Data Structure

Dr. Itimad Raheem Ali

Email : itimadra@uoitc.edu.iq

Lecture 1

Assessment Description

Assessments:	Type of Assessment Description									
	Weighting	Theory					Practical			
Course Work	Total	T.1	T.2		Assig.	Attend.	T.1	T.2	Proj.	Attend.
	50	15	15		------	3	8	8	-----	1
Final	Total	Theory					Practical			
	50	33					17			
Total	100									

Data Structure

And its types

Data structure

It is a logical way to storing the data and it also define mechanism of retrieve data.

Data Structures

- "Once you succeed in writing the programs for complicated algorithms, they usually run extremely fast. The computer doesn't need to understand the algorithm, it’s task is only to run the programs.“
- There are a number of facts to good programs: they must
  - run correctly
  - run efficiently
  - be easy to read and understand
  - be easy to debug *and*
  - be easy to modify.

Data Structure (Cont.)

What is Data Structure ?

- A scheme for organizing related pieces of information
- A way in which sets of data are organized in a particular system
- An organized aggregate of data items
- A computer interpretable format used for storing, accessing, transferring and archiving data
- The way data is organized to ensure efficient processing: this may be in lists, arrays, stacks, queues or trees

Data structure is a specialized format for organizing and storing data so that it can be accessed and worked with in appropriate ways to make an a program efficient

Data Structures (Cont.)

- Data Structure = Organized Data + Allowed Operations

There are two design aspects to every data structure:

the interface part

The publicly accessible functions of the type. Functions like creation and destruction of the object, inserting and removing elements (if it is a container), assigning values etc.

the implementation part :

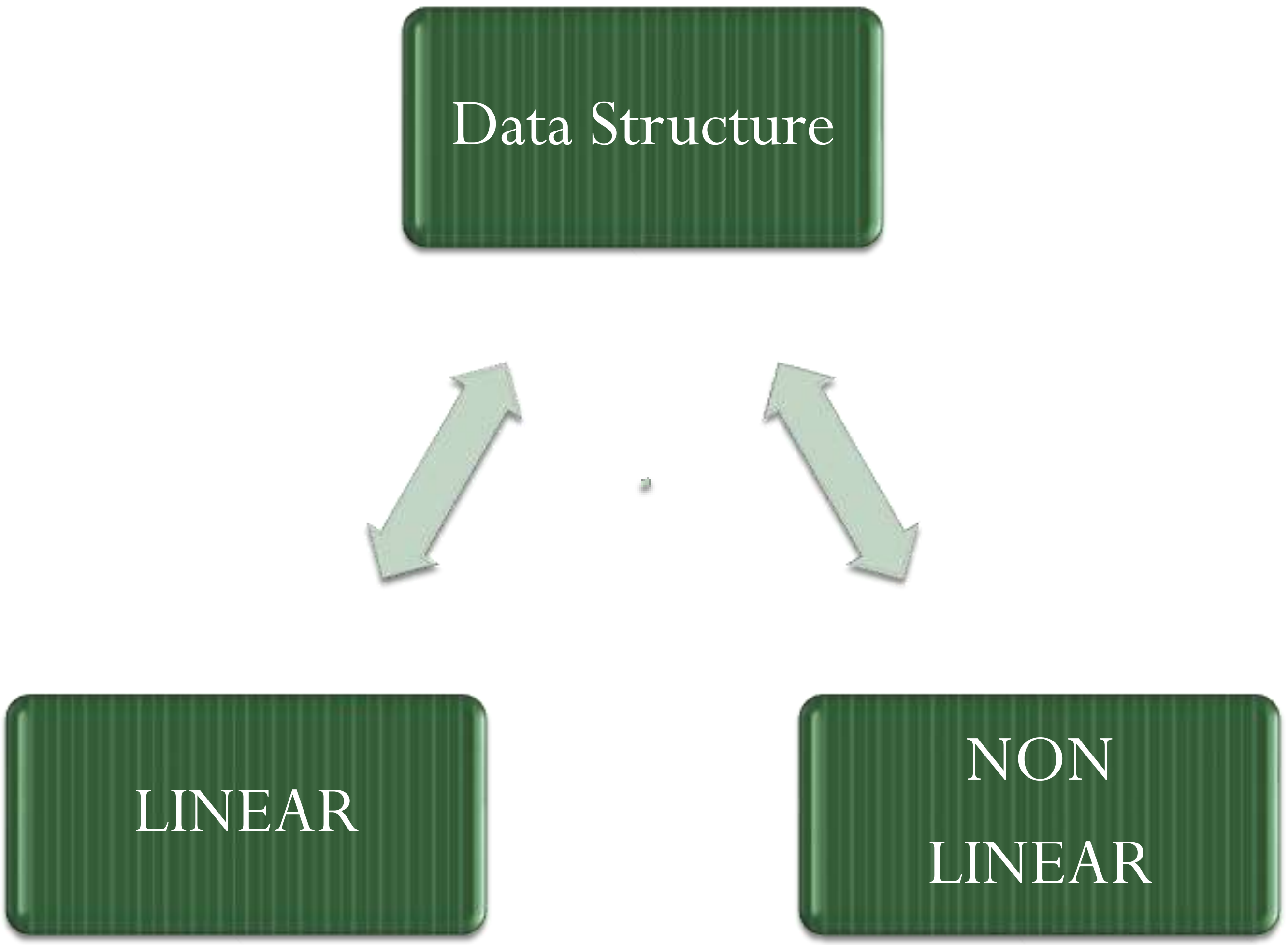
Internal implementation should be independent of the interface. Therefore, the details of the implementation aspect should be hidden out from the users.

Collections

•Programs often deal with collections of items.
•These collections may be organized in many ways and use many different program structures to represent them, yet, from an abstract point of view, there will be a few common operations on any collection.

create	Create a new collection
add	Add an item to a collection
delete	Delete an item from a collection
find	Find an item matching some criterion in the collection
destroy	Destroy the collection

Types of Data Structure



Data Structure Operation

Traversing: Accessing each record exactly once so that certain item in the record may be processed.

Searching: finding the location of the record with a given key value .

Insertion : add a new record to the structure

Deletion : removing a record from the structure

Linear Data Structure

1. **Array**
2. **Stack**
3. **Queue**
4. **Linked List**

Array

:-An array is a collection of homogeneous type of data elements.

:-An array is consisting of a collection of elements .

Operation Performed On Array

1.Traversing

2.Search

3.Insertion

4.Deletion

5.Sorting

6.Merging

Representation of Array

AAA	**1**
BBB	2
CCC	3
DDD	4
EEE	5

Array Representation

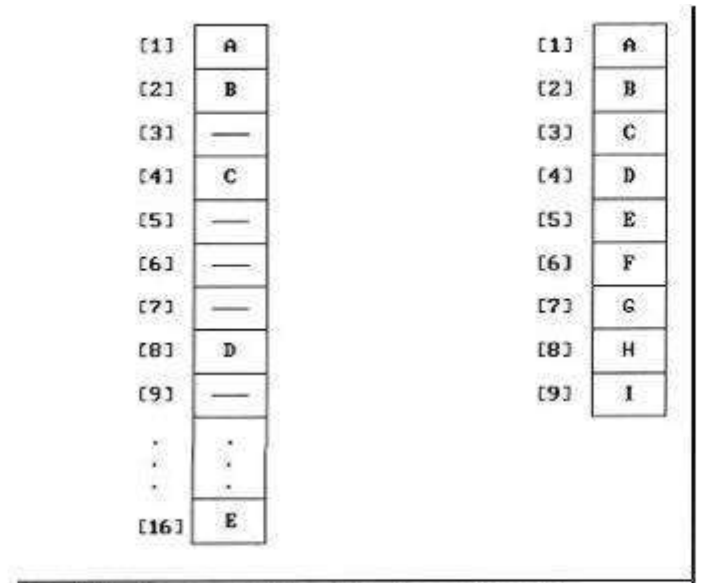


Figure 5.11: Array representation of binary trees of Figure 5.9

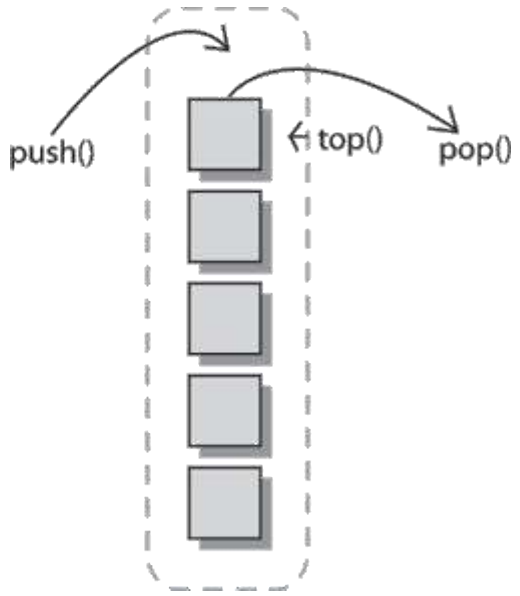
Stack

A Stack is a list of elements in which an element may be inserted or deleted at one end which is known as TOP of the stack.

Representation of Stack

EEE TOP
DDD
CCC
BBB
AAA

Stack Representation



Queue

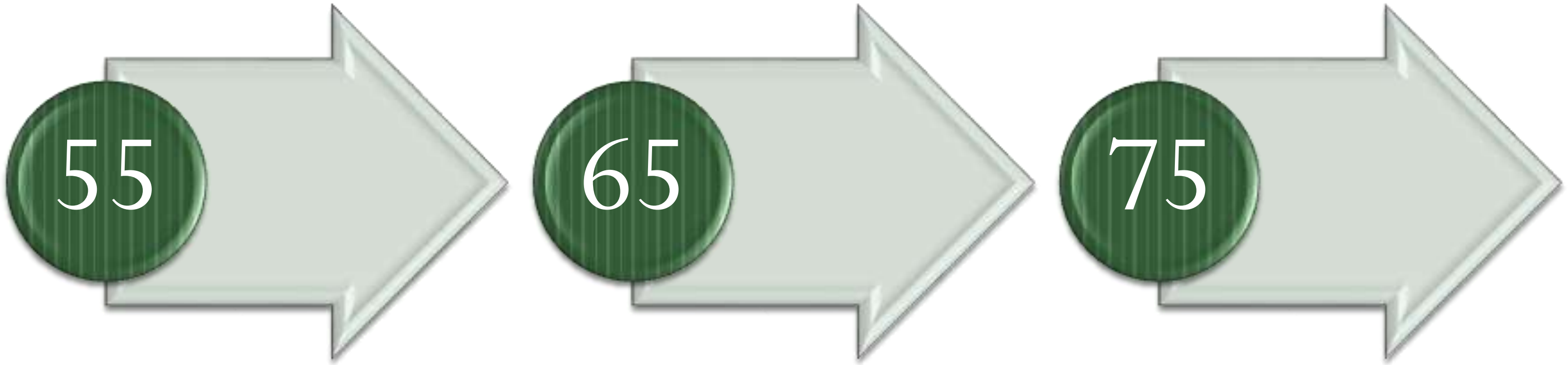
A queue is a linear list of element in which insertion can be done at one end which is known as front and deletion can be done which is known as rear.

Operation Performed On Queue

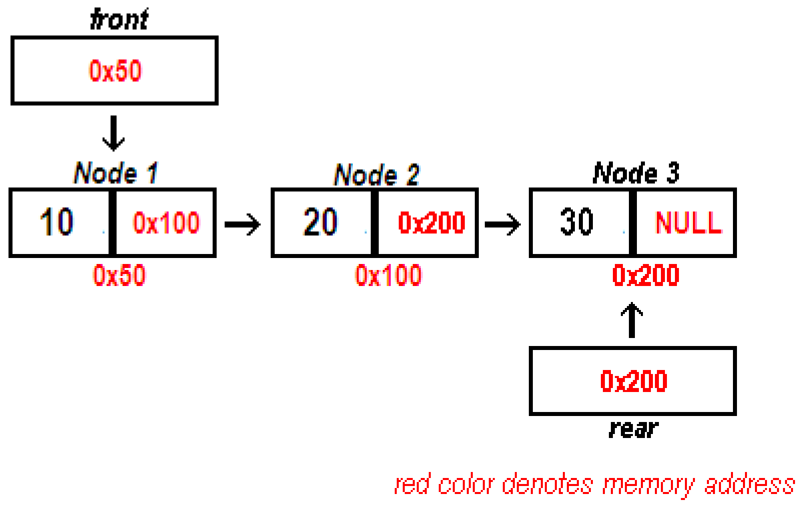
Insertion : Add A New Element In Queue

Deletion: Removing An Element In Queue

Representation of Queue



Queue Representation



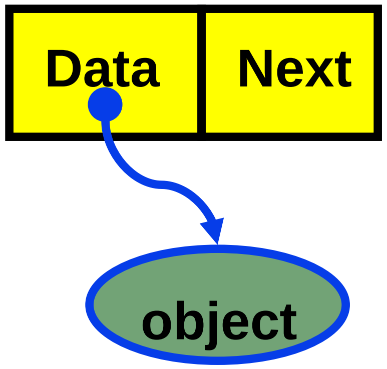
Linked Lists

- **The linked list is a very flexible dynamic data structure: items may be added to it or deleted from it at will**
  - Dynamically allocate space for each element as needed
  - Include a pointer to the next item
  - the number of items that may be added to a list is limited only by the amount of memory available

Linked list can be perceived as connected (linked) nodes

Each node of the list contains

- the data item
- a pointer to the next node
- The last node in the list contains a NULL pointer to indicate that it is the end or *tail* of the list.



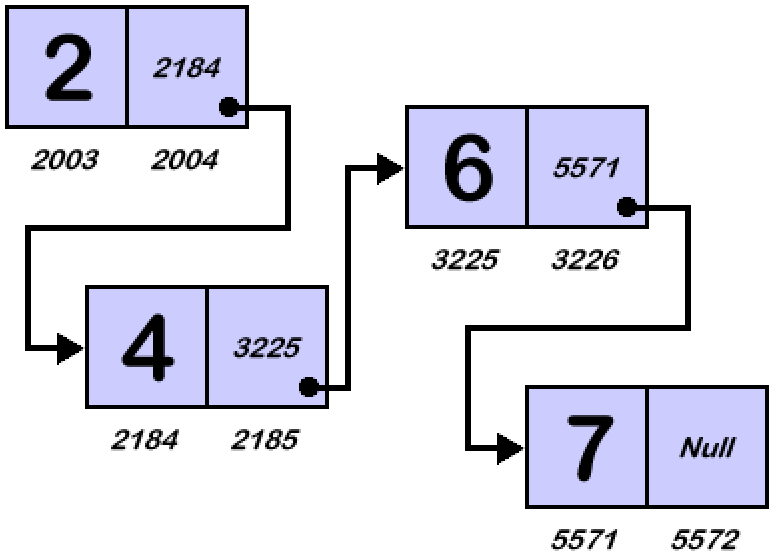
Linked List

A Linked list is a linear collection of data elements .It has two part one is info and other is link part.info part gives information and link part is address of next node

Operation Performed on Linked List

1. **Traversing**
2. **Searching**
3. **Insertion**
4. **Deletion**

Linked Representation



2.Non Linear

1. Tree
2. Graph

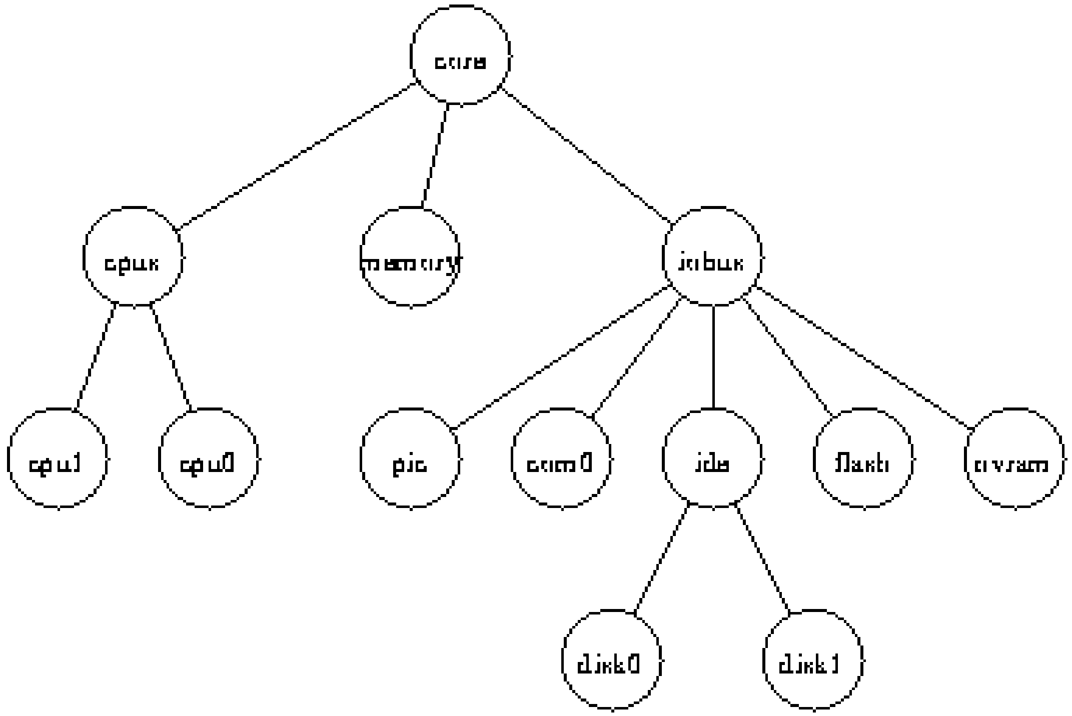
Tree

*In computer science, a **tree** is a widely-used data structure that emulates a hierarchical tree structure with a set of linked nodes.*

Operation On Tree

1. Insertion
2. Deletion
3. Searching

Tree Representation



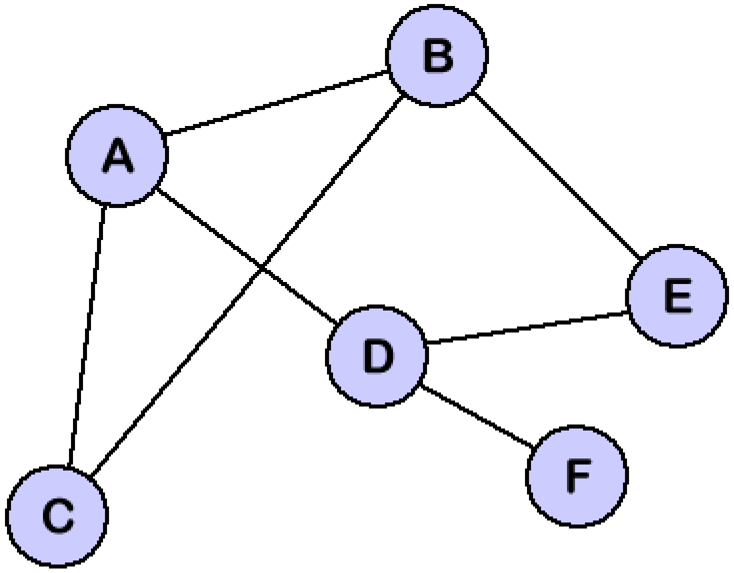
Graph

A graph data structure may also associate to each edge some edge value, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).

Operation Performed Graph

1. Searching
2. Insertion
3. Deletion

Graph Representation



Data Structure

Dr. Itimad Raheem Ali

Email : itimadra@uoitc.edu.iq

Lecture 2

Variables in Python

- Variables need not be declared first in python. They can be used directly. Variables in python are case sensitive as most of the other programming languages.
  Example:

```
a = 3
A = 4
print a
print A
```

The output is :

```
3  4
```

Expressions in Python

- Arithmetic operations in python can be performed by using arithmetic operators and some of the in-built functions.

```
a = 2
b = 3
c = a + b
print c
d = a * b
print d
```

The output is :

```
    5  6
```

Strings in Python

A string is a sequence of characters. It can be declared in python by using double-quotes. Strings are immutable, i.e., they cannot be changed.

```
# Assigning string to a variable
a = "This is a string"
print (a)
```

```
Output:
This is a string
```

Lists in Python

Lists are one of the most powerful tools in python. They are just like the arrays declared in other languages. But the most powerful thing is that list need not be always homogeneous. A single list can contain strings, integers, as well as objects. Lists can also be used for implementing stacks and queues. Lists are mutable, i.e., they can be altered once declared.

```
# Declaring a list
L = [1, "a" , "string" , 1+2]
print L
L.append(6)
print L
L.pop()
print L
print L[1]
```

```
The output is :
[1, 'a', 'string', 3]
[1, 'a', 'string', 3, 6]
[1, 'a', 'string', 3]
 a
```

Iterations in Python

Iterations or looping can be performed in python by ‘for’ and ‘while’ loops. Apart from iterating upon a particular condition, we can also iterate on strings, lists, and tuples.

Example 1: Iteration by while loop for a condition

```
i = 1
while (i <10):
        print(i)
        i += 1
```

The output is :

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

Example 2: Iteration by for loop on string

```
s = "Hello World"
  for i in s :
        print (i)
```

The output is :

```
H e l l o W o r l d
```

Example 3: Iteration by for loop on list

```
L = [1, 4, 5, 7, 8, 9]
for i in L:
        print (i)
```

The output is :

```
1 4 5 7 8 9
```

Example 4 : Iteration by for loop for range

```
for i in range(0, 10):
        print (i)
```

The output is :

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

Conditions in Python

- Conditional output in python can be obtained by using if-else and elif (else if) statements.

```
a = 3
b = 9
if b % a == 0 :
        print "b is divisible by a"
elif b + 1 == 10:
        print "Increment in b produces 10"
else:
        print "You are in else statement"
```

The output is :

```
b is divisible by a
```

Functions in Python

A function in python is declared by the keyword 'def' before the name of the function. The return type of the function need not be specified explicitly in python. The function can be invoked by writing the function name followed by the parameter list in the brackets.

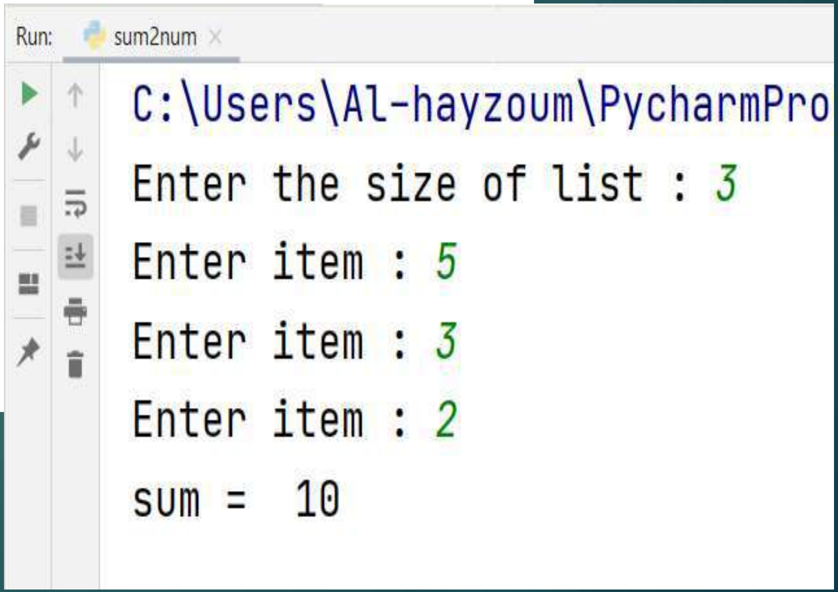
```
# Function for checking the divisibility
# Notice the indentation after function declaration
# and if and else statements
def checkDivisibility(a, b):
        if a % b == 0 :
                print "a is divisible by b"
        else:
                print "a is not divisible by b"
#Driver program to test the above function
checkDivisibility(4, 2)
```

The output is :

```
a is divisible by b
```

Ex: Write a program to read and write integer no.s and then find the summation of it. **By using functions**

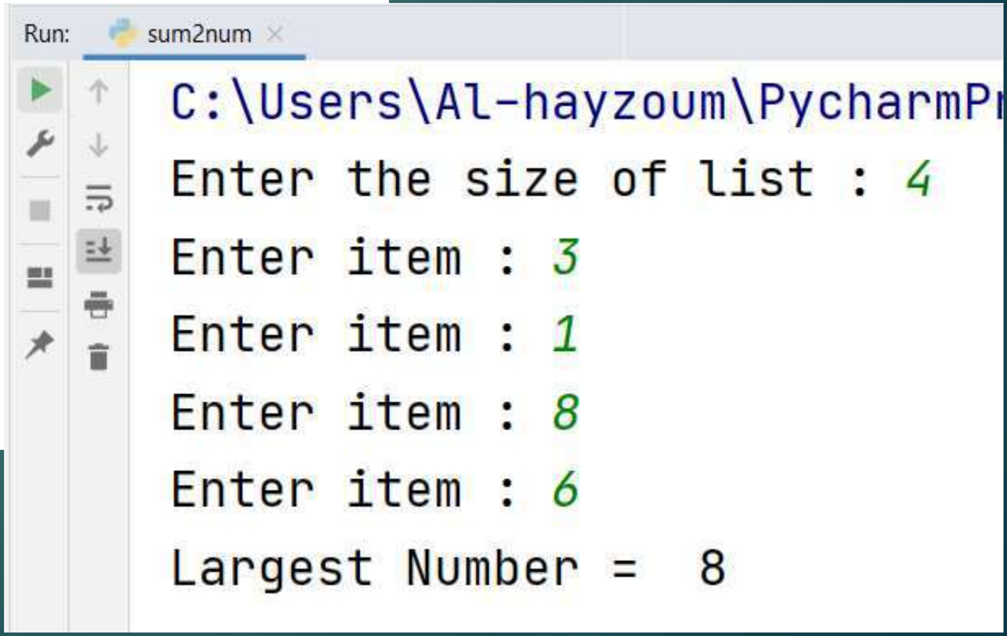
```python
def read(ls = []):
    size = int(input('Enter the size of list : '))
    for i in range(size):
        x = int(input('Enter item : '))
        ls.append(x)
    return ls
def sum(ls):
    s = 0
    for i in ls:
        s += i
    return s
ls=read()
s=sum(ls)
print('sum = ',s)
```



```
C:\Users\Al-hayzoum\PycharmPro
Enter the size of list : 3
Enter item : 5
Enter item : 3
Enter item : 2
sum =  10
```

Example 3: - Write a python program to get largest number from a list using functions.

```
def read(ls = []):
    size = int(input('Enter the size of list : '))
    for i in range(0,size,1):
        x = int(input('Enter item : '))
        ls.append(x)
    return ls
def max(ls):
    Max = ls[0]
    for i in range(1, len(ls), 1):
        if ls[i] > Max:
            Max = ls[i]
    return Max
ls=read()
M=max(ls)
print ('Largest Number = ',M)
```



```
C:\Users\Al-hayzoum\PycharmP
Enter the size of list : 4
Enter item : 3
Enter item : 1
Enter item : 8
Enter item : 6
Largest Number =  8
```

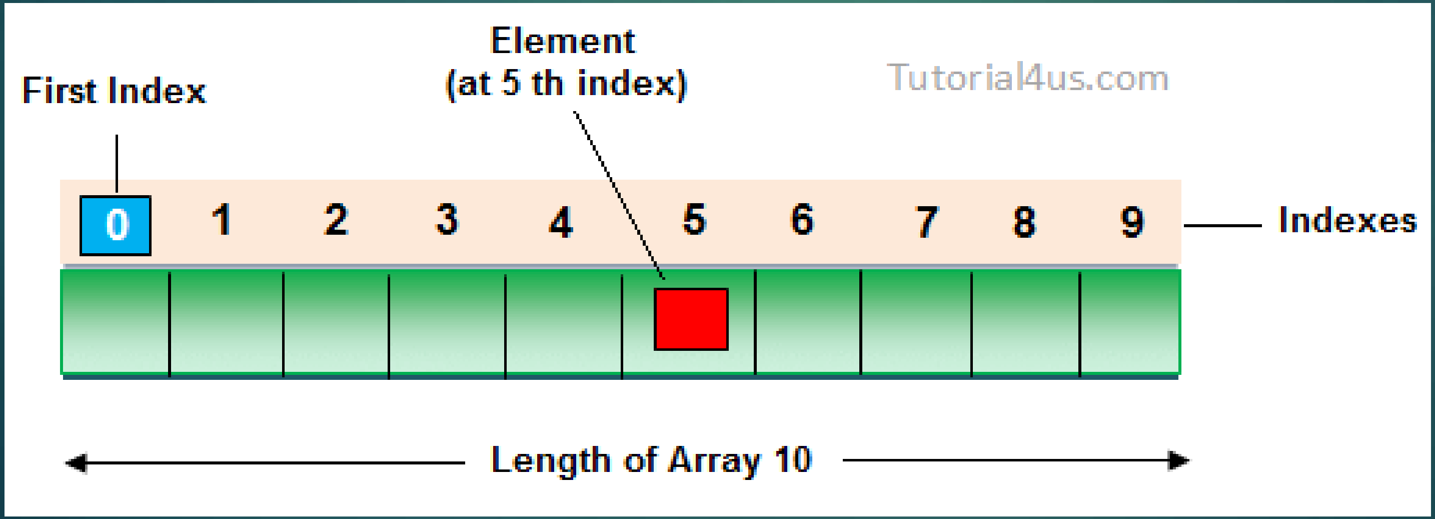
Arrays

- Introducing Arrays
- Declaring Array Variables, Creating Arrays, and Initializing Arrays
- Adding element to Arrays
- Multidimensional Arrays
- Search and Sorting Methods

Introducing Arrays

Array is a data structure that represents a collection of the same types of data.

An Array of 10 Elements



Creating a Array

Array in Python can be created by importing `array` module. ***`array(data_type, value_list)`*** is used to create an array with data type and value list specified in its arguments.

```
# Python program to demonstrate
# Creation of Array importing "array" for array creations
import array as arr
# creating an array with integer type
a = arr.array('i', [1, 2, 3])

# printing original array
print ("The new created array is : ", end =" ")
for i in range (0, 3):
        print (a[i], end =" ")
print()

# creating an array with float type
b = arr.array('d', [2.5, 3.2, 3.3])

# printing original array
print ("The new created array is : ", end =" ")
for i in range (0, 3):
        print (b[i], end =" ")
```

Output :

```
The new created array is :
 1 2 3
The new created array is :
 2.5 3.2 3.3
```

Adding Elements to a Array

Elements can be added to the Array by using built-in `insert()` function. Insert is used to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array. `append()` is also used to add the value mentioned in its arguments at the end of the array.

```
# Python program to demonstrate adding Elements to a Array
# importing "array" for array creations
import array as arr
# array with int type
a = arr.array('i', [1, 2, 3])

print ("Array before insertion : ", end =" ")
for i in range (0, 3):
        print (a[i], end =" ")
print()
# inserting array using insert() function
a.insert(1, 4)
print ("Array after insertion : ", end =" ")
for i in (a):
        print (i, end =" ")
print()
# array with float type
b = arr.array('d', [2.5, 3.2, 3.3])
print ("Array before insertion : ", end =" ")
for i in range (0, 3):
        print (b[i], end =" ")
print()
# adding an element using append()
b.append(4.4)

print ("Array after insertion : ", end =" ")
for i in (b):
        print (i, end =" ")
print()
```

Output :

```
Array before insertion :
 1 2 3
Array after insertion :
 1 4 2 3
Array before insertion :
 2.5 3.2 3.3
Array after insertion :
 2.5 3.2 3.3 4.4
```

Removing Elements from the Array

Elements can be removed from the array by using built-in `remove()` function but an Error arises if element doesn't exist in the set. `Remove()` method only removes one element at a time, to remove range of elements, iterator is used. `pop()` function can also be used to remove and return an element from the array, but by default it removes only the last element of the array, to remove element from a specific position of the array, index of the element is passed as an argument to the `pop()` method.

Output:

```
The new created array is :
 1 2 3 1 5
The popped element is :
 3
The array after popping is :
 1 2 1 5
The array after removing is :
 2 1 5
```

```
# Python program to demonstrate removal of elements in a Array
# importing "array" for array operations
import array
# initializing array with array values
arr = array.array('i', [1, 2, 3, 1, 5])

# printing original array
print ("The new created array is : ", end ="")
for i in range (0, 5):
        print (arr[i], end =" ")
print ("\r")
# using pop() to remove element at 2nd position
print ("The popped element is : ", end ="")
print (arr.pop(2))

# printing array after popping
print ("The array after popping is : ", end ="")
for i in range (0, 4):
        print (arr[i], end =" ")
print("\r")
# using remove() to remove 1st occurrence of 1
arr.remove(1)

# printing array after removing
print ("The array after removing is : ", end ="")
for i in range (0, 3):
        print (arr[i], end =" ")
```

Searching element in a Array

In order to search an element in the array we use a python in-built `index()` method. This function returns the index of the first occurrence of value mentioned in arguments.

```
# Python code to demonstrate
# searching an element in array


# importing array module
import array

# initializing array with array values
arr = array.array('i', [1, 2, 3, 1, 2, 5])

# printing original array
print ("The new created array is : ", end ="")
for i in range (0, 6):
        print (arr[i], end =" ")

print ("\r")
# using index() to print index of 1st occurrence of 2
print ("The index of 1st occurrence of 2 is : ", end ="")
print (arr.index(2))

# using index() to print index of 1st occurrence of 1
print ("The index of 1st occurrence of 1 is : ", end ="")
print (arr.index(1))
```

Output:

```
The new created array is :
 1 2 3 1 2 5
The index of 1st occurrence of 2
is :
 1
The index of 1st occurrence of 1
is :
 0
```

Updating Elements in a Array

In order to update an element in the array we simply reassign a new value to the desired index we want to update.

```
# Python code to update an element in array
# importing array module
import array

# initializing array with array values
arr = array.array('i', [1, 2, 3, 1, 2, 5])

# printing original array
print ("Array before updation : ", end ="")
for i in range (0, 6):
        print (arr[i], end =" ")
print ("\r")
# updating a element in a array
arr[2] = 6
print("Array after updation : ", end ="")
for i in range (0, 6):
        print (arr[i], end =" ")
print()
# updating a element in a array
arr[4] = 8
print("Array after updation : ", end ="")
for i in range (0, 6):
        print (arr[i], end =" ")
```

Output:

```
Array before updation :
 1 2 3 1 2 5
Array after updation :
 1 2 6 1 2 5
Array after updation :
 1 2 6 1 8 5
```

- **append()**:- This function is used to **add the value** mentioned in its arguments at the **end** of the array.
- **insert(i,x)** :- This function is used to **add the value(x) at the ith position** specified in its argument.

```
# Python code to demonstrate the working of
# array(), append(), insert()

# importing "array" for array operations
import array

# initializing array with array values
# initializes array with signed integers
arr = array.array('i', [1, 2, 3])

# printing original array
print ("The new created array is : ",end=" ")
for i in range (0, 3):
    print (arr[i], end=" ")

print("\r")

# using append() to insert new value at end
arr.append(4);

# printing appended array
print("The appended array is : ", end="")
for i in range (0, 4):
    print (arr[i], end=" ")

# using insert() to insert value at specific position
# inserts 5 at 2nd position
arr.insert(2, 5)

print("\r")

# printing array after insertion
print ("The array after insertion is : ", end="")
for i in range (0, 5):
    print (arr[i], end=" ")
```

Output:

```
The new created array is :
 1 2 3

The appended array is :
1 2 3 4

The array after insertion is :
1 2 5 3 4
```

- **pop()**:- This function **removes the element at the position** mentioned in its argument and returns it.
- **remove()**:- This function is used to **remove the first occurrence** of the value mentioned in its arguments.

```python
# Python code to demonstrate the working of
# pop() and remove()

# importing "array" for array operations
import array

# initializing array with array values
# initializes array with signed integers
arr= array.array('i',[1, 2, 3, 1, 5])

# printing original array
print ("The new created array is : ",end="")
for i in range (0,5):
    print (arr[i],end=" ")

print ("\r")

# using pop() to remove element at 2nd position
print ("The popped element is : ",end="")
print (arr.pop(2));

# printing array after popping
print ("The array after popping is : ",end="")
for i in range (0,4):
    print (arr[i],end=" ")

print("\r")

# using remove() to remove 1st occurrence of 1
arr.remove(1)

# printing array after removing
print ("The array after removing is : ",end="")
for i in range (0,3):
    print (arr[i],end=" ")
```

Output

```
The new created array is :
1 2 3 1 5

The popped element is :
3

The array after popping is :
1 2 1 5

The array after removing is :
2 1 5
```

- **index()** :- This function returns the **index of the first occurrence** of value mentioned in arguments.
- **reverse()** :- This function **reverses** the array.

```
# Python code to demonstrate the working of
# index() and reverse()

# importing "array" for array operations
import array

# initializing array with array values
# initializes array with signed integers
arr= array.array('i',[1, 2, 3, 1, 2, 5])

# printing original array
print ("The new created array is : ",end="")
for i in range (0,6):
    print (arr[i],end=" ")

print ("\r")

# using index() to print index of 1st
occurrenece of 2
print ("The index of 1st occurrence of 2 is :
",end="")
print (arr.index(2))

#using reverse() to reverse the array
arr.reverse()

# printing array after reversing
print ("The array after reversing is :
",end="")
for i in range (0,6):
    print (arr[i],end=" ")
```

Output:

```
The new created array is :
1 2 3 1 2 5
The index of 1st occurrence of 2 is :
1
The array after reversing is :
5 2 1 3 2 1
```

- **count()** :- This function **counts the number of occurrences** of argument mentioned in array.
- **extend(arr)** :- This function **appends a whole array** mentioned in its arguments to the specified array.

```
# Python code to demonstrate the working of
# count() and extend()
# importing "array" for array operations
import array
# initializing array 1 with array values
# initializes array with signed integers
arr1 = array.array('i',[1, 2, 3, 1, 2, 5])

# initializing array 2 with array values
# initializes array with signed integers
arr2 = array.array('i',[1, 2, 3])

# using count() to count occurrences of 1 in array
print ("The occurrences of 1 in array is : ")
print (arr1.count(1))

# using extend() to add array 2 elements to array 1
arr1.extend(arr2)
print ("The modified array is : ")
for i in range (0,9):
        print (arr1[i])
```

Output

```
The occurrences of 1 in array is :
2

The modified array is :
1 2 3 1 2 5 1 2 3
```

- **pow(a, b)** :- This function is used to compute value of **a raised to the power b (a******b)**.

- **sqrt()** :- This function returns the **square root** of the number.

```
# Python code to demonstrate the working of
# pow() and sqrt()

# importing "math" for mathematical operations
import math

# returning the value of 3**2
print ("The value of 3 to the power 2 is : ", end="")
print (math.pow(3,2))

# returning the square root of 25
print ("The value of square root of 25 : ", end="")
print (math.sqrt(25))
```

Output:

```
The value of 3 to the power 2 is: 9.0

The value of square root of 25 is:
5.0
```

Matching a Pattern with Text

re.match() : This function attempts to match pattern to whole string. The re.match function returns a match object on success, None on failure. **re.match(pattern, string, flags=0) pattern : Regular expression to be matched. string : String where pattern is searched flags : We can specify different flags using bitwise OR (|).**

```
# A Python program to demonstrate working
# of re.match().
import re

# a sample function that uses regular expressions
# to find month and day of a date.
def findMonthAndDate(string):
        regex = r"([a-zA-Z]+) (\d+)"
        match = re.match(regex, string)

        if match == None:
                print ("Not a valid date")
                return
        print ("Given Data: %s" % (match.group()))
        print ("Month: %s" % (match.group(1)))
        print ("Day: %s" % (match.group(2)))
# Driver Code
findMonthAndDate("Jun 24")
print("")
findMonthAndDate("I was born on June 24")
```

Output:

```
Given Data: Jun 24 Month: Jun Day: 24
Not a valid date
```

Finding all occurrences of a pattern

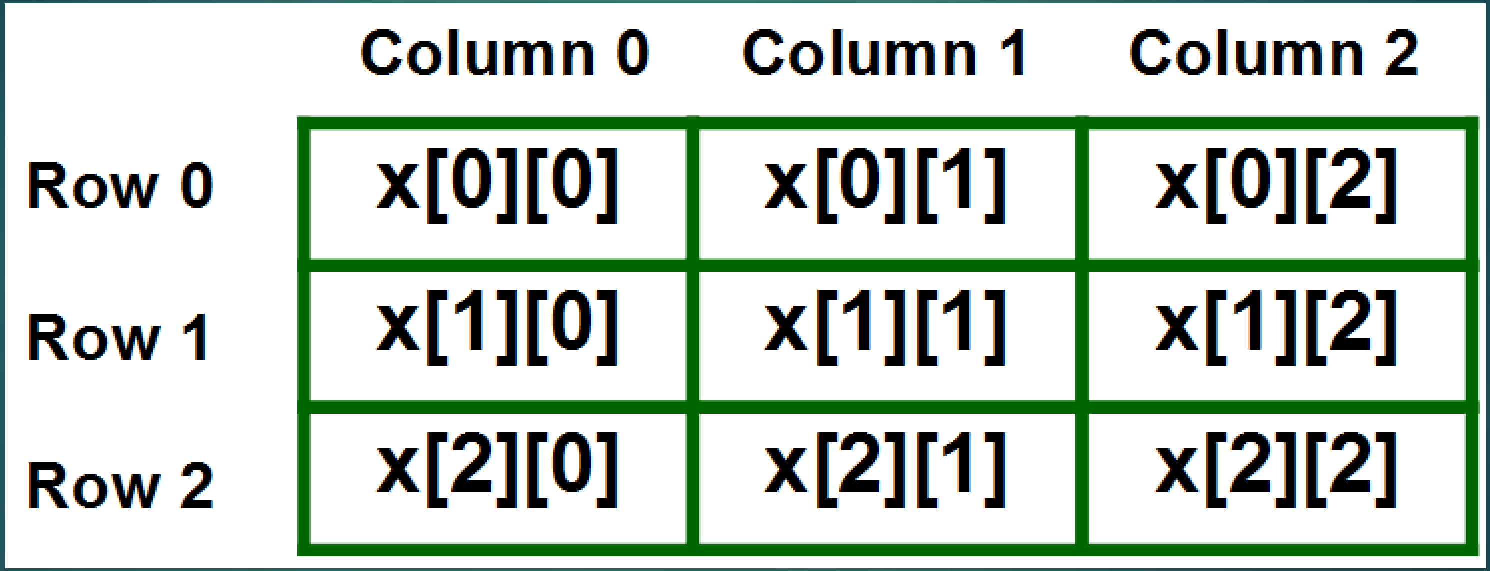
re.findall() : Return all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order found.

```
# A Python program to demonstrate working of
# findall()
import re

# A sample text string where regular expression
# is searched.
string = ""Hello my Number is 123456789 and
my friend's number is 987654321""

# A sample regular expression to find digits.
regex = '\d+'

match = re.findall(regex, string)
print(match)
```

Output :

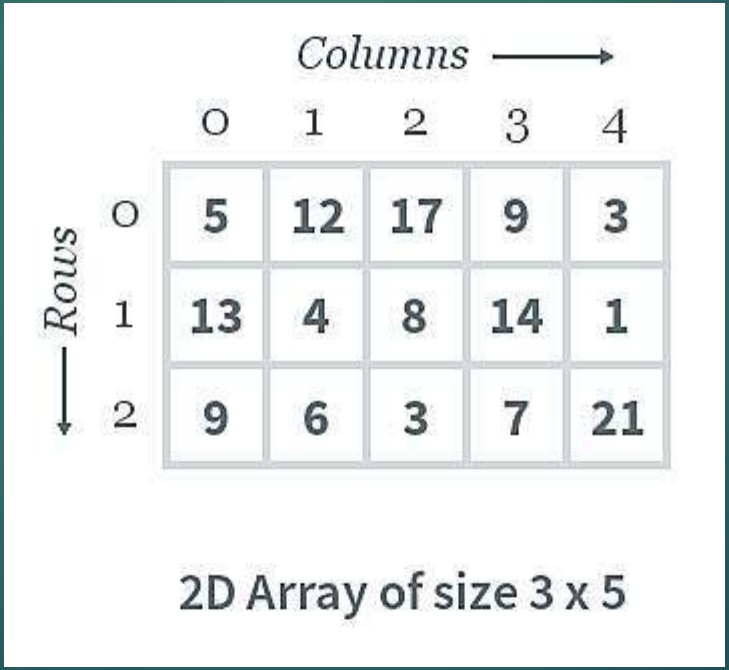
```
['123456789', '987654321']
```

Multidimensional Array Illustration



	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Multidimensional Array Illustration



Columns
Rows

	0	1	2	3	4
0	5	12	17	9	3
1	13	4	8	14	1
2	9	6	3	7	21

2D Array of size 3 x 5

Multidimensional Arrays

For example, a two-dimensional array consists of a certain number of rows and columns:

```
const int NUMROWS = 3;
const int NUMCOLS = 7;
int Array[NUMROWS][NUMCOLS];
```

	0	1	2	3	4	5	6
0	4	18	9	3	-4	6	0
1	12	45	74	15	0	98	0
2	84	87	75	67	81	85	79

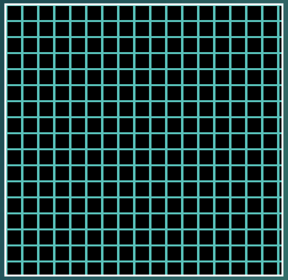
`Array[2][5]`	3rd value in 6th column
`Array[0][4]`	1st value in 5th column

The declaration must specify the number of rows and the number of columns, and both must be constants.

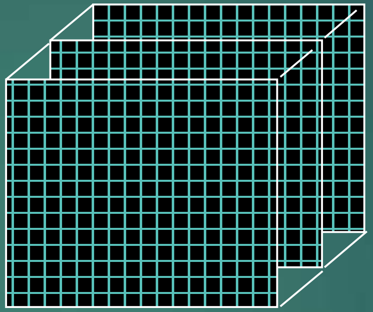
Higher-Dimensional Arrays

An array can be declared with multiple dimensions.

2 Dimensional

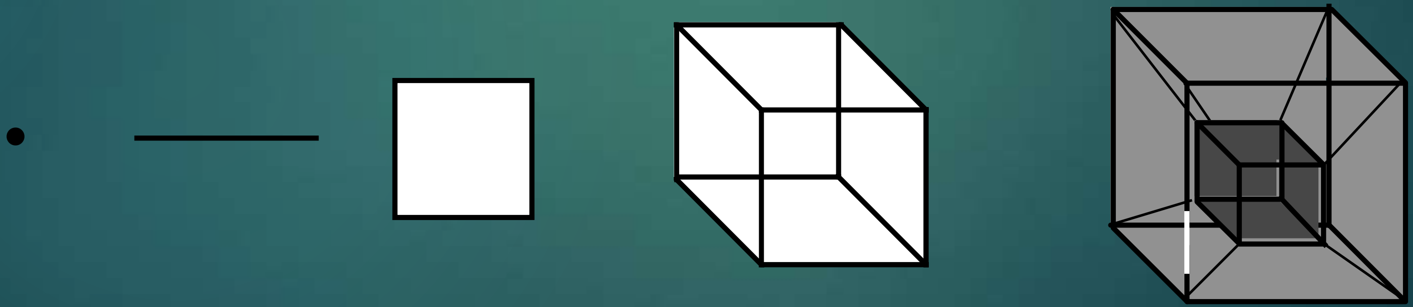


3 Dimensional



```
double
Coord[100][100][100];
```

Multiple dimensions get difficult to visualize graphically.



```
# First method to create a 1 D array
N = 5
arr = [0]*N
print(arr)
```

```
# Second method to create a 1 D
array
N = 5
arr = [0 for i in range(N)]
print(arr)
```

```
# Using above first method to
create a
# 2D array
rows, cols = (5, 5)
arr = [[0]*cols]*rows
print(arr)
```

```
# Using above second method to
create a
# 2D array
rows, cols = (5, 5)
arr = [[0 for i in range(cols)] for j in
range(rows)]
print(arr)
```

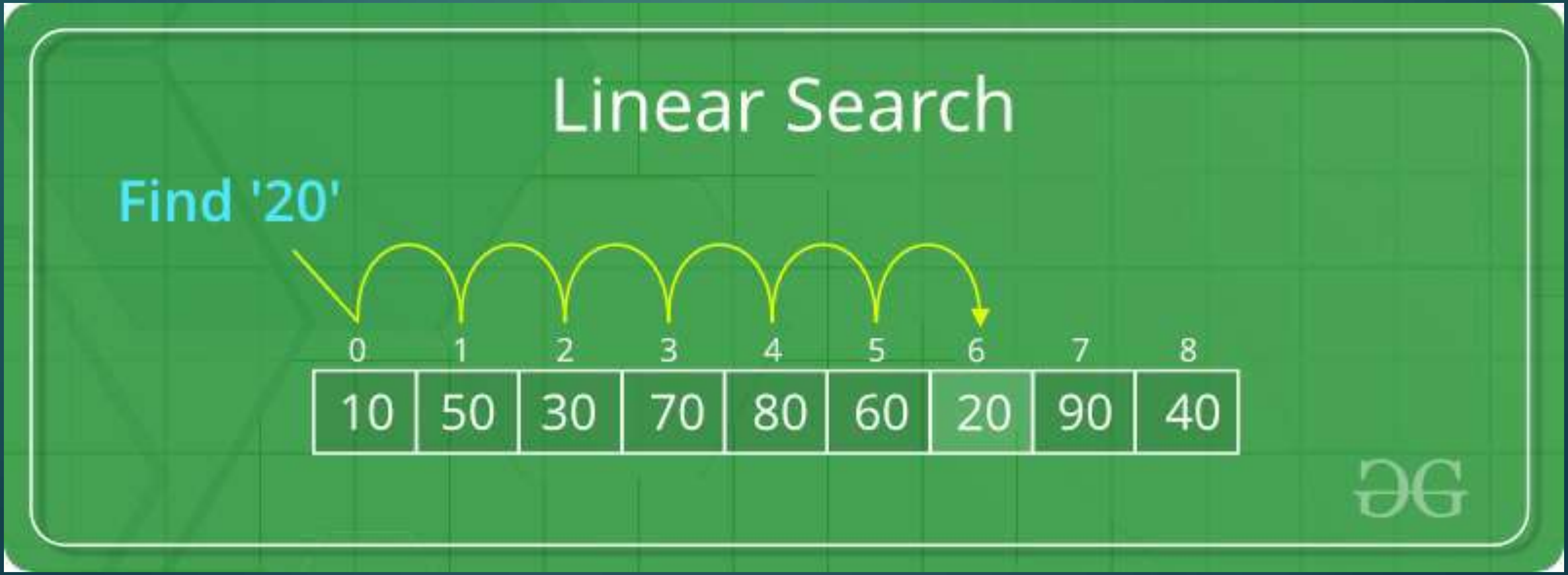
Searching Arrays

Searching is the process of looking for a specific element in an array; for example, discovering whether a certain score is included in a list of scores.

Searching is a common task in computer programming. There are many algorithms and data structures devoted to searching.

Linear Search

linear search approach compares the key element, key, with each element in the array list[]. The method continues to do so until the key matches an element in the list or the list is exhausted without a match being found. If a match is made, the linear search returns the index of the element in the array that matches the key. If no match is found, the search returns -1.

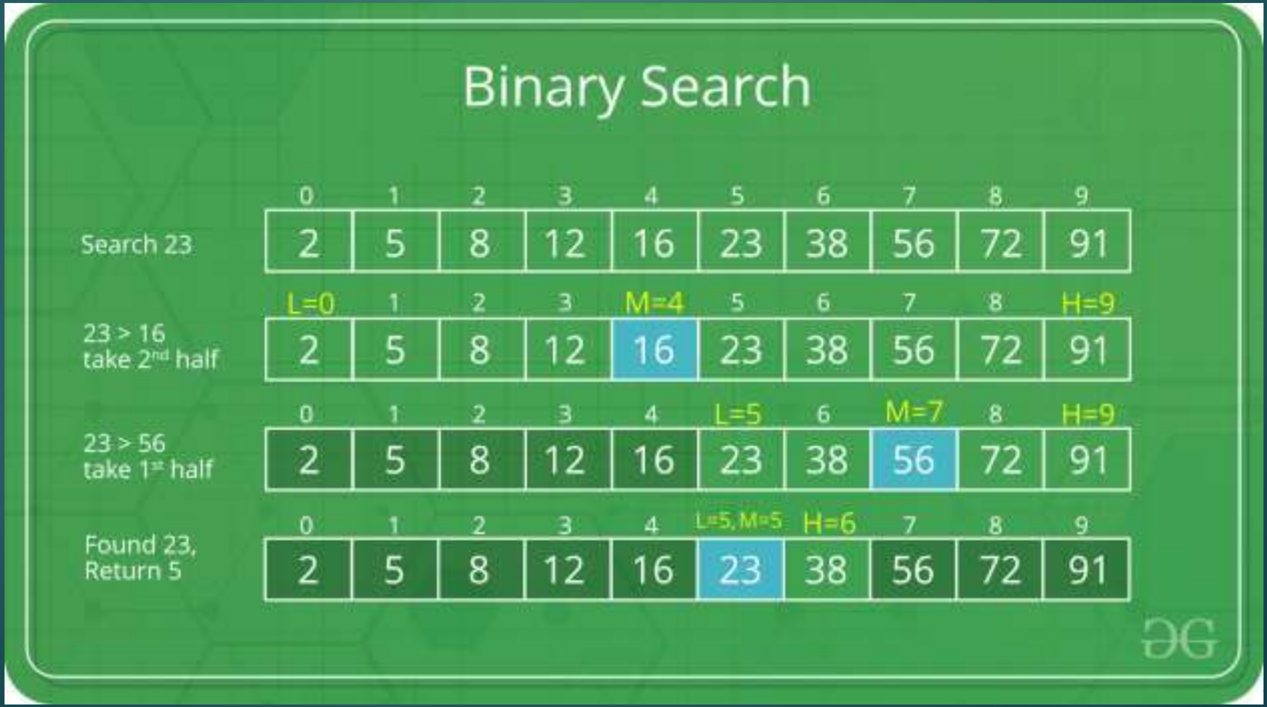


Linear Search

```
# Python3 code to linearly search x in arr[].
# If x is present then return its location, otherwise return -1
def search(arr, n, x):
        for i in range(0, n):
                if (arr[i] == x):
                        return i
        return -1
# Driver Code
arr = [2, 3, 4, 10, 40]
x = 10
n = len(arr)

# Function call
result = search(arr, n, x)
if(result == -1):
        print("Element is not present in array")
else:
        print("Element is present at index", result)
```

Binary search

Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.



Binary Search

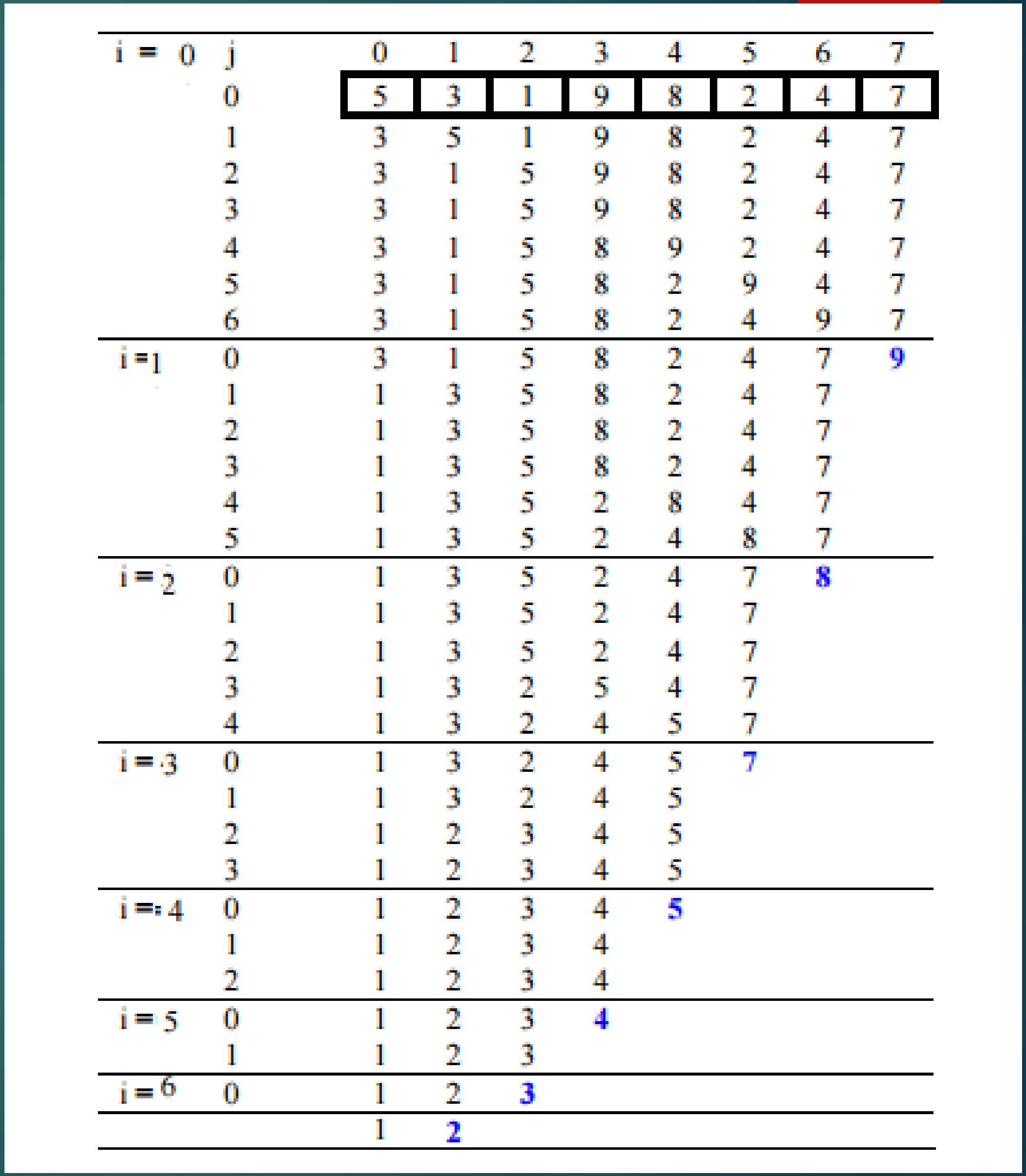
```
# It returns location of x in given array arr
# if present, else returns -1
def binarySearch(arr, l, r, x):
        while l <= r:
                mid = l + (r - l) // 2;
                # Check if x is present at mid
                if arr[mid] == x:
                        return mid
                # If x is greater, ignore left half
                elif arr[mid] < x:
                        l = mid + 1
                else:
                        r = mid - 1
        return -1
# Driver Code
arr = [ 2, 3, 4, 10, 40 ]
x = 10
# Function call
result = binarySearch(arr, 0, len(arr)-1, x)
if result != -1:
        print ("Element is present at index % d" %
result)
else:
        print ("Element is not present in array")
```

Sorting Array

- **Sorting** is the basic operation in computer science. Sorting is the process of arranging data in some given sequence or order (in increasing or decreasing order).
- For example you have an array which contain 10 elements as follow;
  10, 3 ,6 12, 4, 17, 5, 9
- After sorting value must be;
  3, 4, 5, 6, 9, 10, 12, 17

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.



i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i = 1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	4				
	1	1	2	3					
i = 6	0	1	2	3					
		1	2						

Bubble Sort

```
# Python program for implementation of Bubble Sort
def bubbleSort(arr):
        n = len(arr)

        # Traverse through all array elements
        for i in range(n):
        # Last i elements are already in place
                for j in range(0, n-i-1):
                # traverse the array from 0 to n-i-1
                # Swap if the element found is greater than the next element

                if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
# Driver code to test above
arr = [64, 34, 25, 12, 22, 11, 90]
bubbleSort(arr)
print ("Sorted array is:")
for i in range(len(arr)):
        print ("%d" %arr[i]),
```

Output:

```
Sorted array: 11 12 22 25 34 64 90
```

Data Structure

Dr. Itimad Raheem Ali

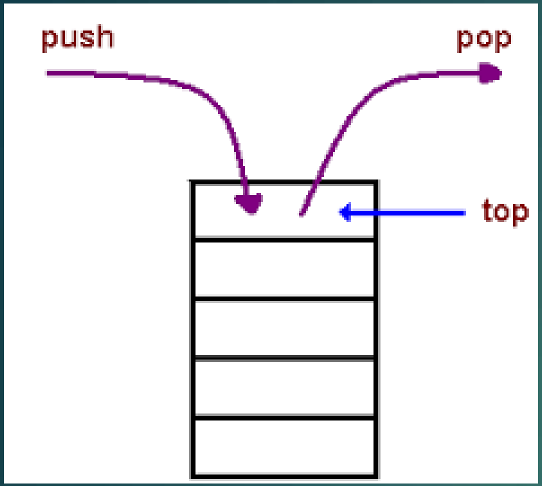
Email : itimadra@uoitc.edu.iq

Lecture 3

Stacks

- Stacks are a straight forward and simple data structure
- Access is allowed only at one point of the structure, normally termed the *top* of the stack, so the operations are limited:
  - push (add item to stack)
  - pop (remove top item from stack)
  - top (get top item without removing it)
  - clear

FILO(First In Last Out)



Some real life examples of Stack
A stack of cafeteria trays
A stack of pennies
A stack of shoe boxes
A stack of neatly folded shirts

Stack Operations

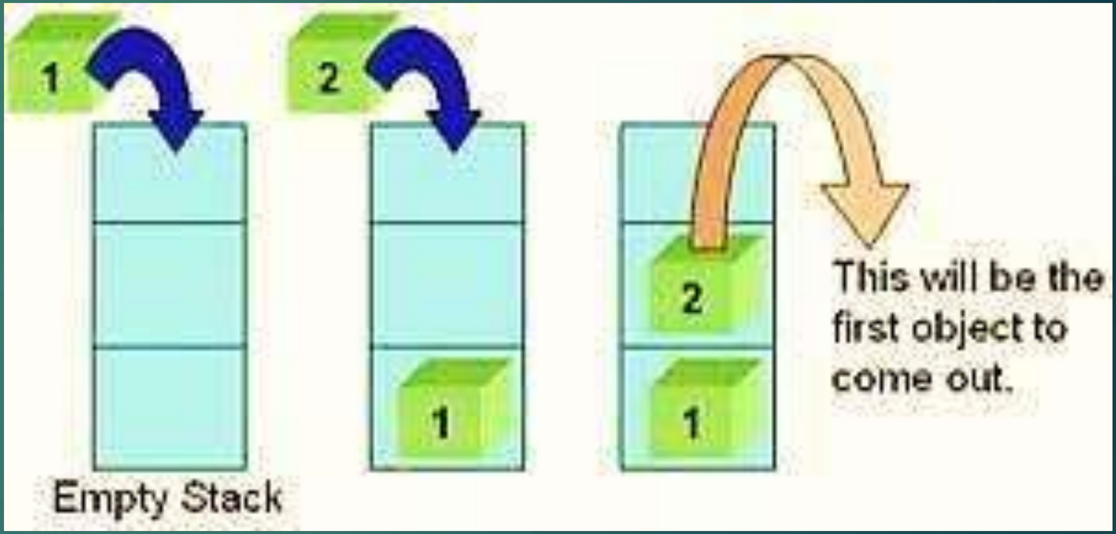
Assume a simple stack for integers.

Stack s = new Stack();

s.push(1)

s.push(2)

s.pop()



//what are contents of stack?

Example

- Let we consider S means (Stacking) and U means (Unstacking) and the input numbers are : (13, 2, 25, 14, 18)

What are the result after execute :

1) SSUSUSUSUU

13	2		25		14		18		
S	S	U	S	U	S	U	S	U	U
		2		25		14		18	13

2) SSSUSSUUUU

13	2	25		14	18				
S	S	S	U	S	S	U	U	U	U
			25			18	14	2	13

H.W.

If u have (A,B,C,D,E) what are the result of :

- SSUSSSU
- SSUSSSUU
- SSUSSSUUUU

Stack Operations

The functions associated with stack are:

empty(): Returns whether the stack is empty,
Time Complexity : O(1)

size(): Returns the size of the stack,
Time Complexity : O(1)

top(): Returns a reference to the top most element of the stack,
Time Complexity : O(1)

push(g): Adds the element ‘g’ at the top of the stack,
Time Complexity : O(1)

pop(): Deletes the top most element of the stack,
Time Complexity : O(1)

Stack Operations (Add and Delete)

```
# Python program to
# demonstrate stack implementation
stack = [ ]
# append() function to push element in the stack
stack.append('a')
stack.append('b')
stack.append('c')
print('Initial stack')
print(stack)
# pop() fucntion to pop element from stack in LIFO order
print('\nElements poped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())
print('\nStack after elements are poped:')
print(stack)
```

Output:

```
Initial stack
 ['a', 'b', 'c']
Elements poped from stack:
 c b a
Stack after elements are
poped: []
```

Reverse a string using stack

Given a string, reverse it using stack. For example "GeeksQuiz" should be converted to "ziuQskeeG".

```
# Python program to reverse a string using stack
# Function to create an empty stack. It initializes size of stack as 0
def createStack():
        stack=[]
        return stack
# Function to determine the size of the stack
def size(stack):
        return len(stack)
# Stack is empty if the size is 0
def isEmpty(stack):
        if size(stack) == 0:
                return true
# Function to add an item to stack . It increases size by 1
def push(stack,item):
        stack.append(item)
#Function to remove an item from stack. It decreases size by 1
def pop(stack):
        if isEmpty(stack): return
        return stack.pop()
# A stack based function to reverse a string
def reverse(string):
        n = len(string)
        # Create a empty stack
        stack = createStack()
# Push all characters of string to stack
        for i in range(0,n,1):
                push(stack,string[i])
# Making the string empty since all characters are saved in stack
        string=""
# Pop all characters of string and put them back to string
        for i in range(0,n,1):
                string+=pop(stack)
        return
# Driver program to teststring above functions
string="GeeksQuiz"
string = reverse(string)
print("Reversed string is " + string)
```

Output: `Reversed string is ziuQskeeG`

Delete middle element of a stack

```
# Python code to delete middle of a stack  without using
additional data structure.
# Deletes middle of stack of size  n. Curr is current item
number
class Stack:
        def __init__(self):
                self.items = []
        def isEmpty(self):
                return self.items == []
        def push(self, item):
                self.items.append(item)
        def pop(self):
                return self.items.pop()
        def peek(self):
                return self.items[len(self.items)-1]
         def size(self):
                return len(self.items)
         def deleteMid(st, n, curr) :
# If stack is empty or all items are traversed
         if (st.isEmpty() or curr == n) :
                 return
         # Remove current item
                x = st.peek()
                st.pop()
         # Remove other items
                deleteMid(st, n, curr+1)
         # Put all items back except middle
                if (curr != int(n/2)) :
                        st.push(x)
```

```
# Driver function to test above functions
st = Stack()
# push elements into the stack
st.push('1')
st.push('2')
st.push('3')
st.push('4')
st.push('5')
st.push('6')
st.push('7')
deleteMid(st, st.size(), 0)

# Printing stack after deletion
# of middle.
while (st.isEmpty() == False) :
                p = st.peek()
                st.pop()
                print (str(p) + " ", end="")
# This code is contributed by
# Manish Shaw (manishshaw1)
```

```
Input  : Stack[] = [1, 2, 3, 4, 5]
Output : Stack[] = [1, 2, 4, 5]
Input  : Stack[] = [1, 2, 3, 4, 5, 6]
Output : Stack[] = [1, 2, 4, 5, 6]
```

Mathematical Calculations

What is 3 + 2 * 4?   2 * 4 + 3?   3 * 2 + 4?

- The precedence of operators affects the order of operations. A mathematical expression cannot simply be evaluated left to right.
- A challenge when evaluating a program.
- *Lexical analysis* is the process of interpreting a program.
- Involves Tokenization

What about 1 - 2 - 4 ^ 5 * 3 * 6 / 7 ^ 2 ^ 2

Infix and Postfix Expressions

- The way we are used to writing expressions is known as infix notation
- Postfix expression does not require any precedence rules
- 3 2 * 1 + is postfix of 3 * 2 + 1
- evaluate the following postfix expressions and write out a corresponding infix expression:

2 3 2 4 * + *	1 2 3 4 ^ * +
1 2 - 3 2 ^ 3 * 6 / +	2 5 ^ 1 -

Evaluation of Postfix Expressions

- Easy to do with a stack
- given a proper postfix expression:
  - get the next token
  - if it is an operand push it onto the stack
  - else if it is an operator
    - pop the stack for the right hand operand
    - pop the stack for the left hand operand
    - apply the operator to the two operands
    - push the result onto the stack
  - when the expression has been exhausted the result is the top (and only element) of the stack

Infix to Postfix

- Convert the following equations from infix to postfix:
  - 2 ^ 3 ^ 3 + 5 * 1

    2 3 3 ^ ^ 5 1 * +
  - 11 + 2 - 1 * 3 / 3 + 2 ^ 2 / 3

    11 2 + 1 3 * 3 / - 2 2 ^ 3 / +

Problems:

parentheses in expression

Infix to Postfix Conversion

- Requires operator precedence parsing algorithm
  - parse v. To determine the syntactic structure of a sentence or other utterance

Operands: add to expression

Close parenthesis: pop stack symbols until an open parenthesis appears

Operators:

Pop all stack symbols until a symbol of lower precedence appears. Then push the operator

End of input: Pop all remaining stack symbols and add to the expression

Simple Example

Infix Expression: 3 + 2 * 4

PostFix Expression:

Operator Stack:

Simple Example

Infix Expression: + 2 * 4

PostFix Expression: 3

Operator Stack:

Simple Example

Infix Expression:	2 * 4
PostFix Expression:	3
Operator Stack:	+

Simple Example

Infix Expression:	* 4
PostFix Expression:	3 2
Operator Stack:	+

Simple Example

Infix Expression:	4
PostFix Expression:	3 2
Operator Stack:	+ *

Simple Example

Infix Expression:	
PostFix Expression:	3 2 4
Operator Stack:	+ *

Simple Example

Infix Expression:	
PostFix Expression:	3 2 4 *
Operator Stack:	+

Simple Example

Infix Expression:

PostFix Expression: 3 2 4 * +

Operator Stack:

Examples

- a+b*c-(d/e+f*g*h) $\Rightarrow$ abc*+de/fg*h*+-
- a+b*c $\Rightarrow$ abc*+

```
# Python program to convert an infix  expression to a postfix
expression
# using two precedence function
# To check if the input character is an operator or a '('
def isOperator(input):
        switch = {
                '+': 1,
                '-': 1,
                '*': 1,
                '/': 1,
                '%': 1,
                '(': 1,
        }
        return switch.get(input, False)

# To check if the input character is an operand
def isOperand(input):
        if ((ord(input) >= 65 and ord(input) <= 90) or
        (ord(input) >= 97 and ord(input) <= 122)):
            return 1

        return 0
# Function to return precedence value if operator is present in stack
def inPrec(input):
        switch = {
                '+': 2,
                '-': 2,
                '*': 4,
                '/': 4,
                '%': 4,
                '(': 0,
        }
        return switch.get(input, 0)
# Function to return precedence value if operator is present outside
stack.
def outPrec(input):
        switch = {
                '+': 1,
                '-': 1,
                '*': 3,
                '/': 3,
                '%': 3,
                '(': 100,
        }
        return switch.get(input, 0)
# Function to convert infix to postfix
def inToPost(input):
```

```
    i = 0
        s = []
        # While input is not NULL iterate
        while (len(input) != i):
                # If character an operand
                if (isOperand(input[i]) == 1):
                        print(input[i], end = "")
                # If input is an operator, push
                elif (isOperator(input[i]) == 1):
                        if (len(s) == 0 or
                                outPrec(input[i]) >
                                inPrec(s[-1])):
                                s.append(input[i])

                        else:
                        while(len(s) > 0 and outPrec(input[i]) <
                            inPrec(s[-1])):
                                print(s[-1], end = "")
                                s.pop()
                                s.append(input[i])
                # Condition for opening bracket
                elif(input[i] == ')'):
                        while(s[-1] != '('):
                                print(s[-1], end = "")
                                s.pop()
                        # If opening bracket not present
                                if(len(s) == 0):
                                print('Wrong input')
                                exit(1)
                        # pop the opening bracket.
                        s.pop()
                        i += 1
                # pop the remaining operators
        while(len(s) > 0):
                if(s[-1] == '('):
                        print('Wrong input')
                        exit(1)

                print(s[-1], end = "")
                s.pop()

# Driver code
input = "a+b*c-(d/e+f*g*h)"
inToPost(input)
```

H.W.

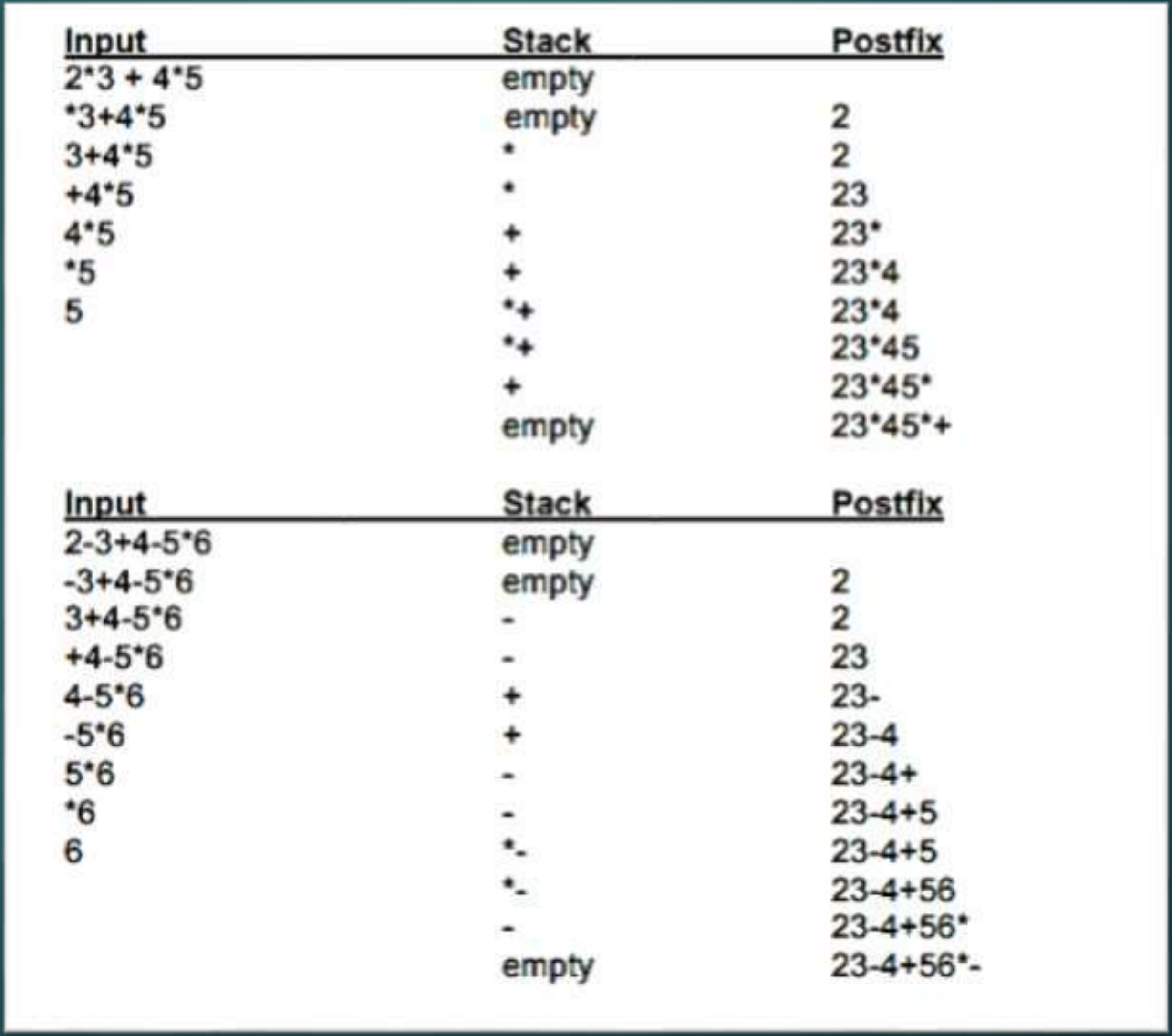
- 3+4*5/6
- (30+23)*(43-21)/(84+7)
- (4+8)*(6-5)/((3-2)*(2+2))

(x+y)/ (w-z)* A

Examples

Convert the following expression from infix to postfix

- 2*3+4*5
- 2-3+4-5*6

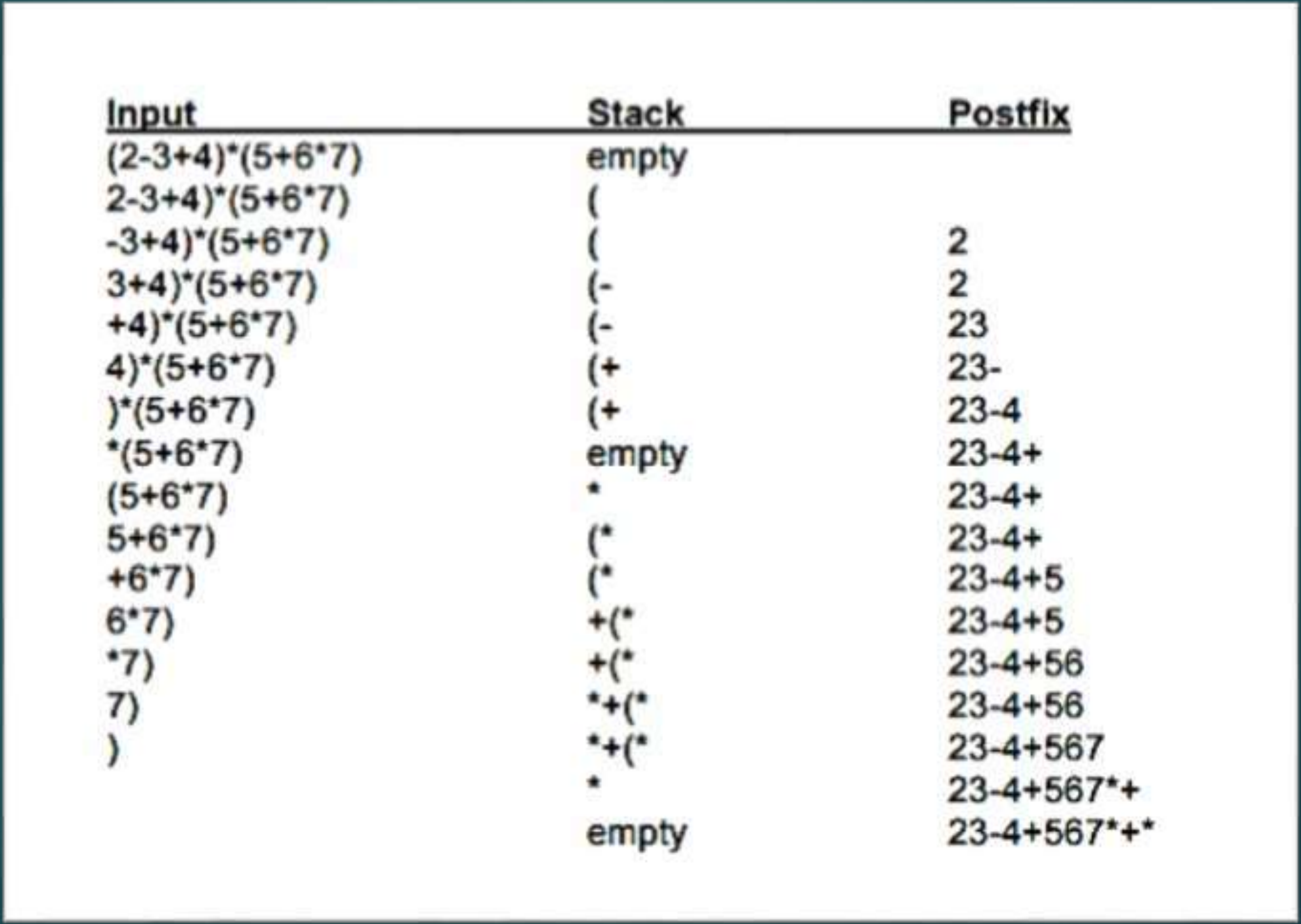


Input	Stack	Postfix
2*3 + 4*5	empty	
*3+4*5	empty	2
3+4*5	*	2
+4*5	*	23
4*5	+	23*
*5	+	23*4
5	*+	23*4
	*+	23*45
	+	23*45*
	empty	23*45*+

Input	Stack	Postfix
2-3+4-5*6	empty	
-3+4-5*6	empty	2
3+4-5*6	-	2
+4-5*6	-	23
4-5*6	+	23-
-5*6	+	23-4
5*6	-	23-4+
*6	-	23-4+5
6	*-	23-4+5
	*-	23-4+56
	-	23-4+56*
	empty	23-4+56*-

Example

- (2-3+4)*(5+6*7)



Input	Stack	Postfix
(2-3+4)*(5+6*7)	empty	
2-3+4)*(5+6*7)	(	
-3+4)*(5+6*7)	(	2
3+4)*(5+6*7)	(-	2
+4)*(5+6*7)	(-	23
4)*(5+6*7)	(+	23-
)*(5+6*7)	(+	23-4
*(5+6*7)	empty	23-4+
(5+6*7)	*	23-4+
5+6*7)	(*	23-4+
+6*7)	(*	23-4+5
6*7)	+(*	23-4+5
*7)	+(*	23-4+56
7)	*+(*	23-4+56
)	*+(*	23-4+567
	*	23-4+567*+
	empty	23-4+567*+*

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
  - Validate XML
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Data Structure

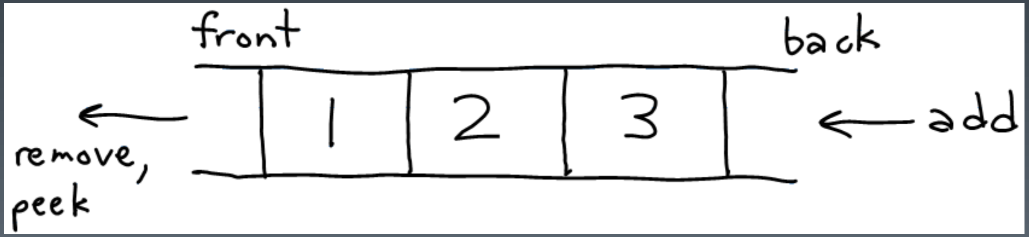
Dr. Itimad Raheem Ali

Email : itimadra@uoitc.edu.iq

Lecture 4

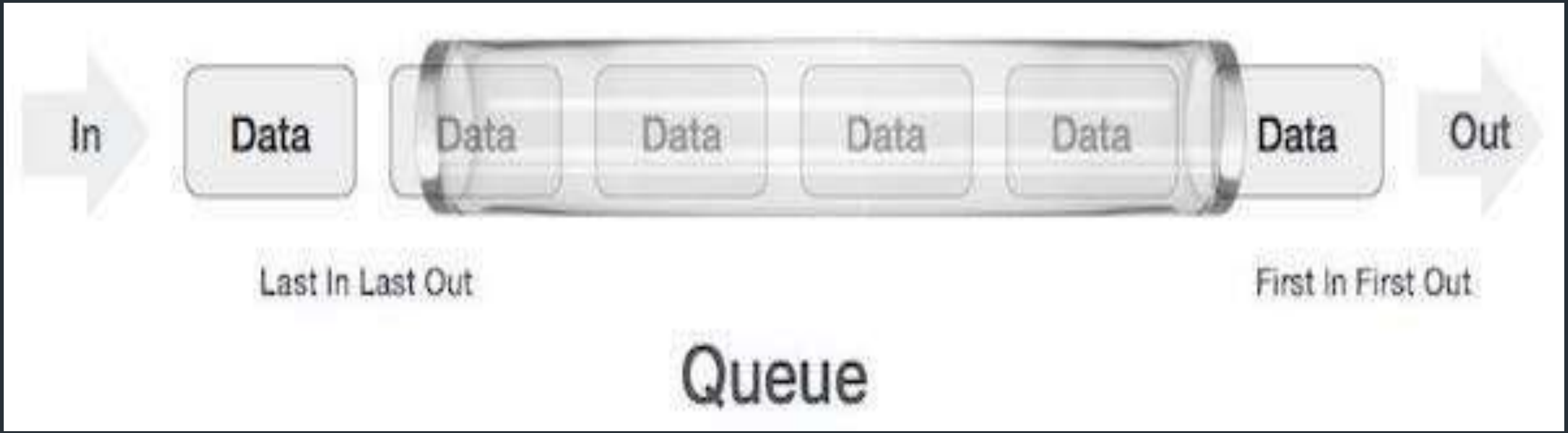
Queues

- Closely related to Stacks
- Like a line
  - In Britain people don’t “get in line” they “queue up”.
- Queues are a first in first out data structure
  - FIFO (or LILO)
- Add items to the end of the queue
- Access and remove from the front

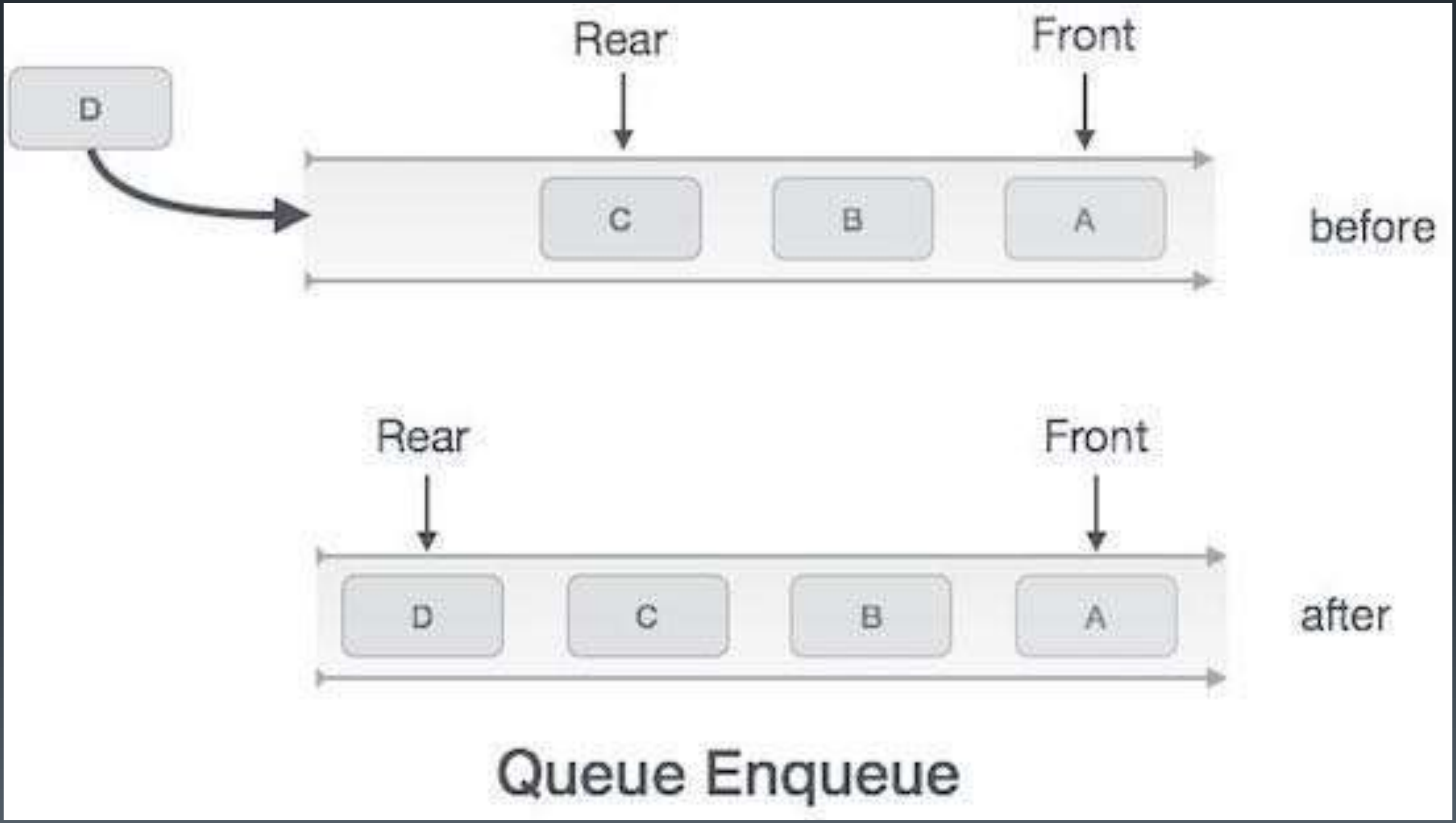


Queue operations

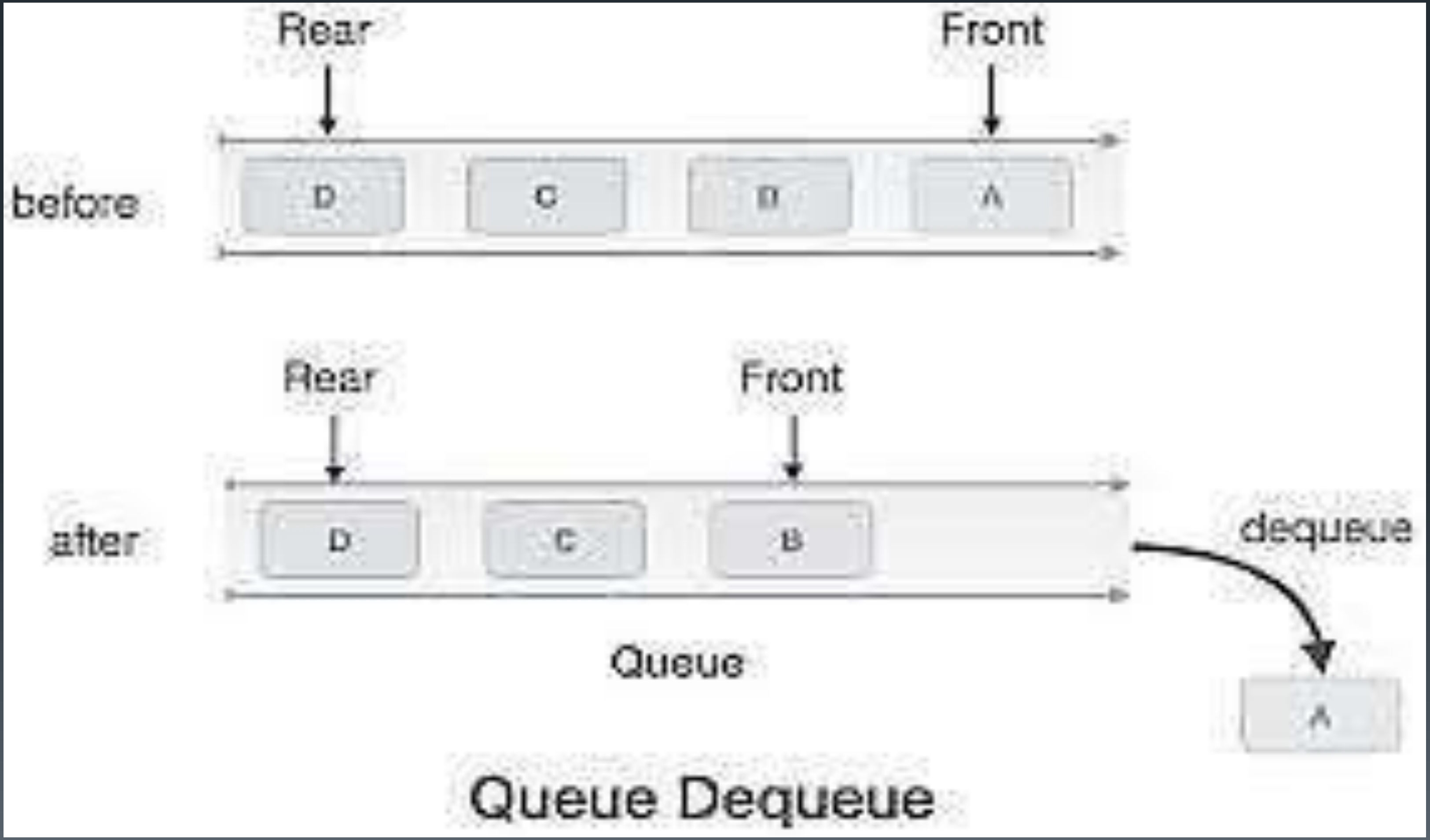
- `add(Object item)`
  - `enqueue(Object item)`
- `Object remove()`
  - `dequeue()`
- `boolean isEmpty()`
- Specify in an interface, allow varied implementations


In
Data
Data
Data
Data
Data
Data
Out
Last In Last Out
First In First Out
Queue

Queue Operations



Queue Operations



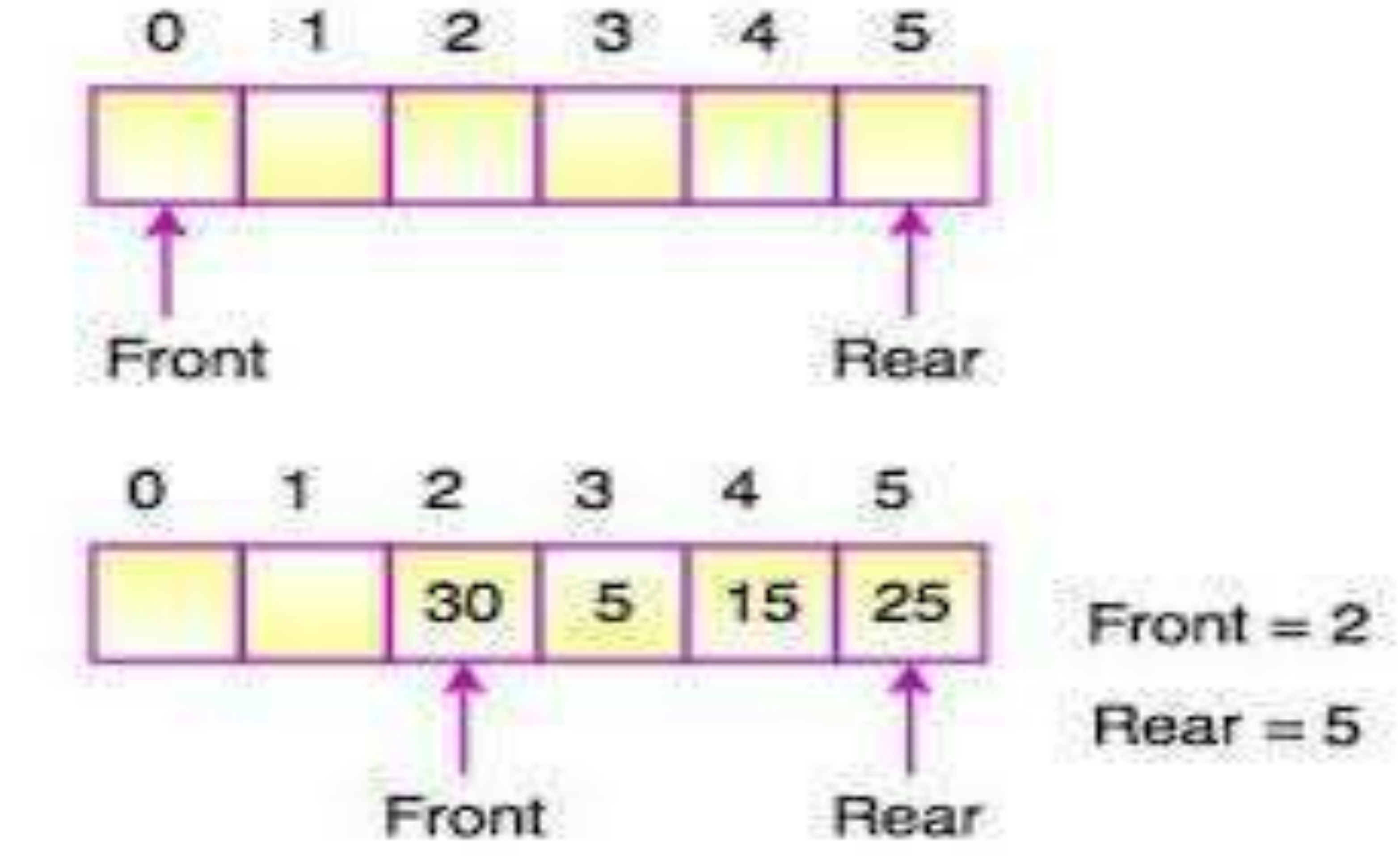


Fig. Implementation of Queue using Array

Applications of Queues

- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
  - queue of network data packets to send

- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Queue Operations

Insert and remove element

```
r =-1
f =-1
def create():
    Q=[]
    return Q

def isempty(Q):
    global f
    if f == -1:
        return True

def enqueue(Q, item):
    global r,f
    r = r+1
    Q.insert(r,item)
    if r == 0:
        f = 0
```

```
def dequeue(Q):
    global f,r
    if isempty(Q):
        print("empty")
    else:
        item=Q[f]
        if f == r:
            f = -1
            r = -1
        else:
            f = f+1
        return item
Q = create()
enqueue(Q,4)
enqueue(Q,7)
print(dequeue(Q))
print(dequeue(Q))
```

Operations on Deque:

Mainly the following four basic operations are performed on queue:

insertFront(): Adds an item at the front of Deque.

insertLast(): Adds an item at the rear of Deque.

deleteFront(): Deletes an item from front of Deque.

deleteLast(): Deletes an item from rear of Deque.

In addition to above operations, following operations are also supported

getFront(): Gets the front item from queue.

getRear(): Gets the last item from queue.

isEmpty(): Checks whether Deque is empty or not.

isFull(): Checks whether Deque is full or not.

Operations on deque

```
# Python code to demonstrate working of  append(), appendleft(), pop(), and popleft()

# importing "collections" for deque operations
import collections

# initializing deque
de = collections.deque([1,2,3])

# using append() to insert element at right end inserts 4 at the end of deque
de.append(4)

# printing modified deque
print ("The deque after appending at right is : ")
print (de)

# using appendleft() to insert element at right end inserts 6 at the beginning of deque
de.appendleft(6)
# printing modified deque
print ("The deque after appending at left is : ")
print (de)

# using pop() to delete element from right end deletes 4 from the right end of deque
de.pop()

# printing modified deque
print ("The deque after deleting from right is : ")
print (de)

# using popleft() to delete element from left end  deletes 6 from the left end of deque
de.popleft()

# printing modified deque
print ("The deque after deleting from left is : ")
print (de)
```

count() :- This function **counts the number of occurrences** of value mentioned in arguments.

```
# Python code to demonstrate working of # insert(), index(), remove(), count()

# importing "collections" for deque operations
import collections
# initializing deque
        de = collections.deque([1, 2, 3, 3, 4, 2, 4])

# using index() to print the first occurrence of 4
        print ("The number 4 first occurs at a position : ")
        print (de.index(4,2,5))

# using insert() to insert the value 3 at 5th position
        de.insert(4,3)
# printing modified deque
        print ("The deque after inserting 3 at 5th position is : ")
        print (de)

# using count() to count the occurrences of 3
        print ("The count of 3 in deque is : ")
        print (de.count(3))

# using remove() to remove the first occurrence of 3
        de.remove(3)

# printing modified deque
        print ("The deque after deleting first occurrence of 3 is : ")
        print (de)
```

Output:

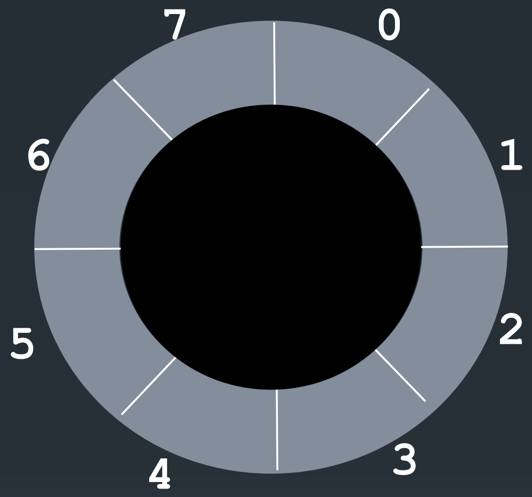
```
The number 4 first occurs at a
position : 4
The deque after inserting 3 at
5th position is : deque([1, 2,
3, 3, 3, 4, 2, 4])
The count of 3 in deque is : 3
The deque after deleting first
occurrence of 3 is : deque([1,
2, 3, 3, 4, 2, 4])
```

Circular Queue

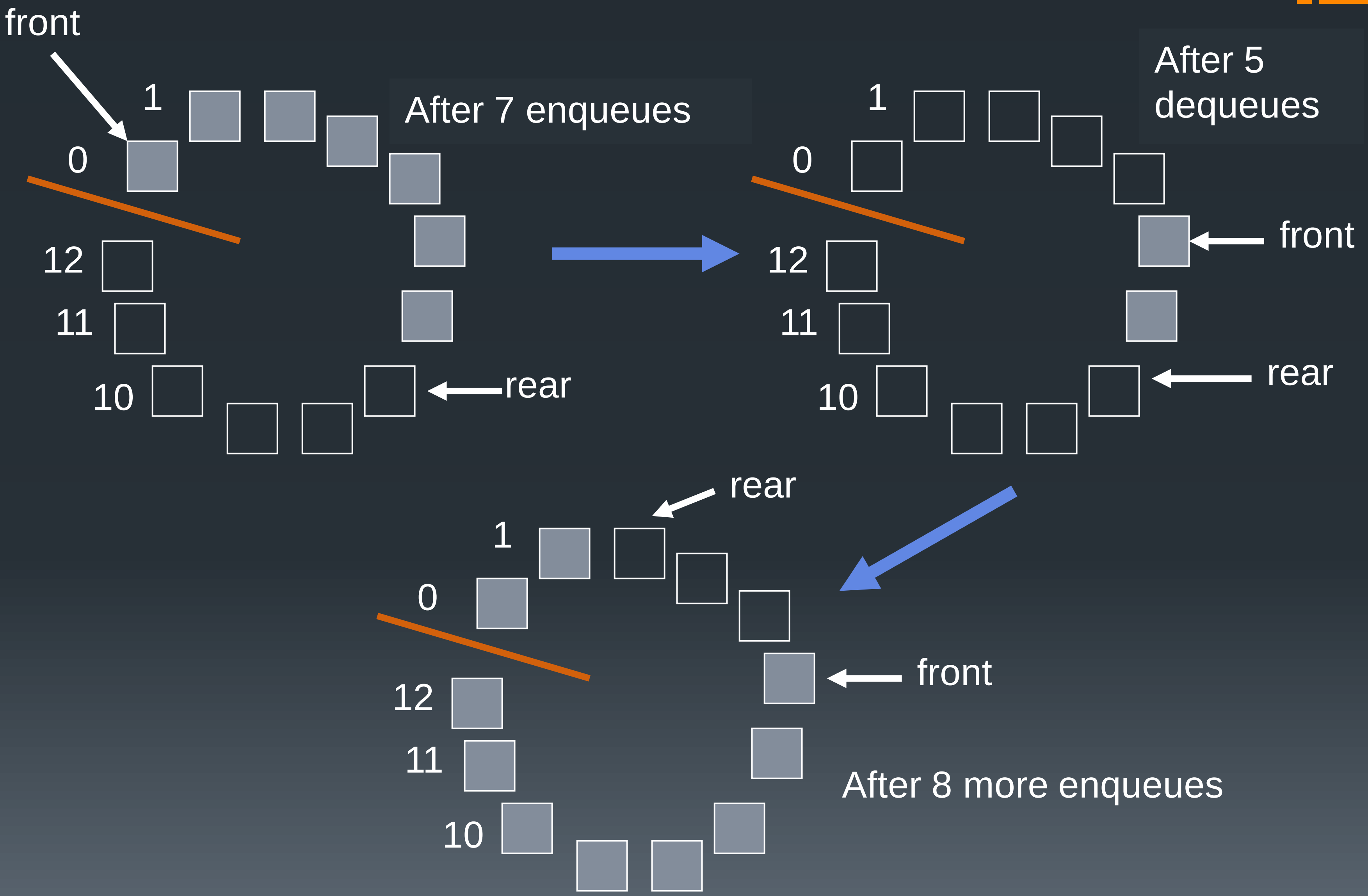
Circular array is an array that conceptually loops around on itself

- The last index is thought to "***precede***" index 0
- In an array whose last index is **n**, the location "***before***" index **0** is index **n**; the location "***after***" index **n** is index **0**

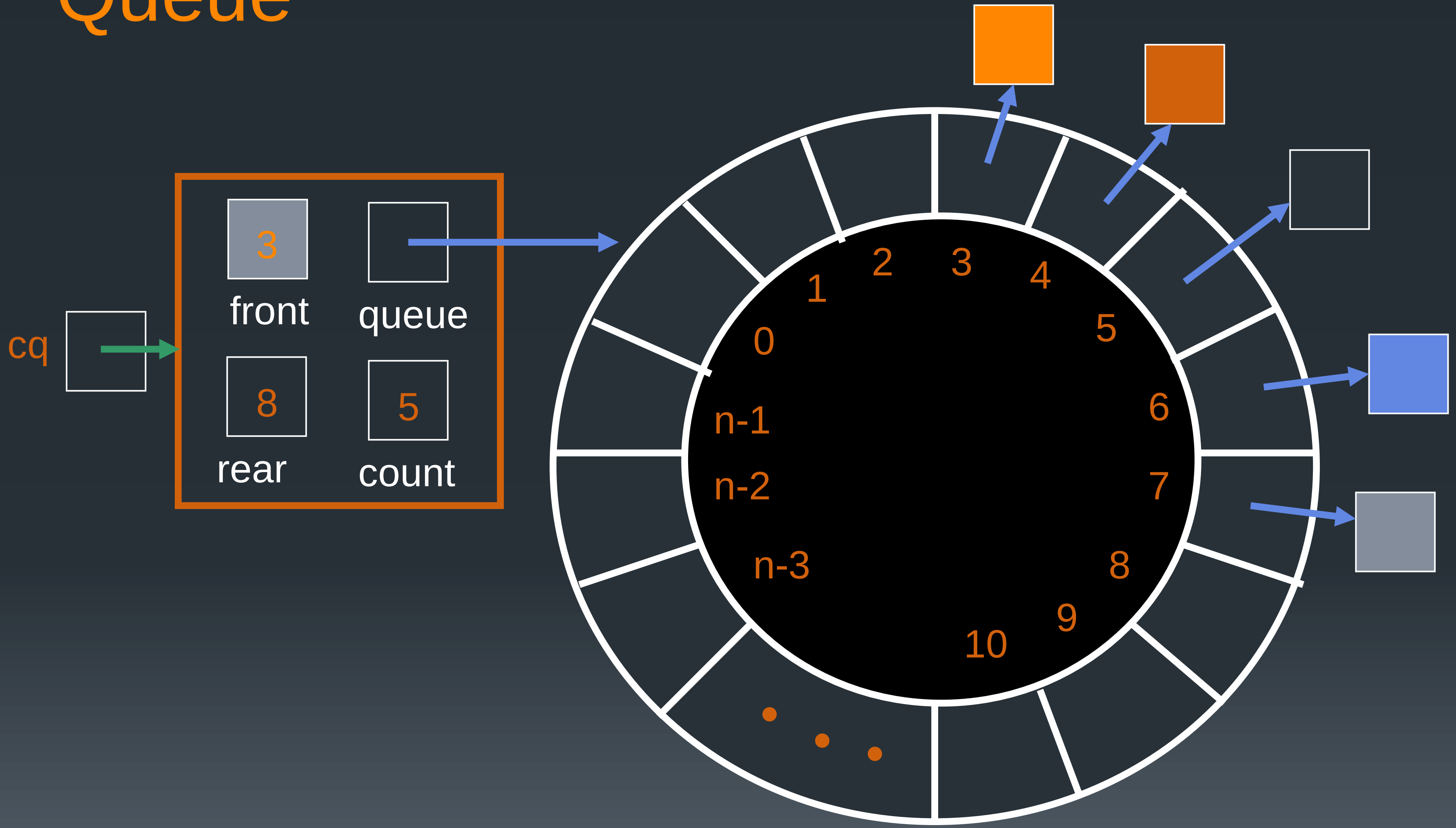
Need to keep track of where the ***front*** as well as the ***rear*** of the queue are at any given time



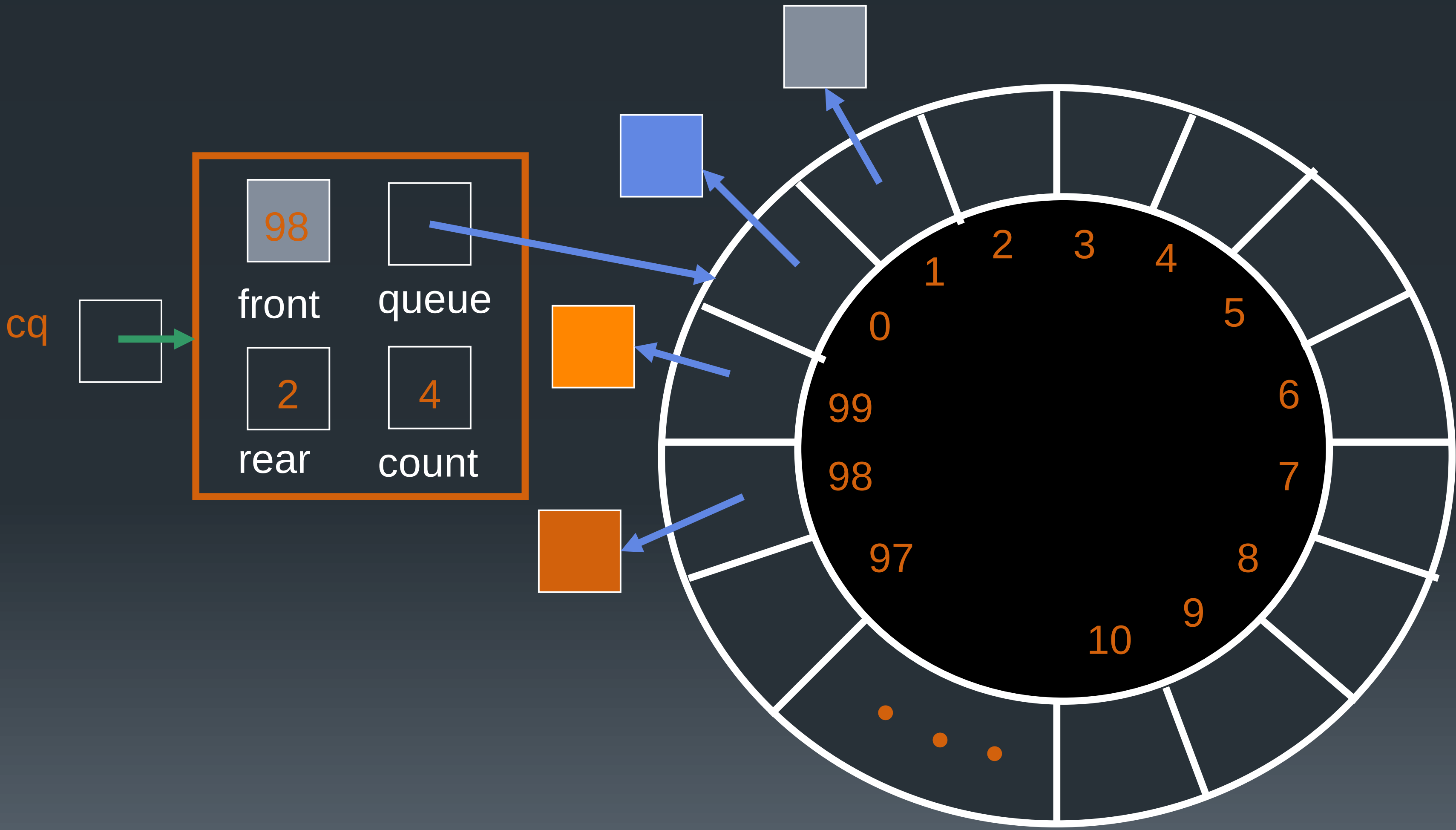
Conceptual Example of a Circular Queue



Circular Array Implementation of a Queue

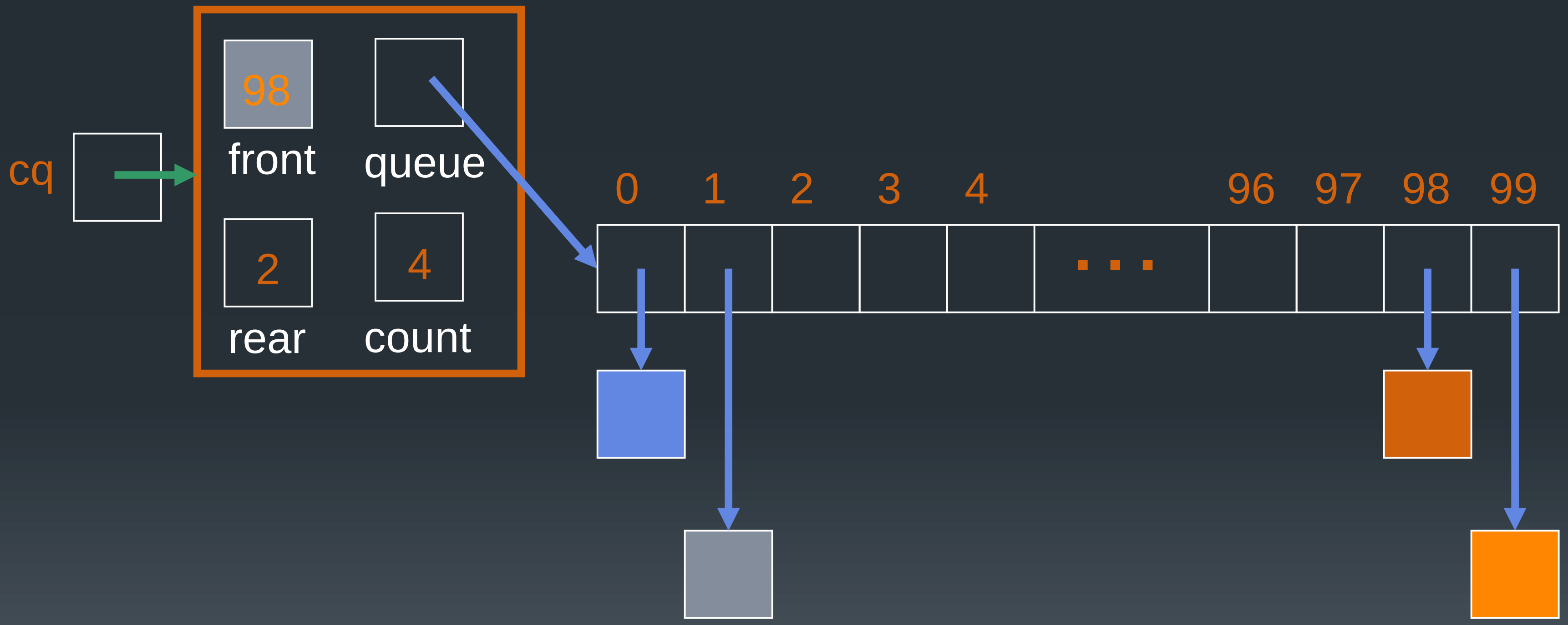


A Queue Straddling the End of a Circular Array

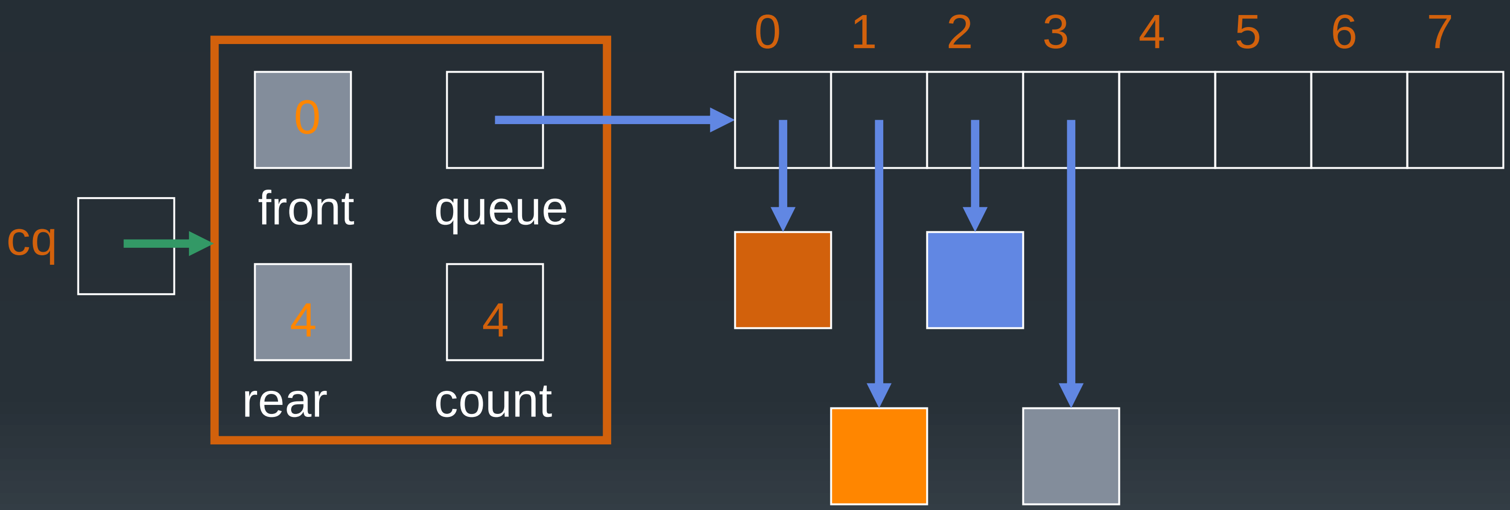


Circular Queue Drawn Linearly

Queue from previous slide

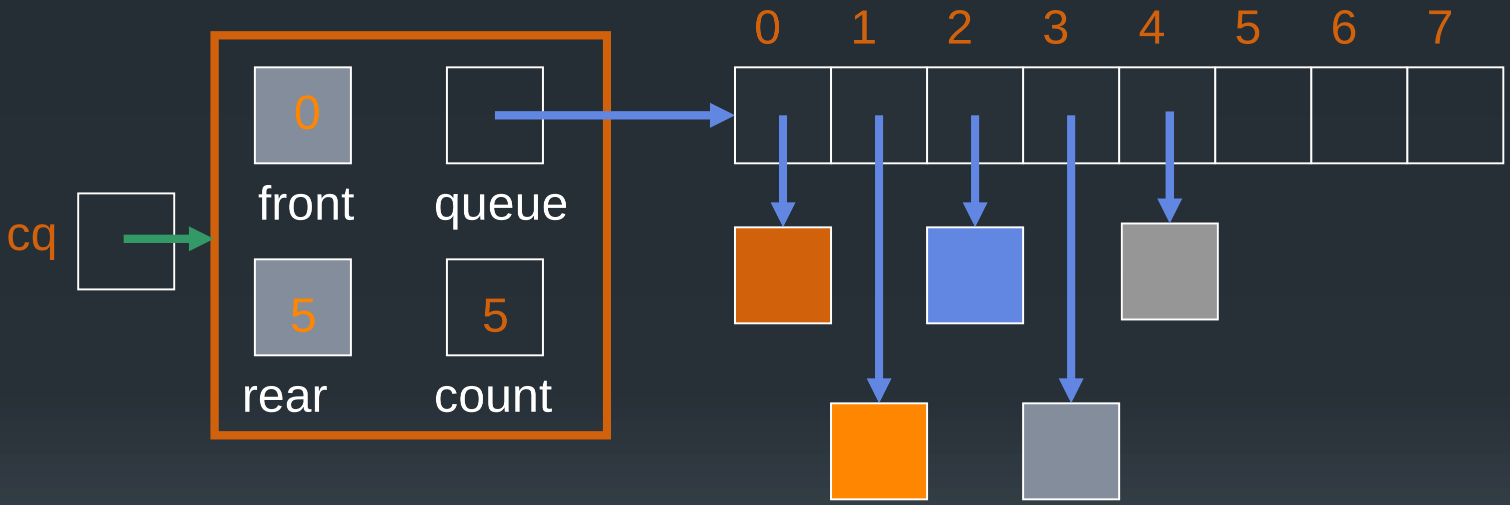


Better: copy the queue elements in order to the *beginning* of the new array



New element is added at rear = (rear+1) % queue.length

See *expandCapacity()* in *CircularArrayQueue.java*



Operations of C. Queue

```
class CircularQueue():
      # constructor
    def __init__(self, size): # initializing the class
        self.size = size

        # initializing queue with none
        self.queue = [None for i in range(size)]
        self.front = self.rear = -1

    def enqueue(self, data):
        # condition if queue is full
        if ((self.rear + 1) % self.size == self.front):
            print(" Queue is Full\n")
        # condition for empty queue
        elif (self.front == -1):
            self.front = 0
            self.rear = 0
            self.queue[self.rear] = data
        else:
            # next position of rear
            self.rear = (self.rear + 1) % self.size
            self.queue[self.rear] = data
    def dequeue(self):
        if (self.front == -1): # codition for empty queue
            print ("Queue is Empty\n")

        # condition for only one element
        elif (self.front == self.rear):
            temp=self.queue[self.front]
            self.front = -1
            self.rear = -1
            return temp
        else:
            temp = self.queue[self.front]
            self.front = (self.front + 1) % self.size
            return temp
```

```
def display(self):

        # condition for empty queue
        if(self.front == -1):
            print ("Queue is Empty")

        elif (self.rear >= self.front):
            print("Elements in the circular queue are:",
                                              end = " ")
            for i in range(self.front, self.rear + 1):
                print(self.queue[i], end = " ")
            print ()

        else:
            print ("Elements in Circular Queue are:",
                                            end = " ")
            for i in range(self.front, self.size):
                print(self.queue[i], end = " ")
            for i in range(0, self.rear + 1):
                print(self.queue[i], end = " ")
            print ()

        if ((self.rear + 1) % self.size == self.front):
            print("Queue is Full")
  # Driver Code
ob = CircularQueue(5)
ob.enqueue(14)
ob.enqueue(22)
ob.enqueue(13)
ob.enqueue(-6)
ob.display()
print ("Deleted value = ", ob.dequeue())
print ("Deleted value = ", ob.dequeue())
ob.display()
ob.enqueue(9)
ob.enqueue(20)
ob.enqueue(5)
ob.display()
```

Output:

```
Elements in Circular Queue are: 14 22 13 -6
Deleted value = 14 Deleted value = 22
Elements in Circular Queue are: 13 -6
Elements in Circular Queue are: 13 -6 9 20 5
Queue is Full
```

Data Structure

Dr. Itimad Raheem Ali

Email : itimadra@uoitc.edu.iq

Lecture 5

Recursion

- Sometimes, the best way to solve a problem is by solving a <u>smaller version</u> of the exact same problem first

- Recursion is a technique that solves a problem by solving a <u>smaller problem</u> of the same type

Recursion vs. iteration

- Iteration can be used in place of recursion
  - An iterative algorithm uses a *looping construct*
  - A recursive algorithm uses a *branching structure*
- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code

How do I write a recursive function?

- Determine the <u>size factor</u>
- Determine the <u>base case(s)</u>

    (the one for which you know the answer)
- Determine the <u>general case(s)</u>

    (the one where the problem is expressed as a smaller version of itself)
- Verify the algorithm

    (use the "Three-Question-Method")

Three-Question Verification Method

1. The Base-Case Question:

   Is there a nonrecursive way out of the function, and does the routine work correctly for this "base" case?
2. The Smaller-Caller Question:

   Does each recursive call to the function involve a smaller case of the original problem, leading inescapably to the base case?
3. The General-Case Question:

   Assuming that the recursive call(s) work correctly, does the whole function work correctly?

Simple example on recursion works

```
# A Python 3 program to demonstrate working of recursion
def printFun(test):

        if (test < 1):
                return
        else:
                print(test, end=" ")
                printFun(test-1) # statement 2
                print(test, end=" ")
                return
# Driver Code
test = 3
printFun(test)
```

Output :

```
3  2  1
```

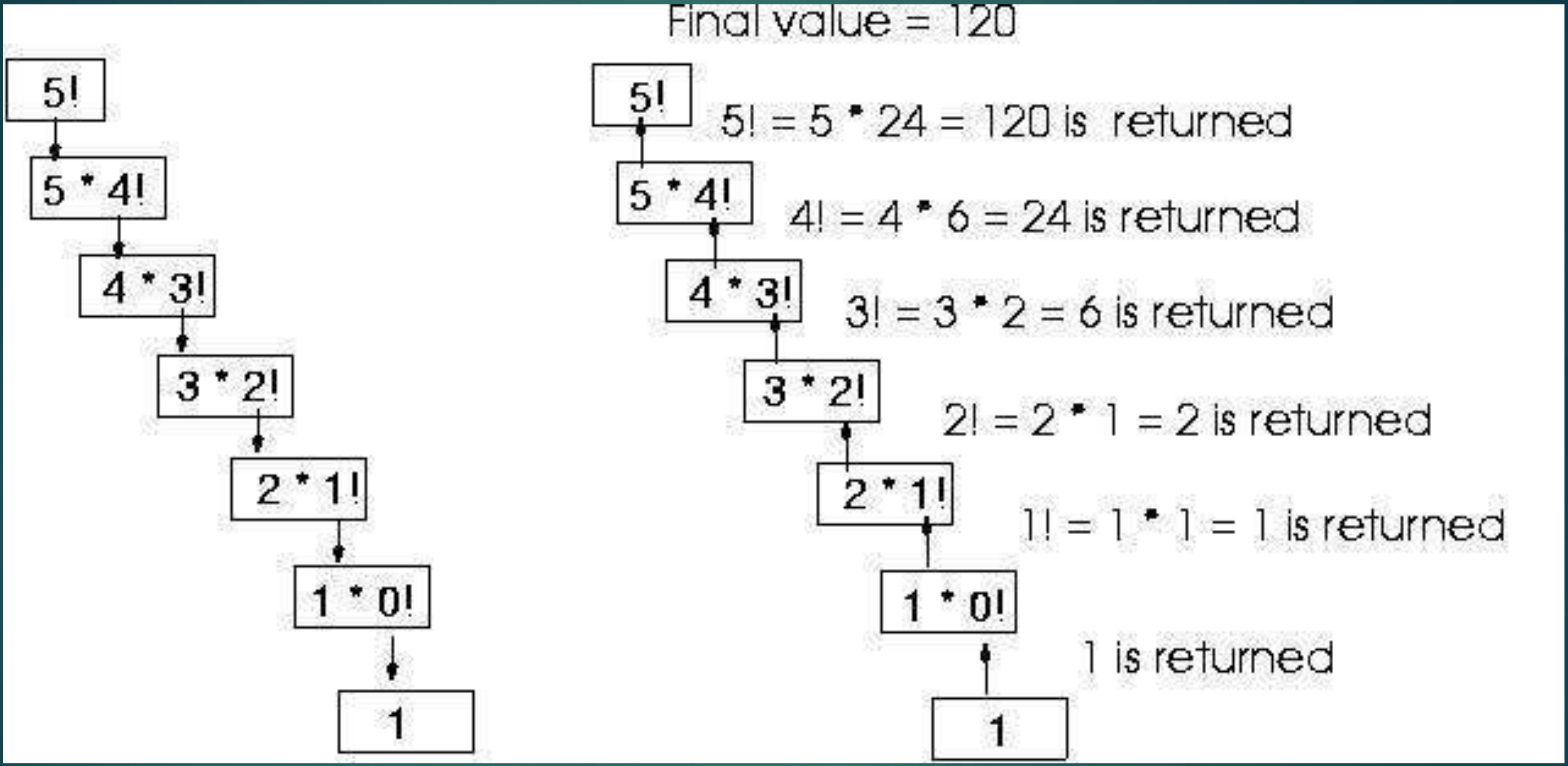
Problems defined recursively

- There are many problems whose solution can be defined recursively

  Example: *n factorial*

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)!*n & \text{if } n > 0 \end{cases} \quad \text{(\textit{recursive} solution)}$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ 1*2*3*\ldots*(n-1)*n & \text{if } n > 0 \end{cases} \quad \text{(\textit{closed form} solution)}$$


Final value = 120
5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1 * 0!
1
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1! = 1 * 1 = 1 is returned
1 * 0!
1 is returned
1

Example 1: Factorial function

```
# Using Recursive
# Python 3 program to find factorial of given
number

# Function to find factorial of given number
def factorial(n):

	if n == 0:
		return 1

	return n * factorial(n-1)

# Driver Code
num = 5;
print("Factorial of", num, "is",
factorial(num))
```

```
# using iterative
# Python 3 program to find factorial of given numb

# Function to find factorial of given number
def factorial(n):

	res = 1

	for i in range(2, n+1):
		res *= i
	return res

# Driver Code
num = 5;
print("Factorial of", num, "is",
factorial(num))
```

```
Output:
Factorial of 5 is 120
```

Example 2: Power Function

POWER

Recurrence: $x^n = (x)(x^{n-1})$

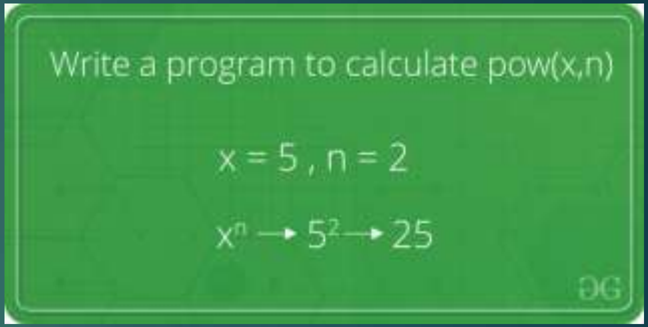
Example: $2^3 = (2)(2^2) = (2)(2)(2) = 8$

Base case: $x^1 = x$

Example 2: Power Function

Example 2: Power Function

```
def power(x, y):
        if y == 0:
            return 1
        if y >= 1:
            return x * power(x, y - 1)
```


Write a program to calculate pow(x,n)

$x = 5, n = 2$

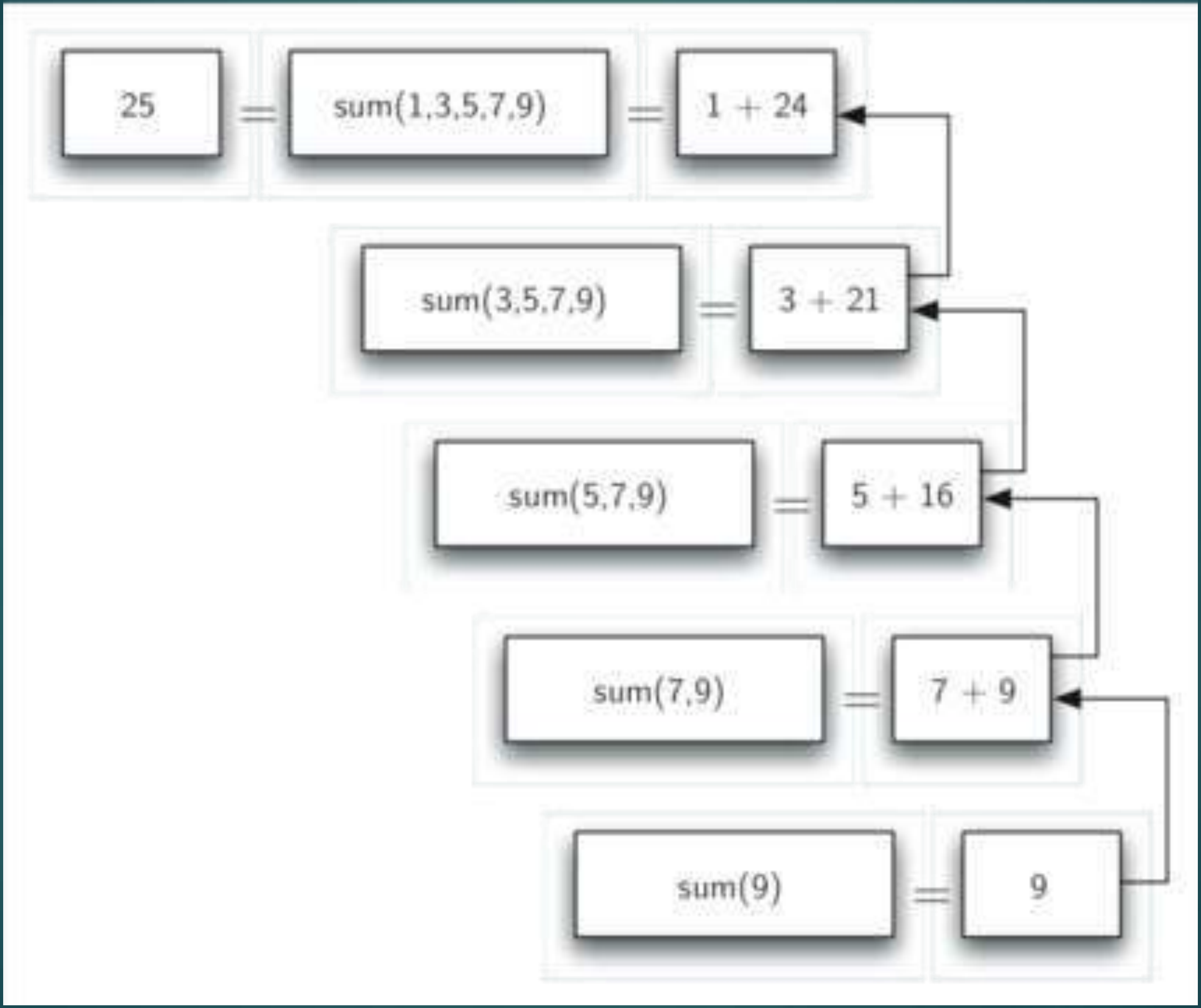
$x^n \rightarrow 5^2 \rightarrow 25$

```
def power(N, P):
# if power is 0 or 1 then number is
returned
            if(P == 0 or P == 1):
            return N
            else:
            return (N*power(N, P-1))

# Driver program
N = 5
P = 2
print(power(N, P))
```

Output :

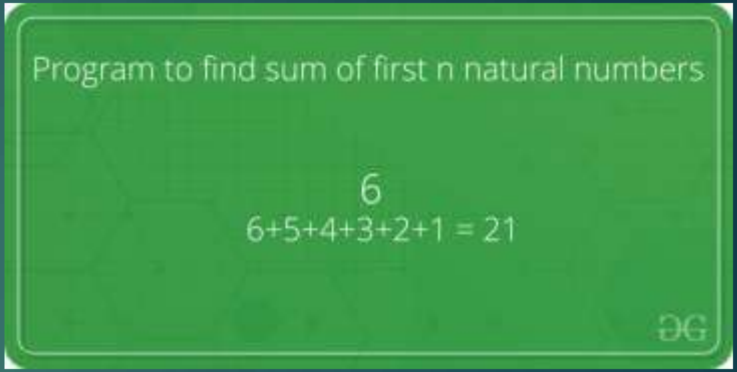
```
25
```

Example 3: Summation Function



Example 3: Summation Function

```
#Using Recursion
# Python code to find sum
# of natural numbers up to
# n using recursion

# Returns sum of first n natural numbers
def recurSum(n):
    if n <= 1:
        return n
    return n + recurSum(n - 1)

# Driver code
n = 5
print(recurSum(n))
```



```
Input : 3
Output : 6
Explanation : 1 + 2 + 3 = 6

Input : 5
Output : 15
Explanation : 1 + 2 + 3 + 4 + 5 = 15
```

Example 4: Multiplication Function

```
# Python3 to find Product of
# 2 Numbers using Recursion
# recursive function to calculate
# multiplication of two numbers
def product( x , y ):
        # if x is less than y swap
        # the numbers
        if x < y:
                return product(y, x)
        # iteratively calculate y
        # times sum of x
        elif y != 0:
                return (x + product(x, y - 1))

        # if any of the two numbers is
        # zero return zero
        else:
                return 0
# Driver code
x = 5
y = 2
print( product(x, y))
```

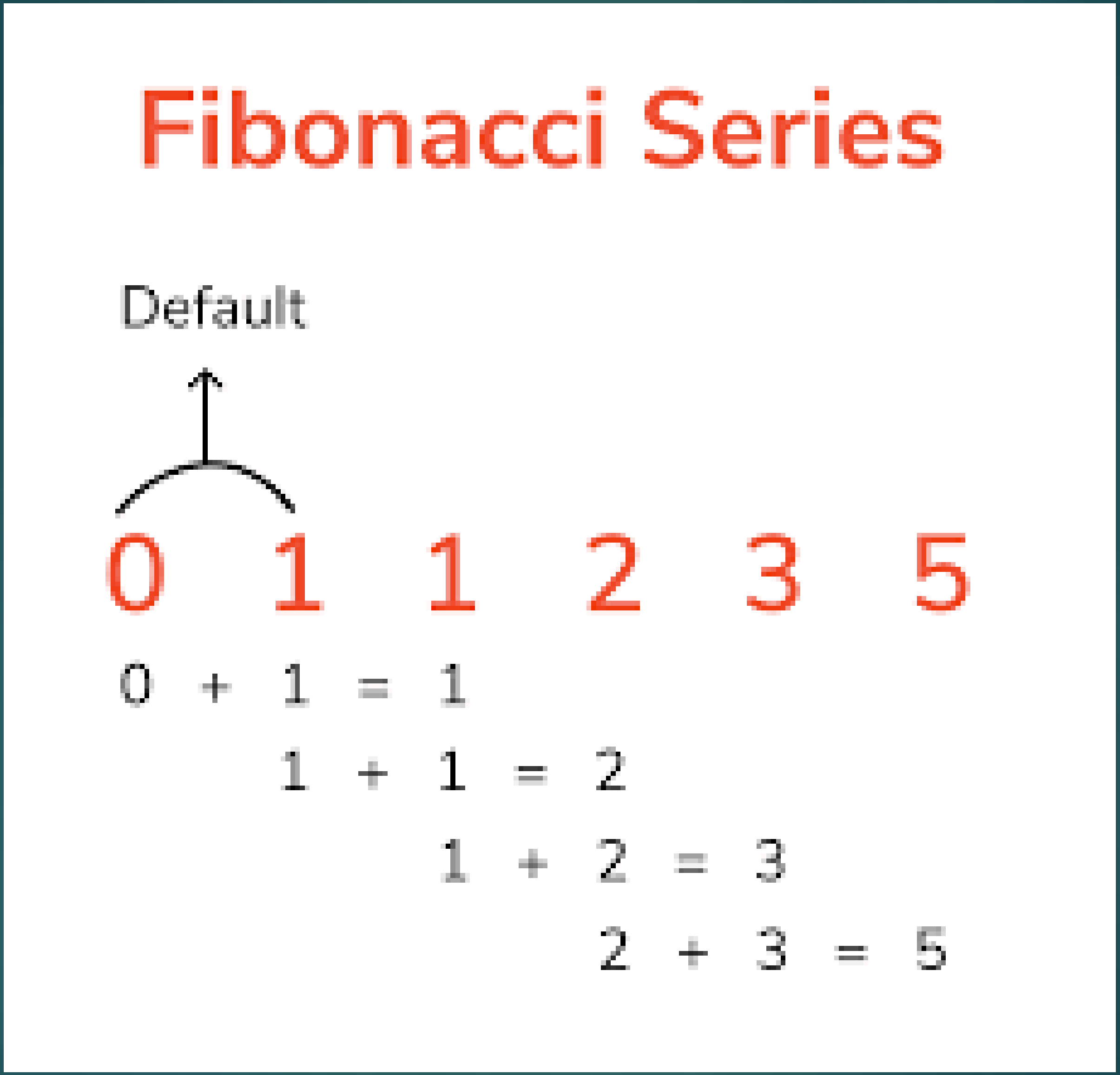
Output :`10`

mult2 (4,3)   return (n+mult2(n,m-1))

Return 4 +mult2 (4,2)   Return 4+8

Return 4 + mult2 (4,1)   Return 4+4

Return 4   Return 4



Fibonacci Series

Default

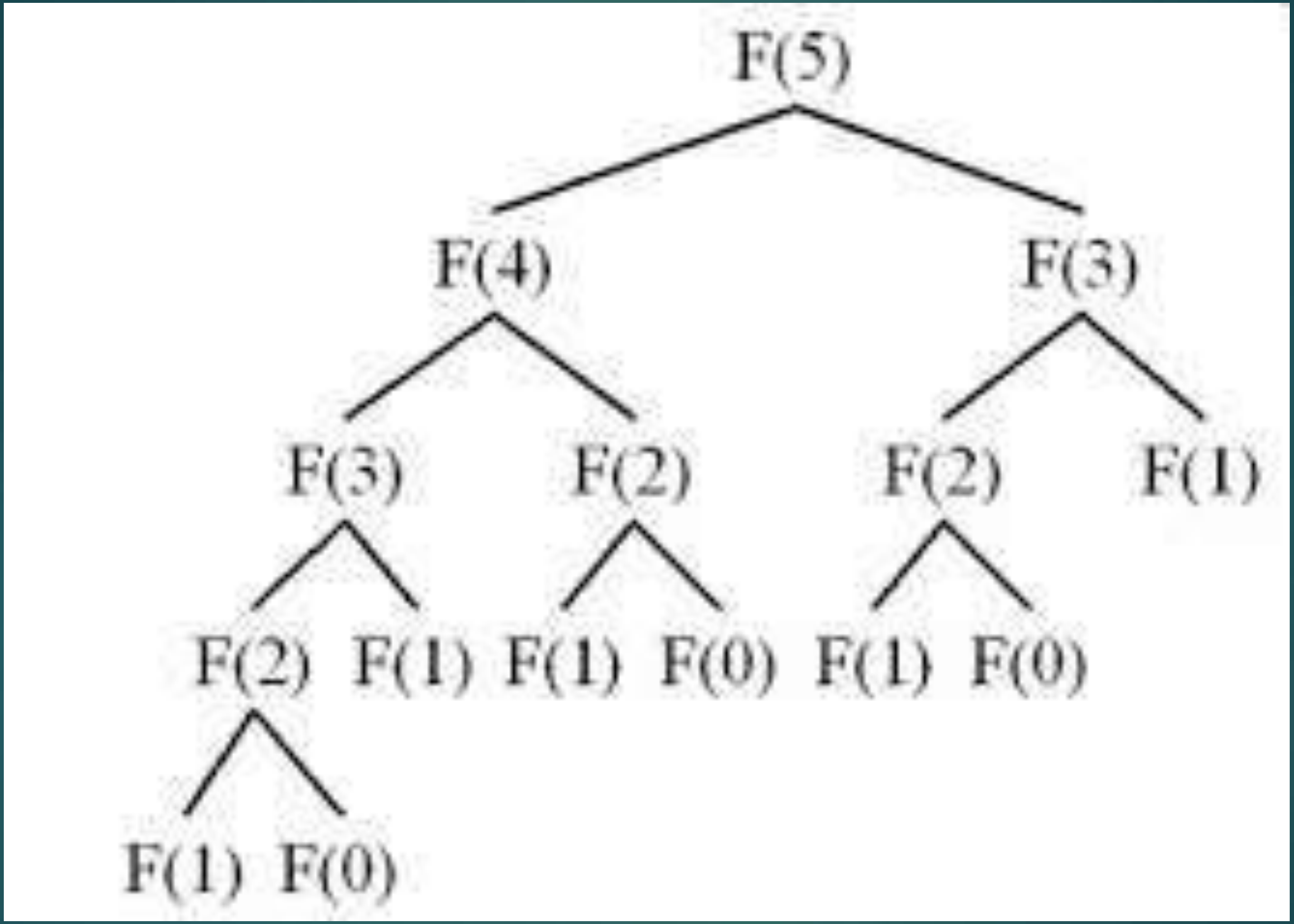
0 1 1 2 3 5

0 + 1 = 1

1 + 1 = 2

1 + 2 = 3

2 + 3 = 5


F(5)
F(4)
F(3)
F(3)
F(2)
F(2)
F(1)
F(2)
F(1)
F(1)
F(0)
F(1)
F(0)
F(1)
F(0)

Example 5: Fibonacci Function
0,1,1,2,3,5,8,13,21,34,55….

```
def recur_fibo(n):
    if n <= 1:
        return n
    else:
        return(recur_fibo(n-1) + recur_fibo(n-2))

n = 10 # check if the number of terms is valid
if n <= 0:
    print("Plese enter a positive integer")
else:
    print("Fibonacci sequence:")
for i in range(n):
    print(recur_fibo(i))
```

Output
Fibonacci sequence :

```
0
1
 1
 2
 3
 5
 8
 13
 21
 34
```

Reverse Example

```
def reverse(str):
    if str == "":
        return str   # or return “”
    else:
        return reverse(str[1:]) + str[0]

print reverse("reenigne")
```

Data Structure

Dr. Itimad Raheem Ali

Email : itimadra@uoitc.edu.iq

Lecture 6

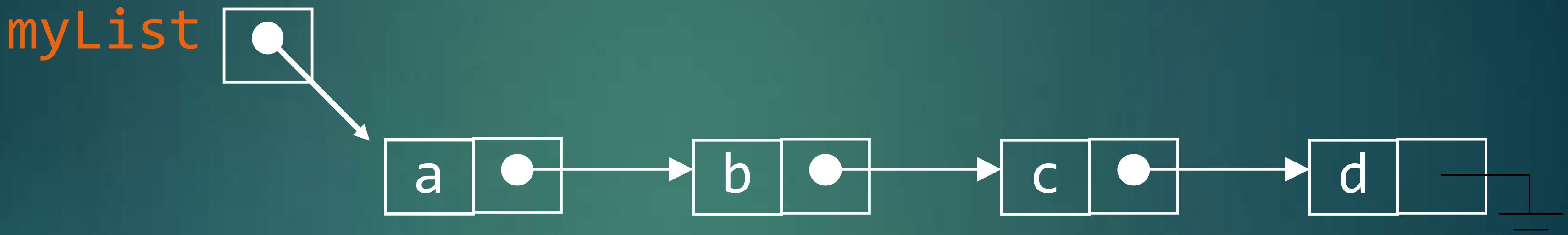
Linked List

What's wrong with Array ?

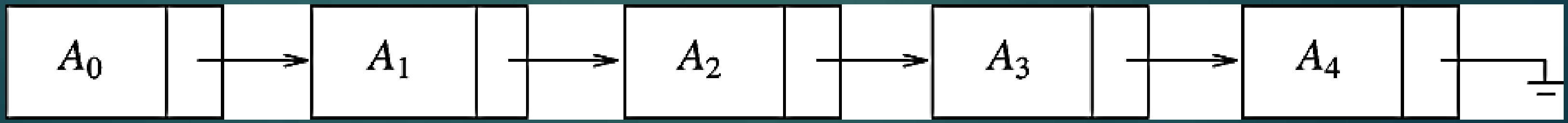
- Disadvantages of arrays as storage data structures:
  - slow searching in unordered array
  - slow insertion in ordered array
  - Fixed size
- Linked lists solve some of these problems
- Linked lists are general purpose storage data structures and are versatile.

Introduction

- A linked list consists of:
  - A sequence of nodes



Each node contains a value
and a link (pointer or reference) to some other node
The last node contains a null link
The list may (or may not) have a header

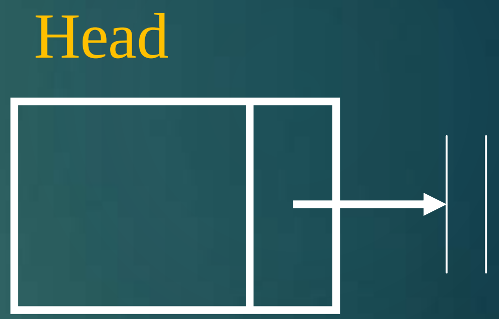


The Work to Add the Node

- Create the new node
- Fill in the data field
- Deal with the next field
  - Point to nil (if inserting to end)
  - Point to current (front or middle)

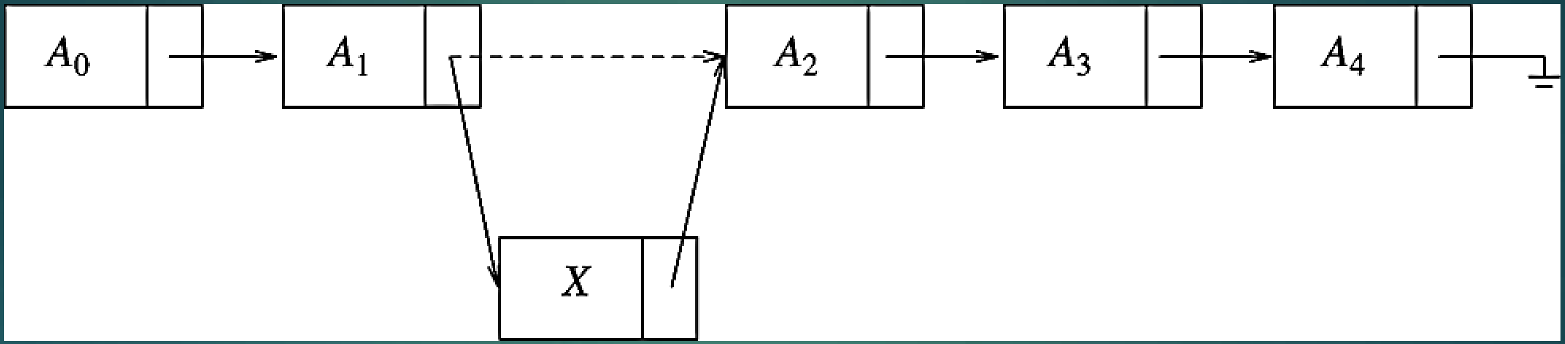
The Node Definition

```
# Node class
class Node:

# Function to initialize the node object
  def __init__(self, data):
    self.data = data  # Assign data
    self.next = None  # Initialize
                      # next as null

# Linked List class
class LinkedList:

# Function to initialize the Linked List object
  def __init__(self):
    self.head = None
```

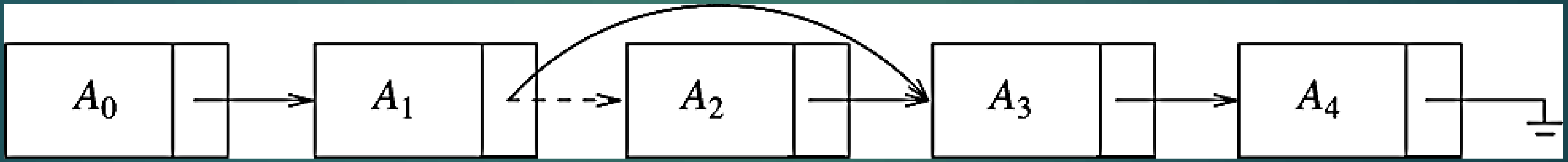


Implementation of a Linked List

Insertion

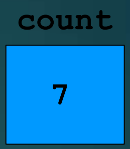


Deletion



Pointer Variable Declarations and Initialization

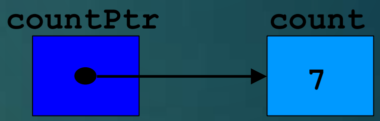
- Pointer variables
  - Contain memory addresses as values
  - Normally, variable contains specific value (direct reference)
  - Pointers contain address of variable that has specific value (indirect reference)



- Pointer declarations
  - ***** indicates variable is pointer

```
int *myPtr;
```

declares pointer to **int**, pointer of type **int ***



  - Multiple pointers require multiple asterisks

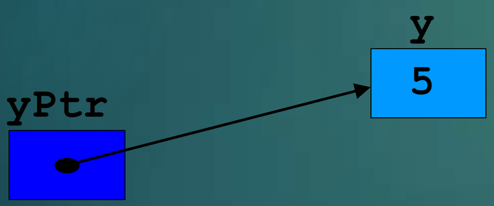
```
int *myPtr1, *myPtr2;
```

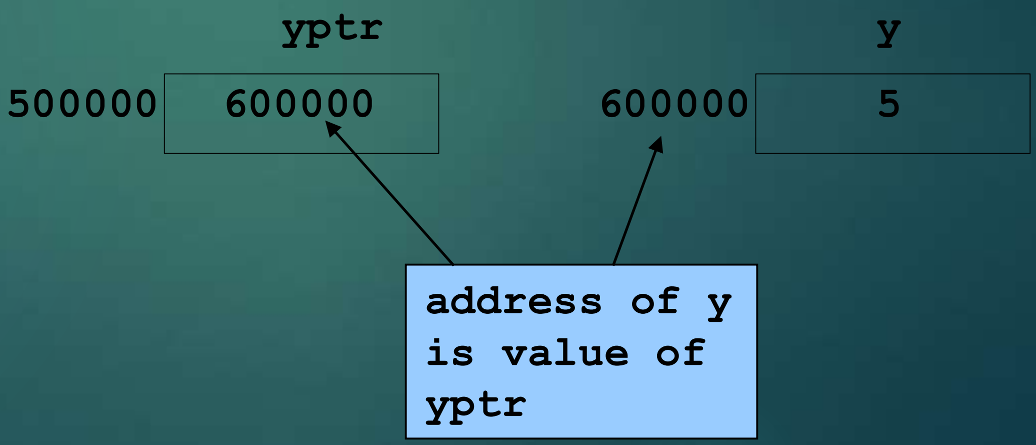
Pointer Operators

- `&` (address operator)
  - Returns memory address of its operand
  - Example

```
int y = 5;
 int *yPtr;
 yPtr = &y;     // yPtr gets address of y
```

  - `yPtr` "points to" `y`





Pointer Operators

- `*` (indirection/dereferencing operator)
  - Returns synonym for object its pointer operand points to
  - **`*yPtr`** returns **`y`** (because **`yPtr`** points to **`y`**).
  - dereferenced pointer is value

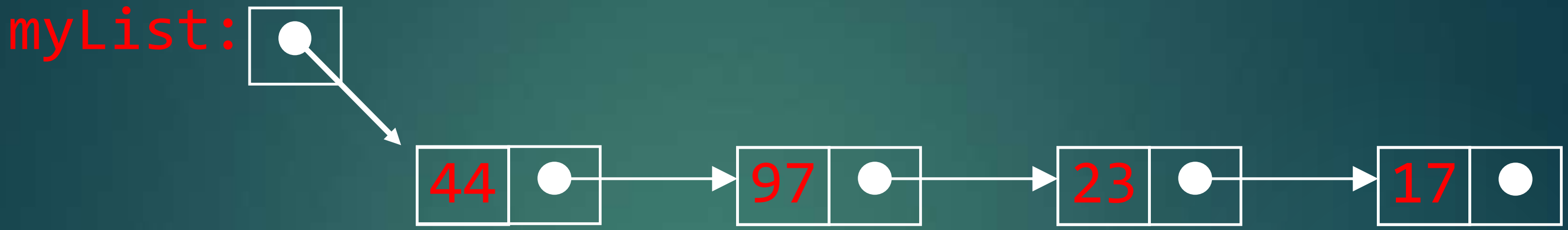
```
        *yptr = 9;      // assigns 9 to y
```

- `*` and **`&`** are inverses of each other

More terminology

- A node's successor is the next node in the sequence
- A node's predecessor is the previous node in the sequence
- A list's length is the number of elements in it
  - A list may be empty (contain no elements)

Creating links



```
# Node class
class Node:

    # Function to initialize the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize
# next as null
# Linked List class
class LinkedList:

    # Function to initialize the Linked
    # List object
    def __init__(self):
        self.head = None
```

Passing Arguments

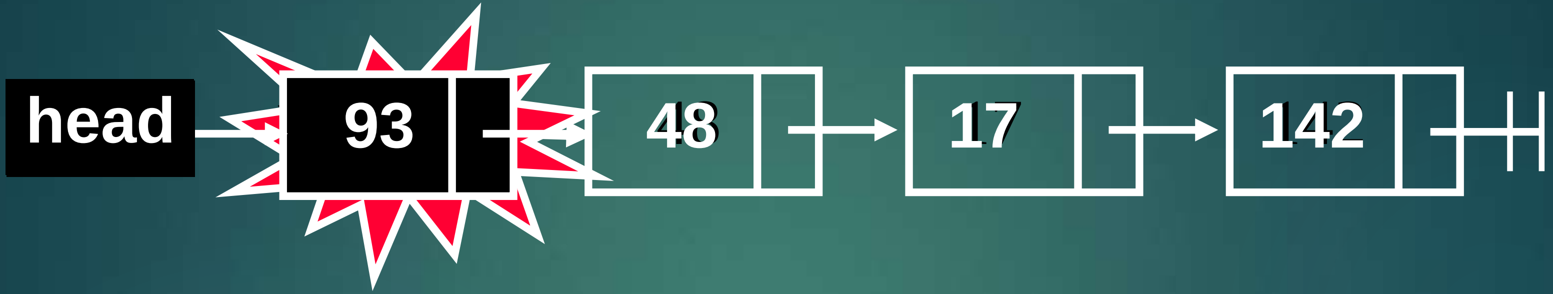
- 3 ways to pass arguments to function
  - Pass-by-value
  - Pass-by-reference with reference arguments
  - Pass-by-reference with pointer arguments
- `return` can return one value from function
- Arguments passed to function using reference arguments
  - Modify original values of arguments
  - More than one value “returned”

Passing Arguments

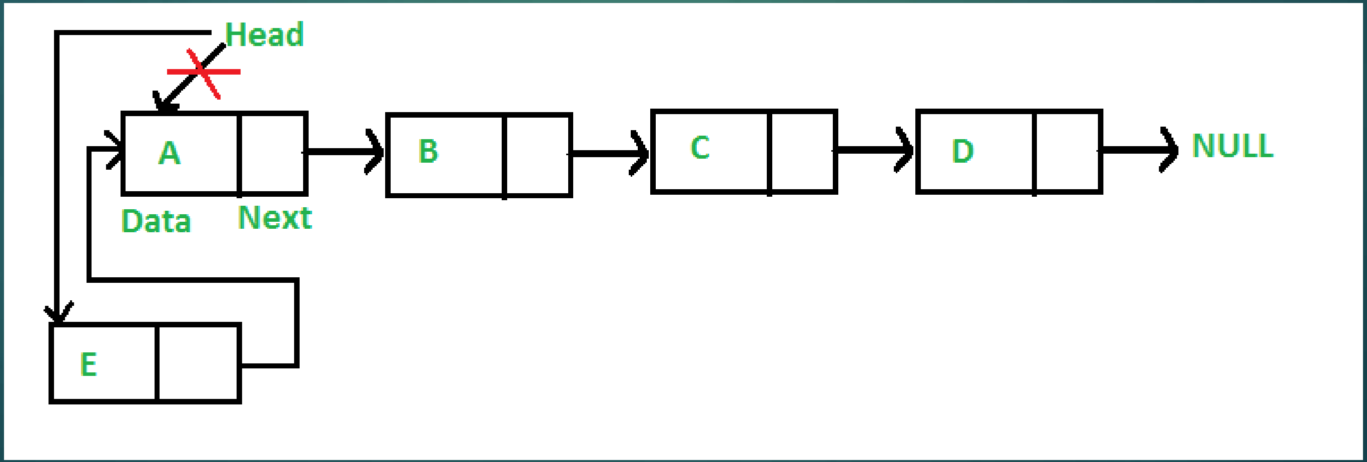
- Pass-by-reference with pointer arguments
  - Simulate pass-by-reference
    - Use pointers and indirection operator
  - Pass address of argument using **&** operator
  - Arrays not passed with **&** because array name already pointer
  - ***** operator used as alias/nickname for variable inside of function

Insert element to linked list

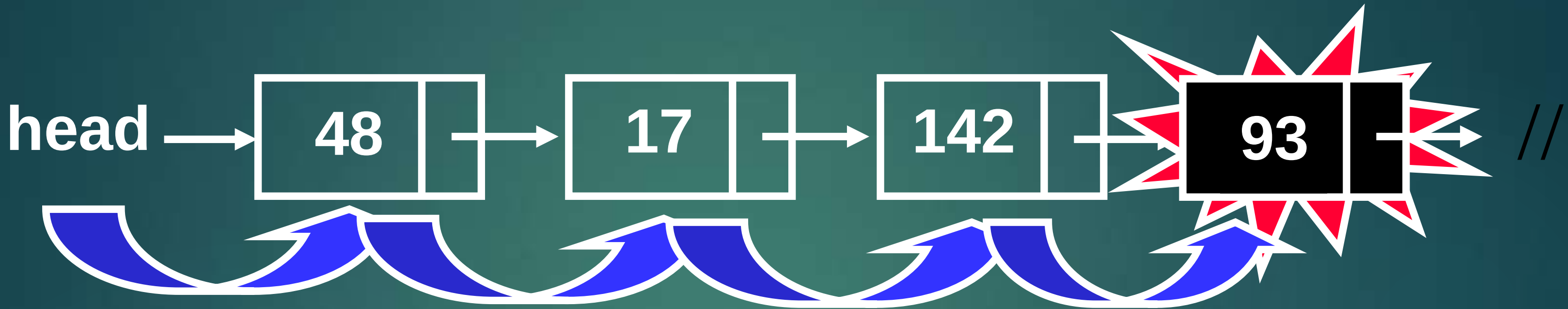
Inserting to the Front



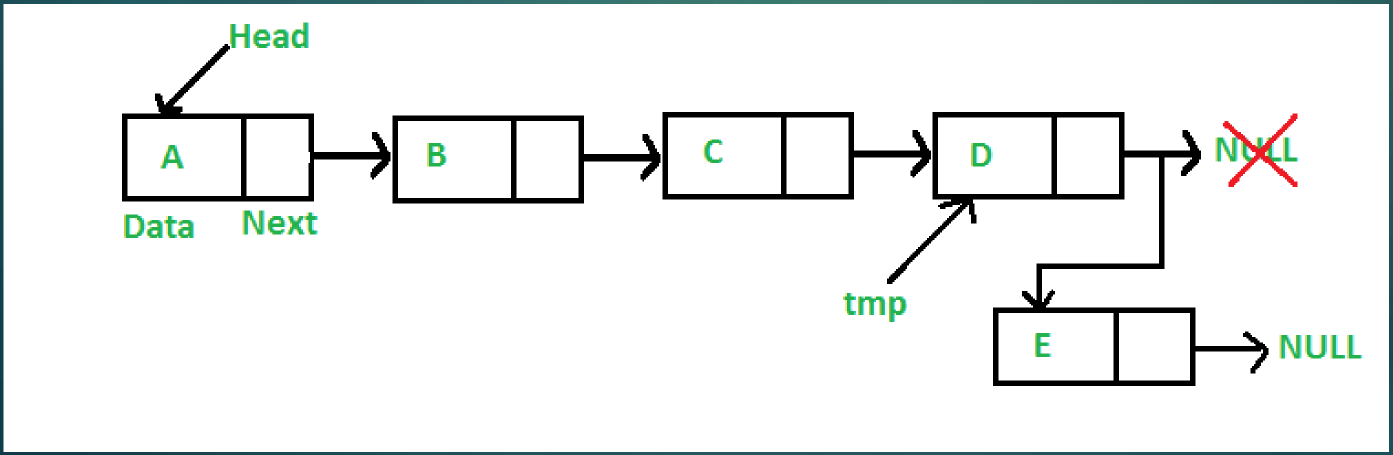
- There is no work to find the correct location
- Empty or not, head will point to the right location



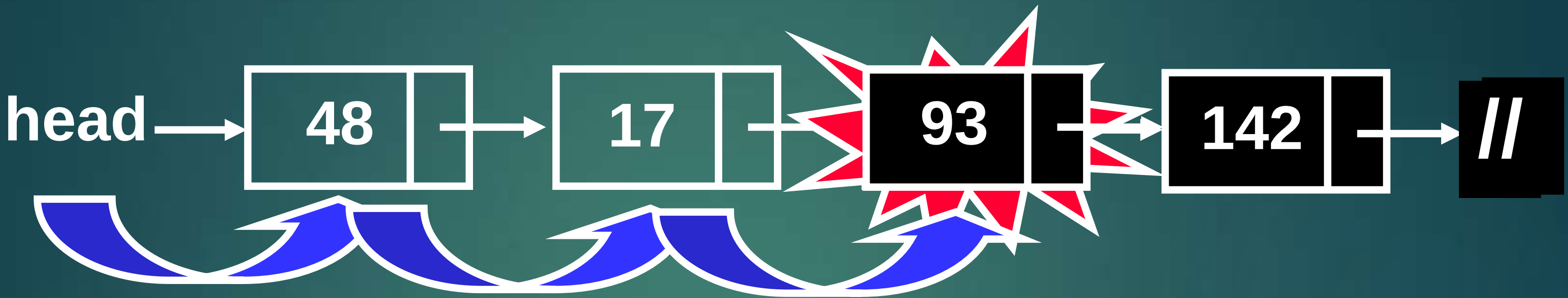
Inserting to the End



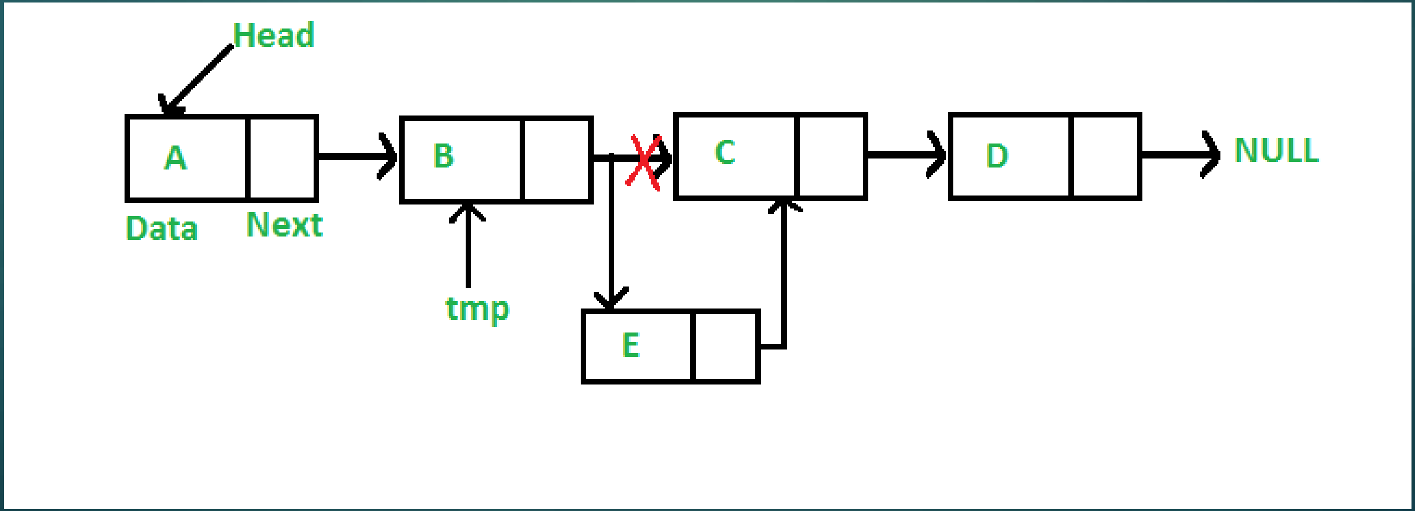
- Find the end of the list (when at NIL)



Inserting to the Middle

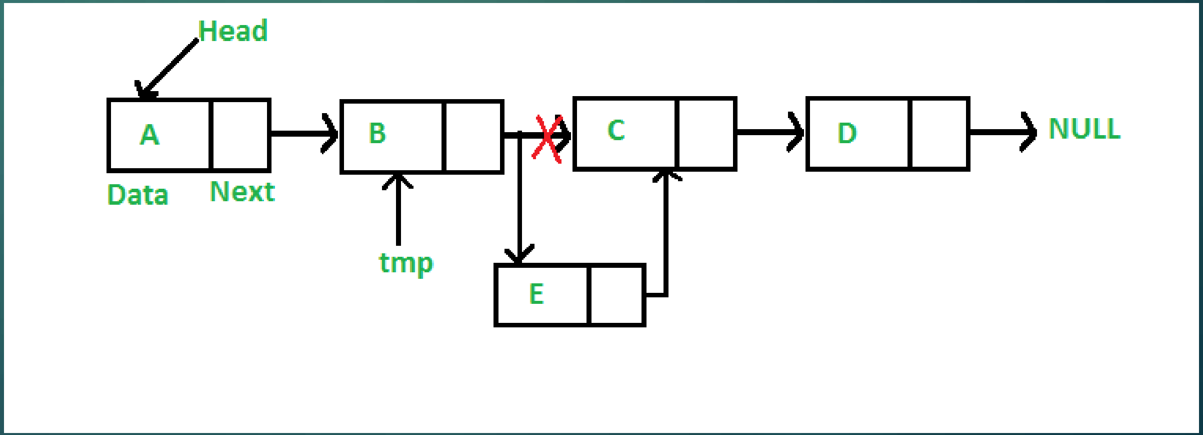


- Used when order is important
- Go to the node that should follow the one to add

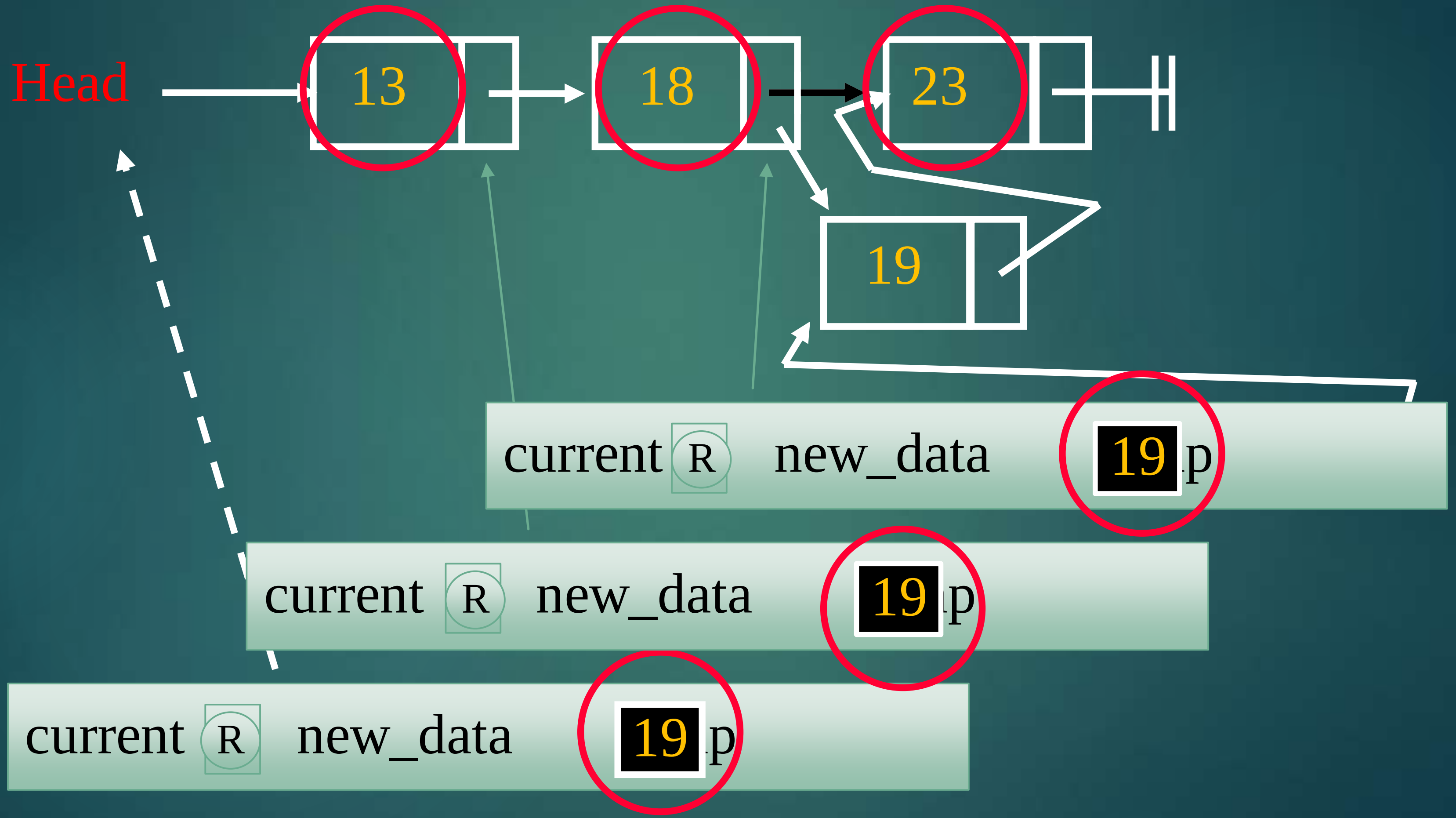


Inserting In Order into a Linked List

- Recursively traverse until you find the correct place to insert
  - Compare to see if you need to insert before current
  - If adding largest value, then insert at the end
- Perform the commands to do the insertion
  - Create new node
  - Add data
  - Update the next pointer of the new node
  - Update current to point to new node



Inserting In Order into a Linked List



```
# Node class
class Node:
# Function to initialise the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize next as null
# Linked List class contains a Node object
class LinkedList:
    # Function to initialize head
    def __init__(self):
        self.head = None
    # Functio to insert a new node at the beginning
    def push(self, new_data):
        # 1 & 2: Allocate the Node &
        #       Put in the data
        new_node = Node(new_data)
        # 3. Make next of new Node as head
        new_node.next = self.head
        # 4. Move the head to point to new Node
        self.head = new_node


 # Code execution starts here
 if __name__=='__main__':
     # Start with the empty list
     llist = LinkedList()
     # Insert 6. So linked list becomes 6->None
     llist.append(6)
     # Insert 7 at the beginning. So linked list becomes 7->6->None
     llist.push(7);
     # Insert 1 at the beginning. So linked list becomes 1->7->6->None
     llist.push(1);
     # Insert 4 at the end. So linked list becomes 1->7->6->4->None
     llist.append(4)
     # Insert 8, after 7. So linked list becomes 1 -> 7-> 8-> 6-> 4-> None
     llist.insertAfter(llist.head.next, 8)
     print 'Created linked list is:',
     llist.printList()
```

Output:

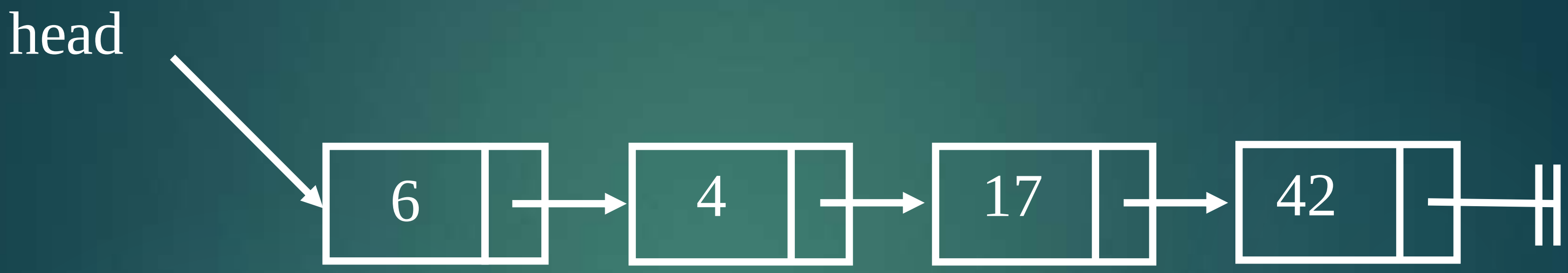
```
Created Linked list is: 1 7 8 6 4
```

```
# This function is in LinkedList class. Inserts a new node after the given prev_node.
This method is defined inside LinkedList class shown above */
    def insertAfter(self, prev_node, new_data):
        # 1. check if the given prev_node exists
        if prev_node is None:
            print "The given previous node must inLinkedList."
            return
        # 2. create new node &      # 3 Put in the data
        new_node = Node(new_data)
        # 4. Make next of new Node as next of prev_node
        new_node.next = prev_node.next
        # 5. make next of prev_node as new_node
        prev_node.next = new_node
# This function is defined in Linked List class Appends a new node at the end.
# defined inside LinkedList class shown above */
    def append(self, new_data):
        # 1. Create a new node
        # 2. Put in the data
        # 3. Set next as None
        new_node = Node(new_data)
        # 4. If the Linked List is empty, then make the
        # new node as head
        if self.head is None:
            self.head = new_node
            return
        # 5. Else traverse till the last node
        last = self.head
        while (last.next):
            last = last.next
        # 6. Change the next of last node
        last.next = new_node
# Utility function to print the linked list
    def printList(self):
        temp = self.head
        while (temp):
            print temp.data,
            temp = temp.next
```

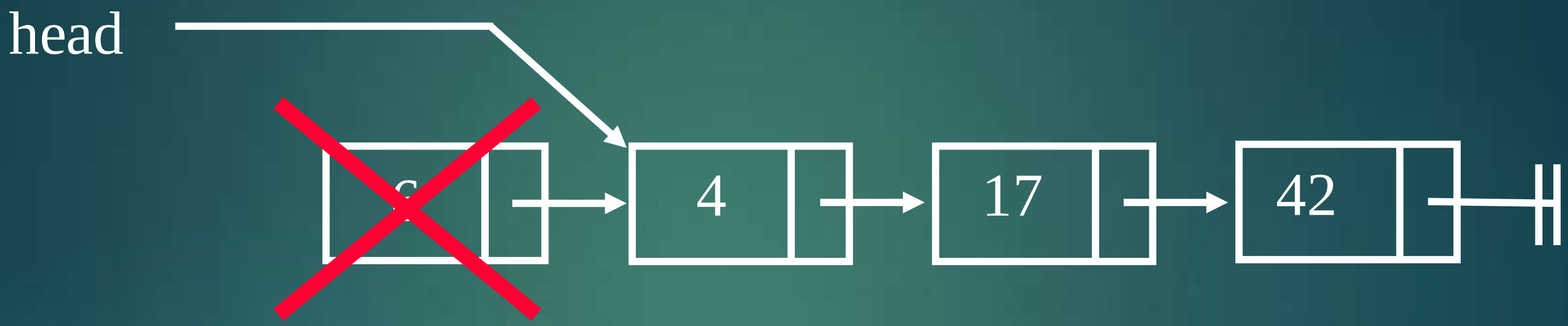
Summary

- Inserting into a linked list involves two steps:
  - Find the correct location
  - Do the work to insert the new value
- We can insert into any position
  - Front
  - End
  - Somewhere in the middle
    (to preserve order)

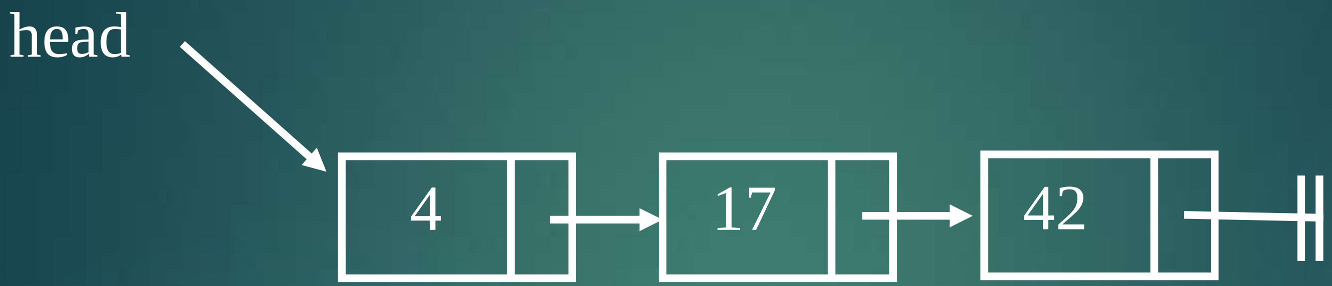
Deleting an Element from a Linked List



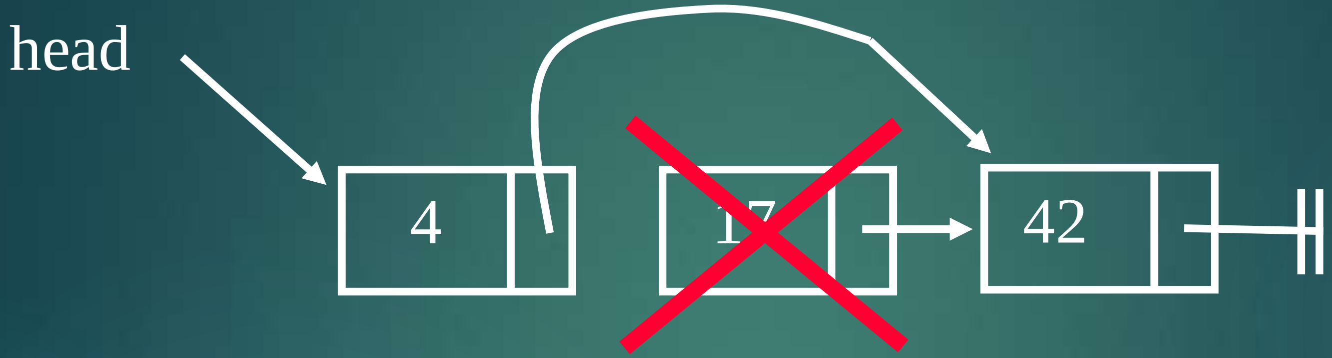
- **Begin with an existing linked list**
  - **Could be empty or not**
  - **Could be ordered or not**



- **Begin with an existing linked list**
  - Could be empty or not
  - Could be ordered or not



- Begin with an existing linked list
  - Could be empty or not
  - Could be ordered or not



- **Begin with an existing linked list**
  - Could be empty or not
  - Could be ordered or not

Finding the Match

- **We'll examine three situations:**
  - **Delete the first element**
  - **Delete the first occurrence of an element**
  - **Delete all occurrences of a particular element**

Deleting the First Element

```
# Python3 program to remove first node of  linked list.
import sys

# Link list node
class Node:
        # Function to removdef __init__(self, data):
                self.data = data
                self.next = None
# Delete the first node of the linked list
def removeFirstNode(head):
        if not head:
                return None
        temp = head
        # Move the head pointer to the next node
        head = head.next
        temp = None
        return head
# Function to push node at head
def push(head, data):
        if not head:
                return Node(data)
        temp = Node(data)
        temp.next = head
        head = temp
        return head
```

```
# Driver code
if __name__=='__main__':

        # Start with the empty list
        head = None
# Use push() function to construct  the below list
# 8 -> 23 -> 11 -> 29 -> 12
        head = push(head, 12)
        head = push(head, 29)
        head = push(head, 11)
        head = push(head, 23)
        head = push(head, 8)
        head = removeFirstNode(head)
                while(head):
                print("{} ".format(head.data), end ="")
                head = head.next
```

Examples:
```
Input : 1 -> 2 -> 3 -> 4 -> 5 -> NULL
Output : 2 -> 3 -> 4 -> 5 -> NULL
```

Output:23 11 29 12

Deleting from a Linked List

- Deletion from a linked list involves two steps:
  - Find a match to the element to be deleted (traverse until nil or found)
  - Perform the action to delete
- Performing the deletion is trivial:

```
current = current.next
```

  - This removes the element, since nothing will point to the node.

```
# A complete working Python3 program to  demonstrate deletion in singly linked list with class
# Node class
class Node:
    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None
class LinkedList:
    # Function to initialize head
    def __init__(self):
        self.head = None
    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node
    # Given a reference to the head of a list and a key,
    # delete the first occurence of key in linked list
    def deleteNode(self, key):
        # Store head node
        temp = self.head
        # If head node itself holds the key to be deleted
        if (temp is not None):
            if (temp.data == key):
                self.head = temp.next
                temp = None
                return
        while(temp is not None):
            if temp.data == key:
                break
            prev = temp
            temp = temp.next
        # if key was not present in linked list
        if(temp == None):
            return
        # Unlink the node from linked list
        prev.next = temp.next
        temp = None
    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print (" %d" %(temp.data)),
            temp = temp.next
# Driver program
llist = LinkedList()
llist.push(7)
llist.push(1)
llist.push(3)
llist.push(2)
print ("Created Linked List: ")
llist.printList()
llist.deleteNode(1)
print ("\nLinked List after Deletion of 1:")
llist.printList()
```

```
Created Linked List: 2 3 1 7
Linked List after Deletion of 1: 2 3 7
```

Deleting All Occurrences

- Deleting all occurrences is a little more difficult.

- Traverse the entire list and don’t stop until you reach nil.
- If you delete, recurse on current
- If you don’t delete, recurse on current .next

```
# Python3 program to delete all occurrences of a given key in linked list
# Link list node
class Node:
	def __init__(self, data):
		self.data = data
		self.next = None
# Given a reference (pointer to pointer) to the head of
# a list and an int, inserts a new node on the front of the list.
def push(head_ref, new_data):
	new_node = Node(0)
	new_node.data = new_data
	new_node.next = (head_ref)
	(head_ref) = new_node
	return head_ref
# Given a reference (pointer to pointer) to the head of a list and a key,
#deletes all occurrence of the given key
# in linked list
def deleteKey(head_ref, key):
	# Store head node
	temp = head_ref
	prev = None
	# If head node itself holds the key
	# or multiple occurrences of key
	while (temp != None and temp.data == key):
		head_ref = temp.next	# Changed head
		temp = head_ref			# Change Temp
	# Delete occurrences other than head
	while (temp != None):
		# Search for the key to be deleted,
		# keep track of the previous node
		# as we need to change 'prev.next'
		while (temp != None and temp.data != key):
			prev = temp
			temp = temp.next

		# If key was not present in linked list
		if (temp == None):
			return head_ref
		# Unlink the node from linked list
		prev.next = temp.next
		# Update Temp for next iteration of outer loop
		temp = prev.next
	return head_ref
# This function prints contents of linked list
# starting from the given node
def printList(node):
	while (node != None):
		print(node.data, end=" ")
		node = node.next
# Driver Code
if __name__ == '__main__':
	# Start with the empty list
	head = None
	head = push(head, 7)	head = push(head, 2)
	head = push(head, 3)	head = push(head, 2)
	head = push(head, 8)	head = push(head, 1)
	head = push(head, 2)	head = push(head, 2)
	key = 2 # key to delete
	print("Created Linked List: ")	printList(head)
	# Function call
	head = deleteKey(head, key)
	print("\nLinked List after Deletion is: ")
	printList(head)
```

Output

```
Created Linked List: 2 2 1 8 2 3 2 7
Linked List after Deletion is: 1 8 3 7
```

Delete all the nodes of singly linked list

```
# Python3 program to delete all  the nodes of singly linked list

# Node class
class Node:
	# Function to initialise the node object
	def __init__(self, data):
		self.data = data # Assign data
		self.next = None # Initialize next as null
# Constructor to initialize the node object
class LinkedList:
	# Function to initialize head
	def __init__(self):
		self.head = None
	def deleteList(self):
		# initialize the current node
		current = self.head
		while current:
			prev = current.next # move  next node
			# delete the current node
			del current.data
			# set current equals prev node
			current = prev

# push function to add node in front of llist
	def push(self, new_data):
		# Allocate the Node &
		# Put in the data
		new_node = Node(new_data)
		# Make next of new Node as head
		new_node.next = self.head
		# Move the head to point to new Node
		self.head = new_node
# Use push() to construct below  list 1-> 12-> 1-> 4-> 1
if __name__ == '__main__':

	llist = LinkedList()
	llist.push(1)
	llist.push(4)
	llist.push(1)
	llist.push(12)
	llist.push(1)
	print("Deleting linked list")
	llist.deleteList()

	print("Linked list deleted")
```

Output:
```
Deleting linked list
Linked list deleted
```

Summary

- Deletion involves:
  - Getting to the correct position
  - Moving a pointer so nothing points to the element to be deleted

- Can delete from any location
  - Front
  - First occurrence
  - All occurrences

Data Structure

Dr. Itimad Raheem Ali

Email : itimadra@uoitc.edu.iq

Lecture 7

Lists

Lists are just like dynamic sized arrays, declared in other languages (vector in C++ and ArrayList in Java).

Lists need not be homogeneous always which makes it a most powerful tool in Python.

A single list may contain DataTypes like Integers, Strings, as well as Objects.

Lists are mutable, and hence, they can be altered even after their creation.

List in Python are ordered and have a definite count. The elements in a list are indexed according to a definite sequence and the indexing of a list is done with 0 being the first index.

Each element in the list has its definite place in the list, which allows duplicating of elements in the list, with each element having its own distinct place and credibility.

Note- Lists are a useful tool for preserving a sequence of data and further iterating over it.

Creating a List

Lists in Python can be created by just placing the sequence inside the square brackets[]. Unlike Sets, list doesn’t need a built-in function for creation of list.

```
# Python program to demonstrate Creation of List

# Creating a List
List = []
print("Blank List: ")
print(List)
# Creating a List of numbers
List = [10, 20, 14]
print("\nList of numbers: ")
print(List)
```

```
# Creating a List of strings and accessing
using index
List = ["Geeks", "For", "Geeks"]
print("\nList Items: ")
print(List[0])
print(List[2])
# Creating a Multi-Dimensional List (By
Nesting a list inside a List)
List = [['Geeks', 'For'] , ['Geeks']]
print("\nMulti-Dimensional List: ")
print(List)
```

Output:

```
Blank List: []
List of numbers: [10, 20, 14]
List Items Geeks Geeks
Multi-Dimensional List: [['Geeks', 'For'],
['Geeks']]
```

```
# Creating a List with the use of Numbers
# (Having duplicate values)
List = [1, 2, 4, 4, 3, 3, 3, 6, 5]
print("\nList with the use of Numbers: ")
print(List)

# Creating a List with mixed type of values
# (Having numbers and strings)
List = [1, 2, 'Geeks', 4, 'For', 6, 'Geeks']
print("\nList with the use of Mixed Values: ")
print(List)
```

Output:

```
List with the use of Numbers: [1, 2, 4, 4, 3, 3, 3, 6, 5]
List with the use of Mixed Values: [1, 2, 'Geeks', 4, 'For', 6, 'Geeks']
```

Knowing the size of List

```
# Creating a List
List1 = []
print(len(List1))

# Creating a List of numbers
List2 = [10, 20, 14]
print(len(List2))
```

Adding Elements to a List

```
# Python program to demonstrate  Addition of elements in a List
# Creating a List
List = []
print("Initial blank List: ")
print(List)
# Addition of Elements in the List
List.append(1)
List.append(2)
List.append(4)
print("\nList after Addition of Three elements: ")
print(List)
# Adding elements to the List  using Iterator
for i in range(1, 4):
        List.append(i)
print("\nList after Addition of elements from 1-3: ")
print(List)
# Adding Tuples to the List
List.append((5, 6))
print("\nList after Addition of a Tuple: ")
print(List)
# Addition of List to a List
List2 = ['For', 'Geeks']
List.append(List2)
print("\nList after Addition of a List: ")
print(List)
```

Output:

```
Initial blank List: []
List after Addition of Three elements:
[1, 2, 4]

List after Addition of elements from 1-3:
[1, 2, 4, 1, 2, 3]

List after Addition of a Tuple:
[1, 2, 4, 1, 2, 3, (5, 6)]

List after Addition of a List:
[1, 2, 4, 1, 2, 3, (5, 6), ['For', 'Geeks']]
```

Removing Elements from the List

```
# Python program to demonstrate
# Removal of elements in a List

# Creating a List
List = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
print("Intial List: ")
print(List)

# Removing elements from List
# using Remove() method
List.remove(5)
List.remove(6)
print("\nList after Removal of two elements: ")
print(List)

# Removing elements from List
# using iterator method
for i in range(1, 5):
        List.remove(i)
print("\nList after Removing a range of elements: ")
print(List)
```

Output:

```
Intial List:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

List after Removal of two elements:
[1, 2, 3, 4, 7, 8, 9, 10, 11, 12]

List after Removing a range of elements:
[7, 8, 9, 10, 11, 12]
```

Iterate over a list in Python

List is equivalent to arrays in other languages, with the extra benefit of being dynamic in size.
In Python, the list is a type of container in Data Structures, which is used to store multiple data at the same time. Unlike Sets, lists in Python are ordered and have a definite count.
There are multiple ways to iterate over a list in Python.

Method #1: Using For loop

```
# Python3 code to iterate over a list
list = [1, 3, 5, 7, 9]

# Using for loop
for i in list:
        print(i)
```

Method #2: For loop and range()

```
# Python3 code to iterate over a list
list = [1, 3, 5, 7, 9]
# getting length of list
length = len(list)
# Iterating the index same
# as 'for i in range(len(list))'
for i in range(length):
        print(list[i])
```

Method #3: Using while loop

```
# Python3 code to iterate over a
# list
list = [1, 3, 5, 7, 9]

# Getting length of list
length = len(list)
i = 0

# Iterating using while loop
while i < length:
        print(list[i])
        i += 1
```

Iterations in Python

In the following iterations, the for loop is internally(we can't see it) using iterator object to traverse over the iterables

```
# Sample built-in iterators

# Iterating over a list
print("List Iteration")
l = ["geeks", "for", "geeks"]
for i in l:
        print(i)
# Iterating over a tuple (immutable)
print("\nTuple Iteration")
t = ("geeks", "for", "geeks")
for i in t:
        print(i)
```

```
# Iterating over a String
print("\nString Iteration")
s = "Geeks"
for i in s :
        print(i)

# Iterating over dictionary
print("\nDictionary Iteration")
d = dict()
d['xyz'] = 123
d['abc'] = 345
for i in d :
        print("%s %d" %(i, d[i]))
```

Output :

```
List Iteration
geeks
for
geeks
Tuple Iteration
geeks
for
geeks
String Iteration G e e k s
Dictionary Iteration
xyz 123 abc 345
```

Abstract vs. Concrete Data Types

- Abstract Data Type (ADT)
  - Mathematical description of an object and the set of operations on the object
    - List, Stack, Tree, Heap, Graph, …
  - One ADT may specialize another ADT
  - One ADT may implement another ADT
- Concrete Data Type
  - Implementation of an ADT using some set of *primitive* data types and operations of known complexity
    - Primitives: integers, arrays, pointers or references
  - Object-oriented programming languages (Java, C++) let you *explicitly* define new concrete data types that correspond to ADT's.

List ADT

$$(\; A_1 \;\; A_2 \;\; \ldots \;\; A_{n-1} \;\; A_n \;)$$

`length = n`

- Mathematical description: a *sequence* of items
  - A_i precedes A_{i+1} for $1 \le i < n$
- Operations
  - First() = position
  - Value(position) = item
  - Next(position) = position
  - Length() = integer
  - Insert(item,position)
  - Delete(position)

What other operations might be useful?

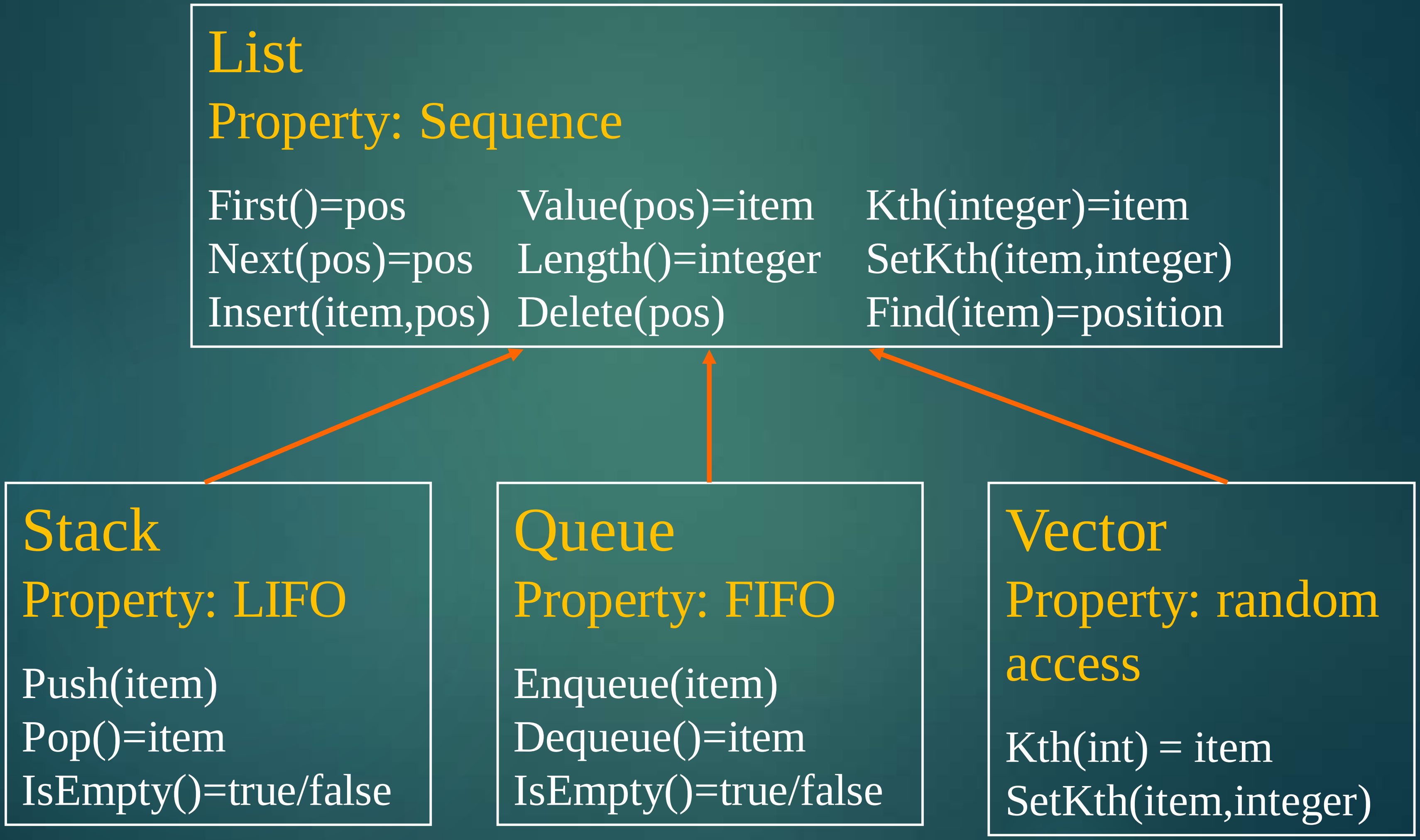
List ADT

$$(A_1 \ A_2 \ \ldots \ A_{n-1} \ A_n)$$

`length = n`

- Mathematical description: a *sequence* of items
  - A_i precedes A_{i+1} for $1 \leq i < n$
- Operations
  - First() = position
  - Value(position) = item
  - Next(position) = position
  - Length() = integer
  - Insert(item,position)
  - Delete(position)

Specialization Hierarchy



List
Property: Sequence

First()=pos
Next(pos)=pos
Insert(item,pos)
Value(pos)=item
Length()=integer
Delete(pos)
Kth(integer)=item
SetKth(item,integer)
Find(item)=position

Stack
Property: LIFO

Push(item)
Pop()=item
IsEmpty()=true/false

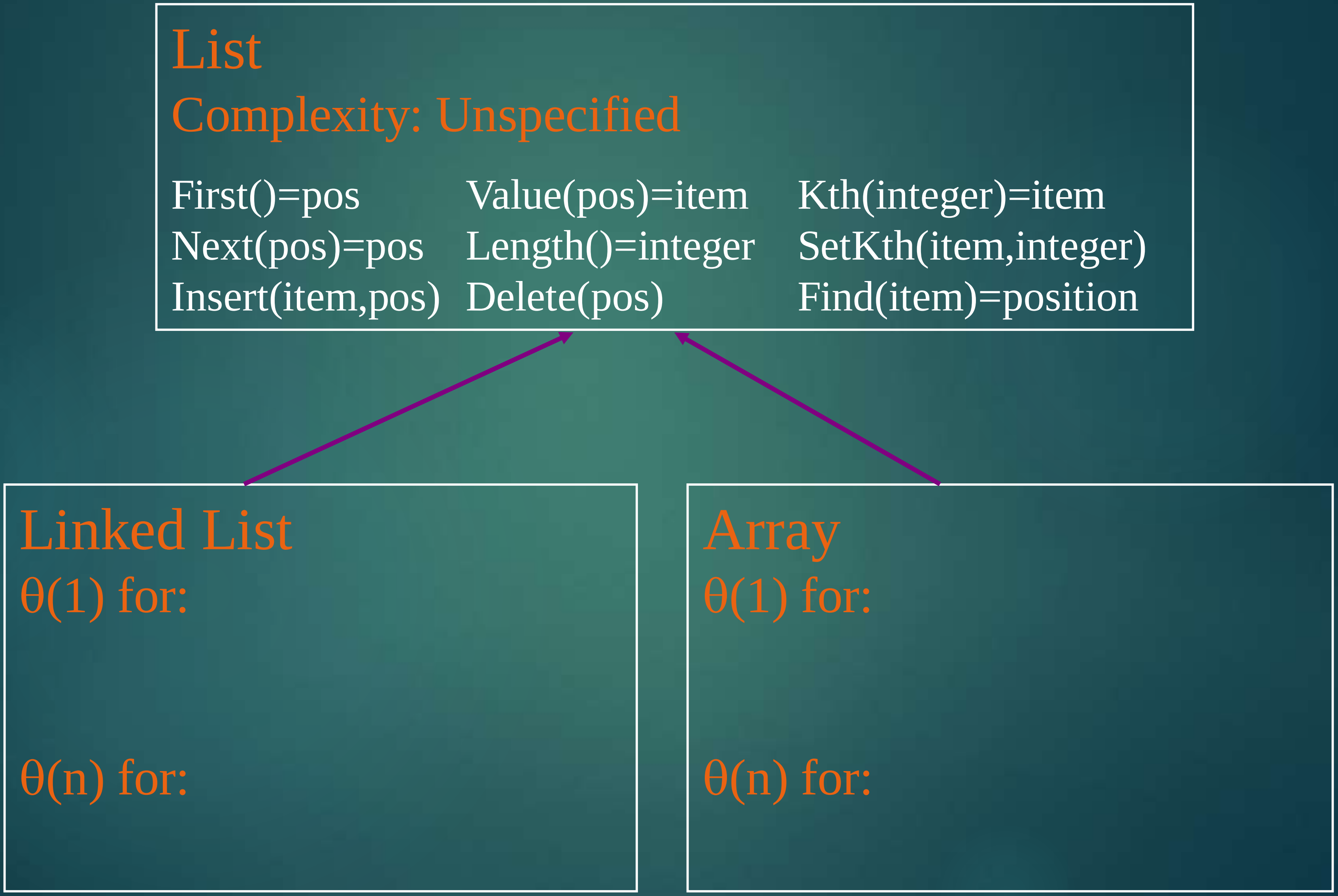
Queue
Property: FIFO

Enqueue(item)
Dequeue()=item
IsEmpty()=true/false

Vector
Property: random access

Kth(int) = item
SetKth(item,integer)

Implementation Hierarchy



List

Complexity: Unspecified

First()=pos	Value(pos)=item	Kth(integer)=item
Next(pos)=pos	Length()=integer	SetKth(item,integer)
Insert(item,pos)	Delete(pos)	Find(item)=position

Linked List

θ(1) for:

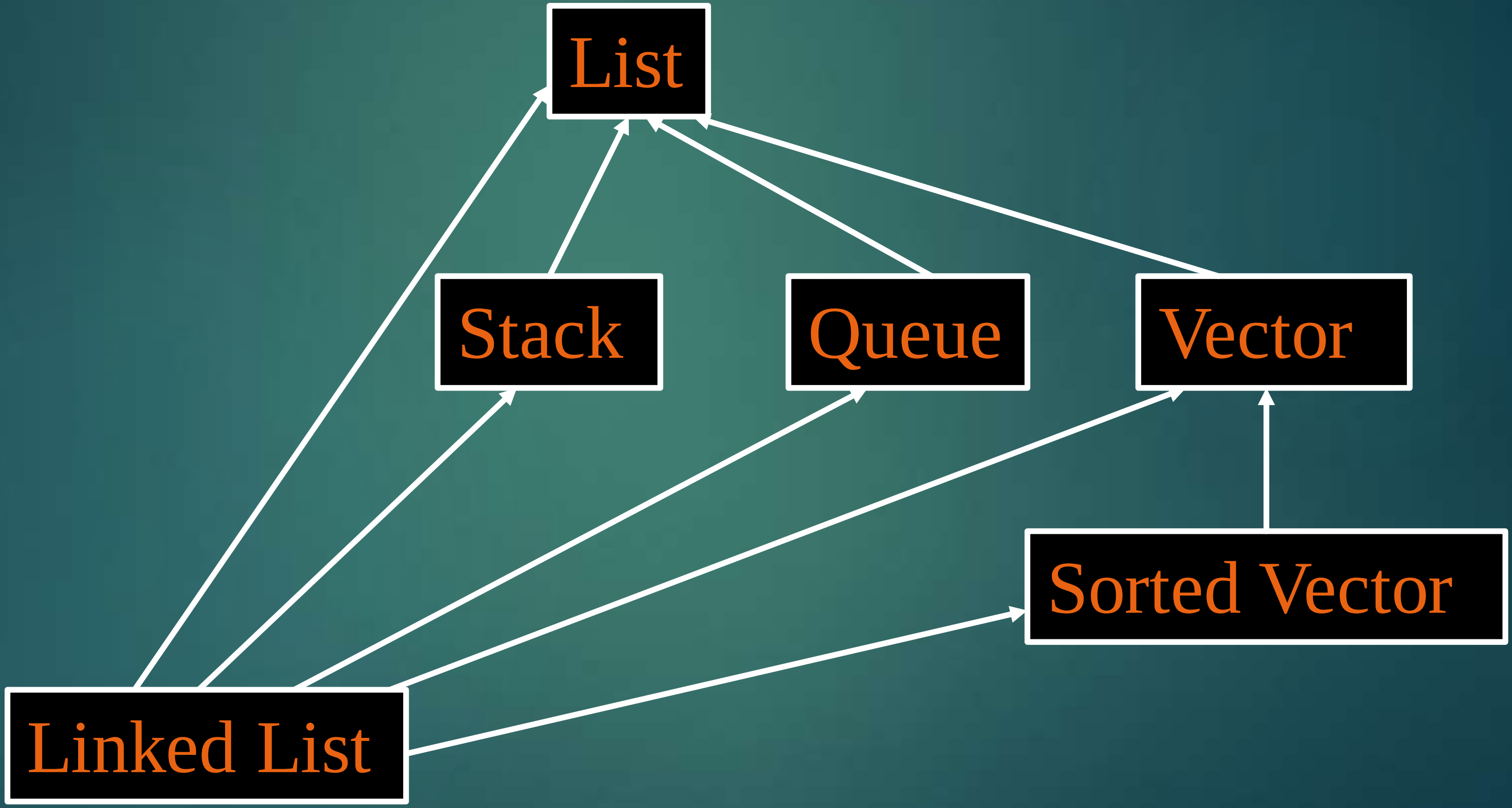
θ(n) for:

Array

θ(1) for:

θ(n) for:

Specialization and Implementation Hierarchies



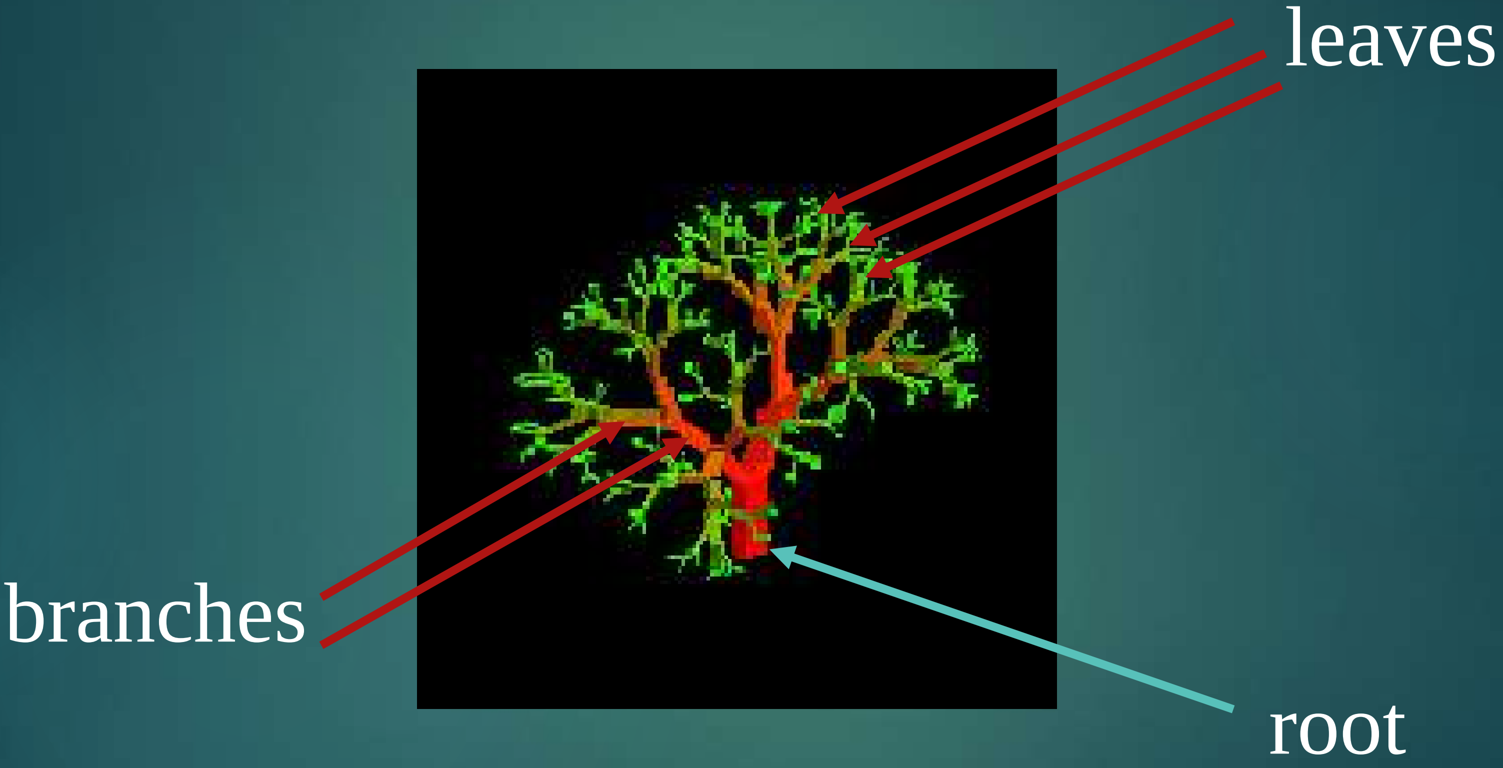
Data Structure

Dr. Itimad Raheem Ali

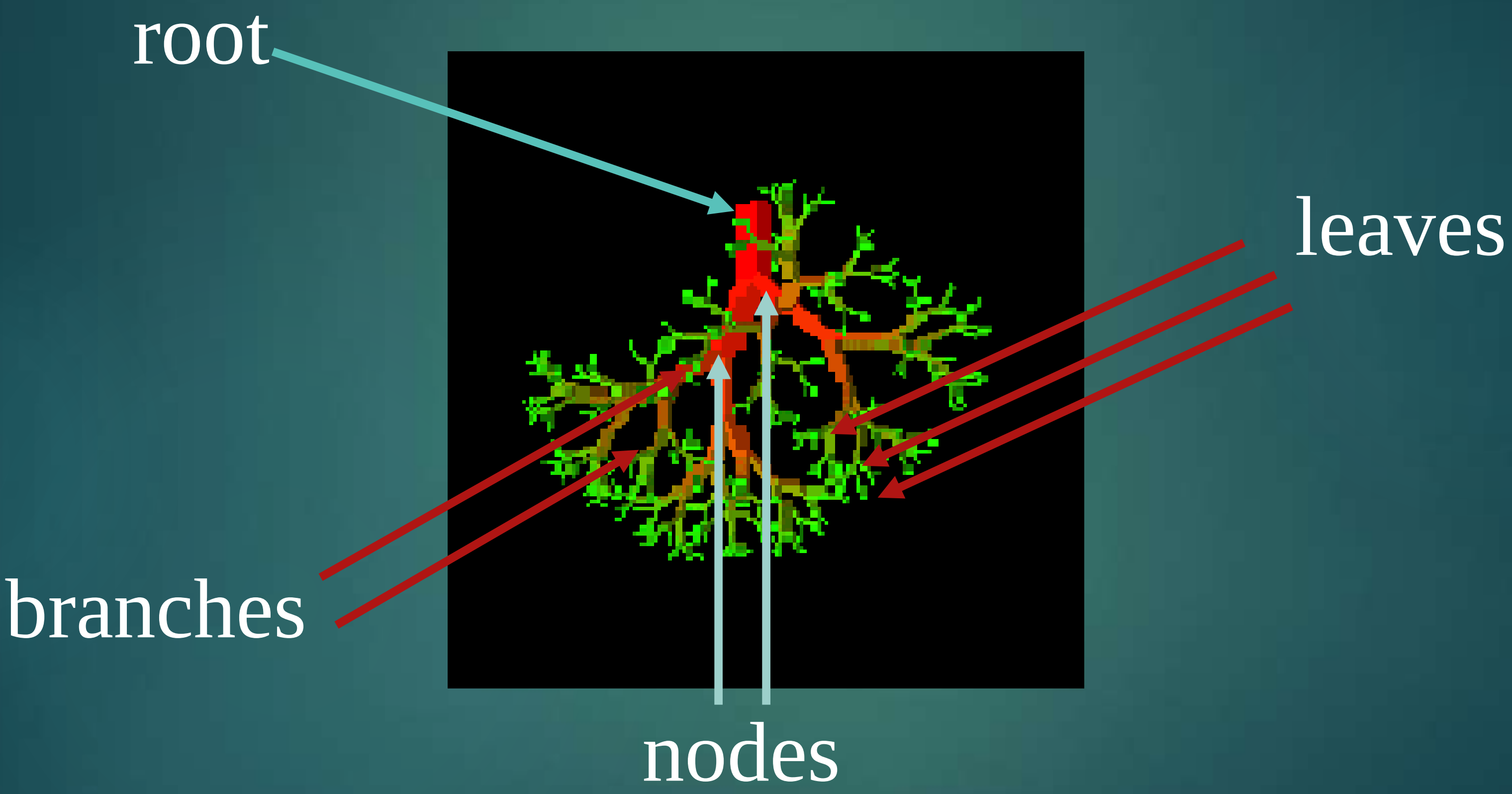
Email : itimadra@uoitc.edu.iq

Lecture 8+9

Nature View of a Tree

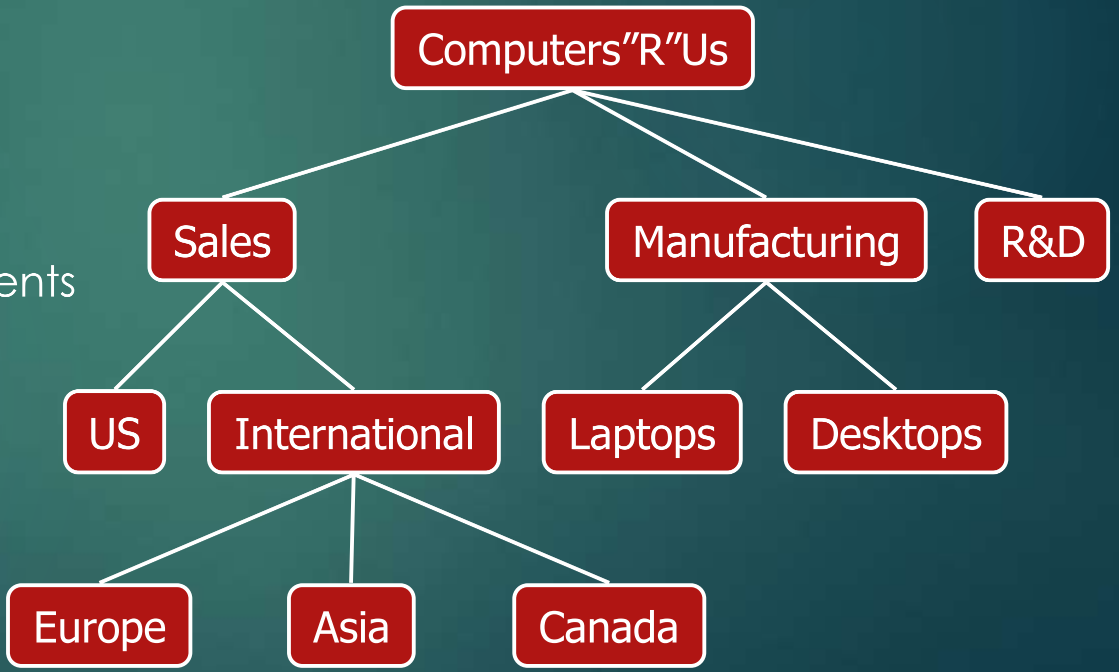


Computer Scientist’s View



What is a Tree

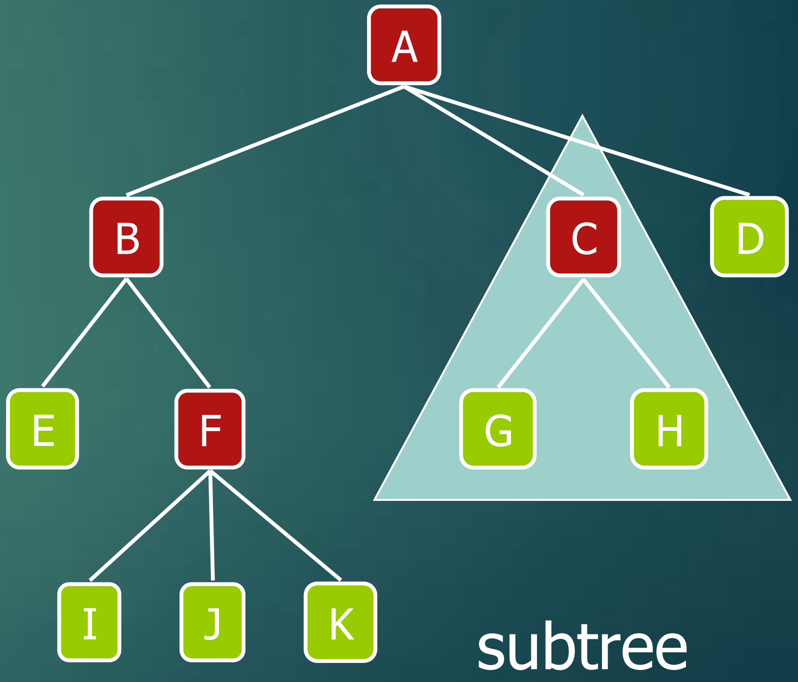
- A tree is a finite nonempty set of elements.
- It is an abstract model of a hierarchical structure.
- consists of nodes with a parent-child relation.
- Applications:
  - Organization charts
  - File systems
  - Programming environments



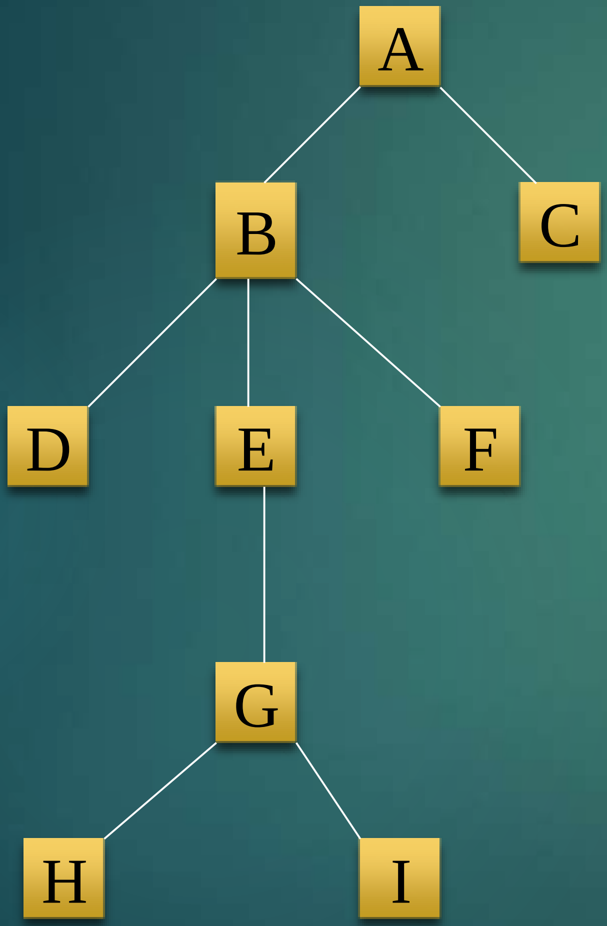
Tree Terminology

- **Root**: node without parent (A)
- **Siblings**: nodes share the same parent
- **Internal node**: node with at least one child (A, B, C, F)
- **External node** (leaf): node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node: parent, grandparent, grand-grandparent, etc.
- **Descendant** of a node: child, grandchild, grand-grandchild, etc.
- **Depth** of a node: number of ancestors
- **Height** of a tree: maximum depth of any node (3)
- **Degree** of a node: the number of children of this node
- **Degree** of a tree: the maximum degree among all node.

- **Subtree**: tree consisting of a node and its descendants



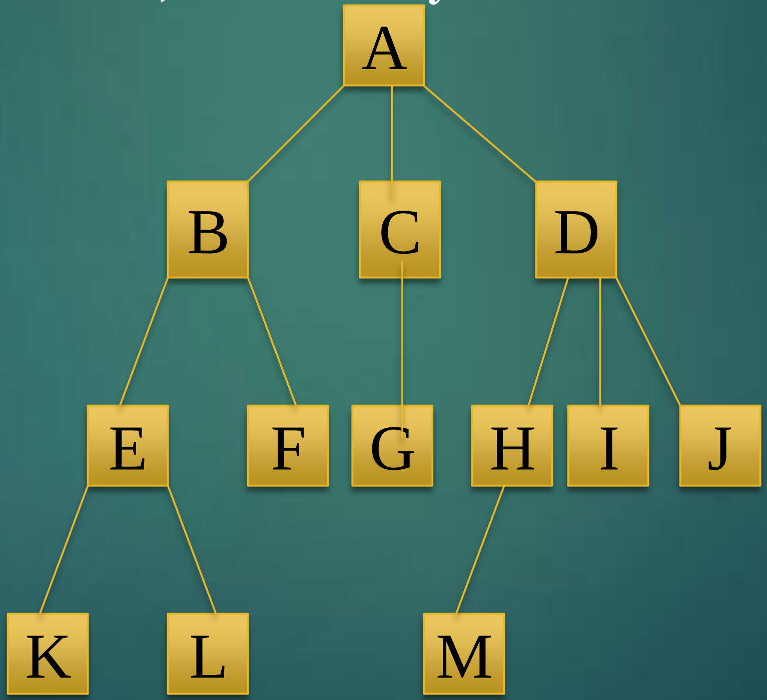
Tree Properties



Property	Value
Number of nodes	9
Height	4
Root Node	A
Leaves	C, D, F, H, I
Interior nodes	B, E, G
Ancestors of H	G, E, B, A
Descendants of B	D, E, F, G, H, I
Siblings of E	D, F
Right subtree of A	
Degree of this tree	3
Degree of node G	2

Intuitive Representation of Tree Node

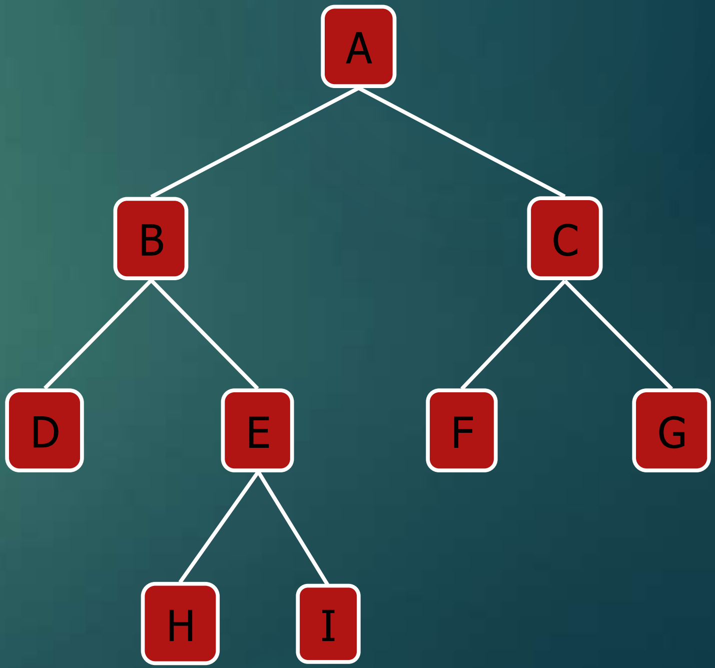
- List Representation
  - (A (B (E (K, L), F), C (G), D (H (M), I, J)))
  - The root comes first, followed by a list of links to sub-trees



Binary Tree

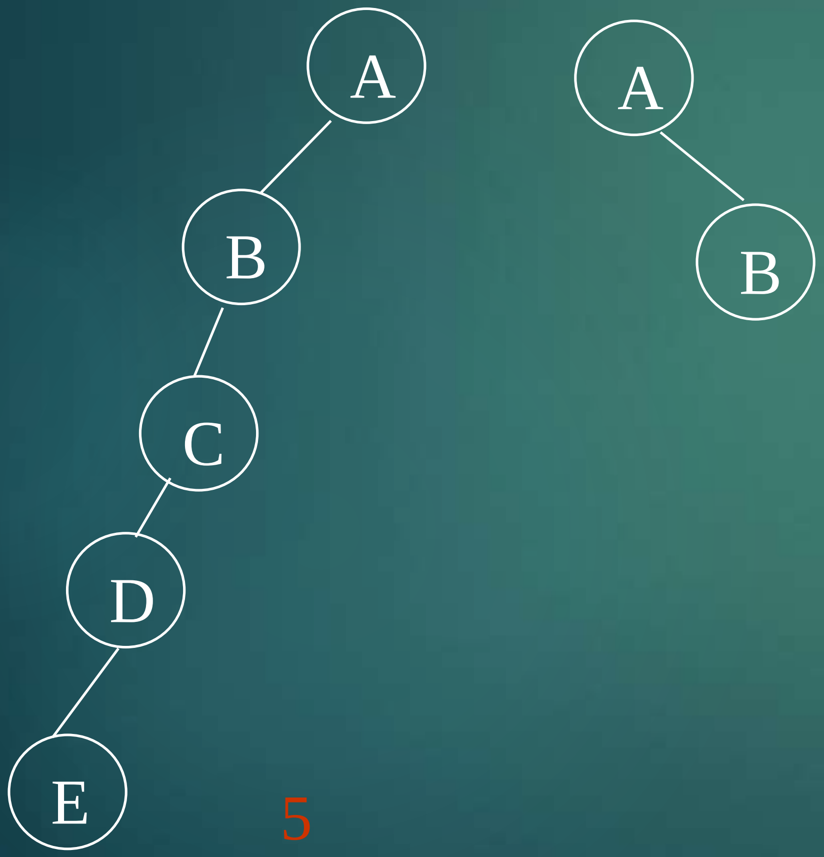
- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (degree of two)
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, OR
  - a tree whose root has an ordered pair of children, each of which is a binary tree

- Applications:
  - arithmetic expressions
  - decision processes
  - searching

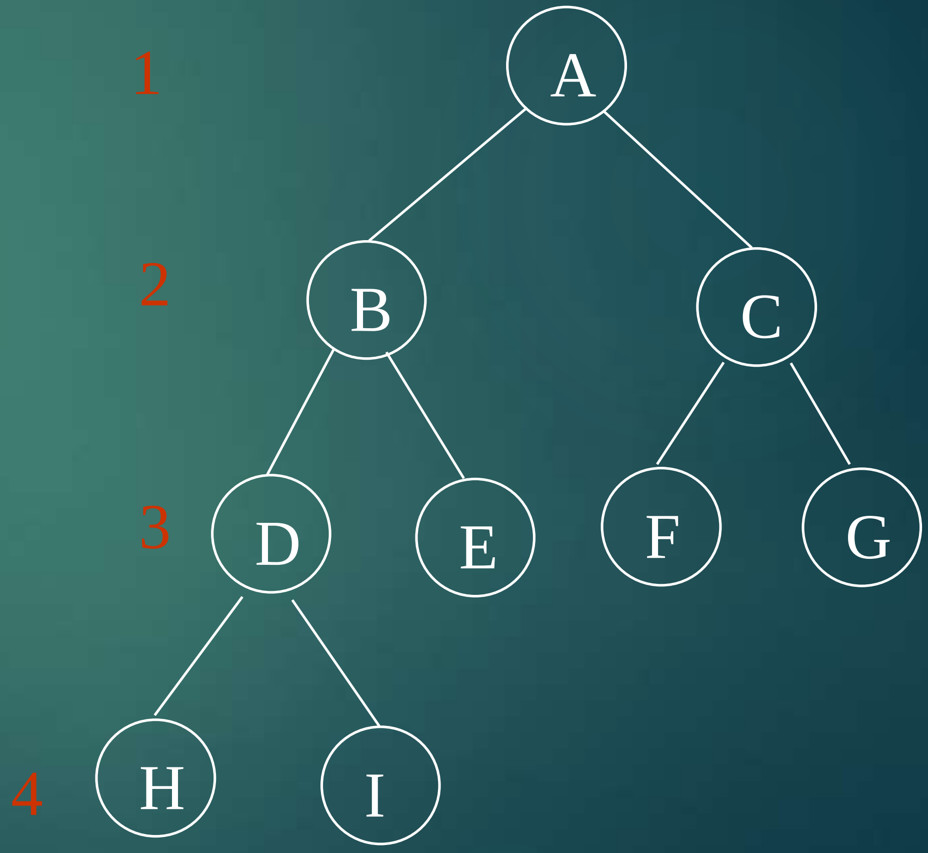


Examples of the Binary Tree

Skewed Binary Tree

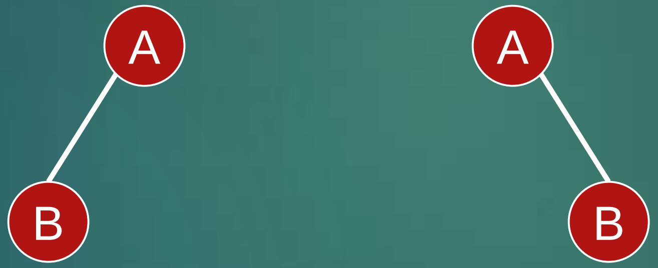


Complete Binary Tree



Differences Between A Tree and A Binary Tree

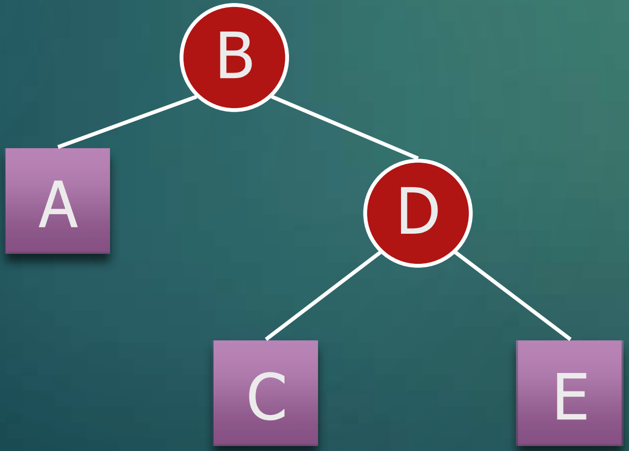
- The subtrees of a binary tree are ordered; those of a tree are not ordered.

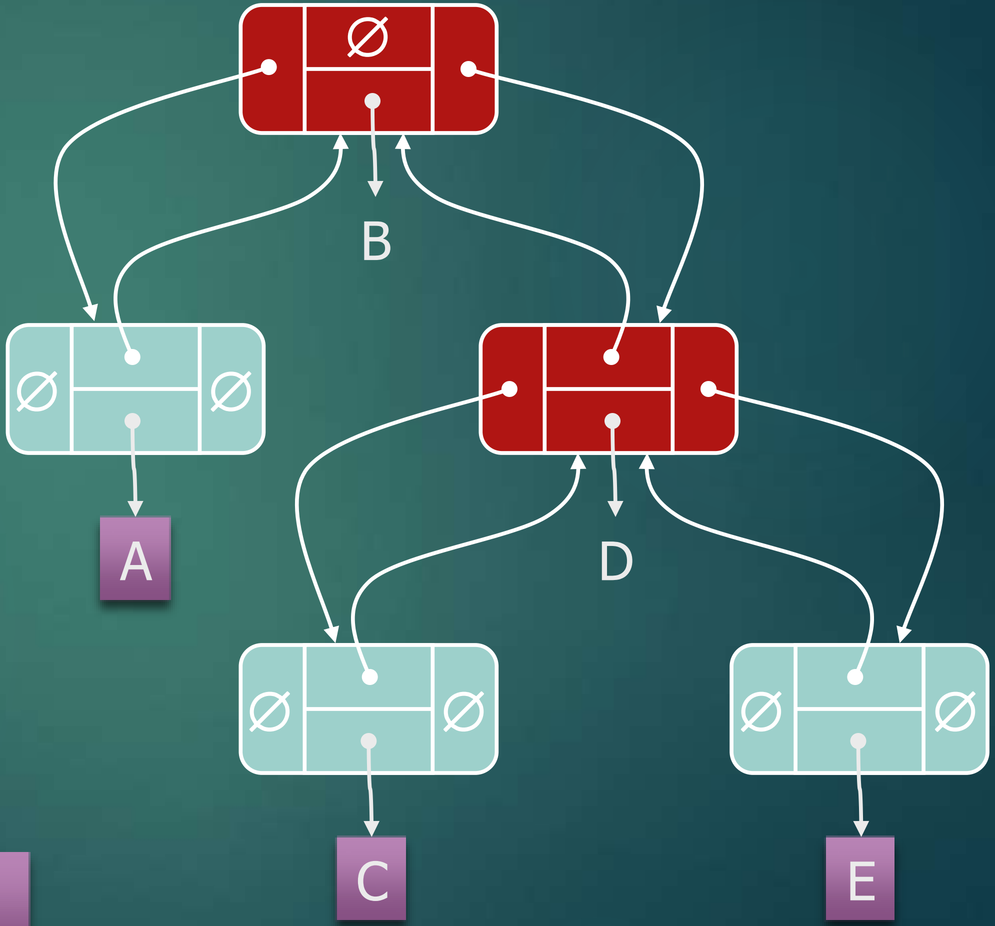


- Are different when viewed as binary trees.
- Are the same when viewed as trees.

Data Structure for Binary Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node





What are BFS and DFS for Binary Tree?

A Tree is typically traversed in two ways:

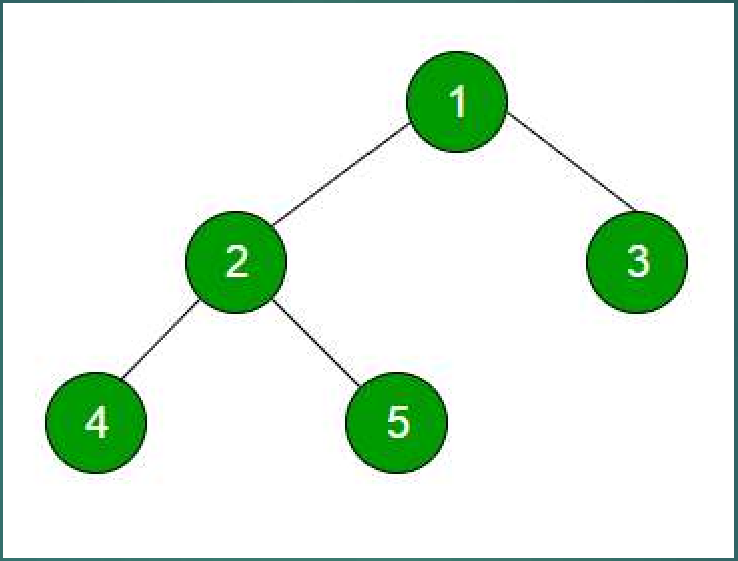
- Breadth First Traversal (Or Level Order Traversal)

- Depth First Traversals
    - Inorder Traversal (Left-Root-Right)
    - Preorder Traversal (Root-Left-Right)
    - Postorder Traversal (Left-Right-Root)

Breadth-first search (BFS)

- **Breadth-first search** (**BFS**) is an algorithm for traversing or searching tree or graph data structures.
- It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

Breadth-first search (BFS)



Level order traversal of the above tree is 1 2 3 4 5

# Recursive Python program for level order traversal of Binary Tree

```
# A node structure
class Node:
        # A utility function to create a new node
        def __init__(self, key):
                self.data = key
                self.left = None
                self.right = None
# Function to print level order traversal of tree
def printLevelOrder(root):
        h = height(root)
        for i in range(1, h+1):
                printGivenLevel(root, i)
# Print nodes at a given level
def printGivenLevel(root , level):
        if root is None:
                return
        if level == 1:
                print(root.data,end=" ")
        elif level > 1 :
                printGivenLevel(root.left , level-1)
                printGivenLevel(root.right , level-1)
```

```
def height(node):
        if node is None:
                return 0
        else :
        # Compute the height of each subtree
                lheight = height(node.left)
                rheight = height(node.right)

                #Use the larger one
                if lheight > rheight :
                        return lheight+1
                else:
                        return rheight+1
# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
print("Level order traversal of binary tree is -")
printLevelOrder(root)
```

Output

```
Level Order traversal of binary tree is 1 2 3 4 5
```

Depth First Traversals

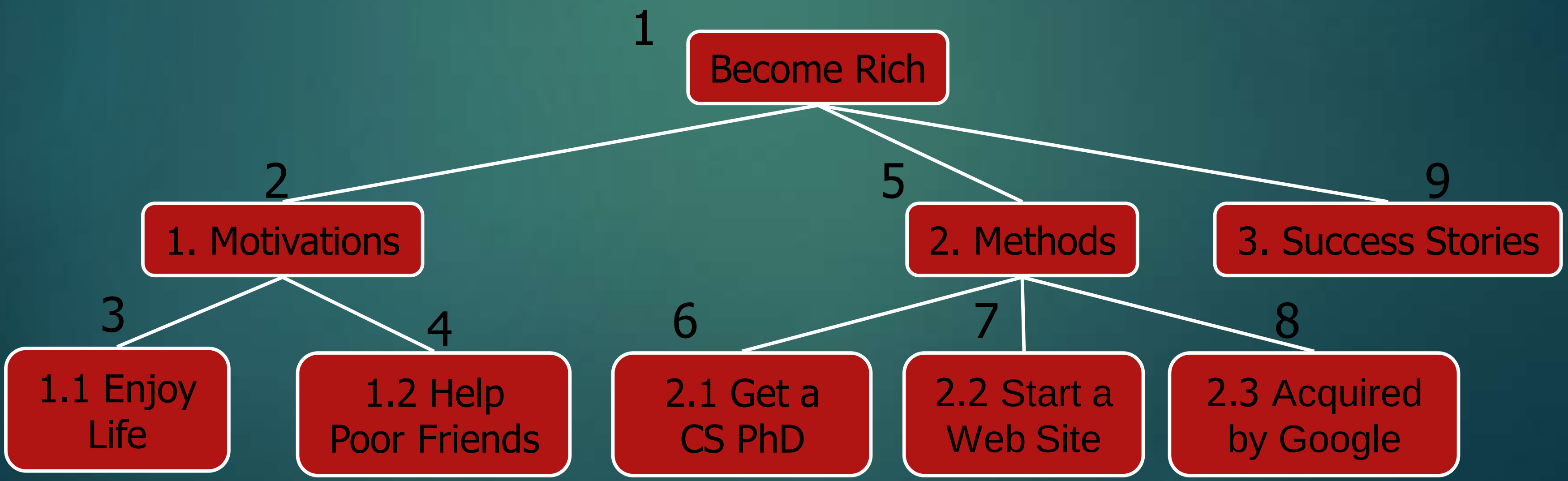
- Preorder
- Postorder
- Inorder

- Inorder
  - Traverse the left subtree, i.e., call Inorder(left-subtree)
  - Visit the root.
  - Traverse the right subtree, i.e., call Inorder(right-subtree)
- Preorder:
  - Visit the root.
  - Traverse the left subtree, i.e., call Preorder(left-subtree)
  - Traverse the right subtree, i.e., call Preorder(right-subtree)
- Postorder
  - Traverse the left subtree, i.e., call Postorder(left-subtree)
  - Traverse the right subtree, i.e., call Postorder(right-subtree)
  - Visit the root.

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

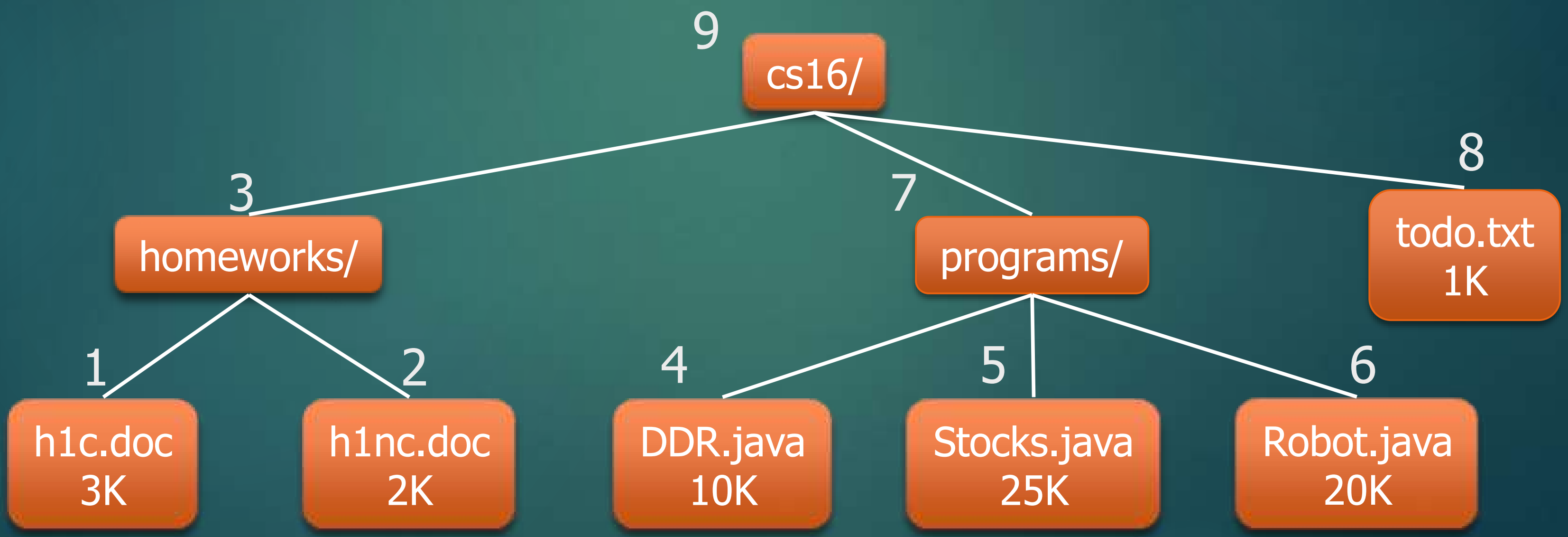
```
Algorithm preOrder(v)
    visit(v)
    for each child w of v
        preorder (w)
```



Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

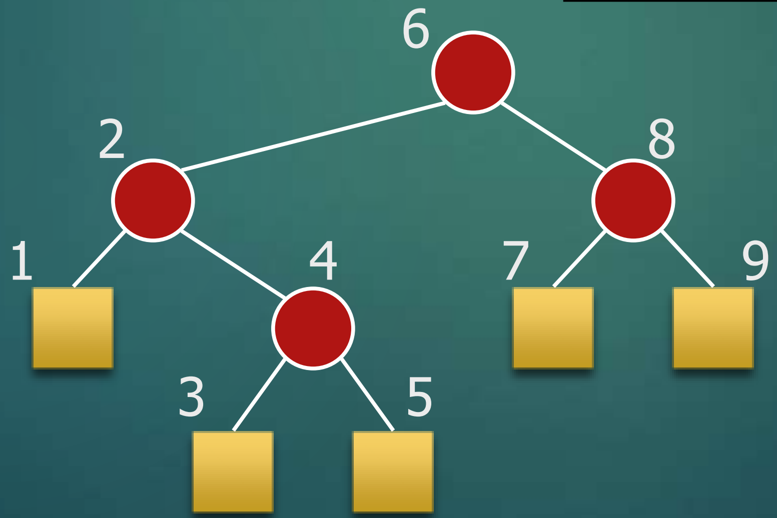
```
Algorithm postOrder(v)
    for each child w of v
        postOrder (w)
    visit(v)
```



Inorder Traversal

- In an inorder traversal a node is visited after its left subtree and before its right subtree

```
Algorithm inOrder(v)
    if isInternal (v)
        inOrder (leftChild (v))
    visit(v)
    if isInternal (v)
        inOrder (rightChild (v))
```



InOrder(root) visits nodes in the following order:

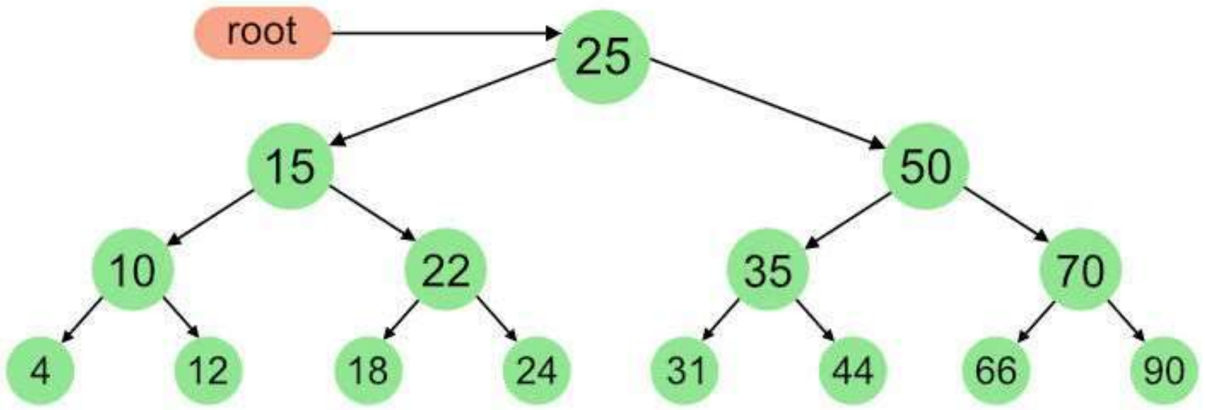
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



# Python program for tree traversals

```
# A class that represents an individual node in a
# Binary Tree
class Node:
        def __init__(self,key):
                self.left = None
                self.right = None
                self.val = key
# A function to do inorder tree traversal
def printInorder(root):

        if root:
                # First recur on left child
                printInorder(root.left)

                # then print the data of node
                print(root.val),

                # now recur on right child
                printInorder(root.right)
# A function to do postorder tree traversal
def printPostorder(root):
        if root:
                # First recur on left child
                printPostorder(root.left)

                # the recur on right child
                printPostorder(root.right)

                # now print the data of node
                print(root.val),
```

```
# A function to do preorder tree traversal
def printPreorder(root):
        if root:
                # First print the data of node
                print(root.val),

                # Then recur on left child
                printPreorder(root.left)

                # Finally recur on right child
                printPreorder(root.right)
```

```
# Driver code
root = Node(1)
root.left       = Node(2)
root.right      = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
print "Preorder traversal of binary tree is"
printPreorder(root)

print "\nInorder traversal of binary tree is"
printInorder(root)

print "\nPostorder traversal of binary tree is"
printPostorder(root)
```

Output:

```
Preorder traversal of binary tree is
 1 2 4 5 3
Inorder traversal of binary tree is
 4 2 5 1 3
Postorder traversal of binary tree is
 4 5 2 3 1
```

insert element in binary tree

```
class newNode():

        def __init__(self, data):
                self.key = data
                self.left = None
                self.right = None
""" Inorder traversal of a binary tree"""
def inorder(temp):
        if (not temp):
                return
        inorder(temp.left)
        print(temp.key,end = " ")
        inorder(temp.right)
"""function to insert element in binary tree """
def insert(temp,key):
        if not temp:
                root = newNode(key)
                return
        q = []
        q.append(temp)
        # Do level order traversal until we find
        # an empty place.
while (len(q)):
                temp = q[0]
                q.pop(0)
        if (not temp.left):
                temp.left = newNode(key)
                break
        else:
                q.append(temp.left)

        if (not temp.right):
                temp.right = newNode(key)
                break
        else:
                q.append(temp.right)
# Driver code
if __name__ == '__main__':
        root = newNode(10)
        root.left = newNode(11)
        root.left.left = newNode(7)
        root.right = newNode(9)
        root.right.left = newNode(15)
        root.right.right = newNode(8)

        print("Inorder traversal before insertion:", end = " ")
        inorder(root)
        key = 12
        insert(root, key)
        print()
        print("Inorder traversal after insertion:", end = " ")
        inorder(root)
```

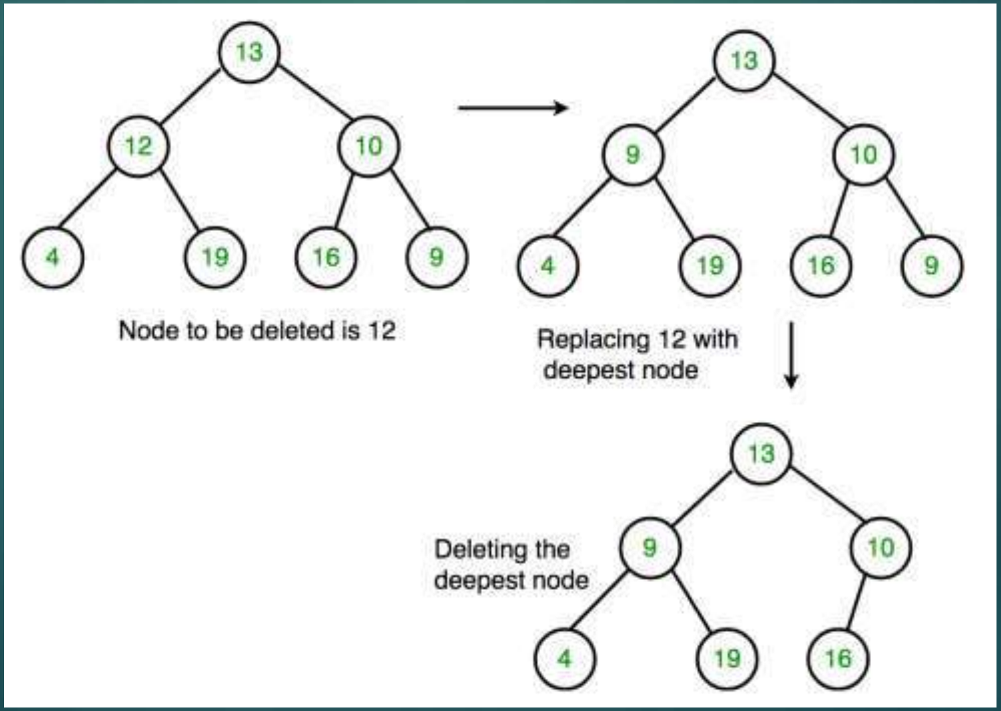
Delete element from binary tree

Algorithm

1. Starting at root, find the deepest and rightmost node in binary tree and node which we want to delete.

2. Replace the deepest rightmost node's data with node to be deleted.

3. Then delete the deepest rightmost node.



```
# class to create a node with data, left child and right child.
class Node:
        def __init__(self,data):
                self.data = data
                self.left = None
                self.right = None

# Inorder traversal of a binary tree
def inorder(temp):
        if(not temp):
                return
        inorder(temp.left)
        print(temp.data, end = " ")
        inorder(temp.right)
# function to delete the given deepest node (d_node) in binary tree
def deleteDeepest(root,d_node):
        q = []
        q.append(root)
        while(len(q)):
        temp = q.pop(0)
        if temp is d_node:
        temp = None
        return
        if temp.right:
        if temp.right is d_node:
        temp.right = None
return
        else:
        q.append(temp.right)
        if temp.left:
                if temp.left is d_node:
                temp.left = None
                        return
        else:
        q.append(temp.left)
```

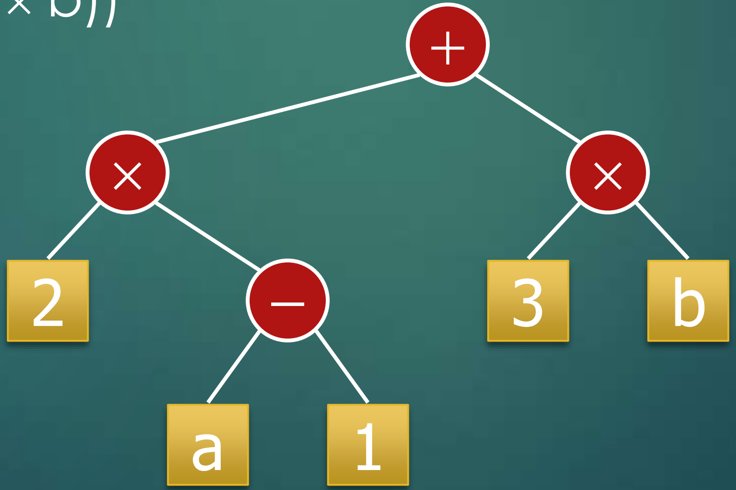
```
# function to delete element in binary tree
def deletion(root, key):
        if root == None :
                return None
        if root.left == None and root.right == None:
                if root.key == key :
                        return None
                else :
                        return root
        key_node = None
        q = []
        q.append(root)
        while(len(q)):
                temp = q.pop(0)
                if temp.data == key:
                        key_node = temp
                if temp.left:
                        q.append(temp.left)
                if temp.right:
                        q.append(temp.right)
        if key_node :
                x = temp.data
                deleteDeepest(root,temp)
                key_node.data = x               return root
# Driver code
if __name__=='__main__':
        root = Node(10)                 root.left = Node(11)
        root.left.left = Node(7)        root.left.right = Node(12)
        root.right = Node(9)            root.right.left = Node(15)
        root.right.right = Node(8)      print("The tree before the deletion:")
        inorder(root)   key = 11        root = deletion(root, key)
        print()         print("The tree after the deletion;")
        inorder(root)
```

Output `Inorder traversal before deletion: 7 11 12 10 15 9 8 Inorder traversal after deletion: 7 8 12 10 15 9`

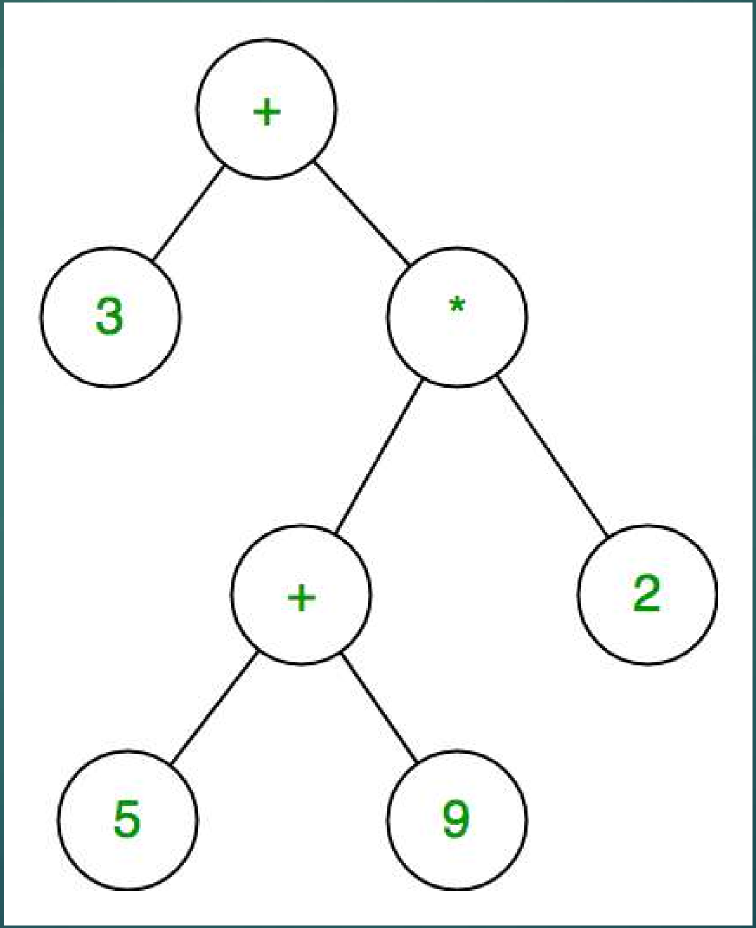
Arithmetic Expression Tree

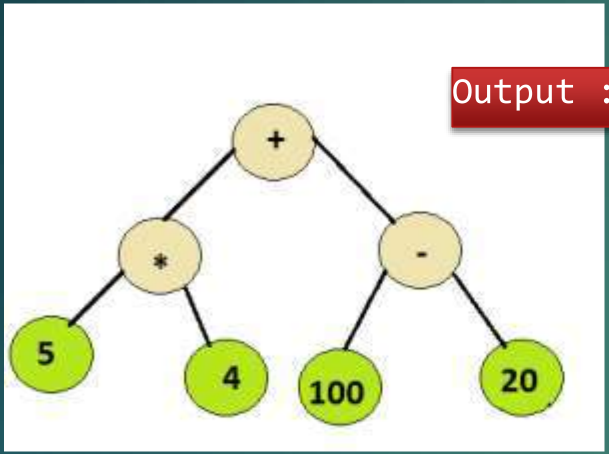
- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression

$(2 \times (a - 1) + (3 \times b))$

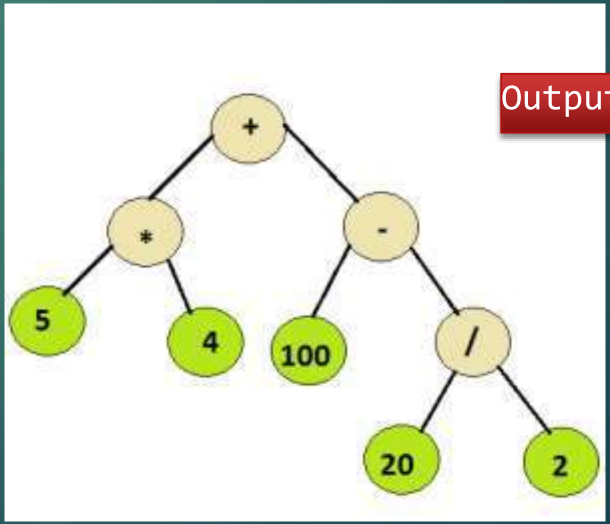


3 + ((5+9)*2)





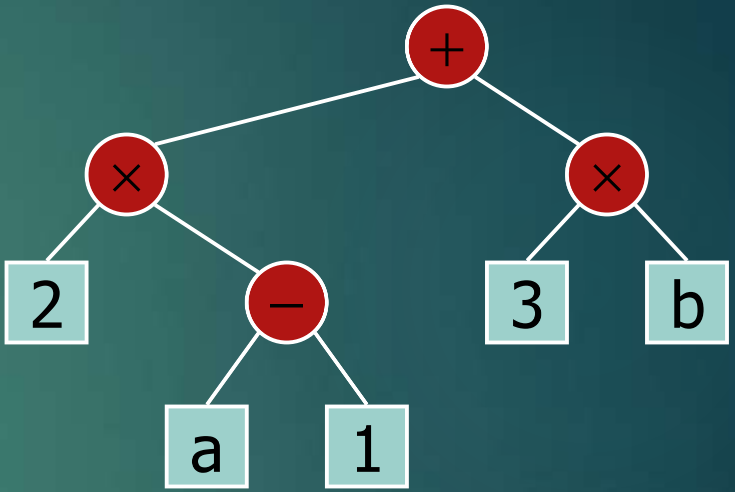
Output : 100



Output : 110

Print Arithmetic Expressions

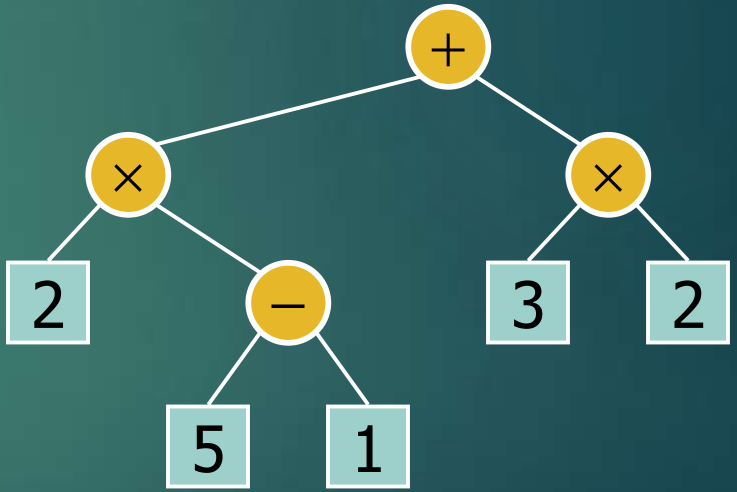
- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree



$((2 \times (a - 1)) + (3 \times b))$

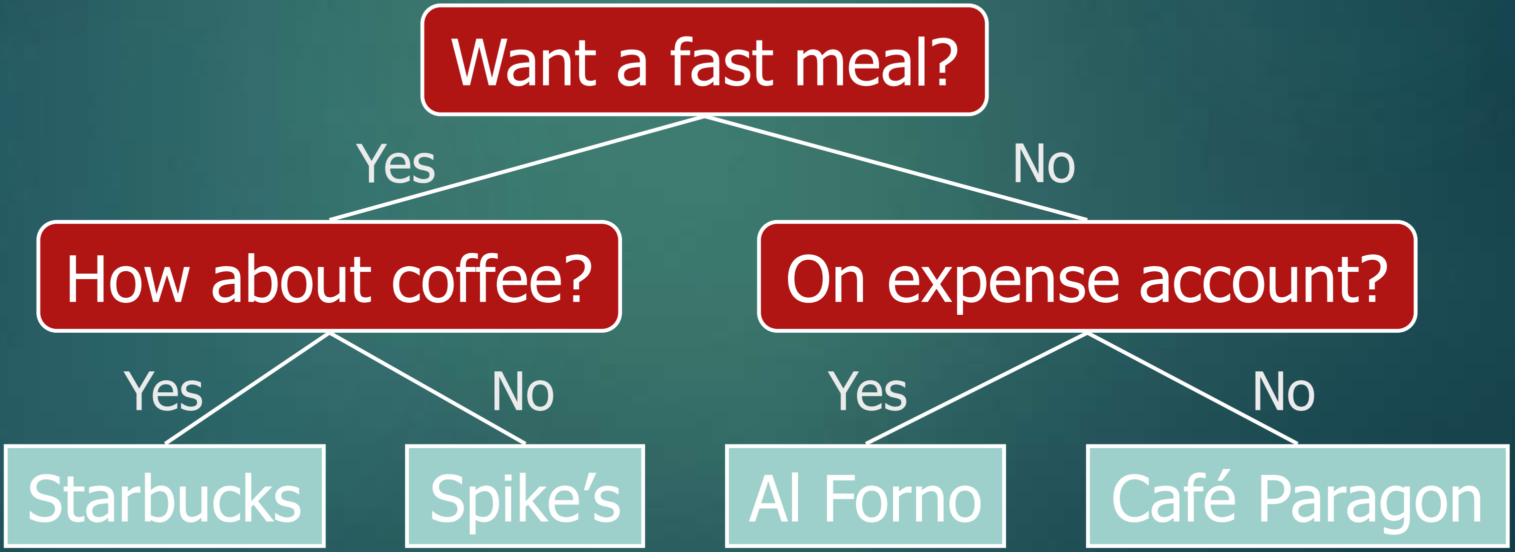
Evaluate Arithmetic Expressions

- recursive method returning the value of a subtree
- when visiting an internal node, combine the values of the subtrees



Decision Tree

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



Maximum Number of Nodes in a Binary Tree

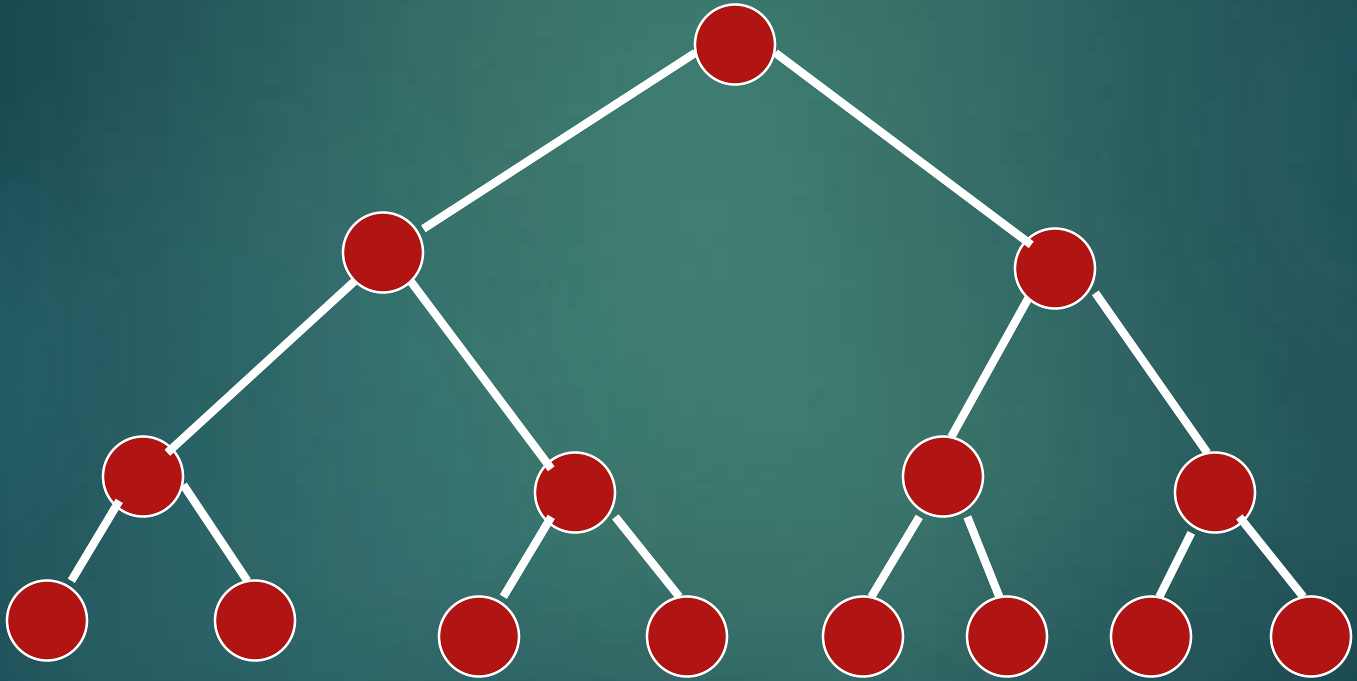
- The maximum number of nodes on depth i of a binary tree is 2^i, i>=0.
- The maximum nubmer of nodes in a binary tree of height k is 2^{k+1}-1, k>=0.

Prove by induction.

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

Full Binary Tree

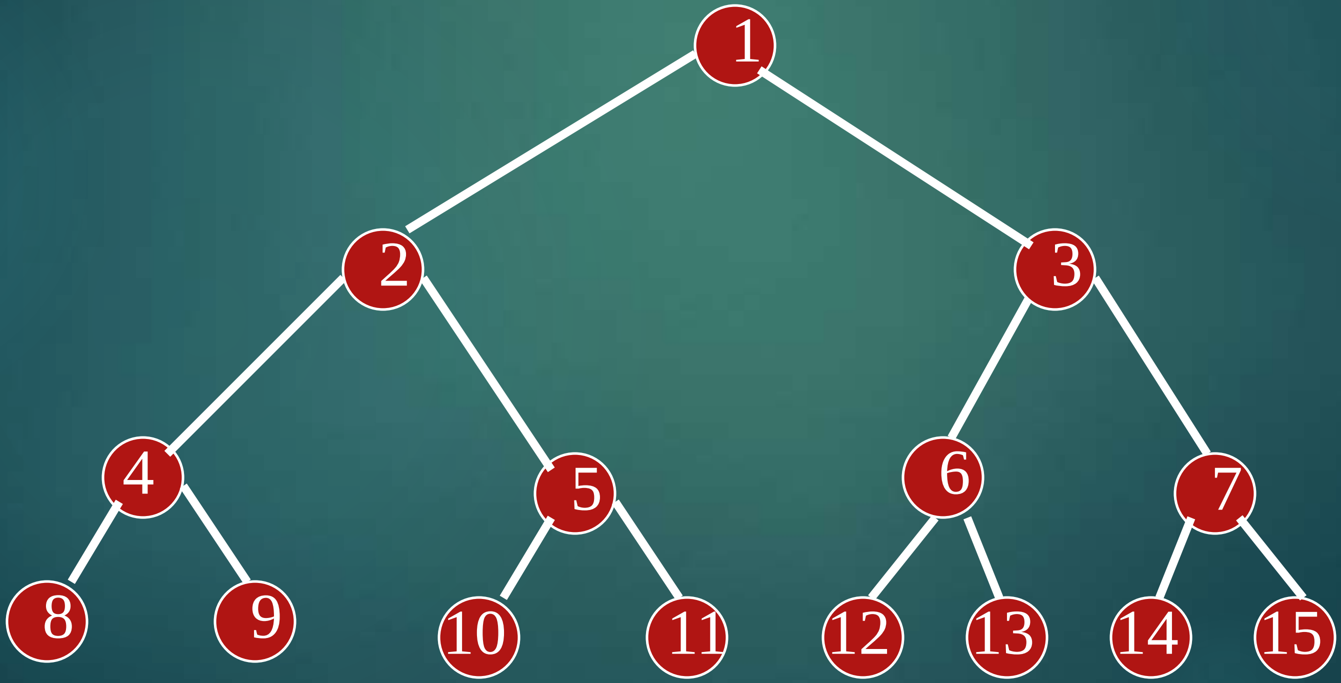
- A full binary tree of a given height k has $2^{k+1}-1$ nodes.



Height 3 full binary tree.

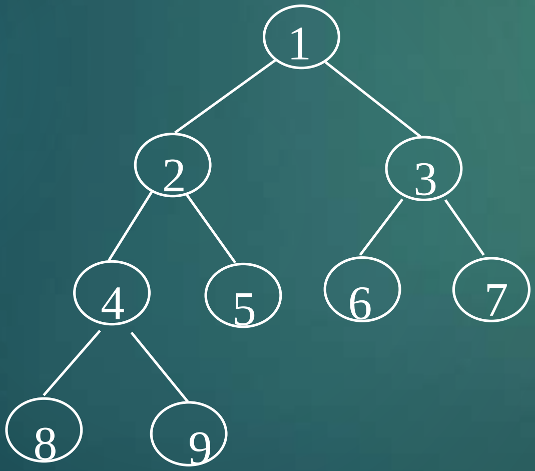
Labeling Nodes In A Full Binary Tree

- Label the nodes 1 through $2^{k+1} - 1$.
- Label by levels from top to bottom.
- Within a level, label from left to right.

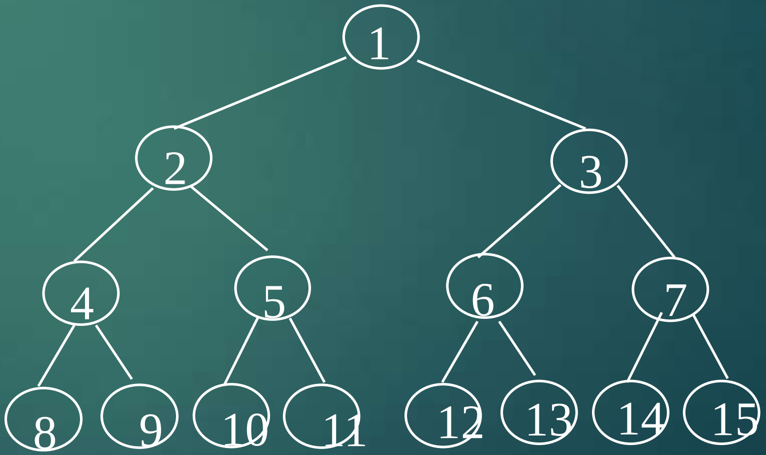


Complete Binary Trees

- A labeled binary tree containing the labels 1 to n with root 1, branches leading to nodes labeled 2 and 3, branches from these leading to 4, 5 and 6, 7, respectively, and so on.
- A binary tree with *n* nodes and level *k* is complete *iff* its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of level *k*.



Complete binary tree



Full binary tree of depth 3

Creativity: pathLength(tree) = Σ depth(v) $\forall v \in$ tree

Algorithm *pathLength*(*v, n*)

Input: a tree node ***v*** and an initial value ***n***

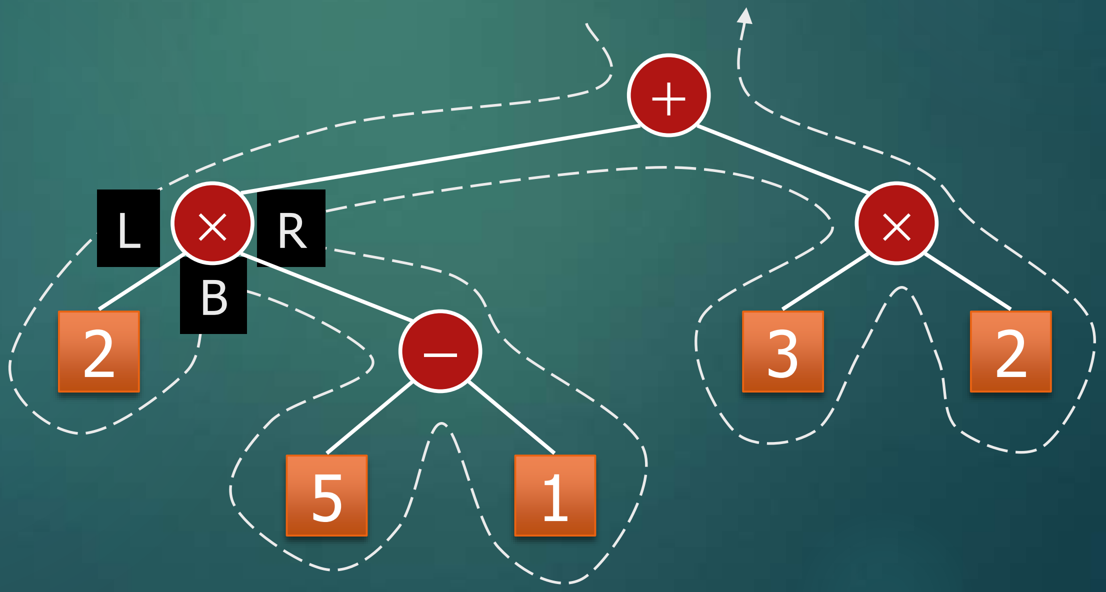
Output: the pathLength of the tree with root ***v***

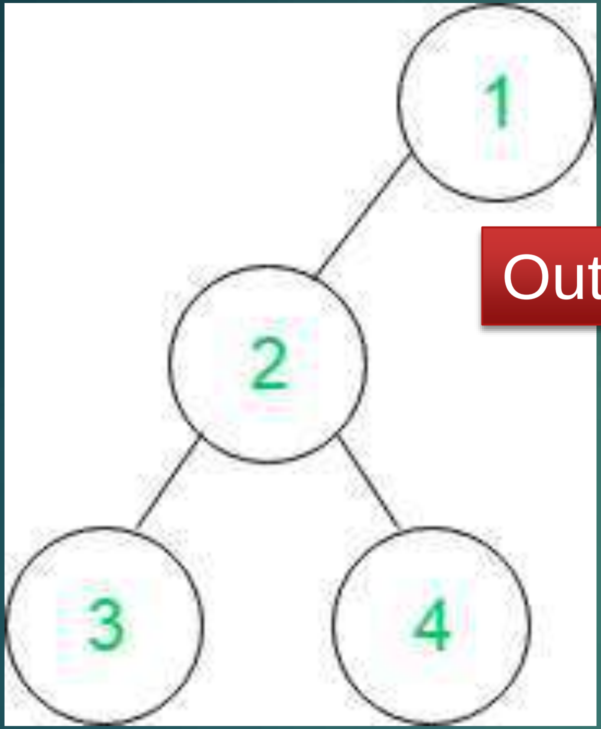
Usage: pl = pathLength(root, 0);

```
if isExternal (v)
    return n
return
    (pathLength(leftChild (v), n + 1) +
pathLength(rightChild (v), n + 1) + n)
```

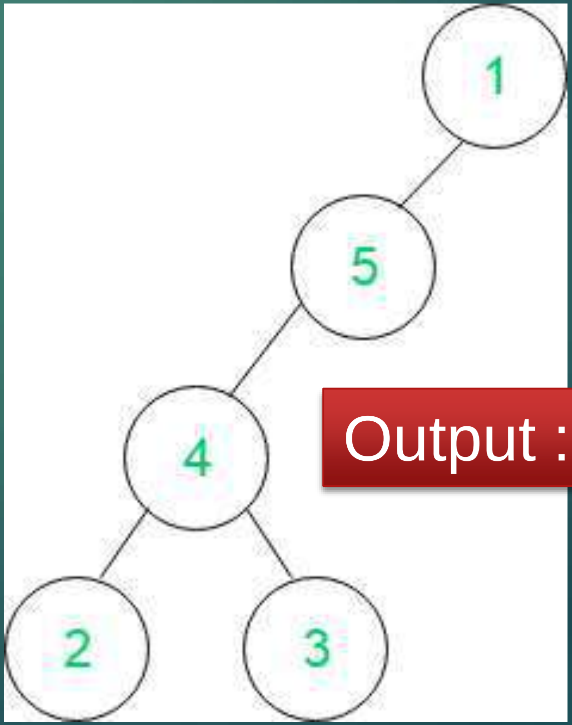
Euler Tour Traversal

- Generic traversal of a binary tree
- Includes a special cases the preorder, postorder and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (preorder)
  - from below (inorder)
  - on the right (postorder)





Output : 1 2 3 2 4 2 1



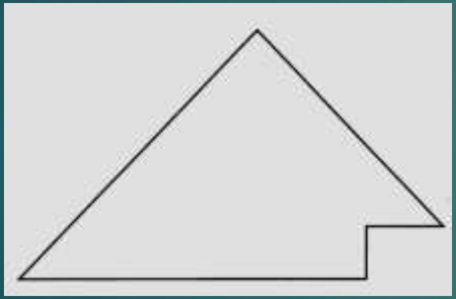
Output : 1 5 4 2 4 3 4 5 1

Euler Tour Traversal

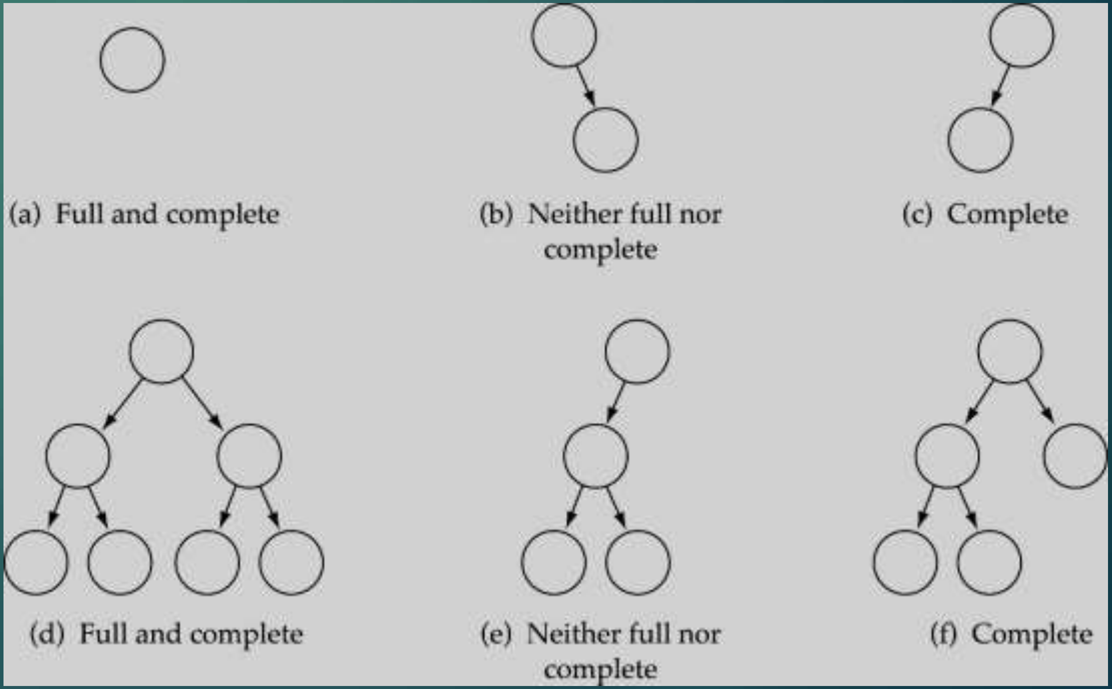
```
eulerTour(node v)
  perform action for visiting node on the left;
  if v is internal then
      eulerTour(v->left);
  perform action for visiting node from below;
  if v is internal then
      eulerTour(v->right);
  perform action for visiting node on the right;
```

Complete Binary Tree

- A binary tree that is either full or full through the next-to-last level
- The last level is full from left to right (i.e., leaves are as far to the left as possible)



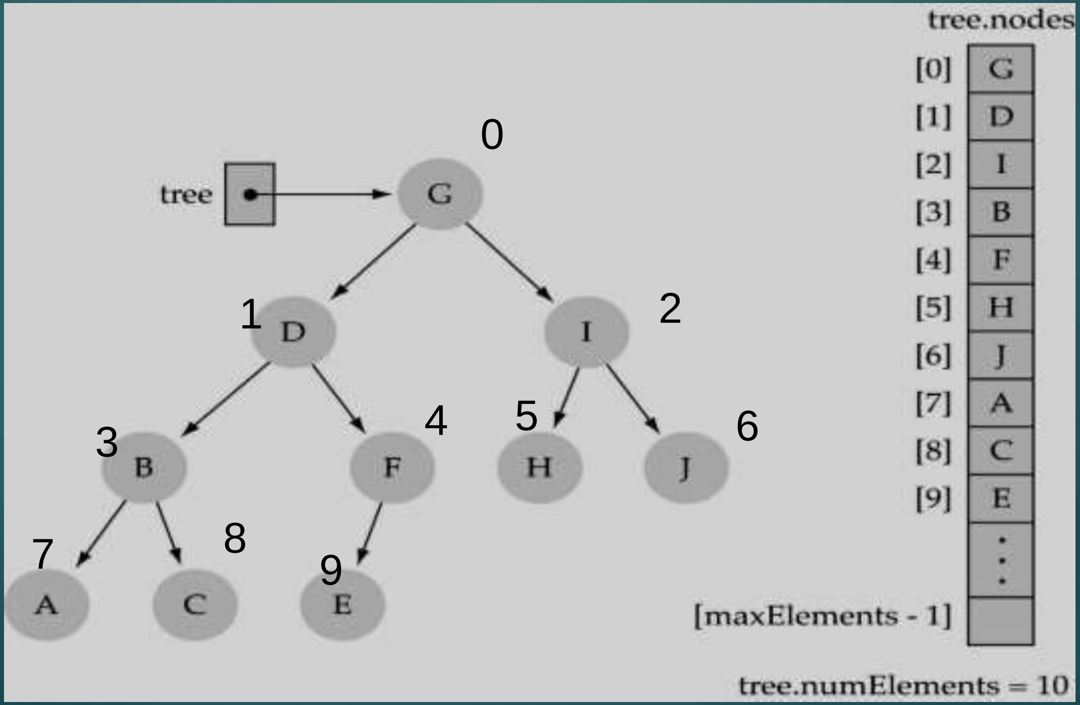
Complete Binary Tree



Array-based representation of binary trees

- Memory space can be saved (no pointers are required)
- Preserve parent-child relationships by storing the tree elements in the array

  (i) level by level, and *(ii)* left to right

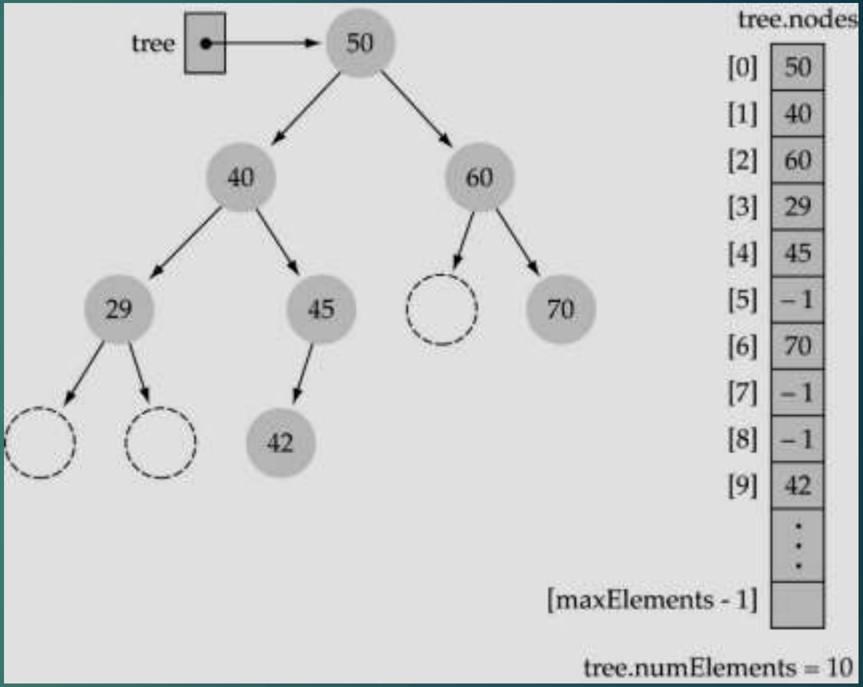


Array-based representation of binary trees (cont.)

- Parent-child relationships:
  - left child of *tree.nodes[index]* = *tree.nodes[2*index+1]*
  - right child of *tree.nodes[index]* = *tree.nodes[2*index+2]*
  - parent node of *tree.nodes[index]* = *tree.nodes[(index-1)/2]* (int division-truncate)
- Leaf nodes:
  - *tree.nodes[numElements/2]* to *tree.nodes[numElements - 1]*

Array-based representation of binary trees (cont.)

- Full or complete trees can be implemented easily using an array-based representation (elements occupy contiguous array slots)
- "Dummy nodes" are required for trees which are not full or complete



Data Structure

Dr. Itimad Raheem Ali

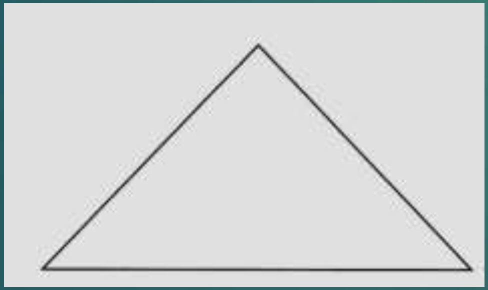
Email : itimadra@uoitc.edu.iq

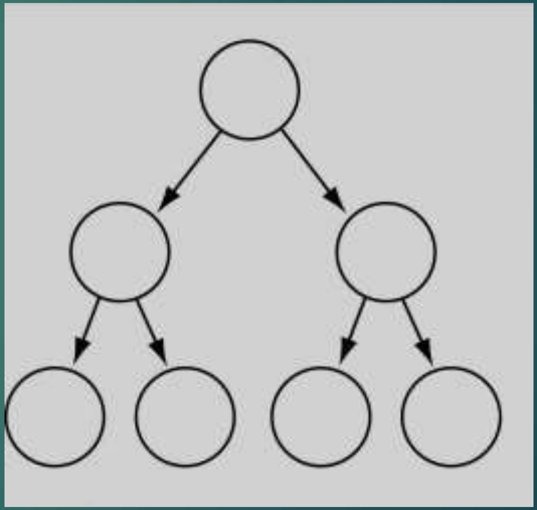
Lecture 10

Heaps

Full Binary Tree

- Every non-leaf node has two children
- All the leaves are on the same level

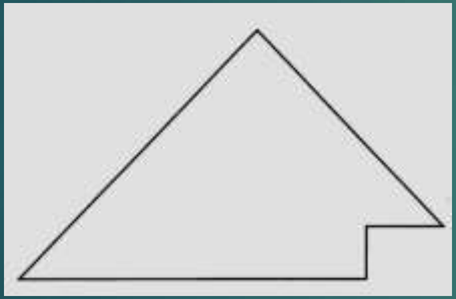




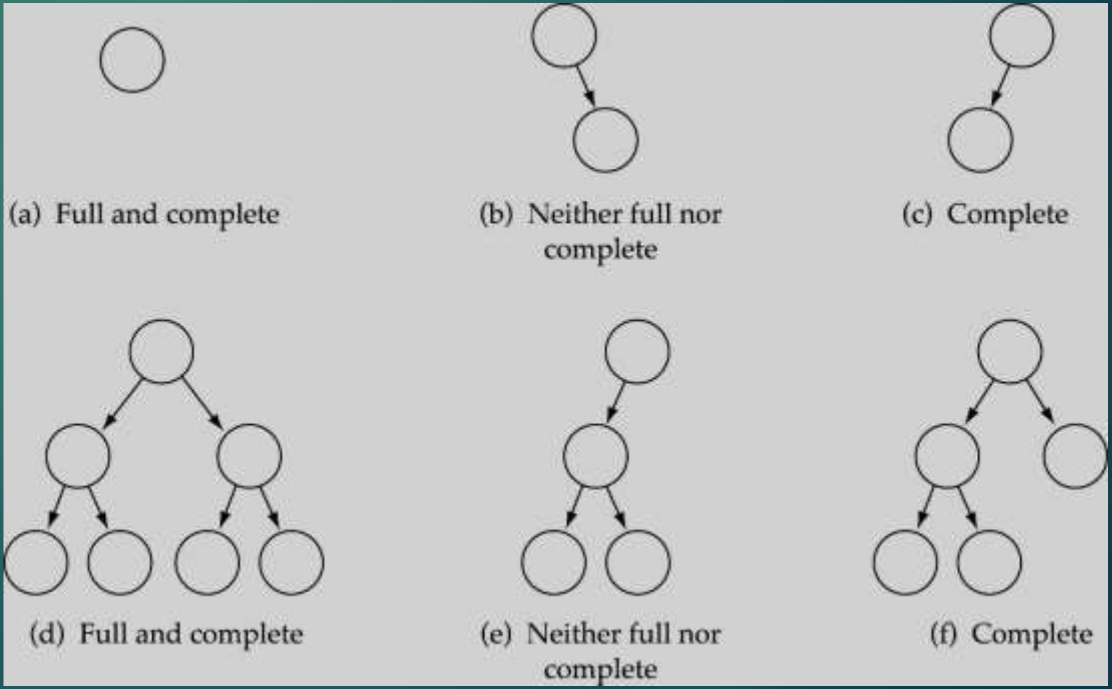
Full Binary Tree

Complete Binary Tree

- A binary tree that is either full or full through the next-to-last level
- The last level is full from left to right (i.e., leaves are as far to the left as possible)



Complete Binary Tree



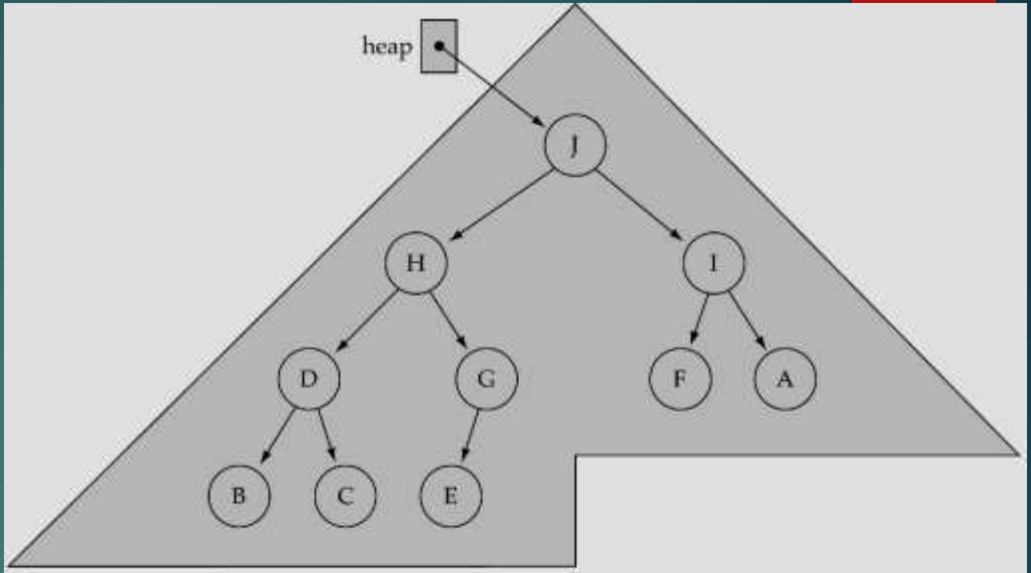
What is a heap?

Heap is a Binary Tree with following properties:

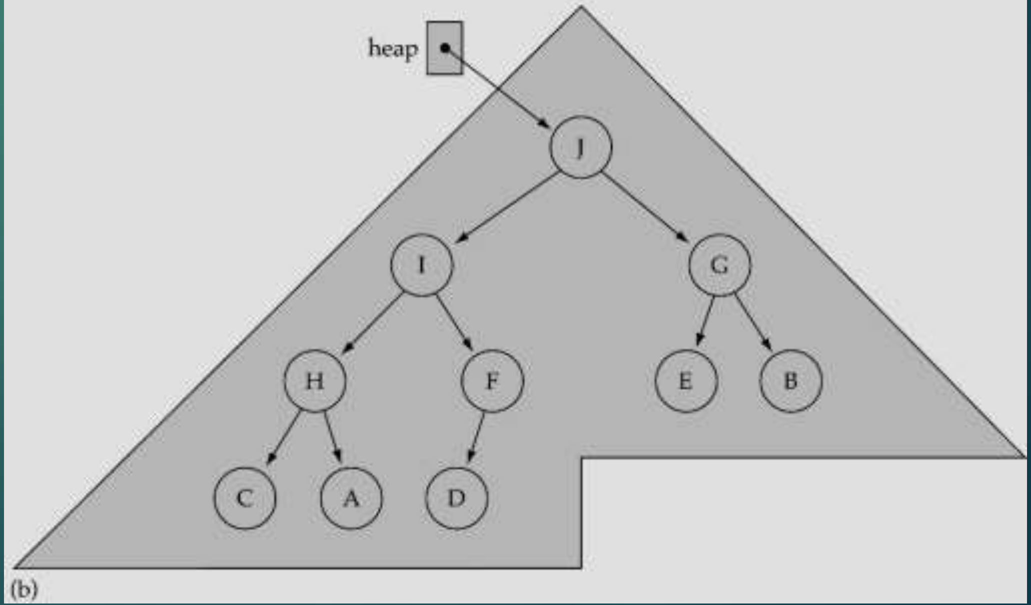
1) It’s a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to MinHeap.

What is a heap? (cont.)



(a)



(b)

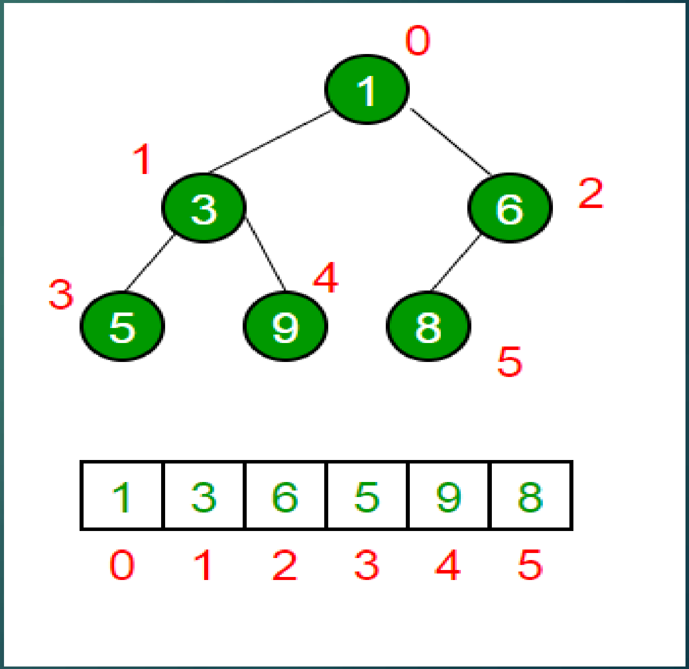
How is Heap represented?

A Heap is a Complete Binary Tree. Heap is typically represented as an array.

- •The root element will be at Arr[0].
- •Below table shows indexes of other nodes for the ith node, i.e., Arr[i]:

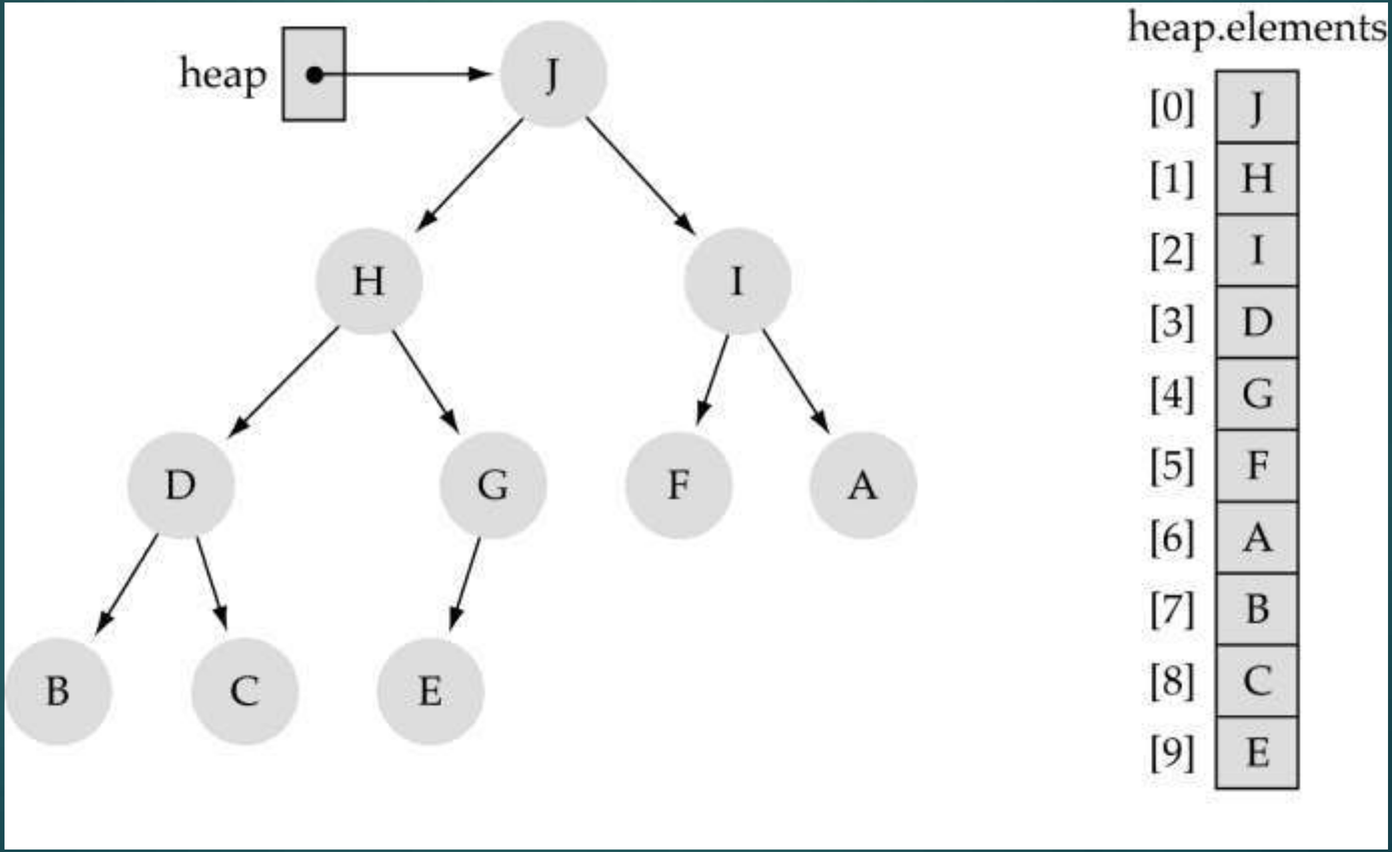
The traversal method use to achieve Array representation is **Level Order**

Arr[(i-1)/2]	Returns the parent node
Arr[(2*i)+1]	Returns the left child node
Arr[(2*i)+2]	Returns the right child node



How is Heap represented?

- From *Property 2*, the largest value of the heap is always stored at the root



Applications of Heaps:

1) Heap Sort: Heap Sort uses Binary Heap to sort an array in O(nLogn) time.

2) Priority Queue: Priority queues can be efficiently implemented using Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time.

3) Graph Algorithms: The priority queues are especially used in Graph Algorithms like Dijkstra's Shortest Path and Prim's Minimum Spanning Tree.

4) Many problems can be efficiently solved using Heaps. See following for example.

a) K'th Largest Element in an array.

b) Sort an almost sorted array/

c) Merge K Sorted Arrays.

Operations on Min Heap:

1) getMini(): It returns the root element of Min Heap. Time Complexity of this operation is O(1).

2) extractMin(): Removes the minimum element from MinHeap. Time Complexity of this Operation is O(Logn) as this operation needs to maintain the heap property (by calling heapify()) after removing root.

3) decreaseKey(): Decreases value of key. The time complexity of this operation is O(Logn).

4) insert(): Inserting a new key takes O(Logn) time. We add a new key at the end of the tree. IF new key is greater than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

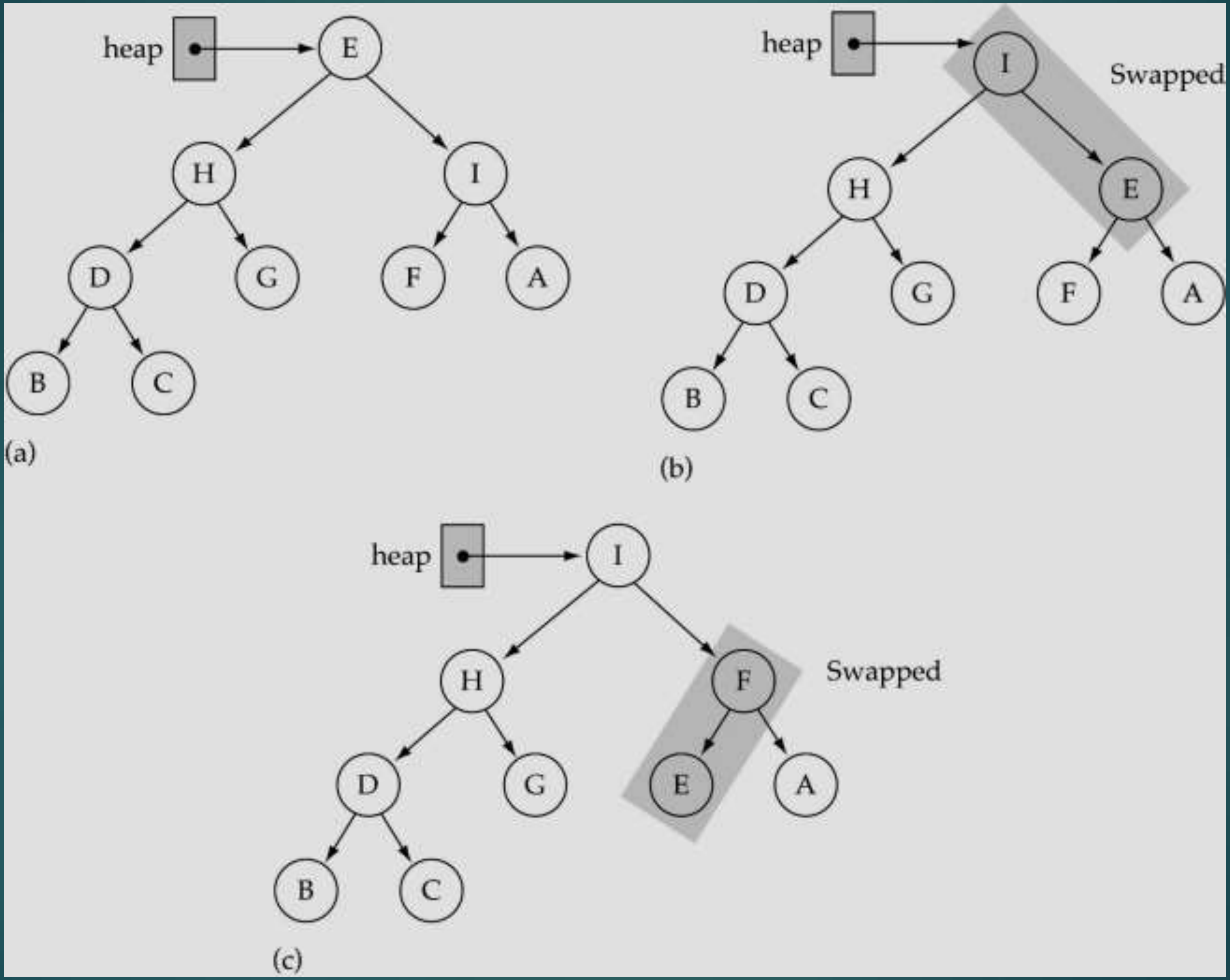
5) delete(): Deleting a key also takes O(Logn) time. We replace the key to be deleted with minum infinite by calling decreaseKey().

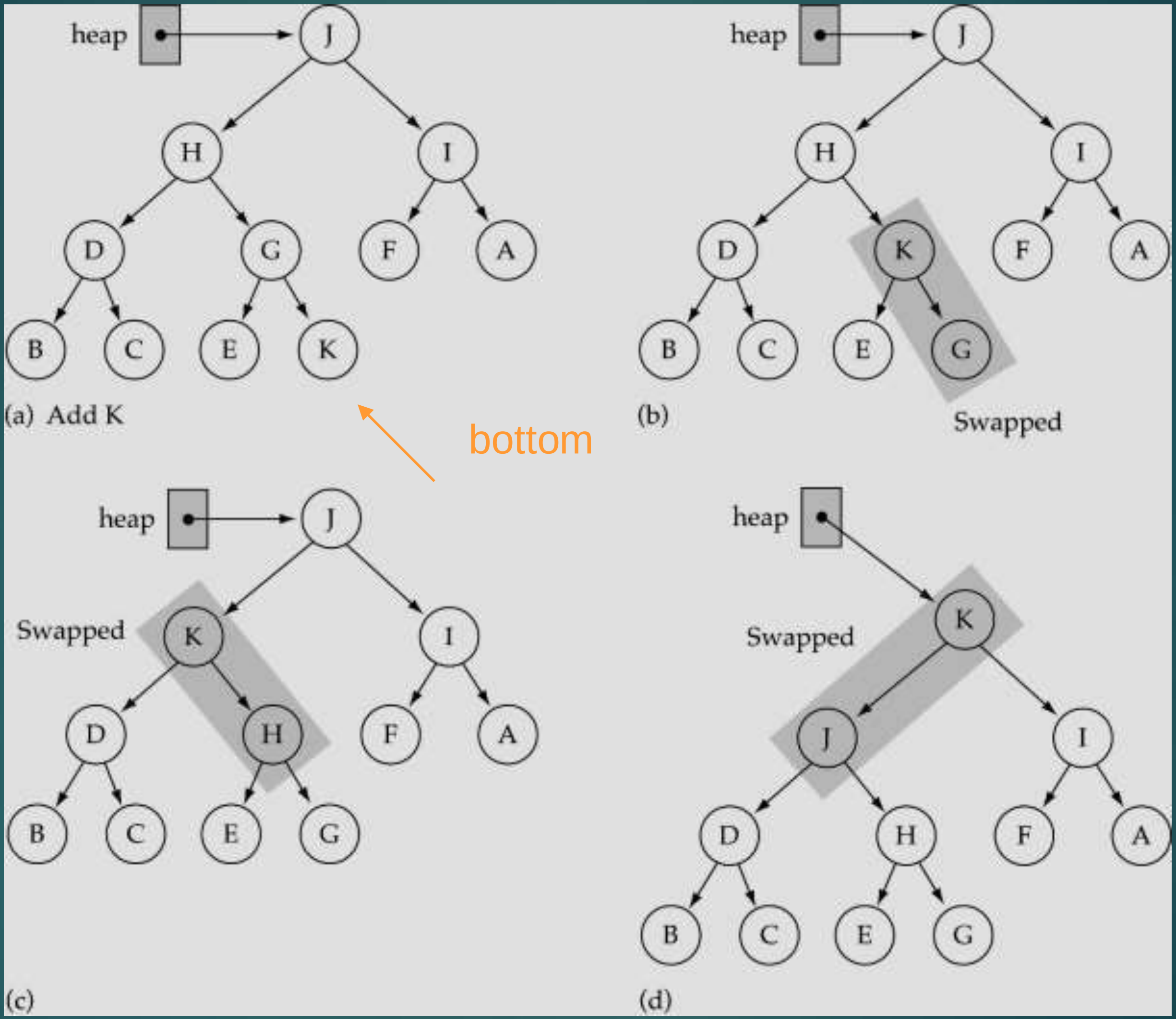
```
# A Python program to demonstrate common binary heap operations

# Import the heap functions from python library
from heapq import heappush, heappop, heapify

# heappop - pop and return the smallest element from heap
# heappush - push the value item onto the heap, maintaining
# heap invarient
# heapify - transform list into heap, in place, in linear time

# A class for Min Heap
class MinHeap:

    # Constructor to initialize a heap
    def __init__(self):
        self.heap = []

    def parent(self, i):
        return (i-1)/2
    # Inserts a new key 'k'
    def insertKey(self, k):
        heappush(self.heap, k)

  # Decrease value of key at index 'i' to new_val
    # It is assumed that new_val is smaller than heap[i]
    def decreaseKey(self, i, new_val):
        self.heap[i] = new_val
        while(i != 0 and self.heap[self.parent(i)] > self.heap[i]):
                # Swap heap[i] with heap[parent(i)]
                self.heap[i] , self.heap[self.parent(i)] = (
                self.heap[self.parent(i)], self.heap[i])
    # Method to remove minium element from min heap
    def extractMin(self):
        return heappop(self.heap)

# This functon deletes key at index i. It first reduces
# value to minus infinite and then calls extractMin()
        def deleteKey(self, i):
        self.decreaseKey(i, float("-inf"))
        self.extractMin()

        # Get the minimum element from the heap
        def getMin(self):
                return self.heap[0]

# Driver pgoratm to test above function
heapObj = MinHeap()
heapObj.insertKey(3)
heapObj.insertKey(2)
heapObj.deleteKey(1)
heapObj.insertKey(15)
heapObj.insertKey(5)
heapObj.insertKey(4)
heapObj.insertKey(45)

print heapObj.extractMin(),
print heapObj.getMin(),
heapObj.decreaseKey(2, 1)
print heapObj.getMin()
```

Output:

```
2 4 1
```

Examples on heap sort

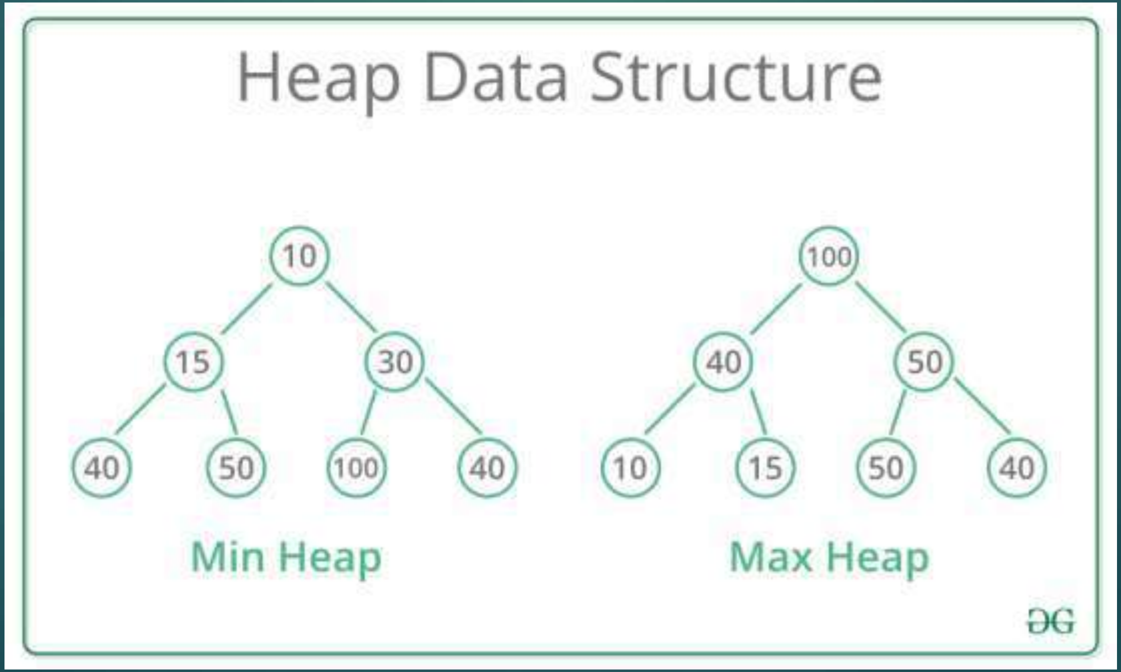



heap
(a) Add K
heap
(b)
Swapped
bottom
heap
Swapped
(c)
heap
Swapped
(d)

Heaps can be of two types:

1.Max-Heap: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.

2.Min-Heap: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.



- The root of the tree A[1] and given index *i* of a node, the indices of its parent, left child and right child can be computed

```
PARENT ($i$)
    return floor($i/2$)
LEFT ($i$)
    return $2i$
RIGHT ($i$)
    return $2i + 1$
```

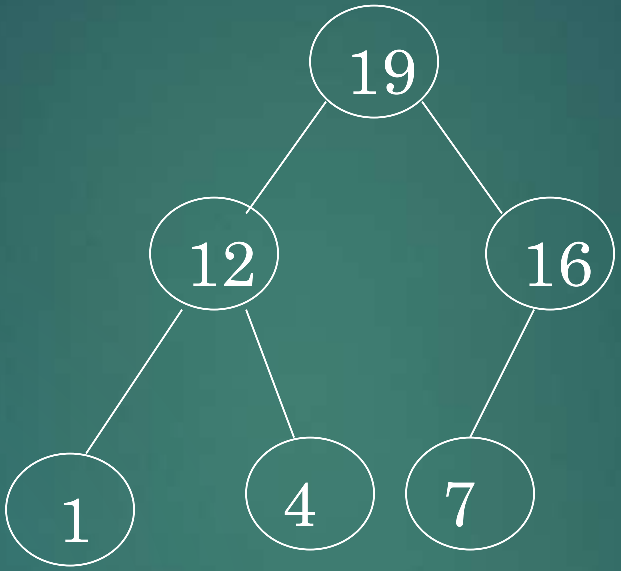
Definition

- Max Heap
  - Store data in ascending order
  - Has property of

    A[Parent(i)] ≥ A[i]
- Min Heap
  - Store data in descending order
  - Has property of

    A[Parent(i)] ≤ A[i]

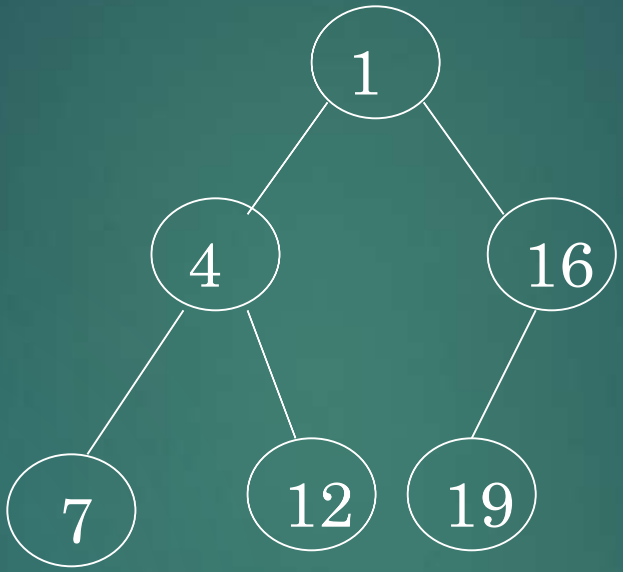
Max Heap Example



19	12	16	1	4	7

Array A

Min heap example



1	4	16	7	12	19

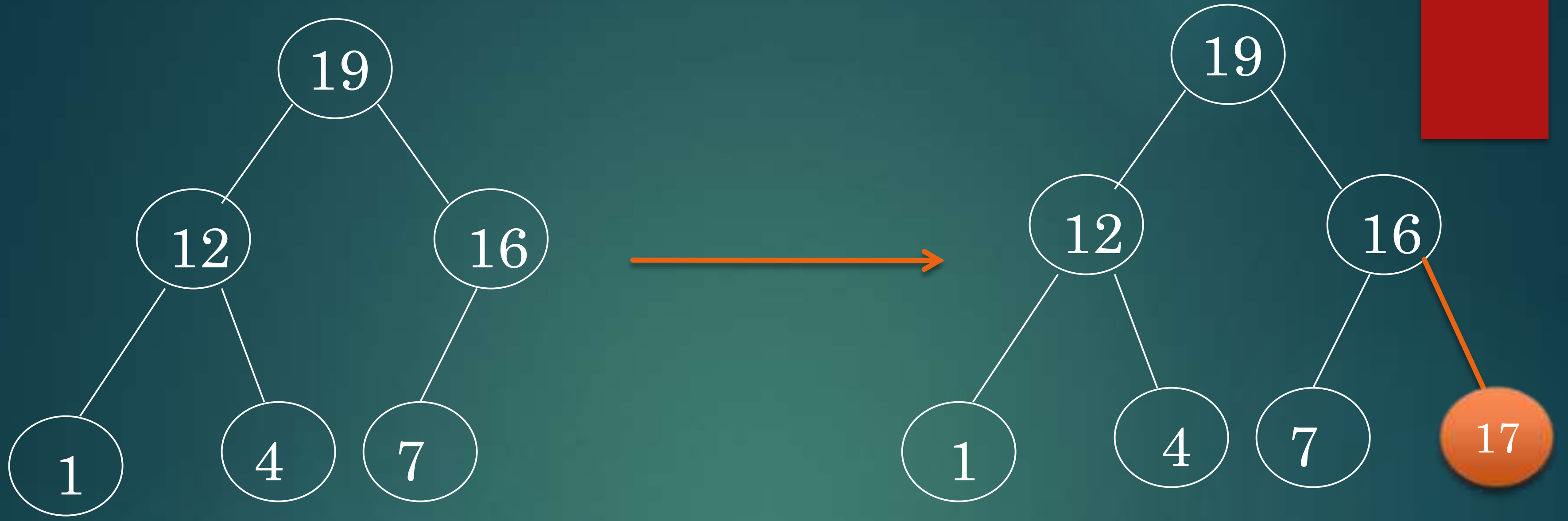
Array A

Insertion

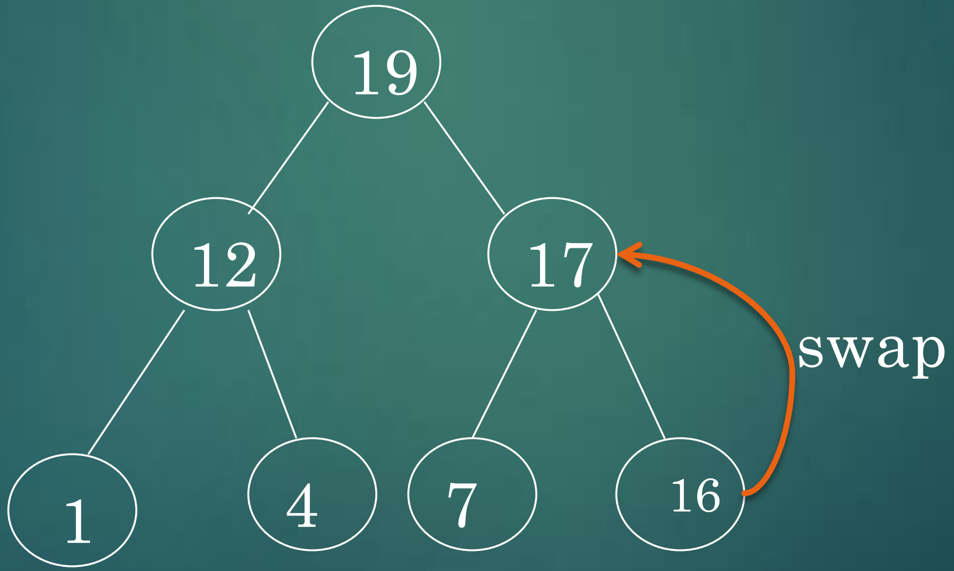
- Algorithm
  1. Add the new element to the next available position at the lowest level
  2. Restore the max-heap property if violated
     - General strategy is percolate up (or bubble up): if the parent of the element is smaller than the element, then interchange the parent and child.

     OR

     Restore the min-heap property if violated
     - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent and child.



Insert 17

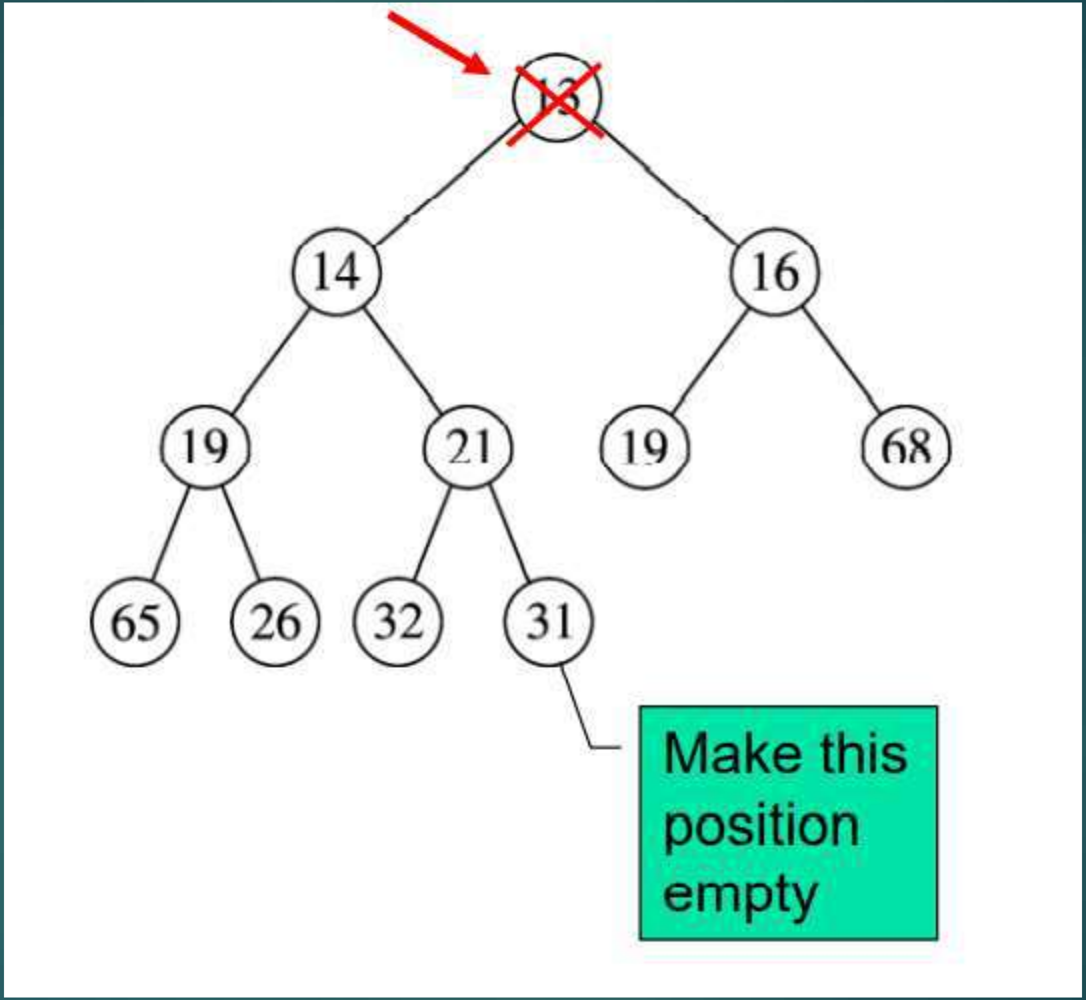


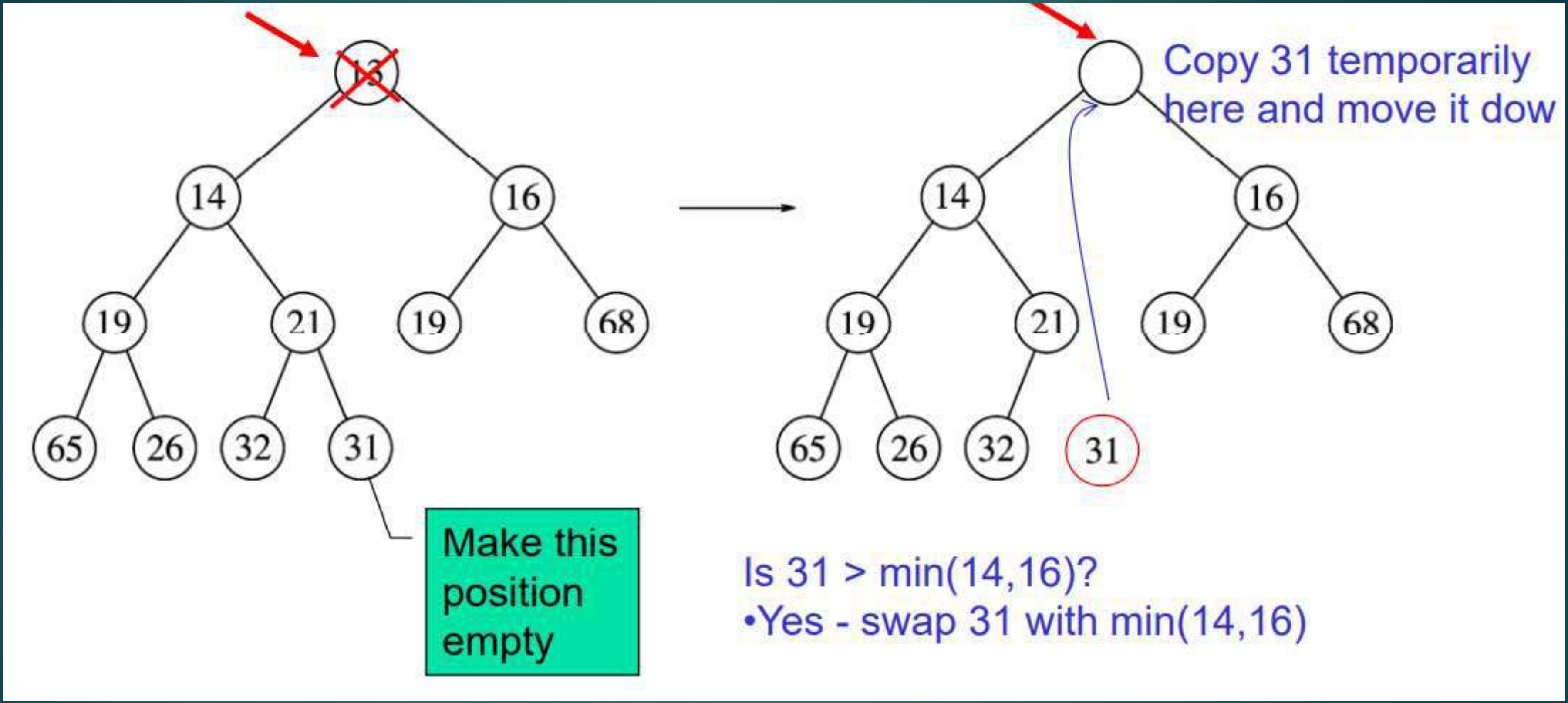
Percolate up to maintain the heap property

Deletion

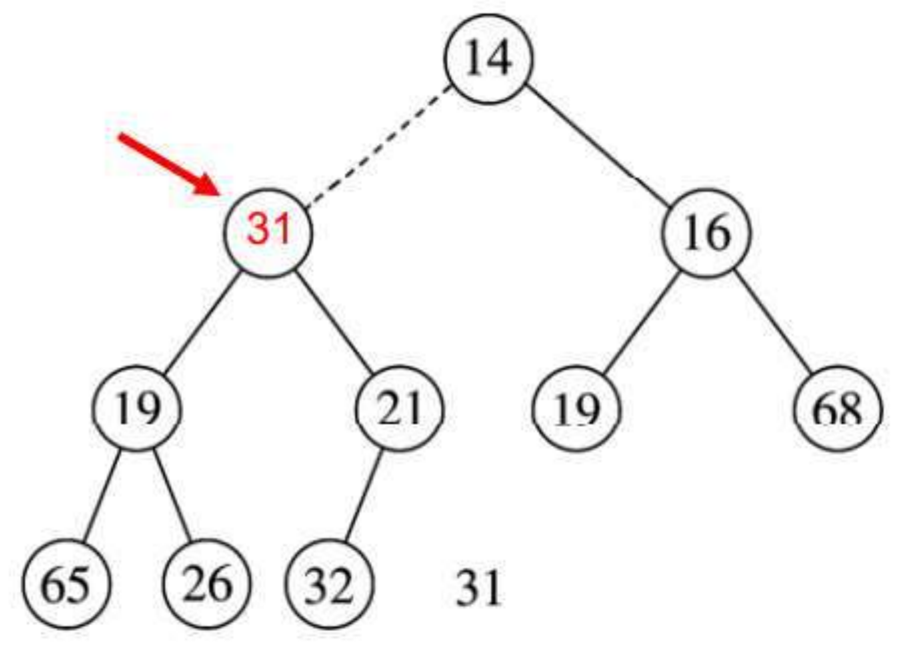
- Delete max
  - Copy the last number to the root (overwrite the maximum element stored there).
  - Restore the max heap property by percolate down.

- Delete min
  - Copy the last number to the root (overwrite the minimum element stored there).
  - Restore the min heap property by percolate down.


13
14
16
19
21
19
68
65
26
32
31
Make this position empty

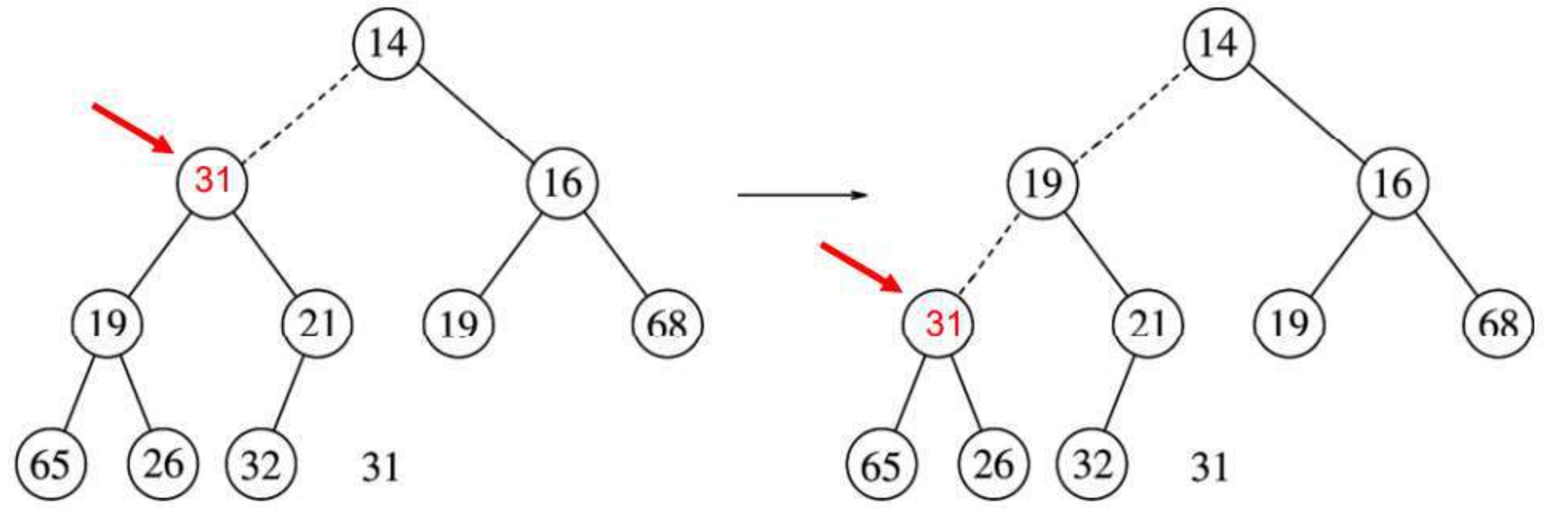


Is 31 > min(14,16)?
•Yes - swap 31 with min(14,16)



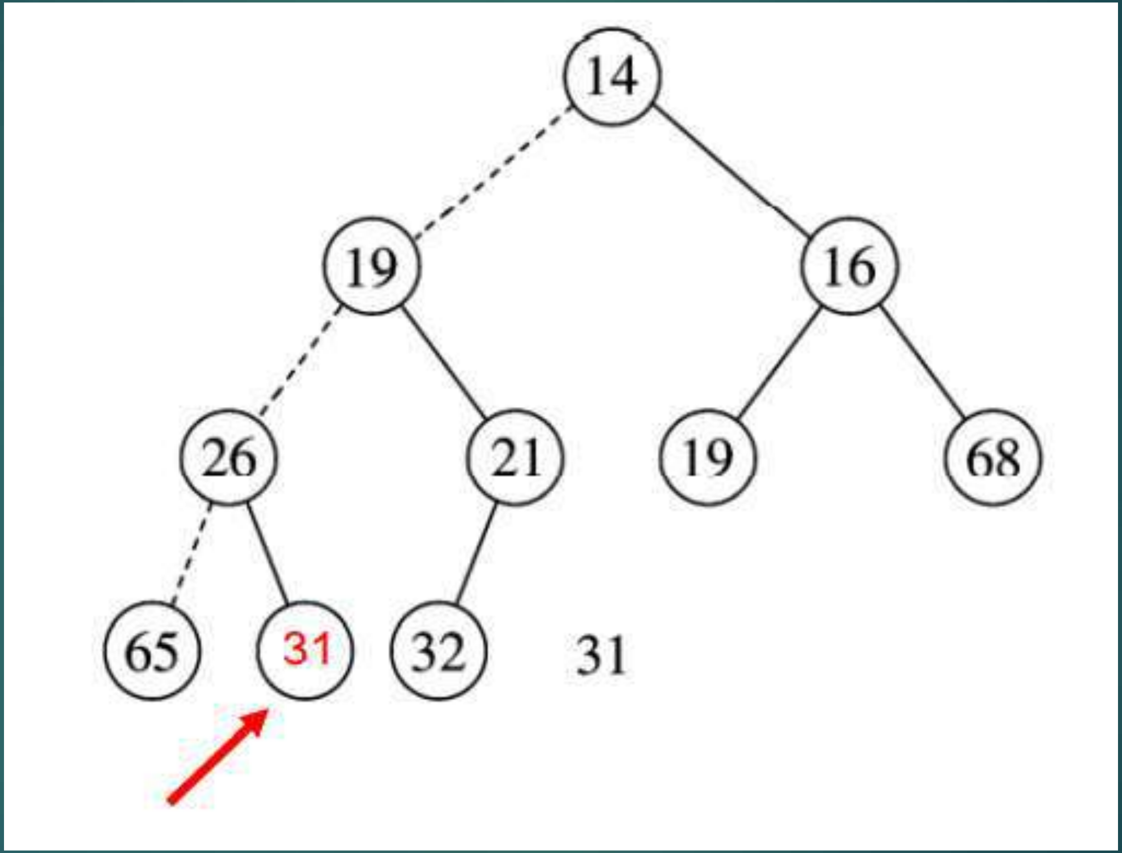
Is 31 > min(19,21)?

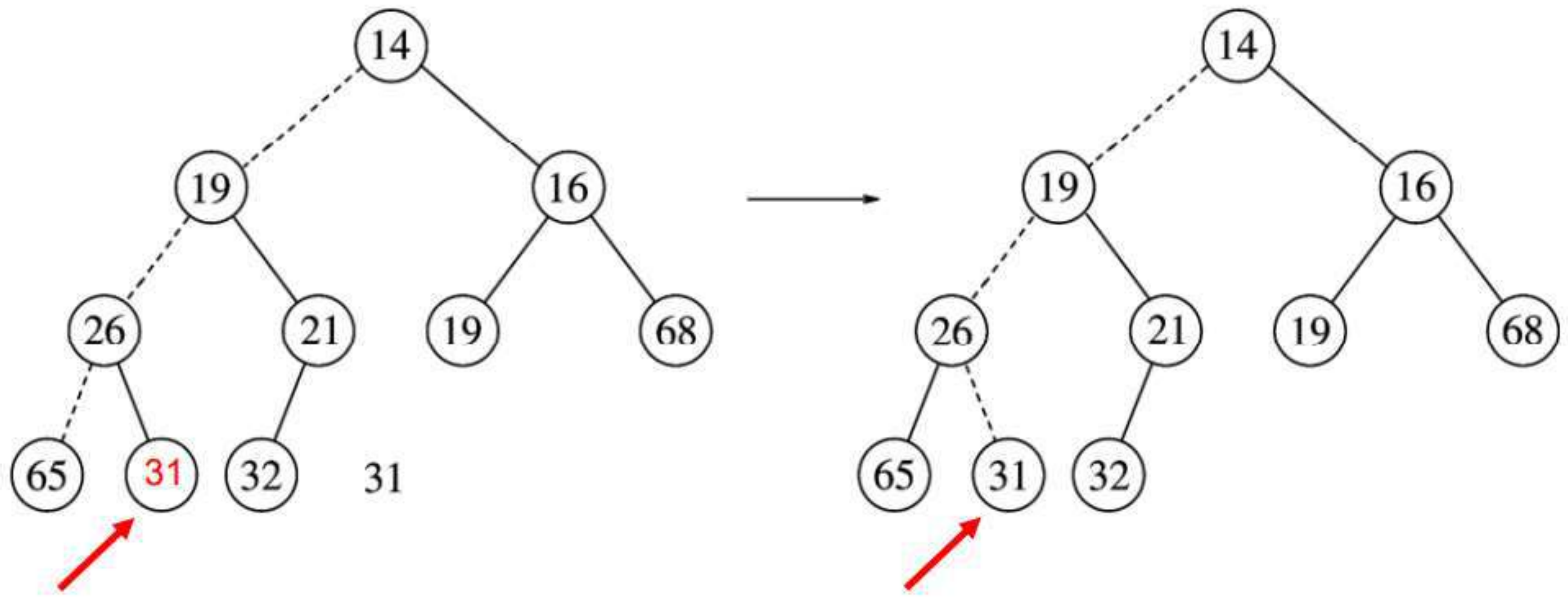
•Yes - swap 31 with min(19,21)



Is 31 > min(19,21)?
•Yes - swap 31 with min(19,21)

Is 31 > min(65,26)?
•Yes - swap 31 with min(65,26)


14
19
16
26
21
19
68
65
31
32
31



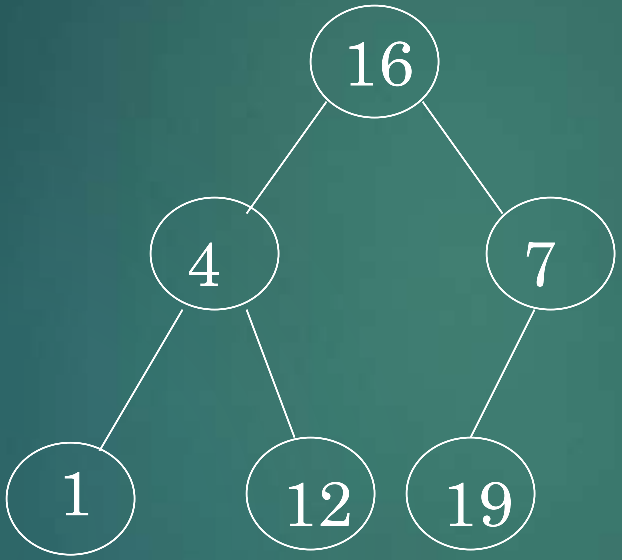
✓ Heap order prop

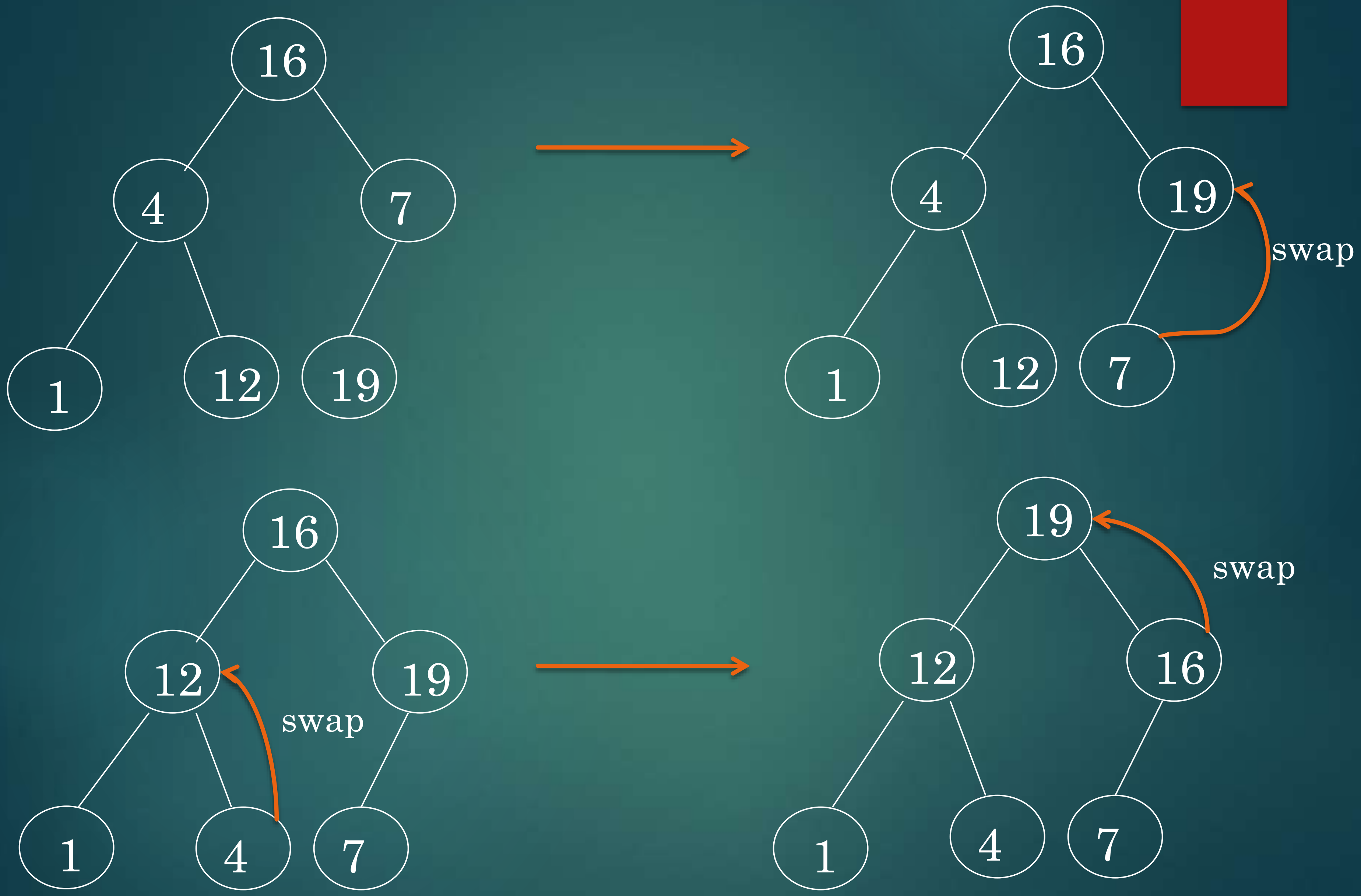
✓ Structure prop

Example: Convert the following array to a heap

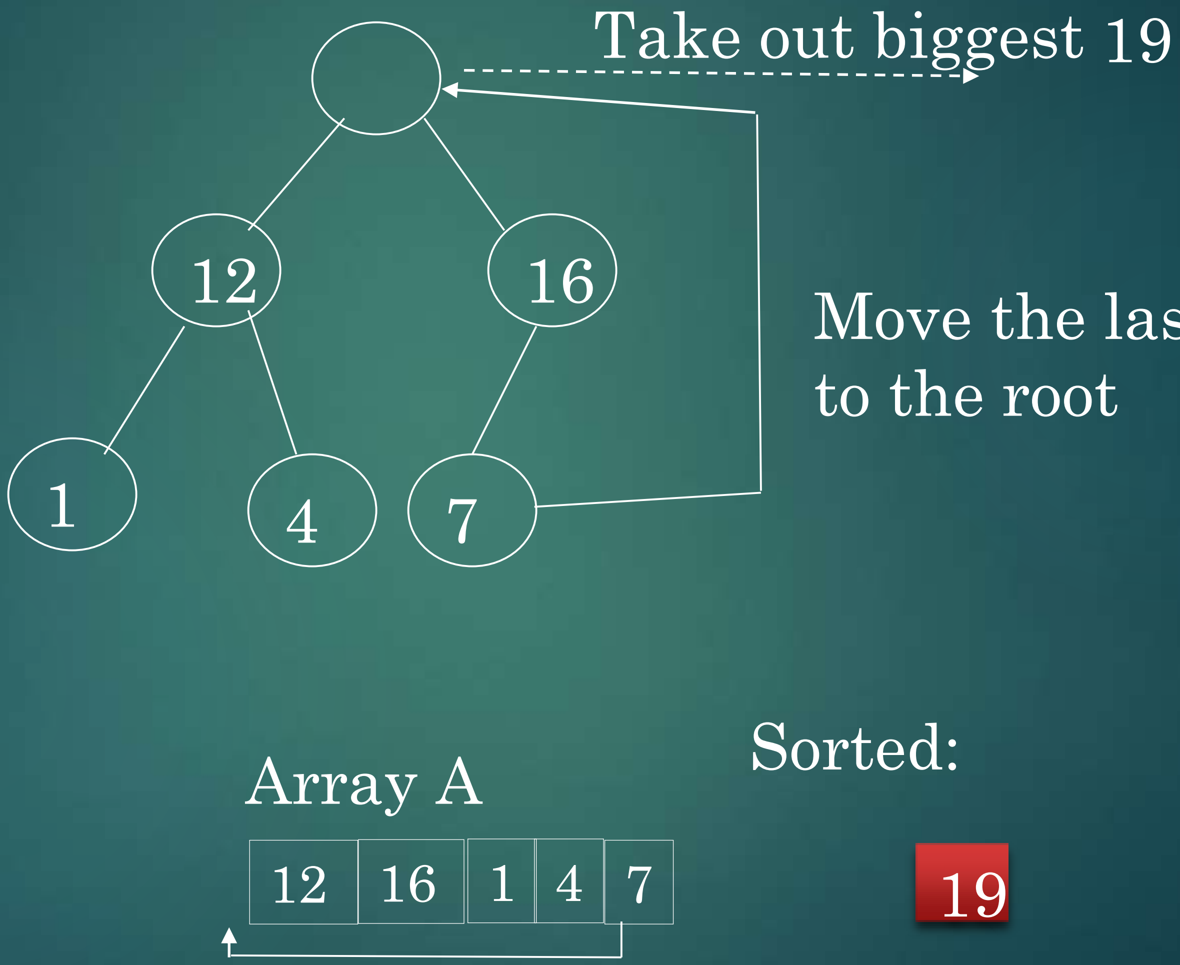
16	4	7	1	12	19

Picture **the array as a complete binary tree:**



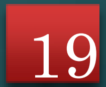

16
4
7
1
12
19
16
4
19
1
12
7
swap
16
12
19
1
4
7
swap
19
12
16
1
4
7
swap

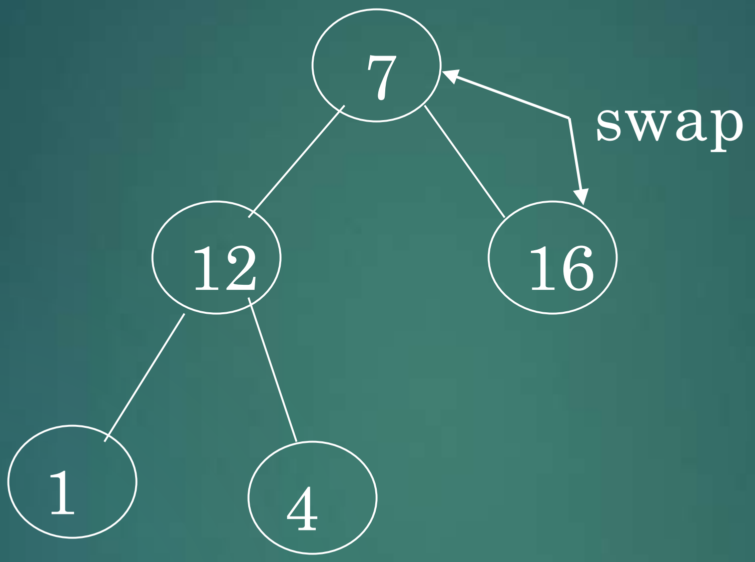
Example of Heap Sort



Move the last element to the root

Sorted:



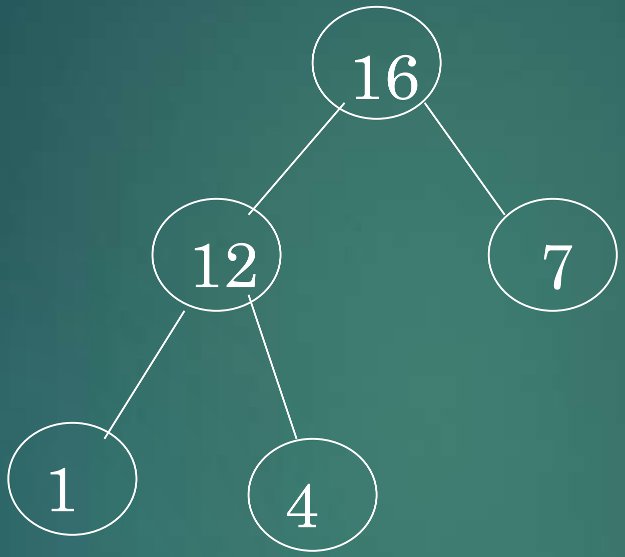


Array A

7	12	16	1	4

Sorted:

19

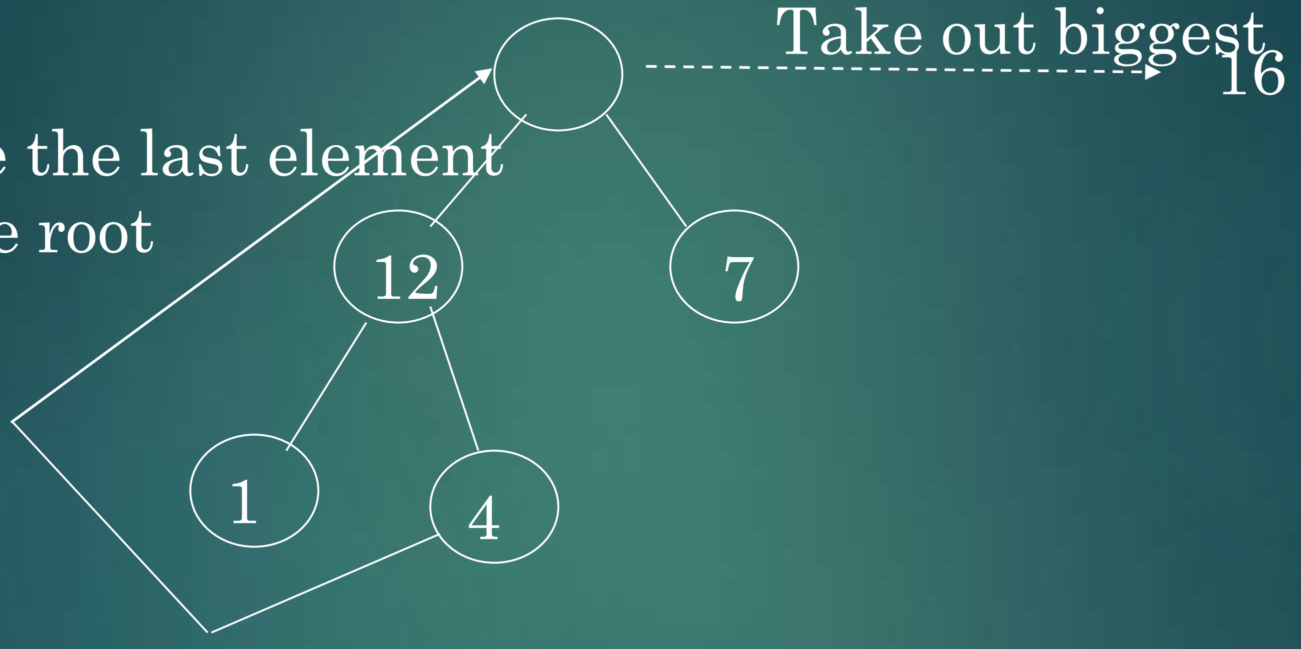


Array A

16	12	7	1	4

Sorted:

19

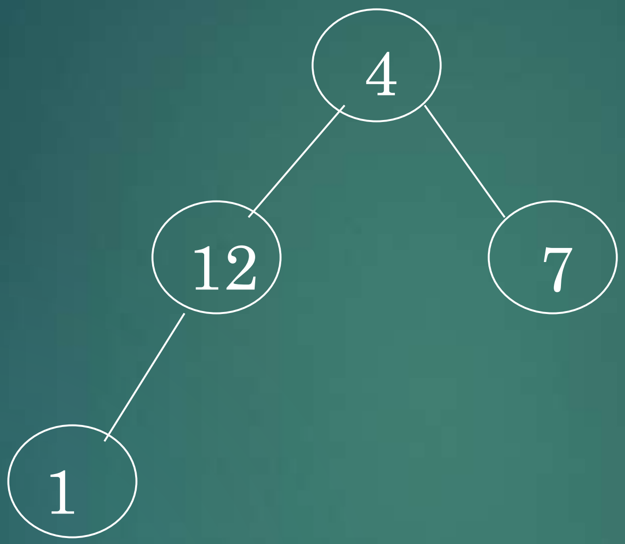


Array A

12	7	1	4

Sorted:

16	19



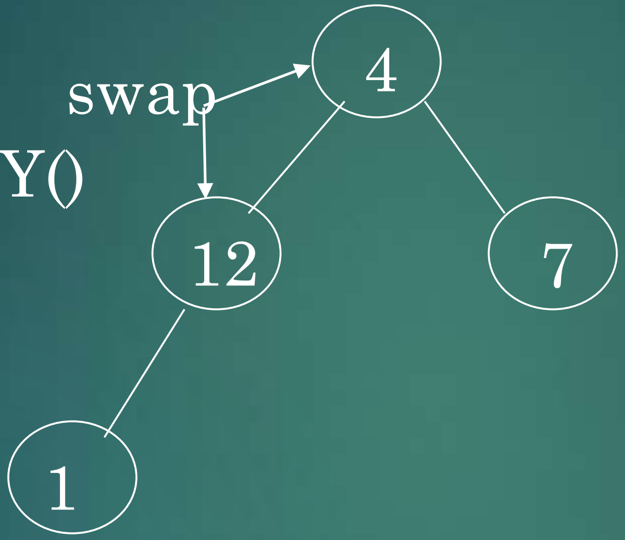
Array A

4	12	7	1

Sorted:

16	19

HEAPIFY()

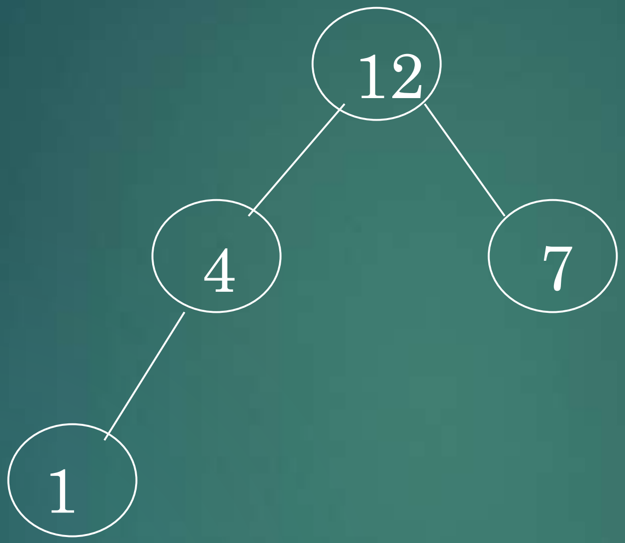


Array A

4	12	7	1

Sorted:

16	19



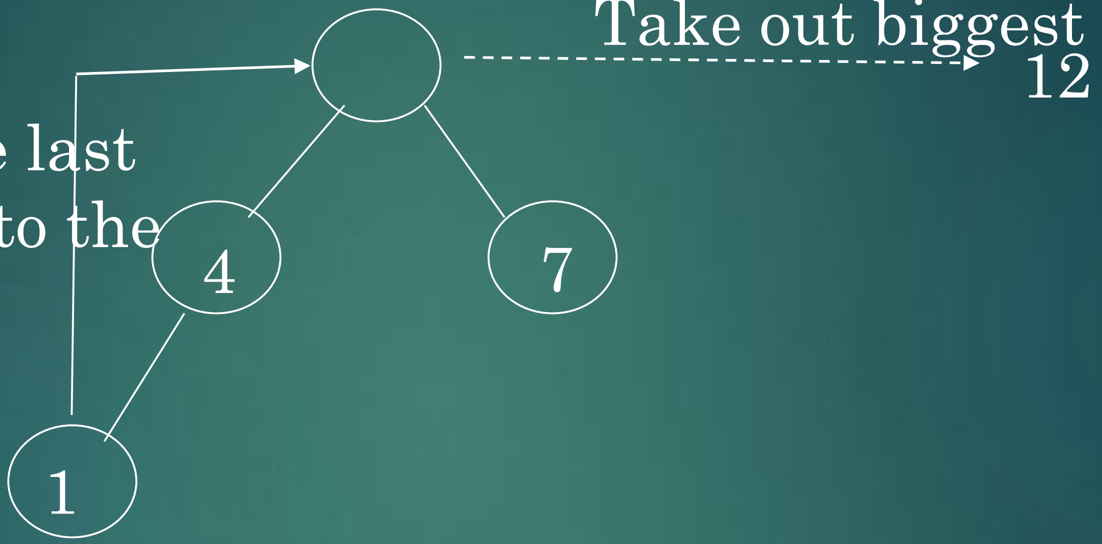
Array A

12	4	7	1

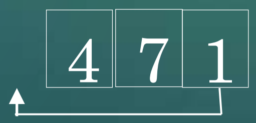
Sorted:

16	19

Move the last element to the root

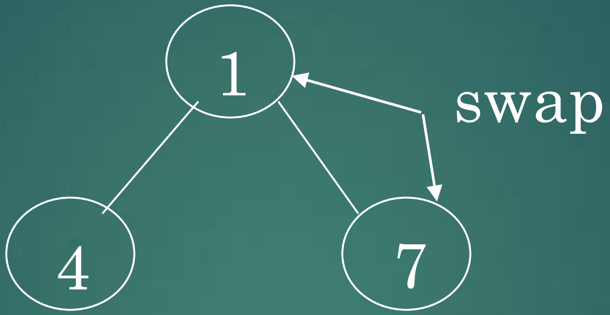


Array A



Sorted:

12	16	19

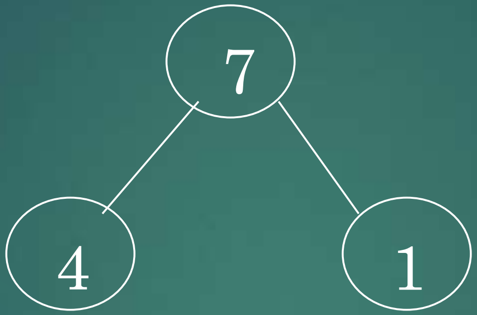


Array A

1	4	7

Sorted:

12	16	19

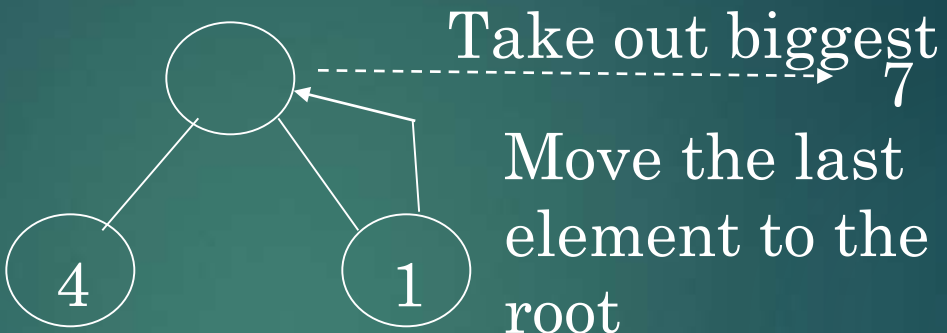


Array A

7	4	1

Sorted:

12	16	19



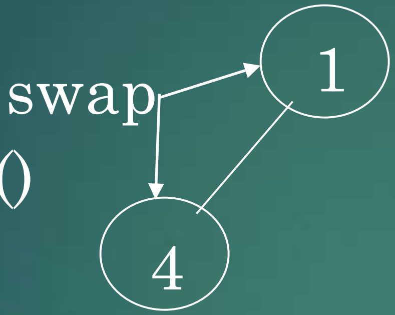
Array A

1	4

Sorted:

7	12	16	19

HEAPIFY()



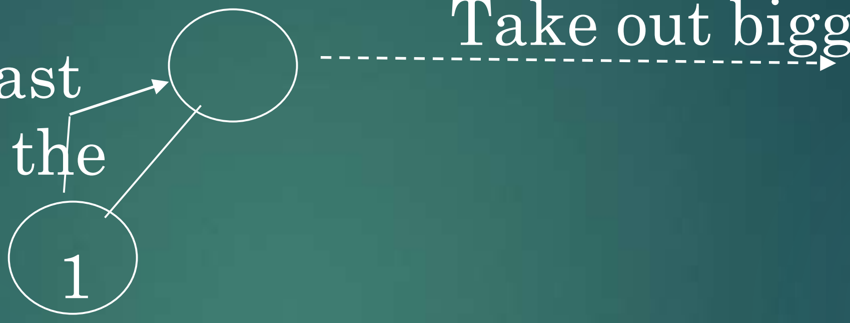
Array A

4	1

Sorted:

7	12	16	19

Move the last element to the root



Take out biggest
4

Array A

1

Sorted:

4	7	12	16	19

Take out biggest

Array A

Sorted:

1	4	7	12	16	19

Sorted:

1	4	7	12	16	19

Possible Application

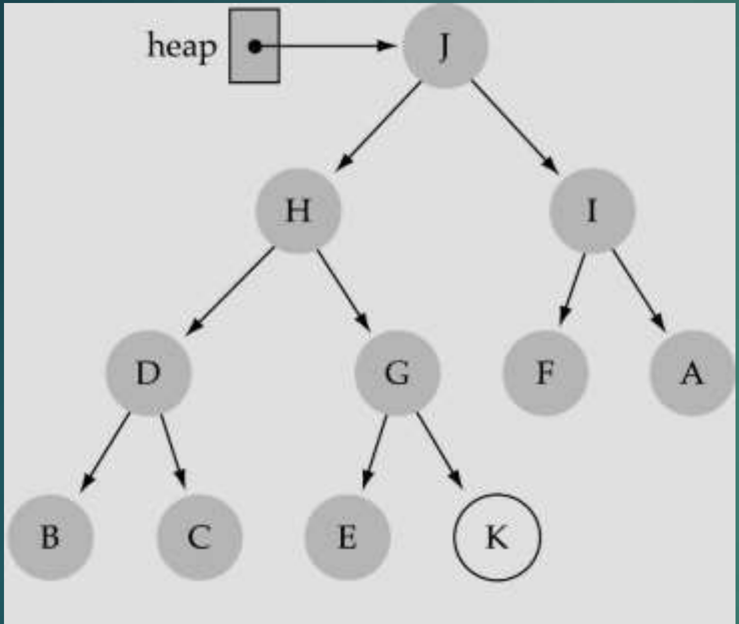
- When we want to know the task that carry the highest priority given a large number of things to do
- Interval scheduling, when we have a lists of certain task with start and finish times and we want to do as many tasks as possible
- Sorting a list of elements that needs and efficient sorting algorithm

Inserting a new element

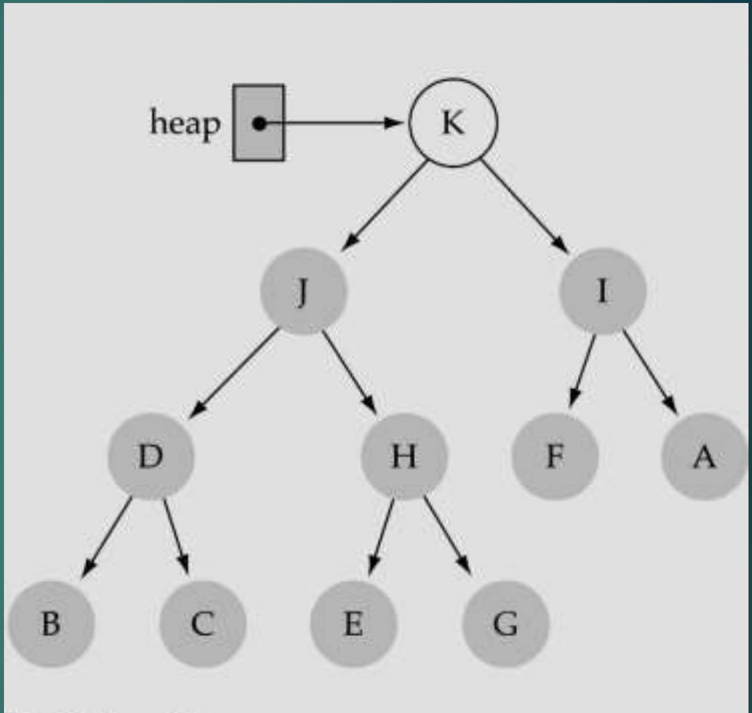
1) (1) Insert the new element in the next bottom leftmost place

2) (2) Fix the heap property by calling *ReheapUp*

Inserting a new element

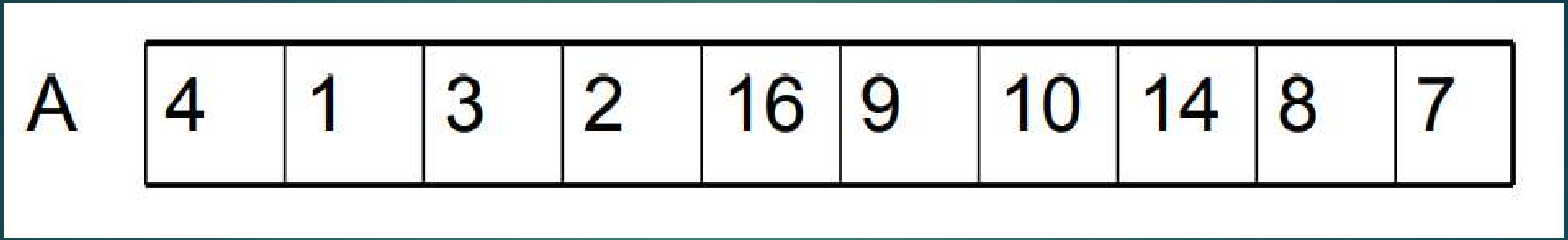


(a) Add K

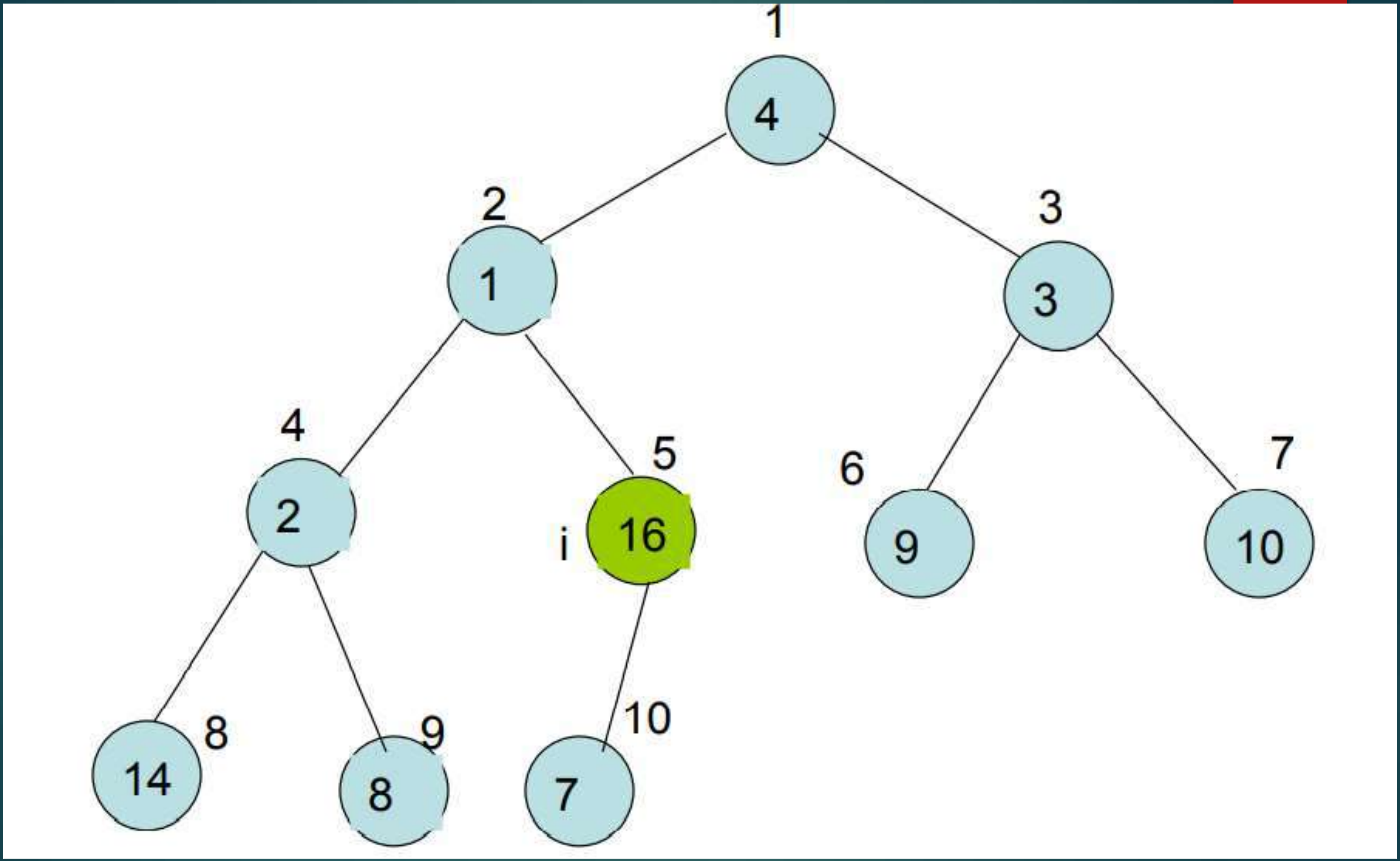


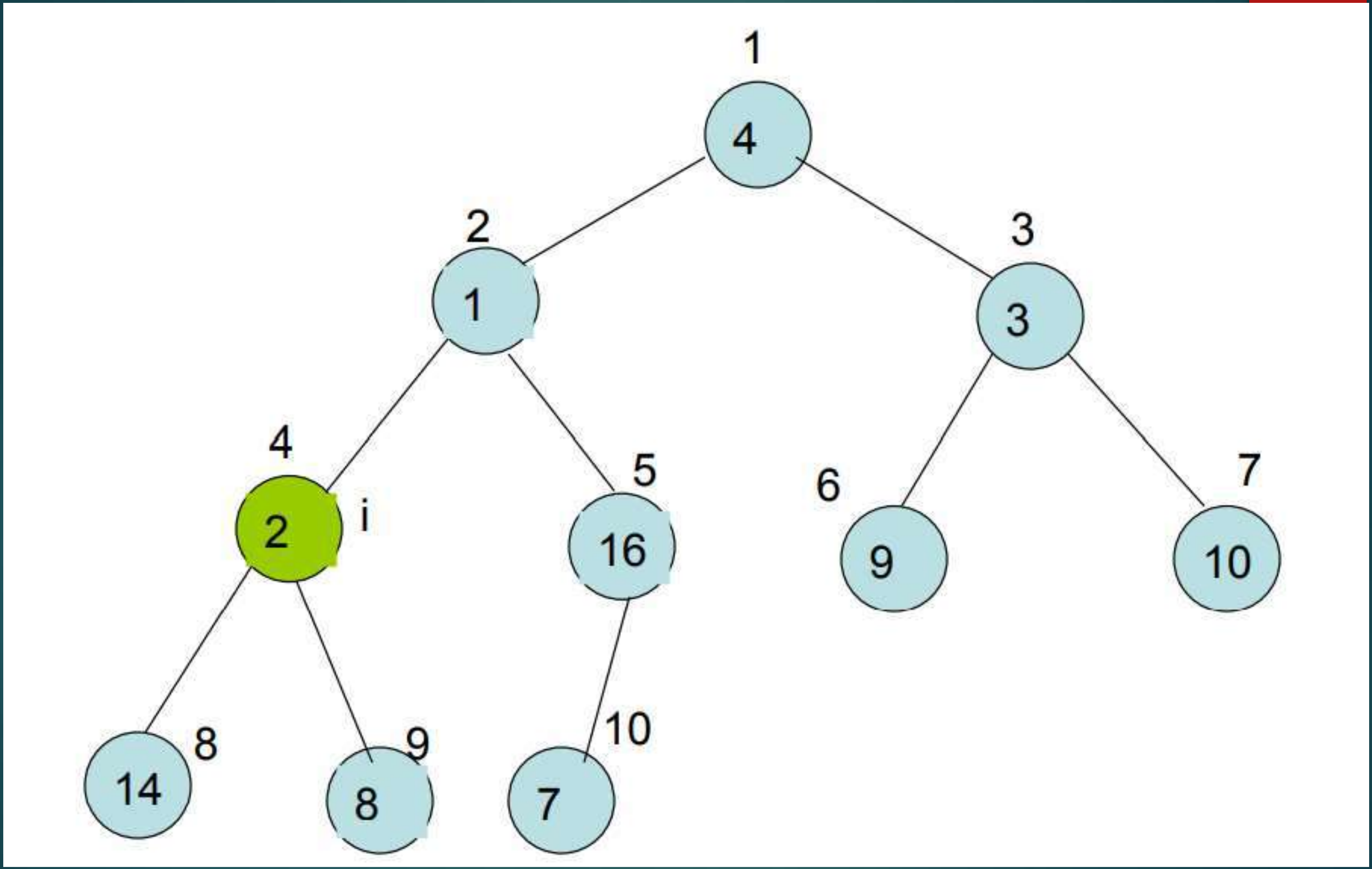
(b) ReheapUp

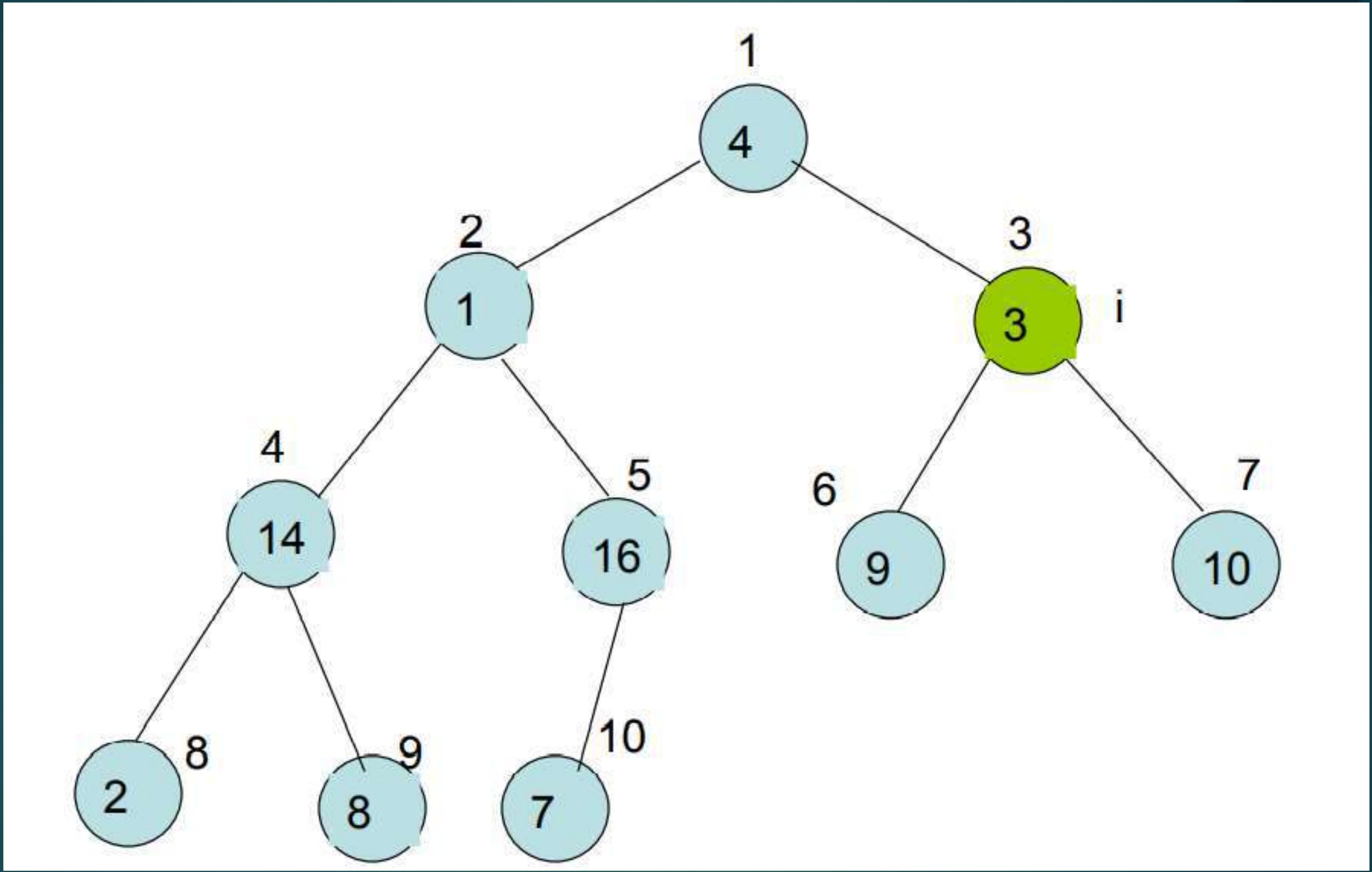
Building a Heap

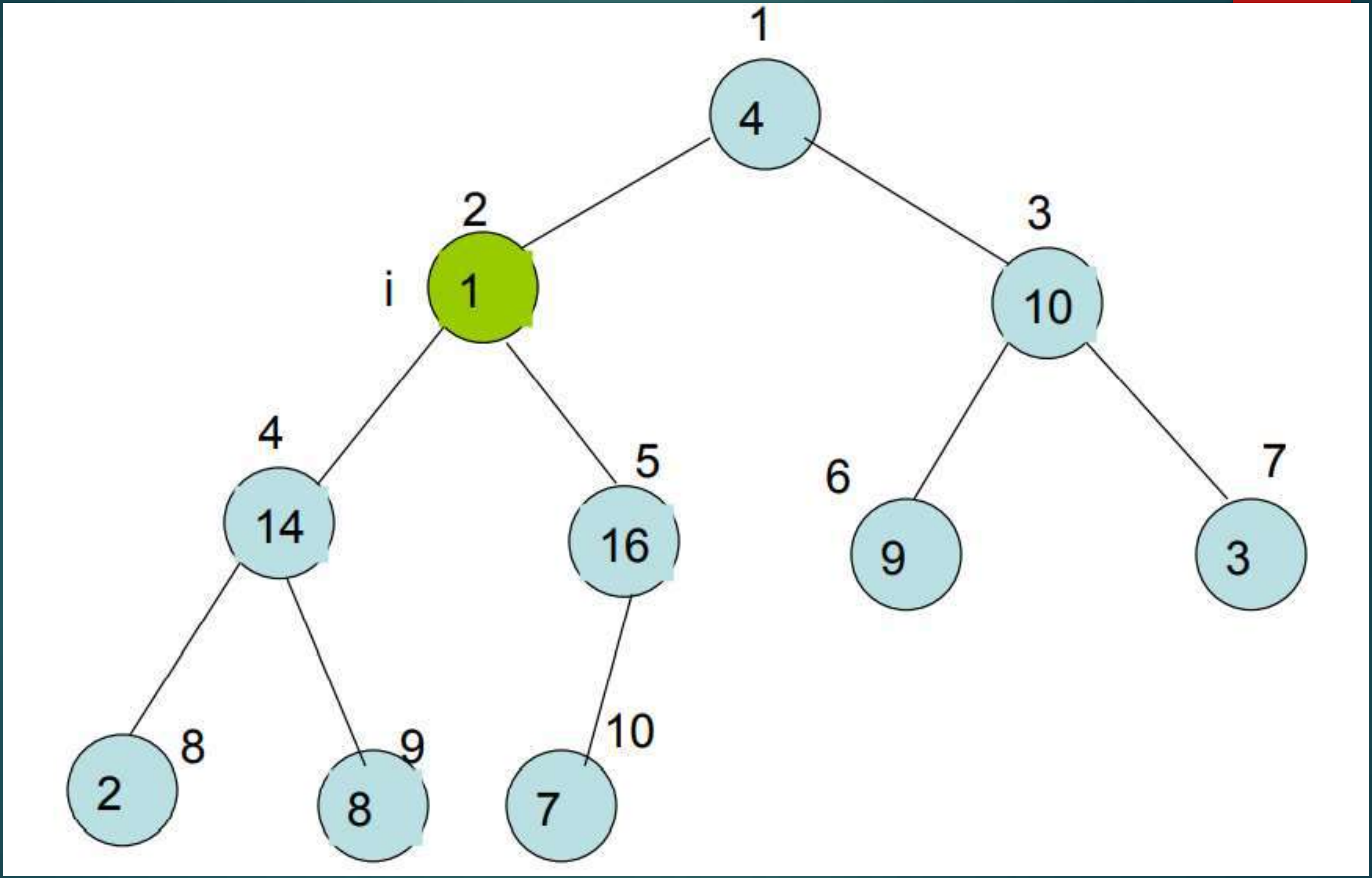


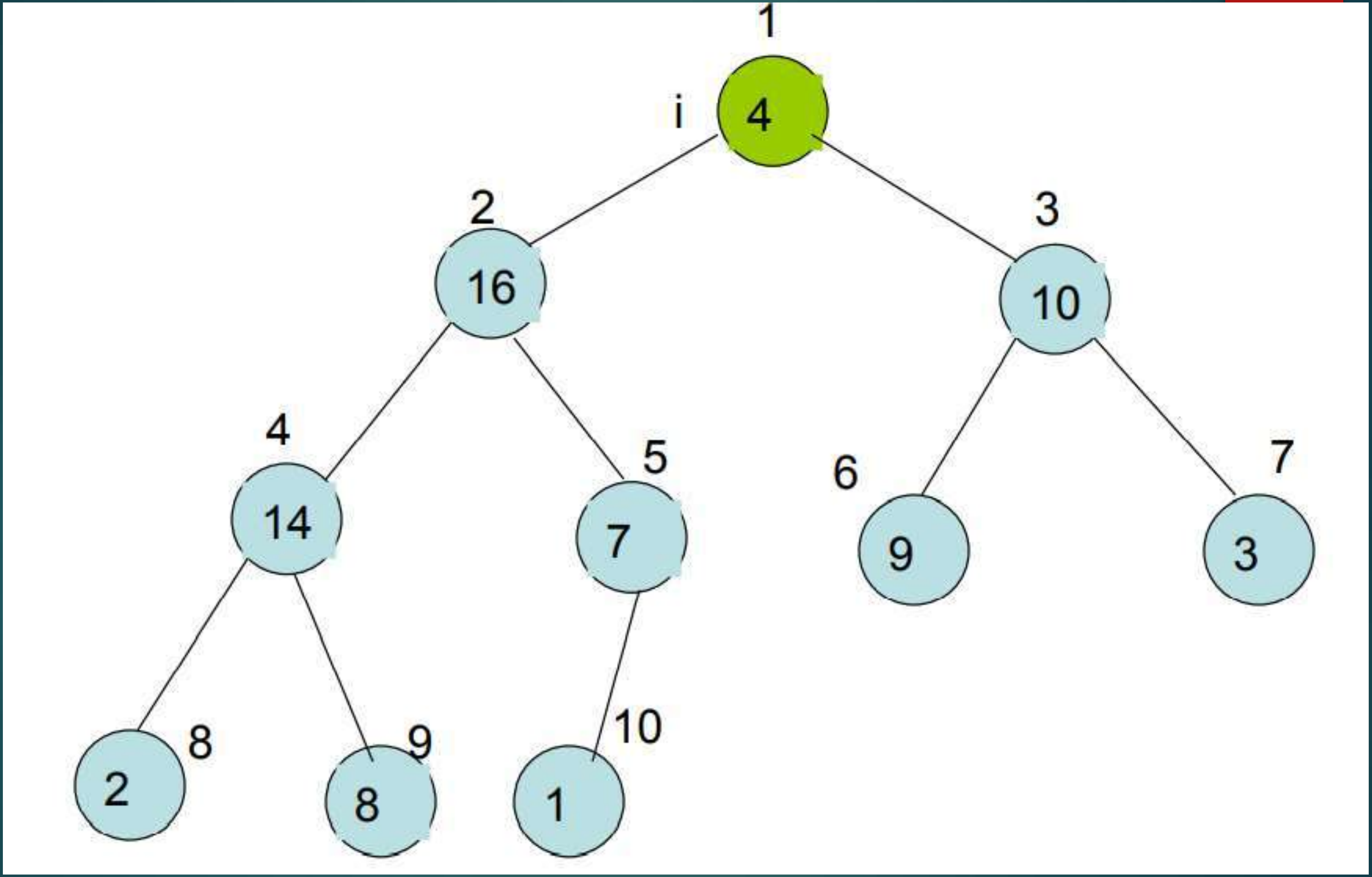
A	4	1	3	2	16	9	10	14	8	7

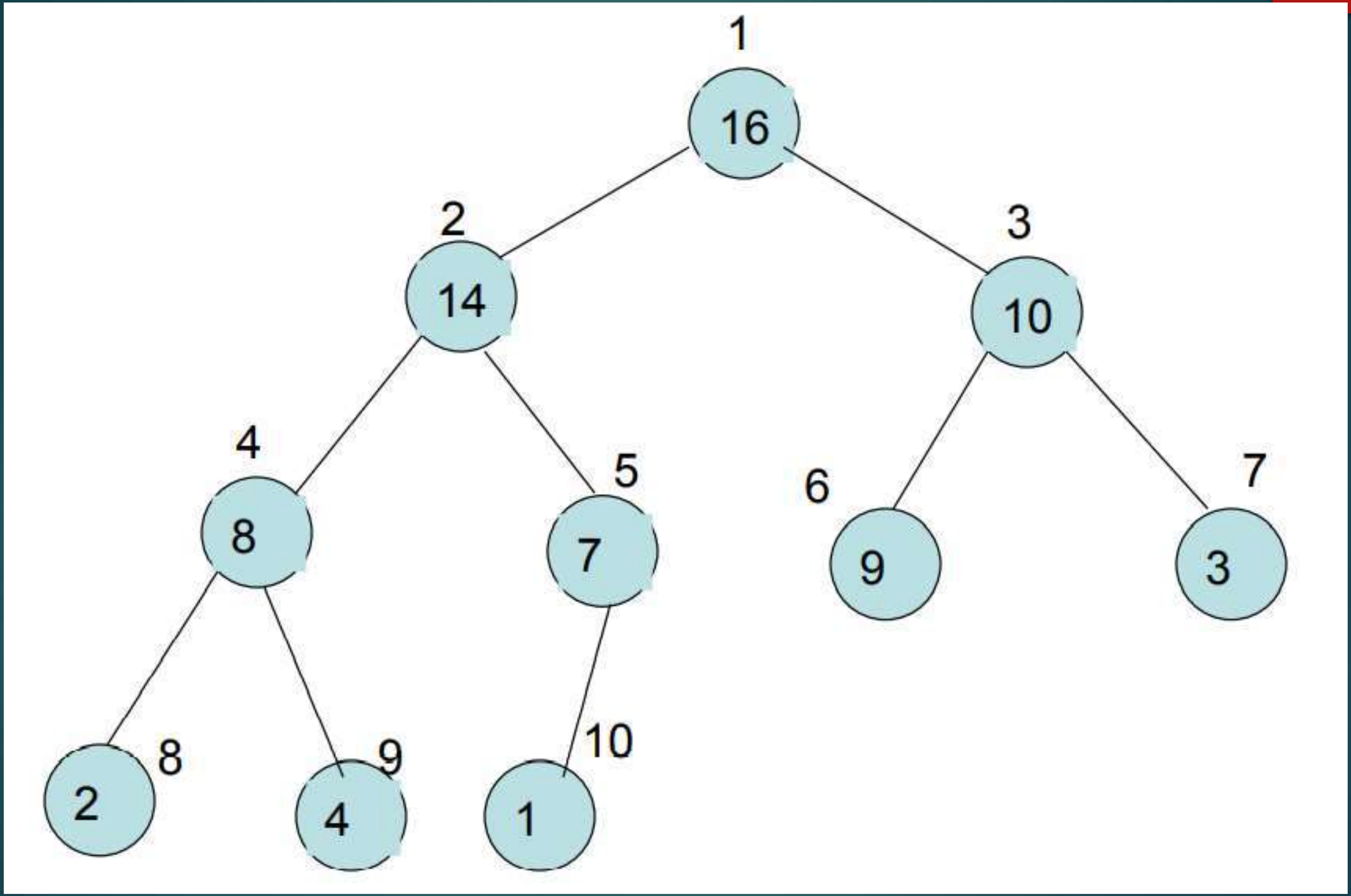

1
4
2
1
3
3
4
2
5
16
i
6
9
7
10
8
14
9
8
10
7


1
4
2
1
3
3
4
2
i
5
16
6
9
7
10
8
14
9
8
10
7


1
4
2
1
3
3
i
4
14
5
16
6
9
7
10
8
2
9
8
10
7


1
4
2
1
i
3
10
4
14
5
16
6
9
7
3
8
2
9
8
10
7


1
i
4
2
16
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1


1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
4
10
1

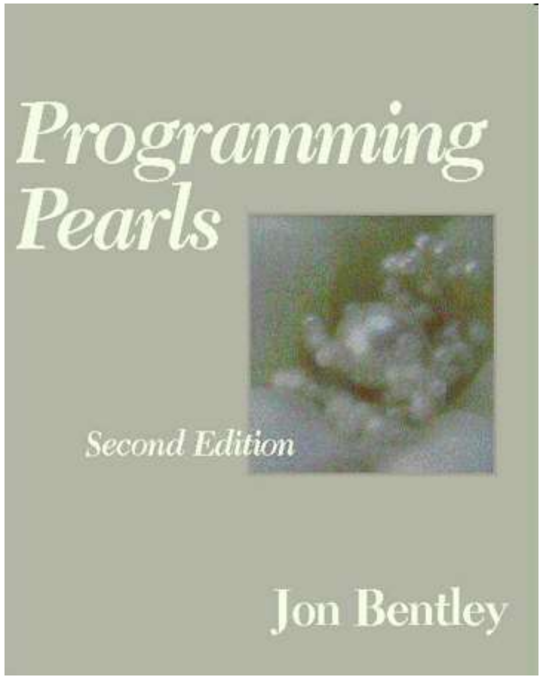
Hash Tables and Maps

"**hash collision** n.

[from the techspeak] (var. `hash clash') When used of people, signifies a confusion in associative memory or imagination, especially a persistent one (see **thinko**). True story: One of us was once on the phone with a friend about to move out to Berkeley. When asked what he expected Berkeley to be like, the friend replied: "Well, I have this mental picture of naked women throwing Molotov cocktails, but I think that's just a collision in my hash tables."

-The Hacker's Dictionary

Programming Pearls by Jon Bentley



- Jon was *senior programmer* on a large programming project.
- Senior programmer spend a lot of time helping junior programmers.
- Junior programmer to Jon: "I need help writing a *<u>sorting algorithm</u>*."

A Problem

- From *Programming Pearls (Jon in Italics)*

Why do you want to write your own sort at all? Why not use a sort provided by your system?
I need the sort in the middle of a large system, and for obscure technical reasons, I can't use the system file-sorting program.
What exactly are you sorting? How many records are in the file? What is the format of each record?
The file contains at most ten million records; each record is a seven-digit integer.
Wait a minute. If the file is that small, why bother going to disk at all? Why not just sort it in main memory?
Although the machine has many megabytes of main memory, this function is part of a big system. I expect that I'll have only about a megabyte free at that point.
Is there anything else you can tell me about the records?
Each one is a seven-digit positive integer with no other associated data, and no integer can appear more than once.

Questions

- When did this conversation take place?
- What were they sorting?
- How do you sort data when it won't all fit into main memory?
- Speed of file i/o?

A Solution

```
/* phase 1: initialize set to empty */
for i = [0, n)
    bit[i] = 0

/* phase 2: insert present elements into the set */
for each i in the input file
    bit[i] = 1

/* phase 3: write sorted output */
for i = [0, n)
    if bit[i] == 1 write i on the output file
```

Some Structures so Far

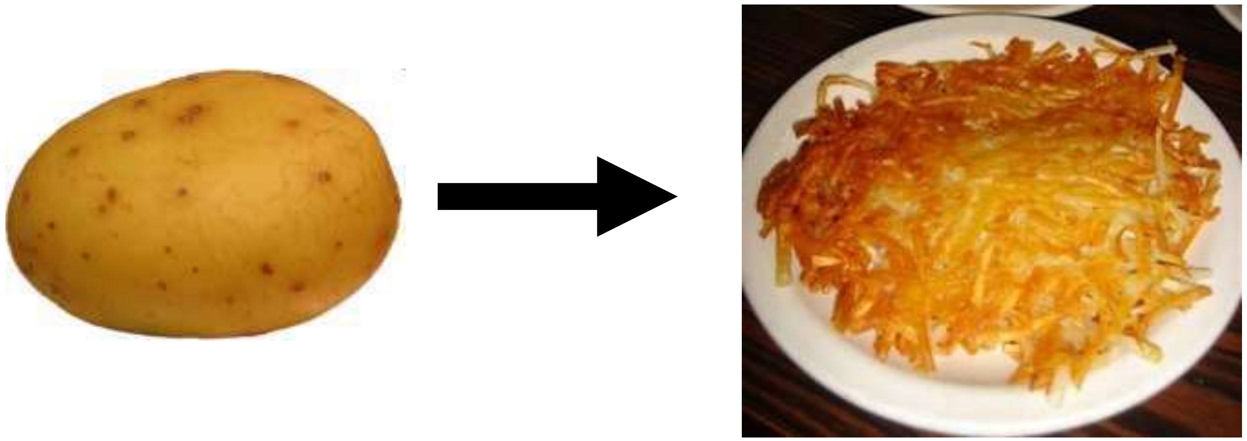
- ArrayLists
  - O(1) access
  - O(N) insertion (average case), better at end
  - O(N) deletion (average case)
- LinkedLists
  - O(N) access
  - O(N) insertion (average case), better at front and back
  - O(N) deletion (average case), better at front and back
- Binary Search Trees
  - O(log N) access if balanced
  - O(log N) insertion if balanced
  - O(log N) deletion if balanced

Why are Binary Trees Better?

- Divide and Conquer
  - reducing work by a factor of 2 each time
- Can we reduce the work by a bigger factor? 10? 1000?
- An ArrayList does this in a way when *accessing* elements
  - *but must use an integer value*
  - *each position holds a single element*

Hash Tables

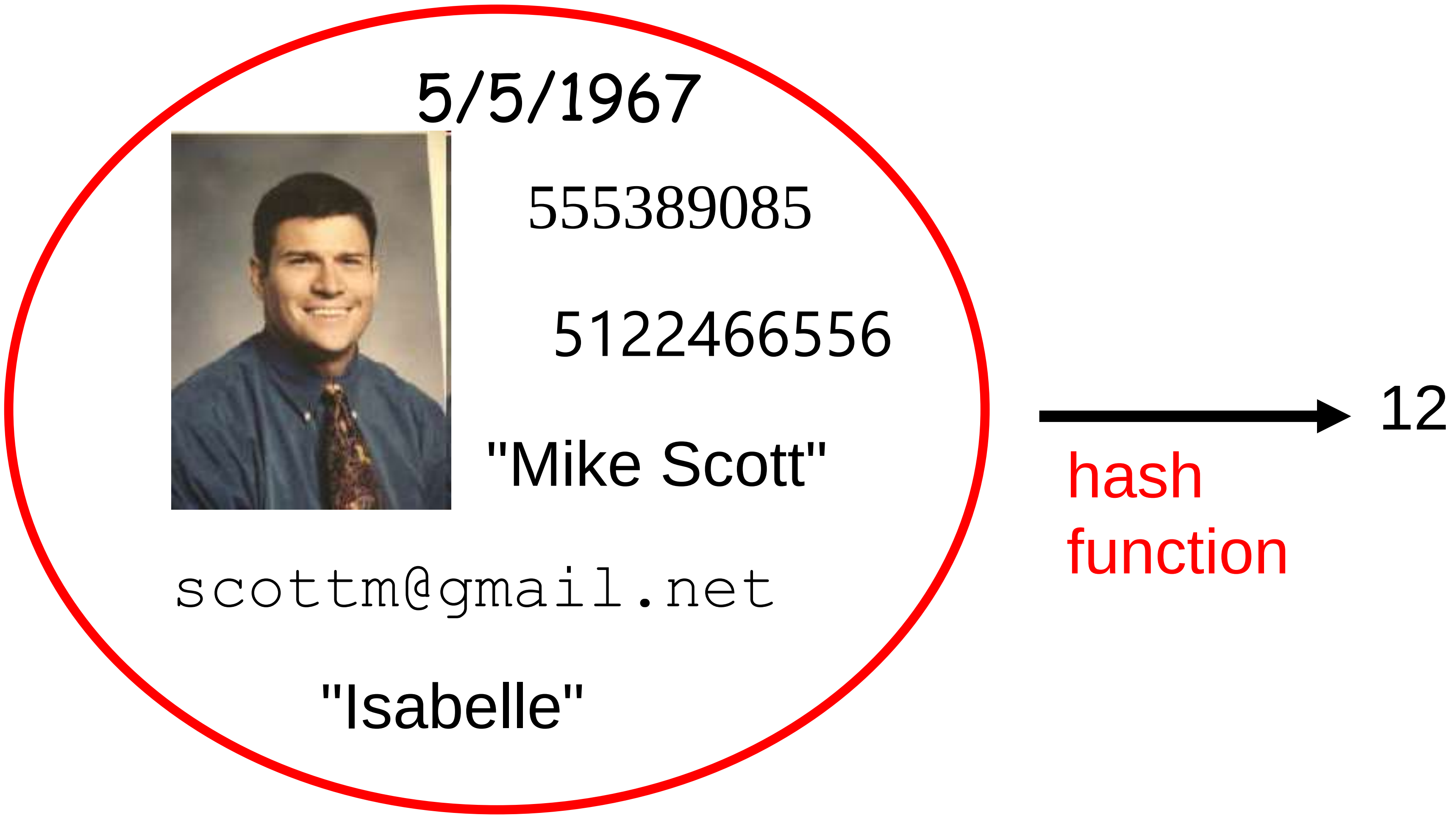
- Hash Tables overcome the problems of ArrayList while maintaining the fast access, insertion, and deletion in terms of N (number of elements already in the structure.)



Hash Functions

- Hash: "From the French hatcher, which means 'to chop'. "
- *to hash* to mix randomly or shuffle (To cut up, to slash or hack about; to mangle)
- Hash Function: Take a large piece of data and reduce it to a smaller piece of data, usually a single integer.
  - A function or algorithm
  - The input need not be integers!

Hash Function



Simple Example

- Assume we are using names as our *key*
  - *take 3rd letter of name, take int value of letter (a = 0, b = 1, ...), divide by 6 and take remainder*
- What does "Bellers" hash to?
- L -> 11 -> 11 % 6 = 5

Result of Hash Function

- Mike = (10 % 6) = 4
- Kelly = (11 % 6) = 5
- Olivia = (8 % 6) = 2
- Isabelle = (0 % 6) = 0
- David = (21 % 6) = 3
- Margaret = (17 % 6) = 5 (uh oh)
- Wendy = (13 % 6) = 1
- This is an imperfect hash function. A perfect hash function yields a one to one mapping from the keys to the hash values.
- What is the maximum number of values this function can hash perfectly?

More on Hash Functions

- Normally a two step process
  - transform the key (which may not be an integer) into an integer value
  - Map the resulting integer into a valid index for the hash table (where all the elements are stored)
- The transformation can use one of four techniques
  - mapping, folding, shifting, casting

Hashing Techniques

- Mapping
  - As seen in the example
  - integer values or things that can be easily converted to integer values in key
- Folding
  - partition key into several parts and the integer values for the various parts are combined
  - the parts may be hashed first
  - combine using addition, multiplication, shifting, logical exclusive OR

More Techniques

- Shifting
  - an alternative to folding
  - A fold function

```
int hashVal = 0;
int i = str.length() - 1;
while(i > 0){
    hashVal += (int) str.charAt(i);
    i--;
}
```

results for "dog" and "god" ?

Shifting and Casting

- More complicated with shifting

```
int hashVal = 0;
int i = str.length() - 1;
while(i > 0)
{ hashVal = (hashVal << 1) + (int) str.charAt(i);
  i--;
}
```

different answers for "dog" and "god"

Shifting may give a better range of hash values when compared to just folding

Casts

- Very simple
  - essentially casting as part of fold and shift when working with chars.

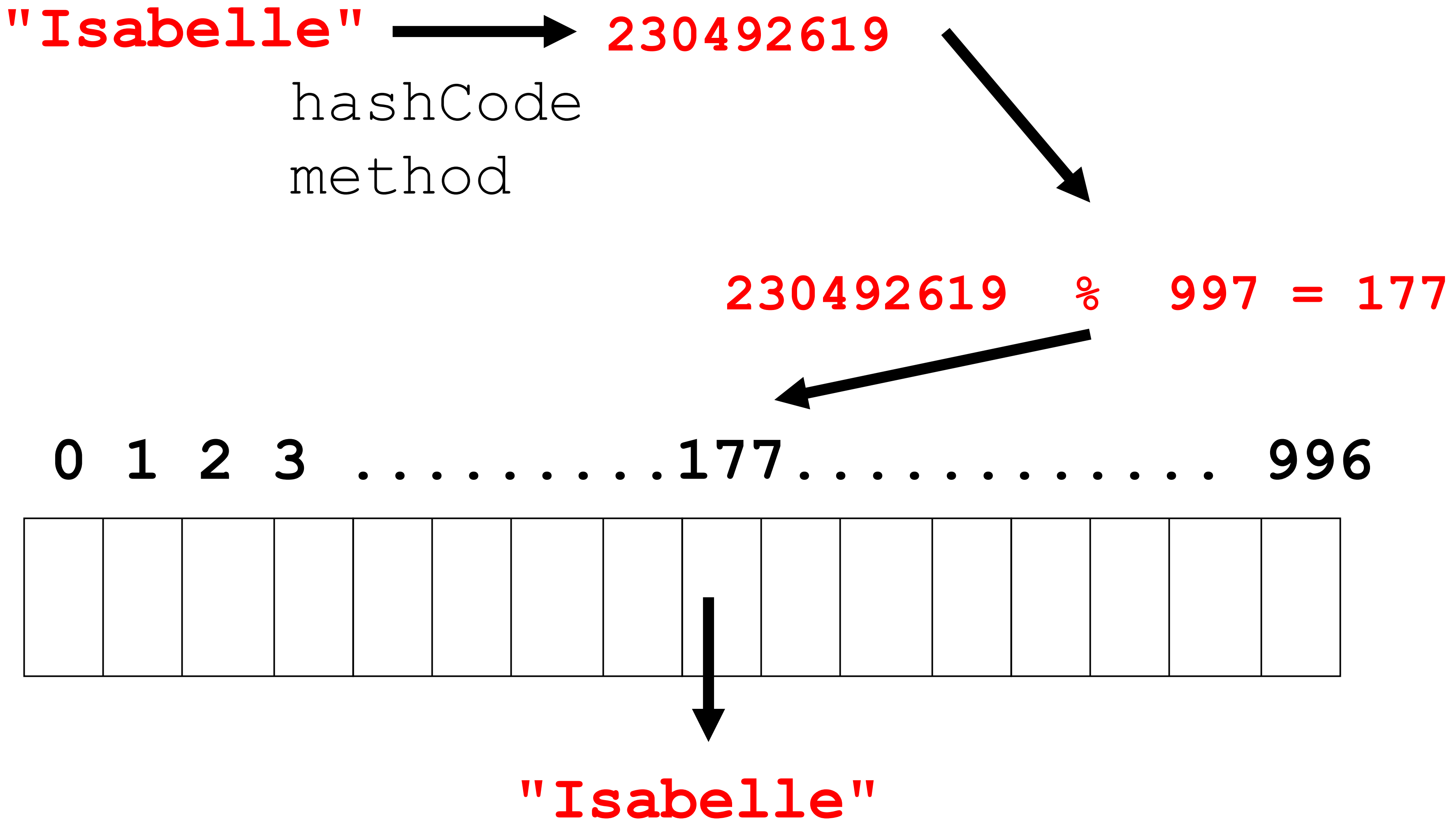
The Java String class hashCode method

```
public int hashCode()
{   int h = hash;
    if (h == 0)
    {   int off = offset;
       char val[] = value;
       int len = count;
        for (int i = 0; i < len; i++)
        {   h = 31*h + val[off++];
        }
        hash = h;
    }
    return h;
}
```

Mapping Results

- Transform hashed key value into a legal index in the hash table
- Hash table is normally uses an array as its underlying storage container
- Normally get location on table by taking result of hash function, dividing by size of table, and taking remainder
  - index = key mod n
  - n is size of hash table
  - empirical evidence shows a prime number is best
  - 1000 element hash table, make 997 or 1009 elements

Mapping Results



Handling Collisions

- What to do when inserting an element and already something present?

Open Address Hashing

- Could search forward or backwards for an open space
- Linear probing:
  - move forward 1 spot. Open?, 2 spots, 3 spots
  - reach the end?
  - When removing, insert a blank
  - null if never occupied, blank if once occupied
- Quadratic probing
  - 1 spot, 2 spots, 4 spots, 8 spots, 16 spots
- Resize when *load factor* reaches some limit

Chaining

- Each element of hash table be another data structure
  - linked list, balanced binary tree
  - More space, but somewhat easier
  - everything goes in its spot
- Resize at given load factor or when any chain reaches some limit: (relatively small number of items)
- What happens when resizing?
  - Why don't things just collide again?

Hash Tables in Java

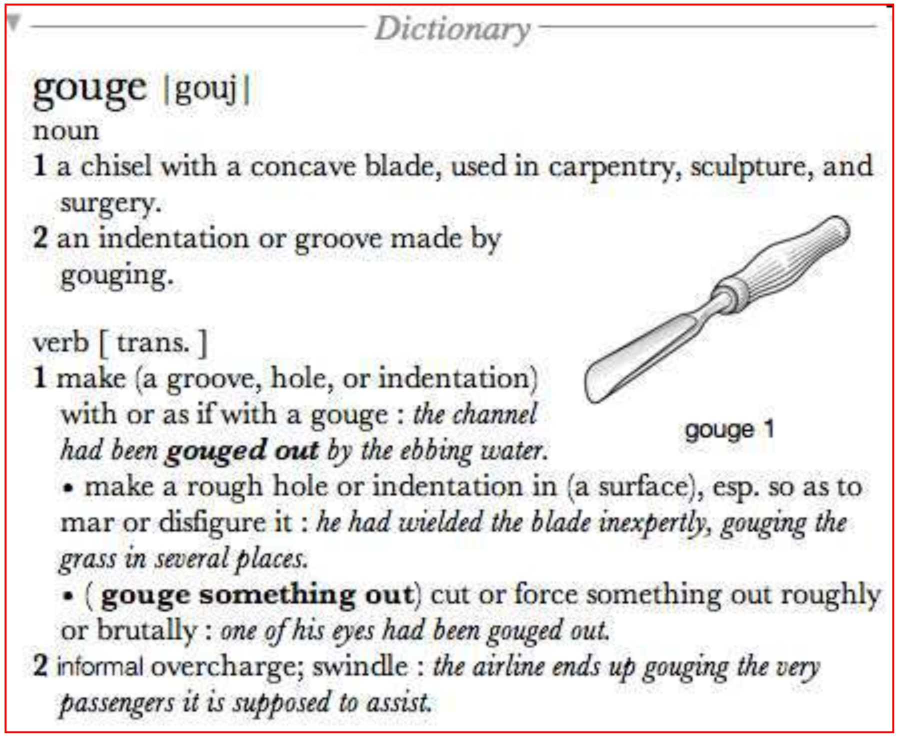
- `hashCode` method in `Object`
- `hashCode` and `equals`
  - "If two objects are equal according to the equals (`Object`) method, then calling the `hashCode` method on each of the two objects must produce the same integer result. "
  - if you override `equals` you need to override `hashCode`

Hash Tables in Java

- HashTable class
- HashSet class
  - implements Set interface with internal storage container that is a HashTable
  - compare to TreeSet class, internal storage container is a Red Black Tree
- HashMap class
  - implements the Map interface, internal storage container for keys is a hash table

Maps (a.k.a. Dictionaries)

```
A -> 65
```



Dictionary

gouge |gouj|

noun

1 a chisel with a concave blade, used in carpentry, sculpture, and surgery.

2 an indentation or groove made by gouging.

verb [trans.]

1 make (a groove, hole, or indentation) with or as if with a gouge : *the channel had been* ***gouged out*** *by the ebbing water.*

• make a rough hole or indentation in (a surface), esp. so as to mar or disfigure it : *he had wielded the blade inexpertly, gouging the grass in several places.*

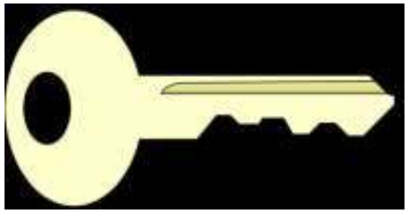
• (**gouge something out**) cut or force something out roughly or brutally : *one of his eyes had been gouged out.*

2 informal overcharge; swindle : *the airline ends up gouging the very passengers it is supposed to assist.*

Maps

- Also known as:
  - table, search table, dictionary, associative array, or associative container
- A data structure optimized for a very specific kind of search / access
  - with a *bag* we access by asking "is X present"
  - with a *list* we access by asking "give me item number X"
  - with a *queue* we access by asking "give me the item that has been in the collection the longest."
- In a *map* we access by asking "give me the *value* associated with this *key."*

Keys and Values



- Dictionary Analogy:
  - The *key* in a dictionary is a word: *foo*
  - The *value* in a dictionary is the definition: *First on the standard list of metasyntactic variables used in syntax examples*
- A key and its associated value form a pair that is stored in a map
- To retrieve a value the key for that value must be supplied
  - A List can be viewed as a Map with integer keys

More on Keys and Values

- Keys must be unique, meaning a given key can only represent one value
  - but one value may be represented by multiple keys
  - like synonyms in the dictionary.
    Example:
    factor: n.See coefficient of X
  - *factor* is a key associated with the same value (definition) as the key *coefficient of X*

The Map<K, V> Interface in Java

- `void clear()`
  - Removes all mappings from this map (optional operation).
- `boolean containsKey(Object key)`
  - Returns true if this map contains a mapping for the specified key.
- `boolean containsValue(Object value)`
  - Returns true if this map maps one or more keys to the specified value.
- `Set<K> keySet()`
  - Returns a Set view of the keys contained in this map.

The Map Interface Continued

- `V get(Object key)`
  - Returns the value to which this map maps the specified key.
- `boolean isEmpty()`
  - Returns true if this map contains no key-value mappings.
- `V put(K key, V value)`
  - Associates the specified value with the specified key in this map

The Map Interface Continued

- `Vremove(Object key)`
  - Removes the mapping for this key from this map if it is present
- `int size()`
  - Returns the number of key-value mappings in this map.
- `Collection<V> values()`
  - Returns a collection view of the values contained in this map.

Implementing a Map

- Two common implementations of maps are to use a binary search tree or a hash table as the internal storage container
  - HashMap and TreeMap are two of the implementations of the Map interface
- `HashMap` uses a hash table as its internal storage container.
  - keys stored based on hash codes and size of hash tables internal array

TreeMap implementation

- Uses a Red - Black tree to implement a Map
- relies on the `compareTo` method of the keys
- somewhat slower than the HashMap
- keys stored in sorted order

Sample Map Problem

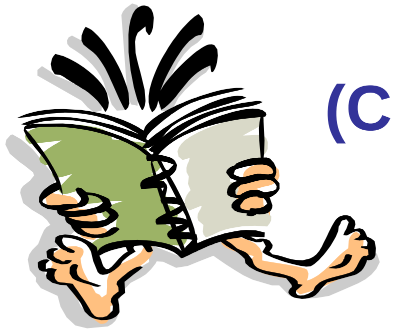
Determine the frequency of words in a file.

```
File f = new File(fileName);
Scanner s = new Scanner(f);
Map<String, Integer> counts =
      new Map<String, Integer>();
while( s.hasNext() ){
  String word = s.next();
  if( !counts.containsKey( word ) )
    counts.put( word, 1 );
  else
    counts.put( word,
        counts.get(word) + 1 );
}
```

Analysis of Algorithms
CS 477/677

Hashing

Instructor: George Bebis



(Chapter 11)

The Search Problem

- Find items with **keys** matching a given **search key**
  - Given an array A, containing n keys, and a search key x, find the index i such as x=A[i]
  - As in the case of sorting, a key could be part of a large record.

example of a record

Key	**other data**

Applications

- Keeping track of customer account information at a bank
  – Search through records to check balances and perform transactions
- Keep track of reservations on flights
  – Search to find empty seats, cancel/modify reservations
- Search engine
  – Looks for all documents containing a given word

Special Case: Dictionaries

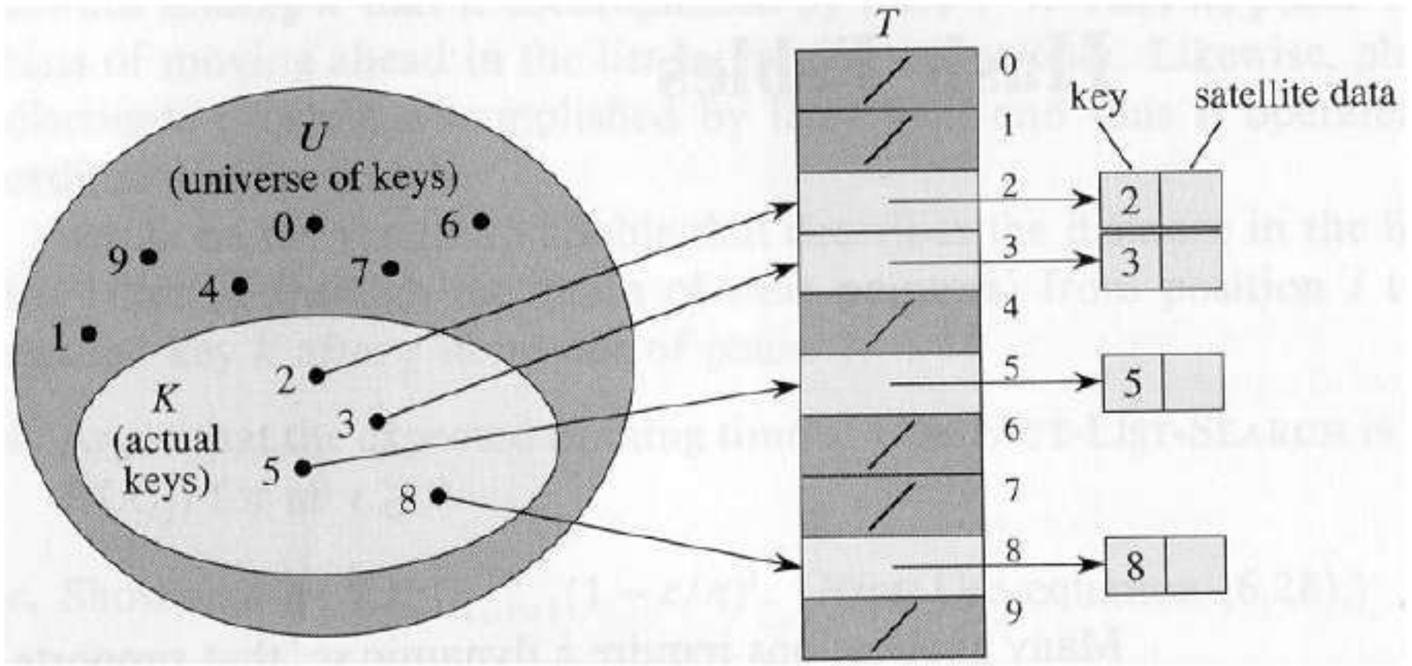
- **Dictionary** = data structure that supports mainly two basic operations: insert a new item and return an item with a given key
- Queries: return information about the set S:
  - Search (S, k)
  - Minimum (S), Maximum (S)
  - Successor (S, x), Predecessor (S, x)
- Modifying operations: change the set
  - Insert (S, k)
  - Delete (S, k) – not very often

Direct Addressing

- Assumptions:
  - Key values are distinct
  - Each key is drawn from a universe U = {0, 1, . . . , m - 1}
- Idea:
  - Store the items in an array, indexed by keys

- **Direct-address table** representation:
  - An array T[0 . . . m - 1]
  - Each **slot**, or position, in T corresponds to a key in U
  - For an element x with key k, a pointer to x (or x itself) will be placed in location T[k]
  - If there are no elements with key k in the set, T[k] is empty, represented by NIL

Direct Addressing (cont'd)



(insert/delete in O(1) time)

Operations

```
Alg.: DIRECT-ADDRESS-SEARCH(T, k)
    return T[k]

Alg.: DIRECT-ADDRESS-INSERT(T, x)
    T[key[x]] ← x

Alg.: DIRECT-ADDRESS-DELETE(T, x)
    T[key[x]] ← NIL
```

- Running time for these operations: $O(1)$

Comparing Different Implementations

- Implementing dictionaries using:
  - Direct addressing
  - Ordered/unordered arrays
  - Ordered/unordered linked lists

	Insert	Search
direct addressing	O(1)	O(1)
ordered array	O(N)	O(lgN)
ordered list	O(N)	O(N)
unordered array	O(1)	O(N)
unordered list	O(1)	O(N)

Examples Using Direct Addressing

Example 1:

(i) Suppose that the keys are integers from 1 to 100 and that there are about 100 records

(ii) Create an array A of 100 items and store the record whose key is equal to i in $A[i]$

Example 2:

(i) Suppose that the keys are nine-digit social security numbers

(ii) We can use the same strategy as before but it very inefficient now: an array of 1 billion items is needed to store 100 records !!

- $|U|$ can be very large
- $|K|$ can be much smaller than $|U|$

Hash Tables

- When K is much smaller than U, a **hash table** requires much less space than a **direct-address table**
  - Can reduce storage requirements to $|K|$
  - Can still get $O(1)$ search time, but on the <u>average</u> case, not the worst case

Hash Tables

Idea:

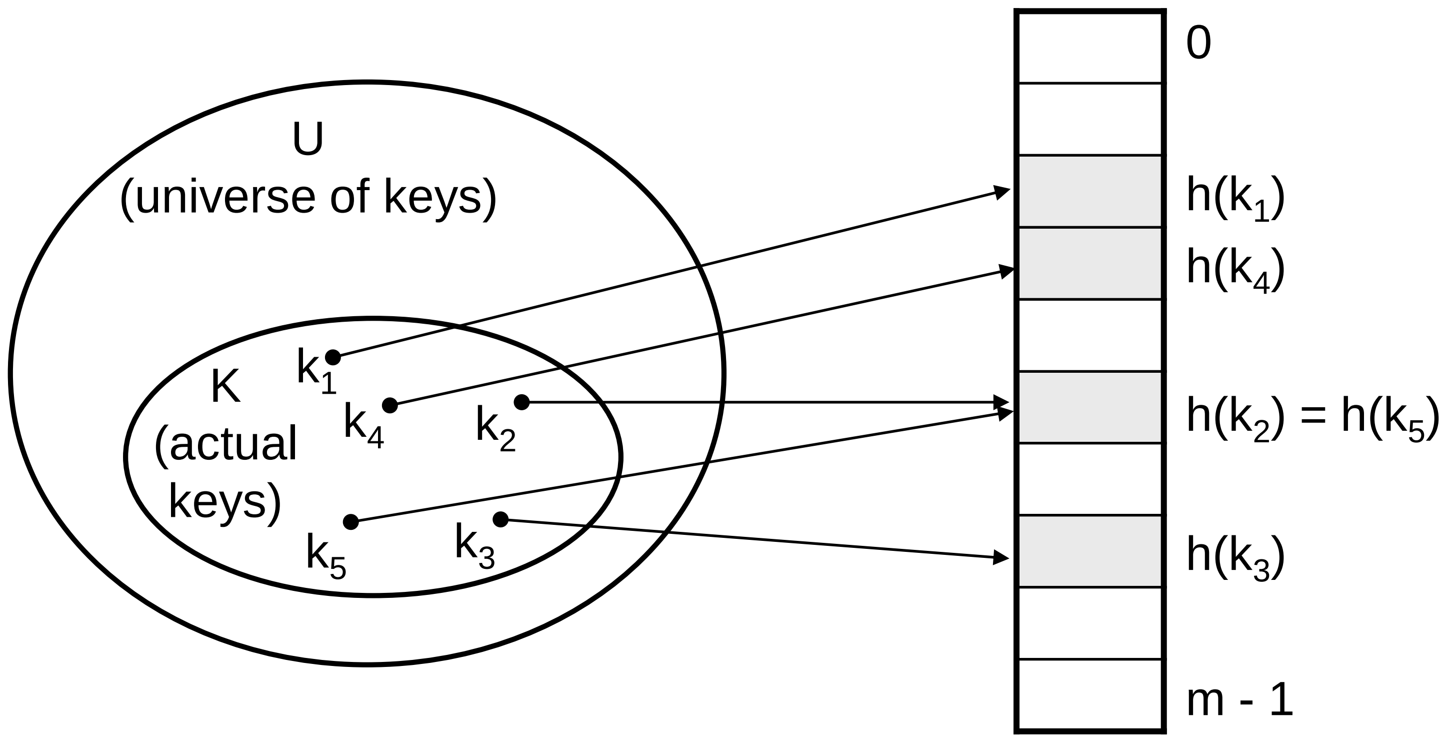
- Use a function h to compute the slot for each key
- Store the element in slot $h(k)$

- A **hash function** h transforms a key into an index in a hash table $T[0...m-1]$:

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

- We say that k **hashes** to slot $h(k)$
- Advantages:
  - Reduce the range of array indices handled: m instead of $|U|$
  - Storage is also reduced

Example: HASH TABLES



Revisit Example 2

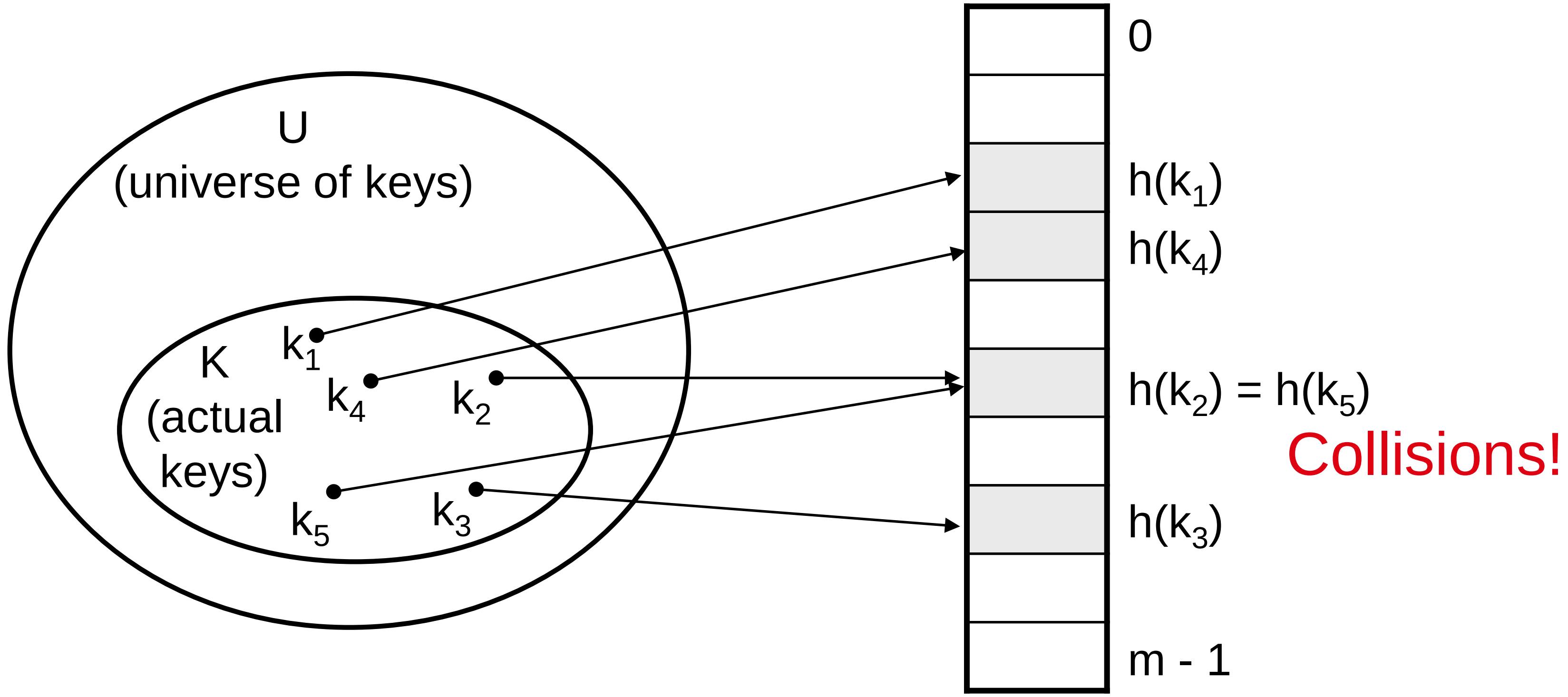
Suppose that the keys are nine-digit social security numbers

<u>Possible hash function</u>

$h(ssn) = sss$ mod 100 (last 2 digits of ssn)

e.g., if ssn = 10123411 then h(10123411) = 11)

Do you see any problems with this approach?



Collisions

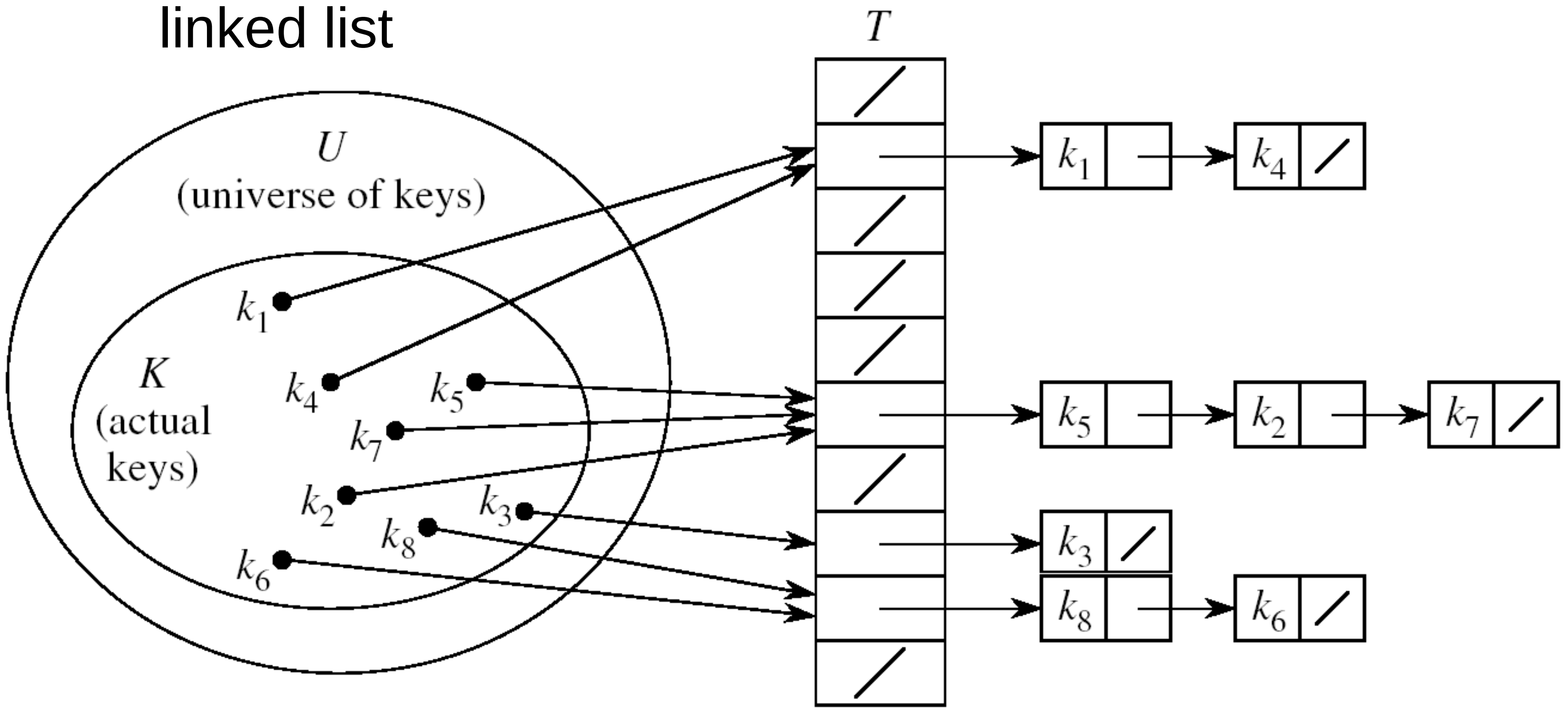
- Two or more keys hash to the same slot!!
- For a given set K of keys
  - If $|K| \leq m$, collisions may or may not happen, depending on the hash function
  - If $|K| > m$, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions completely is hard, even with a good hash function

Handling Collisions

- We will review the following methods:
  - Chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing
- We will discuss chaining first, and ways to build “good” functions.

Handling Collisions Using Chaining

- **Idea**:
  - Put all elements that hash to the same slot into a linked list



  - Slot j contains a pointer to the head of the list of all elements that hash to j

Collision with Chaining - Discussion

- Choosing the size of the table
  - Small enough not to waste space
  - Large enough such that lists remain short
  - Typically 1/5 or 1/10 of the total number of elements
- How should we keep the lists: ordered or not?
  - Not ordered!
    - Insert is fast
    - Can easily remove the most recently inserted elements

Insertion in Hash Tables

Alg.: CHAINED-HASH-INSERT(T, x)

```
        insert x at the head of list T[h(key[x])]
```

- Worst-case running time is $O(1)$
- Assumes that the element being inserted isn't already in the list
- It would take an additional search to check if it was already inserted

Deletion in Hash Tables

Alg.: CHAINED-HASH-DELETE(T, x)

```
        delete x from the list T[h(key[x])]
```

- Need to find the element to be deleted.
- Worst-case running time:
  - Deletion depends on searching the corresponding list

Searching in Hash Tables

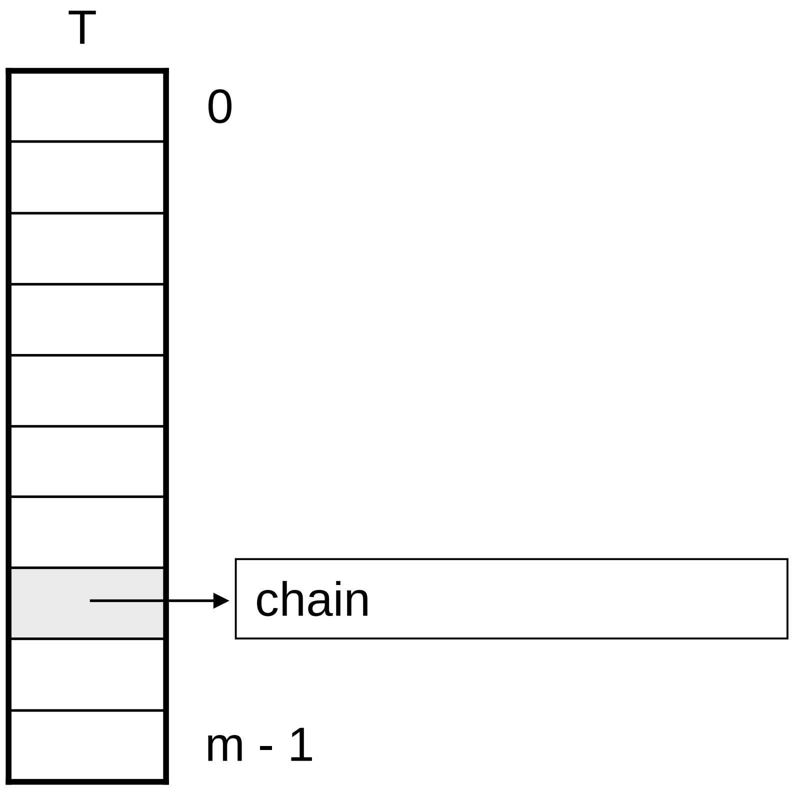
Alg.: CHAINED-HASH-SEARCH(T, k)

search for an element with key k in list $T[h(k)]$

- Running time is proportional to the length of the list of elements in slot $h(k)$

Analysis of Hashing with Chaining: Worst Case

- How long does it take to search for an element with a given key?
- Worst case:
  - All n keys hash to the same slot
  - Worst-case time to search is $\Theta(n)$, plus time to compute the hash function



Analysis of Hashing with Chaining: Average Case

- Average case
  - depends on how well the hash function distributes the n keys among the m slots
- **Simple uniform hashing** assumption:
  - Any given element is equally likely to hash into any of the m slots (i.e., probability of collision Pr(h(x)=h(y)), is 1/m)
- Length of a list:

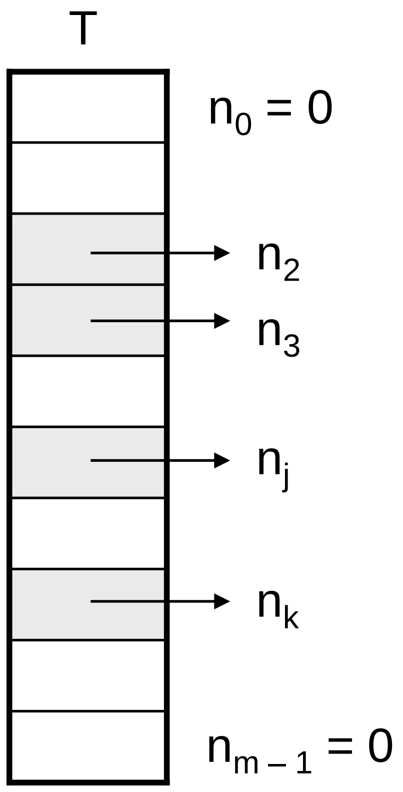
$$T[j] = n_j, \qquad j = 0, 1, \ldots, m-1$$

- Number of keys in the table:

$$n = n_0 + n_1 + \cdots + n_{m-1}$$

- Average value of n_j:

$$E[n_j] = \alpha = n/m$$

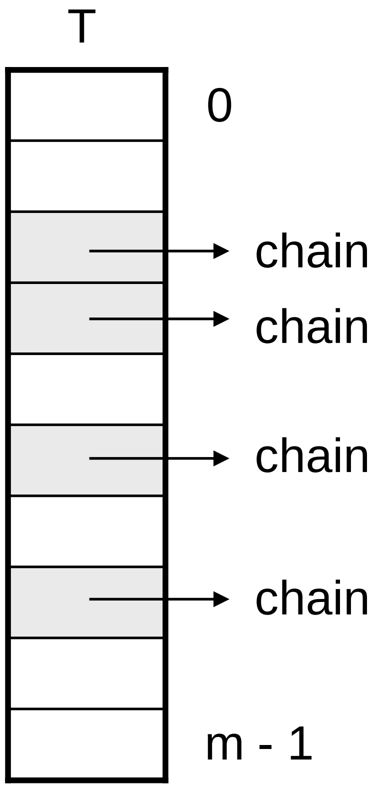


Load Factor of a Hash Table

- Load factor of a hash table T:

$$\alpha = n/m$$

  - n = # of elements stored in the table
  - m = # of slots in the table = # of linked lists
- α encodes the average number of elements stored in a chain
- α can be <, =, > 1



Case 1: Unsuccessful Search (i.e., item not stored in the table)

Theorem

An unsuccessful search in a hash table takes expected time $\Theta(1+\alpha)$ under the assumption of simple uniform hashing

(i.e., probability of collision Pr(h(x)=h(y)), is 1/m)

Proof

- Searching unsuccessfully for any key k
  - need to search to the end of the list T[h(k)]
- Expected length of the list:
  - $E[n_{h(k)}] = \alpha = n/m$
- Expected number of elements examined in an unsuccessful search is α
- Total time required is:
  - O(1) (for computing the hash function) + α → $\Theta(1+\alpha)$

Case 2: Successful Search

<u>Successful search:</u> $\Theta(1 + \frac{a}{2}) = \Theta(1 + a)$ time on the average

(search half of a list of length a plus $O(1)$ time to compute $h(k)$)

Analysis of Search in Hash Tables

- If m (# of slots) is proportional to n (# of elements in the table):
- $n = O(m)$
- $\alpha = n/m = O(m)/m = O(1)$

$\Rightarrow$ Searching takes constant time on average

Hash Functions

- A hash function transforms a key into a table address
- **What makes a good hash function?**

  (1) Easy to compute

  (2) Approximates a random function: for every input, every output is equally likely (simple uniform hashing)
- In practice, it is very hard to satisfy the simple uniform hashing property
  - i.e., we don't know in advance the probability distribution that keys are drawn from

Good Approaches for Hash Functions

- Minimize the chance that closely related keys hash to the same slot
  - Strings such as pt and pts should hash to different slots
- **Derive a hash value that is independent from any patterns that may exist in the distribution of the keys**

The Division Method

- **Idea:**
  - Map a key k into one of the m slots by taking the remainder of k divided by m

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

- **Advantage**:
  - fast, requires only one operation
- **Disadvantage**:
  - Certain values of m are bad, e.g.,
    - power of 2
    - non-prime numbers

Example - The Division Method

- If $m = 2^p$, then $h(k)$ is just the least significant p bits of k
  - $p = 1 \Rightarrow m = 2$

    $\Rightarrow h(k) = \{0, 1\}$, least significant 1 bit of k
  - $p = 2 \Rightarrow m = 4$

    $\Rightarrow h(k) = \{0, 1, 2, 3\}$, least significant 2 bits of k
- Choose m to be a prime, not close to a power of 2
  - Column 2: k mod 97
  - Column 3: k mod 100

	m 97	m 100
16838	57	38
5758	35	58
10113	25	13
17515	55	15
31051	11	51
5627	1	27
23010	21	10
7419	47	19
16212	13	12
4086	12	86
2749	33	49
12767	60	67
9084	63	84
12060	32	60
32225	21	25
17543	83	43
25089	63	89
21183	37	83
25137	14	37
25566	55	66
26966	0	66
4978	31	78
20495	28	95
10311	29	11
11367	18	67

The Multiplication Method

Idea:

- Multiply key k by a constant A, where $0 < A < 1$
- Extract the fractional part of kA
- Multiply the fractional part by m
- Take the floor of the result

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor = \lfloor m\,\underbrace{(k\,A \bmod 1)}_{\text{fractional part of } kA = kA - \lfloor kA \rfloor} \rfloor$$

- **Disadvantage:** Slower than division method
- **Advantage:** Value of m is not critical, e.g., typically 2^p

Example – Multiplication Method

- The value of m is not critical now (e.g., $m = 2^p$)

assume $m = 2^3$

```
     .101101 (A)
      110101 (k)
--------------
1001010.0110011 (kA)
```

discard: 1001010

shift .0110011 by 3 bits to the left

011.0011

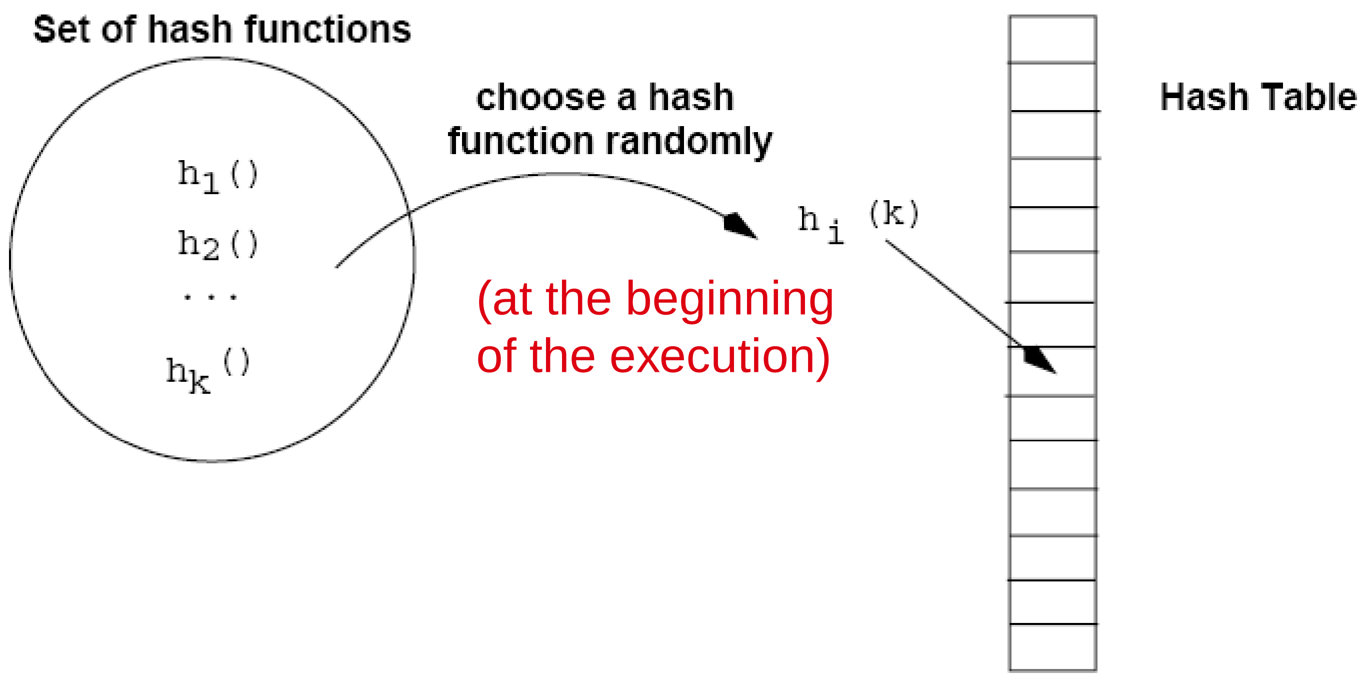
take integer part: 011

thus, h(110101)=011

Universal Hashing

- In practice, keys are not randomly distributed
- Any fixed hash function might yield $\Theta(n)$ time
- Goal: hash functions that produce random table indices irrespective of the keys
- Idea:
  - Select a hash function at random, from a designed class of functions at the beginning of the execution

Universal Hashing



(at the beginning of the execution)

Definition of Universal Hash Functions

H={h(k): U→(0,1,..,m-1)}

H is said to be universal if

$$\text{for } x \neq y,\ |(\mathbf{h()} \in \mathbf{H}\mathbf{:}\ \mathbf{h(x)=h(y)}|\mathbf{=}|\mathbf{H}|\mathbf{/m}$$

(notation: |H|: number of elements in H - cardinality of H)

How is this property useful?

- What is the probability of collision in this case ?

It is equal to the probability of choosing a function $h \in U$ such that $x \neq y$ --> $h(x) = h(y)$ which is

$$\Pr(h(x)=h(y)) = \frac{|H|/m}{|H|} = \frac{1}{m}$$

Universal Hashing – Main Result

With universal hashing the chance of collision between distinct keys k and l is no more than the $1/m$ chance of collision if locations $h(k)$ and $h(l)$ were randomly and independently chosen from the set $\{0, 1, \ldots, m - 1\}$

Designing a Universal Class of Hash Functions

- Choose a prime number p large enough so that every possible key k is in the range $[0 \ldots p - 1]$

  $Z_p = \{0, 1, \ldots, p - 1\}$ and $Z_p^* = \{1, \ldots, p - 1\}$
- Define the following hash function

  $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m,$

  $\forall\ a \in Z_p^*$ and $b \in Z_p$
- The family of all such hash functions is

  $\mathcal{H}_{p,m} = \{h_{a,b}: a \in Z_p^* \text{ and } b \in Z_p\}$

The class $\mathcal{H}_{p,m}$ of hash functions is universal

- a , b: chosen randomly at the beginning of execution

Example: Universal Hash Functions

E.g.: $p = 17, m = 6$

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$$

$$h_{3,4}(8) = ((3 \cdot 8 + 4) \bmod 17) \bmod 6$$

$$= (28 \bmod 17) \bmod 6$$

$$= 11 \bmod 6$$

$$= 5$$

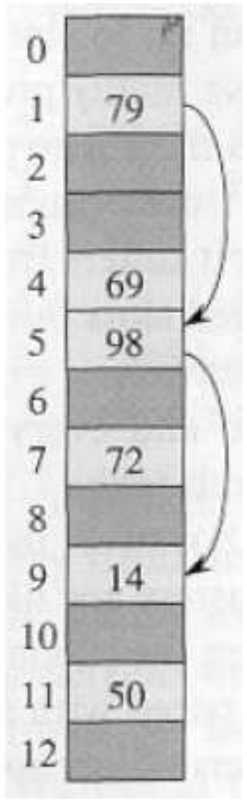
Advantages of Universal Hashing

- Universal hashing provides good results on average, independently of the keys to be stored
- Guarantees that no input will always elicit the worst-case behavior
- Poor performance occurs only when the random choice returns an inefficient hash function – this has small probability

Open Addressing

- If we have enough contiguous memory to store all the keys ($m > N$) $\Rightarrow$ store the keys in the table itself
- No need to use linked lists anymore
- Basic idea:
  - <u>Insertion:</u> if a slot is full, try another one, until you find an empty one
  - <u>Search:</u> follow the same sequence of probes
  - <u>Deletion:</u> more difficult ... (we'll see why)
- Search time depends on the length of the probe sequence!

e.g., insert 14



Generalize hash function notation:

- A hash function contains two arguments now:

  (i) Key value, and (ii) Probe number

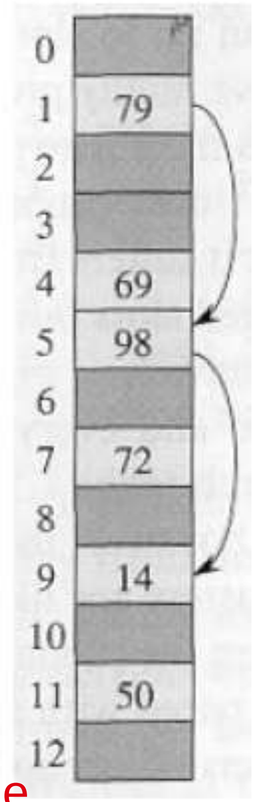
  $h(k,p), \quad p=0,1,\ldots,m-1$

- Probe sequences

  $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$

  – Must be a permutation of $\langle 0,1,\ldots,m-1\rangle$
  – There are $m!$ possible permutations
  – Good hash functions should be able to produce all $m!$ probe sequences

insert 14



Example

<1, 5, 9>

Common Open Addressing Methods

- Linear probing
- Quadratic probing
- Double hashing

- Note: None of these methods can generate more than m^2 different probing sequences!

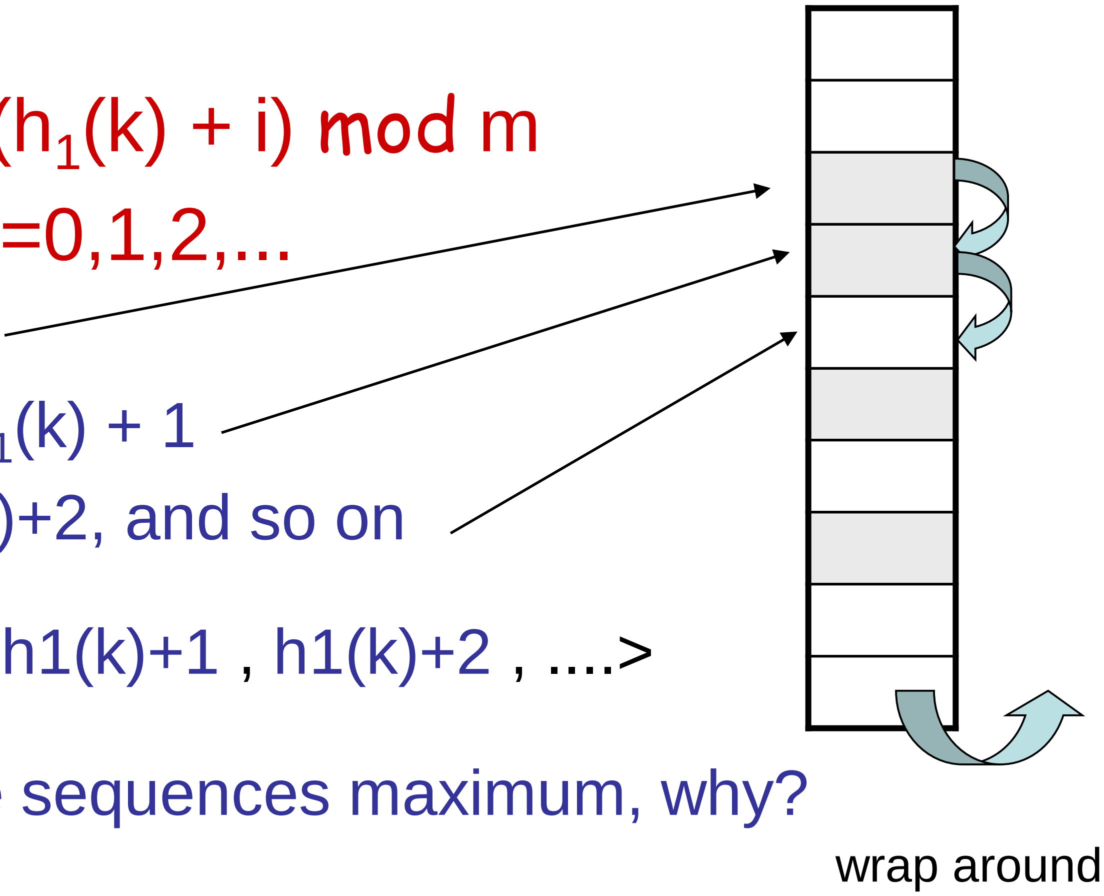
Linear probing: Inserting a key

- Idea: when there is a collision, check the next available position in the table (i.e., probing)

$$h(k,i) = (h_1(k) + i) \bmod m$$

$$i=0,1,2,\ldots$$

- First slot probed: $h_1(k)$
- Second slot probed: $h_1(k) + 1$
- Third slot probed: $h_1(k)+2$, and so on

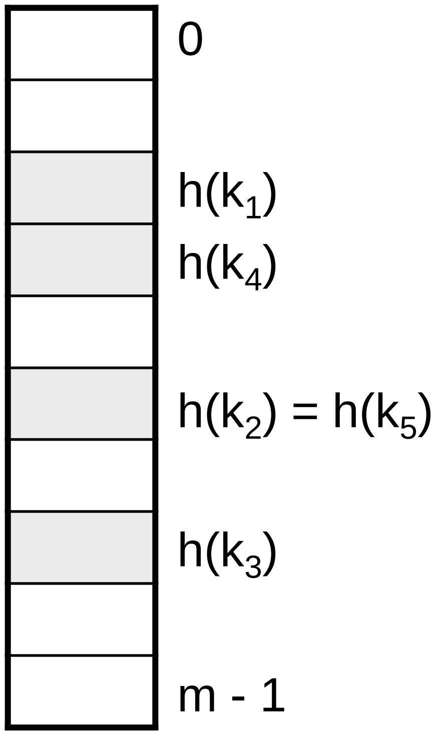


probe sequence: < h1(k), h1(k)+1 , h1(k)+2 , ...>

- Can generate m probe sequences maximum, why?

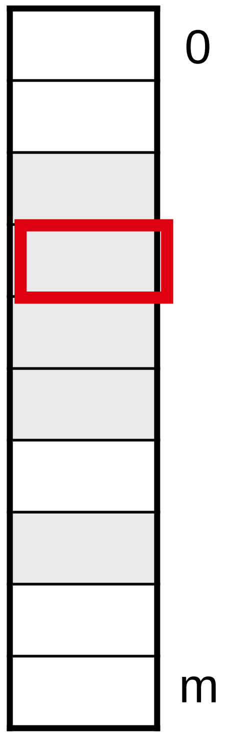
Linear probing: Searching for a key

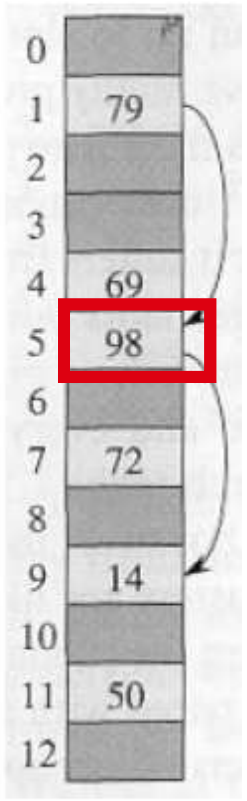
- Three cases:
  - (1) Position in table is occupied with an element of equal key
  - (2) Position in table is empty
  - (3) Position in table occupied with a different element
- Case 2: probe the next higher index until the element is found or an empty position is found
- The process wraps around to the beginning of the table



Linear probing: Deleting a key

- Problems
  - Cannot mark the slot as empty
  - Impossible to retrieve keys inserted after that slot was occupied
- Solution
  - Mark the slot with a sentinel value DELETED
- The deleted slot can later be used for insertion
- Searching will be able to find all the keys



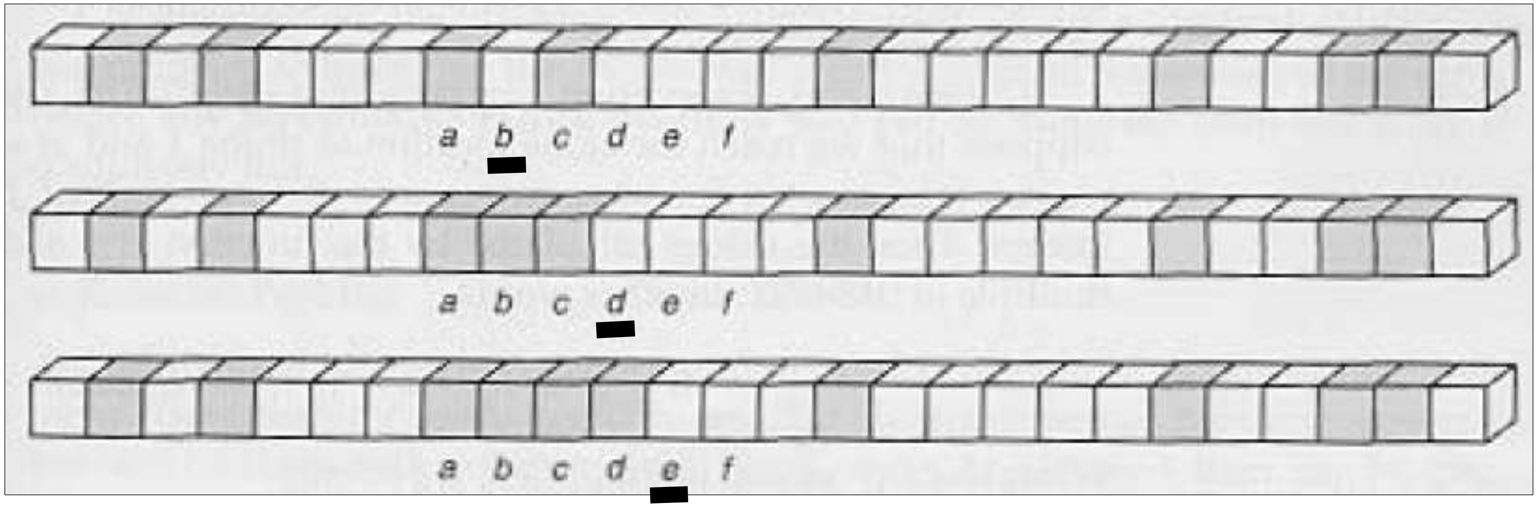


Primary Clustering Problem

- Some slots become more likely than others
- Long chunks of occupied slots are created

$\Rightarrow$ search time increases!!

initially, all slots have probability 1/m



Slot b: 2/m

Slot d: 4/m

Slot e: 5/m

Quadratic probing

$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, where $h' : U --> (0, 1, \ldots, m-1)$

i=0,1,2,...

- Clustering problem is less serious but still an issue (*secondary clustering*)

- How many probe sequences quadratic probing generate ? m

(the initial probe position determines the probe sequence)

Double Hashing

(1) Use one hash function to determine the first slot

(2) Use a second hash function to determine the increment for the probe sequence

$$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m, \quad i=0,1,\ldots$$

- Initial probe: $h_1(k)$
- Second probe is offset by $h_2(k) \bmod m$, so on ...
- Advantage: avoids clustering
- Disadvantage: harder to delete an element
- Can generate m^2 probe sequences maximum

Double Hashing: Example

$h_1(k) = k \bmod 13$

$h_2(k) = 1 + (k \bmod 11)$

$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod 13$

- Insert key 14:

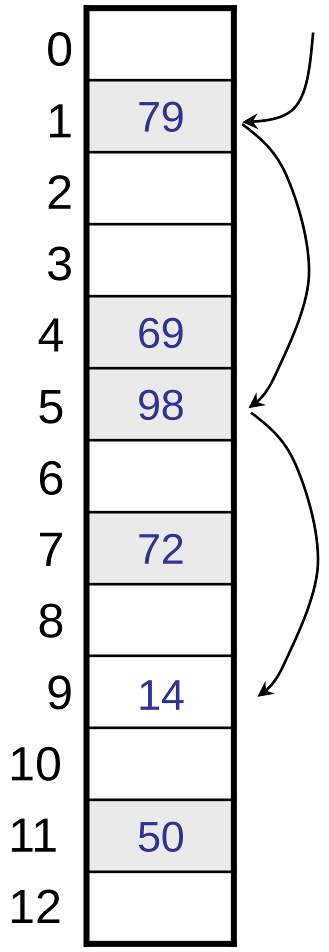
$h_1(14,0) = 14 \bmod 13 = 1$

$h(14,1) = (h_1(14) + h_2(14)) \bmod 13$

$= (1 + 4) \bmod 13 = 5$

$h(14,2) = (h_1(14) + 2\, h_2(14)) \bmod 13$

$= (1 + 8) \bmod 13 = 9$



Analysis of Open Addressing

- Ignore the problem of clustering and assume that all probe sequences are equally likely

Unsuccessful retrieval:

Prob(*probe hits an occupied cell*)= a (load factor)

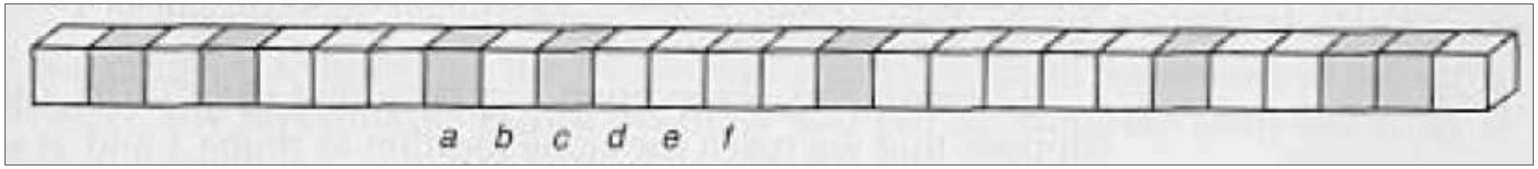
Prob(*probe hits an empty cell*)= $1-a$

probability that a probe terminates in 2 steps: $a(1-a)$

probability that a probe terminates in k steps: $a^{k-1}(1-a)$

What is the average number of steps in a probe ?

$$E(\#steps) = \sum_{k=1}^{m} ka^{k-1}(1-a) \leq \sum_{k=0}^{\infty} ka^{k-1}(1-a) = (1-a)\,\frac{1}{(1-a)^2} = \frac{1}{1-a}$$



Analysis of Open Addressing (cont'd)

Successful retrieval:

$$E(\#steps) = \frac{1}{a}\,ln(\frac{1}{1-a})$$

Example (similar to **Exercise 11.4-4, page 244**)

Unsuccessful retrieval:

a=0.5	E(#steps) = 2
a=0.9	E(#steps) = 10

Successful retrieval:

a=0.5	E(#steps) = 3.387
a=0.9	E(#steps) = 3.670