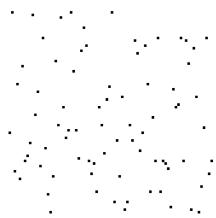
Heapsort



Consider the sorting problem discussed in another chapter. Namely, consider the problem of permuting the elements of an array `v[1..n]` to put them in increasing order. In other words, rearrange the elements of the array so that `v[1] ≤ . . . ≤ v[n]`.

The other chapter analysed some simple algorithms for the problem. The present chapter examines Heapsort, an algorithm discovered by J.W.J. Williams in 1964. Unlike the simple algorithms, Heapsort is linearithmic, even in the worst case.

We shall assume that the indices of the array are `1..n`, rather than the usual `0..n-1`. This convention will make the code a little simpler.

Table of contents:

Arrays and binary trees

In order to discuss the Heapsort, we must learn to see the binary tree hidden in any array. The set of indices of any array `v[1..m]` can be understood as a binary tree in the following way:

- the index `1` is the *root* of the tree;
- the *parent* of any index `c` is `c/2` (of course `1` has no parent);
- the *left child* of um index `p` is `2p` (this child exists only if `2p ≤ m`);
- the *right child* of `p` is `2p+1` (this child exists only if `2p+1 ≤ m`).

1	2	3	4	5	6	7	8	9	10	11	12	13	14
999	999	999	999	999	999	999	999	999	999	999	999	999	999

To make the binary tree stand out, we can draw the array in *layers*, so that every child sits in the layer immediately below that of its parent. The figure below is such a drawing of the array `v[1..56]`. (The numbers in the boxes are the indices `i` rather than the values `v[i]`.) Observe that each layer, except perhaps the last, has twice as many elements as the previous one. It follows that the number of layers in an array `v[1..m]` is exactly `1 + lg(m)`, where [lg](m) is the floor of [log] `m`.

```
1
2 3
4 5 6 7
8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
```

Exercises 1

1. Let `m` be a number of the form 2^k`-1`. Show that more than half of the elements of any array `v[1..m]` is in the last layer.

Heap

The mechanism behind the Heapsort algorithm is a data structure, known as heap, that sees the array as a binary tree. There are two flavors of the structure: max-heap and min-heap; we shall consider here only the first flavor and omit the "max-" prefix.

A heap, then, is an array in which the value of each parent is greater than or equal to the value of each of its two children. More precisely, an array `v[1..m]` is a *heap* if

`v[c/2] ≥ v[c]`

for `c = 2, . . . , m`. Here, as in the rest of this chapter, we shall agree that expressions figuring as indices of arrays are always computed in integer arithmetic. Hence, the value of the expression `c/2` is `⌊c/2⌋`, i.e., the floor of `c/2`.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
999	888	777	555	666	777	555	222	333	444	111	333	666	333

Occasionally, we consider certain "defective" heaps: we say that an array `v[1..m]` is a heap *except perhaps for index* `k` if the inequality `v[c/2] ≥ v[c]` holds for every `c` distinct from `k`.

Exercises 2

1. Is the array 161 41 101 141 71 91 31 21 81 17 16 a heap?
2. Show that every decreasing array is a heap. Show that the converse is not true.
3. ★ Write a function to decide whether an array `v[1..m]` is a heap.
4. ★ Show that `v[1..m]` is a heap if and only if each index `p` has the following properties: (*a*) `v[p] ≥ v[2p]` provided `2p ≤ m` and (*b*) `v[p] ≥ v[2p+1]` provided `2p+1 ≤ m`.
5. Suppose that `v[1..m]` is a heap and `p` is an index smaller than `m/2`. Is it true that `v[2p] ≥ v[2p+1]`? Is it true that `v[2p] ≤ v[2p+1]`?
6. Suppose that `v[1..m]` is a heap and let `i < j` be two indices such that `v[i] < v[j]`. If `v[i]` and `v[j]` are interchanged, will `v[1..m]` still be a heap? Repeat the exercise under the hypothesis `v[i] > v[j]`.
7. ★ Suppose that `v[1..m]` is a heap and the elements of the array are pairwise distinct. Of course the largest element of the array is `v[1]`. Where can the second largest element be? Where can the third largest element be? Is it true that the third largest element is a child of the second largest element?

Building a heap

It is easy to rearrange the elements of an array `v[1..m]` of integers so that it becomes a heap. Just repeat the following process: while the value of a child is larger than the value of its parent, swap the values of parent and child and move one step up, towards the root. More precisely, while `v[c/2] < v[c]`, do `swap (v[c/2], v[c])` and then `c = c/2`. The swap operation is

```
#define swap (A, B) {int t = A; A = B; B = t;}
```

Here is the complete code:

```
// Rearranges an array v[1..m] so that
// it decomes a heap.

static void
buildheap (int m, int v[])
{
   for (int k = 1; k < m; ++k) {
      // v[1..k] is a heap
      int c = k+1;
      while (c > 1 && v[c/2] < v[c]) {  // 5
         swap (v[c/2], v[c]);  // 6
         c /= 2;  // 7
      }
   }
}
```

(The keyword static indicates that `buildheap` is an auxiliary function that cannot be called directly by the user of Heapsort.)

At the beginning of every iteration controlled by the "`for`", the array `v[1..k]` is a heap. In the course of the iteration, `v[k+1]` moves up the heap (towards the root) until it finds its correct place and is thus incorporated into the heap.

In each repetition of the pair of lines `6-7`, the index `c` jumps from one layer of the array to the previous layer. Hence, this pair of lines can be repeated at most `lg(k)` times for each fixed `k`. As a consequence, the total number of comparisons (line `5` of the code) between elements of the array is at most m lg(m) .

(As we shall see further down, it is possible to do the job with just m comparisons.)

Exercises 3

1. IMPORTANT. Criticize the following ideia: to transform an array into a heap, just rearrange it in decreasing order (using Mergesort or Quicksort, for example).
2. Prove that function `buildheap` is correct. Begin by establishing the invariants of the iterative process that span the lines `5` to `7`.
3. Discuss the following version of the `buildheap` function:

```
for (int k = 1; k < m; ++k) {
   int c = k+1, x = v[k+1];
   while (c > 1 && v[c/2] < x) {
      v[c] = v[c/2];
      c /= 2; }
   v[c] = x; }
```

The sieve function

The core of many algorithms that manipulate heaps is a function that, unlike `buildheap`, moves *down* the heap, away from the root. This function, which we call `sieve`, receives an arbitrary array `v[1..m]` and

moves `v[1]` down to its correct position,

jumping from one layer to the next. How is this done? If `v[1]` ≥ `v[2]` and `v[1]` ≥ `v[3]`, nothing needs to be done. If `v[1]` < `v[2]` and `v[2]` ≥ `v[3]`, just swap `v[1]` with `v[2]` and move `v[2]` down to its correct position. In the other two cases, do something similar. (See a draft of the algorithm in pseudocode.) In the following example, each line of the figure shows the state of the array at the beginning of an iteration:

85	99	98	97	96	95	94	93	92	91	90	89	97	86

99	85	98	97	96	95	94	93	92	91	90	89	97	86

99	97	98	85	96	95	94	93	92	91	90	89	97	86

99	97	98	93	96	95	94	85	92	91	90	89	97	86

We can now write the code of the function. Each iteration begins with an index `p` and chooses a child `c` of `p` which has the largest value:

```
static void
sieve (int m, int v[]) {
   int c = 2;
   while (c <= m) {
      if (c < m && v[c] < v[c+1]) ++c;
      // c is a most valuable child of c/2
      if (v[c/2] >= v[c]) break;
      swap (v[c/2], v[c]);
```

```
            c *= 2;
        }
    }
```

The function will be applied to arrays that are heaps except perhaps for one or two indices. The function can, therefore, be documented as follows:

```
    // Receives an array v[1..m] that is a heap
    // except perhaps for indices 2 and 3 and
    // rearranges the array so that it becomes
    // a heap.
```

The following version is a little better, because it moves fewer elements of the array from one place to another (and does fewer divisions of `c` by `2`):

```
    static void
    sieve (int m, int v[])
    {
       int p = 1, c = 2, x = v[1];
       while (c <= m) {
          if (c < m && v[c] < v[c+1]) ++c;
          if (x >= v[c]) break;
          v[p] = v[c];
          p = c, c = 2*p;
       }
       v[p] = x;
    }
```

Performance. The `sieve` function is very fast. It does at most `lg(m)` iterations, since the array has `1 + lg(m)` layers. Each iteration involves two comparisons between elements of the array and therefore the total number of comparisons is at most

$$2\,\mathrm{lg}(m)\,.$$

The *time* spent is proportional to the number de comparisons and therefore proportional to $\log m$ in the worst case.

Exercises 4

1. Why is the following implementation of the `sieve` function incorrect?

```
   int p = 1, c = 2;
   while (c <= m) {
      if (v[p] < v[c]) {
         swap (v[p], v[c]);
         p = c;
         c = 2*p; }
      else {
         if (c < m && v[p] < v[c+1]) {
            swap (v[p], v[c+1]);
            p = c+1;
            c = 2*p; }
         else break; } }
```

2. Is the following alternative code of the `sieve` function correct?

```
   for (int c = 2; c <= m; c *= 2) {
      if (c < m && v[c] < v[c+1]) ++c;
      int p = c/2;
```

```
      if (v[p] >= v[c]) break;
      swap (v[p], v[c]); }
```

3. Discuss the following variant of the code of `sieve`:

```
int x = v[1], c = 2;
while (c <= m) {
   if (c < m && v[c] < v[c+1]) ++c;
   if (x >= v[c]) break;
   v[c/2] = v[c];
   c *= 2; }
v[c/2] = x;
```

4. Recursive. Write a recursive version of the function `sieve`.
5. ★ Generalized sieve. Suppose that an array `v[1..m]` is a heap except perhaps for indices `2p` and `2p+1`. Write a function `sieve2` that will receive `v[1..m]` and `p` and transform the array into a heap.
6. ★ Building a heap faster. Show that the following function has the same effect as [`buildheap`](), that is, it transforms array `v[1..m]` into a heap:

```
void buildheap2 (int m, int v[]) {
   for (int p = m/2; p >= 1; --p)
      sieve2 (p, m, v);
}
```

   Show that `buildheap2` does at most `(m lg(m))/2` comparisons between elements of the array. Refine your analysis to show that the function actually does at most `m` comparisons.
7. Does the following code fragment transform array `v[1..m]` into a heap?

```
for (int p = 1; p <= m/2; ++p)
   sieve2 (p, m, v);
```

8. Priority Queue. A *priority queue* is a set of `n` objects (numbers, for example) subject to two operations: (1) deletion of a largest element and (2) insertion of a new object. If the set is kept in a heap, these two operations can be made very fast. Implement the two operations so that each takes time proportional to log `n` in the worst case.

The Heapsort algorithm

We can now put together all the pieces discussed above and write an algorithm that will rearrange an array `v[1..n]` in increasing order. The algorithm has two phases: the first transforms the array into a heap and the second pulls elements from the heap in decreasing order. (See a draft of the algorithm.)

```
// Rearranges the elements of array v[1..n]
// in increasing order.

void
heapsort (int n, int v[])
{
   buildheap (n, v);
   for (int m = n; m >= 2; --m) {
      swap (v[1], v[m]);
      sieve (m-1, v);
   }
}
```

At the beginning of each iteration of the "`for`", the following invariant properties hold:

- `v[1..m]` is a heap,
- `v[1..m]` ≤ `v[m+1..n]`,
- `v[m+1..n]` is in increasing order, and

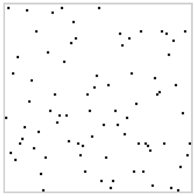
- `v[1..n]` is a permutation of the original array.

The expression "`v[1..m]` ≤ `v[m+1..n]`" is a shorthand for "each element of `v[1..m]` is smaller than or equal to every element of `v[m+1..n]`".

1				heap				m			increasing		n
888	777	666	555	444	333	222	111	000	999	999	999	999	999
				small elements							large elements		

It follows that `v[1..n]` will be in increasing order when `m` becomes equal to `1`. This shows that the algorithm is correct.

Animations of Heapsort



The animation at right (copied from Simon Waldherr / Golang Sorting Visualization) shows Heapsort running on an array `v[0..79]` of positive numbers. Each element `v[i]` of the array is represented by a point (`i`,`v[i]`). (For some reason, the animation does not execute the last two iterations.)

Here is a sample of other animations:

- Heapsort, Wikipedia entry.
- Heapsort animation, by David Galles (University of San Francisco).
- Animation of 15 sorting algorithms, by Timo Bingmann, on YouTube.
- Sorting Algorithms Animations on Toptal.

Exercises 5

1. Use the `heapsort` function to sort the array 16 15 14 13 12 11 10 9 8 7 6 5 4 . Show the state of the array at the beginning of each iteration.
2. Describe the state of the array at the beginning of a generic iteration of the Heapsort algorithm?
3. Correctness check. Write a program to test, experimentally, the correctness of your implementation of the Heapsort algorithm. (See the analogous exercise for Insertionsort.)
4. Does the `heapsort` function produce a stable rearrangement of the array? In other words, does it preserve the relative order of elements that have the same value?
5. Write a function with prototype `heap_sort (int *v, int n)` that will rearrange an array `v[0..n-1]` (notice the indices) in increasing order. (This can be done by an appropriate call to the `heapsort` function.)
6. Min-Heap. Write a function that will rearrange an array `v[1..n]` in *decreasing* order. (Hint: Adapt the definition of heap and rewrite the `sieve` function.)
7. Emulate a heap by means of a set of cells interconnected by pointers. Each cell will have four fields: a `value` and three pointers, one pointing the parent of the cell, another pointing the right child, and another pointing the left child. Write an appropriate version of the `sieve` function. (See the *Binary trees* chapter.)

Performance of Heapsort

How many comparisons between elements of the array does the `heapsort` function execute? The auxiliary function `buildheap` does at most `n lg(n)` comparisons. Next, the `sieve` function is called approximately `n` times and each of these calls does at most `2 lg(n)` comparisons. Hence, the total number of comparisons is at most

`3 n lg(n).`

The *time* consumed by `heapsort` is proportional to the number of comparisons between elements of the array, and therefore proportional to `n` log `n` in the worst case. (But the proportionality factor is larger than that of Mergesort and Quicksort.)

Exercises 6

1. PERFORMANCE TEST. Write a program to time your implementation of the Heapsort algorithm. (See the analogous exercise for Mergesort.)

See chapter 14 of the *Programming Pearls* by Jon Bentley.

Updated 2019-02-02
https://www.ime.usp.br/~pf/algorithms/

CS-IME-USP