Module 2: Classical Algorithm Design Techniques

Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

Module Topics

- 2.1 Brute Force
- 2.2 Decrease and Conquer
- 2.3 Divide and Conquer
- 2.4 Transform and Conquer
- 2.5 Space-Time Tradeoff: Sorting and Hashing

2.1 Brute Force

Brute Force String Matching

- *pattern*: a string of *m* characters to search for
- *text*: a (longer) string of *n* characters to search in
- problem: find a substring in the text that matches the pattern

Brute-force algorithm

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until

- all characters are found to match (successful search); or
- a mismatch is detected

Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

```
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 N  O  B  O  D  Y  _  N  O  T  I  C  E  D  _  H  I  M
 N  O  T
    N  O  T
       N  O  T
          N  O  T
             N  O  T
                N  O  T
                   N  O  T
                      N  O  T
```

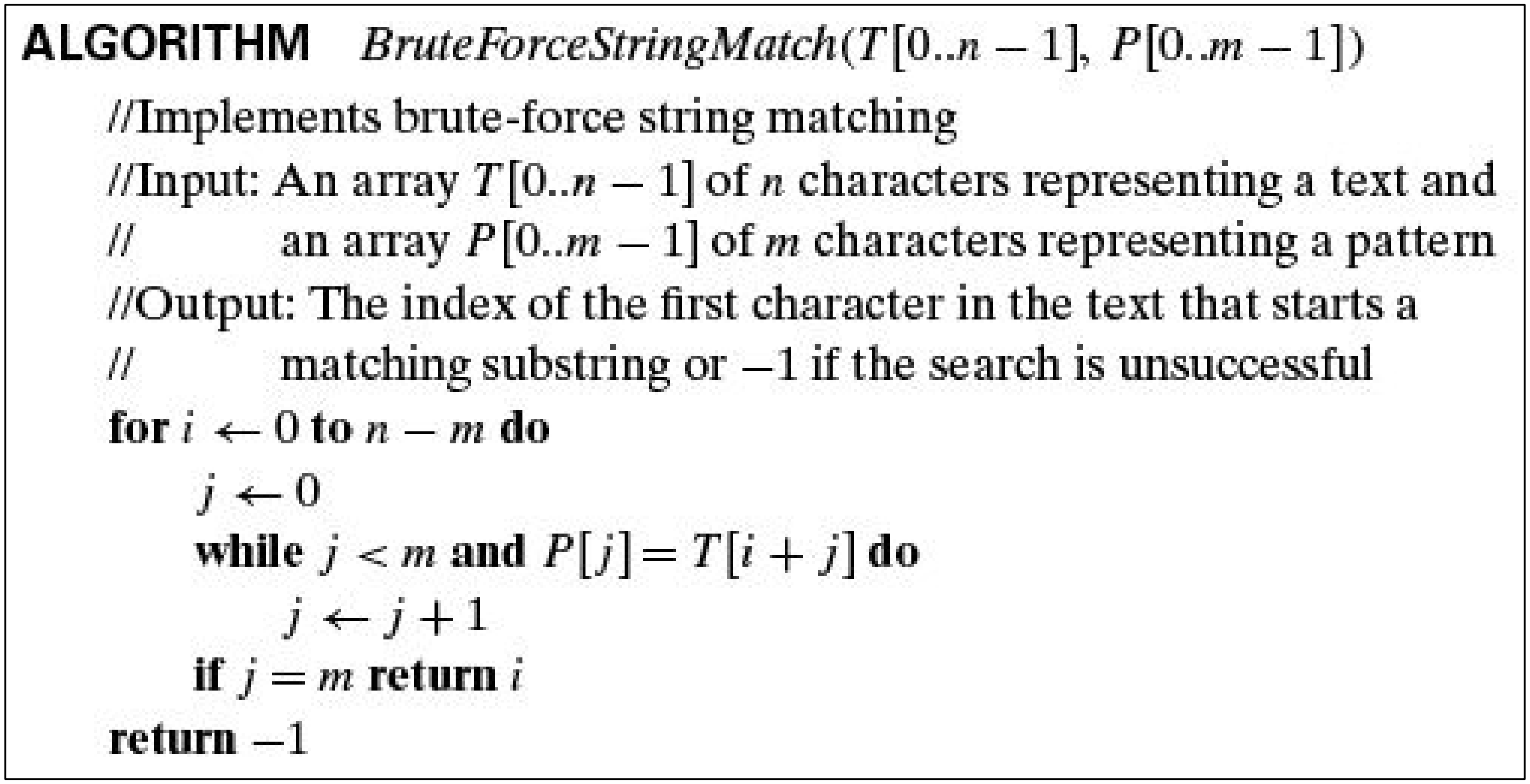
n = 18

```
0 1 2
N O T
```

m = 3

(at the worst-case if 'NOT' did not appear before this)

Brute Force String Matching



```
ALGORITHM  BruteForceStringMatch($T[0..n-1]$, $P[0..m-1]$)
    //Implements brute-force string matching
    //Input: An array $T[0..n-1]$ of $n$ characters representing a text and
    //         an array $P[0..m-1]$ of $m$ characters representing a pattern
    //Output: The index of the first character in the text that starts a
    //         matching substring or $-1$ if the search is unsuccessful
    for $i \leftarrow 0$ to $n - m$ do
        $j \leftarrow 0$
        while $j < m$ and $P[j] = T[i + j]$ do
            $j \leftarrow j + 1$
        if $j = m$ return $i$
    return $-1$
```

Best-case = m comparisons
At the worst case, the algorithm may have to make all *m* comparisons before shifting the pattern, and this can happen for each of the *n* – *m* + 1 tries. Thus, in the worst case, the algorithm makes *m* (*n* – *m* + 1) character comparisons. m << m(n-m+1). The overall time complexity is O(*nm*)

Brute Force String Matching
Examples

How many comparisons are made by the brute-force string matching algorithm in searching for the following patterns in a binary text of 1000 zeros?

00000000000000000000000000000000 … 000 (1000 zeros)

a) 00001

There will be a total of (1000 – 5 + 1) iterations. In each iteration, we will have to do 5 comparisons, as the first 4 bits will match and only the last bit will not match. Hence, the total number of comparisons = 996 * 5 = 4,980

b) 10000

There will be a total of (1000 – 5 + 1) iterations. In each iteration, the first comparison itself will be a failure. Hence, there will be a total of 996 * 1 = 996 comparisons.

c) 01010

There will be a total of (1000 – 5 + 1) iterations. In each iteration, we will do 2 comparisons (the first comparison will be successful and the second one is not). Hence, there will be a total of 996*2 = 1,992 comparisons.

Brute Force String Matching
Examples

Consider the problem of counting the number of sub strings that start with an A and end with a B in a given string of alphabets: DAAXBABAGBD.

Scan the given string from left to right. Initialize the number of sub strings to zero. Keep track of the number of As encountered. Each time a B is encountered, set the number of sub strings to be number of sub strings + the number of As encountered until then. Since we do a linear pass on the given string and do one comparison per character, the time complexity is Θ(n), where n is the length of the string.

		D	A	A	X	B	A	B	A	G	B	D
# A's	0	0	1	2	2	2	3	3	4	4	4	4
# desired substrings	0	0	0	0	0	2	2	5	5	5	9	**9**

2.2 Decrease and Conquer

Decrease by One: Insertion Sort

- Given an array A[0…n-1], at any time, we have the array divided into two parts: A[0,…,i-1] and A[i…n-1].
  - The A[0…i-1] is the sorted part and A[i…n-1] is the unsorted part.
  - In any iteration, we pick an element *v* = A[i] and scan through the sorted sequence A[0…i-1] to insert *v* at the appropriate position.
    - The scanning is proceeded from right to left (i.e., for index j running from i-1 to 0) until we find the right position for *v*.
    - During this scanning process, v = A[i] is compared with A[j].
    - If A[j] > v, then we v has to be placed somewhere before A[j] in the final sorted sequence. So, A[j] cannot be at its current position (in the final sorted sequence) and has to move at least one position to the right. So, we copy A[j] to A[j+1] and decrement the index j, so that we now compare v with the next element to the left.

$$\underbrace{A[0] \le \cdots \le A[j]}_{\text{smaller than or equal to } A[i]} < \underbrace{A[j+1] \le \cdots \le A[i-1]}_{\text{greater than } A[i]} \;\Big|\; A[i] \cdots A[n-1]$$

    - If A[j] ≤ v, we have found the right position for v; we copy v to A[j+1]. This also provides the stable property, in case v = A[j].

Insertion Sort
Pseudo Code and Analysis

```
ALGORITHM InsertionSort($A[0..n-1]$)
    //Sorts a given array by insertion sort
    //Input: An array $A[0..n-1]$ of $n$ orderable elements
    //Output: Array $A[0..n-1]$ sorted in nondecreasing order
    for $i \leftarrow 1$ to $n-1$ do
        $v \leftarrow A[i]$
        $j \leftarrow i-1$
        while $j \geq 0$ and $A[j] > v$ do
            $A[j+1] \leftarrow A[j]$
            $j \leftarrow j-1$
        $A[j+1] \leftarrow v$
```

The comparison A[j] > v is the basic operation.

Best Case (if the array is already sorted): the element v at A[i] will be just compared with A[i-1] and since A[i-1] ≤ A[i] = v, we retain v at A[i] itself and do not scan the rest of the sequence A[0…i-1]. There is only one comparison for each value of index i.

$$\sum_{i=1}^{n-1} 1 = (n-1) - 1 + 1 = (n-1)$$

Worst Case (if the array is reverse-sorted): the element v at A[i] has to be moved all the way to index 0, by scanning through the entire sequence A[0…i-1].

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

Insertion Sort: Analysis and Example

Average Case: On average for a random input sequence, we would be visiting half of the sorted sequence A[0…i-1] to put A[i] at the proper position.

$$C(n)=\sum_{i=1}^{n-1}\sum_{j=i-1}^{(i-1)/2}1=\sum_{i=1}^{n-1}\frac{(i-1)}{2}+1=\sum_{i=1}^{n-1}\frac{(i+1)}{2}=\Theta(n^2)$$

Example: Given sequence (also initial): **45** 23 8 12 90 21

Index
-1

Iteration 1 (v = 23):

45 45 8 12 90 21
23 45 8 12 90 21

Iteration 2 (v = 8):

23 45 45 12 90 21
23 23 45 12 90 21
8 23 45 12 90 21

Iteration 3 (v = 12):

8 23 45 45 90 21
8 23 23 45 90 21
8 12 23 45 90 21

Iteration 4 (v = 90):

8 12 23 45 90 21
9 12 23 45 90 21

Iteration 5 (v = 21):

9 12 23 45 90 90
9 12 23 45 45 90
9 12 23 23 45 90
9 12 21 23 45 90

Overall time complexity

$$\lim_{n\to\infty}\frac{Best-case}{Worst-case}$$

$$=\lim_{n\to\infty}\frac{n-1}{\left(\frac{n(n-1)}{2}\right)}$$

$$=\lim_{n\to\infty}\frac{2}{n}=0$$

O(n²)

The **colored** elements are in the sorted sequence and the circled element is at index *j* of the algorithm.

2.3 Divide and Conquer

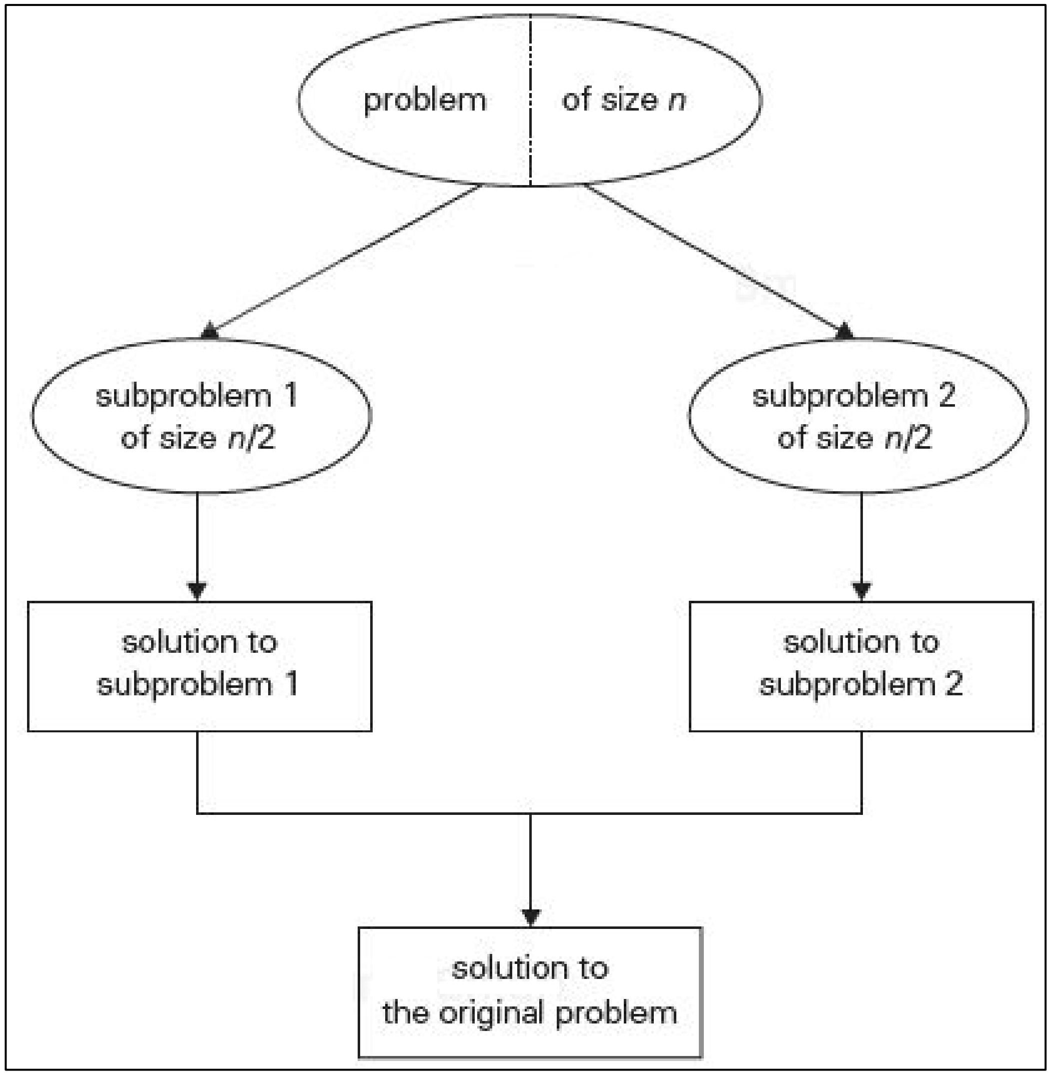
Divide-and-Conquer

The most-well known algorithm design strategy:

1. We divide a problem of instance size 'n' into several sub problems (each of size n/b);

2. Solve 'a' of these sub problems (a ≥ 1; b > 1) recursively and

3. Combine the solutions to these sub problems to obtain a solution for the larger problem.



Typical Case of Divide and Conquer Problems

Master Theorem to Solve Recurrence Relations

- Assuming that size n is a power of b to simplify analysis, we have the following recurrence for the running time, T(n) = a T(n/b) + f(n)
  - where f(n) is a function that accounts for the time spent on dividing an instance of size n into instances of size n/b and combining their solutions.
- Master Theorem:

If $f(n) \in \Theta(n^d)$ where $d \geq 0$, then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

The same results hold good for O and Ω too.

Examples:

1) T(n) = 4T(n/2) + n
a = 4; b = 2; d = 1 ➔ a > b^d
$$T(n) = \Theta\left(n^{\log_2 4}\right) = \Theta(n^2)$$

2) T(n) = 4T(n/2) + n^2
a = 4; b = 2; d = 2 ➔ a = b^d
$$T(n) = \Theta\left(n^2 \log n\right)$$

3) T(n) = 4T(n/2) + n^3
a = 4; b = 2; d = 3 ➔ a < b^d
$$T(n) = \Theta\left(n^3\right)$$

4) T(n) = 2T(n/2) + 1
a = 2; b = 2; d = 0 ➔ a > b^d
$$T(n) = \Theta\left(n^{\log_2 2}\right) = \Theta(n)$$

Master Theorem: More Problems

$T(n) = 3T(n/3) + \sqrt{n}$

$T(n) = 3T(n/3) + n^{(1/2)}$

$a = 3;\ b = 3;\ d = 1/2$

$b^d = 3^{1/2} = 1.732$

$a = 3 > b^d = 1.732$

$T(n) = \Theta(n^{\log_3 3}) = \Theta(n)$

$T(n) = 4T(n/2) + \log n$

$a = 4;\ b = 2;\ d < 1$, because $\log n < n^1$

$b^d = 2^{<1} < 2$

$a > b^d$

$T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$

$T(n) = 6T(n/3) + n^2 \log n$

$a = 6;\ b = 3;\ 2 < d < 3$, because $\log n < n$ and hence $n^2 \log n < n^3$

$b^d = 3^{2<d<3} > 9 > a$

$a < b^d$

Hence, $T(n) = \Theta(n^d) = \Theta(n^2 \log n)$

Merge Sort

- Split array A[0..*n*-1] in two about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

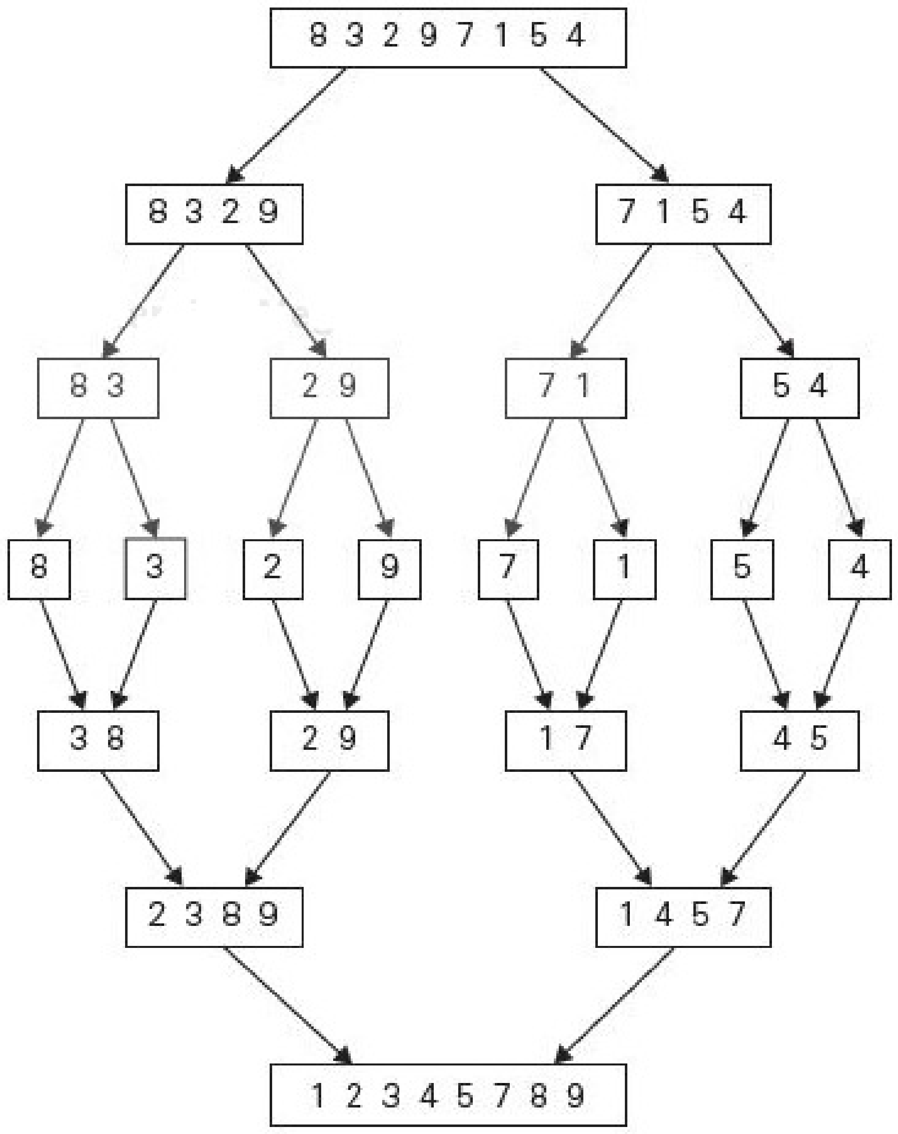
Merge Sort

```
ALGORITHM  Mergesort(A[0..n − 1])
    //Sorts array A[0..n − 1] by recursive mergesort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    if $n > 1$
        copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
        copy $A[\lfloor n/2 \rfloor..n - 1]$ to $C[0..\lceil n/2 \rceil - 1]$
        Mergesort($B[0..\lfloor n/2 \rfloor - 1]$)
        Mergesort($C[0..\lceil n/2 \rceil - 1]$)
        Merge(B, C, A)
```

Merge Algorithm

```
ALGORITHM  $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$
    //Merges two sorted arrays into one sorted array
    //Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
    //Output: Sorted array $A[0..p+q-1]$ of the elements of $B$ and $C$
    $i \leftarrow 0$;  $j \leftarrow 0$;  $k \leftarrow 0$
    while $i < p$ and $j < q$ do
        if $B[i] \leq C[j]$
            $A[k] \leftarrow B[i]$;  $i \leftarrow i + 1$
        else $A[k] \leftarrow C[j]$;  $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    if $i = p$
        copy $C[j..q-1]$ to $A[k..p+q-1]$
    else copy $B[i..p-1]$ to $A[k..p+q-1]$
```

Example for Merge Sort



Analysis of Merge Sort

The recurrence relation for the number of key comparisons C(n) is

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1,\ C(1) = 0$$

At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needed to be processed is reduced by one. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., smaller elements may come from the alternating arrays). Hence, the worst case of $C_{merge}(n) = n - 1$ for $n > 1$.

$$C(n) = 2C(n/2) + (n-1) \qquad f(n) = n - 1 = \Theta(n)$$

Hence, according to Master Theorem,

$$a = 2, b = 2, d = 1 \qquad C(n) \in \Theta(n \log n)$$

$$a = b^d$$

Binary Search

- Binary search is a Θ(log n), highly efficient search algorithm, in a sorted array.
- It works by comparing a search key K with the array's middle element A[m]. If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if K < A[m], and for the second half if K > A[m].
- Though binary search in based on a recursive idea, it can be easily implemented as a non-recursive algorithm.

$$\boxed{\begin{array}{c} K \\ \updownarrow \\ \underbrace{A[0]\dots A[m-1]}_{\substack{\text{search here if}\\ K<A[m]}} \quad A[m] \quad \underbrace{A[m+1]\dots A[n-1]}_{\substack{\text{search here if}\\ K>A[m]}} \end{array}}$$

Binary Search

Example

Search Key
K = 70

l=0	r=12	m=6
l=7	r=12	m=9
l=7	r=8	m=7

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	*l*						*m*						*r*
iteration 2								*l*		*m*			*r*
iteration 3								*l,m*	*r*				

```
ALGORITHM BinarySearch(A[0..n − 1], K)
    //Implements nonrecursive binary search
    //Input: An array A[0..n − 1] sorted in ascending order and
    //          a search key K
    //Output: An index of the array's element that is equal to K
    //          or −1 if there is no such element
    l ← 0;   r ← n − 1
    while l ≤ r do
        m ← ⌊(l + r)/2⌋
        if K = A[m] return m
        else if K < A[m] r ← m − 1
        else l ← m + 1
    return −1
```

Worst-case # Key Comparisons

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 1$$

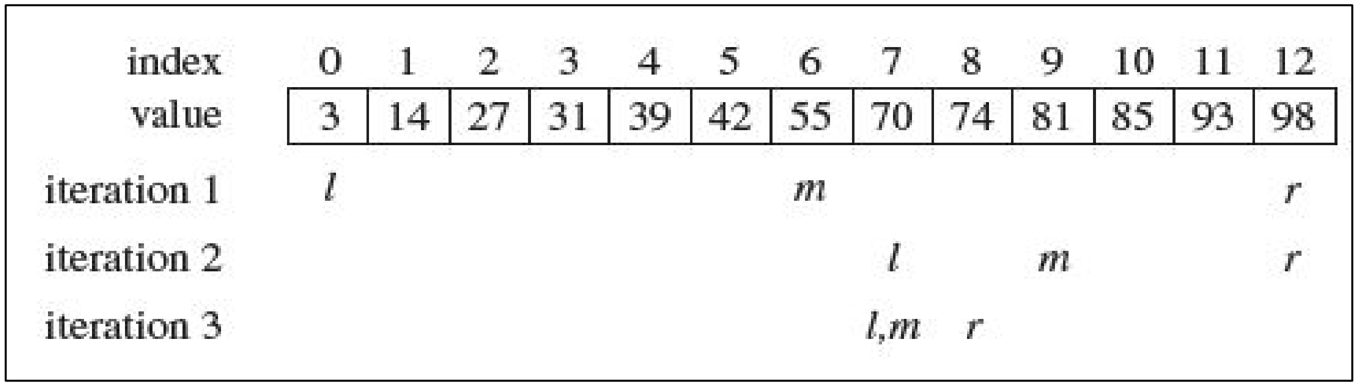
$$C_{worst}(2^k) = k + 1 = \log_2 n + 1.$$

$$C_{worst}(n) = \Theta(\log n)$$

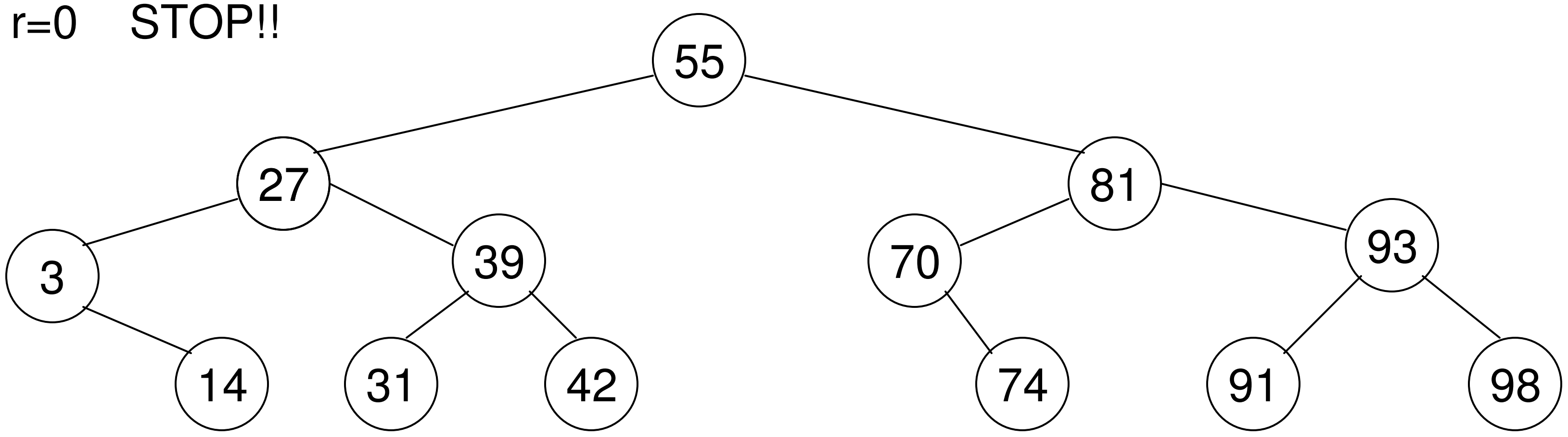
Unsuccessful Search

Search K = 10

l=0	r=12	m=6
l=0	r=5	m=2
l=0	r=1	m=0
l=1	r=1	m=1
l=1	r=0	STOP!!



index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	*l*						*m*						*r*
iteration 2								*l*		*m*			*r*
iteration 3								*l,m*	*r*				

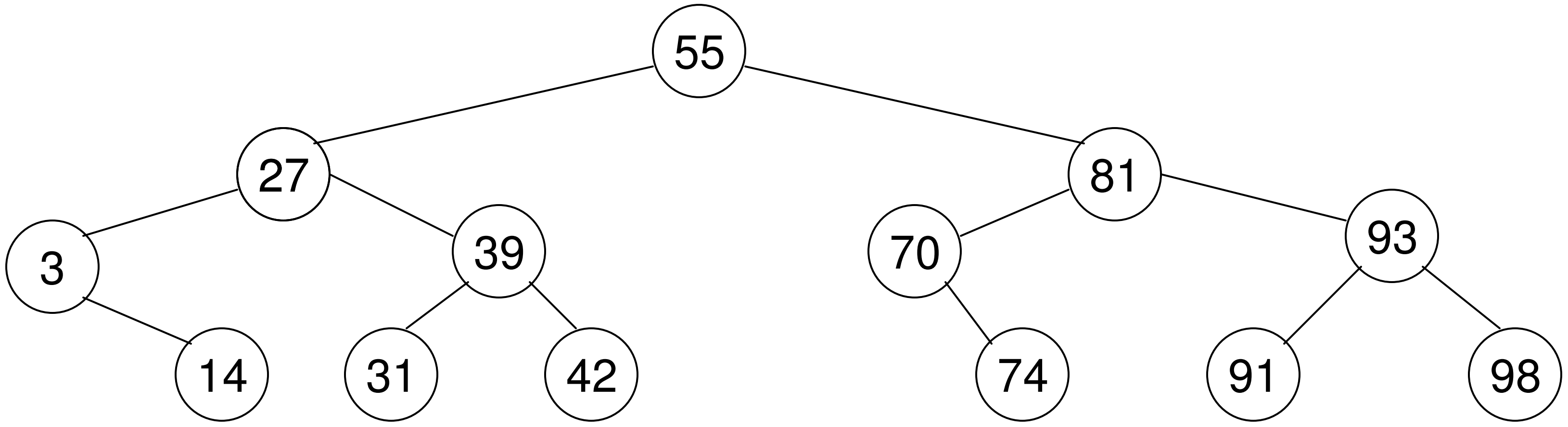


The keys that will require the largest number of comparisons: 14, 31, 42, 74, 91, 98

<u>Average # Comparisons for Successful Search</u>

Keys	# comparisons
55	1
27, 81	2
3, 39, 70, 93	3
14, 31, 42, 74, 91, 98	4

Avg # comparisons
= [Sum of the product of the # keys with certain # comparisons] / [Total Number of keys]
= [(1)(1) + (2)(2) + (3)(4) + (4)(6)] /13
= 3.15



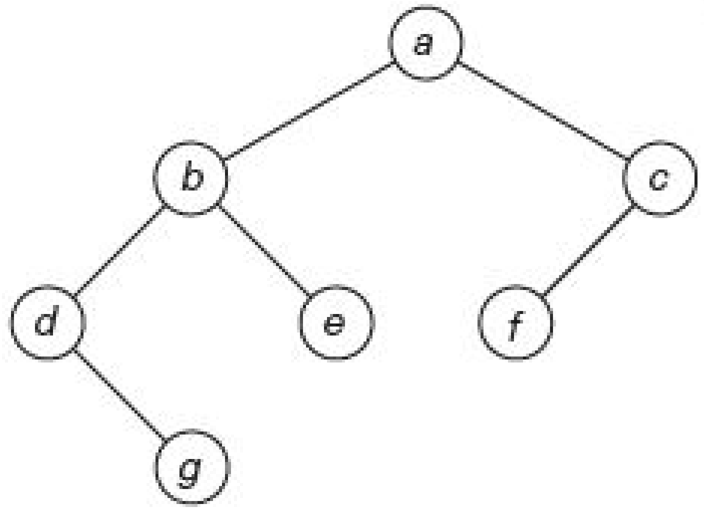
Average # Comparisons for Unsuccessful Search

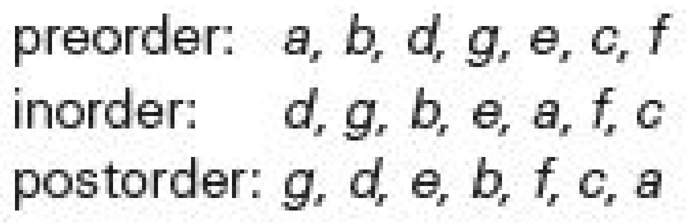
Range of Keys for Unsuccessful search	# comparisons
< 3	3
> 3 and < 14	4
> 14 and < 27	4
> 27 and < 31	4
> 31 and < 39	4
> 39 and < 42	4
> 42 and < 55	4
> 55 and < 70	3
> 70 and < 74	4
> 74 and < 81	4
> 81 and < 91	4
> 91 and < 93	4
> 93 and < 98	4
> 98	4

Avg = [4*12 + 3*2] / 14
= 3.86

Binary Tree Traversals

- A binary tree is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees, called the left and right sub tree of the root.
- The most important divide-and-conquer algorithms for binary trees are the three classic traversals: pre-order, in-order and post-order. All the three traversals visit the nodes of a binary tree recursively, i.e., by visiting the tree's root and its left and right sub trees. They differ only by the timing of the root's visit:
  - **<u>Pre-order traversal:</u>** the root is visited before the left and right sub trees are visited (in that order).
  - **<u>In-order traversal:</u>** the root is visited after visiting its left sub tree but before visiting the right sub tree.
  - **<u>Post-order traversal:</u>** the root is visited after visiting the left and right sub trees (in that order).



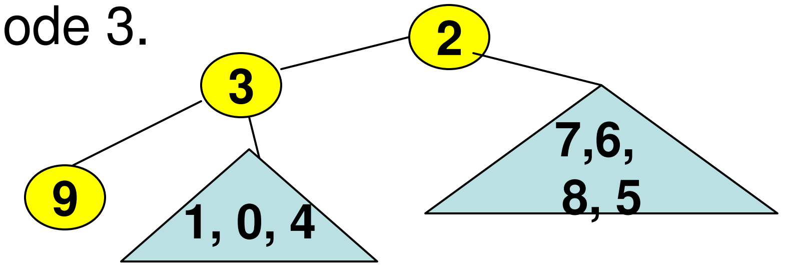


preorder:	*a, b, d, g, e, c, f*
inorder:	*d, g, b, e, a, f, c*
postorder:	*g, d, e, b, f, c, a*

Example to Construct a Binary Tree

- Question: Draw a binary tree with 10 nodes labeled 0, 1, …, in such a way that the in-order and post-order traversals of the tree yield the following lists: 9, 3, 1, 0, 4, 2, 7, 6, 8, 5 (in-order) and 9, 1, 4, 0, 3, 6, 7, 5, 8, 2 (post-order).
- Solution: Note that the post-order traversal always lists the root node of the binary tree as the last node. Hence node '2' is the root node of the binary tree. The in-order traversal lists nodes 9, 3, 1, 0, 4 as the nodes before node '2'. Hence these nodes are in the left sub tree of node 2 and nodes 7, 6, 8, 5 are in the right sub tree of node 2.
- Applying the above logic recursively to the left and right sub trees, we find that the post-order traversal lists the nodes (9, 3, 1, 0, 4) of the left sub tree in the order 9, 1, 4, 0, 3. Hence node 3 is the root node among these nodes. The in-order traversal lists nodes 1, 0, 4 after node 3. Hence, these three nodes constitute the right sub tree of node 3. And node 9 is in the left sub tree of node 3.

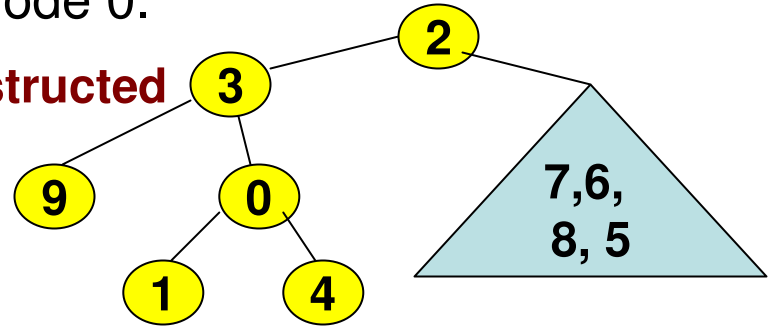
Tree constructed so far:



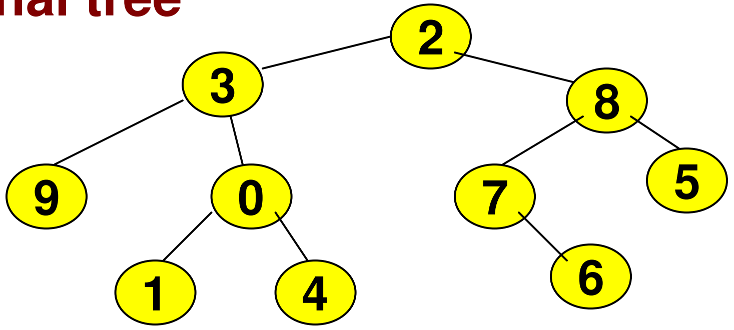
Example to Construct a Binary Tree

- The nodes (1, 0, 4) in the right sub tree of node 3 are listed in the post-order traversal as 1, 4, 0. Hence node 0 is the root of this sub tree. Node 0 is in between nodes 1 and 4 in the in-order list. Hence node 1 should be the left of node 0 and node 4 should be to the right of node 0.

Tree constructed so far:



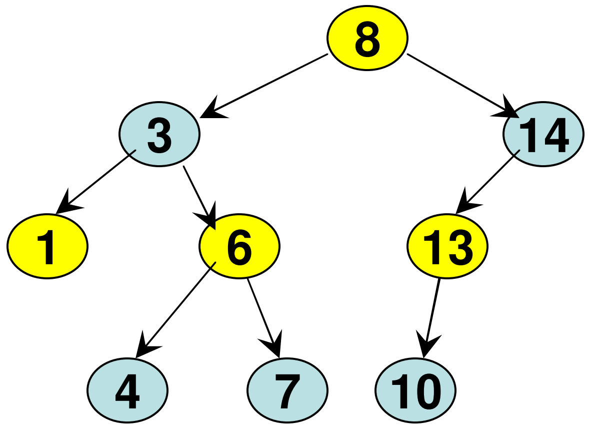
Final tree



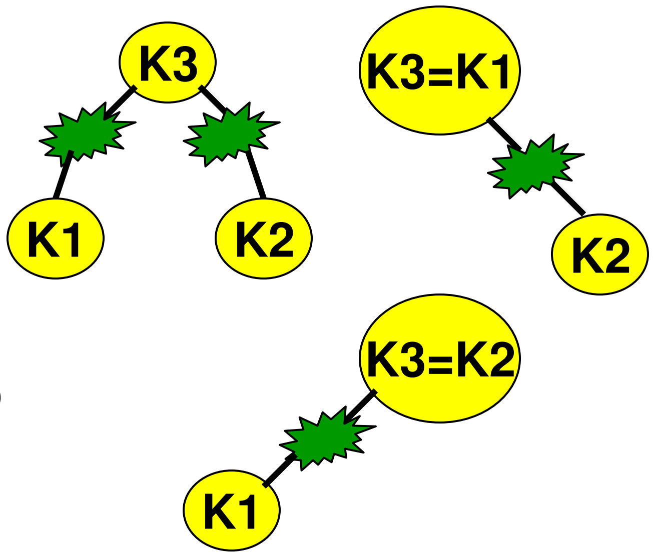
- Continuing our analysis on the right sub tree with nodes (7, 6, 8, 5), we notice that these nodes are listed in the post-order traversal as 6, 7, 5, 8. Hence node 8 should be the root. The position of node 8 in the above in-order list implies that nodes 7, 6 are in the left sub tree of node 8 and node 5 is to the right of node 8.
- Nodes (7, 6) in the left sub tree of node 8 are listed in the post-order traversal as 6, 7. Hence, node 7 should be the root node of this sub tree and according to the in-order list, node 6 should be to the right of node 7.

Binary Search Tree and its Traversal

- A binary search tree (BST) is a sorted binary tree such that:
  - The left sub tree of a node contains only nodes with keys less than the node's key.
  - The right sub tree of a node contains only nodes with keys greater than the node's key.
  - Both the left and right sub trees must also be binary search trees.
- An in-order traversal of a binary search tree lists the keys of the nodes in the tree in a sorted order.
  - **<u>Proof:</u>** Let there be two keys K1 and K2 at two different nodes of a BST such that K1 < K2. So, K1 has to be relatively to the left of K2 somewhere. Let K3 be the key located at their nearest common ancestor.
  - If K3 is different from K1 and K2, then the definition of the BST ensures that K1 and K2 are located in the left and right sub trees of K3 and that K1 is visited before visiting K2.
  - If K3 coincides with K1, then K2 is in the right sub tree of K1. Likewise, if K3 coincides with K2, then K1 is in the left sub tree of K1. Either way, an in-order traversal visits K1 before K2.



In-Order Traversal
1 3 4 6 7 8 10 13 14



Transform and Conquer

Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

Transform-and-Conquer

- This group of techniques solves a problem by a transformation:
  - To a different problem for which an algorithm is already available (problem reduction)
    - Greatest Common Divisor (GCD), Least Common Multiple (LCM), Counting paths in a graph
  - To a different representation of the same instance (representation change)
    - Heap sort

Greatest Common Divisor (GCD)

- Problem: Given two non-zero positive integers, m and n (without loss of generality $m \geq n$), we want to find the gcd(m, n), defined as the largest integer that divides both m and n evenly, i.e., with a remainder of zero.
- Euclid's algorithm
- Theorem: GCD(m, n) = GCD (n, m mod n); for any integer m and n
  - GCD(m, 0) = m; GCD (m, 1) = 1
- Pseudo code
- Input: m, n ($m > 0$ and $n > 0$)
- Output: GCD (m, n)
- Begin Algorithm Euclid (m, n)

```
 **while** n ≠ 0 **do**
     r ← m mod n
     m ← n
     n ← r
 **end while**
 return m // as the gcd
```

- End Algorithm

Note: Euclid's algorithm is an example of an algorithm for the **variable-size-decrease and conquer** technique. The problem size decreases in each iteration of the algorithm; but the decrease is neither by a constant nor by a constant factor.

Also, note that the algorithm is guaranteed to stop because, the second integer gets smaller with each iteration and the algorithm stops when the second integer reaches 0.

GCD Examples

- Note that GCD (n, 0) = n and GCD (n, 1) = 1
- If GCD(m, n) = 1, then ‘m’ and ‘n’ are said to be relatively prime.
- If we are to find the GCD(m, n) where m > n using a brute force approach, it would take n-1 divisions (assuming that we do not divide m and n by 1).
- <u>Examples</u>
- GCD(120, 45)

  GCD(120, 45) = GCD(45, 120 <u>mod</u> 45) = GCD(45, 30)

  GCD(45, 30) = GCD(30, 45 <u>mod</u> 30) = GCD(30, 15)

  GCD(30, 15) = GCD(15, 30 <u>mod</u> 15) = GCD(15, 0) = 15

  # divisions = 3
- GCD(63, 8) = GCD(8, 63 <u>mod</u> 8) = GCD(8, 7)

  GCD(8, 7) = GCD(7, 8 <u>mod</u> 7) = GCD(7, 1) = 1   # divisions = 2

  Since GCD(63, 8) = 1, we say 63 and 8 are relatively prime.

Problem Reduction

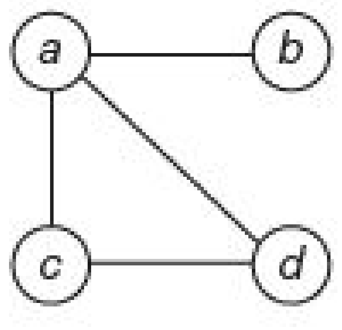
- This variation of transform-and-conquer solves a problem by a transforming it into different problem for which an algorithm is already available.
- To be of practical value, the combined time of the transformation and solving the other problem should be smaller than solving the problem as given by another method.

<u>Examples:</u>

- Computing lcm(*m*, *n*) via computing gcd(*m, n*)

$$\boxed{\operatorname{lcm}(m, n) = \frac{m \cdot n}{\gcd(m, n)}}$$

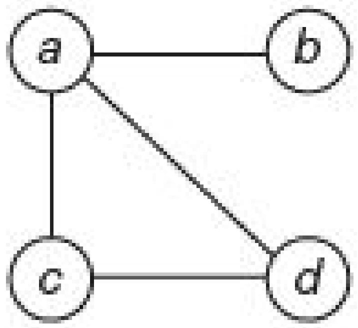
- Counting number of paths of length *n* in a graph by raising the graph's adjacency matrix to the *n*-th power:



$$A = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 1 & 1 \\ b & 1 & 0 & 0 & 0 \\ c & 1 & 0 & 0 & 1 \\ d & 1 & 0 & 1 & 0 \end{array} \qquad A^2 = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 3 & 0 & 1 & 1 \\ b & 0 & 1 & 1 & 1 \\ c & 1 & 1 & 2 & 1 \\ d & 1 & 1 & 1 & 2 \end{array}$$

To find # Paths of Length ‘n’

# Paths of Length 4: Find A^4.



$$A^2 = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 3 & 0 & 1 & 1 \\ b & 0 & 1 & 1 & 1 \\ c & 1 & 1 & 2 & 1 \\ d & 1 & 1 & 1 & 2 \end{array} \quad \mathbf{x} \quad A^2 = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 3 & 0 & 1 & 1 \\ b & 0 & 1 & 1 & 1 \\ c & 1 & 1 & 2 & 1 \\ d & 1 & 1 & 1 & 2 \end{array}$$

$$A^4 = \begin{array}{c|cccc} & \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} \\ \hline \mathbf{a} & 11 & 2 & 6 & 6 \\ \mathbf{b} & 2 & 3 & 4 & 4 \\ \mathbf{c} & 6 & 4 & 5 & 6 \\ \mathbf{d} & 6 & 4 & 6 & 7 \end{array}$$

To find the number of paths length 4 between vertices b and c, just simply do a pair-wise multiplication and addition of the elements corresponding to the row for vertex ‘b’ in A^2 with the elements corresponding to the column for vertex ‘c’ in A^2.

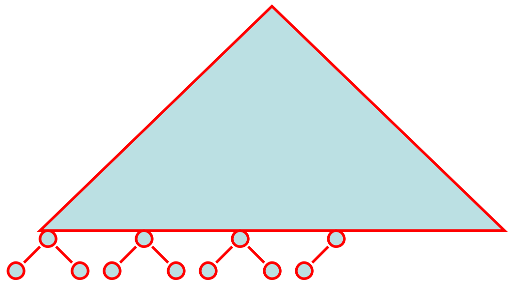
Note: Rule for Matrix Multiplication
To find the value of an entry in cell (i, j) in the product matrix P = A * B,
Do a pair-wise multiplication and addition of the elements in row ‘i’ of the first matrix A and the elements in column ‘j’ of the second matrix B.

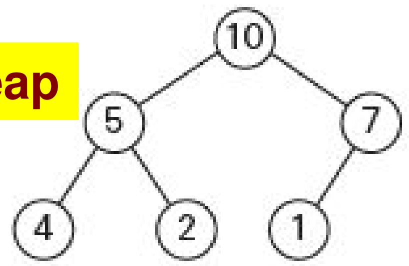
Heap

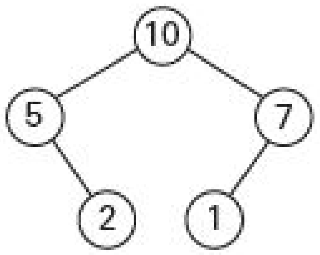
Definition A *heap* is a binary tree with keys at its nodes (one key per node) such that:

- It is essentially complete, i.e., all its levels are full except possibly the last level, where only some rightmost keys may be missing

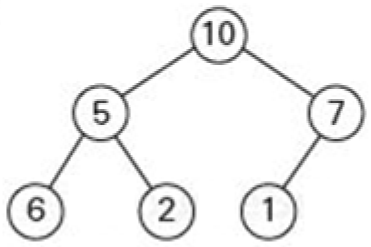
Example for a Heap







Not a Heap



Not a Heap

- The key at each node is ≥ keys at its children (Max. Heap)
- We will focus on Max. Heap in this chapter. Note that for a Min. Heap, the value for the key at a node is <= the value for the keys at its children. [In other words, Max. Heap is the one whose root has the largest value; Min. Heap is the one whose Root has smallest value]
- Heap’s elements are ordered top down (along any path down from its root), but they are not ordered left to right

Important Properties of a Heap

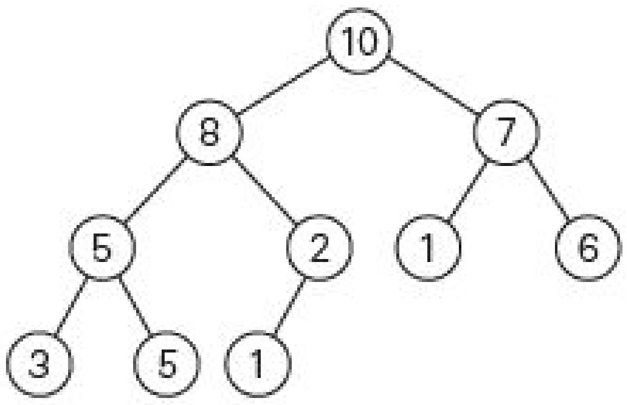
- Given *n,* there exists a unique binary tree with *n* nodes that is essentially complete, with height, $h = \lfloor \log_2 n \rfloor$ The root contains the largest key (Max. Heap)
- The sub tree rooted at any node of a heap is also a heap
- A heap can be represented as an array

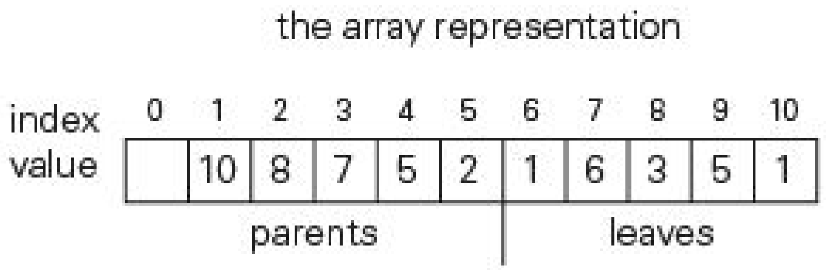
- <u>Use of Max. Heap to Implement a Priority Queue</u>
  - A priority queue (implemented as a Max. Heap) is not FIFO-based. Here the elements are stored in the decreasing order of the key values.
  - Heap can be used to maintain the elements of a priority queue such that the element whose key has the highest priority is at the top of the heap and is removed from the heap as a result of a dequeue operation.
  - Any insertion to the heap will also be taken care of through the "heapification" step and the element will be inserted at its appropriate position in the heap.

Heap’s Array Representation

Store heap’s elements in an array (whose elements indexed, for convenience, 1 to *n*) in top-down left-to-right order

<u>Example:</u>





the array representation

index	0	1	2	3	4	5	6	7	8	9	10
value		10	8	7	5	2	1	6	3	5	1

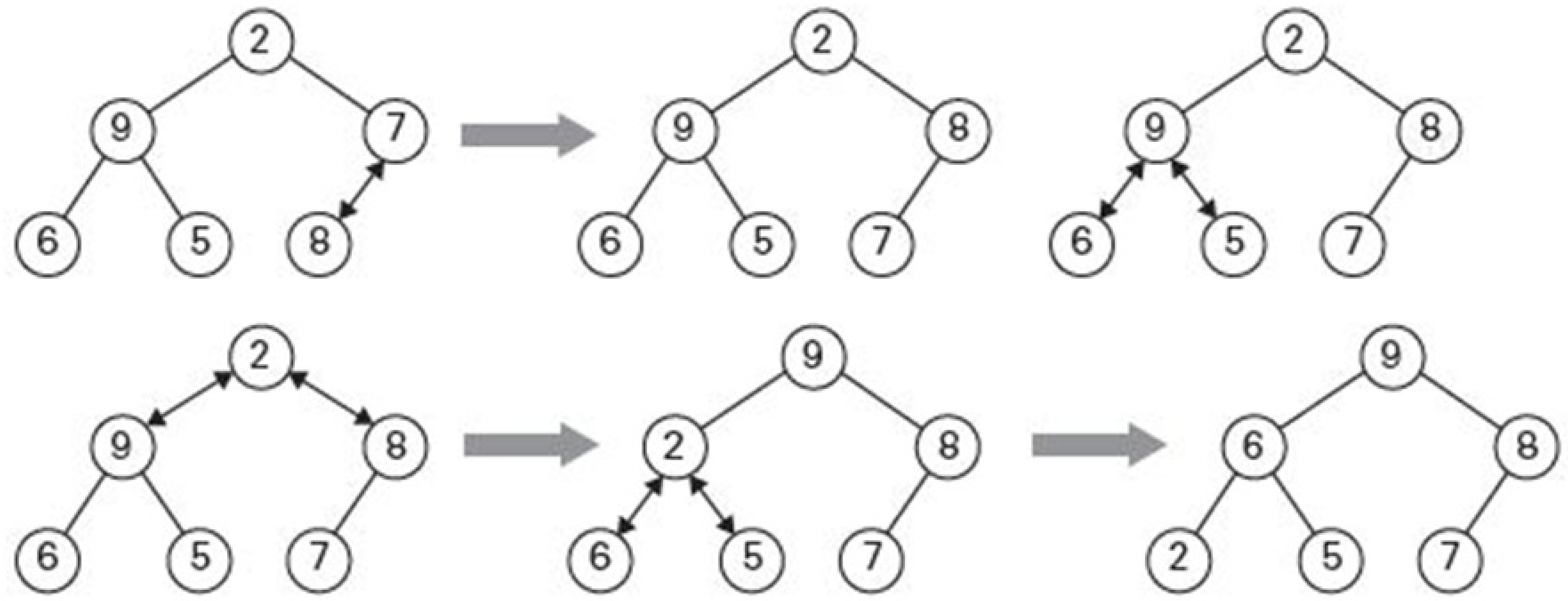
parents | leaves

- Left child of node *j* is at 2*j*
- Right child of node *j* is at 2*j*+1
- Parent of node *j* is at $\lfloor j/2 \rfloor$
- Parental nodes are represented in the first $\lfloor n/2 \rfloor$ locations
- For convenience, it is better to start the array index from 1. Index 0 can be filled with a dummy sentinel value, like -10000, that will not be part of the heap.

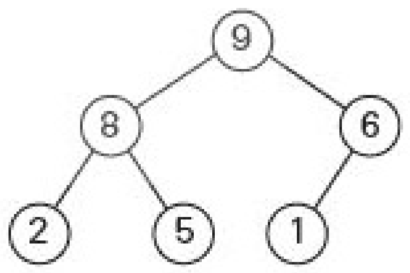
Heap Construction (Bottom-Up)

- Step 0: Initialize the structure with keys in the order given
- Step 1: Starting with the last (rightmost) parental node, fix the heap rooted at it, if it doesn't satisfy the heap condition: keep exchanging it with its largest child until the heap condition holds
- Step 2: Repeat Step 1 for the preceding parental node

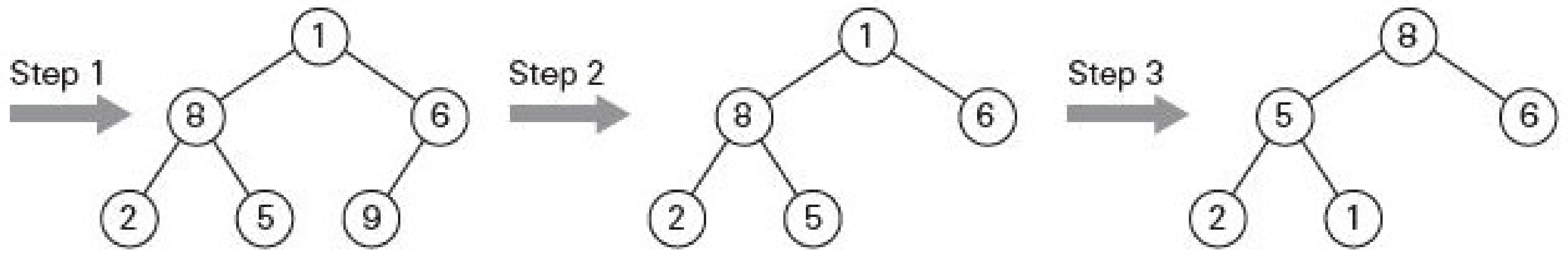
Example: **Given initial list:** **2, 9, 7, 6, 5, 8** **Heapified Array: 9 6 8 2 5 7**



Deleting the root key from the Heap



Removing key corresponding to Root node ‘9’



The key to be deleted is swapped with the last key after which the smaller tree is “heapified” by exchanging the new key in its root with the right most key among its leaf nodes (at the maximum height), until the parent dominance requirement is satisfied

Heap Sort

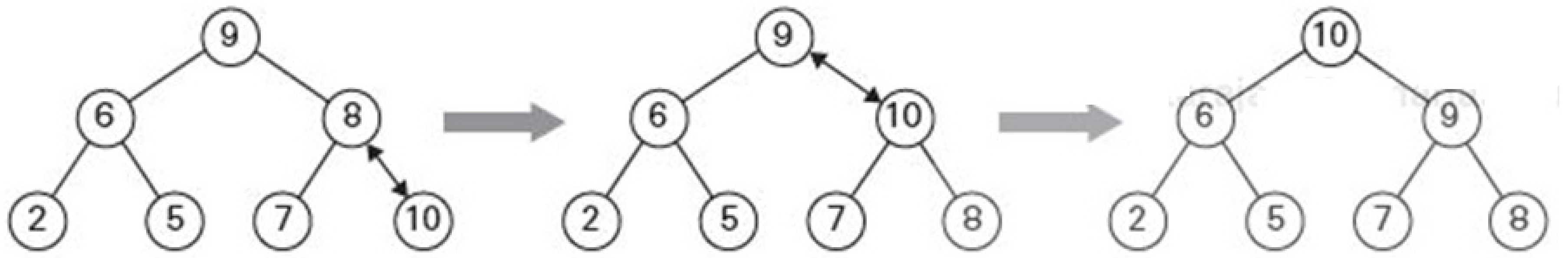
- Stage 1:
  - (Bottom-up approach) Construct a heap for a given list of *n* keys: Θ(n) time
  - (Top-down approach) Construct a heap by inserting one key at a time to an already existing heap: Θ(nlogn) time
- Stage 2: Repeat operation of root removal *n*-1 times: Θ(nlogn) time
  - Exchange keys in the root and in the last (rightmost) leaf
  - Decrease heap size by 1
  - If necessary, swap new root with larger child until the heap condition holds

Overall time complexity of Heap Sort = Θ(nlogn), for heaps constructed using bottom-up and top-down strategies

Insertion of a New Element into a Heap

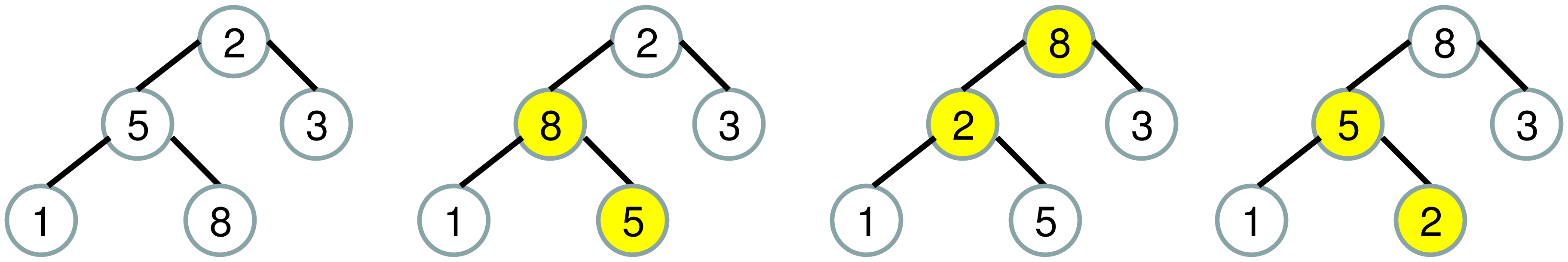
- Used as the Top-Down approach
- Insert the new element at last position in heap.
- Compare it with its parent and, if it violates heap condition, exchange them
- Continue comparing the new element with nodes up the tree until the heap condition is satisfied
- Efficiency: Θ(log n)

Example: Inserting Key ’10’ into the heap

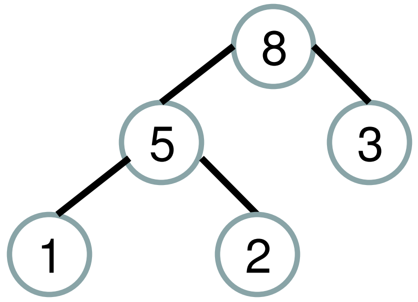


Example 1
2, 5, 3, 1, 8

Bottom-Up Construction



Proper (Initial) Heap

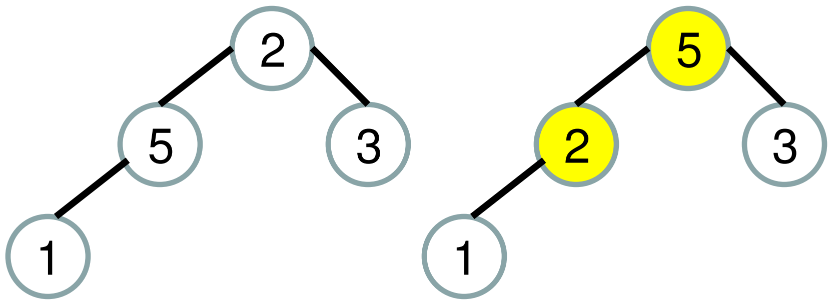


Sorting the Array

Initial Array (satisfying the heap property)

```
-10000   8        5        3        1        2
```

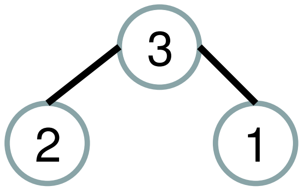
Iteration # 1: Remove key 8



Array sorting in progress

```
-10000   5        2        3        1        8
```

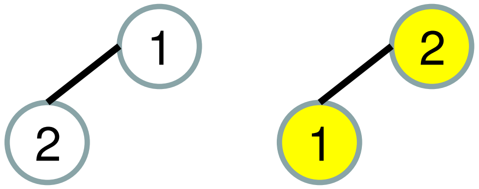
Iteration # 2: Remove key 5



Array sorting in progress

-10000 3 2 1 5 8

Iteration # 3: Remove key 3

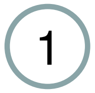


Array sorting in progress

-10000 2 1 3 5 8

Array sorting in progress

Iteration # 4: Remove key 2



-10000 1 2 3 5 8

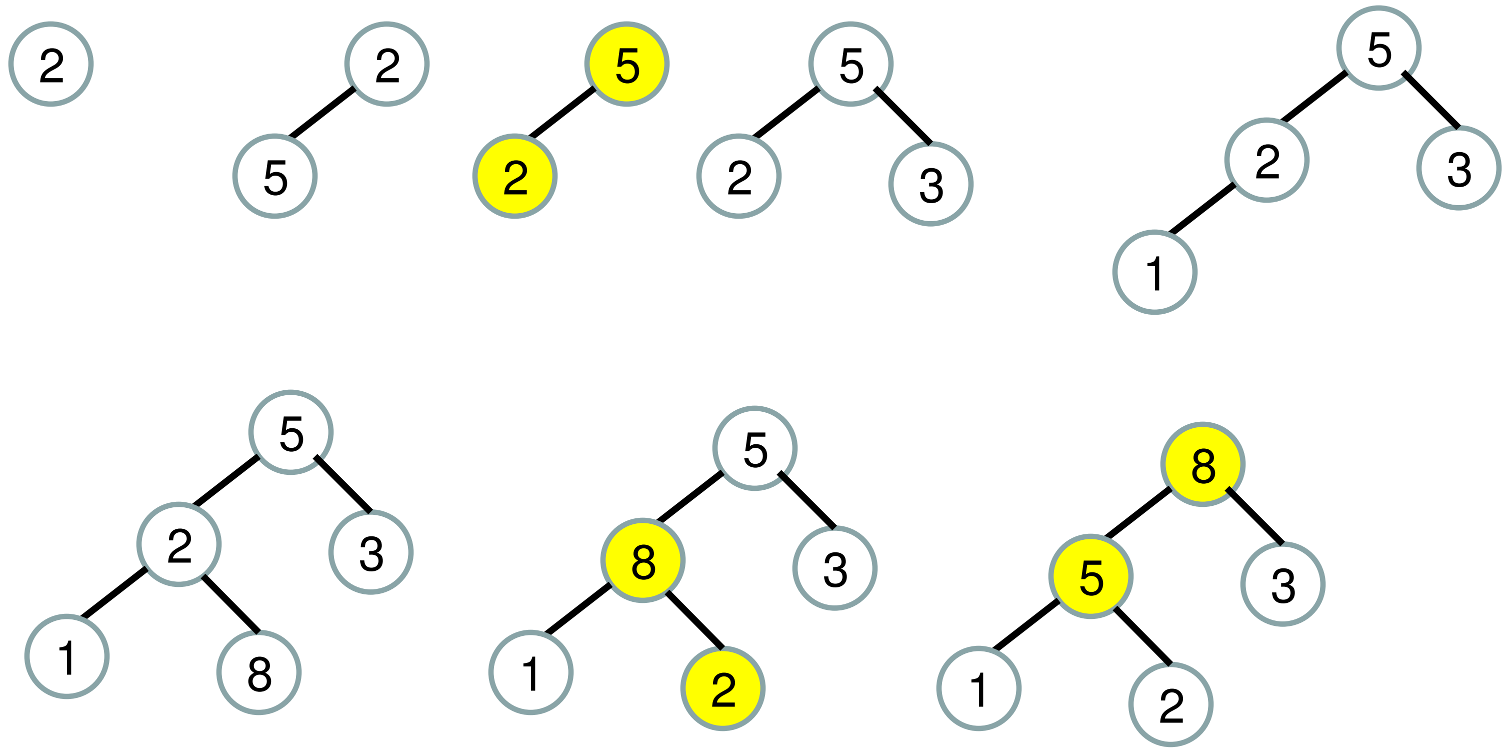
Final sorted array

Iteration # 5: Remove key 1

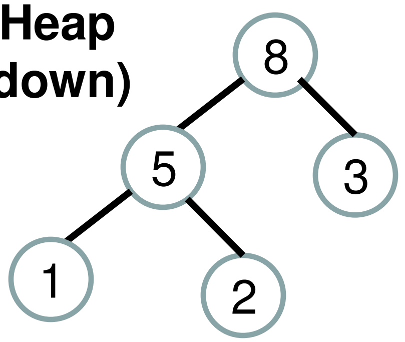
-10000 1 2 3 5 8

Example 1
2, 5, 3, 1, 8

Top-Down Construction

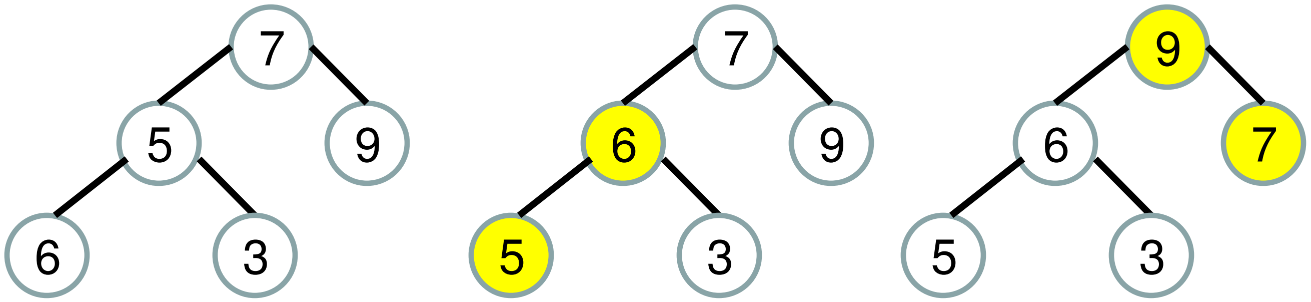


Final-Heap (Top-down)



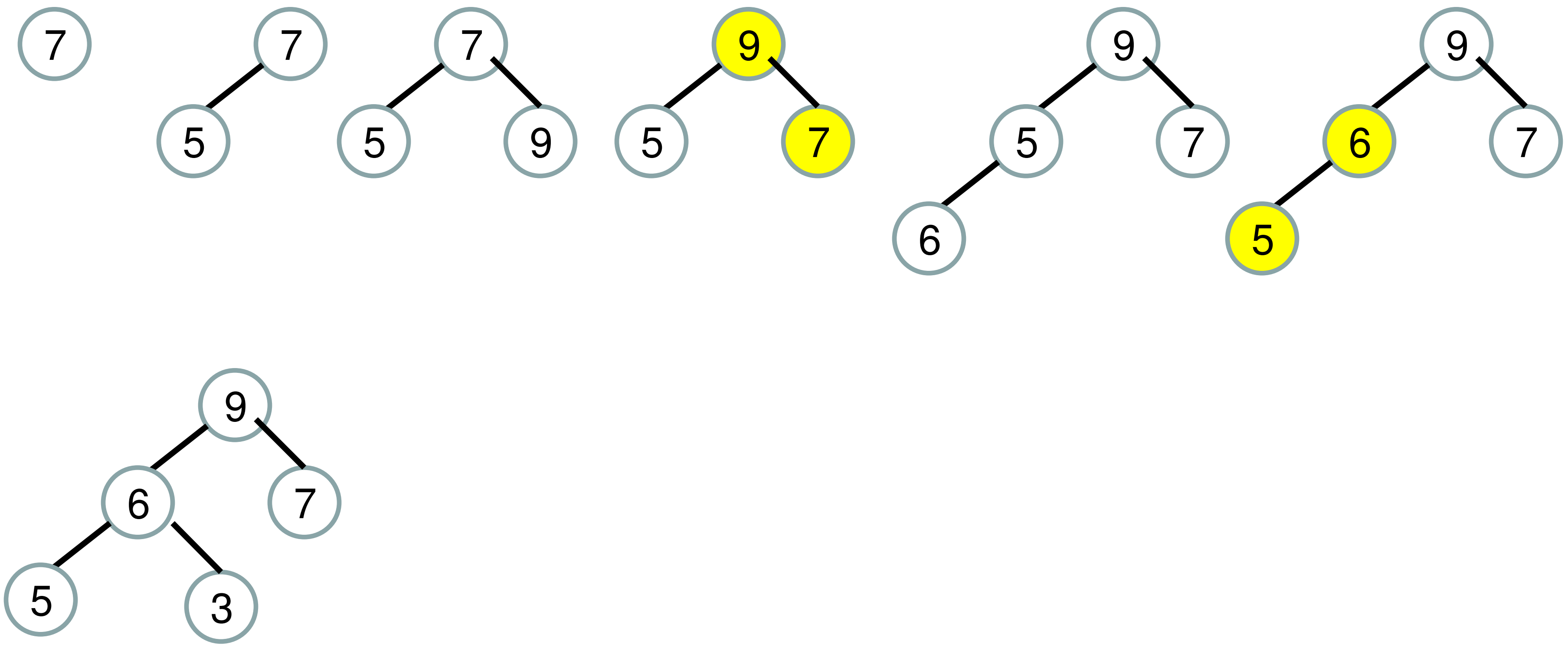
Example 2
7, 5, 9, 6, 3

Bottom-Up Construction



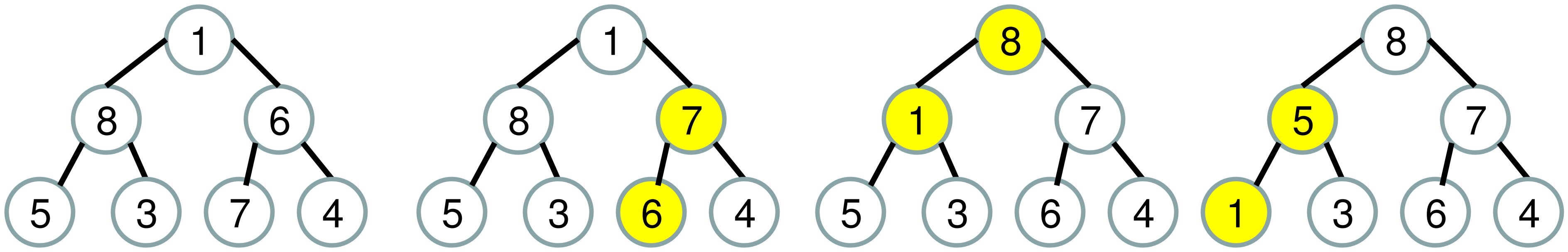
Example 2
7, 5, 9, 6, 3

Top-Down Construction

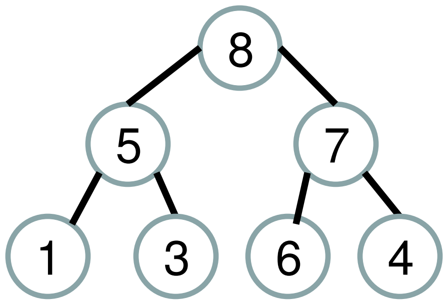


Example 3
1, 8, 6, 5, 3, 7, 4

Bottom-Up Construction



Proper (Initial) Heap



Sorting the Array

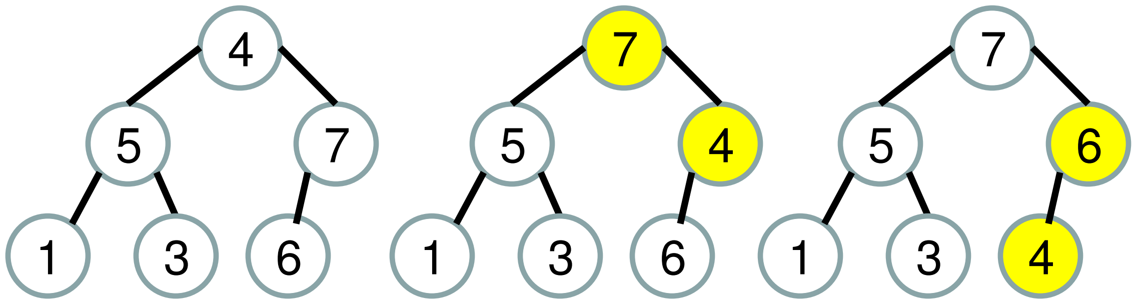
Initial Array (satisfying the heap property)

```
-10000   8     5     7     1     3     6     4
```

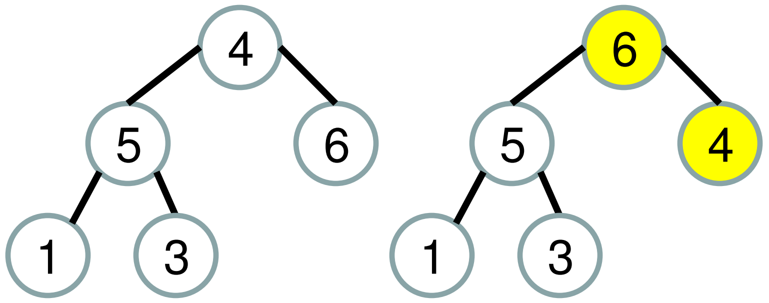
Array sorting in progress

```
-10000   7     5     6     1     3     4     8
```

Iteration # 1: Remove key 8



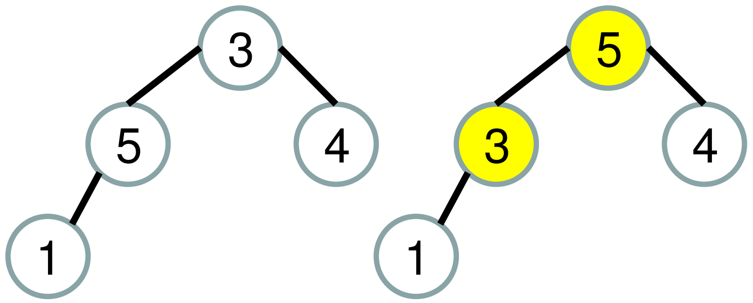
Iteration # 2: Remove key 7



Array sorting in progress

-10000 6 5 4 1 3 7 8

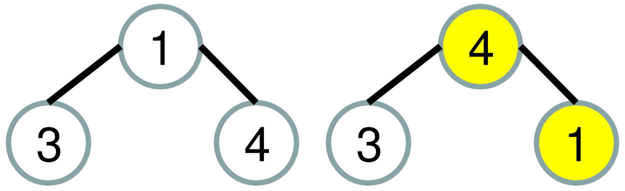
Iteration # 3: Remove key 6



Array sorting in progress

-10000 5 3 4 1 6 7 8

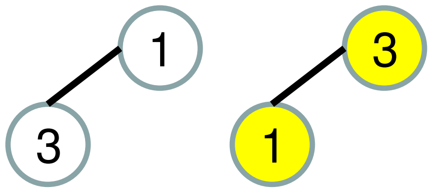
Iteration # 4: Remove key 5



Array sorting in progress

-10000 4 3 1 5 6 7 8

Iteration # 5: Remove key 4



Array sorting in progress

-10000 3 1 4 5 6 7 8

Iteration # 6: Remove key 3

Array sorting in progress

-10000 1 3 4 5 6 7 8

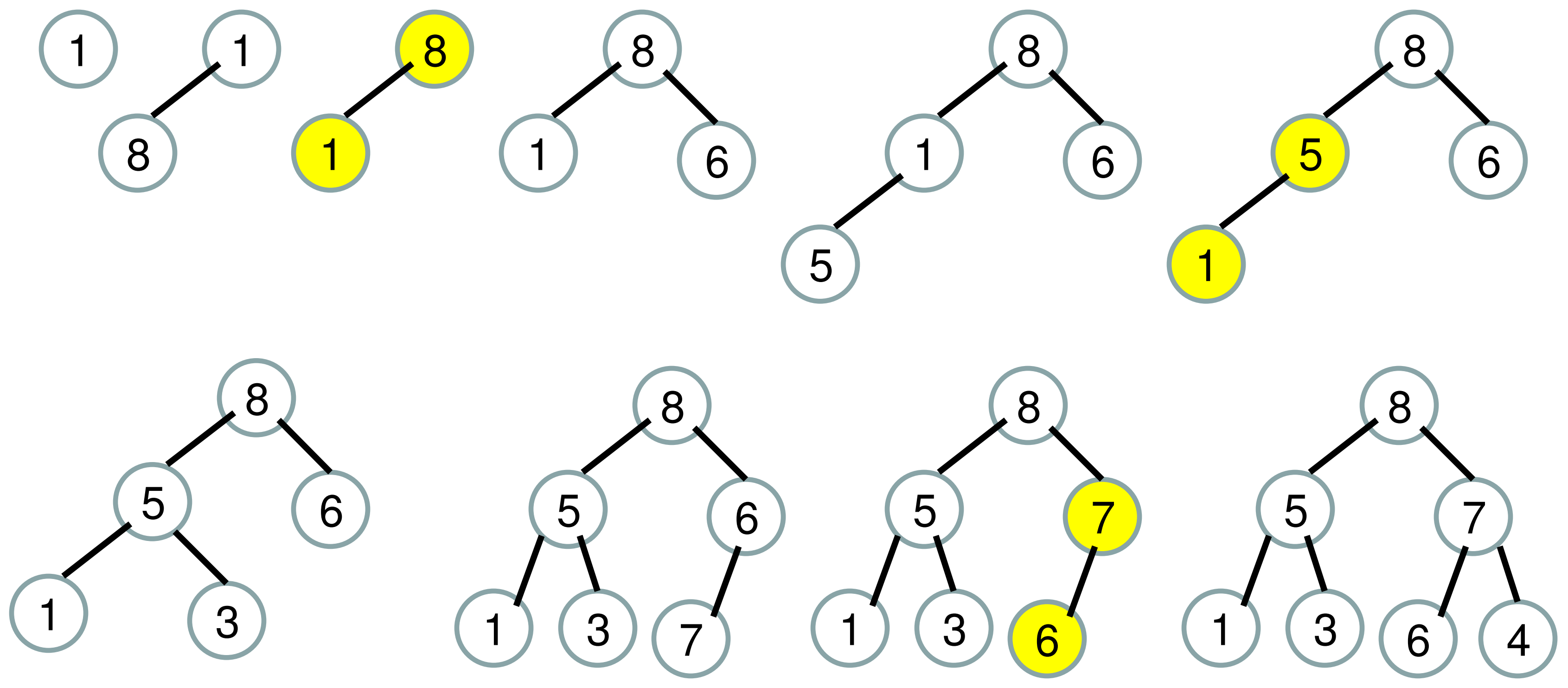
Iteration # 7: Remove key 1

Array sorting in progress

-10000 1 3 4 5 6 7 8

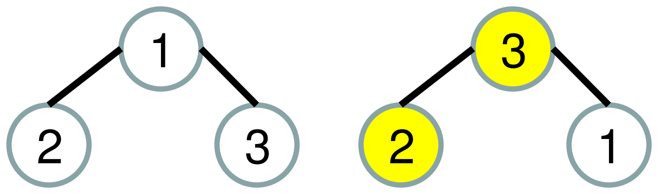
Example 3
1, 8, 6, 5, 3, 7, 4

Top-down Construction



Example 4

1, 2, 3



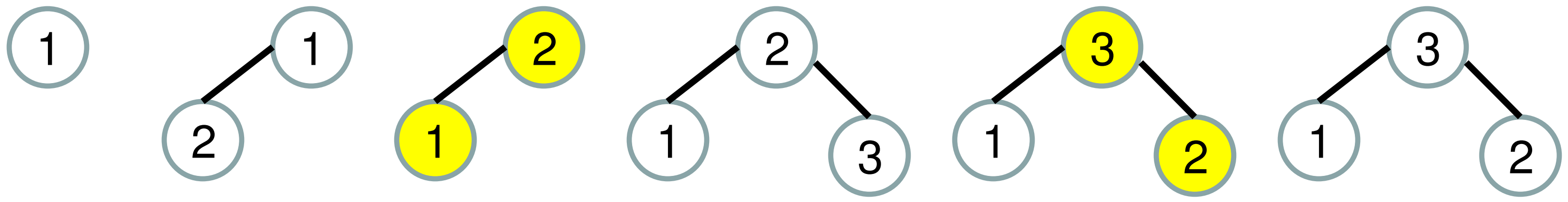
Bottom-Up Construction

Array (satisfying the heap property)

-10000 3 2 1

1, 2, 3

Top-Down Construction



Array (satisfying the heap property)

-10000 3 1 2

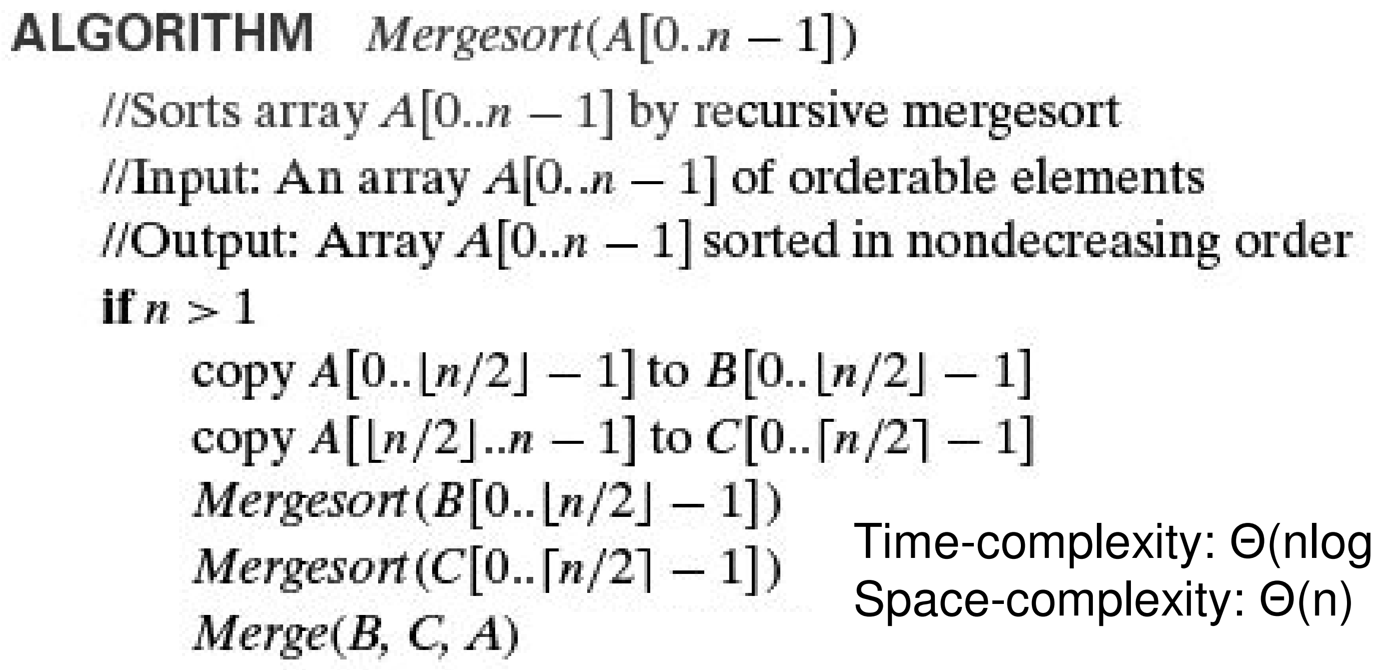
Thus, for a given input sequence, the arrays (satisfying the heap property) that are constructed using the bottom-up approach and the top-down approach need not always be the same, as observed in the above example.

2.5 Space-Time Tradeoff

In-place vs. Out-of-place Algorithms

- An algorithm is said to be "in-place" if it uses a minimal and/or constant amount of extra storage space to transform or process an input to obtain the desired output.
  - Depending on the nature of the problem, an in-place algorithm may sometime overwrite an input to the desired output as the algorithm executes (as in the case of in-place sorting algorithms); the output space may sometimes be a constant (for example in the case of string-matching algorithms).
- Algorithms that use significant amount of extra storage space (sometimes, additional space as large as the input – example: merge sort) are said to be out-of-place in nature.
- Time-Space Complexity Tradeoffs of Sorting Algorithms:
  - In-place sorting algorithms like Selection Sort, Bubble Sort, Insertion Sort and Quick Sort have a worst-case time complexity of $\Theta(n^2)$.
  - On the other hand, Merge sort has a space-complexity of $\Theta(n)$, but has a worst-case time complexity of $\Theta(n\log n)$.

Time and Space Complexity Analysis of Recursive Sorting Algorithms



```
ALGORITHM  Mergesort(A[0..n − 1])
    //Sorts array A[0..n − 1] by recursive mergesort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    if n > 1
        copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
        copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
        Mergesort(B[0..⌊n/2⌋ − 1])
        Mergesort(C[0..⌈n/2⌉ − 1])
        Merge(B, C, A)
```

Time-complexity: Θ(nlogn)
Space-complexity: Θ(n)

- Merge Sort:
- Is the algorithm in-place?
- Answer: No, The algorithm requires an equal amount of additional space as that of the original input array for each recursive call of the algorithm. The two sub-arrays B and C are stored in a different memory location and are not part of the original array A.

Time-Space Complexity of Heap Sort

- Heap sort is probably the best algorithm we have seen in this course with respect to time and space complexity. <u>It is an in-place algorithm</u> with all the heapify and element rearrangement operations conductible in the input array itself and no additional space is needed.

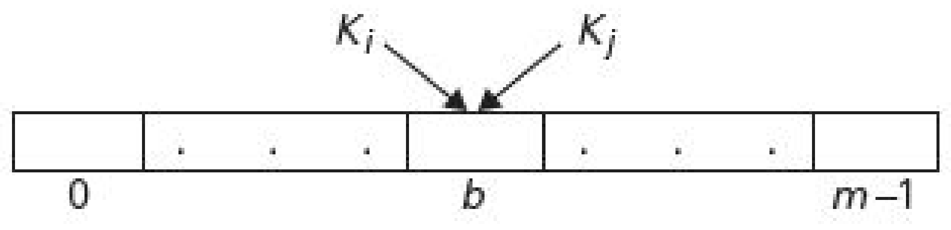
- As we know, there are two stages of heap sort of n elements.
  - <u>**Stage 1:**</u> ***Construct the heap*** – can be done with a top-down strategy in Θ(nlogn) time or with a bottom-up strategy in Θ(n) time.
  - <u>**Stage 2:**</u> ***Remove the root n-1 times***. Each time a root is removed, the binary tree has to be re-heapified to make it become a heap. This involves moving down the new temporary root all the way to an appropriate position in the heap, and moving the largest element among the remaining unsorted elements in the binary tree as the root of the heap. This can be done in Θ(logn) time for each root removal. Hence, Θ(nlogn) time for n-1 root removals.
  - The overall time-complexity of heap sort is thus
  - $\{\underbrace{\Theta(n\log n)}_{\text{Top-down}} \text{ or } \underbrace{\Theta(n)}_{\text{Bottom-up}}\} + \Theta(n\log n) = \Theta(n\log n)$.

Hashing

- A very efficient method for implementing a *dictionary,* i.e., a set with the operations: find, insert and delete
- Based on representation-change and space-for-time tradeoff ideas
- We consider the problem of implementing a dictionary of *n* records with keys K_1, K_2, …, K_n.
- Hashing is based on the idea of distributing keys among a one-dimensional array H[0…m-1] called a hash table.
  - The distribution is done by computing, for each of the keys, the value of some pre-defined function *h* called the ***hash function***.
  - The hash function assigns an integer between 0 and *m*-1, called the hash address to a key.
  - The size of a hash table *m* is typically a prime integer.
- Typical hash functions
  - For non-negative integers as key, a hash function could be h(K)=K mod m;
  - If the keys are letters of some alphabet, the position of the letter in the alphabet (for example, A is at position 1 in alphabet A – Z) could be used as the key for the hash function defined above.
  - If the key is a character string c_0 c_1 … c_{s-1} of characters from an alphabet, then, the hash function could be: $(\sum_{i=0}^{s-1} ord(c_i)) \bmod m$

Collisions and Collision Resolution

If $h(K_1) = h(K_2)$, there is a *collision*



- Good hash functions result in fewer collisions but some collisions should be expected
- Two principal hashing schemes handle collisions differently:
  - *Open hashing*
    – each cell is a header of linked list of all keys hashed to it
  - *Closed hashing*
    - one key per cell
    - in case of collision, finds another cell by
      - *linear probing:* use next free bucket
      - *double hashing:* use second hash function to compute increment

Open Hashing

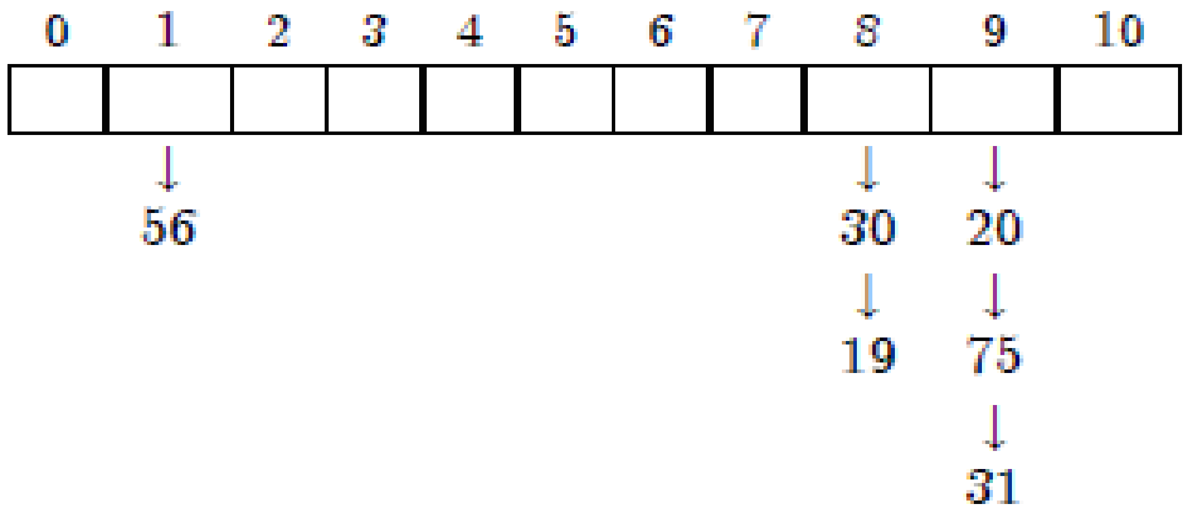
The list of keys: 30, 20, 56, 75, 31, 19

The hash function: $h(K) = K \bmod 11$

The hash addresses:

K	30	20	56	75	31	19
$h(K)$	8	9	1	9	9	8

The open hash table:



The largest number of key comparisons in a successful search in this table is 3 (in searching for $K = 31$).

The average number of key comparisons in a successful search in this table, assuming that a search for each of the six keys is equally likely, is

$$\frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{6} \cdot 2 = \frac{10}{6} \approx 1.7.$$

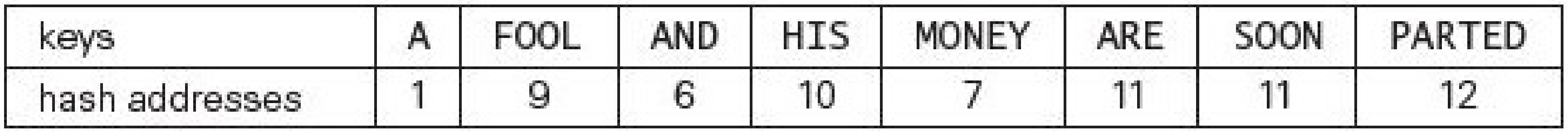
Open Hashing (Separate Chaining)

Keys are stored in linked lists <u>outside</u> a hash table whose elements serve as the lists' headers.

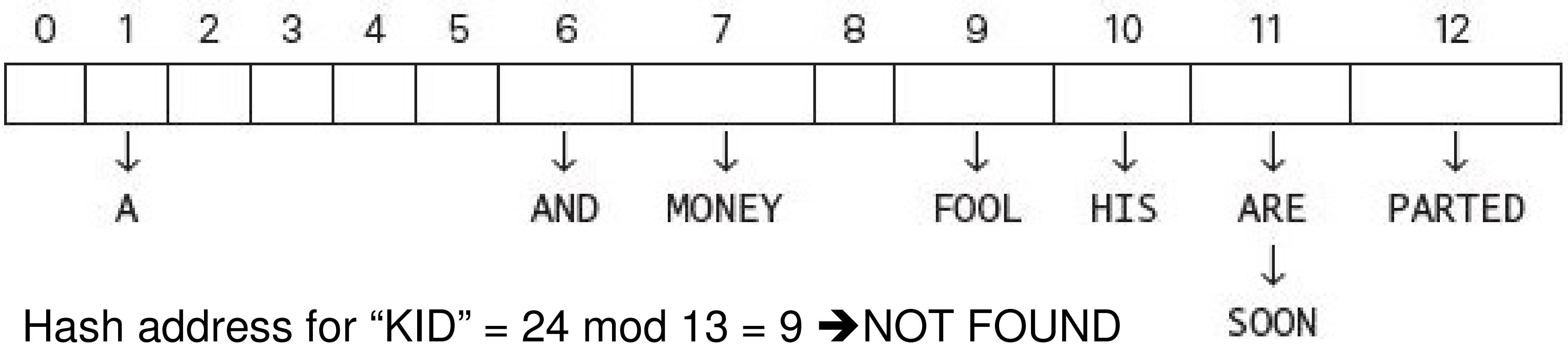
<u>Example:</u> A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED

h(*K*) = sum of *K*'s letters' positions in the alphabet MOD 13

A – 1	D – 4	G – 7	J – 10	M – 13	P – 16	S – 19	V – 22	Y – 25
B – 2	E – 5	H – 8	K – 11	N – 14	Q – 17	T – 20	W – 23	Z – 26
C – 3	F – 6	I – 9	L – 12	O – 15	R – 18	U – 21	X – 24	



keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash addresses	1	9	6	10	7	11	11	12



Hash address for "KID" = 24 mod 13 = 9 ➔NOT FOUND

Open Hashing

- Inserting and Deleting from the hash table is of the same complexity as searching.
- If hash function distributes keys uniformly, average length of linked list will be $\alpha = n/m$. This ratio is called *load factor*.
- Average-case number of key comparisons for a successful search is $\alpha/2$; Average-case number of key comparisons for an unsuccessful search is α.
- Worst-case number of key comparisons is $\Theta(n)$ – occurs if we get a linked list containing all the n elements hashing to the same index. To avoid this, we need to be careful in selecting a proper hashing function.
  - Mod-based hashing functions with a prime integer as the divisor are more likely to result in hash values that are evenly distributed across the keys.
- Open hashing still works if the number of keys, $n >$ the size of the hash table, m.

Closed Hashing

- All keys are stored in the hash table itself without the use of linked lists.
  - The size of the hash table (m) has to be at least as large as the number of keys (n). That is, m >= n ➔ n ≤ m.
- Collision resolution could be avoided through linear probing or through the use of a secondary hash function.
- With linear probing, we check the cell following the one where the collision occurs.
  - If that cell is empty, the new key is installed there.
  - If the next cell is already occupied, the availability of that cell's immediate successor is checked and so on, until we find an empty cell. If the end of the hash table is reached, we wrap around.
- The search for a given key K is done by computing its hash value h(K) and locating the cell with this hash address.
  - If the cell h(K) is empty, the search is unsuccessful.
  - If the cell is not empty, we must compare K with the contents of the cell: if they are equal, we have found a matching key; if they are not, we compare K with a key in the next cell and continue in this manner until we encounter either a matching key (a successful search) or {an empty cell or traversed the whole hash table without finding the key (unsuccessful search)}.

Closed Hashing

The list of keys: 30, 20, 56, 75, 31, 19

The hash function: $h(K) = K \bmod 11$

The hash addresses:

K	30	20	56	75	31	19
$h(K)$	8	9	1	9	9	8

0	1	2	3	4	5	6	7	8	9	10
								30		
								30	20	
	56							30	20	
	56							30	20	75
31	56							30	20	75
31	56	19						30	20	75

The largest number of key comparisons in a successful search is 6 (when searching for $K = 19$).

The average number of key comparisons in a successful search in this table, assuming that a search for each of the six keys is equally likely, is

$$\frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{6} \cdot 6 = \frac{14}{6} \approx 2.3.$$

Lazy Deletion

- With deletions in a Closed Hashing Table, if we simply delete a key, then we may not be able to successfully search for a key that has the same hash value as that of the key being deleted.
- With Lazy Deletion, the previously occupied locations of the deleted keys can be marked by a special symbol (or a dummy value, say -10000) to distinguish them from locations that have been actually occupied.
- The locations containing the special symbols are considered to be available for key insertions.
- However, during a key search, these locations are considered to be occupied.

0	1	2	3	4	5	6	7	8	9	10
31	56	19						30	20	75

Upon deleting key - 20

0	1	2	3	4	5	6	7	8	9	10
31	56	19						30	-1000	75

A search for key 75 requires 2 comparisons

Upon deleting key - 56

0	1	2	3	4	5	6	7	8	9	10
31	-1000	19						30	-1000	75

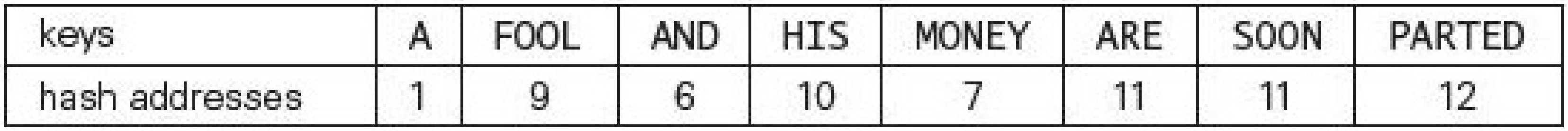
A search for key 54 in the above hash table requires 4 key comparisons…

To insert key 21

0	1	2	3	4	5	6	7	8	9	10
31	21	19						30	-1000	75

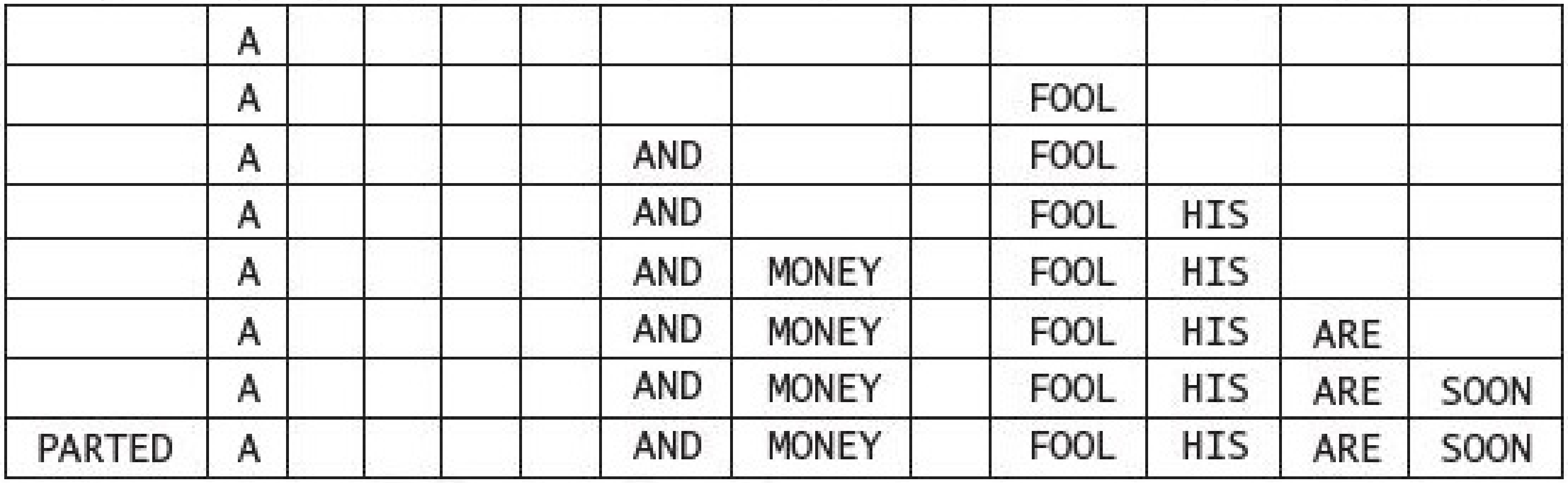
To insert key 21, it would require 3 key comparisons

Example 2 for Closed Hashing



keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash addresses	1	9	6	10	7	11	11	12

h(*K*) = sum of *K*'s letters' positions in the alphabet MOD 13



0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A								FOOL			
	A					AND			FOOL			
	A					AND			FOOL	HIS		
	A					AND	MONEY		FOOL	HIS		
	A					AND	MONEY		FOOL	HIS	ARE	
	A					AND	MONEY		FOOL	HIS	ARE	SOON
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON

Search and insertion operations are straightforward with closed hashing
However, deletion has to be carefully handled. For example, if we simply the delete the key 'ARE' from the hash table (Above), we will be unable to find the key 'SOON' afterward. Because, h(SOON) =11, the algorithm would find the location empty and report unsuccessful search. A simple solution is to use "Lazy Deletion," i.e., to mark previously occupied locations by a special symbol to distinguish them from locations that have not been occupied.