Priority Queues

Review

- **Work lists**: data structures that
  - store elements and
  - give them back one at a time – in some order

- ***Stacks***: retrieve the element inserted most recently

- ***Queues***: retrieve the element that has been there longest

- **Priority queues**: retrieve
  the most “interesting” element

Today

The Work List Interface

- Recall the work list interface template:

Now, fully generic

Work List Interface

```
// typedef void* elem;          // Decided by client

// typedef ______* wl_t;

bool wl_empty(wl_t W)
  /*@requires W != NULL;                                    @*/ ;

wl_t wl_new()
  /*@ensures \result != NULL &&  wl_empty(\result);  @*/ ;

void wl_add(wl_t W, elem e)
  /*@requires W != NULL && e != NULL;                       @*/
  /*@ensures !wl_empty(W);                                  @*/ ;

elem wl_retrieve(wl_t W)
  /*@requires W != NULL && !wl_empty(W);                    @*/
  /*@requires \result != NULL;                              @*/ ;
```

This is not the interface of an actual data structure but a general template for the work lists we are studying

Priority Queues

Priority Queues

… retrieve the *most "interesting"* element

- Elements are given a **priority**
  - retrieves the element with the **highest priority**
  - several elements may have the same priority
- Examples
  - emergency room
    - highest priority = most severe condition
  - processes in an OS
    - highest priority = *well, it's complicated*
  - homework due
    - Highest priority = …

Towards a Priority Queue Interface

- It will be convenient to have a peek function
  - it returns the highest priority element without removing it

Priority Queue Interface

```
// typedef void* elem;          // Decided by client

// typedef ______* pq_t;

bool pq_empty(pq_t Q)
  /*@requires Q != NULL;                               @*/ ;

pq_t pq_new()
  /*@ensures \result != NULL &&  pq_empty(\result); @*/ ;

void pq_add(pq_t Q, elem e)
  /*@requires Q != NULL && e != NULL;                  @*/
  /*@ensures !pq_empty(Q);                             @*/ ;

elem pq_rem (pq_t Q)
  /*@requires Q != NULL && !pq_empty(Q);               @*/
  /*@ensures \result != NULL;                          @*/ ;

elem pq_peek (pq_t Q)
  /*@requires Q != NULL && !pq_empty(Q);               @*/
  /*@ensures \result != NULL && !pq_empty(Q);          @*/ ;
```

This is the work list interface with names changed

Added

How to Specify Priorities?

1. Mention it as part of pq_add

```
void pq_add(pq_t Q, elem e, int priority)
```

- How do we assign a priority to an element?
  - the same element should always be given the same priority
  - priorities should form some kind of order
- Do bigger numbers represent higher or lower priorities?

People are bad at being consistent

Potential for lots of errors

✘

How to Specify Priorities?

2. Make the priority part of an elem
   - and provide a way to retrieve it

```
int get_priority(elem e)
```

   - How do we assign a priority to an element?
     - the same element should always be given the same priority
     - priorities should form some kind of order
   - Do bigger numbers represent higher or lower priorities?

Same issues as (1)

✘

The problem is that assigning a priority to an element is hard for people

but given two elements saying which one has higher priority is easier

How to Specify Priorities?

3. Have a way to tell which of two elements has higher priority

```
bool has_higher_priority(elem e1, elem e2)
```

Given two elements, saying which one has higher priority is easier

- it returns **true** if e1 has **strictly higher priority** than e2
- It is the client who should provide this function
  - only they know what elem is
- For the priority queue library to be **generic**, we turn it into a type definition

```
typedef bool has_higher_priority_fn(elem e1, elem e2);
```

and have pq_new take a priority function as input

✓

The Priority Queue Interface

Priority Queue Interface

```
// typedef void* elem;          // Decided by client

typedef bool has_higher_priority_fn(elem e1, elem e2);

// typedef ______* pq_t;

bool pq_empty(pq_t Q)
  /*@requires Q != NULL;                                   @*/ ;

pq_t pq_new(has_higher_priority_fn* prio)
  /*@requires prio != NULL; @*/
  /*@ensures \result != NULL &&  pq_empty(\result); @*/ ;

void pq_add(pq_t Q, elem e)
  /*@requires Q != NULL && e != NULL;                      @*/
  /*@ensures !pq_empty(Q);                                 @*/ ;

elem pq_rem (pq_t Q)
  /*@requires Q != NULL && !pq_empty(Q);                   @*/
  /*@ensures \result != NULL;                              @*/ ;

elem pq_peek (pq_t Q)
  /*@requires Q != NULL && !pq_empty(Q);                   @*/
  /*@ensures \result != NULL && !pq_empty(Q);              @*/ ;
```

f(e1, e2) returns true if e1 has **strictly higher priority** than e2

We commit to the priority function when creating the queue

Priority Queue Implementations

	Unsorted array/list	*Sorted array/list*	*AVL trees*	*Heaps*
add	*O(1)*	*O(n)*	*O(log n)*	*O(log n)*
rem	*O(n)*	*O(1)*	*O(log n)*	*O(log n)*
peek	*O(n)*	*O(1)*	*O(log n)*	*O(1)*

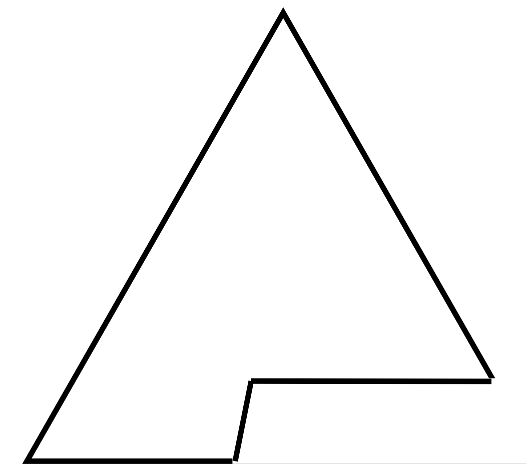
Cost of **add** using arrays are amortized

Heaps

Heaps

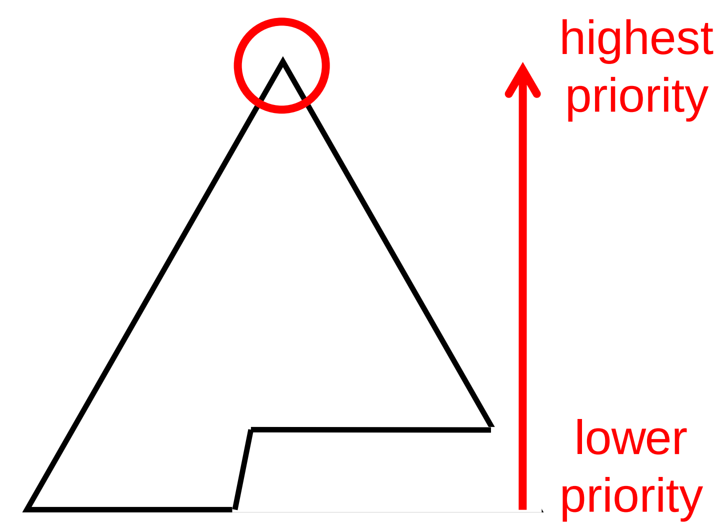
Nothing to do with the memory segment

- A **heap** is a type of binary tree used to implement priority queues



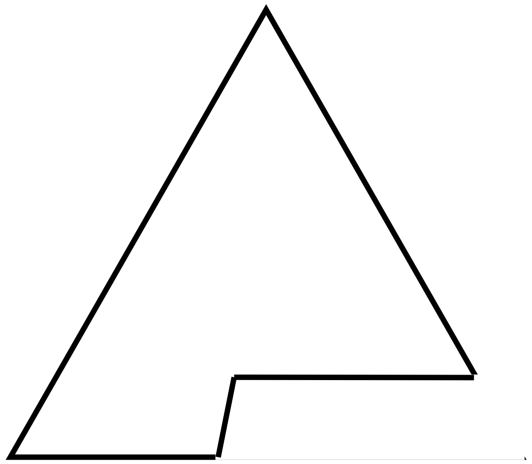
- Since add and rem have cost *O(log n)*, a heap is a **balanced** binary tree
  - in fact, they are as balanced a tree can be

- Since peek has cost O(1), the highest priority element must be at the root
  - in fact, the elements on any path from a leaf to the root are ordered in increasing priority order



Heaps Invariants

1. Shape invariant



2. Ordering invariant

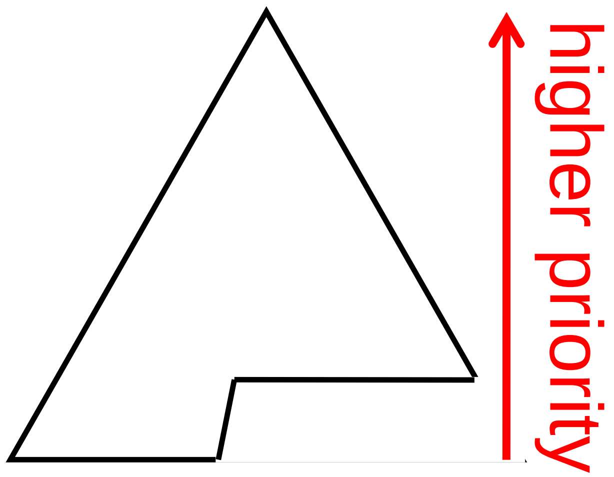
point of view of **child**

- The priority of a child is lower than or equal to the priority of its parent

or equivalently

point of view of **parent**

- The priority of a parent is higher than or equal to the priority of its children



Both points of view will come handy

The Many Things Called Heaps

- A **heap** is a type of binary tree used to implement priority queues
- A **heap** is also any priority queue where priorities are integers
  - it is a **min-heap** if smaller numbers represent higher priorities
  - it is a **max-heap** if bigger numbers represent higher priorities
- A **heap** is the segment of memory we called allocated memory

This is a significant source of confusion

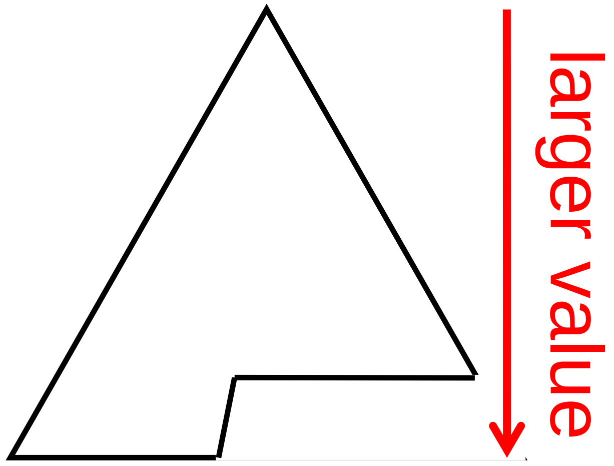
Min-heaps

- Any priority queue where priorities are integers and **smaller numbers** represent **higher priorities**

- In practice, most priority queues are implemented as min-heaps
  - and **heap** is also shorthand for min-heap — more confusion!

- Most of our examples will be min-heaps
  1. Shape invariant
  2. Ordering invariant
     - The **value** of a child is ≥ the **value** of its parent

       or equivalently
     - The **value** of a parent is ≤ the **value** of its children

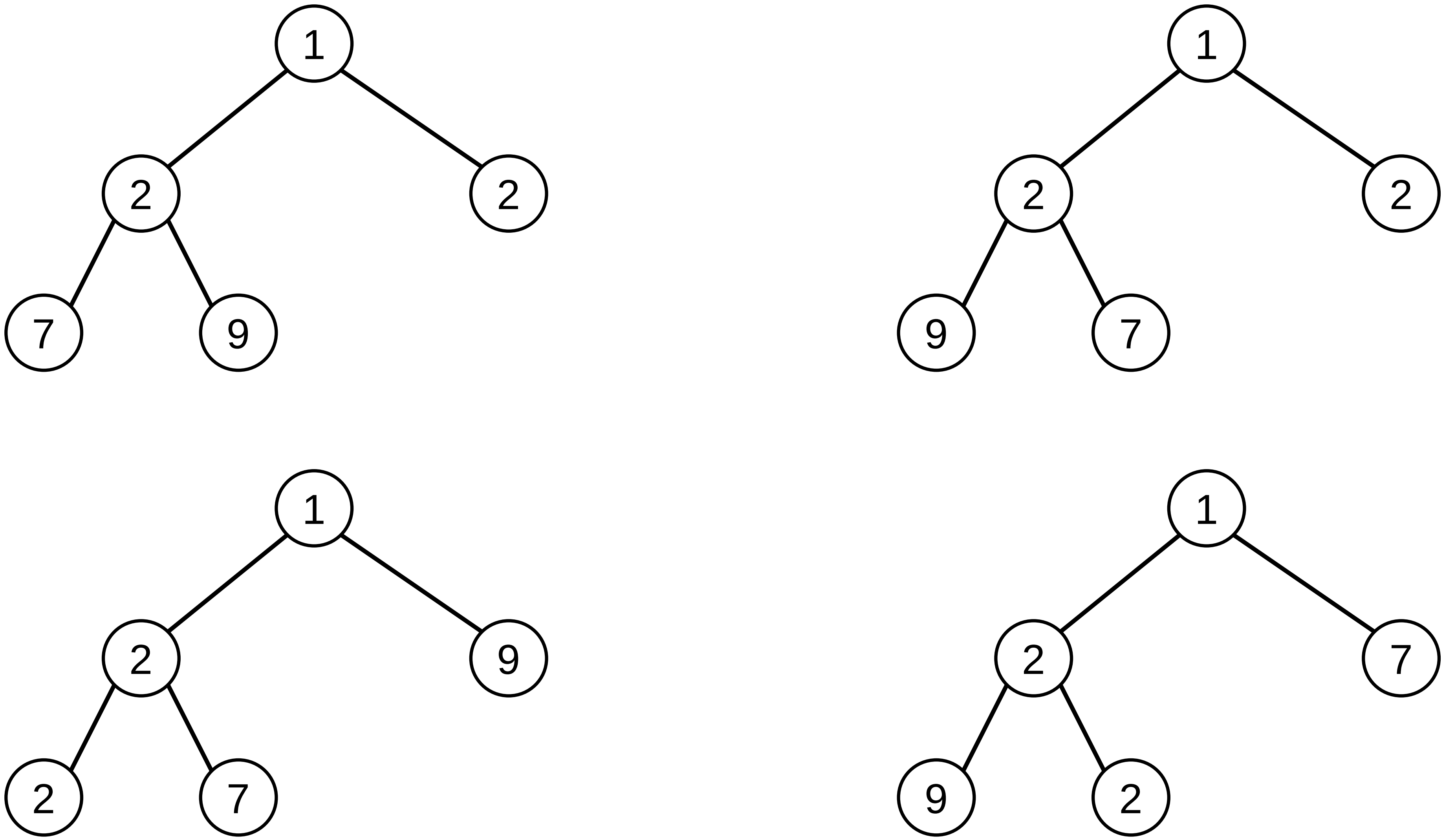


Activity

- Draw a min-heap with values 1, 2, 2, 9, 7

Activity

- Draw a min-heap with values 1, 2, 2, 9, 7

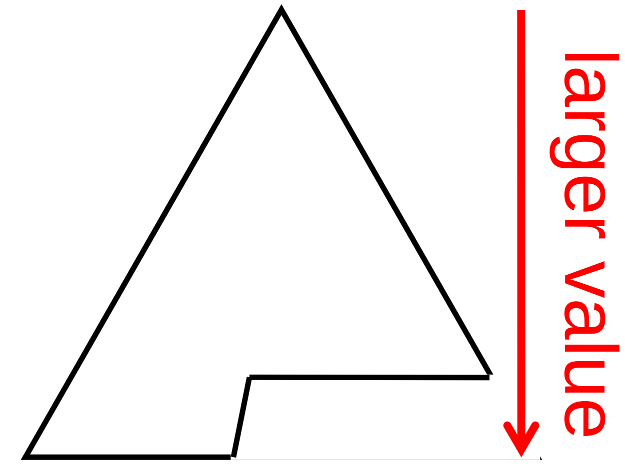


… and several more

Insertion into a Heap

Strategy

- Maintain the shape invariant
- Temporary break and then restore the ordering invariant

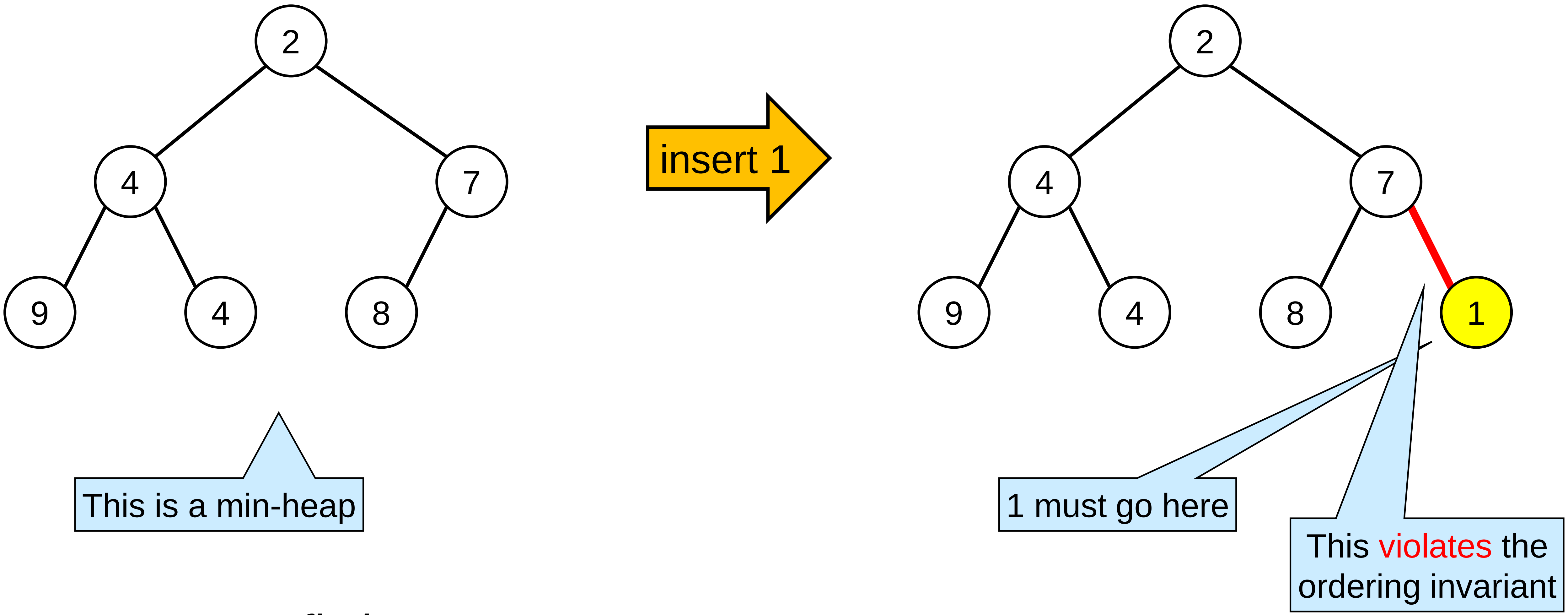


Min-heap version

This is similar to what we did for AVL trees
- maintain the ordering invariant
- temporary break and then restore the height invariant

Example

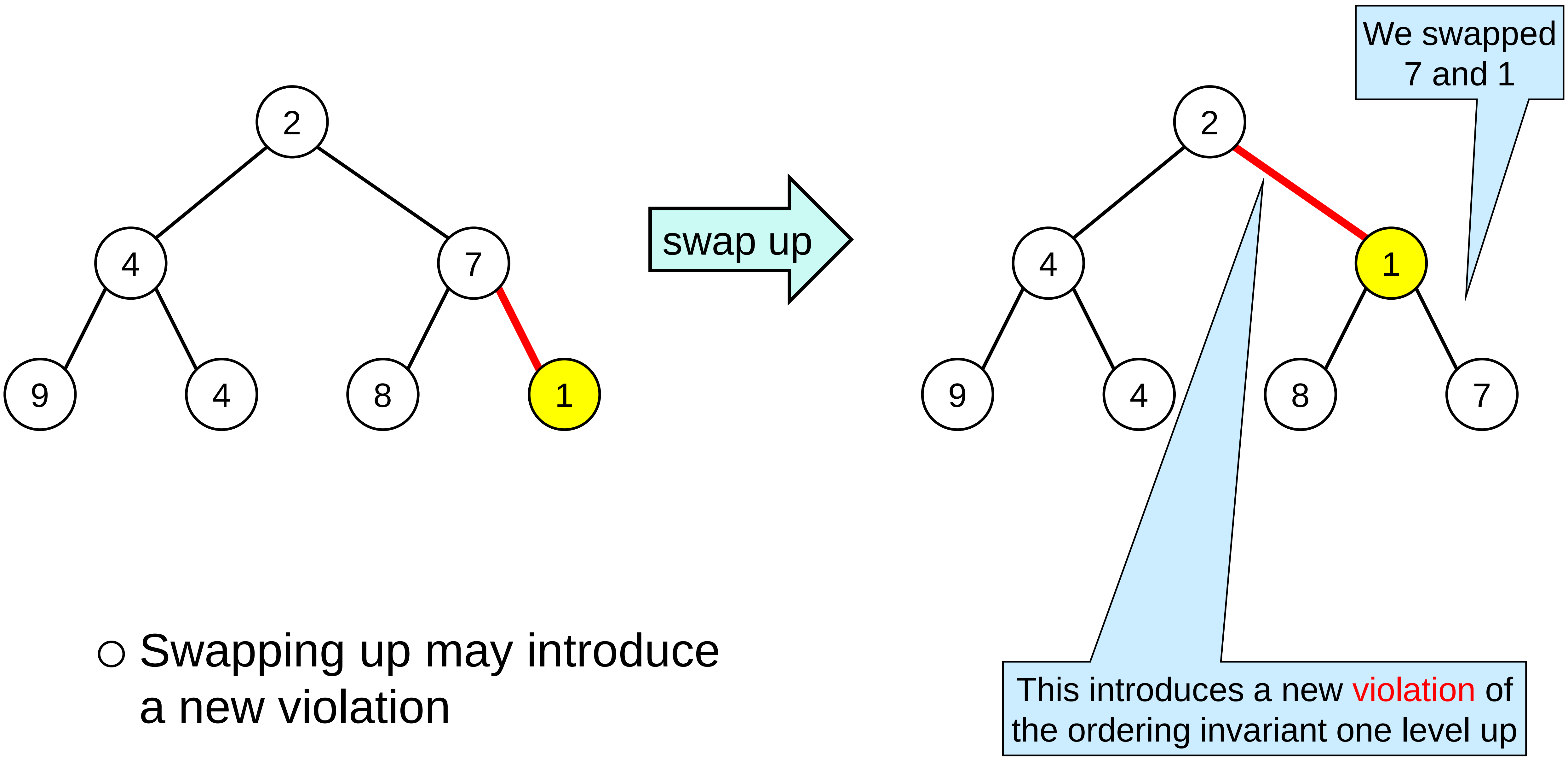
- We start by putting the new element in the only place that maintains the shape invariant
  - but doing so may break the ordering invariant



  - *How to fix it?*

Swapping up

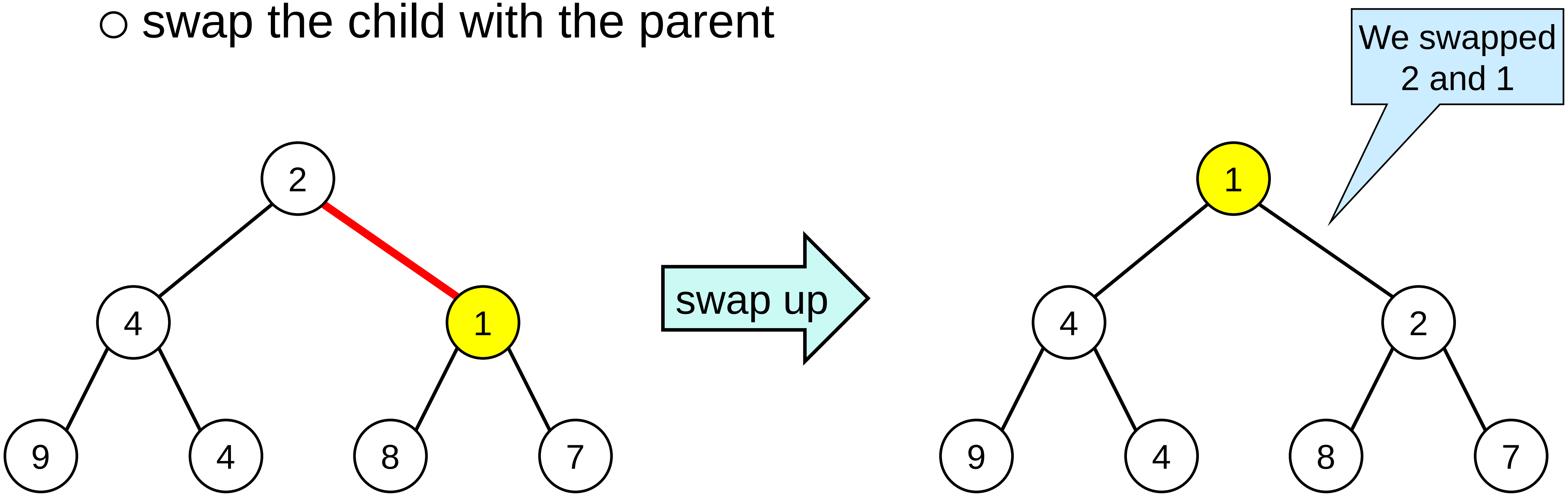
- How to fix the violation?
  - swap the child with the parent



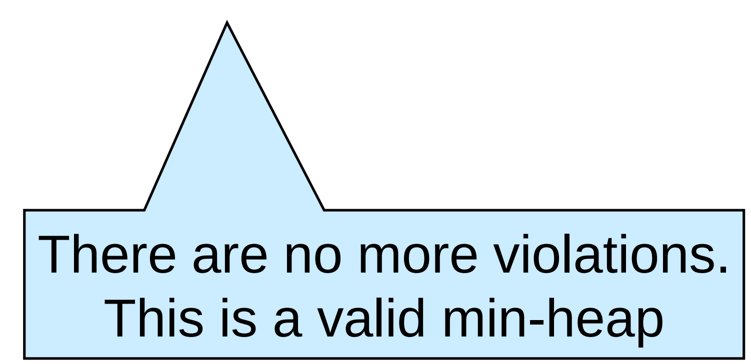
  - Swapping up may introduce a new violation

Swapping up

- How to fix the violation?
  - swap the child with the parent

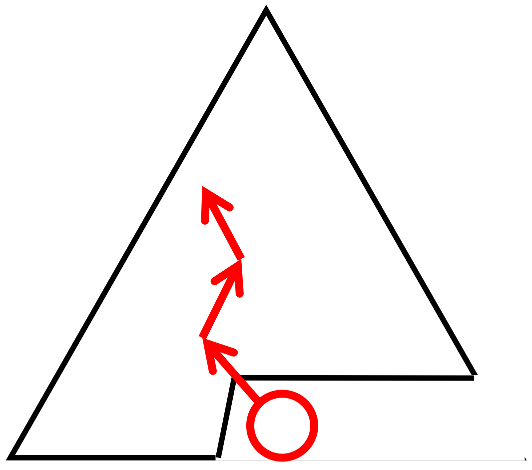


- We stop when no new violation is introduced
  - or we reach the root



Adding an Element

- General procedure
  1. Put the added element in the one place that maintains the shape invariant
     - the leftmost open slot on the last level
       - or, if the last level is full, the leftmost slot on the next level
  2. Repeatedly swap it up with its parent
     - until the violation is fixed
     - or we reach the root
  - There is always **at most one violation**



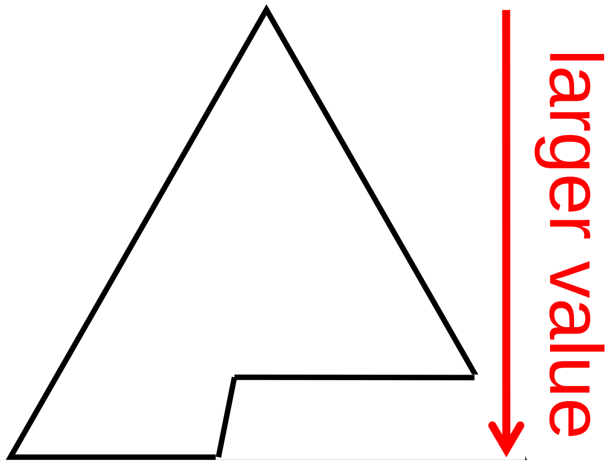
- The overall process is called **sifting up**
- This costs *O(log n)*
  - because we make at most *O(log n)* swaps

For a heap with *n* elements

Removing the Minimal Element of a Heap

Strategy

- Maintain the shape invariant
- Temporary break and then restore the ordering invariant

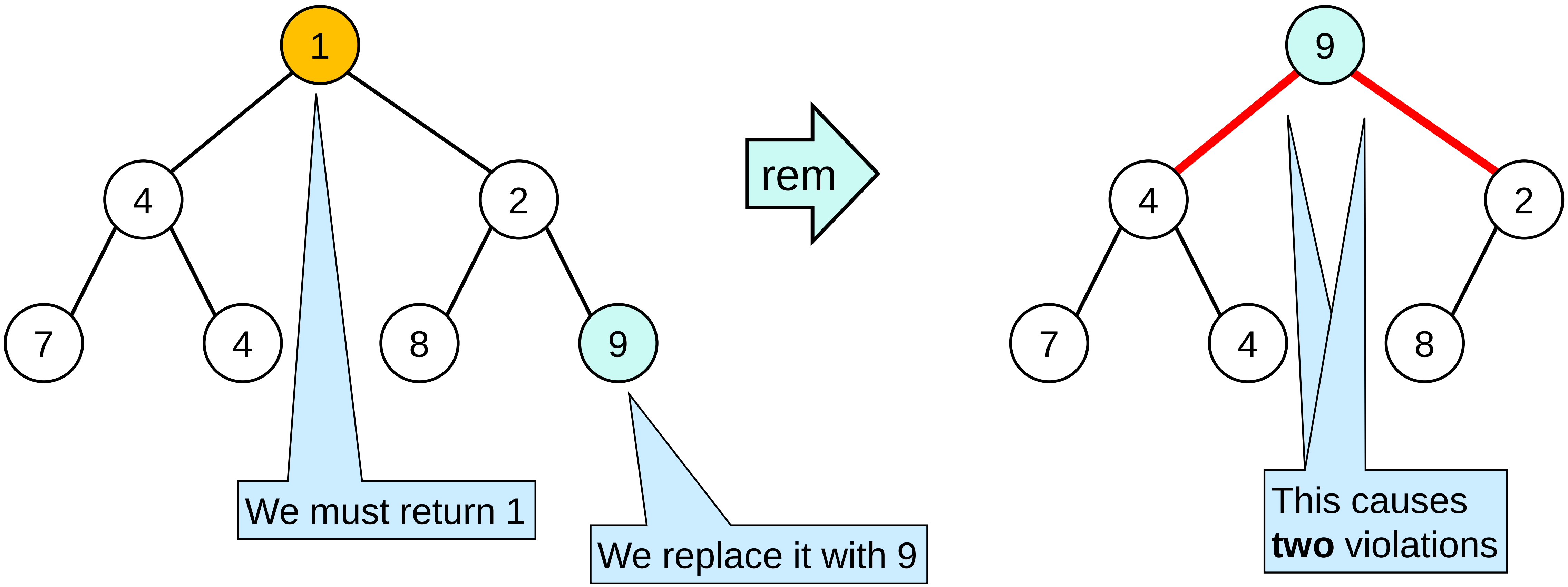


Same as insertion

Min-heap version

Example

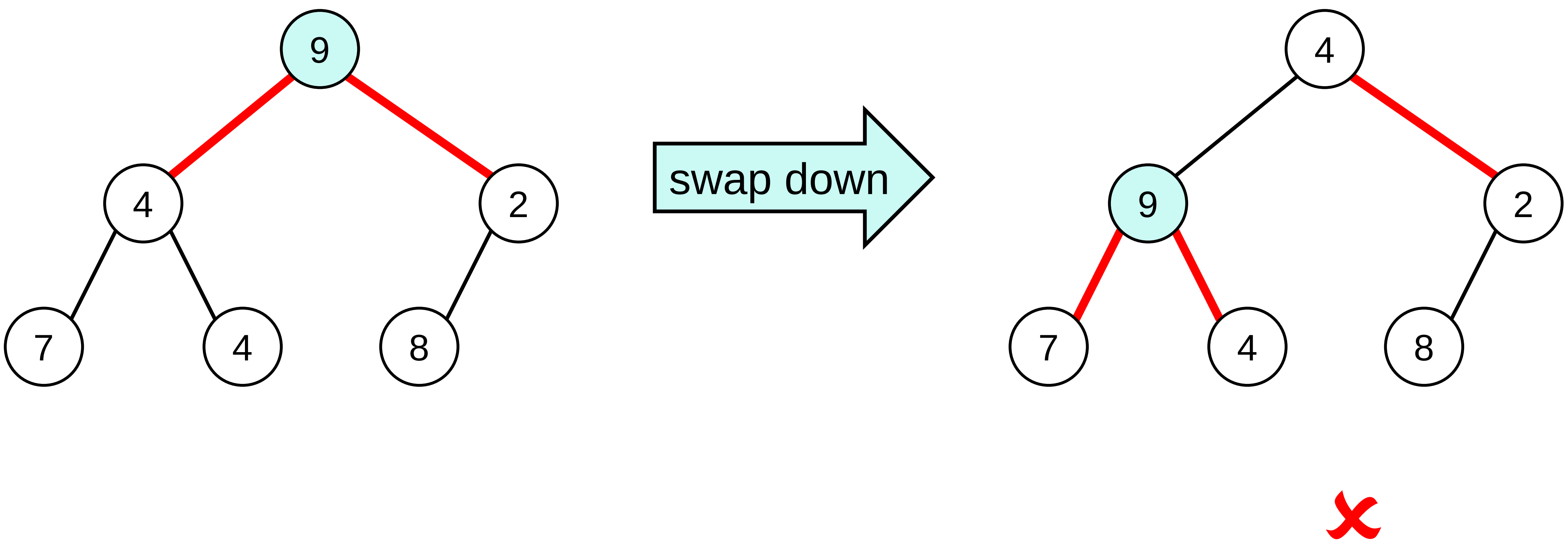
- We must return the root
- We replace it with the only element that maintains the shape invariant



- *Which violation to fix first?*

Swapping down

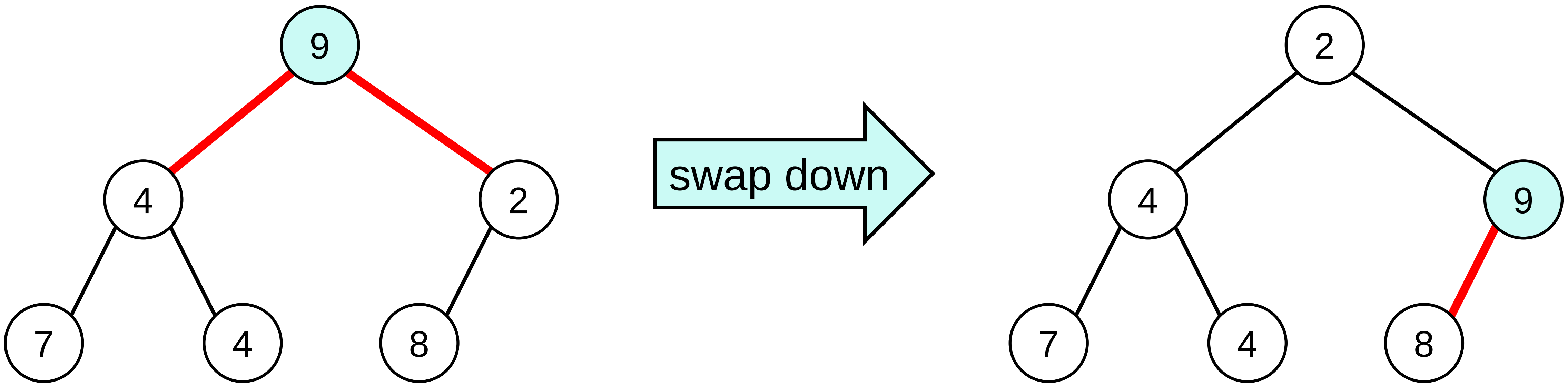
- Which violation to fix first?
  - If we swap 4 and 9, we end up with **three** violations



- *Can we do better?*

Swapping down

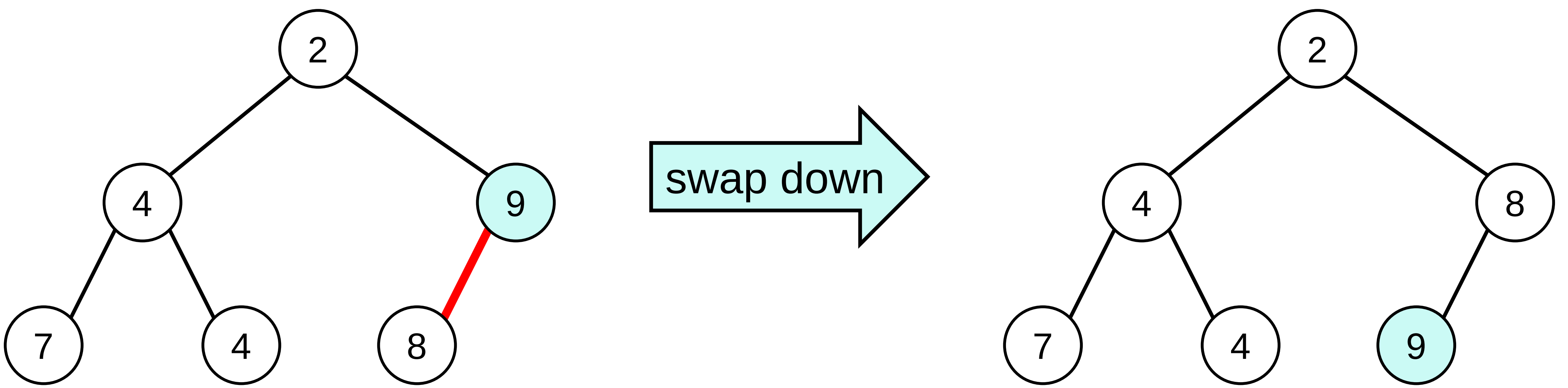
- If we swap 9 and 2, we end up with **one** violation
  - at most two in general



- When swapping down, always **swap with the child with the highest priority**
  - smallest value in a min-heap

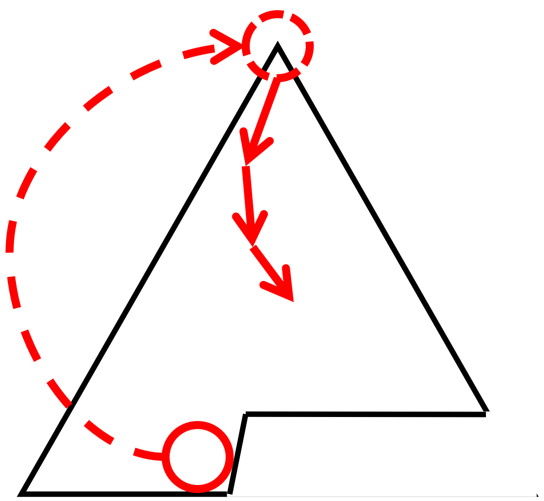
Swapping down

- Always swap the child with the highest priority



- We stop when no new violations are introduced
  - or we reach a leaf

Removing an Element



- General procedure
  1. Return the root
  2. Replace it with the element in the one place that maintains the shape invariant
     - ➢ the rightmost element on the last level
  3. Repeatedly swap it down with its child that has highest priority
     - ➢ until all violations are fixed
     - ➢ or we reach a leaf
  - ○ This guarantees there are always **at most two violations**
- The overall process is called **sifting down**
- This costs *O(log n)*
  - ○ because we make at most *O(log n)* swaps

For a heap with *n* elements

Priority Queue Implementations

	Unsorted array/list	***Sorted array/list***	***AVL trees***	***Heaps***
add	*O(1)*	*O(n)*	*O(log n)*	*O(log n)*
rem	*O(n)*	*O(1)*	*O(log n)*	*O(log n)*
peek	*O(n)*	*O(1)*	*O(log n)*	*O(1)*

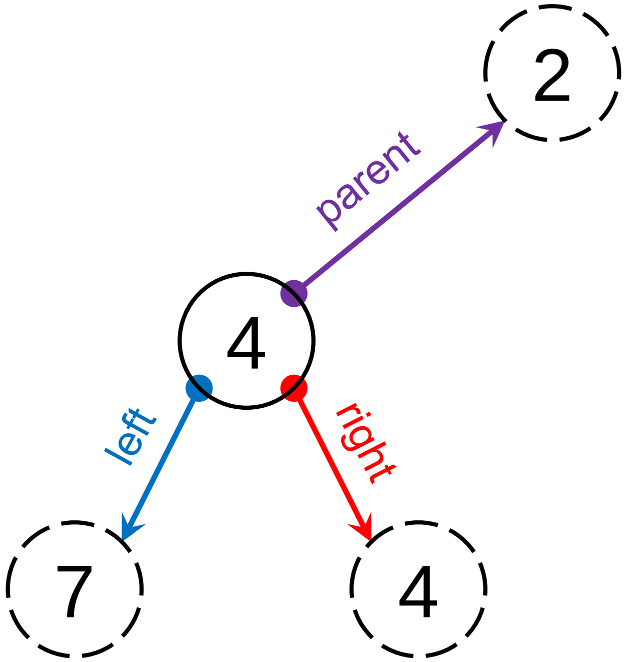
✓

Cost of **add** using arrays are amortized

Representing Heaps

How to Represent a Heap?

- Borrowing from BSTs, we could use pointers
  - left and right child
    - needed when sifting down
  - parent node
    - needed when sifting up




```
typedef struct heap_node heap;
struct heap_node {
  elem data;
  heap* parent;
  heap* left;
  heap* right;
};
```

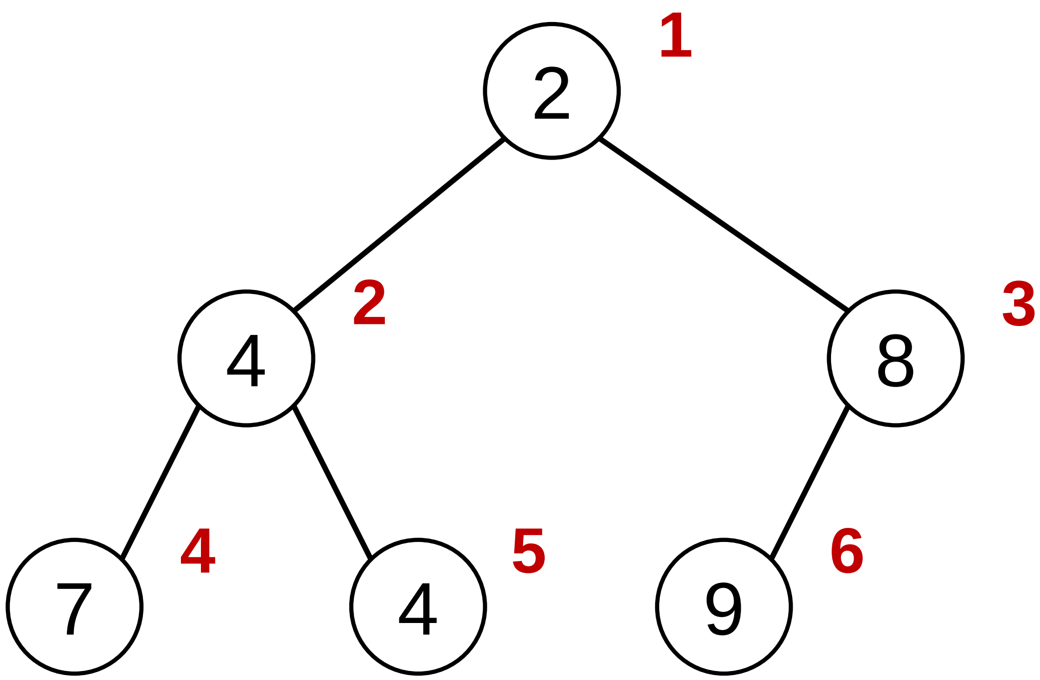
- That's a lot of pointers to keep track of!
  - It also takes up a lot of space

Try writing the swap function!

- *Can we do better?*

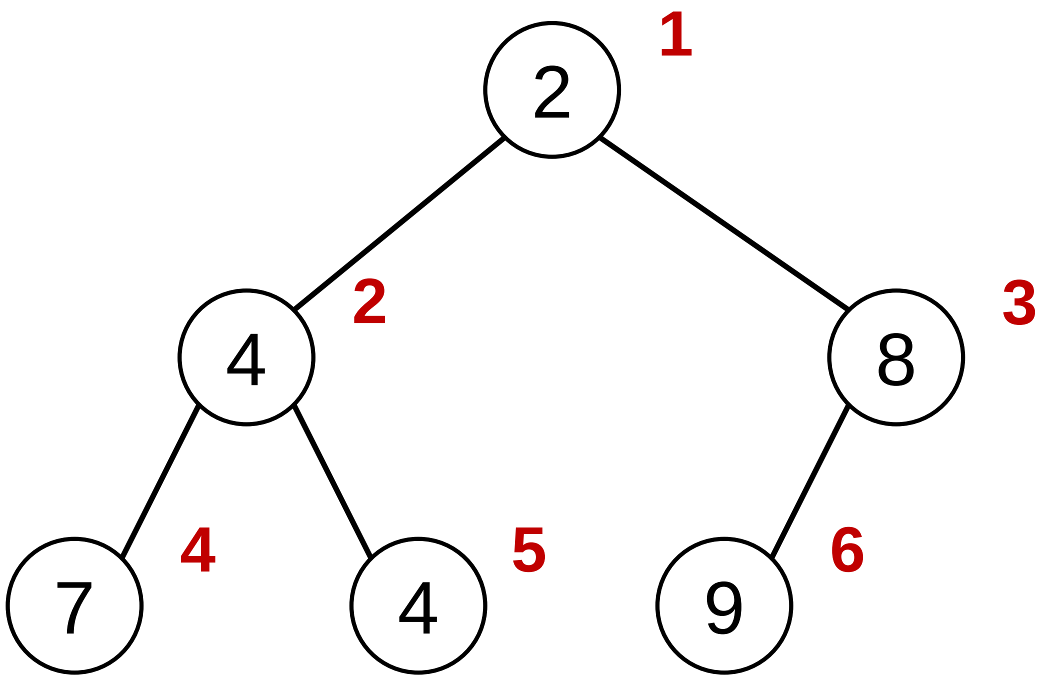
Understanding Heaps

- Let's number the nodes level by level starting at 1



- Observations:
  - If a node has number i, its **left child** has number **2i**
  - If a node has number i, its **right child** has number **2i + 1**
  - If a node has number i, its **parent** has number **i/2**

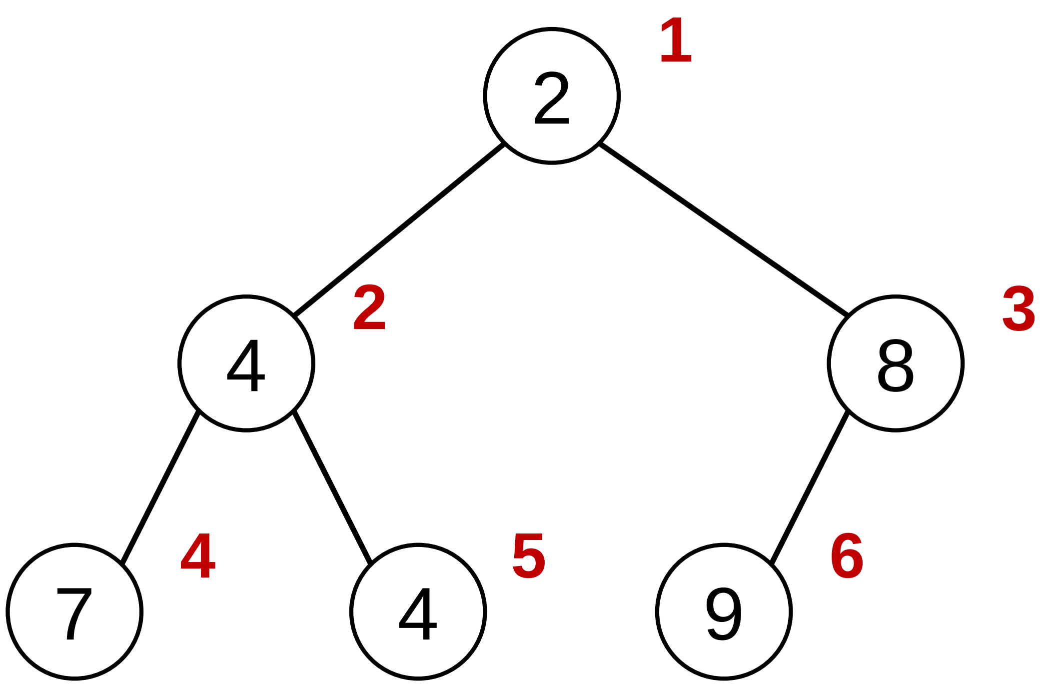
Understanding Heaps



- If a node has number i, its **left child** has number 2i
- If a node has number i, its **right child** has number 2i + 1
- If a node has number i, its **parent** has number i/2

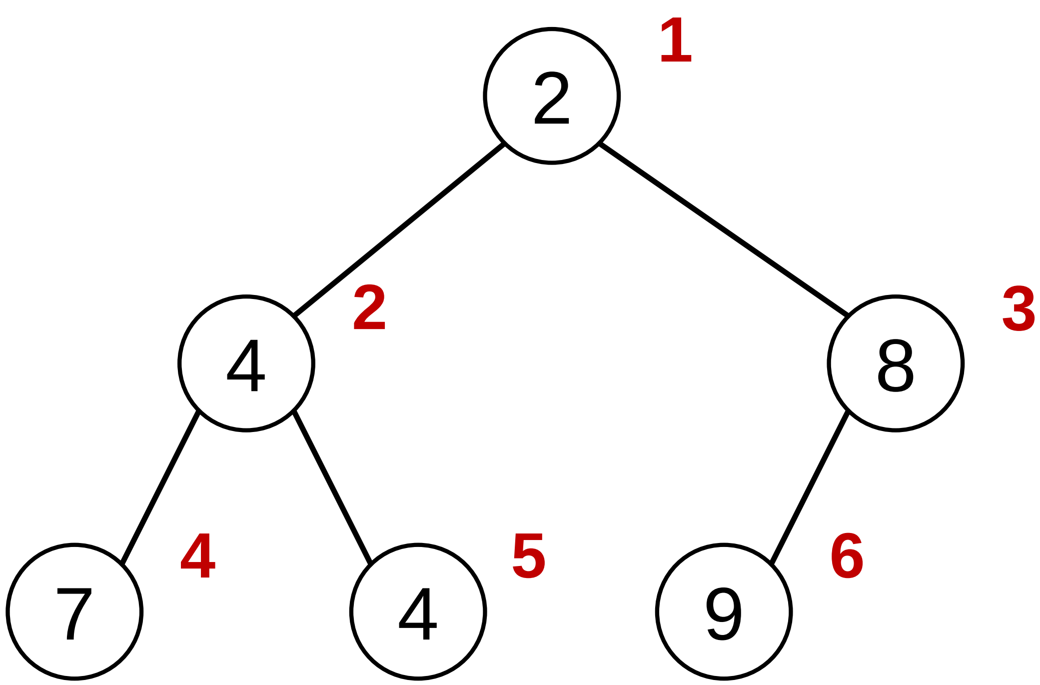
- By numbering nodes this way, we can navigate the tree up and down using **arithmetic**

Understanding Heaps



- By numbering nodes this way, we can navigate the tree up and down using arithmetic
- These numbers are **contiguous** and **start at 1**

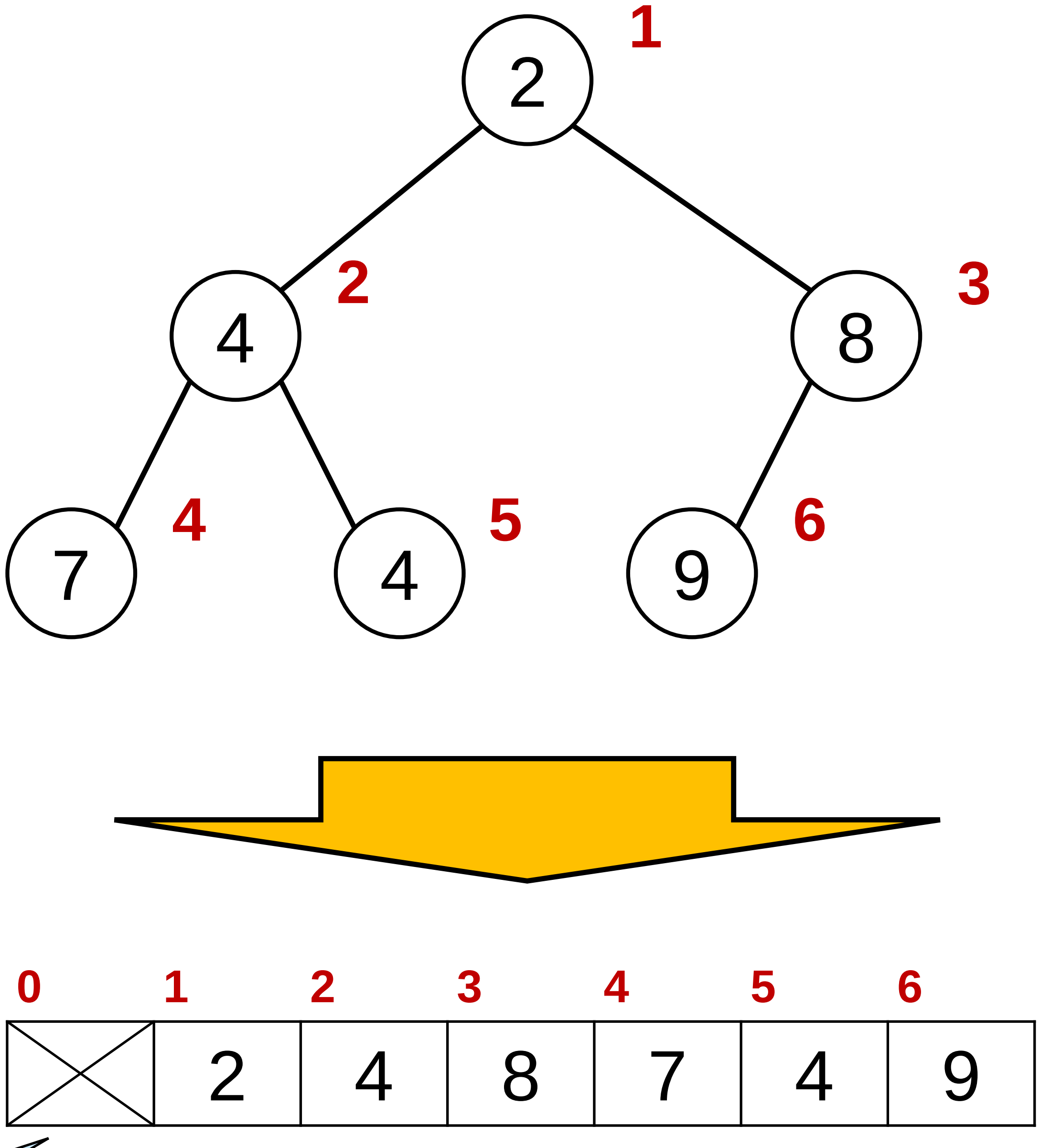
Understanding Heaps



- These numbers are contiguous and start at 1
- Do we know of any data structures that allows accessing data based on consecutive integers?

Arrays!

Representing Heaps using Arrays

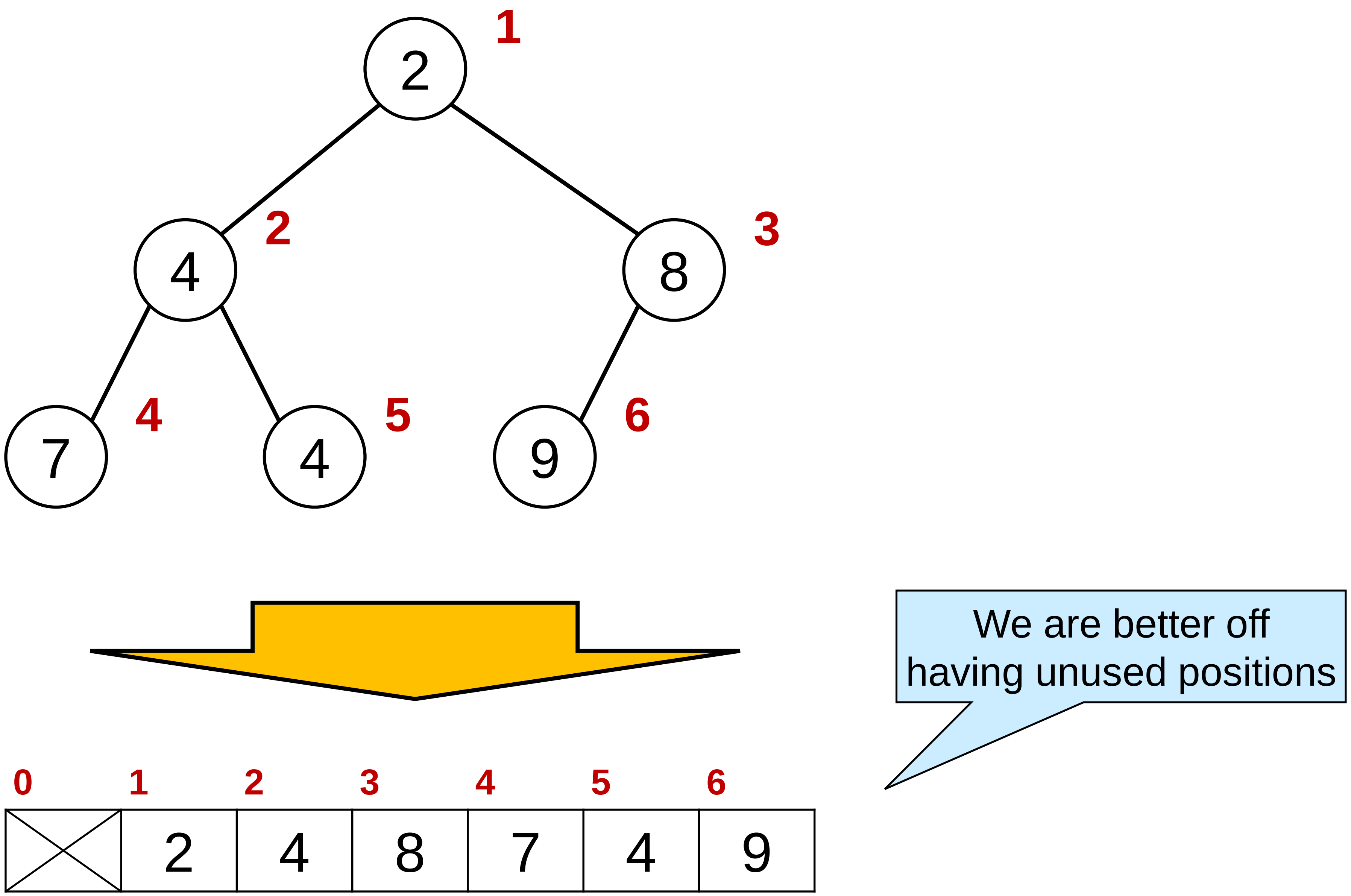


For simplicity,
we do not use index 0

- If a node has number i, its left child has number 2i
- If a node has number i, its right child has number 2i + 1
- if a node has number i, its parent has number i/2

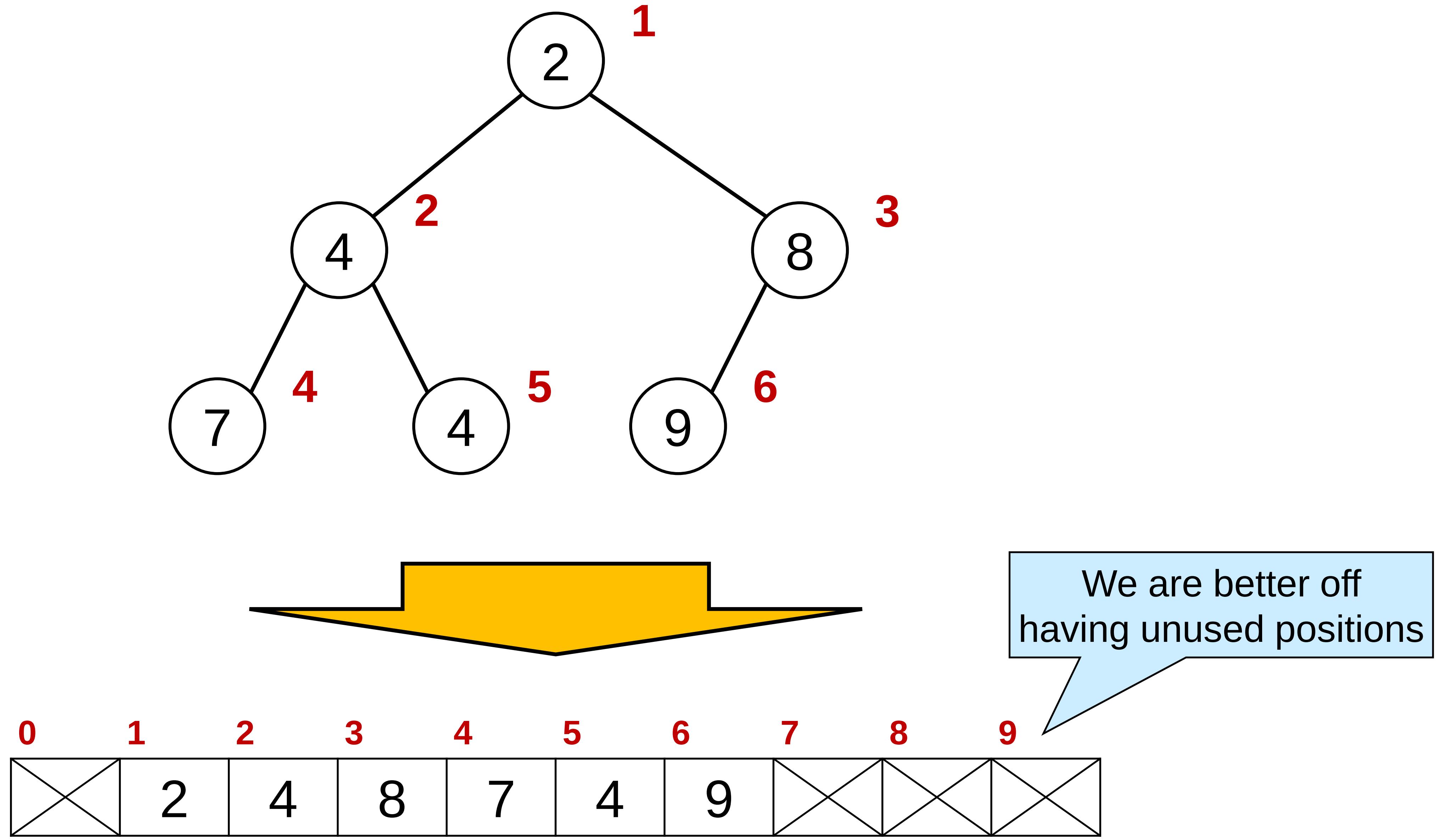
Representing Heaps using Arrays

- add will initially put a new element at index 7
- remove will yank the element at index 6



Representing Heaps using Arrays

- add will initially put a new element at index 7
- remove will yank the element at index 6



Bounded Priority Queues

Types of Work Lists

- The work lists we considered so far were **unbounded**
  - there was no maximum to the number of elements they could hold
- A **bounded work list** has a capacity fixed at creation time
  - we can't add elements once full
- In practice
  - stacks are typically unbounded
  - queues can be either
  - priority queues are often bounded

Priority Queue Interface

```
// typedef void* elem;          // Decided by client

typedef bool has_higher_priority_fn(elem e1, elem e2);

// typedef ______* pq_t;

bool pq_empty(pq_t Q)
  /*@requires Q != NULL;                                        @*/ ;

pq_t pq_new(has_higher_priority_fn* prio)
  /*@requires prio != NULL; @*/
  /*@ensures \result != NULL &&  pq_empty(\result);  @*/ ;

void pq_add(pq_t Q, elem e)
  /*@requires Q != NULL && e != NULL;                           @*/
  /*@ensures !pq_empty(Q);                                      @*/ ;

elem pq_rem (pq_t Q)
  /*@requires Q != NULL && !pq_empty(Q);                        @*/
  /*@ensures \result != NULL;                                   @*/ ;

elem pq_peek (pq_t Q)
  /*@requires Q != NULL && !pq_empty(Q);                        @*/
  /*@ensures \result != NULL && !pq_empty(Q);          @*/ ;
```

The Bounded Priority Queue Interface

- pq_new now takes the capacity of the priority queue
- We need a new function to check if it is full
  - pq_full
- We cannot insert an element into a full priority queue
- A priority queue is not full after removing an element

Bounded Priority Queue Interface

```
// typedef void* elem;          // Decided by client
typedef bool has_higher_priority_fn(elem e1, elem e2);

// typedef ______* pq_t;

bool pq_empty(pq_t Q)
  /*@requires Q != NULL;                                    @*/ ;

bool pq_full(pq_t Q)
  /*@requires Q != NULL;                                    @*/ ;

pq_t pq_new(int capacity, has_higher_priority_fn* prio)
  /*@requires capacity > 0 && prio != NULL; @*/
  /*@ensures \result != NULL &&  pq_empty(\result);      @*/ ;

void pq_add(pq_t Q, elem e)
  /*@requires Q != NULL && !pq_full(Q) && e != NULL;  @*/
  /*@ensures !pq_empty(Q);                                  @*/ ;

elem pq_rem (pq_t Q)
  /*@requires Q != NULL && !pq_empty(Q);                    @*/
  /*@ensures \result != NULL && !pq_full(Q);                @*/ ;

elem pq_peek (pq_t Q)
  /*@requires Q != NULL && !pq_empty(Q);                    @*/
  /*@ensures \result != NULL && !pq_empty(Q);               @*/ ;
```