CS 311: Algorithm Design and Analysis

Lecture 5

Last Lecture we have

- Merge sort again !!!
- Recurrence
- Methods of solving recurrences
  - Substitution Method
  - Master Method
  - Recursion Tree

This Lecture we have

- Recursion Tree
- Quick Sort
- Heap Sort

Recursion Tree

See the example in page 110

Tree → summation

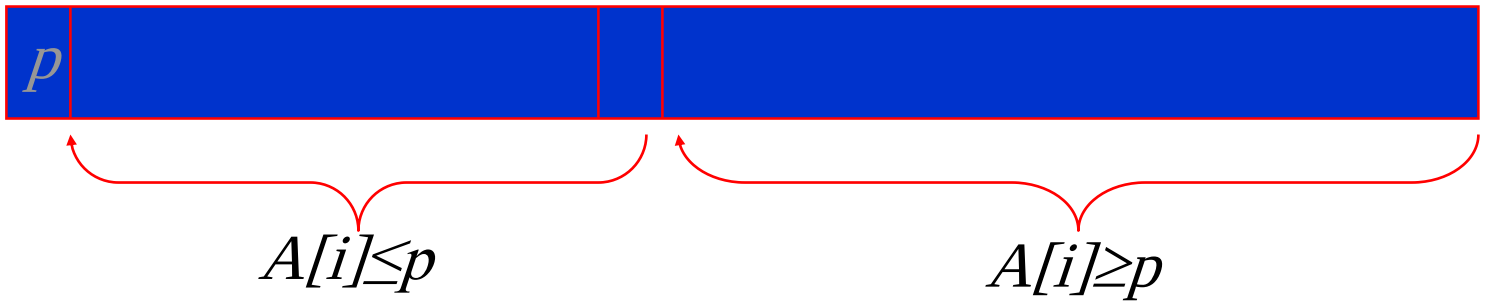
Summation → The 1st item is grater than the last one

Summation → The 1st item is smaller than the last one

Summation → The item are equal

Quicksort by Hoare (1962)

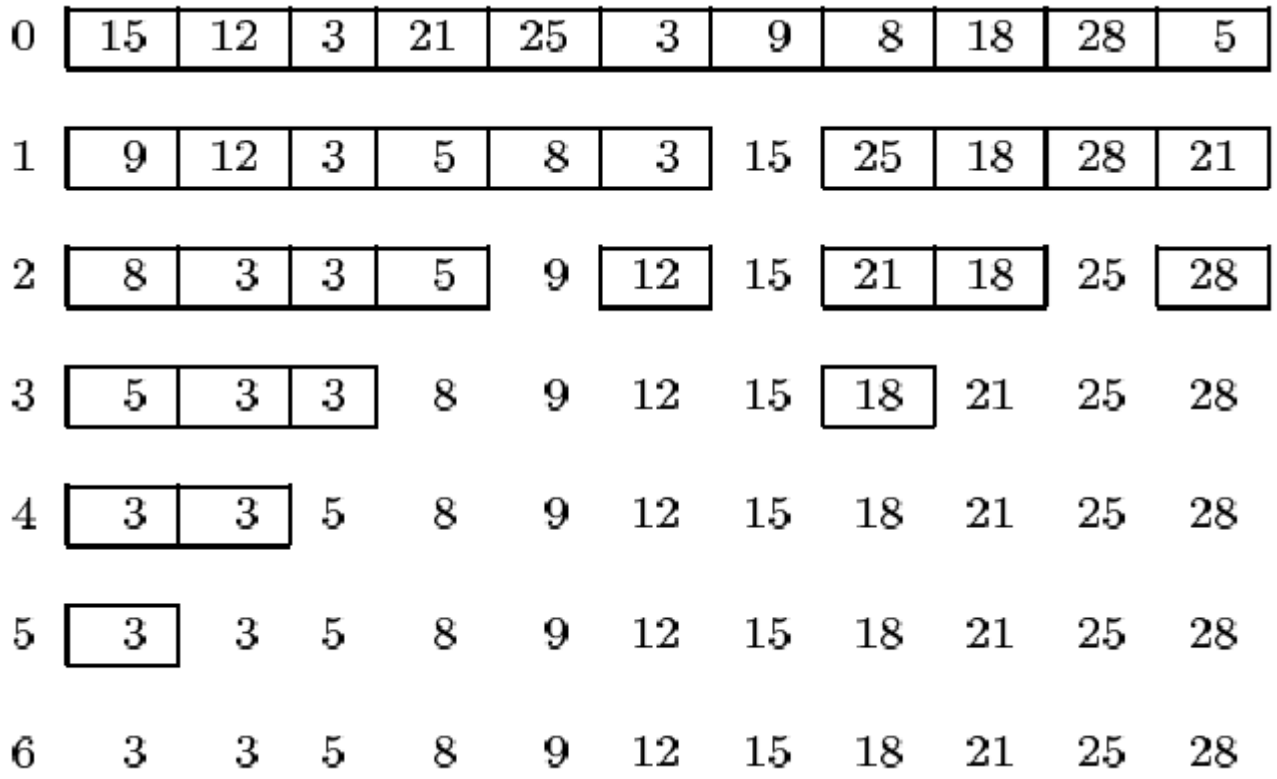
- Select a *pivot* (partitioning element)
- Rearrange the list so that all the elements in the positions before the pivot are smaller than or equal to the pivot and those after the pivot are larger than or equal to the pivot
- Exchange the pivot with the last element in the first (i.e., ≤) sublist – the pivot is now in its final position
- Sort the two sublists recursively



- Apply quicksort to sort the list 7 2 9 10 5 4

Quicksort Example

- Recursive implementation with the left most array entry selected as the pivot element.



Quicksort Algorithm

- Input:
  - Array E and indices first, and last, s.t. elements E[i] are defined for first ≤ i ≤ last
- Ouput:
  - E[first], …, E[last] is a sorted rearrangement of the array

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q-1);
        Quicksort(A, q+1, r);
    }
}
```

Partition In Words

- Partition(A, p, r):
  - Select an element to act as the "pivot" (*which?*)
  - Grow two regions, A[p..i] and A[j..r]
    - All elements in A[p..i] <= pivot
    - All elements in A[j..r] >= pivot
  - Increment i until A[i] >= pivot
  - Decrement j until A[j] <= pivot
  - Swap A[i] and A[j]
  - Repeat until i >= j
  - Return j

From Lecture by David Luebke

Partition Code

```
Partition(A, p, r)
    x = A[p];
    i = p - 1;
    j = r + 1;
    while (TRUE)
        repeat
            j--;
        until A[j] <= x;
        repeat
            i++;
        until A[i] >= x;
        if (i < j)
            Swap(A, i, j);
        else
            return j;
```

Illustrate on
A = {5, 3, 2, 6, 4, 1, 3, 7};

What is the running time of ***partition()*** *?*

From Lecture by David Luebke

Quicksort Analysis

- Partition can be done in O(n) time, where n is the size of the array
- Let $T(n)$ be the number of comparisons required by Quicksort
- If the pivot ends up at position k, then we have
  - $T(n) = T(n-k) + T(k-1) + n$
- To determine best-, worst-, and average-case complexity we need to determine the values of k that correspond to these cases.

Best-Case Complexity

- The best case is clearly when the pivot always partitions the array equally.
- Intuitively, this would lead to a recursive depth of at most lg n calls
- We can actually prove this. In this case
  - $T(n) \approx T(n/2) + T(n/2) + n \Rightarrow \Theta(n \lg n)$

Worst-Case and Average-Case Complexity

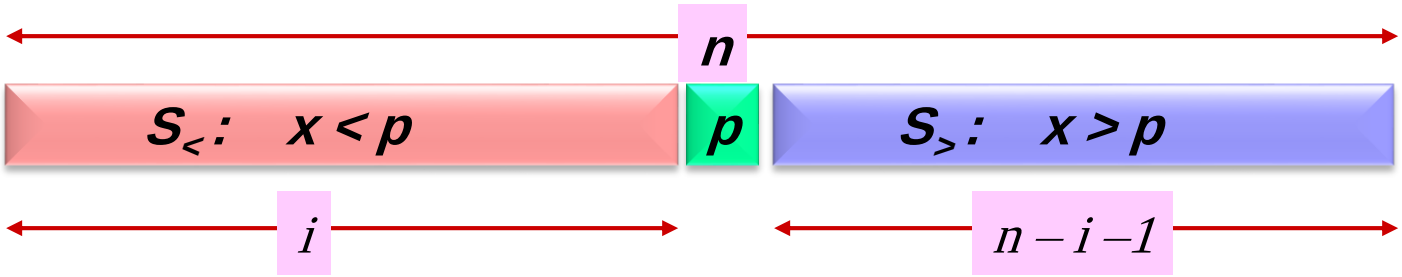
- The worst-case is when the pivot always ends up in the first or last element. That is, partitions the array as unequally as possible.
- In this case
  - $T(n) = T(n-1) + T(1-1) + n = T(n-1) + n$

    $= n + (n-1) + \ldots + 1$

    $= n(n+1)/2 \Rightarrow O(n^2)$
- Average case is rather complex, but is where the algorithm earns its name. The bottom line is:

$$A(n) \approx 1.386 n \lg n \Rightarrow \Theta(n \lg n)$$

QuickSort Average-Case



WLOG Assume: $|S_=| = 1$. If it's larger, it can only help!

$T(n) = T(i) + T(n-i-1) + \Theta(n)$, $T(n) = \Theta(1)$, for n=0,1.

Expected-Case:

$$T(n) = ave_i \{ T(i) + T(n-i-1) + \Theta(n) : \quad i = 0 .. n-1 \}$$

$$= \frac{1}{n} \sum_{i=0}^{n-1} [T(i) + T(n-i-1) + \Theta(n)]$$

$$= \frac{2}{n} \sum_{i=0}^{n-1} T(i) + \Theta(n)$$

$$= \Theta(n \log n)$$

QuickSort Average-Case

Example 2: $T(n)=\frac{2}{n}\sum_{i=0}^{n-1}T(i)+n, \quad \forall n\geq 0 \qquad [\Rightarrow \; T(0)=0]$

1. *Multiply across by n (so that we can subsequently cancel out the summation*
$$nT(n)=2\sum_{i=0}^{n-1}T(i)+n^2, \quad \forall n\geq 0$$
2. *Substitute n-1 for n:* $(n-1)T(n-1)=2\sum_{i=0}^{n-2}T(i)+(n-1)^2, \quad \forall n\geq 1$
3. *Subtract (2) from (1):* $nT(n)-(n-1)T(n-1)=2T(n-1)+2n-1, \quad \forall n\geq 1$
4. *Rearrange:* $nT(n)=(n+1)T(n-1)+2n-1, \quad \forall n\geq 1$
5. *Divide by n(n+1) to make LHS & RHS look alike:* $\frac{T(n)}{n+1}=\frac{T(n-1)}{n}+\frac{2n-1}{n(n+1)}, \quad \forall n\geq 1$
6. *Rename:* $Q(n)=\frac{T(n)}{n+1}, \quad Q(n-1)=\frac{T(n-1)}{n}, \quad \frac{2n-1}{n(n+1)}=\frac{3}{n+1}-\frac{1}{n}$
7. *Simplified recurrence:* $Q(n)=\begin{cases} Q(n-1)+\left[\frac{3}{n+1}-\frac{1}{n}\right] & \forall n\geq 1 \\ 0 & \text{for } n=0 \end{cases}$
8. *Iteration:* $Q(n)=\left(\frac{3}{n+1}-\frac{1}{n}\right)+\left(\frac{3}{n}-\frac{1}{n-1}\right)+\left(\frac{3}{n-1}-\frac{1}{n-2}\right)+\cdots+\left(\frac{3}{2}-\frac{1}{1}\right)+Q(0)=2H(n)-\frac{3n}{n+1}$
9. *Finally:* $T(n)=(n+1)Q(n)=2(n+1)H(n)-3n=\Theta(n\log n).$

n^{th} Harmonic number

Summary of Quicksort

- *Best case*: split in the middle — $\Theta(n \log n)$
- *Worst case*: sorted array! — $\Theta(n^2)$
- *Average case*: random arrays — $\Theta(n \log n)$

Summary of Quicksort

- *Best case*: split in the middle — $\Theta(n \log n)$
- *Worst case*: sorted array! — $\Theta(n^2)$
- *Average case*: random arrays — $\Theta(n \log n)$

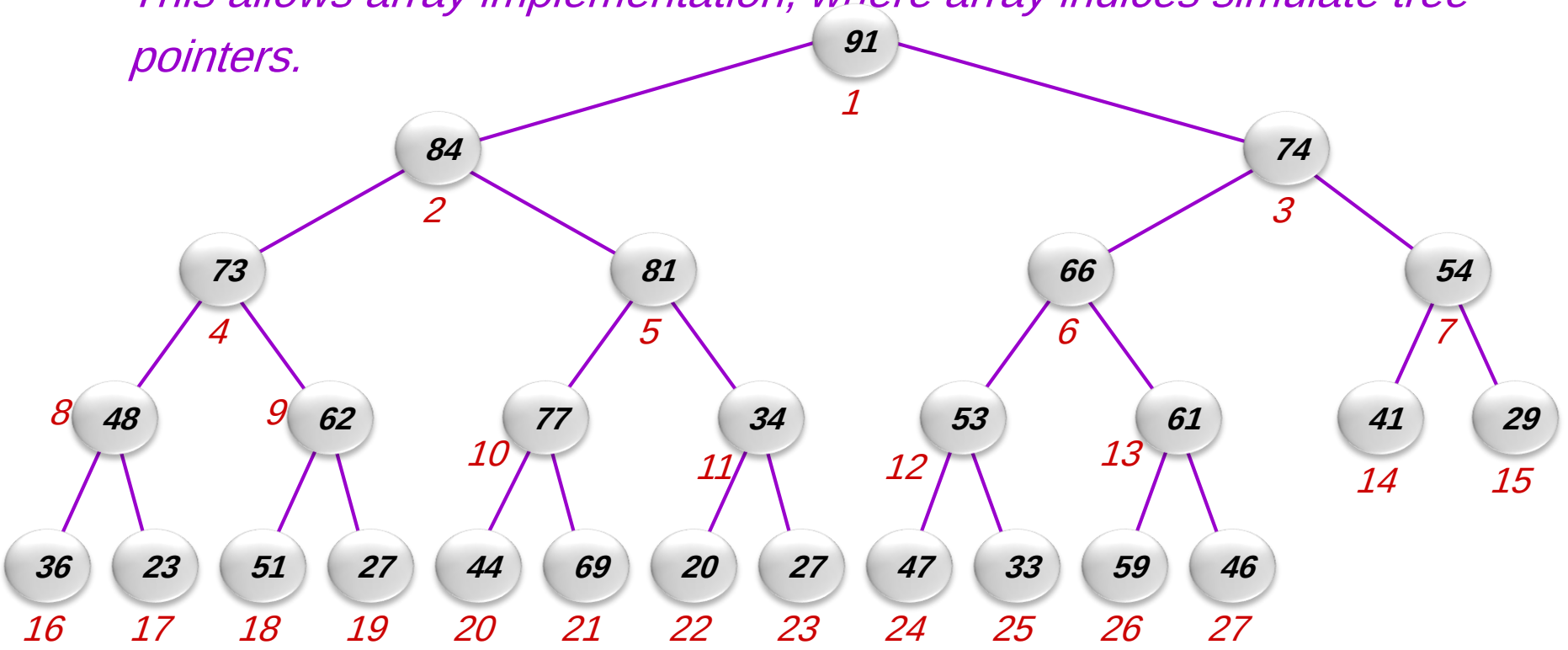
- Considered as the method of choice for internal sorting for large files ($n \geq 10000$)

- Improvements:
  - better pivot selection: median of three partitioning avoids worst case in sorted files
  - switch to insertion sort on small subfiles
  - elimination of recursion

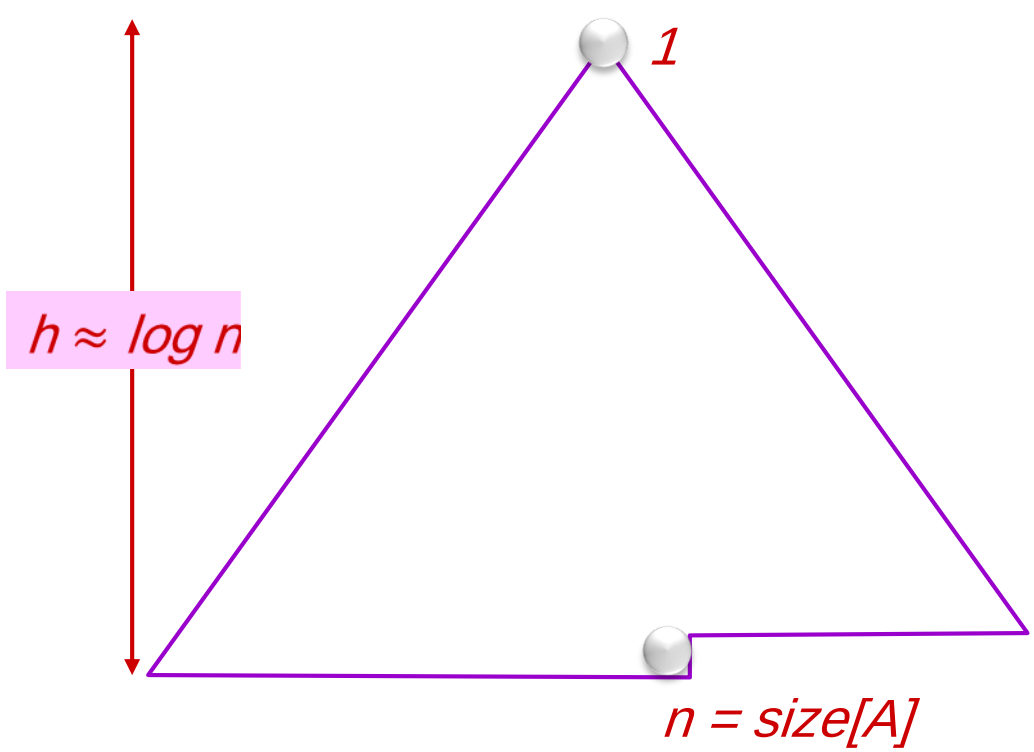
  these combine to 20-25% improvement

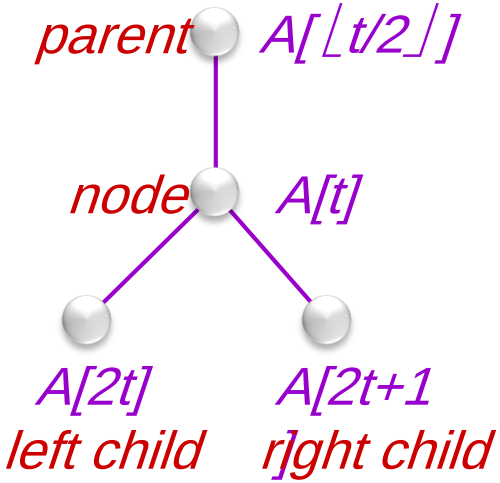
Binary Heap

- *A = a binary tree with one key per node.*
- *Max Heap Order: A satisfies the following partial order:*
  for every node $x \neq root[A]$: $key[x] \leq key[parent(x)]$.
- *Full tree node allocation scheme: nodes of A are allocated in increasing order of level, and left-to-right within the same level.*
- *This allows array implementation, where array indices simulate tree pointers.*



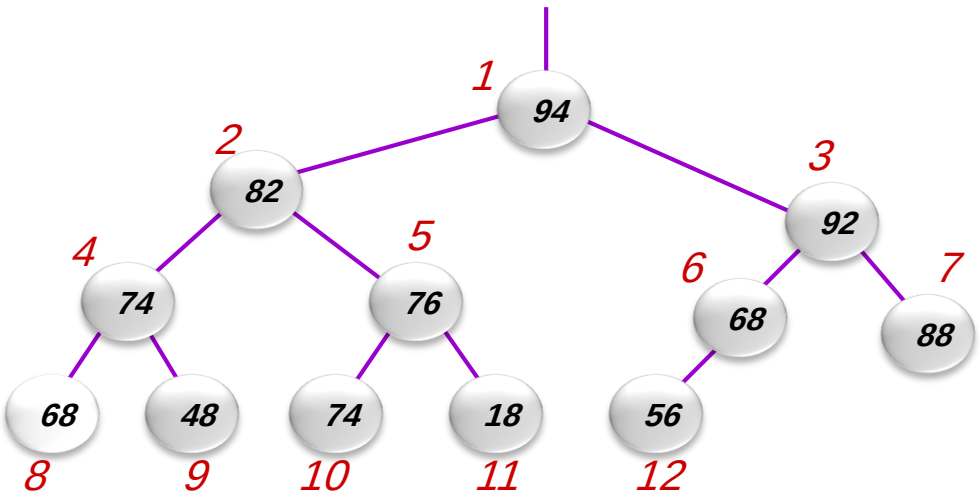
Array as Binary Heap

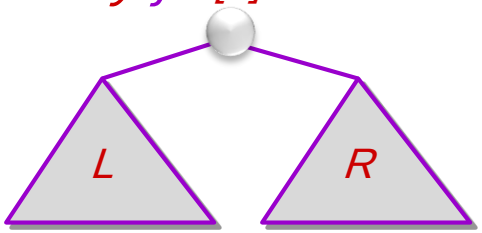


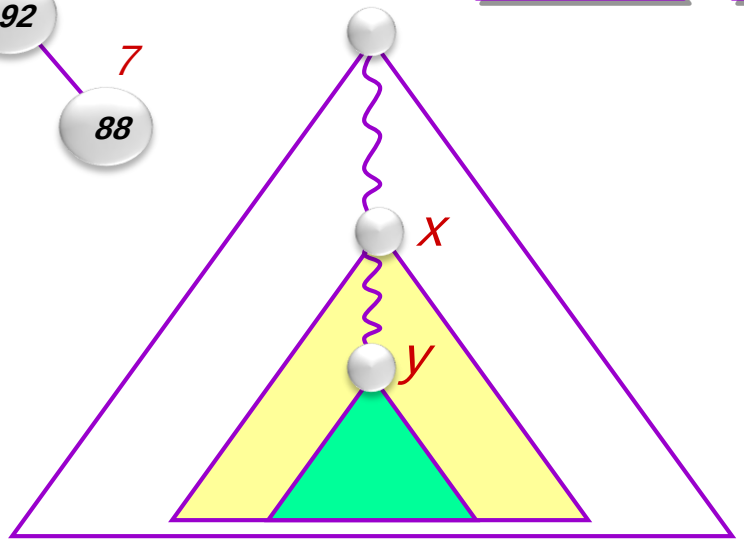


Some MAX Heap Properties

- *Root A[1] contains the maximum item.*
- *Every root-to-leaf path appears in non-increasing order.*
- *Every subtree is also max heap ordered. [Recursive structure]*
- *The key at any node is the largest among all its descendents.*
- *$\forall(x,y) \in AncestorOf : A[x] \geq A[y]$,*
  where AncestorOf = { (x,y) : node x is ancestor of node y }.







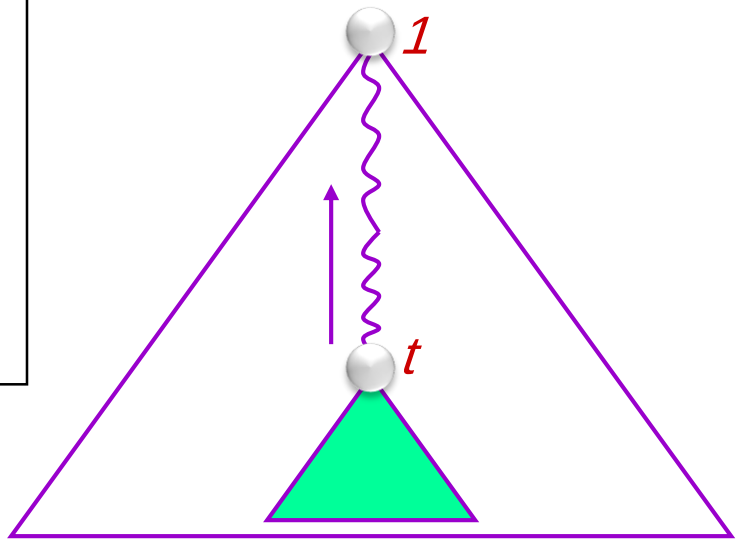
UpHeap

- *A[1..n] = a max-heap.*
- *Suddenly, item A[t] increases in value.*
- *Now A is a "t upward corrupted heap":*

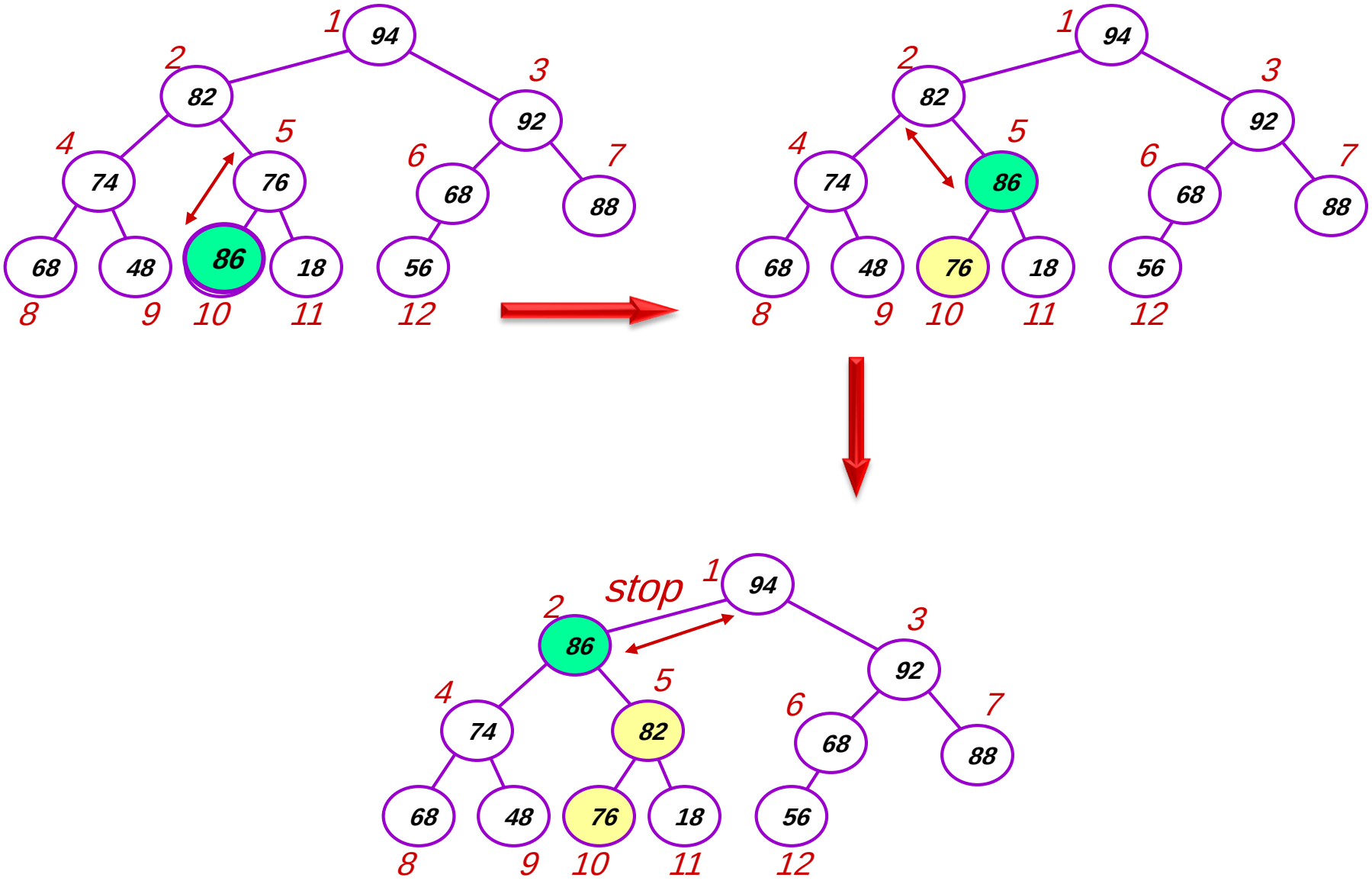
$$\forall (x,y) \in AncestorOf: \quad y \neq t \Rightarrow A[x] \geq A[y].$$

- *Question: how would you rearrange A to make it a max-heap again?*
- *Answer: percolate A[t] up its ancestral path.*

```
procedure UpHeap(A, t)    § O(log n) time
Pre-Cond:  A is a t upward corrupted heap
Post-Cond:  A is rearranged into a max-heap
   p ← ⌊t/2⌋    § parent of t
   if p = 0 or A[p] ≥ A[t] then return
   A[t] ↔ A[p]
   UpHeap(A,p)
end
```



UpHeap Example



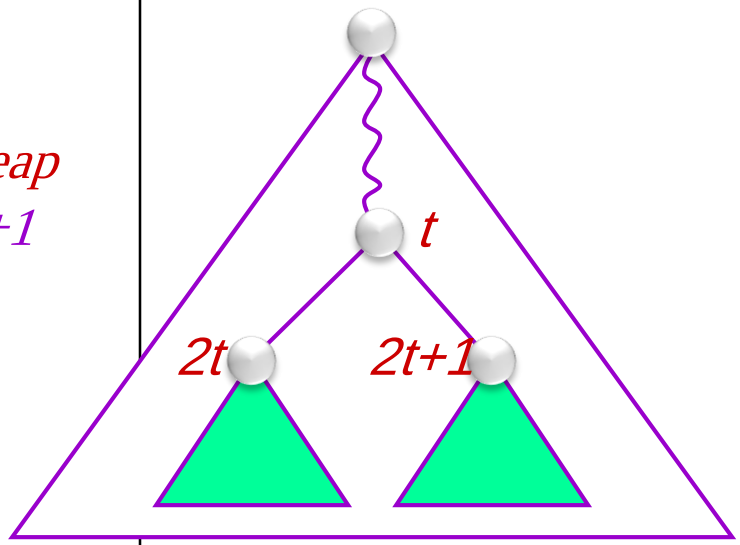
DownHeap (or Heapify)

- *A[1..n] = a max-heap.*
- *Suddenly, item A[t] decreases in value.*
- *Now A is a "t downward corrupted heap":*

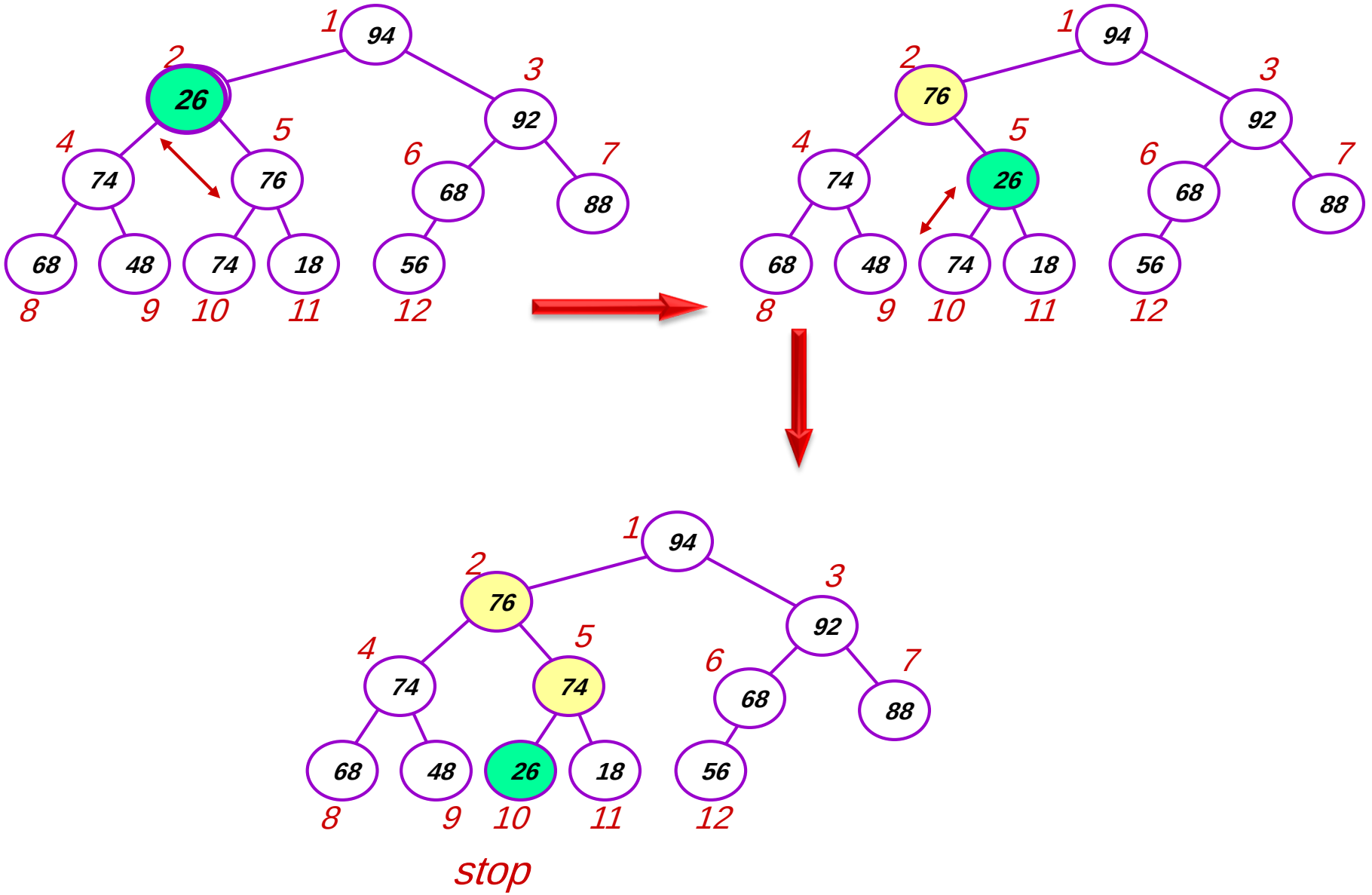
$$\forall (x,y) \in \text{AncestorOf}: \quad x \neq t \Rightarrow A[x] \geq A[y].$$

- *Question: how would you rearrange A to make it a max-heap again?*
- *Answer: demote A[t] down along largest-child path.*

```
procedure DownHeap(A, t)    § O(log n) time
Pre-Cond:  A is a t downward corrupted heap
Post-Cond:  A is rearranged into a max-heap
    c ← 2t          § left child of t
    if  c > size[A] then return      § c not part of heap
    if  c < size[A] and A[c] < A[c+1] then  c ← c+1
    § now c is the largest child of t
     if  A[t] < A[c] then
         A[t] ↔ A[c]
         DownHeap(A, c)
end
```



DownHeap Example



Construct Heap

- *One application of heaps is sorting. But how do we start a heap first?*
- ***Problem:*** *Given array A[1..n], rearrange its items to form a heap.*
- ***Solution: Build Incrementally:***

  That is, make A[1..t] a max heap while incrementing $t \leftarrow 1 .. n$.

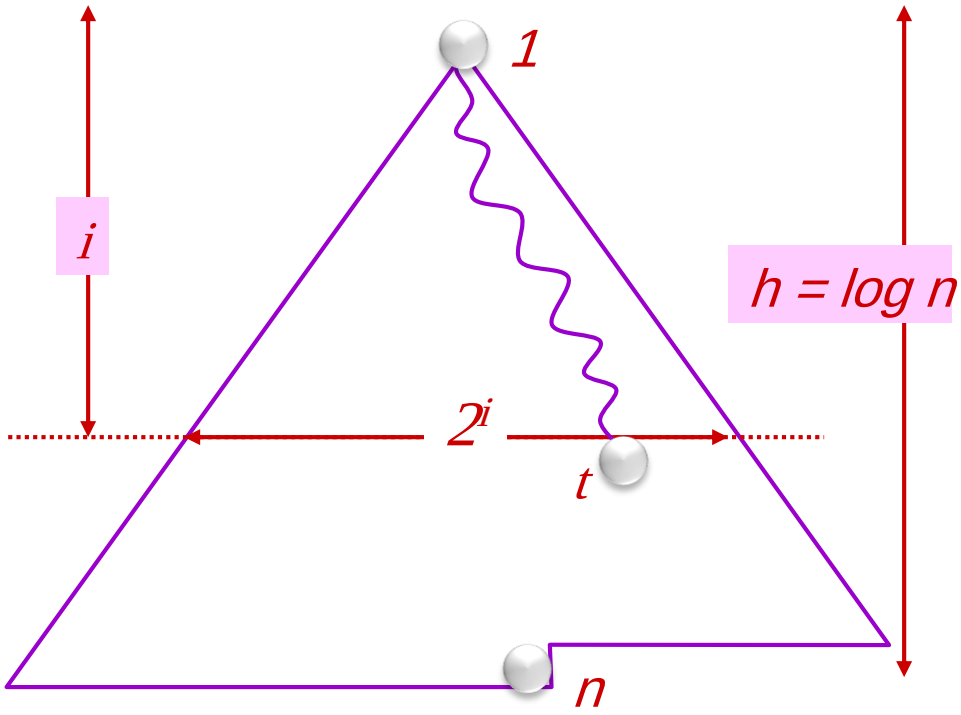
  That is, ***for*** $t \leftarrow 1 .. n$ ***do*** *UpHeap(A, t)* ***end***

$$\text{Time} = \Theta\left(\sum_{i=0}^{h} i 2^i\right)$$

$$= \Theta(h 2^h)$$

$$= \Theta(n \log n)$$

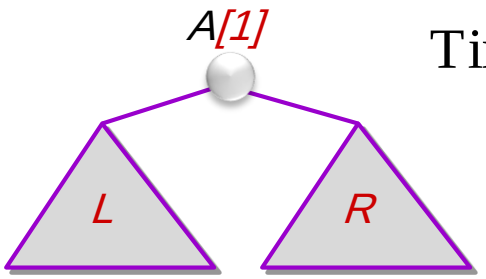
Most nodes are concentrated near the bottom with larger depths but smaller heights. ***Idea: DownHeap is better!***



Heap Construction Algorithm

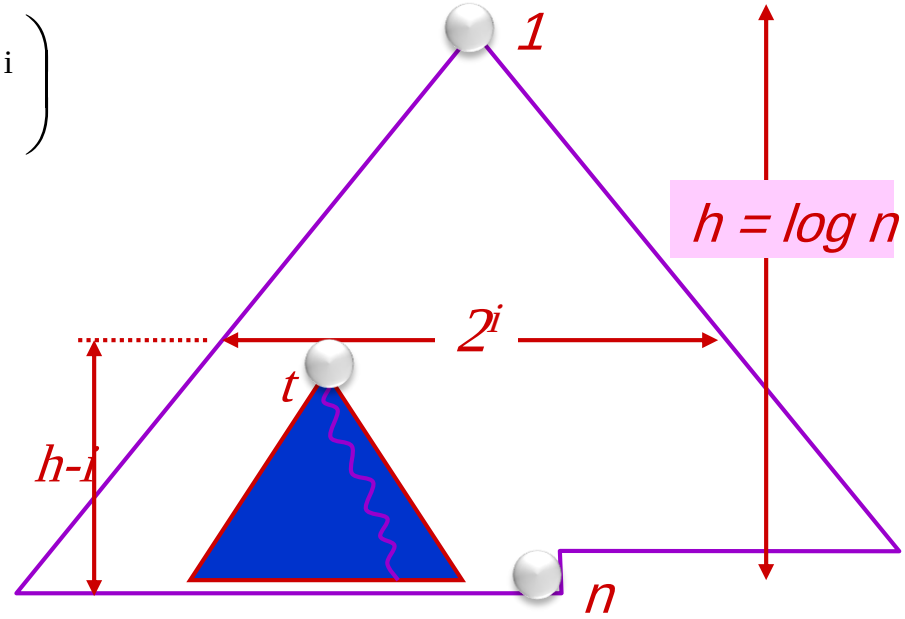
Solution 3: Build Backwards on t *by DownHeap(A,t).*

```
procedure ConstructHeap(A[1..n])   § O(n) time
Pre-Cond:  input is array A[1..n] of arbitrary numbers
Post-Cond:  A is rearranged into a max-heap
    size[A] ← n          § establish last node barrier
    LastNonleafNode ← ⌊n/2⌋
    for t ← LastNonleafNode  downto  1 do  DownHeap(A, t)
end
```

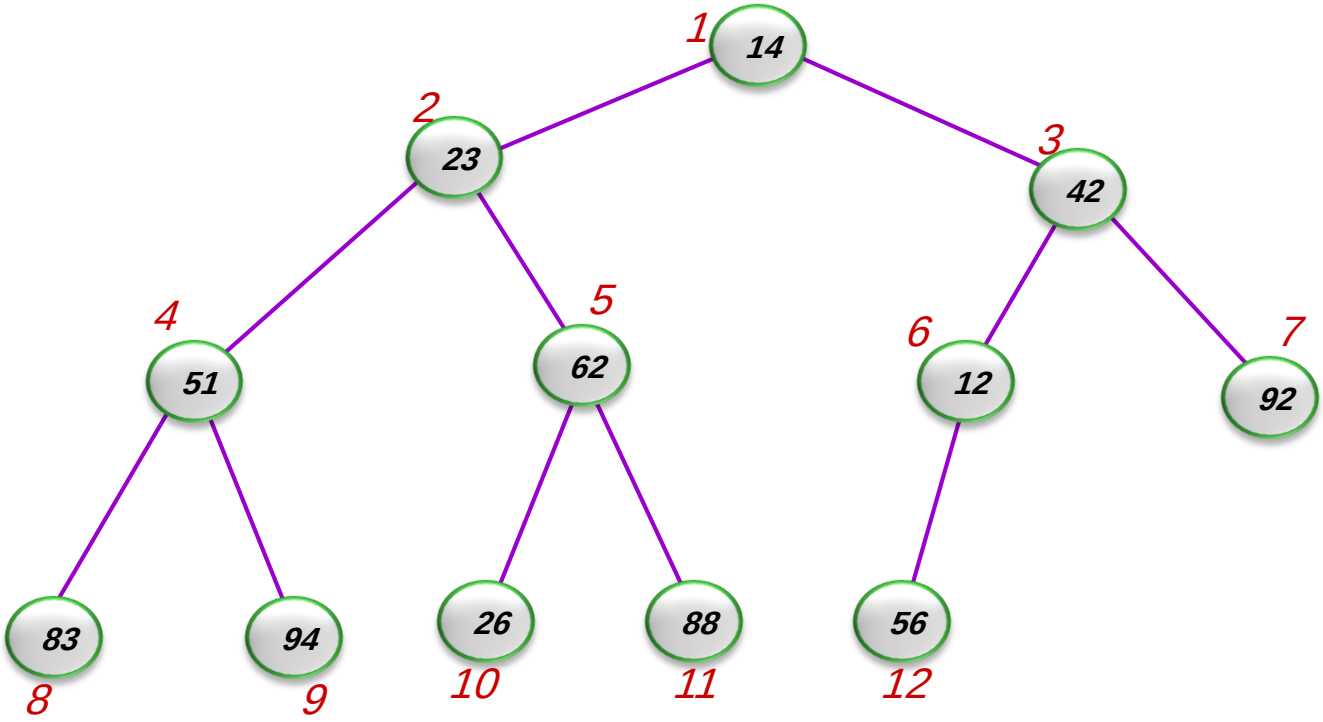


T(n) =
T(|L|) + T(|R|) + O(log n)
⇒ T(n) = O(n)

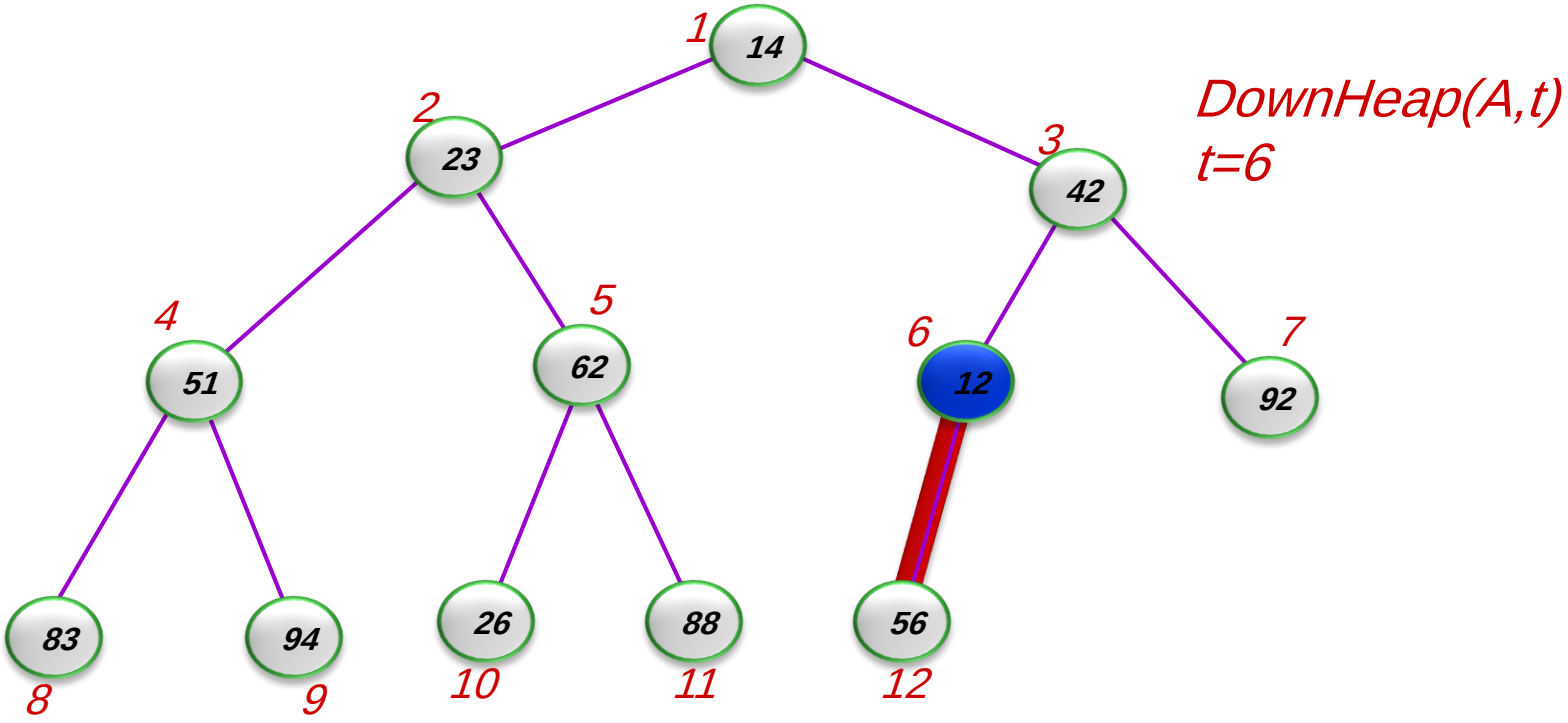
$$\text{Time} = \Theta\left(\sum_{i=0}^{h}(h-i)2^{i}\right)$$
$$= \Theta\left(\sum_{j=0}^{h} j2^{h-j}\right)$$
$$= 2^{h}\,\Theta\left(\sum_{j=0}^{h} j2^{-j}\right)$$
$$= 2^{h}\,\Theta(1)$$
$$= \Theta(n)$$



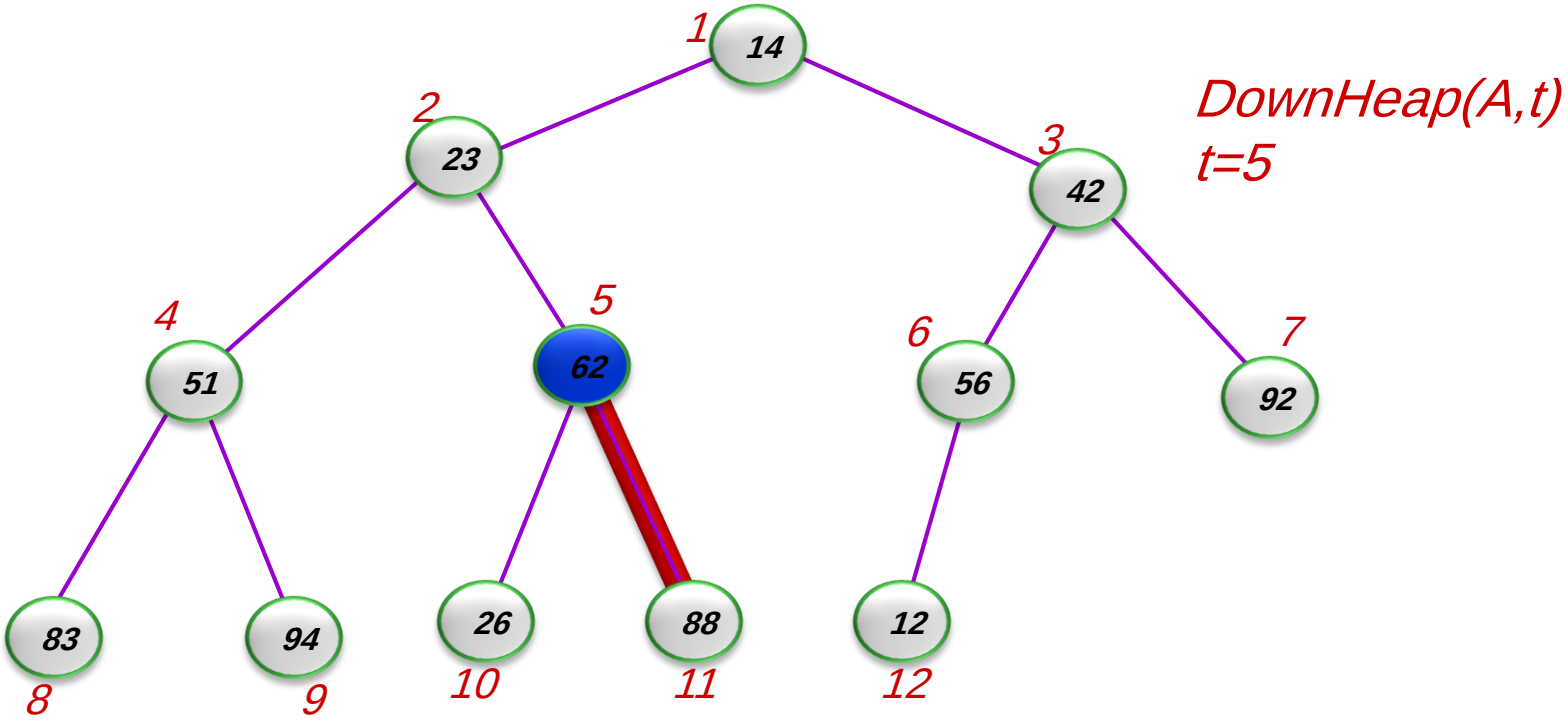
Construct Heap Example



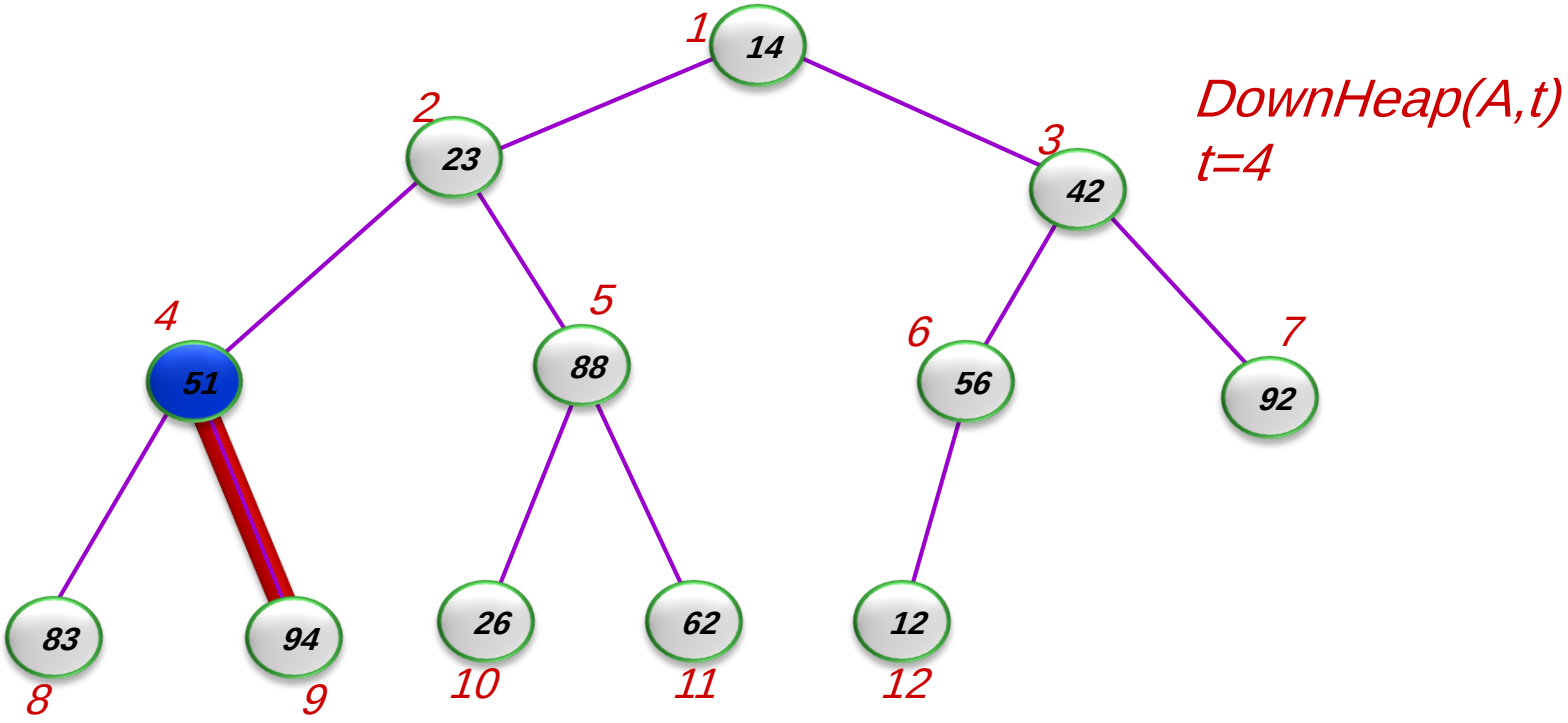
Construct Heap Example



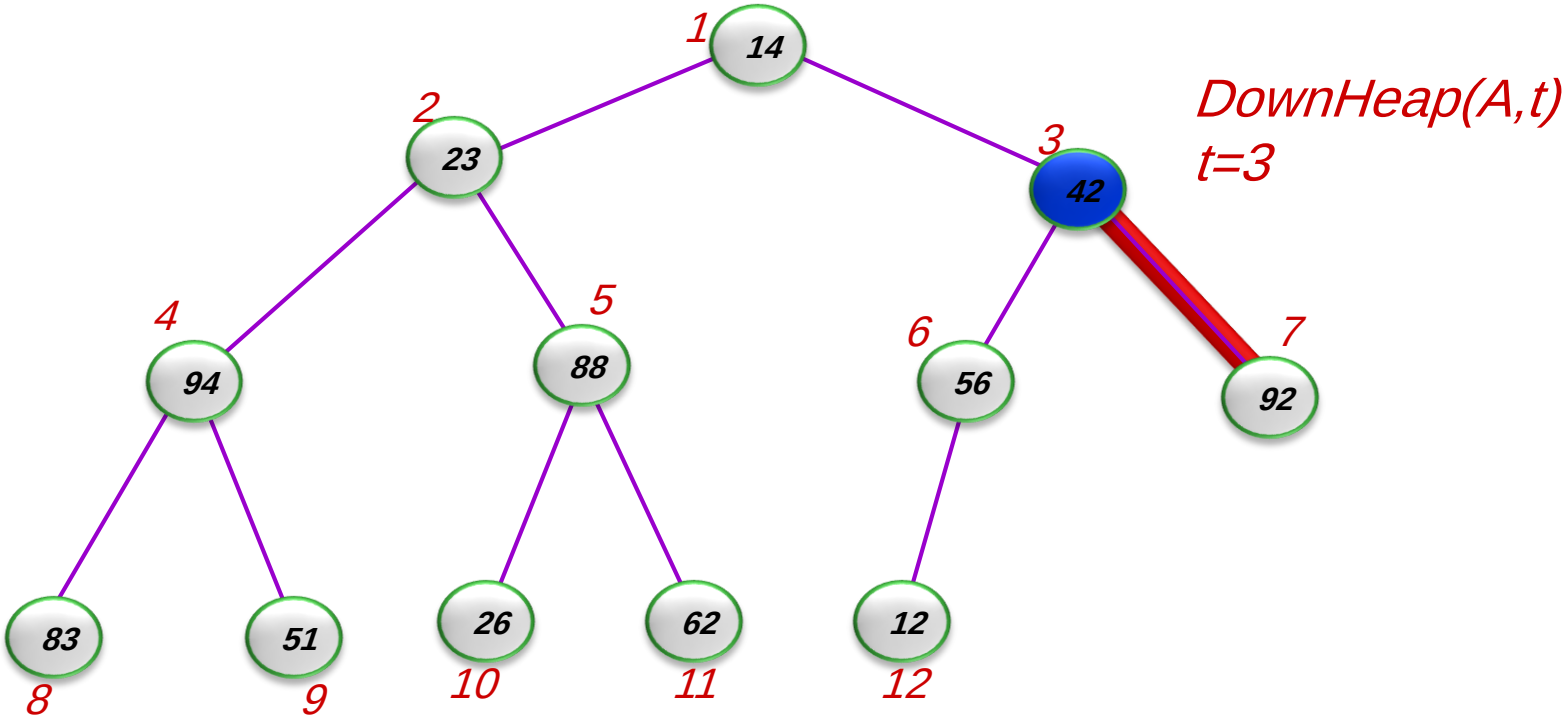
Construct Heap Example



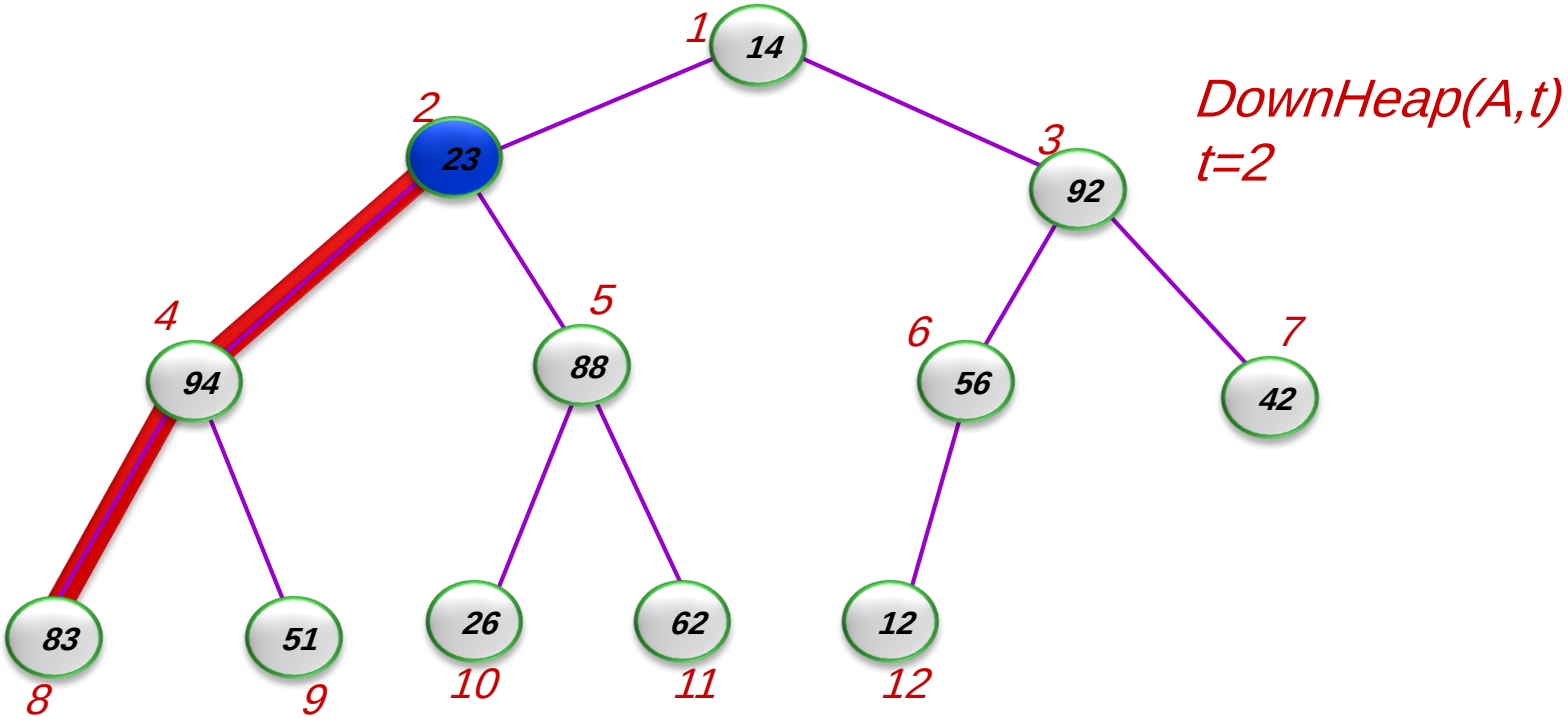
Construct Heap Example



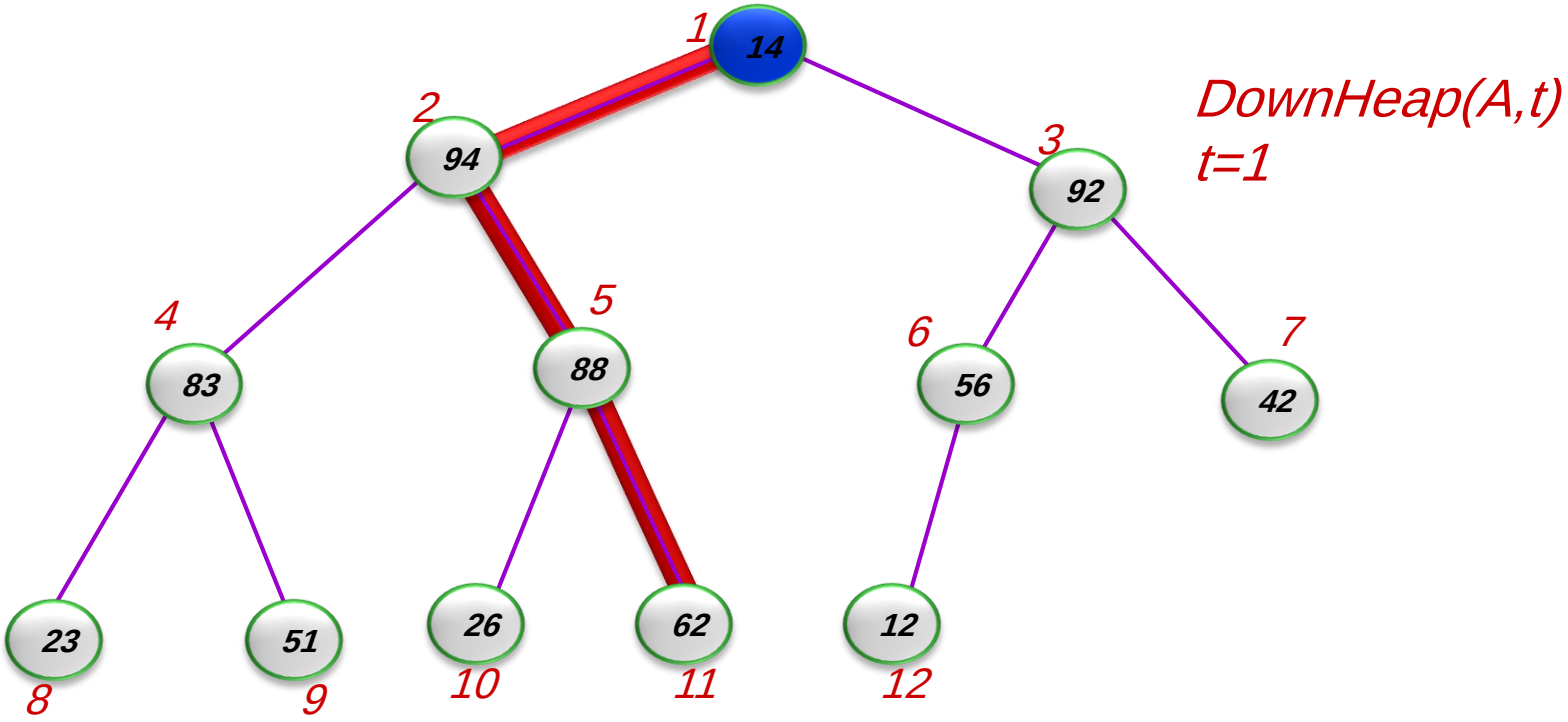
Construct Heap Example



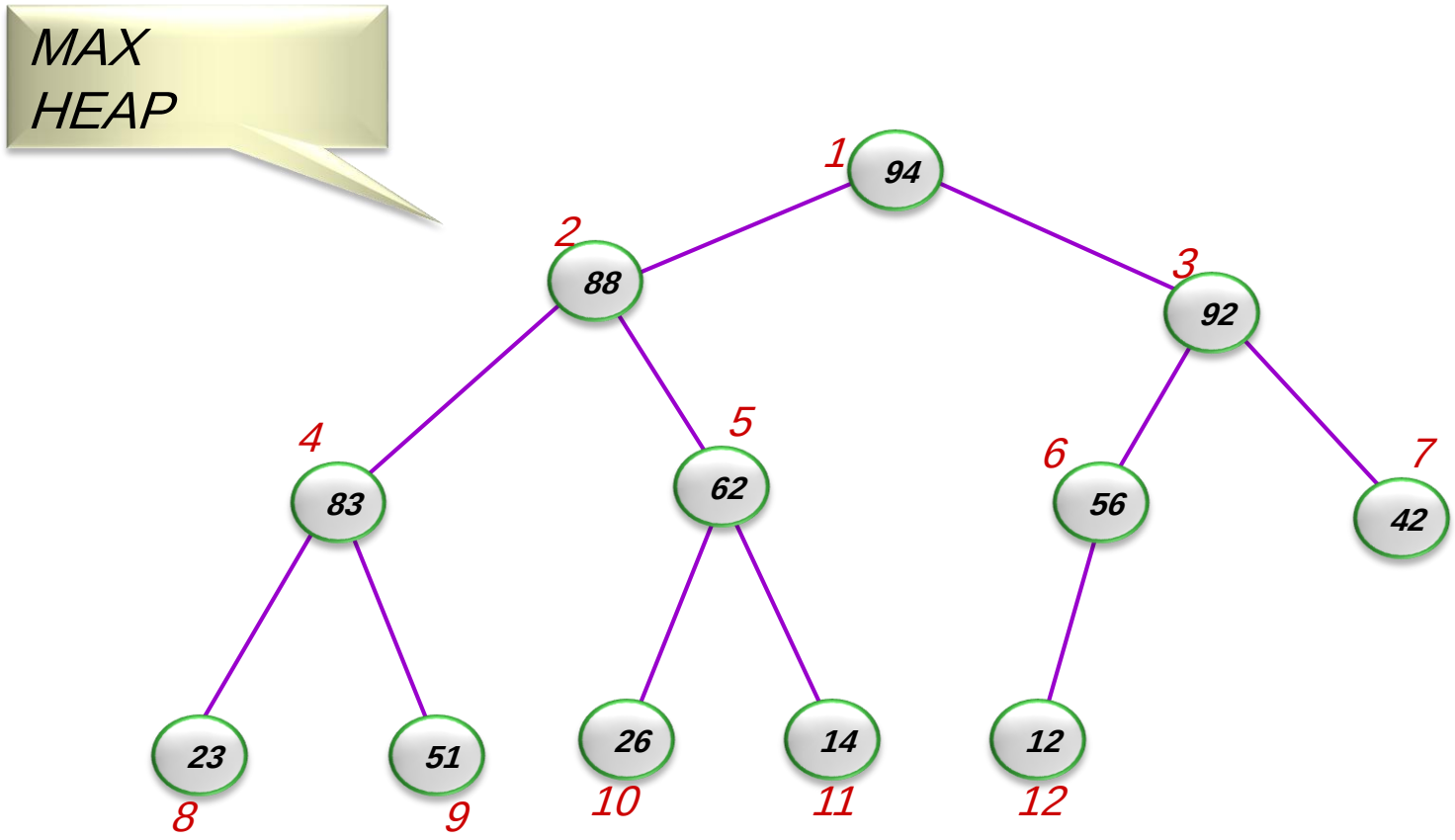
Construct Heap Example



Construct Heap Example



Construct Heap Example



The End