POSCAT Seminar 4 : Adv. Data Structure

yougatup @ POSCAT

Topic

- Topic today
  - Heap (Just for your knowledge)
    - D-ary heap
    - Binomial heap
    - Fibonacci heap
  - ~~Indexed Tree~~
    - ~~Binary Indexed Tree~~
    - ~~Fenwick Tree~~
  - ~~Implementation of Indexed Tree~~

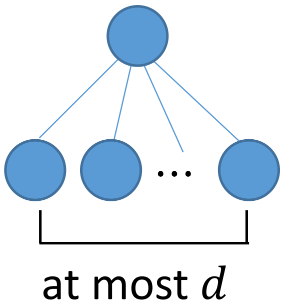
Poscat

Heap

- You already know what it is
  - Definition ?
- Amazingly, there is **faster** data structures
  - Fibonacci heap 은 이론적으로 Time complexity 가장 적음
  - But, 구현해보면 느림 → 실제로 쓰지는 않습니다
  - d-ary heap, Binomial heap, Fibonacci heap
- Operations
  - find_min, delete_min, insert, delete, decrease_key

d-ary heap

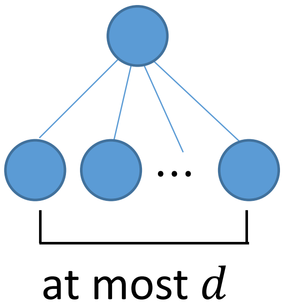
- It has child at most d
  - Extended version of binary heap
  - How it works ?
  - We use it for Prim algorithm



find_min ?
insert ?
delete ?
decrease_key ?
delete_min ?

d-ary heap

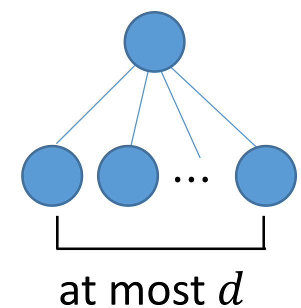
- It has child at most d
  - Extended version of binary heap
  - How it works ?
  - We use it for Prim algorithm



find_min	: root !
insert	: insert to right-most
delete	: decrease a value to -∞, and delete_min
decrease_key	: decrease a value, and rearrange heap
delete_min	: like binary heap ☺

d-ary heap

- It has child at most d
  - Extended version of binary heap
  - How it works ?
  - We use it for Prim algorithm

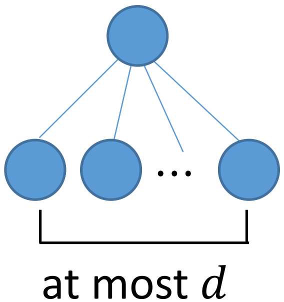


find_min : root !
insert : insert to right-most
delete : decrease a value to -∞, and delete_min
decrease_key : decrease a value, and rearrange heap
delete_min : like binary heap ☺

Time complexity ?

d-ary heap

- It has child at most d
  - Extended version of binary heap
  - How it works ?
  - We use it for Prim algorithm



find_min : O(1)
insert : O($\log_d n$)
delete : O($d \log_d n$)
decrease_key : O($\log_d n$)
delete_min : O($d \log_d n$)

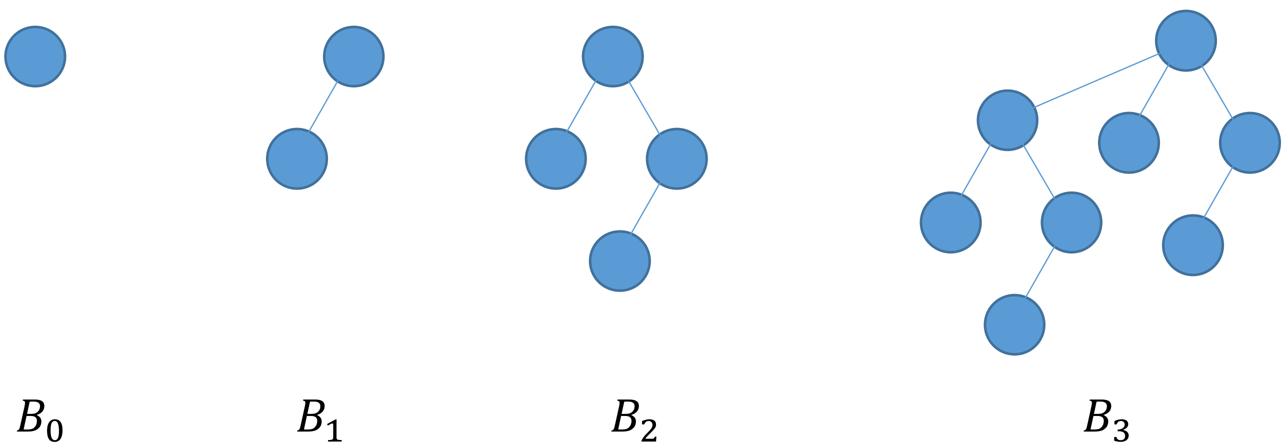
Time complexity ?

Binomial tree

- Definition

A binomial tree of height k (denoted by B_k) is defined as follows

1. B_0 consists of a single node.
2. B_k is formed by joining two B_{k-1} trees, making one's root child of the other



Binomial tree

- Property

1. B_k has 2^k nodes
2. The height of B_k is $k = \log|B_k|$
3. The root of B_k has k children

Use mathematical induction to prove

Therefore, the height and the degree of a node v in a binomial tree are both at most logarithmic in the size of the subtree rooted at v

Binomial tree

- Storing data

How can we store $n \neq 2^k$ nodes using binomial tree ?

Binomial tree

- Storing data

  How can we store $n \neq 2^k$ nodes using binomial tree ?

  Think about binary notation of n

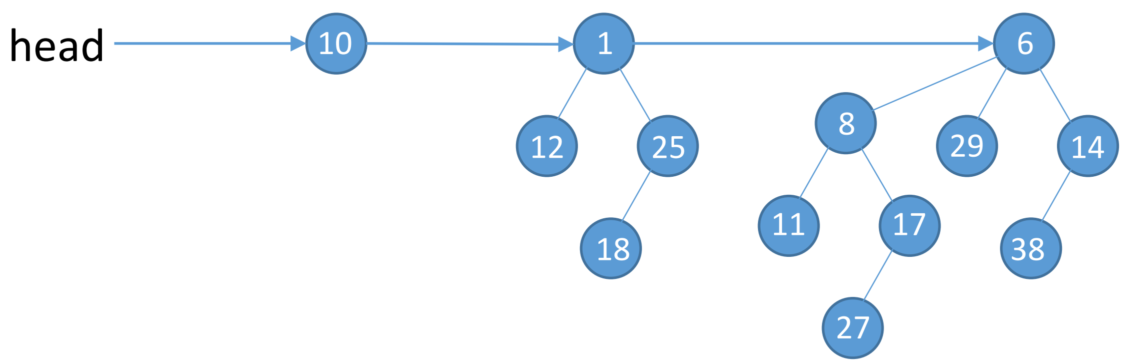
Binomial tree

- Storing data

  How can we store $n \neq 2^k$ nodes using binomial tree ?

  Think about binary notation of n

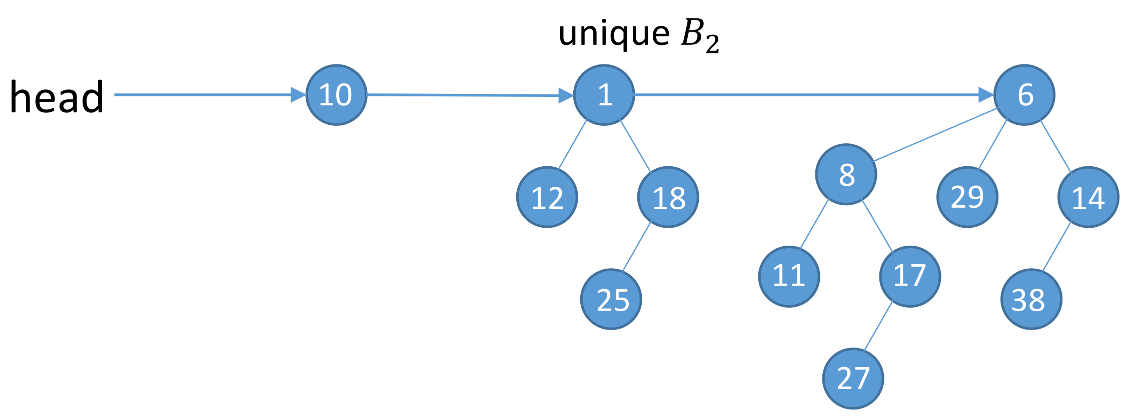
  We can use binomial tree chain ! We call it as root list



Binomial heap

- Definition

A binomial heap is a (set of) binomial tree(s) where each node is associated with a key and the heap-order property is preserved.
We also have the requirement that for any i there is at most one B_i.
In other words, it doesn't contain two B_i in the binomial heap.



Binomial heap

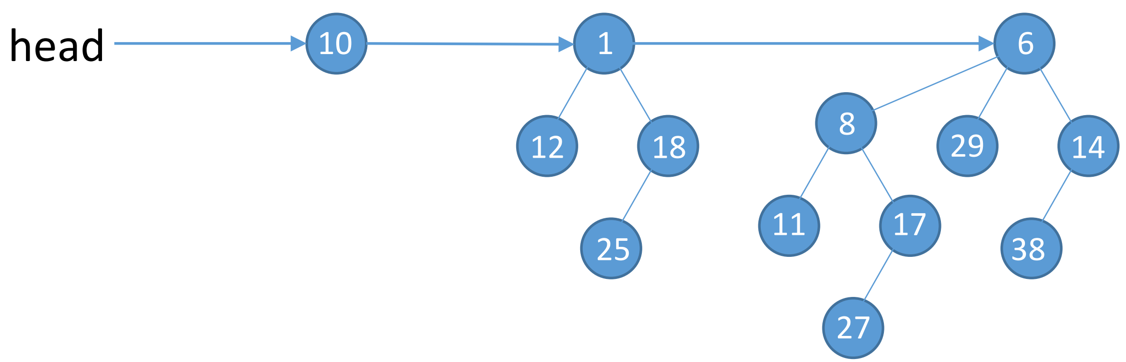
- Operations
  - find_min, delete_min, insert, delete, decrease_key

Binomial heap

- Find_min

We need to consider all the root nodes of binomial tree

It takes $O(\log n)$. Why ?

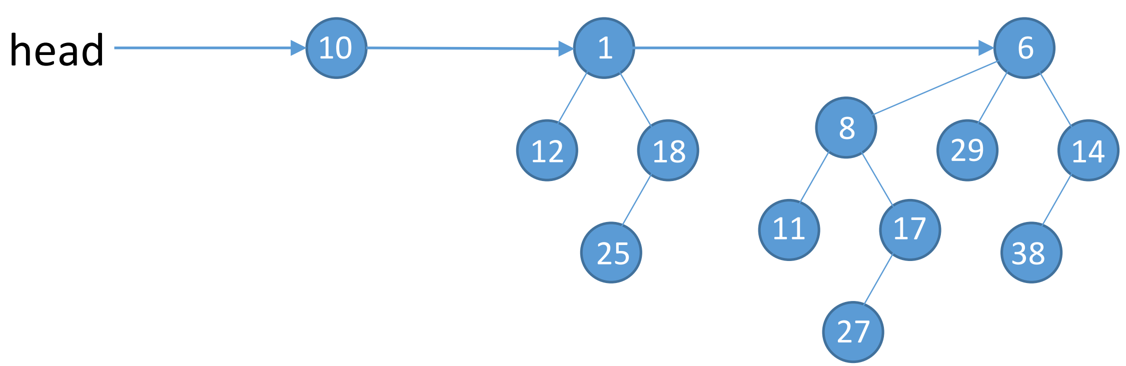


Binomial heap

- Find_min

  We need to consider all the root nodes of binomial tree

  It takes $O(\log n)$. Why ? we have $\log n$ binomial trees

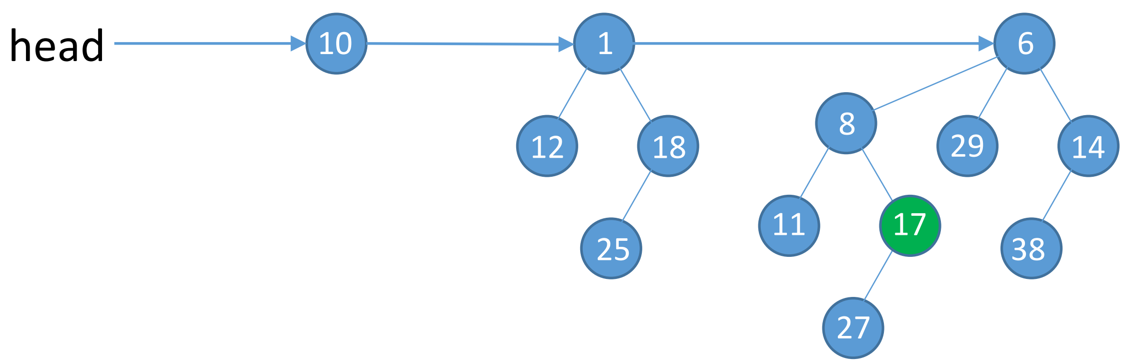


Binomial heap

- Decrease_key

  Decrease the value and rearrange binomial tree

  It takes $O(\log n)$

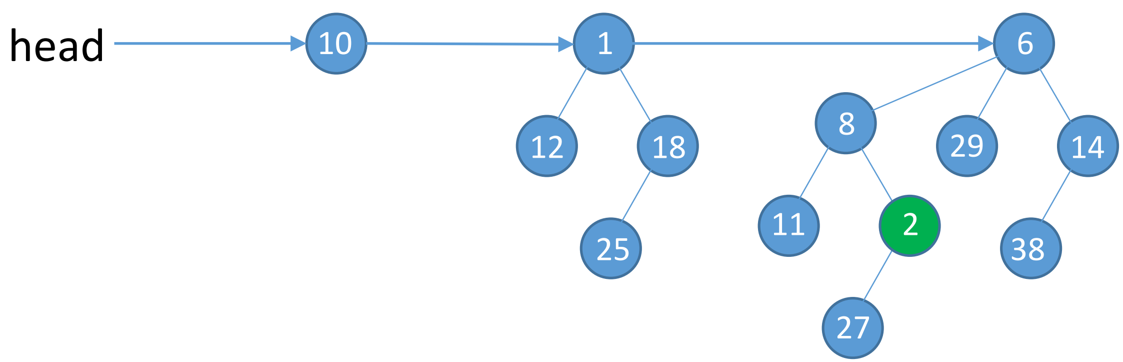


Binomial heap

- Decrease_key

  Decrease the value and rearrange binomial tree

  It takes $O(\log n)$

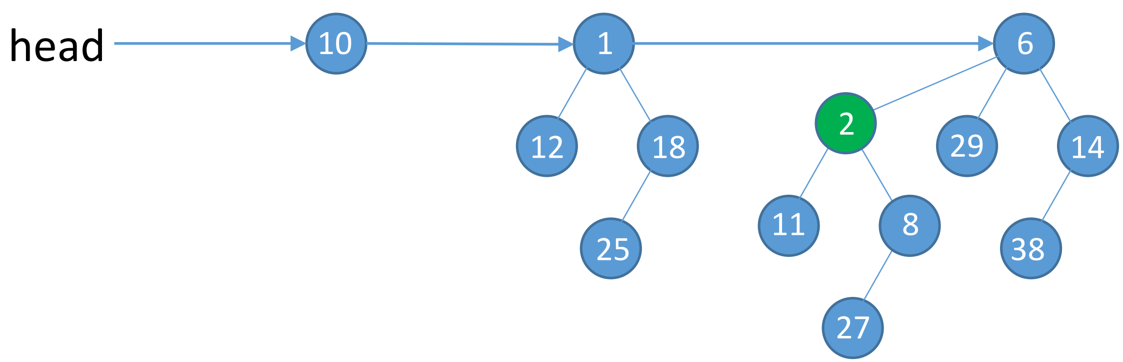


Binomial heap

- Decrease_key

  Decrease the value and rearrange binomial tree

  It takes $O(\log n)$

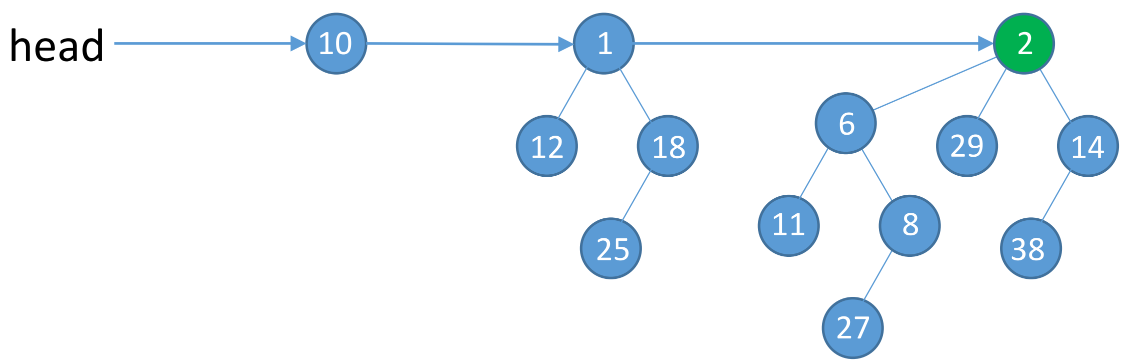


Binomial heap

- Decrease_key

  Decrease the value and rearrange binomial tree

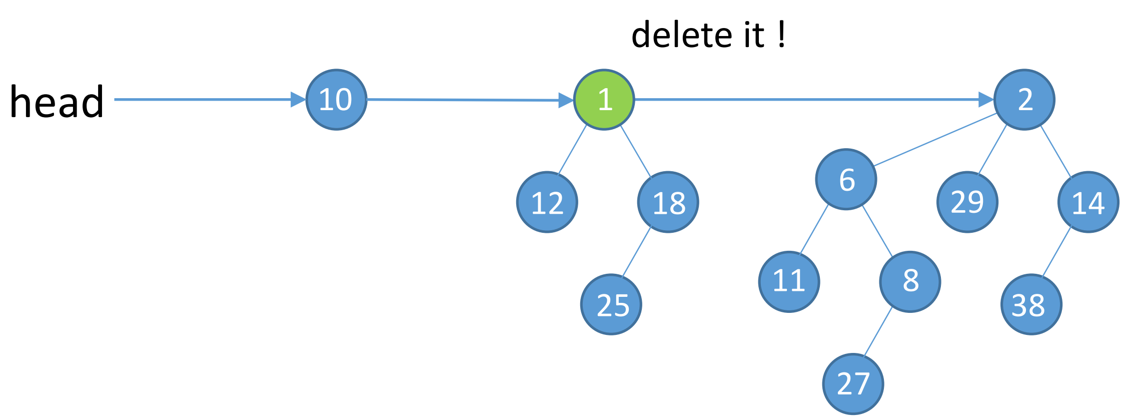
  It takes $O(\log n)$



Binomial heap

- Delete_min

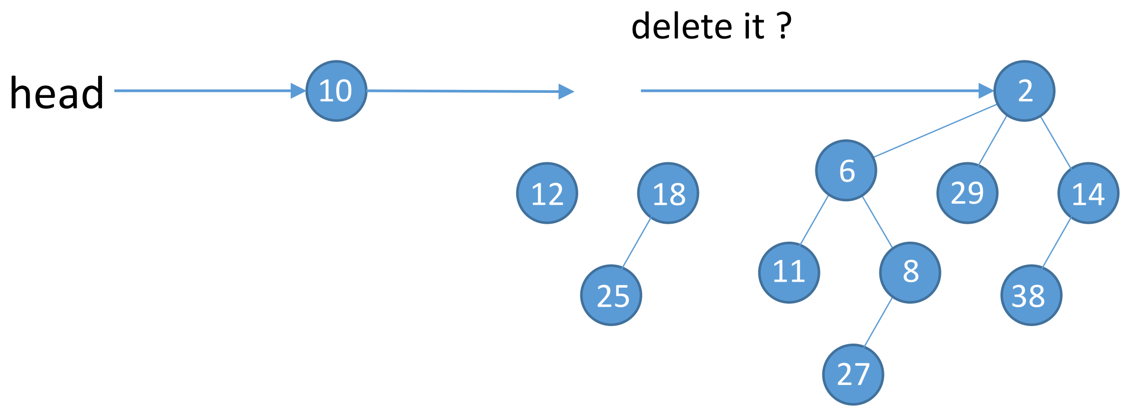
If we delete the root node of binomial tree, it is split into many trees



Binomial heap

- Delete_min

  If we delete the root node of binomial tree, it is **split** into many trees

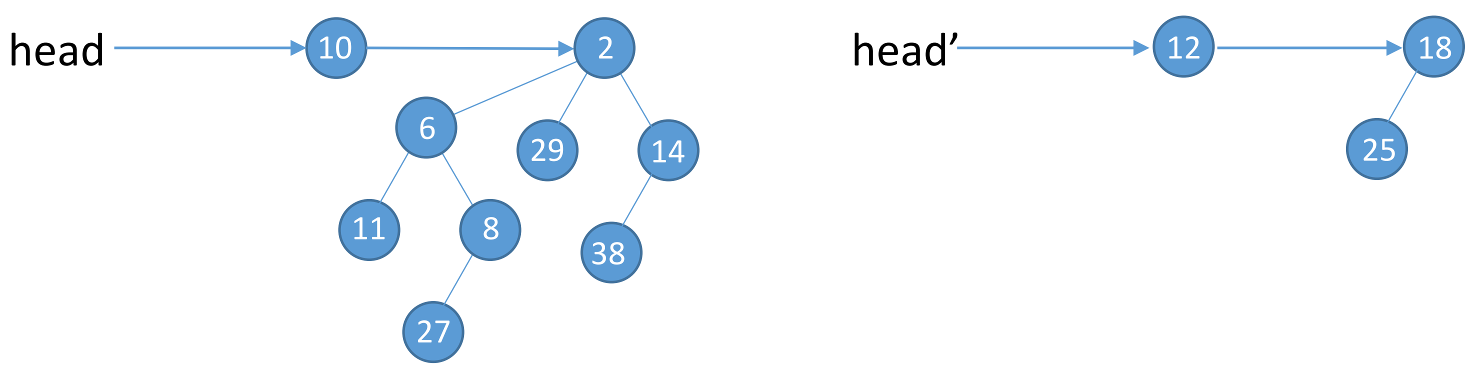


Binomial heap

- Delete_min

  If we delete the root node of binomial tree, it is **split** into many trees

  Make **another chain** ! Then we get another binomial heap.



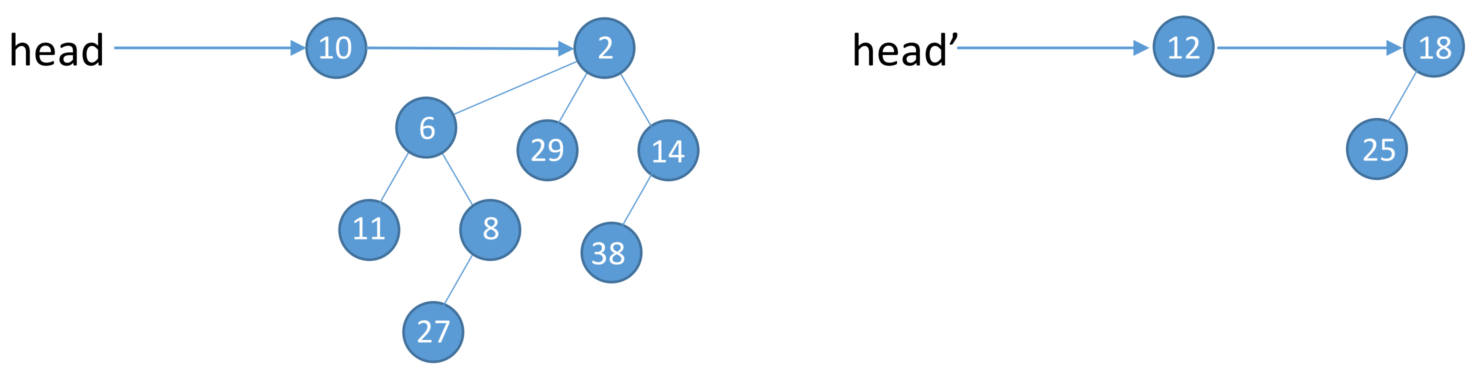
Binomial heap

- Delete_min

  If we delete the root node of binomial tree, it is **split** into many trees

  Make **another chain** ! Then we get another binomial heap.

  Merge two binomial heaps into one ! How ?



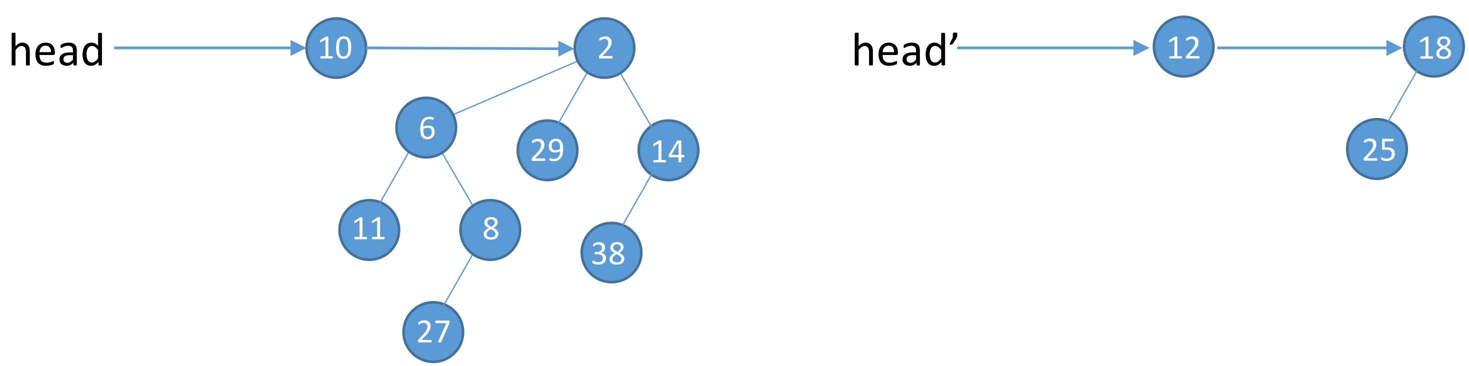
Binomial heap

- Delete_min

  If we delete the root node of binomial tree, it is **split** into many trees

  Make **another chain** ! Then we get another binomial heap.

  Merge two binomial heaps into one ! How ? Merging on merge sort ?



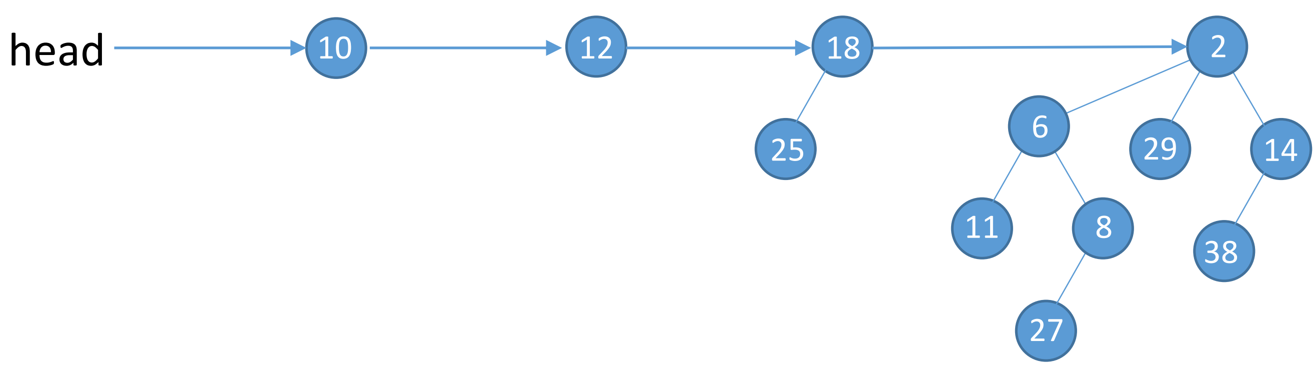
Binomial heap

- Delete_min

  If we delete the root node of binomial tree, it is **split** into many trees

  Make **another chain** ! Then we get another binomial heap.

  Merge two binomial heaps into one !



Binomial heap

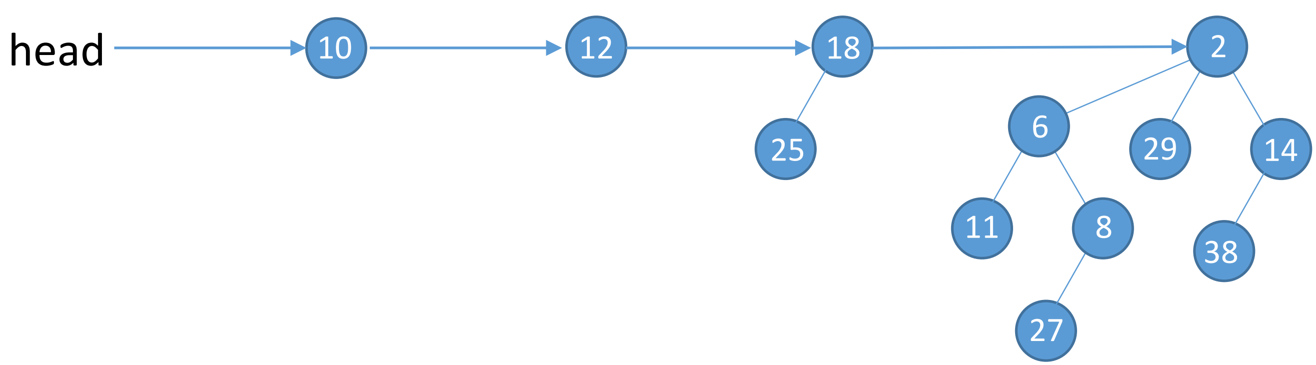
- Delete_min

  If we delete the root node of binomial tree, it is **split** into many trees

  Make **another chain** ! Then we get another binomial heap.

  Merge two binomial heaps into one !

  Rearrange the binomial heap



Binomial heap

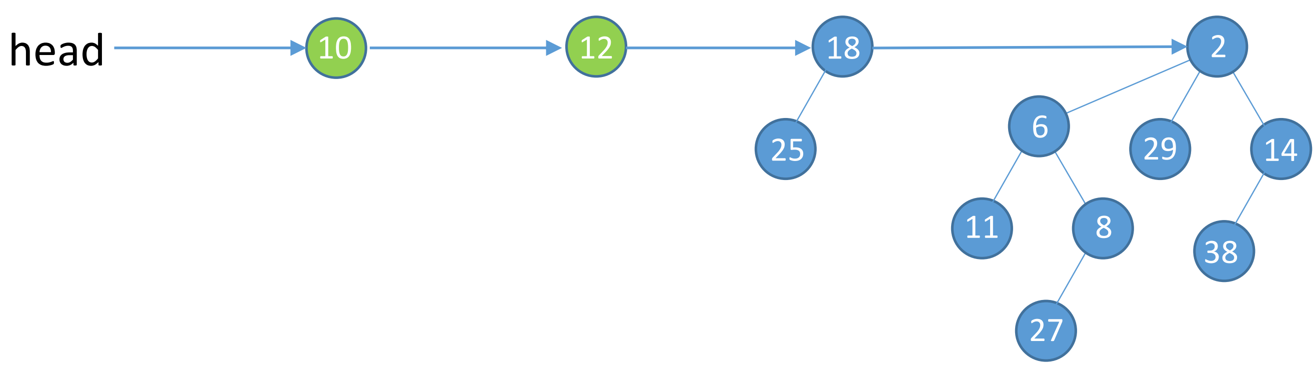
- Delete_min

  If we delete the root node of binomial tree, it is **split** into many trees

  Make **another** chain ! Then we get another binomial heap.

  Merge two binomial heaps into one !

  Rearrange the binomial heap



Binomial heap

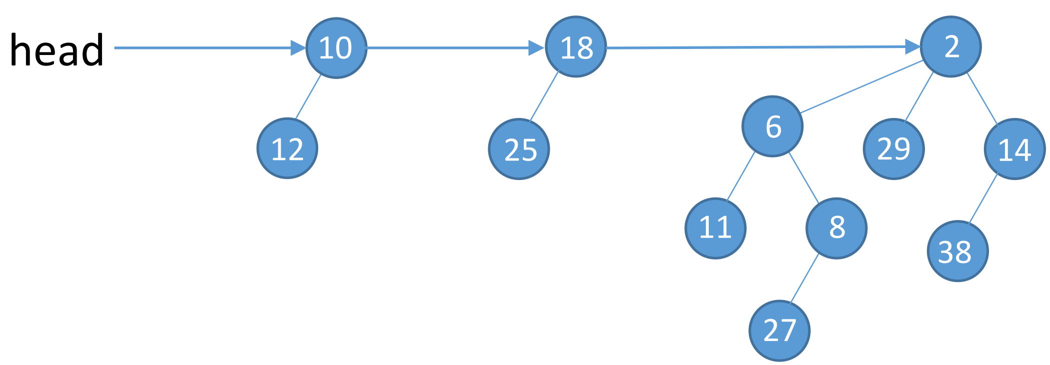
- Delete_min

  If we delete the root node of binomial tree, it is **split** into many trees

  Make **another** chain ! Then we get another binomial heap.

  Merge two binomial heaps into one !

  Rearrange the binomial heap



Binomial heap

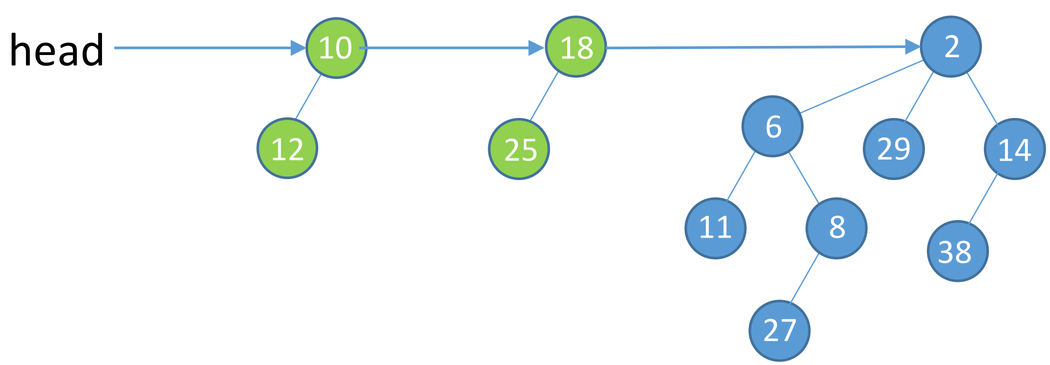
- Delete_min

  If we delete the root node of binomial tree, it is **split** into many trees

  Make **another** chain ! Then we get another binomial heap.

  Merge two binomial heaps into one !

  Rearrange the binomial heap



Binomial heap

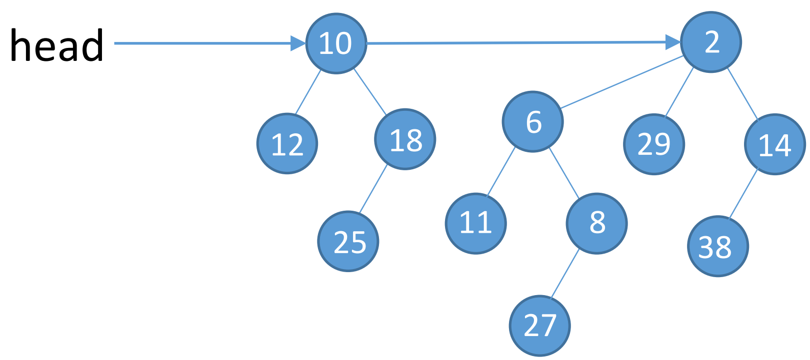
- Delete_min

  If we delete the root node of binomial tree, it is **split** into many trees

  Make **another** chain ! Then we get another binomial heap.

  Merge two binomial heaps into one !

  Rearrange the binomial heap



Binomial heap

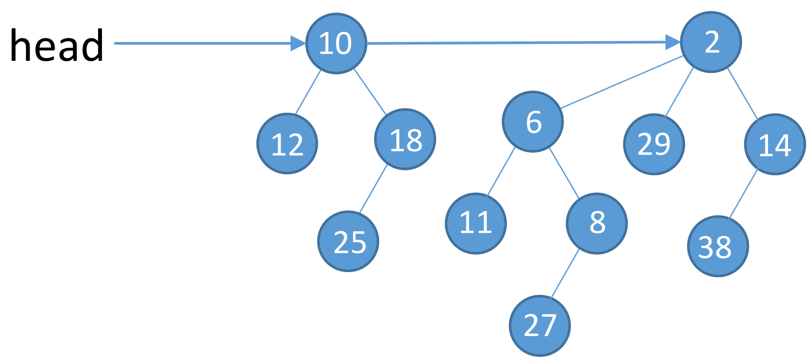
- Delete_min

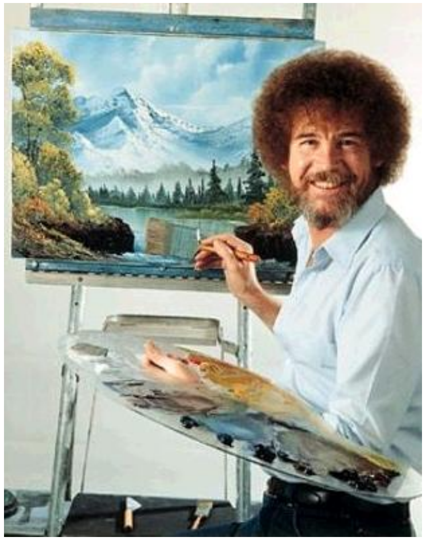
  If we delete the root node of binomial tree, it is **split** into many trees

  Make **another** chain ! Then we get another binomial heap.

  Merge two binomial heaps into one !

  Rearrange the binomial heap

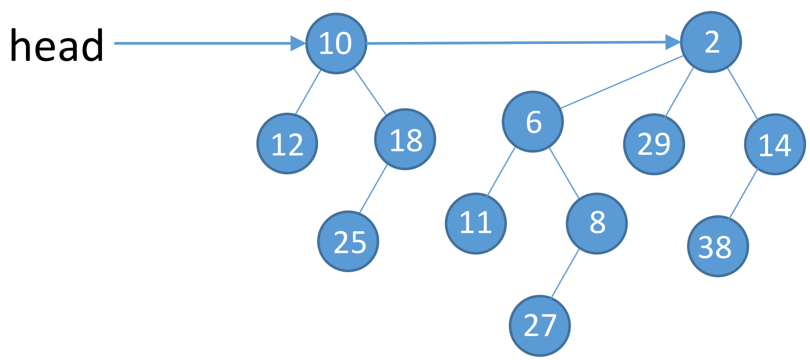




Binomial heap

- Delete_min

  Time complexity ?

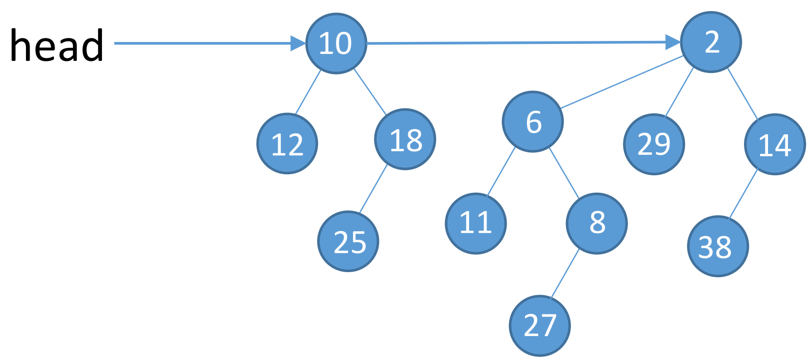


Binomial heap

- Delete_min

  Time complexity ?

  Each node has at most $\log n$ child.

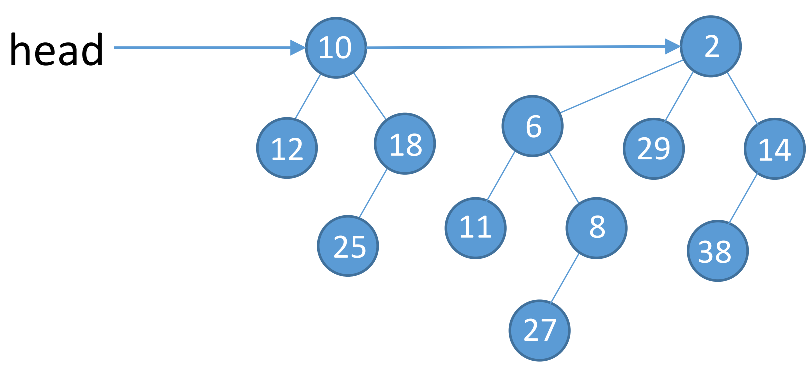


Binomial heap

- Delete_min

  Time complexity ?

  Each node has at most $\log n$ child.
  New binomial heap consists of at most $\log n$ binomial trees

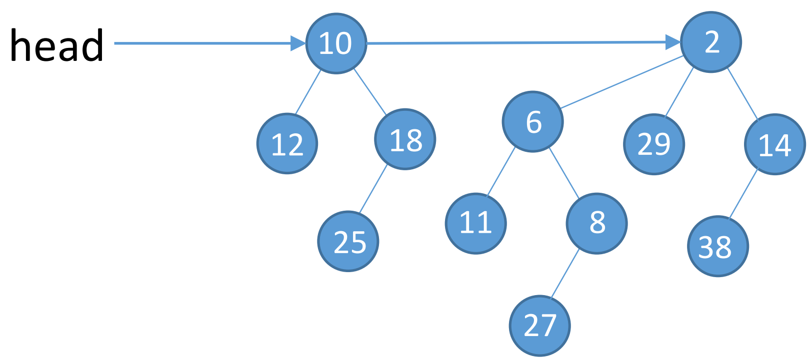


Binomial heap

- Delete_min

Time complexity ?

Each node has at most $\log n$ child.
New binomial heap consists of at most $\log n$ binomial trees
Therefore, Merging and rearrange takes O($\log n$).

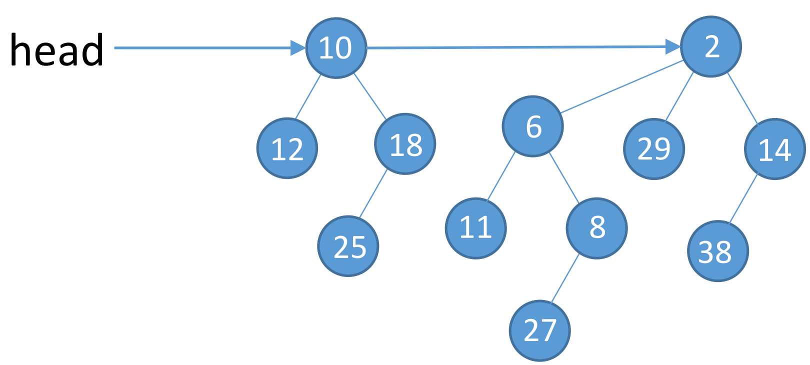


Binomial heap

- Insert

  Insert a new node and rearrange binomial heap

  $O(\log n)$

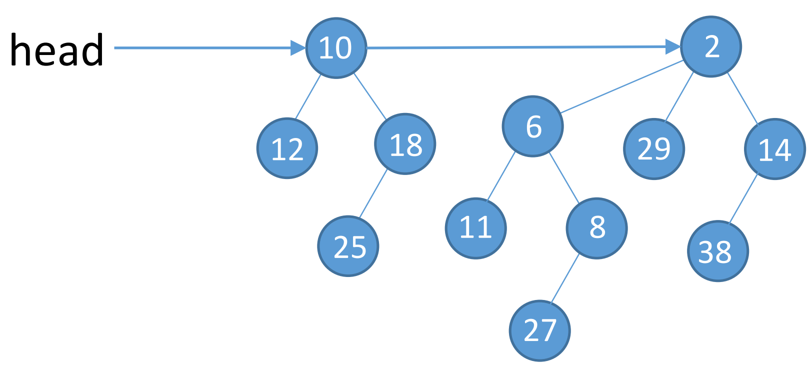


Binomial heap

- Delete

Make it as -∞, and delete_min !

O($\log n$)



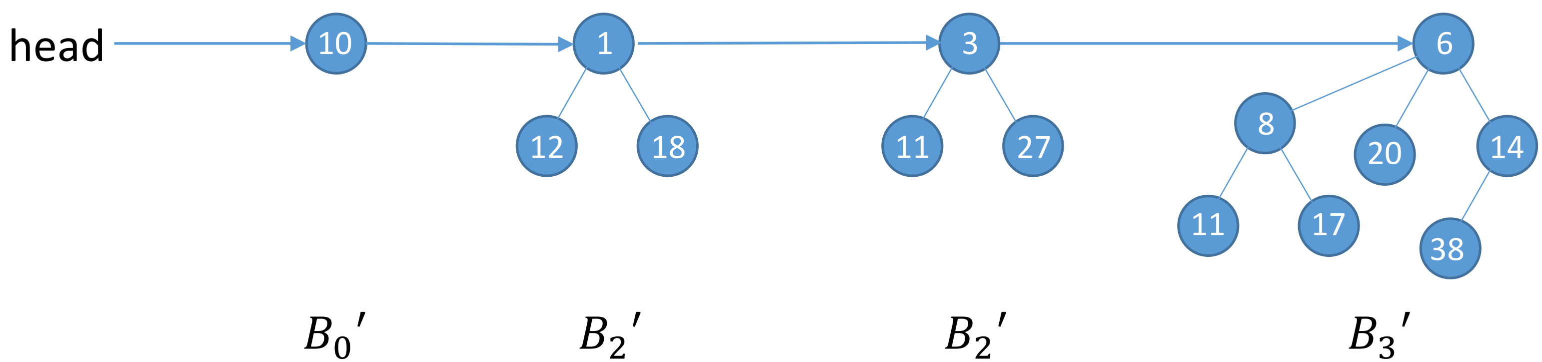
Fibonacci heap

- Theoretically fastest heap
  - But it is just theoretical. It is slow if you implement it
  - We don't use it
- Similar with binomial heap
  - Relaxed version of binomial heap
  - Use a *lazy* update scheme.

Operation	Binary[1]	Binomial[1]	Fibonacci[1]
find-min	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
delete-min	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$*
insert	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
decrease-key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$*
merge	$\Theta(n)$	$O(\log n)$**	$\Theta(1)$

Fibonacci heap

- Main feature
  - Individual trees are not necessarily binomial (let me denote it as B_i')
  - Allow many copies of B_i' for the same i.
- Operations
  - find_min, delete_min, insert, delete, decrease_key

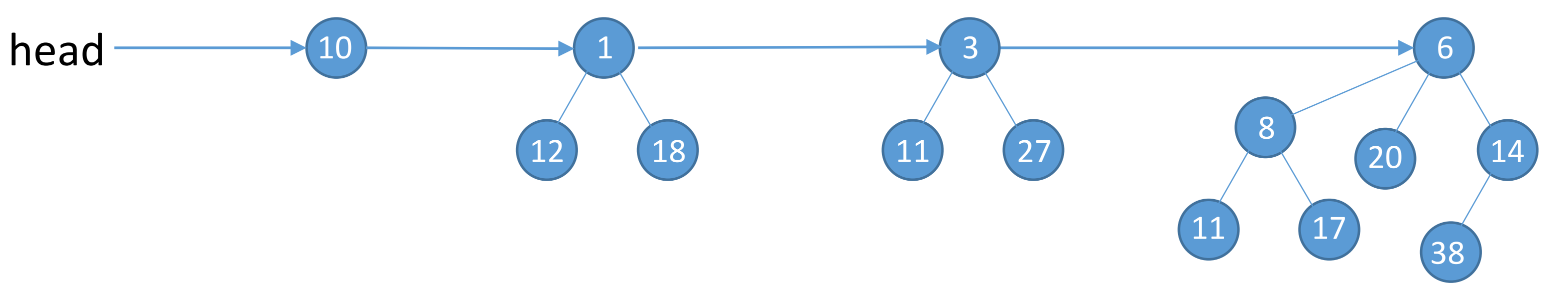


I allow it because I'm lazy ☺

Fibonacci heap

- Find_min

  via pointer !



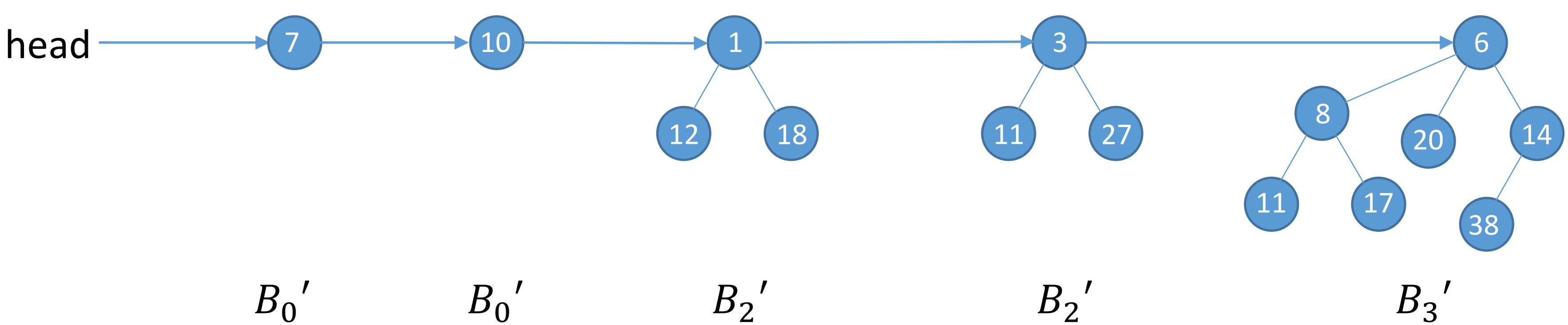
$B_0{}'$ $B_2{}'$ $B_2{}'$ $B_3{}'$

Fibonacci heap

- Insert

  Create a node. No update because I'm lazy ☺

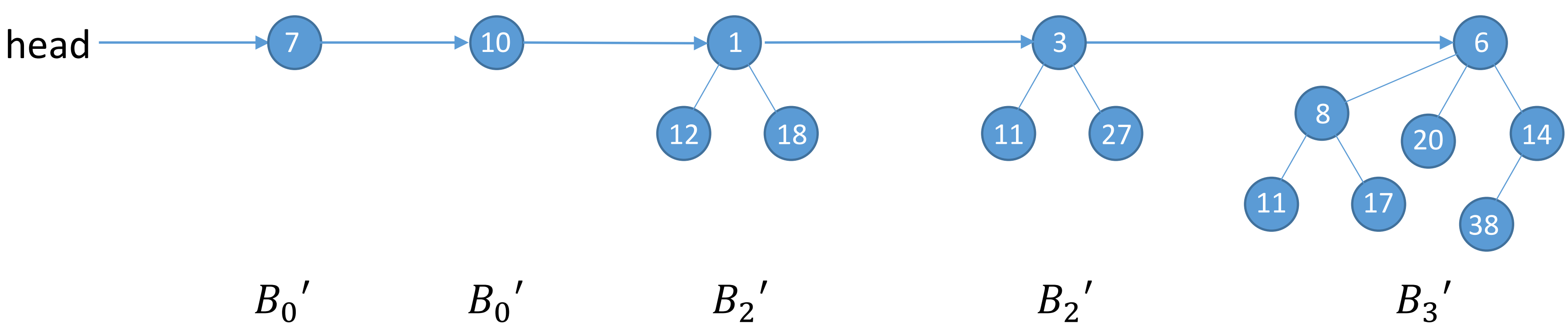
  Give $1 to inserted node. I'll explain the meaning of 'coin' later.



Fibonacci heap

- Delete

  -∞, delete_min



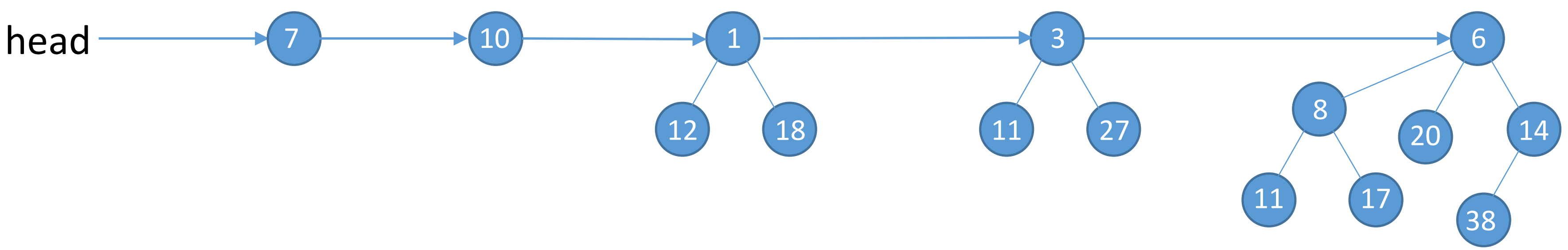
Fibonacci heap

- Delete_min

  Delete min and rearrange Fibonacci heap.

  Time complexity will proportional to the number of trees.

  Is it fast ?



B_0' B_0' B_2' B_2' B_3'

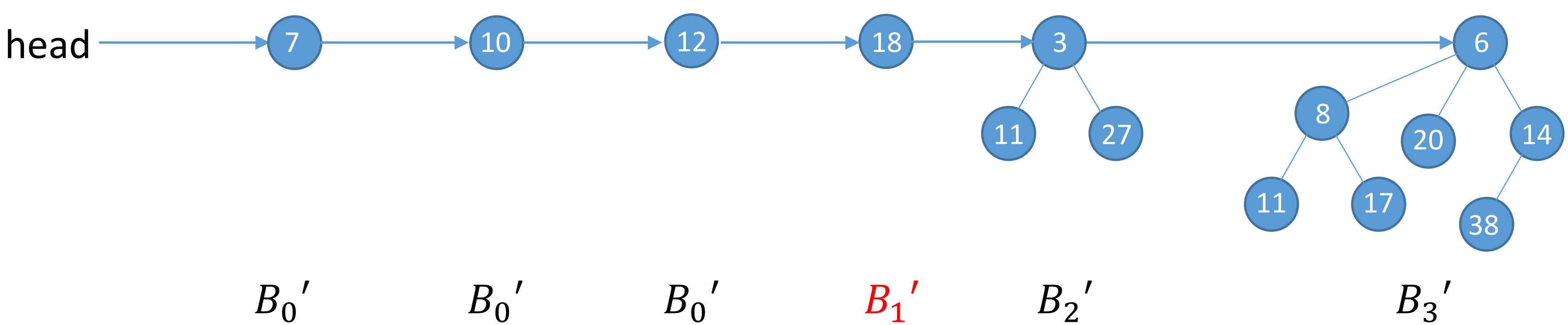
Fibonacci heap

- Delete_min

  Delete min and rearrange Fibonacci heap.

  Time complexity will proportional to the number of trees.

  Is it fast ?



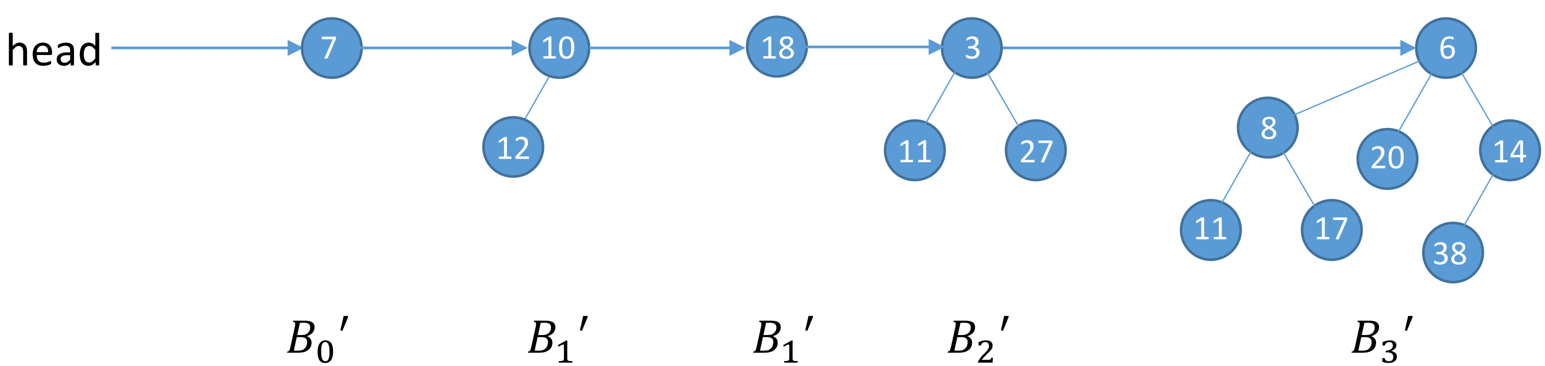
Fibonacci heap

- Delete_min

  Delete min and rearrange Fibonacci heap.

  Time complexity will proportional to the number of trees.

  Is it fast ?



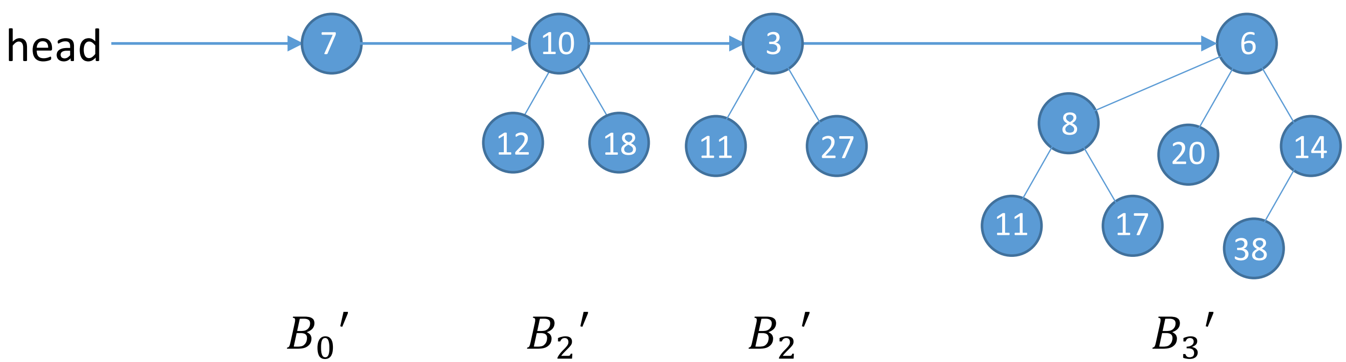
Fibonacci heap

- Delete_min

  Delete min and rearrange Fibonacci heap.

  Time complexity will proportional to the number of trees.

  Is it fast ?



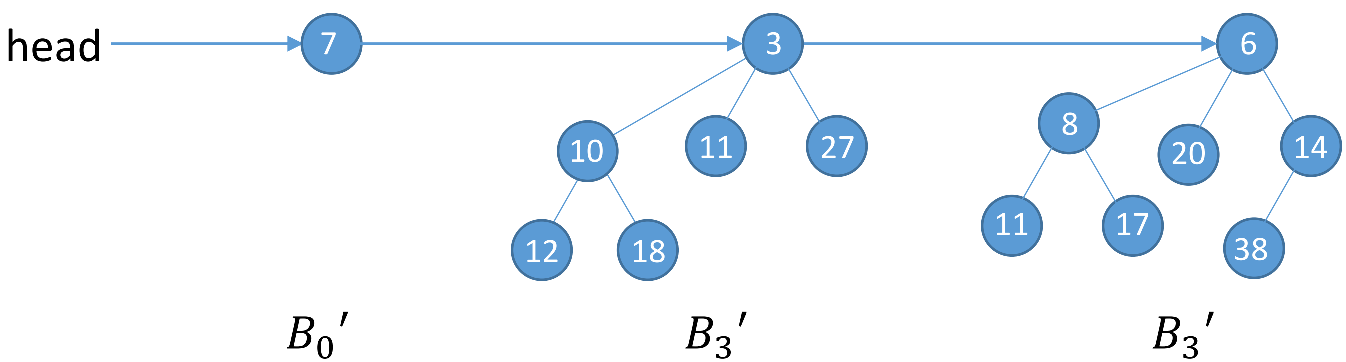
Fibonacci heap

- Delete_min

  Delete min and rearrange Fibonacci heap.

  Time complexity will proportional to the number of trees.

  Is it fast ?



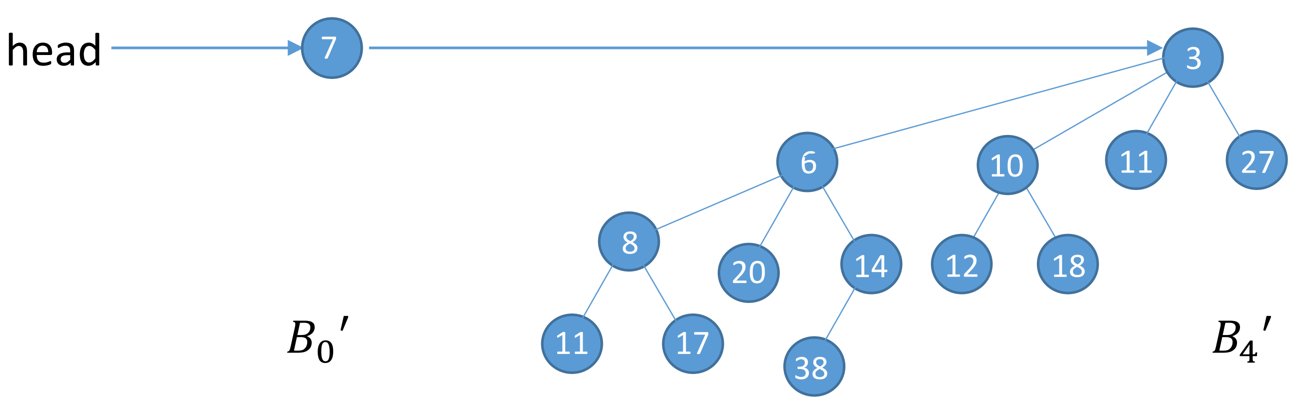
Fibonacci heap

- Delete_min

  Delete min and rearrange Fibonacci heap.

  Time complexity will proportional to the number of trees.

  Is it fast ?



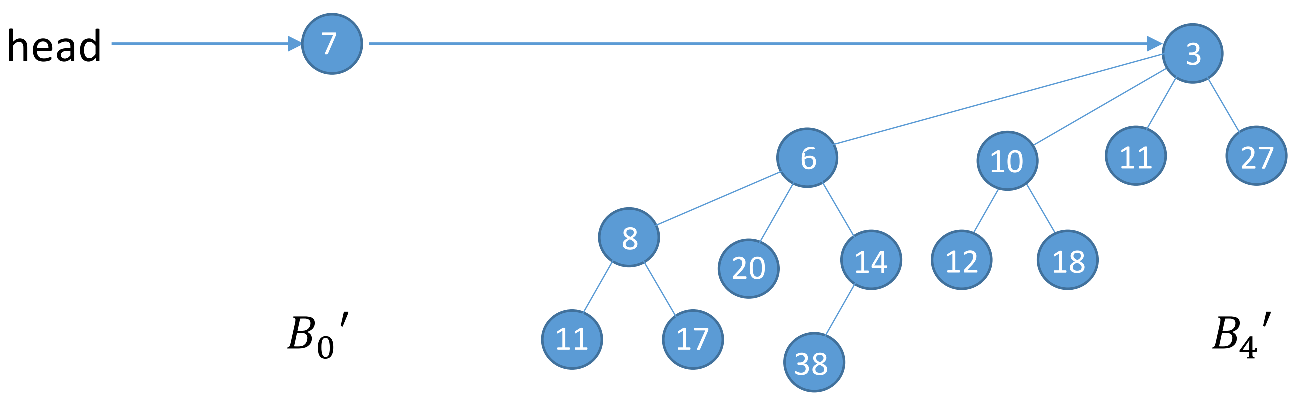
Fibonacci heap

- Delete_min

  Delete min and rearrange Fibonacci heap.

  Time complexity will proportional to the number of trees.

  Is it fast ? → we'll analyze it soon.

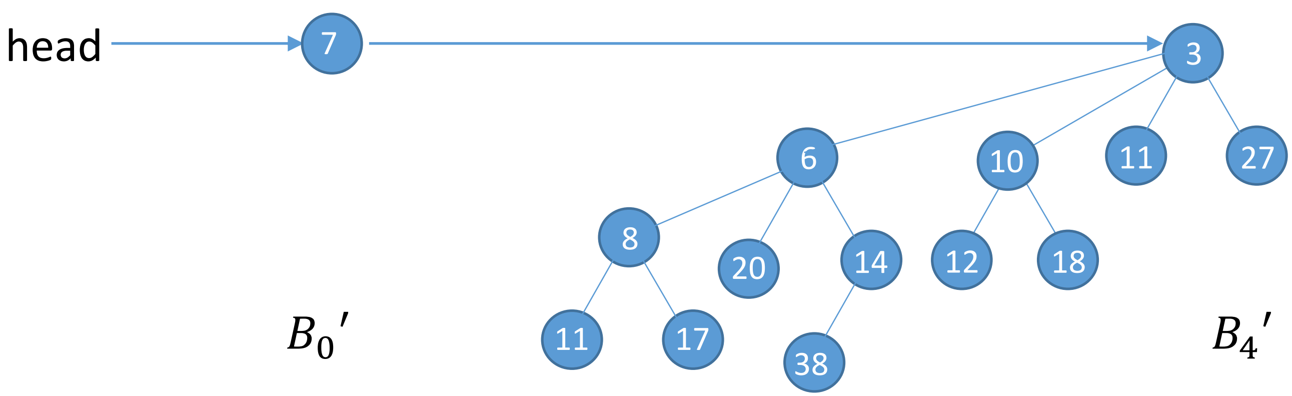


Fibonacci heap

- Decrease_key

  Decrease the value of the element. If the heap-order property is violated, **cut** the link between the node and its parent. (It may produce a result which is not a binomial tree)

  Not just cut, we use "**Cascading cut**"

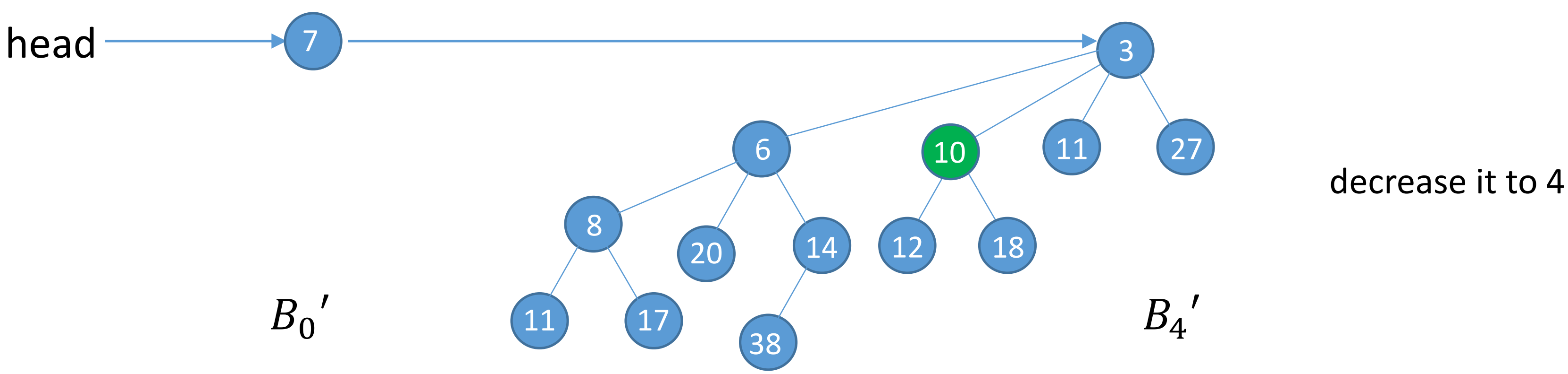


Fibonacci heap

- Decrease_key

  Decrease the value of the element. If the heap-order property is violated, **cut** the link between the node and its parent. (It may produce a result which is not a binomial tree)

  Not just cut, we use "**Cascading cut**"

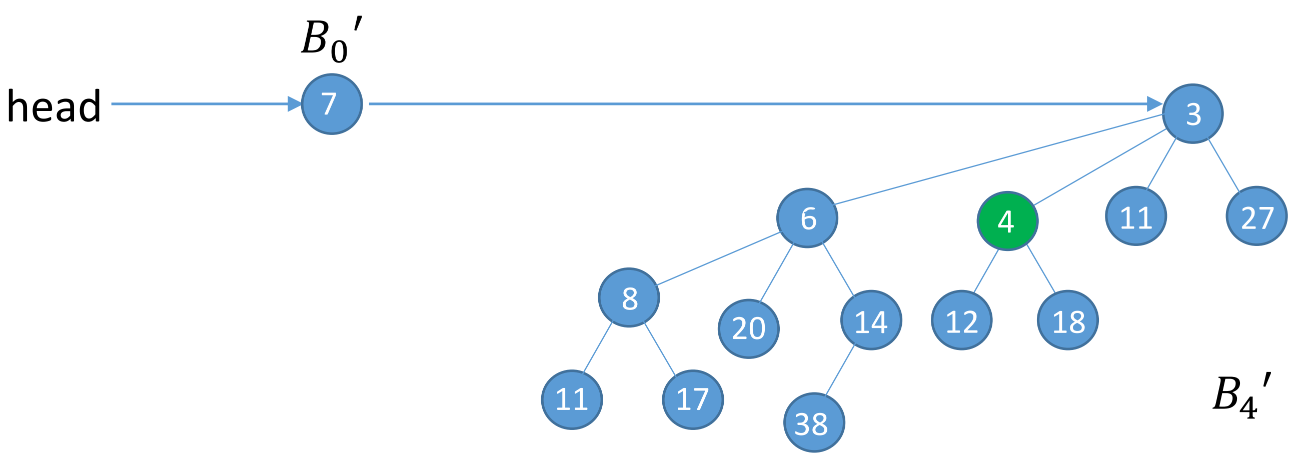


Fibonacci heap

- Decrease_key

  Decrease the value of the element. If the heap-order property is violated, **cut** the link between the node and its parent. (It may produce a result which is not a binomial tree)

  Not just cut, we use "**Cascading cut**"



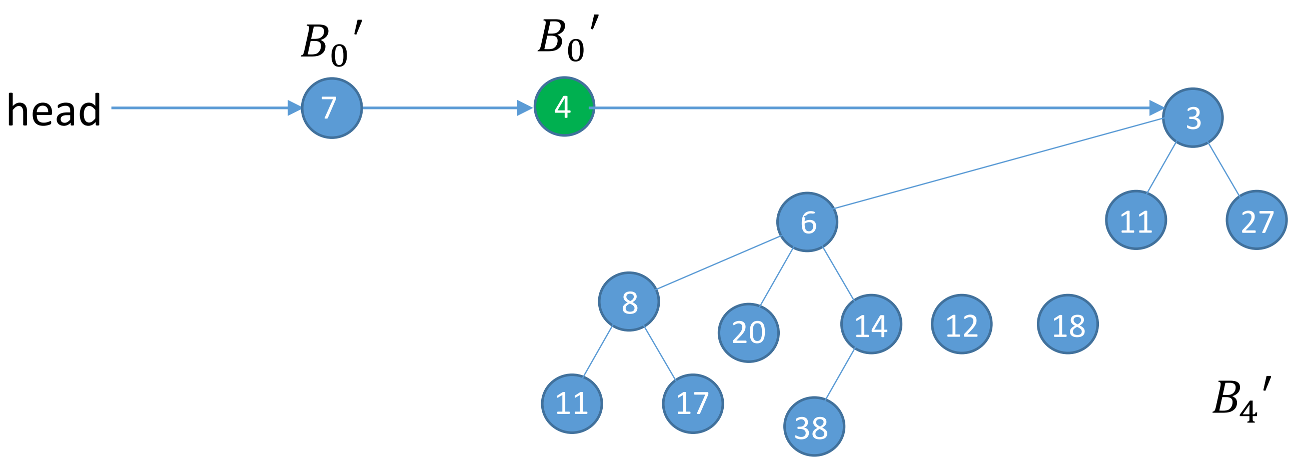
decrease it to 4
→ heap-order is violated

Fibonacci heap

- Decrease_key

  Decrease the value of the element. If the heap-order property is violated, **cut** the link between the node and its parent. (It may produce a result which is not a binomial tree)

  Not just cut, we use "**Cascading cut**"



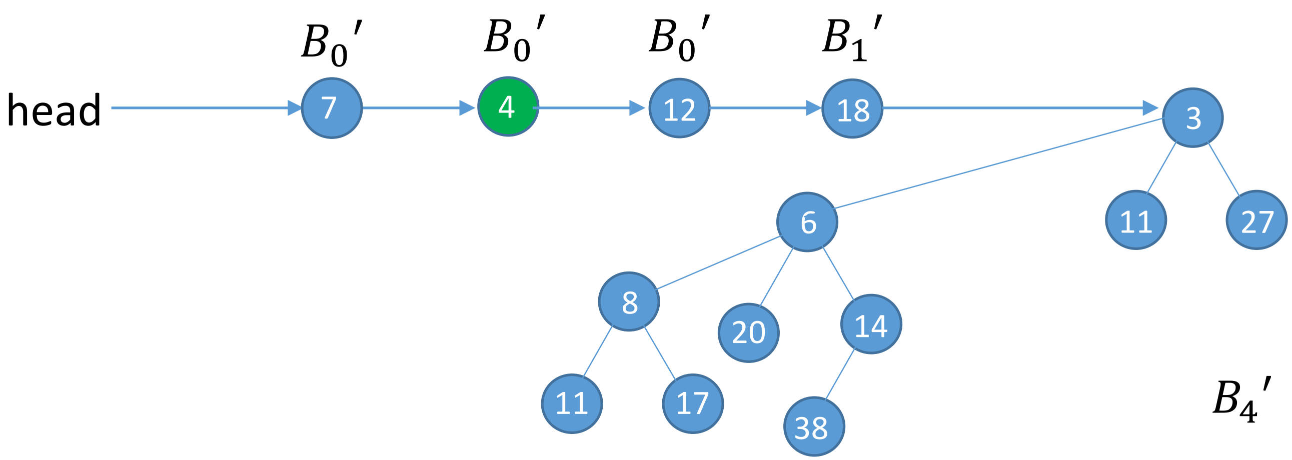
decrease it to 4
→ heap-order is violated
→ Cut !

Fibonacci heap

- Decrease_key

Decrease the value of the element. If the heap-order property is violated, **cut** the link between the node and its parent. (It may produce a result which is not a binomial tree)

Not just cut, we use "**Cascading cut**"

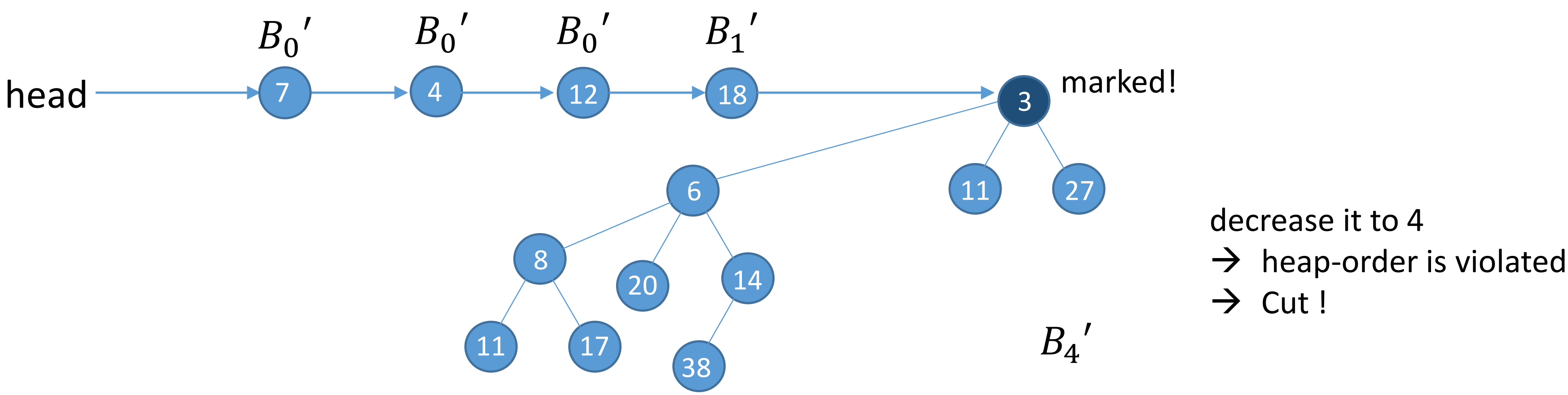


decrease it to 4
→ heap-order is violated
→ Cut !

Cascading cut

- Definition

1. Whenever a node is being cut, *mark the parent* of the cut node in the original tree, and
    - Pay $1 for the cut
    - Store $2 in the parent of the cut node
    - Store $1 in the new root (cut node).

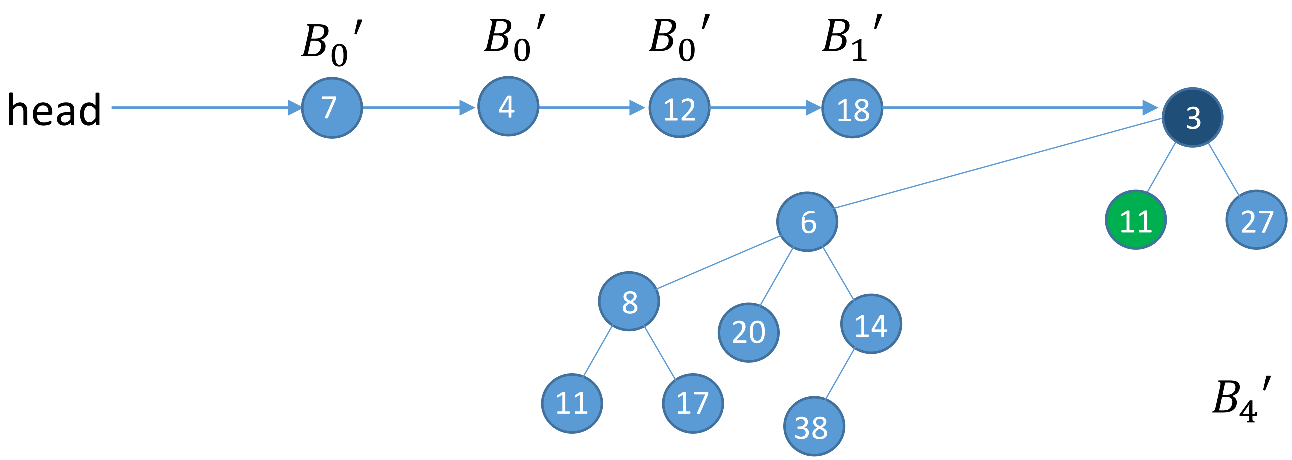


Cascading cut

- Definition

  2. When a 2nd child of a node v is lost (cutting a child of an already marked node), by that time node v will have accumulated \$ 4; recursively cut that node from its parent, marking again the parent of v and using \$4 to pay for the operation before.

     \$1 for the cut, \$2 to its parent, \$1 to the new root



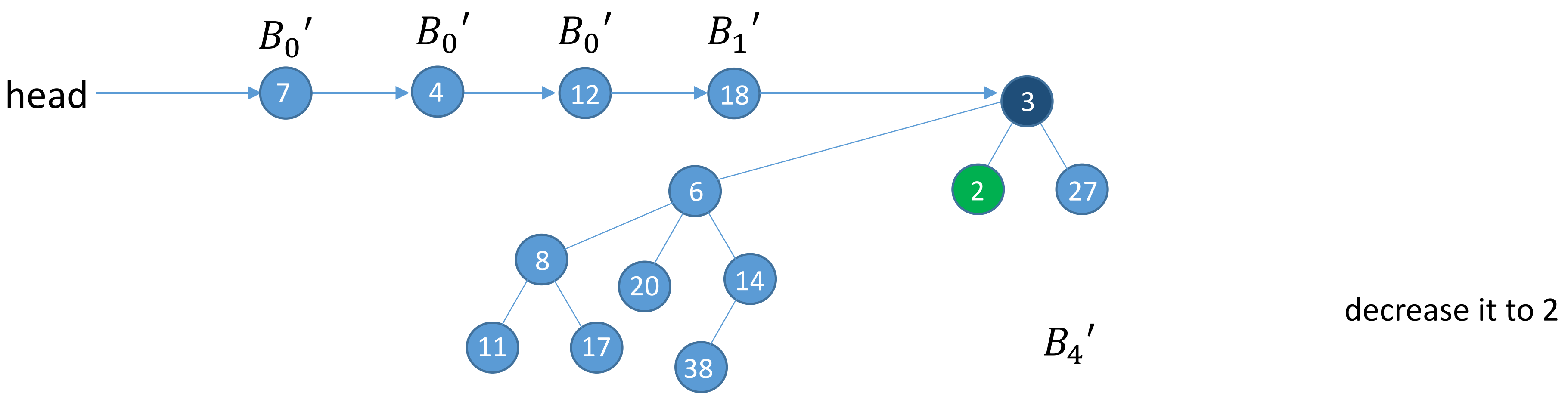
decrease it to 2 !

Cascading cut

- Definition

  2. When a 2nd child of a node v is lost (cutting a child of an already marked node), by that time node v will have accumulated \$ 4; recursively cut that node from its parent, marking again the parent of v and using \$4 to pay for the operation before.

     \$1 for the cut, \$2 to its parent, \$1 to the new root

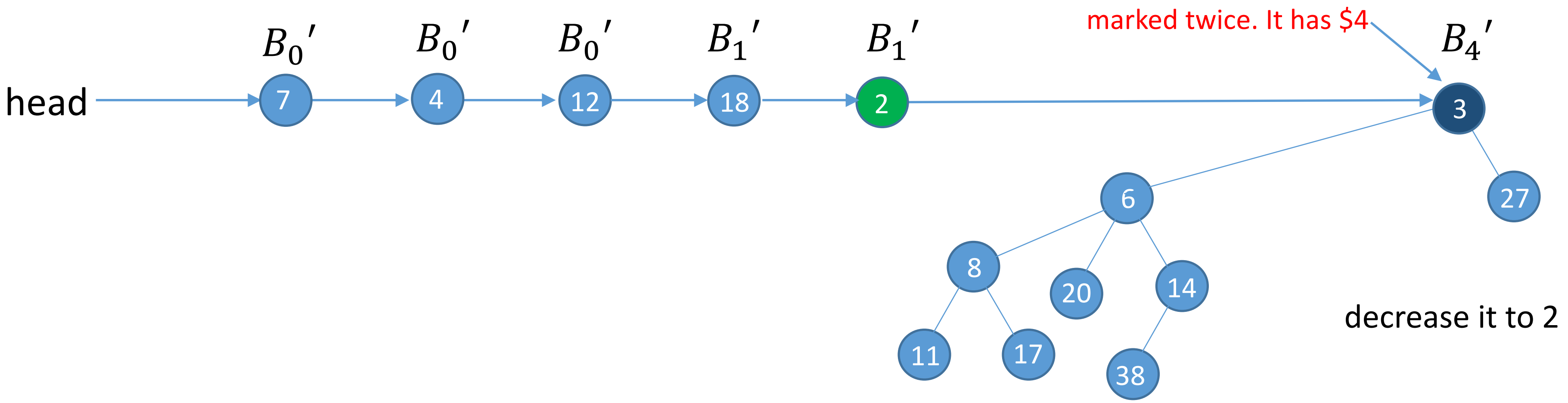


Cascading cut

- Definition

  2. When a 2nd child of a node v is lost (cutting a child of an already marked node), by that time node v will have accumulated \$ 4; recursively cut that node from its parent, marking again the parent of v and using \$4 to pay for the operation before.

     \$1 for the cut, \$2 to its parent, \$1 to the new root

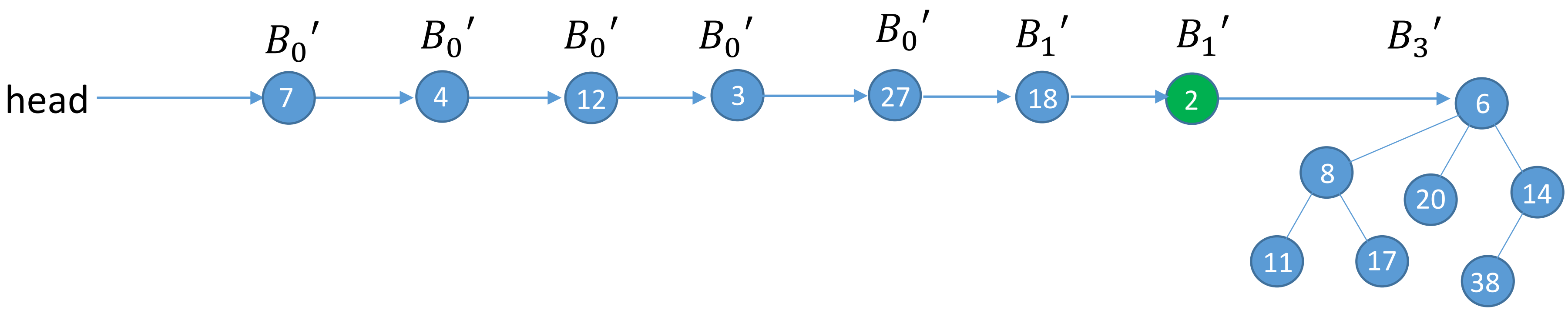


Cascading cut

- Definition

2. When a 2nd child of a node v is lost (cutting a child of an already marked node), by that time node v will have accumulated \$ 4; recursively cut that node from its parent, marking again the parent of v and using \$4 to pay for the operation before.

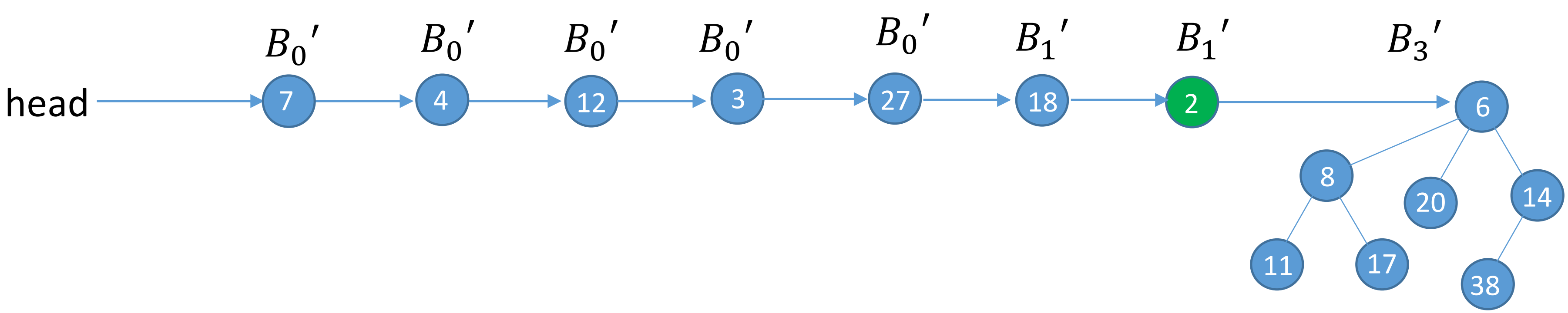
   \$1 for the cut, \$2 to its parent, \$1 to the new root



Cascading cut

- Time complexity of cascading cut

Each node has the coin they already have.
Each cut requires only $4, and decrease_key takes **overall** still amortized time O(1) (Overall cost / The number of operation)

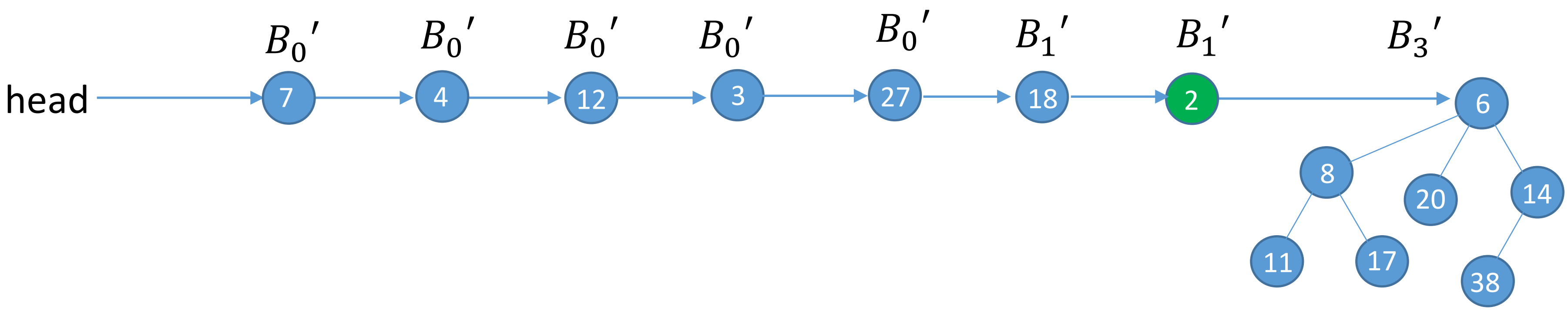


Fibonacci heap

- Decrease_key

Decrease the value of the element. If the heap-order property is violated, **cut** the link between the node and its parent. (It may produce a result which is not a binomial tree)

Not just cut, we use "**Cascading cut**"

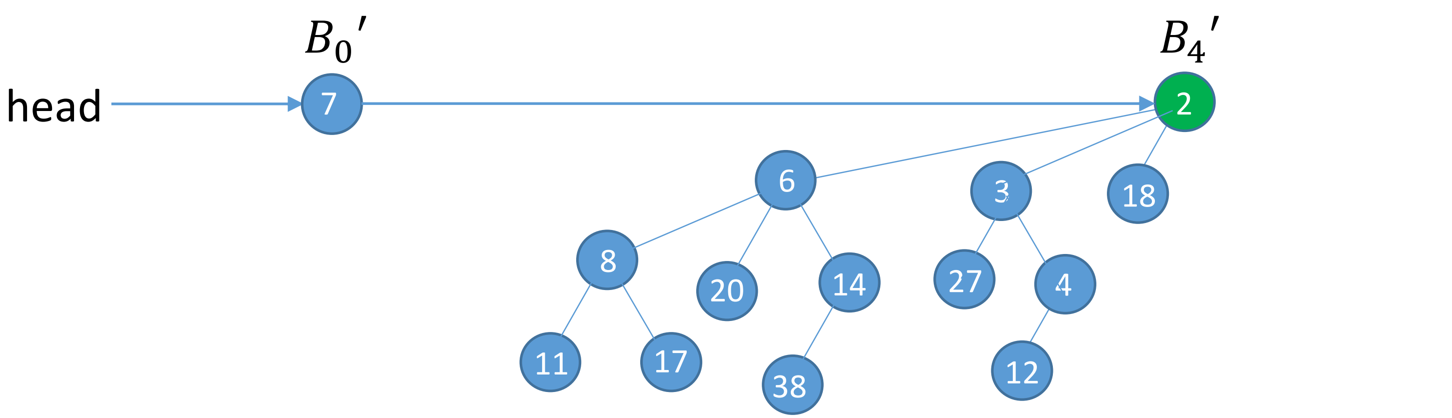


Fibonacci heap

- Decrease_key

Decrease the value of the element. If the heap-order property is violated, **cut** the link between the node and its parent. (It may produce a result which is not a binomial tree)

Not just cut, we use "**Cascading cut**"

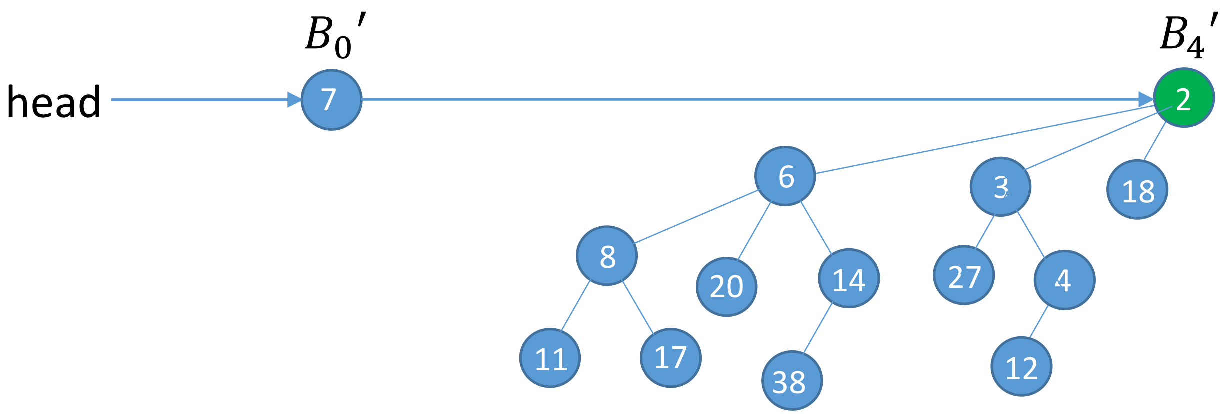


Fibonacci heap

- Analysis

  Cascading cut takes O(1)

  How much is merging cost ?



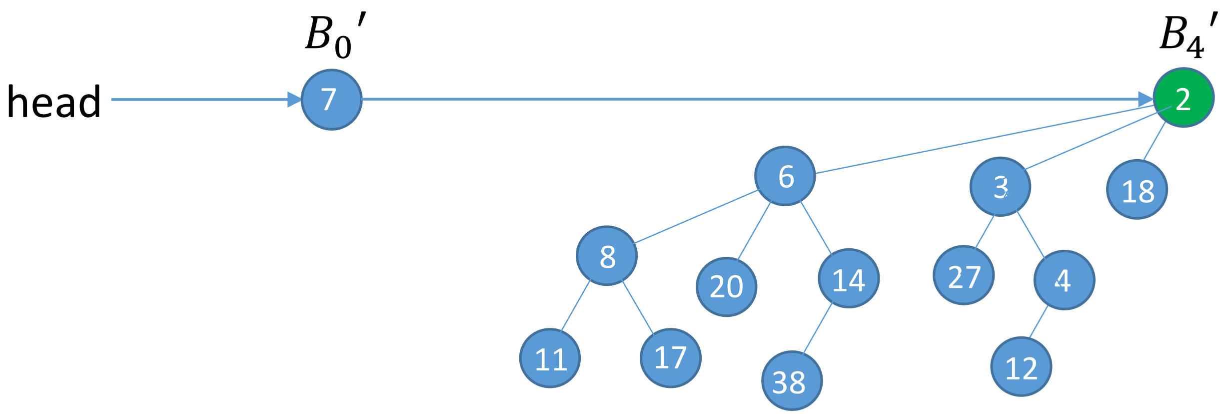
Rearrange !

Fibonacci heap

- Analysis

  Cascading cut takes O(1)

  How much is merging cost ? O(1) because of the coin



Rearrange !

Fibonacci heap

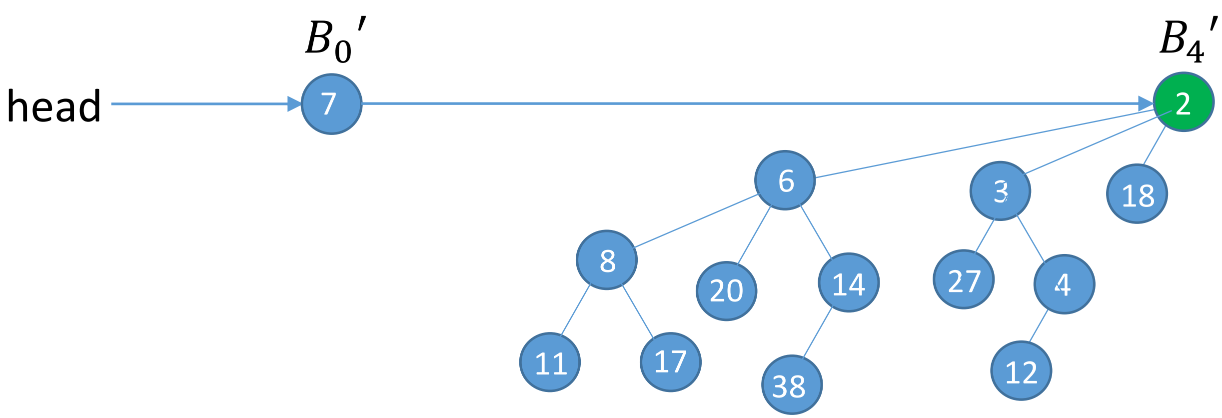
- Analysis

  Cascading cut takes O(1)

  After cut, we get (# of child) additional trees

  How much is merging cost per child ? O(1) because of the coin

  How many child does a node has ?



Rearrange !

Fibonacci heap

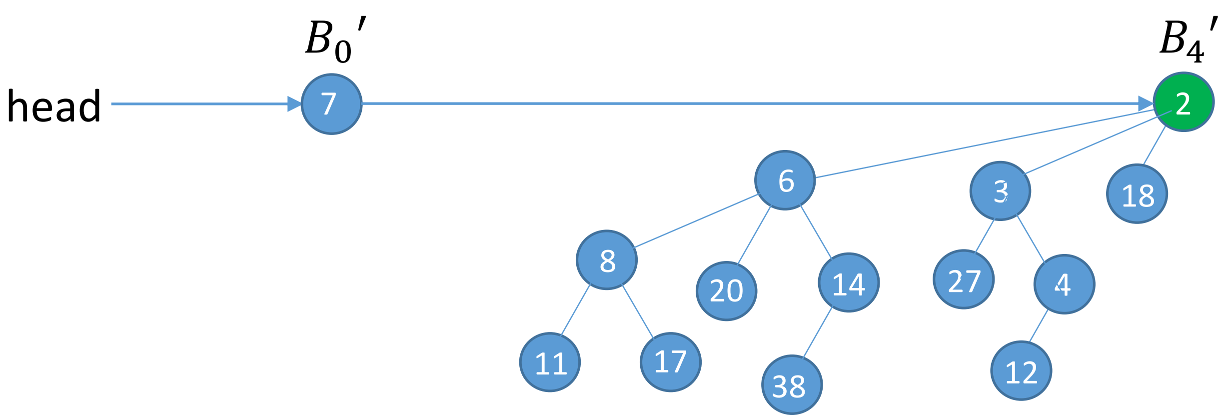
- Analysis

  Cascading cut takes O(1)

  After cut, we get (# of child) additional trees

  How much is merging cost per child ? O(1) because of the coin

  How many child does a node has ? → Degree of a node



Rearrange !

Fibonacci heap

- Analysis

Let x be any node in a Fibonacci heap. Then

size(x) $\geq F_{k+2} \geq \emptyset^k$ where k is the degree of x, ∅ is golden ratio

Proof by induction on k. Consider a node x of degree $k = n + 1$.

Let y_i be the k children of x, from oldest to youngest (i.e. y_0 was made children of x before y_1, and so on). Then

size(x) = 1 + size(y_0) + size(y_1) + … + size(y_{k-1})

Fibonacci heap

- Analysis

Let x be any node in a Fibonacci heap. Then

size(x) $\geq F_{k+2} \geq \emptyset^k$ where k is the degree of x, $\emptyset$ is golden ratio

Proof by induction on k. Consider a node x of degree $k = n + 1$.

The degree of y_i is at least $i - 1$ for $1 \leq i \leq d$ because we merge two trees when their degree is the same.

Fibonacci heap

- Analysis

Let x be any node in a Fibonacci heap. Then

size(x) $\geq F_{k+2} \geq \emptyset^k$ where k is the degree of x, $\emptyset$ is golden ratio

Proof by induction on k. Consider a node x of degree $k = n + 1$.

Therefore,

$$\begin{aligned} \text{size}(x) &= 1 + \text{size}(y_0) + \text{size}(y_1) + \text{size}(y_2) + \ldots + \text{size}(y_{k-1}) \\ &\geq 1 + 1 + F_2 + F_3 + \ldots + F_k \\ &\geq F_{k+2} \quad \geq \emptyset^k \text{ (also it can be proved by induction)} \end{aligned}$$

Fibonacci heap

- Analysis

Therefore, $k \leq \log_{\emptyset} size\ (x)$.

$$\therefore k \leq \log n$$

Therefore, decrease_key takes O($\log n$)