CS 311: Algorithm Design and Analysis

Lecture 6

Last Lecture we have

- Recursion Tree
- Quick Sort
- Heap Sort

This Lecture we have

- Heap Sort Example
- Linear Sorting Algorithms
- Multiplication of large integers
- Tower of Hanoi

HeapSort

procedure ***DeleteMax****(A)*
    if *size[A] = 0* ***then return*** *error*
    MaxItem $\leftarrow$ *A[1]*
    A[1] $\leftarrow$ *A[size[A]]*
    size[A] $\leftarrow$ *size[A] – 1*
    DownHeap(A, 1)
    return *MaxItem*
end

Algorithm ***HeapSort****(A[1..n])*   *§ O(n log n) time*
Pre-Cond: input is array A[1..n] of arbitrary numbers
Post-Cond: A is rearranged into sorted order
    ConstructMaxHeap(A[1..n])
    for *t* $\leftarrow$ *n downto 2* ***do***
        A[t] $\leftarrow$ *DeleteMax(A)*
end

Analysis of Heapsort (continued)

Recall algorithm:

$\Theta(n)$ 1. *Build heap*

2. *Remove root –exchange with last (rightmost) leaf*

3. *Fix up heap (excluding last leaf)*

$\Theta(\log k)$

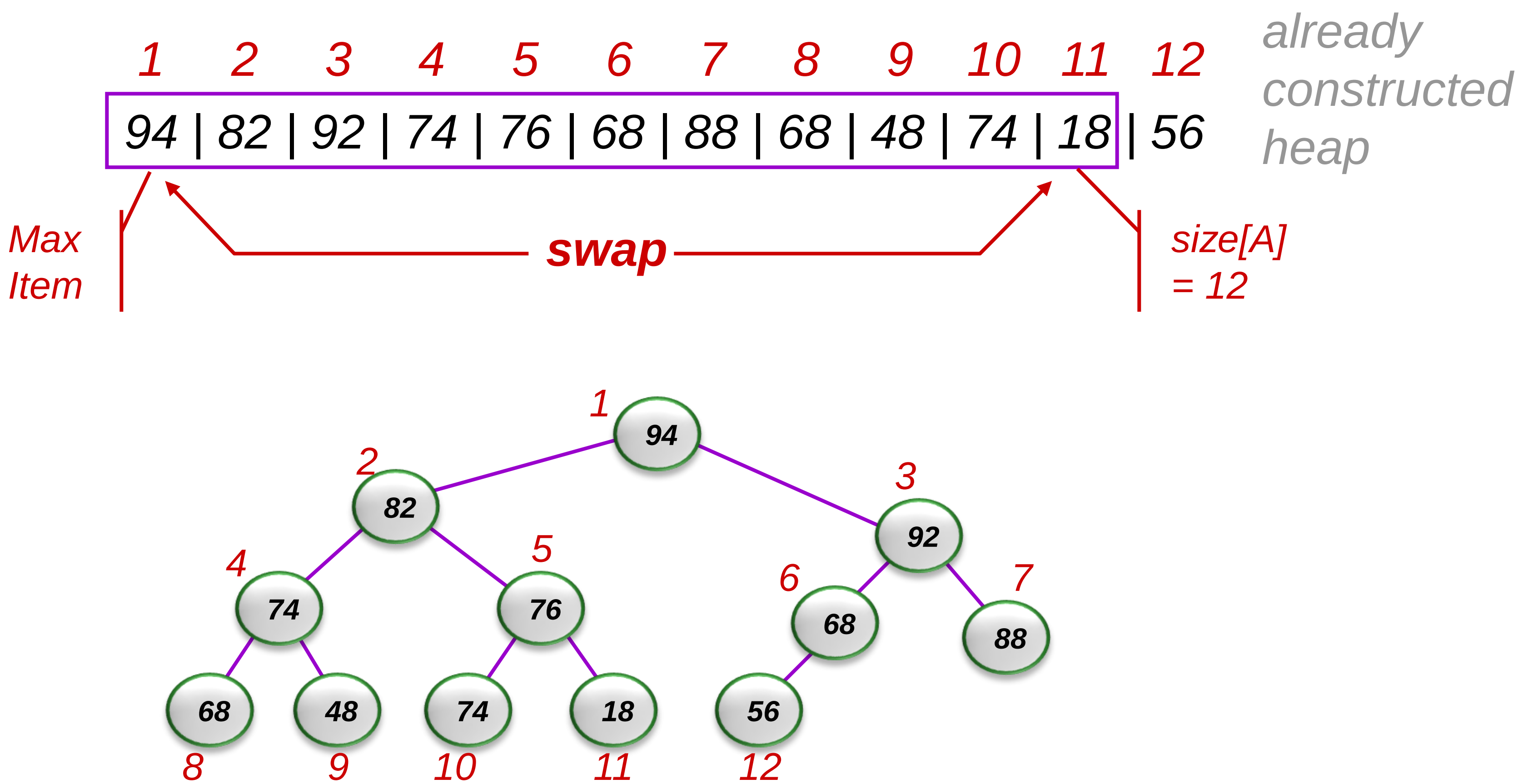
Repeat 2, 3 until heap contains just one node.

k=n – 1, n-2, … 1

Total: $\Theta(n) + \Theta(n \log n) = \Theta(n \log n)$

- ***Note:*** *this is the worst case. Average case also $\Theta(n \log n)$.*

HeapSort Example

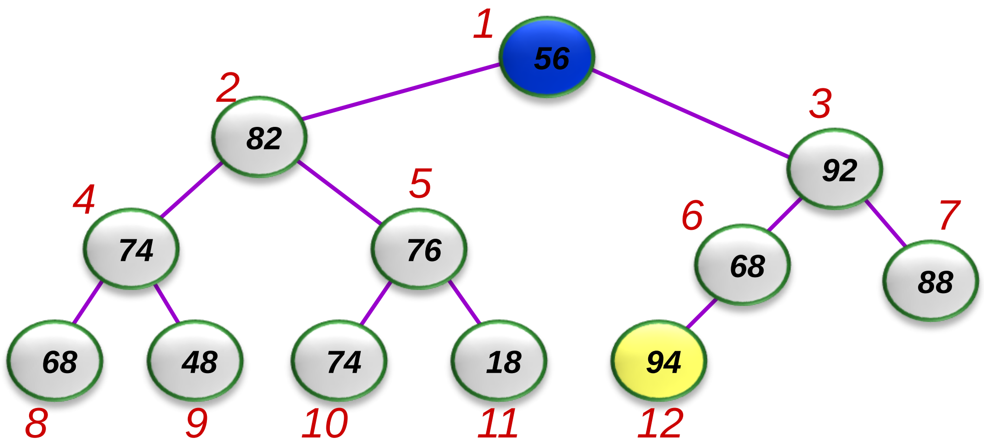


HeapSort Example

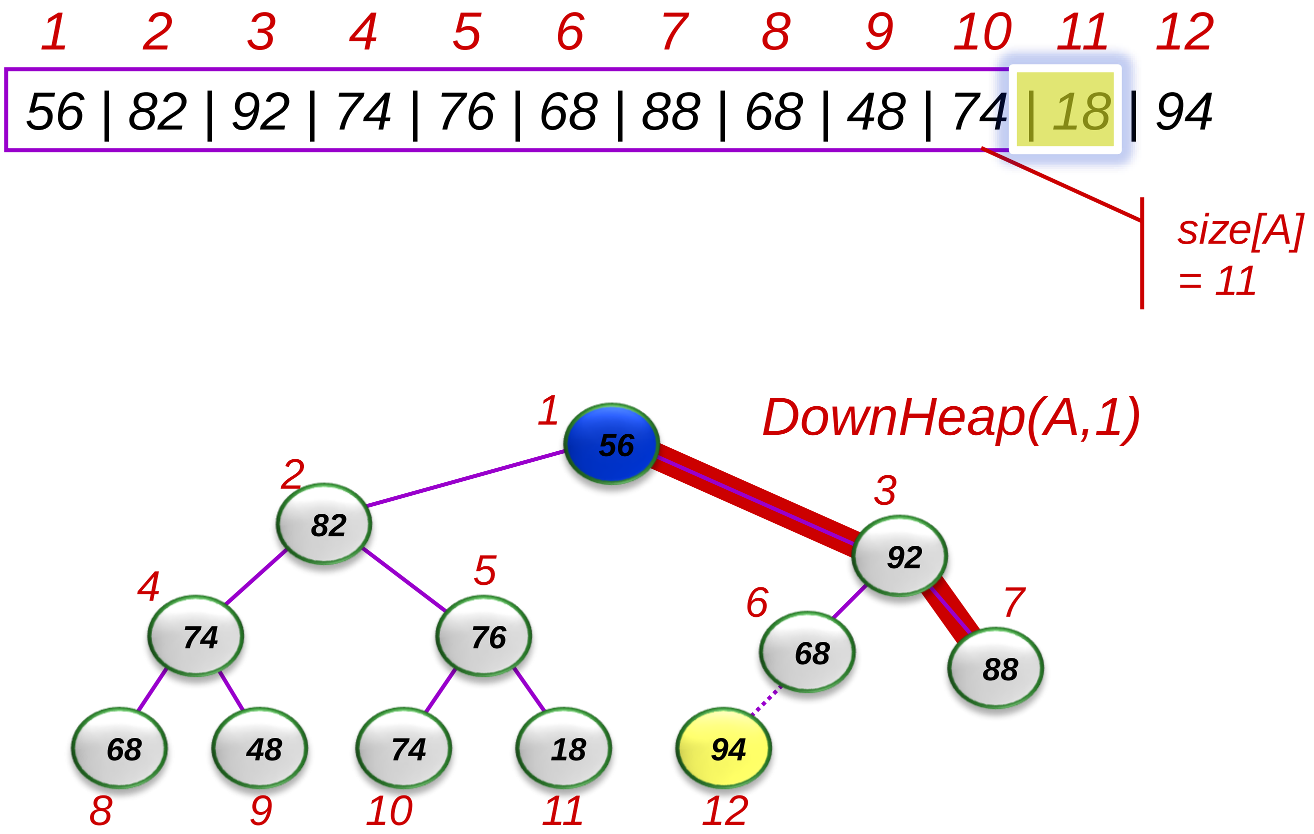
1 2 3 4 5 6 7 8 9 10 11 12

56 | 82 | 92 | 74 | 76 | 68 | 88 | 68 | 48 | 74 | 18 | 94

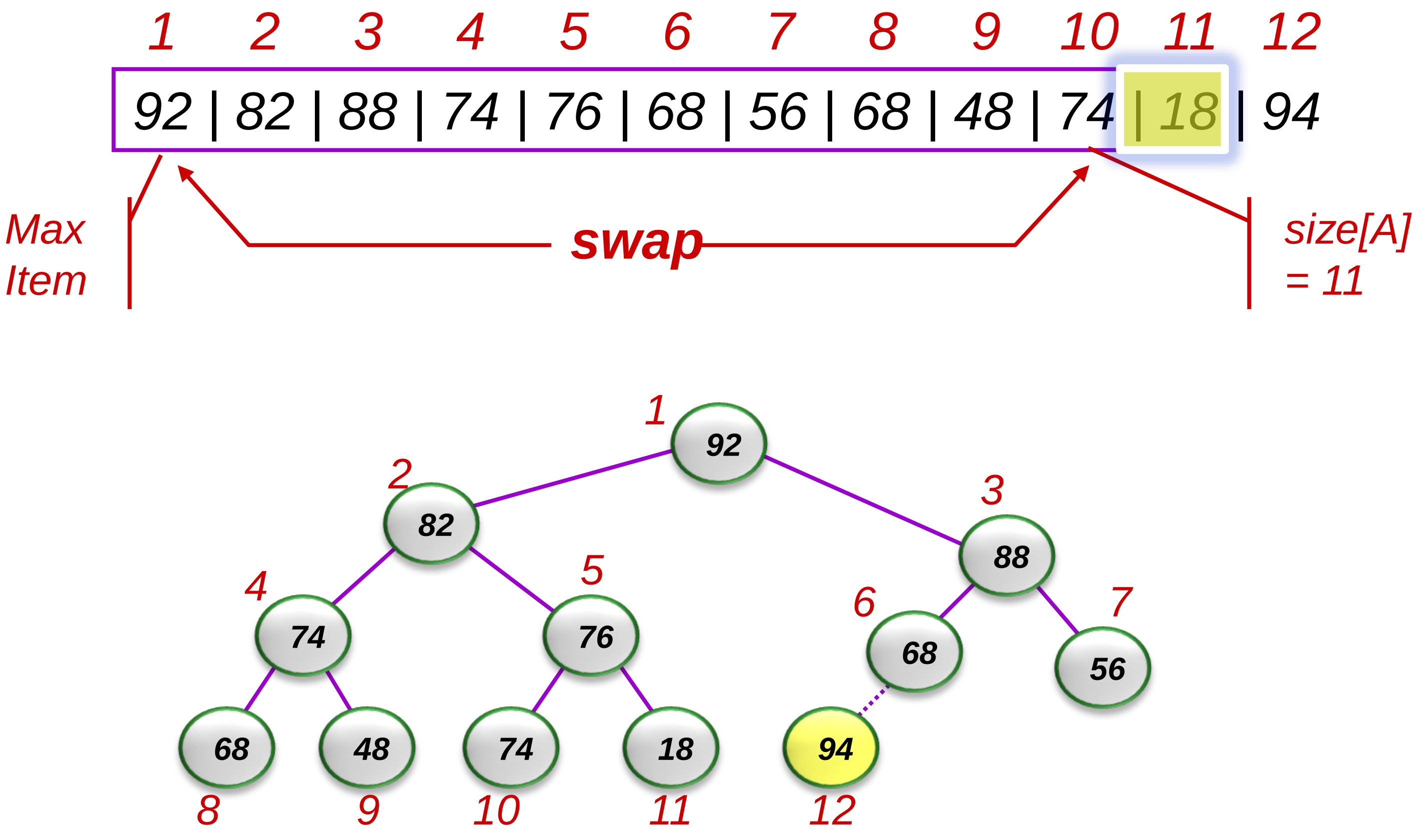
size[A] = 12



HeapSort Example



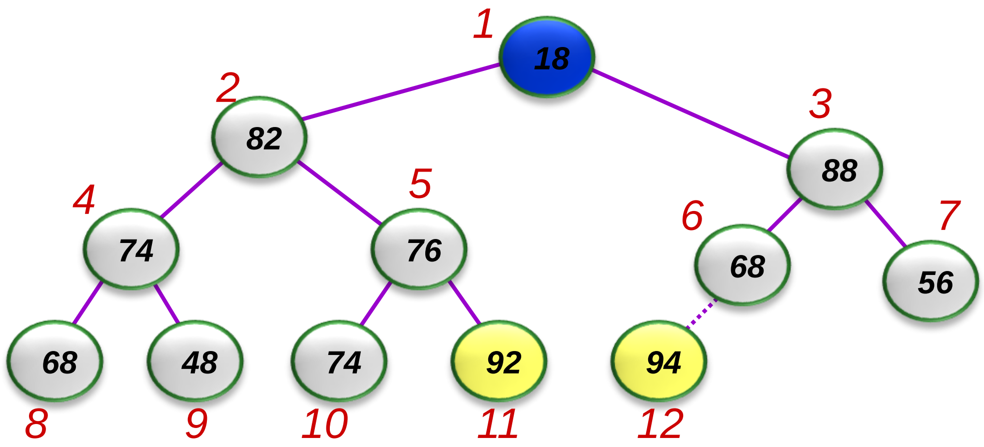
HeapSort Example



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
18	82	88	74	76	68	56	68	48	74	92	94

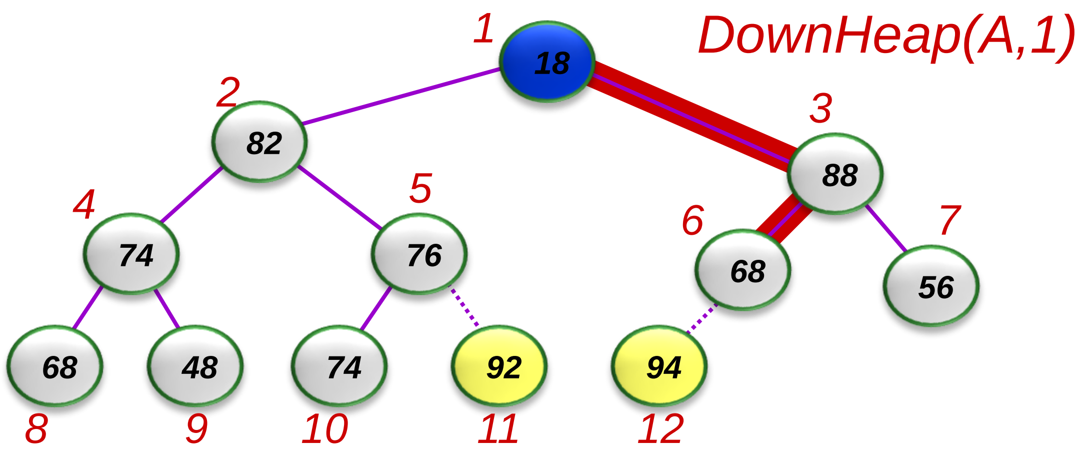
size[A] = 11



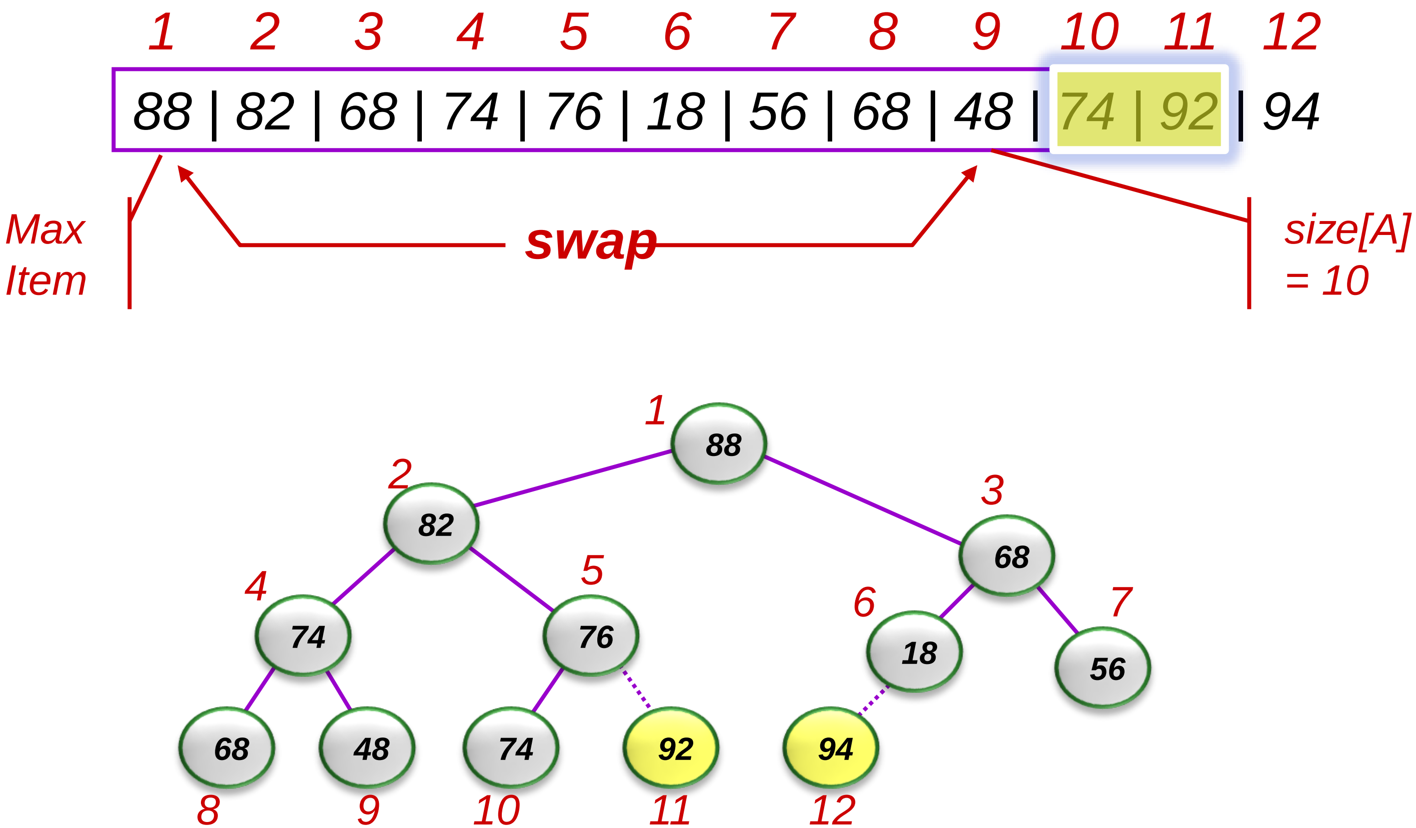
HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
18	82	88	74	76	68	56	68	48	74	92	94

size[A] = 10



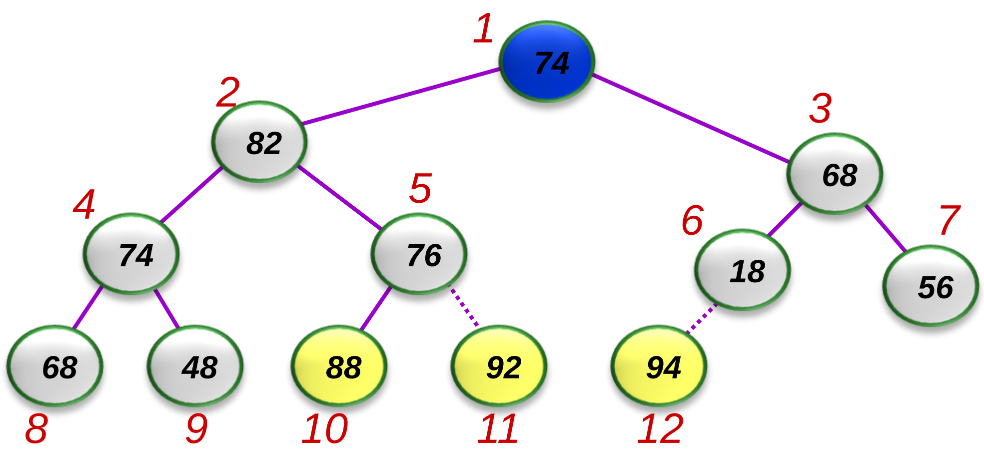
HeapSort Example



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
74	82	68	74	76	18	56	68	48	88	92	94

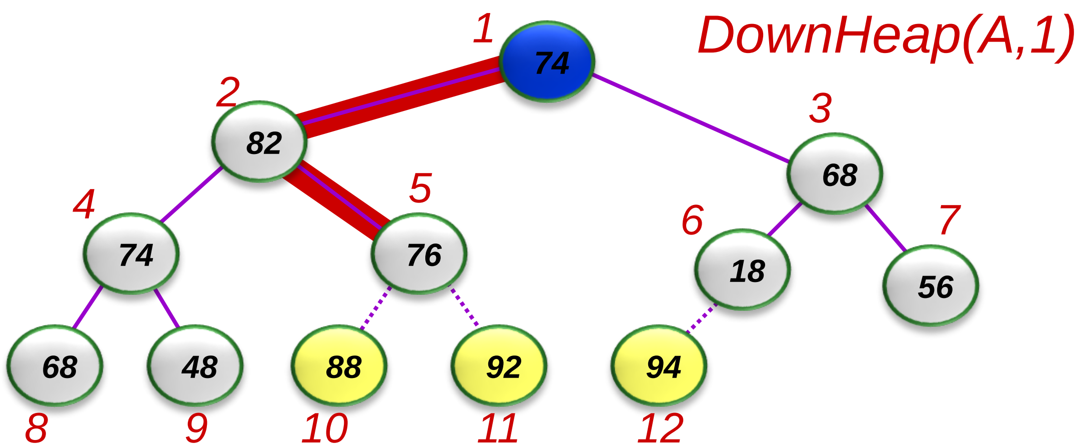
size[A] = 10



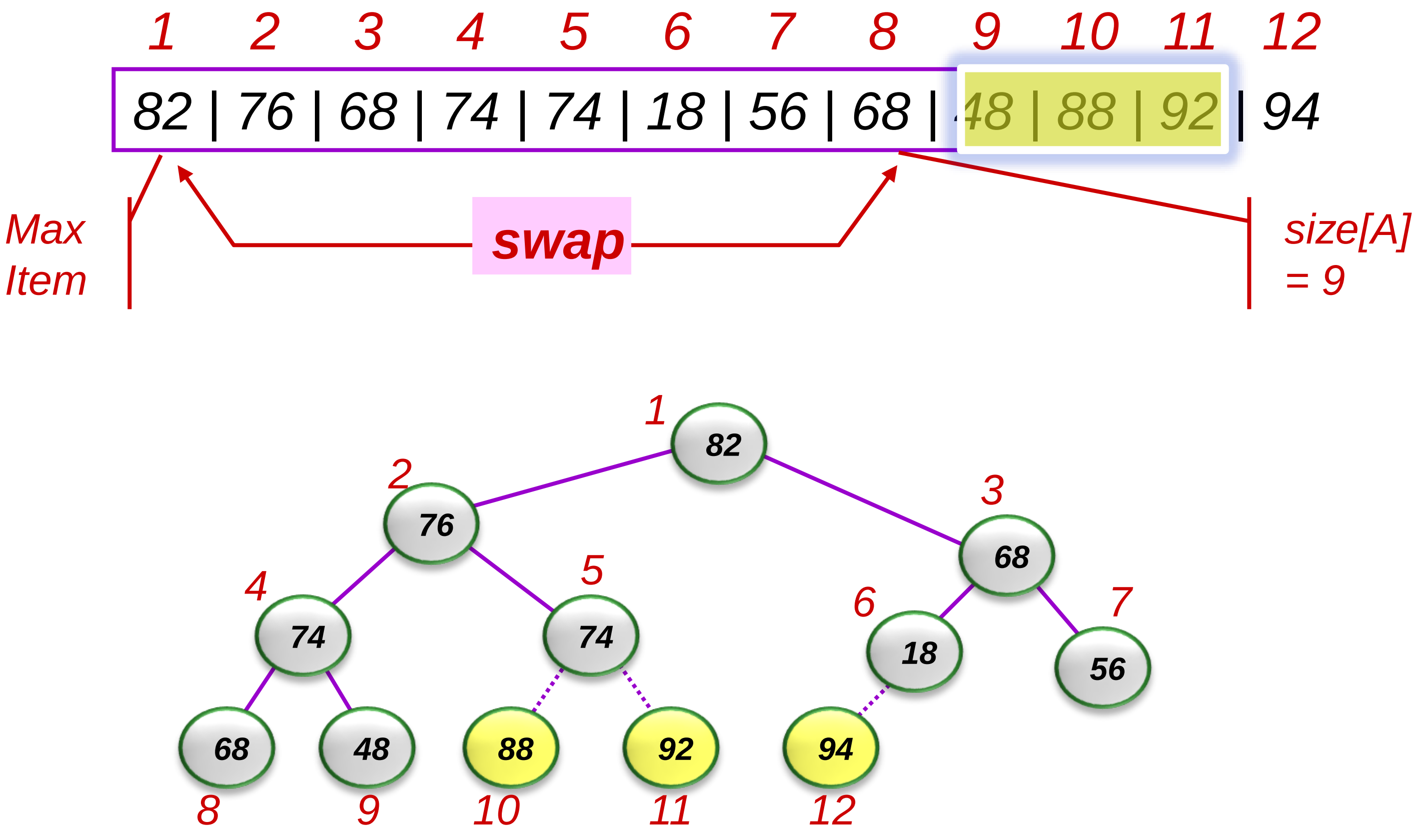
HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
74	82	68	74	76	18	56	68	48	88	92	94

size[A] = 9



HeapSort Example

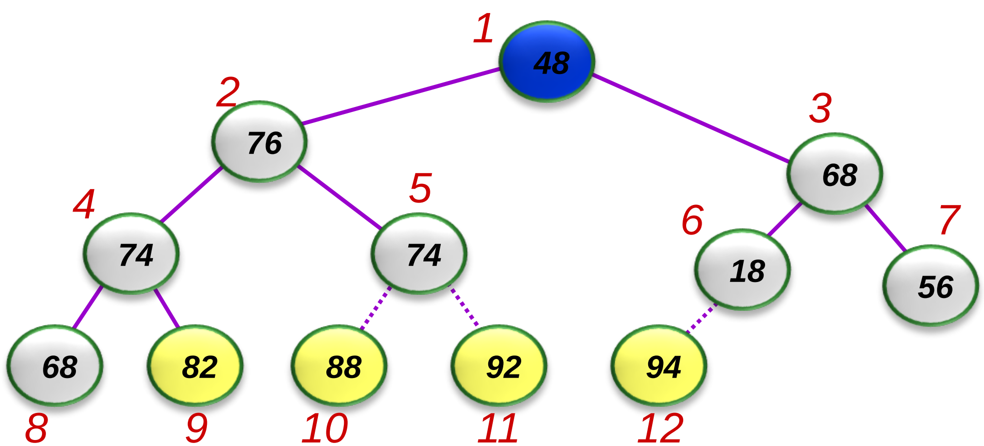


HeapSort Example

1 2 3 4 5 6 7 8 9 10 11 12

48 | 76 | 68 | 74 | 74 | 18 | 56 | 68 | 82 | 88 | 92 | 94

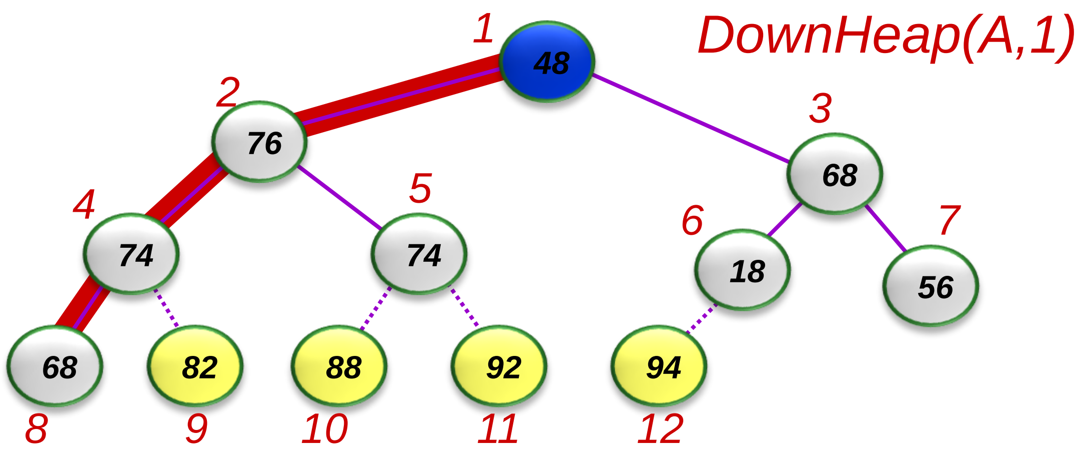
size[A] = 9



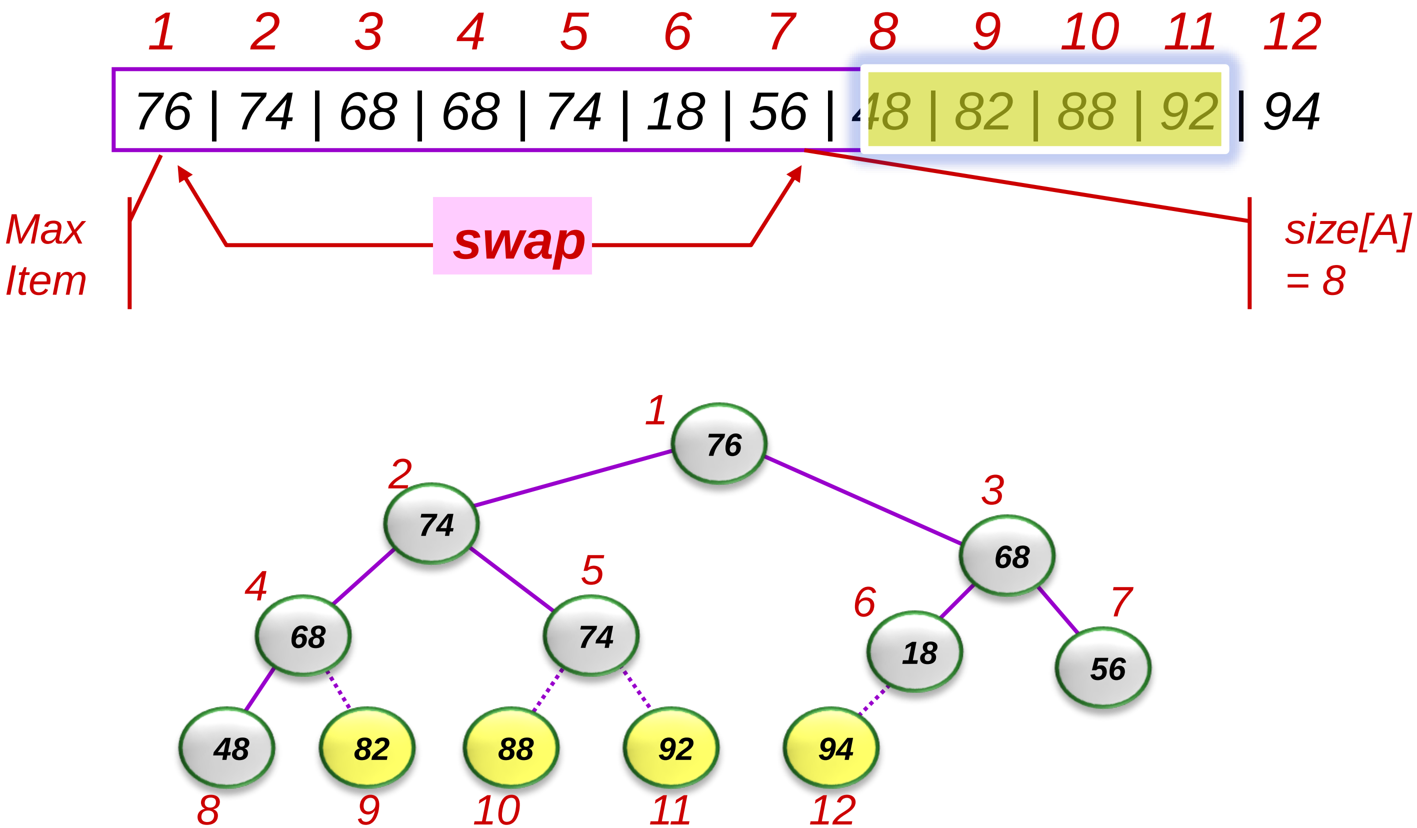
HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
48	76	68	74	74	18	56	68	82	88	92	94

size[A]
= 8



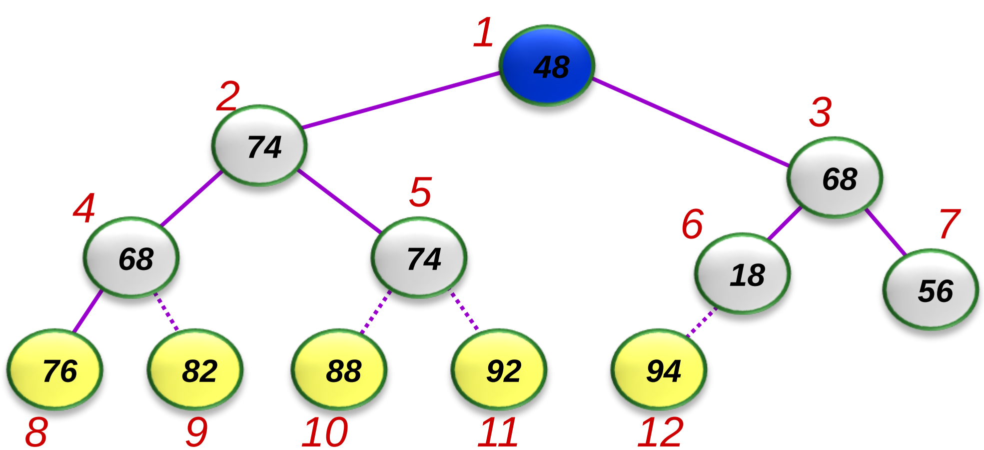
HeapSort Example



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
48	74	68	68	74	18	56	76	82	88	92	94

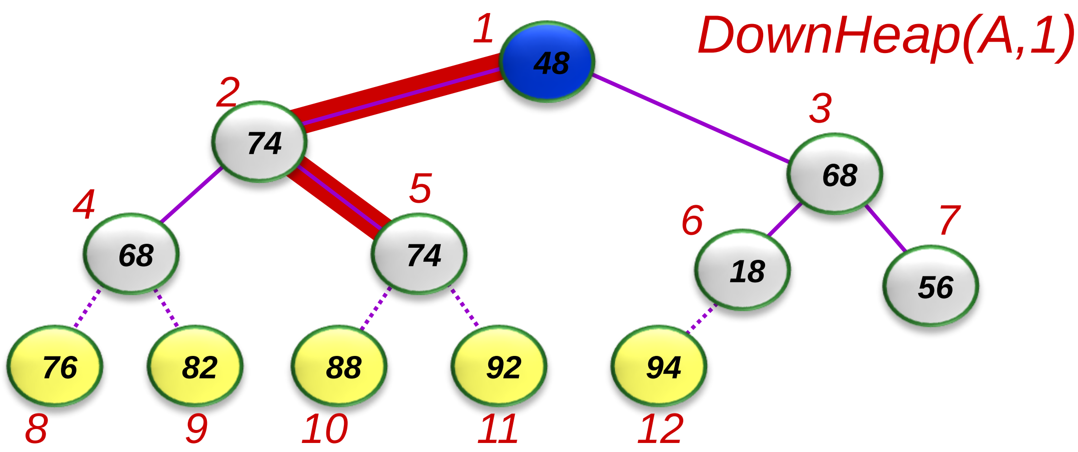
size[A] = 8



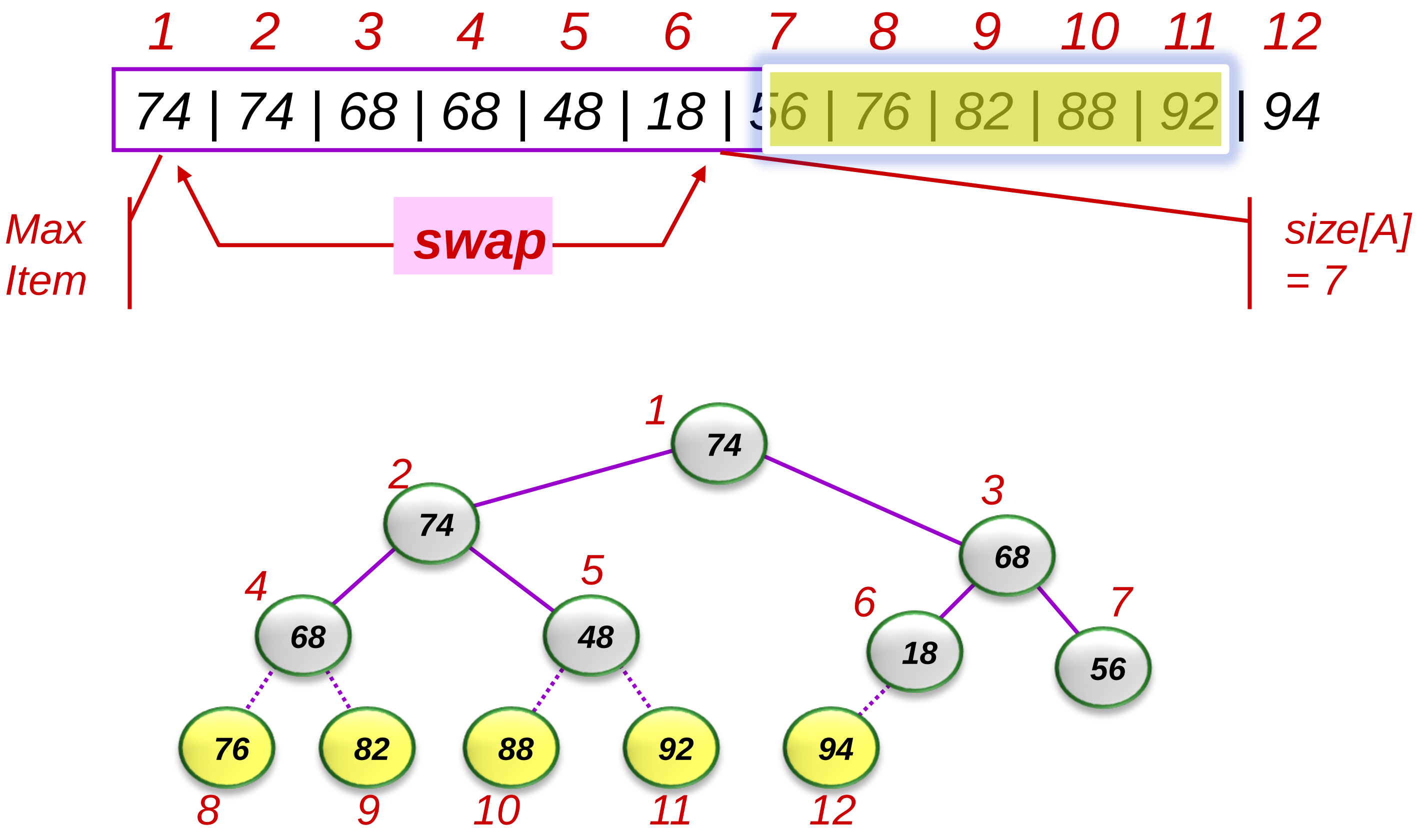
HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
48	74	68	68	74	18	56	76	82	88	92	94

size[A] = 7



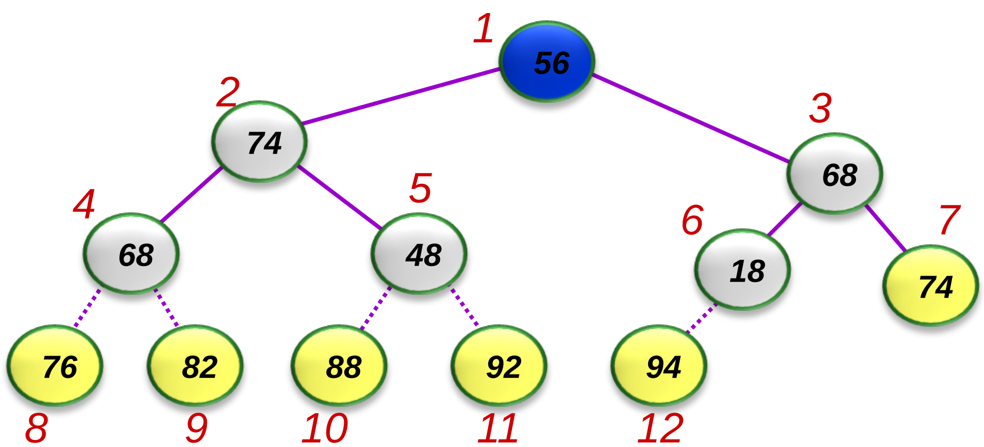
HeapSort Example



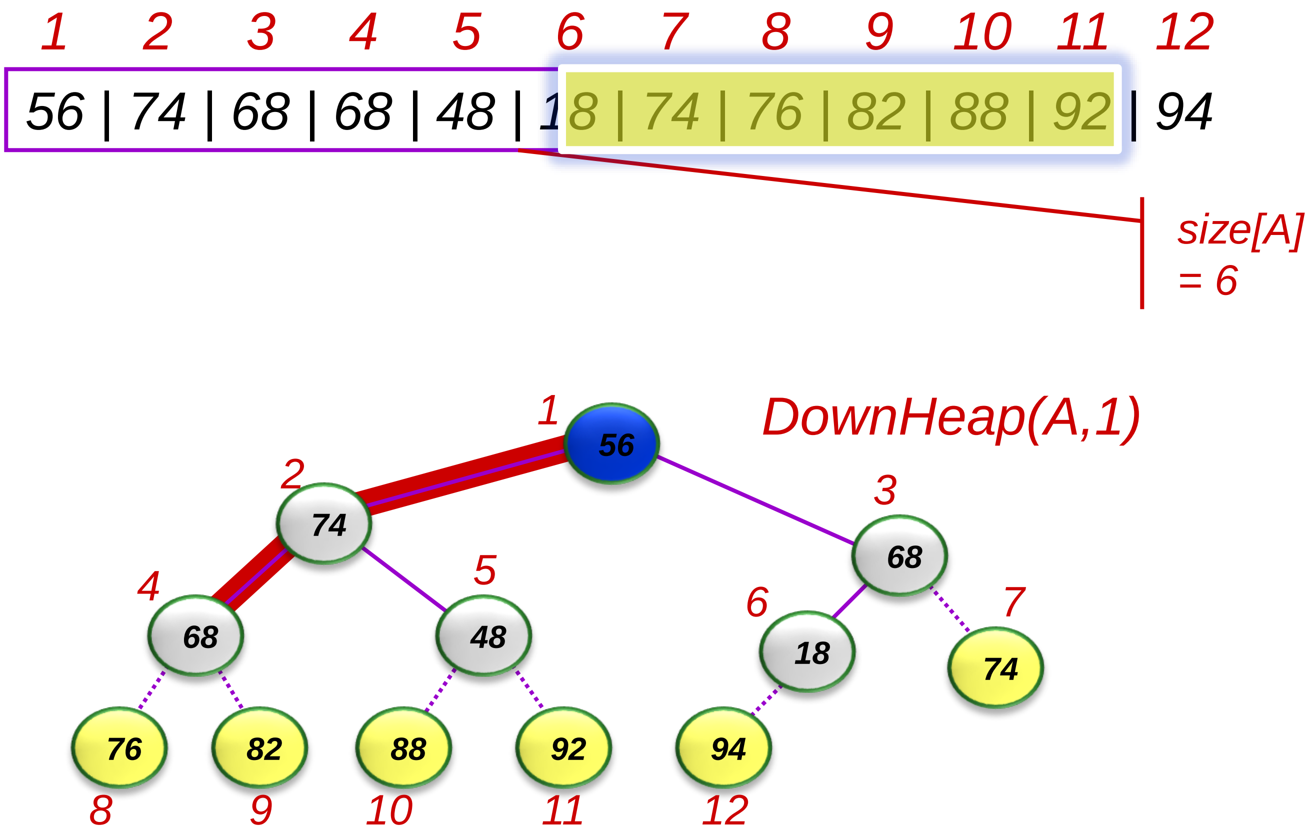
HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
56	74	68	68	48	18	74	76	82	88	92	94

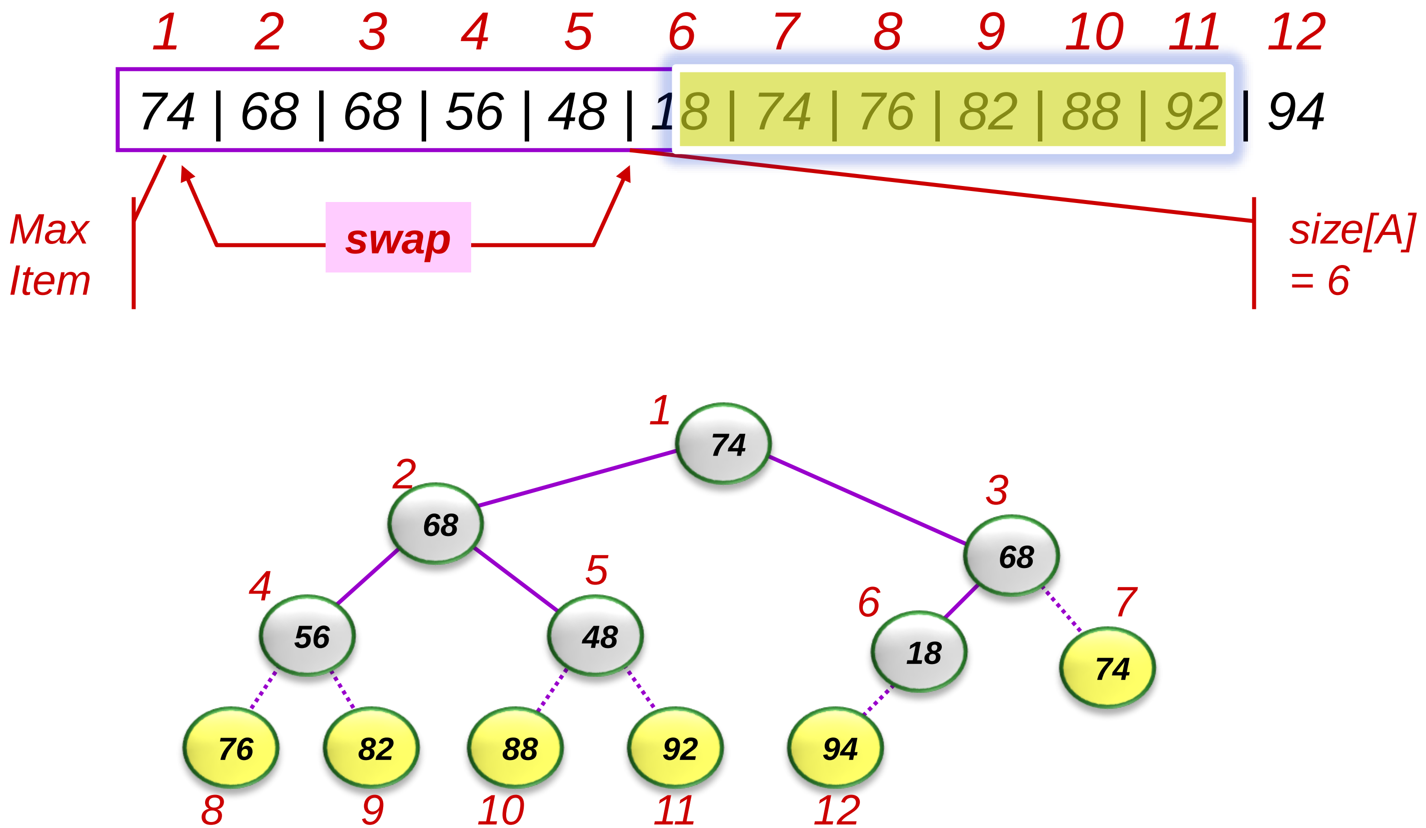
size[A]
= 7



HeapSort Example



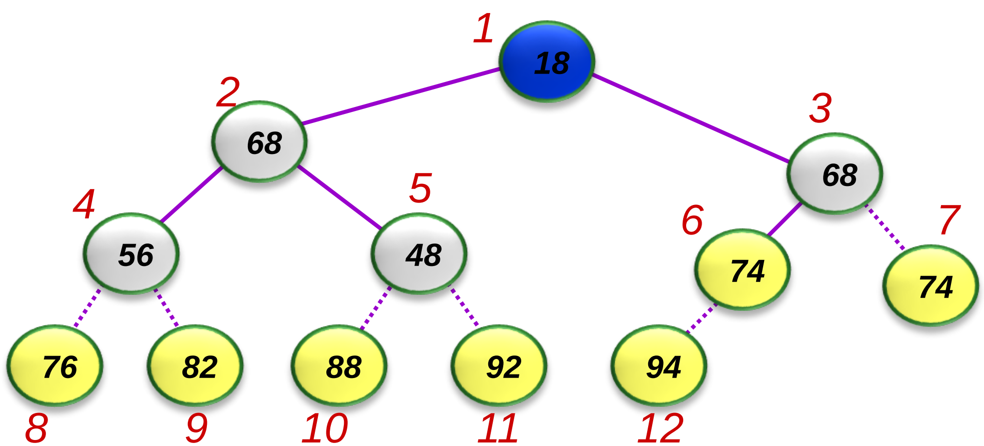
HeapSort Example



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
18	68	68	56	48	74	74	76	82	88	92	94

size[A]
= 6

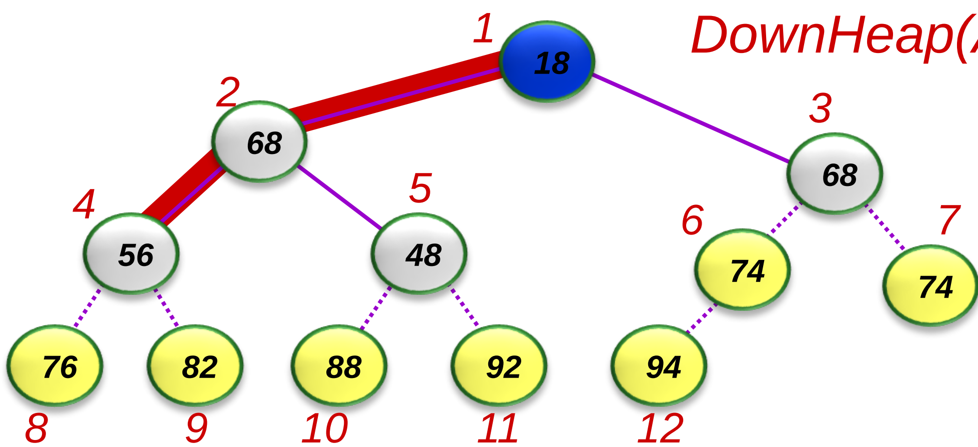


HeapSort Example

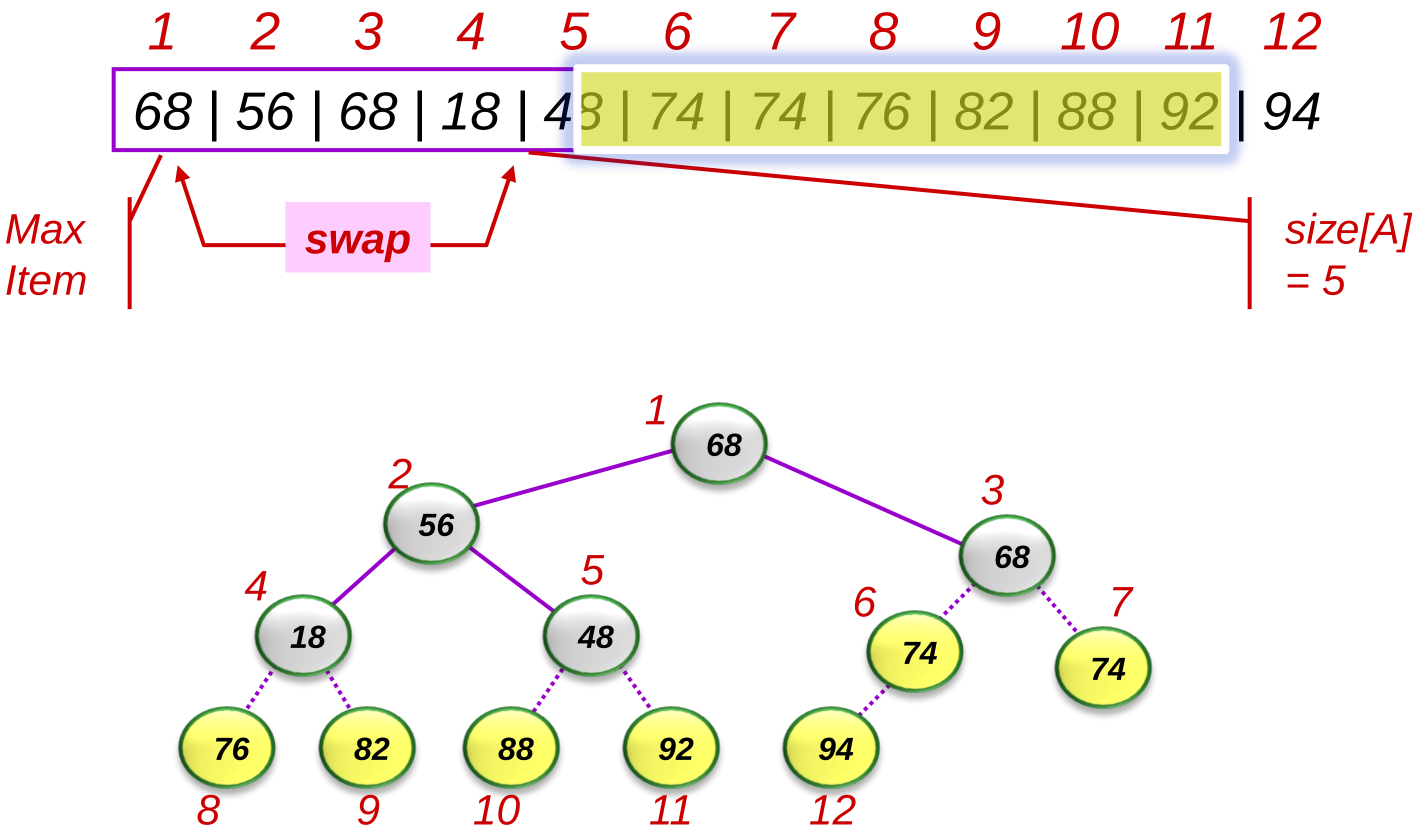
1	2	3	4	5	6	7	8	9	10	11	12
18	68	68	56	48	74	74	76	82	88	92	94

size[A]
= 5

DownHeap(A,1)



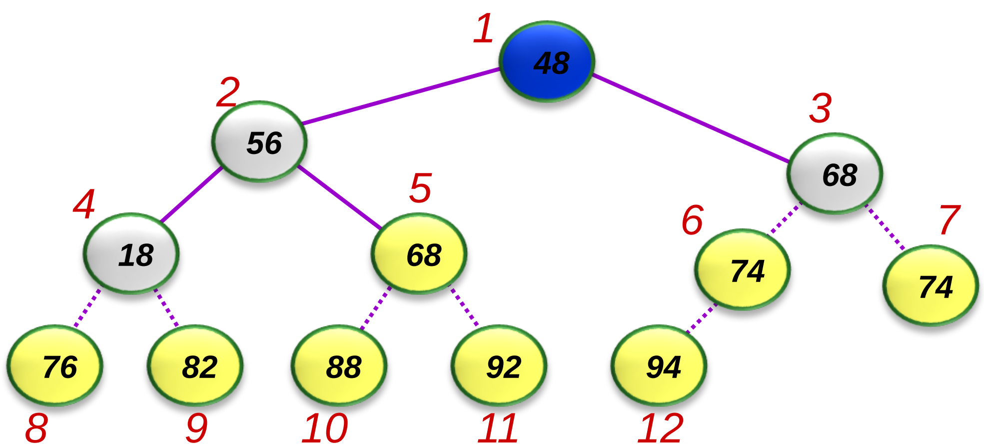
HeapSort Example



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
48	56	68	18	68	74	74	76	82	88	92	94

size[A]
= 5

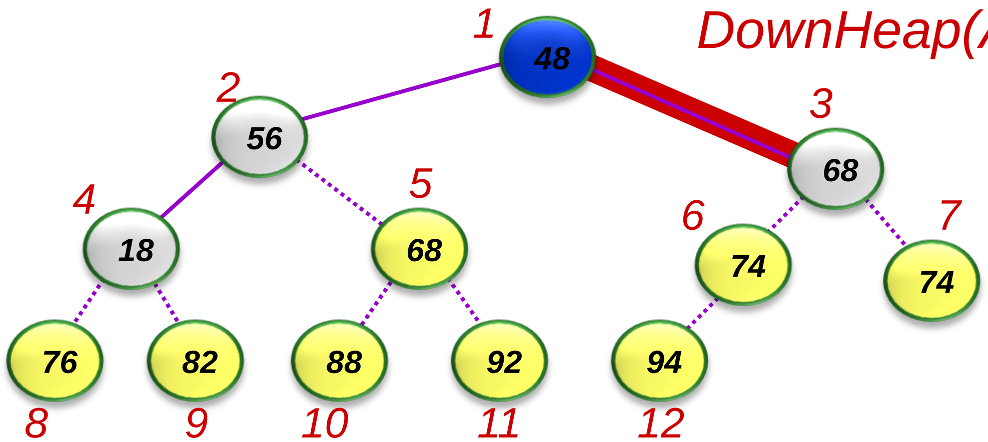


HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
48	56	68	18	68	74	74	76	82	88	92	94

size[A]
= 4

DownHeap(A,1)



HeapSort Example

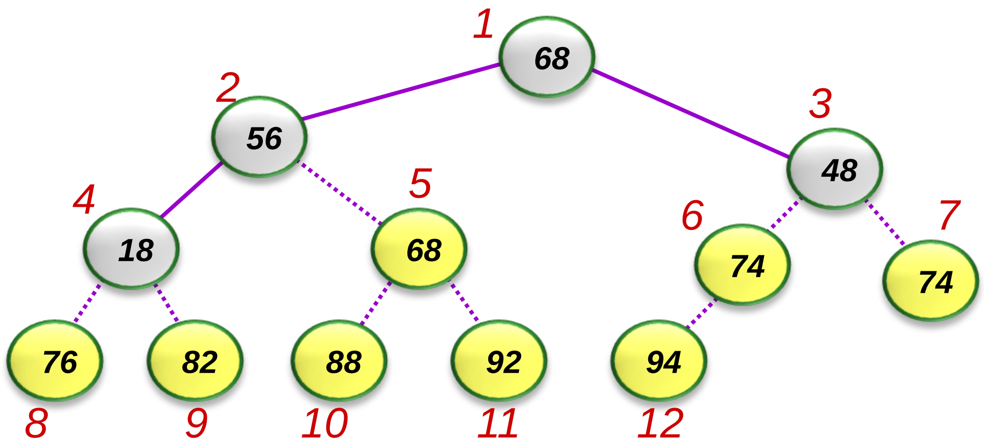
1 2 3 4 5 6 7 8 9 10 11 12

68 | 56 | 48 | 18 | 68 | 74 | 74 | 76 | 82 | 88 | 92 | 94

Max Item

swap

size[A] = 4

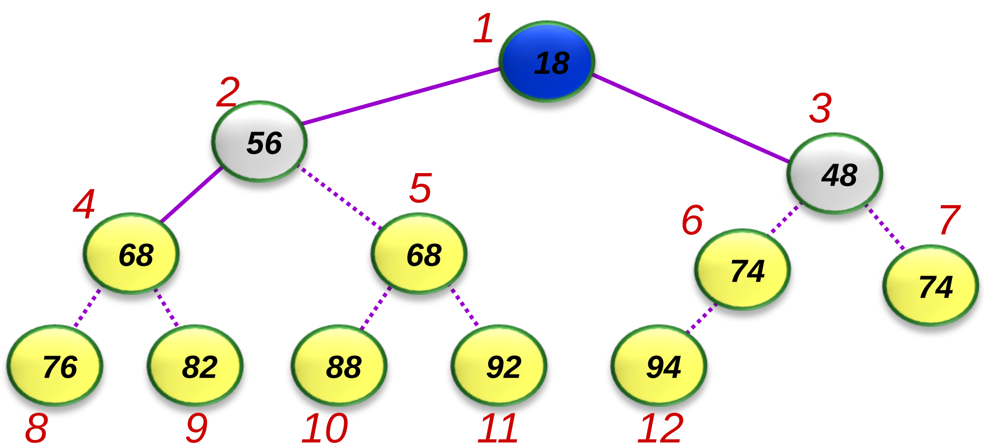


HeapSort Example

1 2 3 4 5 6 7 8 9 10 11 12

18 | 56 | 48 | 68 | 68 | 74 | 74 | 76 | 82 | 88 | 92 | 94

size[A] = 4

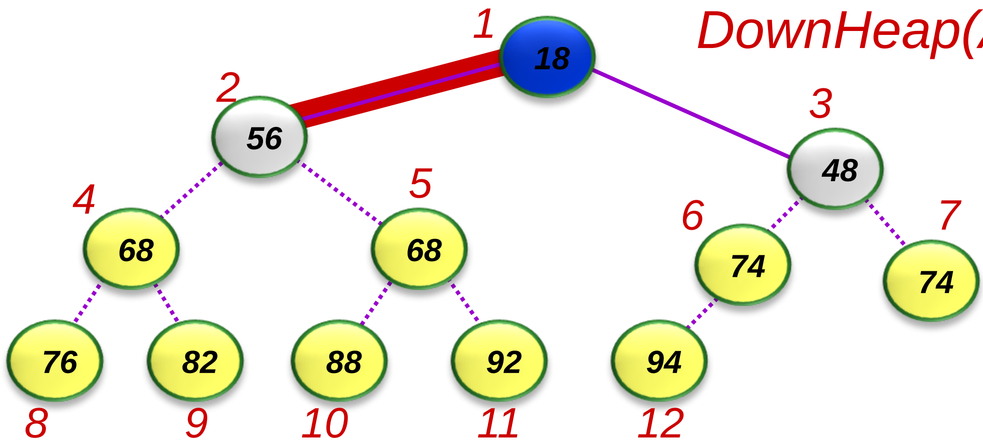


HeapSort Example

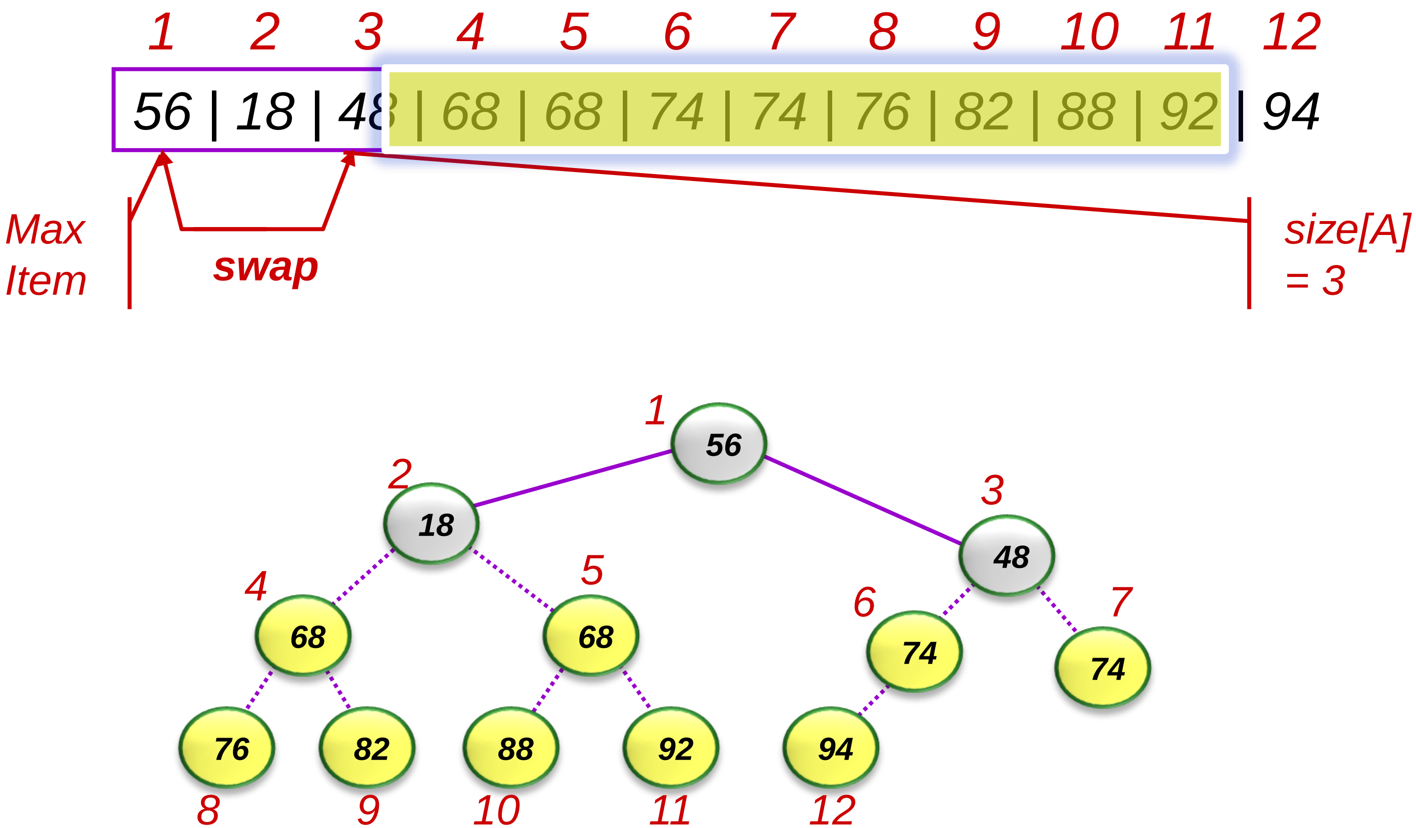
1	2	3	4	5	6	7	8	9	10	11	12
18	56	48	68	68	74	74	76	82	88	92	94

size[A] = 3

DownHeap(A,1)



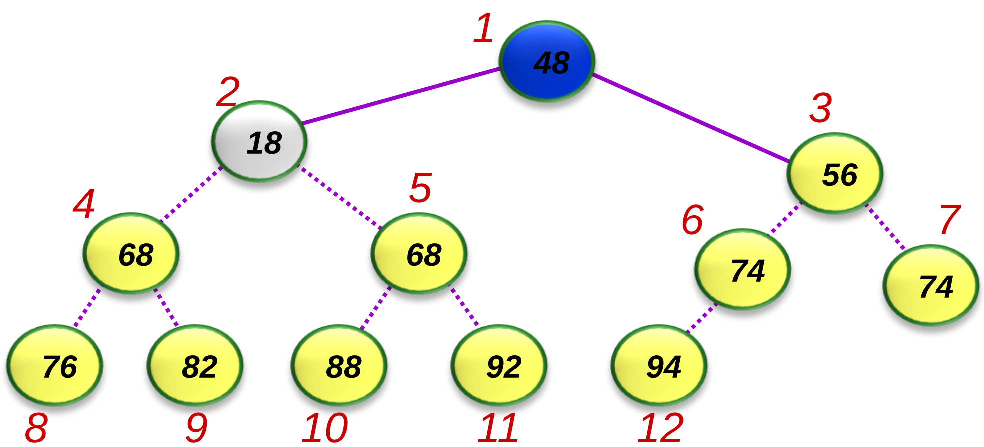
HeapSort Example



HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
48	18	56	68	68	74	74	76	82	88	92	94

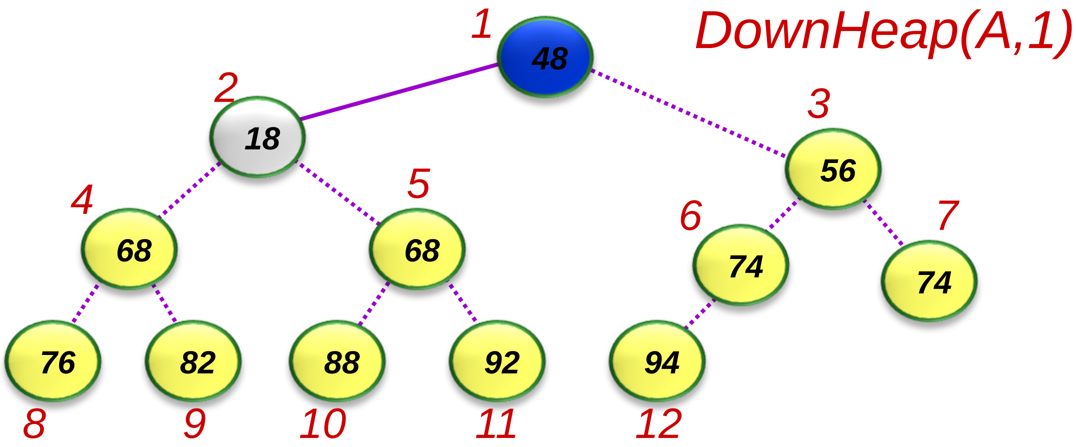
size[A] = 3



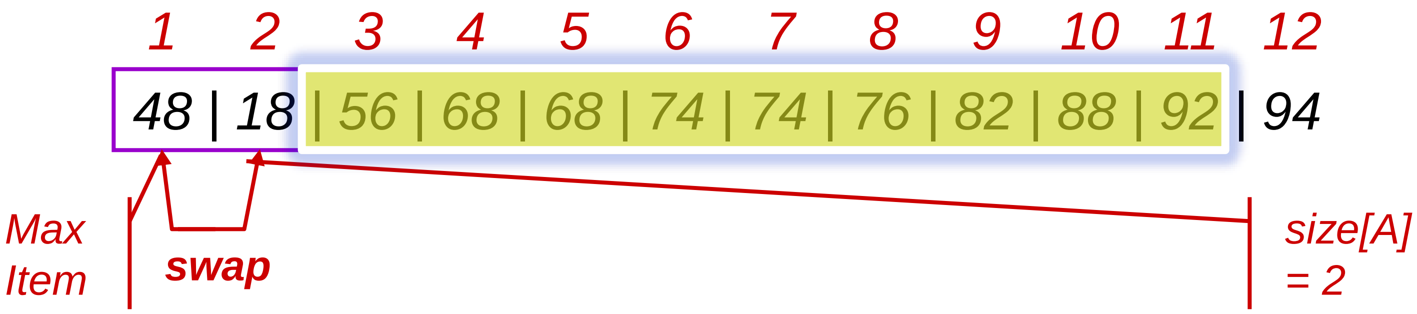
HeapSort Example

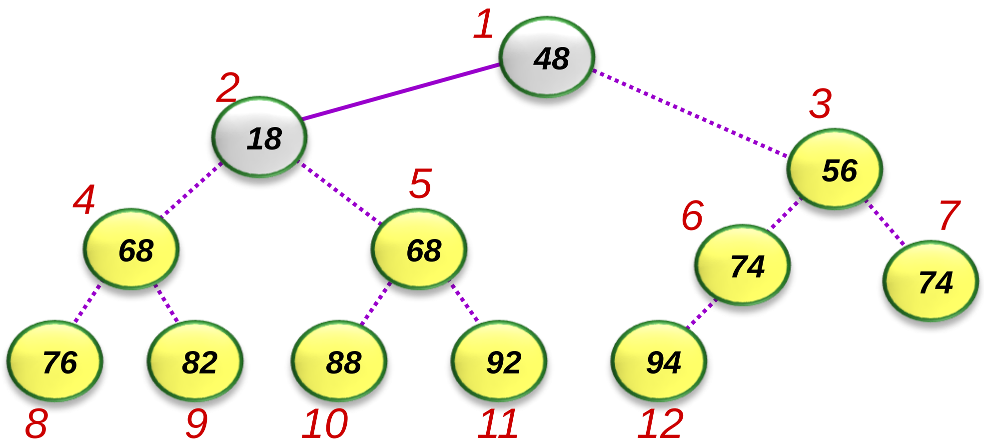
1	2	3	4	5	6	7	8	9	10	11	12
48	18	56	68	68	74	74	76	82	88	92	94

size[A] = 2



HeapSort Example

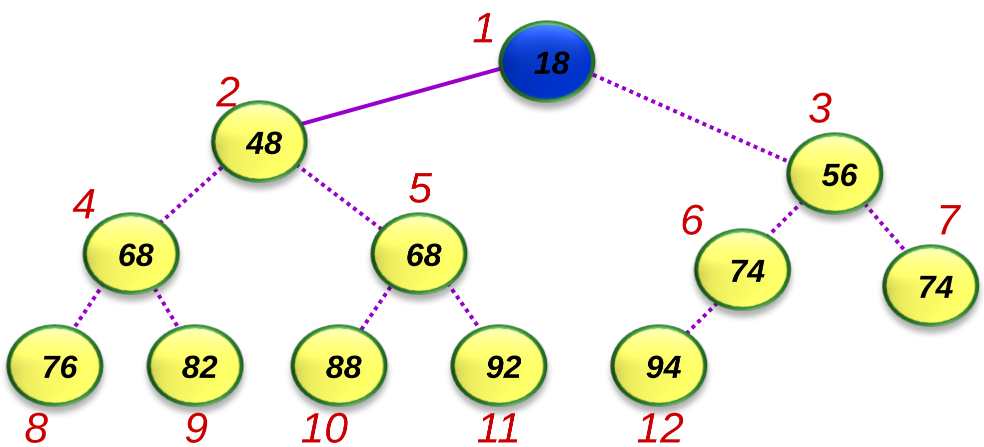




HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
18	48	56	68	68	74	74	76	82	88	92	94

size[A] = 2

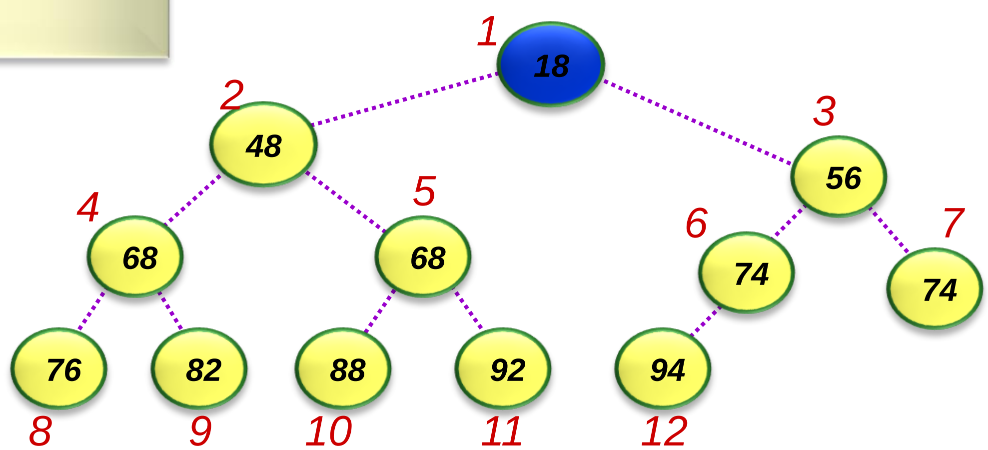


HeapSort Example

1	2	3	4	5	6	7	8	9	10	11	12
18	48	56	68	68	74	74	76	82	88	92	94

size[A] = 1

SORTED ARRAY



Heap as a Priority Queue

A ***Priority Queue*** *(usually implemented with some "heap" structure) is an abstract Data Structure that maintains a set S of items and supports the following operations on it:*

MakeEmptyHeap(S): Make an empty priory queue and call it S.

ConstructHeap(S): Construct a priority queue containing the set S of items.

Insert(x, S): Insert new item x into S (duplicate values allowed)

DeleteMax(S): Remove and return the maximum item from S.

Note: Min-Heap is used if we intend to do DeleteMin instead of DeleteMax.

Priority Queue Operations

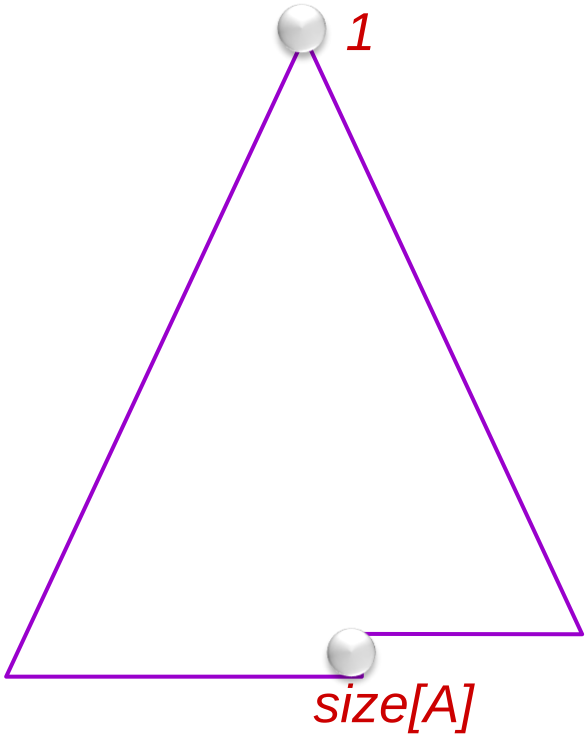
Array A as a binary heap is a suitable implementation.

For a heap of size n, it has the following time complexities:

O(1)	*MakeEmptyHeap(A)*	*size[A]* $\leftarrow$ *0*
O(n)	*ConstructHeap(A[1..n])*	*discussed already*
O(log n)	*Insert(x,A) and DeleteMax(A)*	*see below*

procedure ***Insert***(*x, A*)
  size[A] $\leftarrow$ *size[A] + 1*
  A[size[A]] $\leftarrow$ *x*
  UpHeap(A, size[A])
end

procedure ***DeleteMax***(*A*)
  if *size[A] = 0* ***then return*** *error*
  MaxItem $\leftarrow$ *A[1]*
  A[1] $\leftarrow$ *A[size[A]]*
  size[A] $\leftarrow$ *size[A] – 1*
  DownHeap(A, 1)
  return *MaxItem*
end



Sorting So Far

- Insertion sort:
  - Easy to code
  - Fast on small inputs (less than ~50 elements)
  - Fast on nearly-sorted inputs
  - $O(n^2)$ worst case
  - $O(n^2)$ average (equally-likely inputs) case
  - $O(n^2)$ reverse-sorted case

Sorting So Far

- Merge sort:
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort subarrays
    - Linear-time merge step
  - O(n lg n) worst case
  - Doesn’t sort in place

Sorting So Far

- Heap sort:
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children's keys
  - O(n lg n) worst case
  - Sorts in place
  - Fair amount of shuffling memory around

From lecture by David Luebke

Sorting So Far

- Quick sort:
  - Divide-and-conquer:
    - Partition array into two subarrays, recursively sort
    - All of first subarray < all of second subarray
    - No merge step needed!
  - O(n lg n) average case
  - Fast in practice
  - O(n^2) worst case

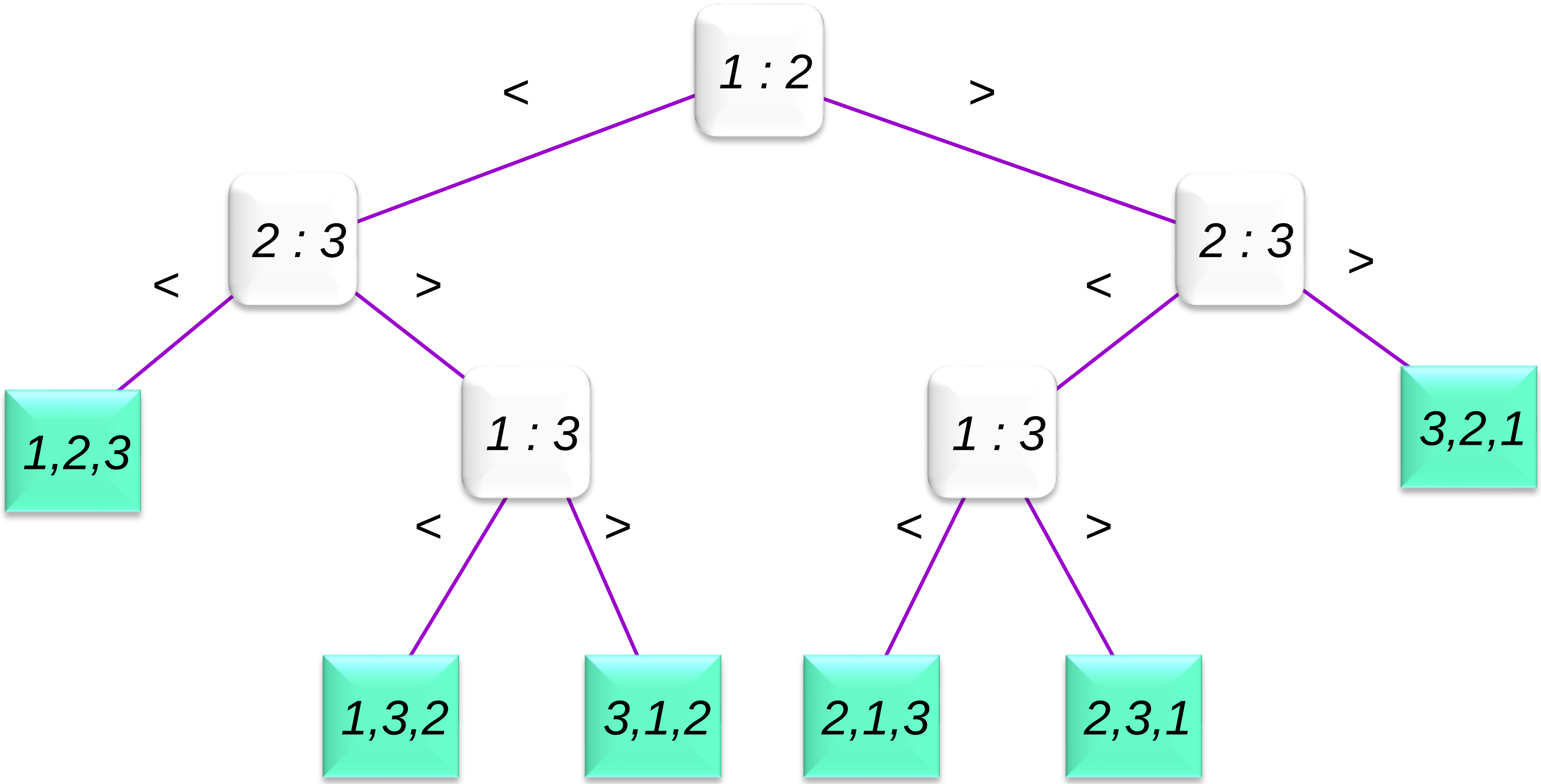
How Fast Can We Sort?

- We will provide a lower bound, then beat it
  - *How do you suppose we'll beat it?*
- First, an observation: all of the sorting algorithms so far are *comparison sorts*
  - The only operation used to gain ordering information about a sequence is the pairwise comparison of two elements
  - Theorem: all comparison sorts are $\Omega(n \lg n)$

Decision Trees

- *Decision trees* provide an abstraction of comparison sorts
  - A decision tree represents the comparisons made by a comparison sort. Every thing else ignored
  - (Draw examples on board)
- *What do the leaves represent?*
- *How many leaves must there be?* ***HomeWork***

Decision Trees



Lower Bound For Comparison Sorting

- Thm: Any decision tree that sorts n elements has height $\Omega(n \lg n)$
- *What's the minimum # of leaves?*
- *What's the maximum # of leaves of a binary tree of height h?*
- Clearly the minimum # of leaves is less than or equal to the maximum # of leaves

Lower Bound For Comparison Sorting

- So we have…
  $n! \le 2^h$
- Taking logarithms:
  $\lg(n!) \le h$
- Stirling's approximation tells us:
  $$n! > \left(\frac{n}{e}\right)^n$$
- Thus: $h \ge \lg\left(\frac{n}{e}\right)^n$

From lecture by David Luebke

Lower Bound For Comparison Sorting

- So we have

$$h \geq \lg\left(\frac{n}{e}\right)^n$$

$$= n \lg n - n \lg e$$

$$= \Omega(n \lg n)$$

- Thus the minimum height of a decision tree is $\Omega(n \lg n)$

From lecture by David Luebke

Lower Bound For Comparison Sorts

- Thus the time to comparison sort n elements is $\Omega(n \lg n)$
- Corollary: Heapsort and Mergesort are asymptotically optimal comparison sorts
- But do we have "Sorting in linear time"!
  - *How can we do better than $\Omega(n \lg n)$?*

Sorting In Linear Time

- Counting sort
  - No comparisons between elements!
  - ***But***…depends on assumption about the numbers being sorted
    - We assume numbers are in the range *1*.. k
  - The algorithm:
    - Input: A[1..n], where A[j] $\in$ {1, 2, 3, …, k}
    - Output: B[1..n], sorted (notice: not sorting in place)
    - Also: Array C[1..k] for auxiliary storage

Counting Sort

```
CountingSort(A, B, k)
    for i=1 to k
        C[i]= 0;
    for j=1 to n
        C[A[j]] += 1;
    for i=2 to k
        C[i] = C[i] + C[i-1];
    for j=n downto 1
        B[C[A[j]]] = A[j];
        C[A[j]] -= 1;
```

Work through example: A={4 1 3 4 3}, k = 4

From lecture by David Luebke

Counting Sort

```
CountingSort(A, B, k)
        for i=1 to k
                C[i]= 0;
        for j=1 to n
                C[A[j]] += 1;
        for i=2 to k
                C[i] = C[i] + C[i-1];
        for j=n downto 1
                B[C[A[j]]] = A[j];
                C[A[j]] -= 1;
```

Takes time O(k)

Takes time O(n)

What will be the running time?

From lecture by David Luebke

Counting Sort

- Total time: O(n + k)
  - Usually, k = O(n)
  - Thus counting sort runs in O(n) time
- But sorting is Ω(n lg n)!
  - No contradiction--this is not a comparison sort (in fact, there are *no* comparisons at all!)
  - Notice that this algorithm is *stable*

Counting Sort

- Cool! *Why don't we always use counting sort?*
- Because it depends on range k of elements
- *Could we use counting sort to sort 32 bit integers? Why or why not?*
- Answer: no, k too large ($2^{32} = 4{,}294{,}967{,}296$)

Multiplication of large integers

- a , b are both n-digit integers

- If we use the brute-force approach to compute c = a * b, what is the time efficiency?

n-bit Integer Addition vs Multiplication

$$x = x_{n-1}x_{n-2} \cdots x_1x_0 \qquad \text{compute} \quad x + y$$
$$y = y_{n-1}y_{n-2} \cdots y_1y_0 \qquad \text{and} \quad x * y$$

Proof: *A correct algorithm must "look" at every input bit.*
Suppose on non-zero inputs, input bit b is not looked at by the algorithm.
Adversary gives the algorithm the same input, but with bit b flipped.
Algorithm is oblivious to b, so it will give the same answer.
It can't be correct both times!

Elementary School Addition Algorithm has O(n) bit-complexity:

```
   XXXXXXXXXX
 + XXXXXXXXXX
 ------------
  XXXXXXXXXXX
```

The bit-complexity of n-bit addition is $\Theta(n)$.

n-bit Integer Multiplication

$$x = x_{n-1}x_{n-2} \cdots x_1 x_0 \qquad \text{compute} \qquad x + y$$
$$y = y_{n-1}y_{n-2} \cdots y_1 y_0 \qquad \text{and} \qquad x * y$$

Elementary School Multiplication Algorithm has $O(n^2)$ bit-complexity:

```
      XXXX
    * XXXX
  ----------
     XXXXX
    XXXXX
   XXXXX
  XXXXX
  ----------
 XXXXXXXXX
```

Multiplication of large integers (divide-conquer recursive algorithm I)

- $a = a_1 a_0$ and $b = b_1 b_0$
- $c = a * b$

$= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$

$= (a_1 * b_1) 10^n + (a_1 * b_0 + a_0 * b_1) 10^{n/2} + (a_0 * b_0)$

$$T(n) = 4T(n/2) + \Theta(n) \quad \Rightarrow T(n) = \Theta\left(n^{\log_2 4}\right) = \Theta\left(n^2\right)$$

Example

$X = 3141$
$Y = 5927$

$X*Y = 18{,}616{,}707$

$X = 3100 + 41 \ = a_1 \bullet 100 + a_0$
$Y = 5900 + 27 \ = \ b_1 \bullet 100 + b_0$

$$
\begin{aligned}
X*Y &= (a_1 * b_1) \bullet 10000 + (a_1 * b_0 + a_0 * b_1) \bullet 100 + a_0 * b_0 \\
&= (31*59) \bullet 10000 + (31*27 + 41*59) \bullet 100 + 41*27 \\
&= 1829 \bullet 10000 + (837 + 2419) \bullet 100 + 1107 \\
&= 1829 \bullet 10000 + 3256 \bullet 100 + 1107 \\
&= 18290000 + 325600 + 1107 \\
&= 18{,}616{,}707
\end{aligned}
$$

Multiplication of large integers (divide-conquer recursive algorithm II)

- $a = a_1 a_0$ and $b = b_1 b_0$

- $c = a * b$

  $=(a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$

  $=c_2 10^n + c_1 10^{n/2} + c_0,$

where

$c_2 = a_1 * b_1$ is the product of their first halves

$c_0 = a_0 * b_0$ is the product of their second halves

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0.

$X = 3141$
$Y = 5927$

$X*Y = 18{,}616{,}707$

$X = 3100 + 41 \;= a_1 \bullet 100 + a_0$
$Y = 5900 + 27 \;= \; b_1 \bullet 100 + b_0$

$c_1 = (a_1 + a_0)*(b_1 + b_0) = (31+41)*(59+27) = 72 * 86 = 6{,}192$

$c_2 = a_1 * b_1 = 31 * 59 = 1{,}829$

$c_0 = a_0 * b_0 = 41 * 27 = 1{,}107$

$X*Y = c_2 \bullet 10000 + (c_1 - c_2 - c_0) \bullet 100 + c_0$

$= 1829 \bullet 10000 + (6192 - 1829 - 1107) \bullet 100 + 1107$

$= 18{,}616{,}707$

Multiplication of large integers

$$T(n) = 3T(n/2) + \Theta(n) \qquad \Rightarrow T(n) = \Theta\left(n^{\log_2 3}\right) = \Theta\left(n^{1.585\cdots}\right)$$

Complexity of n-bit Integer Multiplication

Known Upper Bounds:

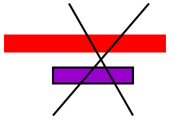
- $O(n^{\log 3}) = O(n^{1.59})$ by divide-&-conquer [Karatsuba-Ofman, 1962]
- $O(n \log n \log\log n)$ by FFT [Schönhage-Strassen, 1971]
- $n \log n\, 2^{O(\log^* n)}$ [Martin Fürer, 2007]

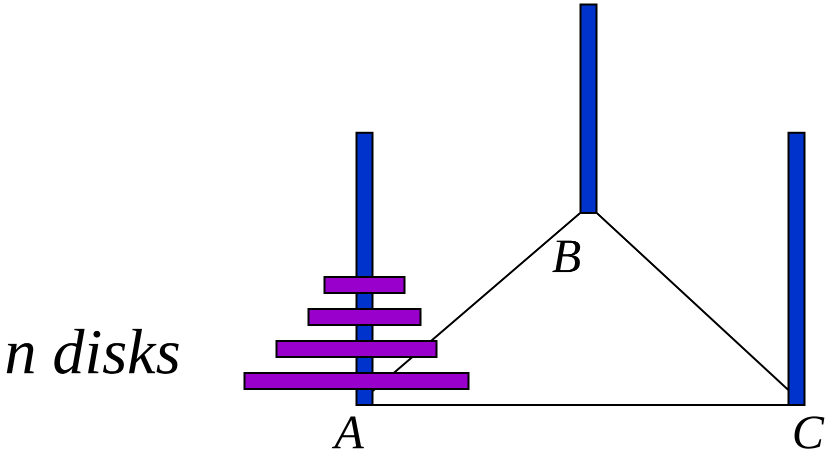
Known Lower Bound:

- $\Omega(n \log n / \log\log n)$ [Fischer-Meyer, 1974]

EXAMPLE: tower of hanoi

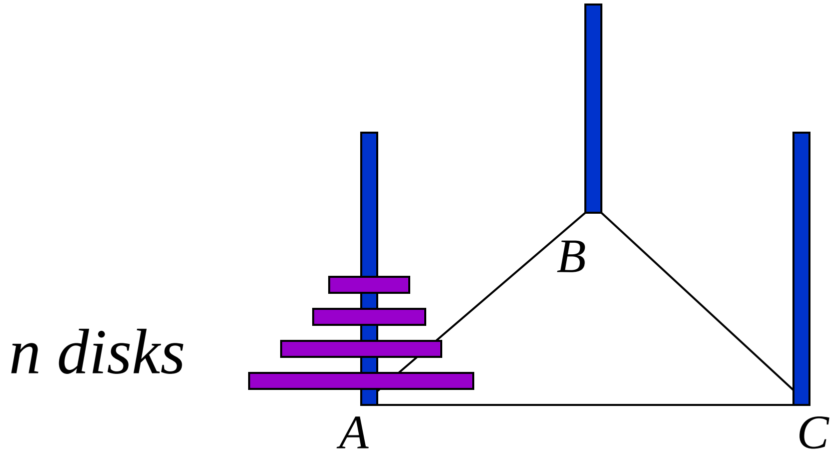
- Problem:
  - Given three pegs (A, B, C) and n disks of different sizes
  - Initially, all the disks are on peg A in order of size, the largest on the bottom and the smallest on top
  - The goal is to move all the disks to peg C using peg B as an auxiliary
  - Only 1 disk can be moved at a time, and a larger disk cannot be placed on top of a smaller one

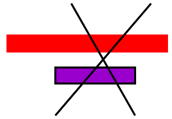




EXAMPLE: tower of hanoi

- Design a recursive algorithm to solve this problem:
  - Given three pegs (A, B, C) and n disks of different sizes
  - Initially, all the disks are on peg A in order of size, the largest on the bottom and the smallest on top
  - The goal is to move all the disks to peg C using peg B as an auxiliary
  - Only 1 disk can be moved at a time, and a larger disk cannot be placed on top of a smaller one

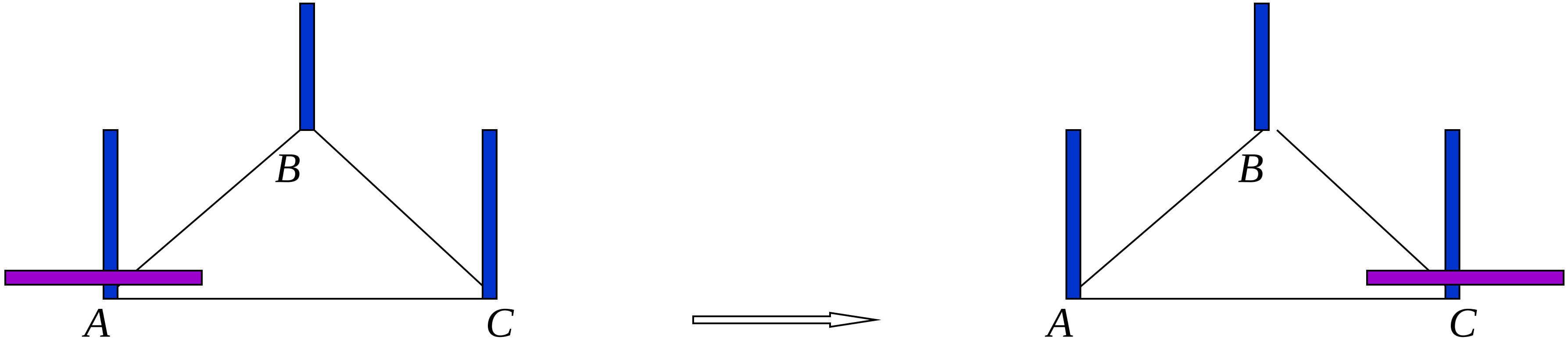




EXAMPLE: tower of hanoi

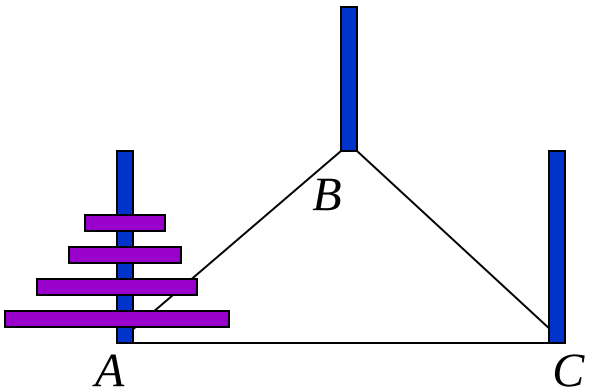
- Solve simple case when n<=1?

  Just trivial

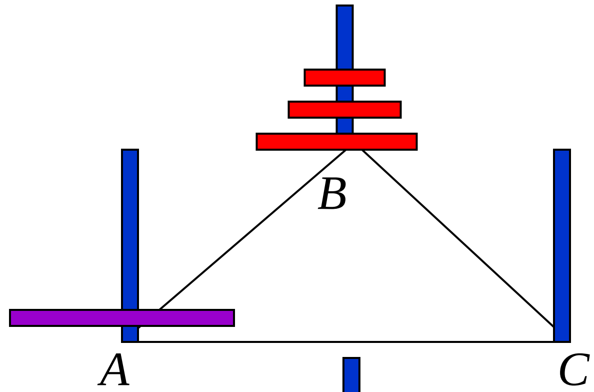


Move(A, C)

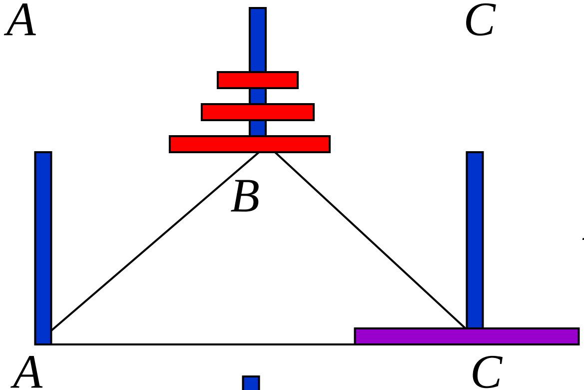
EXAMPLE: tower of hanoi



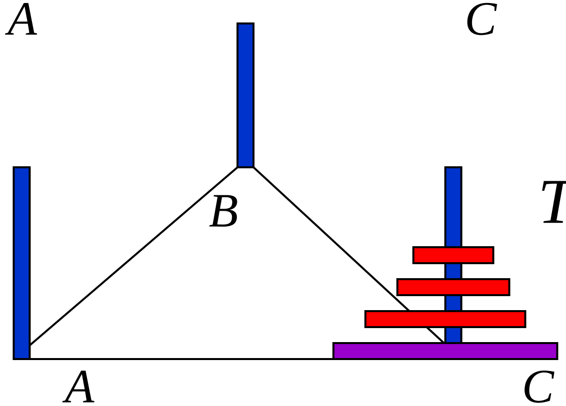
TOWER(n, A, B, C)



TOWER(n-1, A, C, B)



Move(A, C)



TOWER(n-1, B, A, C)

EXAMPLE: tower of hanoi

```
TOWER(n, A, B, C)  {
 if n<1 return;
TOWER(n-1, A, C, B);
Move(A, C);
TOWER(n-1, B, A, C)
}
```

Growth-Rate Functions – Recursive Algorithms

```
void hanoi(int n, char source, char dest, char spare) {      Cost
  if (n > 0) {                                                 c1
    hanoi(n-1, source, spare, dest);                           c2
     cout << "Move top disk from pole " << source              c3
          << " to pole " << dest << endl;
    hanoi(n-1, spare, dest, source);                           c4
} }
```

- The time-complexity function T(n) of a recursive algorithm is defined in terms of itself, and this is known as **recurrence equation** for T(n).
- To find the growth-rate function for a recursive algorithm, we have to solve its recurrence relation.

Growth-Rate Functions – Hanoi Towers

- What is the cost of `hanoi(n,'A','B','C')`?

when n=0

$T(0) = c1$

when n>0

$$
\begin{aligned}
T(n) &= c1 + c2 + T(n-1) + c3 + c4 + T(n-1) \\
&= 2*T(n-1) + (c1+c2+c3+c4) \\
&= \mathbf{2*T(n-1) + c} \quad \leftarrow \text{recurrence equation for the growth-rate function of hanoi-towers algorithm}
\end{aligned}
$$

- Now, we have to solve this recurrence equation to find the growth-rate function of hanoi-towers algorithm

Growth-Rate Functions – Hanoi Towers (cont.)

- There are many methods to solve recurrence equations, but we will use a simple method known as *repeated substitutions*.

$T(n) = 2*T(n-1) + c$

$= 2 * (2*T(n-2)+c) + c$

$= 2 * (2* (2*T(n-3)+c) + c) + c$

$= 2^3 * T(n-3) + (2^2+2^1+2^0)*c$ (assuming n>2)

when substitution repeated i-1th times

$= 2^i * T(n-i) + (2^{i-1}+ \ldots +2^1+2^0)*c$

when i=n

$= 2^n * T(0) + (2^{n-1}+ \ldots +2^1+2^0)*c$

$= 2^n * c1 + (\sum_{i=0}^{n-1} 2^i)*c$

$= 2^n * c1 + (2^n-1)*c = 2^n*(c1+c) - c$ ➔ So, the growth rate function is **$O(2^n)$**

The End