11 Efficient Heaps

11.1 Mergeable Priority Queue

A *mergeable priority queue* is an abstract data type that stores a collection of items, each with a *prior*, and supports the following operations.

- INSERT(Q, x, p): Insert a new object x with priority p into priority queue Q. This operation can also be used to create a new priority queue containing just one key.
- FINDMIN(Q): Return the item with the smallest priority in priority queue Q.
- DELETEMIN(Q): Remove the item with the smallest priority in priority queue Q, and return the removed item.
- MERGE(Q, Q'): Replace the priority queues Q and Q' with a new heap containing all items in both Q and Q'.[1]
- DECREASEKEY(x, p): Change the priority of item x to a smaller value p. Here x is a pointer/reference/handle directly to item x inside the priority queue data structure, typically returned by INSERT.[2]

A data structure implementing this abstract data type is called a *mergeable heap*.

If we never had to use DELETEMIN, mergeable heaps would be completely trivial. Each "heap" just stores to maintain the single record (if any) with the smallest key. INSERTS and MERGES require only one comparison to decide which record to keep, so they take constant time. FINDMIN obviously takes constant time as well.

If we need DELETEMIN, but we don't care how long it takes, we can still implement mergeable heaps so that INSERTS, MERGES, and FINDMINS take constant time. We store the records in a circular doubly-linked list, and keep a pointer to the minimum key. Now deleting the minimum key takes $\Theta(n)$ time, since we have to scan the linked list to find the new smallest key.

Without the MERGE operation, we can support all other operations in at most $O(\log n)$ time using a standard binary heap or even a standard balanced binary search tree (like scapegoat trees, splay trees, or treaps). In fact, binary heaps support FINDMIN and INSERT in constant time, and balanced binary search trees can be modified to support FINDMIN in constant time. However, without considerably more work,[3] both of these standard priority queues require $\Omega(n)$ time to MERGE.

In the rest of this lecture note, I'll describe several mergeable heap data structures that are more efficient than binary heaps or balanced binary search trees. These are summarized in the following table.

[1]Some sources call this operation MELD and refer to this data type as a *meldable* priority queue.

[2]Yes, this operation should be called DECREASEPRIORITY. Historically, priorities are also called "keys", despite the fact that they generally cannot be used to find items in a priority queue.

[3]One exception is Iacono and Özkan's *mergeable dictionary* data structure, which supports INSERT, PRED, SPLIT, and MERGE in $O(\log n)$ amortized time each.

	Insert	ExtractMin	Merge	DecreaseKey	Refs
leftist heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	[?]
binomial heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	[?]
Fibonacci heap*	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	[?]
quake heap*	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	[?]
pairing heap*	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	[?]
	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	[?, ?, ?]
	$O(2^{2\sqrt{\log\log n}})$	$O(\log n)$	$O(2^{2\sqrt{\log\log n}})$	$O(2^{2\sqrt{\log\log n}})$	[?]

Figure 1. Performance of some mergeable heaps. Time bounds for starred* heaps are amortized.

11.2 Leftist Heaps

Let T be an arbitrary binary tree. For any node v in T, we store an integer $v.mind$ defined recursively as follows:

$$v.mind = \begin{cases} 0 & \text{if } v.left = \textsc{Null} \text{ or } v.right = \textsc{Null} \\ 1 + \min\{v.left.mind, v.right.mind\} & \text{otherwise} \end{cases}$$

$v.mind$ is the *min*imum *d*istance from v to a descendant of v that does not have two children, or equivalently, the depth of the largest perfect binary subtree rooted at v. Abusing notation, we define $v.mind = -1$ when $v = \textsc{Null}$. Finally, we call T a ***leftist tree*** if $v.left.mind \geq v.right.mind$ for every node v. Every rooted subtree of a leftist tree is also a leftist tree.

Lemma 1. *Let T be a leftist tree with n vertices and root v.*
(a) The right spine of T has length $v.mind$.
(b) $v.mind \leq \log_2 n$.

A ***leftist heap*** is a leftist tree where every node v stores a number $v.prior$, and these priorities satisfy the *heap* property: If node v is the parent of node w, then $v.prior < w.prior$.

Merging two leftist heaps is relatively straightforward. Let u and v be the roots of two leftist heaps that we want to merge. Without loss of generality, we can assume that u has smaller priority than v; otherwise, swap the two variable names. First, we recursively merge the right subtree of u with v, and attach the result as the new right subtree of v. Next, if necessary, we swap the left and right subtrees of u to restore the leftist property. Finally, we return u as the root of the merged leftist heap.

```
Merge($u, v$):
  if $u = $ Null
      return $v$
  if $v = $ Null
      return $u$
  if $u.prior > v.prior$
      swap $u \longleftrightarrow v$
  $u.right \leftarrow$ Merge($u.right, v$)
  if $u.left.mind < u.right.mind$
      swap $u.left \longleftrightarrow u.right$
  $u.mind \leftarrow 1 + u.right.mind$
  return $u$
```

Because each recursive call moves either u or v to its right child, the leftist property implies that MERGE recurses at most $\log_2 u.size + \log_2 v.size = O(\log n)$ times before reaching a base case. Thus, MERGE runs in $O(\log n)$ time in the worst case.

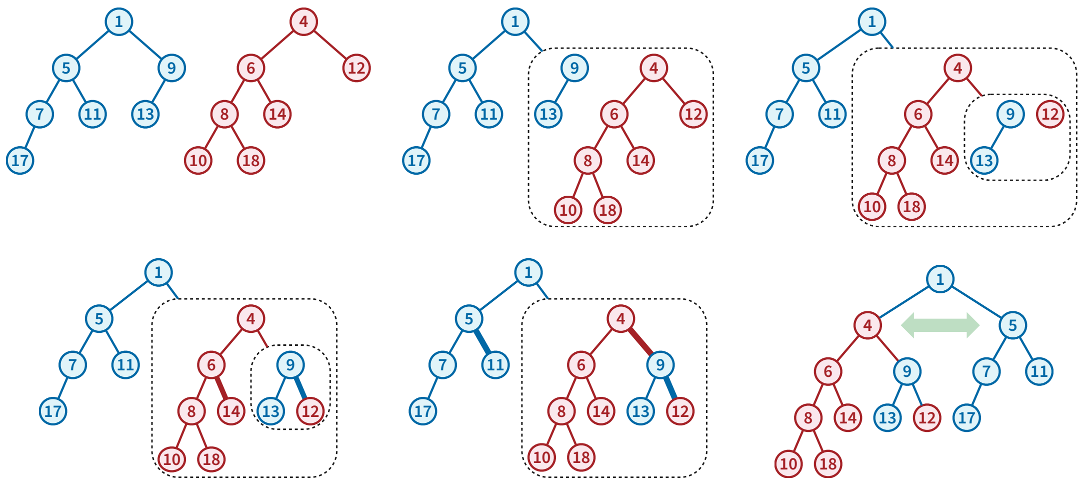

Figure 2. Merging two leftist heaps.

The remaining operations are all performed using MERGE as a subroutine:

- INSERT(H, x): Create a new one-node heap containing x and MERGE it with H.
- EXTRACTMIN(H): Let r be the root node of H. MERGE the left and right subtrees of H and return r.
- DECREASEKEY(x, p): Cut node x from its parent in H, set $x.prior \leftarrow p$, update *mind* values and swap children to make the rest of H leftist again, and then MERGE the rest of H with the subtree rooted at x.

11.3 Binomial Heaps

Now we consider a different heap structure that (at least superficially) looks very different from binary heaps of leftist heaps. Instead of a single binary tree, a *binomial heap* consists of one or more trees, each with a very specific recursive structure.

A kth-order ***binomial tree***, abbreviated B_k, is defined recursively as follows:

- B_0 is a single node.
- B_k is constructed from two copies of B_{k-1} by making the root of one copy a new child of the root of the other.

It is easy to prove by induction that B_k contains exactly 2^k nodes. Moreover, each node in a binomial tree B_k is the root of a binomial tree B_j for some index $j \leq k$. Finally, the order k of a binomial tree B_k is equal to both its depth and the degree of its root.

A single node in a binomial tree can have arbitrarily high degree, so we cannot store pointers directly from nodes to each of their children. Instead, each node v stores its priority $v.prior$, a pointer $v.child$ to its first child, and a pointer $v.next$ to its next sibling. Thus, the children of v are stored in a linked list, starting at $v.child$ and connected by *next* pointers. Each node in this list has priority larger than $v.prior$.

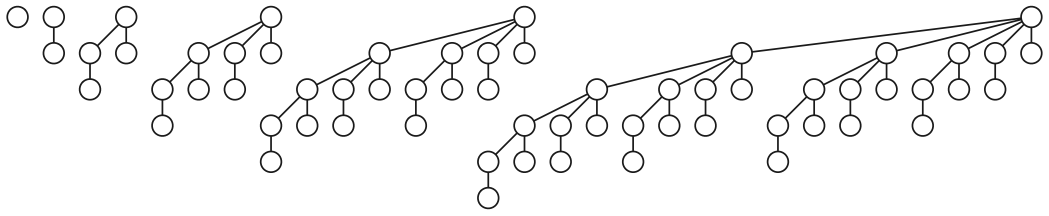

Figure 3. Binomial trees of order 0 through 6.

Thus, binomial trees are actually represented in memory as *binary* trees, where the root has no right child; this binary tree is sometimes called a *half-tree*. Moreover, if we sort the children of every node in every binomial tree by decreasing order, the half-tree representation of B_k consists of a single node, whose left subtree is a *perfect* binary tree with depth k and whose right subtree is empty.

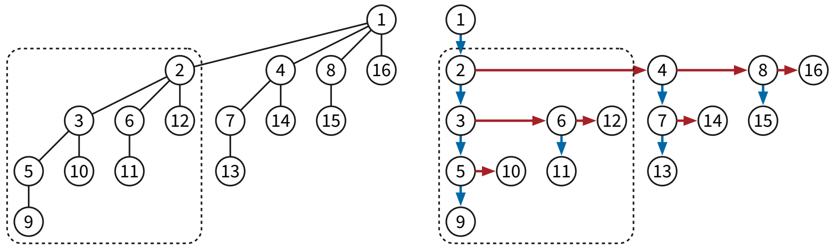

Figure 4. A 5th order binomial tree and its corresponding half-tree.

A ***binomial heap*** is a collection of heap-ordered binomial trees, with at most one binomial tree of each order. If the binomial heap contains n items, it contains a kth order binomial tree if and only if the kth bit of the binary representation of n is equal to 1. This connection to binary numbers is important for implementing the various algorithms. We index the trees using an auxiliary array $B[0..\lfloor\lg n\rfloor]$, where each entry $B[k]$ is either a pointer to the unique kth order binomial tree or NULL if there is no such tree. Finally, we maintain an explicit pointer *min* to the node with smallest priority.

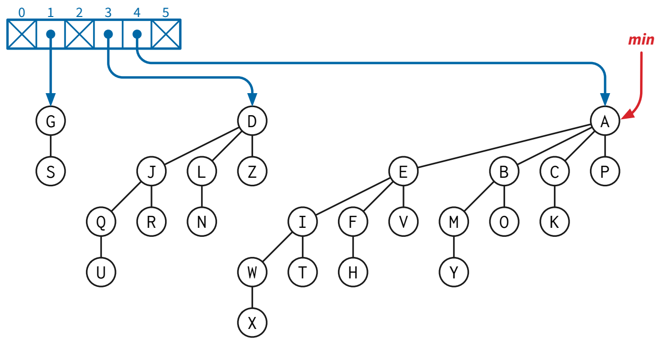

Figure 5. A binomial heap for the English alphabet, constructed by inserting the sequence PACKMYBOXWITHFVEDZNLQURJGS.

- INSERT mirrors the algorithm for incrementing a binary number. We start by creating a new 0th order binomial tree containing the new item. Then we repeatedly *link* pairs of binomial trees with the same order—making one root a new child of the other—until all the binomial trees have distinct orders. The following pseudocode shows the INSERT algorithm in complete gory detail. Because a binomial heap of size n consists of at most $\lg n$ binomial trees, INSERT runs in $O(\log n)$ time.

```
INSERT($p$):
  $newrt \leftarrow$ new node
  $newrt.prior \leftarrow p$
  $newrt.child \leftarrow$ NULL
  $newrt.next \leftarrow$ NULL
  $order \leftarrow 0$
  ⟨⟨Repeatedly link trees with the same order⟩⟩
  while $B[order] \neq$ NULL
      $oldrt \leftarrow B[order]$
      $B[order] \leftarrow$ NULL
      $order \leftarrow order + 1$
      ⟨⟨Link old and new roots⟩⟩
      if $oldrt.prior < newrt.prior$
          swap $oldrt \longleftrightarrow newrt$
      $oldrt.next \leftarrow newrt.child$
      $newrt.child \leftarrow oldrt$
  $B[order] \leftarrow newrt$
  if $B[min].prior > newrt.prior$
      $min \leftarrow newrt$
```

- MERGE similarly mirrors the algorithm for adding two binary numbers. For each order k from 0 up to $\lg(n+m)$, if there is more than one binomial tree of order k, we link two of them into a tree of order $k+1$. Again, the worst-case running time is $O(\log n)$.

- EXTRACTMIN(H) removes the minimum-priority node *min* from H, transforms the children of *min* into a new binomial heap H', and then MERGES the remainder of H with H'. Again, the worst-case running time is $O(\log n)$. See Figure **??** on the next page.

- Finally, DECREASEKEY(x, p) does not change the structure of the binomial trees at all. Instead, we change the priority of x to p, and then we repeatedly swap x with its parent until the heap property is restored. Implementing this operation efficiently requires each node in the binomial heap to store pointers to its parent and to its previous sibling, so that the children of any node are stored in a *doubly*-linked list. (The INSERT and MERGE algorithms must be updated to maintain these pointers.) Because the depth of each tree is at most $O(\log n)$, the worst-case running time of this algorithm is also $O(\log n)$.

11.4 Lazy Binomial ("Fibonacci") Heaps

So far we have only considered priority queues that have the same performance as binary heaps: $O(\log n)$ time per operation. It's not hard to prove, using a reduction from sorting, that *either* INSERT *or* EXTRACTMIN must take $\Omega(\log n)$ time in the worst case,[4] but nothing prevents us from making one of those two operations faster. Indeed, there are now dozens of priority queue structures that support INSERT in only $\boldsymbol{O(1)}$ time.

[4]. . . assuming all branches in those algorithms are based on simple comparisons

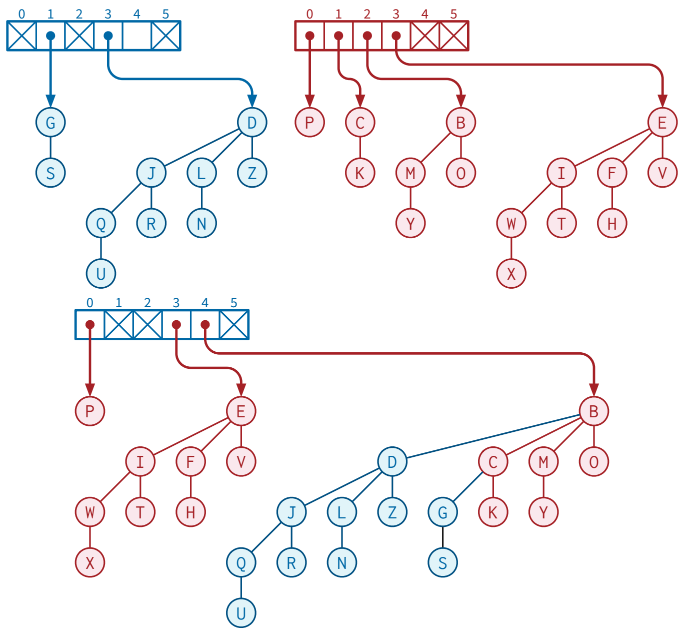

Figure 6. Running ExtractMin on the binomial heap in Figure **??**. Top: After extracting the children of the extracted minimum node into a new binomial heap. Bottom: After Mergeing the two binomial heaps.

The first such data structure is the ***Fibonacci heap***, proposed in 1987 by Michael Fredman and Robert Tarjan. Fibonacci heaps are variants of binomial heaps that are lazy in two different ways. First, we only enforce the restriction that tree sizes are unique when we perform an ExtractMin; specifically, we link trees with the same size as we look for the new minimum element. Second, we use a different algorithm for DecreaseKey that can change the shape of the tree; thus, our structure no longer consists of *binomial* trees, but of more general trees whose shapes and sizes are *close enough* to binomial trees.

A Fibonacci heap is a list of heap-ordered trees, which *for now* you should think of as binomial trees. In the simplest implementation, each node v in a Fibonacci heap maintains *four* pointers:

- $v.child$: The first child of v
- $v.parent$: The parent of v
- $v.next$: The root of the next tree if v is a root, or the next sibling of v otherwise
- $v.prev$: The root of the previous tree if v is a root, or the previous sibling of v otherwise

Thus, the top-level data structure is a *doubly*-linked list of roots, every node stores a pointer to a *doubly*-linked list of its children, and each child has a pointer back to its parent.

Maintaining all these pointers means that Fibonacci heaps are often slower than binary heaps and other variants *in practice*, even though they are faster *in theory*. Even for relatively large values of n, the asymptotic gains are overwhelmed by larger constant factors in the $O()$ notation. I'll describe the (usually) fastest priority queue *in practice* in the next section.

11.4.1 Lazy Insert and Merge

The Insert and Merge algorithms for Fibonacci heaps are identical to the same algorithms on linked lists. Insert adds a single node (that is, a heap-ordered order-0 binomial tree) to the linked list of roots, and MERGE concatenates the two given linked lists. Both operations clearly take $O(1)$ time in the worst case.

EXTRACTING the MINIMUM item is only slightly more complicated. First, we remove the minimum element from the root list and concatenate its list of children to the root list. Except for updating the parent pointers, this takes $O(1)$ time. Then we scan through the root list to find the new smallest key and update the parent pointers of the new roots. This scan could take $\Theta(n)$ time in the worst case. To bring down the amortized deletion time, we apply a CLEANUP algorithm, which connects pairs of equal-size trees until all tree sizes are distinct.

CLEANUP maintains an auxiliary array $B[1..\lfloor\lg n\rfloor]$, where $B[i]$ is a pointer to some previously-visited binomial tree of order i, or NULL if there is no such tree. CLEANUP simultaneously resets the parent pointers of all the new roots and updates the pointer to the minimum key. I've split off the part of the algorithm that links binomial trees of the same order into a separate subroutine MERGEDUPES.

```
CLEANUP():
  min ← head of the root list
  for i ← 0 to ⌊lg n⌋
      B[i] ← NULL
  for all nodes v in the root list
      v.parent ← NULL      (⋆)
      if min.prior > v.prior
          min ← v
      MERGEDUPES(v)
```

```
MERGEDUPES(v):
  w ← B[deg(v)]
  while w ≠ NULL
      B[deg(v)] ← NULL
      if w.prior > v.prior
          swap v ⟷ w
      remove w from the root list     (⋆⋆)
      w.next ← v.child
      v.child ← w        ⟨⟨"Link w to v"⟩⟩
      w ← B[deg(v)]
  B[deg(v)] ← v
```

During a single call to CLEANUP, the lines marked $(*)$ and $(**)$ are each executed at most once for each node in the root list. Thus, the running time of CLEANUP is $O(\log n + r')$, where r' is the length of the root list just before CLEANUP is called. It follows that DELETEMIN runs in time $O(r + \deg(min) + \log n)$, where r is the number of roots just before DELETEMIN is called, and min is the minimum-priority node being extracted. In a binomial heap with n nodes, every node has degree $O(\log n)$, so the worst-case running time of DELETEMIN is $O(r + \log n)$.

We can remove the r term from the *amortized* time by charging the $O(1)$ time spent on each root to the INSERT operation that added that root to the root list. Thus INSERT and MERGE still run in $O(1)$ amortized time, but now EXTRACTMIN runs in $O(\log n)$ amortized time.

11.4.2 Lazy DECREASEKEY

11.5 Pairing Heaps

A ***pairing heap*** is a single heap-ordered rooted tree, with no other required structure. Similar to binomial heaps, each node v stores a priority $v.prior$, a pointer to its first child $v.child$, and a pointer to its next sibling $v.next$. (Just like binomial trees, the actual data structure can be interpreted as a binary tree whose root has only one child.)

All operations in pairing heaps are based on the following extremely simple MERGE algorithm; the root with larger priority becomes the new first child of the root with smaller priority. The MERGE algorithm trivially runs in $O(1)$ time in the worst case. This operation is also referred to as *linking*.

```
MERGE(u, v):
    if $u.prior > v.prior$
        swap $u \leftrightarrow v$
    $v.next \leftarrow u.child$
    $u.child \leftarrow v$
    return $u$
```

INSERTing a new item into a pairing heap H is equivalent to MERGEing a new singleton pairing heap with H. Each INSERT clearly takes $O(1)$ time in the worst case.

The DECREASEKEY algorithm has two cases. If the priority of the target item x can be changed without violating the heap property, we are done. Otherwise, after changing the priority of x, we unlink x from its parent, and then MERGE the subtree rooted at x with the remainder of the pairing heap. Each DECREASEKEY clearly takes $O(1)$ time in the worst case.

The minimum-priority item in any pairing heap is stored at the root. EXTRACTMIN treats the children of the root as a sequence of pairing heaps and MERGES into a single new pairing heap using the subroutine MERGELIST, which merges a list of pairing heaps into a single pairing heap.

```
EXTRACTMIN(v):
    return MERGELIST($v.child$)
```

```
MERGELIST(v):
    if $v$ = NULL or $v.next$ = NULL
        return $v$
    return MERGE(MERGE($v, v.next$), MERGELIST($v.next.next$))
```

In the worst case, EXTRACTMIN runs in $\Theta(n)$ time; consider a pairing heap consisting of one node with $n-1$ children. In the next section, however, we will describe an amortization scheme that reduces the *amortized* time for each pairing heap operation—including EXTRACTMIN—to only $O(\log n)$.

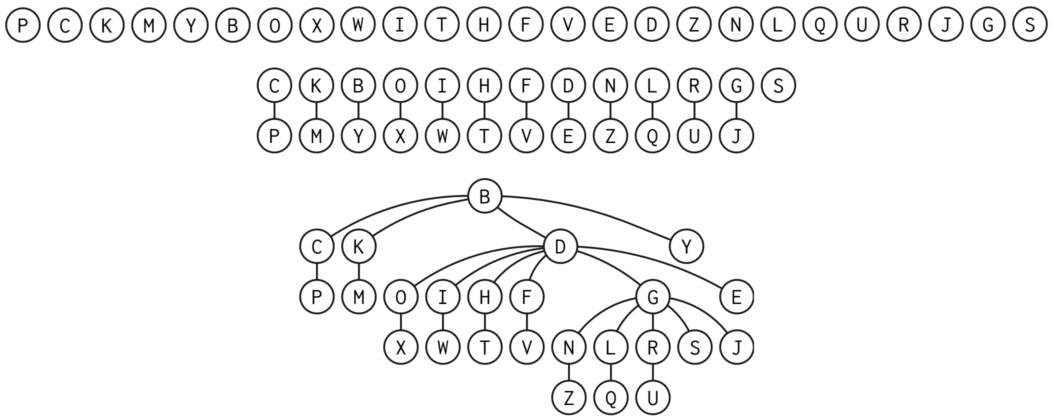

Figure 7. Running EXTRACTMIN on a pairing heap; letters indicate priorities in alphabetical order. The initial pairing heap, created by INSERTing the sequence PACKMYBOXWITHFVEDZNLQURJGS into an empty heap, has a root with priority A and 25 children. Top: After removing the root. Middle: After the pairing phase. Bottom: After the gathering phase.

11.6 Amortized Analysis of Pairing Heaps

The original analysis of EXTRACTMIN in pairing heaps by Fredman *et al.* [?] uses the binary half-tree view, which treats *child* and *next* pointers as *left* and *right* pointers in a binary tree.

MERGEing/linking two adjacent children of a common parent in the multiway-tree view closely resembles a *rotation* in the binary tree view. Specifically, linking a child to its successor looks exactly like rotation, and lining a child to its *predecessor* looks like a rotation followed by swapping two nodes and two subtrees; see Figures **??** and **??** below. This view allows us to leverage amortization arguments about rotations in binary search trees.

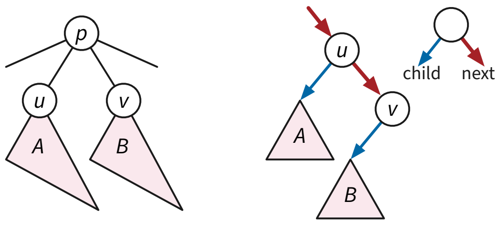

Figure 8. The binary tree view of two adjacent siblings u and v of a parent node p.

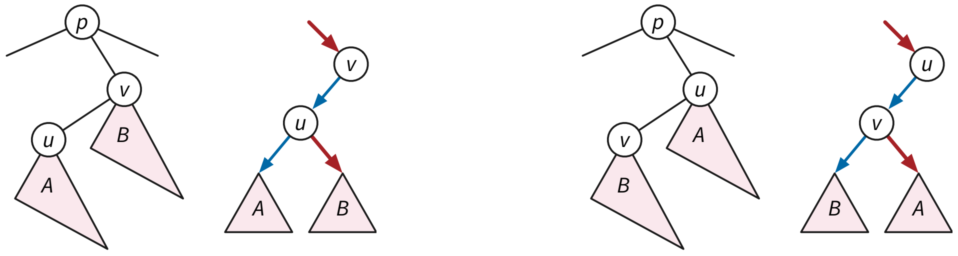

Figure 9. Linking looks like rotation in the binary-tree view. Left: linking u to v. Right: Linking v to u.

The specific order of MERGES in MERGELIST is crucial to the amortized analysis; different orders can lead to significantly worse amortized time bounds for EXTRACTMIN. We can split the execution of MERGELIST into two phases.

- In the *pairing* phase, we MERGE successive *pairs* of subtrees of v—first and second, third and fourth, and so on—in the order they appear in the next-sibling list (that is, in reverse order they were linked to v by earlier MERGES). If v has an odd number of children, the last one is left untouched in this phase. This part of the EXTRACTMIN algorithm gives pairing heaps their name. In the binary tree view, the pairing phase looks *exactly* like adding a new rightmost node, splaying that node to the root (using only roller-coasters), and deleting it; see Figure **??**. Almost exactly the same potential argument as play trees implies the amortized time for the pairing phase is $O(\log n)$.

- In the *gathering* phase, we repeatedly MERGE the *last* two heaps until only one heap remains. The number of links/rotations in this phase is at most the number of links/rotations in the pairing phase, so we can charge the gathering links/rotations to the pairing links/rotations. Moreover, Fredman *et al.* argue that this gathering phase changes the potential of the binary tree by at most $O(\log n)$, so the amortize time for the gatehring phase is also $O(\log n)$.

Later authors refined this amortized analysis to get tight*er* bounds, but ***a completely tight analysis is still an open problem.*** Iacono [?, ?] proved, using a different potential function, that INSERT and MERGE actually run in only $O(1)$ amortized time; their analysis was later simplified by Sinnamon* and Tarjan [?]. Pettie [?] found a different amortization scheme by

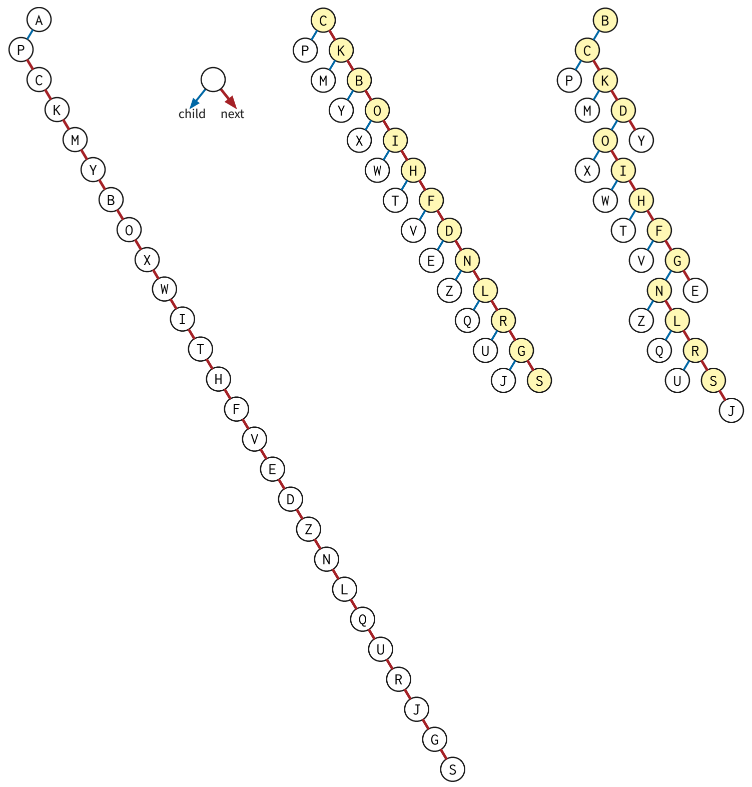

Figure 10. Binary tree view of the same ExtractMin as Figure **??**. The pairing phase compresses the right spine.

which ExtractMin runs in $O(\log n)$ amortized time and all other operations run in $O(2^{2\sqrt{\lg\lg n}})$ amortized time.

References

[1] Timothy M. Chan. Quake heaps: A simple alternative to Fibonacci heaps. *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, 27–32, 2013. Lecture Notes Comput. Sci. 8066, Springer.

[2] Clark Allan Crane*. *Linear Lists and Priority Queues as Balanced Binary Trees*. Outstanding Dissertations in the Computer Sciences. Garland Pub., 1980.

[3] Michael L. Fredman, Robert Sedgewick, Daniel Dominic Sleator, and Robert Endre Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica* 1(1):111–129, 1986.

[4] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM* 34(3):596–615, 1987.

[5] John Iacono*. Improved upper bounds for pairing heaps. *Proc. 7th Scand Workshop Algorithm Theory*, 32–45, 2000. Lecture Notes in Computer Science 1851, Springer.

[6] John Iacono. Improved upper bounds for pairing heaps. Preprint, October 2011. arXiv:1110.4428, Updated version of [**?**].

[7] Seth Pettie. Towards a final analysis of pairing heaps. *Proc. 46th Ann. IEEE Symp. Found. Comput. Sci.*, 174–183, 2005.

[8] Corwin Sinnamon* and Robert E. Tarjan. A nearly-tight analysis of multipass pairing heaps. *Proc. 34th Ann. ACM-SIAM Symp. Discrete Algorithms*, 535–548, 2023.

[9] Jean Vuillemin. A data structure for manipulating priority queues. *Commun. ACM* 21(4):309–315, 1978.

*Starred authors were graduate students at the time that the cited work was published.