Module 5
Quick Sort, Heap Sort, and Lower Bound on Sorting

S. Lakshmivarahan
School of Computer Science
University of Oklahoma
USA-73019
Varahan@ou.edu

Quick Sort

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
X	<u>26</u>	5	37	1	61	11	59	15	48	19
		↑L								↑R

- This is the array to be sorted
- Pick one of the elements of the array X as the <u>**pivot**</u>
- Let X(1)=26 be the pivot

Quick Sort

- The method consists of two searches- **left search** denoted by ↑L, starting at location 2, and **right search** ↑R, starting at location n

Quick Sort

- ↑L starts from location 2. While at each location $i \geq 2$, it compares the content of that location $X(i)$ with the pivot, $X(1)$ at location 1
- If $X(i) > X(1)$, ↑L stops at the location i
- If $X(i) \leq X(1)$, ↑L then moves to the right, until it finds a location j such that $X(j) > X(1)$ where it stops

Quick Sort

- Likewise, $\uparrow R$ first compares X(n) with the pivot, X(1)
- If X(n) < X(1), it stops at that location
- If X(n) ≥ X(1), it then moves left until it finds a location k≤n such that X(k) < X(1), where it stops

Quick Sort

- Thus, ↑L has found the location j, and ↑R has found the location k, such that

  $$X(j) > X(1) \text{ and } X(k) < X(1)$$

- At this time, we swap the contents of X(j) and X(k), ↑L moves to (j+1) and ↑R to (k-1) and the process continues until they cross

Quick Sort

- When they cross, swap the contents X(1) of location 1 (which is the pivot) with that of the location k pointed by the ↑R
- This splits the file into two disjoint subfiles

Quick Sort

- Let us illustrate this process

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
X	<u>26</u>	5	37	1	61	11	59	15	48	19
			↑L							↑R

- Here, j=3 and k=n=10: $\left.\begin{array}{l} x(j) > x(1) \\ x(k) < x(1) \end{array}\right\} \rightarrow$ SWAP

Quick Sort

- We then get

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
X	<u>26</u>	5	19	1	61	11	59	15	48	37
					↑L			↑R		

- Here, j=5, k=8: $\left.\begin{array}{l} x(j) > x(1) \\ x(k) < x(1) \end{array}\right\}$ → SWAP

Quick Sort

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
X	<u>26</u>	5	19	1	15	11	59	61	48	37
						↑R	↑L			

- When they cross, the search stops and swaps the pivot with the contents of the location pointed by ↑R

Quick Sort

- This gives

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
X	11	5	19	1	15	<u>26</u>	59	61	48	37

- Notice that the pivot is now at its correct place in the sorted file
- The elements to the left of the pivot are smaller than it, and those to the right are larger

Quick Sort

- We have effectively split the file into two disjoint parts – left and right subfiles- each of which is sorted by the same method
- Thus, you can see DC at play here

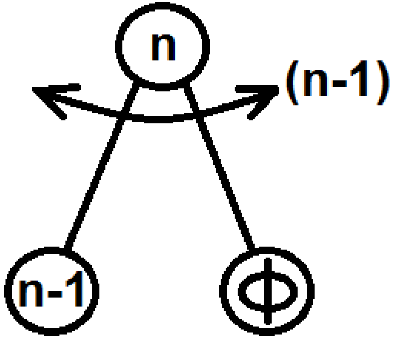
Quick Sort- Observations

- A number of observations are in order

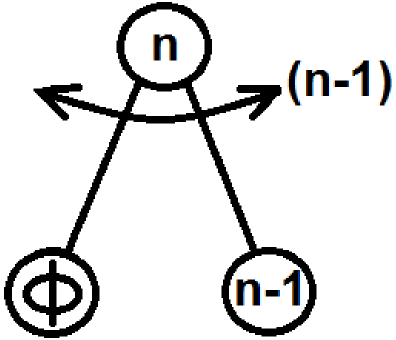
1. Notice that the above procedure would work with any element chosen as the pivot in place of X(1)
2. The question is: Is there a good choice for the pivot?

Quick Sort- Observations

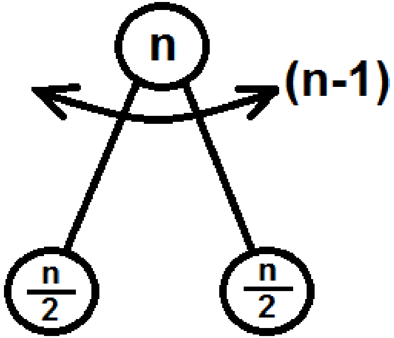
3. To answer this, let us examine the different possibilities that can happen at the end of the first phase



(3a)
Worst Case



(3b)
Worst Case



(3c)
Best Case

First Phase- Possibilities

A. If the pivot happens to the maximum, then ↑L cannot find any item greater than the pivot. In this case, the resulting right subfile would be empty

B. Likewise, if the pivot happens to be the minimum, then ↑R cannot find an item less than the pivot, and consequently, the left subfile would be empty.

First Phase- Possibilities

C. If the pivot happens to be the “median” of the file, then the left and right subfiles would be nearly the same size.
   - Recall, the “median” of a set of numbers is that number which has equal number of numbers less and more
   - **<u>Interpret</u>:**
     - Median Income in a state
     - Median grade in a class

Worst Case Analysis

- Recall at the end of the first phase, ↑L and ↑R together compare every item other than the pivot with the pivot, taking (n-1) comparisons in all
- This is the cost of splitting the file into two parts
- In the worst case, since one of the two subfiles is empty, we get

$$T(n) = T(n-1) + n - 1$$

With T(2) = 1

Worst Case Analysis

- Then, as we have seen earlier

$$T(n) = O(n^2)$$

- **Homework**: Solve the previous recurrence explicitly and find the exact solution

Best Case Analysis

- In this case,

$$T(n) \approx 2T\left(\frac{n}{2}\right) + (n-1)$$

- with T(2) = 1
- Again, as before,

$$T(n) = O(n \log n)$$

- **Homework**: Solve the previous recurrence and find the solution

Average Case Analysis

- Since

$$O(n \log n) < O(n^2)$$

for large n, it would be interesting to compute the average case

- To capture the inherent randomness in this algorithm, let us consider a model that captures the position that the pivot will occupy at the end of the first phase

Average Case Analysis

- Let us postulate that the pivot occupies any one of the n locations with the same probability, p=$^1/_n$
- Given this, we can easily see that

$$T_a(n) = (n-1) + \frac{1}{n}\sum_{j=1}^{n}[T_a(j-1) + T_a(n-j)]$$

where

(n-1) = Initial cost of splitting

$\frac{1}{n}$ = Prob. That pivot is in position j

$T_a(j-1)$ = Average cost of sorting left side

$T_a(n-j)$ = Average cost of sorting right side

Average Case Analysis

- Opening the sum and collecting the terms:

$j = 1: \; T_a(0) + \; T_a(n-1)$

$j = 2: \; T_a(1) + \; T_a(n-2)$

$\dots$

$j = n-1: \; T_a(n-2) + \; T_a(1)$

$j = n \qquad : \; T_a(n-1) + \; T_a(0)$

Recall:
- Ta(0) = 0
- Ta(1) = 0

$$\therefore T_a(n) = (n-1) + \frac{2}{n}\sum_{j=2}^{n-1} T_a(j)$$

Average Case Analysis

- This is called a complete history equation
- Recall, this solution is $T_a(n) = O(n \log n)$ - Refer to the Appendix
- Thus, the average case is closer to the best case, and hence the name Quick-sort
- This implies that the worst case occurs with a very small probability

Pivot Selection

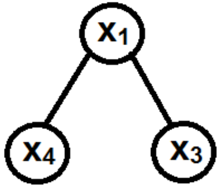
- A useful suggestion to pick the pivot is to divide the given file into three equal parts and pick one location from each of the three parts, say *i*, *j*, and *k*, such that
- $1 \leq i \leq \frac{n}{3} \leq j \leq \frac{2n}{3} \leq k \leq n$
- Then find the median of X(i), X(j), and X(k) and use this median as the pivot

Pivot Selection

- Another useful suggestion is to implement this recursive procedure iteratively, which could save some overhead

- **Homework**: Compare Quick-sort with Selection sort, Bubble sort, and 2-way merge sort

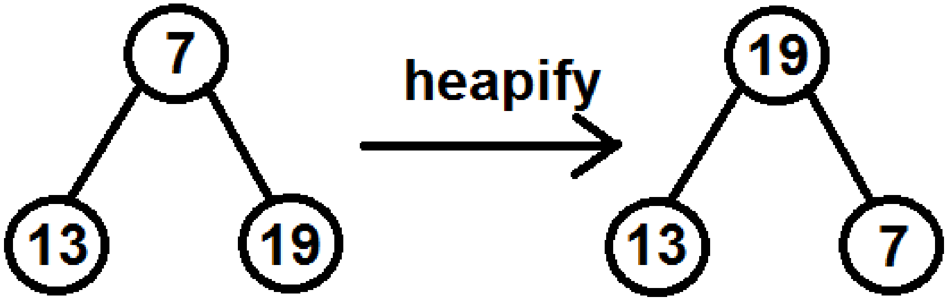
Heap Sort

- Consider 

 where x_i are three distinct numbers
- If $x_1 > \max\{x_2, x_3\}$, it is called a max heap
- If $x_1 > \min\{x_2, x_3\}$, it is called a min heap

Heap Sort

- We will consider max heap in the following
- If it is not already a heap, we need to heapify:
  - Find max$\{x_2, x_3\}$
  - Compare this max with x_1 and swap it with x_1 if $x_1 < \max\{x_2, x_3\}$



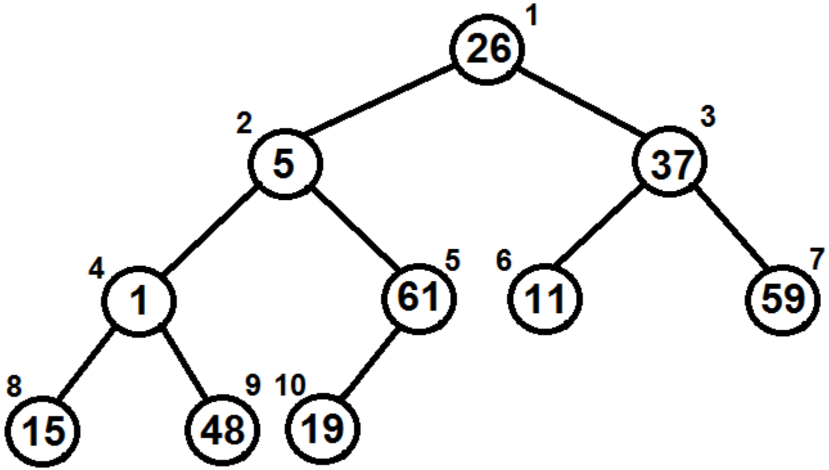
Heap Sort

- Consider an input file

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
X	26	5	37	1	61	11	59	15	48	19

Heap Sort

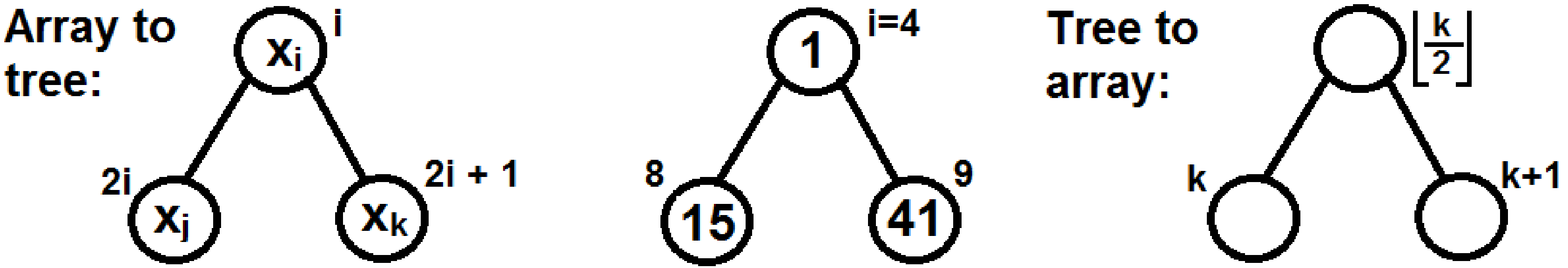
- First, we build a full-binary tree as follows



1. Start with the root
2. Fill each level from left to right
3. Open a new level when a given level is filled

Heap Sort

- There is a close connection between the 1-D array and the 2-D full binary:

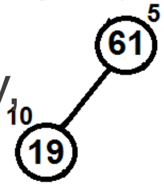


Heap Sort

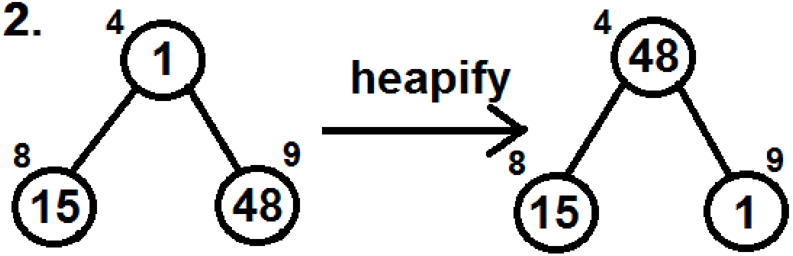
- Consider the element x_i that is at location i in the array and that occupies the root
- Then, its left son x_j in the tree is at location 2i in the array, and right son x_k in the tree is at location (2i+1)
- Thus, one can easily simulate the full binary tree without having to actually build it using a linked list

Initial Heap Formation

- Start heaping from the right end of the bottommost level

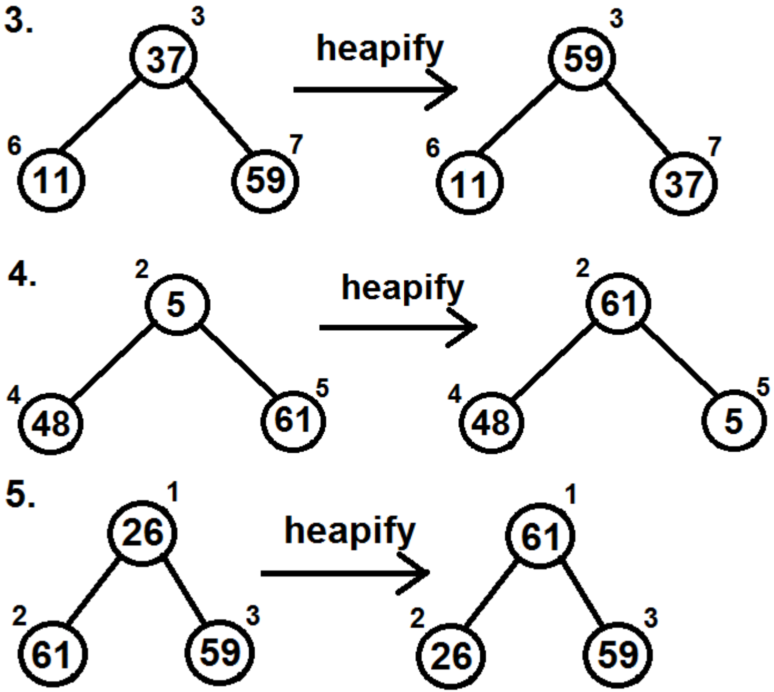
1. Clearly, 

is a local heap

2. 

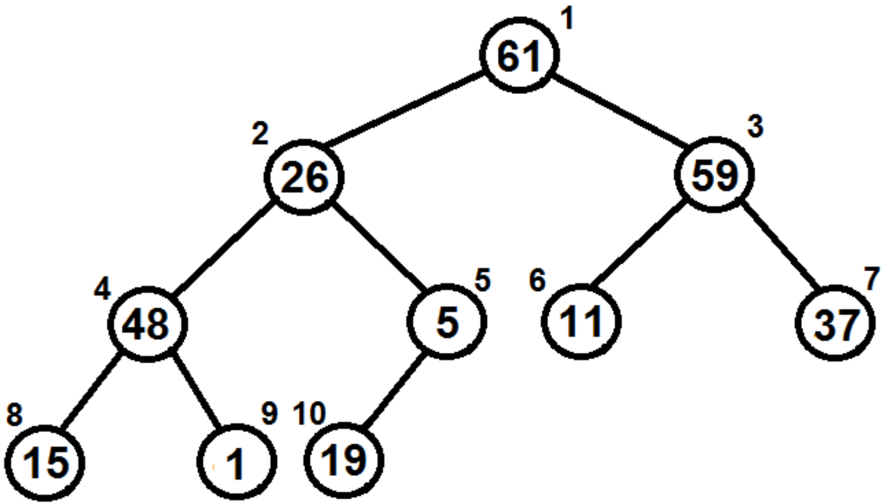
is a local heap

Initial Heap Formation



Initial Heap Formation

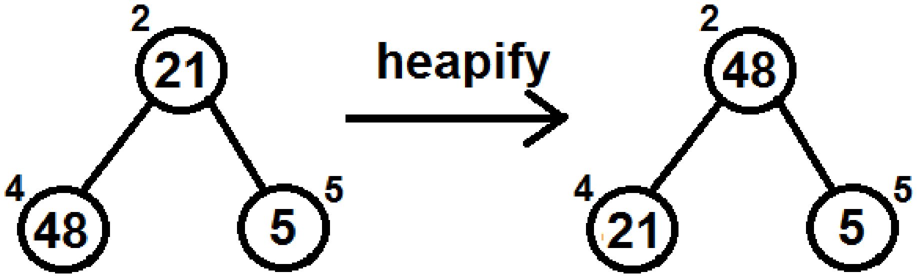
- This gives us the following tree at the end of the first pass

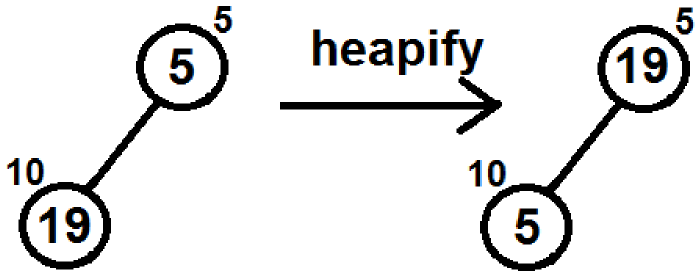


- The right sub tree is a heap, but the left sub tree is **not**

Initial Heap Formation

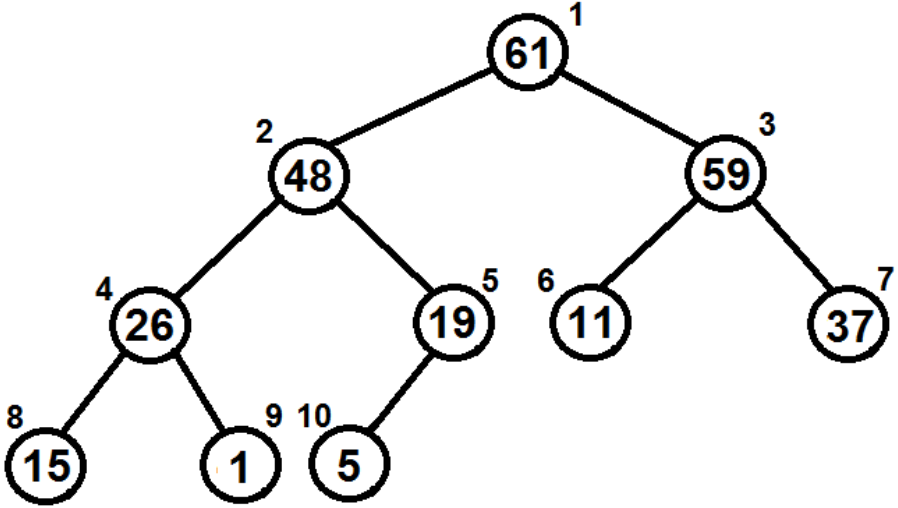
- Adjust:





Initial Heap Formation

- Thus, we get the initial heap as



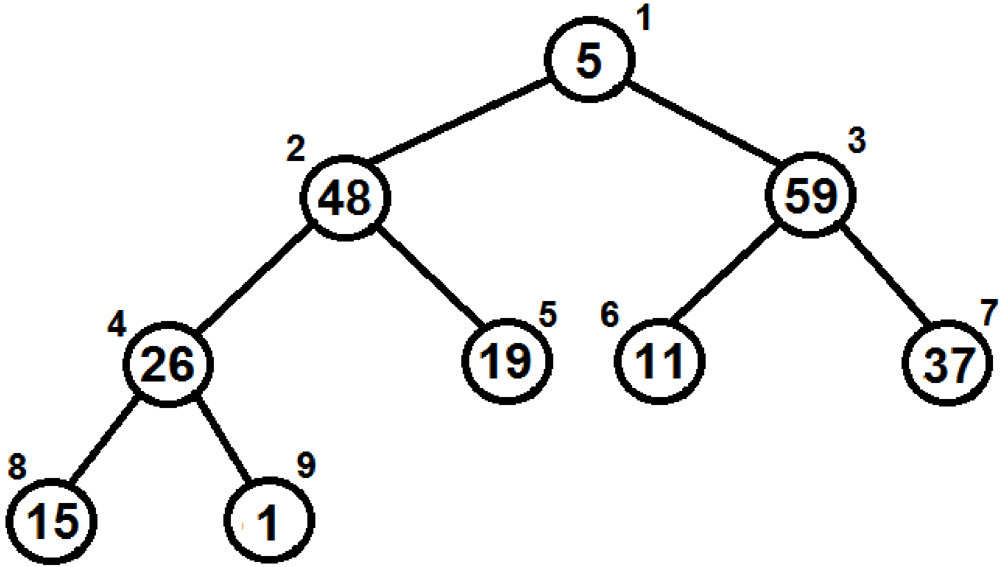
- Thus, the max is at the root

Heap Sort

- Copy the max 61 from the root to the first element of the output file
- This creates a hole there
- To fill this hole, copy the value from the rightmost lead node at the bottommost level into the root

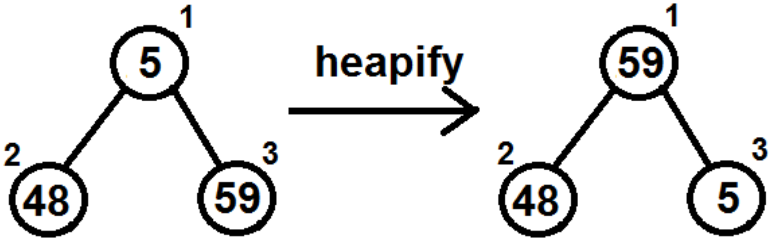
Heap Sort

- This shrinks the size of the tree by 1 and also disturbs the heap

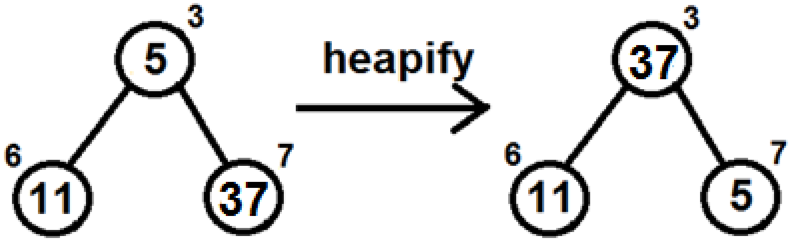


Heap Sort

- Re-heapify:

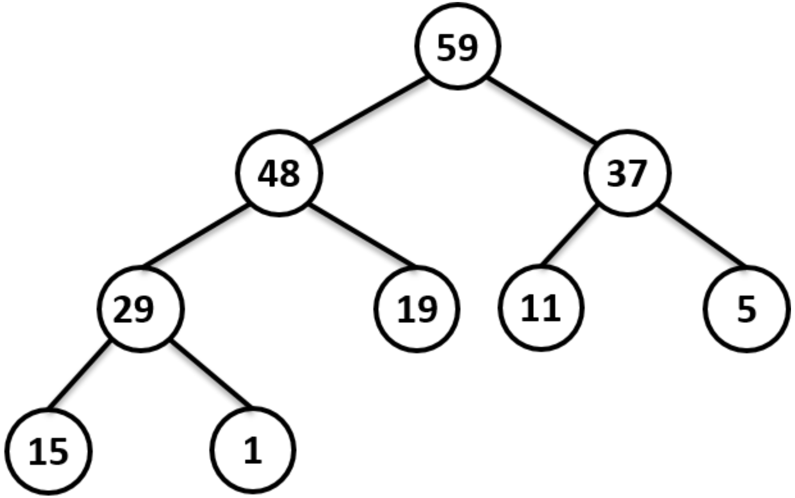


- Then



Heap Sort

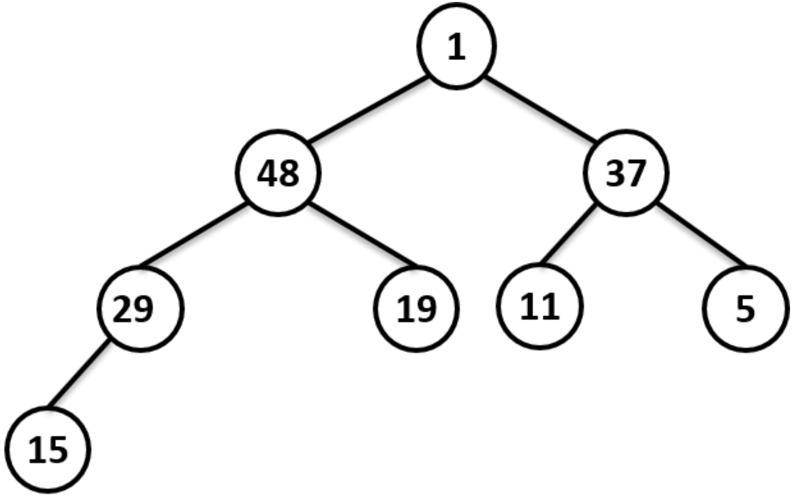
- The new heap is



- Notice the item 5 which was at the root descended to its right place along a path whose length is ≤ [log (n-1)]

Heap Sort

- Now 59 from the root is copied with the sorted list giving 61, 59
- Now copy 1 from the right most end of bottom level giving



Heap Sort

- By repeating this process we get the sorted list
- The cost of sorting, once the initial heap is formed is

$$T(n) = [\log n] + [\log(n-1)] + \cdots + [\log 2]$$

$$=\sum_{i=2}^{n}[\log i]$$

$$=\sum_{i=2}^{n}(1 + \log i) \qquad \text{since } [x] \le 1 + x$$

$$=(n-1) + \sum \log(i) = (n-1) + \log n!$$

$$= \mathrm{O}(n\ logn)$$

Heap Sort

- The cost of initial heapfying is O(n)
- The total cost $= O(n) + O(n \log n) = O(n \log n)$

- **Homework** : Prove that the initial heapfying cost is $O(n)$

Lower Bound on Sorting

- Start with a classification of the sorting algorithms

<u>Elementary</u>

- Bubble sort
- Selection Sort
- Insertion-sort with sequential search

$T(n) = O(n^2)$

<u>Advanced</u>

- 2-way merge sort
- Avg. case quick-sort
- Insertion-sort with binary search
- Heap sort

$T(n) = O(n \log n)$

Lower Bound on Sorting

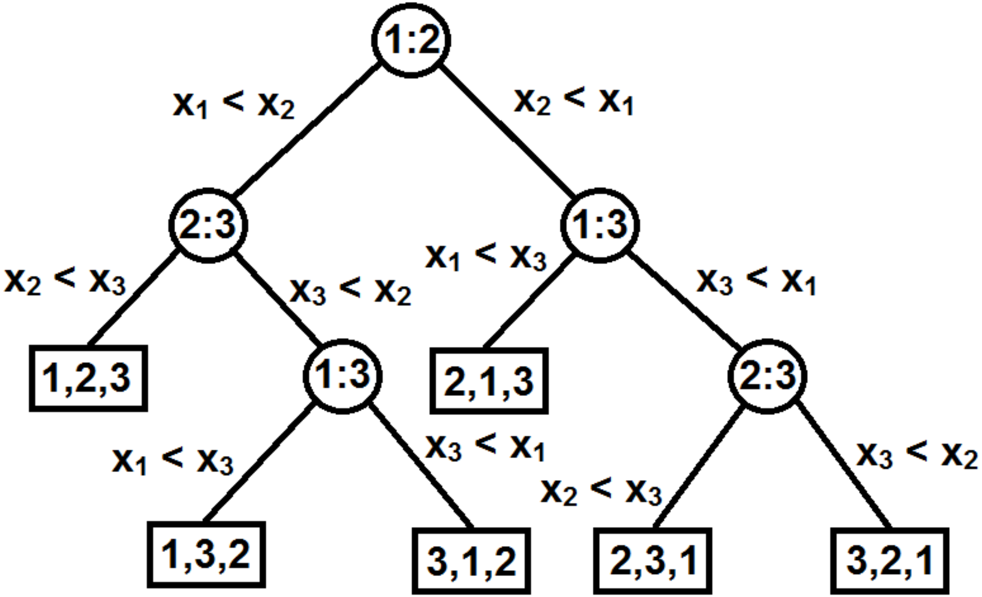
- The question now is whether we can do better than these so called advanced algorithms
- Recall that we are given n items to be sorted with no other information about the range of values they can take
- In the absence of any other information, it turns out that we cannot do better than the advanced algorithms

Lower Bound on Sorting

- To prove this, consider the process of sorting three distinct numbers: x_1, x_2, and x_3
- Since comparison is a binary operation, we can only compare a pair at a time
- So, consider a comparison tree that underlies the sorting process

Lower Bound on Sorting

- In the following, i:j would imply we compare x_i and x_j

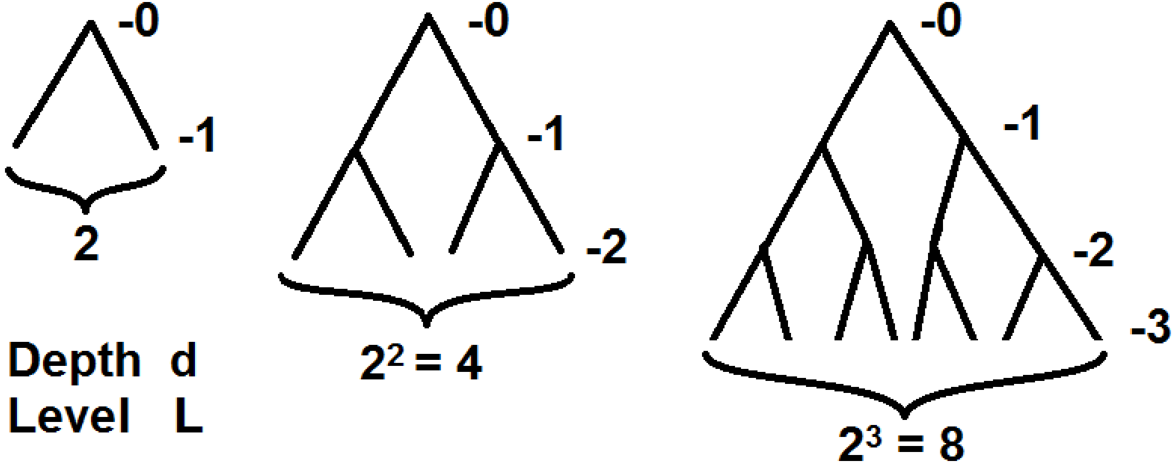


Lower Bound on Sorting

- In this tree, the leafs correspond to the sorted list
- There are 3! = 6 ways to sort the given input of three numbers, x_1, x_2, x_3 depending on their values
- This tree is called an **adaptive tree** since later comparisons depends on the results of the previous ones
- This tree is a **2-tree** in the sense it is a **binary tree** where each internal node has exactly two branches

Lower Bound on Sorting

- Here are some examples of 2-trees of depths 1, 2, 3, …



- Thus, if the root is level L = 0, then there are 2^L nodes at level L

Lower Bound on Sorting

- Now, to be able to sort n distinct items we need a 2-tree whose depth d is such that

$$2^d \geq n!$$

- Taking the logarithm:

$$d \geq \log n! \ \rightarrow ①$$

- That is, the minimum depth d of a 2-tree needed to sort n items is given by ①

Lower Bound on Sorting

- Notice that the leafs do not occur at the same level- refer to the figure for n=3
- Since $\log n! = \Omega\,(n \log n)$, it follows

$$d = \Omega(n \log n) \ \rightarrow 2$$

- Recall that in a tree based algorithm, the depth (length of the longest path) is the time

Lower Bound on Sorting

- That is, we cannot sort n items with a 2-tree of depth less than d in ②
- When n=3, n! = 6. Hence, a 2-tree of depth d=2 is **not** enough, but one with depth d=3 is sufficient
- Since the algorithms in the advanced category are such that

$$T(n) = O(n \log n)$$

- Since we cannot do with less than $n \log n$, these advanced algorithms are called **optimal algorithms**