Implementing Heaps

Bounded Priority Queues

- **Priority queues**:
  a type of work list that
  - stores elements
  - gives back the one with the highest priority

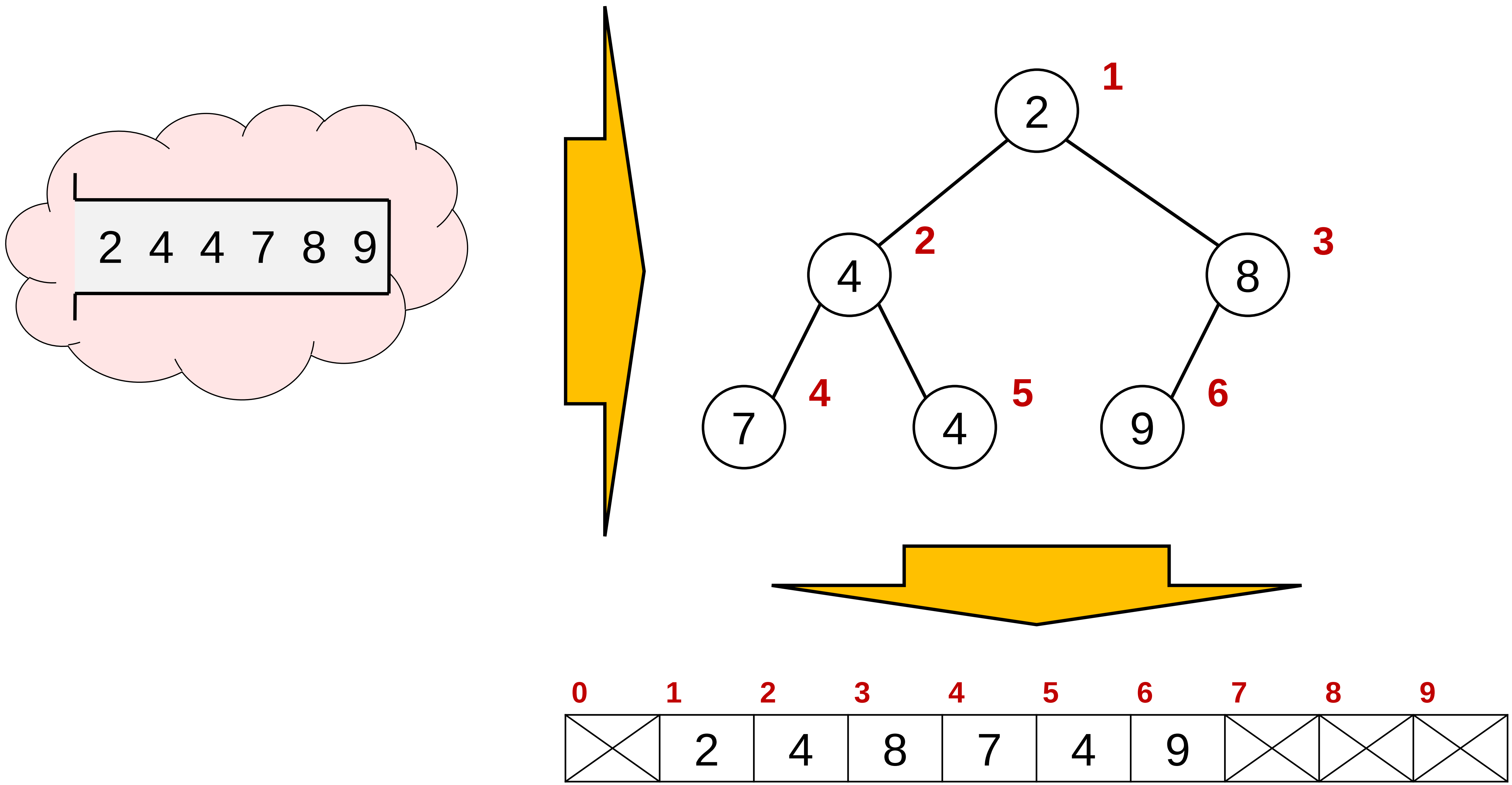
- How big?
  - unbounded
  - **bounded**

Bounded Priority Queue Interface

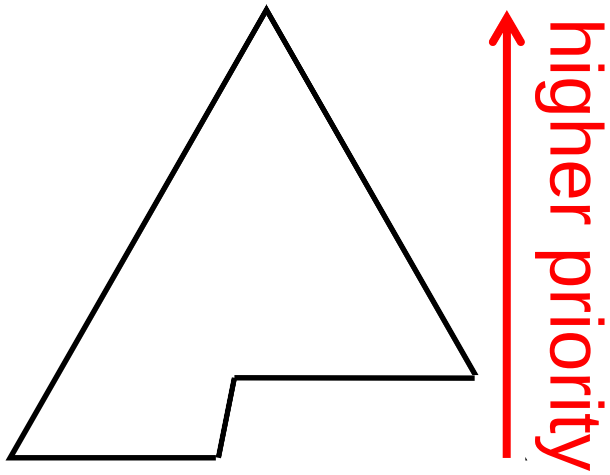
```
// typedef void* elem;          // Decided by client
typedef bool has_higher_priority_fn(elem e1, elem e2);

// 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);              @*/ ;
```

Priority Queues

A **priority queue** viewed as a **heap** implemented as an **array**



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

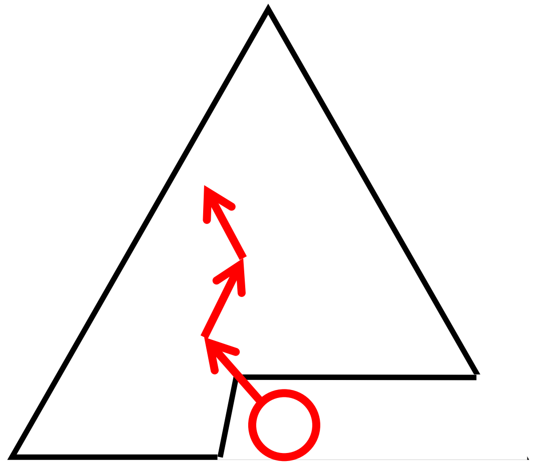
point of view of **parent**

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

Heap Operations

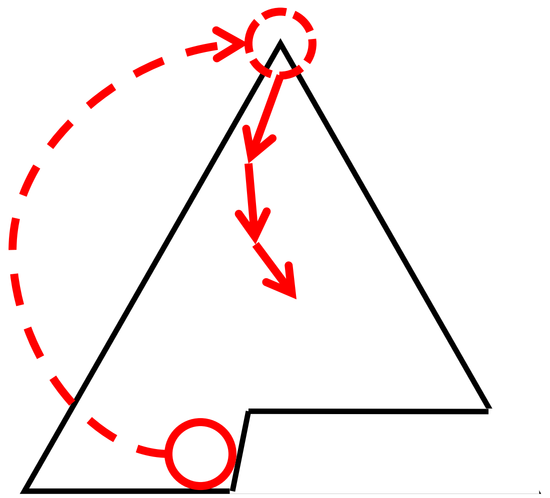
- Insertion
  - place the new element in the leftmost open position in the last level to satisfy the shape invariant
  - sift up to restore the ordering invariant

O(log n)



- Removal
  - replace the root with the element in the rightmost filled position on the last level to satisfy the shape invariant
  - sift down to restore the ordering invariant

O(log n)



Strategy:
- maintain the shape invariant
- temporarily break and then restore the ordering invariant

Priority Queue Implementations

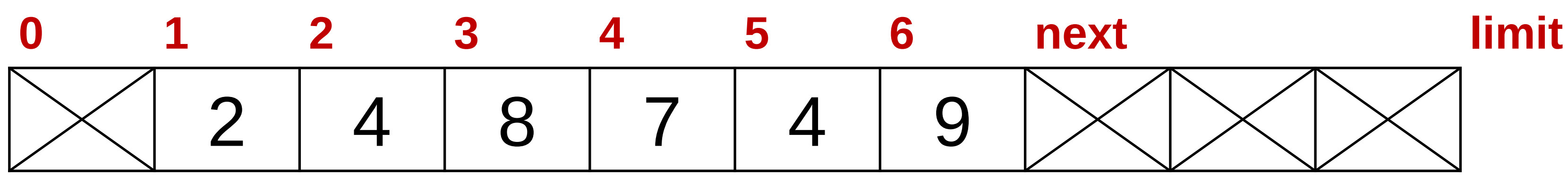
	Unsorted array/list	***Sorted array/list***	***AVL trees***	***Heaps***
add	*O(1)*	*O(n)*	*O(log n)*	*O(log n)*
rem	*O(n)*	*O(1)*	*O(log n)*	*O(log n)*
peek	*O(n)*	*O(1)*	*O(log n)*	*O(1)*

Cost of **add** using arrays are amortized

Only if we can access the bottom-most right-most node in O(1)

Implementing Bounded Heaps

Concrete Type



- The heap data structure needs to store
  - the array that contains the heap elements
  - its true size
    - that's capacity + 1
  - the position where to add the next element
  - the priority function

because we sacrifice index 0

```
typedef struct heap_header heap;
struct heap_header {
  int limit;                             // == capacity + 1
  elem[] data;                           // \length(data) == limit
  int next;                              // 1 <= next  && next <= limit
  has_higher_priority_fn* prior;  // != NULL
};
```

Basic Representation Invariants

```
typedef struct heap_header heap;
struct heap_header {
  int limit;                          // == capacity + 1
  elem[] data;                        // \length(data) == limit
  int next;                           // 1 <= next  && next <= limit
  has_higher_priority_fn* prior;  // != NULL
};
```

- We simply translate the field constraints
  - *and preempt overflow*

```
bool is_heap_safe(heap* H) {
  return H != NULL
      && 1 < H->limit && H->limit <= int_max()/2
      && is_array_expected_length(H->data, H->limit)
      && 1 <= H->next && H->next <= H->limit
      && H->prior != NULL;
}
```

because
right child of i is 2i+1

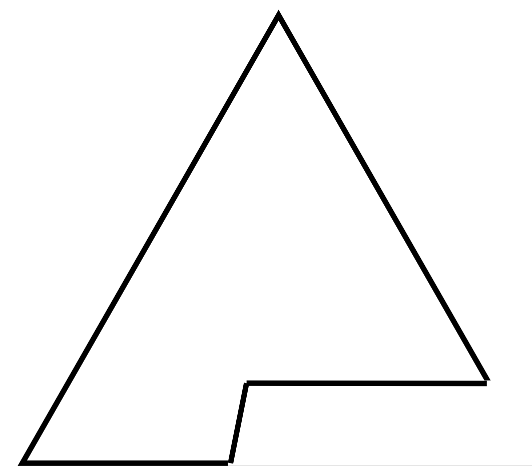
and
2*(int_max()/2) + 1 == int_max()

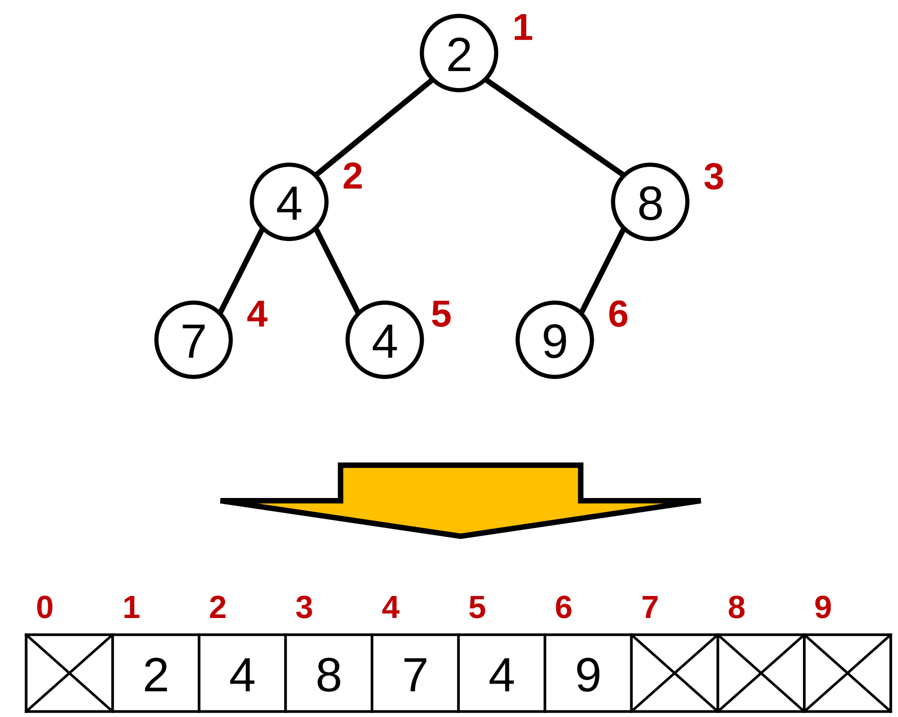
- This checks that basic heap manipulations are **safe**

Heap Invariants

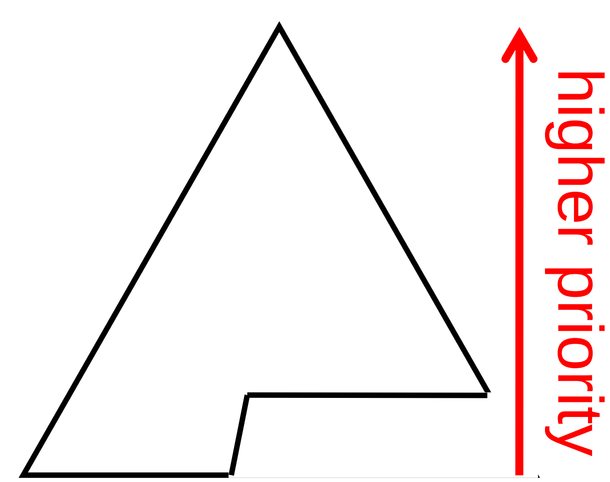
Beyond basic safety, we need to check:

- the shape invariant
  - this is automatic
    - elements are stored
      - level by level
      - from left to right





- the ordering invariant



The Ordering Invariant

```
typedef struct heap_header heap;
struct heap_header {
  int limit;        // == capacity + 1
  elem[] data;  // \length(data) == limit
  int next;        // 1 <= next  && next <= limit
  has_higher_priority_fn* prior; // != NULL
};
```

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

- Let's introduce an abstraction
  - Reason about where a node belongs in the tree
    - ➢ not priorities
    - ➢ not arrays

This will also help with the confusion about min-heaps

- It's Ok for node e1 to be the parent of e2 if
  - e1 has priority higher than or equal to e2
    - ➢ but prior tests if a node has ***strictly*** *higher* priority than another
  - it is **not the case** that
    e2 has strictly higher priority than e1

Min-heap version:
value of e1 ≤ value of e2

Min-heap version:
value of e2 ≮ value of e1

The Ordering Invariant

```
typedef struct heap_header heap;
struct heap_header {
  int limit;       // == capacity + 1
  elem[] data;  // \length(data) == limit
  int next;       // 1 <= next  && next <= limit
  has_higher_priority_fn* prior; // != NULL
};
```

- It's Ok for node e1 to be the parent of e2 if
  - it is **not the case** that
    e2 has strictly higher priority than e1

```
bool ok_above(heap* H, int i1, int i2)
//@requires is_heap_safe(H);
//@requires 1 <= i1 && i1 < H->next;
//@requires 1 <= i2 && i2 < H->next;
{
  elem e1 = H->data[i1];
  elem e2 = H->data[i2];
  return !(*H->prior)(e2, e1);
}
```

H is safe

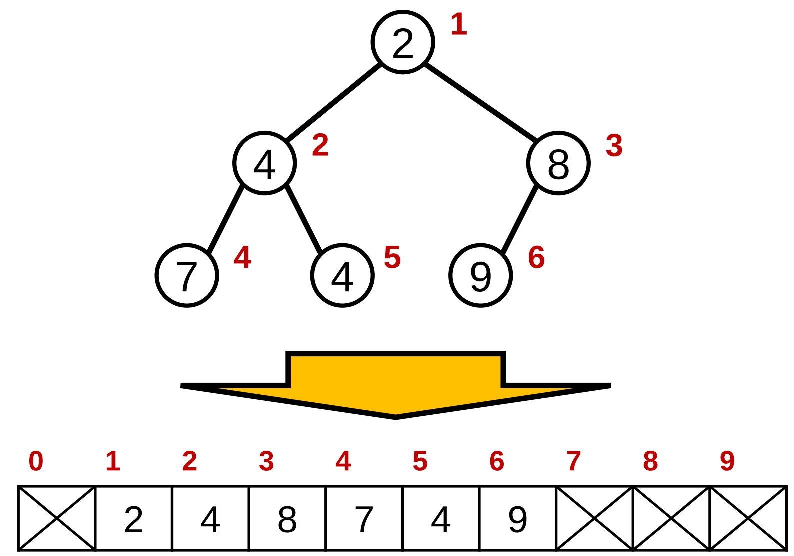
i1 and i2 are in bounds

It is **not the case** that
e2 has strictly higher priority than e1

```
typedef struct heap_header heap;
struct heap_header {
  int limit;        // == capacity + 1
  elem[] data;  // \length(data) == limit
  int next;        // 1 <= next  && next <= limit
  has_higher_priority_fn* prior; // != NULL
};
```

The Ordering Invariant

- The priority of **every** child is lower than or equal to the priority of its parent
  - Every parent is Ok above its children




```
bool is_heap_ordered(heap* H)
//@requires is_heap_safe(H);
{
  for (int child = 2; child < H->next; child++)
  //@loop_invariant 2 <= child && child <= H->next;
  {
    int parent = child/2;
    if (!ok_above(H, parent, child))
      return false;
  }
  return true;
}
```

H is safe

- The root of the tree is at index 1
  - ➢ the first child is at index 2
- *Is this code safe?*

```
typedef struct heap_header heap;
struct heap_header {
  int limit;        // == capacity + 1
  elem[] data;  // \length(data) == limit
  int next;        // 1 <= next  && next <= limit
  has_higher_priority_fn* prior; // != NULL
};
```

The Ordering Invariant

- Is this code safe?

```
1.  bool is_heap_ordered(heap* H)
2.  //@requires is_heap_safe(H);
3.  {
4.    for (int child = 2; child < H->next; child++)
5.    //@loop_invariant 2 <= child && child <= H->next;
6.    {
7.      int parent = child/2;
8.      if (!ok_above(H, parent, child))
9.        return false;
10.   }
11.   return true;
12. }
```

- H->next
  - because H != NULL
    - since is_heap_safe(H)
- ok_above(H, parent, child)

```
bool ok_above(heap* H, int i1, int i2)
//@requires is_heap_safe(H);
//@requires 1 <= i1 && i1 < H->next;
//@requires 1 <= i2 && i2 < H->next;
```

  - 1 <= child && child < H->next
    - because 2 <= child by line 5
    - and child < H->next by line 4
  - 1 <= parent && parent < H->next
    - because parent = child/2 by line 7
    - and 2 <= child && child < H->next
      - by lines 4–5 and math

The Representation Invariant

- A value of type heap must satisfy
  - the basic safety invariants
  - the shape invariant
    - automatic
  - the ordering invariant

```
bool is_heap(heap* H) {
  return is_heap_safe(H)
       && is_heap_ordered(H);
}
```

Constant-time Operations

pq_full, pq_empty, pq_peek

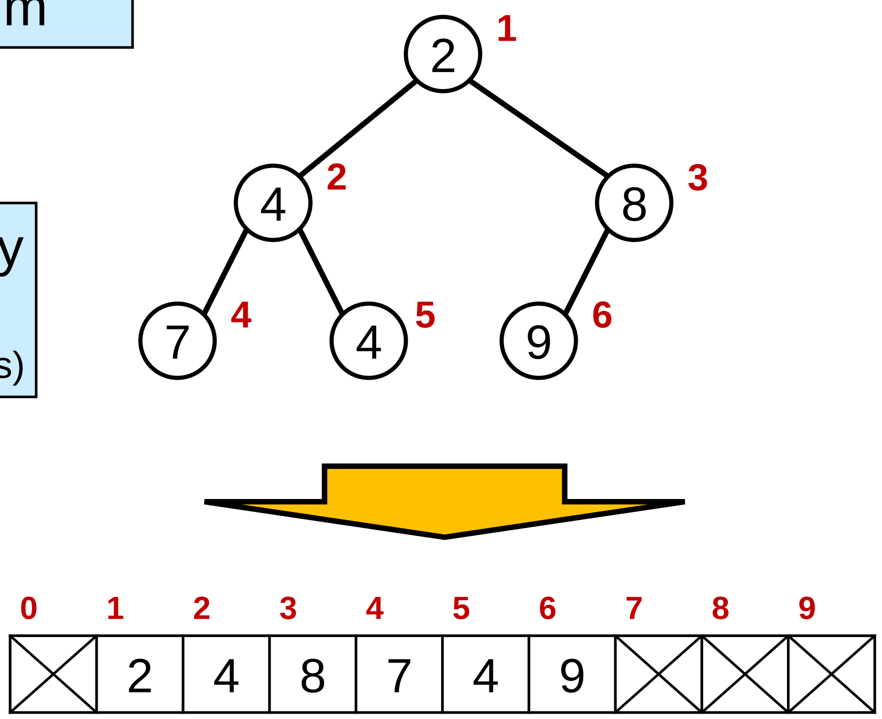
```
typedef struct heap_header heap;
struct heap_header {
  int limit;          // == capacity + 1
  elem[] data;  // \length(data) == limit
  int next;          // 1 <= next  && next <= limit
  has_higher_priority_fn* prior; // != NULL
};
```

O(1)

```
bool pq_full(heap* H)
//@requires is_heap(H);
//@ensures \result == (H->next == H->limit);
{
  return H->next == H->limit;
}
```

We can fill a **bounded** heap to the brim

Implementation-only postcondition (will come in handy in proofs)



O(1)

```
bool pq_empty(heap* H)
//@requires is_heap(H);
{
  return H->next == 1;
}
```

O(1)

```
elem pq_peek(heap* H)
//@requires is_heap(H) && !pq_empty(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  return H->data[1];
}
```

We sacrificed index 0

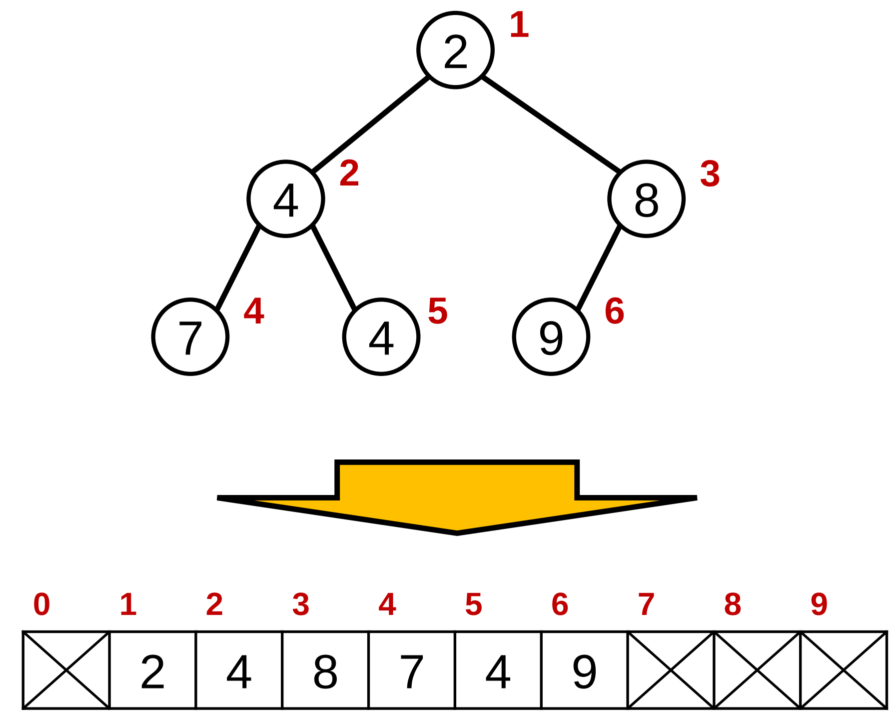
pq_new

```
typedef struct heap_header heap;
struct heap_header {
  int limit;         // == capacity + 1
  elem[] data;  // \length(data) == limit
  int next;         // 1 <= next  && next <= limit
  has_higher_priority_fn* prior; // != NULL
};
```

O(1)

```
heap* pq_new(int capacity, has_higher_priority_fn* prior)
//@requires 0 < capacity && capacity <= int_max()/2 - 1;
//@requires prior != NULL;
//@ensures is_heap(\result);
{
  heap* H = alloc(heap);
  H->limit = capacity + 1;
  H->next = 1;
  H->data = alloc_array(elem, H->limit);
  H->prior = prior;
  return H;
}
```

Overflow!



- To preempt overflow, we must have

  1 < H->limit && H->limit <= int_max()/2

  but H->limit == capacity + 1
- so

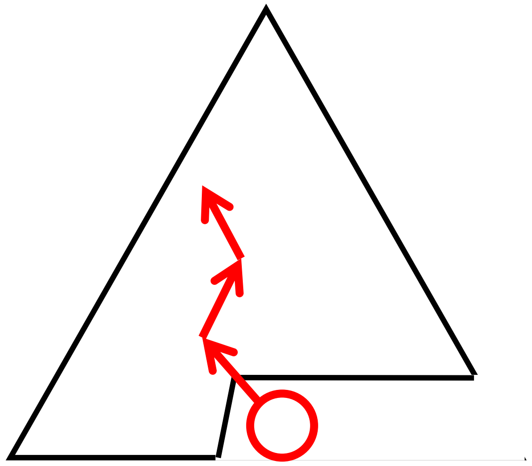
  0 < capacity && capacity <= int_max()/2 - 1

Implementing pq_add

```
typedef struct heap_header heap;
struct heap_header {
  int limit;         // == capacity + 1
  elem[] data;  // \length(data) == limit
  int next;         // 1 <= next  && next <= limit
  has_higher_priority_fn* prior; // != NULL
};
```

pq_add

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;

  int i = H->next - 1;
  while (i > 1)              // sifting up
  {
    int parent = i/2;
    if (ok_above(H, parent, i))
      return;                // no more violations
    swap_up(H, i);
    i = parent;
  }
  return;    // reached the root
}
```



- place the new element in the leftmost open position in the last level to satisfy the shape invariant
- sift up to restore the ordering invariant

Is this code safe?

Safety

```
typedef struct heap_header heap;
struct heap_header {
  int limit;        // == capacity + 1
  elem[] data;  // \length(data) == limit
  int next;         // 1 <= next  && next <= limit
  has_higher_priority_fn* prior; // != NULL
};
```

- Potential safety concerns
  - H is not NULL
  - array access shall be in bound
  - ok_above has preconditions
  - swap_up
    - we haven't implemented it yet

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;

  int i = H->next - 1;
  while (i > 1)           // sifting up
  {
    int parent = i/2;
    if (ok_above(H, parent, i))
      return;              // no more violations
    swap_up(H, i);
    i = parent;
  }
  return;     // reached the root
}
```

Safety

```
typedef struct heap_header heap;
struct heap_header {
  int limit;        // == capacity + 1
  elem[] data;  // \length(data) == limit
  int next;        // 1 <= next  && next <= limit
  has_higher_priority_fn* prior; // != NULL
};
```

- H is not NULL
  - **To show:** H != NULL
    - ➢ is_heap(H) by precondition
    - ➢ is_heap_safe(H) by def. of is_heap
    - ➢ H != NULL by def. of is_heap_safe

✓

```
bool is_heap_safe(heap* H) {
  return H != NULL
      && 1 < H->limit && H->limit <= int_max()/2
      && is_array_expected_length(H->data, H->limit)
      && 1 <= H->next && H->next <= H->limit
      && H->prior != NULL;
}
```

```
bool is_heap(heap* H) {
  return is_heap_safe(H)
      && is_heap_ordered(H);
}
```

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;

  int i = H->next - 1;
  while (i > 1)          // sifting up
  {
    int parent = i/2;
    if (ok_above(H, parent, i))
      return;              // no more violations
    swap_up(H, i);
    i = parent;
  }
  return;     // reached the root
}
```

Safety

```
typedef struct heap_header heap;
struct heap_header {
  int limit;      // == capacity + 1
  elem[] data;  // \length(data) == limit
  int next;       // 1 <= next  && next <= limit
  has_higher_priority_fn* prior; // != NULL
};
```

- Array access shall be in bound
  - **To show:** 0 ≤ H->next
    - 1 ≤ H->next by is_heap(H)
    - 0 ≤ H->next by math ✓
  - **To show:** H->next < H->limit
    - H->next ≤ H->limit by is_heap(H)
    - H->next ≠ H->limit by !pq_full(H)
    - H->next < H->limit by math ✓

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;

  int i = H->next - 1;
  while (i > 1)          // sifting up
  {
    int parent = i/2;
    if (ok_above(H, parent, i))
      return;              // no more violations
    swap_up(H, i);
    i = parent;
  }
  return;    // reached the root
}
```

```
bool is_heap_safe(heap* H) {
  return H != NULL
      && 1 < H->limit && H->limit <= int_max()/2
      && is_array_expected_length(H->data, H->limit)
      && 1 <= H->next && H->next <= H->limit
      && H->prior != NULL;
}
```

```
bool pq_full(
//@requires
{
  return H->next == H->limit;
}
```

Safety

```
typedef struct heap_header heap;
struct heap_header {
  int limit;       // == capacity + 1
  elem[] data;     // \length(data) == limit
  int next;        // 1 <= next  && next <= limit
  has_higher_priority_fn* prior; // != NULL
};
```

- Are the array accesses still in bound **after** we modify H->next ?
  - More generally, is the heap still safe?
    - is is_heap_safe(H) still valid after we increment H->next?
  - **To show:** is_heap_safe(H)
    - No field constraint is affected except next <= limit
    - **To show:** H->next ≤ H->limit
      - right after (H->next)++

❑ H->next ≤ H->limit before	by is_heap(H)
❑ H->next ≠ H->limit before	by !pq_full(H)
❑ H->next < H->limit before	by math
❑ H->next ≤ H->limit after	by math
❑ is_heap_safe(H) after	

✓

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;                    <= is heap safe?

  int i = H->next - 1;
  while (i > 1)            // sifting up
  {
    int parent = i/2;
    if (ok_above(H, parent, i))
      return;              // no more violations
    swap_up(H, i);
    i = parent;
  }
  return;     // reached the root
}
```

```
bool ok_above(heap* H, int i1, int i2)
//@requires is_heap_safe(H);
//@requires 1 <= i1 && i1 < H->next;
//@requires 1 <= i2 && i2 < H->next;
```

Safety

- Preconditions of ok_above are met
  - **To show:** is_heap_safe(H) ✓
    - by new assertion
  - **To show:** 1 ≤ i ✓
    - 1 < i by loop guard
    - 1 ≤ i by math
  - **To show:** i < H->next ✗
    - **?**

      We have nothing to point to!

      Add a loop invariant!
  - **To show:** 1 ≤ parent
  - **To show:** parent < H->next

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;
  //@assert is_heap_safe(H);        Yes!
  int i = H->next - 1;
  while (i > 1)          // sifting up
  {
    int parent = i/2;
    if (ok_above(H, parent, i))
      return;            // no more violations
    swap_up(H, i);
    i = parent;
  }
  return;     // reached the root
}
```

```
bool ok_above(heap* H, int i1, int i2)
//@requires is_heap_safe(H);
//@requires 1 <= i1 && i1 < H->next;
//@requires 1 <= i2 && i2 < H->next;
```

Safety

- Preconditions of ok_above are met
  - **To show:** is_heap_safe(H) ✓
  - **To show:** 1 ≤ i ✓
  - **To show:** i < H->next
    - i < H->next   by LI ✓
  - **To show:** 1 ≤ parent
    - parent = i/2   by code
    - 1 < i   by loop guard
    - 1 ≤ i/2   by math ✓
  - **To show:** parent < H->next
    - i < H->next   by LI
    - i/2 < H->next   by math ✓

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;
  //@assert is_heap_safe(H);
  int i = H->next - 1;
  while (i > 1)              // sifting up
  //@loop_invariant 1 <= i && i < H->next;
  {
    int parent = i/2;
    if (ok_above(H, parent, i))
      return;                // no more violations
    swap_up(H, i);
    i = parent;
  }
  return;    // reached the root
}
```

Validity left as exercise

Safety

- Preconditions of swap_up are met
- Code for swap_up
  - This takes the point of view of a **child** node
    - all nodes are children except the root
      - 2 <= child

```
void swap_up(heap* H, int child)
//@requires is_heap_safe(H);
//@requires 2 <= child && child < H->next;
//@requires !ok_above(H, child/2, child);
//@ensures ok_above(H, child/2, child);
{
  int parent = child/2;
  elem tmp = H->data[child];
  H->data[child] = H->data[parent];
  H->data[parent] = tmp;
}
```

H is safe, but …

… it has an ordering violation at child

swap_up fixes this ordering violation

Safety

```
void swap_up(heap* H, int child)
//@requires is_heap_safe(H);
//@requires 2 <= child && child < H->next;
//@requires !ok_above(H, child/2, child);
//@ensures ok_above(H, child/2, child);
```

- Preconditions of swap_up are met
  - **To show:** is_heap_safe(H) ✓
  - **To show:** 2 ≤ i && i < H->next ✓
  - **To show:** !ok_above(H, i/2, i) ✓
    - parent = i/2 by code
    - !ok_above(H, parent, i) by conditional

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;
  //@assert is_heap_safe(H);
  int i = H->next - 1;
  while (i > 1)            // sifting up
   //@loop_invariant 1 <= i && i < H->next;
   {
    int parent = i/2;
    if (ok_above(H, parent, i))
       return;             // no more violations
    swap_up(H, i);
    i = parent;
  }
  return;    // reached the root
}
```

Correctness of pq_add

Is this Code Correct?

```
bool is_heap(heap* H) {
  return is_heap_safe(H)
      && is_heap_ordered(H);
}
```

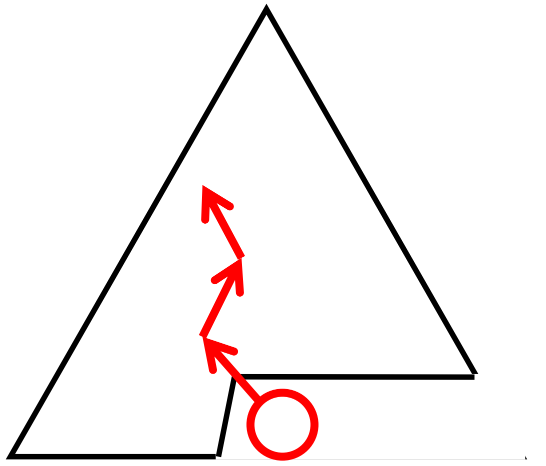
- **To show:** !pq_empty(H) ✓

  Left as exercise

- **To show:** is_heap(H)
  - **To show:** is_heap_safe(H) ✓
  - **To show:** is_heap_ordered(H)

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;
  //@assert is_heap_safe(H);
  int i = H->next - 1;
  while (i > 1)            // sifting up
   //@loop_invariant 1 <= i && i < H->next;
   {
    int parent = i/2;
    if (ok_above(H, parent, i))
       return;             // no more violations
    swap_up(H, i);
    i = parent;
  }
  return;    // reached the root
}
```

Is this Code Correct?

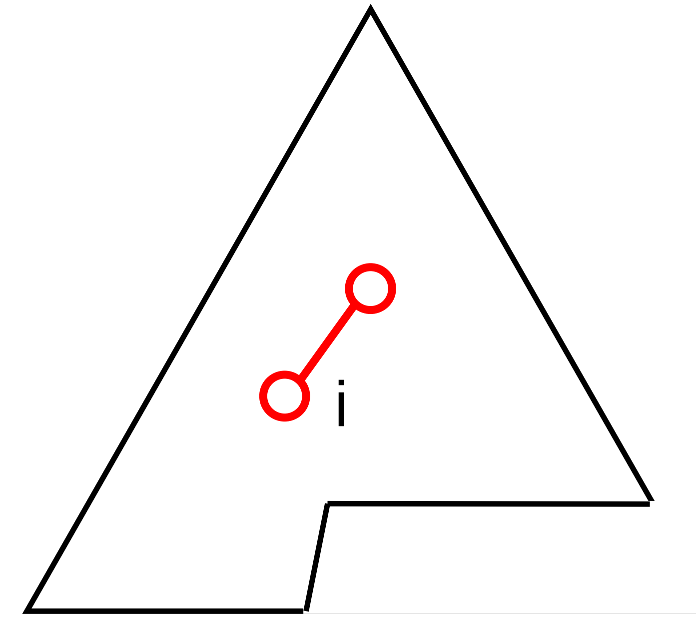


- **To show:** is_heap_ordered(H)
  - We have nowhere to point to! ✘
  - Our usual solution is to add it as an additional loop invariant

    //@loop_invariant is_heap_ordered(H);
  - *But is it valid?*
    - No!
    - We are in the midst of restoring the ordering invariant that we have potentially just broken
      - It will not hold in general

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;
  //@assert is_heap_safe(H);
  int i = H->next - 1;
  while (i > 1)              // sifting up
   //@loop_invariant 1 <= i && i < H->next;
   {
    int parent = i/2;
    if (ok_above(H, parent, i))
       return;               // no more violations
    swap_up(H, i);
    i = parent;
  }
  return;    // reached the root
}
```

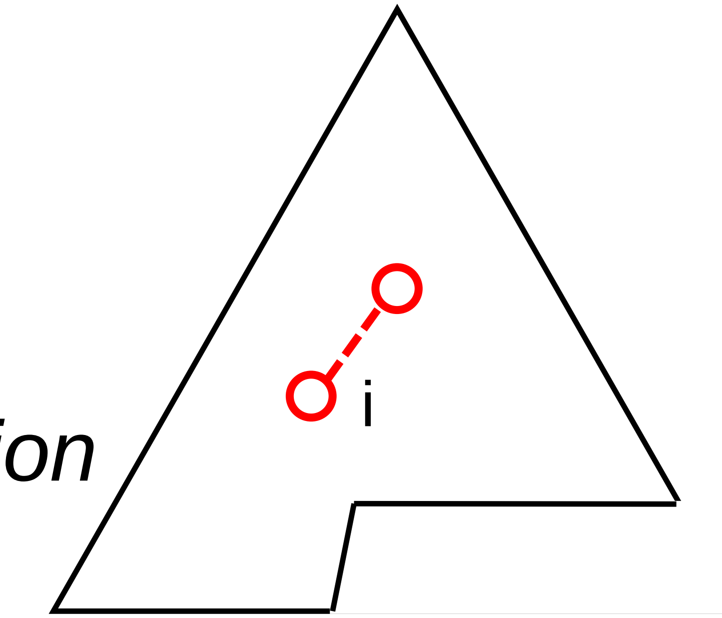
Is this Code Correct?



- **To show:** is_heap_ordered(H)
  - *We are in the midst of restoring the ordering invariant that we have potentially just broken*
  - Can we come up with another loop invariant that can serve our purpose?
  - Note that the ordering invariant **almost** works
    - while sifting up, there is at most one violation
    - and it occurs between i and its parent

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;
  //@assert is_heap_safe(H);
  int i = H->next - 1;
  while (i > 1)             // sifting up
   //@loop_invariant 1 <= i && i < H->next;
   {
    int parent = i/2;
    if (ok_above(H, parent, i))
       return;              // no more violations
    swap_up(H, i);
    i = parent;
  }
  return;    // reached the root
}
```

Weakening the Invariant



- *While sifting up, there is **at most one** violation*
  - *and it occurs between i and its parent*

- Capture this in a weakened version of is_heap_ordered

```
bool is_heap_except_up(heap* H, int x)
//@requires is_heap_safe(H);
//@requires 1 <= x && x < H->next;
{
  for (int child = 2; child < H->next; child++)
  //@loop_invariant 2 <= child && child <= H->next;
  {
    int parent = child/2;
    if (!(child == x ||
          ok_above(H, parent, child)))
       return false;
  }
  return true;
}
```

Exception

Allowed exception at x

- This is the code of is_heap_ordered except that it skips over x
  - if there is a violation there, it turns a blind eye
  - but no other violations are permitted

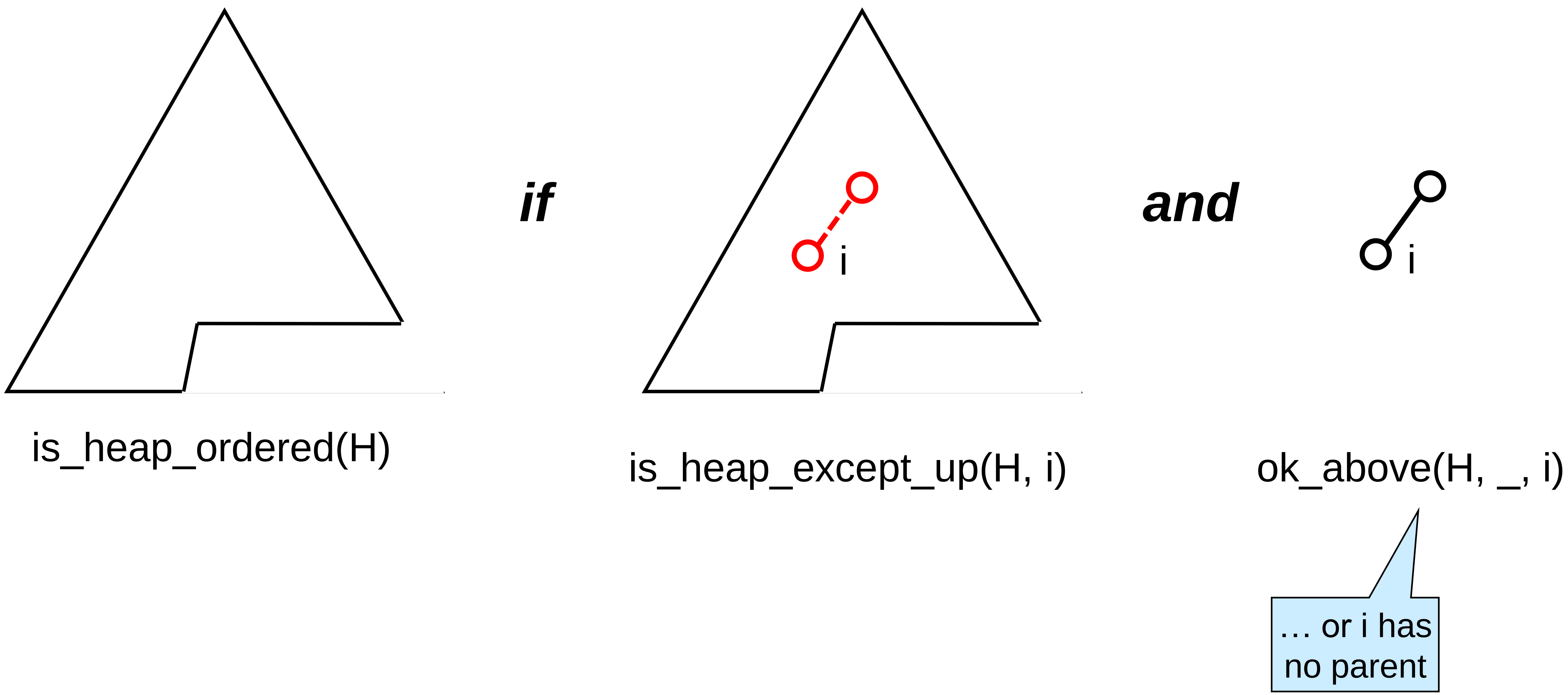
Is this Code Correct?

- **To show:** is_heap_ordered(H)
  - we added a loop invariant
  - This must be true everywhere the function returns
    - inside the loop
    - after the loop

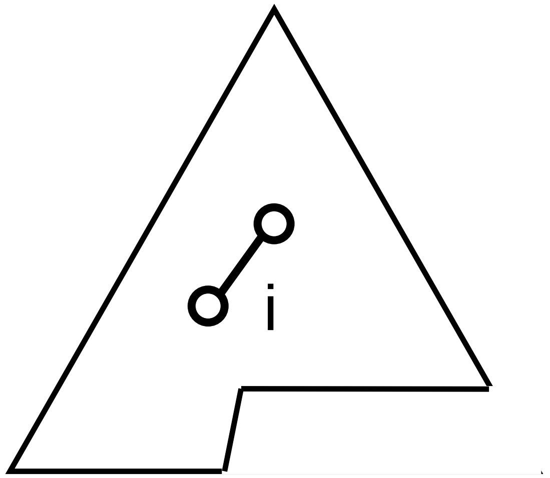
```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;
  //@assert is_heap_safe(H);
  int i = H->next - 1;
  while (i > 1)            // sifting up
   //@loop_invariant 1 <= i && i < H->next;
   //@loop_invariant is_heap_except_up(H, i);
   {
    int parent = i/2;
    if (ok_above(H, parent, i))
       return;             // no more violations
    swap_up(H, i);
    i = parent;
  }
  return;    // reached the root
}
```

Using the Weakened Loop Invariant

- The heap is ordered ***if***
  - it is ordered everywhere except possibly at i ***and***
  - it is actually ordered also at i



Is this Code Correct?



- **To show:** is_heap_ordered(H)
  - when we return inside the loop
    - is_heap_except_up(H, i) by LI-2

```
bool is_heap_except_up(heap* H, int x) {
  for (int child = 2; child < H->next; child++) {
    int parent = child/2;
    if (!(child == x ||
          ok_above(H, parent, child)))
      return false;
  }
  return true;
}
```

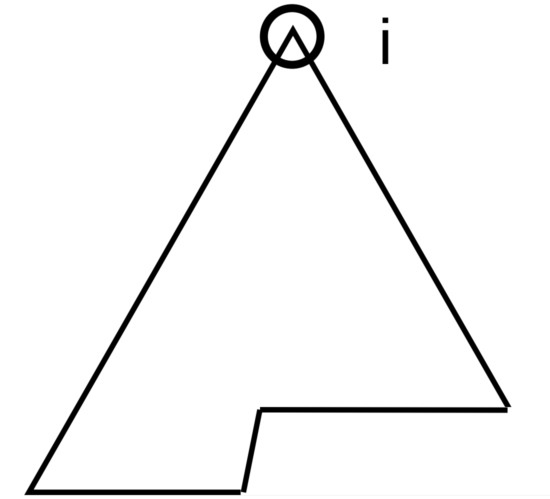
Contracts omitted for succinctness

    - *i is the one allowed exception*
    - ok_above(H, parent, i) by conditional
      - there is **no violation** at i
    - is_heap_ordered(H) by above

✓

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;
  //@assert is_heap_safe(H);
  int i = H->next - 1;
  while (i > 1)           // sifting up
   //@loop_invariant 1 <= i && i < H->next;
   //@loop_invariant is_heap_except_up(H, i);
   {
    int parent = i/2;
    if (ok_above(H, parent, i))
      return;             // no more violations
    swap_up(H, i);
    i = parent;
  }
  return;    // reached the root
}
```

Is this Code Correct?



- **To show:** is_heap_ordered(H)
  - when we return after the loop
    - i == 1 by loop guard and LI-1
    - is_heap_except_up(H, i) by LI-2

Contracts omitted for succinctness

```
bool is_heap_except_up(heap* H, int x) {
  for (int child = 2; child < H->next; child++)  {
    int parent = child/2;
    if (!(child == x ||
          ok_above(H, parent, child)))
      return false;
  }
  return true;
}
```

    - *child starts at 2*
      - *but x is 1*
      - *the root has no parent where to have a violation*
    - is_heap_ordered(H) by above

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;
  //@assert is_heap_safe(H);
  int i = H->next - 1;
  while (i > 1)            // sifting up
   //@loop_invariant 1 <= i && i < H->next;
   //@loop_invariant is_heap_except_up(H, i);
   {

  }
  return;   // reached the root
}
```

Is this Code Correct?

- **To show:** !pq_empty(H) ✓

Left as exercise

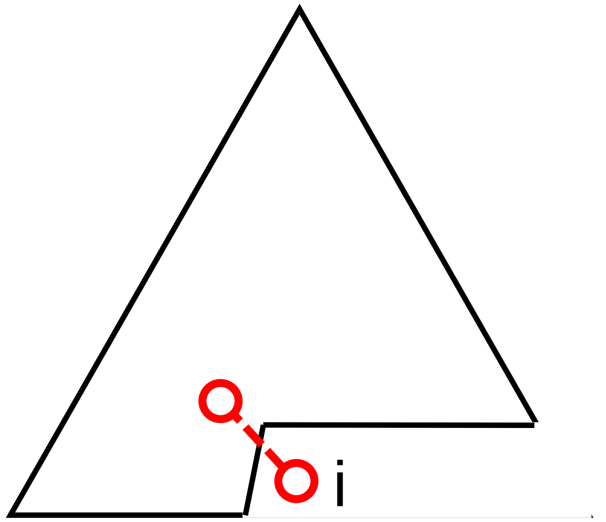
- **To show:** is_heap(H)
  - **To show:** is_heap_safe(H) ✓
  - **To show:** is_heap_ordered(H)

We still need to show that the new loop invariant is valid

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;
  //@assert is_heap_safe(H);
  int i = H->next - 1;
  while (i > 1)           // sifting up
   //@loop_invariant 1 <= i && i < H->next;
   //@loop_invariant is_heap_except_up(H, i);
   {
    int parent = i/2;
    if (ok_above(H, parent, i))
       return;            // no more violations
    swap_up(H, i);
    i = parent;
  }
  //@assert i == 1;
  return;     // reached the root
}
```

Proving the Loop Invariant

Initialization

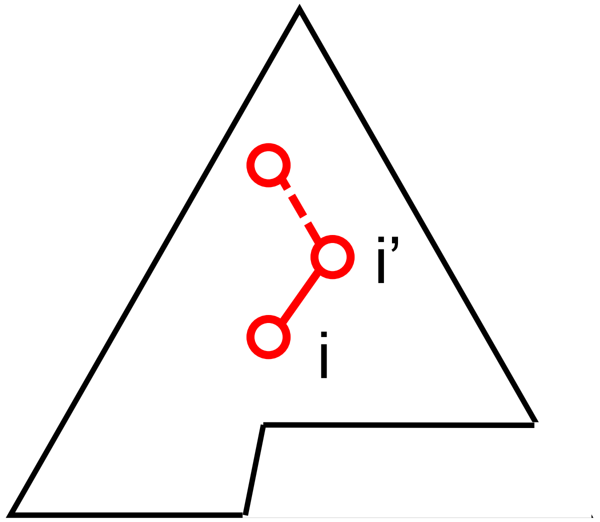


INIT:

- **To show:** is_heap_except_up(H, i) holds initially
  - *refer to H->next before the increment*
  - ➢ is_heap_ordered(H) by is_heap(H)
  - ➢ i == H->next by code
  - ➢ *i is the one allowed exception*
  - ➢ is_heap_except_up(H, i)

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;
  //@assert is_heap_safe(H);
  int i = H->next - 1;
  while (i > 1)            // sifting up
   //@loop_invariant 1 <= i && i < H->next;
   //@loop_invariant is_heap_except_up(H, i);
   {

  }
  //@assert i == 1;
  return;    // reached the root
}
```

Preservation



PRES:

- **To show:**

  if is_heap_except_up(H, i)
  then is_heap_except_up(H, i')

- The proof proceeds by cases on whether
  - i is a left or right child
  - i is the root
  - i has children and how many

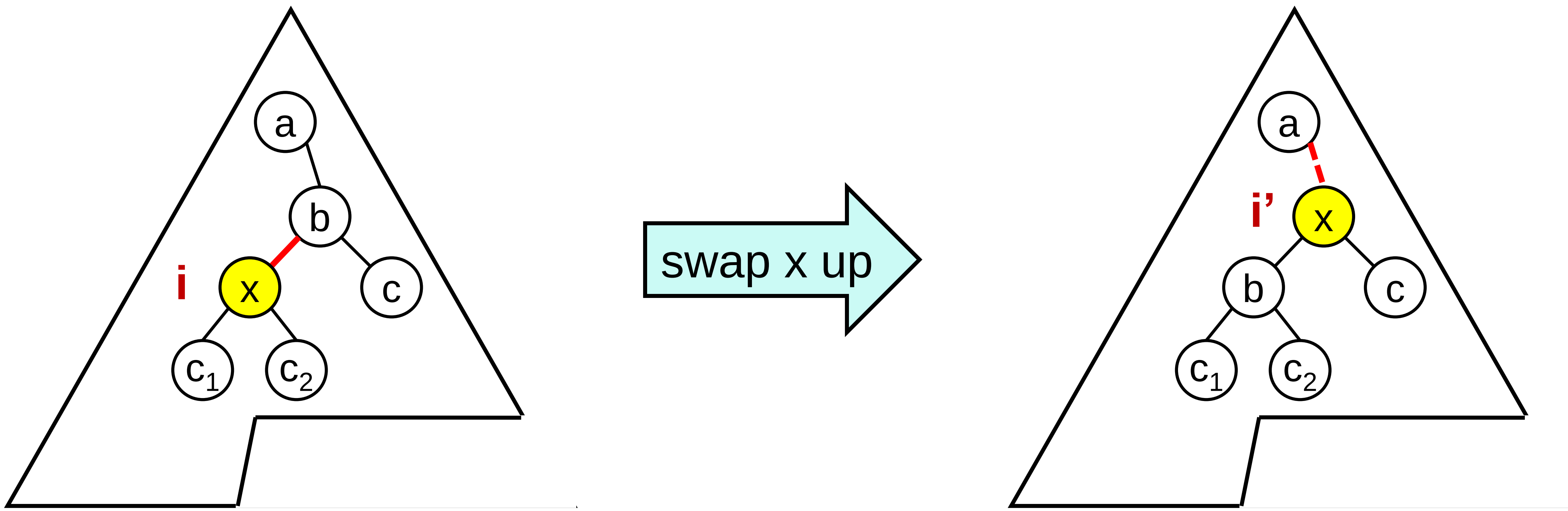
- *We examine one representative case*

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;
  //@assert is_heap_safe(H);
  int i = H->next - 1;
  while (i > 1)          // sifting up
   //@loop_invariant 1 <= i && i < H->next;
   //@loop_invariant is_heap_except_up(H, i);
   {
    int parent = i/2;
    if (ok_above(H, parent, i))
       return;              // no more violations
    swap_up(H, i);
    i = parent;
  }
  //@assert i == 1;
  return;    // reached the root
}
```

Preservation

To show: *if* is_heap_except_up(H, i), *then* is_heap_except_up(H, i')

- We examine one representative case



1. $a \le b$ (order)
2. $b \le c$ (order)
3. $x < b$ (since we swap)
4. $x \le c_1$ (order)
5. $x \le c_2$ (order)

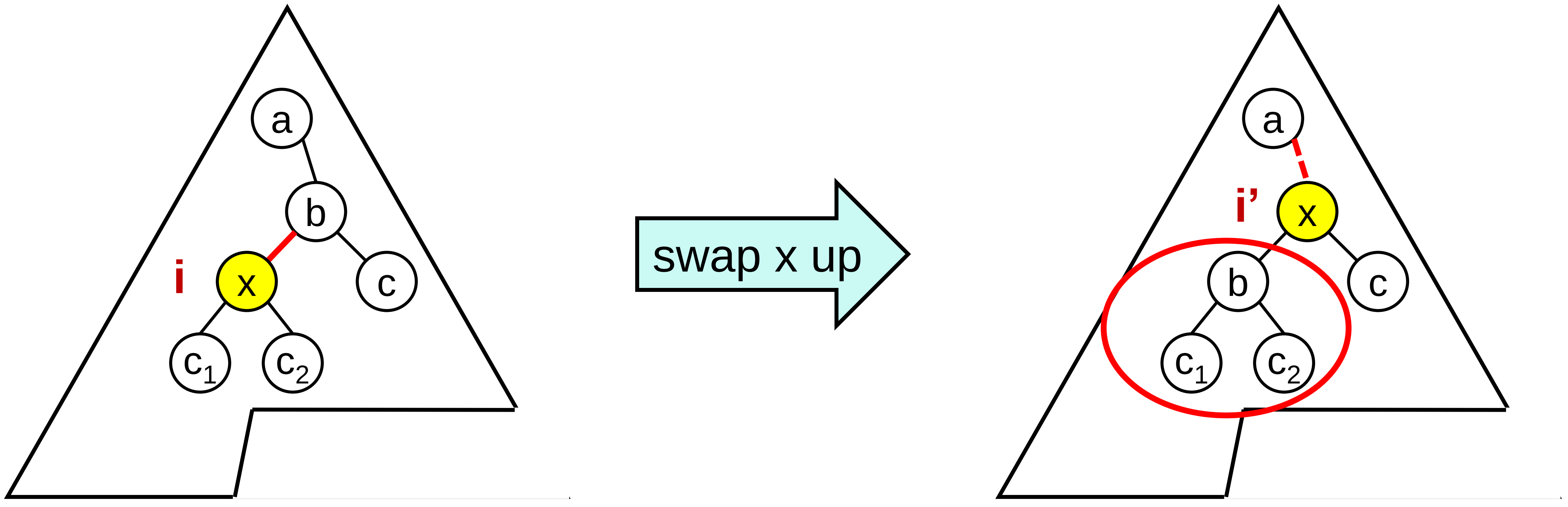
as in a min-heap

swap x up

i. $a\ ?\ x$ (allowed exception)
ii. $x \le c$ (by 3 and 2)
iii. $x \le b$ (by 3)
iv. $b \le c_1$ (??)
v. $b \le c_2$ (??)

We lack supporting evidence

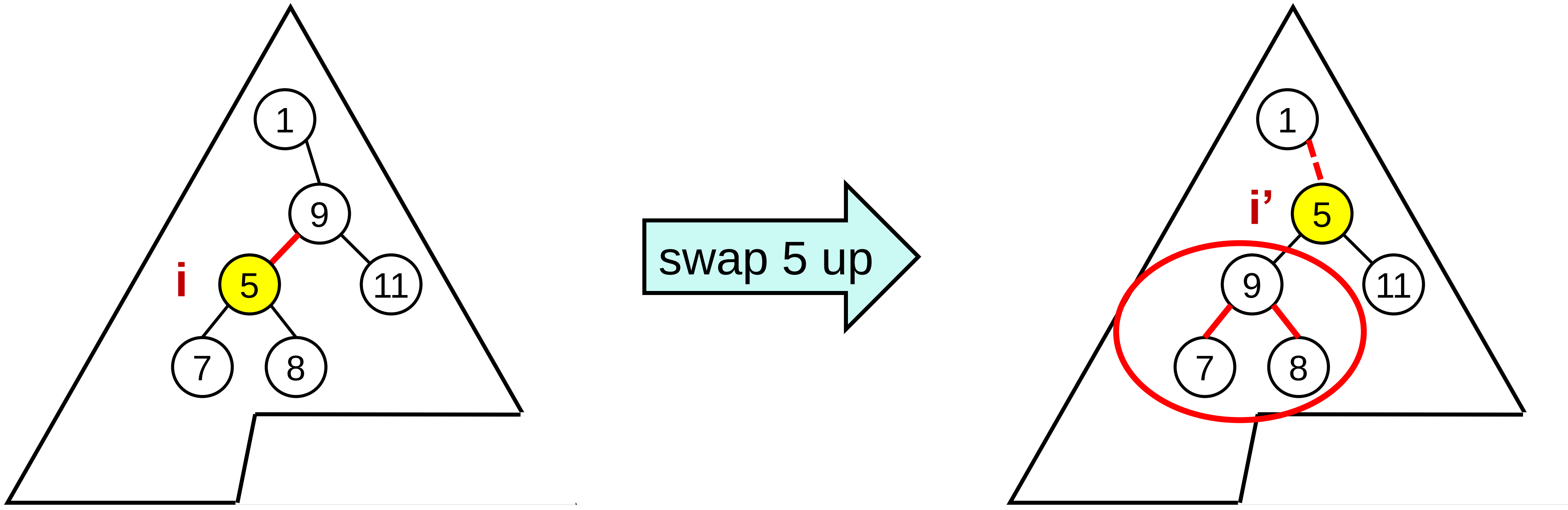
Preservation



- We cannot prove that $b \leq c_1$ and $b \leq c_2$
  - either our current loop invariant are insufficient
    - incorrect or weak
  - or our implementation is incorrect

Can our Loop Invariant be Wrong?

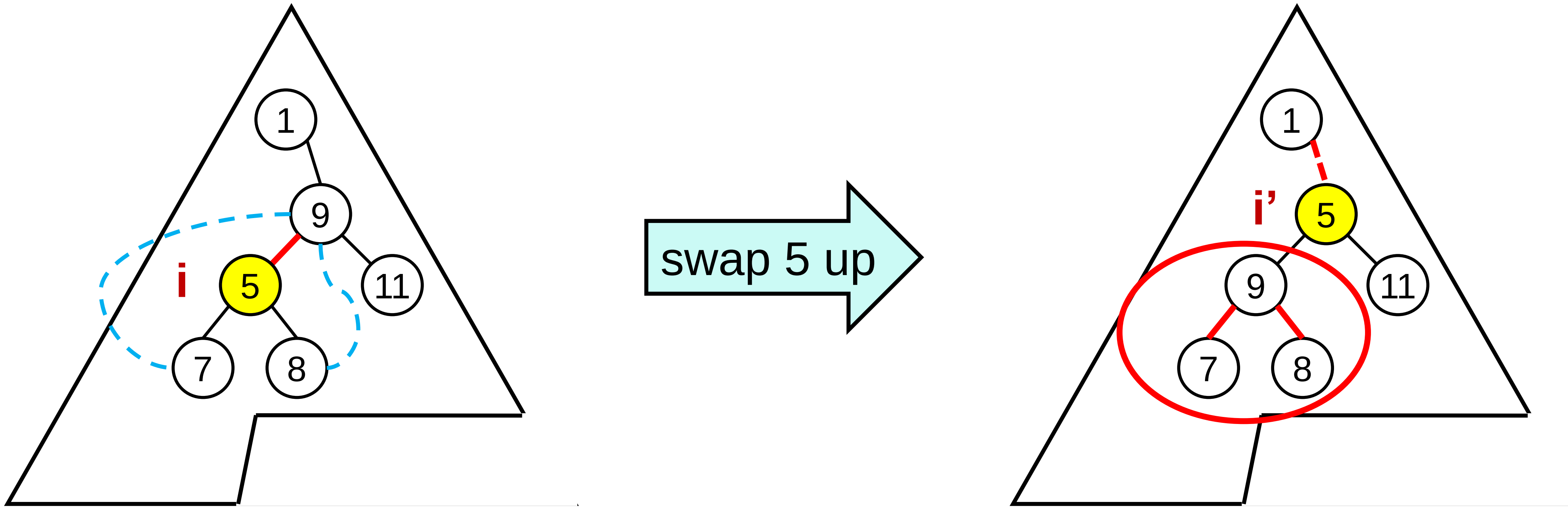
- Counterexample



- But on the previous swap up, either 7 or 8 would have been below 9
  - there would have been another violation above i
  - is_heap_except_up would have failed

Can our Loop Invariant be Wrong?

- Counterexample?



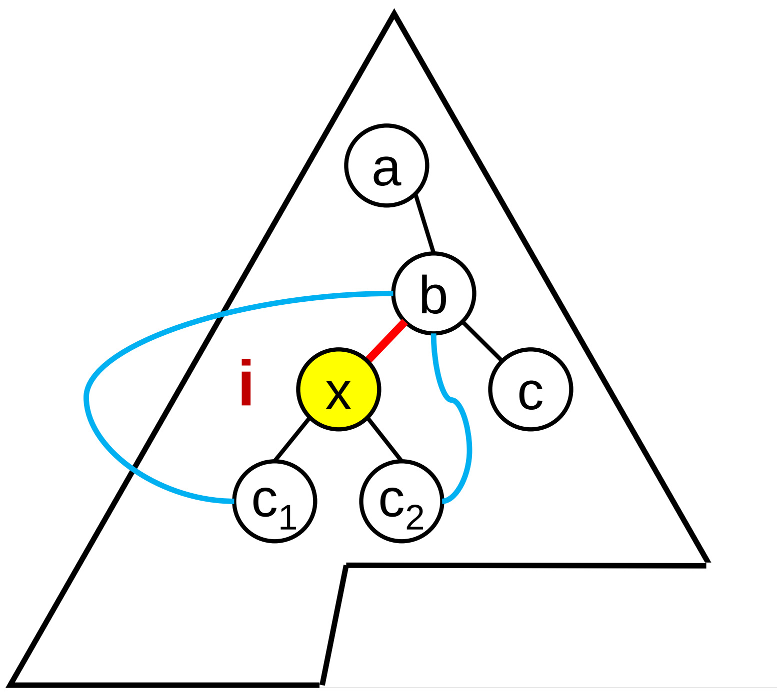
- This should not be possible
  - we should have had 9 ≤ 8 and 9 ≤ 7

- We can capture this with a new loop invariant

  //@loop_invariant grandparent_check(H, i);

Updated Code

- The parent of node i is Ok above the children of i

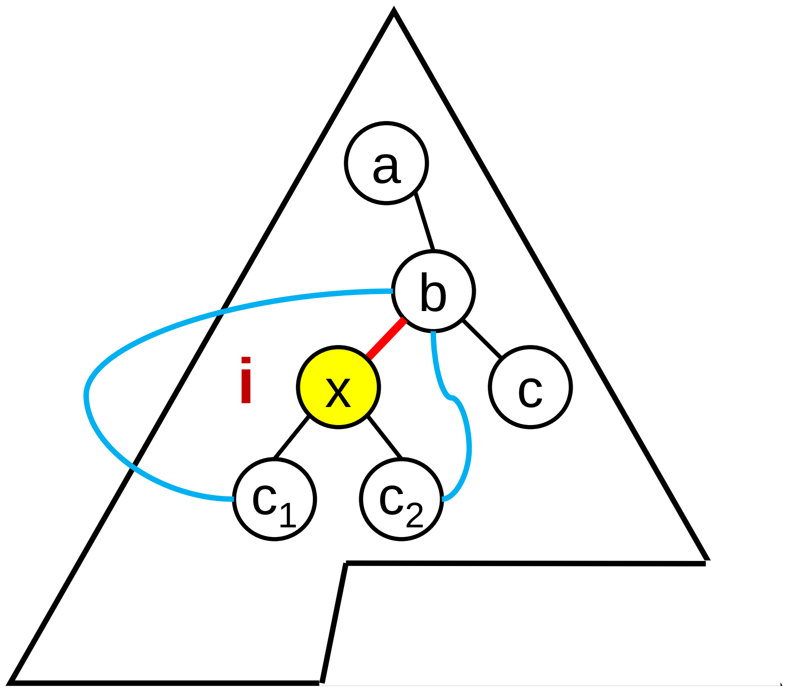


We call this the **grandparent check**

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;
   //@assert is_heap_safe(H);
  int i = H->next - 1;
  while (i > 1)            // sifting up
  //@loop_invariant 1 <= i && i < H->next;
  //@loop_invariant is_heap_except_up(H, i);
  //@loop_invariant grandparent_check(H, i);
  {
    int parent = i/2;
    if (ok_above(H, parent, i))
       return;             // no more violations
    swap_up(H, i);
    i = parent;
  }
  //@assert i == 1;
  return;    // reached the root
}
```

The Grandparent Check

- The parent of node i is Ok above the children of i

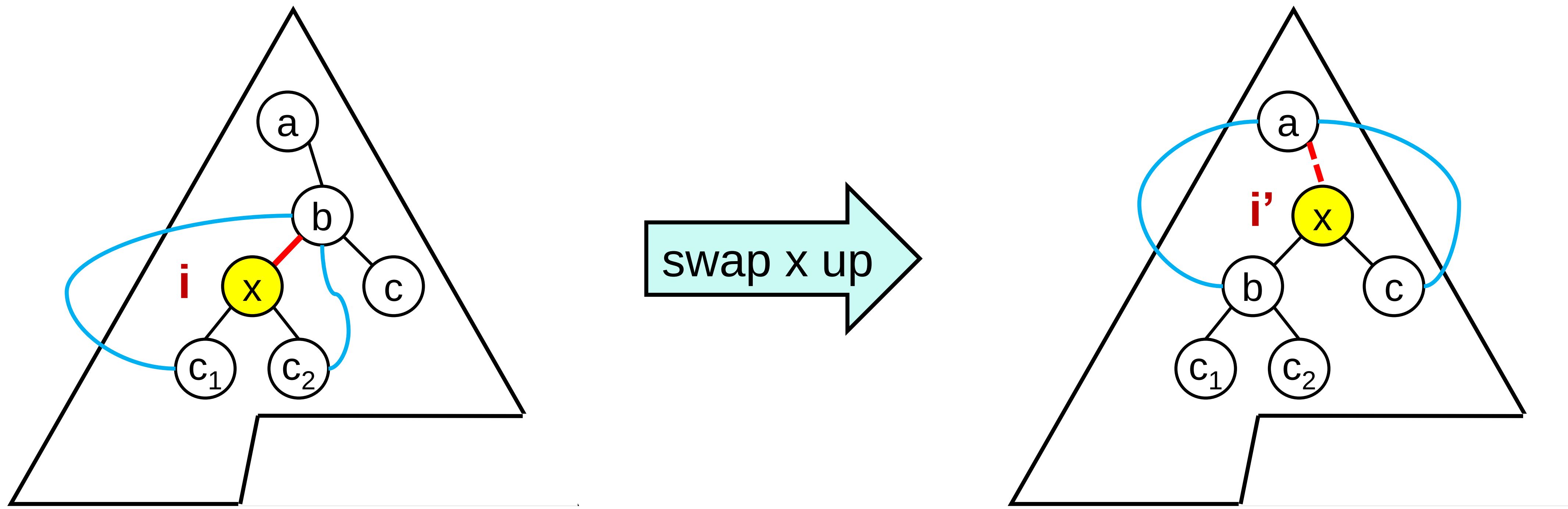



```
bool grandparent_check(heap* H, int i)
//@requires is_heap_safe(H);
//@requires 1 <= i && i < H->next;
{
  int left = 2*i;
  int right = 2*i + 1;
  int grandparent = i/2;

  if (i == 1) return true;     // reached the root
  if (left >= H->next)         // no children
     return true;
  if (right == H->next)        // left child only
     return ok_above(H, grandparent, left);
  return right < H->next       // both children
       && ok_above(H, grandparent, left)
       && ok_above(H, grandparent, right);
}
```

Preservation

To show: *if* is_heap_except_up(H, i), *then* is_heap_except_up(H, i')

- We examine one representative case



1. $a \le b$ (order)
2. $b \le c$ (order)
3. $x < b$ (since we swap)
4. $x \le c_1$ (order)
5. $x \le c_2$ (order)
6. $b \le c_1$ (grandparent check)
7. $b \le c_2$ (grandparent check)

swap x up

i. a ? x (allowed exception)
ii. $x \le c$ (by 3 and 2)
iii. $x \le b$ (by 3)
iv. $b \le c_1$ (by 6)
v. $b \le c_2$ (by 7)
vi. $a \le b$ (by 1)
vii. $a \le c$ (by 1 and 2)

This proves preservation for the new grandparent_check loop invariant

Is this Code Correct?

- **To show:** !pq_empty(H) ✓

Left as exercise

- **To show:** is_heap(H)
  - **To show:** is_heap_safe(H) ✓
  - **To show:** is_heap_ordered(H) ✓
    - **To show:** is_heap_except_up(H, i) ✓
    - **To show:** grandparent_check(H, i) ✓

- This concludes the proof that pq_add is correct
  - apart from the exercises

✓

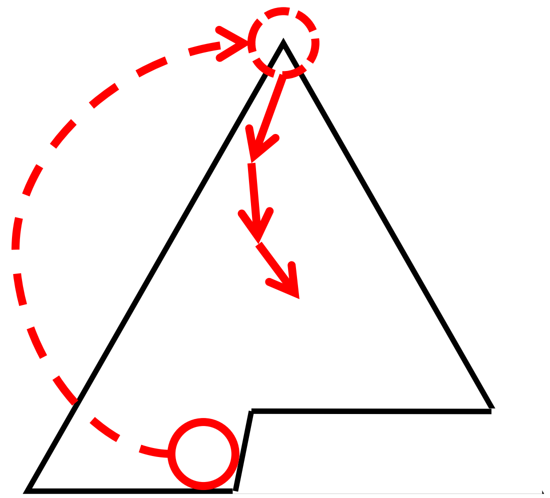
```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H) && !pq_empty(H);
{
  H->data[H->next] = e;
  (H->next)++;
   //@assert is_heap_safe(H);
  int i = H->next - 1;
  while (i > 1)            // sifting up
  //@loop_invariant 1 <= i && i < H->next;
  //@loop_invariant is_heap_except_up(H, i);
  //@loop_invariant grandparent_check(H, i);
  {
    int parent = i/2;
    if (ok_above(H, parent, i))
       return;             // no more violations
    swap_up(H, i);
    i = parent;
  }
   //@assert i == 1;
   return;   // reached the root
}
```

Implementing pq_rem

pq_rem

```
elem pq_rem(heap* H)
//@requires is_heap(H) && !pq_empty(H);
//@ensures is_heap(H) && !pq_full(H);
{
  elem min = H->data[1];
  (H->next)--;

  if (H->next > 1) {
    H->data[1] = H->data[H->next];
    // the ordering invariant may not hold
    sift_down(H);
  }
  return min;
}
```

We replace the root and sift down only if the updated heap is non-empty



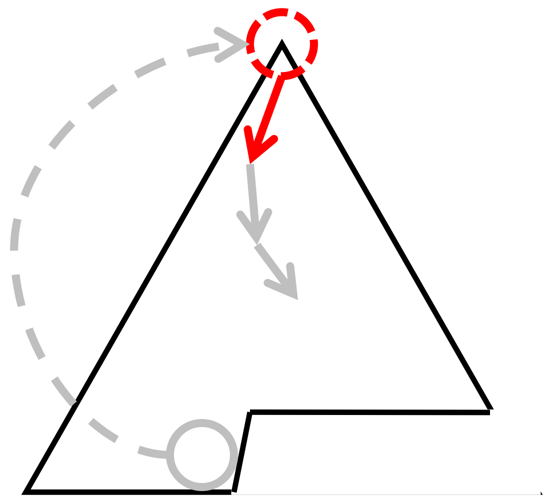
- replace the root with the element in the rightmost filled position on the last level to satisfy the shape invariant
  - the root is H->data[1]
  - that position is H->next – 1
- sift down to restore the ordering invariant
  - we implement it as a separate function

sift_down

```
void sift_down(heap* H)
//@requires is_heap_safe(H);
//@requires H->next > 1 && is_heap_except_down(H, 1);
//@ensures is_heap(H);
{
  int i = 1;
  // …
}
```

root

- To sift down
  - the heap needs to be non-empty
    - H->next > 1
  - the heap is safe
    - is_heap_safe(H)
  - the ordering invariant holds except at the root
    - is_heap_except_down(H, 1)



```
bool is_heap_except_down(heap* H, int x)
//@requires is_heap_safe(H);
//@requires 1 <= x && x < H->next;
{
  for (int child = 2; child < H->next; child++)
  //@loop_invariant 2 <= child;
  {
    int parent = child/2;
    if (!(parent == x ||    // Allowed exception
          ok_above(H, parent, child))) return false;
  }
  return true;
}
```

  - Similar to is_heap_except_up
    - but this time it skips over the parent
      - not the child
    - this allows at most two violations

- sift_down restores the heap invariant

  //@ensures is_heap(H);

sift_down

```
void sift_down(heap* H)
//@requires is_heap_safe(H);
//@requires H->next > 1 && is_heap_except_down(H, 1);
//@ensures is_heap(H);
{
  int i = 1;

  while (2*i < H->next)
  //@loop_invariant 1 <= i && i < H->next;
  //@loop_invariant is_heap_except_down(H, i);
  //@loop_invariant grandparent_check(H, i);
  {
    // …
  }
}
```

i is the index of the **parent** we are currently examining

- As we swap down, the last child we may consider is on the last level

  2*i < H->next

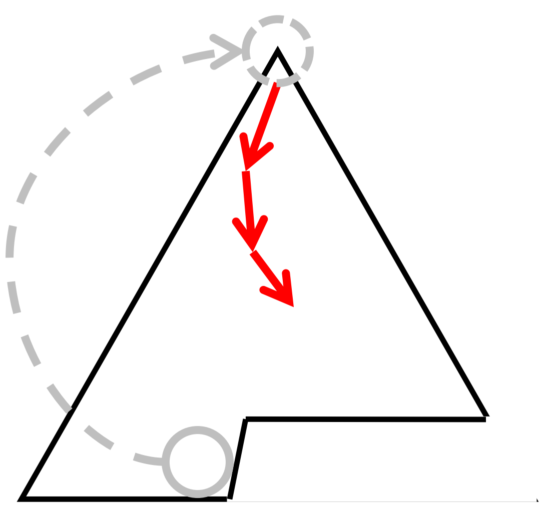
  - 2*i is the left child of i
  - H->next is on the last level

- In an arbitrary iteration
  - the parent must be in bounds

    1 <= i && i < H->next
  - there may be violations down from the parent

    is_heap_except_down(H, i);
  - the parent's parent should be Ok above the children

    grandparent_check(H, i)



sift_down

- If there are no more violations, return early
- Otherwise
  - identify which child to swap
  - swap it up with i
  - examine this child

```
void sift_down(heap* H)
//@requires is_heap_safe(H);
//@requires H->next > 1 && is_heap_except_down(H, 1);
//@ensures is_heap(H);
{
  int i = 1;

  while (2*i < H->next)
  //@loop_invariant 1 <= i && i < H->next;
  //@loop_invariant is_heap_except_down(H, i);
  //@loop_invariant grandparent_check(H, i);
  {
    // Are we done yet?
    if (done_sifting_down(H, i)) return; // No more violations

    // Let's swap!
    int p = child_to_swap_up(H, i);
    swap_up(H, p);
    i = p;
  }
  //@assert i < H->next && 2*i >= H->next;
}
```

i is the index of the **parent** we are currently examining

Are we done Fixing Violations?

- We need to consider several situations
  - i has only a left child
  - i has both children

```
bool done_sifting_down(heap* H, int i)
//@requires is_heap_safe(H);
//@requires 1 <= i && 2*i < H->next;        // i has at least one child
//@requires is_heap_except_down(H, i);  // violation is at i
{
  int left  = 2*i;
  int right = left+1;

  return ok_above(H, i, left)           // All good on the left, and
      && (right >= H->next              // either no right child
          || ok_above(H, i, right));    //    or all good on the right too
}
```

Identifying the Child to Swap

- We need to consider several situations
  - i has only a left child
  - i has both children

```
int child_to_swap_up(heap* H, int i)
//@requires is_heap_safe(H);
//@requires 1 <= i && 2*i < H->next;          // i has at least one child
//@requires is_heap_except_down(H, i); // violation is at i
//@ensures \result/2 == i;                    // returns a child
{
  int left  = 2*i;
  int right = left+1;

  if (right >= H->next ||                  // if no right child, or
      ok_above(H, left, right))            //    left child is smaller or equal
    return left;                           // then left child will go up
  //@assert right < H->next;               // if there is a right child, and
  //@assert ok_above(H, right, left);      //    right child is smaller or equal
  return right;                            // then right child will go up
}
```

min-heap terminology

Sifting Down

- Is this code safe?

  Left as exercise

- Is this code correct?

  Left as exercise

```
bool done_sifting_down(heap* H, int i)
//@requires is_heap_safe(H);
//@requires 1 <= i && 2*i < H->next;          // i has at least one child
//@requires is_heap_except_down(H, i);        // violation is at i
{
  int left  = 2*i;
  int right = left+1;

  return ok_above(H, i, left)                 // All good on the left, and
      && (right >= H->next                    // either no right child
          || ok_above(H, i, right));          //    or all good on the right too
}

int child_to_swap_up(heap* H, int i)
//@requires is_heap_safe(H);
//@requires 1 <= i && 2*i < H->next;          // i has at least one child
//@requires is_heap_except_down(H, i);        // violation is at i
//@ensures \result/2 == i;                    // returns a child
{
  int left  = 2*i;
  int right = left+1;

  if (right >= H->next ||                     // if no right child, or
      ok_above(H, left, right))               //   left child is smaller or equal
    return left;                              // then left child will go up
  //@assert right < H->next;                  // if there is a right child, and
  //@assert ok_above(H, right, left);         //   right child is smaller or equal
  return right;                               // then right child will go up
}

void sift_down(heap* H)
//@requires is_heap_safe(H);
//@requires H->next > 1 && is_heap_except_down(H, 1);
//@ensures is_heap(H);
{
  int i = 1;

  while (2*i < H->next)
  //@loop_invariant 1 <= i && i < H->next;
  //@loop_invariant is_heap_except_down(H, i);
  //@loop_invariant grandparent_check(H, i);
  {
    // Are we done yet?
    if (done_sifting_down(H, i)) return;      // No more violations

    // Let's swap!
    int p = child_to_swap_up(H, i);
    swap_up(H, p);
    i = p;
  }
  //@assert i < H->next && 2*i >= H->next;
}
```