ICS 621: Analysis of Algorithms Fall 2019

Lecture 2 — September 6, 2019

Prof. Nodari Sitchinava *Scribe: Anton Nikolaev and Jeremy Ong*

1 Overview

In the last lecture we covered basics of Amortized Analysis.

In this lecture, we discuss mergeable priority queues, priority queues which support Union operation - merging of two priority queues. We first examine the costs of operations using binary heap mergeable PQ. Then we explore a new mergeable PQ data structure, the binomial heap.

2 Priority Queues

PQ is an ADT (Abstract Data Type) that supports the following operations:

- Make() - create an empty priority queue.
- Insert(Q, x) - insert element x into existing priority queue Q.
- Minimum(Q) - return minimum of the priority queue Q.
- Extract-Min(Q) - extract minimum from the priority queue Q.
- Decrease-Key(Q, x, k) - in priority queue Q decrease key of the element x to k.
- Delete(Q, x) - remove element x from Q.
- Union(Q_1, Q_2) - merge Q_1 and Q_2 into one queue.

The **Binary Heap** data structure (taught in ICS 311) supports these operations with the following costs:

- Make() $O(1)$
- Insert(Q, x) $O(\log n)$
- Minimum(Q) $O(1)$
- Extract-Min(Q) $O(\log n)$
- Decrease-Key(Q, x, k) $O(\log n)$
- Delete(Q, x) $O(\log n)$
- Union(Q_1, Q_2) $O(n)$

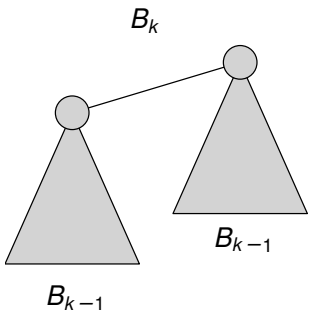
Using the **Binomial Heap** data structure, we can get these costs:

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

3 Binomial Trees

Remember that a binomial tree is defined recursively such that:

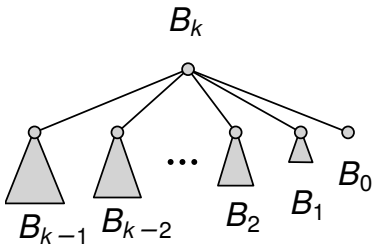
Definition 1. *B_0 is defined as a single node. Each binomial tree B_k of order k is comprised of two trees B_{k-1}, with one B_{k-1} attached as the leftmost child of the root of the other.*



As covered in last week's homework, B_k has the following properties:

1. Size equals to 2^k nodes.
2. Height equals to k.
3. It has $\binom{k}{i}$ nodes at depth i, for all existing depths in the tree.
4. Degree of the root is equal to k.

*Amortized cost



4 Binomial Heaps

A **binomial heap** is a collection of heap-ordered binomial trees.

- Each tree is *heap-ordered*: at every node, the parent is smaller than any child.
- At most one tree B_k for each $k = 1, 2, 3, \ldots, \log n$ is allowed.
- For any arbitrary $n \geq 0$ such heap can be constructed.

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 can be maintained, giving a search time of $O(1)$.

For Union or Insert, the min value pointer can be maintained by taking the min of the two input heaps, for a cost of $O(1)$. For Extract-Min and Delete, the min pointer can be maintained by searching for the new min value at a cost of $O(\log n)$. For Decrease-Key we can update the pointer in constant time, if needed. In all of these cases, maintaining the min pointer does not increase the asymptotic cost of the operation.

procedure Minimum(Q)
  return $Q.min$

Decrease-Key(Q, x, k)

Decrease-Key functions much as it does 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. As the height of every binomial tree in the heap is at most $\log n$, the cost of the operation is $O(\log n)$.

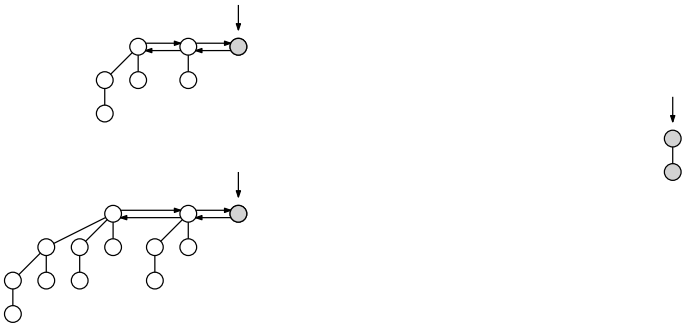
In order to achieve this time while maintaining the links between siblings, doubly-linked sibling lists are required, they allow us to swap parent and child nodes without losing sibling links. Without pointers to previous, we would be forced to iterate through the level at each swap until we found the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

procedure Decrease-Key(Q, x, k)
  $x.key \leftarrow k$
  if $x.parent \neq nil$ **then**
    while $x.key < x.parent.key$ **do**
      $x.parent.prev.next \leftarrow x$

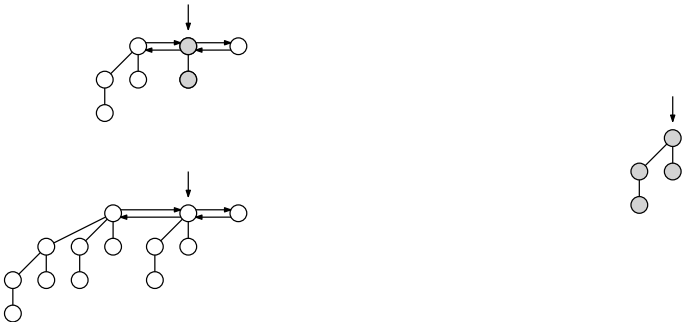
$x.parent.next.prev \leftarrow x$
$x.prev.next \leftarrow x.parent$
$x.next.prev \leftarrow x.parent$
$swap(x.prev, x.parent.prev)$
$swap(x.next, x.parent.next)$
$temp \leftarrow x.parent.parent$
$x.parent.parent \leftarrow x$
$x.parent \leftarrow temp$

Union(Q_1, Q_2)

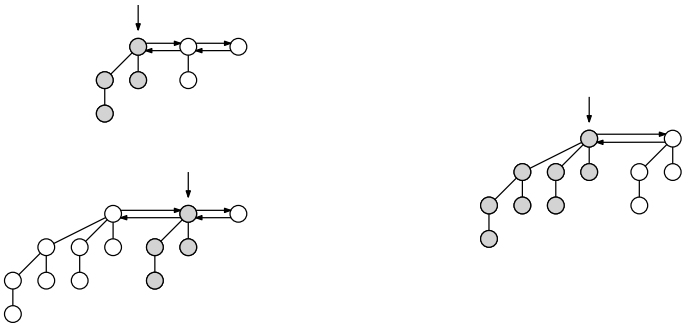
To merge two binomial heaps, merge the constituent binomial trees, starting with the smallest from each heap. Maintain heap order at each step. One pointer should be pointing to the root of the current tree of the two input heaps, and one additional pointer should point to the root of the current tree of the growing output heap.



At each step, merge in the smallest tree from among the two input heap pointers, creating new sibling trees in the output heap as needed.



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.



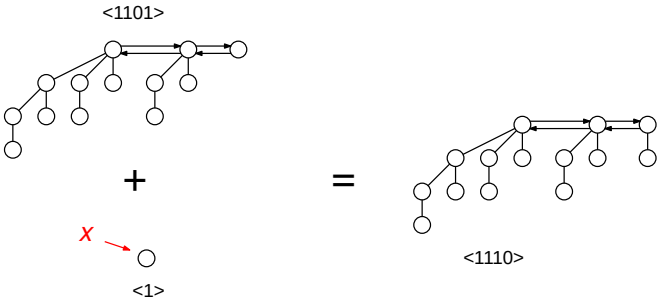
Once all inputs are merged in, operation is complete. Since there are $O(\max k)$ trees to merge, where k is order of the trees with a maximum of $\log n$, the operation takes $O(\log n)$ time.

Insert(Q, x)

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

procedure INSERT(Q, x)
  $Q' \leftarrow$ MAKE()
  $Q'.head \leftarrow x$
  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 heap architecture 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. Adding a tree of degree zero thus will be equivalent to incrementing a binary counter that has a value equal to the binary representation of the existing 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 sum of the binary representations of the two heaps being merged.

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 using LINKEDLIST-EXTRACT, which sets $x.prev.next$ to $x.next$ and $x.next.prev$ to $x.prev$. Set the leftmost child of x as the head of a new binomial heap Q'. Since the siblings of the leftmost child of x are all binomial trees of distinct

size connected as a doubly linked list and ordered by size, this is a valid binomial heap. Next, remove all pointers from the children of x to x. This leaves you with two binomial heaps. Finally merge the two heaps using UNION:

procedure EXTRACT-MIN(Q)
  $x \leftarrow$ MINIMUM(Q)
  $Q' \leftarrow$ MAKE()
  $Q'.head \leftarrow x.leftchild$
  LINKEDLIST-EXTRACT(x)
  for each child y of x **do**
    $y.parent \leftarrow NIL$
  $Q \leftarrow$ UNION(Q, Q')
  return x

Total time is $O(\log n)$, because LINKEDLIST-EXTRACT(x) and all operations before it take $O(1)$ time, updating 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.

Delete(Q, x)

Deleting a node x, can be achieved with a single call to DECREASE-KEY($Q, x, -\infty$) then 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.

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

5 Lazy Binomial Heaps

If $O(\log n)$ is too costly for the UNION operations, we can reduce the cost to $O(1)$ by simply attaching one heap to the other without merging the trees. This will need to be consolidated occasionally, but can be done during each call to EXTRACT-MIN(Q). This leaves us with an amortized cost of EXTRACT-MIN of $O(\log n)$, the same as its original cost.