Mergeable Heaps

November 9, 2022

Abstract

These notes describe the *mergeable heap* abstract data type and cover its implementation via binomial heaps.

1 Mergeable Heaps

A *mergeable heap* is an abstract datatype that is a collection of keys (together with optional satellite data) drawn from a linearly ordered universe. As with all heaps, there are two versions: min-heap and max-heap. The min-heap version, which we will discuss exclusively, supports these basic operations (in no particular order):

Insert(x, H) — insert x into heap H

DeleteMin(H) — delete the item with minimum key from heap H

FindMin(H) — return the item with minimum key in heap H (the heap is unchanged)

DecreaseKey(H, x, k) — decrease the key of node x in H to k (assumes $k \leq x.key$; the node x and any satellite data is accessed externally, i.e., not through H; x's satellite data is unchanged)

Merge(H_1, H_2) — combine heaps H_1 and H_2 into a single heap H (H is returned; H_1 and H_2 are destroyed)

Delete(x, H) — remove item x from heap H (x is accessed externally)

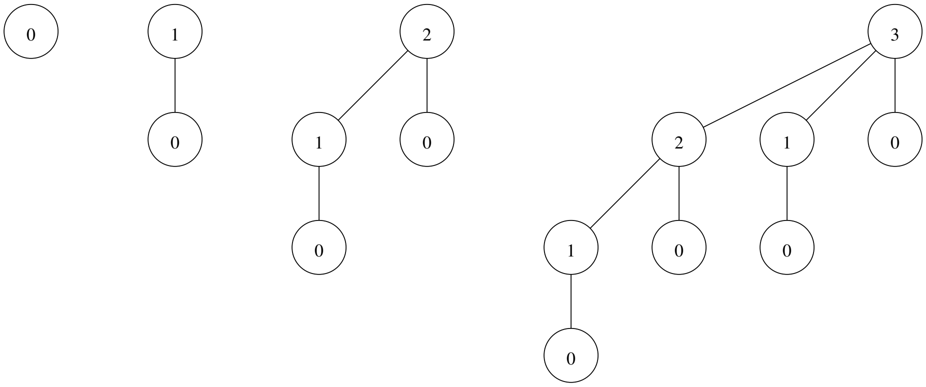
2 Binomial Trees

A *binomial tree* is a rooted, ordered tree given by the following recursive definition:

Definition 1. Let $d \geq 0$ be any integer.

- A *binomial tree of degree* 0 consists of a single node (the root).
- A *binomial tree of degree* $d + 1$ consists of two binomial trees T_1 and T_2 of degree d stuck together in such a way that the root of T_1 is the leftmost child of the root of T_2.

Here are the first four binomial trees, of degrees 0, 1, 2, and 3; the degree of each node is shown:



Basic facts about binomial trees:

1. The number of children of the root equals the degree.
2. Every node in a binomial tree is the root of a binomial (sub)tree.
3. A binomial tree of degree d has height exactly d and size exactly 2^d.
4. The degrees of the children of a degree-d node, from left to right, have degrees $d-1$, $d-2$, $\ldots$, 0.
5. The number of nodes on level i of a degree-d tree (where $0 \le i \le d$) is the binomial coefficient $\binom{d}{i} = \frac{d!}{i!(d-i)!}$. This explains the name, "binomial tree."

All of these facts are easily shown by induction on d except the last one, which is shown by induction on i using the "Pascal's triangle" recurrence: $\binom{d}{i} = \binom{d-1}{i-1} + \binom{d-1}{i}$ for $0 < i < d$ with boundary conditions $\binom{d}{0} = \binom{d}{d} = 1$.

3 Binomial Heaps

A **binomial heap** is a sequence of binomial trees of strictly increasing degree. Data (including keys) are in the tree nodes, each tree being in min-heap order. A node can be implemented as a record (struct) with five fields:

data — This includes the key and a reference to any satellite information.

degree — The degree of the node.

parent — A pointer to the parent node (NULL for the root).

leftmost_child — A pointer to the leftmost child of the node (NULL for a leaf node).

right_sibling — A pointer to the sibling node immediately to the right (NULL for the rightmost child of a parent).

A binomial heap H is a simple linked list of the roots of its trees, where the *right_sibling* pointer of each root points to the root of the next tree (the last tree having a NULL pointer). The attribute $H.trees$ points to the head (first root) of this list or is NULL for an empty heap. $H.min$ is an optional additional attribute that points to the node with minimum key in the heap (necessarily one of the roots, since each tree is in min-heap order). We will assume this attribute is included with H.

A binomial heap has a structure (i.e., arrangement of nodes without regard to the data they contain) uniquely determined by the number of items in the heap. Let's see why. Every natural number n is the unique sum of increasing powers of 2: the exponents correspond to the positions of the 1's in n's binary representation. Since a binomial tree of degree d has exactly 2^d many nodes, a binomial heap of n items must be made up of trees whose degrees are these exponents. For example, a binomial heap with 13 items is made up of trees with degrees 0, 2, and 3 in that order ($13 = 2^0 + 2^2 + 2^3 = 1101$ in binary).

It follows that a binomial heap with n items has $\leq 1 + \lg n$ many trees, each of degree $\leq \lg n$.

3.1 Min-heap operations on binomial heaps

Here are all min-heap operations for a binomial heap except for MERGE (some use MERGE or DECREASEKEY as a subroutine). Times given are worst-case times, assuming H has n items.

FindMin(H) — Return $H.min$. This takes $\Theta(1)$ time.

Insert(x, H) — Create a new heap H' with x as its sole element (a single tree of degree 0). Then set $H := \text{MERGE}(H, H')$. This takes $\Theta(\lg n)$ time.

DecreaseKey(H, x, k) — Change x's key to k, then "bubble up": While $k < x.parent.key$, swap x with its parent (just the data, not the degrees or the pointers, so the structure of the tree does not change). If $k < H.min$, then set $H.min := x$ (which must be a root by this point). This takes $\Theta(\lg n)$ time.

Delete(x, H) — Call DECREASEKEY$(H, x, -\infty)$ then DELETEMIN(H). This takes $\Theta(\lg n)$ time.

DeleteMin(H) — Unlink the tree root r pointed to by $H.min$ from the list of tree roots (which requires finding the predecessor root, if any); reverse the list of r's children so that the degrees are increasing, giving it the structure of a binomial heap H'; set $H := \text{MERGE}(H, H')$ and update $H.min$ if necessary. This takes $\Theta(\lg n)$ time.

The MERGE operation uses the subroutine MERGETREE for combining two binomial trees of the same degree. MERGETREE takes $\Theta(1)$ time.

```
MERGETREE($T_1$, $T_2$) // Precondition: $T_1$ and $T_2$ are binomial trees of the same degree $d$.
    if $T_1.key < T_2.key$:
        SWAP($T_1$, $T_2$) // Pointer swap; now $T_1.key \leq T_2.key$.
    Prepend root of $T_2$ onto the front of the list of $T_1$'s children // I had this backward in lecture.
    Increment $T_1.degree$
    Return $T_1$ // $T_1$ is a "carry tree" of degree $d + 1$.
```

The MERGE(H_1, H_2) operation on heaps H_1 and H_2 takes time $\Theta(\lg n)$, where n is the total number of items in H_1 and H_2 combined. It produces a heap H in three steps:

1. Merge the two linked lists $H_1.trees$ and $H_2.trees$ into a single linked list L in ascending order by degree. This step is just like the recombination phase of MERGESORT. If there are ties, add the tree from H_1 to L first then H_2, so that the tree from H_1 appears in L before the tree from H_2. (This is an arbitrary convention and does not affect run time or correctness.) L may contain duplicate degrees, and if so, they always appear consecutively.

2. In a loop, traverse L from front to rear by advancing a list pointer p, merging trees of equal degree and placing the results back into L as you go. This is accomplished as follows: Initially, p points to the first tree in L and advances through L until it becomes NULL. At any time, let T_1 be the tree that p currently points to, T_2 the tree immediately after T_1 on L (if it exists), and T_3 the tree immediately after T_2 on L (if it exists). Each iteration of the loop applies one of three cases:

   Case 1: If T_2 does not exist or if $T_1.degree < T_2.degree$, then there is nothing to combine; advance p.

   Case 2: If $T_1.degree == T_2.degree == T_3.degree$, then there is nothing to combine. Advance p as in Case 1. (T_1 must have been a "carry tree" resulting from a previous MERGETREE operation; this is the only way to have three trees of equal degree on L.)

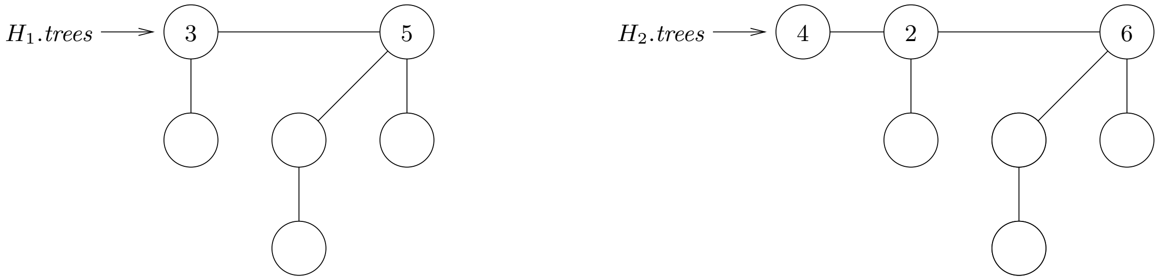
   Case 3: If $T_1.degree == T_2.degree$ and T_3 either does not exist or $T_3.degree > T_2.degree$, then replace T_1 and T_2 on L with MERGETREE(T_1, T_2) and leave p pointing to the combined tree (i.e., don't advance p before the next iteration of the loop).

3. Set $H.trees := L$, and set $H.min$ to either $H_1.min$ or $H_2.min$, whichever has the smaller key value. Return H.

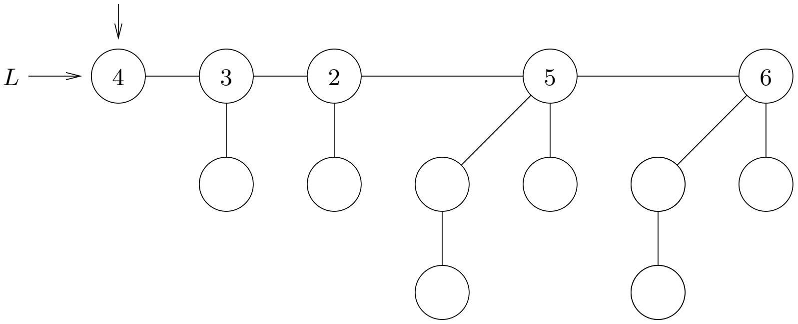
Three things to note: (1) the list L stays in ascending order by degree throughout the loop, and when the loop finishes, L consists of trees of strictly increasing degree; (2) at any time, there are at most three trees in L of the same degree; (3) no actual data is moved during the entire MERGE operation as only pointers change.

Example

Here is a sample MERGE operation. Let H_1 and H_2 be as below (.*min* pointers are omitted, as are non-root key values, which are of no consequence):

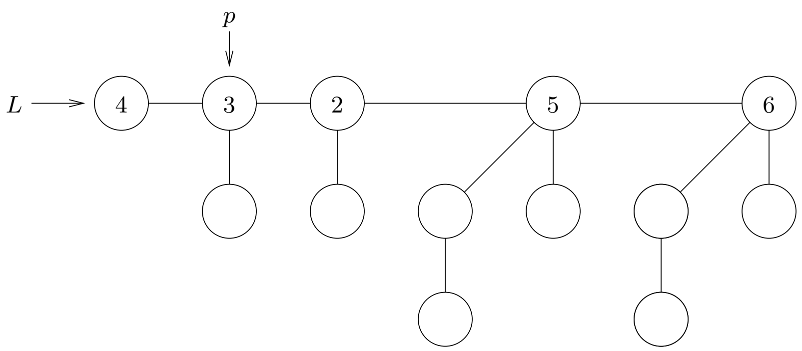


After Step 1 we have

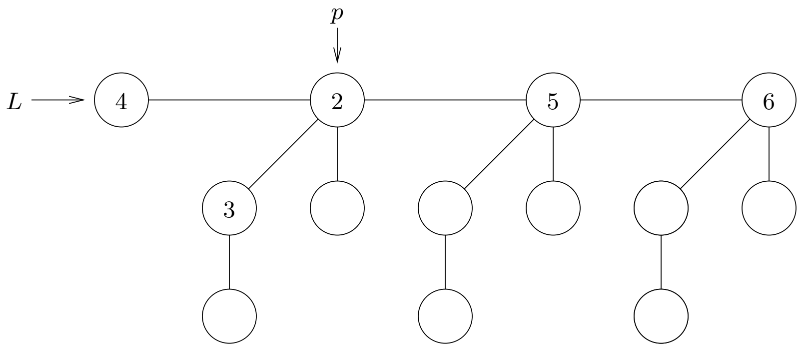


The loop in Step 2 iterates as follows:

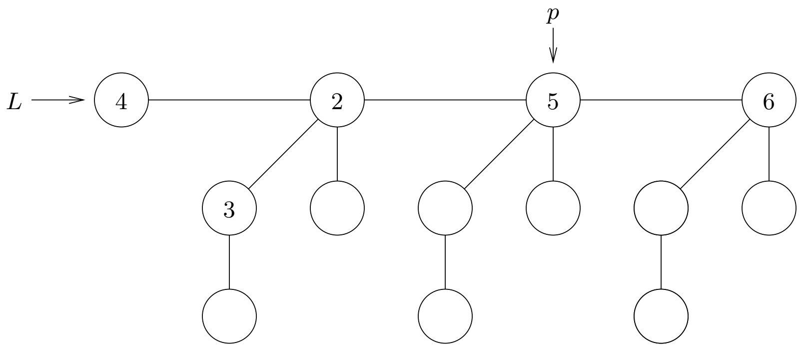
Case 1:



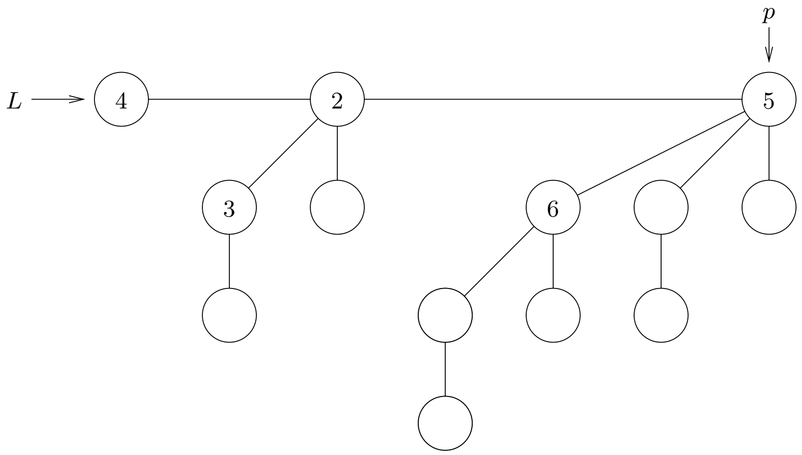
Case 3:



Case 2:



Case 3:



Step 3 sets $H.trees := L$. This is the combined heap.