堆排序

最差时间复杂度
$O(n \log n)$
最优时间复杂度
$O(n \log n)$[1]
平均时间复杂度
$\Theta(n \log n)$
最差空间复杂度
$O(n)$ total, $O(1)$ auxiliary
堆排序（Heapsort）是指利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。

堆节点的访问

通常堆是通过一维数组来实现的。在数组起始位置为0的情形中：

- 父节点i的左子节点在位置(2*i+1);
- 父节点i的右子节点在位置(2*i+2);
- 子节点i的父节点在位置floor((i−1)/2);

堆的操作

在堆的数据结构中，堆中的最大值总是位于根节点。堆中定义以下几种操作：

- 最大堆调整（Max_Heapify）：将堆的末端子节点作调整，使得子节点永远小于父节点
- 创建最大堆（Build_Max_Heap）：将堆所有数据重新排序
- 堆排序（HeapSort）：移除位在第一个数据的根节点，并做最大堆调整的递归运算

实现示例

C语言

```c
#include <stdio.h>
#include <stdlib.h>

void swap(int* a, int* b) {
```

```
        int temp = *b;
        *b = *a;
        *a = temp;
}

void max_heapify(int arr[], int start, int end) {
        //建立父節點指標和子節點指標
        int dad = start;
        int son = dad * 2 + 1;
        while (son < end) { //若子節點指標在範圍內才做比較
                if (son + 1 < end && arr[son] < arr[son + 1]) //先比較兩個子節點大小，選擇最大的
                        son++;
                if (arr[dad] > arr[son]) //如果父節點大於子節點代表調整完畢，直接跳出函數
                        return;
                else { //否則交換父子內容再繼續子節點和孫節點比較
                        swap(&arr[dad], &arr[son]);
                        dad = son;
                        son = dad * 2 + 1;
                }
        }
}

void heap_sort(int arr[], int len) {
        int i;
        //初始化，i從最後一個父節點開始調整
        for (i = len / 2 - 1; i >= 0; i--)
                max_heapify(arr, i, len);
        //先將第一個元素和已排好元素前一位做交換，再從新調整，直到排序完畢
        for (i = len - 1; i > 0; i--) {
                swap(&arr[0], &arr[i]);
                max_heapify(arr, 0, i);
        }
}

int main() {
        int arr[] = { 3, 5, 3, 0, 8, 6, 1, 5, 8, 6, 2, 4, 9, 4, 7, 0, 1, 8, 9, 7, 3, 1, 2, 5, 9, 7, 4, 0, 2, 6 };
        int len = (int) sizeof(arr) / sizeof(*arr);
```

```
        heap_sort(arr, len);
        int i;
        for (i = 0; i < len; i++)
                printf("%d ", arr[i]);
        printf("\n");
        return 0;
}
```

C++

```
#include <iostream>
#include <algorithm>
using namespace std;

void max_heapify(int arr[], int start, int end) {
        //建立父節點指標和子節點指標
        int dad = start;
        int son = dad * 2 + 1;
        while (son < end) { //若子節點指標在範圍內才做比較
                if (son + 1 < end && arr[son] < arr[son + 1]) //先比較兩個子節點大小，選擇最大的
                        son++;
                if (arr[dad] > arr[son]) //如果父節點大於子節點代表調整完畢，直接跳出函數
                        return;
                else { //否則交換父子內容再繼續子節點和孫節點比較
                        swap(arr[dad], arr[son]);
                        dad = son;
                        son = dad * 2 + 1;
                }
        }
}

void heap_sort(int arr[], int len) {
        //初始化，i從最後一個父節點開始調整
        for (int i = len / 2 - 1; i >= 0; i--)
                max_heapify(arr, i, len);
        //先將第一個元素和已经排好的元素前一位做交換，再從新調整(刚调整的元素之前的元素)，直到排序完畢
```

```
        for (int i = len - 1; i > 0; i--) {
                swap(arr[0], arr[i]);
                max_heapify(arr, 0, i);
        }
}

int main() {
        int arr[] = { 3, 5, 3, 0, 8, 6, 1, 5, 8, 6, 2, 4, 9, 4, 7, 0, 1, 8, 9, 7, 3, 1, 2, 5, 9,
7, 4, 0, 2, 6 };
        int len = (int) sizeof(arr) / sizeof(*arr);
        heap_sort(arr, len);
        for (int i = 0; i < len; i++)
                cout << arr[i] << ' ';
        cout << endl;
        return 0;
}
```

Java

```
public class HeapSort {
        private static int[] sort = new int[]{1,0,10,20,3,5,6,4,9,8,12,17,34,11};
        public static void main(String[] args) {
                buildMaxHeapify(sort);
                heapSort(sort);
                print(sort);
        }

        private static void buildMaxHeapify(int[] data){
                //没有子节点的才需要创建最大堆，从最后一个的父节点开始
                int startIndex = getParentIndex(data.length - 1);
                //从尾端开始创建最大堆，每次都是正确的堆
                for (int i = startIndex; i >= 0; i--) {
                        maxHeapify(data, data.length, i);
                }
        }

        /**
         * 创建最大堆
         * @param data
```

```
     * @param heapSize需要创建最大堆的大小，一般在sort的时候用到，因为最多值放在末尾，末尾就不再归入最大堆了
     * @param index当前需要创建最大堆的位置
     */
    private static void maxHeapify(int[] data, int heapSize, int index){
        // 当前点与左右子节点比较
        int left = getChildLeftIndex(index);
        int right = getChildRightIndex(index);

        int largest = index;
        if (left < heapSize && data[index] < data[left]) {
            largest = left;
        }
        if (right < heapSize && data[largest] < data[right]) {
            largest = right;
        }
        //得到最大值后可能需要交换，如果交换了，其子节点可能就不是最大堆了，需要重新调整
        if (largest != index) {
            int temp = data[index];
            data[index] = data[largest];
            data[largest] = temp;
            maxHeapify(data, heapSize, largest);
        }
    }

    /**
     * 排序，最大值放在末尾，data虽然是最大堆，在排序后就成了递增的
     * @param data
     */
    private static void heapSort(int[] data) {
        //末尾与头交换，交换后调整最大堆
        for (int i = data.length - 1; i > 0; i--) {
            int temp = data[0];
            data[0] = data[i];
            data[i] = temp;
            maxHeapify(data, i, 0);
        }
    }
```

```
    /**
     * 父节点位置
     * @param current
     * @return
     */
    private static int getParentIndex(int current){
        return (current - 1) >> 1;
    }

    /**
     * 左子节点position注意括号，加法优先级更高
     * @param current
     * @return
     */
    private static int getChildLeftIndex(int current){
        return (current << 1) + 1;
    }

    /**
     * 右子节点position
     * @param current
     * @return
     */
    private static int getChildRightIndex(int current){
        return (current << 1) + 2;
    }

    private static void print(int[] data){
        int pre = -2;
        for (int i = 0; i < data.length; i++) {
            if (pre < (int)getLog(i+1)) {
                pre = (int)getLog(i+1);
                System.out.println();
            }
            System.out.print(data[i] + " |");
        }
    }

    /**
     * 以2为底的对数
```

```
         * @param param
         * @return
         */
        private static double getLog(double param){
                return Math.log(param)/Math.log(2);
        }
}
```

Python

```
#!/usr/bin/env python
#-*-coding:utf-8-*-

def heap_sort(lst):
    def sift_down(start, end):
        """最大堆调整"""
        root = start
        while True:
            child = 2 * root + 1
            if child > end:
                break
            if child + 1 <= end and lst[child] < lst[child + 1]:
                child += 1
            if lst[root] < lst[child]:
                lst[root], lst[child] = lst[child], lst[root]
                root = child
            else:
                break

    # 创建最大堆
    for start in xrange((len(lst) - 2) // 2, -1, -1):
        sift_down(start, len(lst) - 1)

    # 堆排序
    for end in xrange(len(lst) - 1, 0, -1):
        lst[0], lst[end] = lst[end], lst[0]
        sift_down(0, end - 1)
    return lst
```

```
def main():
    l = [9,2,1,7,6,8,5,3,4]
    heap_sort(l)

if __name__ == "__main__":
    main()
```

JavaScript

```
Array.prototype.heap_sort = function() {
        var arr = this.slice(0);
        function swap(i, j) {
                var tmp = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp;
        }

        function max_heapify(start, end) {
                //建立父節點指標和子節點指標
                var dad = start;
                var son = dad * 2 + 1;
                if (son >= end)//若子節點指標超過範圍直接跳出函數
                        return;
                if (son + 1 < end && arr[son] < arr[son + 1])//先比較兩個子節點大
小，選擇最大的
                        son++;
                if (arr[dad] <= arr[son]) {//如果父節點小於子節點時，交換父子內容再
繼續子節點和孫節點比較
                        swap(dad, son);
                        max_heapify(son, end);
                }
        }

        var len = arr.length;
        //初始化，i從最後一個父節點開始調整
        for (var i = Math.floor(len / 2) - 1; i >= 0; i--)
                max_heapify(i, len);
        //先將第一個元素和已排好元素前一位做交換，再從新調整，直到排序完畢
        for (var i = len - 1; i > 0; i--) {
```

```
                swap(0, i);
                max_heapify(0, i);
        }

        return arr;
};
var a = [3, 5, 3, 0, 8, 6, 1, 5, 8, 6, 2, 4, 9, 4, 7, 0, 1, 8, 9, 7, 3, 1, 2, 5, 9, 7, 4, 0, 2, 6];
console.log(a.heap_sort());
```

PHP

```
<?php
function swap(&$x, &$y) {
        $t = $x;
        $x = $y;
        $y = $t;
}

function max_heapify(&$arr, $start, $end) {
        //建立父節點指標和子節點指標
        $dad = $start;
        $son = $dad * 2 + 1;
        if ($son >= $end)//若子節點指標超過範圍直接跳出函數
                return;
        if ($son + 1 < $end && $arr[$son] < $arr[$son + 1])//先比較兩個子節點大小，選擇最大的
                $son++;
        if ($arr[$dad] <= $arr[$son]) {//如果父節點小於子節點時，交換父子內容再繼續子節點和孫節點比較
                swap($arr[$dad], $arr[$son]);
                max_heapify($arr, $son, $end);
        }
}

function heap_sort($arr) {
        $len = count($arr);
        //初始化，i從最後一個父節點開始調整
        for ($i = $len / 2 - 1; $i >= 0; $i--)
```

```
            max_heapify($arr, $i, $len);
        //先將第一個元素和已排好元素前一位做交換，再從新調整，直到排序完畢
        for ($i = $len - 1; $i > 0; $i--) {
            swap($arr[0], $arr[$i]);
            max_heapify($arr, 0, $i);
        }
        return $arr;
}

$arr = array(3, 5, 3, 0, 8, 6, 1, 5, 8, 6, 2, 4, 9, 4, 7, 0, 1, 8, 9, 7, 3, 1, 2, 5, 9, 7, 4, 0, 2, 6);
$arr = heap_sort($arr);
for ($i = 0; $i < count($arr); $i++)
        echo $arr[$i] . ' ';
?>
```

原地堆排序

基于以上堆相关的操作，我们可以很容易的定义堆排序。例如，假设我们已经读入一系列数据并创建了一个堆，一个最直观的算法就是反复的调用del_max()函数，因为该函数总是能够返回堆中最大的值，然后把它从堆中删除，从而对这一系列返回值的输出就得到了该序列的降序排列。真正的原地堆排序使用了另外一个小技巧。堆排序的过程是：

1. 创建一个堆H[0..n-1]
2. 把堆首（最大值）和堆尾互换
3. 把堆的尺寸缩小1，并调用shift_down(0),目的是把新的数组顶端数据调整到相应位置
4. 重复步骤2，直到堆的尺寸为1

平均复杂度

堆排序的平均时间复杂度为$O(n\log n)$，空间复杂度为$\Theta(1)$。