CS 5084 Introduction to Algorithms - Analysis and Design

Section 2.5 A More Complex Data Structure - Priority Queue

Joe Johnson

Table of Contents

Table of Contents

The Problem - How to Store Elements in Sorted Order with Fast Retrieval

A Common Problem - We would like to store elements of a set, S, in sorted order so that both adding and removing elements to/from S can be done efficiently.

Consider the Stable Matching Problem where we needed to add and remove hospitals from the collection of free hospitals. We didn't have to store them in order, in this case, but one could easily imagine a situation in which an algorithm might like to select elements from the collection according to some priority value.

Consider the real-time event scheduling of processes on a computer, where processes must be scheduled for the CPU, but not in terms of arrival time, but instead, in terms of a priority value.

Background - Some Potential Solutions

Let's consider the following options:

- Insert the elements into a list and have a pointer labeled *Min* to the one with minimum key.
  - This makes adding new elements easy, but extraction of the minimum is hard since we need to do an $O(n)$ search for the new minimum after removing the current min element at *Min*.
- Store the elements in the list, but sort them by key value.
  - This makes it easy to extract the element with the smallest key, but adding a new element becomes problematic.
  - If we store the elements in a linked list, we'll need to traverse the list with each add to find the right place to insert, $\Longrightarrow O(n)$.
  - If we store the elements in an array, then we can use a binary search to find the insertion point, $\Longrightarrow O(logn)$. But then we need to move the higher elements up by one to make room for the new element, $\Longrightarrow O(n)$.

Options

In either approach, we're locked into $O(n)$ time for either the insertion or deletion procedure.

Table of Contents

Definition of a Heap

Definition - A *heap* is a balanced binary tree data structure in which two conditions hold:

1. The tree is complete, meaning (almost) all non-leaf nodes have two child nodes, i.e., *there are no holes in the binary tree*.
2. Each element, v is assigned a key value, $key(v)$.
3. Elements are weakly ordered according to the key value, meaning that for every element v at node i, the element w at node i's parent satisfies $key(w) \leq key(v)$.

A Data Structure for Representing a Heap

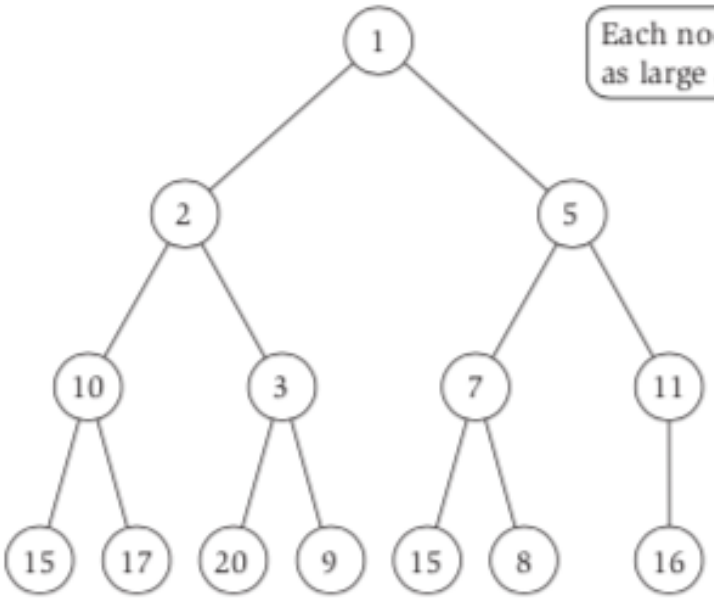
Two options for a data structure to represent a heap:

- We can create a Node class with three pointers - parent, right child, and left child - and build up the heap tree by creating instances of the Node class and setting appropriate references between them.
- Use an array (if we know ahead of time the upper bound on the number of elements in the heap, N). This is simpler...

Representing a Heap Using an Array

- Store the heap in an array, H indexed by $i = 1, 2, 3, \ldots, N$. We will think of the heap nodes as corresponding to the positions in the array.
- $H[1]$ is the root.
- For any node at position i, the children are the nodes at positions $leftChild(i) = 2i$ and $rightChild(i) = 2i + 1$. For example, the two children of the root are at postiions 2 and 3.
- The parent of a node at position i is at position $parent(i) = \lfloor \frac{i}{2} \rfloor$.
- If the heap has $n < N$ elements at some time, we will use the first n positions of the array to store the n heap elements, and use $length(H)$ to denote the number of elements in H.

Representing a Heap Using an Array



Each node's key is at least as large as its parent's.

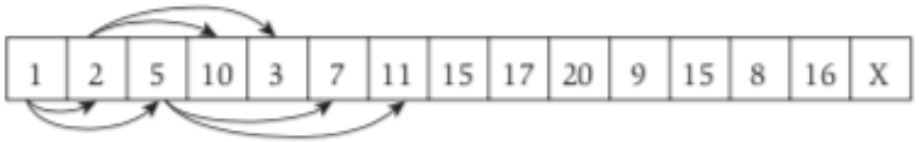


Table of Contents

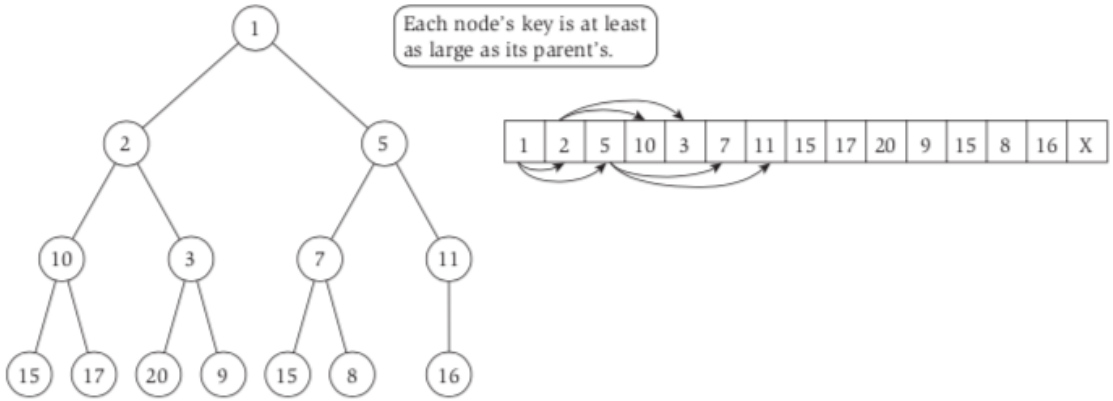
Finding the element with the smallest key value

The element with the smallest key value will always be at the root position. Thus it takes $O(1)$ time to find the minimum-key element.

But what happens if we remove the root element? We need to replace it with another element in the heap. That is, we still have work to do in order to restore the heap to a proper form after removing the root element. So, we're not going to get off *that* easy...!

Let's look at how we add and remove elements so that we can get a better understanding of the complete process that will be involved in removing the minimum element and restoring the heap to a proper state.

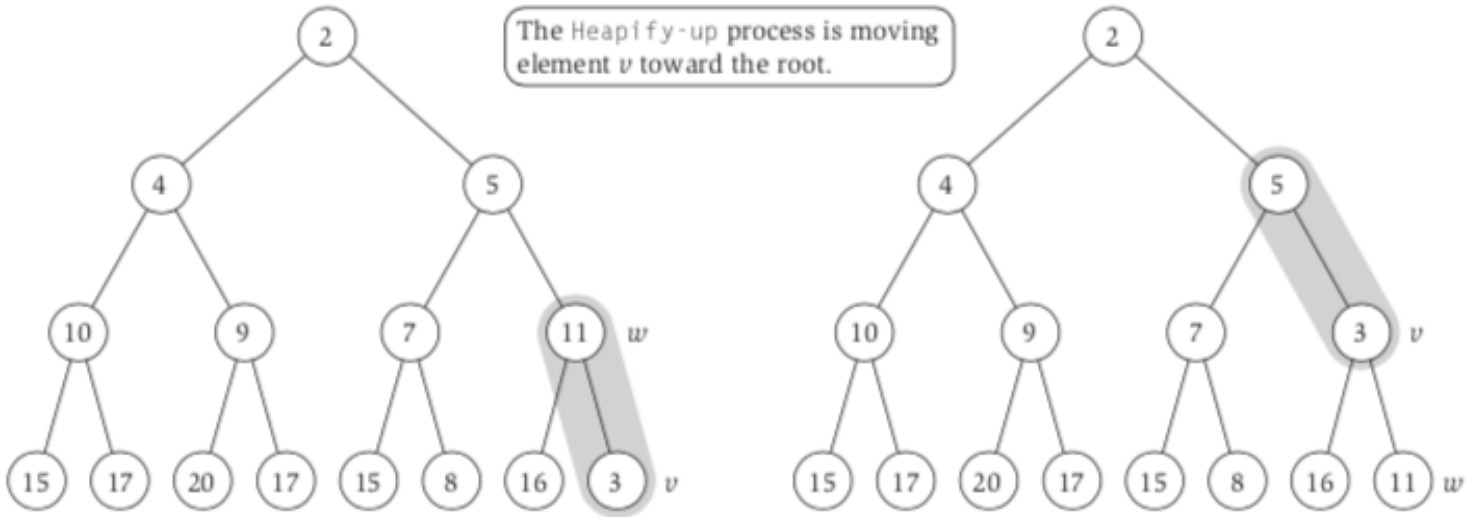
Inserting an element into the heap



We can add a new element v to the final position $i = n + 1$ by setting $H[i] = v$. Unfortunately, this does not maintain the heap property as the key of element v may be smaller than the key of its parent. After this addtion, we have something that *almost* a heap, except for a small "damaged" part where v was pasted on at the end.

Heapify-up

Heapify – up is a procedure to repeatedly swap the child node with a key value in violation of the weak ordering requirement with it's parent until weak ordering is restored.



Heapify-up

```
Heapify-up(H,i):
  If i > 1 then
    let j = parent(i) = ⌊i/2⌋
    If key[H[i]]<key[H[j]] then
      swap the array entries H[i] and H[j]
      Heapify-up(H,j)
    Endif
  Endif
```

Heapify-up - Algorithm Analysis

- Each node swap during the heapify-up procedure can be done in constant time, $\implies f_{swap} = c \implies f_{swap} = O(1)$.
- From the prior diagram, we can see there is (at most) one swap per level, (worst-case).
- How many levels are there in the balanced binary tree? $\lfloor log_2 n \rfloor + 1$.
- Thus, there are $O(log_2 n)$ levels in the balanced binary tree, as a function of the number of elements, n in the tree.
- Runtime: $f_{swap} \cdot (\lfloor log_2 n \rfloor + 1) = c \cdot (\lfloor log_2 n \rfloor + 1)$ $= c \cdot O(logn) = O(logn)$.

Insert an Element - Algorithm Analysis

Inserting an element involves adding the element to the end of the heap, and then running the heapify-up procedure.

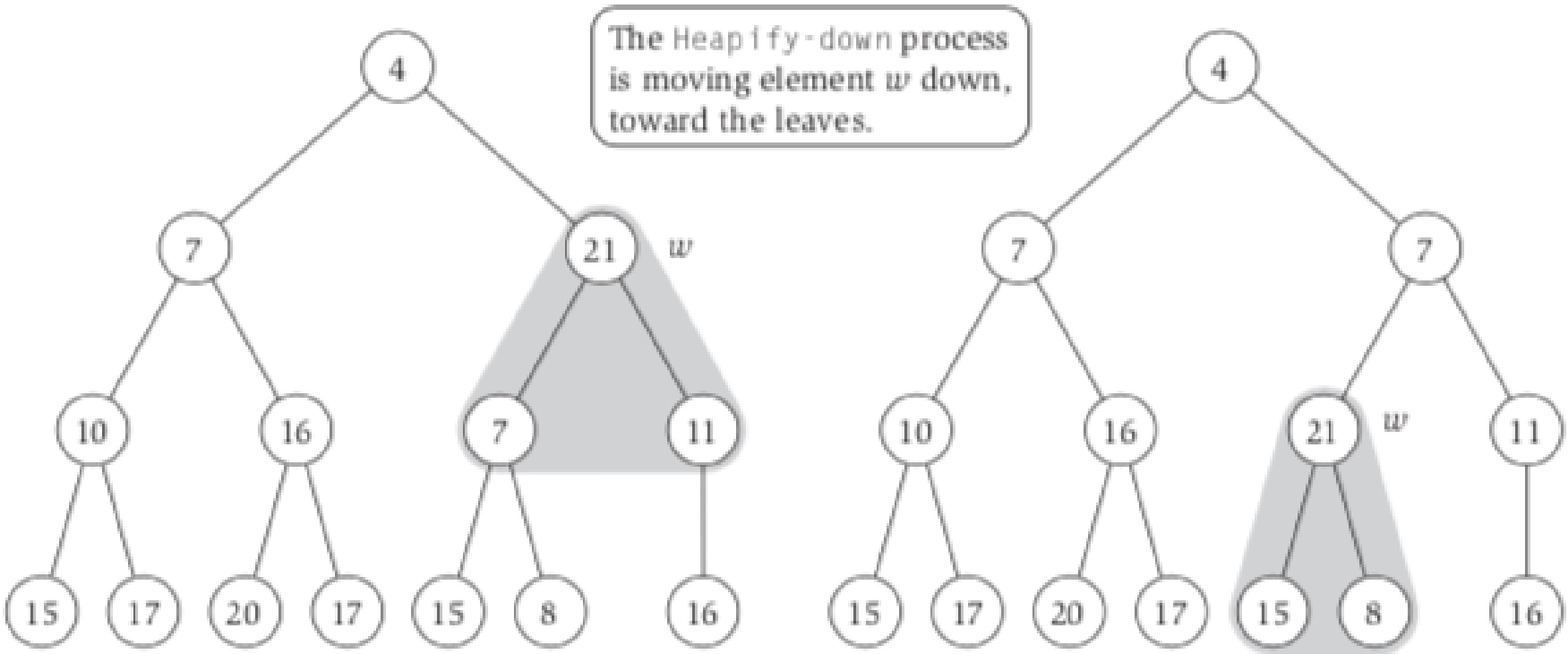
- Add element to the end of the heap: $O(1)$
- Run Heapify-up: $O(logn)$
- Total Runtime: $O(logn)$

Remove the root (min key-value element) from the heap

Procedure:

1. Remove the root element (the element with min key-value).
2. Take the last node in the heap and insert it in the root node position. We may now have a weak ordering violation, as a result.
3. Restore weak ordering in the heap by running the Heapify-down procedure, which is a procedure analagous to heapify-up, except we start with the root and work our way down the heap.

Heapify-down



Heapify-down

```
Heapify-down(H,i):
  Let n = length(H)
  If 2i > n then
    Terminate with H unchanged
  Else if 2i < n then
    Let left = 2i, and right = 2i + 1
    Let j be the index that minimizes key[H[left]] and key[H[right]]
  Else if 2i = n then
    Let j = 2i
  Endif
  If key[H[j]] < key[H[i]] then
     swap the array entries H[i] and H[j]
     Heapify-down(H, j)
  Endif
```

Remove the Root Element - Algorithm Analysis

1. Remove the root element (the element with min key-value). $\Longrightarrow$ $O(1)$.
2. Insert the last node in the heap in the root node position. $\Longrightarrow$ $O(1)$.
3. Restore weak ordering in the heap by running the Heapify-down procedure, $\Longrightarrow$ $O(logn)$ by similar reasoning that we used for analyzing the runtime of Heapify-up.

Runtime for Removing the Root Element: $O(logn)$

Thus, using a heap to implement a priority queue assures (worst-case) performance of at least $O(logn)$ for both addition of a new element and removal of the root (min key-value) element.

Recall that our initial proposed solutions (based on lists/arrays) achieved no better than $O(n)$ performance for one of these insertion/removal operations.

Table of Contents

Priority Queue Implementation

In summary, the core of our implementation of the priority queue consists of a heap (implemented using an array, where the number of elements is bounded by N) along with the Heapify-down and Heapify-up operations. More specifically, we have the following functions:

- $StartHeap(N)$ returns an empty heap, H, that is set to store at most N elements. $\implies$ $O(N)$ as we need to initialize the array of N elements.
- $Insert(H, v)$ insers the item v into the heap, H, using the Heapify-up procedure. $\implies$ $O(logn)$ for a heap with n elements.

Priority Queue Implementation

- *ExtractMin*(H) returns and removes the minimum element with the min-key value from the heap, H, making use of the Heapify-down procedure. $\implies$ $O(logn)$.
- *FindMin*(H) returns but *does not remove* the min-key element. $O(1)$ since this is merely returning a reference to the root node.