Heaps and Heapsort

David Fernández-Baca

Computer Science 311
Iowa State University

February 1, 2022

Priority Queues

Priority Queues

Definition

A priority queue supports the following operations on a set S.

`Insert`(S, x): Inserts the element x into the set S.

`Maximum`(S): Returns the element of S with the largest key.

`ExtractMax`(S): Removes and returns the element of S with the largest key.

`IncreaseKey`(S, x, k): Increases the value of x's key to k. Assumes $k \geq x.\texttt{key}$.

Implementing Priority Queues as (Binary) Heaps

A heap implementation of an n-element priority queue achieves the following running times.

`Insert`	$O(\log n)$
`Maximum`	$O(1)$
`ExtractMax`	$O(\log n)$
`IncreaseKey`	$O(\log n)$

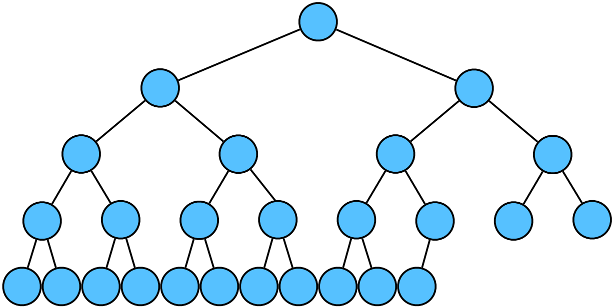
Additionally,

- an n-element heap can be constructed in $O(n)$ time,
- heaps can be used to obtain a $O(n \log n)$ sorting algorithm: `Heapsort`.

Heaps

Heap Shape

A binary tree T has heap shape if T is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.



Storing a Heap as an Array

- Use an array A with two attributes:

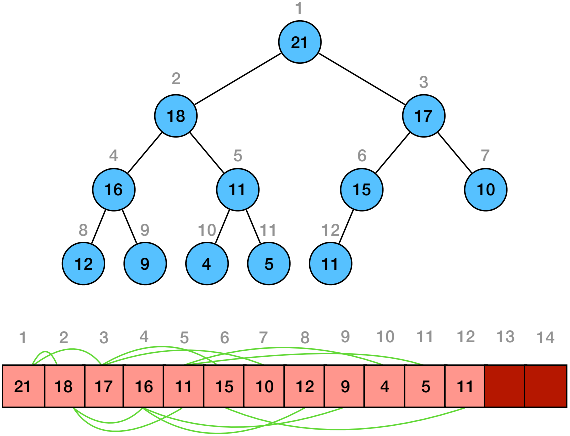
  $A.\texttt{length}$: the number of elements in the array.

  $A.\texttt{size}$: The number of heap elements stored within A.

  - Only the elements in $A[1], \ldots, A[A.\texttt{size}]$ are valid elements of the heap.

- Number nodes consecutively starting with 1 for the root, going level by level from top to bottom and left to right within each level.
- Put node i in $A[i]$.
- Given the index i of a node, it is easy to compute the indices of its parent, left child, and right child.

Storing a Heap as an Array



A.length $= 14$ A.size $= 12$

Parent(i)
  return $\lfloor i/2 \rfloor$

Left(i)
  return $2i$

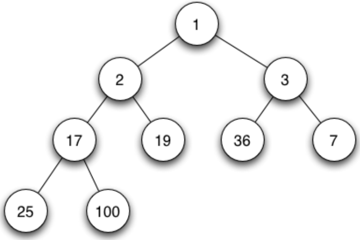
Right(i)
  return $2i + 1$

Min Heaps (not the focus for now)

Definition

In a min-heap, for every node i other than the root,

$$A[\texttt{Parent}(i)] \leq A[i].$$



Source: Wikipedia

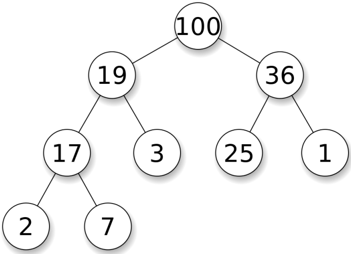
The smallest element in a min-heap is at the root.

Max Heaps

Definition

In a max-heap, for every node i other than the root,

$$A[\texttt{Parent}(i)] \geq A[i].$$



Source: Wikipedia

The largest element in a max-heap is stored at the root.

```
Maximum(A)
  return A[1]
```

Fact

`Maximum` *runs in $O(1)$ time, regardless of the size of the heap.*

Fact

An n-element heap has

- *height $\lfloor \log n \rfloor$ and*
- *at most $\lceil n/2^{h+1} \rceil$ nodes of any height h.*

Maintaining a Heap

Heapify

- Assume we are dealing with max-heaps.
- `Heapify` takes as input an array A and an index i into A.
- `Heapify` assumes that the trees rooted at `Left`(i) and `Right`(i) are heaps.
- $A[i]$ might be smaller than its children, thus violating the heap property.
- `Heapify` percolates the value at $A[i]$ down the heap so that the subtree rooted at index i obeys the heap property.

```
Heapify(A, i)
    l = Left(i)
    r = Right(i)
    if l ≤ A.size and A[l] > A[i] then
        largest = l
    else
        largest = i
    if r ≤ A.size and A[r] > A[largest] then
        largest = r
    if largest ≠ i then
        exchange A[i] with A[largest]
        Heapify(A, largest)
```

Theorem

The running time of `Heapify` *on a node of height* h *is* $O(h)$.

Building a Heap

```
BuildHeap(A)
    A.size = A.length
    for i = ⌊A.length/2⌋ downto 1 do
        Heapify(A, i)
```

Note

- Let $n = A.\texttt{length}$.
- Each of $A[\lfloor n/2 \rfloor + 1], \ldots, A[n]$ is a leaf.
- Thus, each of $A[\lfloor n/2 \rfloor + 1], \ldots, A[n]$ is initially a 1-element heap.

Theorem

`BuildHeap` *converts an array A into a heap.*

Proof.

`BuildHeap` maintains the following.

Loop Invariant

At the start of iteration i of the **for** loop of `Heapify`, each of nodes $i+1, i+2, \ldots, n$ is the root of a heap.

$\square$

Theorem

`BuildHeap` *rearranges an n-element into a heap in $O(n)$ time.*

Proof.

- Recall that an n-element heap has height $\lfloor \log n \rfloor$ and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h.
- `Heapify` takes $O(h)$ time when called on a node of height h.
- Thus, the total cost of `BuildHeap` is

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

- Next, we show that the sum is $O(n)$.

□

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

$$\begin{aligned}
\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot h &\leq n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \\
&\leq n \sum_{h=0}^{\infty} \frac{h}{2^h} \\
&= 2n, \qquad \text{since } \sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}, \text{ for } |x| < 1, \\
&= O(n).
\end{aligned}$$

Implementing Priority Queues Using Heaps

```
ExtractMax(A)
    if A.size < 1 then
        error "underflow"
    max = A[1]
    A[1] = A[A.size]
    A.size = A.size − 1
    Heapify(A, 1)
    return max
```

Fact

`ExtractMax` *runs in* $O(\log n)$ *time on an* n*-element heap.*

```
IncreaseKey(A, i, key)
    if key < A[i] then
        error "new key is smaller than current key"
    A[i] = key
    while i > 1 and A[Parent(i)] < A[i] do
        exchange A[i] with A[Parent(i)]
        i = Parent(i)
```

Fact

`IncreaseKey` *runs in* $O(\log n)$ *time on an* n*-element heap.*

```
Insert(A, key)
    A.size = A.size + 1
    A[A.size] = -∞
    IncreaseKey(A, A.size, key)
```

Fact

`Insert` *runs in* $O(\log n)$ *time on an* n*-element heap.*

Heapsort

Heapsort

To sort an n element array A, call `BuildHeap`(A) to convert A into a max-heap, and then repeat the following $n-1$ times.

- The maximum element is $A[1]$, so we put it into its correct final position by exchanging $A[1]$ with $A[n]$.
- Discard node n by decrementing A.`size` .
- The children of the root remain max-heaps, but the new root element might violate the max-heap property.
- To restore the heap property, call `Heapify`$(A, 1)$, which leaves a heap in $A[1], \ldots, A[n-1]$.

```
Heapsort(A)
    BuildHeap(A)
    for i = A.length downto 2 do
        exchange A[1] with A[i]
        A.size = A.size − 1
        Heapify(A, 1)
```

Correctness of Heapsort

Loop Invariant

At the start of each iteration of the **for** loop,

- subarray $A[1], \ldots, A[i]$ is a max-heap containing the i smallest elements of $A[1], \ldots, A[n]$, and
- subarray $A[i+1], \ldots, A[n]$ contains the $n-i$ largest elements of $A[1], \ldots, A[n]$, sorted.

Theorem

`Heapsort` *takes time* $O(n \log n)$ *to sort an* n*-element array.*

Proof.

`BuildHeap` takes time $O(n)$ and each of the $n - 1$ calls to `Heapify` takes time $O(\log n)$. $\square$

Bibliography

Reference

[CLRS] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms* (3rd edition), MIT Press, 2009.