Computer Science 331
Applications of Binary Heaps

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #26

Outline

HeapSort

A deterministic sorting algorithm that can be used to sort an array of length n using $\Theta(n \log n)$ operations in the worst case

Unlike `MergeSort` (which has the same asymptotic worst-case performance) this algorithm can be used to sort "in place," overwriting the input array with the output array, and using only a constant number of additional registers for storage

A disadvantage of this algorithm is that it is a little bit more complicated than the other asymptotically fast sorting algorithms we are studying (and seems to be a bit slower in practice)

HeapSort

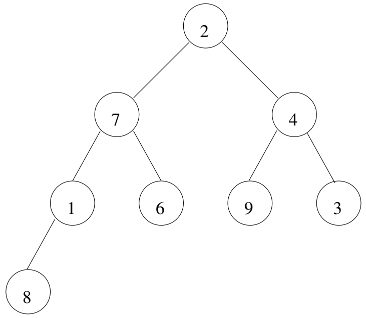
Idea:

- An array `A` of positive length, storing values from some ordered type `T`, can be turned into a Max-Heap of size 1 simply by setting `heap-size(A)` to be 1
- Inserting `A[1]`, `A[2]`, …, `A[A.length-1]` produces a Max-Heap while reordering the entries of `A` (without changing them, otherwise)
- Repeated calls to `deleteMax` will then return the entries, listed in decreasing order, while freeing up the space in `A` where they should be located when sorting the array.

HeapSort

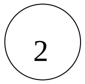
```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```

Example (Input)



0	1	2	3	4	5	6	7
2	7	4	1	6	9	3	8

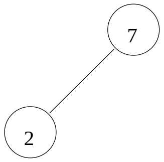
Example: Before First Execution, Loop Body, First Loop



0	1	2	3	4	5	6	7
2	7	4	1	6	9	3	8

heap-size(A) = 1

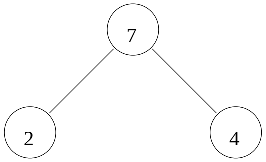
Example: Before Second Execution, Loop Body, First Loop



0	1	2	3	4	5	6	7
7	2	4	1	6	9	3	8

heap-size(A) = 2

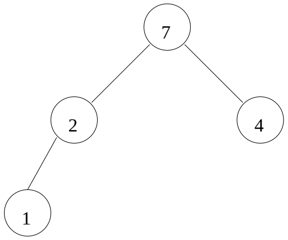
Example: Before Third Execution, Loop Body, First Loop



0	1	2	3	4	5	6	7
7	2	4	1	6	9	3	8

`heap-size(A)` $= 3$

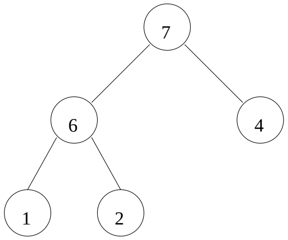
Example: Before Fourth Execution, Loop Body, First Loop



0	1	2	3	4	5	6	7
7	2	4	1	6	9	3	8

`heap-size(A)` $= 4$

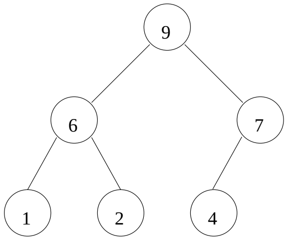
Example: Before Fifth Execution, Loop Body, First Loop



0	1	2	3	4	5	6	7
7	6	4	1	2	9	3	8

`heap-size(A)` $= 5$

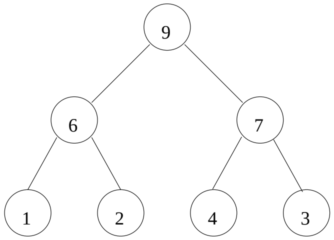
Example: Before Sixth Execution, Loop Body, First Loop



0	1	2	3	4	5	6	7
9	6	7	1	2	4	3	8

`heap-size(A)` $= 6$

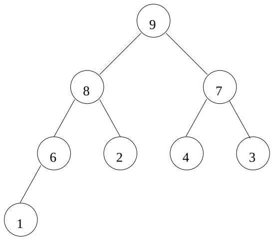
Example: Before Seventh Execution, Loop Body, First Loop



0	1	2	3	4	5	6	7
9	6	7	1	2	4	3	8

`heap-size(A)` = 7

Example: After Seventh Execution, Loop Body, First Loop

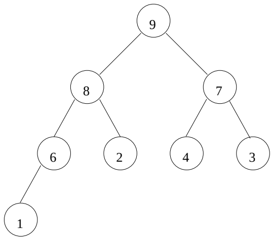


0	1	2	3	4	5	6	7
9	8	7	6	2	4	3	1

`heap-size(A)` = 8

Example: Before First Execution, Loop Body, Second Loop

`i` = 7

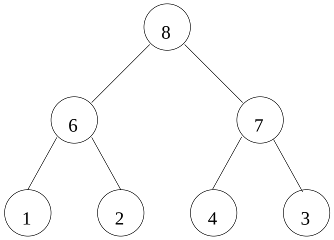


0	1	2	3	4	5	6	7
9	8	7	6	2	4	3	1

`heap-size(A)` = 8

Example: Before Second Execution, Loop Body, Second Loop

`i` = 6

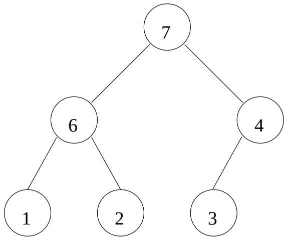


0	1	2	3	4	5	6	7
8	6	7	1	2	4	3	9

`heap-size(A)` = 7

Example: Before Third Execution, Loop Body, Second Loop

i = 5

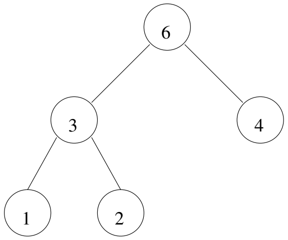


0	1	2	3	4	5	6	7
7	6	4	1	2	3	8	9

`heap-size(A)` = 6

Example: Before Fourth Execution, Loop Body, Second Loop

i = 4

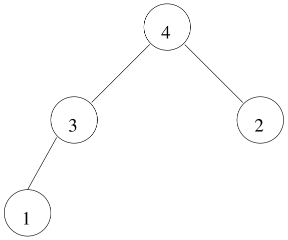


0	1	2	3	4	5	6	7
6	3	4	1	2	7	8	9

`heap-size(A)` = 5

Example: Before Fifth Execution, Loop Body, Second Loop

i = 3

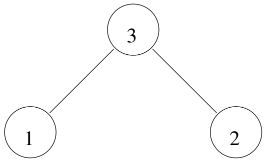


0	1	2	3	4	5	6	7
4	3	2	1	6	7	8	9

`heap-size(A)` = 4

Example: Before Sixth Execution, Loop Body, Second Loop

i = 2

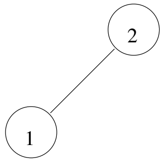


0	1	2	3	4	5	6	7
3	1	2	4	6	7	8	9

`heap-size(A)` = 3

Example: Before Seventh Execution, Loop Body, Second Loop

$\texttt{i} = 1$

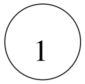


0	1	2	3	4	5	6	7
2	1	3	4	6	7	8	9

$\texttt{heap-size(A)} = 2$

Example: After Seventh Execution, Loop Body, Second Loop

$\texttt{i} = 0$



0	1	2	3	4	5	6	7
1	2	3	4	6	7	8	9

$\texttt{heap-size(A)} = 1$

Stop — array is sorted!

First Loop — Partial Correctness

Loop Invariant: The following properties are satisfied at the beginning of each execution of the body of the first loop.

a) `i` is an integer such that $1 \leq \texttt{i} < \texttt{A.length}$

b) `A` represents a heap with size `i`

c) The entries of the array `A` have been reordered but are otherwise unchanged

At the *end* of each execution of the body of the first loop, the following properties are satisfied.

- `i` is an integer such that $1 \leq \texttt{i} \leq \texttt{A.length}$
- Parts (b) and (c) of the loop invariant are satisfied

On *termination* of this loop `i = A.length`, so `A` represents a heap with size `A.length`, and the entries of `A` have been reordered but are otherwise unchanged.

First Loop — Termination and Efficiency

Loop Variant: $\texttt{A.length} - \texttt{i}$

Application:

- Number of executions of the body of this loop is at most:

$$\texttt{A.length} - 1$$

- The cost of a single execution of the body of this loop is at most: k

$$O(\log n), \text{ where } n = \texttt{A.length}$$

- *Conclusion:* The number of steps used by this loop in the worst case is at most:

$$O(n \log n)$$

Second Loop — Partial Correctness

Loop Invariant: The following properties are satisfied at the beginning of each execution of the body of the second loop.

a) `i` is an integer such that $1 \leq$ `i` $<$ `A.length`

b) `A` represents a heap with size `i` $+ 1$

c) if `i` $<$ `A.length` $- 1$ then `A[j]` $\leq$ `A[i+1]` for every integer `j` such that $0 \leq$ `j` $\leq$ `i`

d) `A[j]` $\leq$ `A[j+1]` for every integer `j` such that `i` $+ 1 \leq$ `j` $<$ `A.length` $- 1$

e) the entries of `A` have been reordered but are otherwise unchanged

---

Second Loop — Partial Correctness

At the *end* of each execution of the body of the second loop, the following properties are satisfied.

- `i` is an integer such that $0 \leq$ `i` $<$ `A.length`
- Parts (b), (c), (d) and (e) of the loop invariant are satisfied

On *termination* `i` $= 0$ and parts (b), (c), (d) and (e) of the loop invariant are satisfied. Notes that, when `i` $= 0$, parts (c) and (d) imply that the array is sorted, as required.

Second Loop — Termination and Efficiency

Loop Variant: `i`

Application:

- Number of executions of the body of this loop is at most:

$$\texttt{A.length} - 1$$

- The cost of a single execution of the body of this loop is at most:

$$O(\log n), \quad \text{where } n = \texttt{A.length}$$

- *Conclusion:* The number of steps used by this loop in the worst case is at most:

$$O(n \log n)$$

Analysis of Worst-Case Running Time, Concluded

Exercise: Show that if `A` is an array with length n, containing n *distinct* entries that already sorted in increasing order, then this `HeapSort` algorithm uses $\Omega(n \log n)$ steps on input `A`.

Conclusion: The worst-case running time of `HeapSort` (when given an input array of length n) is in $\Theta(n \log n)$.

Priority Queues

Definition: A *priority queue* is a data structure for maintaining a multiset `S` of elements, of some type `V`, each with an associated value (of some ordered type `P`) called a *priority*.

A class that implements *max-priority queue* provides the following operations (not, necessarily, with these names):

- `void insert(V value, P priority)`: Insert the given `value` into `S`, using the given `priority` as its priority in this priority queue
- `V maximum()`: Report an element of `S` stored in this priority that has highest priority, without changing the priority queue (or `S`)
- `V extract-max()`: Remove an element of `S` with highest priority from the priority queue (and from `S`) and return this value as output

---

Priority Queues

Priority Queues in Java:

- Class `PriorityQueue` in the Java Collections framework implements a "min-priority queue" — which would provide methods `minimum` and `extract-min` to replace `maximum` and `extract-max`, respectively
- Also implements the `Queue` interface, so the names `insert`, `minimum`, and `extract-min` of methods are replaced by the names `add`, `peek`, and `remove`, respectively.
- Furthermore, the signature of `insert` is a little different — no `priority` is provided — because the values themselves are used *as* their priorities (according to their "natural order")

Priority Queues

Dealing With This Restriction:

- In order to provide more general priorities, one can simply write a class, each of whose objects "has" a value of type `V` (that is, the element of `S` it represents) and that also "has" a value of type `P` (that is, the priority). The class should implement the `Comparable` interface, and `compareTo` should be implemented using the ordering for priorities

Applications:

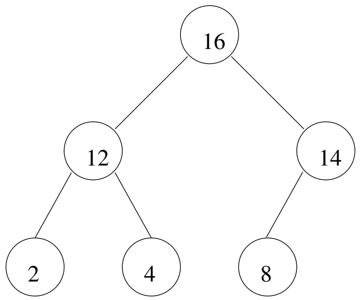
- Scheduling: Priorities reflect the order of requests and determine the order in which they should be served

---

Implementation

Binary Heaps are often used to implement priority queues.

Example: One representation of a max-priority queue including keys $S = \{2, 4, 8, 12, 14, 16\}$ is as follows:



0	1	2	3	4	5	6	7
16	12	14	2	4	8	9	3

$\texttt{A.length} = 8;$
$\texttt{heap-size(A)} = 6$

Implementation of Operations

A “max-priority queue” can be implemented, in a straightforward way, using a Max-Heap.

- `insert`: Use the `insert` method for the binary heap that is being used to implement this priority queue
- `maximum`: Throw an exception if the binary heap has size zero; return data stored at position 0 if the array that represents the heap, otherwise
- `extract-min`: Use the `deleteMax` method for the binary heap that implements this priority queue

Consequence: If the priority queue has size n then `insert` and `extract-min` use $\Theta(\log n)$ operations in the worst case, while `maximum` uses $\Theta(1)$ operations in the worst case.

Binomial and Fibonacci Heaps

Introduction to Algorithms, Chapter 19 and 20

Better than binary heaps if **Union** operation must be supported:

- creates a new heap consisting of all nodes in two input heaps

Function	Binary Heap (worst-case)	Binomial Heap (worst-case)	Fib. Heap (amortized)
Insert	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
Maximum	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
Extract-Max	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
Increase-Key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
Union	$\Theta(n)$	$O(\log n)$	$\Theta(1)$

References

Information about `HeapSort` and priority queues is also available in the textbook.

- Priority queues are discussed in Section2 8.1 and 8.2 of the textbook
- `HeapSort`, and implementing a priority queue using a heap, is discussed in Section 8.3 of the textbook