



BITS Pilani
Pilani Campus

Course Name : Data Structures & Algorithms

Bharat Deshpande
Computer Science & Information Systems



What is a program?

- **Algorithm**

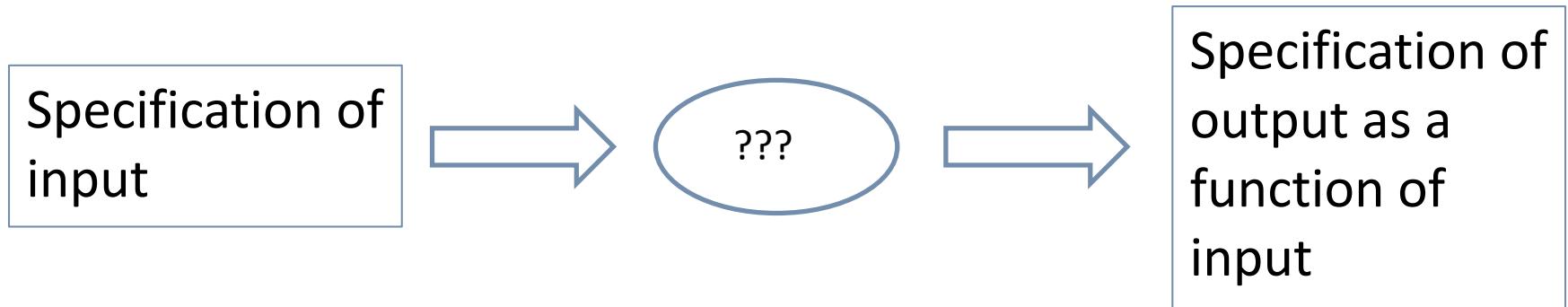
An algorithm is a step-by-step procedure for solving a problem in a finite amount of time.

- **Data Structures**

Is a systematic way of organizing and accessing data, so that data can be used efficiently.

Algorithms + Data Structures = Program

Algorithmic problem

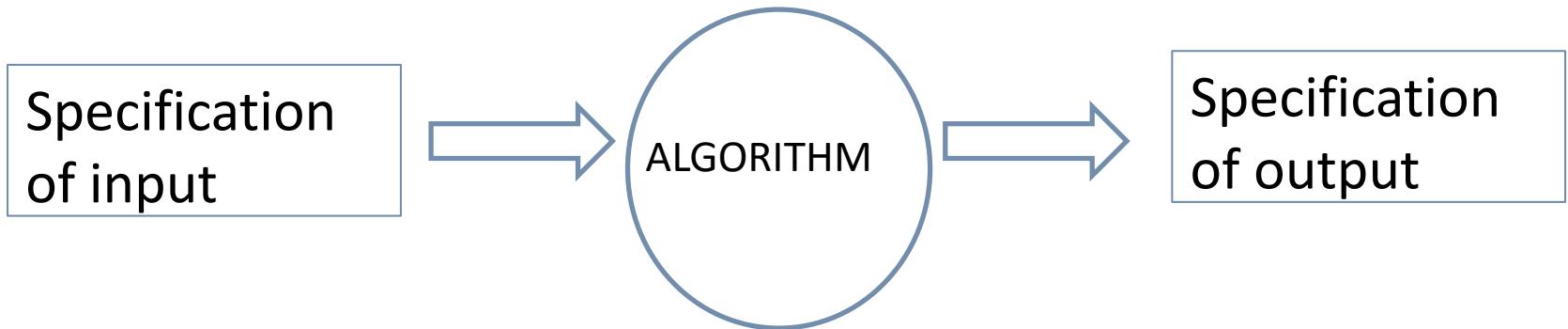


For eg: Sorting of integers

Input Instance : 8,4 ,5,2,10

Output Instance as a permutation of input : 2,4,5,8,10

Algorithmic Solution



- Algorithm describes actions on the input instance.
- Infinitely many correct algorithm for the same problem.

Infinite number of input instances satisfying the specification.

Two key points: Repeatable argument & Correctness

What is good algorithm?

- Resources Used
 - Running time
 - Space used
- Resource Usage
 - Measured proportional to (input) size

Measuring the running time

- Write a program implementing the algorithm.
- Run the program with inputs of varying size and composition.
- Use a method like `System.currentTimeMillis()` to get an accurate measure of the actual running time.

Limitations of experimental studies

- Implementation is a must.
- Execution is possible on limited set of inputs.
- If we need to compare two algorithms we need to use the same environment
(like hardware, software etc)

Analytical model to analyze algorithm



- Algorithm should be analyzed by using general methodology.
- This approach uses:
 - High level description of the algorithm.
 - Takes into account all possible inputs.
 - Allows one to evaluate the efficiency of any algorithm in a way that is independent of the hardware and the software environment.

- A mixture of natural language and high level programming concepts that describes the main ideas behind a generic implementation of a data structure and algorithms.

Algorithm *arrayMax(A, n)*

Input: An array *A* of *n integers*

Output: The maximum element of *A*

currentMax $\leftarrow A[0]$

for *i* $\leftarrow 1$ to *n - 1* do

if *A[i] > currentMax* then *currentMax* $\leftarrow A[i]$

return currentMax

- Is structured than usual prose but less formal than a programming language.
- Expressions
 - Use standard mathematical symbols to describe numeric and Boolean expressions.
 - Uses \leftarrow for assignment.
 - Use = for the equality relationship.
- Method declaration
 - Algorithm name(param1,param2...)

Assumptions

Individual statement considered as “unit” time

- Not applicable for function calls and loops

Individual variable considered as “unit” storage

Often referred to as “algorithmic complexity”

Complexity Example [1]

Example 1 (Y and Z are input)

X = Y * Z;

X = Y * X + Z;

// 2 units of time and 1 unit of storage

// Constant Unit of time and Constant Unit of
storage

Complexity Example [2]

Example 2 (a and N are input)

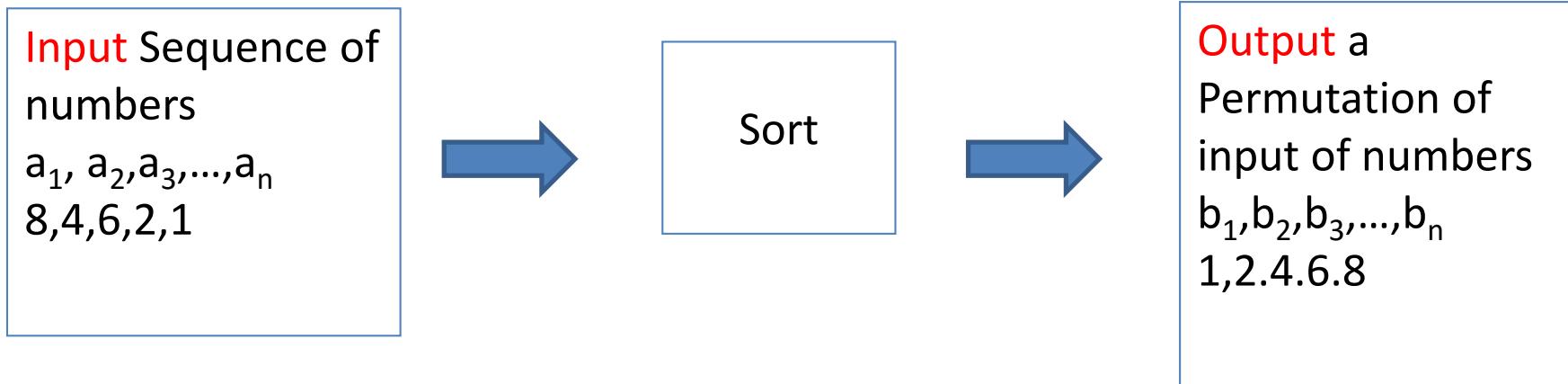
```
j = 0;  
while (j < N) do  
    a[j] = a[j] * a[j];  
    b[j] = a[j] + j;  
    j = j + 1;  
endwhile;  
// 3N + 1 units of time and N+1 units of storage  
// time units prop. to N and storage prop. to N
```

Complexity Example [3]

Example 3 (a and N are input)

```
j = 0;  
while (j < N) do  
    k = 0;  
    while (k < N) do  
        a[k] = a[j] + a[k];  
        k = k + 1;  
    endwhile;  
    b[j] = a[j] + j;  
    j = j + 1;  
endwhile;  
//??? units of time and ??? units of storage  
// time prop. to N2 and storage prop. to N
```

Example of sorting



Correctness(Requirement for the output)

For any input algorithm halts with the output:

- $b_1 < b_2 < b_3 < \dots < b_n$
- $b_1, b_2, b_3, \dots, b_n$ is a permutation of $a_1, a_2, a_3, \dots, a_n$

Running time of algorithm depends on

- Number of elements n.
- How (partially)sorted they are.

Order Notation

- Purpose
 - Capture proportionality
 - Machine independent measurement
 - Asymptotic growth
(i.e. large values of input size N)

Motivation for Order Notation

Examples

- $100 * \log_2 N < N$ for $N > 1000$
- $70 * N + 3000 < N^2$ for $N > 100$
- $10^5 * N^2 + 10^6 * N < 2^N$ for $N > 26$

Asymptotic Analysis

- Goal: To simplify analysis of running time of algorithm .eg $3n^2=n^2$.
- Capturing the essence: how the running time of the algorithm increases with the size of the input in the limit.

Asymptotic Notation

- The big O notation

Definition

Let f and g be functions from the set of integers to the set of real numbers. We say that $f(x)$ is in $O(g(x))$ if there are constants $C >$ and k such that $|f(x)| \leq C |g(x)|$, whenever $x \geq k$.

- This is read as $f(x)$ is **big-oh** of $g(x)$

Note: Pair of C and k is never unique.

Order Notation

Examples

$$g(n) = 17*N + 5$$

$$\lim_{n \rightarrow \infty} g(n) / f(n) = c$$

$\lim_{n \rightarrow \infty} (17*N + 5)/N = 17$. The asymptotic complexity is $O(N)$

$$g(n) = 5*N^3 + 10*N^2 + 3$$

$\lim_{n \rightarrow \infty} (5*N^3 + 10*N^2 + 3) / N^3 = 5$. The asymptotic complexity is $O(N^3)$

$$g(n) = C_1*N^k + C_2*N^{k-1} + \dots + C_k*N + C$$

$$\lim_{n \rightarrow \infty} (C_1*N^k + C_2*N^{k-1} + \dots + C_k*N + C) / N^k = C_1.$$

The asymptotic complexity is $O(N^k)$

$$2^N + 4*N^3 + 16 \text{ is } O(2^N)$$

$$5*N*\log(N) + 3*N \text{ is } O(N*\log(N))$$

$$1789 \text{ is } O(1)$$

Linear Search

```
function search(X, A, N)
    j = 0;
    while (j < N)
        if (A[j] == X) return j;
        j++;
    endwhile;
    return “Not-found”;
```

Linear Search - Complexity

Time Complexity

“if” statement introduces possibilities

- Best-case: $O(1)$
- Worst case: $O(N)$
- Average case: ???

Binary Search Algorithm

Assume: Sorted Sequence of numbers

```
low = 1; high = N;  
while (low <= high) do  
    mid = (low + high) /2;  
    if (A[mid] == x) return x;  
    else if (A[mid] < x) low = mid +1;  
    else high = mid - 1;  
endwhile;  
return Not-Found;
```

Binary Search - Complexity

- Best Case
 - $O(1)$
- Worst case:
 - Loop executes until **low \leq high**
 - Size halved in each iteration
 - $N, N/2, N/4, \dots 1$
 - How many steps ?

Binary Search - Complexity

- Worst case:
 - K steps such that $2^K = N$
i.e. $\log_2 N$ steps is $O(\log(N))$



BITS Pilani
Pilani Campus

Course Name : **Data Structures & Algorithms**

Bharat Deshpande
Computer Science & Information Systems

Abstract Data Type

- Abstract Data Type

In computer science, an **abstract data type (ADT)** is a mathematical model for a certain class of data structures that have similar behavior.

Abstract Data Types (ADTs)

- A method for achieving abstraction for data structures and algorithms
- ADT = model + operations
- Describes what each operation does, but not how it does it
- An ADT is independent of its implementation

Abstract Data Types

- Typical operations on data
 - Add data to a data collection
 - Remove data from a data collection
 - Ask questions about the data in a data collection

Abstract Data Types

- Data abstraction
 - Asks you to think *what* you can do to a collection of data independently of *how* you do it
 - Allows you to develop each data structure in relative isolation from the rest of the solution
 - A natural extension of procedural abstraction

Examples

- **Simple ADTs**

- *Stack*
- *Queue*
- Vector
- Lists
- Sequences
- Iterators

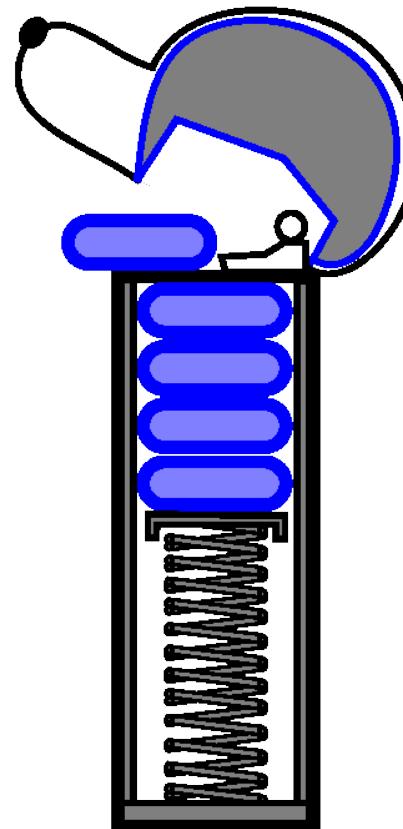
All these are called **Linear Data Structures**

Stacks

- A **stack** is a container of objects that are inserted and removed according to the last-in-first-out (**LIFO**) principle.
- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as “**pushing**” onto the stack. “**Popping**” off the stack is synonymous with removing an item.

Stacks

- A coin dispenser as an analogy:



Stacks: An Array Implementation

- Create a stack using an array by specifying a maximum size N for our stack.
- The stack consists of an N -element array S and an integer variable t , the index of the top element in array S .



- Array indices start at 0, so we initialize t to -1

Stacks: An Array Implementation

- **Pseudo code**

```
Algorithm size()
return t+1
```

```
Algorithm isEmpty()
return (t<0)
```

```
Algorithm top()
if isEmpty() then
    return Error
return S[t]
```

```
Algorithm push(o)
if size() == N then
    return Error
t = t + 1
S[t] = o
```

```
Algorithm pop()
if isEmpty() then
    return Error
t = t - 1
return S[t+1]
```



Stacks: An Array Implementation

The array implementation is simple and efficient (methods performed in $O(1)$).

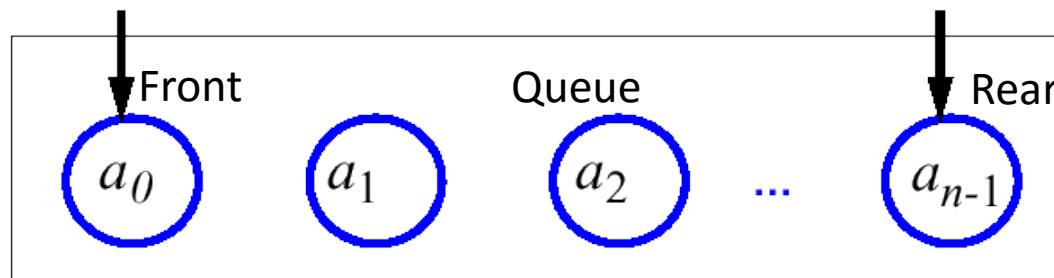
Disadvantage

There is an upper bound, N , on the size of the stack.

The arbitrary value N may be too small for a given application **OR**
a waste of memory.

Queues

- A queue differs from a stack in that its insertion and removal routines follows the **first-in-first-out** (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the **rear** (enqueued) and removed from the **front** (dequeued)



Queues

- The **queue** supports three fundamental methods:
 - **New():ADT** – *Creates an empty queue*
 - **Enqueue(S:ADT, o:element):ADT** - Inserts object *o* at the rear of the queue
 - **Dequeue(S:ADT):ADT** - Removes the object from the front of the queue; an error occurs if the queue is empty
 - **Front(S:ADT):element** - Returns, but does not remove, the front element; an error occurs if the queue is empty

Queues: An Array Implementation

- Create a queue using an array in a circular fashion
- A maximum size N is specified.
- The queue consists of an N -element array Q and two integer variables:
 - f , index of the front element (head – for dequeue)
 - r , index of the element after the rear one (tail – for enqueue)
 - Initially, $f=r=0$ and the queue is empty if $f=r$



Queues

Disadvantage

Repeatedly enqueue and dequeue a single element N times.

Finally, $f=r=N$.

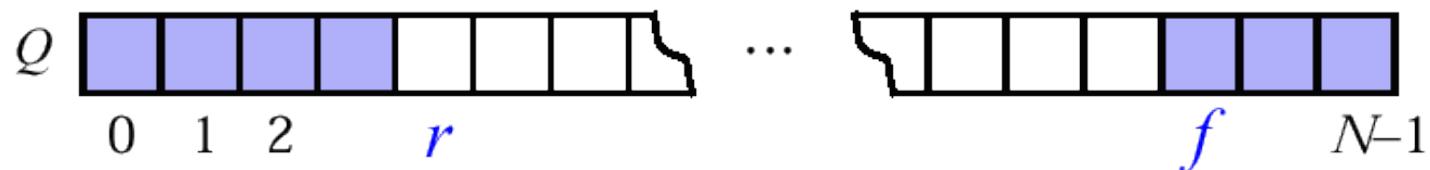
- No more elements can be added to the queue, though there is space in the queue.

Solution

Let f and r wraparound the end of queue.

Queues: An Array Implementation

“wrapped around” configuration



- Each time r or f is incremented, compute this increment as $(r+1)\text{mod}N$ or $(f+1)\text{mod}N$

Queues: An Array Implementation

- Pseudo code

```
Algorithm size()
return ( $N-f+r$ ) mod  $N$ 
```

```
Algorithm isEmpty()
return ( $f=r$ )
```

```
Algorithm front()
if isEmpty() then
    return Error
return  $Q[f]$ 
```

```
Algorithm dequeue()
if isEmpty() then
    return Error
 $Q[f]=\text{null}$ 
 $f=(f+1) \bmod N$ 
```

```
Algorithm enqueue(o)
if size =  $N - 1$  then
    return Error
 $Q[r]=o$ 
 $r=(r + 1) \bmod N$ 
```

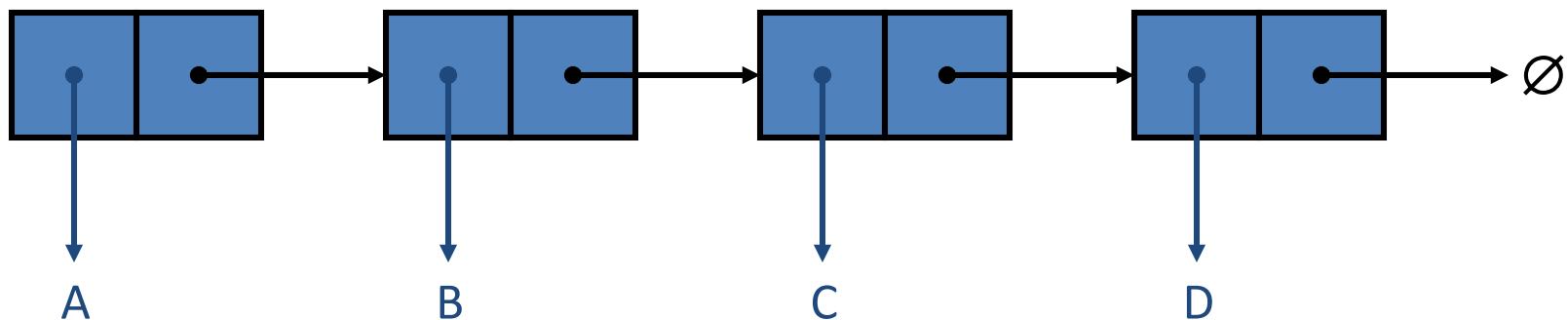
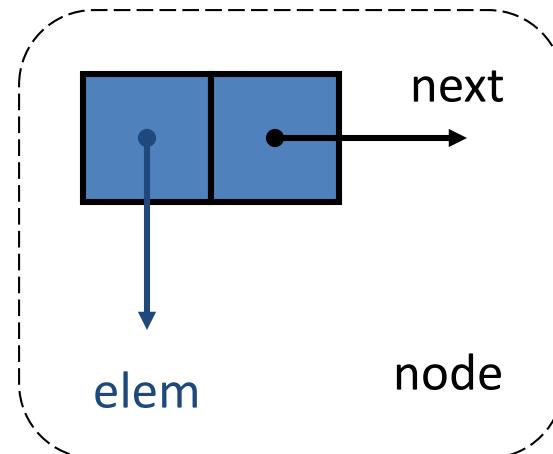
Arrays: pluses and minuses

- + Fast element access.
- Impossible to resize.

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

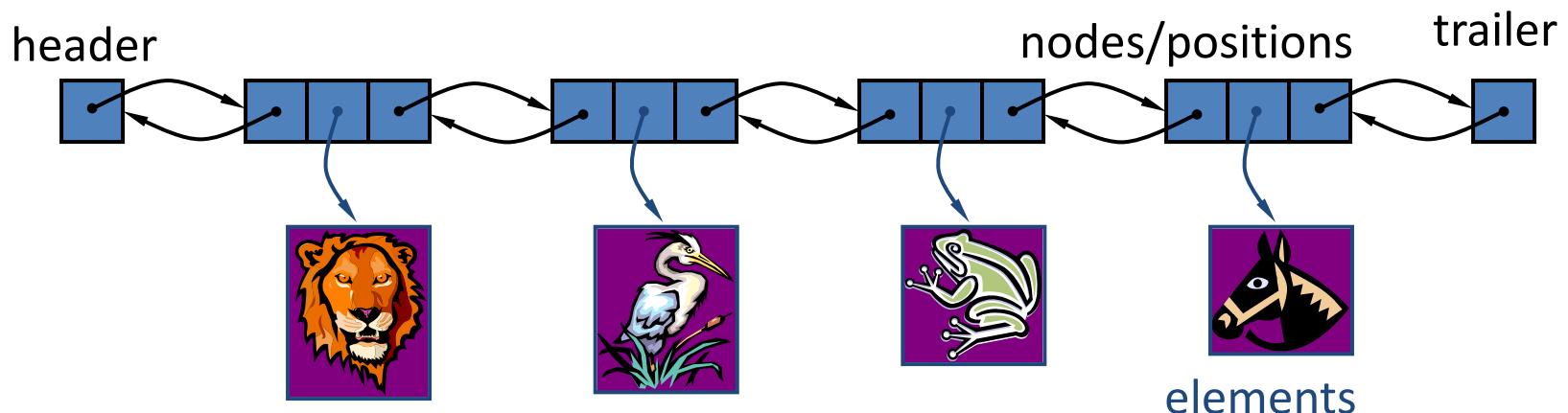
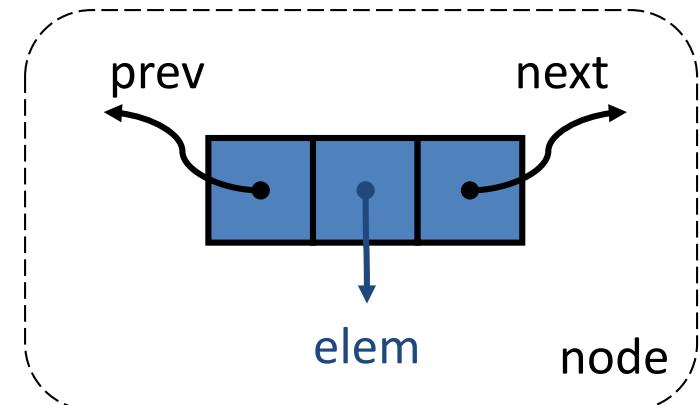
Singly Linked Lists

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



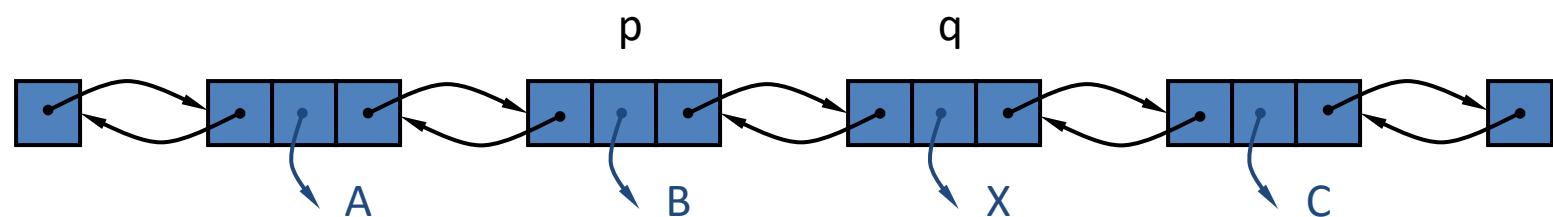
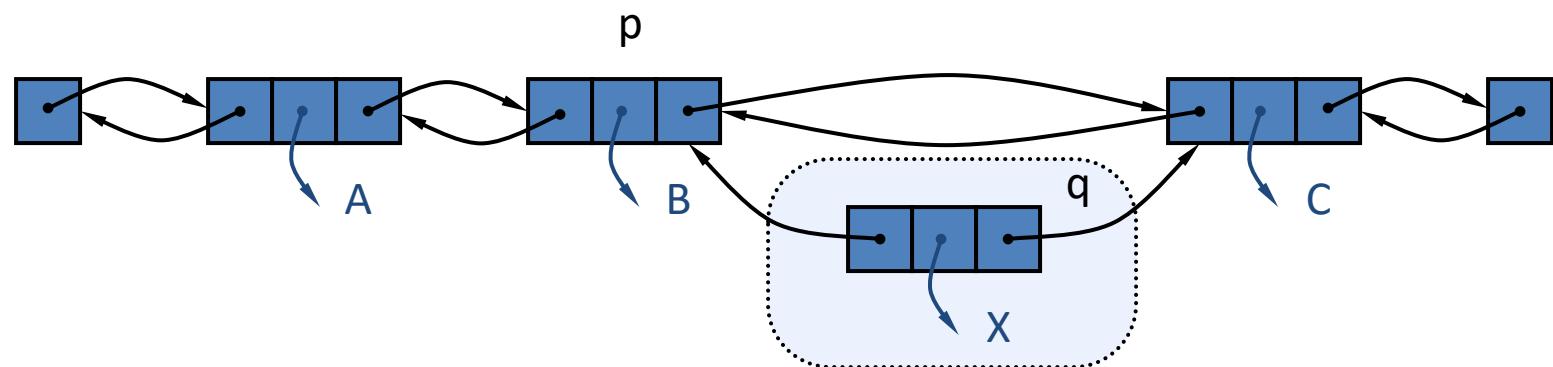
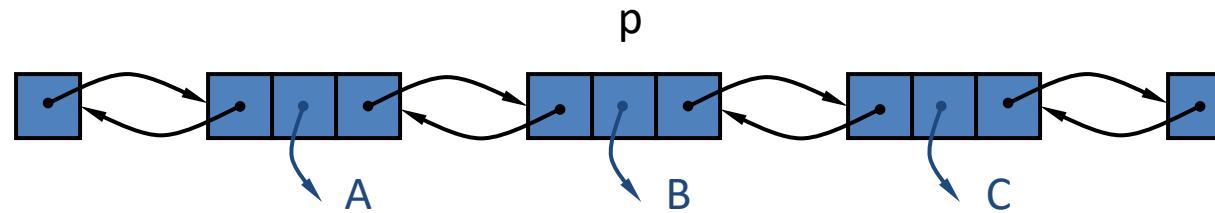
Doubly Linked List

- A doubly linked list is often more convenient!
- Nodes store:
 - element
 - link to the previous node
 - link to the next node
- Special trailer and header nodes



Insertion

- We visualize operation `insertAfter(p, X)`, which returns position q



Insertion Algorithm

Algorithm insertAfter(p, e):

Create a new node v

$v.setElement(e)$

$v.setPrev(p)$ {link v to its predecessor}

$v.setNext(p.getNext())$ {link v to its successor}

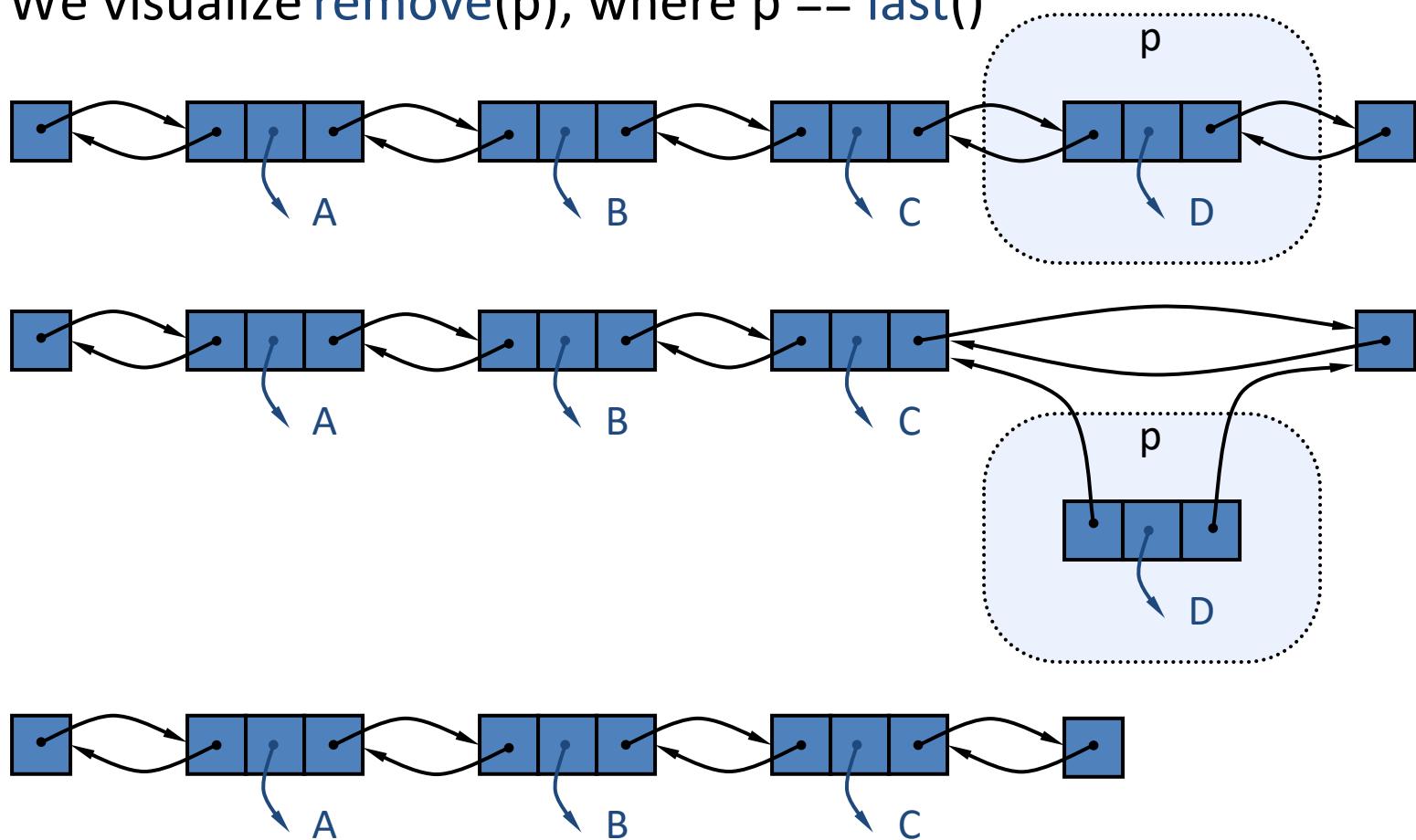
$(p.getNext()).setPrev(v)$ {link p 's old successor to v }

$p.setNext(v)$ {link p to its new successor, v }

return v {the position for the element e }

Deletion

- We visualize `remove(p)`, where `p == last()`



Deletion Algorithm

Algorithm remove(*p*):

t = *p.element* {a temporary variable to hold the
return value}

(*p.getPrev()*).setNext(*p.getNext()*) {linking out *p*}

(*p.getNext()*).setPrev(*p.getPrev()*)

p.setPrev(null) {invalidating the position *p*}

p.setNext(null)

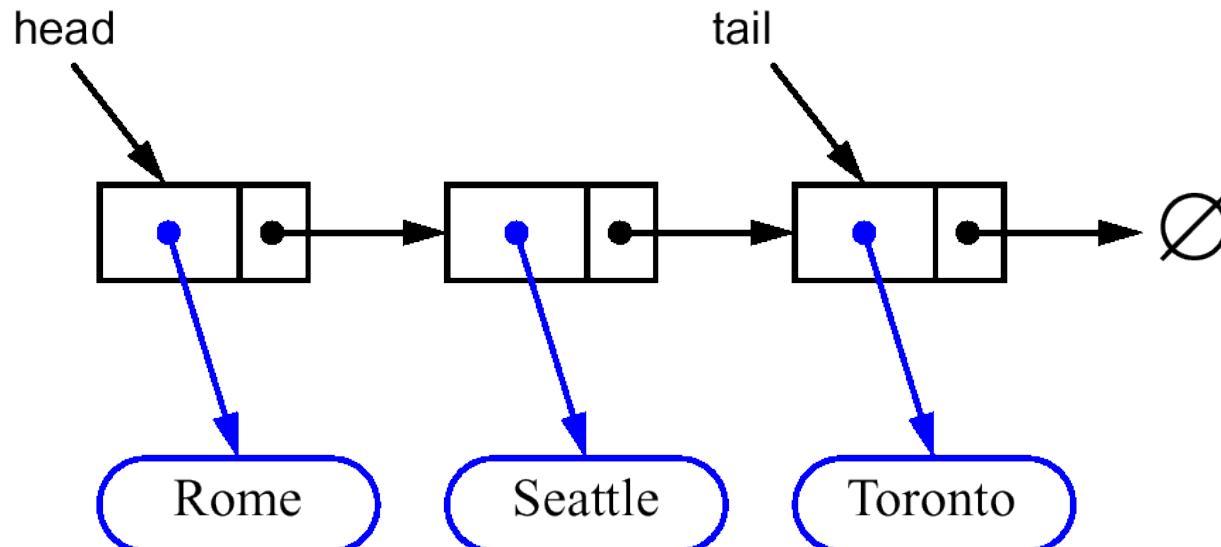
return *t*

Worst-case running time

- In a doubly linked list
 - + insertion at head or tail is in $O(1)$
 - + deletion at either end is on $O(1)$
 - element access is still in $O(n)$

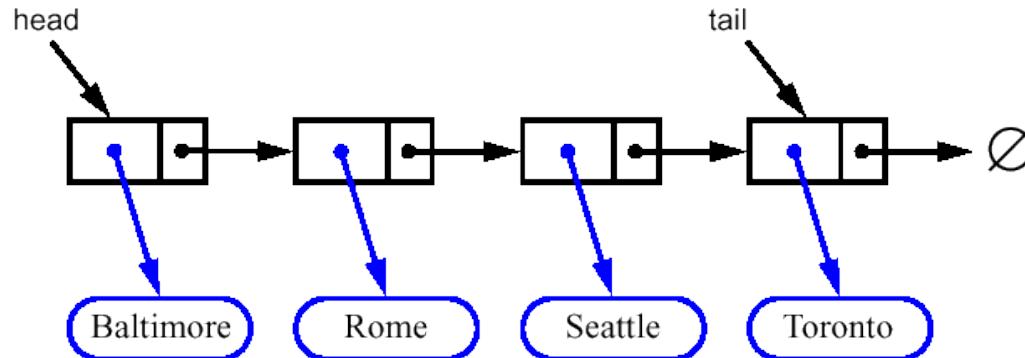
Stacks: Singly Linked List implementation

- Nodes (*data, pointer*) connected in a chain by links

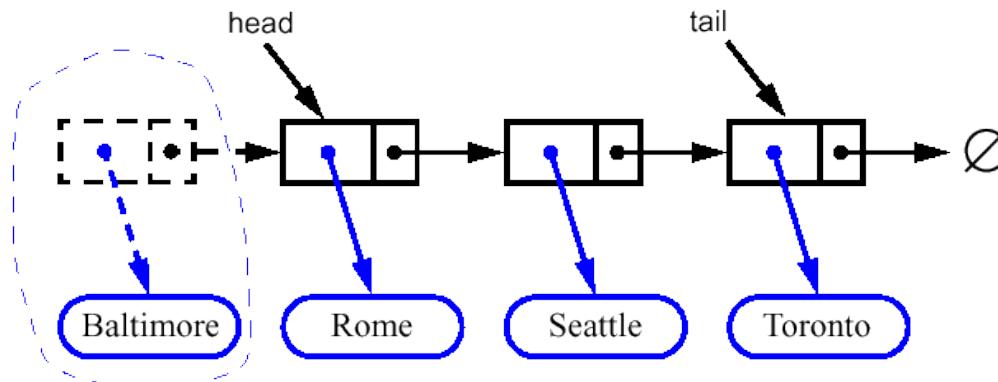


- the head or the tail of the list could serve as the top of the stack

Queues: Linked List Implementation



- Dequeue - advance head reference





BITS Pilani
Pilani Campus

Course Name : **Data Structures & Algorithms**

Bharat Deshpande
Computer Science & Information Systems

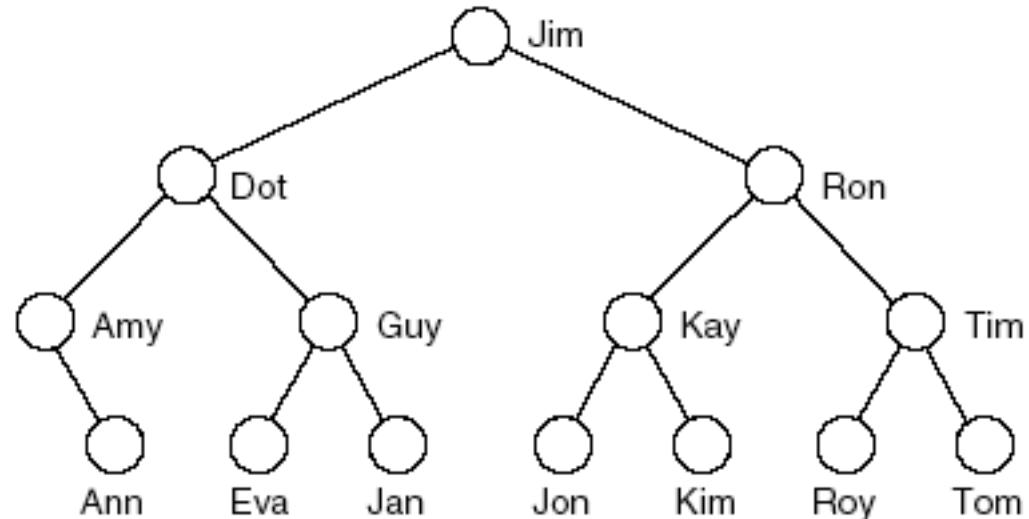
- Here are some of the data structures we have studied so far:
 - Arrays
 - Singly-linked lists and doubly-linked lists
 - Stacks and queues
 - These all have the property that their elements can be adequately displayed in a straight line
- **How to obtain data structures for data that have nonlinear relationships**

Tree

Tree represents hierarchy.

Examples of trees:

- Directory tree
- Family tree
- Company organization chart
- Table of contents

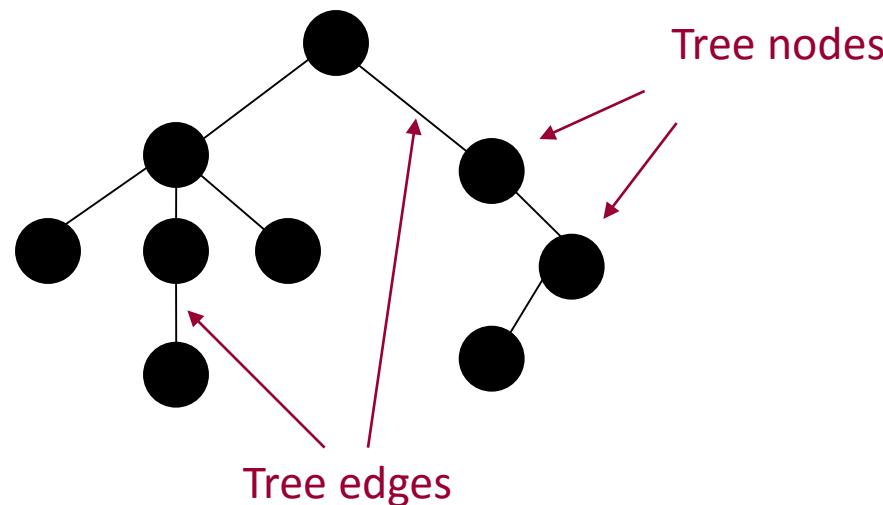


-structure resembles branches of a “tree”, hence the name.

Trees

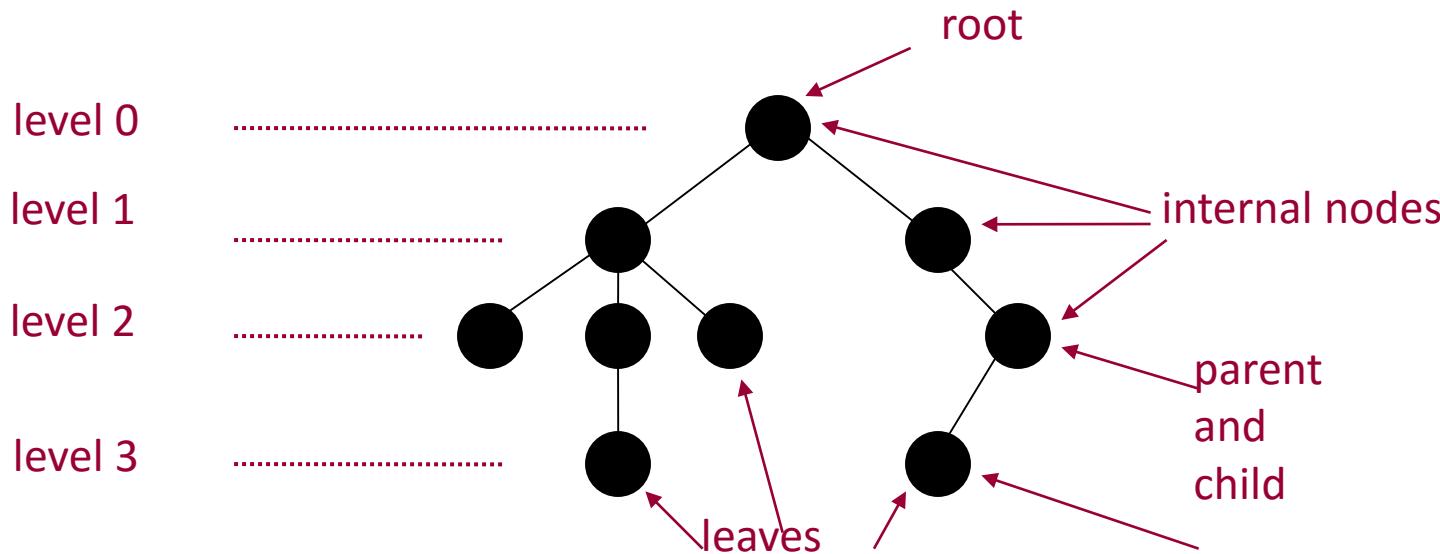
Trees have **nodes**. They also have **edges** that connect the nodes.

- Between two nodes there is *always only one* path.



Trees: More Definitions

- Trees that we consider are rooted. Once the **root** is defined (by the user) all nodes have a specific **level**.
- Trees have **internal nodes** and **leaves**. Every node (except the root) has a **parent** and it also has zero or more **children**.



Tree Terminology (1)

A **vertex (or node)** is a object that can have a name and can carry other associated information

- The first or top node in a tree is called the **root** node.
 - An **edge** is a connection between two vertices
 - A **path** in a tree is a list of distinct vertices in which successive vertices are connected by edges in the tree.
 - The defining property of a tree is that there is precisely one path connecting any two nodes.
 - A disjoint set of trees is called a **forest**
 - Nodes with no children are **leaves, terminal or external nodes**
-

Tree Terminology (2)

Child of a node u :- Any node reachable from u by 1 edge.

Parent node :- If b is a child of a, then a is the parent of b.

- All nodes except root have exactly one parent.

Subtree:-any node of a tree, with all of its descendants.

Depth of a node :

- Depth of root node is 0.

-Depth of any other node is 1 greater than depth of its parent.

Tree Terminology (3)

The size of a tree is the number of nodes in it

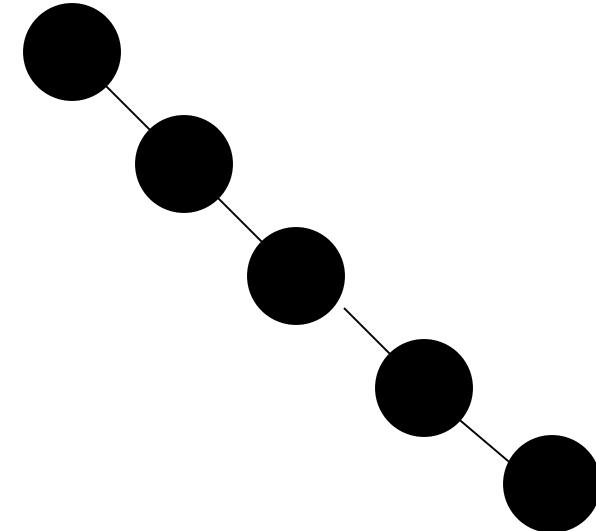
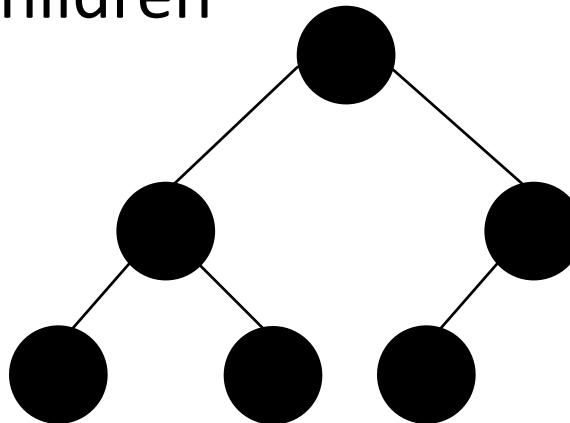
Height : Maximum of all depths.

Each node except the root has exactly one node above it in the tree, (i.e. its parent), and we extend the family analogy talking of children, siblings, or grandparents
Nodes that share parents are called **siblings**.

Binary Trees

Definition: A binary tree is either empty or it consists of a root together with two binary trees called the left subtree and the right subtree.

A **binary tree** is a tree in which each node has *atmost* 2 children



Properties of Binary trees

Proper Binary Tree

- Each internal node has exactly two children

Let

n - number of nodes

n_e – number of external nodes or leaves

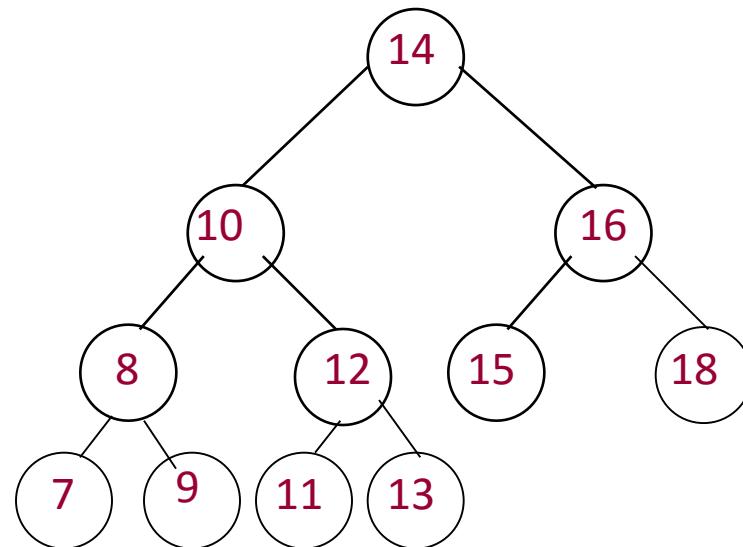
n_i – number of internal nodes

h – height of T , Then the following holds:

- $h+1 \leq n_e \leq 2^h$
- $h \leq n_i \leq 2^h - 1$
- $h + 1 \leq n \leq 2^{h+1} - 1$
- $\log(n+1) - 1 \leq h \leq (n - 1)/2$

Complete Binary Trees

- Nodes in trees can contain **keys** (letters, numbers, etc)
- **Complete binary tree**: A binary tree in which every level, except possibly the deepest, is completely filled. At depth n , the height of the tree, all nodes must be as far left as possible.



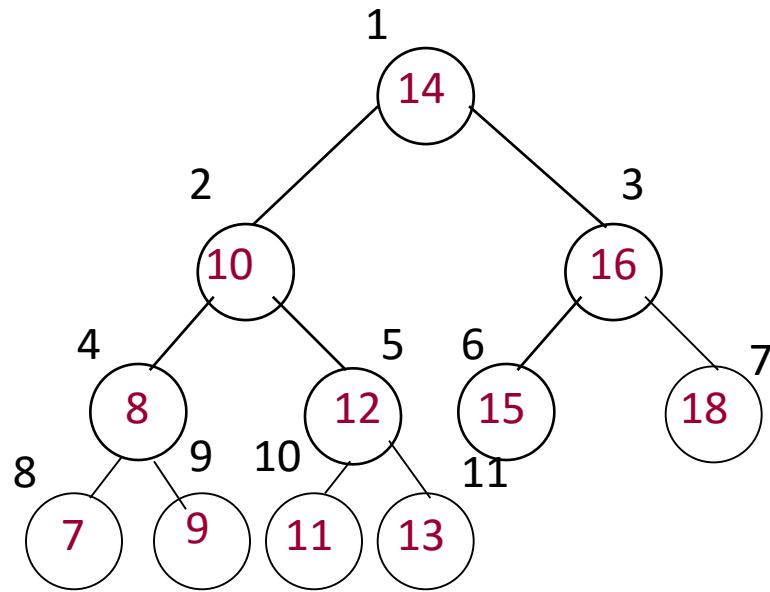
Complete Binary Trees: Array Representation



Complete Binary Trees can be represented in memory with the use of an array A so that all nodes can be accessed in O(1) time:

- Label nodes sequentially top-to-bottom and left-to-right
- Left child of A[i] is at position A[2i]
- Right child of A[i] is at position A[2i + 1]
- Parent of A[i] is at A[i/2]

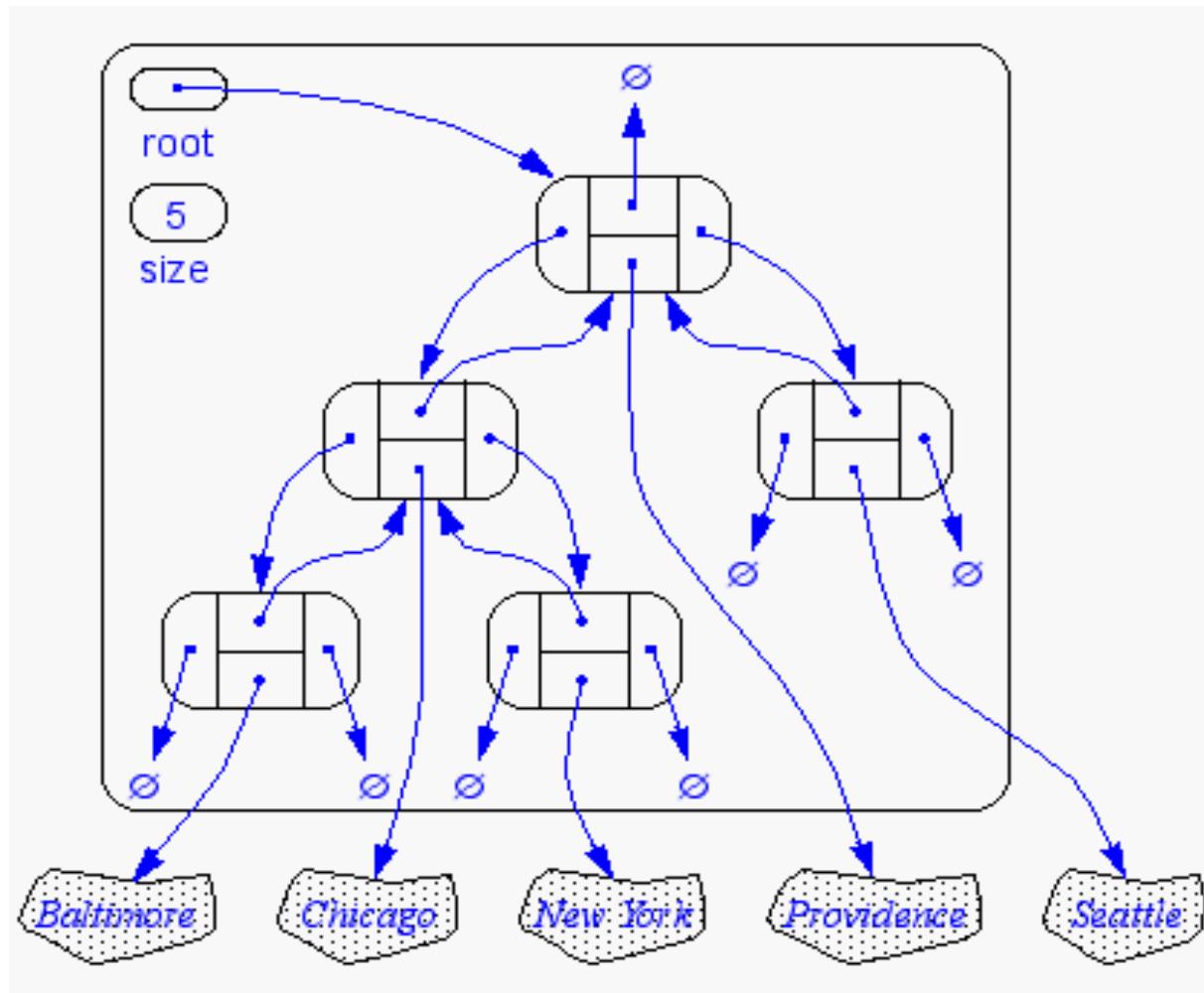
Complete Binary Trees: Array Representation



A **Binary tree** is a linked data structure. Each node contains data (including a key and satellite data), and pointers left, right and p.

- ***Left points to the left child of the node.***
- ***Right points to the right child of the node.***
- ***p points to the parent of the node.***
- If a child is missing, the pointer is NIL.
- If a parent is missing, p is NIL.
- The **root** of the tree is the only node for which p is NIL.
- Nodes for which both left and right are NIL are **leaves**.

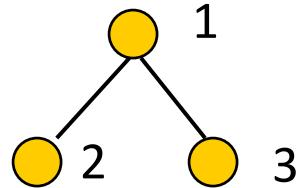
Binary Trees: Linked List Representation



Binary Tree Traversals

- A binary tree is defined recursively: it consists of a root, a left subtree and a right subtree
- To **traverse (or walk)** the binary tree is to visit each node in the binary tree exactly once.
- Tree traversals are naturally recursive.
- Since a binary tree has three parts, there are six possible ways to traverse the binary tree:
 - root, left, right : preorder (root, right, left)
 - left, root, right: inorder (right, root, left)
 - left, right, root: postorder (right, left, root)

Binary Tree Traversals



preorder: 1 2 3

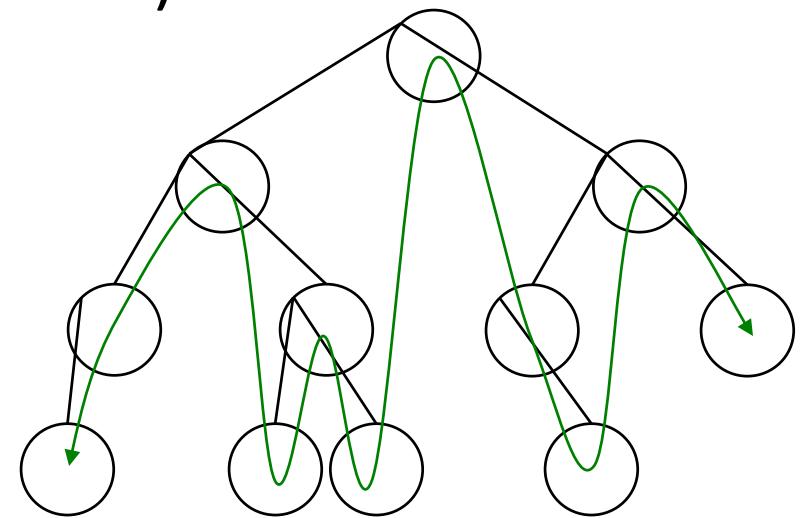
inorder: 2 1 3

postorder: 2 3 1

Tree Traversal: InOrder

In-order traversal

- Visit left subtree (if there is one) In Order
- print the key of the current node
- Visit right subtree (if there is one) In Order

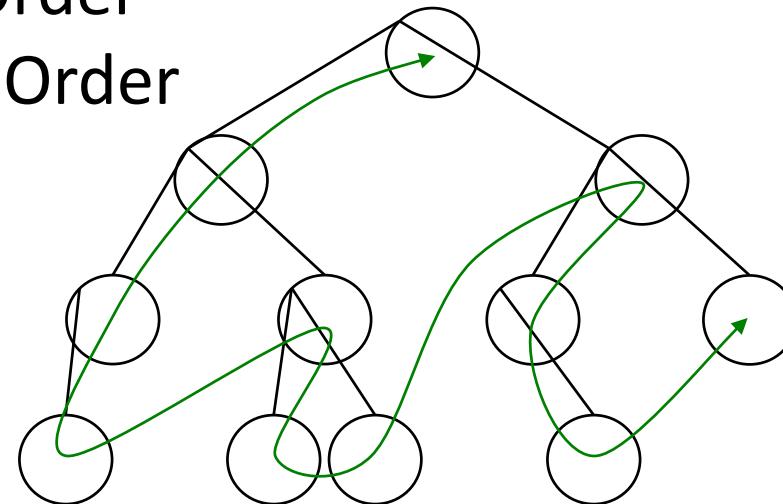


Tree Traversal: PreOrder

Another common traversal is **PreOrder**.

It goes as deep as possible (visiting as it goes) then left to right

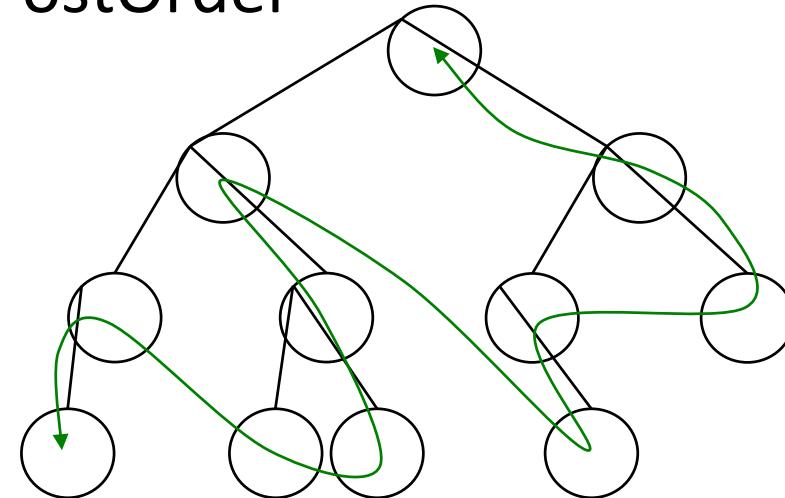
- print root
- Visit left subtree in PreOrder
- Visit right subtree in PreOrder



Tree Traversal: PostOrder

PostOrder traversal also goes as deep as possible, but only visits internal nodes during backtracking.
recursive:

- Visit left subtree in PostOrder
- Visit right subtree in PostOrder
- print root

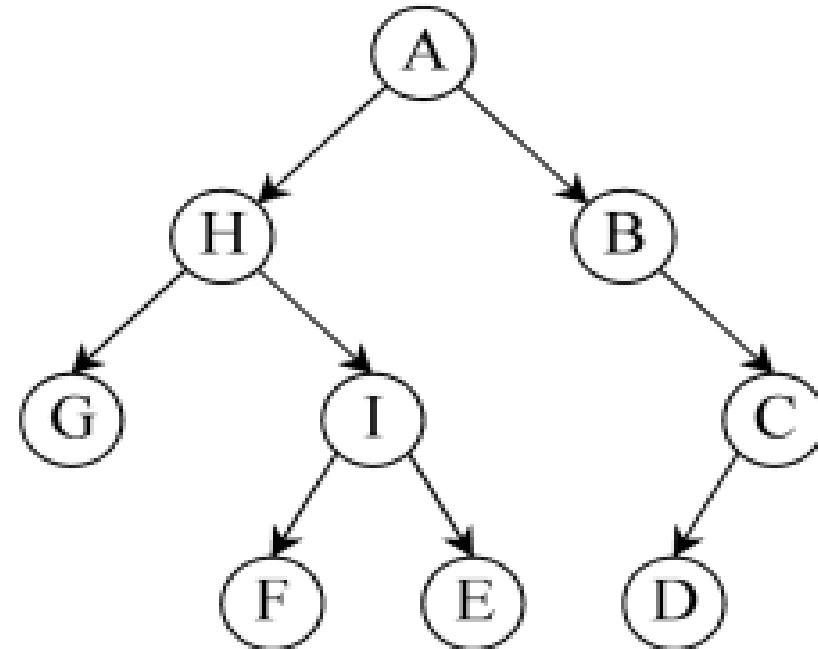


Tree Traversal: examples

Preorder (DLR) traversal yields: A, H, G, I, F, E, B, C, D

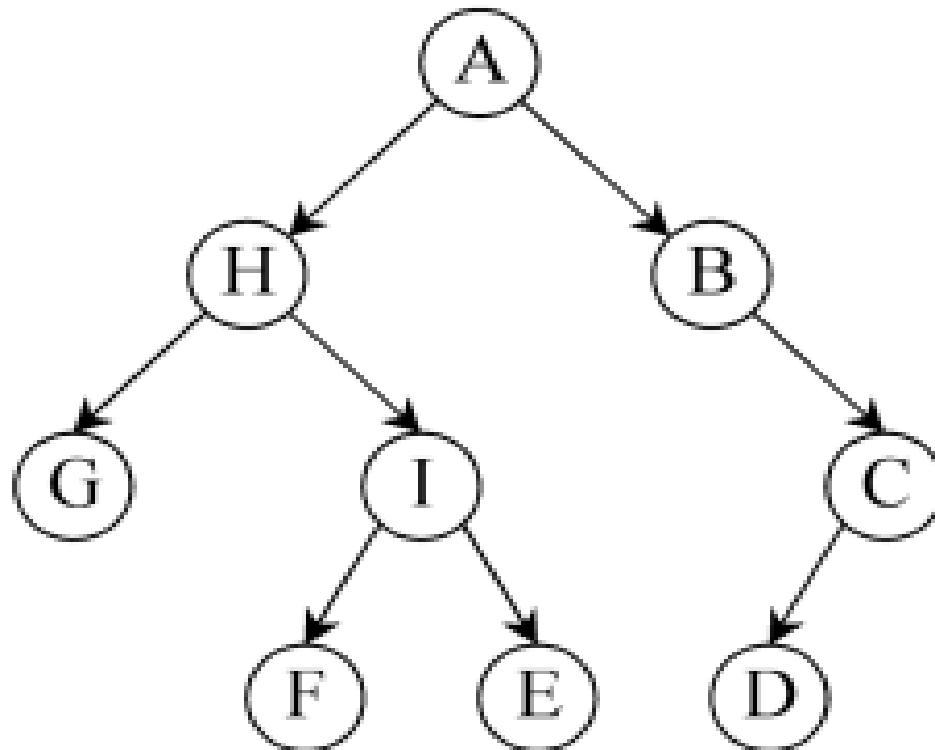
Postorder (LRD) traversal yields: G, F, E, I, H, D, C, B,

In-order (LDR) traversal yields: G, H, F, I, E, A, B, D, C



Tree Traversal: examples

- * Preorder (DLR) traversal yields: A, H, G, I, F, E, B, C, D
- * Postorder (LRD) traversal yields: G, F, E, I, H, D, C, B, A
- * In-order (LDR) traversal yields: G, H, F, I, E, A, B, D, C





BITS Pilani
Pilani Campus

Course Name : **Data Structures & Algorithms**

Bharat Deshpande
Computer Science & Information Systems

Sorting Problem

- Input : A sequence of n numbers
 $\langle a_1, a_2, \dots, a_n \rangle$
- Output : A permutation (reordering)
 $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$
- Solutions : Many!
- First Solution : “Insertion Sort”

Insertion Sort

- **Big idea:**
 - Inserting an element into a sorted list in the appropriate position retains the order.
 - Works the way many people sort a hand of playing cards.
 - Start with an empty left hand and the cards face down on the table.
 - We remove one card from the table and insert it in the correct position in left hand.

Insertion Sort

- To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left.
- **Important :**
At all times the cards in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

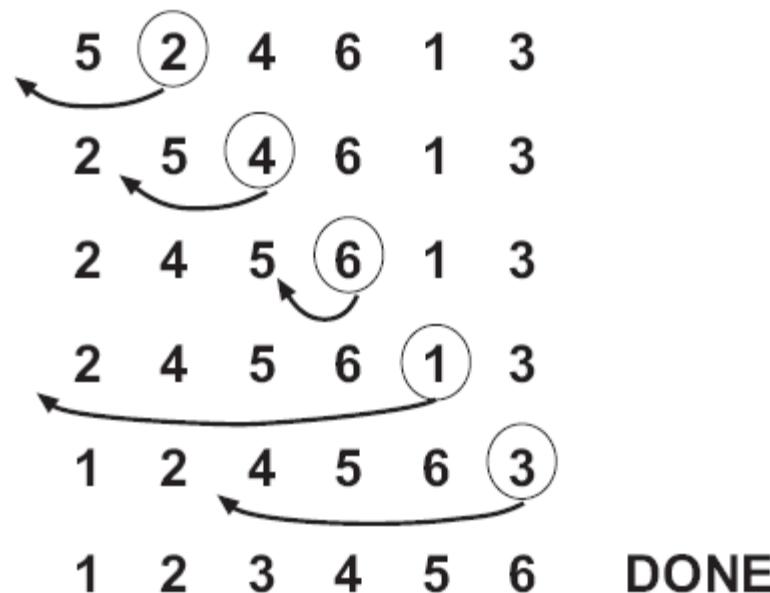
Insertion Sort

Crucial Idea

- Start with a singleton list – sorted trivially.
- Repeatedly insert elements – one at a time
 - while keeping it sorted.
- Initially, x will need to be the second element and a[1] the ‘sorted part’.
- Sorted part is extended by first inserting the 2nd element, then the 3rd & so on.

Insertion Sort (Con't)

- Example:



Insertion Sort – Pseudo Code

```
InsertionSort(A, n) {
    for i = 2 to n {
        key = A[i]
        *insert A[i] into the sorted sequence A[1,...,i-1]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
        }
        A[j+1] = key
    }
}
```

Insertion Sort - Analysis

- **Best Case Analysis**

The best case for insertion sort occurs when the list is already sorted.

In this case, insertion sort requires $n-1$ comparisons i.e., $O(n)$ complexity.

- **Worst Case Analysis**

for each value of i , what is the maximum number of key comparisons possible?

- Answer: $i - 1$

- Thus, the total time in the worst case is

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

Insertion Sort - Analysis

- **Average Case Analysis**
 - We assume that all permutations of the keys are equally likely as input.
 - We also assume that the keys are distinct.
 - We first determine how many key comparisons are done on average to insert one new element into the sorted segment.

Insertion Sort – Average Case

- When we deal with entry i , how far back must we go to insert it?

Answer:

There are i possible positions: not moving at all, moving by one position up to moving by $i - 1$ positions.

- Given randomness, these are equally likely.

Insertion Sort – Average Case

Average no. of comparisons

$$= \frac{1}{i} \sum_{j=1}^{i-1} j + \frac{i-1}{i} = \frac{i-1}{2} + 1 - \frac{1}{i}$$

Total =

$$\sum_{i=1}^{n-1} \left(\frac{i-1}{2} + 1 - \frac{1}{i} \right) = \frac{(n-1)(n-2)}{4} + n - 1 - \sum_{i=1}^{n-1} \frac{1}{i}$$

Insertion Sort – Average Case

- Well known

$$\sum_{i=1}^n \frac{1}{i} \approx \ln n$$

- Thus, the total number of comparisons
= $O(n^2)$

Merge Sort

- **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 a sorted list.
- The general algorithm for the merge sort is as follows:
- If the list is of size greater than 1, then
 - a. Find the mid-position of the list.
 - b. Merge sort the first sublist.
 - c. Merge sort the second sublist.
 - d. Merge the first sublist and the second sublist.

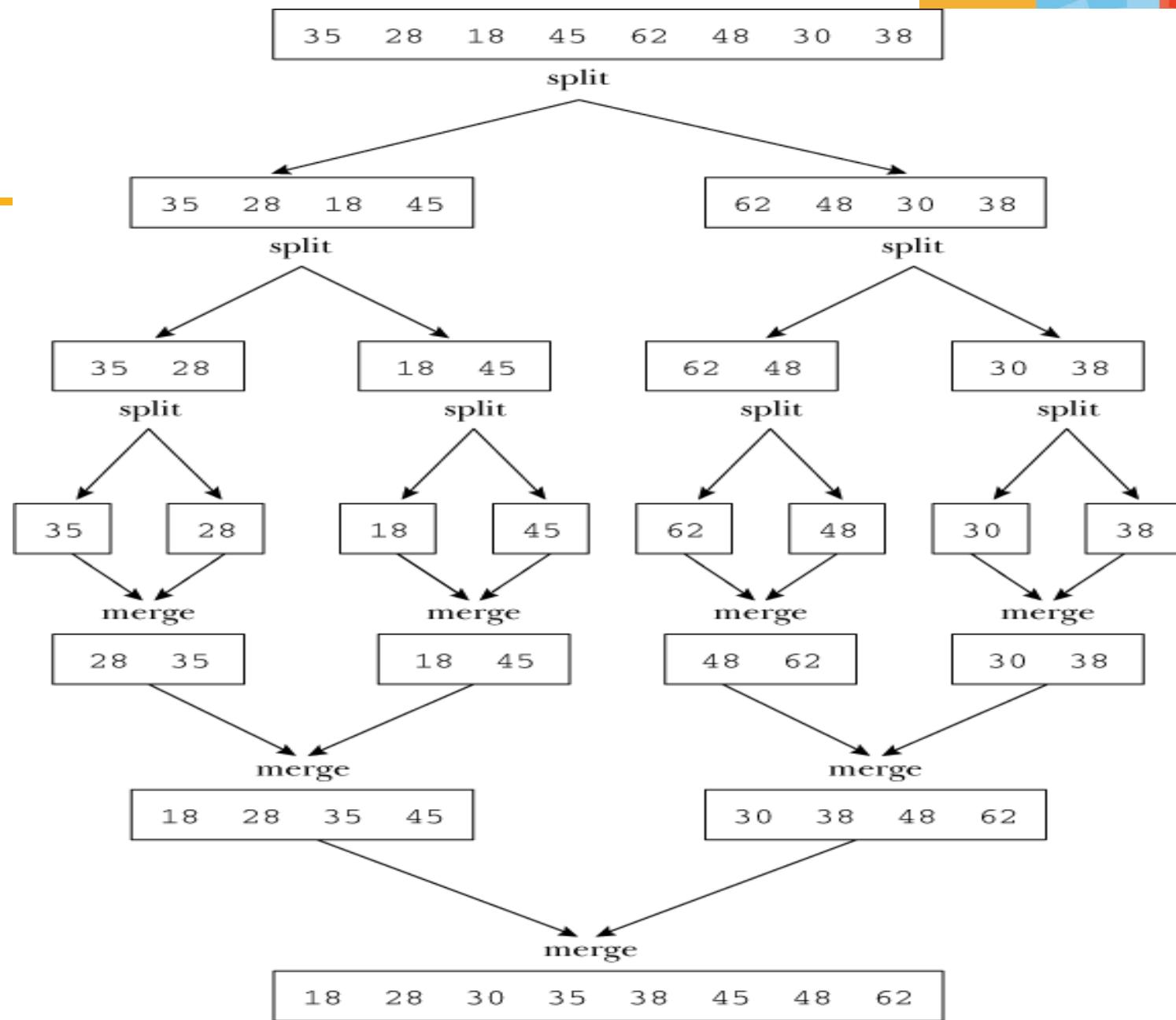


FIGURE 9.22 Merge sort process

Merging two sorted Array

- Once the sublists are sorted, the next step in the merge sort algorithm is to merge the sorted sublists.
- Suppose L_1 and L_2 are two sorted lists as follows:
 - L_1 : 2, 7, 16, 35
 - L_2 : 5, 20, 25, 40, 50
- Merge L_1 and L_2 into a third list, say L_3 .
- The merge process is as follows:
repeatedly compare, using a loop, the elements of L_1 with the elements of L_2 and copy the smaller element into L_3 .

Example

	i	
L_1	2 7 16 35	
	j	
L_2	5 20 25 40 50	
	k	
L_3		

Before Iteration 1

FIGURE 9.23 L_1 , L_2 , and L_3 before and after the first iteration

- First compare $L_1[1]$ with $L_2[1]$ and see that $L_1[1] < L_2[1]$, so copy $L_1[1]$ into $L_3[1]$

Time : If both the list has n elements each then the merging process takes $2n$ time in the worst case.

	i	
L_1	2 7 16 35	
	j	
L_2	5 20 25 40 50	
	k	
L_3	2	

After Iteration 1

Merge Sort (Complexity)

- Running time analysis:
 - $T(n)$: worst-case running time of merge sort to sort n numbers (assume n is a power of 2)

Running time analysis can be modeled as an recurrence equation:

$$T(1) = 1, \text{ if } n=1$$

$$T(n) = 2T(n/2) + n, \text{ if } n>1$$

Merge Sort Complexity (Con't)

- Running time analysis (Con't):

$$T(1) = 1 \quad \text{Initial condition}$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n$$

$$= 4T\left(\frac{n}{4}\right) + 2n$$

$$= 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n$$

$$= 8T\left(\frac{n}{8}\right) + 3n$$

⋮

$$= 2^k T\left(\frac{n}{2^k}\right) + kn \quad \text{Since } n = 2^k, \text{ we have } k = \log_2 n$$

$$= nT(1) + n \log_2 n \quad \text{Since } T(1) = 1$$

$$= n + n \log_2 n$$

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

Design Strategy

Divide and Conquer

- is a general algorithm design paradigm:
 - **Divide**: divide the input data S in two or more disjoint subsets S_1, S_2, \dots
 - **Recur**: solve the subproblems recursively
 - **Conquer**: combine the solutions for S_1, S_2, \dots , into a solution for S
- The base case for the recursion are subproblems of constant size
- Analysis can be done using **recurrence equations**

Merge-Sort Review

- Merge-sort on an input sequence S with n elements consists of three steps:
 - **Divide**: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
 - **Recur**: recursively sort S_1 and S_2
 - **Conquer**: merge S_1 and S_2 into a unique sorted sequence

Algorithm $\text{mergeSort}(S, C)$

Input sequence S with n elements, comparator C

Output sequence S sorted according to C

if $S.\text{size}() > 1$

$(S_1, S_2) \leftarrow \text{partition}(S, n/2)$

$\text{mergeSort}(S_1, C)$

$\text{mergeSort}(S_2, C)$

$S \leftarrow \text{merge}(S_1, S_2)$

Master Method (Appendix)

- Many divide-and-conquer recurrence equations have the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:**

- if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
- if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
- if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.



BITS Pilani
Pilani Campus

Course Name : Data Structures & Algorithms

Bharat Deshpande
Computer Science & Information Systems

Lower Bound for Sorting

- **Merge sort**
 - worst-case running time is $O(N \log N)$
- **Insertion Sort**
- Worst-case running time is $O(N^2)$
- **Heap Sort**
 - worst-case running time is $O(N \log N)$
- Are there better algorithms?
- **Goal:** Prove that any sorting algorithm based on only comparisons takes at least $O(N \log N)$ comparisons in the worst case (worse-case input) to sort N elements.

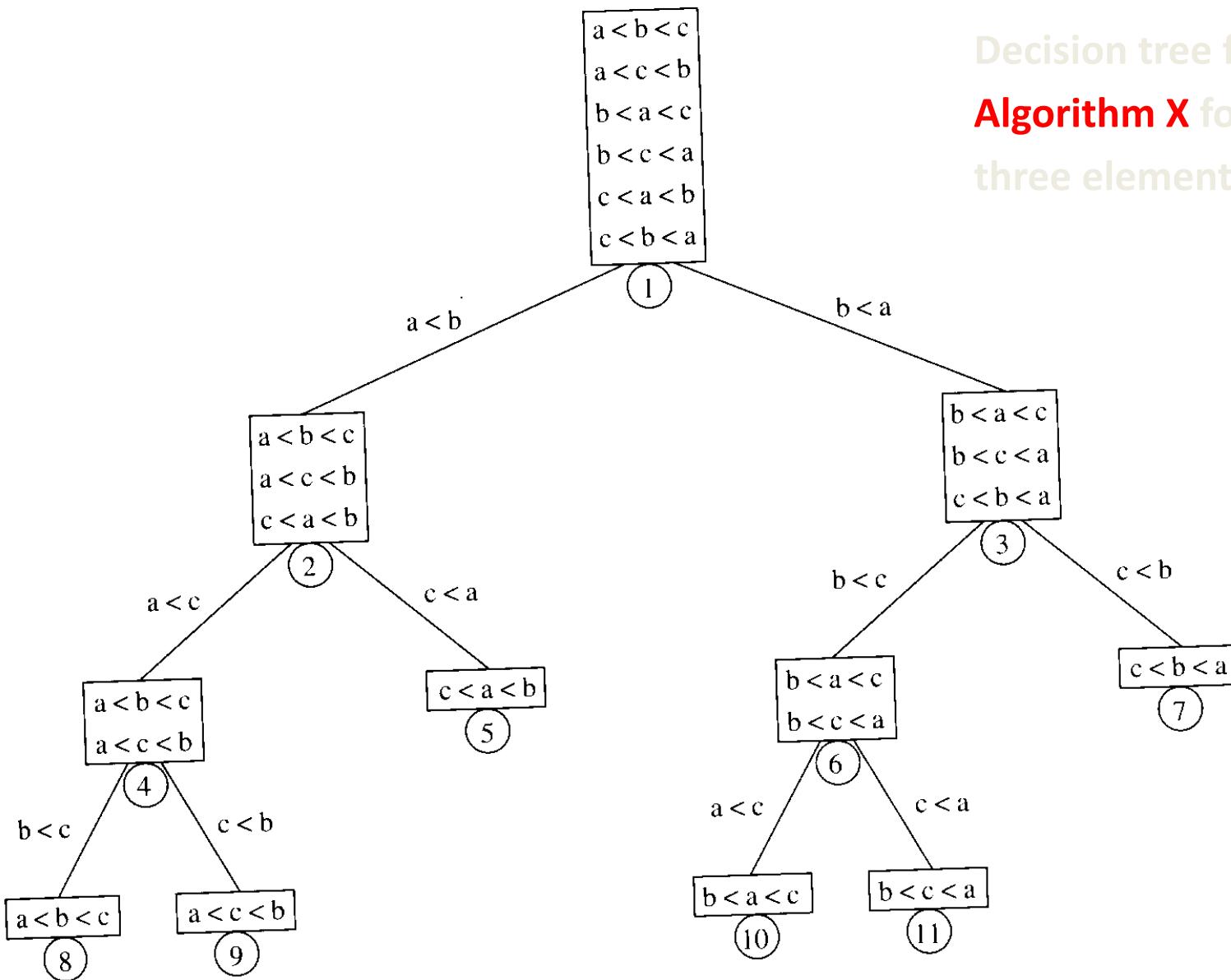
Lower Bound for Sorting (con't..)

- Suppose we want to sort N distinct elements
- How many possible orderings do we have for N elements?
- We can have $N!$ possible orderings (e.g., the sorted output for a,b,c can be $a\ b\ c$, $b\ a\ c$, $a\ c\ b$, $c\ a\ b$, $c\ b\ a$, $b\ c\ a$.)

Lower Bound for Sorting (con't..)

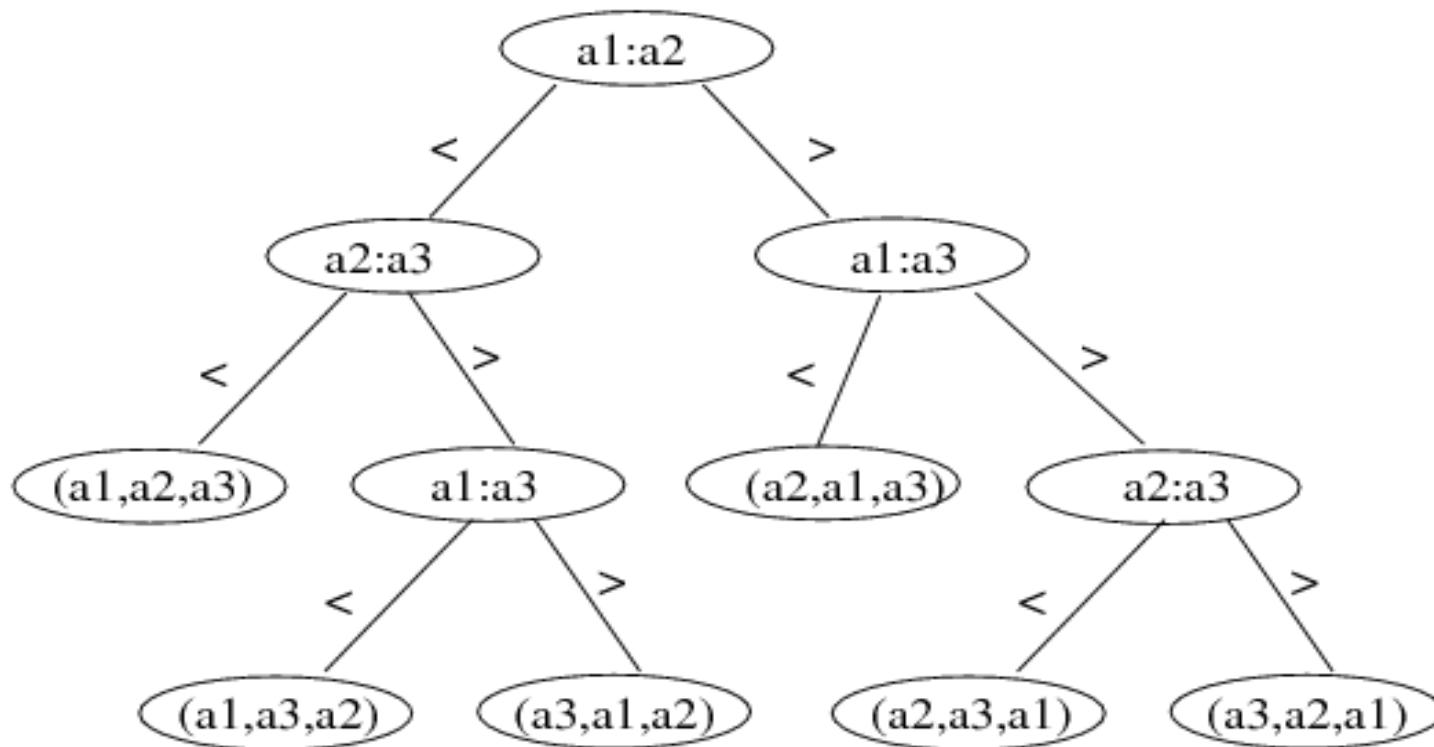
- Any comparison-based sorting process can be represented as a binary **decision tree**.
 - Each node represents a set of possible orderings, consistent with all the comparisons that have been made
 - The tree edges are results of the comparisons

Decision tree for
Algorithm X for sorting
 three elements a, b, c



Lower Bound for Sorting (con't)

- A different algorithm would have a different decision tree
- Decision tree for **Insertion Sort** on 3 elements:



Lower Bound for Sorting (con't)

- The worst-case number of comparisons used by the sorting algorithm is equal to the **depth of the deepest leaf**
 - The average number of comparisons used is equal to the average depth of the leaves
- A decision tree to sort N elements must have **$N!$ leaves**
 - a binary tree of depth d has at most 2^d leaves
⇒ the tree must have depth at least $\lceil \log_2(N!) \rceil$
- Therefore, any sorting algorithm based on only comparisons between elements requires at least $\lceil \log_2(N!) \rceil$ comparisons in the worst case.

Lower Bound for Sorting (con't..)

$$\begin{aligned}\log_2(N!) &= \log(N(N-1)(N-2)\cdots(2)(1)) \\&= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1 \\&\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log(N/2) \\&\geq \frac{N}{2} \log \frac{N}{2} \\&= \frac{N}{2} \log N - \frac{N}{2} \\&= \Omega(N \log N)\end{aligned}$$

- Any sorting algorithm based on comparisons between elements requires $\Omega(N \log N)$ comparisons.
-

Linear time sorting

- Can we do better? (linear time algorithm)

Yes, if the input has special structure

- Counting sort, radix sort, Bucket sort

Bucket Sort

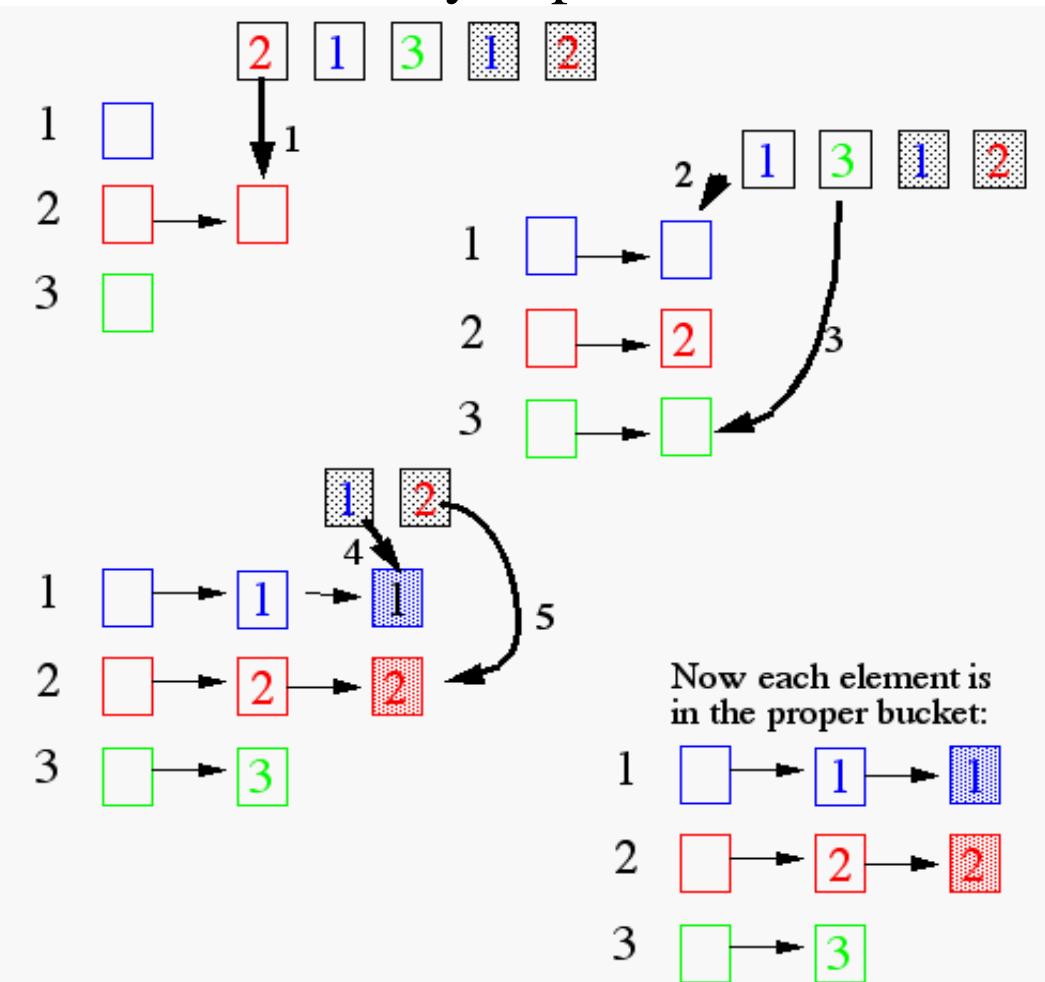
- The bucket sort makes assumptions about the data being sorted
- The n elements to be sorted are integers in the range $[0, N-1]$
- Consequently, we can achieve better than $O(n \ln(n))$ run times

Idea:

- Not based on Comparison
- But using keys as indices into a bucket array.

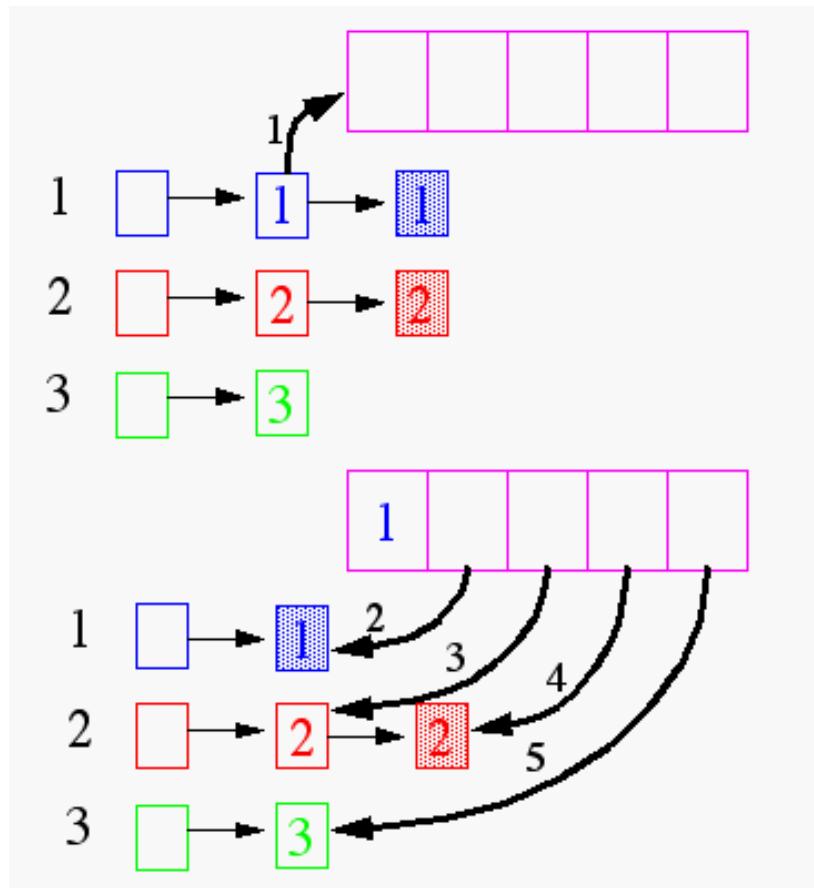
Bucket Sort (Example 1)

Each element of the array is put in one of the N “buckets”



Bucket Sort

Now, pull the elements from the buckets into the array



At last, the sorted array (sorted in a stable way):

1	1	2	2	3
---	---	---	---	---

Bucket Sort Algorithm

bucketSort(s)

Let B be an array of n lists, each of which is initially empty
for each item x in S do

 let k be the key of x

 remove x from S and insert it at the end of bucket B[k]

for i = 0 to n-1 do

 for each item x in list B[i] do

 remove x from B[i] and insert it at the end of S.

Complexity

$O(n + N)$

If N is $O(n)$ then we can sort in $O(n)$ time.



BITS Pilani
Pilani Campus

Course Name : **Data Structures & Algorithms**

Bharat Deshpande
Computer Science & Information Systems

Hash Tables

- **Aim**

To develop a search procedure with running time $O(1)$.

Array is an immediate answer.

Example: Consider a sorted sequence stored in an array.

11	22	28	33	44	49	55	58	66	77	79	88
----	----	----	----	----	----	----	----	----	----	----	----

Focus on finding 44.

Binary search does it in $O(\log n)$ time

Can we retrieve faster than binary search?

Hash Tables

If 44 was stored in A[44], search would be done in one step.

- Problem is if keys are very large.

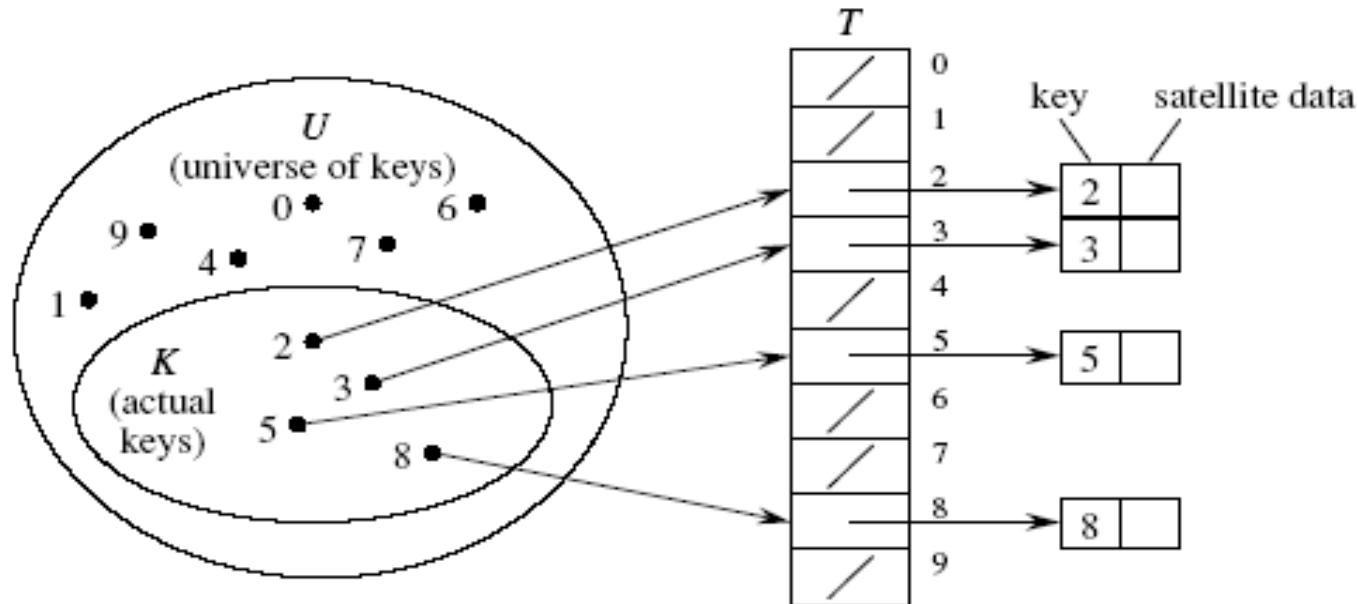
Lot of cells will be empty and so inefficient in terms of space requirement.

- **Answer : Hash Tables**

- Uses an array of size proportional to the number of keys used.
- Instead of using key as an array index, array index is computed from the key.

Array as table

- It is also called **Direct-address Hash Table**.
 - Each **slot**, or position, corresponds to a key in U .
 - If there's an element x with key k , then $T[k]$ contains a pointer to x .
 - Otherwise, $T[k]$ is empty, represented by NIL.



Array as table

- Store the records in a huge array where the index corresponds to the key
 - add - **very fast** $O(1)$
 - delete - **very fast** $O(1)$
 - search - **very fast** $O(1)$
-

Direct Address table

- **Disadvantage** of Direct Address table
 - If the universe U is very large, storing a table of size $|U|$ may be impractical.
 - The set K of keys actually stored may be small relative to U , so that most of the space allocated for table is wasted.

Hash function

- In direct addressing, an element with key k is stored in slot k .
- With hashing, the **hash function** is used to compute the slot $h(k)$

$$h : U \rightarrow \{0, 1, \dots, m-1\}$$

We say that $h(k)$ is the hash value of k .

- **Aim**
To reduce the range of array of indices that needs to be handled

Compression Maps

- **Division Method**

$$h(k) = k \bmod m$$

- Certain values of m may not be good:
- Good values for m are prime numbers which are not close to exact powers of 2. For example, if you want to store 2000 elements then $m=701$ ($m = \text{hash table length}$) yields a hash function: $h(k) = k \bmod 701$

Compression Maps

- **MAD Method (Multiply, Add and Divide)**

Define $h(k) = (ak + b) \text{ mod } n$,

where n is a prime number and a and b are chosen randomly so that $a \text{ mod } n \neq 0$.

- This function is chosen in order to eliminate repeated patterns.

Hash function

- **Problem:**

Two keys may hash to the same slot – **collision**

Ideal situation – avoid collision altogether

- But it is generally **difficult** to design perfect hash. (e.g. when the potential key space is large)

- **Solution :**

Find effective techniques for resolving collision

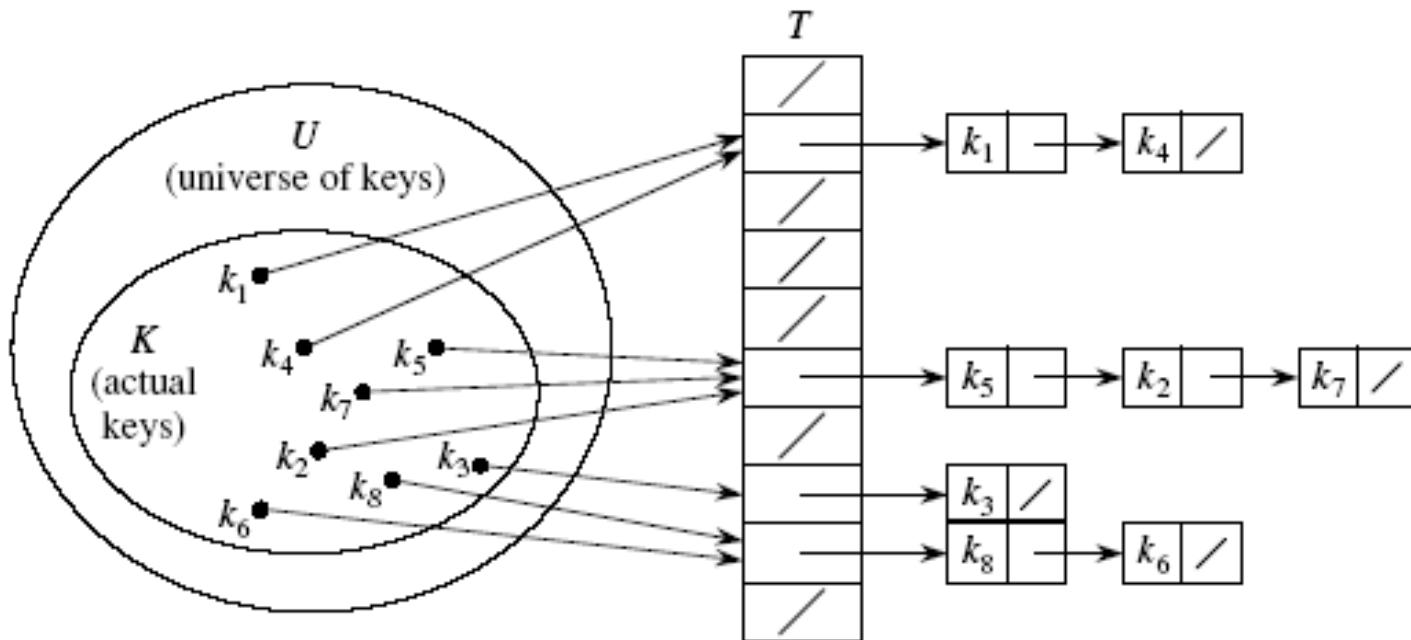
Collision Resolving

- Collision resolution by chaining

In chaining, we put all elements that hash to same slot in a linked list.

- Slot j contains a pointer to the head of the list of all stored elements that hash to j
- If there are no such elements, slot j contains NIL.

Chained Hash Table



Performance Analysis

- **Worst Case**

All keys are hashed to the same slot, worst cast is $O(n)$ time

- **Average Performance**

Given a hash table T with m slots that stores n keys,

Define **load factor α** for T as n/m

- average keys per slot.

Performance Analysis

- **Assume**
 - *Simple uniform hashing.*
 - Any key is equally likely to hash into any of the m slots, independent of where any other key hashes to.
 - $O(1)$ time to compute $h(k)$.
- Time to search for an element with key k depends linearly on the length of the list $T[h(k)]$.
- Expected length of a linked list = load factor = $\alpha = n/m$.
- We consider two cases:
 1. Unsuccessful search
 2. Successful search

Unsuccessful Search

- **Theorem:**
 - An unsuccessful search takes expected time $\Theta(1+\alpha)$.
- Proof:**
- Any key not already in the table is equally likely to hash to any of the m slots.
 - To search unsuccessfully for any key k , need to search to the end of the list $T[h(k)]$, whose expected length is α .
 - Adding the time to compute the hash function, the total time required is $\Theta(1+\alpha)$.

- **Theorem:**
- A successful search takes expected time $\Theta(1+\alpha)$.

Proof:

- The probability that a list is searched is proportional to the number of elements it contains.
- Assume that the element being searched for is equally likely to be any of the n elements in the table.
- The number of elements examined during a successful search for an element x is 1 more than the number of elements examined when the sought for element was inserted.
- The expected length of list to which the i th element is added is $(i-1)/m$

Expected Cost of a Successful Search



Proof contd.

Therefore, the expected no. of elements examined in a successful search is $\left[\frac{1}{n} \sum_{i=1}^n \left(1 + \frac{i-1}{m} \right) \right]$

It can be shown that this is of order $\Theta(1+\alpha)$.

Expected Cost – Interpretation

- If $n = O(m)$, then $\alpha = n/m = O(m)/m = O(1)$.
⇒ Searching takes constant time on average.
- Insertion is $O(1)$ in the worst case.
- Deletion takes $O(1)$ worst-case time when lists are doubly linked.
- Hence, all dictionary operations take $O(1)$ time on average with hash tables with chaining.



BITS Pilani
Pilani Campus

Course Name : **Data Structures & Algorithms**

Bharat Deshpande
Computer Science & Information Systems

Open Addressing

An alternative to chaining for handling collisions.

- **Idea:**
 - Store all keys in the hash table itself.
 - Each slot contains either a key or NIL.
 - To **search** for key k :
 - Examine slot $h(k)$. Examining a slot is known as a **probe**.
 - If slot $h(k)$ contains key k , the search is successful. If the slot contains NIL, the search is unsuccessful.
 - There's a third possibility: **slot $h(k)$ contains a key that is not k** .
 - Compute the index of some other slot, based on k and which probe we are on.
 - Keep probing until we either find key k or we find a slot holding NIL.
- **Advantages:** Avoids pointers; so can use a larger table.

Probe Sequence

- Sequence of slots examined during a key search constitutes a ***probe sequence***.
- Probe sequence must be a permutation of the slot numbers.
 - We examine every slot in the table, if we have to.
 - We don't examine any slot more than once.
- The hash function is extended to:
 - $h : \underbrace{U \times \{0, 1, \dots, m - 1\}}_{\text{probe number}} \rightarrow \underbrace{\{0, 1, \dots, m - 1\}}_{\text{slot number}}$
- $\langle h(k,0), h(k,1), \dots, h(k,m-1) \rangle$ should be a permutation of $\langle 0, 1, \dots, m-1 \rangle$.

Insert Operation

Act as though we were searching, and insert at the first NIL slot found

Hash-Insert(T, k)

1. $i \leftarrow 0$
2. **repeat** $j \leftarrow h(k, i)$
3. **if** $T[j] = \text{NIL}$
4. **then** $T[j] \leftarrow k$
5. **return** j
6. **else** $i \leftarrow i + 1$
7. **until** $i = m$
8. **error** “hash table overflow”

Search Operation

The search algorithm for key k probes the same sequence of slots that the insertion algorithm examined when k was inserted.

Hash-Search (T, k)

1. $i \leftarrow 0$
2. **repeat** $j \leftarrow h(k, i)$
3. **if** $T[j] = k$
4. **then return** j
5. $i \leftarrow i + 1$
6. **until** $T[j] = \text{NIL}$ **or** $i = m$
7. **return** NIL

Deletion Operation

- Cannot just turn the slot containing the key we want to delete to contain NIL.
- Doing so might make it impossible to retrieve any key k during whose insertion we had probed this slot & found it occupied.
- Use a special value **DELETED** instead of NIL when marking a slot as empty during deletion.
 - **Search** should treat DELETED as though the slot holds a key that does not match the one being searched for.
 - **Insert** should treat DELETED as though the slot were empty, so that it can be reused.
- **Disadvantage:** Search time is no longer dependent on α .
 - Hence, chaining is more common when keys have to be deleted.

Probe Sequences

- The ideal situation is ***uniform hashing***:
 - Generalization of simple uniform hashing.
 - Each key is equally likely to have any of them! permutations of $\langle 0, 1, \dots, m-1 \rangle$ as its probe sequence.
- It is **hard to implement** true uniform hashing.
 - **Approximate** with techniques that at least guarantee that the probe sequence is a permutation of $\langle 0, 1, \dots, m-1 \rangle$.
- **Three commonly used techniques:**
 - Linear Probing.
 - Quadratic Probing.
 - Double Hashing.
- All guarantee that $\langle h(k,0), h(k,1), \dots, h(k,m-1) \rangle$ is a permutation of $\langle 0, 1, \dots, m-1 \rangle$.
- None of these fulfills assumptions of uniform hashing.
 - Can't produce all $m!$ probe sequences.

Linear Probing

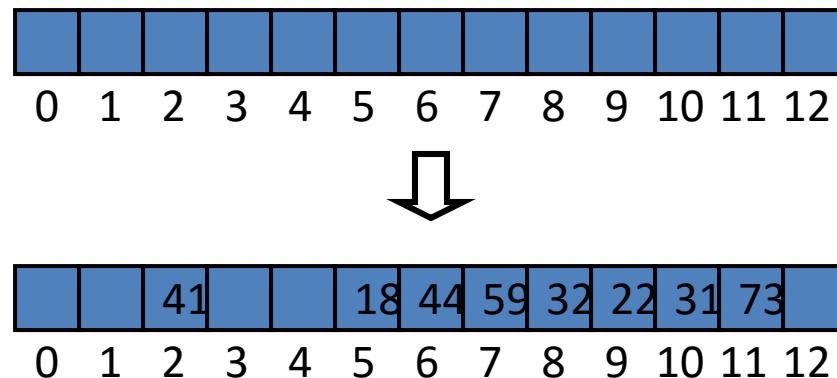
- $$h(k, i) = (h'(k)+i) \bmod m.$$

key Probe number Auxiliary hash function
- The initial probe determines the entire probe sequence.
 - $T[h'(k)], T[h'(k)+1], \dots, T[m-1], T[0], T[1], \dots, T[h'(k)-1]$
 - Hence, **only m distinct probe sequences** are possible.
 - Easy to Implement
- Suffers from ***primary clustering***:
 - Long runs of occupied sequences build up.
 - Long runs tend to get longer, since an empty slot preceded by i full slots gets filled next with probability $(i+1)/m$.
 - Hence, average search and insertion times increase.

Linear Probing : Example

Example:

- $h'(x) = x \bmod 13$
- $h(x) = (h'(x)+i) \bmod 13$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Quadratic Probing

- $$h(k, i) = (h'(k) + c_1i + c_2i^2) \bmod m \quad c_1 \neq c_2$$

key Probe number Auxiliary hash function

- The initial probe position is $T[h'(k)]$, later probe positions are offset by amounts that depend on a quadratic function of the probe number i .
- Must constrain c_1 , c_2 , and m to ensure that we get a full permutation of $\langle 0, 1, \dots, m-1 \rangle$.
- Can suffer from ***secondary clustering***:
 - If two keys have the same initial probe position, then their probe sequences are the same.

Double Hashing

- $$h(k, i) = (h_1(k) + i h_2(k)) \text{ mod } m$$

key Probe number Auxiliary hash functions

- Two auxiliary hash functions.
 - h_1 gives the initial probe. h_2 gives the remaining probes.
- Must have $h_2(k)$ relatively prime to m , so that the probe sequence is a full permutation of $\langle 0, 1, \dots, m-1 \rangle$.
 - Choose m to be a power of 2 and have $h_2(k)$ always return an odd number. Or,
 - Let m be prime, and have $1 < h_2(k) < m$.
- $\Theta(m^2)$ different probe sequences.
 - One for each possible combination of $h_1(k)$ and $h_2(k)$.
 - Close to the ideal uniform hashing.
 - Best method available for open addressing

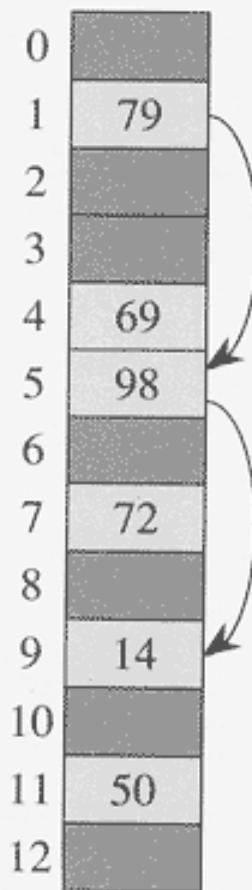


Figure 11.5 Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, the key 14 is inserted into empty slot 9, after slots 1 and 5 are examined and found to be occupied.

Performance of Open Addressing

Theorem:

The expected number of probes in an unsuccessful search in an open-address hash table is at most $1/(1-\alpha)$.

Corollary:

Inserting an element into an open-address table takes $\leq 1/(1-\alpha)$ probes on average.

- If α is a constant, search insertion takes $O(1)$ time.



BITS Pilani
Pilani Campus

Course Name : **Data Structures & Algorithms**

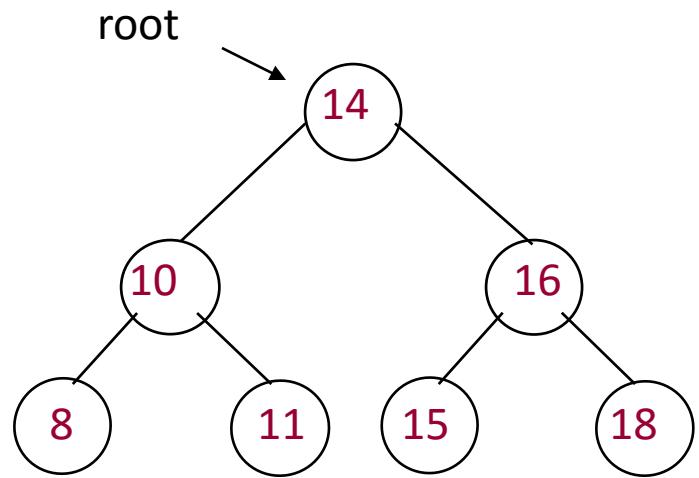
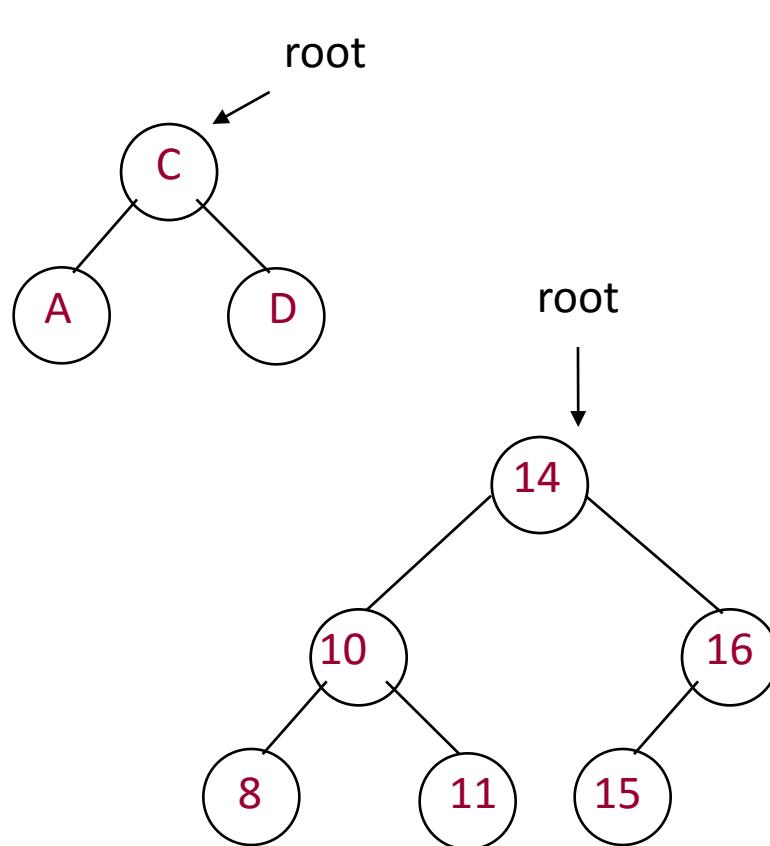
Bharat Deshpande
Computer Science & Information Systems

Binary Search Trees

A **Binary Search Tree (BST)** is a binary tree with the following properties:

- The key of a node is *always greater* than the keys of the nodes in its left subtree
- The key of a node is *always smaller* than the keys of the nodes in its right subtree

Binary Search Trees: Examples

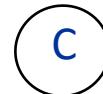


Building a BST

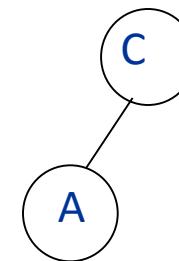
Build a BST from a sequence of nodes read one at a time

Example: Inserting **C A B L M** (in this order!)

1) Insert C

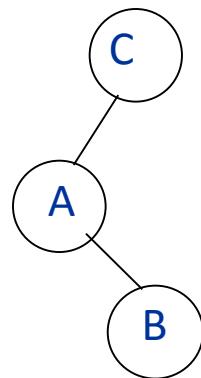


2) Insert A

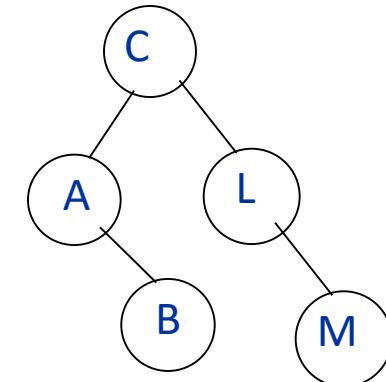


Building a BST

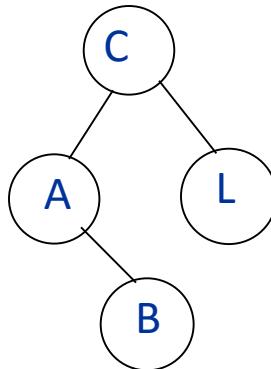
3) Insert B



5) Insert M



4) Insert L

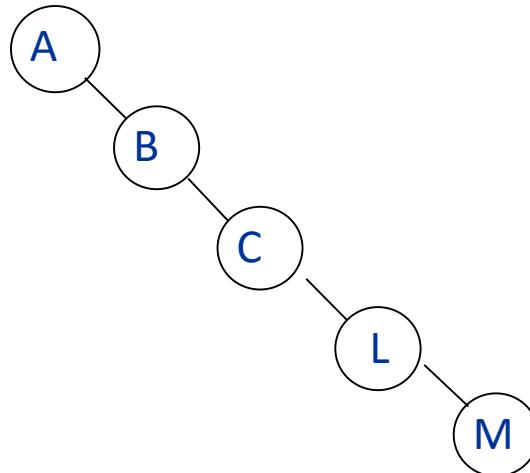


Building a BST

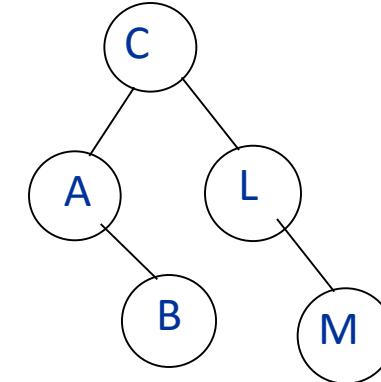
Is there a unique BST for letters A B C L M ?

NO! Different input sequences result in different trees

Inserting: A B C L M

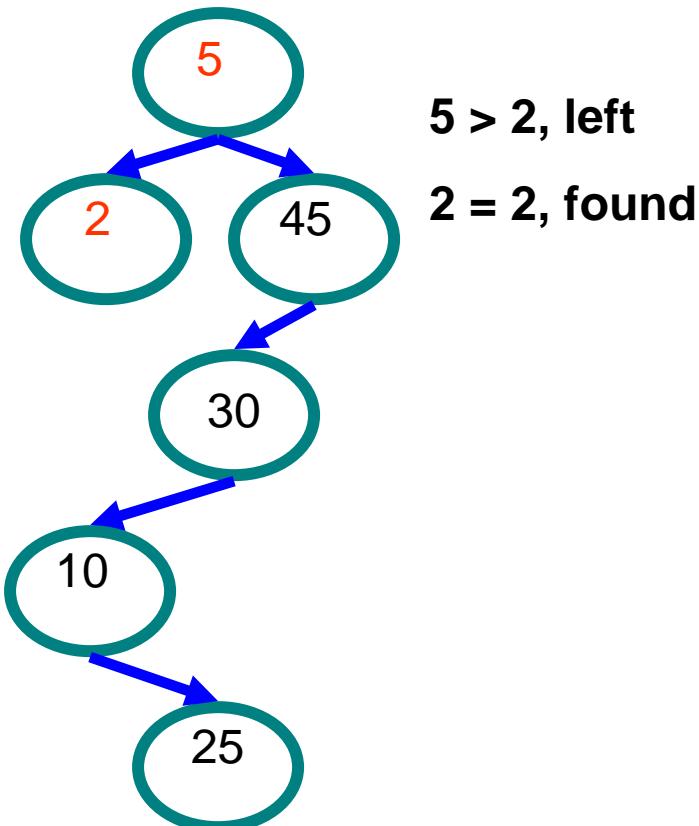
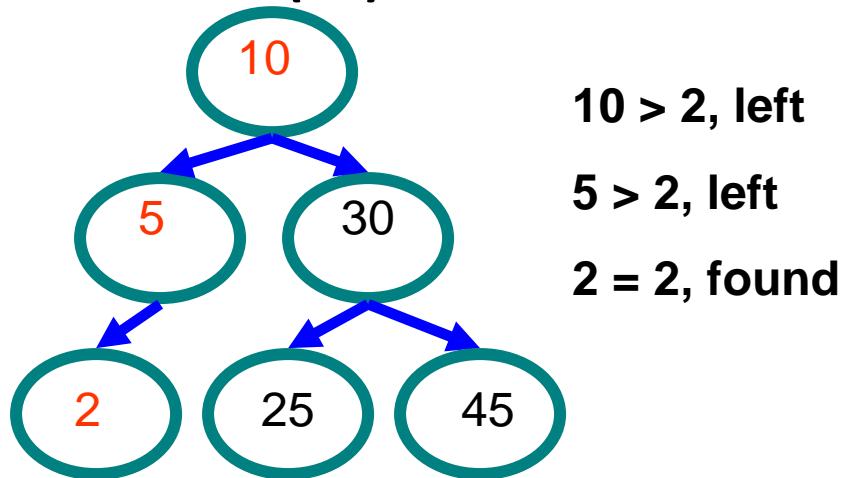


Inserting: C A B L M



Example Binary Searches

- Find (2)



Recursive Search of Binary Tree

```
Node Find( Node n, Value key) {  
    if (n == null)                      // Not found  
        return( n );  
    else if (n.data == key)              // Found it  
        return( n );  
    else if (n.data > key)               // In left subtree  
        return Find( n.left );  
    else                                // In right subtree  
        return Find( n.right );  
}
```

Complexity of Search

- Running time of searching in a BST is proportional to the height of the tree.

If n is the number of nodes in a BST, then

- **Best Case** – $O(\log n)$
- **Worst Case** – $O(n)$

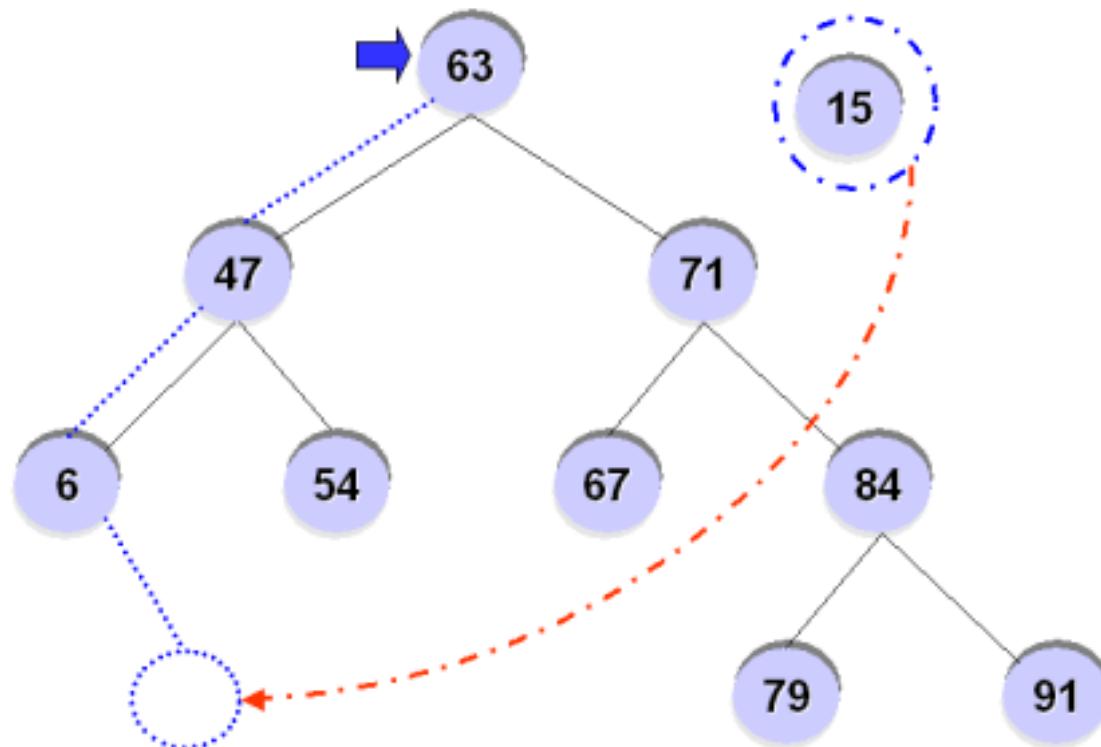
Binary Search Tree - Insertion

Insert Algorithm

- If value we want to insert < key of current node, we have to go to the left subtree
- Otherwise we have to go to the right subtree
- If the current node is empty (not existing) create a node with the value we are inserting and place it here.

Insertion - Example

For example, inserting '15' into the BST?



Binary Search Tree - Deletion

- How do we delete a node from BST?
Similar to the insert function, after deletion of a node, the property of the BST must be maintained.

Binary Search Tree - Deletion

There are 3 possible cases

Case1 : Node to be deleted has no children

→ **We just delete the node.**

Case2 : Node to be deleted has only one child

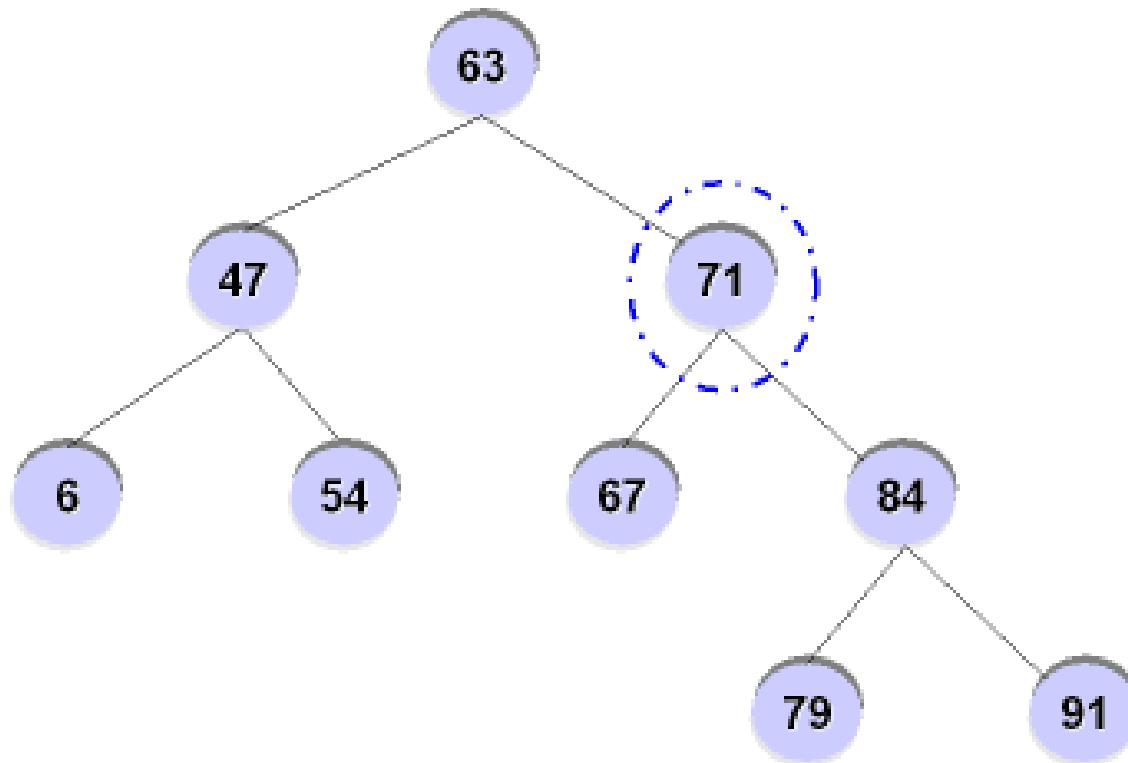
→ **Replace the node with its child
and make the parent of the
deleted node to be a parent of the
child of the deleted node**

Case3 : Node to be deleted has two children

Binary Search Tree - Deletion



Node to be deleted has two children



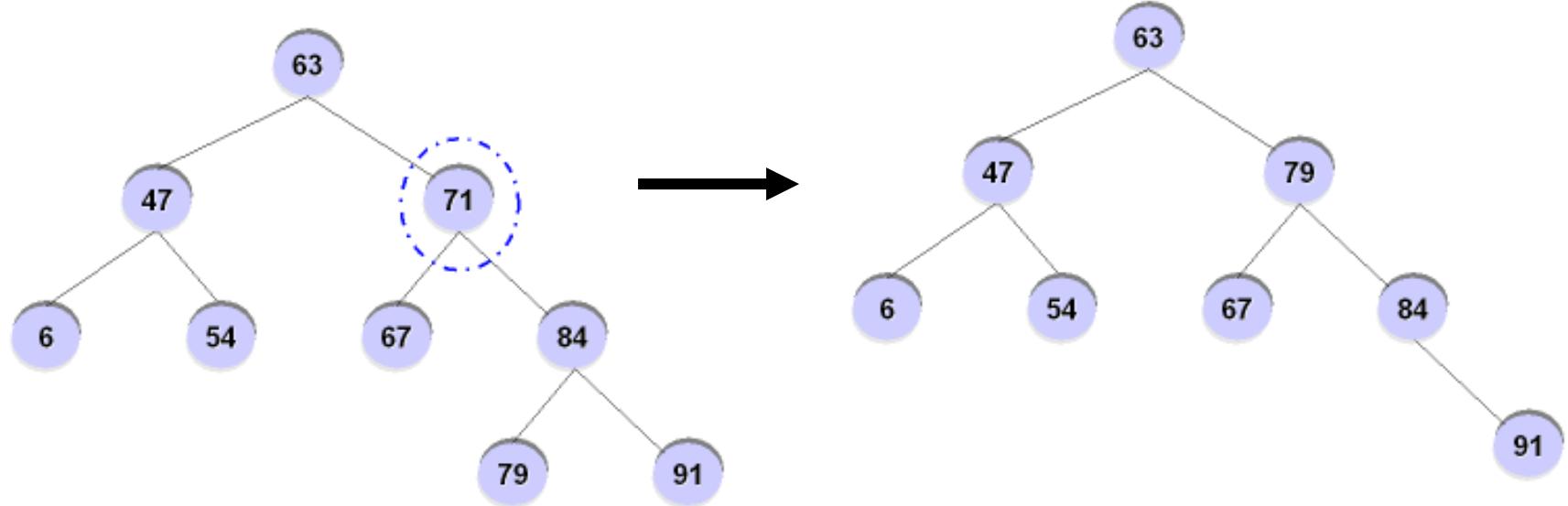
Binary Search Tree - Deletion

Node to be deleted has two children

Steps:

- Find minimum value of right subtree
- Delete minimum node of right subtree but keep its value
- Replace the value of the node to be deleted by the minimum value whose node was deleted earlier.

Binary Search Tree - Deletion





BITS Pilani
Pilani Campus

Course Name : Data Structures & Algorithms

Bharat Deshpande
Computer Science & Information Systems

Binary Search Trees - Disadvantages

- Unbalanced Binary Search Trees are bad
- Worst case Performance is $O(n)$
- No better than sequential searching.

- We look at correcting this problem.

Height Balance Property

- **Height Balance Property**

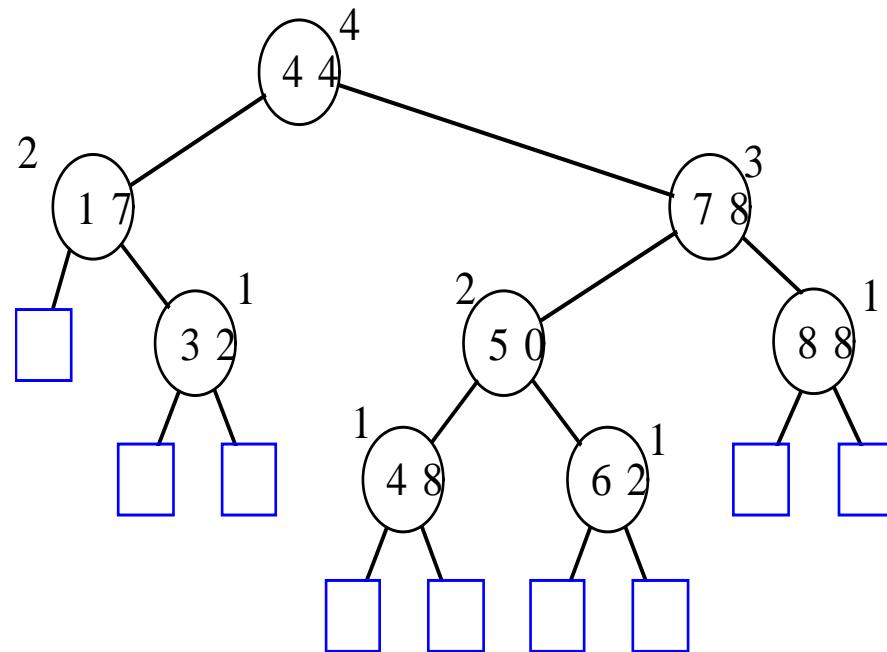
For every internal node v of a binary search tree, the *heights of the children of v can differ by at most 1.*

- It is a property which characterizes the structure of a BST

Definition: Any Binary Search Tree T that satisfies height balance property is said to be an **AVL tree.** (Adelson-Velskii & Landis)

AVL Tree

Example



An example of an AVL tree where the heights are shown next to the nodes:

AVL tree

- Simple Observation

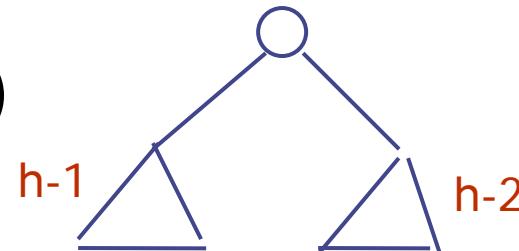
Subtree of an AVL tree is itself an AVL tree.

- Important Consequence

Keeps the height small

Height of an AVL Tree

- **Fact:** The *height* of an AVL tree storing n keys is $O(\log n)$.
- **Proof:** Let us bound $n(h)$: the minimum number of internal nodes of an AVL tree of height h .
- We easily see that $n(1) = 1$ and $n(2) = 2$
- For $h > 2$, AVL tree with minimum no. of nodes is such that both its subtrees are AVL trees with minimum no. of nodes
- Such an AVL tree contains the root node, one AVL subtree of height $n-1$ and another of height $n-2$.
- That is, $n(h) = 1 + n(h-1) + n(h-2)$



Height of an AVL Tree

- Knowing $n(h-1) > n(h-2)$, we get $n(h) > 2n(h-2)$.
- So $n(h) > 2n(h-2)$, $n(h) > 4n(h-4)$, $n(h) > 8n(h-6)$, ...

by induction,

$$n(h) > 2^i n(h-2i)$$

Choosing i such that $h - 2i$ is either 1 or 2

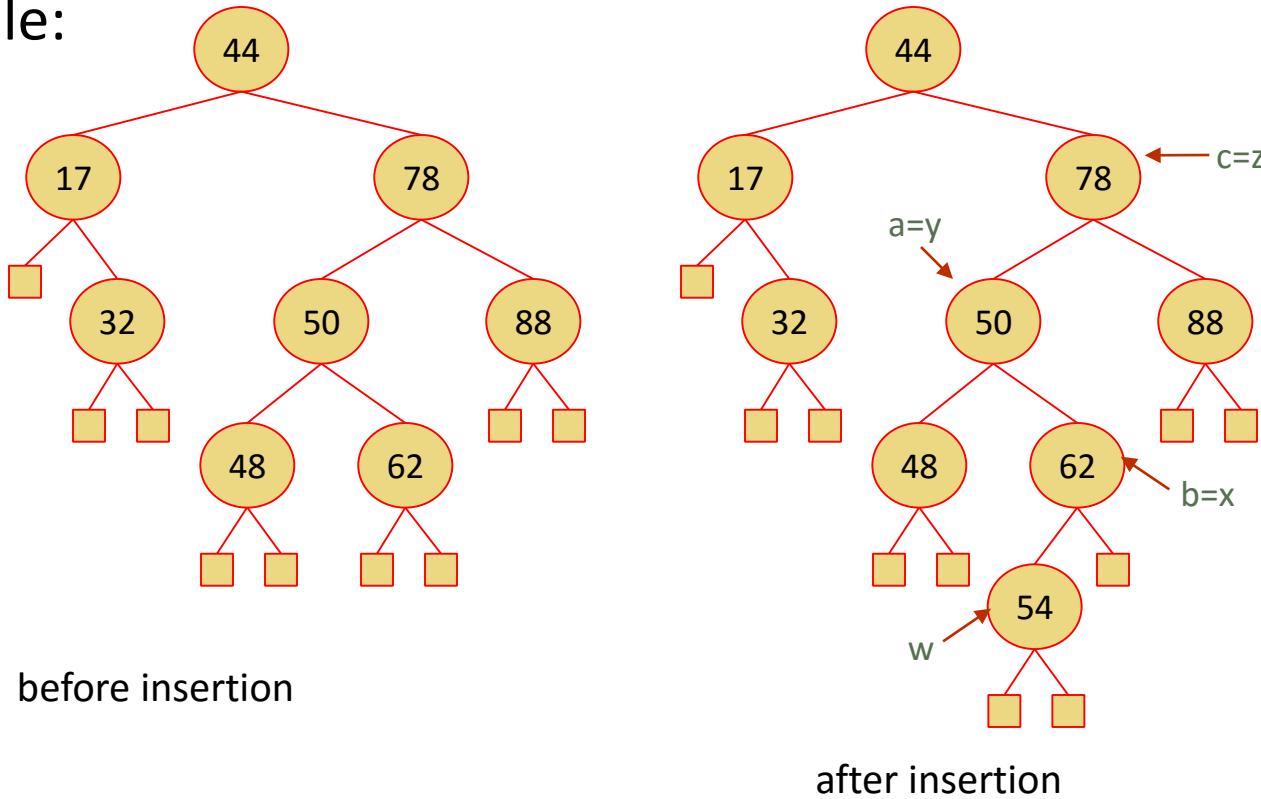
We get: $n(h) > 2^{h/2-1}$

Taking logarithms: $h < 2\log n(h) + 2$

Thus the height of an AVL tree is $O(\log n)$

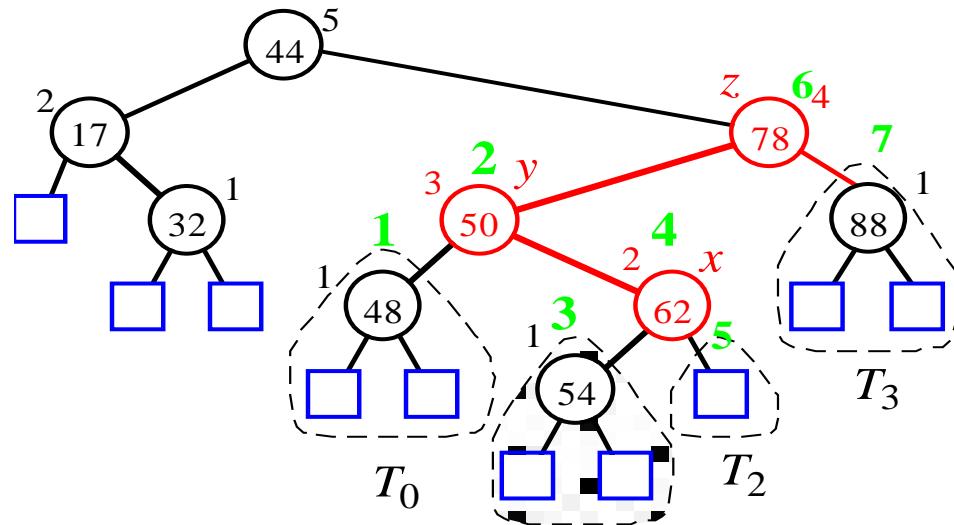
Insertion in an AVL Tree

- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Example:



It is no longer balanced

- Identifying the node at which height balance property is violated.



Search & Repair strategy

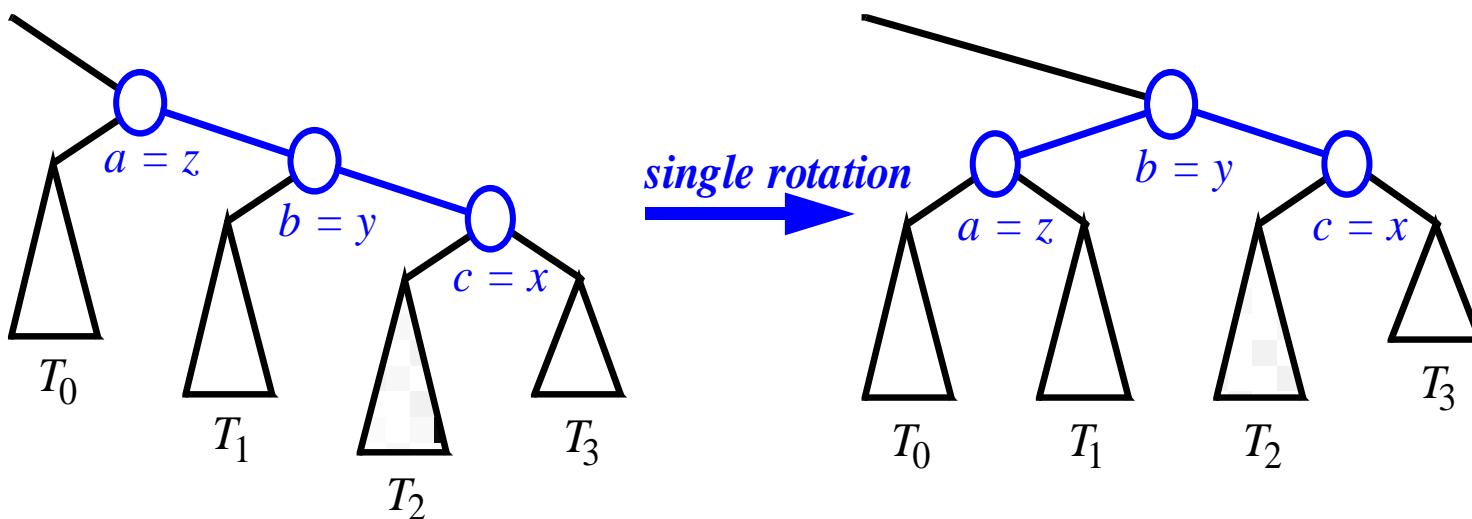
- Let z be the 1st node encountered in going up from the inserted node towards the root at which the height balance property is violated.
- Let y be the child of z with higher height
- Let x be the child of y with higher height
- **Trinode Structuring**
Balance the subtree rooted at z.

-
- Relabel x, y, z as a, b, c such that a precedes b and b precedes c in an inorder traversal of T.
 - There are four possible cases to do this mapping.
 - **Restructuring (Unified way)**

Replace z with the node called b, make the children of this node be a and c while maintaining the inorder relationships of all nodes in T.

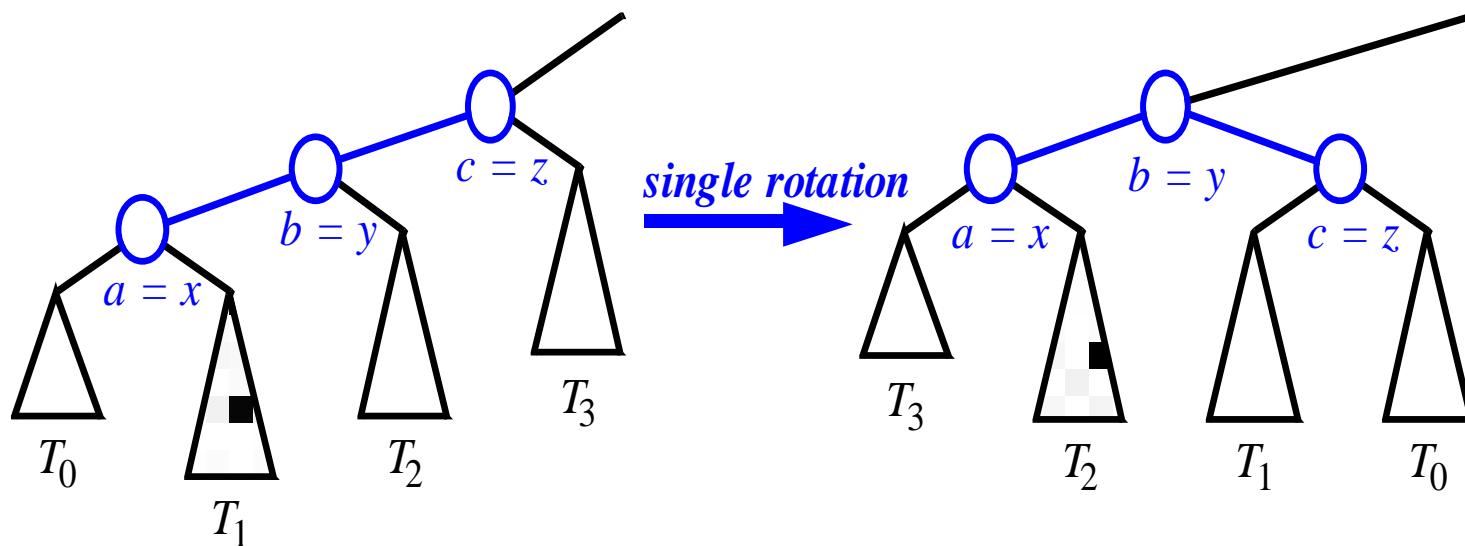
Restructuring (Single Rotation)

- Single Rotations:



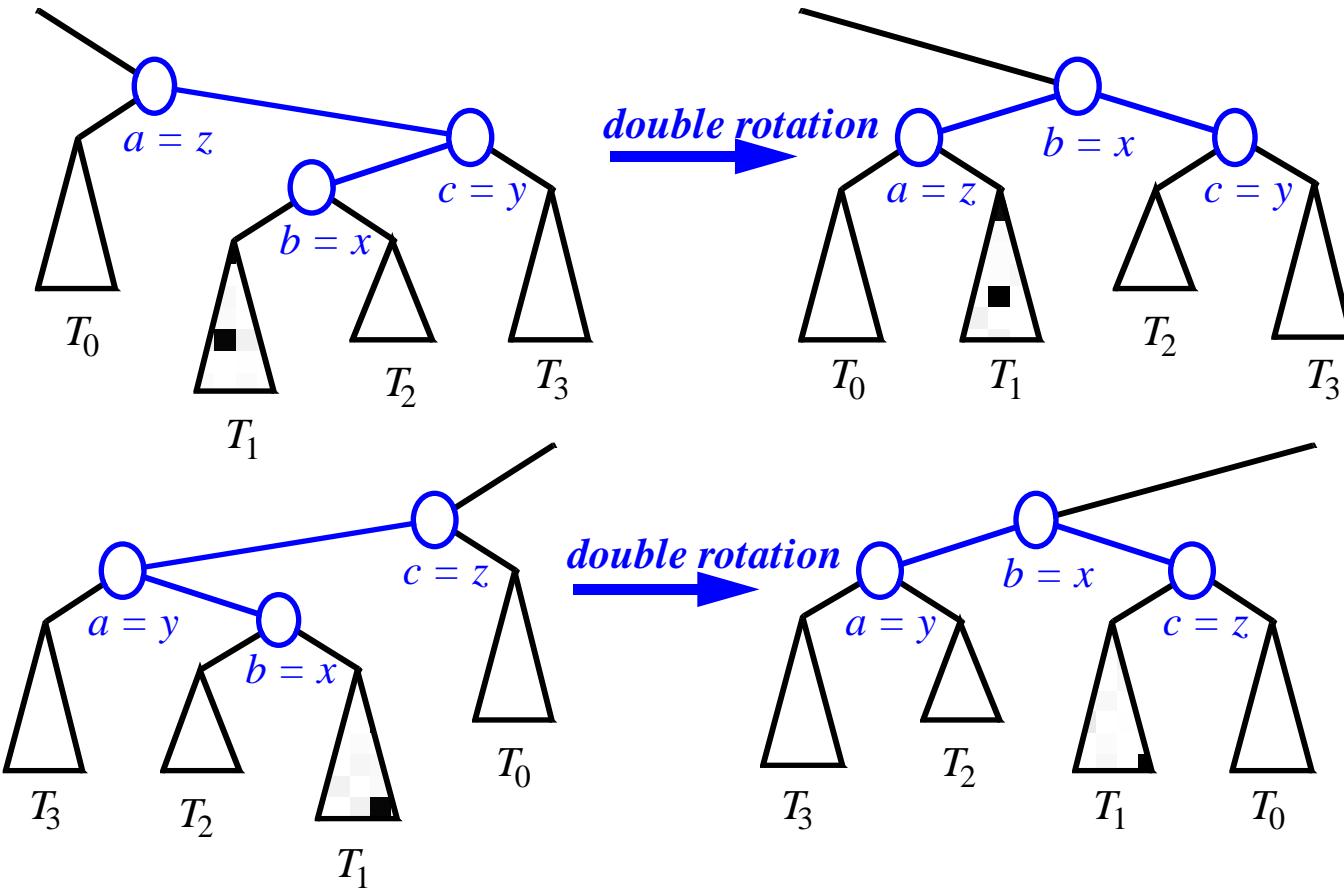
Restructuring (Single Rotation)

- Single Rotations:

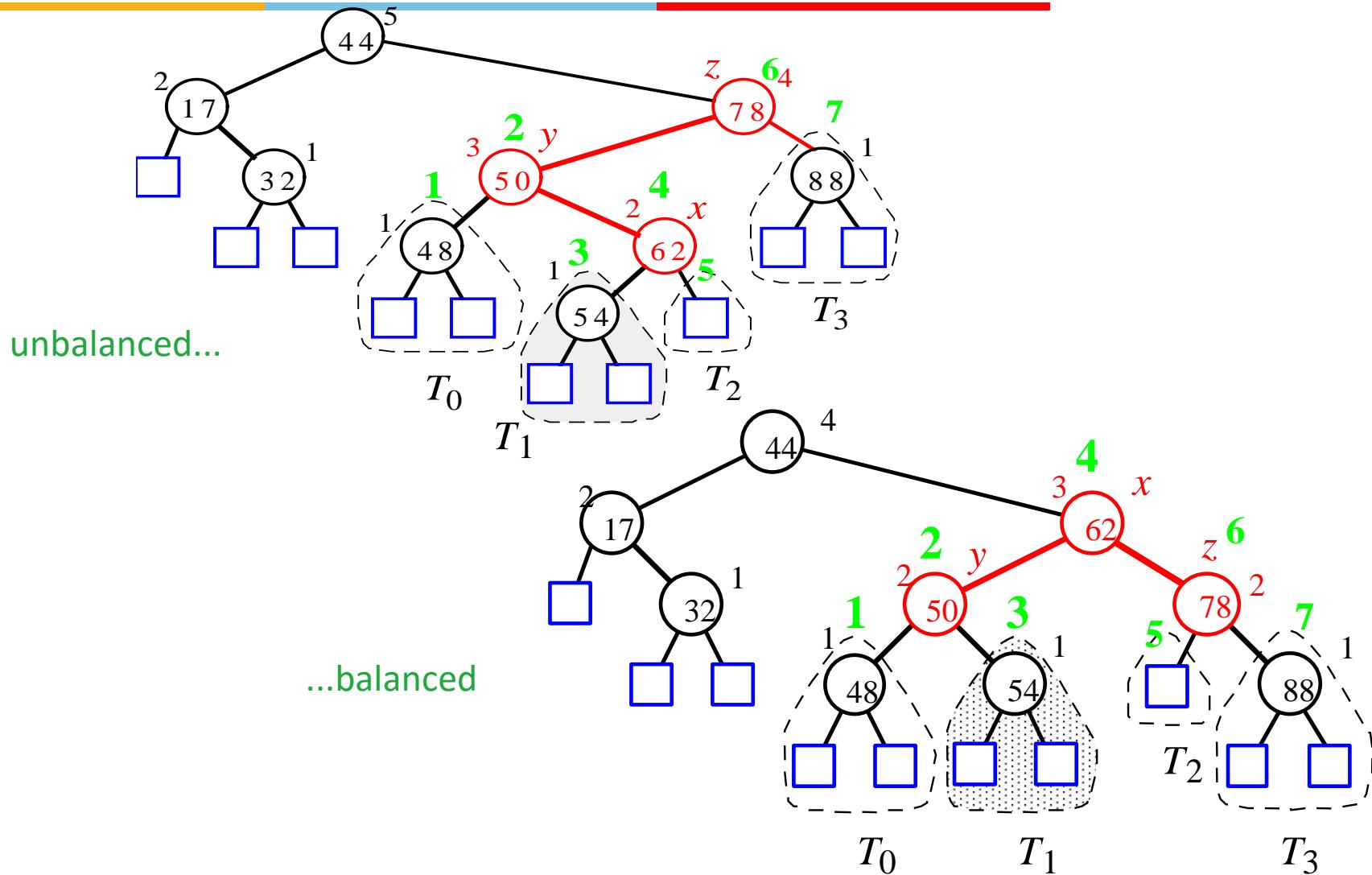


Restructuring (Double Rotation)

- double rotations:



Insertion Example, continued





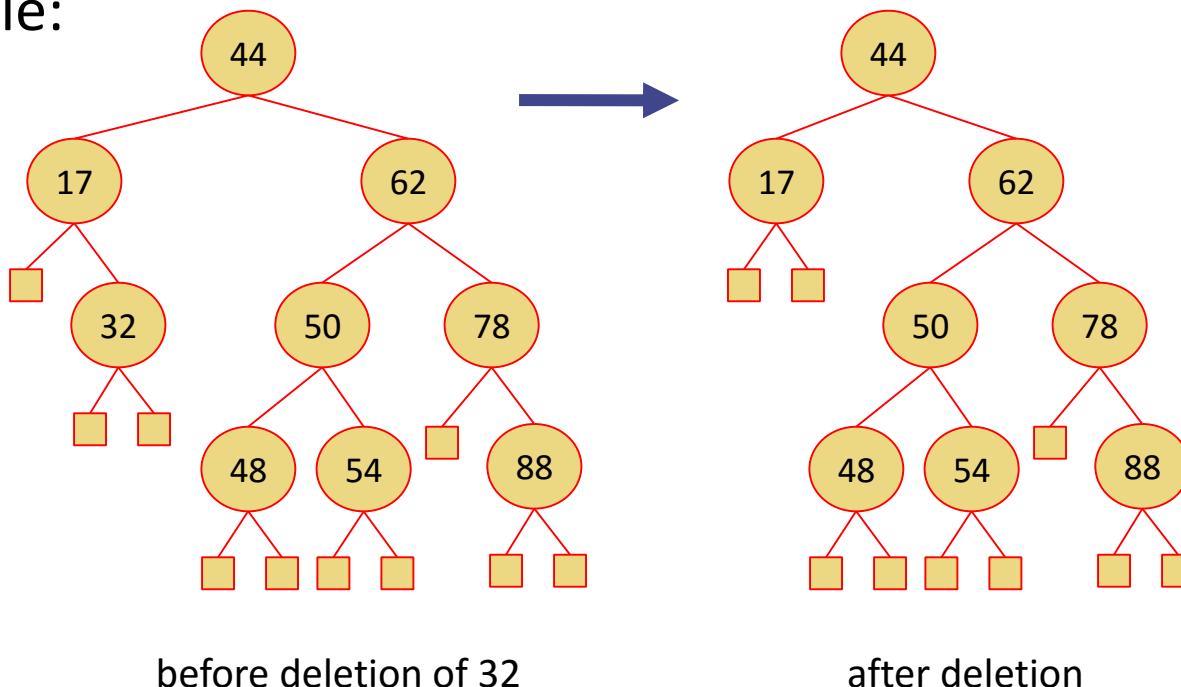
BITS Pilani
Pilani Campus

Course Name : **Data Structures & Algorithms**

Bharat Deshpande
Computer Science & Information Systems

Deletion in an AVL Tree

- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.
- Example:

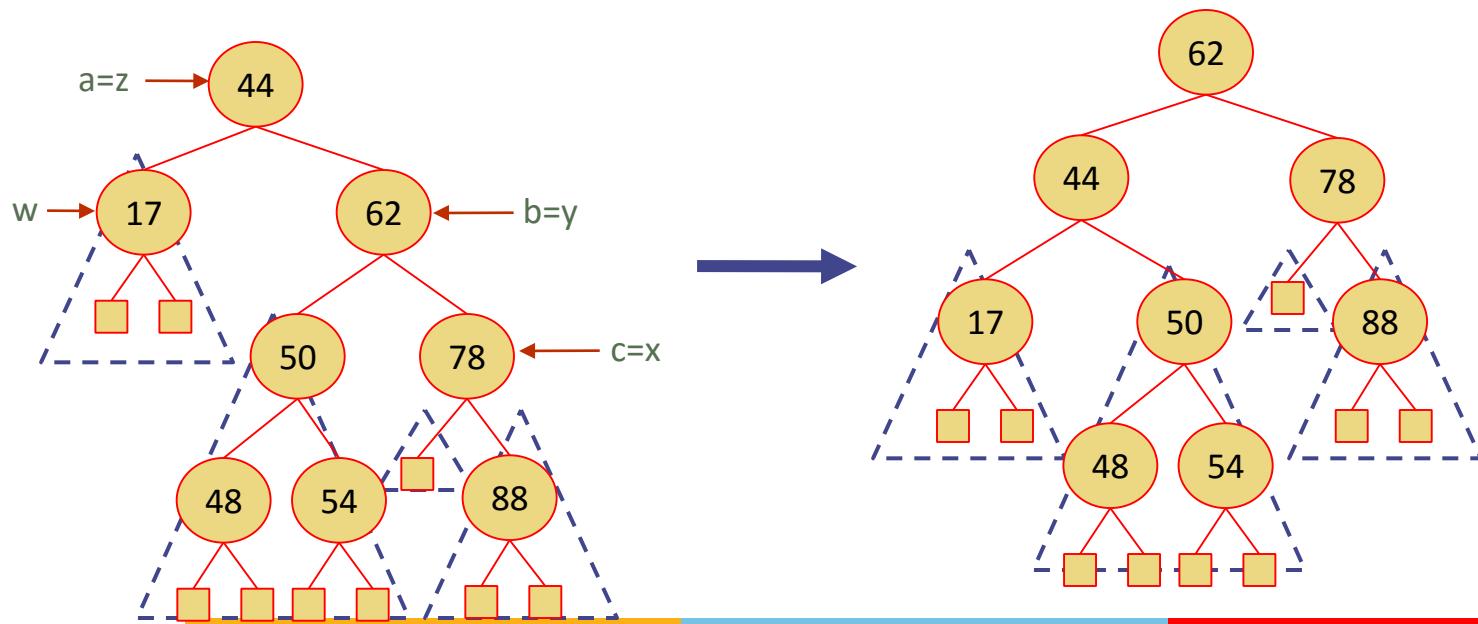


Deletion in an AVL Tree

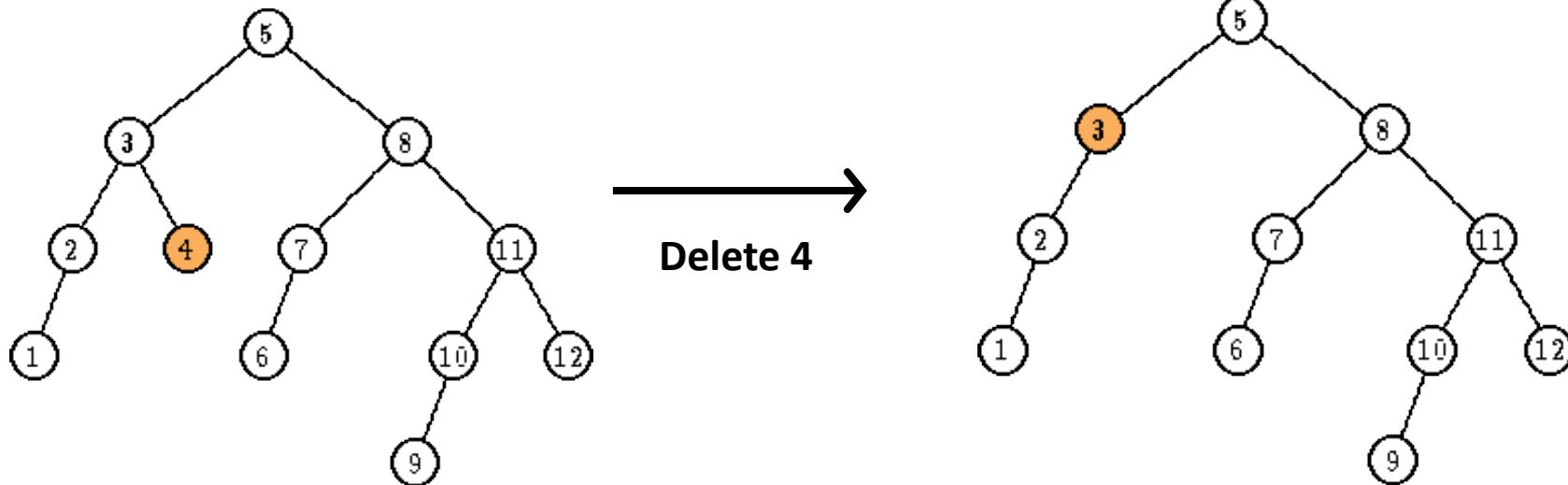
- Let z be the **first unbalanced** node encountered while travelling up the tree from w . Also,
- let y be the child of z with the larger height,
- let x be the child of y defined as follows;
 - If one of the children of y is taller than the other, choose x as the taller child of y .
 - If both children of y have the same height, select x be the child of y on the same side as y (i.e., if y is the left child of z , then x is the left child of y ; and if y is the right child of z then x is the right child of y .)

Trinode Restructuring

- We perform **restructure(x)** to restore balance at z.
- As this restructuring may upset the balance of another node higher in the tree, **we must continue checking for balance until the root of T is reached**

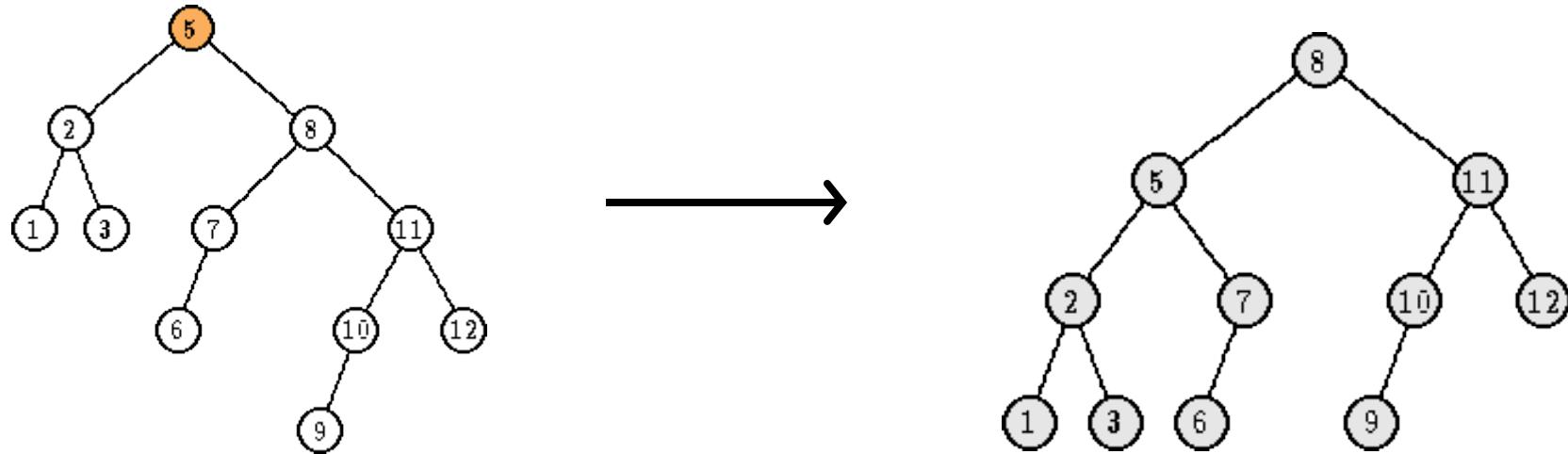


Deletion - Example



Imbalance at 3
Perform rotation with 2

Example – Contd.



Imbalance at 5

Perform rotation with 8

Running Times for AVL Trees



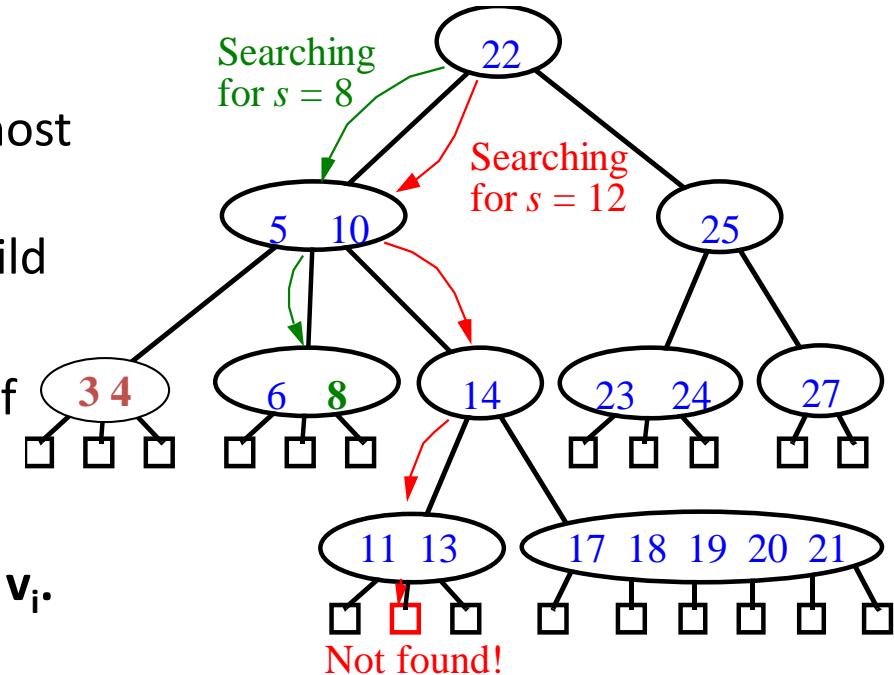
- a single restructure is $O(1)$
 - using a linked-structure binary tree
- find is $O(\log n)$
 - height of tree is $O(\log n)$, no restructures needed
- insert is $O(\log n)$
 - initial find is $O(\log n)$
 - Restructuring up the tree, maintaining heights is $O(\log n)$
- remove is $O(\log n)$
 - initial find is $O(\log n)$
 - Restructuring up the tree, maintaining heights is $O(\log n)$

Multi-Way Search Trees

- Each internal node of a multi-way search tree T:
 - has at least two children, i.e., each internal node is a d-node, $d \geq 2$
 - stores a collection of items of the form (k, x) , where k is a key and x is an element
 - Each d – node with children v_1, \dots, v_d contains $d - 1$ items, stored in increasing order
 - “contains” 2 pseudo-items: $k_0 = -\infty$, $k_d = \infty$. For each item (k, x) stored at a node in the subtree of v rooted at v_i , $i = 1, \dots, D$, we have $k_{i-1} \leq k \leq k_i$
- Children of each internal node are “between” items
- all keys in the subtree rooted at the child fall between keys of those items

Multi-way Searching

- Similar to binary searching
 - If search key $s < k_1$ search the leftmost child
 - If $s > k_{d-1}$, search the rightmost child
- That's it in a binary tree; what about if $d > 2$?
 - Find two keys k_{i-1} and k_i between which s falls, and search the child v_i .



Performance

- **Worst Case – $O(h \log d)$**

where d is the maximum number of children of any node

- If d is constant, then running time is $O(h)$
- **Prime Goal**

Try to keep h as small as possible

(2, 4) or (2, 3, 4) Trees

(2, 4) tree is a multi-way search tree with the following two properties:

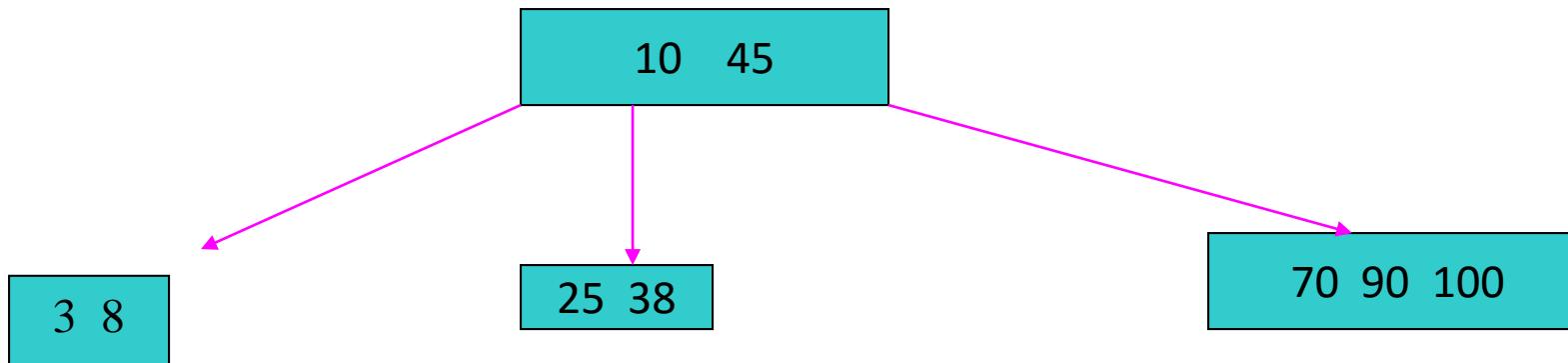
Size property:

Nodes may contain 1, 2 or 3 items & a node with k items has $k + 1$ children

Depth property:

All the external nodes have same depth

Example



Height of (2, 4) tree

Result:

A multi-way search tree storing n items has $n + 1$ external nodes.

- Can be proved by induction

Main Result :

Height of a (2, 4) tree storing n items is $O(\log n)$

Proof

Let h be the height of $(2, 4)$ tree T storing n items.

Then the number of external nodes in T is at most 4^h

& the number of external nodes in T is at least 2^h

Therefore,

$$2^h \leq n + 1 \leq 4^h$$

Taking logarithm in base 2, we get that

$h \leq \log(n+1)$ and $\log(n+1) \leq 2h$, which proves the result.



BITS Pilani
Pilani Campus

Course Name : Data Structures & Algorithms

Bharat Deshpande
Computer Science & Information Systems



Insertion in (2, 4) tree

- To insert an item with key k in a (2, 4) tree T
- Assume that tree T has no element with key k.
- Perform search for k.
This search will terminate at an external node, say z.
- Let v be parent of z
- Insert new into node v and add a new child w(an external node) to v on the left of z.

Observe: Insertion method preserves the depth property.

But it may violate size property.

- If v was previously a 4-node, after insertion will become a 5-node, which is not allowed.
- This violation of size property is called **overflow** at v.
- Overflow needs to be resolved

Insertion in (2, 4) tree

Resolving Overflow

Perform **Split Operation**

- Let v_1, v_2, v_3, v_4, v_5 be children of v .
- Let k_1, k_2, k_3, k_4 be the keys stored at v .
- Replace v with two nodes v' and v'' , where
- v' is a 3-node with children v_1, v_2, v_3 storing keys k_1, k_2
- v'' is a 2-node with children v_4, v_5 storing key k_4
- If v is a root of T , create a new root u or else u be parent of v .

Insertion in (2, 4) tree

- Insert key k into u and make v' & v'' children of u , so that if v was i th child of u , then v' & v'' become i th & $(i+1)$ th child of u .
- One split operation takes $O(1)$ time.
- As a consequence of split operation on a node v , a new overflow may occur at the parent u of v .
- In worst case this propagates all the way up to the root, where it is finally resolved.

Deletion in (2, 4) tree

To remove an item with key k from a (2, 4) tree T.

- Perform search in T for an item with key k.

Key Point:

Removing item can always be reduced to the case where the item to be removed is stored at a node v whose children are external nodes.

(similar to BST)

- Suppose, item k is stored in the ith item at a node z that has only internal node children.
- Find the rightmost internal node v in the subtree rooted at ith child of z, such that children of v are all external nodes.
- Swap the item to be deleted from z with the last item of v.

Deletion in (2, 4) tree

- Remove the item k from v and remove the i th external node of v .
- Removal preserves the depth property.
- But size property may get violated
 - A 2-node may become a 1-node, which is not allowed.
- This is called **underflow**.
- Underflow needs to be resolved.

Deletion in (2, 4) tree

Resolving Underflow

- Check whether immediate sibling of v is a 3-node or a 4-node.
- If such sibling exists, say w, perform a **transfer operation**, in which we move a child of w to v, a key of w to the parent u of v and a key of u to v.

Deletion in (2, 4) tree

- If v has only one sibling or if both the immediate siblings of v are 2-nodes, perform a **fusion operation**, in which we merge v with a sibling, creating a new node v' and move a key from parent u of v to v'
- One transfer/fusion operation takes $O(1)$ time.
- Fusion operation may cause a new underflow to occur at the parent u of v .
- In worst case this propagates all the way up to the root, where it is finally resolved.



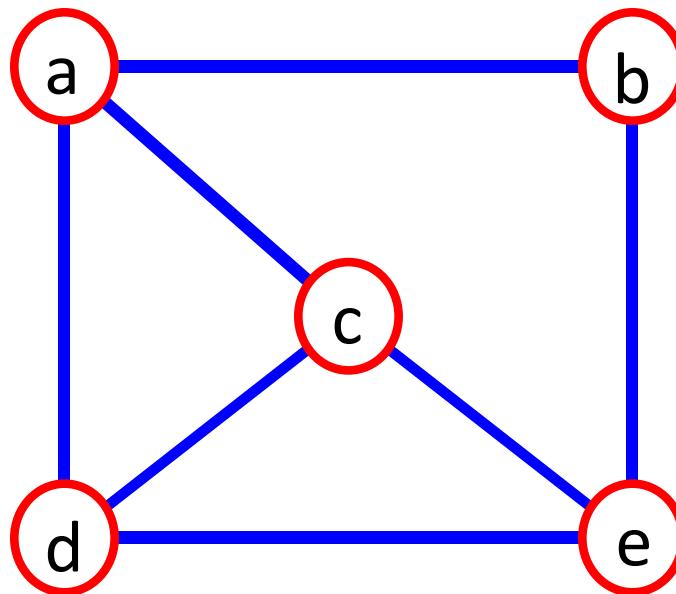
BITS Pilani
Pilani Campus

Course Name : **Data Structures & Algorithms**

Bharat Deshpande
Computer Science & Information Systems

What is a Graph?

- A graph $G = (V, E)$ is composed of:
 - V: set of **vertices**
 - E: set of **edges** connecting the **vertices** in V
- An **edge** $e = (u, v)$ is a pair of **vertices**
- **Example:**

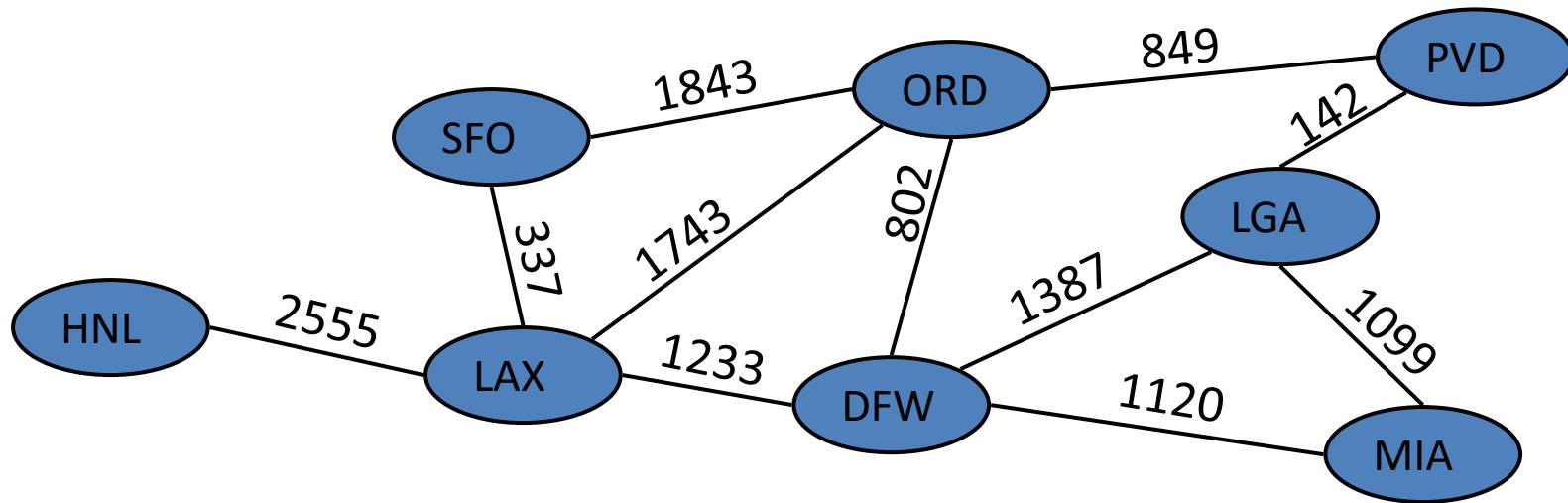


$$V = \{a, b, c, d, e\}$$

$$E = \{(a,b), (a,c), (a,d), (b,e), (c,d), (c,e), (d,e)\}$$

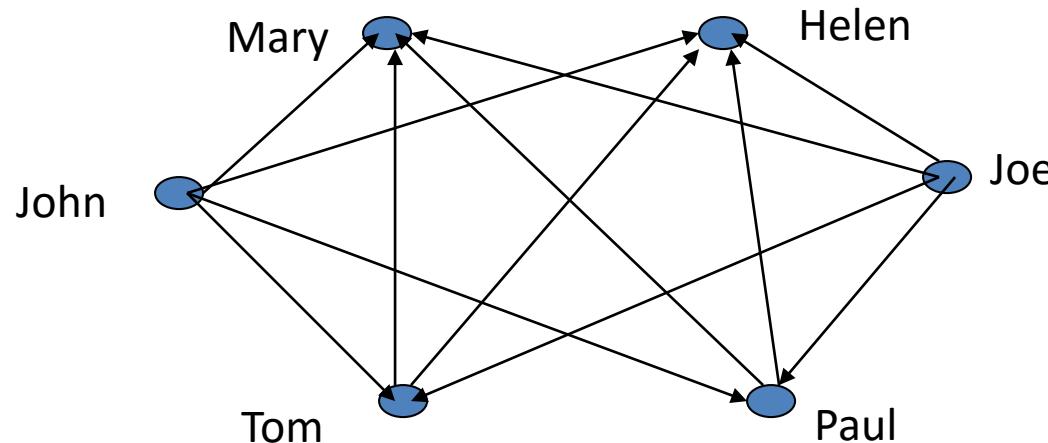
Graph-Example

- Example:
 - A vertex represents an airport and stores the three-letter airport code
 - An edge represents a flight route between two airports and stores the mileage of the route



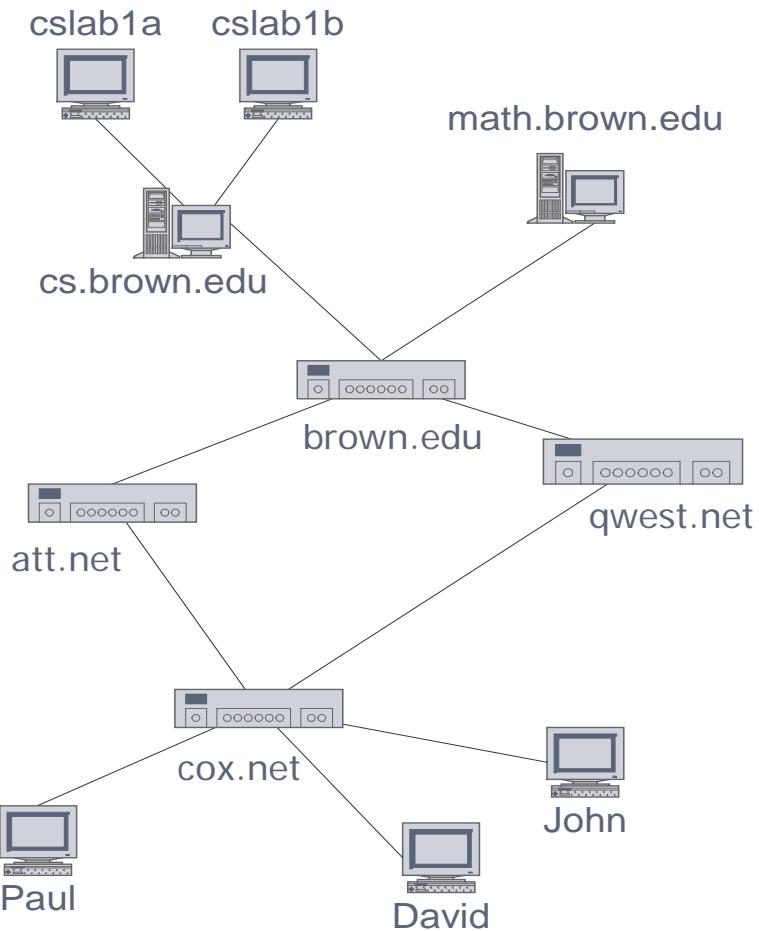
A “Real-life” Example of a Graph

- $V = \text{set of 6 people: John, Mary, Joe, Helen, Tom, and Paul, of ages 12, 15, 12, 15, 13, and 13, respectively.}$
- $E = \{(x,y) \mid \text{if } x \text{ is younger than } y\}$



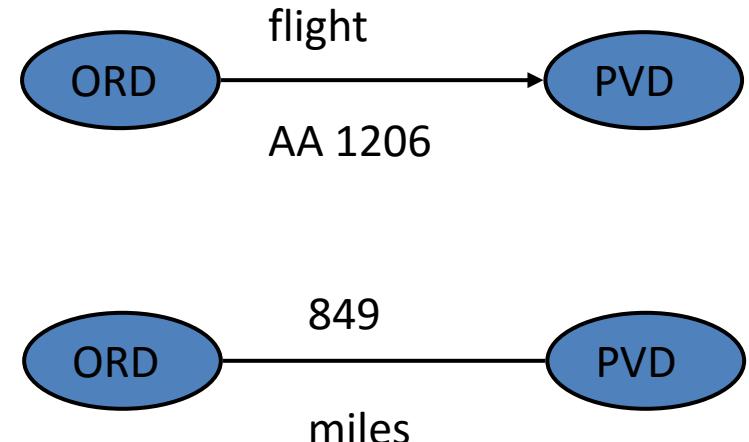
Applications

- Electronic circuits
 - Printed circuit board
 - Integrated circuit
- Transportation networks
 - Highway network
 - Flight network
- Computer networks
 - Local area network
 - Internet
 - Web
- Databases
 - Entity-relationship diagram



Edge Types

- **Directed edge**
 - ordered pair of vertices (u, v)
 - first vertex u is the origin
 - second vertex v is the destination
 - e.g., a flight
- **Undirected edge**
 - unordered pair of vertices (u, v)
 - e.g., a flight route
- **Directed graph (Digraph)**
 - all the edges are directed
 - e.g., flight network
- **Undirected graph**
 - all the edges are undirected
 - e.g., route network



Terminology

- If (v_0, v_1) is an edge in an undirected graph,
 - v_0 and v_1 are **adjacent**
 - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If $\langle v_0, v_1 \rangle$ is an edge in a directed graph
 - v_0 is **adjacent to** v_1 , and v_1 is **adjacent from** v_0
 - The edge $\langle v_0, v_1 \rangle$ is incident on v_0 and v_1

Terminology:



- *Degree of a Vertex*

The **degree** of a vertex is the number of edges incident to that vertex

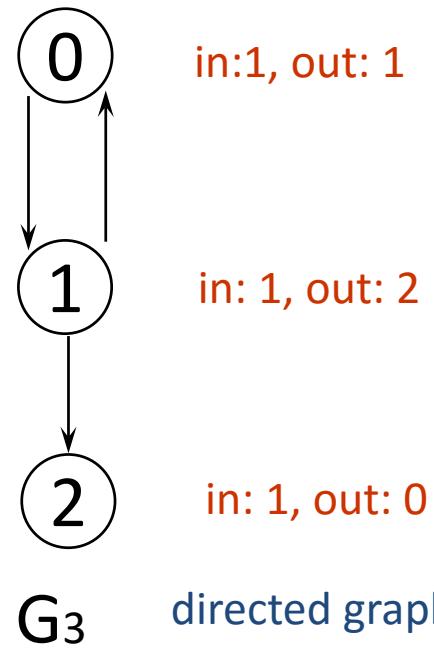
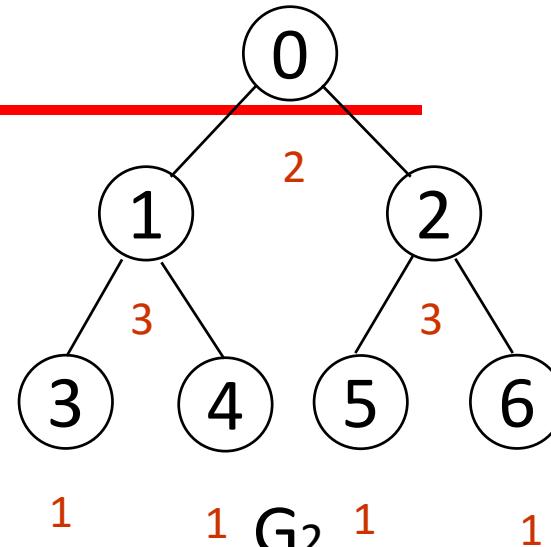
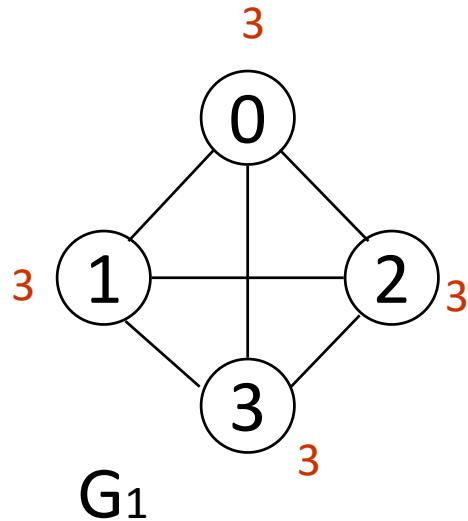
- For directed graph,

- the **in-degree** of a vertex v is the number of edges that have v as the head
- the **out-degree** of a vertex v is the number of edges that have v as the tail
- if d_i is the degree of a vertex i in a graph G with n vertices and e edges, the number of edges is

$$e = \left(\sum_{i=0}^{n-1} d_i \right) / 2$$

Why? Since adjacent vertices each count the adjoining edge, it will be counted twice

Examples



Terminology (cont.)

- **Path**

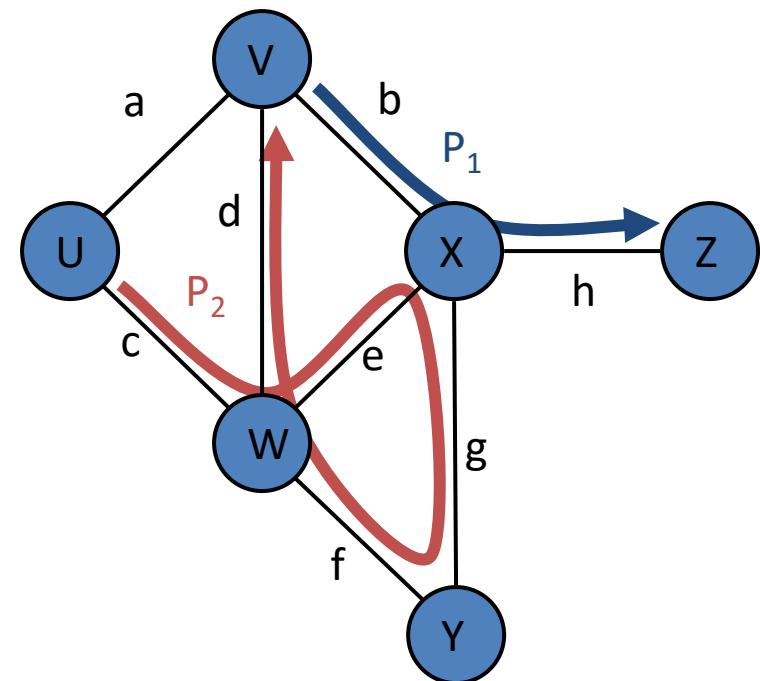
- sequence of alternating vertices and edges
- begins with a vertex
- ends with a vertex
- each edge is preceded and followed by its endpoints

- **Simple path**

- path such that all its vertices and edges are distinct

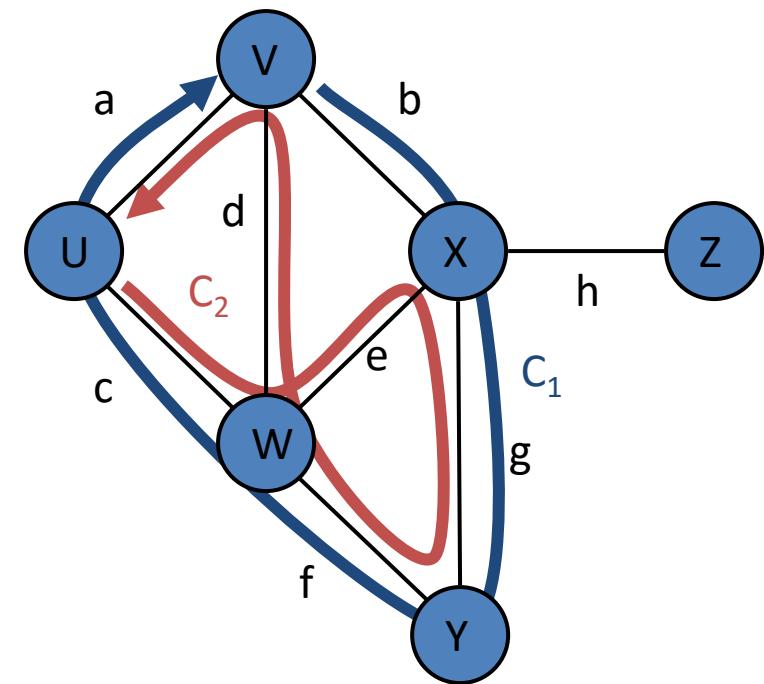
- **Examples**

- $P_1 = (V, b, X, h, Z)$ is a simple path
- $P_2 = (U, c, W, e, X, g, Y, f, W, d, V)$ is a path that is not simple



Terminology (cont.)

- **Cycle**
 - circular sequence of alternating vertices and edges
 - each edge is preceded and followed by its endpoints
 - **Simple cycle**
 - cycle such that all its vertices and edges are distinct
 - **Examples**
 - $C_1 = (V, b, X, g, Y, f, W, c, U, a, \leftarrow)$ is a simple cycle
 - $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, \leftarrow)$ is a cycle that is not simple



Properties

Property 1

$$\sum_v \deg(v) = 2m$$

Proof:

each edge is counted twice

Property 2

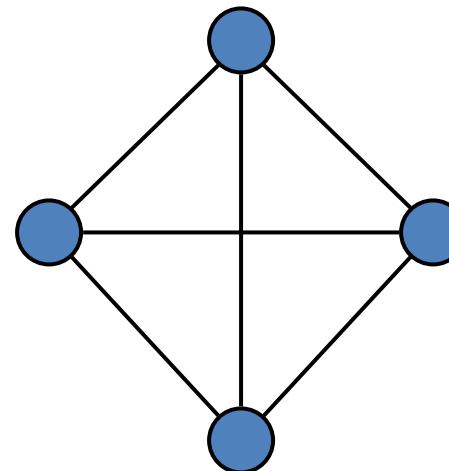
In an undirected graph with no self-loops and no multiple edges
 $m \leq n(n - 1)/2$

Proof:

each vertex has degree at most $(n - 1)$

Notation

n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v

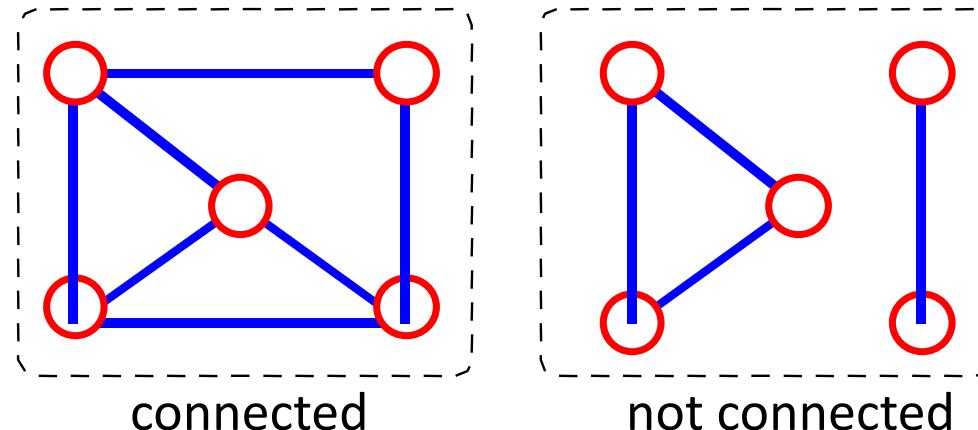


Example

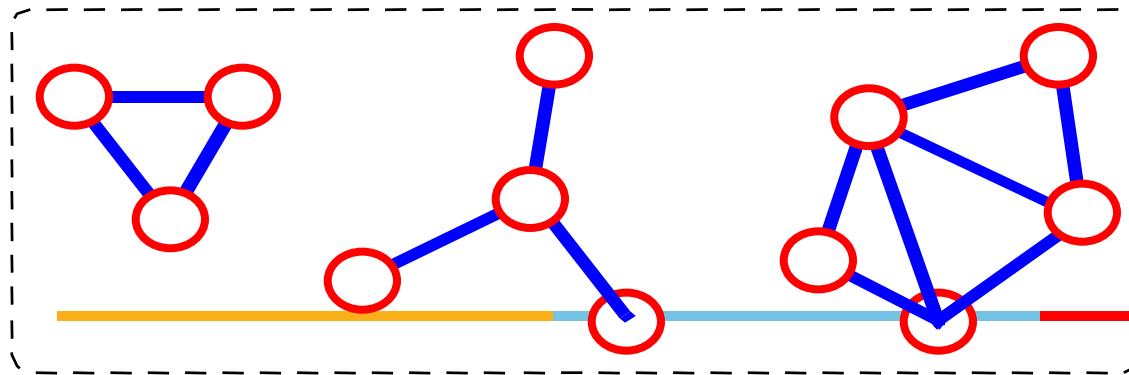
- $n = 4$
- $m = 6$
- $\deg(v) = 3$

Even More Terminology

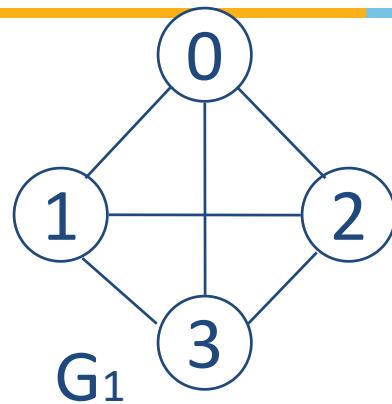
- **connected graph**: any two vertices are connected by some path



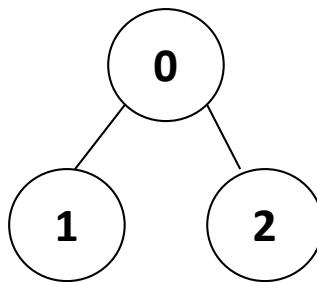
- **subgraph**: subset of vertices and edges forming a graph
- **connected component**: maximal connected subgraph. E.g., the graph below has 3 connected components.



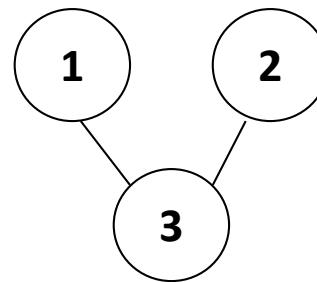
Subgraph Examples



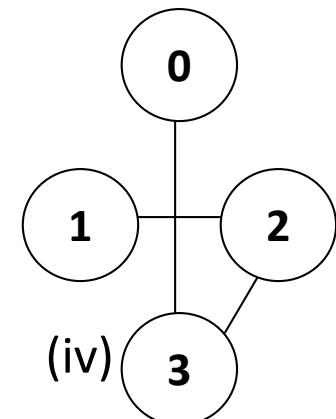
(i)



(ii)

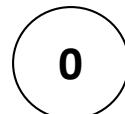
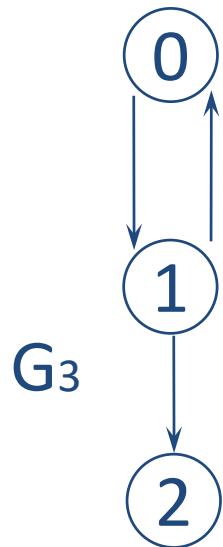


(iii)

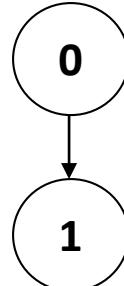


(iv)

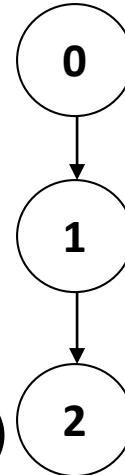
Some of the subgraph of G_1



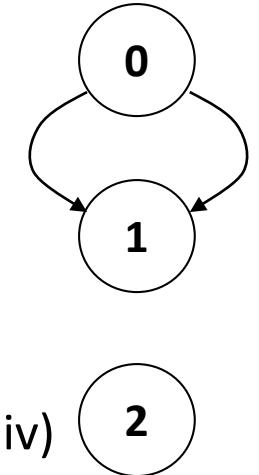
(i)



(ii)



(iii)



(iv)

Some of the subgraph of G_3

Trees

- Tree is a special case of a graph. Each node has zero or more child nodes, which are below it in the tree.
- A tree is a connected acyclic graph
- Already seen.
- **Forest** - collection of trees

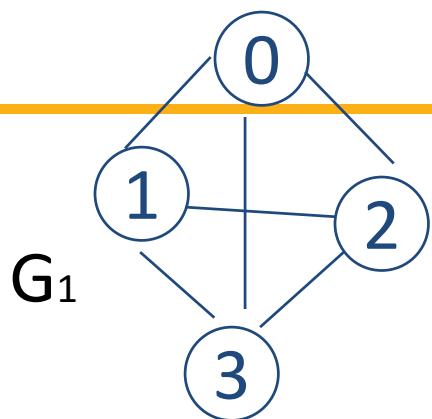
Graph Representations

- Adjacency Matrix
- Adjacency Lists

Adjacency Matrix

- Let $G=(V,E)$ be a graph with n vertices.
- The **adjacency matrix** of G is a two-dimensional n by n array, say adj_mat
- If the edge (v_i, v_j) is in $E(G)$, $\text{adj_mat}[i][j]=1$
- If there is no such edge in $E(G)$, $\text{adj_mat}[i][j]=0$
- The adjacency matrix for an undirected graph is **symmetric**; the adjacency matrix for a digraph need not be symmetric

Examples for Adjacency Matrix



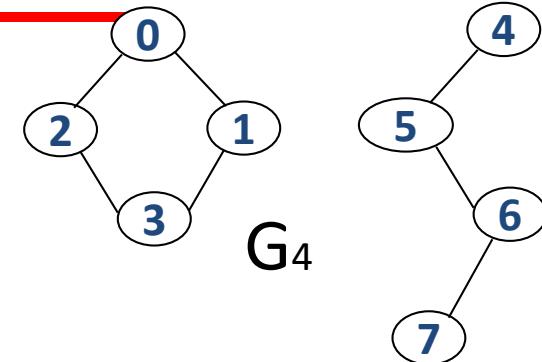
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$



G_2

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

symmetric



G_4

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Merits of Adjacency Matrix

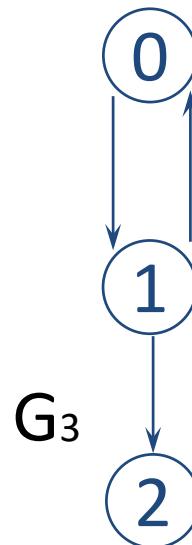
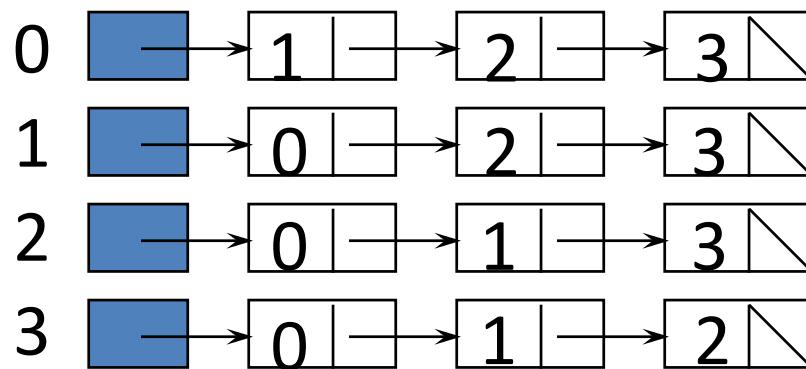
- From the adjacency matrix, to determine the connection of vertices is easy
- The degree of a vertex is $\sum_{j=0}^{n-1} adj_mat[i][j]$
- For a digraph the row sum is the `out_degree`, while the column sum is the `in_degree`

$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \quad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

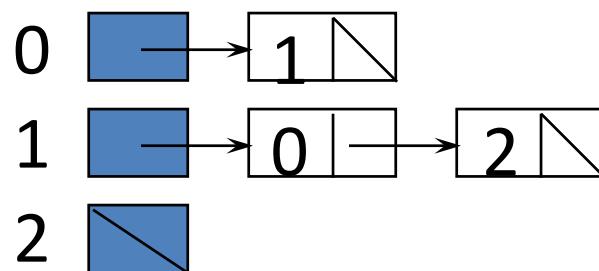
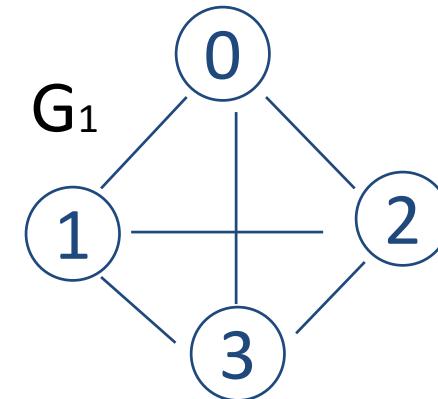
Cons : No matter how few edges the graph has, the matrix takes $O(n^2)$ in memory

Adjacency Lists Representation

- A graph of n nodes is represented by a one-dimensional array L of linked lists, where
 - $L[i]$ is the linked list containing all the nodes adjacent from node i .
 - The nodes in the list $L[i]$ are in no particular order



An undirected graph with n vertices and e edges ==> n head nodes and $2e$ list nodes



Pros and Cons of Adjacency Lists

- **Pros**
 - Saves on space (memory): the representation takes as many memory words as there are nodes and edges.
- **Cons**
 - It can take up to $O(n)$ time to determine if a pair of nodes (i,j) is an edge: one would have to search the linked list $L[i]$, which takes time proportional to the length of $L[i]$.



BITS Pilani
Pilani Campus

Course Name : **Data Structures & Algorithms**

Bharat Deshpande
Computer Science & Information Systems

Graph Traversal

- **Problem:** Search for a certain node or traverse all nodes in the graph.
 - Depth First Search
 - Once a possible path is found, continue the search until the end of the path
 - Breadth First Search
 - Start several paths at a time, and advance in each one step at a time
-

Exploring a Labyrinth Without Getting Lost

- A **depth-first search (DFS)** in an undirected graph G is like wandering in a labyrinth with a string and a can of red paint without getting lost.
- We start at vertex s , tying the end of our string to the point and painting s “visited”. Next we label s as our current vertex called u .
- Now we travel along an arbitrary edge (u, v) .
- If edge (u, v) leads us to an already visited vertex v we return to u .
- If vertex v is unvisited, we unroll our string and move to v , paint v “visited”, set v as our current vertex, and repeat the previous steps.
- The process terminates when our backtracking leads us back to the start index s , and there are no more unexplored edges incident on s .

Depth-First Search

Algorithm DFS(v); Input: A vertex v in a graph

Output: A labeling of the edges as “discovery” edges and “backedges”

for each edge e incident on v **do**

if edge e is unexplored **then** let w be the other endpoint of e

if vertex w is unexplored **then** label e as a **discovery edge**
 recursively call **DFS(w)**

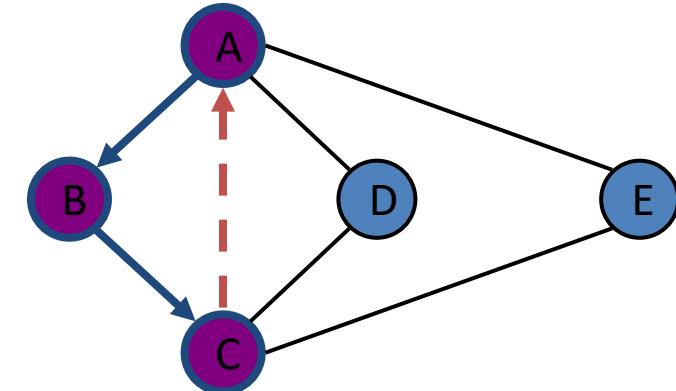
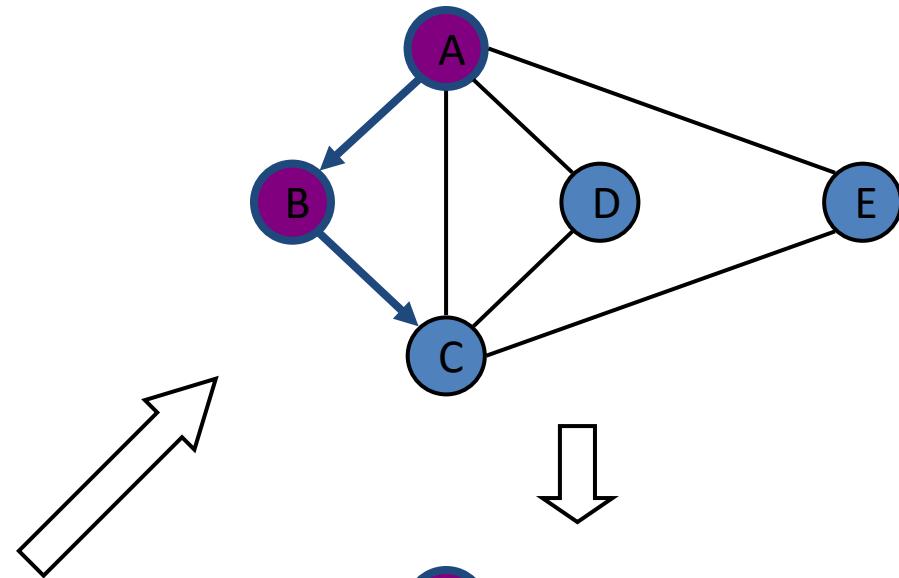
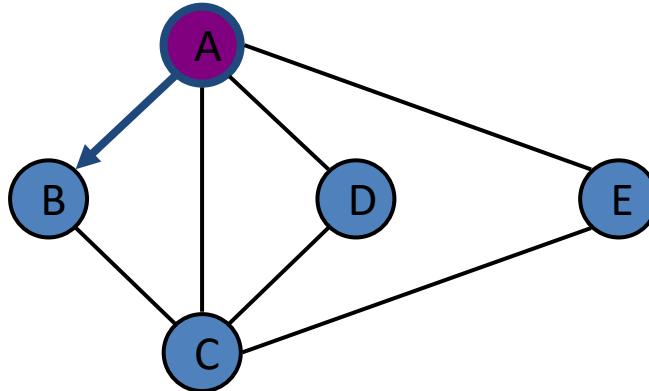
else label e as a **backedge**

Property 1

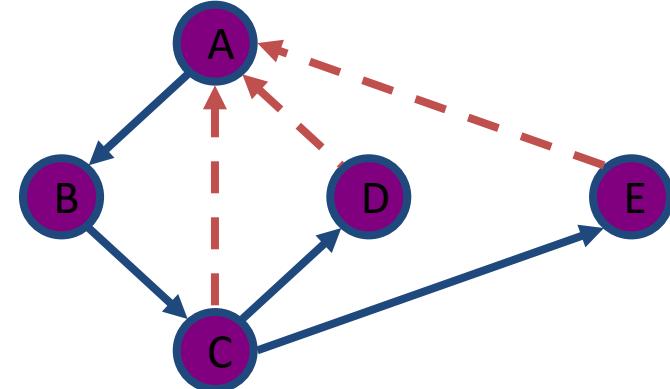
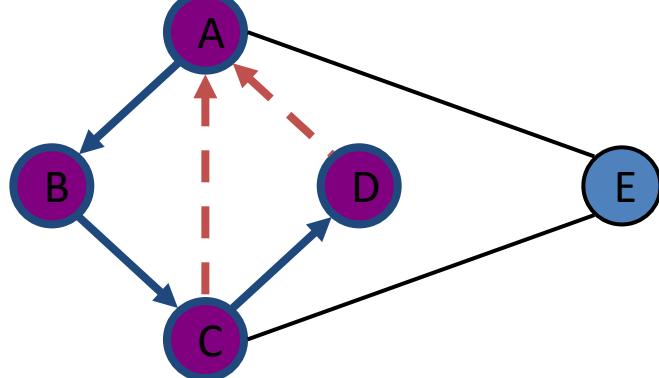
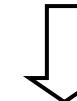
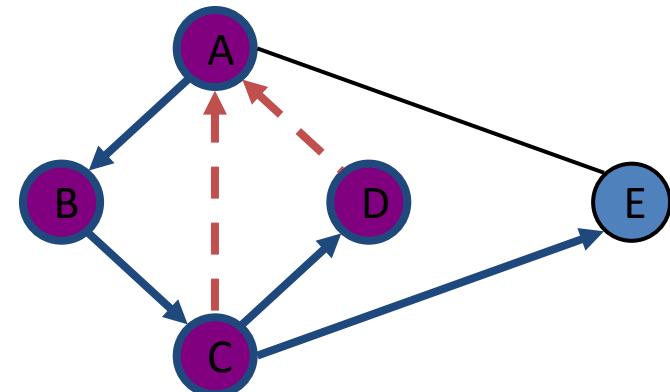
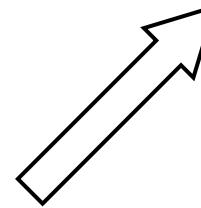
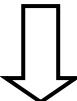
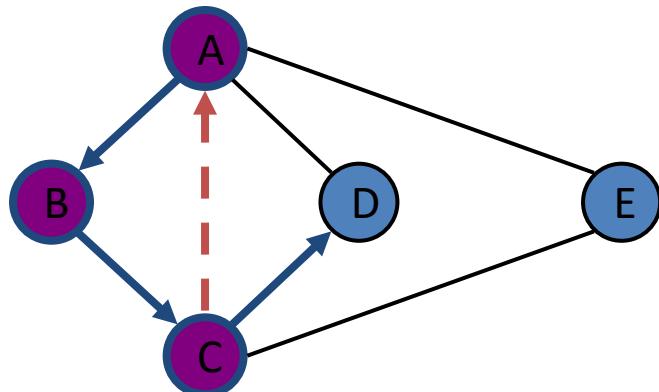
$DFS(G, v)$ visits all the vertices and edges in the connected component of v

Example

-  unexplored vertex
-  visited vertex
- unexplored edge
- discovery edge
- - - → back edge



Example (cont.)



Analysis of DFS

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
 - once as **UNEXPLORED**
 - once as **VISITED**
- Each edge is labeled twice
 - once as **UNEXPLORED**
 - once as **DISCOVERY** or **BACK**
- Method `incidentEdges` is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure

Path Finding

- We can specialize the DFS algorithm to **find a path between two given vertices u and z** using the template method pattern
- We call **$DFS(G, u)$** with u as the start vertex
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex z is encountered, we return the path as the contents of the stack

Cycle Finding

- We can specialize the DFS algorithm to find a simple cycle using the template method pattern
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w

Breadth-First Search

- Instead of going as far as possible, BFS tries to search all paths.
- BFS makes use of a queue to store visited (but not dead) vertices, expanding the path from the earliest visited vertices
- The starting vertex s has level 0, and, as in **DFS**, defines that point as an “anchor.”
- In the first round, the string is unrolled the length of one edge, and all of the nodes that are only one edge away from the anchor are visited.
- These edges are placed into level 1
- In the second round, all the new edges that can be reached by unrolling the string 2 edges are visited and placed in level 2.
- This continues until every vertex has been assigned a level.
- The label of any vertex v corresponds to the length of the shortest path from s to v .

BFS Pseudo-Code



Algorithm BFS(s): Input: A vertex s in a graph

Output: A labeling of the edges as “discovery” edges and “cross edges”

initialize container L_0 to contain vertex s

$i \leftarrow 0$

while L_i is not empty **do**

 create container L_{i+1} to initially be empty

for each vertex v in L_i **do**

if edge e incident on v **do**

 let w be the other endpoint of e

if vertex w is unexplored **then**

 label e as a **discovery edge**

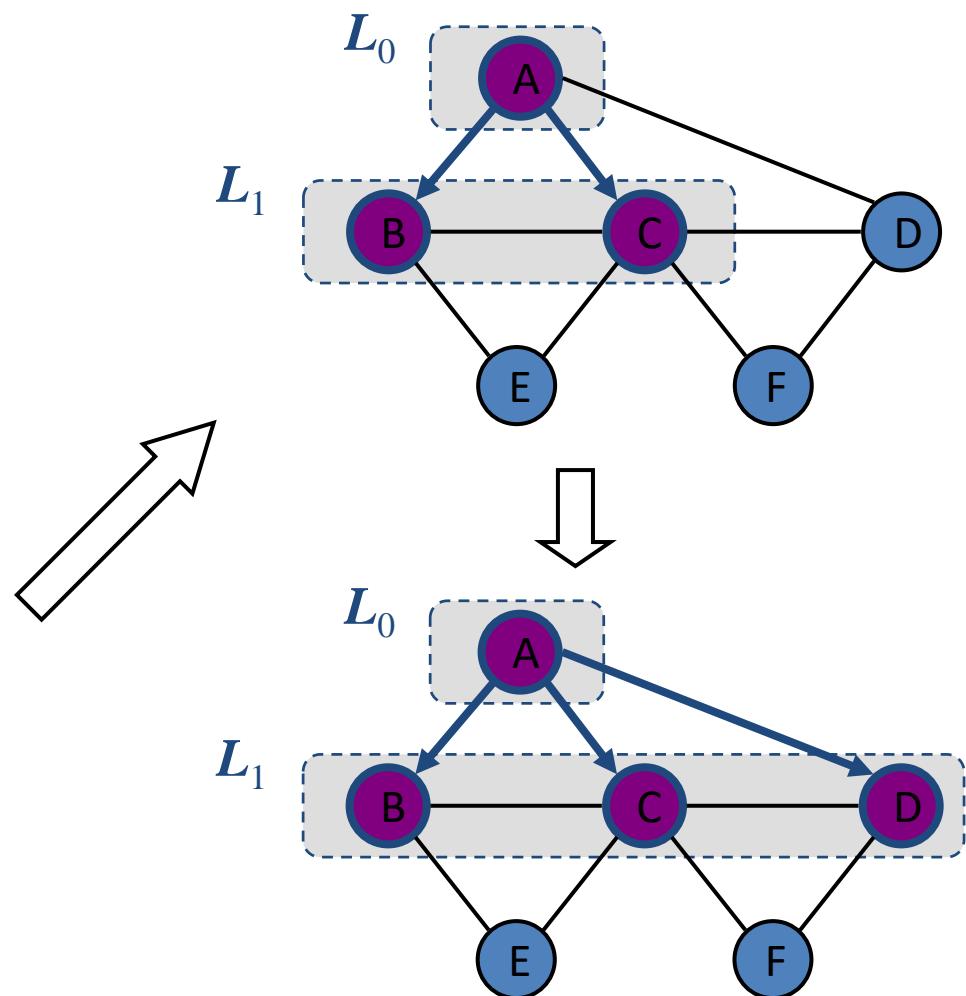
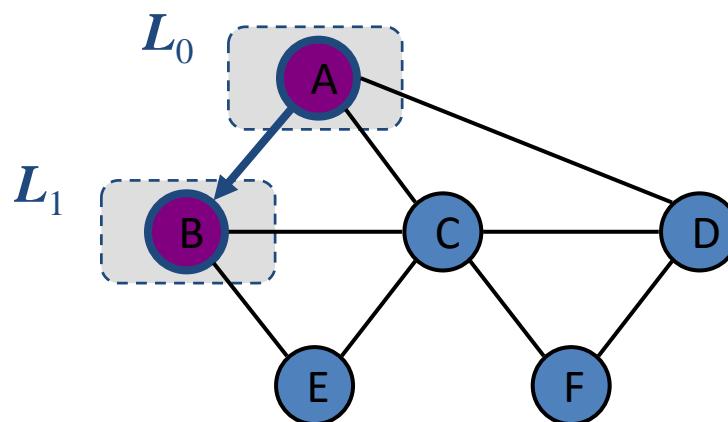
 insert w into L_{i+1}

else label e as a **cross edge**

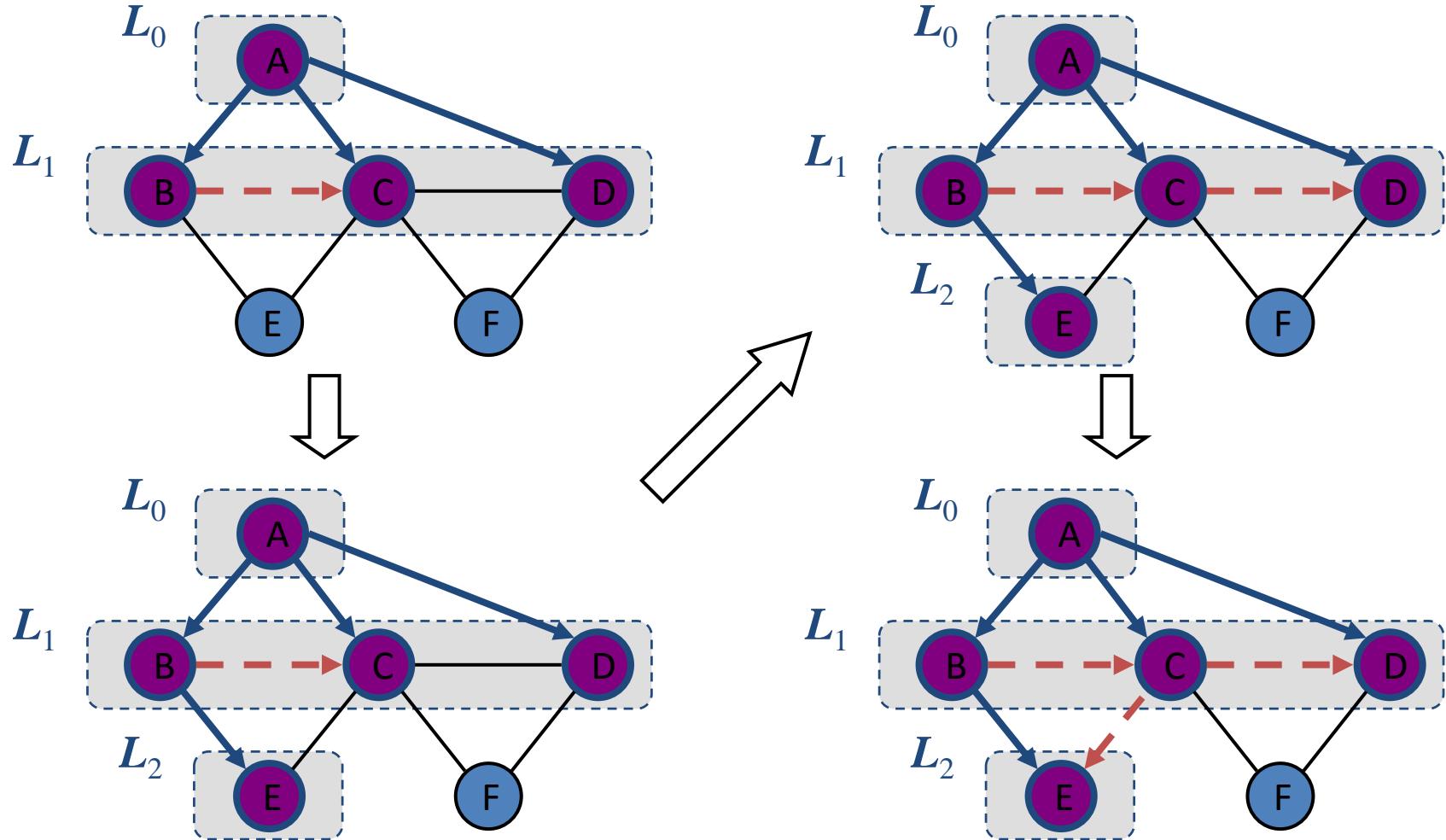
$i \leftarrow i + 1$

Example

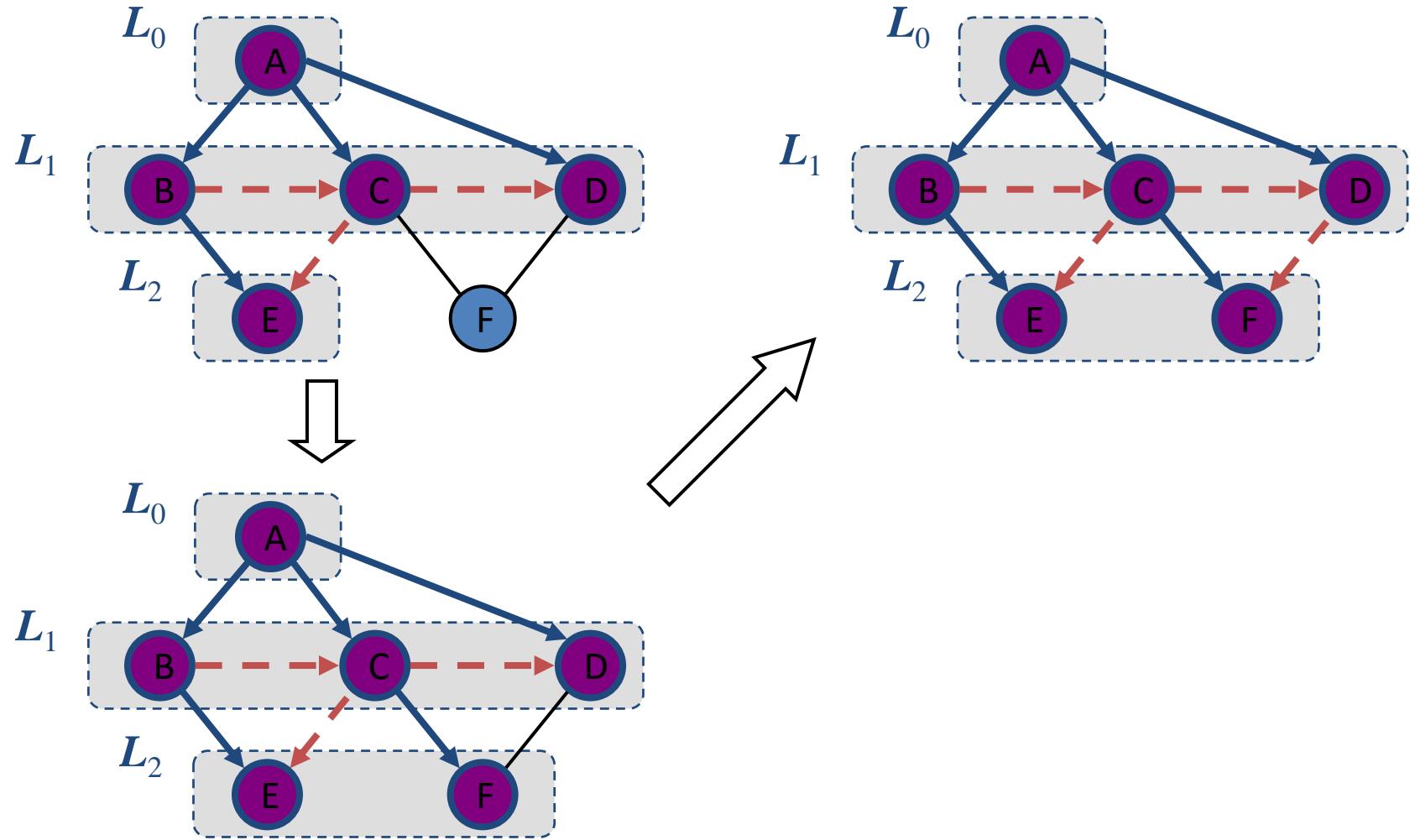
-  unexplored vertex
-  visited vertex
- unexplored edge
- discovery edge
- cross edge



Example (cont.)



Example (cont.)



Analysis

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
 - once as **UNEXPLORED**
 - once as **VISITED**
- Each edge is labeled twice
 - once as **UNEXPLORED**
 - once as **DISCOVERY** or **CROSS**
- Each vertex is inserted once into a sequence L_i
- Method `incidentEdges` is called once for each vertex
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure

Applications

We can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time

- Compute the connected components of G
- Find a simple cycle in G , or report that G is a forest
- Given two vertices of G , find a path in G between them with the minimum number of edges, or report that no such path exists



BITS Pilani
Pilani Campus

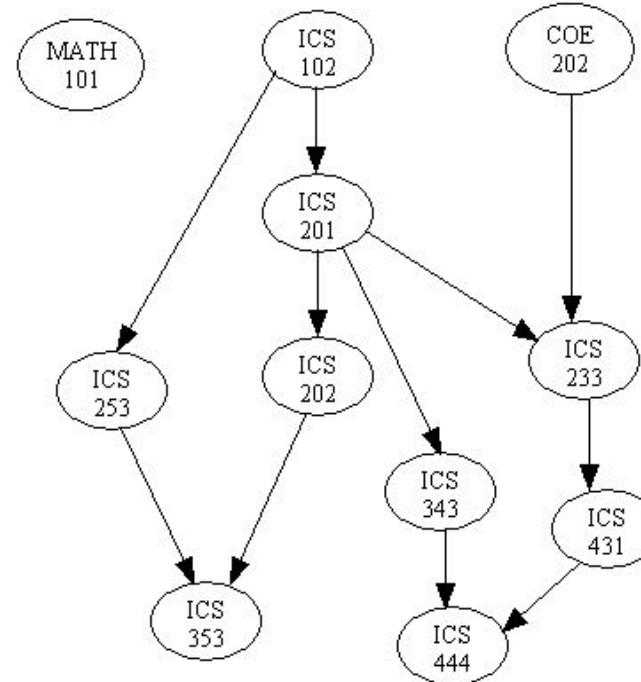
Course Name : Data Structures & Algorithms

Bharat Deshpande
Computer Science & Information Systems

Topological Sort

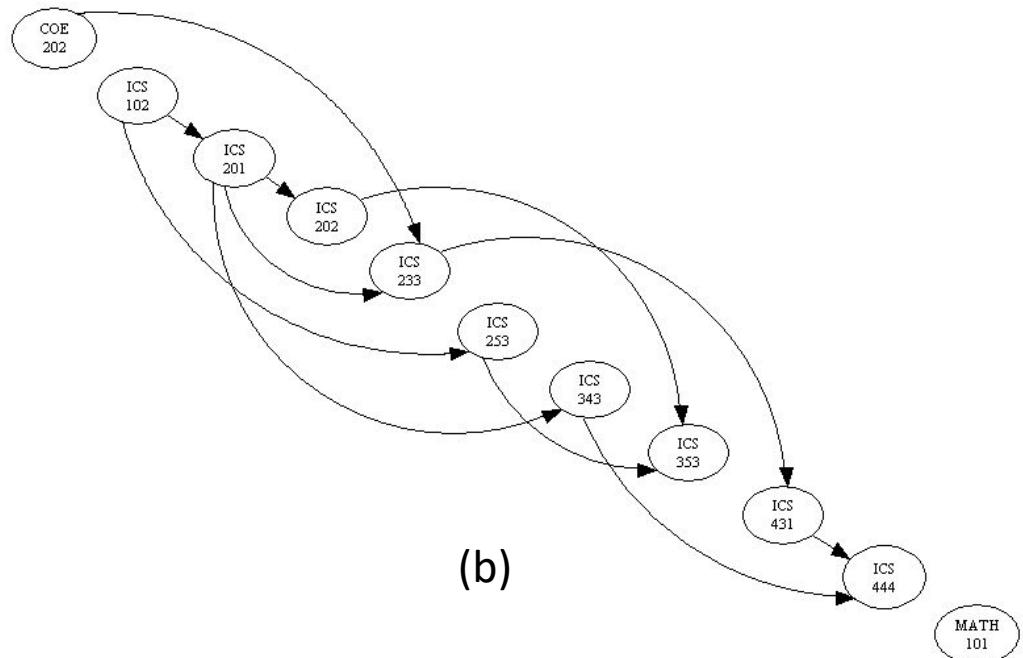
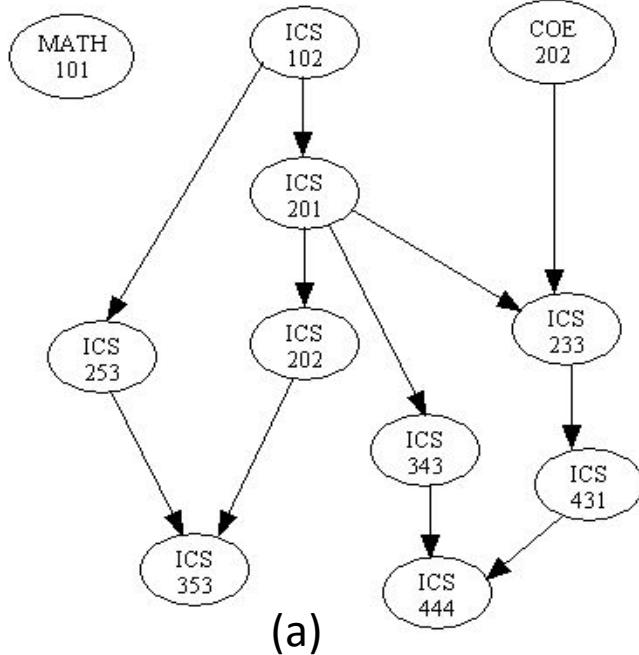
- There are many problems involving a set of tasks in which some of the tasks must be done before others.
- For example, consider the problem of taking a course only after taking its prerequisites.
- Is there any systematic way of linearly arranging the courses in the order that they should be taken?

Yes - Topological sort.



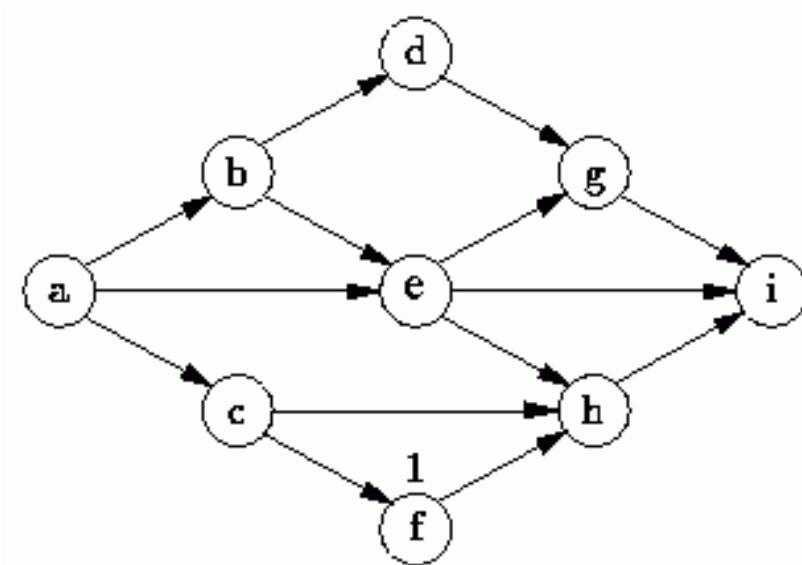
Definition of Topological Sort

- Topological sort is a method of arranging the vertices in a directed acyclic graph (DAG), as a sequence, such that no vertex appears in the sequence before its predecessor.
- The graph in (a) can be topologically sorted as in (b)



Topological Sort is not unique

- Topological sort is not unique.
- The following are all topological sort of the graph below:



$s1 = \{a, b, c, d, e, f, g, h, i\}$

$s2 = \{a, c, b, f, e, d, h, g, i\}$

$s3 = \{a, b, d, c, e, g, f, h, i\}$

$s4 = \{a, c, f, b, e, h, d, g, i\}$
etc.

DFS algorithm with timestamp

DFS(G)

```

1 for each vertex  $u \in V[G]$ 
2     do  $color[u] \leftarrow$  WHITE
3  $time \leftarrow 0$ 
4 for each vertex  $u \in V[G]$ 
5     do if  $color[u] =$  WHITE
6         then DFS-VISIT( $u$ )

```

Initialize all vertices

DFS-VISIT(u)

```

1  $color[u] \leftarrow$  GRAY           ▷ White vertex  $u$  discovered.
2  $d[u] \leftarrow time$            ▷ Mark with discovery time.
3  $time \leftarrow time + 1$        ▷ Tick global time.
4 for each  $v \in Adj[u]$       ▷ Explore all edges  $(u, v)$ .
5     do if  $color[v] =$  WHITE
6         then DFS-VISIT( $v$ )
7  $color[u] \leftarrow$  BLACK        ▷ Blacken  $u$ ; it is finished.
8  $f[u] \leftarrow time$            ▷ Mark with finishing time.
9  $time \leftarrow time + 1$        ▷ Tick global time.

```

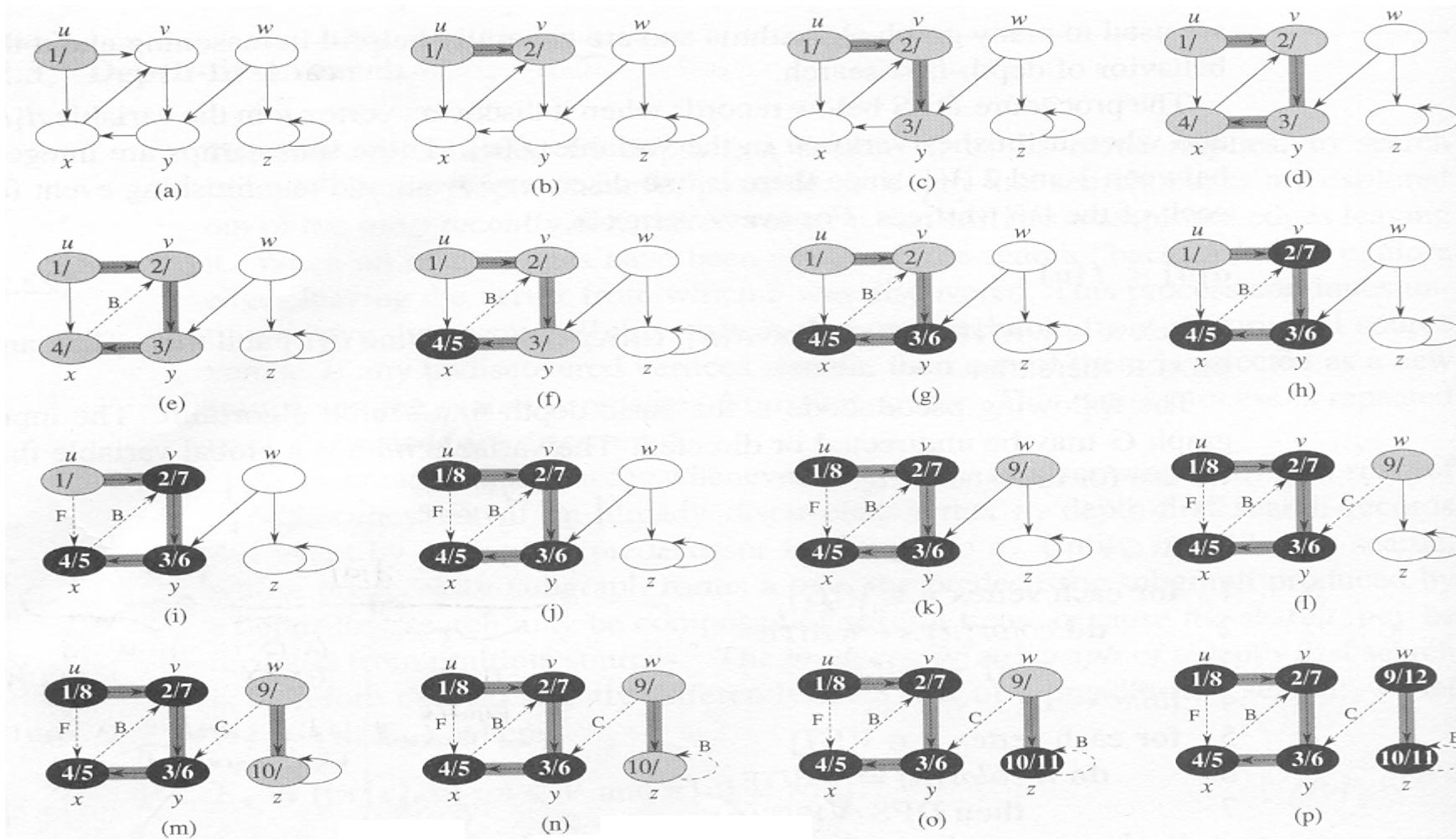
Visit all children recursively

DFS - Example

innovate

achieve

lead



DFS algorithm

- Initialize – color all vertices white
- Visit each and every white vertex using DFS-Visit
- Each call to DFS-Visit(u) roots a new tree of the depth-first forest at vertex u .
- A vertex is **white** if it is undiscovered
- A vertex is **gray** if it has been discovered but not all of its edges have been discovered
- A vertex is **black** after all of its adjacent vertices have been discovered (the adj. list was examined completely)

- When DFS returns, every vertex u is assigned a discovery time $d[u]$, and a finishing time $f[u]$

Topological Sort

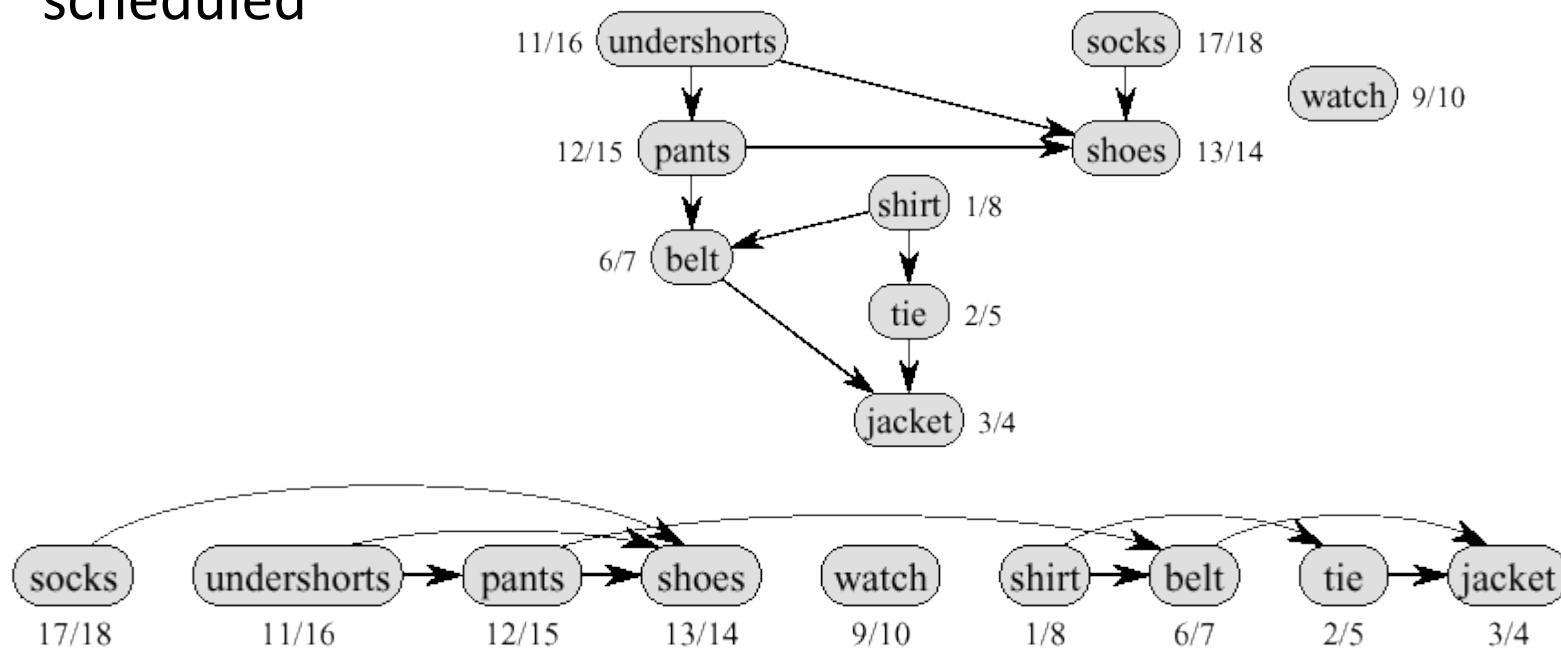
-
- Sorting of a directed acyclic graph (**DAG**)
 - A topological sort of a DAG is a linear ordering of all its vertices such that for any edge (u,v) in the DAG, u appears before v in the ordering
 - The following algorithm topologically sorts a DAG

Topological-Sort(G)

- 1) call DFS(G) to compute finishing times $f[v]$ for each vertex v
 - 2) as each vertex is finished, insert it onto the front of a linked list
 - 3) return the linked list of vertices
- The linked lists comprises a total ordering

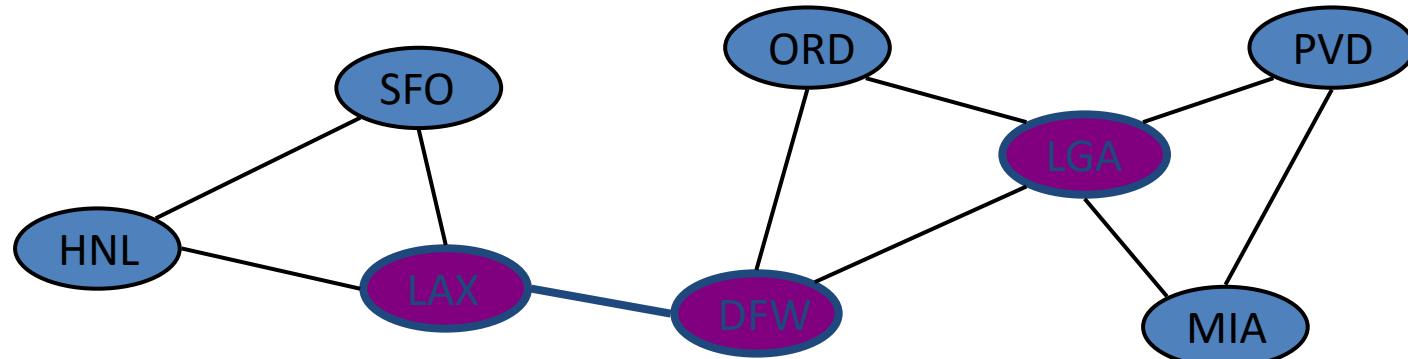
Topological Sort Example

- Precedence relations: an edge from x to y means one must be done with x before one can do y
- Idea:** can schedule task only when all of its subtasks have been scheduled



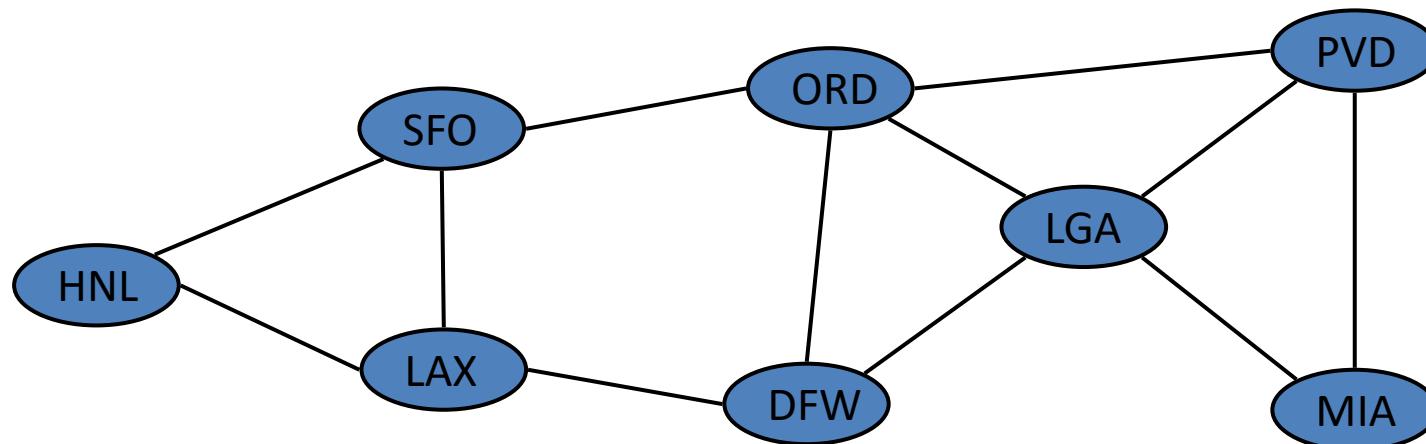
Separation Edges and Vertices

- **Definitions:** Let G be a connected graph
 - A **separation edge** of G is an edge whose removal disconnects G
 - A **separation vertex** of G is a vertex whose removal disconnects G
- **Applications**
 - Separation edges and vertices represent single points of failure in a network and are critical to the operation of the network
- **Example**
 - DFW, LGA and LAX are separation vertices
 - (DFW,LAX) is a separation edge



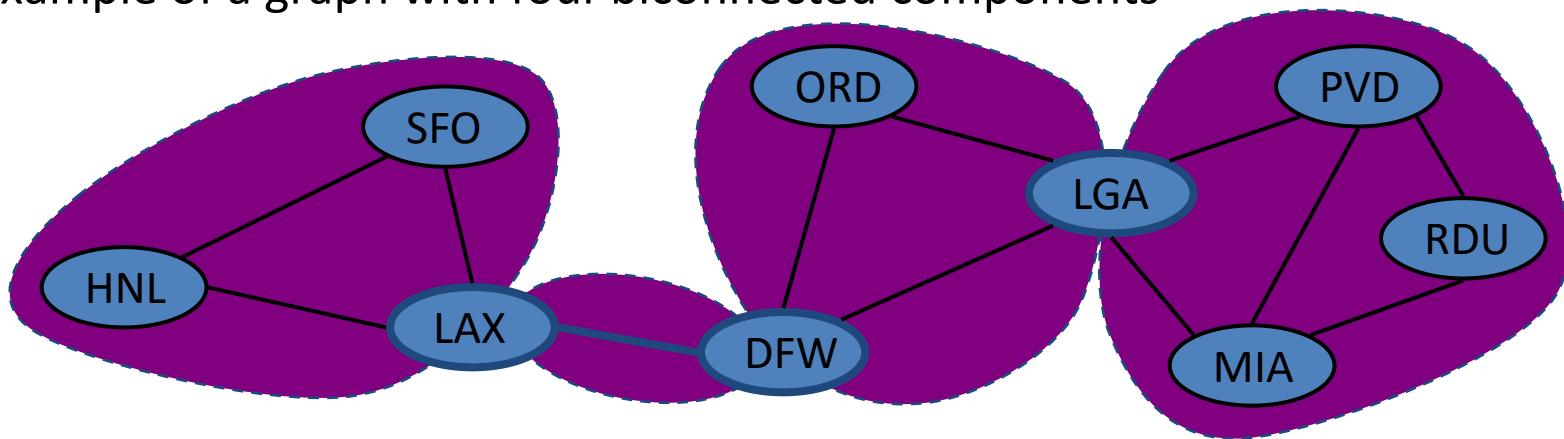
Biconnected Graph

- Equivalent definitions of a **biconnected graph G**
 - Graph G has no separation edges and no separation vertices
 - For any two vertices u and v of G , there are two disjoint simple paths between u and v (i.e., two simple paths between u and v that share no other vertices or edges)
 - For any two vertices u and v of G , there is a simple cycle containing u and v
- **Example**



Biconnected Components

- Biconnected component of a graph \mathbf{G}
 - A maximal biconnected subgraph of \mathbf{G} , or
 - A subgraph consisting of a separation edge of \mathbf{G} and its end vertices
- Interaction of biconnected components
 - An edge belongs to exactly one biconnected component
 - A nonseparation vertex belongs to exactly one biconnected component
 - A separation vertex belongs to two or more biconnected components
- Example of a graph with four biconnected components



Equivalence Classes

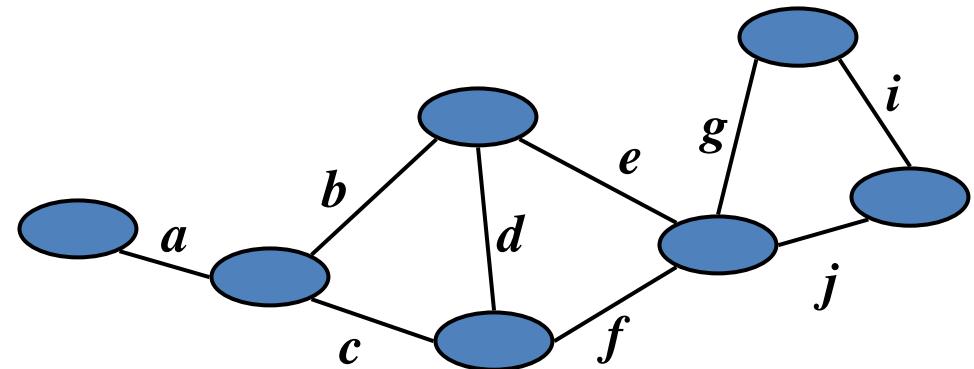
- Given a set S , a relation R on S is a set of ordered pairs of elements of S , i.e., R is a subset of $S \times S$
- An **equivalence relation** R on S satisfies the following properties
 - Reflexive:** $(x, x) \in R$
 - Symmetric:** $(x, y) \in R \Rightarrow (y, x) \in R$
 - Transitive:** $(x, y) \in R \wedge (y, z) \in R \Rightarrow (x, z) \in R$
- An equivalence relation R on S induces a **partition** of the elements of S into **equivalence classes**
- Example** (connectivity relation among the vertices of a graph):
 - Let V be the set of vertices of a graph G
 - Define the relation $C = \{(v, w) \in V \times V \text{ such that } G \text{ has a path from } v \text{ to } w\}$
 - Relation C is an equivalence relation
 - The equivalence classes of relation C are the vertices in each connected component of graph G

Link Relation

- Edges e and f of connected graph G are **linked** if
 - $e = f$, or
 - G has a simple cycle containing e and f

Theorems:

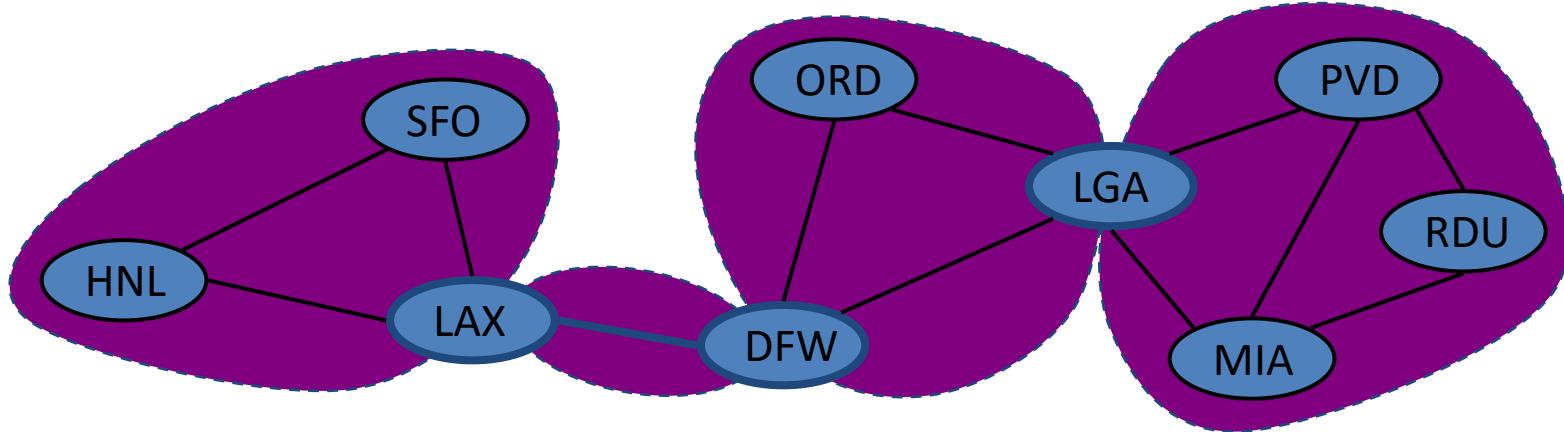
- The link relation on the edges of a graph is an equivalence relation
- A biconnected component of G is the subgraph induced by an equivalence class of linked edges.



Equivalence classes of linked edges:
 $\{a\}$ $\{b, c, d, e, f\}$ $\{g, i, j\}$

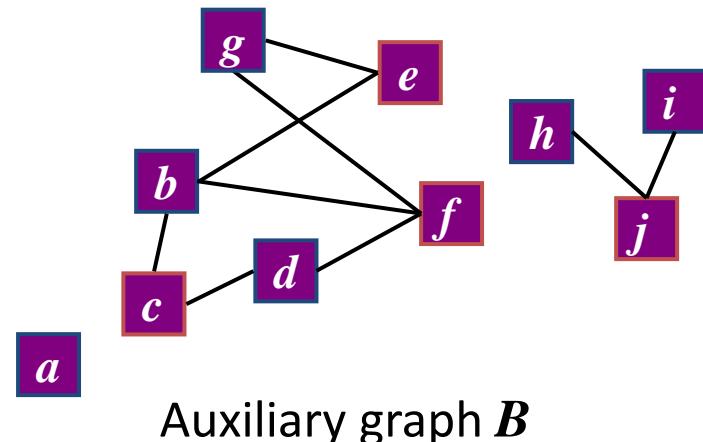
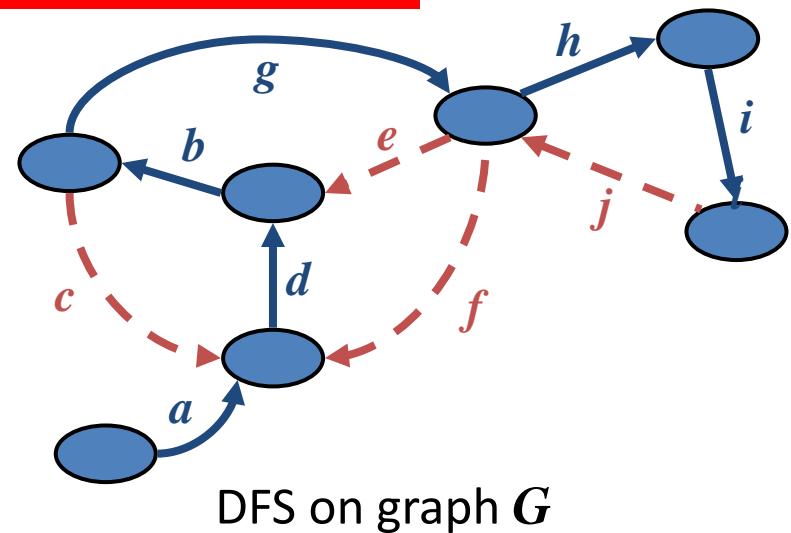
Link Components

- The link components of a connected graph \mathbf{G} are the equivalence classes of edges with respect to the link relation
- A biconnected component of \mathbf{G} is the subgraph of \mathbf{G} induced by an equivalence class of linked edges
- A separation edge is a single-element equivalence class of linked edges
- A separation vertex has incident edges in at least two distinct equivalence classes of linked edge



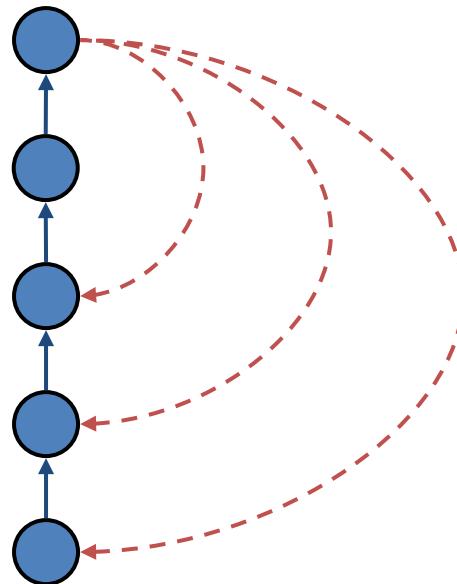
Auxiliary Graph

- **Auxiliary graph B** for a connected graph G
 - Associated with a DFS traversal of G
 - The vertices of B are the edges of G
 - For each **back edge e** of G , B has edges $(e, f_1), (e, f_2), \dots, (e, f_k)$, where f_1, f_2, \dots, f_k are the discovery edges of G that form a simple cycle with e
 - Its connected components correspond to the link components of G

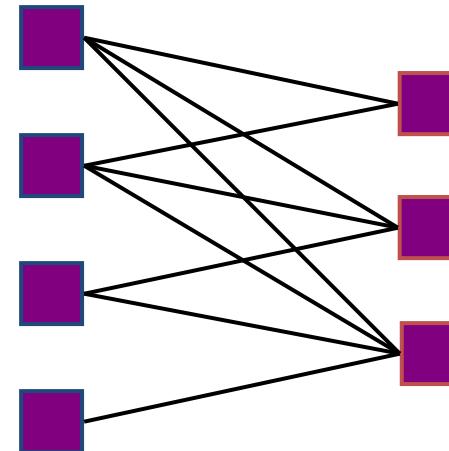


Auxiliary Graph (cont.)

- In the worst case, the number of edges of the auxiliary graph is proportional to nm



DFS on graph G



Auxiliary graph B



BITS Pilani
Pilani Campus

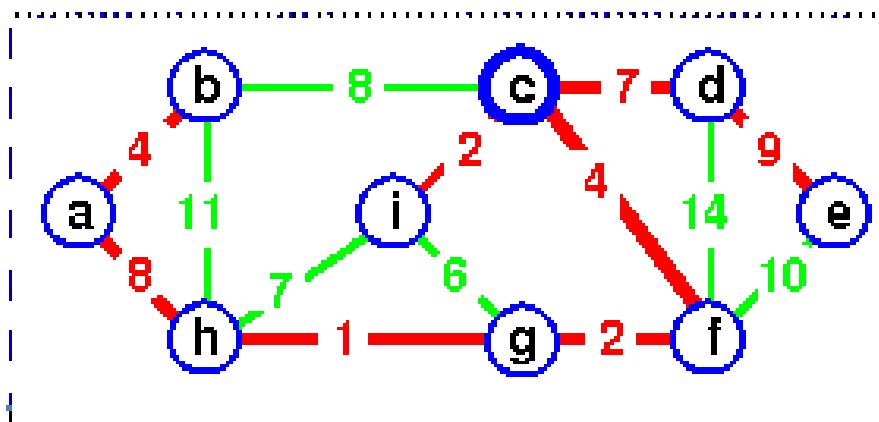
Course Name : **Data Structures & Algorithms**

Bharat Deshpande
Computer Science & Information Systems

Graphs - Definitions

- Spanning Tree

- A **spanning tree** is a set of $|V|-1$ edges that connect all the vertices of a graph

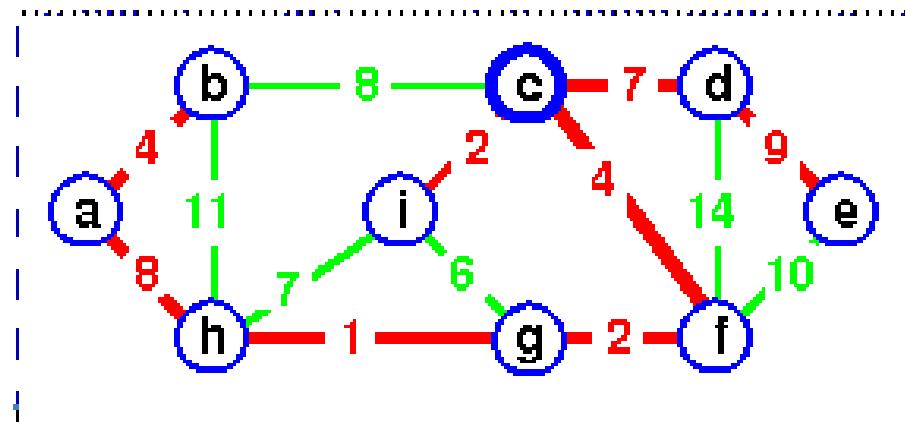


The red path connects all vertices, so it's a spanning tree

Graphs - Definitions

- **Minimum Spanning Tree**

- Generally there is more than one spanning tree
- If a cost c_{ij} is associated with edge $e_{ij} = (v_i, v_j)$
then the **minimum spanning tree** is the set of edges E_{span}
such that $C = \sum (c_{ij} \mid \forall e_{ij} \in E_{\text{span}})$ is a minimum



Other ST's can be formed ..

- Replace 2 with 7
- Replace 4 with 11

The red tree is the
Min ST

Growing a MST

- The procedure manages a set A that is always a subset of some MST.
- At each step, an edge (u, v) is added to A such that $A \cup \{(u, v)\}$ is also a subset of a MST.
- Edge (u, v) is called a **safe edge**.
- **GENERIC-MST(G)**

$A \leftarrow \emptyset$

while A does not form a spanning tree

do find an edge (u, v) that is safe for A

$A \leftarrow A \cup \{(u, v)\}$

return A

- We design two MST algorithms, each of which uses a specific rule to determine the safe edge to be added.

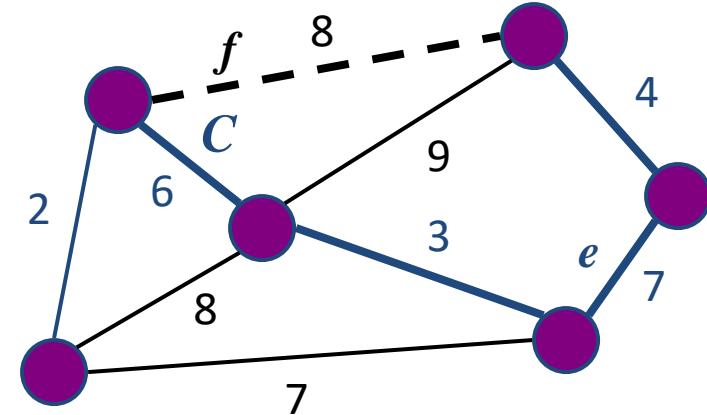
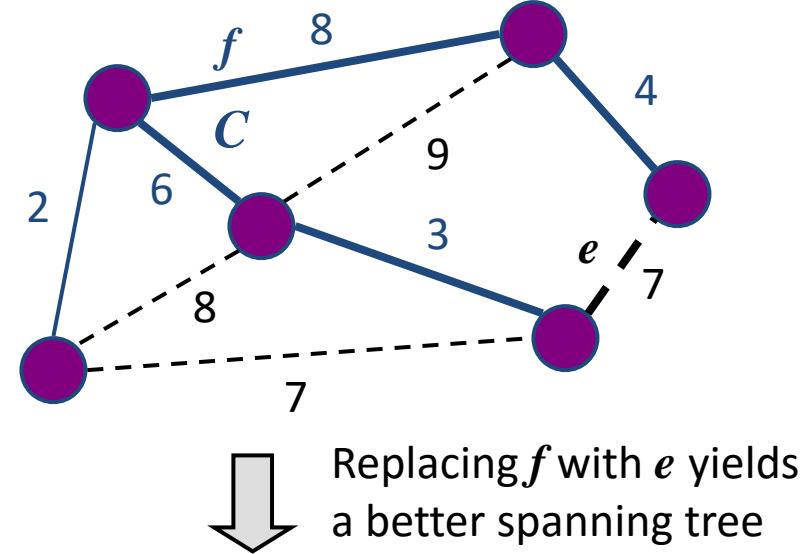
Cycle Property

Cycle Property:

- Let T be a minimum spanning tree of a weighted graph G
- Let e be an edge of G that is not in T and C let be the cycle formed by e with T
- For every edge f of C , $\text{weight}(f) \leq \text{weight}(e)$

Proof:

- By contradiction
- If $\text{weight}(f) > \text{weight}(e)$ we can get a spanning tree of smaller weight by replacing e with f



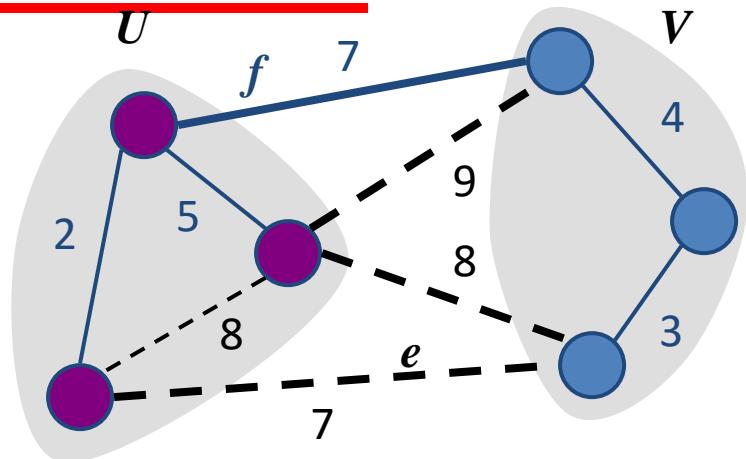
Partition Property

Partition Property:

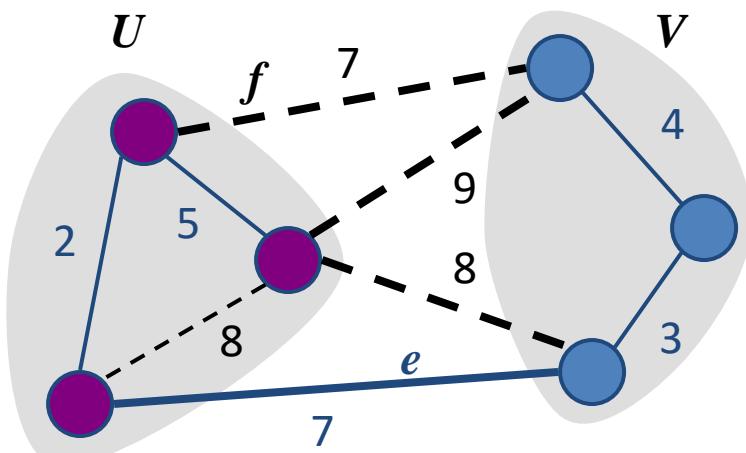
- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

Proof:

- Let T be an MST of G
- If T does not contain e , consider the cycle C formed by e with T and let f be an edge of C across the partition
- By the cycle property,
 $\text{weight}(f) \leq \text{weight}(e)$
- Thus, $\text{weight}(f) = \text{weight}(e)$
- We obtain another MST by replacing f with e



 Replacing f with e yields another MST

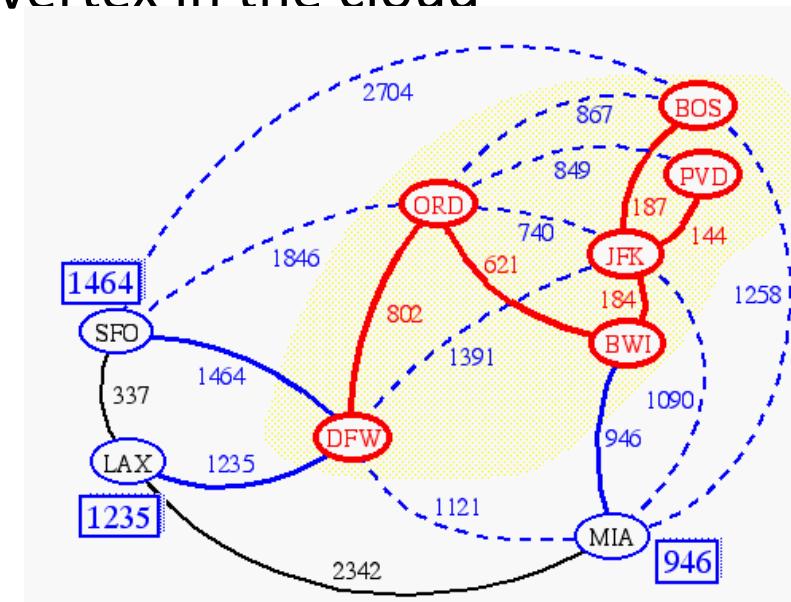


Prim-Jarník's Algorithm

- Initially discovered in 1930 by Vojtěch Jarník, then rediscovered in 1957 by Robert C. Prim
- We pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s
- We store with each vertex v a label $d(v)$ = the smallest weight of an edge connecting v to a vertex in the cloud

At each step:

- We add to the cloud the vertex u outside the cloud with the smallest distance label
- We update the labels of the vertices adjacent to u



Prim's Algorithm – Pseudo Code

MST-PRIM(G, r)

$Q \leftarrow V[G]$ // Q – priority queue initialize

for each $u \in Q$

do $d[u] \leftarrow \infty$

$D[r] = 0$

$\Pi[r] \leftarrow \text{NIL}$

while $Q \neq \emptyset$

do $u \leftarrow \text{EXTRACT-MIN}(Q)$

for each $v \in \text{Adj}[u]$

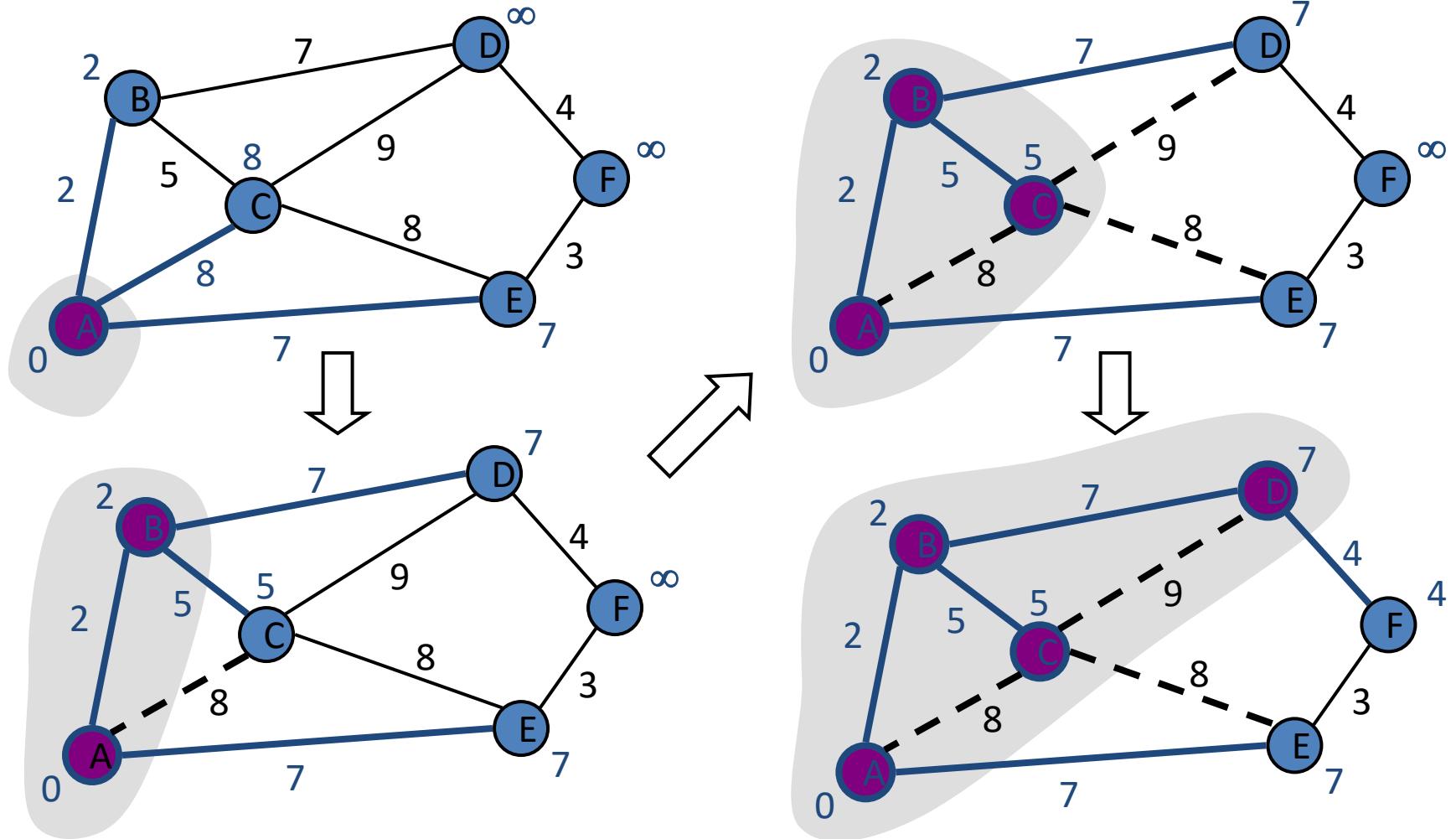
do if $v \in Q$ and $w(u, v) < d[v]$

then $\Pi[v] \leftarrow u$

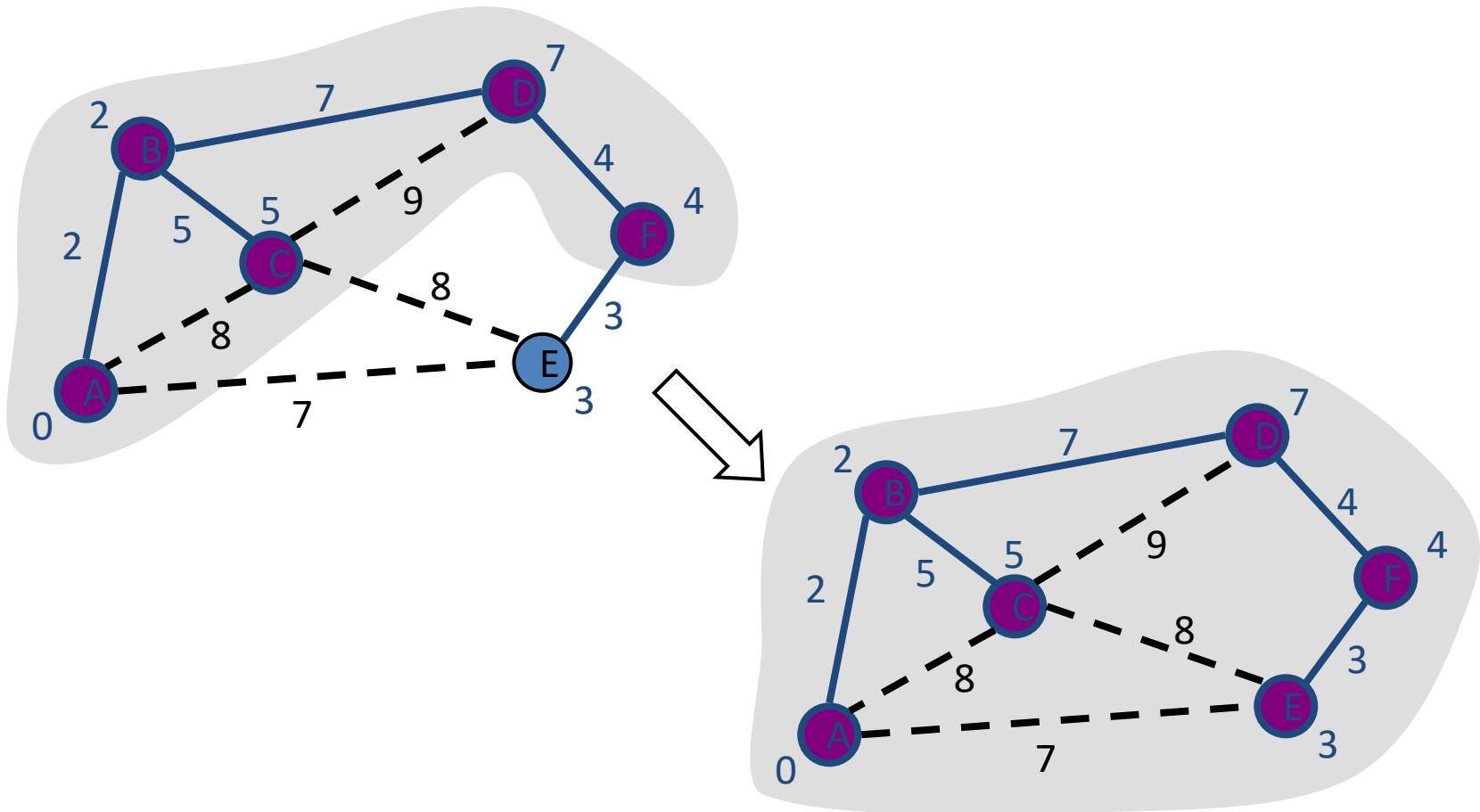
$d[v] \leftarrow w(u, v)$

The running time is $O(m \log n)$

Example



Example (contd.)



Kruskal's Algorithm

- Created in 1957 by Joseph Kruskal

Algorithm

- Starts by choosing an edge in the graph with minimum weight.
- Successively add edges with minimum weight that do not form a cycle with those edges already chosen.
- Terminates after $n-1$ edges have been selected (n is the number of vertices)

Note

In Kruskal's algorithm the set A is a forest.

The safe edge added to A is always a least-weight edge in the graph that connects two distinct components.

Kruskal's algorithm – Pseudo code



Algorithm Kruskal(G)

Input: A weighted graph G .

Output: An MST T for G .

Let P be a partition of the vertices of G , where each vertex forms a separate set.

Let Q be a priority queue storing the edges of G , sorted by their weights

Let T be an initially-empty tree

while Q is not empty **do**

$(u, v) \leftarrow Q.\text{removeMinElement}()$

if $P.\text{find}(u) \neq P.\text{find}(v)$ **then**

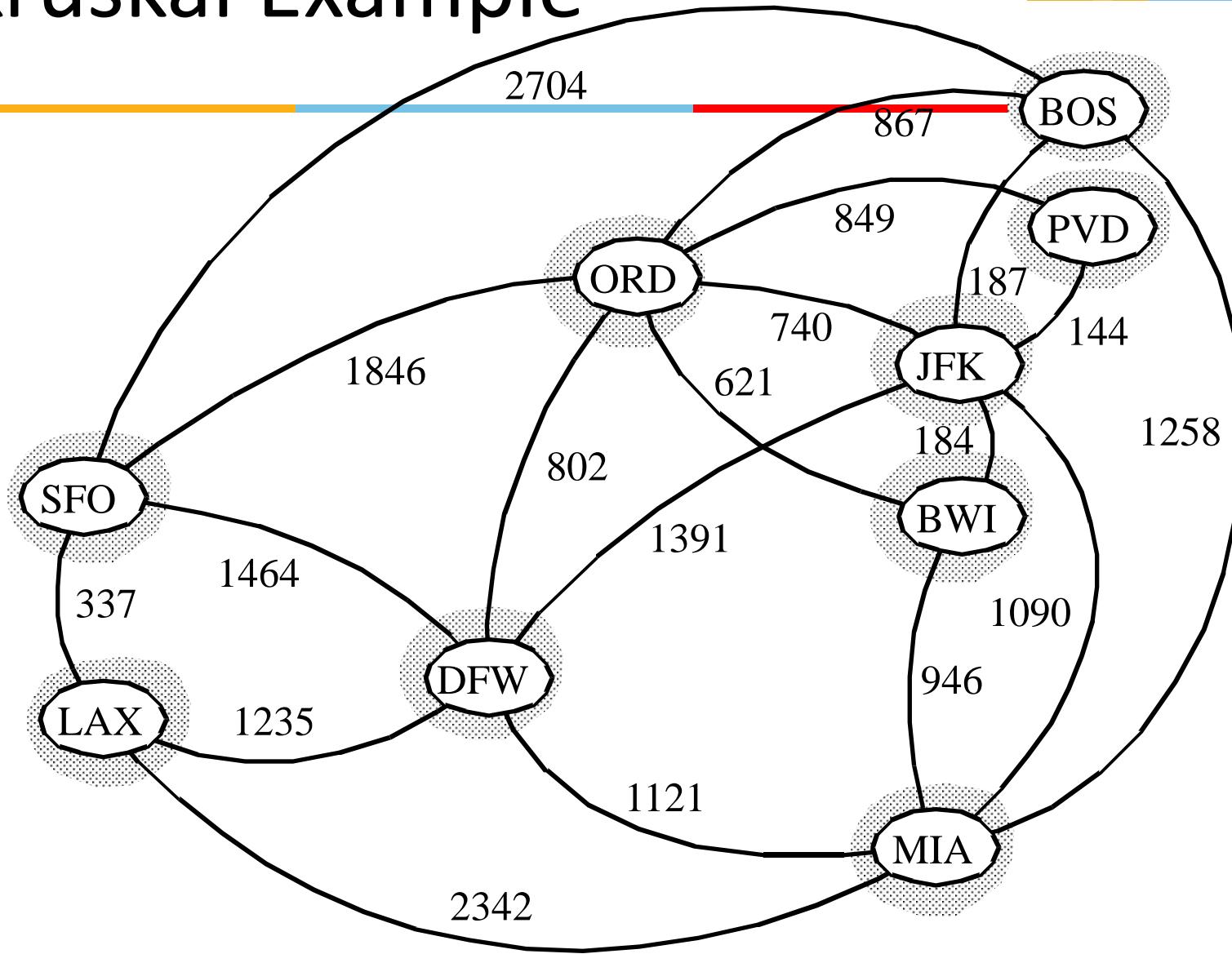
 Add (u, v) to T

$P.\text{union}(u, v)$

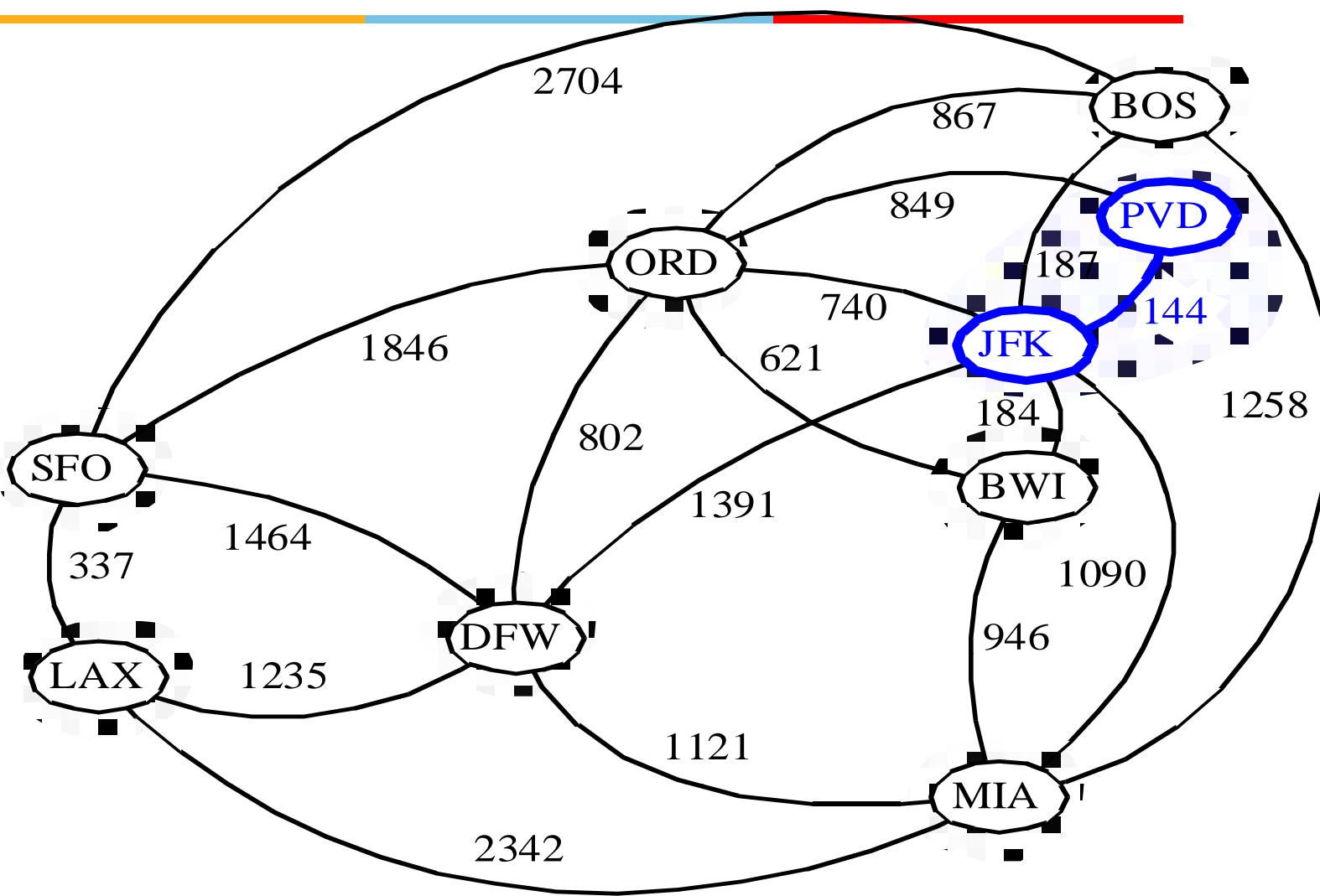
Running time: $O(m \log m)$

return T

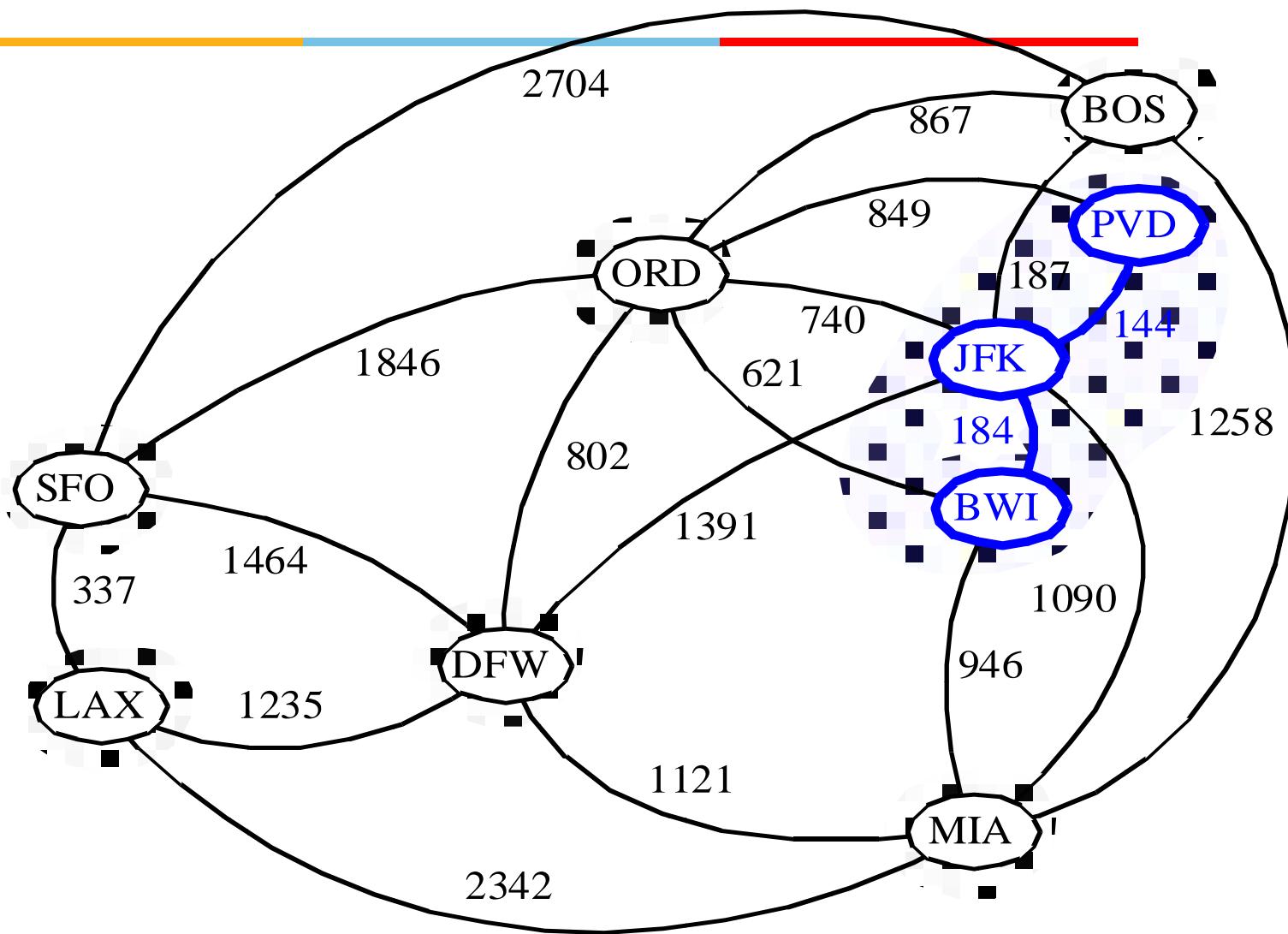
Kruskal Example



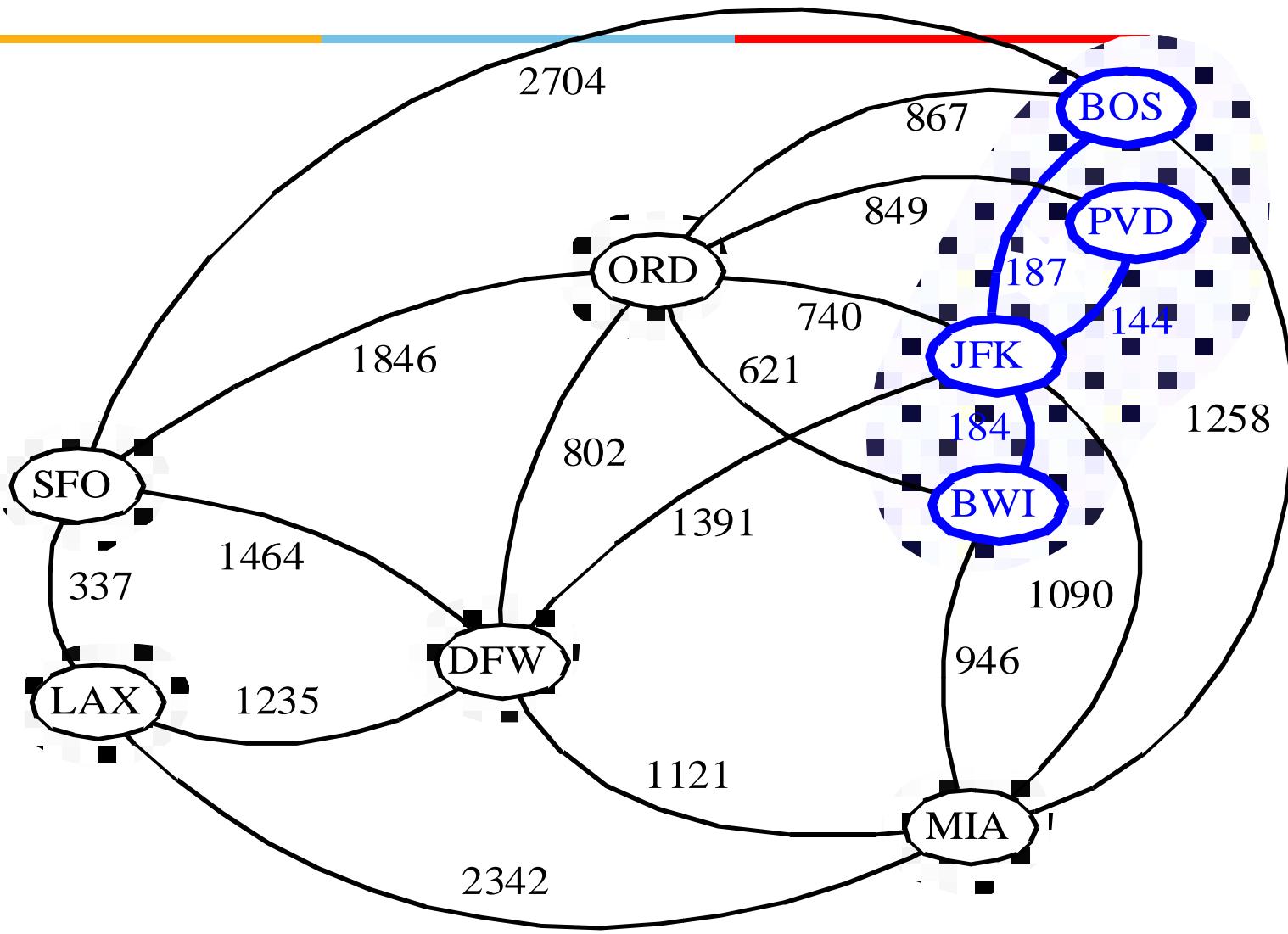
Example



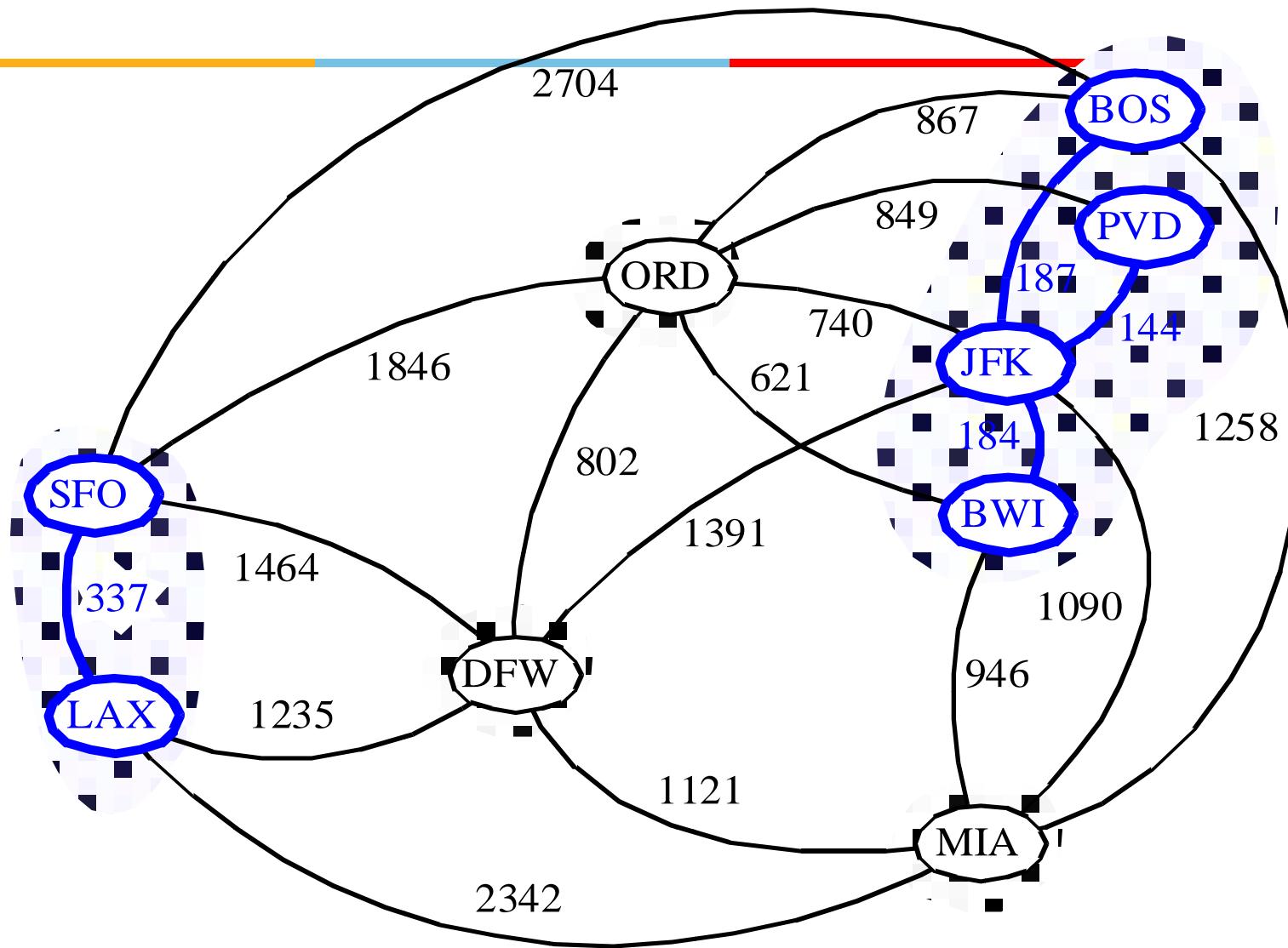
Example



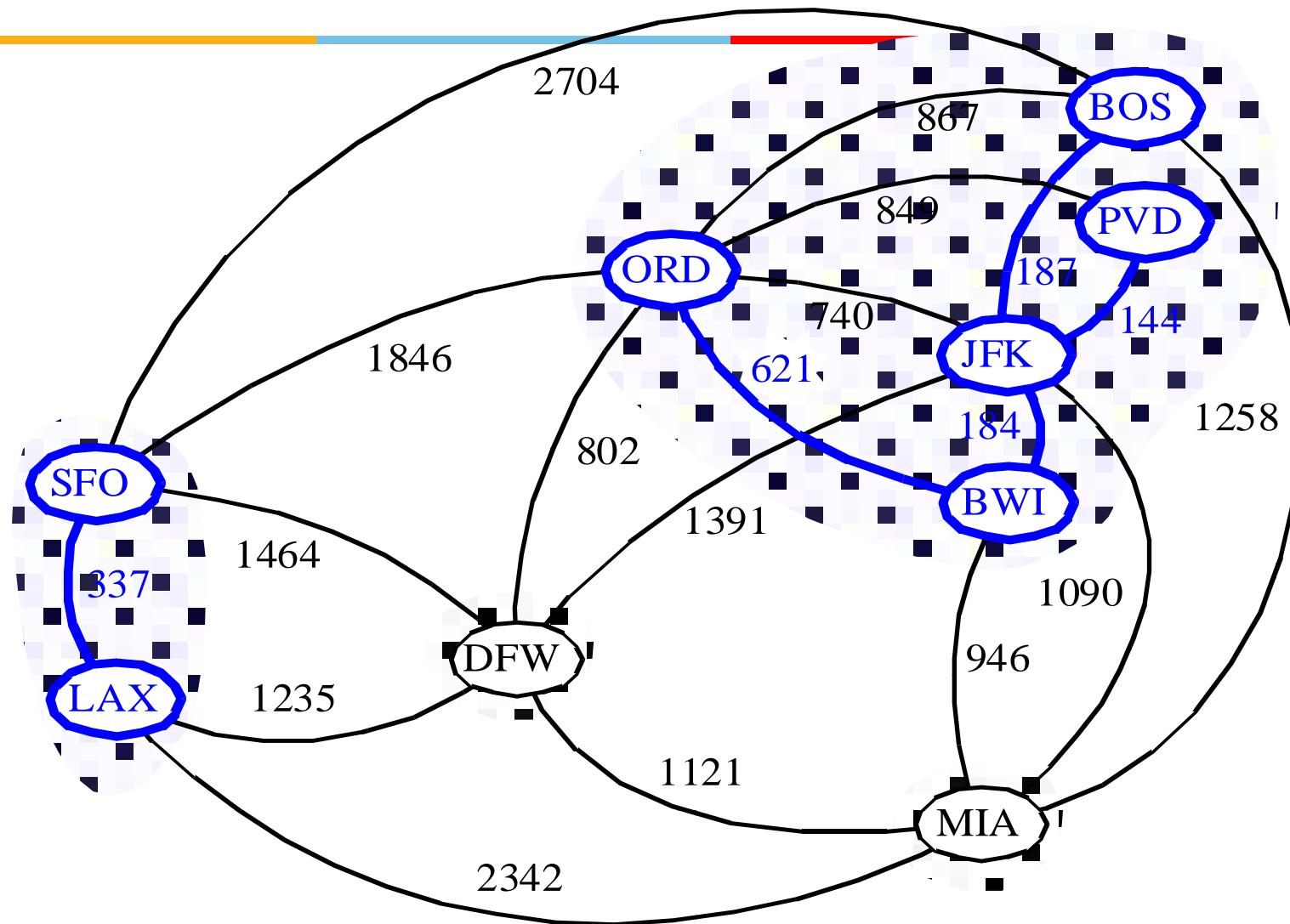
Example



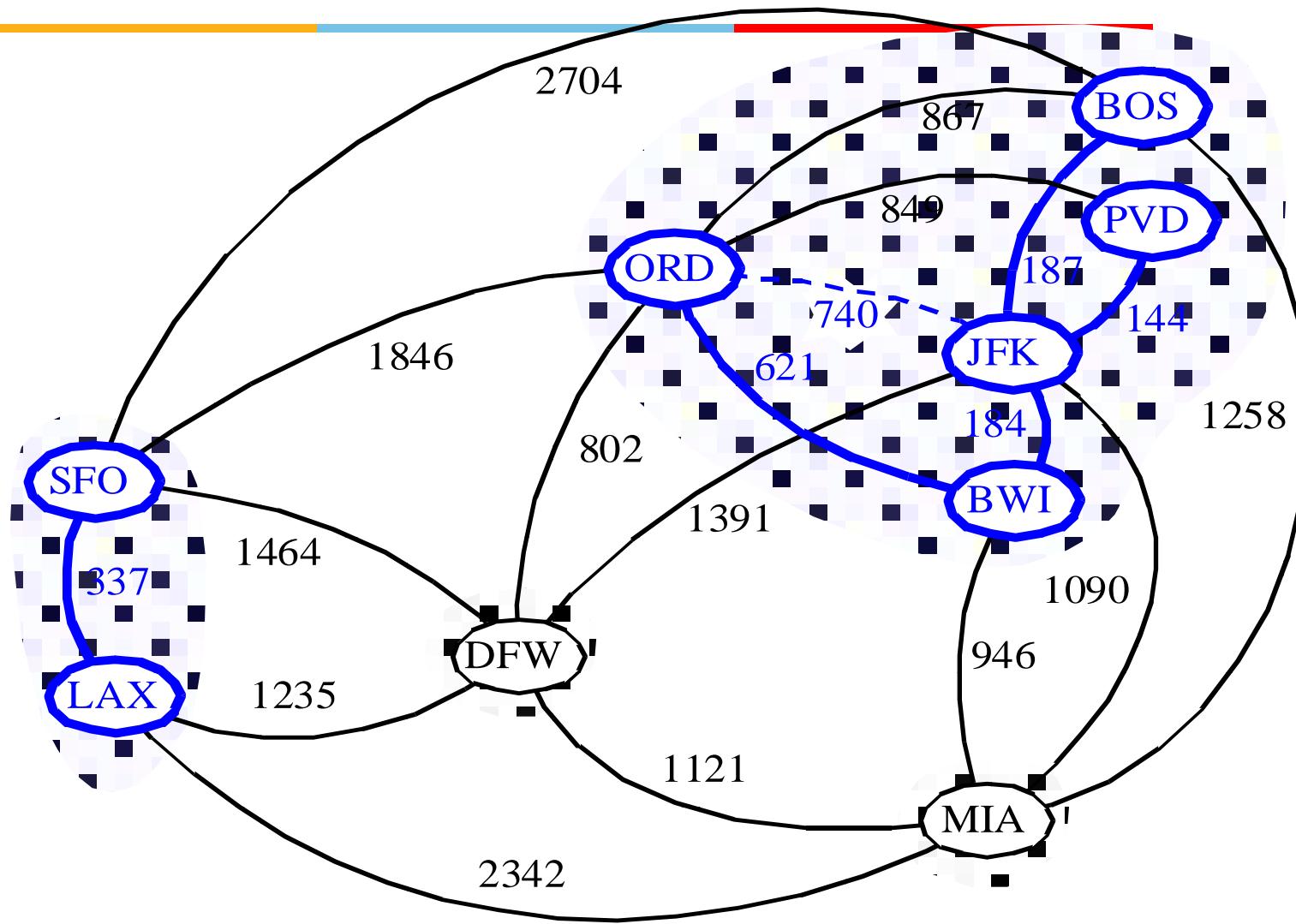
Example



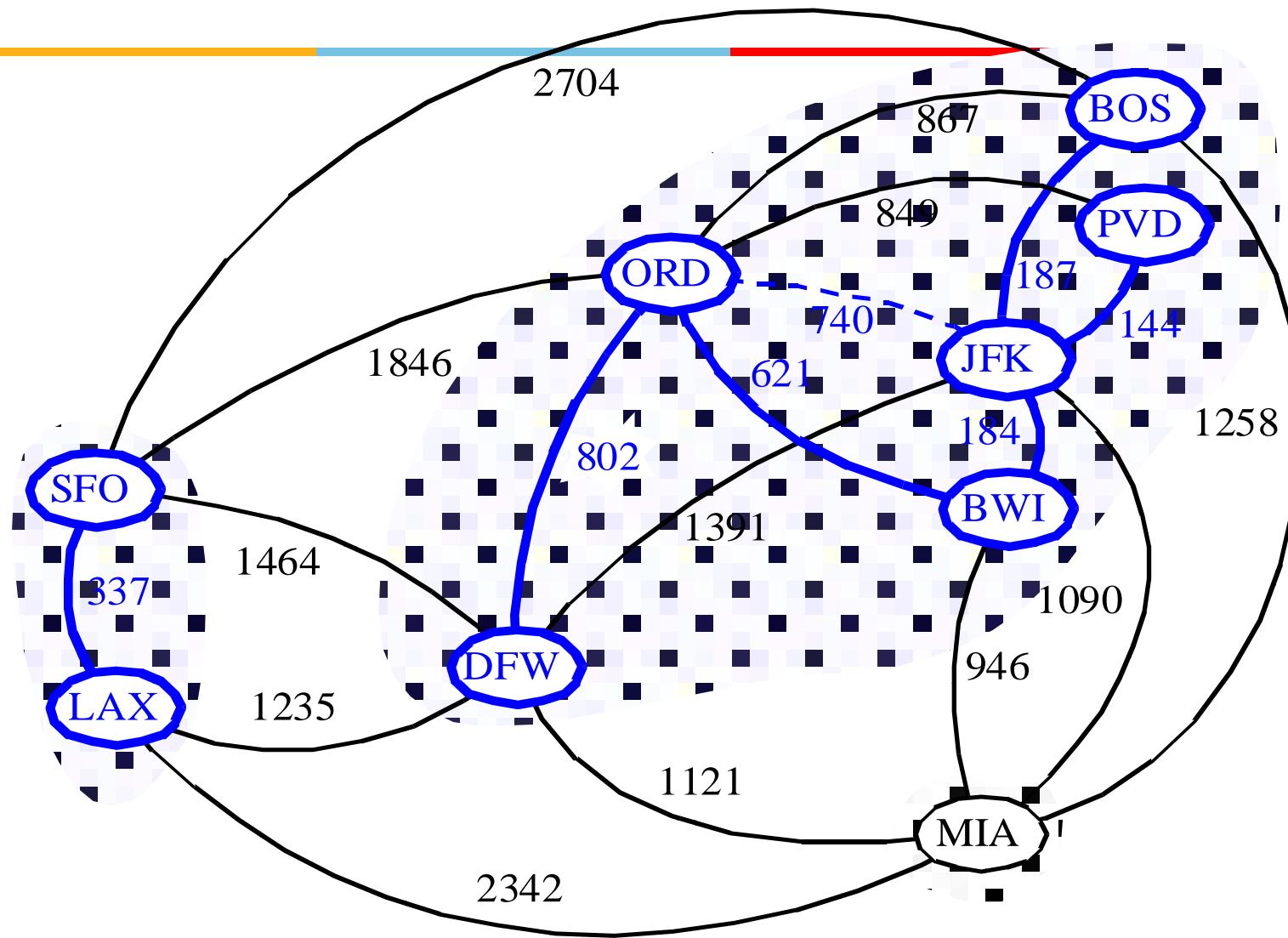
Example



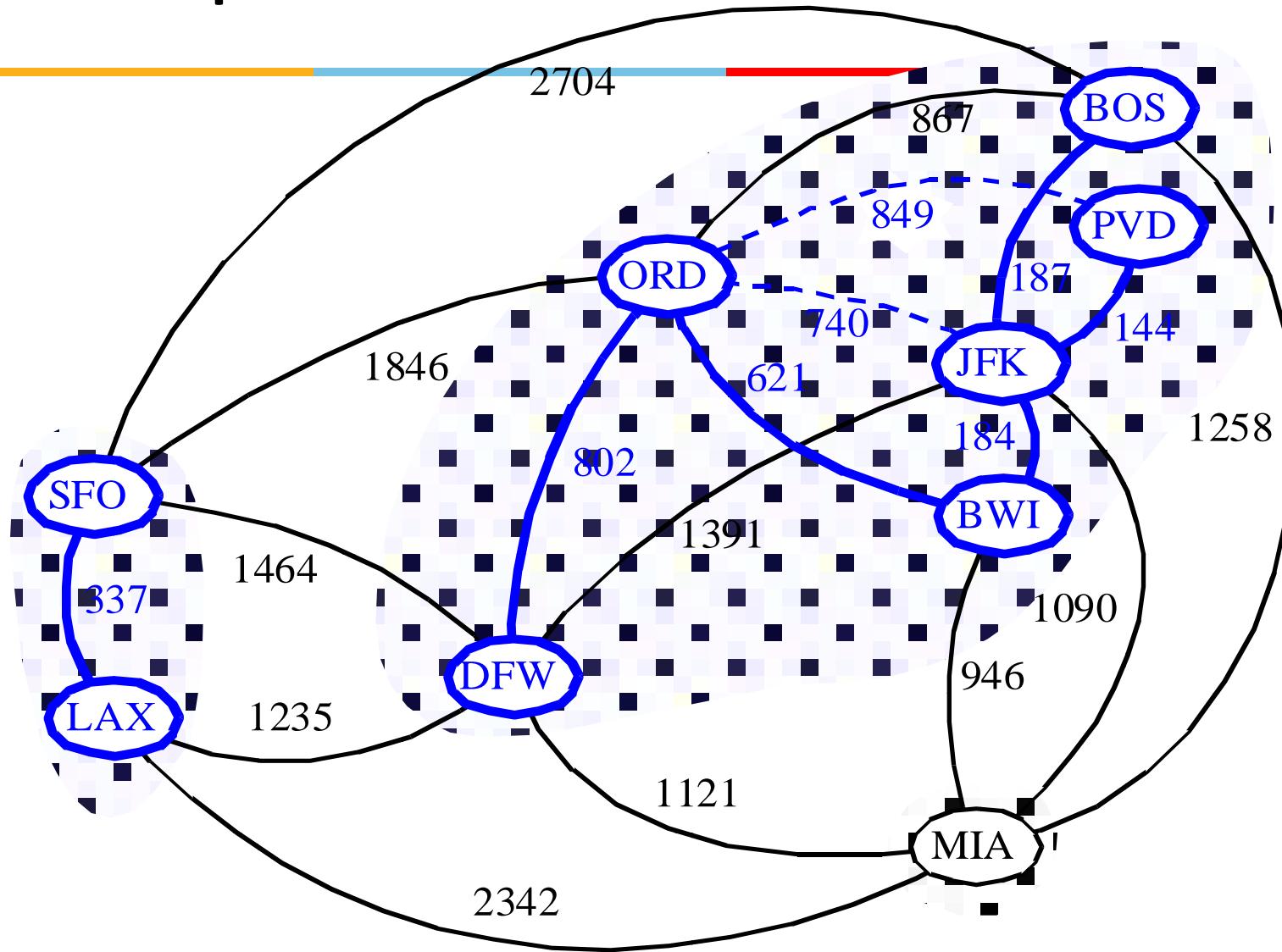
Example



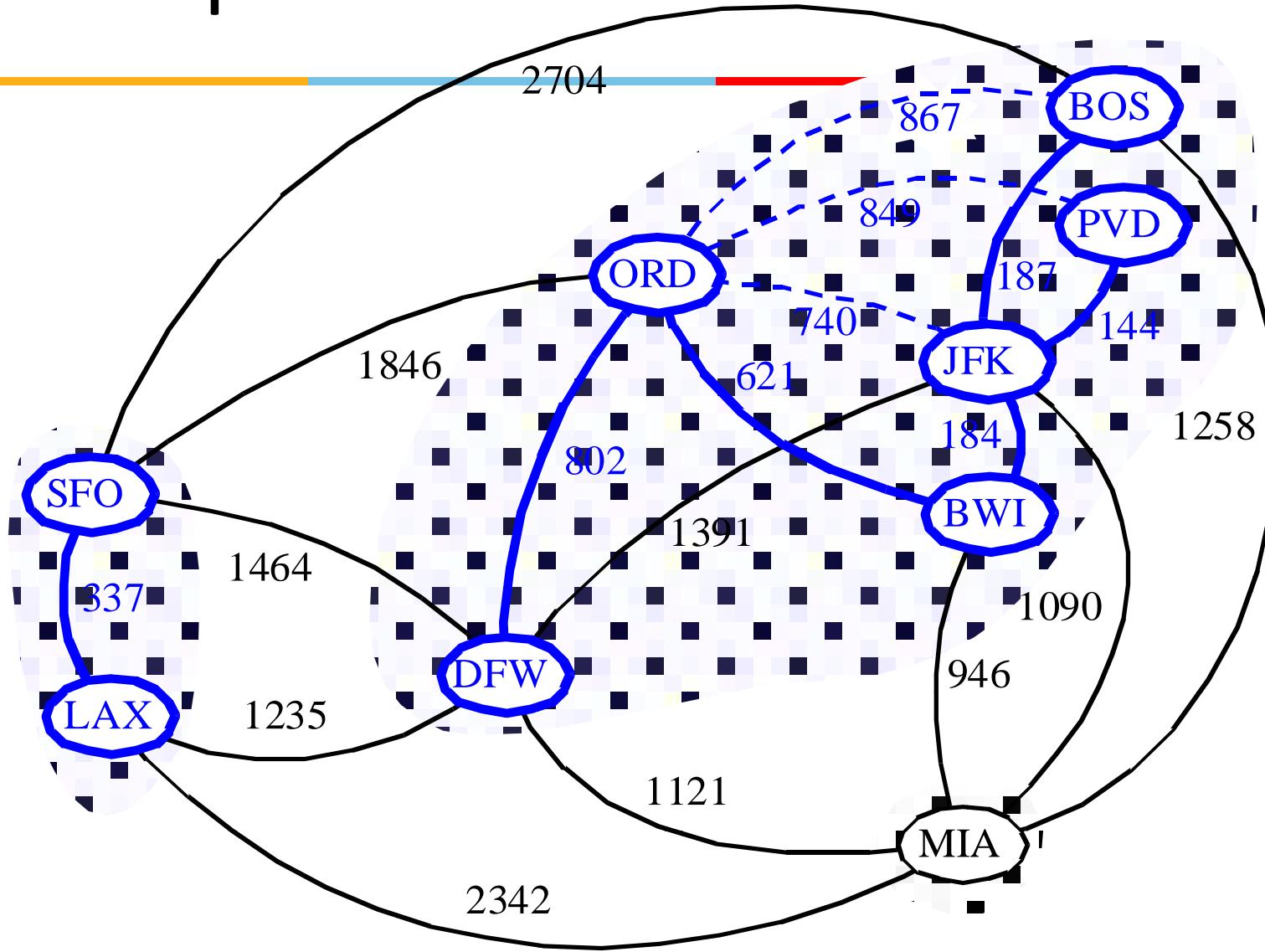
Example



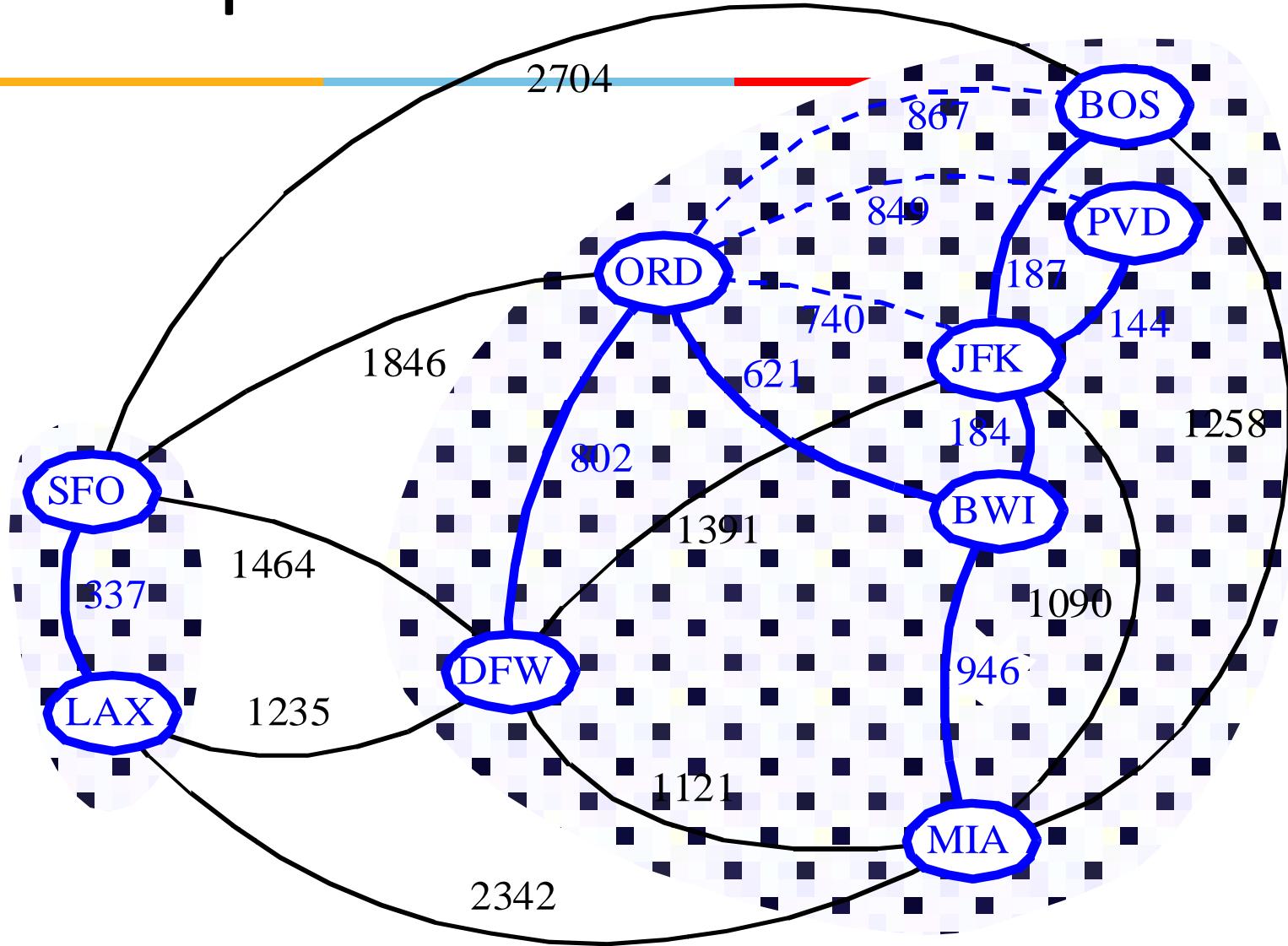
Example



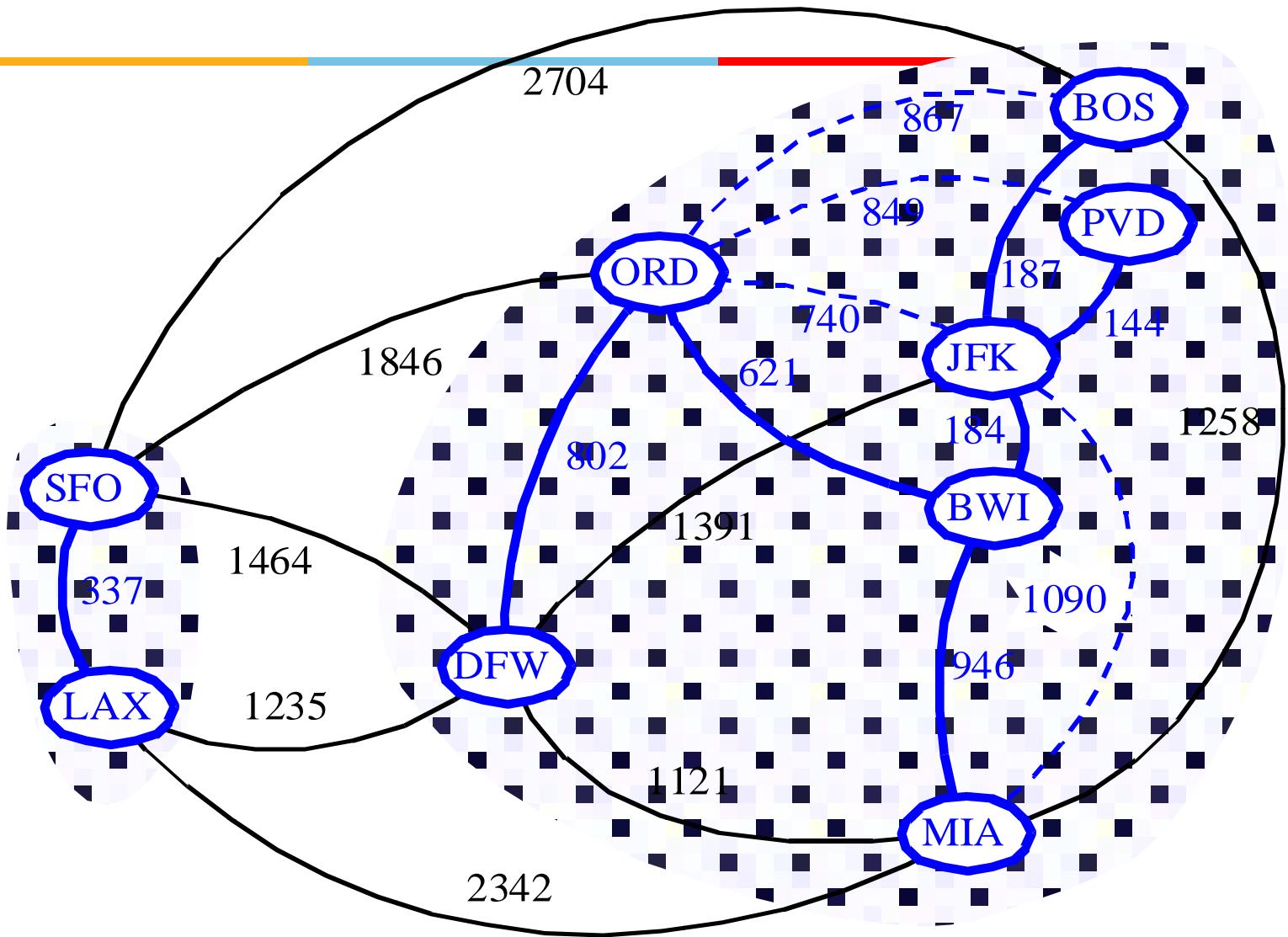
Example



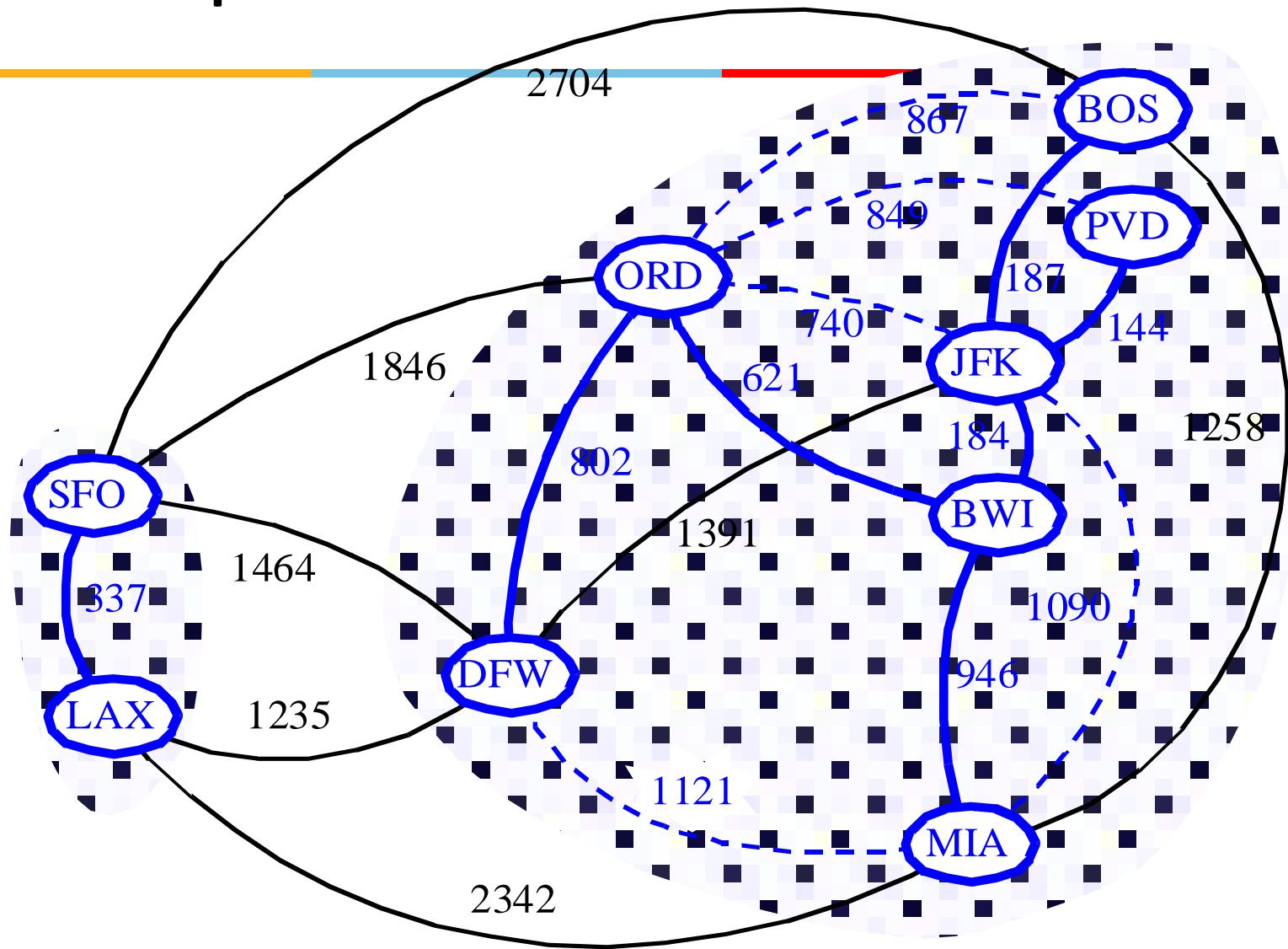
Example



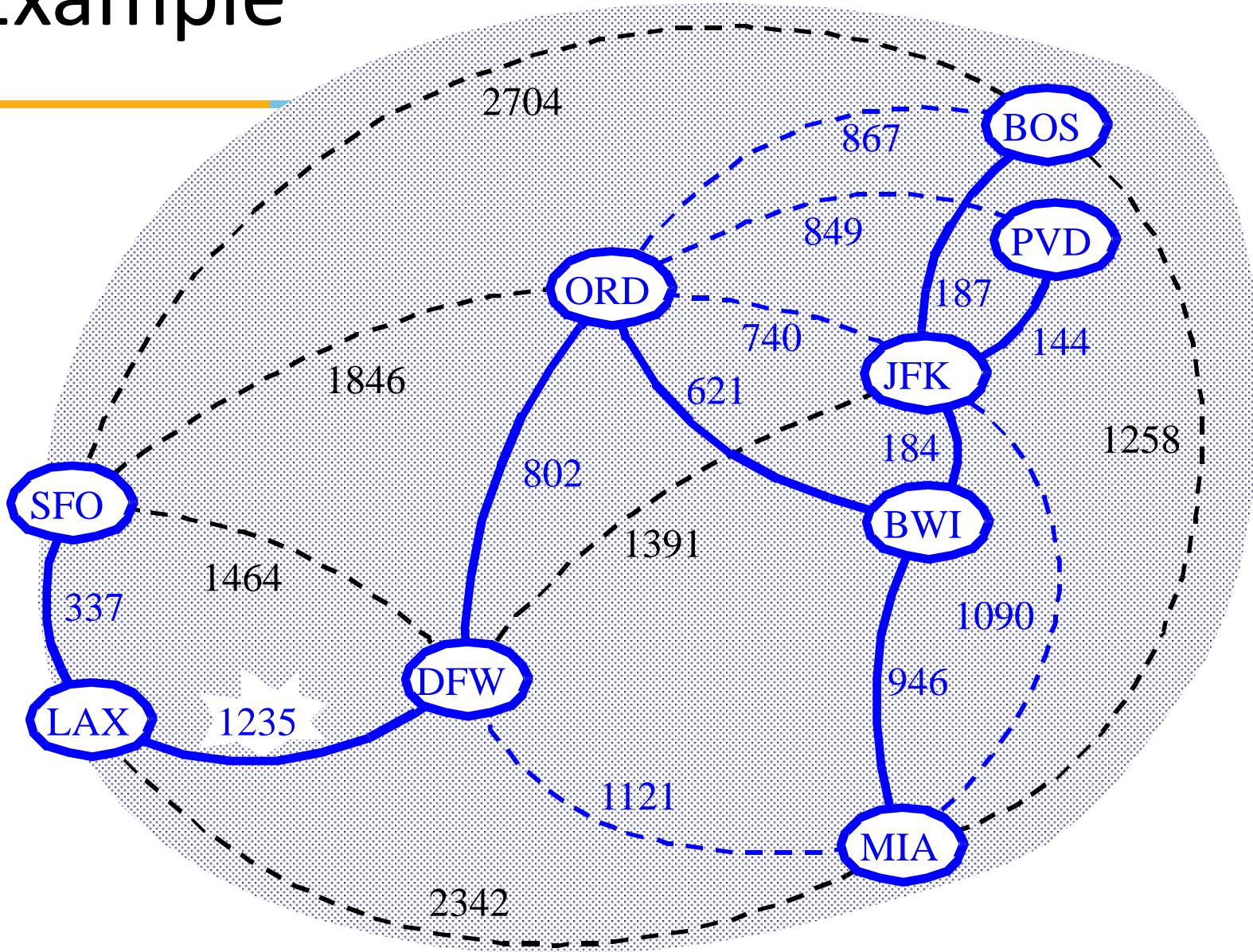
Example



Example



Example





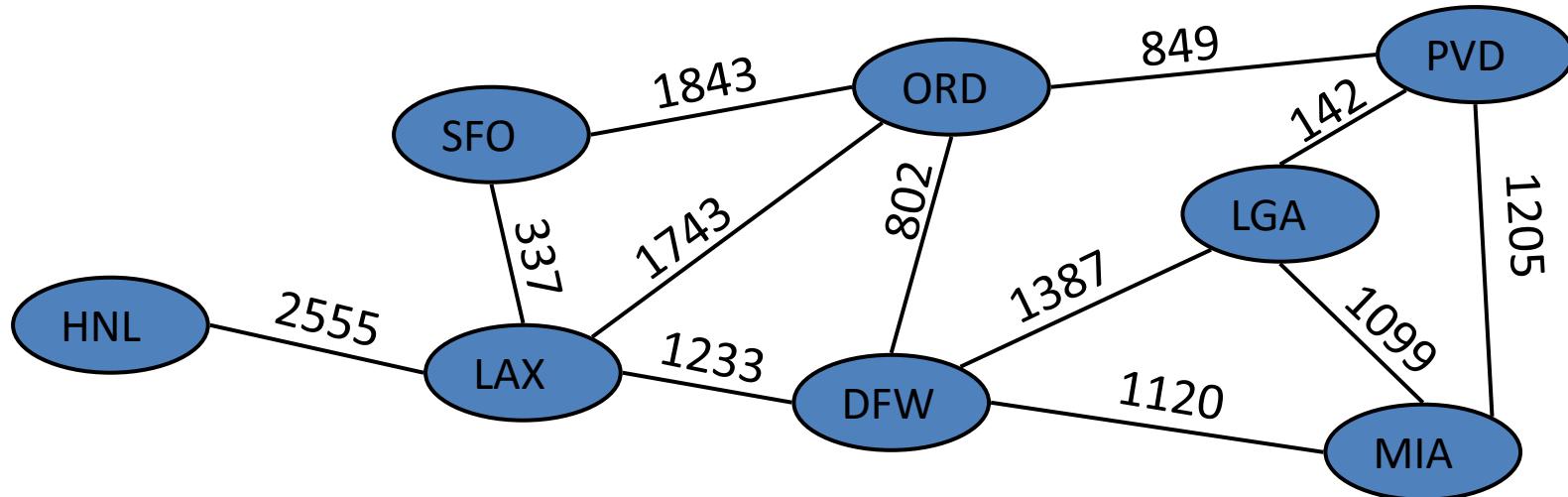
BITS Pilani
Pilani Campus

Course Name : Data Structures & Algorithms

Bharat Deshpande
Computer Science & Information Systems

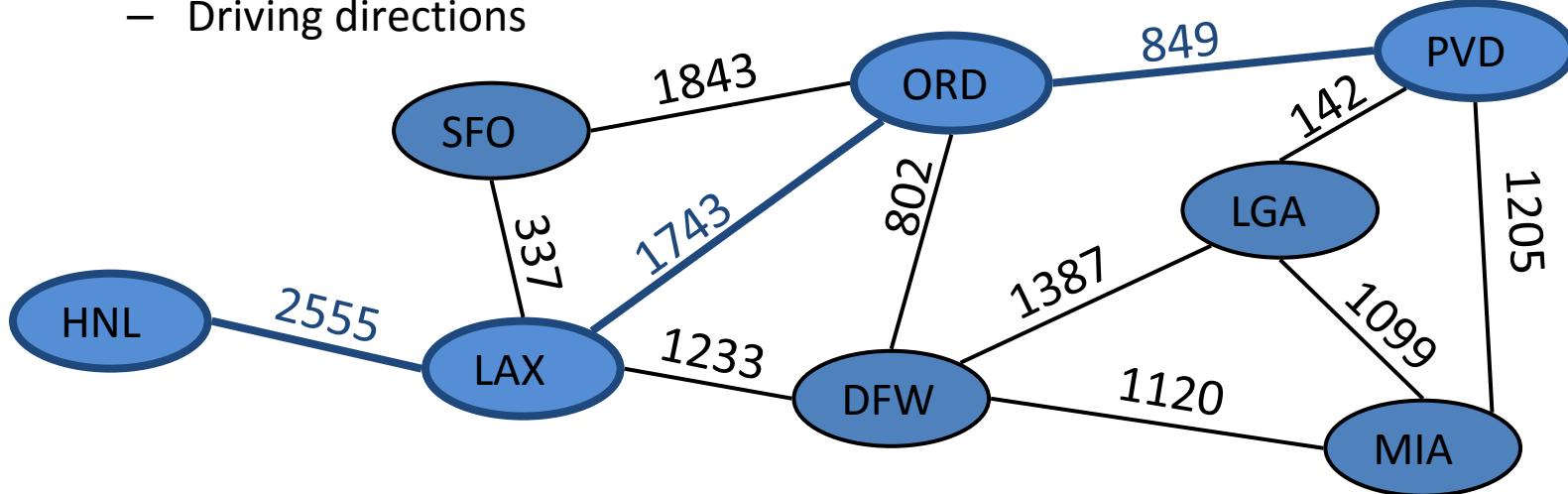
Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the **weight** of the edge
- Edge weights may represent, distances, costs, etc.
- **Example:**
 - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Shortest Paths

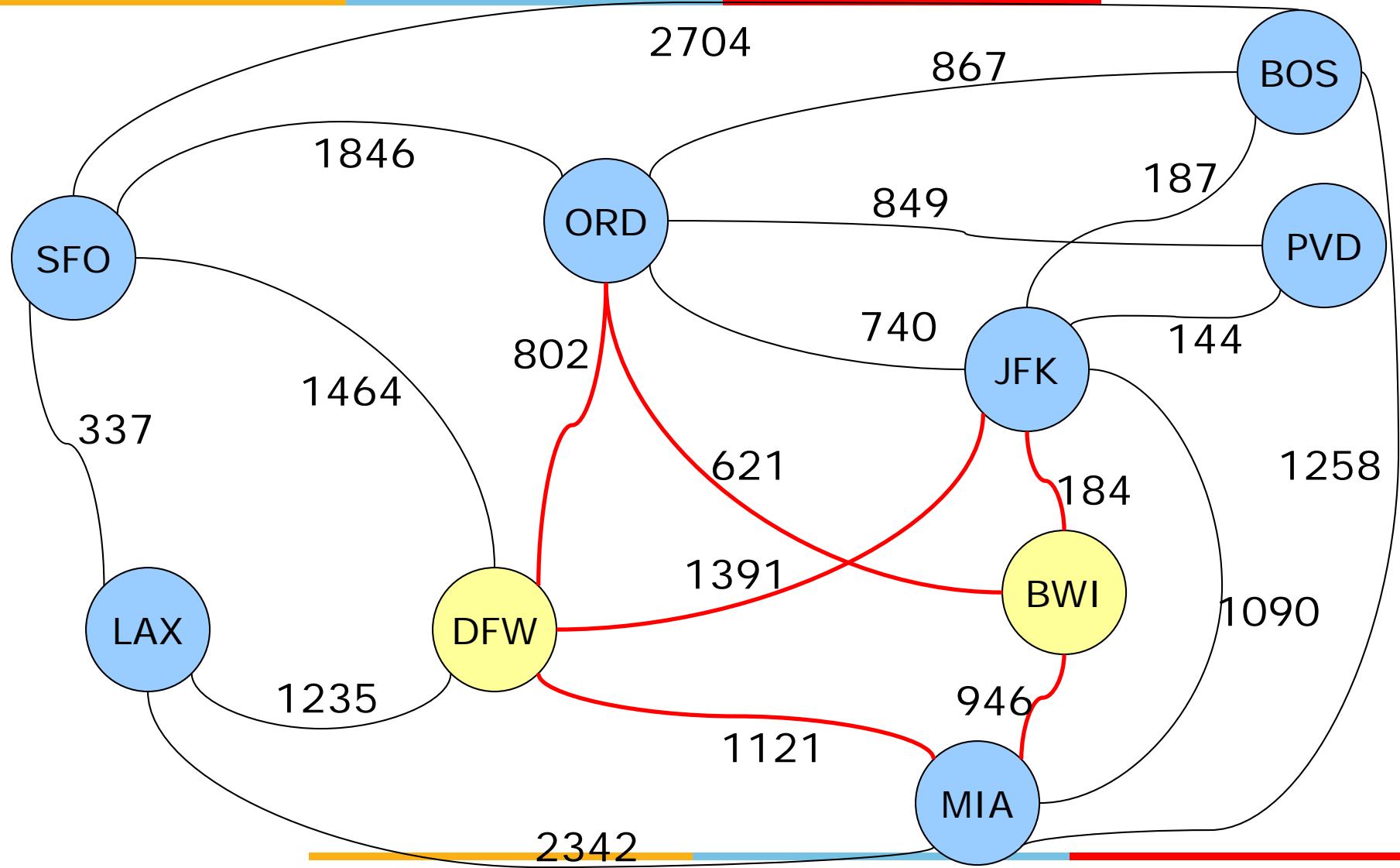
- Given a weighted graph and two vertices u and v , we want to find a path of minimum total weight between u and v .
 - Length of a path is the sum of the weights of its edges.
- Example:**
 - Shortest path between Providence (PVD) and Honolulu (HNL)
- Applications**
 - Internet packet routing
 - Flight reservations
 - Driving directions



Weighted shortest path

- We want to minimize the total mileage.
- Breadth-first search does not work!
 - Minimum number of hops does not mean minimum distance.
 - Consider, for example, BWI-to-DFW in the following map

Three 2-hop routes to DFW



A greedy algorithm

- Assume that every city is infinitely far away.
 - I.e., every city is ∞ miles away from BWI (except BWI, which is 0 miles away).
 - Now perform something similar to breadth-first search, and *optimistically guess that we have found the best path to each city as we encounter it.*
 - If we later discover we are wrong and find a better path to a particular city, then update the distance to that city.

Intuition behind Dijkstra's algorithm



- For our airline-mileage problem, we can start by guessing that every city is ∞ miles away.
 - Mark each city with this guess.
- Find all cities one hop away from BWI, and check whether the mileage is less than what is currently marked for that city.
 - If so, then revise the guess.
- Continue for 2 hops, 3 hops, etc.

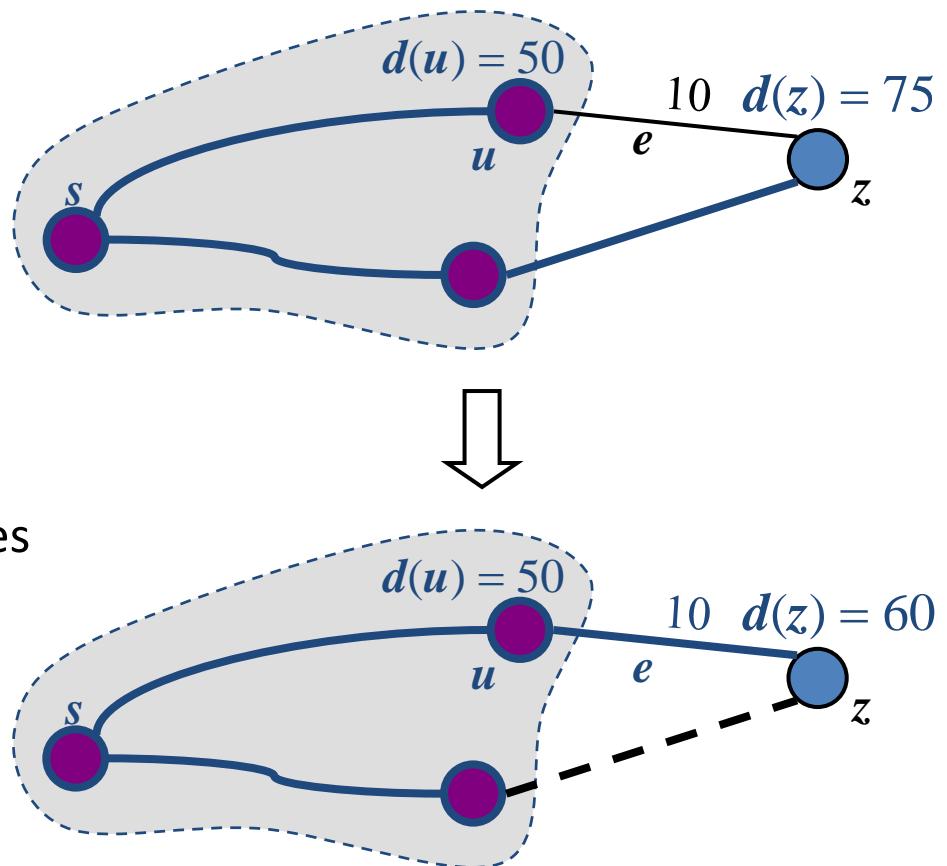
Dijkstra's Algorithm

- **Dijkstra's algorithm** computes the distances of all the vertices from a given start vertex s
- **Assumptions:**
 - the graph is connected
 - the graph is undirected and simple
 - the edge weights are **nonnegative**
- Algorithm grows a **cloud** of vertices, beginning with s and eventually covering all the vertices
- We store with each vertex v a label $d(v)$ representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices
- **At each step**
 - We add to the cloud the vertex u outside the cloud with the smallest distance label, $d(u)$
 - We update the labels of the vertices adjacent to u

Edge Relaxation

- Update procedure is known as **relaxation procedure**.
- Consider an edge $e = (u,z)$ such that
 - u is the vertex most recently added to the cloud
 - z is not in the cloud
- The relaxation of edge e updates distance $d(z)$ as follows:

$$d(z) \leftarrow \min\{d(z), d(u) + \text{weight}(e)\}$$



Dijkstra's algorithm

- Algorithm initialization:
 - Label each node with the distance ∞ , except start node, which is labeled with distance 0.
 - $D[v]$ is the distance label for v .
 - Put all nodes into a priority queue Q , using the distances as labels.

Edge Relaxation

- While Q is not empty do:
 - $u = Q.\text{removeMin}$
 - for each node z one hop away from u do:
 - if $D[u] + \text{miles}(u,z) < D[z]$ then
 - $D[z] = D[u] + \text{miles}(u,z)$
 - change key of z in Q to $D[z]$
- **Note** use of priority queue allows “finished” nodes to be found quickly (in $O(\log N)$ time).
- Running time is $O(m \log n)$

DIJKSTRA's ALGORITHM - WHY IT WORKS

Property 1: Triangle inequality

If $\delta(u,v)$ is the shortest path length between u and v ,
 $\delta(u,v) \leq \delta(u,x) + \delta(x,v)$

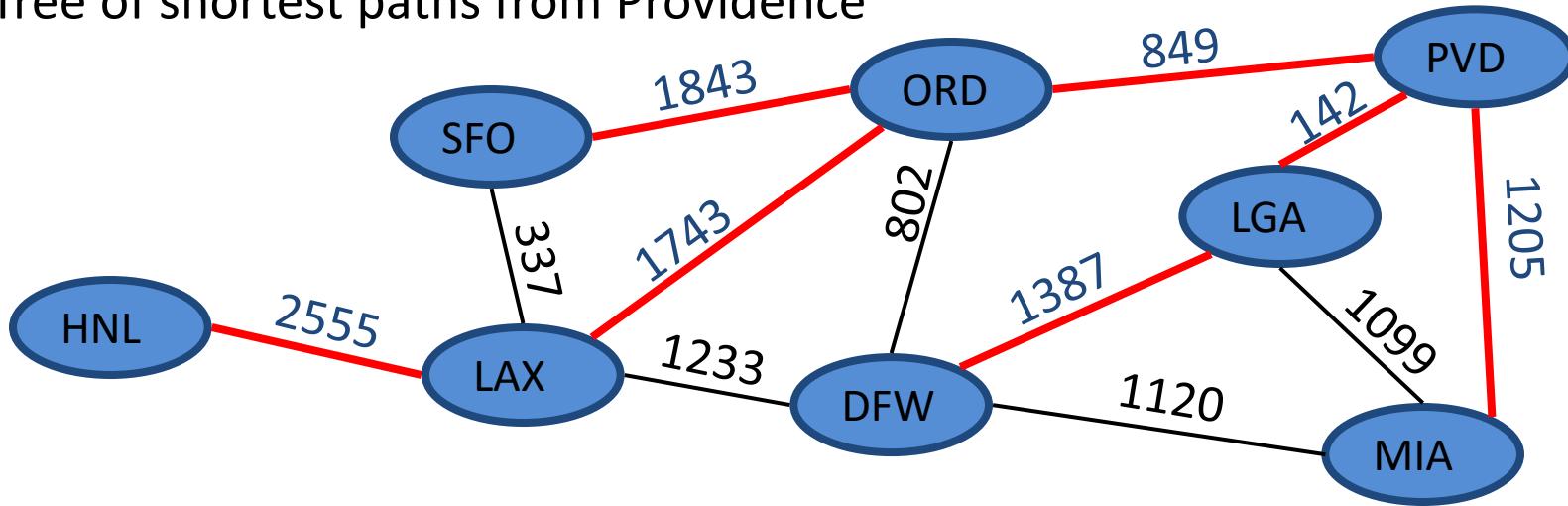
Property 2: A subpath of a shortest path is itself a shortest path

Property 3:

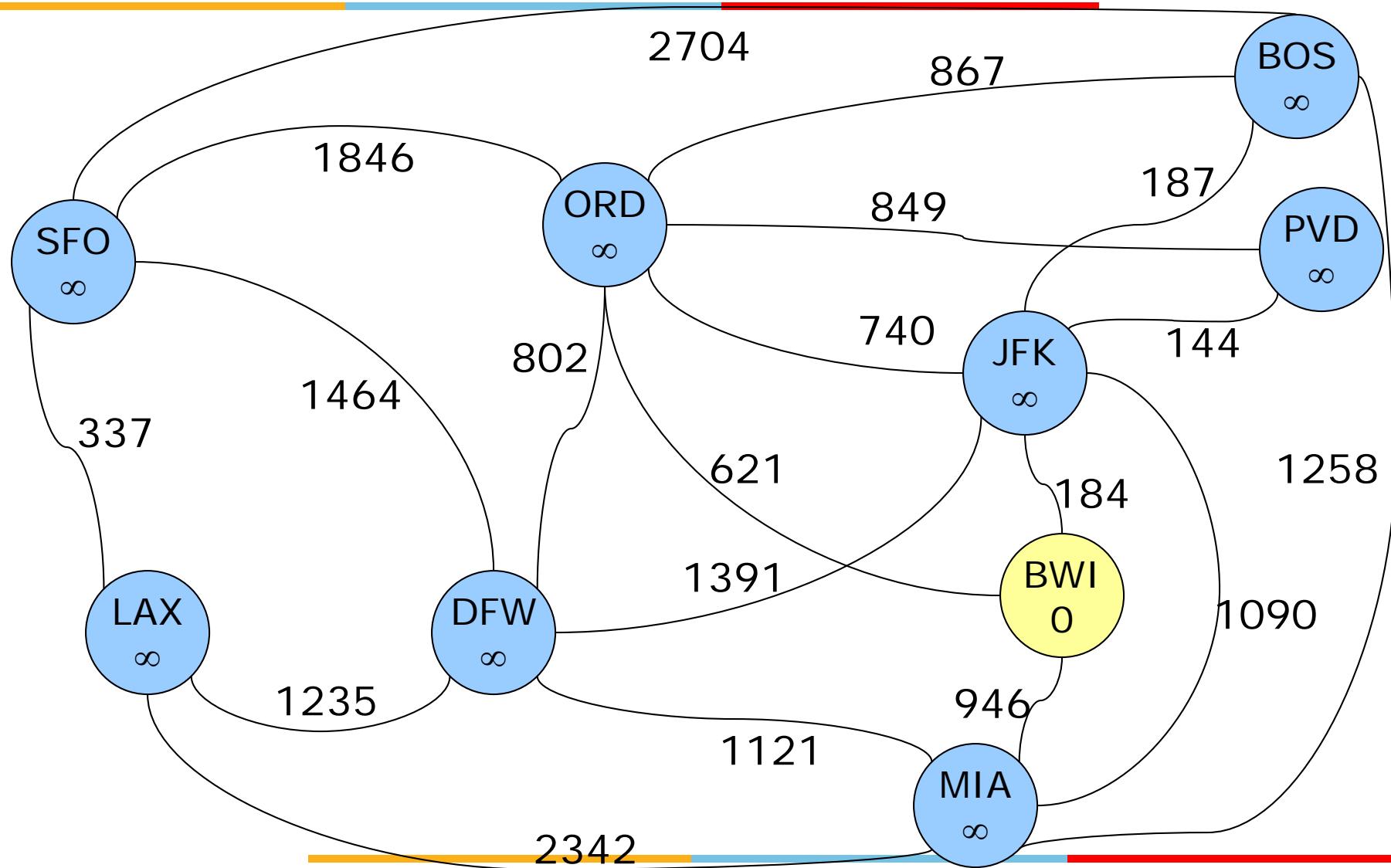
There is a tree of shortest paths from a start vertex to all the other vertices

Example:

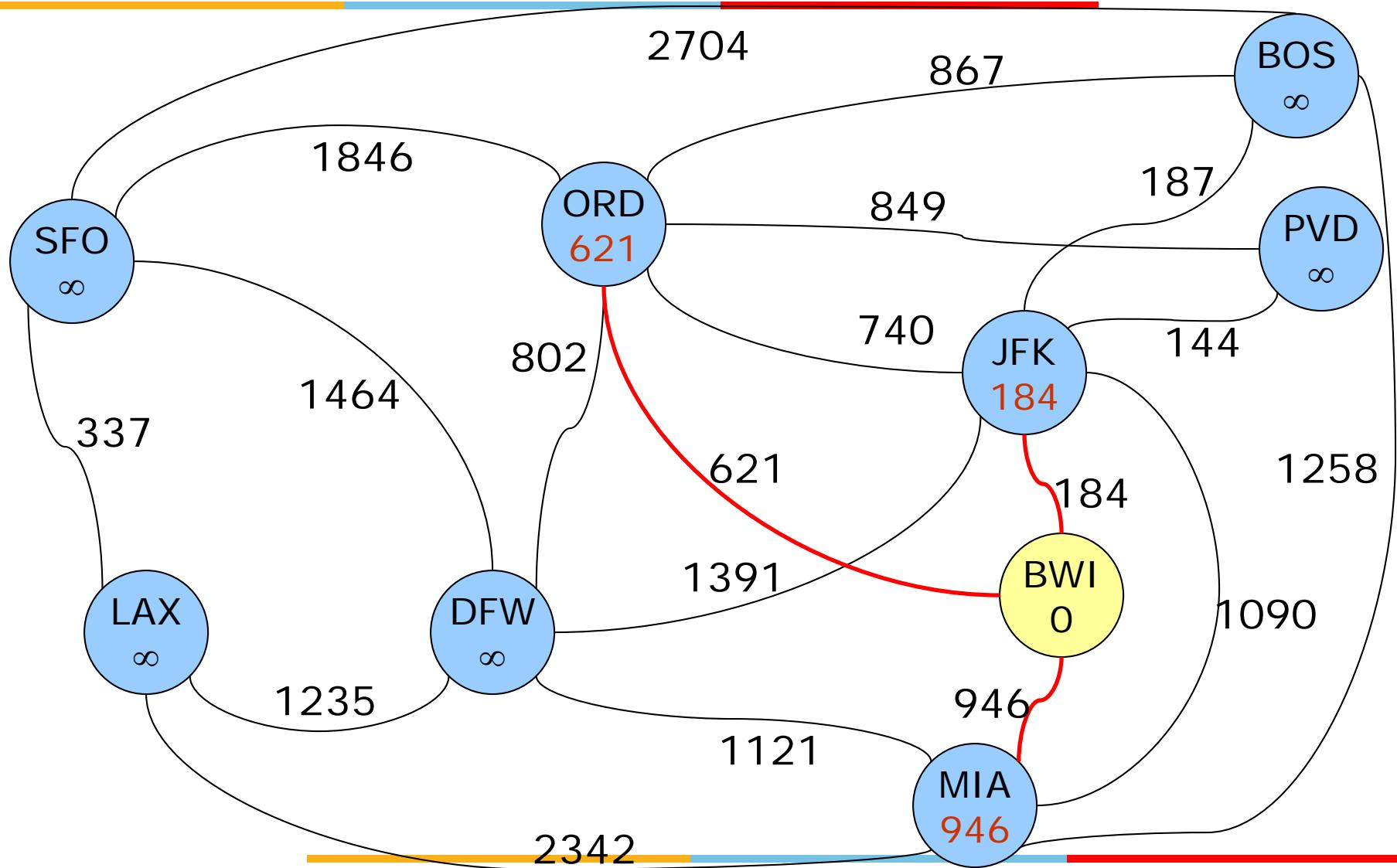
Tree of shortest paths from Providence



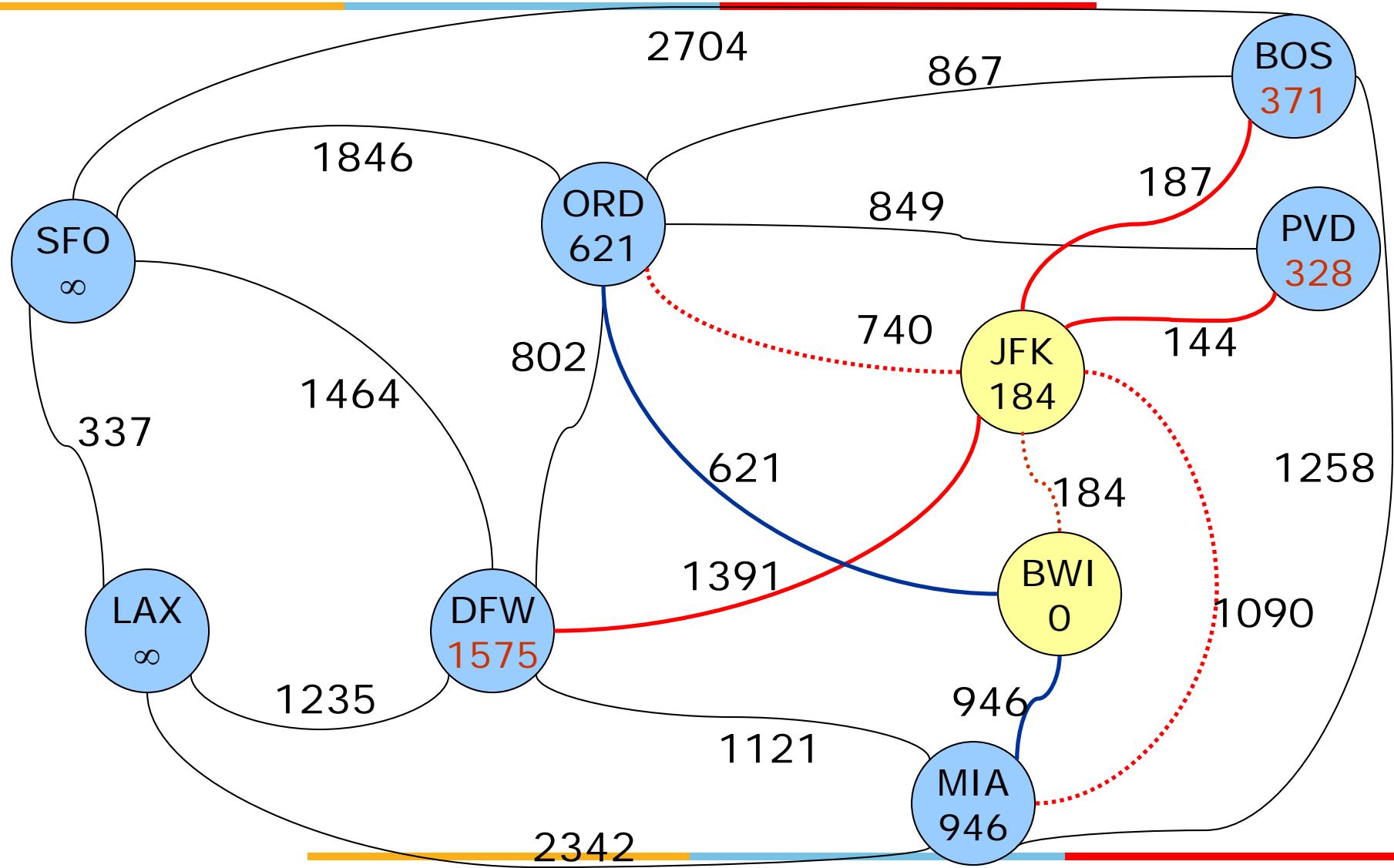
Shortest mileage from BWI



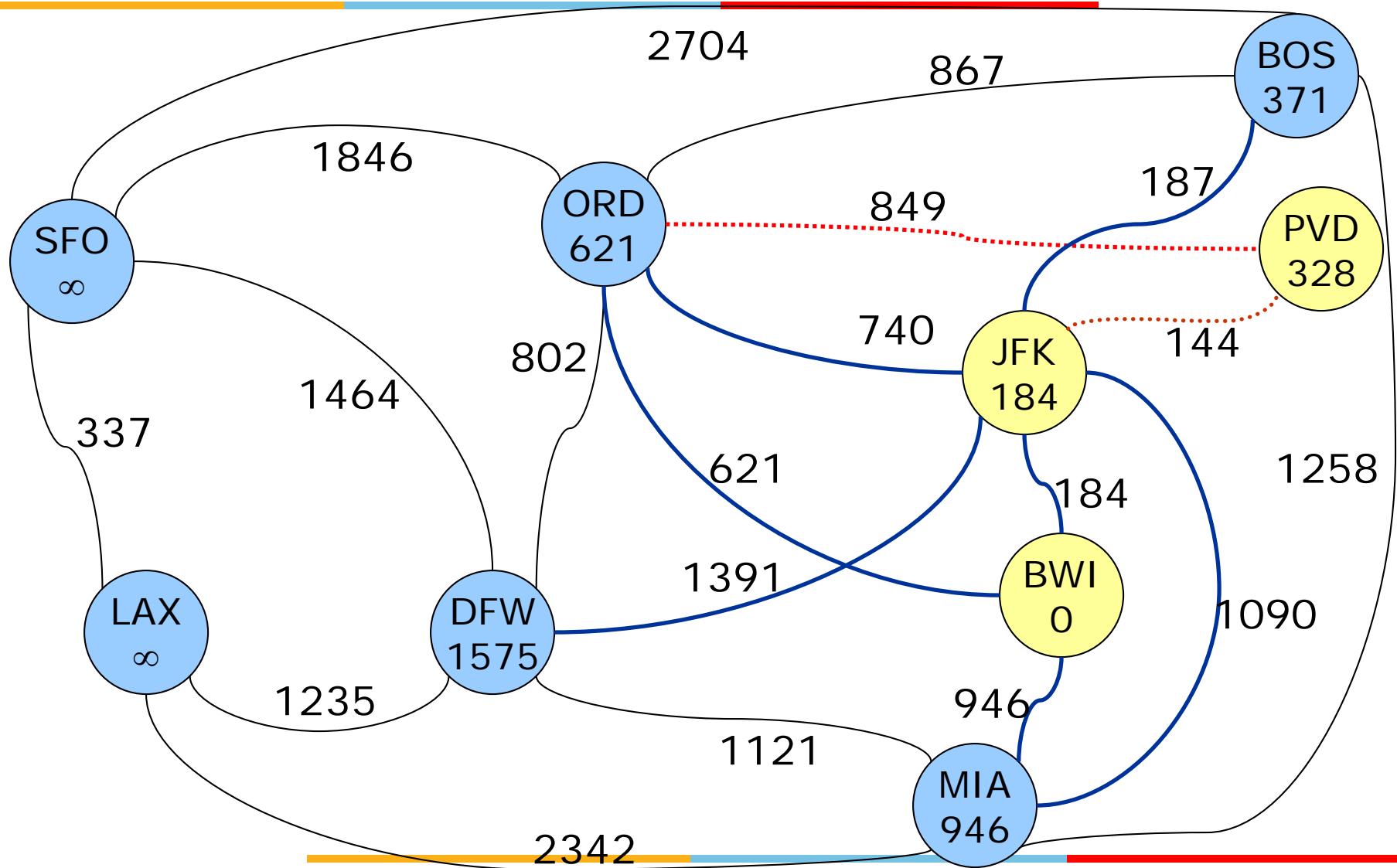
Shortest mileage from BWI



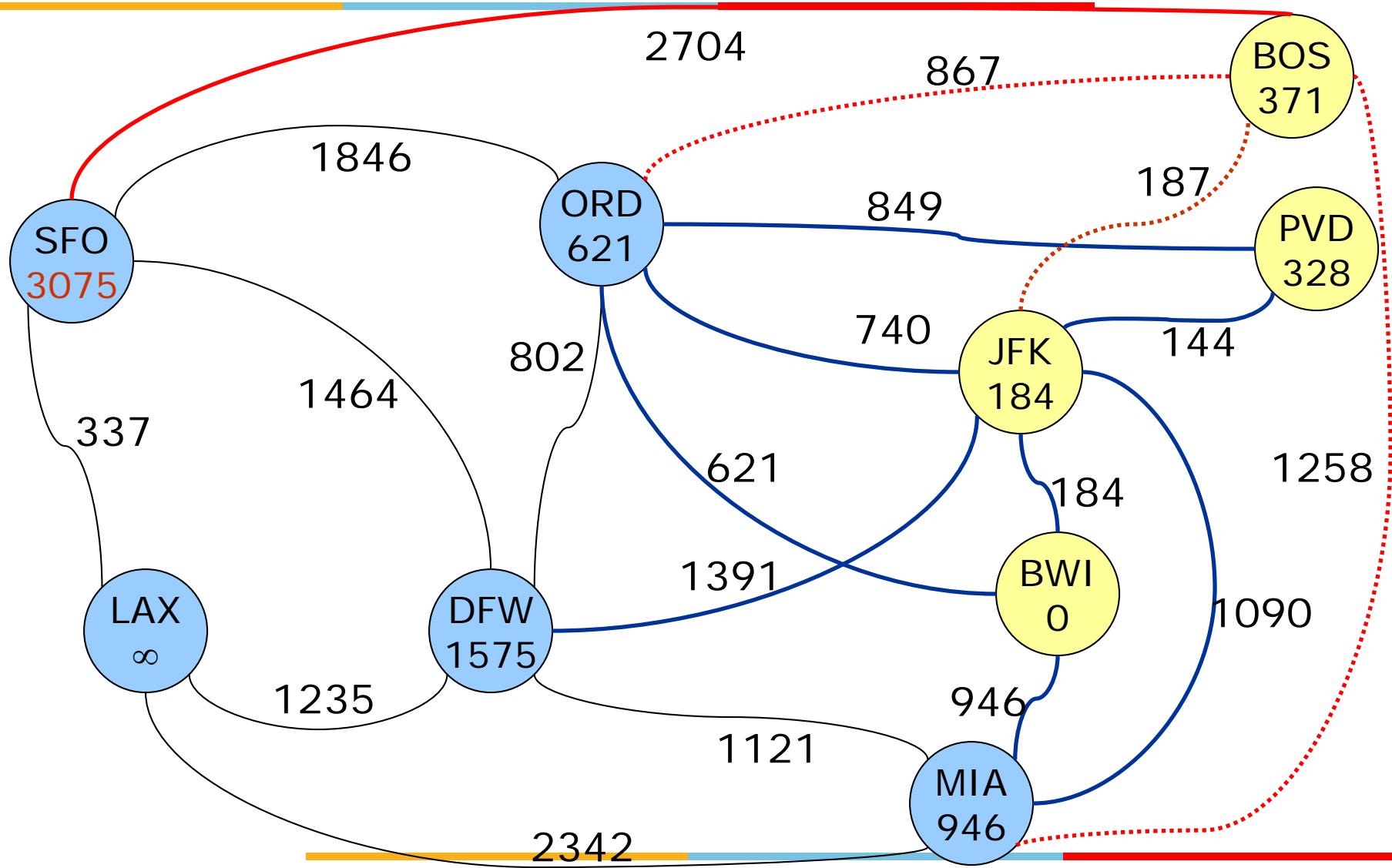
Shortest mileage from BWI



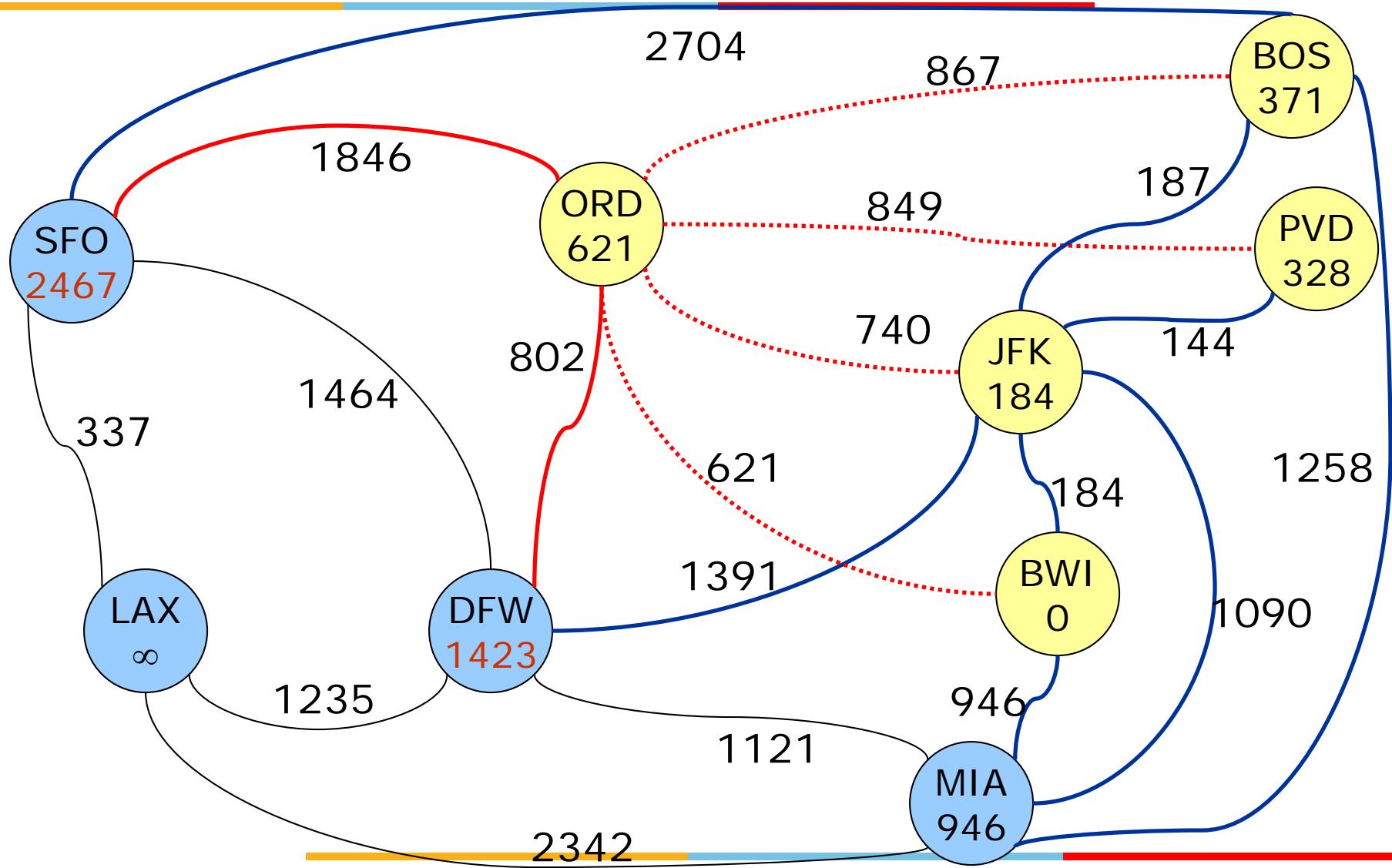
Shortest mileage from BWI



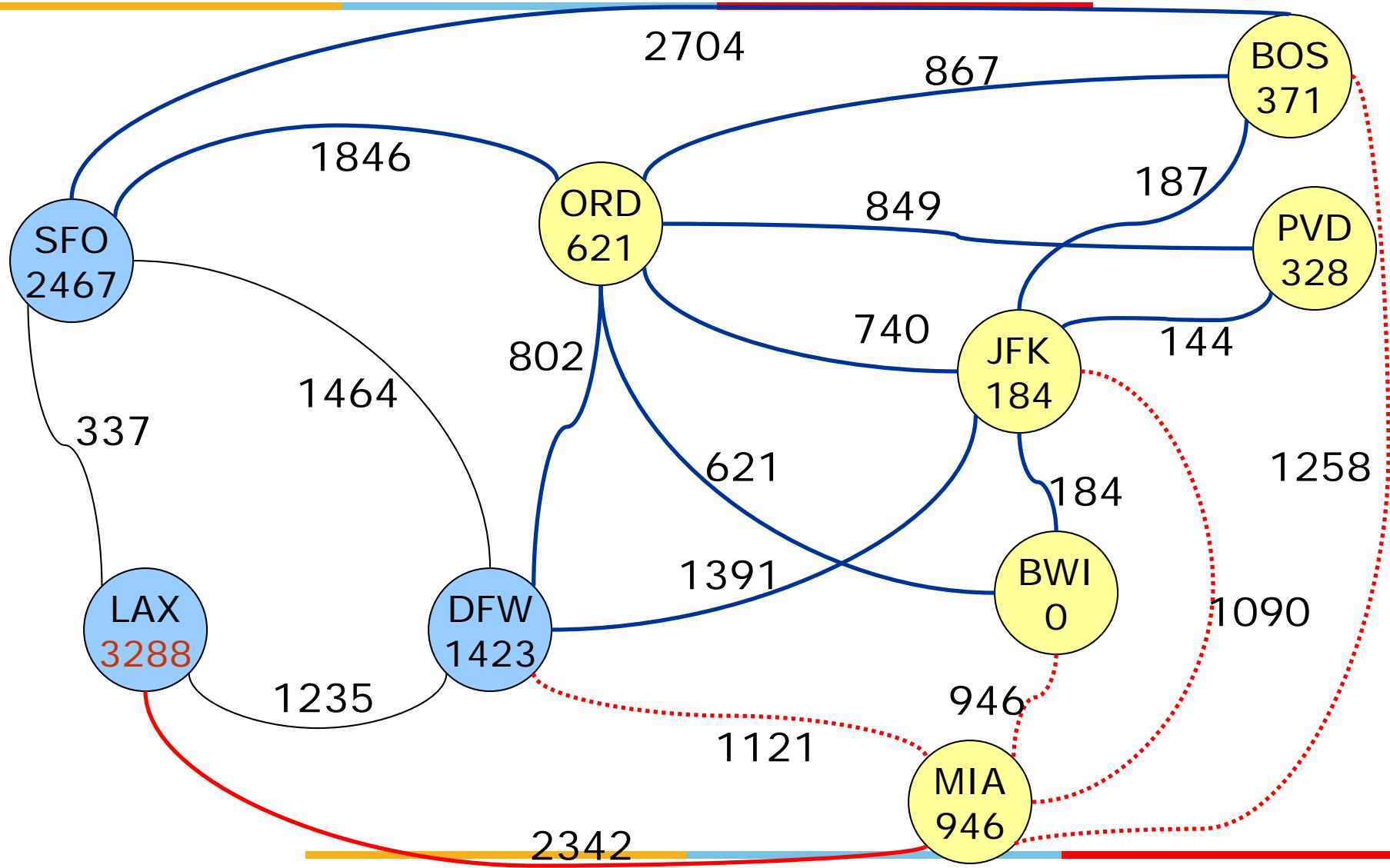
Shortest mileage from BWI



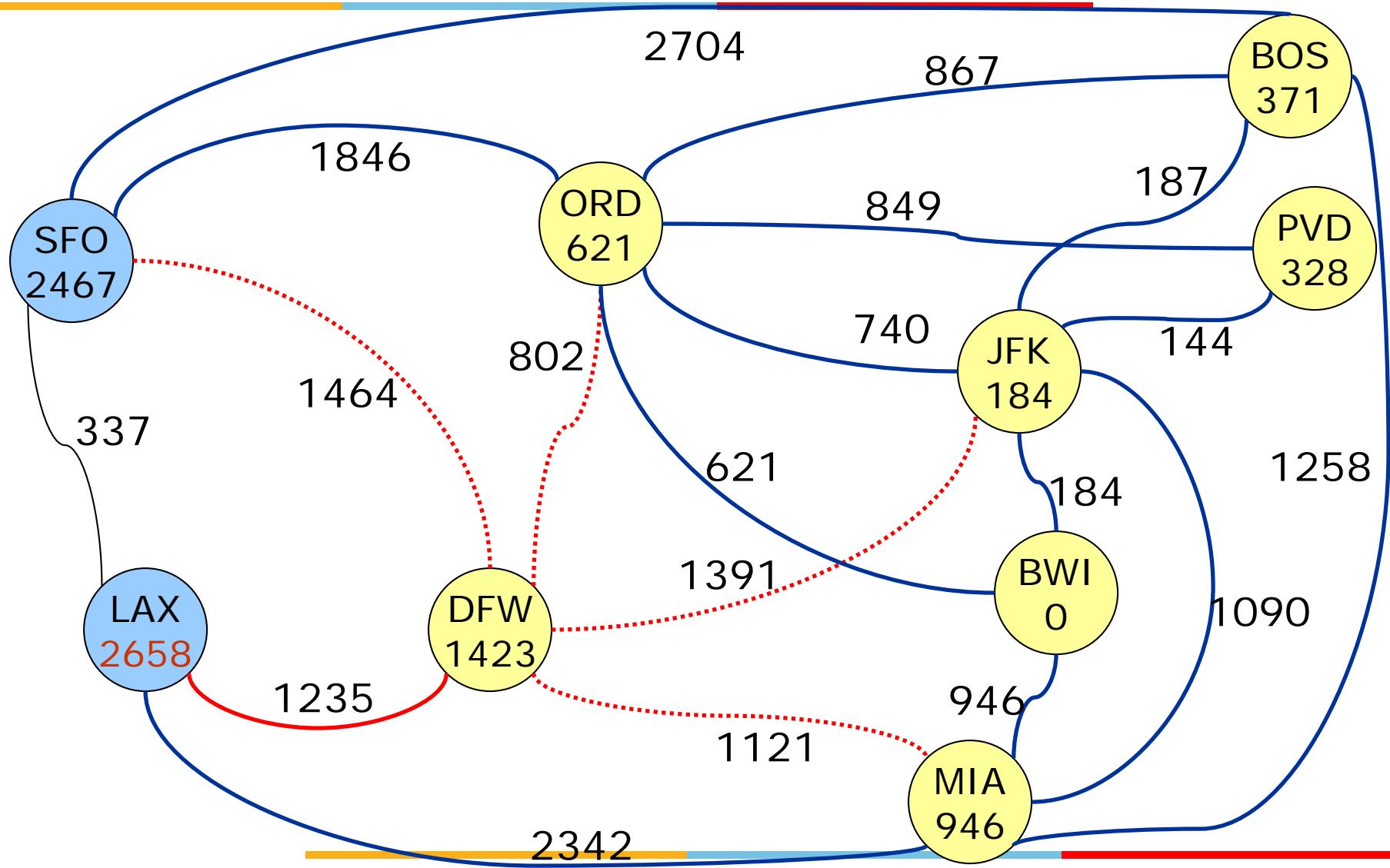
Shortest mileage from BWI



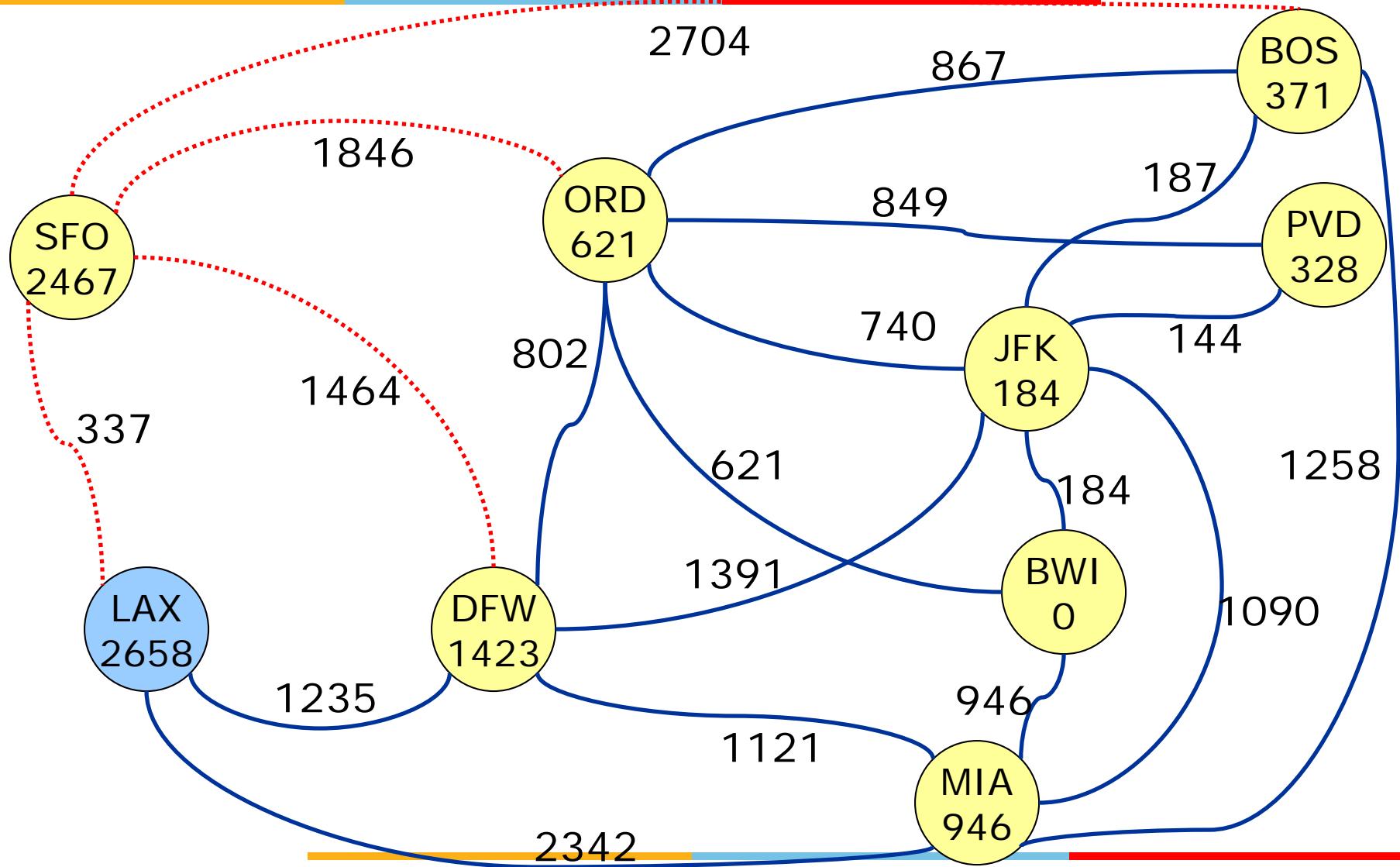
Shortest mileage from BWI



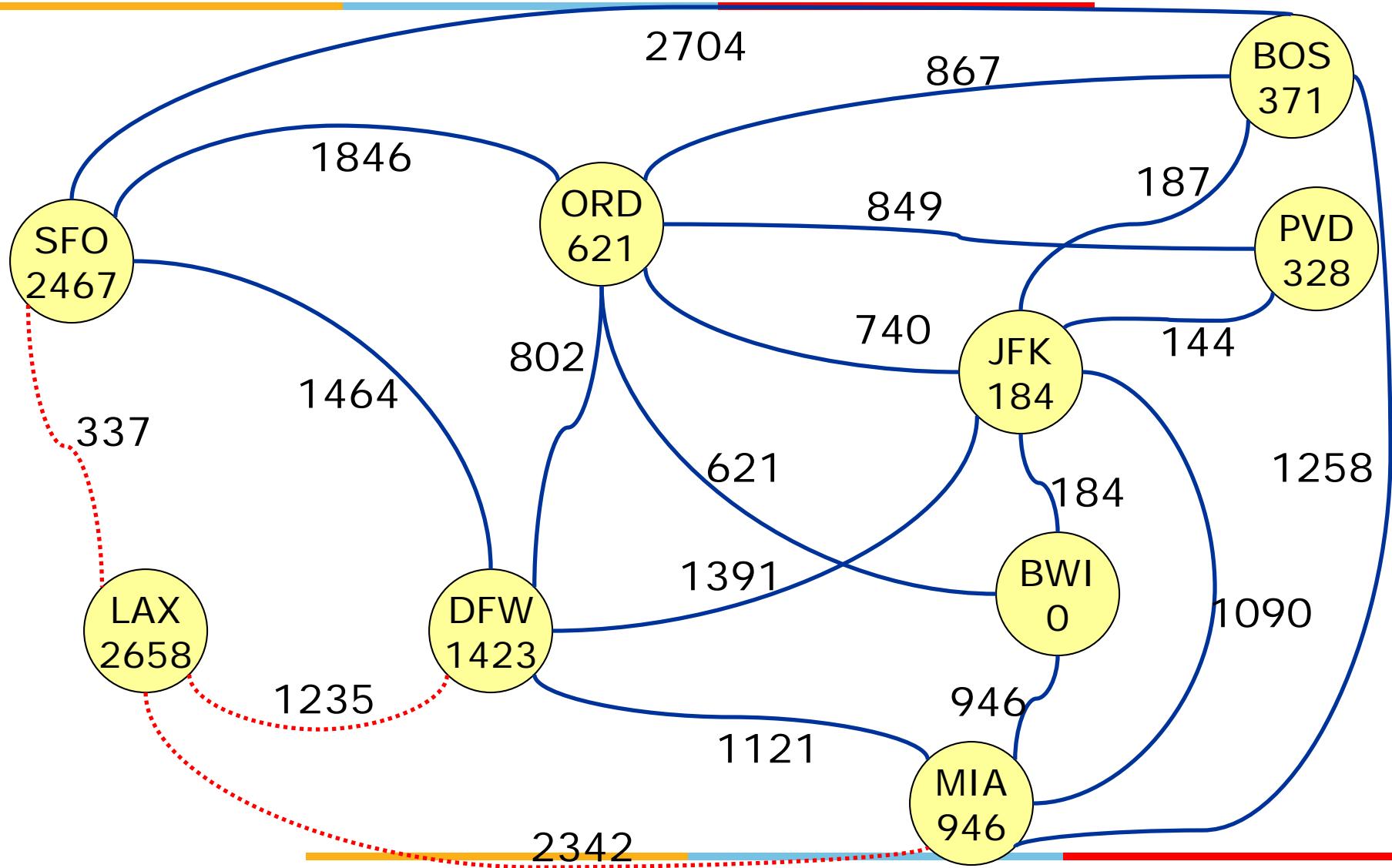
Shortest mileage from BWI



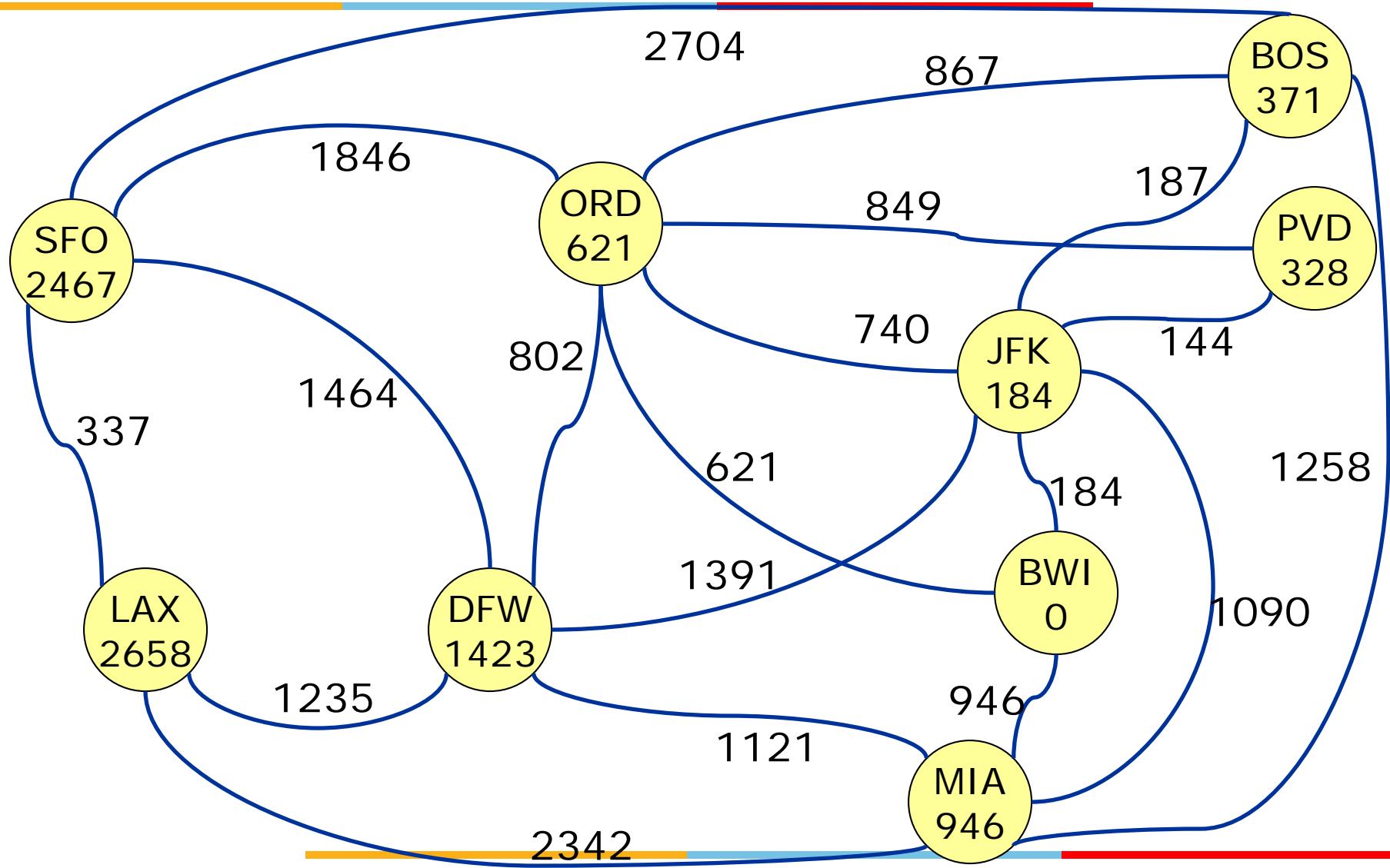
Shortest mileage from BWI



Shortest mileage from BWI

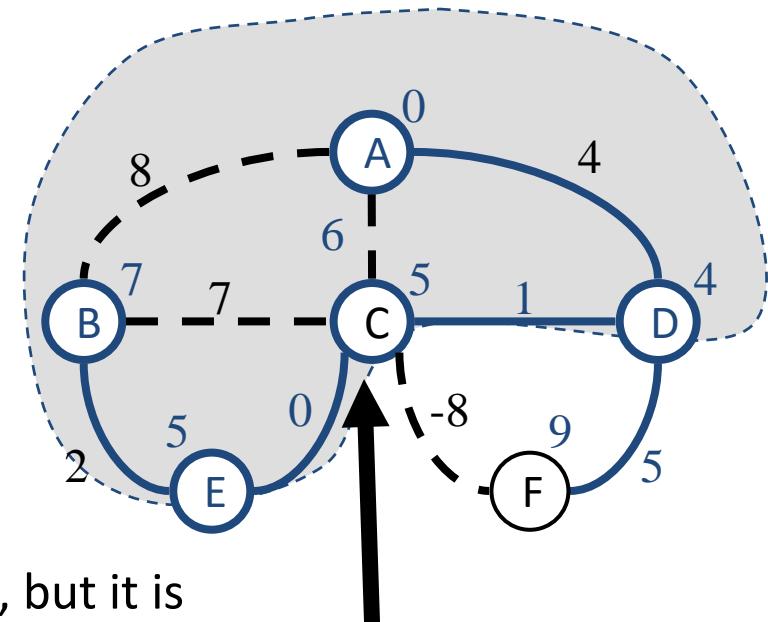


Shortest mileage from BWI



Why It Doesn't Work for Negative-Weight Edges

- Dijkstra's algorithm is based on the greedy method.
It adds vertices by increasing distance.
 - If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



C's true distance is 1, but it is already in the cloud with $d(C)=5$!

Bellman-Ford Algorithm

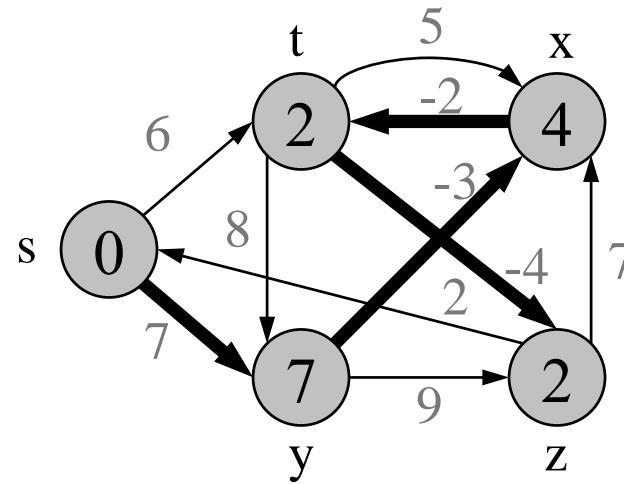
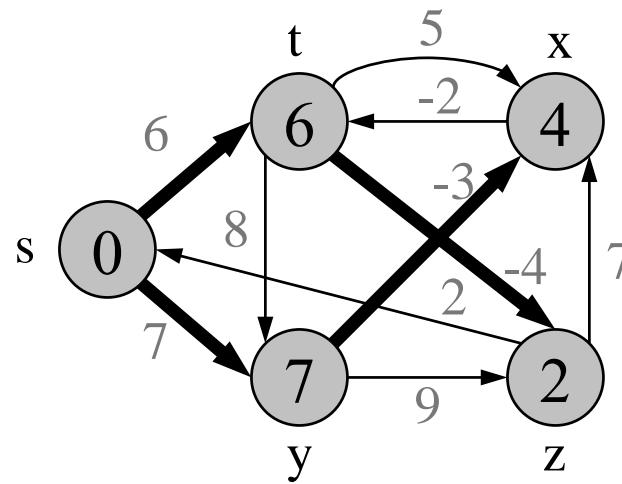
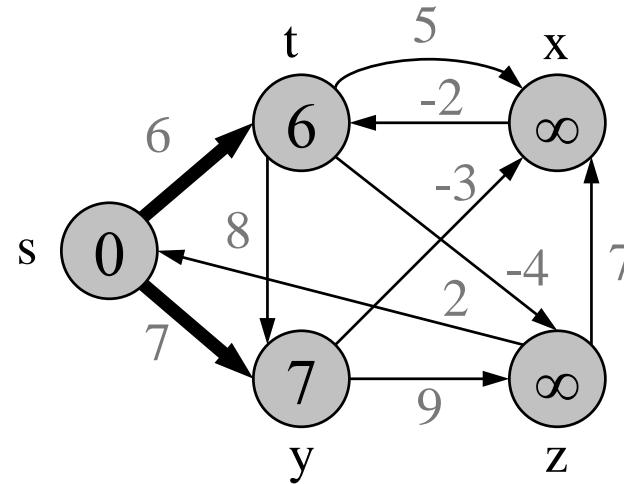
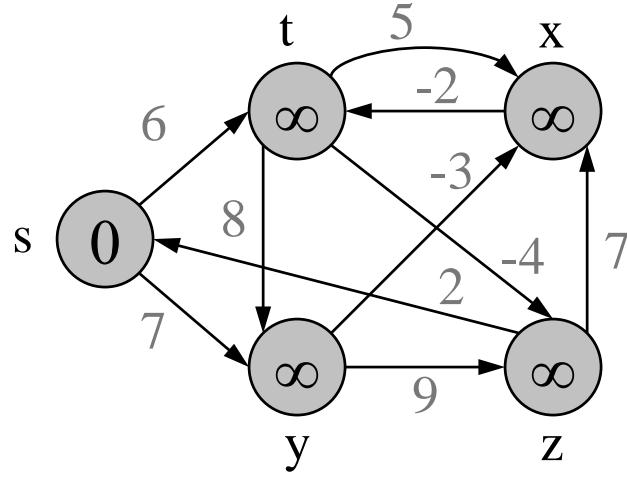
- Works even with negative-weight edges
- Must assume directed edges (for otherwise we would have negative-weight cycles)
- Iteration i finds all shortest paths that use i edges.
- Can be extended to detect a negative-weight cycle if it exists
- Uses edge relaxation as in Dijkstra's but not in conjunction with the greedy method.

Bellman-Ford Algorithm

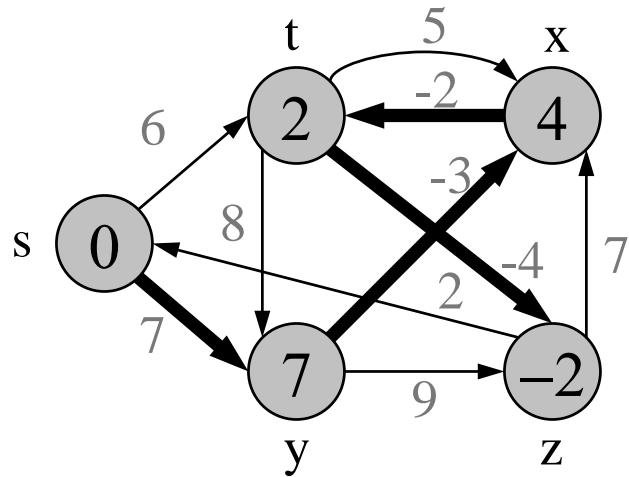
Bellman-Ford(G, s)

```
for each vertex  $u \in V$ 
     $d[u] \leftarrow \infty$ 
    parent[u]  $\leftarrow \text{NIL}$ 
 $d[s] \leftarrow 0$ 
for  $i \leftarrow 1$  to  $|V| - 1$  do
    for each edge  $(u, v)$  outgoing from  $u$  do
        {Perform the relaxation operation on  $(u, v)$ }
        if  $d[u] + w(u, v) < d[v]$  then
             $d[v] \leftarrow d[u] + w(u, v)$ 
if there are no more edges left with potential relaxation operation then
    return the label  $d[u]$  of each vertex  $u$ 
else
    return  $G$  contains a negative weight cycle.
```

Bellman-Ford Example



Bellman-Ford Example



- Bellman-Ford running time:
 - $(|V|-1)|E| + |E| = \Theta(VE)$



BITS Pilani
Pilani Campus

Course Name : **Data Structures & Algorithms**

Bharat Deshpande
Computer Science & Information Systems

Strings

- A **string** is a sequence of characters
- **Examples** of strings:
 - Java program
 - 0101010
 - DNA sequence
 - ordinary text
- An **alphabet** Σ is the set of possible characters for a family of strings
- **Example** of alphabets:
 - {0, 1}
 - {A, C, G, T}

Pattern Matching

- Let P be a string of size m
 - A **substring** $P[i .. j]$ of P is the subsequence of P consisting of the characters with ranks/indices between i and j
 - A prefix of P is a substring of the type $P[0 .. i]$
 - A suffix of P is a substring of the type $P[i .. m - 1]$
- Given strings T (text) and P (pattern), the **pattern matching** problem consists of finding a substring of T equal to P
- **Applications:**
 - text editors
 - search engines
 - biological research

Brute-Force Pattern Matching

- The **brute-force** pattern matching algorithm compares the pattern P with the text T for each possible shift of P relative to T , until either
 - a match is found, or
 - all placements of the pattern have been tried
- Brute-force pattern matching runs in time $O(nm)$
- Example** of worst case:
 - $T = aaa \dots ah$
 - $P = aaah$
 - may occur in images and DNA sequences
 - unlikely in English text

Algorithm *BruteForceMatch(T, P)*

Input text T of size n and pattern P of size m

Output starting index of a substring of T equal to P or -1 if no such substring exists

for $i \leftarrow 0$ to $n - m$

{ test shift i of the pattern }

$j \leftarrow 0$

while $j < m \wedge T[i + j] = P[j]$

$j \leftarrow j + 1$

if $j = m$

return i {match at i }

else

break while loop {mismatch}

return -1 {no match anywhere}

Boyer-Moore Heuristics

- The Boyer-Moore's pattern matching algorithm is based on two heuristics:
Looking-glass heuristic: Compare P with a subsequence of T moving backwards
Character-jump heuristic: When a mismatch occurs at $T[i] = c$
 - If P contains c , shift P to align the last occurrence of c in P with $T[i]$
 - Else, shift P to align $P[0]$ with $T[i + 1]$
-

Last-Occurrence Function

- Boyer-Moore's algorithm preprocesses the pattern P and the alphabet Σ to build the last-occurrence function L mapping Σ to integers, where $L(c)$ is defined as
 - the largest index i such that $P[i] = c$ or
 - -1 if no such index exists
- Example:
 - $\Sigma = \{a, b, c, d\}$
 - $P = abacab$

c	a	b	c	d
$L(c)$	4	5	3	-1

- The last-occurrence function can be represented by an array indexed by the numeric codes of the characters
- The last-occurrence function can be computed in time $O(m + s)$, where m is the size of P and s is the size of Σ

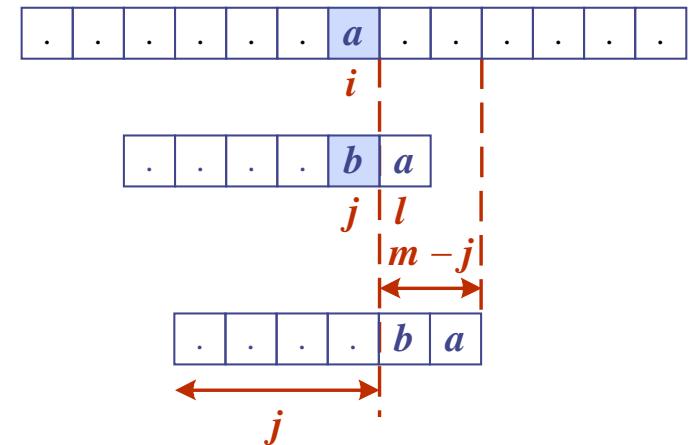
The Boyer-Moore Algorithm

```

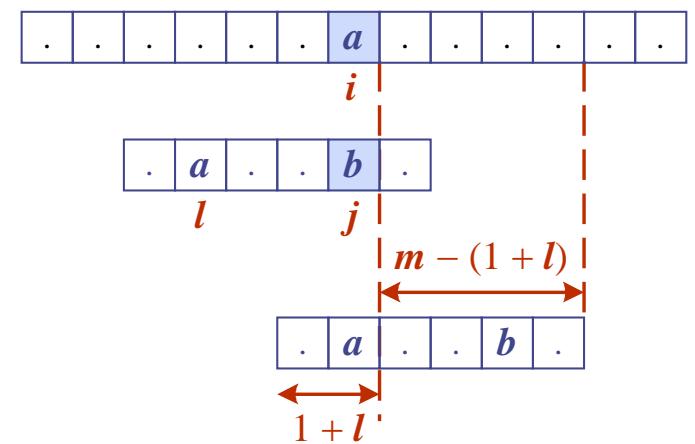
Algorithm BoyerMooreMatch( $T, P, \Sigma$ )
   $L \leftarrow \text{lastOccurrenceFunction}(P, \Sigma)$ 
   $i \leftarrow m - 1$ 
   $j \leftarrow m - 1$ 
  repeat
    if  $T[i] = P[j]$ 
      if  $j = 0$ 
        return  $i$  { match at  $i$  }
      else
         $i \leftarrow i - 1$ 
         $j \leftarrow j - 1$ 
    else
      { character-jump }
       $l \leftarrow L[T[i]]$ 
       $i \leftarrow i + m - \min(j, 1 + l)$ 
       $j \leftarrow m - 1$ 
  until  $i > n - 1$ 
  return  $-1$  { no match }

```

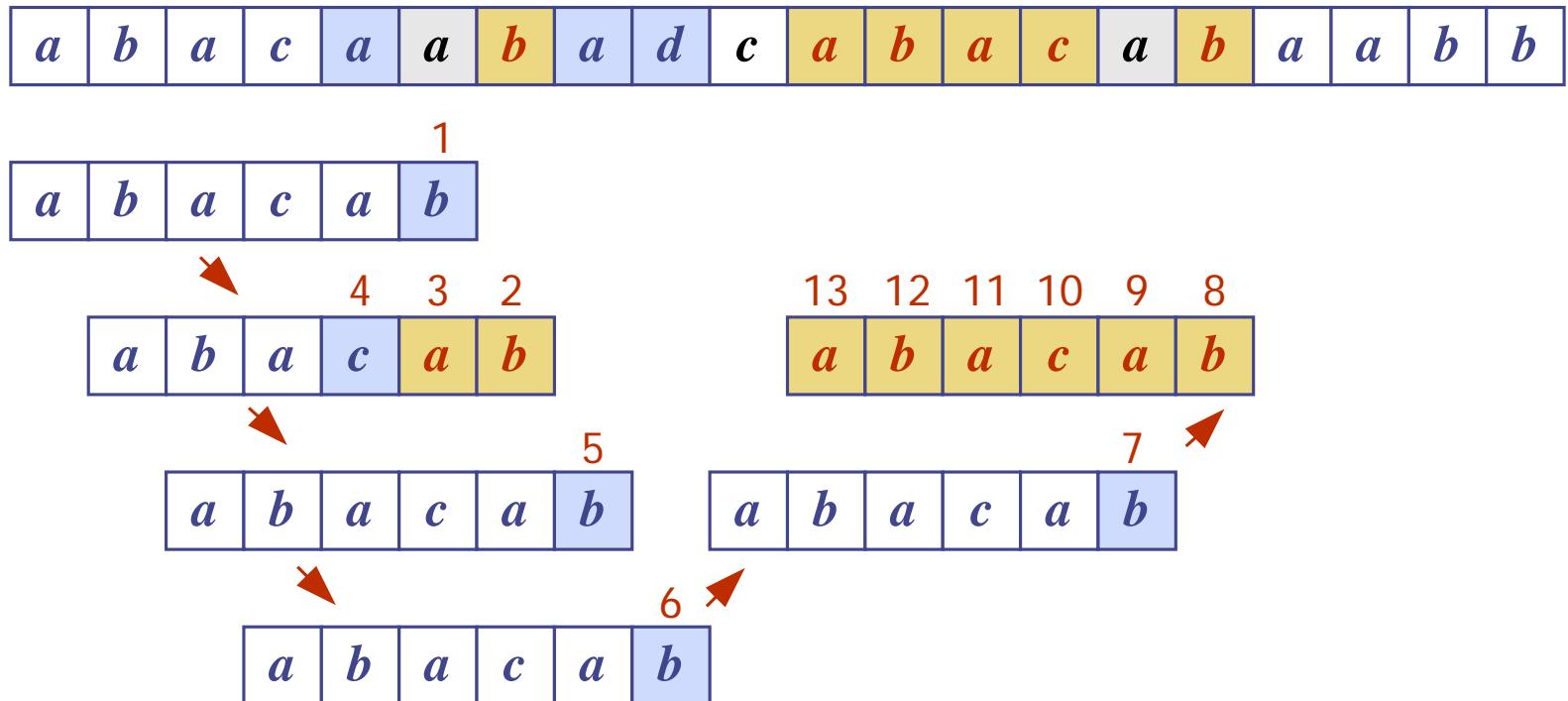
Case 1: $j \leq 1 + l$



Case 2: $1 + l \leq j$

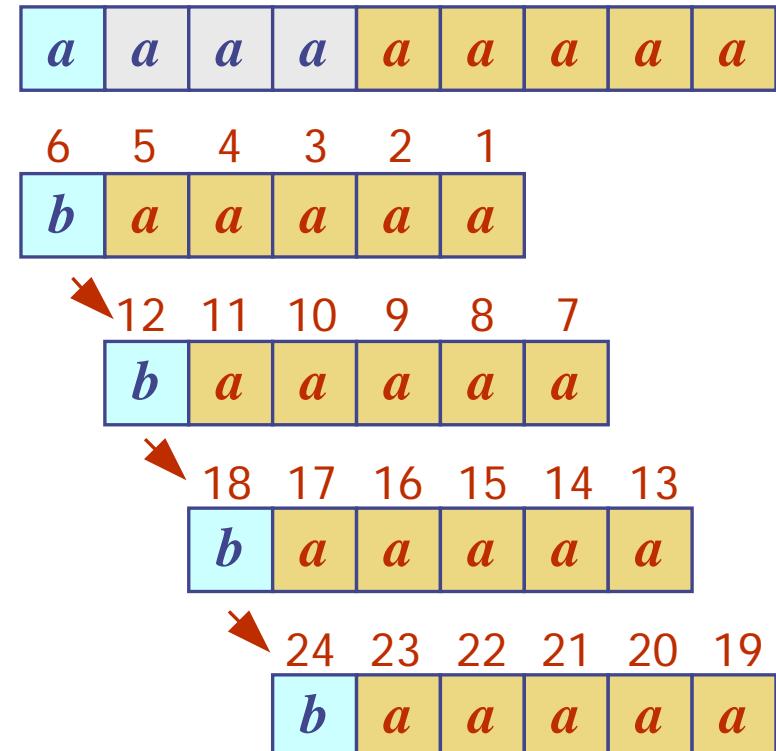


Example



Analysis

- **Boyer-Moore's** algorithm runs in time $O(nm + s)$
- **Example** of worst case:
 - $T = aaa \dots a$
 - $P = baaa$
- The worst case may occur in images and DNA sequences but is unlikely in English text
- Boyer-Moore's algorithm is significantly faster than the brute-force algorithm on English text



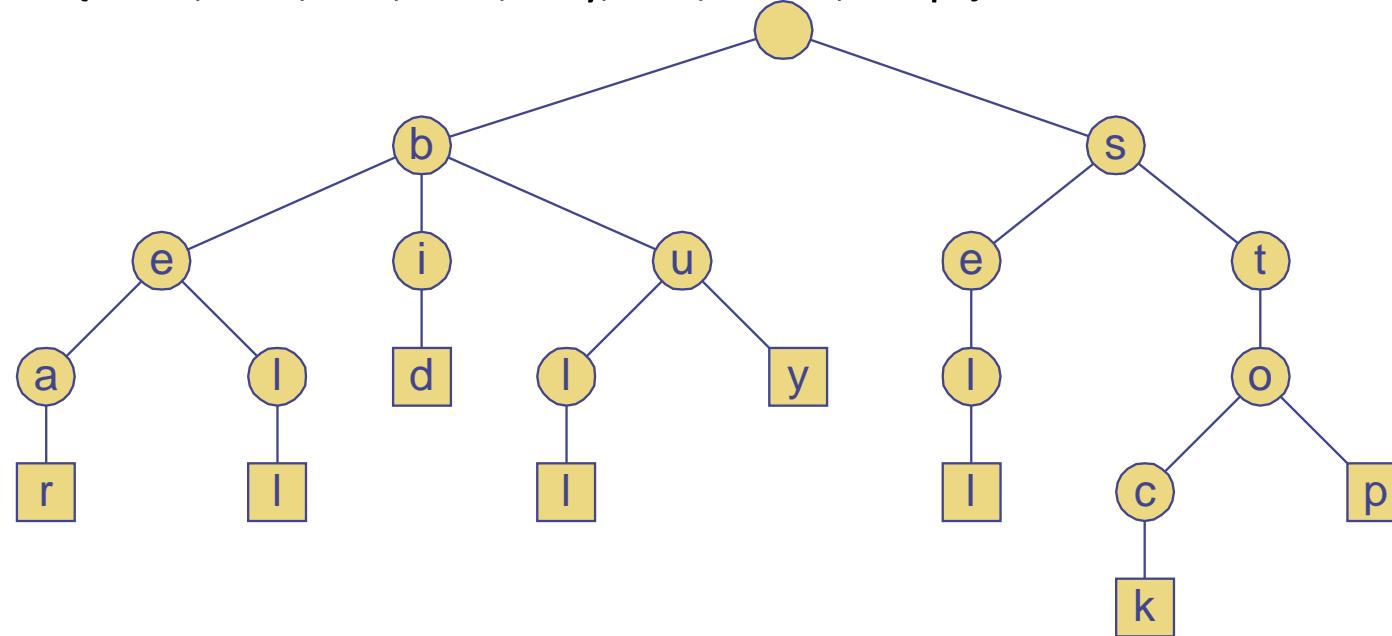
Preprocessing Strings



- Preprocessing the pattern speeds up pattern matching queries
- If the text is large, immutable and searched for often (e.g., works by Shakespeare), we may want to preprocess the text instead of the pattern
- A **trie** is a compact data structure for representing a set of strings, such as all the words in a text
 - A tries supports pattern matching queries in time proportional to the pattern size

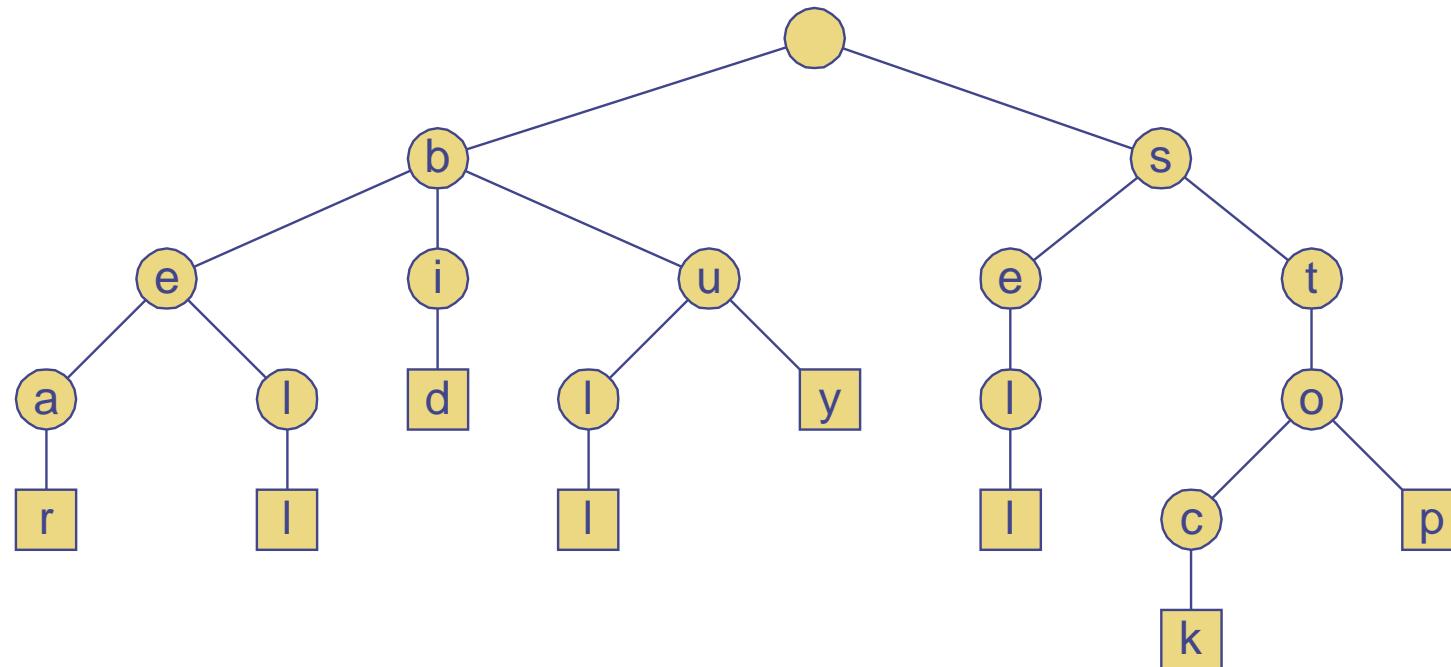
Standard Tries

- **The standard trie** for a set of strings S is an ordered tree such that:
 - Each node but the root is labeled with a character
 - The children of a node are alphabetically ordered
 - The paths from the external nodes to the root yield the strings of S
- **Example:** standard trie for the set of strings
 $S = \{ \text{bear}, \text{bell}, \text{bid}, \text{bull}, \text{buy}, \text{sell}, \text{stock}, \text{stop} \}$



Analysis of Standard Tries

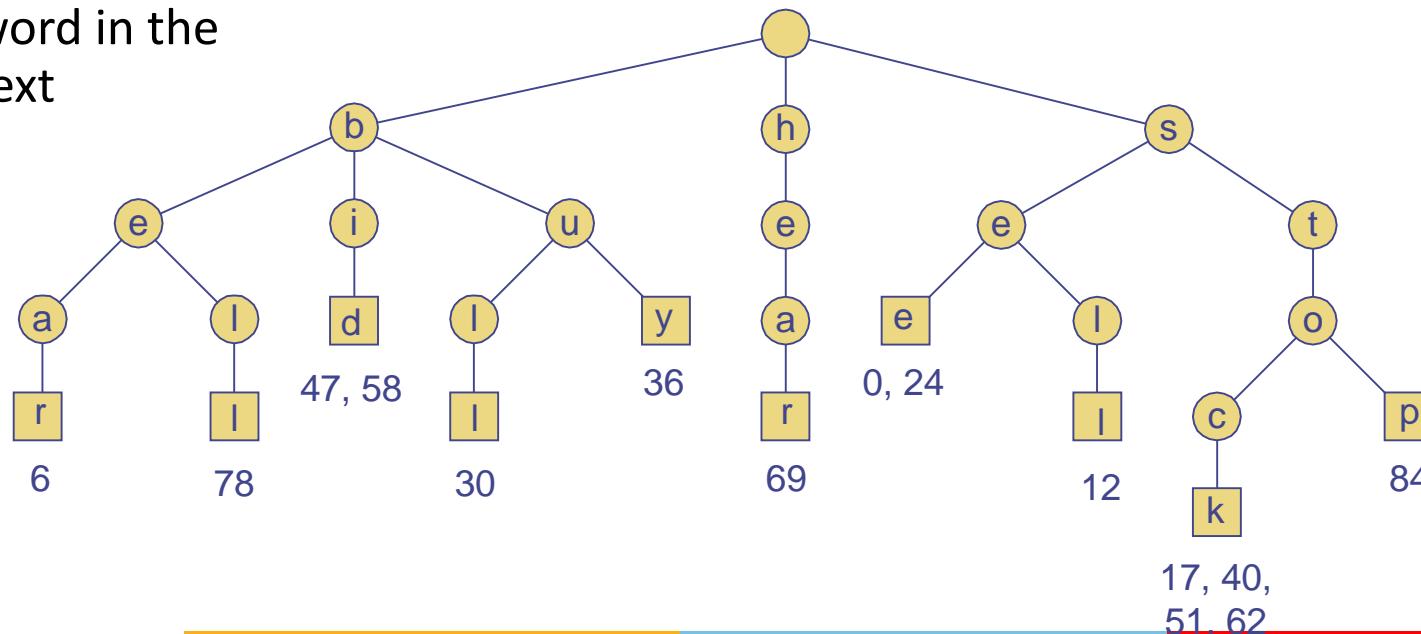
- A standard trie uses $O(n)$ space and supports searches, insertions and deletions in time $O(dm)$, where:
 - n total size of the strings in S
 - m size of the string parameter of the operation
 - d size of the alphabet



Word Matching with a Trie

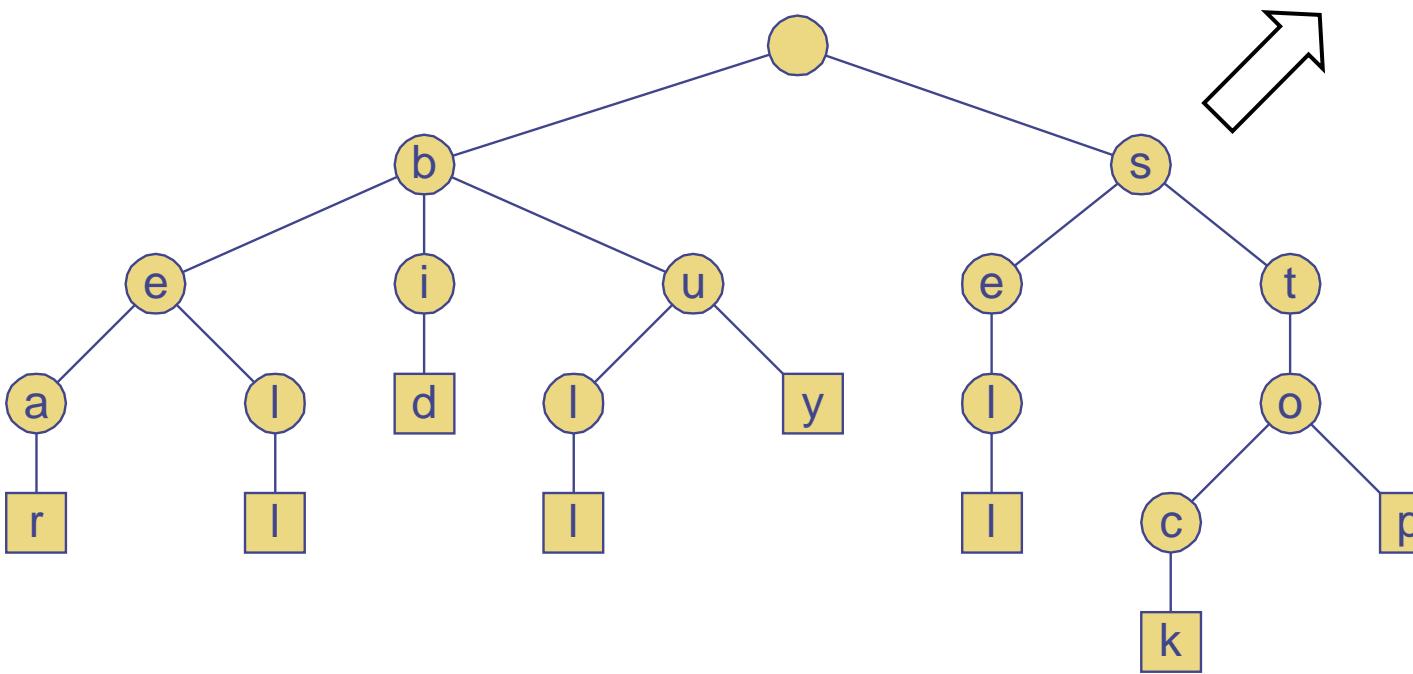
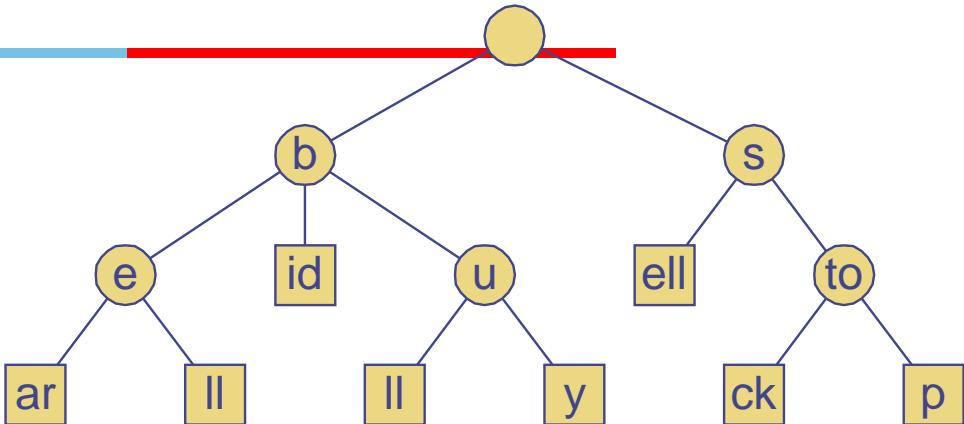
- We insert the words of the text into a trie
- Each leaf stores the occurrences of the associated word in the text

see	a	bear?	sell	i	stock!
0	1	2	3	4	5
see	a	bu	ll?	buy	stock!
24	25	26	27	28	29
bid	s	t	ock!	bid	s
47	48	49	50	51	52
53	54	55	56	57	58
59	60	61	62	63	64
65	66	67	68		
hear	t	he	bel	l?	stop!
69	70	71	72	73	74
75	76	77	78	79	80
81	82	83	84	85	86
87	88				



Compressed Tries

- A compressed trie has internal nodes of degree at least two
- It is obtained from standard trie by compressing chains of “redundant” nodes



- Compact representation of a compressed trie
- **Approach**
 - For an array of strings $S = S[0], \dots S[s-1]$
 - Store ranges of indices at each node
 - Instead of substring
 - Represent as a triplet of integers (i, j, k)
 - Such that $X = s[i][j..k]$
 - Example: $S[0] = "abcd"$, $(0,1,2) = "bc"$
- **Properties**
 - Uses $O(s)$ space, where $s = \#$ of strings in the array
 - Serves as an auxiliary index structure

Compact Representation

- **Example**

S[0] =

0	1	2	3	4
s	e	e		

 S[1] =

0	1	2	3	4
b	e	a	r	

 S[2] =

0	1	2	3	4
s	e	l	l	

 S[3] =

0	1	2	3	4
s	t	o	c	k

S[4] =

0	1	2	3	
b	u	l	l	

 S[5] =

0	1	2	3	
b	u	y		

 S[6] =

0	1	2	3	
b	i	d		

S[7] =

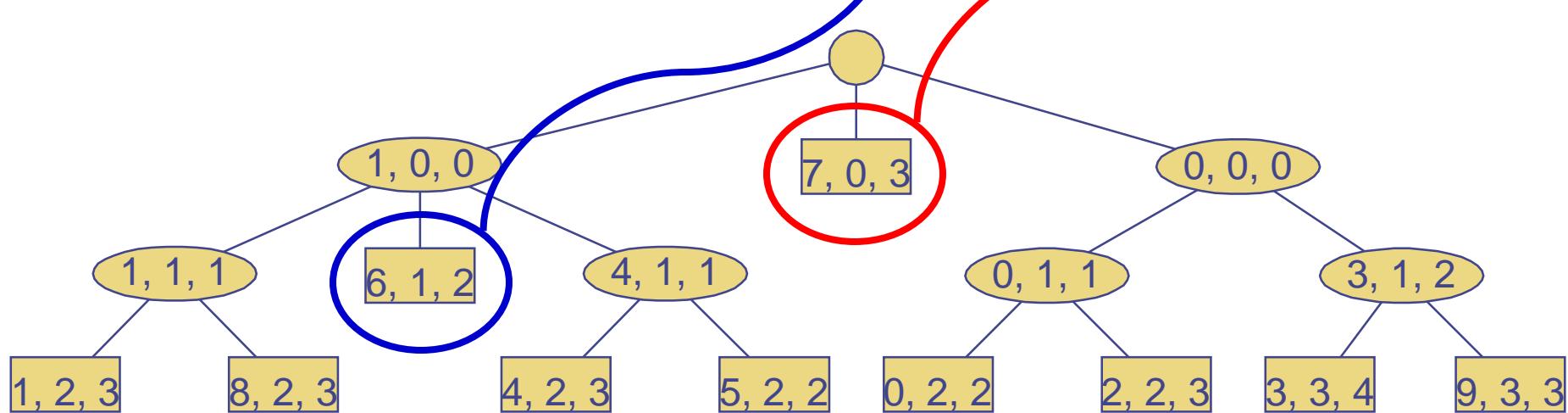
0	1	2	3	
h	e	a	r	

 S[8] =

0	1	2	3	
b	e	l	l	

 S[9] =

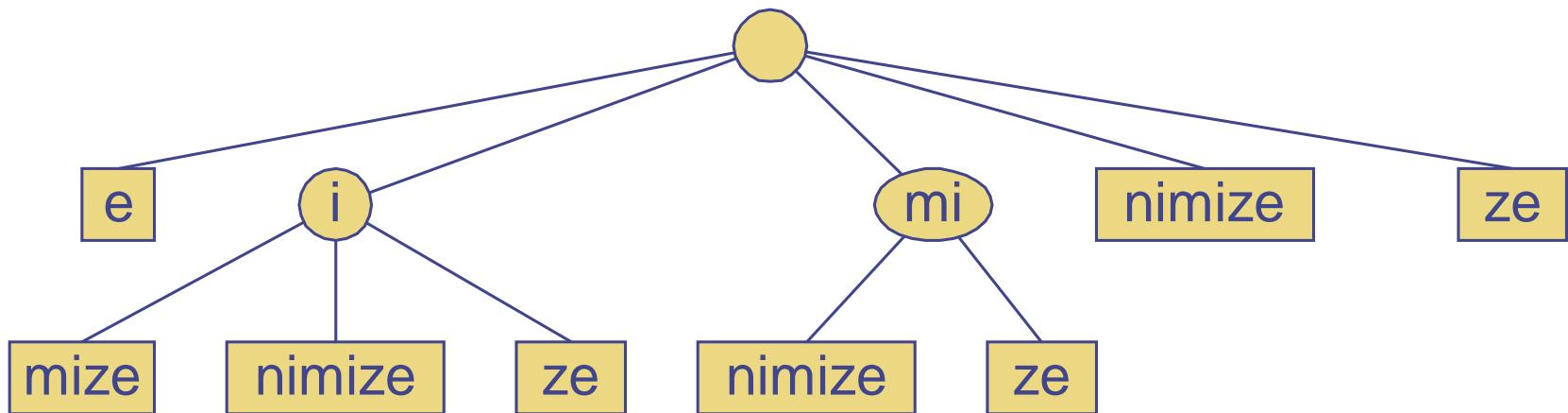
0	1	2	3	
s	t	o	p	



Suffix Trie

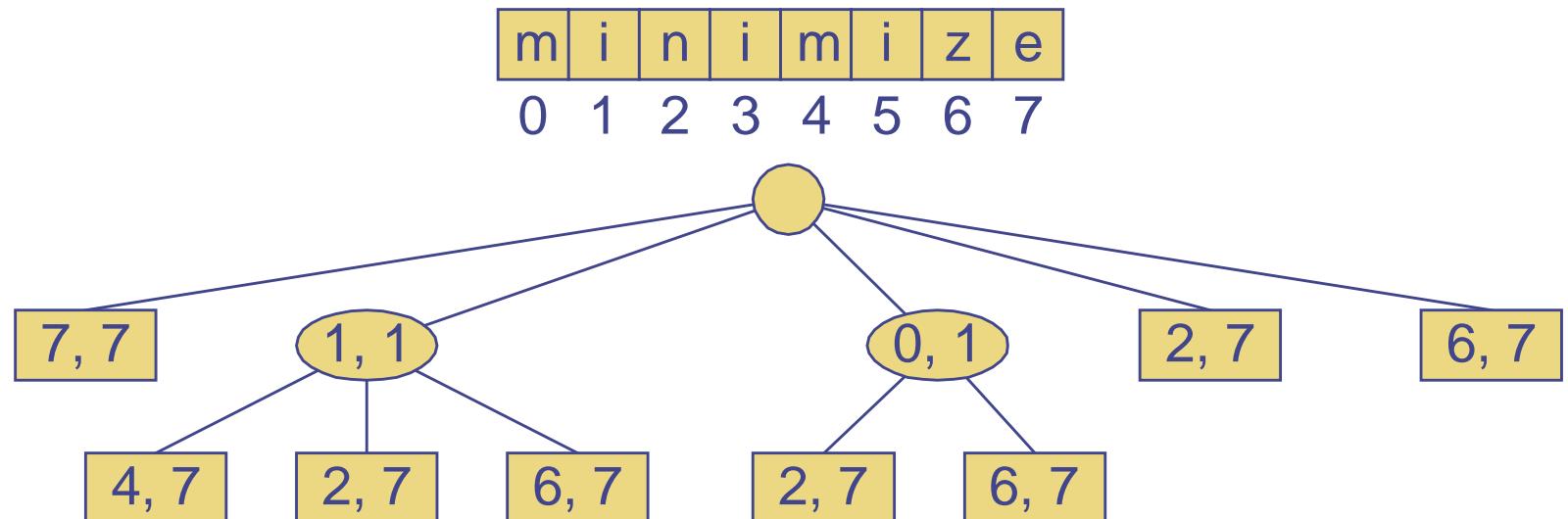
The **suffix trie** of a string X is the compressed trie of all the suffixes of X

m	i	n	i	m	i	z	e
0	1	2	3	4	5	6	7



Analysis of Suffix Tries

- Compact representation of the suffix trie for a string X of size n from an alphabet of size d
 - Uses $O(n)$ space
 - Supports arbitrary pattern matching queries in X in $O(dm)$ time, where m is the size of the pattern
 - Can be constructed in $O(n)$ time





BITS Pilani
Pilani Campus

Course Name : **Data Structures & Algorithms**

Bharat Deshpande
Computer Science & Information Systems

Text Compression

- Given a string X, efficiently encode X into a smaller bit string Y.
- **Easy Solution**

Since there are only 26 characters in English Language & there are 32 bit string of length 5, Each alphabet can be represented by a bit string of length 5.

- This is called a **fixed length code**.

Text Compression

Question

Is it possible to find coding scheme, in which fewer bits are used.

Answer – Variable length codes

- Alphabets that occur more frequently should be encoded using short bit strings & rarely occurring alphabets should be encoded using long bit strings

Example

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Fixed length	000	001	010	011	100	101
Variable length	0	101	100	111	1101	1100

A file with 100000 character contains only the alphabets a – f.

- Fixed length code requires 300000 bits to code the file.
- Variable length coding requires 224000 bits to code the file.

Prefix Code

- **Important**

In variable length code, some method must be used to determine where the bits for each character start and end.

For Example: If e : 0, a : 1, t : 01

Then 0101 could correspond to eat, tea or eaea.

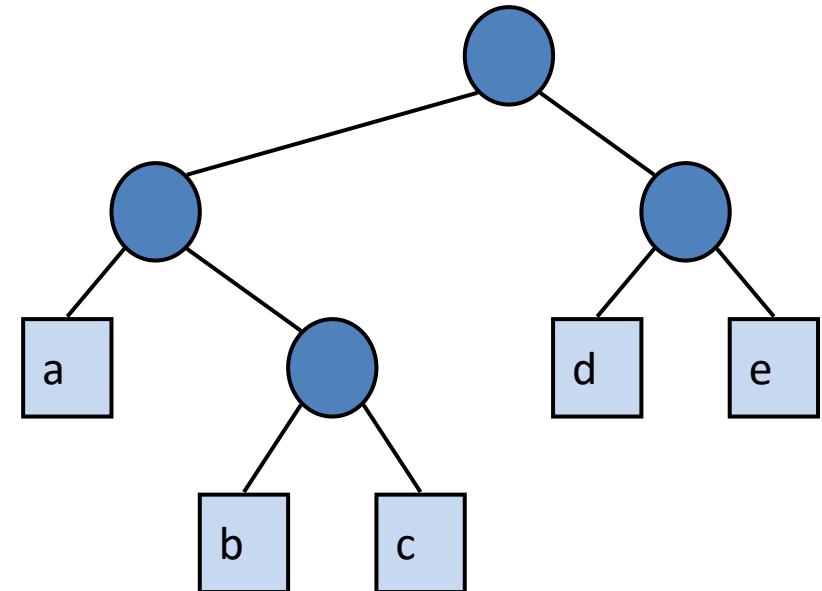
A **prefix code** is a binary code such that no code word is the prefix of another code-word

Encoding Tree

Key: A binary code can be represented by a binary tree.

- Each external node stores a character
- The code word of a character is given by the path from the root to the external node storing the character (0 for a left child and 1 for a right child)

00	010	011	10	11
a	b	c	d	e



The Huffman Coding algorithm-

History

- In 1951, David Huffman and his MIT information theory classmates given the choice of a term paper or a final exam
 - Huffman hit upon the idea of using a frequency-sorted binary tree and quickly proved this method the most efficient.
 - In doing so, the student outdid his professor, who had worked with information theory inventor Claude Shannon to develop a similar code.
 - Huffman built the tree from the bottom up instead of from the top down
-

Huffman Code

- **Greedy algorithm** that constructs an optimal prefix code.
 - Algorithm builds the tree in a bottom up manner.
 - C be set of n alphabets and $f[c]$ be the frequency of $c \in C$.
 - Algorithm begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ **merging operations** to create the final tree.
 - A priority queue Q keyed on f values, is used to identify the two least frequent objects to merge together.
 - The result of the merger of two objects is a new object whose frequency is the sum of two frequencies of the two objects that are merged. The new object is inserted back into Q .

Huffman's Algorithm

- It runs in time $O(n + d \log d)$, where n is the size of X and d is the number of distinct characters of X
- A heap-based priority queue is used as an auxiliary structure

Algorithm *HuffmanEncoding*(X)

```

Input string  $X$  of size  $n$ 
Output optimal encoding trie for  $X$ 
 $C \leftarrow \text{distinctCharacters}(X)$ 
 $\text{computeFrequencies}(C, X)$ 
 $Q \leftarrow$  new empty heap
for all  $c \in C$ 
   $T \leftarrow$  new single-node tree storing  $c$ 
   $Q.\text{insert}(\text{getFrequency}(c), T)$ 
while  $Q.\text{size}() > 1$ 
   $f_1 \leftarrow Q.\text{minKey}()$ 
   $T_1 \leftarrow Q.\text{removeMin}()$ 
   $f_2 \leftarrow Q.\text{minKey}()$ 
   $T_2 \leftarrow Q.\text{removeMin}()$ 
   $T \leftarrow \text{join}(T_1, T_2)$ 
   $Q.\text{insert}(f_1 + f_2, T)$ 
return  $Q.\text{removeMin}()$ 

```

Example

$X = \text{abracadabra}$

Frequencies

a	b	c	d	r
5	2	1	1	2

a	b	c	d	r
5	2	1	1	2



a	b	c	d	r
5	2			2



a		c	d	
5				

