

**LECTURE NOTES FOR CMSC 651
AMORTIZED DATA STRUCTURES**

1 REVISITING DIJKSTRA'S ALGORITHM

Recall the following problem and the usual algorithm for it.

Single Source Shortest Path Problem: Given a weighted graph $G = (V, E, w)$ where $w : E \rightarrow \mathbb{N}$ and given a node s , find, for all $v \in V$, the shortest distance from s to v .

Notation 1.1 If $G = (V, E)$ is a graph then N is the function that, given a node v , returns the set of all the Neighbors of v .

The usual algorithm is called **Dijkstra's algorithms**. Here is the basic idea.

1. Keep a set of nodes Q such that we do NOT know the shortest distance from any $v \in Q$ to s . Initially $Q = V$, though in the first phase s will be deleted from Q . (The distance from s to s is 0.)
2. Keep a function $d : V \rightarrow \mathbb{N}$ such that $d(v)$ is the current estimate of the shortest distance from s to v . Initially $d(s) = 0$ and $(\forall v \neq s)[d(v) = \infty]$ After the first stage we will have
 - (a) $d(s) = 0$
 - (b) $(\forall v \in N(s))[d(v) = w(s, v)]$.
 - (c) $(\forall v \notin N(s) \cup \{s\})[d(v) = \infty]$.
3. Every stage delete from Q the vertex v with the minimal value of d and update the values of d for the neighbors of v .

(NOTE- IN CLASS ON THE BLACKBOARD I WILL GIVE EXAMPLES.)

We formalize this.
DIJKSTRA'S ALGORITHM FOR SSSP

$Q = V$

$d(s) = 0$

for $v \in V - \{s\}$

$d(v) = \infty$

INSERT v along with key $d(v)$ into the data structure Q .

while $Q \neq \emptyset$

$u = \text{FINDMIN}(Q)$

$\text{DELETEMIN}(Q)$

for all $v \in N(u)$ $d(v) = \min(d(v), d(u) + w(u, v))$ (this is a DECKEY)

END OF ALGORITHM

The proof that this is correct is a standard topic in an undergraduate course; hence we omit it.

We are interested in analyzing this algorithm.

- The initialization of variables takes $O(V)$ steps.
- There are $O(V)$ calls to INSERT.
- There are $O(V)$ calls to FINDMIN.
- There are $O(V)$ calls to DELETEMIN.
- How many calls are there to DECKEY ? When a vertex u is the result of FINDMIN there are $\deg(u)$ calls to DECKEY. Every vertex is the result of FINDMIN exactly once. Hence the number of calls to DECKEY is $\sum_{u \in V} \deg(u) = 2E = O(E)$.

So we need a data structure that supports FINDMIN, DELETEMIN, and DECKEY. We would like it to be especially fast on DECKEY since that is called more often.

Recall the (ordinary) binary heap data structure. It supports these three operations. If it has V elements in it then the time needed is as follows.

- INSERT takes $O(\log V)$. But doing all of them in a row at the beginning, as we do, takes $O(V)$ total.
- FINDMIN takes $O(1)$.
- DELETEMIN takes $O(\log V)$
- DECKEY takes $O(\log V)$.

So if we use this data structure the algorithm takes

$$O(V + V \log V + E \log V) = O(E \log V).$$

Can we do better?

The real problem is that DECKEY takes $O(\log V)$. We would be happy with a data structure that took *less* time on DECKEY even if it took *more* time on the other operations. In particular, we would be happy with the following:

- INSERT takes $O(\log V)$
- FINDMIN takes $O(\log V)$.
- DELETEMIN takes $O(\log V)$
- DECKEY takes $O(1)$.

If we had such then we would have a data structure then we could implement Dijkstra's algorithm in

$$O(V \log V + E).$$

But note, we don't *quite* need that DECKEY takes $O(1)$. We are going to be doing E DECKEY operations. We need that doing E of them takes $O(E)$. This leads to the study of amortized data structures.

2 BINOMIAL HEAPS

(The examples of Binomial Heaps and the operations in CLRS are very good.)

Def 2.1 We define a set of trees. B_0 is the tree that is just one node. B_{i+1} is formed by taking two copies of B_i , and hanging one of them off of the root of the other. If T is a B_i we say that T is of *type* i .

The following facts are easily verified.

Fact 2.2

1. For all k , B_k has 2^k nodes.
2. The depth of B_k is k .
3. If you take B_k and remove the root you have the set $\{B_0, B_1, \dots, B_{k-1}\}$.
(This can be proved by induction.)

We will be storing keys at the nodes of a *set of* these trees.

Def 2.3 A *labeled* B_i is a B_i with keys (usually numbers) at the nodes. If this labeled B_i is a heap, we call it a heap instead of a labeled tree.

Def 2.4 A *Binomial Heap* is a set of labeled B_i 's such that the following holds.

1. Each labeled B_i is a MIN-heap.
2. For all i there is at most one B_i .
3. The roots of the B_i are a linked list ordered by the type.
4. All heaps B_i also store the type i . If T is a labeled B_i then $ind(T) = i$.
5. A pointer to the MIN is maintained. (This is not standard.)

Lemma 2.5 *If a binomial heap has n nodes then it has at most $\lfloor \lg n \rfloor + 1$ different B_i 's and has depth at most $\lfloor \lg n \rfloor$.*

Proof:

1) Assume, by way of contradiction, that there are $\lfloor \lg n \rfloor + 2$ B_i 's. Since they are all of different indices, one of them is B_i with $i \geq \lfloor \lg n \rfloor + 1$. By Fact 2.2.1 this B_i will have $2^i \geq 2^{\lfloor \lg n \rfloor + 1} > n$ nodes. This is a contradiction.

2) Assume, by way of contradiction, that some B_i has depth $\lfloor \lg n \rfloor + 1$. Then $i = \lfloor \lg n \rfloor + 1$, so B_i is $B_{\lfloor \lg n \rfloor + 1}$ and hence has $2^{\lfloor \lg n \rfloor + 1} > n$ nodes. This is a contradiction. ■

We describe how to carry out UNION, INSERT, FINDMIN, DELETEMIN, DECKEY, DELETE; and analyze their complexities. The operation UNION is not needed for our application; however, it will be used as a subroutine when carrying out the other operations. The operation DELETE is not needed for our application.

2.1 UNION

(See the pictures on Page 464-465 of CLRS.) We need an auxiliary operation that takes two ordinary MIN-heaps (that use B_i 's as the underlying tree) and combines them if possible.

COMBINE(S, T)

if ($\text{type}(S) \neq \text{type}(T)$) then RETURN(ERROR) (program ends here)

if $\text{key}(\text{root}(S)) < \text{key}(\text{root}(T))$ then

R =heap formed by hanging T off of the root of S

RETURN(R)

else

R =heap formed by hanging S off of the root of T

RETURN(R)

END OF ALGORITHM FOR COMBINE

We now present the UNION algorithm

UNION

Input is two binomial heaps. We denote them by

\mathcal{T}_1 which is the linked list of T_{i_1}, \dots, T_{i_p} where $i_1 < i_2 < \dots < i_p$,
and T_i is B_i with the nodes labeled, and

\mathcal{T}_2 is the linked list of T_{j_1}, \dots, T_{j_q} where $j_1 < j_2 < \dots < j_q$, and T_i is
 B_i with the nodes labeled.

Both lists of heaps have pointers to their MINS. Compare the MINS. Keep
the pointer that is pointing to the MIN of the two MINS.

Merge these two lists. We now have a linked list of $T_{k_1}, T_{k_2}, \dots, T_{k_{p+q}}$ where
 $k_1 \leq k_2 \leq \dots \leq k_{p+q}$. Since some of the indices may be equal, we do
not have a binomial heap yet. Note that we have at most 2 of any type.

In this step we go through the list and combine heaps that are of the same
type. For example, we can combine two T 's of type B_5 to get a T of
type B_6 . Initially there are at most 2 of any type. There may be 3 of
the same type as we process the heaps; however, there will never be
more than 3.

```
result =  $\emptyset$  (this will be the new list of labeled heaps)
carry =  $T_{k_1}$  (this will be the possible carry from the last "addition")
for  $i = 2$  to  $p + q$ 
    if (type(carry) < type( $T_{k_i}$ )) then
        result = COMBINE(result, carry)
        carry =  $T_{k_i}$ 
    else if (type(carry) > type( $T_{k_i}$ )) then result = COMBINE(result,  $T_{k_i}$ )
    else if (type(carry) = type( $T_{k_i}$ )) then carry = COMBINE(carry,  $T_{k_i}$ )
```

END OF ALGORITHM FOR UNION

2.2 INSERT

To INSERT a node create a labeled B_0 with the correct key and then do
UNION with what is already there. This takes $O(\log n)$.

2.3 FINDMIN

To FINDMIN we just use the pointer to the min. This takes $O(1)$.

2.4 DELETEMIN

(See page 469 of CLRS)

Remove the MIN. By Fact 2.2.3 the children of the prior MIN now form a binomial heap except that we don't have the pointer to the MIN. Find the MIN of the roots of this Binomial Heap. We now have two binomial heaps. Apply UNION to these two Binomial Heaps. This takes $O(\log n)$.

2.5 DECKEY

This is done exactly the same way as decrease key in ordinary binary heaps. This takes $O(\log n)$ steps. If the key being decreased ends up at the root then there are two cases.

1. The root was the MIN. Now the new root is the MIN since it is smaller. So the pointer to the min still points to this root.
2. The root was not the MIN. Now compare this to the MIN and if it is smaller than redirect the pointer.

Both of the two cases take $O(1)$. Hence the total number of steps is $O(\log n)$.

2.6 DELETE

DELETE is easy since you can use two other operations to achieve it:

DELETE(x)

DECKEY($x, -\infty$) (x is now the minimum key.)

DELETEMIN

END OF ALGORITHM FOR DELETE

Since DECKEY takes $O(\log n)$ time and DELETEMIN takes $O(\log n)$ steps, DELETE takes $O(\log n)$.

2.7 UPSHOT

We summarize how well the Binomial Heap does and compare it with an ordinary Binary Heap.

Operation	Binary Heap	Binomial Heap
INSERT	$O(\log n)$	$O(\log n)$
FINDMIN	$O(1)$	$O(1)$
DELETEMIN	$O(\log n)$	$O(\log n)$
DECKEY	$O(\log n)$	$O(\log n)$
UNION	$O(n)$	$O(\log n)$
DELETE	$O(\log n)$	$O(\log n)$

Note that Binomial heaps are better than Binary heaps for UNION and tied for everything else. DECKEY still takes $O(\log n)$ so these will not help us with Dijkstra's algorithm. Fibonacci heaps will be better than Binomial heaps, but only in the amortized sense; however, this will help us for Dijkstra's algorithm.

3 AMORTIZED DATA STRUCTURES

An *amortized data structure* is one where we are concerned with how many steps it takes to do a sequence of operations, rather than the worst case for one operation. Typically when an operation takes a long time, it also makes the data structure easier to use for later operations.

We look at an easy amortized data structures to see what kinds of techniques can be used to analyze them.

3.1 STACKPLUS

Def 3.1 A STACKPLUS is a data structure that supports the following operations. The data is a list of elements. There is no implied order of the elements (e.g., they are not necessarily sorted).

- PUSH(x): Put x on top of the list
- POP: Removes the top element of the list.

- MULTIPOP(k): Remove the top k elements of the list.

One can implement this with a linked list or with a an unbounded array with a pointer to the top element. Skipping details, the following seems to hold:

PUSH takes $O(1)$ steps.

POP takes $O(1)$ steps.

MULTIPOP(k) takes $O(k)$ steps.

If you do n operations than the most number of elements in the structure can be n . Hence MULTIPOP has worst case $O(n)$. Therefore you have worst case $O(n^2)$.

This does not seem quite right. It would seem that cannot remove alot of elements enough times to actually get $O(n^2)$ operations. This intuition is correct and we show how to pin it down.

3.2 POTENTIAL FUNCTIONS

We discuss Potential Functions in general, then apply them to STACKPLUS. We will later apply them to more complicated data structures (Fib-Heaps and Union-Find with Path Compression).

Def 3.2 Let D be a data structure. Let $\Phi : D \rightarrow \mathbf{N}$ be a function (which we call a potential function). Let c_1, c_2, \dots, c_n be a sequence of operations. Let D_0 be what the structure looks like initially (usually empty). Let D_i ($1 \leq i \leq n$) be what the data structure looks like after c_1, c_2, \dots, c_i are performed. For each i , $1 \leq i \leq n$ the *amortized cost of c_i with respect to (D, D_0, Φ)* is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

Typically, the larger the potential the more complicated the data structure. We are hoping that an expensive operation also makes the data structure simpler, so the amortized cost is small.

Theorem 3.3 Let D , Φ , $\{D_i\}_{i=0}^n$, $\{c_i\}_{i=1}^n$, $\{\hat{c}_i\}_{i=1}^n$, be as in Definition 3.2. If $\Phi(D_n) - \Phi(D_0) \geq 0$ then

$$\sum_{i=1}^n c_i \leq \sum_{i=1}^n \hat{c}_i.$$

Proof:

$$\begin{aligned}\sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= (\sum_{i=1}^n c_i) + \sum_{i=1}^n (\Phi(D_i) - \Phi(D_{i-1}))\end{aligned}$$

The second sum is

$$(\Phi(D_1) - \Phi(D_0)) + (\Phi(D_2) - \Phi(D_1)) + (\Phi(D_3) - \Phi(D_2)) + \dots + (\Phi(D_n) - \Phi(D_{n-1})).$$

Note that most of the terms cancel leaving just $\Phi(D_n) - \Phi(D_0)$. Hence we have

$$\begin{aligned}\sum_{i=1}^n \hat{c}_i &= (\sum_{i=1}^n c_i) + \Phi(D_n) - \Phi(D_0) \\ &\geq (\sum_{i=1}^n c_i)\end{aligned}$$

■

We will use this notion in the following way. When we do an operation we also measure how much it changes the data structure via the potential function. Typically we will show that the amortized cost \hat{c}_i is small (say $O(1)$ or $O(\log^* n)$) so that the sum of these costs is small (say $O(n)$ or $O(n \log^* n)$). Hence an operation that is costly will be balanced off by reducing the potential of the data structure.

We now apply this to STACKPLUS. Let

$$\Phi(D) = \text{the number of elements in } D .$$

- When you do a POP this costs 1, but the potential has decreased by 1. So the amortized cost is 0. (This is not a cause for celebration-the actual cost was already $O(1)$.)
- When you do a PUSH this costs 1, but the potential has increased by 1. So the amortized cost is 2. (This is not a cause for worry-the amortized cost is still $O(1)$.)
- When you do MULTIPOP(k) this costs k , but the potential has decreased by k . Hence the amortized cost is 0. (This is a cause for celebration.)

If you start with an empty stack and do n operations the initial potential $\Phi(D_0) = 0$, and $0 \leq \Phi(D_n)$. So $\Phi(D_n) - \Phi(D_0) \geq 0$. Hence, by Theorem 3.3, we can use the sum of the amortized costs to bound the sum of the actual costs. By the above analysis, the sum of the amortized costs is $O(n)$. Hence n operations take $O(n)$ steps.

4 FIB HEAPS

Recall that a Binomial Heap is a set of MIN-heaps where

- Only use heaps that are labeled B_i 's.
- For each i have at most one labeled B_i .

A Fibonacci Heap will be much less restrictive. Many of the operations will be easy since we do not have to maintain as much structure. But some of the operations will be complicated. The key will be that the operations that are complicated will decrease a potential function. (Why it is called 'Fibonacci Heap' we will see much later.)

Def 4.1 A *Fibonacci Heap* (henceforth Fib-heap) is a collection of MIN-heaps with the following auxiliary information.

1. We keep a pointer to the root with the minimum key
2. The roots are in doubly linked list (we do not require that the order they are in is meaningful).
3. Some nodes are marked. In particular
 - When a node is first inserted it is unmarked.
 - If an unmarked node loses a child in the DECKEY operation then it becomes marked.
 - If a marked node becomes the root of new heap during the DECKEY operation then it becomes unmarked.

Notation 4.2 Let FIB be the function defined by

$$\begin{aligned} \text{FIB}(0) &= 1 \\ \text{FIB}(1) &= 2 \\ \text{FIB}(n) &= \text{FIB}(n-1) + \text{FIB}(n-2) \end{aligned}$$

Notation 4.3 We denote by $D(n)$ a bound on the degree of a node assuming that the structure never holds more than n elements. We will later show that a node of degree d has at least $\text{FIB}(d)$ nodes under it (including itself). This is exponential (roughly $(\frac{\sqrt{5}+1}{2})^d$). Since the Fib-Heap has at most n nodes, $D(n) \leq O(\log n)$.

Note 4.4 $D(n)$ is based on the number of *elements* in the structure, not the number of operations.

Def 4.5 If H is a Fib-heap then

- $t(H)$ is the number of heaps in H .
- $m(H)$ is the number of marked nodes in H .
- $\Phi(H) = [t(H) + 2m(H)]\alpha$ where α is a constant to be named later. This is the potential function.

The intention is that an expensive operation will also make the potential decrease. We analyze the operations UNION, INSERT, FINDMIN, DELETEMIN, DECKEY, and DELETE. Most of these operations will be trivial since there is not much structure to maintain.

4.1 UNION

The UNION of two Fib-heaps is obtained by just concatenating the two doubly linked lists and comparing the two MIN's to see which one is the MINIMAL (and redirect the pointer). The actual cost is $O(1)$. The number of heaps stays the same, and the number of marked nodes stays the same, hence the amortized cost is $O(1)$.

4.2 INSERT

Use UNION of the existing Fib-heap and the one-node Fib-heap. The actual cost is $O(1)$ and the potential function goes up by $O(1)$, hence the amortized cost is $O(1)$.

4.3 FINDMIN

We maintain a pointer to the MIN element, so this takes $O(1)$ steps. The potential does not change so the amortized cost is $O(1)$.

4.4 DELETEMIN

(See pages 484-485 of CLRS for a nice example)

This will be complicated. At the end of this operation we will have less heaps hence the cost of the operation will be balanced out by the decrease in potential.

We first describe it informally.

1. Remove MIN.
2. Make all of the children of MIN into heaps in the doubly linked list. Insert them in order to the left of what was the MIN. (This takes at most $O(D(n))$ steps.)
3. Starting at the root to the right of what was the MIN, traverse the list of heaps and whenever you find two of the same degree combine them by attaching one (the one with the larger root) to the root of the other.
4. Keep doing the previous step until all the heaps have different degrees. Note that this will only be $D(n)$ heaps.
5. Look at all of the roots to find the new MIN. (This will take $O(D(n))$ steps.)

We now describe formally how to traverse the list of heaps and combine those of the same degree. We will keep an array $A[0..D(n)]$. At all times $A[i]$ will either be \perp (null) or have a pointer to a root of a heap of degree i . Initially it is an array of null pointers.

We need a subroutine that combines two (ordinary) heaps of the same degree and updates A .

```

COMBINE( $r, s$ ) ( $r$  and  $s$  are both roots of heaps)

if  $\text{deg}(r) \neq \text{deg}(s)$  then RETURN(ERROR)

 $d = \text{deg}(s)$ 

if  $\text{key}(r) < \text{key}(s)$  then

    Make  $s$  a child of  $r$  (the heap with root  $s$  is destroyed)
     $\text{deg}(r) = \text{deg}(r) + 1$ .
    If  $A[d + 1] = \perp$  then  $A[d + 1]$  gets a pointer to  $r$ 
        else COMBINE( $r, A[d + 1]$ )

else (so  $\text{key}(r) \geq \text{key}(s)$ )

    Make  $r$  a child of  $s$  (the heap with root  $r$  is destroyed)
     $\text{deg}(s) = \text{deg}(s) + 1$ .
    If  $A[d + 1] = \perp$  then  $A[d + 1]$  gets a pointer to  $s$ 
        else COMBINE( $s, A[d + 1]$ )

```

END OF ALGORITHM FOR COMBINE

We now formally describe the procedure TRAVERSE which traverses the list of roots and merges those with the same degree.

TRAVERSE

For $d = 1$ to $D(n)$ $A[d] = \perp$

Traverse the set of roots of heaps. When you are at root r do the following.

Let $d = \deg(r)$.

If $A[d] = \perp$ then let $A[d]$ be a pointer to r .

If $A[d] \neq \perp$

 COMBINE($A[d], r$)

$A[d] = \perp$

END OF ALGORITHM FOR TRAVERSE

Not counting the TRAVERSE step, DELETEMIN takes $O(D(n))$ steps. We now analyze the TRAVERSE step and the change in potential.

Let $t(H)$ be the number of heaps in the Fib-heap before DELETEMIN is executed. The first step creates $O(D(n))$ additional heaps. So TRAVERSE will be looking at $O(t(H) + D(n))$ heaps.

We need to count how many times COMBINE is called. For all $d \leq D(n)$ let a_d be the number of heaps that have degree d after we have removed the min and have the extra heaps, but before any other action is taken. Note that

$$\sum_{i=1}^{D(n)} = t(H) + O(D(n)) = O(T(H) + D(n)).$$

Let A_d be the number of time COMBINE is called with two heaps of degree d . Note that

$$\begin{aligned} A_1 &= \lfloor a_1/2 \rfloor \leq a_1/2 \\ A_2 &= \lfloor a_2/2 + A_1/2 \rfloor \leq a_2/2 + A_1/2 \\ A_3 &= \lfloor a_3/2 + A_2/2 \rfloor \leq a_3/2 + A_2/2 \\ &= \vdots \\ A_{D(n)} &= \lfloor a_{D(n)}/2 + A_{D(n)-1}/2 \rfloor \leq a_{D(n)}/2 + A_{D(n)-1}/2 \end{aligned}$$

Sum both sides to get

$$\sum_{i=1}^{D(n)} A_i \leq \frac{1}{2} \sum_{i=1}^{D(n)} a_i + \frac{1}{2} \sum_{i=1}^{D(n)-1} A_i \leq \frac{1}{2} \sum_{i=1}^{D(n)} a_i + \frac{1}{2} \sum_{i=1}^{D(n)} A_i$$

So

$$\frac{1}{2} \sum_{i=1}^{D(n)} A_i \leq \frac{1}{2} \sum_{i=1}^{D(n)} a_i = O(t(H) + D(n))$$

Hence

$$\sum_{i=1}^{D(n)} A_i = O(t(H) + D(n)).$$

Therefore *COMBINE* is called $O(t(H))$ times. In addition to the $O(D(n) + D(n))$ steps in the rest of the algorithm, this yields $O(t(H) + D(n))$ steps.

Let us now look at how the potential changes. No nodes are marked or unmarked in this process so we need only look at the number of heaps.

Before *DELETEMIN* is called there are $t(H)$ heaps.

After *DELETEMIN* is called there are $D(n)$ heaps.

Hence the change in potential is $(D(n) - t(H))\alpha$ where α is a constant to be specified later. The amortized time is

$$O(t(H) + D(n)) + (D(n) - t(H))\alpha.$$

Choose α large enough so that the $t(H)$'s cancel. Hence the amortized times is $O(D(n))$.

4.5 DECKEY

(Page 491 of CLRS has an okay but not great example of this.)

We are *not* going to decrease the key and trickle up. For the *DECKEY* part we are going to do something simple, but then we are going to lower the potential of the Fib-heap.

We now describe *DECKEY*

DECKEY(x, k) (We are going to decrease the key of x to k .)

Change the key value of x to k .

Let $y = \text{parent}(x)$.

Take the sub-heap rooted at x and make it a new heap to the left of the current MIN (we take the left for the sake of being definite.)

Update the MIN pointer if needed.

If x was marked then make it unmarked.

If y was unmarked then mark it else DECKEY($y, val(y)$)

Intuitively we are moving x , and if its parent was marked, move and unmark its parent, and if its parents parent was marked then move and unmark that, etc. So we need to deal with the continous marked ancestors of x .

How much is the actual cost, and what is the change of potential? Let c such that $x, \text{parent}(x), \dots$ (c times) $\text{parent}(\text{parent}(\dots x \dots))$ are all marked. These are now unmarked, but the next item up is now marked. Hence there are $c - 1$ less marked nodes.

The actual cost is $O(c)$ since you will call DECKEY $O(c)$ times. The number of new heap created is c .

Let H be the old Fib-heap and H' be the new one. The change in potential is

$$((t(H') + 2m(H')) - (t(H) + 2m(H)))\alpha = ((t(H') - t(H)) + 2(m(H') - m(H)))\alpha.$$

Recall that the operation added c new heaps but unmarked c nodes. Hence

$$t(H') - t(H) = c$$

and

$$m(H') - m(H) = -c + 1$$

Hence

$$((t(H') - t(H)) + 2(m(H') - m(H)))\alpha = (c - 2c + 2)\alpha = -c\alpha + O(1).$$

(This is why we had that factor of 2.)

So the amortized cost is

$$O(c) - c\alpha + O(1)$$

Pick α large enough so that this is $O(1)$.

4.6 DELETE

DELETE is easy since you can use two other operations to achieve it:

DELETE(x)

DECKEY($x, -\infty$) (This will make x the minimum key and the subheap rooted at x will be its own heap.)

DELETEMIN

END OF ALGORITHM FOR DELETE

Since DECKEY has amortized $O(1)$ time and DELETEMIN has amortized $O(D(n))$ steps, DELETE has amortized $O(D(n))$.

4.7 Analysis of Fib-Heaps

UNION, INSERT, FINDMIN, and DECKEY all take amortized $O(1)$ steps. DELETEMIN and DELETE take amortized $O(D(n))$. We need to bound $D(n)$.

Lemma 4.6 *Let x be a node. Assume that at stage s there are nodes y_1, y_2, \dots, y_d which are the children of x in the order they were made children of x . Then $\deg(y_1) \geq 0$ and, for all $i \geq 2$, $\deg(y_i) \geq i - 2$.*

Proof: We do *not* prove this by induction on i ; however, there are two cases, $i = 1$ and $i \geq 2$.

If $i = 1$ then clearly $\deg(y_1) \geq 0$ since a node can never have negative degree.

Look at y_i with $i \geq 2$. Let $t \leq s$ be the stage where y_i is made a child of x . Then, at stage t ,

1. $\deg(x) = \deg(y_i)$.
2. y_1, \dots, y_{i-1} are children of x . (There may be more children that are lost by stage s .)
3. Hence $\deg(y_i) \geq i - 1$.

If between stages t and s y_i loses a child then this will happen during a DECKEY operation, and y_i will be marked. If y_i loses a second child then this will also happen during DECKEY operation; however, in this case, the subheap rooted at y_i will be made its own new heap, hence y_i will cease to be a child of x . Since y_i is a child of x , this does not happen. Hence y_i loses at most one child between stages t and s . Therefore, at stage s , $\deg(y_i) \geq i - 2$.

■

Def 4.7 Let F be defined as follows:

$$\begin{aligned} F(0) &= 1 \\ F(1) &= 2 \\ F(d) &= 2 + \sum_{i=2}^d F(i-2) = 2 + \sum_{i=0}^{d-2} F(i) \end{aligned}$$

Lemma 4.8 *If x has degree at least d then the heap rooted at x has at least $F(d)$ elements (this includes x itself).*

Proof: We prove this by strong induction.

If $d = 0$ then the heap rooted at x is just x itself, so it has $1 = F(0)$ elements.

If $d = 1$ then the heap rooted at x has x itself and a child of x (which may or may not have more children) so there are at least $2 = F(1)$ elements.

Assume the lemma holds for all degrees $e \leq d - 1$. Let x be a node of degree d . Let y_1, \dots, y_d be its children. By Lemma 4.6 $\deg(y_1) \geq 0$ and $(\forall i \geq 1)[\deg(y_i) \geq i - 2]$. By the induction hypothesis

1. the number of nodes in the heap rooted at y_1 is $1 = F(0)$.
2. for $i \geq 2$ the number of nodes in the heap rooted at y_i is at least $F(i - 2)$.

Hence the number of nodes in the heap rooted at x is at least

$$1 + F(0) + \sum_{i=2}^d F(i-2) = F(d).$$

■

We now need to estimate $F(d)$. First lets look at some of its values

1. $F(0) = 1$.
2. $F(1) = 1 + F(0) = 2$.
3. $F(2) = 1 + F(0) + F(0) = 3$.
4. $F(3) = 1 + F(0) + F(0) + F(1) = 5$
5. $F(4) = 1 + F(0) + F(0) + F(1) + F(2) = 8$

This looks just like the Fibonacci numbers!

Lemma 4.9 For all k , $F(k) = FIB(k)$.

Proof: We will *not* prove this by induction. Instead we will show that $F(k)$ and $FIB(k)$ both solve the same problem.

Imagine that you want to tile the set $\{1, 2, \dots, k\}$ with tiles that are either 1×1 or 1×2 . For example you can cover $\{1, 2, 3\}$ with either

- three 1-tiles
- one 1-tile then one 2-tile
- one 2-tile then one 1-tile

Let $T(k)$ be the number of ways you can tile $\{1, \dots, k\}$.

How many ways can you do this? We solve this problem two ways.

Method 1: Clearly $T(1) = 1$ and $T(2) = 2$. We consider tilings of $\{1, \dots, k\}$ where $k \geq 2$. Either there is a 1-tile covering k or there is a 2-tile cover $\{k-1, k\}$. In the first case there are $k-1$ continuous spots left, so there are $T(k-1)$ ways to finish the tiling. In the second case there are $k-2$ continuous spots left, so there are $T(k-2)$ ways to finish the tiling. Hence

$$T(k) = T(k-1) + T(k-2)$$

This is not quite $FIB(k)$ since the base case is not the same; however it is easy to see that $T(k) = FIB(k-1)$. Hence $FIB(k) = T(k+1)$.

END of Method 1

Method 2: Clearly $T(1) = 1$ and $T(2) = 2$. We consider tilings of $\{1, \dots, k\}$ where $k \geq 2$. There are many cases.

- No 2-tile are used. There is only 1 way to tile.
- The first 2-tile used is on the space $(1, 2)$. There are $k - 2$ continous spots remaining. There are $T(k - 2)$ ways to tile.
- The first 2-tile used is on the space $(2, 3)$. There are $k - 3$ continous spots remaining. There are $T(k - 3)$ ways to tile.
- \vdots
- The first 2-tile used is on the space $(i, i + 1)$. There are $k - (i + 1)$ continous spots remaining. There are $T(k - (i + 1))$ ways to tile.
- \vdots
- The first 2-tile used is on the space $(k - 2, k - 1)$. There are $k - (k - 1)$ continous spots remaining. There are $T(k - (k - 1)) = T(1)$ ways to tile.
- The first 2-tile used is on the space $(k - 1, k)$. There are 0 tiles remaining. So there is only one way to do this. (Note that we have not defined $T(0)$ so we cannot reduce to that case.)

The first case and the last case together contribute 2. The rest of the cases contribute (reading from bottom to top) $T(1) + T(2) + \dots + T(k - 2)$
Hence

$$T(k) = 2 + \sum_{i=1}^{k-2} T(i).$$

We want to show that $F(k) = T(k + 1)$.

Let $G(k) = T(k + 1)$ and we will show that $G(k)$ satisfies the same recurrence and base case as $F(k)$.

Note that $G(0)$ exists

$$G(0) = T(1) = 1$$

$$G(1) = T(2) = 2$$

$$G(k) = T(k+1) = 2 + \sum_{i=1}^{(k+1)-2} T(i) = 2 + \sum_{i=1}^{k-1} T(i) = 2 + \sum_{i=0}^{k-2} T(i+1) = 2 + \sum_{i=0}^{k-2} G(i).$$

Hence G satisfies the exact same base case and recurrence as F . Therefore $F(k) = G(k)$. But also recall that $G(k) = T(k + 1)$ Hence $F(k) = T(k + 1)$.

END of Method 2

Since $FIB(k) = T(k + 1)$ and $F(k) = T(k + 1)$ we have $FIB(k) = F(k)$.

■

We now have to get an estimate on FIB . We find an exact solution for FIB . The method we use can be used to solve any recurrence of a similar type.

Recall that

Notation 4.10 FIB is the function defined by

$$\begin{aligned} FIB(0) &= 1 \\ FIB(1) &= 2 \\ FIB(n) &= FIB(n - 1) + FIB(n - 2) \end{aligned}$$

This looks exponential. We guess that α^n is a solution and see what α has to be

$$\alpha^n = \alpha^{n-1} + \alpha^{n-2}$$

$$\alpha^2 = \alpha + 1$$

$$\alpha^2 - \alpha - 1 = 0.$$

This has two solutions:

$$\alpha_1 = \frac{1+\sqrt{5}}{2}$$

and

$$\alpha_2 = \frac{1-\sqrt{5}}{2}.$$

It is easy to show that if f_1 and f_2 are solutions to FIB then any linear combination of f_1 and f_2 are solutions to FIB . Hence, for any constants A, B

$$A\alpha_1^n + B\alpha_2^n$$

is a solution to FIB .

The constants A, B can be determined by the initial conditions $FIB(0) = 1$ and $FIB(1) = 2$. We leave this to the reader.

Note that $0 < \alpha_2 < 1$, so α_2^n goes to 0. Therefore

$$FIB(n) = \Theta(\alpha_1^n).$$

Theorem 4.11 *If H is a Fib-heap with n elements in it then the maximum degree of node, $D(n)$, is $O(\log n)$.*

Proof: If a node x has degree d then the subheap rooted at x has $\Theta(\alpha_1^d)$ elements. Since $\Theta(\alpha_1^d) \leq n$ and $\alpha_1 > 1$, $d = O(\log n)$. ■

Now that we know $D(n)$ we can finish our table.

Operation	Binary Heap	Binomial Heap	Fib-Heap (amortized)
INSERT	$O(\log n)$	$O(\log n)$	$O(1)$
FINDMIN	$O(1)$	$O(1)$	$O(1)$
DELETEMIN	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECKEY	$O(\log n)$	$O(\log n)$	$O(1)$
UNION	$O(n)$	$O(\log n)$	$O(1)$
DELETE	$O(\log n)$	$O(\log n)$	$O(\log n)$

Fib-Heaps are better than Binary Heaps and Binomial Heaps; however, the cost is amortized. Nevertheless, we have achieved our goal. With Fib-Heaps Dijkstra's algorithm can be implemented in

$$O(V \log V + E)$$

steps.