Amortized Analysis & Fibonacci Heaps

Bin Wang

School of Software
Tsinghua University

March 27, 2018

Outline

How large should a hash table or an array be?

Goal

Make the table as small as possible, but large enough so that it won't overflow.

Problem

What if we don't know the proper size in advance?

How large should a hash table or an array be?

Goal

Make the table as small as possible, but large enough so that it won't overflow.

Problem

What if we don't know the proper size in advance?

How large should a hash table or an array be?

Solution

Dynamic tables.

Whenever the table overflows, “grow” it by allocating (via **malloc** or **new**) a new, larger table. Move all items from the old table into the new one, and free the storage for the old table.

How large should a hash table or an array be?

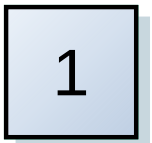
Solution

Dynamic tables.

Whenever the table overflows, “grow” it by allocating (via **malloc** or **new**) a new, larger table. Move all items from the old table into the new one, and free the storage for the old table.

Example of a dynamic table

1. INSERT



2. INSERT

3. INSERT

4. INSERT

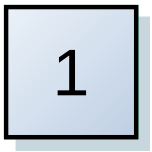
5. INSERT

6. INSERT

7. INSERT

Example of a dynamic table

1. INSERT



2. INSERT

overflow

3. INSERT

4. INSERT

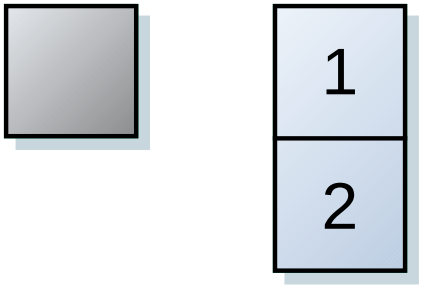
5. INSERT

6. INSERT

7. INSERT

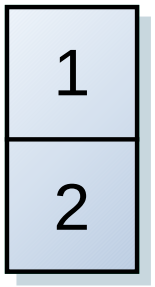
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT



Example of a dynamic table

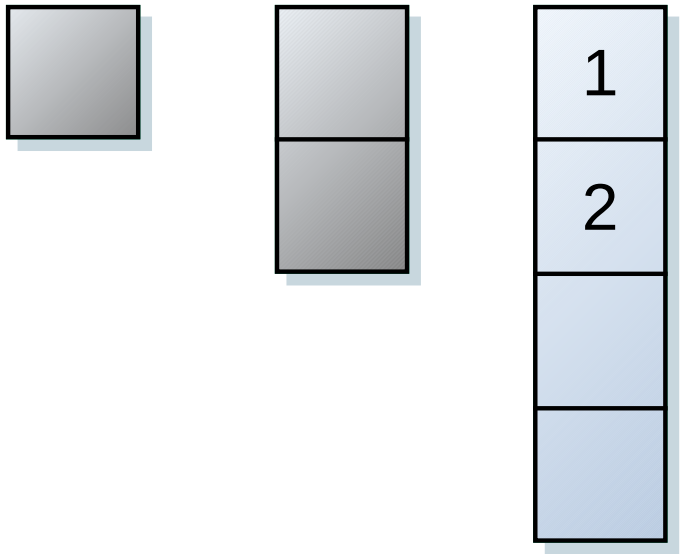
1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT



overflow

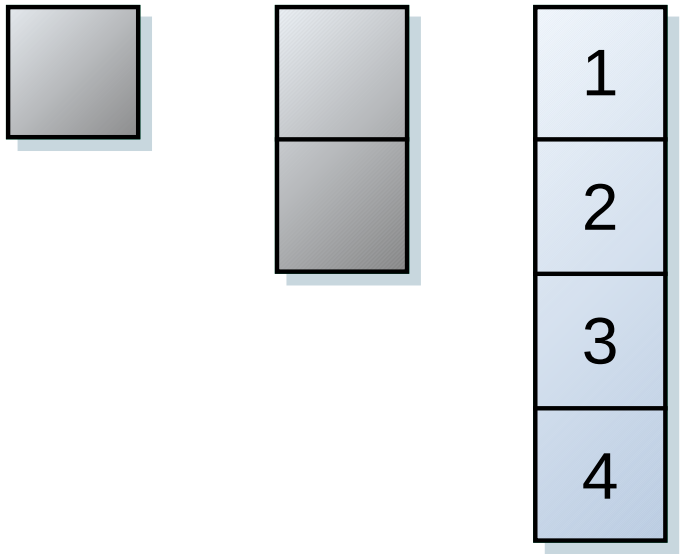
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT



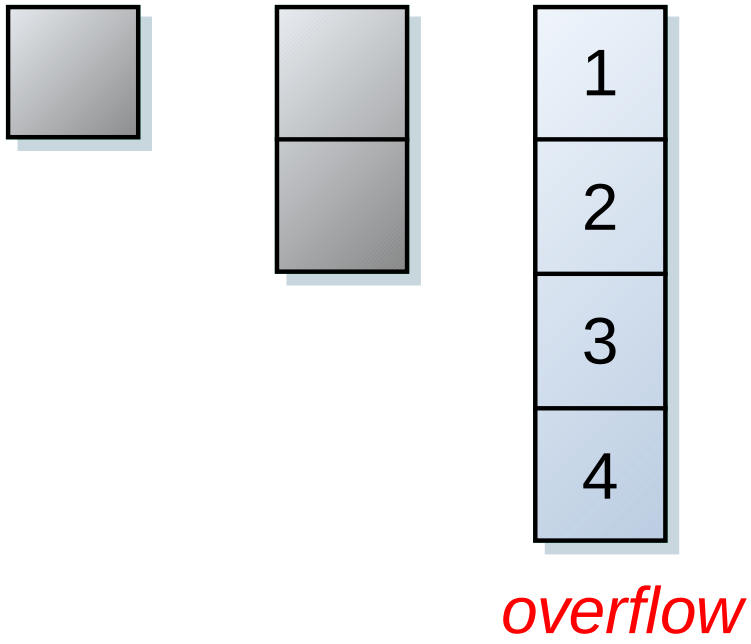
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT



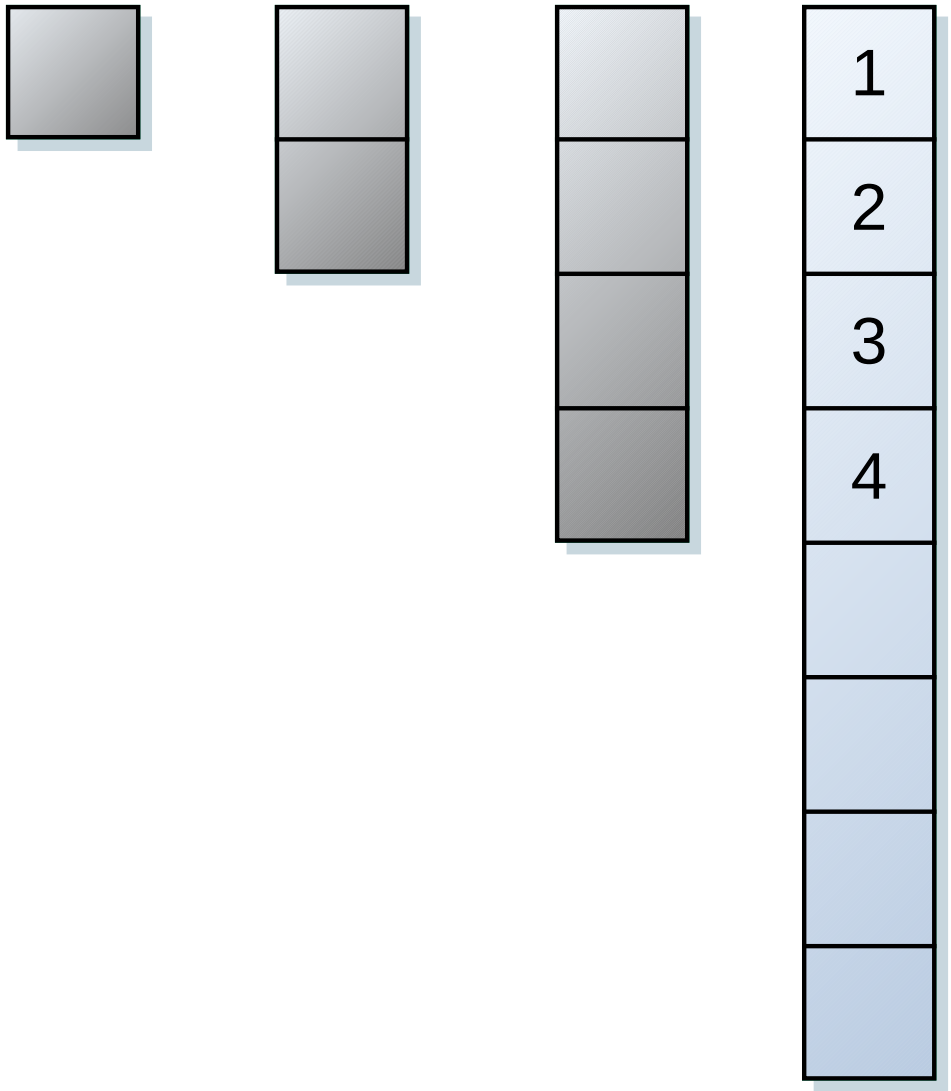
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT



Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT



Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT

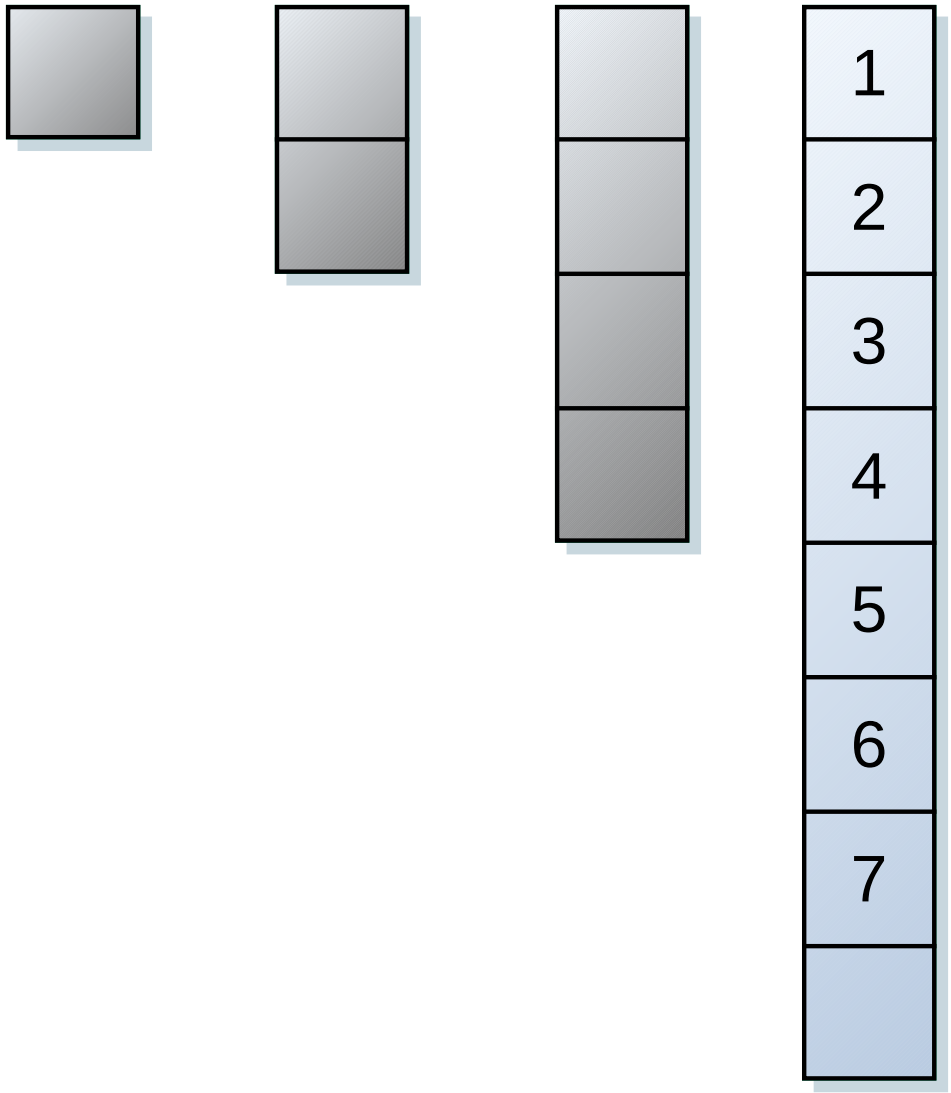


Table-Insert

TABLE-INSERT(T, x)

```
 1  if T.size == 0
 2        allocate T.table with 1 slot
 3        T.size = 1
 4  if T.num == T.size
 5        allocate new-table with 2 · T.size slots
 6        insert all items in T.table into new-table
 7        free T.table
 8        T.table = new-table
 9        T.size = 2 · T.size
10  insert x into T.table
11  T.num = T.num + 1
```

Worse-case analysis

A cursory analysis

Consider a sequence of n insertions. The worst-case time to execute one insertion is $\Theta(n)$. Therefore, the worst-case time for n insertions is $n \cdot \Theta(n) = \Theta(n^2)$.

WRONG!

In fact, the worst-case cost for n insertions is only $\Theta(n) \ll \Theta(n^2)$

Worse-case analysis

A cursory analysis

Consider a sequence of n insertions. The worst-case time to execute one insertion is $\Theta(n)$. Therefore, the worst-case time for n insertions is $n \cdot \Theta(n) = \Theta(n^2)$.

WRONG!

In fact, the worst-case cost for n insertions is only $\Theta(n) \ll \Theta(n^2)$

Worse-case analysis

A tighter analysis

Let c_i be the cost of the ith insertion.

$$c_i = \begin{cases} i & \text{if } i - 1 = 2^m, m = 0, 1, 2, \ldots \\ 1 & \text{otherwise} \end{cases}$$

i	1	2	3	4	5	6	7	8	9
$size_i$									
c_i									

Worse-case analysis

A tighter analysis

Let c_i be the cost of the ith insertion.

$$c_i = \begin{cases} i & \text{if } i-1 = 2^m, m = 0, 1, 2, \ldots \\ 1 & \text{otherwise} \end{cases}$$

i	1	2	3	4	5	6	7	8	9
$size_i$									
c_i									

Worse-case analysis

A tighter analysis

Let c_i be the cost of the ith insertion.

$$c_i = \begin{cases} i & \text{if } i-1 = 2^m, m = 0, 1, 2, \ldots \\ 1 & \text{otherwise} \end{cases}$$

i	1	2	3	4	5	6	7	8	9
$size_i$	1	2	4	4	8	8	8	8	16
c_i									

Worse-case analysis

A tighter analysis

Let c_i be the cost of the ith insertion.

$$c_i = \begin{cases} i & \text{if } i-1 = 2^m, m = 0, 1, 2, \ldots \\ 1 & \text{otherwise} \end{cases}$$

i	1	2	3	4	5	6	7	8	9
$size_i$	1	2	4	4	8	8	8	8	16
c_i	1	2	3	1	5	1	1	1	9

Worse-case analysis

A tighter analysis

Let c_i be the cost of the ith insertion.

$$c_i = \begin{cases} i & \text{if } i-1 = 2^m, m = 0, 1, 2, \ldots \\ 1 & \text{otherwise} \end{cases}$$

i	1	2	3	4	5	6	7	8	9
$size_i$	1	2	4	4	8	8	8	8	16
c_i	1	1	1	1	1	1	1	1	1
		1	2		4				8

Worse-case analysis

A tighter analysis

$$\text{Cost of } n \text{ insertions} = \sum_{i=1}^{n} c_i$$

Worse-case analysis

A tighter analysis

$$\text{Cost of } n \text{ insertions} = \sum_{i=1}^{n} c_i$$

$$\leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j$$

Worse-case analysis

A tighter analysis

$$\begin{aligned}\text{Cost of } n \text{ insertions} &= \sum_{i=1}^{n} c_i \\ &\leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< 3n\end{aligned}$$

Worse-case analysis

A tighter analysis

$$\begin{aligned}\text{Cost of } n \text{ insertions} &= \sum_{i=1}^{n} c_i \\ &\leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< 3n \\ &= \Theta(n)\end{aligned}$$

Worse-case analysis

A tighter analysis

$$\begin{aligned}\text{Cost of } n \text{ insertions} &= \sum_{i=1}^{n} c_i \\ &\leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< 3n \\ &= \Theta(n)\end{aligned}$$

Thus, the average cost of each dynamic-table operation is $\Theta(n)/n = \Theta(1)$.

Amortized analysis

Definition

An ***amortized analysis*** is any strategy for analyzing a sequence of operations to show that the ***average*** cost per operation is small, even though a single operation within the sequence might be expensive.

Amortized analysis

An amortized analysis guarantees the **average** performance of each operation in the **worst case**.

- the **aggregate** method,
- the **accounting** method,
- the **potential** method.

Amortized analysis

An amortized analysis guarantees the **average** performance of each operation in the **worst case**.

- the **aggregate** method,
- the **accounting** method,
- the **potential** method.

Aggregate analysis

Definition

- In ***aggregate analysis***, we show that for all n, a sequence of n operations takes worst-case time $T(n)$ in total.
- In the worst case, the average cost, or **amortized cost**, per operation is therefore $T(n)/n$.
- This amortized cost applies to each operation, even when there are ***several types*** of operations in the sequence.

Incrementing a binary counter

Definition

- Consider the problem of incrementing a k-bit binary counter that counts upward from 0.
- We use an array $A[0..k-1]$ of bits as the counter.

Incrementing a binary counter

INCREMENT(A)

```
1  i = 0
2  while i < A.length and A[i] == 1
3        A[i] = 0
4        i = i + 1
5  if i < A.length
6        A[i] = 1
```

Incrementing a binary counter

A cursory analysis

- A single execution of INCREMENT take time $\Theta(k)$ in the worst case, in which array A contains all 1's.
- Thus, a sequence of n INCREMENT operations on an initially zero counter takes time $O(nk)$ in the worst case.

Incrementing a binary counter

value	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	cost
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	
2	0	0	0	0	0	1	0	
3	0	0	0	0	0	1	1	
4	0	0	0	0	1	0	0	
5	0	0	0	0	1	0	1	
6	0	0	0	0	1	1	0	
7	0	0	0	0	1	1	1	
8	0	0	0	1	0	0	0	
9	0	0	0	1	0	0	1	
10	0	0	0	1	0	1	0	

Incrementing a binary counter

value	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	cost
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	1
2	0	0	0	0	0	1	0	3
3	0	0	0	0	0	1	1	4
4	0	0	0	0	1	0	0	7
5	0	0	0	0	1	0	1	8
6	0	0	0	0	1	1	0	
7	0	0	0	0	1	1	1	
8	0	0	0	1	0	0	0	
9	0	0	0	1	0	0	1	
10	0	0	0	1	0	1	0	

Incrementing a binary counter

value	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	cost
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	1
2	0	0	0	0	0	1	0	3
3	0	0	0	0	0	1	1	4
4	0	0	0	0	1	0	0	7
5	0	0	0	0	1	0	1	8
6	0	0	0	0	1	1	0	10
7	0	0	0	0	1	1	1	11
8	0	0	0	1	0	0	0	15
9	0	0	0	1	0	0	1	16
10	0	0	0	1	0	1	0	18

Incrementing a binary counter

A tighter analysis

- Not all bits flip each time INCREMENT is called.
- In general, for $i = 0, 1, \ldots, \lfloor \lg n \rfloor$, bit $A[i]$ flips $\lfloor n/2^i \rfloor$ times in a sequence of n INCREMENT operations on an initially zero counter.
- For $i > \lfloor \lg n \rfloor$, bit $A[i]$ never flips at all.

Incrementing a binary counter

A tighter analysis(cont.)

- The total number of flips in the sequence is thus

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \lfloor n/2^i \rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i}$$

Incrementing a binary counter

A tighter analysis(cont.)

- The total number of flips in the sequence is thus

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \lfloor n/2^i \rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i}$$
$$= 2n.$$

Incrementing a binary counter

A tighter analysis(cont.)

- The worst-case time for a sequence of n INCREMENT operations on an initially zero counter is therefore $O(n)$.
- The average cost of each operaton, and therefore the amortized cost per operation, is $O(n)/n = O(1)$.

Incrementing a binary counter

A tighter analysis(cont.)

- The worst-case time for a sequence of n INCREMENT operations on an initially zero counter is therefore $O(n)$.
- The average cost of each operaton, and therefore the amortized cost per operation, is $O(n)/n = O(1)$.

The accounting method

Definition

- Assign differing charges to different operations, with some operations charged more or less than they actually cost. The amount we charge an operation is called its **amortized cost**. The ith operation's amortized cost is $\hat{c}_i$.
- Any amount not immediately consumed is stored in the bank as **credit** for use by subsequent operations.

The accounting method

Definition

- Assign differing charges to different operations, with some operations charged more or less than they actually cost. The amount we charge an operation is called its **amortized cost**. The ith operation's amortized cost is $\hat{c}_i$.
- Any amount not immediately consumed is stored in the bank as **credit** for use by subsequent operations.

The accounting method

Definition

- The credit in the bank must not go negative! we require

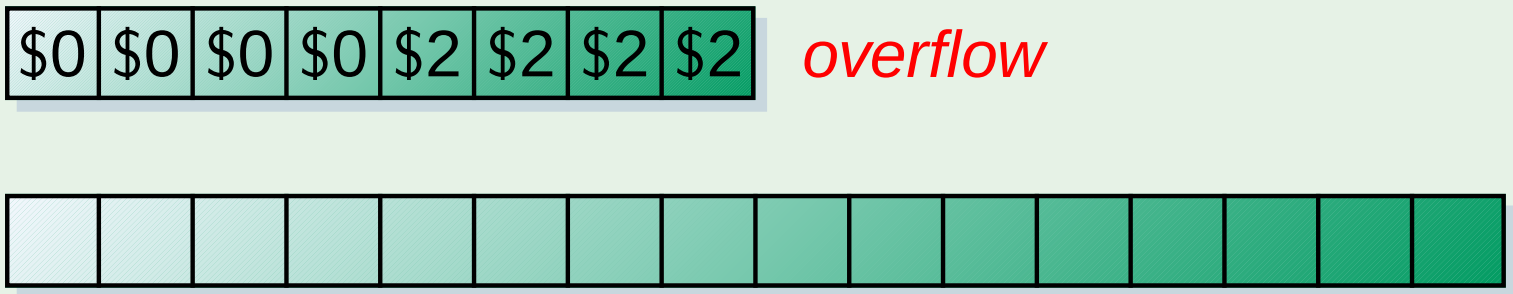
$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

for all n.

Accounting analysis of dynamic tables

- Charge an amortized cost of $\hat{c}_i = \$3$ for the ith insertion.
  - $\$1$ pays for the immediate insertion.
  - $\$2$ is stored for later table doubling.
- When the table doubles, $\$1$ pays to move a recent item, and $\$1$ pays to move an old item.

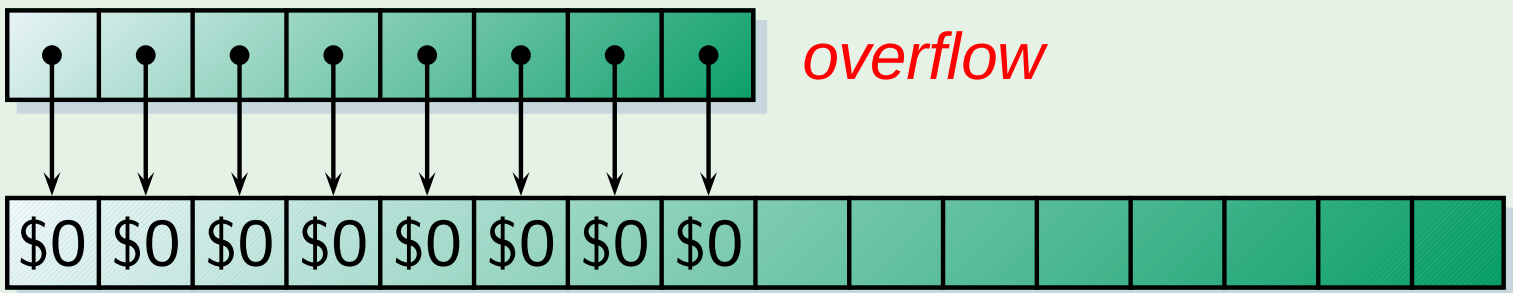
Example



Accounting analysis of dynamic tables

- Charge an amortized cost of $\hat{c}_i = \$3$ for the ith insertion.
  - $\$1$ pays for the immediate insertion.
  - $\$2$ is stored for later table doubling.
- When the table doubles, $\$1$ pays to move a recent item, and $\$1$ pays to move an old item.

Example



Accounting analysis of dynamic tables

- Charge an amortized cost of $\hat{c}_i = \$3$ for the ith insertion.
  - $\$1$ pays for the immediate insertion.
  - $\$2$ is stored for later table doubling.
- When the table doubles, $\$1$ pays to move a recent item, and $\$1$ pays to move an old item.

Example

Accounting analysis of dynamic tables

Table: Credit never goes negative. Thus, the sum of the amortized costs provides an **upper bound** on the sum of the true costs.

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
$\hat{c}_i$										
$Credit_i$										

Accounting analysis of dynamic tables

Table: Credit never goes negative. Thus, the sum of the amortized costs provides an **upper bound** on the sum of the true costs.

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
$\hat{c}_i$	2	3	3	3	3	3	3	3	3	3
$Credit_i$										

Accounting analysis of dynamic tables

Table: Credit never goes negative. Thus, the sum of the amortized costs provides an **upper bound** on the sum of the true costs.

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
$\hat{c}_i$	2	3	3	3	3	3	3	3	3	3
$Credit_i$	1	2	2	4						

Accounting analysis of dynamic tables

Table: Credit never goes negative. Thus, the sum of the amortized costs provides an **upper bound** on the sum of the true costs.

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
$\hat{c}_i$	2	3	3	3	3	3	3	3	3	3
$Credit_i$	1	2	2	4	2	4	6	8		

Accounting analysis of dynamic tables

Table: Credit never goes negative. Thus, the sum of the amortized costs provides an **upper bound** on the sum of the true costs.

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
$\hat{c}_i$	2	3	3	3	3	3	3	3	3	3
$Credit_i$	1	2	2	4	2	4	6	8	2	4

Incrementing a binary counter

- Charge an amortized cost of $\hat{c}_i = \$2$ to set a bit to 1.
  - Use \$1 to pay for the actual setting of the bit.
  - Place the other dollar on the bit as credit to be used later when we flip the bit back to 0.
- At any point in time, every 1 in the counter has a dollar of credit on it, and thus we needn't charge anything to reset a bit to 0.

Incrementing a binary counter

- Charge an amortized cost of $\hat{c}_i = \$2$ to set a bit to 1.
  - Use \$1 to pay for the actual setting of the bit.
  - Place the other dollar on the bit as credit to be used later when we flip the bit back to 0.
- At any point in time, every 1 in the counter has a dollar of credit on it, and thus we needn't charge anything to reset a bit to 0.

Incrementing a binary counter

- Charge an amortized cost of $\hat{c}_i = \$2$ to set a bit to 1.
  - Use \$1 to pay for the actual setting of the bit.
  - Place the other dollar on the bit as credit to be used later when we flip the bit back to 0.
- At any point in time, every 1 in the counter has a dollar of credit on it, and thus we needn't charge anything to reset a bit to 0.

Incrementing a binary counter

- Charge an amortized cost of $\hat{c}_i = \$2$ to set a bit to 1.
  - Use $\$1$ to pay for the actual setting of the bit.
  - Place the other dollar on the bit as credit to be used later when we flip the bit back to 0.
- At any point in time, every 1 in the counter has a dollar of credit on it, and thus we needn't charge anything to reset a bit to 0.

Incrementing a binary counter

- The amortized cost of an INCREMENT operation is at most 2 dollars. ◂ INCREMENT
- The number of 1's in the counter is never negative, and thus the amount of credit is always nonnegative.
- Thus, for n INCREMENT operations, the total amortized cost is $O(n)$.

Incrementing a binary counter

- The amortized cost of an INCREMENT operation is at most 2 dollars. ◂ INCREMENT
- The number of 1's in the counter is never negative, and thus the amount of credit is always nonnegative.
- Thus, for n INCREMENT operations, the total amortized cost is $O(n)$.

Incrementing a binary counter

- The amortized cost of an INCREMENT operation is at most 2 dollars. ◂ INCREMENT
- The number of 1's in the counter is never negative, and thus the amount of credit is always nonnegative.
- Thus, for n INCREMENT operations, the total amortized cost is $O(n)$.

Potential method

Idea

View credit stored as the **potential energy** of the dynamic set.

Framework

1. Start with an initial data structure D_0.
2. Operation i transforms D_{i-1} to D_i.
3. The cost of operation i is c_i.

Potential method

Idea

View credit stored as the **potential energy** of the dynamic set.

Framework

1. Start with an initial data structure D_0.
2. Operation i transforms D_{i-1} to D_i.
3. The cost of operation i is c_i.

Potential method

Idea

View credit stored as the **potential energy** of the dynamic set.

Framework

4. Define a **potential function** $\Phi : \{D_i\} \to \mathbb{R}$, such that $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i.
5. The **amortized cost** $\hat{c}_i$ with respect to Φ is defined to be $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

Potential method

Understanding potentials

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

- **Potential difference**
  $\Delta\Phi_i = \Phi(D_i) - \Phi(D_{i-1})$.
- If $\Delta\Phi_i > 0$, then $\hat{c}_i > c_i$. Operation i stores work in the data structure for later use.
- If $\Delta\Phi_i < 0$, then $\hat{c}_i < c_i$. The data structure delivers up stored work to help pay for operation i.

Potential method

Understanding potentials

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

- **Potential difference**
  $\Delta\Phi_i = \Phi(D_i) - \Phi(D_{i-1})$.
- If $\Delta\Phi_i > 0$, then $\hat{c}_i > c_i$. Operation i stores work in the data structure for later use.
- If $\Delta\Phi_i < 0$, then $\hat{c}_i < c_i$. The data structure delivers up stored work to help pay for operation i.

Potential method

Understanding potentials

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

- **Potential difference**
  $\Delta\Phi_i = \Phi(D_i) - \Phi(D_{i-1})$.
- If $\Delta\Phi_i > 0$, then $\hat{c}_i > c_i$. Operation i stores work in the data structure for later use.
- If $\Delta\Phi_i < 0$, then $\hat{c}_i < c_i$. The data structure delivers up stored work to help pay for operation i.

Potential method

The amortized costs bound the true costs

The total amortized cost of n operations is

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

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

Potential method

The amortized costs bound the true costs

The total amortized cost of n operations is

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

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

Potential method

The amortized costs bound the true costs

The total amortized cost of n operations is

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

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

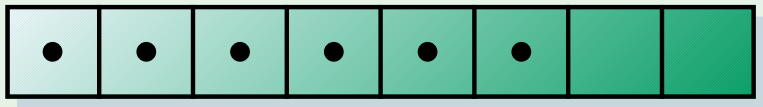
$$\geq \sum_{i=1}^{n} c_i$$

Potential analysis of dynamic table

Define the potential of the table after the ith insertion by $\Phi(D_i) = 2 \cdot num_i - size_i$.

- So $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$, for all i.

Example



$$\Phi = 2 \cdot 6 - 8 = 4$$

Potential analysis of dynamic table

Case 1: $i-1$ is not an exact power of 2.

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) \\ &\quad - (2 \cdot num_{i-1} - size_{i-1})\end{aligned}$$

Potential analysis of dynamic table

Case 1: $i-1$ is not an exact power of 2.

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) \\ &\quad - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) \\ &\quad - (2(num_i - 1) - size_i)\end{aligned}$$

Potential analysis of dynamic table

Case 1: $i-1$ is not an exact power of 2.

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) \\ &\qquad - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) \\ &\qquad - (2(num_i - 1) - size_i) \\ &= 3\end{aligned}$$

Potential analysis of dynamic table

Case 2: $i - 1$ is an exact power of 2.

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

Potential analysis of dynamic table

Case 2: $i - 1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= num_i + (2 \cdot num_i - size_i)) \\ &\quad - (2 \cdot num_{i-1} - size_{i-1})\end{aligned}$$

Potential analysis of dynamic table

Case 2: $i-1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= num_i + (2 \cdot num_i - size_i)) \\ &\qquad - (2 \cdot num_{i-1} - size_{i-1}) \\ &= num_i + (2 \cdot num_i - 2(num_i - 1)) \\ &\qquad - (2(num_i - 1) - (num_i - 1))\end{aligned}$$

Potential analysis of dynamic table

Case 2: $i-1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= num_i + (2 \cdot num_i - size_i)) \\ &\qquad - (2 \cdot num_{i-1} - size_{i-1}) \\ &= num_i + (2 \cdot num_i - 2(num_i - 1)) \\ &\qquad - (2(num_i - 1) - (num_i - 1)) \\ &= num_i + 2 - (num_i - 1)\end{aligned}$$

Potential analysis of dynamic table

Case 2: $i-1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= num_i + (2 \cdot num_i - size_i)) \\ &\qquad - (2 \cdot num_{i-1} - size_{i-1}) \\ &= num_i + (2 \cdot num_i - 2(num_i - 1)) \\ &\qquad - (2(num_i - 1) - (num_i - 1)) \\ &= num_i + 2 - (num_i - 1) \\ &= 3\end{aligned}$$

Therefore, n insertions cost $O(n)$ in the worst case.

Table contraction

When to contract?

- When $\alpha(T) = num[T]/size[T] < 1/2$?
- $\alpha(T) < 1/4$ is a good choice.

Potential function

$$\Phi(T) = \begin{cases} 2 \cdot num[T] - size[T] & \text{if } \alpha(T) \geq 1/2, \\ size[T]/2 - num[T] & \text{if } \alpha(T) < 1/2. \end{cases}$$

Table contraction

When to contract?

- When $\alpha(T) = num[T]/size[T] < 1/2$?
- $\alpha(T) < 1/4$ is a good choice.

Potential function

$$\Phi(T) = \begin{cases} 2 \cdot num[T] - size[T] & \text{if } \alpha(T) \geq 1/2, \\ size[T]/2 - num[T] & \text{if } \alpha(T) < 1/2. \end{cases}$$

Table contraction

When to contract?

- When $\alpha(T) = num[T]/size[T] < 1/2$?
- $\alpha(T) < 1/4$ is a good choice.

Potential function

$$\Phi(T) = \begin{cases} 2 \cdot num[T] - size[T] & \text{if } \alpha(T) \geq 1/2, \\ size[T]/2 - num[T] & \text{if } \alpha(T) < 1/2. \end{cases}$$

Table contraction

Analysis of TABLE-INSERT when $\alpha_i < 1/2$

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (size_i/2 - num_i) \\ &\quad - (size_{i-1}/2 - num_{i-1})\end{aligned}$$

Table contraction

Analysis of TABLE-INSERT when $\alpha_i < 1/2$

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (size_i/2 - num_i) \\ &\quad - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) \\ &\quad - (size_i/2 - (num_i - 1))\end{aligned}$$

Table contraction

Analysis of TABLE-INSERT when $\alpha_i < 1/2$

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (size_i/2 - num_i) \\
&\quad - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (size_i/2 - num_i) \\
&\quad - (size_i/2 - (num_i - 1)) \\
&= 0.
\end{aligned}$$

Table contraction

Analysis of TABLE-INSERT when $\alpha_{i-1} < 1/2$ but $\alpha_i \geq 1/2$

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) \\ &\quad - (size_{i-1}/2 - num_{i-1})\end{aligned}$$

Table contraction

Analysis of TABLE-INSERT when $\alpha_{i-1} < 1/2$ but $\alpha_i \geq 1/2$

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot num_i - size_i) \\
&\quad - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (2(num_{i-1} + 1) - size_{i-1}) \\
&\quad - (size_{i-1}/2 - num_{i-1})
\end{aligned}$$

Table contraction

Analysis of TABLE-INSERT when $\alpha_{i-1} < 1/2$ but $\alpha_i \geq 1/2$

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot num_i - size_i) \\
&\quad - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (2(num_{i-1} + 1) - size_{i-1}) \\
&\quad - (size_{i-1}/2 - num_{i-1}) \\
&= 3 \cdot num_{i-1} - \frac{3}{2} size_{i-1} + 3
\end{aligned}$$

Table contraction

Analysis of TABLE-INSERT when $\alpha_{i-1} < 1/2$ but $\alpha_i \geq 1/2$

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 3\alpha_{i-1} size_{i-1} - \frac{3}{2} size_{i-1} + 3\end{aligned}$$

Table contraction

Analysis of TABLE-INSERT when $\alpha_{i-1} < 1/2$ but $\alpha_i \geq 1/2$

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 3\alpha_{i-1} size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&< \frac{3}{2} size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\
&= 3.
\end{aligned}$$

Table contraction

Analysis of TABLE-DELETE when $\alpha_{i-1} < 1/2, \alpha_i \geq 1/4$

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (size_i/2 - num_i) \\ &\quad - (size_{i-1}/2 - num_{i-1})\end{aligned}$$

Table contraction

Analysis of TABLE-DELETE when $\alpha_{i-1} < 1/2, \alpha_i \geq 1/4$

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (size_i/2 - num_i) \\
&\quad - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (size_i/2 - num_i) \\
&\quad - (size_i/2 - (num_i + 1)) \\
&= 2.
\end{aligned}$$

Table contraction

Analysis of TABLE-DELETE when $\alpha_{i-1} < 1/2, \alpha_i < 1/4$

Due $size_i/2 = size_{i-1}/4 = num_{i-1} = num_i + 1$

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= (num_i + 1) + (size_i/2 - num_i) \\ &\quad - (size_{i-1}/2 - num_{i-1})\end{aligned}$$

Table contraction

Analysis of TABLE-DELETE when $\alpha_{i-1} < 1/2, \alpha_i < 1/4$

Due $size_i/2 = size_{i-1}/4 = num_{i-1} = num_i + 1$

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= (num_i + 1) + (size_i/2 - num_i) \\
&\quad - (size_{i-1}/2 - num_{i-1}) \\
&= (num_i + 1) + ((num_i + 1) - num_i) \\
&\quad - ((2 \cdot num_i + 2) - (num_i + 1))
\end{aligned}$$

Table contraction

Analysis of TABLE-DELETE when $\alpha_{i-1} < 1/2, \alpha_i < 1/4$

Due $size_i/2 = size_{i-1}/4 = num_{i-1} = num_i + 1$

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= (num_i + 1) + (size_i/2 - num_i) \\
&\qquad - (size_{i-1}/2 - num_{i-1}) \\
&= (num_i + 1) + ((num_i + 1) - num_i) \\
&\qquad - ((2 \cdot num_i + 2) - (num_i + 1)) \\
&= 1.
\end{aligned}$$

Incrementing a binary counter

- We define the ***potential*** of the counter after the ith INCREMENT operation to be b_i, the number of 1's in the counter after the ith operation. See the table

Incrementing a binary counter

- We define the ***potential*** of the counter after the ith INCREMENT operation to be b_i, the number of 1's in the counter after the ith operation. See the table
- The potential difference is

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &\leq (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i .\end{aligned}$$

t_i reprents the reseted bits by the ith INCREMENT operation.

Incrementing a binary counter

- The amortized cost is therefore

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2\end{aligned}$$

Structure of Fibonacci heaps

Definition

- A ***Fibonacci heap*** is a collection of ***min-heap-ordered trees***.
- Trees within Fibonacci heaps are rooted but ***unordered***.
- The children of x are linked together in a circular, doubly linked list, which we call the ***child list*** of x.

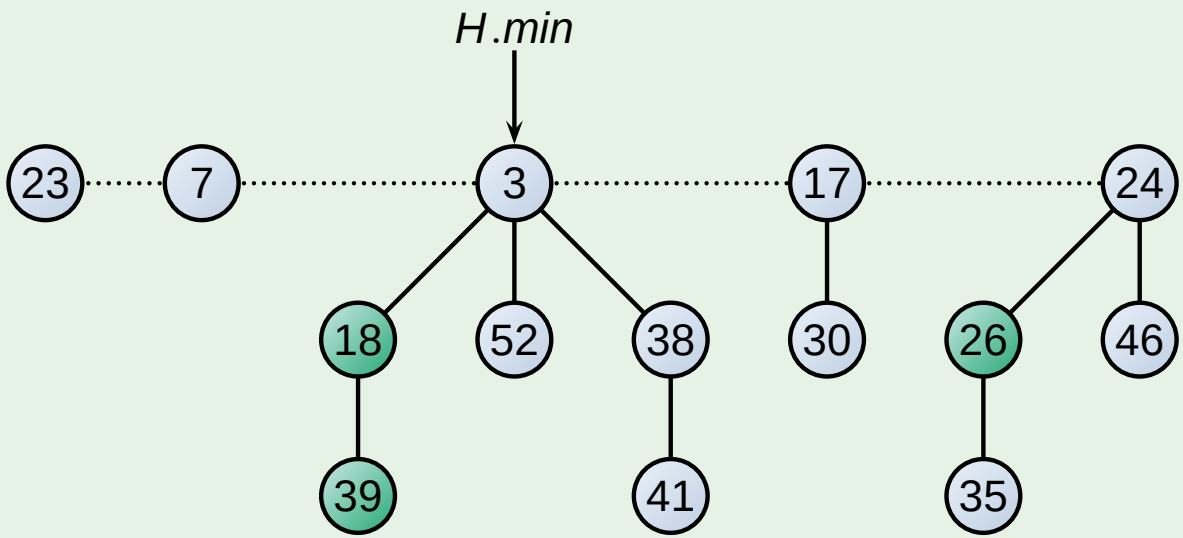
Structure of Fibonacci heaps

Definition

- A given Fibonacci heap H is accessed by a pointer $H.min$ to the root of a tree containing a minimum key; this node is called the ***minimum node*** of the Fibonacci heap.
- The roots of all the trees in a Fibonacci heap are linked together using their left and right pointers into a circular, doubly linked list called the ***root list*** of the Fibonacci heap.

Structure of Fibonacci heaps

Example



Structure of Fibonacci heaps

Definition

- The number of children in the child list of node x is stored in $x.degree$. The boolean-valued field $x.mark$ indicates whether node x has lost a child since the last time x was made the child of another node.
- The number of nodes currently in H is kept in $H.n$.

Structure of Fibonacci heaps

Potential function

For a given Fibonacci heap H, we indicate by $t(H)$ the number of trees in the root list of H and by $m(H)$ the number of marked nodes in H. The potential of Fibonacci heap H is then defined by

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

Structure of Fibonacci heaps

Maximum degree

We assume that there is a known upper bound $D(n)$ on the maximum degree of any node in an n-node Fibonacci heap.

Mergeable-heap operations

Creating a new Fibonacci heap

MAKE-FIB-HEAP()

```
1  allocate an object H
2  H.min = NIL
3  H.n = 0
4  return H
```

Amortized cost :
$O(1)$

Mergeable-heap operations

Inserting a node

FIB-HEAP-INSERT(H, x)

```
1  x.degree = 0, x.p = NIL
2  x.child = NIL, x.mark = FALSE
3  if H.min == NIL
4        create a root list for H containing just x
5        H.min = x
6  else insert x into H's root list
7        if x.key < H.min.key
8              H.min = x
9  H.n = H.n + 1
```

Mergeable-heap operations

Inserting a node

Potential difference :

$$((t(H)+1)+2m(H))-(t(H)+2m(H))=1$$

Amortized cost : $O(1)$

Finding the minimum node

The minimum node of a Fibonacci heap H is given by the pointer $H.min$, so we can find the minimum node in $O(1)$ actual time.

Mergeable-heap operations

Inserting a node

Potential difference :

$$((t(H)+1)+2m(H))-(t(H)+2m(H))=1$$

Amortized cost : $O(1)$

Finding the minimum node

The minimum node of a Fibonacci heap H is given by the pointer $H.min$, so we can find the minimum node in $O(1)$ actual time.

Mergeable-heap operations

Uniting two Fibonacci heaps

FIB-HEAP-UNION(H_1, H_2)

1 H = MAKE-FIB-HEAP()
2 $H.min = H_1.min$
3 concatenate the root lists of H_2 and H
4 **if** ($H_1.min$ == NIL) or
($H_2.min \neq$ NIL and $H_2.min.key < H_1.min.key$)
5 $H.min = H_2.min$
6 $H.n = H_1.n + H_2.n$
7 **return** H

Mergeable-heap operations

Uniting two Fibonacci heaps

Potential difference :

$$\begin{aligned}\Phi(H) - (\Phi(H_1) + \Phi(H_2)) &= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) \\ &\quad + (t(H_2) + 2m(H_2))) \\ &= 0.\end{aligned}$$

Amortized cost : $O(1)$

Mergeable-heap operations

FIB-HEAP-EXTRACT-MIN(H)

```
 1  z = H.min
 2  if z ≠ NIL
 3        for each child x of z
 4              add x to the root list of H
 5              x.p = NIL
 6        remove z from the root list of H
 7        if z == z.right
 8              H.min = NIL
 9        else H.min = z.right
10              CONSOLIDATE(H)
11        H.n = H.n − 1
12  return z
```

Mergeable-heap operations

CONSOLIDATE(H)

```
1   let A[0..D(H.n)] be a new array
2   for i = 0 to D(H.n)
3       A[i] = NIL
4   for each node w in the root list of H
5       x = w, d = x.degree
6       while A[d] ≠ NIL
7           y = A[d] // a same degree node
8           if x.key > y.key
9               exchange x with y
10          FIB-HEAP-LINK(H, y, x)
11          A[d] = NIL
12          d = d + 1
```

Mergeable-heap operations

```
13          A[d] = x
14    H.min = NIL
15    for i = 0 to D(H.n)
16          if A[i] ≠ NIL
17                if H.min == NIL
18                      create a root list for H
19                      H.min = A[i]
20                else insert A[i] to H's root list
21                      if A[i].key < H.min.key
22                            H.min = A[i]
```

Mergeable-heap operations

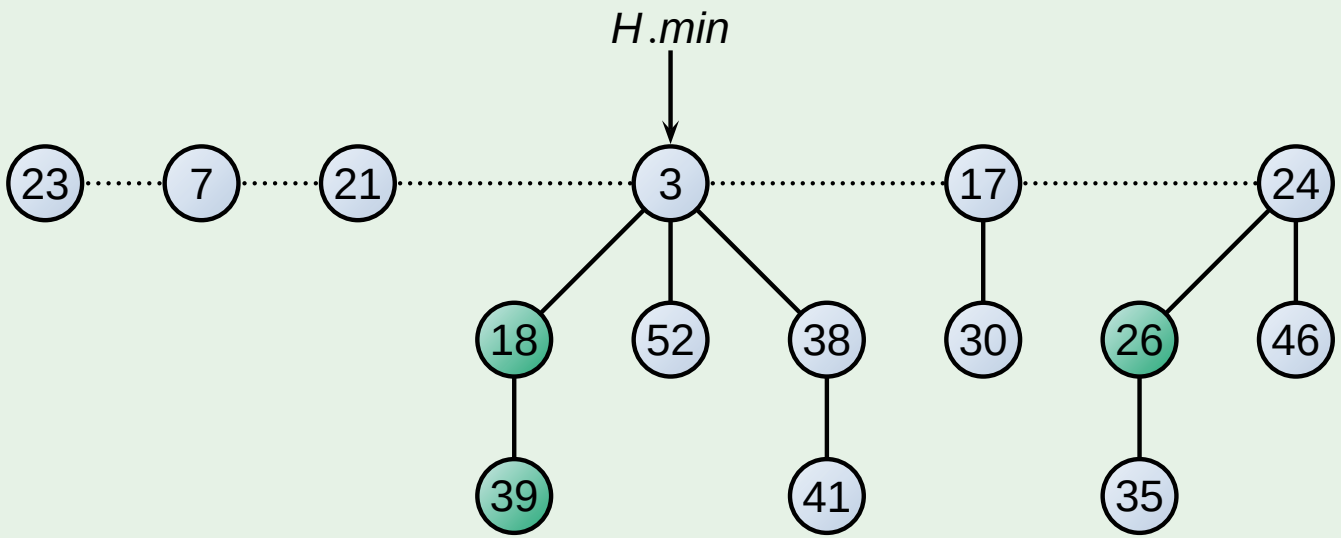
FIB-HEAP-LINK(H, y, x)

1 remove y from the root list of H
2 make y a child of x, incrementing $x.degree$
3 $y.mark = \text{FALSE}$

Skip example

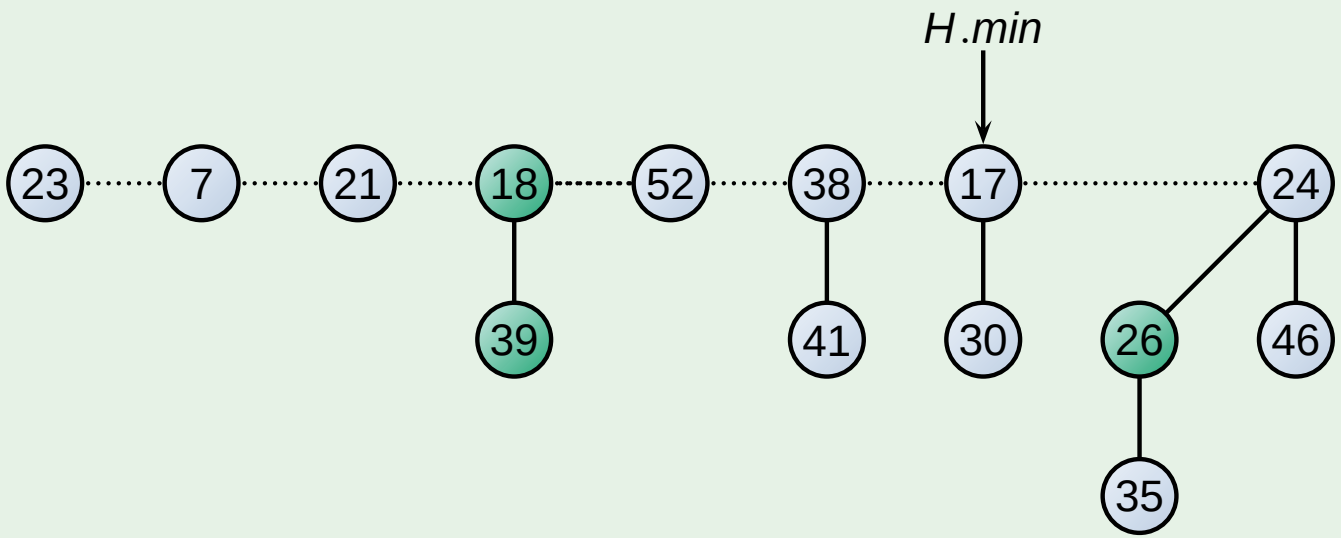
Mergeable-heap operations

Example



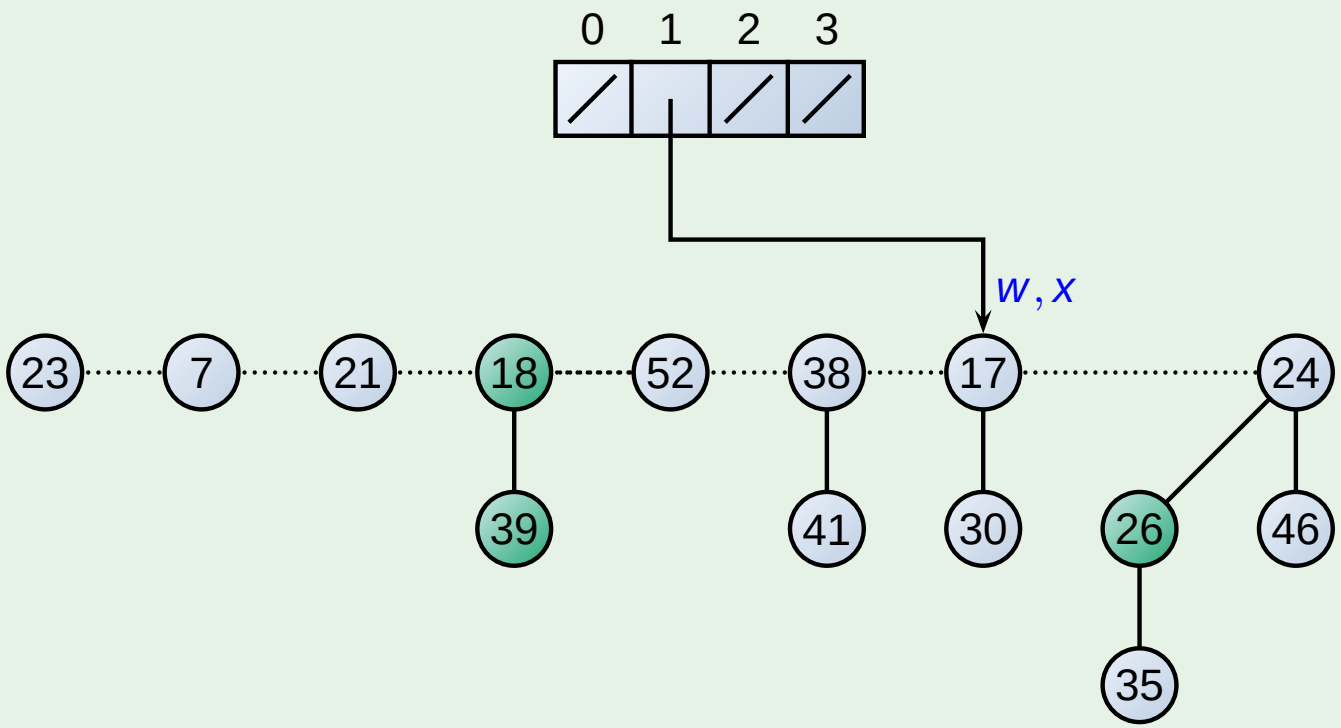
Mergeable-heap operations

Example



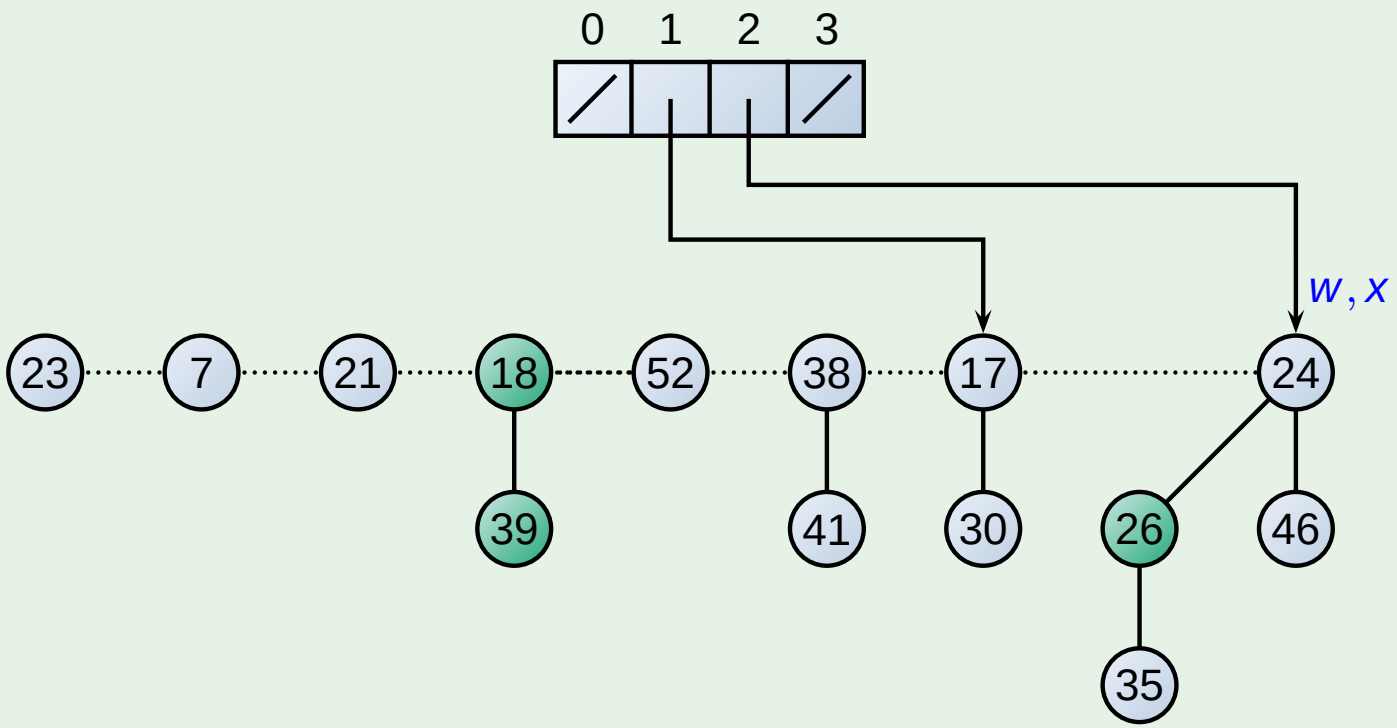
Mergeable-heap operations

Example



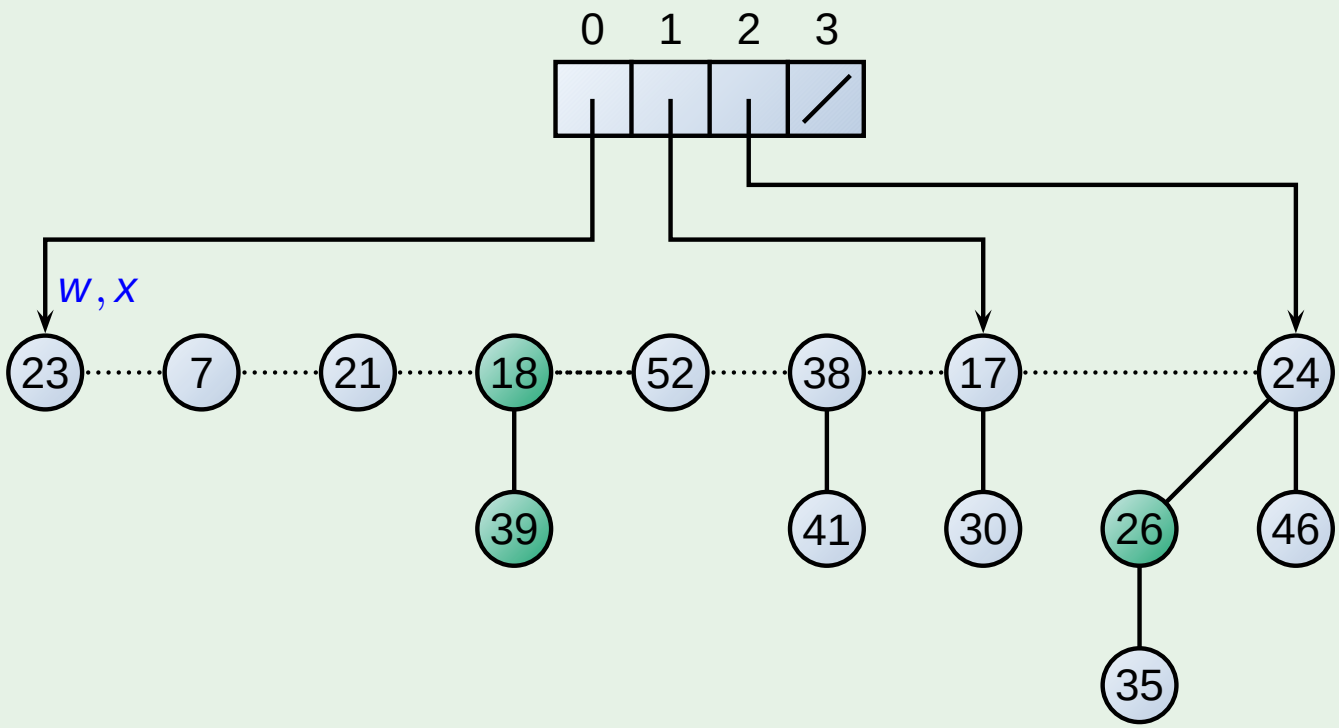
Mergeable-heap operations

Example



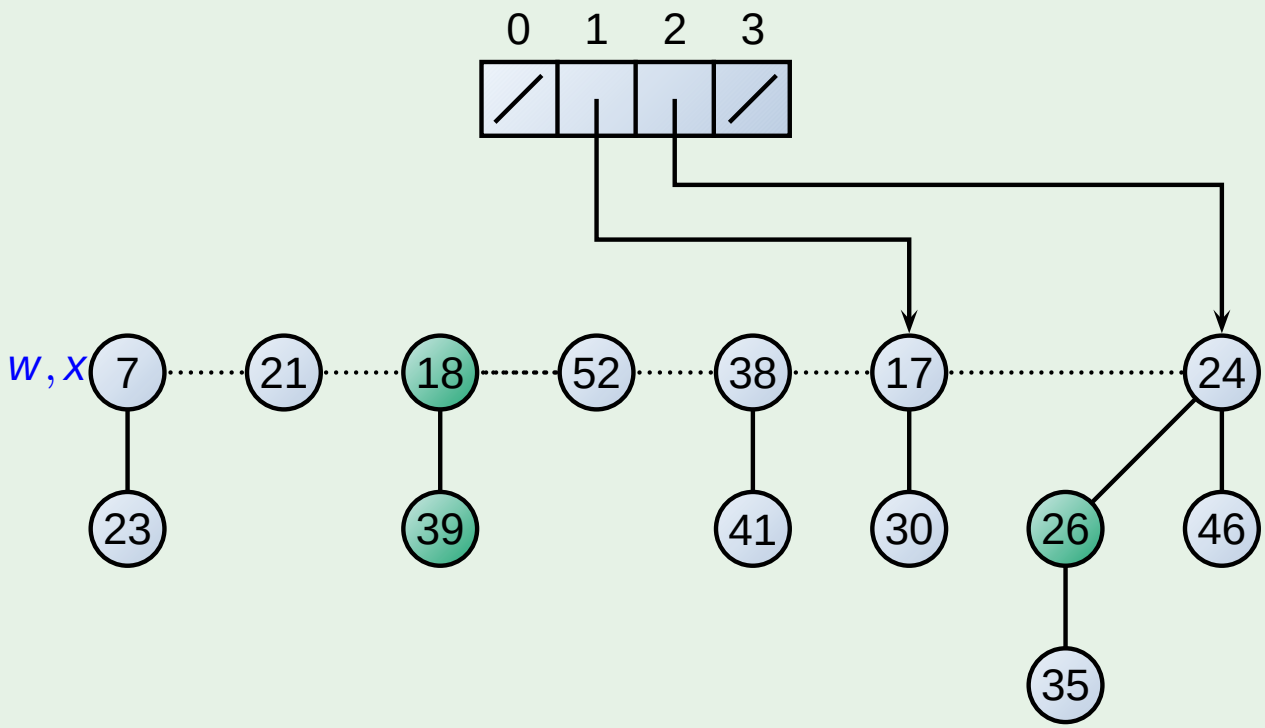
Mergeable-heap operations

Example



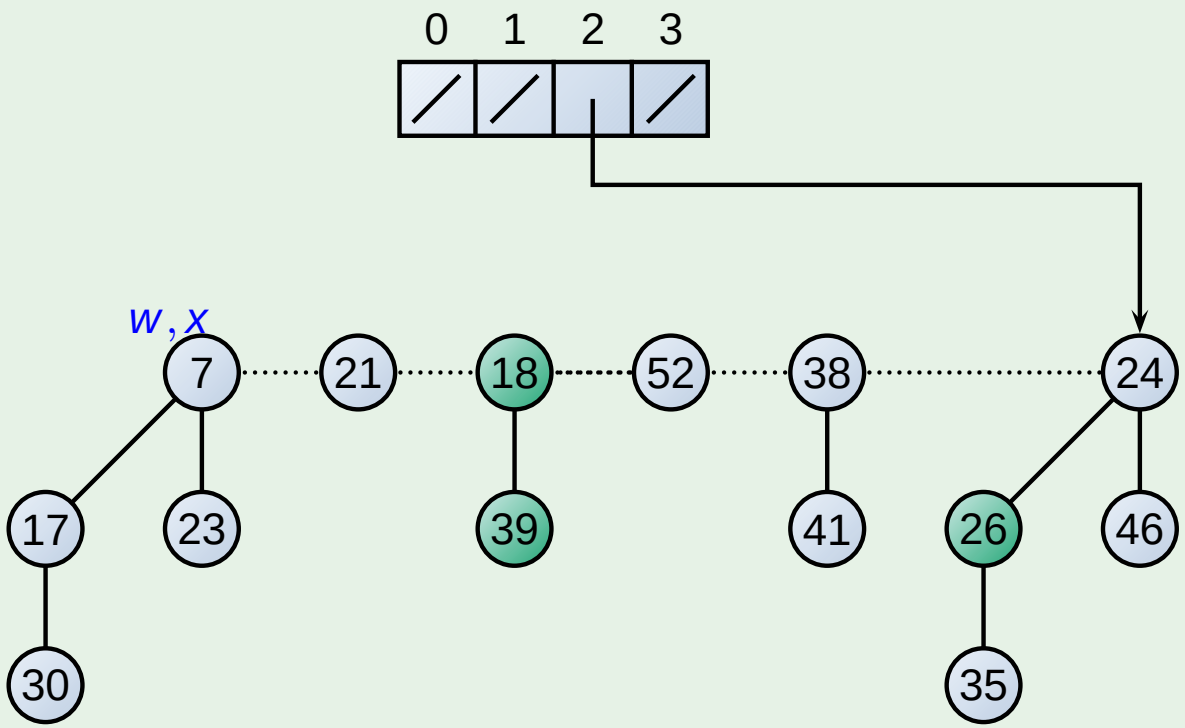
Mergeable-heap operations

Example



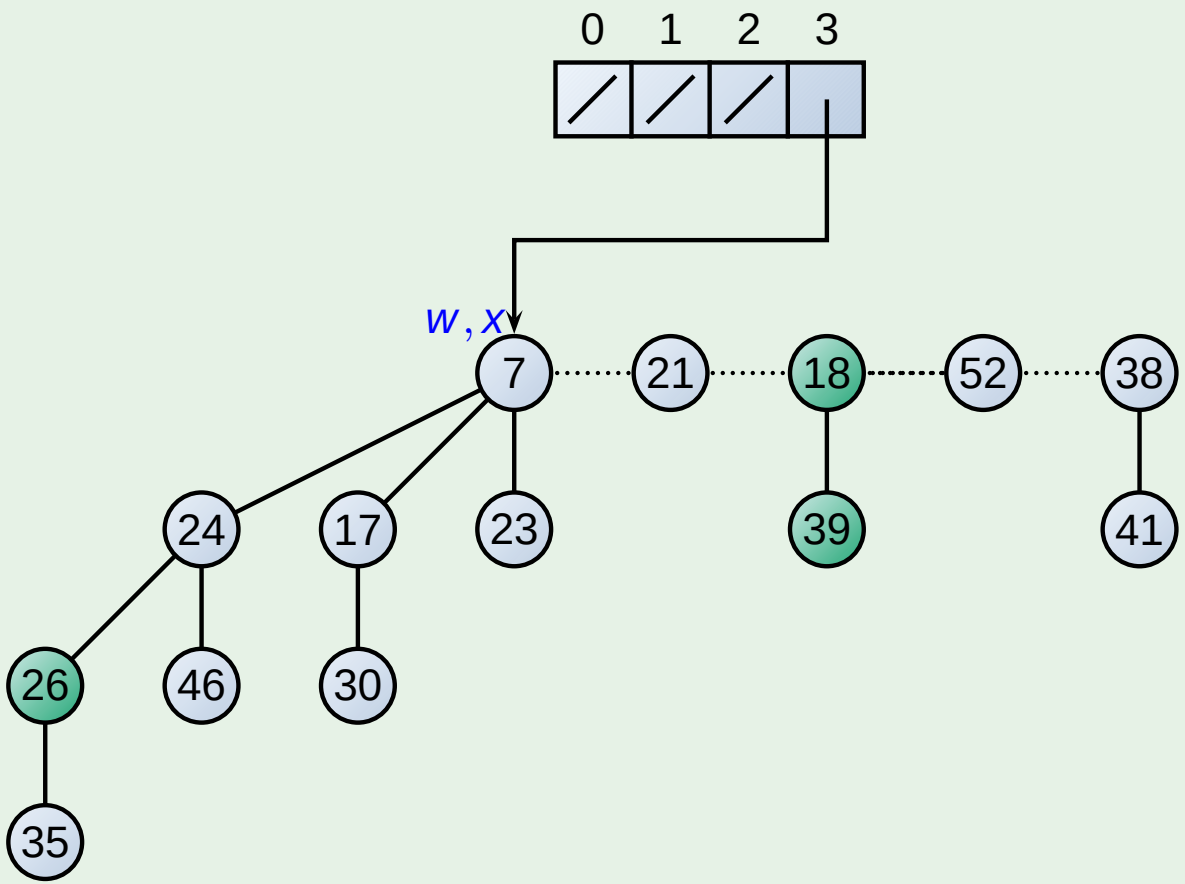
Mergeable-heap operations

Example



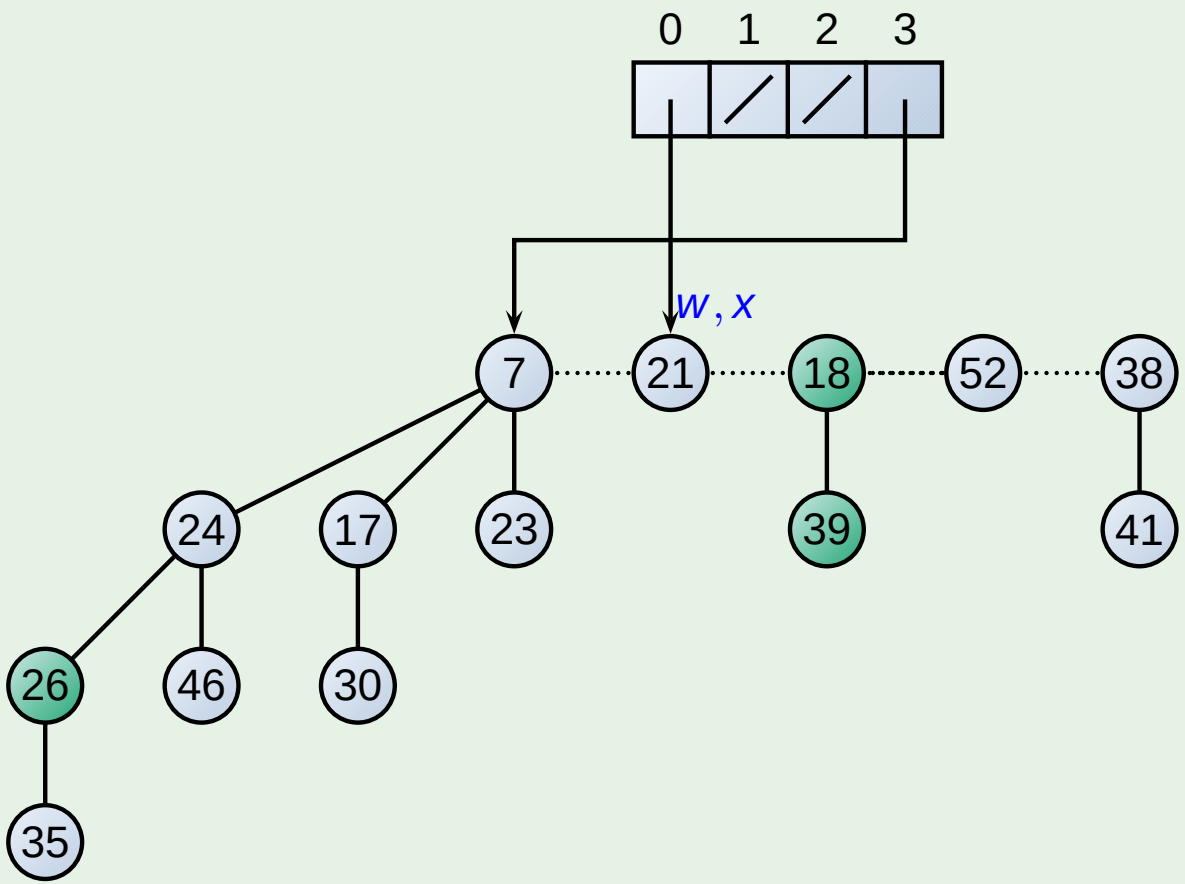
Mergeable-heap operations

Example



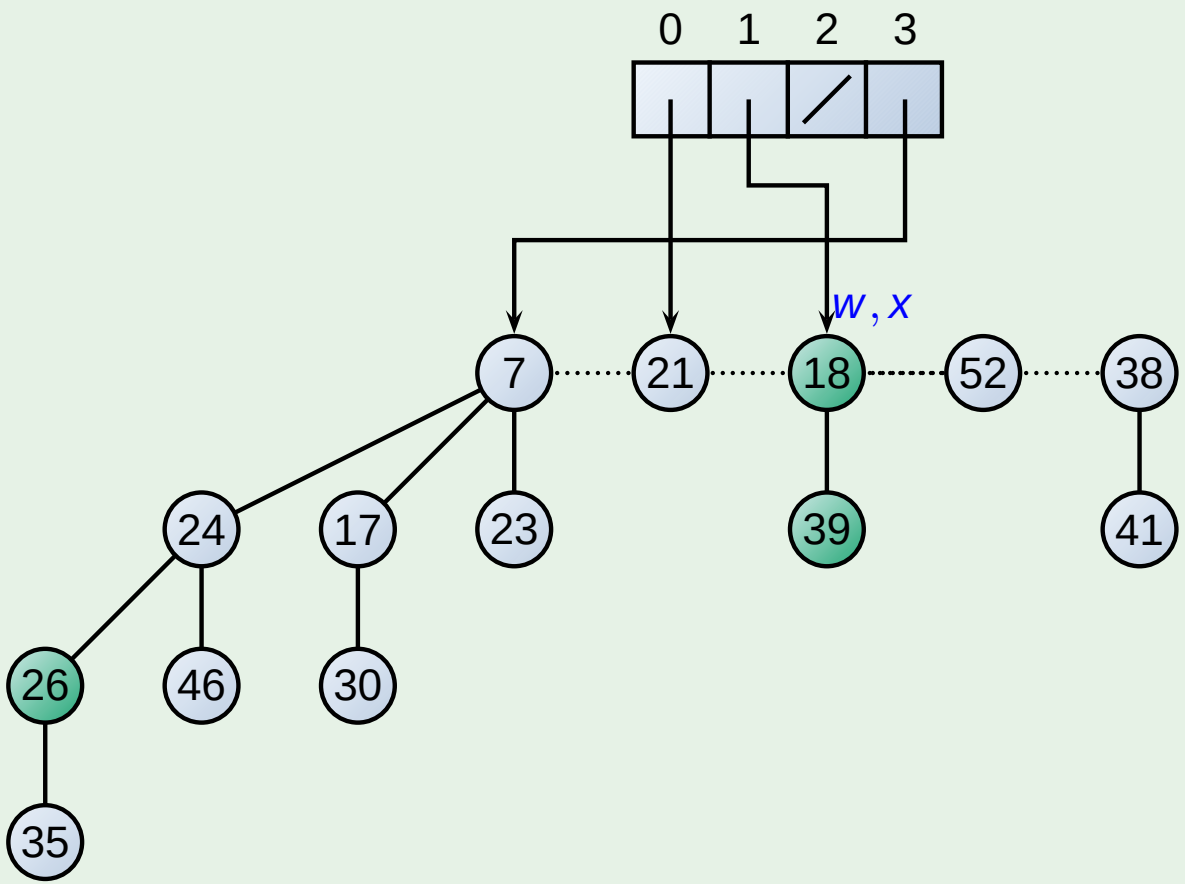
Mergeable-heap operations

Example



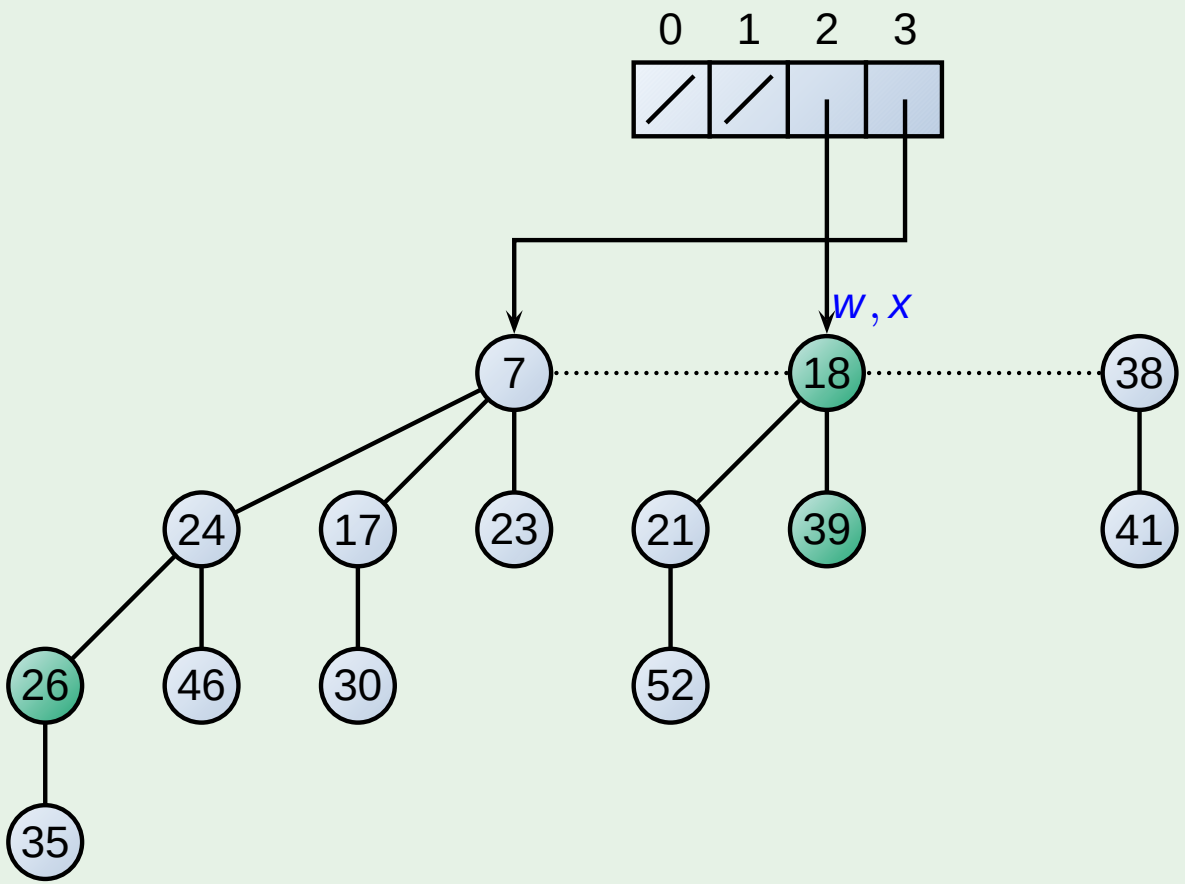
Mergeable-heap operations

Example



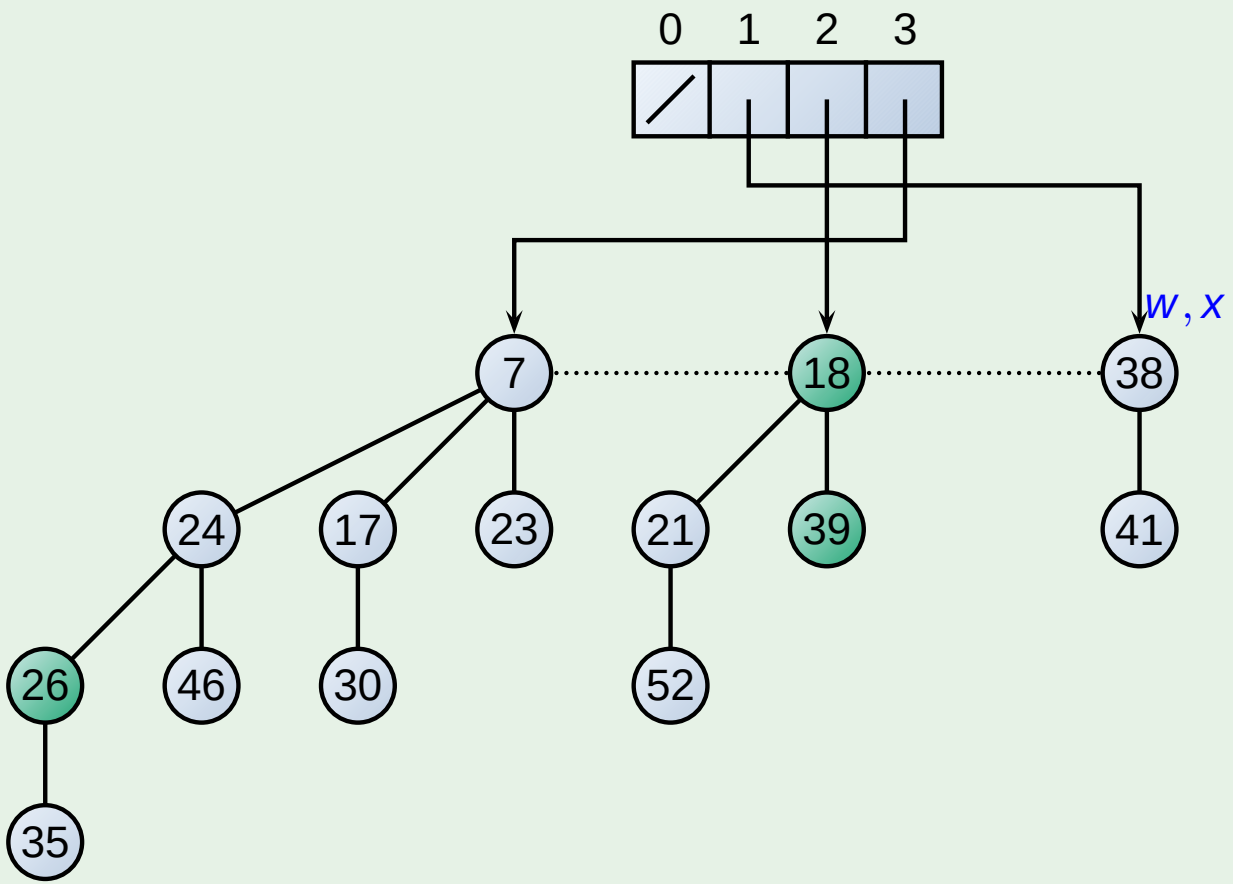
Mergeable-heap operations

Example



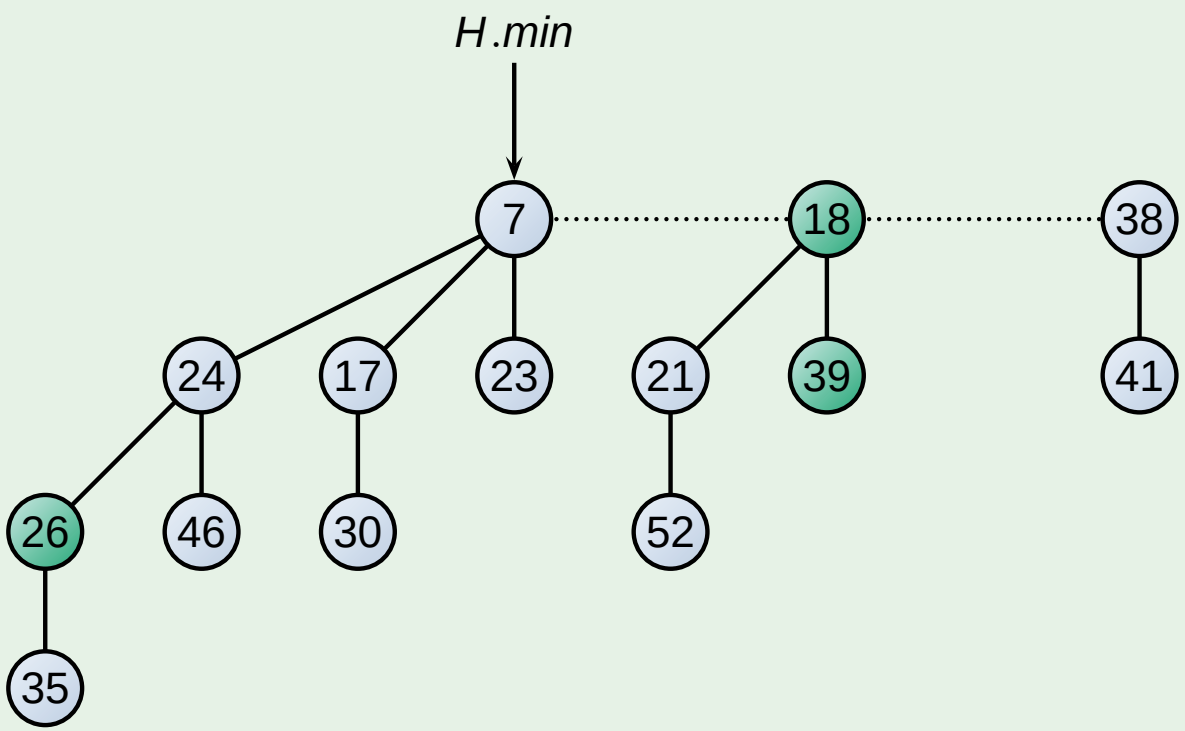
Mergeable-heap operations

Example



Mergeable-heap operations

Example



Mergeable-heap operations

Amortized cost

$$\begin{aligned}\hat{c} &= c + \Delta\phi \\ &= O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) \\ &\qquad - (t(H) + 2m(H)) \\ &= O(D(n)) + O(t(H)) - t(H) \\ &= O(D(n)).\end{aligned}$$

Decreasing a key and deleting a node

Decreasing a key

FIB-HEAP-DECREASE-KEY(H, x, k)

```
1  if k > x.key
2        error "new key is greater"
3  x.key = k
4  y = x.p
5  if y ≠ NIL and x.key < y.key
6        CUT(H, x, y)
7        CASCADING-CUT(H, y)
8  if x.key < H.min.key
9        H.min = x
```

Decreasing a key and deleting a node

Decreasing a key

CUT(H, x, y)

1 remove x from the child list of y, decrementing $y.degree$
2 add x to the root list of H
3 $x.p = \text{NIL}$
4 $x.mark = \text{FALSE}$

Decreasing a key and deleting a node

Decreasing a key

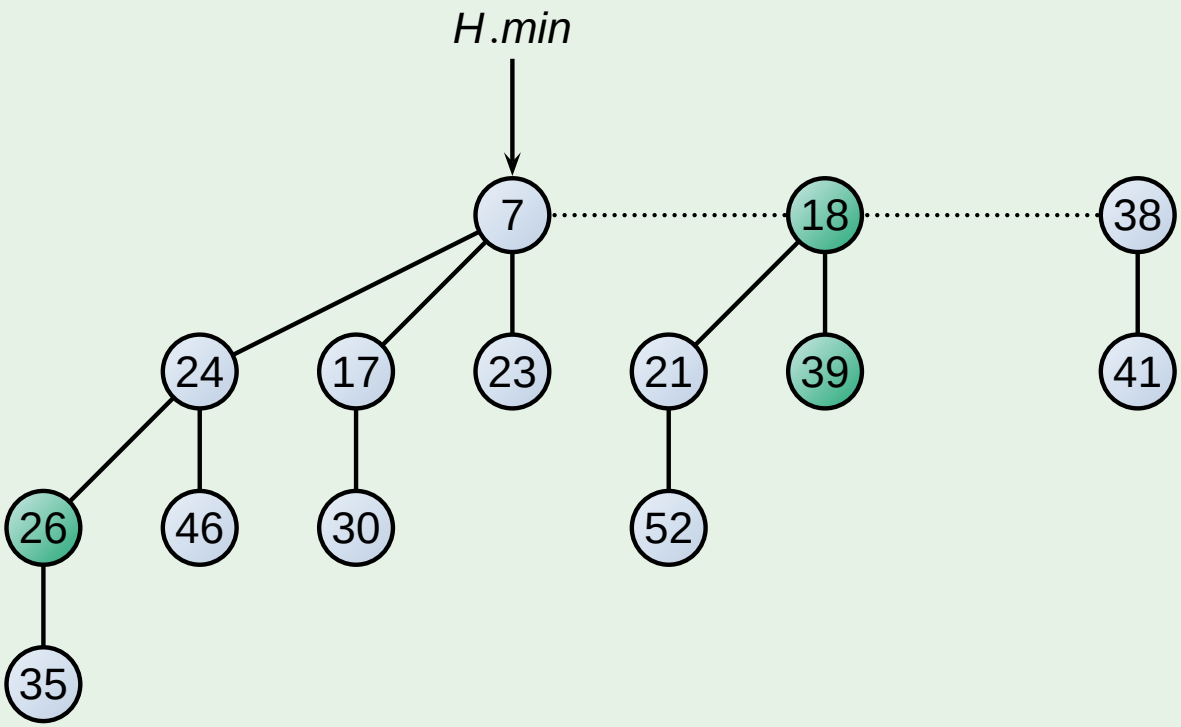
CASCADING-CUT(H, y)

```
1  z = y.p
2  if z ≠ NIL
3        if y.mark == FALSE
4              y.mark = TRUE
5        else CUT(H, y, z)
6              CASCADING-CUT(H, z)
```

Skip example

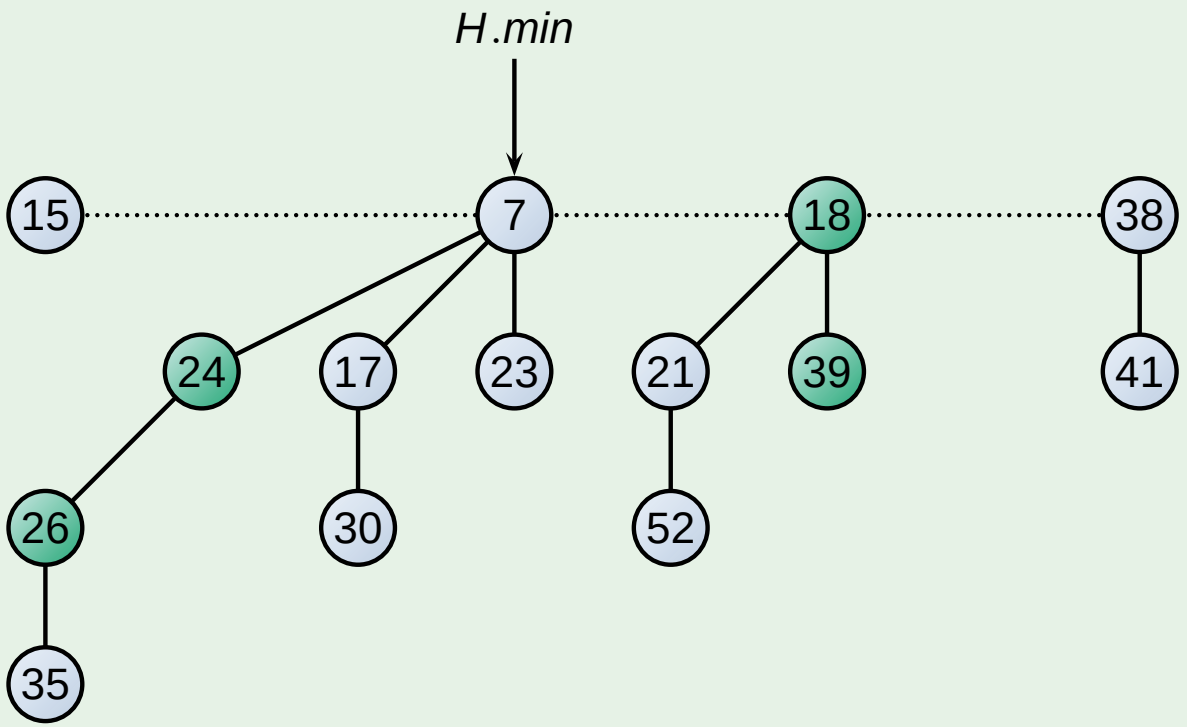
Decreasing a key and deleting a node

Example



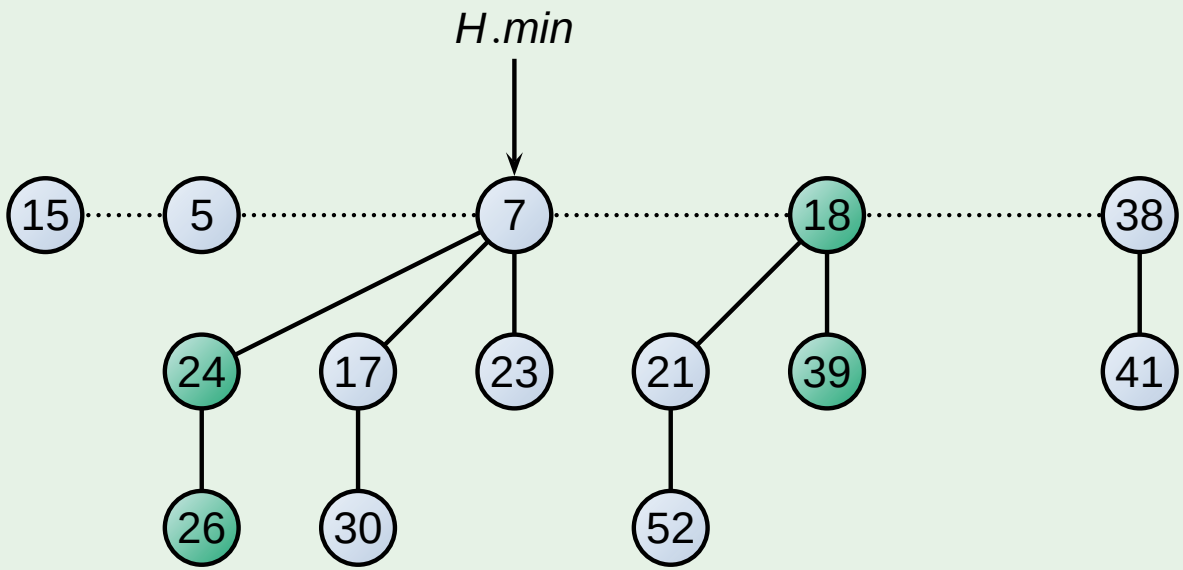
Decreasing a key and deleting a node

Example



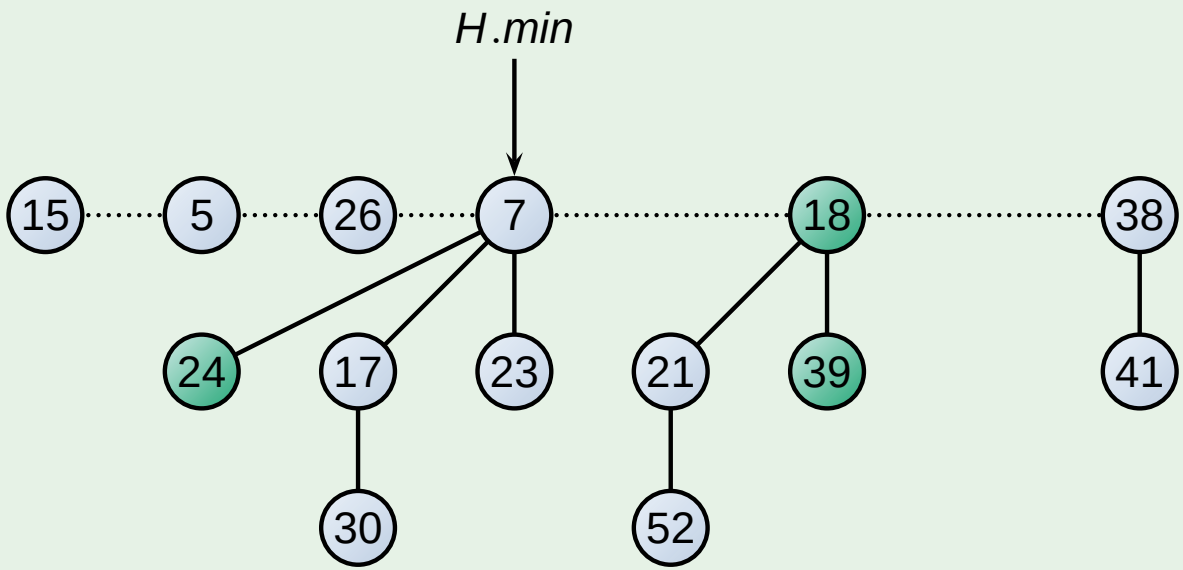
Decreasing a key and deleting a node

Example



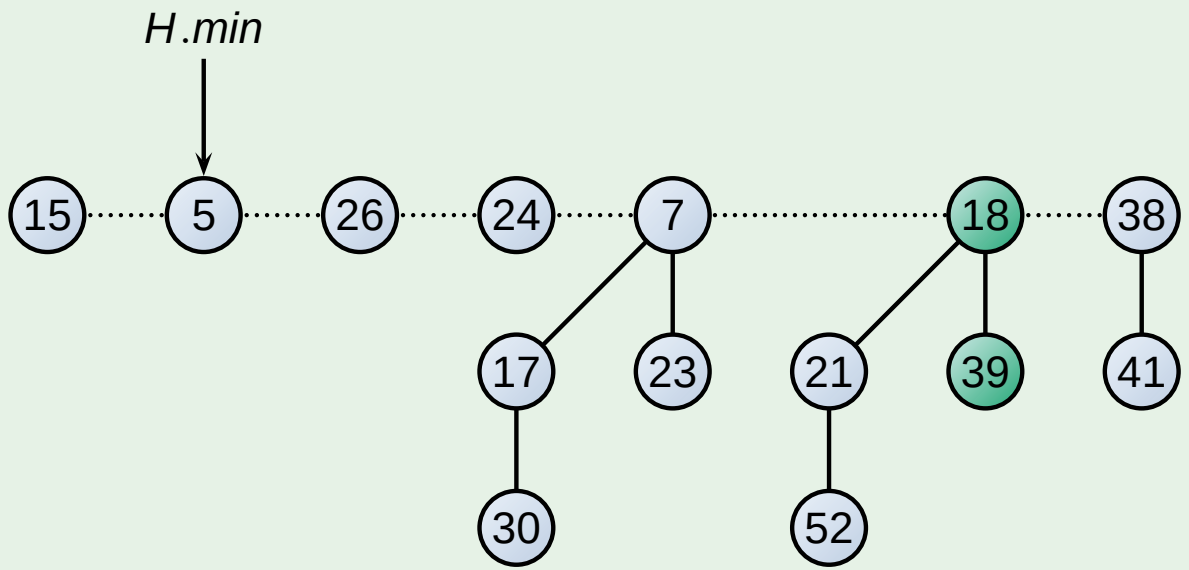
Decreasing a key and deleting a node

Example



Decreasing a key and deleting a node

Example



Decreasing a key and deleting a node

Amortized cost

Suppose that CASCADING-CUT is recursively called c times from a given invocation of FIB- HEAP-DECREASE-KEY.

$$\begin{aligned}\hat{c} &= c + \Delta\phi \\ &= O(c) + ((t(H) + c) + 2(m(H) - c + 2)) \\ &\qquad - (t(H) + 2m(H)) \\ &= O(c) + 4 - c \\ &= O(1).\end{aligned}$$

Decreasing a key and deleting a node

Deleting a node

FIB-HEAP-DELETE(H, x)

1 FIB-HEAP-DECREASE-KEY$(H, x, -\infty)$
2 FIB-HEAP-EXTRACT-MIN(H)

Bounding the maximum degree

Lemma 19.1

Let x be any node in a Fibonacci heap, and suppose that $x.degree = k$. Let $y_1, y_2, \ldots, y_k$ denote the children of x in the order in which they were linked to x, from the earliest to the latest. Then, $y_1.degree \geq 0$ and $y_i.degree \geq i - 2$ for $i = 2, 3, \ldots, k$.

Bounding the maximum degree

Proof.

- For $i \geq 2$, we note that when y_i was linked to x, all of $y_1, y_2, \ldots, y_{i-1}$ were children of x, so we must have had $x.degree = i - 1$.
- Node y_i is linked to x only if $x.degree = y_i.degree$, so we must have also had $y_i.degree = i - 1$ at that time.
- Since then, node y_i has lost at most one child, since it would have been cut from x if it had lost two children.

We conclude that $y_i.degree \geq i - 2$. □

Bounding the maximum degree

Lemma 19.2

$$F_k = \begin{cases} 0 & \text{if } k = 0, \\ 1 & \text{if } k = 1, \\ F_{k-1} + F_{k-2} & \text{if } k > 1. \end{cases}$$

For all integers $k \geq 0$,

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i .$$

Bounding the maximum degree

Proof.

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &= F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) \\ &= 1 + \sum_{i=0}^{k} F_i . \end{aligned}$$

Bounding the maximum degree

Lemma 19.3

For all intergers $k \geq 0$, the $(k+2)$nd Fibonacci number satisfies $F_{k+2} \geq \phi^k$.

Proof.

$$\begin{aligned} F_{k+2} &= F_{k+1} + F_k \\ &\geq \phi^{k-1} + \phi^{k-2} \\ &= \phi^{k-2}(\phi + 1) \\ &= \phi^{k-2} \cdot \phi^2 = \phi^k. \end{aligned}$$

□

Bounding the maximum degree

Lemma 19.4

Let x be any node in a Fibonacci heap, and let $k = x.degree$. Then, $size(x) \geq F_{k+2} \geq \phi^k$, where $\phi = (1 + \sqrt{5})/2$.

$size(x)$ is defined to be the number of nodes, including x, in the subtree ***rooted*** at x.

Bounding the maximum degree

Proof

Let s_k denote the ***minimum*** possible value of $size(z)$ over all nodes z such that $z.degree = k$. Trivially, $s_0 = 1$, $s_1 = 2$.

The number s_k is at most $size(x)$, and clearly, the value of s_k increases monotonically with k.

Bounding the maximum degree

Proof.

$$size(x) \geq s_k = 2 + \sum_{i=2}^{k} s_{y_i.degree}$$

$$\geq 2 + \sum_{i=2}^{k} s_{i-2}$$

Bounding the maximum degree

Proof.

$$size(x) \geq s_k = 2 + \sum_{i=2}^{k} s_{y_i.degree}$$

$$\geq 2 + \sum_{i=2}^{k} s_{i-2}$$

$$\geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2}$$

$$\geq \phi^k$$

Bounding the maximum degree

Corollary 19.5

The maximum degree $D(n)$ of any node in an n-node Fibonacci heap is $O(\lg n)$.

Proof.

Let x be any node in an n-node Fibonacci heap, and let $k = x.degree$. By Lemma 19.4, we have $n \geq size(x) \geq \phi^k$. Taking base $-\phi$ logarithms gives us $k \leq \log_\phi n$. The maximum degree $D(n)$ of any node is thus $O(\lg n)$. □

Conclusion

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

History

- In 1987, Michael L. Fredman and Robert E. Tarjan introduced Fibonacci heaps. Their paper also describes the application of Fibonacci heaps to the problems of single-source shortest paths, all-pairs shortest paths, weighted bipartite matching, and the minimum-spanning-tree problem.
- In 1988, Driscoll, Gabow, Shrairman, and Tarjan developed “relaxed heaps” as an alternative to Fibonacci heaps.