

# Data Structures

## Lecture 1: Introduction to Data Structures



By  
**Ravi Kant Sahu**

*Asst. Professor*

Lovely Professional University, Punjab



# Contents

- Basic Terminology
- Classification of Data Structures
- Data Structure Operations
- Review Questions



# Basic Terminology

- Data: are values or set of values.
- Data Item: is a single unit of values.
- Data Items are divided into two categories:
  - Group Items: Data items that are divided into sub-items.
  - Elementary Items: Data items that are not divided into sub-items.



# Data Structures

Arrangement of Data to simplify the processing



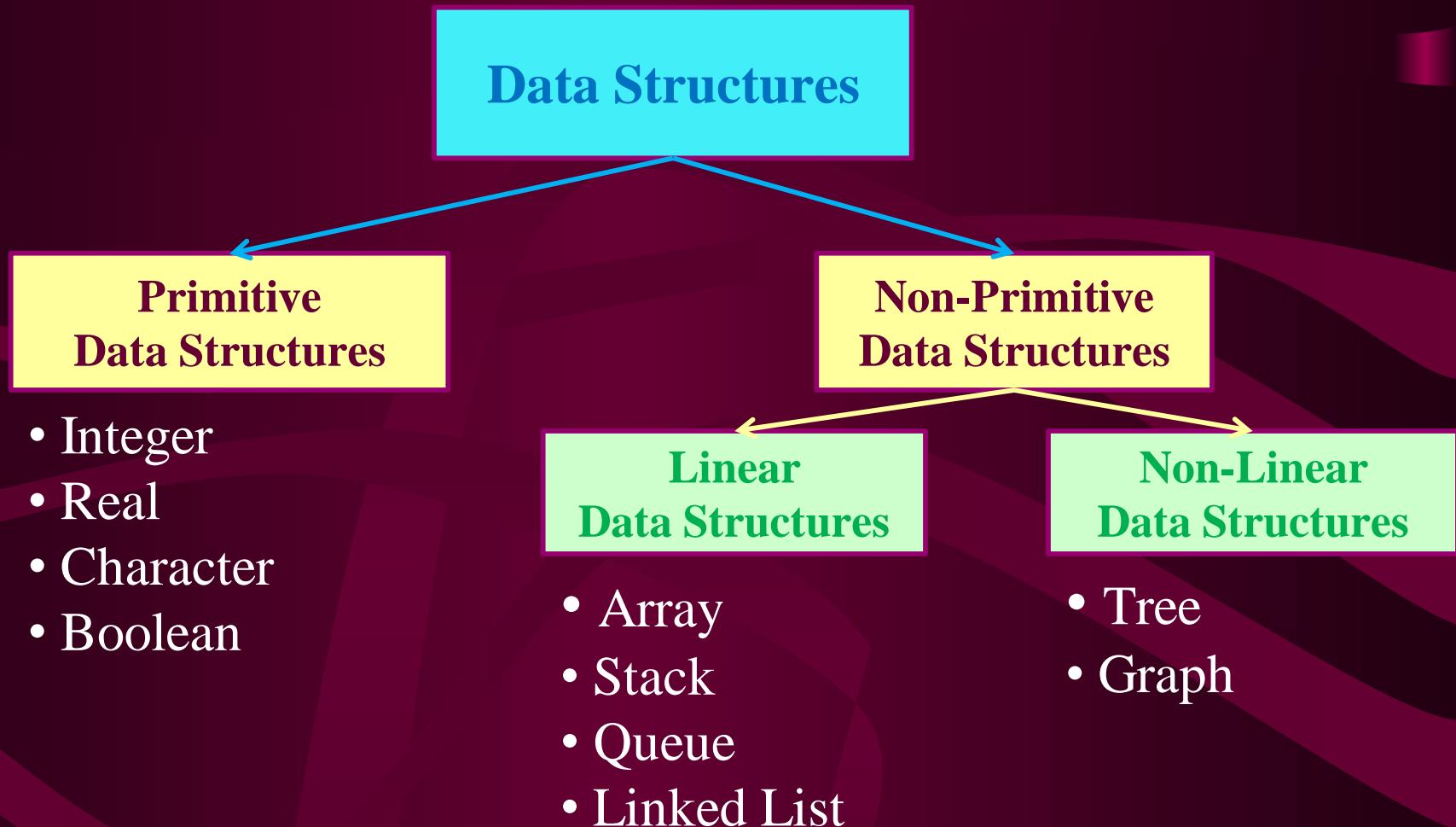
# Data Structure

- Organization of data needed to solve the problem.

“ Logical or mathematical model of a particular organization of data is called a Data Structure.”



# Classification of Data Structures



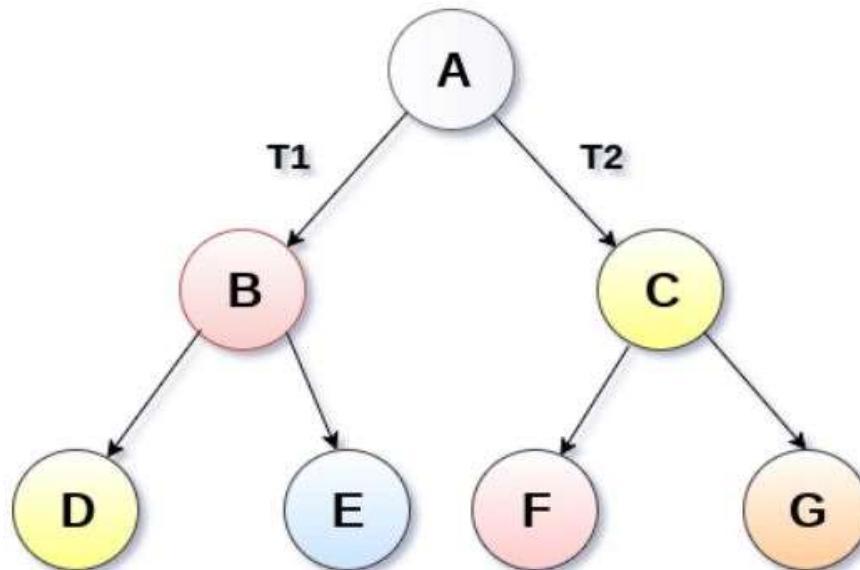
Array :

3	8	1	0	5	-2	32
---	---	---	---	---	----	----

Indices:

0    1    2    3    4    5    6

**Root Node**





# Data Structure Operations

Data Structures are processed by using certain operations.

1. Traversing: Accessing each record exactly once so that certain items in the record may be processed.
2. Searching: Finding the location of the record with a given key value, or finding the location of all the records that satisfy one or more conditions.
3. Inserting: Adding a new record to the structure.
4. Deleting: Removing a record from the structure.



# Special Data Structure-Operations

- **Sorting:** Arranging the records in some logical order (Alphabetical or numerical order).
- **Merging:** Combining the records in two different sorted files into a single sorted file.



Why so many Data Structures?



Questions



# Review Questions

- What is the difference between Data and Information?  
Explain with example.
- What is the difference between Linear and Non-Linear  
Data Structure?
- Differentiate between Sorting and Merging.
- How Searching is different from Traversing?



# Linear Vs Non-Linear Data Structure

- LDS is sequential in nature i.e. every data item is related to its previous and next data item only.
- In LDS, data can be traversed in a single run only.
- Implementation of Non-linear DS is difficult.
- Example: LDS- Array, Linked List, Queue, Stack etc.
- Non-linear DS- Tree and Graphs

# Data Structures

## Lecture 2-3: Asymptotic Notations & Complexity Analysis



By  
**Ravi Kant Sahu**

*Asst. Professor,*

Lovely Professional University, Punjab



# Contents

- Basic Terminology
- Complexity of Algorithm
- Asymptotic Notations
- Linear Search and Binary Search
- Review Questions



# Basic Terminology

- **Algorithm:** is a finite step by step list of well-defined instructions for solving a particular problem.
- **Complexity of Algorithm:** is a function which gives running time and/or space requirement in terms of the input size.
- **Time** and **Space** are two major measures of efficiency of an algorithm.



# Algorithm

Specification of Input  
(e.g. any sequence of  
natural numbers)

**Algorithm**

Specification of Output  
as a function of Input  
(e.g. sequence of sorted  
natural numbers)



# Characteristics of Good Algorithm

- Efficient
  - Running Time
  - Space used
- Efficiency as a function of input size
  - Size of Input (5, 5555555555)
  - Number of Data elements



# Time-Space Tradeoff

- By increasing the amount of space for storing the data, one may be able to reduce the time needed for processing the data, or vice versa.



# Complexity of Algorithm

- Time and Space used by the algorithm are two main measures for efficiency of any algorithm  $M$ .
- Time is measured by counting the number of key operations.
- Space is measured by counting the maximum of memory needed by the algorithm.



- Complexity of Algorithm is a function  $f(n)$  which gives running time and/or space requirement of algorithm  $M$  in terms of the size  $n$  of the input data.
- Worst Case: The maximum value of  $f(n)$  for any possible input.
- Average Case: The expected or average value of  $f(n)$ .
- Best Case: Minimum possible value of  $f(n)$ .



Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Rate of Growth

- The rate of growth of some standard functions  $g(n)$  is:

$$\log_2 n < n < n \log_2 n < n^2 < n^3 < 2^n$$

$n$	$g(n)$	$\log n$	$n$	$n \log n$	$n^2$	$n^3$	$2^n$
5	3	5	5	15	25	125	32
10	4	10	10	40	100	$10^3$	$10^3$
100	7	100	100	700	$10^4$	$10^6$	$10^{30}$
1000	10	$10^3$	$10^3$	$10^4$	$10^6$	$10^9$	$10^{300}$



# ASYMPTOTIC NOTATIONS

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Asymptotic Notations

- Goal: to simplify analysis of running time .
- Useful to identify how the running time of an algorithm increases with the size of the input in the limit.



# Asymptotic Notations

## Special Classes of Algorithms

- Logarithmic:  $O(\log n)$
- Linear:  $O(n)$
- Quadratic:  $O(n^2)$
- Polynomial:  $O(n^k)$ ,  $k \geq 1$
- Exponential:  $O(a^n)$ ,  $a > 1$



# Big-Oh ( $O$ ) Notation

- Asymptotic upper bound
- $f(n) = O(g(n))$ , if there exists constants  $c$  and  $n_0$  such that,  
$$f(n) \leq c g(n) \text{ for all } n \geq n_0$$
- $f(n)$  and  $g(n)$  are functions over non-negative integers.
- Used for Worst-case analysis.



# Big-Oh ( $O$ ) Notation

- Simple Rule:

Drop lower order terms and constant factors.

*Example:*

- $50n \log n$  is  $O(n \log n)$
- $8n^2 \log n + 5n^2 + n$  is  $O(n^2 \log n)$



# Big-Omega ( $\Omega$ ) Notation

- Asymptotic lower bound
- $f(n) = \Omega(g(n))$ , if there exists constants  $c$  and  $n_0$  such that,  
$$c g(n) \leq f(n) \text{ for all } n \geq n_0$$
- Used to describe Best-case running time.



# Big-Theta ( $\Theta$ ) Notation

- Asymptotic tight bound
- $f(n) = \Theta(g(n))$ , if there exists constants  $c_1, c_2$  and  $n_0$  such that,

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for } n \geq n_0$$

- $f(n) = \Theta(g(n))$ , iff  $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$



# Little-Oh ( $o$ ) Notation

- Non-tight analogue of Big-Oh.
- $f(n) = o(g(n))$ , if **for every**  $c$  there exists  $n_0$  such that,

$$f(n) < c g(n) \text{ for all } n \geq n_0$$

- $f(n) = o(g(n))$ , iff  
 $f(n) = O(g(n))$  and  $f(n) \neq \Omega(g(n))$
- Used for comparisons of running times.



# SEARCHING

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Searching

## 1. Linear Search:

- Compares the item of interest with each element of Array one by one.
- Traverses the Array sequentially to locate the desired item.



# Linear Search Algorithm

- **LINEAR (DATA, N, ITEM, LOC)**

1. [Insert ITEM at the end of DATA] Set Data [N]= ITEM.
2. [Initialize Counter] Set LOC=0.
3. [Search for ITEM.]  
Repeat while DATA [LOC] != ITEM  
    Set LOC = LOC +1.  
[End of loop.]
4. [Successful?] if LOC = N, then Print “ITEM not found”.  
    else return LOC.
5. Exit.



## 2. Binary Search

- **BINARY ( DATA, LB, UB, ITEM, LOC )**
    1. [Initialize Segment Variables]  
Set BEG = LB, END = UB and MID = INT ((BEG+END)/2).
    2. Repeat Steps 3 and 4 while BEG < END and DATA [MID] != ITEM.
      3. If ITEM < DATA[MID], then:  
Set END = MID - 1.  
Else:  
Set BEG = MID + 1.  
[End of if Structure.]
      4. Set MID = INT ((BEG+END)/2).  
[End of Step 2 Loop.]
      5. If DATA [MID] = ITEM, then: Print “Item Found at ” LOC  
Else:  
Set LOC = NULL.  
[End of if structure.]
    6. Exit.
- Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Limitations of Binary Search

- Although the complexity of Binary Search is  $O(\log n)$ , it has some limitations:
  1. the list must be sorted
  2. one must have direct access to the middle element in any sublist.



# Questions



# Review Questions

- When an algorithm is said to be better than the other?
- Can an algorithm have different running times on different machines?
- How the algorithm's running time is dependent on machines on which it is executed?



# Review Questions

Find out the complexity:

```
function ()  
{  
    if (condition)  
    {  
        for (i=0; i<n; i++) { // simple statements}  
    }  
    else  
    {  
        for (j=1; j<n; j++)  
            for (k=n; k>0; k--) { // simple statement}  
    }  
}
```



# Review Questions

Find out the complexity:

- void complex (int n) {  
    for (int i=1;i\*i<=n;i++)  
        for (int j=1;j\*j<=n;j++)  
            { cout<<" \* "; } }
- void more\_complex (int n)  
{ for(int i=1;i<=n/3;i++) {  
    for(int j=1;j<=n;j+=4) {  
        cout<<" \* ";   break; }  
    break;  
}

# **Complexity Analysis**

# Basic rules

1. Nested loops are multiplied together.
2. Sequential loops are added.
3. Only the largest term is kept, all others are dropped.
4. Constants are dropped.
5. Conditional checks are constant (i.e. 1).

# Example 1

- `for(int i = 0; i < n; i++) {`
- `cout << i << endl;`
- `}`

- Ans:  $O(n)$

# Example 2

- `for(int i = 0; i < n; i++) {`
- `for(int j = 0; j < n; j++){`
- `//do swap stuff, constant time`
- `}`
- `}`

- Ans  $O(n^2)$

# Example 3

- `for(int i = 0; i < n; i++) {`
- `for(int j = 0; j < i; j++){`
- `//do swap stuff, constant time`
- `}`
- `}`

- Ans: Outer loop is still 'n'. The inner loop now executes 'i' times, the end being  $(n-1)$ . We now have  $n(n-1)$ . This is still in the bound of  $O(n^2)$

# Example 4

- `for(int i = 0; i < 2*n; i++) {`
- `cout << i << endl;`
- `}`

- At first you might say that the upper bound is  $O(2n)$ ; however, we drop constants so it becomes  $O(n)$

# Example 5

- ```
for(int i = 0; i < n; i++) {
```
- ```
cout << i << endl;
```
- }
- 
- ```
for(int i = 0; i < n; i++) {
```
- ```
for(int j = 0; j < i; j++){
```
- ```
//do constant time stuff
```
- }
- }

- Ans : In this case we add each loop's Big O, in this case  $n+n^2$ .  $O(n^2+n)$  is not an acceptable answer since we must drop the lowest term. The upper bound is  $O(n^2)$ . Why? Because it has the largest growth rate

# Example 6

- `for(int i = 0; i < n; i++) {`
- `for(int j = 0; j < 2; j++){`
- `//do stuff`
- `}`
- `}`

- Ans: Outer loop is 'n', inner loop is 2, this we have  $2n$ , dropped constant gives up  $O(n)$

# Example 7

- `for(int i = 1; i < n; i *= 2) {`
- `cout << i << endl;`
- `}`

- There are n iterations, however, instead of simply incrementing, 'i' is increased by  $2^*i$  itself each run. Thus the loop is  $\log(n)$ .

# Example 8

- `for(int i = 0; i < n; i++) {`
- `for(int j = 1; j < n; j *= 2){`
- `//do constant time stuff`
- `}`
- `}`

- Ans:  $n \log(n)$

# Example 9

- `for(int i = 0; i < n; i *= 2) {`
- `cout << i << endl;`
- `}`

# Data Structures

## Lecture 5: Linear Array



By  
**Ravi Kant Sahu**

*Asst. Professor,*

**Lovely Professional University, Punjab**



# Contents

- Basic Terminology
- Linear Array
- Memory Representation of Linear Array
- Traversing Array
- Insertion and Deletion in Array
- Sorting (Bubble Sort)
- Searching (Linear Search and Binary Search)
- Review Questions



# Basic Terminology

- **Linear Data Structures:** A data structure is said to be linear if its elements form a *sequence* or a linear list.
- **Linear Array:** is a list of a finite number  $n$  of *homogeneous* data elements such that:
  - (a) the elements of the array are referenced by an index set consisting of *n consecutive numbers*.
  - (b) the elements of the array are stored respectively in successive memory locations.



# Key Terms

- Size / Length of Array
- Index of Array
- Upper bound and Lower bound of Array



# Memory Representation of Arrays

|    |   |    |    |     |
|----|---|----|----|-----|
| 19 | 5 | 42 | 18 | 199 |
|----|---|----|----|-----|



|      |
|------|
| 1001 |
| 1002 |
| 1003 |
| 1004 |
| 1005 |
| 1006 |
| 1007 |
| 1008 |
| 1009 |
| 1010 |
| 1011 |
| 1012 |



# Traversing Linear Array

- Suppose we have to count the number of element in an array or print all the elements of array.
- Algorithm 1: (Using While Loop)
  1. [Initialize Counter.] Set  $K = LB$ .
  2. Repeat Step 3 and 4 while  $K \leq UB$ .
    3. [Visit Element.] Apply PROCESS to  $A[K]$ .
    4. [Increase Counter.] Set  $K = K + 1$ .  
[End of Step 2 Loop.]
  5. Exit.



## Algorithm 2: (Using for loop)

1. Repeat for  $K = LB$  to  $UB$   
    Apply PROCESS to  $A[K]$ .  
    [ End of Loop.]
2. Exit.

**Question.1:** Find the Number of elements in an array which are greater than 25.

**Question 2:** Find out the sum of all the two digit numbers in an array.



# Insertion and Deletion in an Array

- Two types of insertion are possible:
  - Insertion at the end of array
  - Insertion in the middle of the array



# Insertion into a Linear Array

- Algorithm: (Insertion of an element ITEM into K<sup>th</sup> position in a Linear Array A)
  1. [Initialize Counter.] Set  $J = N$ .
  2. Repeat Steps 3and 4 while  $J >= K$ .
  3. [Move J<sup>th</sup> element downward] Set  $A[J+1] = A[J]$ .
  4. [Decrease Counter.] Set  $J = J-1$ .  
[End of Step 2 loop]
  5. [Insert element.] Set  $A[K] = ITEM$ .
  6. [Reset N]  $N = N+1$ .
  7. Exit



# Deletion into a Linear Array

Algorithm: (Delete K<sup>th</sup> element from Linear Array A)

1. Repeat for J = K to N-1.
2. [Move (J+1)<sup>th</sup> element upward] Set A[J] = A[J+1].  
[End of loop.]
3. [Reset the number of elements N] Set N = N-1.
4. Exit



# Searching

## 1. Linear Search:

- Compares the item of interest with each element of Array one by one.
- Traverses the Array sequentially to locate the desired item.



# Linear Search Algorithm

- **LINEAR\_SEARCH (DATA, N, ITEM, LOC)**
1. [Insert ITEM at the end of DATA] Set Data [N+1]= ITEM.
  2. [Initialize Counter] Set LOC=1.
  3. [Search for ITEM.]  
Repeat while DATA [LOC] != ITEM  
    Set LOC = LOC +1.  
[End of loop.]
  4. [Successful?] if LOC = N + 1, then Set LOC = 0.
  5. Exit.



## 2. Binary Search

- **BINARY ( DATA, LB, UB, ITEM, LOC )**
  1. [Initialize Segment Variables]  
Set BEG = LB, END = UB and MID = INT ((BEG+END)/2).
  2. Repeat Steps 3 and 4 while BEG <= END and DATA [MID] != ITEM.
  3.     If ITEM < DATA[MID], then:  
           Set END = MID - 1.  
       Else:  
           Set BEG = MID + 1.  
           [End of if Structure.]
  4.     Set MID = INT ((BEG+END)/2).  
       [End of Step 2 Loop.]
  5. If DATA [MID] = ITEM, then: Set LOC= MID  
       Else:  
           Set LOC = NULL.  
           [End of if structure.]
  6. Exit.     Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Limitations of Binary Search

- Although the complexity of Binary Search is  $O(\log n)$ , it has some limitations:
  1. the list must be sorted
  2. one must have direct access to the middle element in any sublist.



# Sorting (Bubble Sort)

- Sorting refers to the operation of rearranging the elements of A so they are in some particular order.
- Complexity of Bubble Sort Algorithm is:  $O(n^2)$
- Example of Bubble Sort



# Bubble Sort Algorithm

**Bubble (DATA, N)**

1. Repeat Step 2 and 3 for K=1 to N-1.
2. [Initialize Pass Pointer P] Set P=1.
3. [Execute Pass] Repeat while P <= N-K.
  - (a) if DATA [P] > DATA [P+1], then:  
            Interchange DATA [P] and DATA[P+1]  
            [End of if Structure.]
  - (b) Set P = P+1.  
            [End of Inner Loop.]

[End of Step1 Outer Loop.]
4. Exit



Questions

# Data Structures

## Lecture 6: Linear Array



By

**Ravi Kant Sahu**

*Asst. Professor,*

**Lovely Professional University, Punjab**



# Contents

- Basic Terminology
- Linear Array
- Memory Representation of Linear Array
- Traversing Array



# Basic Terminology

- **Linear Data Structures:** A data structure is said to be linear if its elements form a *sequence* or a linear list.
- **Linear Array:** is a list of a finite number  $n$  of *homogeneous* data elements such that:
  - (a) the elements of the array are referenced by an index set consisting of *n consecutive numbers*.
  - (b) the elements of the array are stored respectively in successive memory locations.



# Key Terms

- Size / Length of Array
- Index of Array
- Upper bound and Lower bound of Array



# Memory Representation of Arrays

|    |   |    |    |     |
|----|---|----|----|-----|
| 19 | 5 | 42 | 18 | 199 |
|----|---|----|----|-----|



|      |
|------|
| 1001 |
| 1002 |
| 1003 |
| 1004 |
| 1005 |
| 1006 |
| 1007 |
| 1008 |
| 1009 |
| 1010 |
| 1011 |
| 1012 |



# Traversing Linear Array

- Suppose we have to count the number of element in an array or print all the elements of array.
- Algorithm 1: (Using While Loop)
  1. [Initialize Counter.] Set  $K = LB$ .
  2. Repeat Step 3 and 4 while  $K \leq UB$ .
    3. [Visit Element.] Apply PROCESS to  $A[K]$ .
    4. [Increase Counter.] Set  $K = K + 1$ .  
[End of Step 2 Loop.]
  5. Exit.



# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



## Algorithm 2: (Using for loop)

1. Repeat for  $K = LB$  to  $UB$   
    Apply PROCESS to  $A[K]$ .  
    [ End of Loop.]
2. Exit.

**Question.1:** Find the Number of elements in an array which are greater than 25.

**Question 2:** Find out the sum of all the two digit numbers in an array.



# Insertion and Deletion in an Array

- Two types of insertion are possible:
  - Insertion at the end of array
  - Insertion in the middle of the array



# Work Space

.

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Insertion into a Linear Array

- Algorithm: (Insertion of an element ITEM into K<sup>th</sup> position in a Linear Array A)
  1. [Initialize Counter.] Set  $J = N$ .
  2. Repeat Steps 3and 4 while  $J >= K$ .
  3. [Move J<sup>th</sup> element downward] Set  $A[J+1] = A[J]$ .
  4. [Decrease Counter.] Set  $J = J-1$ .  
[End of Step 2 loop]
  5. [Insert element.] Set  $A[K] = ITEM$ .
  6. [Reset N]  $N = N+1$ .
  7. Exit



# Work Space

.

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Deletion into a Linear Array

Algorithm: (Delete K<sup>th</sup> element from Linear Array A)

1. Repeat for J = K to N-1.
2. [Move (J+1)<sup>th</sup> element upward] Set A[J] = A[J+1].  
[End of loop.]
3. [Reset the number of elements N] Set N = N-1.
4. Exit



# Questions

# Data Structures

## Lecture 6 & 7: Multi-Dimensional Array

By

Ravi Kant Sahu  
Asst. Professor



**Lovely Professional University, Punjab**



# Outlines

- Introduction
- Two-Dimensional Arrays
- Memory Representation of Two-Dimensional Arrays
- Multidimensional Array
- Review Questions



# Introduction

- Arrays where elements are referenced, respectively, by two or more subscripts.
- Some programming languages allow up to 7 dimensional arrays.
- Normally we have Two-Dimensional and Three-Dimensional Arrays.



# Two-Dimensional Array

- A two-dimensional  $m \times n$  array  $A$  is a collection of  $m \times n$  data elements such that each element is specified by a pair of integers ( e.g.  $j, k$ ), called subscripts, with the property that

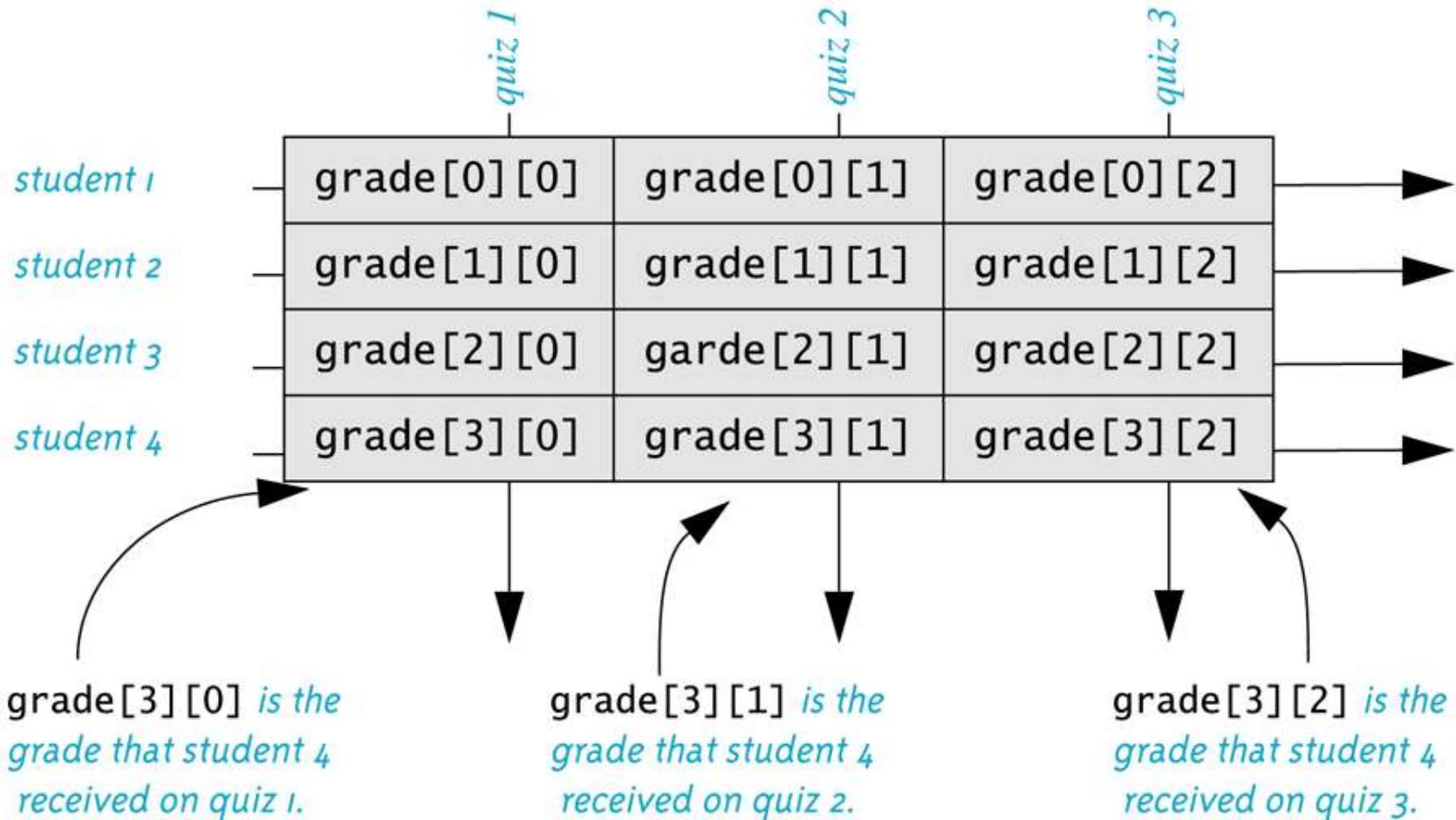
$$1 \leq j \leq m$$

*and*       $1 \leq k \leq n$

- The element of  $A$  with first subscript  $j$  and second subscript  $k$  will be denoted by  $A_{j,k}$  or  $A[j, k]$ .
- Two-dimensional arrays are called ***Matrices*** in mathematics and ***Tables*** in business applications.
- Two-dimensional arrays are sometimes known as ***Matrix Arrays***.



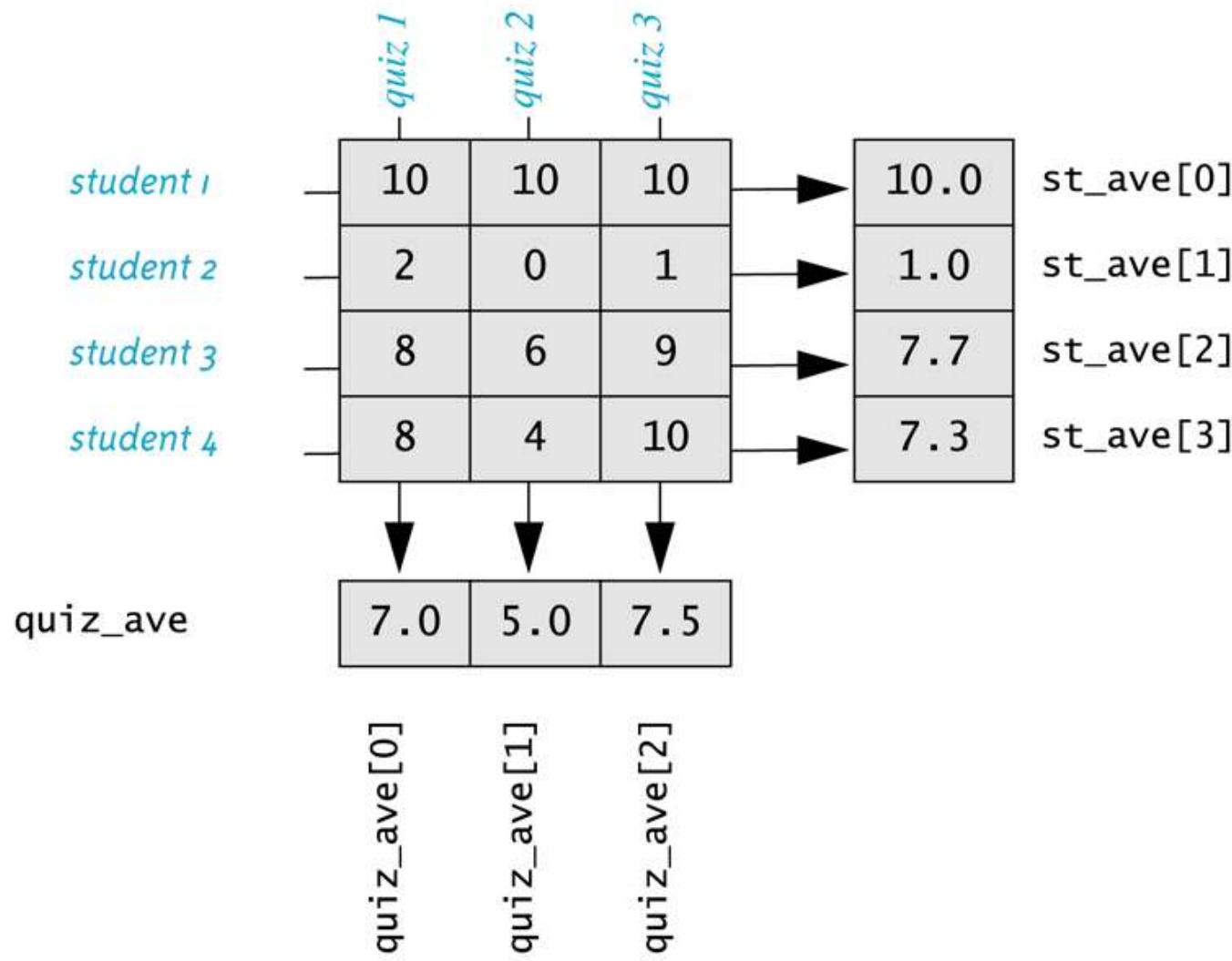
## The Two-Dimensional Array grade





## The Two-Dimensional Array grade (Another View)

---





# Memory Representation

- A Two-Dimensional array will be represented in memory by a block of  $m*n$  sequential memory locations.
- Two-Dimensional array is stored in the memory in following two orders:
  1. **Column-major Order:** Column by column.
  2. **Row-major Order:** Row by row.



# Column-Major Order

|        |  |  |                 |
|--------|--|--|-----------------|
| (1, 1) |  |  |                 |
| (2, 1) |  |  | <b>Column 1</b> |
| (3, 1) |  |  |                 |
| (1, 2) |  |  |                 |
| (2, 2) |  |  | <b>Column 2</b> |
| (3, 2) |  |  |                 |
| (1, 3) |  |  |                 |
| (2, 3) |  |  | <b>Column 3</b> |
| (3, 3) |  |  |                 |
| (1, 4) |  |  |                 |
| (2, 4) |  |  | <b>Column 4</b> |
| (3, 4) |  |  |                 |



# Row-Major Order

|        |       |
|--------|-------|
| (1, 1) | Row 1 |
| (1, 2) |       |
| (1, 3) |       |
| (1, 4) |       |
| (2, 1) | Row 2 |
| (2, 2) |       |
| (2, 3) |       |
| (2, 4) |       |
| (3, 1) | Row 3 |
| (3, 2) |       |
| (3, 3) |       |
| (3, 4) |       |



**Column-Major Order:**

$$\text{LOC } (A[j, k]) = \text{Base } (A) + w [M (k-1) + (j-1)]$$

**Row-Major Order:**

$$\text{LOC } (A[j, k]) = \text{Base } (A) + w [N (j-1) + (k-1)]$$



# General Multidimensional Array

- An n-dimensional  $m_1 \times m_2 \times \dots \times m_n$  array is a collection of  $m_1.m_2....m_n$  data elements in which each element is specified by a list of n integers (such as  $k_1, k_2, k_3...k_n$ ) called subscripts, with the property that:

$$1 \leq k_1 \leq m_1, 1 \leq k_2 \leq m_2, \dots, 1 \leq k_n \leq m_n$$





# Review Questions

- Given, in a 2-D Array the lower bound and upper bound for first index is 3 and 11 and that for second index is 5 and 9. Find out the size of the array.
- In any 2-D Array, which are the elements that will be always having the same memory address for both column-major order & Row-major order?
- Consider a  $20 \times 5$  matrix array Marks. Suppose Base (Marks) = 1002 and words per memory cell w=4. Using Column-major order and row-major order, find out the marks of 3<sup>rd</sup> and 5<sup>th</sup> test of student 11.

# Data Structures

## Lecture 7: Searching in Array



By

**Ravi Kant Sahu**

*Asst. Professor,*

Lovely Professional University, Punjab



# Contents

- Sorting (Bubble Sort)
- Searching (Linear Search and Binary Search)
- Review Questions



# Searching

## 1. Linear Search:

- Compares the item of interest with each element of Array one by one.
- Traverses the Array sequentially to locate the desired item.



# Linear Search Algorithm

- **LINEAR\_SEARCH (DATA, N, ITEM, LOC)**
1. [Insert ITEM at the end of DATA] Set Data [N+1]= ITEM.
  2. [Initialize Counter] Set LOC=1.
  3. [Search for ITEM.]  
Repeat while DATA [LOC] != ITEM  
    Set LOC = LOC +1.  
[End of loop.]
  4. [Successful?] if LOC = N + 1, then Set LOC = 0.
  5. Exit.



# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



## 2. Binary Search

- **BINARY ( DATA, LB, UB, ITEM, LOC )**
  1. [Initialize Segment Variables]  
Set BEG = LB, END = UB and MID = INT ((BEG+END)/2).
  2. Repeat Steps 3 and 4 while BEG <= END and DATA [MID] != ITEM.
  3.     If ITEM < DATA[MID], then:  
           Set END = MID - 1.  
       Else:  
           Set BEG = MID + 1.  
           [End of if Structure.]
  4.     Set MID = INT ((BEG+END)/2).  
       [End of Step 2 Loop.]
  5. If DATA [MID] = ITEM, then: Set LOC= MID  
       Else:  
           Set LOC = NULL.  
           [End of if structure.]
  6. Exit.     Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Limitations of Binary Search

- Although the complexity of Binary Search is  $O(\log n)$ , it has some limitations:
  1. the list must be sorted
  2. one must have direct access to the middle element in any sublist.



# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Sorting (Bubble Sort)

- Sorting refers to the operation of rearranging the elements of A so they are in some particular order.
- Complexity of Bubble Sort Algorithm is:  $O(n^2)$
- Example of Bubble Sort



# Bubble Sort Algorithm

**Bubble (DATA, N)**

1. Repeat Step 2 and 3 for K=1 to N-1.
2. [Initialize Pass Pointer P] Set P=1.
3. [Execute Pass] Repeat while P <= N-K.
  - (a) if DATA [P] > DATA [P+1], then:  
            Interchange DATA [P] and DATA[P+1]  
            [End of if Structure.]
  - (b) Set P = P+1.  
            [End of Inner Loop.]

[End of Step1 Outer Loop.]
4. Exit



# Work Space

.

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Questions

# Data Structures & Algorithms

## Insertion Sort

By

Ravi Kant Sahu  
Asst. Professor



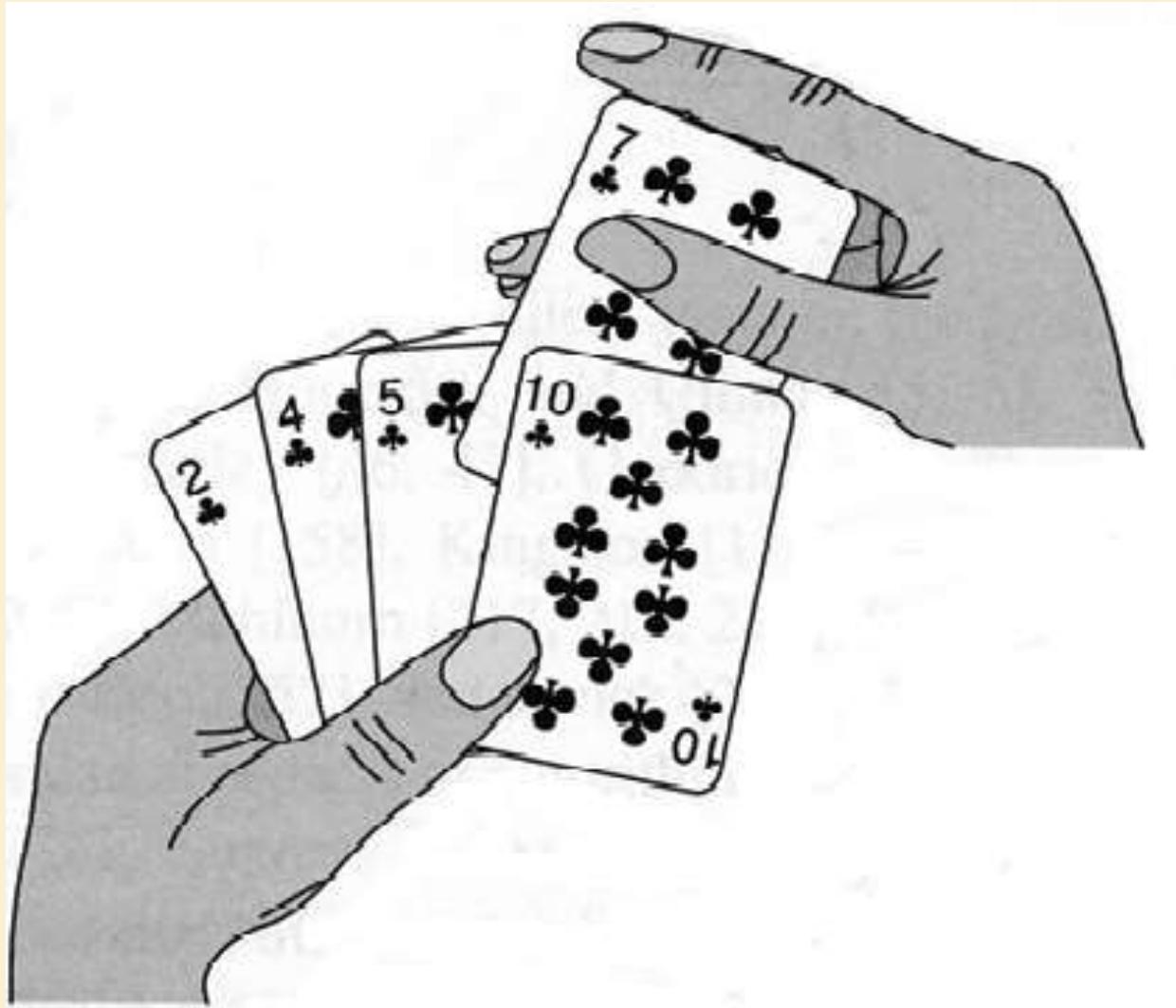
Lovely Professional University, Punjab



# INSERTION SORT



# INSERTION SORT



Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Insertion Sort

## INSERTION\_SORT (A, N)

1. Repeat Step 2 to 4 FOR J = 2 to N
2. Set Key = A[J] and I = J – 1.
3. Repeat Steps 4 and 5 **WHILE I > 0 & Key < A[I]**
4.       Set A[I+1] = A[I]
5.       Set I = I – 1.  
          [End of step 3 Loop.]
6.       Set A[I+1] = Key.  
          [End of Step 1 loop]
7. Return



# Work Space

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Insertion Sort Complexity

- This Sorting algorithm is frequently used when  $n$  is very small.
- Worst case occurs when array is in reverse order. The inner loop must use  $K - 1$  comparisons.

$$\begin{aligned}f(n) &= 1 + 2 + 3 + \dots + (n - 1) = n(n - 1)/2 \\&= O(n^2)\end{aligned}$$

- In average case, there will be approximately  $(K - 1)/2$  comparisons in the inner loop.

$$\begin{aligned}f(n) &= 1 + 2 + 3 + \dots + (n - 1)/2 = n(n - 1)/4 \\&= O(n^2)\end{aligned}$$



# Exercise

Apply Insertion Sort to sort the following array in descending order:

**7, 3, 9, 4, 6, 2, 4**



Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)

# Data Structures

Topic: Selection Sort

By

Ravi Kant Sahu  
Asst. Professor



Lovely Professional University, Punjab

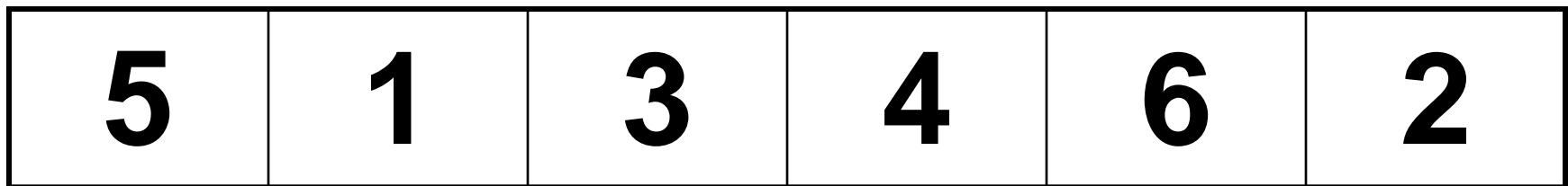
# Selection Sort

## Selection\_Sort (A, n)

1. Set  $j = 1$
2. Repeat Step 2 to 4 While  $j < n$  :
  3. Set  $\text{Min} = j$  and  $i = j+1$ .
  4. Repeat step 5 while  $i \leq n$ :
    5. IF:  $a[i] < a[\text{Min}]$   
THEN: Set  $\text{Min} = i$ .  
[End of step 4 loop.]
    6. IF:  $\text{Min} \neq j$   
THEN: swap ( $a[j]$ ,  $a[\text{Min}]$ ).  
[End of step 2 loop.]
  7. Return.

# Work Space

# Selection Sort



Comparison

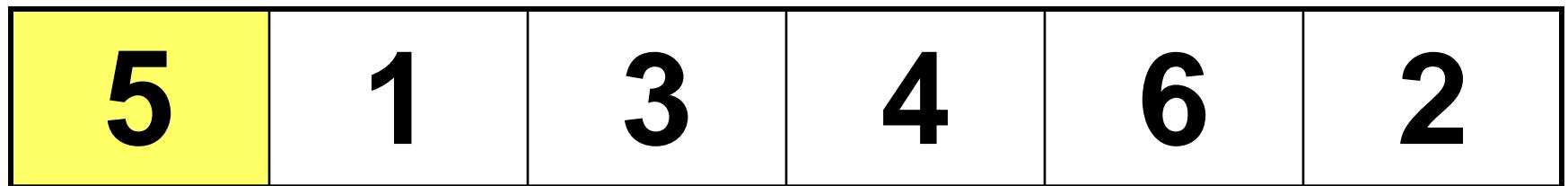


Data Movement



Sorted

# Selection Sort



Comparison

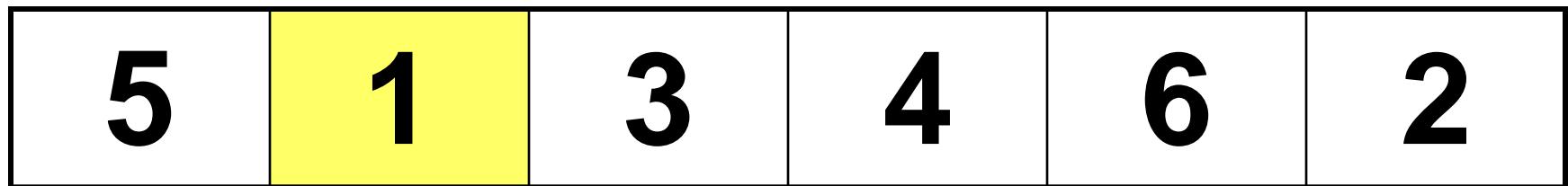


Data Movement



Sorted

# Selection Sort



Comparison

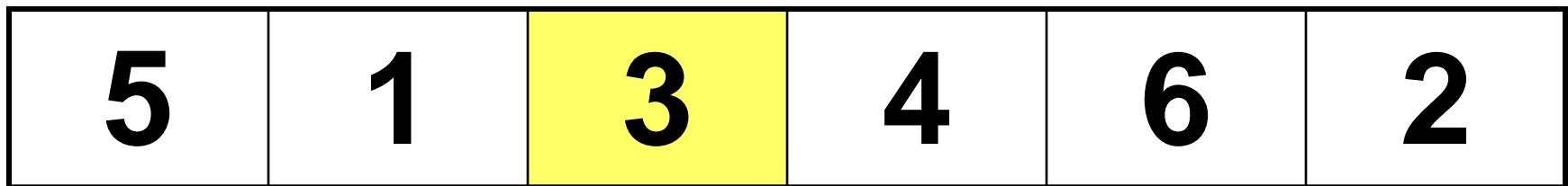


Data Movement



Sorted

# Selection Sort



Comparison

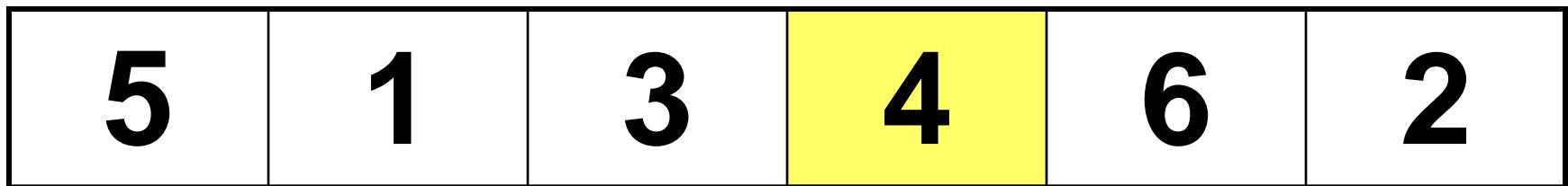


Data Movement



Sorted

# Selection Sort



Comparison

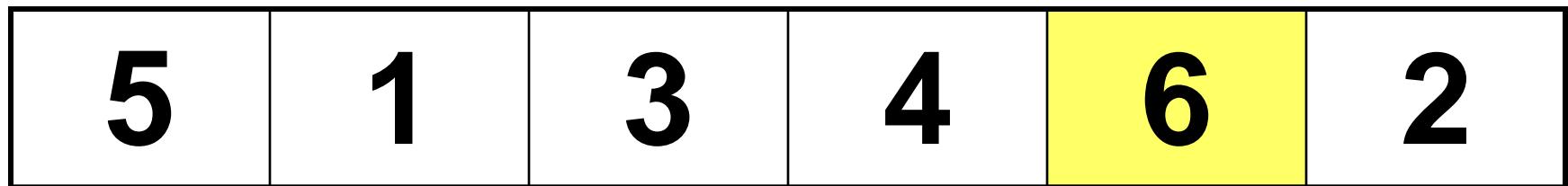


Data Movement



Sorted

# Selection Sort



Comparison



Data Movement



Sorted

# Selection Sort



Comparison

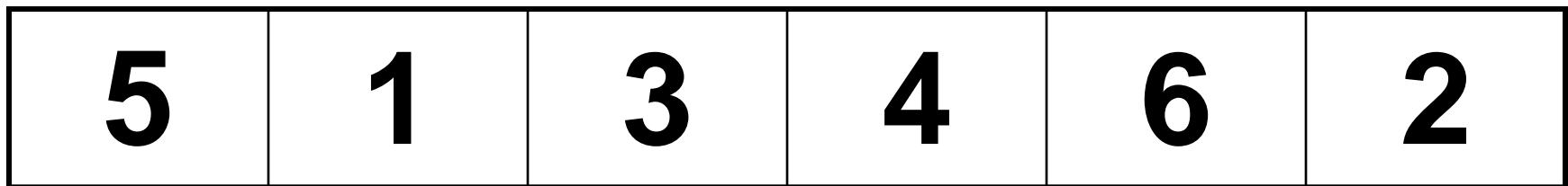


Data Movement



Sorted

# Selection Sort



↑  
Min



Comparison

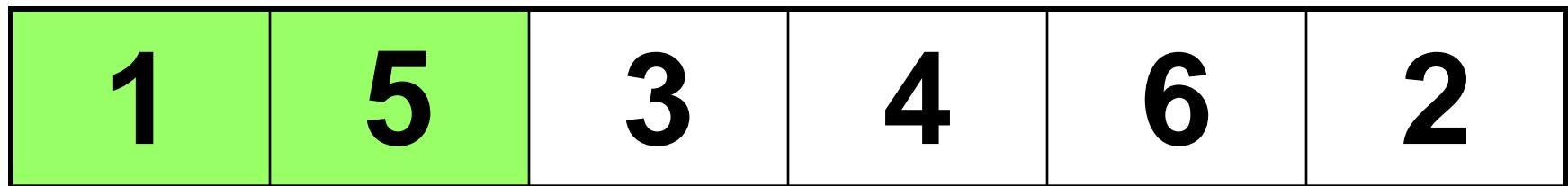


Data Movement



Sorted

# Selection Sort



Comparison



Data Movement



Sorted

# Selection Sort



Comparison

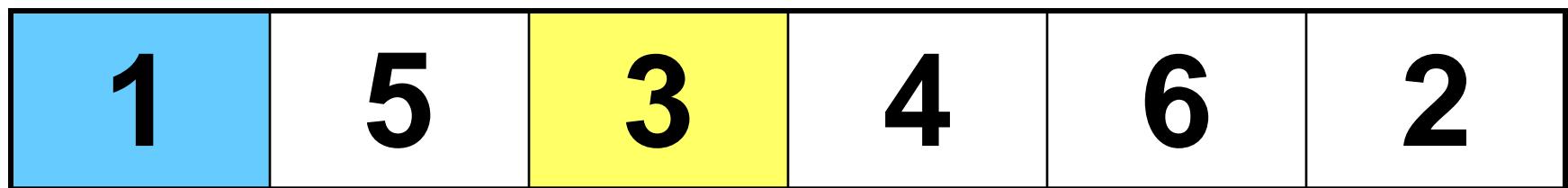


Data Movement



Sorted

# Selection Sort



Comparison

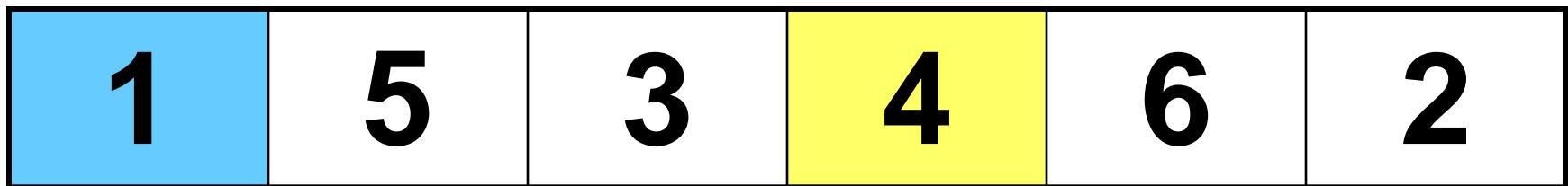


Data Movement



Sorted

# Selection Sort



Comparison

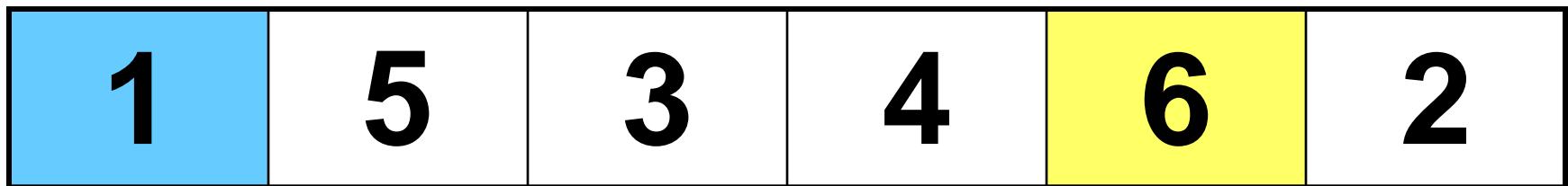


Data Movement



Sorted

# Selection Sort



Comparison



Data Movement



Sorted

# Selection Sort



Comparison



Data Movement



Sorted

# Selection Sort



Comparison

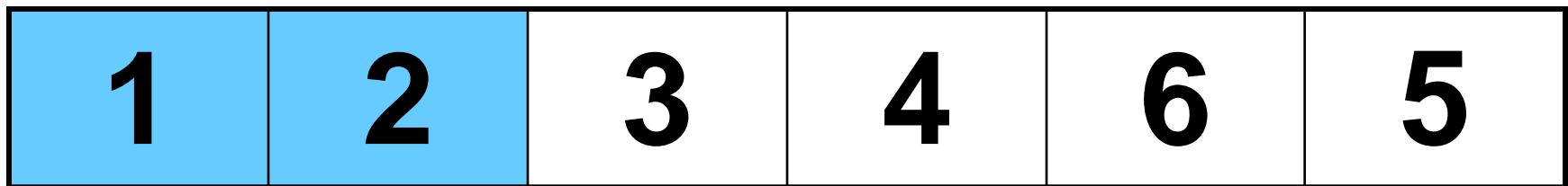


Data Movement



Sorted

# Selection Sort



Comparison

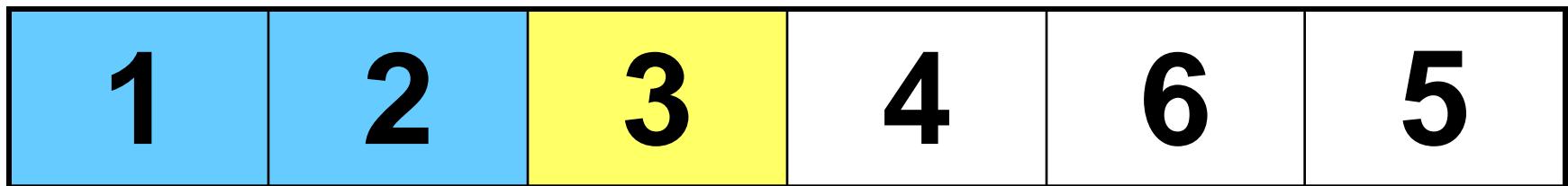


Data Movement



Sorted

# Selection Sort



Comparison

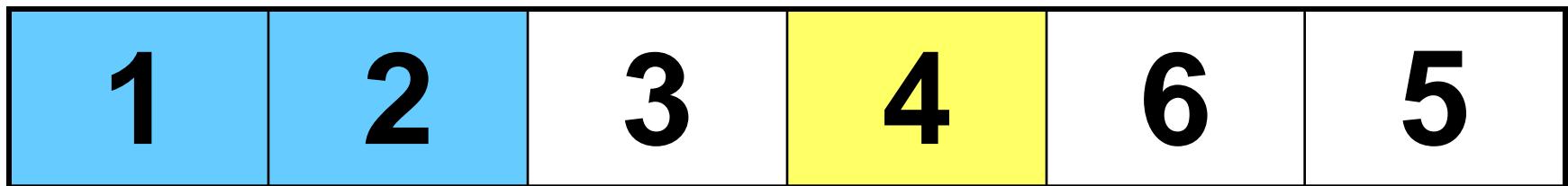


Data Movement



Sorted

# Selection Sort



Comparison

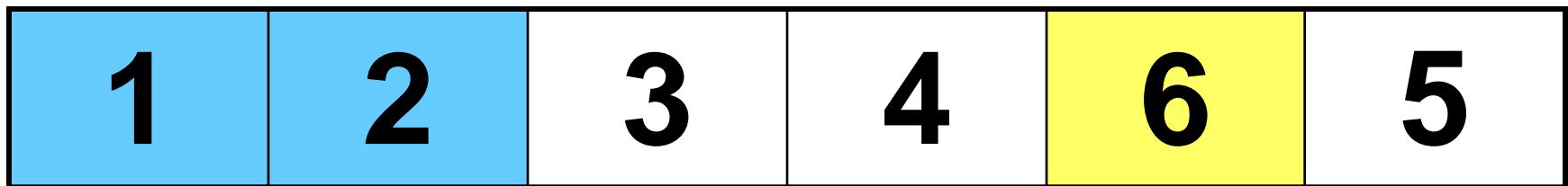


Data Movement



Sorted

# Selection Sort



Comparison

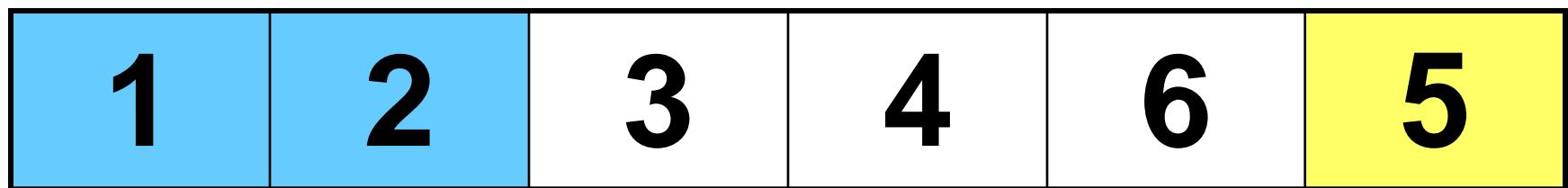


Data Movement



Sorted

# Selection Sort



Comparison

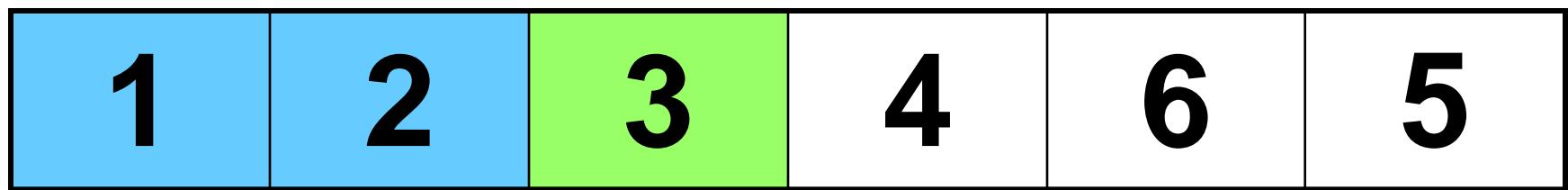


Data Movement



Sorted

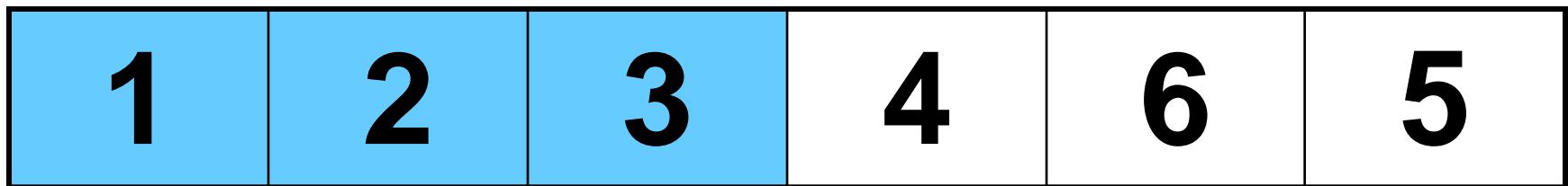
# Selection Sort



↑  
Min

- █ Comparison
- █ Data Movement
- █ Sorted

# Selection Sort



Comparison

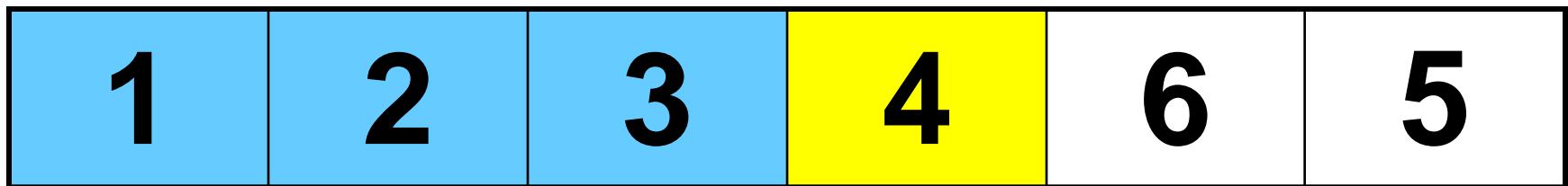


Data Movement



Sorted

# Selection Sort



Comparison

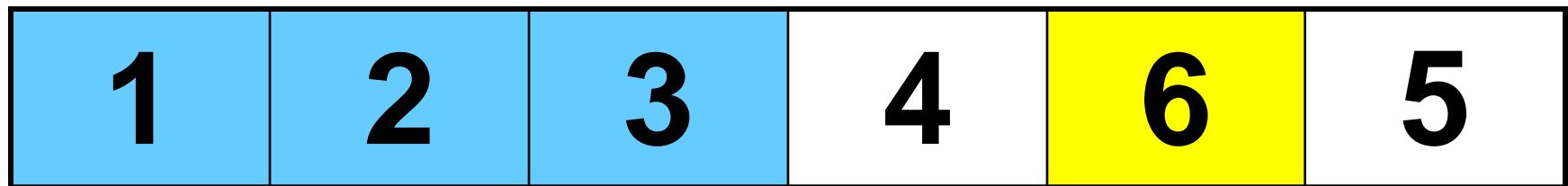


Data Movement



Sorted

# Selection Sort



Comparison

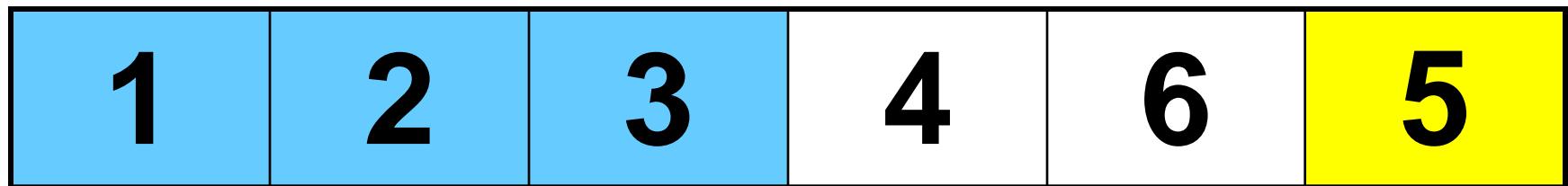


Data Movement



Sorted

# Selection Sort



↑  
Min



Comparison

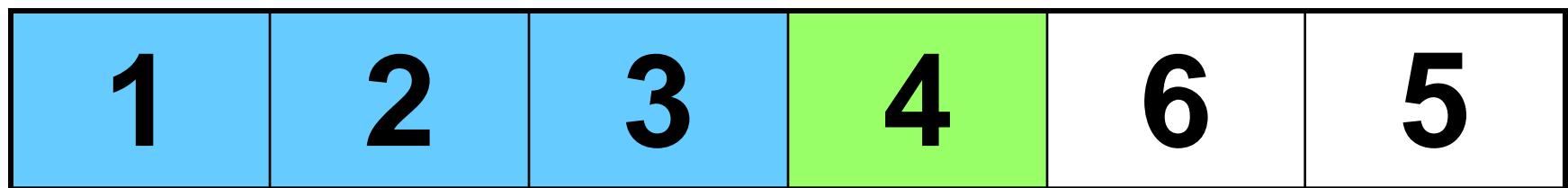


Data Movement



Sorted

# Selection Sort



Comparison



Data Movement



Sorted

# Selection Sort



Comparison

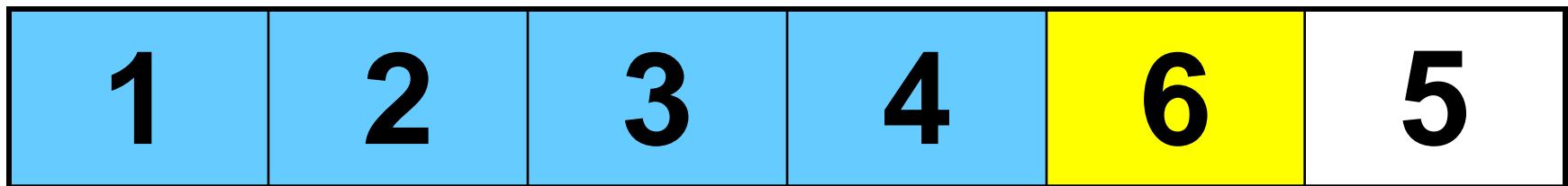


Data Movement



Sorted

# Selection Sort



Comparison



Data Movement



Sorted

# Selection Sort



Comparison



Data Movement



Sorted

# Selection Sort



Comparison

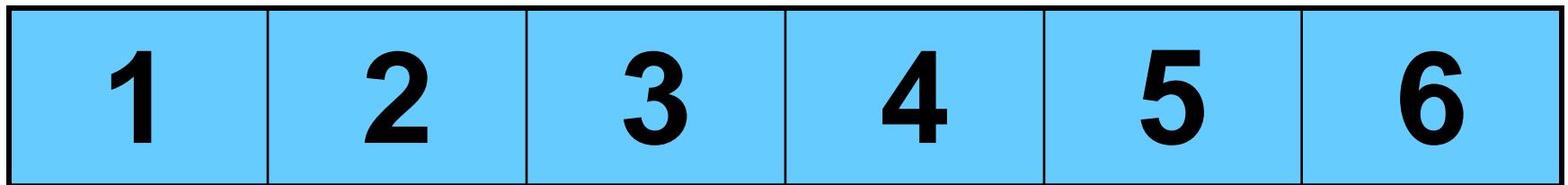


Data Movement



Sorted

# Selection Sort



**DONE!**



Comparison



Data Movement



Sorted

# Data Structures

## Lecture 8: Linked List

By

Ravi Kant Sahu

Asst. Professor



**Lovely Professional University, Punjab**



# Outlines

- Introduction
- Why Linked List?
- Types of Linked List
- Memory Representation of Linked Lists
- Difference between Singly Linked List and Arrays
- Review Questions



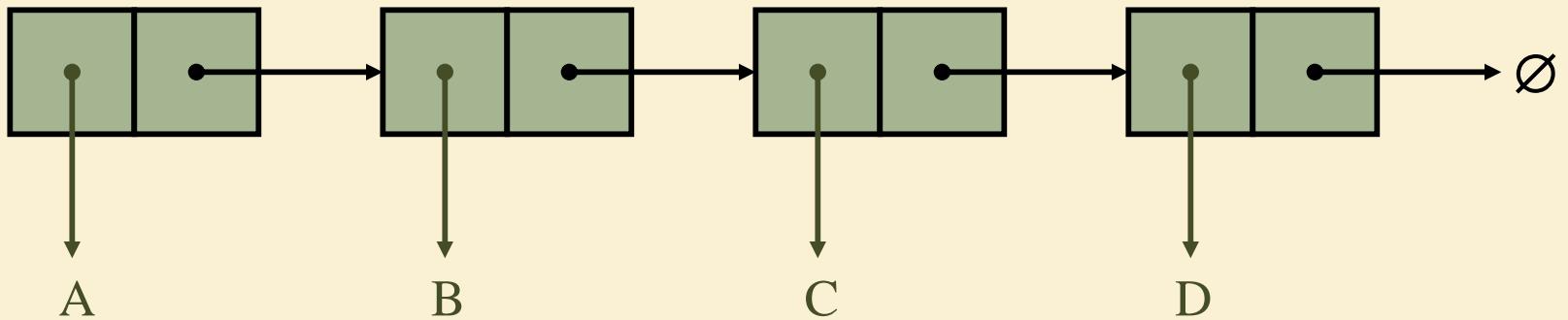
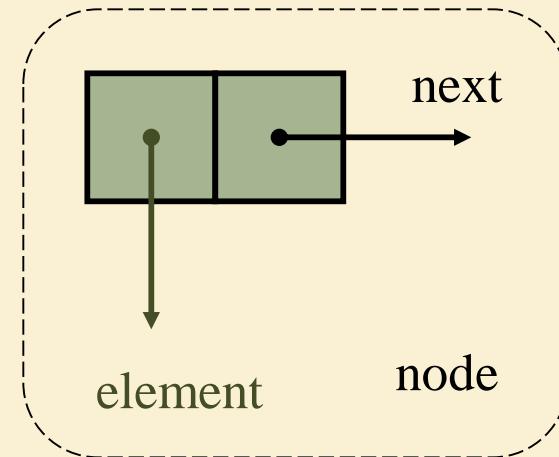
# Introduction

- A linked list (One-way list) is a linear collection of data elements, called nodes, where the linear order is given by means of pointers.
- Each node is divided into two parts.
- First part contains the information of the element.
- Second part contains the address of the next node in the list.



# Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node





# Key Points

## □ Linked list

- Linear collection of nodes
- Connected by pointer links
- Accessed via a pointer to the first node of the list
- Link pointer in the last node is set to null to mark the list's end
- Linked list contains a *List Pointer Variable* called START or NAME, which contains the address of the first node.



# Work Space

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Why Linked List?

## Arrays: pluses and minuses

- + Fast element access.
- Impossible to resize.

## Need of Linked List

- Many applications require resizing!
- Required size not always immediately available.

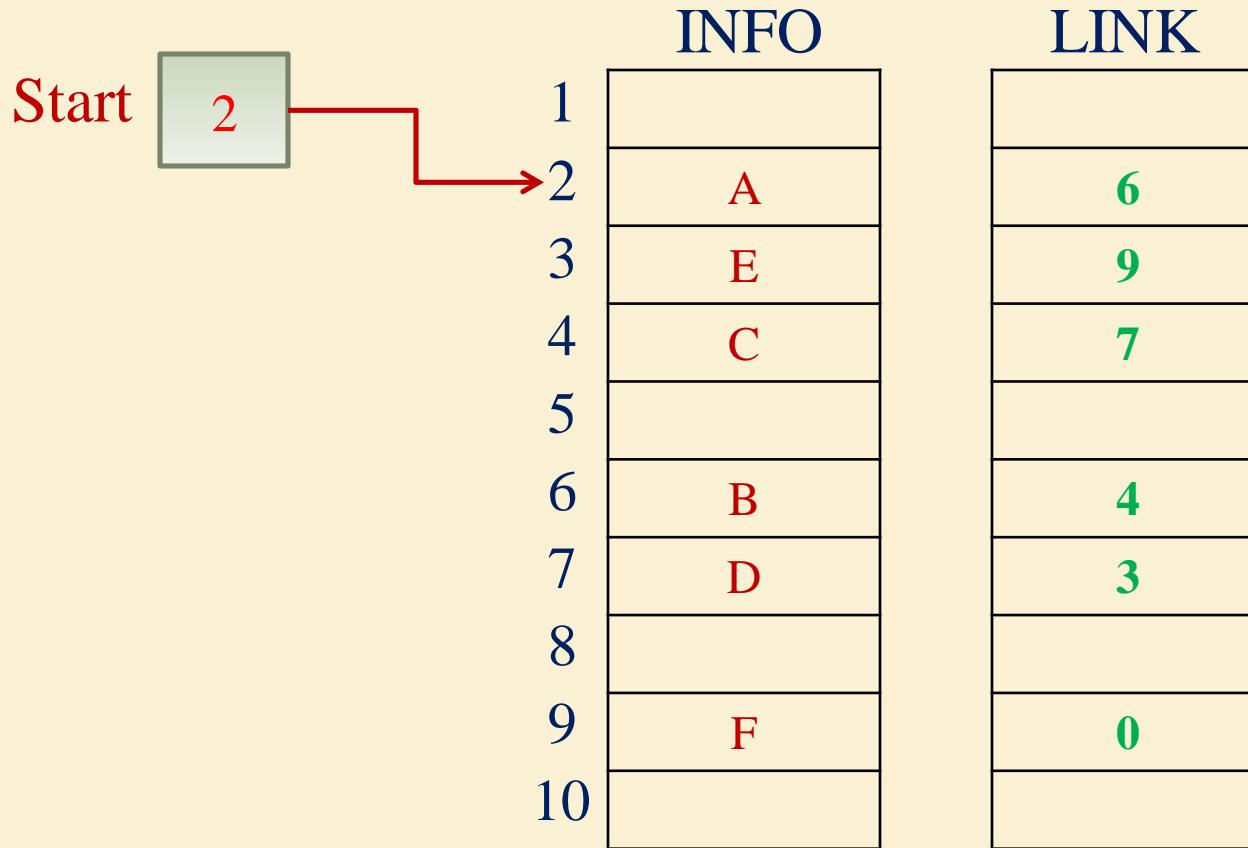


- Use a linked list instead of an array when
  - You have an unpredictable number of data elements
  - You want to insert and delete quickly.



# Memory Representation

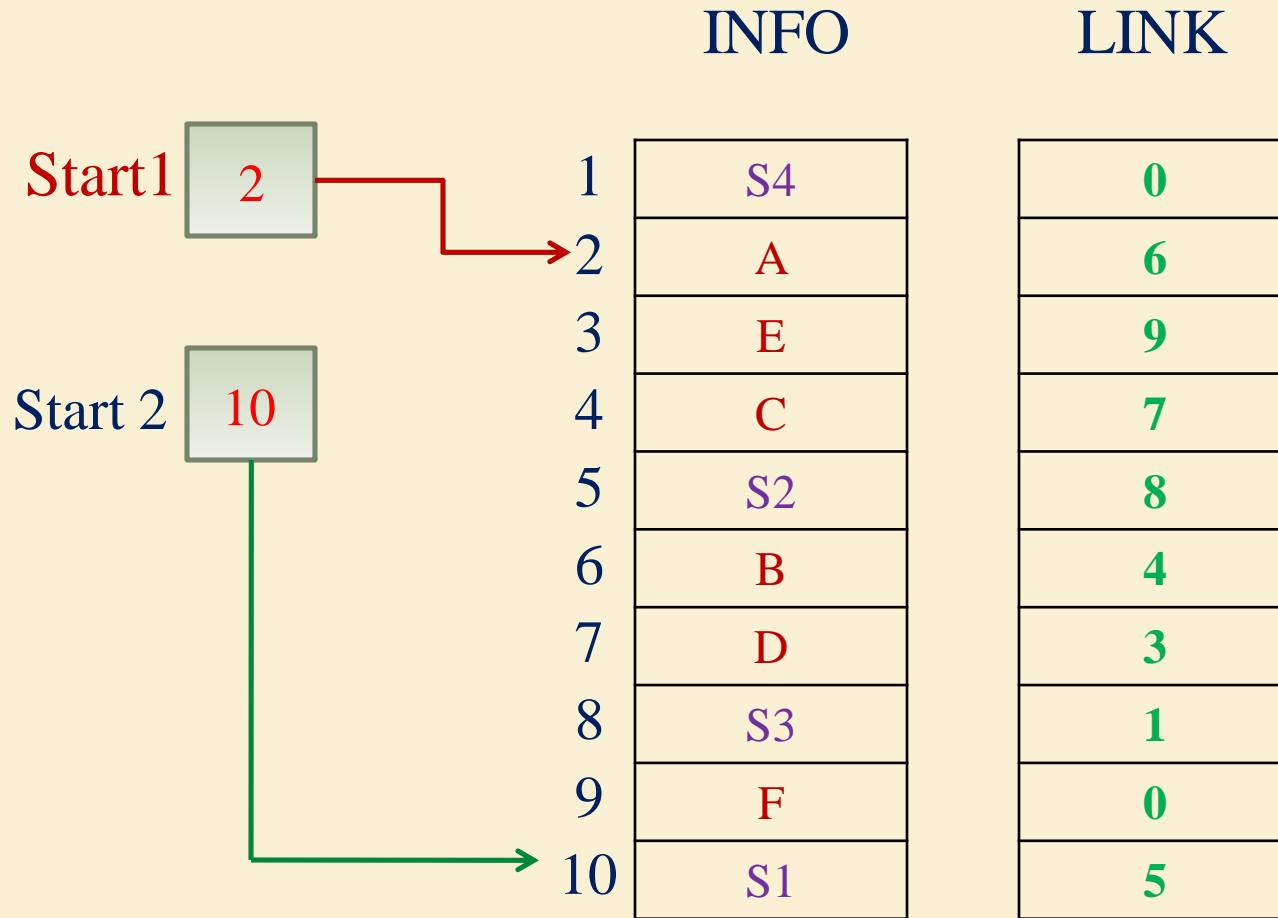
- Linked lists are maintained in memory using linear arrays or Parallel arrays.





# Memory Representation (2)

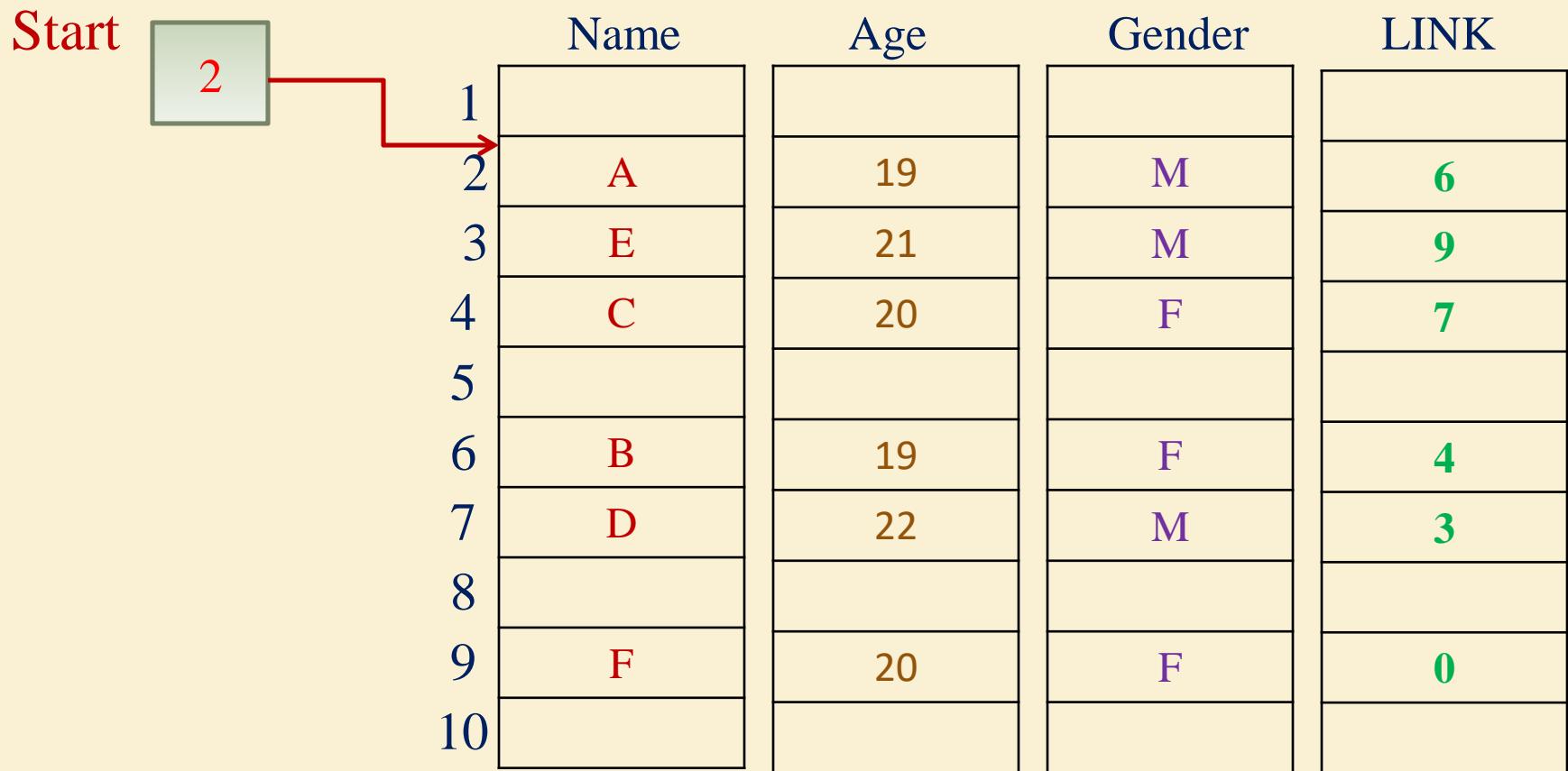
- Multiple lists in memory





# Memory Representation (3)

- INFO part of a node may be a record with multiple data items.
- Records in memory using Linked lists

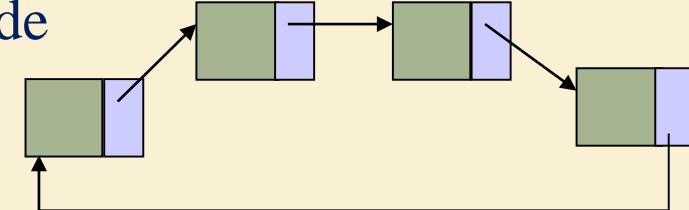




# Types of Linked Lists

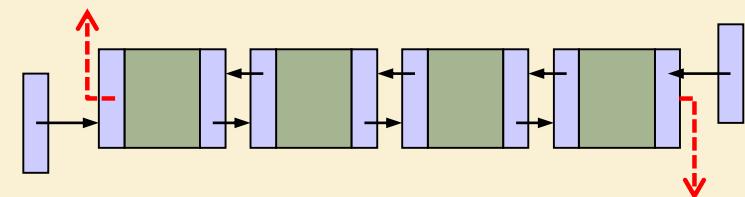
## 1. Singly linked list

- Begins with a pointer to the first node
- Terminates with a null pointer
- Only traversed in one direction



## 2. Circular, singly linked

- Pointer in the last node points back to the first node



## 3. Doubly linked list

- Two “start pointers” – first element and last element
- Each node has a forward pointer and a backward pointer
- Allows traversals both forwards and backwards

## 4. Circular, doubly linked list

- Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node



# Difference between Singly Linked List and Arrays

| Singly linked list                                                                                                                                                                                                                   | Array                                                                                                                                                                                                                    |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Elements are stored in linear order, accessible with links.</li><li>• Do not have a fixed size.</li><li>• Cannot access the previous element directly.</li><li>• No binary search.</li></ul> | <ul style="list-style-type: none"><li>• Elements are stored in linear order, accessible with an index.</li><li>• Have a fixed size.</li><li>• Can access the previous element easily.</li><li>• Binary search.</li></ul> |



Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Review Questions

- Why we need Linked List?
- What are the two different Fields in a node of linked list.
- How Linked lists are different from Arrays?
- Why Binary search is not possible in Linked list?
- What is the difference between Circular Linked List and Doubly Linked List?

# Data Structures

## Lecture 5: Linked List Operations

By

Ravi Kant Sahu  
Asst. Professor



**Lovely Professional University, Punjab**



# Outlines

- Introduction
- Traversing a Linked List
- Searching a Linked List
- Memory Allocation & Garbage Collection
- Overflow and Underflow
- Review Questions



# Introduction

- A linked list (One-way list) is a linear collection of data elements, called nodes, where the linear order is given by means of pointers.
- Each node is divided into two parts.
- First part contains the information of the element.
- Second part contains the address of the next node in the list.



# Traversing a Linked List

- PTR is a pointer variable which points to the node currently being processed.
- LINK [PTR] points to the next node to be processed.

## □ Algorithm (Traversing a linked list)

1. Set PTR = START. [*Initialize pointer PTR*]
2. Repeat step 3 and 4 while PTR ≠ NULL.
  3. Apply PROCESS to INFO[PTR].
  4. Set PTR = LINK [PTR]. [*PTR points to next node*]  
[*End of Step 2 Loop.*]
5. EXIT



# Problems

- Write an algorithm to find out the sum of ALTERNATE NODES in a linked list containing integers.
- Write an algorithm to modify each element of an integer linked list such that
  - (a) each element is double of its original value.
  - (b) each element is sum of its original value and its previous element.
- Write an algorithm to find out the maximum and minimum data element from an integer linked list.



# SEARCHING A LINKED LIST



# Searching a Linked List (1)

- List is Unsorted

**SEARCH (INFO, LINK, START, ITEM, LOC)**

1. Set PTR = START, LOC=NULL.
2. Repeat Step 3 While PTR ≠ NULL.
  3. If ITEM = INFO [PTR], then:  
Set LOC = PTR, and EXIT.  
Else:  
Set PTR = LINK [PTR]. *[PTR points to next node]*  
*[End of if Structure.]*  
*[End of step 2 Loop.]*
4. IF LOC = NULL, Then Print “ITEM not Found” *[Search is Unsuccessful.]*
5. Exit . *[Return LOC and Exit.]*



# Searching a Linked List (2)

## □ List is Sorted

**SEARCH (INFO, LINK, START, ITEM, LOC)**

1. Set PTR = START and LOC = NULL.
2. Repeat Step 3 While PTR  $\neq$  NULL.
  3. If ITEM > INFO [PTR], then:  
PTR = LINK [PTR]. *[PTR points to next node]*  
Else if ITEM = INFO [PTR], then:  
Set LOC = PTR, and EXIT. *[Search is successful.]*  
Else:  
Set LOC = NULL, and EXIT. *[ITEM exceeds INFO[PTR]...]*  
*[End of if Structure.]*  
*[End of step 2 Loop.]*
  4. Return LOC .
  5. Exit.



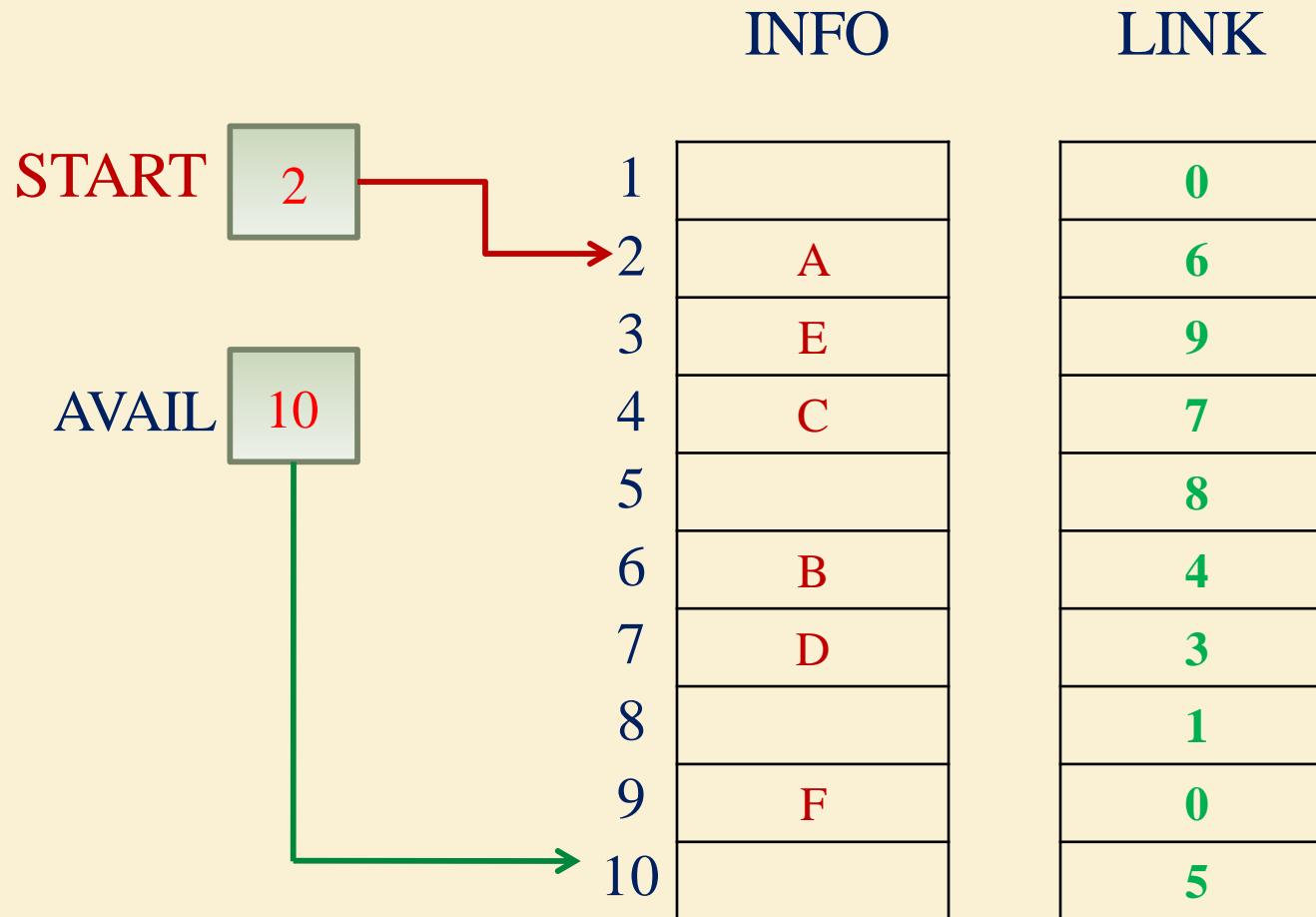
# Memory Allocation

- Together with the linked list, a special list is maintained which consists of unused memory cells.
- This list has its own pointer.
- This list is called *List of available space* or *Free-storage list* or *Free pool*.



# Free Pool

- Linked list with free pool or list of Available space.





# Garbage Collection

- Garbage collection is a technique of collecting all the deleted spaces or unused spaces in memory.
- The OS of a computer may periodically collect all the deleted space onto the free-storage list.
- Garbage collection may take place when there is only some minimum amount of space or no space is left in free storage list.
- Garbage collection is invisible to the programmer.



# Garbage Collection Process

- Garbage collection takes place in two steps.
1. The computer runs through all lists tagging those cells which are currently in use.
  2. Then computer runs through the memory, collecting all the untagged spaces onto the free storage list.



# Overflow and Underflow

□ **Overflow:** When a new data are to be inserted into a data structure but there is no available space i.e. the free storage list is empty.

- Overflow occurs when  $AVAIL = NULL$ , and we want insert an element.
- Overflow can be handled by printing the '*OVERRLOW*' message and/or *by adding space* to the underlying data structure.



# Overflow and Underflow

- **Underflow:** When a data item is to be deleted from an empty data structure.
- Underflow occurs when  $START = NULL$ , and we want to delete an element.
- Underflow can be handled by printing the '*UNDERFLOW*' message.



Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Review Questions

- Write an algorithm to find out the maximum and minimum data element from an integer linked list.
- What is the condition of Overflow and underflow?
- What are the steps followed during garbage collection?

# Data Structures

## Lecture 6: Insertion in Linked List

By

Ravi Kant Sahu  
Asst. Professor



**Lovely Professional University, Punjab**



# Outlines

- Insertion Algorithm
  - Insertion at the beginning
  - Insertion after a given node
  - Insertion into a sorted list
- Review Questions

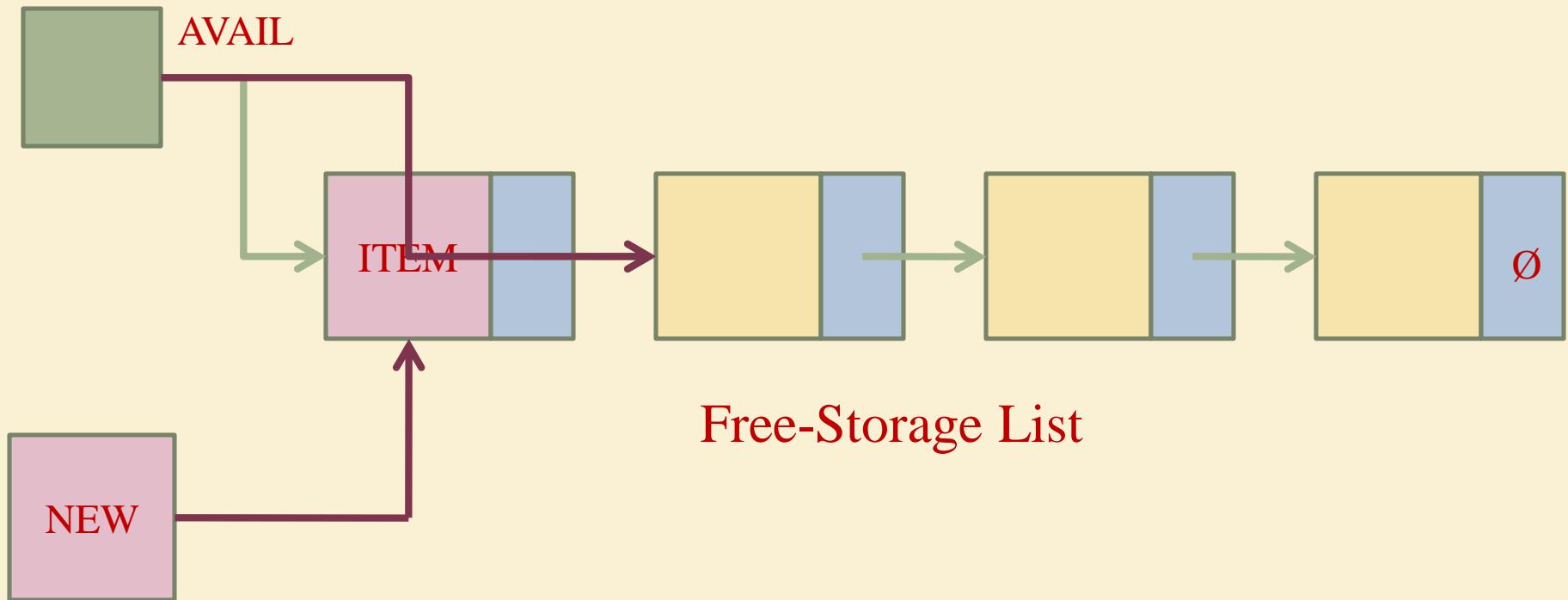


# Insertion into a Linked List

- New node N (which is to be inserted) will come from AVAIL list.
  - First node in the AVAIL list will be used for the new node N.
  
- Types of insertion:
  - Insertion at the beginning
  - Insertion between two nodes

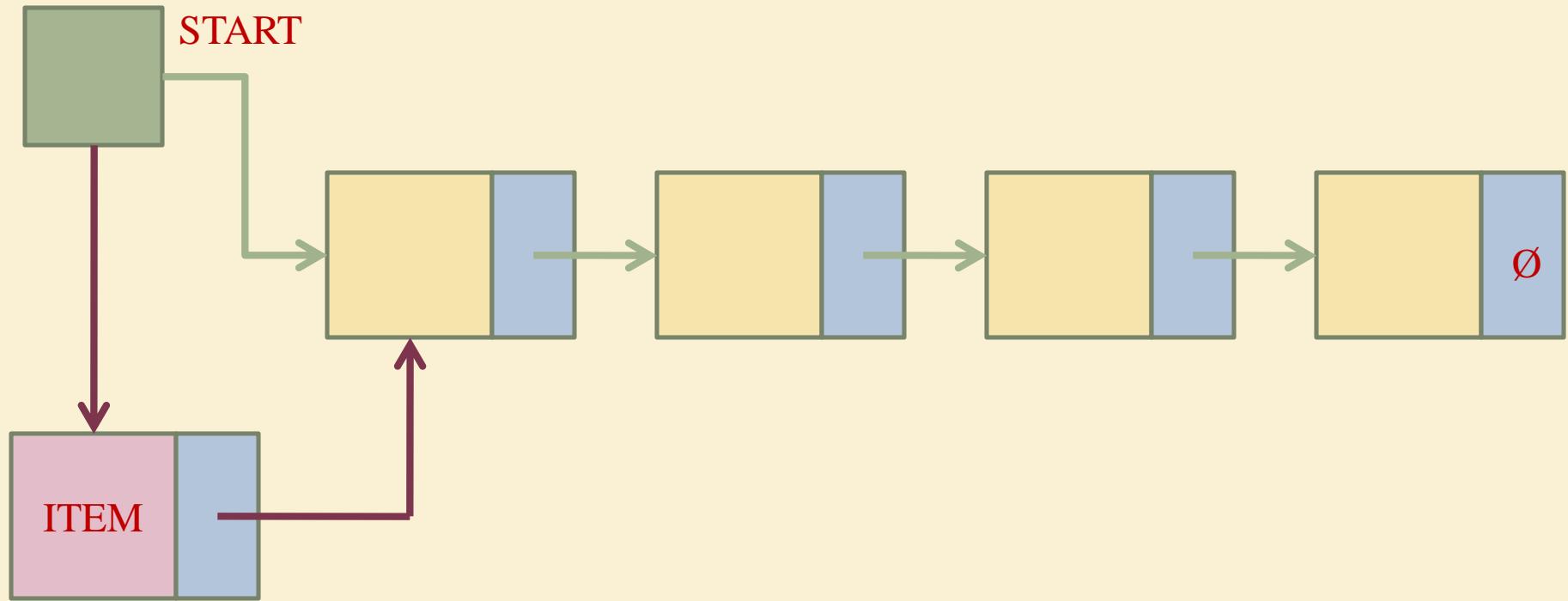


# Checking the Available List





# Insertion at the beginning of Linked List





# Work Space

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



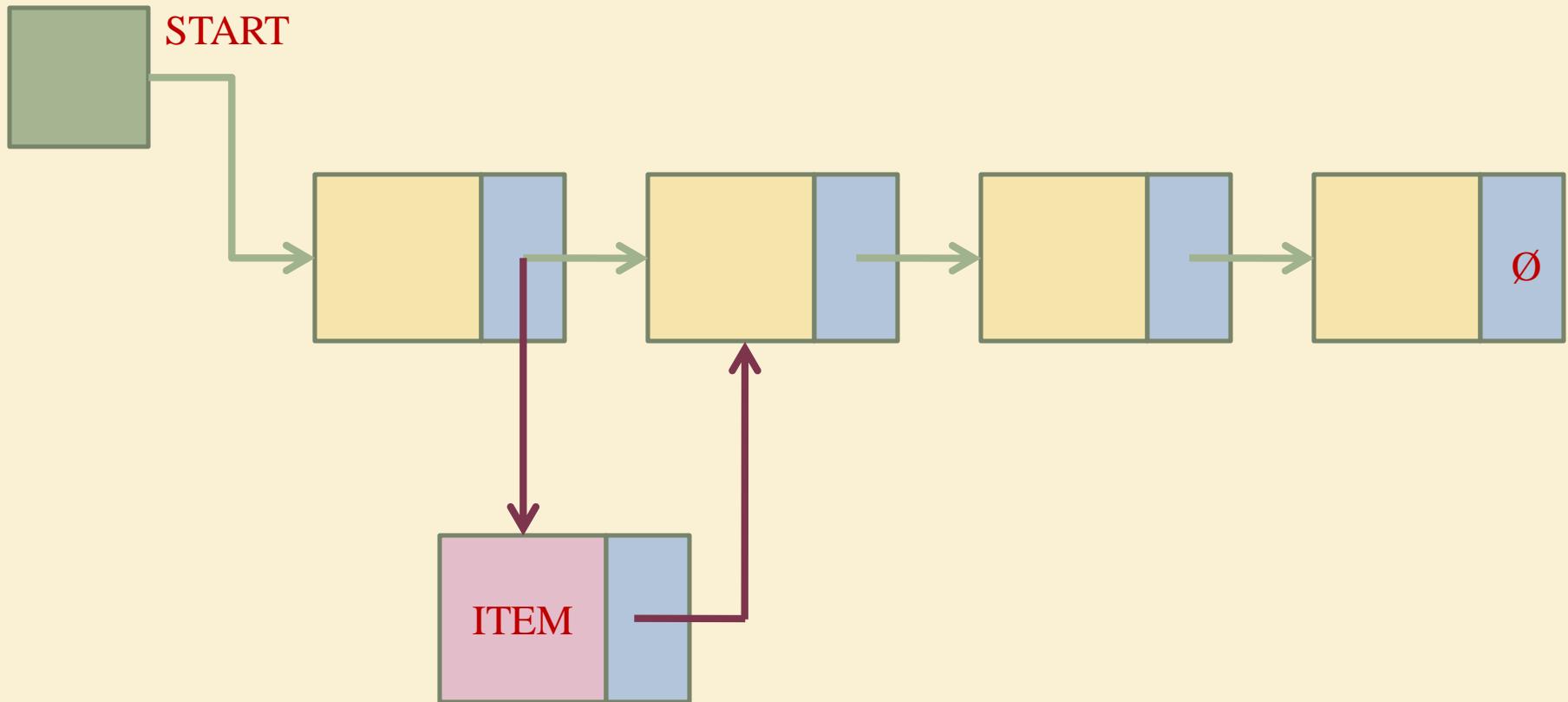
# Insertion Algorithm (Beginning of the list)

**INSFIRST (INFO, LINK, START, AVAIL, ITEM)**

1. [OVERFLOW?] If  $AVAIL = \text{NULL}$ , then: Write: OVERFLOW, and Exit.
2. [Remove first node from AVAIL list]  
Set  $NEW = AVAIL$  and  $AVAIL = \text{LINK} [AVAIL]$ .
3. Set  $INFO [NEW] = ITEM$ . [Copy the new data to the node].
4. Set  $LINK [NEW] = START$ . [New node points to the original first node].
5. Set  $START = NEW$ . [ $START$  points to the new node.]
6. EXIT.



# Insertion after a given node





# Insertion Algorithm (After a given node)

**INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM)**

1. [OVERFLOW?] If  $AVAIL = \text{NULL}$ , then: Write: OVERFLOW, and Exit.
2. [Remove first node from AVAIL list]  
Set  $NEW = AVAIL$  and  $AVAIL = \text{LINK} [AVAIL]$ .
3. Set  $INFO [NEW] = ITEM$ . [Copy the new data to the node].
4. If  $LOC = \text{NULL}$ , then: [Insert as a first node]  
Set  $\text{LINK} [NEW] = \text{START}$  and  $\text{START} = NEW$ .  
Else: [Insert after node with location LOC.]  
Set  $\text{LINK} [NEW] = \text{LINK} [LOC]$  and  $\text{LINK} [LOC] = NEW$ .  
[End of If structure.]
5. Exit.



# Insertion into a sorted Linked List

- If ITEM is to be inserted into a sorted linked list. Then ITEM must be inserted between nodes A and B such that:

$$\text{INFO [A]} < \text{ITEM} < \text{INFO [B]}$$

- First of all find the location of Node A
- Then insert the node after Node A.

**INSERT (INFO, LINK, START, AVAIL, ITEM)**

1. CALL FIND\_A (INFO, LINK, START, AVAIL, ITEM)  
*[USE Algorithm FIND\_A to find the location of node preceding ITEM.]*
2. CALL INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM)  
*[Insert ITEM after a given node with location LOC.]*
3. Exit.



## FIND\_A (INFO, LINK, START, ITEM, LOC)

1. [List Empty?] If  $START = \text{NULL}$ , then: Set  $LOC = \text{NULL}$ , and Return.
2. [Special Case?] If  $ITEM < \text{INFO} [START]$ , then: Set  $LOC = \text{NULL}$ , and Return.
3. Set  $SAVE = START$  and  $PTR = \text{LINK} [START]$ . [Initializes pointers]
4. Repeat step 5 and 6 while  $PTR \neq \text{NULL}$ .
  5. If  $ITEM < \text{INFO} [PTR]$  then:  
Set  $LOC = SAVE$ , and Return.  
[End of If Structure.]
  6. Set  $SAVE = PTR$  and  $PTR = \text{LINK} [PTR]$ . [Update pointers]  
[End of Step 4 Loop.]
7. Set  $LOC = SAVE$ .
8. Return.



Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Review Questions

- What is the condition for the list being empty?
- How will you insert a node in a linked list after a given node?
- Which pointer fields are changed when:
  - a node is inserted after a given node
  - a node is inserted at the end of list
  - a node is inserted at the beginning of the list.



# Review Questions

- Correct the following algorithm such that ptr contains the address of the last node:
  1. Set PTR = START
  2. Repeat Steps 3 WHILE PTR != NULL
  3.   Set LINK[PTR] = PTR.  
[End of the Loop]
  4. Return PTR.

# Data Structures

## Lecture 7: Deletion from Linked List

By

Ravi Kant Sahu  
Asst. Professor



**Lovely Professional University, Punjab**



# Outlines

- Deletion from a Linked List
- Deletion Algorithm
  - Deleting the Node following a given Node
  - Deleting a Node with a given ITEM of Information
- Review Questions

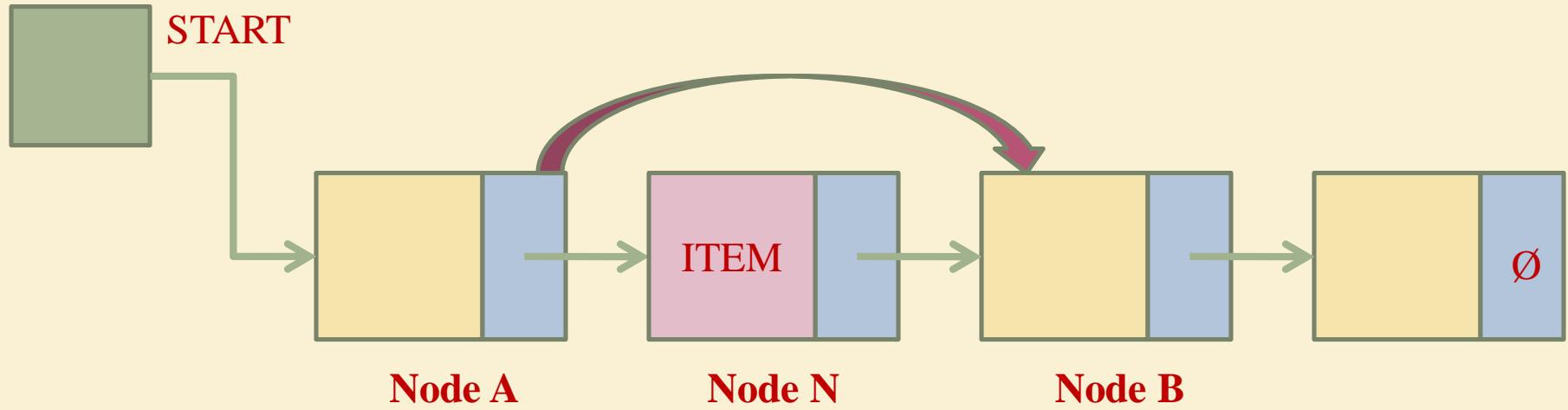


# Deletion from a Linked List

- A node N is to be deleted from the Linked List.
  - Node N is between node A and node B.
  
- Deletion occurs as soon as the next pointer field of node A is changed so that it points to node B.
  
- Types of Deletion:
  - Deleting the node following a given node
  - Deleting the Node with a given ITEM of Information.



# Deletion from Linked List





# Maintaining the AVAIL List

- After the deletion of a node from the list, memory space of node N will be added to the beginning of AVAIL List.
- If LOC is the Location of deleted node N:

$$LINK[LOC] = AVAIL$$
$$AVAIL = LOC$$



# Work Space

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Deleting the Node Following a given Node

**DEL (INFO, LINK, START, AVAIL, LOC, LOCP)**

1. If  $\text{LOCP} = \text{NULL}$ , then:

Set  $\text{START} = \text{LINK} [\text{START}]$ . [Delete First node.]

Else:

Set  $\text{LINK} [\text{LOCP}] = \text{LINK} [\text{LOC}]$ . [Delete node N.]  
[End of If Structure.]

2. [Return Deleted node to the AVAIL list]

Set  $\text{LINK} [\text{LOC}] = \text{AVAIL}$  and  $\text{AVAIL} = \text{LOC}$ .

3. EXIT.



# Deleting the Node with a given ITEM of Information

**DELETE (INFO, LINK, START, AVAIL, ITEM)**

1. Call **FIND\_B (INFO, LINK, START, ITEM, LOC, LOCP)**  
*[Find the Location of node N and its preceding node]*
2. If LOC = NULL, then: Write: ITEM not in LIST and EXIT.
3. *[Delete node].*  
If LOCP = NULL, then:  
    Set START = LINK [START]. *[Delete First node]*  
Else:  
    Set LINK [LOCP] = LINK [LOC].  
*[End of If Structure.]*
4. *[Return Deleted node to the AVAIL list]*  
Set LINK [LOC] = AVAIL and AVAIL= LOC.
5. EXIT.



## FIND\_B (INFO, LINK, START, ITEM, LOC, LOCP)

1. [List Empty?] If START = NULL, then:  
    Set LOC = NULL, LOCP = NULL and Return.  
    [End of If Structure.]
2. [ITEM in First node?] If INFO [START] = ITEM, then:  
    Set LOC = START, and LOCP = NULL, and Return.  
    [End of If Structure.]
3. Set SAVE = START and PTR = LINK [START]. [Initializes pointers]
4. Repeat step 5 and 6 while PTR  $\neq$  NULL.
5.   If INFO [PTR] = ITEM, then:  
      Set LOC = PTR and LOCP = SAVE, and Return.  
      [End of If Structure.]
6.   Set SAVE = PTR and PTR = LINK [PTR]. [Update pointers]  
    [End of Step 4 Loop.]
7. Set LOC = NULL. [Search Unsuccessful.]
8. Return.



# Exercise

- Write an algorithm to delete the last node of a linked list.  
[Hint: Find out the address of last node and second last node]



# Work Space

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)

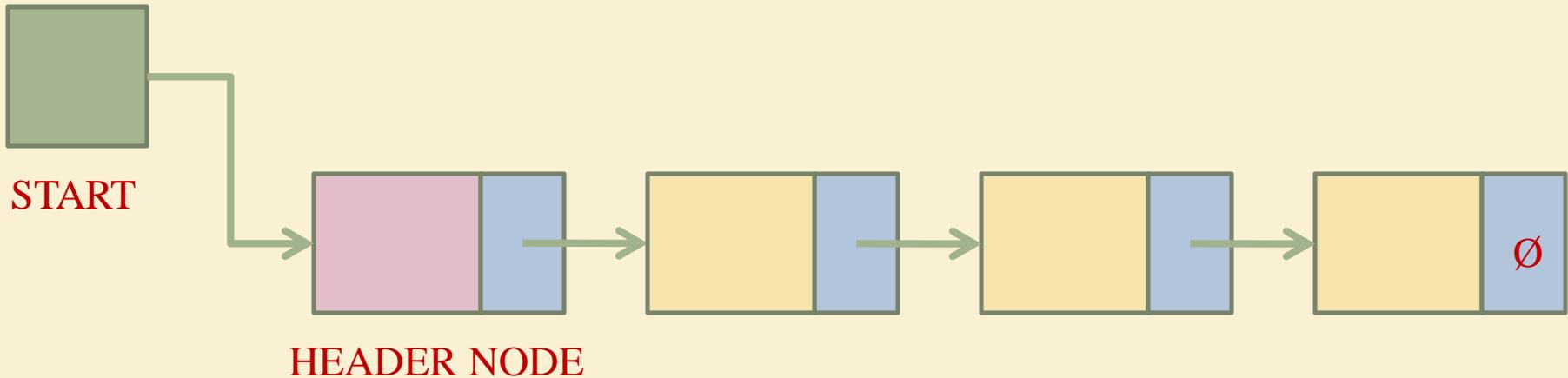


# HEADER LINKED LIST



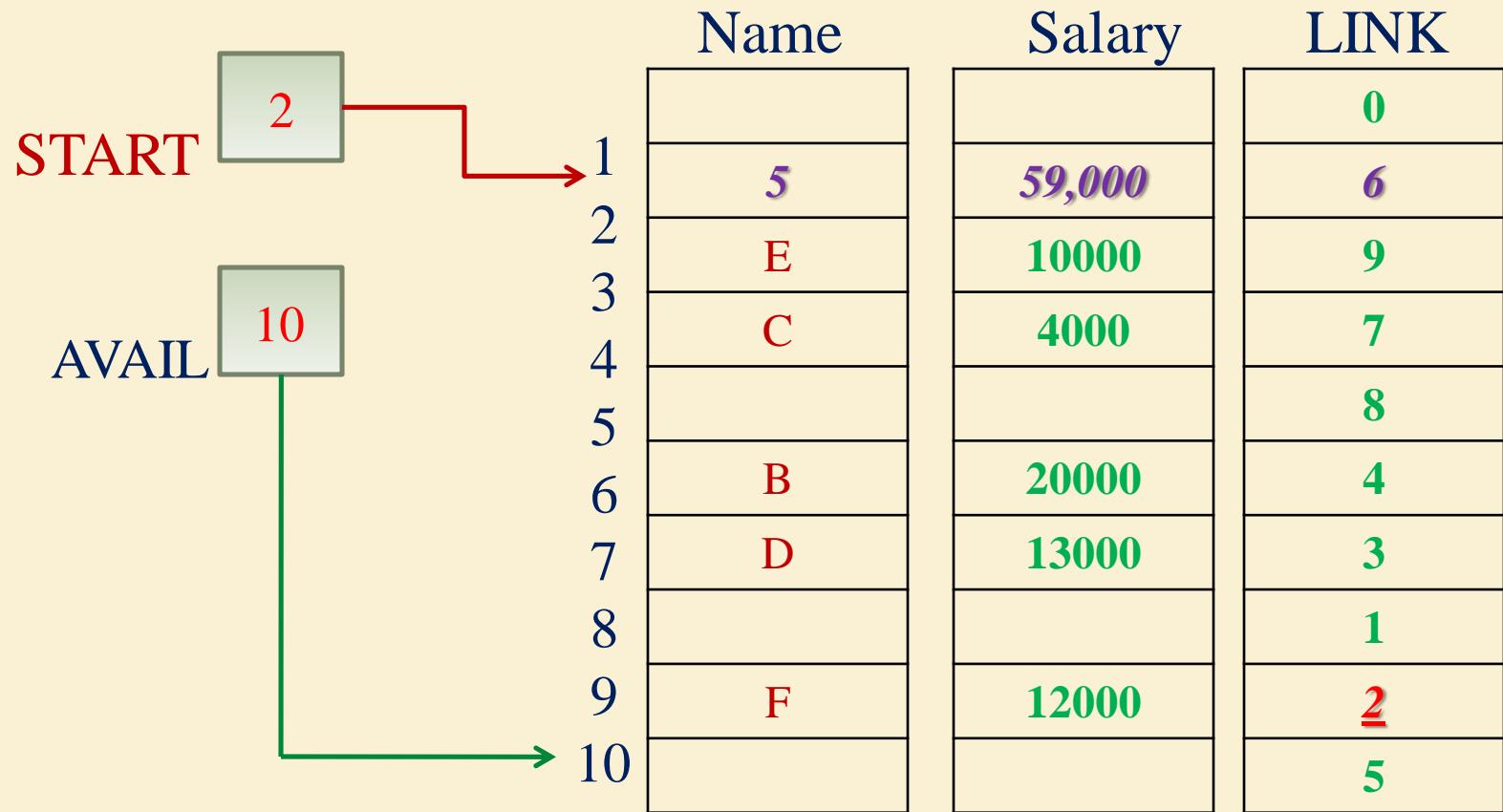
# Header Linked List

- A header linked list which always contains a special node, called the header node, at the beginning of the list.





# Header Linked List





# Advantages of Header Linked List

- Header linked list contains a special node at the top.
- This header node need not represent the same type of data that succeeding nodes do.
- It can have data like, number of nodes, any other data...
- Header node can access the data of all the nodes in the linked list.

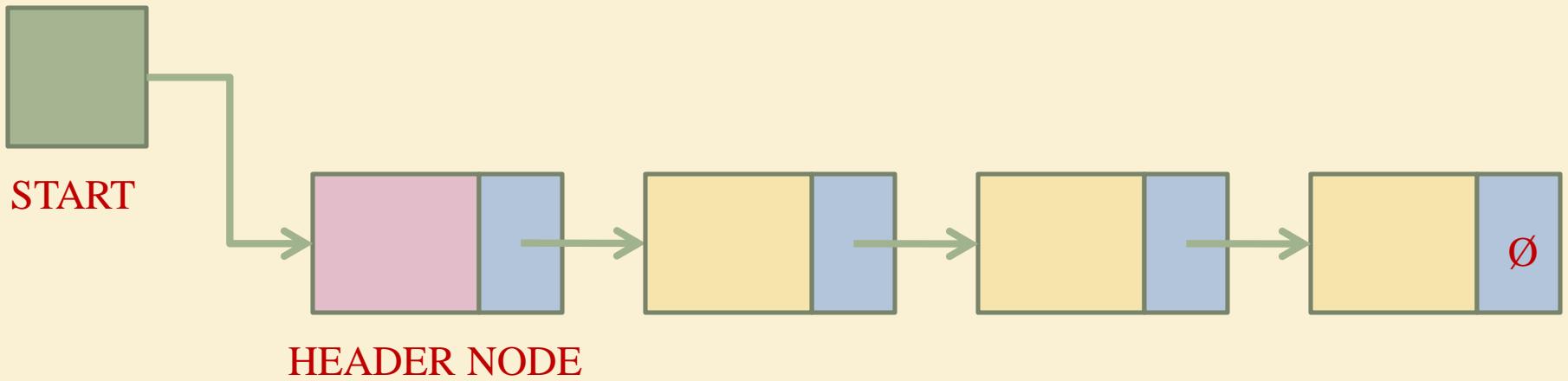


# Types of Header Linked List

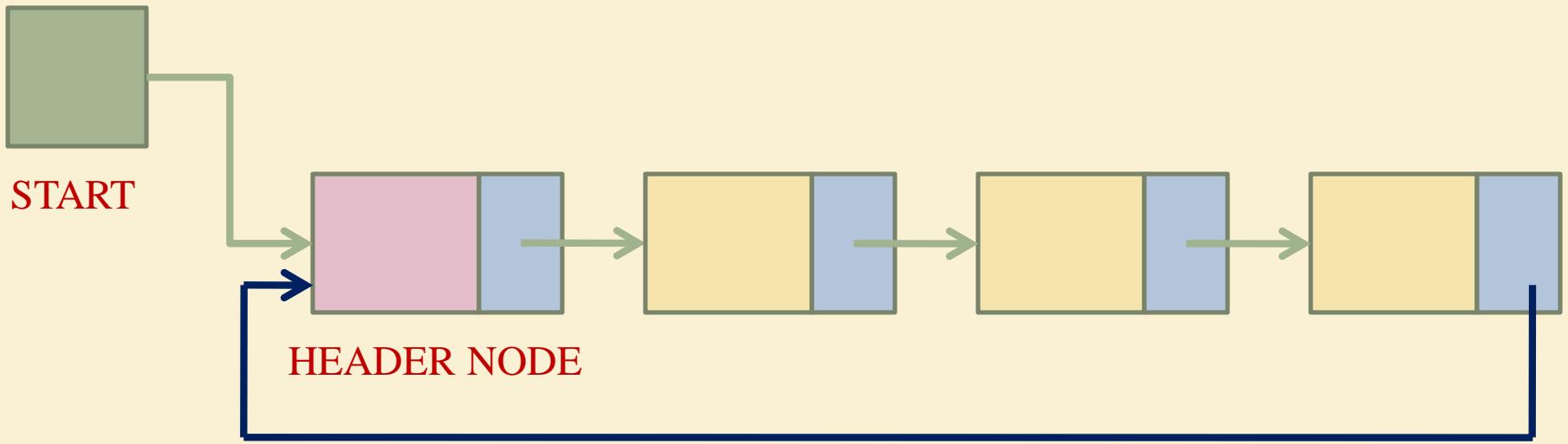
- A **Grounded header list** is a header list where the last node contains the null pointer.
  
- A **Circular header list** is a header list where the last node points back to the header node.

## Note:

- Unless otherwise stated or implied, header list will always be circular list.
- Accordingly, in such a case, the header node also acts as a sentinel indicating the end of the list.



## Grounded Header List



## Circular Header List



- If Link [START] = NULL,  
then, Grounded Header List is Empty.
- If Link [START] = START,  
then, Circular Header List is Empty.



# Traversing a Circular Header List

## □ Algorithm (Traversing a Circular Header list)

1. Set PTR = LINK [START]. *[Initialize pointer PTR]*
2. Repeat step 3 and 4 while PTR  $\neq$  START.
  3.     Apply PROCESS to INFO[PTR].
  4.     Set PTR = LINK [PTR]. *[PTR points to next node]*  
*[End of Step 2 Loop.]*
5. EXIT



# Use of Header Linked List

- Header Linked lists are frequently used for maintaining *Polynomials* in memory.



Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Review Questions

- What is the condition for the list being empty?
  
- Which pointer fields are changed when:
  - a node is deleted after a given node
  - a node is deleted which is at the end of list
  - a node is deleted at the beginning of the list.

# Data Structures

## Lecture 8: Header Linked List

By

Ravi Kant Sahu  
Asst. Professor



**Lovely Professional University, Punjab**



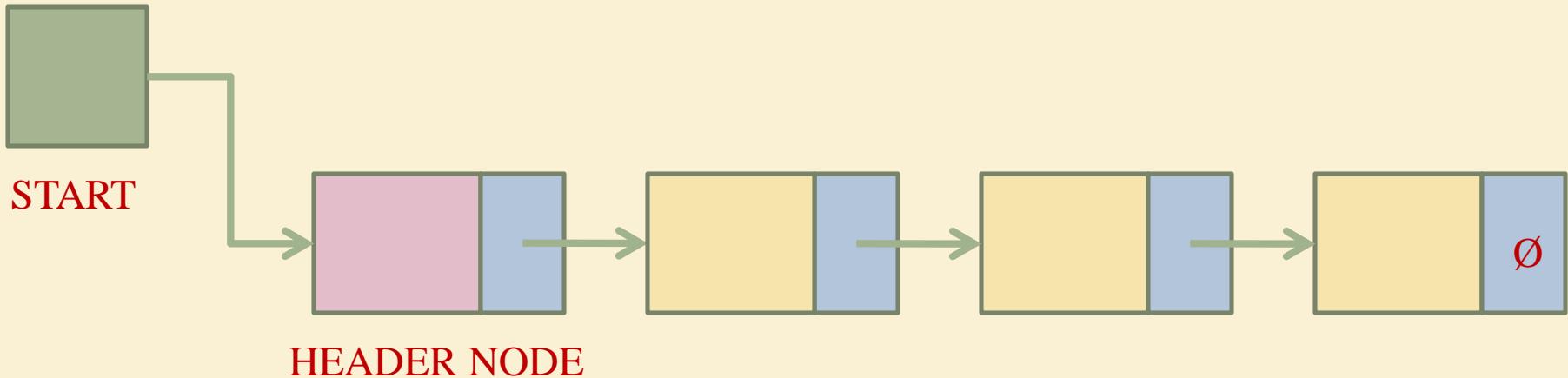
# Outlines

- Introduction
- Header Linked List
- Advantages of Header Linked List
- Types of Header Linked List
- Review Questions



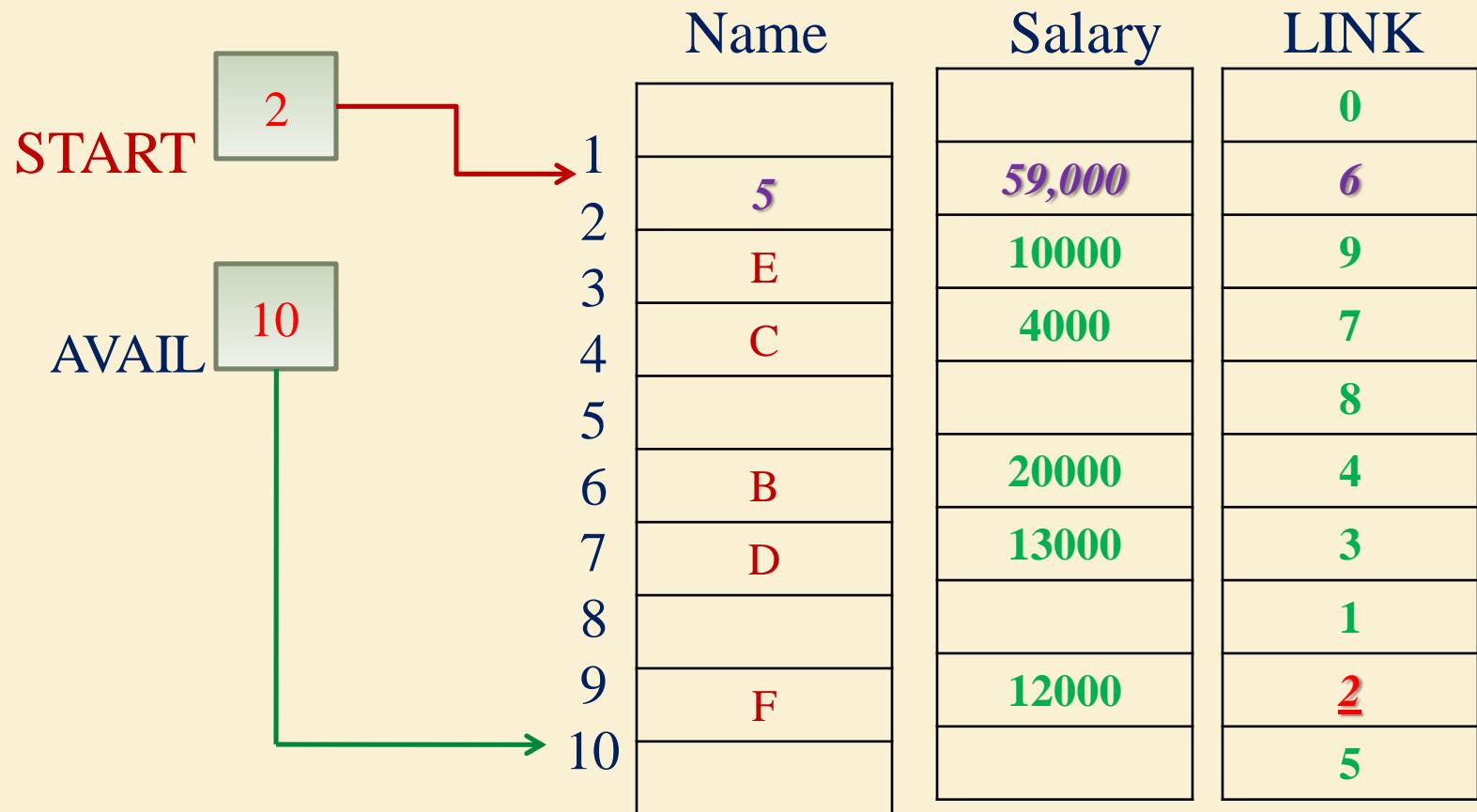
# Header Linked List

- A header linked list which always contains a special node, called the header node, at the beginning of the list.





# Header Linked List





# Advantages of Header Linked List

- Header linked list contains a special node at the top.
- This header node need not represent the same type of data that succeeding nodes do.
- It can have data like, number of nodes, any other data...
- Header node can access the data of all the nodes in the linked list.

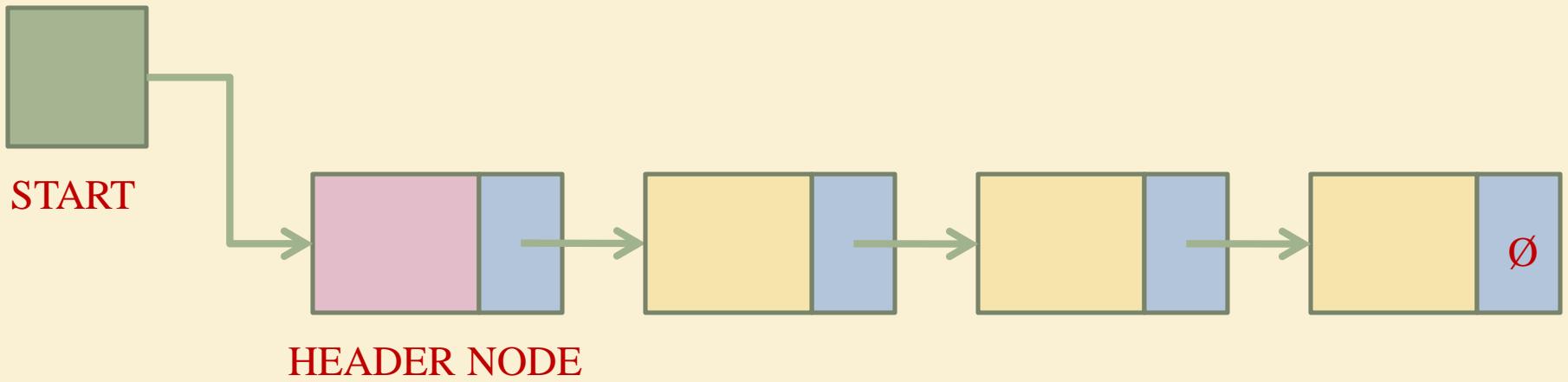


# Types of Header Linked List

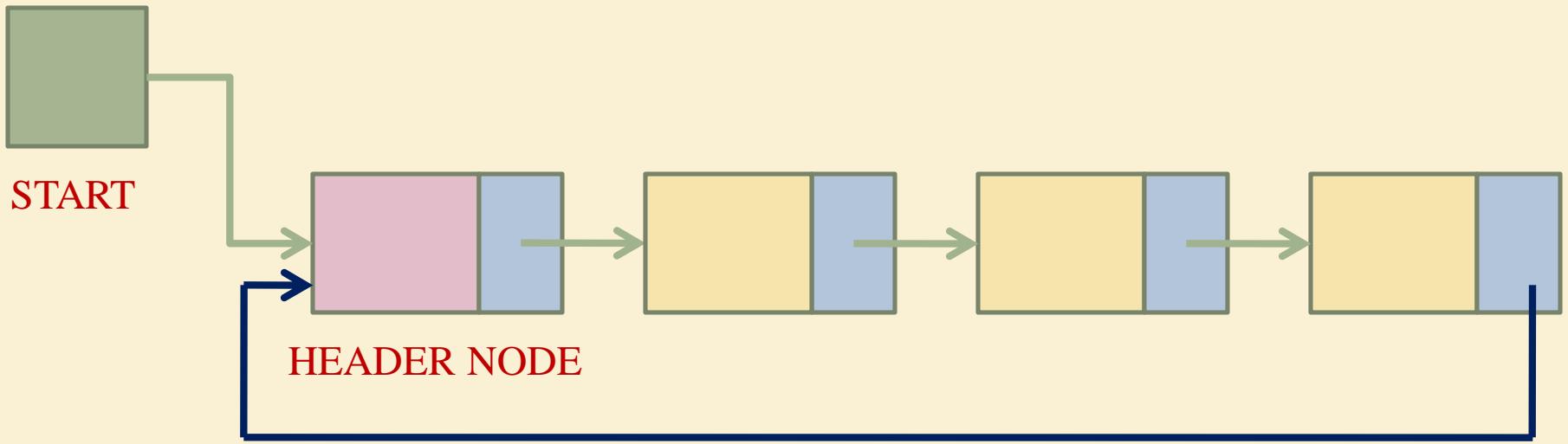
- A **Grounded header list** is a header list where the last node contains the null pointer.
  
- A **Circular header list** is a header list where the last node points back to the header node.

## Note:

- Unless otherwise stated or implied, header list will always be circular list.
- Accordingly, in such a case, the header node also acts as a sentinel indicating the end of the list.



## Grounded Header List



## Circular Header List



- If Link [START] = NULL,  
then, Grounded Header List is Empty.
- If Link [START] = START,  
then, Circular Header List is Empty.



# Traversing a Circular Header List

## □ Algorithm (Traversing a Circular Header list)

1. Set PTR = LINK [START]. *[Initialize pointer PTR]*
2. Repeat step 3 and 4 while PTR  $\neq$  START.
  3.     Apply PROCESS to INFO[PTR].
  4.     Set PTR = LINK [PTR]. *[PTR points to next node]*  
*[End of Step 2 Loop.]*
5. EXIT



# Use of Header Linked List

- Header Linked lists are frequently used for maintaining *Polynomials* in memory.



Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Review Questions

- What is Header Node?
- How a Linked List is different from Header linked list?
- What is Grounded Header list and circular header list?

# Data Structures

## Lecture 9: Two-Way List

By

Ravi Kant Sahu  
Asst. Professor



Lovely Professional University, Punjab



# Outlines

- Introduction
- Two-Way List
- Two-Way Header List
- Operations on Two-Way List
  - Traversing
  - Searching
  - Deleting
  - Inserting
- Review Questions



# Introduction

- If a list can be traversed in only one direction, it is called One-way List.
- List Pointer variable START points to the first node or header node.
- Next-pointer field LINK is used to point to the next node in the list.
- Only next node can be accessed.
- Don't have access to the preceding node.



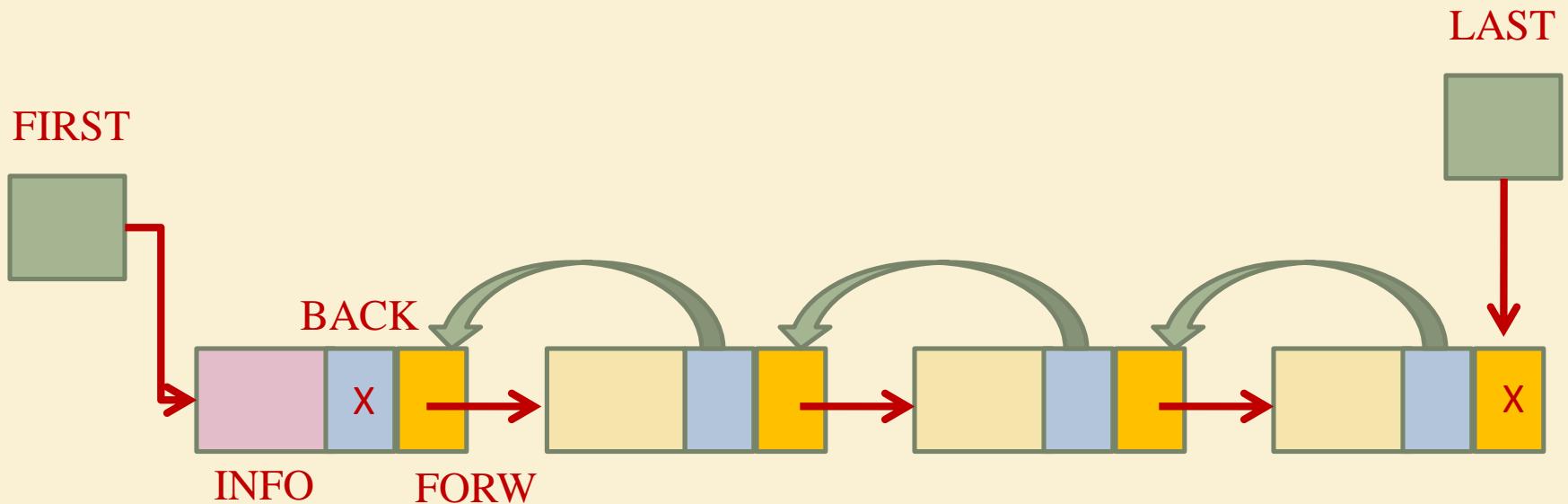
# Two-Way List

- A Two-Way list is a linear collection of data elements, called nodes, where each node N is divided into three parts:
  1. An information field ***INFO*** which contains the data of N.
  2. A pointer field ***FORW*** which contains the location of the next node in the list.
  3. A pointer field ***BACK*** which contains the location of the preceding node in the list.



# Two-Way List ...

- The Two-Way list requires two list pointer variables:
  1. FIRST: points to the first node in the list.
  2. LAST: points to the last node in the list.





# Key Points

- If LOCA and LOCB are the locations of node A and node B respectively, then

$$\begin{aligned} \text{FORW [LOCA]} &= \text{LOCB}, \text{ iff} \\ \text{BACK [LOCB]} &= \text{LOCA} \end{aligned}$$

- Two way lists are maintained in memory by means of linear arrays as in one way lists.
- But now we require two pointer arrays BACK and FORW.



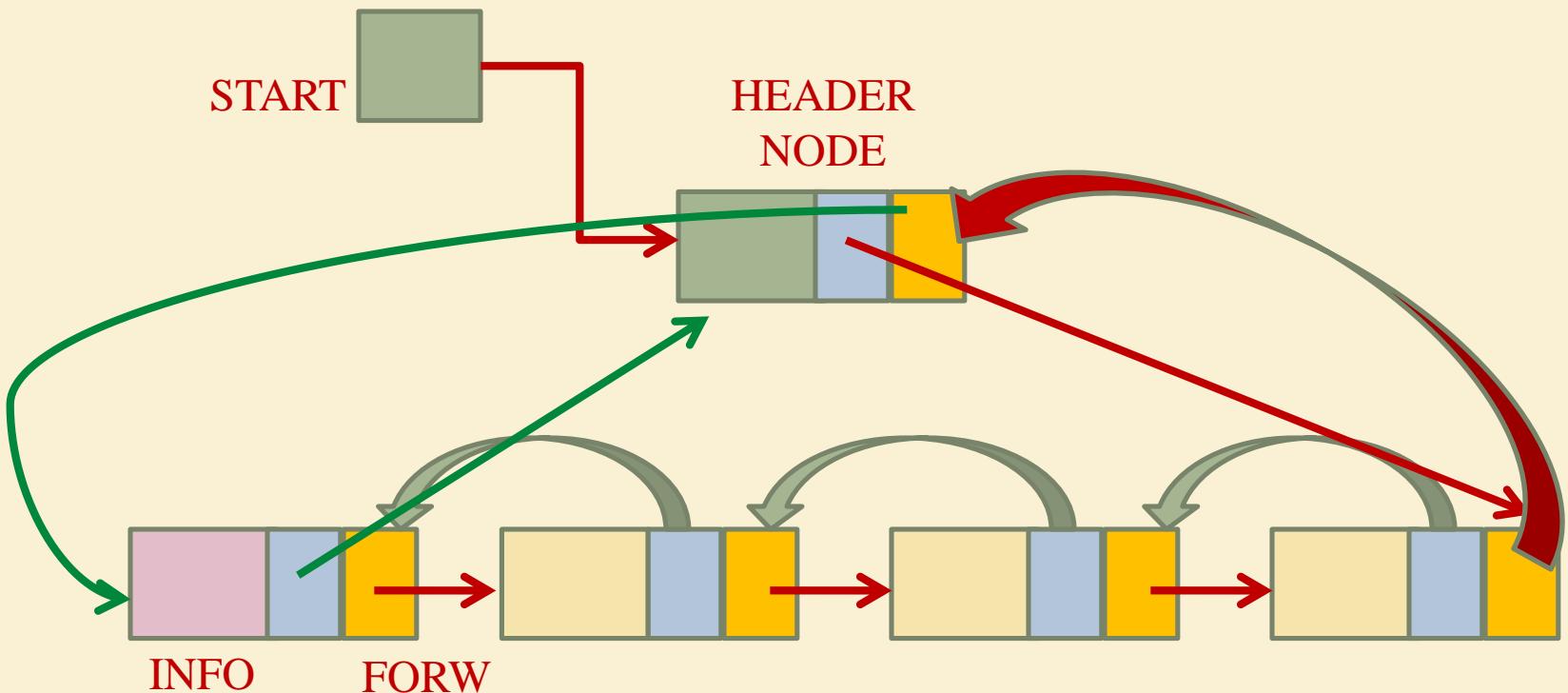
# Work Space

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Two-Way Header List

- It is circular because the two end nodes point back to the header node.
- Only one list pointer variable START is required.





# Insertion in a Two-way List

- INST\_TWL (INFO, FORW, BACK, START, AVAIL, LOCA, LOCB, ITEM)
  1. [OVERFLOW?] If AVAIL = NULL, then: Write: Overflow and Exit.
  2. [Remove Node from AVAIL List and copy the Item into it.]  
Set NEW = AVAIL, AVAIL = FORW [AVAIL],  
INFO [NEW] = ITEM.
  3. [Insert Node into the list.]  
Set FORW [LOCA] = NEW, FORW [NEW] = LOCB,  
BACK [LOCB] = NEW, and BACK [NEW] = LOCA.
  4. Exit.



# Deletion in a Two-way List

- DEL\_TWL (INFO, FORW, BACK, START, AVAIL, LOC)
  1. [Delete Node.]  
Set FORW [BACK [LOC] ] = FORW [LOC] and  
BACK [FORW [LOC] ] = BACK [LOC].
  2. [Return Node to AVAIL list.]  
Set FORW [LOC] = AVAIL and AVAIL = LOC.
  3. Exit.



# Exercise

Write an algorithm to insert an item Y before X in a Two-Way Linked List.

Write an algorithm to check whether the Linked list contains loop or not?

Write an algorithm to find out the intersection of 2 linked lists.



Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Review Questions

- What is the advantage of Two-way List over Linked List?
- How Traversing is done in TWL?
- What is the difference between Deletion in Linked List and TWL?
- How TWL is more efficient than Linked List in case of Searching?

# Data Structures

## Lecture 14: Stacks

By

Ravi Kant Sahu

Asst. Professor



**Lovely Professional University, Punjab**



# Outlines

- Introduction
- Basic Operations
- Array Representation of Stacks
- Linked Representation of Stacks
- Minimizing Overflow
- Review Questions



# Introduction

## Stacks:

- A stack is a list of elements in which an element may be *inserted or deleted only at one end*, called the “*TOP*” of the Stack.
- Stack is a LIFO (Last In First Out) data structure.
- Elements are *removed from a stack in the reverse order* of that in which they were inserted into the stack.



# Examples

- Stack of Dishes
- Stack of Books
- A packet of Biscuits etc.
- Note: AVAIL List is also implemented using STACK.

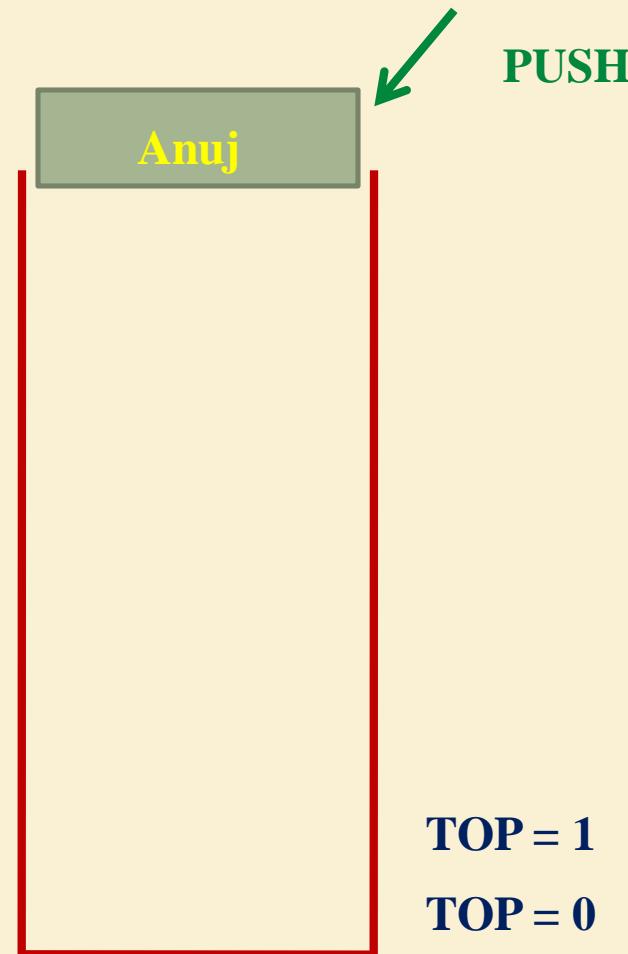


# Basic Operations

- **Push:** Process of inserting an element into Stack.
- **Pop:** Process of deleting the top most element from stack.



# STACK



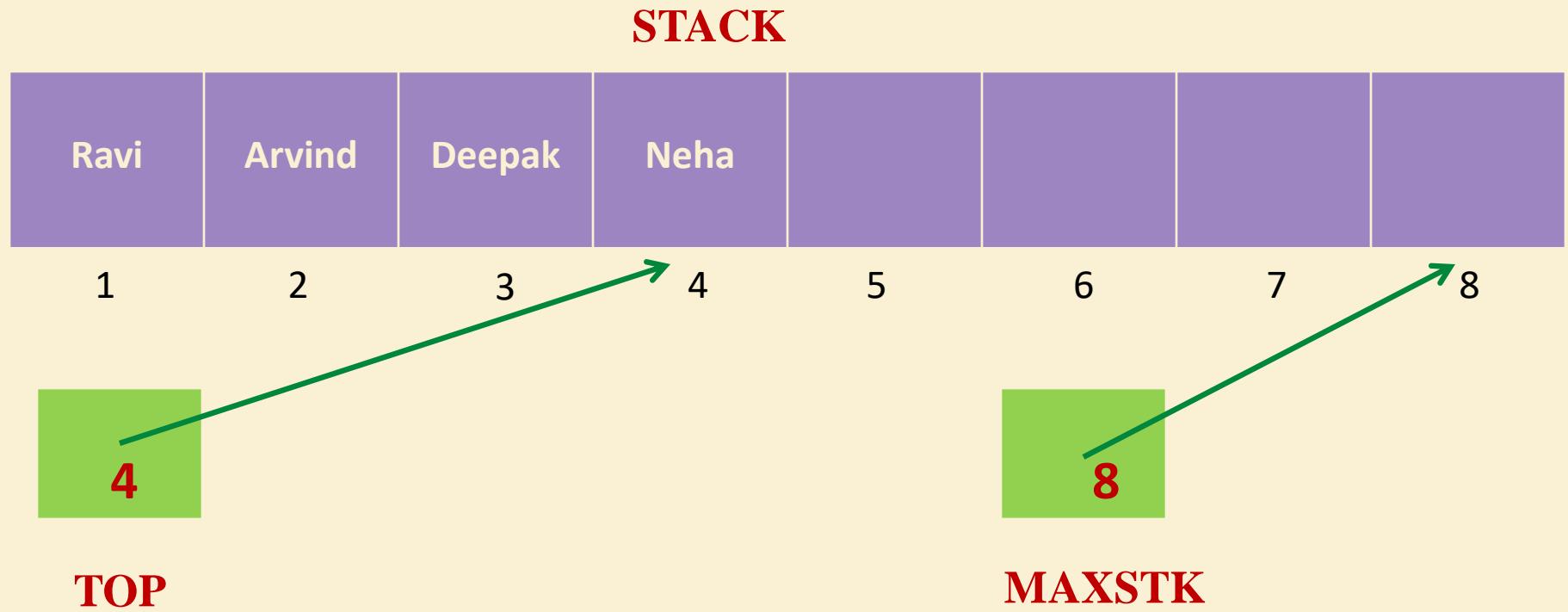


# Work Space

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Array Representation of Stacks





# Array Representation of Stacks

PUSH (STACK, TOP, MAXSTK, ITEM)

1. [Stack already filled?]  
If  $\text{TOP} = \text{MAXSTK}$ , then: Print: OVERFLOW and Return.
2. Set  $\text{TOP} = \text{TOP} + 1$ . [Increase TOP by 1.]
3. Set  $\text{STACK}[\text{TOP}] = \text{ITEM}$ . [Insert ITEM into the TOP.]
4. Return



# Array Representation of Stacks

POP (STACK, TOP, ITEM)

1. [Stack has an ITEM to be removed?]  
If  $\text{TOP} = 0$ , then: Print: UNDERFLOW and Return.
2. Set  $\text{ITEM} = \text{STACK} [\text{TOP}]$ . [ITEM is the TOP element.]
3. Set  $\text{TOP} = \text{TOP} - 1$ . [Decrease TOP. by 1.]
4. Return



# Work Space

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Linked Representation of Stacks

PUSH\_LINKSTACK (INFO, LINK, TOP, AVAIL, ITEM)

1. [Overflow?] If AVAIL = NULL, then: Print: OVERFLOW and Exit.
2. Set NEW = AVAIL and AVAIL = LINK [AVAIL].  
[Remove first node from AVAIL List.]
3. Set INFO [NEW] = ITEM.
4. Set LINK [NEW] = TOP.  
[NEW node points to the original top node in the stack.]
5. Set TOP = NEW.  
[Reset TOP to point to new node . ]
6. Exit.



# Linked Representation of Stacks

POP\_LINKSTACK (INFO, LINK, TOP, AVAIL, ITEM)

1. [Underflow?] If  $\text{TOP} = \text{NULL}$ , then: Print: UNDERFLOW and Exit.
2. Set  $\text{ITEM} = \text{INFO} [\text{TOP}]$ .  
[Copy the top element into Item.]
3. Set  $\text{TEMP} = \text{TOP}$  and  $\text{TOP} = \text{LINK} [\text{TOP}]$ .  
[Remember the old value of TOP and Reset TOP.]
4. Set  $\text{LINK} [\text{TEMP}] = \text{AVAIL}$  and  $\text{AVAIL} = \text{TEMP}$ .  
[Return deleted node to AVAIL List.]
5. Exit.



# Minimizing Overflow

- Stack involves a Time-space Trade-off.
- By reserving a great deal of space for stack, we can minimize number of overflows.



# Arithmetic Expressions

- Arithmetic Expressions involve *constants* and *operations*.
- Binary operations have different levels of precedence.
  - First : Exponentiation (^)
  - Second: Multiplication (\*) and Division (/)
  - Third : Addition (+) and Subtraction (-)



# Example

- Evaluate the following Arithmetic Expression:

$$5^2 + 3 * 5 - 6 * 2 / 3 + 24 / 3 + 3$$

- First:

$$25 + 3 * 5 - 6 * 2 / 3 + 24 / 3 + 3$$

- Second:

$$25 + 15 - 4 + 8 + 3$$

- Third:

47



# Polish Notation

- **Infix Notation:** Operator symbol is placed between the two operands.

Example:

$$(5 * 3) + 2 \quad \& \quad 5 * (3 + 2)$$



# Polish Notation

- Polish Notation: The Operator Symbol is placed before its two operands.

Example: + A B, \* C D, / P Q etc.

- Named after the Polish Mathematician Jan Lukasiewicz.
- The order in which the operations are to be performed is completely determined by the positions of operators and operands in the expression.



# Examples

- $(A + B) * C =$   
 $* + ABC$
- $A + (B * C) =$   
 $+ A * BC$
- $(A + B) / (C - D) =$   
 $/ +AB -CD$
- Also Known as Prefix Notation.



# Reverse Polish Notation

- Reverse Polish Notation: The Operator Symbol is placed after its two operands.

Example: A B+, C D\*, P Q/ etc.

- Also known as Postfix Notation.



# Work Space

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Evaluation of Postfix Expression

- P is an arithmetic expression in Postfix Notation.
1. Add a right parenthesis ")" at the end of P.
  2. Scan P from left to right and Repeat Step 3 and 4 for each element of P until the sentinel ")" is encountered.
  3. If an operand is encountered, put it on STACK.
  4. If an operator @ is encountered, then:
    - (A) Remove the two top elements of STACK, where A is the top element and B is the next to top element.
    - (B) Evaluate B @ A.
    - (C) Place the result of (B) back on STACK.

[End of if structure.]

[End of step 2 Loop.]
  5. Set VALUE equal to the top element on STACK.
  6. Exit.



# Review Questions

- Evaluate the following Post-fix Expression:
  - 12 7 3 - / 2 1 5 + \* +



# Infix to Postfix Transformation

## POLISH (Q, P)

1. PUSH "(" on to STACK and add ")" to the end of Q.
2. Scan Q from left to right and Repeat steps 3 to 6 for each element of Q until the STACK is empty:
  3. If an operand is encountered, add it to P.
  4. If a left parenthesis is encountered, push it onto STACK.
  5. If an operator is encountered, then:
    - (a) Repeatedly POP from STACK and add to P each operator (On the TOP of STACK) which has the same precedence as or higher precedence than @.
    - (b) Add @ to STACK.
- [End of If structure.]
6. If a right parenthesis is encountered, then:
  - (a) Repeatedly POP from STACK and add to P each operator (On the TOP of STACK.) until a left parenthesis is encountered.
  - (b) Remove the left parenthesis. [Don't add the left parenthesis to P.]
- [End of If Structure.]
- [End of step 2 Loop.]
7. Exit.



# Work Space

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# APPLICATIONS OF STACK

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Towers of Hanoi

## □ TOWER (N, BEG, AUX, END)

1. If  $N = 1$ , Then:

- (a) Write: BEG → END.
- (b) Return.

[End of If Structure.]

2. Call TOWER (N-1, BEG, END, AUX).

[Move N-1 disks from peg BEG to peg AUX.]

3. Write BEG → END.

4. Call TOWER (N-1, AUX, BEG, END).

[Move N-1 disks from peg AUX to peg END.]

5. Return.



# Review Questions

- What is the need of Stacks?
- How Array representation of Stack is different from Linked Representation?
- How will you handle Overflow and Underflow?



# Review Questions

□ Convert the following infix expression to Post-fix:

- $12 / (7-3) + 2 * (1+5)$
- $(5^3) / (3+9) - (5*0) + 2$



Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)

# Data Structures

## Lecture 11: Stacks (Polish Notation)

By

Ravi Kant Sahu  
Asst. Professor



**Lovely Professional University, Punjab**



# Outlines

- Arithmetic Expressions
- Polish Notation
- Evaluation of a Postfix Expression
- Transforming Infix expression to Postfix Expression
- Review Questions



# Arithmetic Expressions

- Arithmetic Expressions involve *constants* and *operations*.
- Binary operations have different levels of precedence.
  - First : Exponentiation (^)
  - Second: Multiplication (\*) and Division (/)
  - Third : Addition (+) and Subtraction (-)



# Example

- Evaluate the following Arithmetic Expression:

$$5^2 + 3 * 5 - 6 * 2 / 3 + 24 / 3 + 3$$

- First:

$$25 + 3 * 5 - 6 * 2 / 3 + 24 / 3 + 3$$

- Second:

$$25 + 15 - 4 + 8 + 3$$

- Third:

47



# Polish Notation

- **Infix Notation:** Operator symbol is placed between the two operands.

Example:

$$(5 * 3) + 2 \quad & \quad 5 * (3 + 2)$$



# Polish Notation

- Polish Notation: The Operator Symbol is placed before its two operands.

Example: + A B, \* C D, / P Q etc.

- Named after the Polish Mathematician Jan Lukasiewicz.
- The order in which the operations are to be performed is completely determined by the positions of operators and operands in the expression.



# Examples

- $(A + B) * C =$   
 $* + ABC$
- $A + (B * C) =$   
 $+ A * BC$
- $(A + B) / (C - D) =$   
 $/ +AB -CD$
- Also Known as Prefix Notation.



# Reverse Polish Notation

- Reverse Polish Notation: The Operator Symbol is placed after its two operands.

Example: A B+, C D\*, P Q/ etc.

- Also known as Postfix Notation.



# Evaluation of Postfix Expression

- P is an arithmetic expression in Postfix Notation.
1. Add a right parenthesis ")" at the end of P.
  2. Scan P from left to right and Repeat Step 3 and 4 for each element of P until the sentinel ")" is encountered.
  3. If an operand is encountered, put it on STACK.
  4. If an operator @ is encountered, then:
    - (A) Remove the two top elements of STACK, where A is the top element and B is the next to top element.
    - (B) Evaluate B @ A.
    - (C) Place the result of (B) back on STACK.

[End of if structure.]

[End of step 2 Loop.]
  5. Set VALUE equal to the top element on STACK.
  6. Exit.



# Review Questions

- Evaluate the following Post-fix Expression:
  - 12 7 3 - / 2 1 5 + \* +



# Infix to Postfix Transformation

## POLISH (Q, P)

1. PUSH "(" on to STACK and add ")" to the end of Q.
2. Scan Q from left to right and Repeat steps 3 to 6 for each element of Q until the STACK is empty:
  3. If an operand is encountered, add it to P.
  4. If a left parenthesis is encountered, push it onto STACK.
  5. If an operator is encountered, then:
    - (a) Repeatedly POP from STACK and add to P each operator (On the TOP of STACK) which has the same precedence as or higher precedence than @.
    - (b) Add @ to STACK.  
[End of If structure.]
  6. If a right parenthesis is encountered, then:
    - (a) Repeatedly POP from STACK and add to P each operator (On the TOP of STACK.) until a left parenthesis is encountered.
    - (b) Remove the left parenthesis. [Don't add the left parenthesis to P.]  
[End of If Structure.]
  7. Exit.



# Work Space

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Review Questions

- Convert the following infix expression to Post-fix and then evaluate that postfix expression using Stack:
  - $12 / (7-3) + 2 * (1+5)$
  - $(5^3) / (3+9) - (5*0) + 2$



Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Review Questions

- What is the need of Stacks?
- How Array representation of Stack is different from Linked Representation?
- How will you handle Overflow and Underflow?

# Data Structures

## Lecture 13: Queue

By

Ravi Kant Sahu  
Asst. Professor



**Lovely Professional University, Punjab**



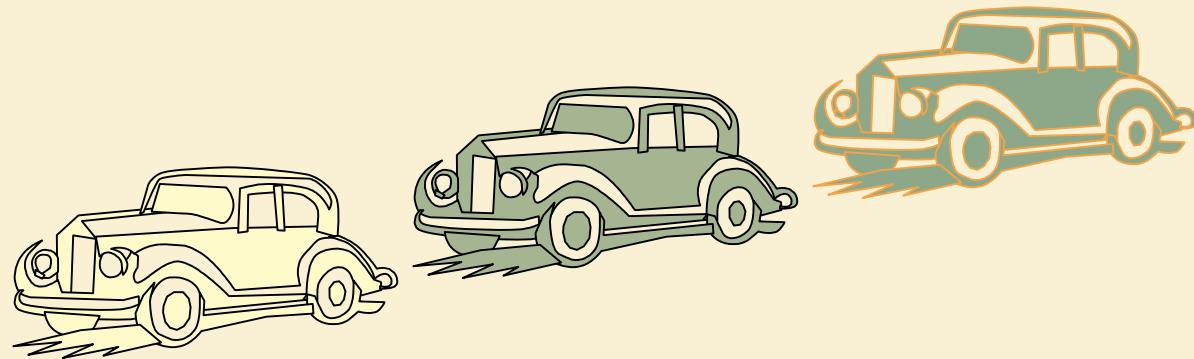
# Outlines

- Introduction
- Examples of Queues
- Application of Queues
- Basic Operations on Queue
- Representation of Queues
  - Array Representation
  - Linked Representation
- Priority Queues
- Review Questions



# Introduction

- A Queue is a linear list of elements in which deletions can take place only at one end, called the ‘FRONT’ and insertions can take place only at the other end, called the ‘REAR’.
  
- Queues are also called as FIFO list.





# ARRAY REPRESENTATION

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Work Space

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Insertion into a Queue

**Q\_INSERT (QUEUE, N, FRONT, REAR, ITEM)**

1. If  $\text{FRONT} = 1$  and  $\text{REAR} = N$ , or If  $\text{FRONT} = \text{REAR}+1$ , then:  
Write: OVERFLOW and Return. [Overflow]
2. If  $\text{FRONT} = 0$ , then [Queue initially empty]  
Set  $\text{FRONT} = 1$  and  $\text{REAR} = 1$ .  
Else if  $\text{REAR} = N$ , then:  
Set  $\text{REAR} = 1$ .  
Else:  
Set  $\text{REAR} = \text{REAR} + 1$ .  
[End of if Structure.]
3. Set  $\text{Queue}[\text{REAR}] = \text{ITEM}$ .
4. Return.



# Deletion in a Queue

**Q\_DELETE (QUEUE, N, FRONT, REAR, ITEM)**

1. If  $\text{FRONT} = 0$ , then:  
    Write: UNDERFLOW and Return.                          [Underflow]
2. Set  $\text{ITEM} = \text{QUEUE} [\text{FRONT}]$ .
3. If  $\text{FRONT} = \text{REAR}$ , then                                  [Queue has only one element]  
    Set  $\text{FRONT} = 0$  and  $\text{REAR} = 0$ .  
  
Else if  $\text{FRONT} = N$ , then:  
    Set  $\text{FRONT} = 1$ .  
Else:  
    Set  $\text{FRONT} = \text{FRONT} + 1$ .  
[End of if Structure.]
4. Return.



# LINKED REPRESENTATION

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Work Space

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Insertion into a Queue

**LINK\_Q\_INSERT (INFO, LINK, FRONT, REAR, AVAIL, ITEM)**

1. [Available space?] If  $AVAIL = \text{NULL}$ , then:  
                Write “OVERFLOW” and Exit.
2. Set  $NEW = AVAIL$ , and  $AVAIL = \text{LINK} [AVAIL]$ .
3. Set  $\text{INFO} [NEW] = ITEM$  and  $\text{Link} [NEW] = \text{NULL}$ .
4. If  $FRONT = \text{NULL}$ , then:  $FRONT = REAR = NEW$ .  
Else: Set  $\text{LINK} [REAR] = NEW$  and  $REAR = NEW$ .
5. Exit



# Deletion in a Queue

**LINK\_Q\_DELETE (INFO, LINK, FRONT, REAR, AVAIL, ITEM)**

1. [Queue Empty?] If  $\text{FRONT} = \text{NULL}$ , then:  
Write “UNDERFLOW” and Exit.
2. Set  $\text{TEMP} = \text{FRONT}$ .
3. Set  $\text{ITEM} = \text{INFO} [\text{TEMP}]$ .
4. IF:  $\text{FRONT} = \text{REAR}$  THEN: Set  $\text{FRONT} = \text{NULL}$  and  $\text{REAR} = \text{NULL}$
5. ELSE: Set  $\text{FRONT} = \text{LINK} [\text{FRONT}]$
6. Set  $\text{LINK} [\text{TEMP}] = \text{AVAIL}$  and  $\text{AVAIL} = \text{TEMP}$ .
7. Exit



# Deques

- Deque (aka Double-ended queue) is a linear list in which elements can be added or removed at either end but not in the middle.
- Deques are represented using circular array. Thus DEQUE[1] comes after DEQUE[N]
- Two variations of Deque are:
  1. Input-restricted deque
  2. Output-restricted deque



# Variations of Deque

## Input-Restricted Deque:

allows insertion only at one end while deletion in both ends of the list.

## Output-Restricted Deque:

allows deletion only at one end while insertion at both the ends of the list.



# Work Space

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Priority Queue

- A Priority Queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are processed comes from the following rules:
  - An element of higher priority is processed before any element of lower priority.
  - Two elements with the same priority are processed according to the order in which they were added to the queue.



# Example

- **Time sharing systems:** Programs of high priority are processed first.



Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)

# Data Structures

## Lecture 15: Priority Queues

By

Ravi Kant Sahu  
Asst. Professor



**Lovely Professional University, Punjab**



# Outlines

- Introduction
- One-Way List Representation of Priority Queues
- Array Representation of Priority Queues



# Introduction

- A Priority Queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are processed comes from the following rules:
  - An element of higher priority is processed before any element of lower priority.
  - Two elements with the same priority are processed according to the order in which they were added to the queue.



# Example

- Time sharing systems: Programs of high priority are processed first.



# ONE-WAY LIST REPRESENTATION OF PRIORITY QUEUES

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# One-Way List Representation

- Each node in the list contains three types of information fields (Information INFO, Priority Number PRN, and a link LINK).
  
- A node X precedes a node Y in the list when:
  - X has higher priority than Y.
  - Both X and Y have same priority but X was added before Y.



# Deletion in a Priority Queue

1. Set ITEM = INFO [START].
2. Delete First node from the list.
3. Process ITEM.
4. Exit.



# Insertion in a Priority Queue

1. Traverse the One-Way list until finding a node X whose priority number exceeds N.
2. Insert ITEM in front of node X.
3. If no such node is found, insert the ITEM as the last element of the list.
4. Exit.



# ARRAY REPRESENTATION OF PRIORITY QUEUES

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Array Representation of Priority Queues

- Use a separate queue for each level of priority (for each priority number).
- Each such queue will appear in its own circular array and have its own pointers FRONT and REAR.

| FRONT | REAR | 1 | 2 | 3 | 4 | 5 |
|-------|------|---|---|---|---|---|
| 2     | 4    |   | X | Y | Z |   |
| 3     | 3    |   |   | P |   |   |
| 0     | 0    |   |   |   |   |   |
| 5     | 2    | A | B |   | D |   |



# Deletion in a Priority Queue

- Delete and process the first element in a priority queue maintained by a two-dimensional array QUEUE.
  1. [Find the first non-empty queue.]  
Find the smallest K such that FRONT [K] ≠ NULL.
  2. Delete and process the front element in row K of QUEUE.
  3. Exit.



# Insertion in a Priority Queue

- Insert an ITEM with priority number M to a priority queue maintained by a two-dimensional array QUEUE.
1. Insert ITEM as the REAR element in row K of QUEUE.
  2. Exit.



Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Review Question

Q. 1: What is the difference between Queue, Deque and Priority Queue?

Q. 2: What is the need of Priority Queue?

Q. 3: How Priority Queues are maintained in memory?

# Data Structures

Lecture 12: Stacks  
(Quick Sort & Towers of Hanoi)

By

Ravi Kant Sahu  
Asst. Professor



Lovely Professional University, Punjab



# Towers of Hanoi

## □ TOWER (N, BEG, AUX, END)

1. If N = 1, Then:
  - (a) Write: BEG → END.
  - (b) Return.  
[End of If Structure.]
2. Call TOWER (N-1, BEG, END, AUX).  
[Move N-1 disks from peg BEG to peg AUX.]
3. Write BEG → END.
4. Call TOWER (N-1, AUX, BEG, END).  
[Move N-1 disks from peg AUX to peg END.]
5. Return.

# Quick Sort

Given an array of  $n$  elements (e.g., integers):

- If array only contains one element, return
- Else
  - pick one element to use as *pivot*.
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quick sort two sub-arrays
  - Return results

# Algorithm

## **QUICK(A, N, BEG, END, LOC)**

1. [Initialize] Set LEFT = BEG, RIGHT=END and LOC=BEG.
2. [Scan from right to left]
  - (a) Repeat while  $A[LOC] \leq A[RIGHT]$  and  $LOC \neq RIGHT$   
 $RIGHT = RIGHT - 1$   
[End of Loop]
  - (b) If  $LOC = RIGHT$ , then: Return.
  - (c) If  $A[LOC] > A[RIGHT]$ , Then
    - (i) [ Interchange  $A[LOC]$  and  $A[RIGHT]$  ]  
Set: TEMP =  $A[LOC]$ ,  $A[LOC] = A[RIGHT]$ ,  $A[RIGHT] = TEMP$ .
    - (ii) Set:  $LOC = RIGHT$ .
    - (iii) Go to Step 3.

3. [Scan from left to right]

(a) Repeat while  $A[LOC] \geq A[LEFT]$  and  $LOC \neq LEFT$

$LEFT = LEFT + 1.$

[End of Loop]

(b) If  $LOC = LEFT$ , then: Return.

(c) If  $A[LOC] < A[LEFT]$ , Then

(i) [ Interchange  $A[LOC]$  and  $A[LEFT]$  ]

Set:  $TEMP = A[LOC]$ ,  $A[LOC] = A[LEFT]$ ,  $A[LEFT] = TEMP$ .

(ii) Set:  $LOC = LEFT$ .

(iii) Go to Step 2.

[End of IF structure]

# Quick Sort Algorithm

## QUICK SORT(A, N, LOWER, UPPER, TOP)

1. [Initialize] TOP = NULL.
2. [PUSH boundary values of A on to Stack when N>1 elements]  
If  $N > 1$ , Then:  $TOP = TOP + 1$ ,  $LOWER[1] = 1$ ,  $UPPER[1] = N$ .
3. Repeat Step 4 to 7 while  $TOP \neq NULL$ .
4. [POP sublist from Stack]  
Set:  $BEG = LOWER[TOP]$ ,  $END = UPPER[TOP]$ ,  $TOP = TOP-1$ .
5. Call QUICK(A, N, BEG, END, LOC)
6. [PUSH left sublist on to Stack]  
If  $BEG < LOC-1$ , Then:  
 $TOP = TOP+1$ ,  $LOWER[TOP] = BEG$ ,  $UPPER[TOP] = LOC-1$ .
7. [PUSH right sublist on to Stack]  
If  $END > LOC+1$ , Then:  
 $TOP = TOP+1$ ,  $LOWER[TOP] = LOC+1$ ,  $UPPER[TOP] = END$ .
8. EXIT

# Example

We are given array of n integers to sort:

|    |    |    |    |    |    |   |    |     |
|----|----|----|----|----|----|---|----|-----|
| 40 | 20 | 10 | 80 | 60 | 50 | 7 | 30 | 100 |
|----|----|----|----|----|----|---|----|-----|

# Pick Pivot Element

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

|    |    |    |    |    |    |   |    |     |
|----|----|----|----|----|----|---|----|-----|
| 40 | 20 | 10 | 80 | 60 | 50 | 7 | 30 | 100 |
|----|----|----|----|----|----|---|----|-----|

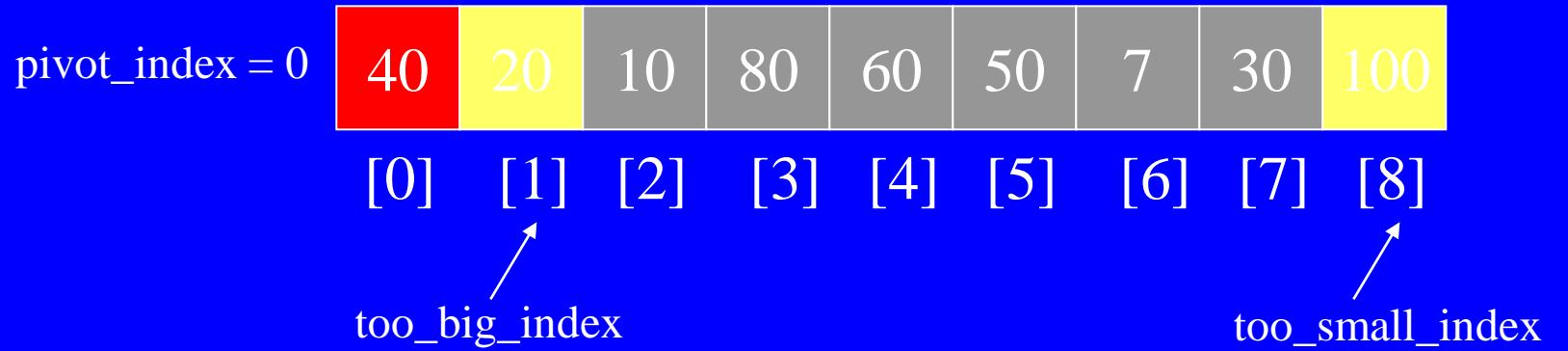
# Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

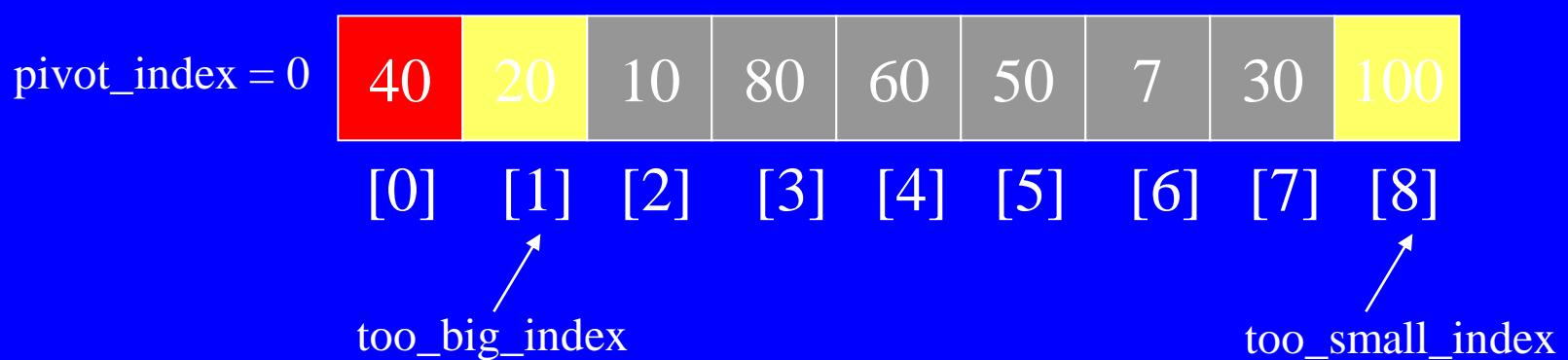
1. One sub-array that contains elements  $\geq$  pivot
2. Another sub-array that contains elements  $<$  pivot

The sub-arrays are stored in the original data array.

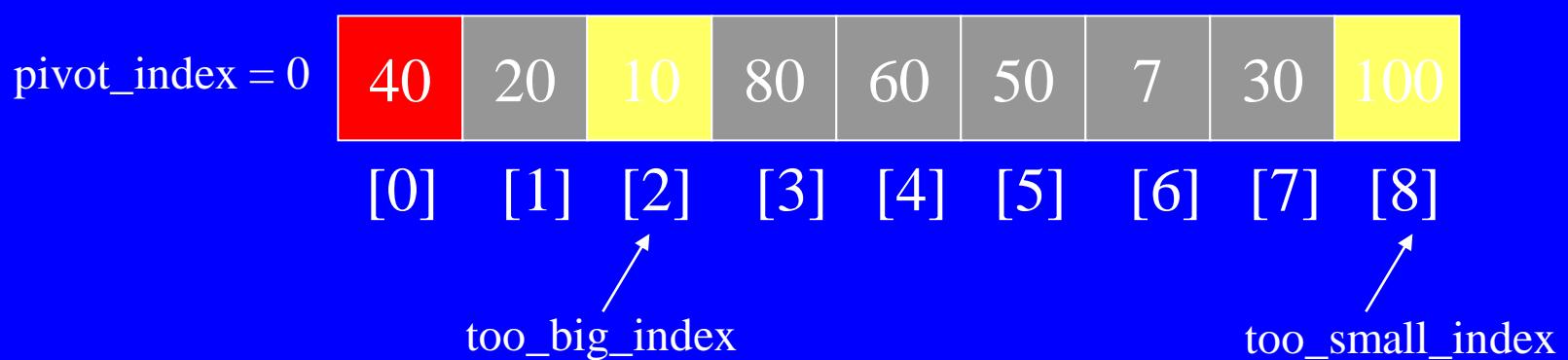
Partitioning loops through, swapping elements below/above pivot.



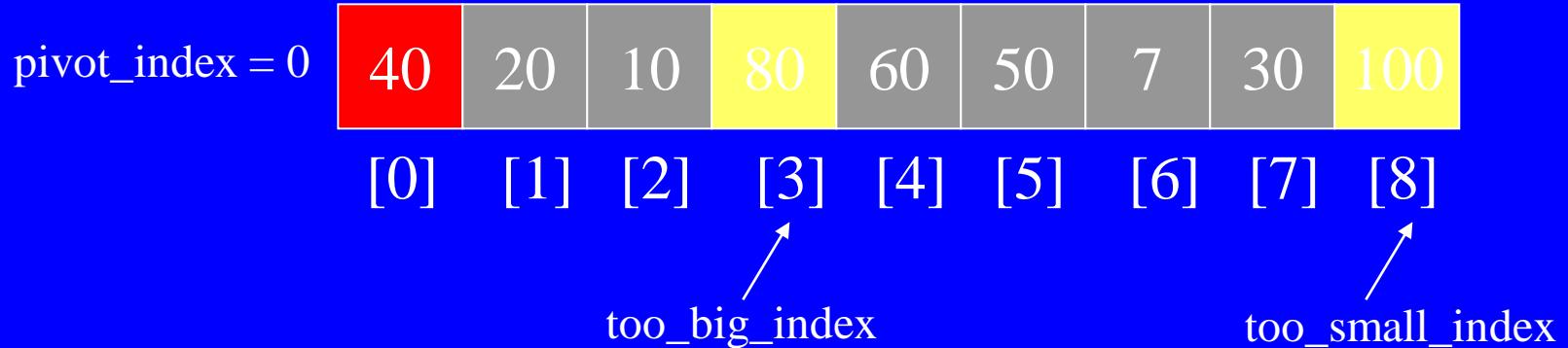
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$



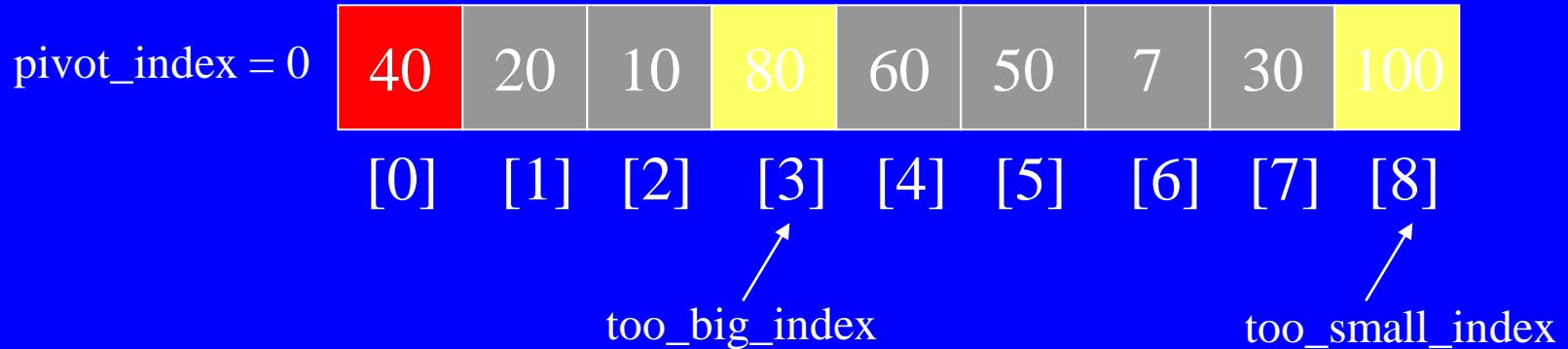
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$



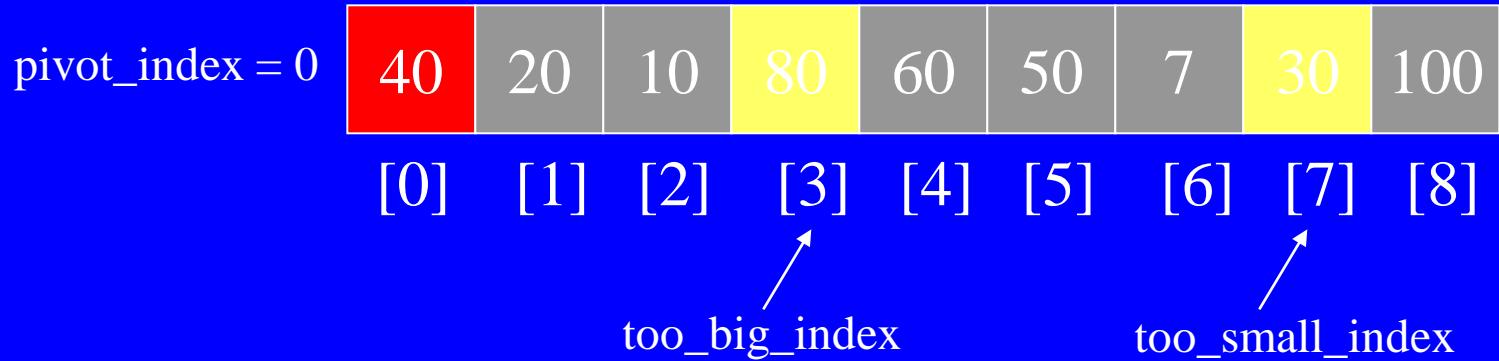
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$



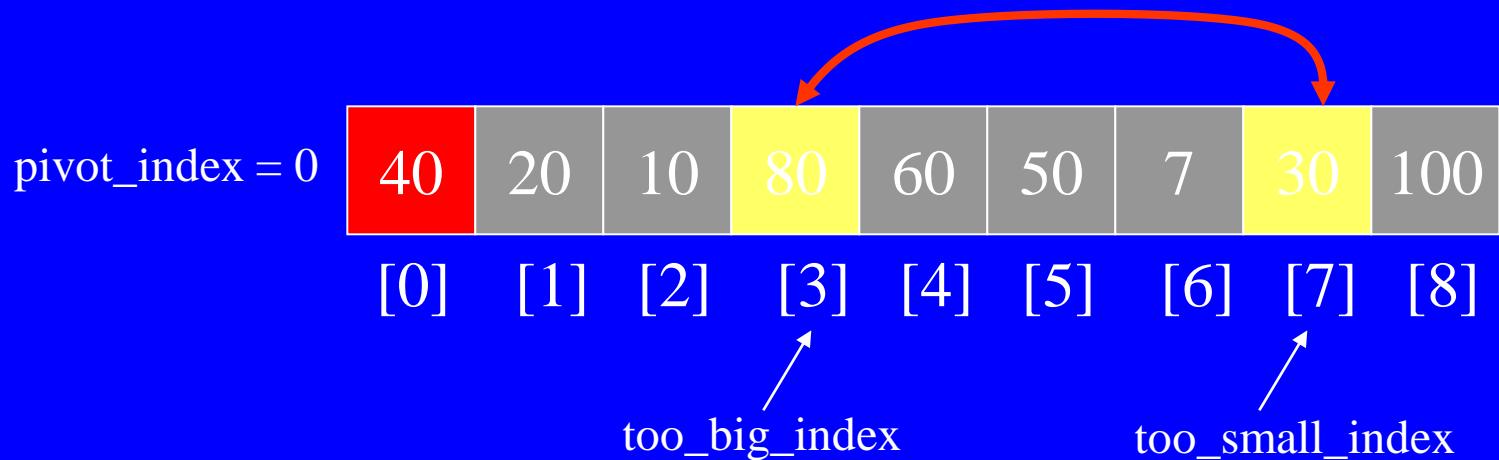
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$



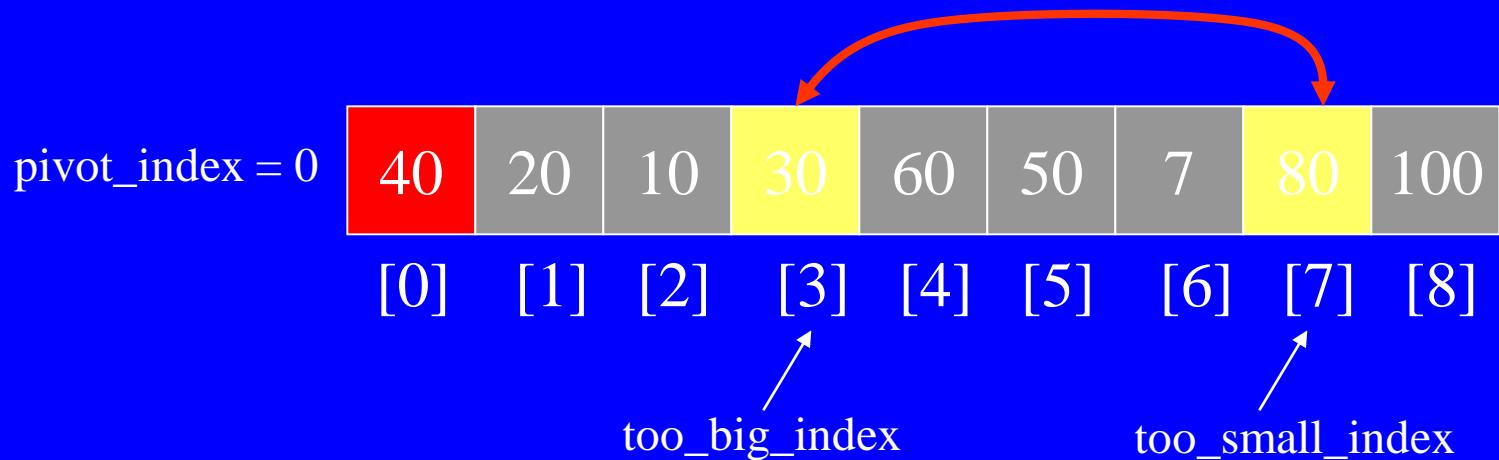
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$



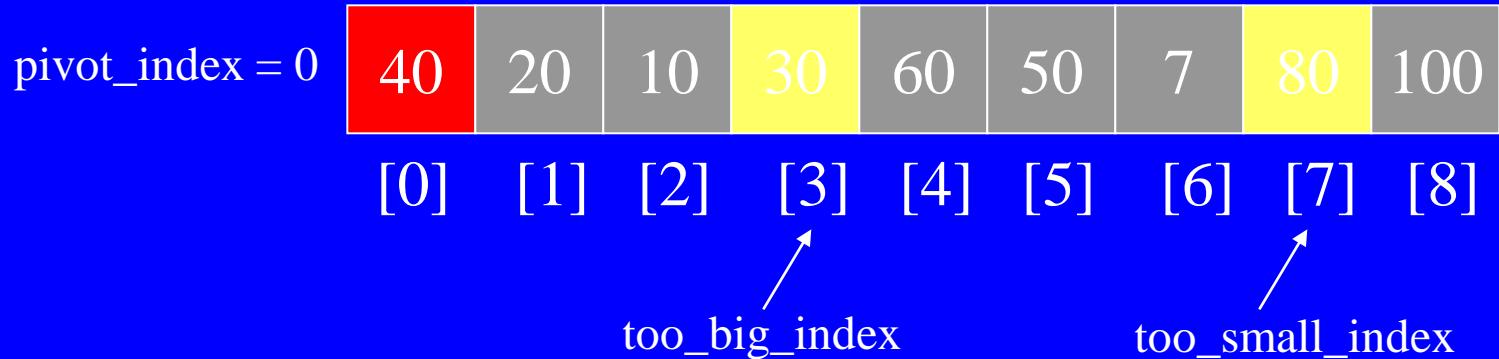
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$



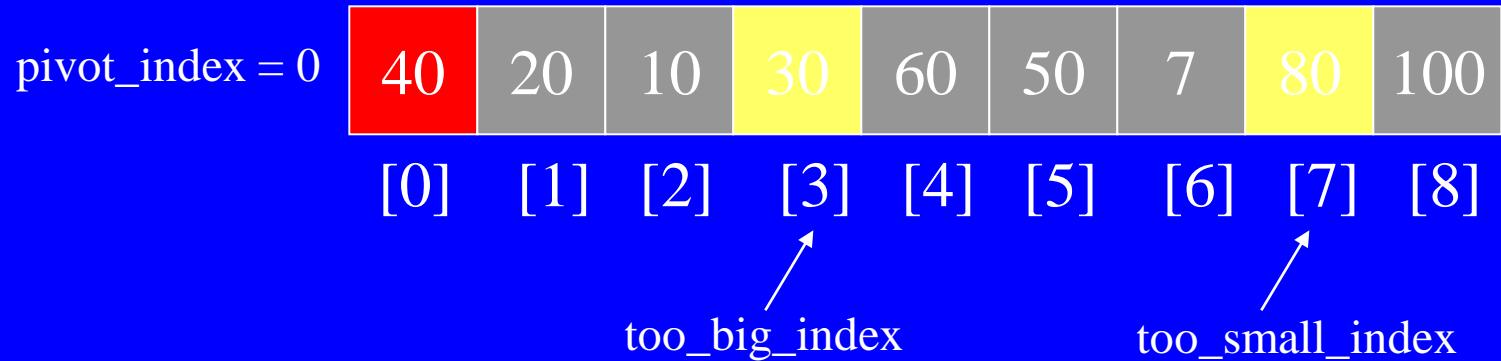
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
        swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$



1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
        swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



- 1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
       $\quad \quad \quad ++\text{too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
       $\quad \quad \quad --\text{too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
      swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.

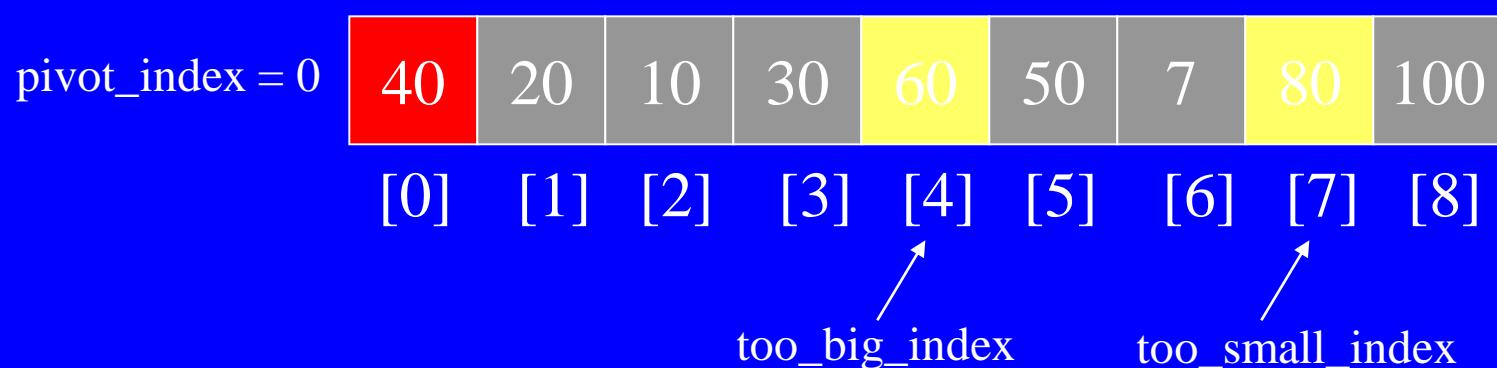


- 1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\quad \quad \quad \text{++too\_big\_index}$

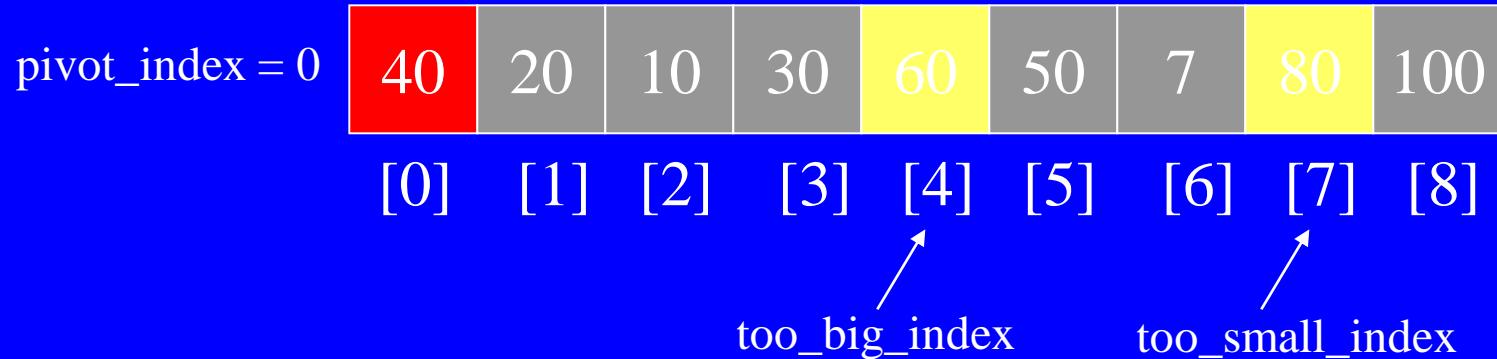
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\quad \quad \quad \text{--too\_small\_index}$

3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
     $\quad \quad \quad \text{swap } \text{data}[\text{too\_big\_index}] \text{ and } \text{data}[\text{too\_small\_index}]$

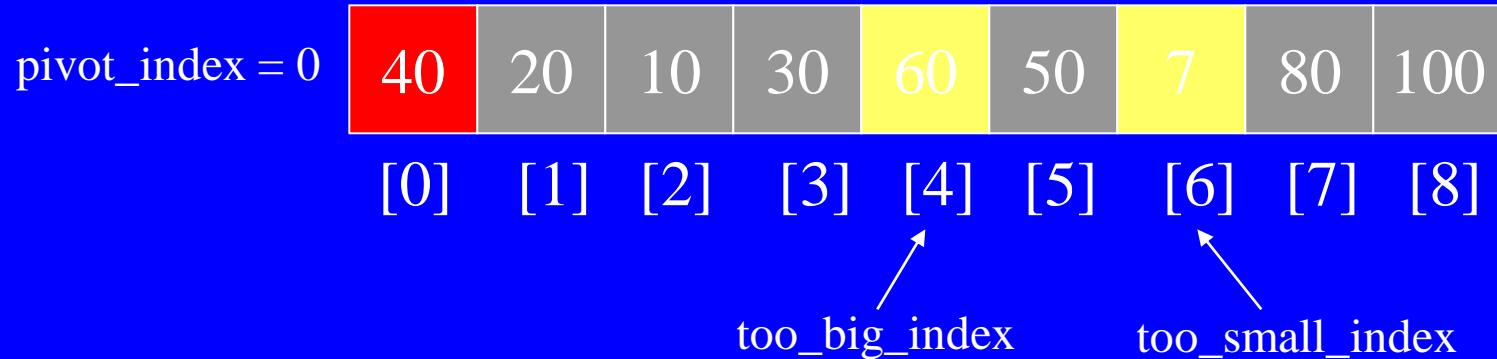
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



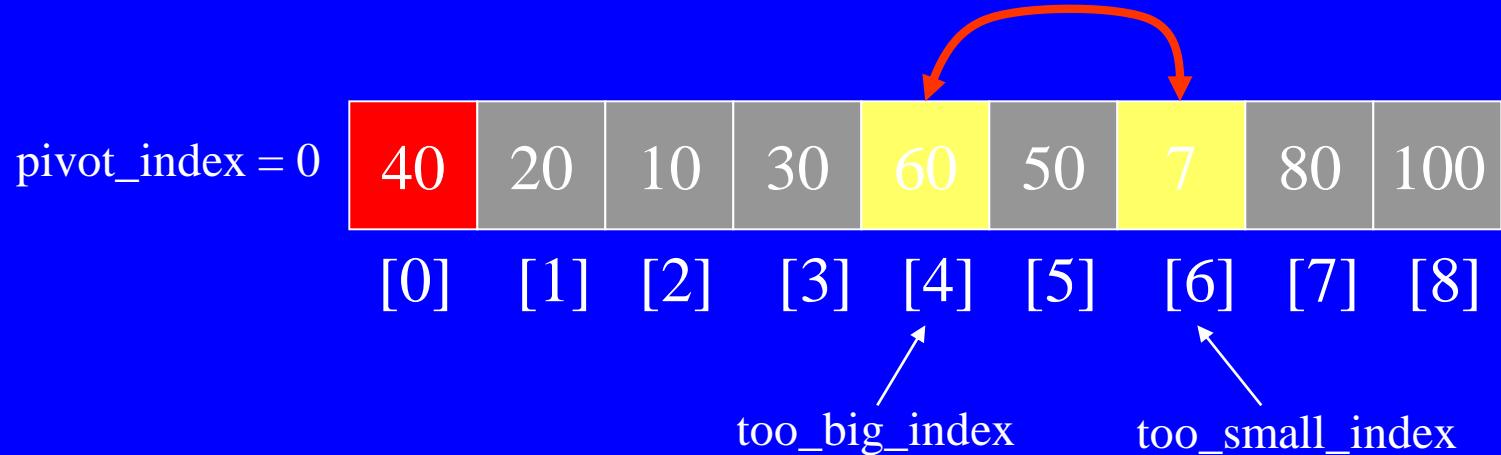
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
- 2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
        swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



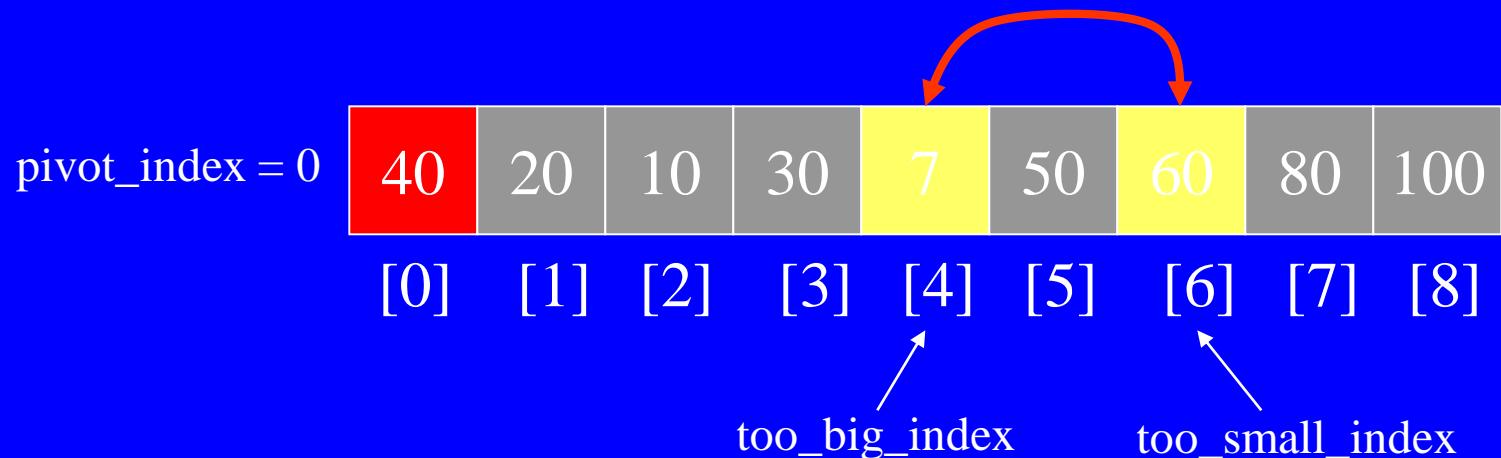
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
- 2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
        swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



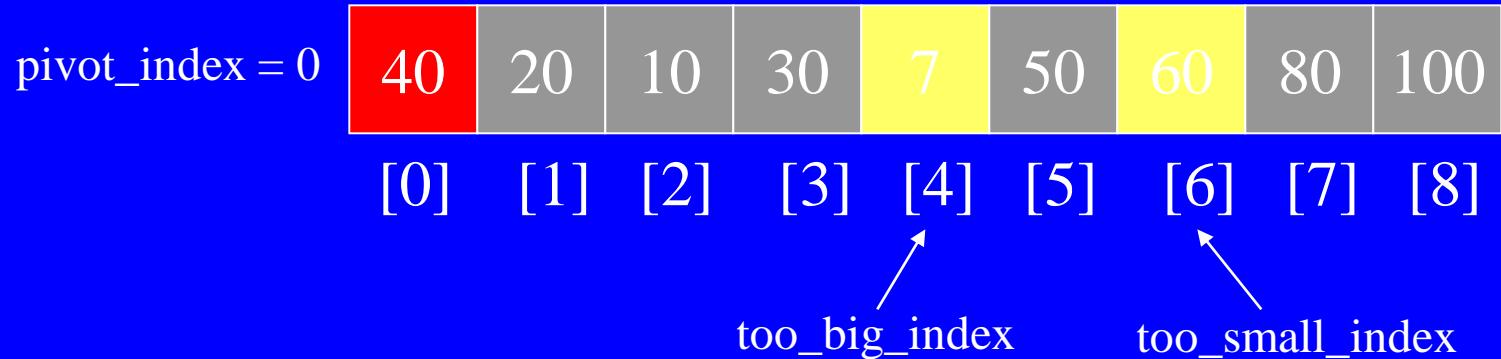
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
- 3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
        swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



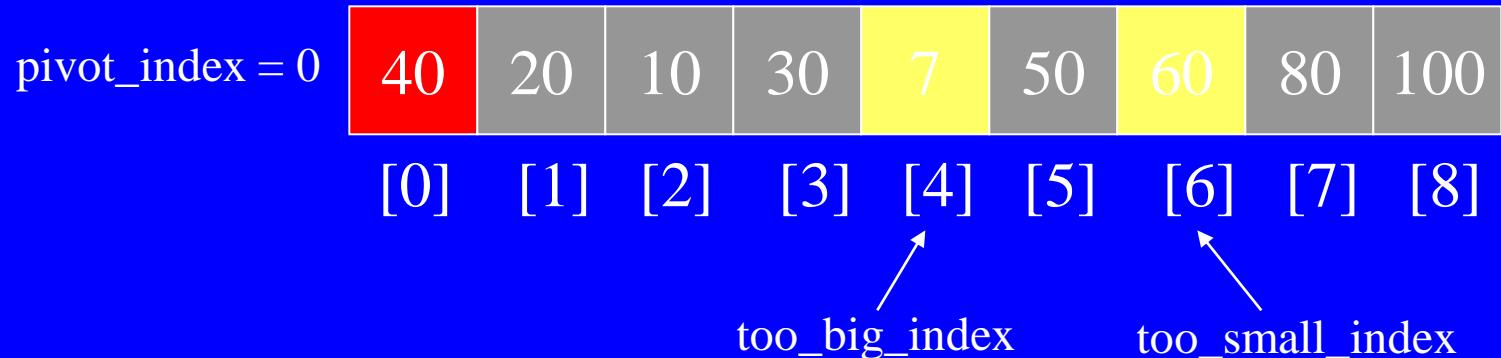
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\quad \quad \quad ++\text{too\_big\_index}$
  2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\quad \quad \quad --\text{too\_small\_index}$
  - 3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
        swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
  4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



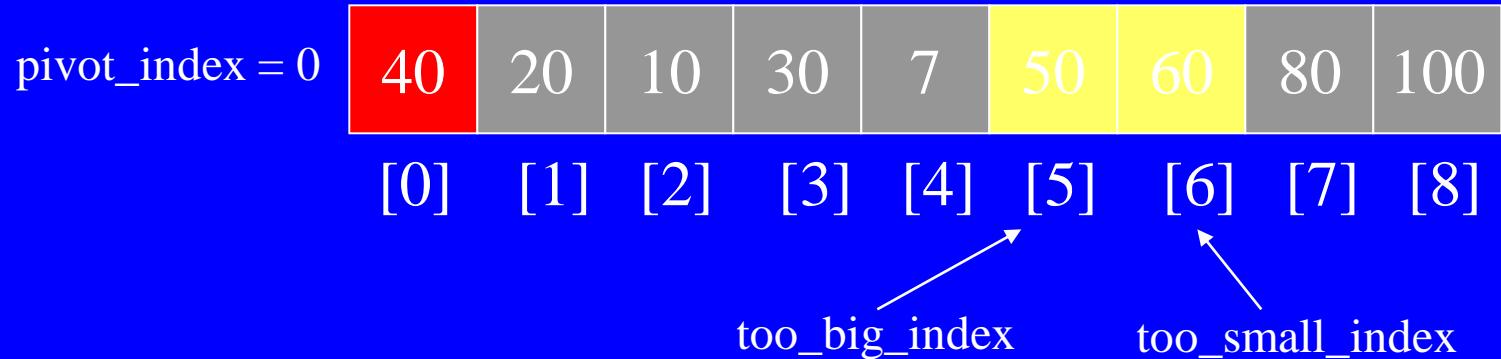
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
        swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
- 4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



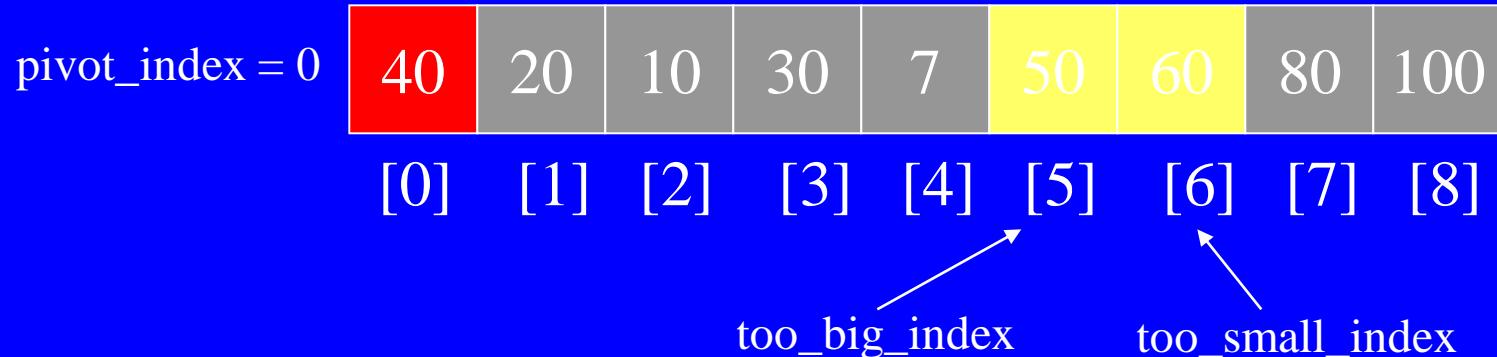
- 1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
 $\quad \quad \quad \quad ++\text{too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
 $\quad \quad \quad \quad --\text{too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
 $\quad \quad \quad \quad \quad \text{swap } \text{data}[\text{too\_big\_index}] \text{ and } \text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



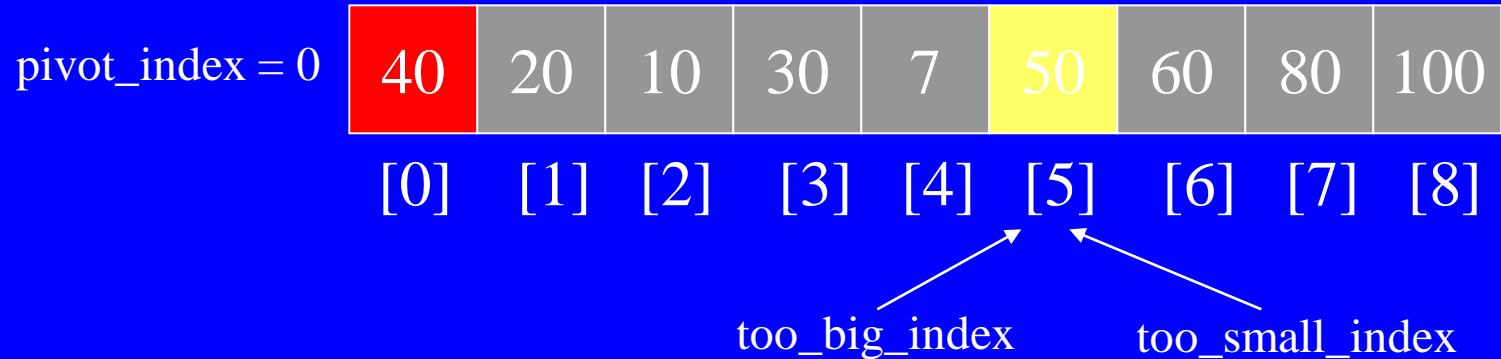
- 1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\quad \quad \quad \text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\quad \quad \quad \text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



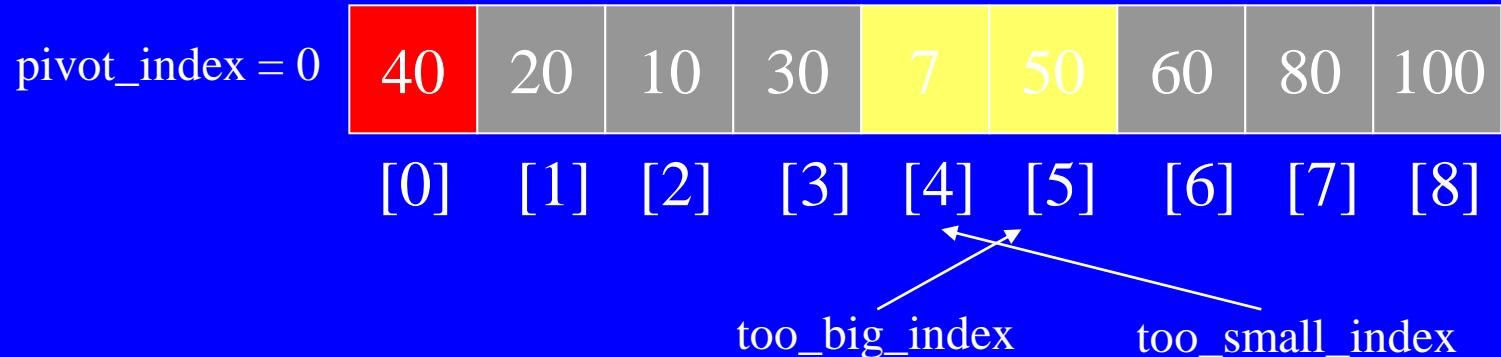
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\quad \quad \quad \text{++too\_big\_index}$
- 2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\quad \quad \quad \text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



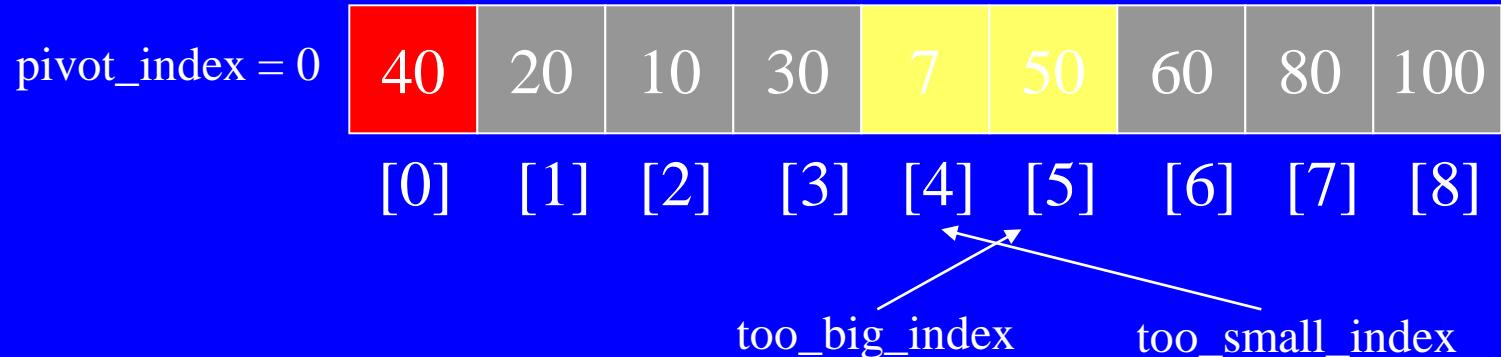
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\quad \quad \quad \text{++too\_big\_index}$
- 2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\quad \quad \quad \text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



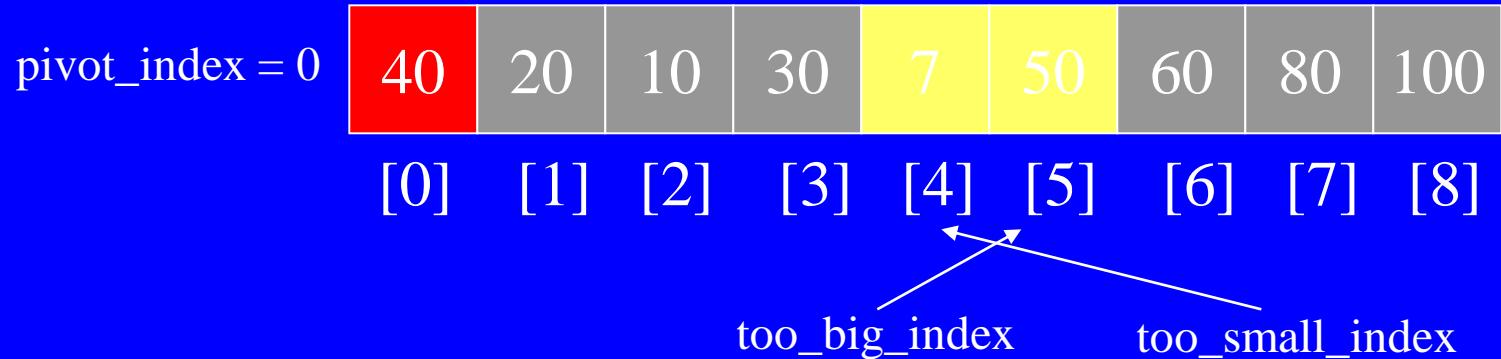
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
 $\quad \quad \quad \text{++too\_big\_index}$
- 2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
 $\quad \quad \quad \text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
 $\quad \quad \quad \text{swap } \text{data}[\text{too\_big\_index}] \text{ and } \text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



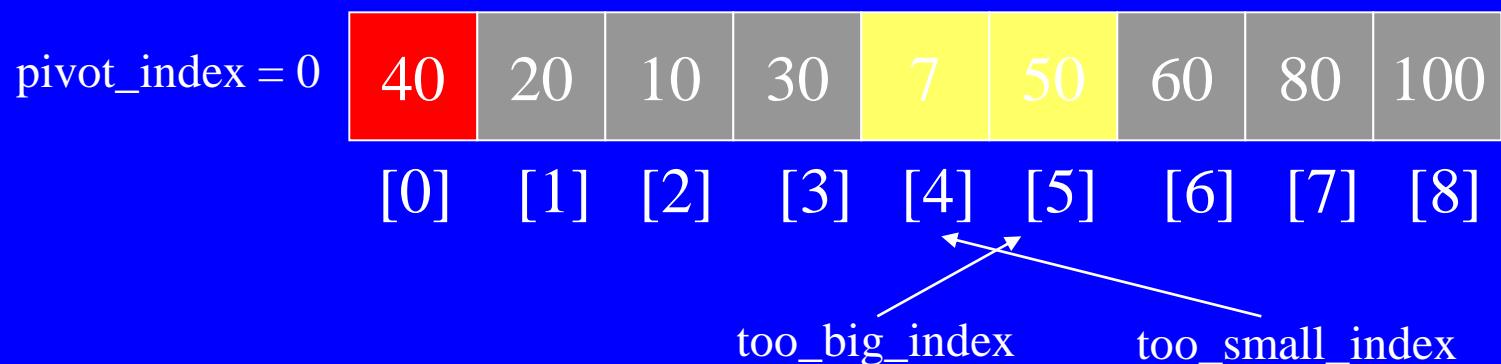
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
 $\quad \quad \quad \text{++too\_big\_index}$
  2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
 $\quad \quad \quad \text{--too\_small\_index}$
- 3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
 swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



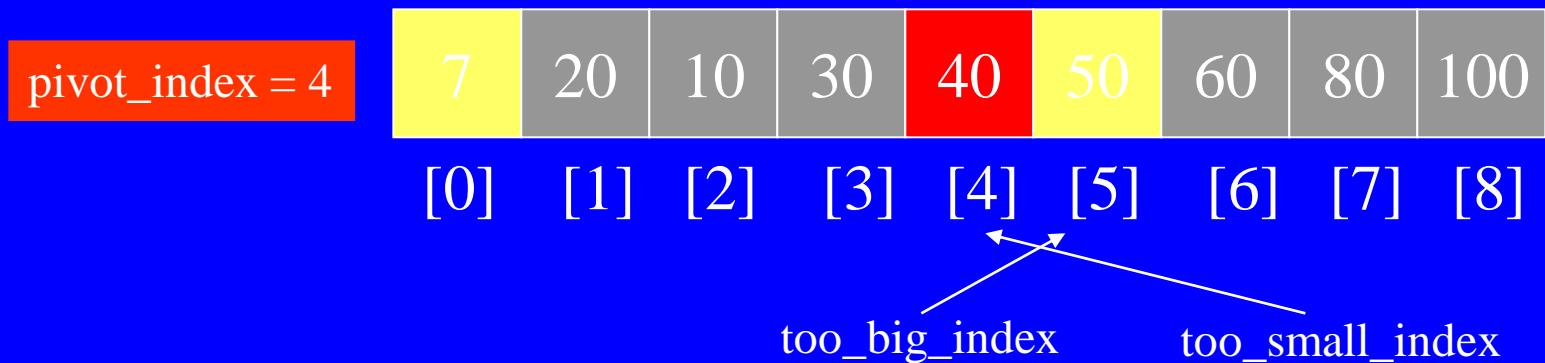
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
        swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
- 4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.



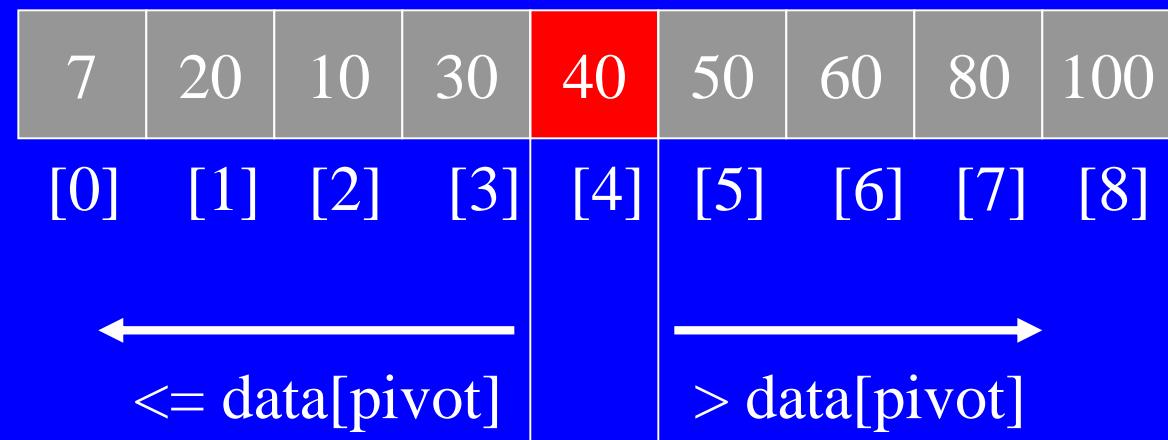
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
        swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
- 5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



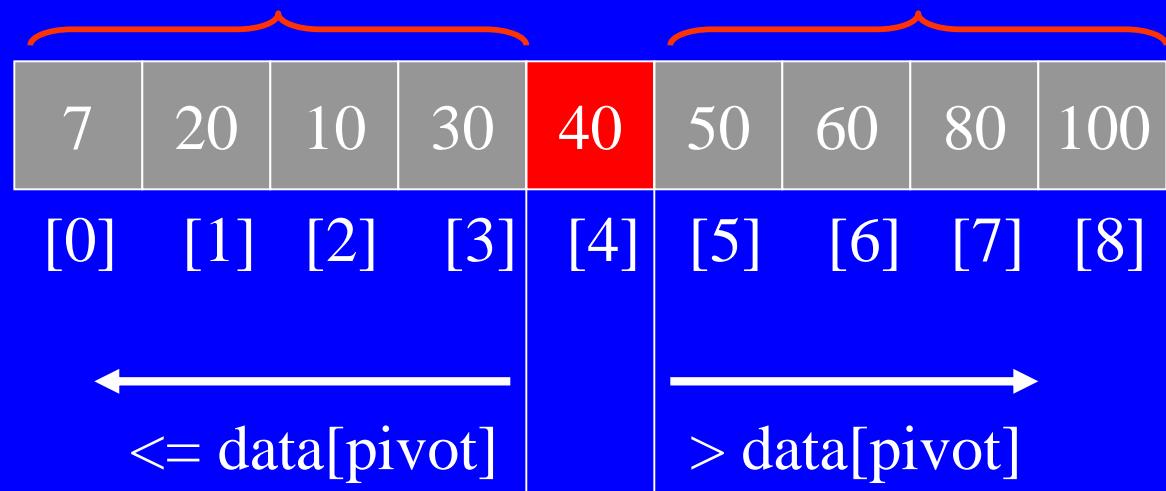
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
        swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
- 5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



# Partition Result



# Recursion: Quicksort Sub-arrays



# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size  $n/2$
    2. Quicksort each sub-array

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size  $n/2$
    2. Quicksort each sub-array
  - Depth of recursion tree?

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size  $n/2$
    2. Quicksort each sub-array
  - Depth of recursion tree?  $O(\log_2 n)$

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size  $n/2$
    2. Quicksort each sub-array
  - Depth of recursion tree?  $O(\log_2 n)$
  - Number of accesses in partition?

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size  $n/2$
    2. Quicksort each sub-array
  - Depth of recursion tree?  $O(\log_2 n)$
  - Number of accesses in partition?  $O(n)$

# Quicksort Analysis

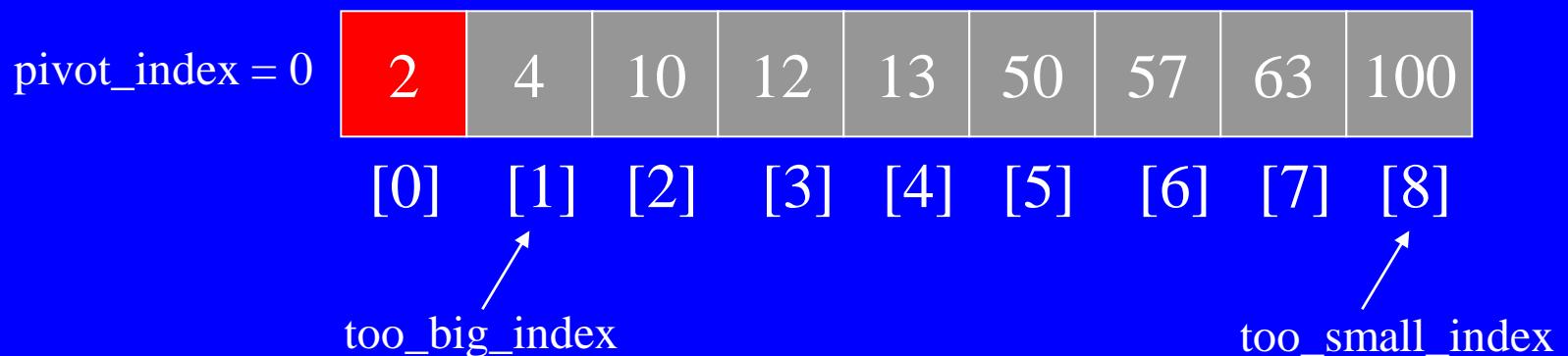
- Assume that keys are random, uniformly distributed.
- Best case running time:  $O(n \log_2 n)$

# Quicksort Analysis

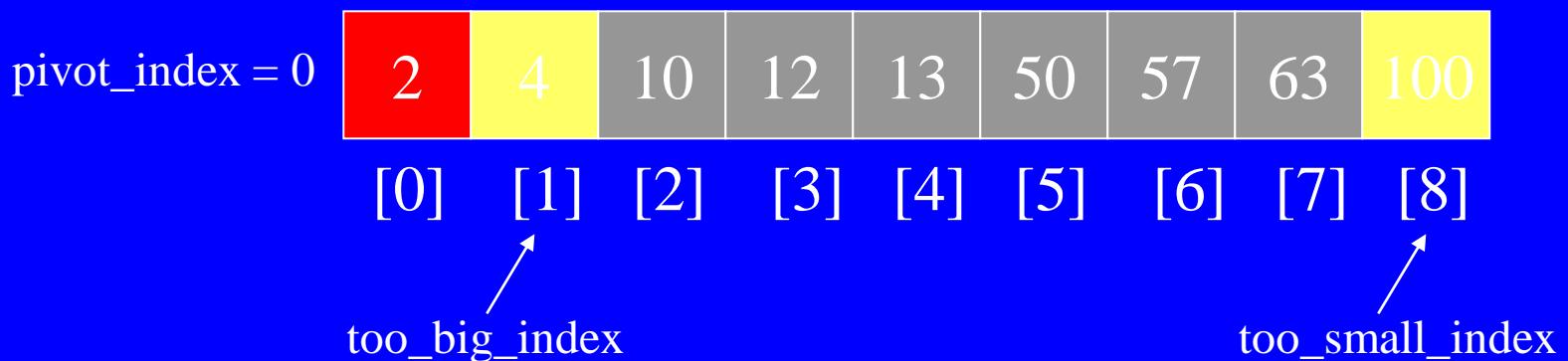
- Assume that keys are random, uniformly distributed.
- Best case running time:  $O(n \log_2 n)$
- Worst case running time?

# Quicksort: Worst Case

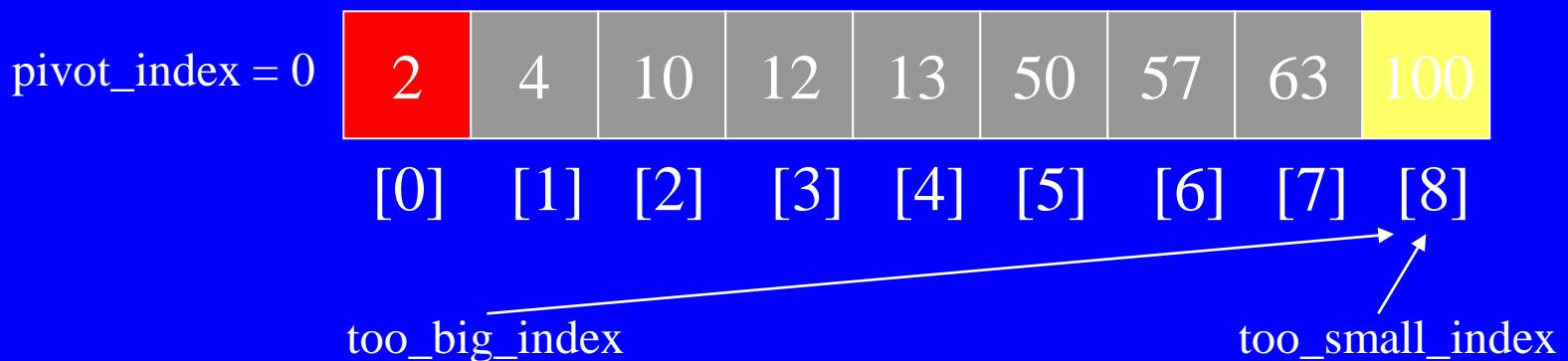
- Assume first element is chosen as pivot.
- Assume we get array that is already in order:



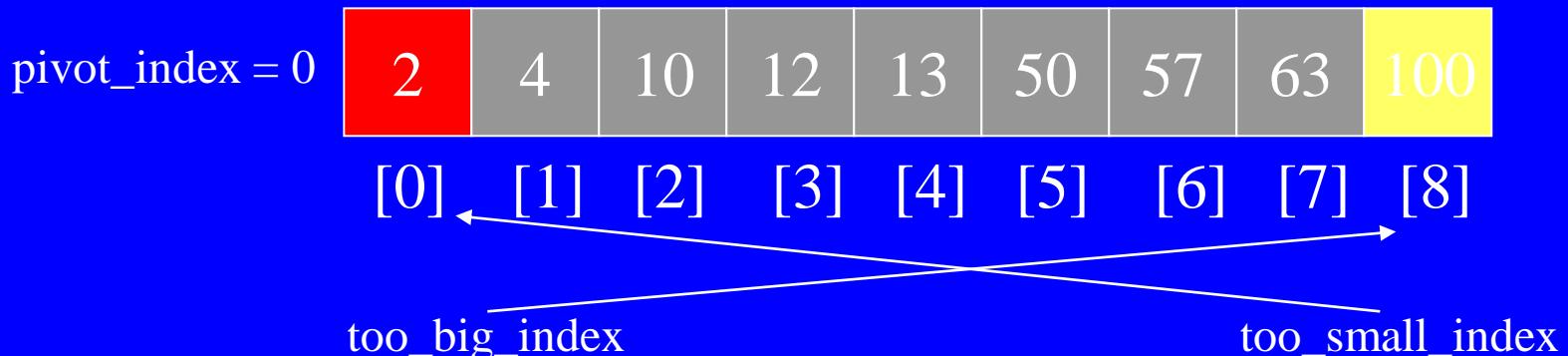
- 1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\quad \quad \quad ++\text{too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\quad \quad \quad --\text{too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
    swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



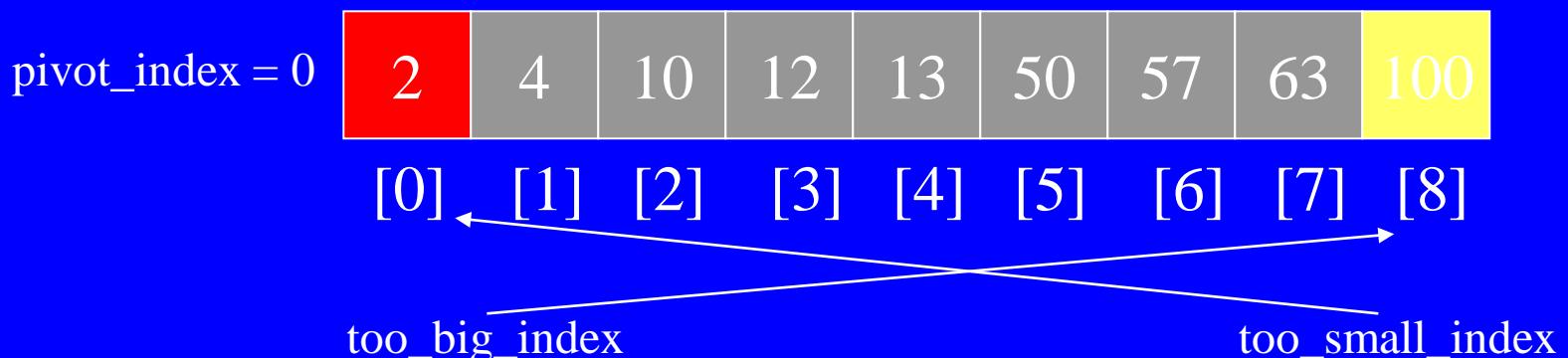
- 1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
 $\quad \quad \quad \text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
 $\quad \quad \quad \text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
 $\quad \quad \quad \text{swap } \text{data}[\text{too\_big\_index}] \text{ and } \text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



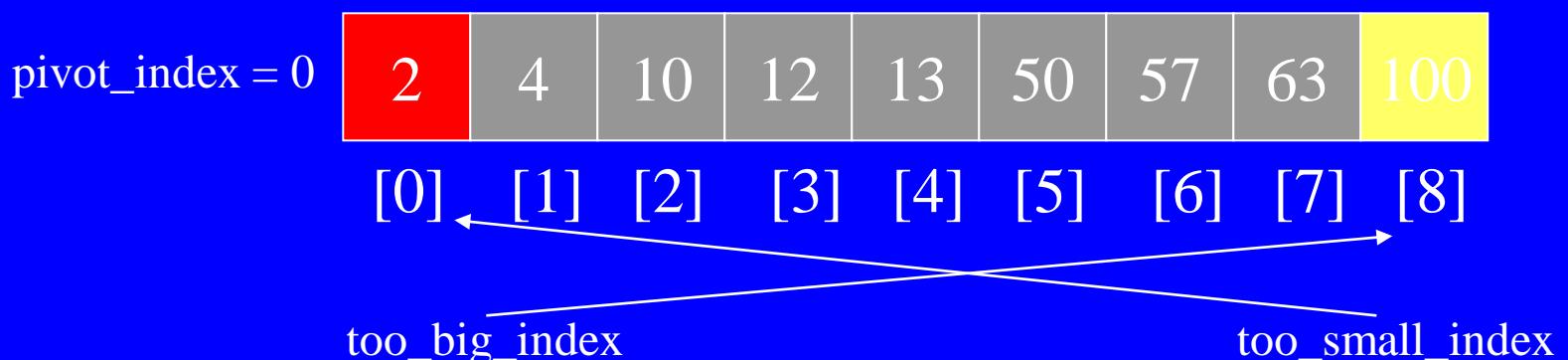
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
- 2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
        swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



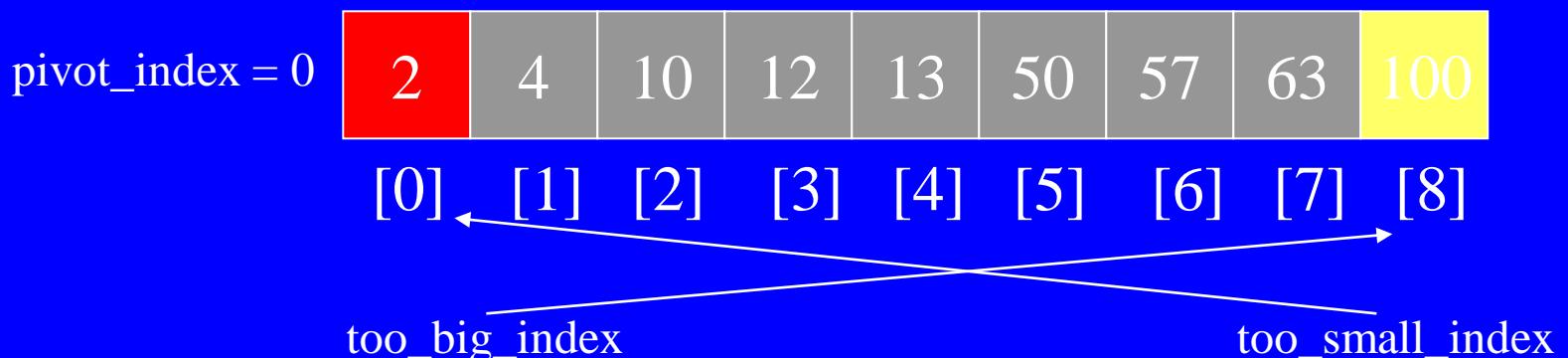
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
  2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
- 3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
     swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
  5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



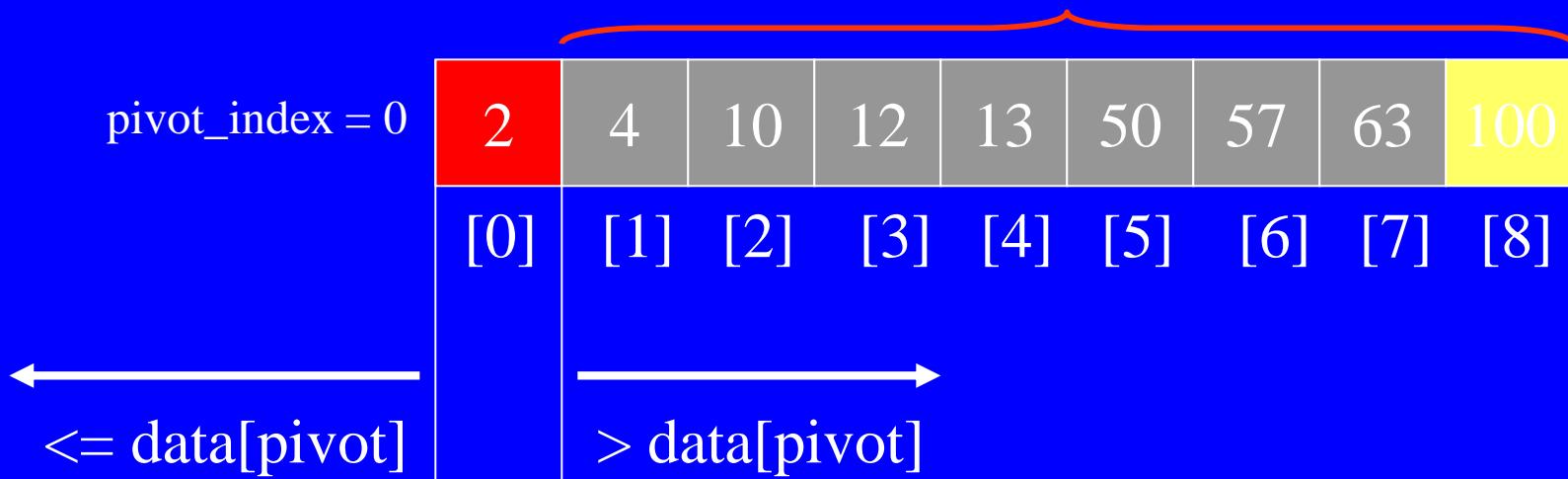
1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
        swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
- 4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
 $\quad \quad \quad \text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
 $\quad \quad \quad \text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
 $\quad \quad \quad \text{swap } \text{data}[\text{too\_big\_index}] \text{ and } \text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
- 5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



1. While  $\text{data}[\text{too\_big\_index}] \leq \text{data}[\text{pivot}]$   
     $\text{++too\_big\_index}$
2. While  $\text{data}[\text{too\_small\_index}] > \text{data}[\text{pivot}]$   
     $\text{--too\_small\_index}$
3. If  $\text{too\_big\_index} < \text{too\_small\_index}$   
        swap  $\text{data}[\text{too\_big\_index}]$  and  $\text{data}[\text{too\_small\_index}]$
4. While  $\text{too\_small\_index} > \text{too\_big\_index}$ , go to 1.
- 5. Swap  $\text{data}[\text{too\_small\_index}]$  and  $\text{data}[\text{pivot\_index}]$



# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time:  $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
      - one sub-array of size 0
      - the other sub-array of size  $n-1$
    2. Quicksort each sub-array
  - Depth of recursion tree?

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time:  $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
      - one sub-array of size 0
      - the other sub-array of size  $n-1$
    2. Quicksort each sub-array
  - Depth of recursion tree?  $O(n)$

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time:  $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
      - one sub-array of size 0
      - the other sub-array of size  $n-1$
    2. Quicksort each sub-array
  - Depth of recursion tree?  $O(n)$
  - Number of accesses per partition?

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time:  $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
      - one sub-array of size 0
      - the other sub-array of size  $n-1$
    2. Quicksort each sub-array
  - Depth of recursion tree?  $O(n)$
  - Number of accesses per partition?  $O(n)$

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time:  $O(n \log_2 n)$
- Worst case running time:  $O(n^2)!!!$

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time:  $O(n \log_2 n)$
- Worst case running time:  $O(n^2)!!!$
- What can we do to avoid worst case?

# Data Structures

## Topic: Merge Sort

By

Ravi Kant Sahu  
Asst. Professor



Lovely Professional University, Punjab

# Divide and Conquer

- ♦ Recursive in structure
  - ♦ *Divide* the problem into sub-problems that are similar to the original but smaller in size
  - ♦ *Conquer* the sub-problems by solving them **recursively**. If they are small enough, just solve them in a straightforward manner.
  - ♦ *Combine* the solutions to create a solution to the original problem

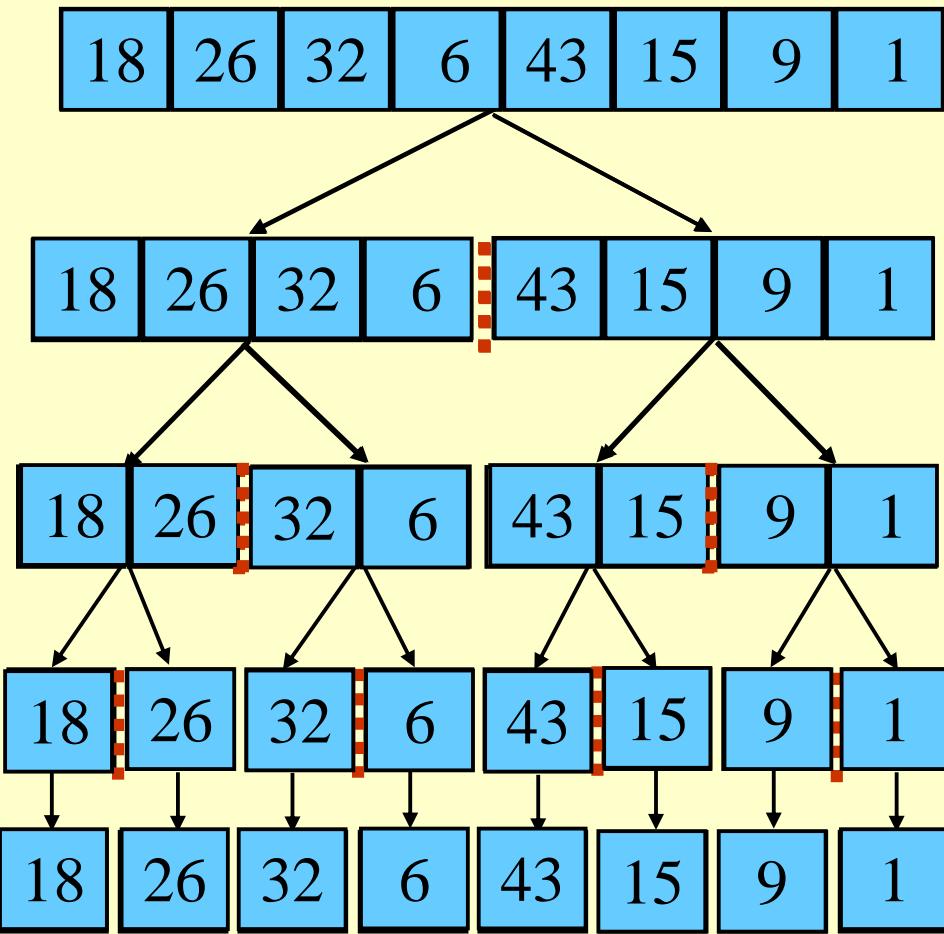
# An Example: Merge Sort

**Sorting Problem:** Sort a sequence of  $n$  elements into non-decreasing order.

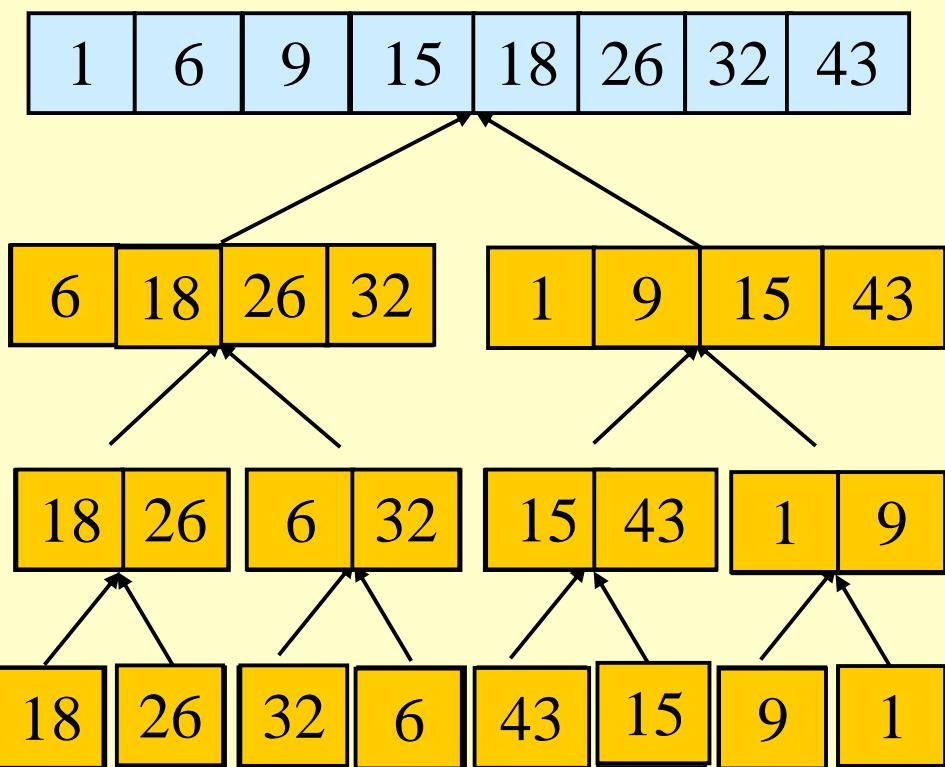
- ◆ ***Divide:*** Divide the  $n$ -element sequence to be sorted into two subsequences of  $n/2$  elements each
- ◆ ***Conquer:*** Sort the two subsequences recursively using merge sort.
- ◆ ***Combine:*** Merge the two sorted subsequences to produce the sorted answer.

# Merge Sort – Example

Original Sequence



Sorted Sequence



# Work Space

# Merge-Sort (A, p, r)

**INPUT:** a sequence of  $n$  numbers stored in array A

**OUTPUT:** an ordered sequence of  $n$  numbers

```
MergeSort (A, p, r) // sort A[p..r] by divide & conquer
1  if  $p < r$ 
2    then  $q \leftarrow \lfloor (p+r)/2 \rfloor$ 
3      MergeSort (A, p, q)
4      MergeSort (A, q+1, r)
5      Merge (A, p, q, r) // merges A[p..q] with A[q+1..r]
```

**Initial Call:** MergeSort(A, 1, n)

# Procedure Merge

**Merge( $A, p, q, r$ )**

```
1   $n_1 \leftarrow q - p + 1$ 
2   $n_2 \leftarrow r - q$ 
3  for  $i \leftarrow 1$  to  $n_1$ 
   do  $L[i] \leftarrow A[p + i - 1]$ 
4
5  for  $j \leftarrow 1$  to  $n_2$ 
   do  $R[j] \leftarrow A[q + j]$ 
7   $L[n_1+1] \leftarrow \infty$ 
8   $R[n_2+1] \leftarrow \infty$ 
9   $i \leftarrow 1$ 
10  $j \leftarrow 1$ 
11 for  $k \leftarrow p$  to  $r$ 
12   do if  $L[i] \leq R[j]$ 
13     then  $A[k] \leftarrow L[i]$ 
14        $i \leftarrow i + 1$ 
15     else  $A[k] \leftarrow R[j]$ 
16        $j \leftarrow j + 1$ 
```

Input: Array containing sorted subarrays  $A[p..q]$  and  $A[q+1..r]$ .

Output: Merged sorted subarray in  $A[p..r]$ .

**Sentinels**, to avoid having to check if either subarray is fully copied at **each step**.

# Work Space

# Merge – Example

A

|     |   |   |   |   |    |    |    |    |     |
|-----|---|---|---|---|----|----|----|----|-----|
| ... | 1 | 6 | 8 | 9 | 26 | 32 | 42 | 43 | ... |
|-----|---|---|---|---|----|----|----|----|-----|

$k$

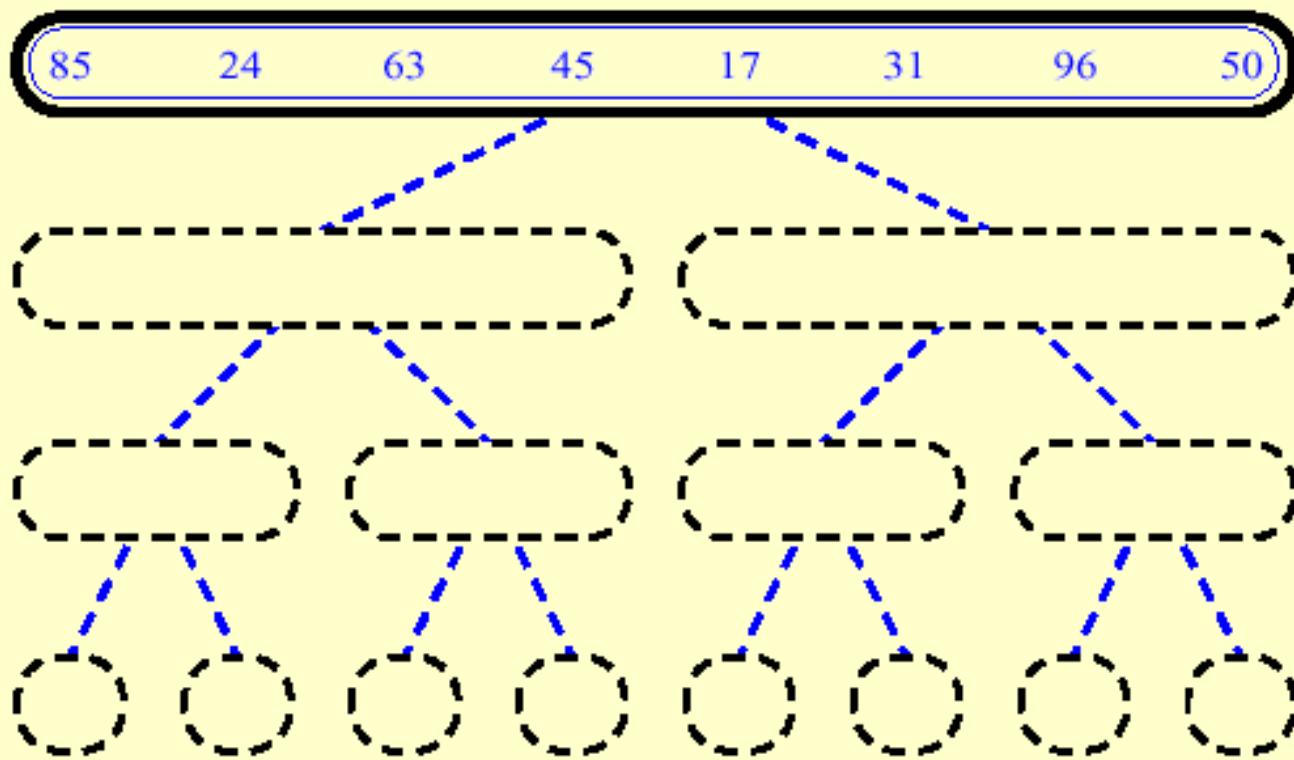
|     |   |   |    |    |          |
|-----|---|---|----|----|----------|
| $L$ | 6 | 8 | 26 | 32 | $\infty$ |
|-----|---|---|----|----|----------|

$i$

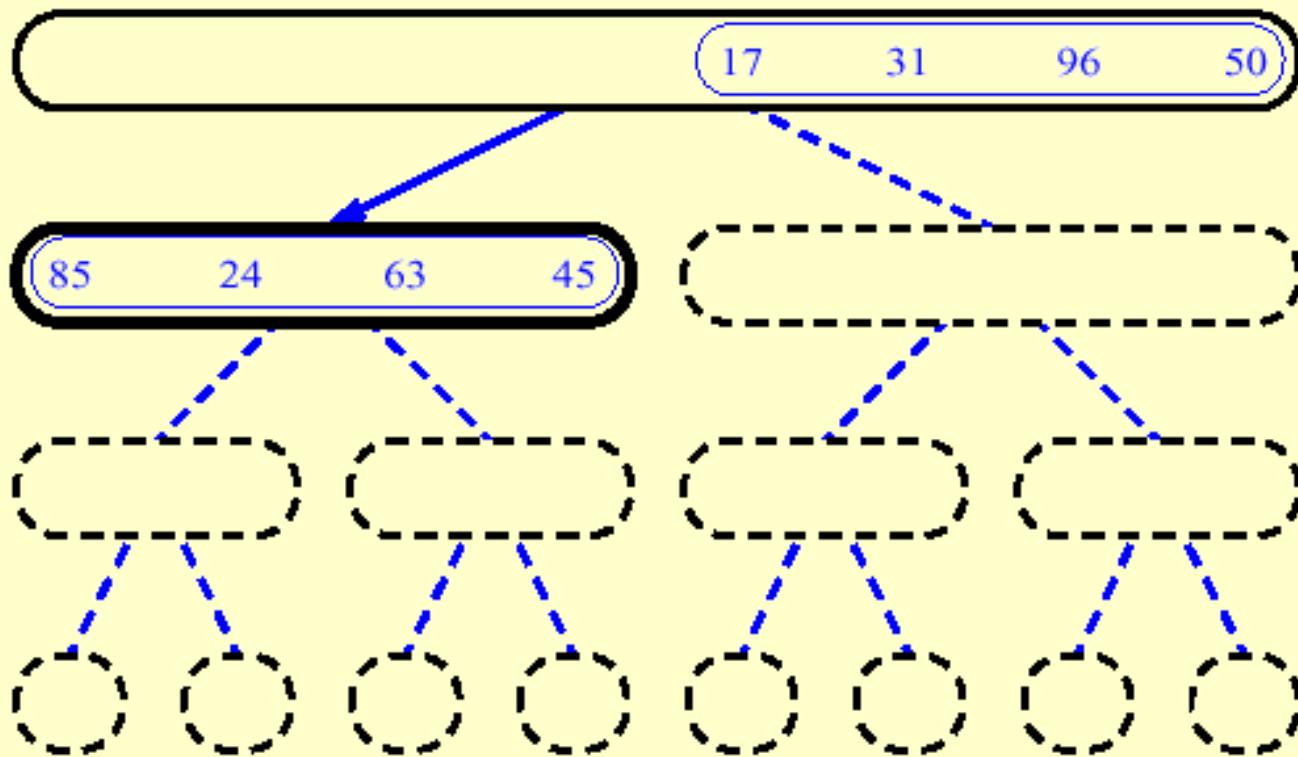
|     |   |   |    |    |          |
|-----|---|---|----|----|----------|
| $R$ | 1 | 9 | 42 | 43 | $\infty$ |
|-----|---|---|----|----|----------|

$j$

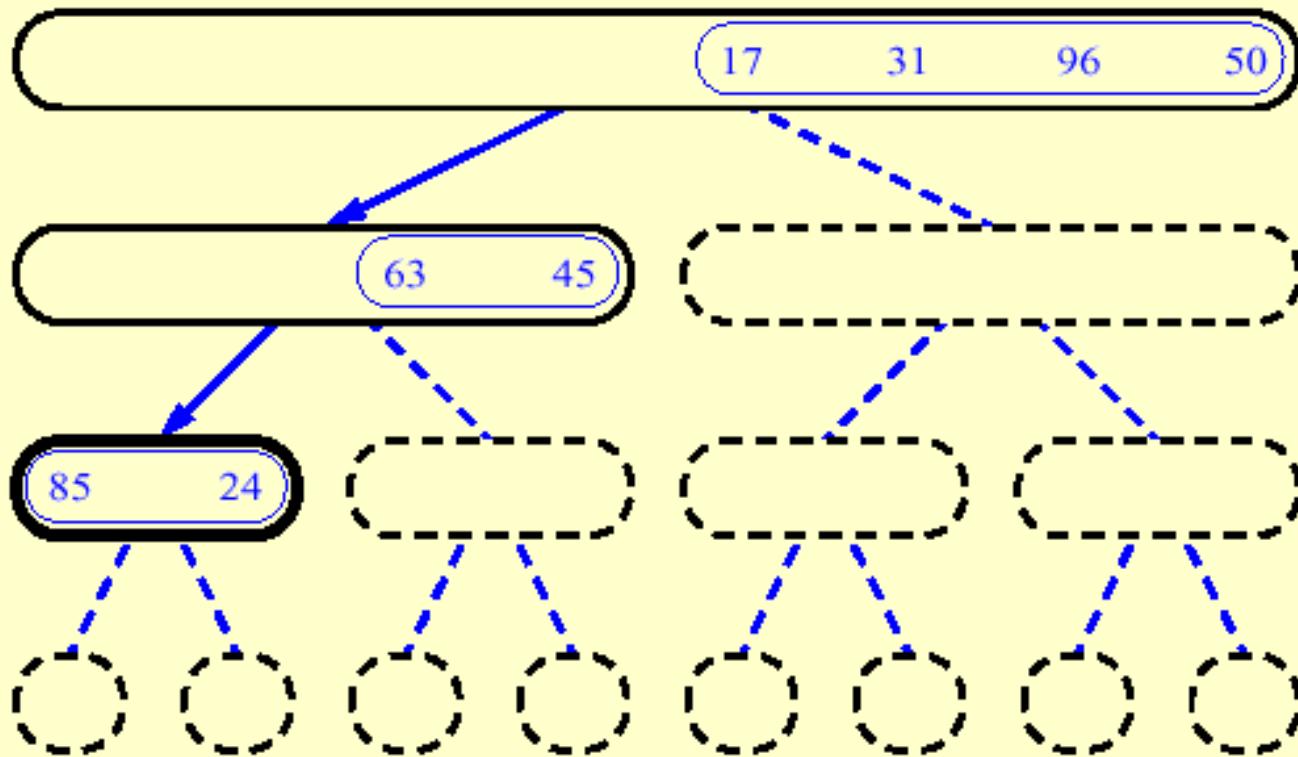
# MergeSort (Example) - 1



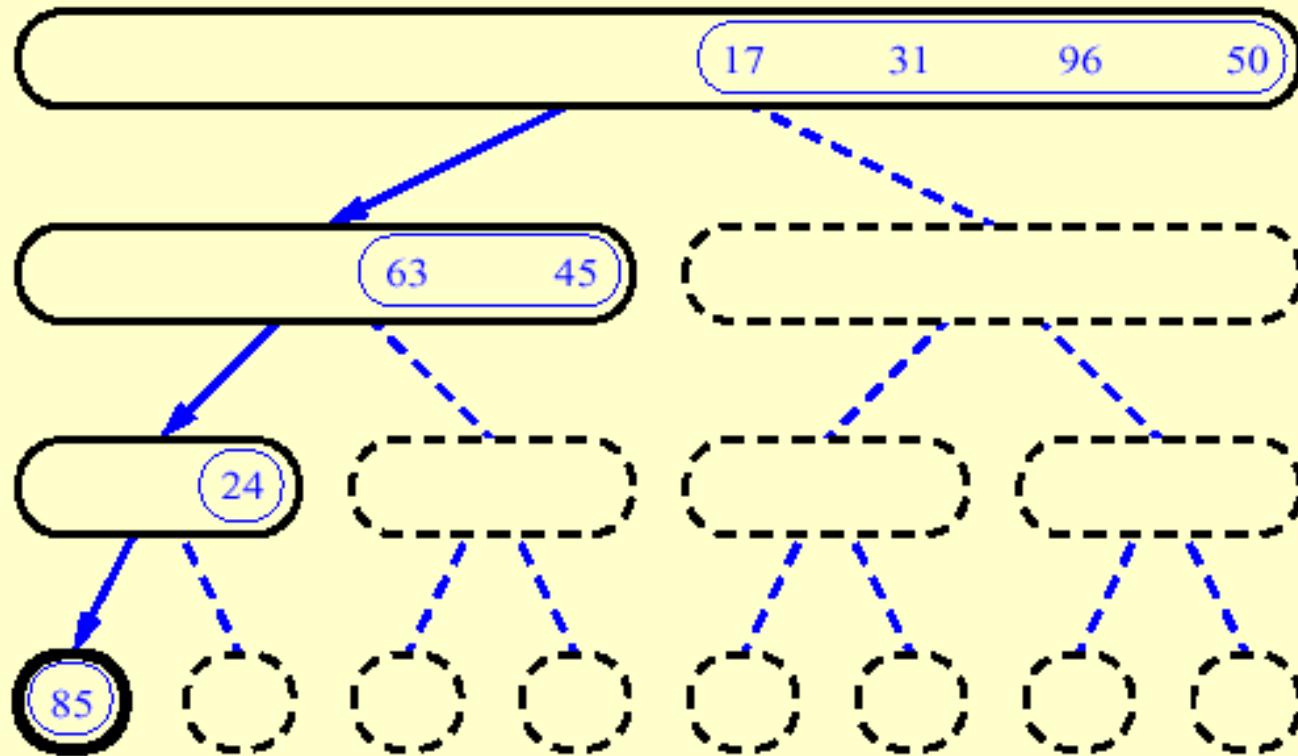
## MergeSort (Example) - 2



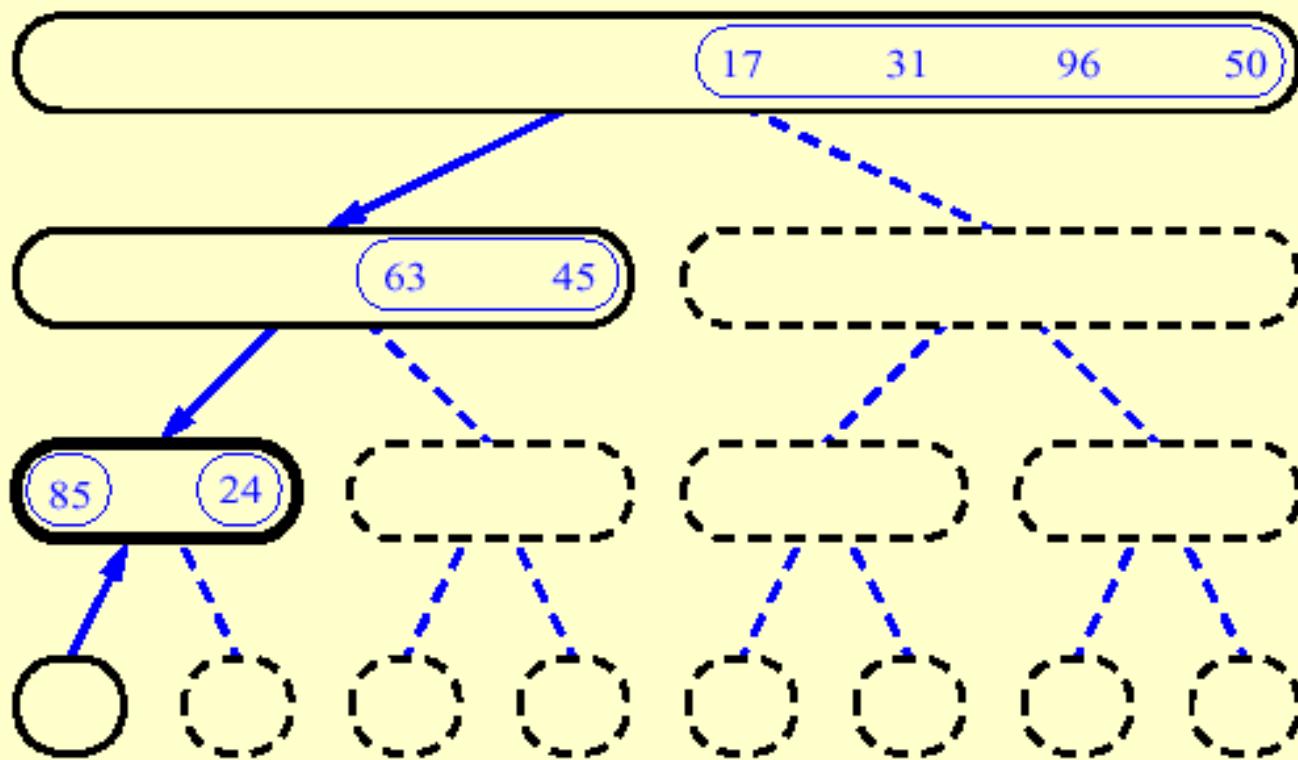
# MergeSort (Example) - 3



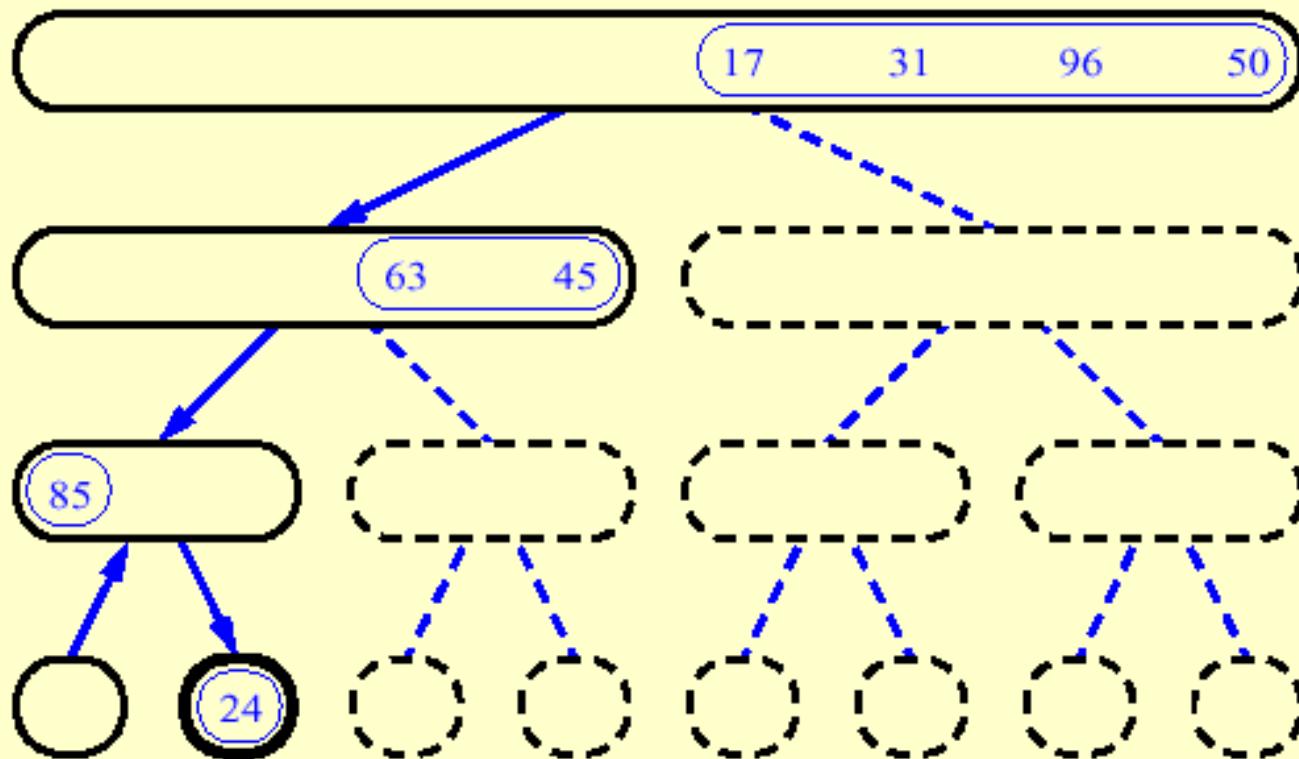
# MergeSort (Example) - 4



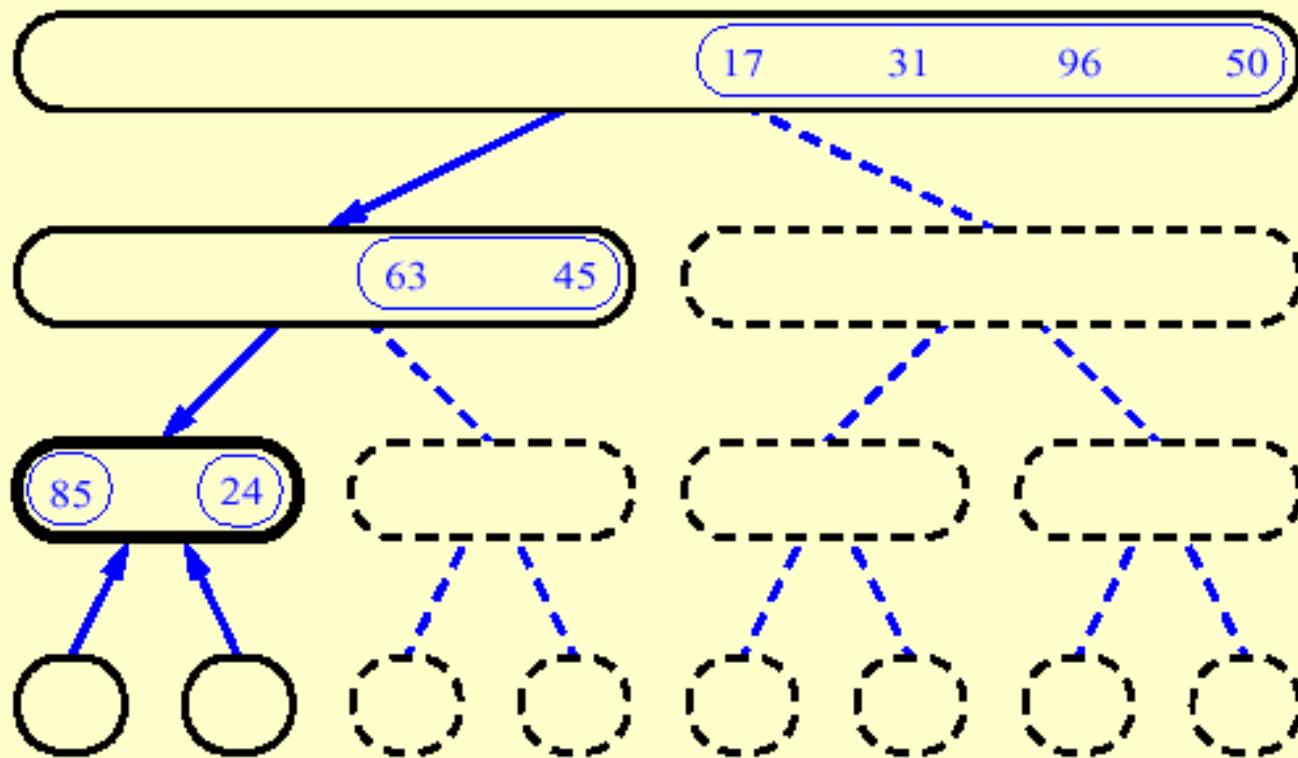
# MergeSort (Example) - 5



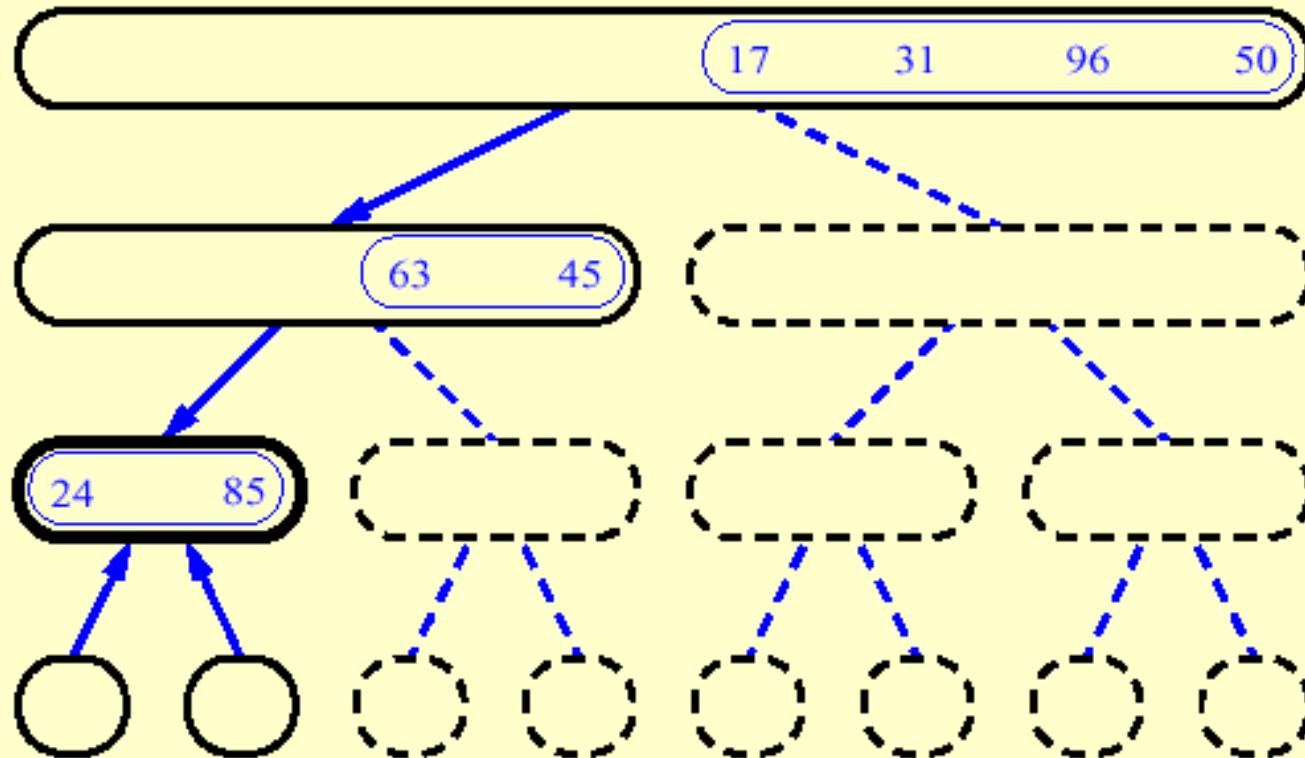
# MergeSort (Example) - 6



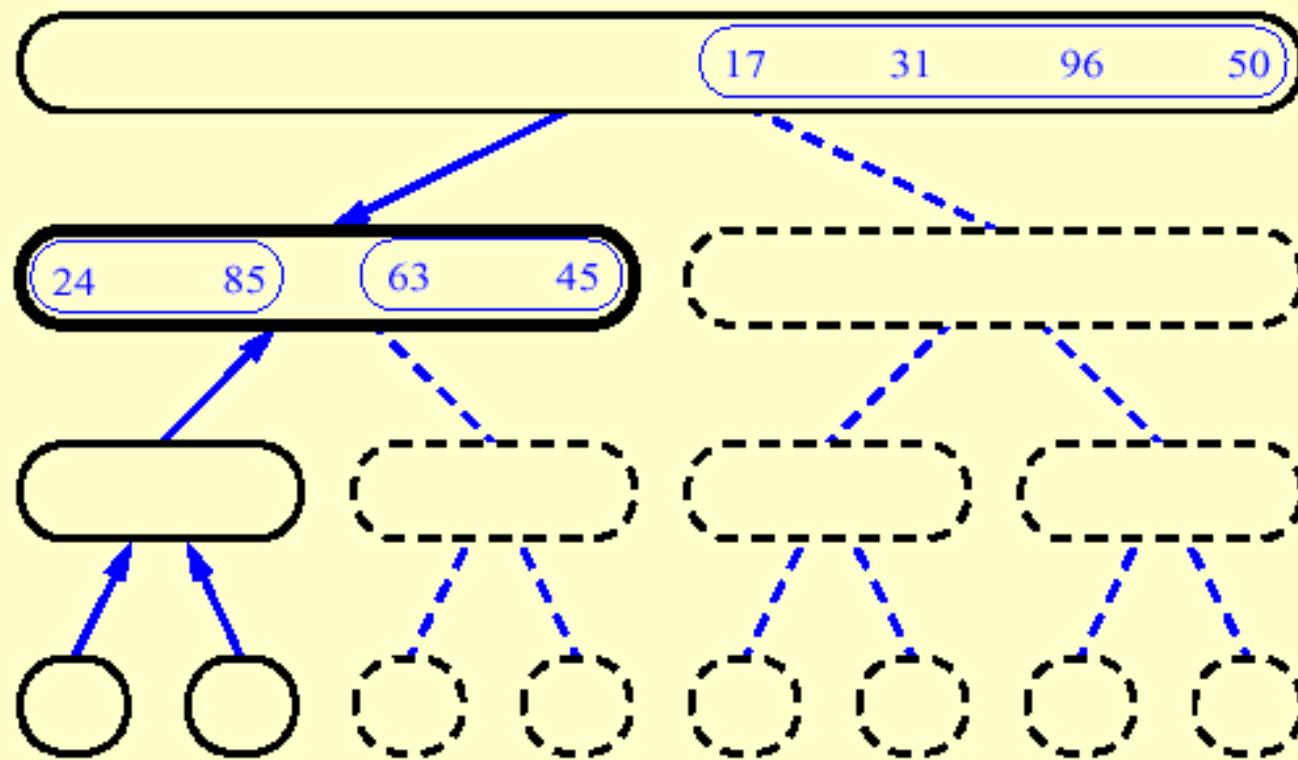
# MergeSort (Example) - 7



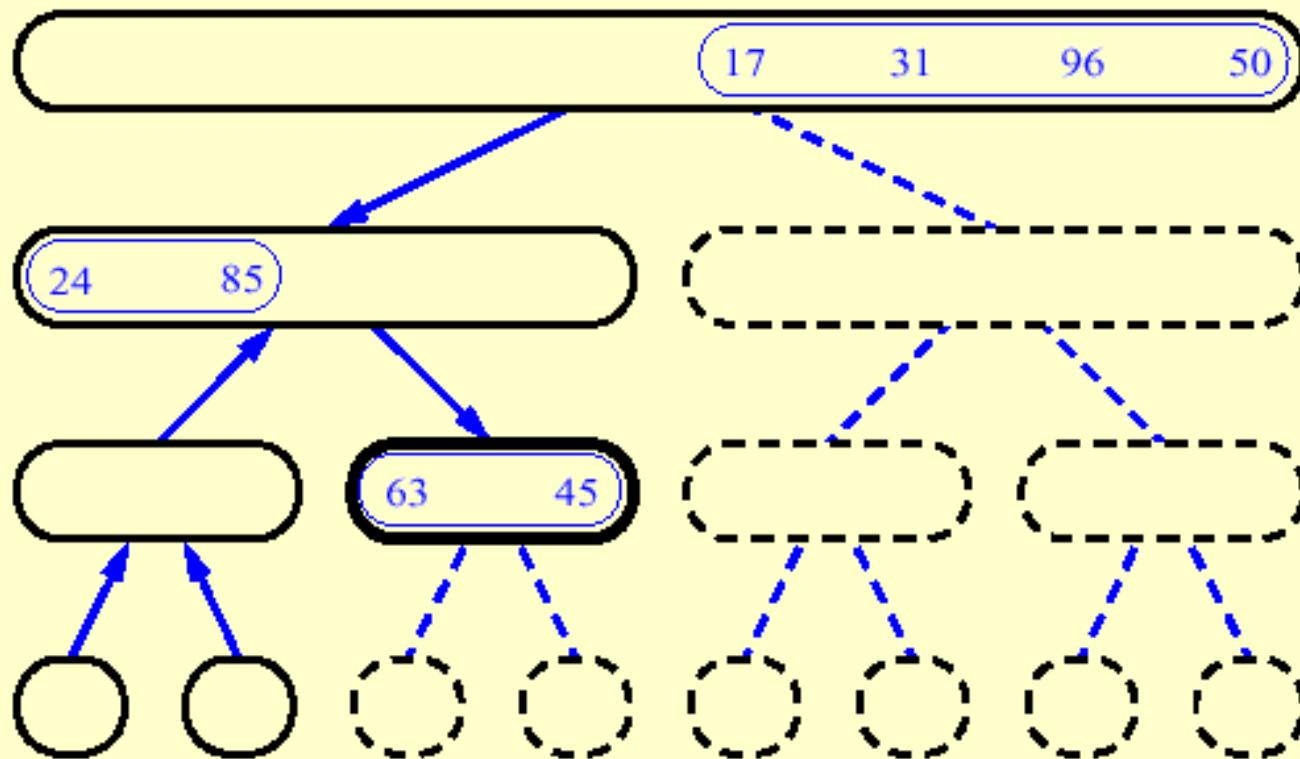
# MergeSort (Example) - 8



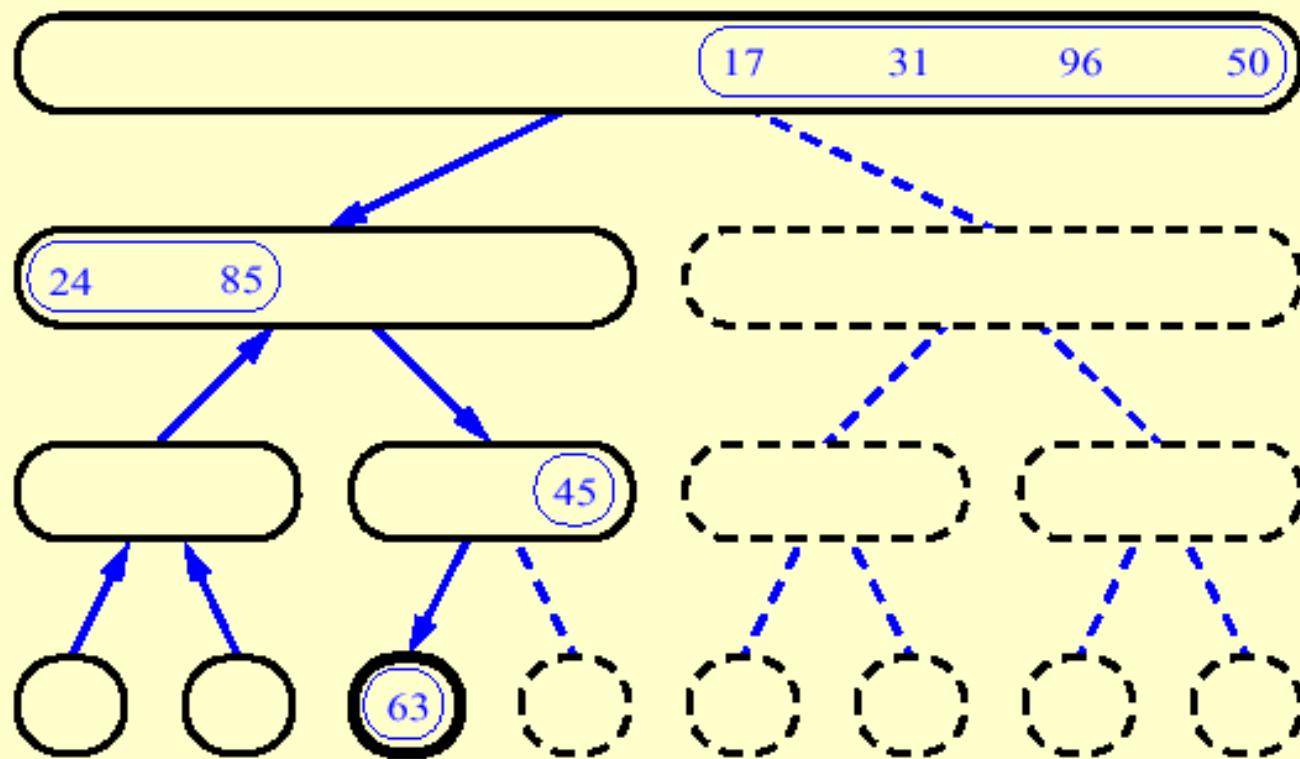
# MergeSort (Example) - 9



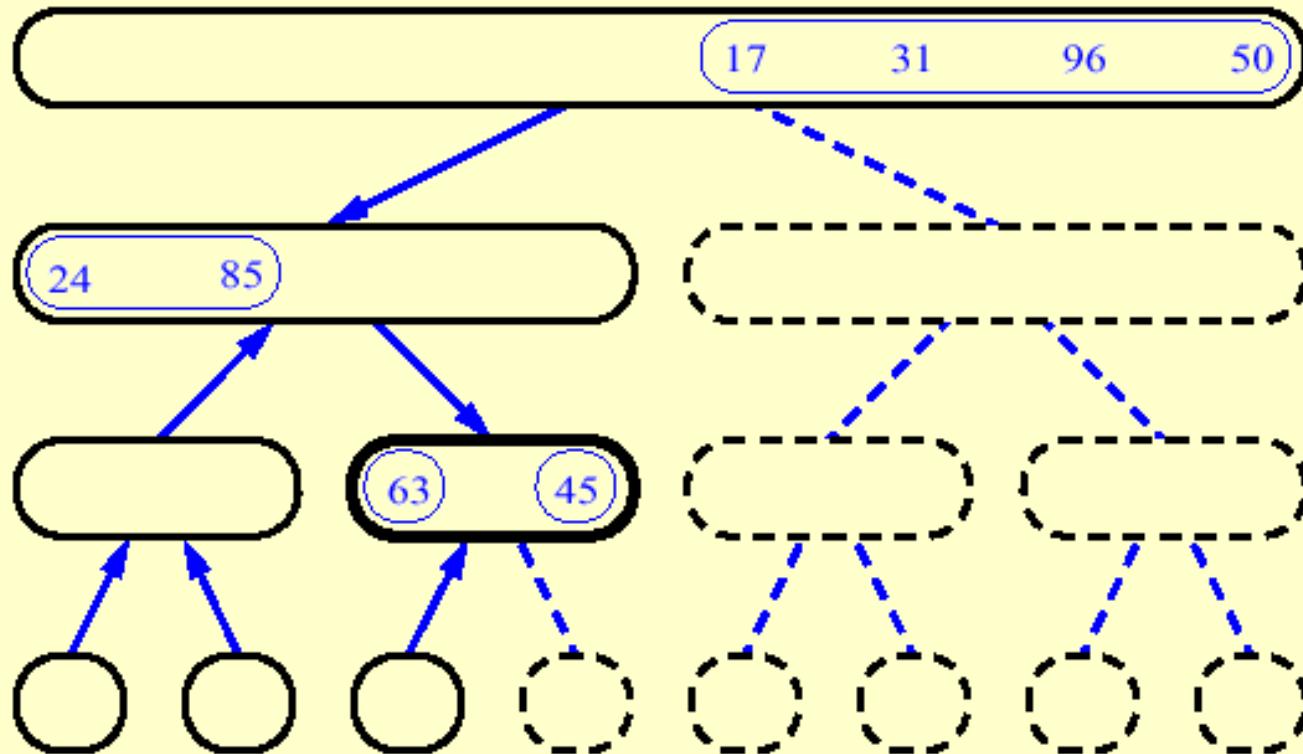
# MergeSort (Example) - 10



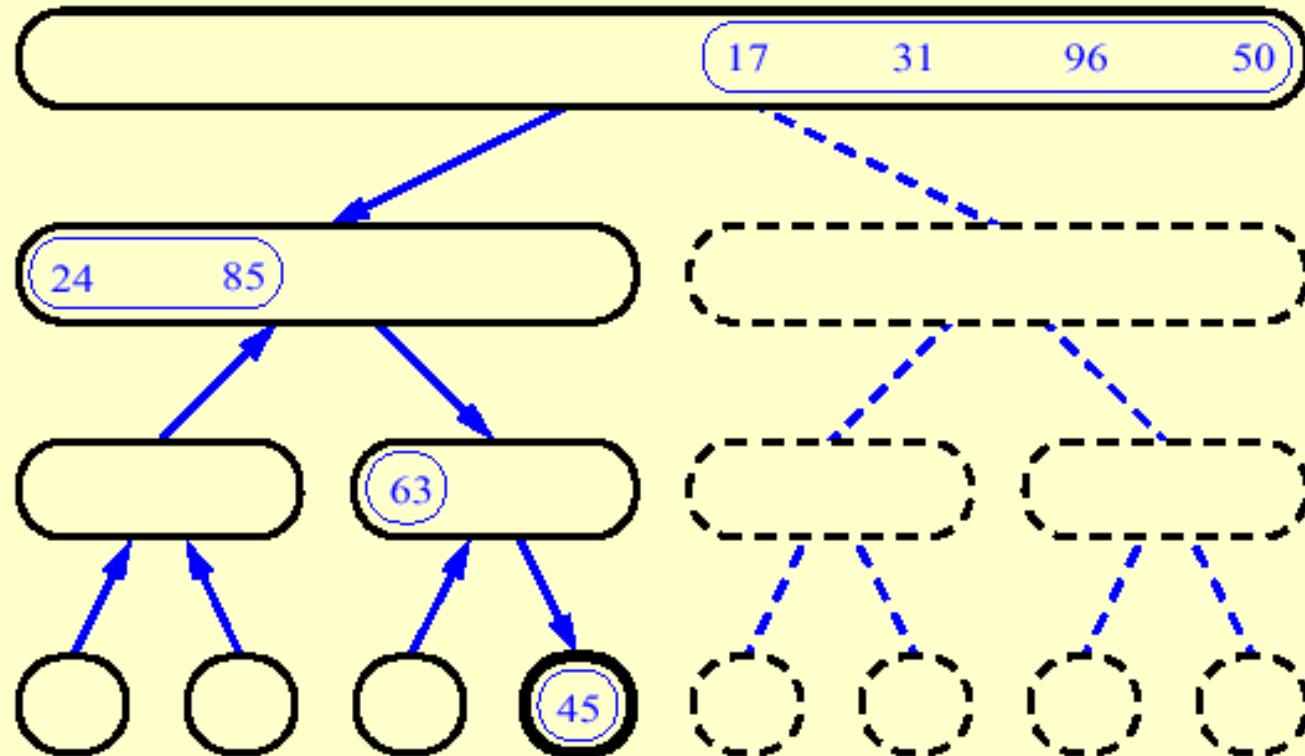
# MergeSort (Example) - 11



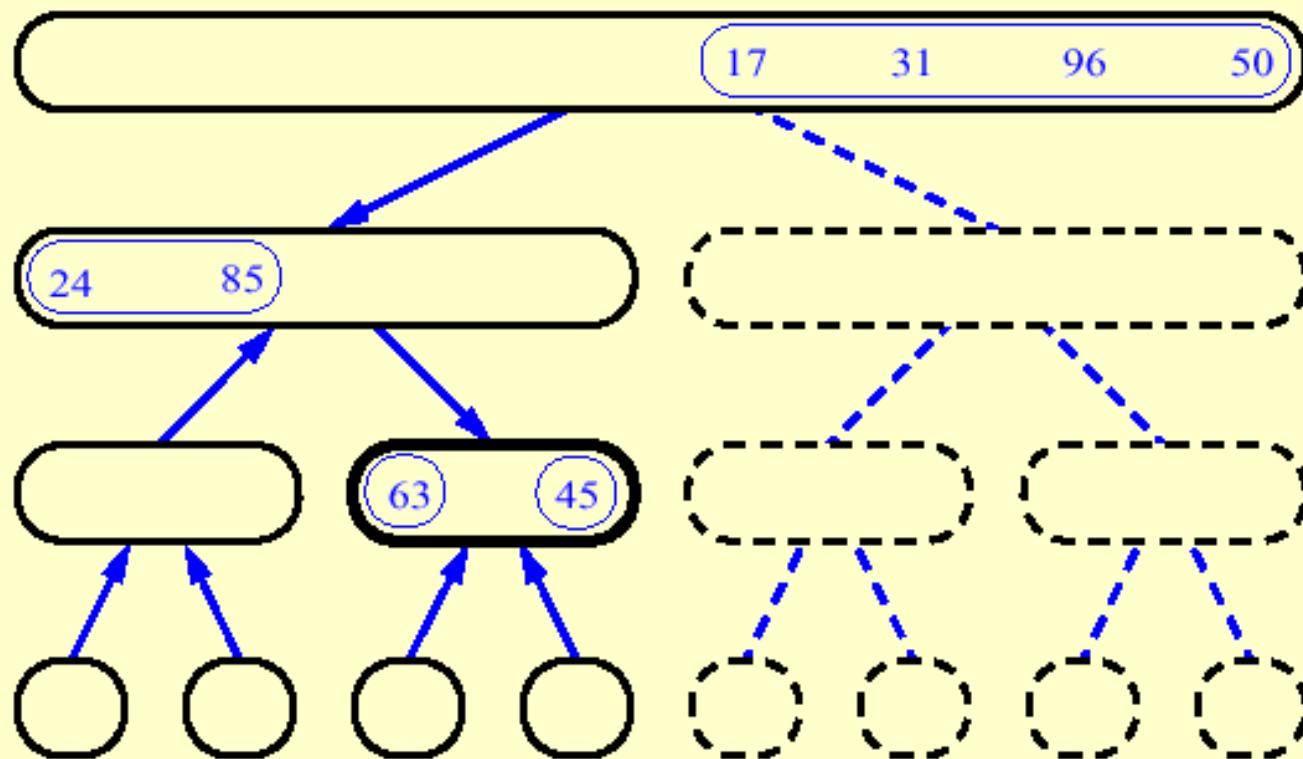
# MergeSort (Example) - 12



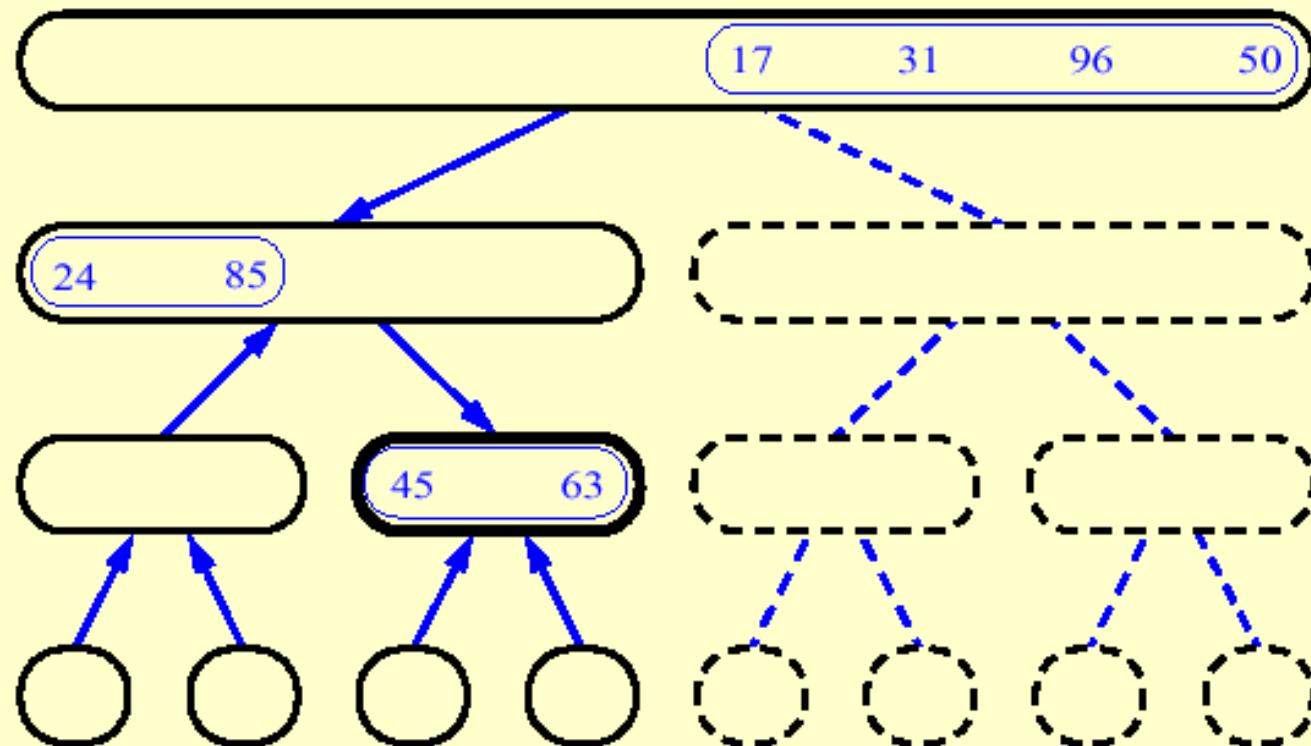
# MergeSort (Example) - 13



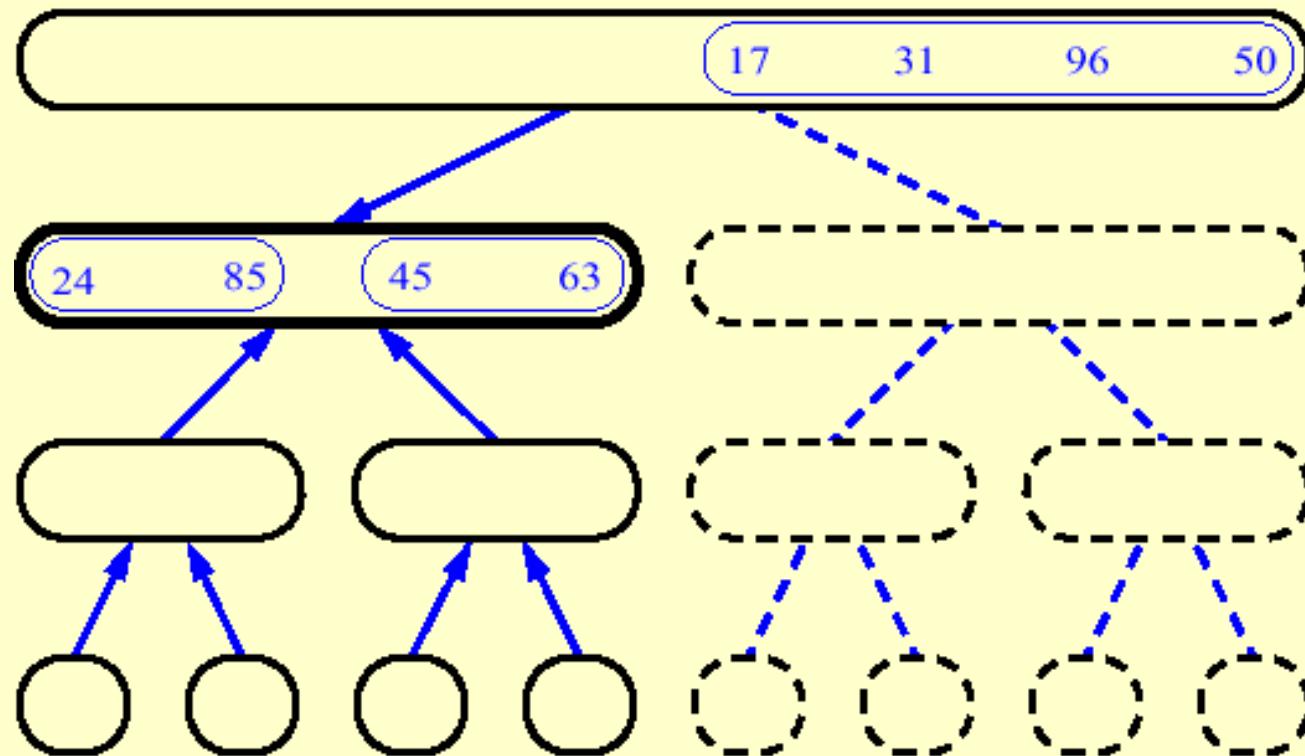
# MergeSort (Example) - 14



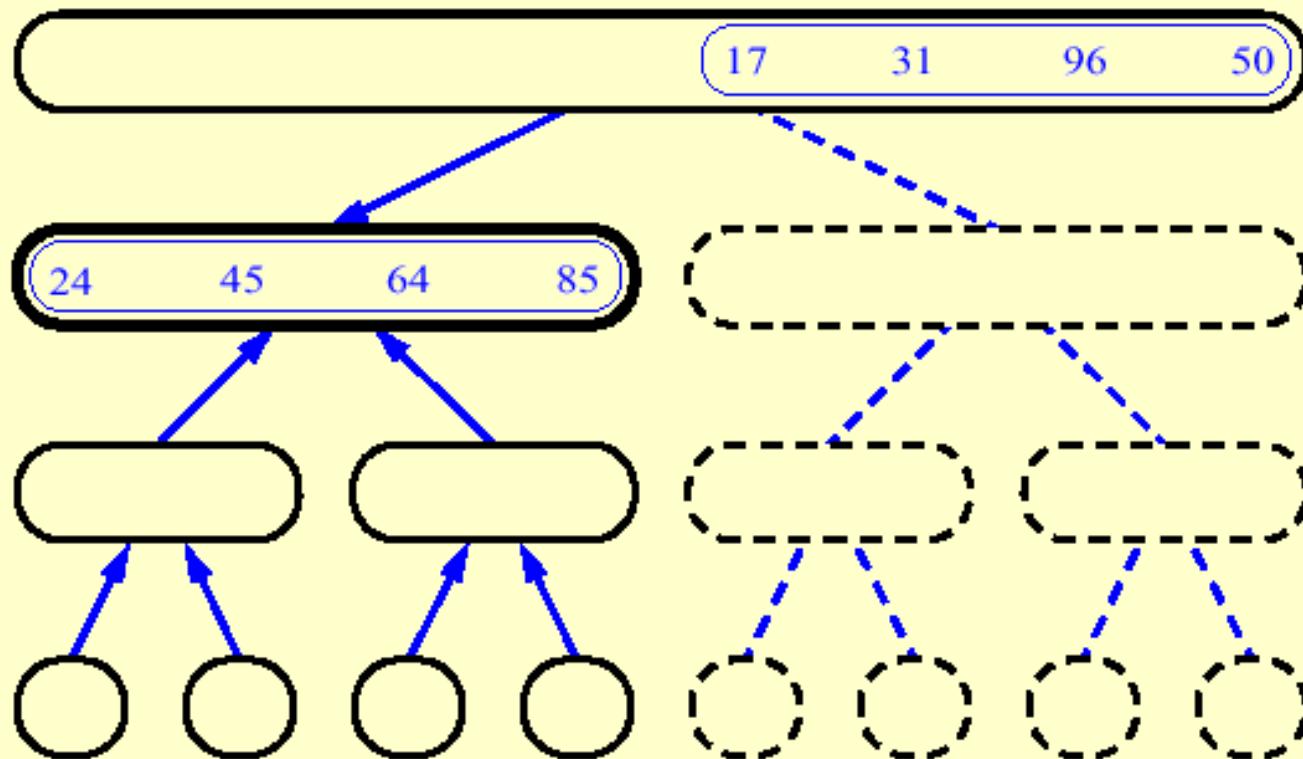
# MergeSort (Example) - 15



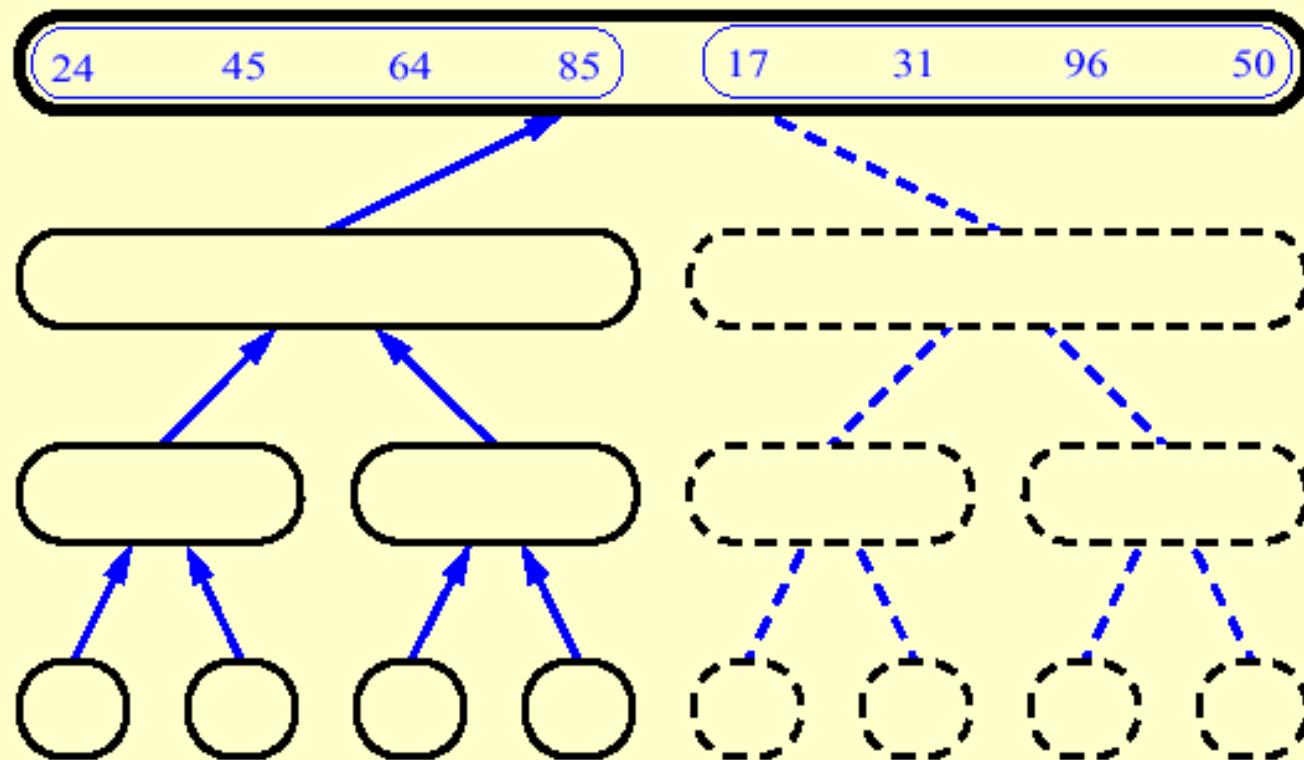
# MergeSort (Example) - 16



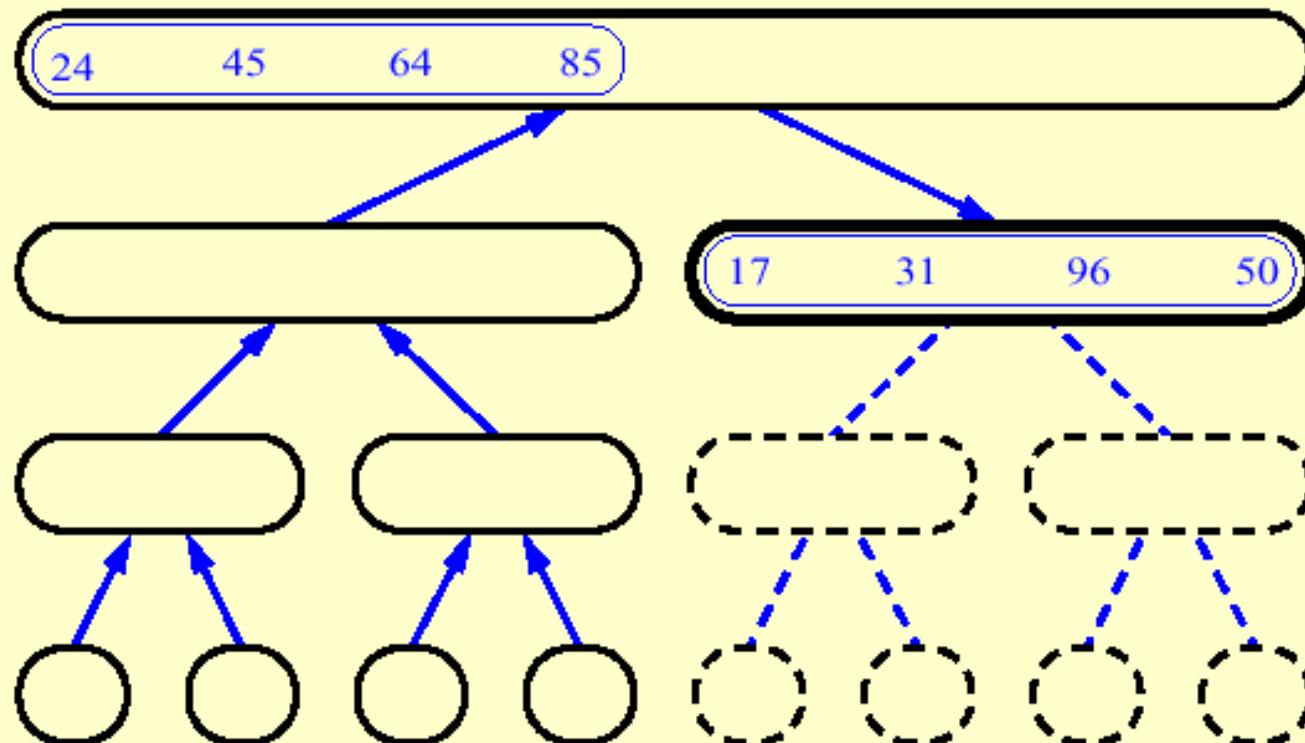
# MergeSort (Example) - 17



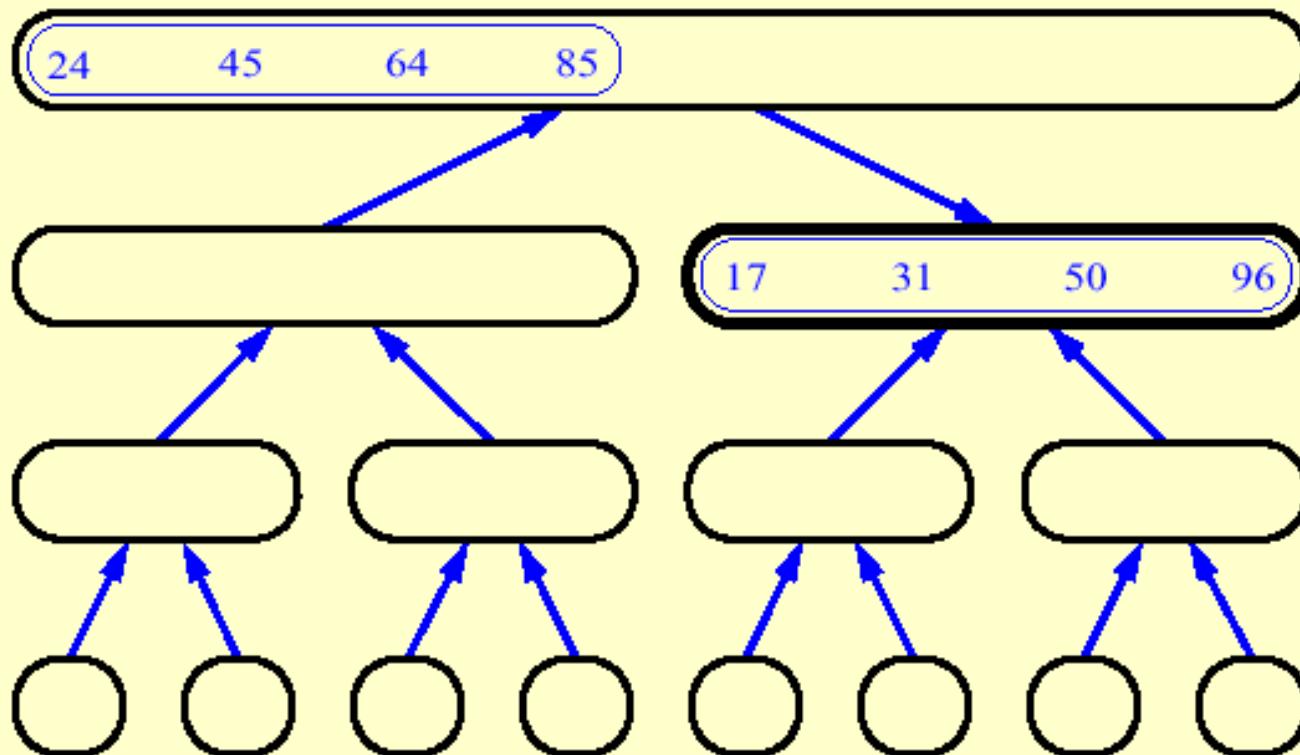
# MergeSort (Example) - 18



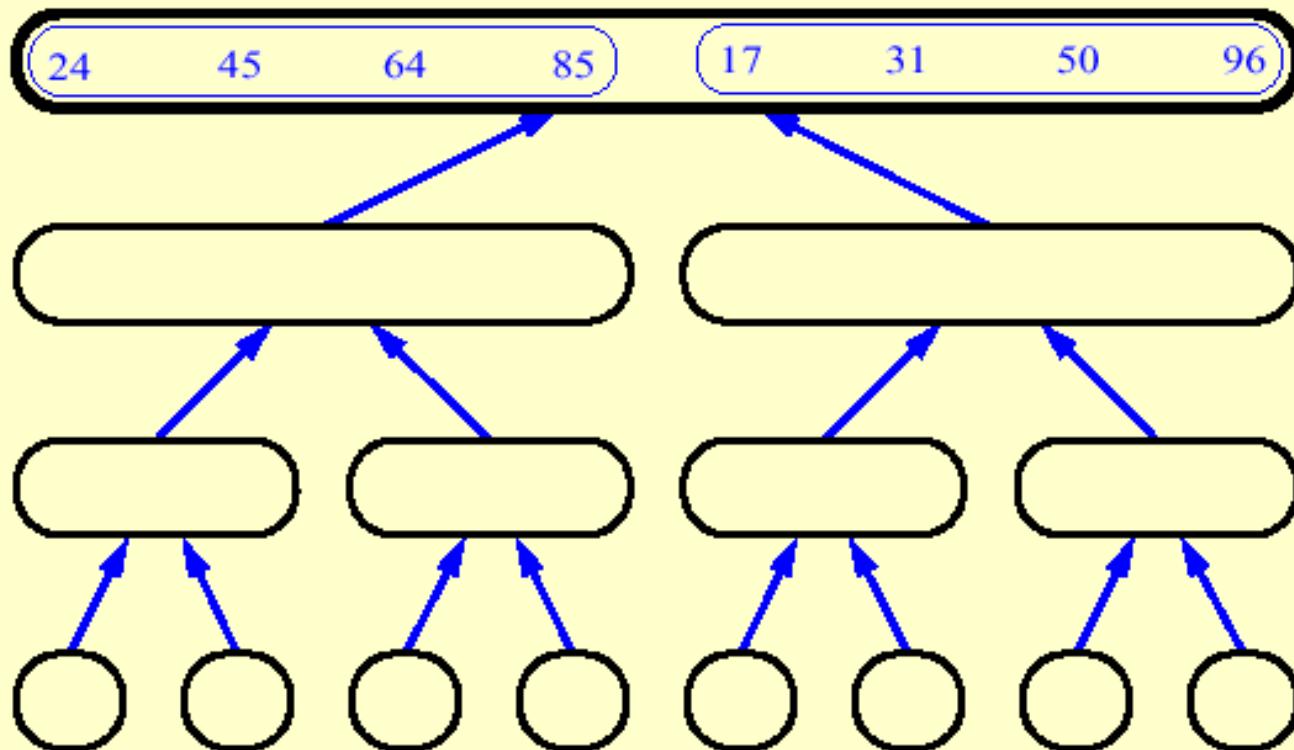
# MergeSort (Example) - 19



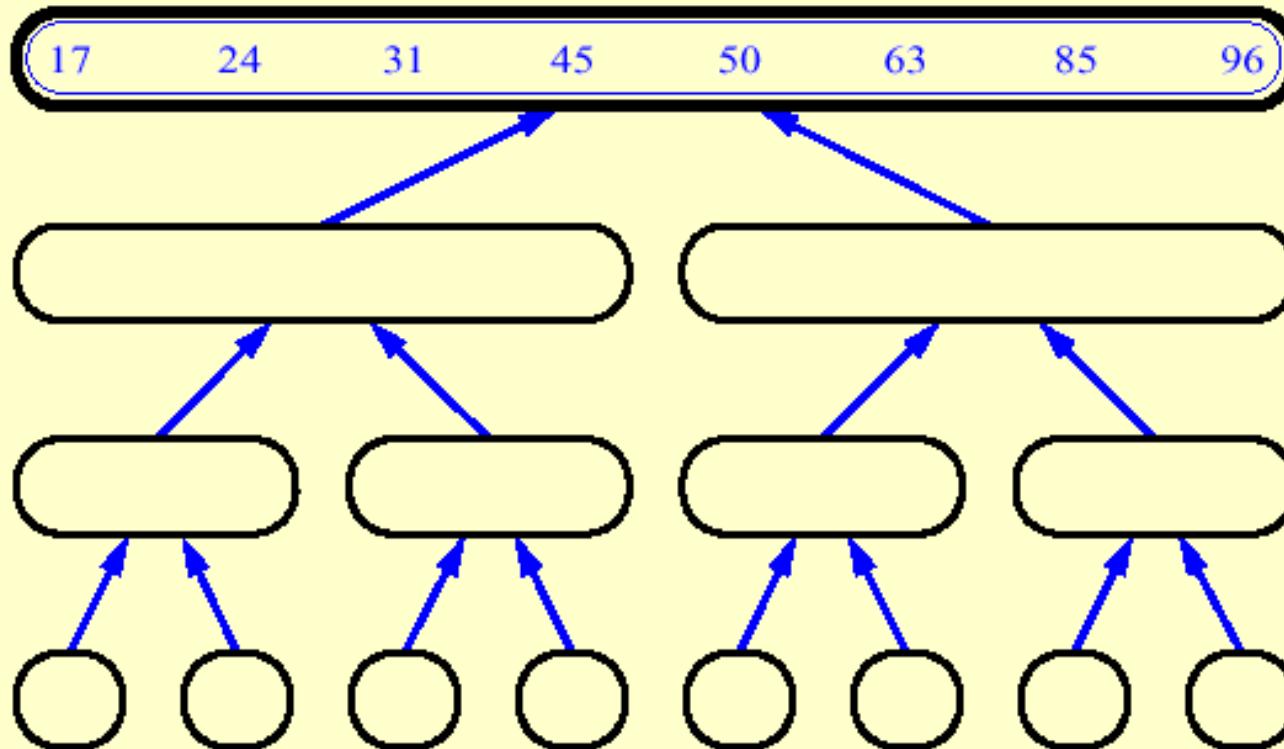
# MergeSort (Example) - 20



# MergeSort (Example) - 21



# MergeSort (Example) - 22





# Data Structures

Topic: Tree



By

**Ravi Kant Sahu**

*Asst. Professor,*

Lovely Professional University, Punjab



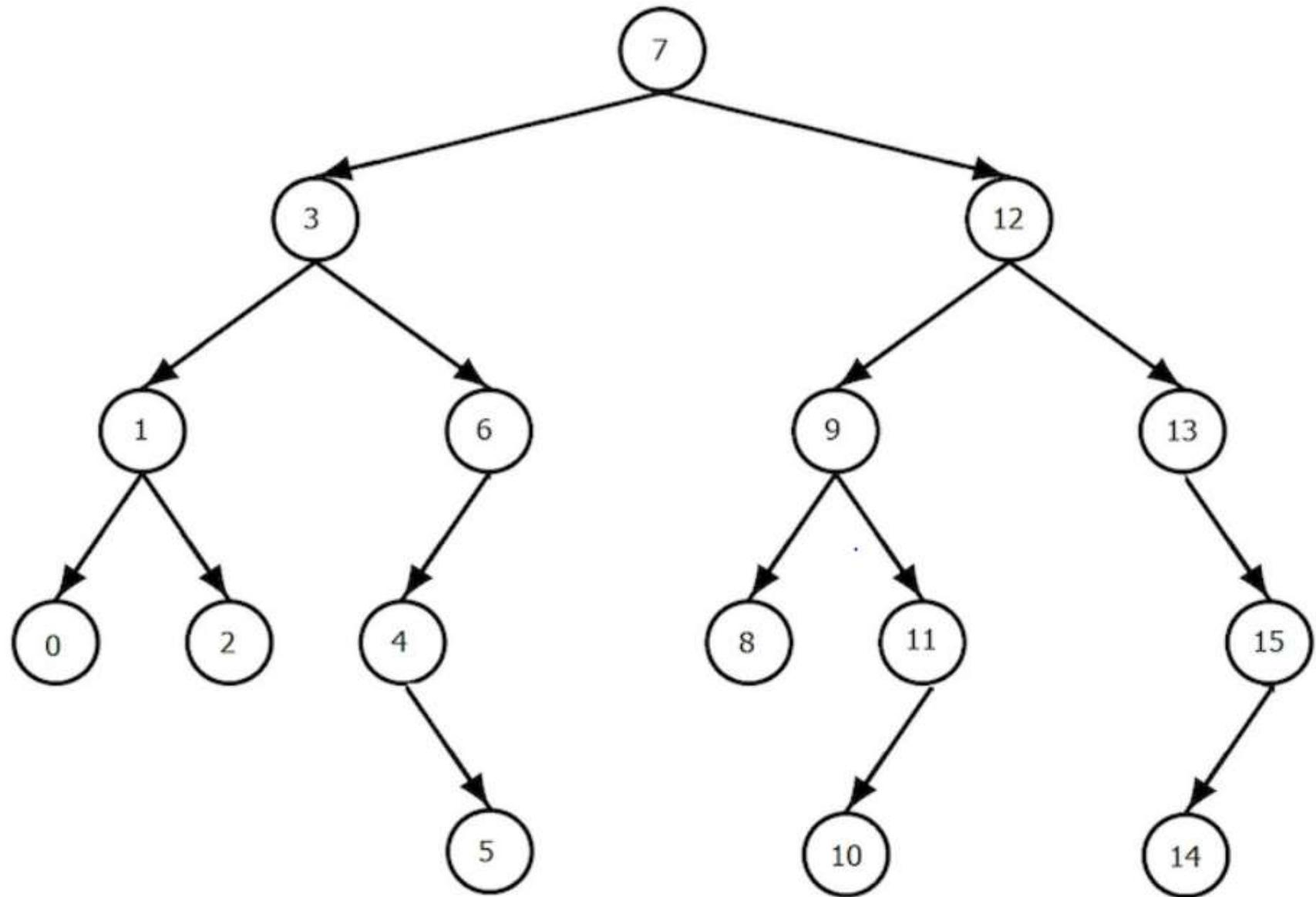
# Contents

- Introduction
- Binary Tree
- Basic Terminology
- Complete Binary Tree
- Extended Binary Tree
- Traversing Binary Tree
  - Pre-order
  - In-order
  - Post-order



# Introduction

- Trees are non-linear data structures.
- Used to represent a hierarchical relationship between the elements.





# Binary Tree

- A Binary Tree  $T$  is defined as a finite set of elements, called nodes, such that:
  - (a)  $T$  is empty (called the null tree or empty tree), or
  - (b)  $T$  contains a distinguished node  $R$ , called the root of  $T$ , and the remaining nodes of  $T$  form an ordered pair of disjoint binary trees  $T_1$  and  $T_2$ .
- $T_1$  and  $T_2$  are called left sub-tree and right subtrees of R.



# Binary Tree

- $T_1$  and  $T_2$  are called left sub-tree and right subtrees of  $R$ .
- If  $T_1$  is nonempty, then its root is called the left successor of  $R$ ; similarly, if  $T_2$  is nonempty, then its root is called the right successor of  $R$ .



# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Basic Terminology

- **Terminal Nodes:** The node with no successor is called terminal node or leaf.
- **Similar Trees:** Two binary trees T1 and T2 are said to be similar if they have the same shape or structure.
- **Copy of Trees:** Two binary trees T1 and T2 are said to be copies if they are similar and if they have the same contents at corresponding nodes.



# Basic Terminology

- Parent:
  - Left Child:
  - Right Child:
  - Siblings:
  - Level:
- 
- A line drawn from the node N to its successor is called *edge*.
  - The sequence of consecutive edges is called *Path*.
  - *Height or Depth* of a Tree is the number of edges in the longest branch of  $T$ .

*Height = Largest Level Number*



# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Complete Binary Tree

- A Binary tree T is said to be complete if all its levels, except possibly the last, have the maximum number of possible nodes and all the nodes at the last level appear as far left as possible.
- The depth of a complete Binary Tree T is:

$$D = \text{floor value} (\log_2 n)$$

**NOTE:** A binary tree in which all the levels are completely filled is known as FULL Binary Tree.



# Extended Binary Tree

- A Binary tree T is said to be extended binary tree or 2-Tree if each node N has either 0 or 2 children.
- Nodes with 0 child are called external nodes.
- Nodes with 2 children are called internal nodes.



# Questions



# MCQ-1

- What is the maximum height of a binary tree containing N nodes? (Height of a tree is the maximum number of edges from root to leaf )  
  
A.  $\log N$   
B. N  
C. N-1  
D. N+1



## MCQ-2

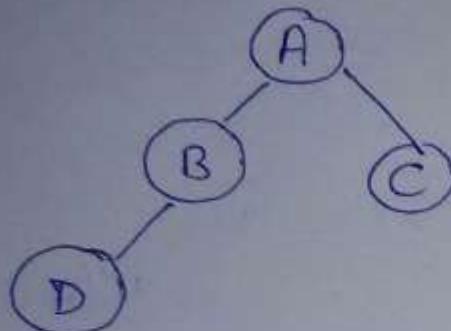
- What is the minimum height of a binary tree containing  $N$  nodes? (Height of a tree is the maximum number of edges from root to leaf )  
  
A.  $\text{ceil}(\log N)$   
B.  $\text{floor}(\log N)$   
C.  $N$   
D.  $N/2$



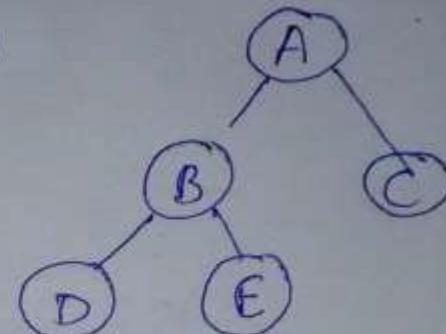
# MCQ-3

- Which of the following is not a Complete Binary Tree?

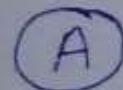
(A)



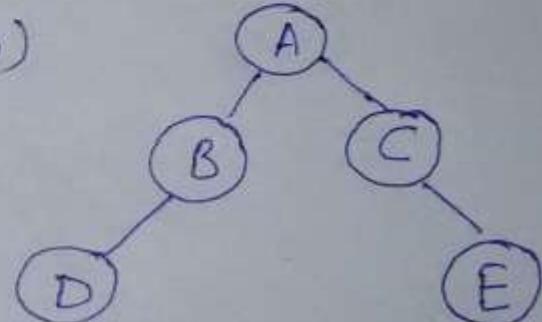
(B)



(C)



(D)

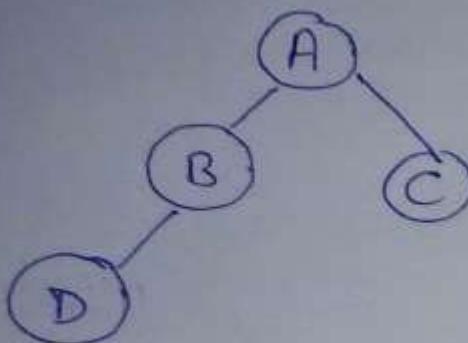




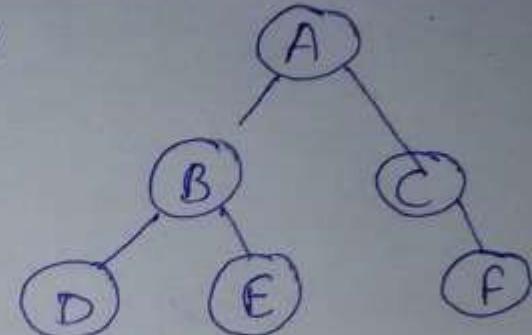
# MCQ-4

- Which of the following is a valid 2-Tree?

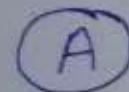
(A)



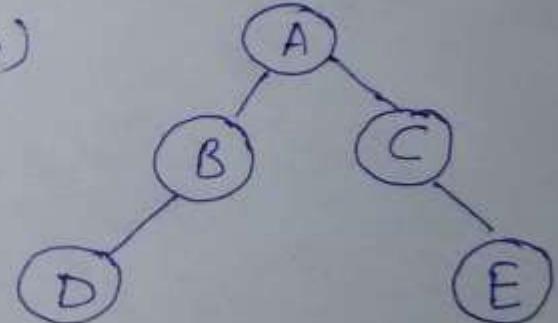
(B)



(C)



(D)





## MCQ-5

- Which of the following is TRUE?
  - A. Every Complete Binary tree is a 2-Tree
  - B. Every 2-Tree is a complete Binary Tree
  - C. Both A & B
  - D. None of These



# Review Questions

- What do you mean by Non-linear Data structures?
- What is the need of trees?
- How Complete Binary Tree is different from 2-Tree?

# Data Structures

Topic: Tree Traversal



By  
**Ravi Kant Sahu**

*Asst. Professor,*

Lovely Professional University, Punjab



# Contents

- Introduction
- Representing Binary Tree in Memory
  - Linked Representation
  - Sequential Representation
- Traversing Binary Tree
  - Preorder
  - Inorder
  - Postorder
- Traversal algorithms using Stacks
- Review Questions



# Introduction

- Trees are non-linear data structures.
- Trees can be represented in memory using linked list (linked representation) and arrays (sequential representation).
- One should have direct access to the root  $R$  of  $T$  and, given a node  $N$  of  $T$ , one should have direct access to the children of  $N$ .



# Linked Representation of Binary Tree

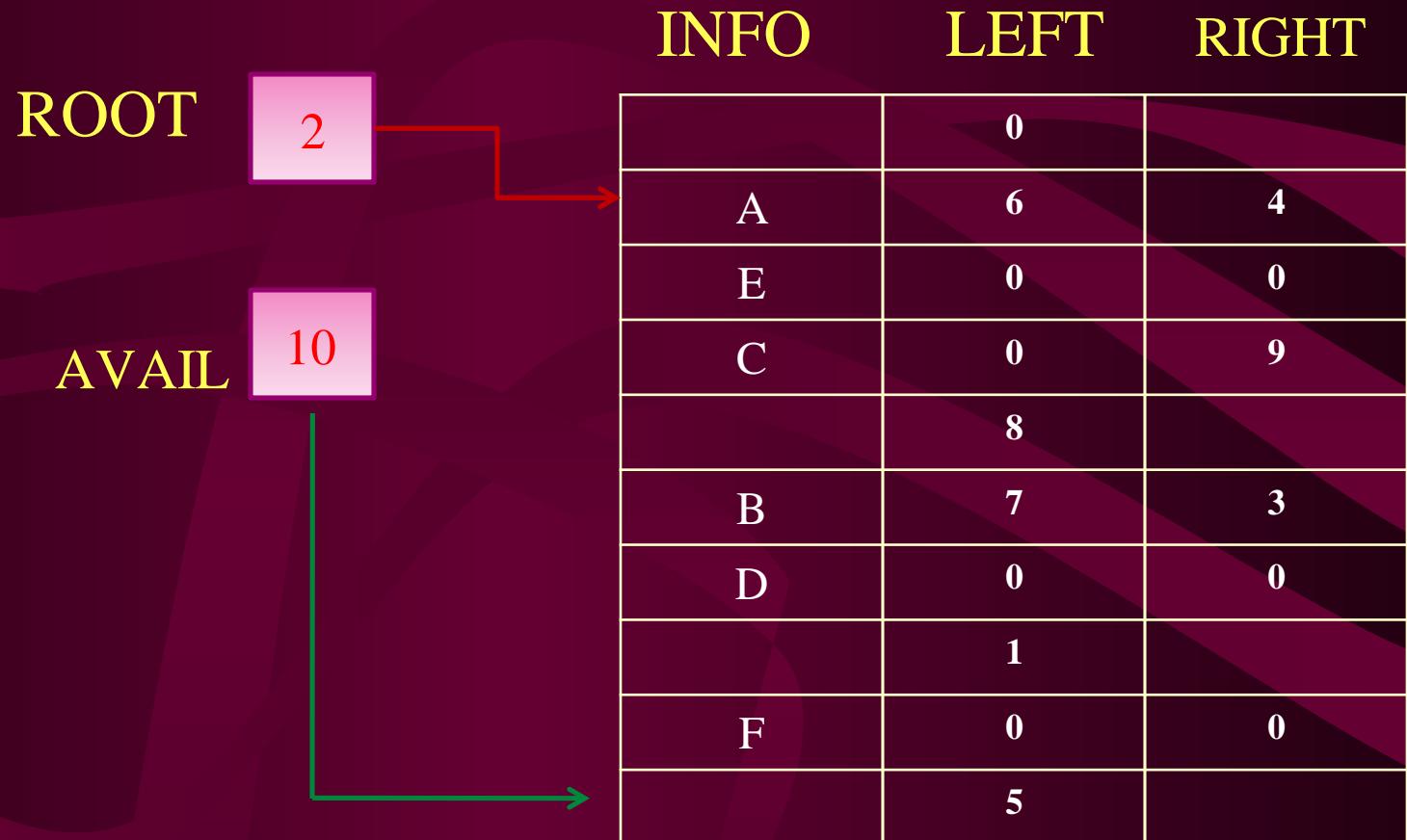
- A pointer variable ROOT and three parallel arrays (INFO, LEFT and RIGHT) are used.
- Each node N of Tree T corresponds to a location K such that:
  - (1) INFO[K] contains the data at the node N.
  - (2) LEFT [K] contains the location of the left child of node N.
  - (3) RIGHT[K] contains the location of the right child of node N.



- If any subtree is empty then corresponding pointer will contain the NULL value.
- If Tree is empty then Root will contain NULL.
- An entire record may be stored at the node N.



# Linked Representation





# Sequential Representation of Binary Tree

- Efficient for the tree T that is complete or nearly complete.
- Use of only a single linear array TREE such that:
  - (a) The Root of T is stored in TREE [1].
  - (b) If a node N occupies Tree [K], then its left child is stored in TREE [2\*K] and right child is stored in TREE [2\*K + 1].



# Preorder Traversal of Binary Tree

PREORDER (INFO, LEFT, RIGHT, ROOT )

1. If: ROOT = NULL Then: Return
2. Print ROOT->INFO
3. Call PREORDER(ROOT -> LEFT)
4. Call PREORDER(ROOT -> RIGHT)



# Inorder Traversal of Binary Tree

**INORDER (INFO, LEFT, RIGHT, ROOT )**

1. If:  $\text{ROOT} = \text{NULL}$  Then: Return
2. Call  $\text{INORDER}(\text{ROOT} \rightarrow \text{LEFT})$
3. Print  $\text{ROOT} \rightarrow \text{INFO}$
4. Call  $\text{INORDER}(\text{ROOT} \rightarrow \text{RIGHT})$



# Postorder Traversal of Binary Tree

**POSTORDER (INFO, LEFT, RIGHT, ROOT )**

1. If:  $\text{ROOT} = \text{NULL}$  Then: Return
2. Call POSTORDER( $\text{ROOT} \rightarrow \text{LEFT}$ )
3. Call POSTORDER( $\text{ROOT} \rightarrow \text{RIGHT}$ )
4. Print  $\text{ROOT} \rightarrow \text{INFO}$



# Binary Tree Creation

Construct the Binary tree from the Inorder and Preorder  
Traversals:

INORDER: 4, 2, 1, 7, 5, 8, 3, 6

PREORDER: 1, 2, 4, 3, 5, 7, 8, 6



# Binary Tree Creation

Construct the Binary tree from the Inorder and Postorder  
Traversals:

IN-ORDER:      E X D M K L A T P Z

POST-ORDER:    X M D E K A P Z T L



# Review Question

Given the In-order and Pre-order Traversals of a Binary Tree, Construct the Tree and show all the steps.

IN-ORDER: G, E, T, B, M, K, S, L

PRE-ORDER: T, G, E, K, B, M, S, L



# Review Question

Given the Post-order and Inorder Traversals of a Binary Tree,  
Find out its Preorder Traversal.

POST-ORDER: 18, 23, 28, 25, 20, 55, 50, 65, 60, 30

IN-ORDER: 18, 20, 23, 25, 28, 30, 50, 55, 60, 65



# Questions

# Data Structures

Topic: Binary Search Tree



By

**Ravi Kant Sahu**

*Asst. Professor,*

Lovely Professional University, Punjab



# Contents

- Introduction
- Searching in BST
- Insertion in BST
- Deletion in BST
- Review Questions



# Binary Search Tree

Binary Search Tree is a binary tree which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Searching in BST

SEARCH\_BST(INFO, LEFT, RIGHT, ROOT, ITEM, LOC)

1. Set PTR = ROOT and LOC = NULL.
2. Repeate step 3 while PTR!=NULL:
  3. If ITEM = INFO [PTR], then:  
Set LOC = PTR and Exit.  
else If ITEM < INFO[PTR], then:  
Set PTR = LEFT [PTR].  
else  
Set PTR = RIGHT [PTR].  
[End of Loop]
4. If LOC = NULL, then Write “Search unsuccessful”.
5. Return LOC.



# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Insertion in BST

INS\_BST(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)

1. If AVAIL = NULL, then: Write “OVERFLOW” and Exit.
2. Call FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
3. If LOC != NULL, then Exit.
4. [Create NEW Node]
  - (a) Set NEW = AVAIL, AVAIL = LEFT[AVAIL] and INFO[NEW] = ITEM.
  - (b) Set LEFT[NEW] = NULL and RIGHT[NEW] = NULL.
5. [Add ITEM to Tree]  
If PAR = NULL, then  
    Set ROOT = NEW.  
Else if ITEM < INFO[PAR], then:  
    Set LEFT[PAR] = NEW.  
Else: Set RIGHT [PAR] = NEW.
6. Exit.



# Insertion in BST...

FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

1. If ROOT = NULL, then:  
Set PAR = NULL and LOC = NULL and Return.
2. Set PTR = ROOT and SAVE = NULL.
3. Repeate while PTR!= NULL
4.     If ITEM = INFO[PTR], then  
        Set PAR = SAVE and LOC = PTR and Return.  
        else if ITEM < INFO [PTR], then:  
            Set SAVE = PTR and PTR = LEFT [PTR].  
        else  
            Set SAVE = PTR and PTR = RIGHT [PTR].  
    [End of Loop]
6. [Search Unsuccessful]  
If PTR= NULL, then Return PAR= SAVE and LOC = NULL.



# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Deletion in BST

Find the location of node N which contains the ITEM.

## CASE 1:

N has no children. Then N is deleted from T by simply replacing the location of N in the parent node P(N) by Null Pointer.

## CASE 2:

N has exactly one child. Then N is deleted from T by simply replacing the location of N in P(N) by the location of the only child of N.

## CASE 3:

N has two children. Let S(N) denote the inorder successor of N. Then N is deleted from T by first deleting S(N) from T and then replacing node N in T by the node S(N).



# Deletion in BST

**DELETE\_BST ( INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)**

1. Call FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
2. If LOC = NULL, then: Write “ITEM not in Tree” and Exit.
3. If RIGHT[LOC] != NULL and LEFT [LOC] != NULL, then:  
Call DELETE\_B (INFO, LEFT, RIGHT, ROOT, LOC, PAR)  
Else:  
    Call DELETE\_A (INFO, LEFT, RIGHT, ROOT, LOC, PAR))
4. Set LEFT[LOC] = AVAIL and AVAIL = LOC.
5. Exit.



# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Deletion in BST (1)

**DELETE\_A( INFO, LEFT, RIGHT, ROOT, LOC, PAR)**

1. If  $\text{LEFT}[\text{LOC}] = \text{NULL}$ , and  $\text{RIGHT}[\text{LOC}] = \text{NULL}$ , then:

    Set CHILD = NULL.

    Else if  $\text{LEFT}[\text{LOC}] \neq \text{NULL}$ , then:

        Set CHILD = LEFT [LOC].

    Else

        Set CHILD = RIGHT [LOC]

2. If  $\text{PAR} \neq \text{NULL}$ , then:

    If  $\text{LOC} = \text{LEFT}[\text{PAR}]$ , then:

        Set  $\text{LEFT}[\text{PAR}] = \text{CHILD}$ .

    Else: Set  $\text{RIGHT}[\text{PAR}] = \text{CHILD}$ .

    Else: Set  $\text{ROOT} = \text{CHILD}$ .

3. Return



# Deletion in BST(2)

**DELETE\_B(INFO, LEFT, RIGHT, ROOT, LOC, PAR)**

1. [Find SUC and PARSUC]
  - (a) Set PTR = RIGHT[LOC] and SAVE = LOC.
  - (b) Repeat while LEFT[PTR] != NULL:  
Set SAVE = PTR and PTR = LEFT[PTR]
  - (c) Set SUC = PTR and PARSUC = SAVE.
2. [Delete Inorder Successor]  
Call DELETE\_A(INFO, LEFT, RIGHT, ROOT, SUC, PARSUC)
3. [Replace node N by its inorder successor.]
  - (a) If PAR != NULL, then:  
If LOC = LEFT[PAR], then:  
Set LEFT[PAR] = SUC  
Else: Set RIGHT [PAR] = SUC  
Else: Set ROOT = SUC.
  - (b) Set LEFT[SUC] = LEFT[LOC] and  
RIGHT[SUC] = RIGHT[LOC].
4. Return



Questions



# Review Questions

- What is the maximum size of array required to represent a tree with height  $h$ , using sequential representation?
- Sequential representation of tree is more efficient than linked. When?

# Data Structures and Algorithms

---

## AVL Search Tree



By

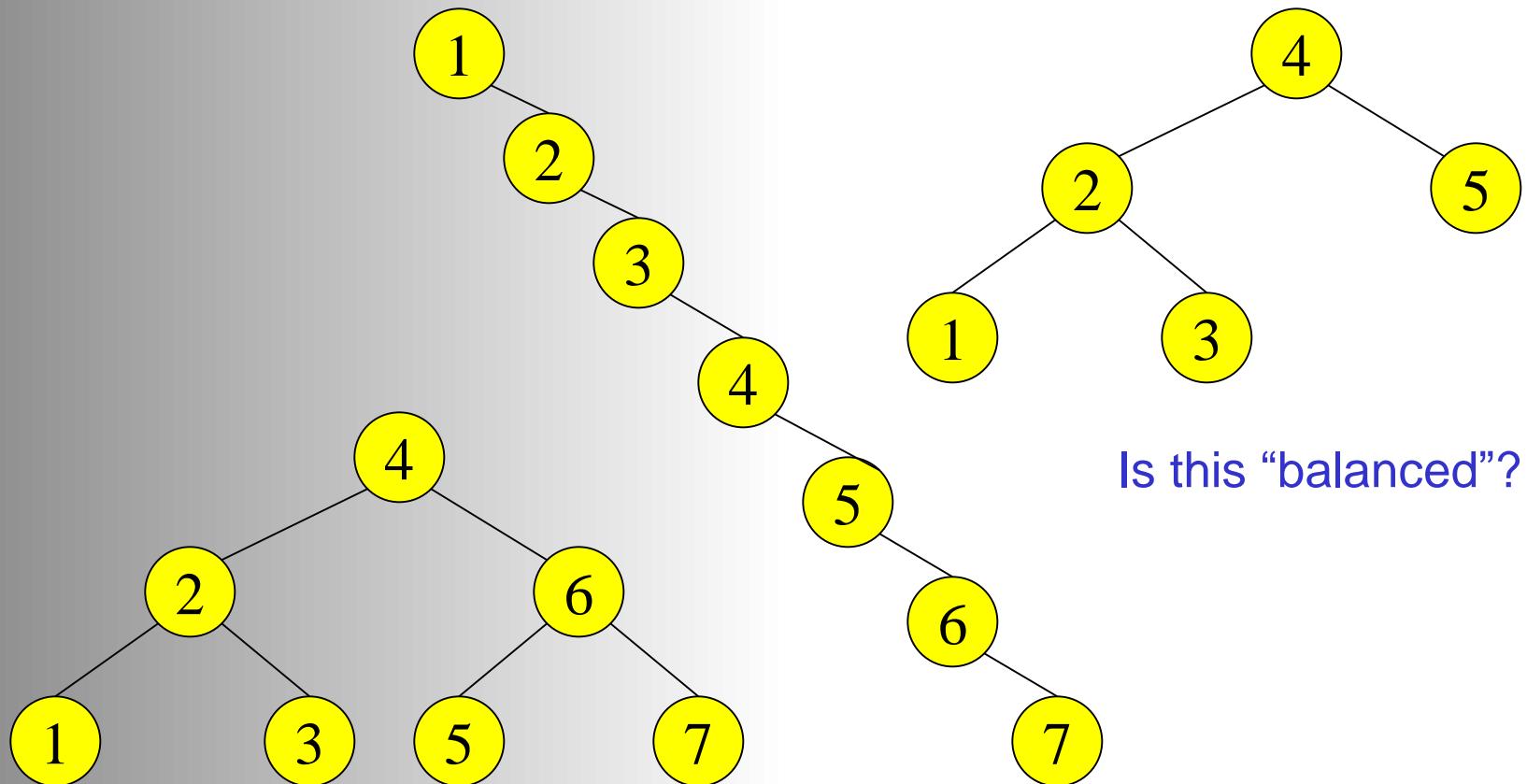
**Ravi Kant Sahu**

*Asst. Professor,*

Lovely Professional University, Punjab

# Balanced and Unbalanced BST

---



# AVL Search Tree

---

- Skewed Binary Search Tree:  
Worst case time complexity is  $O(n)$ .
- Adelson-Velskii and Landis introduced height balanced tree in 1962.
- Balance factor of a node =  
 $\text{height(left sub-tree)} - \text{height(right sub-tree)}$

# AVL - Good but not Perfect Balance

---

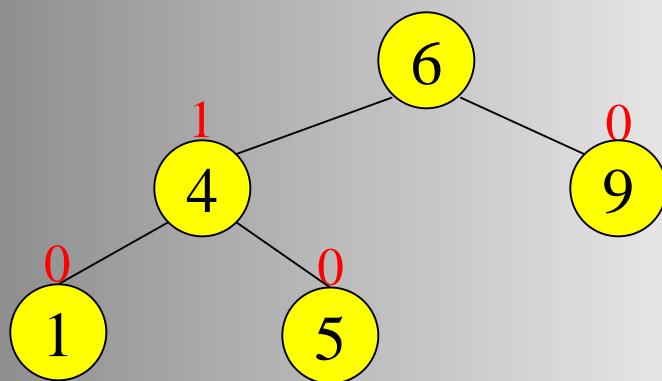
- AVL trees are height-balanced binary search trees.
- An AVL tree has balance factor calculated at every node
  - › For every node, heights of left and right sub-tree can differ by no more than 1
  - › Balance Factor of a node is -1, 0 or 1 in AVL.

# WORK SPACE

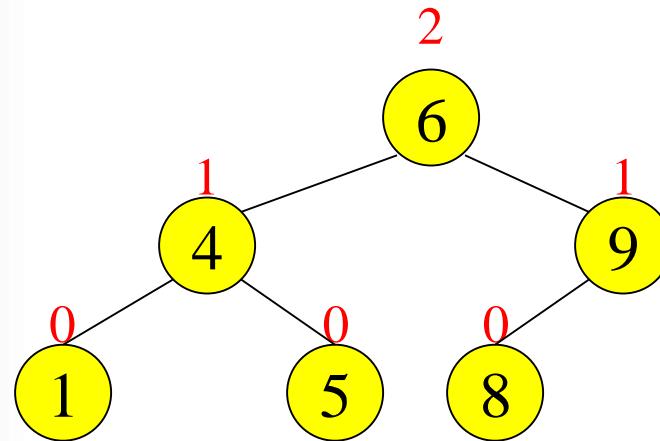
---

# Node Heights

Tree A (AVL)  
height=2 BF=1-0=1



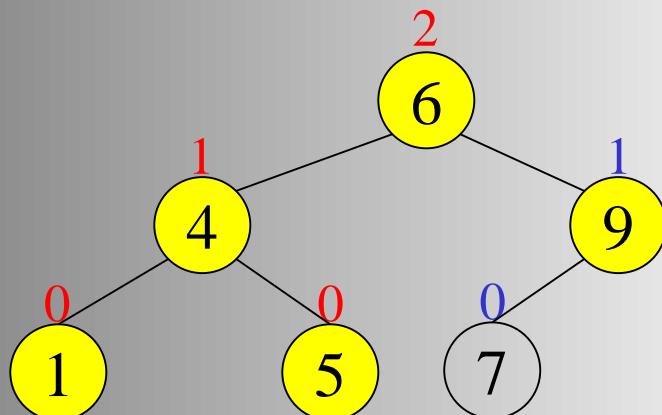
Tree B (AVL)



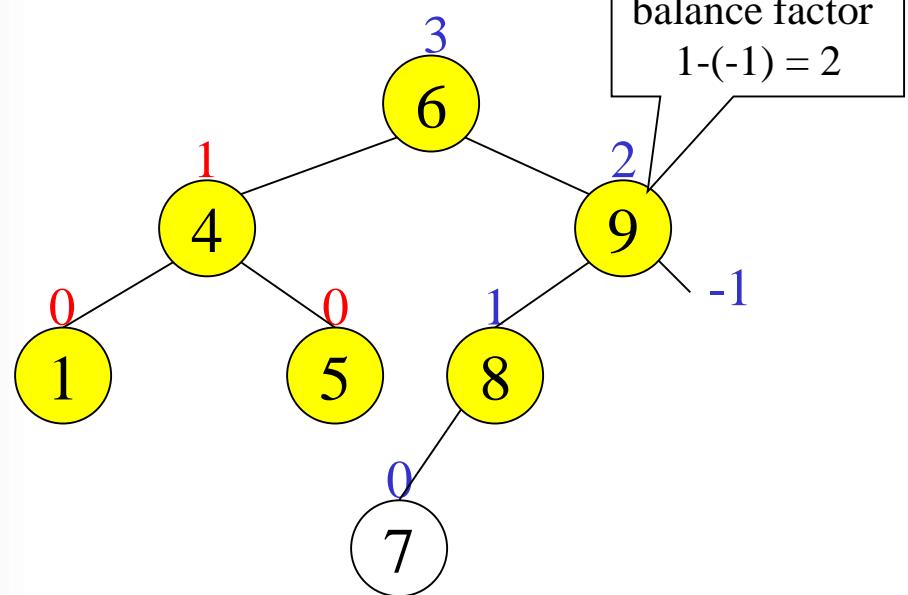
height of node =  $h$   
balance factor =  $h_{\text{left}} - h_{\text{right}}$

# Node Heights after Insert 7

Tree A (AVL)



Tree B (not AVL)



# Basic Concepts

## LR and RL Rotation

---

- Find Out the first Node from the bottom which has BF other than 1, 0, -1, call it A and its descendent towards the newly inserted node as B.
- **LR Rotation:** If newly inserted node is in the right subtree of left subtree of A.
  - Apply RR rotation on B
  - Then Apply LL rotation on A
- **RL Rotation:** If newly inserted node is in the left subtree of right subtree of A.
  - Apply LL rotation on B
  - Then Apply RR rotation on A

# Insert and Rotation in AVL Trees

---

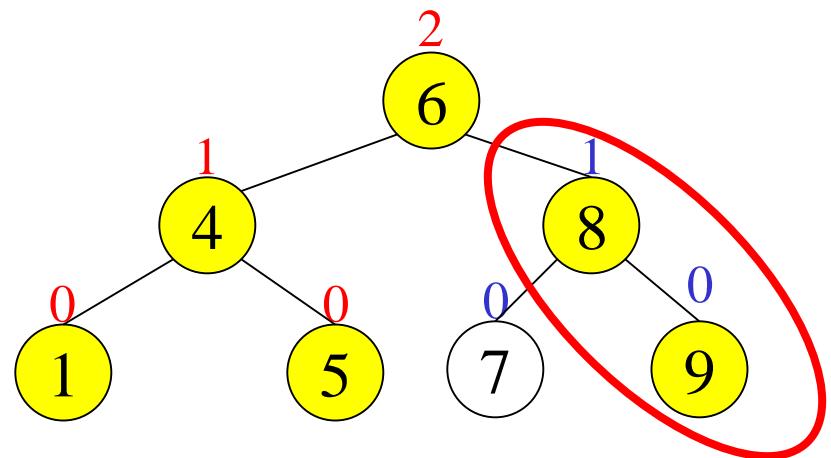
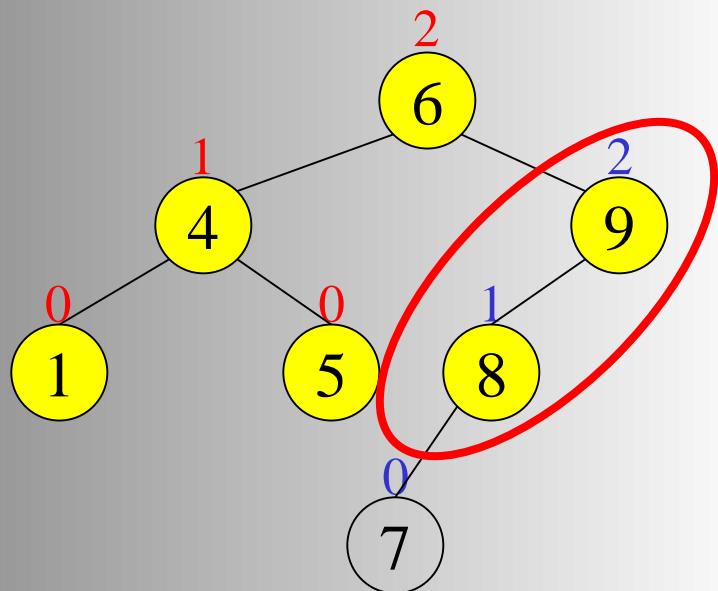
- Insert operation may cause balance factor to become 2 or –2 for some node
  - › only nodes on the path from insertion point to root node have possibly changed in height
  - › So after the Insert, go back up to the root node by node, updating heights
  - › If a new balance factor (the difference  $h_{\text{left}} - h_{\text{right}}$ ) is 2 or –2, adjust tree by *rotation* around the node

# WORK SPACE

---

# Single Rotation in an AVL Tree

---



# Insertions in AVL Trees

---

Let the node that needs rebalancing be  $\alpha$ .

There are 4 cases:

**Outside Cases** (require single rotation) :

1. Insertion into **left** subtree **of left** child of  $\alpha$ .
2. Insertion into **right** subtree **of right** child of  $\alpha$ .

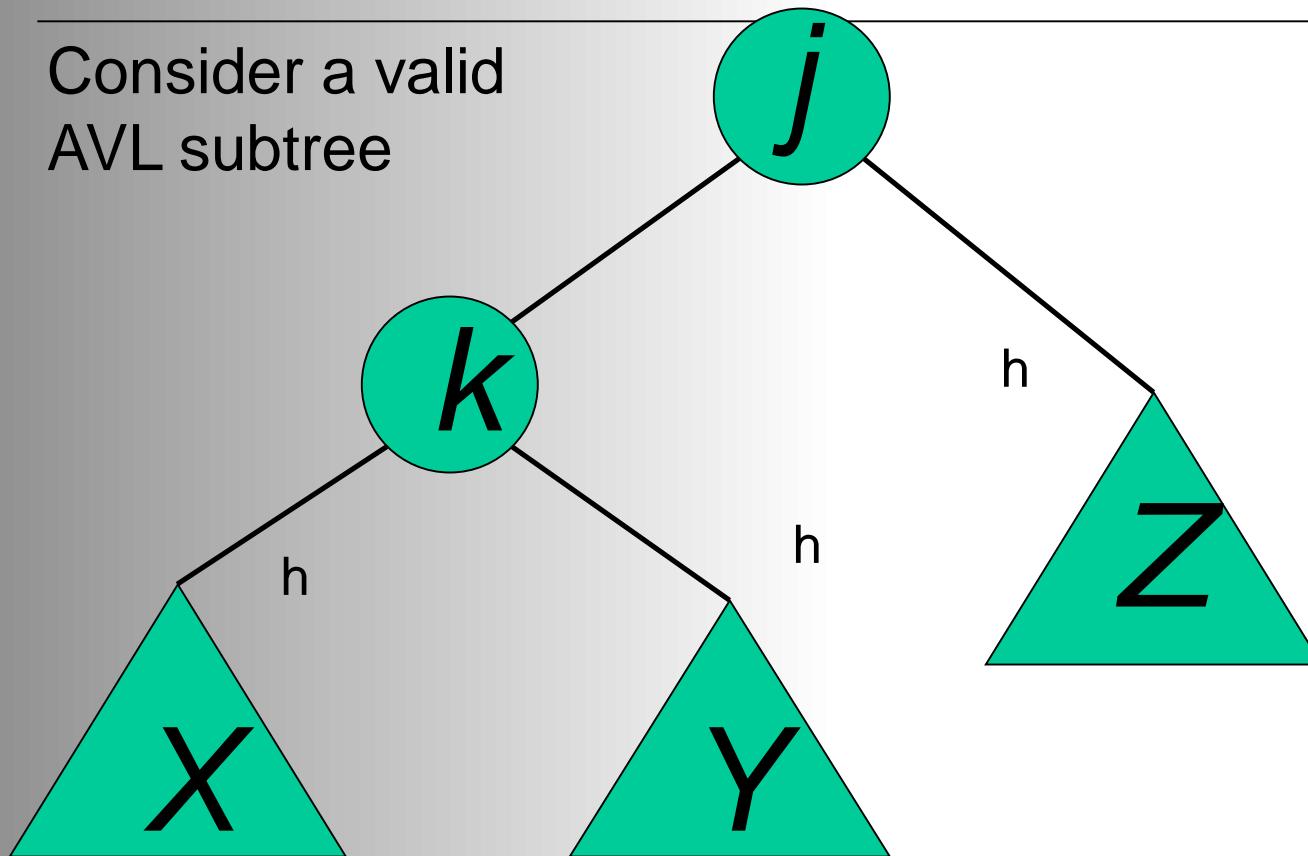
**Inside Cases** (require double rotation) :

3. Insertion into **right** subtree **of left** child of  $\alpha$ .
4. Insertion into **left** subtree **of right** child of  $\alpha$ .

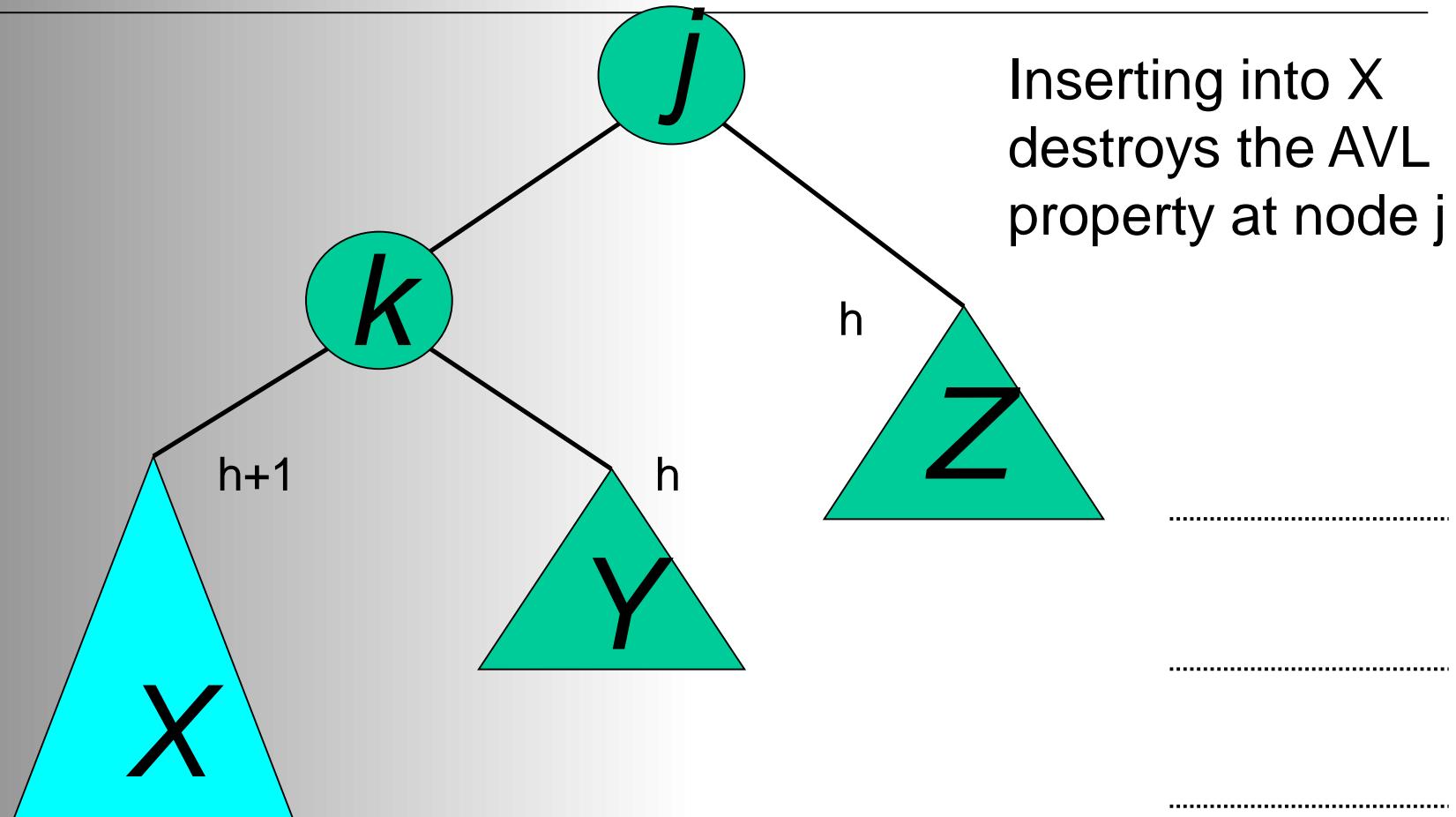
The rebalancing is performed through four separate rotation algorithms.

# AVL Insertion: Outside Case

Consider a valid  
AVL subtree



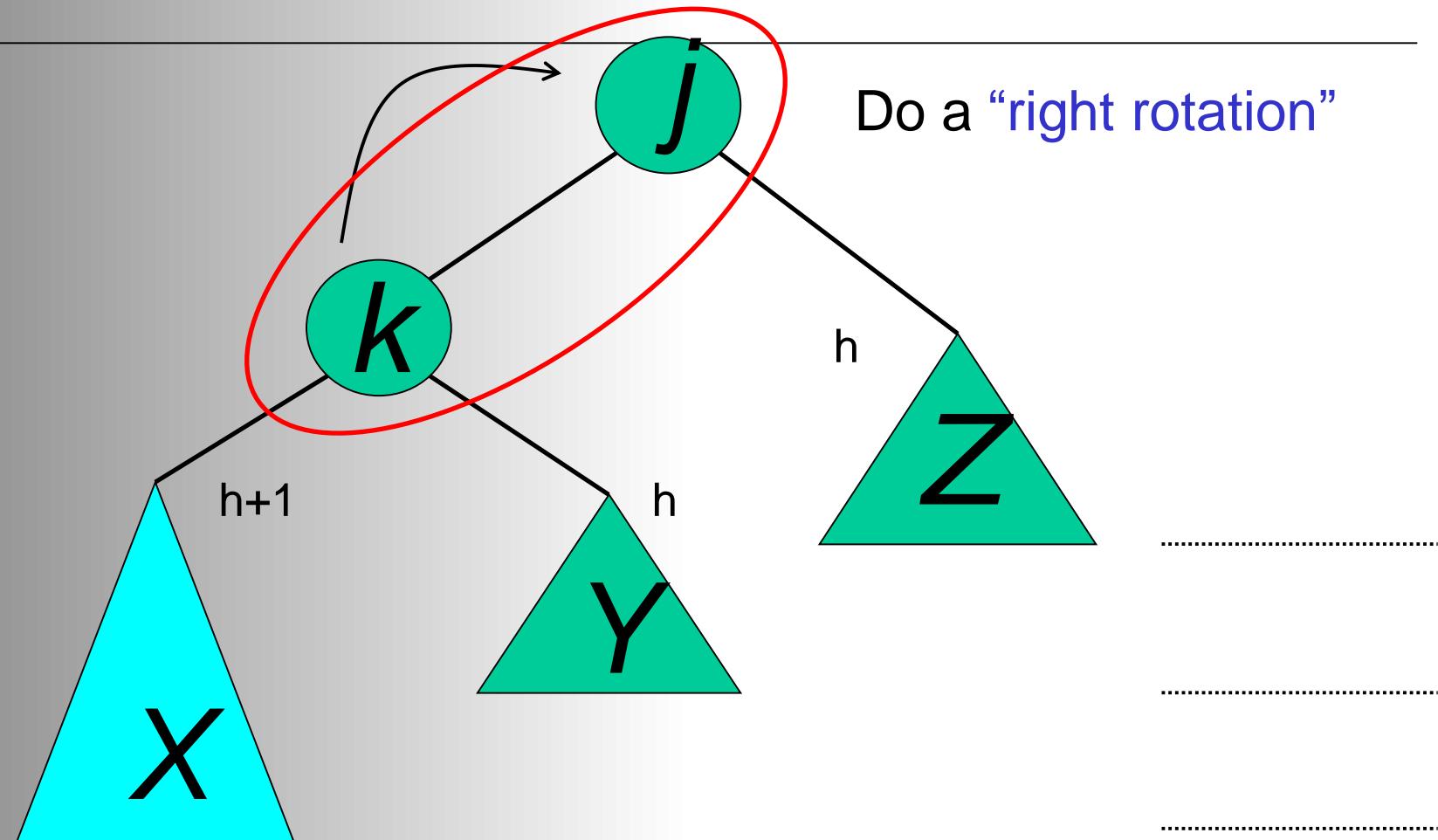
# AVL Insertion: Outside Case



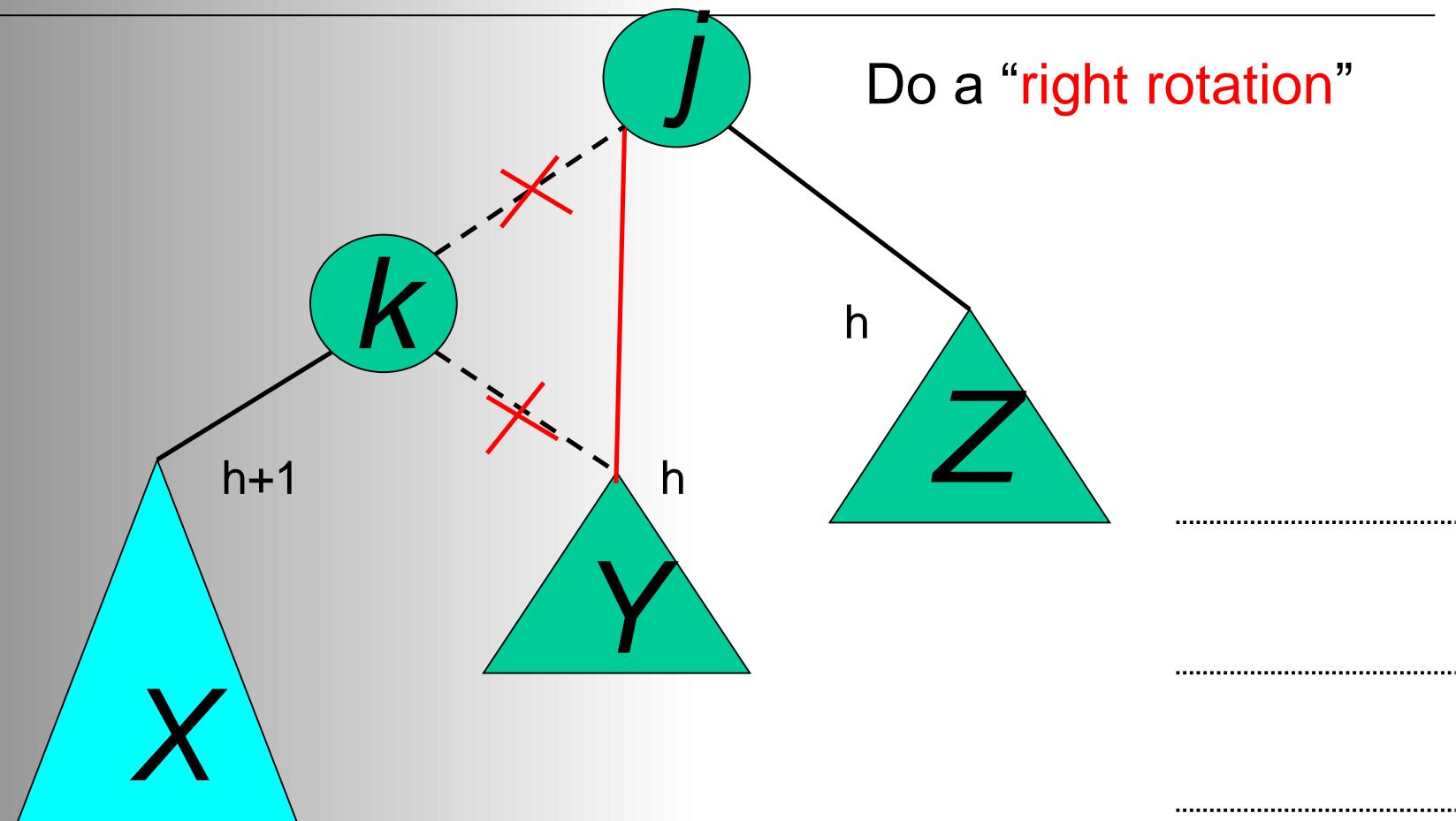
# WORK SPACE

---

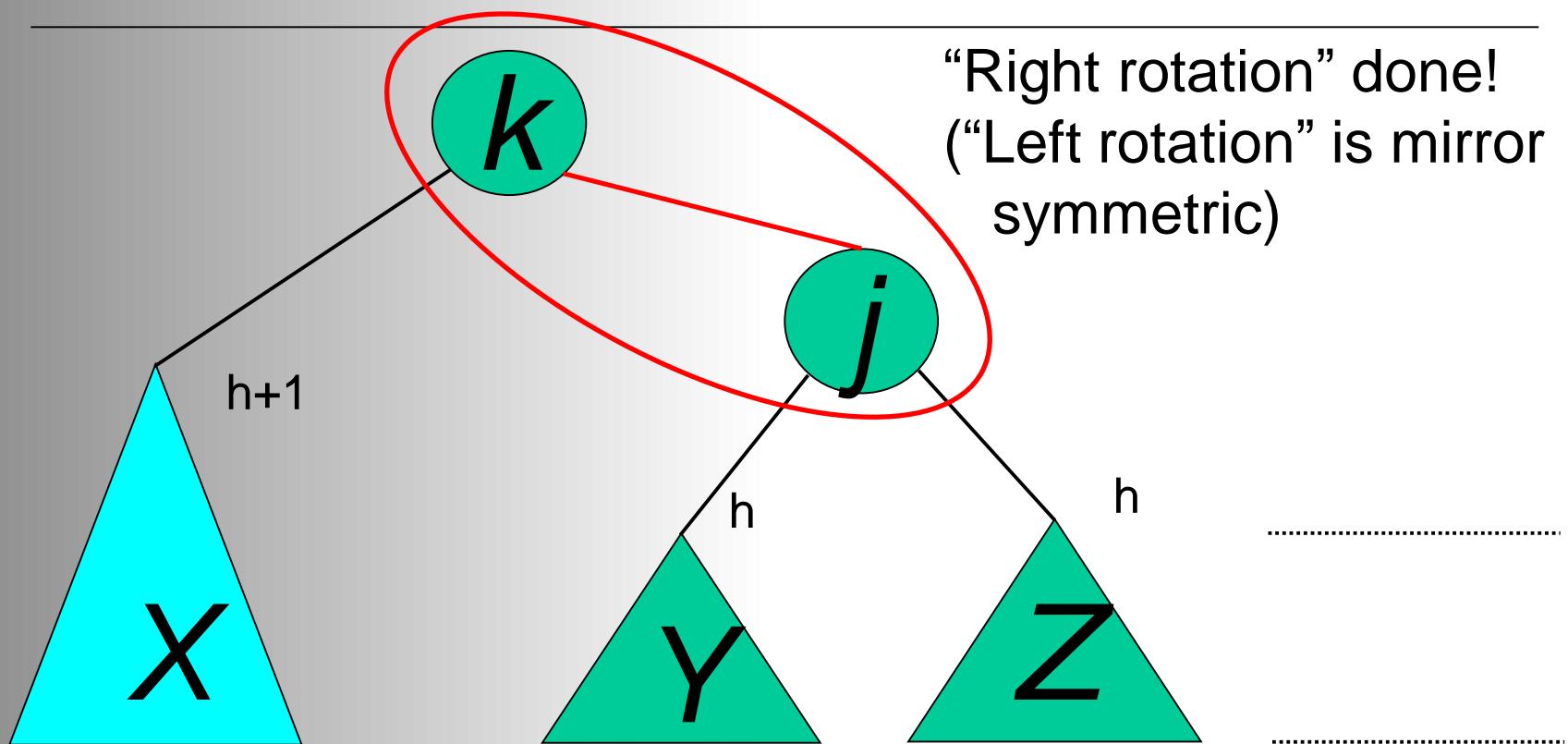
# AVL Insertion: Outside Case



# Single right rotation



# Outside Case Completed



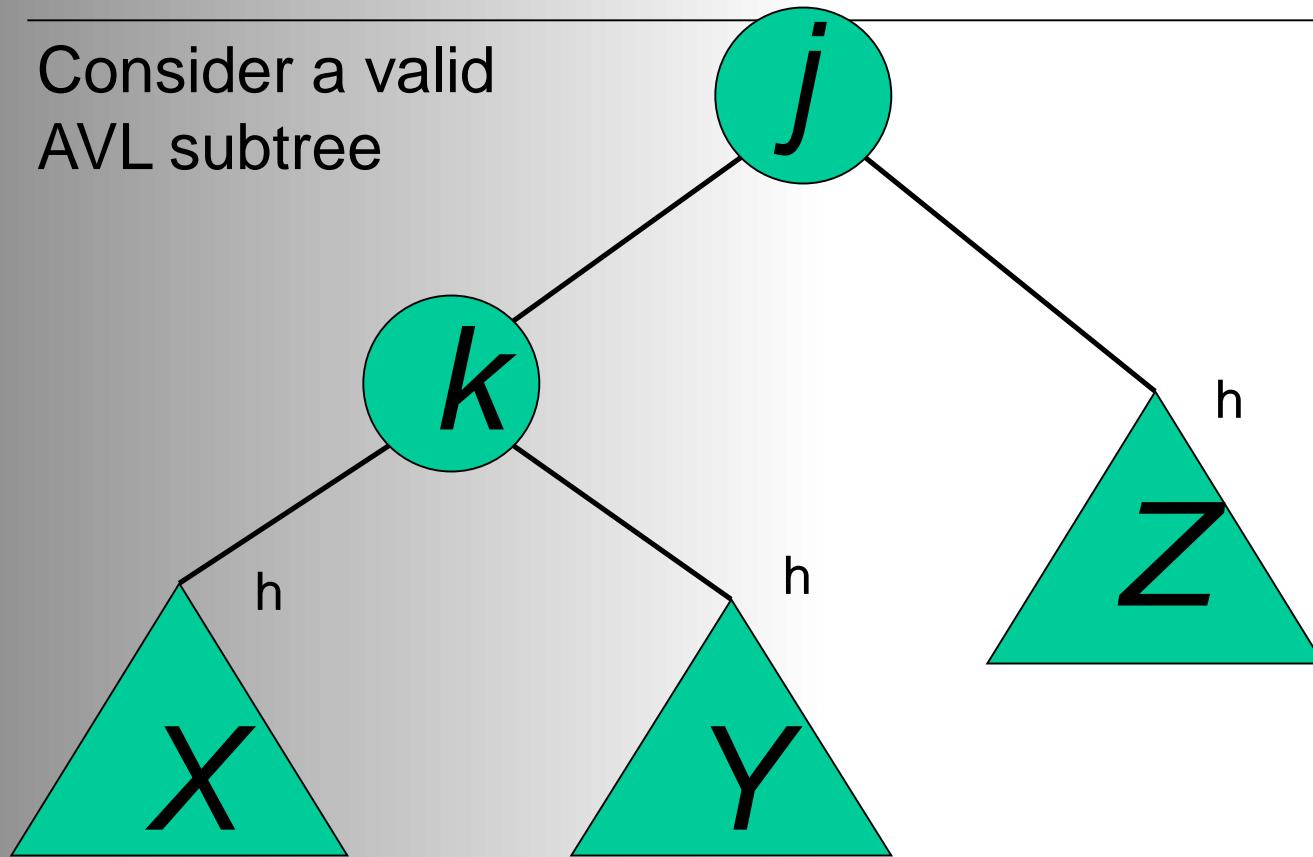
AVL property has been restored!

# WORK SPACE

---

# AVL Insertion: Inside Case

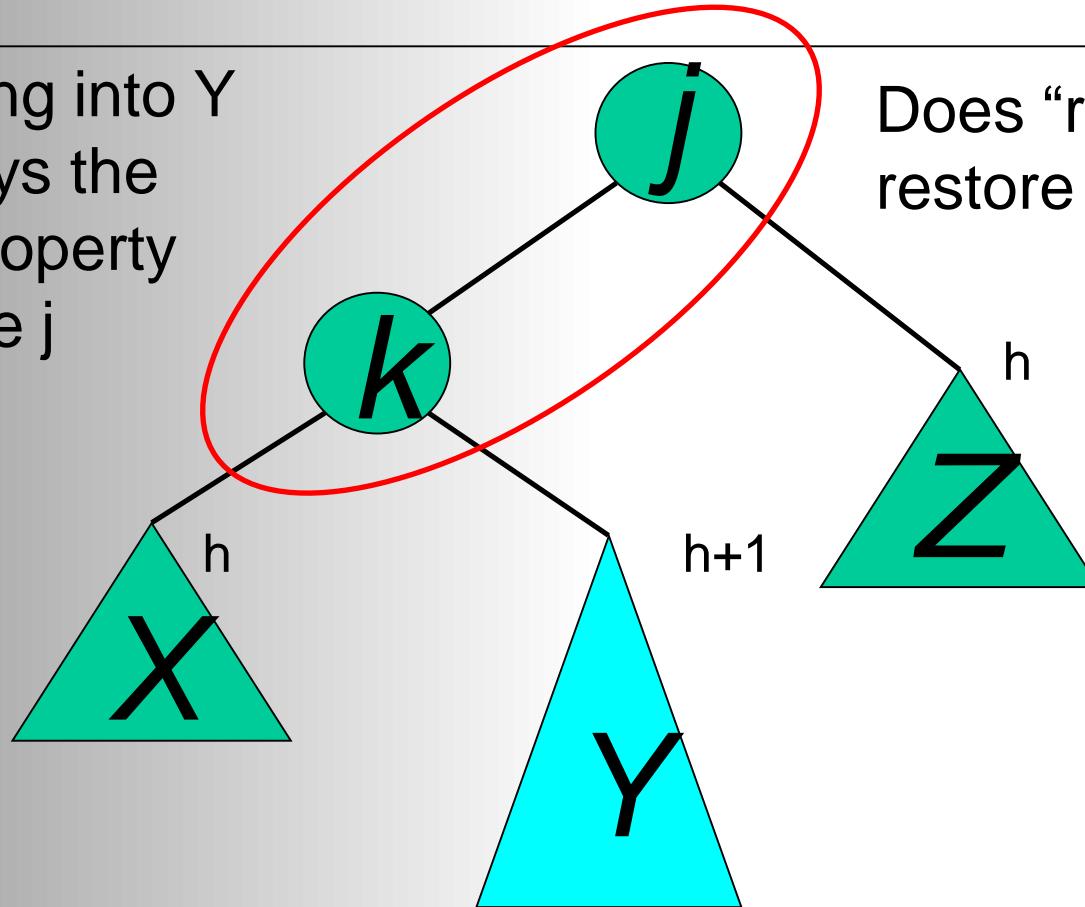
Consider a valid  
AVL subtree



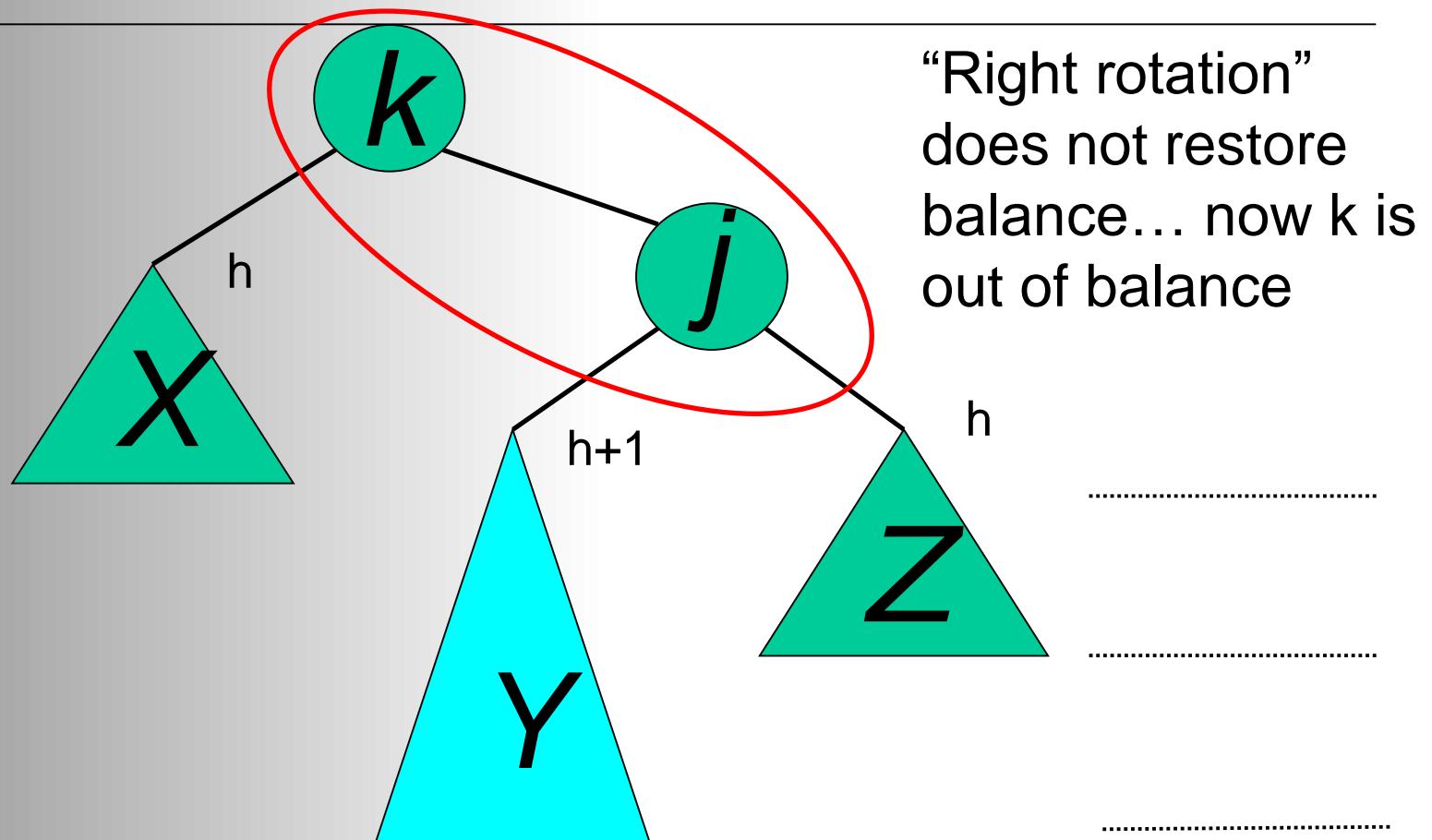
# AVL Insertion: Inside Case

Inserting into Y  
destroys the  
AVL property  
at node j

Does “right rotation”  
restore balance?

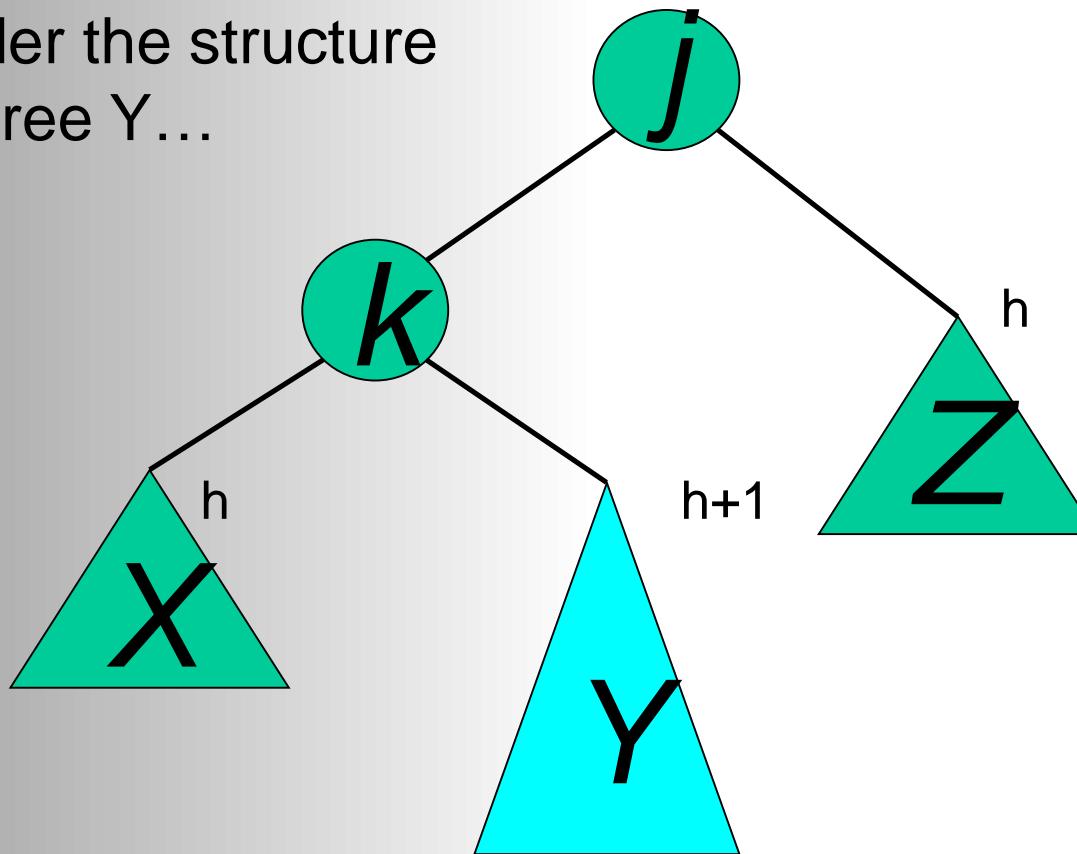


# AVL Insertion: Inside Case



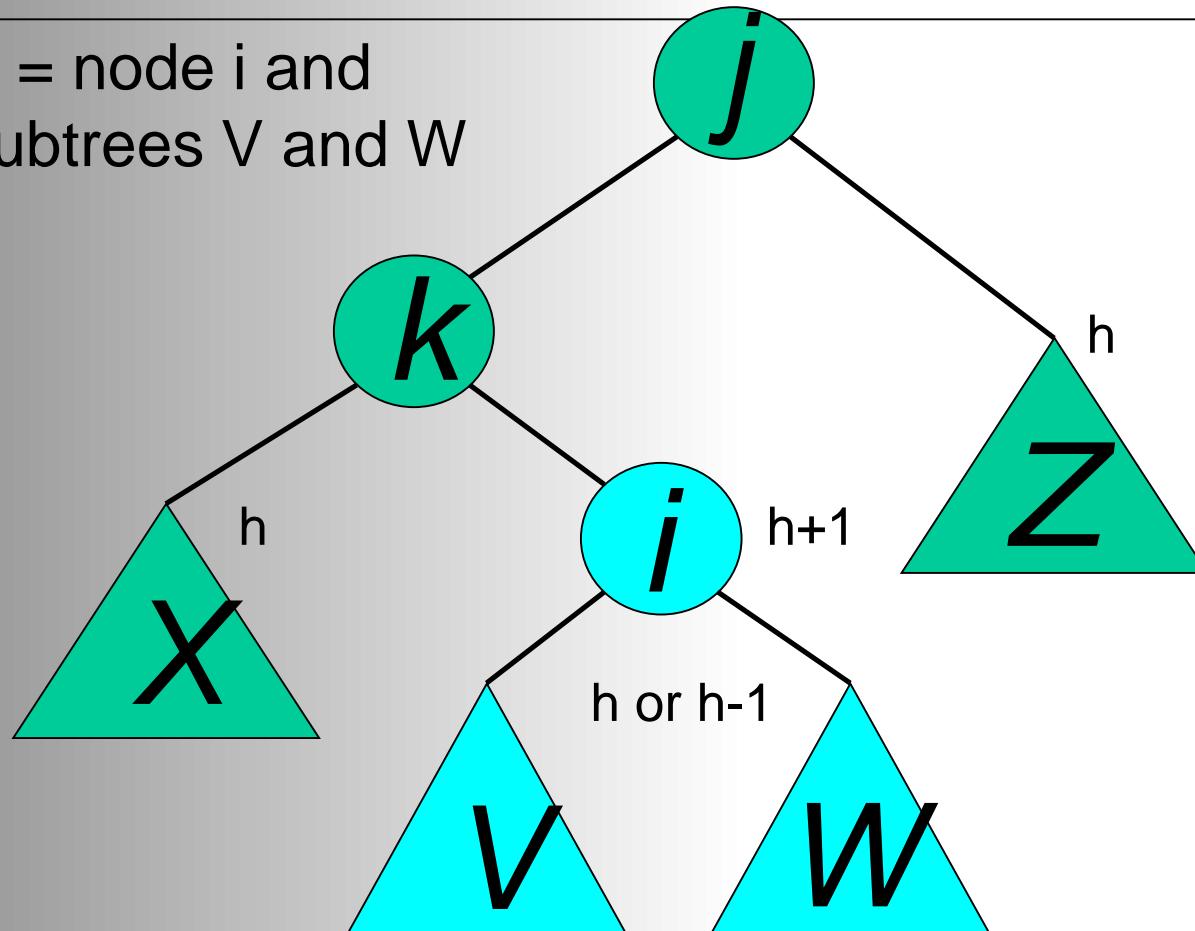
# AVL Insertion: Inside Case

Consider the structure  
of subtree Y...

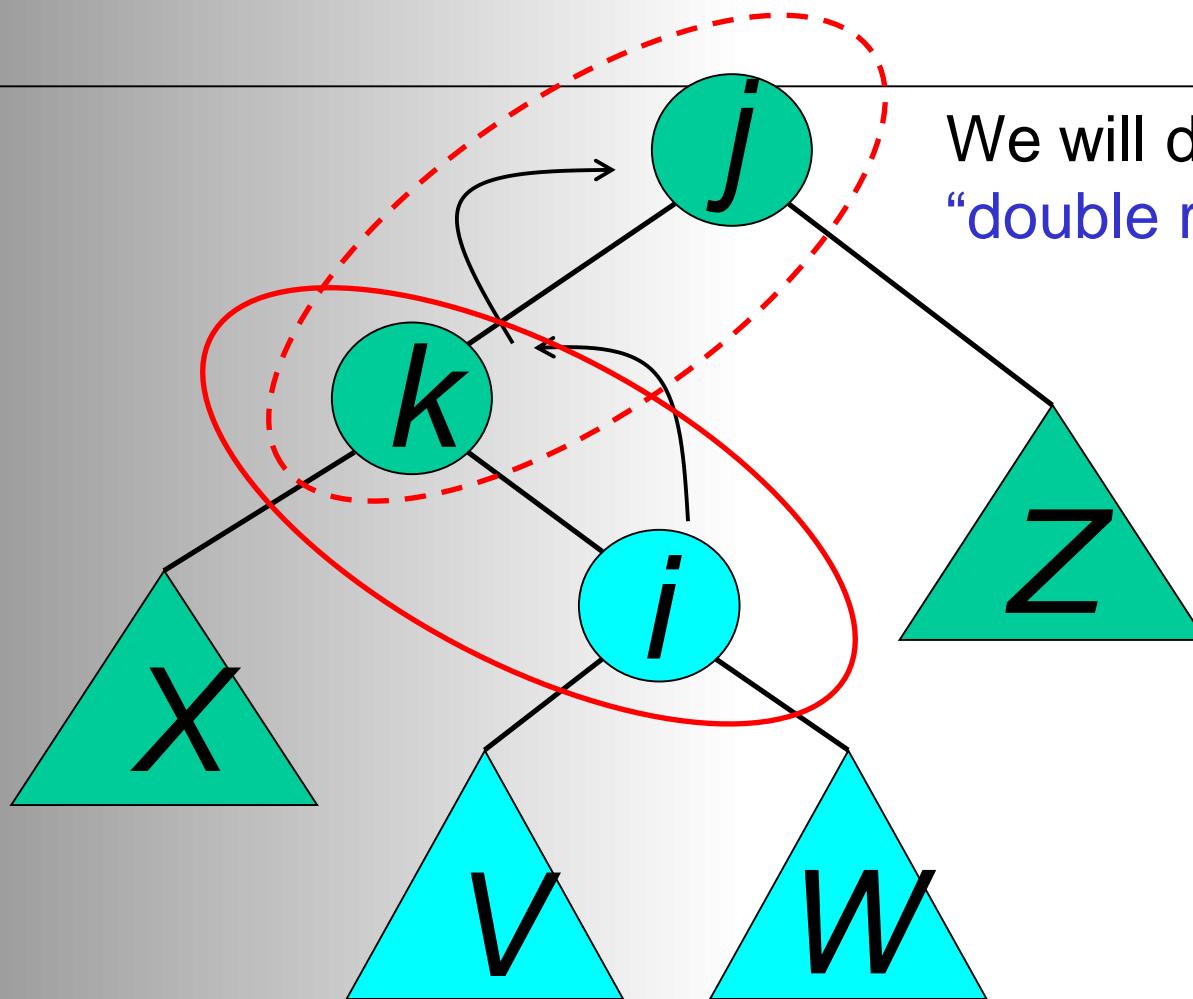


# AVL Insertion: Inside Case

Y = node i and  
subtrees V and W

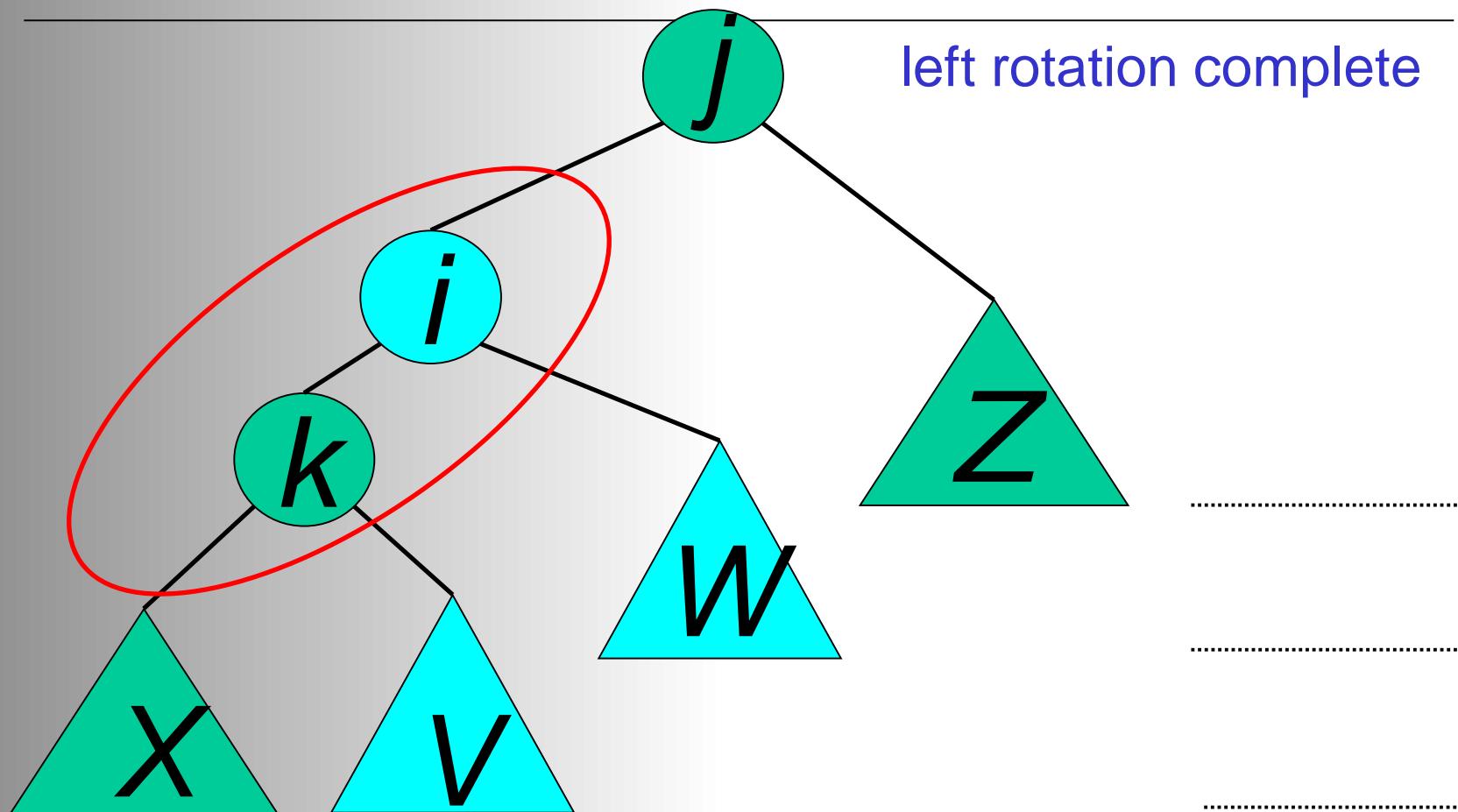


# AVL Insertion: Inside Case

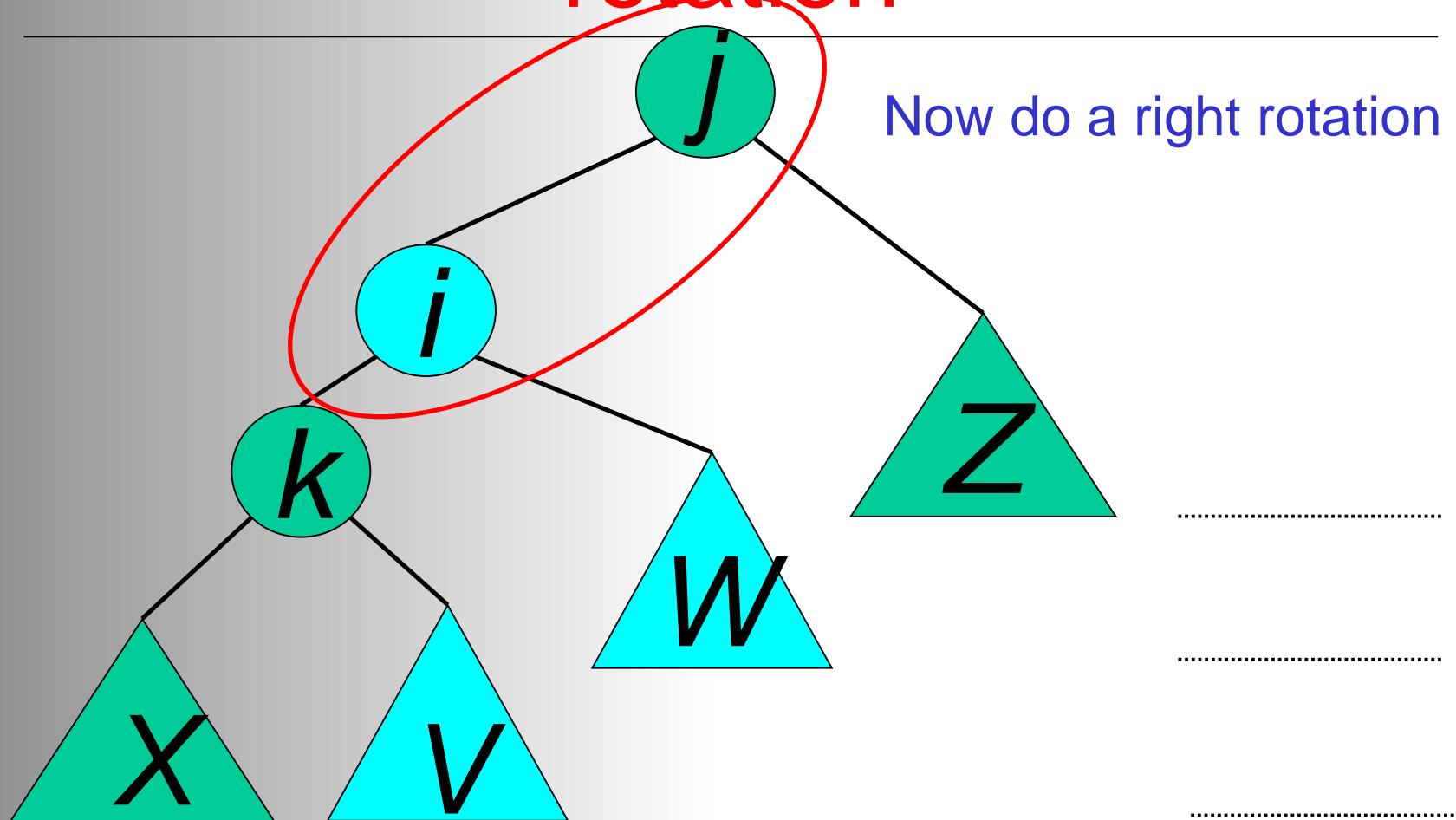


We will do a **left-right**  
“double rotation” . . .

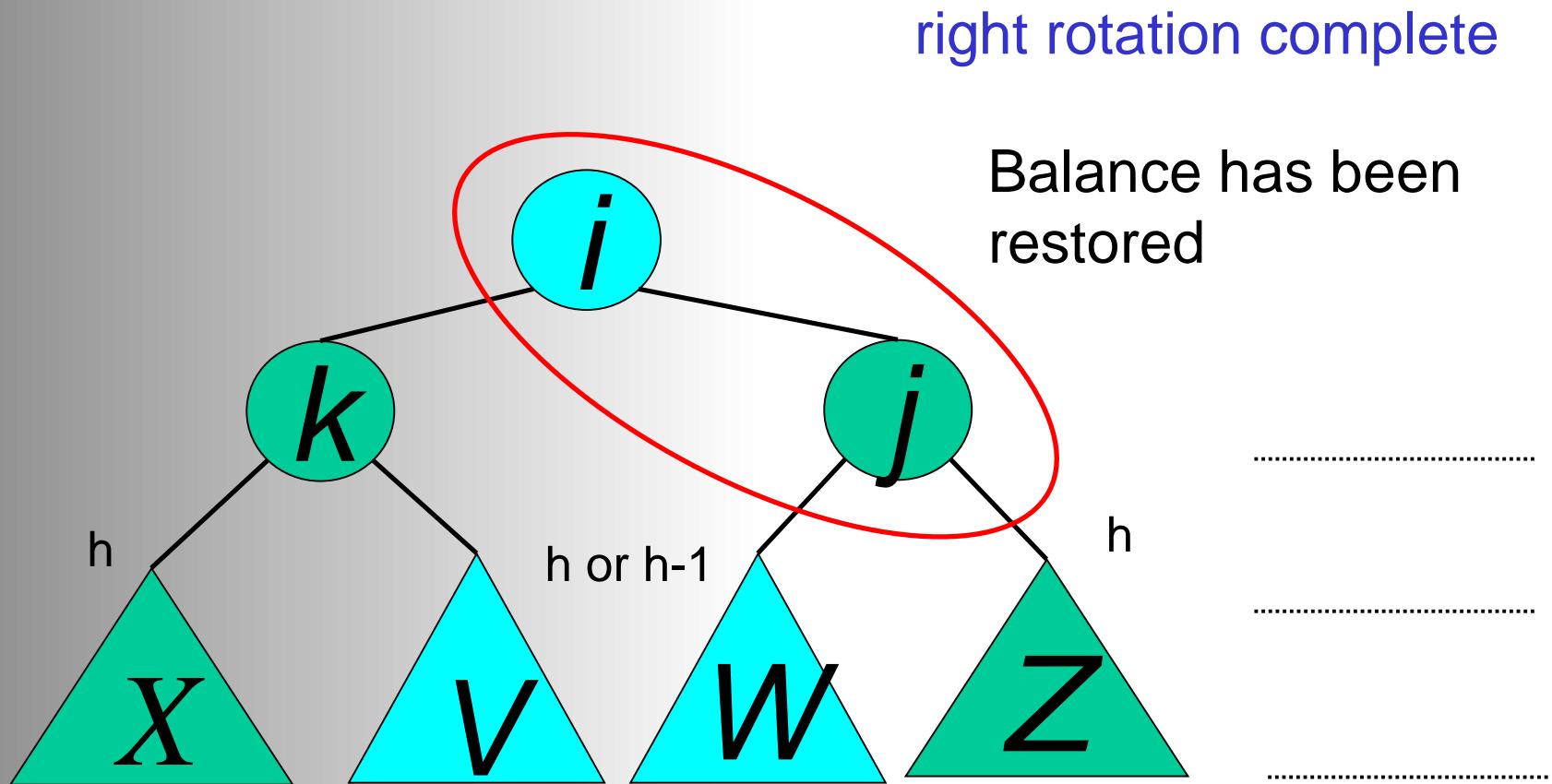
# Double rotation : first rotation



# Double rotation : second rotation



# Double rotation : second rotation



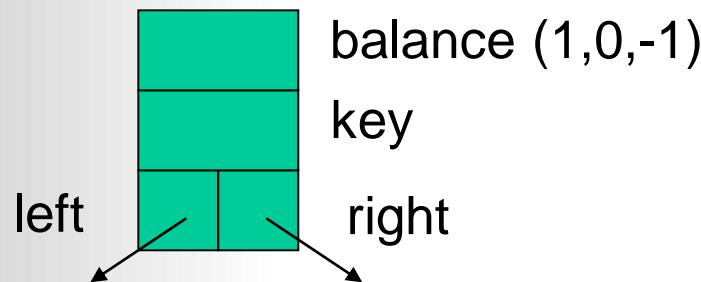
# Exercise

---

- Construct an AVL Search Tree by inserting the following elements:
- 50, 20, 80, 10, 30, 5, 15, 17, 19, 14, 16, 18
- F, C, E, T, J, Z, D, B, A, Y

# Implementation

---



No need to keep the height; just the difference in height, i.e. the **balance** factor; this has to be modified on the path of insertion even if you don't perform rotations

Once you have performed a rotation (single or double) you won't need to go back up the tree

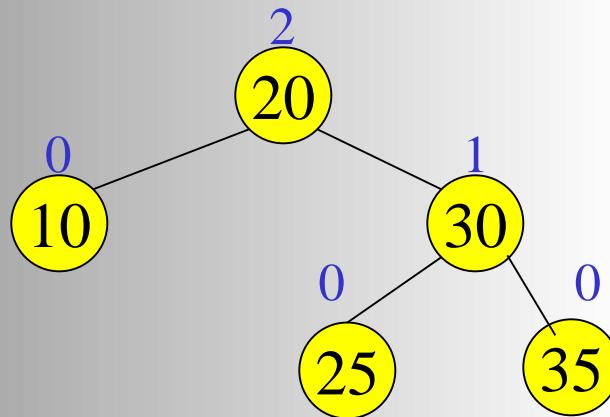
# Insertion in AVL Trees

---

- Insert at the leaf (as for all BST)
  - › only nodes on the path from insertion point to root node have possibly changed in height
  - › So after the Insert, go back up to the root node by node, updating heights
  - › If a new balance factor (the difference  $h_{\text{left}} - h_{\text{right}}$ ) is 2 or  $-2$ , adjust tree by *rotation* around the node

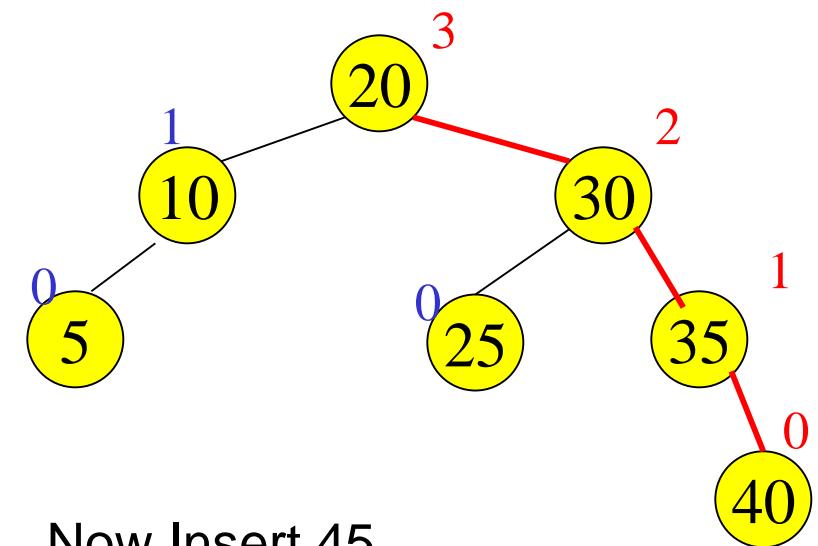
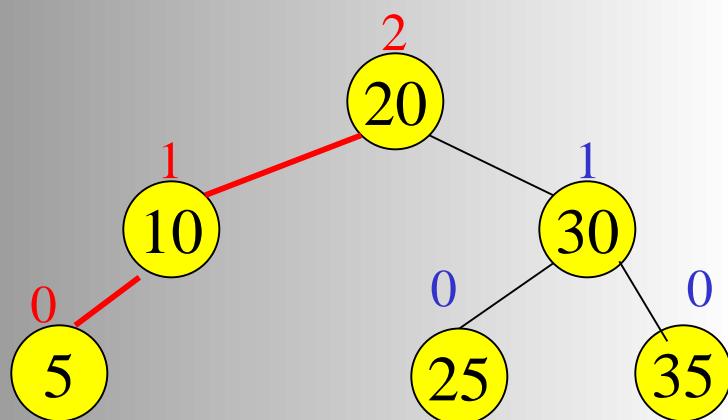
# Example of Insertions in an AVL Tree

---

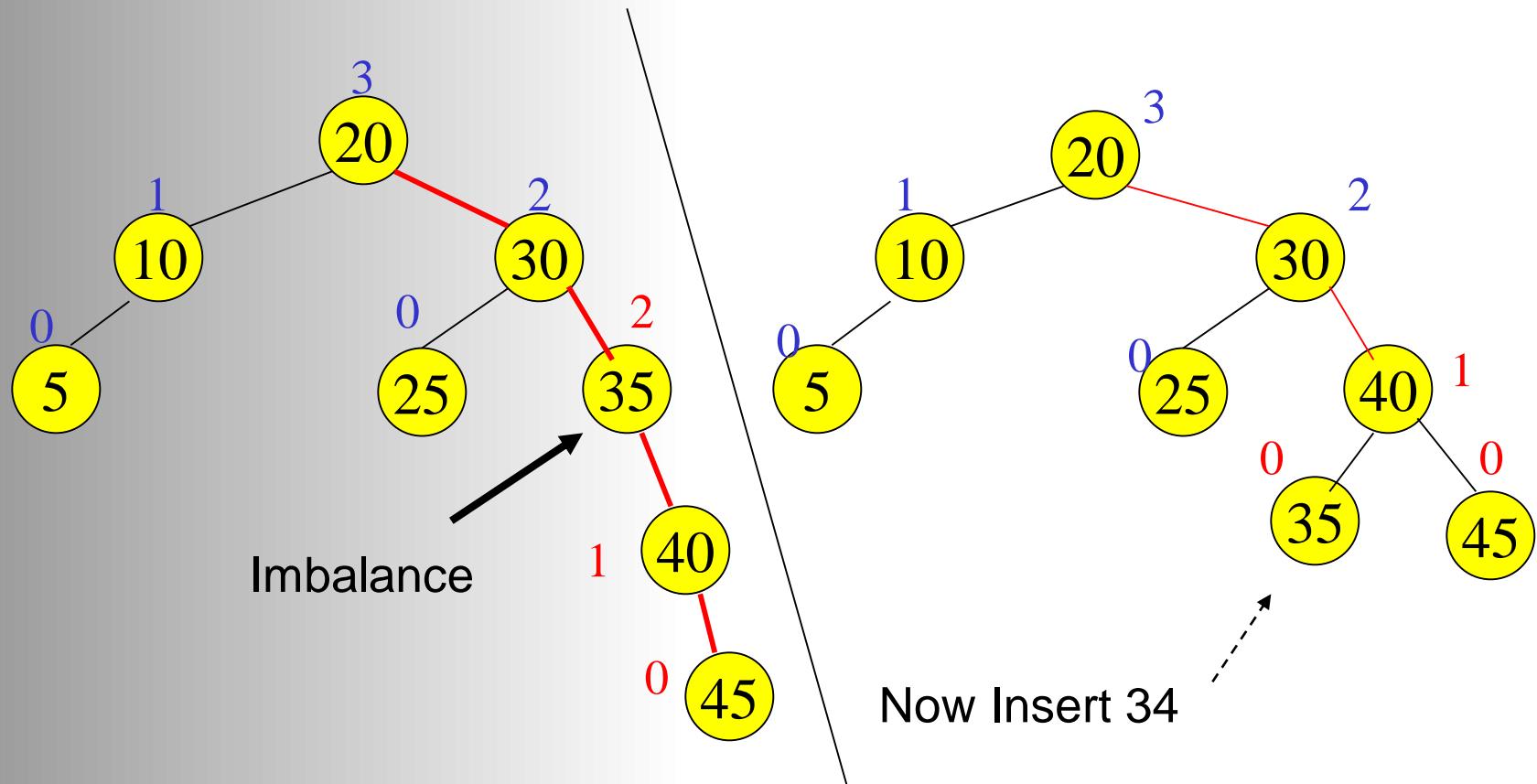


Insert 5, 40

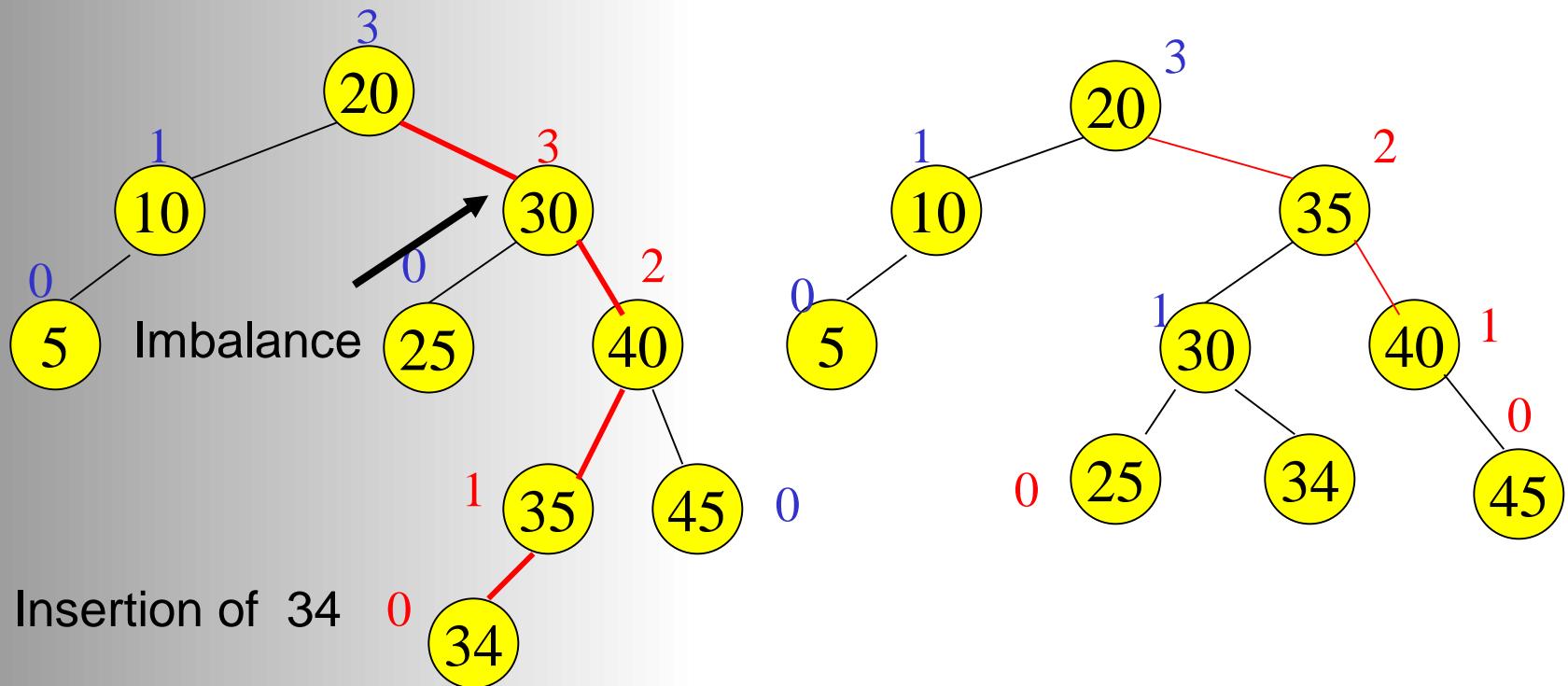
# Example of Insertions in an AVL Tree



# Single rotation (outside case)



# Double rotation (inside case)



# Pros and Cons of AVL Trees

---

## Arguments for AVL trees:

1. Search is  $O(\log N)$  since AVL trees are **always balanced**.
2. Insertion and deletions are also  $O(\log n)$
3. The height balancing adds no more than a constant factor to the speed of insertion.

## Arguments against using AVL trees:

1. Difficult to program & debug; more space for balance factor.
2. Asymptotically faster but rebalancing costs time.
3. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).

# Data Structures

Topic: Deletion in AVL Tree



By  
**Ravi Kant Sahu**

*Asst. Professor,*

Lovely Professional University, Punjab



# Contents

- Introduction
- R0, R1 and R-1 Rotation
- L0, L1 and L-1 Rotation



# Introduction

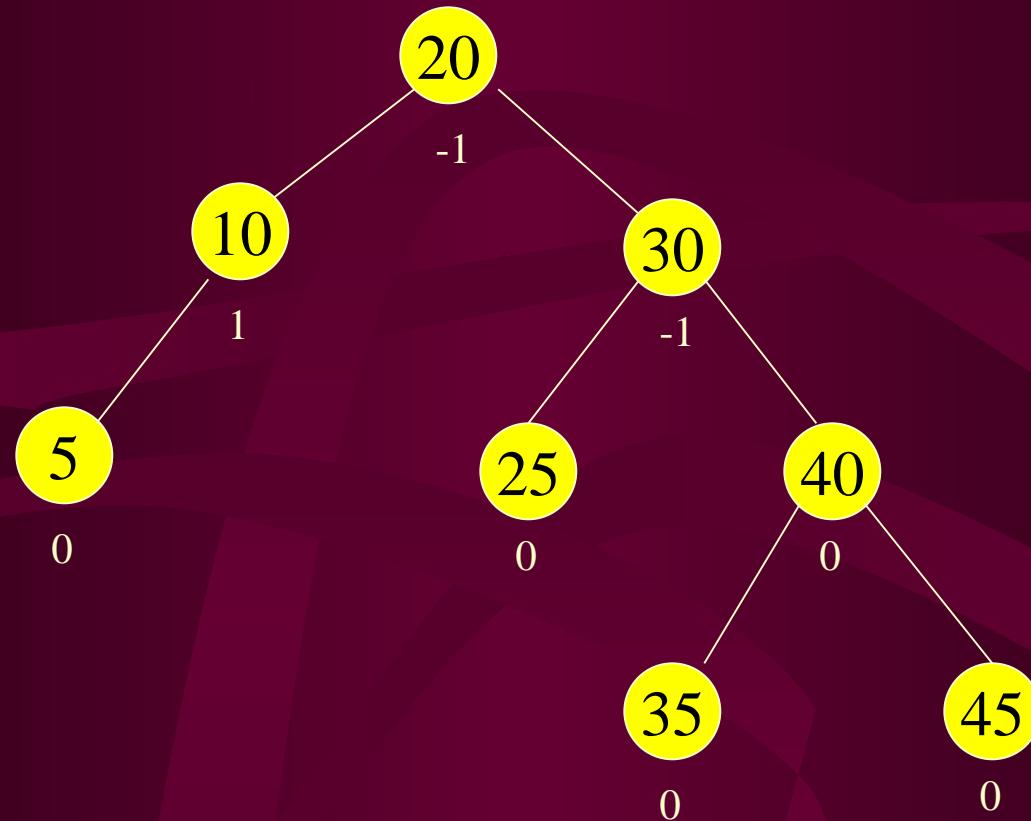
- An element can be deleted from AVL tree which may change the BF of a node such that it results in unbalanced tree.
- Some rotations will be applied on AVL tree to balance it.
- R rotation is applied if the deleted node is in the right subtree of node A (A is the node with balance factor other than 0, 1 and -1).
- L rotation is applied if the deleted node is in the left subtree of node A.

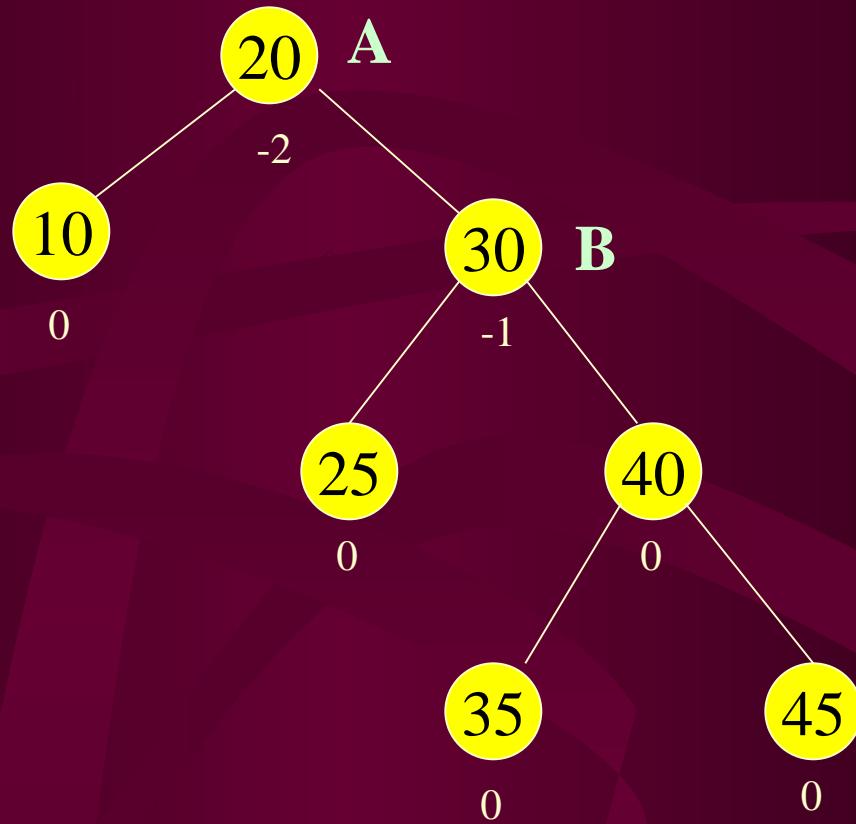


# Introduction

- Suppose we have deleted node X from the tree.
- A is the closest ancestor node on the path from X to the root node, with a balance factor -2 or +2.
- B is the desendent of node A on the opposite subtree of deleted node i.e. if the deleted node is on left side the B is the desendent on the right subtree of A or the root of right subtree of A.

# Delete 5







# R Rotation

- R Rotation is applied when the deleted node is in the right subtree of node A.
- There are three different types of rotations based on the balanced factor of node B.
- **R0 Rotation:** When the balance Factor of node B is 0.
  - Apply LL Rotation on node A.
- **R1 Rotation:** When the balance Factor of node B is +1.
  - Apply LL Rotation on node A.
- **R-1 Rotation:** When the balance Factor of node B is -1.
  - Apply LR Rotation(RR rotation on B and LL rotation on node A).

# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# L Rotation

- L Rotation is applied when the deleted node is in the left subtree of node A.
- There are three different types of rotations based on the balanced factor of node B.
- **L0 Rotation:** When the balance Factor of node B is 0.
  - Apply RR Rotation on node A.
- **L-1 Rotation:** When the balance Factor of node B is +1.
  - Apply RR Rotation on node A.
- **L1 Rotation:** When the balance Factor of node B is -1.
  - Apply RL Rotation(LL rotation on B and RR rotation on node A).

# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India

# Important

- Unlike insertion, fixing the node A won't fix the complete AVL tree.
- After fixing A, we may have to fix ancestors of A as well.

# Important

Given the sequential representation of an AVL Tree:

K, G, P, C, H, L, S, A, F, \_, I, \_, M, \_, \_, \_, \_, D

[Where ‘\_’ represents NULL tree]

How many LL and RR rotations will be required if we  
Delete S?

- A. 1 LL and 1 RR
- B. 2 LL and 1 RR
- C. 1 LL and 2 RR
- D. 2 LL and 2 RR



# Questions

# Data Structures

---

Topic: Huffman Coding

By  
**Ravi Kant Sahu**  
Asst. Professor



**Lovely Professional University, Punjab**

---

---

# Huffman Coding: An Application of Binary Trees and Priority Queues

---

# Purpose of Huffman Coding

---

- Proposed by Dr. David A. Huffman in 1952
  - “*A Method for the Construction of Minimum Redundancy Codes*”
- Applicable to many forms of data transmission
  - Example: text files

# The Basic Algorithm

---

- Huffman coding is a form of statistical coding
  - Not all characters occur with the same frequency!
  - Yet all characters are allocated the same amount of space
- 1 char = 1 byte, be it **e** or **x**

# The Basic Algorithm

---

- Any savings in tailoring codes to frequency of character?
- Code word lengths are no longer fixed like ASCII.
- Code word lengths vary and will be shorter for the more frequently used characters.

# Work Space

---

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)

# The (Real) Basic Algorithm

---

1. Scan text to be compressed and tally occurrence of all characters.
  2. Sort or prioritize characters based on number of occurrences in text.
  3. Build Huffman code tree based on prioritized list.
  4. Perform a traversal of tree to determine all code words.
  5. Scan text again and create new file using the Huffman codes.
-

# Building a Tree

## Scan the original text

---

- Consider the following short text:

*Eerie eyes seen near lake.*

- Count up the occurrences of all characters in the text

# Work Space

---

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)

# Building a Tree

## Scan the original text

---

*Eerie eyes seen near lake.*

- What characters are present?

E e r i space  
y s n a r l k .

# Building a Tree

Scan the original text

---

Eerie eyes seen near lake.

- What is the frequency of each character in the text?

| Char  | Freq. | Char | Freq. | Char | Freq. |
|-------|-------|------|-------|------|-------|
| E     | 1     | y    | 1     | k    | 1     |
| e     | 8     | s    | 2     | .    | 1     |
| r     | 2     | n    | 2     |      |       |
| i     | 1     | a    | 2     |      |       |
| space | 4     | l    | 1     |      |       |

# Building a Tree

## Prioritize characters

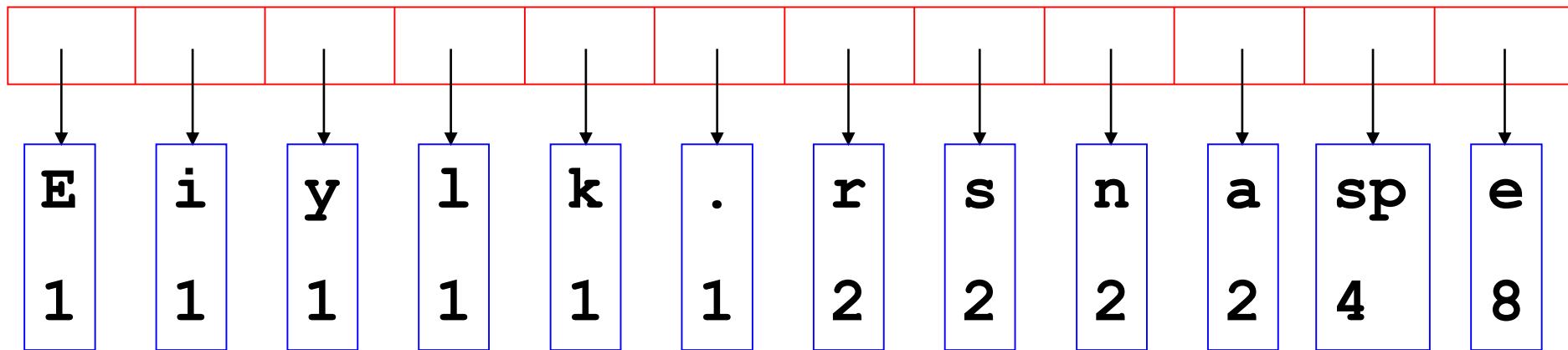
---

- Create binary tree nodes with character and frequency of each character
- Place nodes in a priority queue
  - The lower the occurrence, the higher the priority in the queue

# Building a Tree

---

- The queue after inserting all nodes



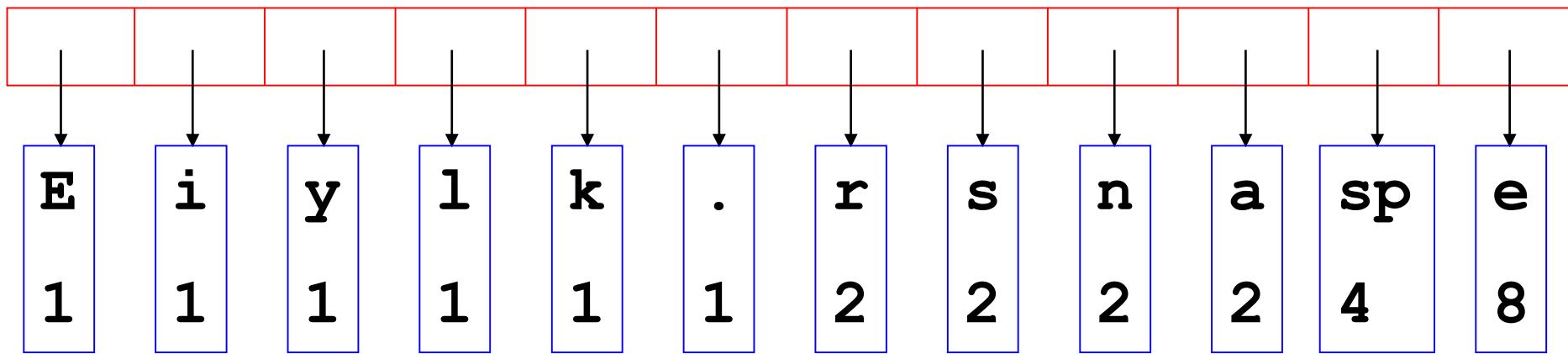
# Building a Tree

---

- While priority queue contains two or more nodes
    - Create new node
    - Dequeue node and make it left subtree
    - Dequeue next node and make it right subtree
    - Frequency of new node equals sum of frequency of left and right children
    - Enqueue new node back into queue
-

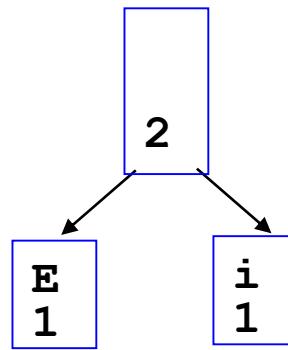
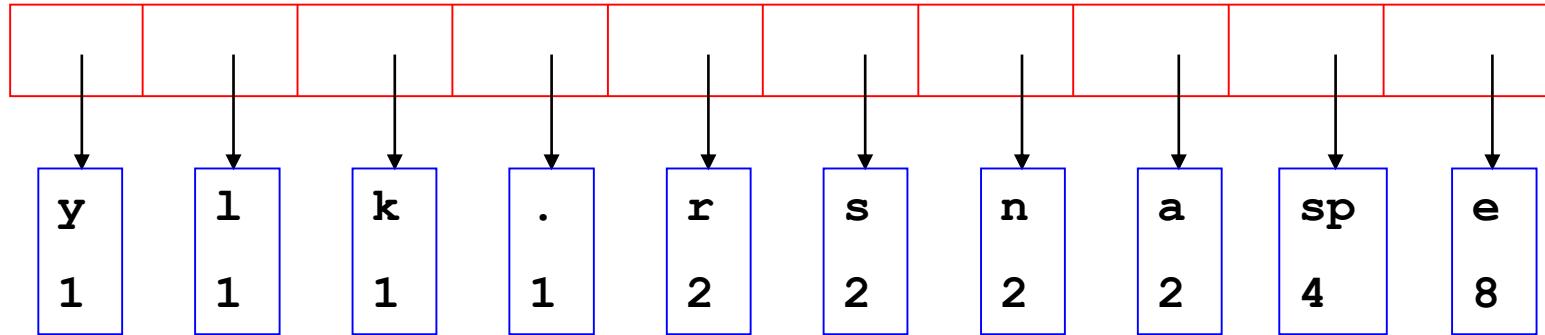
# Building a Tree

---



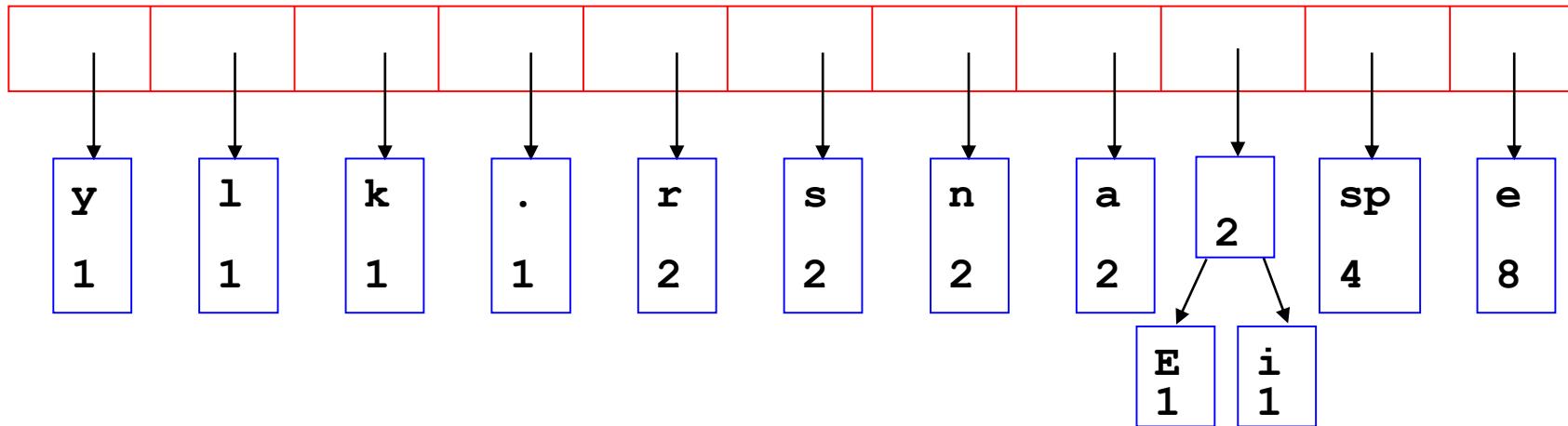
# Building a Tree

---



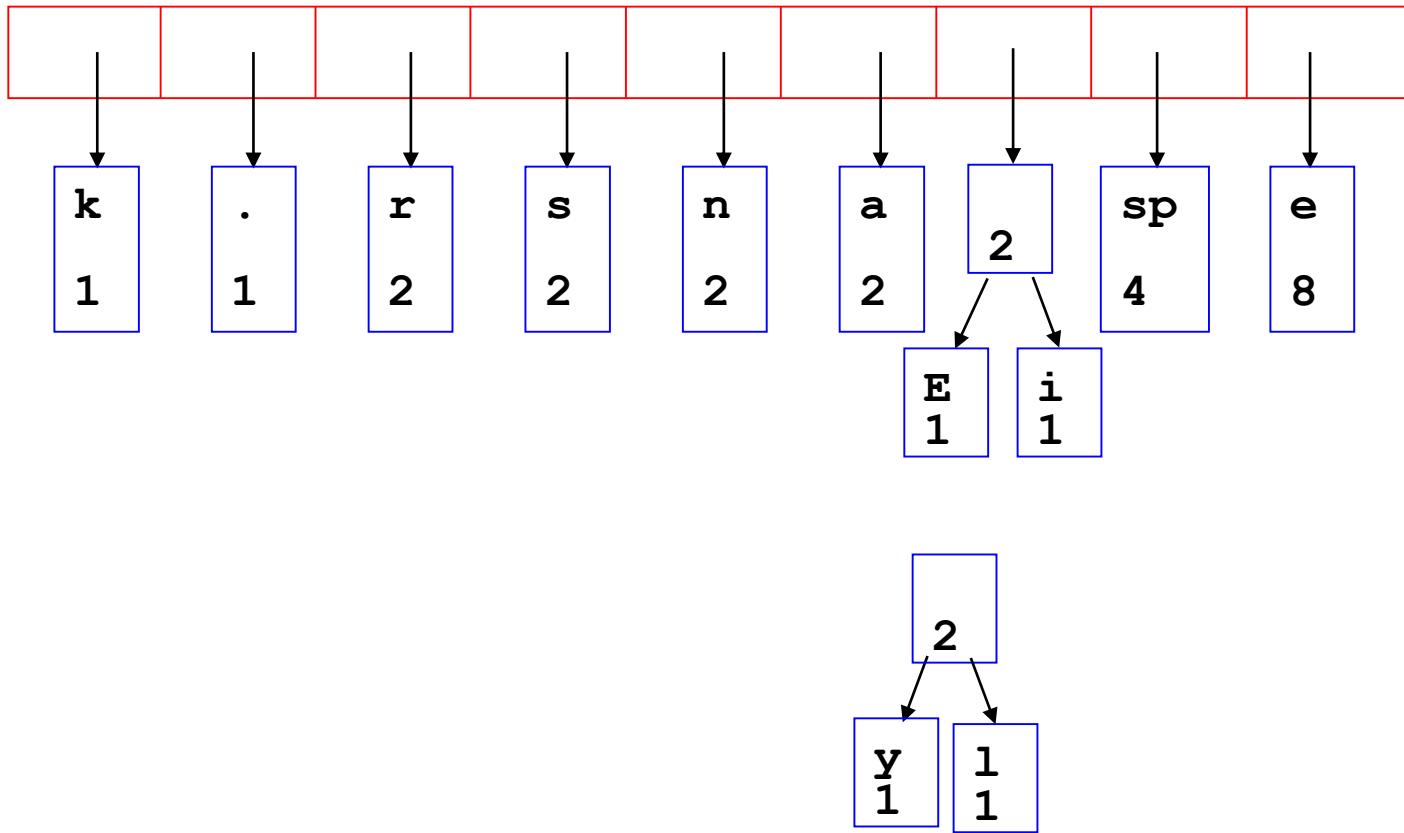
# Building a Tree

---



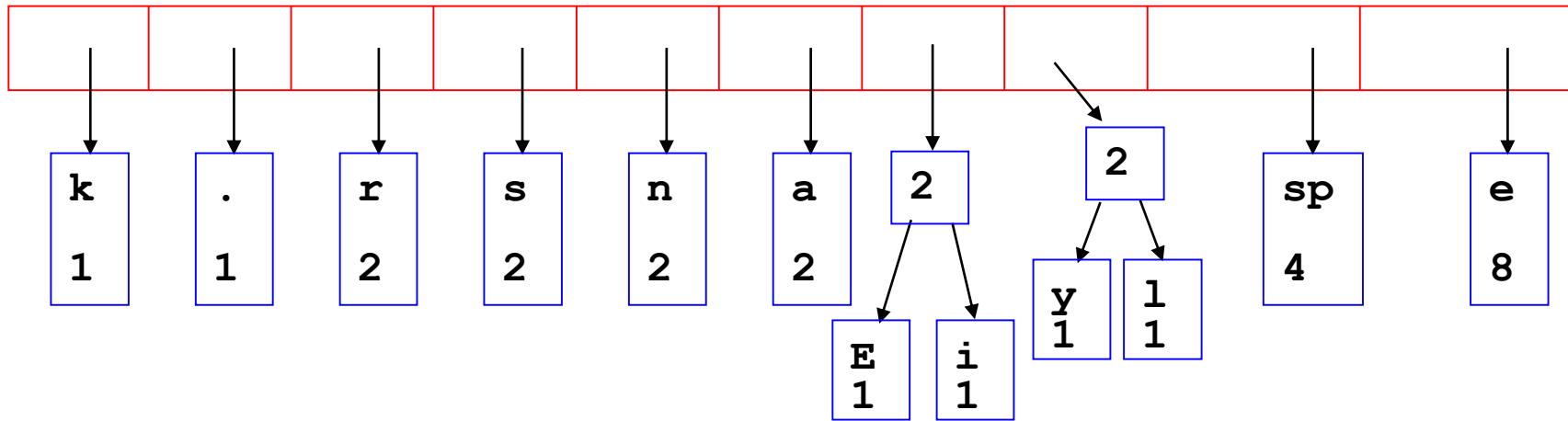
# Building a Tree

---



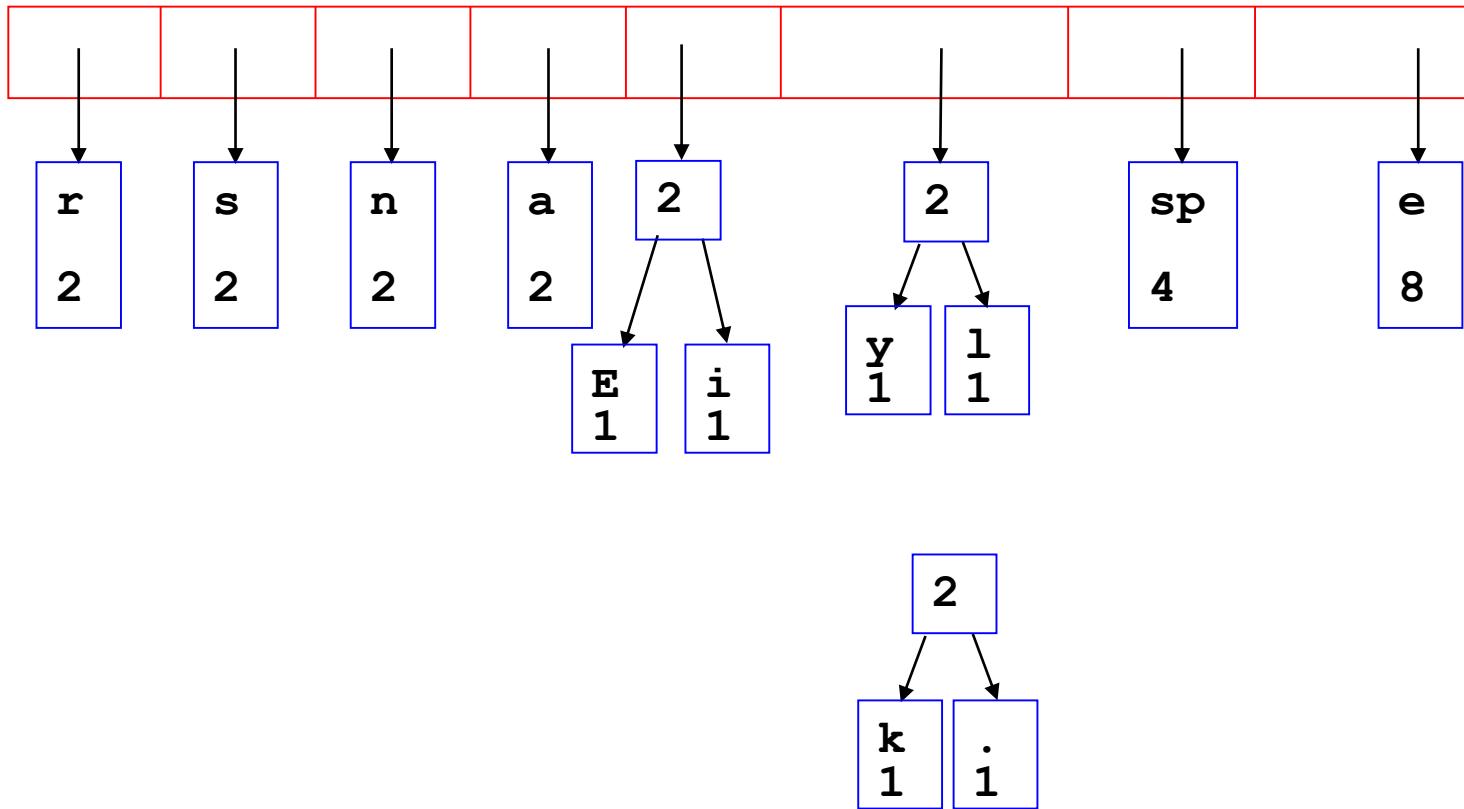
# Building a Tree

---



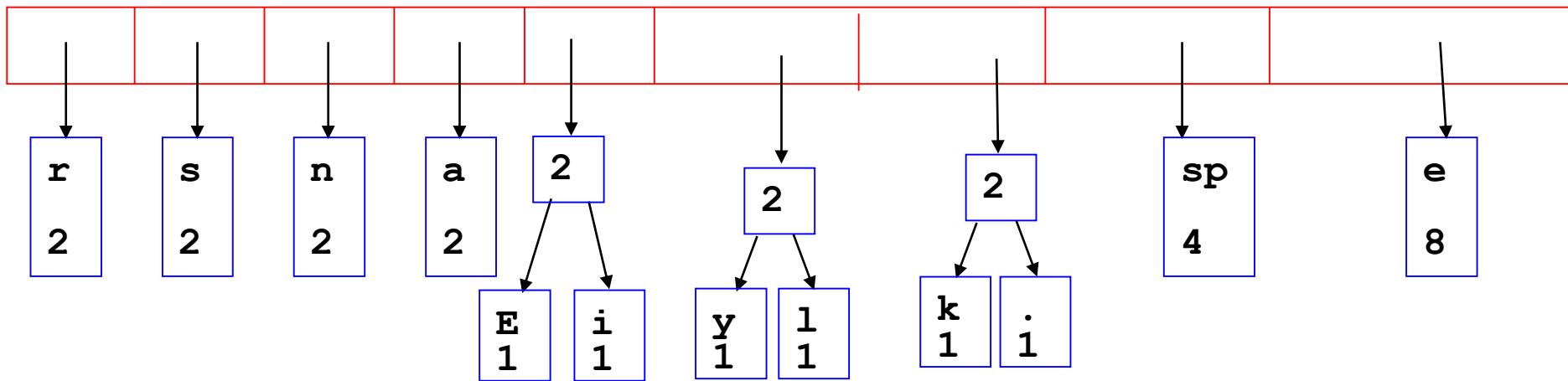
# Building a Tree

---



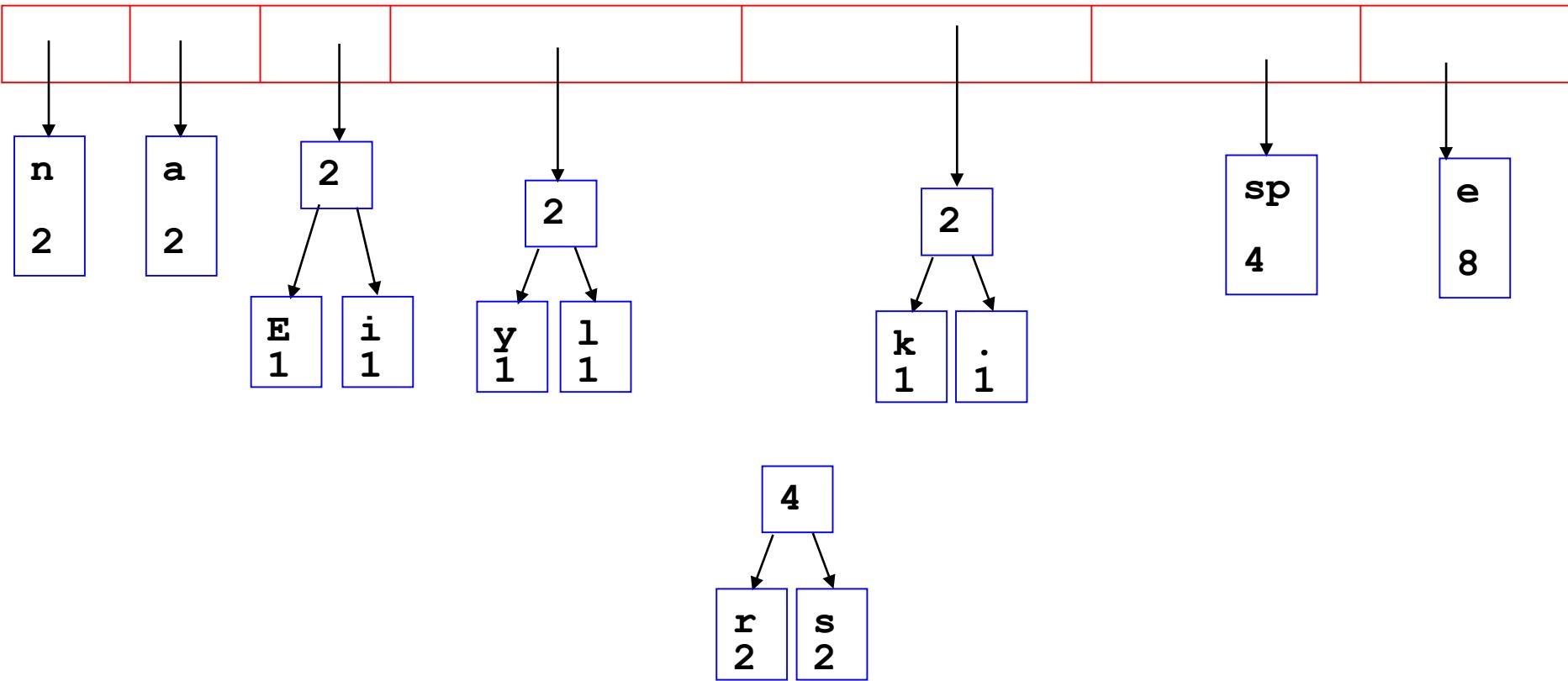
# Building a Tree

---



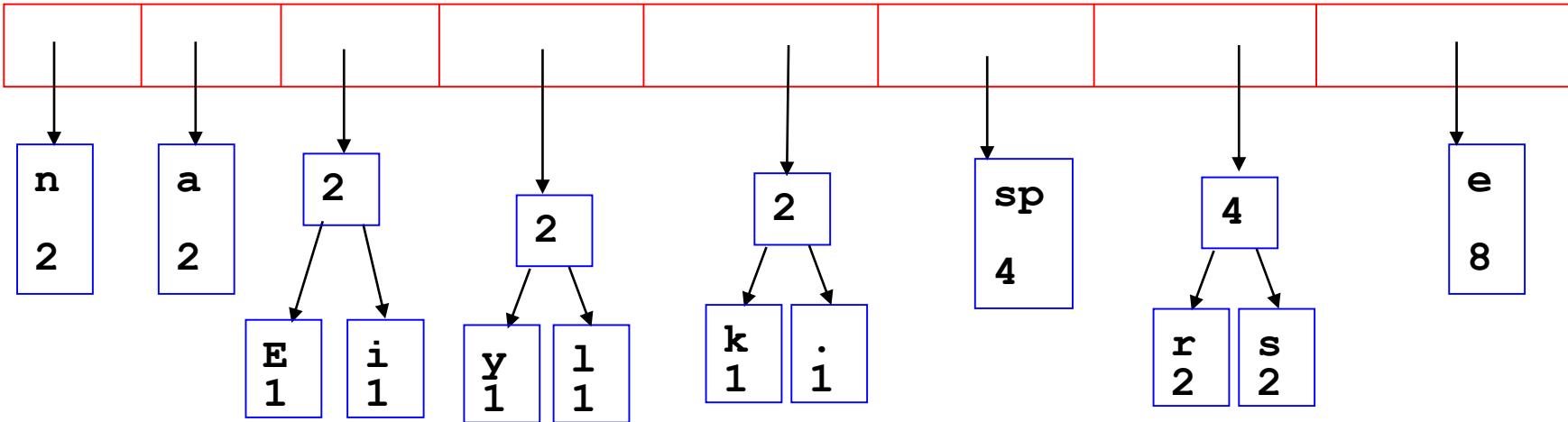
# Building a Tree

---



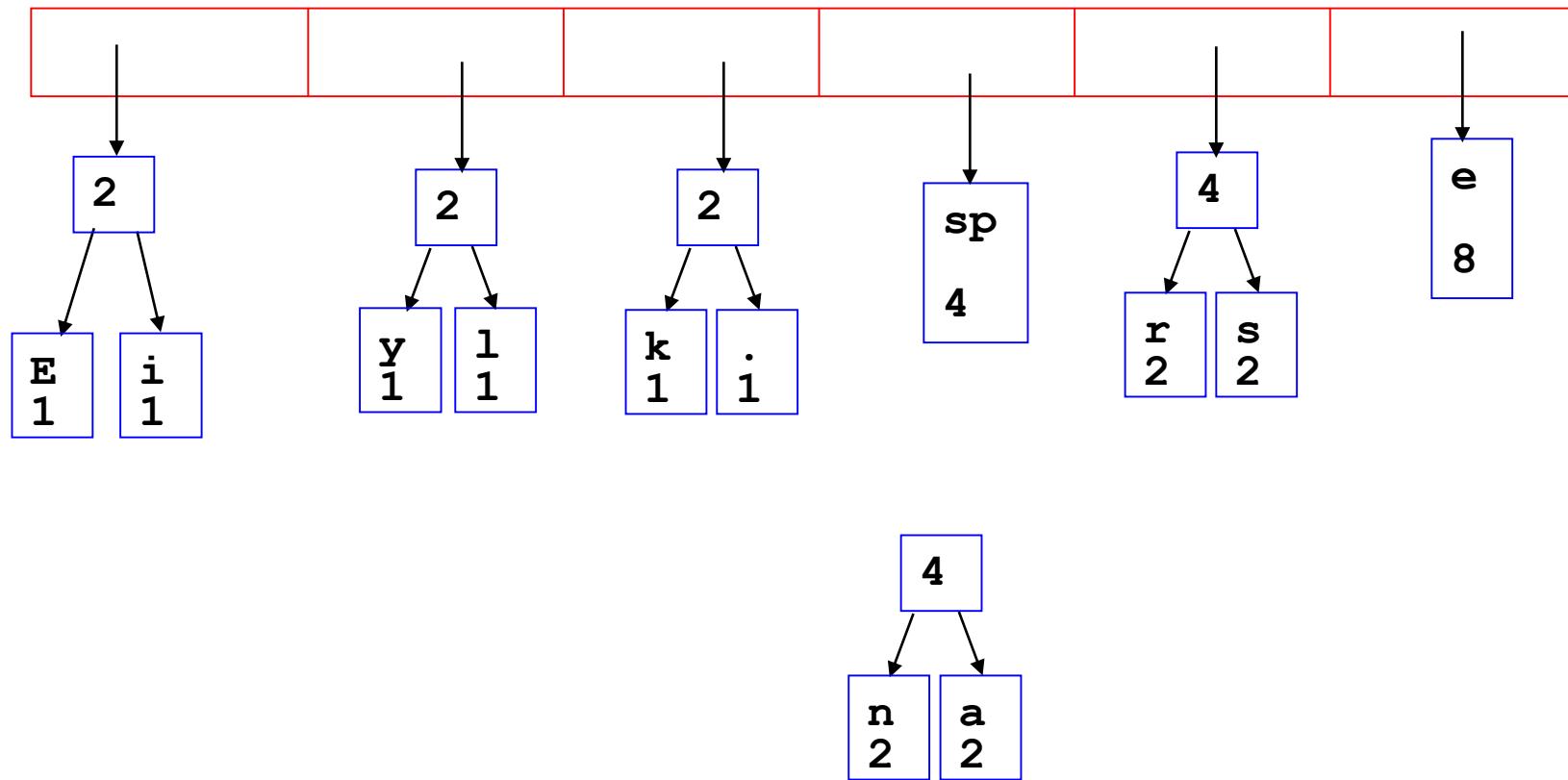
# Building a Tree

---



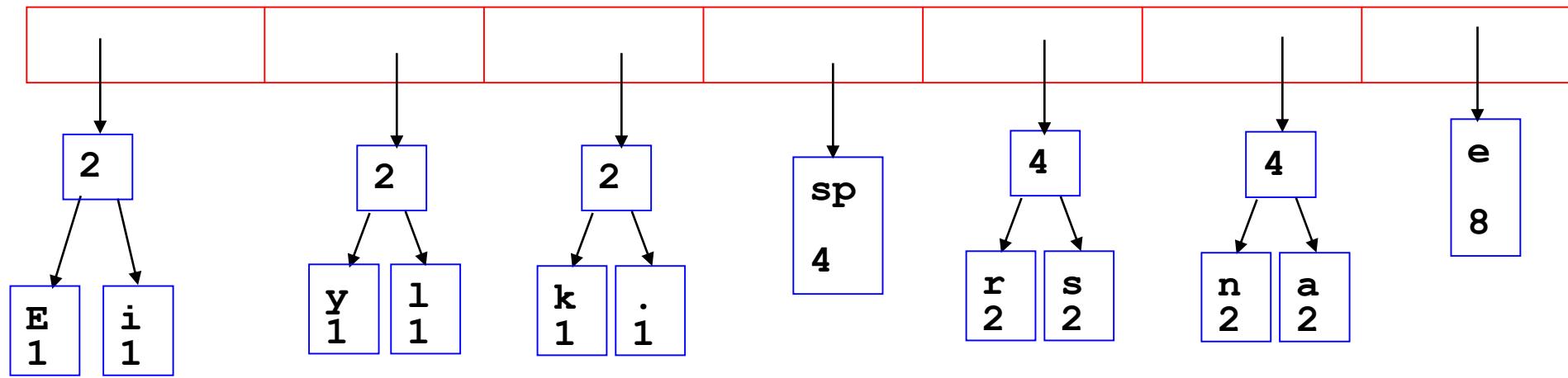
# Building a Tree

---



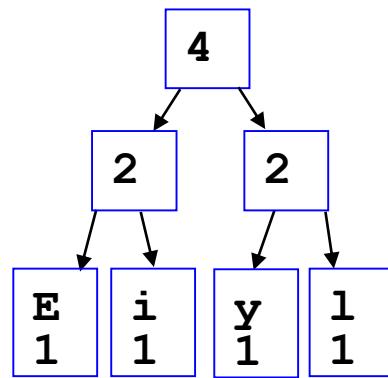
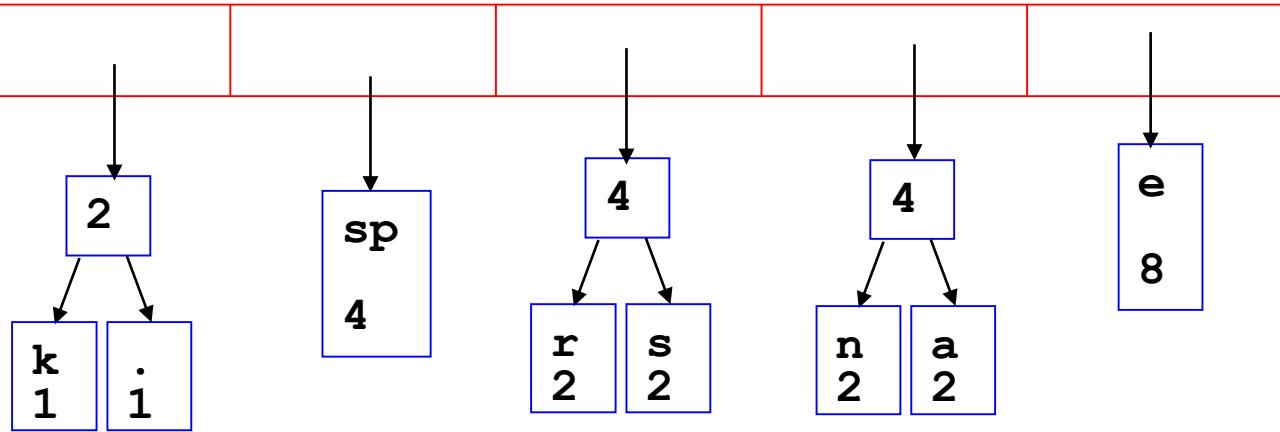
# Building a Tree

---



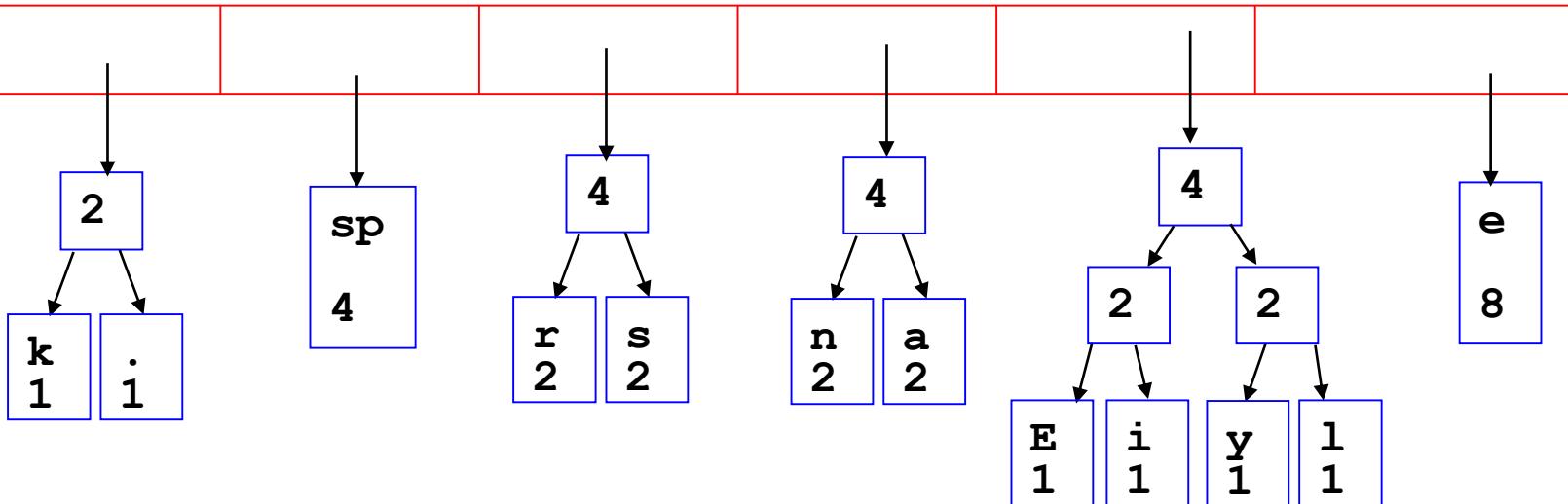
# Building a Tree

---



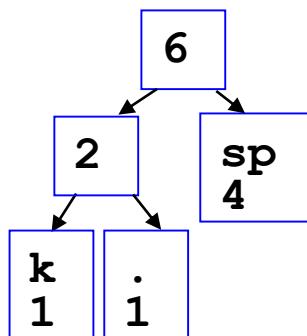
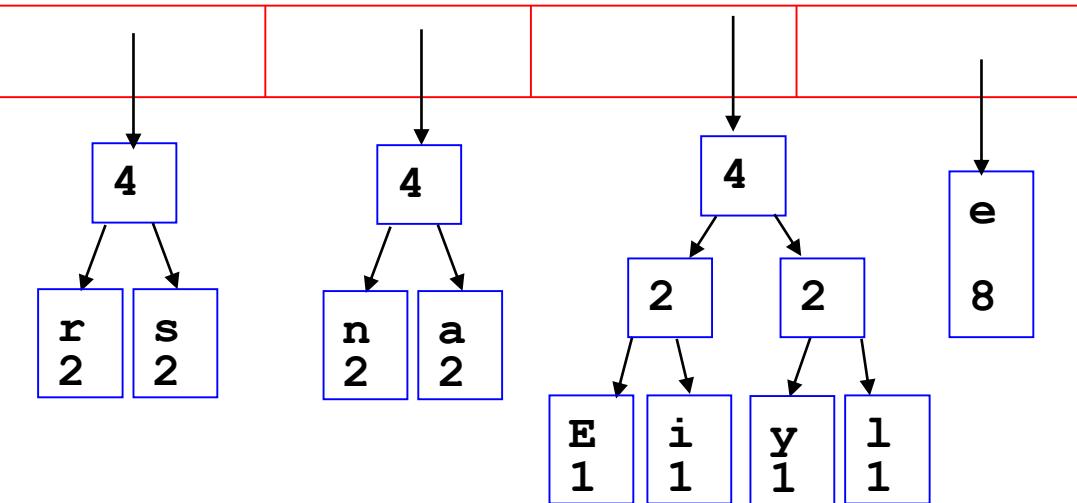
# Building a Tree

---



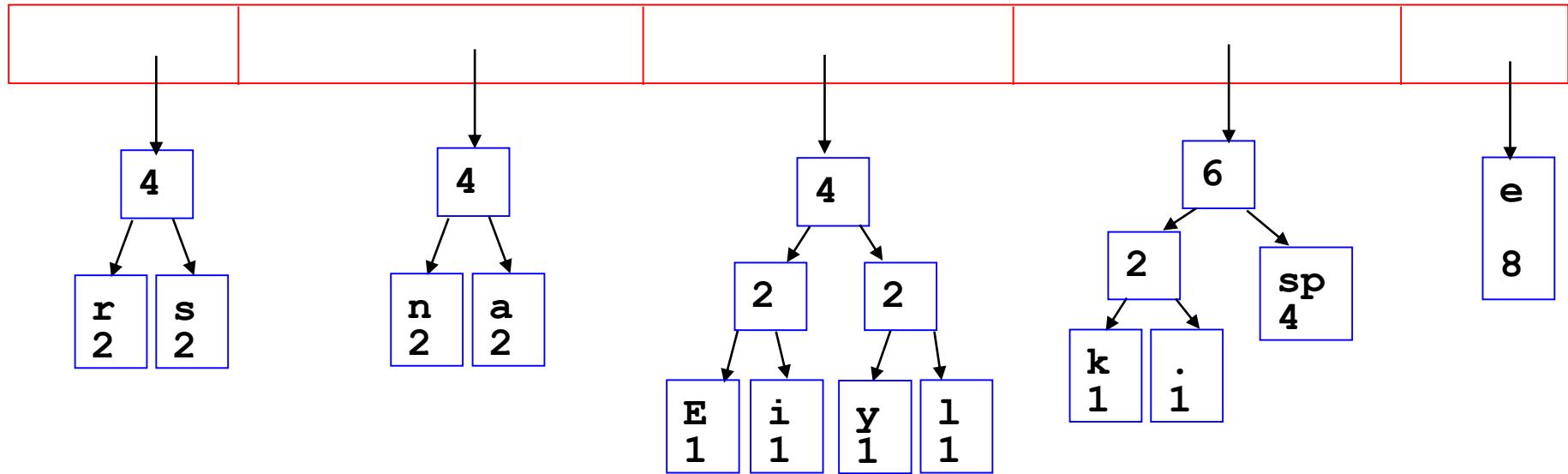
# Building a Tree

---



# Building a Tree

---

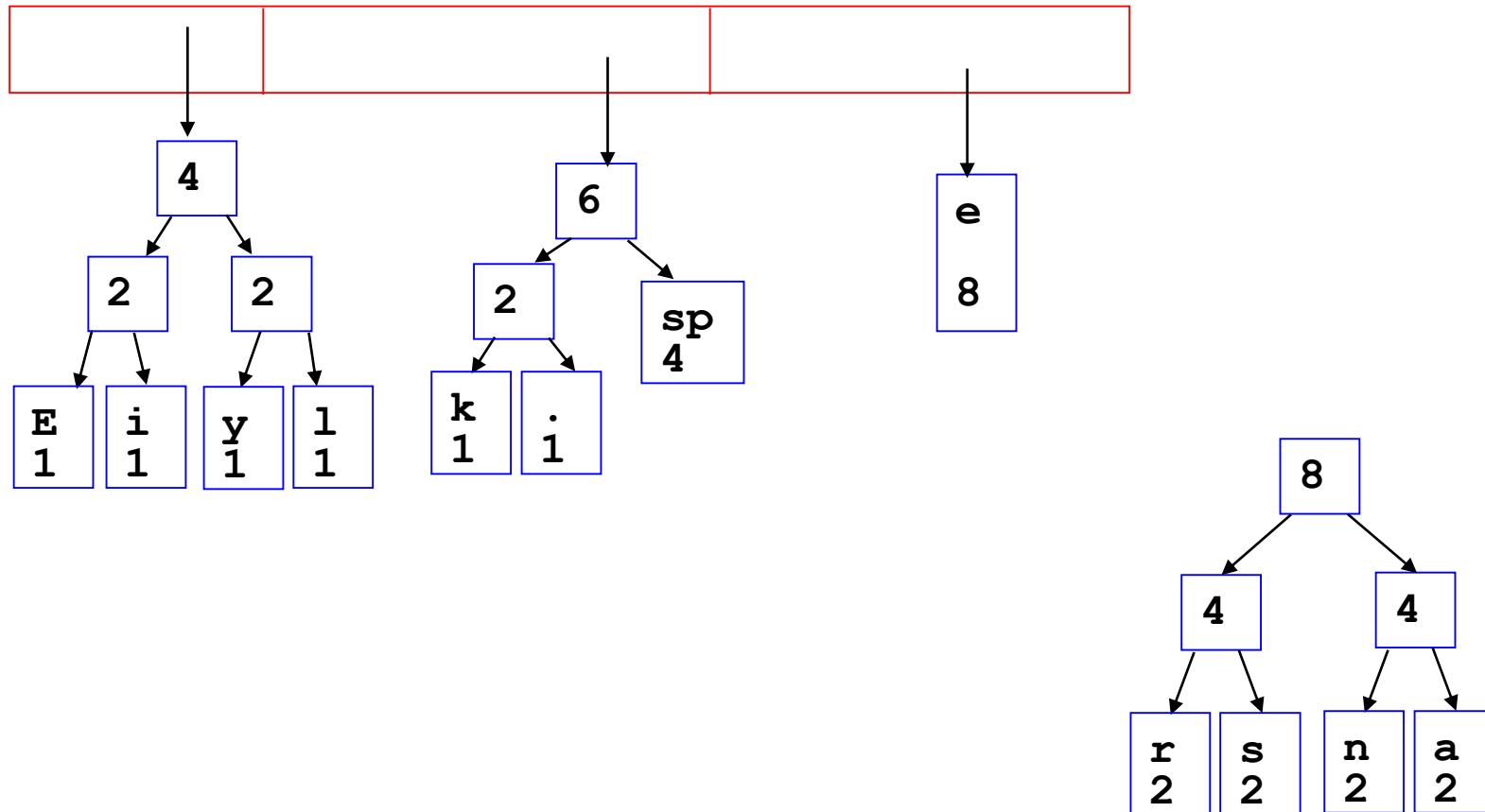


What is happening to the characters with a low number of occurrences?

---

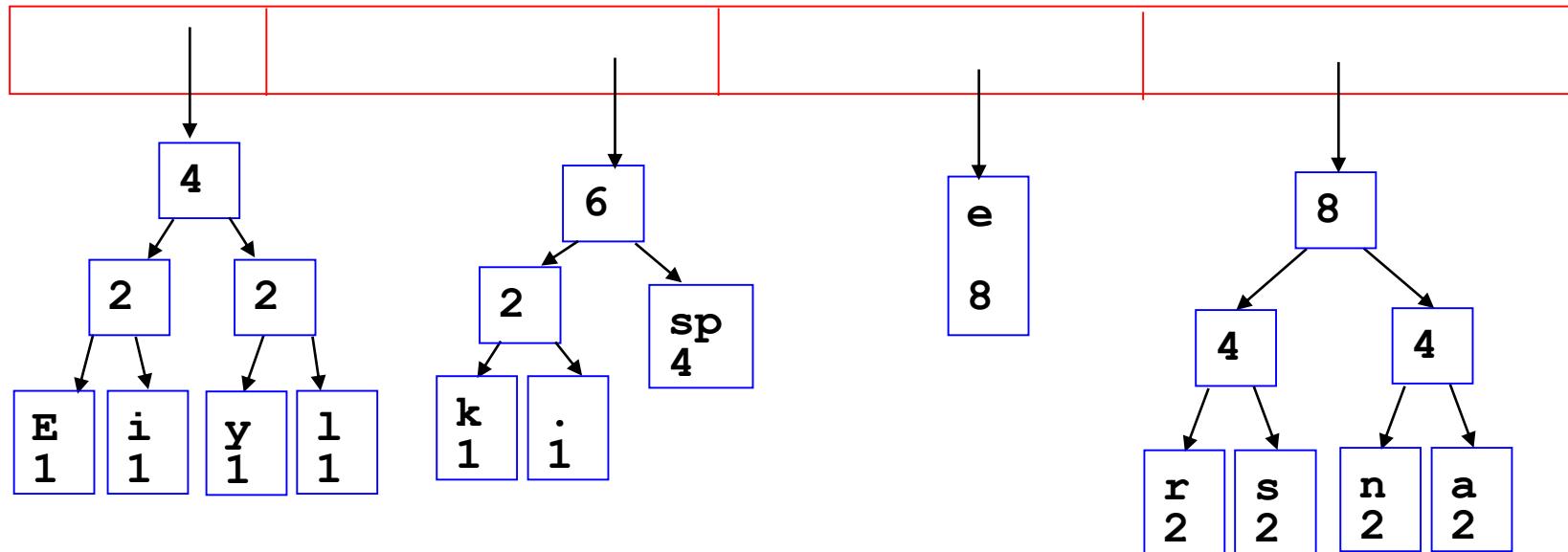
# Building a Tree

---



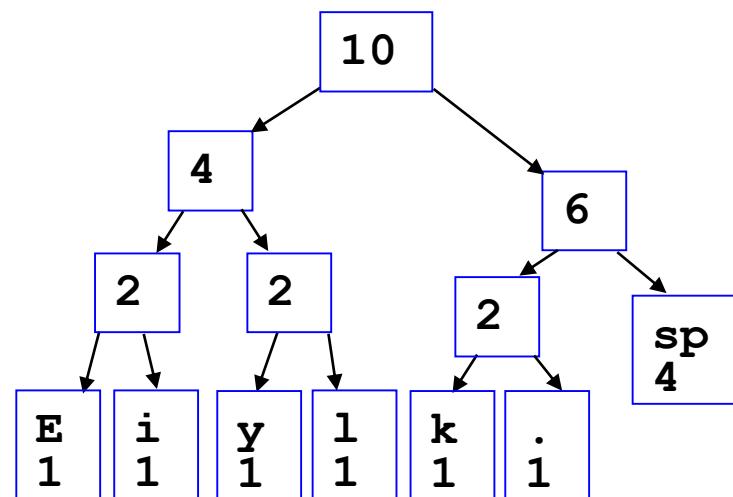
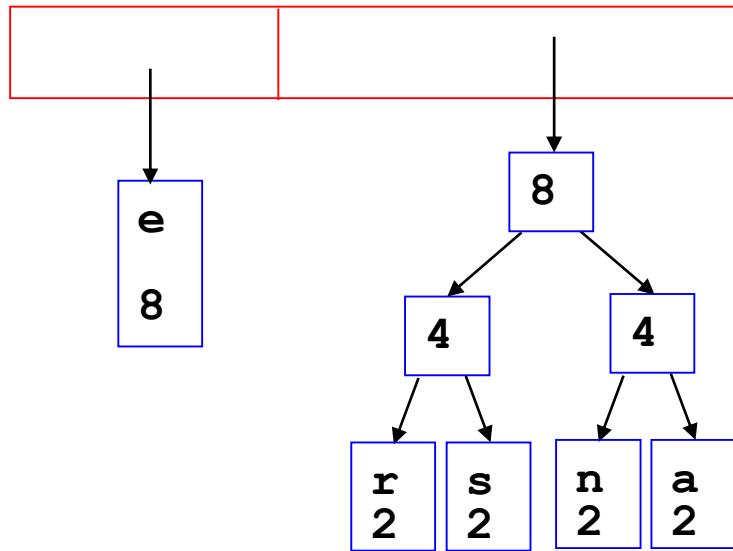
# Building a Tree

---



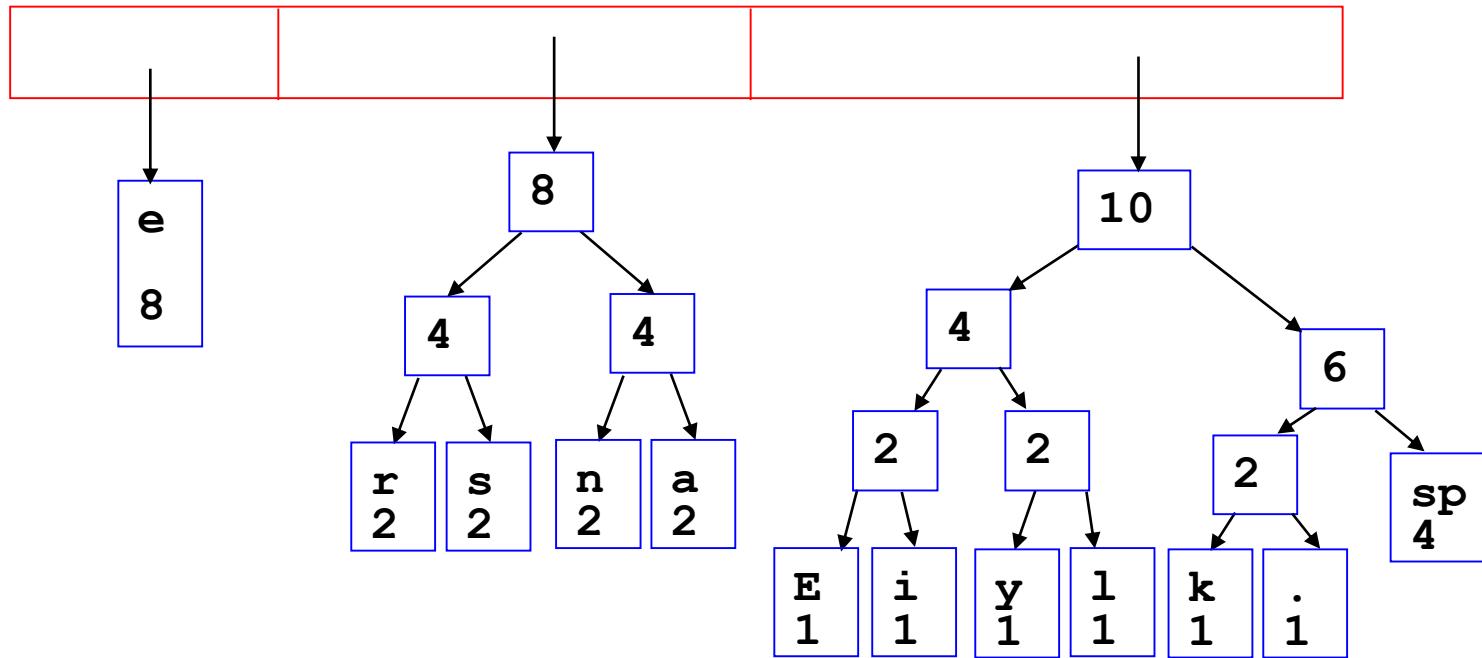
# Building a Tree

---

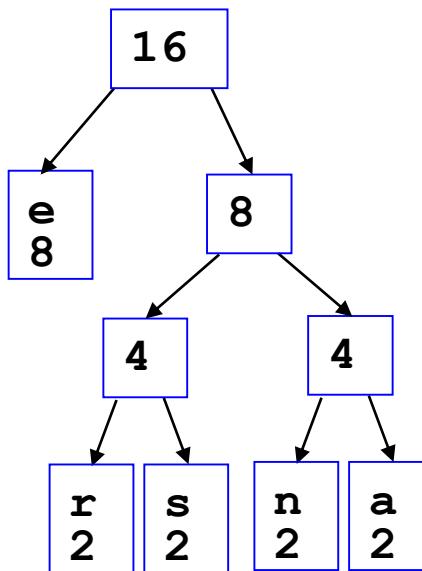
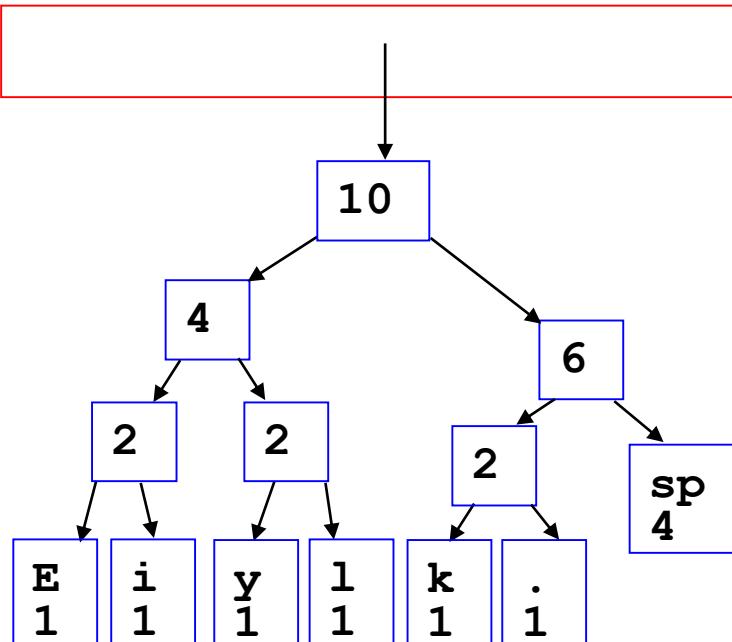


# Building a Tree

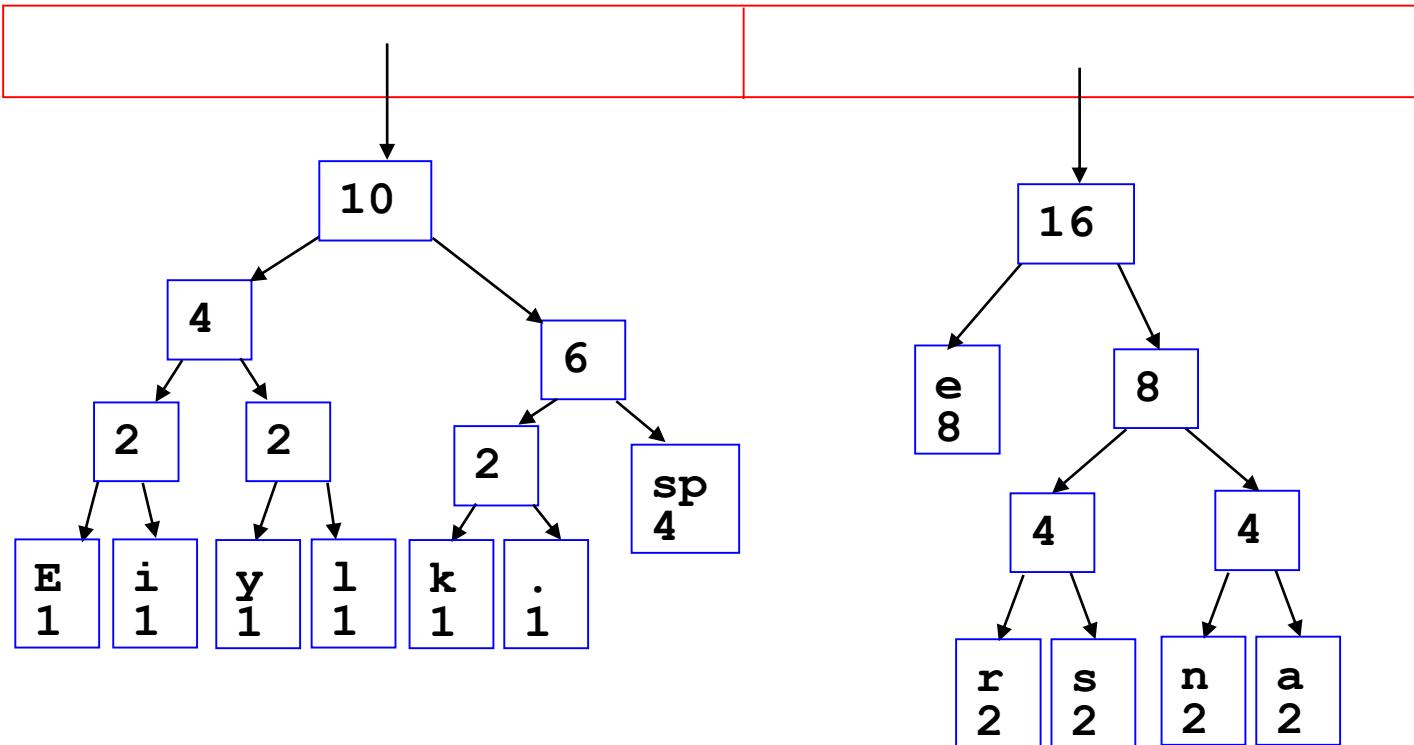
---



# Building a Tree

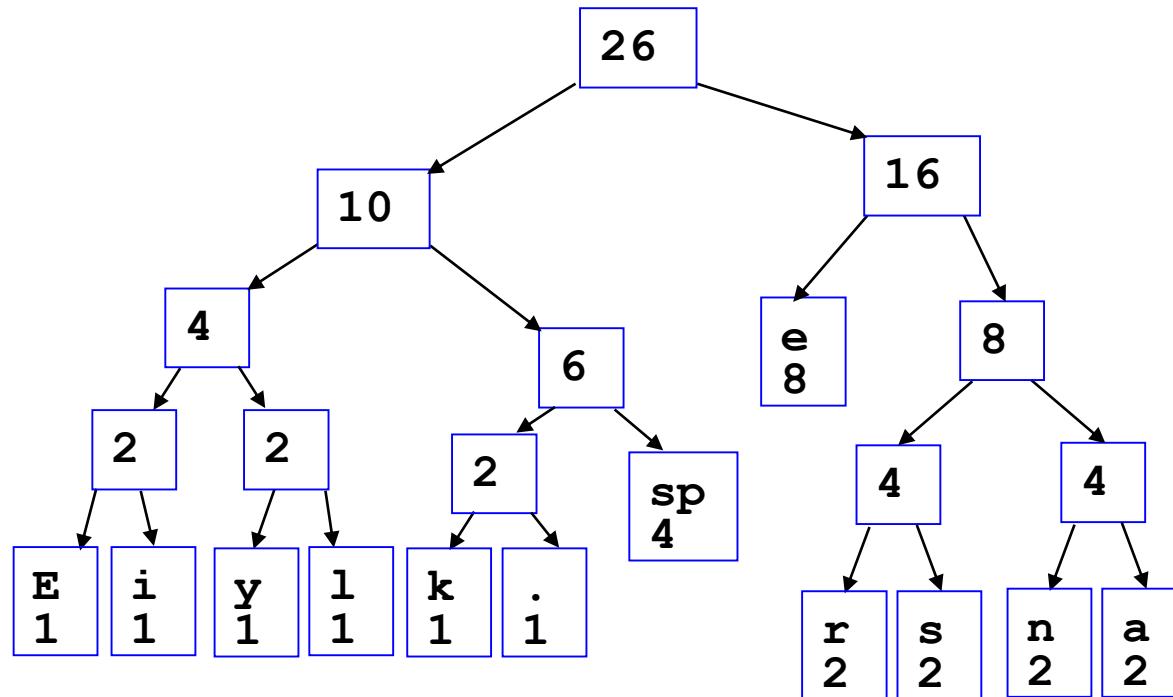


# Building a Tree



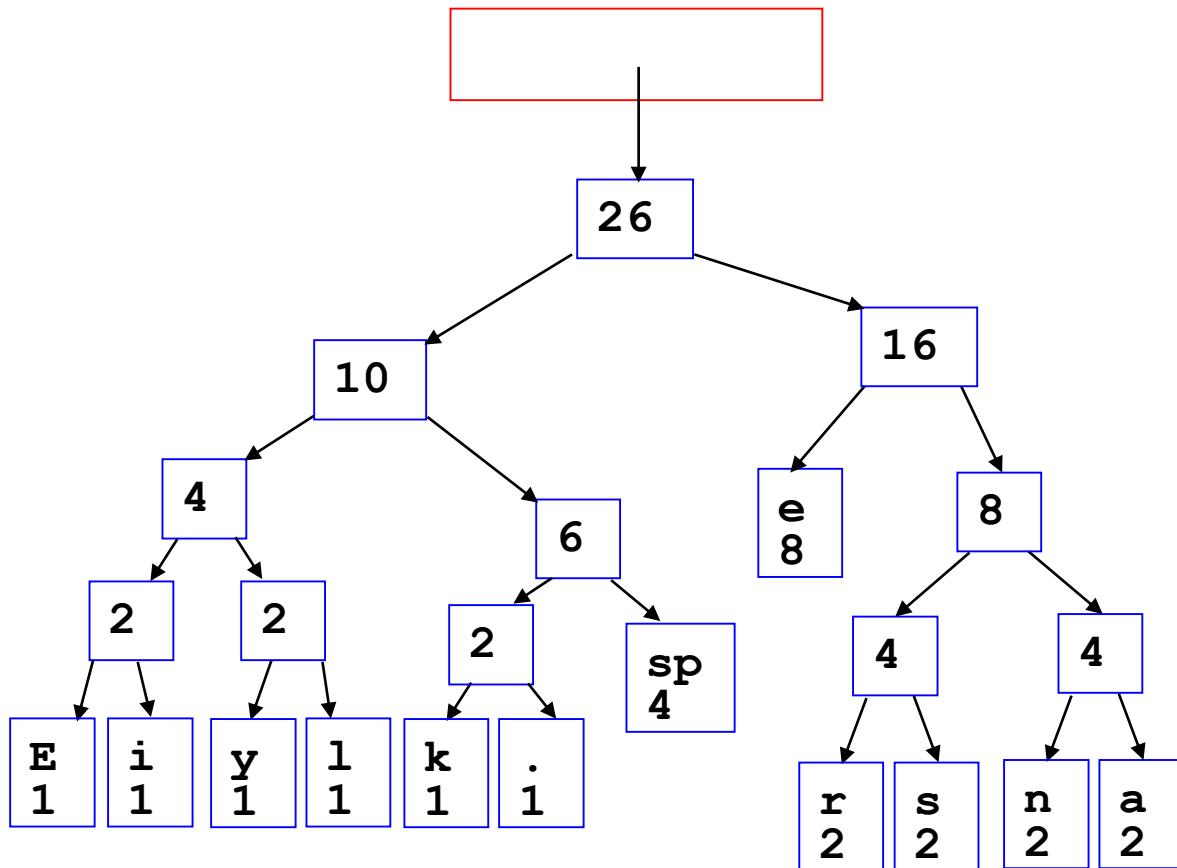
# Building a Tree

---



# Building a Tree

---



- After enqueueing this node there is only one node left in priority queue.

# Building a Tree

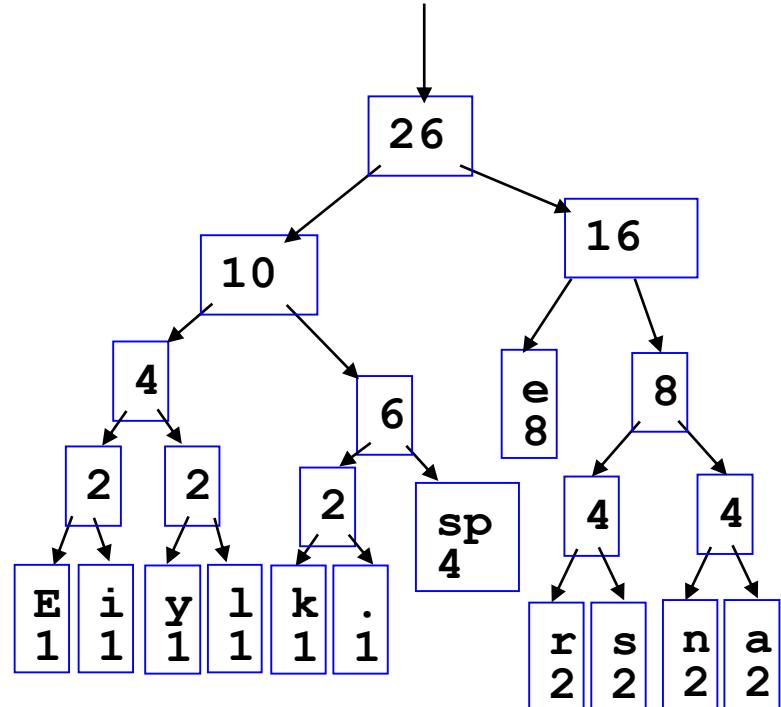
---

Dequeue the single node left in the queue.

This tree contains the new code words for each character.

Frequency of root node should equal number of characters in text.

Eerie eyes seen near lake.

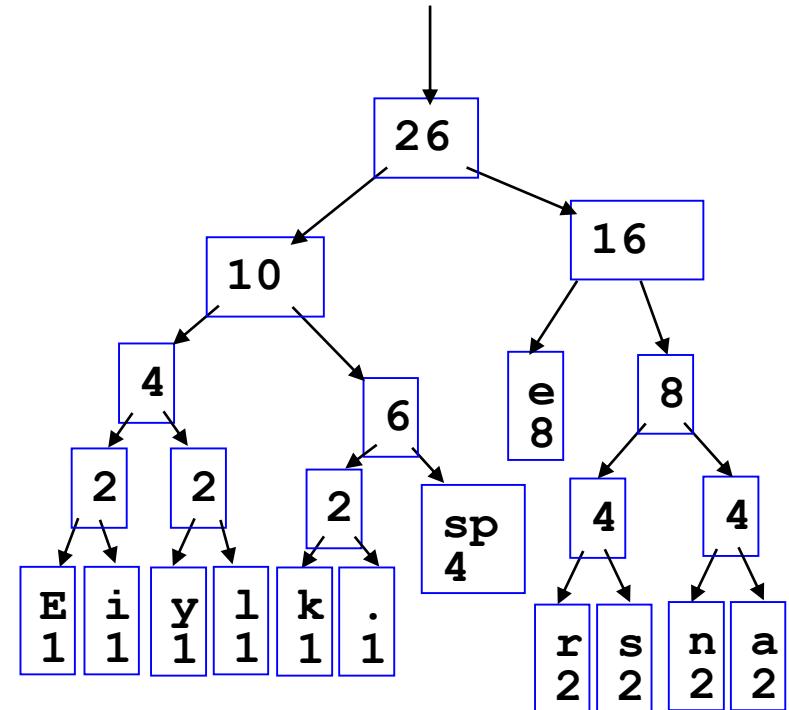


26 characters

# Encoding the File

## Traverse Tree for Codes

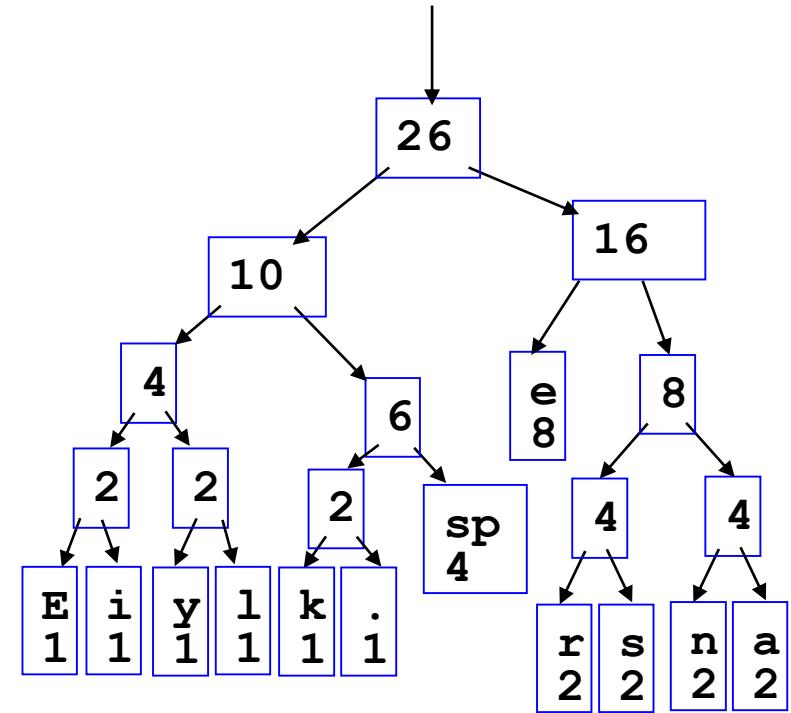
- Perform a traversal of the tree to obtain new code words
- Going left is a 0 going right is a 1
- code word is only completed when a leaf node is reached



# Encoding the File

## Traverse Tree for Codes

| Char  | Code |
|-------|------|
| E     | 0000 |
| i     | 0001 |
| y     | 0010 |
| l     | 0011 |
| k     | 0100 |
| .     | 0101 |
| space | 011  |
| e     | 10   |
| r     | 1100 |
| s     | 1101 |
| n     | 1110 |
| a     | 1111 |



# Encoding the File

---

- Rescan text and encode file using new code words

Eerie eyes seen near lake.

```
0000101100000110011  
1000101011011010011  
111010111110001100  
1111110100100101
```

- Why is there no need for a separator character?

| Char  | Code |
|-------|------|
| E     | 0000 |
| i     | 0001 |
| y     | 0010 |
| l     | 0011 |
| k     | 0100 |
| .     | 0101 |
| space | 011  |
| e     | 10   |
| r     | 1100 |
| s     | 1101 |
| n     | 1110 |
| a     | 1111 |

# Encoding the File

## Results

---

- Have we made things any better?
- 73 bits to encode the text
- ASCII would take  $8 * 26 = 208$  bits
- If modified code used 4 bits per character are needed. Total bits  $4 * 26 = 104$ . Savings not as great.

```
0000101100000110011  
1000101011011010011  
111010111110001100  
1111110100100101
```

# Decoding the File

---

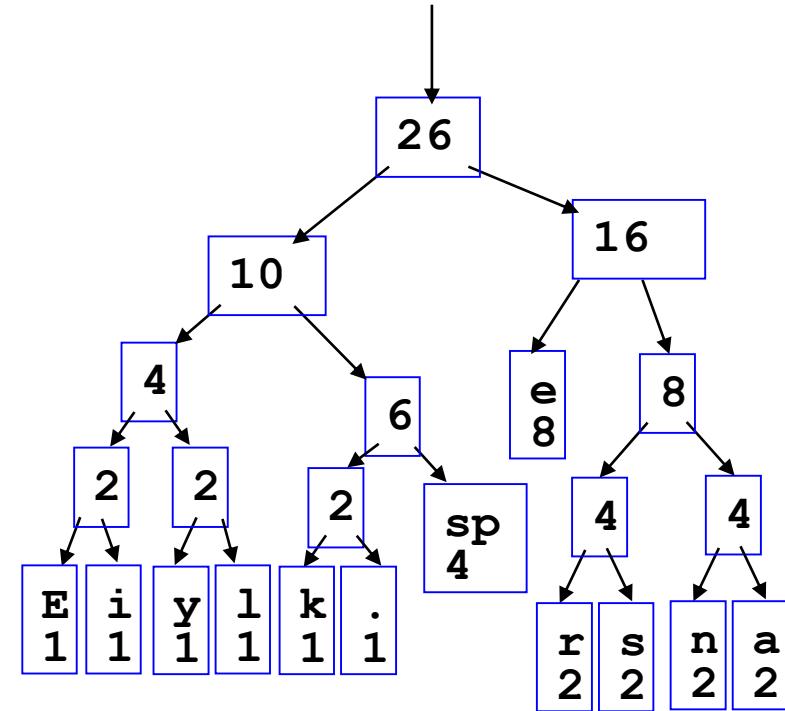
- How does receiver know what the codes are?
- Tree constructed for each text file.
  - Considers frequency for each file
  - Big hit on compression, especially for smaller files
- Tree predetermined
  - based on statistical analysis of text files or file types
- Data transmission is bit based versus byte based

# Decoding the File

---

- Once receiver has tree it scans incoming bit stream
- 0 → go left
- 1 → go right

```
101000110111101111  
01111110000110101
```



# Summary

---

- Huffman coding is a technique used to compress files for transmission
  - Uses statistical coding
    - more frequently used symbols have shorter code words
  - Works well for text and fax transmissions
  - An application that uses several data structures
-

# Exercise

---

Encode the following using Huffman encoding:

- she saw a ship in sea
- larry looks lazy
- he has a hat

# Data Structures

Topic: Heap



By  
**Ravi Kant Sahu**

*Asst. Professor,*

Lovely Professional University, Punjab



# Contents

- Introduction
- Insertion in Heap
- Deleting the Root of Heap
- Heap Sort
- Heap Sort Complexity
- Review Questions



# Introduction

- **Heap (MaxHeap):** A complete binary tree H with n elements is called a Heap if each node N of H has the following property:

“ The value at N is greater than or equal to the value at each of the children of N.”
- If the value at N is less than or equal to the value at each of the children of N, then it is called MinHeap.
- Heaps are maintained in memory by using linear array TREE.



# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Insertion in a Heap

- INSERT\_HEAP(TREE, N, ITEM)
  1. Set N = N+1, and PTR = N.
  2. Repeat Step 3 to 6 while PTR > 1
  3. Set PAR =  $\lfloor PTR / 2 \rfloor$
  4. If ITEM <= TREE [PAR], then:  
Set TREE[PTR] = ITEM and Return.
  5. Else: Set TREE[PTR] = TREE[PAR].
  6. Set PTR = PAR.
  7. Set TREE[1] = ITEM.
  8. Return.



# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Example

- Create a Heap from the following list of numbers:

40, 30, 50, 20, 60, 55, 70, 60, 65, 50



# Deletion of Root of a Heap

- Assign Root R to some variable ITEM.
- Replace the deleted node R by the last node L of Heap H so that H is still complete tree, **but not a Heap**.
- Call MaxHeapify to maintain the heap property.



# Build MaxHeap

**BUILD\_MAX-HEAP(A)**

1.  $\text{heapsize}[A] = \text{length}[A]$
2. Repeat for  $i = \lfloor \text{length}[A]/2 \rfloor$  to 1
3.     Call **MAX\_HEAPIFY(A, i)**
4. Exit



# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Maintaining Heap Property

## MAX\_HEAPIFY(A, i)

1. Set:  $l = \text{LEFT}(i)$
2. Set:  $r = \text{RIGHT}(i)$
3. If  $l \leq \text{heapsize}[A]$  and  $A[l] > A[i]$ , then:
  4.  $\text{largest} = l$ .
5. Else:  $\text{largest} = i$ .
6. If  $r \leq \text{heapsize}[A]$  and  $A[r] > A[\text{largest}]$ , then:
  7.  $\text{largest} = r$ .
8. If  $\text{largest} \neq i$ , then:
  9. Exchange  $A[i] \leftrightarrow A[\text{largest}]$ .
  10.  $\text{MAX\_HEAPIFY}(A, \text{largest})$ .



# Heap Sort

## HEAP\_SORT (A)

1. BUILD\_MAXHEAP (A)
2. Repeat for  $i = \text{length}[A]$  to 2
  3. Exchange  $A[1] \leftrightarrow A[i]$ .
  4. Heapsize[A] = Heapsize [A] – 1.
  5. Call MAX\_HEAPIFY(A, 1).
6. Exit.



# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Complexity of Heap Sort

Average and Worst case Complexity of Heap sort =  $O(n \log n)$ .



# Questions



# Review Questions

- What do you mean by Heap?
- What is the complexity of Heap sort?
- Calculate the complexity of Heap sort.
- How will you insert an element in a heap?
- What is the difference between Maxheap and Minheap?

# Data Structures

Topic: Hashing

By

Ravi Kant Sahu  
Asst. Professor



Lovely Professional University, Punjab



# Outlines

- Introduction
- Hashing
- Hash Functions
  - Division Method
  - Midsquare Method
  - Folding Method
- Collision Resolution
- Open Addressing : Linear Probing & Modifications
  - Quadratic Probing
  - Double Hashing



# Introduction

- The search time of all the algorithms depends on the number  $n$  of the elements in the collection of Data.
- A searching technique which is essentially independent of  $n$  is called Hash Addressing or Hashing.
- $F$  is a file with  $n$  records and a set  $K$  of Keys which uniquely determine the records in  $F$ .
- $F$  is maintained in memory by a table  $T$  of  $m$  memory locations and  $L$  is a set of memory addresses of the locations in  $T$ .



# Example

- Suppose a company with 168 employees assigns a 5 digit Emp\_No. to each employee which is used as primary key in Employee file.
  
- Emp\_No can be used as address of record in memory but we will require 100000 memory locations.



# Example



# Hashing

- Hashing is a searching technique which is independent of the number of elements in file.
- The general idea of using the Key to determine the address of records is an excellent idea. But it must be modified to prevent the wastage of space.
- The Modification takes the form of a function H from the set K of keys into the set L of memory addresses.

$$H: K \rightarrow L$$



# Hash Functions

- Hash function H is a mapping between set of Keys K and set of memory locations L.

$$H: K \rightarrow L$$

- Such a function H may not yield distinct values.
- It is possible that two different keys K<sub>1</sub> and K<sub>2</sub> will yield the same hash address.
- This situation is called Collision.



# Hash Functions...

- Two principle criteria used in selecting a hash function H are:
  1. H should be very easy and quick to compute.
  2. H should be uniformly distribute the hash address throughout the set L. So that the number of collisions are minimized.
- Some popular hash Functions are:
  - Division Method
  - Midsquare Method
  - Folding Method



# Division Method

- Choose a number  $m$  larger than the number  $n$  of Keys(usually a prime number) . Hash function is defined as:

$$H(K) = k \pmod{m}$$

or       $H(K) = k \pmod{m} + 1$

(when we want hash address to range from 1 to m rather than 0 to m-1)



# Example



# Midsquare Method

- The key is squared and some digits are deleted from both sides to obtain 1 digits.

$$H(k) = 1$$

where 1 is obtained by deleting digits from both ends of  $k^2$ .



# Example



# Folding Method

- The key  $k$  is partitioned into a number of parts  $k_1, k_2, k_3 \dots k_r$ , where each part (except possibly the last) has the same number of digits as the required address.
- Then the parts are added together, ignoring the last carry.

$$H(k) = k_1 + k_2 + \dots + k_r$$



# Example



# Hash Table

- A hash table (also hash map) is a data structure used to implement an associative array, a structure that can map keys to values.
- A hash table uses a hash function to compute an index into an array of buckets or slots (or in memory), from which the correct value can be found.
- Ideally, the hash function should assign each possible key to a unique bucket, but this ideal situation is rarely achievable in practice.



# Collision Resolution

Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)



# Collision Resolution

- Collision is a situation when two or more keys map to the same memory location.
- Load Factor: The ratio of number n of keys in K to the number m of hash addresses in L.

$$\lambda = n/m$$

- Efficiency of a hash function is measured by the average number of probes needed to find the location of record with a given key k.



# Open Addressing

## ➤ Linear Probing

- Resolves collisions by placing the data into the next open slot in the table.
- We assume that the table  $T$  with  $m$  locations is circular, so that  $T[1]$  comes after  $T[m]$ .



# Example

The keys 94, 25, 105, 169, 38, 80 and 151 are inserted into an initially empty hash table of size 13 using Division Method. Linear Probing is used for Collision Resolution.

Find the appropriate hash function and draw the resultant Hash Table after placing the keys and calculate the Average number of probes for Unsuccessful search.

# Linear Probing

- Insert pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45
- divisor = b (number of buckets) = 17.
- $H = \text{key \% 17}$ .

| 0  | 4 | 8  | 12 | 16 |
|----|---|----|----|----|
| 34 | 0 | 45 |    |    |

  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

|   |    |   |    |    |    |    |    |    |
|---|----|---|----|----|----|----|----|----|
| 6 | 23 | 7 | 28 | 12 | 29 | 11 | 30 | 33 |
|---|----|---|----|----|----|----|----|----|



# Performance of Linear Probing

- Average number of probes for a successful search

$$S(\lambda) = 0.5 (1 + \frac{1}{(1 - \lambda)})$$

- Average number of probes for an unsuccessful search

$$U(\lambda) = 0.5 (1 + \frac{1}{(1 - \lambda)^2})$$



# Problems with Linear Probing

- Identifiers(keys) tend to cluster together
- Adjacent cluster tend to coalesce
- Increase the search time
- Worst Case Complexity is  $\Theta(n)$ . This happens when all keys are in the same cluster.
- Two techniques are used to minimize clustering:
  1. Quadratic Probing
  2. Double Hashing



# Quadratic Probing

- Quadratic probing uses a quadratic function of  $i$  as the increment
- Examine buckets  $H(x), (H(x)+i^2)\% b$



# Double Hashing

- A second hash function  $H`$  is used for resolving the collision.
- Let  $H(k) = h$  and  $H`(k) = h` \neq m$   
Then we linearly search the locations with addresses:  
$$h, h+h`, h+2h`, h+3h`, \dots$$



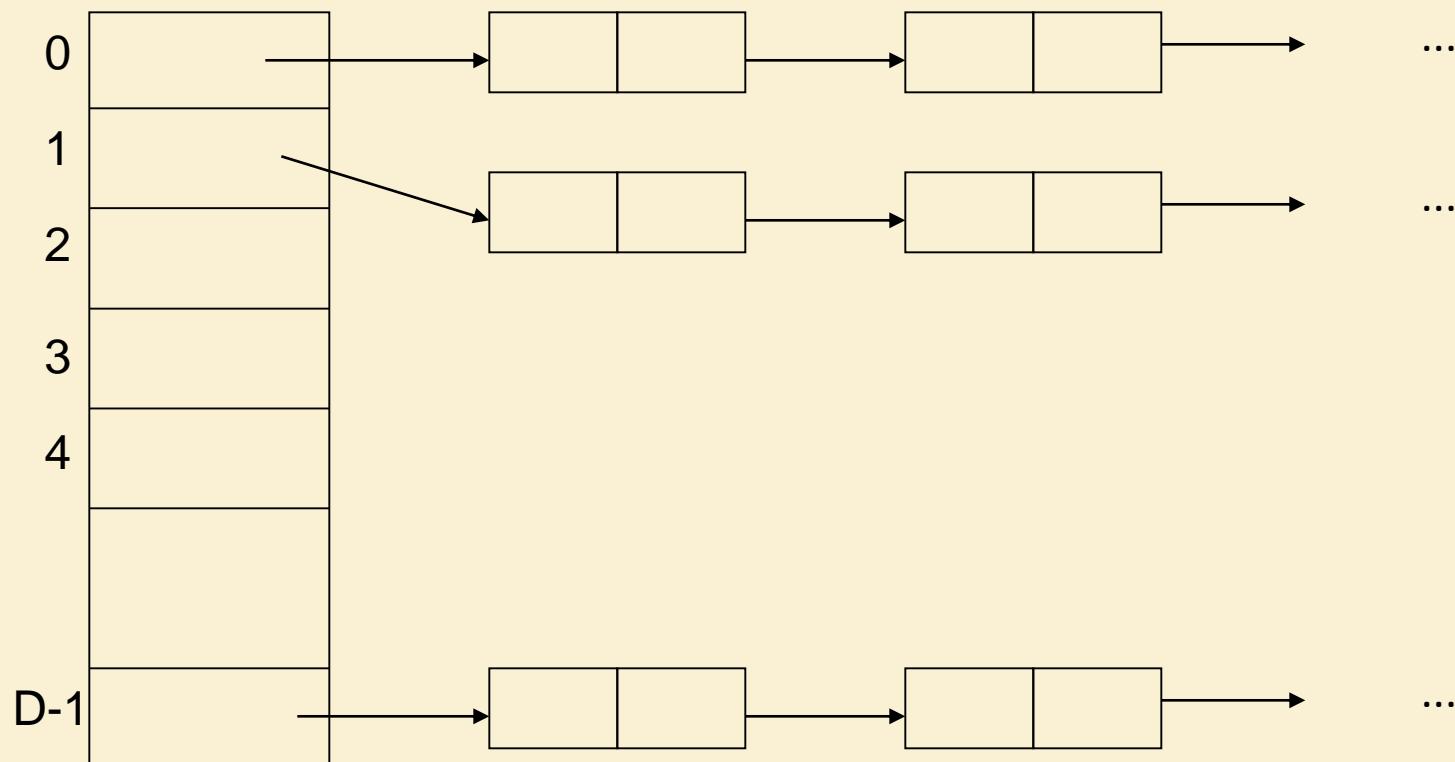
Given a hash table with addresses 13 to 23. The keys 10, 83, 44, 72, 42, 56 and 9 are inserted into an initially empty hash table using **Division Method**.

**Use Double Hashing**  $[H'(K) = (K \bmod 7) + 1]$  for Collision Resolution. Find the average Number of Probs for Successful Search.

# Open Hashing or Separate Chaining

- Each bucket in the hash table is the head of a linked list.
- All elements that hash to a particular bucket are placed on that bucket's linked list.
- Records within a bucket can be ordered in several ways
  - by order of insertion,
  - by key value order, or
  - by frequency of access order

# Open Hashing Data Organization



# Analysis

- Open hashing is most appropriate when the hash table is kept in main memory, implemented with a standard linked list.
- We hope that number of elements per bucket roughly equal in size, so that the lists will be short.
- If there are  $n$  elements in set, then each bucket will have roughly  $n/D$  elements.
- If we can estimate  $n$  and choose  $D$  to be roughly as large, then the average bucket will have only one or two members

# Brainstorming-1

In a hash table of size 13 which index positions would the following two keys map to?

27, 130

- (A) 1, 10
- (B) 13, 0
- (C) 1, 0
- (D) 2, 3

# Brainstorming-2

Suppose you are given the following set of keys to insert into a hash table that holds exactly 11 values:

113 , 117 , 97 , 100 , 114 , 108 , 116 , 105 , 99

Which of the following best demonstrates the contents of the has table after all the keys have been inserted using linear probing?

- (A) 100, \_\_, \_\_, 113, 114, 105, 116, 117, 97, 108, 99
- (B) 99, 100, \_\_, 113, 114, \_\_, 116, 117, 105, 97, 108
- (C) 100, 113, 117, 97, 14, 108, 116, 105, 99, \_\_, \_\_
- (D) 117, 114, 108, 116, 105, 99, \_\_, \_\_, 97, 100, 113

# Brainstorming-3

Suppose you are given the following set of keys to insert into a hash table that is capable of holding exactly 12 values:

93 , 47 , 97 , 106 , 15 , 121 , 108 , 31 , 9

Find out the average number of probes for Successful Search and Unsuccessful search if Linear Probing is used for Collision Resolution.



Ravi Kant Sahu, Asst. Professor @ Lovely Professional University, Punjab (India)

# Data Structures

Topic: Graphs



By  
**Ravi Kant Sahu**

*Asst. Professor,*

Lovely Professional University, Punjab



# Contents

- Introduction
- Basic Terminology
- Sequential Representation of Graphs
  - Adjacency Matrix
  - Path Matrix
- Linked Representation of Graphs
- Warshall's Algorithm: Shortest Path
- Review Questions



# Introduction

- A Graph  $G$  is a collection of:
  1. A set  $V$  of elements called Nodes or Vertices.
  2. A set  $E$  of Edges such that each edge  $e$  in  $E$  is identified with a unique pair  $[u, v]$  of nodes in  $V$ , denoted by  $e = [u, v]$ .

$$G = (V, E)$$

- Nodes  $u$  and  $v$  are called end points of edge  $e$  and also known as adjacent nodes or neighbors.



# Basic Terminology

- Degree of a node: Degree of a node,  $\deg(u)$ , is the number of edges containing  $u$ .
- If  $\deg(u) = 0$ , then node  $u$  is called Isolated Node.
- A Path  $P$  of length  $n$  from node  $u$  to a node  $v$  is defined as a sequence of  $n+1$  nodes.

$$P = (v_0, v_1, v_2, \dots, v_n)$$



# Path

- **Simple Path:** The path is said to be simple if all the nodes are distinct.
- **Closed Path:** A path is said to be closed if first and last node are same i.e.  $v_0 = v_n$ .
- **Cycle:** A cycle is a closed simple path with length 3 or more.
- A cycle with length  $k$  is called  $k$ -cycle.
- **Connected Graph:** If there is always a path between any two of its nodes.



# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Graph

- **Connected Graph:** A graph G is connected iff there is a simple path between any two nodes in G.
- **Complete Graph:** A graph G is said to be complete if every node  $u$  in G is adjacent to every other node v in G. i.e. each node u is directly connected to all other nodes v in Graph G.
- A complete graph with  $n$  nodes will have  $n(n-1)/2$  edges.



# Connected Graph...

- **Connected Graph:** A graph  $G$  is connected iff there is a simple path between any two nodes in  $G$ .
- A connected graph  $T$  without any cycle is called a Tree graph or Free tree.
- There is a unique simple path  $P$  between any two nodes  $u$  and  $v$  in  $T$ .
- If  $T$  is a finite tree with  $m$  nodes, then  $T$  will have  $m-1$  edges.



# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Labeled Graphs...

- A graph  $G$  is said to be labeled if its edges are assigned data.
- **Weighted Graph:** Graph  $G$  is said to be weighted if each edge  $e$  is assigned a non-negative numerical value  $w(e)$  called the weight or length of edge.
- If no other information about weights are given in a graph, then assume the weight  $w(e) = 1$  for each edge.



# Multi-Graph

- Multiple Edges: Distinct edges  $e$  and  $e'$  are called multiple edges if they connect the same endpoints,  
*i.e. if  $e = [u, v]$  and  $e' = [u, v]$ .*
- Loops: An edge  $e$  is called a loop if it has identical endpoints ,  
*i.e. if  $e = [u, u]$  .*
- Note: Definition of a graph does not allow any loop or multiple edge in Graph.
- A Graph with loops or multiple edges is called a Multi-graph.



# Directed-Graph

- A Directed-graph G, also called Digraph or Graph, is same as the multigraph except that each edge e in G is assigned a direction i.e. each edge e is identified by an ordered pair (u, v ) of nodes in G, rather than an unorderd pair [u, v].

## Terminology:

- Edge  $e = (u, v)$  is called an Arc.
  - $e$  begins at  $u$  and ends at  $v$ .
  - $u$  is called origin and  $v$  is called destination.
  - $u$  is adjacent to  $v$  and  $v$  is adjacent to  $u$ .
  - $u$  is predecessor of  $v$  and  $v$  is sucessor of  $u$ .



# Directed-Graph...

- A Directed-graph G is said to be connected, or *strongly connected*, if for each pair  $u, v$  of nodes in G there is a path from u to v and there is also a path from v to u.
- G is said to be *unilaterally connected* if for any pair  $u, v$  of nodes in G there is a path from u to v or a path from v to u.
- A directed graph is said to be simple if G has no parallel edges.
- A simple graph may have loops but it can't have more than one loop at a given node.



# Degree of Graph

- **Outdegree:** Outdegree of a node  $u$  in  $G$  is the number of edges beginning at  $u$ .
- **Indegree:** Indegree of a node  $u$  in  $G$  is the number of edges ending at  $u$ .
- **Source:** A node  $u$  is called a source if it has a positive outdegree but zero indegree.
- **Sink:** A node  $u$  is called a sink if it has a positive indegree but zero outdegree.



# Sequential Representation

- There are two ways of representing a graph in memory:
  - Sequential Representation
    - Adjacency Matrix
    - Path Matrix
  - Linked Representation



# Adjacency Matrix

- Let  $G$  is a simple directed graph with  $m$  nodes and the nodes of  $G$  have been ordered and are called  $v_1, v_2, \dots, v_m$ .
- The adjacency matrix  $A = a_{ij}$  of graph  $G$  is the  $m \times m$  matrix defined as below:
  - $a_{ij} = 1$ , if  $v_i$  is adjacent to  $v_j$ , i.e. if there is an edge  $(v_i, v_j)$
  - $a_{ij} = 0$ , otherwise
- Suppose  $G$  is an undirected graph, then the adjacency matrix  $A$  of  $G$  will be a symmetric matrix i.e. one in which  $a_{ij} = a_{ji}$ .



# Adjacency Matrix

- Let  $A$  be the adjacency matrix of a graph  $G$ . Then  $a_K(i, j)$ , the  $ij$  entry in the matrix  $A^K$ , gives the number of paths of length  $K$  from  $v_i$  to  $v_j$ .



# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Path Matrix

- Let  $G$  is a simple directed graph with  $m$  nodes  $v_1, v_2, \dots, v_m$ .
- The Path matrix or Reachability matrix  $P = p_{ij}$  of graph  $G$  is the  $m \times m$  matrix defined as below:

$p_{ij} = 1$ , if there is a path from  $v_i$  to  $v_j$

$p_{ij} = 0$ , otherwise



# Path Matrix...

- Let A be the adjacency matrix and P be the path matrix of a diagraph G. Then  $P_{ij} = 1$ , iff there is a non-zero number in the  $ij$  entry of the matrix

$$B_m = A + A^2 + A^3 + \dots + A^m$$

- Path matrix is obtained by replacing the non-zero entries in  $B_m$  by 1.
- Graph G is strongly connected iff path matrix P of G has no zero entries.



# Warshall's Algorithm: Path Matrix

- A directed graph  $G$  with  $M$  nodes is maintained in memory by its adjacency matrix  $A$ . This algorithm finds the path matrix  $P$  of the graph  $G$ .

1. Repeat for  $i, j = 1, 2, \dots, M$ :  
    If  $A[i, j] = 0$ , then: Set  $P[i, j] = 0$ .  
    Else: Set  $P[i, j] = 1$ .
2. Repeat Step 3 and 4 for  $k = 1, 2, \dots, M$ :
3.     Repeat Step 4 for  $i = 1, 2, \dots, M$ :
4.         Repeat for  $j = 1, 2, \dots, M$ :  
           Set  $P[i, j] = P[i, j] \vee (P[i, k] \wedge P[k, j])$ .  
           [End of Step 4 Loop.]  
        [End of Step 3 Loop.]  
    [End of Step 2 Loop.]
5. Exit.



# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Floyd-Warshall Algorithm: Shortest Path

*Floyd–Warshall algorithm* is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights.

1. Repeat for  $i, j = 1, 2, \dots, M$ :  
    IF:  $W[i, j] = 0$ , then Set  $Q[i, j] = \infty$ .  
    Else: Set  $Q[i, j] = W[i, j]$ .
2. Repeat Step 3 and 4 for  $k = 1, 2, \dots, M$ :
3.     Repeat Step 4 for  $i = 1, 2, \dots, M$ :
  4.         Repeat for  $j = 1, 2, \dots, M$ :  
           Set  $Q[i, j] = \text{Min} (Q[i, j], (Q[i, k] + Q[k, j]))$   
           [End of Step 4 Loop.]  
           [End of Step 3 Loop.]  
           [End of Step 2 Loop.]
5. Exit.



# Work Space

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Linked Representation of Graph

- Linked representation contains two lists:
  1. A node list **NODE**: each element corresponds to a node in G.



**NODE**: Name or key value of Node

**NEXT**: Pointer to the next Node

**ADJ**: Pointer to the first element in the adjacency list of node.

2. An edge list **EDGE**: each element corresponds to an edge in G.



**DEST**: Location of the terminal node of edge in NODE.

**LINK**: link together the edges with the same initial node.



# Questions

# Data Structures

Topic: Graph Traversal (BFS and DFS)



By  
**Ravi Kant Sahu**

*Asst. Professor,*

Lovely Professional University, Punjab



# Contents

- Introduction
- Graph Traversal
  - BFS
  - DFS
- Review Questions



# Introduction

- There are two different ways of Traversing a Graph.
  - Breadth First Search (BFS) : uses Queue
  - Depth First Search (DFS) : uses Stack
- During Traversal, each node  $N$  of  $G$  will be in one of the three states:
  - STATUS = 1: Ready State (initial state)
  - STATUS = 2: Waiting State (waiting in Queue/Stack)
  - STATUS = 3: Processed State



# Example

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Breadth First Search

1. Initialize all nodes to Ready State (STATUS = 1).
2. Put the starting node A in Queue and change its status to Waiting State (STATUS = 2).
3. Repeat step 4 and 5 until Queue is empty:
  4. Remove the front node N of the Queue.  
Process N and set the status of N to STATUS=3.
  5. Add to the Rear of Queue all the neighbors of N that are in STATUS = 1. and change their status to STATUS = 2.  
[End of step 3 Loop.]
6. Exit.



# Example

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Depth First Search

1. Initialize all nodes to Ready State (STATUS = 1).
2. Push the starting node A onto STACK and change its status to Waiting State (STATUS = 2).
3. Repeat step 4 and 5 until STACK is empty:
  4. POP the TOP node N from the STACK.  
Process N and set the status of N to STATUS=3.
  5. PUSH onto STACK all the neighbors of N  
that are in STATUS = 1, and change their status to  
STATUS = 2.  
[End of step 3 Loop.]
6. Exit.



# Example

Ravi Kant Sahu, Asst. Professor @ LPU Phagwara (Punjab) India



# Questions



# Review Questions

- What do you mean by Heap?
- What is the complexity of Heap sort?
- Calculate the complexity of Heap sort.
- How will you insert an element in a heap?
- What is the difference between Maxheap and Minheap?