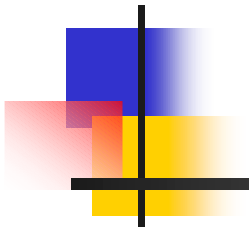
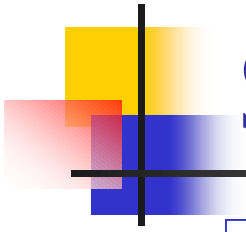


Navigation Piles with Applications to Sorting, Priority Queues, and Priority Deques

Jyrki Katajainen and Fabio Vitale
Department of Computing, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark
{jyrki|fabio}@diku.dk

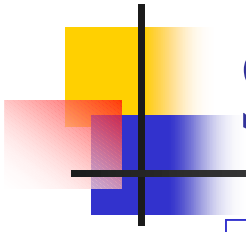
Navigation Piles

- **Using of the new data structure**
  - Sorting algorithms
  - Priority queues
  - Priority deques
  - Implementation of other data structures



Structure Properties

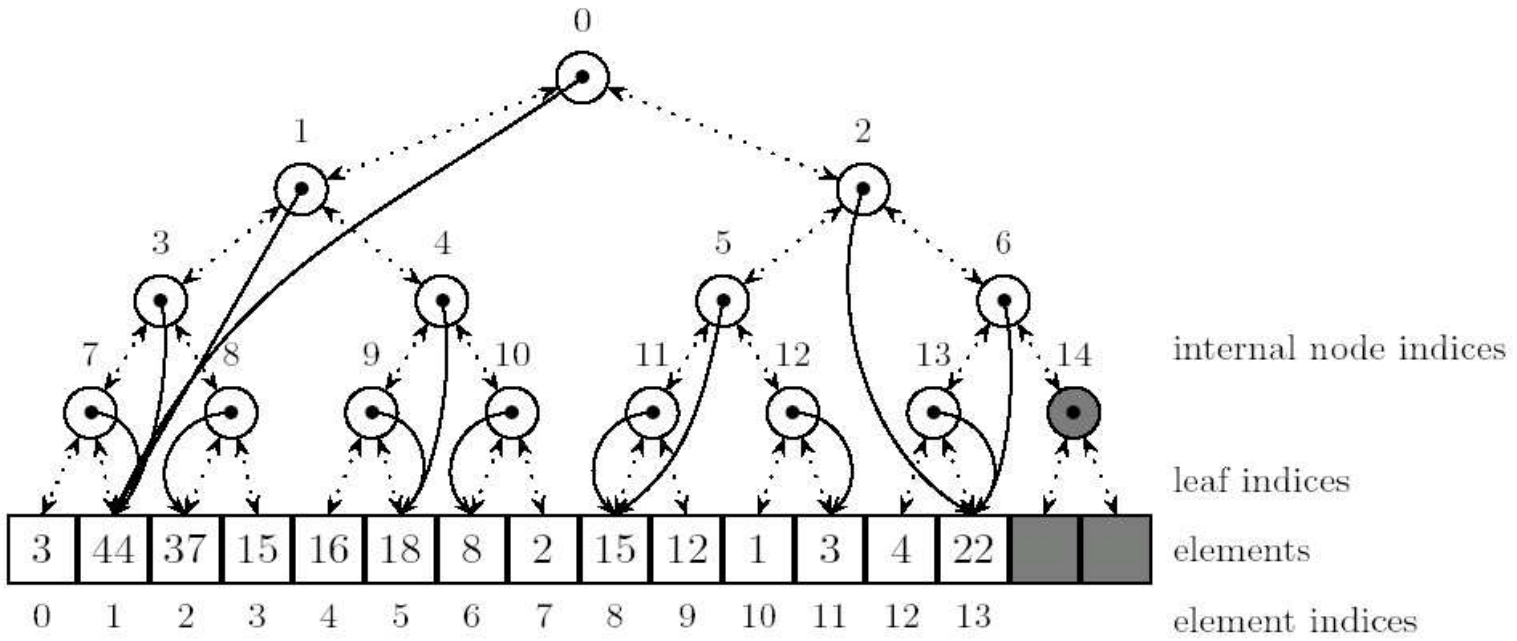
- **Shape**
  - It is a complete binary tree of size $2^{\eta+1}-1 \quad (n \leq 2^{\eta})$
- **Leaves**
  - The first n leaves store one element each
  - The remaining are empty
- **Branch nodes**
  - Each branch node stores the index – inside the leaf sequence dominated by the node – containing the relative top element
- **Representation**
  - The sequence $A[0..n)$ stores the elements
  - The sequence $B[0..2^{\eta+1})$ stores the navigation information



Structure Elements

- **Element indices:**
  Indices of the elements stored in $A[0..n)$
- **Bit indices**
  Indices of the bits stored in $B[0..2^{\eta+1})$
- **Branch node indices**
- **Leaf indices**
- **Levels** (no larger than η)
- **Offsets**
  Indices stored in the branch nodes

Navigation Pile Example



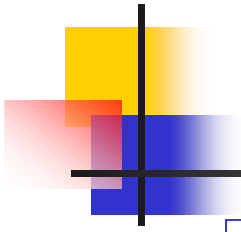
A navigation pile of size 14 and capacity 16. The normal parent/child relationships are shown with dotted arrows and the references indicated by the offsets with solid arrows. The gray nodes are not in use.

Number of extra bits

- Number of branch nodes with depth δ $(0 \le \delta < \eta)$
  - 2^{δ}
- Maximum number of leaves dominated by a branch node whose height is γ
  - 2^{γ}
- Number of bits used for the offsets
  - $\sum_{\delta=0}^{\eta-1} 2^{\delta}(\eta - \delta) < 2^{\eta+1}$

Operations

- **Construction**
  - Visiting the branch nodes in a bottom-up manner (depth-first order visit to improve the cache performances)
  - $\boldsymbol{n}$**-1** element comparisons
    - n-1 branch nodes with more than 1 child
  - $\boldsymbol{n}$ element moves
    - n elements copied to the container sequence
  - $\boldsymbol{O(n)}$ instructions
    - $O(1)$ instructions for every comparison and move



Operations

- ***top()***
  - $O(1)$ instructions
    - Return the index stored at the root
- ***push()*** $(n < 2^{\eta})$
  - $\lceil \log_2(\lceil \log_2 n \rceil + 1) \rceil$ element comparisons
    - Binary search on the path from the new last leaf to the root
  - 1 element move
    - The element is appended at the end of the sequence $A[0..n)$
  - $O(\log n)$ instructions

Operations

- ***pop()***

  ℓ : index of the last leaf $(n - 1)$

  m: index of the top element

  - **Case 1**: $m = \ell$
    - The top element is erased and the offsets on the path from the new last leaf to the root are updated

pop() Case 1

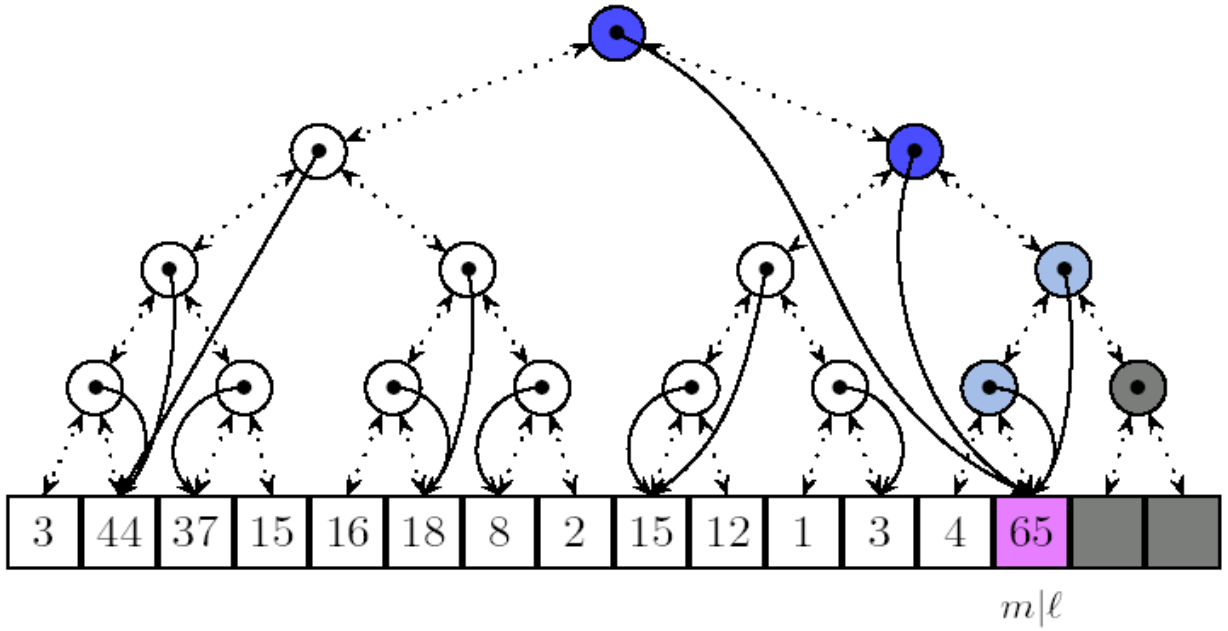


Illustration of Case 1. The offset of the root is refering the last leaf. The leaves whose contents may change are indicated in magenta. In dark blue are indicated the branch nodes whose updating offset require one element comparison each. When updating the contents of the light blue branch nodes, no element comparisons are necessary.

Operations

i_1: first ancestor of the last leaf having two children (in use) $(n > 1)$

i_2: second ancestor of the last leaf having two children (in use) $(\ell \neq 2^k \ \forall k \in \mathbb{N})$

k, j_1: index of the leaves referred respectively by the offset stored at the branch node with index i_2 and at the first child of that one with index i_1 (or $j_1 = \ell - 1$ if ℓ is odd)

- **Case 2**: $m \neq \ell$ **and** $k \neq \ell$ **(or** $m \neq \ell$ **and** ℓ **is a power of** 2**)**
  - $A[m]$ ß $A[n]$ and the element copied is erased
  - The offsets stored at the branch node on the path from $[i_1 .. i_2)$ are updated to refer the leaf at the position j_1
  - The offsets stored on the path from the leaf at the position m to the root are updated

pop() Case 2

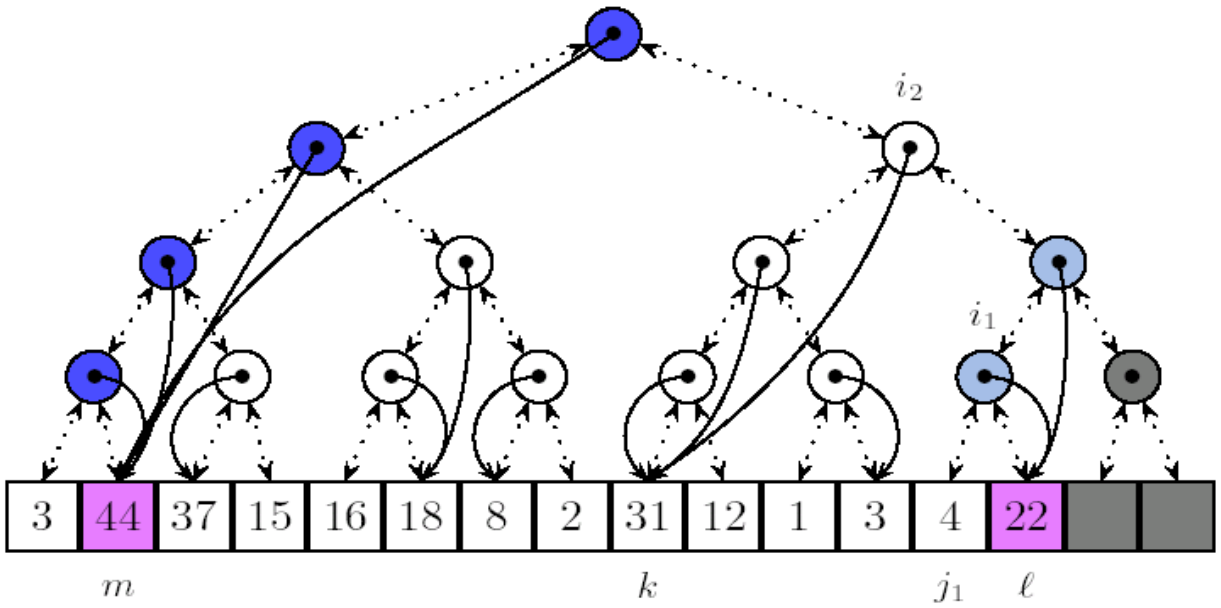


Illustration of Case 2. The offset of i_2 is not refering the last leaf. The offsets of the branch nodes in light blue will refer the leaf whose content is the number 4, and the the top element (44) will be overwritten by the current last leaf (22).

Operations

j_2 : index of the element referred by the offset stored at the first child of the branch node with index i_2

- **Case 3**: $m \neq \ell$ **and** $k = \ell$
  - $A[m]$ ß $A[j_2]$
  - $A[j_2]$ ß $A[\ell]$
  - The element at the last leaf is erased
  - The offsets stored at the branch node on the path from $[i_1 .. i_2)$ are updated to refer the leaf at the position j_1
  - The offsets stored on the path from i_2 (including it) upwards are updated to refer the leaf in the position j_2 if they referred earlier the last leaf
  - The offsets stored on the path from the leaf at the position m to the root are updated

pop() Case 3

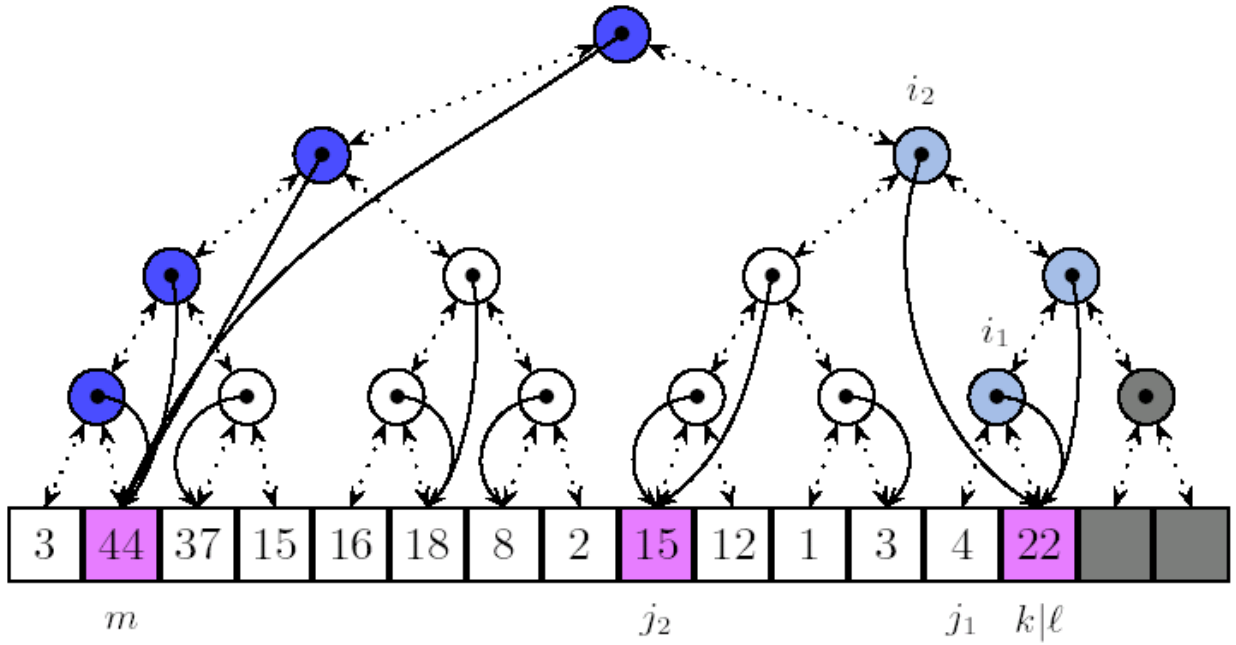
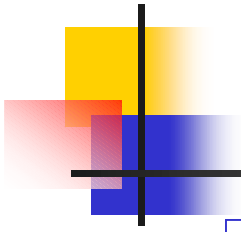


Illustration of Case 3. The offset of i_2 is refering the last leaf. The offsets of the branch nodes in light blue from the bottom til i_2 (excluding it) will refer the leaf j_1 (4); the offset relative to the branch node i_2 will refer the leaf j_2 (15); the the top element (44) will be overwritten by the element in the position j_2 (15); finally the element of the current last leaf (22) will overwrite that one of the leaf j_2.

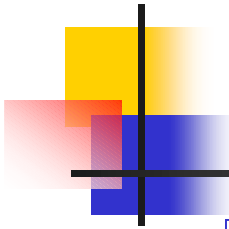
Operations

- At most $\lceil \log_2(n-1) \rceil$ element comparisons
  - In each of the three cases only one path updating involves element comparisons
  - The depth of the root becomes $\lceil \log_2(n-1) \rceil$ and at most one element comparisons is done at each level
- **0, 1 or 2** moves
- $O(\log n)$ instructions



Sorting

- In connection with *pop()* the top element is not overwritten, but it is saved at the earlier last leaf (*pop_and_save()* function)
- The basic version of the sorting algorithm (Pilesort) is a sequence of n-1 *pop_and_save()* operations
- The amount of extra space needed is at most **$4n$ bits** because $2^{\lceil \log_2 n \rceil} < 2n$



Sorting

- Element comparisons
  - n-1 comparisons for the construction
  - $\sum_{k=2}^{n-1} \lceil \log_2 k \rceil < n \log_2 n - 0.91n$
  - If we find the bottom element during the construction with at most $\lceil n/2 \rceil + 1$ comparisons (and keeping a separate copy of it) in order to reduce the number of moves, the bound becomes $\boldsymbol{n \log_2 n + 0.59n + O(1)}$
- Element moves
  - Not more than $4n + O(1)$ in the basic version
  - Finding at the beginning the bottom element the bound becomes $\boldsymbol{2.5n + O(1)}$ (for every odd value of ℓ if we need 3 moves for the execution of *pop_and_save()*, we will need at most 2 moves in the following iteration)

Priority Queues

- Dynamization of the container storing the elements
  - Dynamization techniques developed by Katajainen and Mortensen
  - Space-efficient resizable arrays
- Dynamization of the bit sequence
  - Bit sequence divided into blocks B_* of size 2^5 and B_h of size 2^h for $h \geq 5$
  - Bit sequence of the block B_* is kept in memory at all time
  - The largest block can be empty (deamortization of the cost of allocations and deallocations in proximity of block boundaries)
  - If there is an empty block and the largest nonempty block lacks more than 2^5 elements, the bit sequences for the largest block is freed

Priority Queues (Illustration)

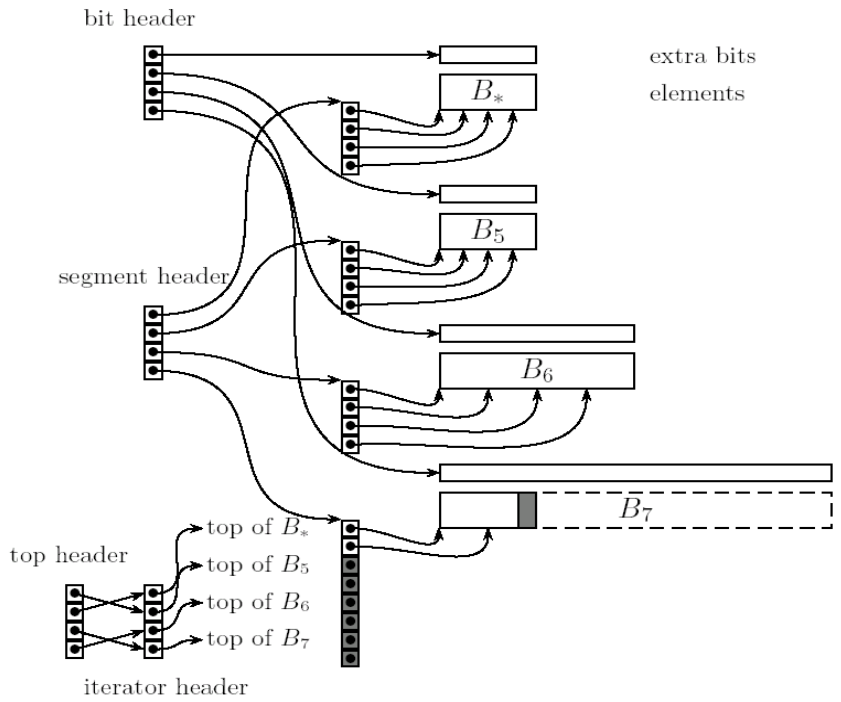
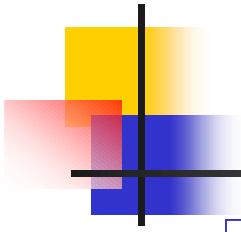


Illustration of the Priority Queue structure. A collection of navigation piles storing 77 elements. A space-efficient resizable array is used as the container for the elements. The gray areas denote empty memory segments allocated for dynamization purposes.



Priority Queues

- **Space**
  - 2^{h+1} bits for every B_h $(h \geq 5)$
  - $\boldsymbol{4n + O(1)}$ extra bits for all the priority queue
- **Construction**
  - $\boldsymbol{n + \Theta((\log_2 n)(\log_2 \log_2 n))}$ element comparisons
  - $\boldsymbol{n}$ element moves
  - $\boldsymbol{O(n)}$ instructions
- ***top()***
  - We assume that *push()* and *pop()* maintain the top and iterator headers
  - $\boldsymbol{O(1)}$ instructions (following the cursor of the iterator headers)

Priority Queues

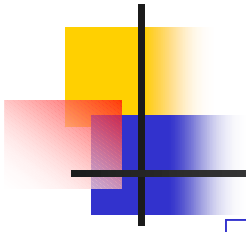
- ***push()***
  - The element is inserted into the largest nonempty block (or, if it is full, into a new larger block and a new navigation pile)
  - $2\log_2 \log_2 n + O(1)$ element comparisons (the iterator and the top headers can need to be updated)
  - 1 element move
  - $O(\log n)$ instructions
- ***pop()***
  - The top element is erased by overwriting with an element taken from the largest nonempty block
  - We need to update also the top and iterator header
  - $\log_2 n + \log_2 \log_2 n + O(1)$ element comparisons
  - 2 element moves
  - $O(\log n)$ instructions

Priority Deques

- We pair the elements and we sort the resulting pairs
- If n is odd, we keep one element in a special block B_0
- The elements get partitioned in two distinct collections:
  top-element candidates and bottom-element candidates
- We build the extra information for the two priority queues
- **Construction**
  - Not more than $\mathbf{1.5}n+\Theta((\log_2 n)(\log_2 \log_2 n))$ comparisons and $n+1$ moves
- ***top()*** and ***bottom()***
  - B_0 can cause $\mathbf{1}$ comparison

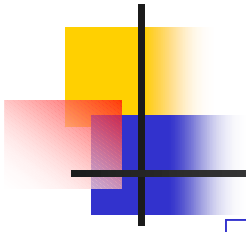
Priority Deques

- ***push()***
  - If B_0 is empty the element being inserted is copied there
  - If B_0 contains an element this and the new one become twins, and this pairs is added to the data structure
  - In the latter case the resource bounds are 2 **times** those compared to the *push()* function



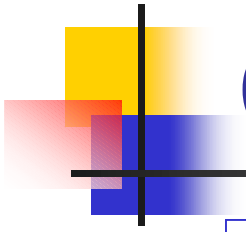
Priority Deques

- ***pop_top()*** and ***pop_bottom()***
  - These two functions are symmetric
  - We consider *pop_top()*
  - If B_0 is empty we move the twin of the top to B_0, take a pair of elements from the last nonempty block
  - We carry out a partial path updating in the last nonempty block
  - We move the pair of elements from the last nonempty block to the block containing the top element and its twin updating a full path of extra information for both the elements



Priority Deques

- If B_0 contains an element this is used now as the new twin of the twin of the top element
- After a single comparison the two elements are moved in their correct locations and we update at most two paths of extra bits
- The resource bounds are about twice as high as those for the priority queues



Conclusions

- It have been showed how it is possible to build and use a new data structure with which, in spite of low memory requirements, the worst-case bounds for the number of element comparisons, element moves and other instructions are close to the absolute minimum
- Is it possible to get similar performance bounds with a sublinear number of extra bits (for example, a sorting algorithm using $n\log_2 n + o(n\log_2 n)$ comparisons, $O(n)$ moves, and $o(n)$ extra bits)?
- Is it possible to improve the *push()* function requiring a constant time without making worse the worst-case bounds for the other operations?