15-122: Principles of Imperative Computation

Recitation Week 9

Nivedita Chopra, Rob Simmons

Visualizing heaps

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

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

Priority queue client and library interface

We use heaps to efficiently implement the *priority queue* interface.

```
1  /* Client interface */
2  // typedef _______ elem;
3  typedef void* elem;
4
5  // f(x,y) returns true if e1 is STRICTLY higher priority than e2
6  typedef bool higher_priority_fn(elem e1, elem e2);
7
8  /* Library interface */
9  //typedef ______* heap_t;
10
11 bool heap_empty(heap_t P);
12 bool heap_full(heap_t P);
13 heap_t heap_new(int capacity, higher_priority_fn* priority)
14   /*@requires capacity > 0 && priority != NULL; @*/
15   /*@ensures heap_empty(\result); @*/ ;
16
17 void heap_add(heap_t H, void* e)
18   /*@requires !heap_full(P) && e != NULL; @*/ ;
19
20 void* heap_rem(heap_t H) // Removes highest priority element
21   /*@requires !heap_empty(P); @*/
22   /*@ensures \result != NULL; @*/ ;
```

Checkpoint 0

If the client's `elem` type is picked to be `void*`, will this client interface cause `heap_new(20, &higher_priority)` to return a min-heap, a max-heap, or something else?

```
1  bool higher_priority(void* x, void* y)
2  //@requires x != NULL && \hastag(int*, x);
3  //@requires y != NULL && \hastag(int*, y);
4  {
5    return *(int*)x > *(int*)y;
6  }
```

Checkpoint 1

Define a client interface that ensures that, in a priority queue of (pointers to) strings, the *longest* strings always gets returned first.

Deletion of the lowest-priority element from a heap

```
1  void* heap_rem(heap* H)
2  //@requires is_heap(H) && !heap_empty(H);
3  //@ensures is_heap(H) && \result != NULL;
4  {
5    int i = H->next;
6    void* min = H->data[1];
7    (H->next)--;
8    if (H->next > 1) {
9      H->data[1] = H->data[H->next];
10     sift_down(H);
11   }
12   return min;
13 }
14
15 void sift_down(heap* H)
16 //@requires _____________________________________________________;
17 //@ensures is_heap(H);
18 {
19   int i = 1;
20
21   while (_____________________________________________)
22   //@loop_invariant 1 <= i && i < H->next;
23   //@loop_invariant is_heap_except_down(H, i);
24   //@loop_invariant grandparent_check(H, i);
25   {
26     int left = 2*i;
27     int right = left+1;
28
29     if (_____________________________________________)
30       return;
31     if (_____________________________________________) {
32       swap_up(H, left);
33       i = left;
34     } else {
35       //@assert _____________________________________;
36       swap_up(H, right);
37       i = right;
38     }
39   }
40 }
```

Checkpoint 2

(a) Check that the preconditions imply the loop invariants hold initially, and that they are satisfied when `sift_down` is called from `pq_rem`.

(b) Show that the grandparent check is necessary as a loop invariant.

(c) Prove that the loop invariants imply the postcondition for the return on line 17 and on line 32.