Tutorial 05 - Examples, potential function method
CS 240E Winter 2023
University of Waterloo
Monday, February 13, 2023

1. **Nearest smaller elements.** Given an array of n integers, give an algorithm to find, for each element x, the first smaller element that appears to the left of x in the array (or conclude that no such element exists). Worst-case time and auxiliary space should be in $O(n)$.

   For example, for the array $[1, 3, 5, 2, 4, 15, 9]$, the output is,

   $$[\text{null}, 1, 3, 1, 2, 4, 4],$$

   since there are no elements smaller than 1 to the left of 1; for 4 the first smaller element to its left is 2.

2. **Fibonacci heaps.** The Fibonacci heap is data structure for priority queue operations. Several of its operations have better amortized running time those of binary and binomial heaps.

   The number of items in the heap(s) at the time of an operation is denoted by n. We also assume that our heaps are min-oriented, and that all keys are distinct.

	Binary heap (worst-case)	Fibonacci heap (amortized)
insert	$\Theta(\log n)$	$\Theta(1)$
delete-min	$\Theta(\log n)$	$O(\log n)$
merge	$\Theta(n)$	$\Theta(1)$
decrease-key	$\Theta(\log n)$	$\Theta(1)$

   Table 1: Runtimes for operations on two implementations of a heap.

   If we do not need the *merge* operation, ordinary binary heaps work well.

   From the theoretic analysis standpoint, Fibonacci heaps are especially useful when we have to *decrease-key* often. Some algorithms for graph problems call *decrease-key* once per edge: when a graph has many edges, the $\Theta(1)$ amortized time is a big improvement over the $\Theta(\log n)$ worst-case time.

   A Fibonacci heap is a collection of rooted trees with a min-heap ordering. Every node x knows its parent and children. The children of x are linked together in a circular, doubly linked list, which we refer to as the *child list* of x. Each child y in a child list has pointers $y.left$ and $y.right$ pointing to the y's left and right siblings respectively. If y is an only child, then $y.left = y.right = y$. We note that siblings may appear in any order in the child list.

   We also store the number of children of a node in $x.degree$ (not counting the parent pointer). Each node also contains a boolean attribute $x.mark$, which indicates whether x has lost a child since the last time x was made child of another node. Until we look at *decrease-key*, we set all *mark* attributes to *false*. This field is primarily used for the *decrease-key* operation.

We access a Fibonacci heap H by a pointer *H.min* to the *minimum node*: root of a tree containing a minimum key. If H is empty, *H.min* is *null*.

The roots are linked together using their *left* and *right* pointers into a circular, doubly linked list called the *root list*. Trees may appear in any order within a root list.

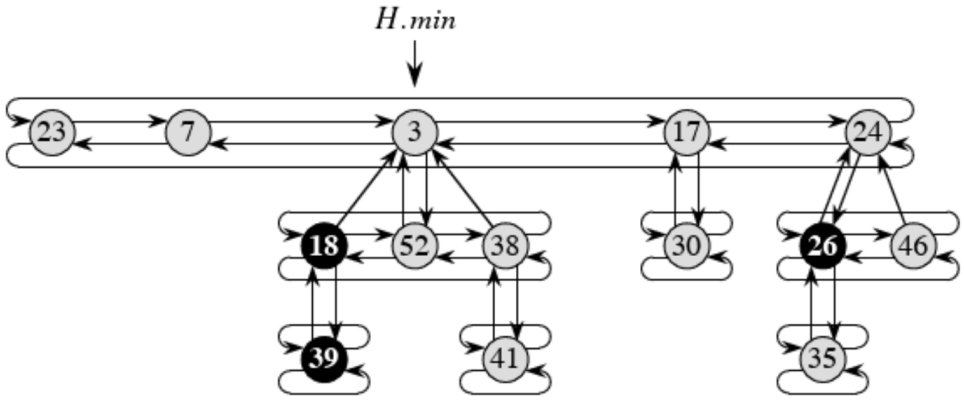

Figure 1: A Fibonacci heap [CLRS].

For a given Fibonacci heap H, we denote by:

- $t(H)$: the number of trees in the root list
- $m(H)$: the number of marked nodes in H

Define the potential function,

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

(a) Verify that Φ is indeed a potential function.

(b) Compute the potential of the Fibonacci heap in Figure 1.

Define time units so that one time unit is sufficiently large to cover the cost of any constant time operation we encounter.

Let $D(n)$ be an upper bound on the maximum degree of any node in an n-node Fibonacci heap. It can be shown that, $D(n) \leq \lfloor \log n \rfloor$.

The idea behind the operations is **to delay work as long as possible**.

(c) We implement $insert(H, x)$ by simply adding x to the root list of H in constant time.

```
insert(H, x):
    // pre: x.key initialized
    x.deg = 0; x.parent = null; x.child = null; x.mark = false;
    insert x into H's root list
    if x.key < H.min.key: H.min = x.key
    ++H.n
```

Show that the amortized cost of *insert* is constant.

(d) We implement merge by simply concatenating root lists (also in constant time):

```
merge(H1, H2):
    // post: returns a new Fib. heap: H = H1 U H2
    H's root list = (H1's root list) concatenate (H2's root list)
    H.min = element with smaller key of { H1.min,  H2.min }
    H.n = H1.n + H2.n
    return H
```

Show that the amortized cost of *merge* is constant.

(e) *delete-min* is the operation where the delayed work of consolidating the trees in the root list finally occurs.

```
delete_min(H):
    // pre: H is not empty
    z = H.min
    for each child x of z:
        moving x to root list of H // update left and right pointers
        x.parent = null
    remove z from root list of H
    if z == z.right:
        // no item in root list besides z (after moving z's children)
        H.min = null
    else:
        consolidate(H)
    --H.n
    return z
```

The subroutine *consolidate* repeatedly executes the following steps until every root in the root list has a distinct degree value:

i. find two roots x, y with the same degree, and $x.key \leq y.key$;
ii. **Link** y to x: remove y from root list, make y child of x (incrementing degree of x and clearing the mark on y).

This procedure is very similar to *binomial-heap::make-proper* from lecture.

```
consolidate(H):
    n = H.n
    // compute log H.n (an upper bound on D(H.n))
    for (l = 0; n > 1; n /= 2):
        ++l
    A = array of size l+1, initialized all null
    for each node w in the root list of H:
        x = w
        d = x.degree
```

```
        while A[d] != null:
            y = A[d]        // another root with same degree as x
            if x.key > y.key:
                swap(x, y)
            link(H, y, x)       // as above in (ii)
            A[d] = null
            d++
        A[d] = x
    H.min = null
    for i = 0..l:
        if(A[i] != null):
            insert A[i] into H's root list  // updating H.min if necessary
```

Show that the amortized cost of *delete-min* is $O(\log n)$.

Hint: Show that it is in $O(D(n))$, by first showing that total actual work in extracting the minimum node is in $O(D(n) + t(H))$.

3. **Binary counter.** A *binary n-bit counter* counts upward from zero as an array n bits (the left-most bit is least significant). It supports the operation *increment*, which adds 1 to the counter:

```
increment(A[0..n-1]):
    i = 0
    while (A[i] != 0):
          A[i] = 0
          ++i
    A[i] = 1
```

The running time for *increment* is $\Theta(k)$, where k is the final value of variable i, which is $\Theta(n)$ in the worst case. Show the amortized cost per *increment* of $\Theta(1)$.