Lecture Notes on
Restoring Invariants

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 17
October 23, 2012

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 16 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 The Heap Ordering Invariant

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

and write

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

The remaining code has to check the ordering invariant. It turns out to be simpler 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 (!(H != NULL)) return false;
  //@assert \length(H->heap) == H->limit;
  if (!(1 <= H->next && H->next <= H->limit)) 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 (!(H != NULL)) return false;
  //@assert \length(H->data) == H->limit;
  if (!(1 <= H->next && H->next <= H->limit)) 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;
}
```

4 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;
}
```

5 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 $\mathit{priority}(H, i) < \mathit{priority}(H, i/2)$. But we need a loop invariant to make sure that it respects the upper bound. The index i starts at $H{\rightarrow}\mathit{next} - 1$, so it should always be strictly less that $H{\rightarrow}\mathit{next}$.

```
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 $H{\rightarrow}\mathit{next}$, 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.

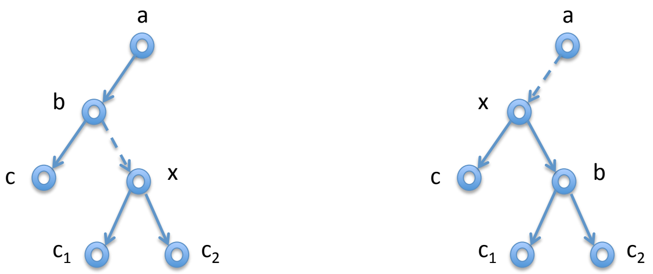
```
int i = H->next - 1;
while (i > 1 && priority(H,i) < priority(H,i/2))
  //@loop_invariant 1 <= i && i < H->next;
  //@loop_invariant is_heap_except_up(H, i);
  {
    swap(H->data, i, i/2);
    i = i/2;
  }
```

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 (H == NULL) return false;
  //@assert \length(H->data) == H->limit;
  if (!(1 <= H->next && H->next <= H->limit)) 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 $\mathit{is\_heap\_except\_up}(H, 1)$ is equivalent to $\mathit{is\_heap}(H)$. 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:

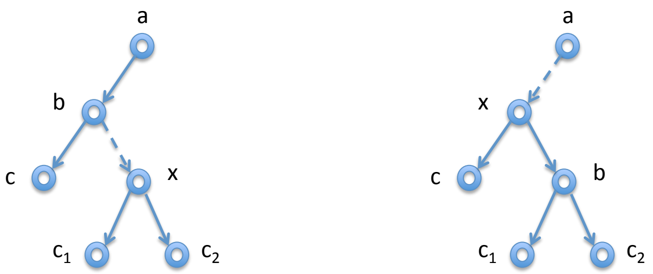
$a \leq b$	(1)	order
$b \leq c$	(2)	order
$x \leq c_1$	(3)	order
$x \leq c_2$	(4)	order
$x < b$	(5)	since we swap

$a \ ? \ x$	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 (H == NULL) return false;
  //@assert \length(H->data) == H->limit;
  if (!(1 <= H->next && H->next <= H->limit)) 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:

$1 \leq i < \mathit{next}$ (LI 1)

$\mathit{is\_heap\_except\_up}(H, i)$ (LI 2)

Either $i \leq 1$ or $\mathit{priority}(H, i) \geq \mathit{priority}(H, i/2)$ Negated loop guard

We distinguish the two cases.

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

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

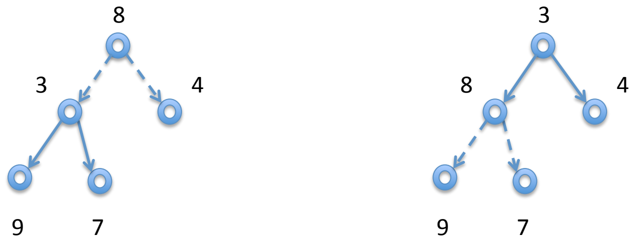
6 Deleting the Minimum and Sifting Down

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 $\mathit{next} - 1$), and delete the last element (now the element that was previously at the root) from the heap by decrementing next.

```
elem pq_delmin(heap H)
//@requires is_heap(H) && !pq_empty(H);
//@ensures is_heap(H);
{
  int n = H->next;
  elem min = H->data[1];
  H->data[1] = H->data[n-1];
  H->next = n-1;
  if (H->next > 1) sift_down(H, 1);
  return min;
}
```

Next we need to restore the heap invariant by sifting down from the root, with `sift_down(H, 1)`. 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, n)` which verifies that the heap invariant is satisfied in H, expect possibly at n. This time, though, it is between n and its children where things may go wrong, rather than between n and its parent as in `is_heap_except_up(H, n)`. In the pictures below this would be at $n = 1$ on the left and $n = 2$ on the right.



We change the test accordingly. Anticipating the earlier problem, we again require that the children of the exceptional node are less than their grandparent.

```
bool is_heap_except_down(heap H, int n) {
  if (H == NULL) return false;
  //@assert \length(H->data) == H->limit;
  if (!(1 <= H->next && H->next <= H->limit)) return false;
  /* check parent <= node for all nodes except root (i = 1) */
  /* and children of n (i/2 = n) */
  for (int i = 2; i < H->next; i++)
    //@loop_invariant 2 <= i;
    {
      if (i/2 != 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;
}
```

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 $2 * i \geq \mathit{next}$ because if there is no left child, there cannot be a right one, either. So the outline of our function shapes up as follows:

```
void sift_down(heap H, int i)
//@requires 1 <= i && i < H->next;
//@requires is_heap_except_down(H, i);
//@ensures is_heap(H);
{ int n = H->next;
  int left = 2*i;
  int right = left+1;
  while (left < n)
    //@loop_invariant 1 <= i && i < n;
    //@loop_invariant left == 2*i && right == 2*i+1;
    //@loop_invariant is_heap_except_down(H, i);
  ...
}
```

We also have written down three loop invariants: the bounds for i, the heap invariant (everywhere, except possibly at i, looking down), and the invariant defining local variables *left* and *right*, standing for the left and right children of i.

We want to return from the function if we have restored the invariant, that is $\mathit{priority}(H, i) \leq \mathit{priority}(H, 2 * i)$ and $\mathit{priority}(H, i) \leq \mathit{priority}(H, 2 * i + 1)$. However, the latter reference might be out of bounds, namely if we found a node that has a left child but not a right child. So we have to guard this access by a bounds check. Clearly, when there is no right child, checking the left one is sufficient.

```
  while (left < n)
    //@loop_invariant 1 <= i && i < n;
    //@loop_invariant left == 2*i && right == 2*i+1;
    //@loop_invariant is_heap_except_down(H, i);
    { if (priority(H,i) <= priority(H,left)
          && (right >= n || priority(H,i) <= priority(H,right)))
```

```
        return;
      ...
    }
```

If this test fails, we have to determine the smaller of the two children. If there is no right child, we pick the left one, of course. Once we have found the smaller one we swap the current one with the smaller one, and then make the child the new current node i.

```
void sift_down(heap H, int i)
//@requires 1 <= i && i < H->next;
//@requires is_heap_except_down(H, i);
//@ensures is_heap(H);
{ int n = H->next;
  int left = 2*i;
  int right = left+1;
  while (left < n)
    //@loop_invariant 1 <= i && i < n;
    //@loop_invariant left == 2*i && right == 2*i+1;
    //@loop_invariant is_heap_except_down(H, i);
    { if (priority(H,i) <= priority(H,left)
           && (right >= n || priority(H,i) <= priority(H,right)))
        return;
      if (right >= n || priority(H,left) < priority(H,right)) {
        swap(H->data, i, left);
        i = left;
      } else {
        //@assert right < n && priority(H, right) <= priority(H,left);
        swap(H->data, i, right);
        i = right;
      }
      left = 2*i;
      right = left+1;
    }
  //@assert i < n && 2*i >= n;
  //@assert is_heap_except_down(H, i);
  return;
}
```

Before the second return, we know that `is_heap_except_down(H,i)` and $2 * i \geq next$. 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.

At this point we should give a proof that is_heap_except_down is really an invariant. This is left as Exercise 4.

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.

```
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);

  print("Testing heap of size "); printint(n);
  print(" "); printint(num_tests); print(" times\n");
  for (int j = 0; j < num_tests; j++) {
    for (int i = 0; i < n; i++) {
      pq_insert(H, rand(gen));
    }
    for (int i = 0; i < n; i++) {
      A[i] = pq_delmin(H);
    }
    assert(pq_empty(H)); /* heap not empty */
    assert(is_sorted(A, 0, n)); /* heapsort failed */
  }
  print("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 all together we get $O(2 * n * \log(n)) = O(n * \log(n))$. Heapsort is asymptotically as good as mergesort (Lecture 7) or as good as the expected complexity of quicksort with random pivots (Lecture 8).

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.

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?*