Sorting (cont.)

Alexandre David
B2-206
Heap Data Structure

- Binary tree with some properties:
  - **root** = A[1] - root of the tree
  - **parent** (i) = i/2 - for a node i
  - **left** (i) = 2i - left child
  - **right** (i) = 2i + 1 - right child
  - **max-heap** property: A[parent(i)] ≥ A[i]
  - (or **min-heap** A[parent(i)] ≤ A[i])
  - height is $\Theta(\log n)$. 
Heap Data Structure

![Heap Diagram]

- root = A[1]
- parent(i) = i/2
- left(i) = 2i
- right(i) = 2i + 1
- max-heap: A[parent(i)] ≥ A[i]

Array: [9 5 8 4 3 7 6 1 2]

Max-heap: root = greatest element.

satisfying heap properties

seen as a binary tree:

double number of nodes
Heap Sort

- **Idea:**
  - Use a heap data structure.
  - Max-heap for ascending: 1\textsuperscript{st} element is the max ⇒ move it at the end, restore the heap, loop.

- **Basic procedures:**
  - **max-heapify:** maintains max-heap property $O(\lg n)$, knowing that sub-trees are heaps.
  - **build-max-heap:** computes a max-heap from scratch in $O(n)$.
  - **heap-sort:** sorts an array in-place in $O(n \lg n)$. 
Idea of Heap Sort

1: Build a heap.

2: Repeat
- move 1st at the end
- restore heap
Idea of Heap Sort

2: Repeat
- move 1st at the end
- restore heap
Idea of Heap Sort

2: Repeat
- move 1st at the end
- restore heap
Maintaining a Heap

Procedure **max-heapify**(A,i):
- Assume that left(i) and right(i) are max-heaps.
- A[i] may be smaller than its children, we’re going to fix that.
- Propagate A[i] down on the “right” path.
- When the algorithm terminates, the max-heap property is restored.
Maintaining a Heap

max-heapify(A, i)
    if A[left(i)] > A[right(i)] then
        child = left(i)
    else
        child = right(i)
    fi
        swap(A[i], A[child])
        max-heapify(A, child)
    fi

Clearly $O(\log n)$. Why is it correct?
Maintaining a Heap

max-heapify(A,2)
Maintaining a Heap

max-heapify(A,4)
Maintaining a Heap

leaf reached - stop

Possible to stop before if $A[i] \geq \max(A[\text{left}], A[\text{right}])$. 
Building a Heap

- We can use **max-heapify** in a bottom-up manner to build the tree incrementally.

```plaintext
build-max-heap(A):
heap_size(A) = length(A)
for i = length(A)/2 downto 1 do
    max-heapify(A, i)
done
```

- Seems $O(n \log n)$ but is in fact $O(n)$.
- Why do we start from length(A)/2?
- Why does it work?
Building a Heap

max-heapify

start
Building a Heap

max-heapify

[4]: 2
[8]: 9
[9]: 14
[10]: 1
[2]: 4
[5]: 16
[1]: 7
[3]: 3
[6]: 8
[7]: 10
Building a Heap

max-heapify
Building a Heap

max-heapify

[1]: 7

[2]: 4

[4]: 14

[8]: 9

[9]: 2

[5]: 16

[10]: 1

[3]: 10

[6]: 8

[7]: 3
Building a Heap

max-heapify
Building a Heap

Diagram of Heap Structure:

1. Root: [16]
2. Left Child: [14]
   - Left Grandchild: [9]
     - Grandchild: [7]
   - Right Grandchild: [2]
3. Right Child: [10]
   - Left Grandchild: [4]
   - Right Grandchild: [1]
4. Left Child: [10]
   - Left Grandchild: [3]
5. Right Child: [8]
6. Right Child: [3]
All Together: Heap Sort

- Build a max-heap.
- Repeat:
  - Put the root at the end of the heap,
  - max-heapify on the smaller heap.

```python
heap-sort(A):
    build-max-heap(A)
    for i = length(A) downto 2 do
        swap(A[1], A[i])
        heap_size(A) = heap_size(A) - 1
        max-heapify(A, 1)
    done
```

Clearly \( O(n \log n) \) but also \( \Omega(n \log n) \).
Heap Sort

After building the heap:

[1]: 16
[2]: 14
[3]: 10
[4]: 9
[5]: 4
[6]: 8
[7]: 3
[8]: 7
[9]: 2
[10]: 1

swap and decrease the heap
Heap Sort

max-heapify

[1]: 1
[2]: 14
[3]: 10
[4]: 9
[5]: 4
[6]: 8
[7]: 3
[8]: 7
[9]: 2
[10]: 16
Heap Sort

[1]: 14
[2]: 9
[3]: 10
[4]: 7
[5]: 4
[6]: 8
[7]: 3
[8]: 1
[9]: 2
[10]: 16

swap and decrease the heap
Heap Sort

max-heapify

[1]: 2
[2]: 9
[3]: 10
[4]: 7
[5]: 4
[6]: 8
[7]: 3
[8]: 1
[9]: 14
[10]: 16
Heap Sort

[1]: 10
[2]: 9
[3]: 8
[4]: 7
[5]: 4
[6]: 2
[7]: 3
[8]: 1
[9]: 14
[10]: 16
Heap Sort

[1]: 1
[2]: 9
[3]: 8
[4]: 7
[5]: 4
[6]: 2
[7]: 3
[8]: 10
[9]: 14
[10]: 16
Heap Sort

[1]: 9
[2]: 7
[3]: 8
[4]: 1
[5]: 4
[6]: 2
[7]: 3
[8]: 10
[9]: 14
[10]: 16
Heap Sort

Heap: [3, 7, 8, 1, 4, 2, 9, 10, 14, 16]
Heap Sort

[1]: 8
[2]: 7
[3]: 3
[4]: 1
[5]: 4
[6]: 2
[7]: 9
[8]: 10
[9]: 14
[10]: 16
etc...
Heap Sort
Heap Sort

[1]: 4
[2]: 2
[3]: 3
[4]: 1
[5]: 7
[6]: 8
[7]: 9
[8]: 10
[9]: 14
[10]: 16
Heap Sort

[1]: 3
[2]: 2
[3]: 1
[4]: 4
[5]: 7
[6]: 8
[7]: 9
[8]: 10
[9]: 14
[10]: 16
Heap Sort
Heap Sort

[1]: 1
[2]: 2
[3]: 3
[4]: 4
[5]: 7
[6]: 8
[7]: 9
[8]: 10
[9]: 14
[10]: 16
Priority Queues

- Another use of the heap data structure.
- Exploit the max-heap property (or min-heap).
- Operations:
  - insert an element, $O(\log n)$
  - get the max (or the min) element, $O(1)$
  - remove the max (or the min) element, $O(\log n)$
  - increase the key of an element. $O(\log n)$
- Used in the STL (C++).
Important Procedure: Increase Key

- Propagate the key upwards until the max-heap property is restored.

```
heap-increase-key(A,i,key):
  if key < A[i] then error
  A[i] = key
  while i > 1 and A[parent(i)] < A[i] do
    swap(A[i], A[parent(i)])
    i = parent(i)
  done
```
Increasing a Key