关于堆数据结构相关操作时间复杂度的分析

XX*

XX 学院　XX 大学

December 2019

摘　要

本文从理论上分析了堆数据结构相关的各个操作的时间复杂度，对其中的 *build-heap* 操作从计算次数的视角**精确地**分析了时间复杂度，并进行了实验评估. 最终得出结论，即使是对于特定数据结构的特定操作，仍然有可能存在时间复杂度差异很大的实现方法.

1　堆数据结构

本文中的 “堆”（Heap），是指计算机科学中的一种特殊的树状数据结构. 当一棵完全树满足**堆序性**时，可以被称为 “堆”：“给定堆中任意节点 P 和 C，若 P 是 C 的父节点，那么 P 的值会小于等于（或大于等于）C 的值” [2]. 若父节点的值恒小于等于子节点的值，此堆称为最小堆（min heap）；反之，若父节点的值恒大于等于子节点的值，此堆称为最大堆（max heap）. 在堆中最顶端的那一个节点，称作根节点（root node），根节点本身没有父节点（parent node）.

堆的最常见实现是二叉堆，这意味着，其中的树是一棵完全二叉树. 图1，就是一个节点键值均在 1 到 100 之间的最大二叉堆实例.

2　堆相关操作

堆最早出现于 J. W. J. Williams 在 1964 年发表的堆排序（heap sort）[1]，当时他提出了二叉堆作为此算法的数据结构. 为了使堆能够完成将一列数从小到大排序这一过程，则大致应有以下几个涉及堆的操作：

*邮箱: XXXX@XXXXXX.edu.cn，学号: XXXXXXXX

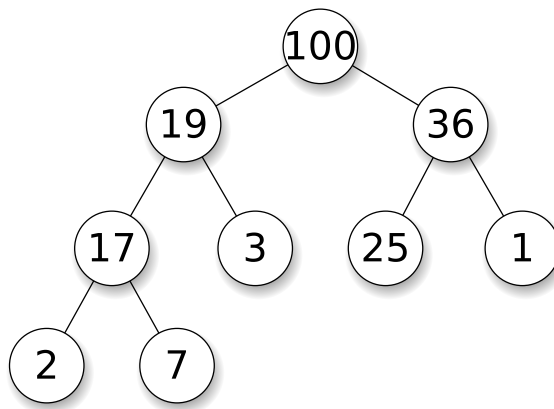

图 1: 最大二叉堆

- *create-heap*：创建一个空的堆；
- *insert*：插入一个新的键（也被称为 *push*）；
- *extract-min*：从最小堆中返回最小值的节点后，将其从堆中删除（也被称为 *pop*）；
- *size*：返回堆的大小，也即堆的节点数；
- *is-empty*：堆为空则返回 true，否则返回 false.

实际上，为了实现 *push* 和 *pop* 操作，堆还需要两个内部的操作 *sift-up* 和 *sift-down*，这也是堆最基本、最核心的操作：

- *sift-up*：根据需要在树中向上移动节点，通常用于插入操作后维持堆序性；
- *sift-down*：与 *sift-up* 类似，根据需要在树中向下移动节点，通常用于删除操作后维持堆序性.

更进一步地，其实也并不一定需要一个创建空堆的操作，而是使用：

- *build-heap*：从给定的元素序列创建一个堆.

本文将会详细讨论 *build-heap* 在不同实现下的不同时间复杂度，下文会将“堆”与“二叉堆”以及“复杂度”与“时间复杂度”作为等价的词使用.

3 时间复杂度分析

3.1 基本操作的时间复杂度

记 n 为堆中元素个数，由于堆是一个完全二叉树，所以树的高度 h 等于 $\lfloor \log_2 n \rfloor + 1$，也即，该堆的层数是 $\mathbf{O}(\log n)$ 的. 在此基础上，可以对堆相关操作进行复杂度分析.

create-heap 创建一个空的堆，（除了设置计数器外）不涉及到任何其他的操作，复杂度 $\mathbf{O}(1)$.

size 引入一个计数器，在创建堆时置为 0，在添加或者删除节点时相应地修改. 该操作直接返回计数器的值，复杂度 $\mathbf{O}(1)$.

is-empty 只需要判断是否存在根元素，复杂度 $\mathbf{O}(1)$.

sift-up 向上移动一个节点，最差情况下从最底层到最顶层每一层都需要比较和交换一遍，由于比较节点是交换节点的必要条件，只考虑交换节点的操作，则 $T(n) = h - 1$，复杂度 $\mathbf{O}(\log n)$.

sift-down 向下移动一个节点，与 *sift-up* 类似，从最顶层到最底层每一层都交换，不考虑多了一次比较节点的操作，则 $T(n) = h - 1$，复杂度 $\mathbf{O}(\log n)$.

之后的分析将涉及最常见的实现二叉堆的方式，即支持随机访问的线性列表（更具体地，一维数组），记为 $a[\,]$，并且该列表的下标从 1 开始（**1-base**）. 当堆中的节点按照层次序（从上到下、从左到右）排列在列表中时，可以推出任一节点 $a[i]$ 的两个子节点分别是 $a[2i]$ 和 $a[2i+1]$（最底层的叶节点没有子节点），任一节点 $a[i]$ 的父节点是 $a[\lfloor \frac{i}{2} \rfloor]$（最顶层的根节点没有父节点）.

push(insert) 在列表末尾再添加一个节点，并对该新节点调用 *sift-up* 操作，复杂度即为 *sift-up* 的复杂度，$\mathbf{O}(\log n)$.

pop(extract-min) 引入一个临时变量记录根节点的值，将根节点与列表最后一个节点交换，然后删除最后一个节点（即原来的根节点），再对新的根节点调

用 *sift-down* 操作，最后返回临时变量的值. 由于比 *sift-down* 操作多一次交换，所以 $T(n) = h - 1 + 1 = h$，复杂度 $\mathbf{O}(\log n)$.

3.2 *build-heap* 操作的时间复杂度

由于 *build-heap* 是一个更高层次（**抽象程度更高**）的操作，讨论其多种实现并分析不同实现的时间复杂度是很有意义的. 记 *input* 为输入的原始序列，对于 *input*，除了可以被迭代以及可以返回元素个数，并没有任何特殊要求.

(1) 算法 1

根据上文中定义的基本操作，可以很容易地想到一个朴素的算法来实现 *build-heap* 操作，伪代码如下.

```
BUILD-HEAP(input)
1  heap = CREATE-HEAP()
2  for each element x ∈ input
3      INSERT(heap, x)
4  return heap
```

该算法的流程是很直观的，首先创建一个空的堆，然后再将元素一个一个地插入. 为了避免在复杂度分析中引入误差导致错误，这里并不直接引用 *insert* 的大 **O** 复杂度，而是重新通过计算次数 $T(n)$ 来得到大 **O** 复杂度. 根据上文所述，*insert* 跟 *sift-up* 的计算次数相同，为 $T(n^*) = h^* - 1$. 容易发现，对于插入时处在同一层的节点，计算次数总是相等的，所以分层来考虑是比较便捷的. 定义最顶层为第 1 层，那么除了最底层的叶子节点，第 k 层的节点个数恒为 2^{k-1}，且每个节点的计算次数为 $k-1$，所以总的计算次数为

$$T(n) = \sum_{k=1}^{h-1} (2^{k-1} \times (k-1)) + (n - \sum_{k=1}^{h-1} 2^{k-1}) \times (h-1)$$

记 $S(h)$ 为第一项，即 $S(h) \triangleq \sum_{k=1}^{h-1} (2^{k-1} \times (k-1))$，则 $S(h)$ 可以通过简单的计算得到.

$$S(h) = 0 \cdot 2^0 + 1 \cdot 2^1 + \cdots + (h-2) \cdot 2^{h-2} \tag{1}$$

等式两边同时乘以 2，则

$$2S(h) = 0 \cdot 2^1 + 1 \cdot 2^2 + \cdots + (h-2) \cdot 2^{h-1} \tag{2}$$

用 (2) 式减去 (1) 式，可得

$$\begin{aligned} 2S(h) - S(h) &= -(2^1 + 2^2 + \cdots + 2^{h-2}) + (h-2) \cdot 2^{h-1} \\ S(h) &= -(2^{h-1} - 2) + (h-2) \cdot 2^{h-1} \\ S(h) &= (h-3) \cdot 2^{h-1} + 2 \end{aligned}$$

所以，

$$\begin{aligned} T(n) &= S(h) + (n - \sum_{k=1}^{h-1} 2^{k-1}) \times (h-1) \\ &= (h-3) \cdot 2^{h-1} + 2 + (n - (2^{h-1} - 1)) \times (h-1) \\ &= n(h-1) + h + 1 - 2^h \end{aligned}$$

当 $n \geq 1$ 时，可得 $h \geq 1, h+1 \leq 2^h, T(n) \leq n(h-1)$，故该算法的复杂度为 $\mathbf{O}(n \log n)$.

(2) 算法 2

正如上文所述，朴素的算法1其实就是顺序调用了 n 次 *insert*，而 *insert* 背后其实是调用了 *sift-up*. 那么也可以直接去调用 *sift-up* 操作，伪代码如下.

```
BUILD-HEAP(input)
1  heap = CREATE-HEAP()
2  i = 0
3  for each element x ∈ input
4        i = i + 1
5        heap.a_i = x
6  heap.size = i                              // Not a heap yet
7  for j = 2 to i
8        SIFT-UP(heap, j)
9  return heap
```

该算法的流程有一些不同，第一步它先将所有的元素都放入了堆的线性列表中，然后从前往后调用了 $n-1$ 次 *sift-up* 函数. 容易发现，在不考虑第一步放入的情况下，它的计算次数与算法1完全相同，即 $T(n) = n(h-1) + h + 1 - 2^h \leq n(h-1)$，复杂度为 $\mathbf{O}(n \log n)$.

(3) 算法 3

算法2虽然在复杂度上并没有什么改进，但是它极大地启发了我们. 如果所有的节点都已经创建好了，那么最顶层的根节点可以不参与顺序调用 *sift-up* 的过程. 同样道理地，如果所有的节点都已经创建好了，那么最底层的叶节点也可以不参与逆序调用 *sift-down* 的过程，伪代码如下.

```
BUILD-HEAP(input)
 1  heap = CREATE-HEAP()
 2  i = 0
 3  for each element x ∈ input
 4      i = i + 1
 5      heap.a_i = x
 6  heap.size = i                         // Not a heap yet
 7  i = ⌊i/2⌋
 8  for j = i downto 1
 9      SIFT-DOWN(heap, j)
10  return heap
```

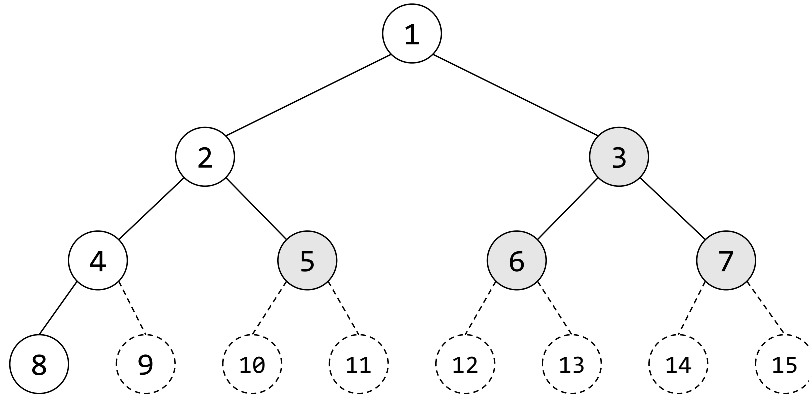

图 2: 带有虚拟节点的最小二叉堆

有了算法2的铺垫，该算法还是比较好理解的，第一步先将所有的元素都放入了堆的线性列表中，然后从后往前调用了 $\lfloor \frac{n}{2} \rfloor$ 次 *sift-down* 函数. 仍然按照层次来分析，可以计算出堆在最底层缺少了 $2^h - 1 - n$ 个节点，记为 m. 为了计算上的简便，可以添加 m 个虚拟节点以满二叉堆分析计算次数，那

么有一部分节点会比真实情况多交换 1 次，如图2中浅灰色背景的节点，最后将这部分次数减去即可. 设满二叉堆的计算次数为 $S(h)$，则

$$S(h) = 2^{h-1} \times 0 + 2^{h-2} \times 1 + \cdots + 2^0 \times (h-1) \tag{3}$$

等式两边同时乘以 2，则

$$2S(h) = 2^h \times 0 + 2^{h-1} \times 1 + \cdots + 2^1 \times (h-1) \tag{4}$$

用 (4) 式减去 (3) 式，可得

$$\begin{aligned} 2S(h) - S(h) &= 2^{h-1} + 2^{h-2} + \cdots + 2^1 - (h-1) \\ S(h) &= 2^h - 1 - h \end{aligned}$$

可以发现，m 个最底层的虚拟节点对次底层的节点贡献了 $\lfloor \frac{m}{2} \rfloor$ 次虚拟的交换次数，而 $\lfloor \frac{m}{2} \rfloor \geq \frac{m-(2-1)}{2}$，以此类推，可以得到

$$\begin{aligned} T(n) &= S(h) - (\lfloor \frac{m}{2} \rfloor + \lfloor \frac{m}{2^2} \rfloor + \cdots + \lfloor \frac{m}{2^{h-1}} \rfloor) \\ &\leq 2^h - 1 - h - (\frac{m-(2-1)}{2} + \frac{m-(2^2-1)}{2^2} + \cdots + \frac{m-(2^{h-1}-1)}{2^{h-1}}) \\ &= 2^h - 1 - h - (m+1)(\frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^{h-1}}) + (h-1) \\ &= 2^h - 2 - (2^h - 1 - n + 1)(1 - \frac{1}{2^{h-1}}) \\ &= n(1 - \frac{1}{2^{h-1}}) \end{aligned}$$

当 $n \geq 1$ 时，可得 $h \geq 1, \frac{1}{2^{h-1}} \leq 1, T(n) \leq n$，故该算法的复杂度为 $\mathbf{O}(n)$.

4 实验评估

本文中使用一亿个（100000000 个）32 位无符号整数来对序列建堆操作所需的时间进行评估.

首先需要写出实验代码（参见附录 A）用于生成数据，作者利用该程序生成了 3 个随机的输入序列。再写出实验代码（参见附录 B、C、D）实现上述 3 种算法，将这 3 份代码无参数编译，输入生成的 3 个序列分别运行 3 轮，实验结果如表1所示。

可以看到，算法3的运行时间明显比算法1和算法2短，与上文中计算的大 $\mathbf{O}$ 复杂度相吻合. 至于为什么没有达到加快几十倍的效果，本文中并不讨论.

表 1: 运行时间对比

	轮次	test0 用时 (s)	test1 用时 (s)	test2 用时 (s)	平均用时 (s)	总平均用时 (s)
算法 1	第 1 轮	2.852108	2.932702	2.840246	2.875019	2.879898
	第 2 轮	2.902857	2.855334	2.948962	2.902384	
	第 3 轮	2.837630	2.841070	2.908170	2.862290	
算法 2	第 1 轮	2.714576	2.725721	2.754266	2.731521	2.728887
	第 2 轮	2.771832	2.635403	2.772449	2.726561	
	第 3 轮	2.765820	2.689524	2.730389	2.728578	
算法 3	第 1 轮	2.014048	1.993455	2.044974	2.017492	2.022245
	第 2 轮	1.989275	2.009542	2.015313	2.004710	
	第 3 轮	2.011517	2.041653	2.080431	2.044534	

5 结论

本文从理论上分析了堆数据结构相关的各个操作的时间复杂度，对其中的 *build-heap* 操作从计算次数的视角**精确地**分析了时间复杂度，并进行了实验评估. 在作者有限的了解中，并没有人做过这样精确的计算.

根据本文的分析和实验结果，可以发现，即使是对于特定数据结构的特定操作，仍然有可能存在复杂度差异很大的实现方法.

但是需要指出的是，其实复杂度最优的算法3是一种非常奇特的算法，首先它生成了一个并不满足堆序性的“堆”，破坏了该数据结构的基本性质，其次必须在可随机访问的线性列表上逆序调用若干次 *sift-down* 操作，并且与前文所述的“*sift-down* 通常用于删除操作后维持堆序性”是相违背的.

心路历程

11 月某天，在我本科学校的 OJ 群里学弟 YYY 发了一份 *Dijkstra* 算法的 C++ 代码截图，询问为什么仍然时间超限，复杂度是否正确. 当时看到他使用了`std::make_heap()`，我理所当然地认为这个语句的复杂度是 $\mathbf{O}(n\log n)$. 直到另一个学弟 ZZ 出来说`std::make_heap()` 的复杂度是 $\mathbf{O}(n)$ 时，我才产生疑问并且开始认真思考这个问题.

感谢 XX 大学 XX 学院 XX 专业 XX 级的 YYY 同学和 XX 专业 XX 级的 ZZ 同学，给了我本次论文的灵感，以及重新学习堆这个数据结构的机会.

在本文的完成过程中，我了解到《算法导论》一书在简短的说明后，给出了建堆过程时间复杂度是 $\mathbf{O}(n)$ 的结论 [3]. 尽管我曾经阅读过这本书籍，但我还是没能第一时间想清楚这个问题. 这也就恰恰说明了，很多时候如果一个结论自己并没有仔细思考和推导过，往往是理解不深刻的，也是容易遗忘的.

本文的结论还让我回想起分布式系统中需要维持的一致性的强弱实际上是直接影响执行效率的，在只需要确保**最终一致性**的情况下，效率往往是**最高**的.

参考文献

[1] J. w. j. williams - wikipedia. https://en.wikipedia.org/wiki/J._W._J._Williams. Accessed December 3, 2019.

[2] 堆 - 维基百科，自由的百科全书. https://zh.wikipedia.org/wiki/%E5%A0%86%E7%A9%8D. Accessed December 3, 2019.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

附录

A gen.cpp

```
#include "testlib.h"
#include <bits/stdc++.h>
using namespace std;

const int C1E8 = 100000000;

int main(int argc, char* argv[])
{
    registerGen(argc, argv, 1);
    int n=C1E8;
    printf("%d\n",n);
    for (int i=1;i<=n;i++) {
        printf("%lld%c",rnd.next(0LL,1LL<<32-1)," \n"[i==n]);
    }
    return 0;
}
```

B insert.cpp

```
#include <stdio.h>
#include <time.h>
#include <algorithm>

using namespace std;

const int N=100000000;
unsigned input[N+5];
unsigned a[N+5];

void sift_up(int p)
{
```

```
    while (p>1 && a[p]<a[p/2]) {
        swap(a[p],a[p/2]);
        p/=2;
    }
}

void insert(int p,int x)
{
    a[p]=x;
    sift_up(p);
}

int main()
{
    int n;
    scanf("%d", &n);
    for (int i=1;i<=n;i++) {
        scanf("%u",&input[i]);
    }
    clock_t start = clock();
    for (int i=1;i<=n;i++) {
        insert(i,input[i]);
    }
    clock_t end = clock();
    printf("Use time: %.6fs\n",(end - start)*1.0/CLOCKS_PER_SEC);
    return 0;
}
```

C sift-up.cpp

```
#include <stdio.h>
#include <time.h>
#include <algorithm>
```

```
using namespace std;

const int N=100000000;
unsigned input[N+5];
unsigned a[N+5];

void sift_up(int p)
{
    while (p>1 && a[p]<a[p/2]) {
        swap(a[p],a[p/2]);
        p/=2;
    }
}

int main()
{
    int n;
    scanf("%d", &n);
    for (int i=1;i<=n;i++) {
        scanf("%u",&input[i]);
    }
    clock_t start = clock();
    for (int i=1;i<=n;i++) {
        a[i]=input[i];
    }
    for (int i=2;i<=n;i++) {
        sift_up(i);
    }
    clock_t end = clock();
    printf("Use time: %.6fs\n",(end - start)*1.0/CLOCKS_PER_SEC);
    return 0;
}
```

D sift-down.cpp

```cpp
#include <stdio.h>
#include <time.h>
#include <algorithm>

using namespace std;

const int N=100000000;
unsigned input[N+5];
unsigned a[N+5];

void sift_down(int p, int n)
{
    while (p*2<=n) {
        int q=p*2;
        if (p*2+1<=n && a[p*2+1]<a[p*2]) {
            q+=1;
        }
        if (a[p]<=a[q]) {
            break;
        }
        swap(a[p],a[q]);
        p=q;
    }
}

int main()
{
    int n;
    scanf("%d", &n);
    for (int i=1;i<=n;i++) {
        scanf("%u",&input[i]);
    }
```

```
    clock_t start = clock();
    for (int i=1;i<=n;i++) {
        a[i]=input[i];
    }
    for (int i=n/2;i>=1;i--) {
        sift_down(i,n);
    }
    clock_t end = clock();
    printf("Use time: %.6fs\n",(end - start)*1.0/CLOCKS_PER_SEC);
    return 0;
}
```