Lecture Notes on Restoring Invariants

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 14
October 12, 2010

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 13: 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 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 in a precondition that the heap is not null and that the length of the array matches the given `limit`. The latter must be checked in a precondition, because in C and C0, the length of an array is not available to us at runtime.

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 test 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

```
bool is_heap(heap H)
//@requires H != NULL && \length(H->heap) == H->limit;
{ int i;
  assert(1 <= H->next && H->next <= H->limit, "'next' out of bounds");
  ...
  return true;
}
```

and write

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

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. For this we start at index 2, iterating through the array and comparing each node $A[i]$ with its 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.

```
bool is_heap(heap H)
//@requires H != NULL && \length(H->heap) == H->limit;
{ int i;
  if (!(1 <= H->next && H->next <= H->limit)) return false;
  for (i = 2; i < H->next; i++)
    if (!(H->heap[i/2] <= H->heap[i])) return false;
  return true;
}
```

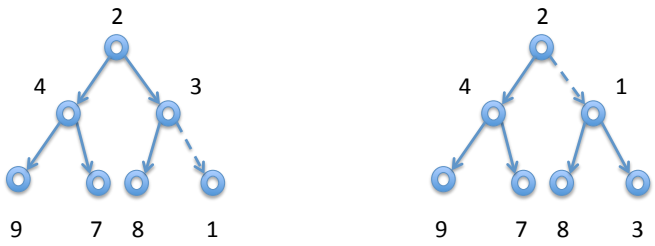
3 Insert and Sifting Up

Insertion is very short; we only need to make sure the heap is not full. Since this is something quick to check and might be overlooked by the client, we use an explicit assert statement which is always executed, even if full dynamic checking of invariants is not enabled. After that, we place the new element at `next`, increment the `next` index, and then call `sift_up` in order to restore the invariant.

```
void sift_up(heap H, int n);

void heap_insert(heap H, int x)
//@requires is_heap(H);
//@requires !heap_full(H);
//@ensures is_heap(H);
{
  assert(!heap_full(H), "cannot insert into full heap");
  H->heap[H->next] = x;
  H->next++;
  sift_up(H, H->next-1);
}
```

The difficult part is, of course, the definition of `sift_up`. But we cannot write this yet, because we have no way to express a precondition. Clearly, H is not a valid heap, because the ordering invariant might be violated at n, the index of the new element we inserted. In the left tree in the picture below, this would be at $n = 7$ where the new element with key 1 is stored.



After one step of sifting up we obtain the tree on the right, where the invariant is now violated at index $n = 3$.

So we need a new function that checks if a given argument of type `heap` is *almost* a heap, but where the parent of the given node n might be smaller

than its parent. We call this `is_heap_except_up`. It is literally the same as `is_heap`, except if the index i is equal to n we do not compare with its parent.

```
bool is_heap_except_up(heap H, int n)
//@requires H != NULL && \length(H->heap) == H->limit;
{ int i;
  if (!(1 <= H->next && H->next <= H->limit)) return false;
  for (i = 2; i < H->next; i++)
    if (!(i == n || H->heap[i/2] <= H->heap[i])) return false; // (*)
  return true;
}
```

We can think of the crucial line $(*)$ as

```
//@assert i == n || H->heap[i/2] <= H->heap[i];
```

rewritten as a conditional returning false.

Now the code for `sift_up` almost writes itself. We start with a heap except at n and modify it to be a true heap.

```
void sift_up(heap H, int n)
//@requires 1 <= n && n < H->limit;
//@requires is_heap_except_up(H, n);
//@ensures is_heap(H);
{ ... }
```

The body of the function consists of a loop that walks up the tree, swapping the new element with its parent or directly returning from the function if the invariant has been restored before we reach the root. We reach the root when the loop index i has become 1, which means we remain in the loop as long as $i > 1$.

```
void sift_up(heap H, int n)
//@requires 1 <= n && n < H->limit;
//@requires is_heap_except_up(H, n);
//@ensures is_heap(H);
{ int i = n;
  while (i > 1)
    //@loop_invariant is_heap_except_up(H, i);
    {
      if (H->heap[i/2] <= H->heap[i]) return;
```

```
      swap(H->heap, i/2, i);  /* swap i with parent */
      i = i/2;                /* consider parent next */
    }
  //@assert i == 1;
  //@assert is_heap_except_up(H, 1);
  return;
}
```

Note how `is_heap_except_up(H, i)` serves as a loop invariant. At any stage in the computation, we have a valid heap *except* at the node we have inserted and are sifting upward in the tree.

It is clear that if we return from the middle of the loop, then `is_heap` must be satisfied, since it is satisfied everywhere except at i (by loop invariant) and it is satisfied at i when the condition is true. So we can return.

When we exit the loop and we have reached the root, we note that H is a heap except at the root. But the root has no parent, so the heap invariant must be true everywhere. Looking at the code for `is_heap_except_up` we can see that the exception $i = n$ can never be true when $n = 1$ because the loop index starts at 2.

The postcondition is therefore established before both `return` statements.

4 Deleting the Minimum and Sifting Down

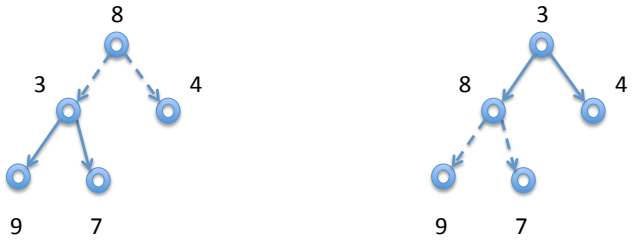
Recall that deleting the minimum swaps the root with the last element in the current heap and then applies the *sifting down* operation to restore the invariant. As with insert, the operation itself is rather straightforward, although there are a few subtleties. First, we have to check that H is a heap, 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 $next$.

```
int heap_delmin(heap H)
//@requires is_heap(H);
//@requires !heap_empty(H);
//@ensures is_heap(H);
{ assert(!heap_empty(H), "cannot delete from empty heap");
  { int x = H->heap[1];
    swap(H->heap, 1, H->next-1);
    H->next--;
    // next is the line we got wrong: H is not necessarily a heap
```

```
    // because invariant might be violated at the root, and
    // heap_empty requires H to be a heap
    //  if (!heap_empty(H)) sift_down(H, 1);
    if (H->next > 1) sift_down(H, 1);
    return x;
  }
}
```

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.

```
bool is_heap_except_down(heap H, int n)
//@requires H != NULL && \length(H->heap) == H->limit;
{ int i;
  if (!(1 <= H->next && H->next <= H->limit)) return false;
  for (i = 2; i < H->next; i++)
    if (!(i/2 == n || H->heap[i/2] <= H->heap[i])) return false; // (*)
  return true;
}
```

In the crucial line marked $(*)$, the heap invariant must be satisfied *except* if the parent $i/2$ of the current node i is equal to n.

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 $2 * i \geq next$ 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 n)
//@requires 1 <= n && n < H->next;
//@requires is_heap_except_down(H, n);
//@ensures is_heap(H);
{ int i = n;
  while (2*i < H->next)
    //@loop_invariant is_heap_except_down(H, i);
    { ... }
}
```

We also have written down the loop invariant: the heap invariant is satisfied everywhere, except possibly at i, looking down.

We want to return from the function if we have restored the invariant, that is $heap[i] \leq heap[2 * i]$ and $heap[i] \leq heap[2 * i + 1]$. 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 (2*i < H->next)
    //@loop_invariant is_heap_except_down(H, i);
    {
      if (H->heap[i] <= H->heap[2*i] &&
          (2*i+1 >= H->next || H->heap[i] <= H->heap[2*i+1]))
        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 n)
//@requires 1 <= n && n < H->next;
```

```
//@requires is_heap_except_down(H, n);
//@ensures is_heap(H);
{ int i = n;
  while (2*i < H->next)
    //@loop_invariant is_heap_except_down(H, i);
    {
      if (H->heap[i] <= H->heap[2*i] &&
          (2*i+1 >= H->next || H->heap[i] <= H->heap[2*i+1]))
        return;
      if (2*i+1 >= H->next || H->heap[2*i] <= H->heap[2*i+1]) {
        swap(H->heap, i, 2*i);
        i = 2*i;
      } else {
        swap(H->heap, i, 2*i+1);
        i = 2*i+1;
      }
    }
  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.

5 Heapsort

We rarely discuss testing in these notes, but occasionally 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 cool 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 i; int j;
  int seed = 0xc0c0ffee;
  rand_t gen = init_rand(seed);
  int[] A = alloc_array(int, n);
  heap H = heap_new(n);

  print("Testing heap of size "); printint(n);
  print(" "); printint(num_tests); print(" times\n");
  for (j = 0; j < num_tests; j++) {
    for (i = 1; i < n; i++) {
      heap_insert(H, rand(gen));
    }
    for (i = 1; i < n; i++) {
      A[i] = heap_delmin(H);
    }
    assert(heap_empty(H), "heap not empty");
    assert(is_sorted(A, 1, n), "heapsort failed");
  }
  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 or quicksort (the latter we haven't discussed yet).

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.

6 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 *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.*