A Rotation of Data Structures*

Thomas Cook, Alexander Mentis, and Chris Okasaki†
Department of Electrical Engineering and Computer Science
United States Military Academy
West Point, NY 10996
{thomas.cook,alexander.mentis,chris.okasaki}@westpoint.edu

Abstract

This paper provides options for varying the details of certain topics from semester to semester in a data structures course that focuses—at least to some extent—on the implementation of data structures. The data structures considered are variations on balanced binary search trees and mergeable heaps (priority queues).

1 Introduction

Congratulations! You've just finished teaching Data Structures for the first time, and it went pretty well. As you begin to plan your second iteration, you have some ideas for how to teach certain topics better, but you're not sure whether to leave everything else exactly the same or to make changes even to parts of the course that seemed to go well. Here are some suggestions for some easy changes to make and why you might want to.

Curriculum design often operates at two levels of scale, the collection or sequence of topics that make up a course and the collection or sequence of courses that make up an academic major. But there is another scale less often discussed: planned variations across multiple semesters or years of the same course. We consider such variations for a Data Structures course.

Data Structures courses differ across a number of dimensions, such as programming language used, how much emphasis is placed on object-oriented pro-

*This paper is authored by an employee(s) of the United States Government and is in the public domain. Non-exclusive copying or redistribution is allowed, provided that the article citation is given and the authors and agency are clearly identified as its source.

†The views expressed in this article are those of the authors and do not reflect the official policy or position of the Department of the Army, DOD, or the U.S. Government.

gramming, and the balance between designing and/or implementing data structures versus using data structures provided by some language or library. Despite these differences, certain data structures are extremely commonly taught, including linked lists, hash tables, binary search trees, and heaps. We will focus on the latter two.

Both binary search trees and heaps offer a similar opportunity. After an introductory version for which there is very little significant variation (unbalanced binary search trees and the standard heap data structure[24]), it is common—at least in courses that don't focus mainly on the use of data structures from standard libraries—to progress to a more efficient successor (balanced binary search trees or heaps with an efficient merge). Pedagogically, these more efficient successors serve as useful vehicles for practicing working with non-trivial invariants as well as illustrating analysis techniques, especially those related to $O(\log n)$ running times. However, for both binary search trees and heaps, there are many possible choices for this successor that work essentially equally well, such as AVL trees[1] or red-black trees[10] (as successors for unbalanced binary search trees), or leftist heaps[7] or binomial heaps[23] (as successors for standard heaps).

We suggest varying the choice of successor with each offering of the course, moving systematically through a cycle of choices several years long. This can offer a number of advantages, including decreased opportunities for cheating and increased enrichment for the teaching staff (including teaching assistants, if any).

On the other hand, teachers assigned to a Data Structures course may not be experts in data structures and may not be familiar with a wide variety of alternatives to choose from. In this paper, we provide a curated selection of such choices and also discuss a number of "mix-ins" that can be sprinkled on each main choice.

2 Successors for Binary Search Trees

This section describes eight potential successors to unbalanced binary search trees. For each data structure, we describe the basic ideas—especially any invariants beyond the usual ordering invariants—and include citations where more complete details can be found.

For all of the data structures below, the search, insert, and delete operations run in $O(\log n)$ time, either in the worst case (for the first five) or in the expected case (for the last three).

For the first five alternatives (the ones with worst-case bounds), deletion is substantially more complicated than insertion and may not be worth the class time it would take to present. However, for the last three (the ones with

expected bounds), deletion is surprisingly easy. One tradeoff is that many students may not have enough background in probability to understand the analysis behind those expected bounds.
Here are the data structures:

- **AVL trees**[1][12, pp. 459–464]: AVL trees satisfy the invariant that, for every node, its left and right subtrees differ in height by at most one. When that difference becomes two as the result of an insertion or deletion, the invariant is restored with either a left or right single rotation or a left or right double rotation.

- **2-3 trees**[2]: 2-3 trees contain both 2-nodes (binary nodes with one key and two children), and 3-nodes (ternary nodes with two keys and three children). Because of the ternary nodes, these trees are not strictly *binary* search trees. A 2-3 tree is perfectly balanced in the sense that all paths from the root to a leaf are the same length. When an insert causes a leaf to be placed on a new level, balance is restored either by growing a 2-node into a 3-node or, when a 3-node would otherwise need to grow into a 4-node, by splitting the would-be 4-node into two 2-nodes with a 2-node parent. In the latter case, the changes continue to propagate up the tree.

- **Red-Black trees**[10][15][6, Chp. 13]: In a red-black tree, nodes are colored either red or black, with the invariants that (1) no red node has a red child, and (2) every path from the root to a leaf contains the same number of black nodes. When an insert causes a red node to have a red child, this can be fixed by a combination of rotations and re-coloring.

- **AA trees**[3, 20]: An AA tree is a variation of a red-black tree where a left child cannot be red. This limitation makes AA trees simpler to implement than regular red-black trees. An AA tree can also be viewed as a 2-3 tree where a 3-node is encoded as a black node with a red right child.

- **Weight-balanced trees**[8]: In a weight-balanced tree, no subtree can have a weight smaller than a fixed fraction of the weight of its parent tree, where the weight of a tree is its size + 1. (The +1 makes it easier to handle empty subtrees.) One advantage of weight-balanced trees is that the size information stored for balancing is also independently valuable to users of the data structure.

- **Treaps**[4]: A treap is binary search tree where every node has both a key and a randomly-chosen priority. The nodes of a treap are ordered by keys in the normal search tree ordering, and also ordered by priority in

the normal heap ordering (a child has a smaller priority than its parent, but may have a larger or smaller priority than its sibling). The benefit of treaps is that they are unusually easy to implement (especially delete!), but the tradeoff is that the running time of the standard operations is $O(\log n)$ expected time rather than $O(\log n)$ worst-case time.

- **Skip Lists**[17][13, Chp. 13]: A skip list is *not* a binary search tree, but serves a similar purpose. A skip list is similar to a linked list where every node has a pointer to its successor. However, a node in a skip list has multiple pointers to successors at different distances. These extra pointers allow an algorithm to *skip* ahead in the list. Like treaps, the running time of the standard operations in skip lists is $O(\log n)$ expected time rather than worst-case time. Deletions in skip lists are unusually straightforward.

- **Zip trees**[21]: A zip tree is a binary search tree that is isomorphic to a skip list, with algorithms that more closely resemble the operations on skip lists than the normal operations on search trees. In particular, zip trees replace the rotation operations used by most balanced binary search trees with operations for merging (zipping) and unmerging (unzipping) paths through a tree. Like treaps and skip lists, the running times of the standard operations are $O(\log n)$ expected time rather than worst-case time and deletions are unusually straightforward.

Important: The data structures above were ordered to emphasize commonalities between the various alternatives. This order should *not* be taken as a recommended order to use the data structures in successive offerings of a course. For use in a course, it is probably better to choose less related data structures in successive offerings.

2.1 Mix-ins

Once you have chosen a particular data structure, such as AVL trees, there are a number of smaller decisions that provide further sources of variation. We call these "mix-ins", because you can (mostly) mix any combination of these into the main data structure. Each choice will have small or large effects on the implementation.

- **Mutable or immutable**: Many courses will likely default to mutable implementations. However, all but one of these eight data structures can easily be made immutable using path copying[18] because path copying works very well on trees. The exception is a skip list, which is represented as a dag rather than a tree. But a zip tree can be viewed as the tree-version of a skip list and can easily be made immutable.

- **Header node**: One common use of a header node is to keep track of the current root of a mutable tree. That way, even if a particular operation makes a different node the new root, any part of the code that uses the header node can "see" the change. This use of header nodes typically does not apply to immutable trees because each different version of the tree has a different root. However, even immutable trees might use a header node to hold overall information about the tree, such as the size.

- **Parent pointers**: In a binary tree, a node typically includes pointers to its left and right children and may or may not also include a pointer to its parent. Parent pointers often allow operations like search, insert, and delete to be implemented with loops instead of recursion, with search using one downward loop and insert/delete using two loops: a downward loop followed by an upward loop. Note, however, that parent pointers are almost entirely incompatible with immutable data structures.

- **Key vs. key-and-value**: Almost any binary search tree can be adapted to represent either a set or a map. For a set, each node would store a key; for a map, each node would store a key together with an associated value.

- **Lazy delete**[22, p. 41]: A simple way to implement deletion in almost any binary search tree is to set a flag on a node whose element is being deleted, but to otherwise leave the node in the tree. Navigation through a flagged node goes left or right in the usual way, but if the key in the node matches the search key, the search is considered to fail. This scheme works very well as long as there are relatively few such deleted nodes in the tree, but degrades if there are too many deleted nodes. One way to handle this situation is to rebuild the tree whenever the fraction of deleted nodes exceeds some threshold, such as 50%. The rebuilding step takes $O(n)$ time to remove all the deleted nodes and usually restructures the remaining nodes to be perfectly balanced. Despite the fact that the rebuilding step takes $O(n)$ time, the delete operation still runs in $O(\log n)$ amortized time. (However, this analysis does not apply for immutable trees.)

- **Leaf trees**[12, p. 486][11]: All keys (and values, if any) are stored at the leaves of the tree. Interior nodes contain keys that are used for navigation only—a key stored in an interior node may or may not also appear in a leaf node. For example, inserting 3 into a singleton leaf tree containing 1 might result in a tree with 1 and 3 at the leaves and 1 at an interior node. A search method would go left at an interior node if the search key was less than or equal to the element at the node. An advantage of

this scheme is that deletion is greatly simplified because deletion always involves a leaf node.

3 Successors for Standard Heaps

This section describes five potential successors to the standard heap data structure for implementing priority queues. Unlike the standard heap data structure, each of these successors supports an efficient *merge* operation and is implemented with nodes and pointers.

All of these data structures share the same ordering invariant: an item in a parent node has higher or equal priority to items in its children, where higher priority means a larger value in a max-heap or a smaller value in a min-heap. Items in sibling nodes have no implied priority ordering with each other.

- **Leftist Heaps**[7][12, pp. 149–150]: A leftist heap is a heap-ordered binary tree where every left subtree is larger than its corresponding right subtree, either in height[7, 12] or in size (number of nodes)[5]. The height/size is stored in each node, and whenever a right sibling becomes larger than its left sibling, the two subtrees are swapped. Two leftist heaps are merged by merging the rightmost paths of the two trees (as if they were sorted lists) and swapping the children of any node whose right subtree becomes larger than the left subtree as a result of this merge. The `min` (or `max`) operation runs in $O(1)$ worst-case time, and the `insert`, `merge`, and `deleteMin` (or `deleteMax`) operations run in $O(\log n)$ worst-case time.

- **Maxiphobic Heaps**[14]: A maxiphobic heap is a heap-ordered binary tree where every subtree stores its height or size. When two heaps are merged, the root with the higher priority becomes the new root. Of the three remaining trees (the left and right subtrees of that root plus the other tree being merged), the two smaller trees are merged and become one child of the new root with the larger tree becoming the other child of the new root. The running times of the major operations are the same as for leftists heaps.

- **Skew Heaps**[19][13, Chp. 6]: A skew heap is again a heap-ordered binary tree, but, unlike leftist or maxiphobic heaps, a node in a skew heap does not store its height or size. The merge operation is similar to the merge operation of leftist heaps except that the left and right children of *every* node in the two rightmost paths are swapped during the merge instead of only swapping when the right child becomes larger than the left. The running times of the major operations are the same as for leftist heaps

except that the bounds for `insert`, `merge`, and `deleteMin`/`deleteMax` are amortized bounds rather than worst-case bounds.

- **Binomial Heaps**[23][13, Chp. 7]: A binomial heap (also known as a binomial queue) is a collection of binomial trees where each tree has a unique rank. A binomial tree of rank k is a node with k subtrees of ranks $k-1$ to 0. A binomial tree of rank k has size 2^k, creating a close correspondence between a binomial heap and the binary number representing the size of the heap. Insertion in a binomial heap is similar to incrementing a binary number and merging two binomial heaps is similar to adding two binary numbers. The running times of all the major operations are $O(\log n)$ worst-case time, but the `min` (or `max`) operation can easily be improved to $O(1)$ worst-case time. For mutable binomial heaps, a closer analysis improves the running times of `insert` and `merge` to $O(1)$ amortized time.

- **Pairing Heaps**[9][13, Chp. 7]: Pairing heaps are fast and relatively easy to implement, but are quite difficult to analyze[16]. A pairing heap is implemented as a heap-ordered multi-way tree, often using the first-child/next-sibling representation to make it a binary tree. The `insert` and `merge` operations add the the tree with the lower-priority root as the new first child of the other tree's root. The `deleteMin` operation removes the root and merges its children in two passes: the first pass merges the children in pairs from front to back, and the second pass merges the results from the first pass from back to front.

3.1 Mix-ins

Some of the mix-ins for binary search trees still apply to the varieties of heap structures discussed here, sometimes with minor differences.

- **Mutable or immutable**: Leftist heaps, maxiphobic heaps, and binomial heaps can be made either mutable or immutable, but skew heaps and pairing heaps should be considered mutable-only (at least at the undergraduate level) because the amortized bounds of skew heaps and pairing heaps are problematic as immutable data structures.

- **Header nodes**: Because merge combines two different heaps, care must be taken at the end of a merge to reset one of the two header nodes to be empty (if header nodes are being used).

- **Parent pointers**: If either `delete` or `decreaseKey` (below) are to be supported, parent pointers will likely be necessary.

- **Key vs. key-and-value**: Either approach can be used with any of these heaps.
- **Lazy delete**: See **delete** below.
- **Leaf trees**: Not useful for heaps.

Heaps introduce several extra potential mix-ins:

- **Access to the minimum (or maximum) element**: Because a binomial heap involves multiple trees, it takes $O(\log n)$ time to find the `min` (or `max`). This can be reduced to $O(1)$ time by explicitly tracking the overall minimum (or maximum) element. (The other kinds of heaps above already support finding the `min`/`max` in $O(1)$ time.)
- **Delete**: Some heaps support an operation to delete an element that is not the `min`/`max`. However, because heaps do not support searching for an element, such an operation typically requires returning a *handle* to the new node whenever a new element is inserted. That handle then provides immediate access to the desired node. Working with these handles typically forces the implementation to be mutable and to support parent pointers. (If *lazy delete* is used, parent pointers might not be required.)
- **DecreaseKey**: Some heaps support an operation to increase the priority of an existing node. This is used with min-oriented heaps by Dijkstra's algorithm, where increasing the priority means decreasing the key. Like `delete`, supporting `decreaseKey` typically involves handles and forces the implementation to be mutable and to support parent pointers.

4 Conclusion

We have presented a curated selection of data structures for use in a data structures course as follow-ups to the unbalanced binary search trees and standard heaps that are studied in most such courses. Rotating between the various data structures described here provides variety across successive offerings of the course. This variety will largely be invisible to students within a single offering, but nonetheless provides at least two benefits across multiple offerings:

- Enrichment: Teaching staff—including TAs—can benefit from exposure to different data structures than what they learned when they were students in a data structures course.
- Reduction in Cheating: With eight variations on binary search trees and five variations on heaps, it can easily be multiple years between

repetitions of the same base data structure. Factor in mix-ins that have a significant effect on the code and it might be a decade between exact repetitions. The mix-ins also make it substantially harder to find exact code on the internet. Of course, there is no silver bullet; students can still cheat, but if they do, it is more likely to be from somebody else in the same semester, which is easier to detect compared to copying from an earlier semester or from code on the internet.

References

[1] Georgy Adelson-Velsky and Evgenii Landis. "An algorithm for the organization of information". In: *Proceedings of the USSR Academy of Sciences* 146 (1962), pp. 263–266.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] Arne Andersson. "Balanced Search Trees Made Simple". In: *Proceedings of Algorithms and Data Structures, Third Workshop, WADS '93*. 1993, pp. 60–71.

[4] Cecilia R. Aragon and Raimund G. Seidel. "Randomized search trees". In: *30th Annual Symposium on Foundations of Computer Science*. 1989, pp. 540–545.

[5] Seonghun Cho and Sartaj Sahni. "Weight-Biased Leftist Trees and Modified Skip Lists". In: *ACM Journal of Experimental Algorithmics* 3 (Sept. 1998), 2–es.

[6] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd ed. MIT Press, 2009.

[7] Clark Allan Crane. "Linear lists and priority queues as balanced binary trees". Technical Report STAN-CS-72-259. PhD thesis. Stanford University, 1972.

[8] J. Nievergelt and E. M. Reingold. "Binary Search Trees of Bounded Balance". In: *SIAM Journal on Computing* 2.1 (1973), pp. 33–43.

[9] Michael L. Fredman et al. "The pairing heap: A new form of self-adjusting heap". In: *Algorithmica* 1 (1986), pp. 111–129.

[10] Leo J. Guibas and Robert Sedgewick. "A dichromatic framework for balanced trees". In: *19th Annual Symposium on Foundations of Computer Science*. 1978, pp. 8–21.

[11] Anne Kaldewaij and Victor J. Dielissen. "Leaf Trees". In: *Science of Computer Programming* 26 (1–3 May 1996), pp. 149–165.

[12] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Second. Addison-Wesley, 1997.

[13] Dinesh P. Mehta and Sartaj Sahni, eds. *Handbook of Data Structures and Applications*. Chapman & Hall/CRC, 2005.

[14] Chris Okasaki. "Alternatives to Two Classic Data Structures". In: *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*. 2005, pp. 162–165.

[15] Chris Okasaki. "Red-black trees in a functional setting". In: *Journal of Functional Programming* 9.4 (1999), pp. 471–477.

[16] Seth Pettie. "Towards a Final Analysis of Pairing Heaps". In: *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005)*. IEEE Computer Society, 2005, pp. 174–183.

[17] William Pugh. "Skip Lists: A Probabilistic Alternative to Balanced Trees". In: *Communications of the ACM* 33.6 (1990), pp. 668–676.

[18] Neil Sarnak and Robert E. Tarjan. "Planar Point Location Using Persistent Search Trees". In: *Communications of the ACM* 29.7 (1986), pp. 669–679.

[19] Daniel Dominic Sleator and Robert Endre Tarjan. "Self-Adjusting Heaps". In: *SIAM Journal on Computing* 15.1 (1986), pp. 52–69.

[20] Porter Eugene Smith and James H. Graham. "A Simple Balanced Search Tree". In: *Proceedings of the 1993 ACM Conference on Computer Science*. 1993, pp. 461–465.

[21] Robert E. Tarjan, Caleb Levy, and Stephen Timmel. "Zip Trees". In: *ACM Transactions on Algorithms* 17.4 (2021).

[22] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[23] Jean Vuillemin. "A Data Structure for Manipulating Priority Queues". In: *Communications of the ACM* 21.4 (1978), pp. 309–315.

[24] John W. J. Williams. "Algorithm 232: Heapsort". In: *Communications of the ACM* 7.6 (1964), pp. 347–348.