

15-122: Principles of Imperative Computation

R9: Mind your Priorities and Queues

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 H);
12 bool heap_full(heap_t H);
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, elem x)
18 /*@requires !heap_full(H); @*/ ;
19
20 elem heap_rem(heap_t H) // Removes highest priority element
21 /*@requires !heap_empty(H); @*/
```

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

Write a function (of type `higher_priority_fn`) that ensures that, in a priority queue of (pointers to) strings, the longest strings always gets returned first. The `string_length` function may be helpful.

Deletion of the highest-priority element from a heap

You may need the following functions: `is_safe_heap`, `ok_above`

```
1 elem heap_rem(heap* H)
2 //@requires is_heap(H) && !heap_empty(H);
3 //@ensures is_heap(H);
4 {
5     int i = H->next;
6     elem 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     while ( _____ )
21         //@loop_invariant 1 <= i && i < H->next;
22         //@loop_invariant is_heap_except_down(H, i);
23         //@loop_invariant grandparent_check(H, i);
24     {
25         int left = 2*i;
26         int right = left+1;
27         if ( _____ )
28             return;
29         if ( _____ ) {
30             swap_up(H, left);
31             i = left;
32         } else {
33             //@assert _____;
34             swap_up(H, right);
35             i = right;
36         }
37     }
38 }
```

Checkpoint 2

- Check that the preconditions imply the loop invariants hold initially, and that they are satisfied when `sift_down` is called from `pq_rem`.
- Show that the grandparent check is necessary as a loop invariant.
- Prove that the loop invariants imply the postcondition for the return on line 28 and on line 37.