实验三　堆排序与快速排序

May 19, 2022

1　前言

1.1　堆排序

堆排序是基于二叉堆而实现的排序，它可以在 $O(\log n)$ 的时间内插入、删除元素，并 $O(1)$ 的返回当前集合的最大值或最小值，借助选择排序的思想，堆排序可以在 $O(n \log n)$ 的时间内完成数组的排序。

二叉堆是一个完全二叉树，当树中任意一个节点的权值都大于等于它儿子的权值，那么它是一个大根堆，当树中任意一个节点的权值都小于等于它儿子的权值，那么它是一个小根堆。

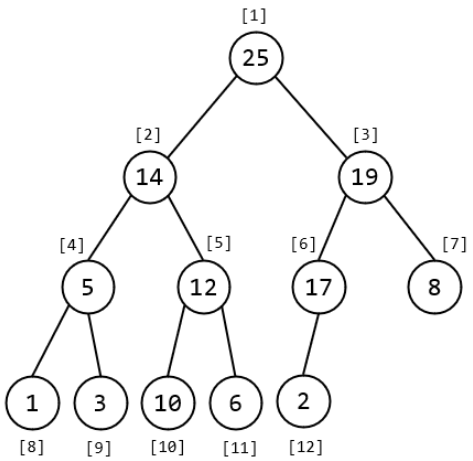

图 1: 大根堆

以下标为 1 开始的数组 heap:[25,14,19,5,12,17,8,1,3,10,6,2] 表示的二叉堆如图 1 所示。容易发现，节点 x 的左孩子为 $2x$，右孩子为 $2x+1$，父亲节点为 $\lfloor x/2 \rfloor$。因此，我们可以很容易的用下标为 1 的数组来表示一个二叉堆。

更进一步的，长度为 n 的数组若要表示为一个完全二叉树，只有下标范围在 $[1, n/2]$ 中的元素才会作为非叶子节点。基于这一性质，我们可以直接从 $\lfloor n/2 \rfloor$

倒序的建堆，当建立以 i 节点为根的二叉堆时，我们假定它的两个子树：以 $2i$ 和 $2i+1$ 为根结点的子树都已经是二叉堆。构建的伪代码如下：

Algorithm 1 Max-Heapify

Require: the binary trees rooted at 2i and 2i+1 are submaxheaps

```
1: function MAX-HEAPIFY(A, i)
2:     l = 2 × i
3:     r = 2 × i + 1
4:     if l ≤ A.heapsize and A[l] > A[i] then
5:         largest = l
6:     else
7:         largest = i
8:     if r ≤ A.heapsize and A[r] > A[largest] then
9:         largest = r
10:    if largest ≠ i then
11:        swap(A[i], A[largest])
12:        MAX-HEAPIFY(A, largest)
```

由此，我们以 Down-To-Top 的顺序建立二叉堆：

Algorithm 2 Build-Max-Heap

```
1: function BUILD-MAX-HEAP(A)
2:     A.heapsize = A.length
3:     for i = ⌊A.length/2⌋ downto 1 do
4:         MAX-HEAPIFY(A, i)
```

由于堆顶元素为堆中最大值，所以我们依旧按照 Down-To-Top 的顺序对数组进行排序：

Algorithm 3 HEAPSORT

```
1: function HEAPSORT(A)
2:     BUILD-MAX-HEAP(A)
3:     for i = A.length downto 2 do
4:         swap(A[1], A[i])
5:         A.heapsize = A.heapsize − 1
6:         MAX-HEAPIFY(A, 1)
```

看起来建堆的复杂度为 $O(n \log n)$，实则不然。假设树高 $h = \log n$，第 h 层为叶子结点，在对第 x 层的节点 i 调整时，MAX-HEAPIFY 最多会调用 $h - x$

次，而第 x 层的结点数最多有 2^{x-1} 个。所以总调用次数为：$\sum_{x=1}^{h} 2^{x-1}(h-x)$。该式是一个等比数列，可以用错位相减法求渐近复杂度，化简可得 $O(n)$。

1.2 快速排序

快速排序（Quicksort），又称分区交换排序（partition-exchange sort），简称快排，是一种被广泛运用的排序算法。它的原理是分治算法，主要过程如下：

1. 将数列划分为两部分（左侧所有元素都小于等于右侧）；
2. 递归两个子序列分别进行快速排序；
3. 不用合并，因为此时数列已经完全有序。

关键的问题是如何在线性时间内将数组划分为具有相对大小关系的两部分（左边小于等于右边）。为了保证平均复杂度，一般是在原数组中随机选择一个数作为 pivot，通过维护一前一后两个指针 i 和 j 来实现数组的划分操作：

Algorithm 4 Quick Sort

```
1: function PARTITION(A, p, q)
2:     swap(A[p], A[rand(p, q)]) ▷ Randomly choose a number in the range p to q and move it to the first place
3:     pivot = A[p]
4:     i = p
5:     for j = p + 1 to q do
6:         if A[j] < pivot then
7:             i = i + 1
8:             swap(A[i], A[j])
9:     swap(A[i], A[p])
10:    return i
11: function QUICK-SORT(A, l, r)
12:    if l < r then
13:        m = PARTITION(A, l, r)
14:        QUICK-SORT(A, l, m − 1)
15:        QUICK-SORT(A, m + 1, r)
16: QUICK-SORT(A, 1, n)                                  ▷ Initial call
```

其实，快速排序没有指定应如何具体实现划分，不论是选择 pivot 的过程还是划分的过程，都有不止一种实现方法。

本节实验的目标是实现堆排序与快速排序。

2 实验项目结构

- include
  - util.hpp 常用函数头文件
  - MySort.hpp 基类 *MySort*
  - InsertionSort.hpp 插入排序
  - MergeSort.hpp 归并排序
  - HeapSort.hpp 堆排序
  - QuickSort.hpp 快速排序
  - ThreeWayQuickSort.hpp 三路快速排序（拓展题）
- data
  - sample
  - test
  - generate
- performance
  - performance.cpp 性能测试程序代码
- main.cpp 主程序代码

本实验代码结构与实验一类似。

请注意，每次对修改完代码之后，需要重新编译运行 main.cpp，如果直接执行上次编译好的 main.exe 或 main，新的修改将不会生效。

3 实验内容

3.1 堆排序

基于前言中介绍的堆排序思想，实现 HeapSort.hpp 中的堆排序函数。

```
class HeapSort: public MySort {
public:
    int heap_size;
    int length;
    void max_heapify(std::vector<int>& nums, int i) {
        // 请在这里完成你的代码
    }
    void build_max_heap(std::vector<int>& nums) {
        // 请在这里完成你的代码
    }
```

```
    void mysort(std::vector<int>& nums) {
        length = nums.size();
        nums.insert(nums.begin(), 0); // 在开头插入一个元素，使得待排序
            元素下标从 1 开始

        // 请在这里完成你的代码

        nums.erase(nums.begin()); // 删除开头元素
    }
};
```

3.2 快速排序

基于前言中介绍的快速排序思想，实现 QuickSort.hpp 中的快速排序函数。

```
class QuickSort: public MySort {
public:
    int partition(std::vector<int>& nums, int p, int q) {
        // 请在这里完成你的代码
    }
    void quick_sort(std::vector<int>& nums, int l, int r) {
        // 请在这里完成你的代码
    }
    void mysort(std::vector<int>& nums) {
        if(nums.size() == 0) return;
        quick_sort(nums, 0, nums.size() - 1);
    }
};
```

4 实验思考

1. 以第一组测试数据（直接打断点即可观察）为例，分别观察堆排序和快速排序过程中数组元素的变化情况（需要在报告中给出每一步数组的情况）。

   - input

     5

     1 6 2 10 2

   - output

     1 2 2 6 10

2. 比较插入排序、归并排序、堆排序与快速排序的性能差距（通过编译运行 performance.cpp 来对比）。

5 拓展实验

5.1 三路快速排序

如果数组原本就是升序或者降序的，那么基于随机选择 pivot 的快速排序算法要比朴素的快速排序算法快很多。但如果数组中存在着大量的重复数据，基于随机选择 pivot 的快速排序算法最差复杂度仍旧会变成 $O(n^2)$ 的。一种叫做三**路快速排序**的优化方法可以解决这个问题，与原始的快速排序不同，三路快速排序在随机选取分界点 pivot 后，将待排数列划分为三个部分：小于 pivot、等于 pivot 以及大于 pivot 。这样做即实现了将与分界元素相等的元素聚集在分界元素周围这一效果。Algorithm 5 展示了三路快速排序的伪代码。

Algorithm 5 Three-Way-Quick-Sort

```
1:  function THREE-WAY-QUICK-SORT(A, p, q)
2:      if p >= q then
3:          return
4:      pivot = A[rand(p, q)]
5:      i = p                                  ▷ i 指向当前要处理的元素
6:      j = p                  ▷ 排序过程中，[p, j − 1] 表示小于 pivot 的区间
7:      k = q                  ▷ 排序过程中，[k + 1, q] 表示大于 pivot 的区间
8:      while i <= k do
9:          if A[i] < pivot then
10:             swap(A[i], A[j])
11:             i = i + 1
12:             j = j + 1
13:         else if A[i] > pivot then
14:             swap(A[i], A[k])
15:             k = k − 1
16:         else
17:             i = i + 1
18:     THREE-WAY-QUICK-SORT(A, p, j − 1)
19:     THREE-WAY-QUICK-SORT(A, k + 1, q)
```

三路快速排序在处理含有多个重复值的数组时，效率远高于原始快速排序。其最佳时间复杂度为 $O(n)$。

5.2 内省排序

内省排序（Introsort 或 Introspective sort）是快速排序和堆排序的结合，由 David Musser 于 1997 年发明。内省排序其实是对快速排序的一种优化，保证了最差时间复杂度为 $O(n \log n)$。

内省排序将快速排序的最大递归深度限制为 $\lfloor \log n \rfloor$，超过限制时就转换为堆排序。这样既保留了快速排序内存访问的局部性，又可以防止快速排序在某些情况下性能退化为 $O(n^2)$。

从 2000 年 6 月起，C++ STL 中的 sort() 函数的实现采用了内省排序算法，在此实现中，当数据数量少于 16 时，将采用插入排序。Algorithm 6 展示了内省排序的伪代码。

Algorithm 6 Introsort

function SORT(A)
    maxdepth = $\lfloor \log_2^{length(A)} \rfloor \times 2$
    INTROSORT($A, maxdepth$)
function INTROSORT(A, maxdepth)
    $n = length(A)$
    if $n < 16$ **then**
        INSERTIONSORT(A)
    else if $maxdepth == 0$ **then**
        HEAPSORT(A)
    else
        p = PARTITION(A)
        INTROSORT($A[1 : p-1], maxdepth - 1$)
        INTROSORT($A[p+1 : n], maxdepth - 1$)

拓展实验仅要求根据上述三路快速排序的伪代码完成 ThreeWayQuickSort.hpp 的实现。

```
class ThreeWayQuickSort: public MySort {
public:
    void three_way_quick_sort(std::vector<int>& nums, int p, int q) {
        // 请在这里完成你的代码
    }
    void mysort(std::vector<int>& nums) {
        if(nums.size() == 0) return;
        three_way_quick_sort(nums, 0, nums.size() - 1);
    }
};
```