode.getParent().setRed(false);
                    gNode.setRed(true);
                    leftRotate(gNode);
                    balanceAfterInsert(currentNode);
                } else
                /*
                 * Case 6: If node P is red and node U is black,
 and the inserted node is the right
```

```
                 * of node P, and the node P is the left of node
 G, then let the node P be the new
                 * current node. Finally, perform left rotate on
 the current node and
                 * balanceAfterInsert on it
                 */
                if (currentNode == currentNode.getParent().getRi
ght()
                        && (gNode != null && currentNode.getPare
nt() == gNode.getLeft())) {
                    RbNode oldParent = currentNode.getParent();
                    leftRotate(oldParent);
                    balanceAfterInsert(oldParent);
                } else
                /*
                 * Case 7:If node P is red and node U is black,
and the inserted node is the left of
                 * node P, and the node P is the right of node G
, then let the node P be the new
                 * current node. Finally perform right rotate on
 the current node and
                 * balanceAfterInsert on it
                 */
                if (currentNode == currentNode.getParent().getLe
ft()
                        && (gNode != null && currentNode.getPare
nt() == gNode.getRight())) {
                    RbNode oldParent = currentNode.getParent();
                    rightRotate(oldParent);
                    balanceAfterInsert(oldParent);
                }
            }
        }
    }

    private boolean uNodeIsBlackOrNull(RbNode uNode) {
        return (uNode == null || !uNode.isRed());
    }
```

這邊是左旋轉的部分:

```
private void leftRotate(RbNode pivot) {
        RbNode oldRight = pivot.getRight();
        RbNode leftOfOldRight = null;
        if (oldRight != null) {
            leftOfOldRight = oldRight.getLeft();
        }

        if (pivot.getParent() != null) {
            // Determine it's the left ot right
            boolean isLeft = (pivot.getParent().getLeft() == piv
ot);
            if (isLeft) {
                pivot.getParent().setLeft(oldRight);
            } else {
                pivot.getParent().setRight(oldRight);
            }

            if (oldRight != null) {
                oldRight.setParent(pivot.getParent());
            }
        } else {
            oldRight.setParent(null);
            oldRight.setRed(false);
            root = oldRight;
        }

        if (oldRight != null) {
            oldRight.setLeft(pivot);
        }
        pivot.setParent(oldRight);

        pivot.setRight(leftOfOldRight);
        if (leftOfOldRight != null) {
            leftOfOldRight.setParent(pivot);
        }
    }
```

這邊是右旋轉的部分:

```
private void rightRotate(RbNode pivot) {
        RbNode oldLeft = pivot.getLeft();
        RbNode rightOfOldLeft = null;
        if (oldLeft != null) {
            rightOfOldLeft = oldLeft.getRight();
        }
        if (pivot.getParent() != null) {
            // Determine it's the left ot right
            boolean isLeft = (pivot.getParent().getLeft() == piv
ot);
            if (isLeft) {
                pivot.getParent().setLeft(oldLeft);
            } else {
                pivot.getParent().setRight(oldLeft);
            }

            if (oldLeft != null) {
                oldLeft.setParent(pivot.getParent());
            }
        } else {
            oldLeft.setParent(null);
            oldLeft.setRed(false);
            root = oldLeft;
        }

        if (oldLeft != null) {
            oldLeft.setRight(pivot);
        }
        pivot.setParent(oldLeft);

        pivot.setLeft(rightOfOldLeft);
        if (rightOfOldLeft != null) {
            rightOfOldLeft.setParent(pivot);
        }
    }
```

# RB Tree Delete (紅黑樹的刪除)

紅黑樹節點的刪除: 在前面二元樹的筆記, 你可以看得出來刪除節點是很麻煩的, 那紅黑樹是一定會更複雜的, 因爲紅黑樹在做刪除動作的時候還需要保證刪除節點後的樹也是平衡的. 因此, 通常在實際的開發中, 多數情況下不用眞的去做紅黑樹節點的刪除, 而是採用其他的手段, 譬如: 僅標示此節點被刪除, 而不是眞的刪除, 這樣你的樹就不用動了, 故在業務邏輯處理的過程中, 判斷一下這個節點是不是被砍掉了, 若是的話就跳過即可. 這其實跟很多公司在實作刪除資料的時候, 不是眞的去砍資料, 而是把該筆資料在DB裡面用一個"Deleted"之類的欄位爲true來作爲已經刪除的意思.

紅黑樹的刪除演算法:

1. 如果刪除節點是葉子節點
    i. 如果刪除節點是紅色的, 那就直接砍了, 不做別的事, 因爲紅色不會影響黑色高度.
    ii. 如果刪除節點是黑色的, 那就要建立一個空節點來頂替被刪除的節點, 然後按照後面會提到的調整步驟進行調整
2. 如果刪除節點有一個子節點, 那就把後來要頂替被刪除節點的那個節點變成等頂替節點, 如果刪除節點爲黑色, 而且頂替節點也爲黑色, 那麼把頂替節點作爲當前節點, 然後按照後面的調整步驟進行調整
3. 若刪除節點有兩個子節點, 那麼, 找到其中序後繼節點, 把這兩個節點的資料進行交換, 不要複製顏色, 也不要改變其原有的父子等關係, 然後重新進行刪除的動作, 此處的刪除是指對交換資料後的中序後繼節點做刪除, 如原本刪除節點爲**node(id = m, value = i),** 中序後繼節點爲**node(id = n, value = j),** 則資料交換後變成**node(id = m, value = j), node(id = n, value = i),** 然後對**node(id=n)**進行刪除的動作. 由於其中序後繼節點只可能是葉子節點或著只有一個子節點, 這樣就會簡化成前面的兩種情形了.

刪除步驟中提到的調整演算法:

1. 若當前節點是紅色的, 則直接把當前節點變成黑色的即可
2. 若當前節點是黑色的, 且同時爲根節點, 則什麼都不用做
3. 若當前節點是黑色且兄弟節點爲紅色, 當前節點爲父節點的左子節點, 則把兄弟節點變成父節點的顏色, 把父節點變成紅色, 然後於父節點上執行左旋, 再重新

開始判斷

4. 若當前節點是黑色且兄弟節點爲紅色, 當前節點爲父節點的右子節點, 則把兄弟節點變成父節點的顏色, 把父節點變成紅色, 然後於父節點上執行右旋, 再重新開始判斷
5. 若當前節點是黑色且父節點和兄弟節點都是黑色, 兄弟節點的兩個子節點全爲黑色, 則把兄弟節點變紅, 然後把父節點當成新的當前節點, 再重新開始判斷
6. 若當前節點是黑色且兄弟節點爲黑色, 兄弟節點的兩個子節點全爲黑色, 但是父節點是紅色, 則把兄弟節點變紅色, 父節點變黑色
7. 若當前節點是黑色且兄弟節點爲黑色, 兄弟節點的左子節點爲紅色, 右子節點爲黑色, 且當前節點是父節點的左子節點, 則把兄弟節點變紅色, 兄弟左子節點變黑, 然後對兄弟節點執行右旋, 再重新開始判斷
8. 若當前節點是黑色且兄弟節點爲黑色, 兄弟節點的左子節點爲黑色, 右子節點爲紅色, 且當前節點是父節點的右子節點, 則把兄弟節點變紅色, 兄弟右子節點變黑, 然後對兄弟節點執行右旋, 再重新開始判斷
9. 若當前節點是黑色且兄弟節點爲黑色, 兄弟節點的右子節點爲紅色, 左子節點顏色不拘, 且當前節點是父節點的左子節點, 則把兄弟節點變成當前節點父節點的顏色, 並把當前節點的父節點變黑, 兄弟節點的右子節點變黑, 然後以當前節點的父節點爲支點執行左旋
10. 若當前節點是黑色且兄弟節點爲黑色, 兄弟節點的左子節點爲紅色, 左子節點顏色不拘, 且當前節點是父節點的右子節點, 則把兄弟節點變成當前節點父節點的顏色, 並把當前節點的父節點變黑, 兄弟節點的左子節點變黑, 然後以當前節點的父節點爲支點執行右旋

~~寫到這邊我腦海中第一個浮現的是這個畫面:~~

紅黑樹的效率:
紅黑樹的搜尋、插入與刪除的時間複雜度都是O(logn). 基本上跟二元樹是一樣的, 但實際上由於紅黑樹在插入和刪除的時候, 因爲需要保持平衡的關係, 所以會比二元樹慢一些. 另外, 紅黑樹需要額外的空間來記錄顏色的資訊

其它的平衡樹:
AVL Tree(發明者爲: Adelson-Velskii及Landis)是最早的一種平衡樹, 其要求節點左子樹和右子樹的高度相差不超過1.
當插入和刪除節點的時候, 都要進行平衡的動作, 也就是每次操作會掃描整棵樹兩次, 一次向下搜尋節點, 一次向上平衡整棵樹.
AVL Tree的效能基本上不如紅黑樹, 也不常用, 了解一下就好.

# RB Tree Delete Implementation

完整的原始碼點我

刪除的主要邏輯:

```
public boolean delete(int key) {
        // 1: Find the node you want to delete
        RbNode current = root;
        RbNode parent = root;

        // The substitution of the deleted node
        RbNode substitution = null;

        current = this.findOneNode(root, key);
        if (current == null) {
            return true;
        }
        parent = current.getParent();

        // 2: If the node has no child
        if ((current.getLeft() == null || isEmptyNode(current.ge
tLeft()))
                && (current.getRight() == null || isEmptyNode(cu
rrent.getRight()))) {
            hasNoChildren(parent, current, isLeft);
            if (!current.isRed()) {
                substitution = new RbNode(SUBSTITUTION_ID, SUBST
ITUTION_DATA, false);
                substitution.setParent(current.getParent());

                if (parent != null) {
                    if (isLeft) {
                        parent.setLeft(substitution);
                    } else {
                        parent.setRight(substitution);
                    }
                }
```

```
            }
            current.setParent(null);
        }
        // 3: If the node has only one child
        // 3.1: Only has left child
        else if (current.getRight() == null || isEmptyNode(curre
nt.getRight())) {
            oneLeftChild(parent, current, isLeft);
            if (!current.isRed() && current.getLeft().isRed()) {
                current.getLeft().setRed(false);
            } else if (!current.isRed() && !current.getLeft().is
Red()) {
                substitution = current.getLeft();
            }
        }
        // 3.2: Only has right child
        else if (current.getLeft() == null || isEmptyNode(curren
t.getLeft())) {
            oneRightChild(parent, current, isLeft);
            if (!current.isRed() && current.getRight().isRed())
{
                current.getRight().setRed(false);
            } else {
                substitution = current.getRight();
            }
        }
        // 4: If the node has two child
        else {
            // 4.1: Find the in-order successor
            RbNode successor = getSuccessor(current);

            /*
             * 4.2: Swap the successor and current node without
copying color and changing
             * relationship
             */
            RbNode tempNode = new RbNode(successor.getId(), succ
essor.getData(), successor.isRed());
            successor.setId(current.getId());
```

```
            successor.setData(current.getData());

            current.setId(tempNode.getId());
            current.setData(tempNode.getData());

            // 4.3: Execute delete operation again
            delete(successor.getId());
        }

        // After the delete operation, execute balance operation
        if (substitution != null) {
            balanceAfterDelete(substitution);
        }

        return true;
    }
```

在刪除過程中搜尋節點:

```
private RbNode findOneNode(RbNode node, int key) {
        if (node != null) {
            if (node.getId() == key) {
                return node;
            }

            RbNode tempNode = findOneNode(node.getLeft(), key);
            if (tempNode != null) {
                if (tempNode == tempNode.getParent().getLeft())
{
                    isLeft = true;
                } else {
                    isLeft = false;
                }
                return tempNode;
            }

            tempNode = findOneNode(node.getRight(), key);
            if (tempNode != null) {
                if (tempNode == tempNode.getParent().getLeft())
{
                    isLeft = true;
                } else {
                    isLeft = false;
                }
                return tempNode;
            }
        }
        return null;
    }
```

删除之後的平衡操作:

```
private void balanceAfterDelete(RbNode currentNode) {
        if (currentNode.isRed()) {
            currentNode.setRed(false);
        } else if (currentNode.getParent() == null) {
            currentNode.setRed(false);
        } else if (!currentNode.isRed()) {
```

```
            RbNode bNode = (currentNode == currentNode.getParent
().getLeft()) ?
            currentNode.getParent().getRight() : currentNode.get
Parent().getLeft();

            if (bNode.isRed() && currentNode == currentNode.getP
arent().getLeft()) {
                bNode.setRed(currentNode.getParent().isRed());
                currentNode.getParent().setRed(true);
                leftRotate(currentNode.getParent());
                balanceAfterDelete(currentNode);
            } else if (bNode.isRed() && currentNode == currentNo
de.getParent().getRight()) {
                bNode.setRed(currentNode.getParent().isRed());
                currentNode.getParent().setRed(true);
                rightRotate(currentNode.getParent());
                balanceAfterDelete(currentNode);
            } else if (bNode.isRed() && !currentNode.getParent()
.isRed()
                    && (bNode.getLeft() == null || !bNode.getLef
t().isRed())
                    && (bNode.getRight() == null || !bNode.getRi
ght().isRed())) {
                bNode.setRed(true);
                currentNode = currentNode.getParent();
                balanceAfterDelete(currentNode);
            } else if (!bNode.isRed() && currentNode.getParent()
.isRed()
                    && (bNode.getLeft() == null || !bNode.getLef
t().isRed())
                    && (bNode.getRight() == null || !bNode.getRi
ght().isRed())) {
                bNode.setRed(true);
                currentNode.getParent().setRed(false);
            } else if (!bNode.isRed() && (currentNode == current
Node.getParent().getLeft())
                    && (bNode.getLeft() != null && bNode.getLeft
().isRed())
                    && (bNode.getRight() == null || !bNode.getRi
ght().isRed())) {
```

```
                bNode.setRed(true);
                bNode.getLeft().setRed(false);
                rightRotate(bNode);
                balanceAfterDelete(currentNode);
            } else if (!bNode.isRed() && (currentNode == current
Node.getParent().getRight())
                    && (bNode.getRight() != null && bNode.getRig
ht().isRed())
                    && (bNode.getLeft() == null || !bNode.getLef
t().isRed())) {
                bNode.setRed(true);
                bNode.getRight().setRed(false);
                leftRotate(bNode);
                balanceAfterDelete(currentNode);
            } else if (!bNode.isRed() && (currentNode == current
Node.getParent().getLeft())
                    && (bNode.getRight() != null && bNode.getRig
ht().isRed())) {
                bNode.setRed(currentNode.getParent().isRed());
                currentNode.getParent().setRed(false);
                bNode.getRight().setRed(false);
                leftRotate(currentNode.getParent());
            } else if (!bNode.isRed() && (currentNode == current
Node.getParent().getRight())
                    && (bNode.getLeft() != null && bNode.getLeft
().isRed())) {
                bNode.setRed(currentNode.getParent().isRed());
                currentNode.getParent().setRed(false);
                bNode.getLeft().setRed(false);
                rightRotate(currentNode.getParent());
            }
        }
    }
```

# B-Tree(B樹)

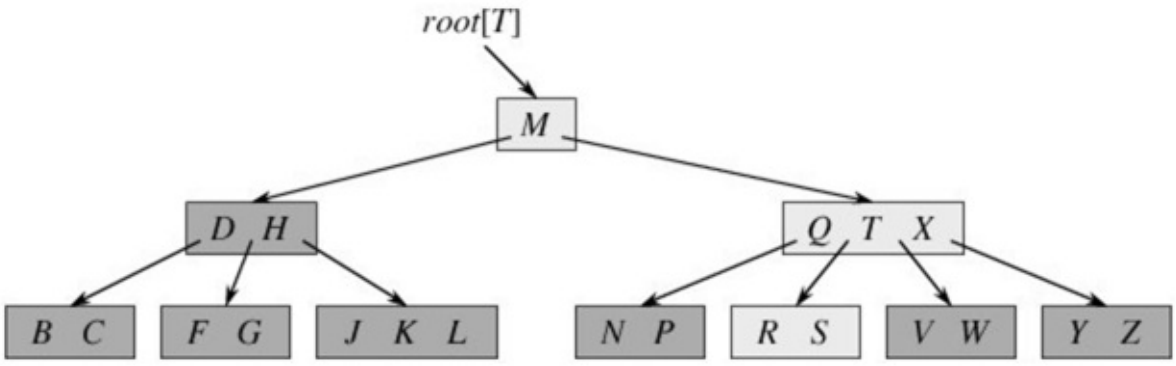
B-Tree: B是Balance, 平衡的意思, 是一種平衡的多路搜尋樹, 主要是用於磁碟等外部儲存的一種資料結構, 例如用於文件索引.

磁碟存取資料:

1. 磁碟存取資料的基本過程
    i. 根據磁柱號碼使磁頭移動到所需要的柱面上, 這一個過程被稱爲定位或是查找
    ii. 根據磁盤面號來確定指定盤面上的磁道
    iii. 盤面確定以後, 盤面開始旋轉, 將指定磁區的磁軌段移動至磁頭下
2. 磁碟讀取資料是以block爲基本單位的, 因此儘量將相關資訊存放在同一個磁區, 同一磁軌中, 以便在讀寫資料時儘量減少磁頭來回移動的次數, 避免過多的查找時間.
3. 當大量的資料儲存在磁碟中時, 如何高效地查詢磁碟中的資料, 就需要一種合理高效的資料結構了.

使用時機:
檔案系統或著是資料庫爲了增加搜尋的效率, 就可以採用此資料結構. 在資料量變大時, 若用線性搜尋檔案中的紀錄, 效率其實是不好的, 所以這時候就可以建立索引(index), 但資料量又再度增加時, index檔案也會變得很龐大. 爲此, 需要一個有效率的搜尋索引, 才能有效率的找到結果.

B-Tree的特性:




對於一顆m階(階指的是子節點的最大數目)的B-Tree, 有如下特性:

1. 根節點要就是葉子, 不然至少有兩個子節點
2. 每個節點最多含有m個節點(m ≥ 2)

3. 除了根節點和葉節點之外, 其他每個節點至少有[ceil(m/2)]個子節點
4. 所有的葉節點都出現在同一層上
5. 有s個子節點的非葉節點具有n = s - 1個關鍵字, 即s = n + 1
6. 每個非葉節點中包含有n個關鍵字訊息: (n, C0, K1, C1, K2, C2, ..., Kn, Cn), 其中:
    i. Ki (i = 1...n)爲關鍵字, 且關鍵字按順序升序排序K(i-1) < Ki
    ii. Ci爲指向子樹根的節點, 且指標C(i-1)指向子樹中所有節點的關鍵字均小於Ki, 但都大於K(i-1)
    iii. 關鍵字的個數n必須滿足: [ceil(m/2) - 1] ≤ n ≤ m-1

B-Tree的高度:

1. 就是B-Tree不包含葉節點的層數
2. 由於磁碟存取時的I/O次數, 與B樹的高度成正比, 高度越低, I/O次數也就越小, 因此理解如何計算B-Tree的高度是很有必要的
3. 假設若B-Tree總共包含N個關鍵字, 則此樹的葉節點可以有N+1個, 而所有的葉節點都在第K層, 可以得出:
    i. 因爲根至少有兩個子節點, 因此第二層至少有兩個節點
    ii. 除了根和葉之外, 其它節點至少有ceil(m/2)個子節點
    iii. 因此在第3層至少有2*ceil(m/2)個節點
    iv. 在第4層至少有2*(ceil(m/2)^2)個節點
    v. 在第k層至少有2*(ceil(m/2)^(k-2))個節點, 於是有: N+1 ≥ 2*(ceil(m/2)^(K-2)), 這就可以算出: K ≤ log(ceil(m/2))((N+1)/2)+2
    vi. 由於計算B-Tree高度是不包含葉節點的層數, 所以B-Tree的高度 ≤ log(ceil(m/2))((N+1)/2) + 1

# Algorithm

我覺得演算法眞的是CS最美的一塊領域, 其集合了抽象、邏輯與美學於一體, 也是考驗一個人能否在這個領域有所成就的指標之一.

# Sort(排序)

排序

# Bubble Sort (泡沫排序)

以後再補

```java
package idv.carl.sorting.bubblesort;

/**
 * @author Carl Lu
 */
public class BubbleSort {

    private static void swap(int[] input, int i, int j) {
        int tmp = input[i];
        input[i] = input[j];
        input[j] = tmp;
    }

    public static int[] sortDesc(int[] input) {
        for (int i = 0; i < input.length - 1; i++) {
            for (int j = 0; j < input.length - 1 - i; j++) {
                if (input[j] < input[j + 1]) {
                    swap(input, j, j + 1);
                }
            }
        }
        return input;
    }

    public static int[] sortAsc(int[] input) {
        for (int i = 0; i < input.length - 1; i++) {
            for (int j = 0; j < input.length - 1 - i; j++) {
                if (input[j] > input[j + 1]) {
                    swap(input, j, j + 1);
                }
            }
        }
        return input;
    }
}
```

原始碼點我

Unit test

```
package idv.carl.sorting.bubblesort;

import static org.junit.Assert.assertArrayEquals;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

/**
 * @author Carl Lu
 */
public class BubbleSortTest {

    private static int[] input;
    private static int[] expectedDesc = new int[] {45, 34, 26, 13
, 12, 9, 7, 3, 1, -1};
    private static int[] expectedAsc = new int[] {-1, 1, 3, 7, 9
, 12, 13, 26, 34, 45};

    @Before
    public void init() {
        input = new int[] {12, 45, 1, 3, -1, 34, 13, 7, 9, 26};
    }

    @After
    public void destroy() {
        input = null;
    }

    @Test
    public void testSortDesc() {
        assertArrayEquals(expectedDesc, BubbleSort.sortDesc(inpu
t));
    }

    @Test
    public void testSortAsc() {
        assertArrayEquals(expectedAsc, BubbleSort.sortAsc(input)
);
    }
```

```
  }
```

原始碼點我

# Selection Sort (選擇排序)

我懶得寫了, 直接上code, 有空再補內文.

```java
package idv.carl.sorting.selectionsort;

/**
 * @author Carl Lu
 */
public class SelectionSort {

    private static void swap(int[] input, int i, int j) {
        int tmp = input[i];
        input[i] = input[j];
        input[j] = tmp;
    }

    public static int[] sortDesc(int[] input) {
        for (int i = 0; i < input.length - 1; i++) {
            for (int j = i + 1; j < input.length; j++) {
                if (input[i] < input[j]) {
                    swap(input, i, j);
                }
            }
        }
        return input;
    }

    public static int[] sortAsc(int[] input) {
        for (int i = 0; i < input.length - 1; i++) {
            for (int j = i + 1; j < input.length; j++) {
                if (input[i] > input[j]) {
                    swap(input, i, j);
                }
            }
        }
        return input;
    }

}
```

原始碼點我

Unit test:

```
package idv.carl.sorting.selectionsort;

import static org.junit.Assert.assertArrayEquals;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

/**
 * @author Carl Lu
 */
public class SelectionSortTest {

    private static int[] input;
    private static int[] expectedDesc = new int[] {45, 34, 26, 13
, 12, 9, 7, 3, 1, -1};
    private static int[] expectedAsc = new int[] {-1, 1, 3, 7, 9
, 12, 13, 26, 34, 45};

    @Before
    public void init() {
        input = new int[] {12, 45, 1, 3, -1, 34, 13, 7, 9, 26};
    }

    @After
    public void destroy() {
        input = null;
    }

    @Test
    public void testSortDesc() {
        assertArrayEquals(expectedDesc, SelectionSort.sortDesc(i
nput));
    }

    @Test
    public void testSortAsc() {
        assertArrayEquals(expectedAsc, SelectionSort.sortAsc(inp
```

```
ut));
    }

}
```

原始碼點我

# Insertion Sort (插入排序)

以後再補說明

```
package idv.carl.sorting.insertionsort;

/**
 * @author Carl Lu
 */
public class InsertionSort {

    private static void swap(int[] input, int i, int j) {
        int tmp = input[i];
        input[i] = input[j];
        input[j] = tmp;
    }

    public static int[] sortDesc(int[] input) {
        for (int i = 0; i < input.length - 1; i++) {
            for (int j = i + 1; j > 0; j--) {
                if (input[j - 1] < input[j]) {
                    swap(input, j - 1, j);
                }
            }
        }
        return input;
    }

    public static int[] sortAsc(int[] input) {
        for (int i = 0; i < input.length - 1; i++) {
            for (int j = i + 1; j > 0; j--) {
                if (input[j - 1] > input[j]) {
                    swap(input, j - 1, j);
                }
            }
        }
        return input;
    }
}
```

原始碼點我

Unit test:

```
package idv.carl.sorting.insertionsort;

import static org.junit.Assert.assertArrayEquals;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

/**
 * @author Carl Lu
 */
public class InsertionSortTest {

    private static int[] input;
    private static int[] expectedDesc = new int[] {45, 34, 26, 13
, 12, 9, 7, 3, 1, -1};
    private static int[] expectedAsc = new int[] {-1, 1, 3, 7, 9
, 12, 13, 26, 34, 45};

    @Before
    public void init() {
        input = new int[] {12, 45, 1, 3, -1, 34, 13, 7, 9, 26};
    }

    @After
    public void destroy() {
        input = null;
    }

    @Test
    public void testSortDesc() {
        assertArrayEquals(expectedDesc, InsertionSort.sortDesc(i
nput));
    }

    @Test
    public void testSortAsc() {
        assertArrayEquals(expectedAsc, InsertionSort.sortAsc(inp
ut));
    }
```

```
  }
```

原始碼點我

# Merge Sort(合併排序)

思想: 採用**Divide and Conquer**的做法, 將資料序列分成兩個子序列, 排序每一半, 然後再把排序好的兩個子序列合併成爲一個有序的序列.

效率: Merge Sort的時間複雜度是**O(nlogn)**, 主要是複製(因爲此演算法通常都不是in-place sorting)跟比較會花比較多時間

```
package idv.carl.sorting;

import java.util.Arrays;

/**
 * @author Carl Lu
 */
public class MergeSort {

    public static void main(String args[]) {
        MergeSort mergeSort = new MergeSort();
        int[] input = new int[] {4, 8, 10, 1, 5, 9, 7};
        mergeSort.mergeSort(input);
        Arrays.stream(input).forEach(data -> System.out.print(da
ta + " "));
    }

    public void mergeSort(int[] data) {
        int[] temp = new int[data.length];
        doMergeSort(data, temp, 0, data.length - 1);
    }

    private void doMergeSort(int[] data, int[] temp, int from, i
nt to) {
        if (from >= to) {
            return;
        }
        // Step1. Calculate the bound index
        int mid = (from + to) >> 1;
        // Step2. Sort the left part
```

```
        doMergeSort(data, temp, from, mid);
        // Step3. Sort the right part
        doMergeSort(data, temp, mid + 1, to);
        // Step4. Merge the two parts
        merge(data, temp, from, mid + 1, to);
    }

    private void merge(int[] data, int[] temp, int from, int mid
, int to) {
        // The index for merged data in temp array
        int count = 0;
        // The min index of the left part
        int minLeft = from;
        // The max index of the left part
        int maxLeft = mid - 1;

        // Step1. Get values from left part and compare with the
 right part
        while (from <= maxLeft && mid <= to) {
            if (data[from] < data[mid]) {
                // If left < right, add the left value into temp
                temp[count++] = data[from++];
            } else {
                // If left > right, add the right value into temp

                temp[count++] = data[mid++];
            }
        }

        // Step2. Handle the remaining part:
        // 2.1 For the left part
        while (from <= maxLeft) {
            temp[count++] = data[from++];
        }
        // 2.2 For the right part
        while (mid <= to) {
            temp[count++] = data[mid++];
        }

        // Step3. Copy the final results from temp array into da
```

```
ta array
        for (int i = 0; i < (to - minLeft + 1); i++) {
            data[minLeft + i] = temp[i];
        }
    }

}
```

原始碼網址點我

# Quick Sork(快速排序)

思想:先根據樞紐值(**pivot**)將資料序列分成兩個子序列, 使左邊序列的所有值都小於**pivot**, 且右邊都大於**pivot**, 然後採用同樣的方法來對每個子序列進行快速排序, 最後得到排好順序的資料.

1. 快速排序的重點之一, 就在於選取合理的**pivot**, 也就是通過pivot來把整個資料序列分成兩個序列.
2. 目前常用的方式是"三資料項取中(a.k.a. **Balanced Quick Sort**)", 即對資料的第一個, 中間一個及最後一個位置的資料, 找到這三者的中間項. 譬如說: 第一個爲1, 中間爲19, 最後一個爲7, 故取三者的中間(1, 7, 19)的話, 即爲7.
3. 另一個重點, 就在於把資料分成兩個序列, 且要滿足條件(左邊小於pivot, 右邊大於pivot).
4. 時間複雜度爲: **O(nlogn),** 最差爲**O(n^2)** -> 在partition的部分, 因爲要將所有的元素都拿來跟pivot比過一次, 所以迭代所有元素的時間複雜度是O(n), 合併雖然會因爲實作而異, 但了不起就是O(n)了; 因此重點就會是在切割的次數, 畢竟切割次數會被pivot的選擇影響到, 最好的情況下, 就是每次都切割成相同大小的子陣列, 這樣就是切割logn次. 假如運氣太差了, 即每次取pivot都取到該數列的最大或最小值(碰到完全排好的資料然後你pivot又只抓最右邊或最左邊就會變這樣), 如此一來, 變成將原本的陣列切成0跟n-1(不包含pivot), 這樣就會有n次的遞迴呼叫, 故時間複雜度最佳爲O(nlogn), 最差爲O(n^2).
5. 空間複雜度: 最佳爲**O(nlogn),** 最差爲**O(n^2)**
   -> 由於每次都會把資料分成兩份子陣列, 因此會申請兩個新的子陣列記憶體空間, 對每個遞迴來說這部分的空間複雜度就是O(n), 而遞迴呼叫的最佳情況爲logn次, 最差爲n次, 所以空間複雜度的最佳爲O(nlogn), 最差爲O(n^2).
6. In-place版本: 這種做法可以讓我們不用爲子陣列申請新的記憶體空間, 所以每次遞迴都只需要O(1)的空間複雜度(for swap function), 而通常此版本的一個特色就是僅需要做**partition**的操作, 不必再有**merge**了, 因爲**partition**的時候也做完**merge**了.

以下的範例就是in-place的版本, pivot是直接取中間的元素.

```
package idv.carl.sorting.quicksort;
```

```
/**
 * @author Carl Lu
 */
public class QuickSortByMiddlePivot {

    public static void sort(int[] data) {
        if (data == null || data.length == 0) {
            return;
        }
        quickSort(data, 0, data.length - 1);
    }

    private static void quickSort(int[] data, int leftBound, int
 rightBound) {
        int left = leftBound;
        int right = rightBound;

        int pivot = data[leftBound + ( rightBound - leftBound )
/ 2];

        while (left < right) {
            while (data[left] < pivot) {
                left++;
            }

            while (data[right] > pivot) {
                right--;
            }

            if (left <= right) {
                swap(data, left, right);
                left++;
                right--;
            }
        }

        if (leftBound < right) {
            quickSort(data, leftBound, right);
        }
```

```
        if (left < rightBound) {
            quickSort(data, left, rightBound);
        }
    }

    private static void swap(int[] data, int left, int right) {
        int temp = data[left];
        data[left] = data[right];
        data[right] = temp;
    }

}
```

原始碼點我

測試程式:

```
package idv.carl.sorting.quicksort;

import org.junit.Test;

import java.util.Arrays;

import static org.junit.Assert.*;

/**
 * @author Carl Lu
 */
public class QuickSortByMiddlePivotTest {

    @Test
    public void testForNullInput() {
        int[] actual = null;
        QuickSortByMiddlePivot.sort(actual);
        assertNull(actual);
    }

    @Test
    public void testForEmptyInput() {
```

```
        int[] actual = new int[] {};
        QuickSortByMiddlePivot.sort(actual);
        assertEquals(0, actual.length);
    }

    @Test
    public void testForNormalCase() {
        int[] actual = new int[] {9, -32, 102, 4, -5, 8, 0, 7, 77
, 1, 30};
        int[] expected = new int[] {-32, -5, 0, 1, 4, 7, 8, 9, 30
, 77, 102};
        QuickSortByMiddlePivot.sort(actual);
        assertTrue(Arrays.equals(expected, actual));
    }

    @Test
    public void testForDuplicateElement() {
        int[] actual = new int[] {7, 7, 8, 8, 7, 7, 8, 8, 9, 9, -
1, -1, 2, 2, 0};
        int[] expected = new int[] {-1, -1, 0, 2, 2, 7, 7, 7, 7,
8, 8, 8, 8, 9, 9};
        QuickSortByMiddlePivot.sort(actual);
        assertTrue(Arrays.equals(expected, actual));
    }

}
```

原始碼點我

關於程式的解說, 可以看這個commit.

# Merge Sort v.s. Quick Sort

- 兩者其實非常相似, 都是把資料分成兩邊, 直到不能再分了, 才把資料合起來. 不過quick sort最大的特色就是會有**partition**的這個動作, 講白了就是把數字分好大小後再繼續往下分, 所以這樣循環下去分到最小後, 也表示完成了排序的動作了.
- 在大部分的worst case下, merge sort是優於quick sort的, 再加上merge sort的worst case跟quick sort的best case之時間複雜度是一樣的, 這樣看來似乎是merge sort比較快(理論上).
- 可是實際上來說, 如果兩者都用遞迴的方式去實作的話, quick sort的method call爲N, 那merge sort就會是2N-1(註1), 即merge sort多花了一倍的method call. 如果用迴圈來做, merge sort會花比較多時間在記憶體上面, 因爲它不是in-place sorting.
- 不過merge sort有一個很好的特性: 它是穩定的(**stable sorting**), 穩定的意思是說, 排序前與排序後, 擁有相同key值的兩個資料, 彼此之間的順序是一樣的.
- 最後, 這兩者都是Divide and Conquer的做法, 只是**quick sort**爲先苦後樂(遞迴之前的partition比較麻煩, 遞迴完後就沒事了); 而**merge sort**是先樂後苦(進入遞迴之前都沒事, 但是遞迴之後的合併動作就累了).

註1: 參閱原始碼, 注意merge function的step1~step2, 這是第一次的N, 即排序當前分割後的結果, 而step3, 把排序好的結果放回原本的array, 也會有一次N.

# Search

# Binary Search

一般來說, binary search是用來在已排序好的集合中搜尋用的方法, 以下是在一個排序好的array中找出指定值的index之做法, 分別爲遞迴版跟迴圈版, 時間複雜度爲O(logn):

```
package idv.carl.leetcode.algorithms.easy.binarysearchinarray;

/**
 * @author Carl Lu
 */
public class BinarySearch {

    public static int searchRecursively(int[] input, int key, int
 from, int to) {
        if (from > to) {
            return -1;
        }

        /*
         * int mid = (from + to) >> 1;
         *
         * It also works, however,
         * except when (from + to) produces int overflow.
         */
        int mid = from + ((to - from) >> 1);

        if (input[mid] > key) {
            return searchRecursively(input, key, from, mid - 1);
        } else if (input[mid] < key) {
            return searchRecursively(input, key, mid + 1, to);
        } else {
            return mid;
        }
    }

    public static int searchIteratively(int[] input, int key) {
```

```
        int from = 0;
        int to = input.length - 1;

        while (from <= to) {
            int mid = from + ((to - from) >> 1);
            if (input[mid] > key) {
                to = mid - 1;
            } else if (input[mid] < key) {
                from = mid + 1;
            } else {
                return mid;
            }
        }

        return -1;
    }

}
```

原始碼點我

測試程式:

```
package idv.carl.leetcode.algorithms.easy.binarysearchinarray;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

/**
 * @author Carl Lu
 */
public class BinarySearchTest {

    private static int[] input = new int[] {1, 2, 4, 5, 6, 7, 8,
9, 10, 12};

    @Test
    public void testSearchRecursively() {
        assertEquals(3, BinarySearch.searchRecursively(input, 5,
0, input.length));
    }

    @Test
    public void testSearchRecursivelyIfNoMatching() {
        assertEquals(-1, BinarySearch.searchRecursively(input, 3
, 0, input.length));
    }

    @Test
    public void testSearchIteratively() {
        assertEquals(3, BinarySearch.searchIteratively(input, 5)
);
    }

    @Test
    public void testSearchIterativelyIfNoMatching() {
        assertEquals(-1, BinarySearch.searchIteratively(input, 3
));
    }
}
```

原始碼點我

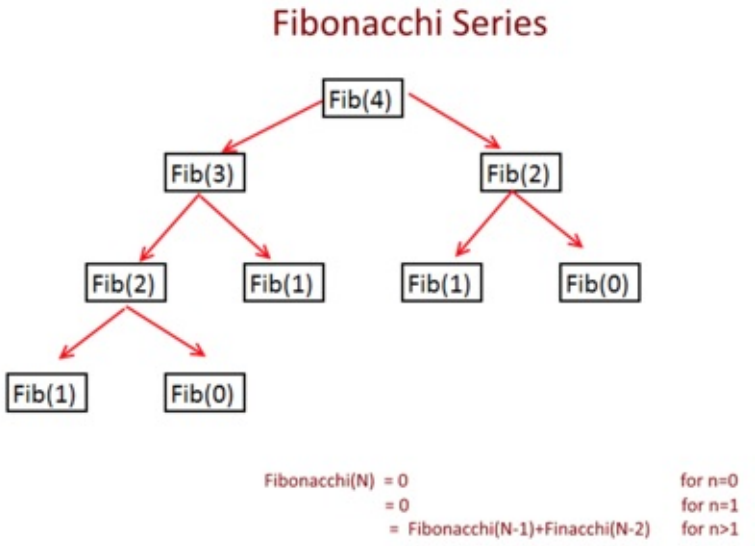
遞迴只應天上有, 凡人應當用迴圈 (嘆)

# Dynamic Programming (動態規劃)

要確認一個問題是否可以用動態規劃去解, 首先要確認是否有以下兩點特性:

1. 重疊子問題 **(Overlapping Sub-problems)**
2. 最佳子結構 **(Optimal Substructure)**

重疊子問題: 即出現需要重複解同樣的子問題的情境. 在遞迴中, 我們每次都要去重解這些子問題, 不過在動態規劃中, 我們只需要去解一次並且將各個子問題的解儲存起來以供將來使用即可. 比較明顯的案例大概就是Fibonacci數列了.




最佳子結構: 如果一個問題可以用與子問題相同的解法來解的話, 我們稱此問題具有最佳子結構的特性.

綜合上述兩點, 可知Fibonacci數列是可以透過動態規劃來求解的.

動態規劃的方法:

1. Bottom-Up
2. Top-Down

Bottom-Up: 假設要解的問題之輸入爲N, 那麼就從最小的可能輸入開始解並且儲存其結果, 這樣之後要求比較大的解的時候就可以用之前儲存的結果來求值.

# Fibonacci Series (斐波那契數列)

關於Fibonacci series的定義如下:

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$ (n≧2)

其在數學上是以遞迴的方式定義的.

以下就簡單列出遞迴解法跟動態規劃解法

## 遞迴:

```
    /*
     * Recursive solution.
     *
     * This solution cannot solve big input.
     * You can found that it will go very slowly after n ≥ 35
     *
     * Time Complexity: O(2^n)
     */
    public static long findFibonacci(int n) {
        long result = 0;

        for (int i = 1; i <= n; i++) {
            if (n <= 1) {
                result = n;
            } else {
                result = findFibonacci(n - 1) + findFibonacci(n
- 2);
            }
        }

        return result;
    }
```

Python版本:

```
def fibo(n):

    output = []
    a = 1
    b = 1

    for i in range(n):
        output.append(a)
        a, b = b, a + b

    return output

fibo(10)
```

原始碼點我

## 動態規劃:

```
    /*
     * Dynamic Programming – Memoization
     * (Bottom-Up Approach)
     *
     * Store the sub-problems result so that you don’t have to c
alculate again.
     * So first check if solution is already available,
     * if yes then use it else calculate and store it for future
.
     *
     * Time Complexity: O(n) , Space Complexity : O(n)
     */
    public static long findFibonacci(int n) {
        long fib[] = new long[n + 1];

        fib[0] = 0l;
        fib[1] = 1l;

        for (int i = 0; i <= n; i++) {
            if (i > 1) {
                fib[i] = fib[i - 1] + fib[i - 2];
            }
        }

        return fib[n];
    }
```

Python版本:

```
def fibonacci(n):

    a = 1
    b = 1

    for i in range(n):
        yield a
        a, b = b, a + b

for num in fibonacci(10):
    print num
```

原始碼點我

測試程式:

```
package idv.carl.leetcode.algorithms.easy.fibonacci;

import static org.junit.Assert.assertEquals;

import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

/**
 * @author Carl Lu
 */
public class FibonacciTest {

    @Rule
    public Timeout timeout = Timeout.seconds(3);

    @Test
    public void testForRecursiveWay() {
        assertEquals(55, FibonacciRecursive.findFibonacci(10));
    }

    @Test
    public void testForDynamicProgrammingWay() {
```

```
        assertEquals(55, FibonacciDynamicProgramming.findFibonac
ci(10));
    }

    /*
     * This will failed.
     */
    @Test
    public void testTimeoutForRecursiveWay() {
        assertEquals(12586269025L, FibonacciRecursive.findFibona
cci(50));
    }

    @Test
    public void testTimeoutForDynamicProgrammingWay() {
        assertEquals(12586269025L, FibonacciDynamicProgramming.f
indFibonacci(50));
    }

    @Test
    public void testTimeoutForDynamicProgrammingWay2() {
        assertEquals(7540113804746346429L, FibonacciDynamicProgr
amming.findFibonacci(92));
    }

}
```

原始碼點我

# Find Longest Common Suffix

定義: 找出兩個字串中最長的共同suffix

情境:

- String 1: Cornfield, String 2: outfield -> LCS: field
- String1: Manhours, String 2: manhole -> LCS: NULL (anho不是suffix)

這種問題基本上就是要用dynamic programming去解的, 如果以表格來分析的話大概就會長得像下面這張圖:

Longest Common Suffix

| | i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|
| j | | C | o | r | n | f | i | e | l | d |
| 0 | o | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | u | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | t | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | f | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 4 | i | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 |
| 5 | e | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 |
| 6 | l | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 |
| 7 | d | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 |

以下是程式的部分:

```
package idv.carl.leetcode.algorithms.medium.longestcommonsuffix;

/**
 * @author Carl Lu
 */
public class FindLongestCommonSuffix {

    /*
     * This is solved by dynamic programming.
     */
    public static String findLongestCommonSuffix(String str1, St
ring str2) {
```

```
        StringBuilder result = new StringBuilder();

        if (isValidString(str1) && isValidString(str2)) {
            int[][] num = new int[str1.length()][str2.length()];
            int maxLength = 0;
            int lastSubStringBegin = 0;

            for (int i = 0; i < str1.length(); i++) {
                for (int j = 0; j < str2.length(); j++) {
                    // If the chars matched
                    if (str1.charAt(i) == str2.charAt(j)) {
                        if ((i == 0) || (j == 0)) {
                            num[i][j] = 1;
                        } else {
                            num[i][j] = 1 + num[i - 1][j - 1];
                        }

                        if (num[i][j] > maxLength) {
                            // Reset the max length if the new o
nt is grater than the old one
                            maxLength = num[i][j];
                            int currentSubStringBegin = i - num[
i][j] + 1;
                            /*
                             * If the new detected substring is
derived from the last result,
                             * just append the char to the last
result
                             */
                            if (lastSubStringBegin == currentSub
StringBegin) {
                                result.append(str1.charAt(i));
                            } else {
                                /*
                                 * However, if the new detected
substring is derived from the new
                                 * start point, the result shoul
d be updated
                                 */
                                lastSubStringBegin = currentSubS
```

```
tringBegin;
                                result = new StringBuilder();
                                result.append(str1.substring(las
tSubStringBegin, i + 1));
                            }
                        }
                    }
                }
            }

            String resultStr = result.toString();
            if (resultStr.length() == 0 || !isSuffix(resultStr,
str1, str2)) {
                return "NULL";
            } else {
                return resultStr;
            }

        } else {
            return "NULL";
        }
    }

    private static boolean isValidString(String input) {
        return !(input == null || input.isEmpty());
    }

    // To ensure the result is the common suffix
    private static boolean isSuffix(String result, String str1,
String str2) {
        boolean isSuffix = false;
        int lengthOfResult = result.length();

        String str1Sub = str1.substring(str1.length() - lengthOf
Result);
        String str2Sub = str2.substring(str2.length() - lengthOf
Result);

        if (str1Sub.equals(str2Sub)) {
            isSuffix = true;
```

```
            }

            return isSuffix;
        }
    }
```

原始碼點我

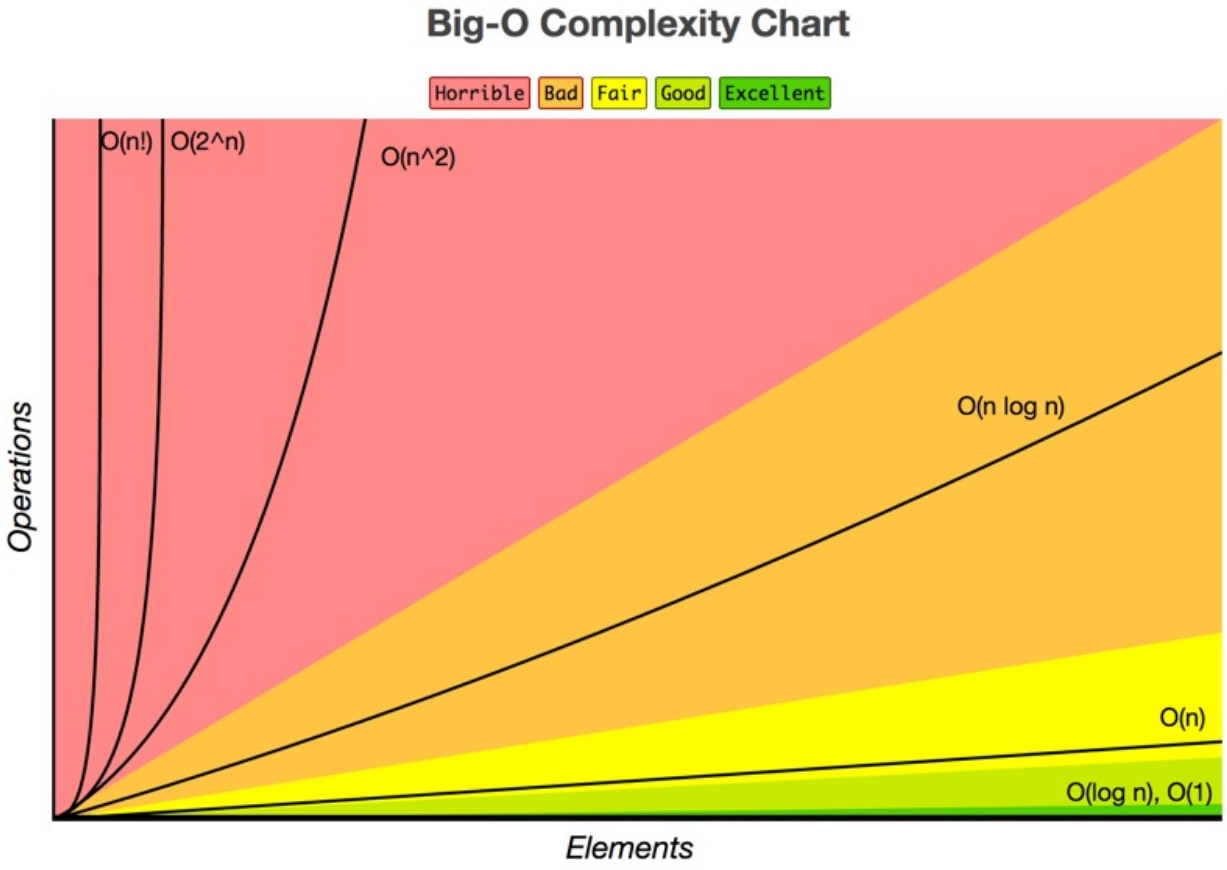
測試的部分:

```
package idv.carl.leetcode.algorithms.medium.longestcommonsuffix;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

/**
 * @author Carl Lu
 */
public class FindLongestCommonSuffixTest {

    @Test
    public void testForNormalCase() {
        assertEquals("field", FindLongestCommonSuffix.findLonges
tCommonSuffix("Cornfield", "outfield"));
    }

    @Test
    public void testForNullCase() {
        assertEquals("NULL", FindLongestCommonSuffix.findLongest
CommonSuffix("Manhours", "manhole"));
    }

    @Test
    public void testForInvalidInputCase() {
        assertEquals("NULL", FindLongestCommonSuffix.findLongest
CommonSuffix(null, "outfield"));
        assertEquals("NULL", FindLongestCommonSuffix.findLongest
CommonSuffix(null, null));
        assertEquals("NULL", FindLongestCommonSuffix.findLongest
CommonSuffix("", ""));
        assertEquals("NULL", FindLongestCommonSuffix.findLongest
CommonSuffix("", null));
    }

}
```

原始碼點我

# Time Complexity Cheat Sheet

## Big-O Complexity Chart



# Common Data Structure Operations

| Data Structure | Time Complexity | | | | | | | | Space Complexity |
|---|---|---|---|---|---|---|---|---|---|
| | Average | | | | Worst | | | | Worst |
| | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion | |
| Array | Θ(1) | Θ(n) | Θ(n) | Θ(n) | O(1) | O(n) | O(n) | O(n) | O(n) |
| Stack | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Queue | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Singly-Linked List | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Doubly-Linked List | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Skip List | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n log(n)) |
| Hash Table | N/A | Θ(1) | Θ(1) | Θ(1) | N/A | O(n) | O(n) | O(n) | O(n) |
| Binary Search Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |
| Cartesian Tree | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | O(n) | O(n) | O(n) | O(n) |
| B-Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Red-Black Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Splay Tree | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| AVL Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| KD Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |

# Array Sorting Algorithms

| Algorithm | Time Complexity | | | Space Complexity |
|---|---|---|---|---|
| | Best | Average | Worst | Worst |
| Quicksort | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(log(n)) |
| Mergesort | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(n) |
| Timsort | Ω(n) | Θ(n log(n)) | O(n log(n)) | O(n) |
| Heapsort | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(1) |
| Bubble Sort | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| Insertion Sort | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| Selection Sort | Ω(n^2) | Θ(n^2) | O(n^2) | O(1) |
| Tree Sort | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(n) |
| Shell Sort | Ω(n log(n)) | Θ(n(log(n))^2) | O(n(log(n))^2) | O(1) |
| Bucket Sort | Ω(n+k) | Θ(n+k) | O(n^2) | O(n) |
| Radix Sort | Ω(nk) | Θ(nk) | O(nk) | O(n+k) |
| Counting Sort | Ω(n+k) | Θ(n+k) | O(n+k) | O(k) |
| Cubesort | Ω(n) | Θ(n log(n)) | O(n log(n)) | O(n) |

Those resources are refered from here.