EE360C: Algorithms

Priority Queues

Spring 2018

Department of Electrical and Computer Engineering
University of Texas at Austin

Motivation

Motivation: Stable Marriage

The stable marriage algorithm needs a data structure that maintains the dynamically changing set of all free men. The algorithm needs to be able to:

- add elements to the set
- delete elements from the set
- select an element from the set, based on some assigned *priority*

Motivation: Sort a List of Numbers

Sort

Instance: Nonempty list $x_1, x_2, \ldots, x_n$ of integers

Solution: A permutation $y_1, y_2, \ldots y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$ for all $1 \leq i < n$

Possible Algorithm

- Store all of the numbers in a data structure D
- Repeatedly find the smallest number in D, output it, and remove it

To get $O(n \log n)$ running time, each "find minimum" step must take $O(\log n)$ time

Candidate Data Structures for Sorting

The data structure we select must support inserting a new element, finding the minimum element, and deleting the minimum element.

- **List:** Insertion and deletion take $O(1)$ time, but finding the minimum requires scanning the list and takes $\Omega(n)$ time
- **Sorted array:** Finding the minimum takes $O(1)$ time, but insertion and deletion take $\Omega(n)$ time in the worst case

Priority Queue

Enter the Priority Queue

- Store a set S of elements, where each element v has a priority value $\texttt{key}(v)$
- Smaller key values denote higher priorities
- Operations supported:
  - find the element with the smallest key
  - remove the element with the smallest key
  - insert a new element
  - delete an element
- Key update and element deletion require knowledge of the position of the element in the priority queue

An Example Application

Consider the problem of real-time scheduling of processes on a computer

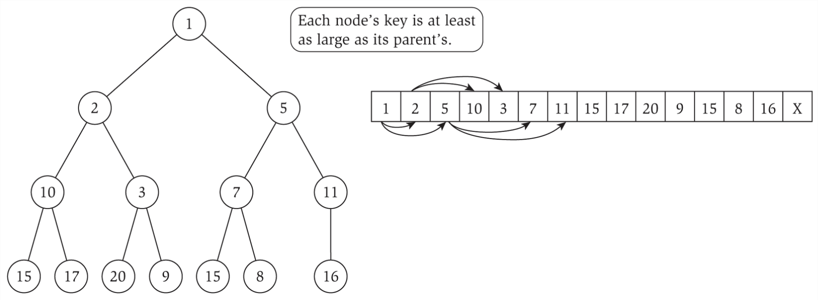
- each process has a priority
- processes *do not* arrive in order of their priorities
- we need to maintain a set of *active processes* with the ability to quickly extract the one with the highest priority so it can be scheduled
- using a priority queue keyed by process priority, scheduling the highest priority process entails simply finding the one with the lowest priority key

Heaps

Heaps

- Combine the benefits of both lists and sorted arrays
- Conceptually, a heap is a balanced binary tree
- Heap order: For every element v at node i, the element w at i's parent satisfies $\texttt{key}(w) \leq \texttt{key}(v)$
- We can implement a heap in a pointer-based data structure
- Alternatively, assume a maximum number N of elements is known in advance
- Store nodes of the heap in an array
    - Node at index i has children at indices $2i$ and $2i + 1$ and parent at index $\lfloor i/2 \rfloor$
    - Index 1 is the root
    - How do you know that a node at index i is a leaf? If $2i > n$, the number of elements in the heap.

A Heap Example

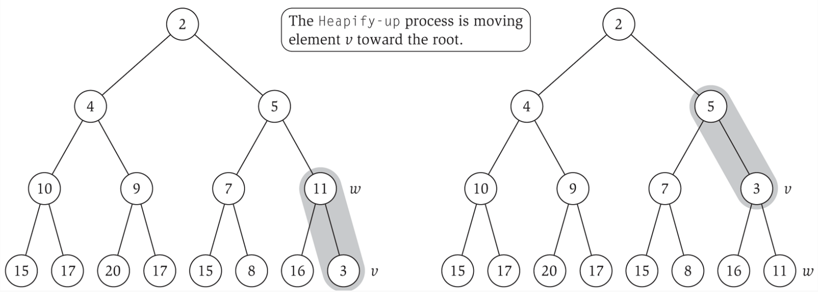


Inserting an Element: `Heapify-up`

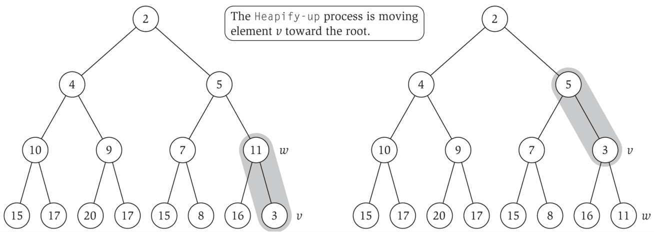
1. Insert a new element at $n+1$
2. Fix the heap order using `Heapify-up`$(H, n+1)$

```
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 **Example**



Correctness of Heapify-Up



- H is almost a heap with key of $H[i]$ too small if there is a value $\alpha \geq$key$(H[i])$ such that increasing key$(H[i])$ to α makes H a heap
- **Claim:** The procedure Heapify-Up(H, i) fixes the heap property in $O(\log n)$ time, assuming that the array H is almost a heap with the key of $H[i]$ too small.
- **Corollary:** Using Heapify-Up we can insert a new element in a heap of n elements in $O(\log n)$ time. (Why?)

Correctness of `Heapify-Up`

- H is almost a heap with key of $H[i]$ too small if there is a value $\alpha \geq$`key`$(H[i])$ such that increasing `key`$(H[i])$ to α makes H a heap
- **Claim:** The procedure `Heapify-Up`(H, i) fixes the heap property in $O(\log n)$ time, assuming that the array H is almost a heap with the key of $H[i]$ too small.
- Prove by induction on i:
  - Base case: $i = 1$. H[1] is the root, so if it's too small, then H is already a heap.
  - Inductive step: H is almost a heap with key of $H[i]$ too small. Let $j =$`parent`(i) and β be its key. Swapping the elements at $H[i]$ and $H[j]$ takes $O(1)$ time. After the swap, H is a heap or almost a heap with the key of $H[j]$ too small, since setting its key to β would make H a heap. Finally, by the inductive hypothesis, the recursive call to `Heapify-Up` fixes the heap property.

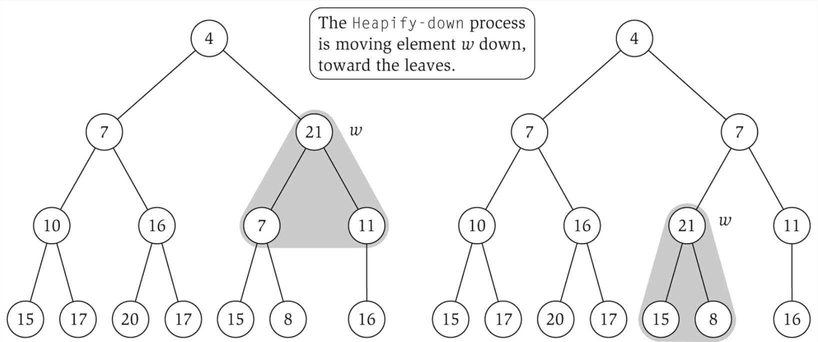
Deleting an Element: `Heapify-down`

Suppose H has $n+1$ elements

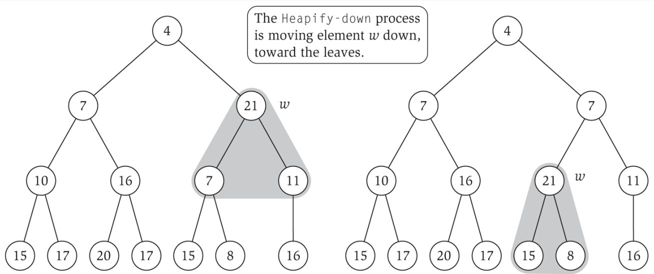
1. Delete element at $H[i]$ by moving element at $H[n+1]$ to $H[i]$
2. If element at $H[i]$ is too small, fix heap order using `Heapify-up`(H, i)
3. If element at $H[i]$ is too large, fix heap order using `Heapify-down`(H, i)

```
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)
```

`Heapify-down` **Example**



`Heapify-down` **Correctness**



- H is almost a heap with key of $H[i]$ too big if there is a value $\alpha \leq$`key`$(H[i])$ s.t. decreasing `key`$(H[i])$ to α makes H a heap
- **Claim:** The procedure `Heapify-Down`(H, i) fixes the heap property in $O(\log n)$ time, assuming that the array H is almost a heap with the key of $H[i]$ too big.
- **Corollary:** Using `Heapify-Down` we can delete an element from a heap of n elements in $O(\log n)$ time. (Why?)

`Heapify-down` **Correctness**

- H is almost a heap with key of $H[i]$ too big if there is a value $\alpha \leq$`key`$(H[i])$ such that decreasing `key`$(H[i])$ to α makes H a heap
- **Claim:** The procedure `Heapify-Down`(H, i) fixes the heap property in $O(\log n)$ time, assuming that the array H is almost a heap with the key of $H[i]$ too big.
- Proof by reverse induction on i. Suppose H has n elements.
  - Base case: $2i > n$. Then i is a leaf, hence H is a heap.
  - Inductive step: Let j be the child of i with smaller key value and denote its key value β. Swapping the elements at $H[i]$ and $H[j]$ takes $O(1)$ time. The resulting array is a heap or almost a heap with $H[j]$ too big, since setting its key to β makes it a heap. Since $j \geq 2i$, by the inductive hypothesis, the recursive call to `Heapify-Down` fixes the heap property.

In Class Exercise 1

Problem

Naively, we can build a heap out of an arbitrary array using successive calls to HEAPIFY-DOWN, starting at element $\lfloor \text{length}[H]/2 \rfloor$ and going down to 1. If each call to HEAPIFY-DOWN takes $O(\log n)$ time and we have $O(n/2)$ such calls, we can build a heap in $O(n \log n)$ time. Prove that this process is actually faster than $O(n \log n)$ (i.e., provide a *tighter* bound on the process's running time). Starters:

- What is the height of an n-element heap?
- How many nodes are there at height h of an n-element heap?

In Class Exercise 1: continued

What is the height of an n-element heap?

$O(\log n)$ (it's a (nearly) complete binary tree).

In Class Exercise 1: continued

How many nodes are there at height h of an n-element heap?

Key Observation

The number of leaves in a complete binary tree is $\lceil n/2 \rceil$.

Proposition

In an n-element heap, there are $\lceil n/2^{h+1} \rceil$ nodes at height h.

Proof (by induction on h)

Base case: $h = 0$ (the leaves). This is trivially true from the observation above.

Inductive step: Suppose that the claim is true for $h - 1$. Let N_h be the number of nodes at height h in an n-node tree T. Consider T' formed by removing the leaves of T. T' has $n' = n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$ nodes. Nodes at height h in T are at height $h - 1$ in T' (because T' is missing the bottom level of T). Let N'_{h-1} denote the number of nodes at height $h - 1$ in T'.
$N_h = N'_{h-1} = \lceil n'/2^h \rceil = \lceil \lfloor n/2 \rfloor / 2^h \rceil \leq \lceil (n/2)/2^h \rceil = \lceil n/2^{h+1} \rceil$.

In Class Exercise 1: continued

Problem

Naively, we can build a heap out of an arbitrary array using successive calls to HEAPIFY-DOWN, starting at element $\lfloor \text{length}[H]/2 \rfloor$ and going down to 1. If each call to HEAPIFY-DOWN takes $O(\log n)$ time and we have $O(n/2)$ such calls, we can build a heap in $O(n \log n)$ time. Prove that this process is actually faster than $O(n \log n)$ (i.e., provide a *tighter* bound on the process's running time). Starters:

- What is the height of an n-element heap? $O(\log n)$
- How many nodes are there at height h of an n-element heap? $\lceil n/2^{h+1} \rceil$

Problem

Naively, we can build a heap out of an arbitrary array using successive calls to HEAPIFY-DOWN, starting at element $\lfloor \text{length}[H]/2 \rfloor$ and going down to 1. If each call to HEAPIFY-DOWN takes $O(\log n)$ time and we have $O(n/2)$ such calls, we can build a heap in $O(n \log n)$ time. Prove that this process is actually faster than $O(n \log n)$ (i.e., provide a *tighter* bound on the process's running time).

Solution

The time required by HEAPIFY-DOWN, when called on a node at height h is $O(h)$. The total cost of building a heap is bounded above by:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O(n \sum_{h=1}^{\lfloor \log n \rfloor} \frac{h}{2^h}) = O(n)$$

The last step is because (looking up the summation):

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

HeapSort

Sorting with a Priority Queue

Sort

Instance: Nonempty list $x_1, x_2, \ldots, x_n$ of integers

Solution: A permutation $y_1, y_2, \ldots y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$ for all $1 \leq i < n$

Final Algorithm

- Insert each number in a priority queue H
- Repeatedly find the smallest number in H, output it, and delete it from H

Each insertion and deletion takes $O(\log n)$ time for a total running time of $O(n \log n)$

In Class Exercise 2

Problem

One of your classmates claims that he built an alternative data structure (other than a heap) for representing a priority queue. He claims that, using his new data structure, INSERT, MAX, and EXTRACTMAX all take constant ($O(1)$) time in the worst case. Give a very simple proof that he is mistaken.

Solution

If this were true, we could comparison sort in $O(n)$ time. But we've already proven that this is not possible.

Questions