

# *BLM5106- Advanced Algorithm Analysis and Design*

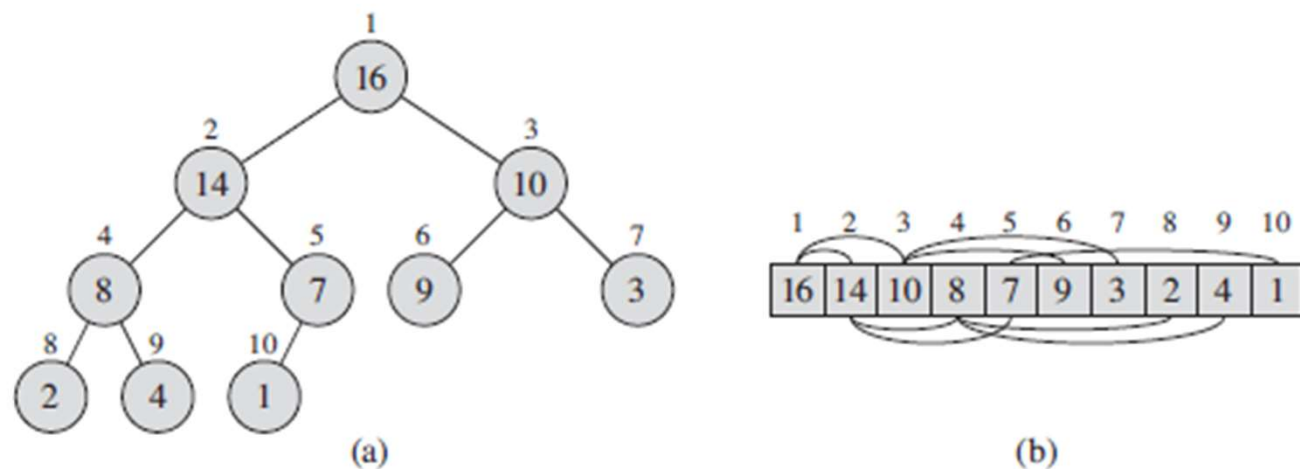
H. İrem Türkmen

Damon Wischik, Computer Laboratory, Cambridge  
University. Lent Term 2021

# HEAPS

- Heaps, which are a type of specialized tree-based data structure, are primarily known for their efficient implementation of priority queues.
- Additional areas where heaps find usage are:
  - **Heap Sort**
  - **Graph Algorithms (Dijkstra's Algorithm, Prim)**
  - **Median Maintenance**
  - **Memory Management**

# BINARY- HEAPS



**Figure 6.1** A max-heap viewed as (a) a binary tree and (b) an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

PARENT( $i$ )

1 return  $\lfloor i/2 \rfloor$

LEFT( $i$ )

1 return  $2i$

RIGHT( $i$ )

1 return  $2i + 1$

# Max / Min Heap Property

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a *heap property*, the specifics of which depend on the kind of heap. In a *max-heap*, the *max-heap property* is that for every node  $i$  other than the root,

$$A[\text{PARENT}(i)] \geq A[i] ,$$

A *min-heap* is organized in the opposite way;  
the *min-heap property* is that for every node  $i$  other than the root,  
 $A[\text{PARENT}(i)] \leq A[i]$  .

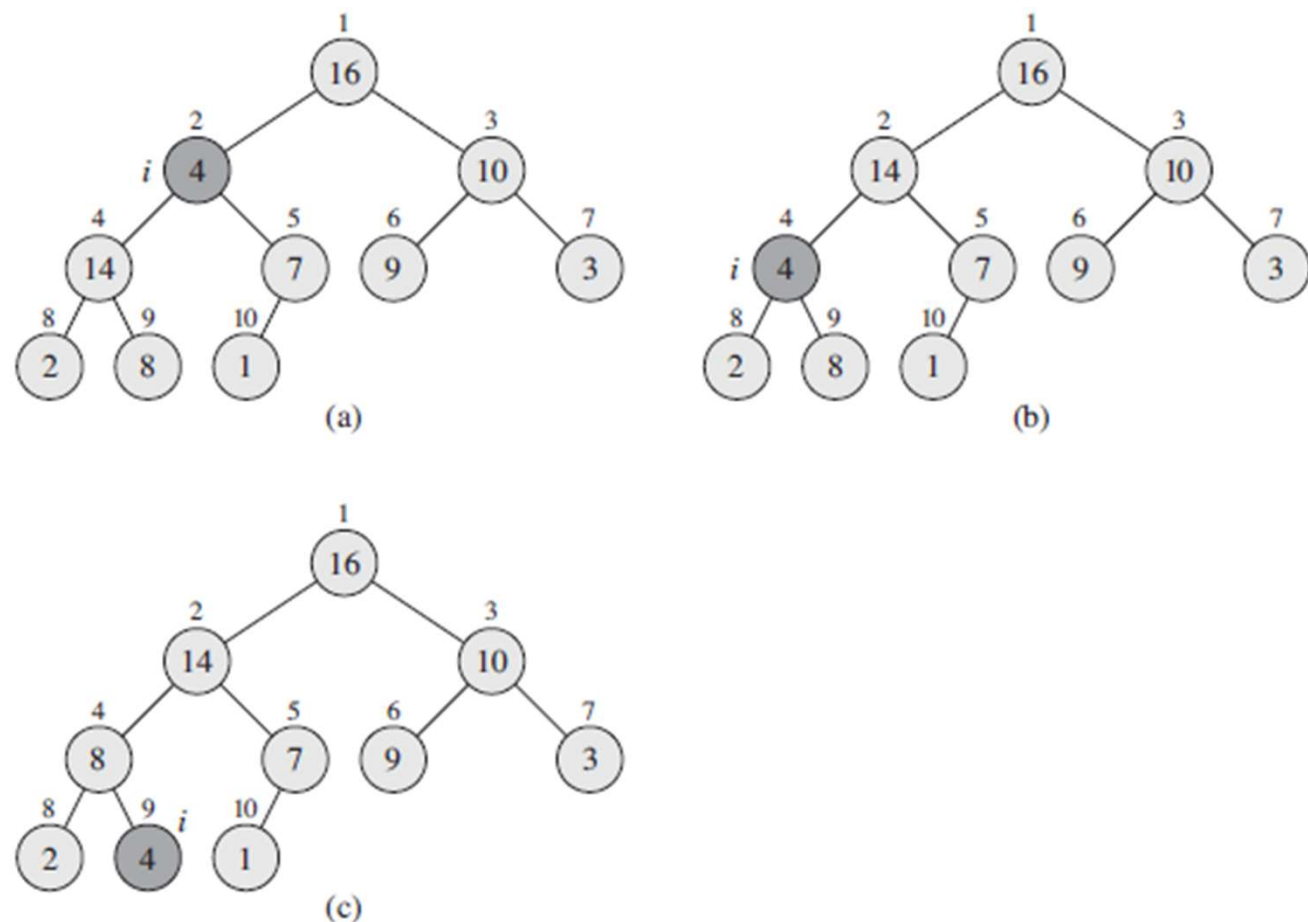
The smallest element in a min-heap is at the root.

# Maintaining the heap property

MAX-HEAPIFY( $A, i$ )

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4       $\text{largest} = l$ 
5  else  $\text{largest} = i$ 
6  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
7       $\text{largest} = r$ 
8  if  $\text{largest} \neq i$ 
9      exchange  $A[i]$  with  $A[\text{largest}]$ 
10     MAX-HEAPIFY( $A, \text{largest}$ )
```

# MAX-HEAPIFY

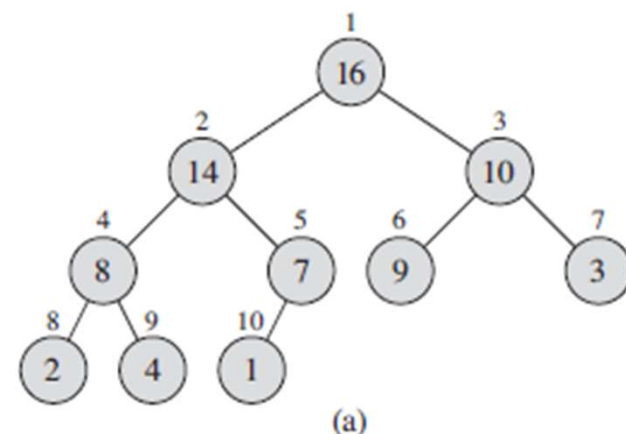


**Figure 6.2** The action of  $\text{MAX-HEAPIFY}(A, 2)$ , where  $A.\text{heap-size} = 10$ . (a) The initial configuration, with  $A[2]$  at node  $i = 2$  violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in (b) by exchanging  $A[2]$  with  $A[4]$ , which destroys the max-heap property for node 4. The recursive call  $\text{MAX-HEAPIFY}(A, 4)$  now has  $i = 4$ . After swapping  $A[4]$  with  $A[9]$ , as shown in (c), node 4 is fixed up, and the recursive call  $\text{MAX-HEAPIFY}(A, 9)$  yields no further change to the data structure.

# BUILD-MAX-HEAP

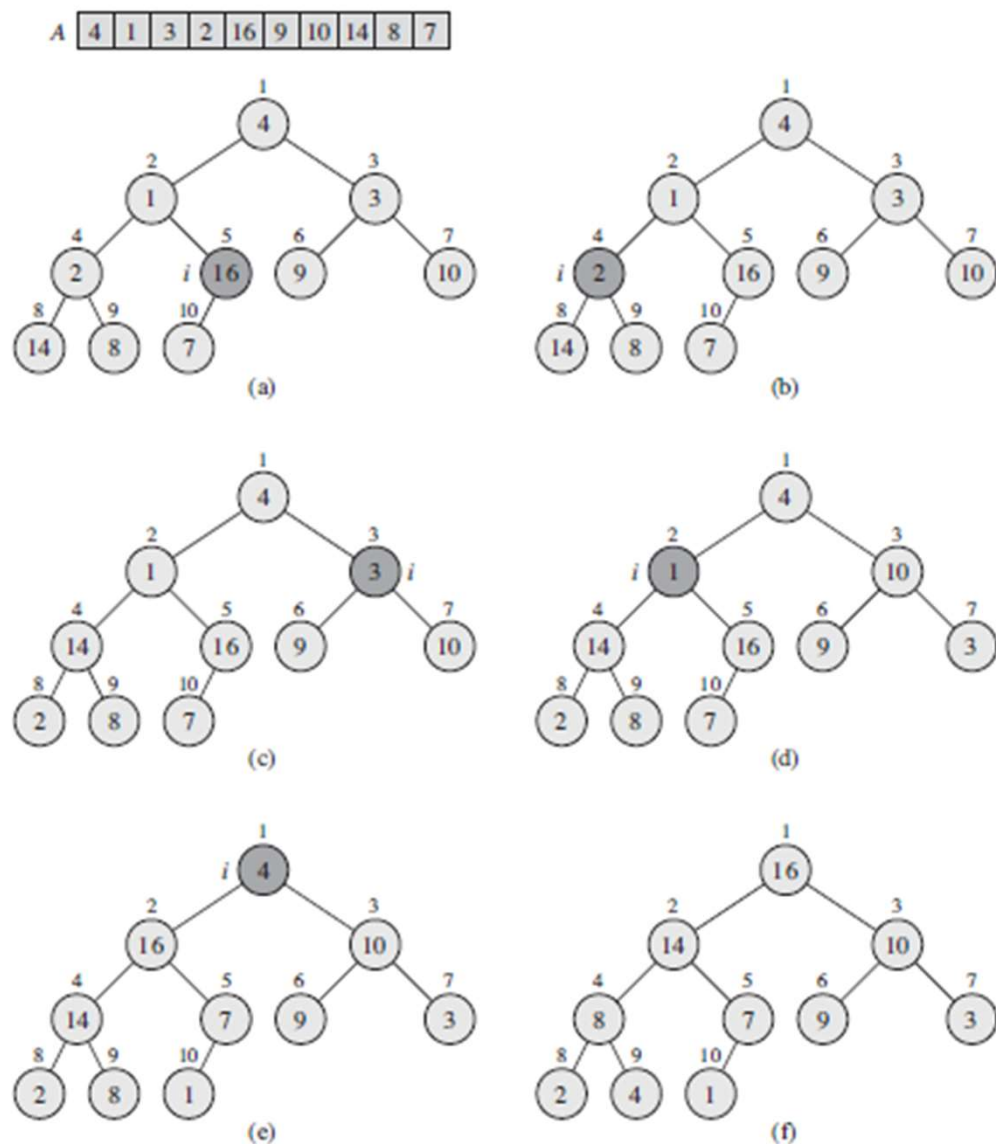
**BUILD-MAX-HEAP(*A*)**

```
1  A.heap-size = A.length
2  for i =  $\lfloor A.length/2 \rfloor$  downto 1
3      MAX-HEAPIFY(A, i)
```



**Initialization:** Prior to the first iteration of the loop,  $i = \lfloor n/2 \rfloor$ . Each node  $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$  is a leaf and is thus the root of a trivial max-heap.





**Figure 6.3** The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. (a) A 10-element input array  $A$  and the binary tree it represents. The figure shows that the loop index  $i$  refers to node 5 before the call MAX-HEAPIFY( $A, i$ ). (b) The data structure that results. The loop index  $i$  for the next iteration refers to node 4. (c)–(e) Subsequent iterations of the for loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. (f) The max-heap after BUILD-MAX-HEAP finishes.



# PRIORITY QUEUES

- All of the work in this section is a build-up to a advanced implementation of the Priority Queue, called the Fibonacci Heap.

**AbstractDataType** PriorityQueue:

*# Holds a dynamic collection of items.*

*# Each item has a value/payload v, and a key/priority k.*

*# Extract the item with the smallest key*

`Pair<Key, Value> popmin()`

*# Add v to the queue, and give it key k*

`push(Value v, Key k)`

*# For a value already in the queue, give it a new (lower) key*

`decreasekey(Value v, Key newk)`

*# Sometimes we also include methods for:*

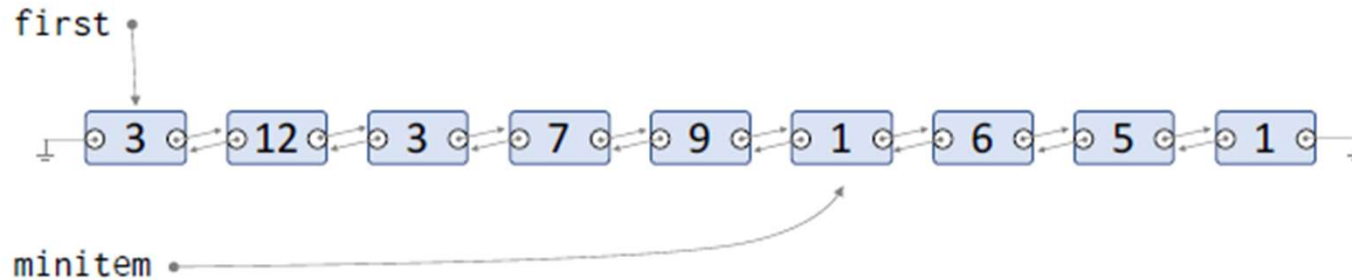
*# merge two priority queues*

*# delete a value*

*# peek at the item with smallest key, without removing it*

# LINKED LIST PRIORITY QUEUE

Here's a very simple priority queue. It uses a doubly-linked list to store all the items, and it also keeps a pointer to the smallest item.



`push(v, k)` is  $O(1)$ :

just attach the new item to the front of the list, and if  $k < \text{minitem.key}$  then update `minitem`

`decreasekey(v, newk)` is also  $O(1)$ :

update `v`'s key, and if  $\text{newk} < \text{minitem.key}$  then update `minitem`

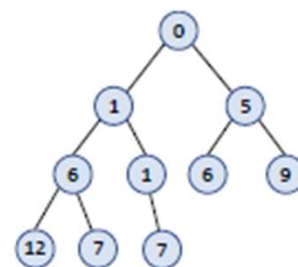
`popmin()` is  $O(n)$ :

we can remove `minitem` in  $O(1)$  time, but to find the new `minitem` we have to traverse the entire list.

# PRIORITY QUEUES

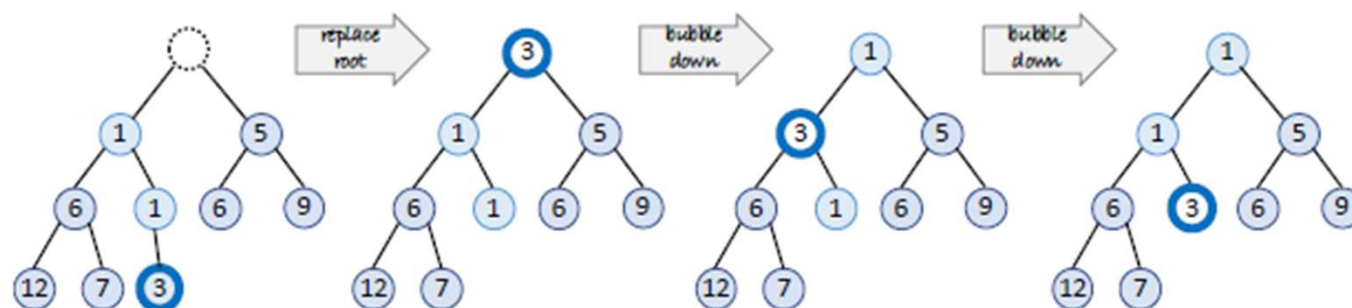
	popmin	push	decreasekey
binary heap	$O(\log N)$	$O(\log N)$	$O(\log N)$
binomial heap	$O(\log N)$	$O(1)$ amortized	$O(\log N)$
linked list	$O(N)$	$O(1)$	$O(1)$
Fibonacci heap	$O(\log N)$ amortized	$O(1)$ amortized	$O(1)$ amortized

# BINARY HEAP

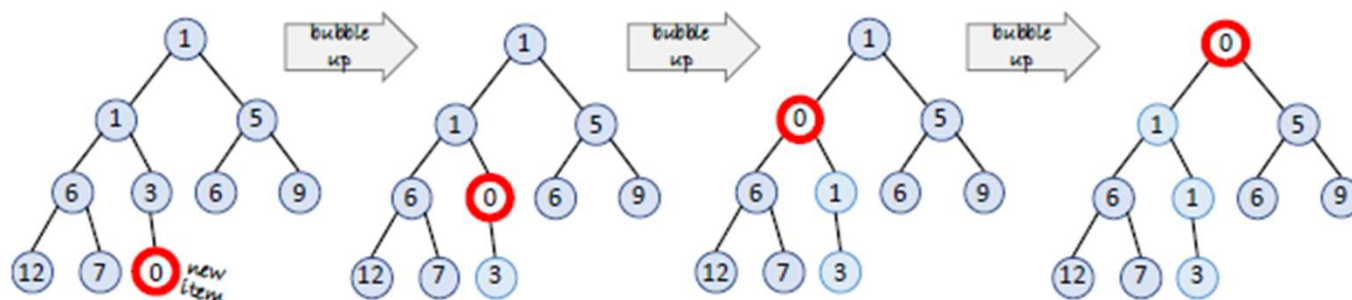


The heap property:  
each node's key is  $\leq$  those  
of its children

To implement **popmin** we extract the root item, replace the root by the end element, and then bubble it far enough down so as to satisfy the heap property. The number of bubble-down steps is limited by the height of the tree, so **popmin** is  $O(\log n)$ .

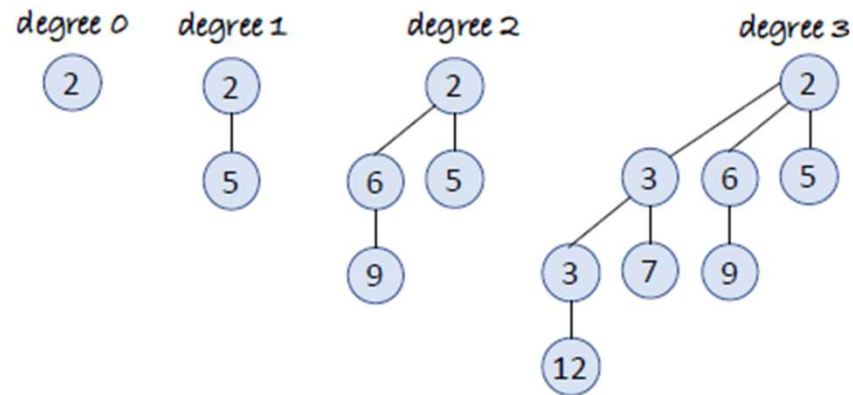


To implement **push** we append the new item to the very end, and then bubble it far enough up the tree so as to satisfy the heap property. Again, the number of bubble-up steps is  $O(\log n)$ . And **decreasekey** is very similar.

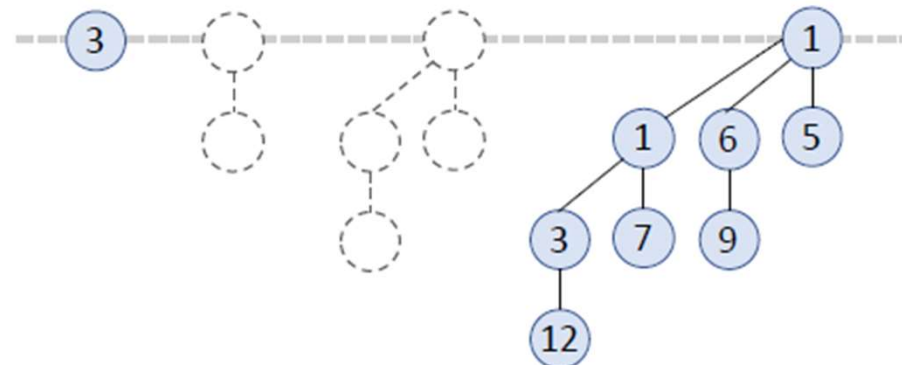


# Mergeable Heaps: BINOMIAL HEAP

A *binomial tree of degree 0* is a single node. A *binomial tree of degree  $k$*  is a tree obtained by combining two binomial trees of degree  $k - 1$ , by appending one of the trees to the root of the other.

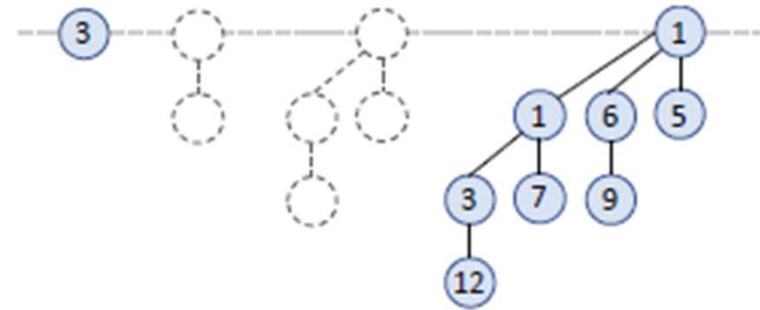


A *binomial heap* is a collection of binomial trees, at most one for each tree degree, each obeying the heap property i.e. each node's key is  $\leq$  those of its children. Here is a binomial heap consisting of one binomial tree of degree 0, and one of degree 3. (The dotted parts in the middle indicate 'there is no tree of degree 1 or 2'.)





# BINOMIAL HEAP



Here are some basic properties of binomial trees and heaps.

1. A binomial tree of degree  $k$  has  $2^k$  nodes
2. A binomial tree of degree  $k$  has height  $k$
3. In a binomial tree of degree  $k$ , the root node has  $k$  children (which is why we call it 'degree')
4. In a binomial tree of degree  $k$ , the root node's  $k$  children are binomial trees of all the degrees  $k - 1, k - 2, \dots, 0$ .
5. In a binomial heap with  $n$  nodes, the 1s in the binary expansion of the number  $n$  correspond to the degrees of trees contained in the heap. For example, a heap with 9 nodes (binary  $1001 = 2^3 + 2^0$ ) has one tree of degree 3 and one tree of degree 0.
6. If a binomial heap contains  $n$  nodes, it contains  $O(\log n)$  binomial trees, and the largest of those trees has degree  $O(\log n)$ .

# BINOMIAL HEAP

`popmin()` is  $O(\log n)$ :

First scan the roots of all the trees in the heap, at cost  $O(\log n)$  since there are that many trees, to find which root to remove. Cut it out from its tree. Its children form a binomial heap, by property 4. Merge this heap with what remains of the original one, as described below, at cost  $O(\log n)$ .

`decreasekey(v, newk)` is  $O(\log n)$ :

Proceed as with a normal binary heap, applied to the tree to which  $v$  belongs. The entire heap has  $O(n)$  nodes, so this tree has  $O(n)$  nodes and height  $O(\log n)$ , so the cost of `decreasekey` is  $O(\log n)$ .

`merge(h1, h2)` is  $O(\log n)$ :

To merge two binomial heaps, start from degree 0 and go up, as if doing binary addition, but instead of adding digits in place  $k$  we merge binomial trees of degree  $k$ , keeping the tree with smaller root on top. If  $n$  is the total number of nodes in both heaps together, then there are  $O(\log n)$  trees in each heap, and  $O(\log n)$  operations in total.

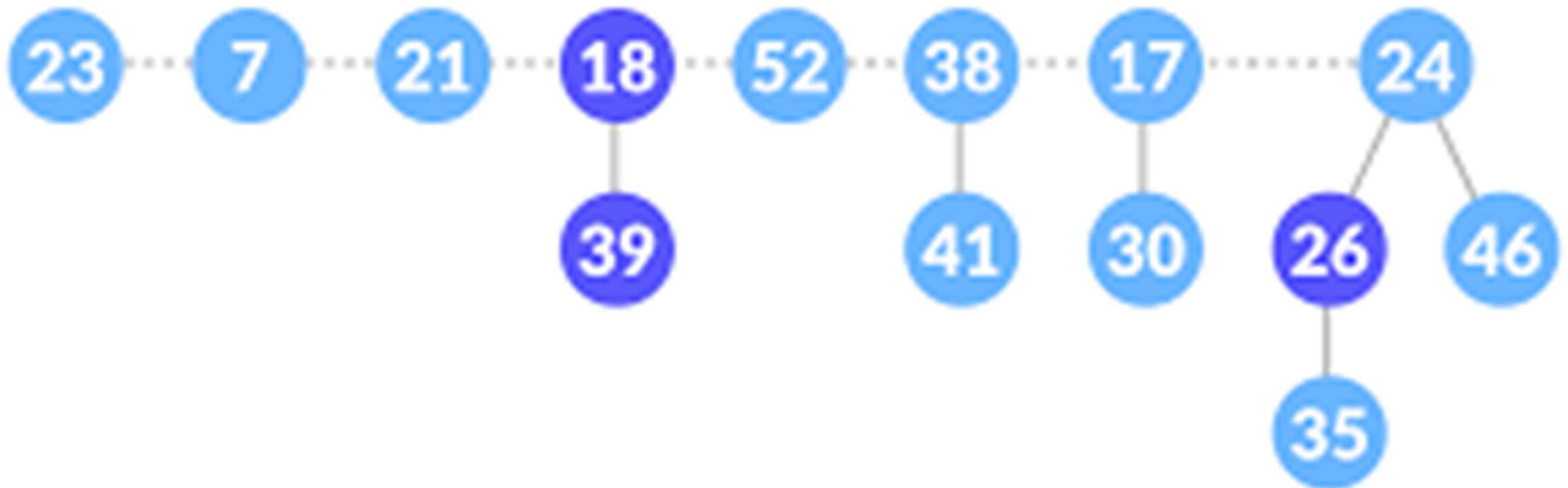
`push(v, k)` is  $O(\log n)$ :

Treat the new item as a binomial heap with only one node, and merge it as described below, at cost  $O(\log n)$ , where  $n$  is the total number of nodes. It can be shown that the amortized cost is  $O(1)$  — see the example sheet.



# Fibonacci Heap

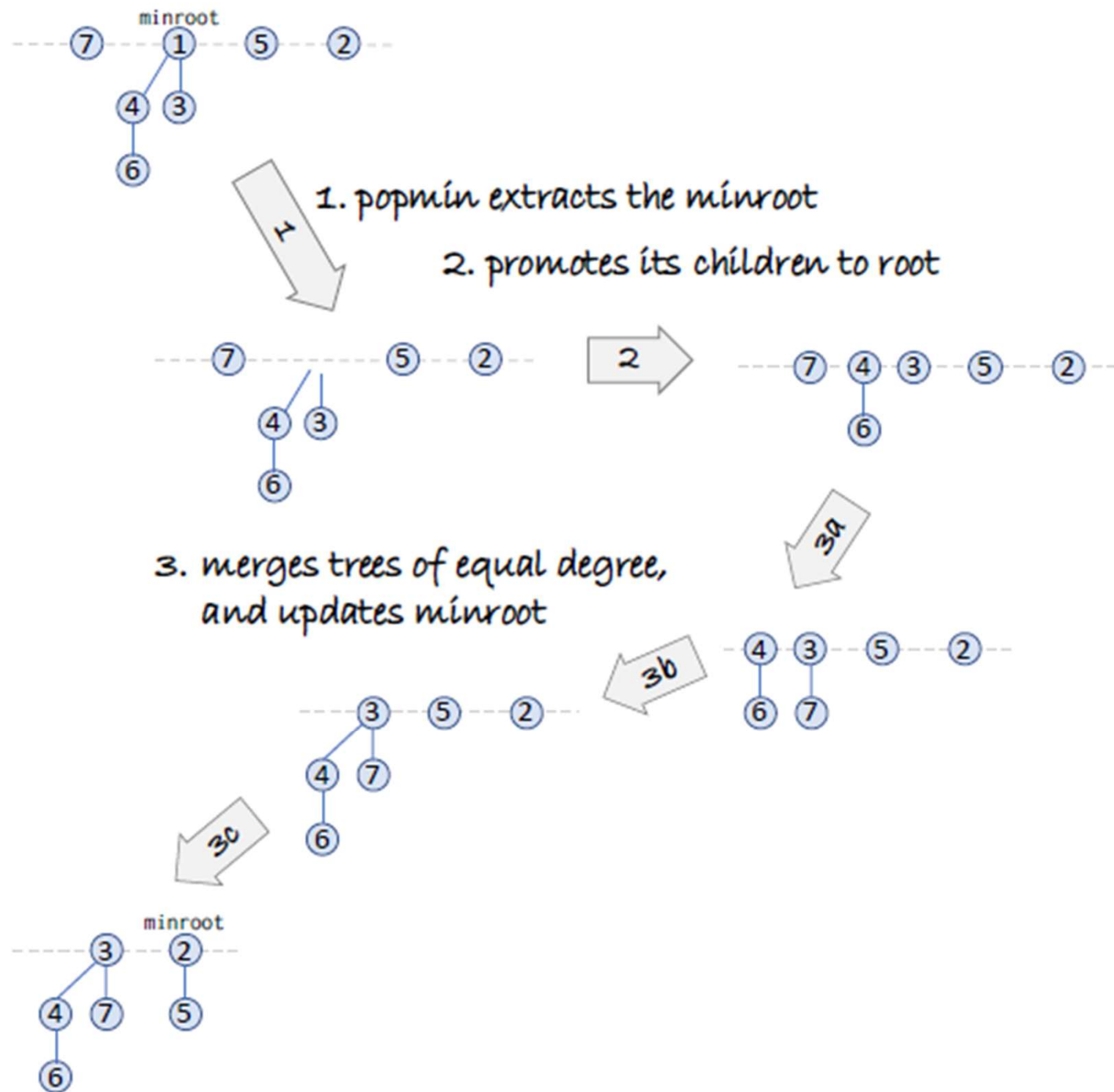
- The Fibonacci heap is a fast priority queue. It was developed specifically to speed up Dijkstra's algorithm.



# Fibonacci Heap

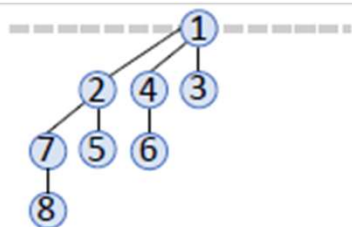
```
1 # Maintain a list of heaps (i.e. store a pointer to the root of each heap)
2 roots = []
3
4 # Maintain a pointer to the smallest root
5 minroot = None
6
7 def push(v, k):
8     create a new heap h consisting of a single item (v, k)
9     add h to the list of roots
10    update minroot if k < minroot.key
11
12 def popmin():
13    take note of minroot.value and minroot.key
14    delete the minroot node, and promote its children to be roots
15    # cleanup the roots
16    while there are two roots with the same degree:
17        merge those two roots, by making the larger root a child of the smaller
18    update minroot to point to the smallest root
19    return the value and key from line 13
```

# PopMin

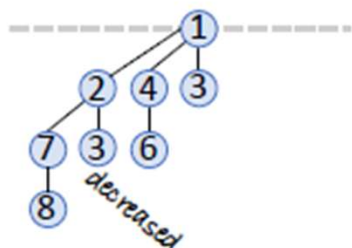


# Decreasekey

- If we can decrease the key of an item in-place (i.e. if its parent is still  $\leq$  the new key), then that's all that decreasekey needs to do.
- If however the node's new key is smaller than its that's all that decreasekey needs to do. If however the node's new key is smaller than its parent, we need to do something to maintain the heap.
- If we just cut out nodes and dump them in the root list, we might end up with trees that are shallow and wide
- This would make popmin very costly, since it has to iterate through all minroot's children.
- To make popmin reasonably fast, we need to keep the maximum degree small. The Fibonacci heap achieves this via two rules:
  1. Lose one child, and you get marked as a 'loser' node.
  2. Lose two children, and you get dumped into the root list (and your mark is removed).



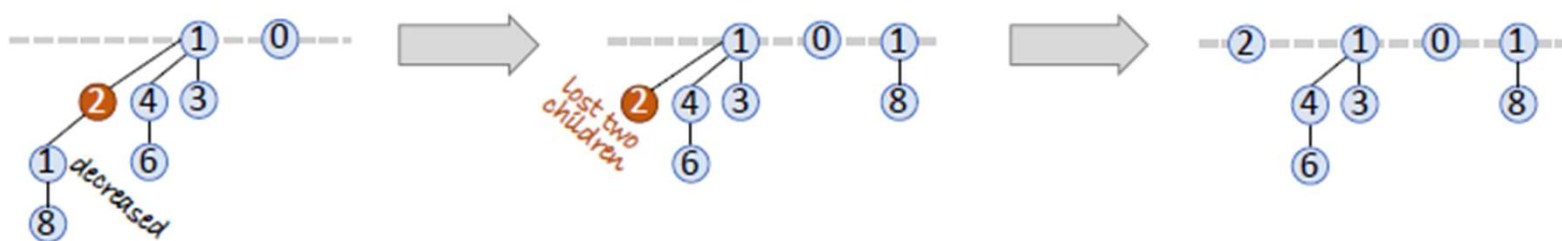
decreasekey from 5 to 3



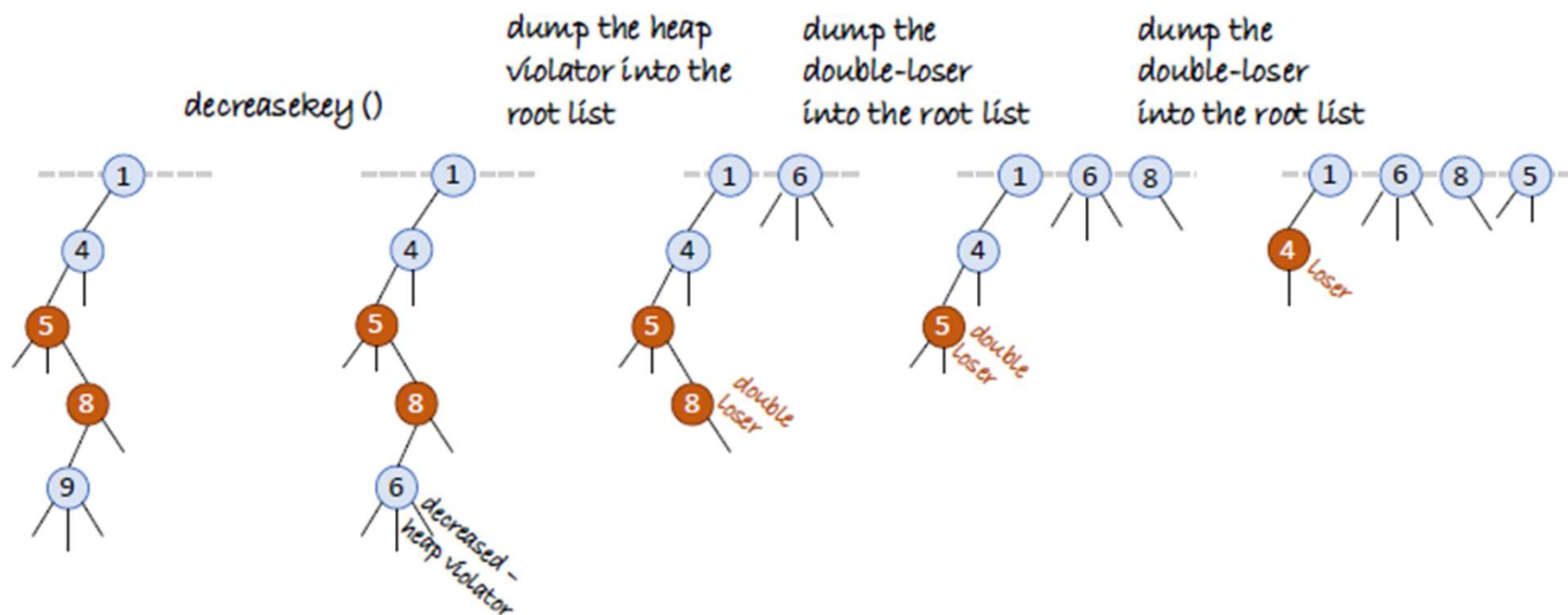
decreasekey again to 0 — move 0 to maintain the heap



decreasekey from 7 to 1 — move 1 to maintain the heap — move the double-loser to root



Here is another example of the operation of decreasekey, this time highlighting what happens if there are multiple loser ancestors.



# Amortized Analysis of Fibonacci Heaps

- Push()
- Decreasekey()
- Popmin()



# Disjoint Sets

- The DisjointSet data structure (also known as union-find or merge-find) is used to keep track of a dynamic collection of items in disjoint sets
- **Union-Find Operations**
- **Graph Algorithms**
- **Image Processing and Segmentation**
- **Network Connectivity**
- **Kruskal's Minimum Spanning Tree Algorithm**

# Kruskal Algorithm

- Lets remember

# Disjoint Sets



First, here's the specification of the DisjointSet abstract data type.

**AbstractDataType** DisjointSet:

*# Holds a dynamic collection of disjoint sets*

*# Return a handle to the set containing an item.*

*# The handle must be stable, as long as the DisjointSet is not modified.*

**Handle** get\_set\_with(**Item** x)

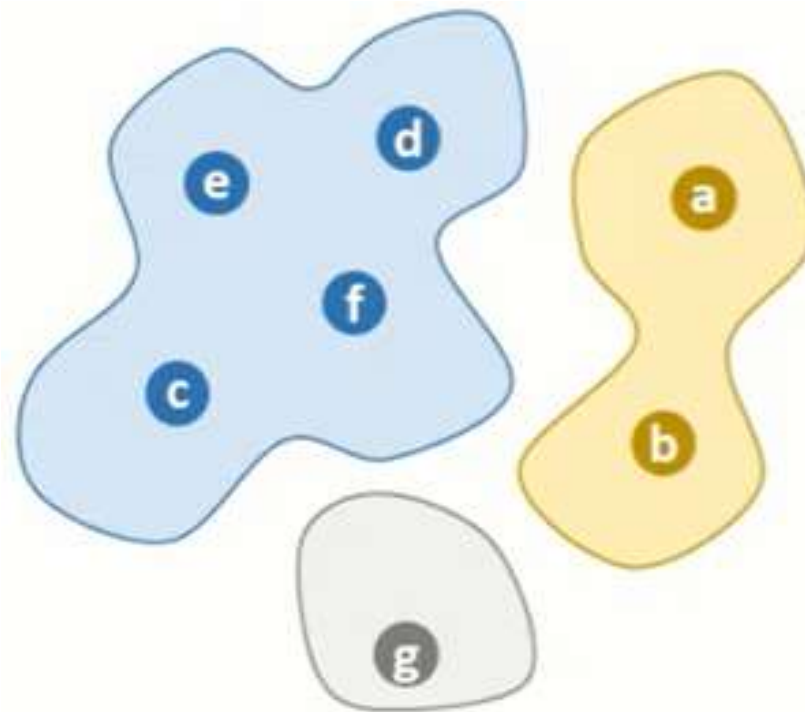
*# Add a new set consisting of a single item (assuming it's not been added already)*

**add\_singleton**(**Item** x)

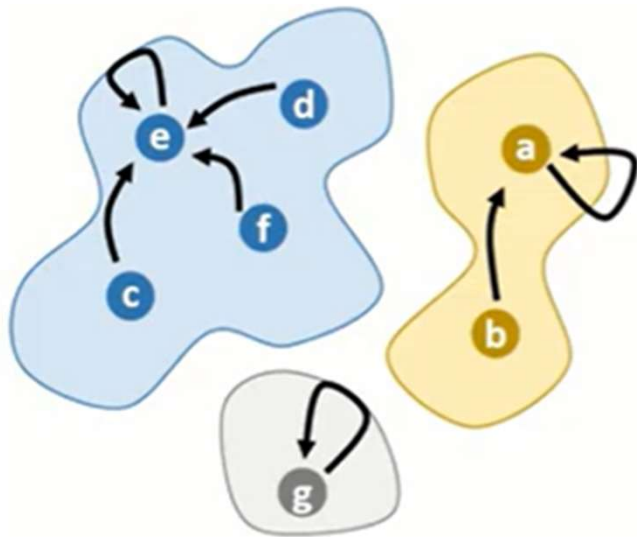
*# Merge two sets into one*

**merge**(**Handle** x, **Handle** y)

# How to implement?



# Implementation 0

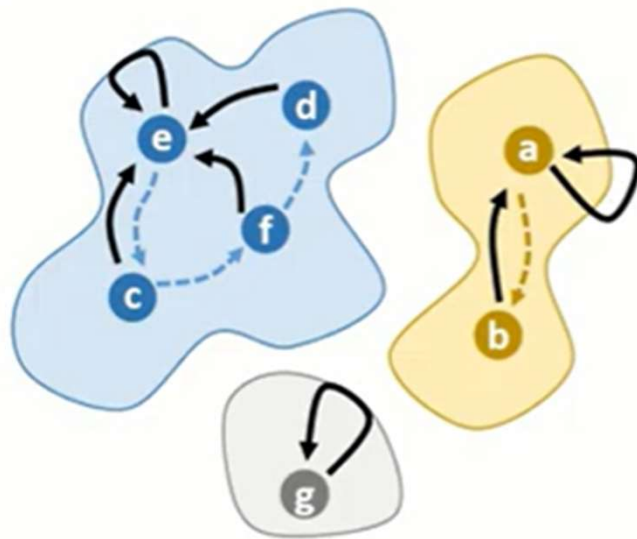


Each item points to a representative item for its set

```
mysets = {a:a, b:a, c:e, d:e, e:e, f:e, g:g}
```

```
def merge(x,y):  
    for every item in the entire collection:  
        if the item's set is y then update it to be x
```

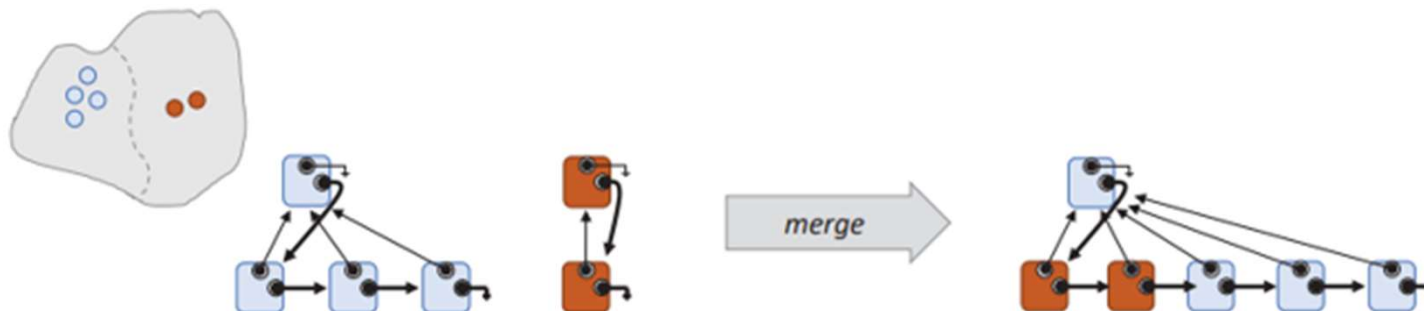
# Implementaion 1 – Flat Forest



## IMPLEMENTATION 1 "FLAT FOREST"

Each item points to a representative item for its set  
Each set has a linked list, starting at the representative

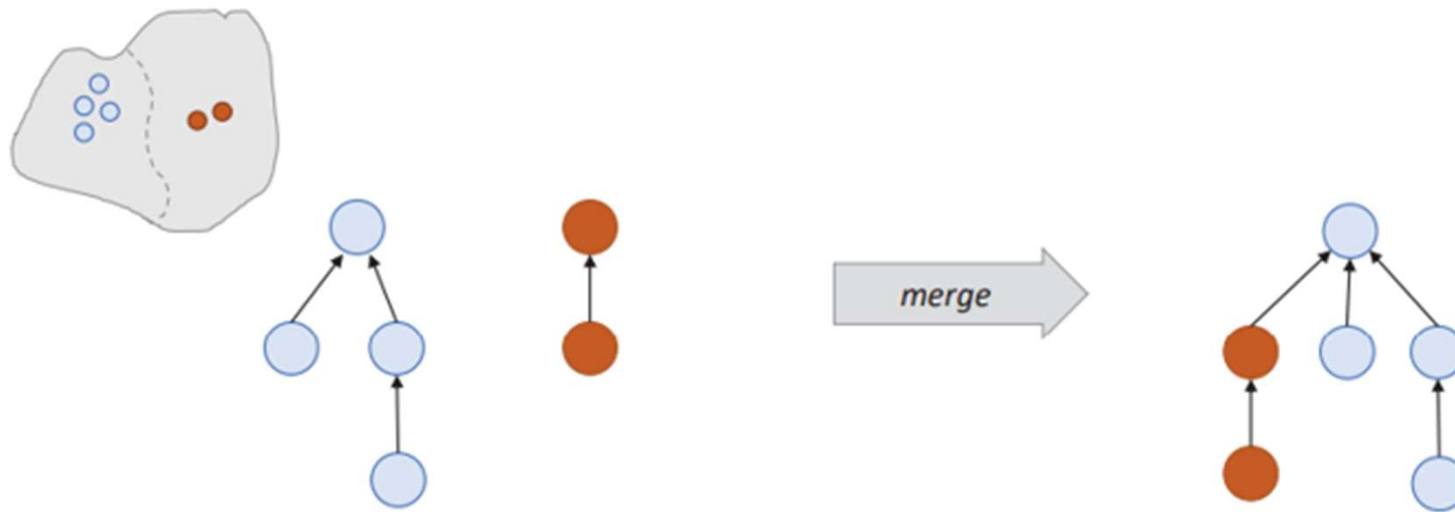
```
def merge(x,y):  
    for every item in set y:  
        update it to belong to set x
```



- What about complexity of merge and get set with for these implementations?
- Wighted-union heuristics..



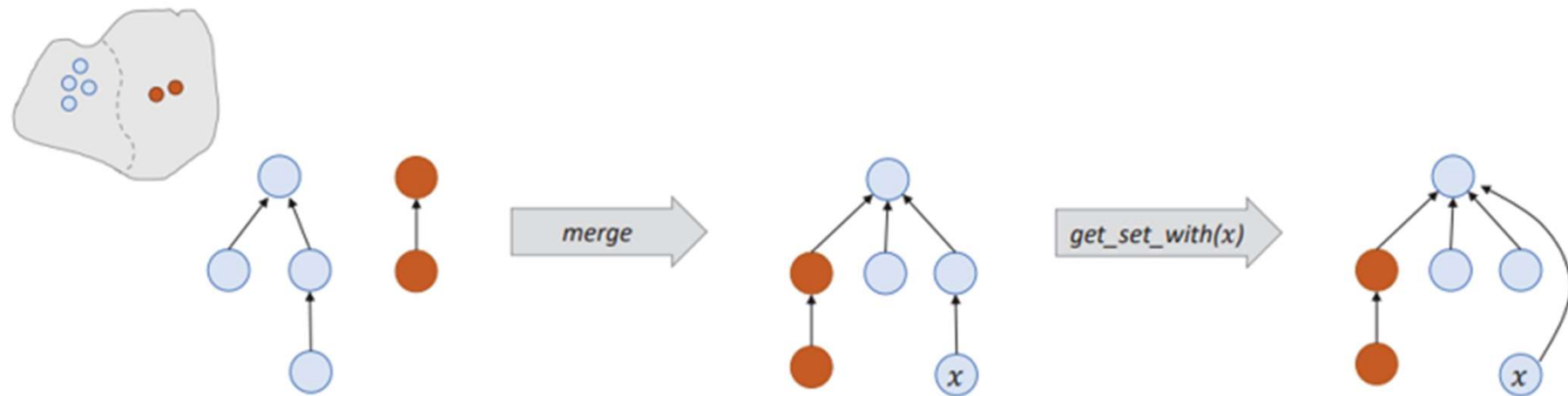
# Implementaion 2 – Deep Forest



- Merge  $O(1)$
- Get\_set\_with  $O(h)$
- Union by rank

# Implementaion 3 – Lazy Forest

- Path -compression



```
def get_set_with(x):  
    walk up the tree from x to the root  
    walk up again, and make all items point to root  
    return this root
```