Illinois Institute of Technology
Department of Computer Science

Lectures 15–16: October 20–25, 2010

CS 430 Introduction to Algorithms
Fall Semester, 2010

1 Binomial heaps

Consider the following operations on heaps:

- MAKE-HEAP: creates a new empty heap
- INSERT: inserts a new element into a heap
- MINIMUM: returns the minimum element in a heap
- EXTRACT-MIN: returns the minimum element in a heap and removes it from the heap
- UNION: creates a new heap consisting of the elements of two existing heaps
- DECREASE-KEY: changes the value of some element in a heap to a smaller value
- DELETE: removes an element from a heap

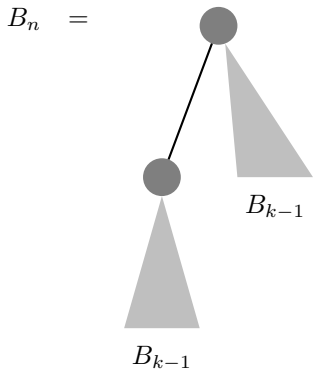
Recall the binary heap, as used in heapsort.[1] The binary heap can be viewed as a binary tree, although it is generally implemented as an array, where the children of element i are elements $2i$ and $2i + 1$. Most of these operations can be implemented efficiently ($\Theta(\log n)$ time or better) on a binary heap, but there is no efficient way to implement UNION: the best way is to concatenate the arrays containing the two heaps and re-heapify the resulting mess, requiring $\Theta(n)$ time. Our goal with binomial heaps is to support all of these operations in no worse than $\Theta(\log n)$.

1.1 Binomial trees

A binomial heap[2] is a collection of binomial trees. A binomial tree is an ordered tree defined recursively:

B_0 =

[1] See CLRS, chapter 6.
[2] Binomial heaps are the subject of Problem 19-2 on pages 527–529 in CLRS.



It follows directly from this definition that B_k has 2^k nodes, the height of B_k is k, there are $\binom{k}{i}$ nodes at depth i in B_k,[3] and the root of B_k has k children. As such, the height of B_k and the maximum degree of B_k are both logarithmic in the number of nodes, yielding a "bushy" tree.

1.2 Binomial heaps

A binomial heap is a collection of binomial trees in which each binomial tree is *heap-ordered*: each node is greater than or equal to its parent.[4] Also, at most one instance of B_i may occur for any i.

Since each B_i can occur at most once, we can make an analogy between binomial heaps and binary numbers, where B_0 is represented by the rightmost (least significant) bit, B_1 is represented by the second rightmost bit, and so on. Since the binary representation of an integer n has $\lfloor \lg n \rfloor + 1$ bits, a binomial heap of n nodes consists of at most $\lfloor \lg n \rfloor + 1$ binomial trees.

1.3 Operations

Operations on binomial heaps are not difficult:

- MAKE-HEAP: To create a new heap, we simply allocate the necessary space and return an empty heap in $O(1)$ time.
- MINIMUM: The minimum element must be the root of one of the trees in the heap. Since there are no more than $\lfloor \lg n \rfloor + 1$ trees, this consists simply of finding the minimum of at most $\lfloor \lg n \rfloor + 1$ elements, which can be done in $O(\log n)$ time.
- UNION: To join two binomial heaps, we begin by merging their collections of trees. Since there may now be two trees of the same degree, we perform an operation analogous to binary addition to combine trees as necessary. Since there are no more than $\lfloor \lg n \rfloor + 1$ pairs of trees to combine in this fashion, this can be done in $O(\log n)$ time.
- INSERT: Insertion is a special case of union: we create a singleton binomial heap containing the element we would like to insert and compute the union of this heap with the original heap. This clearly takes $O(\log n)$ time.

[3] Hence the term *binomial heap*.
[4] This same heap-ordering property holds in binary heaps.

- EXTRACT-MIN: Since the minimum element must be at the root of one of the subtrees, we first compare these elements. When we remove the root from its subtree, breaking it into a collection of smaller trees. We then union of these pieces with the other trees in the heap. This takes $\Theta(\log n)$ time.
- DECREASE-KEY: We begin by decreasing the key as necessary, and then we "bubble up" the node through the heap so the heap-ordering property holds. Since the height of a tree in the heap is at most $\lfloor \lg n \rfloor$, this takes $\Theta(\log n)$ time.
- DELETE: We can delete by applying DECREASE-KEY and EXTRACT-MIN: decrease the key to some value smaller than any other in the heap (CLRS uses $-\infty$, although any small enough value is sufficient), and then extract this element from the heap as the new minimum. This yields a time bound of $\Theta(\log n)$.

1.4 Comparison of time bounds

We can compare time bounds as follows:

Operation	Binary heap	Binomial heap
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\log n)$	$O(\log n)$
MINIMUM	$\Theta(1)$	$O(\log n)$
EXTRACT-MIN	$\Theta(\log n)$	$\Theta(\log n)$
UNION	$\Theta(n)$	$O(\log n)$
DECREASE-KEY	$\Theta(\log n)$	$\Theta(\log n)$
DELETE	$\Theta(\log n)$	$\Theta(\log n)$

2 Fibonacci heaps

Binomial heaps support the standard heap operations in logarithmic time. While logarithmic time may be sufficient for some applications, it may not be efficient enough for all purposes. In these lectures we develop the Fibonacci heap,[5] which supports heap operations that do not delete elements in constant amortized time. In essence, a Fibonacci heap is a "lazy" binomial heap in which the necessary housekeeping is delayed until the last possible moment: deletion.

Structurally, a Fibonacci heap is similar to a binomial heap: it consists of a collection of trees. Unlike with binomial heaps, we do not insist that the trees be binomial trees; we only require the heap property, that the key of a parent be smaller than the keys of its children. We add a number of items to the structure as well. Each node in the heap may be *marked*—as we will see later, a node is marked if one of its children has been removed. We also store the the *degree*, or number of children, of each node. As well, we maintain a pointer to the *minimum* node in the tree.

To analyze Fibonacci heaps, we use the potential method. The potential function we use is

$$\Phi(H) = t(H) + 2m(H)$$

where $t(H)$ is the number of trees in H and $m(H)$ is the number of marked nodes in H. As well, we define $D(n)$ to be an upper bound on the maximum degree of any node in an n-node Fibonacci heap.

[5] Fibonacci heaps are the subject of chapter 19 in CLRS.

2.1 Operations

In the analysis of these operations, we let H be the heap before the operation and H' be the heap after.

- MAKE-HEAP: As in binomial heaps, making a new heap consists of simply allocating the necessary space. As the potential of the empty heap is 0, the amortized cost of this operation is simply its actual cost, $O(1)$.
- INSERT: To insert a node into a heap, we simply add the new node to the tree list with no attempt to consolidate it with the existing trees. If the new node is smaller than the current minimum, we also update the min pointer.[6] The amortized cost, then, is

$$\begin{aligned} c_{am} &= c_{act} + \Phi(H') - \Phi(H) \\ &= O(1) + t(H') + 2m(H') - t(H) - 2m(H) \\ &= O(1) + (t(H) + 1) + 2m(H) - t(H) - 2m(H) \\ &= O(1) + 1 \\ &= O(1) \end{aligned}$$

- MINIMUM: As we maintain a pointer to the minimum element in the heap, we can simply follow this pointer. The potential does not change, so the amortized time of this operation is simply its actual time, $O(1)$.
- UNION: Two heaps are united by simply combining their tree lists, as before with no attempt to consolidate trees. The minimum of the resulting Fibonacci heap is the smaller of the minimums for the uncombined trees. The amortized cost is

$$\begin{aligned} c_{am} &= c_{act} + \Phi(H') - \Phi(H_1) - \Phi(H_2) \\ &= O(1) + t(H') + 2m(H') - t(H_1) - 2m(H_1) - t(H_2) - 2m(H_2) \\ &= O(1) \end{aligned}$$

- EXTRACT-MIN: When we extract the minimum node, we must perform the housekeeping we postponed until this point. Note that simply returning a pointer to the minimum node is trivial, as is physically removing it from the structure; however we must also update the pointer to the minimum node, and that is where the difficulty arises. (If we *never* perform the housekeeping and simply search among the trees for the new minimum, the resulting structure will be a mere linked list with a fancy name.)

  Analyzing the pseudocode on page 513 of CLRS, the actual cost of extracting the minimum node is $O(D(n) + t(H))$. By definition we have, for the original heap, $\Phi(H) = t(H) + 2m(H)$. Afterwards, the potential becomes $\Phi(H') = t(H') + 2m(H') = (D(n) - 1) + 2m(H)$.

  Thus we have

$$\begin{aligned} c_{am} &= c_{act} + \Phi(H') - \Phi(H) \\ &= O(D(n) + t(H)) + D(n) - 1 + 2m(H) - t(H) - 2m(H) \\ &= O(D(n)) + O(t(H)) - t(H) \end{aligned} \tag{1}$$

  Although we cannot claim that (1) is equivalent to $O(D(n))$, we can claim equivalence if the $t(H)$ term is multiplied by the constant hidden in the O notation. Fortunately, since we defined the potential

[6] This is simply a special case of the UNION operation, as we shall see.

function for the purposes of the analysis, we can scale it as necessary for this equivalence to hold. Thus we conclude that the amortized cost of EXTRACT-MIN is $O(D(n))$. We will later show that this is, indeed, a logarithmic bound.

- DECREASE-KEY: As in binomial heaps, when we decrease the key of a node, the modified node may violate the heap property. In spirit with the lazy approach, rather than repairing the situation on the spot, we can simply remove the subtree rooted at the decreased node from the tree it is in and place it at the top level. However, if we want to bound $D(n)$ (as we do, since we have seen that EXTRACT-MIN runs in $O(D(n))$ time), we need to restrain this activity. The way we do this is by indicating in a node, by marking it, if a child has been removed. If a node is already marked and we must remove a (second) child, the node itself is removed from its parent, and the process continues upward until an unmarked node is encountered.

  Analyzing the pseudocode on page 519 of CLRS, the actual cost involved is $O(c)$, where c is the number of recursive calls to the "cascading-cut" procedure. As always, the potential of the original heap is $\Phi(H) = t(H)+2m(H)$. The potential of the new heap is $\Phi(H') = t(H')+2m(H') = (t(H)+c)+2(m(H)-c+2)$. This gives us

$$\begin{aligned} c_{am} &= c_{act} + \Phi(H') - \Phi(H) \\ &= O(c) + (t(H) + c) + 2(m(H) - c + 2) - t(H) - 2m(H) \\ &= O(c) + 4 - c \\ &= O(c) - c \end{aligned} \tag{2}$$

  As before, we can scale up the units of potential as necessary to dominate the constant factor hidden in the $O(c)$ notation in (2), yielding an amortized bound of $O(1)$.

- DELETE: As in binomial heaps, this can be implemented simply by decreasing the key of the node to a value smaller than any other in the heap and then extracting the minimum. As DECREASE-KEY runs in $O(1)$ amortized time and EXTRACT-MIN runs in $O(D(n))$ amortized time, this operation runs in $O(D(n))$ amortized time.

2.2 Bounding $D(n)$

Lemma 19.1 states that the degrees of the first two children linked to some node x are at least 0, the degree of the third is at least 1, the degree of the fourth is at most 2, and so on. More formally, if $\mathit{degree}(x) = k$ and $y_1, y_2, \ldots, y_k$ denote the children of x in the order they were linked to x, then $\mathit{degree}(y_1) \geq 0$ and $\mathit{degree}(y_i) \geq i - 2$ for $i = 2, \ldots, k$. The proof is straightforward and can be found in CLRS.

Lemma 19.3 states that, if k is the degree of some node x, then $\mathit{size}(x) \geq F_{k+2}$, where $\mathit{size}(x)$ is the number of nodes in the subtree rooted at x.[7] The proof is by induction on k.

Since $F_{k+1} < \phi^k \leq F_{k+2}$, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803$, the maximum degree of any node in an n-node Fibonacci heap is $O(\log n)$. It follows, then, that the operations of EXTRACT-MIN and DELETE, which we previously showed to run in $O(D(n))$ amortized time, in fact run in $O(\log n)$ time.

[7] Recall that the kth Fibonacci number F_k is defined as

$$F_k = \begin{cases} k & \text{if } k = 0 \text{ or } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

2.3 Comparison of time bounds

We can update our earlier table as follows:

Operation	Binary heap	Binomial heap	Fibonacci heap
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
UNION	$\Theta(n)$	$O(\log n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
DELETE	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$

Note that the times indicated for the Fibonacci heap are amortized times while the times for binary and binomial heaps are worst-case per-operation times.