Lecture 12

Algorithms & Data Structures

Goldsmiths Computing

January 21, 2019

Outline

Outline

Lecture

- Implicit data structures
  1. multidimensional arrays
  2. binary search trees
- Binary search on arrays

VLE activities

Hashing quiz

Statistics so far:

- A attempts: average mark B
- C students: average mark D
  - E under 4.00, F over 6.99, G at 10.00

Quiz closes at 16:00 on Friday 19th January

- no extensions
- grade is
  - 0 (for no attempt)
  - $30 + 70 \times (\text{score}/10)^2$

Outline

Introduction

Heaps

Implicit heaps

Insert

Require: heap :: Heap
  function INSERT!(heap,object)
    s $\leftarrow$ NEXT(heap)
    p $\leftarrow$ PARENT(s)
    while p $\neq$ NIL $\wedge$ p.key $<$ object **do**
      s.key $\leftarrow$ p.key
      s $\leftarrow$ p; p $\leftarrow$ PARENT(p)
    end while
    s.key $\leftarrow$ object
  end function

Constructing a heap incrementally

```
function MAKE-HEAP(S)
    H ← new Heap()
    for 0 ≤ i < LENGTH(S) do
        INSERT!(H,S[i])
    end for
    return H
end function
```

Complexity analysis

to build a heap with N elements, incrementally:

- each incremental addition takes $\Omega(h)$ time (h is the *current* height of the tree)
- in the worst case, there are $\frac{N}{2}$ nodes with height $\log(N)$

$$\Longrightarrow \Omega(N \log(N)), \text{ and in fact } \Theta(N \log(N)))$$

Outline

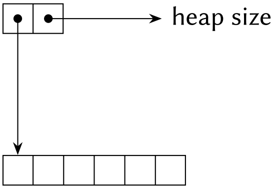
Implicit representation

implicit representations, previously:

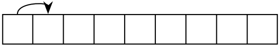
- dope vector (multidimensional array)
- sorted sequence (binary search tree)
- partially-sorted sequence (insertion sort)

Implicit heap

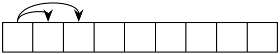
- an array
- a heap size (must be ≤ array length)



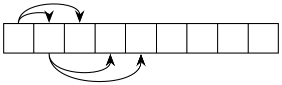
Parents and children



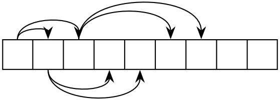
Parents and children



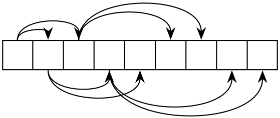
Parents and children



Parents and children



Parents and children



Parents and children

For zero-based arrays

```
function LEFT(i)
    return 2×i+1
end function
function RIGHT(i)
    return 2×i+2
end function
function PARENT(i)
    return ⌊(i−1)/2⌋
end function
```

(one-based arrays have simpler calculations, but generalise less well)

Heapify

Given a root with two (max-)heaps as children, make the root be a valid max heap.

function MAX-HEAPIFY(a,i)
    l ← LEFT(i)
    r ← RIGHT(i)
    largest ← i
    if l < a.heapsize ∧ a[l] > a[largest] **then**
        largest ← l
    end if
    if r < a.heapsize ∧ a[r] > a[largest] **then**
        largest ← r
    end if
    if largest ≠ i **then**
        SWAP(a[i],a[largest])
        MAX-HEAPIFY(a,largest)
    end if
end function

(Also called siftDown)

Complexity analysis

Time complexity

$$T(N) \leq T\left(\frac{2N}{3}\right) + \Theta(1)$$

$$\Longrightarrow \Theta(\log(N)) \text{ or } \Theta(h)$$

Constructing a heap in one go

Half of the nodes are already heaps!

function BUILD-MAX-HEAP(a)
    a.heapsize $\leftarrow$ a.length
    for $\left\lfloor \frac{\text{a.length}}{2} \right\rfloor < j \leq 0$ **do**
        MAX-HEAPIFY(a,j)
    end for
end function

Complexity analysis

First analysis

- $\frac{N}{2}$ calls to MAX-HEAPIFY
- each takes time $O(\log(N))$

$$\implies O(N \log(N))$$

Complexity analysis

First analysis

- $\frac{N}{2}$ calls to MAX-HEAPIFY
- each takes time $O(\log(N))$

$$\implies O(N \log(N))$$

Improved bound

- most calls to MAX-HEAPIFY are near the leaves
- height of most trees is small

$$T(h) \leq O\left(1 \times \frac{N}{2} + 2 \times \frac{N}{2^2} + 3 \times \frac{N}{2^3} + \ldots + h \times \frac{N}{2^h}\right)$$

But $\sum_{k=0}^{\infty} \frac{k}{2^k} = 2$ (proof?)

$$\implies O(N)$$

Operations

insert!

function INSERT!(heap,k)
  heap[heap.heapsize] ← k
  i ← heap.heapsize
  heap.heapsize ← heap.heapsize + 1
  while i > 0 ∧ heap[PARENT(i)] < heap[i] **do**
    SWAP(heap[i],heap[PARENT(i)])
    i ← PARENT(i)
  end while
end function

Operations

extract-max!

```
function EXTRACT-MAX!(heap)
    max ← heap[0]
    heap[0] ← heap[heap.heapsize-1]
    heap.heapsize ← heap.heapsize - 1
    MAX-HEAPIFY(heap,0)
    return max
end function
```

Complexity analysis

insert!

- at most h calls to SWAP

$$\implies \Theta(\log(N))$$

extract-max!

- same as MAX-HEAPIFY

$$\implies \Theta(\log(N))$$

Heapsort

```
function HEAPSORT(array)
    BUILD-MAX-HEAP(array)
    while array.heapsize > 0 do
        i ← array.heapsize
        array[i] ← EXTRACT-MAX!(array)
    end while
    return array
end function
```

Complexity analysis

- N calls to EXTRACT-MAX!
- each call takes $O(\log N)$ time

$$\implies O(N \log N)$$

- worst case, the first $\frac{N}{2}$ calls to EXTRACT-MAX! each do $\lceil \log N \rceil$ work

$$\implies \Theta(N \log N)$$

Priority queues

A priority queue tracks items along with priorities, and provides access to the highest-priority item.

maximum	return the highest-priority item
extract-max!	remove and return the highest-priority item
insert![o]	insert an item into the priority queue

(exactly the same as the heap operations)

Work

1. Reading
   - CLRS, chapter 6
2. Questions from CLRS

   6-1 Building a heap using insertion
3. Lab work

   3.1 (week of 28th January) implement an implicit heap class, with methods for:
      - computing the parent and children indices from a given index
      - constructing a heap in-place from a provided array input
      - inserting items into the heap (maintaining the heap property)
      - removing and returning the maximum element from the heap (maintaining the heap property)
      - performing heapsort

   3.2 (week of 28th January) measure the difference in operations between constructing a heap in-place and by repeated insertions. When (if ever) does the difference in scaling become noticeable?