Priority Queues

Like queues, stacks, and unbounded arrays, a priority queue is a data structure which supports an addition (enq/add/push) and deletion (deq/pop/rem) operation. Unlike stacks and queues, which have fixed behavior defining which element gets removed (FIFO for queues, LIFO for stacks), priority queues allow the client to define the order. This is done by requiring the client to provide a *priority function*, `bool higher_priority(elem x,elem y)`, which returns true when the priority of `x` is strictly greater than the priority of `y`.

Apart from this difference, there are a lot of similarities in the interface function we'll implement:

- `pq_new` creates a new priority queue
- `pq_add` adds elements
- `pq_rem` removes elements
- `pq_peek` is an extra function we implement that allows us to see what element would be removed next without actually removing it

Often, we'll want to implement a *bounded* priority queue. In this case, we'll need to pass a value `max_capacity` to `pq_new`, and will also find it helpful to have a function `pq_full`. There are a few different implementation options:

- Using an unordered array of length `max_capacity`. Insertion is `O(1)`, but both `pq_peek` and `pq_rem` are `O(n)` because we have to scan over all values (and then swap it with the last value in the array and decrement `n`).
- Using an ordered array. Insertion is `O(n)`, because while we can find the spot to insert in `O(log n)`, we then have to shift items which is of the order `O(n)`. Finding the highest priority element is `O(1)`, and we can arrange the array so that the highest-priority element goes at the end to have deletion be `O(1)`.
- Using an AVL tree. Based on the AVL height invariant, we know that insertion will always be `O(log n)`. Deletion will also be `O(log n)`.
- Using a heap, where `pq_add` and `pq_rem` are on the same order as AVL trees but `pq_peek` can be done in constant time (because we just have to look at the root, with no restoring of invariants required).

Table 1. Operation Complexities

	`pq_add`	`pq_rem`	`pq_peek`
unordered array	`O(1)`	`O(n)`	`O(n)`
ordered array	`O(n)`	`O(1)`	`O(1)`
AVL tree	`O(log n)`	`O(log n)`	`O(log n)`
heap	`O(log n)`	`O(log n)`	`O(1)`

When evaluating data structure options, we generally consider the complexity of running `n` insertions and deletions (because this is typically what we'll do with priority queues). A heap is more efficient in these operations than arrays: ``O(2nlog(n)) → O(nlogn)`vs. `O(2n^2) → O(n^2)``.

Min-heaps

Min-heaps have the structure of binary trees, but with special properties. The key is the min-heap *ordering invariant*: the key of each node must always be less-than-or-equal-to the key of its children. Based on this property, we know that for any min-heap we can find the minimal element at the root of the tree (thus, `O(1)`).

We also have a *shape invariant* for heaps, where the shape is determined entirely based on the number of elements within it: we fill the tree level by level from left to right. This enforces balance.

Adding to heaps

Based on the shape invariant, we know where to insert a new node. However, we don't know that the insertion won't violate the ordering variant. To resolve this, we insert it into the spot anyways and then make changes that restore the invariant. We simply need to swap the inserted node with each parent that is strictly greater than it until it is greater than its parent or we reach the root. This operation is called *sifting up*, and its complexity is `O(l)` where `l` is the number of levels in the tree. We can express this in terms of `n` as `O(log n)`.

Removing the Minimal Element

We obviously can't just delete the root element and be done, because we won't have any tree left. Instead, we swap the last node with the root and then delete the new last node. This violates the ordering invariant (except in a limited number of small cases), however, so we need to resolve this. To do so, we examine the children of the root node and (if they are smaller than it) we swap the node with its *smaller* child. By choosing the smaller child, we ensure that the new parent will be smaller than both its new children (its previous child and the sibling it was previously smaller than). This operation is called *sifting down*, and we repeat it until the ordering invariant is satisfied or we reach a leaf.

Representing Heaps as Arrays

We can represent heaps with structs with pointers, but arrays provide an elegant option. Starting at root index `i=1`, we append a 0 to the binary representation of `i` to obtain the index of the left child and a 1 to obtain the index of the right child. Thus, for a node at index `i`, its left child would be at `2i`, its right child at `2i+1`, and its parent at `i/2`.

Restoring Invariants

Next, we examine how to restore invariants in code. This is important because many of the previously-mentioned operations involve violating the order invariant and then reestablishing it. Consider the following structure for our implementation of heaps:

```
typedef struct heap_header heap;
struct heap_header{
    int limit;                    /* limit = capacity+1 */
    int next;                     /* 1 <= next && next <= limit */
    elem[] data;                  /* length(data) == limit */
    higher_priority_fun* prior;   /* != NULL */
}
```

prior is the function the client must provide to tell how ordering should be done. limit is used because we'll actually have one extra spot in our array based on the 1-index system described above.

We first implement a basic function to check if the data structure we're dealing with is a heap. This doesn't check that the heap is valid, but is just a basic screening function to ensure that basic preconditions are met.

```
bool isSafeHeap(heap* H){
    return H != NULL
        && (1 <= H->next && H->next <= H->limit)
        && isArrayExpectedLength(H->data,H->limit)
        && H->prior != NULL
}
```

We check that the heap is defined, next is within our range (1-index) and less than limit, that the array is the length that we expect based on H→limit, and that it is associated with some priority function. Next, we implement a function for checking if objects are correctly placed in relation to one another:

```
bool okAbove(heap* H, int i, int j)
//@requires isSafeHeap(H);
//@requires 1 <= i && i < H->next;
//@requires 1 <= j && j < H->next;
{
    return !(*H->prior)(H->data[j],H->data[i]);
}
```

We also define a function swapUp that swaps an element with its parent:

```
void swap_up(heap* H, int i)
//@requires isSafeHeap(H);
//@requires 2 <= i && i < H->next;
{
    elem tmp = H->data[i];
    H->data[i] = H->data[i/2] // bit shift 1 right is equivalent to dividing by 2
    H->data[i/2] = tmp;
}
```

To check the ordering invariant where we require children to be greater than or equal to their parents, we iterate through the array and compare the value of each node except `i=1` to its parent.

```
bool isHeapOrdered(heap* H)
//@requires isSafeHeap(H);
{
    for(int i = 2; i < H->next; ++i){
        if(!ok_above(H,i/2,i)) return false;
    }
    return true;
}
```

Finally, we use this function with `isSafeHeap` to check if the heap is valid:

```
bool isHeap(heap* H){
    return isSafeHeap(H)&& isHeapOrdered(H);
}
```

Creating Heaps

We first implement two functions, `heapEmpty` and `heapFull`:

```
bool heapEmpty(heap* H)
//@requires isHeap(H);
{
    return H->next == 1;
}

bool heapFull(heap* H)
//@requires isHeap(H);
{
    return H->next == H->limit;
}
```

To create our heap, we allocate memory and initialize fields:

```
heap* heapNew(int capacity, higher_priority_fn* prior)
//@requires capacity > 0 && prior != NULL;
//@ensures isHeap(\result) && heap_empty(result);
{
    heap* H = alloc(heap);
    H->limit = capacity+1;
    H->next = 1;
    H->data = alloc_array(elem,capacity+1);
    H->prior = prior;
    return H;
}
```

Insert and Sifting Up

We know where to insert new elements, at `H->next` based on the shape invariant. After insertions, we increment `next`.

```
void heapAdd(heap* H, elemx)
//@requires isHeap(H) && !heapFull(H);
//@ensures isHeap(H);
{
    H->data[H->next] = x;
    (H->next)++;
}
```

This insertion violates the order invariant, however. Thus, we need to *sift up* until the invariant is true again. An implementation will look roughly like:

```
int i = H->next -1;
while(i > 1 && !okAbove(H,i/2,i)){
    //@loop_invariant 1 <= i && i < H->next;
    //@loop_invariant isHeapExceptUp(H,i);
    swapUp(H->data,i);
    i = i/2;
}
```

Note that `isHeapExceptUp` is a modified version of `isHeap` that checks if the heap excluding the index we are at satisfies the heap invariants. It is written as follows:

```
bool isHeapExceptUp(heap* H, int n)
//@requires isSafeHeap(H);
//@requires 1 <= n && n < H->next;
{
    for(int i = 2; i < H->next; ++i){
    //@loop_invariant 2 <= i;
        if(!(i == n || okAbove(H,i/2,i))) return false;
    }
    return true;
}
```

Note that `isHeapExceptUp(H,1)` is equivalent to `isHeap`. If we try to outline a proof of correctness, we consider what happens when we `siftUp` a node `x`. We need to know to continue to loop if the previous children of `x` are greater than their new parent, which is their previous grandparent. Thus, we write an additional function to check for this invariant:

```
bool grandparentCheck(heap* H, int n)
//@requires isSafeHeap(H);
//@requires i <= n && n < H->next;
{
    if (n==1) return true;
    if(n*2 >= H->next) return true; // no children
    if*n*2 + 1 == H->next)
        return okAbove(H,n/2,n*2);
    return okAbove(H,n/2,n*2) && okAbove(H,n/2,n*2+1);
}
```

If we add this function check to the invariants of our previously defined function, we have function that provably restores our heap variants:

```
int i = H->next -1;
while(i > 1 && !okAbove(H,i/2,i)){
    //@loop_invariant 1 <= i && i < H->next;
    //@loop_invariant isHeapExceptUp(H,i);
    //@loop_invariant grandparentCheck(H,i);
    swapUp(H->data,i);
    i = i/2;
}
```

To complete (vague) proof, we examine our post-conditions. From the loop invariants, we know:

- `1 ⇐ i < next`
- `isHeapExceptUp(H,i)`
- `i⇐1` or `okAbove(H,i/2,i)`

TODO: postcondition case analysis

Deleting the Minimum and Sifting Down

As described above, deleting the minimum involves swapping the root and last element and then sifting the root down until the heap invariant is restored. We carry this out by first checking that H is a non-empty heap, swapping the minimal element with the element at index next-1, and deleting the last element by decrementing next.

```
elem heapRem(heap* H)
//@requires isHeap(H) && !isEmpty(H);
//@ensures isHeap(H);
{
    int i = H->next;
    elem min = H->data[1];
    (H->next)--;

    if(H->next > 1){
        H->data[1] = H->data[H->next];
        sift_down(H);
    }

    return min;
}
```

After the swap and removal, so long as there is at least one element we need to sift_down. When sifting down, we need a similar function to isHeapExceptUp, except in this case checking the invariant between each node and its children rather than each node and its parent. With this minor change, implementation is very similar to what we did before:

```
bool isHeapExceptDown(heap H, int n)
//@requires isSafeHeap(H);
//@requires 1<= n && n < H->next;
{
    for)int i = 2; i < H->next; ++i){
    //@loop_invariant 2 <= i;
        if(!(i/2 == n || okAbove(H,i/2,i))) return false;
    }
    return true;
}
```

With this function to use as an invariant, we now can write siftDown:

```
void siftDown(heap H)
//@requires isSafeHeap(H) && H->next > 1 && isHeapExceptDown(H,1);
//@ensures isHeap(H);
{
    int i = 1;

    // while there is a left child...
    while(2*i < H->next)
    //@loop_invariant 1 <= i && i < H->next;
    //@loop_invariant isHeapExceptDown(H,i);
    //@loop_invariant grandparentCheck(H,i);
    {
        int left = 2*i;
        int right = left + 1;

        if(okAbove(H,i,left)
            && (right >= H->next || okAbove(H,i,right)))
          return;
        if(right >= H->next || okAbove(H,i,right)){
            swapUp(H,left);
            i=left;
        }
        else{
            //@assert right < H->next && okAbove(H,right,left);
            swapUp(H,right);
            i=right;
        }
        //@assert i < H->next && 2*i >= H->next;
        //@assert isHeapExceptDown(H,i);
        return;
    }
}
```

Sources

- 15-122 Principles of Imperative Computation Notes, Lecture 18 2015
- 15-122 Principles of Imperative Computation Notes, Lecture 19 2015