Shatt Al-Arab University
College of Science
Computer Science Department

Lecture #01

Searching Algorithms

Linear and Binary Searching

BY

Assit. Lec. Mustafa H. Hashim

1. Linear Search Algorithm

The **Linear Search Algorithm** is a simple searching algorithm that checks each element in a list or array sequentially until the desired element is found or the entire list has been searched. It's often called a **sequential search** because it processes the elements one by one in order.

How Linear Search Works:

1. **Start** at the beginning of the list.
2. **Compare** the target value (the element you're searching for) with the current element in the list.
3. If the **target matches** the current element, return the index or position of that element.
4. If the **target doesn't match**, move to the next element and repeat the process.
5. **Continue** this until the target is found or you've checked all elements.
6. If the element is not found after checking the whole list, return an indication that the element is not present (like -1).

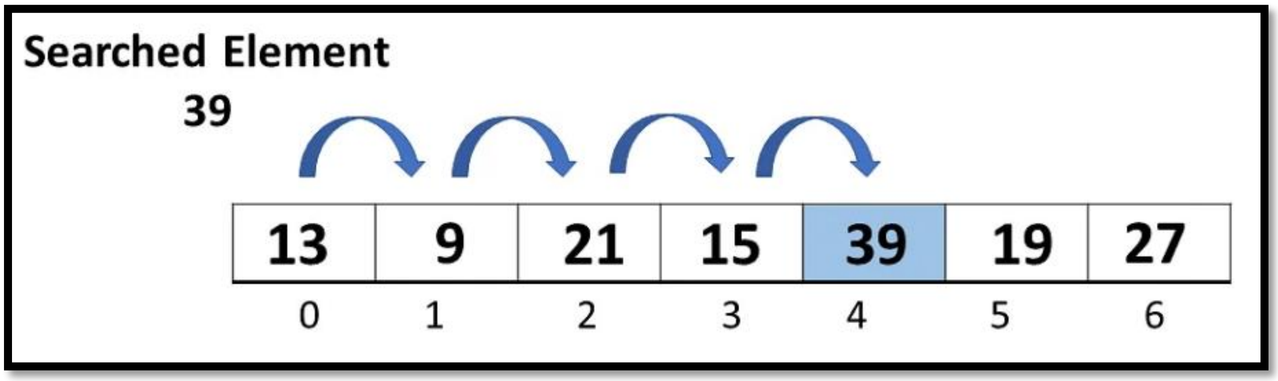
Pseudocode:

```
function linearSearch(array, target):
    for index from 0 to length of array - 1:
        if array[index] == target:
            return index
    return -1  # Target not found
```

Time Complexity:

- **Worst-case time complexity**: O(n), where n is the number of elements in the list. In the worst case, you would need to check every single element.

- **Best-case time complexity**: O(1), if the target element is found at the first position.



Example:

We have a list: [5, 3, 8, 4, 6], and we want to search for 8.

1. Compare 5 with 8 → no match.
2. Compare 3 with 8 → no match.
3. Compare 8 with 8 → match found at index 2.

So, the result would be 2, since 8 is found at the third position (index 2 in 0-based indexing).

When to use Linear Search:

- When the list is unsorted or if the size of the list is small.
- It's easy to implement and doesn't require additional space (it's done in-place).

2. Binary Search Algorithm

The **Binary Search Algorithm** is a more efficient search algorithm than linear search, but it only works on **sorted arrays or lists**. It repeatedly divides the search interval in half, checking whether the target value is in the left or right half of the current interval. This reduces the number of elements to check at each step.

How Binary Search Works:

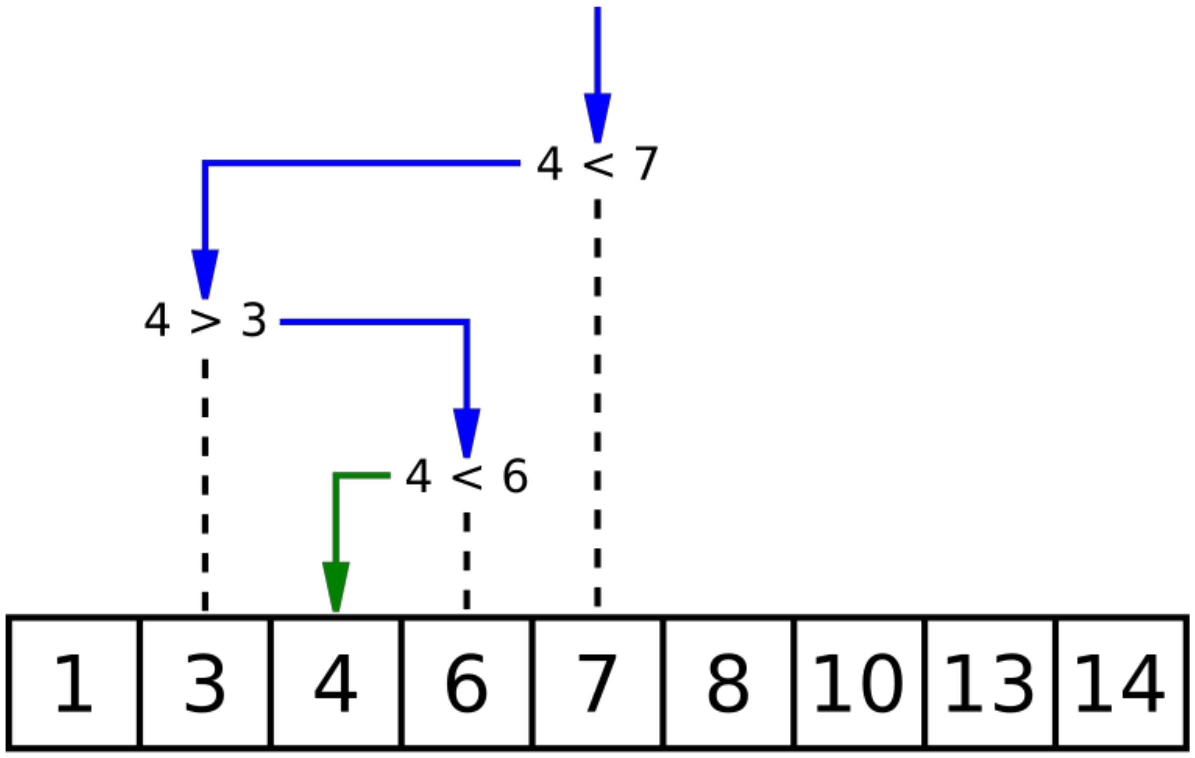
1. **Start** with two pointers: low (initially set to the first index) and high (initially set to the last index).
2. Find the middle index: mid = (low + high) // 2.
3. **Compare** the element at mid with the target:
    - If the element at mid is equal to the target, the search is successful, and you return the index mid.
    - If the element at mid is less than the target, the target must be in the right half of the list, so you update low = mid + 1.
    - If the element at mid is greater than the target, the target must be in the left half of the list, so you update high = mid - 1.
4. Repeat steps 2 and 3 until you either find the target or the search interval becomes invalid (i.e., low > high), indicating that the target is not in the list.

Pseudocode:

```
function binarySearch(array, target):
    low = 0
    high = length of array - 1

    while low <= high:
        mid = (low + high) // 2
        if array[mid] == target:
            return mid  # Found the target, return index
        elif array[mid] < target:
            low = mid + 1  # Target is in the right half
        else:
            high = mid - 1  # Target is in the left half

    return -1  # Target not found
```

Time Complexity:

- **Worst-case time complexity**: O(log n), where n is the number of elements in the list. Each time, the size of the list is halved.
- **Best-case time complexity**: O(1), if the target is found at the middle in the first comparison.



Searching for key=4

Example:

Let's say we have a sorted list: [2, 4, 6, 8, 10, 12, 14, 16, 18], and we want to search for 10.

1. **Initial low = 0, high = 8** (length of the list is 9, so index goes from 0 to 8).
   - Middle index: mid = (0 + 8) // 2 = 4.
   - Element at index 4 is 10, which is the target.

So, the result is that the target 10 is found at index **4**.

```
# Example usage
arr = [2, 4, 6, 8, 10, 12, 14, 16, 18]
target = 10
index = binary_search(arr, target)


if index != -1:
    print(f"Element {target} found at index {index}")
else:
    print(f"Element {target} not found in the list")
```

Output:

```
Element 10 found at index 4
```

When to use Binary Search:

- Use binary search when your data is **sorted**.
- It's much faster than linear search for large datasets because it drastically reduces the number of comparisons needed.

Data Structures and Algorithms II
Second Stage

Shatt Al-Arab University
College of Science
Computer Science Department

Lecture #02

Sorting Algorithms

Bubble Sort, Quick Sort, and Merge Sort

BY

Assit. Lec. Mustafa H. Hashim

1. Bubble Sort

Bubble Sort is one of the simplest sorting algorithms. It works by repeatedly stepping through the list to be sorted, comparing adjacent elements, and swapping them if they are in the wrong order. This process is repeated until the list is sorted. The name "bubble sort" comes from the way larger elements "bubble" to the top of the list with each pass through the array.

How Bubble Sort Works:

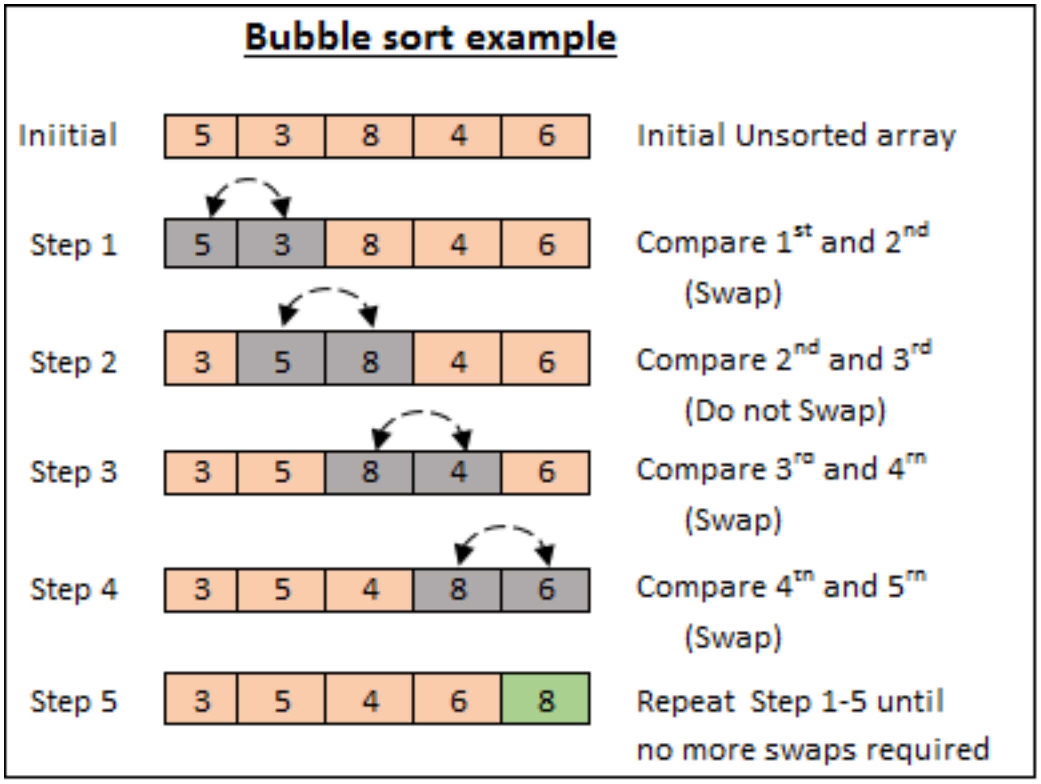
1. **Start** from the first element of the list.
2. **Compare** the current element with the next element.
   - If the current element is greater than the next element, **swap** them.
   - If the current element is smaller or equal, no action is taken.
3. **Move to the next pair** of adjacent elements and repeat the comparison and swap if necessary.
4. **Repeat** the process for each element in the list. After each pass, the largest unsorted element will have "bubbled" up to its correct position.
5. **Stop** when no swaps are needed during a full pass through the list, indicating that the list is fully sorted.

Pseudocode:

```
function bubbleSort(arr):
  n = length of arr
  for i = 0 to n-1:
    # Flag to optimize the process (to stop early if no swaps)
    swapped = false
    for j = 0 to n-i-2:
      if arr[j] > arr[j+1]:
            swap arr[j] and arr[j+1]
            swapped = true
      if not swapped:
         break  # No swaps, so the array is already sorted
```

Time Complexity:

- **Worst-case time complexity**: O(n^2), where n is the number of elements in the list. In the worst case, you might need to make n-1 passes, and for each pass, you make n-i-1 comparisons.
- **Best-case time complexity**: O(n), if the list is already sorted (optimization with the swapped flag).


Bubble sort example
Iniitial
5 3 8 4 6
Initial Unsorted array
Step 1
5 3 8 4 6
Compare 1st and 2nd (Swap)
Step 2
3 5 8 4 6
Compare 2nd and 3rd (Do not Swap)
Step 3
3 5 8 4 6
Compare 3rd and 4th (Swap)
Step 4
3 5 4 8 6
Compare 4th and 5th (Swap)
Step 5
3 5 4 6 8
Repeat Step 1-5 until no more swaps required

Example:

Let’s say we have the following list of integers that we want to sort: [5, 2, 9, 1, 5, 6].

Pass 1:

- Compare 5 and 2: Swap → [2, 5, 9, 1, 5, 6]
- Compare 5 and 9: No swap.
- Compare 9 and 1: Swap → [2, 5, 1, 9, 5, 6]
- Compare 9 and 5: Swap → [2, 5, 1, 5, 9, 6]
- Compare 9 and 6: Swap → [2, 5, 1, 5, 6, 9]

After the first pass, 9 is correctly placed at the end of the list.

Pass 2:

- Compare 2 and 5: No swap.
- Compare 5 and 1: Swap → [2, 1, 5, 5, 6, 9]
- Compare 5 and 5: No swap.
- Compare 5 and 6: No swap.

After the second pass, 6 is correctly placed.

Pass 3:

- Compare 2 and 1: Swap → [1, 2, 5, 5, 6, 9]
- Compare 2 and 5: No swap.
- Compare 5 and 5: No swap.

After the third pass, no more swaps are needed, so the list is sorted.

```
# Example usage

arr = [5, 2, 9, 1, 5, 6]

bubble_sort(arr)

print("Sorted array:", arr)
```

Output:

```
Sorted array: [1, 2, 5, 5, 6, 9]
```

When to Use Bubble Sort:

- **Educational purposes**: It's simple to understand and explain, but not efficient for large datasets.
- **Small datasets**: It can be practical for small lists or arrays, but for larger datasets, other algorithms like Merge Sort, Quick Sort, or even Insertion Sort are typically preferred due to their better time complexities.

2. Quick Sort Algorithm

Quick Sort is a highly efficient **divide and conquer** sorting algorithm that works by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays—those less than the pivot and those greater than the pivot. The sub-arrays are then sorted recursively.

How Quick Sort Works:

1. **Choose a pivot** element from the array. The pivot can be selected in various ways, such as choosing the first element, the last element, the middle element, or a random element. (Different strategies can affect performance, but the basic idea remains the same.)
2. **Partitioning**:
   - Rearrange the elements in the array so that all elements smaller than the pivot come before it, and all elements larger than the pivot come after it. This step places the pivot in its correct sorted position.

3. **Recursively apply** the above steps to the sub-arrays (those to the left and right of the pivot).
4. **Base case**: The recursion terminates when the sub-array has only one element or is empty, meaning the array is sorted.

Pseudocode:

```
function quickSort(arr):
    if length of arr <= 1:
        return arr

    pivot = arr[0]  # Or choose a random pivot or the last element
    left = []
    right = []

    for each element in arr[1:]:
        if element < pivot:
            append element to left
        else:
            append element to right

    return quickSort(left) + [pivot] + quickSort(right)
```

Time Complexity:

- **Best-case time complexity**: O(n log n), which happens when the pivot divides the array roughly in half each time.

- **Worst-case time complexity**: $O(n^2)$, which occurs when the pivot is always the smallest or largest element (e.g., when the array is already sorted or nearly sorted).
- **Average-case time complexity**: $O(n \log n)$, which is the expected performance for most inputs.

Example:

Let's say we have the following list: [8, 2, 7, 1, 5, 3] and we want to sort it using quick sort.

Step 1: Choose a pivot (e.g., the first element: 8).

- Left sub-array: [2, 7, 1, 5, 3] (all elements less than 8)
- Right sub-array: [] (no elements greater than 8)
- The pivot 8 is now in its correct position.

Step 2: Apply quick sort on the left sub-array [2, 7, 1, 5, 3].

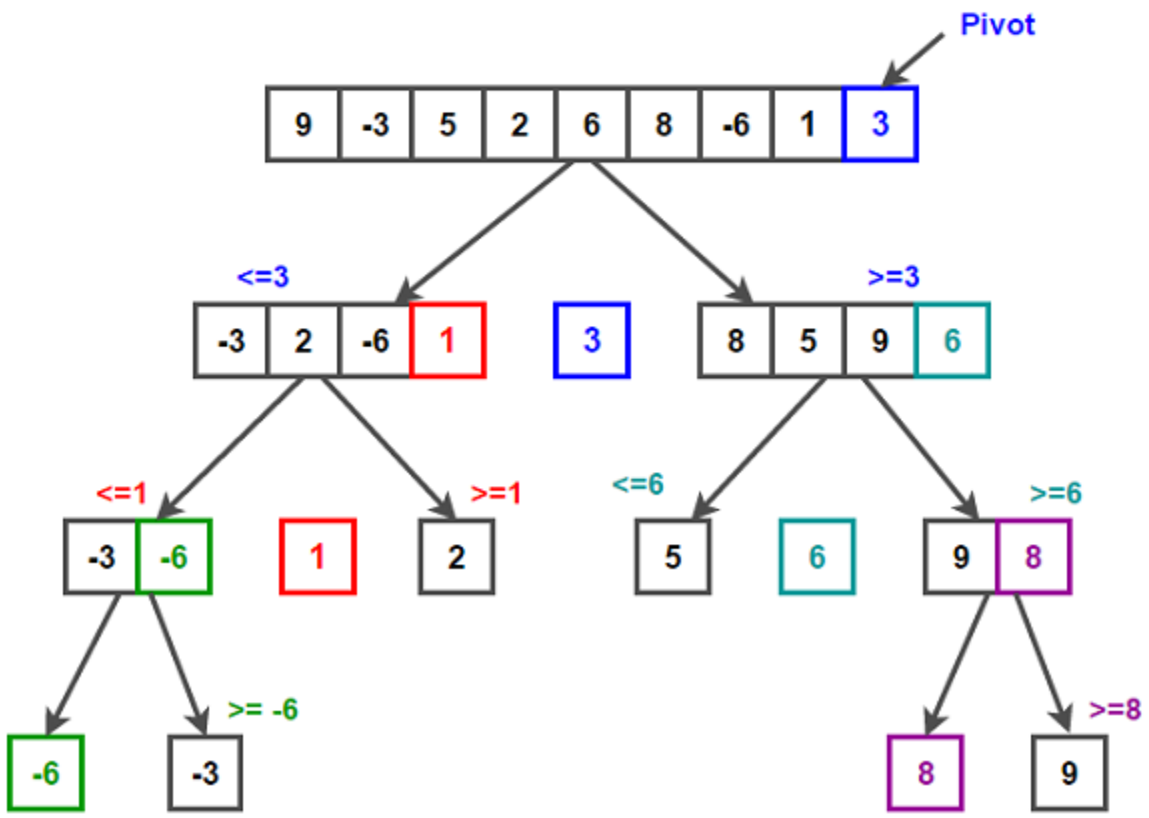
1. **Choose pivot**: 2.
    - Left sub-array: [1]
    - Right sub-array: [7, 5, 3]
    - The pivot 2 is now in its correct position.
2. Apply quick sort on [1] and [7, 5, 3].
    - For [1], no further sorting is needed.
    - For [7, 5, 3], choose a pivot (7):
        - Left sub-array: [5, 3]
        - Right sub-array: []
        - The pivot 7 is now in its correct position.
    - Apply quick sort on [5, 3]:

- Choose a pivot (5):
  - Left sub-array: [3]
  - Right sub-array: []
- The pivot 5 is now in its correct position.

Now all elements are sorted.

Step 3: Combine all sorted sub-arrays:

Final sorted array: [1, 2, 3, 5, 7, 8]



When to Use Quick Sort:

- **Fast on average**: Quick Sort is generally one of the fastest sorting algorithms for large datasets due to its average-case time complexity of O(n log n).
- **Efficient in-place**: It doesn't require extra memory for sorting, as it sorts the array in place (except for recursion stack space).
- **Best for random or unsorted data**: It's great for general-purpose sorting but can perform poorly if the data is already sorted or nearly sorted unless you optimize the pivot selection strategy.

3. Merge Sort Algorithm

Merge Sort is a highly efficient, **divide and conquer** sorting algorithm. It works by dividing the input array into two halves, recursively sorting each half, and then merging the two sorted halves back together.

How Merge Sort Works:

1. **Divide** the array into two halves.
    - This is done recursively, splitting the array until each sub-array has only one element (a single-element array is trivially sorted).
2. **Conquer** by recursively sorting the two halves.
3. **Combine (Merge)** the two sorted halves into a single sorted array. This is done by comparing elements of the two halves one by one and putting them in the correct order.

Merge Sort Algorithm:

1. **Base case**: If the array has only one element or is empty, it's already sorted.
2. **Recursive case**:
    - Split the array into two halves.
    - Recursively sort each half.

- Merge the two sorted halves into a single sorted array.

Pseudocode:

```
function mergeSort(arr):
   if length of arr <= 1:
      return arr

   mid = length of arr // 2
   left = mergeSort(arr[:mid])  # Recursively sort the left half
   right = mergeSort(arr[mid:])  # Recursively sort the right half

   return merge(left, right)  # Merge the sorted halves

function merge(left, right):
   result = []
   i, j = 0, 0

   while i < length of left and j < length of right:
      if left[i] < right[j]:
         append left[i] to result
         i += 1
      else:
         append right[j] to result
         j += 1
```

```
    # If there are remaining elements in left or right
    result += left[i:]
    result += right[j:]

    return result
```

Time Complexity:

- **Worst-case time complexity**: O(n log n), where n is the number of elements in the array. This is because the array is divided into two halves at each level (log n divisions), and merging takes linear time (O(n)).
- **Best-case time complexity**: O(n log n), because merge sort always divides the array in half and then merges the sorted halves, regardless of the initial order of elements.
- **Space complexity**: O(n), because merge sort requires additional space for the merged arrays.

Example:

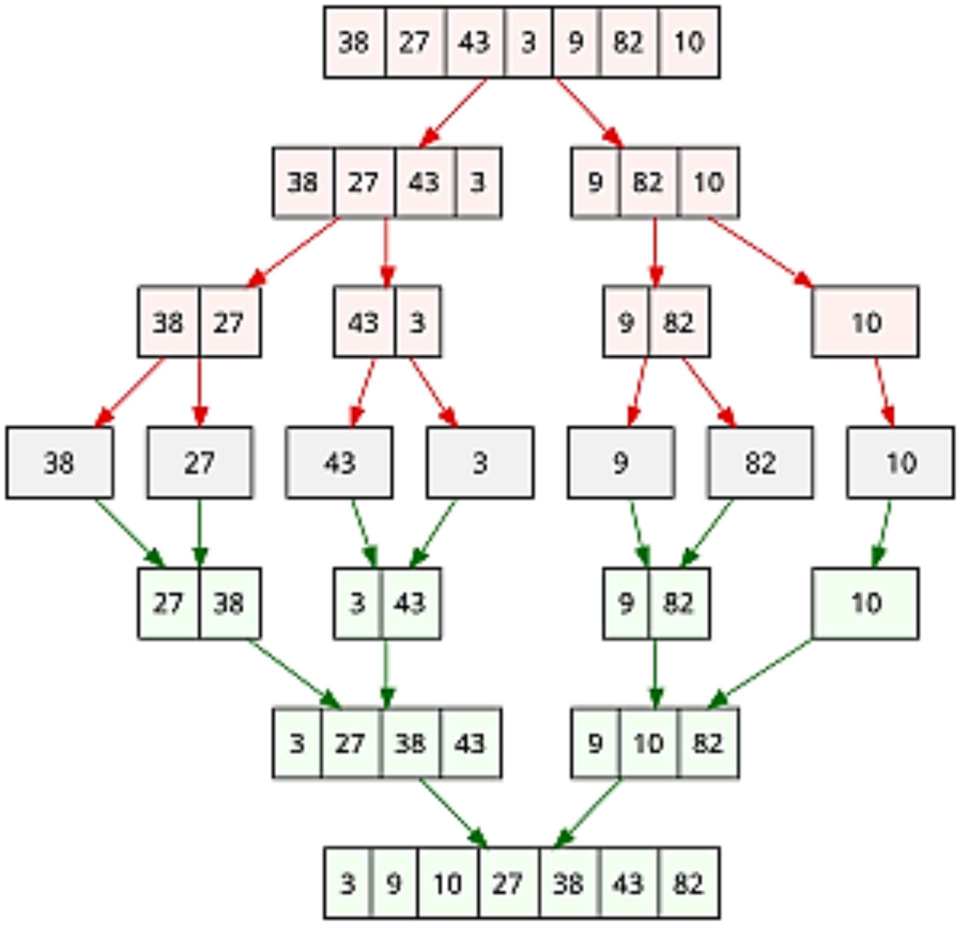
Let's walk through the process of sorting the array [38, 27, 43, 3, 9, 82, 10] using merge sort.

1. **Initial array**: [38, 27, 43, 3, 9, 82, 10]
    - Split into two halves: [38, 27, 43] and [3, 9, 82, 10].
2. **Sort the left half** ([38, 27, 43]):
    - Split into two halves: [38] and [27, 43].
    - [38] is already sorted.
    - Sort [27, 43]: split into [27] and [43], both are sorted.
    - Merge [27] and [43] → [27, 43].

   - Merge [38] and [27, 43] → [27, 38, 43].

3. **Sort the right half** ([3, 9, 82, 10]):
   - Split into two halves: [3, 9] and [82, 10].
   - Sort [3, 9]: merge them → [3, 9].
   - Sort [82, 10]: merge them → [10, 82].
   - Merge [3, 9] and [10, 82] → [3, 9, 10, 82].
4. **Merge the two sorted halves**: [27, 38, 43] and [3, 9, 10, 82].
   - Compare and merge: [3, 9, 10, 27, 38, 43, 82].

The sorted array is: [3, 9, 10, 27, 38, 43, 82].



When to Use Merge Sort:

- **Large datasets**: Merge Sort is efficient for large datasets because of its O(n log n) time complexity.
- **Stable sort**: Merge Sort is a **stable sort**, meaning that it preserves the relative order of equal elements (which may be important in certain applications).
- **External sorting**: Merge Sort is commonly used for sorting large files that cannot fit into memory (external sorting), as it can be efficiently implemented with disk-based storage.

Advantages:

- **Guaranteed O(n log n)** time complexity in all cases (unlike Quick Sort, which can degrade to $O(n^2)$ in the worst case).
- **Stable sorting**: This can be important when sorting objects with multiple fields.
- **Suitable for linked lists**: Merge Sort works very well with linked lists as there's no need for random access to elements.

Disadvantages:

- **Space complexity**: Merge Sort requires O(n) additional space for the merged arrays, which may not be suitable for memory-limited environments.

Here's a comparison between **Bubble Sort**, **Quick Sort**, and **Merge Sort** based on various factors such as time complexity, space complexity, ease of implementation, and use cases.

1. Time Complexity:

Algorithm	Best-case	Average-case	Worst-case
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)	O(n log n)

- **Bubble Sort**: The best case occurs when the list is already sorted (O(n)), but in general, it's inefficient for larger datasets with a worst-case and average time complexity of $O(n^2)$.
- **Quick Sort**: The best and average cases occur when the pivot divides the array into nearly equal halves, leading to O(n log n) time complexity. However, if the pivot is poorly chosen (e.g., always the smallest or largest element), the worst-case complexity becomes $O(n^2)$.
- **Merge Sort**: Always has O(n log n) time complexity, both in the best, average, and worst cases. It's highly predictable.

2. Space Complexity:

Algorithm	Space Complexity
Bubble Sort	O(1) (in-place)
Quick Sort	O(log n) (in-place with recursion stack)
Merge Sort	O(n) (extra space for the merged arrays)

- **Bubble Sort**: Operates **in-place**, meaning it doesn't require additional memory apart from the input array, so it has O(1) space complexity.

- **Quick Sort**: Works **in-place** as well, with only recursion consuming additional space, so its space complexity is O(log n) due to the recursion stack.
- **Merge Sort**: Requires O(n) extra space because it creates temporary sub-arrays during the merging process.

3. Stability:

Algorithm	Stable
Bubble Sort	Yes
Quick Sort	No (in general, though it can be made stable)
Merge Sort	Yes

- **Bubble Sort**: It's **stable**, meaning that equal elements retain their relative order after sorting.
- **Quick Sort**: Typically **unstable** because it can reorder equal elements.
- **Merge Sort**: **Stable**, ensuring that equal elements retain their relative order.

4. Ease of Implementation:

Algorithm	Ease of Implementation
Bubble Sort	Very easy (simple to understand and implement)
Quick Sort	Moderate (requires understanding of partitioning and recursion)
Merge Sort	Moderate (requires extra space for merging and recursion)

- **Bubble Sort**: **Easiest** to implement because it is conceptually simple (just swap adjacent elements if they're in the wrong order).

- **Quick Sort**: **Moderately easy** but requires handling recursion and partitioning logic, which can be tricky for beginners.
- **Merge Sort**: **Moderate** to implement due to the need for recursion and extra space for merging, but it's straightforward once you understand the divide and conquer approach.

5. Performance on Large Datasets:

Algorithm	Performance on Large Datasets
Bubble Sort	Poor ($O(n^2)$)
Quick Sort	Excellent ($O(n \log n)$ average)
Merge Sort	Good ($O(n \log n)$)

- **Bubble Sort**: Performs **poorly** on large datasets due to its $O(n^2)$ time complexity.
- **Quick Sort**: Performs **exceptionally well** on large datasets in practice because its average time complexity is $O(n \log n)$. Its worst case ($O(n^2)$) can be avoided with good pivot selection.
- **Merge Sort**: Has **good** performance on large datasets with $O(n \log n)$ time complexity, but it requires additional memory for merging.

6. Use Cases:

Algorithm	Best Use Case
Bubble Sort	Small datasets or educational purposes (not suitable for large datasets)
Quick Sort	Large, unsorted datasets where average performance is critical; in-place sorting
Merge Sort	Large datasets where stable sorting is needed; external sorting for large files

- **Bubble Sort**: Best for small arrays or as an introductory sorting algorithm for learning purposes. It is **not recommended** for large datasets.
- **Quick Sort**: A **great choice** for general-purpose sorting, especially for large datasets where **average-case performance** matters. However, it should be avoided on nearly sorted data unless optimized with a good pivot strategy.
- **Merge Sort**: Excellent for sorting large datasets, particularly when a **stable sort** is required. It's also the go-to choice when **external sorting** (sorting very large files) is needed because it can be implemented with a file-based approach.

7. Summary Table:

Feature	Bubble Sort	Quick Sort	Merge Sort
Time Complexity	$O(n^2)$	$O(n \log n)$ (avg), $O(n^2)$ (worst)	$O(n \log n)$
Space Complexity	$O(1)$	$O(\log n)$	$O(n)$
Stability	Stable	Unstable	Stable
Ease of Implementation	Easy	Moderate	Moderate
Performance on Large Datasets	Poor ($O(n^2)$)	Excellent ($O(n \log n)$ avg)	Good ($O(n \log n)$)
Best Use Case	Small datasets, educational	Large unsorted datasets	Large datasets, stable sort needed, external sorting

Conclusion:

- **Bubble Sort** is simple but inefficient for larger datasets.
- **Quick Sort** is generally the fastest for average cases, making it a good choice for large datasets, but it can be inefficient in the worst case if the pivot selection is poor.
- **Merge Sort** guarantees $O(n \log n)$ performance and is stable, but it requires extra memory and might not be as fast in practice as Quick Sort for certain data types.

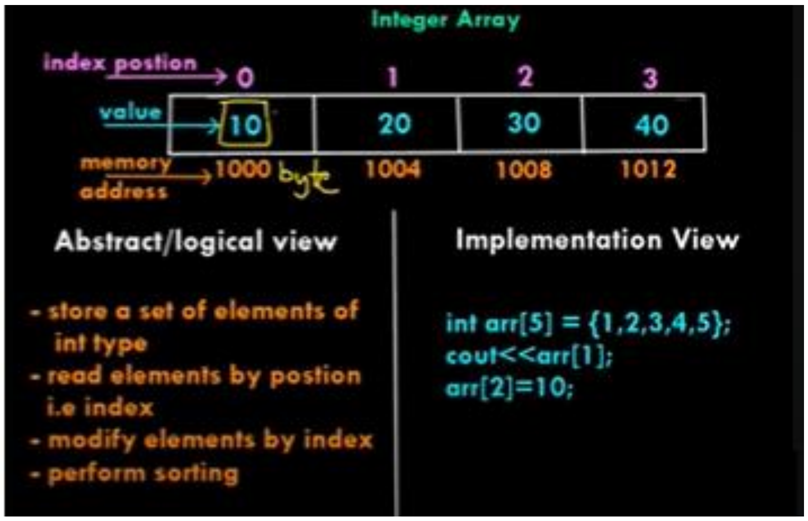
Each of these sorting algorithms has its strengths and weaknesses, and the choice of algorithm depends on the specific requirements of your application. If you're working with a large, unsorted dataset, Quick Sort or Merge Sort is often a better choice than Bubble Sort.

Lecture #03

Non-linear Data Structure (Abstract Data Structure-ADT):

Data structures where data elements are not arranged sequentially or linearly are called non-linear data structures. In a non-linear data structure, single level is not involved. Therefore, we can't traverse all the elements in single run only. Whereas in non-linear data structure, multiple levels are involved, then, we can traverse all the elements.
Examples of none-linear data structure are trees and graphs.

ADTs are entities that are definitions of data and operation but not have implements details.

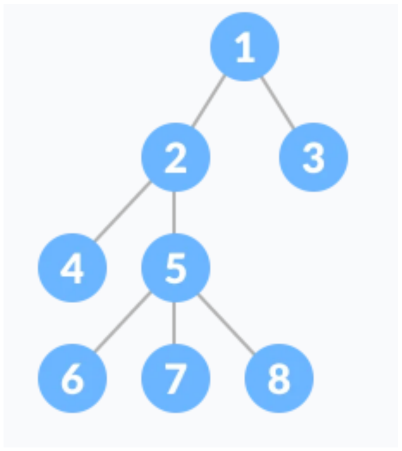


- store a set of elements of int type
- read elements by postion i.e index
- modify elements by index
- perform sorting

```
int arr[5] = {1,2,3,4,5};
cout<<arr[1];
arr[2]=10;
```

Tree Data Structure

A tree (upside down) is a nonlinear hierarchical data structure that consists of nodes connected by edges with parent-child relationship.



- Each node (except the top node) has a parent and zero or more children nodes.
- Each node has data in any type as char, number, and string.
- First node that starts with the tree called root.

Tree Terminologies

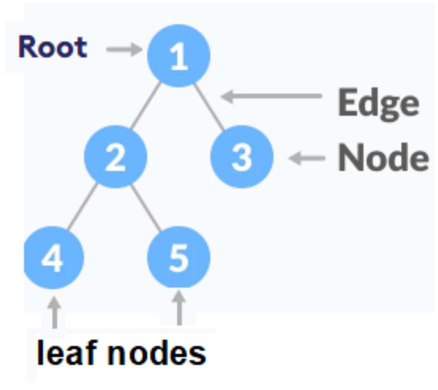
Node is an entity that contains a value and pointers to its child nodes.

Leaf node (also called as external nodes) is the node which does not have a child.

Non-leaf (also called as internal node) is a node with at least one child.
[The root node is also said to be Internal Node]

Edge is the link between any two nodes.

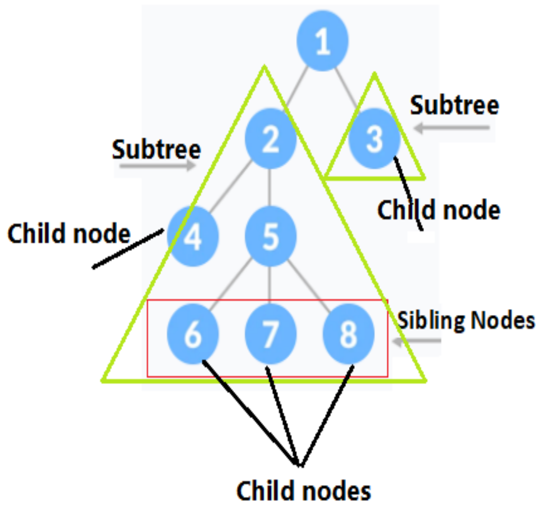
Root is the topmost node of a tree (without any parent).



Sibling Nodes: nodes that have the same parent.

Ancestor node (N): any predecessor node on a path from node N to root. [The root node doesn't have any ancestors. **[2,1 are ancestor of 5]**

Descendant node (N): any successor node on a path from node N to leaf **[6, 7, 8 are descendant of 5]**



Child node: If the node is a descendant of any node, then the node is known as a child node.

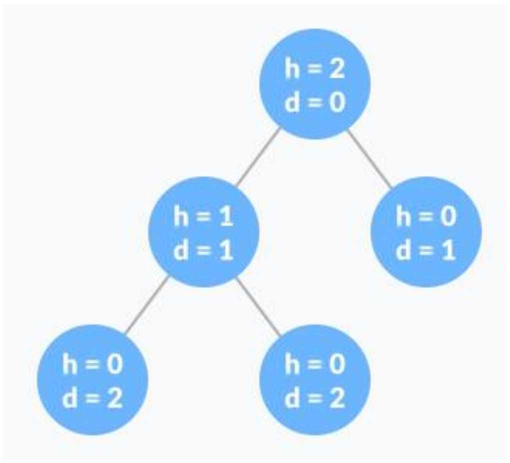
Parent: If the node contains any sub-node, then that node is said to be the parent of that sub-node. **[5 is parent of 6,7,8]**

Subtree : tree consisting of a node and its descendants

Path: the sequence of nodes from one node (source node) to another node(destination node).

Height of a Node N: is the longest path from node N to leaf. **[Height of leaf node = 0]**

Depth of a Node N: is the number of edges from the root to the node N. In other words: it is the number of nodes it passes from root through down to node N. **[depth of root node = 0]**

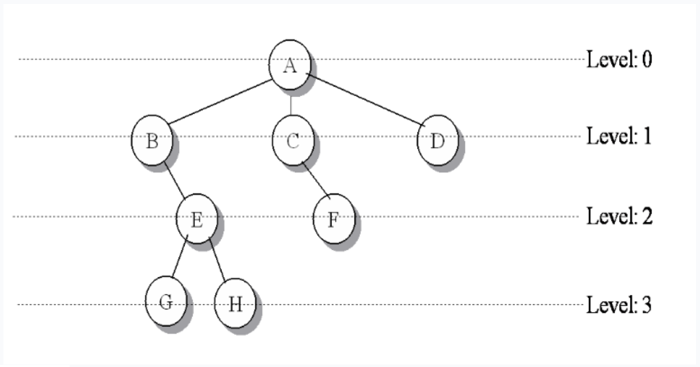


Height of a Tree: is height of the root node.

Depth of a Tree: the total number of edges from root node to a leaf node in the longest path.[maximum depth of any node]

Degree of a Node N: **number of children of that node**. **[degree of leaf =0]**

Degree of a Tree: maximum degree among of nodes.



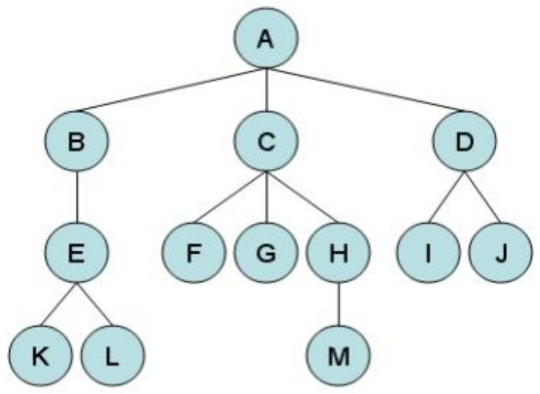
Level

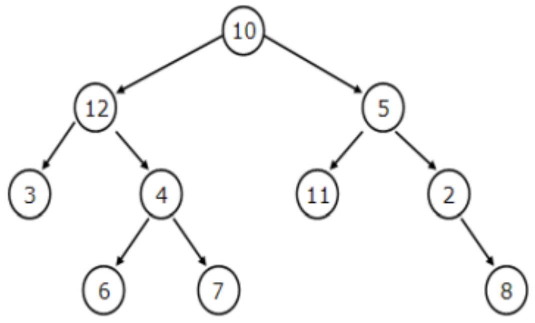
the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on...

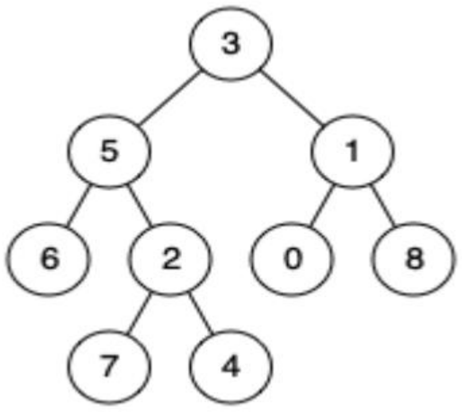
Depth of each node in any level equal to that level number.

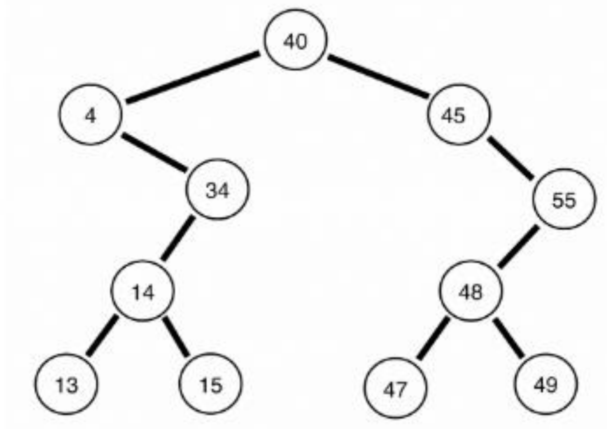
Question: Consider the trees below.

1. Which node is the root?
2. What are the internal nodes?
3. How many descendants does node (?) have?
4. What is the depth of node (?)?
5. What are the internal nodes?
6. How many descendants does node (?) have?
7. How many ancestors does node (?) have?
8. What is the depth of node (?)?
9. What are the siblings of node (?)?
10. Which nodes are in the subtree rooted at node (?)?
11. What is the height of the tree?





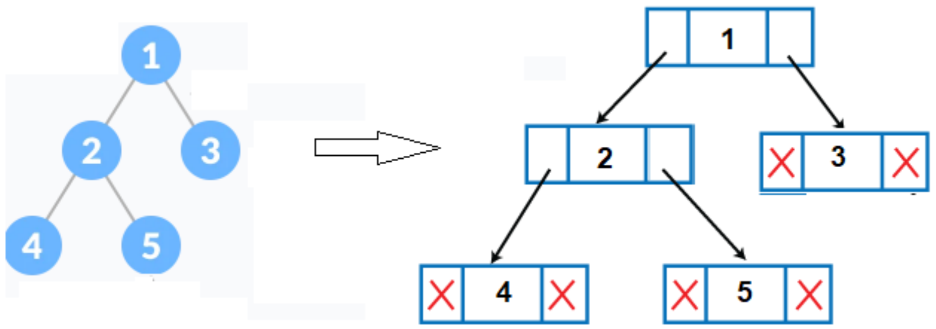




A Linked Structure for Tree

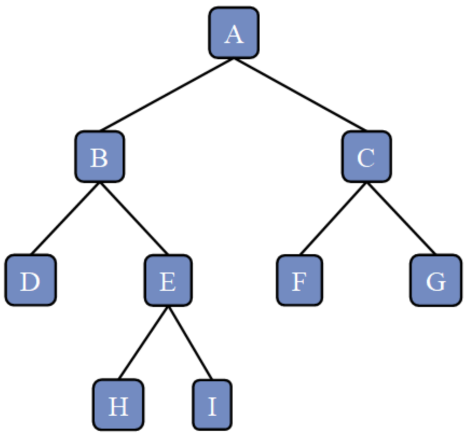
The tree data structure can be created by creating the nodes dynamically with linked list. A tree node is represented by an object storing

- Element
- A parent node
- A sequence of children nodes



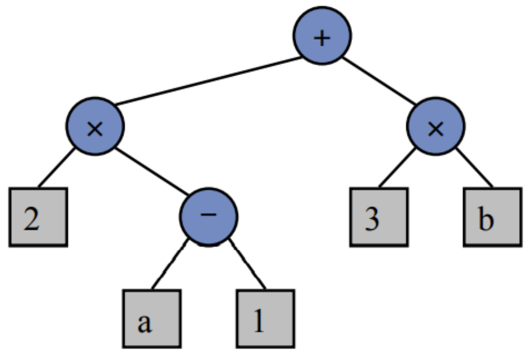
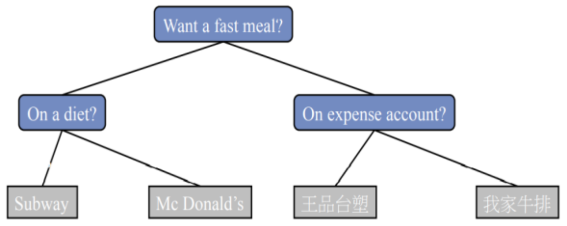
Binary Tree

Binary tree is a tree data structure in which each parent node can have at most two children. [Children ordered pair (left and right].

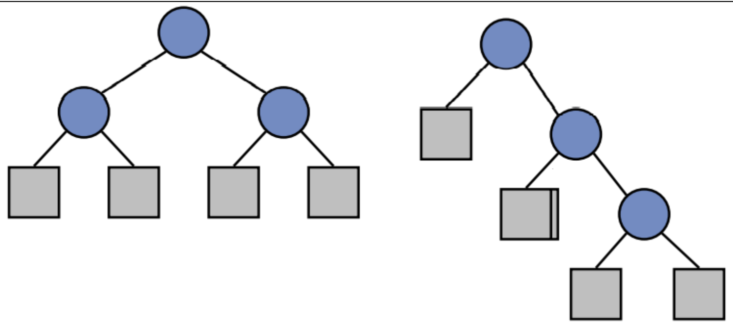
A binary tree is a tree with the following properties: ▪ Each internal node has at most two children (left child and right child) ▪ The children of a node are an ordered pair (left and right)	

Applications of Binary Tree

1. To build Arithmetic Expression Tree
2. To build Decision Tree

Binary tree associated with an arithmetic expression - internal nodes: operators - external nodes: operands Example : arithmetic tree for the expression : (2 X (a - 1) + (3 X b))	
Binary tree associated with a decision process - internal nodes: questions with yes/no answer - external nodes: decisions Example: dining decision	 Want a fast meal? On a diet? On expense account? Subway Mc Donald's 王品台塑 我家牛排

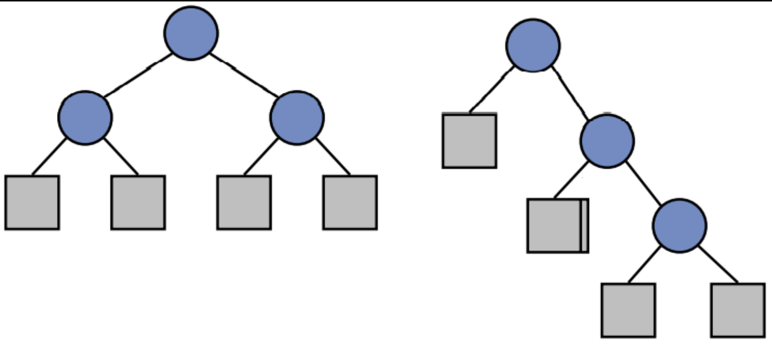
Proper Binary Trees

Each internal node has exactly 2 children	

- n :number of total nodes - e :number of external nodes - i :number of internal nodes - h :height (maximum depth of a node)	Properties: **1.** $e = i + 1$ **2.** $n = 2e - 1$ **3.** $h \le i$ **4.** $h \le (n - 1)/2$ **5.** $e \le 2^h$ 6. $h \ge \log_2 e$ **7.** $h \ge \log_2 (n + 1) - 1$

from the two tress:
n = 7 , e = 4 , i = 3, h=2

- ✓ $e = i + 1$
- ✓ $n = 2e - 1$
- ✓ $h <= i$
- ✓ $h <= (n - 1)/2$
- ✓ 5. $e <= 2^h$
- ✓ 6. $h >= \log_2 e=2$
- ✓ 7. $h >= \log_2 (n + 1) - 1=2$



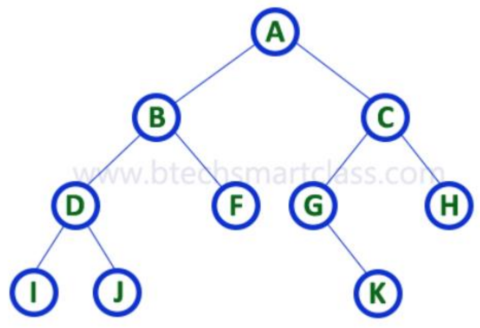
Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows:

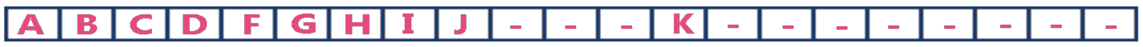
1. **Array Representation**
2. **Linked List Representation**

Array Representation

Consider the following binary tree:



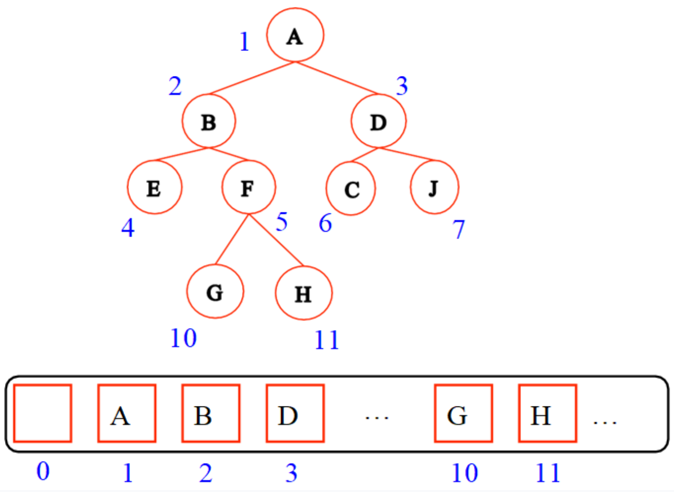
In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.
Consider the above example of a binary tree and it is represented as follows:



A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

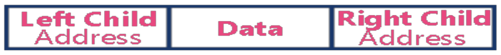
- A[0] is always empty
- A[i] is empty if there is no node in the ith position
- The array size N is 2(h+1)

Consider the following binary tree:

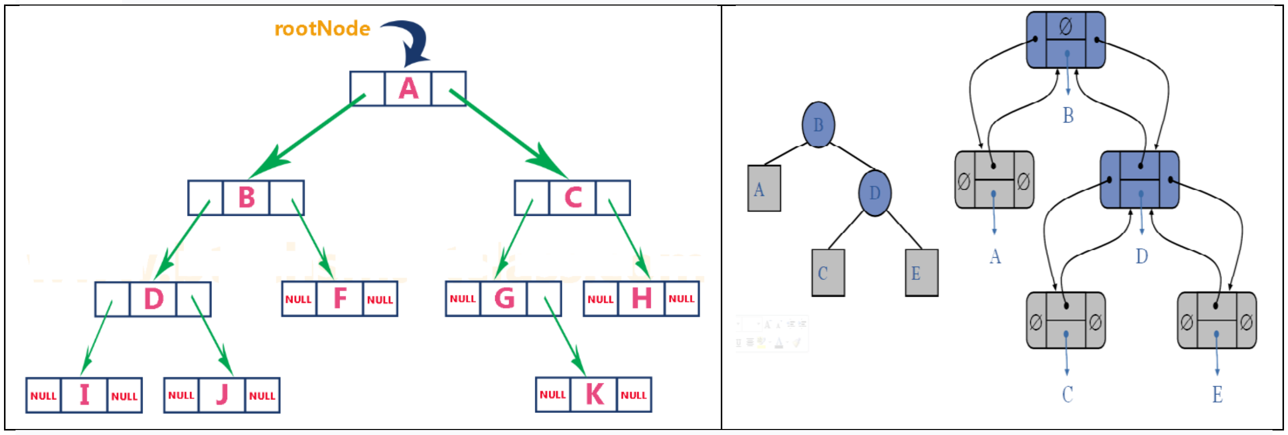


Linked List Representation

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.
In this linked list representation, a node has the following structure...



The above example of the binary tree represented using Linked list representation is shown as follows:

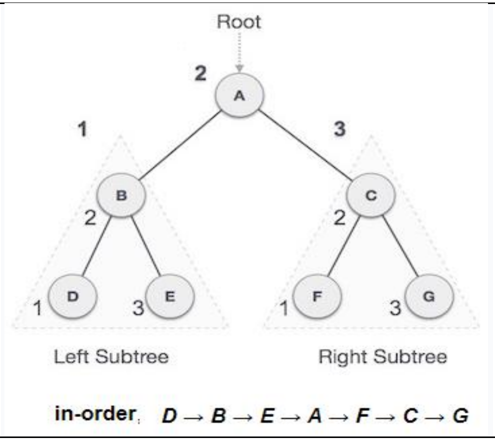
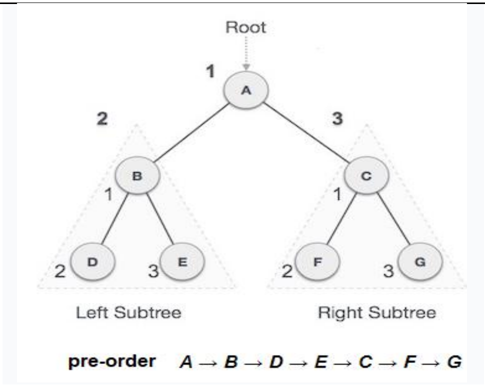
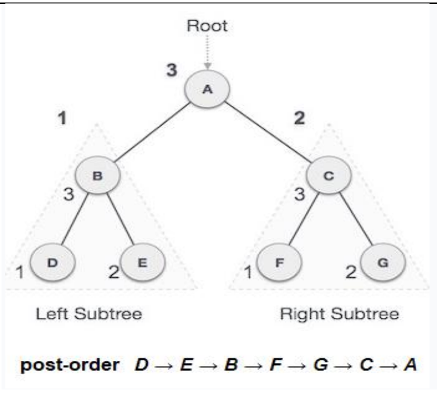


Operations of Tree

1. Insert
2. Delete
3. Search
4. Traversal

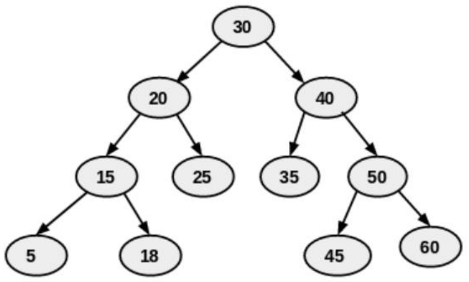
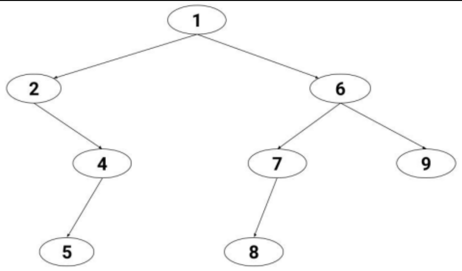
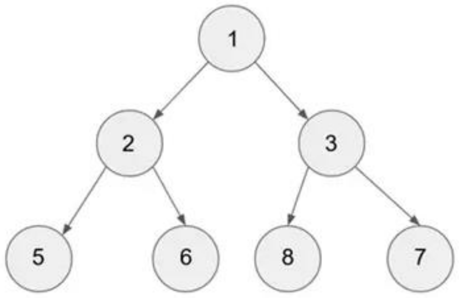
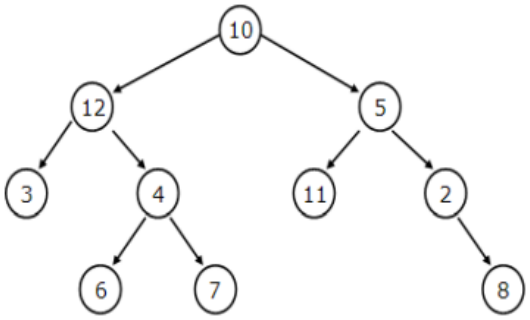
Tree Traversal

- In order to perform any operation on a tree, you need to reach to the specific node.
- Traversing a tree means visiting every node in the tree.
- The Types of tree traversal are:
  1. **inorder**
  2. **preorder**
  3. **postorder.**

Inorder traversal	**Pre-order traversal**	**Post-order traversal**
1. visit all the nodes in the left subtree 2. Visit the root node 3. Visit all the nodes in the right subtree	1. Visit the root node 2. Visit all the nodes in the left subtree 3. Visit all the nodes in the right subtree	1. Visit all the nodes in the left subtree 2. Visit all the nodes in the right subtree 3. Visit the root node
 Root Left Subtree Right Subtree in-order D → B → E → A → F → C → G	 Root Left Subtree Right Subtree pre-order A → B → D → E → C → F → G	 Root Left Subtree Right Subtree post-order D → E → B → F → G → C → A

Consider the trees below.

1. What is the preorder traversal of the tree?
2. What is the inorder traversal of the tree?
3. What is the postorder traversal of the tree?

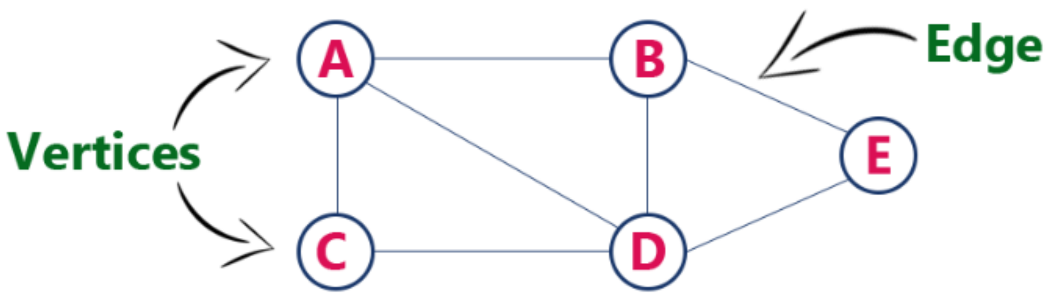
 30 20 40 15 25 35 50 5 18 45 60	 1 2 6 4 7 9 5 8
 1 2 3 5 6 8 7	 10 12 5 3 4 11 2 6 7 8

Lecture #04

Graphs

What is a graph?

- Graph is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges that relate the nodes to each other.
- The set of edges describes relationships among the vertices.



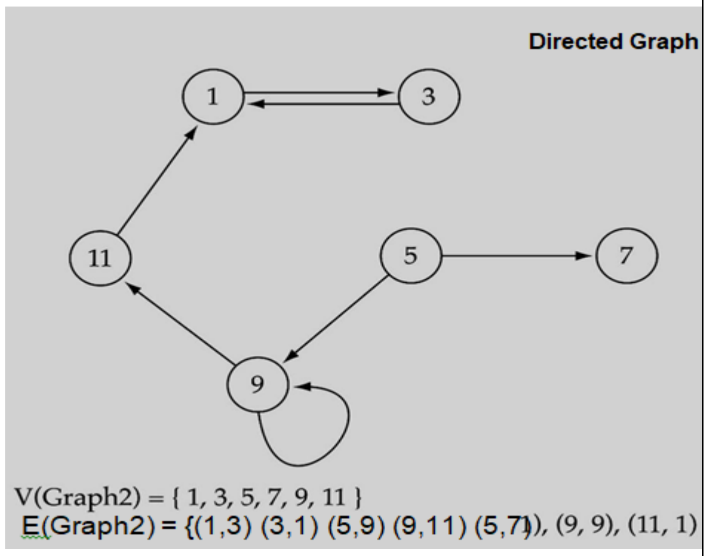
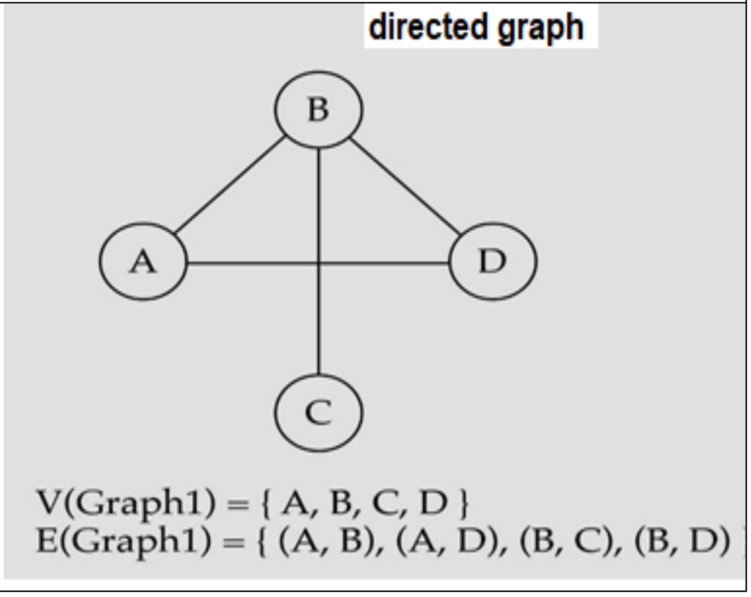
Formal definition of Graphs

Generally, a graph **G** is represented as **G = (V , E)**, where **V is set of vertices** and **E is set of edges**. In the above figure the following is a graph with 5 vertices and 7 edges.

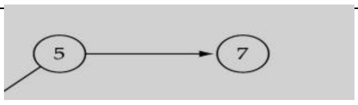
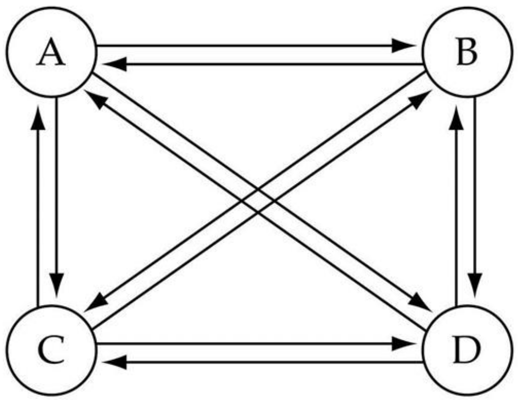
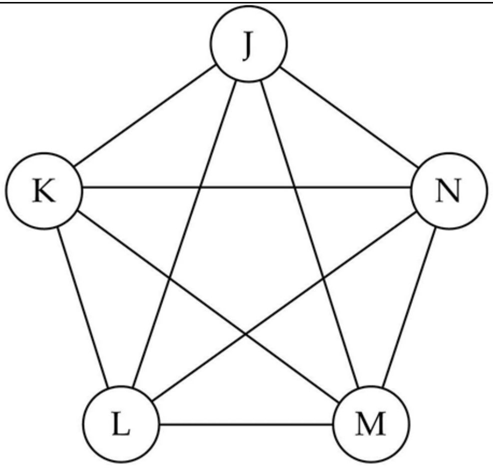
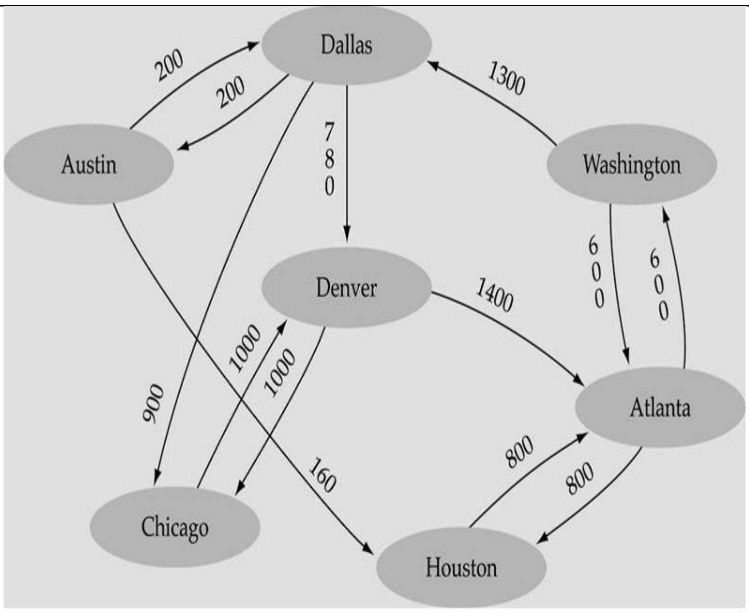
V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.

Directed vs undirected graphs

Directed	Undirected graphs
When the edges in a graph have a direction, the graph is called directed (or *digraph*) **Warning**: if the graph is directed, the order of the vertices in each edge is important	When the edges in a graph have no direction, the graph is called *undirected*

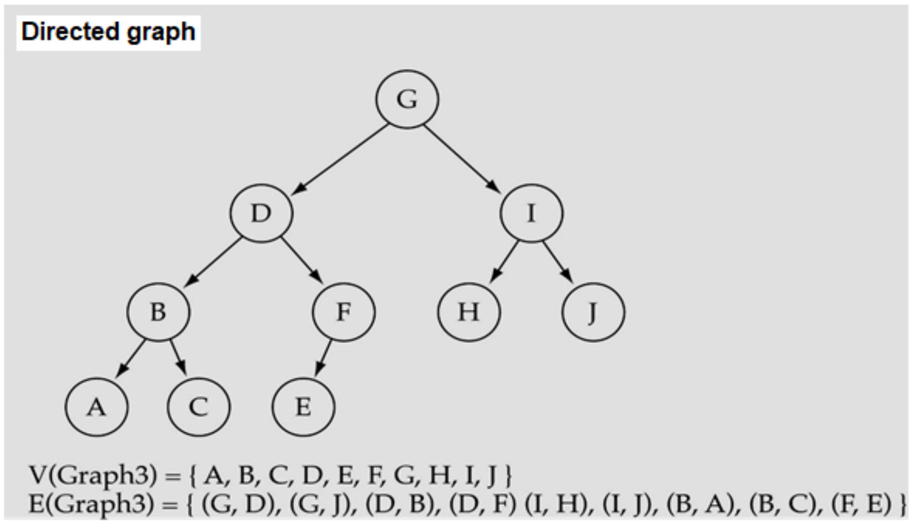

V(Graph2) = { 1, 3, 5, 7, 9, 11 }
E(Graph2) = {(1,3) (3,1) (5,9) (9,11) (5,7), (9, 9), (11, 1)
	
	V(Graph1) = { A, B, C, D }
	E(Graph1) = { (A, B), (A, D), (B, C), (B, D)

Graph terminology

Adjacent node: two nodes are adjacent if they are connected by an edge. 5 is adjacent to 7 7 is adjacent from 5	
Path: a sequence of edges connect a sequence of vertices.	
Complete graph: a graph in which every vertex is directly connected to every other vertex.	
The number of edges in a complete directed graph with N vertices are: *N * (N-1)*	
The number of edges in a complete undirected graph with N vertices are: *N * (N-1) /2*	
Weighted graph: a graph in which each edge carries a value.	 Dallas Austin Washington Denver Atlanta Chicago Houston 200 200 1300 780 1400 600 600 900 1000 1000 160 800 800

Trees vs graphs

Trees are special cases of graphs



V(Graph3) = { A, B, C, D, E, F, G, H, I, J }
E(Graph3) = { (G, D), (G, J), (D, B), (D, F) (I, H), (I, J), (B, A), (B, C), (F, E) }

Graph Representing

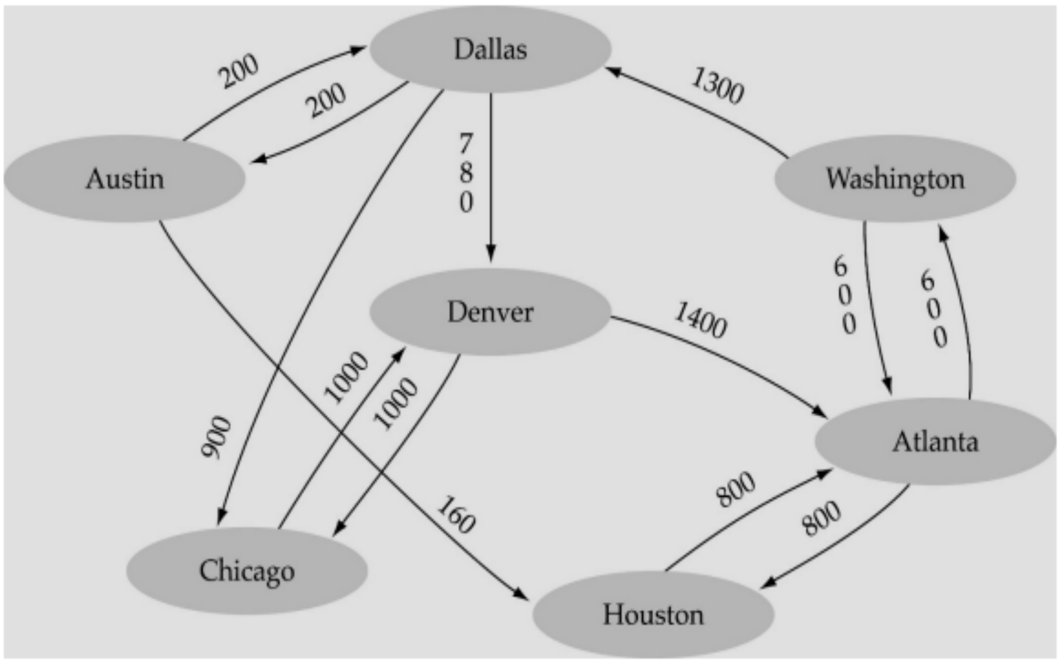
The two most common ways of representing graphs are:

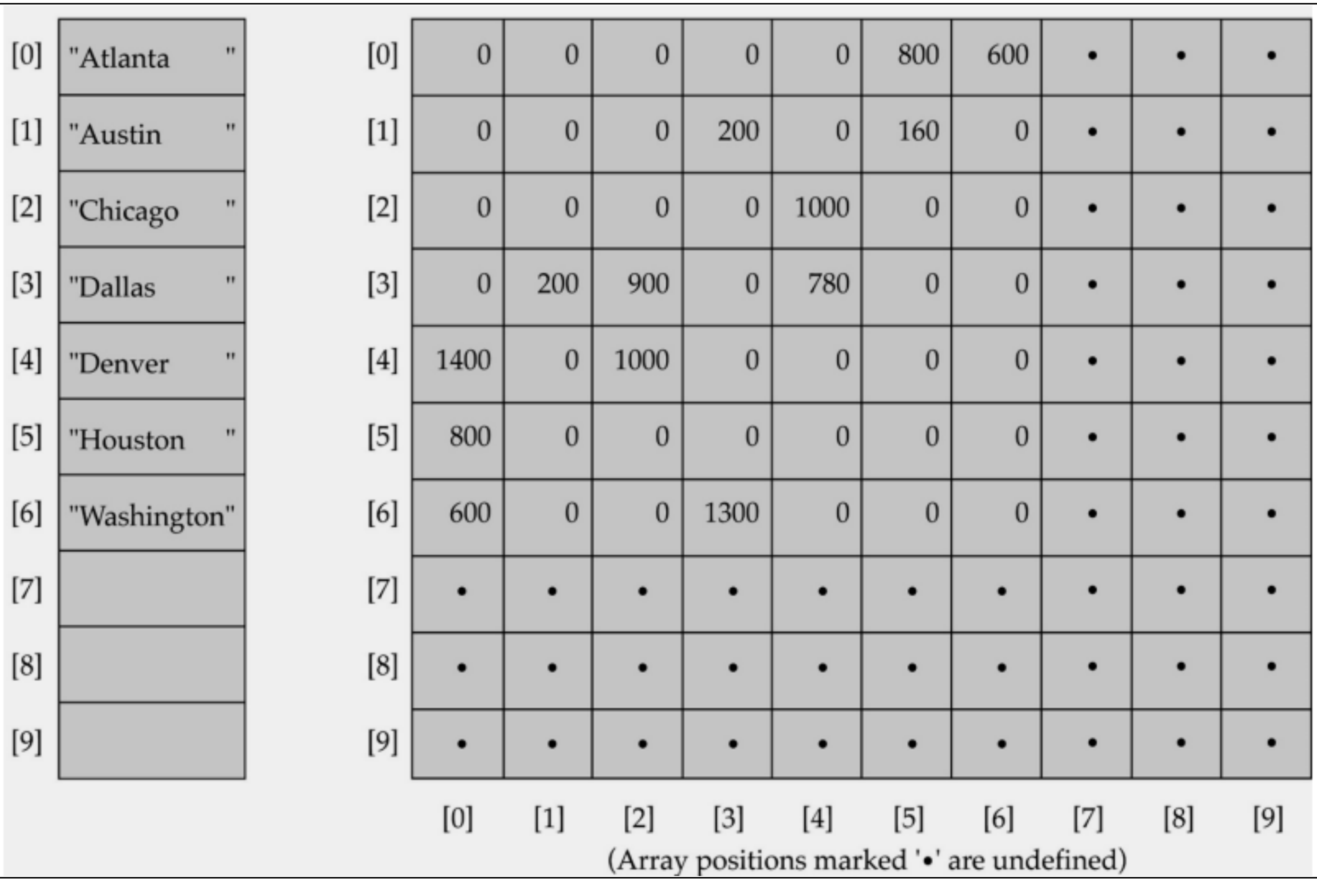
1. Adjacency matrix

2. Adjacency List

In Adjacency matrix way use array.

- A 1D array is used to represent the vertices.
- A 2D array (adjacency matrix) is used to represent the edges.



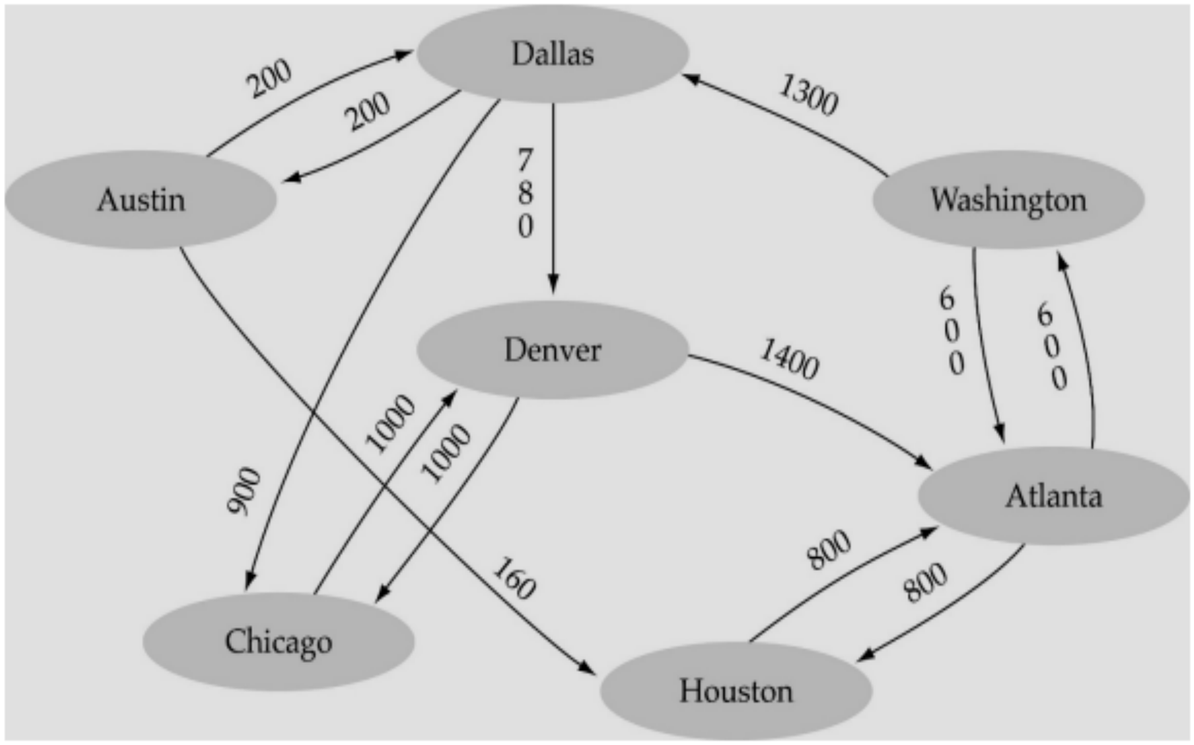


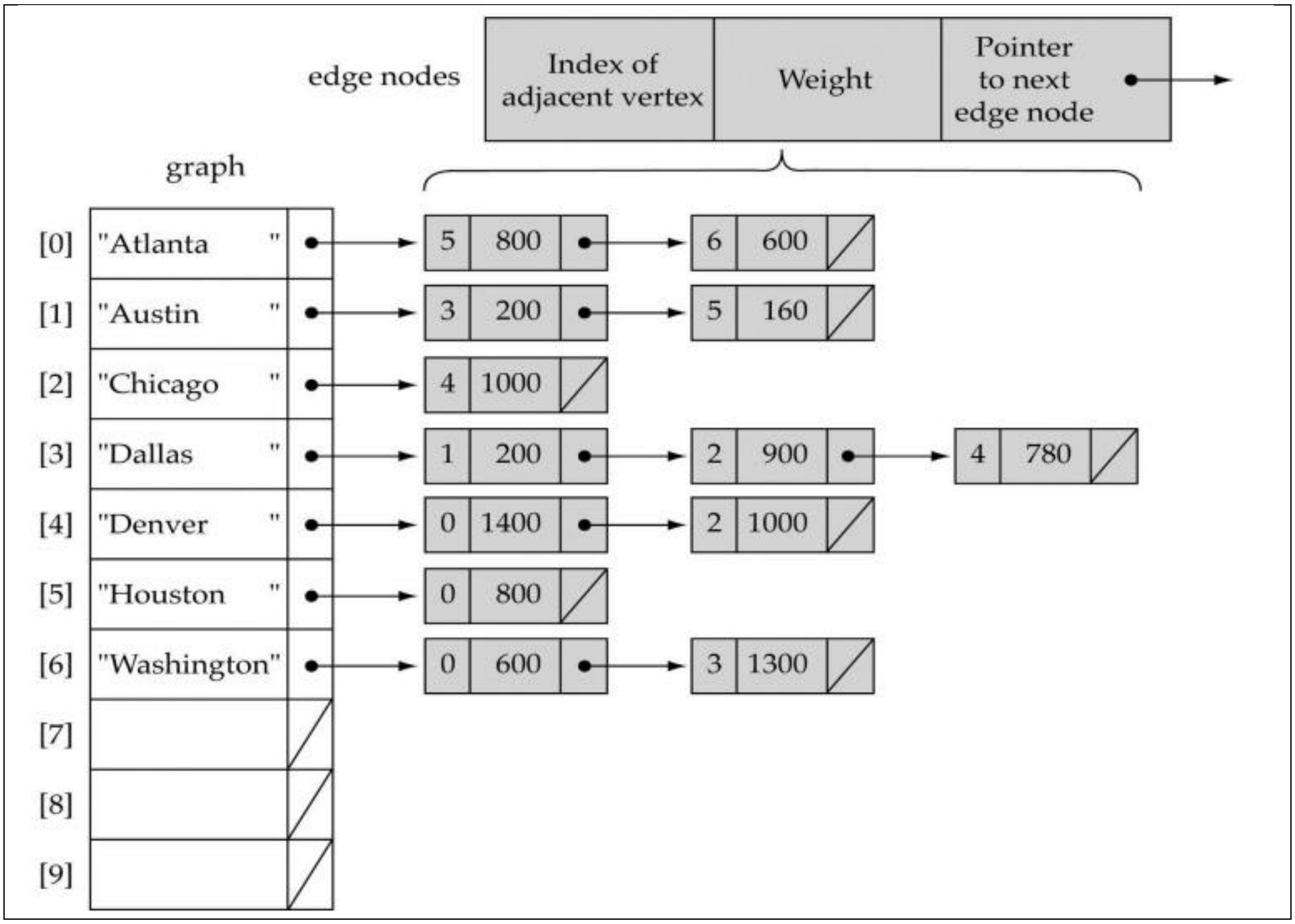
			[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	"Atlanta "	[0]	0	0	0	0	0	800	600	•	•	•
[1]	"Austin "	[1]	0	0	0	200	0	160	0	•	•	•
[2]	"Chicago "	[2]	0	0	0	0	1000	0	0	•	•	•
[3]	"Dallas "	[3]	0	200	900	0	780	0	0	•	•	•
[4]	"Denver "	[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	"Houston "	[5]	800	0	0	0	0	0	0	•	•	•
[6]	"Washington"	[6]	600	0	0	1300	0	0	0	•	•	•
[7]		[7]	•	•	•	•	•	•	•	•	•	•
[8]		[8]	•	•	•	•	•	•	•	•	•	•
[9]		[9]	•	•	•	•	•	•	•	•	•	•

(Array positions marked '•' are undefined)

In Adjacency list way use Linked-list

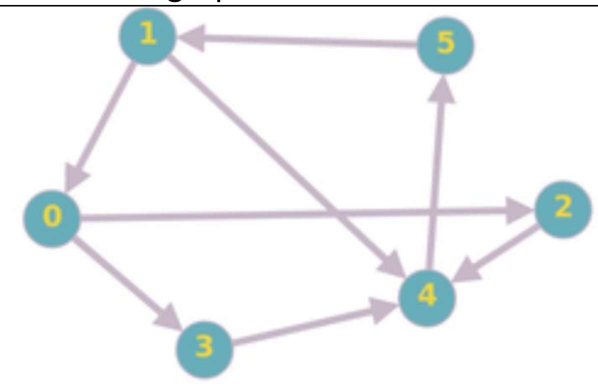
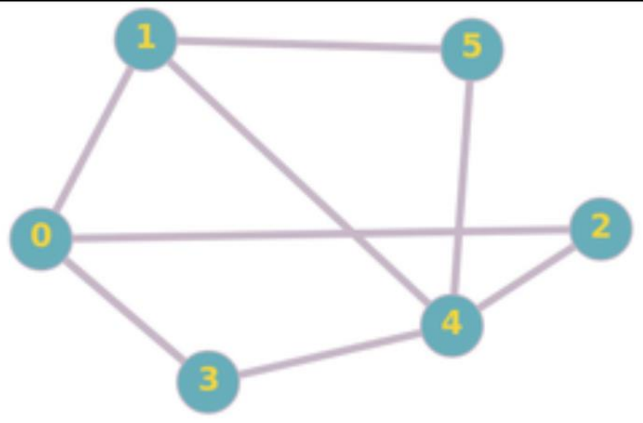
- A 1D array is used to represent the vertices
- A list is used for each vertex *v* which contains the vertex which are adjacent from *v* (adjacency list)




edge nodes
Index of adjacent vertex
Weight
Pointer to next edge node
graph
[0] "Atlanta "
[1] "Austin "
[2] "Chicago "
[3] "Dallas "
[4] "Denver "
[5] "Houston "
[6] "Washington"
[7]
[8]
[9]
5 800
6 600
3 200
5 160
4 1000
1 200
2 900
4 780
0 1400
2 1000
0 800
0 600
3 1300

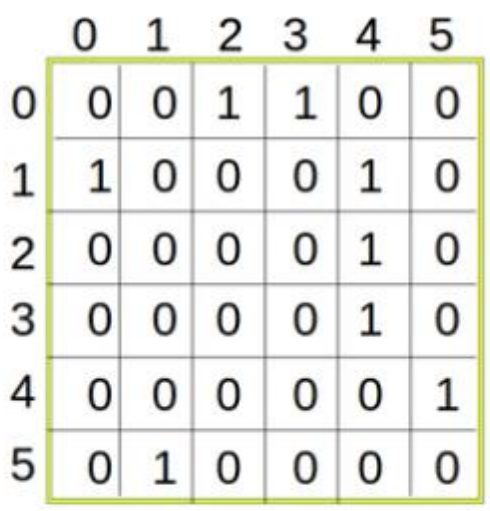
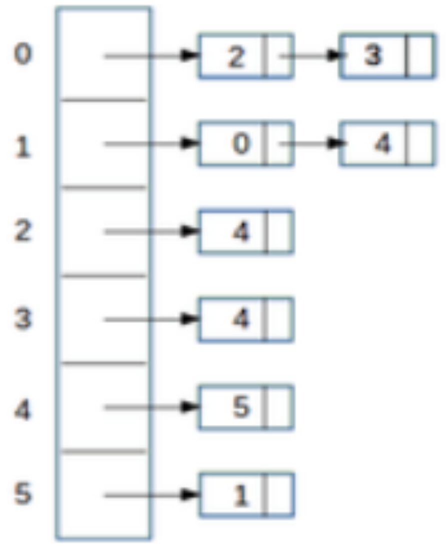
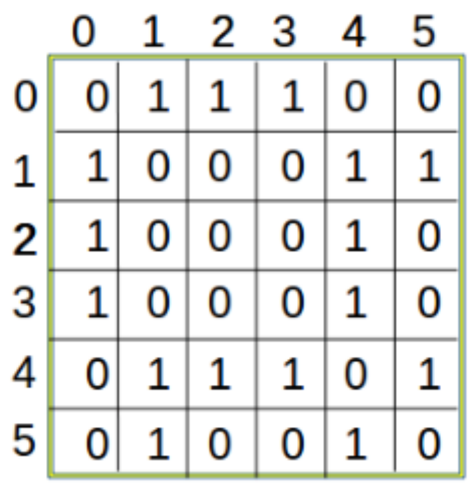
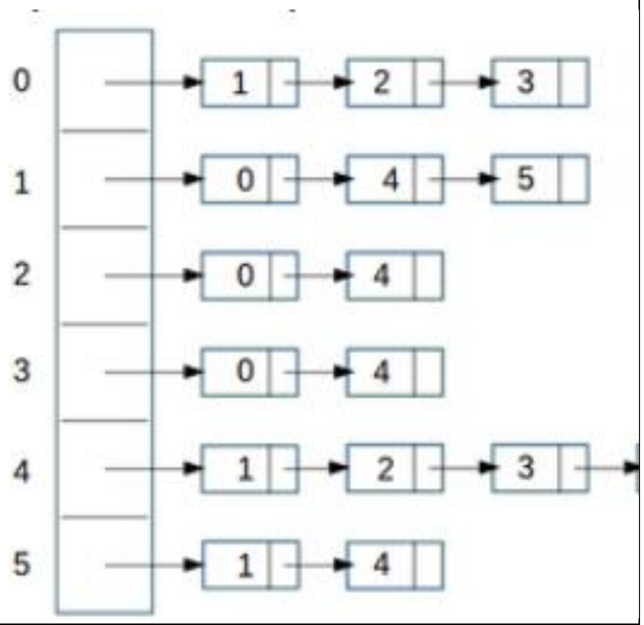
Example

Consider the graphs below.

 Graph1	 Graph2

1. Use an adjacency list to represent this graph.
2. Use an adjacency matrix to represent this graph

Answer:

	Adjacency matrix	Adjacency list
Graph1	 $\begin{array}{c\|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 1 \\ 5 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}$	
Graph2	 $\begin{array}{c\|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 2 & 1 & 0 & 0 & 0 & 1 & 0 \\ 3 & 1 & 0 & 0 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 1 & 0 & 1 \\ 5 & 0 & 1 & 0 & 0 & 1 & 0 \end{array}$	 0 → 1 → 2 → 3 1 → 0 → 4 → 5 2 → 0 → 4 3 → 0 → 4 4 → 1 → 2 → 3 → 5 → 1 → 4

Graph traversal

Graph traversal (also known as graph search) refers to the process of visiting (checking and/or updating) each vertex in a graph. Such traversals are classified by the order in which the vertices are visited.

There are two graph traversal techniques and they are as follows:

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

DFS (Depth First Search)

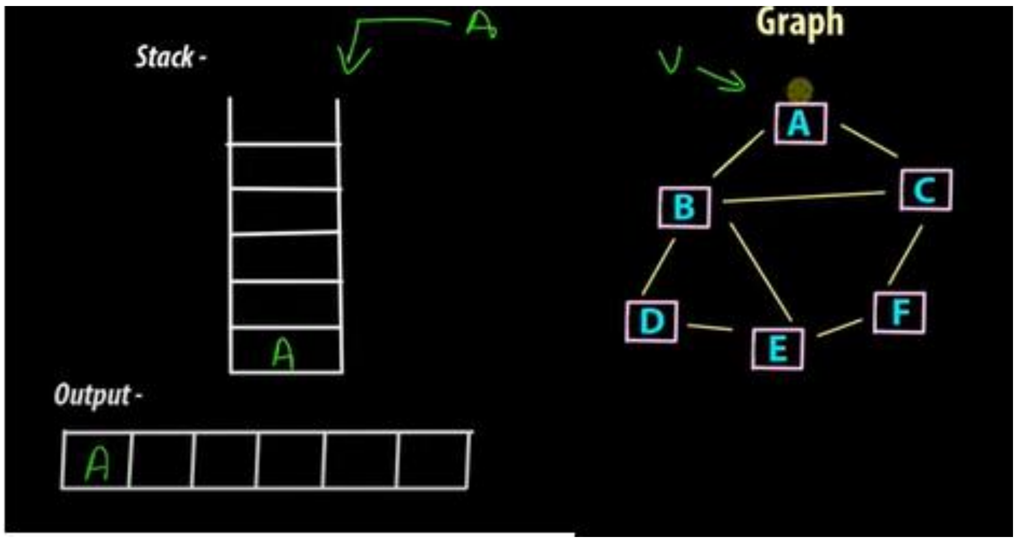
In DFS traversal, the **stack data structure** is used, which works on the LIFO (Last In First Out) principle. In DFS, traversing can be started from any node, or we can say that any node can be considered as a root node.

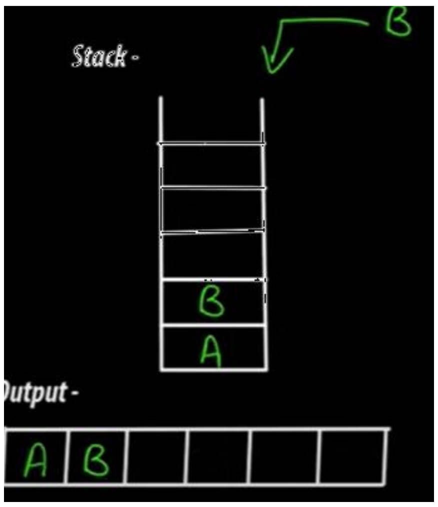
In this algorithm one starting vertex is given, and when an adjacent vertex is found, it moves to that adjacent vertex first and try to traverse in the same manner.

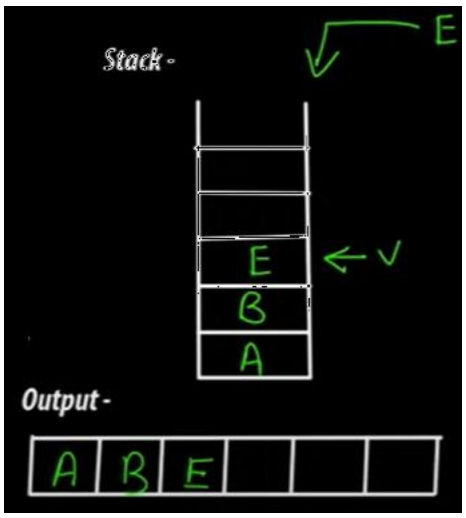
Steps:

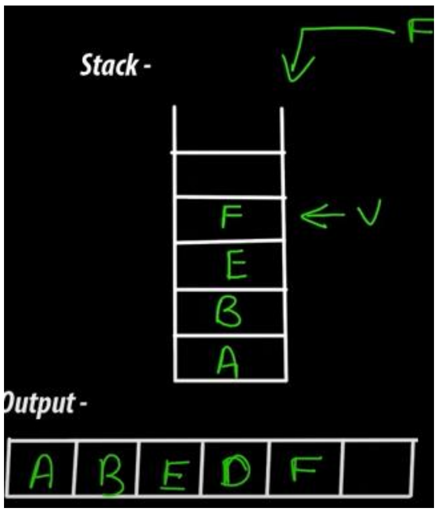
1. Define a Stack
2. Set current vertex V
3. Add current vertex V to stack
4. Print current vertex V
5. Add any 1 neighbor (unvisited i.e. not in stack previously) of V to stack
6. If current vertex has all its neighbors already visited, pop it from stack & backtrack
7. Check remaining vertices in the stack for any unvisited vertices.
8. Repeat from step 4 till stack empty

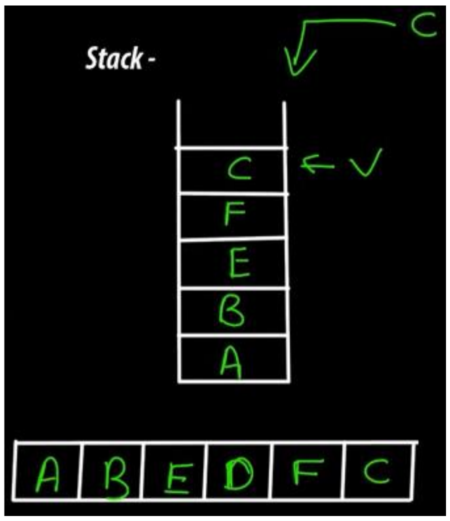
Example 1 : What is the possible DFS traversal for this graph? (Start with A).



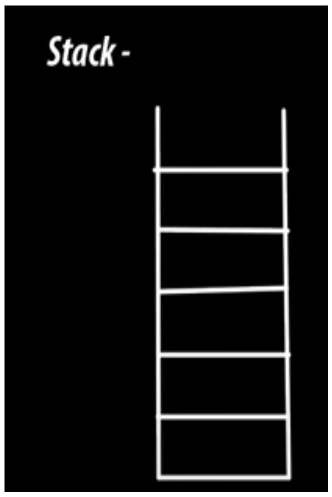




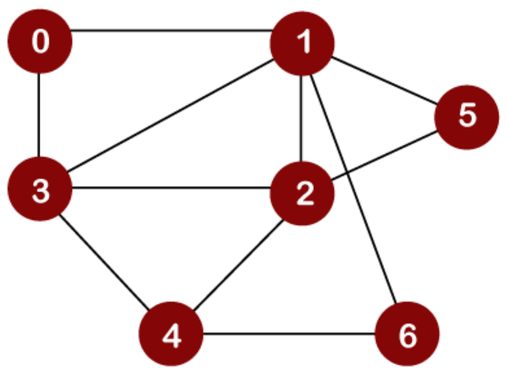


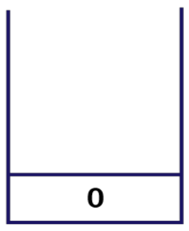
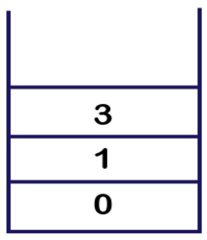
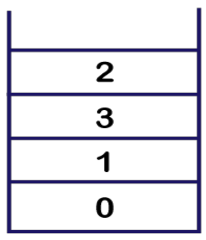
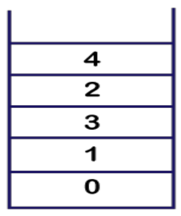
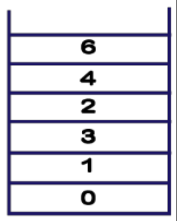
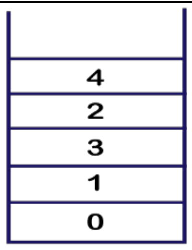
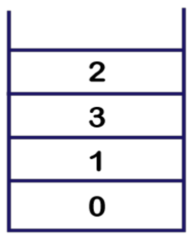
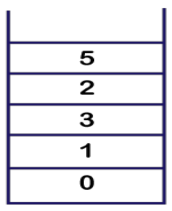
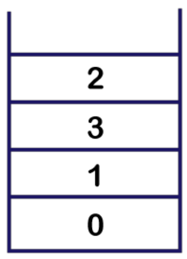
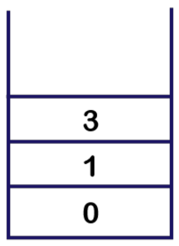
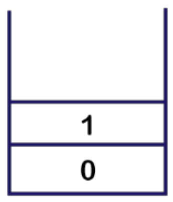
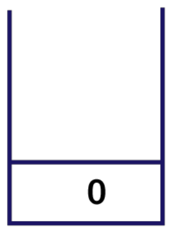
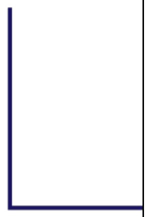


Stack After backtrack



Example 2: What is the possible DFS traversal for this graph? (Start with 0).



 Output : 0	 Output : 0 1 3	 Output : 0 1 3 2	 Output : 0 1 3 2 4	 Output : 0 1 3 2 4 6
 Output : 0 1 3 2 4 6	 Output : 0 1 3 2 4 6	 Output : 0 1 3 2 4 6 5	backtrack	
				 Empty

BFS (Breadth First Search)

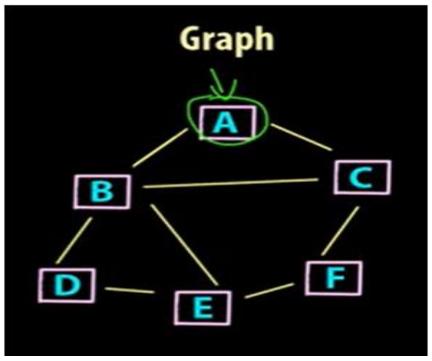
Stands for **BFS**. It is also known as **level order traversal**. The Queue data structure is used for the Breadth First Search traversal.

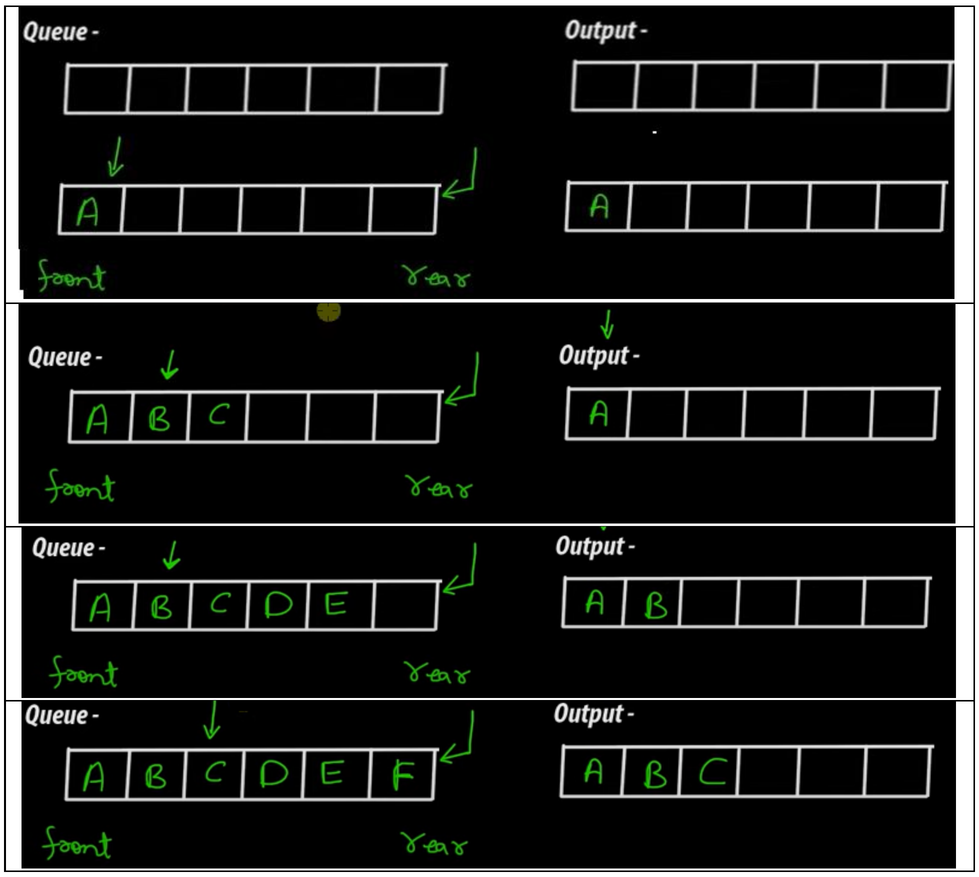
In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one. After completing all of the adjacent vertices, it moves further to check another vertex and checks its adjacent vertices again.

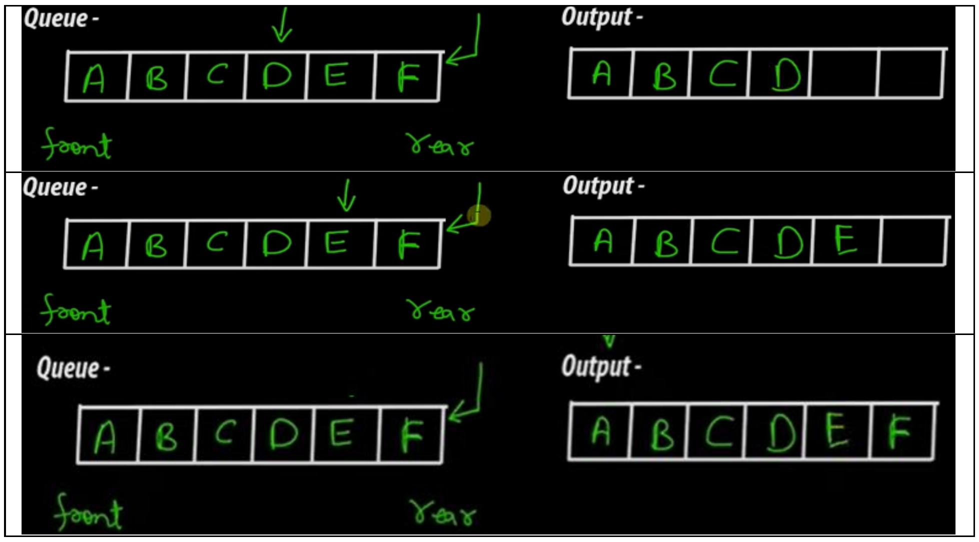
Steps:

1. Define a Queue
2. Set current vertex V
3. Add current vertex V to queue
4. Print current vertex V
5. Add all neighbors (unvisited i.e. previously not in queue) of V to queue (in any order)
6. Repeat from step 4 till queue empty

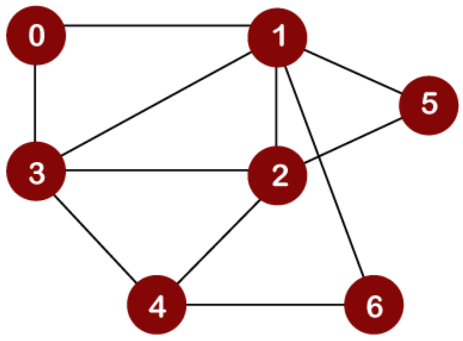
Example 1: What is the possible Breadth First traversal for this graph? (Start with A).

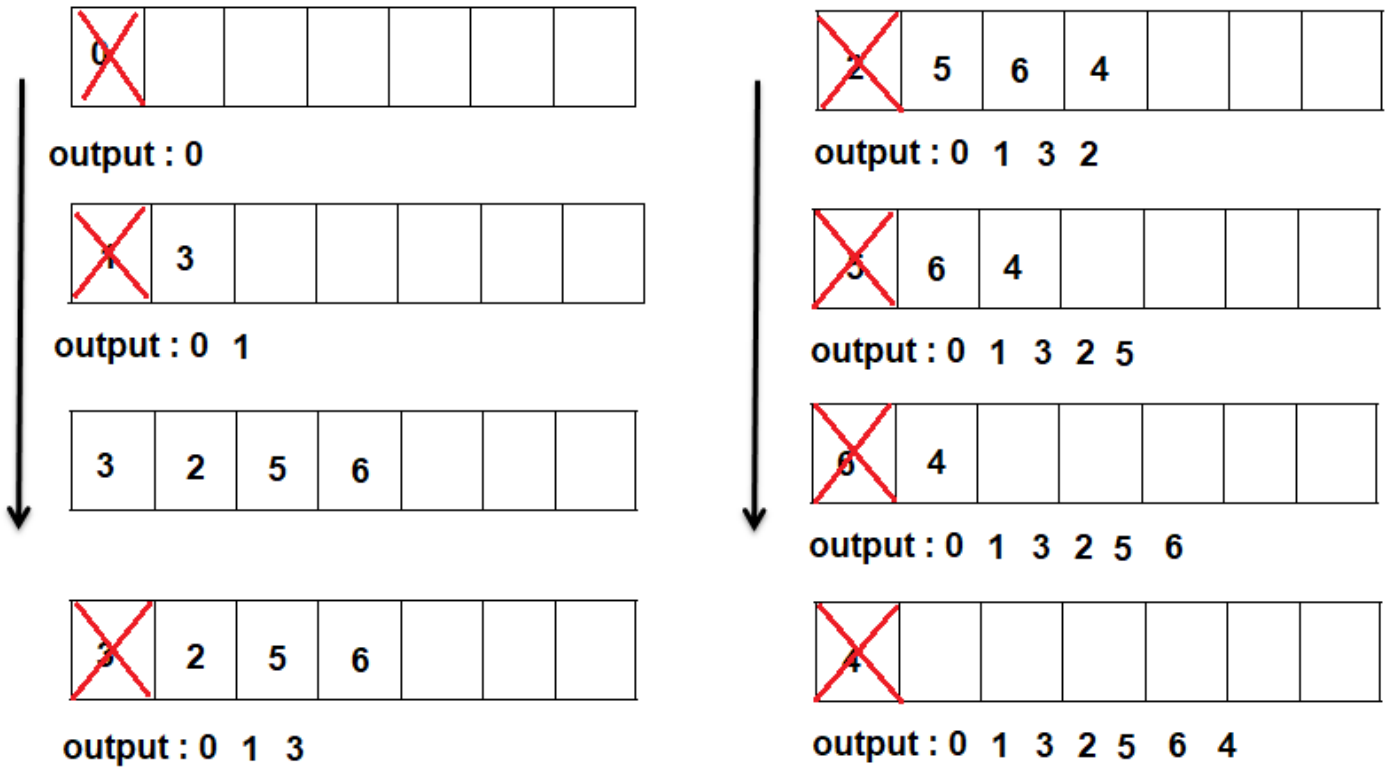






Example 2: What is the possible Breadth First traversal for this graph? (Start with 0).



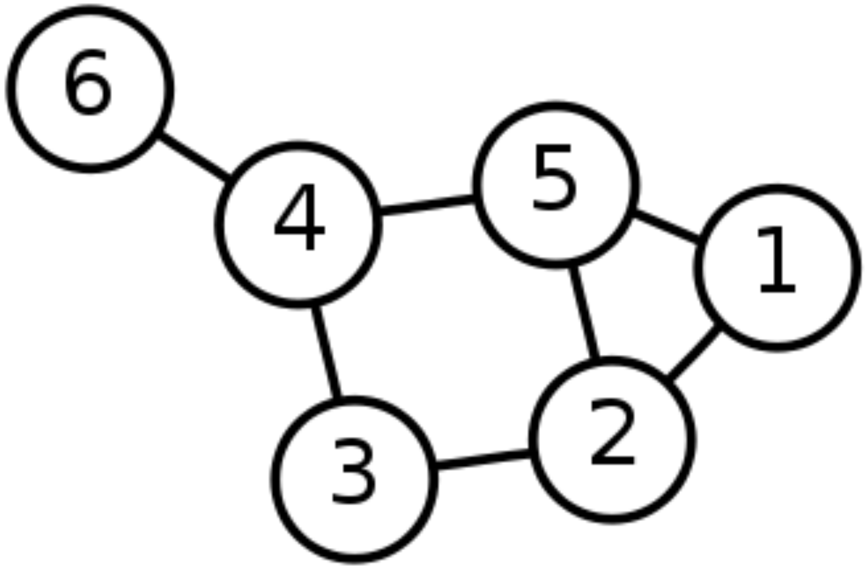


Lecture #05

Dijkstra's Algorithm

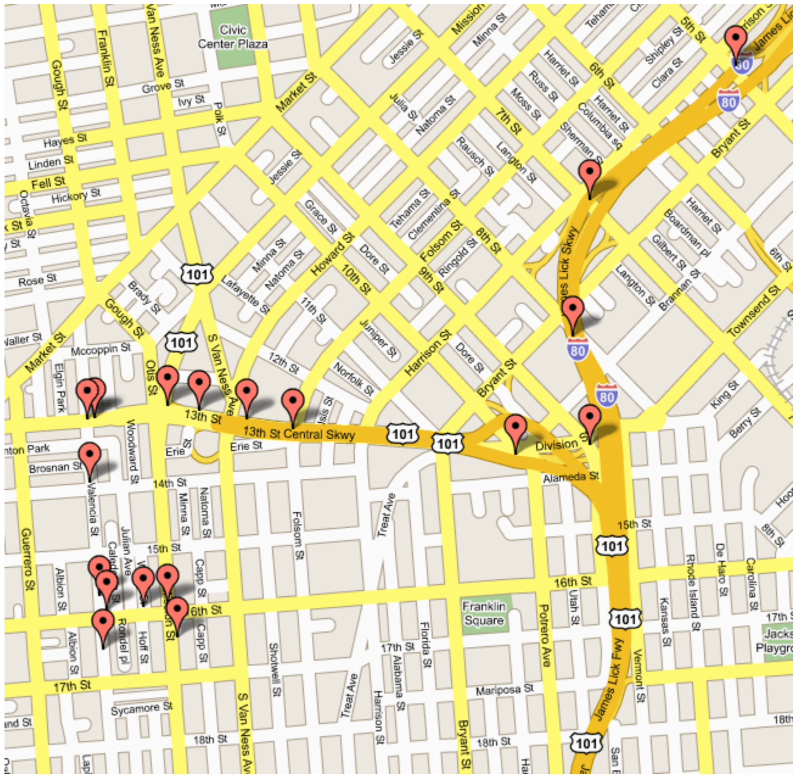
Single-Source Shortest Path Problem

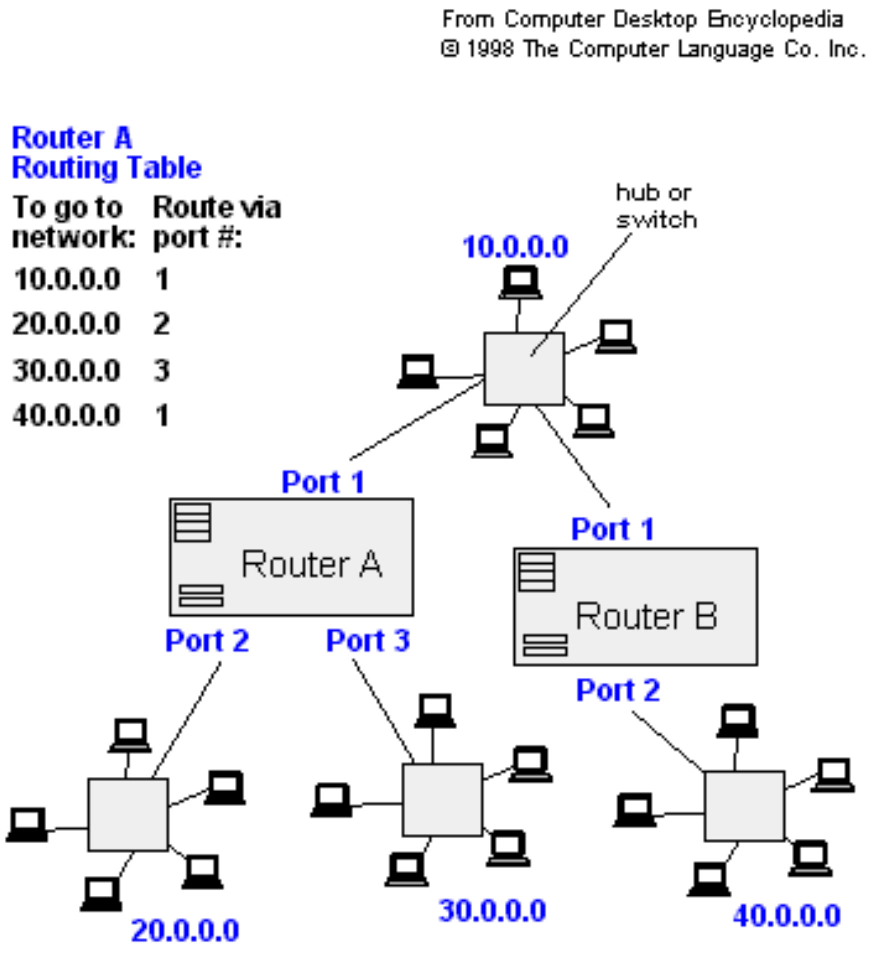
<u>**Single-Source Shortest Path Problem**</u> - The problem of finding shortest paths from a source vertex *v* to all other vertices in the graph.



Applications

- Maps (Map Quest, Google Maps)
- Routing Systems





To go to network:	Route via port #:
10.0.0.0	1
20.0.0.0	2
30.0.0.0	3
40.0.0.0	1

Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Input: Weighted graph G={E,V} and source vertex $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

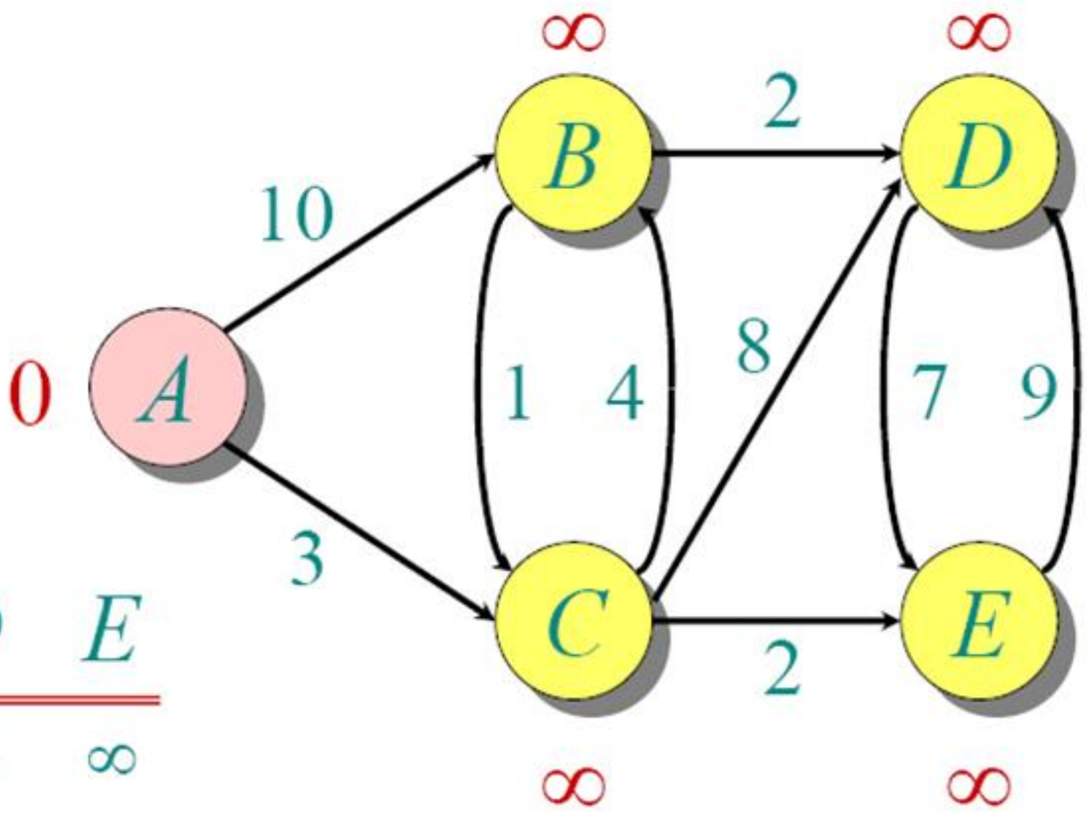
Approach

- The algorithm computes for each vertex u the distance to u from the start vertex v, that is, the weight of a shortest path between v and u.
- the algorithm keeps track of the set of vertices for which the distance has been computed, called the cloud C
- Every vertex has a label D associated with it. For any vertex u, D[u] stores an approximation of the distance between v and u. The algorithm will update a D[u] value when it finds a shorter path from v to u.
- When a vertex u is added to the cloud, its label D[u] is equal to the actual (final) distance between the starting vertex v and vertex u.

Dijkstra pseudocode

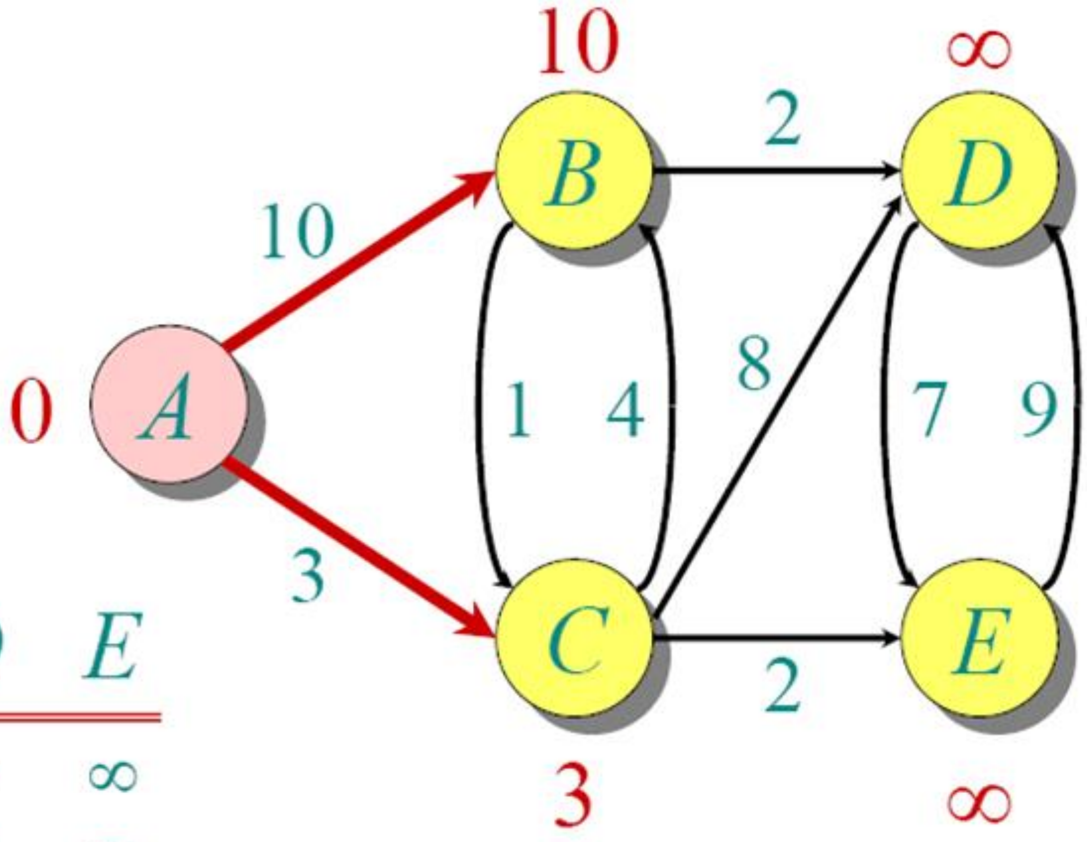
```
Dijkstra(v1, v2):
   for each vertex v:                    // Initialization
       v's distance := infinity.
       v's previous := none.
   v1's distance := 0.
   List := {all vertices}.

   while List is not empty:
       v := remove List vertex with minimum distance.
       mark v as known.
       for each unknown neighbor n of v:
           dist := v's distance + edge (v, n)'s weight.

           if dist is smaller than n's distance:
               n's distance := dist.
               n's previous := v.

   reconstruct path from v2 back to v1,
   following previous pointers.
```

Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

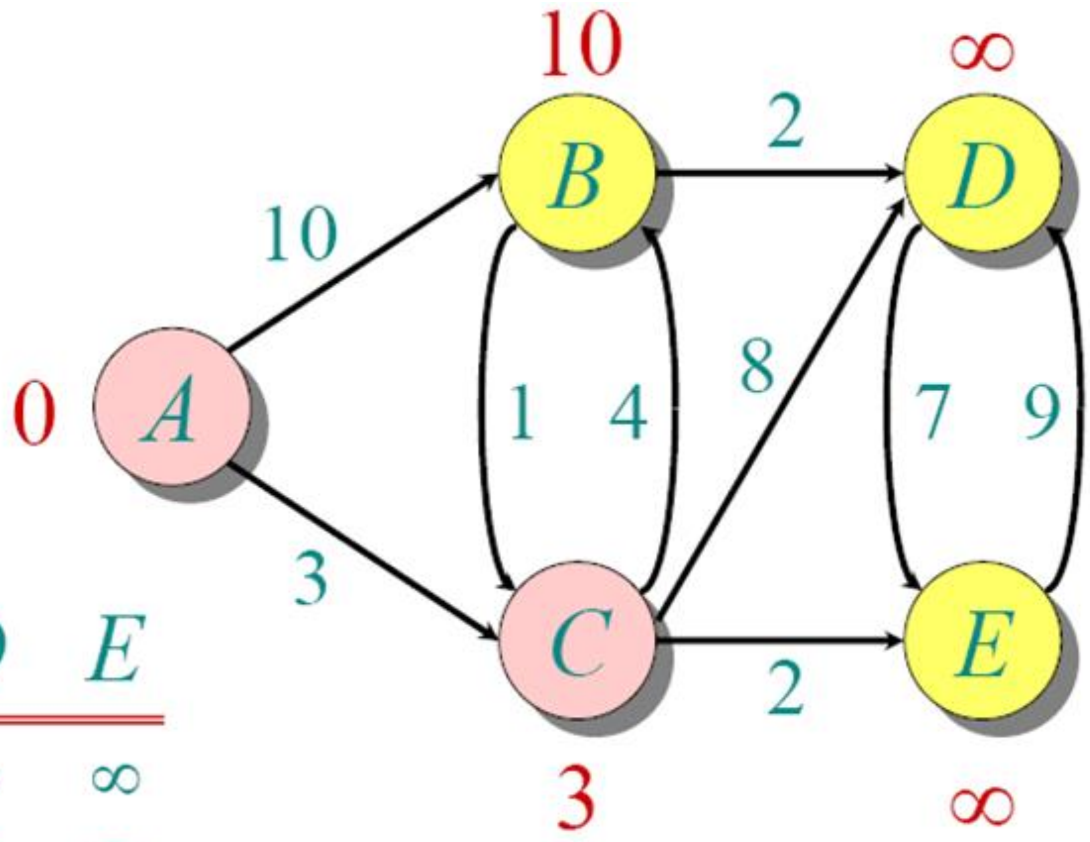
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

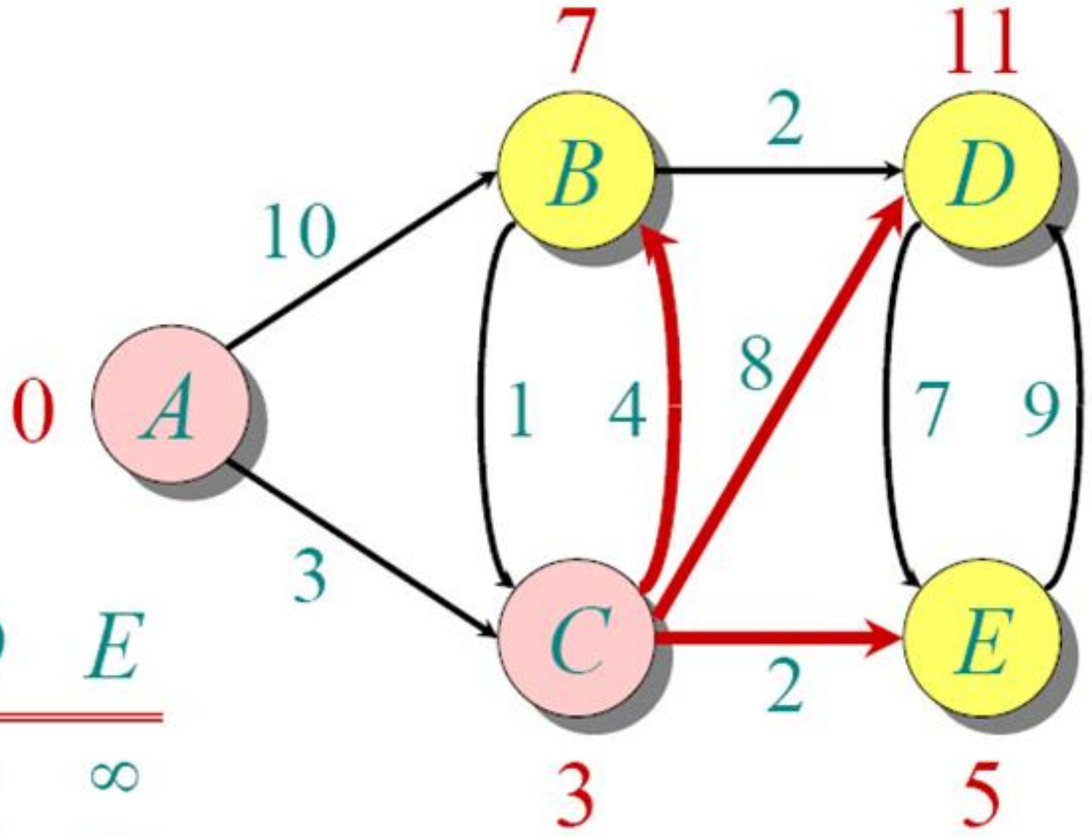
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { *A*, *C* }

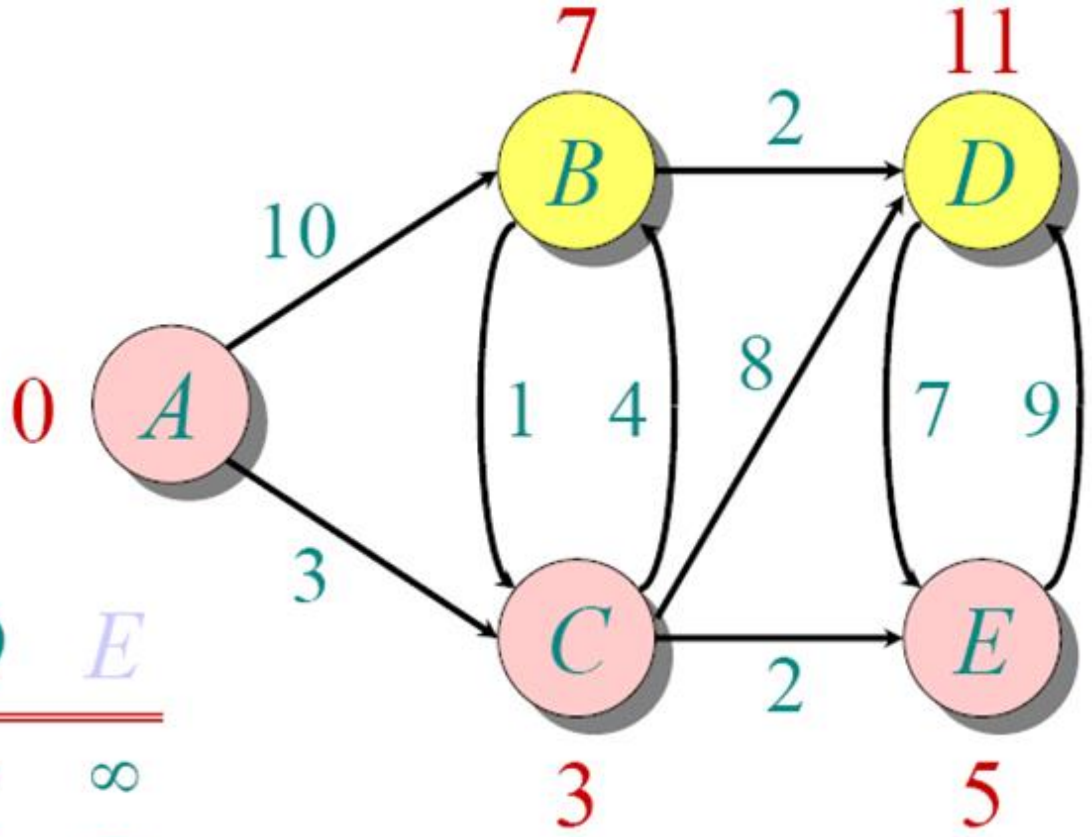
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C }

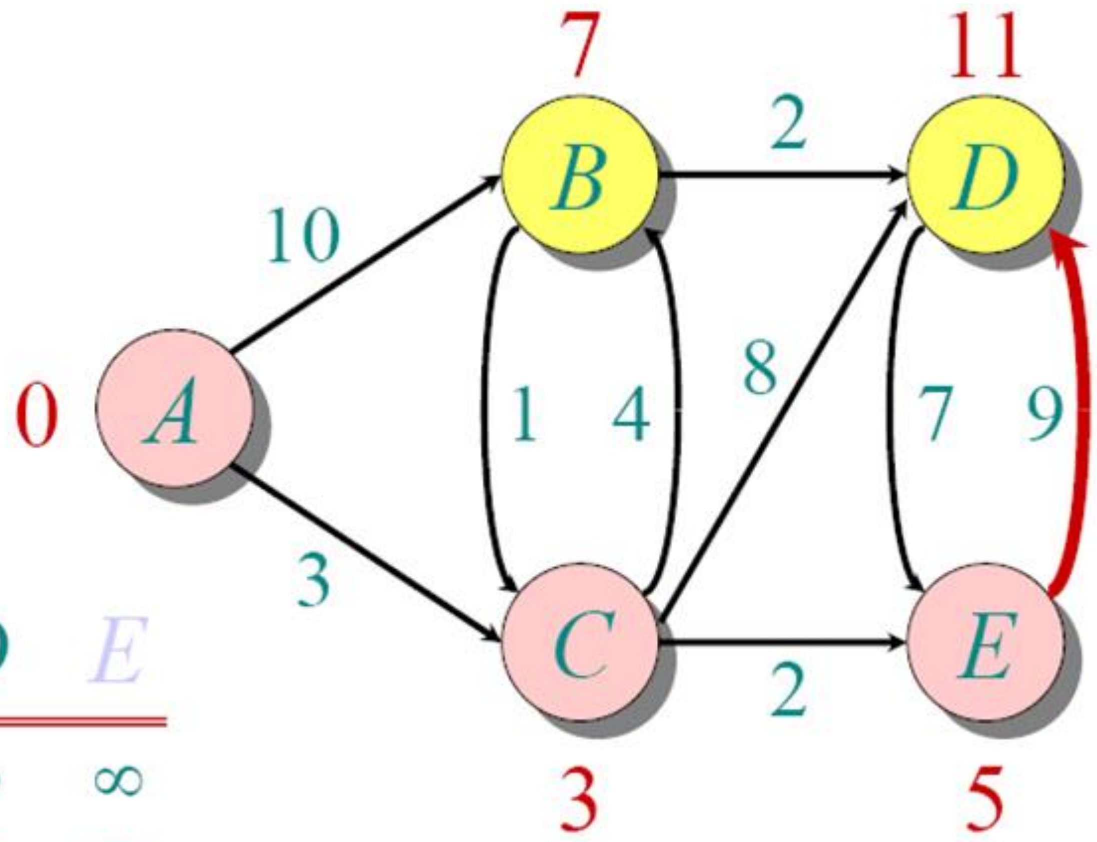
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }

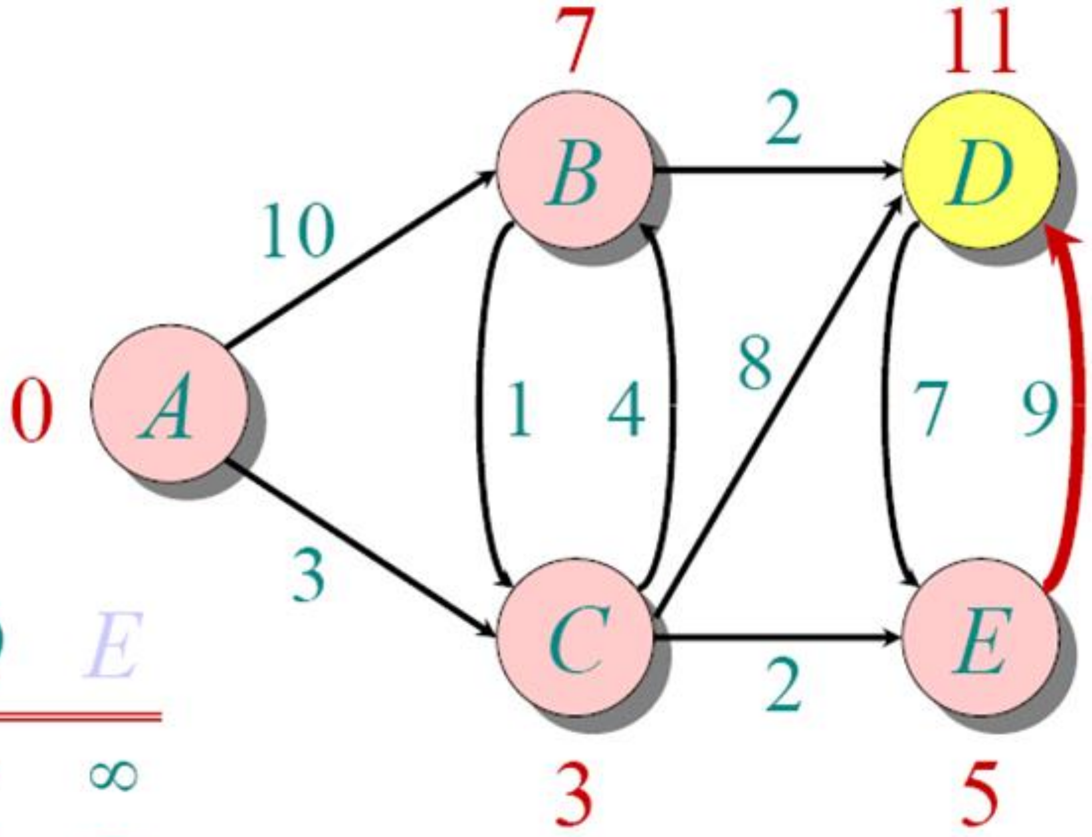
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E }

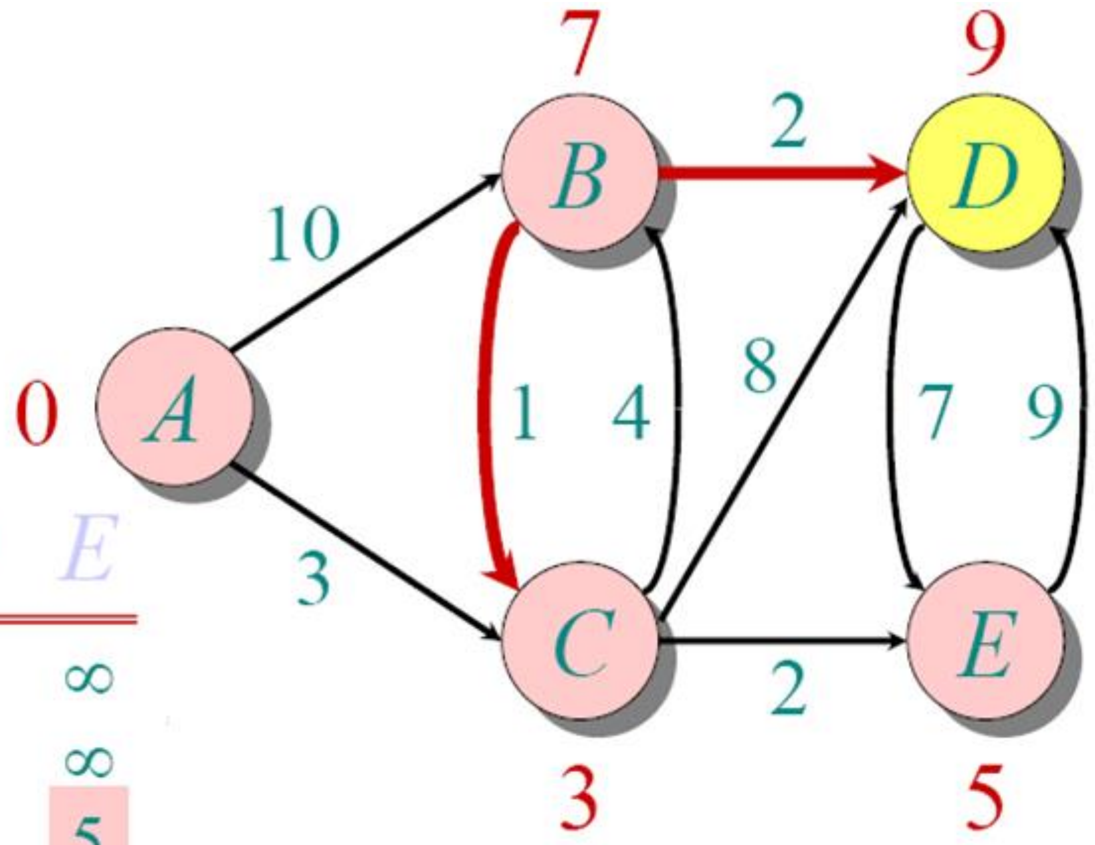
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

Another Example

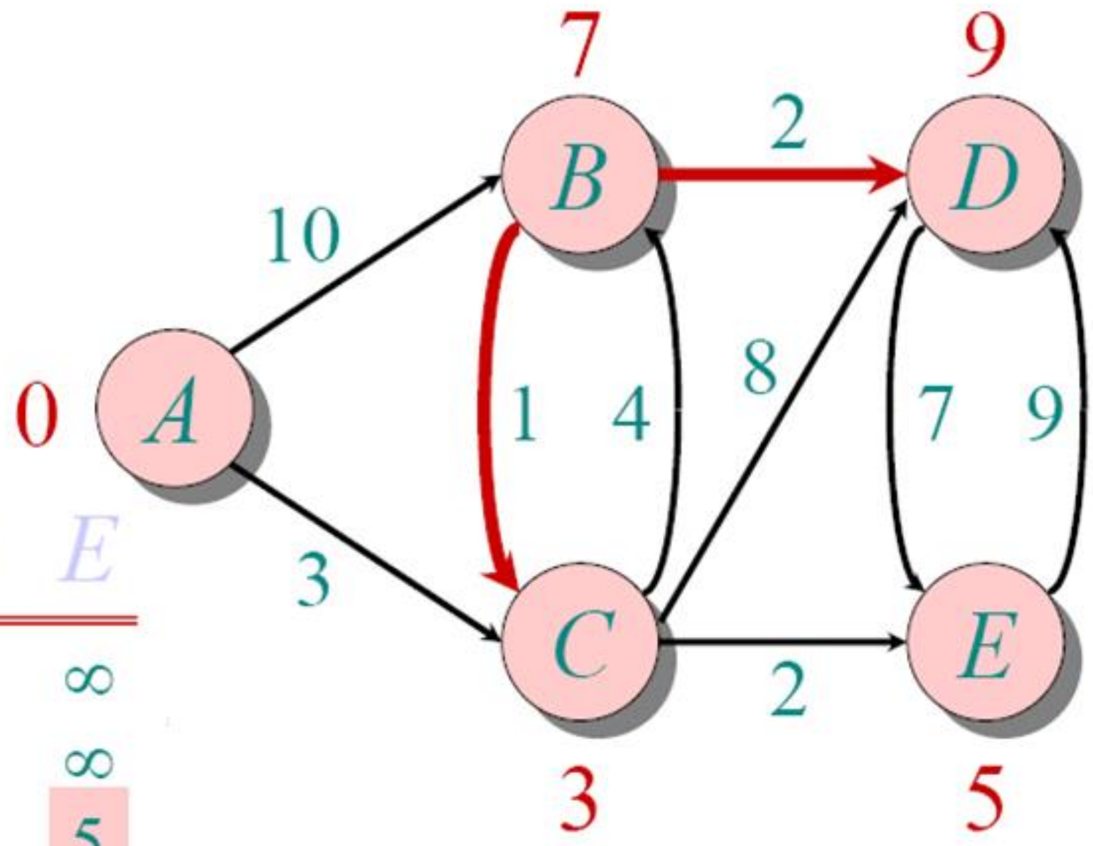


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B }

Another Example

Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	



S: { A, C, E, B, D }

Dijkstra's Pseudo Code

- Graph *G*, weight function *w*, root *s*

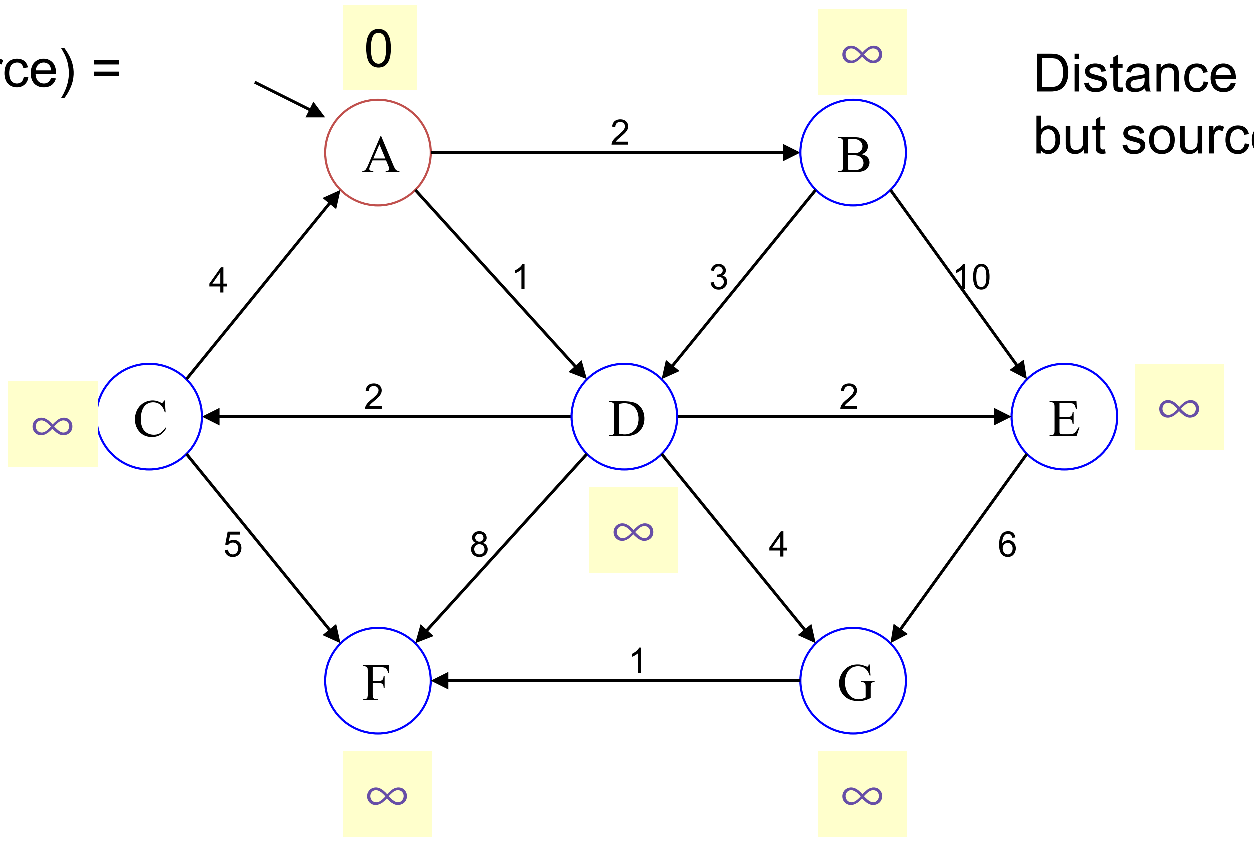
```
DIJKSTRA($G, w, s$)
for each $v \in V$
        do $d[v] \leftarrow \infty$
$d[s] \leftarrow 0$
$S \leftarrow \emptyset$    $\triangleright$ Set of discovered nodes
$Q \leftarrow V$
while $Q \neq \emptyset$
        do $u \leftarrow$ EXTRACT-MIN($Q$)
            $S \leftarrow S \cup \{u\}$
            for each $v \in Adj[u]$
                do if $d[v] > d[u] + w(u, v)$
                    then $d[v] \leftarrow d[u] + w(u, v)$
```

relaxing edges

Example: Initialization

Distance(source) = 0

Distance (all vertices but source) = ∞

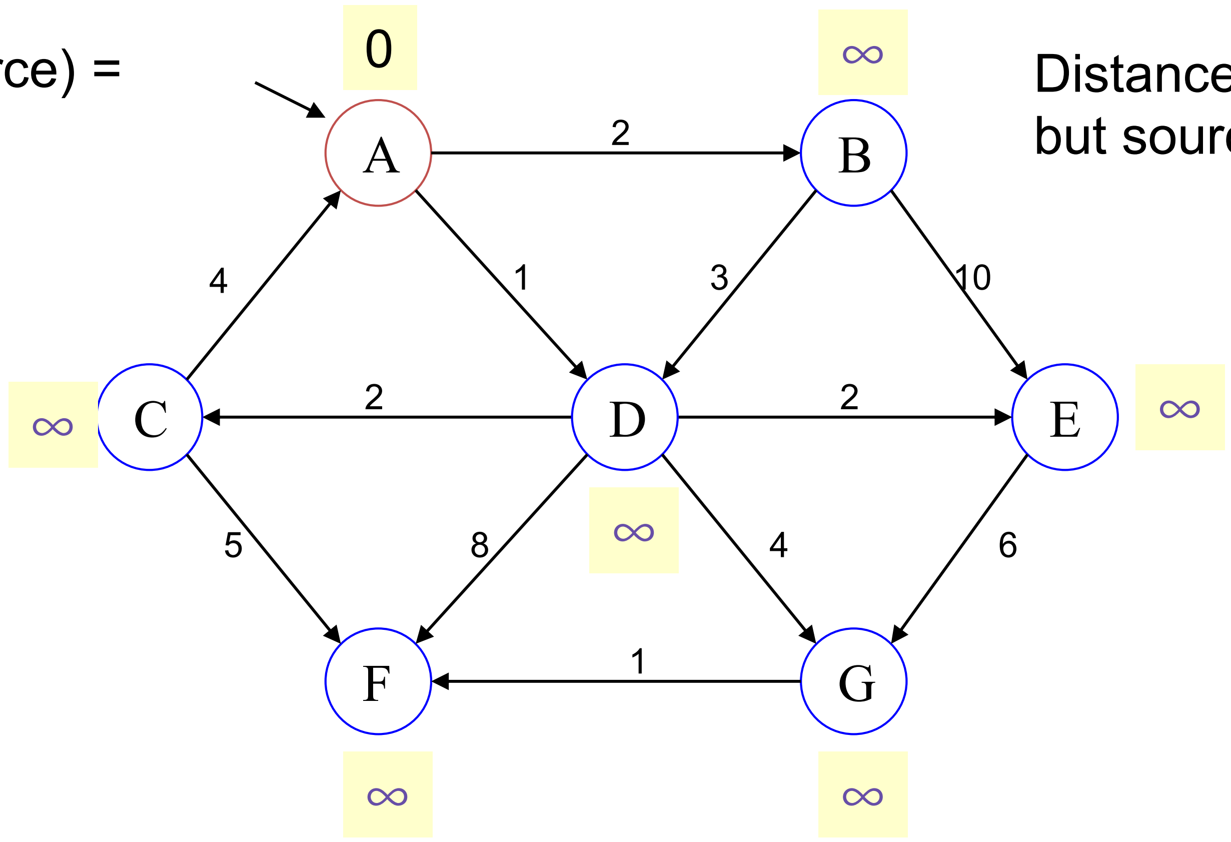


Pick vertex in List with minimum distance.

Example: Initialization

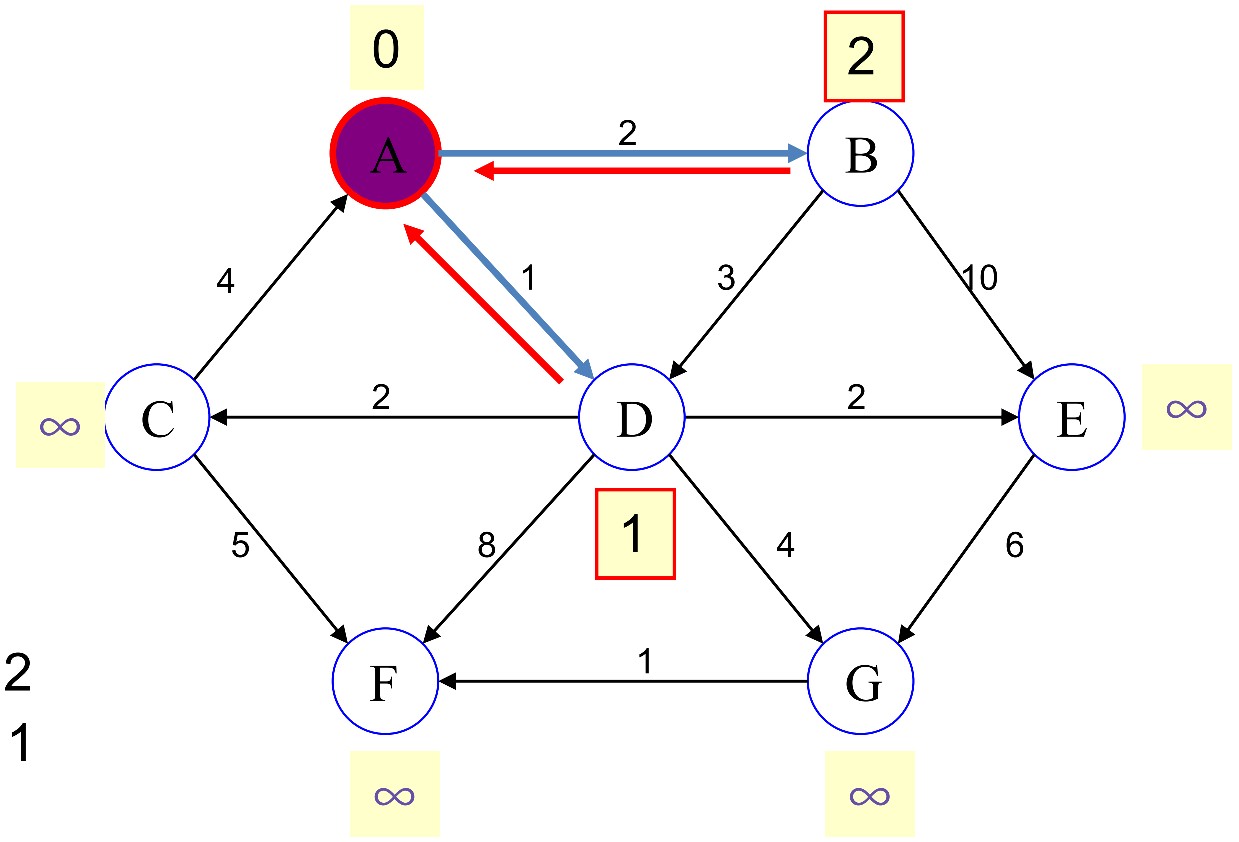
Distance(source) = 0

Distance (all vertices but source) = ∞



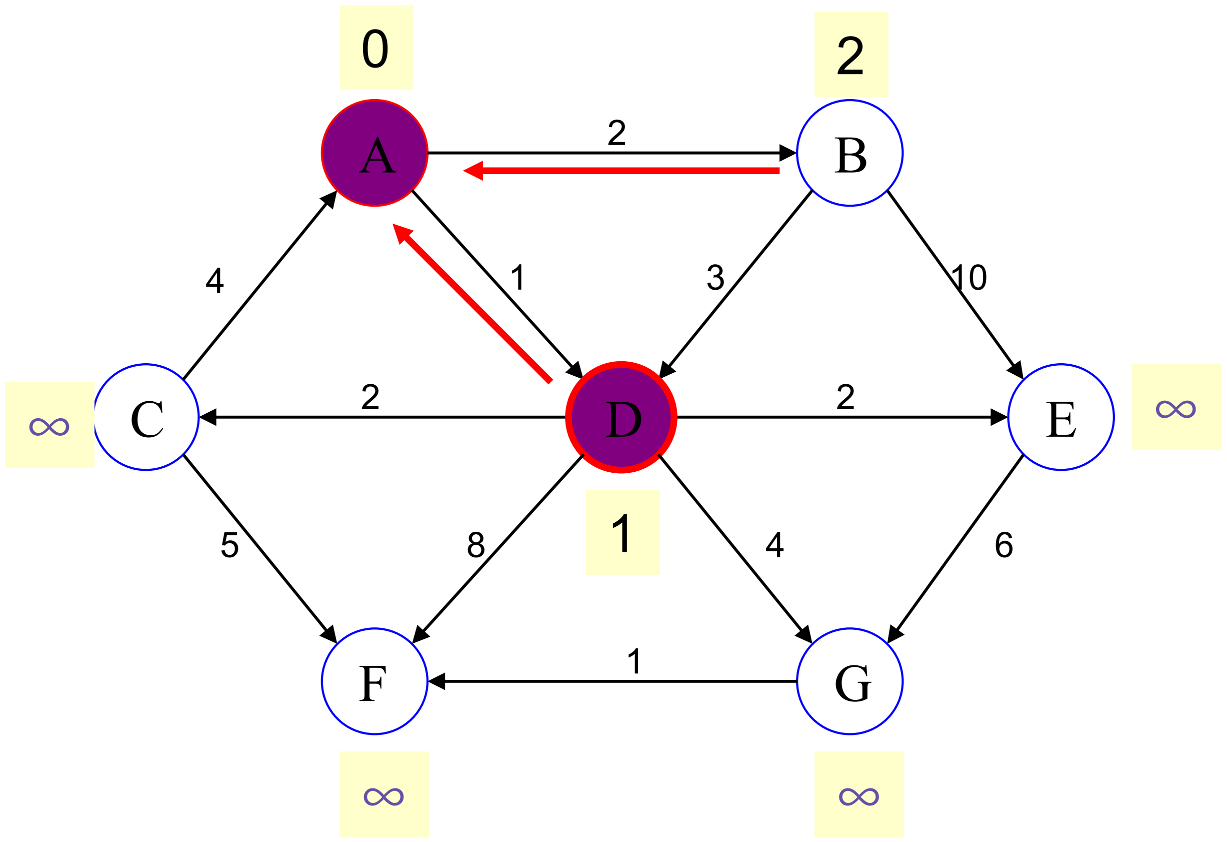
Pick vertex in List with minimum distance.

Example: Update neighbors' distance



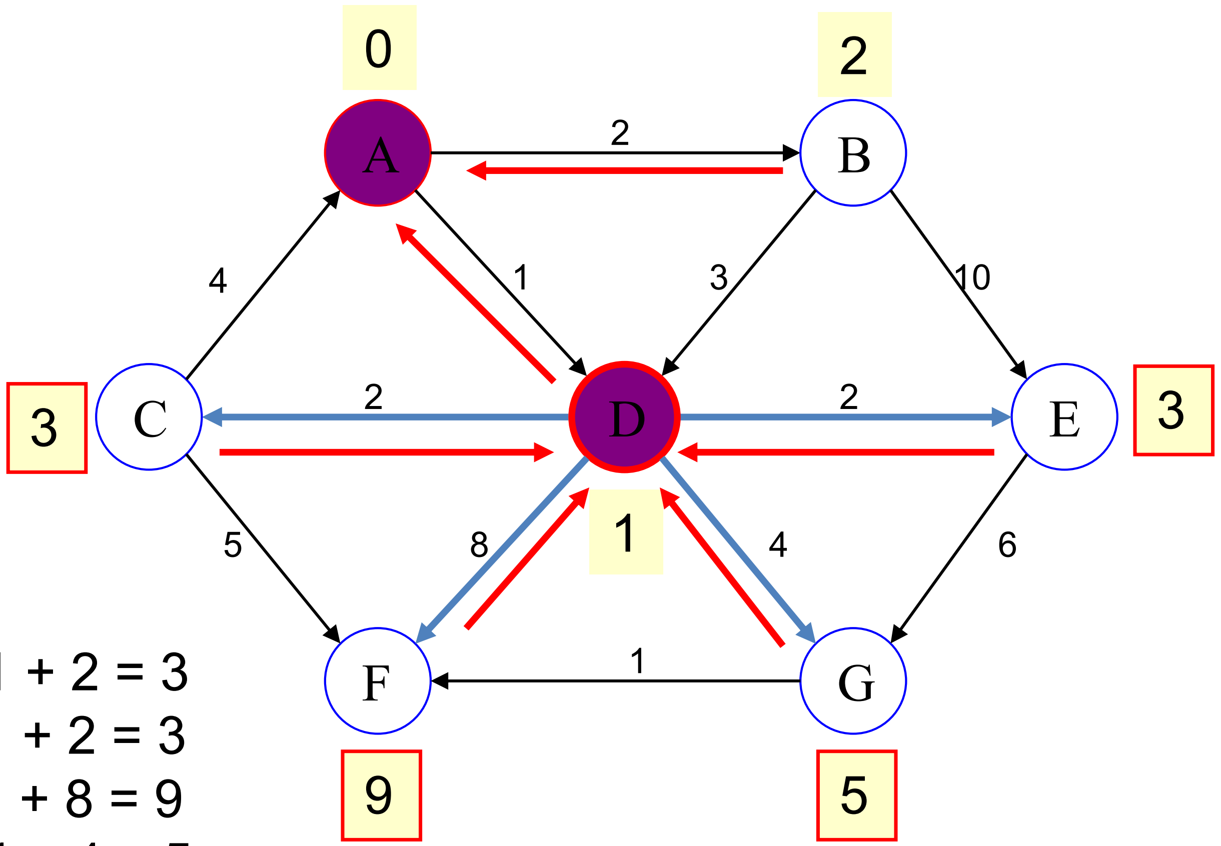
Distance(B) = 2
Distance(D) = 1

Example: Remove vertex with minimum distance



Pick vertex in List with minimum distance, i.e., D

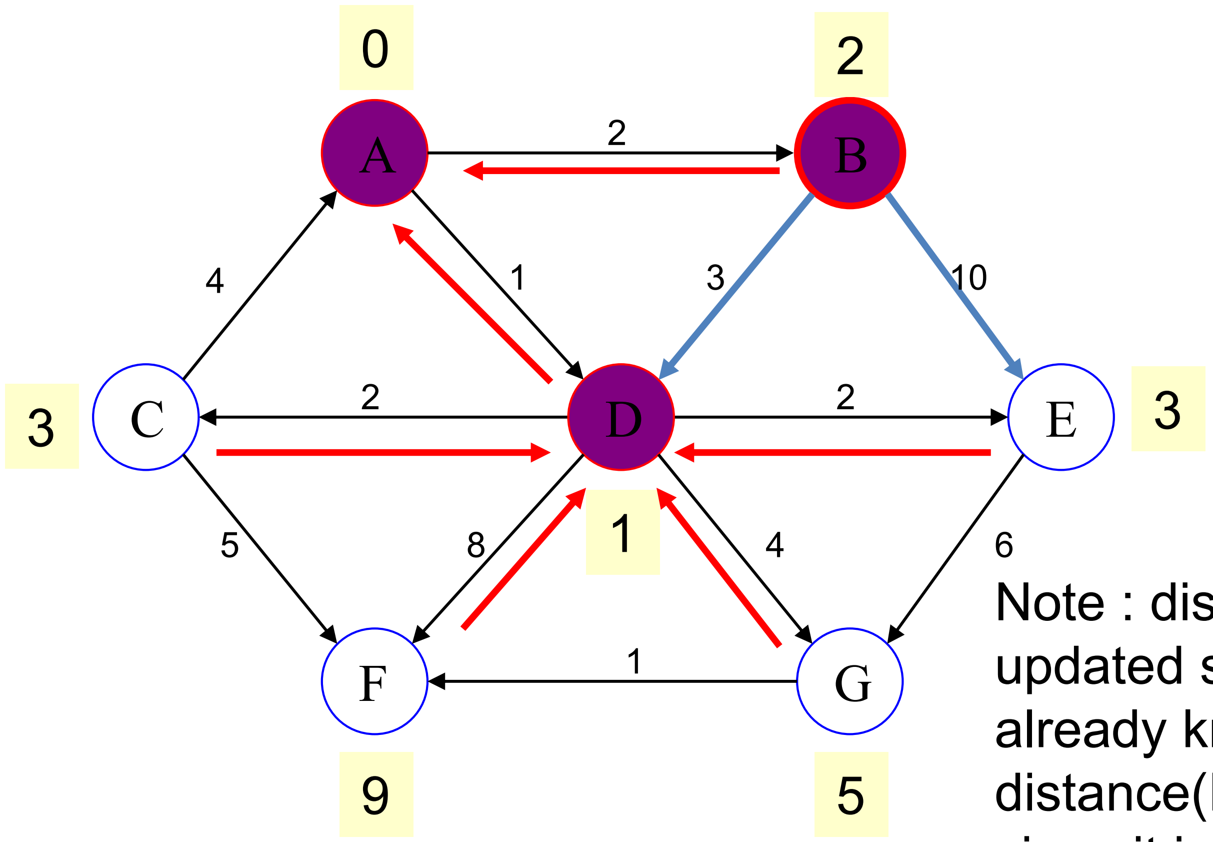
Example: Update neighbors



Distance(C) = 1 + 2 = 3
Distance(E) = 1 + 2 = 3
Distance(F) = 1 + 8 = 9
Distance(G) = 1 + 4 = 5

Example: Continued...

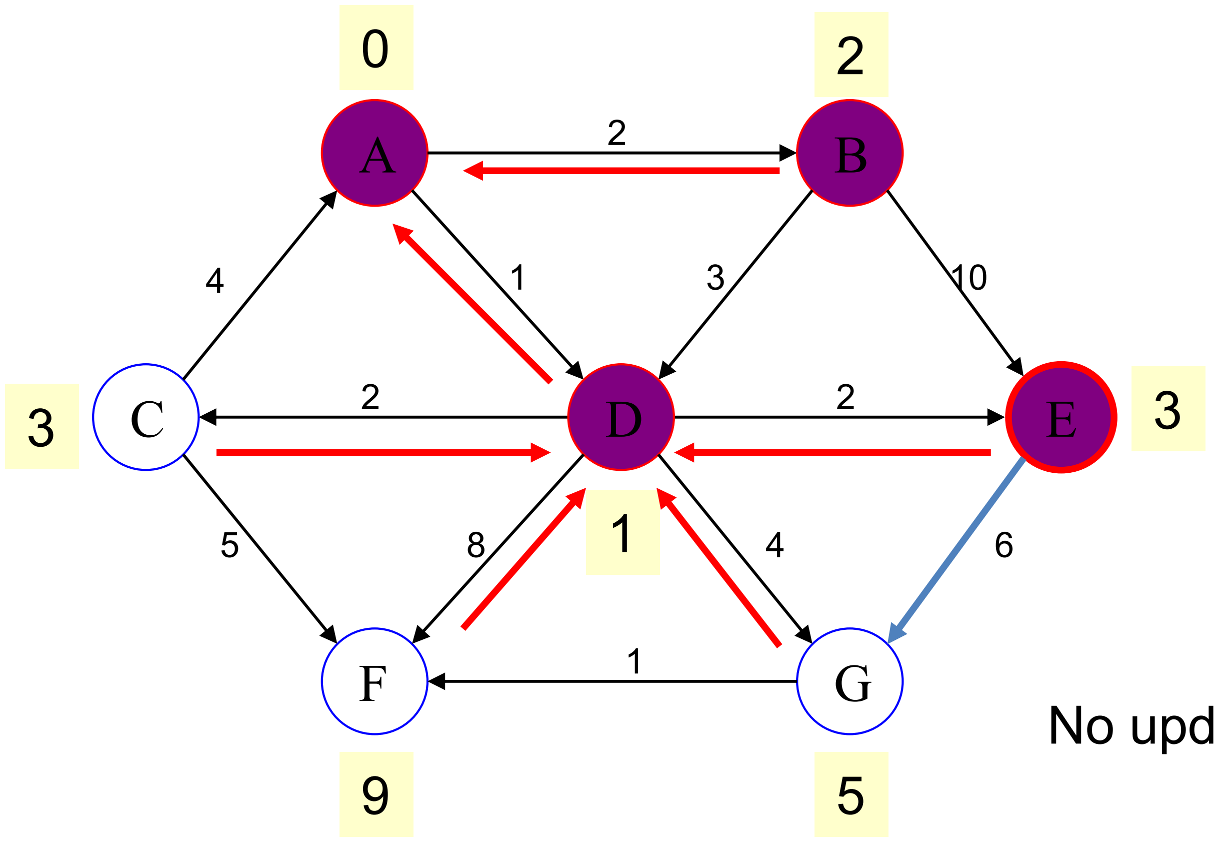
Pick vertex in List with minimum distance (B) and update neighbors



Note : distance(D) not updated since D is already known and distance(E) not updated since it is larger than previously computed

Example: Continued...

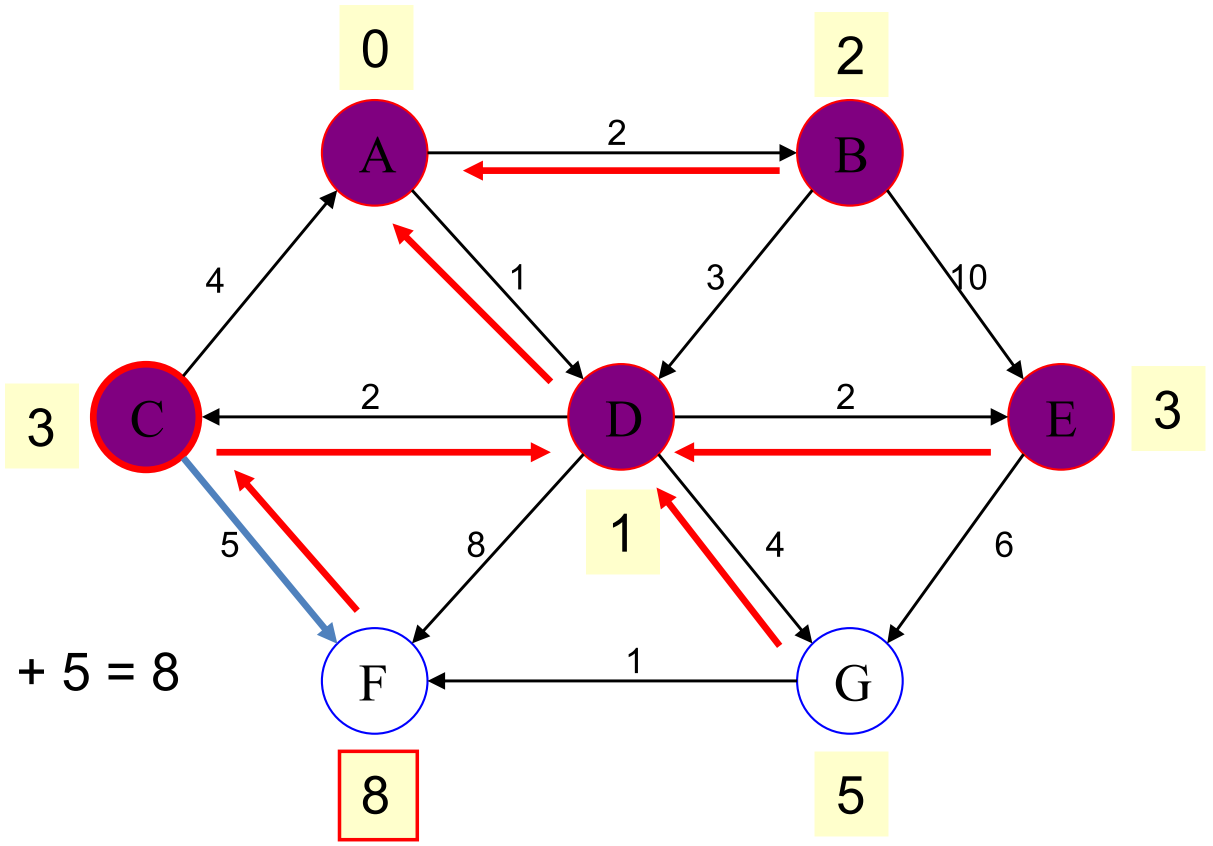
Pick vertex List with minimum distance (E) and update neighbors



No updating

Example: Continued...

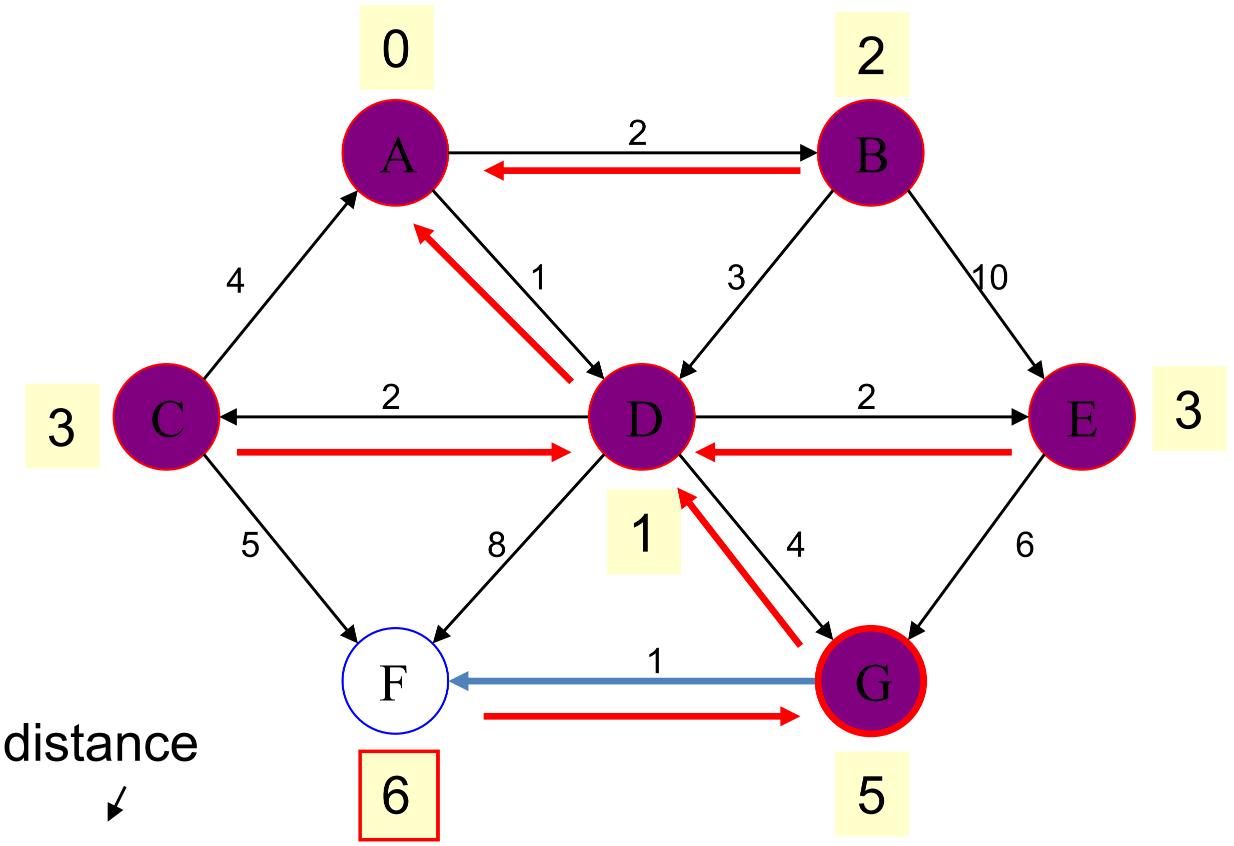
Pick vertex List with minimum distance (C) and update neighbors



Distance(F) = 3 + 5 = 8

Example: Continued...

Pick vertex List with minimum distance (G) and update neighbors

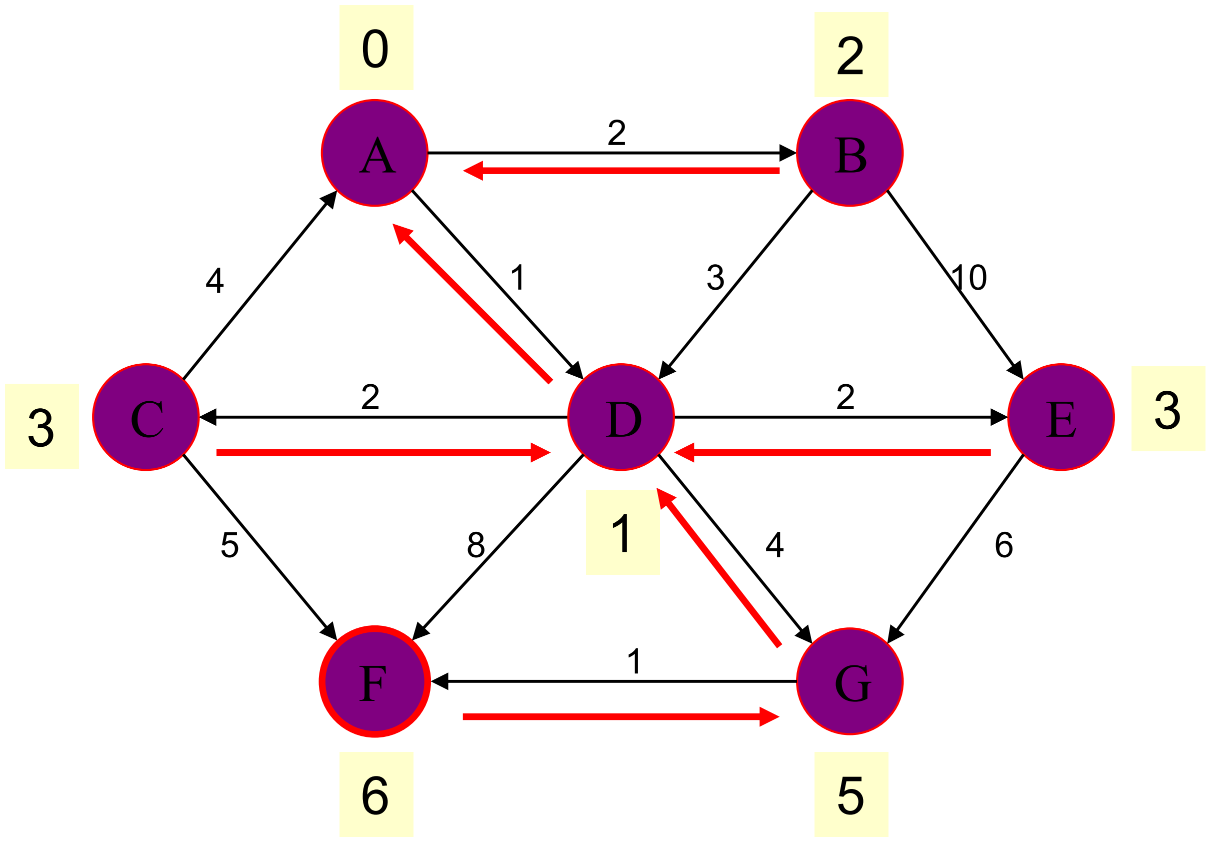


Previous distance

$\swarrow$

Distance(F) = min (8, 5+1) = 6

Example (end)



Pick vertex not in S with lowest cost (F) and update neighbors

Time Complexity: Using List

The simplest implementation of the Dijkstra's algorithm stores vertices in an ordinary linked list or array

- Good for dense graphs (many edges)

- |V| vertices and |E| edges
- Initialization O(|V|)
- While loop O(|V|)
  - Find and remove min distance vertices O(|V|)
- Potentially |E| updates
    - Update costs O(1)

Total time $O(|V^2| + |E|) = O(|V^2|)$

Time Complexity: Priority Queue

For sparse graphs, (i.e. graphs with much less than $|V^2|$ edges) Dijkstra's implemented more efficiently by *priority queue*

- Initialization O(|V|) using O(|V|) buildHeap
- While loop O(|V|)
  - Find and remove min distance vertices O(log |V|) using O(log |V|) deleteMin

- Potentially |E| updates
  - Update costs O(log |V|) using decreaseKey

Total time O(|V|log|V| + |E|log|V|) = O(|E|log|V|)
- |V| = O(|E|) assuming a connected graph

Lecture #06

Maps and Hash Tables

What is Maps?

Maps (sometimes called associative arrays) are an abstract data structure (ADT)

- It stores a collection of (key-value) pairs.
- Each key is unique and allows for quick access to values.
- There cannot be duplicate keys.
- The key is used to decide where to store the value in the structure. In other words, the key associated with value can be viewed as the address for that value.
- Maps provide an alternative approach to searching.

Operations on map

- get(k) : return the value associated with key k if exits in the map
- put(k,k): map the key k to the value v.
- remove(k): remove key k and its value from the map
- size()Returns the number of elements present in the map
- isEmpty()method return True if no key-value is present in the map else return false.

Example: suppose key=integer, value=letter

Operation	M={}
put(5,A)	M={(5,A)}
put(7,B)	M={(5,A), (7,B)}
put(2,C)	M={(5,A), (7,B), (2,C)}
put(8,D)	M={(5,A), (7,B), (2,C), (8,D)}
put(2,E)	M={(5,A), (7,B), (2,E), (8,D)}
get(7)	return B
get(4)	return null
get(2)	return E
remove(5)	M={(7,B), (2,E), (8,D)}
size()	return 3
isEmpty()	return false
remove(2)	M={(7,B), (8,D)}
get(2)	return null

Hashing

The purpose of hashing is to achieve search, insert and delete an element in complexity O(1) [in constant time]. In this technique, convert a range of key values into a range of indexes in the hash table by using a hashing function.

Main concepts of hashing

- The hash table
- The hash function

Hash Table

A hash table is a data structure that provides a mapping from keys to values (also called hash values) using a technique called hashing.

- Keys must be unique but the values may be repeated.
- We refer to these values as key-values pairs.
- key-values pairs can be any type as integer, string .., they are need to hashed by using hash function.

Hash function

A hash function (h(k)) is a function that maps a key 'k' to number in a fixed range.

A good hash function satisfies two basic properties:

1) It should be very fast to compute.
2) It should reduce (not prevent) the number of collisions.

Notes: The size of the table N and the hash function are decided by the user

In the following different methods to find a good hash function:

1. **Division Method**

If k is a key and m is the size of the hash table, the hash function h() is calculated as:

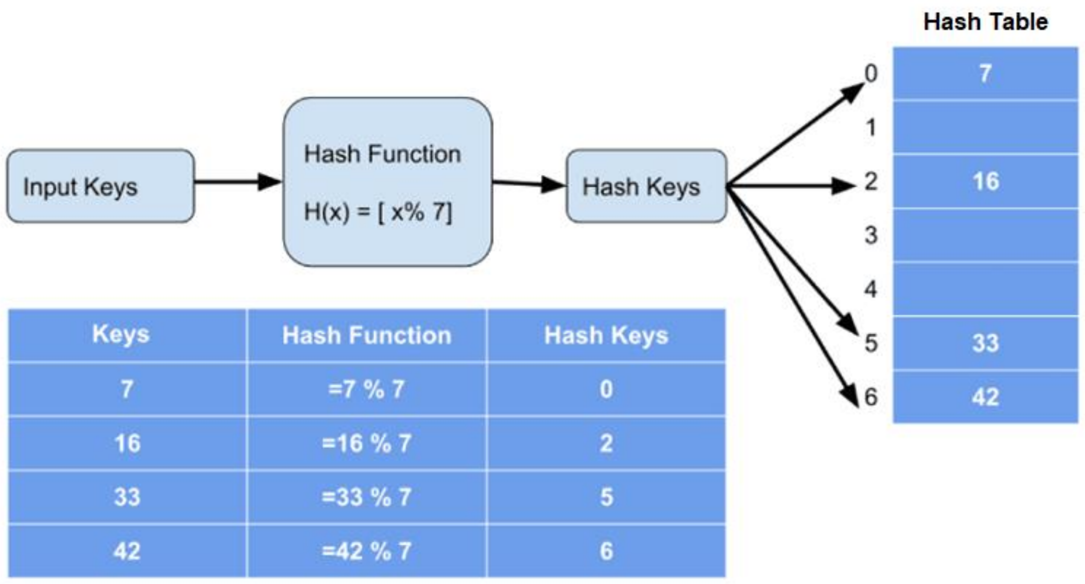
h(k) = k mode m (**best for m is prime**)

2. **Multiplication Method**

h(k) = $\lfloor m(kA \bmod 1) \rfloor$, (**best for m = 2^n**)where

- k is a key and m is the size of the hash table
- $\lfloor\ \rfloor$ gives the floor value
- A is any constant. The value of A lies between 0 and 1

How does Hashing in Data Structure Work?



Keys	Hash Function	Hash Keys
7	=7 % 7	0
16	=16 % 7	2
33	=33 % 7	5
42	=42 % 7	6

Hash table size

- By "size" of the hash table we mean how many slots or buckets it has.
- Choice of hash table size depends in part on choice of hash function, and collision resolution strategy.
- But a good general is:
  [The hash table should be an array with length about 1.3 times the maximum number of keys that will actually be in the table, and Size of hash table array should be a prime number.]
- If you underestimate the number of keys, you may have to create a larger table and rehash the entries when it gets too full; if you overestimate the number of keys, you will be wasting some space

Why prefer table size is prime number?

If size of hash table is prime number will produce the most wide-spread distribution of keys in hash table.

But if it not prime, every key that shares a common factor with the table size will be hashed into a value that is a multiple of this factor.

Why use mod in hash function?

Generally, hash functions calculate an integer value from the key, to ensure this integer value is within the length of the hash table. The result will range somewhere from 0 to the table_size-1.

Hash Collision

Hashing in data structure falls into a collision when the hash function generates the same location (hash value) for two keys. The collision creates a problem because each location in a hash table is supposed to store only one value.

Collision Resolution Techniques

- It is process of finding an alternate location.
- The collision resolution techniques can be named as-

1. Open Hashing (Closed Addressing)
   Separate Chaining)
2. Closed Hashing (Open Addressing)
   A. Linear Probing
   B. Quadratic Probing
   C. Double Hashing

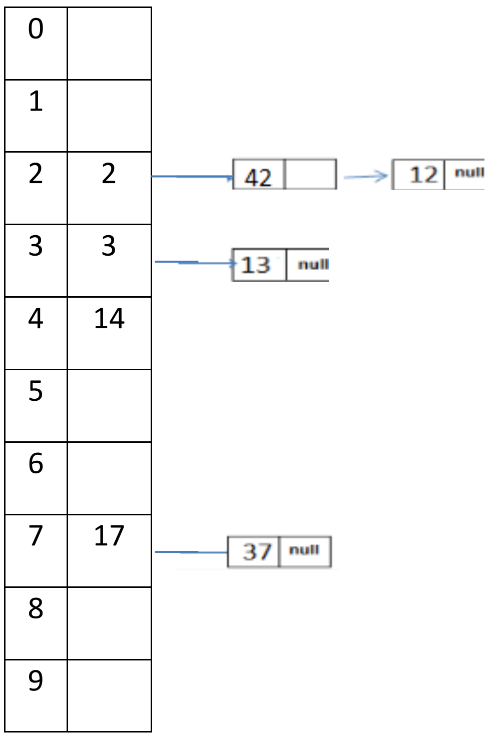
Open Hashing - Separate Chaining

In chaining, if a hash function produces the same location for multiple elements, these elements are stored in the same location by using a linked list.

Example 1: Use division method and opened hashing (closed addressing)(**chaining**) to insert the elements below into a hash table of size **10** .

3, 2, 42, 4, 12, 14,17,13,37

Hash Table



Key(k)	Location(h(key))	probe
3	3 % 10 = 3	1
2	2 % 10 = 2	1
42	42 % 10 = 2	2
12	12 % 10 = 2	3
14	14 % 10 = 4	1
17	17 % 10 = 7	1
13	13 % 10 = 3	2
37	37 % 10 = 7	2

Example 2:

Use opened hashing (**chaining**) to insert the elements below into a hash table of size 9.

7, 42, 25, 70, 14, 38, 8, 21, 34, 11, 48, 26, 93, 125

Open Addressing

Unlike chaining, open addressing doesn't store multiple elements into the same location. Here, each location is either filled with a single key or left **null**.

A. Linear Probing

In this technique search the next empty location in the hash table by looking into the next location until we find an empty location.

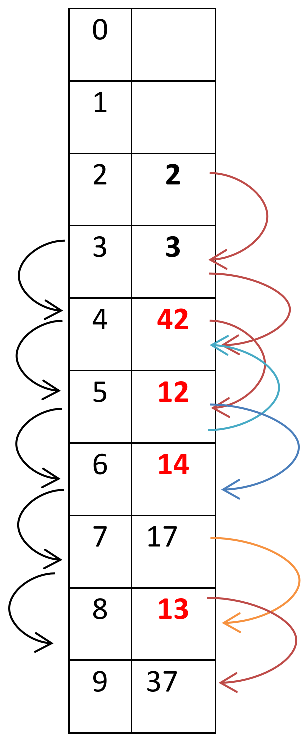
Probe: The list of locations which a method for open addressing produces as alternatives in case of a collision.

[next location = (collision location) % table_size , i=0..table_size-1]

Example 1: Use division method and closed hashing (opened addressing)(**Linear Probing**) to insert the elements below into a hash table of size **10**.

3, 2, 42, 4, 12, 14,17,13,37

Hash Table



Key(k)	Location(h(key))	probe
3	3 % 10 = 3	1
2	2 % 10 = 2	1
42	42 % 10 = 2	3
12	12 % 10 = 2	4
14	14 % 10 = 4	3
17	17 % 10 = 7	1
13	13 % 10 = 3	6
37	37 % 10 = 7	3

Example 2:
Use closed hashing (**Linear Probing**) to insert the elements below into a hash table of size 9.
7, 42, 25, 70, 14, 38, 8, 21, 34, 11, 48, 26, 93, 125

B. Quadratic Probing

It works similar to linear probing but the spacing between the location is increased (greater than one) by using the following relation:

[next location = (collision location+i^2) % table_size , i=0..table_size-1]

Example 1: Use division method and closed hashing (opened addressing)(**Quadratic Probing**) to insert the elements below into a hash table of size **10**.

3, 2, 42, 4, 12, 14,17,13,37

Hash Table

0	**12**
1	
2	**2**
3	**3**
4	**14**
5	
6	**42**
7	17
8	**37**
9	**13**

Key(k)	Location(h(key))	probe
3	3 % 10 = 3	1
2	2 % 10 = 2	1
42	42 % 10 = 2	3
12	12 % 10 = 2	4
14	14 % 10 = 4	1
17	17 % 10 = 7	1
13	13 % 10 = 3	5
37	37 % 10 = 7	2

Example 2:
Use closed hashing (**Linear Probing**) to insert the elements below into a hash table of size 9.
7, 42, 25, 70, 14, 38, 8, 21, 34, 11, 48, 26, 93, 125

Data Structures and Algorithms II
Second Stage

Shatt Al-Arab University
College of Science
Computer Science Department

Lecture #06

Heap Data Structure

Definition, Implementation of MIN and MAX Heap, and Heap Sort

BY

Assit. Lec. Mustafa H. Hashim

Heap Data Structure in Detail (with Java Examples)

1. What is a Heap?

A **Heap** is a specialized tree-based data structure that satisfies the **heap property**. It is commonly implemented as a **binary heap**, which is a complete binary tree where elements follow a specific order.

There are two types of heaps:

1. **Min Heap** – The parent node is always smaller than its children.
2. **Max Heap** – The parent node is always greater than its children.

2. Properties of a Heap

- **Complete Binary Tree:** All levels are filled except possibly the last one.
- Heap Property:
  - Min Heap: **Parent node ≤ Child nodes.**
  - Max Heap: **Parent node ≥ Child nodes.**
- Efficient Operations:
  - Insertion: O(log n)
  - Deletion (Extract Root): O(\log n)
  - Heapify (Rearranging Heap): O(\log n)
  - Building Heap: O(n)O(n)

3. Representation of Heap in Array

A heap can be represented as an array where:

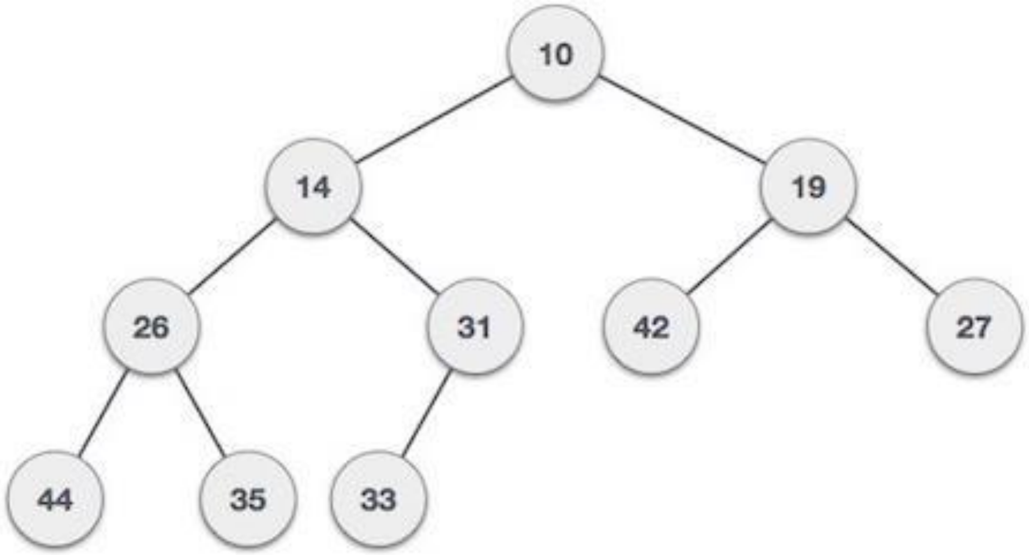
- **For index i:**
  - **Parent:** $(i - 1) / 2$
  - **Left Child:** $2 * i + 1$
  - **Right Child:** $2 * i + 2$

Q: Consider the elements given below.

35, 33, 42, 10, 14, 19, 27, 44, 26, 31

1. Construct a **min-heap** (as a tree) from the above elements. Then show how the heap will be stored in the array.

Tree:

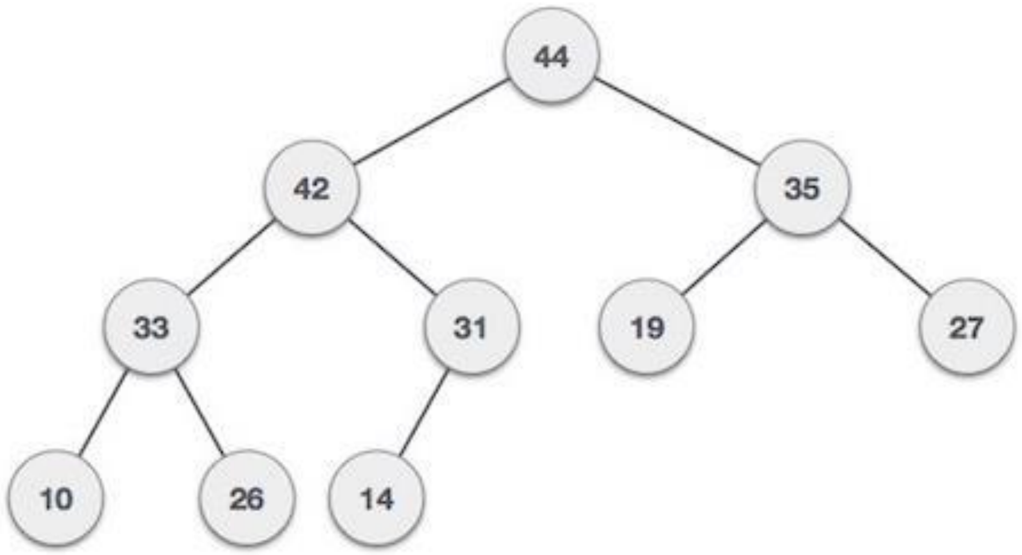


Array:

10	14	19	26	31	42	27	44	35	33

2. Construct a max-heap (as a tree) from the above elements. Then show how the heap will be stored in the array.

Tree



Array:

44	42	35	33	31	19	27	10	26	14

4. Heap Operations

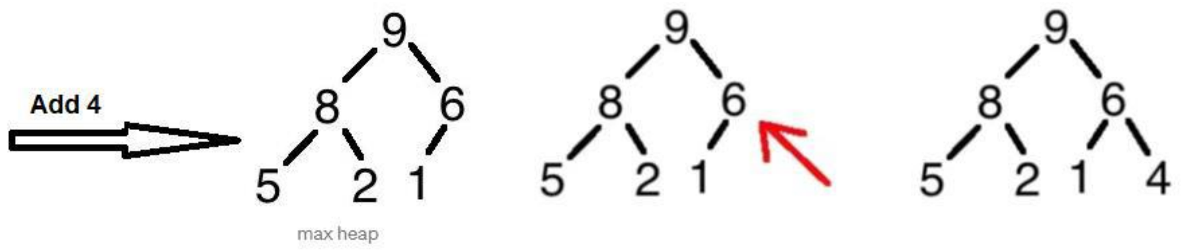
A. Insertion in Heap

1. Insert the element at the last available position.
2. Compare with the parent.
3. Swap if needed (heapify-up or percolate-up).
4. Repeat until the heap property is restored.

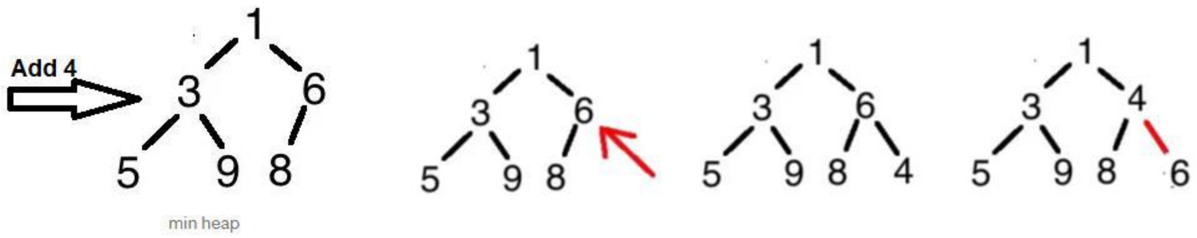
B. Deletion (Extract Min / Max)

1. Remove the root element.
2. Replace it with the last element.
3. Heapify down (compare with children and swap accordingly).
4. Repeat until the heap property is restored.

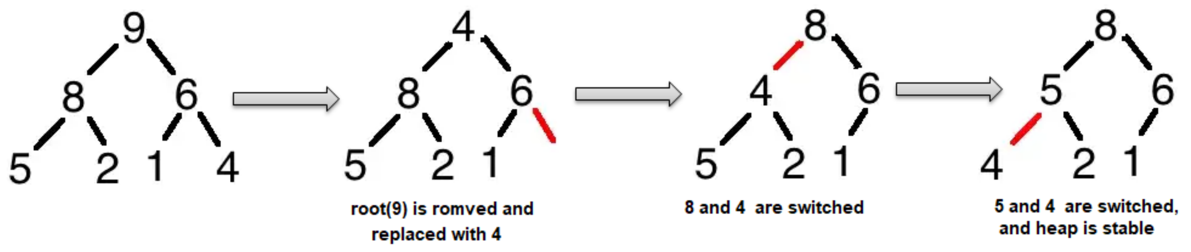
Add and remove elements to heap
 Add 4 to **Max** heap



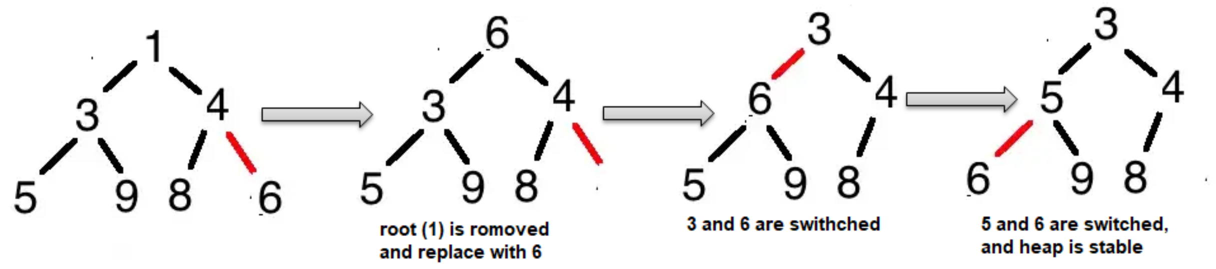
 Add 4 to **Min** heap



 Remove from max heap


root(9) is romved and replaced with 4
8 and 4 are switched
5 and 4 are switched, and heap is stable

 Remove from min heap


root (1) is romoved and replace with 6
3 and 6 are swithched
5 and 6 are switched, and heap is stable

6. Heap Sort Algorithm

Heap Sort is a comparison-based sorting algorithm that uses a heap to sort elements.

Steps for Heap Sort

1. Build a Max Heap from the array.
2. Swap the root element (largest) with the last element.
3. Reduce heap size and apply happify-down.
4. Repeat until the array is sorted.

Applications of Heap

- **Priority Queues**: Used in scheduling algorithms.
- **Graph Algorithms**: Dijkstra's shortest path, Prim's MST.
- **Heap Sort**: Efficient sorting algorithm.
- **Job Scheduling**: Managing task execution based on priority.
- **Memory Management**: Heap memory allocation in programming languages.

7. Time Complexity Analysis

Operation	**Time Complexity**
Insert	O(log n)
Delete (Extract)	O(log n)
Get Min/Max	O(1)
Build Heap	O(n)

PATTERN MATCHING

What is pattern matching?

Pattern matching is the process of checking whether a specific sequence of characters exists among the given data. In the classic pattern-matching problem, we are given a text string of length **n** and a pattern string of length **m** (where m ≤ n), and must determine whether the pattern is a substring of the text. If so, we may want to find the lowest index within the text at which the pattern begins, or perhaps all indices at which the pattern begins. In the following three pattern-matching algorithms:

- brute force
- The Boyer-Moore Algorithm
- The Knuth-Morris-Pratt Algorithm

BRUTE FORCE

A brute force algorithm will try all possible solutions to the problem, only stopping when it finds one that is the actual solution. This can be time-consuming but it is effective in finding short-term solutions to difficult problems.

An implementation of the Boyer-Moore algorithm:

```
/** Returns the lowest index at which substring pattern begins in text (or else −1).*/
public static int findBrute(char[ ] text, char[ ] pattern) {
  int n = text.length;
  int m = pattern.length;
  for (int i=0; i <= n − m; i++) {                // try every starting index within text
    int k = 0;                                    // k is index into pattern
    while (k < m && text[i+k] == pattern[k])      // kth character of pattern matches
      k++;
    if (k == m)                                   // if we reach the end of the pattern,
      return i;                                   // substring text[i..i+m-1] is a match
  }
  return −1;                                      // search failed
}
```

Performance

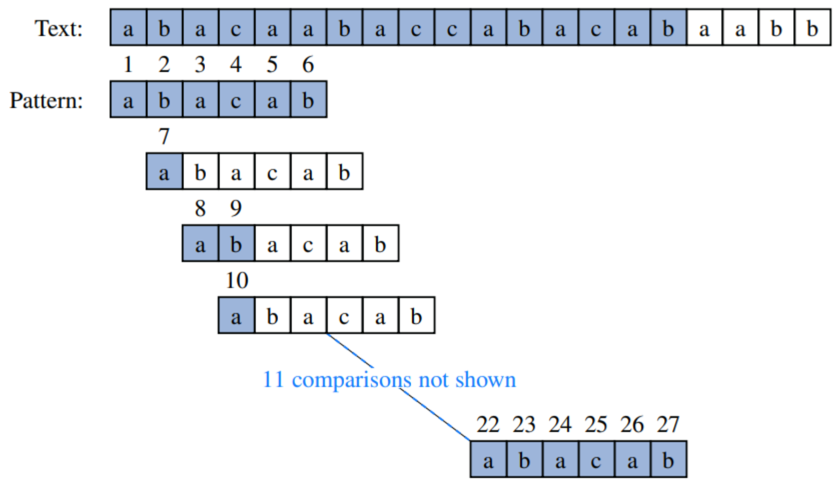
The complexity of the brute force algorithm will be n*m, where "n" is the length of the text string and "m" is the length if the pattern at the worst case.

Example : *Suppose we are given the text string*

text = "abacaabaccabacabaabb"

and the pattern string

pattern = "abacab"



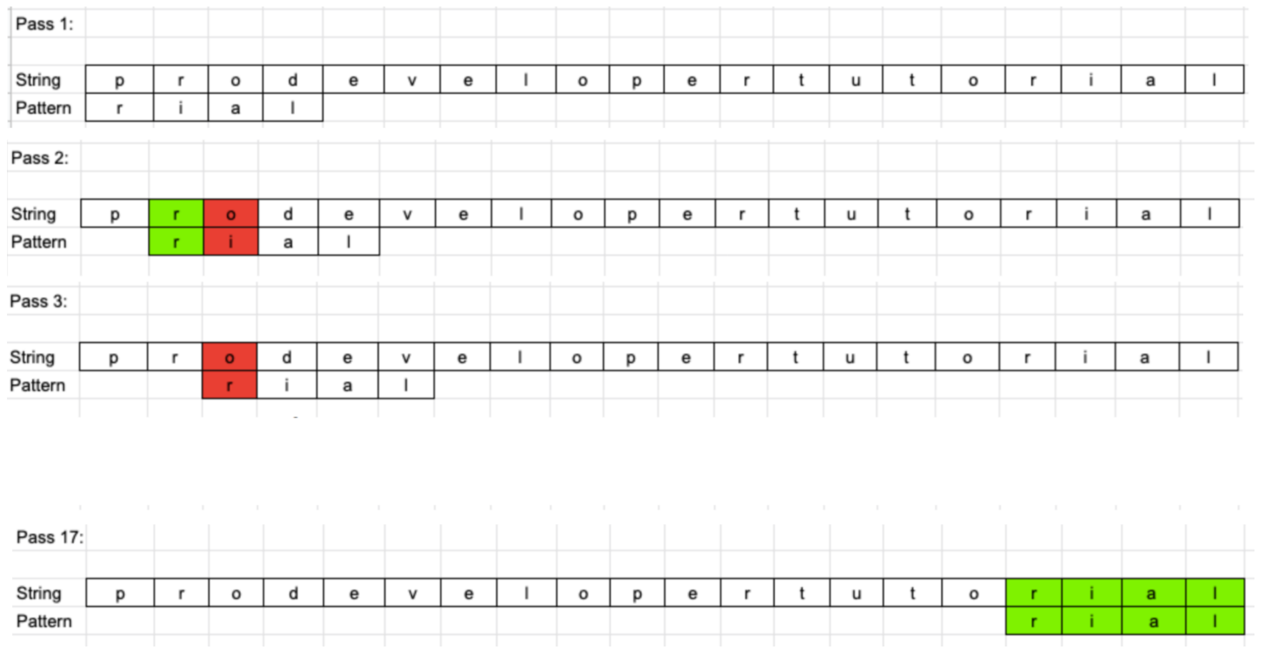
The algorithm performs 27 character comparisons, indicated above with numerical labels.

Example : *Suppose we are given the text string*

text = "prodevelopertutorial"

and the pattern string

pattern = "rial"



Pass 1:

String	p	r	o	d	e	v	e	l	o	p	e	r	t	u	t	o	r	i	a	l
Pattern	r	i	a	l																

Pass 2:

String	p	🟩 r	🟥 o	d	e	v	e	l	o	p	e	r	t	u	t	o	r	i	a	l
Pattern		🟩 r	🟥 i	a	l															

Pass 3:

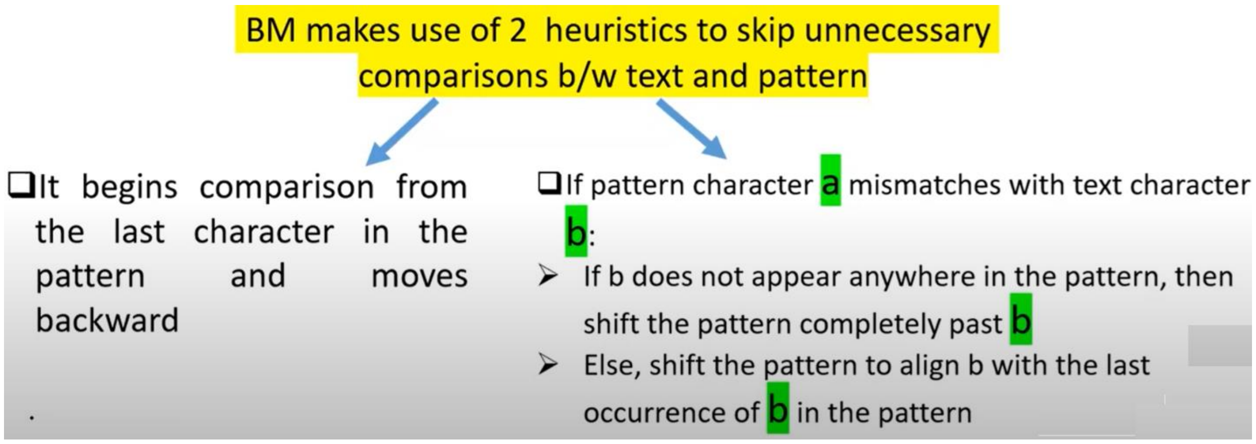
String	p	r	🟥 o	d	e	v	e	l	o	p	e	r	t	u	t	o	r	i	a	l
Pattern			🟥 r	i	a	l														

Pass 17:

String	p	r	o	d	e	v	e	l	o	p	e	r	t	u	t	o	🟩 r	🟩 i	🟩 a	🟩 l
Pattern																	🟩 r	🟩 i	🟩 a	🟩 l

The Boyer-Moore Algorithm

The main idea of the Boyer-Moore algorithm is to improve the running time of the brute-force algorithm by adding two potentially time-saving heuristics. Roughly stated, these heuristics are as follows:

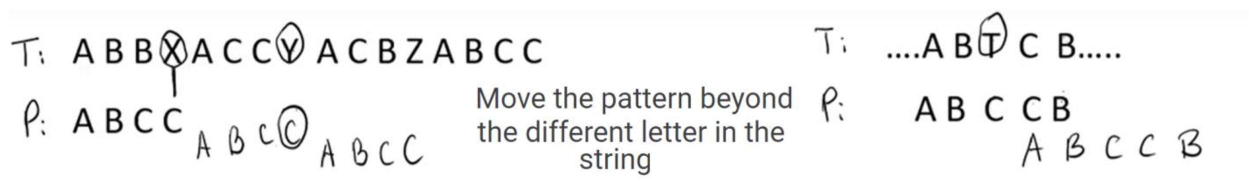


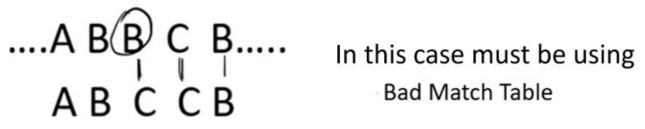
BM makes use of 2 heuristics to skip unnecessary comparisons b/w text and pattern

- ❑It begins comparison from the last character in the pattern and moves backward
- ❑If pattern character a mismatches with text character b:
  - ➢ If b does not appear anywhere in the pattern, then shift the pattern completely past b
  - ➢ Else, shift the pattern to align b with the last occurrence of b in the pattern

An implementation of the Boyer-Moore algorithm:

```
/** Returns the lowest index at which substring pattern begins in text (or else −1).*/
public static int findBoyerMoore(char[ ] text, char[ ] pattern) {
  int n = text.length;
  int m = pattern.length;
  if (m == 0) return 0;                                // trivial search for empty string
  Map<Character,Integer> last = new HashMap<>();       // the 'last' map
  for (int i=0; i < n; i++)
    last.put(text[i], −1);                             // set −1 as default for all text characters
  for (int k=0; k < m; k++)
    last.put(pattern[k], k);                           // rightmost occurrence in pattern is last
  // start with the end of the pattern aligned at index m−1 of the text
  int i = m−1;                                         // an index into the text
  int k = m−1;                                         // an index into the pattern
  while (i < n) {
    if (text[i] == pattern[k]) {                       // a matching character
      if (k == 0) return i;                            // entire pattern has been found
      i−−;                                             // otherwise, examine previous
      k−−;                                             // characters of text/pattern
    } else {
      i += m − Math.min(k, 1 + last.get(text[i]));     // case analysis for jump step
      k = m − 1;                                       // restart at end of pattern
    }
  }
  return −1;                                           // pattern was never found
}
```

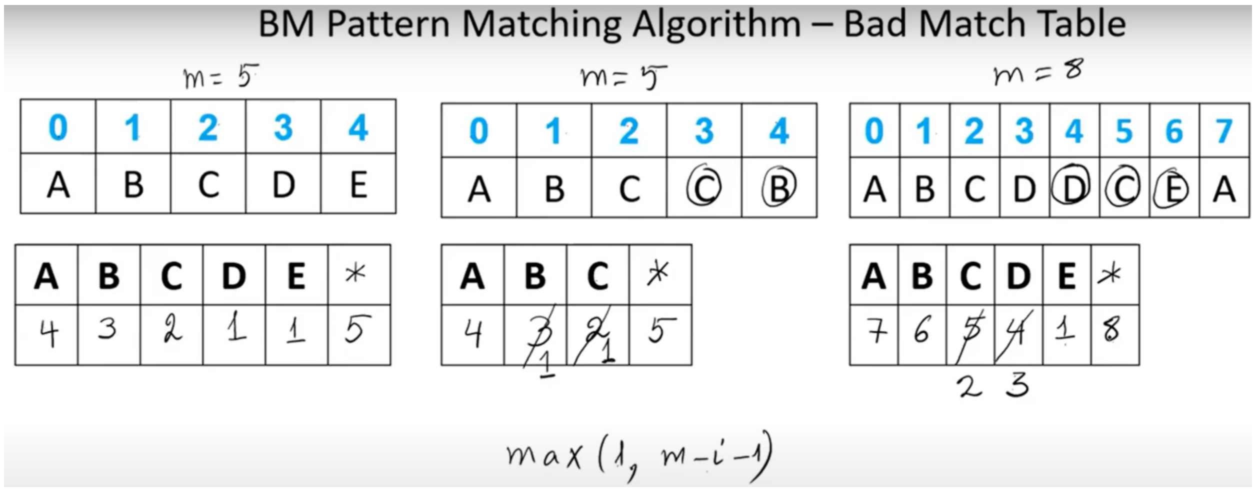

T: A B B X A C C Y A C B Z A B C C
P: A B C C
A B C C
A B C C
Move the pattern beyond the different letter in the string
T: ....A B T C B.....
P: A B C C B
A B C C B



How compute Bad Match Table(BMT)?

- Pattern of size m
- BMT must be does not contain any repetitive characters
- Iterate over the pattern and compute the values of BMT from the :
  Max(1,m-i-1), I is the actual index of the character in the pattern

Examples:



BM Pattern Matching Algorithm – Bad Match Table

$m=5$

0	1	2	3	4
A	B	C	D	E

A	B	C	D	E	*
4	3	2	1	1	5

$m=5$

0	1	2	3	4
A	B	C	C	B

A	B	C	*
4	~~3~~ 1	~~2~~ 1	5

$m=8$

0	1	2	3	4	5	6	7
A	B	C	D	D	C	E	A

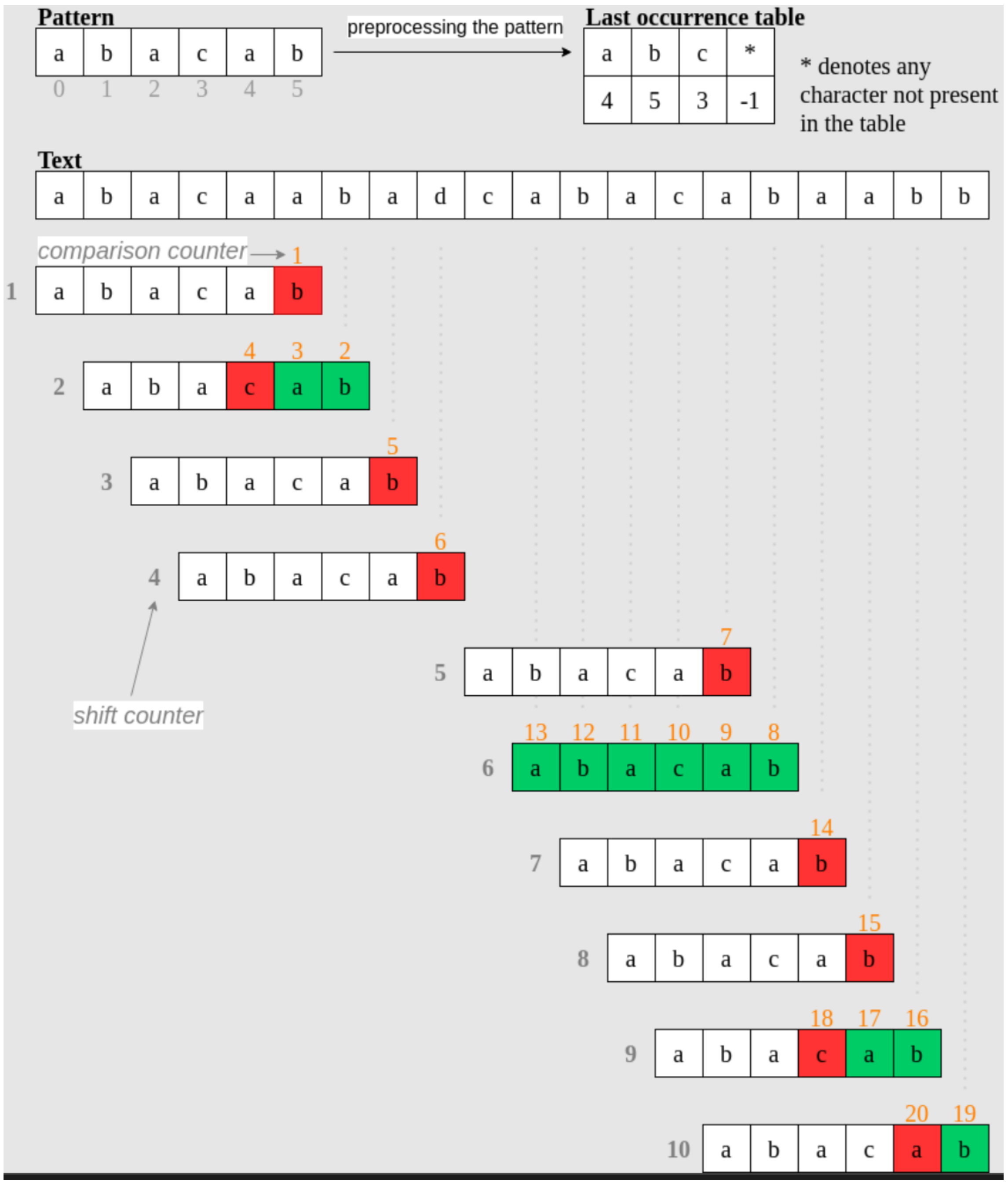
A	B	C	D	E	*
7	6	~~5~~ 2	~~4~~ 3	1	8

$max(1, m-i-1)$

Performance

In the worst-case the performance of the Boyer-Moore algorithm is O(mn), where m is the length of the substring and n is the length of the string.

Example



The algorithm performs 13 character comparisons, which are indicated with numerical labels.

The Knuth-Morris-Pratt Algorithm

KMP algorithm is used to find a **"Pattern"** in a **"Text"**. This algorithm compares character by character from left to right. But whenever a mismatch occurs, it uses a preprocessed table called **"Prefix Table"** to skip characters comparison while matching. Sometimes prefix table is also known as **LPS Table**. Here LPS stands for **"Longest proper Prefix which is also Suffix"**.

Steps for Creating LPS Table (Prefix Table)

- **Step 1 -** Define a one dimensional array with the size equal to the length of the Pattern. (LPS[size])
- **Step 2 -** Define variables **i & j**. Set i = 0, j = 1 and LPS[0] = 0.
- **Step 3 -** Compare the characters at **Pattern[i]** and **Pattern[j].**
- **Step 4 -** If both are matched then set **LPS[j] = i+1** and increment both i & j values by one. Goto to Step 3.
- **Step 5 -** If both are not matched then check the value of variable 'i'. If it is '0' then set **LPS[j] = 0** and increment 'j' value by one, if it is not '0' then set **i = LPS[i-1]**. Goto Step 3.
- **Step 6-** Repeat above steps until all the values of LPS[] are filled.

Example for creating KMP Algorithm's LPS Table (Prefix Table)

Consider the following Pattern

	0	1	2	3	4	5	6
Pattern :	A	B	C	D	A	B	D

Let us define LPS[] table with size 7 which is equal to length of the Pattern

	0	1	2	3	4	5	6
LPS							

Step 1 - Define variables i & j. Set i = 0, j = 1 and LPS[0] = 0.

	0	1	2	3	4	5	6
LPS	0						

i = 0 and j = 1

Step 2 - Campare Pattern[i] with Pattern[j] ===> A with B.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0					

i = 0 and j = 2

Step 3 - Campare Pattern[i] with Pattern[j] ===> A with C.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0				

i = 0 and j = 3

Step 4 - Campare Pattern[i] with Pattern[j] ===> A with D.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0			

i = 0 and j = 4

Step 5 - Campare Pattern[i] with Pattern[j] ===> A with A.
Since both are matching set LPS[j] = i+1 and increment both i & j value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1		

i = 1 and j = 5

Step 6 - Campare Pattern[i] with Pattern[j] ===> B with B.
Since both are matching set LPS[j] = i+1 and increment both i & j value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	

i = 2 and j = 6

Step 7 - Campare Pattern[i] with Pattern[j] ===> C with D.
Since both are not matching and i !=0, we need to set i = LPS[i-1]
===> i = LPS[2-1] = LPS[1] = 0.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	

i = 0 and j = 6

Step 8 - Campare Pattern[i] with Pattern[j] ===> A with D.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

Here LPS[] is filled with all values so we stop the process. The final LPS[] table is as follows...

	0	1	2	3	4	5	6
LPS	**0**	**0**	**0**	**0**	**1**	**2**	**0**

How to use LPS Table?

We use the LPS table to decide how many characters are to be skipped for comparison when a mismatch has occurred.

When a mismatch occurs, check the LPS value of the previous character of the mismatched character in the pattern. If it is '0' then start comparing the first character of the pattern with the next character to the mismatched character in the text. If it is not '0' then start comparing the character which is at an index value equal to the LPS value of the previous character to the mismatched character in pattern with the mismatched character in the Text.

How the KMP Algorithm Works?

Let us see a working example of KMP Algorithm to find a Pattern in a Text...

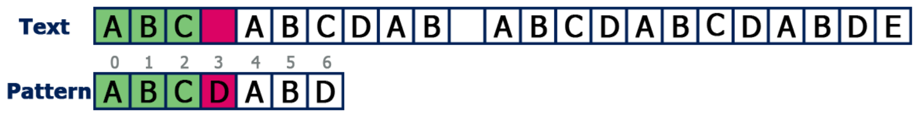
Consider the following Text and Pattern

Text : ABC ABCDAB ABCDABCDABDE
Pattern : ABCDABD

LPS[] table for the above pattern is as follows...

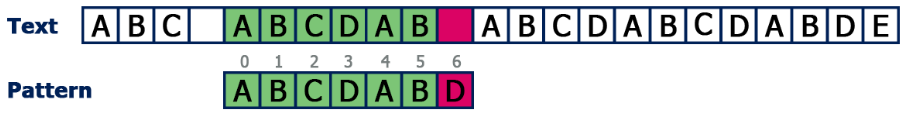
	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

Step 1 - Start comparing first character of Pattern with first character of Text from left to right



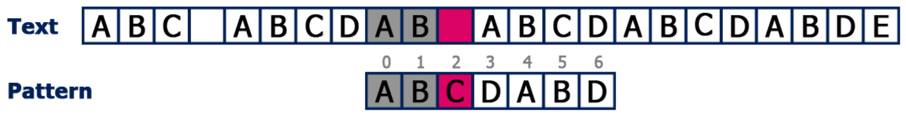
Here mismatch occured at Pattern[3], so we need to consider LPS[2] value. Since LPS[2] value is '0' we must compare first character in Pattern with next character in Text.

Step 2 - Start comparing first character of Pattern with next character of Text.



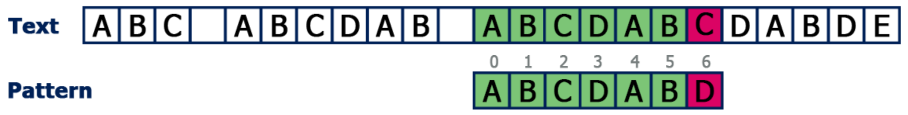
Here mismatch occured at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 3 - Since LPS value is '2' no need to compare Pattern[0] & Pattern[1] values



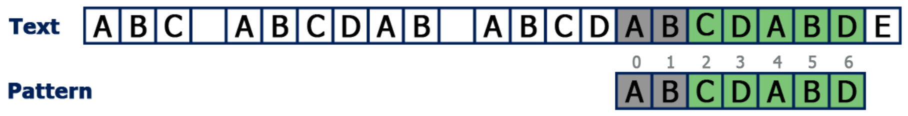
Here mismatch occured at Pattern[2], so we need to consider LPS[1] value. Since LPS[1] value is '0' we must compare first character in Pattern with next character in Text.

Step 4 - Compare Pattern[0] with next character in Text.



Here mismatch occured at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 5 - Compare Pattern[2] with mismatched character in Text.



Here all the characters of Pattern matched with a substring in Text which is starting from index value 15. So we conclude that given Pattern found at index 15 in Text.

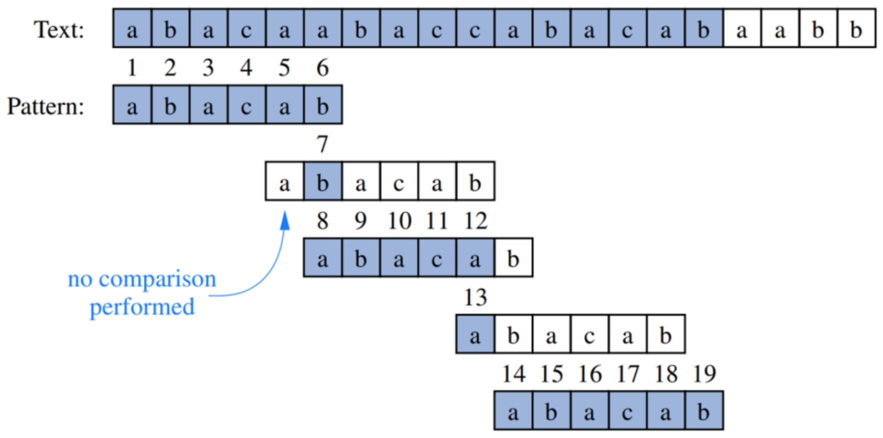
An implementation of the Knuth-Morris-Pratt Algorithm:

```
/** Returns the lowest index at which substring pattern begins in text (or else −1).*/
public static int findKMP(char[ ] text, char[ ] pattern) {
  int n = text.length;
  int m = pattern.length;
  if (m == 0) return 0;                      // trivial search for empty string
  int[ ] fail = computeFailKMP(pattern);     // computed by private utility
  int j = 0;                                 // index into text
  int k = 0;                                 // index into pattern
  while (j < n) {
    if (text[j] == pattern[k]) {             // pattern[0..k] matched thus far
      if (k == m − 1) return j − m + 1;      // match is complete
      j++;                                   // otherwise, try to extend match
      k++;
    } else if (k > 0)
      k = fail[k−1];                         // reuse suffix of P[0..k-1]
    else
      j++;
  }
  return −1;                                 // reached end without match
}
```

Performance

The Knuth-Morris-Pratt algorithm performs pattern matching on a text string of length n and a pattern string of length m in O(n+m) time.

Example1: Given a string Text and pattern as follows:



The primary algorithm performs 19 character comparisons, which are indicated with numerical labels.

Example2: Given a string 'T' and pattern 'P' as follows:

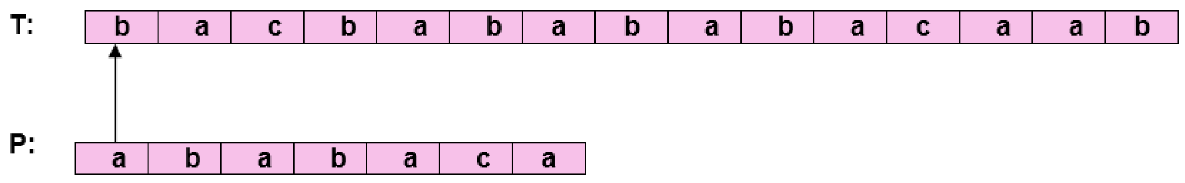
T:	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

P:	a	b	a	b	a	c	a

n = size of T = 15 and m = size of P = 7

Step1: i=1, q=0

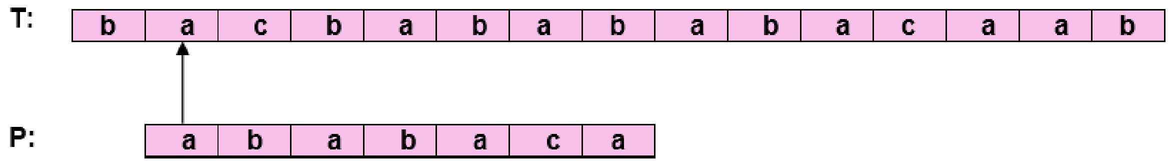
Comparing P [1] with T [1]



P [1] does not match with T [1]. 'p' will be shifted one position to the right.

Step2: i = 2, q = 0

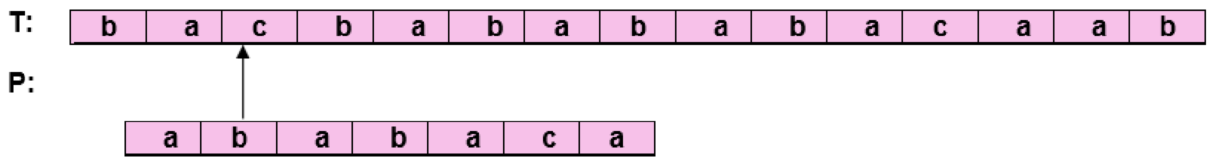
Comparing P [1] with T [2]



P [1] matches T [2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

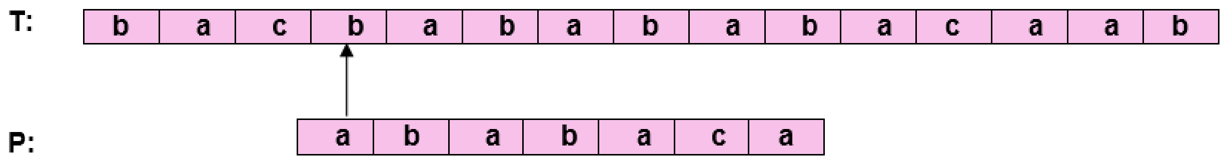
Comparing P [2] with T [3]   P [2] doesn't match with T [3]



Backtracking on p, Comparing P [1] and T [3]

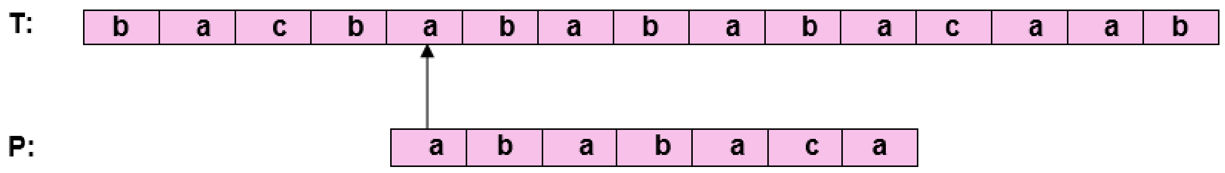
Step4: i = 4, q = 0

Comparing P [1] with T [4]   P [1] doesn't match with T [4]



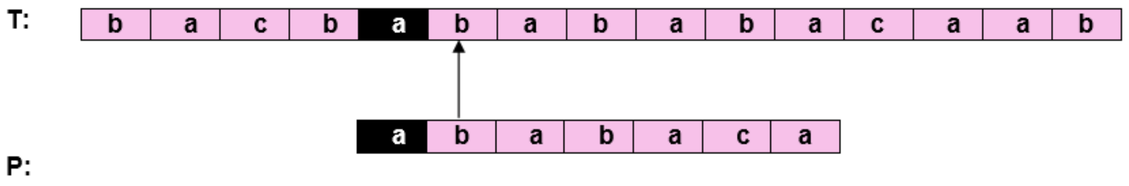
Step5: i = 5, q = 0

Comparing P [1] with T [5]   P [1] match with T [5]



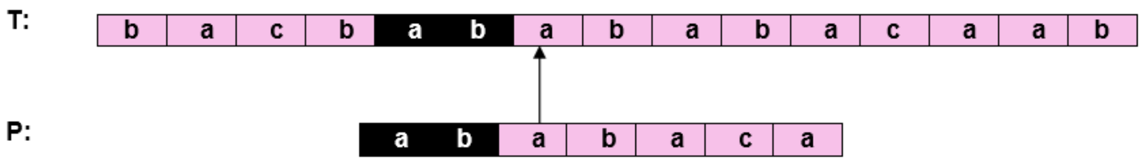
Step6: i = 6, q = 1

Comparing P [2] with T [6]   P [2] matches with T [6]



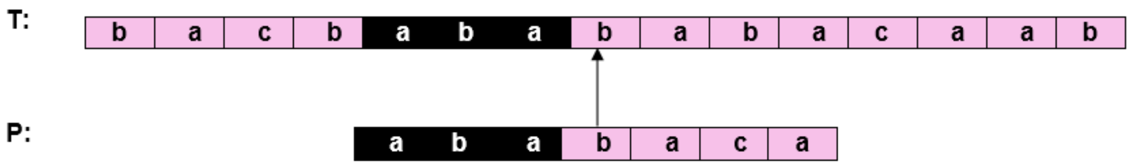
Step7: i = 7, q = 2

Comparing P [3] with T [7]   P [3] matches with T [7]



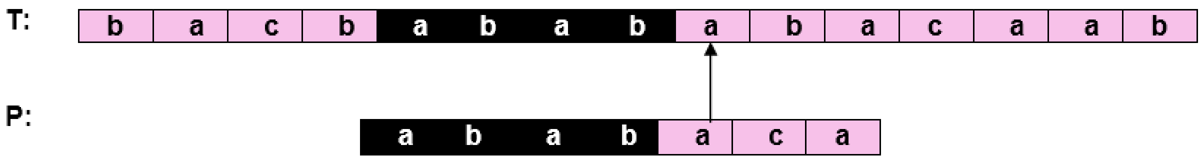
Step8: i = 8, q =3

Comparing P [4] with T [8]   P [4] matches with T [8]



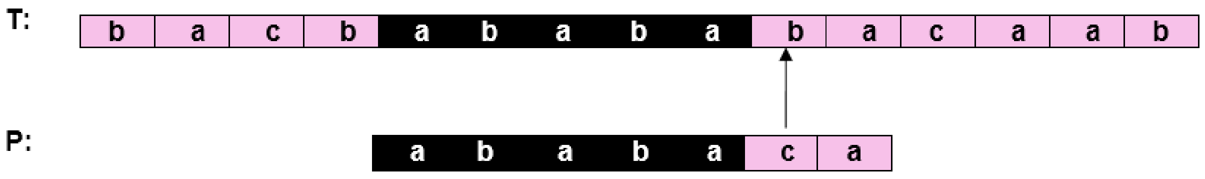
Step9: i = 9, q = 4

Comparing P [5] with T [9]   P [5] matches with T [9]



Step10: i = 10, q = 5

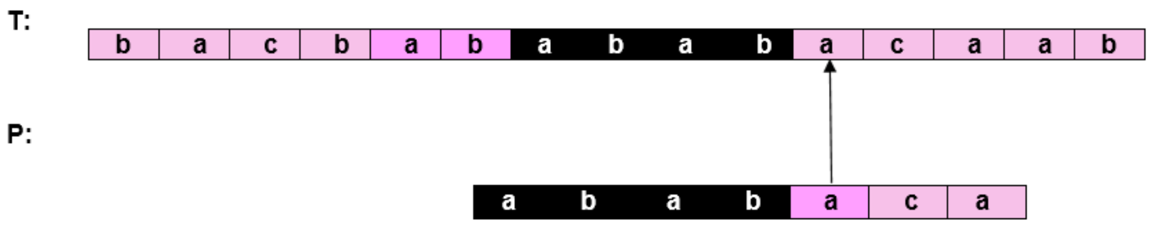
Comparing P [6] with T [10]   P [6] doesn’t match with T [10]



Backtracking on p, Comparing P [4] with T [10] because after mismatch q = π [5] = 3

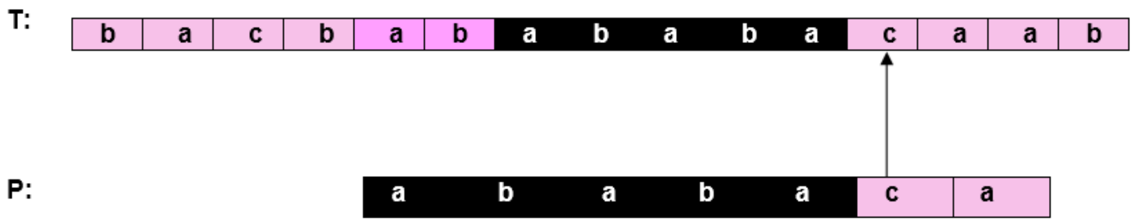
Step11: i = 11, q =4

Comparing P [5] with T [11]   P [5] match with T [11]



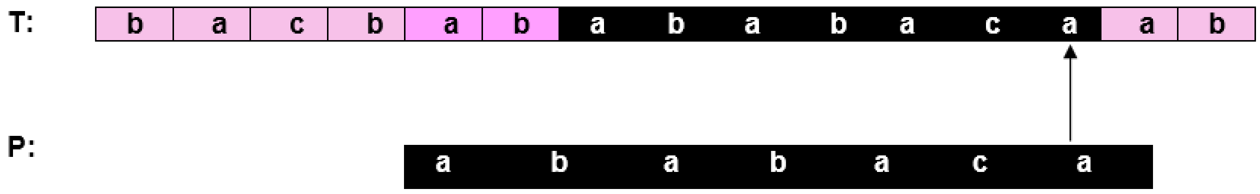
Step12: i = 12, q = 5

Comparing P [6] with T [12]   P [6] matches with T [12]



Step13: i = 3, q = 6

Comparing P [7] with T [13]   P [7] matches with T [13]



Pattern 'P' has been found to complexity occur in a string 'T.' The total number of shifts that took place for the match to be found is i-m = 13 - 7 = 6 shifts.