15-122: Principles of Imperative Computation

Recitation 17 — Josh Zimmerman

Priority Queues: More on Interface vs. Implementation

Priority queues provide a really great example of the distinction programmers make all the time between interface and implementation, so we're going to spend a few minutes re-emphasizing this as it's really important to understand it.

Think back to the interface of priority queues. It lets us do things like make a new priority queue, insert an element into a priority queue, remove the minimum element from a priority queue, see what the minimum element is, and a few others.

Notice that this interface doesn't tell us anything about how the priority queue is implemented. In practice, there are multiple different approaches used to implement priority queues. It's possible to use a sorted array, which is good in cases where insertion into the priority queue is relatively rare, but removal of the minimum is fairly common. On the other hand, if insertion is frequent but removal is rare, it might make more sense to use an unordered array and just always insert at the end. If we have a balance between the two it makes more sense to use a heap, since it provides reasonably fast insert and delmin, making it good for a situation when we insert around as often as we delmin.

All of these approaches make sense in different contexts and should be completely interchangeable (except for their runtimes). If the author of the priority queue decides that a different implementation is better, they can update their code to use the new implementation of priority queues without affecting the client.

So, a client that uses a priority queue should be able to use any one of the different implementations interchangeably in case the author of the priority queue implementation decides that a different implementation is better and needs to switch their code to use a different implementation of priority queues.

In this class, we'll talk about heaps for the most part, since they're the best if we do both inserts and delmins, which is a common case in practice.

It's worth noting that even heaps have multiple possible implementations: we can represent them as an array or more like the way we represent BSTs, with structs that contain data and pointers to the child nodes. Thus, even if a client knows that a priority queue is a heap, they still shouldn't assume internal details of the heap representation, since either one (and probably others) can be used interchangeably.

Heaps

Heaps can be hard to understand, so to help we're going to play with a visualization, which you can find at `http://www.cs.usfca.edu/~galles/visualization/Heap.html`.

`is_heap_except_up`

When we were writing our `pq_insert`, we found that we sometimes needed to break and then restore the heap ordering invariant: sometimes, we needed the current element we're looking at to be smaller than its parents. We fix this eventually, but in the meantime that invariant could be violated.

We use the `is_heap_except_up` function to check that this is the only violation. It checks that the data structure we're given satisfies all of the heap invariants except that the element at index `n` in the array representing the heap is larger than its parents (and that the children of that element are smaller than it). It does check that the children of the element at index `n` (if it has children) are smaller than the parent of the element at index `n` (if it has a parent).

Note that `is_heap_except_up(H, 1)` is equivalent to `is_heap(H)`. (Think about why this is and make sure you understand it.)

```
1  bool is_heap_except_up(heap H, int n) {
2    if (H == NULL) {
3      return false;
4    }
5    //@assert \length(H->data) == H->limit;
6    if (!(1 <= H->next && H->next <= H->limit)) {
7      return false;
8    }
9
10   /* check parent <= node for all nodes except root (i = 1) and n */
11   for (int i = 2; i < H->next; i++)
12   //@loop_invariant 2 <= i;
13   {
14     if (i != n && !(priority(H, i/2) <= priority(H, i))) {
15       return false;
16     }
17     /* for children of node n, check grandparent */
18     if (i/2 == n && (i/2)/2 >= 1 && !(priority(H, (i/2)/2) <= priority(H, i))) {
19       return false;
20     }
21   }
22   return true;
23 }
```

Take a minute to make sure you're comfortable with this code as it's important for the proof that inserting into a heap results in a valid heap.

is_heap_except_down

Similarly to how we needed `is_heap_except_up` to help with insertion, we need an `is_heap_except_down(H, n)` function to help with deletion. It checks that the heap order invariant holds for all nodes except children of the one at index `n`.

```
1  bool is_heap_except_down(heap H, int n) {
2    if (H == NULL) {
3      return false;
4    }
5    //@assert \length(H->data) == H->limit;
6    if (!(1 <= H->next && H->next <= H->limit)) {
7      return false;
8    }
9    /* check parent <= node for all nodes except root (i = 1) */
10   /* and children of n (i/2 = n) */
11   for (int i = 2; i < H->next; i++)
12   //@loop_invariant 2 <= i;
13   {
14     if (i/2 != n && !(priority(H, i/2) <= priority(H, i))) {
15       return false;
16     }
17     /* for children of node n, check grandparent */
18     if (i/2 == n && (i/2)/2 >= 1 && !(priority(H, (i/2)/2) <= priority(H, i))) {
19       return false;
20     }
21   }
22   return true;
23 }
```

Other than one difference on line 14 (and comments), this code is identical to the code for `is_heap_except_up`.

pq_delmin

The general strategy for deleting the minimum element from a heap is to put the element at the bottom right of the heap into the top and repeatedly swap it with its children until the order invariant is restored.

There are some subtleties involved here, so we'll talk some about the code and go over examples with the visualization.

```
1 void sift_down(heap H, int i)
2 //@requires 1 <= i && i < H->next;
3 //@requires is_heap_except_down(H, i);
4 //@ensures is_heap(H);
5 {
6   int n = H->next;
7   int left = 2*i;
8   int right = left + 1;
9   while (left < n)
10  //@loop_invariant 1 <= i && i < n;
11  //@loop_invariant left == 2*i && right == 2*i+1;
12  //@loop_invariant is_heap_except_down(H, i);
13  {
14    if (priority(H,i) <= priority(H,left)
15       && (right >= n || priority(H,i) <= priority(H,right))) {
16      return;
17    }
18    if (right >= n || priority(H,left) < priority(H,right)) {
19      swap(H->data, i, left);
20      i = left;
21    }
22    else {
23      //@assert right < n && priority(H, right) <= priority(H,left);
24      swap(H->data, i, right);
25      i = right;
26    }
27    left = 2*i;
28    right = left+1;
29  }
30  //@assert i < n && 2*i >= n;
31  //@assert is_heap_except_down(H, i);
32  return;
33 }
34
35 elem pq_delmin(heap H)
36 //@requires is_heap(H) && !pq_empty(H);
37 //@ensures is_heap(H);
38 {
39   int n = H->next;
40   elem min = H->data[1];
41   H->data[1] = H->data[n-1];
42   H->next = n-1;
43   if (H->next > 1) sift_down(H, 1);
44   return min;
45 }
```

pq_insert

When inserting into a priority queue, we first put the new element in the bottom-right slot of the heap and then repeatedly swap it with its parent until the heap order invariant is restored.

```
1  void pq_insert(heap H, elem e)
2  //@requires is_heap(H) && !pq_full(H);
3  //@ensures is_heap(H);
4  {
5     H->data[H->next] = e;
6     (H->next)++;
7     /* H is no longer a heap! */
8     /* ordering invariant could be violated at index H->next - 1 */
9     /* remainder could be pulled out as a function sift_up */
10    int i = H->next - 1;
11    while (i > 1 && priority(H,i) < priority(H,i/2))
12    //@loop_invariant 1 <= i && i < H->next;
13    //@loop_invariant is_heap_except_up(H, i);
14    {
15       swap(H->data, i, i/2);
16       i = i/2;
17    }
18    //@assert is_heap(H);
19    return;
20 }
```