Lecture Notes on Restoring Invariants

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 16
October 22, 2013

1 Introduction

In this lecture we will implement operations on heaps. 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 Lecture 15 on priority queues and keep them close at hand.

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

2 The Heap Structure

We use the following header struct to represent heaps.

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

Since the significant array elements start at 1, as explained in the previous lecture, the *limit* must be one greater than the desired capacity. The *next* index must be between 1 and *limit*, and the element array must have exactly *limit* elements.

3 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 and that the length of the array matches the given `limit`. The latter must be checked in an annotation, because, in C and C0, 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*. As a general stylistic choice, when writing functions that check data structure invariants and have to return a boolean, we think of the tests like assertions. If they would *fail*, we return `false` instead. Therefore we usually write negated conditionals and return `false` if the negated condition is true. In the code below, we think

```
//@assert H != NULL;
//@assert 1 <= H->next && H->next <= H->limit;
//@assert \length(H->heap) == H->limit;
```

and write

```
bool is_safe_heap(heap H) {
  if (!(H != NULL)) return false;
  if (!(1 <= H->next && H->next <= H->limit)) return false;
  //@assert \length(H->heap) == H->limit;
  return true;
}
```

This is not sufficient to know that we have a valid heap! The specification function `is_safe_heap` 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.

4 The Heap Ordering Invariant

It turns out to be simpler to specify the ordering invariant in the second form, 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. As a matter of programming style, we always put the parent to left in any comparison, to make it easy to see that we are comparing the correct elements. We also write `struct heap_header* H` for the argument to emphasize that the argument H is not necessarily a heap.

```
bool is_heap(struct heap_header* H) {
  if (!is_safe_heap(H)) return false;
  /* check parent <= node for all nodes except root (i = 1) */
  for (int i = 2; i < H->next; i++)
    //@loop_invariant 2 <= i;
    if (!(H->data[i/2] <= H->data[i])) return false;
  return true;
}
```

The test in the loop is not quite right, but lets just verify that it is at least *safe*

- We can dereference `H->data` because we have checked that H is not null.
- We can access `H->data` at i, because (by loop invariant) $i \geq 2 \geq 1$ and (by the loop guard), $i < H{\rightarrow}next$. The latter implies safety since $H{\rightarrow}next \leq H{\rightarrow}limit = \mathsf{length}(H{\rightarrow}data)$.
- We can access `H->data` at $i/2$, because $i/2 \geq 1$ since $i \geq 2$ (by loop invariant) and $i/2 < i < \mathsf{length}(H{\rightarrow}next)$.

Why is it incorrect? Recall that in our interface we specified heaps to contain data of type `elem`, and that no assumption should be made about this type except that the client provides a function `elem_priority`. So we need to extract the priority from the data element.

```
if (!(elem_priority(H->data[i/2]) <= elem_priority(H->data[i])))
  return false;
```

We commonly need to access the priority of data stored in the heap, so we separate this out as a function. The only tricky aspect of this function is its contract. We cannot require the argument to be a heap, since in the `is_heap` function we don't know this yet! It would also make `is_heap` and the `priority` function mutually recursive, leading to nontermination. But we need to say enough so that access to the heap array is safe.

```
int priority(struct heap_header* H, int i)
//@requires H != NULL;
//@requires 1 <= i && i < H->next;
//@requires H->next <= \length(H->data);
{
  return elem_priority(H->data[i]);
}
```

The middle line is a little stronger than we need for safety, but it is important that we never access an element that is meaningless, like the one stored at index 0, and the ones stored at $H \rightarrow next$ and beyond. Then the final version of our `is_heap` function is:

```
bool is_heap(struct heap_header* H) {
  if (!is_safe_heap(H)) return false;
  for (int i = 2; i < H->next; i++)
    //@loop_invariant 2 <= i;
    if (!(priority(H, i/2) <= priority(H, i))) return false;
  return true;
}
```

5 Creating Heaps

We start with the simple code to test if 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).

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

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

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

```
heap pq_new(int capacity)
//@requires capacity > 0;
//@ensures is_heap(\result) && pq_empty(\result);
{
  heap H = alloc(struct heap_header);
  H->limit = capacity+1;
  H->next = 1;
  H->data = alloc_array(elem, capacity+1);
  return H;
}
```

6 Insert and Sifting Up

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

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

By inserting e 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:

```
  int i = H->next - 1;
  while (i > 1 && priority(H,i) < priority(H,i/2))
    {
      swap(H->data, i, i/2);
      i = i/2;
    }
```

Here, `swap` is the standard function, swapping two elements of the array. Setting `i = i/2` is moving up in the array, 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 $priority(H, i) < priority(H, i/2)$. But we need a loop invariant to make sure that it respects the upper bound. The index i starts at $H{\rightarrow}next - 1$, so it should always be strictly less that $H{\rightarrow}next$.

```
int i = H->next - 1;
while (i > 1 && priority(H,i) < priority(H,i/2))
  //@loop_invariant 1 <= i && i < H->next;
  {
    swap(H->data, i, i/2);
    i = i/2;
  }
```

One small point regarding the loop invariant: we just incremented $H{\rightarrow}next$, 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 or equal to its children. We therefore postulate a function `is_heap_expect_up` and use it as a loop invariant.

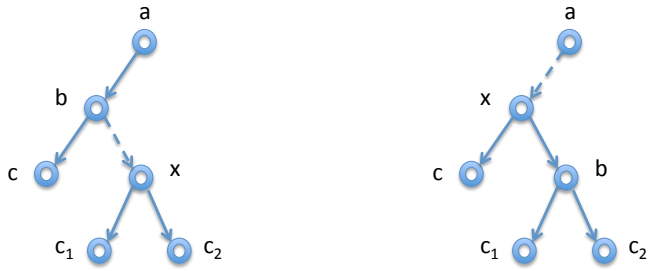
```
int i = H->next - 1;
while (i > 1 && priority(H,i) < priority(H,i/2))
  //@loop_invariant 1 <= i && i < H->next;
  //@loop_invariant is_heap_except_up(H, i);
  {
    swap(H->data, i, i/2);
    i = i/2;
  }
```

The next step is to write this function. We copy the `is_heap` function, but check a node against its parent only when it is different from the distinguished element where the exception is allowed.

```
bool is_heap_except_up(heap H, int n) {
  if (!is_safe_heap(H)) return false;
  for (int i = 2; i < H->next; i++)
    //@loop_invariant 2 <= i;
    {
      if (i != n && !(priority(H, i/2) <= priority(H, i)))
        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 i starts at 2, so the exception $i \neq n$ 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:

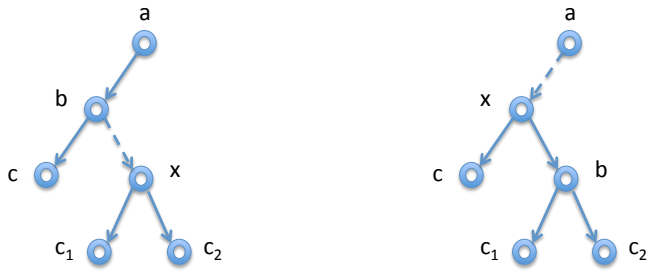
$a \leq b$	(1)	order
$b \leq c$	(2)	order
$x \leq c_1$	(3)	order
$x \leq c_2$	(4)	order
$x < b$	(5)	since we swap

$a \mathbin{?} x$	allowed exception
$x \leq c$	from (5) and (2)
$x \leq b$	from (5)
$b \leq c_1$	??
$b \leq c_2$	??

So 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 $a \leq b$).

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

$a \ ? \ x$	allowed exception
$a \leq c$	from (1) and (2)
$a \leq b$	(1)
$x \leq c$	from (5) and (2)
$x \leq b$	from (5)
$b \leq c_1$	(6)
$b \leq c_2$	(7)

Success! We just need to update the code for `is_heap_except_up` to check this additional property.

```
bool is_heap_except_up(heap H, int n) {
  if (!is_safe_heap(H)) return false;
  for (int i = 2; i < H->next; i++)
    //@loop_invariant 2 <= i;
    {
      if (i != n && !(priority(H, i/2) <= priority(H, i)))
        return false;
      /* for children of node n, check grandparent  */
      if (i/2 == n && (i/2)/2 >= 1
          && !(priority(H, (i/2)/2) <= priority(H,i)))
        return false;
    }
  return true;
}
```

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

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

$1 \leq i < next$ (LI 1)
$is\_heap\_except\_up(H, i)$ (LI 2)
Either $i \leq 1$ or $priority(H, i) \geq priority(H, i/2)$ Negated loop guard

We distinguish the two cases.

Case: $i \leq 1$. Then $i = 1$ from (LI 1), and $is\_heap\_except\_up(H, 1)$. As observed before, that is equivalent to $is\_heap(H)$.

Case: $priority(H, i) \geq priority(H, i/2)$. Then the only possible index i where $is\_heap\_except\_up(H, i)$ makes an exception and does not check whether $priority(H, i/2) \leq priority(H, i)$ is actually no exception, and we have $is\_heap(H)$.

7 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.

Exercises

Exercise 1 *Write a recursive version of* `is_heap`.

Exercise 2 *Write a recursive version of* `is_heap_except_up`.

Exercise 3 *Write a recursive version of* `is_heap_except_down`.

Exercise 4 *Give a diagrammatical proof for the invariant property of sifting down for delete (called* `is_heap_except_down`*), along the lines of the one we gave for sifting up for insert.*

Exercise 5 *Say we want to extend priority queues so that when inserting a new element and the queue is full, we silently delete the element with the lowest priority (= maximal key value) before adding the new element. Describe an algorithm, analyze its asymptotic complexity, and provide its implementation.*

Exercise 6 *Using the invariants described in this lecture, write a function* `heapsort` *which sorts a given array in place by first constructing a heap, element by element, within the same array and then deconstructing the heap, element by element.*
*[**Hint:** It may be easier to sort the array in descending order and reverse in a last pass or use so called max heaps where the maximal element is at the top]*

Exercise 7 *Is the array* `H->data` *of a heap always sorted?*