ICS 621: Analysis of Algorithms Fall 2021

Lecture 2 — 10 September, 2021

Prof. Nodari Sitchinava *Scribe: In Woo Park, Darlene Agbayani, Michael Rogers*

1 Priority Queues

Previously, we covered the basics of amortized analysis using aggregate, accounting, and potential methods. We then proved binomial tree properties in the homework. In today's lecture, we discuss ***priority queues***.

Recall from undergraduate algorithms, that a priority queue (PQ) is an Abstract Data Type (ADT) that supports the following operations: ***create*** an empty priority queue, ***insert*** an element, ***find and/or remove*** the ***minimum*** element, change an element's priority by ***decreasing its key*** to a lower value, and ***delete*** an item from the priority queue.

Today we will consider one more operation, UNION(Q_1, Q_2), which ***combines*** two queues Q_1 and Q_2 and returns a new queue consisting of the all the items of Q_1 and Q_2. Our goal is to do this as efficiently as possible. For example, we can implement UNION(Q_1, Q_2) in any priority queue using the above operations by extracting the items from Q_1 and Q_2 and inserting them into the new queue one item at a time. Such implementation will take $O(n \log n)$ time in the binary heap, where n is the total number of items in the combined queue. Alternatively, if we recall that a binary heap is implemented as an array and that building a new binary heap from an array of elements in arbitrary order takes linear time, we can implement UNION in binary heaps in $O(n)$ time by copying all items from the two arrays of Q_1 and Q_2 into a new array and running the linear time algorithm for building a new binary heap.

Today we will consider another data structure that implements priority queue ADT, called ***binomial heap***. It implements UNION operation a lot faster than a binary heap. We will consider a regular binomial heap and a "lazy" binomial heap. See Table 1 for comparison between running times of the operations of the binary and binomial heaps.

2 Brief review of binary heaps

Recall that the tree structure of a binary heap data structure is implemented implicitly using arrays without explicit pointers by storing the nodes of the complete binary tree in the level order. Then we can find the left and right children of any node stored at index i of the array of the binary heap implementation at indices $2i$ and $2i + 1$. Also recall, the heap property requires that the key at every node be smaller than the keys of the two children of that node. Recursively, this means that any node's key is smaller than every key in the subtree rooted at that node.

	Binary	**Binomial**	**Lazy Binomial**
MAKE()	$O(1)$	$O(1)$	$O(1)$
INSERT(Q, x)	$O(\log n)$	$O(1)^*$	$O(1)$
MINIMUM(Q)	$O(1)$	$O(1)$	$O(1)$
EXTRACT-MIN(Q)	$O(\log n)$	$O(\log n)$	$O(\log n)^*$
DECREASE-KEY(Q, x, k)	$O(\log n)$	$O(\log n)$	$O(\log n)$
DELETE(Q, x)	$O(\log n)$	$O(\log n)$	$O(\log n)^*$
UNION(Q_1, Q_2)	$O(n)$	$O(\log n)$	$O(1)$

Table 1: Running times for operations of the three implementations of the priority queue. The running times marked with * are amortized.

3 Review of Binomial Trees

Let us review the properties of binomial trees. Recall that from the homework, a *binomial tree* B_k of order k is defined recursively as follows:

- A binomial tree B_0 of order 0 is a single node.
- For all $k > 0$, a binomial tree B_k of order k consists of two binomial trees of order $k-1$, with the root of one tree connected as a new *leftmost* child of the root of the other.

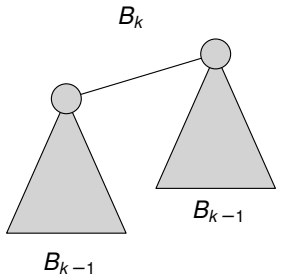

Figure 1: Construction of a binomial tree of order k from two binomial trees of order $k-1$.

As we have proven in the homework, the binomial tree B_k:

1. consists of 2^k nodes,
2. has height k,
3. has exactly $\binom{k}{i}$ nodes at depth i, for every depth, $i = 0, 1, \ldots, k$ in the tree, and
4. the root has degree k, which is greater than that of any other node; moreover, as Figure 2 below shows, if we number the children of the root from left to right by $k-1, k-2, \ldots, 2, 1, 0$, then the child i is the root of a subtree B_i.

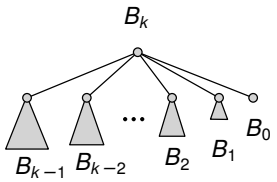

Figure 2: The children of the root of any binomial tree are the roots of binomial trees.

4 Binomial Heaps

A ***binomial heap*** is a collection of binomial trees. Each binomial tree is *heap-ordered*, i.e., at every node of the tree, the key of the parent is smaller than the key of any child, thus, the root node stores the smallest key among the ones stored in that tree. Every binomial heap on n items maintains the property that there is at most **one** binomial tree B_k for each $k = 1, 2, \ldots, \lfloor \log n \rfloor$, ordered by the sizes of the trees, i.e., there will not be two binomial trees of the same order in the heap. The heap also maintains a *head* pointer to the tree of the smallest order in the heap. To keep the the size of each nodes the same regardless of the number of children, the nodes of each binomial tree is stored in the *left-child, right-sibling* representation: every node v stores a pointer to the left-most child of v as well as to the right of v. To implement all priority queue operations efficiently, each node v will also store pointers to the left sibling of v and to the parent of v. Figure 3 illustrates an example of a binomial heap.

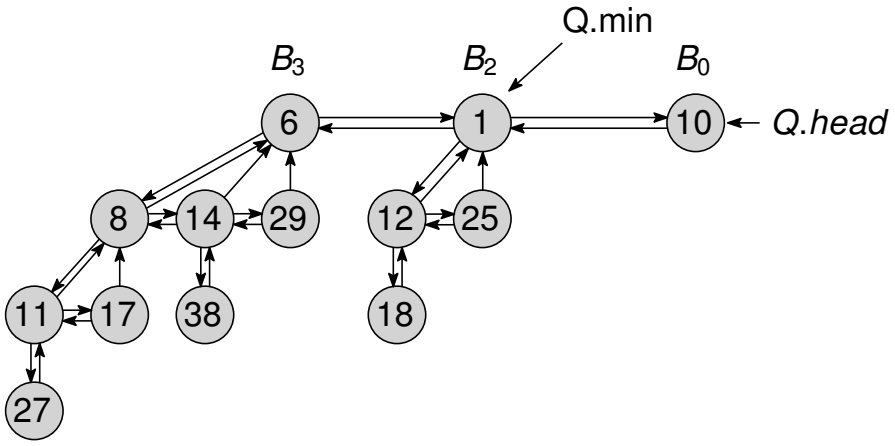

Figure 3: Binomial Heap with a pointer to the minimum element.

4.1 Minimum(Q)

The minimum node of the entire heap must be the root of one of the trees since every tree is heap-ordered. Since there are at most $\log n$ trees, finding the minimum is a $O(\log n)$ search. However, a

pointer to the minimum node, seen above as $Q.min$ (Figure 3), can be maintained, thereby reducing the cost of the operation to constant $O(1)$ time.

For Union or Insert, the minimum value pointer can be maintained in constant time by taking the minimum of the two input heaps. For Extract-Min and Delete, the minimum pointer can be maintained by searching for the new minimum value at a cost of $O(\log n)$. We can also update the pointer in constant time during the Decrease-Key operation, if needed. In all of these cases, maintaining the min pointer does not increase the asymptotic cost of the operation.

```
Minimum(Q)
1  return Q.min
```

4.2 Decrease-Key(Q, x, k)

Decrease-Key is implemented similar to its implementation in a binary heap. The key of the target node is decreased, then the node is repeatedly swapped with its parent until the heap order is restored. Since the height of every binomial tree in the heap is at most $\log n$, the cost of the operation is $O(\log n)$.

In order to achieve this time while maintaining the links between siblings, doubly-linked sibling lists are required, they allow us to swap parent and child nodes without losing sibling links. Without pointers to previous, we would be forced to iterate through the level at each swap until we found the left siblings of the nodes we are swapping, costing $O(\log n)$ at each swap.

```
Decrease-Key(Q, x, k)
 1  if k < x.key
 2      x.key = k
 3      while x.parent ≠ nil and x.key < x.parent.key
 4          x.parent.prev.next = x
 5          x.parent.next.prev = x
 6          x.prev.next = x.parent
 7          x.next.prev = x.parent
 8          swap(x.prev, x.parent.prev)
 9          swap(x.next, x.parent.next)
10          temp = x.parent.parent
11          x.parent.parent = x
12          x.parent = temp
```

4.3 Union(Q_1, Q_2)

To merge two binomial heaps, we will merge the constituent binomial trees, starting with the tree of the smallest order from each heap. We will maintain the invariant that there is at most one binomial tree of each order in the merged heap. One pointer should be pointing to the root of the current tree of the two input heaps, and one additional pointer should point to the root of the current tree of the growing output heap.

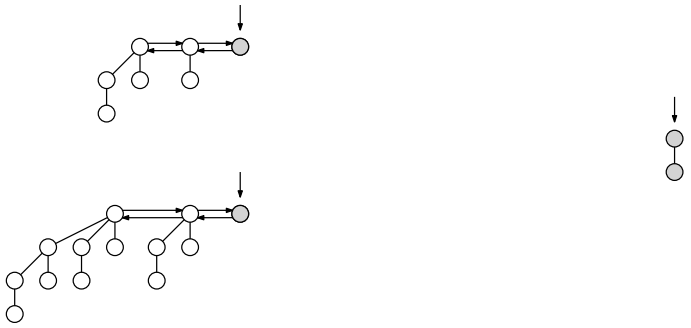

Figure 4: One step of the union procedure.

At each step, copy the tree of the smallest order from among the two input heap pointers, creating new sibling trees in the output heap as needed. If there are two trees of the same order, let's say of order k, combine them to create a tree of order $k+1$. When creating trees of larger order, maintain the heap order, i.e., make the tree whose root stores the larger key to be the child of the root of the tree whose root stores the smaller key.

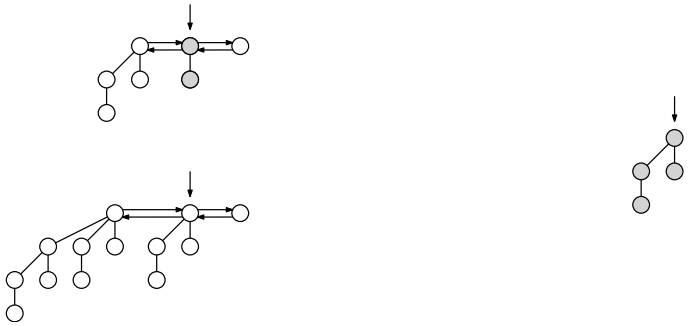

Figure 5: Use pointers to help determine when/where to merge heaps.

If all three pointers point to trees of the same size, merge the two from the input heaps as a new sibling tree in the output heap.

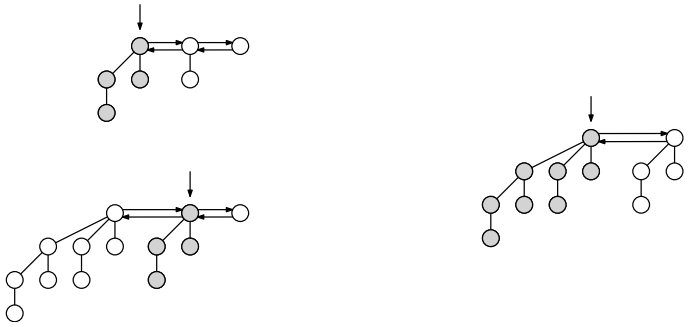

Figure 6: Link heaps accordingly until the process is complete.

Once all inputs are merged in, operation is complete. Since there are $O(\max k)$ trees to merge, where k is order of the trees with a maximum of $\log n$, the operation takes $O(\log n)$ time.

4.4 Insert(Q, x)

Inserting a node into a binomial heap is equivalent to merging two heaps, one with only a single node.

INSERT(Q, x)
1 $Q' =$ MAKE()
2 $Q'.head = x$
3 UNION(Q, Q')

In the worst case, this will take $O(\log n)$ time, the same as in UNION. We can use the same amortized analysis potential as we used in incrementing a binary counter to show that amortized cost of operation is $O(1)$. For every binomial heap structure there exists a binary representation that we can get by writing down indicators of presence of tree of order k, from the highest k in that heap to zero. A heap consisting of a single item is a binomial tree of order zero. Adding a tree of order zero thus will be equivalent to incrementing a binary counter that has a value equal to the binary representation of the existing heap.

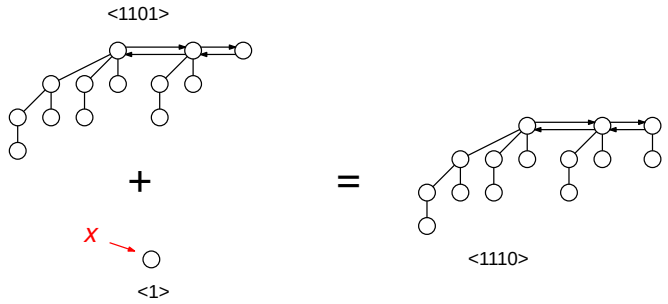

Figure 7: Binary representation of k binomial trees in the binomial heap.

The above shows the binary representation of each binomial heap. Note that INSERT, like UNION, creates a new binomial heap with a binary representation equal to the sum of the binary representations of the two heaps being merged.

4.5 Extract-Min(Q)

To extract the minimum node, x, of a binomial heap, first locate the minimum, which must be one of the tree roots. Then remove x from the linked list of roots by setting $x.prev.next$ to $x.next$ and $x.next.prev$ to $x.prev$. The children of x is a collection of heap-ordered binomial trees. Moreover, since the children are connected to each other using double-linked lists, they form a root list of a binomial heap themselves, let's call it Q'. To incorporate the children of x into the root list, we can use the UNION operation on the original heap with the tree rooted at x removed and the heap Q' formed by the children of x:

EXTRACT-MIN(Q)
1 x = MINIMUM(Q)
2 Q' = MAKE()
3 $Q'.head = x.leftchild$
4 LINKEDLIST-EXTRACT(x)
5 **for each** child y of x
6 $\quad y.parent = NIL$
7 Q = UNION(Q, Q')
8 **return** x

The total time to run EXTRACT-MIN(Q) is $O(\log n)$: lines 1-4, including the call to LINKEDLIST-EXTRACT(x) cost constant $O(1)$ time; the **for** loop on line 5-6, responsible for removing the parent pointers of x's children, takes time proportional to the number of those children, which is at most $O(\log n)$; and UNION(Q, Q') takes $O(\log n)$ time.

4.6 Delete(Q, x)

Deleting a node x, can be achieved with a single call to DECREASE-KEY($Q, x, -\infty$), followed by a single call to EXTRACT-MIN(Q), giving a total cost of $O(\log n)$ since it will be the sum of the costs of these two procedures.

DELETE(Q, x)
1 DECREASE-KEY($Q, x, -\infty$)
2 EXTRACT-MIN(Q)

5 Lazy Binomial Heaps

We can improve the $O(\log n)$ cost of the UNION operations to $O(1)$ by relaxing the requirement that there is at most one binomial tree of each order in the heap. Then we can delay the consolidation of trees until EXTRACT-MIN(Q) operation and implement UNION simply by attaching one heap to the other without merging the trees. As we will show in the homework, we can implement EXTRACT-MIN in amortized $O(\max\{k, \log n\}) = O(\log n)$ time.

However, for Lazy Binomial Heaps, although each binomial tree is still heap-ordered, there can be an arbitrary number of trees in the root list. Moreover, we must change the doubly-linked lists that connect the siblings to circular doubly-linked lists (see Figure 8 for an example).

The LAZY-UNION(Q_1, Q_2) is "lazy" because it delays the merging of trees of the same order into larger binomial trees and simply splices Q_1 into Q_2 to the node right of the minimum element in Q_2. It doesn't matter which Lazy Binomial Heap is spliced into which, and it doesn't matter where the heap is spliced into. The minimum pointer simply provides a quick ($O(1)$-time) way to find a location to splice the heaps.

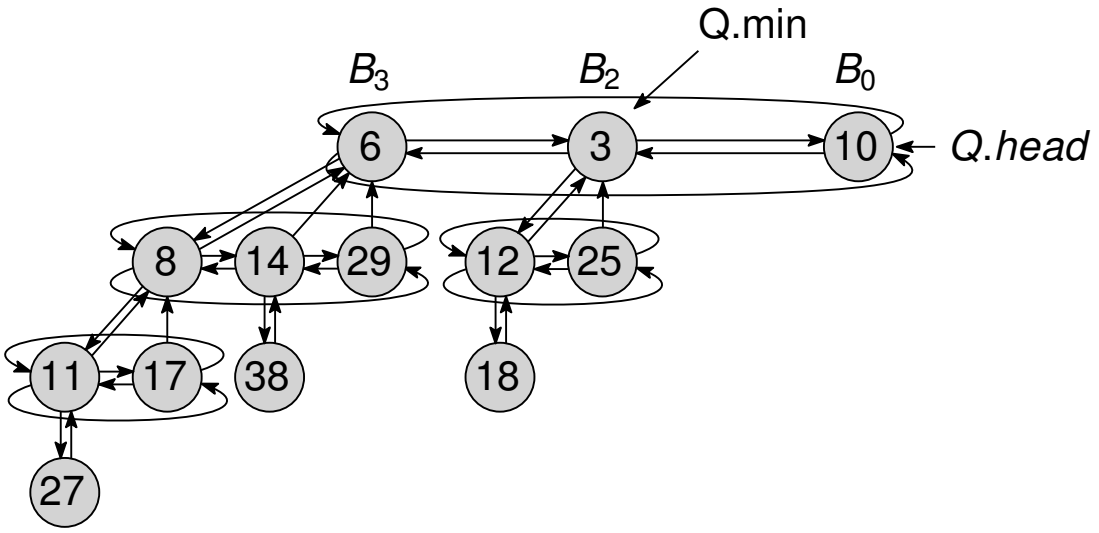

Figure 8: Heap-ordered Lazy Binomial Tree

LAZY-UNION(Q_1, Q_2)
1 $L_1 = Q_1.min.left$
2 $R_2 = Q_2.min.right$
3 $L_1.right = R_2$
4 $R_2.left = L_1$
5 $Q_2.min.right = Q_1.min$
6 $Q_1.min.left = Q_2.min$
7 **if** $Q_1.min.key < Q_2.min.key$
8 $\quad Q_2.min = Q_1.min$
9 **return** Q_2

Observe that after we splice Q_1 into Q_2 to the right of $Q_2.min$, LAZY-UNION(Q_1, Q_2) updates the *left* and *right* pointers to maintain the property that sibling lists are doubly-linked circular lists. However, unless we do something about it, the rule that Heap-ordered Binomial Trees have at most one tree B_k, for $k = 0, 1, 2, \ldots, \lfloor \log n \rfloor$ is not kept.

Notice that every operation in UNION(Q_1, Q_2) is constant. Therefore, the worst-case runtime of the algorithm is $O(1)$.

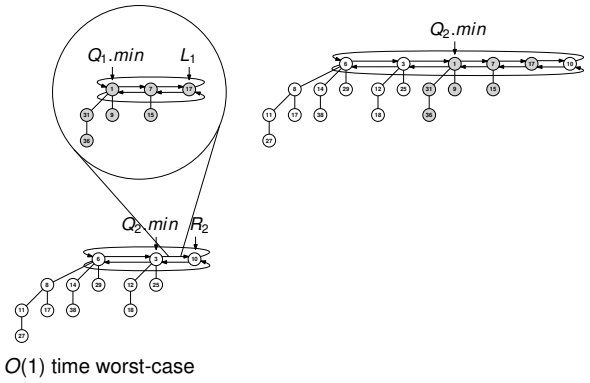

Figure 9: Lazy Union of Lazy Binomial Heaps

6 Summary

Binomial heap is a collection of binomial trees, each tree maintaining the property that every parent key is less than that of the child keys and that only one such binomial tree of order k can be in a binomial heap. Furthermore, the top layer in the binomial heap is made up of the minimum element in each of its respective binomial trees. We also explored, in depth, the operations used in mergeable priority queues using a standard binomial heap and analyzed the cost of each operation. In the homework, we will design an efficient algorithm, EXTRACT-MIN(Q) to make use of the reduced cost of the Lazy Binomial Heap LAZY-UNION(Q_1, Q_2) and LAZY-INSERT(Q, x) operations. We will also practice analyzing the amortized cost of the algorithm using the Potential Method.