Lecture 18
Restoring Invariants

15-122: Principles of Imperative Computation (Spring 2022)
Frank Pfenning

In this lecture we will implement heaps and operations on them. The theme of this lecture is reasoning with invariants that are partially violated, and making sure they are restored before the completion of an operation. We will only briefly review the algorithms for inserting and deleting the minimal node of the heap; you should read the notes for the previous lecture on priority queues and keep them close at hand.

Additional Resources

- Review slides (https://cs.cmu.edu/~15122/slides/review/18-heap.pdf)
- Code for this lecture (https://cs.cmu.edu/~15122/code/18-heap.tgz)

Computational Thinking: We convert ideas developed diagrammatically in the last lecture into working code.

Algorithms and Data Structures: Temporarily violating and restoring invariants is a common theme in algorithms. It is a technique you need to master.

Programming: We practice writing generic code involving arrays.

1 The Heap Structure

We use the following header struct to represent heaps.

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

The field `prior` is provided by the client and tells us how to compare elements.

```
// f(x,y) returns true if e1 has STRICTLY higher priority than e2
typedef bool has_higher_priority_fn(elem e1, elem e2);
```

Since the significant array elements start at 1, as explained in the previous lecture, the `limit` must be one greater than the desired capacity. To prevent overflows, we will also require that `limit` be less than or equal to `int_max()/2`. The `next` index must be between `1` and `limit`, and the element array must have exactly `limit` elements.

2 Minimal Heap Invariants

Before we implement the operations, we define a function that checks the heap invariants. The shape invariant is automatically satisfied due to the representation of heaps as arrays, but we need to carefully check the ordering invariants. It is crucial that no instance of the data structure that is not a true heap will leak across the interface to the client, because the client may then incorrectly call operations that require heaps with data structures that are not.

First, we check that the heap is not `NULL`, that `limit` has acceptable values, and that the length of the array matches the given `limit`. The latter must be checked in an annotation, because, in C0 and C1, the length of an array is not available to us at runtime except in contracts. Second, we check that `next` is in range, between 1 and `limit`. Finally, we check that the client-provided comparison is defined and non-`NULL`.

```
1  bool is_heap_safe(heap* H) {
2    return H != NULL
3        && (1 < H->limit && H->limit <= int_max()/2)
4        && is_array_expected_length(H->data, H->limit)
5        && (1 <= H->next && H->next <= H->limit)
6        && H->prior != NULL;
7  }
```

This is not sufficient to know that we have a valid heap! The specification function `is_heap_safe` is the minimal specification function we need to be able to access the data structure; we want to make sure anything we pass to the user additionally satisfies the ordering invariant.

This invariant acts as the precondition of some of our helper functions. We first use the client's function, accessible as `H->prior`, to express a more useful concept for our implementation: that the element in index `i` can be correctly placed as the parent of the element in index `j` in the heap.

```
9  bool ok_above(heap* H, int i1, int i2)
10 //@requires is_heap_safe(H);
11 //@requires 1 <= i1 && i1 < H->next;
12 //@requires 1 <= i2 && i2 < H->next;
13 {
14   elem e1 = H->data[i1];
15   elem e2 = H->data[i2];
16   return !(*H->prior)(e2, e1);
17 }
```

This function returns `true` whenever the element `e1` at position `i1` has priority higher than or equal to the element `e2` at position `i2`.

A second helper function that uses `is_heap_safe` swaps an element with its parent:

```
17 void swap_up(heap* H, int child)
18 //@requires is_heap_safe(H);
19 //@requires 2 <= child && child < H->next;
20 //@requires !ok_above(H, child/2, child);  // parent == child/2
21 //@ensures ok_above(H, child/2, child);
22 {
23   int parent = child/2;
24   elem tmp        = H->data[child];
25   H->data[child]  = H->data[parent];
26   H->data[parent] = tmp;
```

```
27 }
```

3 The Heap Ordering Invariant

It turns out to be simpler to specify the ordering invariant in the second form seen in the last lecture, which stipulates that each node except the root needs to be greater or equal to its parent. To check this we iterate through the array and compare the priority of each node `data[i]` with its parent, except for the root ($i = 1$) which has no parent.

```
29 bool is_heap_ordered(heap* H)
30 //@requires is_heap_safe(H);
31 {
32   for (int child = 2; child < H->next; child++)
33   //@loop_invariant 2 <= child;
34   {
35     int parent = child/2;
36     if (!ok_above(H, parent, child)) return false;
37   }
38
39   return true;
40 }
41
42 bool is_heap(heap* H) {
43   return is_heap_safe(H) && is_heap_ordered(H);
44 }
```

Observe that the loop starts at index 2, since the root of the heap is stored at index 1.

4 Creating Heaps

We start with the simple code to test if a priority queue implemented as a heap is empty or full, and to allocate a new (empty) heap. A heap is empty if the next element to be inserted would be at index 1. A heap is full if the next element to be inserted would be at index `limit` (the size of the array).

```
1 bool pq_empty(heap* H)
2 //@requires is_heap(H);
3 {
```

```
4   return H->next == 1;
5 }
6
7 bool pq_full(heap* H)
8 //@requires is_heap(H);
9 {
10  return H->next == H->limit;
11 }
```

To create a new heap, we allocate a struct and an array and set all the right initial values.

```
13 heap* pq_new(int capacity, has_higher_priority_fn* prior)
14 //@requires 0 < capacity && capacity <= int_max()/2 - 1;
15 //@requires prior != NULL;
16 //@ensures is_heap(\result) && pq_empty(\result);
17 {
18   heap* H = alloc(heap);
19   H->limit = capacity+1;
20   H->next = 1;
21   H->data = alloc_array(elem, H->limit);
22   H->prior = prior;
23   return H;
24 }
```

Note that `H->data[0]` is unused. We could have allocated an array of exactly `capacity` at the cost of complicating our index operations. The precondition `capacity <= int_max()/2 - 1` wards against overflows.

5 Insert and Sifting Up

The shape invariant tells us exactly where to insert the new element: at the index `H->next` in the data array. Then we increment the `next` index.

```
26 void pq_add(heap* H, elem x)
27 //@requires is_heap(H) && !pq_full(H);
28 //@ensures is_heap(H);
29 {
30   H->data[H->next] = x;
31   (H->next)++;           // basic invariants hold
32   // but ordering invariant may be violated
33   // ...
```

By inserting x in its specified place, we have, of course, violated the ordering invariant. We need to *sift up* the new element until we have restored the invariant. The invariant is restored when the new element is bigger than or equal to its parent or when we have reached the root. We still need to sift up when the new element is less than its parent. This suggests the following code:

```
33  int i = H->next - 1;      // element we just added
34  while (i > 1 && !ok_above(H,i/2,i)) {
35    swap_up(H, i);
36    i = i/2;
37  }
```

Setting `i = i/2` is moving up in the tree, to the place we just swapped the new element to.

At this point, as always, we should ask why accesses to the elements of the priority queue are safe. By short-circuiting of conjunction, we know that $i > 1$ when we ask whether `H->data[i/2]` is okay above `H->data[i]`. But we need a loop invariant to make sure that it respects the upper bound. The index `i` starts at `H->next - 1`, so it should always be strictly less that `H->next`.

```
34  while (i > 1 && !ok_above(H,i/2,i))
35  //@loop_invariant 1 <= i && i < H->next;
36  {
37    swap_up(H, i);
38    i = i/2;
39  }
```

One small point regarding the loop invariant: we just incremented `H->next`, so it must be strictly greater than 1 and therefore the invariant $1 \leq i$ must be satisfied.

But how do we know that swapping the element up the tree restores the ordering invariant? We need an additional loop invariant which states that H is a valid heap *except at index* i. Index i may be smaller than its parent, but it still needs to be less than or equal to its children. We therefore postulate a function `is_heap_except_up` and use it as a loop invariant.

```
34  while (i > 1 && !ok_above(H,i/2,i))
35  //@loop_invariant 1 <= i && i < H->next;
36  //@loop_invariant is_heap_except_up(H, i);
```

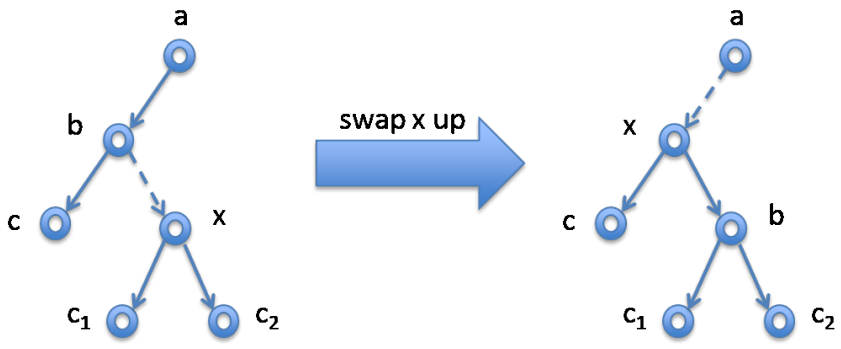
The next step is to write this function. We copy the `is_heap_ordered` function, but check a node against its parent only when it is different from the

distinguished element where the exception is allowed. The differences are highlighted.

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

We observe that `is_heap_except_up(H, 1)` is equivalent to `is_heap(H)`. That's because the loop over *child* starts at 2, so the exception $child \neq i$ is always true.

Now we try to prove that this is indeed a loop invariant, and therefore our function is correct. Rather than using a lot of text, we verify this properties on general diagrams. Other versions of this diagram are entirely symmetric. On the left is the relevant part of the heap before the swap and on the right is the relevant part of the heap after the swap. The relevant nodes in the tree are labeled with their priority. Nodes that may be above a or below c, c_1, c_2 and to the right of a are not shown. These do not enter into the invariant discussion, since their relations between each other and the shown nodes remain fixed. Also, if x is in the last row the constraints regarding c_1 and c_2 are vacuous.



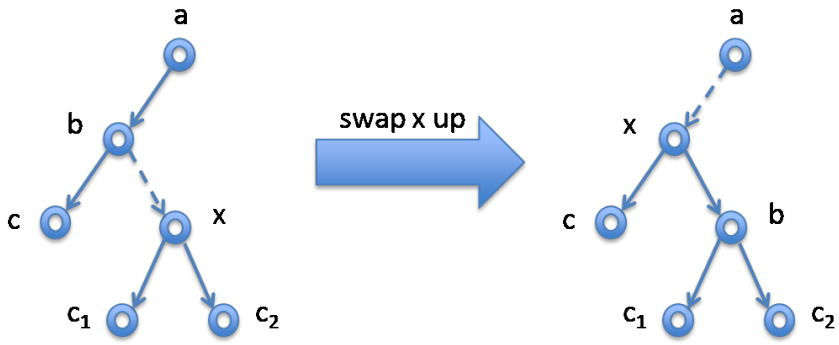
We know the following properties on the left from which the properties shown on the right follow as shown:

Current iteration			Next iteration	
$a \leq b$	(1)	order	$a \; ? \; x$	allowed exception
$b \leq c$	(2)	order	$x \leq c$	from (5) and (2)
$x \leq c_1$	(3)	order	$x \leq b$	from (5)
$x \leq c_2$	(4)	order	$b \leq c_1$	??
$b > x$	(5)	since we swap	$b \leq c_2$	??

(For this and similar examples, we'll assume that we're using a min-heap.) Our invariant gives us no way to know that $b \leq c_1$ and $b \leq c_2$. We see that simply stipulating the (temporary) invariant that every node is greater or equal to its parent except for the one labeled x is not strong enough. It is not necessarily preserved by a swap.

But we can strengthen it a bit. You might want to think about how before you move on to the next page.

The strengthened invariant also requires that the children of the potentially violating node x are greater or equal to their grandparent! Let's reconsider the diagrams.



We have more assumptions on the left now ((6) and (7)), but we have also two additional proof obligations on the right ($a \leq c$ and $a \leq b$).

Current iteration			Next iteration	
$a \leq b$	(1)	order	$a \; ? \; x$	allowed exception
$b \leq c$	(2)	order	$a \leq c$	from (1) and (2)
$x \leq c_1$	(3)	order	$a \leq b$	(1)
$x \leq c_2$	(4)	order	$x \leq c$	from (5) and (2)
$b > x$	(5)	since we swap	$x \leq b$	from (5)
$b \leq c_1$	(6)	grandparent	$b \leq c_1$	(6)
$b \leq c_2$	(7)	grandparent	$b \leq c_2$	(7)

Success! We just need an additional function that checks this loop invariant:

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

  if (i == 1)          return true;        // Reached the root
  if (left >= H->next) return true;        // No children
  if (right == H->next)                    // Left child only
```

```
    return ok_above(H, grandparent, left);
  //@assert right < H->next;              // Both children
  return ok_above(H, grandparent, left)
      && ok_above(H, grandparent, right);
}
```

Using this additional invariant, we have a loop that provably restores the `is_heap` invariant.

```
34  while (i > 1 && !ok_above(H,i/2,i))
35  //@loop_invariant 1 <= i && i < H->next;
36  //@loop_invariant is_heap_except_up (H, i);
37  //@loop_invariant grandparent_check(H, i);
38  {
39    swap_up(H, i);
40    i = i/2;
41  }
```

Note that the strengthened loop invariants (or, rather, the strengthened definition of what it means to be a heap except in one place) is not necessary to show that the postcondition of `pq_add` (i.e., `is_heap(H)`) is implied.

Postcondition: If the loop exits, we know the loop invariants and the negated loop guard are `true`:

$1 \leq i < next$	(LI 1)
`is_heap_except_up(H, i)`	(LI 2)
Either $i \leq 1$ or `ok_above(H, i/2, i)`	Negated loop guard

We distinguish the two cases.

Case: `i <= 1`. Then $i = 1$ from (LI 1), and `is_heap_except_up(H, 1)`. As observed before, that is equivalent to `is_heap(H)`.

Case: `ok_above(H, i/2, i)`. Then the only possible index i where `is_heap_except_up(H, i)` makes an exception and does not check whether `ok_above(H, i/2, i)` is actually no exception, and we have `is_heap(H)`.

Overall, the function `pq_add` is as follows

```
26 void pq_add(heap* H, elem e)
27 //@requires is_heap(H) && !pq_full(H);
28 //@ensures is_heap(H);
29 {
30   H->data[H->next] = e;
31   (H->next)++;
32   /* H may no longer be a heap! */
33
34   int i = H->next - 1;
35   while(i > 1 && !ok_above(H,i/2,i))
36   //@loop_invariant 1 <= i && i < H->next;
37   //@loop_invariant is_heap_except_up(H, i);
38   //@loop_invariant grandparent_check(H, i);
39   {
40     swap_up(H, i);
41     i = i/2;
42   }
43 }
```

6 Deleting the Minimum and Sifting Down

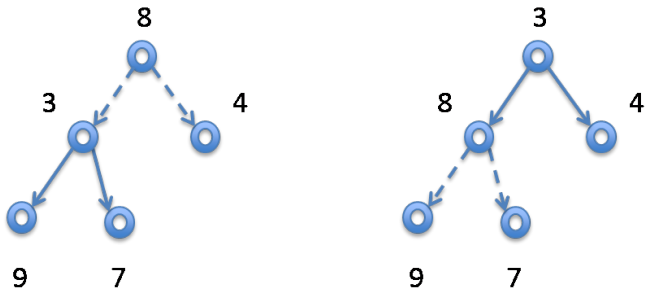
Recall that deleting the minimum swaps the root with the last element in the current heap and then applies the *sifting down* operation to restore the invariant. As with insert, the operation itself is rather straightforward, although there are a few subtleties. First, we have to check that H is a heap, and that it is not empty. Then we save the minimal element, swap it with the last element (at `next-1`), and delete the last element (now the element that was previously at the root) from the heap by decrementing `next`.

```
80 elem pq_rem(heap* H)
81 //@requires is_heap(H) && !pq_empty(H);
82 //@ensures is_heap(H);
83 {
84   elem min = H->data[1];
85   (H->next)--;
86
87   if (H->next > 1) {
88     H->data[1] = H->data[H->next];  // Swap last element in
```

```
89      // Ordering invariant may be violated
90      sift_down(H);
91    }
92    return min;
93  }
```

Next we need to restore the heap invariant by sifting down from the root, with `sift_down(H)`. We only do this if there is at least one element left in the heap.

But what is the precondition for the sifting down operation? Again, we cannot express this using the functions we have already written. Instead, we need a function `is_heap_except_down(H, i)` which verifies that the heap invariant is satisfied in H, except possibly at i. This time, though, it is between i and its children where things may go wrong, rather than between i and its parent as in `is_heap_except_up(H, i)`. In the pictures below this would be at $i = 1$ on the left and $i = 2$ on the right.



We change the test accordingly.

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

```
}
```

With this we can have the right invariant to write our `sift_down` function. The tricky part of this function is the nature of the loop. Our loop index i starts at n (which actually will always be 1 when this function is called). We have reached a leaf if $2i \geq next$ because if there is no left child, there cannot be a right one, either. So our function shapes up as follows:

```
75 void sift_down(heap* H)
76 //@requires is_heap_safe(H);
77 //@requires H->next > 1 && is_heap_except_down(H, 1);
78 //@ensures is_heap(H);
79 {
80   int i = 1;
81
82   while (2*i < H->next)
83   //@loop_invariant 1 <= i && i < H->next;
84   //@loop_invariant is_heap_except_down(H, i);
85   //@loop_invariant grandparent_check(H, i);
86   {
87     // Nothing to do: the invariant is restored already!
88     if (done_sifting_down(H, i)) return;
89
90     // Need to swap
91     int p = child_to_swap_up(H, i);
92     swap_up(H, p);
93     i = p;
94   }
95   //@assert i < H->next && 2*i >= H->next;
96 }
```

We have three loop invariants: the bounds for i, the heap invariant (everywhere, except possibly at i, looking down), and the grandparent check, which we anticipate from our previous problems.

The body of the loop first checks if we have restored the ordering invariant, and exits the function if this is the case. Otherwise it identifies the child that needs to be swapped up, swaps it with its parent, and makes the child the new current node i.

Before returning after the loop, we know that `is_heap_except_down(H,i)` and $2i \geq next$. This means there is no node j in the heap such that $j/2 = i$ and the exception in `is_heap_except_down` will never apply. H is indeed a heap.

The call `done_sifting_down(H, i)` checks if we have restored the ordering invariant. This is the case if the element in index i is okay above all of its children. However, there may be either 1 or 2 children (the loop guard checks that there will be at least one). So we have to guard this access by a bounds check. Clearly, when there is no right child, checking the left one is sufficient.

```
45 bool done_sifting_down(heap* H, int i)
46 //@requires is_heap_safe(H);
47 //@requires 1 <= i && 2*i < H->next;    // i has at least one child
48 //@requires is_heap_except_down(H, i);  // violation is at i
49 {
50   int left   = 2*i;
51   int right  = left + 1;
52
53   return ok_above(H, i, left)         // All good on the left, and
54       && (right >= H->next            // either no right child
55           || ok_above(H, i, right));  //     or all good on the right too
56 }
```

Notice the precondition of this function: the heap is safe, `i` is a valid parent node with at least one child, and the ordering invariant may be invalidate at most between `i` and one of its children.

The function `child_to_swap_up(H, i)` determines the smaller of the two children of node `i`. Observe that it has the same preconditions as `done_sifting_down`. If there is no right child, we pick the left one, of course.

```
58 int child_to_swap_up(heap* H, int i)
59 //@requires is_heap_safe(H);
60 //@requires 1 <= i && 2*i < H->next;    // i has at least one child
61 //@requires is_heap_except_down(H, i);  // violation is at i
62 //@ensures \result/2 == i;              // returns a child
63 {
64   int left   = 2*i;
65   int right  = left + 1;
66
67   if (right >= H->next ||             // if no right child, or
68       ok_above(H, left, right))       //    left child is smaller or equal
69     return left;                      // then left child will go up
70   //@assert right < H->next;          // if there is a right child, and
```

```
71   //@assert ok_above(H, right, left); //    right child is smaller or equal
72   return right;                       // then right child will go up
73 }
```

At this point we should give a proof that `is_heap_except_down` is really an invariant. This is left as Exercise 6.

7 Heapsort

We rarely discuss testing in these notes, but it is useful to consider how to write decent test cases. Mostly, we have been doing random testing, which has some drawbacks but is often a tolerable first cut at giving the code a workout. It is *much* more effective in languages that are type safe such as C0, and even more effective when we dynamically check invariants along the way.

In the example of heaps, one nice way to test the implementation is to insert a random sequence of numbers, then repeatedly remove the minimal element until the heap is empty. If we store the elements in an array in the order we take them out of the heap, the array should be sorted when the heap is empty! This is the idea behind heapsort. We first show the code, using the random number generator we have used for several lectures now, then analyze the complexity. As the priority function, we use `int_lt(x,y)` which returns `true` if and only if `x<y`. The standard loop invariants have been omitted for conciseness.

```
int main() {
  int n = (1<<9)-1;          // 1<<9 for -d; 1<<13 for timing
  int num_tests = 10;        // 10 for -d;  100 for timing
  int seed = 0xc0c0ffee;
  rand_t gen = init_rand(seed);
  int[] A = alloc_array(int, n);
  heap* H = pq_new(n, &int_lt);

  printf("Testing heap of size %d, %d times\n", n, num_tests);
  for (int j = 0; j < num_tests; j++) {
    for (int i = 0; i < n; i++) {
      pq_add(H, rand(gen));
    }
    for (int i = 0; i < n; i++) {
      A[i] = pq_rem(H);
```

```
    }
    assert(pq_empty(H));          // heap not empty
    assert(is_sorted(A, 0, n));   // heapsort failed
  }
  printf("Passed all tests!\n");
  return 0;
}
```

Now for the complexity analysis. Inserting n elements into the heap is bounded by $O(n \log n)$, since each of the n inserts is bounded by $\log n$. Then the n element deletions are also bounded by $O(n \log n)$, since each of the n deletions is bounded by $\log n$. So altogether we get $O(2n \log n) = O(n \log n)$. Heapsort is asymptotically as good as mergesort or as good as the expected complexity of quicksort with random pivots.

The sketched algorithm uses $O(n)$ auxiliary space, namely the heap. One can use the same basic idea to do heapsort in place, using the unused portion of the heap array to accumulate the sorted array.

Testing, including random testing, has many problems. In our context, one of them is that it does not test the strength of the invariants. For example, say we write no invariants whatsoever (the weakest possible form), then compiling with or without dynamic checking will always yield the same test results. We really should be testing the invariants themselves by giving examples where they are not satisfied. However, we should not be able to construct such instances of the data structure on the client side of the interface. Furthermore, within the language we have no way to "capture" an exception such as a failed assertion and continue computation.

8 Summary

We briefly summarize key points of how to deal with invariants that must be temporarily violated and then restored.

1. Make sure you have a clear high-level understanding of why invariants must be temporarily violated, and how they are restored.

2. Ensure that at the interface to the abstract type, only instances of the data structure that satisfy the full invariants are being passed. Otherwise, you should rethink all the invariants.

3. Write predicates that test whether the partial invariants hold for a data structure. Usually, these will occur in the preconditions and

loop invariants for the functions that restore the invariants. This will force you to be completely precise about the intermediate states of the data structure, which should help you a lot in writing correct code for restoring the full invariants.

9 Exercises

Exercise 1 (sample solution on page 19)**.** *Here is an example of the underlying data in a min heap. Draw what the corresponding heap looks like.*

- `size = 12`
- `next = 10`
- `data = [X, 3, 5, 4, 7, 12, 8, 6, 9, 13, X, X]`

Exercise 2 (sample solution on page 19)**.** *Is the array* `H->data` *of a heap sorted in descending priority order? In ascending priority order?*

Exercise 3 (sample solution on page 19)**.** *Most data representation invariants we have seen are written as recursive functions, but* `is_heap_ordered` *was instead given in an imperative style. Write a recursive version of* `is_heap_ordered`*.*

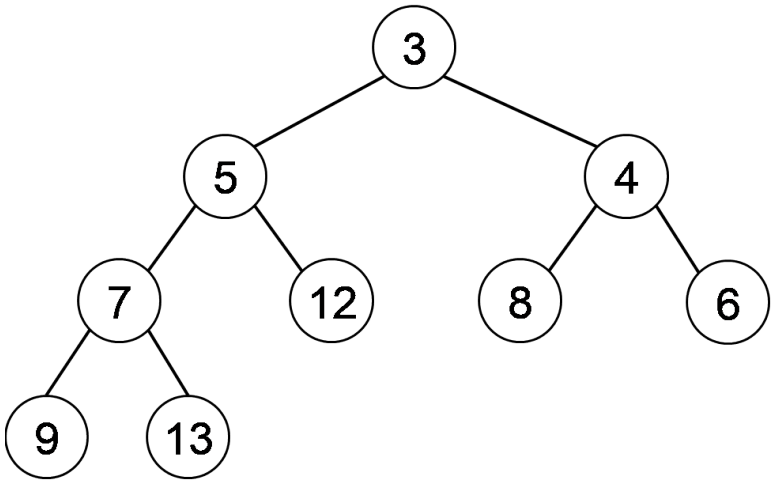
Exercise 4 (sample solution on page 21)**.** *Above, we separated out the sift down operation into its own function* `sift_down`*. Do the same for sift up.*

Exercise 5 (sample solution on page 22)**.** *We want to extend the priority queue interface so that, when inserting a new element and the queue is full, we silently delete an element with the lowest priority to make space for the new element. Describe an algorithm, analyze its asymptotic complexity, and provide its implementation as the function* `pq_bump`*.*

Exercise 6 (Sifting Down)**.** *sample solution on page 23*
] Give a diagrammatic proof for the ordering invariant properties of sifting down for delete (called `is_heap_except_down`*), along the lines of the one we gave for sifting up for insert. You may limit yourself to the case where there is a violation between a node x and its right child c but not its left child b, where x has a parent node and four grandchildren nodes. You are welcome to carry out this proof for other cases.*

Sample Solutions

Solution of exercise 1 The corresponding heap is as follows:



Solution of exercise 2 Neither. Consider the array `H->data[X, 3, 8, 4, 9]` underlying a min-heap. It represents a valid heap, but it is sorted in neither ascending or descending priority order.

The only relationships that must hold about are the priorities in the array `H->data` is that, as we follow a path deeper into the tree, the priorities we encounter never increase (the priority of a child node is lower than or equal to the the priority of its parent).

Solution of exercise 3 We split the recursive version of `is_heap_ordered` into two functions. One, still called `is_heap_ordered`, has the same prototype as the imperative version, so that it can replace it without any change in the rest of the code. The other, `is_heap_ordered_helper`, is a helper function used to recurse through the tree. There are two reasons to organize this code in this way:

- The helper function `is_heap_ordered_helper` needs to know at what node in the tree to operate, which is passed as an additional argument. The imperative version of `is_heap_ordered` didn't need this information as it started from the root.
- The root of the heap does not have a parent, meaning that we can specialize the code of our recursive `is_heap_ordered` (which operated there) to this position.

The resulting code is as follows:

```
bool is_heap_ordered_helper(heap* H, int child)
//@requires is_heap_safe(H);
//@requires child > 1;
{
  if (child >= H->next) return true;

  int parent = child/2;
  if (!ok_above(H, parent, child)) return false;
  return is_heap_ordered_helper(H, child * 2)
      && is_heap_ordered_helper(H, child * 2 + 1);
}

bool is_heap_ordered(heap* H)
//@requires is_heap_safe(H);
{
  return is_heap_ordered_helper(H, 2)
      && is_heap_ordered_helper(H, 3);
}
```

The function `is_heap_ordered` simply calls the helper function on the two children of the root of the heap, at positions 2 and 3 respectively. Note that these children may not exist.

This is taken care of by the helper function `is_heap_ordered_helper`. It first checks whether the node it is being called on exists — if it doesn't, this part of the the heap is trivially ordered. It then computes the index of the parent, whose existence is ensured by a dedicated precondition. Then, it checks if this parent is indeed OK above the input node — and returns `false` otherwise. Finally, this function calls itself on the two children of the input node.

This all sounds reasonable, but heaps have a lots of edge cases to consider. Let's look into this:

- *What if the heap contains only a root node?* In this case, `H->next == 1`. Then, both recursive calls to `is_heap_ordered_helper` will immediately return `true`, causing `is_heap_ordered` to also return `true`. But this is exactly what we expect since a heap with a single node trivially obeys the ordering invariant.

- *What if the heap is empty?* The exact same argument we just discussed also applies in this case.

For further practice, write recursive variants of the functions `is_heap_except_up` and `is_heap_except_down`.

Solution of exercise 4 The benefits of spinning off the sifting-up process within `pq_add` is minimal as it largely amounts to splitting this function in two (this is different from the more complex `pq_rem`, where splitting the code in several functions make it much more clear and modular). One minor opportunity for optimization is that `pq_add` can return immediately when when inserting in an empty heap rather than triggering the (nearly vacuous) sifting-up process.

The resulting code is as follows:

```
void sift_up(heap* H)
//@requires is_heap_safe(H);
//@requires H->next > 2 && is_heap_except_up(H, H->next - 1);
//@ensures is_heap(H);
{
  int i = H->next - 1;

  while (i > 1)
  //@loop_invariant 1 <= i && i < H->next;
  //@loop_invariant is_heap_except_up(H, i);
  //@loop_invariant grandparent_check(H, i);
  {
    // Nothing to do: the invariant is restored already!
    if (ok_above(H, i/2, i)) return;

    // Need to swap
    swap_up(H, i);
    i = i/2;
  }
}
```

```
void pq_add(heap* H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H);
{
  H->data[H->next] = e;
  (H->next)++;
  /* H may no longer be a heap! */

  if (H->next > 2)
    sift_up(H);
}
```

Solution of exercise 5 At least some of the elements with the lowest priorities are located among the leaves of the priority queue. The starting index of the leaves of the tree is `H->next/2`. The algorithm first searches through the leaves of the tree to find an element with the lowest priority. Then, it replaces this value with our new element and sift up until the heap invariants are restored, just like in the normal `pq_add`. The runtime of this is $O(n)$, because about half of the n nodes in the heap are leaves.

The function `get_minimal_element` simply goes through the leaves of the heap (from index `H->next/2` up to index `H->next` excluded) looking for an element with minimal priority. Of course, there may be several such elements. *Will this implementation return the first or the last one? This is for you to determine.*

```
int get_minimal_element(heap* H)
//@requires is_heap(H);
//@ensures H->next/2 <= \result && \result < H->next;
{
  elem min = H->data[H->next/2];
  int min_index = H->next/2;
  for (int i = H->next/2 + 1; i < H->next; i++) {
    if (ok_above(H, min, H->data[i])) {
      min = H->data[i];
      min_index = i;
    }
  }
  return min_index;
}
```

The function `pq_bump` differs from `pq_add` in just one respect: it needs

to take special actions if the heap is full. This is when it needs to identify the index of a minimal element (by calling `get_minimal_element`) and replace it with the new element before triggering the sifting-up process.

```
void pq_bump(heap* H, elem e)
//@requires is_heap(H);
//@ensures is_heap(H);
{
  int index;
  if (pq_full(H)) {
    index = get_minimal_element(H);
  }
  else {
    index = H->next;
    H->next++;
  }
  H->data[index] = e;

  /* H may no longer be a heap! */
  int i = index;
  while (i > 1 && !ok_above(H,i/2,i))
  //@loop_invariant 1 <= i && i < H->next;
  //@loop_invariant is_heap_except_up(H, i);
  //@loop_invariant grandparent_check(H, i);
  {
    swap_up(H, i);
    i = i/2;
  }
}
```

Solution of exercise 6 Recall the code for the function `sift_down`:

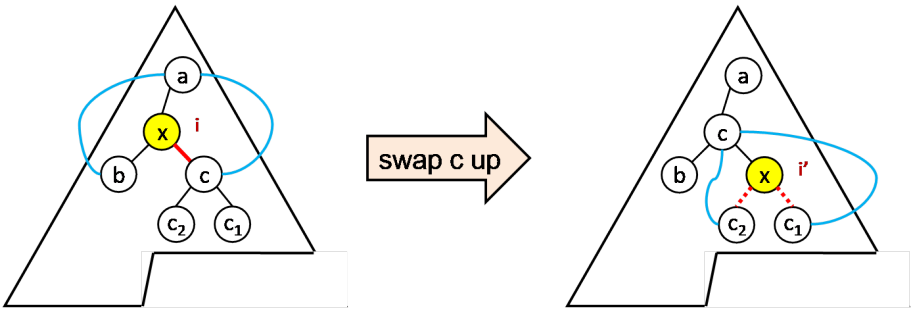
```
void sift_down(heap* H)
//@requires is_heap_safe(H);
//@requires H->next > 1 && is_heap_except_down(H, 1);
//@ensures is_heap(H);
{
  int i = 1;

  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;
}
```

The loop invariant `is_heap_except_down(H,i)` ensures that the entire heap is ordered correctly, except for a possible violation between the node x at index `i` and its children. The loop invariant `grandparent_check(H,i)` ensures that the parent of `i` is Ok above the children of (the node, x, at index) `i`. We want to show that both of these loop invariants are preserved by an arbitrary iteration of the loop.

The diagram for the requested case of the proof is as follows:



The figure on the left describes the heap at the beginning of the current

iteration for this case: x (at index `i`) has a parent a, two children b and c, four grandchildren (only the children c_1 and c_2 of c are displayed). There is a violation between x and c, indicated with a red line, but not between x and b. The blue lines denote the grandparent check at x.

Because there is a violation between x and c, but not between x and b, this iteration of the loop will swap x and c. The resulting diagram is displayed on the right. We need to prove that all the relations depicted by solid lines on the right are valid. As a result of swapping x and c, violations between x and c_1 or c_2 (or both) are allowed — we do not need to prove them.

For simplicity, let's assume the heap we are dealing with is a min-heap. Then, the relations depicted on the left are written mathematically as the following assumptions:

1. $a \leq x$ (order assumption)
2. $x \leq b$ (assumption that there is no violation between x and b)
3. $x > c$ (violation assumption)
4. $c \leq c_1$ (order assumption)
5. $c \leq c_2$ (order assumption)
6. $a \leq b$ (grandparent check assumption)
7. $a \leq c$ (grandparent check assumption)

Given these assumptions, we prove that the relations on the right picture are valid as follows:

i. $a \leq c$ by *(7)*

ii. $c \leq b$ by *(3)* and *(2)*

iii. $c \leq x$ by *(3)*

iv. $x \mathbin{?} c_1$ (allowed exception)

v. $x \mathbin{?} c_2$ (allowed exception)

vi. $c \leq c_1$ by *(4)*

vii. $c \leq c_2$ by *(5)*

Note that the proof would be almost identical had there been a violation between x and both of its children b and c, but $c \leq b$.