ADVANCED PYTHON PROGRAMMING

Advanced Data Structure and Algorithm

Peter Lo

Linear Structure

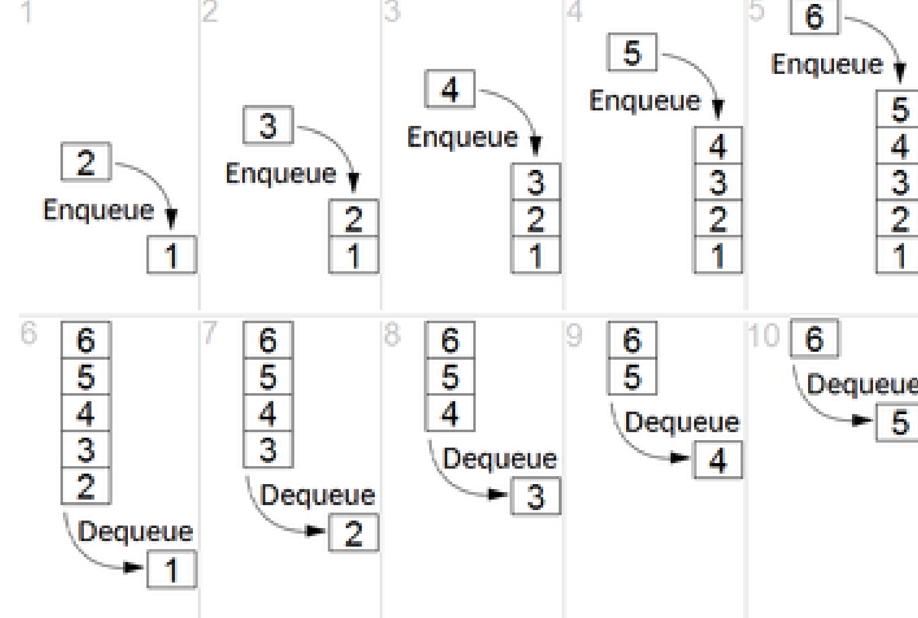
Queue, Stack, Linked List and Tree

Queue

- A queue is a line of people or things waiting to be handled, usually in sequential order starting at the beginning or top of the line or sequence.
- In computer technology, a queue is a sequence of work objects that are waiting to be processed.

Queue (Cont.)

- Queue has two main operations in data structure:
  - **Enqueue**: append an element to the tail of the queue
  - **Dequeue**: remove an element from the head of the queue
- With a queue you remove the item least recently added (First-In First-Out / FIFO)



Implement Queue by List

- It’s possible to use a regular list as a queue but this is not ideal from a performance perspective.
- Lists are quite slow for this purpose because inserting or deleting an element at the beginning requires shifting all of the other elements by one.
- Using a list as a makeshift queue in Python is not recommend unless you’re dealing with a small number of elements only.

Implement Queue by List

```
myQueue = []                # Define the Queue
myQueue.append("1")         # Enqueue "1"
myQueue.append("2")         # Enqueue "2"
myQueue                     # Print the queue
```

```
['1', '2']
```

```
myQueue.pop(0)                # Dequeue "1"
```

```
'1'
```

Implement Queue by collections.deque

- The deque class implements a double-ended queue that supports adding and removing elements from either end.
- Python’s deque objects are implemented as doubly-linked lists which gives them excellent performance for enqueuing and dequeuing elements, but poor performance for randomly accessing elements in the middle of the queue.
- Because deques support adding and removing elements from either end equally well, they can serve both as queues and as stacks.
- *collections.deque* is a great default choice if you’re looking for a queue data structure in Python’s standard library.

Implement Queue by collections.deque

```
from collections import deque

myQueue = deque()                # Define the Queue
myQueue.append("1")              # Enqueue "1"
myQueue.append("2")              # Enqueue "2"
myQueue                          # Print the queue
```

```
deque(['1', '2'])
```

```
myQueue.popleft()               # Dequeue "1"
```

```
'1'
```

Implement Queue by queue.Queue

- This queue implementation in the Python standard library is synchronized and provides locking semantics to support multiple concurrent producers and consumers.
- The queue module contains several other classes implementing multi-producer, multi-consumer queues that are useful for parallel computing.
- Depending on your use case the locking semantics might be helpful, or just incur unneeded overhead. In this case you'd be better off with using collections.deque as a general purpose queue.

Implement Queue by queue.Queue

```
from queue import Queue

myQueue = Queue()           # Define the Queue
myQueue.put("1")            # Enqueue "1"
myQueue.put("2")            # Enqueue "2"
myQueue                     # Print the queue
```

```
<queue.Queue at 0x7f0958567a58>
```

```
myQueue.get()               # Dequeue "1"
```

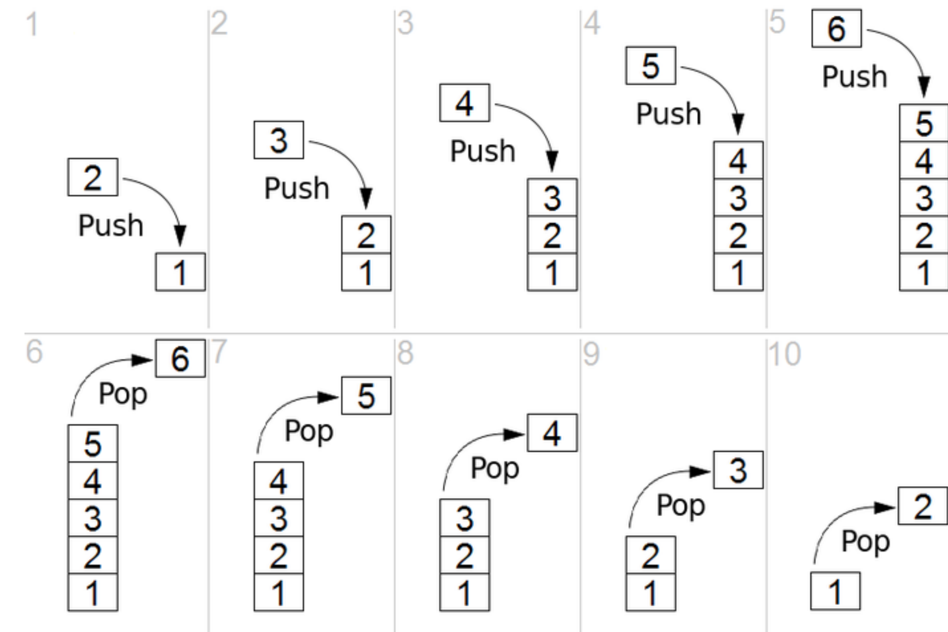
```
'1'
```

Stack

- A Stack is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end. This end is commonly referred to as the “top.” The end opposite the top is known as the “base

Stack (Cont.)

- Stack has two main operations in data structure:
  - **Push**: append an element on top of the stack
  - **Pop**: remove an element from the top of the stack
- With a stack you remove the item most recently added (Last-In First-Out / LIFO).



Implement Stack by List

- Python's built-in list type makes a decent stack data structure as it supports push and pop operations in amortized time.
- Python's lists are implemented as dynamic arrays internally which means they occasional need to resize the storage space for elements stored in them when elements are added or removed. The list over-allocates its backing storage so that not every push or pop requires resizing and you get an amortized time complexity for these operations.
- The downside is that this makes their performance less consistent than the stable inserts and deletes provided by a linked list based implementation. On the other hand lists do provide fast time random access to elements on the stack which can be an added benefit.

Implement Stack by List

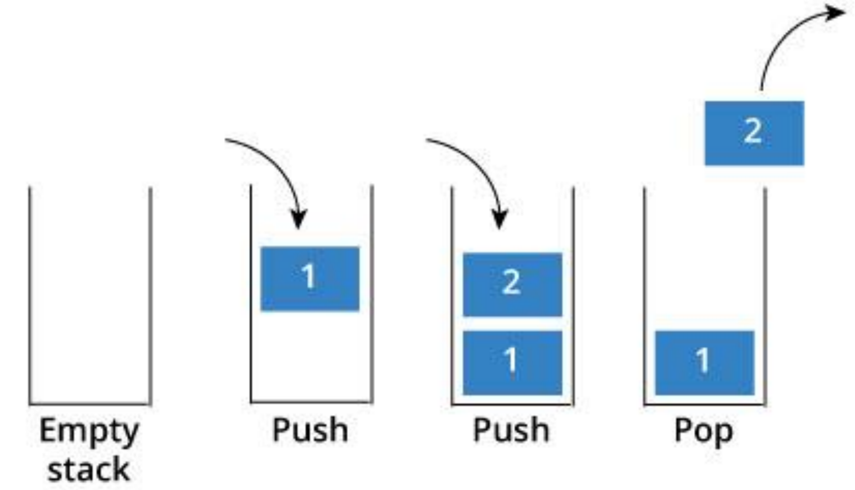
```
myStack = []                # Define the Stack

myStack.append(1)           # Push "1"
myStack.append(2)           # Push "2"
myStack                     # Print the Stack
```

```
[1, 2]
```

```
myStack.pop()               # Pop "2"
```

```
2
```



Implement Stack by collections.deque

- The deque class implements a double-ended queue that supports adding and removing elements from either end in non-amortized time.
- Because deques support adding and removing elements from either end equally well, they can serve both as queues and as stacks.
- Python's deque objects are implemented as doubly-linked lists which gives them excellent and consistent performance for inserting and deleting elements, but poor performance for randomly accessing elements in the middle of the stack.
- **collections.deque** is a great choice if you're looking for a stack data structure in Python's standard library with the performance characteristics of a linked-list implementation.

Implement Stack by collections.deque

```
from collections import deque

myStack = deque()         # Define the Stack
myStack.append(1)         # Push "1"
myStack.append(2)         # Push "2"
myStack                   # Print the Stack
```

```
deque([1, 2])
```

```
myStack.pop()             # Pop "2"
```

```
2
```

Empty stack Push Push Pop

Implement Stack by queue.LifoQueue

- This stack implementation in the Python standard library is synchronized and provides locking semantics to support multiple concurrent producers and consumers.
- The queue module contains several other classes implementing multi-producer, multi-consumer queues that are useful for parallel computing.
- Depending on your use case the locking semantics might be helpful, or just incur unneeded overhead. In this case you'd be better off with using a list or a deque as a general purpose stack.

Implement Stack by queue.LifoQueue

```
from queue import LifoQueue

myStack = LifoQueue()          # Define the Stack

myStack.put(1)                 # Push "1"
myStack.put(2)                 # Push "2"
myStack                        # Print the Stack
```

```
<queue.LifoQueue at 0x7f9e725062b0>
```

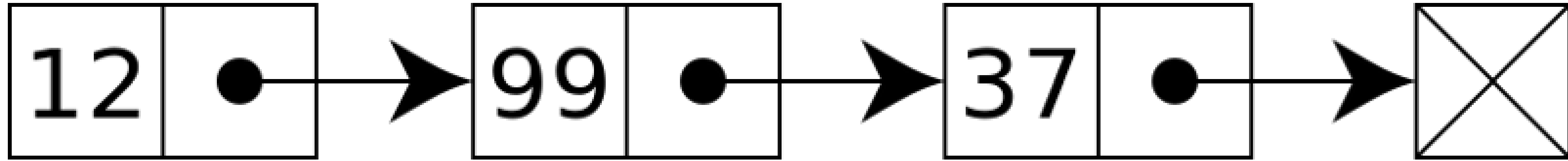
```
myStack.get()                  # Pop "2"
```

```
2
```

Empty stack | Push | Push | Pop

Linked List

- A linked list is a sequence of data elements, which are connected together via links.
- Each data element contains a connection to another data element in form of a pointer.



  - A linked list consists of nodes 12
  - Each node consists of a value 12 and a pointer to another node

Why use Linked Lists?

Advantage

- A linked list saves memory
  - It only allocates the memory required for values to be stored.
- Linked list nodes can live anywhere in the memory.
  - Each linked list node can be flexibly moved to a different address.

Disadvantage

- Linear look up time
  - When looking for a value in a linked list, you have to start from the beginning of chain, and check one element at a time for a value you're looking for.

Node Class

- The basic building block for linked list implementation is node. Each node object must hold at least two pieces of information:
  - The node must contain the list item itself. We call this the data field of the node.
  - Each node must hold a reference to the next node.
- To construct a node, you need to supply the initial data value for the node. The Node class also includes the usual methods to access and modify the data and the next reference

Node Class (Cont.)

```
class Node:
    def __init__(self, initdata):
        self.data = initdata
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self, newdata):
        self.data = newdata

    def setNext(self, newnext):
        self.next = newnext
```

The special Python reference value None play an important role in the Node class and linked list. A reference to None will denote the fact that there is no next node. Note in the constructor that a node is initially created with next set to None. Since this is sometimes referred to as "grounding the node," we will use the standard ground symbol to denote a reference that is referring to None. It is always a good idea to explicitly assign None to your initial next reference values

```
temp = Node(93)
temp.getData()
```

```
93
```

temp

data next

93

Linked List Class

- The linked list will be built from a collection of nodes, each linked to the next by explicit references. Once we find the first node, each item after that can be found by successively following the next links.
- The Linked List class must maintain a reference to the first node. Note that each list object will maintain a single reference to the head of the list.

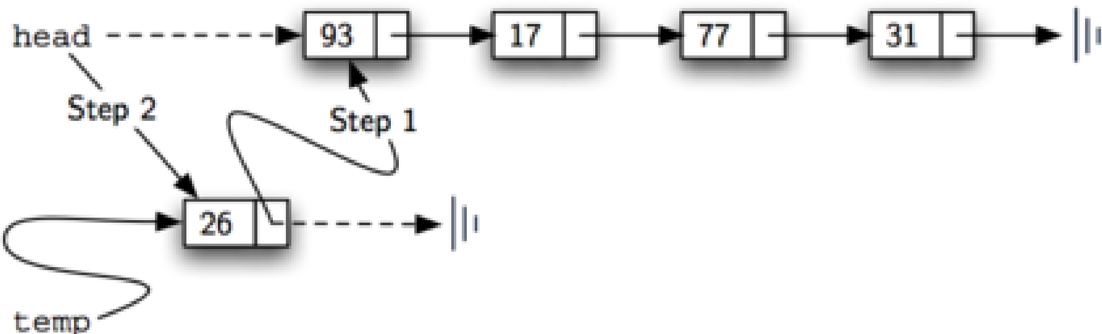
```
class LinkedList:
    def __init__(self):
        self.head = None
```

- Initially when we construct a list, there are no items.

```
mylist = LinkedList()
```

Add Node to Linked List

- Each item of the list must reside in a node object. Then creates a new node and places the item as its data. Now we must complete the process by linking the new node into the existing structure.
  1. Changes the next reference of the new node to refer to the old first node of the list. Now that the rest of the list has been properly attached to the new node.
  2. Modify the head of the list to refer to the new node.

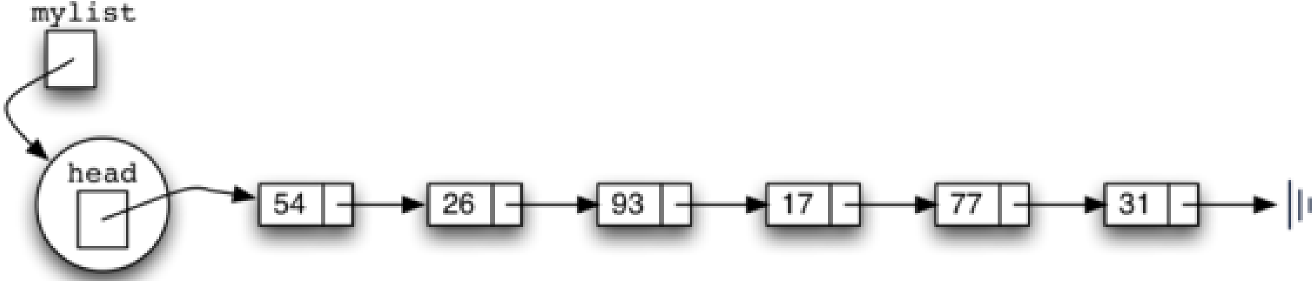


Add Node to Linked List (Cont.)

```
def add(self,item):
    temp = Node(item)
    temp.setNext(self.head)
    self.head = temp
```

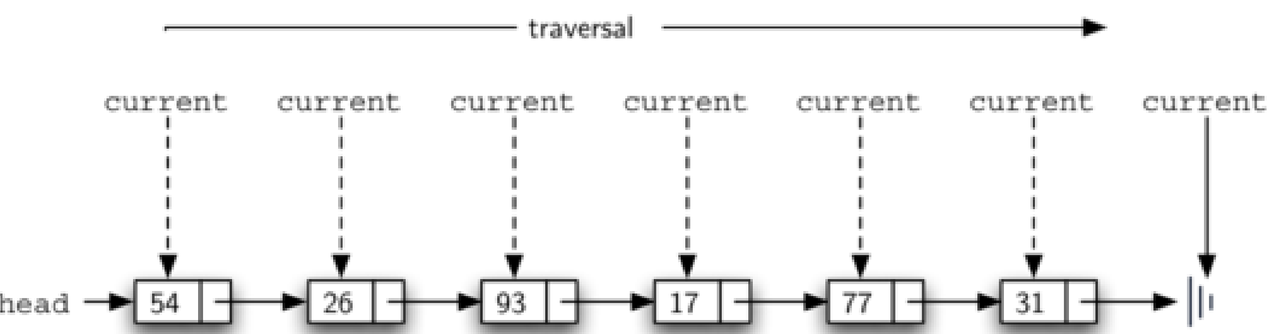
```
mylist = LinkedList()
mylist.add(31)
mylist.add(77)
mylist.add(17)
mylist.add(93)
mylist.add(26)
mylist.add(54)
```

Line 2 creates a new node and places the item as its data. Now we must complete the process by linking the new node into the existing structure. Line 3 changes the next reference of the new node to refer to the old first node of the list. Now that the rest of the list has been properly attached to the new node, we can modify the head of the list to refer to the new node. The assignment statement in line 4 sets the head of the list.



Linked List Traversal

- Linked List traversal is the process of systematically visiting each node.
- We use an external reference that starts at the first node in the list.
- As we visit each node, we move the reference to the next node by “traversing” the next reference.



Count the List Size

```
def size(self):
    current = self.head
    count = 0
    while current != None:
        count = count + 1
        current = current.getNext()
    return count
```

```
mylist.size()
```

```
6
```

To implement the size method, we need to traverse the linked list and keep a count of the number of nodes that occurred. The external reference is called current and is initialized to the head of the list in line 2. At the start of the process we have not seen any nodes so the count is set to 0. Lines 4–6 actually implement the traversal. As long as the current reference has not seen the end of the list (None), we move current along to the next node via the assignment statement in line 6. Again, the ability to compare a reference to None is very useful. Every time current moves to a new node, we add 1 to count. Finally, count gets returned after the iteration stops.

Print the List Node

```
def printList(self):
    current = self.head
    while current is not None:
        print (current.data)
        current = current.getNext()
```

```
mylist.printList()
```

```
54
26
93
17
77
31
```

To implement the print method, we need to traverse the linked list and keep a count of the number of nodes that occurred. The external reference is called current and is initialized to the head of the list in line 2. Lines 3–5 actually implement the traversal. As long as the current reference has not seen the end of the list (None), we move current along to the next node via the assignment statement in line 5. Again, the ability to compare a reference to None is very useful. Every time current moves to a new node, we print the value. Finally, whole list printed out after the iteration stops.

Search in Linked List

- Searching for a value in a linked list implementation of an unordered list also uses the traversal technique.
- As we visit each node in the linked list we will ask whether the data stored there matches the item we are looking for.
- We may not have to traverse all the way to the end of the list. In fact, if we do get to the end of the list, that means that the item we are looking for must not be present. Also, if we do find the item, there is no need to continue.

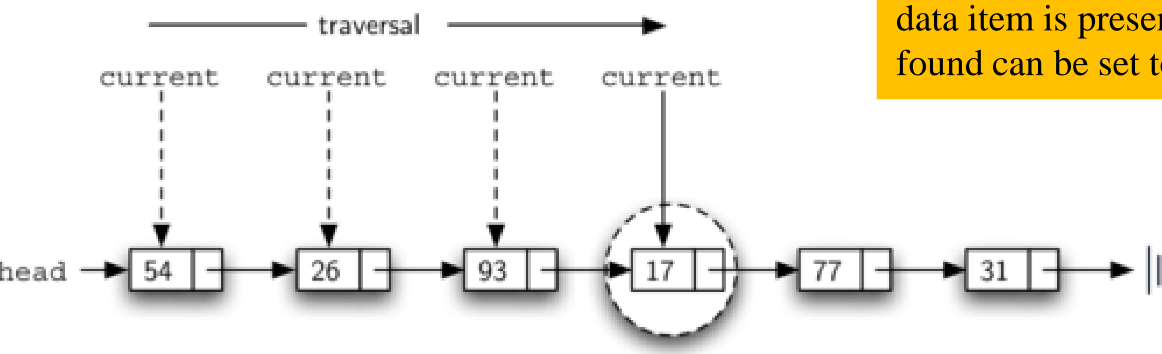
Search in Linked List (Cont.)

```
def search(self,item):
    current = self.head
    found = False
    while current != None and not found:
        if current.getData() == item:
            found = True
        else:
            current = current.getNext()
    return found
```

```
mylist.search(17)
```

```
True
```

The traversal is initialized to start at the head of the list (line 2). We also use a Boolean variable called found to remember whether we have located the item we are searching for. Since we have not found the item at the start of the traversal, found can be set to False (line 3). The iteration in line 4 takes into account both conditions discussed above. As long as there are more nodes to visit and we have not found the item we are looking for, we continue to check the next node. The question in line 5 asks whether the data item is present in the current node. If so, found can be set to True.

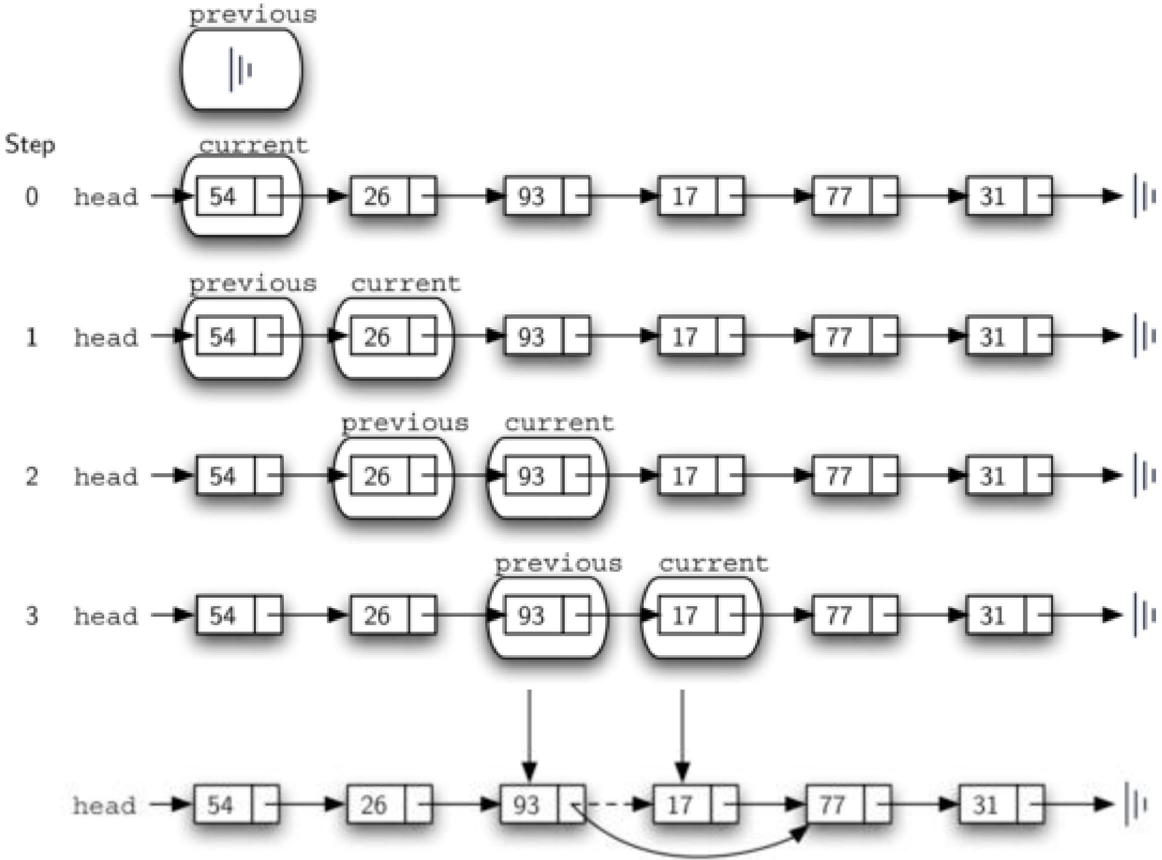


Remove a Node in Linked List

- The remove method requires two logical steps.
  - Traverse the list looking for the item that want to remove: starting with an external reference set to the head of the list, we traverse the links until we discover the item we are looking for.
  - Remove the item once we find the item by modifying the link in the previous node so that it refers to the node that comes after current.

Remove a Node in Linked List (Cont.)

- Movement of previous and current as they progress down the list looking for the node containing the value 17.



Remove a Node in Linked List (Cont.)

```
def remove(self,item):
    current = self.head
    previous = None
    found = False
    while not found:
        if current.getData() == item:
            found = True
        else:
            previous = current
            current = current.getNext()
    if previous == None:
        self.head = current.getNext()
    else:
        previous.setNext(current.getNext())
```

```
mylist.remove(17)
mylist.printList()
```

```
54
26
93
77
31
```

Lines 2–3 assign initial values to the two references. Note that current starts out at the list head as in the other traversal examples. previous, however, is assumed to always travel one node behind current. For this reason, previous starts out with a value of None since there is no node before the head. The Boolean variable found will again be used to control the iteration.

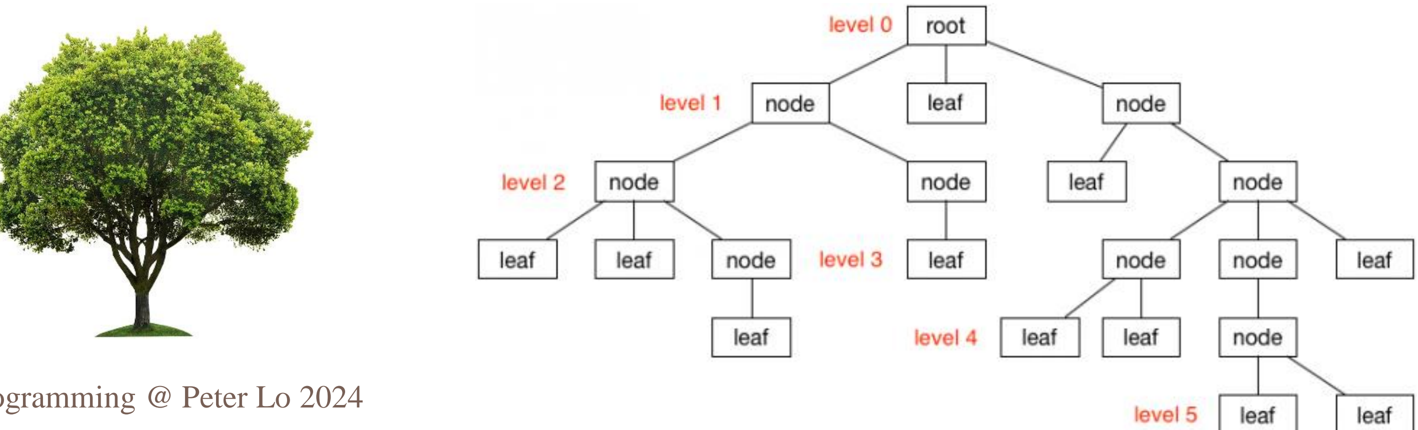
In lines 6–7 we ask whether the item stored in the current node is the item we wish to remove. If so, found can be set to True. If we do not find the item, previous and current must both be moved one node ahead. Again, the order of these two statements is crucial. previous must first be moved one node ahead to the location of current. At that point, current can be moved. This process is often referred to as “inch-worming” as previous must catch up to current before current moves ahead.

Tree Structure

Binary Tree

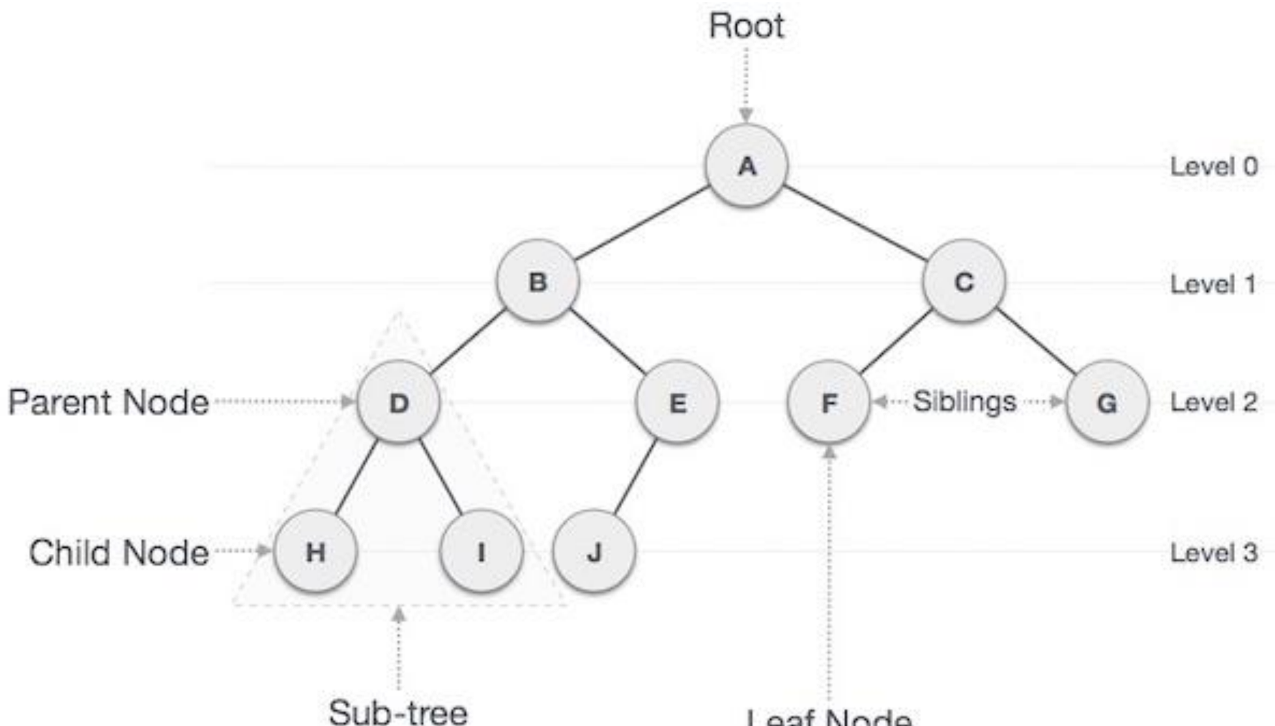
Tree Structure

- In computer science, a tree is a data structure that is modeled after nature.
- Unlike trees in nature, the tree data structure is upside down: the root of the tree is on top.
- A tree consists of nodes and its connections are called edges. The bottom nodes are also named leaf nodes.
- A tree may not have a cycle.



Binary Tree

- A binary tree is a data structure where every node has at most two children (left and right child).
- The root of a tree is on top. Every node below has a node above known as the parent node.

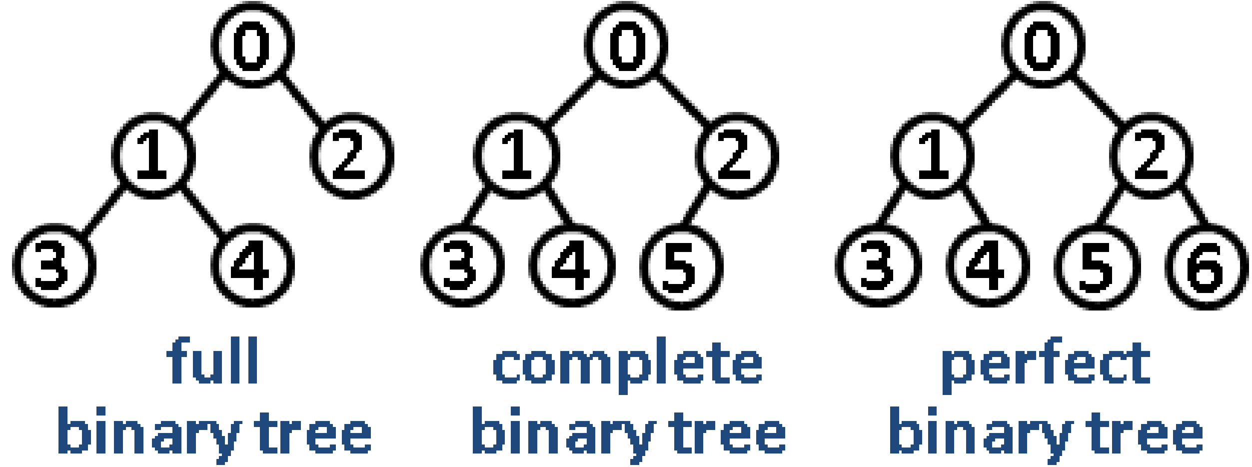


Terminology

Term	Explanation
Path	The sequence of nodes along the edges of a tree
Root	The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
Parent	Any node except the root node has one edge upward to a node called parent.
Child	The node below a given node connected by its edge downward is called its child node.
Leaf	The node which does not have any child node is called the leaf node.
Subtree	Subtree represents the descendants of a node.
Level	Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
Key	Key represents a value of a node based on which a search operation is to be carried out for a node.

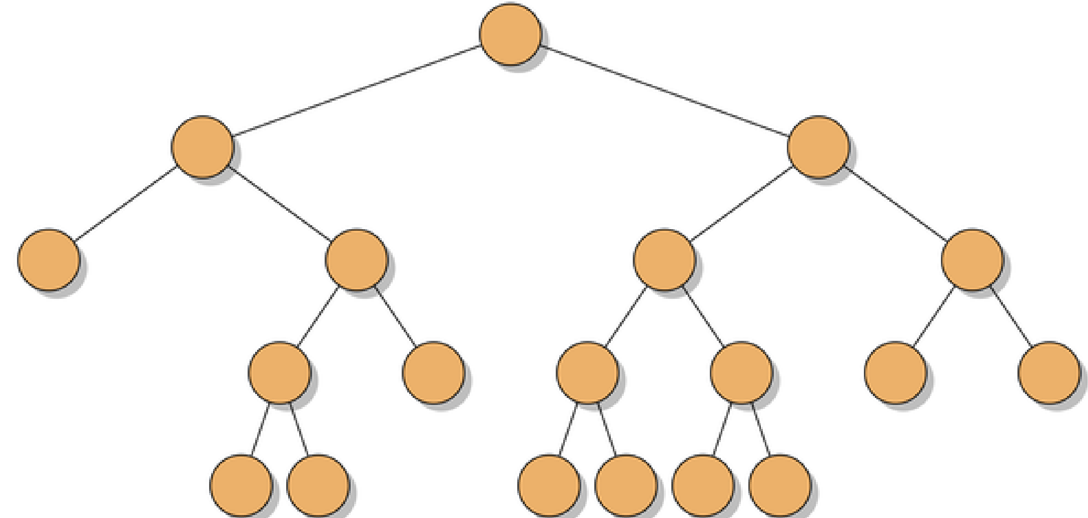
Types of Binary Trees

- There are three type of Binary Tree



Full Binary Tree

- A Binary Tree is full if every node has 0 or 2 children. Following are examples of full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaves have two children.



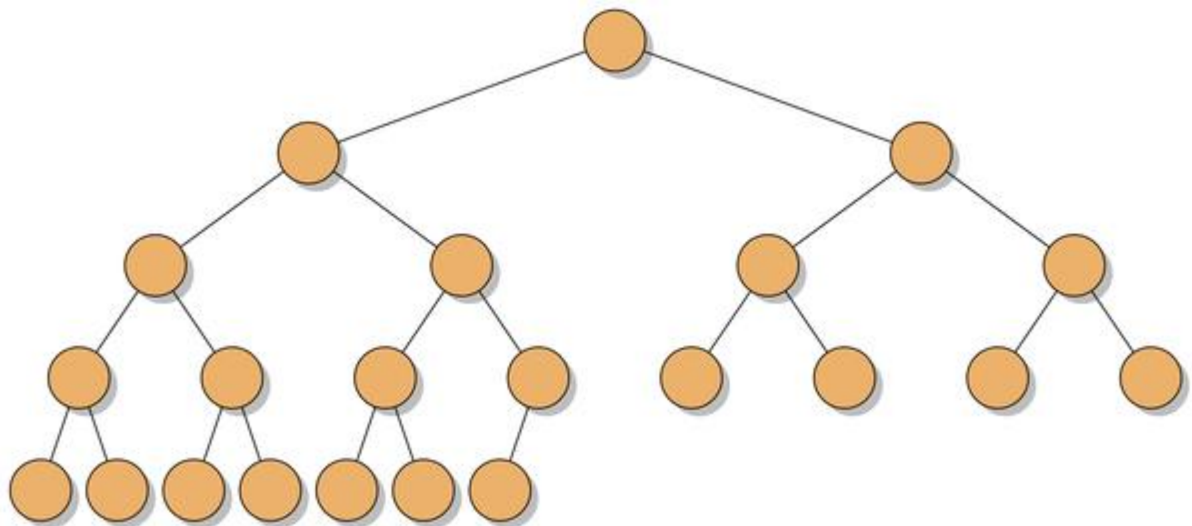
In a Full Binary tree, number of leaf nodes is number of internal nodes plus 1:

- $L = I + 1$

Where L = Number of leaf nodes, I = Number of internal nodes

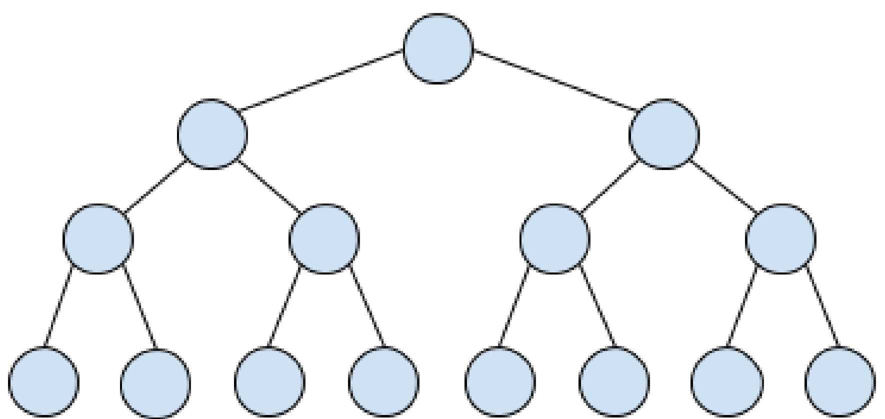
Complete Binary Tree

- A Binary Tree is complete if all levels are completely filled except possibly the last level, and the last level has all keys as left as possible



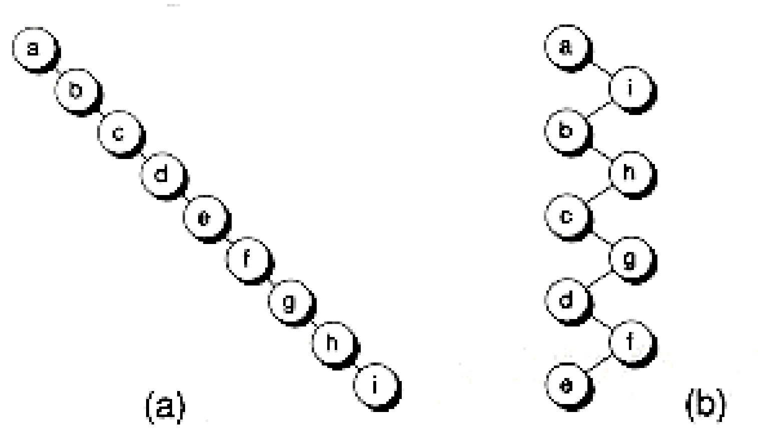
Perfect Binary Tree

- A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.
- A Perfect Binary Tree of height h (where height is number of nodes on path from root to leaf) has 2h – 1 node.



Degenerate Tree

- A degenerate (or pathological) tree is where each parent node has only one associated child node.
- This means that performance-wise, the tree will behave like a linked list data structure



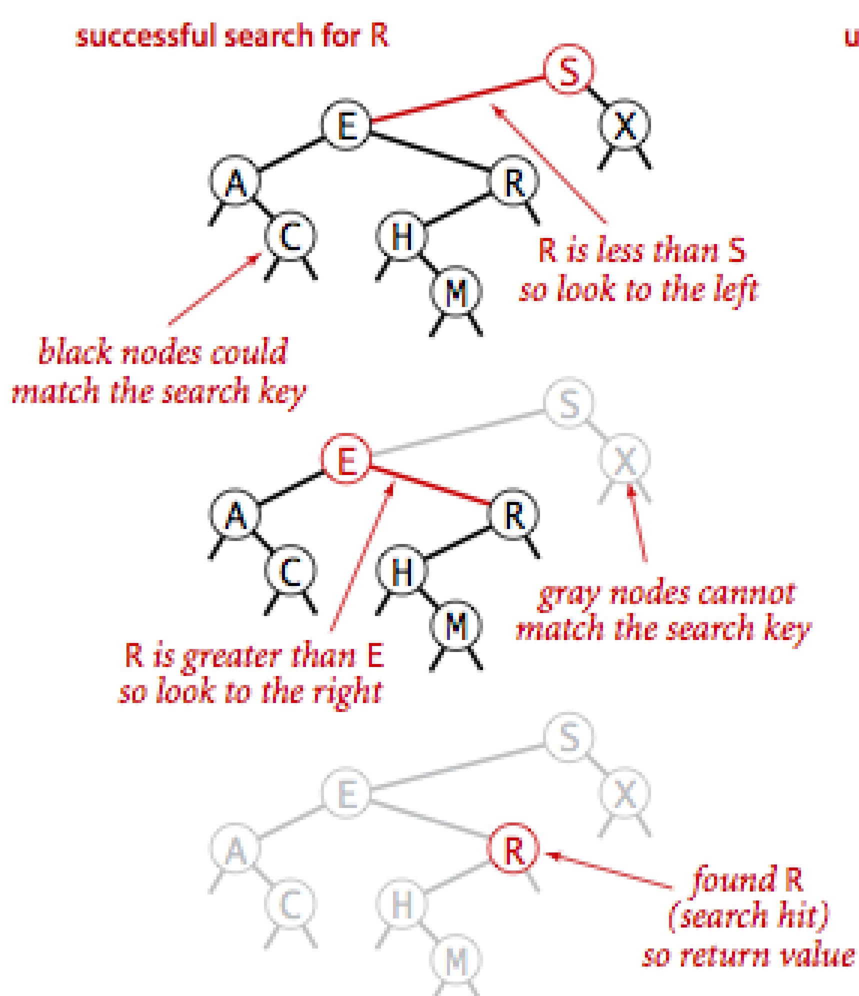
Binary Search Tree

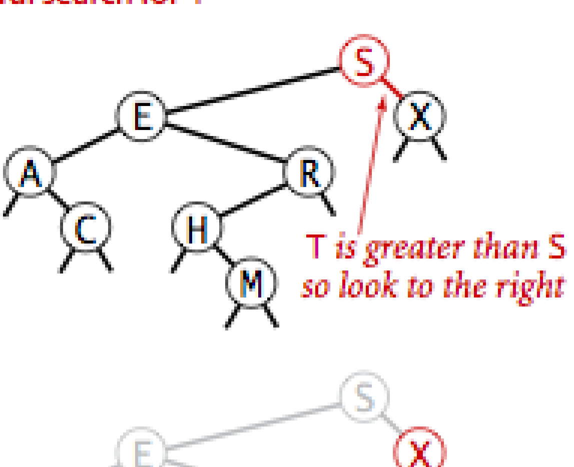
- Binary Search Tree is a data structure that quickly allows us to maintain a sorted list of numbers.
- The properties that separates a binary search tree from a regular binary tree is
  - It is called a binary tree because each tree node has maximum of two children.
  - All nodes of left subtree are less than root node
  - All nodes of right subtree are more than root node
  - Both subtrees of each node are also BST (i.e. they have the above two properties)

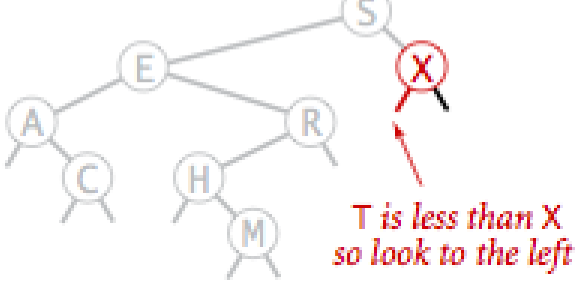
Searching in BST

- A recursive algorithm to search for a key in a BST follows immediately from the recursive structure:
  - If the tree is empty, we have a search miss
  - If the search key is equal to the key at the root, we have a search hit.
- Otherwise, we search recursively in the appropriate subtree.

Example







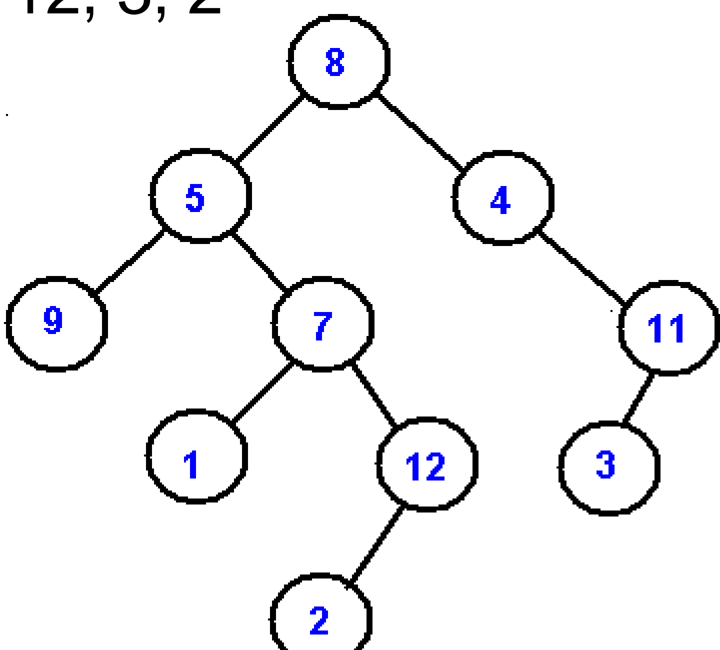
Binary Tree Traversal

- Traversal is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal.
  - Depth-First Traversal
    - Pre Order Traversal – visit the parent first and then left and right children
    - In Order Traversal – visit the left child, then the parent and the right child
    - Post Order Traversal – visit left child, then the right child and then the parent
  - Breadth-First Traversal
    - Level Order Traversal – visits nodes by levels from top to bottom and from left to right

preorder inorder postorder

Example

- Consider the following tree and its traversals:
  - Pre Order: 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
  - In Order: 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
  - Post Order: 9, 1, 2, 12, 7, 5, 3, 11, 4, 8
  - Level Order: 8, 5, 4, 9, 7, 11, 1, 12, 3, 2



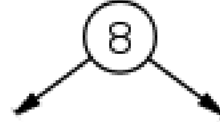
Create Node Class

- A new class named Node with 3 attributes is create to represent a tree node:
  - Left node
  - Right node
  - Node's data

```
class Node:
    def __init__(self, data):    # Node constructor
        self.left = None
        self.right = None
        self.data = data
```

- When you create a node, both left and right node equal to None. Let's create a tree node containing the value 8.

```
root = Node(8)
```



Insert a Tree Node

- We need a method to help us populate our tree.
- This method takes the node's data as an argument and inserts a new node in the tree.

```
def insertLeft(self, data):    # Insert Left node with data
    if self.data:
        if self.left is None:
            self.left = Node(data)
        else:
            self.left.insertLeft(data)
    else:
        self.data = data

def insertRight(self, data):    # Insert Right node with data
    if self.data:
        if self.right is None:
            self.right = Node(data)
        else:
            self.right.insertRight(data)
    else:
        self.data = data
```

Example

- The inset method is called recursively as we are locating the place where to add the new node.

```
root = Node(8)
root.insertLeft(3)
root.insertRight(10)
root.insertLeft(1)
```

8
3
10
1

- Detail Algorithm:
  - Left child for root node "8" is None, attach the new node "3" to it.
  - Right child for root node "8" is None, attach the new node "10" to it.
  - Left child for root node "8" is not None, go to node "3"
  - Left child for root node "3" is None, attach the new node "1" to it.

Using binarytree Library

- binarytree (https://pypi.org/project/binarytree/) is a Python library which provides a simple API to generate, visualize, inspect and manipulate binary trees.
- It allows you to skip the tedious work of setting up test data, and dive straight into practicing your algorithms.
- Heaps and Binary Search Tree (BST) are also supported.

```
!pip install binarytree
```

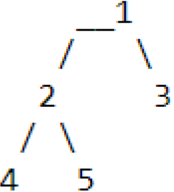
```
Collecting binarytree
  Downloading https://files.pythonhosted.org/packages/4f/df/43fe21208abbb67a37c9aa1b970de64
tree-4.0.0.tar.gz
Building wheels for collected packages: binarytree
  Running setup.py bdist_wheel for binarytree ... done
  Stored in directory: /home/nbuser/.cache/pip/wheels/f0/a2/28/6186a7ac05e3a38d3f3823bc6a38
Successfully built binarytree
Installing collected packages: binarytree
Successfully installed binarytree-4.0.0
```

Example

```python
from binarytree import Node

# Create a sample tree
Simple_Tree = Node(1)
Simple_Tree.left = Node(2)
Simple_Tree.right = Node(3)
Simple_Tree.left.left = Node(4)
Simple_Tree.left.right = Node(5)

# Print the Tree
print(Simple_Tree)
```



Example (Cont.)

```
# Inorder Traversal
Simple_Tree.inorder
```

```
[Node(4), Node(2), Node(5), Node(1), Node(3)]
```

```
# Preorder Traversal
Simple_Tree.preorder
```

```
[Node(1), Node(2), Node(4), Node(5), Node(3)]
```

```
# Postorder Traversal
Simple_Tree.postorder
```

```
[Node(4), Node(5), Node(2), Node(3), Node(1)]
```

```
# Levelorder Traversal
Simple_Tree.levelorder
```

```
[Node(1), Node(2), Node(3), Node(4), Node(5)]
```

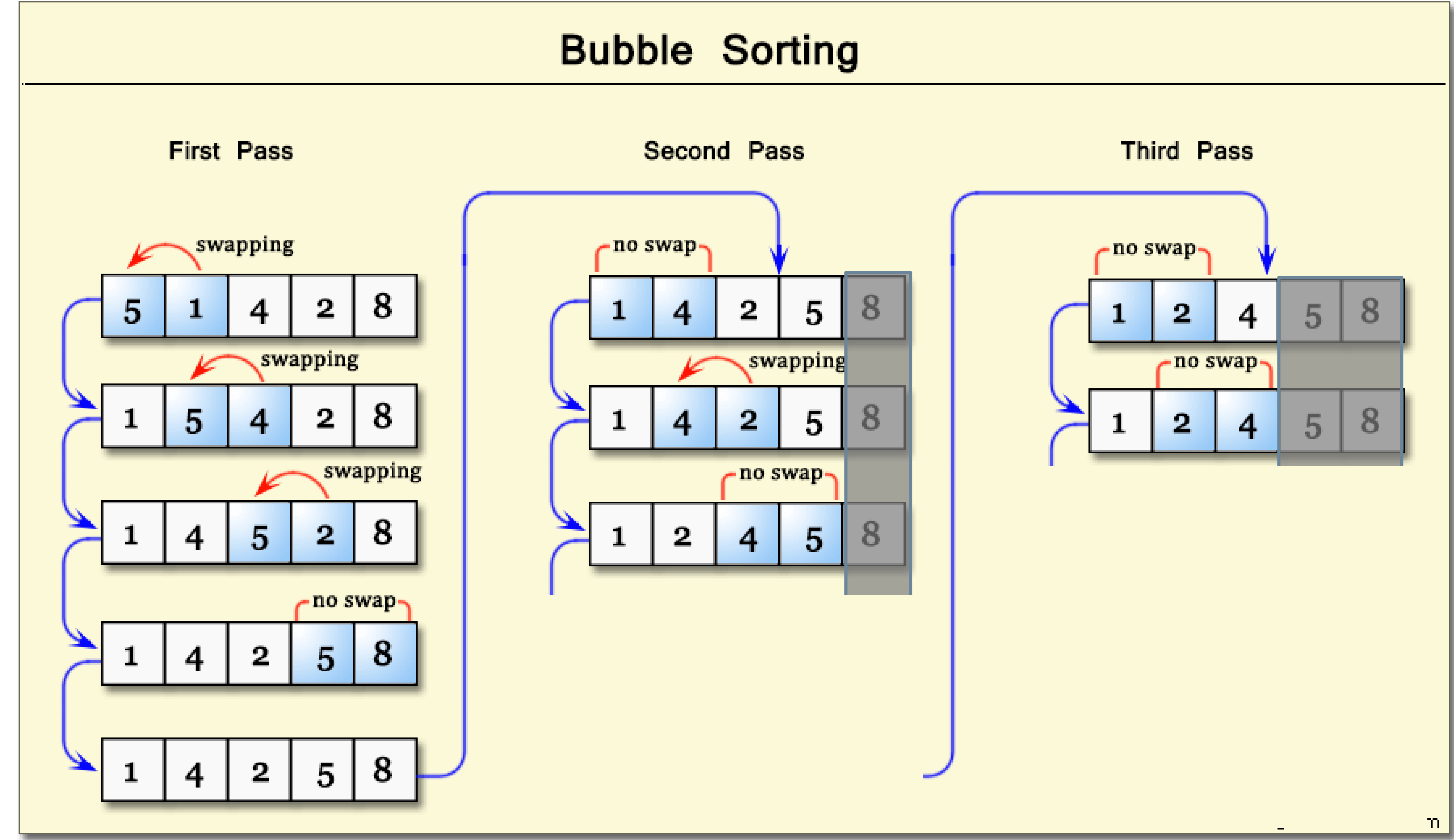
Sorting Algorithms

Put Elements of List in Certain Order

Bubble Sort

- The bubble sort makes multiple passes through a list.
- It compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. In essence, each item “bubbles” up to the location where it belongs.
- If there are n items in the list, then there are n−1 pairs of items that need to be compared on the first pass. It is important to note that once the largest value in the list is part of a pair, it will continually be moved along until the pass is complete.

Bubble Sort Workflow



Example

```
def BubbleSort(arr):
    # Get the length of input list
    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 > next element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

# Sort and print out result
InputList = [5, 4, 2, 1, 3]
BubbleSort(InputList)
print ("Sorted List:", InputList)
```

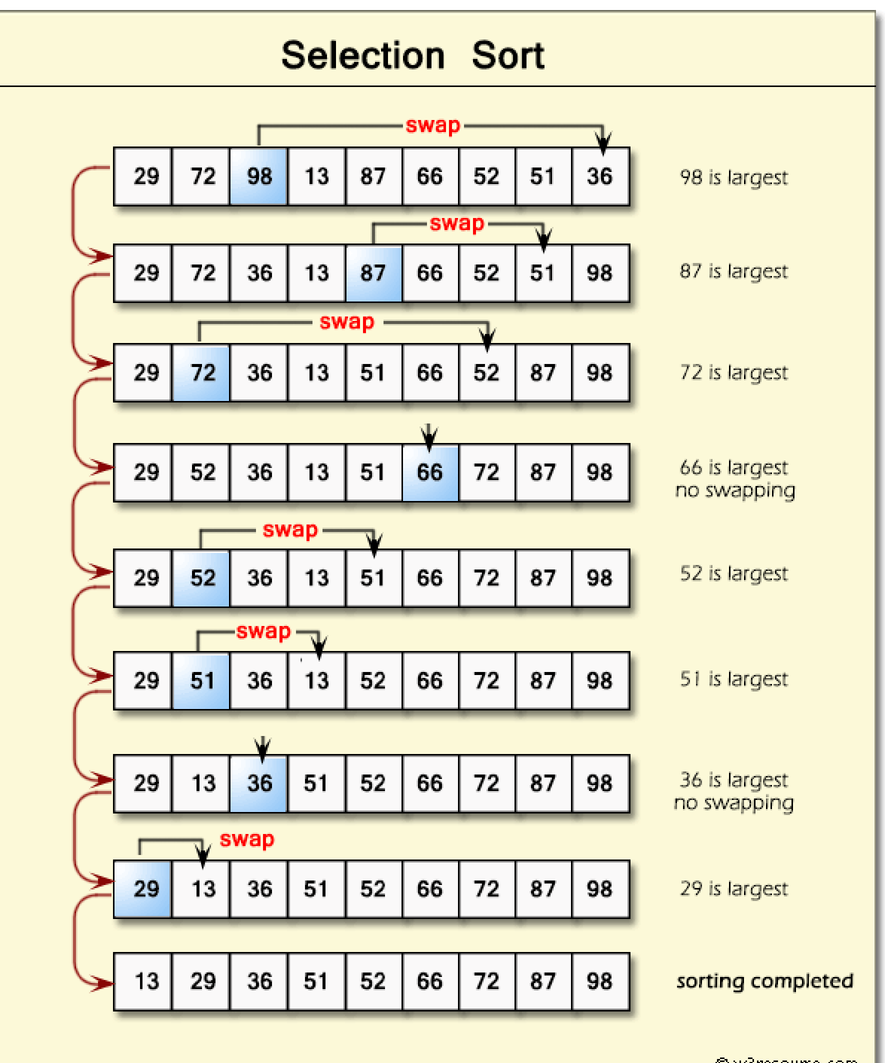
```
Sorted List: [1, 2, 3, 4, 5]
```

Selection Sort

- The selection sort improves on the bubble sort by making only one exchange for every pass through the list.
- In order to do this, a selection sort looks for the smallest/largest value as it makes a pass and, after completing the pass, places it in the proper location.
- As with a bubble sort, after the first pass, the largest item is in the correct place. After the second pass, the next largest is in place. This process continues and requires $n-1$ passes to sort n items, since the final item must be in place after the $(n-1)$ pass.

Selection Sort Workflow

Array	Description
42 | 16 | 84 | 12 | 77 | 26 | 53	The array, before the selection sort operation begins.
12 | 16 | 64 | **42** | 77 | 26 | 53	The smallest number (**12**) is swapped into the first element in the structure.
12 | **16** | 84 | 42 | 77 | 26 | 53	In the data that remains, **16** is the smallest; and it does not need to be moved.
12 | 16 | **26** | 42 | 77 | **84** | 53	**26** is the next smallest number, and it is swapped into the third position.
12 | 16 | 26 | **42** | 77 | 84 | 53	**42** is the next smallest number; it is already in the correct position.
12 | 16 | 26 | 42 | **53** | 84 | **77**	**53** is the smallest number in the data that remains; and it is swapped to the appropriate position.
12 | 16 | 26 | 42 | 53 | **77** | **84**	Of the two remaining data items, **77** is the smaller; the items are swapped. *The selection sort is now complete.*



Example (Find the Smallest)

```python
def SelectionSort(arr):
    # Get the length of input list
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):

        # Find the minimum element in remaining unsorted list
        index = i
        for j in range(i+1, n):
            if arr[index] > arr[j]:
                index = j

        # Swap the found minimum element with the first element
        arr[i], arr[index] = arr[index], arr[i]


# Sort and Print out the result
InputList = [20, 8, 5, 10, 7]
SelectionSort(InputList)
print ("Sorted List:", InputList)
```

```
Sorted List: [5, 7, 8, 10, 20]
```

20	8	5	10	7

5	8	20	10	7

5	7	20	10	8

5	7	8	10	20

5	7	8	10	20

Example (Find the Largest)

```
def SelectionSort(arr):
    # Get the length of input list
    n = len(arr)

    # Traverse through all array elements
    for i in range(n-1, 0, -1):

        # Find the maximum element in remaining unsorted list
        index = 0
        for j in range(1, i+1):
            if arr[j] > arr[index]:
                index = j

        # Swap the found maximum element with the last element
        arr[i], arr[index] = arr[index], arr[i]


# Sort and Print out the result
InputList = [10, 20, 31, 5, 12]
SelectionSort(InputList)
print ("Sorted List:", InputList)
```

```
Sorted List: [5, 10, 12, 20, 31]
```

10	20	31	5	12

10	20	12	5	31

10	5	12	20	31

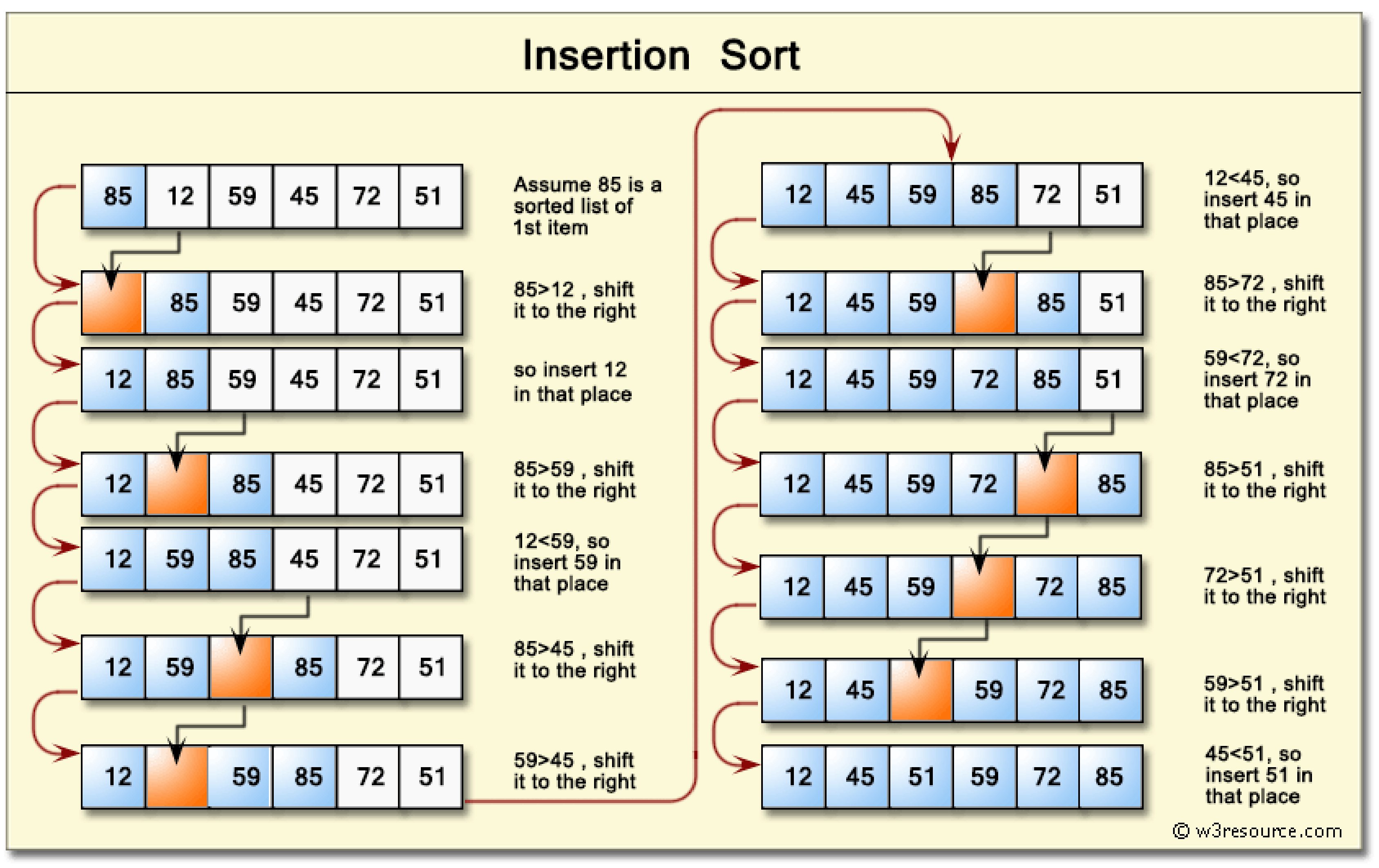
10	5	12	20	31

5	10	12	20	31

Insertion Sort

- The insertion sort always maintains a sorted sublist in the lower positions of the list. Each new item is then “inserted” back into the previous sublist such that the sorted sublist is one item larger.
- We begin by assuming that a list with one item (position 0) is already sorted. On each pass, one for each item 1 through $n-1$, the current item is checked against those in the already sorted sublist. As we look back into the already sorted sublist, we shift those items that are greater to the right. When we reach a smaller item or the end of the sublist, the current item can be inserted

Insertion Sort Workflow



Example

```python
def InsertionSort(arr):
    # Get the length of input list
    n = len(arr)

    # Traverse through all array elements (except first element)
    for i in range(1, n):
        currentvalue = arr[i]
        position = i

        while position > 0 and arr[position-1] > currentvalue:
            arr[position] = arr[position-1]
            position = position - 1

        arr[position] = currentvalue

InputList = [5, 3, 1, 2, 6, 4]
InsertionSort(InputList)
print(InputList)
```

```
[1, 2, 3, 4, 5, 6]
```

[0]	[1]	[2]	[3]	[4]	[5]
5	3	1	2	6	4
3	5	1	2	6	4
1	3	5	2	6	4
1	2	3	5	6	4
1	2	3	4	5	6

Merge Sort

- Merge Sort is a recursive algorithm that continually splits a list in half.
- If the list is empty or has one item, it is sorted by definition. If the list has more than one item, we split the list and recursively invoke a merge sort on both halves.
- Once the two halves are sorted, the fundamental operation, called a merge, is performed.
- Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list.

Merge Sort Workflow

1. Divide the list into two parts
2. Divide the list into two parts again
3. Break each element into single part
4. Sort the element from smallest to largest
5. Merge the divided sorted arrays together
6. The array has been sorted

Example

```python
def MergeSort(alist):
    if len(alist)>1:
        mid = len(alist)//2
        lefthalf = alist[:mid]
        righthalf = alist[mid:]

        MergeSort(lefthalf)
        MergeSort(righthalf)

        i=0
        j=0
        k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1

        while i < len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j < len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1

InputList = [5, 8, 3, 9, 1, 2]
MergeSort(InputList)
print(InputList)
```

```
[1, 2, 3, 5, 8, 9]
```

MergeSort (Arr, 0, 5) | 5 | 8 | 3 | 9 | 1 | 2 |

MergeSort (Arr, 0, 2) | 5 | 8 | 3 |

MergeSort (Arr, 3, 5) | 9 | 1 | 2 |

MergeSort (Arr, 0, 1) | 5 | 8 |

MergeSort (Arr, 2, 2) | 3 |

MergeSort (Arr, 3, 4) | 9 | 1 |

MergeSort (Arr, 5, 5) | 2 |

MergeSort (Arr, 0, 0) | 5 |

MergeSort (Arr, 1, 1) | 8 |

MergeSort (Arr, 3, 3) | 9 |

MergeSort (Arr, 4, 4) | 1 |

Merge (Arr, 0, 0, 1) | 5 | 8 |

Merge (Arr, 3, 3, 4) | 1 | 9 |

Merge (Arr, 0, 2) | 3 | 5 | 8 |

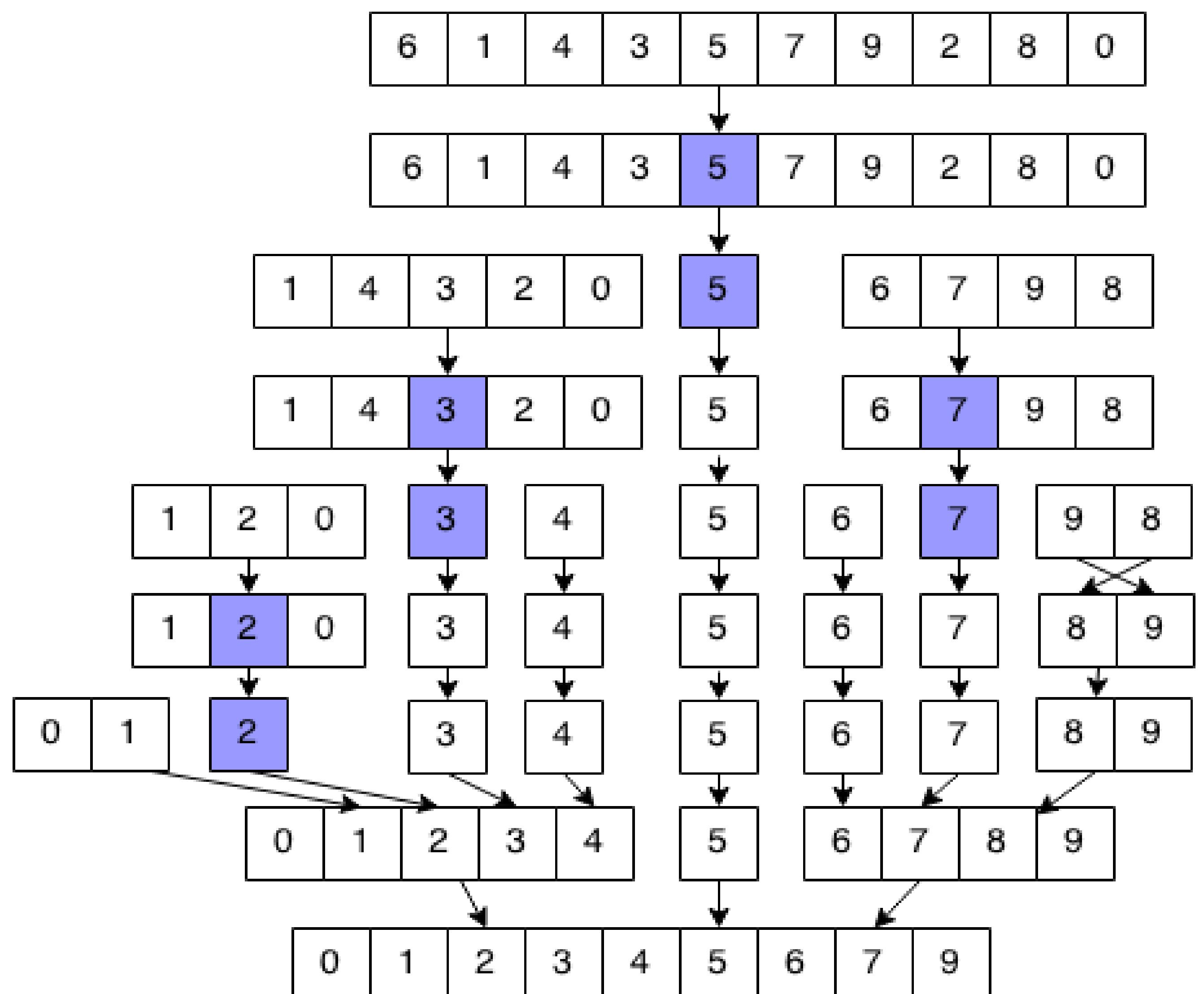
Merge (Arr, 3, 5) | 1 | 2 | 9 |

Merge (Arr, 0, 5) | 1 | 2 | 3 | 5 | 8 | 9 |

Quick Sort

- The Quick Sort algorithm consists of three steps:
  - Divide: Partition the list
    - To partition the list, we first choose a Pivot from the list for which we hope about half the elements will come before and half after.
    - Then we partition the elements so that all those with value less than the pivot come in one sub list and all those with greater values come in another
  - Recursion: Recursively sort the sub lists separately
  - Conquer: Put the sorted sub lists together

Quick Sort Workflow



Example

```
def partition(arr,low,high):
    i = ( low - 1 )         # Index of smaller element
    pivot = arr[high]       # Take last element as pivot

    for j in range(low , high):
        # If current element <= pivot
        if arr[j] <= pivot:
            # increment index of smaller element
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]

    arr[i+1], arr[high] = arr[high], arr[i+1]
    return ( i + 1 )

def QuickSortHelper(arr, low, high):
    if low < high:
        # pi is partitioning index, arr[p] is now at right place
        pi = partition(arr, low, high)

        # Separately sort elements before partition and after partition
        QuickSortHelper(arr, low, pi-1)
        QuickSortHelper(arr, pi+1, high)

def QuickSort(arr):
    QuickSortHelper(arr, 0, len(arr)-1)

InputList = [10, 80, 30, 90, 40, 50, 70]
QuickSort(InputList)
print ("Sorted List:", InputList)
```

```
Sorted List: [10, 30, 40, 50, 70, 80, 90]
```

{10, 80, 30, 90, 40, 50, 70}
Partition around 70 (Last element)
{10, 30, 40, 50}
{90, 80}
Partition around 50
Partition around 80
{10, 30, 40}
{ }
{ }
{90}
Partition around 40
{10, 30}
{ }
Partition around 30
{10}
{ }

Comparison of Sorting Algorithm

	Worst Case	Average Case
Selection Sort	n^2	n^2
Bubble Sort	n^2	n^2
Insertion Sort	n^2	n^2
Merge Sort	n x log n	n x log n
Quick Sort	n^2	n x log n

Python sort() Function

- Python lists have a built-in list.sort() method that modifies the list in-place. There is also a sorted() built-in function that builds a new sorted list from an iterable.

```
Input_List = ['3', '5', '1', '2', '4']

# Sort the list in ascending order
ResultList = sorted(Input_List)

print(ResultList)
```

```
['1', '2', '3', '4', '5']
```

```
Input_List = ['3', '5', '1', '2', '4']

# Sort the list in descending order
ResultList = sorted(Input_List, reverse=True)

print(ResultList)
```

```
['5', '4', '3', '2', '1']
```

Which algorithm does Python sorted() use?

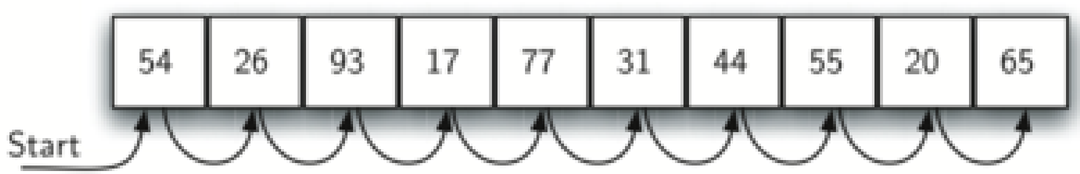
- Timsort has been Python's standard sorting algorithm since version 2.3.
- Timsort is a hybrid sorting algorithm, derived from Merge Sort and Insertion Sort, designed to perform well on many kinds of real-world data.
- It was invented by Tim Peters in 2002 for use in the Python programming language.
- The algorithm finds subsets of the data that are already ordered, and uses the subsets to sort the data more efficiently. This is done by merging an identified subset, called a run, with existing runs until certain criteria are fulfilled.

Searching Algorithms

Sequential Search, Binary Search

Sequential Search

- Starting at the first item in the list, we simply move from item to item, following the underlying sequential ordering until we either find what we are looking for or run out of items.
- If we run out of items, we have discovered that the item we were searching for was not present



Example

```python
def SequentialSearch(Input_List, Search_Item):
    # Define the found flag
    found = False

    # Travel through the list
    for i in Input_List:
        # Stop if item found
        if i == Search_Item:
            found = True
            break

    # Return result
    return found

InputList = [5, 9, 10, 2, 90, 4]
print(SequentialSearch(InputList, 2))
```

Linear Search for "2" in 6 elements array

Given Array

5	9	10	2	90	4	5 != 2
0	1	2	3	4	5	

5	9	10	2	90	4	9 != 2
0	1	2	3	4	5	

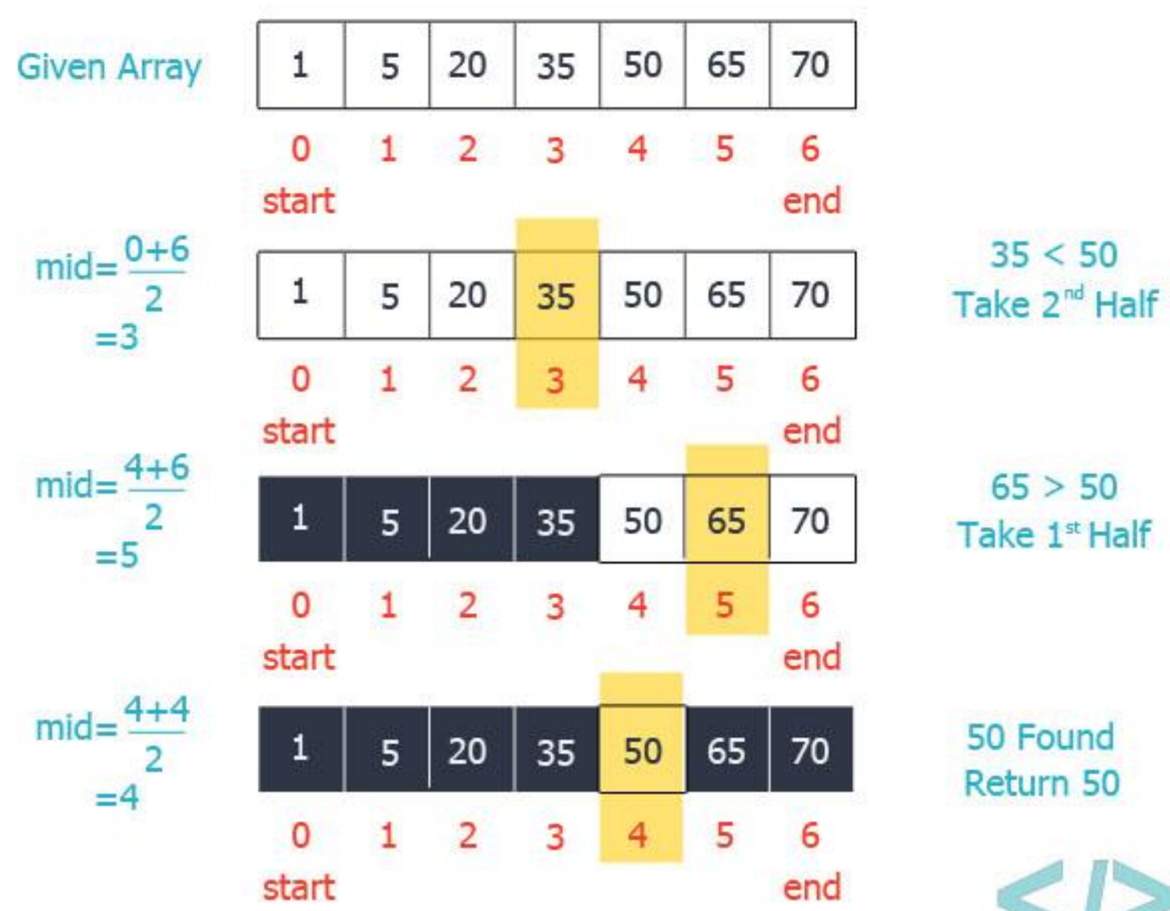
5	9	10	2	90	4	10 != 2
0	1	2	3	4	5	

5	9	10	2	90	4	2 found at index 3.
0	1	2	3	4	5	

Binary Search

- Instead of searching the list in sequence, a binary search will start by examining the middle item.
- If that item is the one we are searching for, we are done. If it is not the correct item, we can use the ordered nature of the list to eliminate half of the remaining items.
- If the item we are searching for is greater than the middle item, we know that the entire lower half of the list as well as the middle item can be eliminated from further consideration. The item, if it is in the list, must be in the upper half.
- We can then repeat the process with the upper half. Start at the middle item and compare it against what we are looking for. Again, we either find it or split the list in half, therefore eliminating another large part of our possible search space.

Binary Search Workflow



Example

```python
def BinarySearch(Input_List, Search_Item):
    first = 0
    last = len(Input_List)-1
    found = False

    while first<=last and not found:
        midpoint = (first + last)//2
        if Input_List[midpoint] == Search_Item:
            found = True
        else:
            if Search_Item < Input_List[midpoint]:
                last = midpoint-1
            else:
                first = midpoint+1

    # Return result
    return found

InputList = [1, 2, 5, 6, 7, 9, 11, 13]
print(BinarySearch(InputList, 9))
```

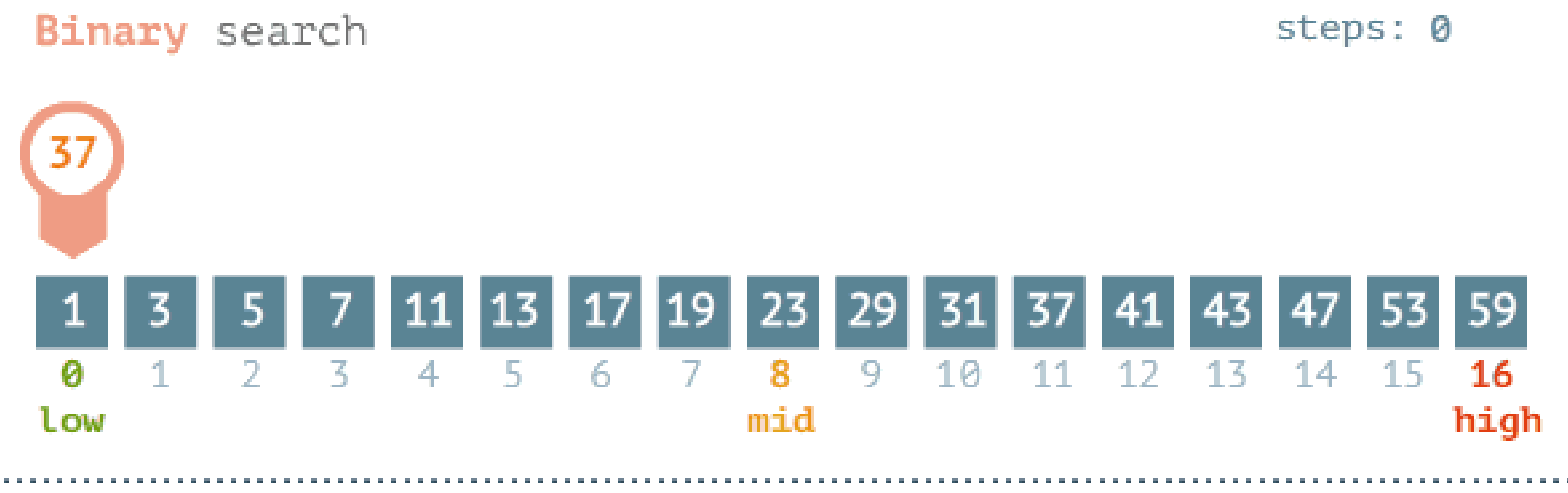
```
True
```

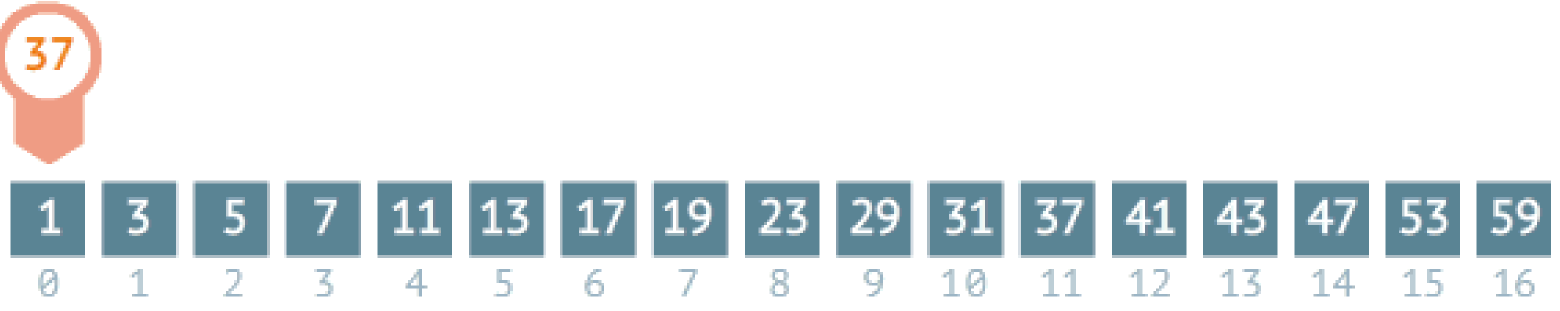
1 2 5 6 7 9 11 13
Step 1:
9 > 7 (choose right)

7 9 11 13
Step 2:
9 < 11 (choose left)

7 9
Step 3:
9 = 9 (key located)

Comparison





www.penjee.com