CH08-320201

Algorithms and Data Structures

ADS

Lecture 6

Dr. Kinga Lipskoch

Spring 2019

Matrix Multiplication (4)

SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)

$\Theta(1)$

$n = A.rows$
let C be a new $n \times n$ matrix
if $n == 1$
    $c_{11} = a_{11} \cdot b_{11}$

$\Theta(1)^*$

else partition A, B, and C
    $C_{11} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{11})
        $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{21})
    $C_{12} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{12})
        $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{22})
    $C_{21} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{11})
        $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{21})
    $C_{22} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{12})
        $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{22})
return C

$\Theta(n^2)$

$8T(n/2)$

* with index calculations,
otherwise $\Theta(n^2)$ for copying entries

Matrix Multiplication (5)

Recurrence:

$$T(n) = \underbrace{8}_{\text{\#subproblems}}T(\underbrace{n/2}_{\text{subproblem size}}) + \underbrace{\Theta(n^2)}_{\text{work dividing and combining}}$$

$n^{log_b a} = n^{log_2 8} = n^3, f(n) = \Theta(n^2)$
meaning that $f(n) = \Theta(n^{3-\epsilon})$, where $\epsilon = 1$
Case 1: $T(n) = \Theta(n^3)$.

Not better than the standard algorithm.

Matrix Multiplication (6)

- Lessons learned:
  - #additions goes away
    (constant factor)
  - #multiplications not $\Rightarrow$ recursive case
    (they make the tree "bushy")
- What to do?
  - Try to reduce #multiplications
  - It is ok to have more additions

Matrix Multiplication (7)

Strassen's idea:
Multiply matrices with 7 multiplications and 18 additions.

$$
\begin{aligned}
P_1 &= a \cdot (f - h) \\
P_2 &= (a + b) \cdot h \\
P_3 &= (c + d) \cdot e \\
P_4 &= d \cdot (g - e) \\
P_5 &= (a + d) \cdot (e + h) \\
P_6 &= (b - d) \cdot (g + h) \\
P_7 &= (a - c) \cdot (e + f)
\end{aligned}
\qquad
\begin{aligned}
r &= P_5 + P_4 - P_2 + P_6 \\
s &= P_1 + P_2 \\
t &= P_3 + P_4 \\
u &= P_5 + P_1 - P_3 - P_7
\end{aligned}
$$

$$
\left[\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right] = \left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \cdot \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right]
$$

$$
C \quad = \quad A \quad \cdot \quad B
$$

Matrix Multiplication (8)

Strassen's idea:

$$
\begin{aligned}
P_1 &= a \cdot (f - h) \\
P_2 &= (a + b) \cdot h \\
P_3 &= (c + d) \cdot e \\
P_4 &= d \cdot (g - e) \\
P_5 &= (a + d) \cdot (e + h) \\
P_6 &= (b - d) \cdot (g + h) \\
P_7 &= (a - c) \cdot (e + f)
\end{aligned}
$$

$$
\begin{aligned}
r &= P_5 + P_4 - P_2 + P_6 \\
&= (a + d)(e + h) \\
&\quad + d(g - e) - (a + b)h \\
&\quad + (b - d)(g + h) \\
&= ae + ah + de + dh \\
&\quad + dg - de - ah - bh \\
&\quad + bg + bh - dg - dh \\
&= ae + bg
\end{aligned}
$$

$$
\left[\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right] = \left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \cdot \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right]
$$

$$
C = A \cdot B
$$

Matrix Multiplication (9)

Strassen's algorithm:

1. Divide:
   Partition A and B into $(n/2) \times (n/2)$ submatrices.
   Form terms to be multiplied using $+$ and $-$.
2. Conquer:
   Perform 7 multiplications of $(n/2) \times (n/2)$ submatrices recursively.
3. Combine:
   Form C using $+$ and $-$ on $(n/2) \times (n/2)$ submatrices.

Matrix Multiplication (10)

Complexity of Strassen's algorithm:
Recurrence:

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

$n^{log_b a} = n^{log_2 7} = n^{2.81}$
Case 1: $T(n) = \Theta(n^{\lg 7})$

2.81 may not seem much smaller than 3, but the difference is in the exponent, therefore the impact on running time is significant.

Strassen's algorithm beats the standard algorithm for $n \geq 32$ or so.

Matrix Multiplication (11)

Best known algorithm:
Latest improvement in 2014 in the following publication:
Francois LeGall, Powers of Tensors and Fast Matrix Multiplication, 30 Jan 2014

$T(n) = O(n^{2.3728639})$

- Only of theoretical interest.
- Most approaches that are faster than Strassen's are not used in practice.
- They are only faster for very large n.
- One cannot get better than $O(n^2)$, cf. Case 3.

Intermediate Conclusion

- Definitions
- First example of an algorithm – Insertion Sort
- Asymptotic analysis
- First powerful concept – Divide & Conquer
- Solve recurrences for analysis

Recall: Sorting Problem

- Input:
  - Sequence $< a_1, a_2, ..., a_n >$ of numbers
- Output:
  - Permutation $< a'_1, a'_2, ..., a'_n >$
  - Such that $a'_1 \leq a'_2 \leq ... \leq a'_n$

Recall: Insertion & Merge Sort

Time complexity:

	Insertion Sort	Merge Sort
Best case	$\Theta(n)$	$\Theta(n \lg n)$
Average case	$\Theta(n^2)$	$\Theta(n \lg n)$
Worst case	$\Theta(n^2)$	$\Theta(n \lg n)$

Visualizations:

http://www.sorting-algorithms.com/insertion-sort
http://www.sorting-algorithms.com/merge-sort

What about storage space complexity?

In-situ Sorting

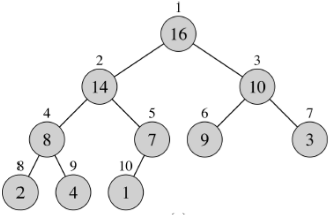
- Definition:
  - In-situ algorithms refer to algorithms that operate with $\Theta(1)$ memory
- In-situ sorting:
  - Sorting algorithms that need only a constant number of additional storings
- Insertion Sort:
  - In-situ sorting
- Merge Sort:
  - Not in-situ sorting

Heap Sort: Motivation

- Try to develop an in-situ sorting algorithm with asymptotic runtime $\Theta(n \lg n)$.
- Use a sophisticated data structure to support the computations.

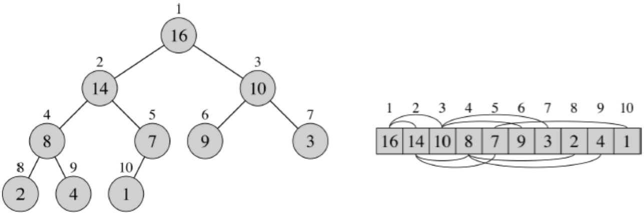
Heap: Data structure

Defintion:
A (binary) heap data-structure is an array which can be viewed as a nearly complete binary tree: each level is completely full except possibly the last level, which is filled from left to right.



Heap as an Array (1)

A heap can be stored as an array:

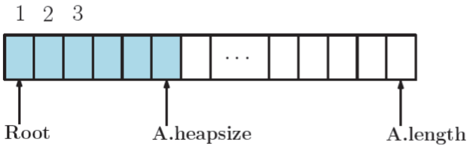


Heap as an Array (2)

The array A representing the heap has two attributes:

- *A.length*
- *A.heapsize*

such that $0 \leq A.heapsize \leq A.length$.
There are only *A.heapsize* valid elements of the heap.

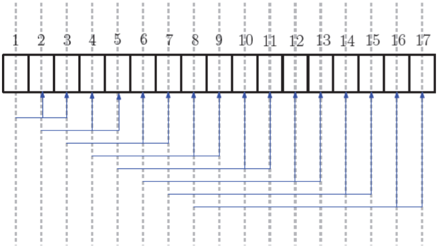


$A[1]$ is the root of the heap (root of the binary tree).

Heap as an Array (3)

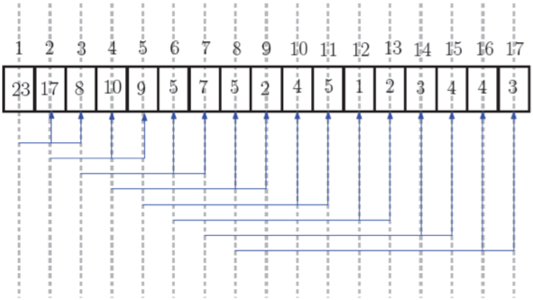
Given the index i of an element of A, we can calculate:

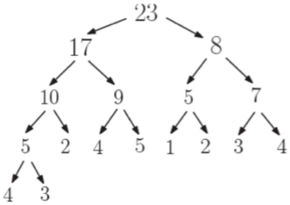
- `Parent(i): return floor(i/2);`
  `// Right shift by 1 bit`
- `Left(i): return 2i;`
  `// Left shift by 1 bit`
- `Right(i): return 2i + 1;`
  `// Left shift by 1 bit and set LSB to 1`



Max-Heap Property

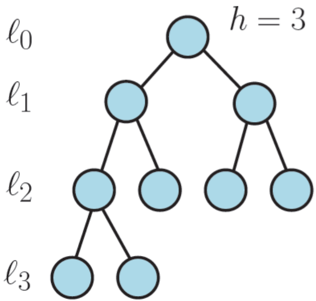
In a max-heap, for every node i (other than the root), $A[Parent(i)] \geq A[i]$.





Recall: Height of a Tree

- The height of a node x is the length of the longest simple downward path from x to a leaf.
- The height of a tree is the height of its root.



Heap Height (1)

Theorem:
A heap with n elements has height $h = \lfloor \lg n \rfloor$.

Heap Height (2)

Proof:

Heap height h implies that there are $h+1$ levels (levels 0 to h).
As a heap is a nearly complete binary tree, the last guaranteed complete level is level $h-1$.
The level h may be incomplete, but it has at least one element.

The number of elements in complete levels 0 to $h-1$ is
$1+2+2^2+...+2^{h-1}=2^h-1$.
So, $n>2^h-1$ or (since it is an integer) $n \geq 2^h$.

If all levels 0 to h were complete, the number of elements would be $2^{h+1}-1$.
So, $n \leq 2^{h+1}-1$.

Heap Height (3)

Proof (continued):
Combining the two inequalities:
$2^h \leq n \leq 2^{h+1} - 1$

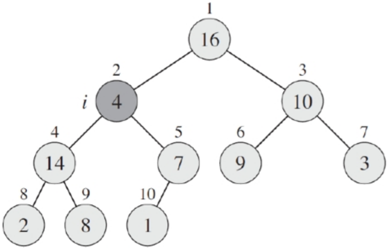
As $2^{h+1} > 2^{h+1} - 1 \geq 2^h$ for $h \geq 0$,
$h + 1 > \lg(2^{h+1} - 1) \geq h$

Thus, $\lg(2^{h+1} - 1) = h + \alpha$ with $\alpha \in [0, 1)$, which leads to
$h \leq \lg n \leq h + \alpha$ with $\alpha \in [0, 1)$.

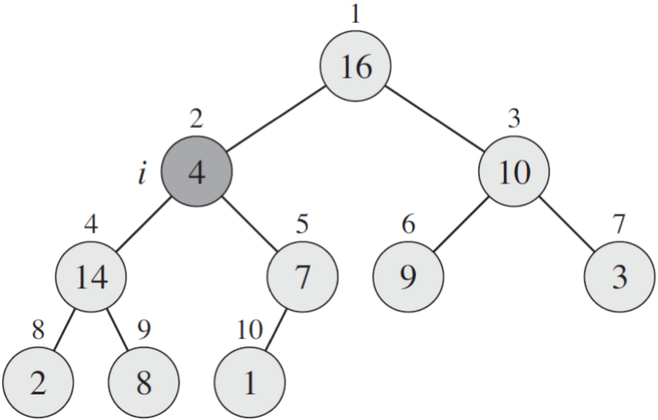
Hence, $h = \lfloor \lg n \rfloor$.

Max-Heapify(A, i) (1)

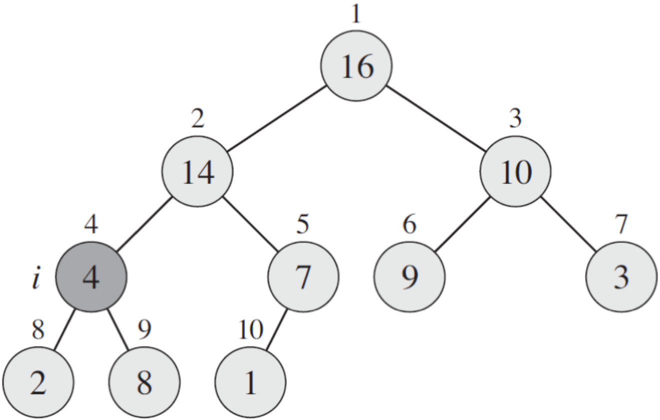
Precondition:
When *Max-Heapify*(A, i) is called, binary-trees rooted at *Left*(i) and *Right*(i) are valid max-heaps, but $A[i]$ may be smaller than its children.



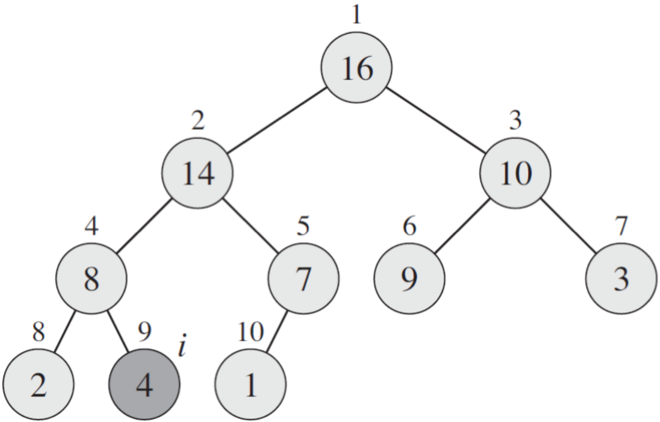
Max-Heapify(A, i) (2)



Max-Heapify(A, i) (3)



Max-Heapify(A, i) (4)



Max-Heapify(A, i) (5)

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```