15-122: Principles of Imperative Computation

Recitation 16 & 17 — Nivedita Chopra

Priority Queues & Heaps

- A *priority queue* is simply like a queue except that the next element to be dequeued always has the maximum priority.
- A priority queue is an *abstract data type*, which means that we don't know what data structure lies beneath it. All we know is the *interface* i.e. we have a way to insert into it (`pq_insert`) and to get the element of maximum priority from it (`pq_delmin`).
- We usually think of the element with the least integer value being the element with maximum priority. Hence `pq_delmin` gives the element with maximum priority. Also, this means our heaps are *min-heaps*, in contrast to *max-heaps* in which the elements with larger integer value have larger priority.
- A heap is an *implementation* for a priority queue.
- A heap has two important invariants:
  (a) **Ordering Invariant :** Any element in the heap is less than or equal to its children.
  (b) **Shape invariant:** An element is inserted in a manner that, at any point, only the last level is unfilled and elements are filled in this level from left to right.
- As mentioned in lecture, for now, we'll think of an array that stores the values in the heap, such that the children of the element at index i are at $2i$ and $2i + 1$
  Notice that this means that we can't use the index 0 of the array!
- While writing code, we'll see that we need one important invariant for heaps — the element which will be accessed/deleted from the heap must have minimum value. To maintain this invariant, while inserting or deleting from the priority queue, we need to do some special operations.
- We'll also see some important concepts that are very common while dealing with operations on complex data structures:
  (a) The invariants that hold for the operation as a whole (which are true at the beginning and at the end of the operation) are not necessarily true while the operation is being performed. So we are permitted to ***temporarily** violate invariants* while performing the operation and then restore them at the end.
  (b) Thus, to prove correctnesss, we need to find variations of the original invariants that hold true while the operation is being performed and reason that these will imply the stronger invariants that are true at the end of the operation.

Checkpoint 0

When representing heaps as arrays, why can't we use the index 0 of the array?

Playing with heaps!

Use the visualization at `http://www.cs.usfca.edu/~galles/visualization/Heap.html` to insert the following elements into the heap, in the given order.

$$5, 3, 6, 7, 2, 6$$

(Here's some space if you want to draw the final heap obtained)

Some observations about the heap you see now:

- The smallest element in the heap (i.e. 2) is at the root. Convince yourself that this must always hold true because of the ordering invariant
- The largest element in the heap (i.e. 7) is in the last level of the heap. Again, convince yourself that this must always hold true by the ordering invariant.
- The heap can contain duplicates, since the ordering invariant states that the children of an element must be greater than *or equal to* the element

Now perform a few operations of removing the smallest element from the heap.

Checkpoint 1

What is the time complexity of the insert and remove minimum functions? Justify your answer.

Discussing the code for deletion from a heap

When we delete the minimum element from the heap, we notice that while we're performing the operation, the ordering invariant of the heap is violated i.e. the intermediate heaps don't satisfy the ordering invariant.

To prove that the ordering invariant is satisfied at the end of the delete operation, we need to come up with an invariant that will hold of the intermeditate heaps and that is strong enough to prove the ordering invariant for the final heap.

Turns out an invariant that holds at each step is that the heap invariant is satisfied at all points, except at the current element, which may be larger than its children. We express this invariant as the `is_heap_except_down` function.

```
bool is_heap_except_down(heap H, int n) {
 if (!is_safe_heap(H)) 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;
}
```

Using this `is_heap_except_down` function as an invariant, we can write the `sift_down` and `pq_delmin` function 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);
 {
   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;
}

```

```
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;
}
```

Checkpoint 2

(Optional) Come up with a proof of correctness for the `pq_delmin` function. This need not be super formal, but you should be able to convince yourself that the loop invariant is preserved and that the loop invariant and negation of the loop guard imply the postcondition.