IT501: Algorithms

Heapify

Anish Mathuria
Faculty Block 1, 1105
Recap

• Heap: Almost complete binary tree
  – filled on all levels, except last, where filled from left to right

• Min-heap ordered
  – every child greater than (or equal to) parent

• Max-heap ordered
  – every child less than (or equal to) parent
Exercise

Consider a binary max-heap of size 751 whose elements are all distinct integers. The largest element in the heap must appear in position 1.

a) List all the positions where the smallest element of the above max-heap can appear.

b) List all the positions where the smallest element of the above max-heap cannot appear.
Heap operations

• With a heap, we can perform at least these operations efficiently
  – Insert a new element
  – Find the largest element
  – Remove the largest element
Remove the largest element

- The element to be removed is in the root. Removing it leaves the root empty.
- The only node we can delete from the tree, and still have a nearly complete tree, is the last node, i.e. node $p$ (4).
Remove the largest element

- So we move the element in node $p$ (4) to the root (node $q$), and remove node $p$ from the tree.
- We still have a nearly complete binary tree, and the heap property can fail only at the children of the root (nodes $r$ and $s$).
Remove the largest element

- Given a nearly complete binary tree, in which the heap property can fail only at the children of the root, we can make the tree into a heap using a procedure called $max$-$heapify()$.
Remove the largest element

- Among the root and its two children (nodes $q$, $r$, $s$), we find the largest element. (Two comparisons will suffice.)
  - In this case, the largest (12) occurs in node $r$

- Two possibilities
  - If the largest of these three elements were to occur in the root (not the case here), we would be done
  - If the largest occurs in a child of the root (as happens here, node $r$), we exchange the element in the root with the element in this child
    - In our case we exchange 4 and 12

Heap property could fail at $r$ and $s$
Remove the largest element

- This guarantees that the heap property holds at both children of the root, but may cause it to fail at the children of the node exchanged with the root (the children of node $r$, in our case)

- We apply the same process recursively to the subtree rooted at $r$, i.e., invoke `max-heapify()` recursively

Heap property must hold at $r$ and $s$, but could fail at the children of $r$
Algorithm max-heapify(A, i, n):
{ Create a max-heap rooted at A[i]. It is assumed that the trees rooted
 at A[2i] and A[2i+1] are max-heaps. A[i] may be smaller than its
 children. }

largest := i
if (2i ≤ n and A[2i] > A[i])
    largest := 2i
if (2i + 1 ≤ n and A[2i+1] > A[largest])
    largest := 2i + 1
if (largest ≠ i)
    then
        swap(A[i], A[largest])
        max-heapify(A, largest, n)
Max-heapify – iterative

Algorithm max-heapify(A, i, n):
while (2i ≤ n)
    largest := i
    if (A[2i] > A[i])
        then
            largest := 2i
    if (2i + 1 ≤ n and A[2i+1] > A[largest])
        then
            largest := 2i + 1
    if (largest ≠ i)
        then
            swap(A[i], A[largest])
            i := largest
    else
        i := n+1
Cost of max-heapify(A, 1, n)

- At most 2 comparisons per level
- If height of heap is $m$, then no more than $2m$ comparisons are required
- For a $n$-node heap, $m = \lfloor \log n \rfloor$
- So the worst number of comparisons is $2 \lfloor \log n \rfloor$
Heapify() Example

A = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]
Heapify() Example

A = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]
Heapify() Example

A =  

\[
\begin{array}{cccccccc}
16 & 4 & 10 & 14 & 7 & 9 & 3 & 2 & 8 & 1
\end{array}
\]
Heapify() Example

A = [16, 14, 10, 4, 7, 9, 3, 2, 8, 1]
Heapify() Example

A = 16 14 10 4 7 9 3 2 8 1
Heapify() Example

A = [16, 14, 10, 4, 7, 9, 3, 2, 8, 1]
Heapify() Example

A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
Heapify() Example

A = 16 14 10 8 7 9 3 2 4 1
Heapify() Example

A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
Deletion algorithm

Algorithm heap-extract-max(A, n):
{ Removes the minimum element in a heap of size n represented in an array A. }

max := A[1]
n := n – 1
max-heapify(A, 1, n)
return max
Cost of extract-max

- Heapify is invoked after the size of heap is decreased by 1
- So the worst case number of comparisons is:
  \[ 2 \lceil \log (n - 1) \rceil \]
Heaps

• Other operations which can be performed efficiently
  – Increase or decrease the element in a known position
  – Remove the element in a known position
Heaps

• Other operations which cannot be performed efficiently
  – Given $x$, decide if the heap contains an element equal to $x$
  – Given $k$, find the $k^{th}$ largest element in the heap (unless $k$ is 1, or at least is very close to 1)
  – Given $x$, remove $x$ from the heap, if it is present
Could a heap be useful for sorting?

• What will be true of the values if we remove all the elements of a heap using extract Max?
• How to create a heap?
• What do we do if we want a descending sort?
Could a heap be useful for sorting?

- Do we need a second array to hold sorted elements after removing them from the heap?
- No! Put them back in the same array (in-place sorting)
Sorting a heap

• We know that the maximum value is at A[1] in a heap
• We put that value at the end of the array, by swapping it with the last element of the heap
• We then reduce the size of the heap by 1, so that the sorted element is no longer part of the heap.
• Heapify the new, smaller heap (A[1] is not necessarily in the correct position for a heap). This will put a new maximum at the top
• Repeat this process until all the nodes have been sorted
Sorting a heap

1. **Heapify**

   - Swap: x and y
   - New heap: [x, y, ..., i, ..., n]
   - Sorted: [x, i, ..., n]

2. **Heapify**

   - New heap: [x, y, ..., i, ..., n]
   - Sorted: [x, i, ..., n]