Priority Queues

A priority queue is a data structure that allows us to keep track of a set of elements and access them in order according to their priority. If we’re working with priority queues, which have type pq, we know we have the following interface functions:

- Constructing new ones of a given size: \( \text{pq pq_new(int capacity)} \)
- Checking whether they’re full: \( \text{bool pq_full(pq P)} \)
- Checking whether they’re empty: \( \text{bool pq_empty(pq P)} \)
- Finding the element with highest priority: \( \text{elem pq_min(pq P)} \)
  - In class, ints were elements and smaller numbers had higher priority
- Removing the element with the highest priority: \( \text{elem pq_delmin(pq P)} \)
- Inserting a new element, e, into the priority queue: \( \text{void pq_insert(pq P, elem e)} \)

We could construct a priority queue using lots of different data structures. Each has its own advantages. Can you think of some for priority queues implemented as sorted arrays? Unsorted arrays? Tree-like structs with pointers back and forth between nodes?

Heaps

We use a heap to implement priority queues in this class. Heaps involve storing each of the elements in an array of size \( \text{capacity+1} \). We put the first element in at index 1. This convention allows us to find the parent and children of a node quite easily. For the element at index \( i \), its parent (if it’s not the root) is at index \( i/2 \), and its children (if the heap has sufficiently many elements) are at index \( 2i \) and \( 2i+1 \).

This implementation allows us to maintain the shape invariant of the heap, as well as the ordering invariant pretty efficiently. Recall that the shape invariant is that we build the heap left to right from the top, and the ordering invariant is that each element is greater than its parent and less than its children. This picture, taken from Prof. Pfenning’s lecture notes, illustrates the invariants being followed for heaps from size 1 to 7.

When we insert into a heap, we have to break the ordering invariant temporarily. We restore it in the end, but how do we know? We can use some functions to help us out along the way. Consider, for example, the following function,
is_heap_except_up(heap H, int n). It checks that, except for node n, the ordering invariant is satisfied. Note that is_heap_except_up(H, 1) == is_heap(H).

bool is_heap_except_up(heap H, int n) {
    if (H == NULL) {
        return false;
    }
    //@assert \length(H->data) == H->limit;
    if (!(1 <= H->next && H->next <= H->limit)) {
        return false;
    }
    /* check parent <= node for all nodes except root (i = 1) and n */
    for (int i = 2; i < H->next; i++) {
        //@loop_invariant 2 <= i;
        {
            /* check ordering invariant except at node n */
            if (i != n && !(priority(H, i/2) <= priority(H, i))) {
                return false;
            }
        }
    }
    return true;
}

Look at how it fits into the insertion function:

void pq_insert(heap H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H);
{
    int n = H->next;
    H->data[n] = e;
    (H->next)++;
    /* H is no longer a heap! */
    /* ordering invariant could be violated at n */
    /* remainder could be pulled out as a function sift_up */
    int i = n;
    while (i > 1 && priority(H,i) < priority(H,i/2)) {
        //@loop_invariant 1 <= i && i < H->next;
        //@loop_invariant is_heap_except_up(H, i);
        {
            swap(H->data, i, i/2);
            i = i/2;
        }
        //@assert is_heap(H);
    }
    return;
}

Let’s prove is_heap_except_up(H, i) as a loop invariant for pq_insert:

Init:
We know by the precondition that H is a heap, and since we’ve only modified the n\textsuperscript{th} position, or the last position in the heap, we know that the rest of the heap is still valid.

Preservation:
Assume that is_heap_except_up(H, i) is true. We want to show that is_heap_except_up(H, i’) is true by the end of the loop iteration.
We know by the loop condition that $\text{priority}(H, i) < \text{priority}(H, i/2)$. So to restore a proper ordering, we need element $i$ to switch with $i/2$. Swapping them does just that. We also divide $i$ by 2, giving us $i' = i/2$. Then the priority of the element at position $i$ is greater than both of its children.

We know that it’s greater than the one we swapped it with because we checked explicitly. We know it’s greater than the other child because it’s greater than the other child’s old parent, and ordering is restored, and thus $\text{is_heap_except_up}(H, i')$ is true.