Algorithms and Data Structures for New Models of Computation

Article *in* IT Professional · January 2021

DOI: 10.1109/MITP.2020.3042858

3 authors:

Paul Evan Black
National Institute of Standards and Technology
81 PUBLICATIONS **2,262** CITATIONS
SEE PROFILE

David Flater
71 PUBLICATIONS **306** CITATIONS
SEE PROFILE

Irena Bojanova
National Institute of Standards and Technology
65 PUBLICATIONS **971** CITATIONS
SEE PROFILE

Department: Software Technology
Editor: Name, xxxx@email

Algorithms and Data Structures for New Models of Computation

Paul E. Black
National Institute of Standards and Technology (NIST)

David Flater
NIST

Irena Bojanova
NIST

***Abstract*—In the early days of computer science, the community settled on a simple standard model of computing and a basic canon of general purpose algorithms and data structures suited to that model. With isochronous computing, heterogeneous multiprocessors, flash memory, energy-aware computing, cache and other anisotropic memory, distributed computing, streaming environments, functional languages, graphics coprocessors, and so forth, the basic canon of algorithms and data structures is not enough. Software developers know of real-world constraints and new models of computation and use them to design effective algorithms and data structures. These constraints motivate the development of elegant algorithms with broad utility. As examples, we present four algorithms that were motivated by specific hardware nuances, but are generally useful: reservoir sampling, majority of a stream, B-heap, and compacting an array in $\Theta(\log n)$ time.**

STANDARD MODEL AND CANON

In the early days of computer science, the community settled on a simple standard model of computing [5] and a basic canon [14] of general purpose algorithms and data structures. The standard model of computing has (a) one processor that performs a single instruction at a time, (b) copious memory that is uniformly accessible with negligible latency, (c) readily available persistent storage with significant latency, (d) input and output that is insensitive to content or context and orders of magnitude slower, (e) resources and connections that do not change, and (f) no significant concerns about energy use or component failure, permanent or intermittent.

The canon of data structures and algorithms includes lists, quicksort, Boyer-Moore string search, trees, and hash tables. Knuth even began

 Published by the IEEE Computer Society

what was intended to be a complete encyclopedia of computing [10]. Although "new and better" sort routines are regularly sent to the editors of the Dictionary of Algorithms and Data Structures [1], none has been both new and widely applicable. The most recent general-purpose data structure is the skip list, which was invented in 1989 [15]. Work continues in specialized areas, such as graphics, distributed systems, user interaction, cyber-physical systems, and artificial intelligence. Except perhaps for quantum computing algorithms, general textbooks have the same model and cover similar topics.

New Models

Programmers will not be prepared for modern systems with only this background. Few actual computers match the standard model in all aspects. Although hardware, caching, and system support well approximate the standard model, developers must be aware of constraints and idiosyncrasies of the software's execution environment. New constraints of time, space, energy, and hardware motivate new ways of thinking and lead programmers to need new approaches to old problems.

So many advances have been made in hardware and systems that slavishly staying within that standard model forgoes much performance. Enticing advantages are available to exploit in isochronous computing, heterogeneous multiprocessors, flash memory, energy-aware computing, cache and other anisotropic memory, distributed computing, graphics coprocessors, networks of many simple devices, and streaming environments. In some cases, there is enormous waste in maintaining or staying within the standard computing model. In other cases, additional resources are squandered if used without regard to their particular strengths and weaknesses.

For example, flash memories, used in solid state drives and memory cards for cameras and smart phones, wear out. Individual bits in flash memory can be set, but bits can only be cleared by erasing an entire block of memory. Blocks of memory can only be erased up to one hundred thousand times before excessive errors occur [8]. The entire block of memory then becomes unusable. To extend their usable life, many flash memory subsystems use sophisticated algorithms to spread writes throughout memory and still maintain the abstraction of persistent memory that is insensitive to write locations. This is often accomplished through journaling file systems or log-structured file systems, whose patterns of writes match flash memories well [7]. Additional efficiencies may be available to the software developer by using data structures specialized to flash memories, such as [19] or [4]. Taking device characteristics in consideration can improve solid state disk responsiveness by 44 % [8].

Similar constraints and opportunities arise because of multicore machines, in which there is parallelism on every desktop, distributed computing, in which one can use hundreds of machines only for milliseconds, and cloud computing, in which extra processing is always available for a price with some delay to request and provision additional resources. In addition, micropower and ubiquitous sensors are best used by applications that treat input as a stream of values to be processed instead of a static set of data. Power- and thermal-aware computing is of growing importance in mobile devices and huge server installations. Today we expect computer systems distributed across the entire Earth to function as a single, coherent whole, in spite of speed-of-light delays, which often requires careful caching [6].

EXAMPLE ALGORITHMS AND DATA STRUCTURES

In this section, we give four instances of elegant, generally useful algorithms and data structures that were developed for particular circumstances, but are not widely known. They illustrate the opportunity to create new algorithms and data structures motivated by new computing constraints.

Reservoir Sampling

For the first algorithm, consider a stream environment, that is, items arrive one at a time. The program does not know how many items will arrive, or equivalently, an answer may be required after an unspecified number of items has arrived; see **Figure 1**. Since we don't know how many items there will be, we limit resources to be a constant amount of memory, plus a counter that requires $\log n$ bits.

As a real world example, imagine a network

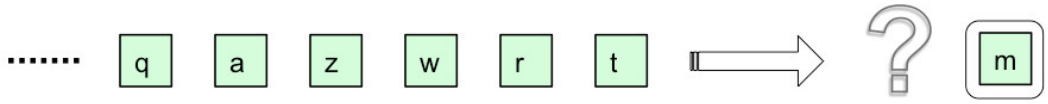

Figure 1. The problem is to randomly select an item from a stream of items of unknown length using a near-constant amount of memory. All items must have an equal chance of being selected.

of tiny sensors detecting pollution. Randomly a sample is requested from every sensor in the network. The tiny sensors cannot store many samples, and for statistical purposes all samples must be equally likely.

The challenge is to choose one of the items randomly with all items having an equal chance of being chosen. If the number of items is known beforehand, one could generate a random number in the range of the number of items. Alternately, one can store all the items, possibly in some data structure that can grow as needed, and choose one when needed. Neither of these standard solutions satisfy the constraint of not knowing the number of items beforehand.

A simple solution is attributed to Alan G. Waterman [11]. It requires storing the item chosen so far and a counter, n, that holds the number of items from the stream so far. To begin, store the first item and set n to 1. For each new item, replace the chosen item by the new item with probability $1/(n+1)$ and increment n. The proof of algorithm correctness is simple, too.

Clearly we end up with one of the items. But does every item have an equal chance of being chosen? We sketch an induction proof. The first item is kept as the chosen item after the first step with a chance of $1/1$, which is correct. The second item is kept with a probability of $1/2$, so either item has a $1/2$ chance of being chosen after the second step.

For the induction step, assume that every one of n items has an equal chance, $1/n$, of being the chosen item. When the $n+1^{\text{st}}$ item arrives, it replaces whatever item had been chosen with probability $1/(n+1)$. The previously chosen item is therefore retained with probability $n/(n+1)$. Multiplying, we find that each previous item has a chance of being kept of $n/(n+1) \times 1/n = 1/(n+1)$; see **Figure 2**. So every one of the $n+1$ items has an equal chance of being chosen.

Outside of a stream processing environment, why should this subtle algorithm be used? Typically everything is in memory and a program can do two passes: the first pass counts the number of items, then the second pass chooses one at random. Suppose that selection is computationally intensive, for example, only numbers that are prime are candidates to be chosen randomly. It would be inefficient to check for primality in the first pass to count, choose a random number k, then repeat the primality checking when counting through to find the k^{th} item. A program might cache the primes identified in the first pass, but then the program is more complicated.

Adapting the reservoir sampling algorithm allows the following elegant program structure:

```
for every item
    if this item is prime then
        decide whether to keep it
```

As several authors have pointed out, reservoir sampling can be easily extended to pick m items at random from a stream. The insight is to keep a new item $n+1$ with probability $m/(n+1)$. If we decide to keep a new item, randomly choose one of m items to replace.

This can be used to efficiently choose a random sample from a database. Typically one selects all candidates from a database, then chooses a random sample. It may be far more efficient to perform a single pass, randomly choosing items in the process.

(Following paragraph not in published article.)

Parallel hardware readily speeds up reservoir sampling. Each parallel thread chooses an item from its subprocesses weighted by the number of items in each subprocess's subtree. The thread returns the item chosen and the total number of items.

Majority in a Stream

The next algorithm we consider also takes place in a stream environment. Suppose items occur multiple times, and one of the items is

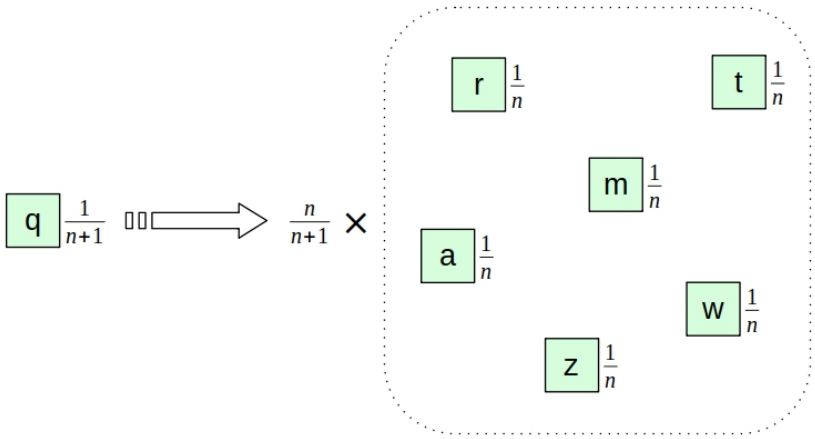

Figure 2. Suppose any one of n previous items was chosen with equal probability. Item $n+1$ is chosen with probability $1/(n+1)$. Previous items have a probability of $n/(n+1) \times 1/n = 1/(n+1)$ to be chosen, too.

known to occur more than half the time. As a real world example, consider an ad hoc set of tiny, low-power sensors, communicating in a ring, that need to select a majority item; see **Figure 3**. Although the sensors can pass their items around the ring, no sensor can store all items to tally the majority item—no sensor even knows the total number of sensors still operating. What algorithm could a single sensor use to identify the majority item in the stream?

In an ad hoc network, votes for the lead node can be circulated and each node can identify the majority node as the lead node using only memory for one item and one counter. Votes may arrive in different orders at each node.

Boyer and Moore gave a straightforward solution [2]. Storage is needed for two things: a tentative majority item and an "unmatched" count. Note that the count is either zero or positive. Begin by setting the count to zero. When an item arrives, if the count is zero, set the tentative majority item to be the arriving item and set the count to one. If the count is positive and the arriving item matches the majority item, increment the count. If the count is positive and the arriving item does not match, decrement the count. At the end, the tentative majority item *is* the majority item, if one exists.

If no item occurs a majority of times, any item might be the tentative item at the end. There is no guarantee that the item with a plurality will be kept.

The correctness of the algorithm can be based on a pairing argument; see **Figure 4**. Every item that is not the majority item can be paired with a majority item somewhere in the stream. That is, the count for each pair of majority and non-majority items will cancel out. Either the non-majority item is tentative and the majority item decrements its count, or the majority item is tentative and the non-majority item decrements the majority item count. In either case, there is at least one majority item that will not be paired—there are more of them than all non-majority items. This item will get a positive count that is not canceled.

Virtual Memory Aware Heap

Next we present an implementation for the classical binary heap that works better with modern memory systems. Binary heaps are used in priority queues and the heapsort algorithm. Typically heaps are implemented with an array. A node at index `i` has child nodes at indexes `2i` and `2i+1` and a parent node at index `i/2`, with one-based indexing [12]. The problem is that for large heaps, (almost) every level ends up in a different virtual memory page, see **Figure 5**. Thus traversing from the root to a leaf causes a page fault for almost every level and occupies a significant amount of cache.

The B-heap [9] is a modification that allocates significant subtrees in a single page, see **Figure 6**. Typically pages are large enough to hold subtrees with a dozen levels. Thus the number of page faults is reduced up to an order of magnitude,

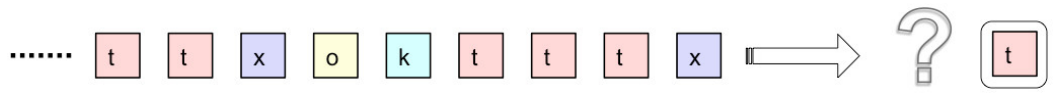

Figure 3. The problem is to find which item in a stream occurs the majority of times using a near-constant amount of memory. In this example, more than half of the items are "t."

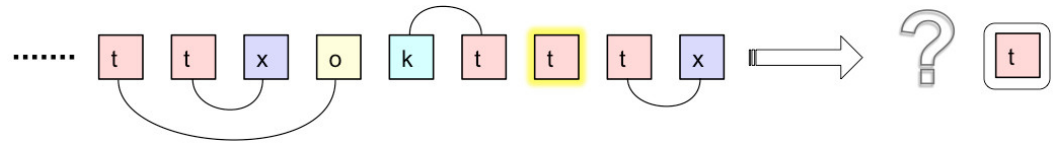

Figure 4. Diagram suggesting the correctness of the algorithm. The majority items can be paired with other items and have some remaining.

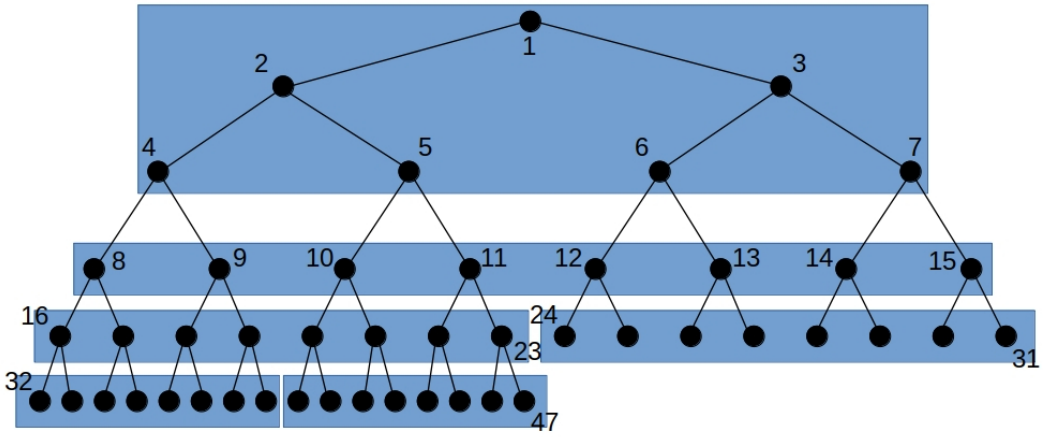

Figure 5. Naive binary heap implementations lead to many page faults and cache pollution since almost every level is in a different virtual page.

using far fewer entries in the cache and page tables. Other variants, such as cache-oblivious algorithms, implicit k-way heaps [13], min-max heaps [16], and van Emde Boas layouts [17], similarly work far better with actual virtual memories.

Compacting an Array

Finally we present a parallel algorithm. With multiple core machines being the norm for personal computers, phones, and even embedded devices, the community must be aware of the opportunities of parallel programming. Cloud and distributed computing makes the need even more urgent.

The problem we consider is compacting an array. That is, copy all non-zero entries from one array to (the beginning of) another array. A simple serial algorithm accomplishes this in $\Theta(n)$ time. Can we accomplish this faster using parallel processing with a minimum of complications?

Vishkin gives a parallel algorithm [18] to compact an array in $\Theta(\log n)$ time, assuming $\Theta(n)$ hardware. The input is a shared array `A` of size n. The output is a shared array `B`, assumed to be big enough. To help explain the algorithm, we present the XMTC code for it as a reference.

```
psBaseReg dest = 0;
spawn (0, n-1) {
    int e = 1;
    if (A[ $ ] != 0) {
        ps(e, dest);
        B[e] = A[ $ ];
    }
}
```

`psBaseReg` declares `dest` as a prefix-sum base variable shared among all threads. The key-

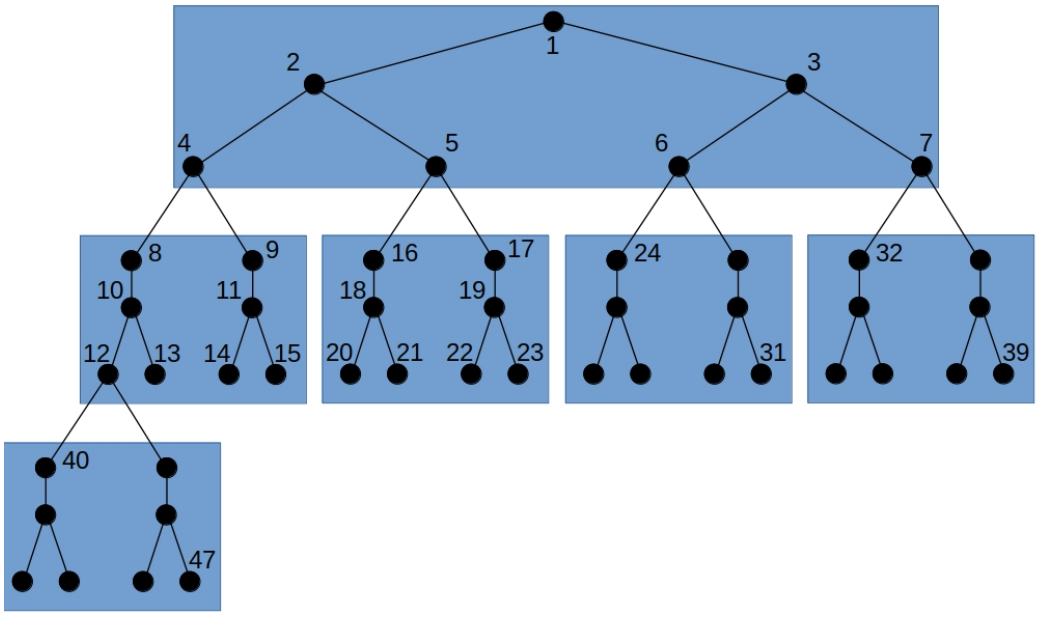

Figure 6. A B-heap allocates subtrees in a single page, which reduces page faults and minimizes cache use. Subtrees have two "root" nodes to use all space in each page.

word `spawn` creates one thread for each index in `A`. Each thread runs its own copy of the code in the body of `spawn`. `$` is the thread number. The interesting part is `ps`, which is the prefix sum operation. Here it atomically copies the value of the shared variable `dest` to `e`, then increments `dest`. It is equivalent to the following two assignments occurring simultaneously:

```
e' = dest;
dest' = dest + e;
```

where the primed variables represent the new values. Since each thread gets a value of `dest` and increments `dest`, each thread gets a unique index for `B`. The indexes for `B` are assigned from 0 to (one less than) the number of non-zero entries.

If threads access `dest` one at a time and `ps` merely serializes the accesses, there is no speed up. In **Figure 7**, we illustrate how the compiler parallelizes executions of `ps` to handle all the updates in $\Theta(\log n)$ time.

The spawned threads are represented by inverted triangles at the bottom of the thread tree in **Figure 7**. Hexagons represent additional threads created to handle intermediate results. Each thread passes the initial value of its `e` (which is 1) up. Each intermediate thread passes up the sum of the `e` value it receives from the left child thread and the `e` value from the right child, e_L+e_R. At the top, the initial value of `dest`, 0, is passed down. Each intermediate thread passes the `dest` it receives from above to its left subtree and `dest` + e_L to the right subtree. All the threads receive distinct `dest` values in $\Theta(\log n)$ time which they use as their `e'` value. Thus every value from `A` is put in a unique (and compact) place in `B`.

Each `dest` value is unique since the least value in a right subtree is just beyond the highest value in the left subtree.

Although compacting an array is not common, prefix sum is used in sorting. Vishkin gives several other commonly used algorithms which can be easily parallelized similarly.

CONCLUSION

New constraints and opportunities in computing, such as highly parallel [3], cloud, and energy-aware computation, can lead to novel, elegant, broadly applicable algorithms and data structures. We call for renewed interest in and resources for research of basic algorithms and data structures. Kamp says [9] programmers should be taught some cache-aware data structures and algorithms to understand the benefits of and need for such improvements. More to the point,

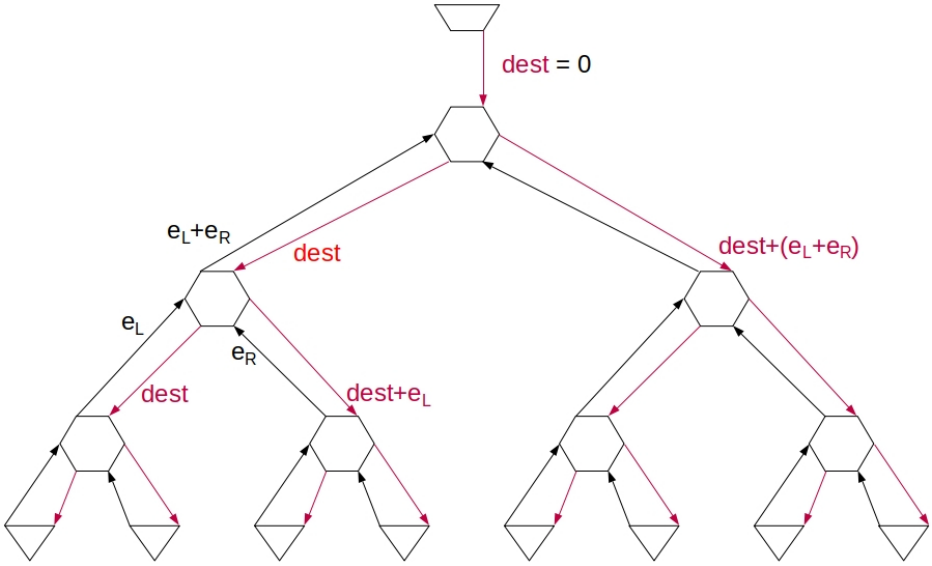

Figure 7. Diagram of threads to compute the prefix sum operation.

such data structures and algorithms must be as conveniently available from standard libraries as "standard model" algorithms currently are. Busy professionals seldom invest extra time or effort to obtain the best tool for a job if an adequate tool is already at hand.

ACKNOWLEDGEMENTS

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

This document was written at the National Institute of Standards and Technology by employees of the Federal Government in the course of their official duties. Pursuant to title 17 Section 105 of the United States Code this is not subject to copyright protection and is in the public domain.

REFERENCES

1. Dictionary of algorithms and data structures. P. E. Black, ed. [Online]. Accessed 21 April 2020. Available: https://doi.org/10.18434/T4/1422485
2. R. S. Boyer and J. S. Moore, "A fast majority vote algorithm," Tech. Report ICSCA-CMP-32, Institute for Computer Science, University of Texas, February 1981.
3. I. Calciu, S. Sen, M. Balakrishnan, and M. K. Aguilera, "How to implement any concurrent data structure," *Comm. of ACM*, vol. 61, no. 12, pp. 97–105, 2018.
4. N. M. M. K. Chowdhury, M. M. Akbar, and M. Kaykobad, "Disktrie: An efficient data structure using flash memory for mobile devices," In M. Kaykobad and M. S. Rahman, editors, *Proc. First Workshop on Algorithms and Computation (WALCOM)*, pp. 76–87, 2007.
5. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.
6. Cloudflare, Network latency, in *How to Make the Internet Faster for Everyone* [Online]. Accessed 24 August 2020. Available: https://www.cloudflare.com/learning/performance/more/speed-up-the-web/
7. E. Gal and S. Toledo, "Algorithms and data structures for flash memories," *ACM Computing Surveys*, vol. 37, no. 2, pp. 138–163, June 2005.
8. L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf, "Characterizing flash memory: Anomalies, observations, and applications," In *Proc. 42nd Ann. IEEE/ACM Int'l Symp. on Microarchitecture (MICRO 42)*, pp. 24–33, New York, NY, USA, 2009.
9. P.-H. Kamp, "You're doing it wrong," *Comm. of ACM*, vol. 53, no. 7, pp. 55–59, 2010.
10. D. E. Knuth, "Artistic programming," *Current Contents*, no. 34, August 1993. Available: http://garfield.library.upenn.edu/classics1993/A1993LQ46300001.pdf

11. D. E. Knuth, *The Art of Computer Programming, Volume 2 Seminumerical Algorithms*, page 144. third edition, 1998. Knuth attributes this to Alan G. Waterman.
12. C. L. Kuszmaul, binary heap, September 2013. in *Dictionary of Algorithms and Data Structures*, P. E. Black, ed. [Online]. Accessed 29 April 2020. Available: https://www.nist.gov/dads/HTML/binaryheap.html
13. D. Naor, C. U. Martel, and N. S. Matloff, "Performance of priority queue structures in a virtual memory environment," *The Computer Journal*, vol. 34, no. 5, pp. 428–437, January 1991.
14. G. V. Neville-Neil, "The data-structure canon," *Comm. of ACM*, vol. 53, no. 4, pp. 33–34, April 2010.
15. W. Pugh, "Concurrent maintenance of skip lists," April 1989. Tech. Report CS-TR-2222, Dept. of Computer Science, U. Maryland.
16. M. Skarupke, "On modern hardware the min-max heap beats a binary heap," August 2020. [Online]. Accessed 10 September 2020. Available: https://probablydance.com/2020/08/31/on-modern-hardware-the-min-max-heap-beats-a-binary-heap/
17. P. van Emde Boas, R. Kaas, and E. Zijlstra, "Design and implementation of an efficient priority queue," *Mathematical systems theory*, vol. 10, pp. 99–127, 1976.
18. U. Vishkin, "Using simple abstraction to reinvent computing for parallelism," *Comm. of ACM*, vol. 54, no. 1, pp. 75–85, January 2011.
19. C.-H. Wu, T.-W. Kuo, and L. P. Chang, "An efficient B-tree layer implementation for flash-memory storage systems," *ACM Trans. Embedded Computing Systems*, vol. 6, no. 3, art. 19, July 2007.

Paul E. Black is a computer scientist at NIST. He earned a Ph.D. at Brigham Young University. He has published in the areas of software assurance and testing, networks and queuing analysis, formal methods, software verification, and quantum computing. He is a senior member of IEEE and a member of ACM and the IEEE Computer Society. He can be contacted at paul.black@nist.gov.

David Flater is a computer scientist in the Software and Systems Division of NIST's Information Technology Laboratory. He earned a Ph.D. in Computer Science from the University of Maryland Graduate School, Baltimore. Software metrology is the current focus of his research. He can be contacted at david.flater@nist.gov.

Irena Bojanova is a computer scientist at NIST. She earned her Ph.D. in Mathematics/Computer Science from the Bulgarian Academy of Sciences. She is a senior member of IEEE. Currently, Irena serves as EIC of IT Professional and Chair of COMPSAC ITiP Symposium and STC conference. She can be contacted at irena.bojanova@nist.gov.