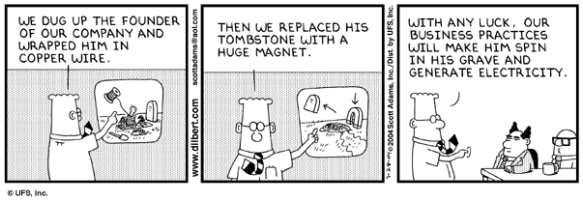


CSE 502N
Fundamentals of Computer Science

Fall 2004

Lecture 11:

Binary Heaps & Heapsort

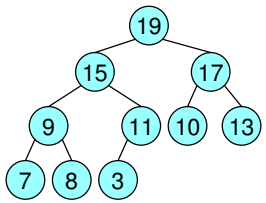
Priority Queues

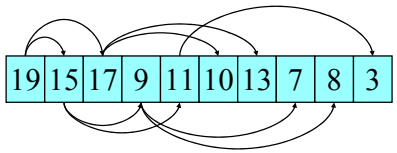
(CLRS 6)

Washington University in St.Louis

Binary heaps

- A binary heap is an array-based data structure that may be viewed as a (nearly) complete binary tree
  - » Each node corresponds to an array element
  - » Tree is filled on all levels except possibly the lowest level which is filled from left to right
- An array A that represents a heap has two attributes:
  - » *length*[A] = the number of elements in the array
  - » *heap-size*[A] = the number of elements in the heap





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

Left(i) **return** $2i$

Right(i) **return** $2i+1$

Heap properties

- A ***max-heap*** is a binary heap that maintains the ***max-heap property***: $A[\text{Parent}(i)] \geq A[i]$
  - » Largest element is stored at the root
- A ***min-heap*** is a binary heap that maintains the ***min-heap property***: $A[\text{Parent}(i)] \leq A[i]$
  - » Smallest element is stored at the root
- The ***height*** of a node is the number of edges on the longest simple downward path from the node to a leaf
- The height of the heap is the height of the root node
  - » Since the heap is a nearly complete binary tree, its height is $\Theta(\lg n)$

Maintaining the heap property

- Max-Heapify assumes the binary trees rooted at Right(i) and Left(i) are max-heaps, but A[i] may be smaller than its children
- What is the running time?
- T(n) = (time to adjust i, Left(i), Right(i)) + (time to run Max-Heapify on one of children)
  - » Maximum size of a child subtree is $2n/3$ (occurs when lowest level is half full)
- $T(n) \le T(2n/3) + \Theta(1)$
- $T(n) = \Theta(\lg n)$

```
MAX-HEAPIFY(A, i)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l ≤ heap-size[A] and A[l] > A[i]
4     then largest ← l
5     else largest ← i
6  if r ≤ heap-size[A] and A[r] > A[largest]
7     then largest ← r
8  if largest ≠ i
9     then exchange A[i] ↔ A[largest]
10         MAX-HEAPIFY(A, largest)
```


Building a heap

- Use Max-Heapify in a bottom-up manner to convert an array into a max-heap
  - » Elements A[($\lfloor n/2 \rfloor + 1$) … n] are all leaves of the tree

```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3      do MAX-HEAPIFY(A, i)
```

- Loop invariant: at the start of each iteration of the for loop of lines 2-3, each node i+1,i+2,…,n is the root of a max-heap
- Initialization: prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$
  - » Each node $\lfloor n/2 \rfloor$+1, $\lfloor n/2 \rfloor$+2,…,n is a leaf and is thus the root of a trivial max-heap
- Maintenance: by the loop invariant Left(i) and Right(i) are both roots of max-heaps
  - » This is the condition required for Max-Heapify to make node i a max-heap root
  - » Max-Heapify preserves property that nodes i,i+1,…,n are max-heap roots
  - » Following Max-Heapify, decrementing i reestablishes the loop invariant
- Termination: at termination, i=0; by the loop invariant each node 1,2,…,n is a max-heap root
  - » Thus, node 1 is a max-heap root

Running time of Build-Max-Heap

- Easy to derive an $O(n \lg n)$ bound
  - » Upper bound for Max-Heapify is $O(\lg n)$
  - » Build-Max-Heap calls Max-Heapify $\lfloor n/2 \rfloor$ times
- Can we derive a tighter bound?
- Time to execute Max-Heapify varies with the height of the node in the tree
  - » Execution time for node of height h is O(h)
- An n-element heap has height $\lfloor \lg n \rfloor$
- An n-element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of any height h

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

- We can show that Build-Max-Heap is $O(n)$

Heapsort

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← length[A] downto 2
3      do exchange A[1] ↔ A[i]
4         heap-size[A] ← heap-size[A] − 1
5         MAX-HEAPIFY(A, 1)
```

- Heapsort sorts array A in place, back to front (max to min)
- At the completion of each iteration of Max-Heapify, A[1] is the maximum element left in the heap
- Each iteration of the **for** loop places the maximum element in its sorted position and reduces the heap-size (removes it from the heap), then runs Max-Heapify to find the maximum element left in the heap
- $T(n) = O(n) + (n\text{-}1)O(\lg n) = O(n \lg n)$

Priority queues

- A ***priority queue*** is a data structure for maintaining a set of S elements with associated *key* values
- Elements may be inserted into a ***priority queue*** at any time
- In ***max-priority queues*** only the maximum element may be removed from the queue
  - » The ***increase-key*** operation may be used to increase the value of an element's key and possibly change the relative ordering of elements
  - » ***Min-priority queues*** implement symmetric operations
- Priority queues are useful for a multitude of scheduling tasks where the highest priority element/job/packet should be scheduled first
- Max-priority queues leverage the constant time Heap-Maximum operation

```
HEAP-MAXIMUM(A)
1  return A[1]
```


Extract-Max

- For priority queues implemented with max-heaps, Heap-Extract-Max can be used to remove the maximum element
- T(n) = O(lg n)

```
HEAP-EXTRACT-MAX(A)
1  if heap-size[A] < 1                          } O(1)
2     then error "heap underflow"
3  max ← A[1]
4  A[1] ← A[heap-size[A]]
5  heap-size[A] ← heap-size[A] − 1
6  MAX-HEAPIFY(A, 1)                            } O(lg n)
7  return max
```


Increase-Key

- Given the index i of an element and a new *key* value (greater than the current *key* value), Heap-Increase-Key increases the key value of the element and ensures max-heap property holds
- After changing key value, Heap-Increase-Key traverses a path from $A[i]$ toward the root
  - » Compares keys of an element and its parent and exchanges if the element's key is greater than the parent's key
- T(n) = O(lg n)

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2     then error "new key is smaller than current key"
3  A[i] ← key
4  while i > 1 and A[PARENT(i)] < A[i]
5        do exchange A[i] ↔ A[PARENT(i)]
6           i ← PARENT(i)
```


Insert

- Given a new key value, Max-Heap-Insert creates a new heap element with the give key value and ensures max-heap property is maintained
- Increases heap-size by 1
- Assigns new element the minimum key value
- Calls Heap-Increase-Key using new element's index and new key value
- T(n) = O(lg n)

```
MAX-HEAP-INSERT(A, key)
1  heap-size[A] ← heap-size[A] + 1
2  A[heap-size[A]] ← −∞
3  HEAP-INCREASE-KEY(A, heap-size[A], key)
```