Review of 4 comparison-based priority queues: Binary, Binomial, Fibonacci and Weak heap : Technical report LUSY-2012/01

Matevž Jekovec, Andrej Brodnik
University of Ljubljana, Faculty of Computer and Information Science,
Tržaška 25, SI-1000 Ljubljana, Slovenia
{matevz.jekovec,andrej.brodnik}@fri.uni-lj.si

Abstract

In this paper we review 4 comparison-based priority queues introduced prior to year 1993 (binary heap, Binomial heap, Fibonacci heap, Weak Heap) and analyse their worst-case, average-case and amortized running times.

Keywords: data structures, priority queues, sorting, binary heap

1 Introduction

Selecting or designing the appropriate data structures and algorithms for your application is the most important process in software design from the performance point of view. In this paper we give a brief overview of 4 comparison-based priority queues published before 1993: the original binary heap[7], Binomial heap[6], Fibonacci heap[4] and the Weak-Heap[2]. The review focuses on the theoretical worst-case, average-case and amortized time complexity of element comparisons and justifies it by providing necessary implementation details.

2 Priority queues

Priority queue is a data structure which offers the following basic operations:

- **INSERT**(h, x) Inserts the element x to the heap h.
- **FIND-MIN**(h) Returns the minimum element in the heap h.
- **DELETE-MIN**(h) Removes the minimum element from the heap h.

In practice priority queues were extended with the following operations:

- **DELETE**(h, x) Deletes the element x from the heap h.
- **MERGE**(h_1, h_2) Merges the heap h_1 and the heap h_2 into a single heap.
- **DECREASE-KEY**(h, x, v) Decreases the element x's priority in heap h to v.

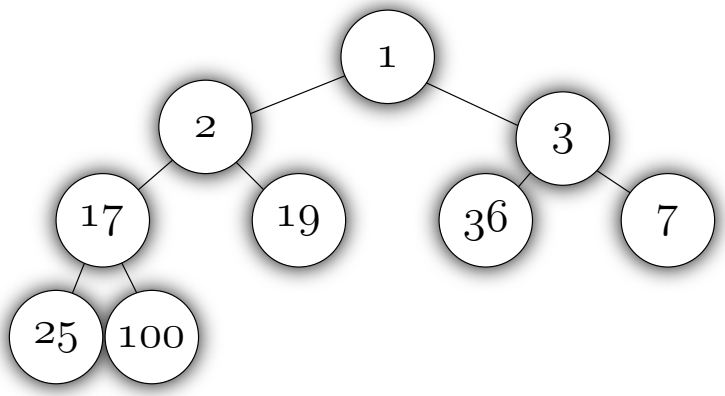

Figure 1: Binary min-heap, $h = 4$.

- **LEFT**(h, x) Finds the closest element left of x in the heap h.
- **RIGHT**(h, x) Finds the closest element right of x in the heap h.
- **FIND-CLOSEST**(h, x) Finds the closest element to x in the heap h.
- **FIND-MAX**(h) Returns the maximum element in the heap h.

We will describe the basic operation implementations for all reviewed priority queues and also a subset of extended operations for some of them.

Priority queues are used in algorithms for finding the shortest paths (Dijkstra), minimum spanning tree (Kruskal, Primm) or in applications like the task scheduler in the operating system and when providing Quality of Service on network switches.

2.1 Binary heap

Binary heap was introduced in 1964 implicitly in the algorithm called "HeapSort"[7]. In this section we will describe the original data structure and ignore some later improvements[3].

The binary heap is a binary tree. Binary min-heap introduces *heap order* which requires that the node is lesser (or greater in binary max-heap) than both of its children. Consequently the smallest (largest) element in the binary heap is the root. Figure 1 shows the binary min-heap. The data structure can be stored inside the array with no additional space required. The element at index i has the left child at index $i*2$ and the right child at index $i*2+1$. The parent of element i is located at $\lfloor i/2 \rfloor$.

2.1.1 Operations

Before looking at the operations we will define the *Heapify(h, i)* helper method first:

1. compare the element i with both of its children,
2. if the smaller (greater in max-heap) child is smaller (greater) than the element i, swap them,
3. if the elements were swapped recursively call *Heapify(h, i/2)*.

If we know that only a single element might be larger than the parent, we can simply compare the potentially larger one.

Operation INSERT adds the element to the leaves from left to right and calls the *Heapify* method on its parent comparing the new element only. This method moves the new element like a bubble towards the root until the *heap order* is satisfied. This requires at most h comparisons per insert operation where h is the current height of the heap. DELETE removes the element from the heap and fills the gap with the last element in the heap (the right-most leaf in the last level). The *Heapify* is called on the element, but this time we want to deepen the element until the *heap order* is reached. The operation requires at most comparisons when removing the root. That is $2(h-1)$, where h is the original height of the tree. FIND-MIN simply returns the root element. It does not require any comparisons and the structure remains intact. MERGE traverses the smaller heap in a bottom-up manner and inserts each element to the larger heap. This requires at most mh comparisons, which happens if each newly inserted element needs to be lifted until the root is reached. Note that m indicates the number of elements in the smaller heap and h is the height of the larger heap. DECREASE-KEY (or INCREASE-KEY in max-heap) simply calls the *Heapify* method on the element's parent only comparing the changed element until the *heap order* is satisfied. This requires at most h comparisons, if the new value is smaller (larger) than the original one, because the new element might only float up. If the new value is larger (smaller), we need to call *Heapify* method on the element which in worst case takes $2(h-1)$ comparisons. Other operations are not defined in the initial paper.

2.1.2 HeapSort

We use the binary max-heap for sorting an array of elements. Initially we call the *Heapify* method on elements indexed from $\lfloor n/2 \rfloor$ to 1. The root now contains the largest element. Afterwards we call DELETE-MIN and fill the array from backwards until the heap is empty. The inline version of the algorithm works as follows:

1. swap the root element with the last element in the heap,
2. decrease the heap size by 1 (we forget the last element and leave it in the array),

3. recursively call *Heapify* on the root until the *heap order* is restored.

4. repeat from step 1 until the heap is empty.

The total time needed for the HeapSort consists of two times: the initial heapify time to build the heap and the heapify times for the new elements when removing the root. The initial time to build a heap where h is the current depth looking bottom-up is described by equation 1.

$$\sum_{h=0}^{\lceil \lg n \rceil} \frac{n}{2^{h+1}} O(h) = O(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^h}) \leq O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n) \tag{1}$$

This means that heapifying the whole structure takes $O(n)$ time and is faster than inserting each element manually which would take $O(n \lg n)$.

The time to heapify new elements can only be done by deepening the root until the *heap order* is restored which takes at most $O(\lg n)$ comparisons per element.

The total time of the HeapSort is therefore $O(n) + nO(\lg n) = O(n \lg n)$ worst-case.

2.2 Binomial heap

Binomial heaps were introduced in 1978 by Vuillemin[6]. The main goal was to speed up the MERGE operation of two heaps (that is $O(mlogn)$ for binary heaps).

The Binomial heap is a forest of Binomial trees with unique degrees k. Every Binomial tree comprises of exactly 2^k nodes. This gives us at least 1 and at most $\lceil log_2 n \rceil$ trees in the non-empty Binomial heap of n elements. The former happens, if n is a power of 2 and the latter, if n is a power of 2 minus 1. The Binomial tree is named after the following statement. Let i be the depth of a node, then node i has exactly $\binom{k}{i}$ descendants where k is the tree degree. The root node always has the largest number of children, that is exactly k.

The data structure was initially implemented using 3 arrays: `INFO` containing the element's label, `LLINK` containing the reference to the left-most child and `RLINK` containing the reference to the element's right sibling.

2.2.1 Operations

Before looking at the MERGE operation of two Binomial heaps, we first define merging of two Binomial trees. This operation is only defined on trees with the same degree. If we use the min-heap order, the tree having the larger root becomes a child of the smaller root element. This way the new Binomial tree degree is increased by 1. Notice that both the *heap order* and the Binomial tree form is preserved. MERGE operation on Binomial heap starts by merging trees from the lower to higher degree. If a Binomial tree doesn't exist in the target Binomial heap, we simply add it. If a Binomial tree already exists in the target Binomial heap, we call the MERGE operation on the two Binomial trees.

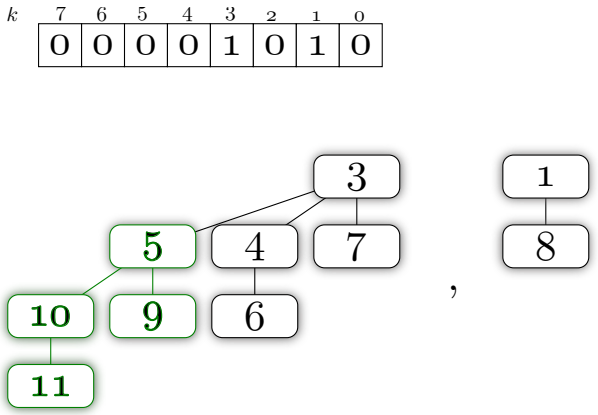

Figure 2: Binomial heap with 10 elements. It comprises of two Binomial trees of degrees $k = 3$ and $k = 1$. The structure of a single Binomial tree can be observed on the left: it comprises of two Binomial trees with degree $k = 2$ (the green one and the black one with 4 elements each) merged together having the smaller of the roots as the new root element 3. On top, the array describing the presence of Binomial trees in the heap.

Recursive merges might occur, if a Binomial tree with the increased degree overlaps the existing ones. Time complexity of the MERGE operation of two Binomial heaps is $\lceil logn \rceil$ (the number of Binomial trees) in worst case, where n is the number of elements in the larger heap. Figure 2 shows a Binomial heap with 10 elements.

Merging is usually implemented using the array of size $\lceil logn \rceil$ containing the structure of the heap. The value at index i is 1, if a Binomial tree with degree i exists in the heap or 0, if such a tree does not exist.

INSERT is implemented by constructing a Binomial heap consisting of a single element and merging the generated Binomial heap with the existing one. This takes exactly the same time as the MERGE operation does.

FIND-MIN operation instead of walking through the Binomial trees roots, simply returns the cached minimum element of the heap. The registry containing the minimum element needs to be checked on every INSERT, DELETE-MIN and DECREASE-KEY operation and then updated accordingly. Reading from this registry requires $O(1)$ time.

DELETE-MIN is defined by removing the minimal root node. By doing this we brake one Binomial tree of degree k to a set of k Binomial trees with degrees ranging from 1 to 2^k exponentially. We then merge these children with the original Binomial heap. This takes at most $\lfloor logn \rfloor$ comparisons, where n is the number of elements before deletion. The worst case happens when n is $2^x - 1$.

DECREASE-KEY simply lifts the given element inside the Binomial tree until the *heap order* rule is satisfied again. This takes at most $\lfloor logn \rfloor - 1$

comparisons.

DELETE is implemented by calling DECREASE-KEY($-\infty$) and DELETE-MIN. This takes the sum of time complexities of these operations.

2.3 Fibonacci heap

The Fibonacci heaps were introduced in 1987 by Fredman and Tarjan[4]. The main motivation was to improve the amortized time of operations, which is a more realistic time when dealing with a sequence of operations than the average or worst-case time.

The Fibonacci heap is based on the Binomial heap. It comprises of set of trees with unique degree k. The trees are initially Binomial trees, but might be broken by at most one missing child per node.

2.3.1 Operations

The MERGE operation of two trees is the same as of the Binomial trees. We can link only the same-degree trees together and put the smaller root element on top. The MERGE operation on two Fibonacci heaps however, is completely different than the one defined on Binomial heaps. Fibonacci heap uses the *lazy melding*. New trees are simply appended to the heap allowing multiple trees of the same degree. Fibonacci heap requires $O(1)$ time for the MERGE operation.

INSERT operation works similar to the one described in the Binomial heap. It creates a new heap containing a single element, but then calls the MERGE operation as described above. This requires $O(1)$ time.

FIND-MIN operation works exactly the same as in the Binomial heap. It requires $O(1)$ time.

DELETE-MIN works the same as in the Binomial heap. It first removes the minimum node and then calls the MERGE operation on trees with the same degree. The latter step is called the *linking step* and should not be mixed with the *lazy melding*, which happens in the MERGE operation of the whole Fibonacci heap. The time complexity of the *linking step* differs from that in the Binomial heap. The number of newly added trees when removing the root node is the same, but the number of existing trees in the heap is no longer $logn$, but might be n. This happens when calling DELETE-MIN the first time after n INSERT operations. Time needed to perform the DELETE-MIN operation is impossible to calculate because it depends on the sequence of previous operations. It might take from $\lfloor logn \rfloor$ to $n - 1$ comparisons.

DELETE operation instead of lifting the node up and removing it as in the Binomial heap, it removes the node from the heap directly and calls the MERGE operation of its children with the heap. Note that the *linking step* is not executed. To preserve some form of the Binomial trees in the heap when deleting, the *cascading cut* is introduced. This rule limits the number of node's lost children to at most 1. If more than 1 child is lost, the node itself is removed from the tree and merged with the heap in the lazy manner. This rule prevents trees from becoming shallow and spread after a series of DELETE operations

on the same tree which would then cause more than *logn* newly added trees in DELETE-MIN operation. The adjective "cascading" origins from the fact that the cut, because of removing the node, might trigger another cut on the parent node recursively. Operation DELETE requires $O(1)$ time on average.

DECREASE-KEY operation decreases the element's value, moves it to its own tree and calls the MERGE operation with the existing heap. *linking step* is not called. However, because the node is removed from the original tree, *cascading cut* might be triggered. This operation requires $O(1)$ on average.

2.3.2 Why Fibonacci?

The *cascading cut* limits every node to lose at most one child before being re-linked. Let S_k be the minimum possible number of descendants of a node including the node itself with k children. It turns out that the number of descendants of the node is at least ϕ^k (where ϕ is the golden ratio constant $\frac{1+\sqrt{5}}{2}$). Let's look at this more closely: Obviously $S_0 = 1$ and $S_1 = 2$. Because we always merge together trees with the same k and because of the *cascading cut* we can write the following: $S_k \geq \sum_{i=0}^{k-2} S_i + 2$ for $k \geq 2$. Finally we can write $S_k \geq F_{k+2} \geq \phi^k$ where F_k is the k-th Fibonacci number.

2.3.3 The amortized analysis

Tarjan introduced the amortized analysis in 1985[5]. In comparison to the average or worst-case complexity, the amortized analysis is done on a sequence of operations. This better reflects the real-world cases and is sometimes easier to calculate than the average-case time complexity per single operation. Tarjan explained the amortization using two points of view:

the banker's view We decide to spend i coins every time when calling the operation. The operation does not necessarily always need exactly i coins — sometimes more, sometimes less. The difference is deposited or withdrawn from the *bank*. The goal of the analysis is to calculate the minimum amount of coins i spent every operation call for a sequence of operations to successfully finish (ie. no negative balance on the *bank*). i is called the average amortized time. It turns out that i is larger than the average-case time complexity of the single operation. The difference is accumulated in the remaining balance on the *bank*. We also decide what the coin is used for — usually for a single comparison or the memory access.

the physicist's view We select the "potential" and calculate the difference of it when calling a sequence of operations. The potential is a data structure property that influences on the required times for operations to finish and can be calculated directly in every state of the data structure (you don't need to know the history of the data structure). Let's name a few examples:

- the tree height in splay tree,
- the difference in subtrees' height in AVL trees,
- the number of trees in the binomial heap,
- the number of black nodes in red-black tree.

The equation 2 describes the behaviour of the potential as described by Tarjan: the sum of actual average times for the operation equals to the initial minus the final potential plus the sum of all amortized times per operation. This is similar to the banker's view — average a_i is the minimum number of coins required for a sequence of operations to successfully finish.

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} (a_i - \Phi_i + \Phi_{i-1}) = \Phi_0 - \Phi_m + \sum_{i=1}^{m} a_i \tag{2}$$

2.3.4 The amortized analysis of the Fibonacci heap

We will analyse the Fibonacci heap from the physicist's point of view. The potential of the Fibonacci heap is the number of the trees p plus two times the number of marked nodes. The marked nodes are those which lost a single child during the DELETE or DECREASE-KEY operation. The node is unmarked when it is removed from the tree.

Let's observe the potential behaviour: MERGE operation increases the potential by the number of the trees in the second heap. INSERT operation increases the potential by exactly 1. FIND-MIN operation does not change the potential. DECREASE-KEY moves the decreased node to its own tree and increases the potential by exactly 1. DELETE operation adds the removed node children to their own trees. These are at most $\lg 2$. DELETE-MIN adds the removed node children to their own trees, but also calls the *linking step* which decreases the potential. DELETE-MIN can increase the potential by $\lg 2$ worst-case.

The *lazy melding* of Fibonacci heap increases the performance of a number of problems. One would say that the actual number of comparisons is eventually the same as in the Binomial heap. This is not true — let's have a sequence of n INSERT operations and a single DELETE-MIN. Binomial heap requires $n - 1$ comparisons in worst case for the INSERT operations (when $n = x^2 - 1; x \in \mathbb{N}$) plus $\lfloor \lg n \rfloor$ comparisons for the DELETE-MIN. Fibonacci heap on the other hand does not require any comparisons for the INSERT operations and then in DELETE-MIN operation first removes the element and then runs the *linking step* requiring a total of $n - 2$ comparisons.

This is also the case for the Dijkstra's shortest-path algorithm. The goal is to find the shortest paths from the initial node to all others in the graph. Let's denote n as the number of nodes and m as the number of edges where $n < m < n^2$. The algorithm requires m DECREASE-KEY and n DELETE-MIN

operations. Fibonacci heap implementation of Dijkstra runs in $O(n \lg n + m)$ time, improved from the fastest known to date Johnson's $O(m \lg_{m/n+2} n)$ bound.

2.4 Weak heap

The weak-heap data structure was first introduced in the technical report[1] and was originally used for sorting[2]. Weak-heap is defined using these three rules:

- every key in the right subtree of a node is smaller than the key stored in the node itself (the *weak-heap order*),
- the root has no left child,
- leaves are only found on the last two levels of the tree (weak-heap is always balanced).

Note that these rules describe the maximum weak-heap in contrast to the minimum *heap order* usually found in the binary heap. We us the maximum weak-heap because it is more appropriate for inline sorting. Figure 3 illustrates this structure.

Let us define $GParent(i)$ as the ith parent, if i is the root of the right subtree or as the parent of the right-most parent, if i is in the outer line of the left subtree. Figure 3 presents the $GParent$ relation visually. The first weak-heap rule mentioned above can be interpreted as "all nodes should be smaller of their $GParent$". We also define $GChildren(i)$ which returns all elements whose $GParent$ is the node i.

The original implementation of the weak-heap uses the array representation. Because the weak-heap is built using a breadth-first manner, we can find out whether a node is in a left or a right subtree by simply checking, if the node's index is even or odd. We retrieve the node's parent by assigning $i = \lfloor i/2 \rfloor$. This is useful from the time complexity point of view because we can simply use right bit-shifting until the right-most bit is 1. Therefore, calculating $GParent$ can be done in $O(1)$ on modern CPUs and does not require any node comparisons.

Let us also define the *max-element-path*. This is a path comprised of $GChildren(root)$ elements. The *max-element-path* always contains 2nd largest element in the weak-heap (the largest element is obviously the root of the weak-heap). Another important observation is the location of the 3rd largest element in the weak-heap. It is located in the *max-element-path* below the 2nd largest element or is one of the $GChildren(2nd\ largest\ element)$.

2.4.1 Operations

Let us also define a helper function called $Merge(i, j)$. This function swaps the elements i and j, if $i < j$ and reverses the left/right subtree of i. If $j > i$, this function does nothing. Therefore, the order of the elements i and j is important! Reversing the smaller node subtrees is necessary in order to preserve the *weak-heap order*. Note that the initial requirement for the $Merge$ is that the jth left subtree is smaller than i. Figure 4 presents this operation.

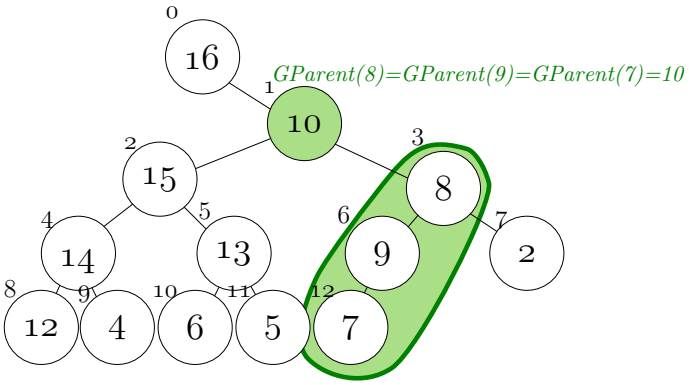

Figure 3: The weak-heap. Node array indices are written in the top-left corner of a node. Node 10 is *GParent* of nodes 8, 9, 7.

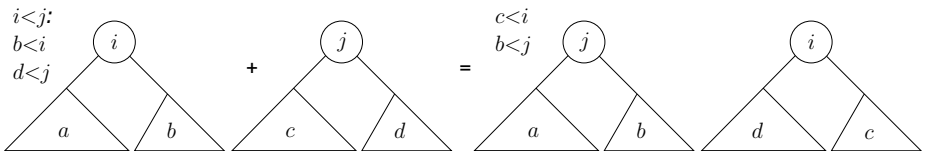

Figure 4: The weak-heap $Merge(i, j)$ operation.

The operation INSERT adds the element i to the weak-heap as a new leaf and recursively calls $Merge(GParent(i), i)$ until the larger $GParent$ is encountered. This "heapifies" the newly added element and preserves the *weak-heap order*. Time complexity depends on the implementation and the measurement unit. If we use the array implementation and count the number of element comparisons, we require from 1 (new element is the left-most node in the weak-heap) to at most $log(n) - 1$ comparisons (new element is the largest inserted and the right-most node in the weak-heap).

The operation FIND-MAX simply returns the weak-heap root element. This operation requires 0 comparisons. Another important observation is the location of the 2nd largest element in the weak-heap. It is one of the nodes having the root element for their $GParent$. There are exactly $\lceil logn \rceil$ of those.

The operation DELETE-MAX works as the following:

1. remove the root element (the maximum element) and replace it with a dummy element with value $-\infty$,
2. call $Merge(-\infty, x)$ where x is the 2nd largest element in the weak-heap,
3. continue using the left subtrees ($x = 2x$) and recursively call $Merge(-\infty, i)$ until the leaf is reached,
4. remove the $-\infty$ element.

When recursively calling the $Merge$ function in the 3rd step, we must never forget to reverse the left/right subtree. This ensures the *weak-heap order* invariance. DELETE-MAX operation requires from $\lfloor logn \rfloor$ ($logn$ to find the largest element in $max - element - path$ and 0 for filling the gap, if the next largest element is the bottom one) to $\lceil 2logn \rceil$ comparisons ($logn$ to find the largest element in $max - element - path$ and another $logn$ to fill all the gaps recursively until we reached the bottom of the $max - element - path$).

2.4.2 The weak-heap sort

If we represent the data in the weak-heap using the breadth first manner, sorting is done in the following steps:

1. weak-heapify the whole data set by calling $Merge(GParent(i), i)$ for each $i = n - 1..1$,
2. remove the root and write it at position n. Our weak-heap gets a shape of an ordinary binary tree,
3. call $MergeForest(i)$ for each $i = n - 1..2$.

The resulting sorted array of elements is located at $1..n$ (data set gets shifted for 1 element to the right).

The weak-heapify operation takes exactly $n - 1$ comparisons. In comparison to the weak-heapify of the newly inserted node described in the previous section

(and the similar behavior noticed in the binary heap), we can only call $Merge$ once per element because the larger element will eventually reach the root and the smallest elements can be anywhere in the heap as long as we satisfy the $weak - heaporder$.

Function $MergeForest(i)$ calls $Merge(i, j)$ for each j on the *max-element-path* going bottom-up. Eventually the largest element appears at the original index i and the *max-element-path* is changed in the way that it contains the new largest element for the next step. Let us investigate this further. The element i is swapped with one of the j elements, if the current value at i is smaller than the one at j. This means that the new j gets smaller and we cannot assure that the elements in the right subtree are smaller anymore. To resolve this, we complete the $Merge$ procedure and we reverse the left/right subtree. We can do this, because we know that the elements in the former left tree are all smaller than our new element j (if it were not so, the j element would have been swapped before). And for the former right tree, we know nothing about their relation to our new j, so we must move them to jth left subtree to assure the *weak-heap order*.

We would like to stress out another observation. By reversing the subtrees in the $MergeForest$ step, we also changed the *max-element-path* for the next step. We would like to prove that the new path really contains the next largest element. One might argue that when we reversed the jth element subtrees, we lost potentially largest elements for the next step located in jth former left subtree. This is not true, because the jth element was the largest one in that moment, which means that j itself is already the largest element candidate for the next step. What if we lost potential elements for the two steps ahead? If that were correct, we would reverse the subtree back again in the next step anyway, because the 3rd largest element can only be present in the 2nd largest element right subtree in that case.

The worst case of the $MergeForest$ operation requires $\frac{n}{2}logn$ comparisons on the last level, $\frac{n}{4}(logn - 1)$ on the pre-last level, $\frac{n}{8}(logn - 2)$ on another level etc. This gives us exactly $(n-1)logn-0.913987n+1$ comparisons for the whole weak-heap.

Finally, if we sum the number weak-heapify and the $MergeForest$ comparisons the weak-heap sort requires at most $(n-1)logn+0.086013n$ comparisons which is better than other comparison-based heap sorts. Empirical results show that the average number of comparisons is $(n-0.5)logn-0.413n$ which is better than the Bottom-Up heapsort from [referenca!], MDRS[referenca!] and even QuickSort [referenca!] when $n > 500$.

2.4.3 Speeding up the $Merge$ operation

In this section we would like to introduce another important detail concerning the weak-heap sort implementation in order to reach the above mentioned times. In the previous section we often used the reversal of the subtrees. But if the elements are stored in the array, this operation takes 2^k swaps, where k is the height of the node. To avoid this we add another *reverse* bit to each node, so

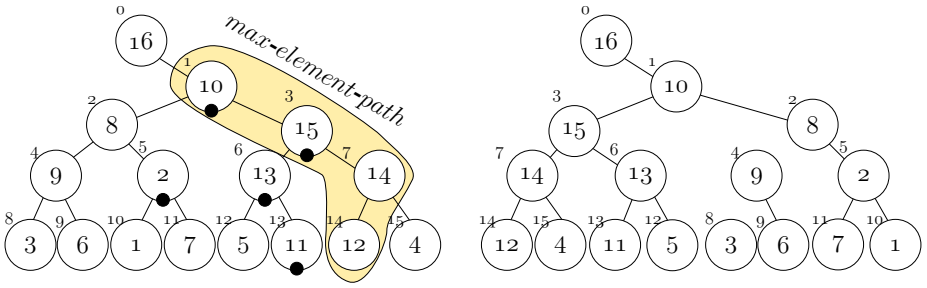

Figure 5: Left side: Weak-heap represented in memory. Reversed nodes are punctuated. $max - element - path$ is shaded. Right side: Interpreted weak-heap.

when we reverse the node, we simply negate this bit.

Two changes are required in the $GParent$ and $max - element - path$ procedures. For the $GParent$, before checking, if the current node index i is even or odd, we simply add the *reverse* bit to the index i. This also affects the interpretation of the *weak-heap order*. For the nodes with *reverse* set to 1, all elements in the left subtree should be smaller than the node i (instead of the right subtree). Redefining $GParent$ also redefines the $max - element - path$. Figure 5 shows the actual weak-heapified array of random numbers from 1..16 in the memory and the interpreted weak-heap on the right.

3 Conclusion

Table 1 shows the time complexities for described operations in the previous chapters. Fibonacci heap with its *lazy melding* and *cascading cut* is the best general-purpose comparison-based heap. It achieves the best amortized performance in comparison to the binary heap or the binomial heap.

Procedure	**Binary heap (worst-case)**	**Binomial heap (worst-case)**	**Fibonacci heap (amortized)**
INSERT	$\Theta(logn)$	$O(logn)$	$\Theta(1)$
FIND-MIN	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
DELETE-MIN	$\Theta(logn)$	$\Theta(logn)$	$O(logn)$
MERGE	$\Theta(n)$	$O(logn)$	$O(1)$
DECREASE-KEY	$\Theta(logn)$	$\Theta(logn)$	$\Theta(1)$
DELETE	$\Theta(logn)$	$\Theta(logn)$	$O(logn)$

Table 1: Running time for operations on three implementations of mergeable heaps. The number of items in the heaps at the time of an operation is denoted by n.

Weak heap on the other hand is specialized on fast inline sorting and is the fastest comparison-based heapsort to date.

References

[1] Dutton, R. The weak-heap data structure. Tech. rep., University of Central Florida, Orlando, FL 32816, 1992.

[2] Dutton, R. Weak-heap sort. *BIT Numerical Mathematics 33*, December 1992 (1993), 372–381.

[3] Floyd, R. Algorithm 245: Treesort. *Communications of the ACM 7*, 12 (1964), 701.

[4] Fredman, M. L., and Tarjan, R. E. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM 34*, 3 (July 1987), 596–615.

[5] Tarjan, R. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods 6*, 2 (1985), 306–318.

[6] Vuillemin, J. A data structure for manipulating priority queues. *Communications of the ACM 21*, 4 (Apr. 1978), 309–315.

[7] Williams, J. Algorithm 232: heapsort. *Communications of the ACM 7*, 6 (1964), 347–348.