Heap-construction programs

Edelkamp, Stefan; Elmasry, Amr; Katajainen, Jyrki

Publication date:
2016

Document version
Publisher's PDF, also known as Version of record

Document license:
Unspecified

Citation for published version (APA):
Heap-Construction Programs

Stefan Edelkamp\textsuperscript{1}, Amr Elmasry\textsuperscript{2}, and Jyrki Katajainen\textsuperscript{3}

\textsuperscript{1} Institute for Artificial Intelligence, University Bremen, Am Fallturm 1, 28359 Bremen, Germany; edelkamp@tzi.de
\textsuperscript{2} Department of Computer Engineering and Systems, Alexandria University, Alexandria 21544, Egypt; elmasry@alexu.edu.eg
\textsuperscript{3} Department of Computer Science, University of Copenhagen, Universitetsparken 5, 2100 Copenhagen East, Denmark; jyrki@di.ku.dk

Abstract. This report together with an accompanying tar ball contains the source code of the programs described and benchmarked in the paper “Heap Construction—50 Years Later”. The programs in this package describe the state of the art in heap construction in 2016.

Keywords. Data structures, binary heaps, heap construction, algorithm engineering, element comparisons, element moves, cache misses, branch mispredictions

Copyright notice

Copyright © 2000–2016 by The authors and Performance Engineering Laboratory (University of Copenhagen)

The programs included in the CPH STL are placed in the public domain. The files may be freely copied and distributed, provided that no changes whatsoever are made. Changes are permissible only if the modified files are given new names, different from the names of existing files in the CPH STL, and only if the modified files are clearly identified as not being part of the library. Usage of the source code in derived works is otherwise unrestricted.

The authors have tried to produce correct and useful programs, but no warranty of any kind should be assumed.

Release date

2016-11-04
In the paper “Heap Construction—50 Years Later”, the heap-construction programs considered were characterized as follows:

**stl:** The `make_heap` function that came with our g++ compiler. On closer inspection, it was seen to rely on the bottom-up `sift_down` policy [5] (see also [8]). Two of the underlying subroutines passed elements by value, so this resulted in some unnecessary element moves.

**F:** Floyd’s Algol program [3] (F) converted into C++. In `sift_down`, this program employed the hole technique, so element swaps were not used.

**basic GM:** Our implementation of the algorithm of Gonnet and Munro [4] (GM) using a tournament tree. For an input of size $N$, the program used a tournament tree requiring $2N - 1$ extra space for indices and a temporary output area requiring $N$ extra space for elements.

**GM:** A space-efficient implementation of GM. This program could be configured to operate in-place ($O(1)$ extra space) or in-situ ($O(lg N)$ extra space). Both versions used $O(1)$ extra space for elements. The program accepted a tuning parameter $\gamma$ and set the size of the bottom trees to the closest power of two larger than, or equal to, $\gamma lg N / lg lg N$. By default, $\gamma = 32$. The in-place variant used a packed array that could store a sequence of integers of equal length compactly in memory. The in-situ variant stored the offsets in an integer array.

**MR:** A space-efficient implementation of the algorithm of McDiarmid and Reed [6] (MR). The program accepted a tuning parameter $\mu$ and set the size of the bottom trees to the closest power of two larger than, or equal to, $\mu lg N$. By default, $\mu = 16$. As in the previous program, both cache and branch optimizations were applied. All the elements on the `sift_down` path were moved cyclically first after the final position of the new element was known as proposed in [7]. When making a bottom tree into a heap, the indices of the element array from the interval $[0..N)$ and those of the packed array from the interval $[0..S)$ were updated in tandem, which doubled the instruction count for index operations. Also this program could be configured to operate in-place (use a packed array of bit pairs) or in-situ (use an array of bytes).

The following optimization options were considered for Floyd’s program:

- **opt1 [2]:** We made sure that `sift_down` was always called with an odd $N$. This way, inside the inner loop, one easy-to-predict branch could be removed.

- **opt2 [2]:** We interpreted the result of an element comparison as an integer and used this value in normal index arithmetic. This way, inside the inner loop, the hard-to-predict branch in “if (condition) $j \leftarrow j + 1$” could be replaced with an assignment “$j \leftarrow j + (condition)$”.

- **opt3 [2]:** We did not make any element moves when the element at the root stayed in its original location.
opt₄ [1]: We visited the nodes in reverse depth-first order instead of reverse breadth-first order.

opt₅ [2]: We made the construction in a single loop by fusing the two loops in `make_heap` and `sift_down`. Inside this loop, conditional moves were used, but two of the element moves were still made conditionally so that the number of element moves would not increase to $5N$. The outcome of these two conditional branches was predicted reasonably well so it was not worth avoiding these branches.

Portability

In the experiments reported in the paper “Heap Construction—50 Years Later”, in each experiment a C array was used to store the input elements. As in our pseudo-code, to access the element with rank $i$ in an array $A$ one would normally write $A[i]$. However, the parameters given for `std::make_heap` are iterators and for them one has to write $*(A + i)$ instead of $A[i]$. For arrays both expressions are valid and the semantics is the same. In this revised version all the programs have been made portable so that they can be used in the same way as C++ standard-library function `make_heap`. However, be aware that in some cases a portable version can be a bit slower than a version using C arrays directly.

References

## Contents

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>stl.h++</td>
<td>The <code>make_heap</code> function that came with our g++ compiler</td>
<td>6</td>
</tr>
<tr>
<td>f.h++</td>
<td>A translation of Floyd’s Algol program (F) [3] into C++</td>
<td>7</td>
</tr>
<tr>
<td>opt1.h++</td>
<td>A modification of F employing opt1 [2]</td>
<td>8</td>
</tr>
<tr>
<td>opt2.h++</td>
<td>A modification of F employing opt1 and opt2 [2]</td>
<td>10</td>
</tr>
<tr>
<td>opt3.h++</td>
<td>A modification of F employing opt1, opt2, and opt3 [2]</td>
<td>11</td>
</tr>
<tr>
<td>opt4.h++</td>
<td>A modification of F employing opt1, opt2, opt3, and opt4 [1]</td>
<td>13</td>
</tr>
<tr>
<td>opt5.h++</td>
<td>A branch-optimized version of F taken from [2]</td>
<td>15</td>
</tr>
<tr>
<td>basic_gm.h++</td>
<td>Our implementation of the algorithm of Gonnet and Munro (GM) [4] using a tournament tree</td>
<td>17</td>
</tr>
<tr>
<td>gm.h++</td>
<td>Our implementation of GM that can be configured to operate in-place or in-situ</td>
<td>23</td>
</tr>
<tr>
<td>tuned_gm.h++</td>
<td>A tuned version of GM that operates in-situ, but uses a logarithmic amount of additional space for elements too</td>
<td>30</td>
</tr>
<tr>
<td>mr.h++</td>
<td>Our implementation of the algorithm of McDiarmid and Reed (MR) [6] that can be configured to operate in-place or in-situ</td>
<td>37</td>
</tr>
<tr>
<td>packed_array.h++</td>
<td>A data structure that can store a sequence of integers of equal length compactly in memory</td>
<td>41</td>
</tr>
<tr>
<td>driver.c++</td>
<td>The driver used for measuring the CPU time, the number of element comparisons, and the number of element moves</td>
<td>44</td>
</tr>
<tr>
<td>test-driver.c++</td>
<td>The driver used for unit testing</td>
<td>50</td>
</tr>
<tr>
<td>makefile</td>
<td>This <code>makefile</code> can be used to redo some of the experiments described in the paper; the scripts to derive the results for the number of instructions, cache misses, and branch mispredictions are not included in this catalogue</td>
<td>52</td>
</tr>
</tbody>
</table>
Heap-construction programs

#include <iterator> // std::iterator
#include <utility>  // std::move

namespace stl {

    template <typename iterator, typename index, typename T, typename comparator>
    void push_heap(iterator a, index hole, index top, T value, comparator less) {
        index parent = (hole - 1) / 2;
        while (hole > top and less(*(a + parent), value)) {
            *(a + hole) = std::move(*(a + parent));
            hole = parent;
            parent = (hole - 1) / 2;
        }
        *(a + hole) = std::move(value); 
    }

    template <typename iterator, typename index, typename T, typename comparator>
    void adjust_heap(iterator a, index hole, index n, T value, comparator less) {
        index const top = hole;
        index second = hole;
        while (second < (n - 1) / 2) {
            second = 2 * (second + 1);
            if (less(*a + second), *(a + (second - 1))) {
                second--;
            }
            *(a + hole) = std::move(*(a + second));
            hole = second;
        }
        if ((n bitand 1) == 0 and second == (n - 2) / 2) {
            second = 2 * (second + 1);
            *(a + hole) = std::move(*(a + (second - 1)));
            hole = second - 1;
        }
        stl::push_heap(a, hole, top, std::move(value), less);
    }

    template <typename iterator, typename comparator>
    void make_heap(iterator a, iterator past, comparator less) {
        using element = typename std::iterator_traits<iterator>::value_type;
        using index = typename std::iterator_traits<iterator>::difference_type;
        if (past - a < 2) {
            return;
        }
    }

}

stl.h++
index const n = past - a;
index parent = (n - 2) / 2;
while (true) {
    element value = std::move(*(a + parent));
    stl::adjust_heap(a, parent, n, std::move(value), less);
    if (parent == 0) {
        return;
    }
    -- parent;
}

f.h++

// Floyd's original Algol program translated into C++

#include <cstdlib>  // std::size_t
#include <iterator>  // std::iterator_traits
#include <utility>   // std::move

namespace f {

constexpr std::size_t root() {
    return 0;
}

template <typename index>
index parent(index i) {
    return (i - 1) / 2;
}

template <typename index>
index left_child(index i) {
    return (i << 1) + 1;
}

template <typename iterator, typename index, typename comparator>
void sift_down(iterator a, index i, index n, comparator less) {
    using element = typename std::iterator_traits<iterator>::
        value_type;
    element x = std::move(*(a + i));
    loop:
        index j = left_child(i);
        if (j < n) {
            if (j < (n - 1)) {
                if (less(*(a + j), *(a + (j + 1)))) {
                    j = j + 1;
                }
            }
            if (less(x, *(a + j))) {

```cpp
*(a + i) = std::move(*(a + j));
i = j;
goto loop;
}
*(a + i) = std::move(x);
}

template<
type
>
void make_heap(iterator first, iterator past, comparator less)
{
using index = std::size_t;
index const n = past - first;
if (n < 2) {
    return;
}
index i = parent(n - 1);
while (true) {
    sift_down(first, i, n, less);
    if (i == root()) {
        break;
    }
    --i;
}
}

opt1.h++

// Floyd's heap-construction program
// Optimization: one easy-to-predict branch removed
#include <algorithm> // std::make_heap
#include <cstddef> // std::size_t
#include <iterator> // std::iterator_traits
#include <utility> // std::move

namespace opt1 {
const expr std::size_t root() {
    return 0;
}

template<
type
>
index parent(index i) {
    return (i - 1) / 2;
}

template<
type
>
index left_child(index i) {
    return (i << 1) + 1;
}
}
template <typename iterator, typename index, typename comparator>
void sift_up(iterator a, index j, comparator less) {
    using element = typename std::iterator_traits<iterator>::value_type;
    element in = std::move(*(a + j));
    while (j > root()) {
        index i = parent(j);
        if (less(*(a + i), in)) {
            *(a + j) = std::move(*(a + i));
            j = i;
        } else {
            break;
        }
    } *(a + j) = std::move(in);
}

template <typename iterator, typename index, typename comparator>
void sift_down(iterator a, index i, index n, comparator less) {
    using element = typename std::iterator_traits<iterator>::value_type;
    element in = std::move(*(a + i));
    loop:
        index j = left_child(i);
        if (j < n) {
            if (less(*(a + j), *(a + (j + 1)))) {
                j = j + 1;
            } if (less(in, *(a + j))) {
                *(a + i) = std::move(*(a + j));
                i = j;
                goto loop;
            }
        }
    *(a + i) = std::move(in);
}

template <typename iterator, typename comparator>
void make_heap(iterator first, iterator past, comparator less) {
    using index = std::size_t;
    index const n = past - first;
    if (n < 3) {
        std::make_heap(first, past, less);
        return;
    } index const m = (n bitand 1) ? n - 1;
    index i = parent(m - 1);
    while (true) {
        sift_down(first, i, m, less);
        if (i == root()) {
break;
            -- i;
        }
        sift_up(first, n - 1, less);
    }
}

opt2.h++

// Floyd's heap-construction program

// Optimizations: 1) one easy-to-predict branch removed; 2) one hard-to-predict branch removed
#include <cstddef> // std::size_t
#include <iterator> // std::iterator_traits
#include <utility> // std::move

namespace opt2 {

    constexpr std::size_t root() {
        return 0;
    }

    template<typename index>
    index parent(index i) {
        return (i - 1) / 2;
    }

    template<typename index>
    index left_child(index i) {
        return (i << 1) + 1;
    }

    template<typename iterator, typename index, typename comparator>
    void sift_up(iterator a, index j, comparator less) {
        using element = typename std::iterator_traits<iterator>::value_type;
        element in = std::move(*a + j);
        while (j > root()) {
            index i = parent(j);
            if (less(*a + i, in)) {
                *a + j = std::move(*a + i);
                j = i;
            } else {
                break;
            }
        }
        *(a + j) = std::move(in);
    }
}
template <typename iterator, typename index, typename comparator>
void sift_down(iterator a, index i, index n, comparator less) {
    using element = typename std::iterator_traits<iterator>::value_type;
    element in = std::move(*(a + i));
    loop:
        index j = left_child(i);
        if (j < n) {
            j = j + less(*(a + j), *(a + (j + 1)));
            if (less(in, *(a + j))) {
                *(a + i) = std::move(*(a + j));
                i = j;
                goto loop;
            }
        }
    *(a + i) = std::move(in);
}

template <typename iterator, typename comparator>
void make_heap(iterator first, iterator past, comparator less) {
    using index = std::size_t;
    index const n = past - first;
    if (n < 3) {
        std::make_heap(first, past, less);
        return;
    }
    index const m = (n bitand 1) ? n : n - 1;
    index i = parent(m - 1);
    while (true) {
        sift_down(first, i, m, less);
        if (i == root()) {
            break;
        }
        --i;
    }
    sift_up(first, n - 1, less);
}

#include <algorithm>  // std::make_heap
#include <cstdlib>    // std::size_t
#include <iterator>   // std::iterator_traits
#include <utility>    // std::move

opt3.h++
namespace opt3 {

constexpr std::size_t root() {
    return 0;
}

template <typename index>
index parent(index i) {
    return (i - 1) / 2;
}

template <typename index>
index left_child(index i) {
    return (i << 1) + 1;
}

template <typename iterator, typename index, typename comparator>
void sift_up(iterator a, index j, comparator less) {
    using element = typename std::iterator_traits<iterator>::value_type;
    element in = std::move(*(a + j));
    while (j > root()) {
        index i = parent(j);
        if (less(*(a + i), in)) {
            *(a + j) = std::move(*(a + i));
            j = i;
        } else {
            break;
        }
    }
    *(a + j) = std::move(in);
}

template <typename iterator, typename index, typename comparator>
void sift_down(iterator a, index i, index n, comparator less) {
    using element = typename std::iterator_traits<iterator>::value_type;
    index j = left_child(i);
    j = j + less(*(a + j), *(a + (j + 1)));
    if (not less(*(a + i), *(a + j))) {
        return;
    }
    element in = std::move(*(a + i));
    *(a + i) = std::move(*(a + j));

    for (loop: 
        i = j;
        j = left_child(i);
        if (j < n) {
            j = j + less(*(a + j), *(a + (j + 1)));
            if (less(in, *(a + j))) {
                break;
            }
        }
        *(a + j) = std::move(in);
    }
}
```cpp
template <typename iterator, typename comparator>
void make_heap(iterator first, iterator past, comparator less) {
    using index = std::size_t;
    index const n = past - first;
    if (n < 3) {
        std::make_heap(first, past, less);
        return;
    }
    index const m = (n bitand 1) ? n : n - 1;
    index i = parent(m - 1);
    while (true) {
        sift_down(first, i, m, less);
        if (i == root()) {
            break;
        }
        --i;
    }
    sift_up(first, n - 1, less);
}
}

// Floyd's heap-construction program
// Optimizations: 1) one easy-to-predict branch removed; 2) one
// hard-to-predict branch removed; 3) no element moves done in
// sift_down if the new element can stay at the root; 4) nodes are
// visited in depth-first order to improve the cache behaviour

#define include <algorithm> // std::make_heap
#define include <cmath> // ilogb
#define include <cstdlib> // std::size_t
#define include <iterator> // defines std::iterator_traits
#define include <utility> // std::move std::swap

namespace opt4 {

    constexpr std::size_t root() {
        return 0;
    }

    template <typename index>
    index parent(index i) {
        return (i - 1) / 2;
    }

```
template <typename index>
index left_child(index i) {
    return (i << 1) + 1;
}

template <typename iterator, typename index, typename comparator>
void sift_up(iterator a, index j, comparator less) {
    using element = typename std::iterator_traits<iterator>::
        value_type;
    element in = std::move(*a + j);
    while (j > root()) {
        index i = parent(j);
        if (less(*a + i, in)) {
            *(a + j) = std::move(*(a + i));
            j = i;
        } else {
            break;
        }
    }
    *(a + j) = std::move(in);
}

template <typename iterator, typename index, typename comparator>
void sift_down_1_3(iterator a, index i, index n, comparator less) {
    index j = left_child(i);
    if (j ≥ n) {
        return;
    } j = j + less(*(a + j), *(a + (j + 1)));
    if (not less(*(a + i), *(a + j))) {
        return;
    } std::swap(*(a + i), *(a + j));
}

template <typename iterator, typename index, typename comparator>
void sift_down(iterator a, index i, index n, comparator less) {
    using element = typename std::iterator_traits<iterator>::
        value_type;
    index j = left_child(i);
    j = j + less(*(a + j), *(a + (j + 1)));
    if (not less(*(a + i), *(a + j))) {
        return;
    } element in = std::move(*(a + i));
    loop:
    *(a + i) = std::move(*(a + j));
    i = j;
    j = left_child(j);
```cpp
if (j < n) {
    j = j + less(*(a + j), *(a + (j + 1))); 
    if (less(in, *(a + j))) {
        goto loop; 
    }
}
*(a + i) = std::move(in);
}

template <typename iterator, typename comparator>
void make_heap(iterator first, iterator past, comparator less) {
    using index = std::size_t;
    index const n = past - first;
    if (n < 5) {
        std::make_heap(first, past, less);
        return;
    }
    index const m = (n bitand 1) ? n : n - 1;
    index j = (1 << ilogb(m)) - 2;
    index const i = parent(j + 1);
    while (j ≥ i) {
        sift_down_1_3(first, j, m, less);
        index z = j;
        while ((z bitand 1) == 1) {
            z = parent(z);
            sift_down(first, z, m, less);
        }
        --j;
    }
    sift_up(first, n - 1, less);
}

opt5.h++

// A program that constructs a binary heap in a single loop
// Optimizations: 1) branches avoided in general; 2) a compiler
// should use conditional moves, not conditional branches;
// 3) construction is done in a single loop to avoid the branch
// mispredictions caused by the inner loop when stepping out of it
#include <algorithm> // std::make_heap
#include <cstdlib> // std::size_t
#include <iterator> // std::iterator_traits
#include <utility> // std::move
namespace opt5 {
    constexpr std::size_t root() {
        return 0;
    }
}
template <typename index>
index parent(index i) {
    return (i - 1) / 2;
}

template <typename index>
index left_child(index i) {
    return (i << 1) + 1;
}

template <typename iterator, typename index, typename comparator>
void sift_up(iterator a, index j, comparator less) {  
    using element = typename std::iterator_traits<iterator>::
        value_type;
    element in = std::move(*(a + j));
    while ((j > root())) {
        index i = parent(j);
        if (less(*(a + i), in)) {
            *(a + j) = std::move(*(a + i));
            j = i;
        } else {
            break;
        }
    }
    *(a + j) = std::move(in);
}

template <typename iterator, typename comparator>
void make_heap(iterator first, iterator past, comparator less) {  
    using index = std::size_t;
    using element = typename std::iterator_traits<iterator>::
        value_type;
    index const n = past - first;
    if (n < 3) {
        std::make_heap(first, past, less);
        return;
    }
    index const m = (n bitand 1) ? n : n - 1;
    index i = parent(m - 1);
    index j = i;
    index hole = j;
    element in = std::move(*(first + j));
    while (true) {
        hole = (i == j) ? j : hole;
        if (i == j) {
            in = std::move(*(first + j));
        }
        j = left_child(j);
        j += less(*(first + j), *(first + (j + 1)));
        *(first + hole) = std::move(*(first + j));
hole = less(in, *(first + j)) ? j : hole;
bool done = (left_child(j) ≥ m);
if (done) {
    *(first + hole) = std::move(in);
}
if (done and i == root()) {
    break;
}
i = (done) ? i - 1 : i;
j = (done) ? i : j;
sift_up(first, n - 1, less);
}

basic_gm.h++

// Gonnet & Munro

#include <algorithm> // std::copy std::make_heap
#include <cstdlib> // std::size_t
#include <functional> // std::less
#include <iterator> // std::iterator_traits

namespace basic_gm {

constexpr std::size_t root() {
    return 0;
}

template <typename index>
index parent(index i) {
    return (i - 1) / 2;
}

template <typename index>
index left_child(index i) {
    return 2 * i + 1;
}

template <typename index>
index right_child(index i) {
    return 2 * i + 2;
}

template <typename index>
index odd(index i) {
    return (i bitand 1) == 1;
}

template <typename index>
index even(index i) {

return (i & 1) == 0;
}

template <typename index>
index sibling(index i)
{
    return i + odd(i) - even(i);
}

template <typename index>
bool is_inside(index i, index n)
{
    return i < n;
}

template <typename integer>
bool is_power_of_2(integer n)
{
    return __builtin_popcount(n) == 1;
}

template <typename iterator, typename index, typename comparator>
void sift_down(iterator a, index i, index n, comparator less)
{
    using value_type = typename std::iterator_traits<iterator>::value_type;
    element x = std::move(*a + i);
    index j = left_child(i);
    while (is_inside(j, n))
    {
        if (is_inside(j + 1, n))
        {
            j = j + less(*a + j), *(a + (j + 1));
        }
        if (not less(x, *(a + j)))
        {
            break;
        }
        *(a + i) = std::move(*(a + j));
        i = j;
        j = left_child(i);
    }
    *(a + i) = std::move(x);
}

template <typename iterator, typename index, typename comparator>
void make_heap(iterator a, index K, index N, comparator less)
{
    if (not is_inside(left_child(K), N))
    {
        return;
    }
    index L = K;
    while (is_inside(left_child(L), N))
    {
        L = left_child(L);
    }
    index R = K;
    while (is_inside(right_child(R), N))
    {
        R = right_child(R);
    }
    if (R < L)
R = right_child(R);
}
do {
  L = parent(L);
  R = parent(R);
  index J = R;
  while (true) {
    sift_down(a, J, N, less);
    if (J == L) {
      break;
    }
    J -= 1;
  }
} while (L != R);
}

template<typename input, typename index, typename output>
void copy_heap(input a, index k, index h, output o) {
  index i = k;
  while (true) {
    index j = i;
    while (j <= k) {
      *(o + j) = std::move(*(a + j));
      ++j;
    }
    if (h == 0) {
      return;
    }
    i = left_child(i);
    k = right_child(k);
  }
}

template<typename tournament, typename index>
void populate_tournament(tournament t, index n, index K, index L, index R, index E) {
  index i = parent(n);
  t[i] = E;
  ++i;
  index I = R;
  while (I >= L) {
    t[i] = I;
    ++i;
    index Z = I;
    while (odd(Z) and Z != K) {
      Z = parent(Z);
      t[i] = Z;
      ++i;
    }
    --I;
  }
}
template <typename tournament, typename index, typename input, typename comparator>
void run_tournament(tournament t, index n, input a, comparator less)
{
    index i = parent(n - 1);
    while (true) {
        index C = t[left_child(i)];
        index D = t[right_child(i)];
        t[i] = less(*(a + C), *(a + D)) ? D : C;
        if (i == root()) {
            break;
        }
        -- i;
    }
}

template <typename tournament, typename index, typename input, typename comparator>
void update(tournament t, index j, index h, index E, input a, comparator less)
{
    index M = t[j]; // location of the old champion
    index k = j;
    do {
        k = left_child(k);
        k = k + (t[k] != M);
        -- h;
    } while (h != 0);
    t[k] = E;
    do {
        k = parent(k);
        index C = t[left_child(k)];
        index D = t[right_child(k)];
        t[k] = less(*(a + C), *(a + D)) ? D : C;
    } while (k != j);
}

template <typename tournament, typename index, typename input, typename output, typename comparator>
index handle_base_case(tournament t, index j, input a, output o, index J, comparator less)
{
    index champion = t[j];
    *(o + J) = std::move(*(a + champion));
    index k = left_child(j);
    index l = k + (t[k] == champion);
    index w = sibling(l);
    index L = left_child(J);
    index W = right_child(J);
    index second = t[l];
    *(o + L) = std::move(*(a + second));
    k = left_child(l);
}
index ll = k + (t[k] == second);
index lv = sibling(ll);
*(o + left_child(L)) = std::move(*(a + t[ll]));
k = left_child(lw);
index lw = k + (t[k] == second);
*(o + right_child(L)) = std::move(*(a + t[lw]));
k = left_child(ll);
index lll = k + (t[k] == t[ll]);
k = left_child(w);
index wll = k + (t[k] == champion);
index ww = sibling(wl);
k = left_child(ww);
index wwl = k + (t[k] == champion);
index excess = t[wwl];
k = left_child(ww);
index wll = k + (t[k] == t[wwl]);
*(o + left_child(W)) = std::move(*(a + t[wwl]));
index U = t[wwl];
index V = t[lll];
if (less(*(a + U), *(a + V))) {
    *(o + W) = std::move(*(a + V));
    *(o + right_child(W)) = std::move(*(a + U));
} else {
    *(o + W) = std::move(*(a + U));
    *(o + right_child(W)) = std::move(*(a + V));
}
return excess;

template <typename tournament, typename index, typename input,
          typename output, typename comparator>
index convert_tournament(tournament t, index j, index h, input a,
output o, index J, comparator less) {
    // J: current output location in the heap
    // j: current root in the tournament tree
    if (h == 3) {
        return handle_base_case(t, j, a, o, J, less);
    }
    index champion = t[j];
    *(o + J) = std::move(*(a + champion));
    index k = left_child(j);
    k = k + (t[k] == champion);
    index excess = convert_tournament(t, k, h - 1, a, o, left_child(J), less);
    k = sibling(k);
    update(t, k, h - 1, excess, a, less);
    return convert_tournament(t, k, h - 1, a, o, right_child(J), less );
}
template <typename input, typename index, typename tournament, 
    typename output, typename comparator>
index make_heap(input a, index K, index N, index E, tournament t, 
    output o, comparator less) {
    if (not is_inside(K, N)) {
        return E;
    }
    index h = 0;
    index L = K;
    while (is_inside(left_child(L), N)) {
        ++h;
        L = left_child(L);
    }
    index height = h;
    index R = K;
    h = 0;
    while (is_inside(right_child(R), N)) {
        ++h;
        R = right_child(R);
    }
    index n = 1 << (height + 1);
    if (n < 8) {
        copy_heap(a, K, h, o);
        make_heap(o, K, N, less);
        return E;
    }
    populate_tournament(t, 2 * n - 1, K, L, R, E);
    run_tournament(t, 2 * n - 1, a, less);
    E = convert_tournament(t, root(), height + 1, a, o, K, less);
    return E;
}

template <typename iterator, typename comparator>
void make_heap(iterator a, iterator past, comparator less) {
    using index = std::size_t;
    using element = typename std::iterator_traits<iterator>::
        value_type;
    index const n = past - a;
    if (n < 2) {
        return;
    }
    index* t = (index*) malloc(((4 * n) / 3) * sizeof(index));
    element* o = (element*) malloc(n * sizeof(element)); // no
    element moves
    index i = n; // invariant: everything moved up to this point
    if (odd(n)) {
        *(o + n - 1) = std::move(*(a + n - 1));
        i = n - 1;
    }
    do {
        index j = sibling(i);
        i = parent(i);
index excess = i;
excess = make_heap(a, j, n, excess, t, o, less);
o[i] = std::move(*a + excess));
} while (i ≠ root());
i = n - 1;
do {
i = parent(i);
sift_down(o, i, n, less);
} while (i ≠ root());
for (index i = 0; i ≠ n; ++i) {
*(a + i) = std::move(*(o + i));
}
free(t);
free(o);
}
}
gm.h++

// In-situ/In-place Gonnet & Munro
// # define IN_PLACE

#include <stddef> // std::size_t
#include <iterator> // std:iterator_traits
#include <vector>
#include <utility> // std::move

#ifndef FACTOR
#define FACTOR 32
#endif

namespace gm {

constexpr std::size_t root() {
  return 0;
}

template <typename index>
index parent(index i) {
  return (i - 1) / 2;
}

template <typename index, typename height>
index ancestor(index i, height h) {
  return (i + 1) / (1 << h) - 1;
}
template<typename index>
index left_child(index i) {
  return 2 * i + 1;
}

template<typename index>
index right_child(index i) {
  return 2 * i + 2;
}

template<typename index>
index odd(index i) {
  return (i bitand 1) == 1;
}

template<typename index>
index even(index i) {
  return (i bitand 1) == 0;
}

template<typename index>
index sibling(index i) {
  return i + odd(i) - even(i);
}

template<typename index>
bool is_inside(index i, index n) {
  return i < n;
}

template<typename index>
bool general_case(index i, index n) {
  return left_child(left_child(left_child(i))) < parent(n);
}

std::size_t ilogb(std::size_t x) {
  asm("bsr \%0, \%0"
       : "=r"(x)
       : "0" (x)
       );
  return x;
}

template<typename offset, typename index>
index offset_to_index(offset i, index J) {
  if (index(i) != index(0)) {
    index h = ilogb(index(i));
    return (1 << h) * J + index(i) - 1;
  }
  return parent(J);
}
template<typename iterator, typename index, typename comparator>
void sift_down(iterator a, index i, index n, comparator less) {
    using element = typename std::iterator_traits<iterator>::
        value_type;
    element x = std::move(*(a + i));
    index j = left_child(i);
    while (is_inside(j, n)) {
        if (is_inside(j + 1, n)) {
            j = j + less(*(a + j), *(a + (j + 1)));
        }
        if (not less(x, *(a + j))) {
            break;
        }
        *(a + i) = std::move(*(a + j));
        i = j;
    }
    *(a + i) = std::move(x);
}

template<typename iterator, typename index, typename comparator>
void make_heap(iterator a, index K, index N, comparator less) {
    index L = K;
    while (is_inside(left_child(L), N)) {
        L = left_child(L);
    }
    index R = K;
    while (is_inside(right_child(R), N)) {
        R = right_child(R);
    }
    do {
        L = parent(L);
        R = parent(R);
        index J = R;
        while (true) {
            sift_down(a, J, N, less);
            if (J == L) {
                break;
            }
            J -= 1;
        }
    } while (L != R);
}

template<typename tree, typename offset>
void populate_tournament(tree& t, offset n) {
    offset i = 0;
    for (offset j = parent(n); is_inside(j, n); ++j) {
        t[j] = i;
    }
i = i + 1;
}

template<typename tree, typename offset, typename iterator,  
          typename index, typename comparator>
void run_tournament(tree& t, offset n, iterator a, index J,  
                      comparator less) {
  offset j = parent(n - 1);
  while (true) {
    offset k = t[left_child(j)];
    index L = offset_to_index(k, J);
    k = t[right_child(j)];
    index R = offset_to_index(k, J);
    k = left_child(j) + less(*a + L), *(a + R));
    t[j] = t[k];
    if (j == root<offset>()) {
      break;
    }
    j = j - 1;
  }
}

template<typename tree, typename offset, typename iterator,  
          typename index, typename comparator>
void update(tree& t, offset i, offset n, offset excess, iterator a,  
            index J, comparator less) {
  offset j = i; // root of the subtree considered
  auto top = t[j]; // offset of the old top
  t[j] = excess;
  while (is_inside(left_child(j), n)) {
    j = left_child(j) + (t[right_child(j)] == top);  
    t[j] = excess;
  }
  while (j != i) {
    j = parent(j);
    offset k = t[left_child(j)];
    index L = offset_to_index(k, J);
    k = t[right_child(j)];
    index R = offset_to_index(k, J);
    k = left_child(j) + less(*a + L), *(a + R));
    t[j] = t[k];
  }
}

template<typename tree, typename offset, typename permutation,  
          typename iterator, typename index, typename comparator>
offset handle_base_case(tree& t, offset j, permutation& sigma,  
                        iterator a, index J, comparator less) {
  auto champion = t[j];
  sigma[j + 1] = champion;
  offset k = left_child(j);
offset l = k + (t[k] == champion);
offset w = sibling(l);
auto finalist = t[l];
sigma[l + 1] = finalist;
k = left_child(l);
offset ll = k + (t[k] == finalist);
offset lw = sibling(ll);
sigma[ll + 1] = t[ll];
k = left_child(lw);
offset llw = k + (t[lw] == t[k]);
sigma[lw + 1] = t[llw];
k = left_child(ll);
offset lll = k + (t[ll] == t[k]);
k = left_child(w);
offset wll = k + (t[k] == champion);
offset ww = sibling(wll);
k = left_child(ww);
offset wwl = k + (t[k] == champion);
offset excess = t[wwl];
k = left_child(wl);
auto semifinalist = t[wl];
offset wll = k + (t[k] == semifinalist);
sigma[wl + 1] = t[wl];
index K = offset_to_index(semifinalist, J);
index L = offset_to_index(t[lll], J);
bool line = less(*(a + K), *(a + L));
sigma[w + 1] = (line) ? t[lll] : semifinalist;
sigma[ww + 1] = (line) ? semifinalist : t[lll];
return excess;
}

template<typename tree, typename offset, typename permutation,
  typename iterator, typename index, typename comparator>
void convert_tournament(tree& t, offset n, permutation& sigma,
  iterator a, index J, comparator less) {

#ifdef IN_PLACE
  cphstl::packed_array stack(ilogb(n));
  stack.resize(ilogb(n));
#else
  std::vector<offset> stack;
  stack.resize(ilogb(n));
#endif

offset s = 0; // stack empty
offset j = root<offset>();
while (true) {
  offset k = t[j];
sigma[j + 1] = k;
  if (general_case(j, n)) {
    offset loser = left_child(j) + (t[j] == t[left_child(j)]);
    offset winner = left_child(j) + (t[j] != t[left_child(j)]);
stack[s] = winner;
++s;
    j = loser;
}
else {
    offset excess = handle_base_case(t, j, sigma, a, J, less);
    if (s ≠ 0) {
        -- s;
        j = stack[s];
        update(t, j, n, excess, a, J, less);
    }
    else {
        sigma[0] = excess;
        return;
    }
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
)
}
)
}
)
})
}
)
}
)
}
)
}
)
}
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
template <typename iterator, typename comparator>
void make_heap(iterator a, iterator past, comparator less) {
    using index = std::size_t;
    index n = past - a;
    if (n < 2) {
        return;
    }
    index lg_n = ilogb(n);
    index lg_lg_n = ilogb(std::max(index(2), lg_n));
    index s = 8;
    index h = 3;
    while (s < FACTOR * lg_n / lg_lg_n) {
        s = 2 * s;
        h = h + 1;
    }
    if (n < 2 * s) {
        make_heap(a, root(), n, less);
        return;
    }
    #ifdef IN_PLACE
    cphstl::packed_array sigma(h + 1);
    sigma.resize(2 * s);
    cphstl::packed_array t(h + 1);
    t.resize(4 * s);
    cphstl::packed_array b(1);
    b.resize(2 * s);
    #else
    std::vector<int> sigma;
    sigma.resize(2 * s);
    std::vector<int> t;
    t.resize(4 * s);
    std::vector<unsigned char> b;
    b.resize(2 * s);
    #endif
    index const R = (1 << lg_n) - 1;
    index K = ancestor(R - 1, h - 1);
    index J = ancestor(n - 1, h - 1);
    index I = parent(K + 1);
    if (I < J) {
        J = parent(J);
    }
    while (K > J) {
        populate_tournament(t, 2 * s - 1);
        run_tournament(t, 2 * s - 1, a, K, less);
        convert_tournament(t, 2 * s - 1, sigma, a, K, less);
        permute_in_place(sigma, s, a, K, b);
        for (index Z = K; (Z bitand 1) == 1; ) {
            Z = parent(Z);
        }
    }
}
sift_down(a, Z, n, less);
}  // K;
}  
make_heap(a, J, n, less);
for (index Z = J; (Z bitand 1) == 1; ) {
    Z = parent(Z);
    sift_down(a, Z, n, less);
}
while (K > I) {
    -- K;
    populate_tournament(t, 4 * s - 1);
    run_tournament(t, 4 * s - 1, a, K, less);
    convert_tournament(t, 4 * s - 1, sigma, a, K, less);
    permute_in_place(sigma, 2 * s, a, K, b);
    for (index Z = K; (Z bitand 1) == 1; ) {
        Z = parent(Z);
        sift_down(a, Z, n, less);
    }
}
}
}
}
}
}
}
}

// Gonnet & Munro using extra space for elements
#include <algorithm>  // std::copy std::make_heap
#include <cstddef>  // std::size_t
#include <functional>  // std::less
#include <iterator>  // std::iterator_traits
#include <vector>  // std::vector
#ifndef FACTOR
#define FACTOR 12
#endif
namespace tuned_gm {

constexpr std::size_t root() {
    return 0;
}

template <typename index>
index parent(index i) {
    return (i - 1) / 2;
}

template <typename index, typename height>
index ancestor(index i, height h) {
    return (i + 1) / (1 << h) - 1;
}
template<typename index>
index left_child(index i) {
    return 2 * i + 1;
}

template<typename index>
index right_child(index i) {
    return 2 * i + 2;
}

template<typename index>
index odd(index i) {
    return (i bitand 1) == 1;
}

template<typename index>
index even(index i) {
    return (i bitand 1) == 0;
}

template<typename index>
index sibling(index i) {
    return i + odd(i) - even(i);
}

template<typename index>
bool is_inside(index i, index n) {
    return i < n;
}

template<typename integer>
bool is_power_of_2(integer n) {
    return __builtin_popcount(n) == 1;
}

template<typename iterator, typename index, typename comparator>
void sift_down(iterator a, index i, index n, comparator less) {
    using element = typename std::iterator_traits<iterator>::
        value_type;
    element x = std::move(*(a + i));
    index j = left_child(i);
    while (is_inside(j, n)) {
        if (is_inside(j + 1, n)) {
            j = j + less(*(a + j), *(a + (j + 1)));
        }
        if (not less(x, *(a + j))) {
            break;
        }
        *(a + i) = std::move(*(a + j));
        i = j;
        j = left_child(i);
template <typename iterator, typename index, typename comparator>
void make_heap(iterator a, index K, index N, comparator less) {
    if (not is_inside(left_child(K), N)) {
        return;
    }
    index L = K;
    while (is_inside(left_child(L), N)) {
        L = left_child(L);
    }
    index R = K;
    while (is_inside(right_child(R), N)) {
        R = right_child(R);
    }
    if (R < L) {
        R = right_child(R);
    }
    do {
        L = parent(L);
        R = parent(R);
        index J = R;
        while (true) {
            sift_down(a, J, N, less);
            if (J == L) {
                break;
            }
            J -= 1;
        }
    } while (L != R);
}

template <typename input, typename index, typename output>
void move_heap(input& a, index j, index n, output& o) {
    index i = j;
    index k = j;
    index p = root();
    while (i < n) {
        j = i;
        while (j <= k) {
            *(o + p) = std::move(*(a + j));
            ++j;
            ++p;
        }
        i = left_child(i);
        k = right_child(k);
    }
}

template <typename tournament, typename index>
void populate_tournament(tournament& t, index n, index J, index E)
{
    index i = parent(n);
    t[i] = E;
    ++i;
    index I = J;
    index K = J;
    while (i < n) {
        J = I;
        while (J ≤ K) {
            t[i] = J;
            ++i;
            ++J;
        }
        I = left_child(I);
        K = right_child(K);
    }
}

template <typename tournament, typename index, typename input, typename comparator>
void run_tournament(tournament& t, index n, input const& a, comparator less)
{
    index i = parent(n - 1);
    while (true) {
        index C = t[left_child(i)];
        index D = t[right_child(i)];
        t[i] = less(*(a + C), *(a + D)) ? D : C;
        if (i == root()) {
            break;
        }
        --i;
    }
}

template <typename tournament, typename index, typename input, typename comparator>
void update(tournament& t, index j, index h, index E, input const& a, comparator less) {
    index M = t[j]; // location of the old champion
    index k = j;
    do {
        k = left_child(k);
        k = k + (t[k] ≠ M);
        --h;
    } while (h ≠ 0);
    t[k] = E;
    do {
        k = parent(k);
        index C = t[left_child(k)];
        index D = t[right_child(k)];
        t[k] = less(*(a + C), *(a + D)) ? D : C;
    } while (true);
}
index wll = k + (t[k] == champion);
index w = sibling(l);
index L = left_child(J);
index second = t[l];
*(o + L) = std::move(*(a + second));
k = left_child(l);
index ll = k + (t[k] == second);
index lw = sibling(ll);
*(o + left_child(L)) = std::move(*(a + t[ll]));
k = left_child(lw);
index lw1 = k + (t[k] == second);
*(o + right_child(L)) = std::move(*(a + t[lw1]));
k = left_child(l1);
index l1l = k + (t[k] == second);
k = left_child(w);
index wl1 = k + (t[k] == champion);
index ww = sibling(wl);
k = left_child(ww);
index wwl = k + (t[k] == champion);
index excess = t[wl];
k = left_child(ww);
index wll = k + (t[k] == t[wl]);
*(o + left_child(W)) = std::move(*(a + t[wl1]));
index U = t[wl];
index V = t[l1];
if (less(*(a + U), *(a + V))) {
  *(o + W) = std::move(*(a + V));
  *(o + right_child(W)) = std::move(*(a + U));
}
else {
  *(o + W) = std::move(*(a + U));
  *(o + right_child(W)) = std::move(*(a + V));
}
return excess;
}

template<typename tournament, typename index, typename input,
typeindex output, typename comparator>
index convert_tournament(tournament& t, index j, index h, input& a,
output& o, index P, comparator less) {
// P: current location in the output heap
// j: current root in the tournament tree
if (h == 3) {
    return handle_base_case(t, j, a, o, P, less);
}

index champion = t[j];
*(o + P) = std::move(*a + champion));
index k = left_child(j);
index excess = convert_tournament(t, k, h - 1, a, o, left_child(P), less);
k = sibling(k);
update(t, k, h - 1, excess, a, less);
return convert_tournament(t, k, h - 1, a, o, right_child(P), less);
}

template<typename input, typename index, typename tournament, 
    typename output, typename comparator>
index make_heap(input& a, index K, index N, index E, tournament& t, 
    output& o, index P, comparator less) {
    index height = 0;
    index L = K;
    while (is_inside(left_child(L), N)) {
        ++height;
        L = left_child(L);
    }
    index n = 1 << (height + 1);
    populate_tournament(t, 2 * n - 1, K, E);
    run_tournament(t, 2 * n - 1, a, less);
    E = convert_tournament(t, root(), height + 1, a, o, P, less);
    return E;
}

template<typename iterator, typename comparator>
void make_heap(iterator a, iterator past, comparator less) {
    using index = std::size_t;
    using element = typename std::iterator_traits<iterator>::::value_type;
    index n = past - a;
    index lg_n = ilogb(std::max(index(2), n));
    index s = 8;
    index h = 3;
    while (s < FACTOR * lg_n) {
        s = 2 * s;
        h = h + 1;
    }
    if (n < 2 * s) {
        make_heap(a, root(), n, less);
        return;
    }
    std::vector<index> t;
    t.resize(4 * s);
element* o = (element*) malloc(n * sizeof(element)); // no element moves

index const R = (1 << lg_n) - 1;
index K = ancestor(R - 1, h - 1);
index J = ancestor(n - 1, h - 1);
index I = parent(K + 1);
if (I < J) {
    J = parent(J);
}
index excess = root();
if (K == J) {
}
else if (K == J + 1) {
    make_heap(a, K, n, less);
}
else {
    move_heap(a, K, n, o);
    --K;
    while (K > J) {
        excess = root();
        excess = make_heap(a, K, n, excess, t, a, K + 1, less);
        *(a + root()) = std::move(*(a + excess));
        for (index Z = K + 1; (Z bitand 1) == 1; ) {
            Z = parent(Z);
            sift_down(a, Z, n, less);
        }
        --K;
    }
    excess = root();
    *(o + (s - 1)) = std::move(*(a + excess));
    excess = make_heap(o, root(), s - 1, s - 1, t, a, K + 1, less);
    *(a + root()) = std::move(*(o + excess));
    for (index Z = K + 1; (Z bitand 1) == 1; ) {
        Z = parent(Z);
        sift_down(a, Z, n, less);
    }
}
make_heap(a, J, n, less);
for (index Z = J; (Z bitand 1) == 1; ) {
    Z = parent(Z);
    sift_down(a, Z, n, less);
}
if (I == J) {
}
else if (J == I + 1) {
    make_heap(a, I, n, less);
    for (index Z = I; (Z bitand 1) == 1; ) {
        Z = parent(Z);
        sift_down(a, Z, n, less);
    }
}
else {
    K = J - 1;
    move_heap(a, K, n, o);
    -- K;
    while (K ≥ I) {
        excess = root();
        excess = make_heap(a, K, n, excess, t, a, K + 1, less);
        *(a + root()) = std::move(*(a + excess));
        for (index Z = K + 1; (Z bitand 1) == 1; ) {
            Z = parent(Z);
            sift_down(a, Z, n, less);
        }
        -- K;
    }
    excess = root();
    *(o + (2 * s - 1)) = std::move(*(o + excess));
    excess = make_heap(o, root(), 2 * s - 1, 2 * s - 1, t, a, I, less);
    *(a + root()) = std::move(*(a + excess));
    for (index Z = I; (Z bitand 1) == 1; ) {
        Z = parent(Z);
        sift_down(a, Z, n, less);
    }
    free(o);
}

mr.h++

// In-situ/In-place McDiarmid & Reed

#ifndef IN_PLACE
#define IN_PLACE
#endif

#define FACTOR

namespace mr {

constexpr std::size root() {

}
return 0;
}

template <typename index>
index parent(index i) {
    return (i - 1) / 2;
}

template <typename index>
index left_child(index i) {
    return (i << 1) + 1;
}

template <typename index>
index right_child(index i) {
    return (i << 1) + 2;
}

template <typename index, typename height>
index ancestor(index i, height h) {
    return (((i + 1) >> h) - 1);
}

bool is_inside(index i, index n) {
    return i < n;
}

template <typename iterator, typename index, typename comparator>
void sift_up(iterator a, index j, comparator less) {
    using element = typename std::iterator_traits<iterator>::value_type;
    element in = std::move(*(a + j));
    while (j > root()) {
        index i = parent(j);
        if (less(*(a + i), in)) {
            *(a + j) = std::move(*(a + i));
            j = i;
        } else {
            break;
        }
    }
    *(a + j) = std::move(in);
}

template <typename iterator, typename index, typename comparator>
void sift_down(iterator a, index i, index n, comparator less) {
    using element = typename std::iterator_traits<iterator>::value_type;
    element x = std::move(*(a + i));
    index j = left_child(i);
while (is_inside(j, n)) {
    j = j + less(*(a + j), *(a + (j + 1)));  
    if (not less(x, *(a + j))) {
        break;
    }
    *(a + i) = std::move(*(a + j));
    i = j;
    j = left_child(i);
    }
    *(a + i) = std::move(x);
}

enum {LEFT = 0, RIGHT = 1, UNDEFINED = 2};

template <typename iterator, typename index, typename comparator, 
          typename bit_vector>
void make_heap(iterator a, index K, index N, comparator less, 
               bit_vector& b) {
    using element = typename std::iterator_traits<iterator>::
      value_type;
    index L = K;
    index R = K;
    index n = root();
    while (is_inside(L, N)) {
      L = left_child(L);
      n = left_child(n);
      R = right_child(R);
    }
    
    for (index i = root(); is_inside(i, n); ++i) {
      b[i] = UNDEFINED;
    }
    index m = n;
    do {
      L = parent(L);
      R = parent(R);
      for (index M = R + 1; M > L;) {
        -- M;
        -- m;
        index J = M;
        index j = m;
        index d = 0;
        while (is_inside(left_child(J), N)) {
          ++ d;
          if (b[j] != UNDEFINED) {
            index flag = b[j];
            J = left_child(J) + flag;
            j = left_child(j) + flag;
          }
          else {
            index flag = less(*(a + left_child(J)), *(a + right_child (J)));
          }
        }
        // Other code
b[j] = flag;
J = left_child(J) + flag;
j = left_child(j) + flag;
}

while (J > M and less(*(a + J), *(a + M))) {
    J = parent(J);
    j = parent(j);
    -- d;
}

if (J ≠ M) {
    element temp = std::move(*(a + M));
    while (d > 0) {
        *(a + ancestor(J, d)) = std::move(*(a + ancestor(J, d – 1)));
        index i = ancestor(j, d);
        b[i] = UNDEFINED;
        -- d;
    }
    *(a + J) = std::move(temp);
}
}
while (L ≠ K);

template <typename iterator, typename comparator>
void make_heap(iterator first, iterator past, comparator less) {
    using index = std::size_t;
    index const n = past – first;
    if (n < 2) {
        return;
    }
    index const m = (n bitand 1) ? n – 1;
    index lg_m = ilogb(m);
    index s = 2;
    index h = 1;
    while (s < FACTOR * lg_m) {
        s = 2 * s;
        h = h + 1;
    }
    if (n ≤ s) {
        std::make_heap(first, past, less);
        return;
    }
    #ifndef IN_PLACE
    cphstl::packed_array extra_space(2);
    extra_space.resize(2 * s);
    #else
    std::vector<unsigned char> extra_space;
    extra_space.resize(2 * s);
    #endif
index const last_leaf = (1 << lg_m) - 2;
index K = ancestor(last_leaf, h - 1);
index I = parent(K + 1);
while (true) {
    make_heap(first, K, m, less, extra_space);
    for (index Z = K; (Z bitand 1) == 1; ) {
        Z = parent(Z);
        sift_down(first, Z, m, less);
    }
    if (K == I) {
        break;
    } -= K;
}
sift_up(first, n - 1, less);
}

Helper

packed_array.h++

/*
 * A packed array stores a sequence a small integers of equal length
 * compactly in memory
 */
#include <climits> // CHAR_BIT
#include <cassert> // std::size_t
#include <vector> // std::vector
namespace cphstl {
   class packed_array {
   public:
      using value_type = std::size_t;
      using size_type = std::size_t;
      enum {word_size = CHAR_BIT * sizeof(value_type)};
   class reference {
   private:
      friend class packed_array;
      value_type* word_pointer;
      value_type offset;
      value_type mask;
reference();

public:

reference(packed_array&amp; a, std::size_t position)
  : word_pointer(a.data[0] + (position &lt;&lt; a.shift) / a.
    word_size),
    offset((position &lt;&lt; a.shift) bitand (a.word_size - 1)),
    mask(a.mask) {
}

~reference() {
}

// For a[i] = x;
template &lt;&lt;typename integer&gt;
reference&lt; integer &gt; operator=(integer x) {
  value_type y = x;
  y = y bitand mask;
  y = y &lt;&lt; offset;
  value_type z = mask &lt;&lt; offset;
  word_pointer &amp;= ~z;
  word_pointer |= y;
  return *this;
}

// For a[i] = a[j];
reference&lt; &gt; operator=(reference const&amp; r) {
  value_type x = *r.word_pointer;
  x = x &gt;&gt; r.offset;
  x = x bitand r.mask;
  value_type y = x bitand mask;
  y = y &lt;&lt; offset;
  value_type z = mask &lt;&lt; offset;
  word_pointer &amp;= ~z;
  word_pointer |= y;
  return *this;
}

// For x = a[i];
operator value_type() const {
  value_type x = *word_pointer;
  x = x &gt;&gt; offset;
  x = x bitand mask;
  return x;
}

bool operator==(reference const&amp; r) const {
  value_type x = (value_type) r;
  value_type y = (value_type) *this;
  return x == y;
}
bool operator=(reference const& r) const {
    return not (*this == r);
}

template <typename integer>
bool operator==(integer x) const {
    value_type y = (value_type) *this;
    return value_type(x) == y;
}

template <typename integer>
bool operator!=(integer x) const {
    return not (*this == x);
}

friend class reference;

template <typename integer>
explicit packed_array(integer min_width_in_bits = 1) {
    n = 0;
    value_type delta = 1;
    shift = 0;
    while (delta < min_width_in_bits) {
        delta = 2 * delta;
        shift = shift + 1;
    }
    mask = (1 << delta) - 1;
}

size_type size() const {
    return n;
}

template <typename integer>
void resize(integer new_size) {
    n = new_size;
    data.resize(((n << shift) + word_size - 1) / word_size);
}

template <typename integer>
reference const operator[](integer position) const {
    return reference(*this, position);
}

template <typename integer>
reference operator[](integer position) {
    return reference(*this, position);
}

private:
Drivers

driver.c++

```cpp
#include <algorithm> // std::random_shuffle std::make_heap std::sort
#include <ctime>    // std::clock_t std::clock CLOCKS_PER_SEC
#include <functional>// std::less
#include <iostream> // std::cout std::cerr
#include <iterator> // std::iterator_traits
#include <utility>  // std::move

template <typename iterator>
bool is_permutation(iterator first, iterator past) {
    using element = typename std::iterator_traits<iterator>::value_type;
    std::sort(first, past);
    for (iterator q = first; q != past; ++q) {
        element i = element(q - first);
        if (*q != i) {
            std::cerr << i << ": element missing " << *q << ": instead" << std::endl;
            return false;
        }
    }
    return true;
}

template <typename iterator, typename comparator>
bool in_heap_order(iterator first, iterator past, comparator less) {
    using index = typename std::iterator_traits<iterator>::difference_type;
    const iterator a = first - 1;
    const index N = past - first;
    bool violated = false;
    for (index i = N; i > 1; i--) {
        if (less(a[i / 2], a[i])) {
            std::cerr << i << ": parent=" << a[i / 2] << ": me=" << a[i] << std::endl;
            violated = true;
        }
    }
    return !violated;
}
```
template <typename iterator, typename index, typename comparator>
bool in_heap_order(iterator a, index j, index n, comparator less) {
    if (j > n) {
        return true;
    }

    bool ok = true;
    index left = 2 * j;
    index right = left + 1;
    if (left ≤ n and less(a[j], a[left])) {
        std::cerr << a[j] << std::endl;
        std::cerr << a[left] << " " << a[right] << std::endl;
        if (2 * left < n) {
            std::cerr << a[2 * left] << " " << a[2 * left + 1] << " ";
            std::cerr << a[2 * right] << " " << a[2 * right + 1] << std::endl;
        }
        ok = false;
    }
    if (right ≤ n and less(a[j], a[right])) {
        std::cerr << a[j] << std::endl;
        std::cerr << a[left] << " " << a[right] << std::endl;
        if (2 * left < n) {
            std::cerr << a[2 * left] << " " << a[2 * left + 1] << " ";
            std::cerr << a[2 * right] << " " << a[2 * right + 1] << std::endl;
        }
        ok = false;
    }
    ok &= in_heap_order(a, 2 * j, n, less);
    ok &= in_heap_order(a, 2 * j + 1, n, less);
    return ok;
}

#include "algorithm.h++"

#define MEASURE_COMPARISONS
long long volatile comparisons = 0;

template <typename T>
class counting_comparator {
    public:
        using first_argument_type = T;
        using second_argument_type = T;
        using result_type = bool;

        bool operator()(T const & a, T const & b) const {
            return not violated;
        }
    }

}
++ comparisons;
return a < b;
}
};
#endif
#ifdef MEASURE_MOVES
long long volatile moves = 0;
#endif

// template<typename T>
class move_counter {
private:
T datum;
move_counter(move_counter const&) = delete;
move_counter& operator=(move_counter const&) = delete;

public:
explicit move_counter()
: datum(0) {
    moves += 1;
}

template<typename number>
explicit move_counter(number x = 0)
: datum(x) {
    moves += 1;
}

move_counter(move_counter&& other) {
    datum = std::move(other.datum);
    moves += 1;
}

move_counter& operator=(move_counter&& other) {
    datum = std::move(other.datum);
    moves += 1;
    return *this;
}

operator T() const {
    return datum;
}

template<typename U>
friend bool operator<(move_counter<U> const&, move_counter<U> const &);

// template<typename U>

friend bool operator==(move_counter<U> const& x, move_counter<U> const& y);

};

template <typename T>
bool operator<(move_counter<T> const& x, move_counter<T> const& y) {
    return x.datum < y.datum;
}

template <typename T>
bool operator==(move_counter<T> const& x, move_counter<T> const& y) {
    return x.datum == y.datum;
}

#endif

template <typename iterator>
void generate(iterator p, iterator r, char z) {
    using element = typename std::iterator_traits<iterator>::value_type;
    switch (z) {
    case 'd':
        for (iterator q = p; q < r; ++q)
            *q = element((r - 1) - q);
        break;
    case 'i':
        for (iterator q = p; q < r; ++q)
            *q = element(q - p);
        break;
    case 'r':
        for (iterator q = p; q < r; ++q)
            *q = element(q - p);
        std::random_shuffle(p, r);
        break;
    case 'z':
        bool t = false;
        for (iterator q = p; q < r; ++q) {
            *q = element(t);
            t = not t;
        }
        break;
    }
}

void usage(char const* program) {
    std::cerr << "Usage: \"" << program
    "\"<N><'i'increasing | 'd'decreasing | 'r'andom | 'b'ool>\""
    << std::endl;
    exit(1);
}

int main(int argc, char** argv) {
ifdef MEASURE_MOVES
using element = move_counter<int>;
#endif

ifdef MEASURE_COMPARISONS
using C = counting_comparator<element>;
#else
using C = std::less<element>;
#endif

unsigned long N = 15;
char method = 'i';
if (argc == 2) {
    N = atoi(argv[1]);
    method = 'i';
}
else if (argc != 3) {
    usage(argv[0]);
} else {
    N = atoi(argv[1]);
    method = *argv[2];
}
if (N > MAXSIZE) {
    std::cerr << "N out of bounds [0.." << MAXSIZE
    << "]" << std::endl;
    usage(argv[0]);
}
switch (method) {
    case 'd':
    case 'i':
    case 'r':
    case 'b':
        break;
    default:
        std::cerr << "Method not in ['d','i','r','b']" << std::endl;
    usage(argv[0]);
}
element* a = new element[MAXSIZE];
element* b = a;
for (volatile unsigned long t = MAXSIZE / N; t > 0; t--) {
    generate(b, b + N, method);
    b = b + N;
}
#endif
moves = 0;

# elif defined(MEASURE_COMPARISONS)
comparisons = 0;
# endif

# if defined(REPETITIONS)
unsigned long const repetitions = REPETITIONS;
#else
unsigned long const repetitions = MAXSIZE / N;
#endif

b = a;

# if not defined(MEASURE_COMPARISONS) and not defined(MEASURE_MOVES)
std::clock_t start = std::clock();
#endif

for (volatile unsigned long t = 0; t < repetitions; ++t) {
    NAME::make_heap(b, b + N, C());
    b = b + N;
}

# if not defined(MEASURE_COMPARISONS) and not defined(MEASURE_MOVES)
std::clock_t stop = std::clock();
#endif

# if not defined(NDEBUG)

b = a;

for (volatile unsigned long t = 0; t < repetitions; ++t) {
    bool ok = in_heap_order(b, b + N, std::less<element>());
    if (!ok) {
        return 1;
    }
    if (method == 'd' or method == 'i' or method == 'r') {
        ok = is_permutation(b, b + N);
        if (!ok) {
            return 2;
        }
    }
    b = b + N;
}
#endif

double t = double(repetitions) * double(N);

# if defined(MEASURE_COMPARISONS)
std::cout.precision(3);
std::cout << N << \	"t" << double(comparisons) / t << std::endl;
# elif defined(MEASURE_MOVES)
std::cout.precision(3);
std::cout << N << \	"t" << double(moves) / t << std::endl;
#else
double ns = 1000000000.0 * double(stop - start) / double(CLOCKS_PER_SEC);
std::cout.precision(4);
std::cout << N << ' ' << ns / t << std::endl;
#endif

delete[] a;
return 0;
}

// test-driver.c++

#include <algorithm> // std::random_shuffle std::make_heap std::sort
#include <cassert> // assert macro
#include <functional> // std::less
#include <iostream> // std::cout std::cerr
#include <vector> // std::vector

template<typename iterator>
void show(iterator a, iterator z) {
    while (a != z) {
        std::cout << long(*a) << " ";
        ++a;
    }
    std::cout << std::endl;
}

template<typename index>
index left_heap_child(index i) {
    return 2 * i + 1;
}

template<typename index>
index right_heap_child(index i) {
    return 2 * i + 2;
}

template<typename iterator, typename index>
void print(iterator a, index j, index N) {
    if (j < N) {
        std::cerr << a[j];
    }
    std::cerr << " ";
}

template<typename iterator, typename index, typename comparator>
bool in_heap_order(iterator a, index j, index n, comparator less) {
    if (j >= n) {
        return true;
    }
    bool ok = true;
    index left = left_heap_child(j);
index right = right_heap_child(j);
if (left < n and less(a[j], a[left])) {
    std::cerr << "left child not in heap order " << j << std::endl;
    print(a, j, n);
    std::cerr << std::endl;
    print(a, left, n);
    print(a, right, n);
    std::cerr << std::endl;
    print(a, left_heap_child(left), n);
    print(a, right_heap_child(left), n);
    print(a, left_heap_child(right), n);
    print(a, right_heap_child(right), n);
    std::cerr << std::endl;
    ok = false;
}
if (right < n and less(a[j], a[right])) {
    std::cerr << "right child not in heap order " << j << std::endl;
    print(a, j, n);
    std::cerr << std::endl;
    print(a, left, n);
    print(a, right, n);
    std::cerr << std::endl;
    print(a, left_heap_child(left), n);
    print(a, right_heap_child(left), n);
    print(a, left_heap_child(right), n);
    print(a, right_heap_child(right), n);
    std::cerr << std::endl;
    ok = false;
}
ok &= in_heap_order(a, left_heap_child(j), n, less);
ok &= in_heap_order(a, right_heap_child(j), n, less);
return ok;
}

template <typename iterator>
bool is_permutation(iterator first, iterator past) {
    using integer = typename std::iterator_traits<iterator>::value_type;
    std::vector<integer> v;
v.resize(past - first);
    std::copy(first, past, v.begin());
    std::sort(v.begin(), v.end());
    for (auto q = v.begin(); q != v.end(); ++q) {
        integer i = integer(q - v.begin());
        if (*q != i) {
            std::cerr << i << " missing " << q << " instead" << std::endl;
            return false;
        }
    }
    return true;
}
template <typename iterator>
void generate(iterator p, iterator r) {
    using element = typename std::iterator_traits<iterator>::value_type;
    for (iterator q = p; q < r; ++q) {
        *q = element(q - p);
    }
    std::random_shuffle(p, r);
}

#include "algorithm.h++" // NAME

int main() {
    int const magic = 20;
    int N = (1 << magic);
    int a[N];
    std::vector<int> testcase;
    for (int n = 0; n ≤ 20000; ++n) // 8448
        testcase.push_back(n);
    for (int h = 3; h  magic; ++h) {
        for (int delta = -5; delta < 6; ++delta) {
            testcase.push_back((1 << h) + delta);
        }
    }
    testcase.push_back(N);
    for (int n: testcase) {
        std::cout << "n: " << n << std::endl;
        generate(a + 0, a + n);
        NAME::make_heap(a + 0, a + n, std::less<int>());
        assert(::in_heap_order(a, 0, n, std::less<int>()));
        assert(::is_permutation(a + 0, a + n));
    }
}

Makefile

CXX=g++
CXXFLAGS=-O3 -std=c++11 -x c++ -Wall -Wextra -DNDEBUG #-DIN_PLACE

header-files:= $(wildcard *.h++)
versions:= $(basename $(header-files))
time-tests:= $(addsuffix .time, $(versions))
log-files:= $(addsuffix .log, $(versions))
comp-tests:= $(addsuffix .comp, $(versions))
mov-tests:= $(addsuffix .move, $(versions))
branch-tests:= $(addsuffix .branch, $(versions))
cache-tests:= $(addsuffix .cache, $(versions))
instruction-tests:= $(addsuffix .count, $(versions))
unittests := $(addsuffix .test, $(versions))
portability-tests := $(addsuffix .port, $(versions))
profilings := $(addsuffix .prof, $(versions))

N = 1023 32767 1048575 33554431
data = r i d b

$(time-tests): %.time : %.h++
  @cp $*.h++ algorithm.h++
  $(CXX) $(CXXFLAGS) -DNAME=-*$ driver.c++
  @for n in $(N) ; do
    ./a.out $n r : \done:
  rm -f algorithm.h++ ./a.out

$(log-files): %.log : %.h++
  @cp $*.h++ algorithm.h++
  $(CXX) $(CXXFLAGS) -DNAME=-*$ driver.c++
  @for n in `cat n.txt` ; do
    ./a.out $n r >> $*.log ; \done:
  rm -f algorithm.h++ ./a.out

$(move-tests): %.move : %.h++
  @cp $*.h++ algorithm.h++
  $(CXX) $(CXXFLAGS) -DNAME=-*$ driver.c++
  @for d in $(data) ; do
    echo $$d ; \for n in $(N) ; do
      ./a.out $n $$d ; \done
    done ; \done:
  rm -f algorithm.h++ ./a.out

$(comp-tests): %.comp : %.h++
  @cp $*.h++ algorithm.h++
  $(CXX) $(CXXFLAGS) -DNAME=-*$ driver.c++
  @for d in $(data) ; do
    echo $$d ; \for n in $(N) ; do
      ./a.out $n $$d ; \done
    done \done:
  rm -f algorithm.h++ ./a.out

$(instruction-tests): %.count : %.h++
  @cp $*.h++ algorithm.h++
  @for n in $(N) ; do
    python instruction_count.py $* $$n ; \rm -f ./a.out ; \rm -f ./cachegrind.out.* ; \done

54

$(cache-tests): %.cache : %.h++
@for n in $(N) ; do \
    python cache_misses.py driver.c++ $* $$n ; \
    rm -f ./a.out ; \
    rm -f ./cachegrind.out.* ; \
    done

$(branch-tests): %.branch : %.h++
@for n in $(N) ; do \
    python branch_mispredictions.py $* $$n ; \
    rm -f ./a.out ; \
    rm -f ./cachegrind.out.* ; \
    done

TESTFLAGS=--O3 --std=c++11 -Wall -Wextra -x c++ -g -DDEBUG

$(unittests): %.test : %.h++
@cp $* algorithm.h++
$(CXX) $(TESTFLAGS) -DNAME=$* test-driver.c++
./a.out

$(portability-tests): %.port : %.h++
@cp $* algorithm.h++
$(CXX) $(TESTFLAGS) -DNAME=$* portability-driver.c++
./a.out

rm -f algorithm.h++ ./a.out

PROFILERFLAGS = -DNDEBUG -Wall -std=c++11 -pedantic -x c++ -g

$(profilings): %.prof : %.h++
@cp $* algorithm.h++
$(CXX) $(PROFILERFLAGS) -DNAME=$* -DMAXSIZE=4194300 driver.c++

valgrind --tool=callgrind --dump-instr=yes --collect-jumps=yes --callgrind-out-file=$*.callgrind.out ./a.out 1048575

rm -f algorithm.h++

# Other tools

find:
find . -type f -exec grep $(word) {} \; | less