Visualizing heaps

Use the visualization at [http://www.cs.usfca.edu/~galles/visualization/Heap.html](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.

```c
/* Client interface */
typedef ______ elem;
typedef void* elem;

// f(x,y) returns true if e1 is STRICTLY higher priority than e2
typedef bool higher_priority_fn(elem e1, elem e2);

/* Library interface */
typedef ______* heap_t;

bool heap_empty(heap_t P);
bool heap_full(heap_t P);
heap_t heap_new(int capacity, higher_priority_fn* priority);
/*@requires capacity > 0 && priority != NULL; @*/
/*@ensures heap_empty(result); @*/;

void heap_add(heap_t H, void* e);
/*@requires !heap_full(P) && e != NULL; @*/;

void* heap_rem(heap_t H) // Removes highest priority element
/*@requires !heap_empty(P); @*/
/*@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?

```c
bool higher_priority(void* x, void* y)
//@requires x != NULL && hastag(int*, x);
//@requires y != NULL && hastag(int*, y);
{
    return *(int*)x > *(int*)y;
}
```

**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

```c
void* heap_rem(heap* H)
//@requires is_heap(H) && !heap_empty(H);
//@ensures is_heap(H) && \result != NULL;
{
    int i = H->next;
    void* min = H->data[1];
    (H->next)--; 
    if (H->next > 1) {
        H->data[1] = H->data[H->next];
        sift_down(H);
    }
    return min;
}

void sift_down(heap* H)
//@requires ____________________________;
//@ensures is_heap(H);
{
    int i = 1;
    while (_)
    { 
        int left = 2*i;
        int right = left+1;
        if (_)
            return;
        if (_)
            swap_up(H, left);
            i = left;
        } else {
            //assert ____________________________;
            swap_up(H, right);
            i = right;
        }
    }
}
```

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.