For advanced data structures like a Python list or a PriorityQueue ...

- We should care about the aggregate cost of a sequence of operations, which might not be as bad as the per-operation worst cases suggest

Fundamental Inequality of Amortization: this defines amortized costs.

- For any sequence of operations (starting from a empty data structure)

$$\text{agg. true cost} \leq \text{agg. amortized cost}$$

- We can obtain amortized costs via a potential function Φ, with

$$\underbrace{c'}_{\text{amortized cost}} = \underbrace{c}_{\text{true cost of op.}} + \Delta\Phi \qquad \Phi \geq 0 \quad \Phi(\text{empty}) = 0$$

- Think of Φ in these ways:
  - — a bank balance, storing up credit to pay for an expensive operation
  - — a measure of the mess in the data structure (that'll have to be cleaned up)
  - — credit stored on parts of the data structure (that'll have to be operated on)

```
class MinList<T>:

    def append(T value):
        # append a new value

    def T min():
        # return the smallest
        # (without removing it)
```

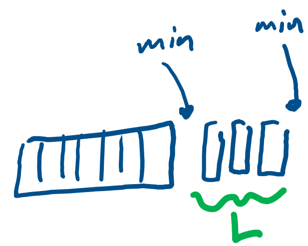
The worst-case cost of min is $O(n)$, where n is the number of items.

QUESTION. What potential function might we use, to show that append and `min` both have amortized cost $O(1)$?

Let $\Phi = L$

Amortized analysis:

- append. $c + \Delta\Phi = O(1) + 1 = O(1)$ am.
- min. $c + \Delta\Phi = O(L) - L = O(1)$ am.



Stage 0

- Use a linked list
- `min` iterates over the entire list

Stage 1

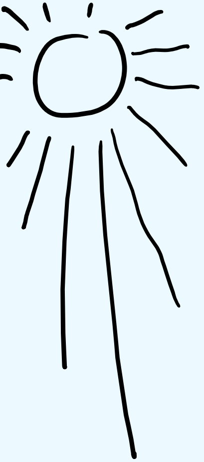
- Use a linked list
- `min` caches its result, so that next time it only needs to iterate over newer values

Stage 2

- Use a linked list
- Store the current minimum, and update it on every append

Stage 3

- `min` caches its result, the same as Stage 1
- ... but we argue it's just as good as Stage 2



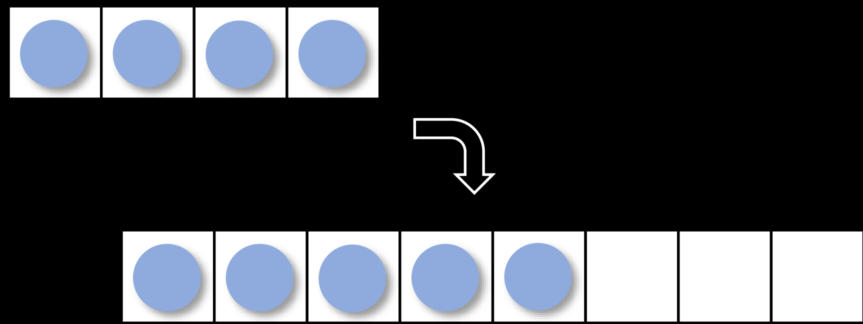
ABSTRACT DATA TYPES

What's important is a correctly specified interface

ALGORITHMS

What's important is good performance

Dynamic array



Store the values in an array.

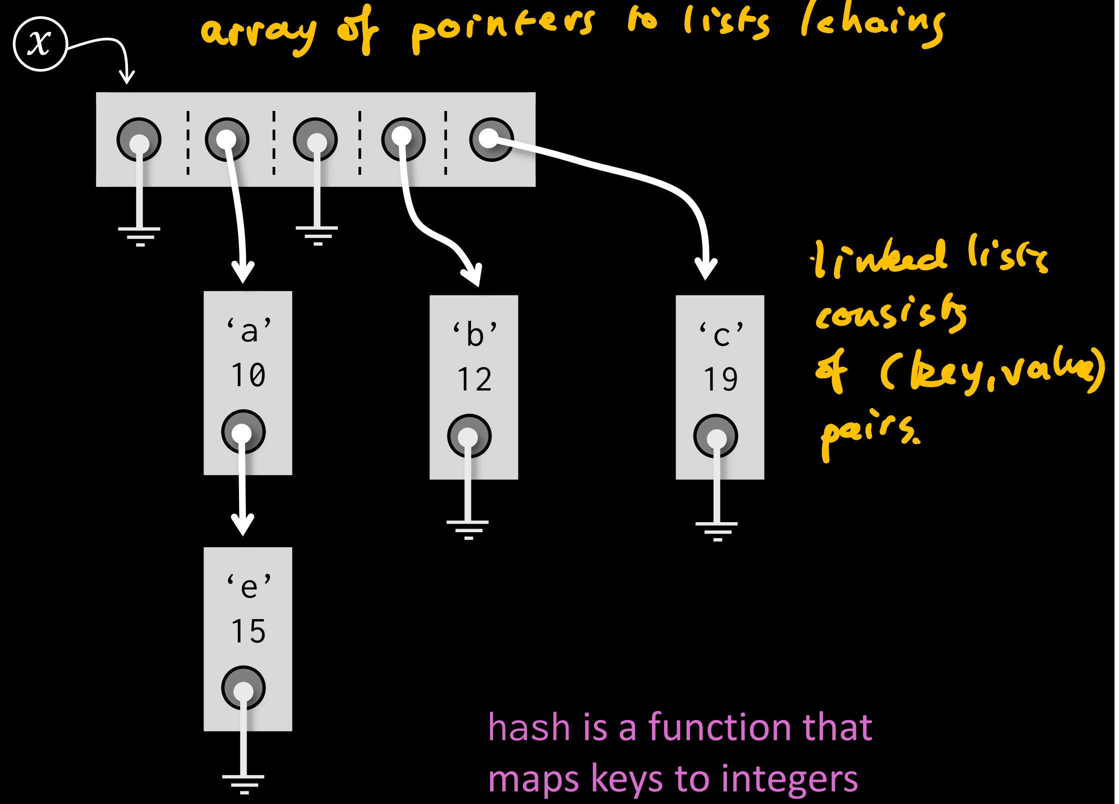
When it gets full, create a new array of double the capacity, and copy items across.

Python list

$$\mathtt{x} = [\cdots]$$

$$\left.\begin{array}{l} \mathtt{x}[i] \\ \mathtt{x.append}(\cdot) \end{array}\right\} \text{both are } O(1) \text{ amortized}$$

4.7 Hash tables with chaining




```
x = array of pointers to lists

def set(k, v):
    b = hash(k) mod len(x)
    chain = x[b]
    if chain contains an item with key k:
        set that item's value to v
    else:
        append (k,v) to chain
```

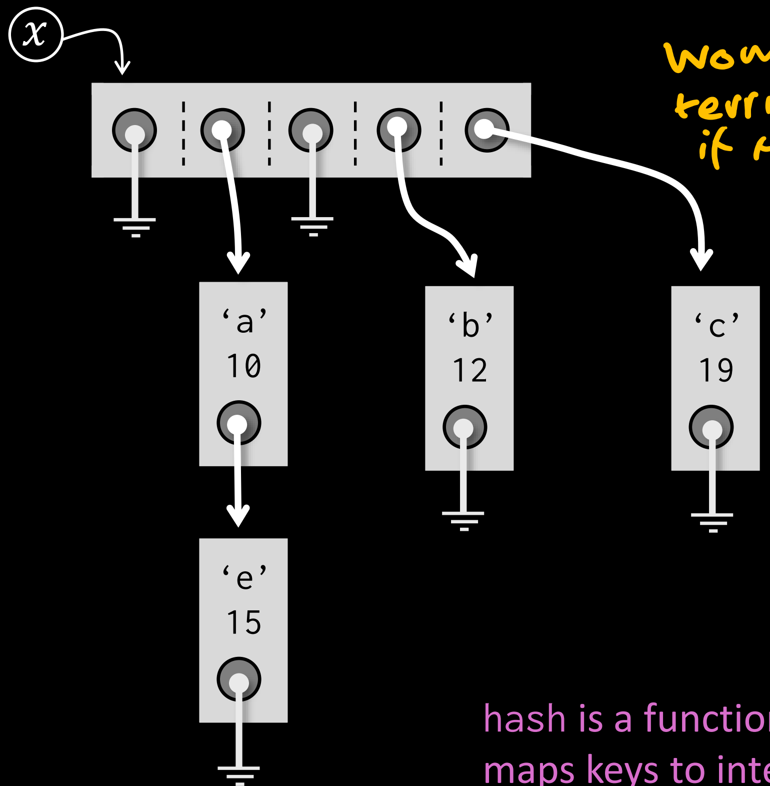
4.2.4 Dictionary

```
x = {'a': 10, 'c': 19, 'b': 12, 'e': 15}
# dictionaries support:
# is_empty(), contains(k), get(k),
# set(k,v), del(k)
```

Python has a built-in hash function.

```
>>> i = 10
>>> hash(i)
10

>>> s = "stoats are awesome"
>>> hash(s)
3267385019077449291
```

4.7 Hash tables with chaining



Won't this have terrible performance if the lists get too long?

hash is a function that maps keys to integers

```
x = array of pointers to lists

def set(k, v):
    b = hash(k) mod len(x)
    chain = x[b]
    if chain contains an item with key k:
        set that item's value to v
    else:
        append (k,v) to chain
```

Define the *load factor* to be $\alpha = n/c$ where n is the number of items stored and c is the capacity of the array i.e. the number of "buckets".

If the hash function is perfect, then every bucket is equally likely, so items will be distributed uniformly. The average size of a chain is then

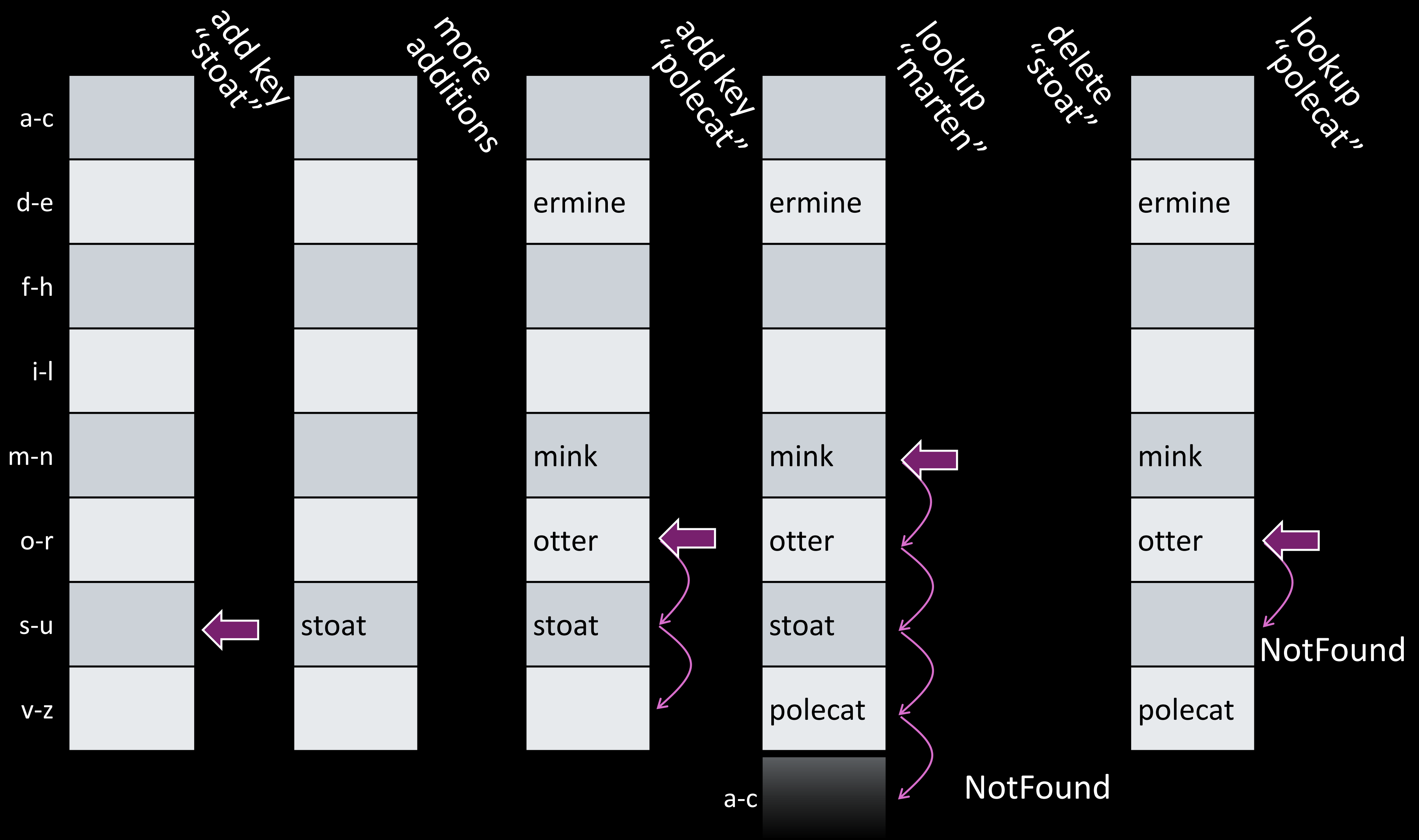
$$\frac{\#\text{items}}{\#\text{buckets}} = \frac{n}{c} = \alpha.$$

For good performance we want to maintain α low. In Python, $\alpha \leq 2/3$.

We can achieve this by using a dynamic array for x. When we need to double its capacity, we'll rehash all the items.

4.7 Hash tables with open addressing

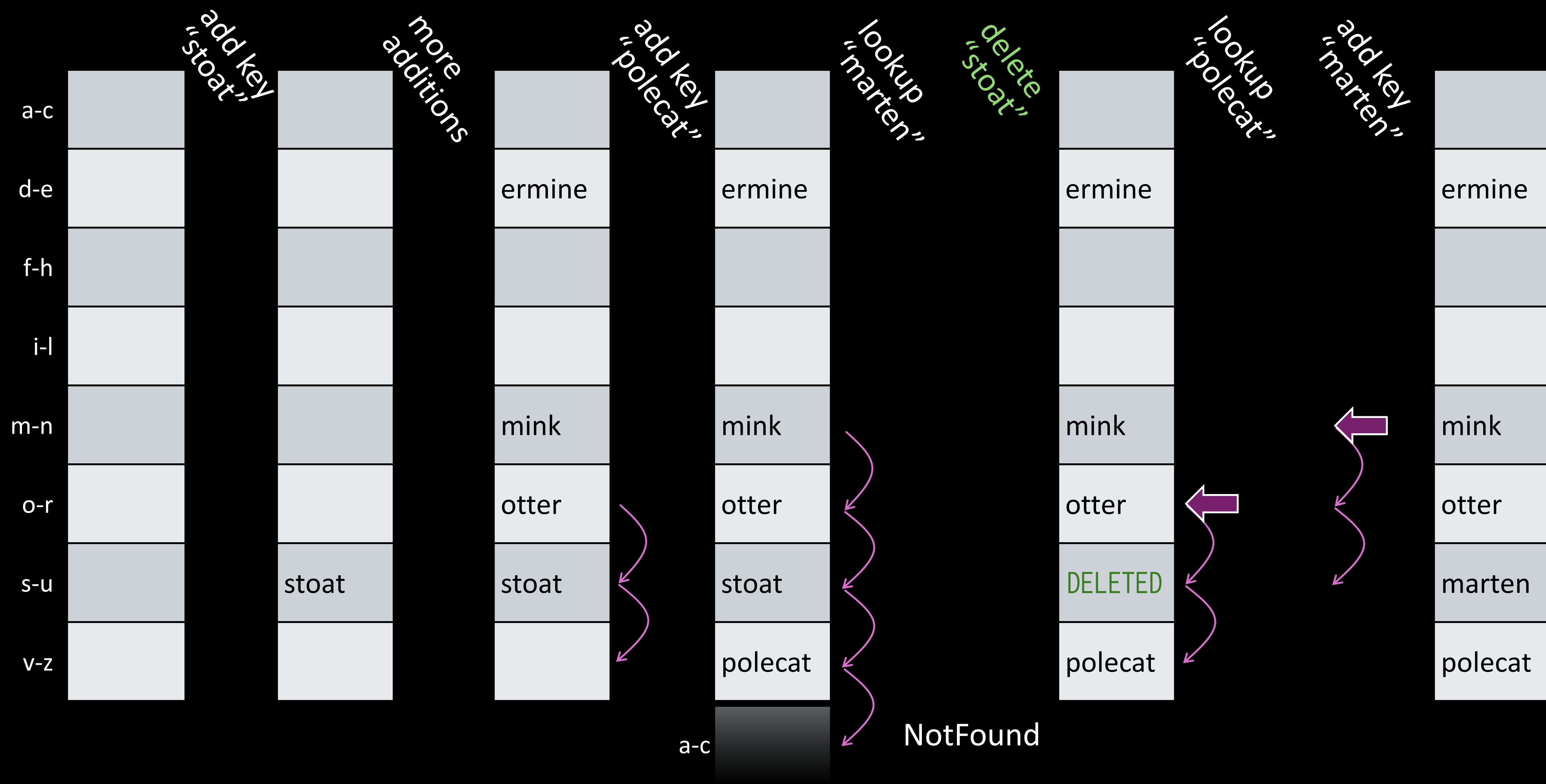
Since we have the technology to dynamically resize arrays, we don't need to use linked lists at all! Just store everything in a big array, and resize + rehash when needed to keep $\alpha \leq 2/3$. (This saves space on pointers, and may help with cache locality.)



We need to be smart about deletions...

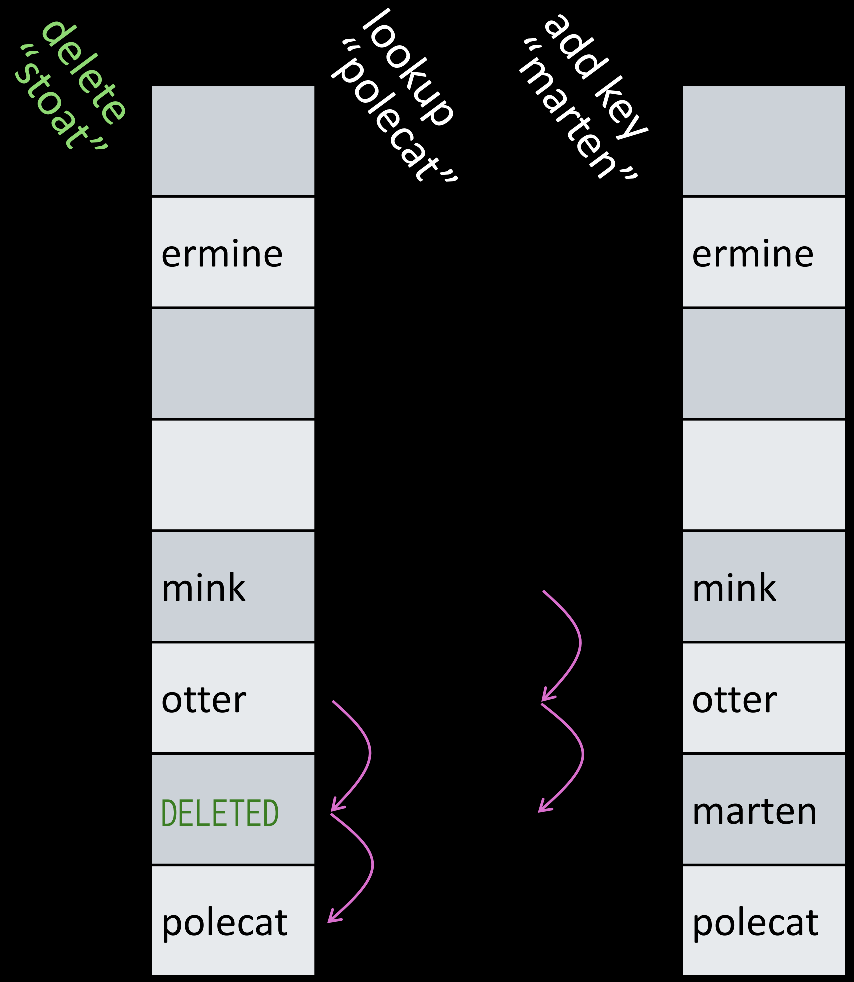
4.7 Hash tables with open addressing

Since we have the technology to dynamically resize arrays, we don't need to use linked lists at all! Just store everything in a big array, and resize + rehash when needed to keep $\alpha \leq 2/3$. (This saves space on pointers, and may help with cache locality.)



4.7 Hash tables with open addressing

EXAMPLE SHEET 3 QUESTION 49
Explain carefully how to implement `delete`, `lookup`, and `add`.



We have used what's called *linear probing*,

`probe(k,j) = (hash(k) + j) mod c # j=attempt number`

Linear probing tends to produce clusters of colliding keys. Better alternatives are *quadratic probing*,

`probe(k,j) = (hash(k) + c*j + d*j²) mod c`

or *double hashing*,

`probe(k,j) = (hash(k) + j*hash₂(k)) mod c`

Scientists Find Optimal Balance o

https://www.quant...

Quanta magazine

ALGORITHMS

Scientists Find Optimal Balance of Data Storage and Time

By STEVE NADIS

February 8, 2024

Seventy years after the invention of a data structure called a hash table, theoreticians have found the most efficient possible configuration for it.

5

In a 1957 paper published in the *IBM Journal of Research and Development*, W. Wesley Peterson identified the main technical challenge that hash tables pose: They need to be fast, meaning that they can quickly retrieve the necessary information. But they also need to be compact, using as little memory as possible. These twin objectives are fundamentally at odds. Accessing and modifying a database can be done more quickly when the hash table has more memory; and operations become slower in hash tables that use less space. Ever since Peterson laid out this challenge, researchers have tried to find the best balance between time and space.

Computer scientists have now mathematically proved that they have found the optimal trade-off. The solution came from a pair of recent papers that complemented each other. “These papers resolve the long-standing open question about the best possible space-time trade-offs, yielding deeply surprising results that I expect will have a significant impact for many years to come,” said Michael Mitzenmacher, a computer scientist at Harvard University who was not involved in either study.

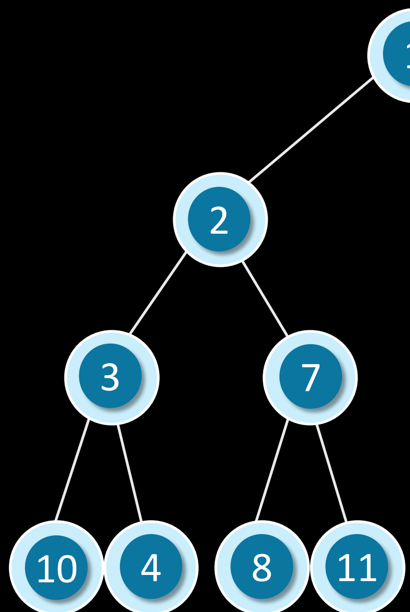
SECTION 7.5

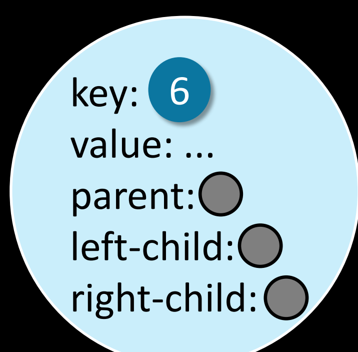
Three priority queues

4.8.1 Binary heap

A *binary heap* is an almost-full binary tree that satisfies the heap property (everywhere in the tree, parent key $\leq$ child keys).

Operations have cost $O(\log n)$, where n is the size of the heap.



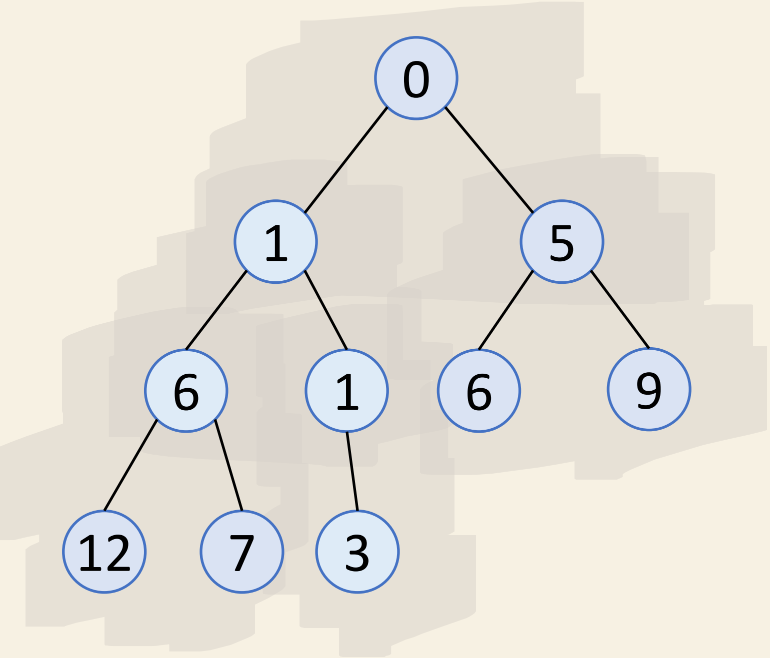


4.8 Priority queue

A *priority queue* holds a dynamic collection of items. Each item has a value v, and a key/priority k.

```
interface PriorityQueue<K,V>:

    boolean is_empty()

    # extract the item with the smallest key
    Pair<K,V> popmin()

    # add a new item, and set its key
    push(K key, V value)

    # for an existing item, give it a new (lower) key
    decreasekey(V value, K newkey)
```

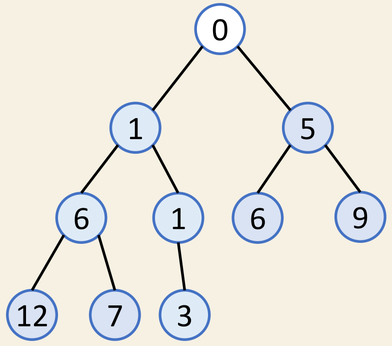
The binary heap



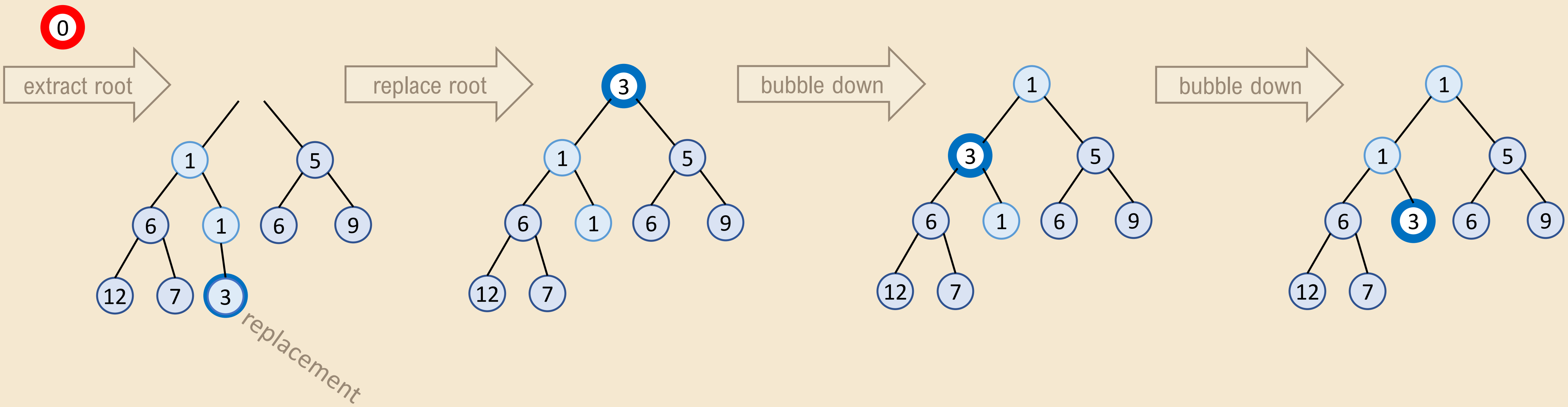
The heap property

every node's key is ≤ those of its children

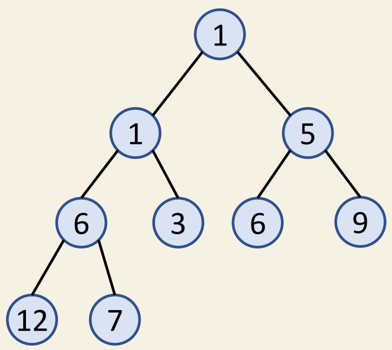
The binary heap



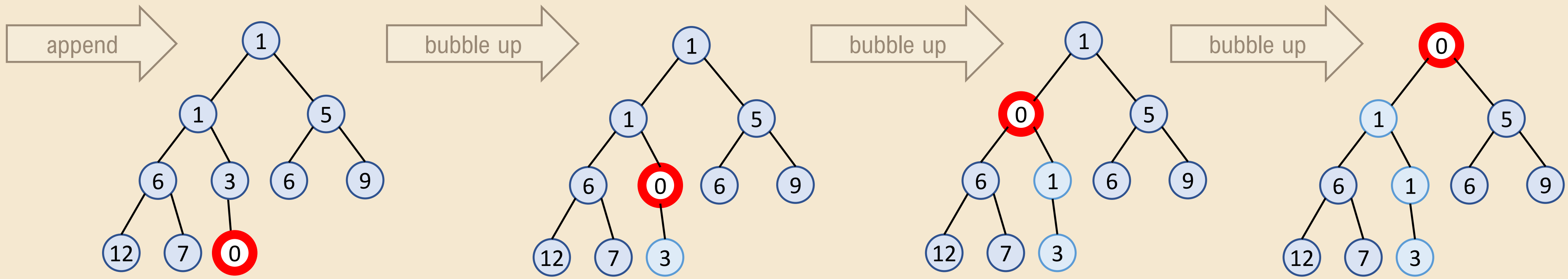
popmin()



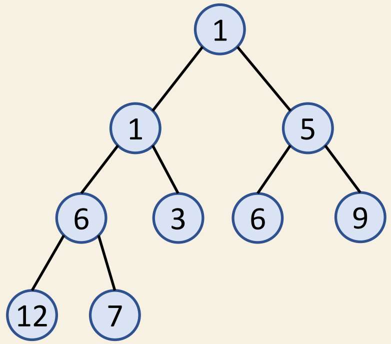
The binary heap



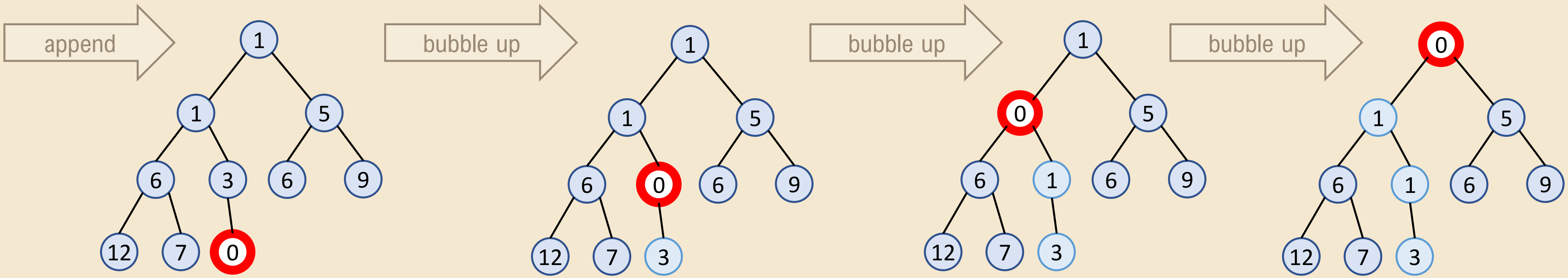
push(*new item*)



The binary heap



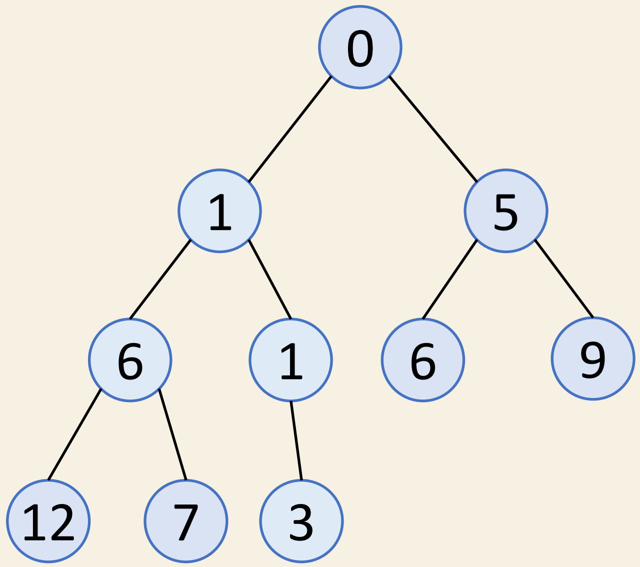
push(*new item*)



decreasekey(*item, new key*)

similar

The binary heap



SHAPE LEMMA

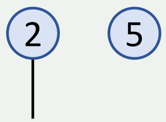
The height is $O(\log N)$
where N is the number of items in the heap

COMPLEXITY ANALYSIS

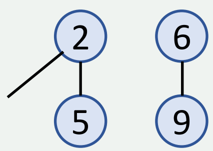
All operations are $O(\log N)$

Binomial trees

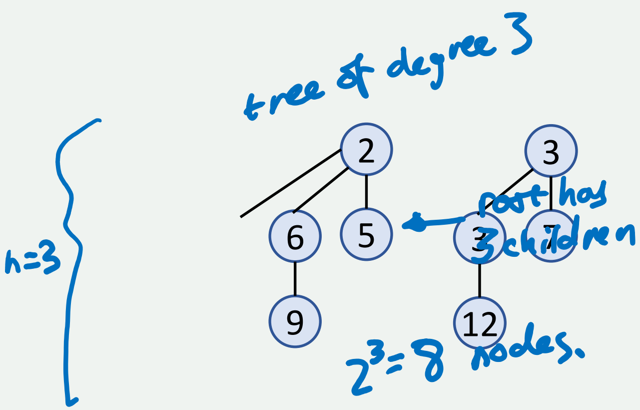
a tree of degree 0



two trees of degree 0
merge to give a tree of degree 1



two trees of degree 1
merge to give a tree of degree 2



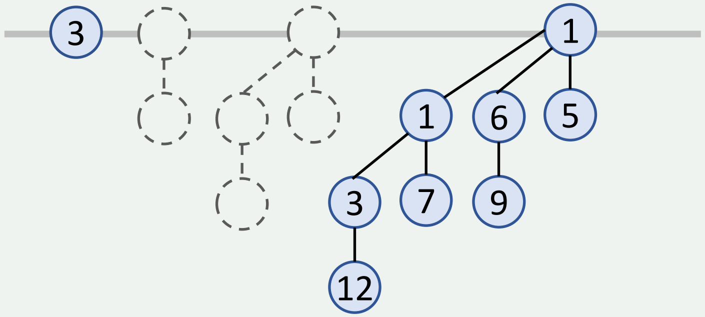
two trees of degree 2
merge to give a tree of degree 3

It's easy to prove by induction that a binomial tree of degree k has

- height k
- 2^k nodes
- k children at the root, all of them binomial trees

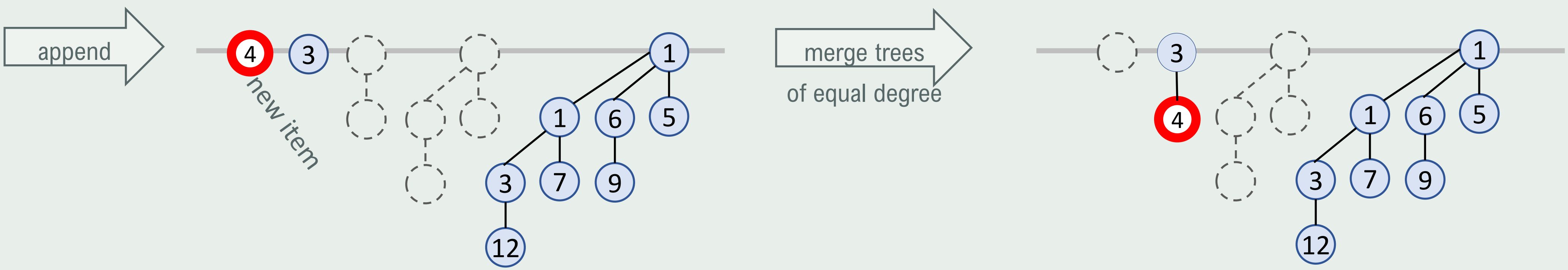
chapter 4.8
& page 63

The binomial heap

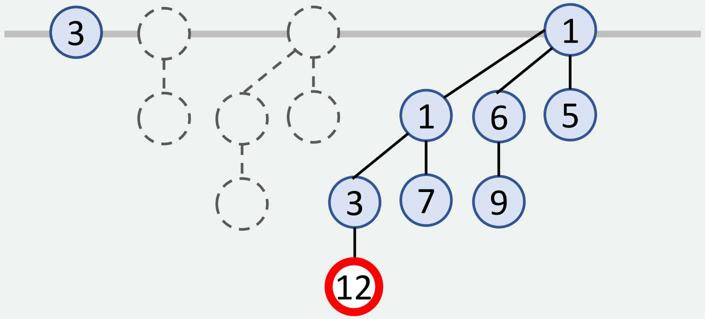


- a list of binomial trees, with at most one of each degree
- each tree is a heap

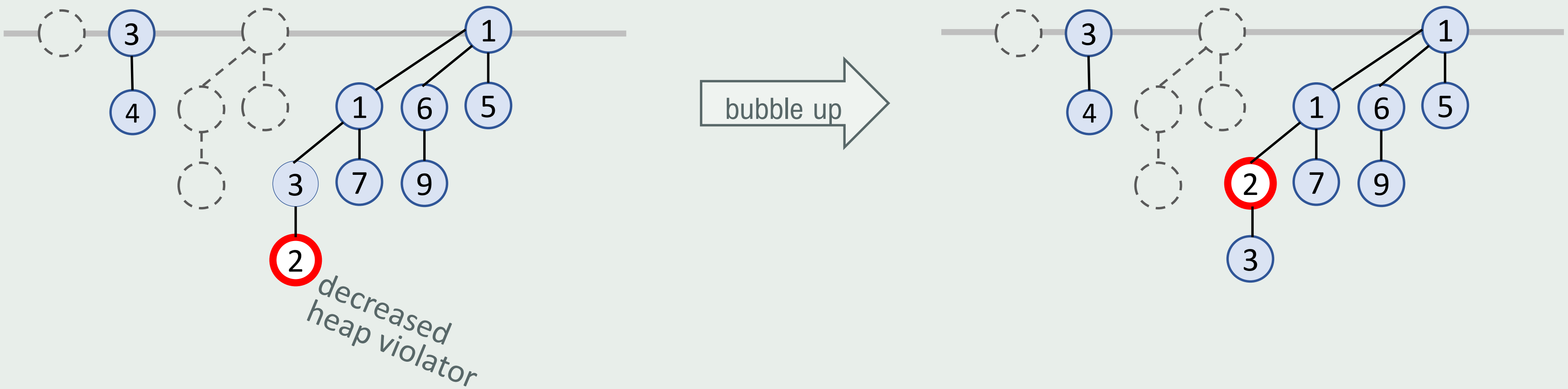
push(*new item*)



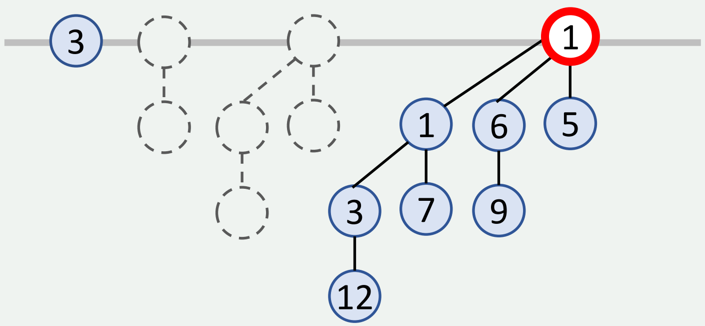
The binomial heap



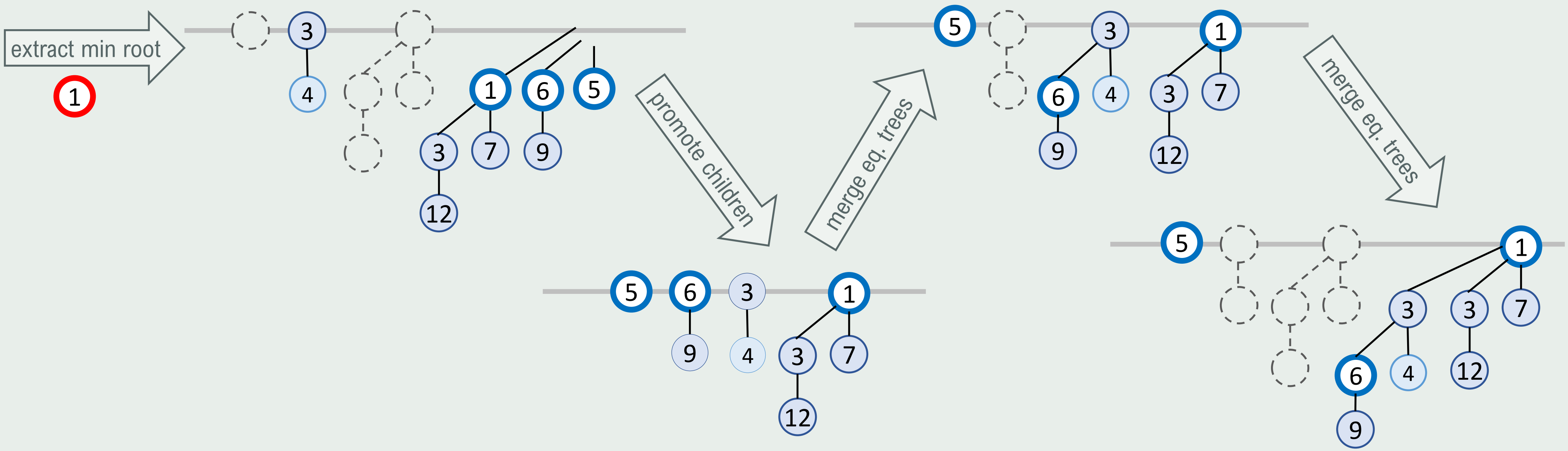
decreasekey(*item, new key*)



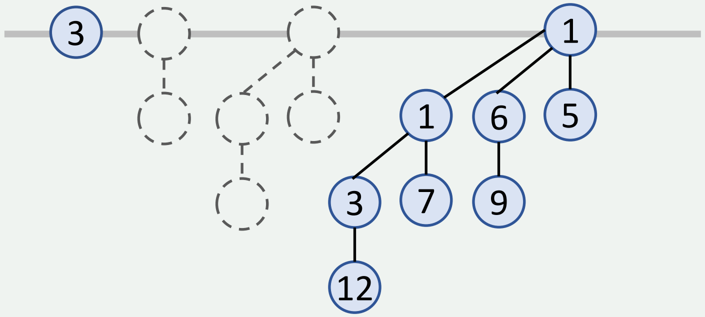
The binomial heap



popmin()



The binomial heap

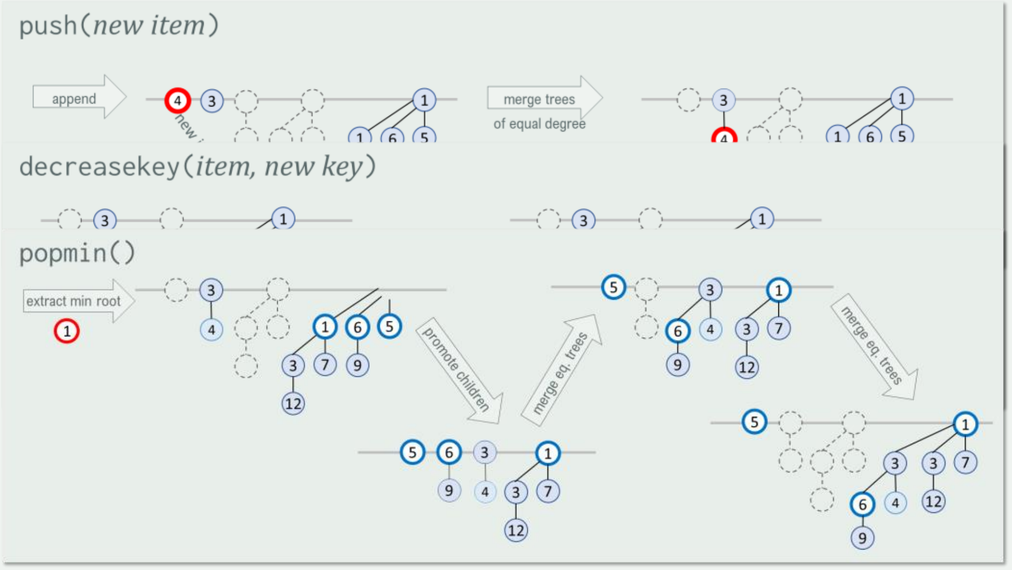


SHAPE THEOREM

- A binomial tree of degree k has 2^k items and $\boxed{\text{height } k}$
- Hence, in a binomial heap with N items, the binary digits of N tell us which binomial trees are present

$N = 9$ items $= \begin{array}{cccc} 2^0 & 2^1 & 2^2 & 2^3 \\ \hline 1 & 0 & 0 & 1 \end{array}$

$\Rightarrow$ #trees $= O(\log N)$.



COMPLEXITY ANALYSIS

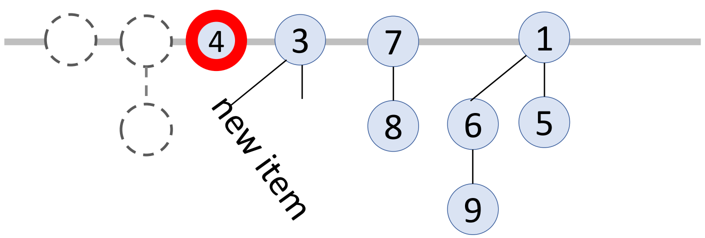
- push() is $O(\log N)$
  we have to merge $O(\log N)$ trees
- decreasekey() is $O(\log N)$
  in the worst case we have to bubble up from the bottom of the largest tree
- popmin() is $O(\log N)$
  scan $O(\log N)$ trees; promote $O(\log N)$ children; do $O(\log N)$ merges to recover the heap

	popmin	push	decreasekey
binary heap	$O(\log N)$	$O(\log N)$	$O(\log N)$
binomial heap	$O(\log N)$	$O(\log N)$	$O(\log N)$

But what about aggregate costs?

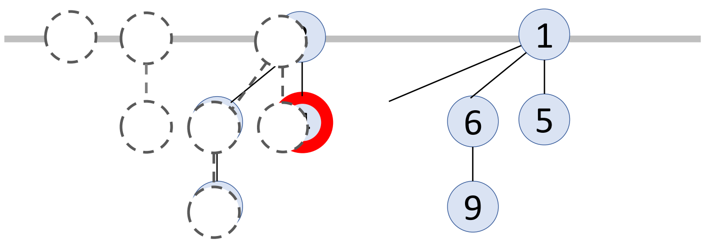
	popmin	push	decreasekey
binary heap	$O(\log N)$	$O(\log N)$	$O(\log N)$
binomial heap	$O(\log N)$	$O(\log N)$	$O(\log N)$

But what about aggregate costs?



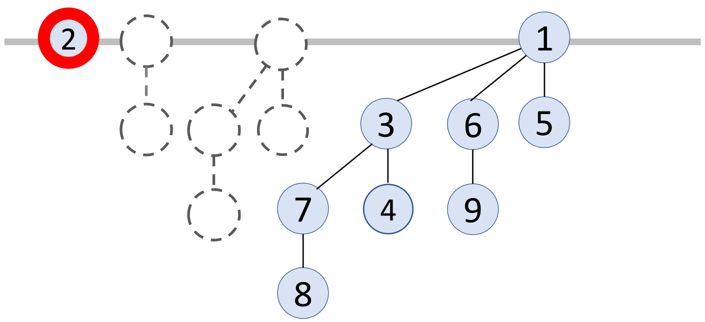
	popmin	push	decreasekey
binary heap	$O(\log N)$	$O(\log N)$	$O(\log N)$
binomial heap	$O(\log N)$	$O(\log N)$	$O(\log N)$

But what about aggregate costs?



	popmin	push	decreasekey
binary heap	$O(\log N)$	$O(\log N)$	$O(\log N)$
binomial heap	$O(\log N)$	$\cancel{O(\log N)}$ $O(1)$ amortized [Ex. sheet 6 q. 2, 4]	$O(\log N)$

But what about aggregate costs?



This subsequent push is $O(1)$, because the first push created space for it.

NEXT TIME. Dijsktra's algorithm makes $O(E)$ calls to push / decreasekey, and only $O(V)$ calls to `popmin`. We can live with $O(\log N)$ for `popmin`, but can we make both push and `decreasekey` be $O(1)$?