Python Developer

Placement support

Poonam Adhikari: Python Developer (GUVI)

Python Variables

Variable is a name that is used to refer to memory location. Python variable is also known as an identifier and used to hold value.

a=50

b = a

Ways of assignment

x=y=z=50

a,b,c=5,10,15

1. print(x)
2. print(y)
3. print(z)

Example

a = 5 → print("The type of a", type(a))

b = 40.5 → print("The type of b", type(b))

c = 1+3j → print("The type of c", type(c))

Check variable belongs to specified class

```
print(" c is a complex number", isinstance(1+3j,complex))
```

More examples

str = "string using double quotes" → **print**(str)

str1 = 'hello Students#string str1

str2 = ' how are you' #string str2

```
print (str1[0:2]) #printing first two character using slice operator
print (str1[4]) #printing 4th character of the string
print (str1*2) #printing the string twice
print (str1 + str2) #printing the concatenation of str1 and str2
```

Data types

- **Integers (default for numbers)**

  z = 5 / 2 # Answer 2, integer division

- **Floats**

  x = 3.456

- **Strings**
  - Can use “” or ‘’ to specify with “abc” == ‘abc’
  - Unmatched can occur within the string: “matt’s”
- **Boolean**

  True or False

Exercise Problem

What will be the output of the following program?

```
In [1]: first_name = "poonam"

In [2]: last_name = "Adhikari"

In [3]: print(first_name+ " " + last_name)
```

Range

Get list of numbers in the given range

Syntax : range(start_index, end_index, step_size)

Example: range(1,10,1)

```
range(1,10) # if step size is not given by default step size is 1

range(10) # if start index is not given by default index is 0
```

End_index is not considered

Data Types

x = "Hello World" **str**

x = 20 **int**

x = 20.5 **float**

x = 1j **complex**

x = True **bool**

x = None **NoneType**

x = ["apple", "banana", "cherry"] **List**

x = ("apple", "banana", "cherry") **tuple**

x = range(6) **range**

x = {"name" : "John", "age" : 36} **dict**

x = {"apple", "banana", "cherry"} **set**

Setting different data types

x = str("Hello World") **str**

x = int(20) **int**

x = float(20.5) **float**

x = complex(1j) **complex**

x = bool(5) **bool**

x = list(("apple", "banana", "cherry")) **list**

x = tuple(("apple", "banana", "cherry")) **tuple**

x = range(6) **range**

x = dict(name="John", age=36) **dict**

x = set(("apple", "banana", "cherry")) **set**

Re-declaration of variables

```
# Declare a variable and initialize
f = 0
print(f)
# re-declaring the variable works
f = 'guru99'
print(f)
```

```
Python5.1
"C:\Users\DK
 5/PythonC
0
guru99
```

you can re-declare the variables, even-after if it is declared once. it works fine

Deleting a variable

Variable = “Hello, world!”

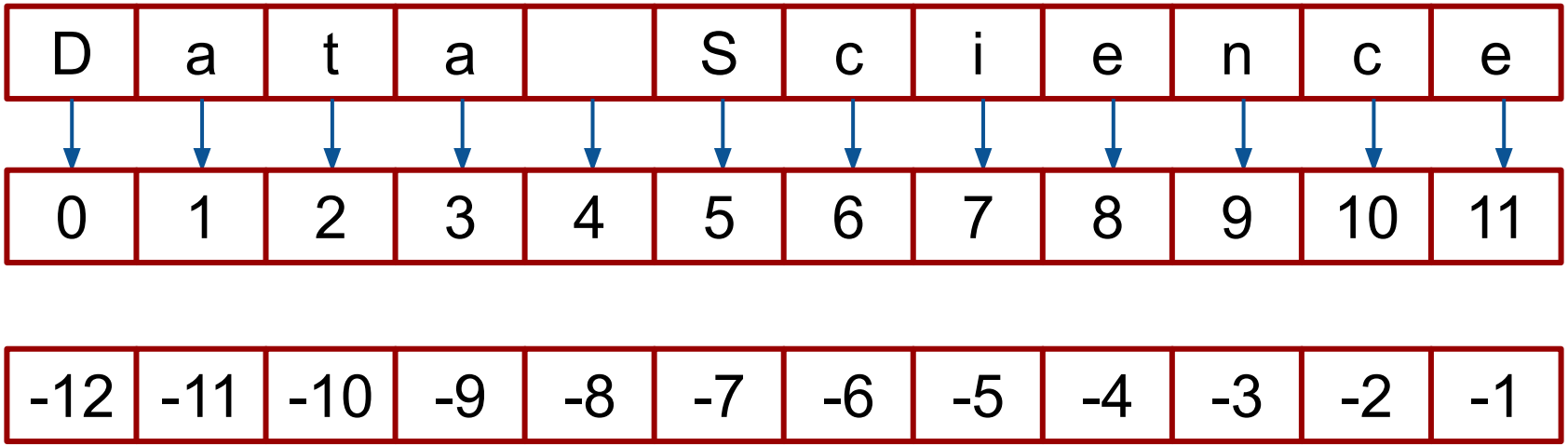
del variable

print(variable)

-> variable is deleted from memory

List

My_str = “Data Science”



List Properties

Ordered: Maintain the order of the data insertion

Changeable: List is mutable and we can modify items

Heterogeneous: List can contain data of different types

Contains duplicate: Allows duplicates data

Lists are mutable

Functions in List

append()	Adds the element at the end of the list.
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified values
extend()	Add the elements of a list(or any iterables), to the end of the current list
index()	Returns the index of the first element with the specific value
pop()	Removes the element at the specific value
remove()	Removes the item with the specified value
sort()	Sorts the list

List

List are comma separated elements in bracket

A = [1,2,3,4]

Elements can be of different data types

A = [1, ‘a’, ‘hello’ ,4.2]

List element can be another list

A = [1, ‘a’, [‘hello’, ‘world’],4.2]

Accessing elements

```
c = [-4, -3, -2, -1]
```

list elements can be accessed by index

```
print(c[0])

print(c[-1])

print(c[-1])
```

Try these

```
print(c[1:])

print(c[-2:])

print(c[:2])
```

Tuples

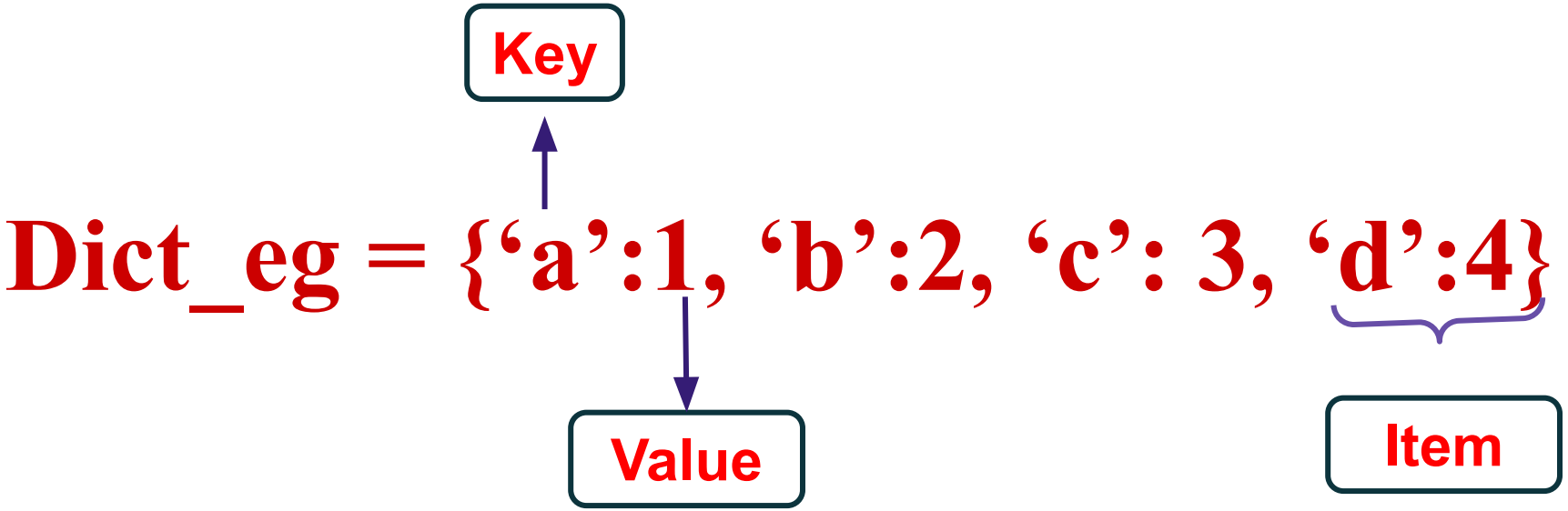
Tuples are like lists

```
a = (1, 2, 3)

b = (a, 'world')

print(a[2])
```

But they are immutable: try

```
a[1] = 0
```

Dictionary



Properties of Dict

Unordered: The items in dict are stored without any index value

Unique: Keys in dictionaries should be Unique

Mutable: We can add/modify/remove key-value after the creation

Functions in dictionaries

clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
get()	Returns the value associated with the given key
from keys()	Returns a dictionary with the specified keys and value
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
update()	Updates the dictionary with the specified key-value pairs

Dictionary

Like a bilingual dictionary, it allows you to associate a key to a value.

weight = {'Amar': 68, 'Akbar': 73, 'Anthony': 55} print(weight)

dict maps keys to values

dict can be used for unordered structured data

Set

Set_eg = {‘ds’, 10, 20.10}

Set properties

Unordered: Set doesn’t maintain the order of the data insertion.

Unchangeable: Set are immutable and we can’t modify items

Heterogeneous: Set can contains data of all types

Unique: Set don’t allows duplicates items

Mutable

Mutable: Objects whose value can change are said to be mutable E.g., list, dictionary, set and user-defined classes

list_values = [1, 2, 3] set_values = (1, 2, 3) print(id(list_values)) print(id(set_values))	list_values += [4, 5, 6] set_values += (4, 5, 6) print(id(list_values)) print(id(set_values))

Immutable

Immutable: whose value is unchangeable once they are created are called immutable int, float, decimal, bool, string

```
number = 42
print(id(number))


number += 1
print(id(number))
```

Operators in Python

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Conditions

- **Equals:** a == b
- **Not Equals:** a != b
- **Less than:** a < b
- **Less than or equal to:** a <= b
- **Greater than:** a > b
- **Greater than or equal to:** a >= b

Conditions

If statement	elif statement	else statement

a = 33
b = 200
if b > a:
  print("b is greater than a")
	a = 33
	b = 33
	if b > a:
	  print("b is greater than a")
	elif a == b:
	  print("a and b are equal")
		a = 200
		b = 33
		if b > a:
		  print("b is greater than a")
		elif a == b:
		  print("a and b are equal")
		else:
		  print("a is greater than b")

If with and

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")
```

If with or

```
a = 200
b = 33
c = 500
if a > b or c< a:
  print("Atleast one condition is True")
```

If with not

```
a = 200
b = 33
c = 500
If not(a>b):
  print("condition is False")
else:
  print("condition is True")
```

For loop

Why you need loop?

If you need to repeat the same, or similar, code a number of times

Options or sequence

Semicolon: **denotes separation**

Syntax: ***for value in range:***

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

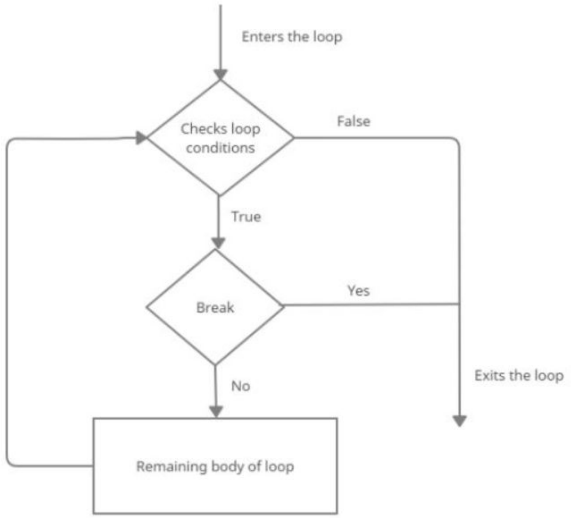
```
for x in "banana":
  print(x)
```

Break

- **Loop control statement**
- Terminate a loop and skip to the next code after the loop; break will help you do that

```
fruits = ['apple','grapes','banana']
for x in fruits:
  print(x)
  break
```

```
fruits = ['apple','grapes','banana']
for x in fruits:
  print(x)
  if x =='grapes':
    break
```



Break / Continue

```
for index in range(0,10):
    if index%2 == 0:
        continue
    else:
        print("Given index is: ",str(index))
```

```
for index in range(1,10):
    if index%3 == 0:
        break
    else:
        print("Given index is: ",str(index))
```

Functions

Introduction:

- Runs when it is called.
- Can pass data, known as parameters
- Can return data as a result

Syntax:

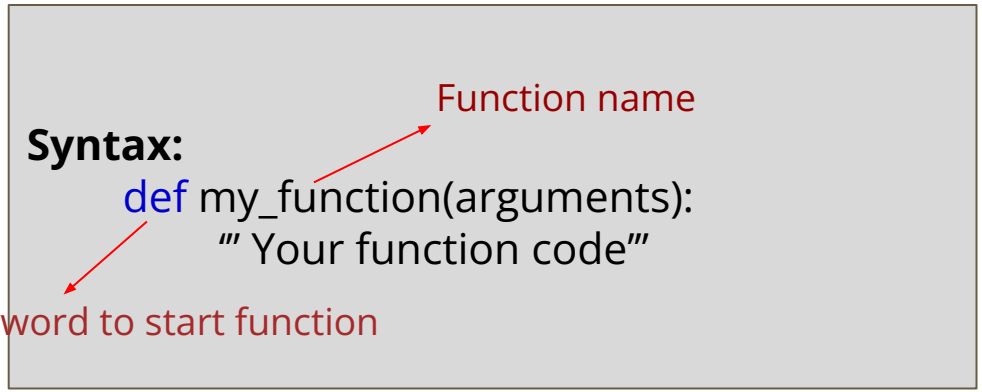
```
def my_function(arguments):
    ''' Your function code'''
```

Keyword to start function

Functions

Introduction:

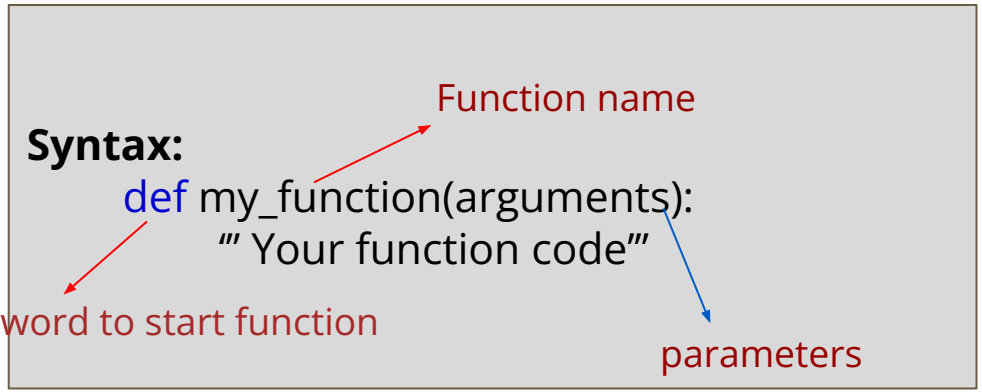
- Runs when it is called.
- Can pass data, known as parameters
- Can return data as a result



Functions

Introduction:

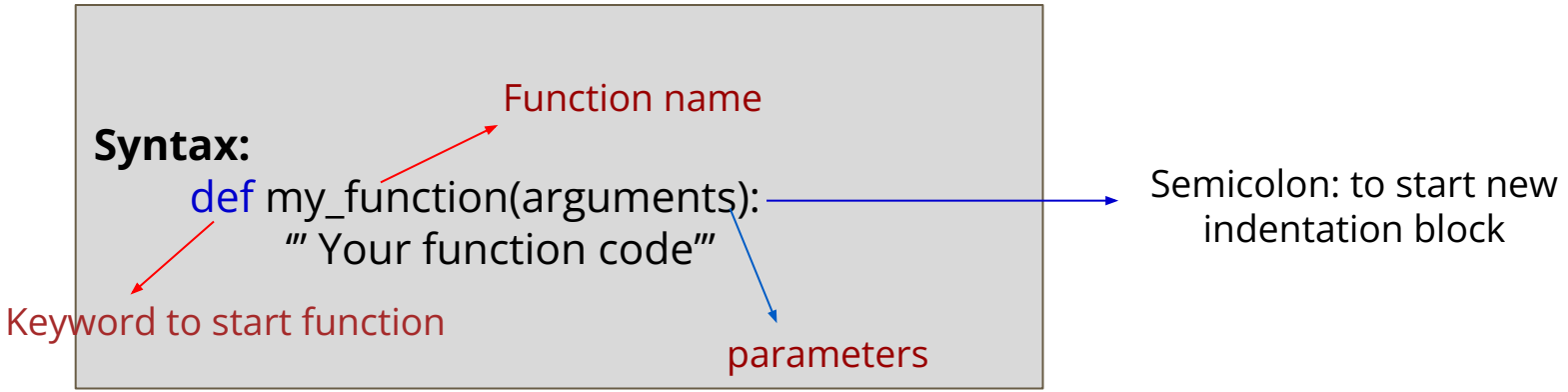
- Runs when it is called.
- Can pass data, known as parameters
- Can return data as a result



Functions

Introduction:

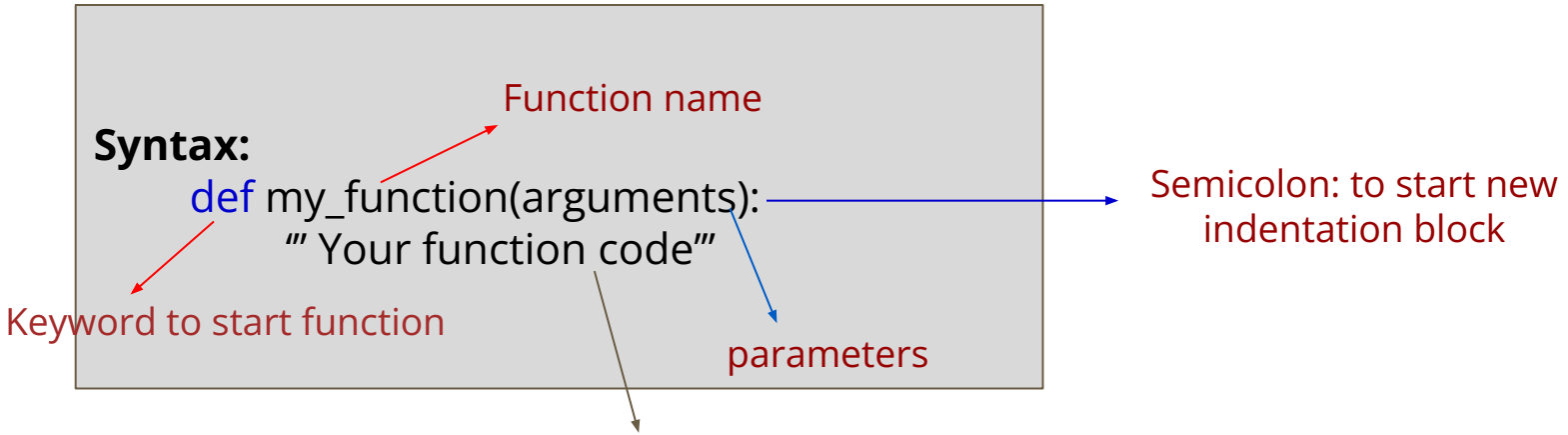
- Runs when it is called.
- Can pass data, known as parameters
- Can return data as a result



Functions

Introduction:

- Runs when it is called.
- Can pass data, known as parameters
- Can return data as a result



Why we need functions

- Functions allow the same piece of code to run multiple times
- Abstractions
- Reusability
- Functions are used to divide the program into simple modules

Writing a function

```
def demo_function():
    print("Welcome")
demo_function()
```

Calling a function

Passing arguments

```
def demo_function(name):
    print("Hi: "+name)
    print("Welcome")
demo_function("Poonam")
```

Returning value

```
def demo_function(first_name, last_name):
    print("Welcome"+first_name+" "+last_name)
    return first_name+" "+last_name
demo_function("Poonam","Adhikari)
```

IMP: You have to pass specified number of arguments
Argument and parameters are same thing

Take any number of arguments

- Use * before the parameter name
- Will receive tuple of arguments

Syntax:

```
def student_names(*arg):
    for parameter in arg:
        print(parameter)
student_names("lalit","ankit",'Krish")
```

Keywords arguments

- Send arguments with the *key = value*
- Order of the arguments does not matter.

Syntax:

```
def student_names(st1, st2, st3):
    print("Third student is " +st3)

student_names(st1 = "lalit", st2= "ankit", st3= "krish")
```

Default arguments and arbitrary arguments

Default arguments

Function without argument, it uses the default value

Syntax:

```
def my_function(city="Chandigarh"):
    print("Your city is: "+city)
my_function("Ropar")
my_function()
```

Default value for city is Chandigarh

Arbitrary arguments

Not sure about number of keywords arguments?

Syntax:

```
def my_function(**args):
    print("Your city is: ",args['name'],args['state'],args['country'])

my_function(name="Ropar",state="Punjab",country="India")
```

List Comprehensions

- List in single line
- Offers a shorter syntax
- The return value is a new list

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
  if "a" == x[0]:
    newlist.append(x)

print(newlist)
```

The *condition* is like a filter that only accepts the items that evaluate to `True`

Every list comprehension in Python includes three elements:

1. **`expression`** is the member itself, a call to a method, or any other valid expression that returns a value. In the example above, the expression is `if "a" in a[0].`
2. **`member`** is the object or value in the list or iterable. In the example above, the member value is **`x`.**
3. **`iterable`** is a list, set, sequence, generator, or any other object that can return its elements one at a time. In the example above, the iterable is `fruits`.

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x[0]]

print(newlist)
```

While Loop

With the while loop we can execute a set of statements as long as a condition is true.

```
In [6]: index = 1
   ...: while index < 4:
   ...:     print(index)
   ...:     index += 1
   ...:
1
2
3
```

Print index as long as index is less than 4:

```
In [5]: index = 1
   ...: while index < 4:
   ...:     if index == 3:
   ...:         break
   ...:     print(index)
   ...:     index += 1
   ...:
1
2
```

Break in while loop

```
In [4]: index = 0
   ...: while index < 4:
   ...:     index += 1
   ...:     if index == 2:
   ...:         continue
   ...:     print(index)
   ...:
1
3
4
```

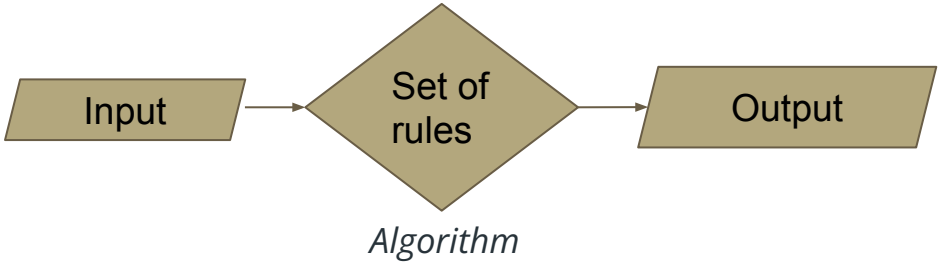
Continue in while loop

Continue

- End the current iteration
- Continues to the next iteration

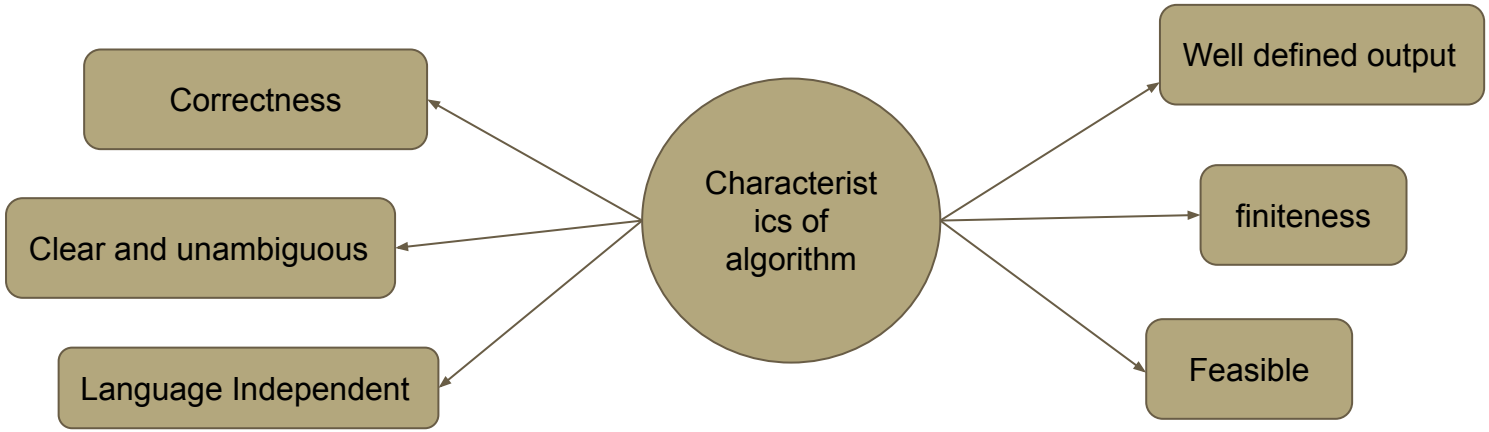
What is algorithm

Set of **finite** steps to **accomplish** a task



Algorithm

Characteristics of Algorithm



Example of Algorithm

Write an algorithm to prepare tea:

1. Turn on stove
2. Hot some water in pan
3. Add some tea leaves
4. Add sugar
5. Add milk
6. Wait till it boil

Program and Algorithm

Algorithm	Program
Required at **design phase**	Required at **implementation phase**
Written in **natural language**	Written in **any language**
No need to follow proper syntax	Need to follow proper syntax of a language
We **analyse** algorithm	We **test** program
Person should have domain knowledge	Programmer

How to write an algorithm

E.g Sum of two numbers

Step 1: start

Step 2: Read/input/enter two numbers (a,b)

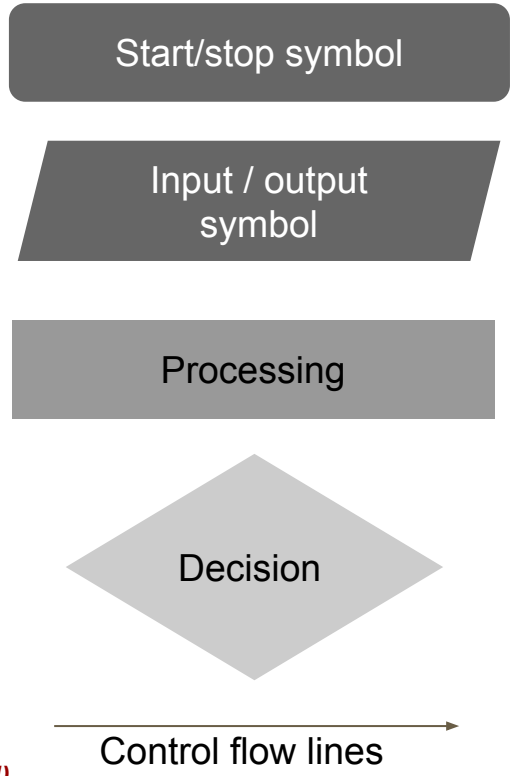
Step 3: sum = a+b

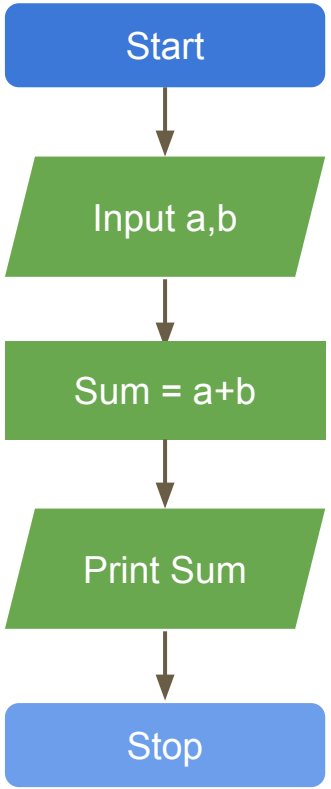
Step 4: print sum

Step 5: stop

Representation of Algorithm

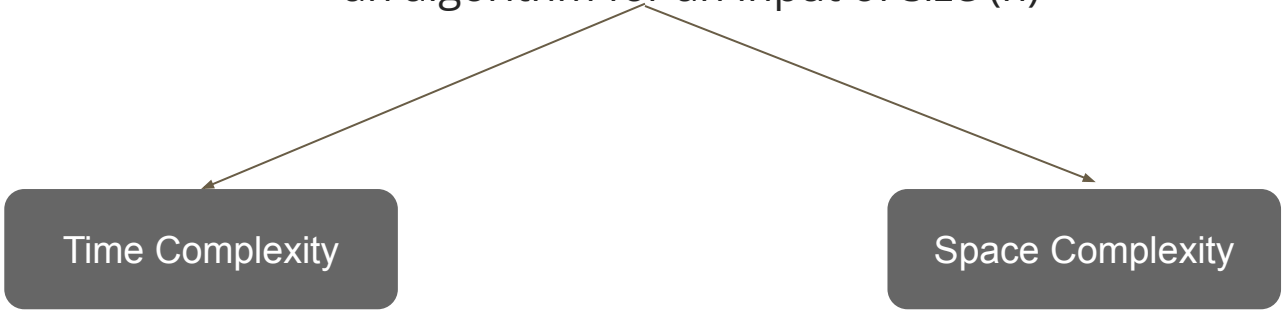
1. Pseudo code
2. Flow chart





Analysis of Algorithm

The complexity of an algorithm computes the **amount of time** and **spaces** required by an algorithm for an input of size (n)



Time required for the execution of that algorithm.

Memory space required for the execution of that algorithm.

Time Complexity

Worst case : the maximum run time

Average case : Average run time

Best case : the minimum run time

Given two algorithms how do we find out which perform better?

Record execution time

Asymptotic Analysis : the best way available for analyzing algorithms

We analyse performance of an algorithm in terms of input size

Big Oh notation

The worst-case running time of a program

Defines an upper bound of an algorithm

1. Ignore constants
   a. 5n = O(n)
2. Ignore low-order terms
   a. (5+15*20) ----------> Constant time (independent of input size N)
   b. For n in range(0,n): print(n) ----------> n*O(1) = O(n)
   c. For n in range(0,n):
      i. For i in range(0,i):
         1. print(n*i)

(n*n) = n^2

$O(1) < O(\log(n)) < O(n) < O(n\log(n)) < O(n^2) < O(2^n) < O(n!)$

$$\log_b(MN) = \log_b(M) + \log_b(N)$$

$$\log_b(N/M) = \log_b(M) - \log_b(N)$$

$$\log_b(M^p) = p \cdot \log_b(M)$$

—log(1) = 0

— log(4) = 2

— log(8) = 3.

Addition of two numbers

x=int(input()) → O(1)

y=int(input()) → O(1)

z=x+y → O(1)

print(z) → O(1)

Sum of n numbers

```
sum=0
n=int(input())
for i in range(n):
    sum=sum+i
print(sum)
```

O(1)

O(1)

O(n)

```
i=0
while(i<10):
    print(i)
    i=i*2
```

O(logn)

```
j=1
while(j<10):
    i=1
     while(i<10):
          print(i)
          i=i*2
    j=j+1
```

O(nlogn)

```
For i in range(n):
    For j in range(n):
        if(a[i]==a[j]):
            print("duplicate found")
```

O(n)

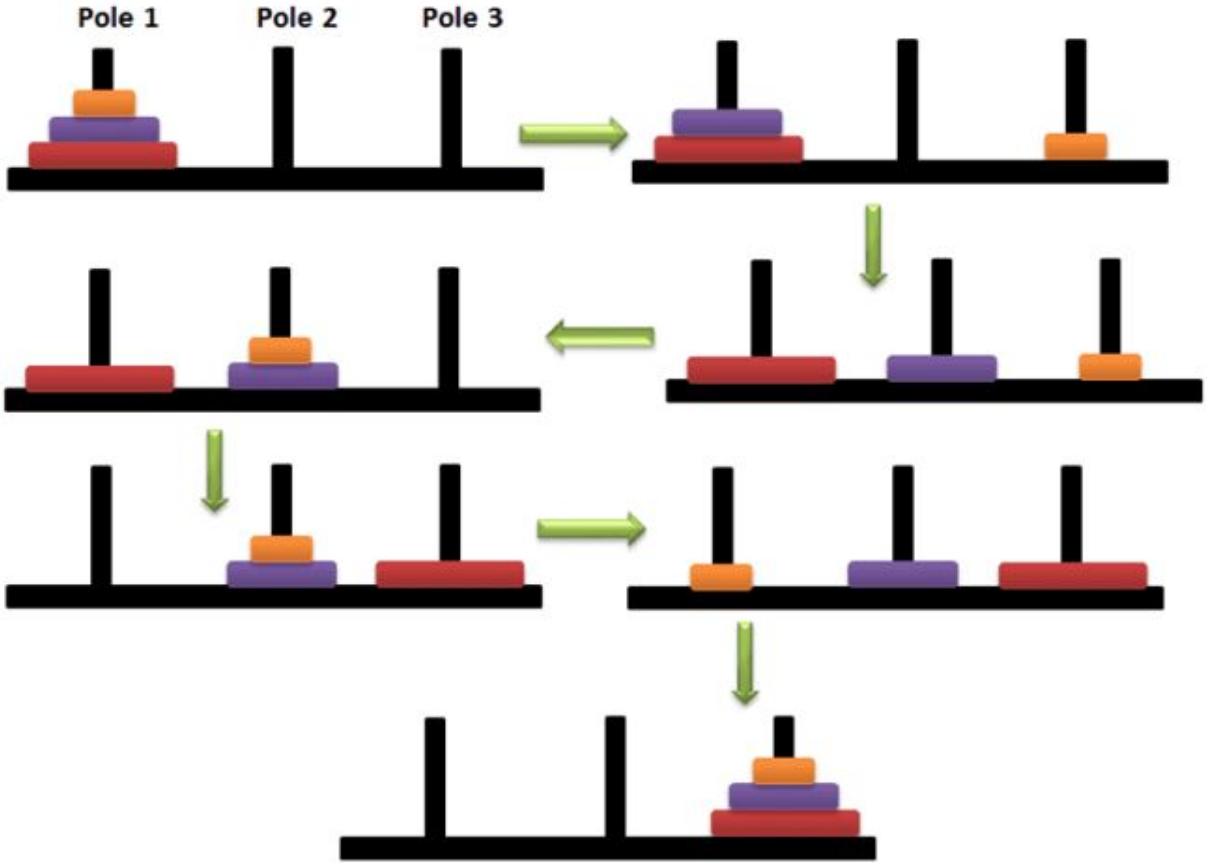
O(n)

i=1 j loop gets executed O(n)
i=2 O(n)
i=3 O(n)
|
|
|
i=n O(n)
Final Answer : O(n^2)

BETTER

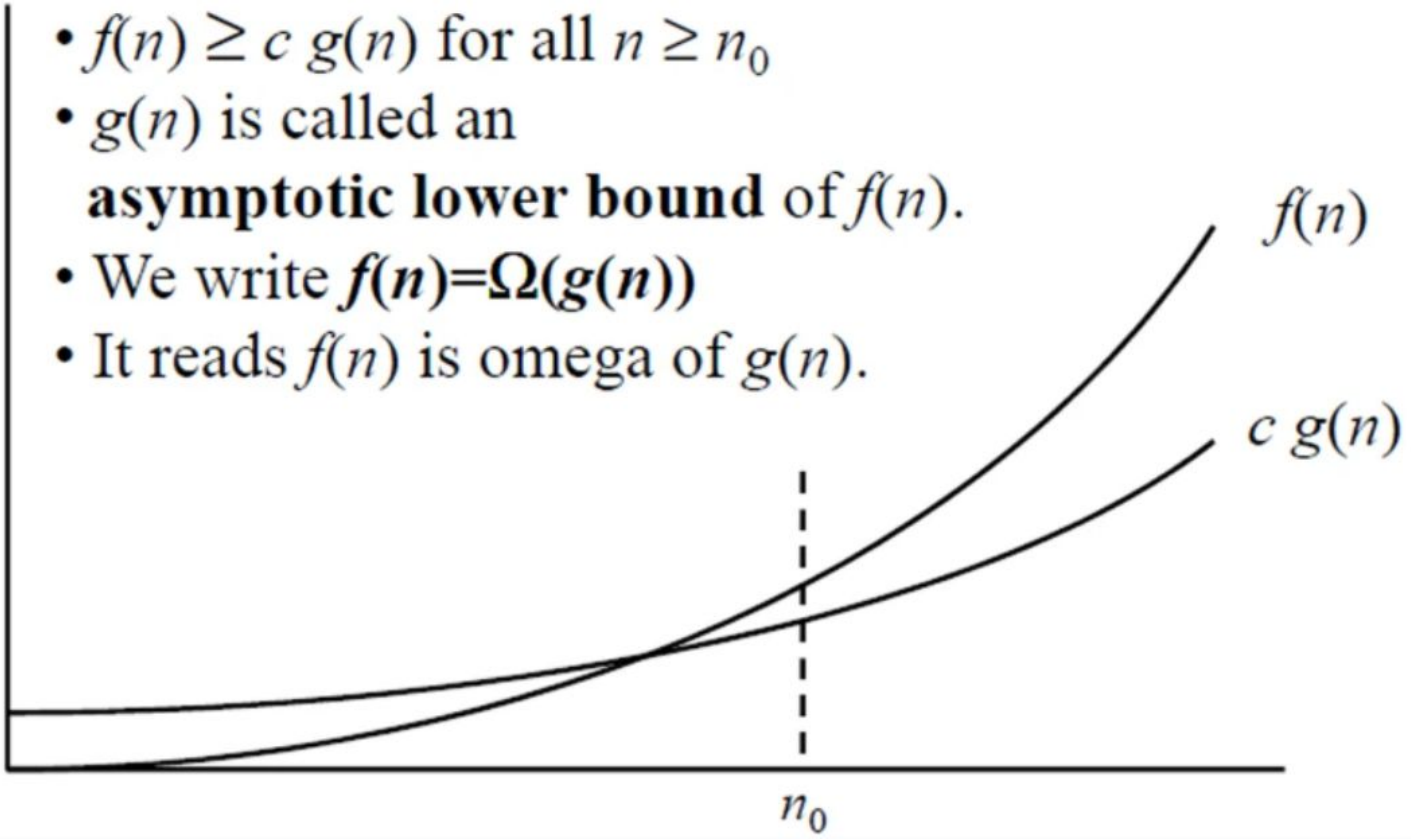
WORSE

- $O(1)$ constant time
- $O(\log n)$ log time
- $O(n)$ linear time
- $O(n \log n)$ log linear time
- $O(n^2)$ quadratic time
- $O(n^3)$ cubic time
- $O(n^k)$ polynomial time
- $O(2^n)$ exponential time


Pole 1
Pole 2
Pole 3

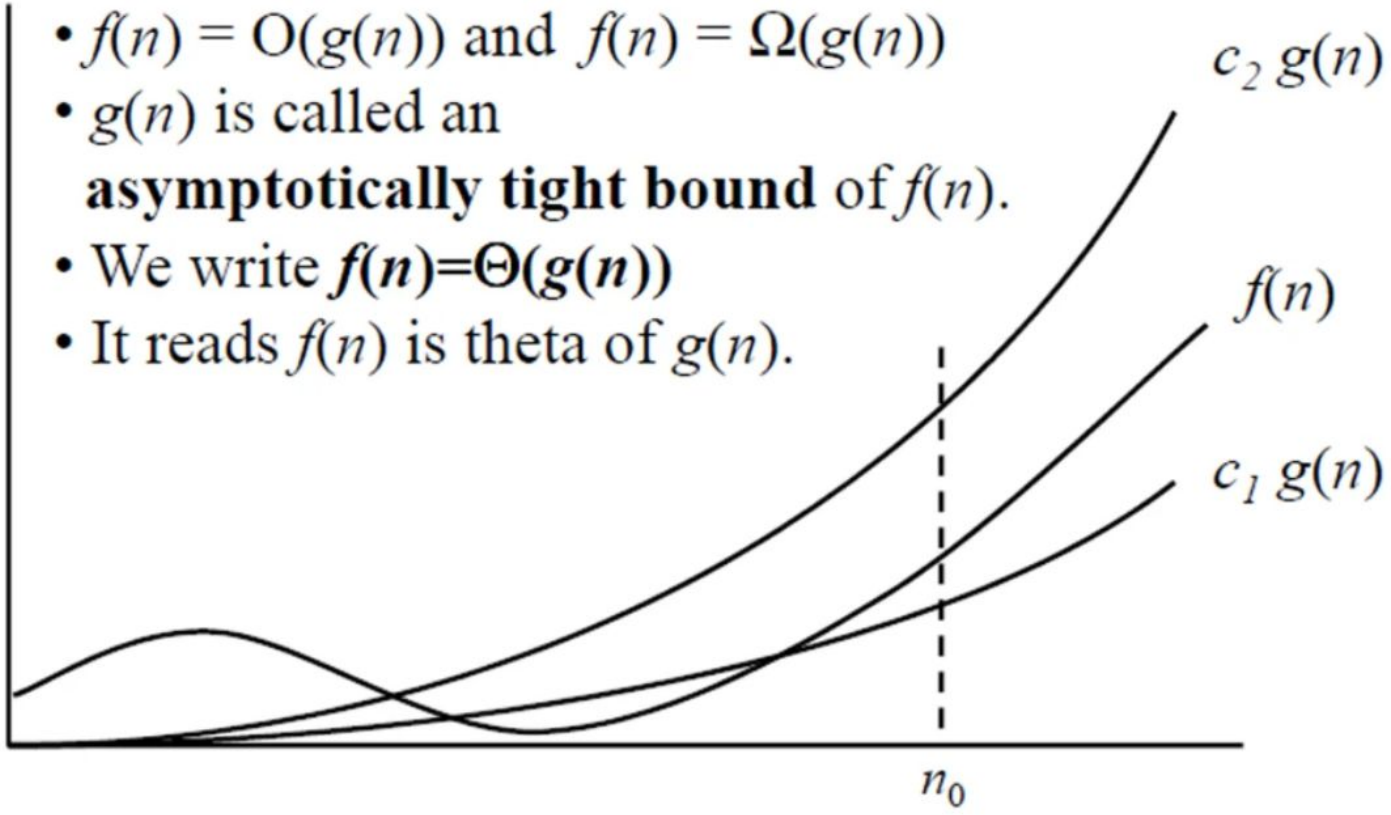
Asymptotic Lower Bound

- $f(n) \geq c\ g(n)$ for all $n \geq n_0$
- $g(n)$ is called an **asymptotic lower bound** of $f(n)$.
- We write $\boldsymbol{f(n)=\Omega(g(n))}$
- It reads $f(n)$ is omega of $g(n)$.

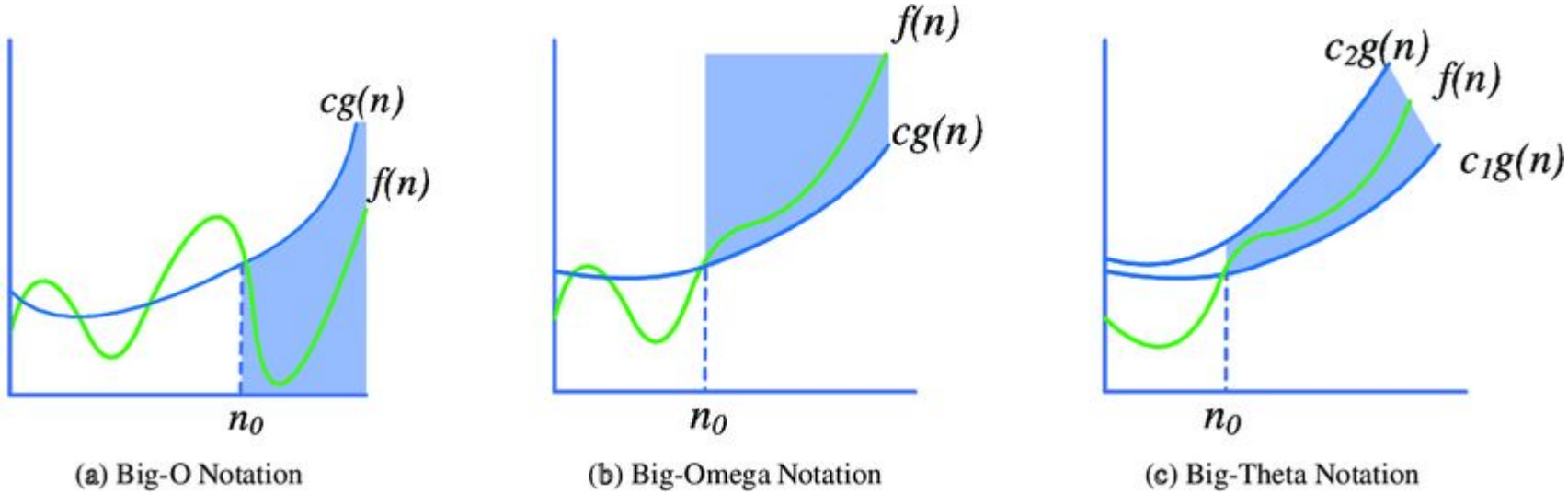


Asymptotically Tight Bound

- $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- $g(n)$ is called an **asymptotically tight bound** of $f(n)$.
- We write $\boldsymbol{f(n)=\Theta(g(n))}$
- It reads $f(n)$ is theta of $g(n)$.



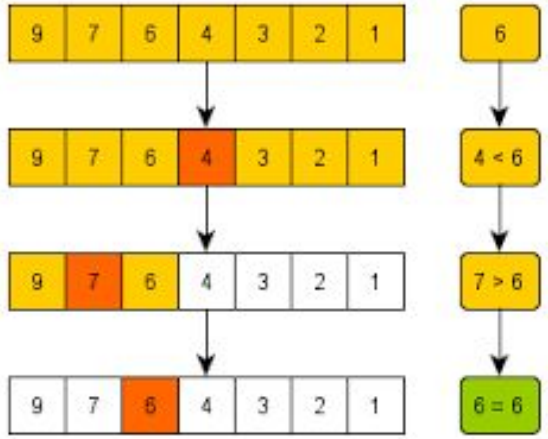
Big-O, Big-Omega, Big-Theta



(a) Big-O Notation (b) Big-Omega Notation (c) Big-Theta Notation

Binary Search

- Begin with the **mid element** of the whole array as a search key.
- If the value of the **search key** is **equal** to the item then **return an index** of the search key.
- If the value of the search key is **less** than the item in the middle of the interval
- Otherwise, narrow it to the upper half.
- Repeatedly check from the second point until the value is found or the interval is empty



Python Program

```
def binary_search(arr, x):
    low = 0
    high = len(arr) - 1
    mid = 0

    while low <= high:

        mid = (high + low) // 2

        if arr[mid] < x:
            low = mid + 1
        elif arr[mid] > x:
            high = mid - 1

        else:
            return mid

    return -1
```

Complexity of Binary Search

Best Case : when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is **O(1).**

Average Case Complexity - The average case time complexity of Binary search is **O(logn).**

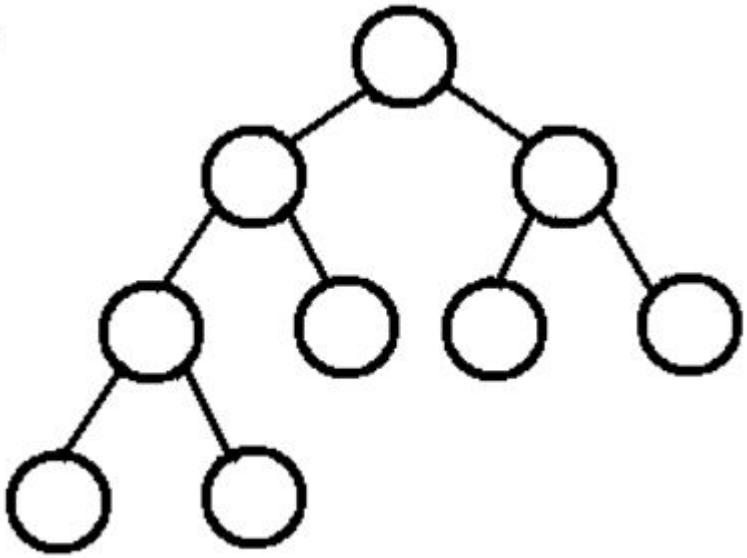
Worst Case Complexity - In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is **O(logn).**

Case	Time Complexity
Best Case	O(1)
Average Case	O(logn)
Worst Case	O(logn)

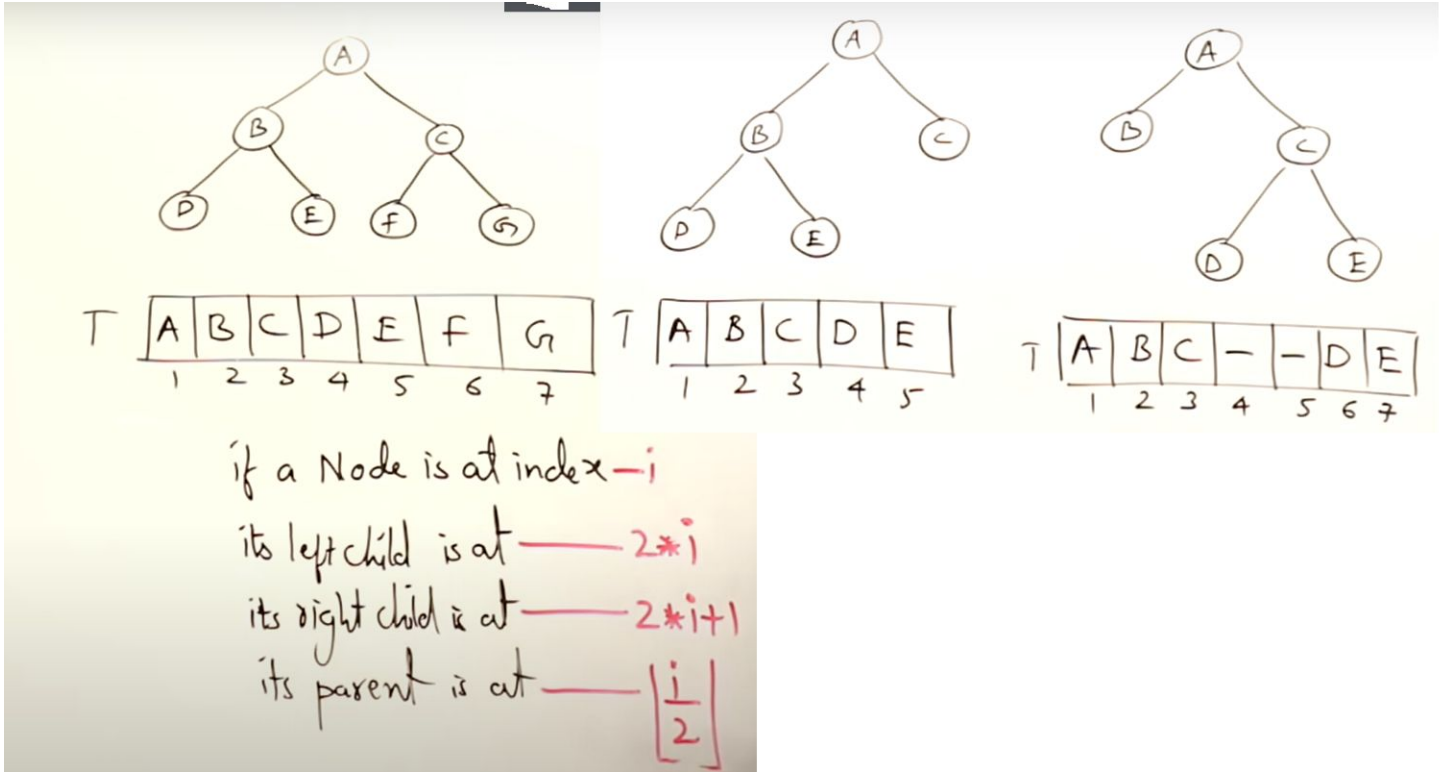
Heap

All levels are filled except the last

Nodes are added from left to right



Heap


A
B
C
D
E
F
G
T
A B C D E F G
1 2 3 4 5 6 7
A
B
C
D
E
T
A B C D E
1 2 3 4 5
A
B
C
D
E
T
A B C – – D E
1 2 3 4 5 6 7
if a Node is at index — i
its left child is at —— 2*i
its right child is at —— 2*i+1
its parent is at —— ⌊i/2⌋

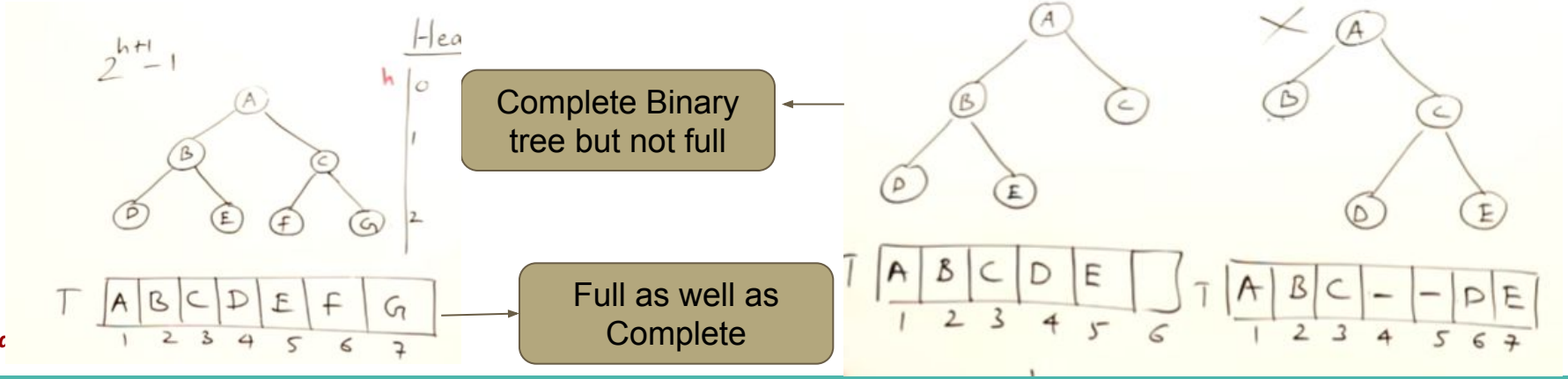
Complete and Full Binary Tree:

Every full binary tree is also complete binary tree

Complete binary tree is full binary tree upto level (l-1)

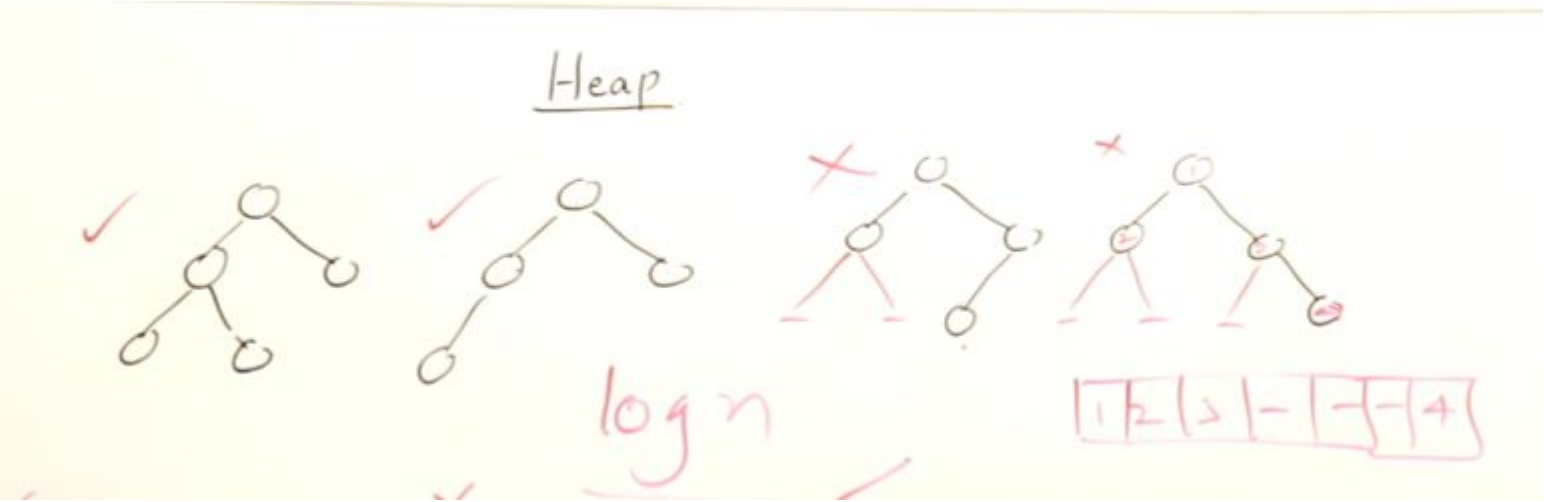
Number of nodes in Full binary tree: 2(l+1)-1, where l starts from 0

Incomplete binary tree



Height of complete binary tree

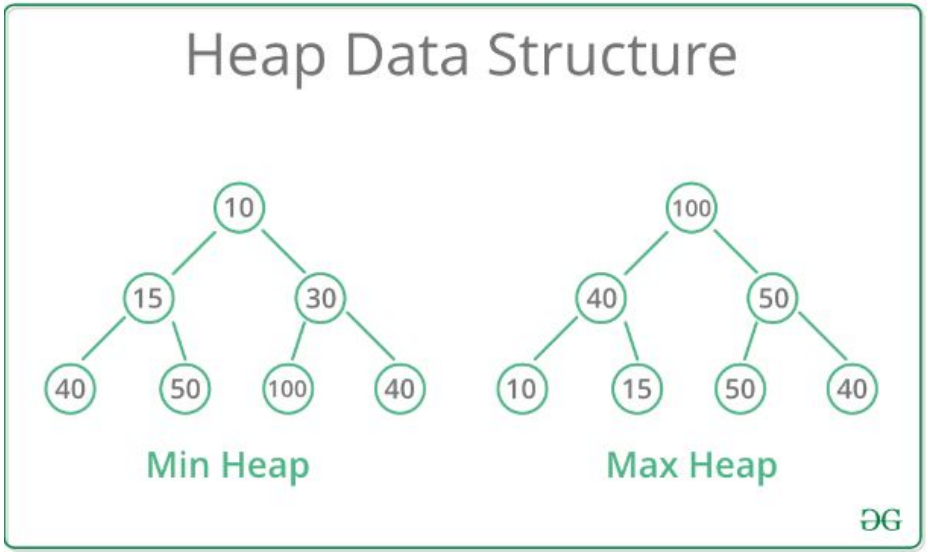
Log(n)



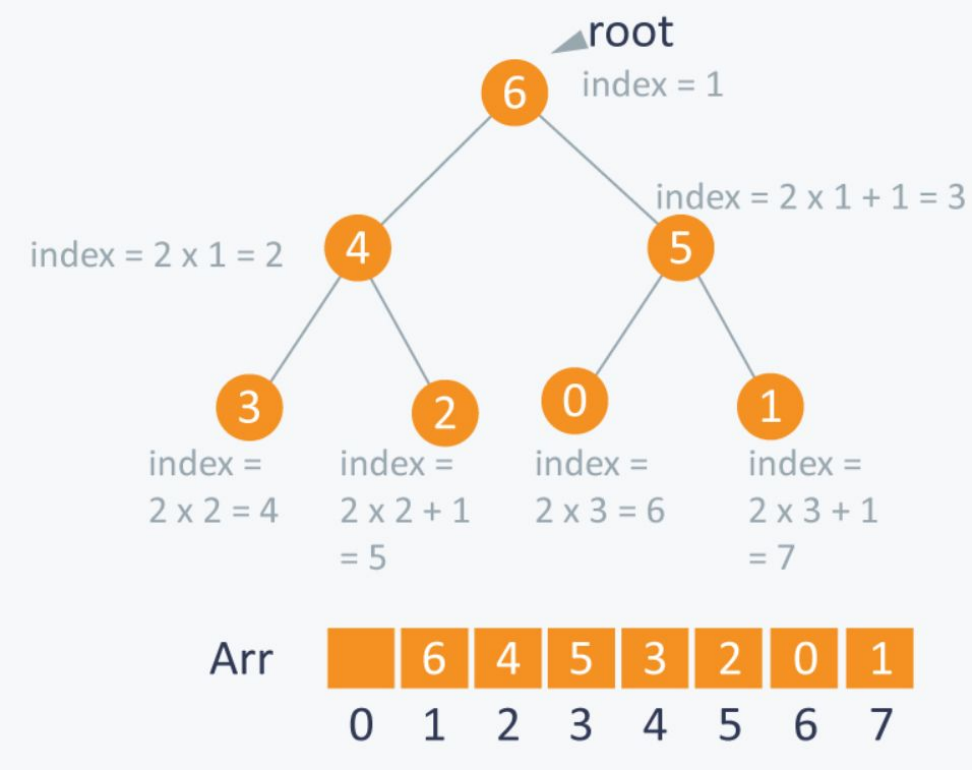
Types of Heap

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.

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



Process

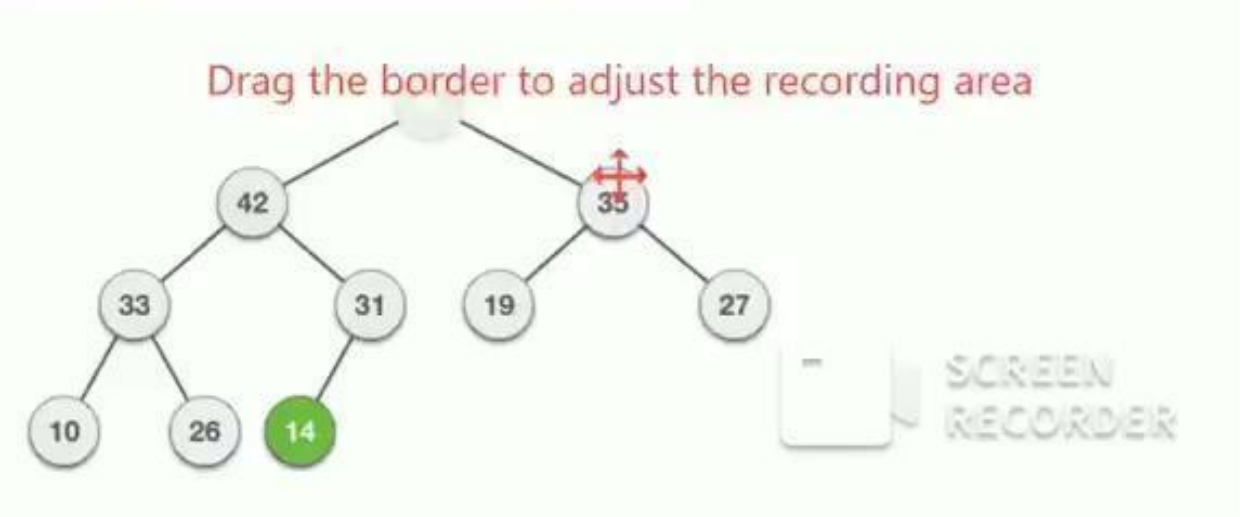

root
6
index = 1
index = 2 x 1 = 2
4
index = 2 x 1 + 1 = 3
5
3
2
0
1
index = 2 x 2 = 4
index = 2 x 2 + 1 = 5
index = 2 x 3 = 6
index = 2 x 3 + 1 = 7
Arr
6 4 5 3 2 0 1
0 1 2 3 4 5 6 7

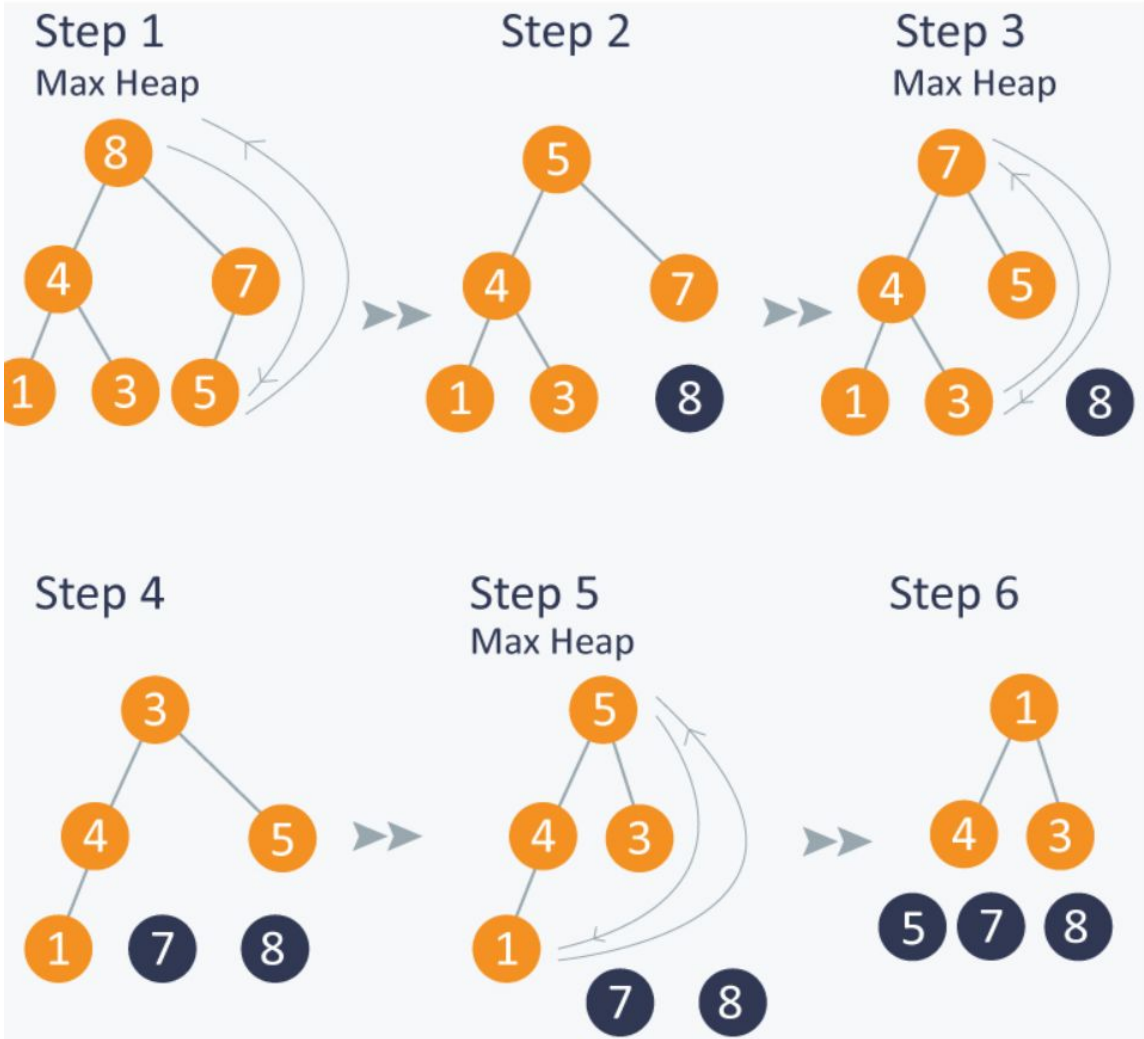
Max Heap construction

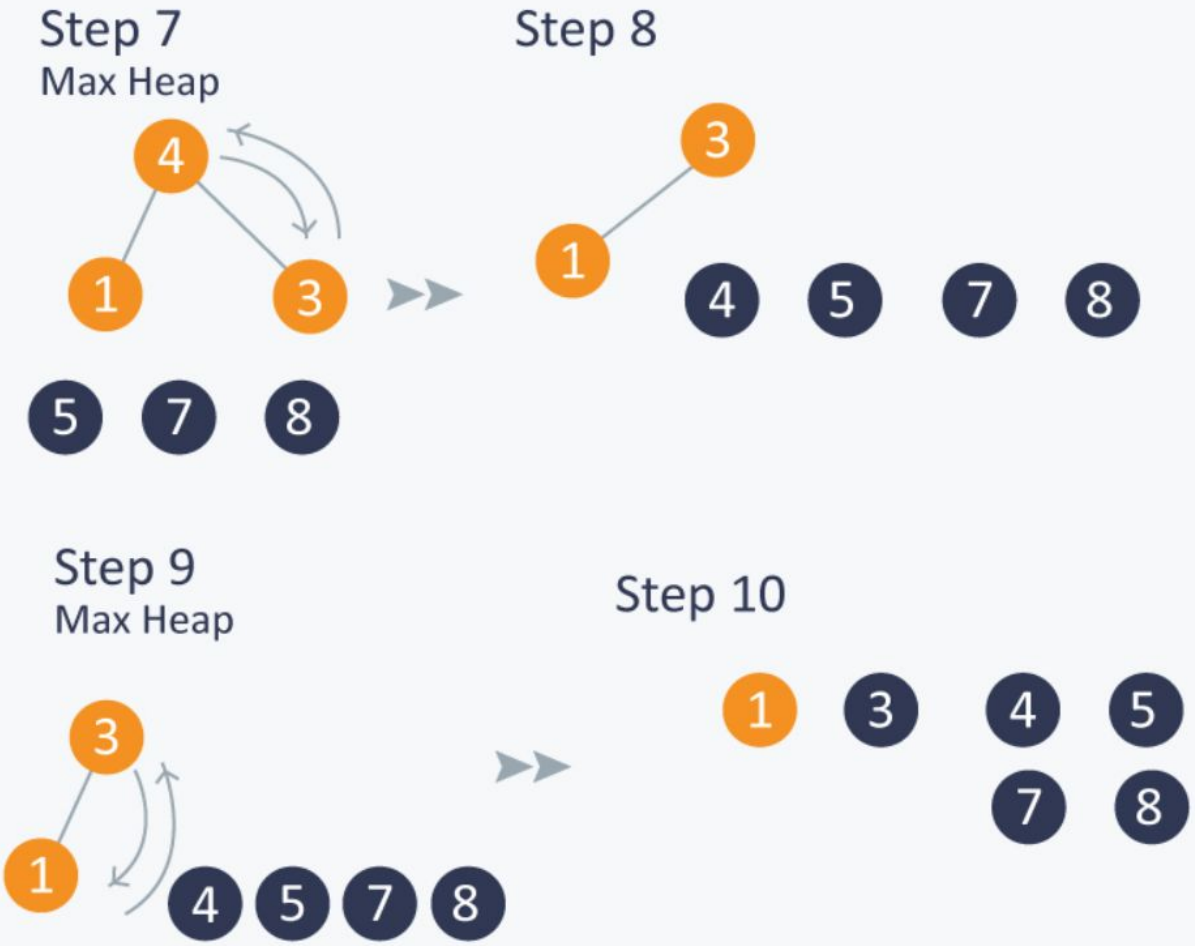
Step 1 – Create a new node at the end of heap.
Step 2 – Assign new value to the node.
Step 3 – Compare the value of this child node with its parent.
Step 4 – If value of parent is less than child, then swap them.
Step 5 – Repeat step 3 & 4 until Heap property holds.

Max heap deletion

Step 1 - Remove root node.
Step 2 - Move the last element of last level to root.
Step 3 - Compare the value of this child node with its parent.
Step 4 - If value of parent is less than child, then swap them.
Step 5 - Repeat step 3 & 4 until Heap property holds.



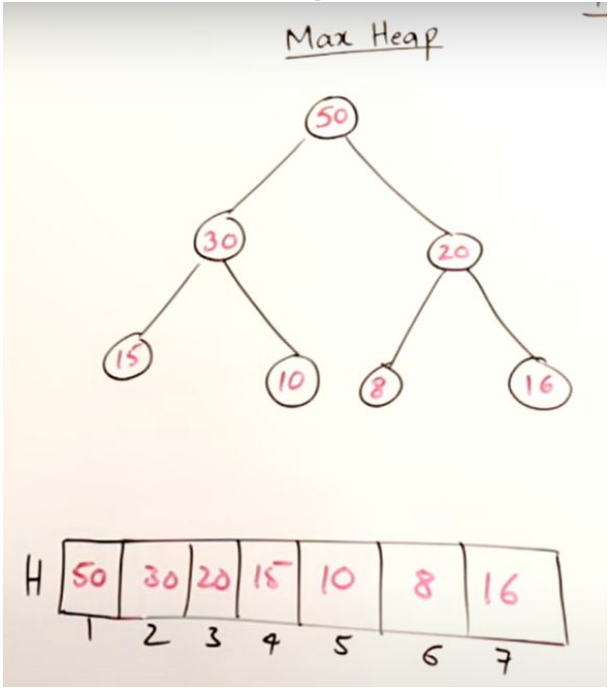

Step 1
Max Heap
8
4
7
1
3
5
Step 2
5
4
7
1
3
8
Step 3
Max Heap
7
4
5
1
3
8
Step 4
3
4
5
1
7
8
Step 5
Max Heap
5
4
3
1
7
8
Step 6
1
4
3
5
7
8


Step 7
Max Heap
4
1
3
5
7
8
Step 8
3
1
4
5
7
8
Step 9
Max Heap
3
1
4
5
7
8
Step 10
1
3
4
5
7
8

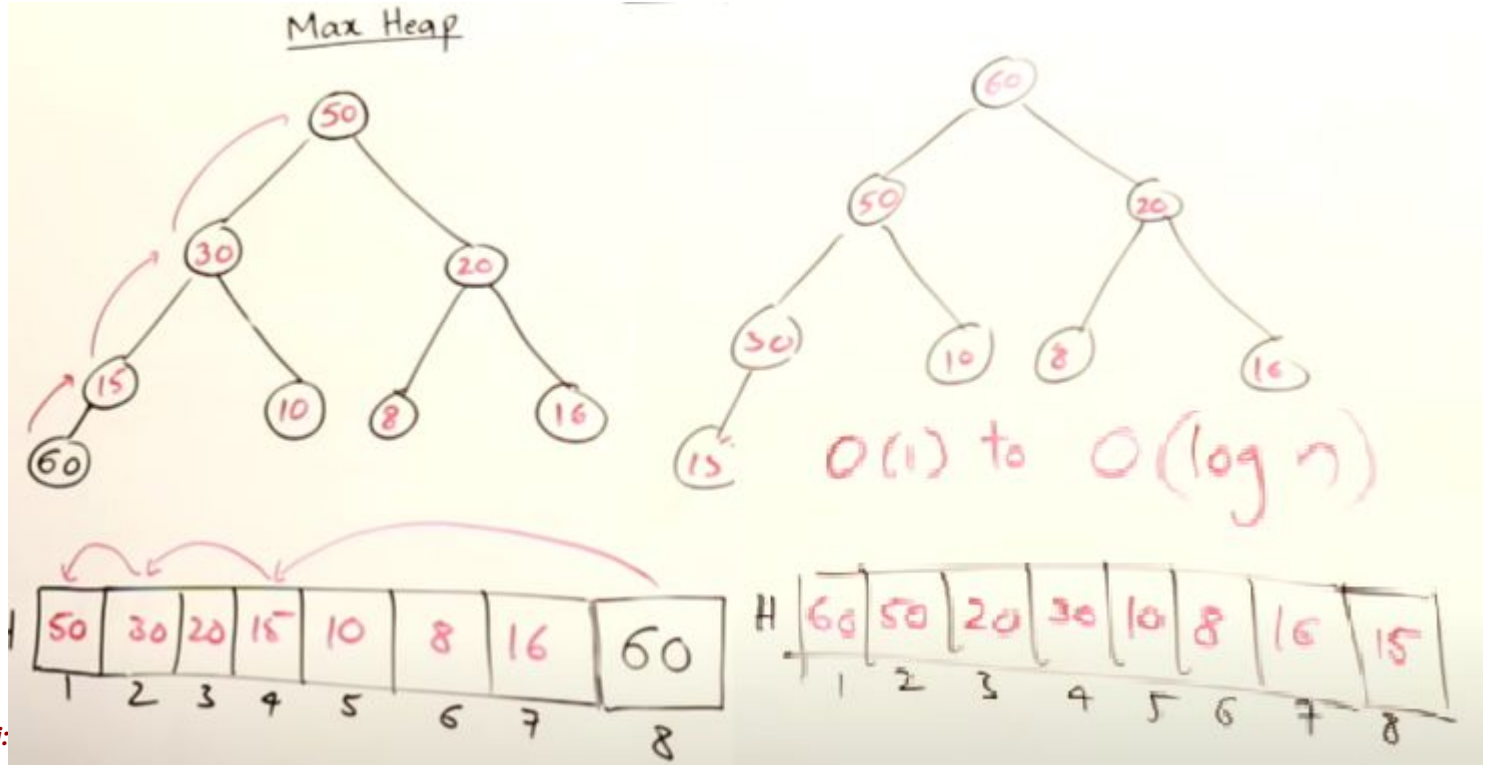
Max Heap

- Root node will have value greater than its child (both left and right)

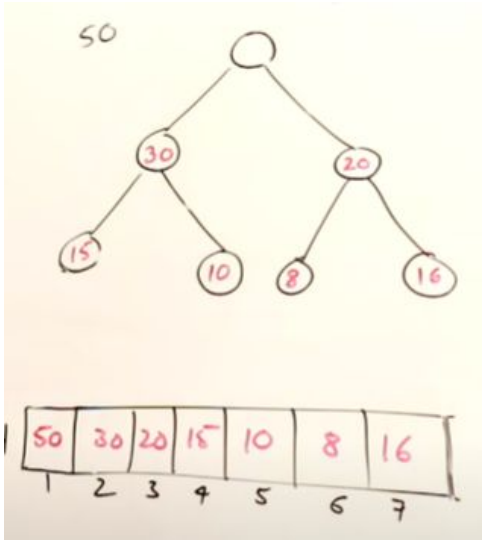
  Insertion in Max Heap

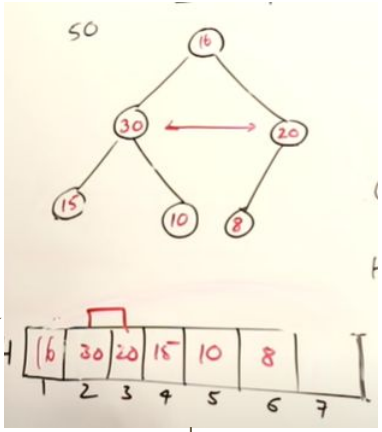


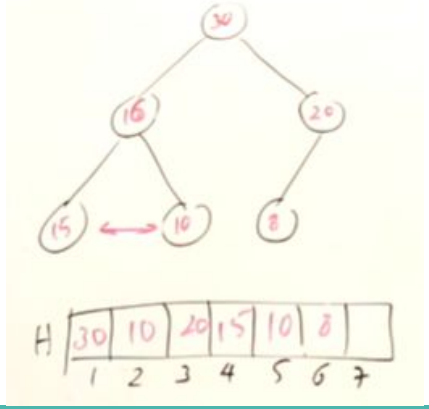
Insertion in Max Heap



Deletion in Heap







Heap Operations

getMax() or getMin()

Heapify()

Insert(k)

Binary Search Trees

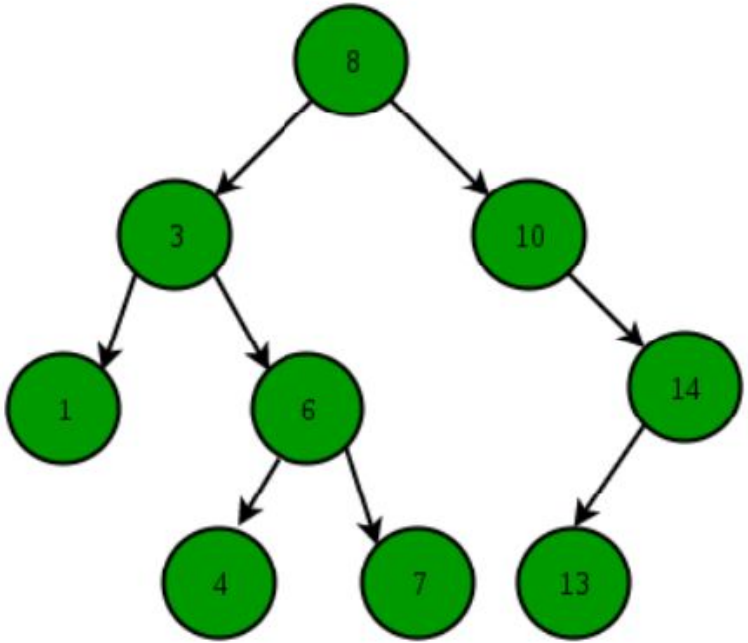
Left subtree < root

Right subtree > root

Left and right subtrees should be BSTs

Binary Search Tree is a node-based binary tree data structure which has the following properties:

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



Linked list

•A linked list is a data structure which can change during execution.

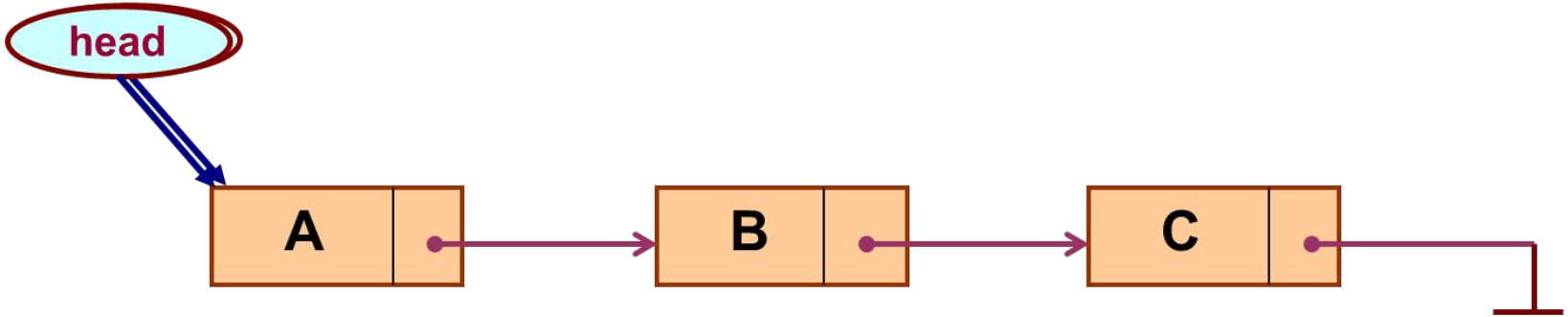
–Successive elements are connected by pointers.

–Last element points to `NULL`.

–It can grow or shrink in size during execution of a program.

–It can be made just as long as required.

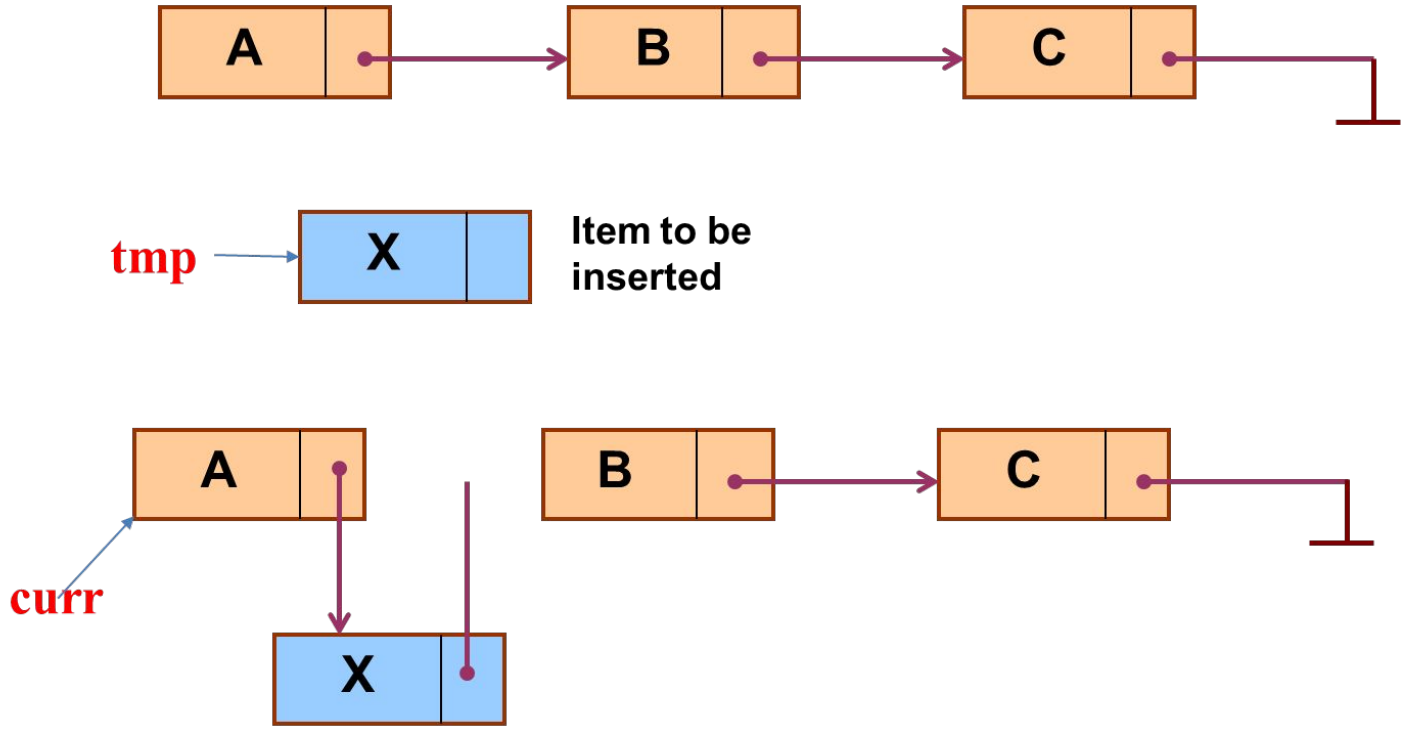
–It does not waste memory space.



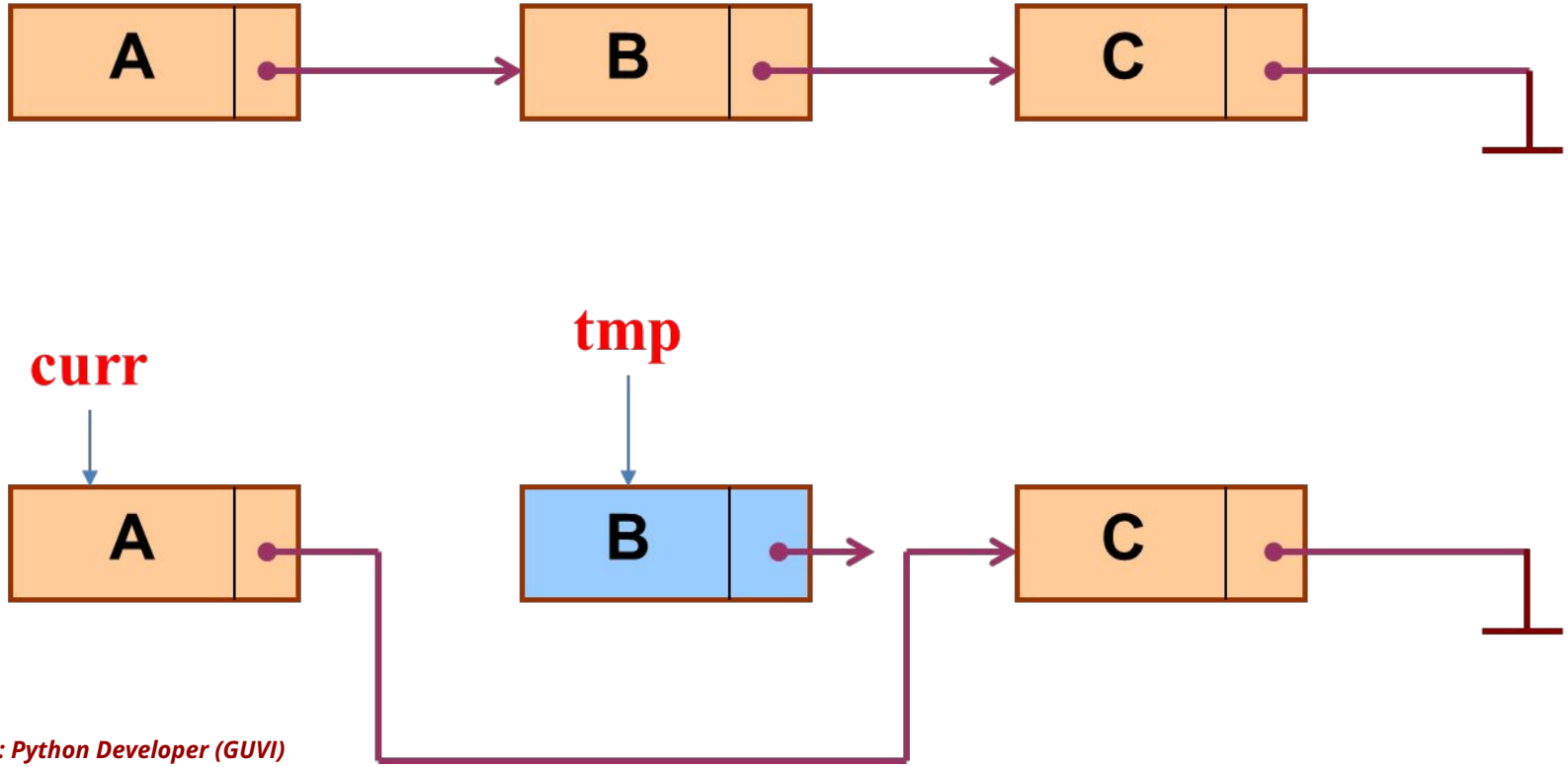
Linked List

- Keeping track of a linked list:
  - Must know the pointer to the first element of the list (called *start*, *head*, etc.).
- Linked lists provide flexibility in allowing the items to be rearranged efficiently.
  - Insert an element.
  - Delete an element.

Insertion: Linked List



Deletion: Linked List



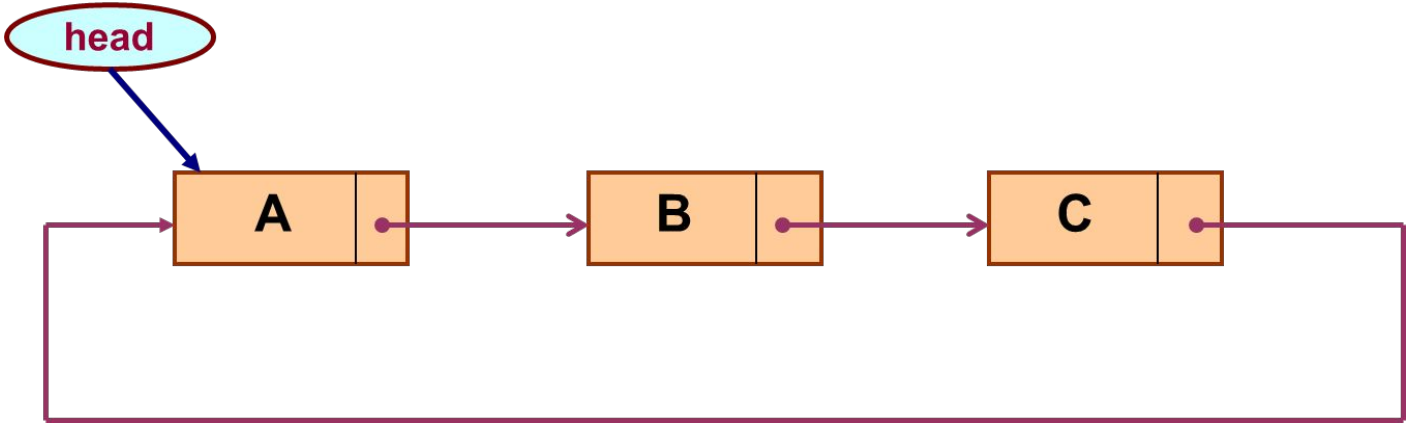
In essence

- For insertion:
  - A record is created holding the new item.
  - The next pointer of the new record is set to link it to the item which is to follow it in the list.
  - The next pointer of the item which is to precede it must be modified to point to the new item.
- For deletion:
  - The next pointer of the item immediately preceding the one to be deleted is altered, and made to point to the item following the deleted item.

Circular Linked List

–Circular linked list

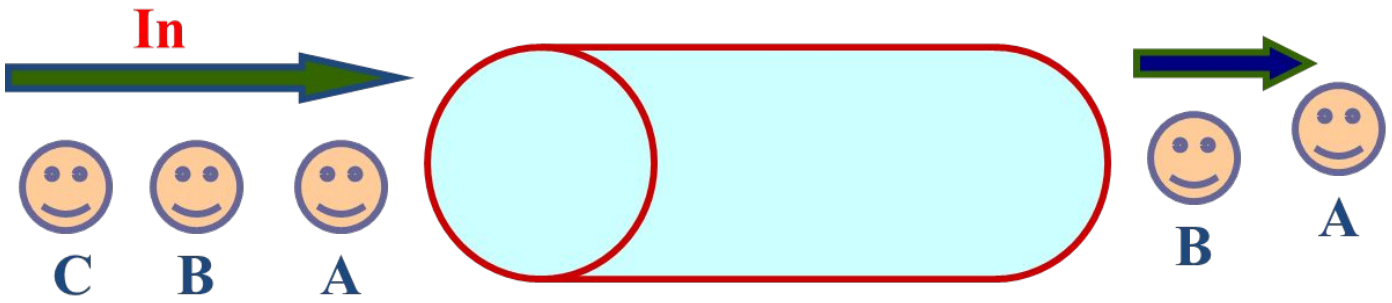
•The pointer from the last element in the list points back to the first element.



List is an Abstract Data Type

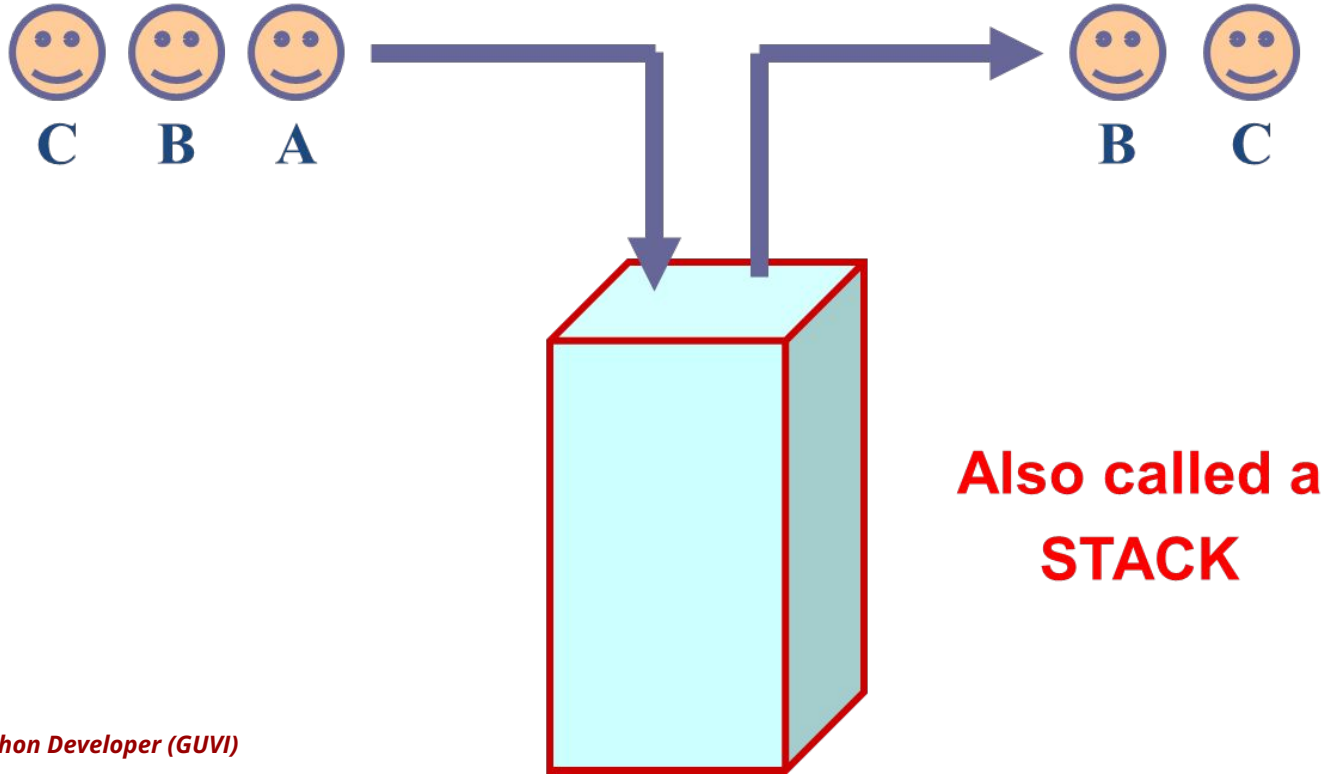
- What is an abstract data type?
  - It is a data type defined by the user.
  - Typically more complex than simple data types like *int*, *float*, etc.
- Why abstract?
  - Because details of the implementation are hidden.
  - When you do some operation on the list, say insert an element, you just call a function.
  - Details of how the list is implemented or how the insert function is written is no longer required.

A First-in First-out (FIFO) List



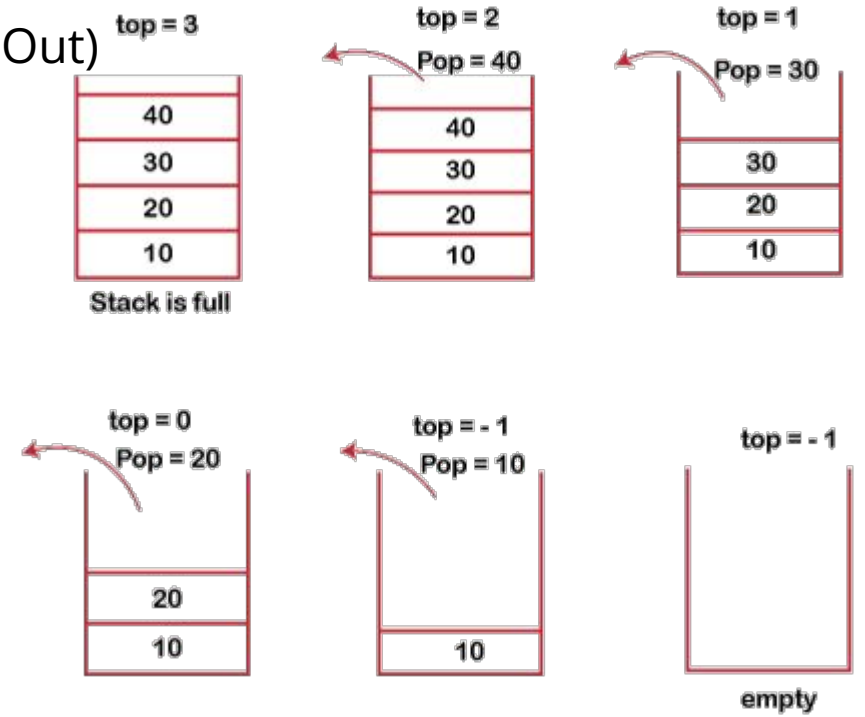
Also called a QUEUE

A Last-in First-out (LIFO) list



Stack

- Linear data structure
- The order may be LIFO(Last In First Out)
- FILO(First In Last Out)



Functions in Stack

empty()

size()

top()

push(a)

pop()

Stack Implementation

```
stack = []

stack.append('a')
stack.append('b')
stack.append('c')

print('Initial stack')
print(stack)

print('\nElements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print(stack)
```

Competitive Coding

Useful concepts to be aware of

Decorators

Decorators

Python decorators is a technique for changing the behavior of an existing function without changing actual code inside the function.

- There should not have any change in the function definition and function call.
  - The code inside the original function should not be changed.
  - There should not have any change in the function call.

```
def myFunc():
  print("Hello, World!")

myFunc()
```

```
def myWrapper(func):
  def myInnerFunc():
    print("Inside wrapper.")
    func()
  return myInnerFunc
```

```
@myWrapper
def myFunc():
  print("Hello, World!")

myFunc()
```

Example

- The original function is called (1).
- There is a function wrapper name specified above the function definition (2). This indicates, that there is a function decorator assigned to the function.
- The decorator function gets called. The program controller passes the function object as a parameter to the decorator function (3).
- The function inside the decorator function gets executed (4).
- The inner function calls the actual function (5).
- The original function starts execution (6)

Decorator Basic Example