



An experimental study of priority queues

By

Claus Jensen

University of Copenhagen

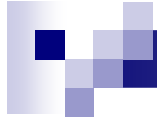


STL

The STL impose the following requirement on the algorithms:

- Top $O(1)$
- Pop $2\log(n)$
- Push $\log(n)$
- Make $3n$

Where n is the number of elements in the priority queue.



The Algorithms benchmarked

- Implicit binary heap
- Navigation piles
- Weight-biased leftist trees



Implicit binary heap

Space requirements

- No extra space needed.

Requirement on the number of Element comparisons.

- Top $O(1)$
- Pop $2 \log n$
- Push $\log n$



Navigation piles

- Introduced by Jyrki Katajainen and Fabio Vitale.



Structure of navigation piles

■ Shape

- It is a left-complete binary tree of size N , where $N \geq n$ (# of elements stored).

■ Leaf

- The leaves are used to store the elements.
- The first n leaves store one element each, the rest are empty.



Structure of navigation piles

- Branch:

- A branch (node) contains the index of the leaf storing the top element among the group of elements attached to the leaves in the subtree rooted by the branch.



Structure of navigation piles

- Representation:

- The elements are stored in sequence $A[0..n)$.
- The branches are stored in sequence $B[0..N)$.

Structure of navigation piles

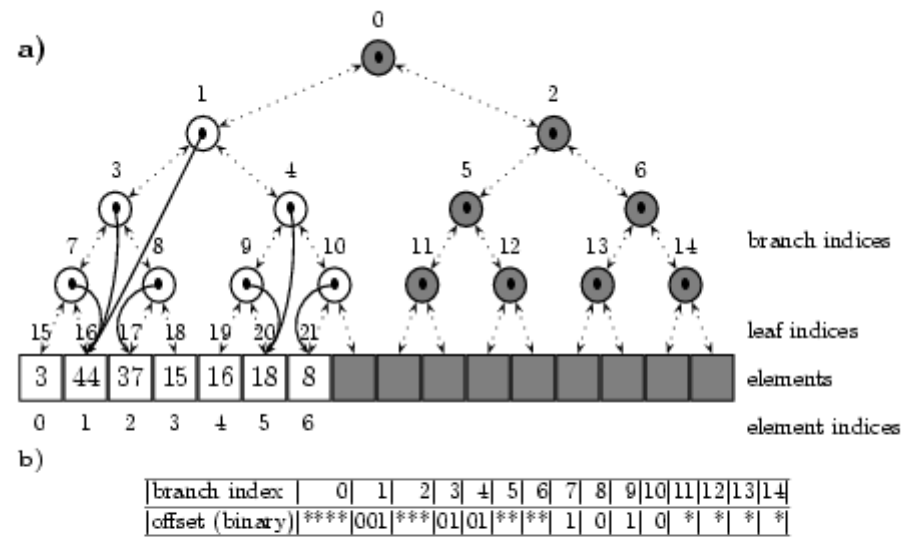


Figure 1. a) A navigation pile of size 7 and capacity 16. The normal parent/child relationships are shown with dotted arrows and the references indicated by the offsets with solid arrows. The gray nodes are not in use. b) The bit representation of the navigation information. Bits marked with * are not in use.



Functions

- Make

- Construction of a navigation pile is done by visiting the branches in a bottom-up manner and computing its index using the indices available at the children.
 - $n-1$ element comparisons
 - n element moves
 - $O(n)$ instructions



Functions

■ Push

- A new element is inserted into the first empty leaf, followed by an update of the branches from the leaf to the root. A binary search strategy can be used in the update.
 - $\log \log n + O(1)$ element comparisons
 - 1 element move
 - $O(\log n)$ instructions



Functions

- Top

- A reference to the top element stored at the root is returned.
 - $O(1)$ instructions



Functions

■ Pop

- The top element is erased and the indices are updated.
- The update is done in three different ways depending on circumstances inside the navigation pile.
 - Ceiling of $\log n$ element comparisons
 - 1 or 2 element moves
 - $O(\log n)$ instructions



Functions (Pop)

- l: Element index of the last leaf.
- m: Element index of the top element.
- i: Branch index of the first ancestor of the last leaf which has two children.
- j: The element index referred to by the first child of i.
- k: The element index referred to by i.



Functions (Pop)

- Case 1: $m = l$
- The top element is erased and the index values on the path from the new last leaf to the root are updated.



Functions (Pop)

- Case 2: $m \neq l$ and $k \neq l$
- $A[m] \leftarrow A[l]$ and the element stored at the last leaf is erased.
 - The content of the branches on the path from the leaf corresponding to the position m up to the root are updated.



Functions (Pop)

- Case 3: $m \neq l$ and $k = l$
 - $A[m] \leftarrow A[j]$, $A[j] \leftarrow A[l]$ and the element stored at the last leaf is erased.
 - The content of the branches on the path $[i, \text{root}]$ are updated with a reference to j , if they earlier referred to the last leaf.
 - The content of the branches on the path from the leaf corresponding to the position m up to the root are updated.



Weight-biased leftist trees

The structure of Weight-biased leftist trees.

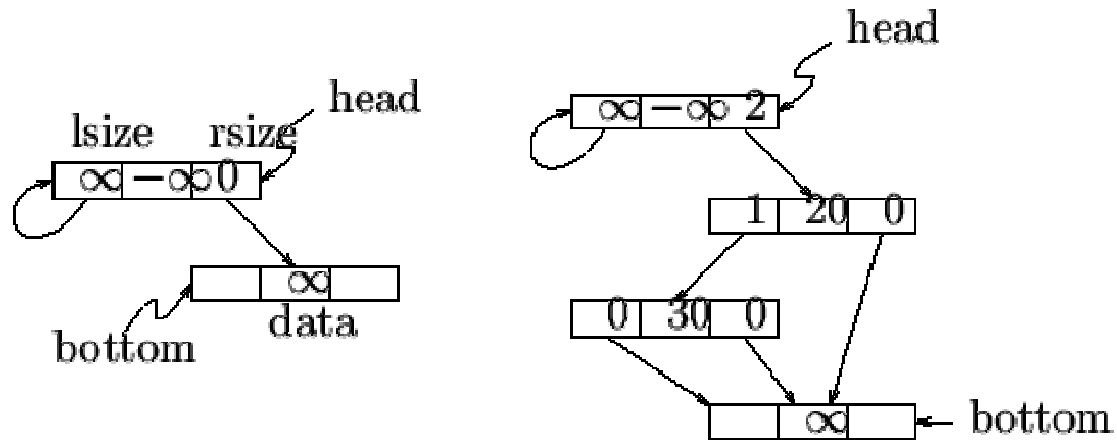
- Nodes connected by pointers.
- Two pointers are used for each node.
- Two integer is used in every node to store the node's right side and left side weight.



Weight-biased leftist trees

- Let T be an extended binary tree. For any internal node x of T , let $\text{LeftChild}(x)$ and $\text{RightChild}(x)$, respectively, denote the left and right children of x . The weight, $w(x)$, of any node x is the number of internal nodes in the subtree with root x .
- Definition: A weight-biased leftist tree (WBLT) is a binary tree such that if it is not empty, then
$$\text{weight}(\text{LeftChild}(x)) \geq \text{weight}(\text{RightChild}(x))$$
for every internal node x .

Weight-biased leftist trees



(a) Empty min-WBLT

(b) Nonempty min-WBLT

Weight-biased leftist trees (Insert)

```
procedure Insert(d ) ;  
{insert d into a min-WBLT}  
begin  
create a node x with x.data = d ;  
t = head ; {head node}  
while (t.right.data.key < d.key ) do  
  begin  
    t.rsize = t.rsize + 1 ;  
    if (t.lsize < t.rsize ) then  
      begin swap t 's children ; t = t.left ; end  
    else t = t.right ;  
    end ;  
x.left = t.right ; x.right = bottom ;  
x.lsize = t.rsize ; x.rsize = 0 ;  
if (t.lsize = t.rsize ) then {swap children}  
  begin  
    t.right = t.left ;  
    t.left = x ; t.lsize = x.lsize + 1 ;  
  end  
else  
  begin t.right = x ; t.rsize = t.rsize + 1 ; end ;  
end ;
```

Weight-biased leftist trees (Delete-min)

```
procedure Delete-min ;
begin
  x = head.right ;
  if (x = bottom ) then return ; {empty tree}
  head.right = x.left ; head.rsize = x.lsize ;
  a = head ;
  b = x.right ; bsize = x.rsize ;
  delete x ;
  if (b = bottom ) then return ;
  r = a.right ;
  while (r ≠ bottom ) do
    begin
      a = bsize + a.rsize ; t = a.rsize ;
      if (a.lsize < t) then {work on a.left}
        begin
          a.right = a.left ; a.rsize = a.lsize ; a.lsize = t ;
          if (r.data.key > b.data.key ) then
            begin a.left = b ; a = b ; b = r ; bsize = t ; end
          else
            begin a.left = r ; a = r ; end
        end
      else
        do symmetric operations on a.right ;
        r = a.right ;
      end ;
    if (a.lsize < bsize ) then
      begin
        a.right = a.left ; a.left = b ;
        a.rsize = a.lsize ; a.lsize = bsize ;
      end
    else
      begin a.right = b ; a.rsize = bsize ; end ;
  end ;
```



Weight-biased leftist trees

Space requirements

- Two pointers and two integers.

Requirement on the number of Element comparisons.

- Top $O(1)$
- Pop (delete-min) $2\log n$
- Push (insert) $\log n$



Programs

- STL binary heap
- Modified version of Weight-biased leftist trees by Seonghun Cho and Sartaj Sahni
- Navigation piles, programmed by me with an starting point in the algorithms presented in the article on Navigation piles by Jyrki Katajainen and Fabio Vitale.



Programs

- The navigation pile data structure was implemented in three different versions.
 - The first version used an STL vector to store the indices at the branches.
 - The second version used pointers instead of indices in an attempt to optimize the performance.



Programs

- The third version packed the offsets as suggested in the original article by Jyrki Katajainen and Fabio Vitale. Offsets indicate the leaf position of the top element within the subtree rooted by the branch. The use of offsets instead of indices reduces the space required to $2N$ bits, where N is the capacity of the navigation pile.



Benchmarks

■ Environment A

- Intel computer:
- CPU: Pentium 3 (1 GHz)
- Main Memory: RAM 384 MB
- Cache level 2: 246 KB
- Operation system: Debian Sid
- C++ Compiler: g++ 3.3



Benchmarks

■ Environment B

- Intel computer:
- CPU: 2 * Pentium 4 (3 GHz)
- Main Memory: RAM 1 GB
- Cache level 2: 1024 KB
- Operation system: Gentoo Linux 3.3.4-r1
- C++ Compiler: g++ 3.3.4



Benchmarks

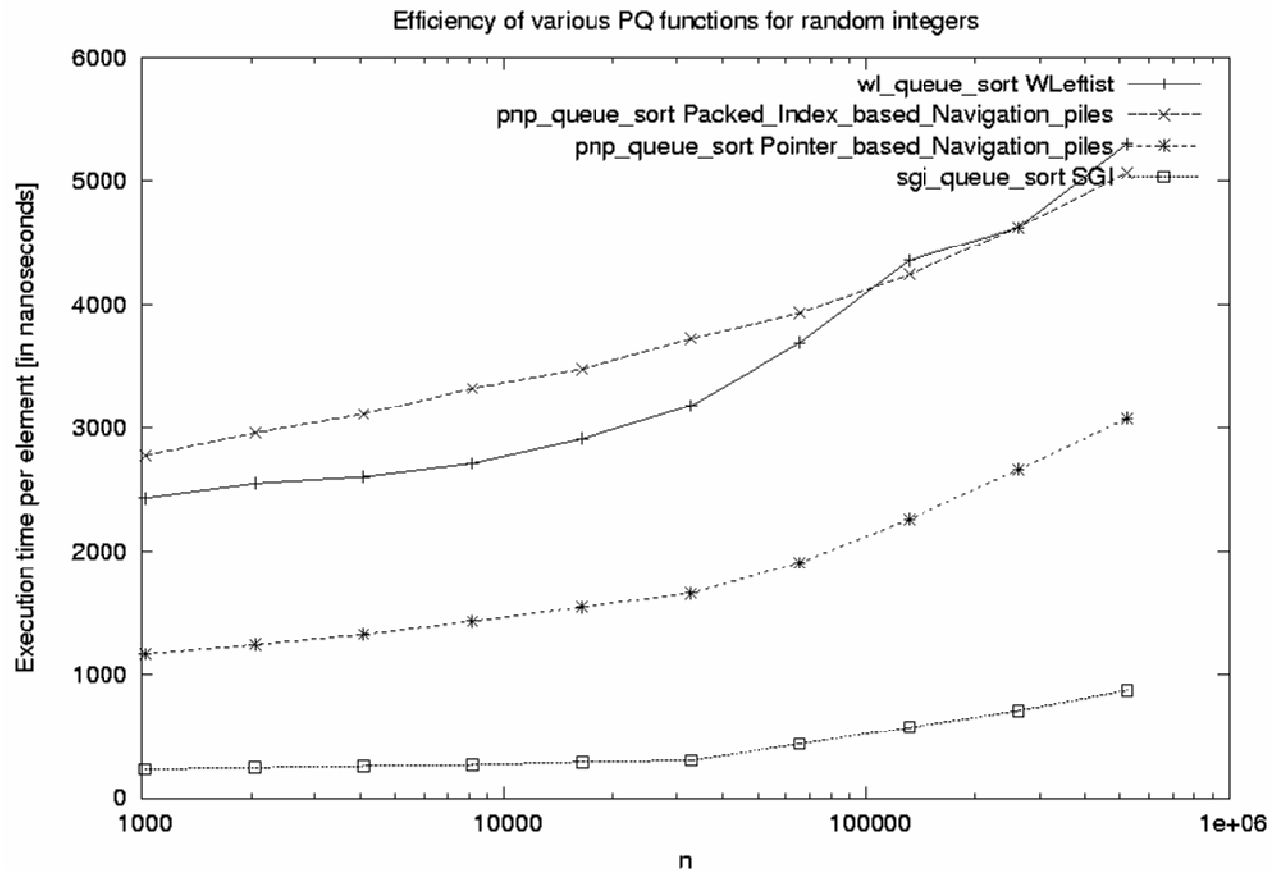
- The input data is unsigned int.
- The benchmarks performed a construction operation followed by a series of pops.
- The CPH STL benchmark tool was used in the production of the benchmarks.



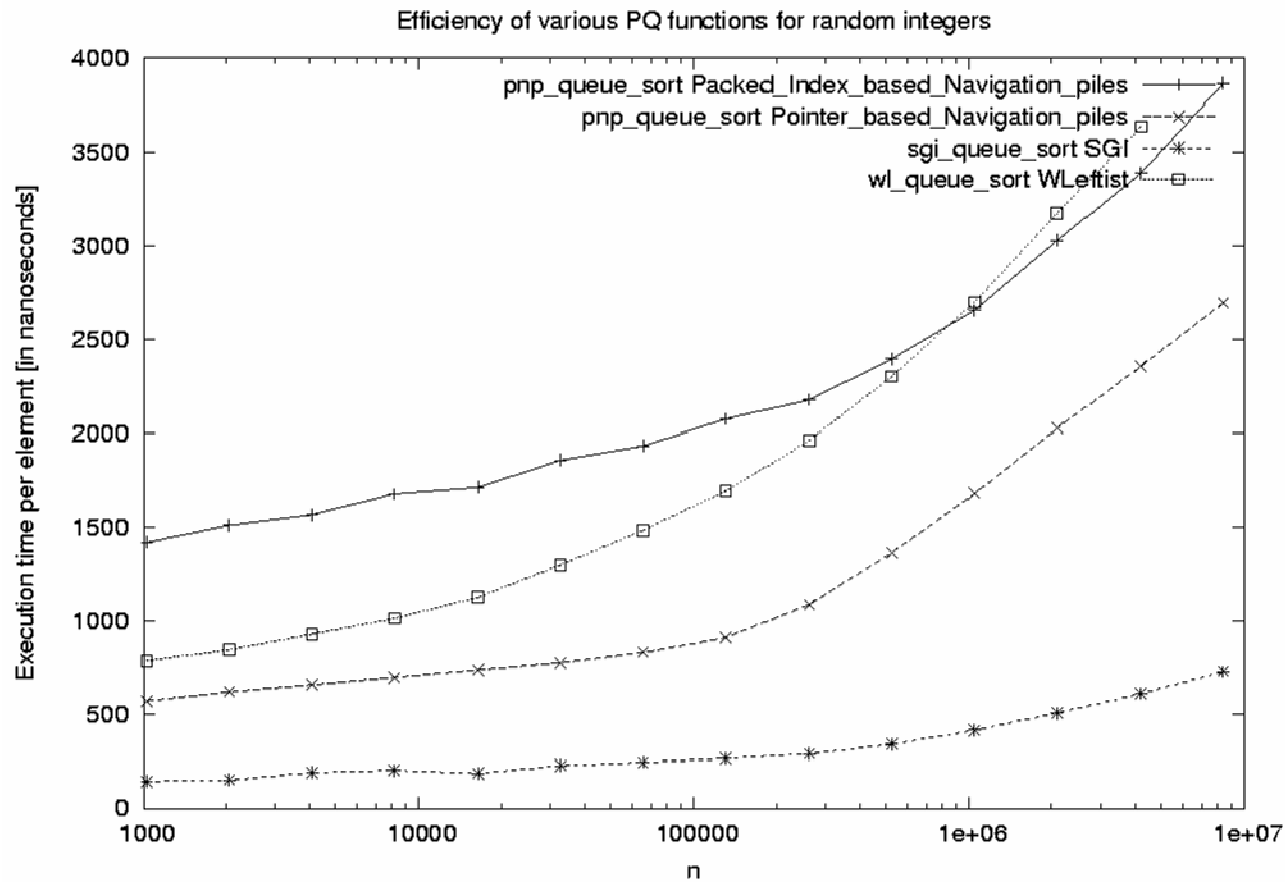
Fastest algorithm

Which is the fastest algorithm, any guess?

Benchmarks (Intel P3)



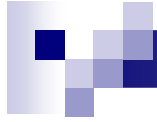
Benchmarks (Intel P4)





Conclusion

- For integers, STL implicit binary heap is to fast for any ting I have found so far.



Ideas

- Other algorithms.
- Other benchmark strategies.



Benchmarks (input data)

	cheap move	expensive move
cheap comparison	unsigned int	big int
expensive comparison	unsigned int In comparison	(int, big int) In comparison