Lecture Notes on Priority Queues

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 15
March 3, 2011

1 Introduction

In this lecture we will look at *priority queues* as an abstract type and discuss several possible implementations. We then pick the implementation as *heaps* and start to work towards an implementation. Heaps have the structure of binary trees, a very common structure since a (balanced) binary tree with n elements has depth $O(\log(n))$. During the presentation of algorithms on heaps we will also come across the phenomenon that invariants must be temporarily violated and then restored. We will study this in more depth in the next lecture. From the programming point of view, we will see a cool way to implement binary trees in arrays which, alas, does not work very often.

2 Priority Queues

Priority queues are a generalization of stacks and queues. Rather than inserting and deleting elements in a fixed order, each element is assigned a *priority* represented by an integer. We always remove an element with the highest priority, which is given by the *minimal* integer priority assigned. Priority queues often have a fixed size. For example, in an operating system the runnable processes might be stored in a priority queue, where certain system processes are given a higher priority than user processes. Similarly, in a network router packets may be routed according to some assigned priorities. In both of these examples, bounding the size of the queues helps to

prevent so-called *denial-of-service attacks* where a system is essentially disabled by flooding its task store. This can happen accidentally or on purpose by a malicious attacker.

Here is an abstract interface to a (bounded) priority queue. Our implementation uses a data structure call a *heap* which we discuss shortly.

```
typedef struct heap* heap;
heap heap_new(int capacity)    /* create new heap of given capacity */
//@requires capacity >= 0;
  ;
bool heap_empty(heap H);        /* is H empty? */
bool heap_full(heap H);         /* is H full? */
void heap_insert(heap H, int x) /* insert x into H */
//@requires !heap_full(H);
  ;
int heap_min(heap H)            /* find minimum */
//@requires !heap_empty(H);
  ;
int heap_delmin(heap H)         /* delete minimum */
//@requires !heap_empty(H);
  ;
```

The interface here is somewhat simplified in that we only insert the integers that represent priorities, rather than data elements to which a priority is attached. This simplification is inessential—it just makes it slightly easier to test the code.

3 Some Implementations

Before we come to heaps, it is worth considering different implementation choices and consider the complexity of various operations.

The first idea is to use an unordered array of size $limit$, where we keep a current index n. Inserting into such an array is a constant-time operation, since we only have to insert it at n and increment n. However, finding the minimum will take $O(n)$, since we have to scan the whole portion of the array that's in use. Consequently, deleting the minimal element also takes $O(n)$: first we find the minimal element, then we swap it with the last element in the array, and decrement n.

A second idea is to keep the array sorted. In this case, inserting an element is $O(n)$. We can quickly (in $O(\log(n))$ steps) find the place i where

it belongs using binary search, but then we need to shift elements to make room for the insertion. This take $O(n)$ copy operations. Finding the minimum is $O(1)$ (since it is stored at index 0 in the array). We can also make deleting it $O(1)$ if we keep the array sorted in descending order, or if we keep two array indices: one for the smallest current element and one for the largest.

To anticipate our analysis, heaps will have logarithmic time for insert and deleting the minimal element.

	insert	delmin	findmin
unordered array	$O(1)$	$O(n)$	$O(n)$
ordered array	$O(n)$	$O(1)$	$O(1)$
heap	$O(\log(n))$	$O(\log(n))$	$O(1)$

4 The Heap Invariant

Typically, when using a priority queue, we expect the number of inserts and deletes to roughly balance. Then neither the unordered nor the ordered array provide a good data structure since a sequence of n inserts and deletes will have worst-case complexity $O(n^2)$.

The idea of the heap is to use something cleverly situated in between. A heap is like an array that is ordered to some extent: enough, that the least element can be found in $O(1)$, but not so rigidly that inserting would take $O(n)$ time. A heap is a binary tree where the invariant guarantees that the least element is at the root. For this to be the case we just require that the key of a node is less or equal to the keys of its children. Alternatively, we could say that each node except the root is greater or equal to its parent.

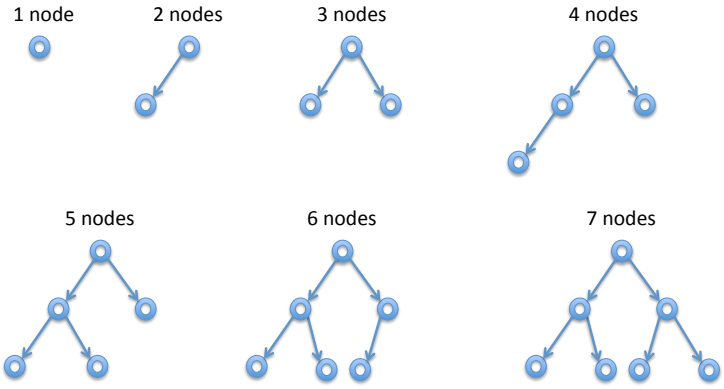
Heap ordering invariant, alternative (1) : The key of each node in the tree is less or equal to all of its childrens' keys.

Heap ordering invariant, alternative (2) : The key of each node in the tree except for the root is greater or equal to its parent's key.

These two characterizations are equivalent. Sometimes it turns out to be convenient to think of it one way, sometimes the other. Either of them implies that the minimal element in the heap is a the root, due to the transitivity of the ordering.

There is a second invariant, not as crucial but convenient, which is that we fill the tree level by level, from left to right. This means the shape of the

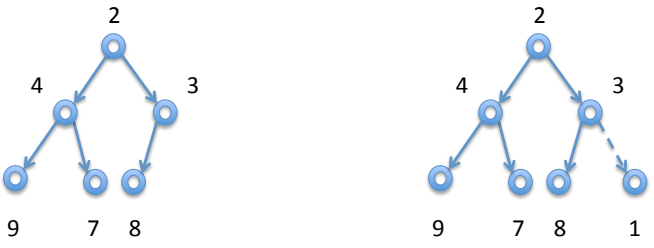
tree is completely determined by the number of elements in it. Here are the shapes of heaps with 1 through 7 nodes.



We call this latter invariant the *shape invariant*.

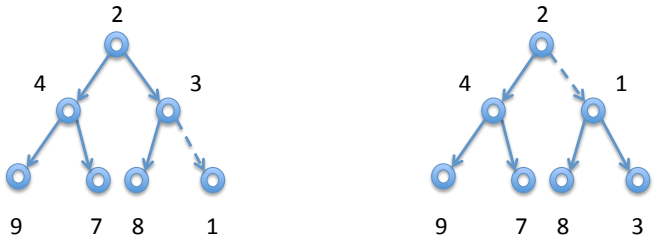
5 Inserting into a Heap

When we insert into a heap, we already know (by the shape invariant) where a new node has to go. However, we cannot simply put the new data element there, because it might violate the ordering invariant. We do it anyway and then work to restore the invariant. We will talk more about temporarily violating a data structure invariant in the next lecture, as we develop code. Let's consider an example. On the left is the heap before insertion of data with key 1; on the right after, but before we have restored the invariant.



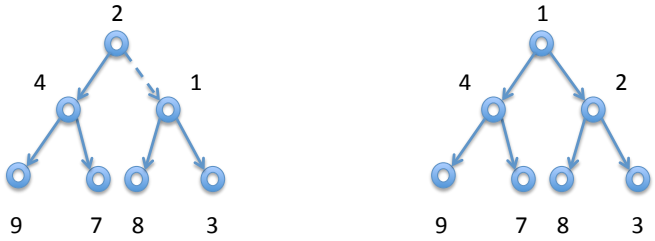
The dashed line indicates where the ordering invariant might be violated. And, indeed, $3 > 1$.

We can fix the invariant at the dashed edge by swapping the two nodes. The result is shown on the right.



The link from the node with key 1 to the node with key 8 will always satisfy the invariant, because we have replaced the previous key 3 with a smaller key (1). But the invariant might now be violated going up the tree to the root. And, indeed $2 > 1$.

We repeat the operation, swapping 1 with 2.



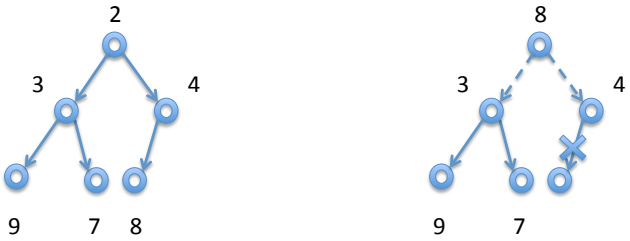
As before, the link between the root and its left child continues to satisfy the invariant because we have replaced the key at the root with a smaller one. Furthermore, since the root node has no parent, we have fully restored the ordering invariant.

In general, we swap a node with its parent if the parent has a strictly greater key. If not, or if we reach the root, we have restored the ordering invariant. The shape invariant was always satisfied since we inserted the new node into the next open place in the tree.

The operation that restores the ordering invariant is called *sifting up*, since we take the new node and move it up the heap until the invariant has been reestablished. The complexity of this operation is $O(l)$, where l is the number of levels in the tree. For a tree of $n \geq 1$ nodes there are $\log(n) + 1$ levels. So the complexity of inserting a new node is $O(\log(n))$, as promised.

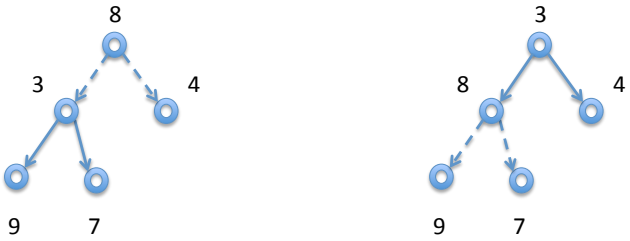
6 Deleting the Minimal Element

To delete the minimal element from the priority queue we cannot simple delete the root node where the minimal element is stored, because we would not be left with a tree. But by the shape invariant we know what the tree has to look like. So we take the *last* element in the tree and move it to the root, and delete that last node.



However, the node that is now at the root might have a strictly greater key one or both of its children, which would violate the ordering invariant.

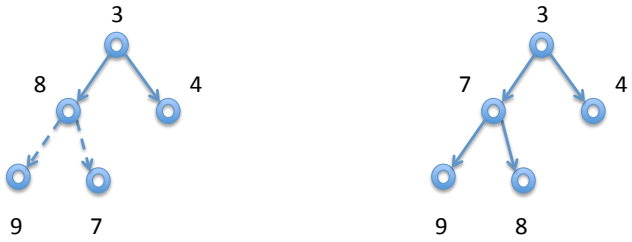
If the ordering invariant in indeed violated, we swap the node with the *smaller* of its children.



This will reestablish the invariant at the root. In general we see this as follows. Assume that before the swap the invariant is violated, and the left child has a smaller key than the right one. It must also be smaller than the root, otherwise the ordering invariant would hold. Therefore, after we swap the root with its left child, the root will be smaller than its left child. It will also be smaller than its right child, because the left was smaller than the right before the swap. When the right is smaller than the left, the argument is symmetric.

Unfortunately, we may not be done, because the invariant might now be violated at place where the root ended up. If not, we stop. If yes, we

compare the children as before and swap with the smaller one.



We stop this downward movement of the new node if either the ordering invariant is satisfied, or we reach a leaf. In both cases we have fully restored the ordering invariant. This process of restoring the invariant is called *sifting down*, since we move a node down the tree. As in the case for insert, the number of operations is bounded by the number of levels in the tree, which is $O(\log(n))$ as promised.

7 Finding the Minimal Element

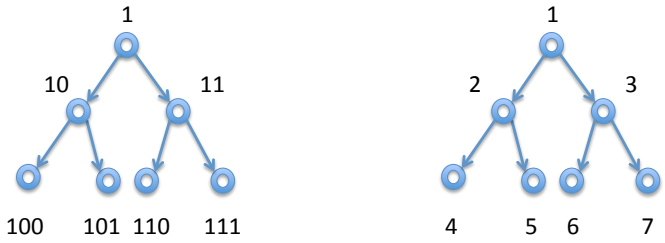
Since the minimal element is at the root, finding the minimal element is a constant-time operation.

8 Representing Heaps as Arrays

A first thought on how to represent a heap would be using structs with pointers. The sift-down operation follows the pointers from nodes to their children, and the sift-up operation follows goes from children to their parents. This means all interior nodes require three pointers: one to each child and one to the parent, the root requires two, and each leaf requires one.

While a pointer structure is not unreasonable, there is a more elegant representation using arrays. We use binary numbers as addresses of tree nodes. Assume a node has index i. Then we append a 0 to the binary representation of i to obtain the index for the left child and a 1 to obtain the index of the right child. We start at the root with the number 1. If we tried to use 0, then the root and its left child would get the same address. The node number for a full three-level tree on the left in binary and on the right

in decimal.



Mapping this back to numeric operations, for a node at index i we obtain its left child as $2*i$, its right child as $2*i+1$, and its parent as $i/2$. Care must be taken, since any of these may be out of bounds of the array. A node may not have a right child, or neither right nor left child, and the root does not have a parent.

With a heap structure we always keep a $next$ index which is the index where the next element should be added. There is also $limit$, which is the size of the array. Note that the capacity of the heap is one less than the size of the array `heap` since we do not use position 0.

```
struct heap {
  int limit;
  int next;
  int[] heap;
};
```

We will write code to check the heap invariant in the next lecture; here we just write the code to allocate a new (empty) heap, and to check if a heap is empty or full.

A heap is empty if the next element to be inserted would be at index 1, which is always the root of the heap.

```
bool heap_empty(heap H)
//@requires is_heap(H);
{
  return (H->next == 1);
}
```

A heap is full if the next element to be inserted would be at index $limit$, which is just beyond the end of the array.

```
bool heap_full(heap H)
//@requires is_heap(H);
{
  return (H->next == H->limit);
}
```

To create a new heap, we allocate a struct and an array and set all the right initial values.

```
heap heap_new(int limit)
//@ensures is_heap(\result) && heap_empty(\result);
{
  heap H = alloc(struct heap);
  int[] heap = alloc_array(int, limit);
  H->limit = limit;
  H->next = 1;
  H->heap = heap;
  return H;
}
```