PG-DAC SEPT-2021

ALGORITHMS & DATA STRUCTURES

SACHIN G. PAWAR

SUNBEAM INSTITUTE OF INFORMATION & TECHNOLOGIES

PUNE & KARAD

Data Structures: Introduction

Name of the Module : Algorithms & Data Structures Using Java.

Prerequisites: Knowledge of programming in C/C++/Java with object oriented concepts.

Weightage : 100 Marks (Theory Exam : 40% + Lab Exam : 40% + Mini Project : 20%).

 # Importance of the Module:

1. CDAC - Syllabus
2. To improve programming skills
3. Campus Placements
4. Applications in Industry work

Data Structures: Introduction

Q. Why there is a need of data structure?

- There is a need of data structure to achieve 3 things in programming:
    1. **efficiency**
    2. **abstraction**
    3. **reusability**

Q. What is a Data Structure?

Data Structure is **a way to store data elements into the memory** (i.e. into the main memory) in **an organized manner** so that operations like **addition, deletion, traversal, searching, sorting** etc... can be performed on it efficiently.

Data Structures: Introduction

Two types of **Data Structures** are there:

1. Linear / Basic data structures : data elements gets stored / arranged into the memory in a **linear manner** (e.g. sequentially) and hence can be accessed linearly / sequentially.

- **Array**
- **Structure & Union**
- **Class**
- **Linked List**
- **Stack**
- **Queue**

2. Non-Linear / Advanced data structures : data elements gets stored / arranged into the memory in a **non-linear manner** (e.g. hierarchical manner) and hence can be accessed non-linearly.

- **Tree (Hierarchical manner)**
- **Graph**
- **Hash Table(Associative manner)**
- **Binary Heap**

Data Structures: Introduction

+ Array: It is a **basic/linear data structure** which is **a collection/list of logically related similar type of data elements** gets stored/arranged into the memory at **contiguos locations**.

+ Structure: It is a **basic/linear data structure** which is **a collection/list of logically related similar and disimmilar type of data elements** gets stored/arranged into the memory **collectively i.e. as a single entity/record.**

sizeof of the structure = sum of size of all its members.

+ Union: Union is same like structure, except, memory allocation i.e. size of union is the size of max size member defined in it and that memory gets shared among all its members for effective memory utilization (can be used in a special case only).

Data Structures: Introduction

Q. What is a Program?

- A Program is **a finite set of instructions written in any programming language** (either in a high level programming language like C, C++, Java, Python or in a low level programming language like assembly, machine etc...) given to the machine to do specific task.

Q. What is an Algorithm?

- An algorithm is **a finite set of instructions written in any human understandable language (like english)**, if followed, acomplishesh a given task.

- **Pseudocode :** It is a **special form of an algorithm**, which is a finite set of instructions written in any human understandable language (like english) **with some programming constraints**, if followed, acomplishesh a given task.

- An algorithm is a template whereas a program is an implementation of an algorithm.

Data Structures: Introduction

 # Algorithm : to do sum of all array elements

Step-1: initially take value of sum is 0.

Step-2: traverse an array sequentially from first element till last element and add each array element into the sum.

Step-3: return final sum.

 # Pseudocode : to do sum of all array elements

```
Algorithm ArraySum(A, n){//whereas A is an array of size n
    sum=0;//initially sum is 0
    for( index = 1 ; index <= size ; index++ ) {
    sum += A[ index ];//add each array element into the sum
    }
    return sum;
}
```

Data Structures: Introduction

- There are two types of Algorithms OR there are two approaches to write an algorithm:

1. iterative (non-recursive) approach :

```
Algorithm ArraySum( A, n){//whereas A is an array of size n
    sum = 0;
    for( index = 1 ; index <= n ; index++ ){
          sum += A[ index ];
    }
       return sum;
}
```

```
for( exp1 ; exp2 ; exp3 ){
    statement/s
}
```

exp1 => initialization
exp2 => termination condition
exp3 => modification

Data Structures: Introduction

2. recursive approach:
While writing recursive algo: we need to take care about 3 things
1. initialization: at the time first time calling to recursive function
2. base condition/termination condition : at the begining of recursive function
3. modification: while recursive function call

Example:

```
Algorithm RecArraySum( A, n, index )
{
   if( index == n )//base condition
   return 0;

   return ( A[ index ] + RecArraySum(A, n, index+1) );
}
```

Data Structures: Introduction

Recursion : it is a process in which we can give call to the function within itself.

function for which recursion is used => recursive function

- there are two types of recursive functions:

1. tail recursive function : recursive function in which recursive function call is the last executable statement.

```
void fun( int n )
{
   if( n == 0 )
      return;

   printf(“%4d”, n);
   fun(n--);//rec function call
}
```

2. non-tail recursive function : recursive function in which recursive function call is not the last executable statement

```
void fun( int n )
{
   if( n == 0 )
      return;

   fun(n--);//rec function call
   printf("%4d", n);
}
```

Data Structures: Introduction

- An Algorithm is a solution of a given problem.
- Algorithm = Solution
- One problem may has many solutions.

For example: Problem => **Sorting :** to arrange data elements in a collection/list of elements either in an ascending order or in descending order.

A1 : Selection Sort
A2 : Bubble Sort
A3 : Insertion Sort
A4 : Quick Sort
A5 : Merge Sort
etc...

- When one problem has many solutions/algorithms, in that case we need to select an efficient solution/algo, and to decide efficiency of an algo's we need to do their analysis.

Data Structures: Introduction

- **Analysis of an algorithm** is a work of determining how much **time** i.e. computer time and **space** i.e. computer memory it needs to run to completion.

- There are two **measures** of an **analysis of an algorithms:**

1. Time Complexity of an algorithm is the amount of **time i.e. computer time** it needs to run to completion.

2. Space Complexity of an algorithm is the amount of **space i.e. computer memory** it needs to run to completion.

Data Structures: Introduction

 # Space Complexity of an algorithm is the amount of space i.e. computer memory it needs to run to completion.

Space complexity = code space + data space + stack space (applicable only for recursive algo)
code space = space required for an instructions
data space = space required for **simple variables, constants & instance variables.**
stack space = space required for **function activation records.**

- Space complexity has **two components:**
1. fixed component: data space (space required for simple vars & constants) and code space.
2. variable component : instance characteristics (i.e. space required for instance vars) and stack space (which is applicable only in recursive algorithms).

Data Structures: Introduction

 # Calculation of space complexity of non-recursive algo:

```
Algorithm ArraySum( A, n){//whereas A is an array of size n
   sum = 0;
   for( index = 1 ; index <= n ; index++ ){
      sum += A[ index ];
   }
   return sum;
}
```

Sp = data space + instance charactristics
simple vars => formal param: A & local vars: sum, index
constants : 0 & 1
instance variable = n, input size of an array = **n units**
data space = 3 units (for simple vars => A, sum & index) + 2 units (for constants => 0 & 1)
=> data space = **5 units**
Sp = (n + 5) units.

Data Structures: Introduction

S = C (code space) + Sp
S = C + (n+5)
S >= (n + 5) ... (as C is constant, it can be neglected)
S >= O(n) => O(n)
Space required for an algo = O(n) => whereas n = input size array.

 # Calculation of space complexity of recursive algorithm:

```
Algorithm RecArraySum( A, n, index ){
    if( index == n )//base condition
    return 0;
    return ( A[ index ] + RecArraySum(A, n, index+1) );
}
```

space complexity = code space + data space + stack space (applicable only in recursive algo)
code space = space required for instructions
data space = space required for variables, constants & instance characteristics
stack space = space required for FAR's.

Data Structures: Introduction

- When any function gets called one entry gets created onto the stack for that function call, referred as **function activation record / stack frame,** it contains **formal params, local vars, return addr, old frame pointer etc...**
In our example of recursive algorithm:
3 units (for A, index & n) + 2 units (for constants 0 & 1) = total 5 **units** of memory is required per function call.
- for size of an array = **n,** algo gets called **(n+1) no. of times.**
Hence, total space required = **5 * (n+1)**
S = 5n + 5
S >= 5n.
S >= 5n
S ~= 5n => O(n), wheras n = size of an array

Data Structures: Introduction

 # Time Complexity:
time complexity = compilation time + execution time
Time complexity has two components :
1. fixed component : compilation time
2. variable component : execution time => it depends on instance char of an algorithm.
Example :

```
Algorithm ArraySum( A, n){//whereas A is an array of size n
    sum = 0;
    for( index = 1 ; index <= n ; index++ ){
    sum += A[ index ];
    }

    return sum;
}
```

Data Structures: Introduction

- for size of an array = 5 => instruction/s inside for loop will execute 5 no. of times
- for size of an array = 10 => instruction/s inside for loop will execute 10 no. of times
- for size of an array = 20 => instruction/s inside for loop will execute 20 no. of times
- for size of an array = n => instruction/s inside for loop will execute n no. of times

 # Scenario-1

Machine-1 : Pentium-4 : Algorithm : input size = 10
Machine-2 : Core i5 : Algorithm : input size = 10

 # Scenario-2

Machine-1 : Core i5 : Algorithm : input size = 10 : system fully laoded with other processes
Machine-2 : Core i5 : Algorithm : input size = 10 : system not fully laoded with other processes.

- it is observed that, execution time is not only depends on instance chars, it also depends on some external factors like hardware on which algorithm is running as well as other conditions, and hence it is not a good practice to decide efficiency of an algo i.e. calculation of time complexity on the basis of an execution time and compilation time, and hence to do analysis of an algorithms **asymptotic analysis** is preferred.

Data Structures: Introduction

 # Asymptotic Analysis : It is a **mathematical way** to calculate time complexity and space complexity of an algorithm **without implementing it in any programming language.**

- In this type of analysis, analysis can be done on the basis of **basic operation** in that algorithm.

e.g. in searching & sorting algorithms **comparison** is the basic operation and hence analysis can be done on the basis of no. of comparisons, in addition of matrices algorithm **addition** is the basic operation and hence on the basis of addition operation analysis can be done.

"Best case time complexity": if an algo takes **min** amount of time to run to completion then it is referred as best case time complexity.

"Worst case time complexity": if an algo takes **max** amount of time to to run to completion then it is referred as worst case time complexity.

"Average case time complexity": if an algo takes **neither min nor max** amount of time to run to completion then it is referred as an average case time complexity.

Data Structures: Introduction

 # Asympotic Notations:

1. Big Omega (Ω) : this notation is used to denote **best case time complexity** – also called as **asymptotic lower bound,** running time of an algorithm cannot be less than its asymptotic lower bound.

2. Big Oh (O) : this notation is used to denote **worst case time complexity** - also called as **asymptotic upper bound,** running time of an algorithm cannot be more than its asymptotic upper bound.

3. Big Theta (θ) : this notation is used to denote an **average case time complexity** - also called as **asymptotic tight bound,** running time of an algorithm cannot be less than its asymptotic lower bound and cannot be more than its asymptotic upper bound i.e. it is **tightly bounded.**

Data Structures: Searching Algorithms

1. Linear Search / Sequential Search:

 # Algorithm :

Step-1 : accept key from the user

Step-2 : start traversal of an array and compare value of the key with each array element sequentially from first element either till match is not found or max till last element, if key is matches with any of array element then return true otherwise return false if key do not matches with any of array element.

```
# Pseudocode:
Algorithm LinearSearch(A, size, key){
   for( int index = 1 ; index <= size ; index++ ){
   if( arr[ index ] ==  key )
   return true;
   }
   return false;
}
```

Data Structures: Searching Algorithms

Best Case: If key is found at very first position in only 1 no. of comparison then it is considered as a best case and running time of an algorithm in this case is **O(1)** => and hence time complexity = **Ω(1)**

Worst Case: If either key is found at last position or key does not exists, in this case maximum **n** no. of comparisons takes place, it is considered as a worst case and running time of an algorithm in this case is **O(n)** => and hence time complexity = **O(n)**

Average Case: If key is found at any in between position it is considered as an average case and running time of an algorithm in this case is **O(n/2) => O(n)** => and hence time complexity = **θ(n)**

Data Structures: Searching Algorithms

2. Binary Search/Logarithmic Search:

- This algorithm follows **divide-and-conquer** approach.

- To apply binary search on an array **prerequisite is that array elements must be in a sorted manner.**

Step-1: accept key from the user

Step-2: in first iteration, find/calculate **mid position** by the formula **mid=(left+right)/2**, (by means of finding mid position big size array gets divided logically into two subarrays, left subarray and right subarray. **Left subarray = left to mid-1** & **right subarray = mid+1 to right**).

Step-3 : compare value of key with an element which is at mid position, if key matches in very first iteration in only one comparison then it is considered as a **best case**, if key matches with mid pos element then return true otherwise if key do not matches then we have to go to next iteration, and in next iteration we go to search key either into the left subarray or into the right subarray.

Step-4 : repeat step-2 & step-3 till either key is not found or max till subarray is valid, if subarray is not valid then key is not found in this case return false.

Data Structures: Searching Algorithms

- as in each iteration 1 comparison takes place and search space is getting reduced by half.

n => n/2 => n/4 => n/8 .....

after iteration-1 => n/2 + 1 => T(n) = $(n/2^1) + 1$

after iteration-2 => n/4 + 2 => T(n) = $(n/2^2) + 2$

after iteration-3 => n/8 + 3 => T(n) = $(n/2^3) + 3$

Lets assume, after k iterations => **T(n) = $(n/2^k) + k$ ..... (equation-I)**

let us assume,

=> **$n = 2^k$**

=> log n = log 2^k (by taking log on both sides)

=> log n = k log 2

=> log n = k (as log 2 ~= 1)

=> k = log n

By substituting value of n & k in equation-I, we get

=> T(n) = $(n / 2^k) + k$

=> T(n) = $(2^k / 2^k) + \log n$

=> T(n) = 1 + log n => T(n) = O(1 + log n **) => T(n) = O(log n).**

Data Structures: Searching Algorithms

```
Algorithm BinarySearch(A, n, key)//A is an array of size "n", and key to be search
{
   left = 1;
   right = n;

   while( left <= right )
   {
      //calculate mid position
      mid = (left+right)/2;
      //compare key with an ele which is at mid position
      if( key == A[ mid ] )//if found return true
         return true;

      //if key is less than mid position element
      if( key < A[ mid ] )
      {
         right = mid-1;//search key only in a left subarray
      }
      else//if key is greater than mid position element
      {
         left = mid+1;//search key only in a right subarray
      }
   }//repeat the above steps either key is not found or max any subarray is valid
   return false;
}
```

Data Structures: Searching Algorithms

Best Case: if the key is found in very first iteration at mid position in only 1 no. of comparison / if key is found at root position it is considered as a best case and running time of an algorithm in this case is O(1) = **Ω(1).**

Worst Case: if either key is not found or key is found at leaf position it is considered as a worst case and running time of an algorithm in this case is O(log n) = **O(log n).**

Average Case: if key is found at non-leaf position it is considered as an average case and running time of an algorithm in this case is O(log n) = **θ(log n).**

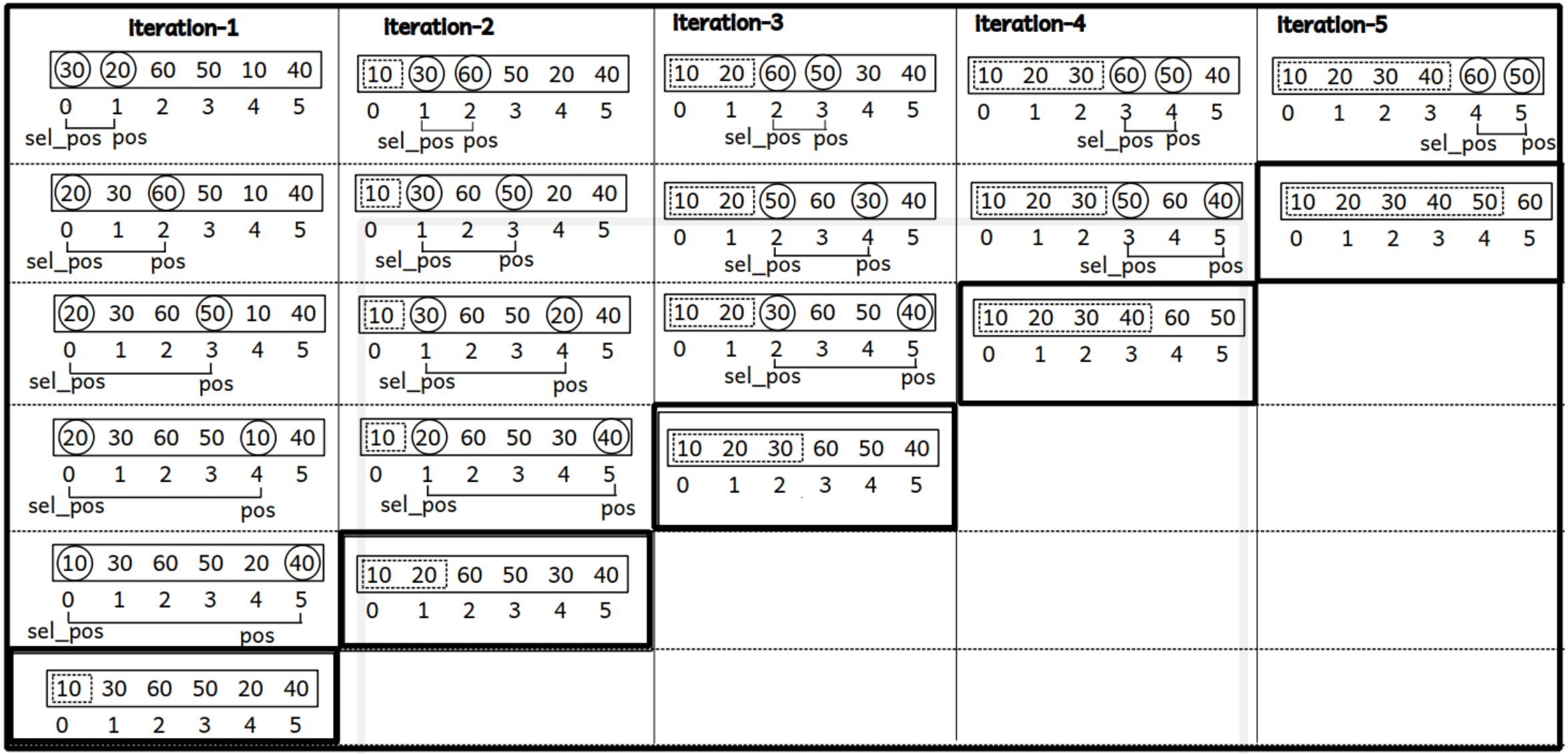
Data Structures: Sorting Algorithms

1. Selection Sort:

- In this algorithm, in first iteration, **first position gets selected** and **element which is at selected position gets compared with all its next position elements**, **if selected position element found greater than any other position element then swapping takes place** and in first iteration **smallest element** gets setteled at first position.

- In the second iteration, **second position gets selected and element which is at selected position gets compared with all its next position elements, if selected position element found greater than any other position element then swapping takes place** and in second iteration **second smallest element** gets setteled at second position, and so on **in maximum (n-1) no. of iterations all array elements gets arranged in a sorted manner.**

Data Structures: Sorting Algorithms



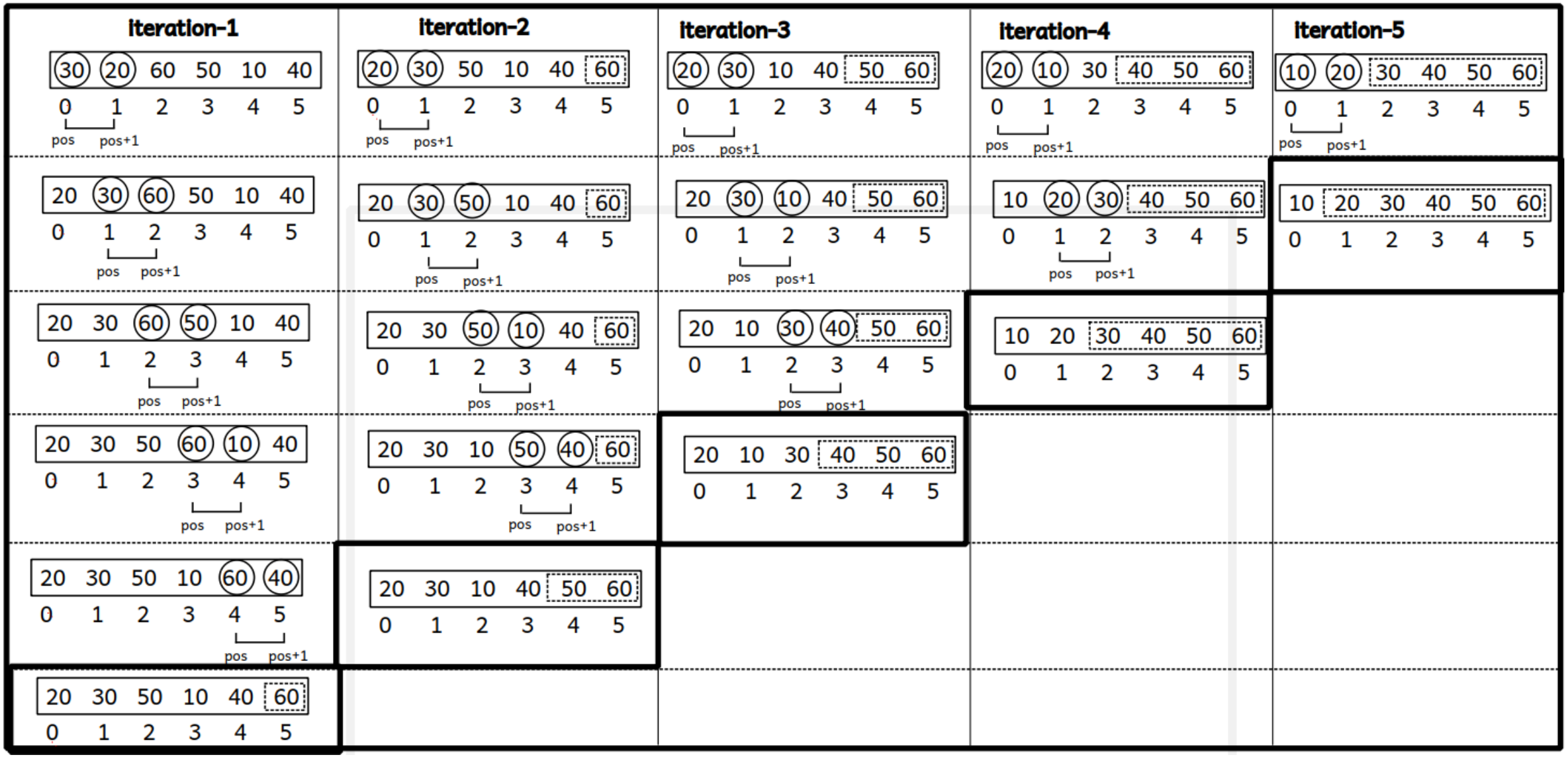
Data Structures: Sorting Algorithms

Best Case : $\Omega(n^2)$
Worst Case : $O(n^2)$
Average Case : $\theta(n^2)$

2. Bubble Sort:

- In this algorithm, **in every iteration elements which are at two consecutive positions gets compared, if they are already in order then no need of swapping between them, but if they are not in order i.e. if prev position element is greater than its next position element then swapping takes place**, and by this logic in first iteration largest element gets setteled at last position, in second iteration second largest element gets setteled at second last position and so on, **in max (n-1) no. of iterations all elements gets arranged in a sorted manner.**

Data Structures: Sorting Algorithms



Data Structures: Sorting Algorithms

Best Case : Ω(n) – if array elements are already arranged in a sorted manner.

Worst Case : O(n²)

Average Case : θ(n²)

3. Insertion Sort:

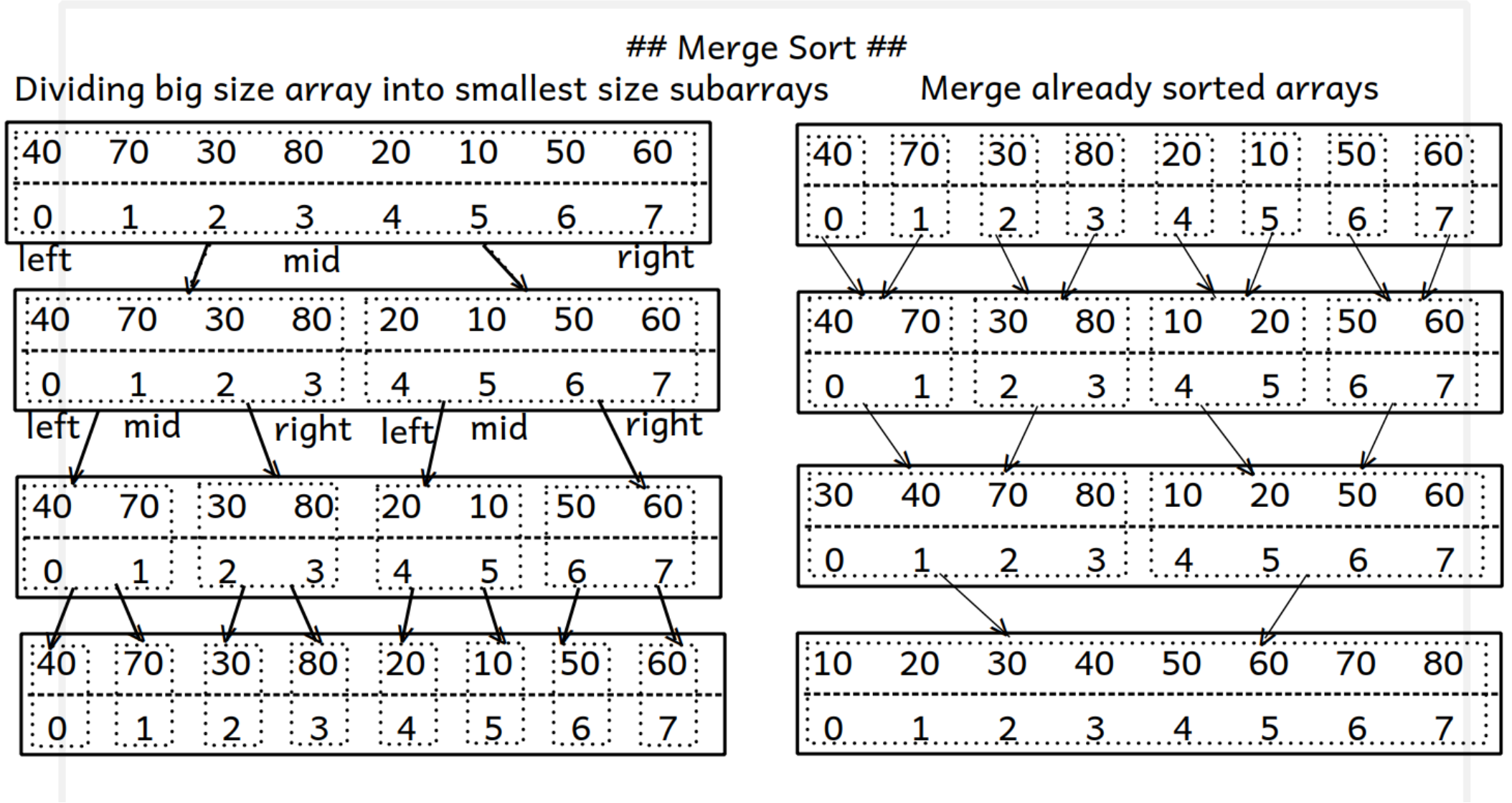
- In this algorithm, in every iteration one element gets selected as a **key element** and key element gets inserted into an array at its appropriate position towards its left hand side elements in a such a way that elements which are at left side are arranged in a sorted manner, and so on, in max **(n-1)** no. of iterations all array elements gets arranged in a sorted manner.
- **This algorithm works efficiently for already sorted input sequence by design** and hence running time of an algorithm is O(n) and it is considered as a best case.

Data Structures: Sorting Algorithms

Best Case : Ω(n) – if array elements are already arranged in a sorted manner.
Worst Case : O(n^2)
Average Case: θ(n^2)
- Insertion sort algortihm is an efficient algorithm for smaller input size array.

4. Merge Sort:
- This algorithm follows **divide-and-conquer** approach.
- In this algorithm, big size array is divided logically into smallest size (i.e. having size 1) subarrays, as if size of subarray is 1 it is sorted, after dividing array into sorted smallest size subarray's, subarrays gets merged into one array step by step in a sorted manner and finally all array elements gets arranged in a sorted manner.
- This algorithm works fine for **even** as well **odd** input size array.
- This algorithm takes extra space to sort array elements, and hence its space complexity is more.

Data Structures: Sorting Algorithms



Data Structures: Sorting Algorithms

Best Case : Ω(n log n)
Worst Case : O(n log n)
Average Case : θ(n log n)

5. Quick Sort:
- This algorithm follows **divide-and-conquer** approach.
- In this algorithm the basic logic is a **partitioning.**
- Partitioning: in parititioning, pivot element gets selected first (it may be either leftmost or rightmost or middle most element in an array), after selection of pivot element all the elements which are smaller than pivot gets arranged towards as its left as possible and elements which are greater than pivot gets arranged as its right as possible, and big size array is divided into two subarray's, so after first pass pivot element gets settled at its appropriate position, elements which are at left of pivot is referred as **left partition** and elements which are at its right referred as a **right partition.**

Data Structures: Sorting Algorithms

Best Case : Ω(n log n)

Worst Case : O(n^2) – worst case rarely occures

Average Case : θ(n log n)

- Quick sort algortihm is an efficient sorting algorithm for larger input size array.

Data Structures: Linked List

- Limitations of an array data structure:

1. Array is static, i.e. size of an array is fixed, its size cannot be either grow or shrink during runtime.

2. Addition and deletion operations on an array are not efficient as it takes O(n) time, and hence to overcome these two limitations of an Array data structure **Linked List** data structure has been designed.

Linked List: It is a basic/linear data structure, which is a collection/list of logically related similar type of elements in which, an address of first element in a collection/list is stored into a pointer variable referred as a head pointer and each element contains actual data and link to its next element i.e. an address of its next element (as well as an addr of its previous element).

- An element in a Linked List is also called as a **Node.**

- Four types of linked lists are there: **Singly Linear Linked List, Singly Circular Linked List, Doubly Linear Linked List and Doubly Circular Linked List.**

Data Structures: Linked List

- Basically we can perform **addition, deletion, traversal** etc... operations onto the linked list data structure.

- We can add and delete node into and from linked list by three ways:

add node into the linked list **at last position**, **at first position** and **at any specific position**, simillarly we can delete node from linked list which is **at first position**, **at last position** and **at any specific position.**

1. Singly Linear Linked List: It is a type of linked list in which

- head always contains an address of first element, if list is not empty.

- each node has two parts:

i. data part : it contains actual data of any primitive/non-primitive type.

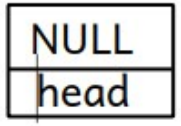
ii. pointer part (next) : it contains an address of its next element/node.

- last node points to NULL, i.e. next part of last node contains NULL.

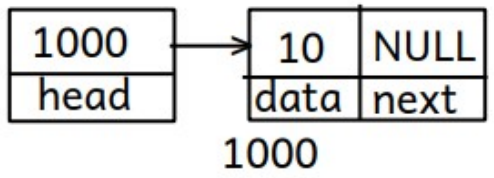
Data Structures: Linked List

SINGLY LINEAR LINKED LIST

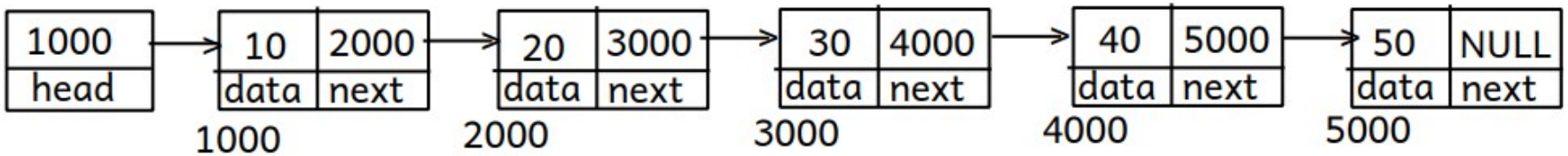
1) singly linear linked list --> list is empty



2) singly linear linked list --> list contains only one node



3) singly linear linked list --> list contains more than one nodes



Data Structures: Linked List

Limitations of Singly Linear Linked List:

- Add node at last position & delete node at last position operations are not efficient as it takes O(n) time.
- We can start traversal only from first node and can traverse the list only in a forward direction.
- Previous node of any node cannot be accesed from it.
- **Any node cannot be revisited** – to overcome this limitation Singly Circular Linked List has been designed.

2. Singly Circular Linked List: It is a type of linked list in which

- head always contains an address of first node, if list is not empty.
- each node has two parts:

i. data part : contains data of any primitive/non-primitive type.

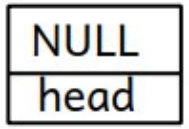
ii. pointer part(next) : contains an address of its next node.

- last node points to first node, i.e. next part of last node contains an address of first node.

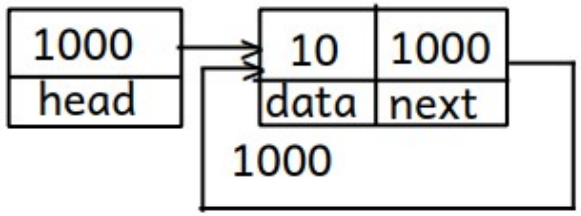
Data Structures: Linked List

SINGLY CIRCULAR LINKED LIST

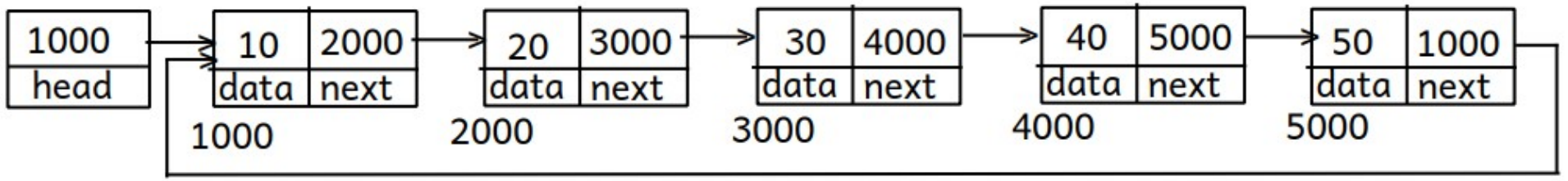
1) singly circular linked list --> list is empty



2) singly circular linked list --> list contains only one node



3) singly circular linked list --> list contains more than one nodes



Data Structures: Linked List

Limitations of Singly Circular Linked List:

- Add last, delete last & add first, delete first operations are not efficient as it takes O(n) time.

- We can starts traversal only from first node and can traverse the SCLL only in a forward direction.

- **Previous node of any node cannot be accesed from it** – to overcome this limitation Doubly Linear Linked List has been designed.

3. Doubly Linear Linked List: It is a linked list in which

- head always contains an address of first element, if list is not empty.

- each node has three parts:

i. data part: contains data of any primitive/non-primitive type.

ii. pointer part(next): contains an address of its next element/node.

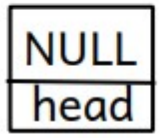
iii .pointer part(prev): contains an address of its previous element/node.

- next part of last node & prev part of first node points to NULL.

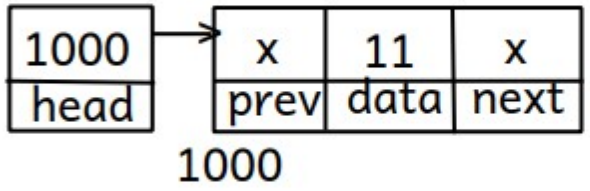
Data Structures: Linked List

DOUBLY LINEAR LINKED LIST

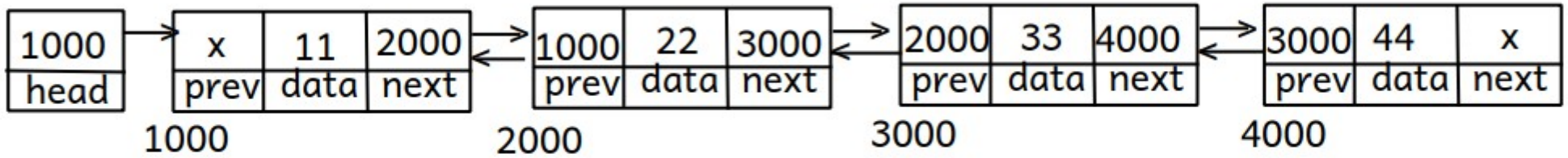
1. doubly linear linked list --> list is empty



2. doubly linear linked list --> list is contains only one node



3. doubly linear linked list --> list is contains more than one nodes



Data Structures: Linked List

Limitations of Doubly Linear Linked List:

- **Add last and delete last** operations are not efficient as it takes **O(n)** time.

- We can starts traversal only from first node, and hence to overcome these limitations **Doubly Circular Linked List** has been designed.

4. Doubly Circular Linked List: It is a linked list in which

- head always contains an address of first node, if list is not empty.

- each node has three parts:

i. data part: contains data of any primitive/non-primitive type.

ii. pointer part(next): contains an address of its next element/node.

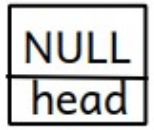
iii .pointer part(prev): contains an address of its previous element/node.

- **next part of last node contains an address of first node** & **prev part of first node contains an address of last node.**

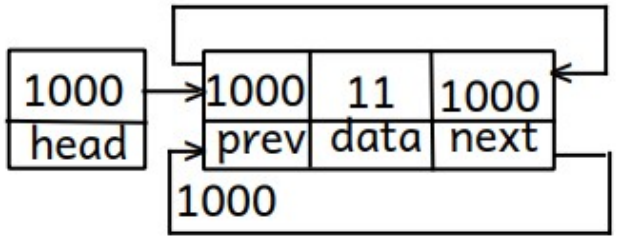
Data Structures: Linked List

DOUBLY CIRCULAR LINKED LIST

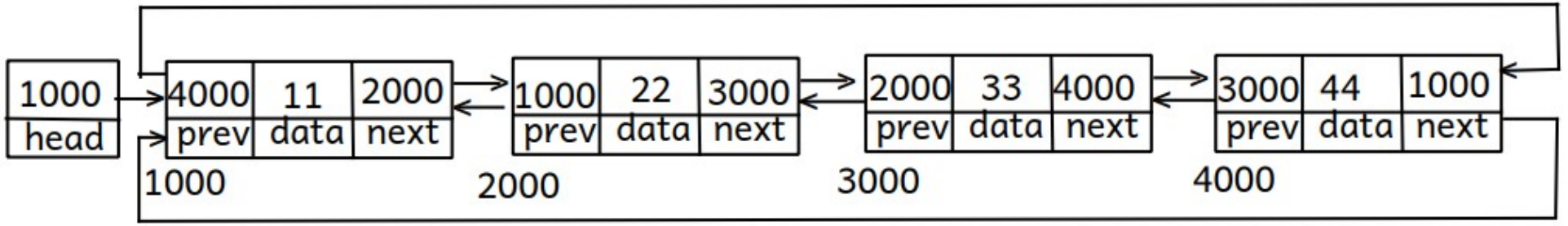
1. doubly circular linked list --> list is empty



2. doubly circular linked list -> list is contains only one node



3. doubly circular linked list --> list is contains more than one nodes



Data Structures: Linked List

Advantages of Doubly Circular Linked List:

- DCLL can be traverse in forward as well as in a backward direction.

- **Add last, add first, delete last & delete first** operations are efficient as it takes **O(1)** time and are convenient as well.

- Traversal can be start either from first node (i.e. from head) or from last node (from head.prev) in O(1) mtime.

- Any node can be revisited.

- Previous node of any node can be accessed from it

 # Array v/s Linked List => Data Structure:

- Array is **static** data structure whereas linked list is **dynamic** data structure.

- Array elements can be accessed by using **random access method** which is **efficient** than **sequential access method** used to access linked list elements.

- **Addition & Deletion operations are efficient** on linked list than on an array.

- Array elements gets stored into the **stack section**, whereas linked list elements gets stored into **heap section.**

- In a linked list **extra space is required to maintain link between elements**, whereas in an array to maintained link between elements is the job of the **compiler**.
- searching operation is faster on an array than on linked list as on linked list we cannot apply binary search.

Data Structures: Stack

Stack: It is a collection/list of logically related similar type elements into which data elements can be added as well as deleted from only one end referred **top** end.

- In this collection/list, element which was inserted last only can be deleted first, so this list works in **last in first out/first in last out** manner, and hence it is also called as **LIFO list**/FILO list.

- We can perform basic three operations on stack in **O(1)** time: **Push, Pop & Peek.**

1. Push : **to insert/add an element onto the stack at top position**

step1: check stack is not full

step2: increment the value of top by 1

step3: insert an element onto the stack at top position.

2. Pop : to delete/remove an element from the stack which is at top position

step1: check stack is not empty

step2: decrement the value of top by 1.

Data Structures: Stack

3. Peek : to get the value of an element which is at top position without push & pop.

step1: check stack is not empty

step2: return the value of an element which is at top position

Stack Empty : top == -1

Stack Full : top == SIZE-1

 # Applications of Stack:

- Stack is used by an OS to control of flow of an execution of program.
- In recursion internally an OS uses a stack.
- undo & redo functionalities of an OS are implemented by using stack.
- Stack is used to implement advanced data structure algorithms like **DFS: Depth First Search** traversal in tree & graph.
- Stack is used in an algorithms to covert given infix expression into its equivalent postfix and prefix, and for postfix expression evaluation.

Data Structures: Stack

- Algorithm to convert given infix expression into its equivalent postfix expression:

Initially we have, an Infix expression, an empty Postfix expression & empty Stack.

```
# algorithm to convert given infix expression into its equivalent postfix expression
step1: start scanning infix expression from left to right
step2:
    if( cur ele is an operand )
        append it into the postfix expression
    else//if( cur ele is an operator )
    {
        while( !is_stack_empty(&s) && priority(topmost ele) >= priority(cur ele) )
        {
            pop an ele from the stack and append it into the postfix expression
        }

        push cur ele onto the stack
    }
step3: repeat step1 & step2 till the end of infix expression
step4: pop all remaining ele's one by one from the stack and append them into the
postfix expression.
```

Data Structures: Stack

- Algorithm to convert given infix expression into its equivalent prefix expression:

Initially we have, an Infix expression, an empty Prefix expression & empty Stack.

```
# algorithm to convert given infix expression into its equivalent prefix:
step1: start scanning infix expression from right to left
step2:
    if( cur ele is an operand )
        append it into the prefix expression
    else//if( cur ele is an operator )
    {
        while( !is_stack_empty(&s) && priority(topmost ele) > priority(cur ele) )
        {
            pop an ele from the stack and append it into the prefix expression
        }

        push cur ele onto the stack
    }
step3: repeat step1 & step2 till the end of infix expression
step4: pop all remaining ele's one by one from the stack and append them into the
prefix expression.
step5: reverse prefix expression - equivalent prefix expression.
```

Data Structures: Queue

Queue: It is a collection/list of logically related similar type of elements into which elements can be added from one end referred as **rear** end, whereas elements can be deleted from another end referred as a **front** end.

- In this list, element which was inserted first can be deleted first, so this list works in **first in first out** manner, hence this list is also called as **FIFO list/LILO list.**

- Two basic operations can be performed on queue in O(1) time.

1. Enqueue: to insert/push/add an element into the queue from rear end.

2. Dequeue: to delete/remove/pop an element from the queue which is at front end.

- There are different types of queue:

1. Linear Queue (works in a fifo manner)

2. Circular Queue (works in a fifo manner)

3. Priority Queue: it is a type of queue in which elements can be inserted from rear end randomly (i.e. without checking priority), whereas an element which is having highest priority can only be deleted first.

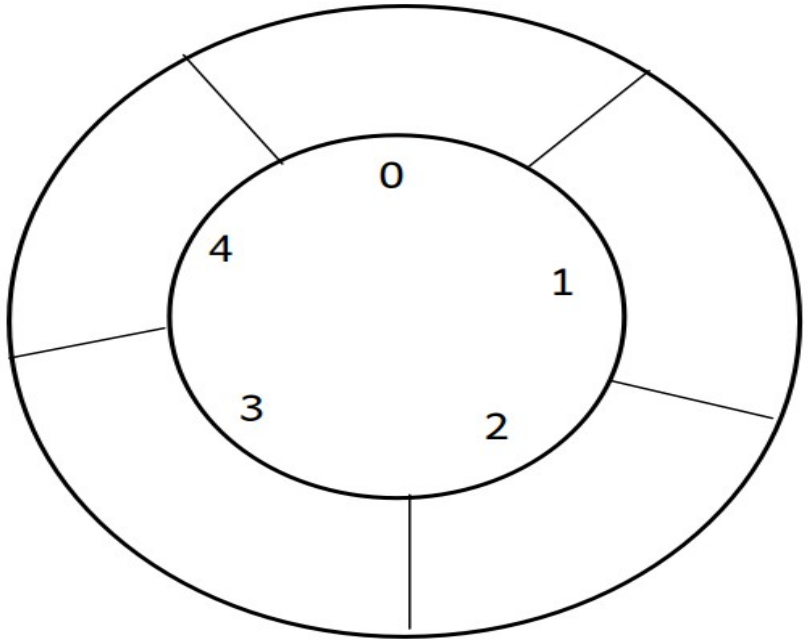
- Priority queue can be implemented by using linked list, whereas it can be implemented efficiently by using **binary heap.**

4. Double Ended Queue (deque) : it is a type of queue in which elements can added as well as deleted from both the ends.

Data Structures: Queue

front=-1

rear=-1



Circular Queue

is_queue_full : front == (rear+1)%SIZE

is_queue_empty : rear == -1 && front == rear

1. "enqueue": to insert/add/push an element into the queue from rear end:

step1: check queue is not full

step2: increment the value of rear by 1 [rear = (rear+1)%SIZE]

step3: push/add/insert an ele into the queue at rear position

step4: if(front == -1)

front = 0

2. "dequeue": to remove/delete/pop an element from the queue which is at front position.

step1: check queue is not empty

step2:

if(front == rear)//if we are deleting last ele

front = rear = -1;

else

increment the value of front by 1 [i.e. we are deleting an ele from the queue]. [front = (front+1)%SIZE]

Data Structures: Queue

Applications of Queue:

- Queue is used to implement OS data structures like **job queue, ready queue, message queue, waiting queue** etc...

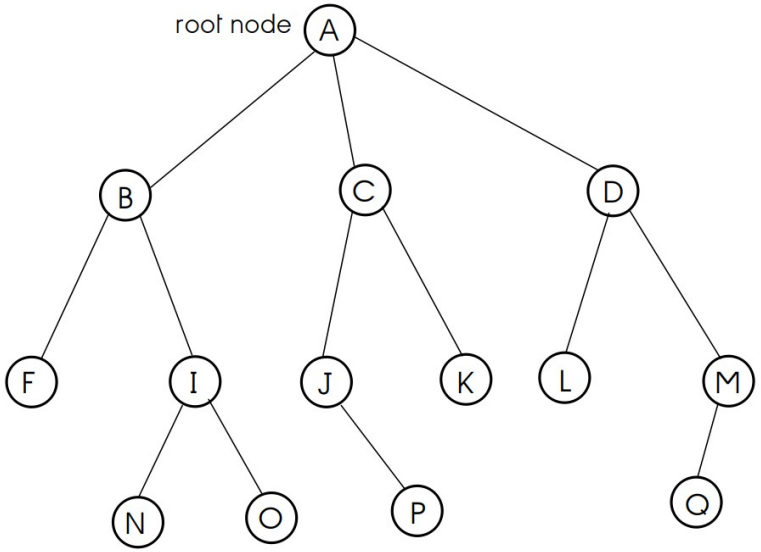
- Queue is used to implement OS algorithms like **FCFS CPU Scheduling, Priority CPU Scheduling, FIFO Page Replacement** etc...

- Queue is used to implenent an advanced data structure algorithms like **BFS: Breadth First Search** Traversal in tree and graph.

- Queue is used in any application/program in which list/collection of elements should works in a **first in first out manner or whereaver it should works according to priority.**

Data Structures: Tree

Tree: It is a **non-linear / advanced data structure** which is a **collection of finite no. of logically related similar type of data elements** in which, there is a first specially designated element referred as a **root element**, and remaining all elements are connected to it in a **hierarchical manner**, follows **parent-child relationship**.



Tree: Data Structure

Data Structures: Tree

- **siblings/brothers:** child nodes of same parent are called as siblings.
- **ancestors:** all the nodes which are in the path from root node to that node.
- **descedents:** all the nodes which can be accessible from that node.
- **degree of a node** = no. of child nodes having that node
- **degree of a tree** = max degree of any node in a given tree
- **leaf node/external node/terminal node:** node which is not having any child node OR node having degree 0.
- **non-leaf node/internal node/non-terminal node:** node which is having any no. of child node/s OR node having non-zero degree.
- **level of a node** = level of its parent node + 1
- **level of a tree** = max level of any node in a given tree (by assuming level of root node is at level 0).
- **depth of a tree** = max level of any node in a given tree.
- as tree data structure can grow upto any level and any node can have any number of child nodes, operations on it becomes unefficient, so restrictions can be applied on it to achieve efficiency and hence there are diefferent types of tree.

Data Structures: Tree

- Binary tree: it is a tree in which each node can have max 2 number of child nodes, i.e. each node can have either 0 OR 1 OR 2 number of child nodes.

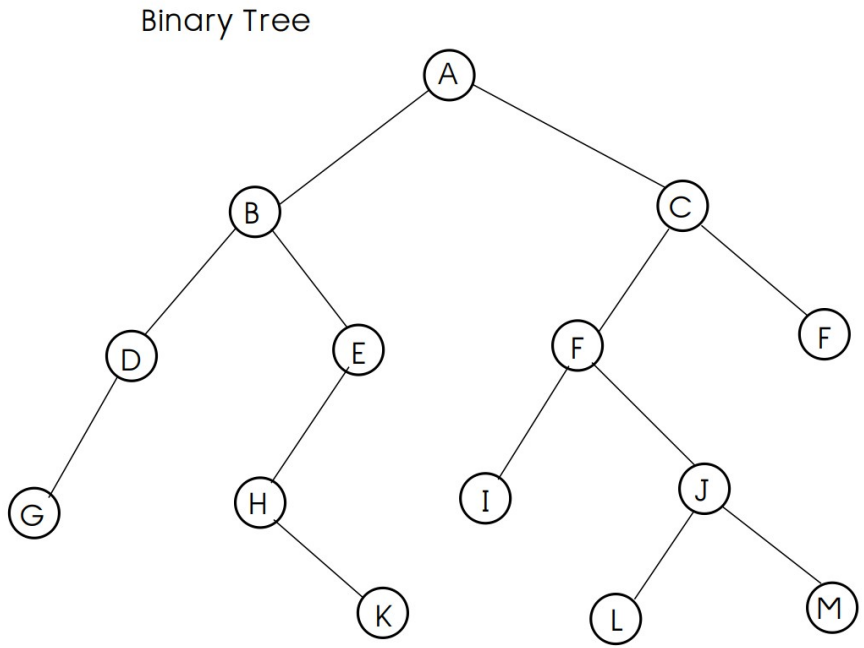
OR

Binary tree: it is a set of finite number of elements having three subsets:

1. root element

2. left subtree (may be empty)

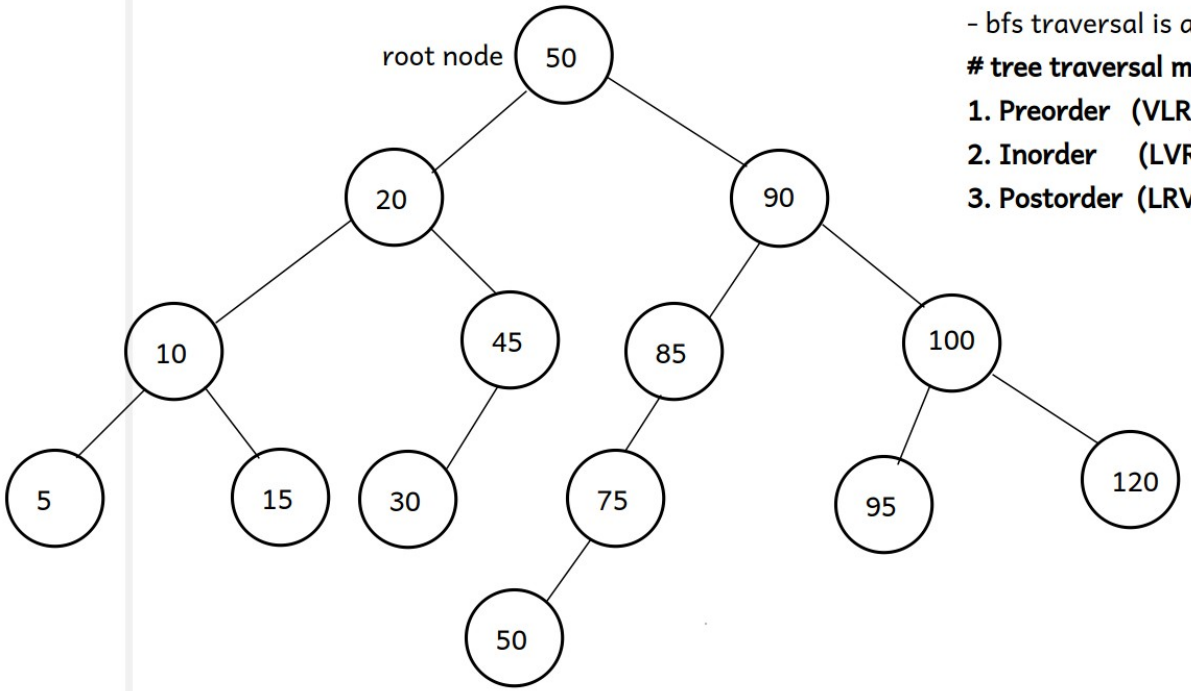
3. right subtree (may be empty)



Data Structures: Tree

- **Binary Search Tree(BST):** it is a **binary tree** in which left child is always smaller than its parent and right child is always greater than or equal to its parent.

input order of an ele's for BST: 50 20 90 85 10 45 30 100 15 75 95 120 5 50



1. dfs traversal: 50 20 10 5 15 45 30 90 85 75 50 100 95 120

2. bfs traversal: 50 20 90 10 45 85 100 5 15 30 75 95 120 50

- bfs traversal is also called as "levelwise traversal".

 # tree traversal methods on BST:

1. Preorder (VLR) : 50 20 10 5 15 45 30 90 85 75 50 100 95 120

2. Inorder (LVR): 5 10 15 20 30 45 50 50 75 85 90 95 100 120

3. Postorder (LRV): 5 15 10 30 45 20 50 75 85 95 120 100 90 **50**

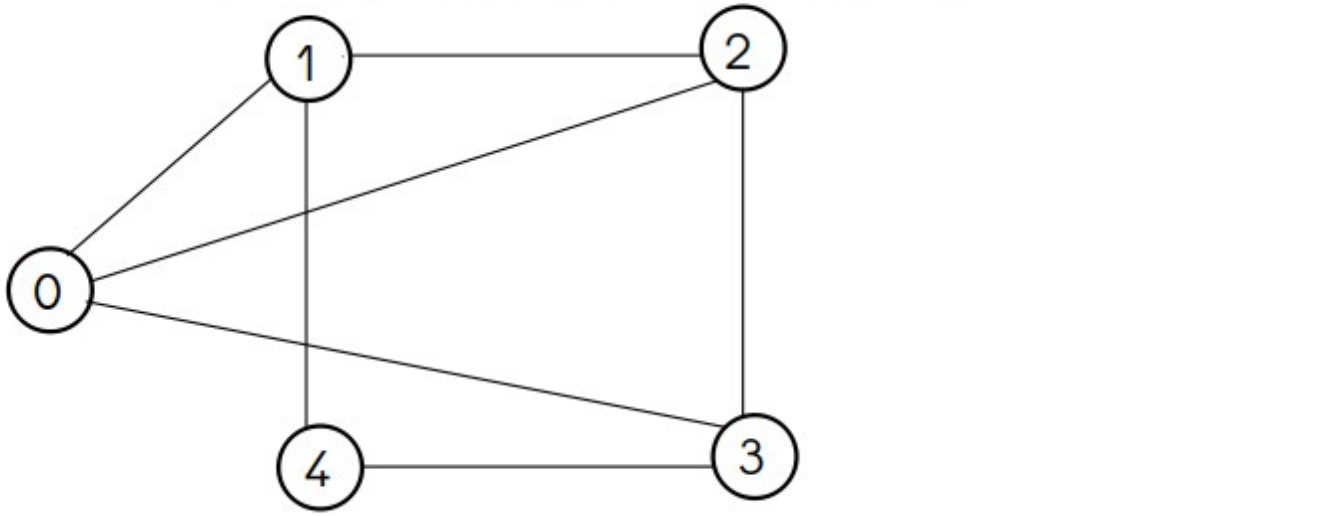
Data Structures: Graph

Graph: It is **non-linear**, **advanced** data structure, which is a collection of logically related similar and disimmilar type of elements which contains:

- set of finite no. of elements referred as a **vertices**, also called as **nodes**, and

- set of finite no. of ordered/unordered pairs of vertices referred as an **edges**, also called as an **arcs**, whereas it may carries weight/cost/value (cost/weight/value may be -ve).

G(V,E): V={0,1,2,3,4}; E={ (0,1),(0,2),(0,3),(1,2),(1,4),(2,3),(3,4) }



Data Structures: Graph

- If there exists a direct edge between two vertices then those vertices are referred as **adjacent vertices** otherwiese **non-adjacent**.

- if we can represent any edge either (u,v) OR (v,u) then it is referred as **unordered pair of vertices i.e. undirected edge.**

(u,v) == (v,u) --> unordered pair of vertices -> undirected edge -> undirected graph

- if we cannot represent any edge either (u,v) OR (v,u) then it is referred as **unordered pair of vertices** i.e. directed edge.

(u,v) != (v,u) --> ordered pair of vertices -> directed edge -> directed graph (di-graph).

- complete graph: if all the vertices are adjacent to remaining all vertices in a given graph.

- connected vertices: if path exists between two vertices then those two vertices are referred as a connected vertices otherwise not-connected.

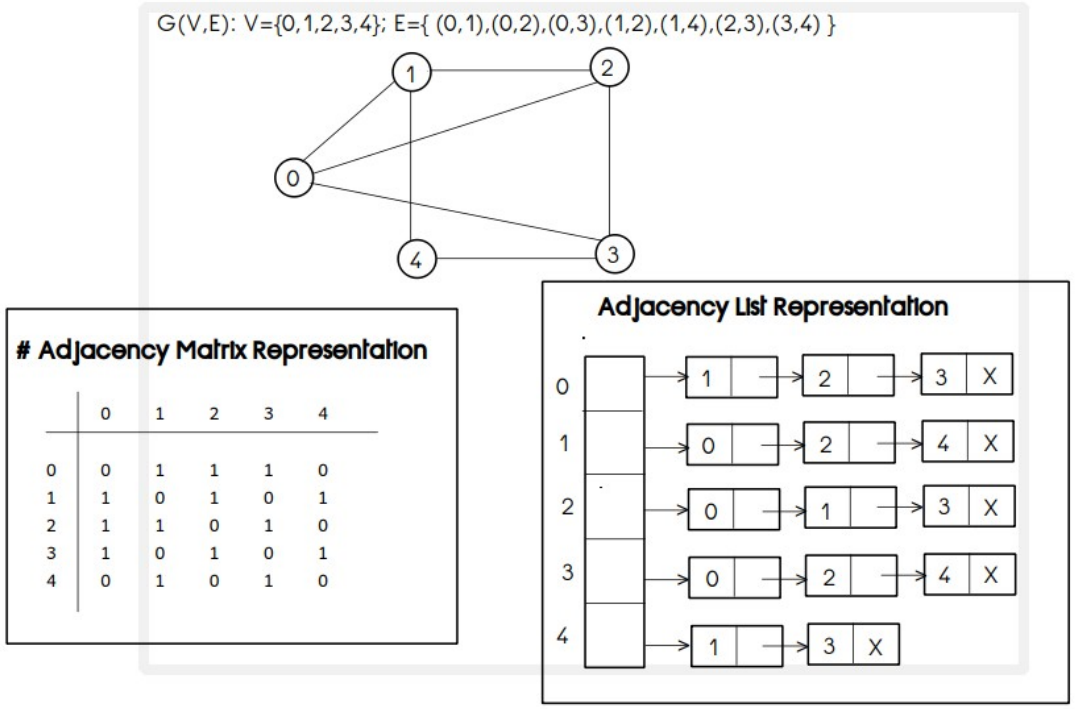
- connected graph: if any vertex is connected to remaining all vertices in a given graph

Data Structures: Graph

- There are two graph representation methods:

1. Adjacency Matrix Representation (2-D Array)

2. Adjacency List Representation (Array of Linked Lists)



Data Structures: Hash Table

Hash Table: it is a **non-linear/advanced data structure** which is a **collection of finite number of logically related similar type of data elements/records** gets stored into the memory in an **associative manner i.e. in a key-value pairs** (for faster searching).

Hashing: It is an improvement over **"Direct Access Table"** in which hash function can be used and the table is reffered as "Hash Table".

Hash Function: it is a function that **converts a given big key value/number into a small practical integer value/key** which is reffered as **hash key/hash code** which is a mapped value can be used as an index in a hash table.

Collision: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some **collision handling technique.**

- There are two **collission handling techniques:**

1. Chaining/Seperate Chaining

2. Open Addressing

Data Structures: Hash Table

1. Chaining:

- The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.

- Advantages:

1. Simple to implement.
2. Hash table never fills up, we can always add more elements to the chain.
3. Less sensitive to the hash function or load factors.
4. It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

- Disadvantages:

1. Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
2. Wastage of Space (Some Parts of hash table are never used).
3. If the chain becomes long, then search time can become O(n) in the worst case.
4. Uses extra space for links.

Data Structures: Hash Table

2. Open Addressing:
- In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.
- Open Addressing is done following ways:

A. Linear Probing:
- In linear probing, we linearly probe/search for next slot.

For example, typical gap between two probes is 1 as taken in below example also.
- let hash(x) be the slot index computed using hash function and S be the table size

If slot **hash(x) % S** is full, then we try **(hash(x) + 1) % S**
If **(hash(x) + 1) % S** is also full, then we try **(hash(x) + 2) % S**
If **(hash(x) + 2) % S** is also full, then we try **(hash(x) + 3) % S**
.....
.....

Clustering: The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

Data Structures: Hash Table

B. Quadratic Probing:

- We look for i^2th slot in ith iteration.
- let **hash(x)** be the slot index computed using hash function.

If slot **hash(x) % S** is full, then we try **(hash(x) + 1*1) % S**
If **(hash(x) + 1*1) % S** is also full, then we try **(hash(x) + 2*2) % S**
If **(hash(x) + 2*2) % S** is also full, then we try **(hash(x) + 3*3) % S**
.....
.....

C. Double Hashing:

- We use another hash function **hash2(x)** and look for **i*hash2(x)** slot in i'th rotation.
- let hash(x) be the slot index computed using hash function.

If slot **hash(x) % S** is full, then we try **(hash(x) + 1*hash2(x)) % S**
If **(hash(x) + 1*hash2(x)) % S** is also full, then we try **(hash(x) + 2*hash2(x))%S**
If **(hash(x) + 2*hash2(x)) % S** is also full, then we try **(hash(x) + 3*hash2(x)) % S**
.....
.....

Hashing, Hash Function and Hash Table

What is Hashing

- Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects.
- Examples of how hashing is used in our lives:
  - In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
  - In libraries, each book is assigned a unique number that can be used to determine information about the book
- **Hashing** is the solution that can be used in almost all such situations and performs extremely well compared to above data structures like Array, Linked List, BST in practice

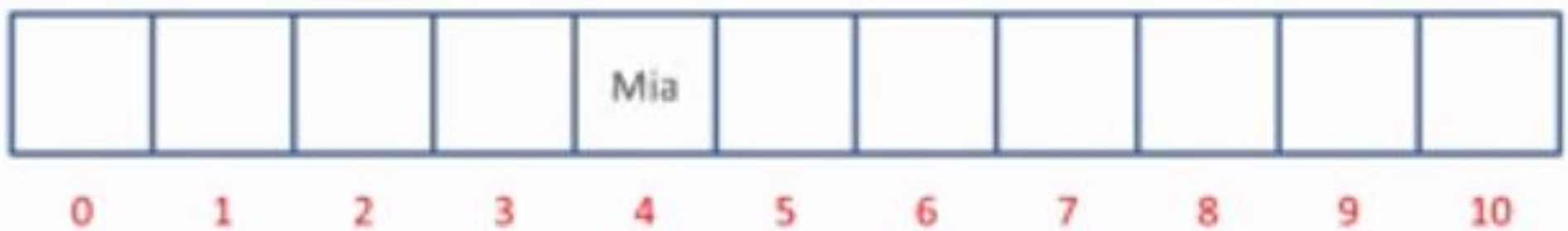
What is Hashing (Cond…)

- With hashing we get O(1) search time on average and O(n) in worst case.
- The idea is to use ***hash function*** that converts a given input number or any other key to a smaller number and uses the small number as index in a table called ***hash table***.
- Hash Function
  - Hash function maps a big number or string to a small integer that can be used as index in hash table
- Hash Table
  - An array that stores pointers to records corresponding to a given input number


0
1
2
3
4
5
6
7
8
9
10

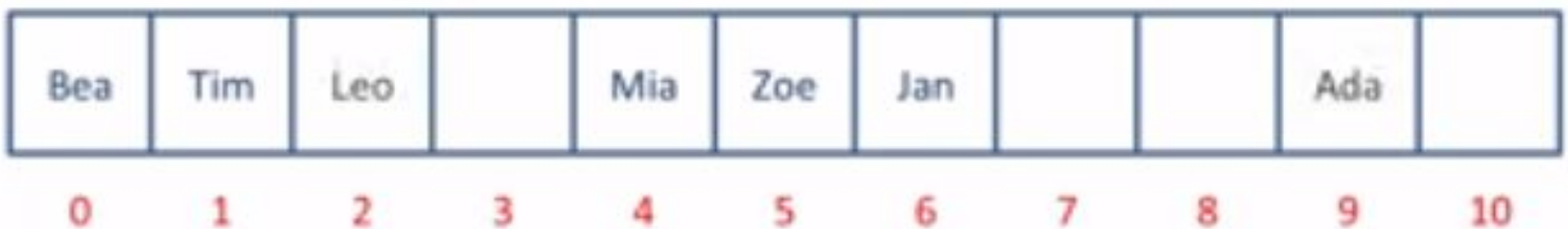
Mia	M	77	i	105	a	97	279	4

Mia	M	77	i	105	a	97	279	4	



Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1

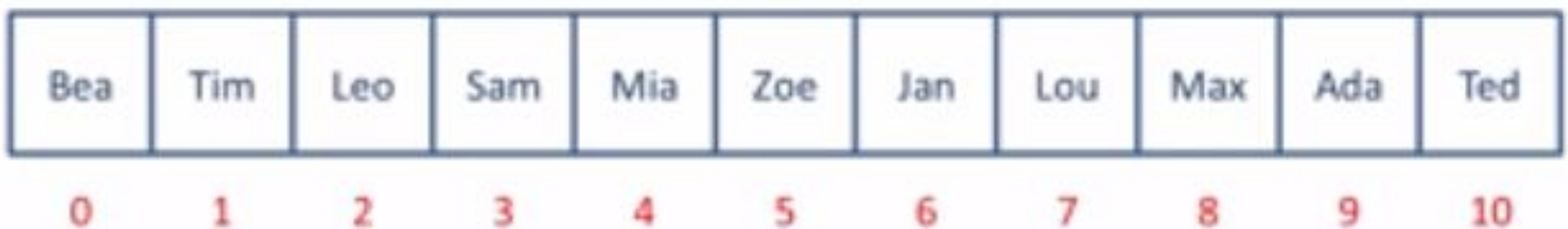
Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Jan	J	74	a	97	n	110	281	6
Ada	A	65	d	100	a	97	262	9
Leo	L	76	e	101	o	111	288	2



Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Jan	J	74	a	97	n	110	281	6
Ada	A	65	d	100	a	97	262	9
Leo	L	76	e	101	o	111	288	2
Sam	S	83	a	97	m	109	289	3
Lou	L	76	o	111	u	117	304	7
Max	M	77	a	97	x	120	294	8
Ted	T	84	e	101	d	100	285	10

Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Ada	Ted
0	1	2	3	4	5	6	7	8	9	10

Index number = *sum ASCII codes* Mod *size of array*

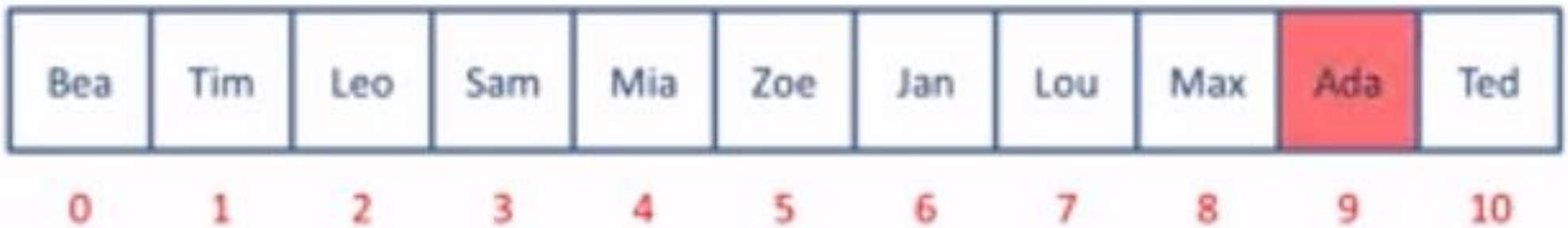


Find Ada

Ada = (65 + 100 + 97) = 262

262 Mod 11 = 9

myData = Array(9)




Bea
27/01/1941
English
Astronomer
Tim
08/06/1955
English
Inventor
Leo
31/12/1945
American
Mathematician
Sam
27/04/1791
American
Inventor
Mia
20/02/1986
Russian
Space Station
Zoe
19/06/1978
American
Actress
Jan
13/02/1956
Polish
Logician
Lou
27/12/1822
French
Biologist
Max
23/04/1858
German
Physicist
Ada
10/12/1815
English
Mathematician
Ted
17/06/1937
American
Philosopher
0
1
2
3
4
5
6
7
8
9
10

Hashing Algorithm

- Calculation applied to a key to transform it into an address
- For numeric keys, divide the key by the number of available addresses, *n*, and take the remainder

address = *key* Mod *n*

- For alphanumeric keys, divide the sum of ASCII codes in a key by the number of available addresses, *n*, and take the remainder
- Folding method divides key into equal parts then adds the parts together
  - The telephone number 01452 8345654, becomes 01 + 45 + 28 + 34 + 56 + 54 = 218
  - Depending on size of table, may then divide by some constant and take remainder

Understand Collision

Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5

Understand Collision

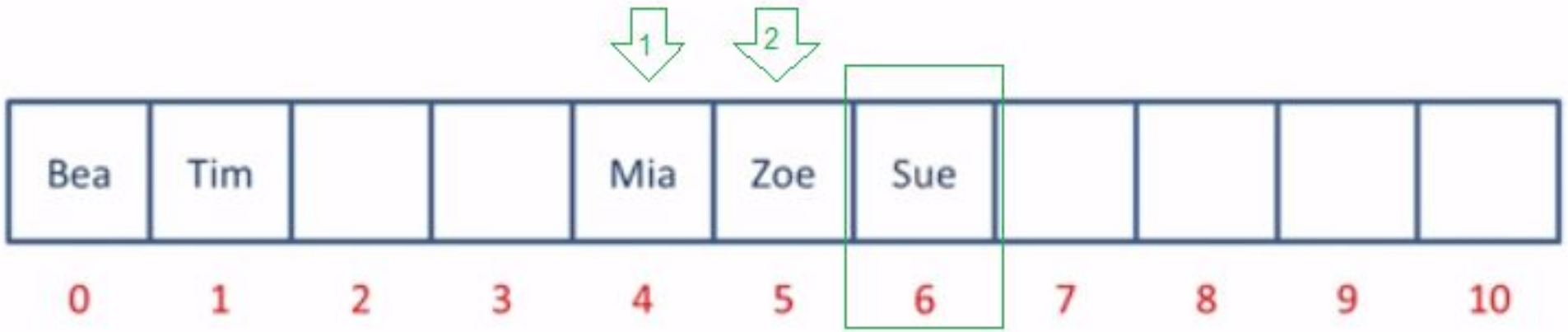
Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4

Collision Handling

- Irrespective of how good a hash function is, collisions are bound to occur.
- Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.
  - Open addressing
    - Linear Probing
    - Quadratic Probing
    - Double Hashing
  - Closed addressing

Open Addressing (Linear)

Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4



Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4
Len	L	76	e	101	n	110	287	1

Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4
Len	L	76	e	101	n	110	287	1
Moe	M	77	o	111	e	101	289	3
Lou	L	76	o	111	u	117	304	7
Rae	R	82	a	97	e	101	280	5
Max	M	77	a	97	x	120	294	8
Tod	T	84	o	111	d	100	295	9

Bea	Tim	Len	Moe	Mia	Zoe	Sue	Lou	Rae	Max	Tod
0	1	2	3	4	5	6	7	8	9	10

Find Rae

Rae = (82 + 97 + 101) = 280

280 Mod 11 = 5

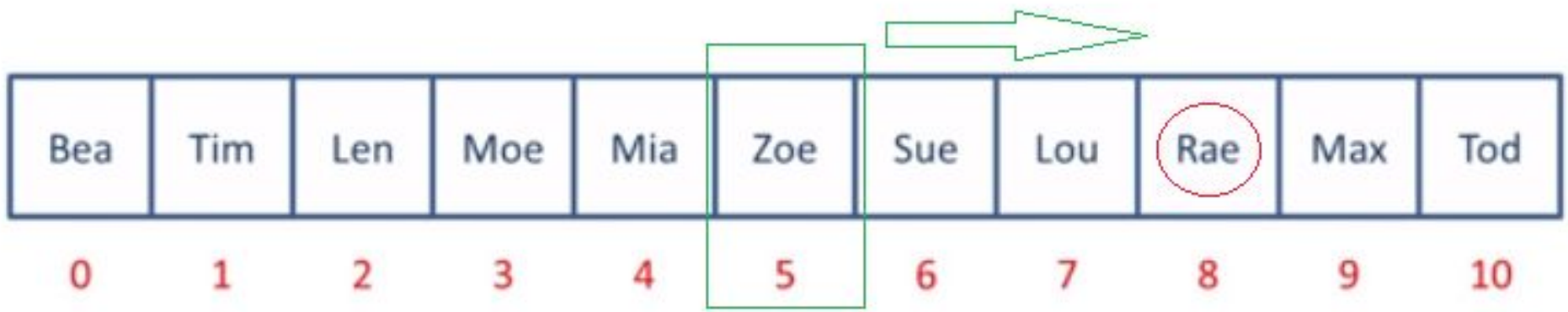
myData = Array(5)

Find Rae

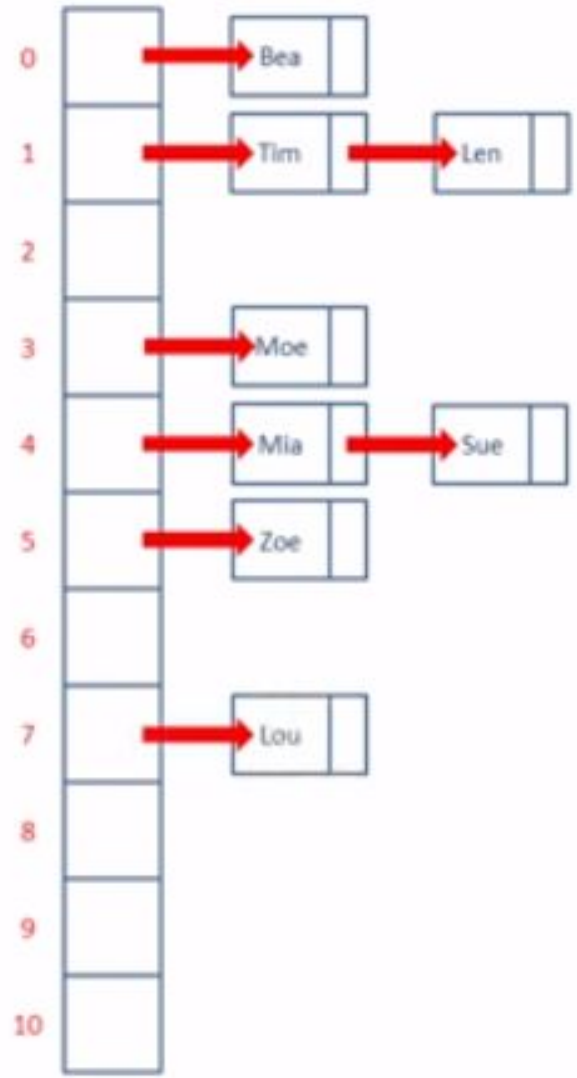
Rae = (82 + 97 + 101) = 280

280 Mod 11 = 5

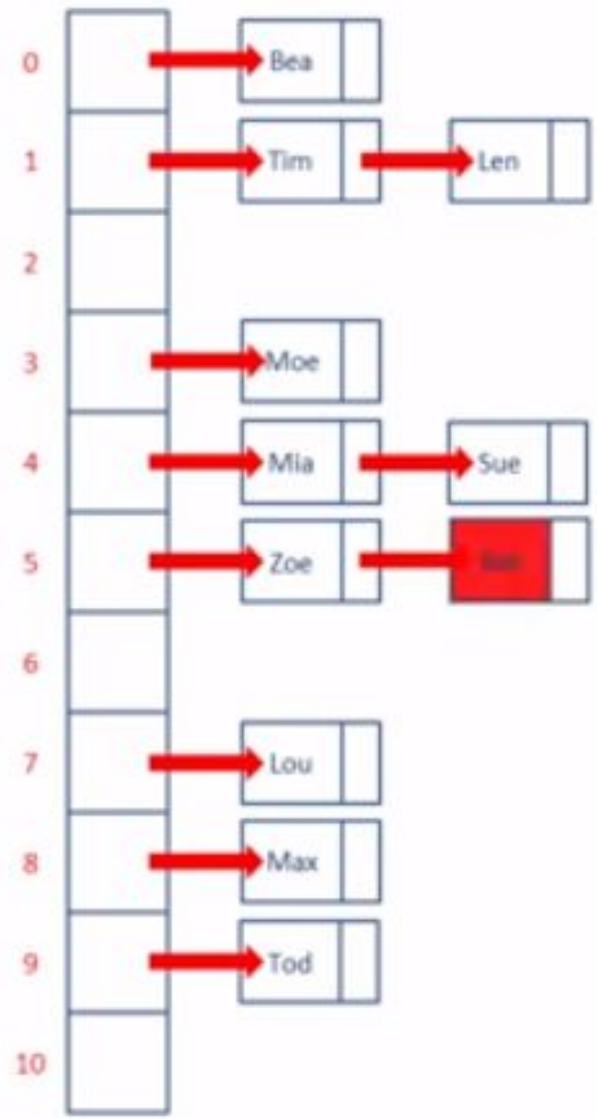
myData = Array(5)



Closed Addressing (Non-Linear)



Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4
Len	L	76	e	101	n	110	287	1
Moe	M	77	o	111	e	101	289	3
Lou	L	76	o	111	u	117	304	7



Find Rae 280 Mod 11 = 5

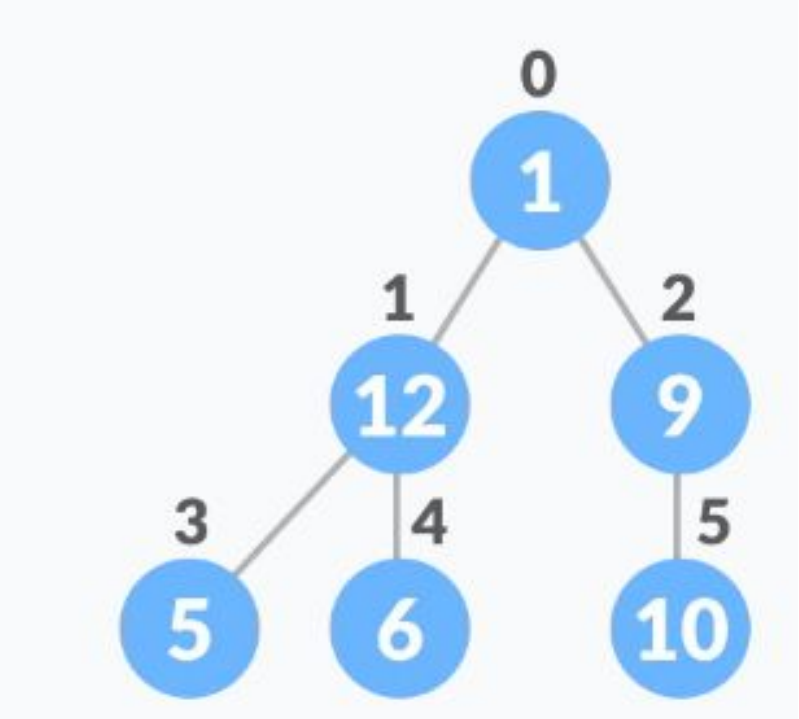
myData = Array(5)

Objectives of Hash Function

- Minimize collisions
- Uniform distribution of hash values
- Easy to calculate
- Resolve any collisions

Heapsort

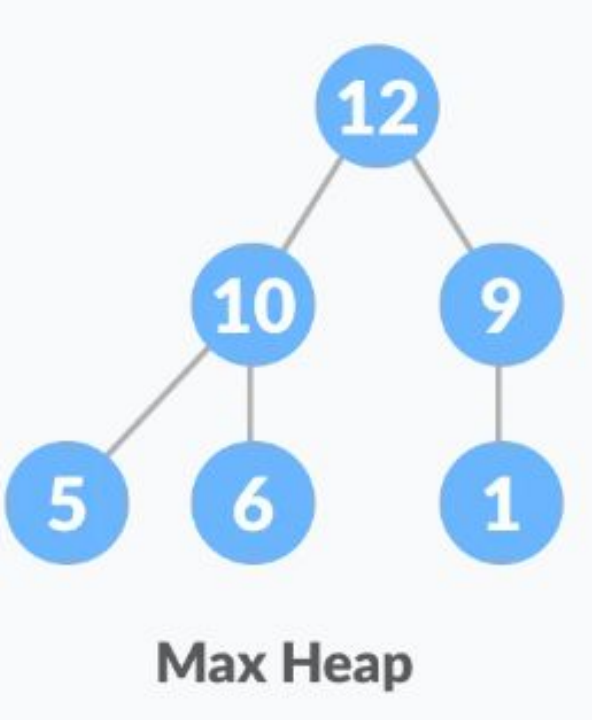
Relationship between Array Indexes and Tree Elements

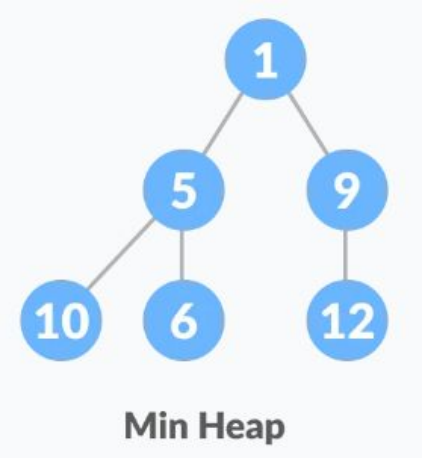


- A complete binary tree has an interesting property that we can use to find the children and parents of any node.
- If the index of any element in the array is i
  - the element in the index 2i+1 will become the left child
  - the element in the index 2i+2 will become the right child
- Also, the parent of any element at index i is given by the lower bound of (i-1)/2.

Heap Data Structure

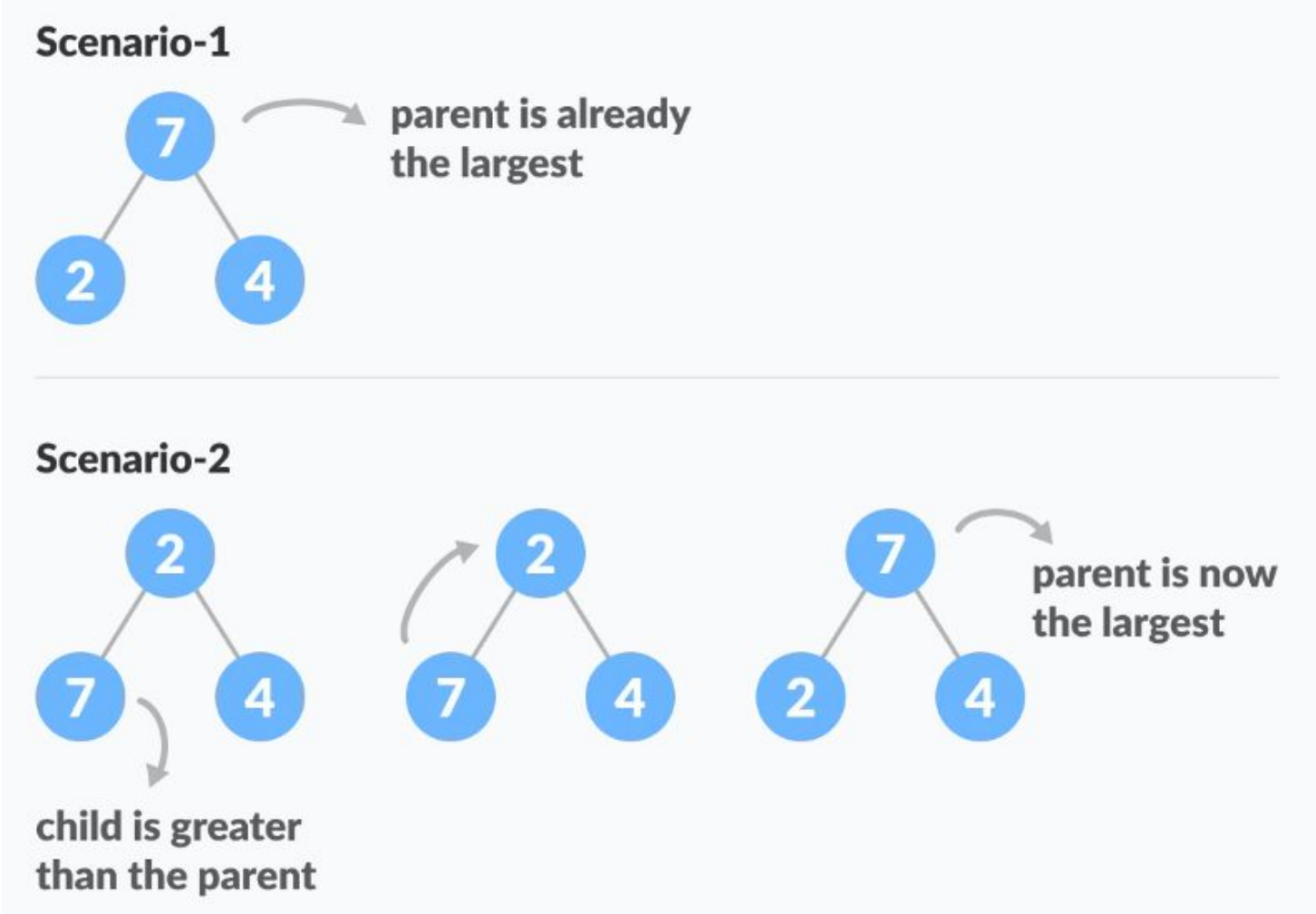
- Heap is a special tree-based data structure
- A binary tree is said to follow a heap data structure if
  - it is a complete binary tree
  - All nodes in the tree follow the property that
    - They are greater than their children : Max-Heap
    - Or all nodes are smaller than their children : Min-Heap

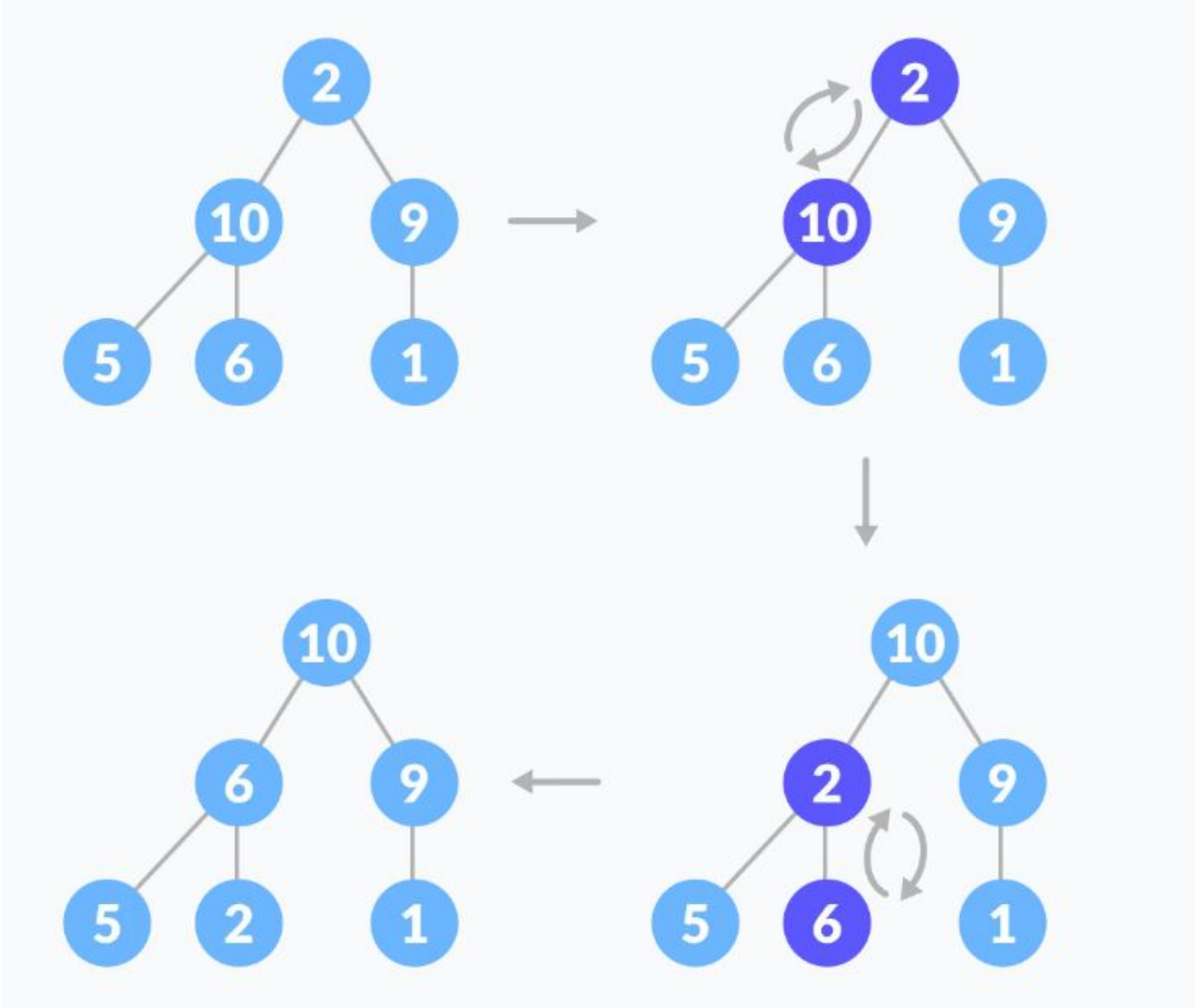




What is **Heapify**

- Heapify is **the process of creating a heap data structure from a binary tree**.




2
10
9
5
6
1
2
10
9
5
6
1
10
2
9
5
6
1
10
6
9
5
2
1

Max-Heapify Operation

Algorithm 1: Max-Heapify Pseudocode

```
Data: B: input array; s: an index of the node
Result: Heap tree that obeys max-heap property
Procedure Max-Heapify(B, s)
    left = 2s;
    right = 2s + 1;
    if left ≤ B.length and B[left] > B[s] then
        largest =left;
    else
        largest =s;
    end
    if right ≤ B.length and B[right] > B[largest] then
        largest =right;
    end
    if largest ≠ s then
        swap(B[s], B[largest]);
        Max-Heapify(B, largest);
    end
end
```

Building **max-heap**

- To build a max-heap from any tree, we can thus start heapifying each sub-tree from the bottom up and end up to the root element.
- Start our algorithm with a node that is at the lowest level of the tree and has children node (n/2 - 1)
- Continue this process and make sure all the subtrees are following the max-heap property

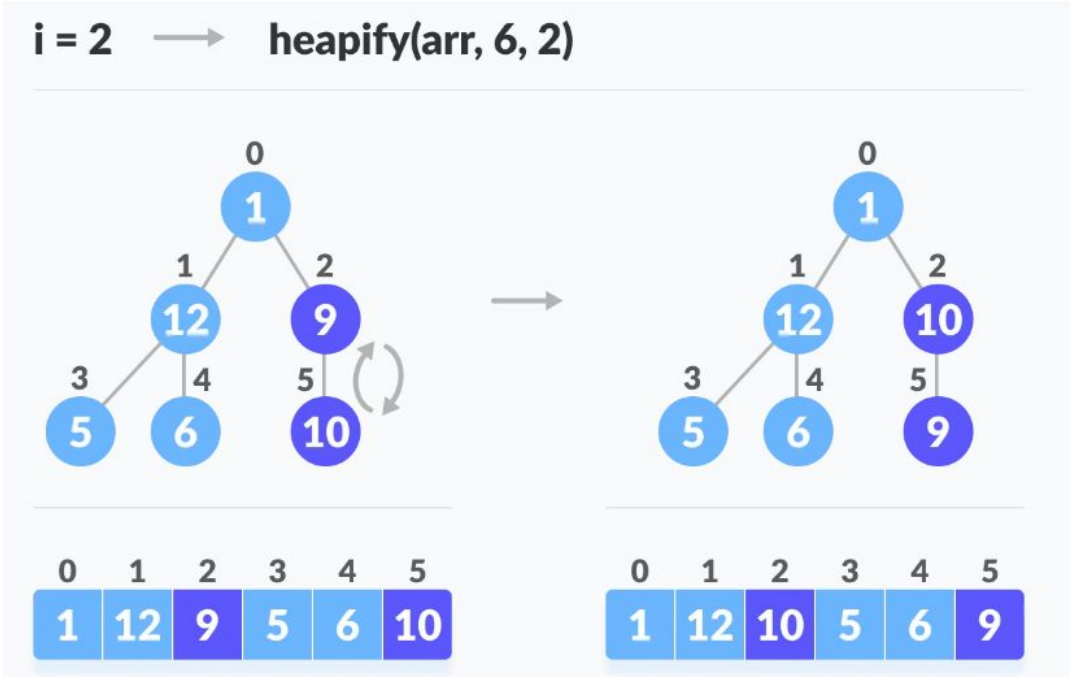
```
// Build heap (rearrange array)
for (int i = n / 2 - 1; i >= 0; i--)
  heapify(arr, n, i);
```

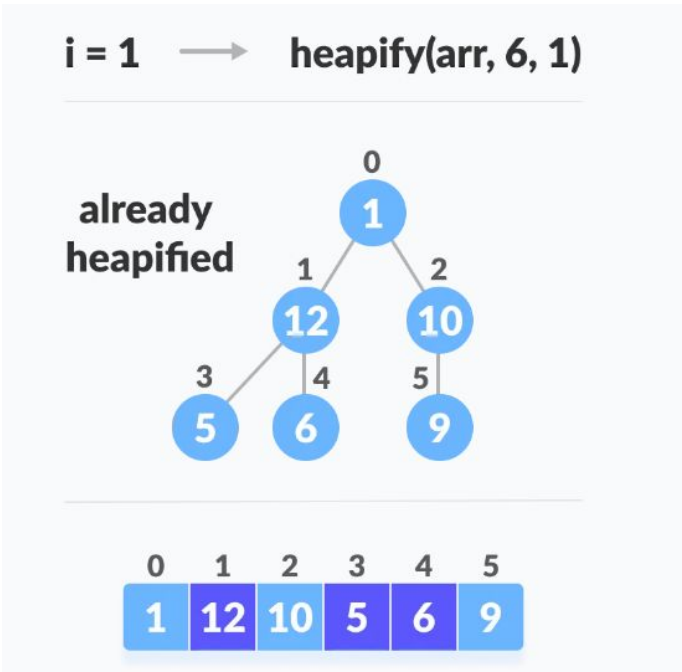
Example

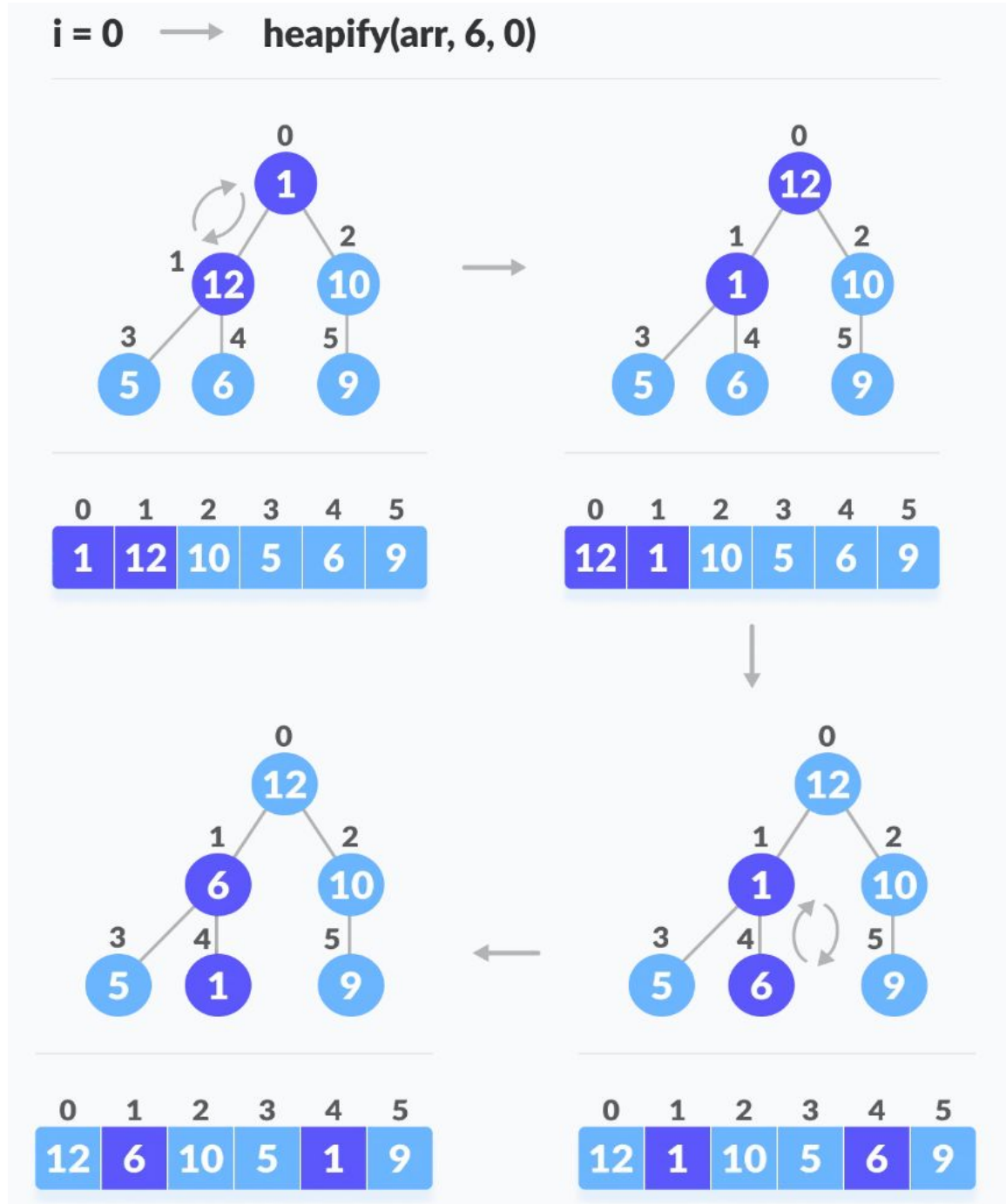
	0	1	2	3	4	5
arr	1	12	9	5	6	10

n = 6

i = 6/2 - 1 = 2 # loop runs from 2 to 0






i = 0
heapify(arr, 6, 0)
0 1 2 3 4 5
1 12 10 5 6 9
0 1 2 3 4 5
12 1 10 5 6 9
0 1 2 3 4 5
12 1 10 5 6 9
0 1 2 3 4 5
12 6 10 5 1 9

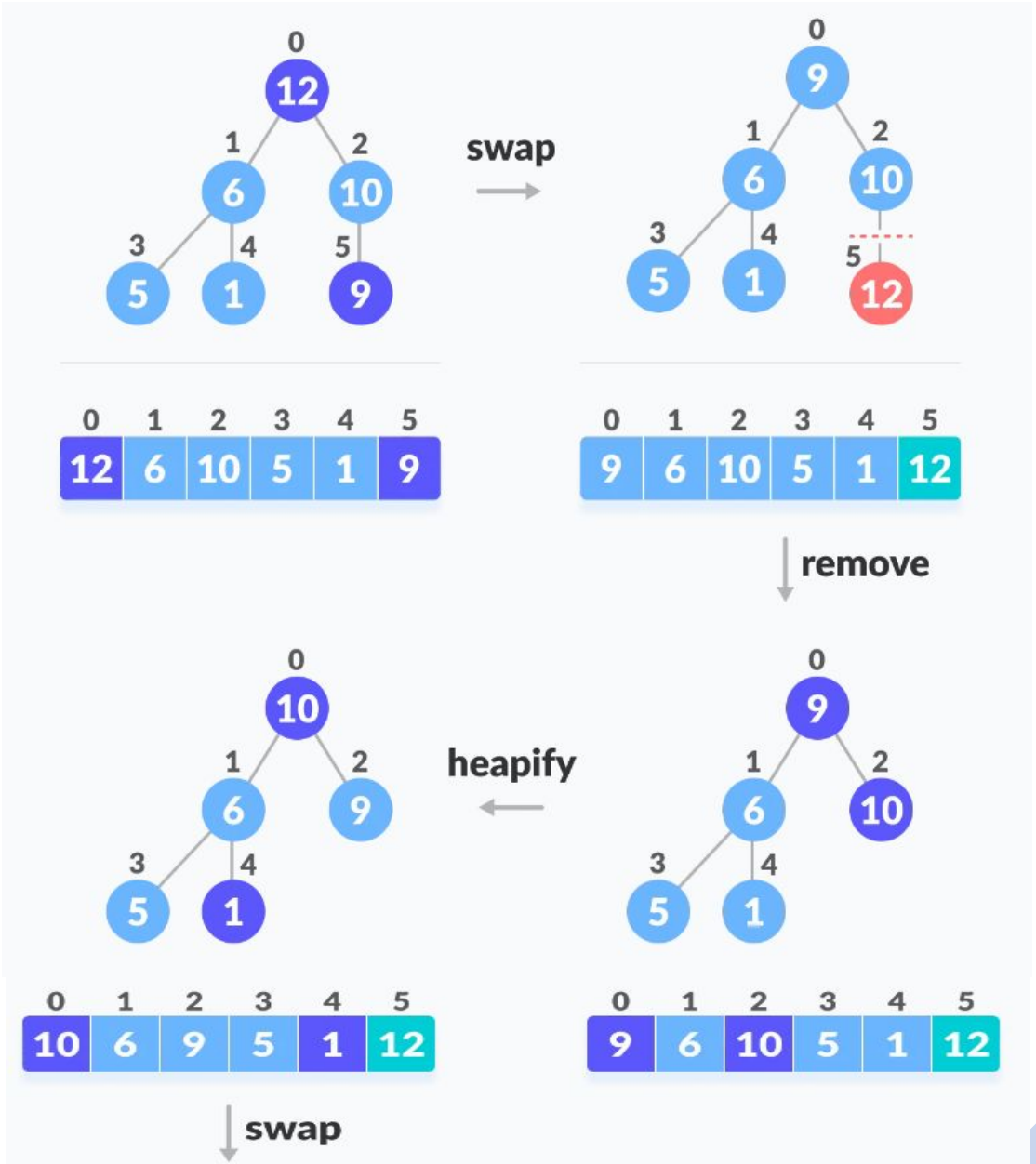
Heap Sort

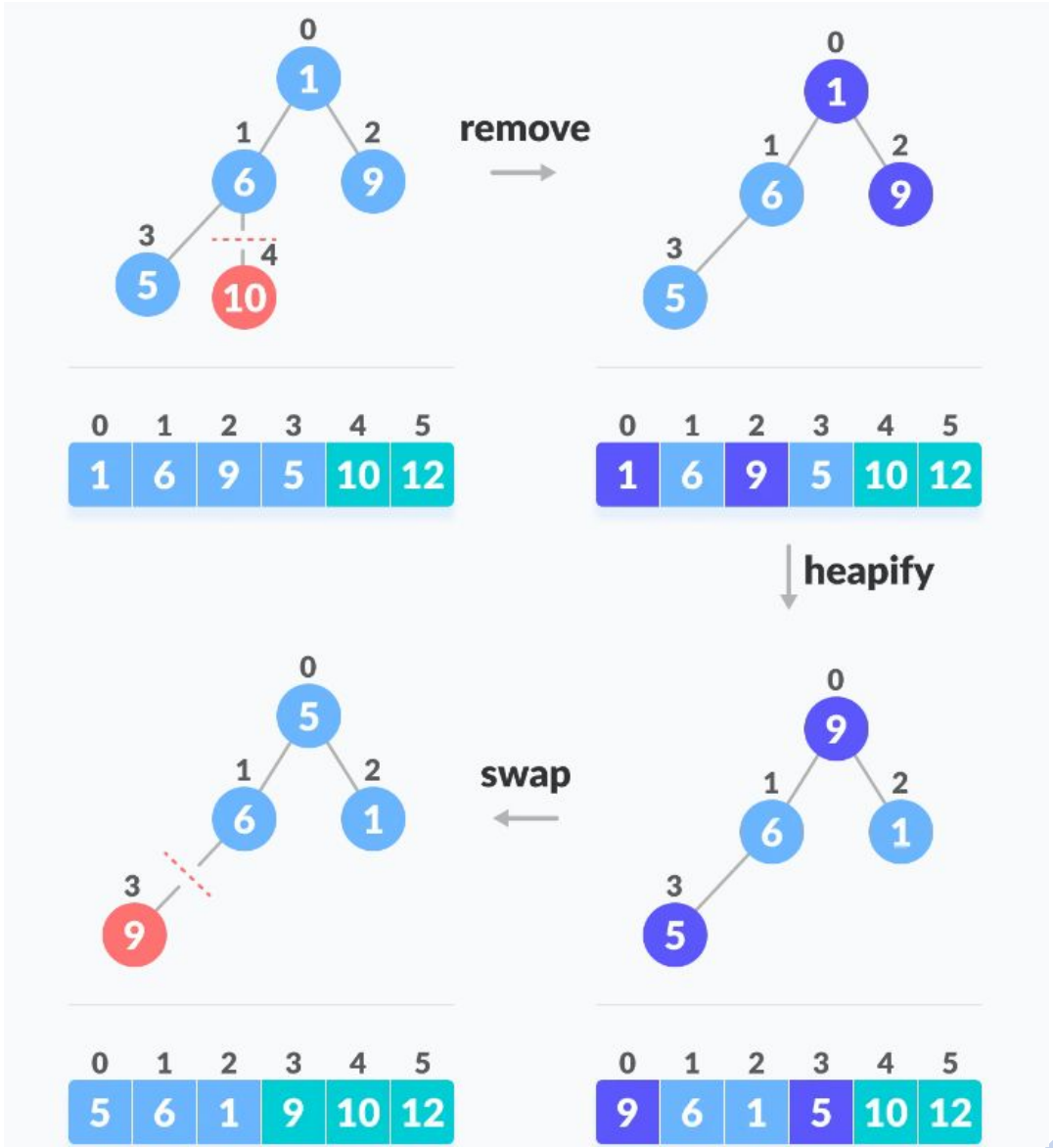
Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.

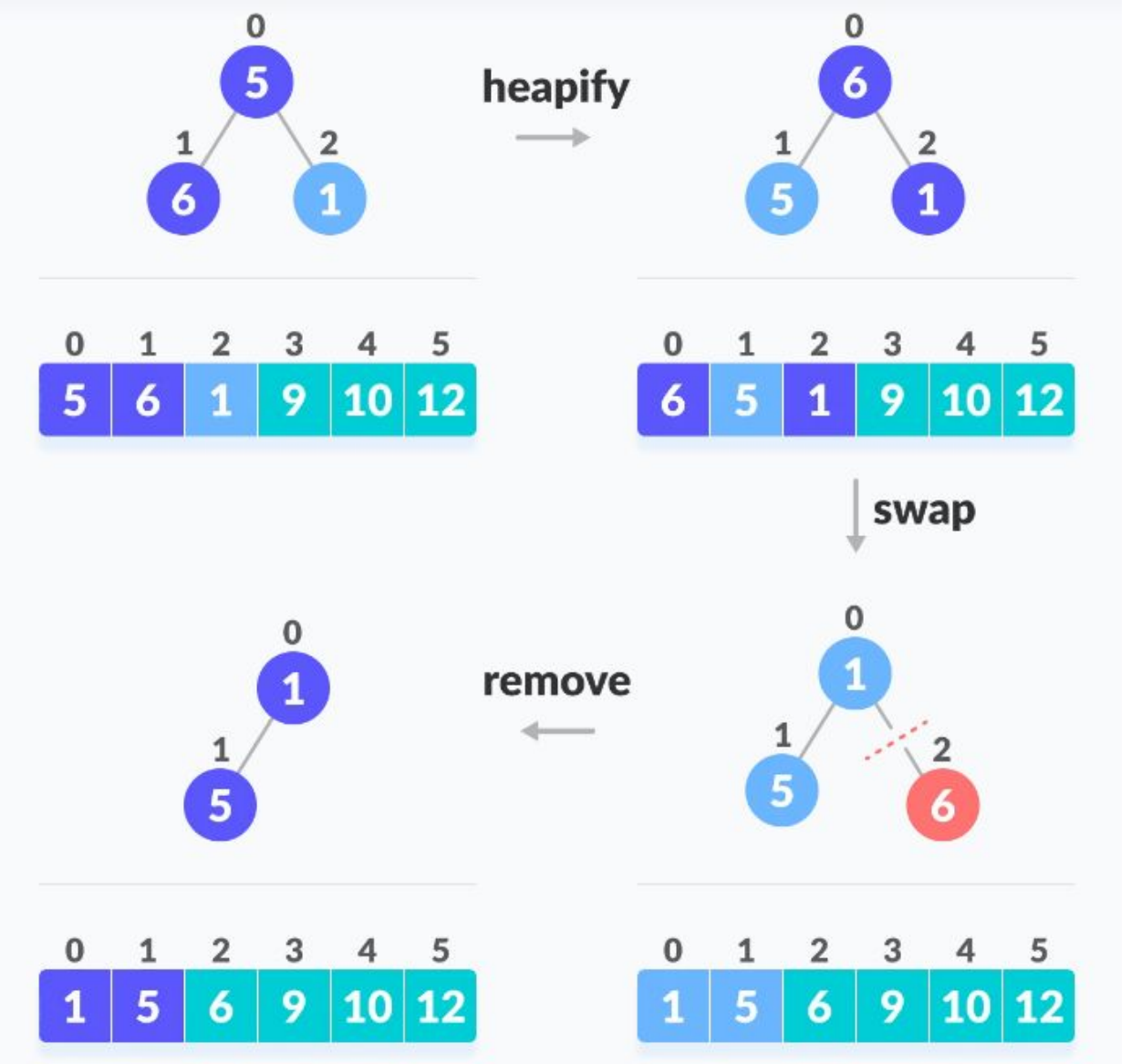
Swap: the root element and the last array element

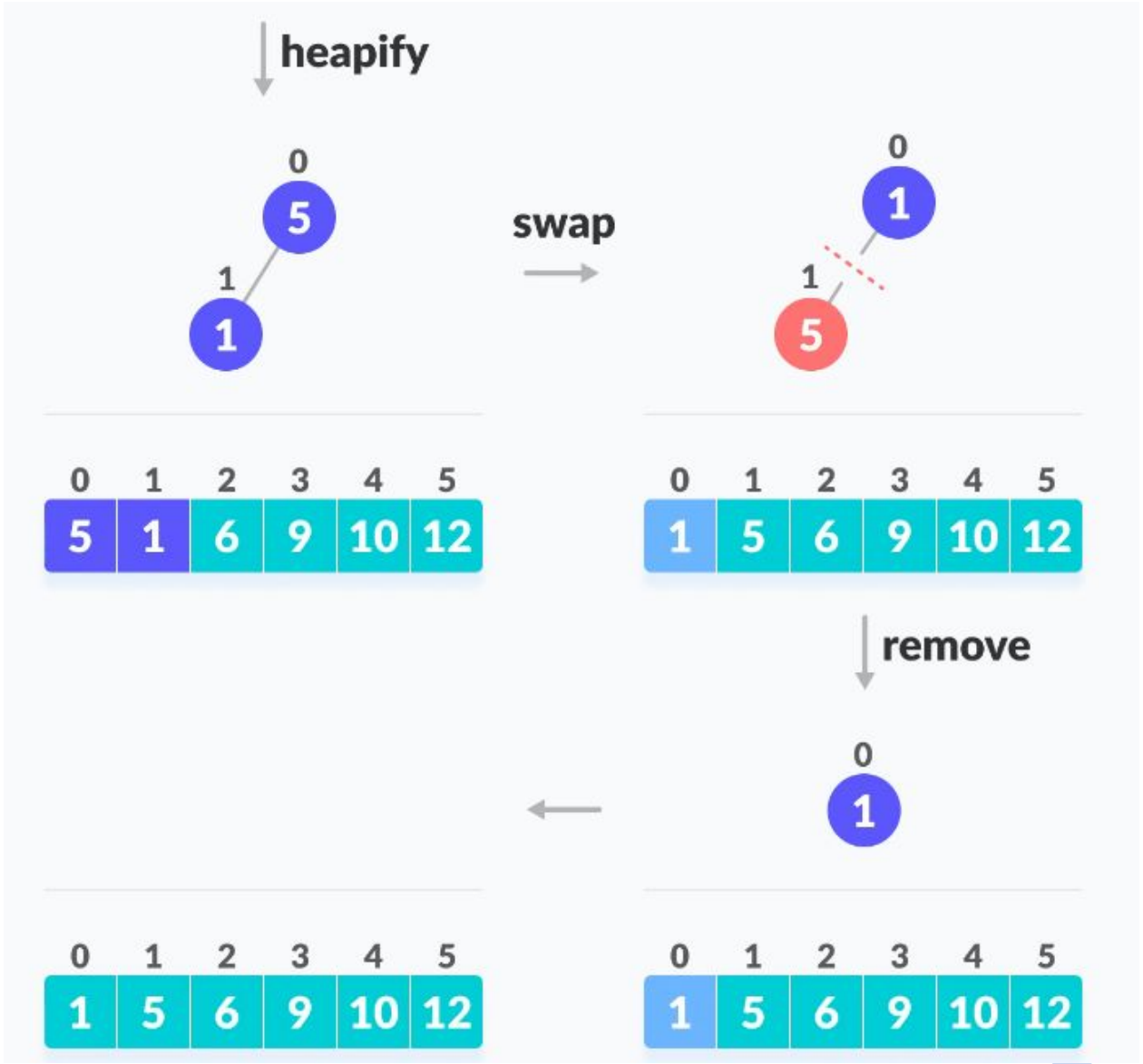
Remove: Reduce the size of the heap by 1.

Heapify: Heapify the root element again so that we have the highest element at root.


0
12
1
6
2
10
3
5
4
1
5
9
swap
0
9
1
6
2
10
3
5
4
1
5
12
0 1 2 3 4 5
12 6 10 5 1 9
0 1 2 3 4 5
9 6 10 5 1 12
remove
0
10
1
6
2
9
3
5
4
1
heapify
0
9
1
6
2
10
3
5
4
1
0 1 2 3 4 5
10 6 9 5 1 12
0 1 2 3 4 5
9 6 10 5 1 12
swap


0
1
1
6
2
9
3
5
4
10
remove
0
1
1
6
2
9
3
5
0 1 2 3 4 5
1 6 9 5 10 12
0 1 2 3 4 5
1 6 9 5 10 12
heapify
0
5
1
6
2
1
3
9
swap
0
9
1
6
2
1
3
5
0 1 2 3 4 5
5 6 1 9 10 12
0 1 2 3 4 5
9 6 1 5 10 12


0
5
1
2
6
1
heapify
0
6
1
2
5
1
0 1 2 3 4 5
5 6 1 9 10 12
0 1 2 3 4 5
6 5 1 9 10 12
swap
0
1
1
5
remove
0
1
1
2
5
6
0 1 2 3 4 5
1 5 6 9 10 12
0 1 2 3 4 5
1 5 6 9 10 12


heapify
0
5
1
1
swap
0
1
1
5
0 1 2 3 4 5
5 1 6 9 10 12
0 1 2 3 4 5
1 5 6 9 10 12
remove
0
1
0 1 2 3 4 5
1 5 6 9 10 12
0 1 2 3 4 5
1 5 6 9 10 12

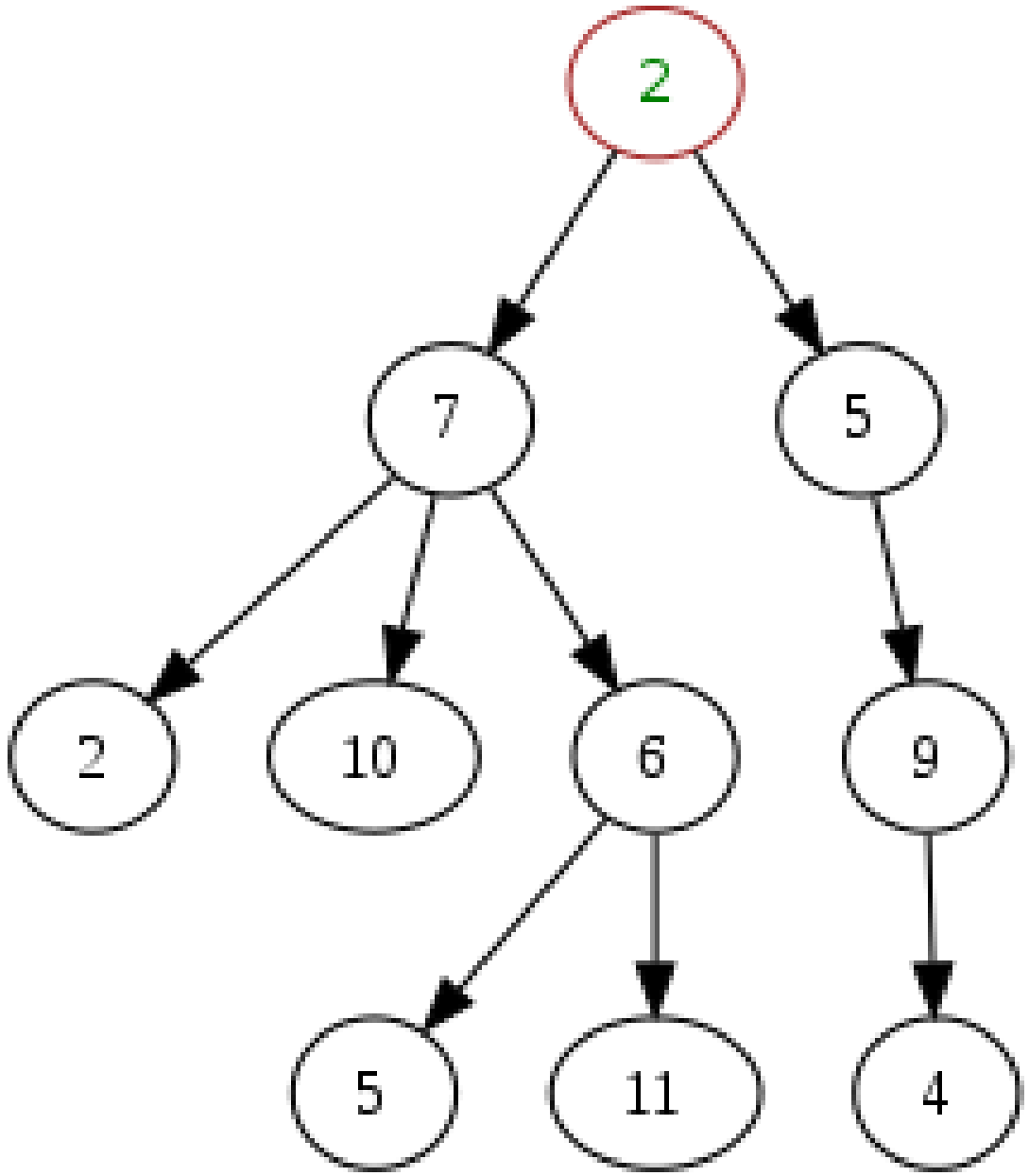
```
// Heap sort
for (int i = n - 1; i >= 0; i--) {
  swap(&arr[0], &arr[i]);

  // Heapify root element to get highest element at root again
  heapify(arr, i, 0);
}
```

Heap Sort Complexity

Time Complexity	
Best	O(nlog n)
Worst	O(nlog n)
Average	O(nlog n)
Space Complexity	O(1)

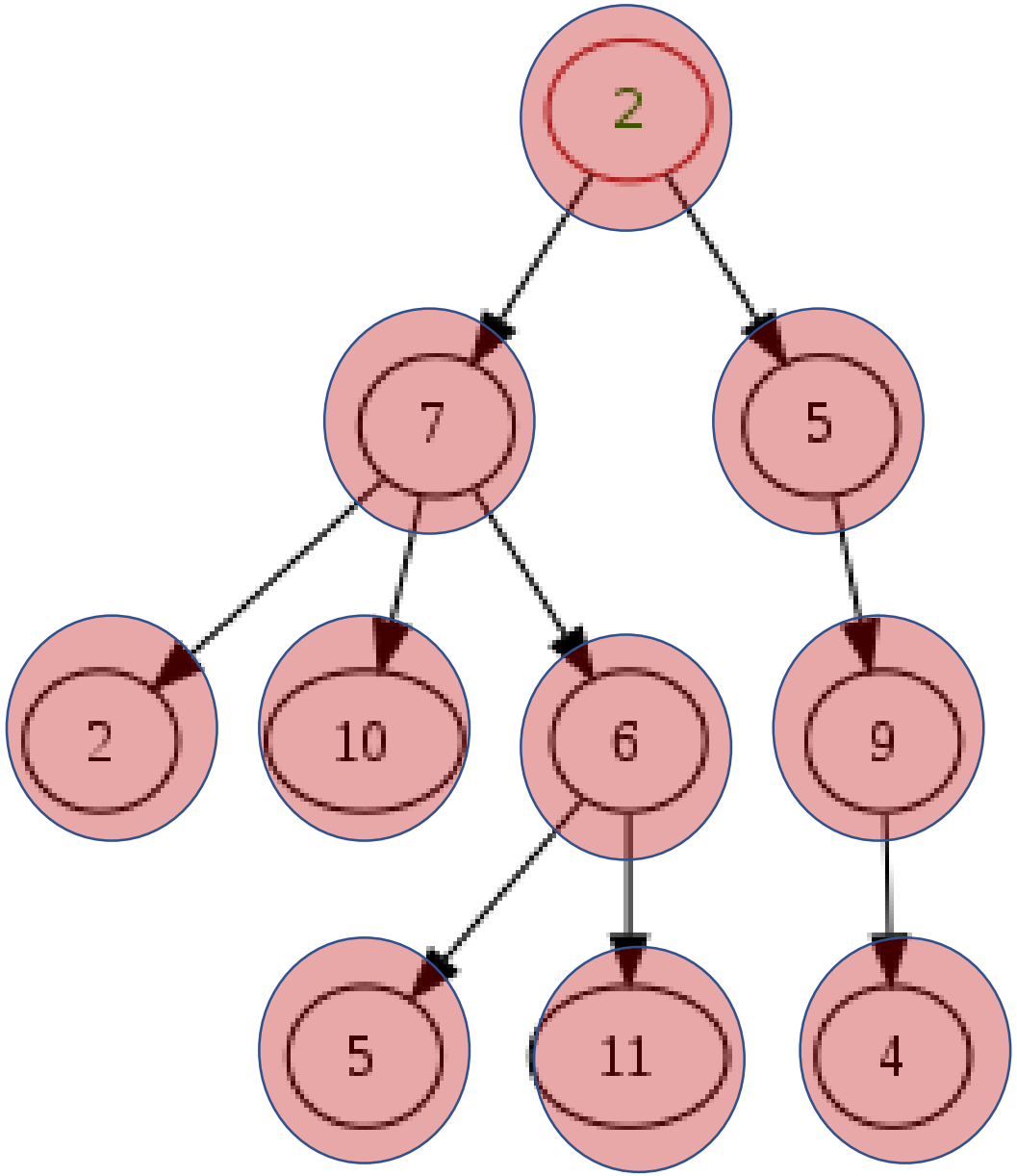
Trees

Introduction to trees



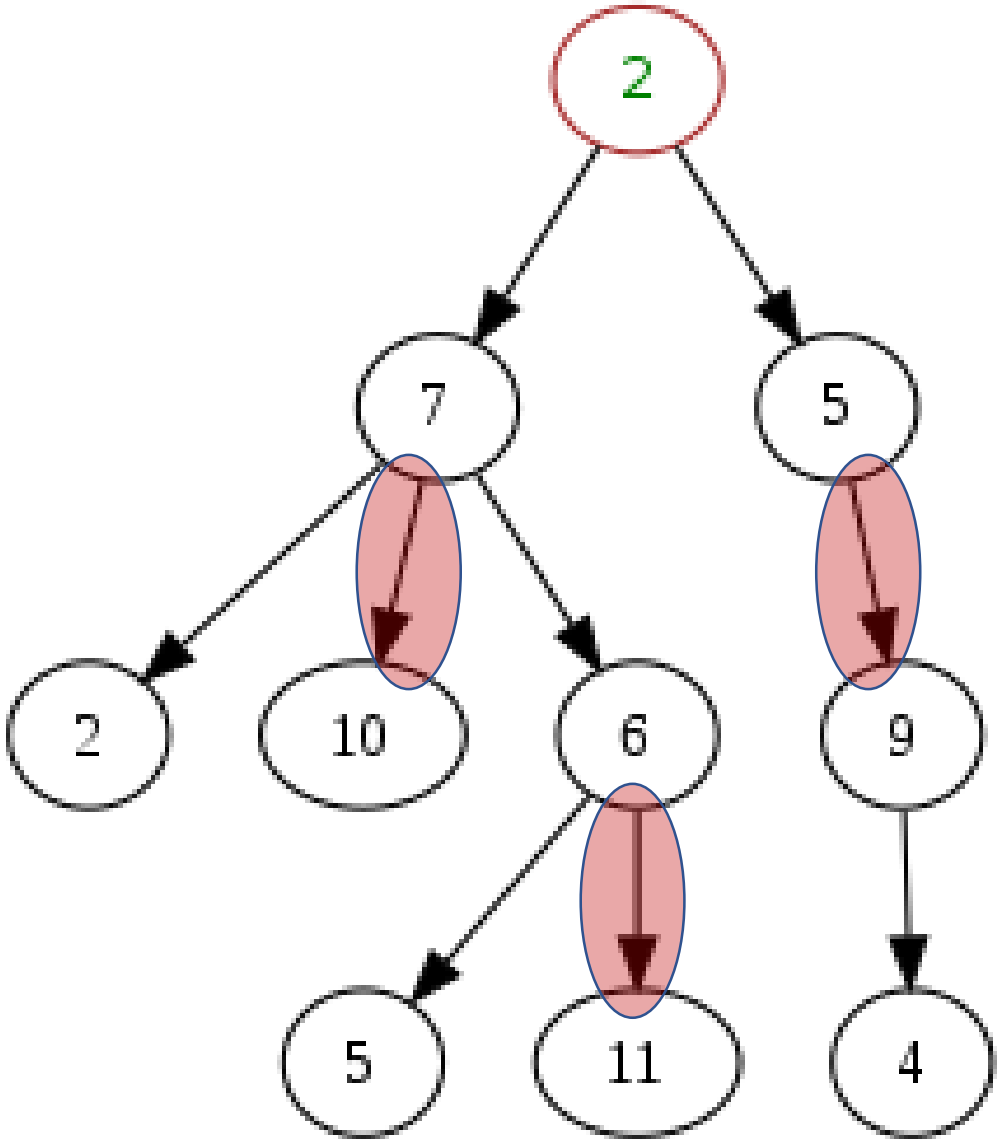
Terminology

- ***Node***
- Edge
- Root
- Path
- Children
- Parent
- Sibling
- Subtree
- Leaf Node



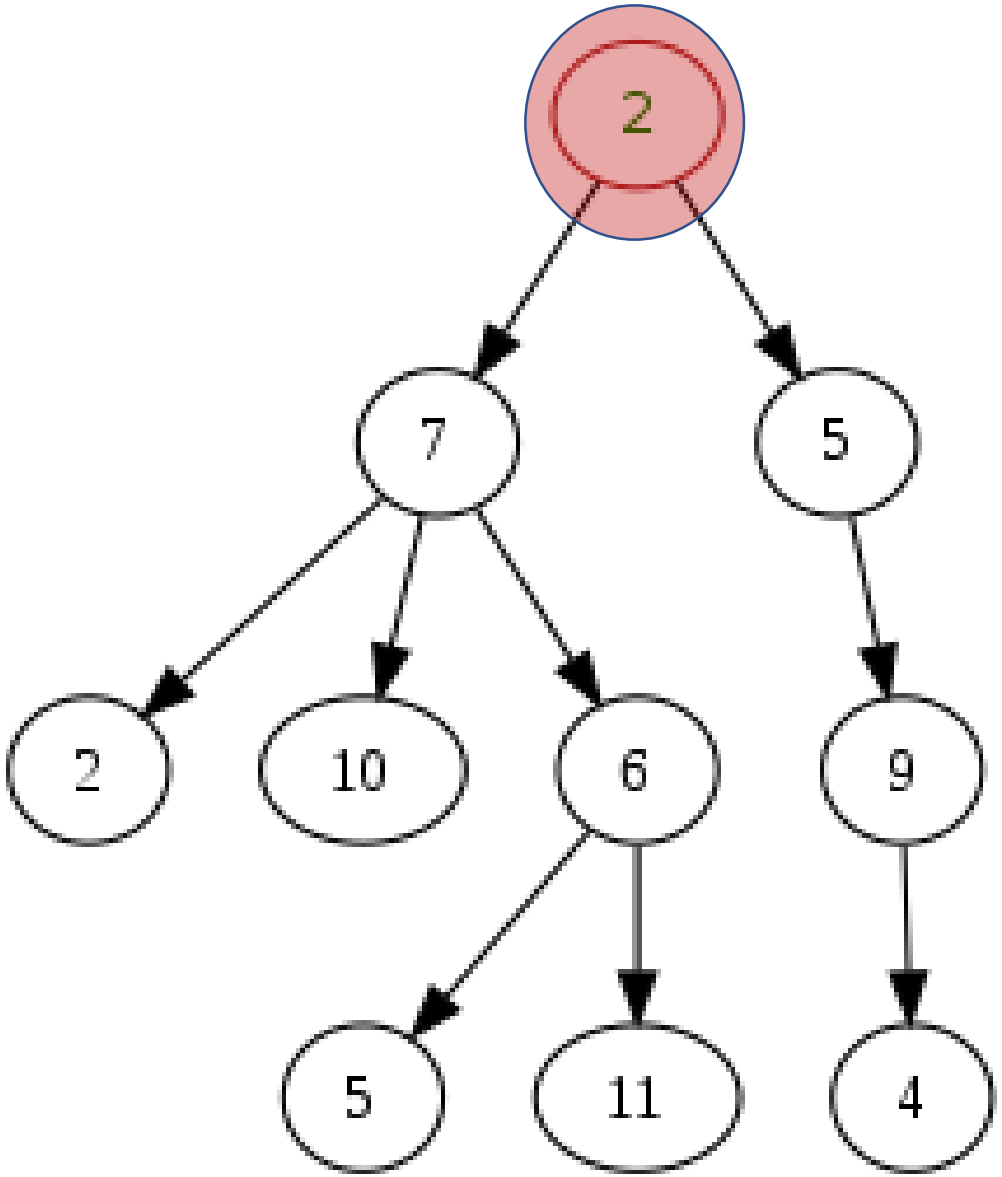
Terminology

- Node
- ***Edge***
- Root
- Path
- Children
- Parent
- Sibling
- Subtree
- Leaf Node



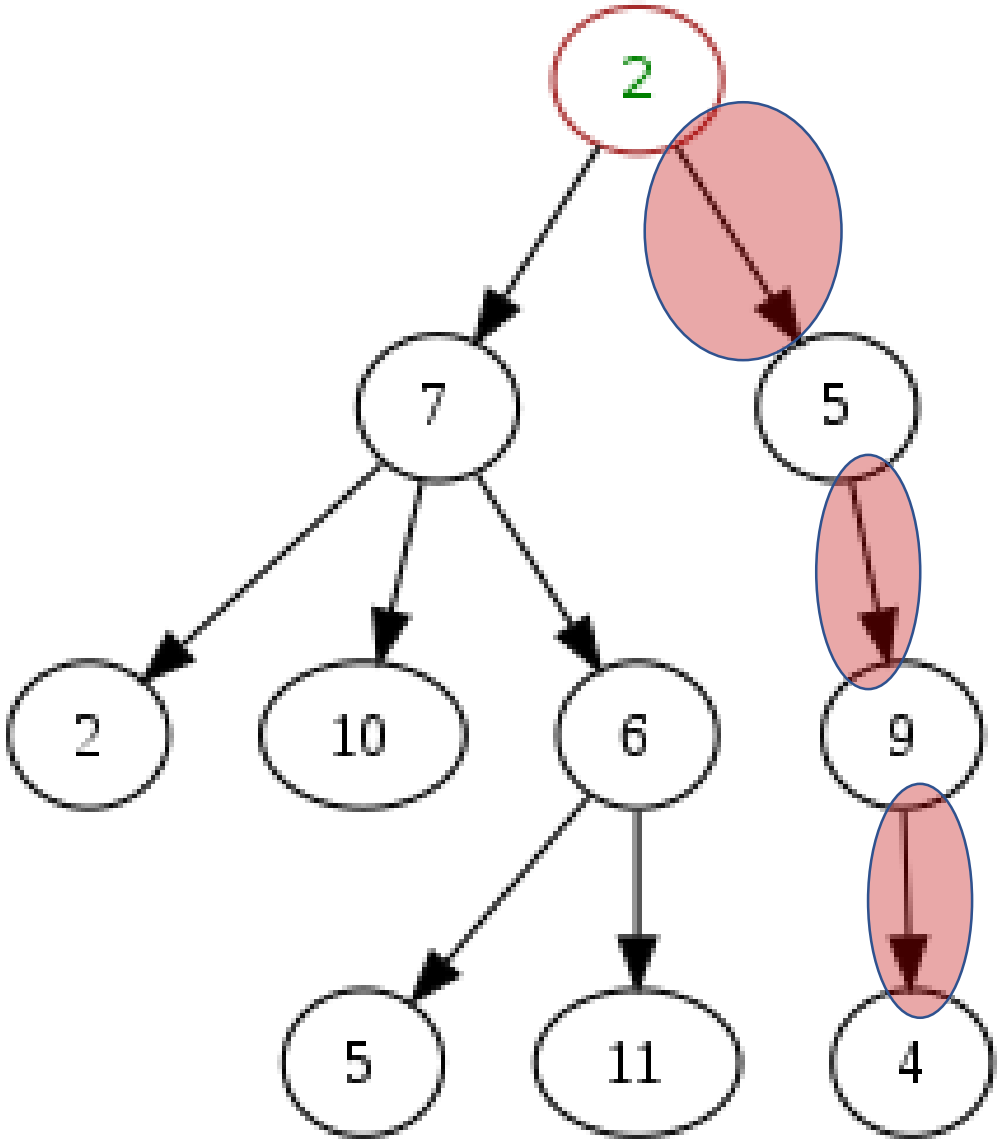
Terminology

- Node
- Edge
- ***Root***
- Path
- Children
- Parent
- Sibling
- Subtree
- Leaf Node



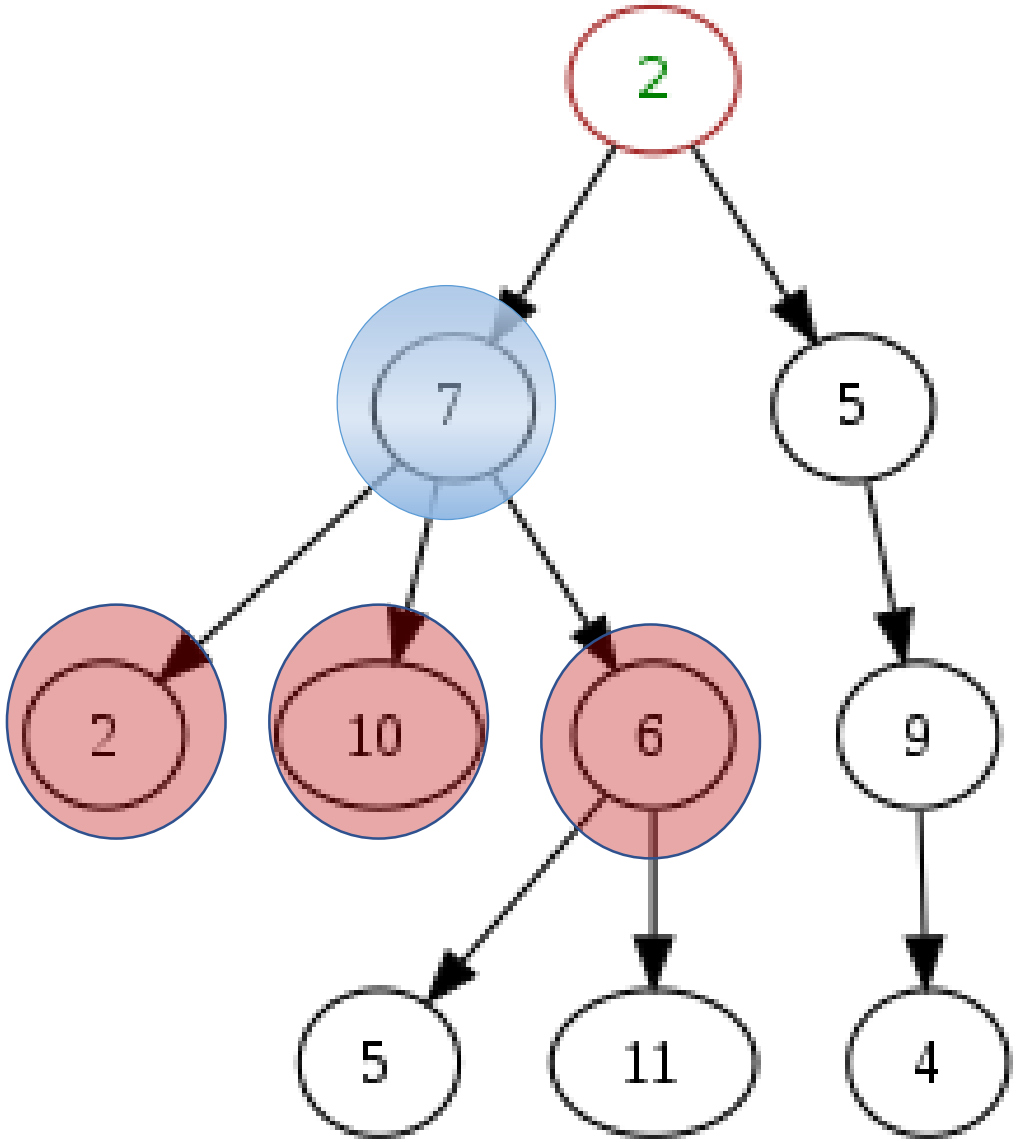
Terminology

- Node
- Edge
- Root
- ***Path***
- Children
- Parent
- Sibling
- Subtree
- Leaf Node



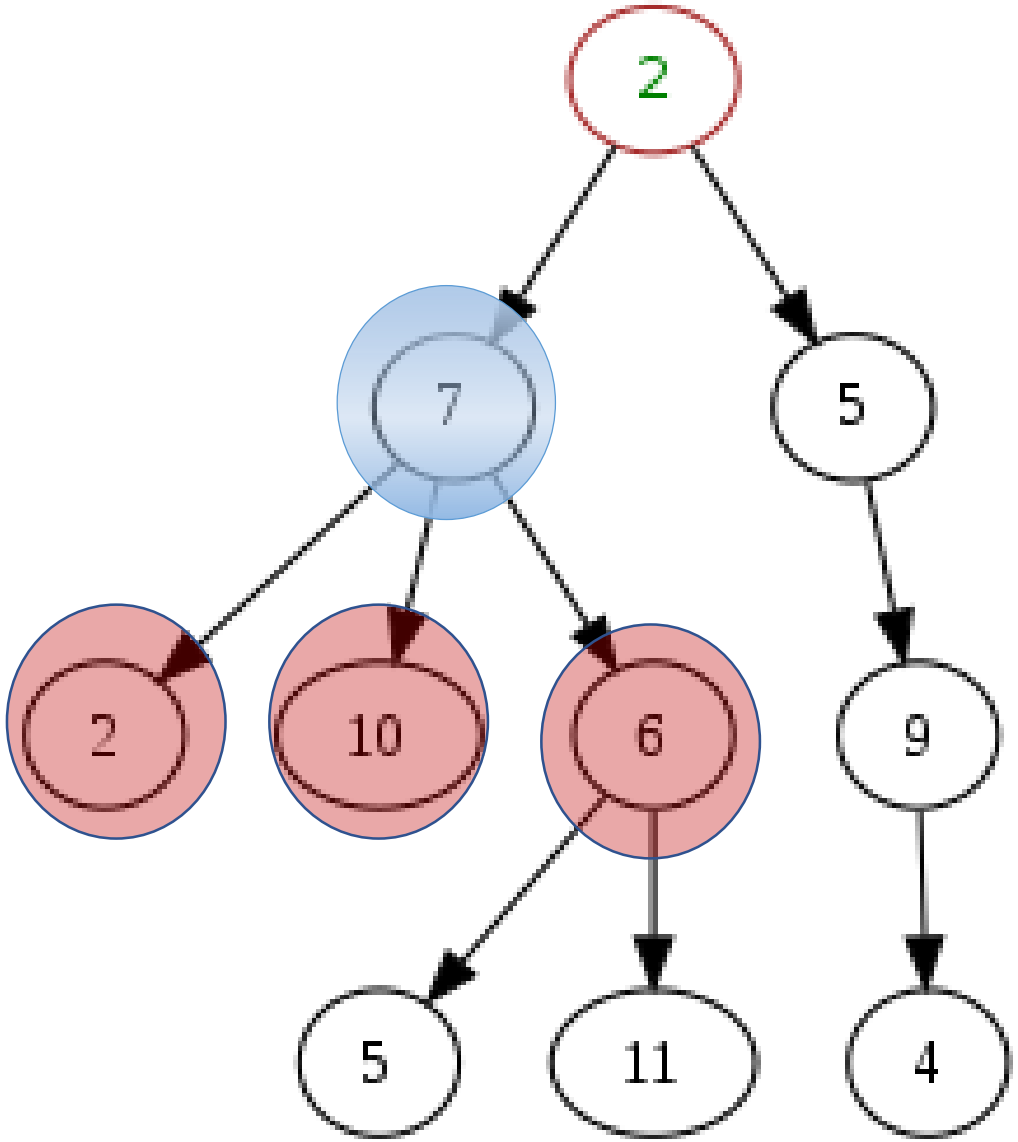
Terminology

- Node
- Edge
- Root
- Path
- ***Children***
- Parent
- Sibling
- Subtree
- Leaf Node



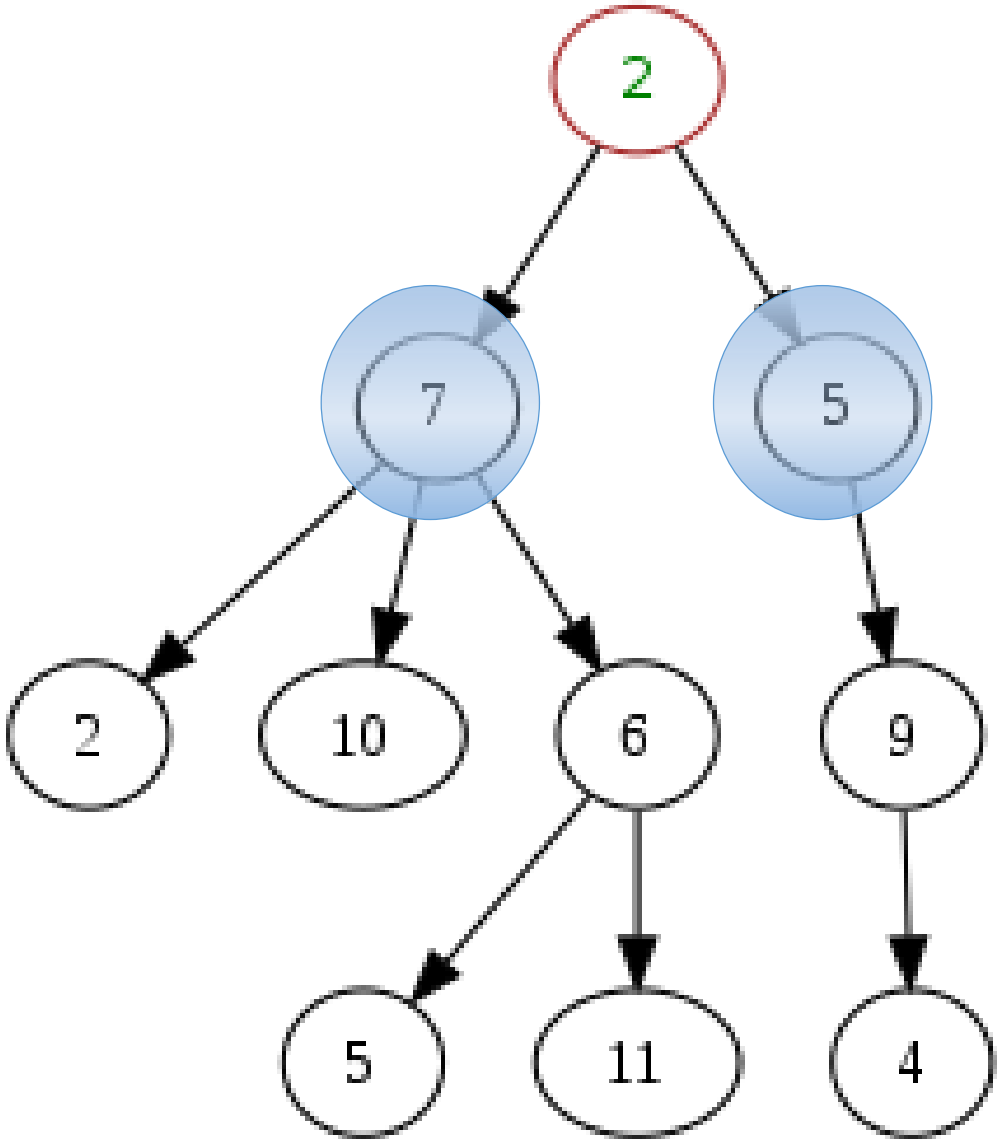
Terminology

- Node
- Edge
- Root
- Path
- Children
- ***Parent***
- Sibling
- Subtree
- Leaf Node



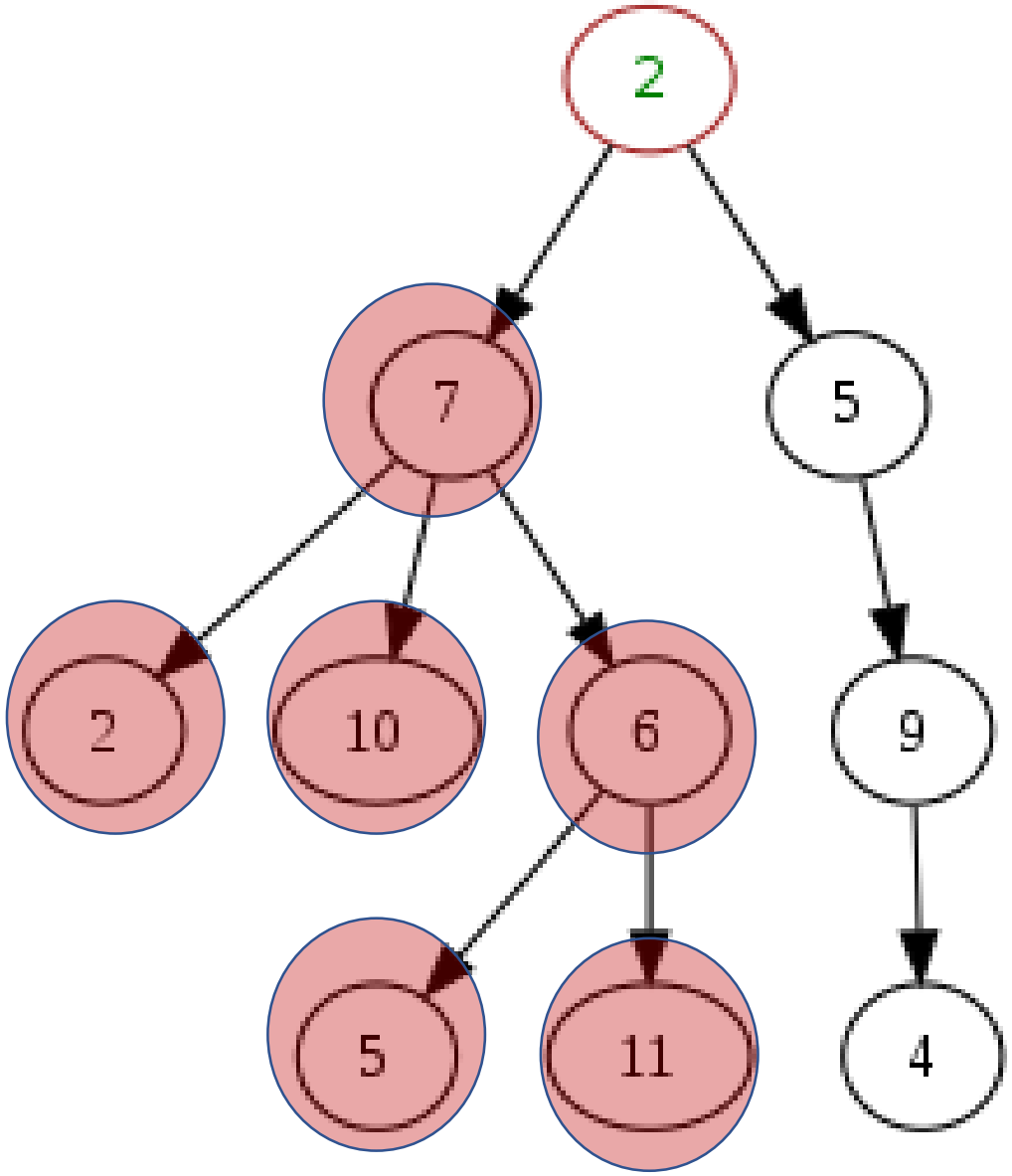
Terminology

- Node
- Edge
- Root
- Path
- Children
- Parent
- ***Sibling***
- Subtree
- Leaf Node



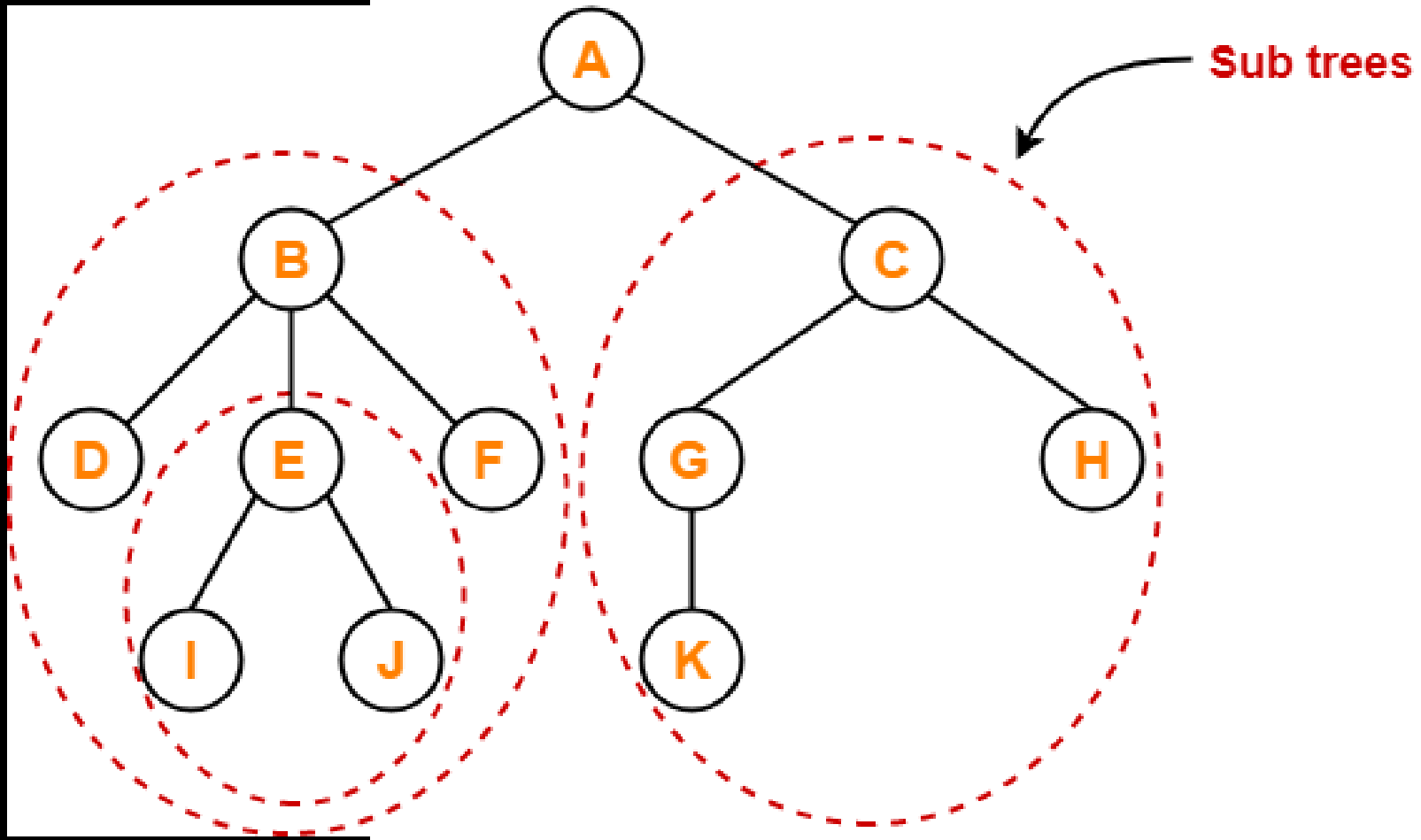
Terminology

- Node
- Edge
- Root
- Path
- Children
- Parent
- Sibling
- ***Subtree***
- Leaf Node



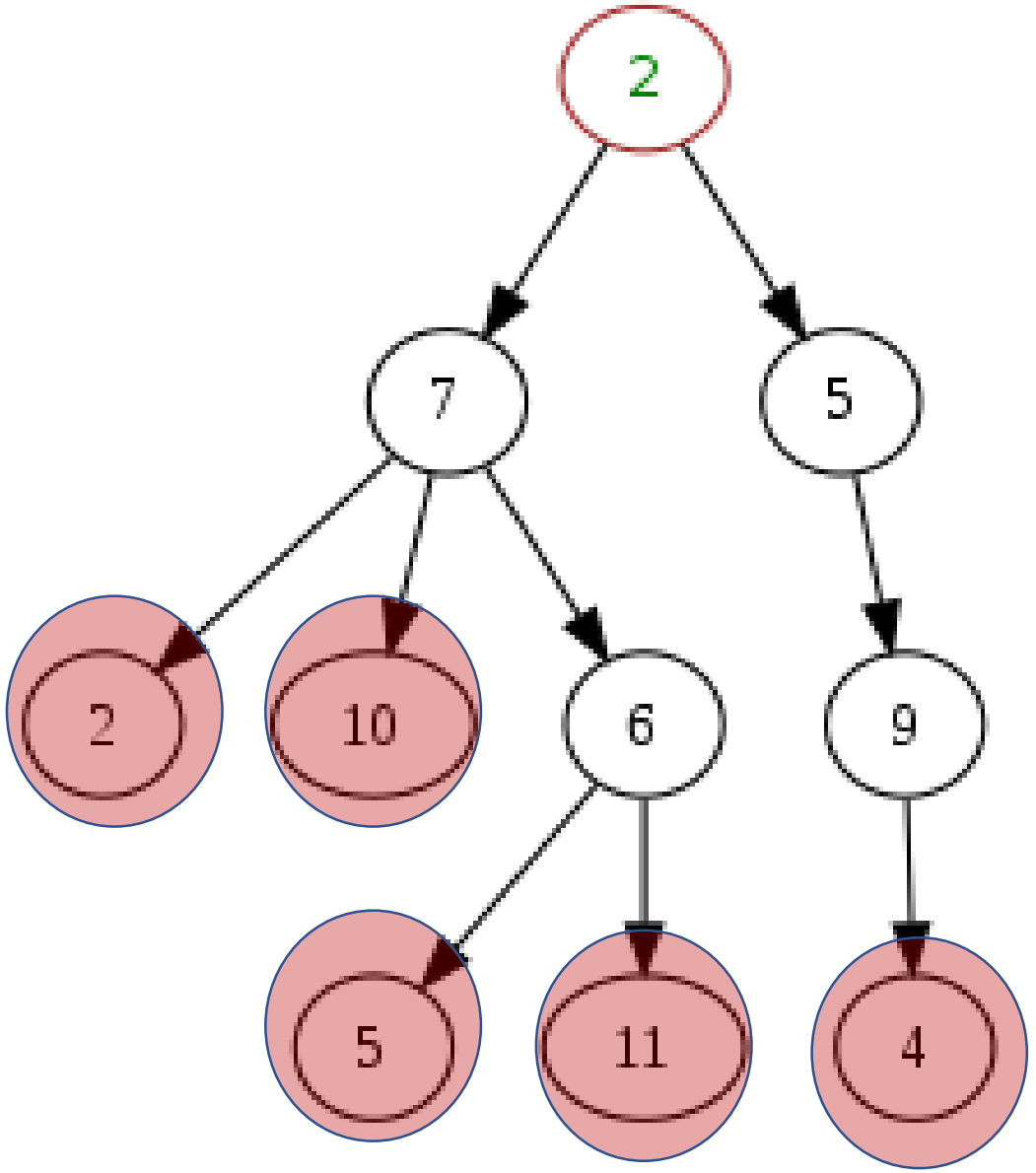
Terminology

- Node
- Edge
- Root
- Path
- Children
- Parent
- Sibling
- ***Subtree***
- Leaf Node

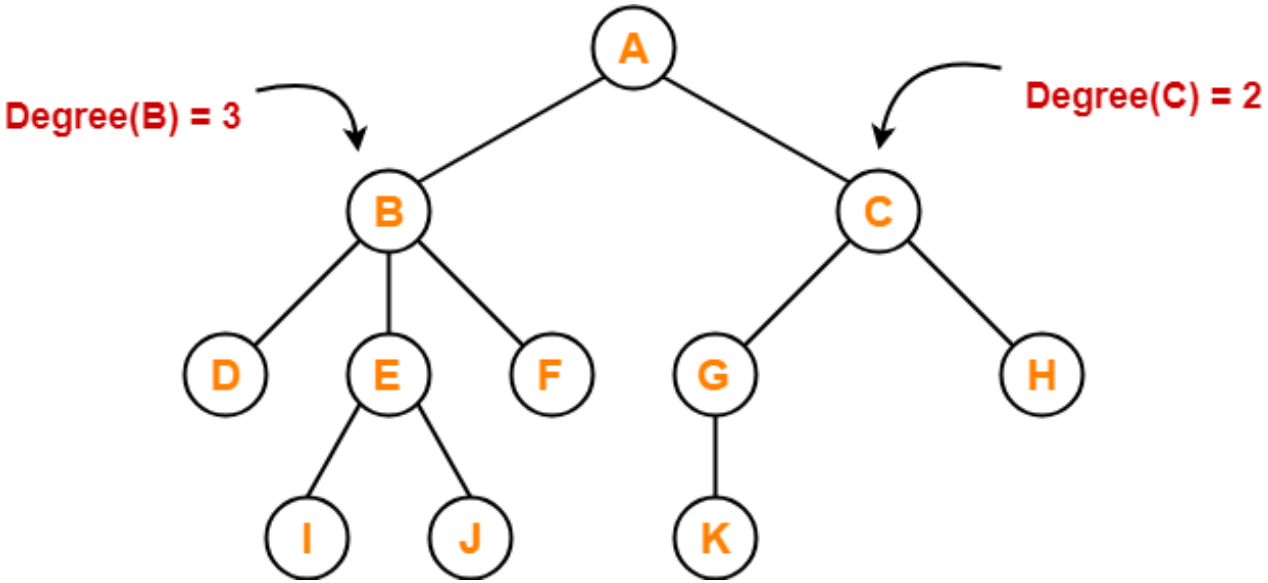


Terminology

- Node
- Edge
- Root
- Path
- Children
- Parent
- Sibling
- Subtree
- ***Leaf Node***

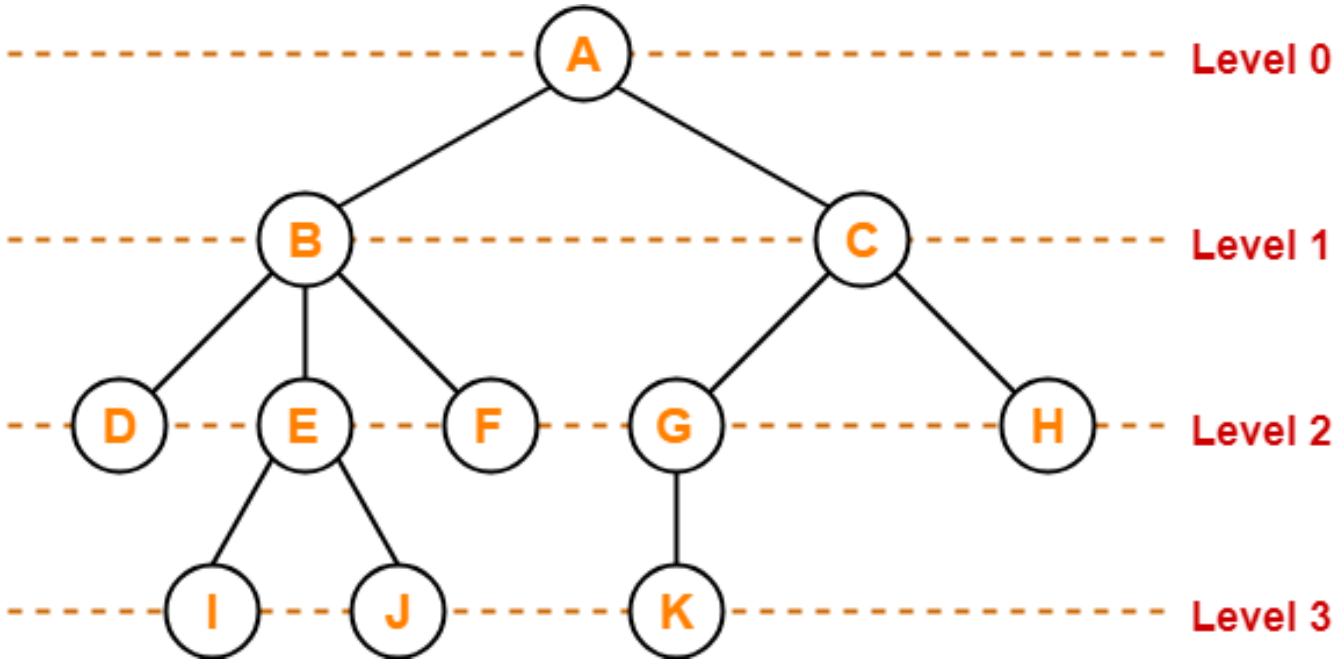


Degree



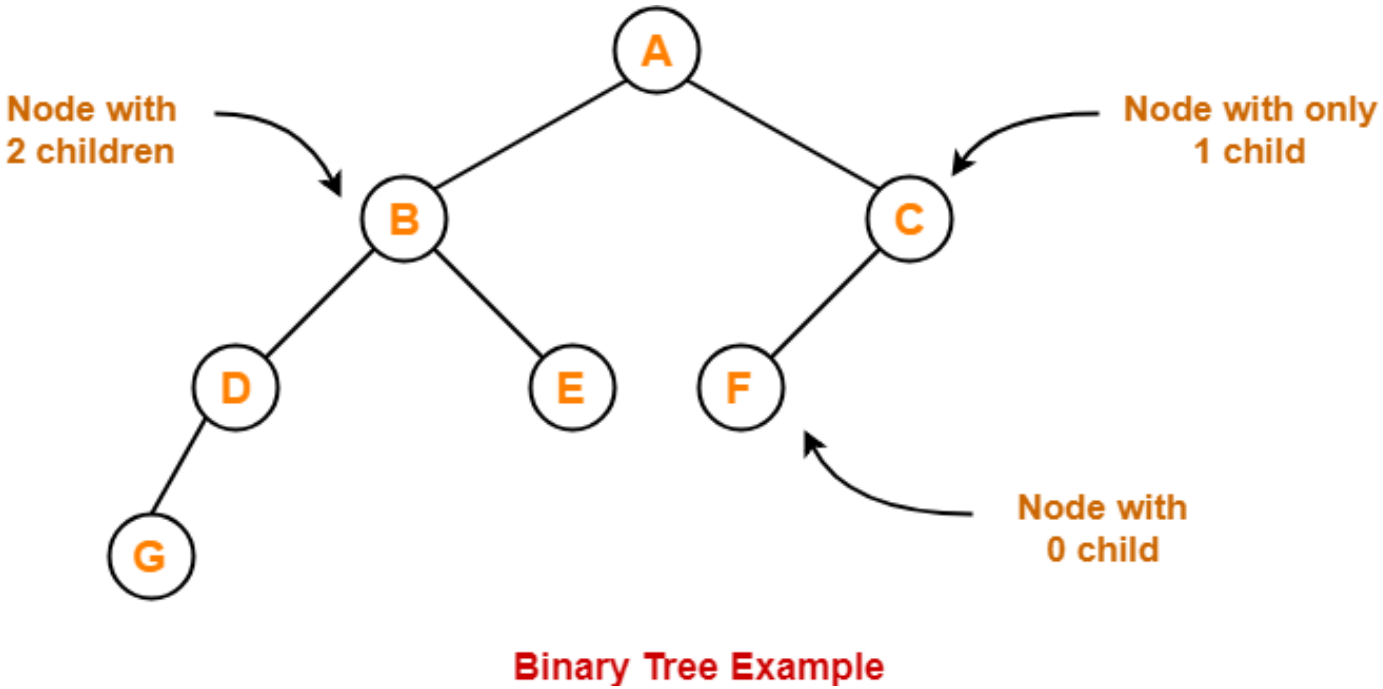
- **Degree of a node** is the total number of children of that node.
- **Degree of a tree** is the highest degree of a node among all the nodes in the tree.

Level



- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.

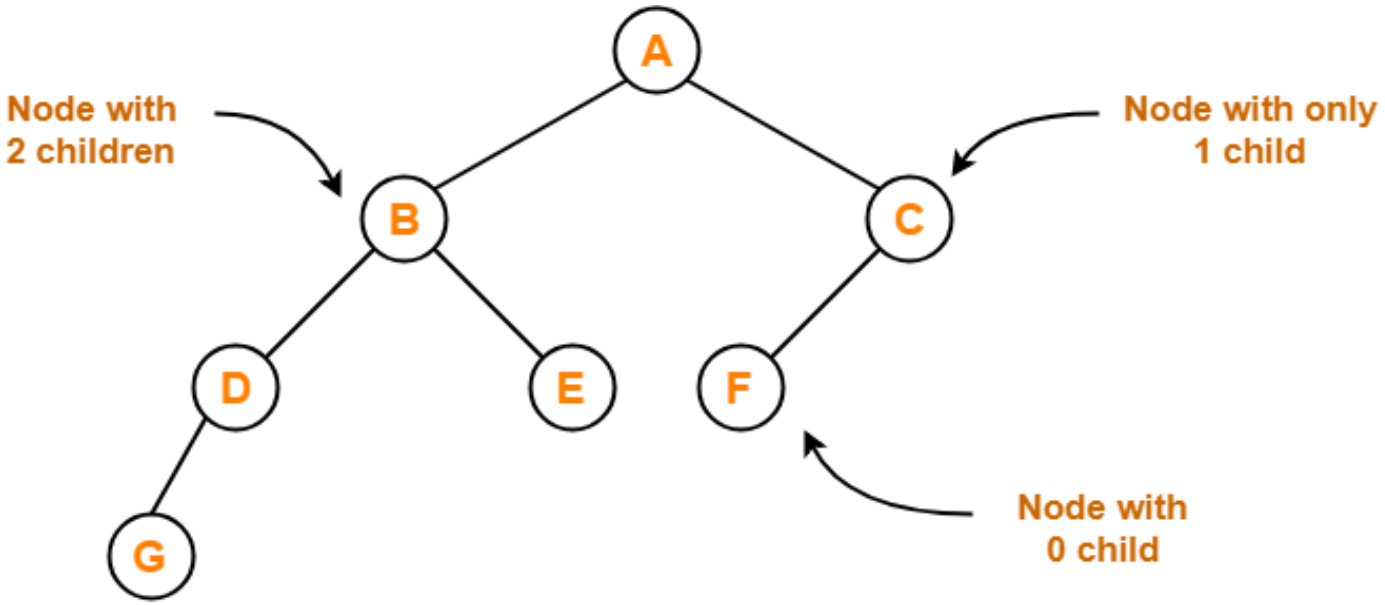
Binary Tree



Binary Tree Example

- Binary tree is a special tree data structure in which each node can have at most 2 children.
- Thus, in a binary tree, each node has either 0 child or 1 child or 2 children.

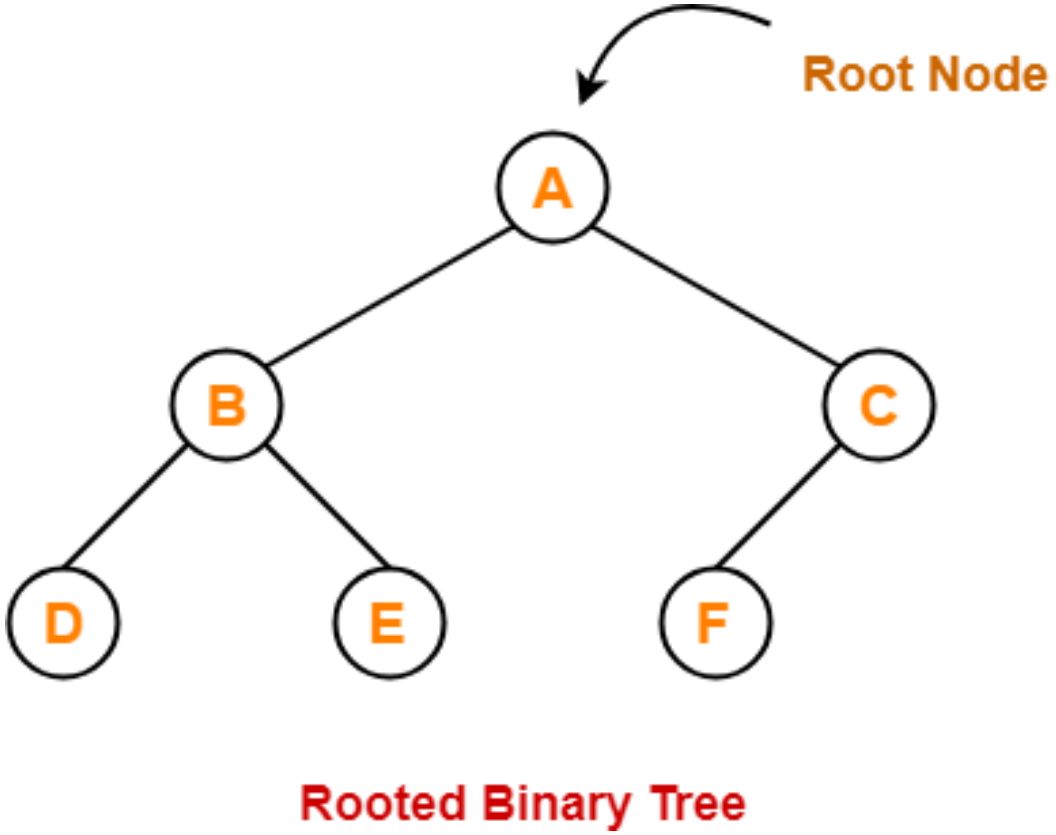
Types of Binary Trees



Binary Tree Example

- Rooted Binary Tree
- Full / Strictly Binary Tree
- Complete / Perfect Binary Tree
- Almost Complete Binary Tree
- Skewed Binary Tree

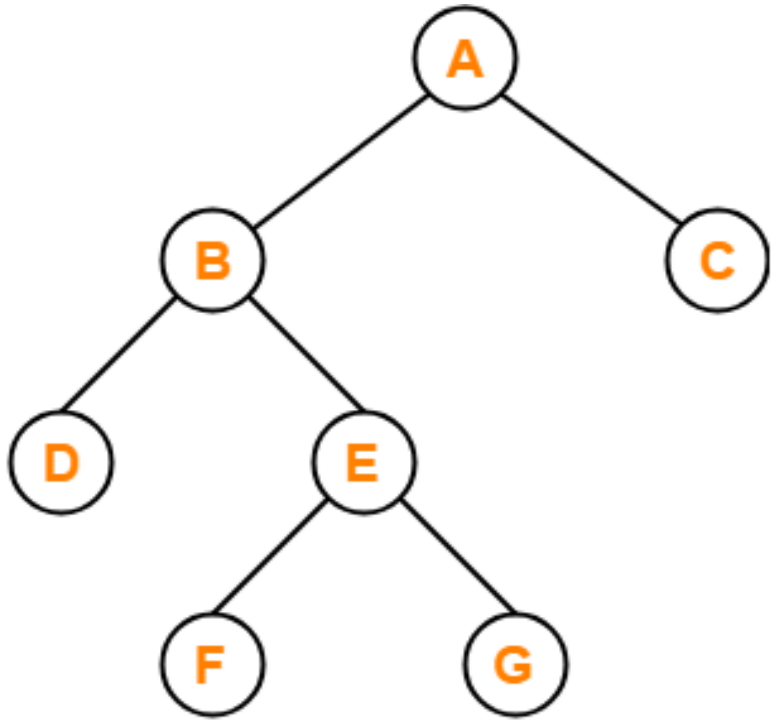
Types of Binary Trees



Rooted Binary Tree

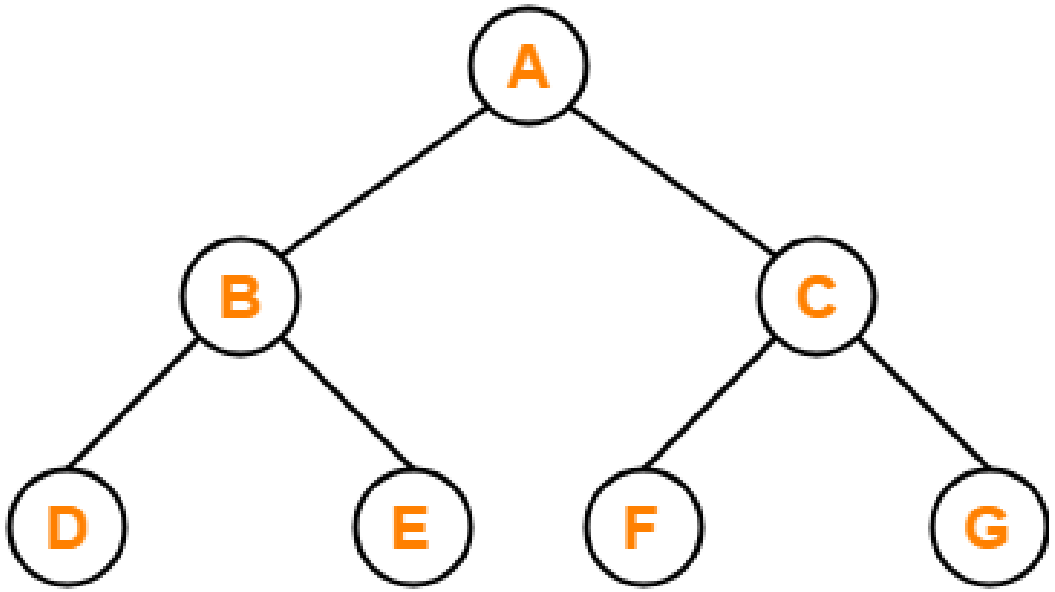
- Rooted Binary Tree
  - *It has a root node.*
  - *Each node has at most 2 children.*

Types of Binary Trees



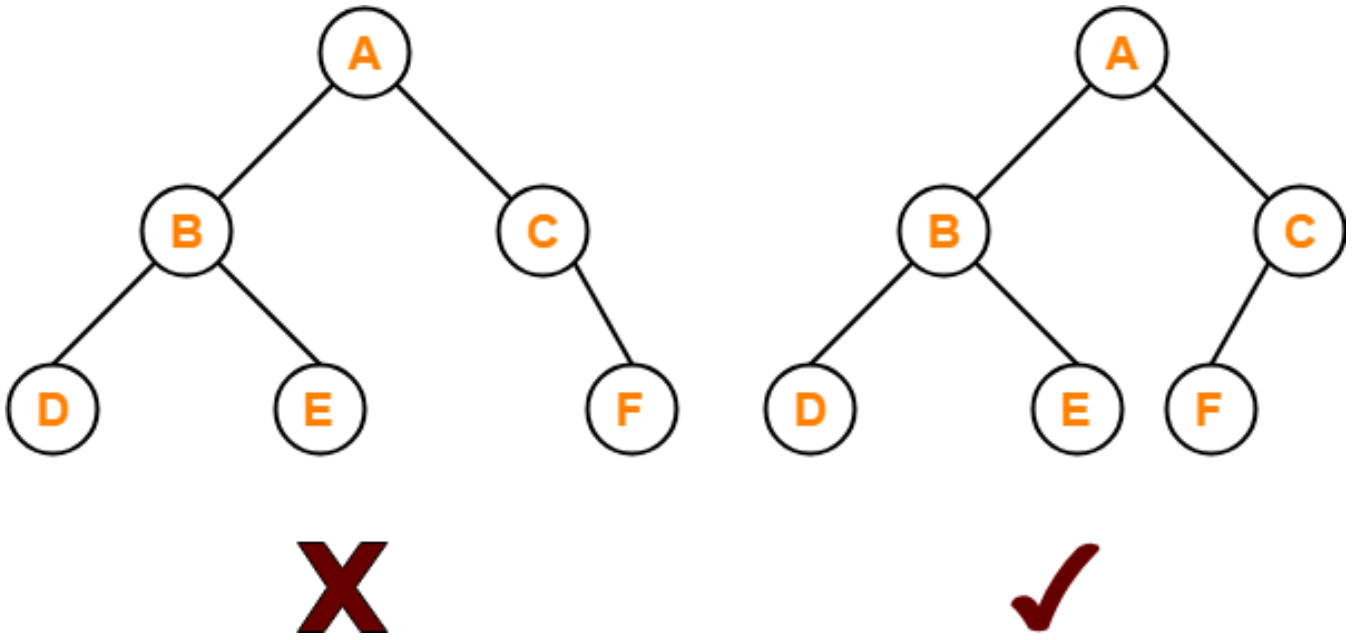
- **Full / Strictly Binary Tree**
  - A binary tree in which every node has either 0 or 2 children is called as a **Full binary tree**.

Types of Binary Trees



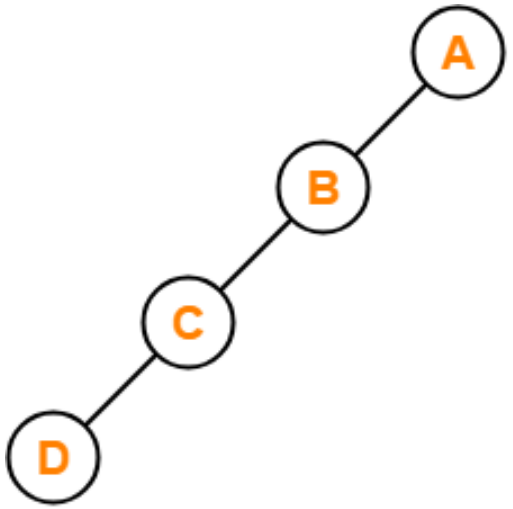
- **Complete / Perfect Binary Tree**
  - Every internal node has exactly 2 children.
  - All the leaf nodes are at the same level.

Types of Binary Trees

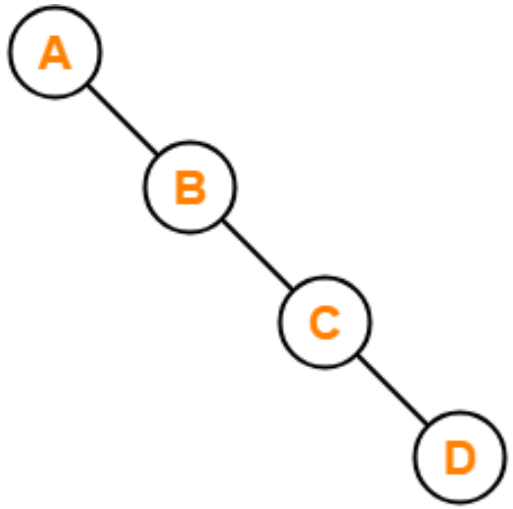


- **Almost Complete Binary Tree**
  - All the levels are completely filled except possibly the last level.
  - The last level must be strictly filled from left to right.

Types of Binary Trees



Left Skewed Binary Tree



Right Skewed Binary Tree

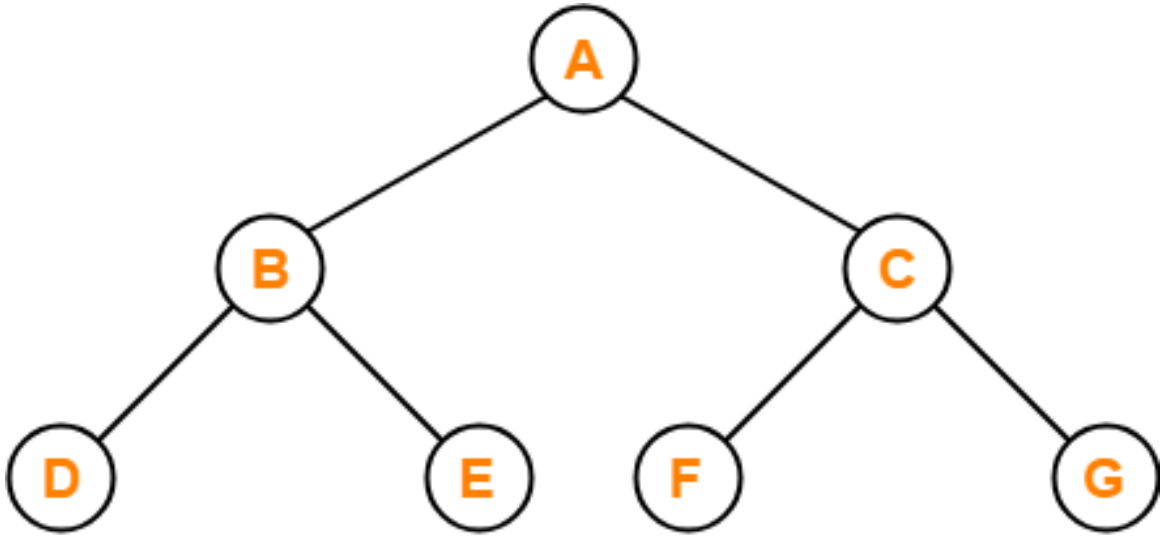
- **Skewed Binary Tree**
  - All the nodes except one node has one and only one child.
  - The remaining node has no child.

Tree Traversal

Tree Traversal Techniques

- **Depth First Traversal**
  - **Preorder Traversal**
  - **Inorder Traversal**
  - **Postorder Traversal**
- **Breadth First Traversal**

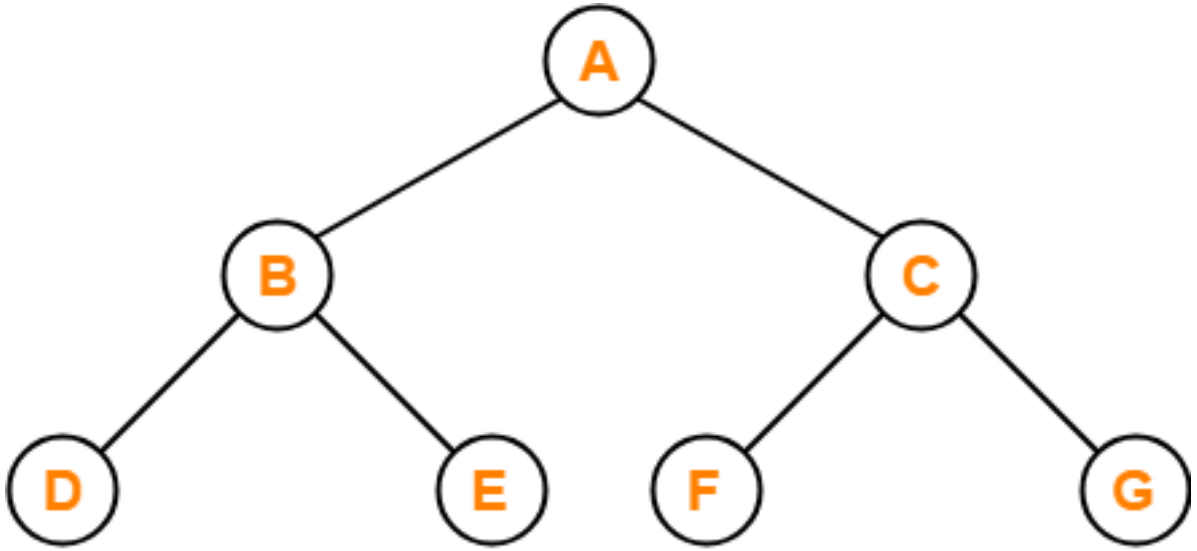
Preorder Traversal



Preorder Traversal : A , B , D , E , C , F , G

- Visit the root
- Traverse the left sub tree
- Traverse the right sub tree

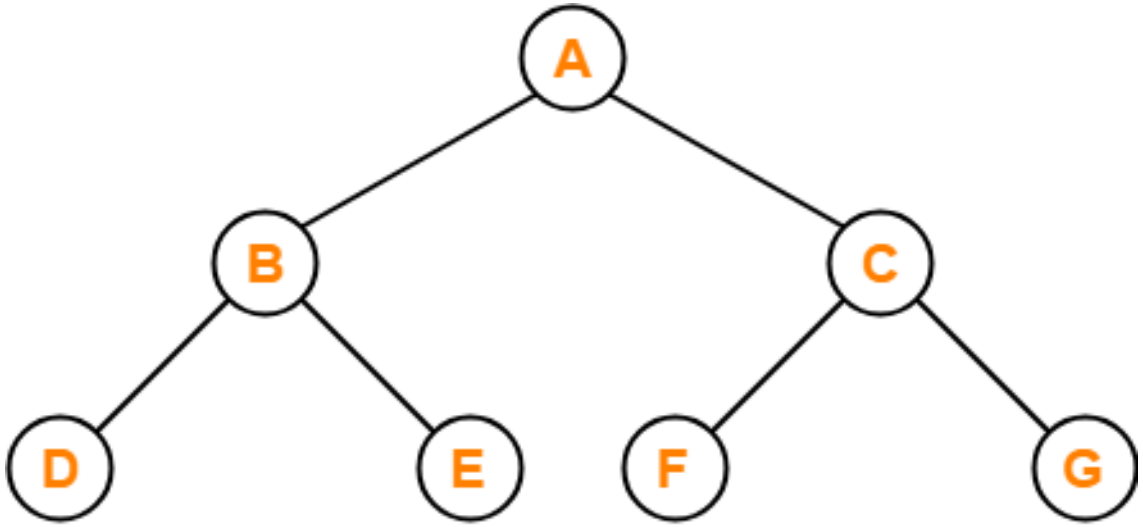
Inorder Traversal



Inorder Traversal : D , B , E , A , F , C , G

- Traverse the left sub tree
- Visit the root
- Traverse the right sub tree

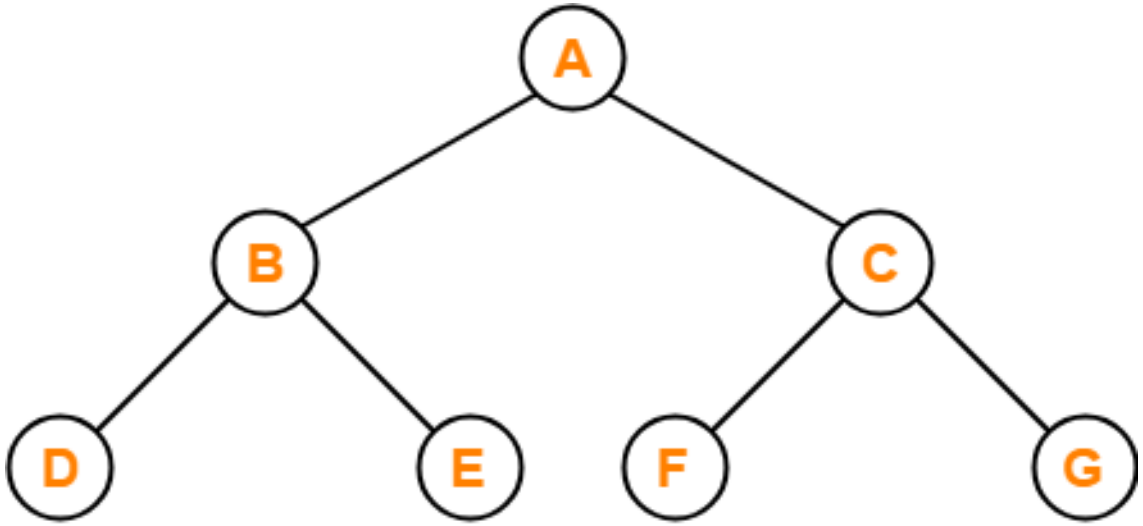
Postorder Traversal



Postorder Traversal : D , E , B , F , G , C , A

- Traverse the left sub tree
- Traverse the right sub tree
- Visit the root

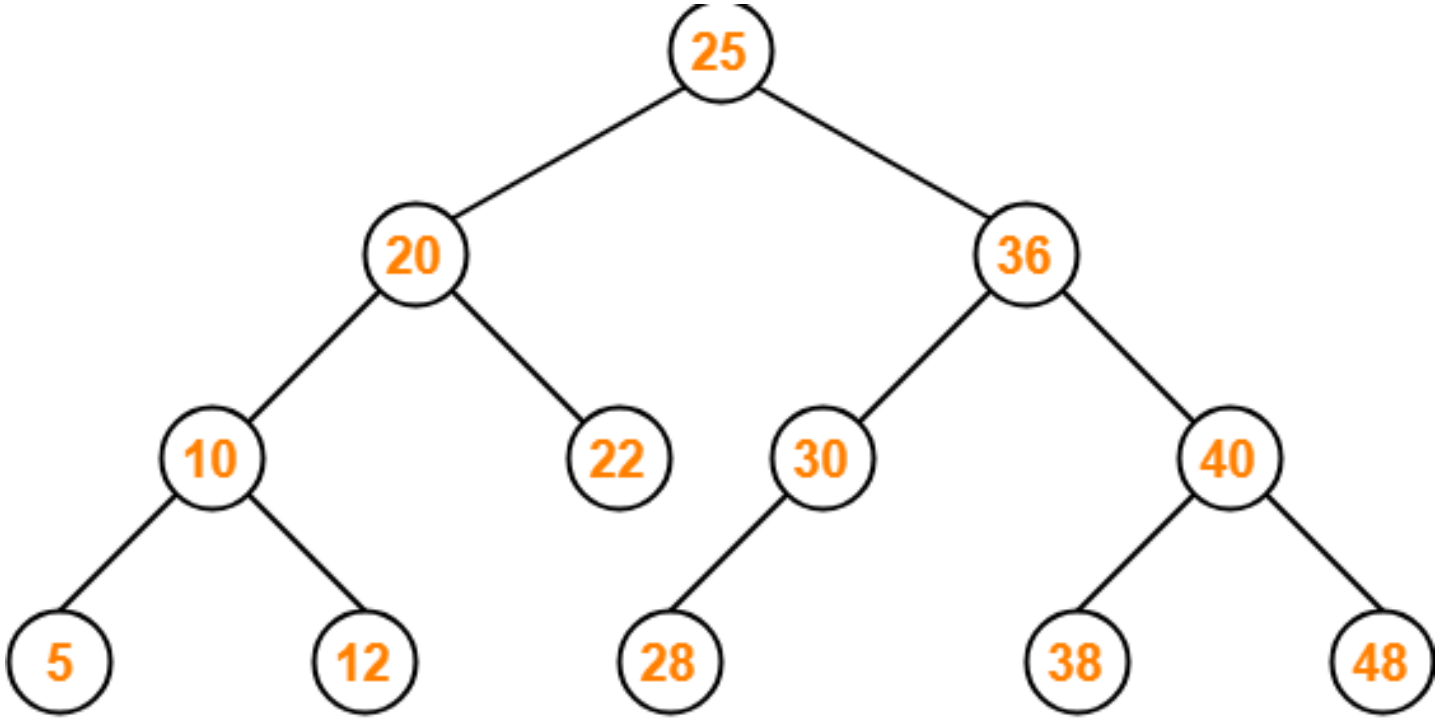
Breadth First Traversal



Level Order Traversal : A , B , C , D , E , F , G

- Breadth First Traversal of a tree prints all the nodes of a tree level by level.
- Breadth First Traversal is also called as **Level Order Traversal**

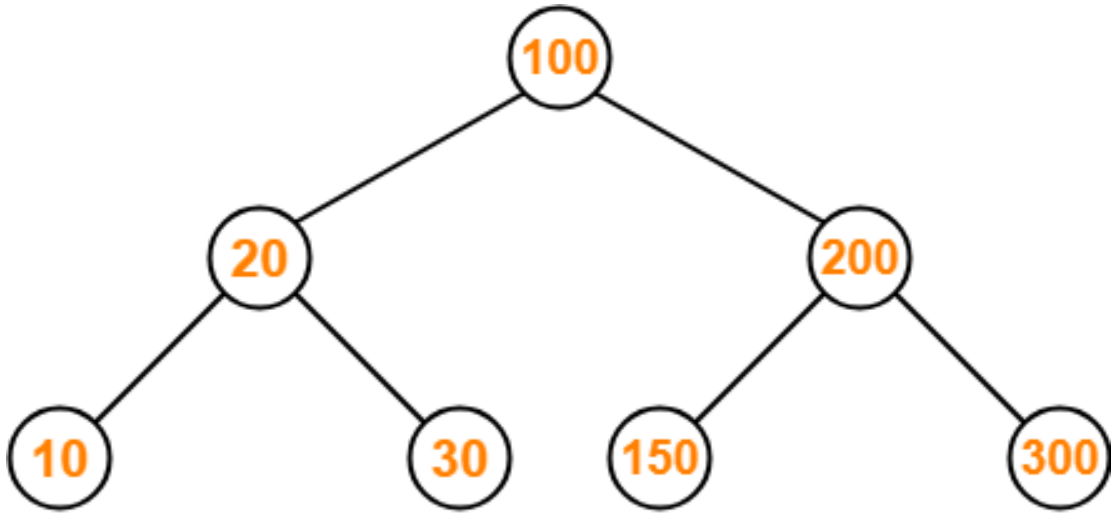
Binary Search Tree



Binary Search Tree

- Smaller values in its left sub tree
- Larger values in its right sub tree

BST Traversal



Binary Search Tree

- **Preorder Traversal-**

  100 , 20 , 10 , 30 , 200 , 150 , 300

- **Inorder Traversal-**

  10 , 20 , 30 , 100 , 150 , 200 , 300

- **Postorder Traversal-**

  10 , 30 , 20 , 150 , 300 , 200 , 100

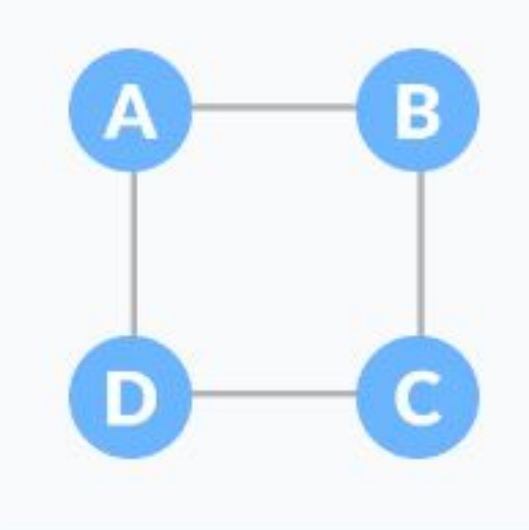
Spanning Tree

Spanning tree

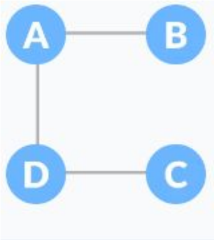
A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges.

The edges may or may not have weights assigned to them.

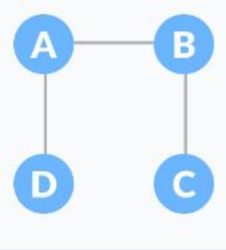
Example of a Spanning Tree



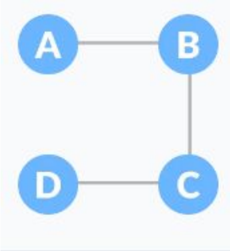
Normal graph



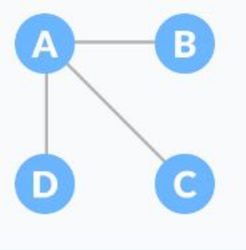
A spanning tree



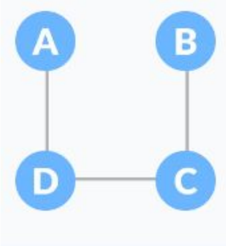
A spanning tree



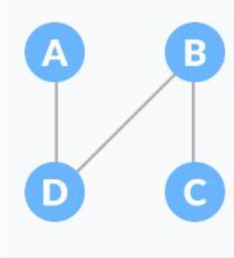
A spanning tree



A spanning tree



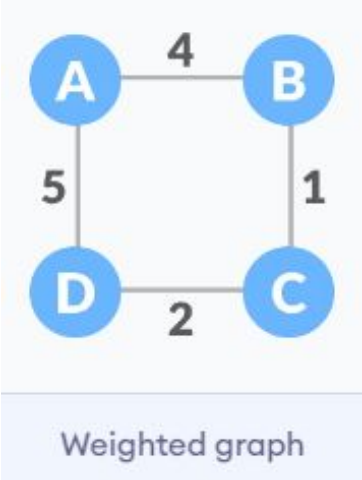
A spanning tree

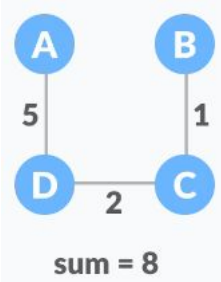


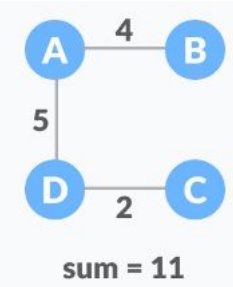
A spanning tree

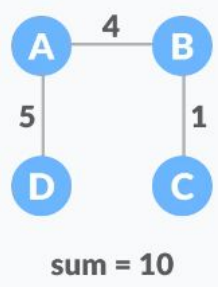
Minimum Spanning Tree

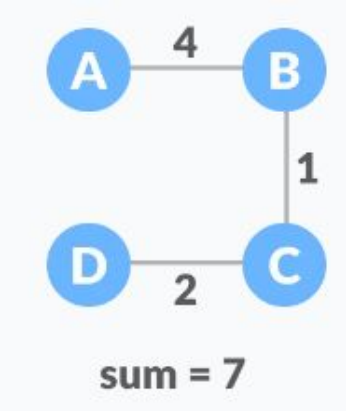
- A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.









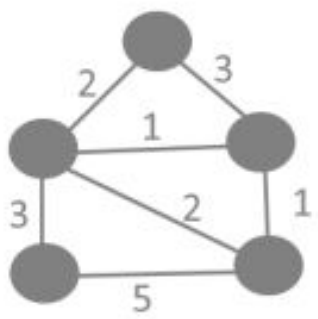


Minimum Spanning Tree

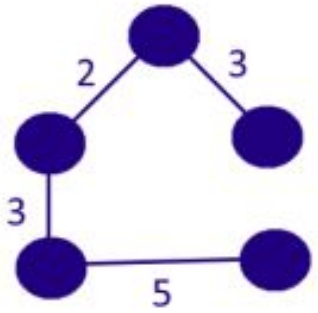
- The minimum spanning tree from a graph is found using the following algorithms:
    - Prim's Algorithm
    - Kruskal's Algorithm

Kruskal's algorithm

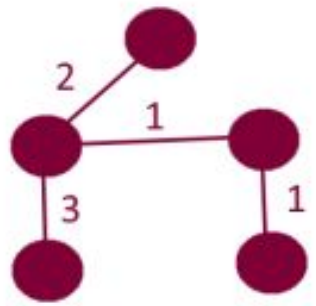
- Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which
  - form a tree that includes every vertex
  - has the minimum sum of weights among all the trees that can be formed from the graph



Graph



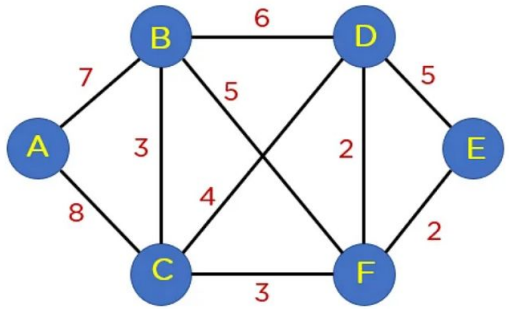
Spanning Tree
Cost = 13

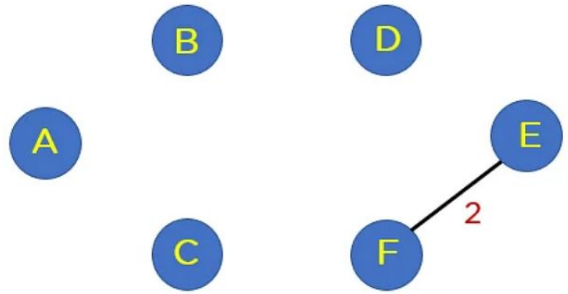


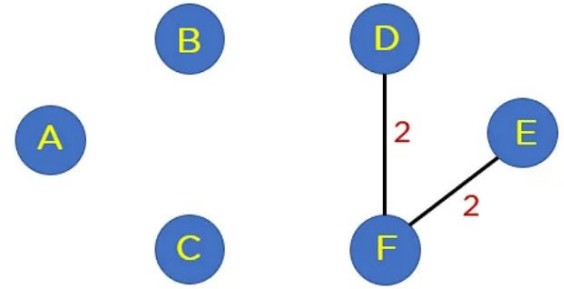
Minimum Spanning
Tree, Cost = 7

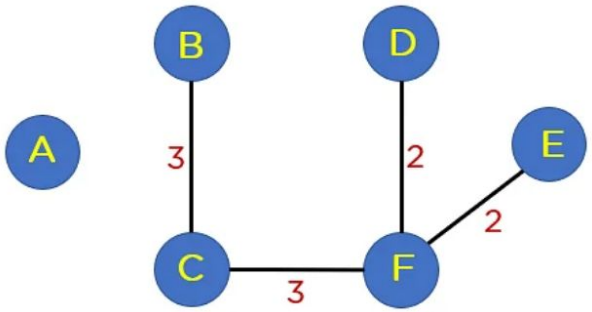
Kruskal's algorithm

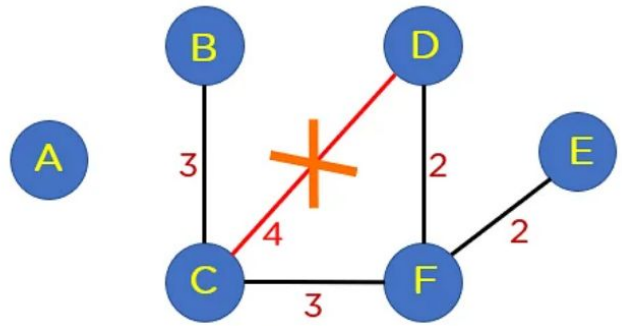
- We start from the edges with the lowest weight and keep adding edges until we reach our goal.
- The steps for implementing Kruskal's algorithm are as follows:
  - Sort all the edges from low weight to high
  - Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
  - Keep adding edges until we reach all vertices.

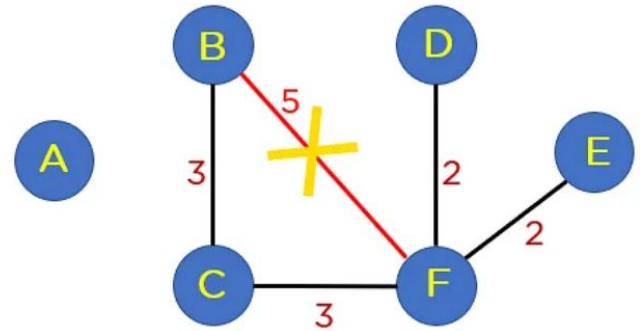

A
B
C
D
E
F
7
6
5
3
5
2
4
8
2
3

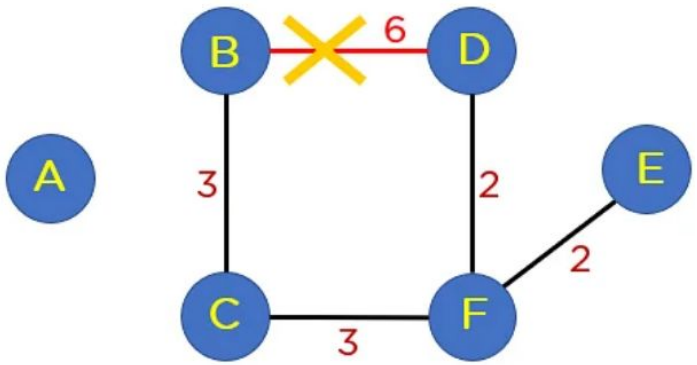

A
B
C
D
E
F
2
2

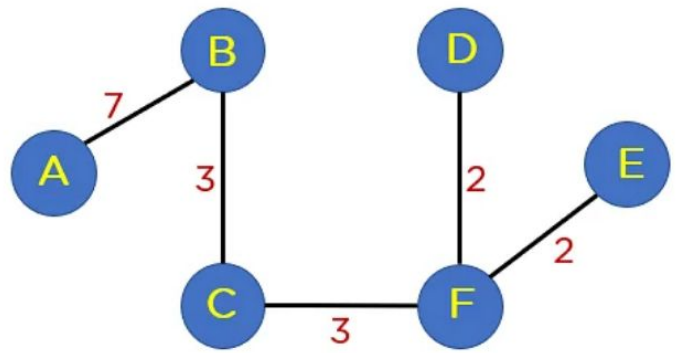

A
B
C
D
E
F
2
2


A
B
C
D
E
F
3
3
2
2


A
B
C
D
E
F
3
4
3
2
2


A
B
C
D
E
F
3
5
3
2
2


A
B
C
D
E
F
3
6
3
2
2


A
B
C
D
E
F
7
3
3
2
2

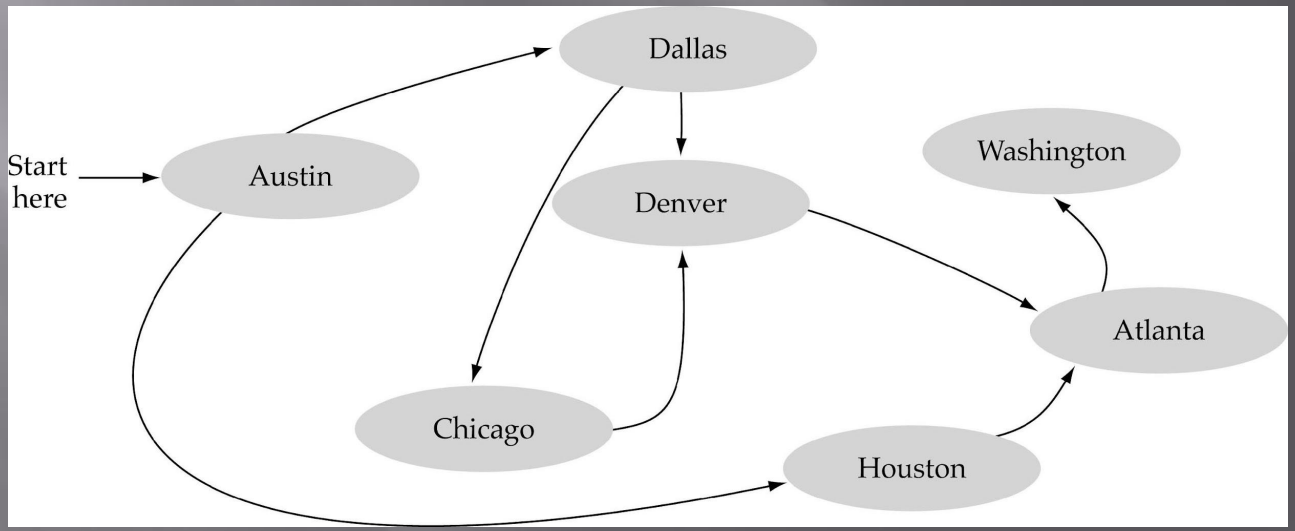
Prim's algorithm

- We start from one vertex and keep adding edges with the lowest weight until we reach our goal.
- The steps for implementing Prim's algorithm are as follows:
  - Initialize the minimum spanning tree with a vertex chosen at random.
  - Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
  - Keep repeating step 2 until we get a minimum spanning tree

GRAPHS

What is a graph?

- A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other
- The set of edges describes relationships among the vertices



Formal definition of graphs

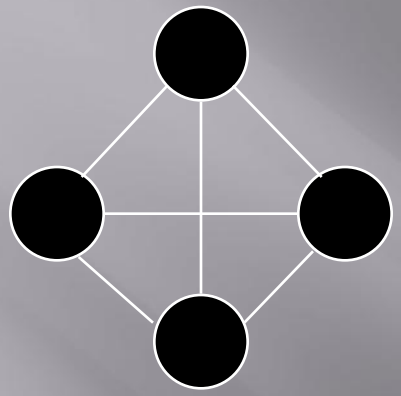
- A graph *G* is defined as follows:

  $G=(V,E)$

  V(G): a finite, nonempty set of vertices

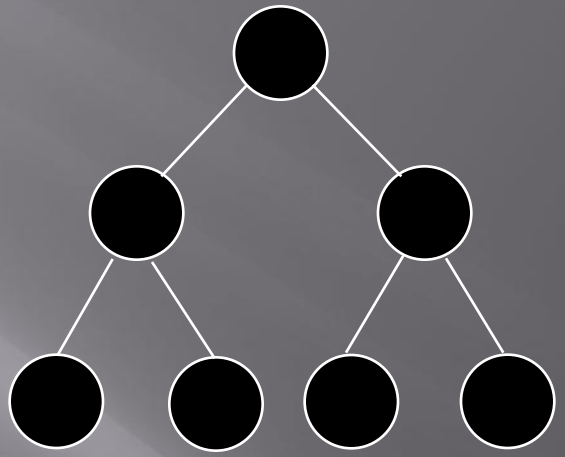
  E(G): a set of edges (pairs of vertices)

Examples for Graph



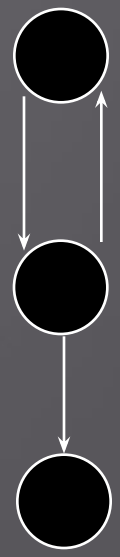
G_1

complete graph



G_2

incomplete graph



G_3

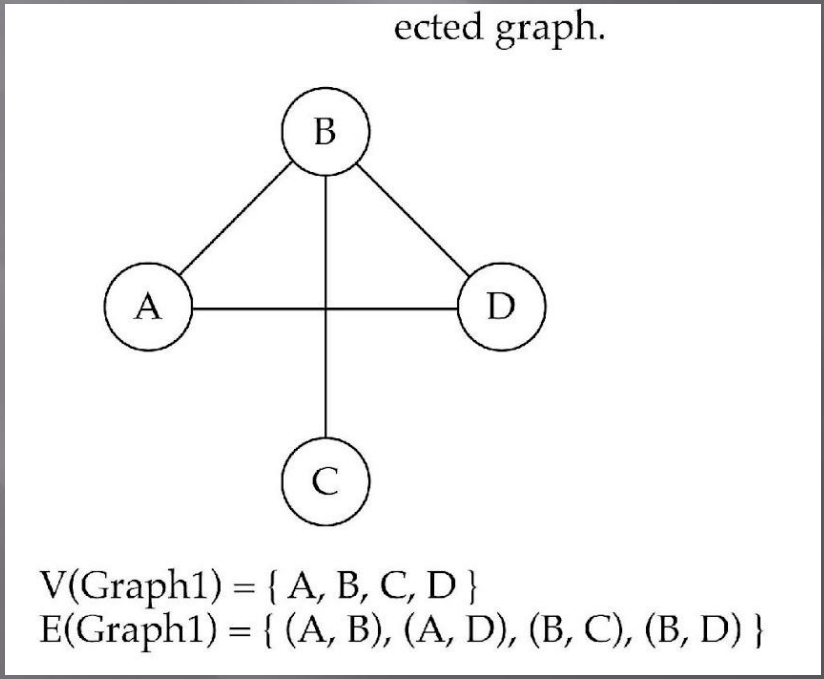
$V(G_1)=\{0,1,2,3\}$
$V(G_2)=\{0,1,2,3,4,5,6\}$
$V(G_3)=\{0,1,2\}$

$E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$
$E(G_2)=\{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$
$E(G_3)=\{<0,1>,<1,0>,<1,2>\}$

complete undirected graph: n(n-1)/2 edges
complete directed graph: n(n-1) edges

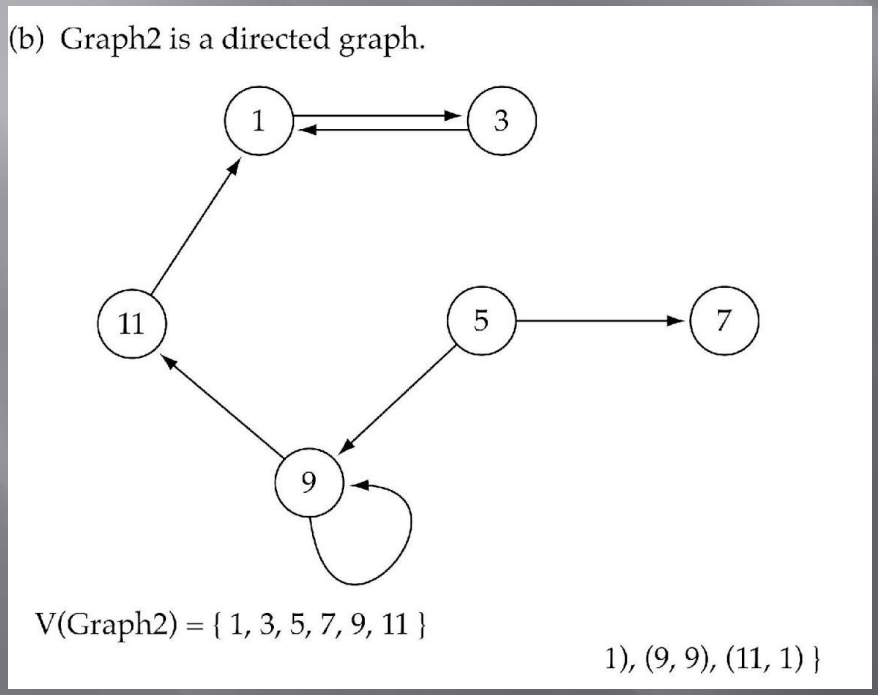
Directed vs. undirected graphs

- When the edges in a graph have no direction, the graph is called *undirected*



Directed vs. undirected graphs (cont.)

- When the edges in a graph have a direction, the graph is called *directed* (or *digraph*)

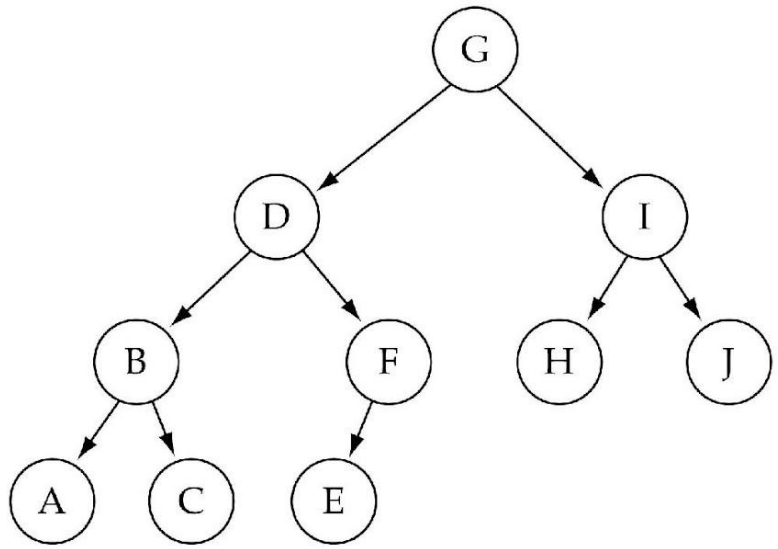


Warning: if the graph is directed, the order of the vertices in each edge is important !!

Trees vs graphs

- Trees are special cases of graphs!!

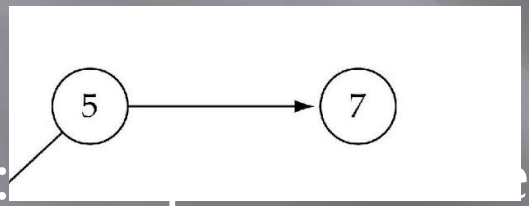
(c) Graph3 is a directed graph.



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 terminology

- Adjacent nodes: two nodes are adjacent if they are connected by an edge



5 is adjacent to 7
7 is adjacent from 5

- Path: [illegible]ertices that connect two nodes in a graph
- Complete graph: a graph in which every vertex is directly connected to every other vertex

Complete Graph

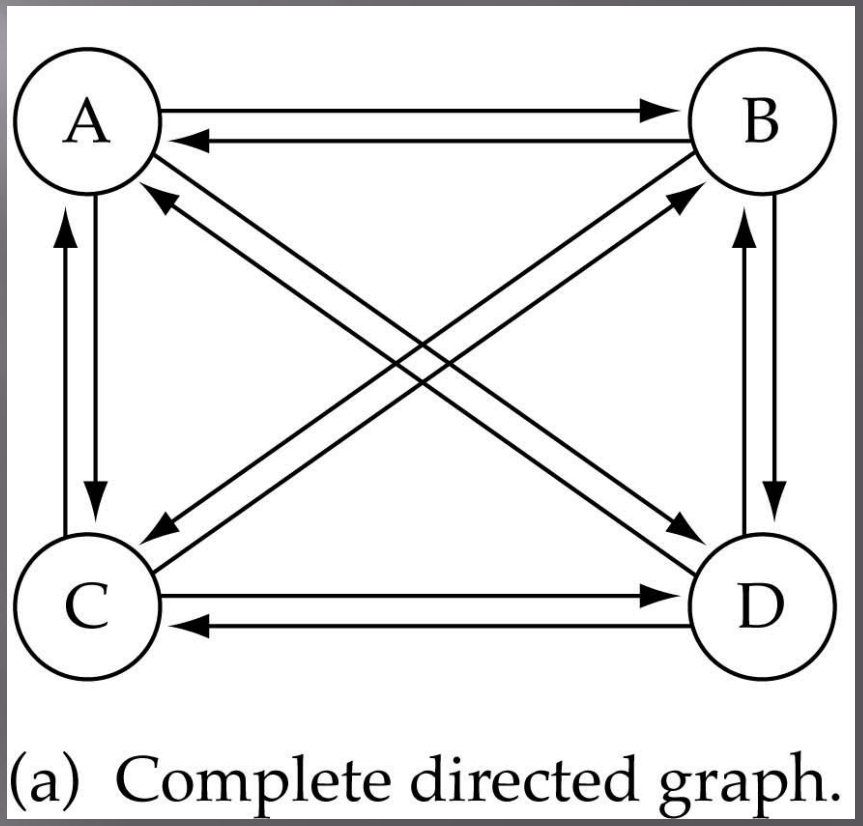
- A complete graph is a graph that has the maximum number of edges
  - for **undirected graph** with n vertices, the maximum number of edges is n(n-1)/2
  - for directed graph with n vertices, the maximum number of edges is n(n-1)
  - example: G1 is a complete graph

Graph terminology (cont.)

- What is the number of edges in a complete directed graph with N vertices?

  *N * (N-1)*

  $O(N^2)$



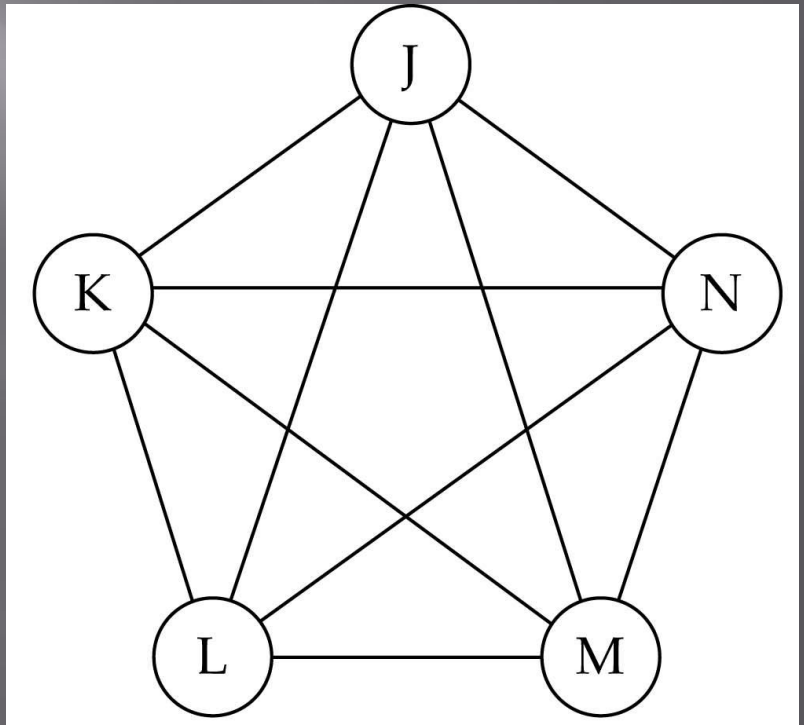
(a) Complete directed graph.

Graph terminology (cont.)

- What is the number of edges in a complete undirected graph with N vertices?

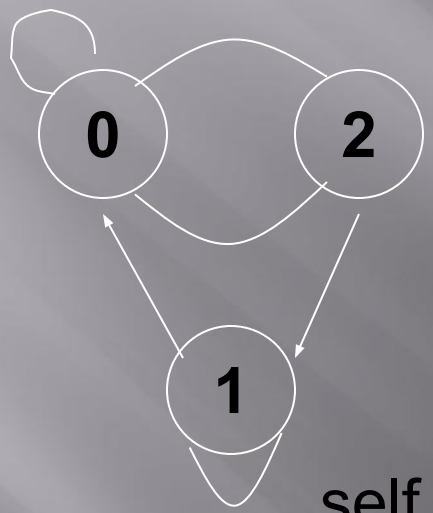
  *N * (N-1) / 2*

  $O(N^2)$



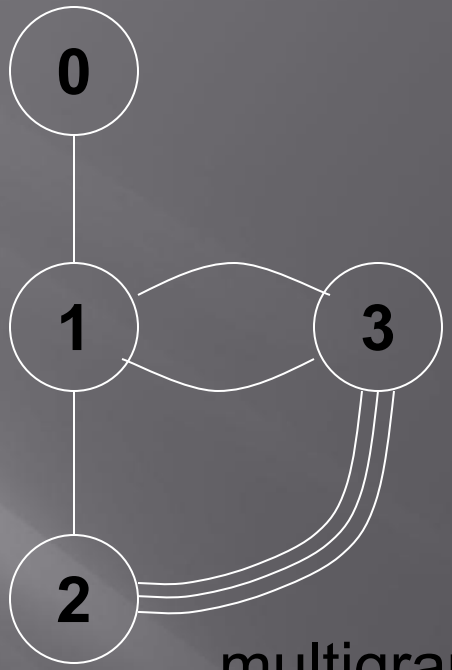
(b) Complete undirected graph.

Example of a graph with feedback loops and a multigraph



self edge

(a)



(b)

multigraph:
multiple occurrences
of the same edge

Simple Path and Style

- A simple path is a path in which all vertices, except possibly the first and the last, are distinct
- A cycle is a simple path in which the first and the last vertices are the same
- In an undirected graph G, two vertices, v_0 and v_1, are connected if there is a path in G from v_0 to v_1
- An undirected graph is connected if, for every pair of distinct vertices v_i, v_j, there is a path from v_i to v_j

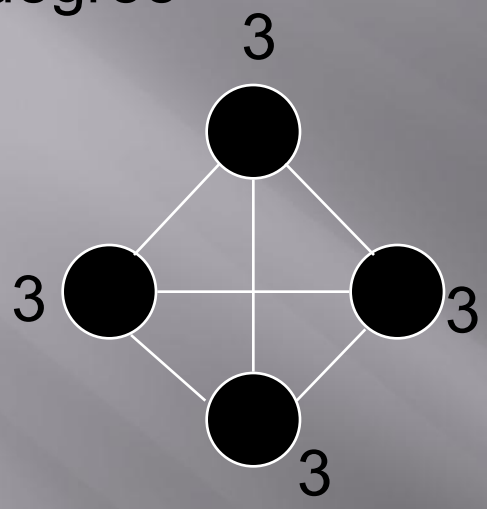
Degree

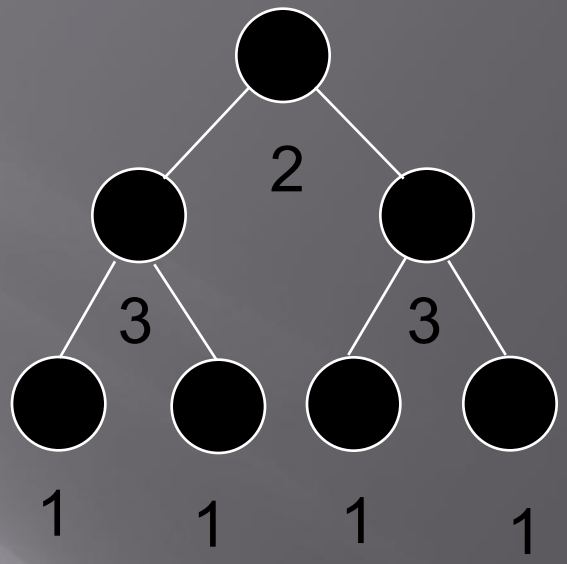
- The degree of a vertex is the number of edges incident to that vertex
- For directed graph,
  - the in-degree of a vertex *v* is the number of edges that have *v* as the head
  - the out-degree of a vertex *v* is the number of edges that have *v* as the tail
  - if d_i is the degree of a vertex *i*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

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

undirected graph

degree





directed graph

in-degree

out-degree

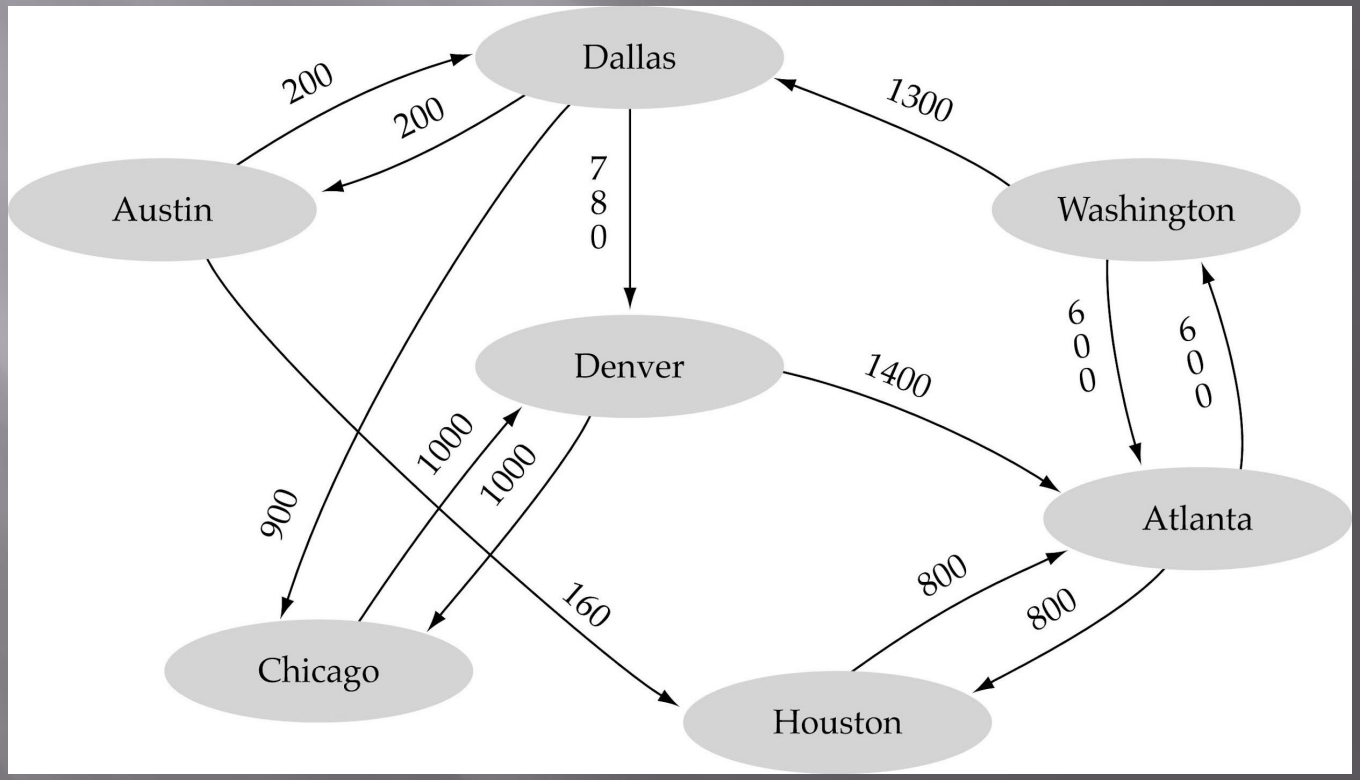
in:1, out: 1

in: 1, out: 2

in: 1, out: 0

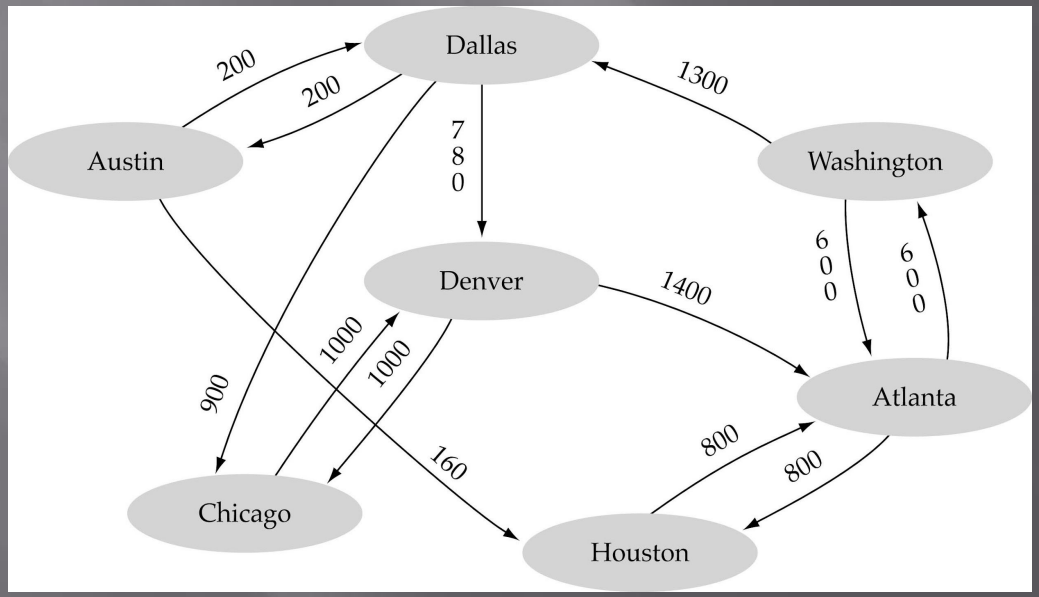
Graph terminology (cont.)

- Weighted graph: a graph in which each edge carries a value



Graph implementation

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



Array-based implementation

graph

.numVertices 7

.vertices

[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

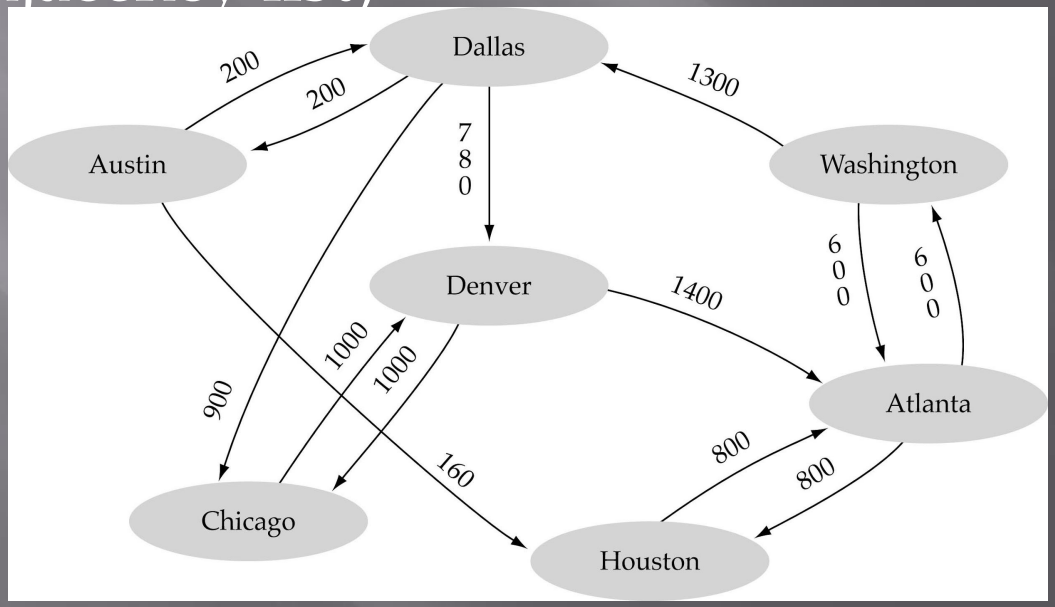
.edges

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

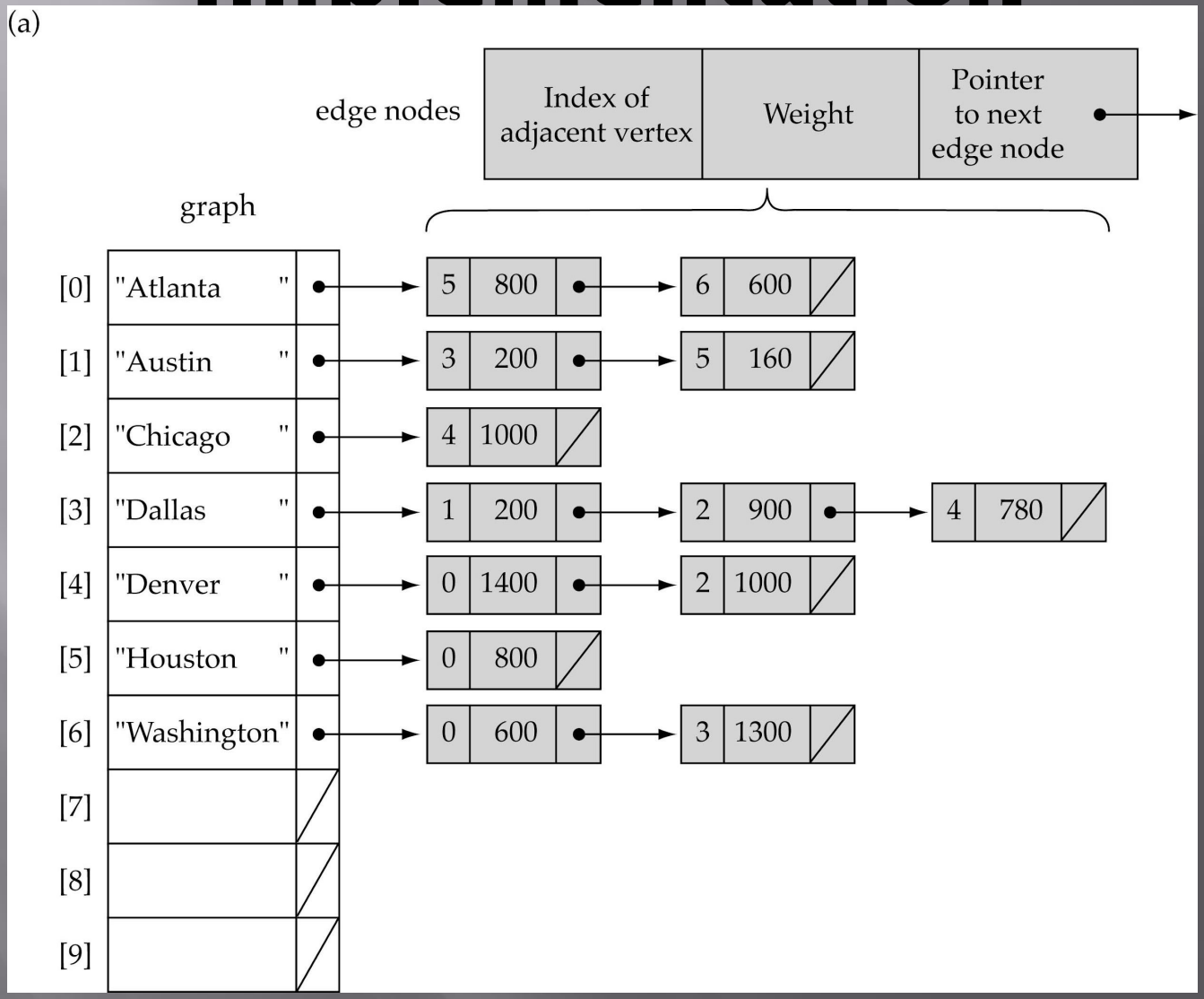
(Array positions marked '•' are undefined)

Graph implementation (cont.)

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



Linked-list implementation



Adjacency matrix vs. adjacency list representation

- **Adjacency matrix**
  - Good for dense graphs -- $|E| \sim O(|V|^2)$
  - Memory requirements: $O(|V| + |E|) = O(|V|^2)$
  - Connectivity between two vertices can be tested quickly
- **Adjacency list**
  - Good for sparse graphs -- $|E| \sim O(|V|)$
  - Memory requirements: $O(|V| + |E|) = O(|V|)$
  - Vertices adjacent to another vertex can be found quickly

Graph specification based on adjacency matrix representation

```
const int NULL_EDGE = 0;

template<class VertexType>
class GraphType {
  public:
    GraphType(int);
    ~GraphType();
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void AddVertex(VertexType);
    void AddEdge(VertexType, VertexType, int);
    int WeightIs(VertexType, VertexType);
    void GetToVertices(VertexType, QueType<VertexType>&);
    void ClearMarks();
    void MarkVertex(VertexType);
    bool IsMarked(VertexType) const;
```

```
  private:
    int numVertices;
    int maxVertices;
    VertexType* vertices;
    int **edges;
    bool* marks;
};
```

(continues)

```
template<class VertexType>
GraphType<VertexType>::GraphType(int maxV)
{
 numVertices = 0;
 maxVertices = maxV;
 vertices = new VertexType[maxV];
 edges = new int[maxV];
 for(int i = 0; i < maxV; i++)
   edges[i] = new int[maxV];
 marks = new bool[maxV];
}

template<class VertexType>
GraphType<VertexType>::~GraphType()
{
 delete [] vertices;
 for(int i = 0; i < maxVertices; i++)
   delete [] edges[i];
 delete [] edges;
 delete [] marks;
}
```

(continues)

```
void GraphType<VertexType>::AddVertex(VertexType vertex)
{
 vertices[numVertices] = vertex;

 for(int index = 0; index < numVertices; index++) {
   edges[numVertices][index] = NULL_EDGE;
   edges[index][numVertices] = NULL_EDGE;
 }

 numVertices++;
}

template<class VertexType>
void GraphType<VertexType>::AddEdge(VertexType fromVertex,
                    VertexType toVertex, int weight)
{
 int row;
 int column;

 row = IndexIs(vertices, fromVertex);
 col = IndexIs(vertices, toVertex);
 edges[row][col] = weight;
}
```

(continues)

```
template<class VertexType>
int GraphType<VertexType>::WeightIs(VertexType fromVertex,
                                    VertexType toVertex)
{
 int row;
 int column;

 row = IndexIs(vertices, fromVertex);
 col = IndexIs(vertices, toVertex);
 return edges[row][col];
}
```

Graph searching

- *Problem*: find a path between two nodes of the graph (e.g., Austin and Washington)
- *Methods*: Depth-First-Search (DFS) or Breadth-First-Search (BFS)

Depth-First-Search (DFS)

- What is the idea behind DFS?
  - Travel as far as you can down a path
  - Back up *as little as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)
- DFS can be implemented efficiently using a

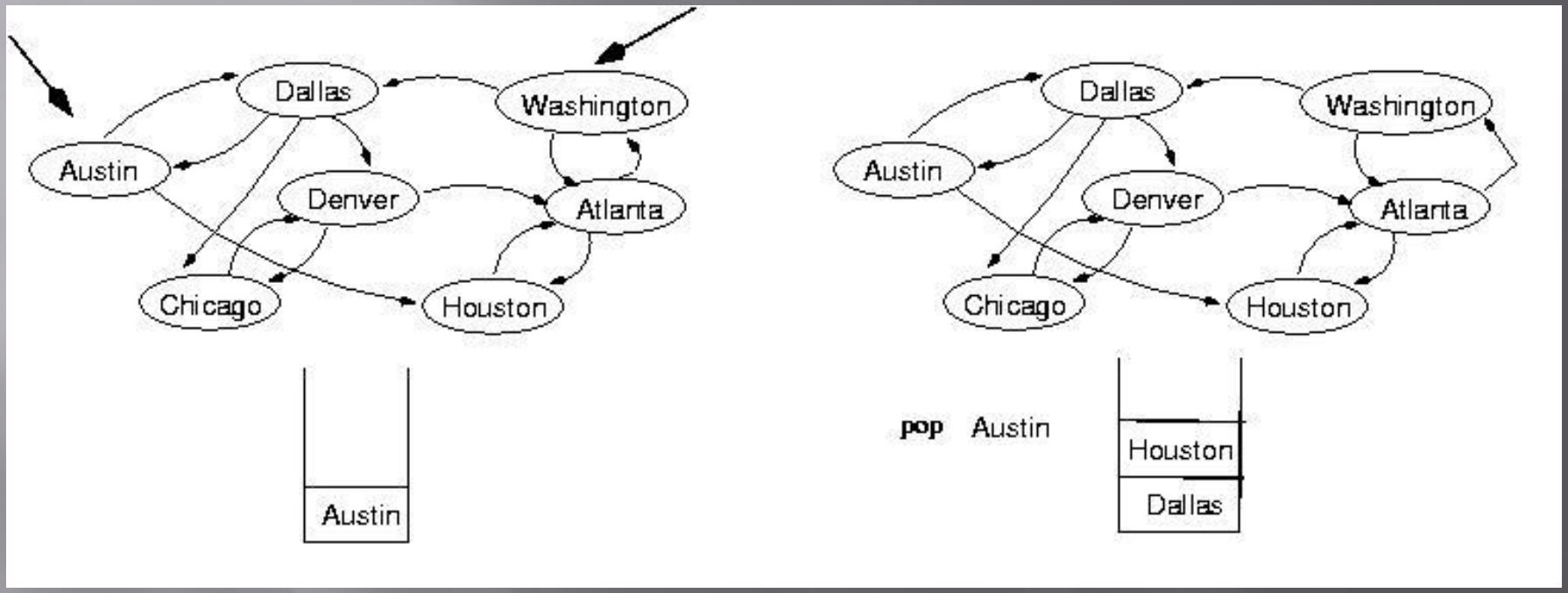
stack

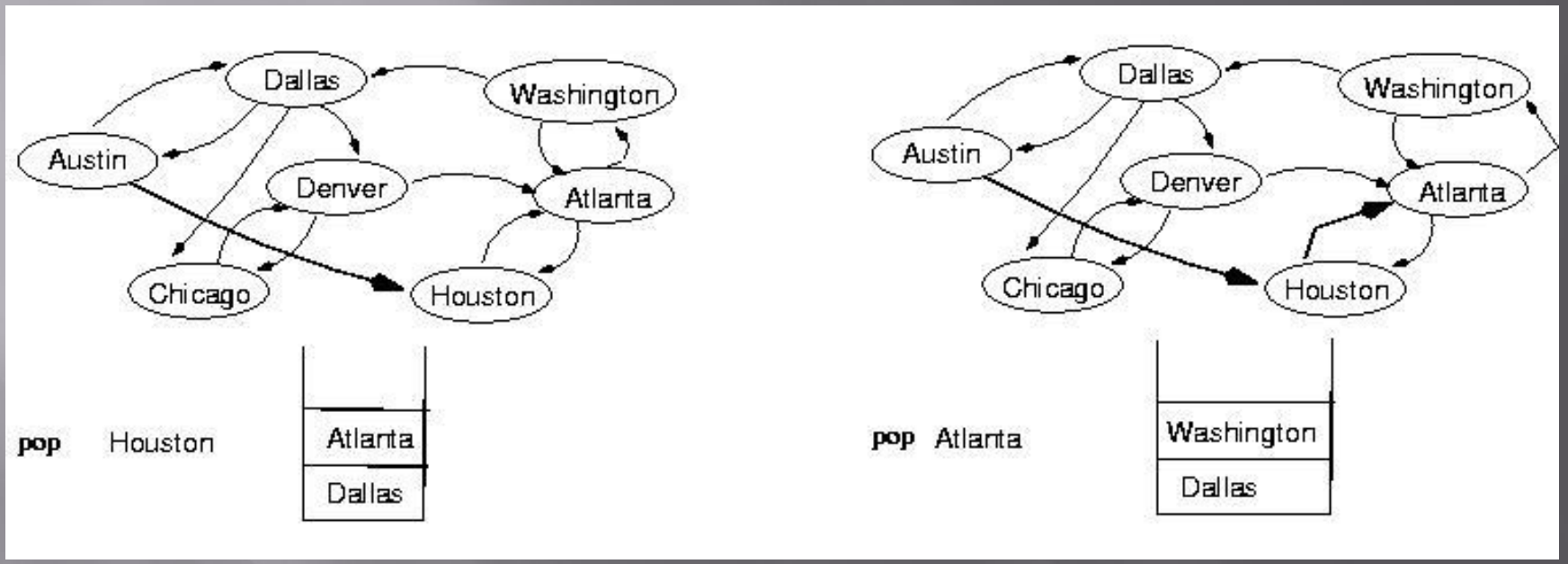
Depth-First-Search (DFS) *(cont.)*

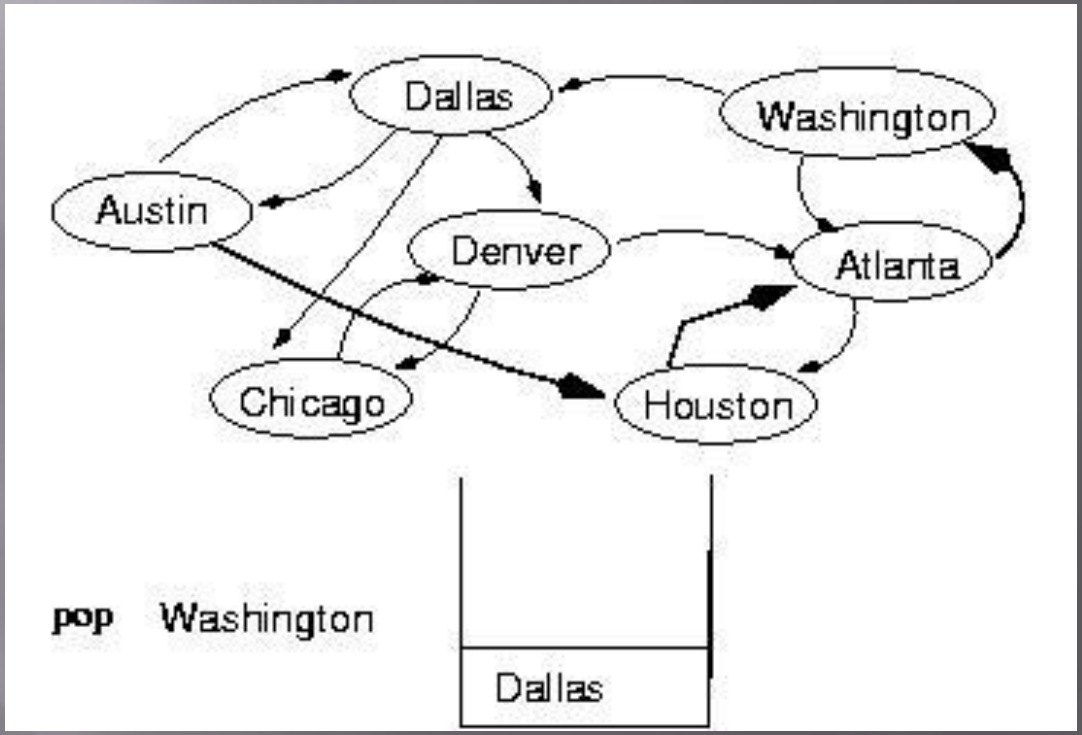
```
Set found to false
stack.Push(startVertex)
DO
  stack.Pop(vertex)
  IF vertex == endVertex
    Set found to true
  ELSE
    Push all adjacent vertices onto stack
WHILE !stack.IsEmpty() AND !found

IF(!found)
  Write "Path does not exist"
```

start

end


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
Austin
Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
pop Austin
Houston
Dallas


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
pop Houston
Atlanta
Dallas
Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
pop Atlanta
Washington
Dallas


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
pop Washington
Dallas

```
template <class ItemType>
void DepthFirstSearch(GraphType<VertexType> graph,
      VertexType startVertex, VertexType endVertex)
{
 StackType<VertexType> stack;
 QueType<VertexType> vertexQ;

 bool found = false;
 VertexType vertex;
 VertexType item;

 graph.ClearMarks();
 stack.Push(startVertex);
 do {
   stack.Pop(vertex);
   if(vertex == endVertex)
     found = true;
```

(continues)

```
  else {
      if(!graph.IsMarked(vertex)) {
        graph.MarkVertex(vertex);
        graph.GetToVertices(vertex, vertexQ);

        while(!vertexQ.IsEmpty()) {
          vertexQ.Dequeue(item);
          if(!graph.IsMarked(item))
            stack.Push(item);
        }
      }
    } while(!stack.IsEmpty() && !found);

    if(!found)
      cout << "Path not found" << endl;
}
```

(continues)

```
template<class VertexType>
void GraphType<VertexType>::GetToVertices(VertexType vertex,
                                  QueTye<VertexType>& adjvertexQ)
{
 int fromIndex;
 int toIndex;

 fromIndex = IndexIs(vertices, vertex);
 for(toIndex = 0; toIndex < numVertices; toIndex++)
   if(edges[fromIndex][toIndex] != NULL_EDGE)
     adjvertexQ.Enqueue(vertices[toIndex]);
}
```

Breadth-First-Searching (BFS)

- What is the idea behind BFS?
  - Look at all possible paths at the same depth before you go at a deeper level
  - Back up *as far as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)

Breadth-First-Searching (BFS) (cont.)

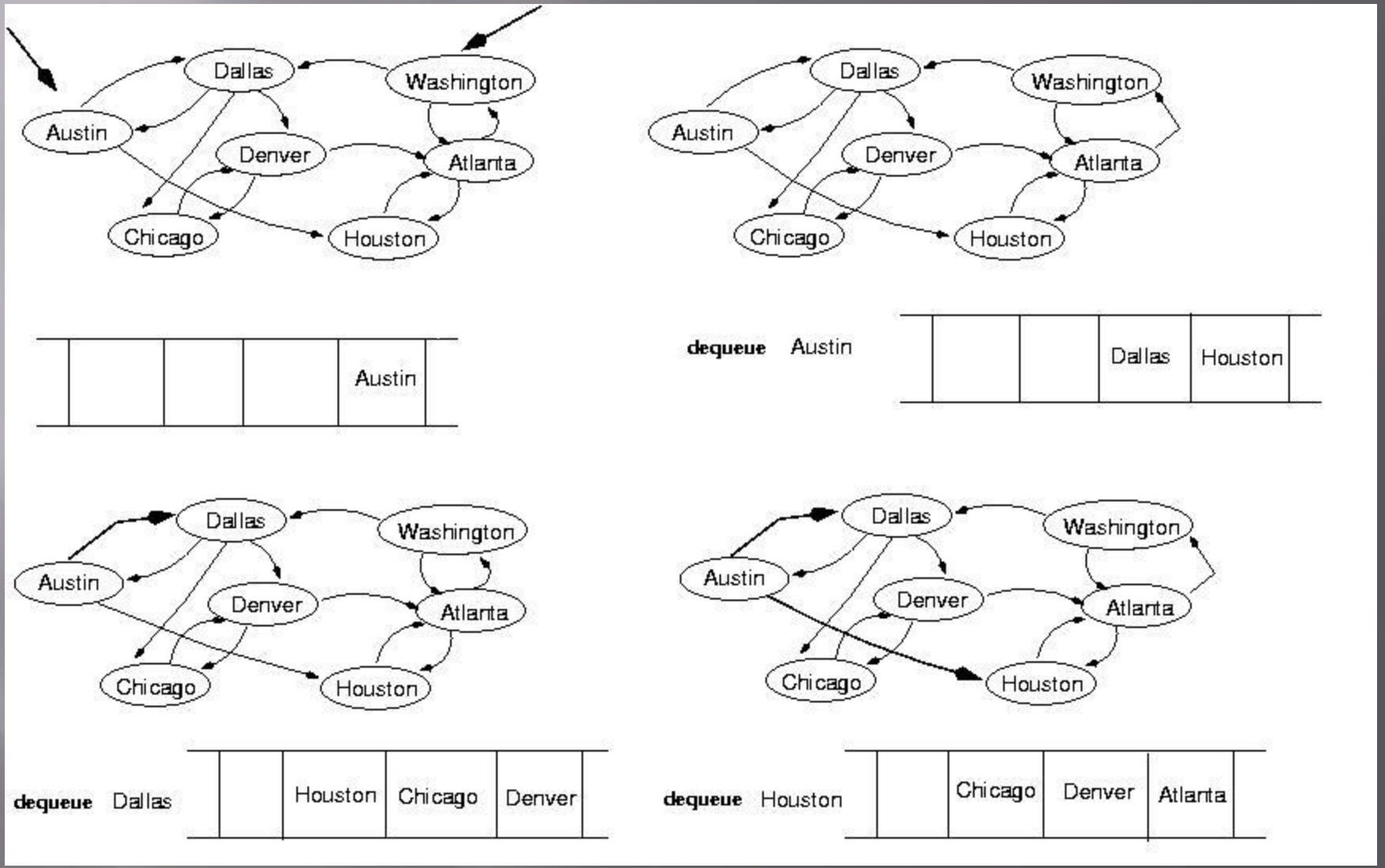
- BFS can be implemented efficiently using a *queue*

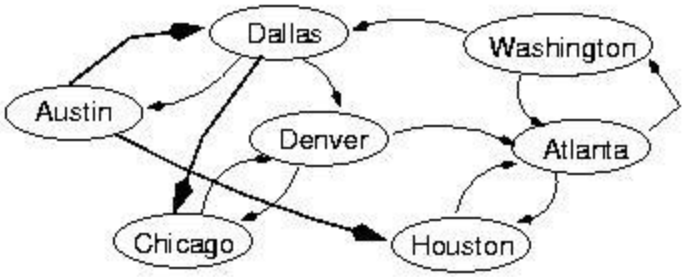
```
Set found to false
queue.Enqueue(startVertex)
DO
  queue.Dequeue(vertex)
  IF vertex == endVertex
    Set found to true
  ELSE
    Enqueue all adjacent vertices onto queue
WHILE !queue.IsEmpty() AND !found
```

```
IF(!found)
  Write "Path does not exist"
```

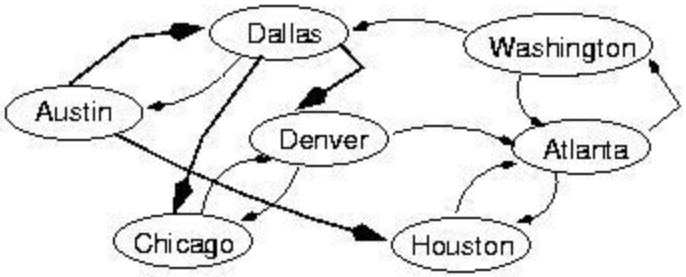
- Should we mark a vertex when it is enqueued or when it is dequeued ?

start end

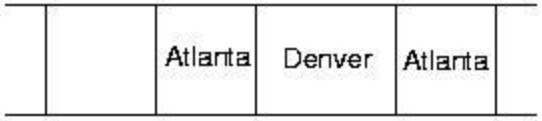

Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
Austin
dequeue Austin
Dallas
Houston
dequeue Dallas
Houston
Chicago
Denver
dequeue Houston
Chicago
Denver
Atlanta

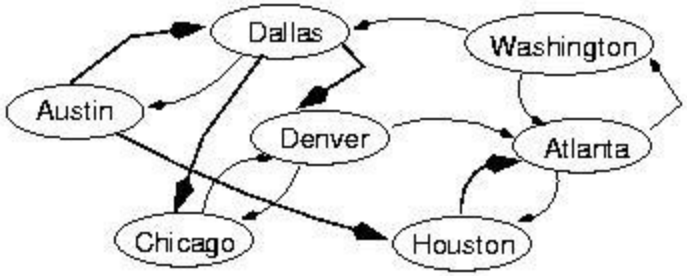


dequeue Chicago

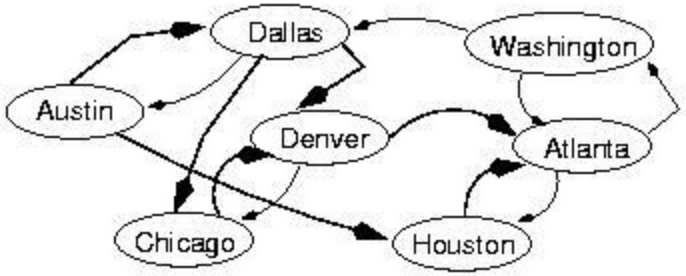


dequeue Denver

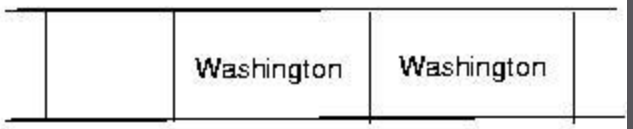


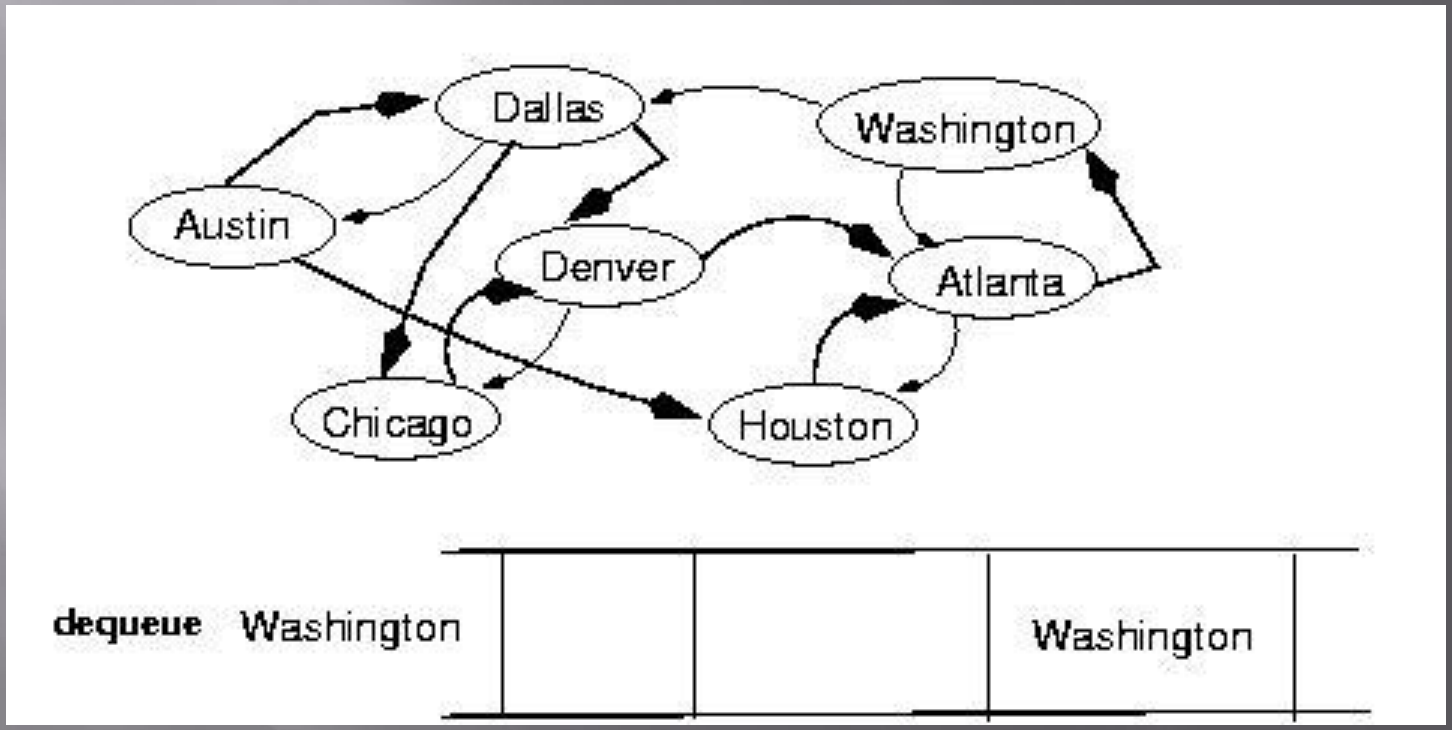


dequeue Atlanta



dequeue Denver,
next: Atlanta




Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
dequeue
Washington
Washington

```
template<class VertexType>
void BreadthFirtsSearch(GraphType<VertexType> graph,
      VertexType startVertex, VertexType endVertex);
{
 QueType<VertexType> queue;
 QueType<VertexType> vertexQ;//

 bool found = false;
 VertexType vertex;
 VertexType item;

 graph.ClearMarks();
 queue.Enqueue(startVertex);
 do {
   queue.Dequeue(vertex);
   if(vertex == endVertex)
     found = true;
```

(continues)

```
  else {
      if(!graph.IsMarked(vertex)) {
        graph.MarkVertex(vertex);
        graph.GetToVertices(vertex, vertexQ);

        while(!vertxQ.IsEmpty()) {
          vertexQ.Dequeue(item);
          if(!graph.IsMarked(item))
            queue.Enqueue(item);
        }
      }
    }
  } while (!queue.IsEmpty() && !found);

  if(!found)
    cout << "Path not found" << endl;
}
```

Single-source shortest-path problem

- There are multiple paths from a source vertex to a destination vertex
- *Shortest path*: the path whose total weight (i.e., sum of edge weights) is minimum
- Examples:
  - Austin->Houston->Atlanta->Washington: 1560 miles
  - Austin->Dallas->Denver->Atlanta->Washington: 2980 miles

Single-source shortest-path problem (cont.)

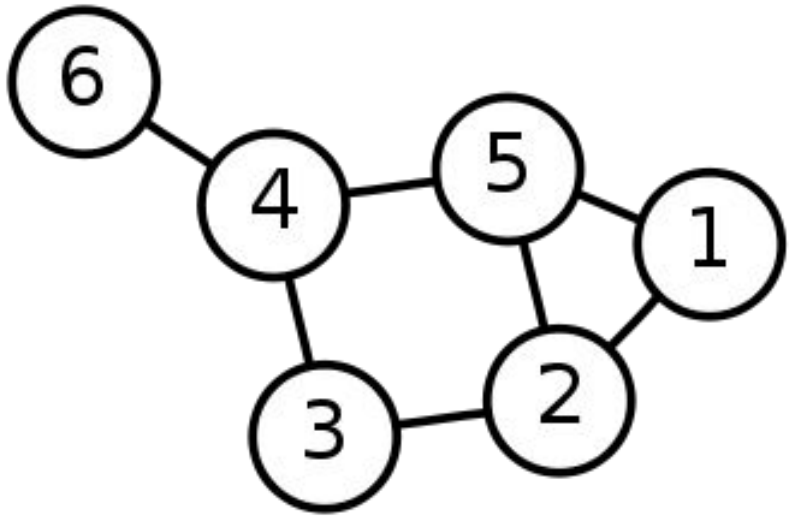
- Common algorithms: *Dijkstra's* algorithm, *Bellman-Ford* algorithm
- BFS can be used to solve the shortest graph problem when the graph is weightless or all the weights are the same
  - (mark vertices before Enqueue)

Thanks

Dijkstra's algorithm

Single-Source Shortest Path Problem

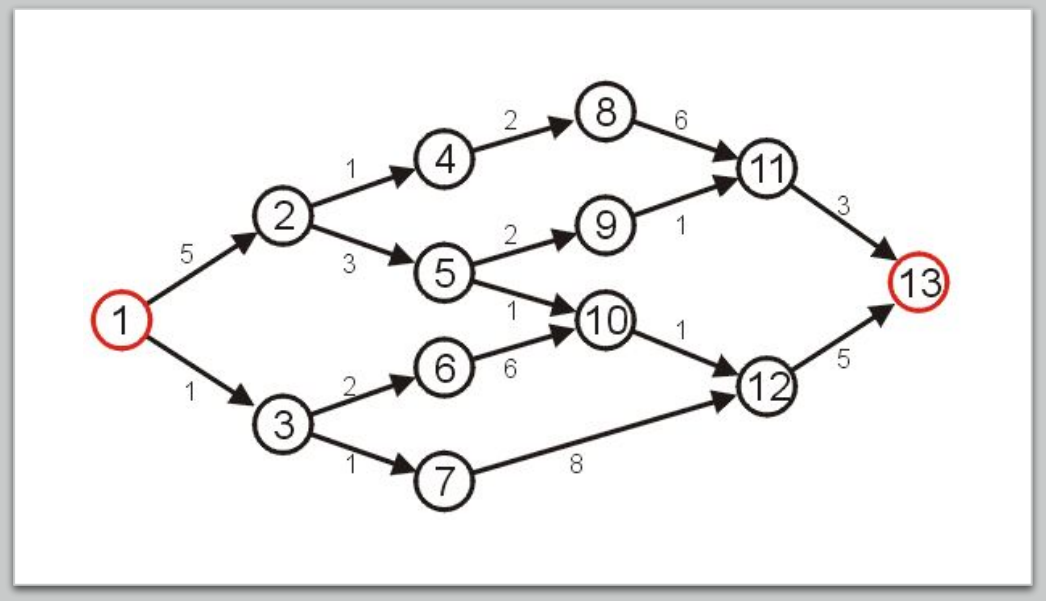
Single-Source Shortest Path Problem - The problem of finding shortest paths from a source vertex *v* to all other vertices in the graph.

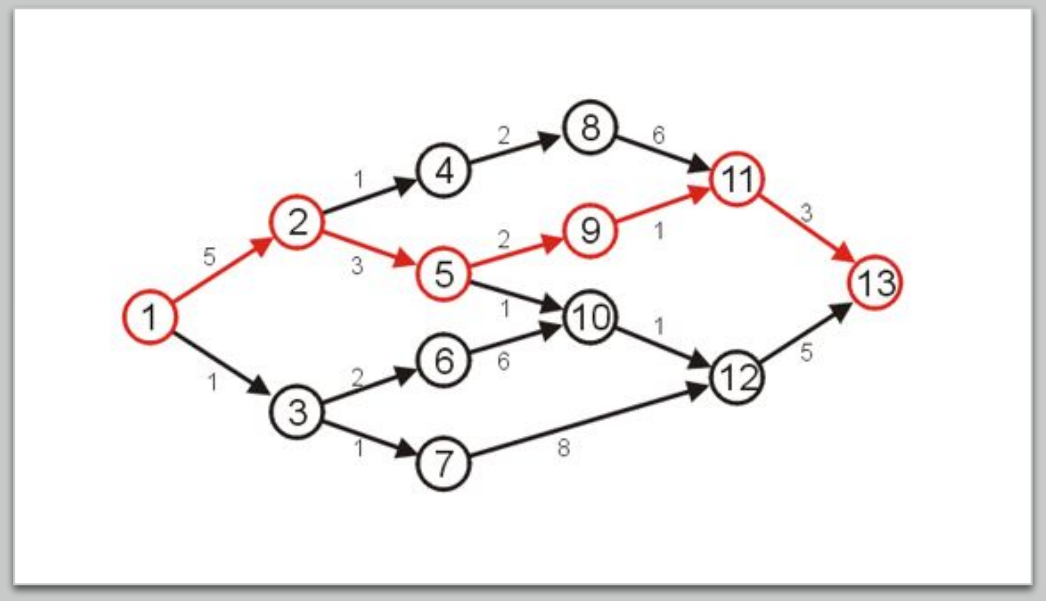


Shortest Path

- Given the graph below, suppose we wish to find the shortest path from vertex 1 to vertex 13

- After some consideration, we may determine that the shortest path is as follows, with length 14
- Other paths exists, but they are longer





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.

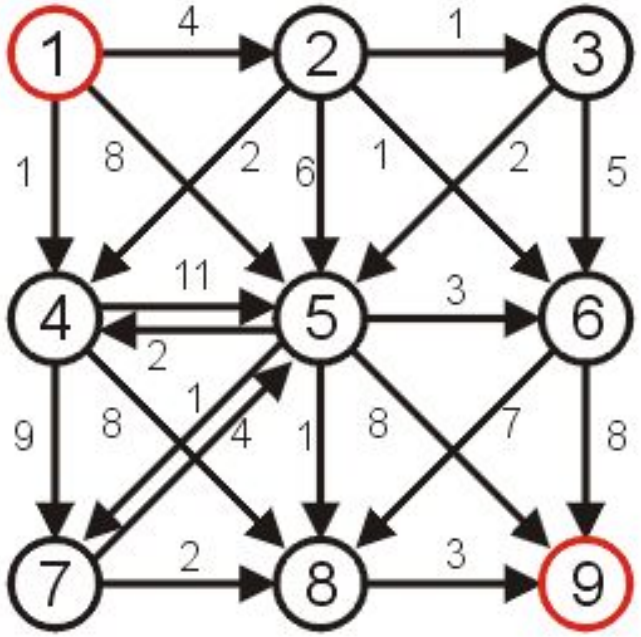
Approach: Greedy

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

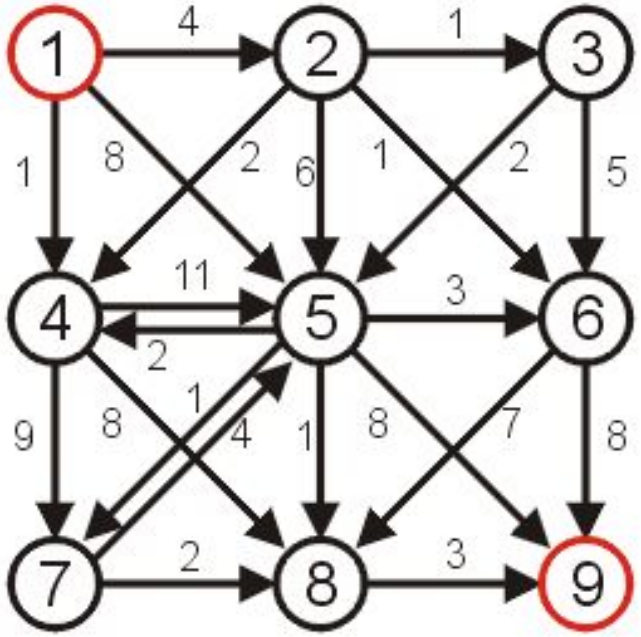
Shortest Path

- Consider the following graph with positive weights and cycles.



Dijkstra's Algorithm

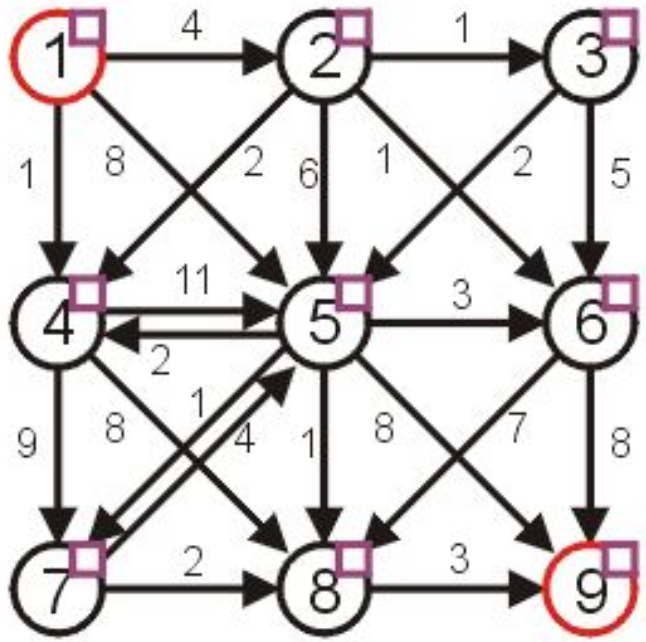
- A first attempt at solving this problem might require an array of Boolean values, all initially false, that indicate whether we have found a path from the source.



1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F
9	F

Dijkstra's Algorithm

- Graphically, we will denote this with check boxes next to each of the vertices (initially unchecked)

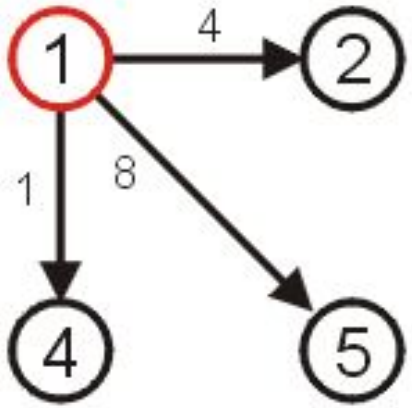


Dijkstra's Algorithm

- We will work bottom up.
  - Note that if the starting vertex has any adjacent edges, then there will be one vertex that is the shortest distance from the starting vertex. This is the shortest reachable vertex of the graph.
- We will then try to extend any ***existing*** paths to new vertices.
- Initially, we will start with the path of length 0
  - this is the trivial path from vertex 1 to itself

Dijkstra's Algorithm

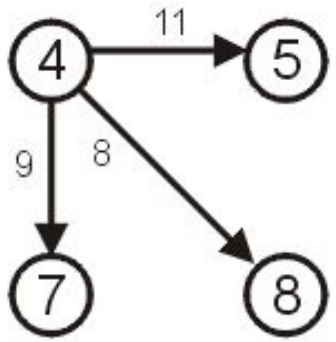
- If we now extend this path, we should consider the paths
  - (1, 2) length 4
  - (1, 4) length 1
  - (1, 5) length 8



The *shortest* path so far is (1, 4) which is c

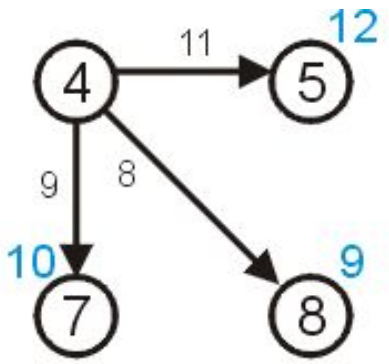
Dijkstra's Algorithm

- Thus, if we now examine vertex 4, we may deduce that there exist the following paths:
  - (1, 4, 5) length 12
  - (1, 4, 7) length 10
  - (1, 4, 8) length 9



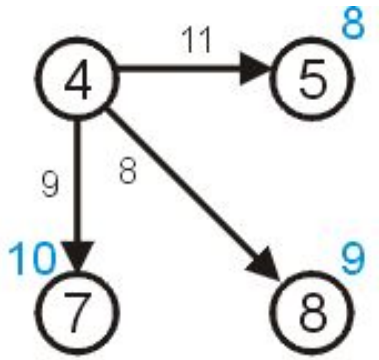
Dijkstra's Algorithm

- We need to remember that the length of that path from node 1 to node 4 is 1
- Thus, we need to store the length of a path that goes through node 4:
  - 5 of length 12
  - 7 of length 10
  - 8 of length 9



Dijkstra's Algorithm

- We have already discovered that there is a path of length 8 to vertex 5 with the path (1, 5).
- Thus, we can safely ignore this longer path.



Dijkstra's Algorithm

- We now know that:
  - There exist paths from vertex 1 to vertices {2,4,5,7,8}.
  - We know that the shortest path from vertex 1 to vertex 4 is of length 1.
  - We know that the shortest path to the other vertices {2,5,7,8} is at most the length listed in the table to the right.

Vertex	Length
1	0
2	4
4	1
5	8
7	10
8	9

Dijkstra's Algorithm

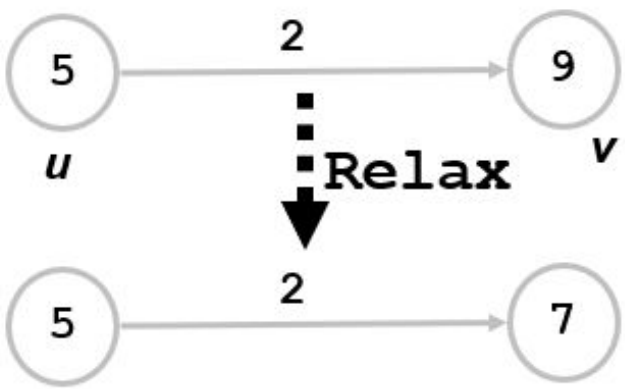
- There cannot exist a shorter path to either of the vertices 1 or 4, since the distances can only increase at each iteration.
- We consider these vertices to be ***visited***

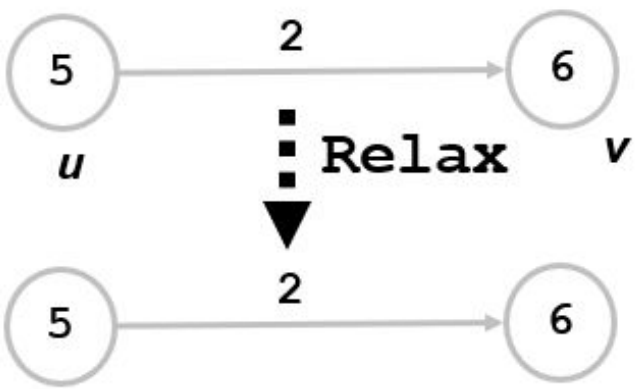
Vertex	Length
1	0
2	4
4	1
5	8
7	10
8	9

Dijkstra's Algorithm

- Maintaining this shortest discovered distance d[*v*] is called **relaxation**:

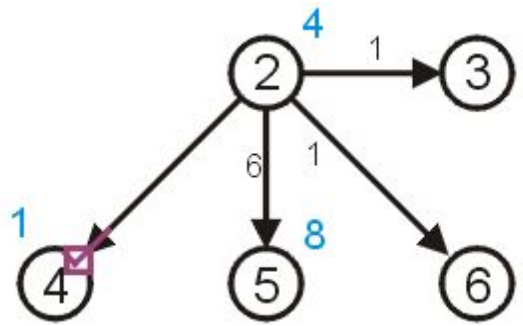
```
Relax(u,v,w) {
    if (d[v] > d[u]+w) then
        d[v]=d[u]+w;
}
```





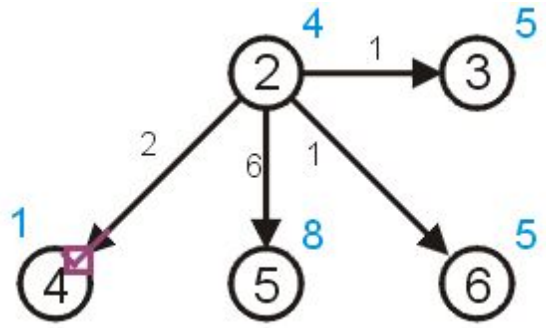
Dijkstra's Algorithm

- In Dijkstra's algorithm, we always take the next unvisited vertex which has the current shortest path from the starting vertex in the table.
- This is vertex 2



Dijkstra's Algorithm

- We can try to update the shortest paths to vertices 3 and 6 (both of length 5) however:
  - there already exists a path of length 8 < 10 to vertex 5 (10 = 4 + 6)
  - we already know the shortest path to 4 is 1



Dijkstra’s Algorithm

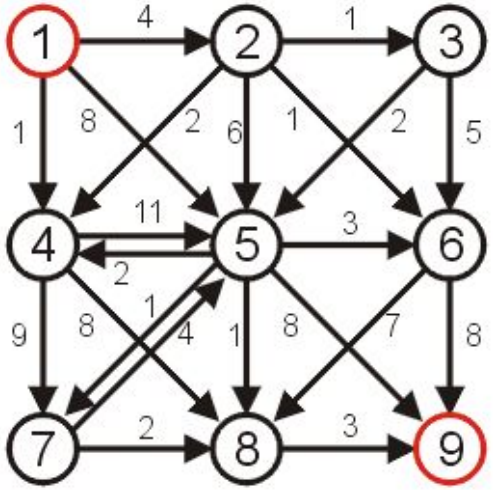
- To keep track of those vertices to which no path has reached, we can assign those vertices an initial distance of either
  - infinity (∞),
  - a number larger than any possible path, or
  - a negative number
- For demonstration purposes, we will use ∞

Dijkstra’s Algorithm

- As well as finding the length of the shortest path, we’d like to find the corresponding shortest path
- Each time we update the shortest distance to a particular vertex, we will keep track of the predecessor used to reach this vertex on the shortest path.

Dijkstra's Algorithm

- We will store a table of pointers, each initially 0
- This table will be updated each time a distance is updated



1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Dijkstra’s Algorithm

- Graphically, we will display the reference to the preceding vertex by a red arrow
  - if the distance to a vertex is ∞, there will be no preceding vertex
  - otherwise, there will be exactly one preceding vertex

Dijkstra's Algorithm

- Thus, for our initialization:
  - we set the current distance to the initial vertex as 0
  - for all other vertices, we set the current distance to ∞
  - all vertices are initially marked as unvisited
  - set the previous pointer for all vertices to null

Dijkstra's Algorithm

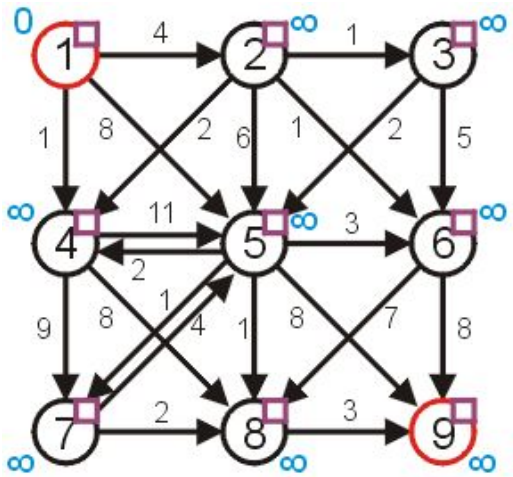
- Thus, we iterate:
  - find an unvisited vertex which has the shortest distance to it
  - mark it as visited
  - for each unvisited vertex which is adjacent to the current vertex:
    - add the distance to the current vertex to the weight of the connecting edge
    - if this is less than the current distance to that vertex, update the distance and set the parent vertex of the adjacent vertex to be the current vertex

Dijkstra's Algorithm

- Halting condition:
  - we successfully halt when the vertex we are visiting is the target vertex
  - if at some point, all remaining unvisited vertices have distance ∞, then no path from the starting vertex to the end vertex exits
- Note: We do not halt just because we have updated the distance to the end vertex, we have to **visit** the target vertex.

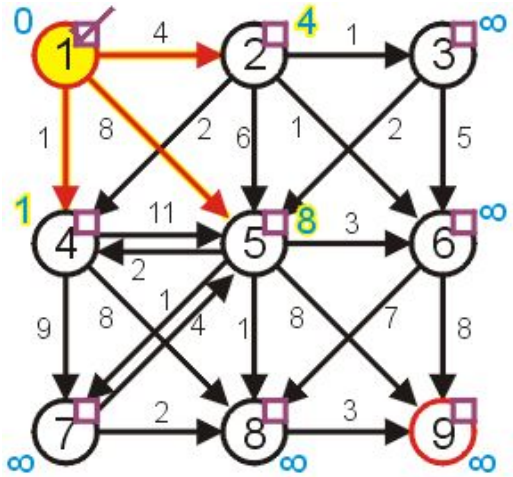
Example

- Consider the graph:
  - the distances are appropriately initialized
  - all vertices are marked as being unvisited



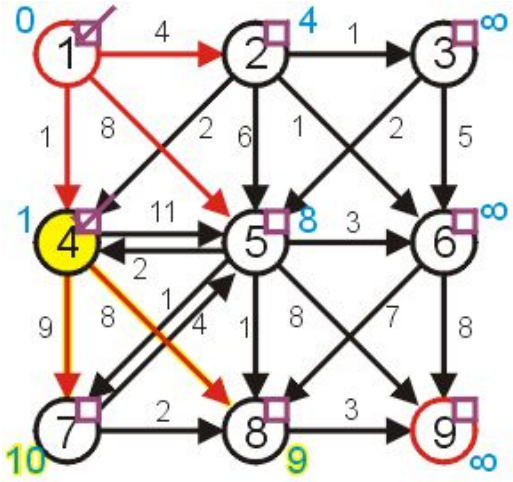
Example

- Visit vertex 1 and update its neighbours, marking it as visited
  - the shortest paths to 2, 4, and 5 are updated



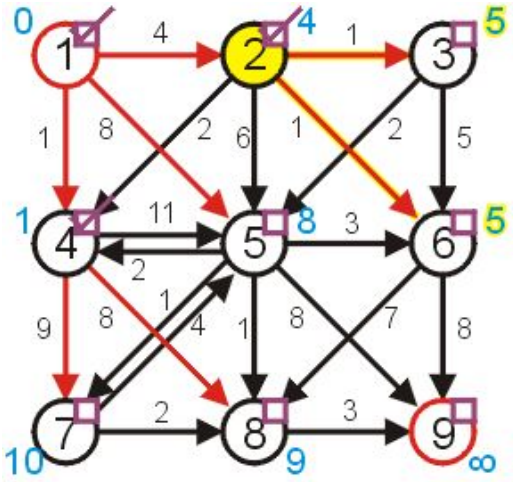
Example

- The next vertex we visit is vertex 4
  - vertex 5 1 + 11 ≥ 8 don't update
  - vertex 7 1 + 9 < ∞ update
  - vertex 8 1 + 8 < ∞ update



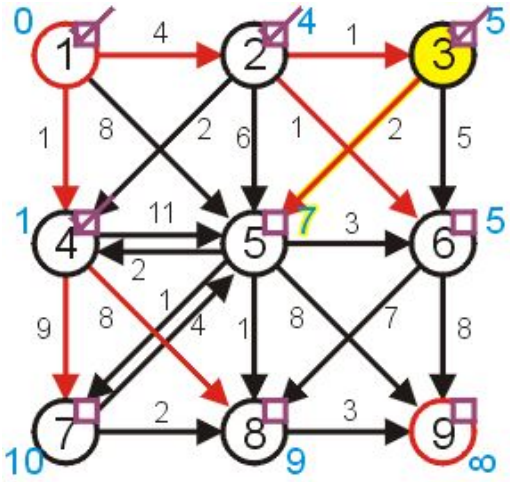
Example

- Next, visit vertex 2
  - vertex 3 $4 + 1 < \infty$ update
  - vertex 4 already visited
  - vertex 5 $4 + 6 \geq 8$ don't update
  - vertex 6 $4 + 1 < \infty$ update



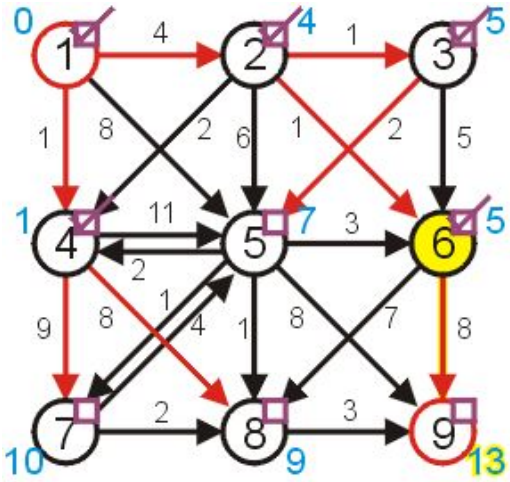
Example

- Next, we have a choice of either 3 or 6
- We will choose to visit 3
  - vertex 5 5 + 2 < 8 update
  - vertex 6 5 + 5 ≥ 5 don't update



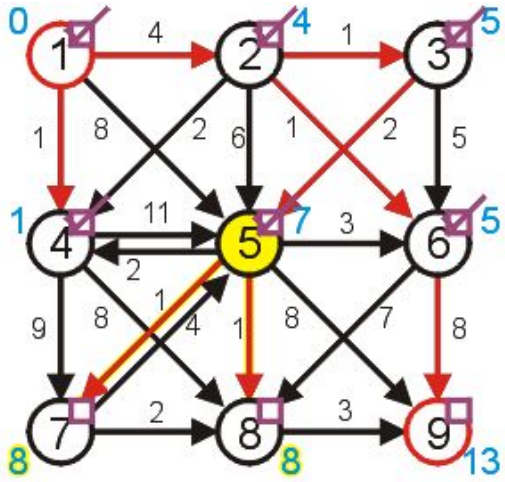
Example

- We then visit 6
  - vertex 8 $5 + 7 \geq 9$ don't update
  - vertex 9 $5 + 8 < \infty$ update



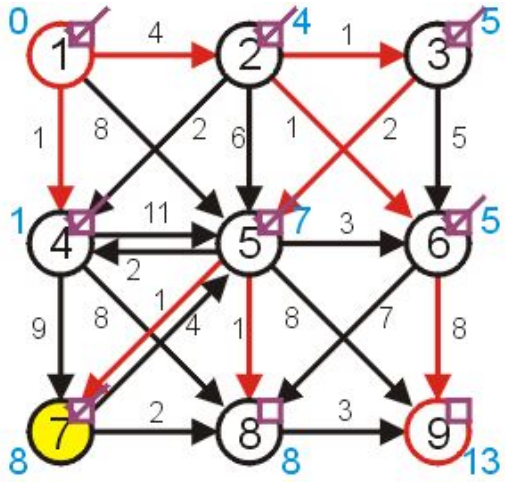
Example

- Next, we finally visit vertex 5:
  - vertices 4 and 6 have already been visited
  - vertex 7 $7 + 1 < 10$ update
  - vertex 8 $7 + 1 < 9$ update
  - vertex 9 $7 + 8 \geq 13$ don't update



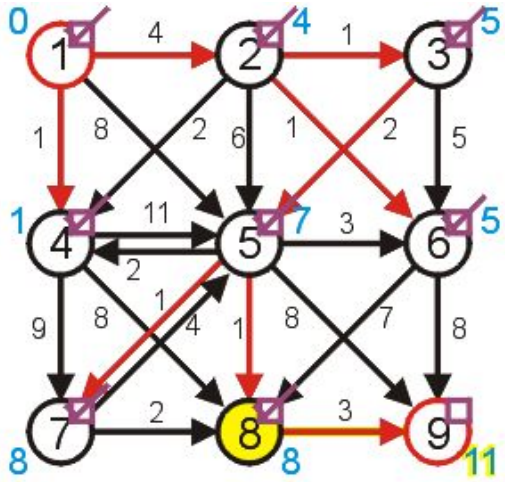
Example

- Given a choice between vertices 7 and 8, we choose vertex 7
  - vertices 5 has already been visited
  - vertex 8 8 + 2 ≥ 8 don't update



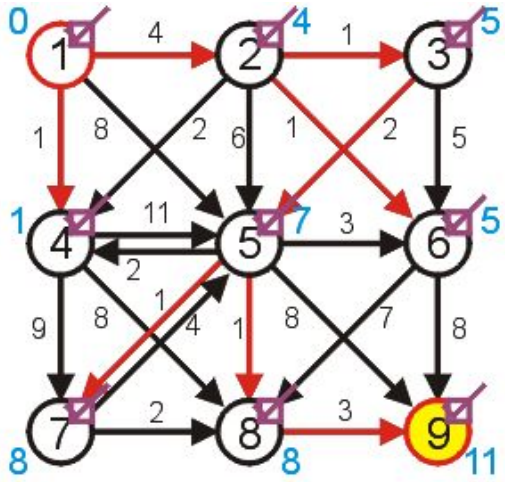
Example

- Next, we visit vertex 8:
  - vertex 9 8 + 3 < 13 update



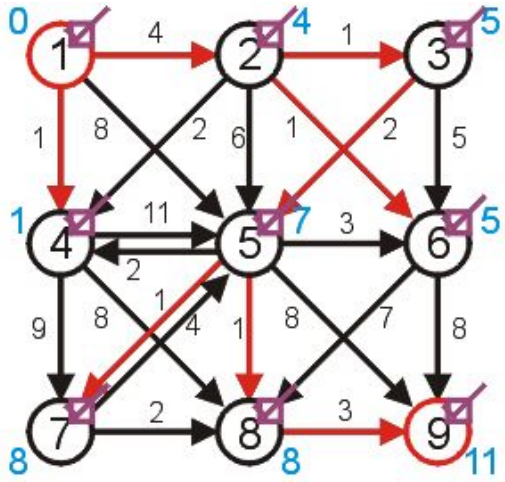
Example

- Finally, we visit the end vertex
- Therefore, the shortest path from 1 to 9 has length 11



Example

- We can find the shortest path by working back from the final vertex:
  - 9, 8, 5, 3, 2, 1
- Thus, the shortest path is (1, 2, 3, 5, 8, 9)



Dijkstra's algorithm

$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
    do $d[v] \leftarrow \infty$
$S \leftarrow \varnothing$
$Q \leftarrow V$ ▷ Q is a priority queue maintaining $V - S$
while $Q \neq \varnothing$
    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 $d[v] \leftarrow d[u] + w(u, v)$
                $p[v] \leftarrow u$

Dijkstra's algorithm

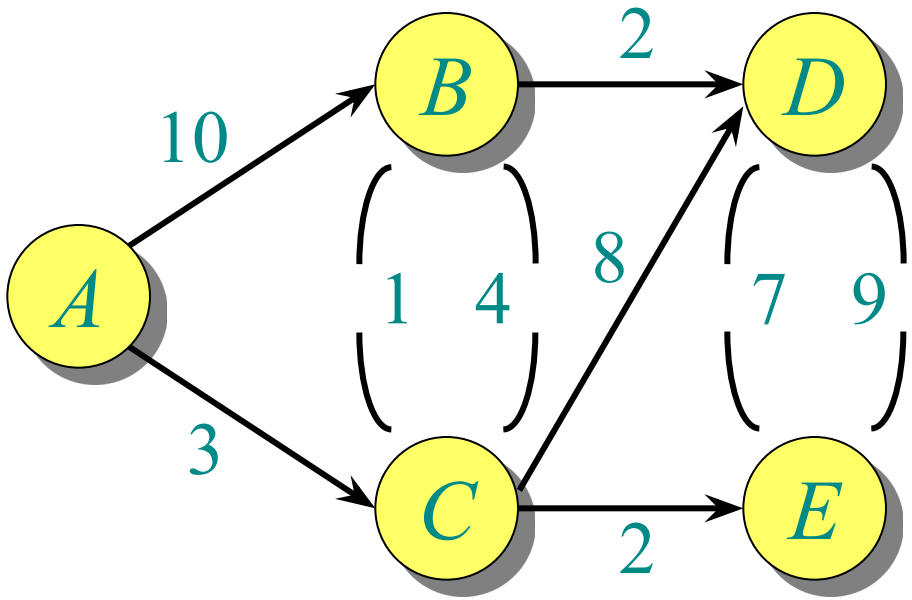
$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
  do $d[v] \leftarrow \infty$
$S \leftarrow \varnothing$
$Q \leftarrow V$ ▷ Q is a priority queue maintaining $V - S$
while $Q \neq \varnothing$
  do $u \leftarrow$ EXTRACT-MIN(Q)
    $S \leftarrow S \cup \{u\}$
    for each $v \in Adj[u]$
      do if $d[v] > d[u] + w(u, v)$
        then $d[v] \leftarrow d[u] + w(u, v)$
          $p[v] \leftarrow u$

relaxation step

Implicit DECREASE-KEY

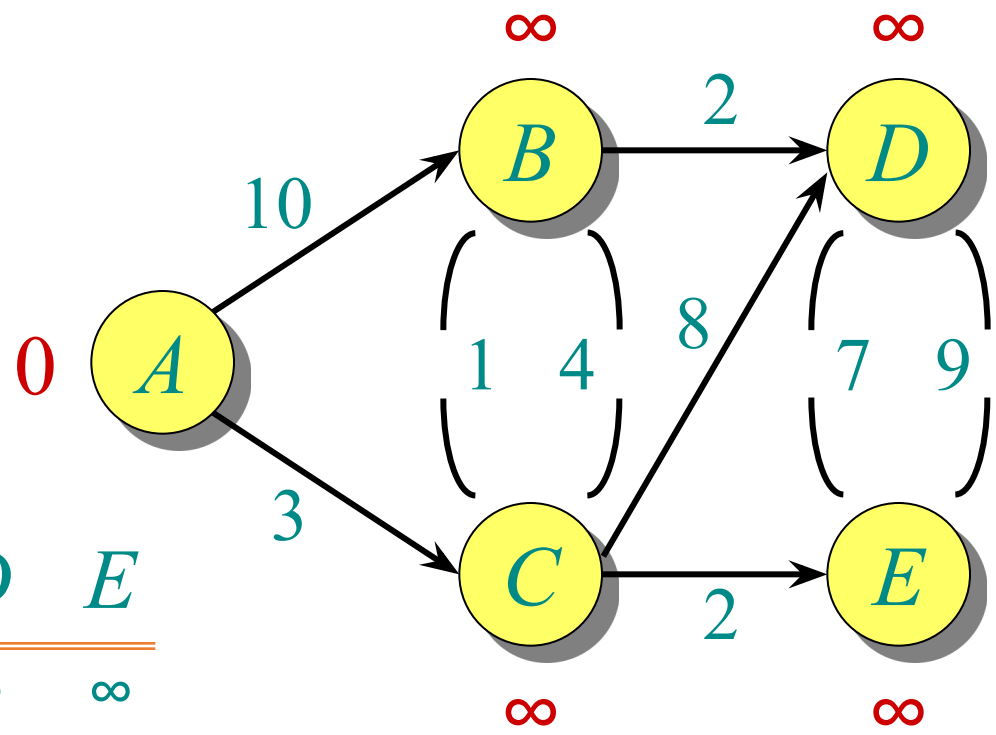
Example of Dijkstra's algorithm

Graph with nonnegative edge weights:



Example of Dijkstra's algorithm

Initialize:

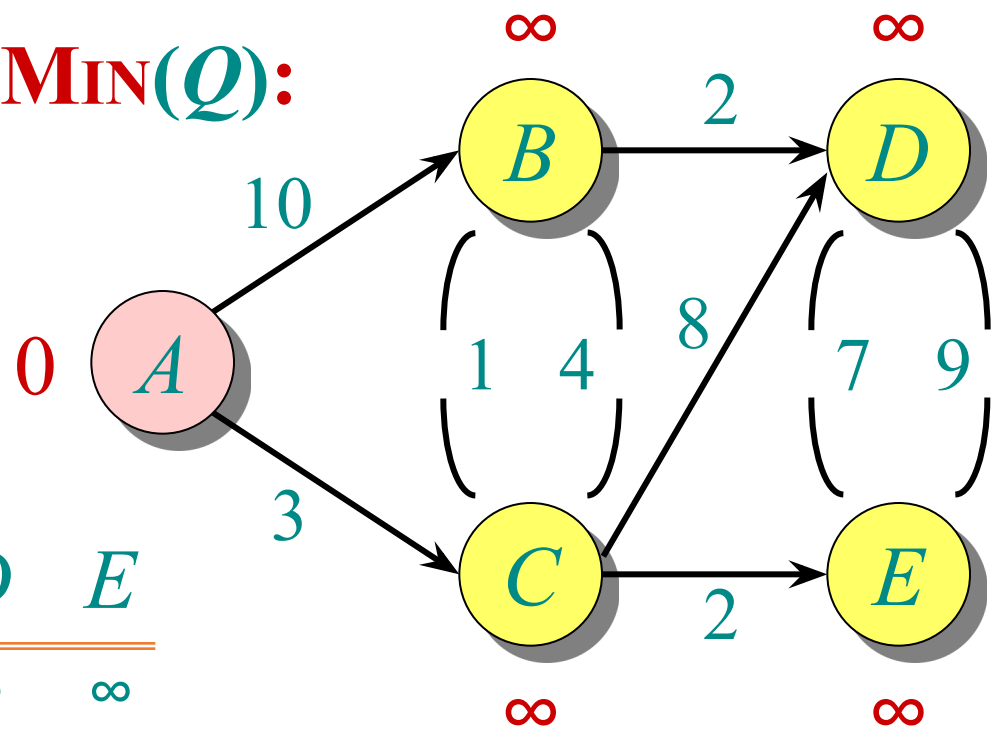


Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}

Example of Dijkstra's algorithm

"*A*" ← EXTRACT-MIN(*Q*):

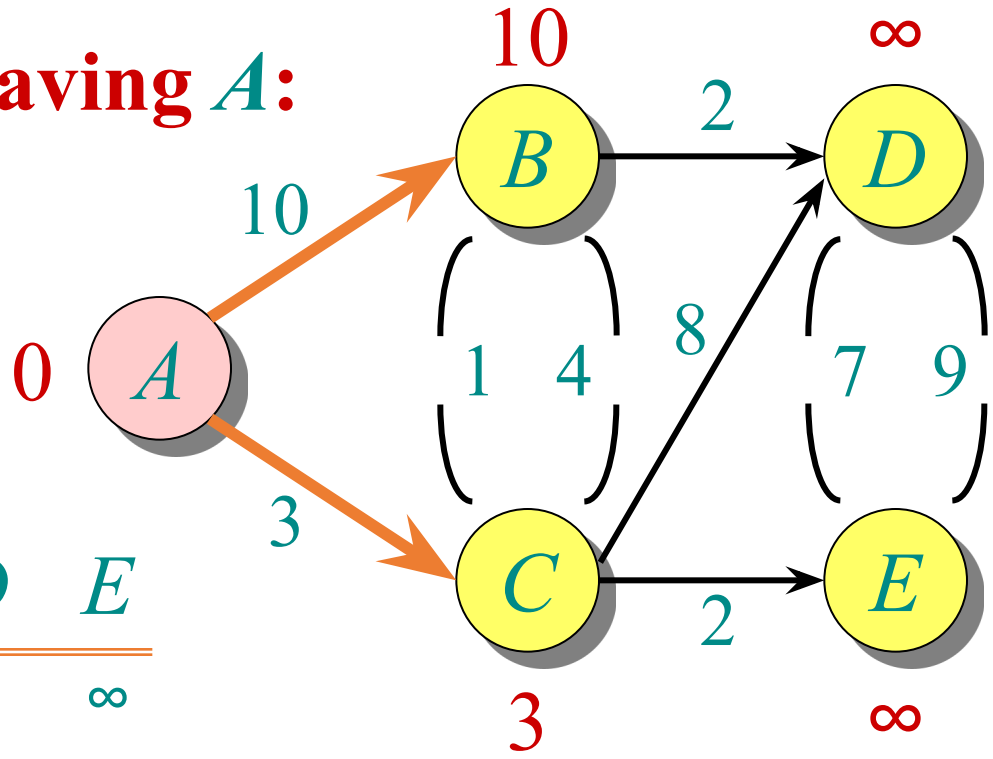


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞

S: { *A* }

Example of Dijkstra's algorithm

Relax all edges leaving *A*:

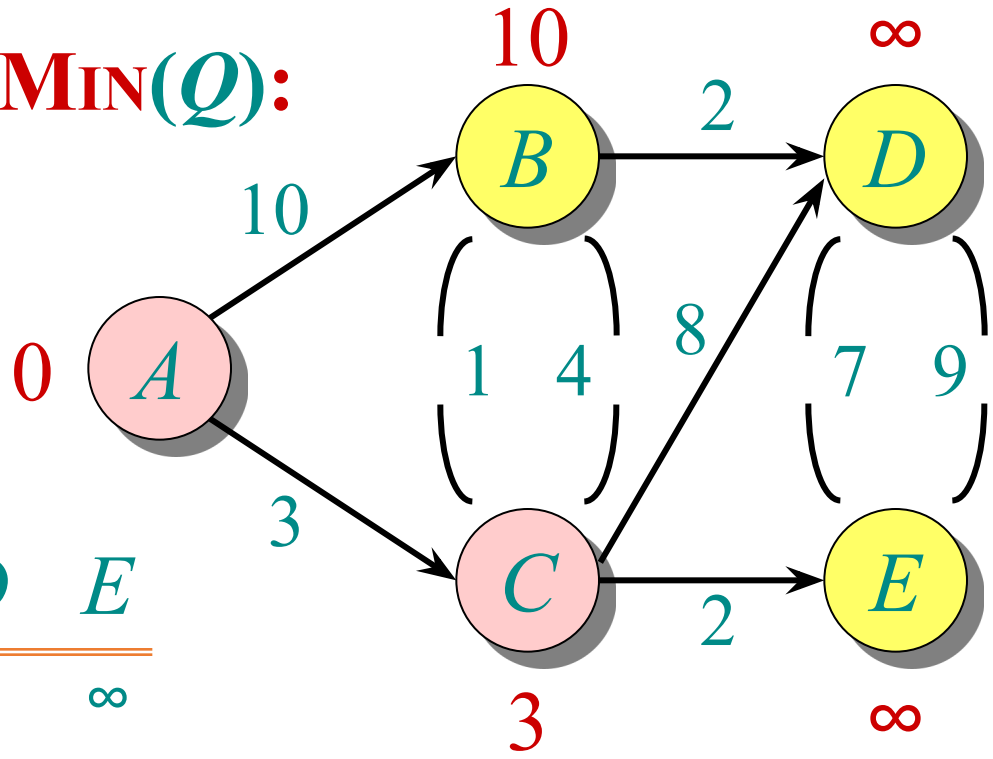


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { *A* }

Example of Dijkstra's algorithm

"*C*" ← EXTRACT-MIN(*Q*):

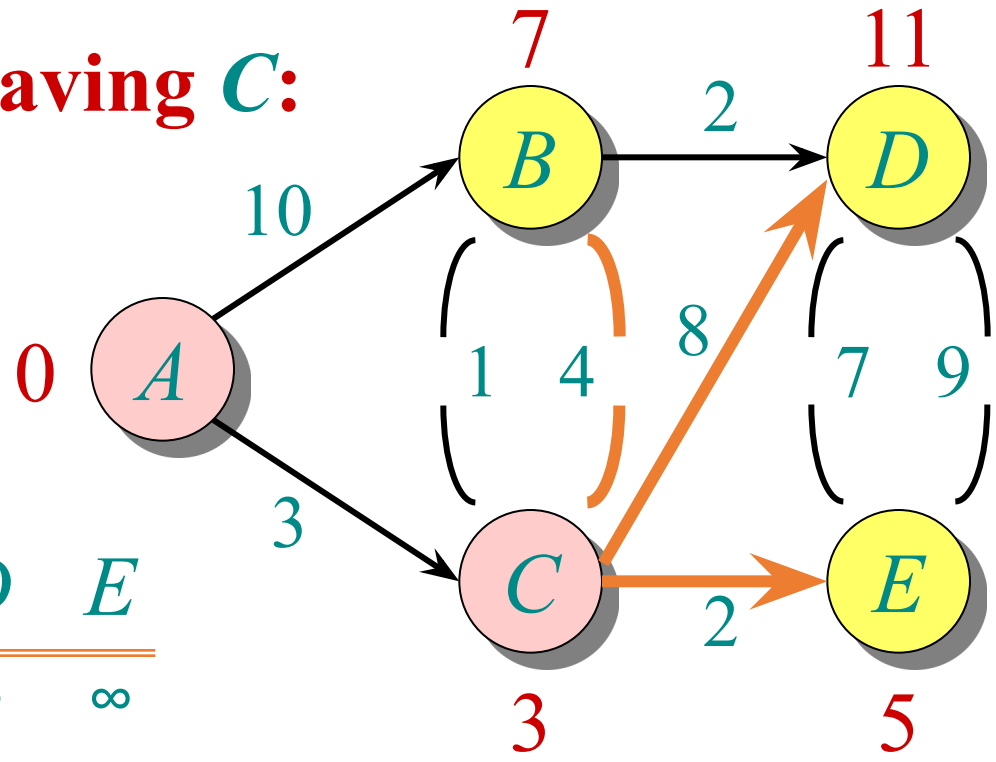


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { *A*, *C* }

Example of Dijkstra's algorithm

Relax all edges leaving *C*:

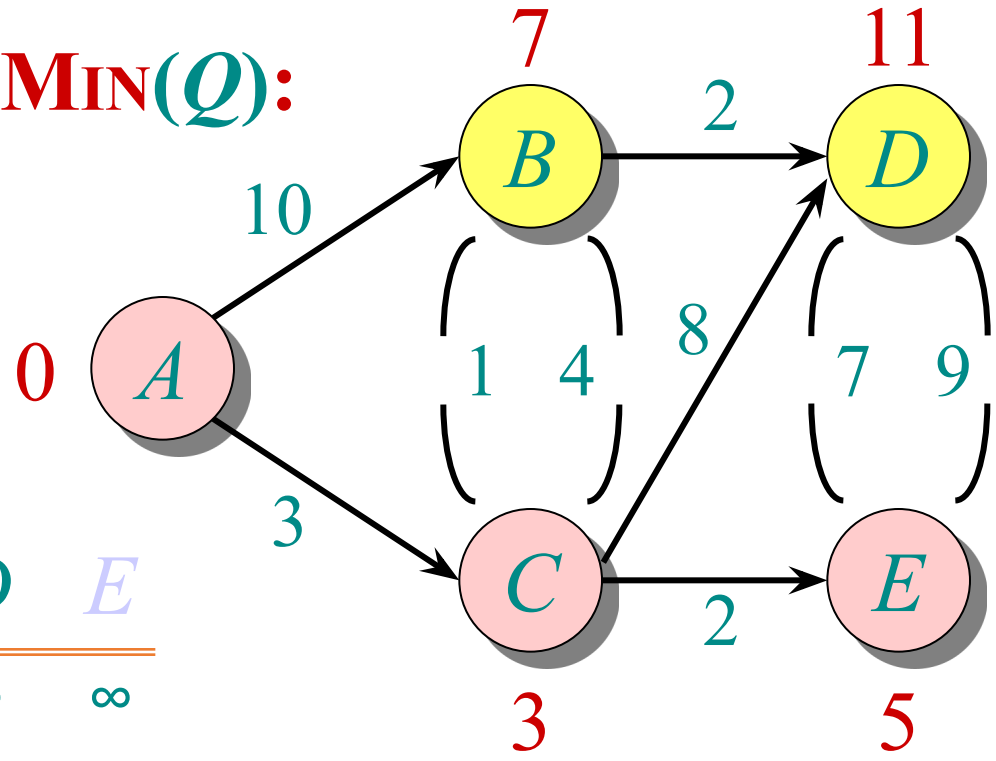


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C* }

Example of Dijkstra's algorithm

"*E*" ← EXTRACT-MIN(*Q*):

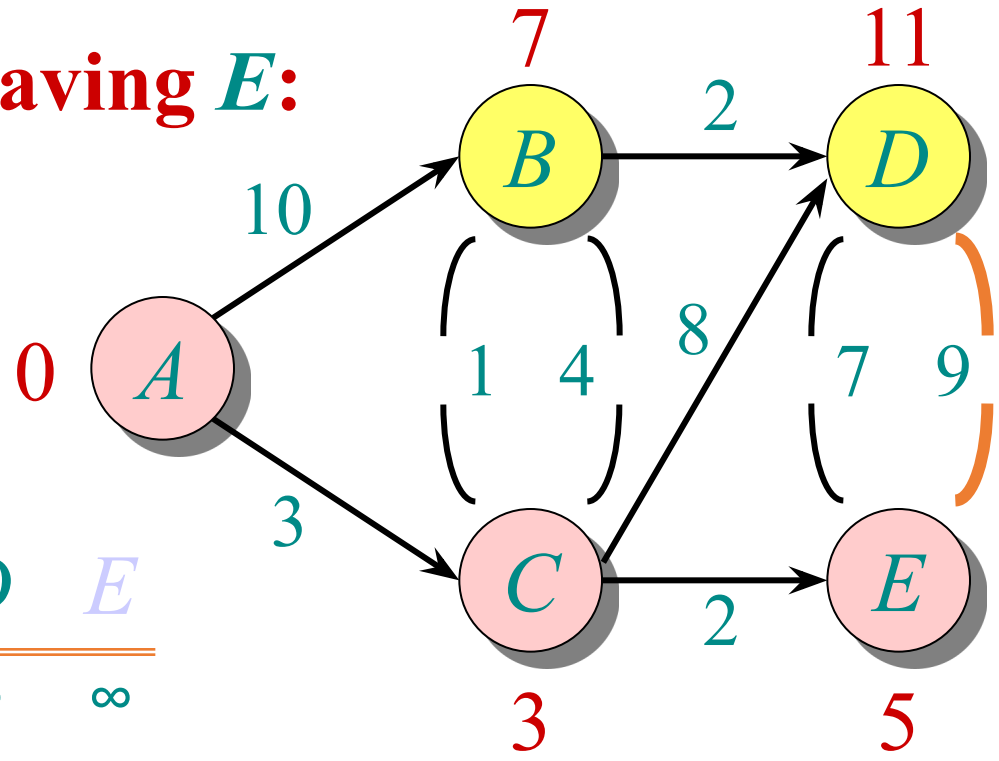


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C*, *E* }

Example of Dijkstra's algorithm

Relax all edges leaving *E*:

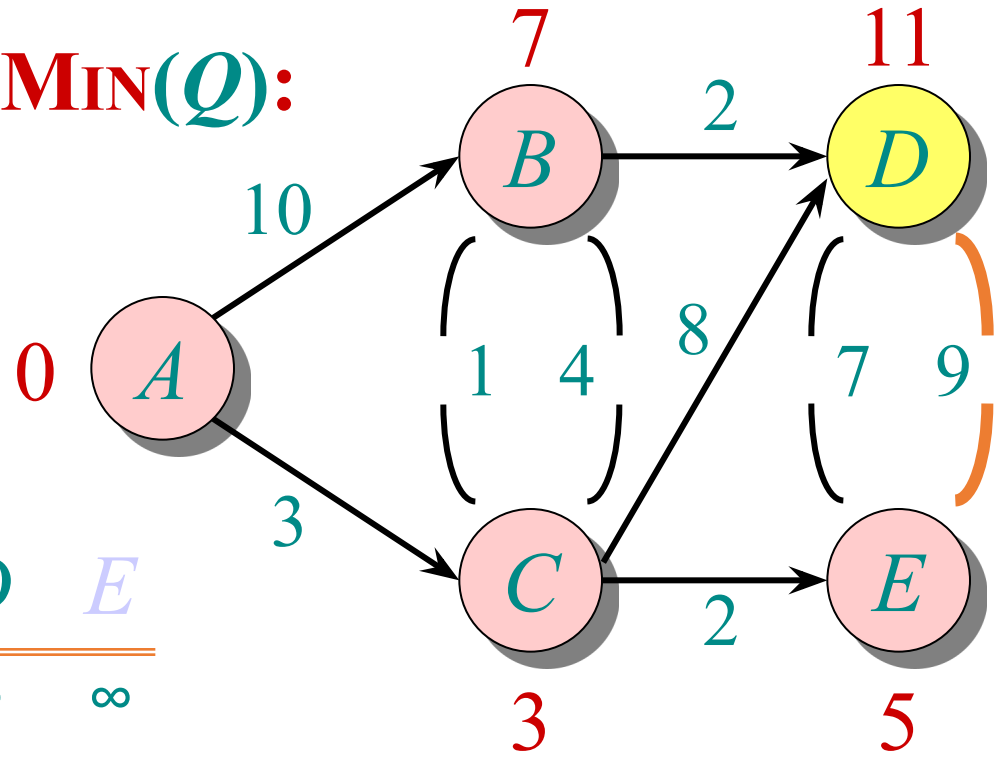


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { *A*, *C*, *E* }

Example of Dijkstra's algorithm

"*B*" ← EXTRACT-MIN(*Q*):

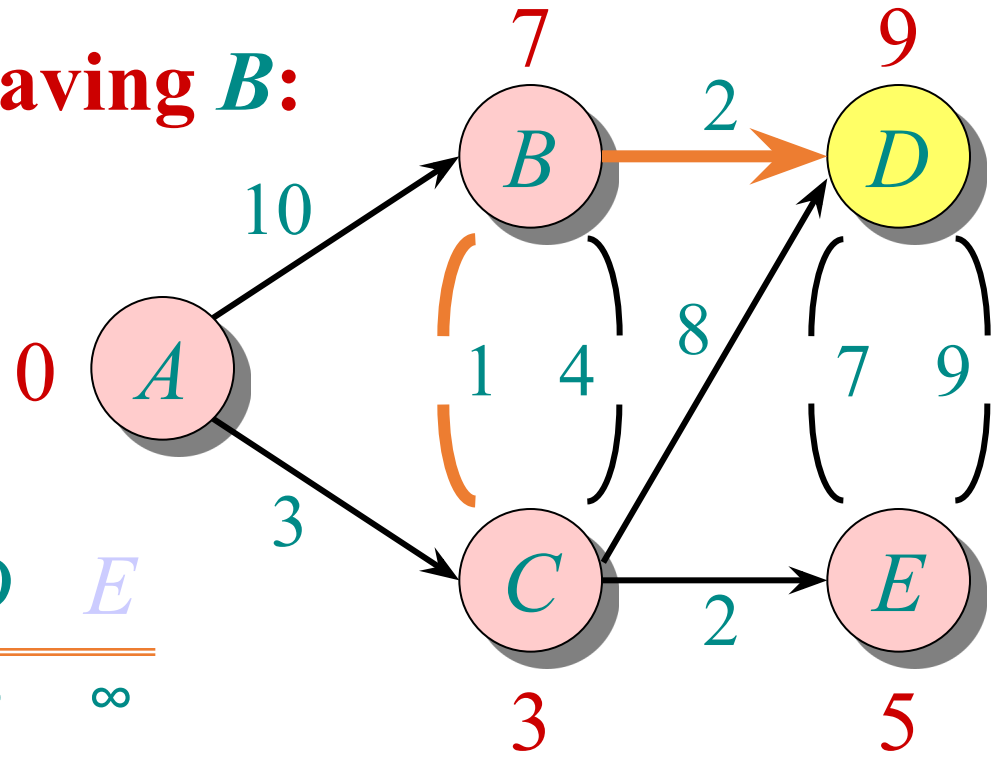


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

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

Example of Dijkstra's algorithm

Relax all edges leaving *B*:

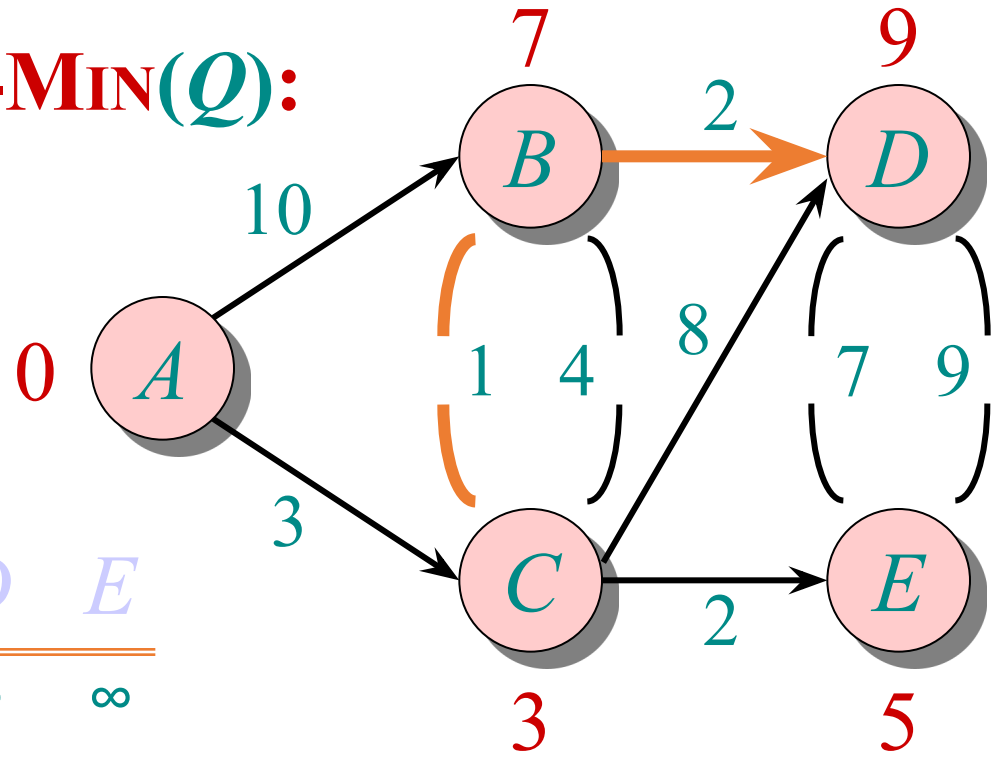


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

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

Example of Dijkstra's algorithm

"*D*" ← EXTRACT-MIN(*Q*):



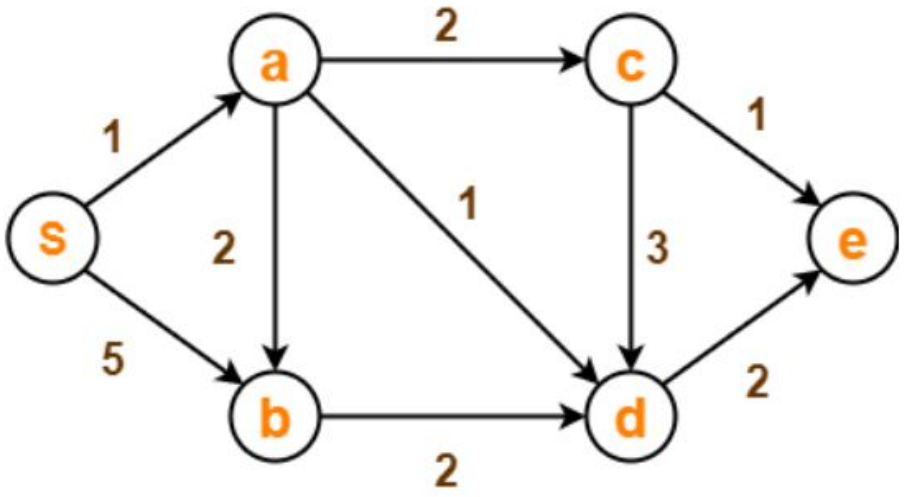
Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

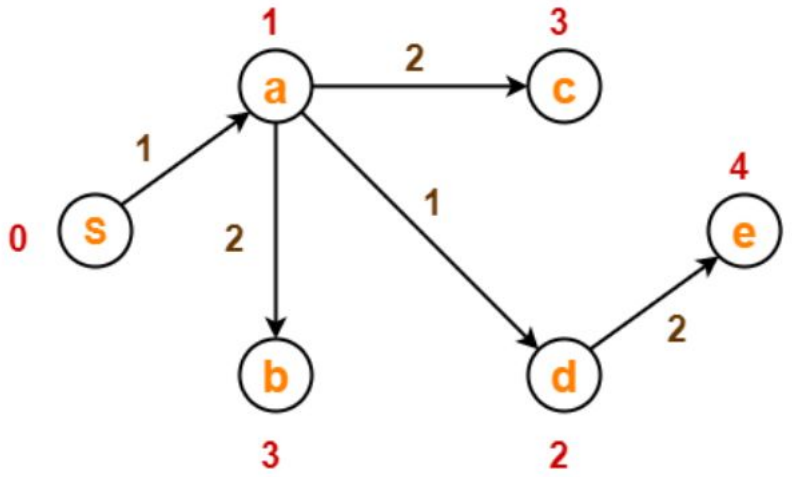
S: { *A*, *C*, *E*, *B*, *D* }

Summary

- Given a weighted directed graph, we can find the shortest distance between two vertices by:
  - starting with a trivial path containing the initial vertex
  - growing this path by always going to the next vertex which has the shortest current path

Example 2



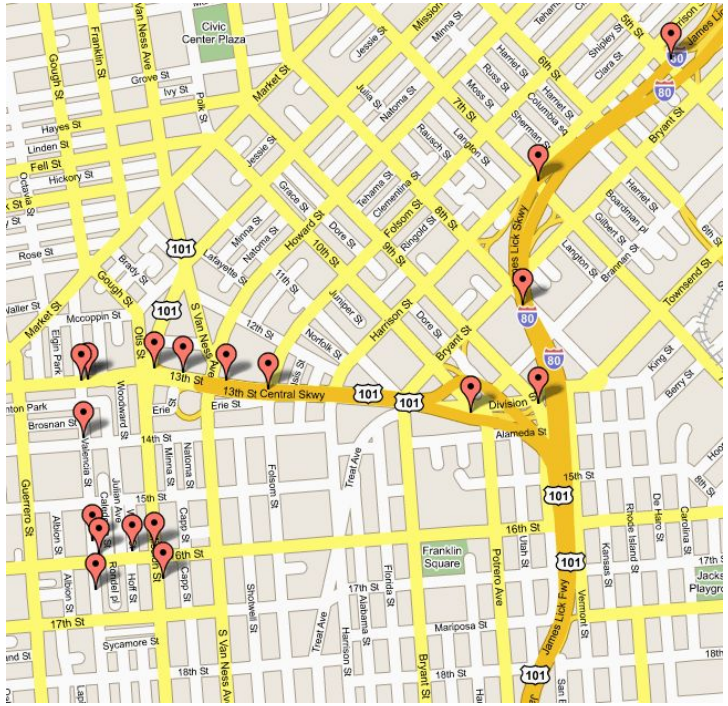


Dijkstra's Algorithm - Why use it?

- As mentioned, Dijkstra's algorithm calculates the shortest path to every vertex.
- However, it is about as computationally expensive to calculate the shortest path from vertex *u* to every vertex using Dijkstra's as it is to calculate the shortest path to some particular vertex *v*.
- Therefore, anytime we want to know the optimal path to some other vertex from a determined origin, we can use Dijkstra's algorithm.

Applications of Dijkstra's Algorithm

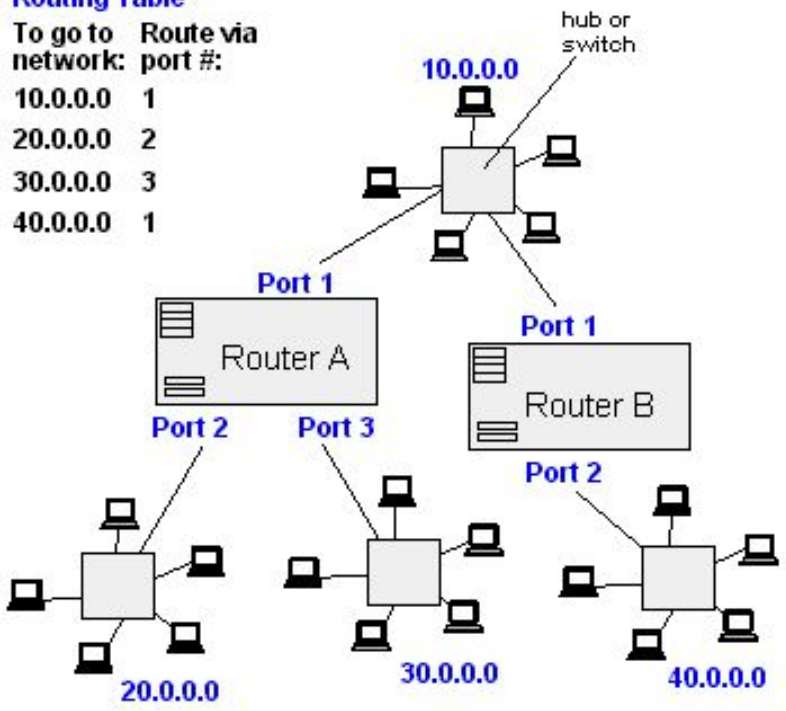
- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- Routing Systems



From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.

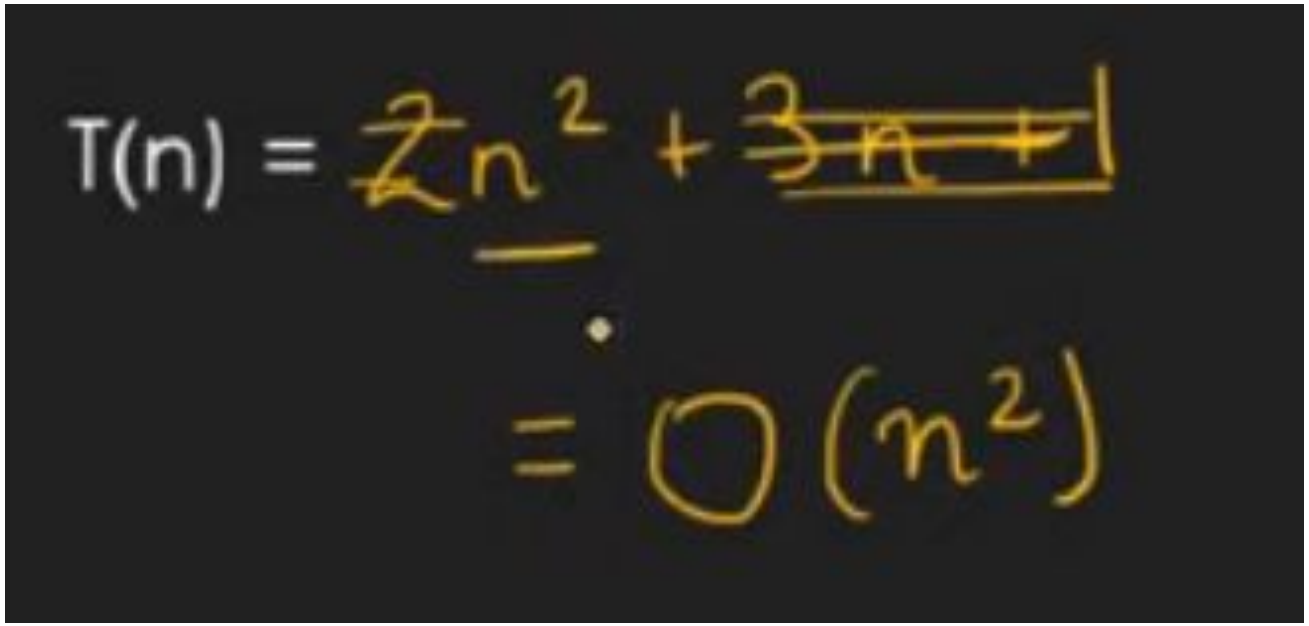
Router A
Routing Table

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



$$T(n) = 2n^2 + 3n + 1$$

- Drop lower order terms
- Drop all the constant multipliers



1) Loop

```
for( i = 1; i<=n; i++ ){
        x=y+z;
}
```

2) Nested Loop

```
for( i = 1; i<=n; i++ ){
        for( j = 1; j<=n; j++ ){
                x=y+z;
        }
}
```

```
for( i = 1; i<=n; i++ ){
        x=y+z; //constant time = c
}
```

$= cn$

$= O(n)$

```
for( i = 1; i<=n; i++ ){ //n times
        for( j = 1; j<=n; j++ ){ //n times
                x=y+z; //constant time
        }
}
```

$= O(n^2)$

4) If-else statements

```
if(condition)
{
    ___ O(n)
}

else
{
    ___ O(n²)
}
```

3) Sequential Statements

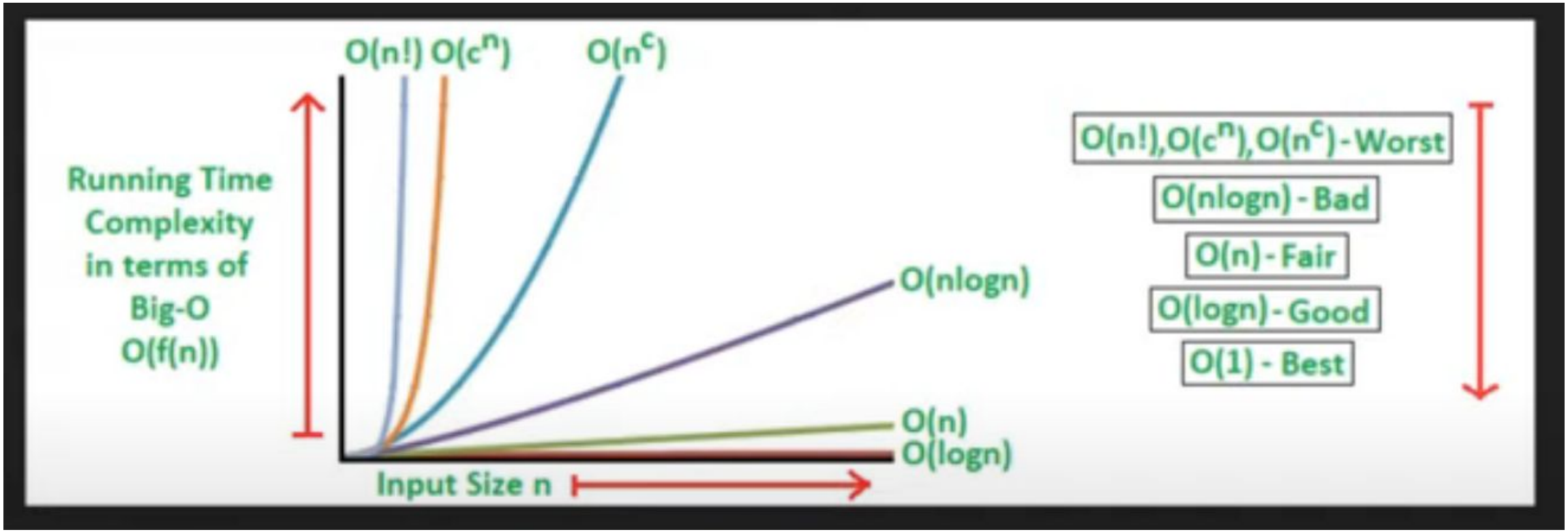
```
i) a = a + b; // constant time = c1

ii) for( i = 1; i<=n; i++ ){
        x = y + z;
    }                          c2n

iii) for( j = 1; j<=n; j++ ){
        c = d + e;
    }                          c3n
```

```
if(condition)
{
    ___ O(n)
}
                               = O(n²)
else
{
    ___ O(n²)
}
```

```
i) a = a + b; // constant time = c1

ii) for( i = 1; i<=n; i++ ){
        x = y + z;
    }                          c2n

iii) for( j = 1; j<=n; j++ ){
        c = d + e;
    }                          c3n
```

$= c_1 + c_2 n + c_3 n$

$= O(n)$



$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^c) < O(n!)$$

```
if(i > j ){
    j>23 ? cout<<j : cout<<i;
}
```

O(1)

```
for(i= 0 ; i < n; i++){
    cout<< i << " " ;
    i++;
}
```

O(n)

```
for(i= 0 ; i < n; i++){
    for(j = 0; j<n ;j++){
        cout<< i << " ";
    }
}
```

$O(n^2)$

```
int i = n;
while(i){
    cout << i << " ";
    i = i/2;
}
```

O(log n)

```
for(i= 0; i < n; i++){
    for(j = 1; j < n; j = j*2){
        cout << i << " ";
    }
}
```

O(n log n)

Algorithm	Time Complexity		
	Best	Average	Worst
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)
Radix Sort	Ω(nk)	θ(nk)	O(nk)

```
int a = 0;
for (i = 0; i < N; i++) {
        for (j = N; j > i; j--) {
                a = a + i + j;
        }
}
```

O(N*N)

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
        for (j = 2; j <= n; j = j * 2) {
                k = k + n / 2;
        }
}
```

O(nLogn)

```
int a = 0, i = N;
while (i > 0) {
        a += i;
        i /= 2;
}
```

O(log N)

```
function(int n)
{
        if (n==1)
        return;
        for (int i=1; i<=n; i++)
        {
                for (int j=1; j<=n; j++)
                {
                        printf("*");
                        break;
                }
        }
}
```

```
static void function(int n)
{
        int count = 0;
        for (int i = n / 2; i <= n; i++)
                for (int j = 1; j <= n; j = 2 * j)
                        for (int k = 1; k <= n; k = k * 2)
                                count++;
}
```

Algorithm classification

- Algorithms that use a similar problem-solving approach can be grouped together
- This classification scheme is neither exhaustive nor disjoint
- The purpose is not to be able to classify an algorithm as one type or another, but to highlight the various ways in which a problem can be attacked

A short list of categories

- Algorithm types we will consider include:
  - Simple recursive algorithms
  - Backtracking algorithms
  - Divide and conquer algorithms
  - Dynamic programming algorithms
  - Greedy algorithms
  - Branch and bound algorithms
  - Brute force algorithms
  - Randomized algorithms

Simple recursive algorithms I

- A simple recursive algorithm:
  - Solves the base cases directly
  - Recurs with a simpler subproblem
  - Does some extra work to convert the solution to the simpler subproblem into a solution to the given problem

Example recursive algorithms

- To count the number of elements in a list:
  - If the list is empty, return zero; otherwise,
  - Step past the first element, and count the remaining elements in the list
  - Add one to the result
- To test if a value occurs in a list:
  - If the list is empty, return false; otherwise,
  - If the first thing in the list is the given value, return true; otherwise
  - Step past the first element, and test whether the value occurs in the remainder of the list

Backtracking algorithms

- Backtracking algorithms are based on a depth-first recursive search
- A backtracking algorithm:
  - Tests to see if a solution has been found, and if so, returns it; otherwise
  - For each choice that can be made at this point,
    - Make that choice
    - Recur
    - If the recursion returns a solution, return it
  - If no choices remain, return failure

Divide and Conquer

- A divide and conquer algorithm consists of two parts:
  - Divide the problem into smaller subproblems of the same type, and solve these subproblems recursively
  - Combine the solutions to the subproblems into a solution to the original problem
- Traditionally, an algorithm is only called divide and conquer if it contains two or more recursive calls

Examples

- Quicksort:
  - Partition the array into two parts, and quicksort each of the parts
  - No additional work is required to combine the two sorted parts
- Mergesort:
  - Cut the array in half, and mergesort each half
  - Combine the two sorted arrays into a single sorted array by merging them

Dynamic programming algorithms

- A dynamic programming algorithm remembers past results and uses them to find new results
- Dynamic programming is generally used for optimization problems
  - Multiple solutions exist, need to find the “best” one
  - Requires “optimal substructure” and “overlapping subproblems”
    - Optimal substructure: Optimal solution contains optimal solutions to subproblems
    - Overlapping subproblems: Solutions to subproblems can be stored and reused in a bottom-up fashion
- This differs from Divide and Conquer, where subproblems generally need not overlap

Fibonacci numbers again

- To find the n^{th} Fibonacci number:
  - If n is zero or one, return one; otherwise,
  - Compute, *or look up in a table,* fibonacci(n-1) and fibonacci(n-2)
  - Find the sum of these two numbers
  - Store the result in a table and return it
- Since finding the n^{th} Fibonacci number involves finding all smaller Fibonacci numbers, the second recursive call has little work to do
- The table may be preserved and used again later

Greedy algorithms

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
- A "greedy algorithm" sometimes works well for optimization problems
- A greedy algorithm works in phases: At each phase:
  - You take the best you can get right now, without regard for future consequences
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum

Example: Counting money

- Suppose you want to count out a certain amount of money, using the fewest possible bills and coins
- A greedy algorithm would do this would be:
  At each step, take the largest possible bill or coin that does not overshoot
  - Example: To make $6.39, you can choose:
    - a $5 bill
    - a $1 bill, to make $6
    - a 25¢ coin, to make $6.25
    - A 10¢ coin, to make $6.35
    - four 1¢ coins, to make $6.39
- For US money, the greedy algorithm always gives the optimum solution

Brute force algorithm

- A brute force algorithm simply tries *all* possibilities until a satisfactory solution is found
  - Such an algorithm can be:
    - Optimizing: Find the *best* solution. This may require finding all solutions, or if a value for the best solution is known, it may stop when any best solution is found
      - Example: Finding the best path for a travelling salesman
    - Satisficing: Stop as soon as a solution is found that is *good enough*
      - Example: Finding a travelling salesman path that is within 10% of optimal

Randomized algorithms

- A randomized algorithm uses a random number at least once during the computation to make a decision
  - Example: In Quicksort, using a random number to choose a pivot
  - Example: Trying to factor a large prime by choosing random numbers as possible divisors