NANYANG TECHNOLOGICAL UNIVERSITY

Data Structures & Algorithms in Python: Summary

Dr. Owen Noel Newton Fernando

College of Computing and Data Science

Overview of SC1007

- **Data Structures**:
  - Concepts of data structures
  - Introduce some classical data structures
    - Linear: Linked list, stack, queue, priority queue
    - Nonlinear: Binary tress and binary search trees
  - Implement these data structures using python
- **Algorithms**:
  - Analysis of Algorithm time complexity and space complexity
  - Introduce to some typical algorithms and their applications
  - Introduce to some algorithm design strategies
  - Implement these algorithms using python

Learning Outcome

- Understand and use data structures such as linked lists, stacks, queues, trees and binary search teess.
- Choose suitable data structures in solving real-world problems, optimizing performance in computational challenges.
- Analyse the time and space complexity of algorithms to evaluate the efficiency of algorithmic solutions.
- Use suitable searching techniques, such as sequential search, binary search, search using hash tables and string search using Tries, to real-world applications requiring fast and efficient data retrieval.

Introduction to Memory Management

Introduction to Memory Management

- Memory management refers to the critical process of allocating, deallocating, and coordinating computer memory effectively.
- The primary objective of memory management is to ensure that all processes execute smoothly and efficiently utilize system resources.
- In Python, this is achieved through a combination of dynamic memory allocation, automatic garbage collection, and reference counting mechanisms.

Programs need memory to run
Memory is a limited resource

Introduction to Memory Management

- Memory management in Python is handled automatically
- Multiple mechanisms work together to manage memory efficiently
- Understanding these concepts helps write better Python code
- Core Memory Management Mechanisms
  - Reference Counting
  - Garbage Collection
  - Memory Pooling
  - Memory Interning
  - Manual Memory Management

Object-Oriented Programming

Object-Oriented Programming

- A programming paradigm based on the concept of "**objects**" that **encapsulate data and behavior**.
- **OOPs is a way of organizing code that uses objects and classes to represent real-world entities and their behavior**
- In OOPs, **object has attributes that have specific data and can perform certain actions using methods.**

Objects, Classes and Methods

Objects:

- Everything in Python is an object.
- Objects can be created, manipulated, and destroyed (e.g., using del or garbage collection).

Classes:

- Classes are **blueprints** for creating objects.
- **A class defines attributes (data) and methods (functions).**
- Example: A Car class defines attributes (color, speed) and methods (drive, brake).
- Each car is an instance of the class.

Methods

- A method is a procedural attribute, like a function that works only with this class.
- Python always passes the object as the first argument. The convention is to use self as the name of the first argument of all methods.

Linked List

Linked List

A LinkedList consists of nodes where each node has data and a pointer to the next node, ending with None. It's represented as sequential nodes connected by pointers in memory.

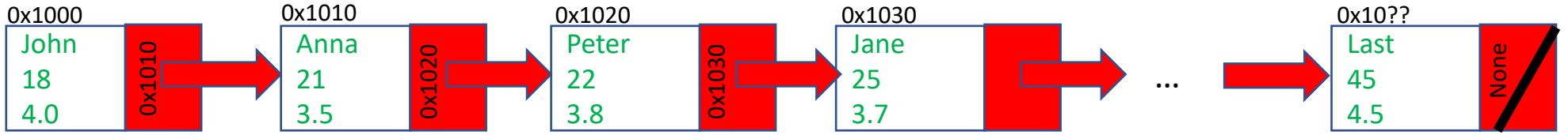
Key Operations:
- **Insert**: Add elements at start or anywhere
- **Delete**: Remove elements from any position
- **Search**: Find elements by traversing through nodes

Types
- Singly Linked List (one direction).
- Doubly Linked List (both directions).
- Circular Linked List (last node points to the first)

Benefits:
Dynamic memory allocation, efficient front insertions, flexible size management without wastage

Use Cases
Implementing stacks, queues, music playlists, browser history navigation

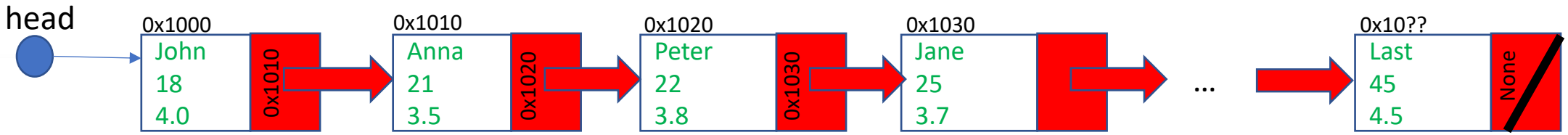
Implementation of a linked list in Python



- A linked list class is used to create and manage a list of nodes.
- The head node is essential to locate the first node in the list.
- Additional pointers like a tail node can improve efficiency by pointing to the last node.
- The class supports operations such as insertion, deletion, and traversal to manage the list effectively.

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

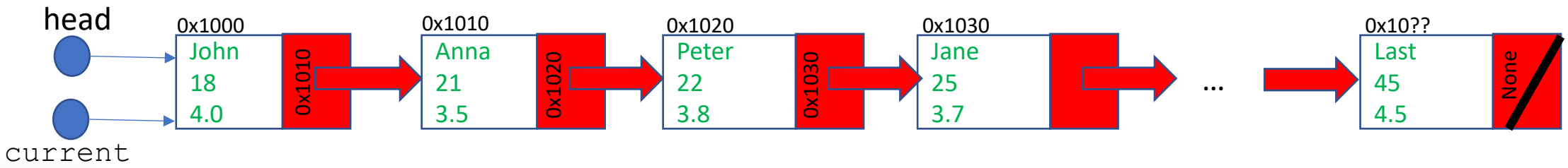
The Linked List



- Just introduce a new member in the linked list class, *size*
- Initialize *size* as zero
- When you add or remove a node, increase or decrease *size* by one accordingly

```
1  class ListNode:
2      def __init__(self, item):
3          self.item = item
4          self.next = None
5
6  class LinkedList:
7      def __init__(self):
8          self.head = None
9          self.size = 0
```

Display each element in the linked list



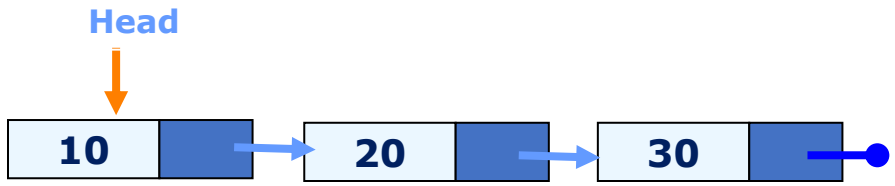

```
1def display(self):
2    current = self.head
3    while current:
4         print(current.data, end=" -> ")
5         current = current.next
6     print("None")
```

Display each element in the linked list

- Given the head pointer of the linked list
- Print all items in the linked list
- From first node to the last node

Display each element in the linked list

```
1  class Node:
2      def __init__(self, data):
3          self.data = data
4          self.next = None
5
6  class LinkedList:
7      def __init__(self):
8          self.head = None
9           self.size = 0
10
11 # Standalone function taking head as parameter
12 def display(head):
13     current = head
14     while current:
15          print(current.data, end=" -> ")
16          current = current.next
17     print("None")
```




```
18 if __name__ == "__main__":
19
20     # Initialize empty linked list object
21     linked_list = LinkedList()
22
23     # Create nodes
24     node1 = Node(10)
25     node2 = Node(20)
26     node3 = Node(30)
27
28     # Link nodes
29     linked_list.head = node1
30     node1.next = node2
31     node2.next = node3
32
33     # Print the linked list
34     # Passes linked_list.head as argument
35     display(linked_list.head)
```

`linked_list.head` is the reference to the first node (`node1`), which acts as the starting point of the linked list.

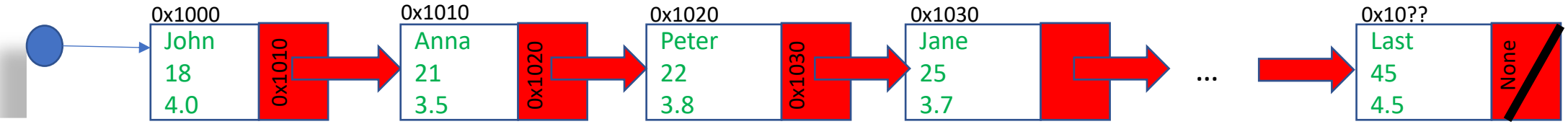
Search the node at index i

```
1   class Node:
2       def __init__(self, data):
3           self.data = data
4           self.next = None
5
6   class LinkedList:
7       def __init__(self):
8           self.head = None
9           self.size = 0
10
11  # Independent function that takes
12  # head and index
13  def findAt(head, index):
14      current = head
15      if not current:
16          return None
17      while index > 0:
18          current = current.next
19          if not current:
20              return None
21          index -= 1
```

```
22  if __name__ == "__main__":
23
24      linked_list = LinkedList()
25
26      node1 = Node(10)
27      node2 = Node(20)
28      node3 = Node(30)
29
30      linked_list.head = node1
31      node1.next = node2
32      node2.next = node3
33
34      # Find node at index 2
35      found_node = findAt(linked_list.head, 2)
36      if found_node:
37          print(f"Node at index 2: {found_node.data}")
38      else:
39          print("Index not found")
```

`linked_list.head` is the reference to the first node (`node1`), which acts as the starting point of the linked list.

SizeList




```
1   class ListNode:
2       def __init__(self, item):
3           self.item = item
4           self.next = None
5
6
7   class LinkedList:
8       def __init__(self):
9           self.head = None
10          self.size = 0
11
12
13  def sizeList(ll): # ll = LinkedList() #
14    return ll.size
```

Insert node at given index

```
1 def insertNode2(ll, index, item):
2     newNode = ListNode(item)
3
4
5     # Case 1: Inserting at the beginning
6     if index == 0:
7         newNode.next = ll.head
8         ll.head = newNode
9         ll.size += 1
10        return True
11
12
13    # Case 2: Inserting anywhere else
14    pre = findNode2(ll, index - 1)
15
16
17    if pre is not None:
18        newNode.next = pre.next
19        pre.next = newNode
20        ll.size += 1
21        return True
22
23
24    return False
```

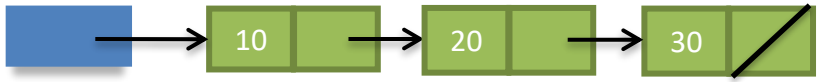
Remove node at given index

```
1 def remove_node(ll, index):
2     if ll is None or index < 0 or index >= ll.size:
3         return -1
4
5
6     if index == 0:
7         cur = ll.head.next
8         del ll.head
9         ll.head = cur
10        ll.size -= 1
11        return 0
12
13
14    pre = find_node(ll, index - 1)
15    if pre is not None:
16        if pre.next is None:
17            return -1
18
19
20        cur = pre.next
21        pre.next = cur.next
22        del cur
23        ll.size -= 1
24        return 0
25    return -1
26
```

Doubly Linked List

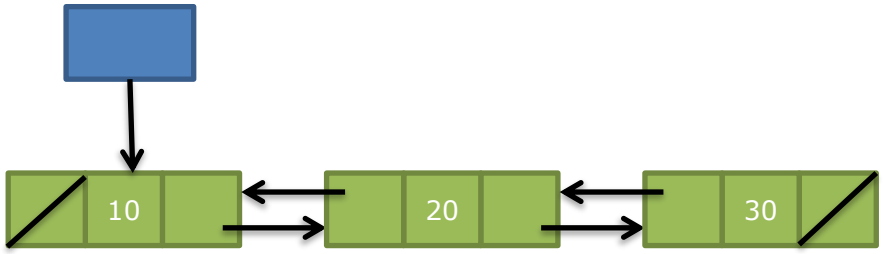
Singly Linked list: Only one link. Traversal of the list is one way only.

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



Doubly Linked List: two links in each node. It can search forward and backward.

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

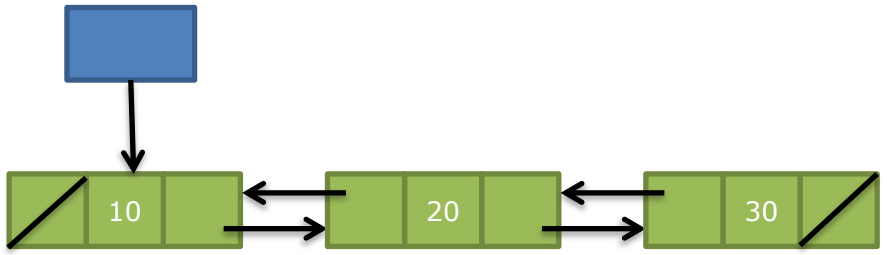


Doubly Linked List: Print

Print is similar to the Singly Linked List

```
1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.next = None
5         self.pre = None
```

```
1 def print_list(head):
2     current = head
3     while current:
4         print(current.data, end=" -> ")
5         current = current.next
6     print("None")
```

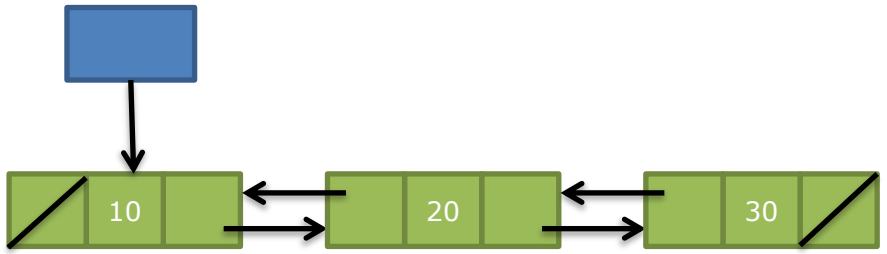


Doubly Linked List: Search

Display is similar to the Singly Linked List's

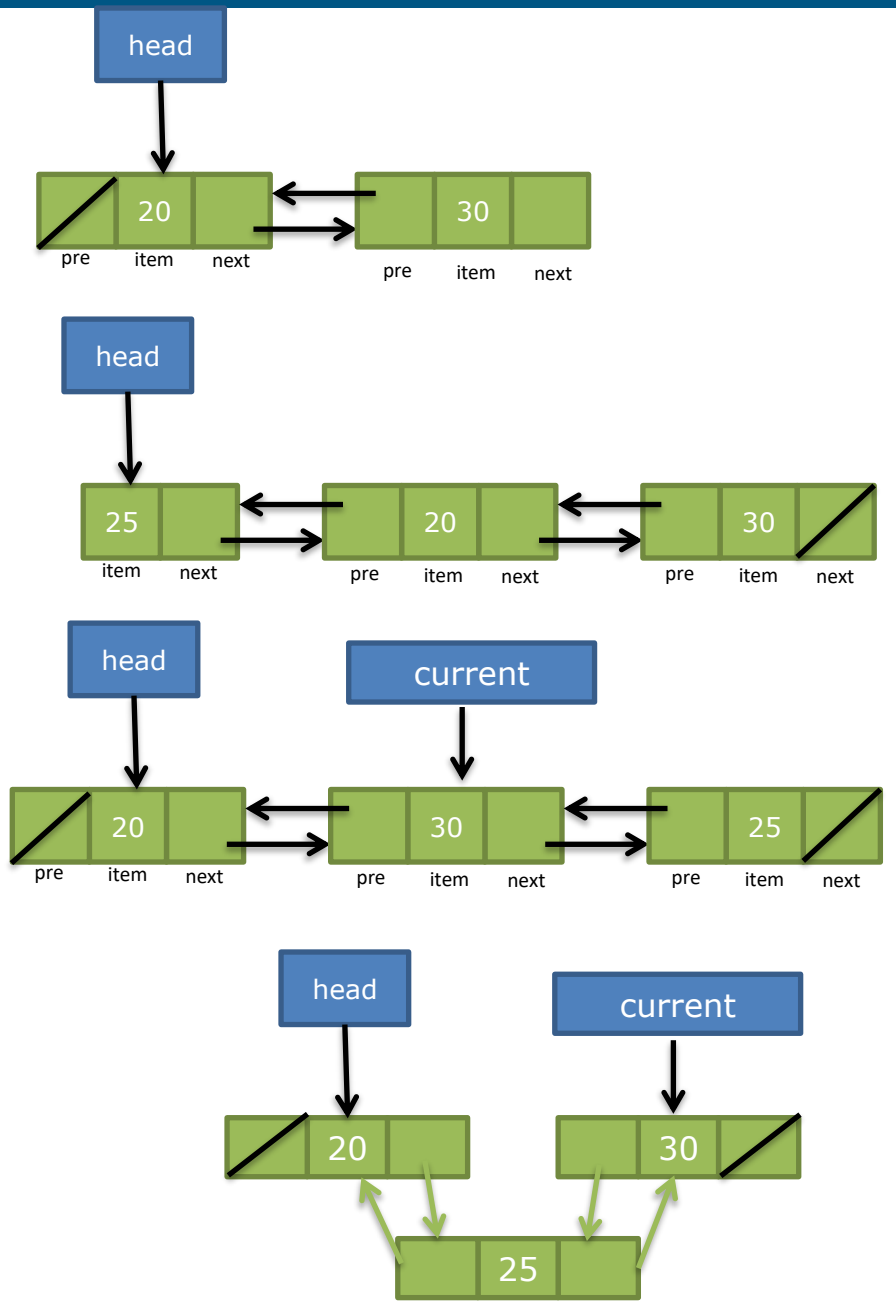
```
1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.next = None
5         self.pre = None
```

```
1 def search(self, data):
2         current = self.head
3         while current:
4             if current.data == data:
5                 return True
6             current = current.next
7         return False
```



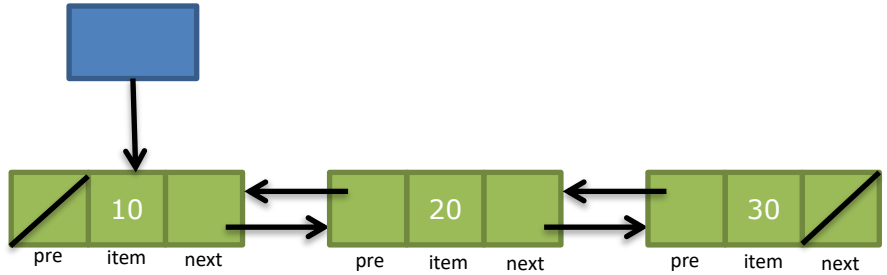
Doubly Linked List: Insertion (Index)

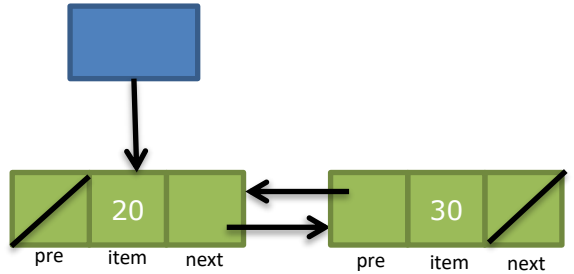
```
1   def insert_at(self, index, data):
2       # If index is invalid
3       if index < 0 or index > self.size:
4           raise ValueError("Invalid position")
5
6       # Create new node
7       new_node = Node(data)
8
9       # If inserting at beginning
10      if index == 0:
11          new_node.next = self.head
12          if self.head:
13              self.head.prev = new_node
14          self.head = new_node
15
16      # Inserting at middle or end
17      else:
18          current = self.head
19          # Traverse to position
20          for i in range(index-1):
21              current = current.next
22
23          # Link new node
24          new_node.prev = current
25          new_node.next = current.next
26          if current.next:
27              current.next.prev = new_node
28          current.next = new_node
28
30      self.size += 1
```

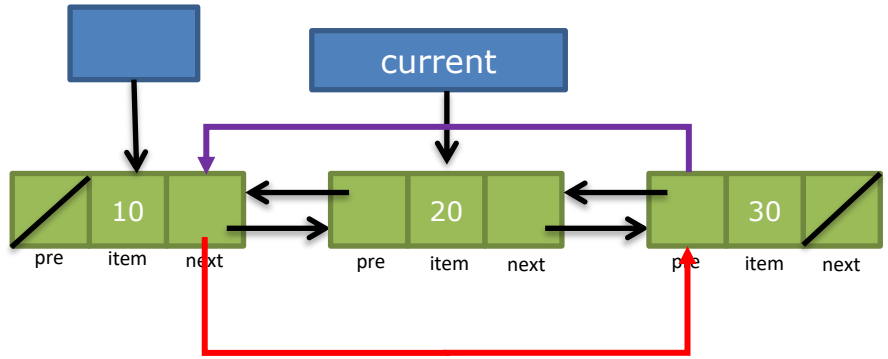


Doubly Linked List: Deletion (Index)

```
1     remove_at(self, index):
2       # Case 1: Check if the list is empty
3       if self.head is None:
4           print("List is empty")
5           return False
6       # Case 2: Validate index
7       if index < 0 or index >= self.size:
8           print("Invalid index")
9           return False
10      # Case 3: Remove the first node (index 0)
11      if index == 0:
12          self.head = self.head.next
13          if self.head:  # If the list is not empty after removal
14              self.head.prev = None
15          self.size -= 1
16          return True
17      # Case 4: Remove from the middle or end
18      current = self.head
19       for i in range(index): # Traverse to the node at the given index
20          current = current.next
21      # Update pointers to remove the node
22      current.prev.next = current.next
23      if current.next:  # If it's not the last node
24          current.next.prev = current.prev
25      self.size -= 1
26      return True
```

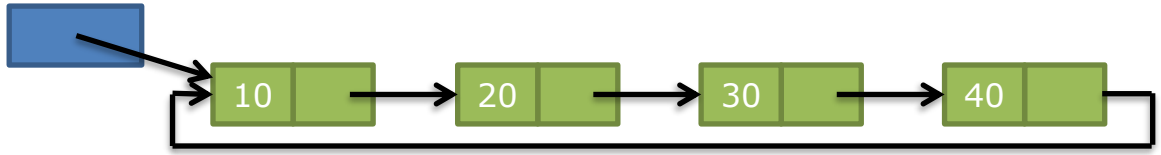




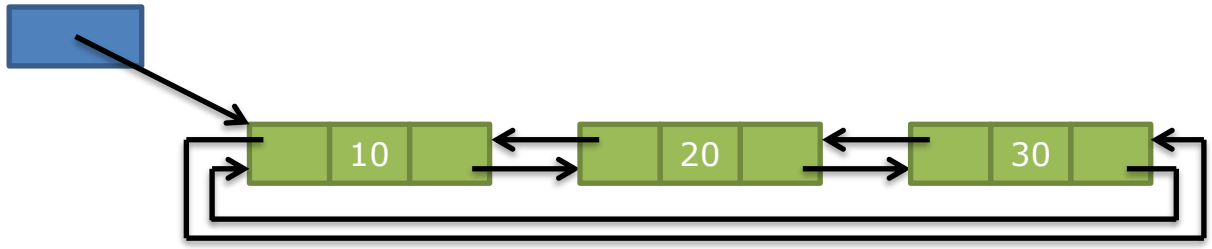


Circular Linked List

- Circular singly linked lists
  - Last node has next pointer pointing to first node



- Circular doubly linked lists
  - Last node has next pointer pointing to first node
  - First node has pre pointer pointing to last node

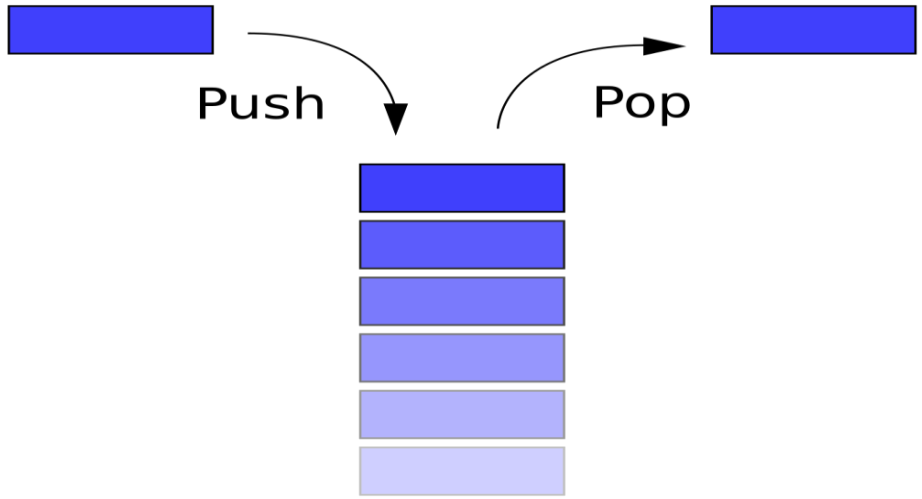


- **Display, Search, Size**: the last node's link is equal to head instead of **None**. The stop criteria needs to change.
- **Insert** and **Delete**: there is no special case at first or last position. The head node may need to update if the first node is affected.

Stacks

What is a stack?

- A *stack* is a Last In, First Out (LIFO) data structure
- Anything added to the stack goes on the "top" of the stack
- Anything removed from the stack is taken from the "top" of the stack
- Things are removed in the reverse order from that in which they were inserted
- Can be implemented by list or linked list



Implementing a Stack class using Linked List

```
class Node:
    def __init__(self, data):
        self.data = data     # stores the value
        self.next = None     # points to next node
```

```
class Stack:
    def __init__(self):
        self.top = None        # points to top node
        self.size = 0          # tracks number of nodes
```

Implementing a Stack class using Linked List

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

```
class LinkedList:
    def __init__(self):
        self.size = 0
        self.head = None
```

```
class Stack:
    def __init__(self):
        self.ll = LinkedList()
```

Core stack operations

`peek()`: Returns the top element without removing it

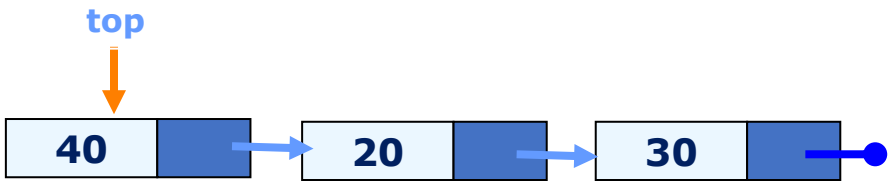
`is_empty()`: Checks if the stack is empty

`get_size()`: Returns the current size of the stack

`push()`: Adds an element to the top of the stack

`pop()`: Removes and returns the top element

Core stack operations: isEmpty()



Version 01

```
def is_empty(self):
    return self.top is None
```

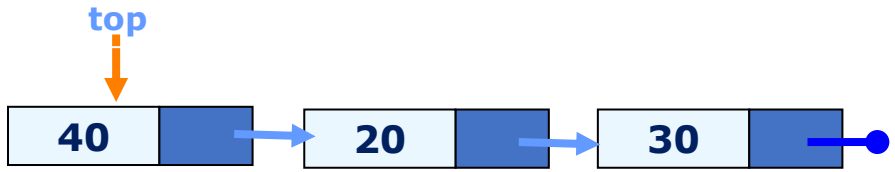
Version 02: Linked List version

```
def is_empty(self):
    return self.ll.size == 0
```

Core stack operations: get_size()

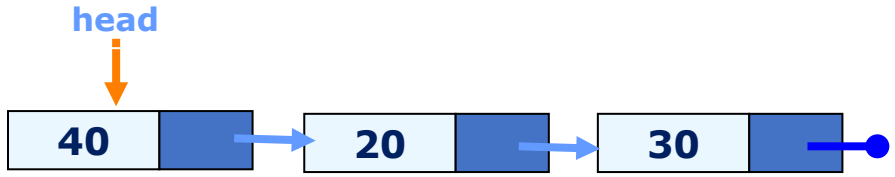
Version 01

```
def get_size(self):
    return self.size
```

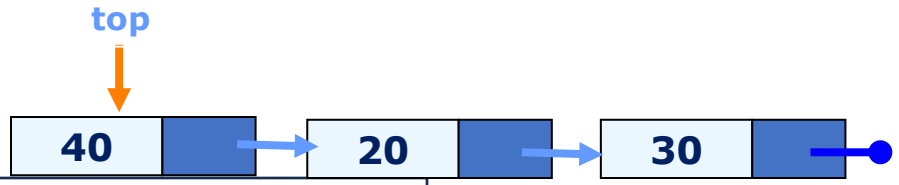


Version 02: Linked List version

```
def get_size(self):
    return self.ll.size
```

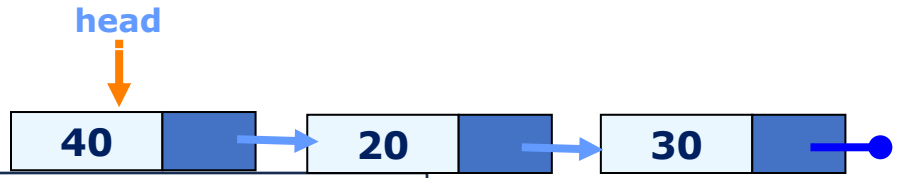


Core stack operations: push()



Version 01

```
def push(self, data):
    new_node = Node(data)
    new_node.next = self.top
    self.top = new_node
    self.size += 1
```



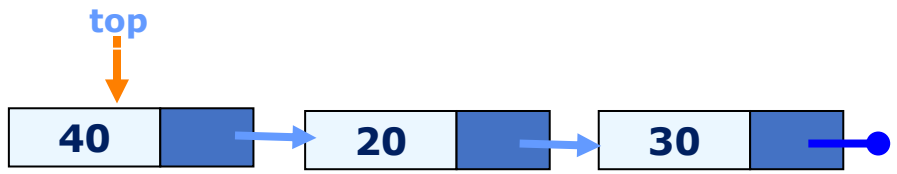
Version 02: Linked List version

```
def push(self, data):
        self.ll.insert_node(0, data)
```

Core stack operations: pop()

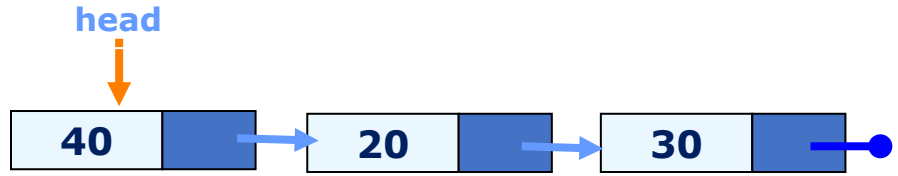
Previous version

```
def pop(self):
    if self.is_empty():
        raise IndexError("Pop from empty stack")
    popped_node = self.top
    self.top = self.top.next
    self.size -= 1
    return popped_node.data
```



New version

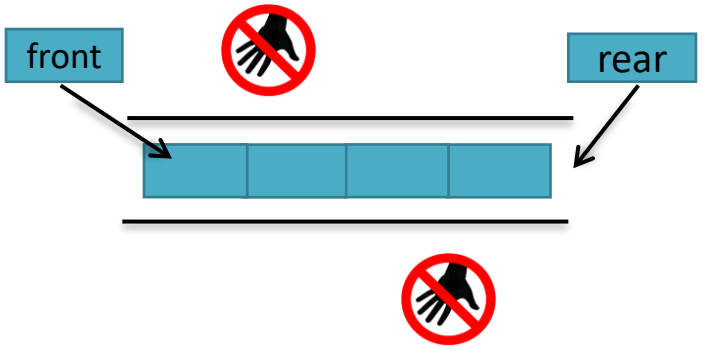
```
def pop(self):
    if self.isEmpty():
        raise IndexError("Pop from empty stack")
    data = self.ll.head.data
    self.ll.remove_node(0)
    return data
```



Queues

Queues

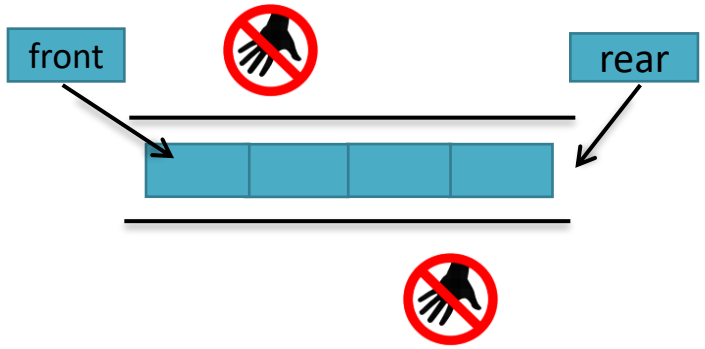
- Elements are added only at the rear and removed from the front
- A queue is a first in, first out (FIFO) data structure
  - Items are removed from a queue in the same order as they were inserted
- Can be implemented by list or linked list



Implementing a Queue using Linked List

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

```
class Queue:
    def __init__(self):
        self.front = None
        self.rear = None
        self.size = 0
```



Due to algorithmic efficiency, a rear pointer is introduced to optimize enqueue operations, allowing new elements to be added efficiently without requiring traversal.

Implementing a Queue class using Linked List

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

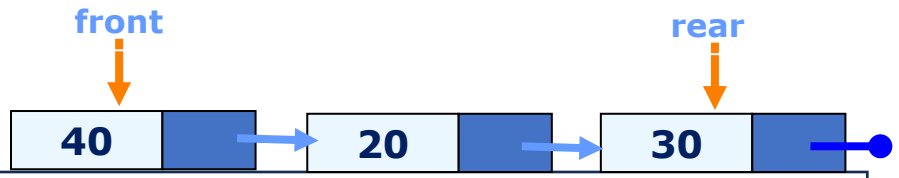
```
class LinkedList:
    def __init__(self):
        self.size = 0
        self.head = None
        self.tail = None
```

```
class Queue:
    def __init__(self):
        self.ll = LinkedList()
```

Core Queue operations

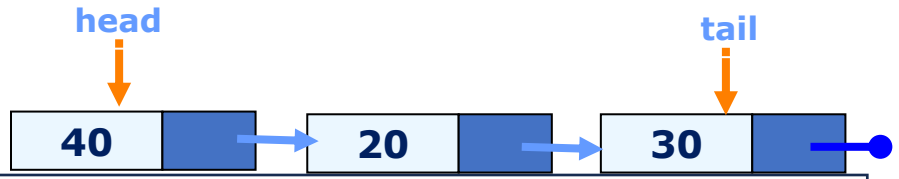
- `getFront()`: Inspect the item at the front of the queue without removing it
- `isEmpty()`: Check if the queue has no more items remaining
- `getSize()`: Returns the current size of the stack
- `enqueue()`: Add an item at the end of the queue
- `dequeue()`: Remove an item from the top of the queue

Core Queue operations: getFront():



Version 01

```
    def getFront(self):
            if self.isEmpty():
                raise IndexError("Peek from empty queue")
            return self.front.data
```



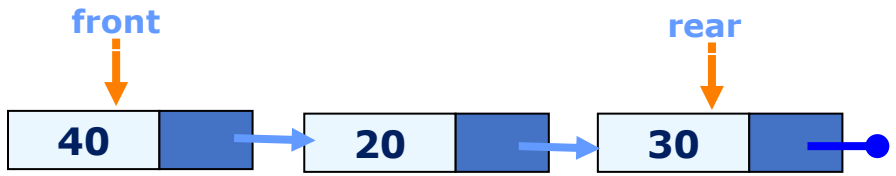
Version 02: Linked List version

```
def getFront(self):
    if self.isEmpty():
        raise IndexError("Peek from empty stack")
    return self.ll.head.data
```

Core stack operations: isEmpty()

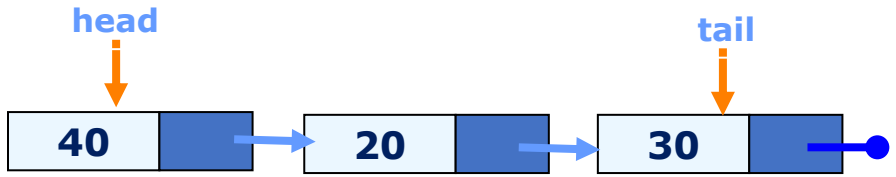
Version 01

```
def isEmpty(self):
    return self.front is None
```



Version 02: Linked List Version

```
def isEmpty(self):
    return self.ll.size == 0
```



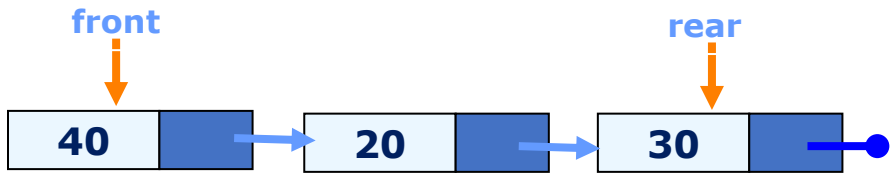
Core stack operations: getSize()

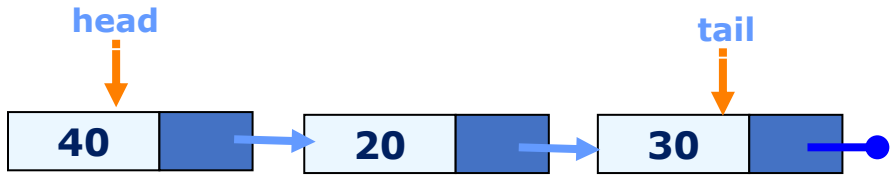
Version 01

```
def getSize(self):
        return self.size
```

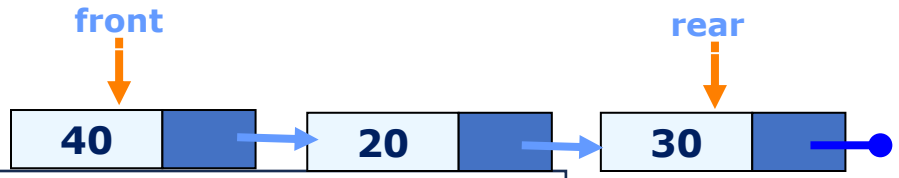
Version 02: Linked List Version

```
def getSize(self):
        return self.ll.size
```



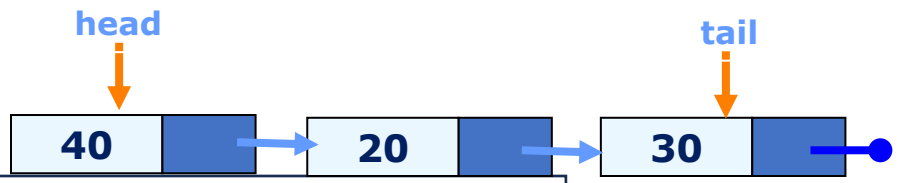


Core stack operations: enqueue()



Version 01

```
def enqueue(self, data):
        new_node = Node(data)
        if self.isEmpty():
            self.front = new_node
        else:
            self.rear.next = new_node
        self.rear = new_node
        self.size += 1
```

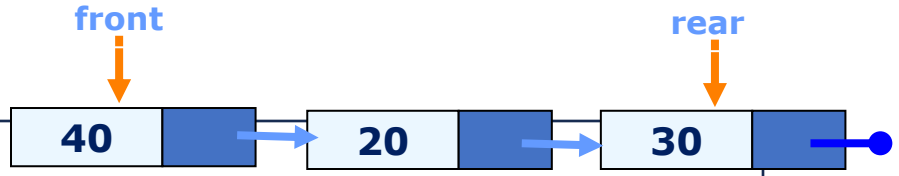


Version 02: Linked List Version

```
def enqueue(self, data):
        self.ll.insert_node(self.ll.size, data)
```

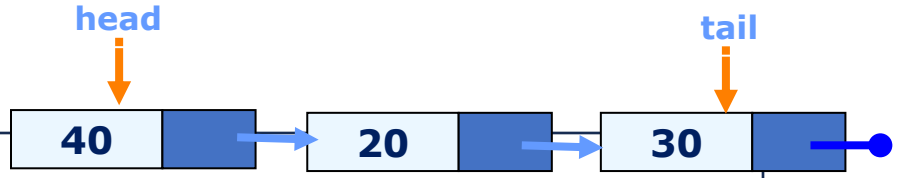
Core stack operations: dequeue()

Version 01




```
def dequeue(self):
        if self.isEmpty():
            raise IndexError("Dequeue from empty queue")
        dequeued_node = self.front
        self.front = self.front.next
        if self.front is None:
            self.rear = None
        self.size -= 1
        return dequeued_node.data
```

Version 02: Linked List version




```
def dequeue(self):
        if self.isEmpty():
            raise IndexError("Dequeue from empty queue")
        data = self.ll.head.data
        self.ll.remove_node(0)
        return data
```

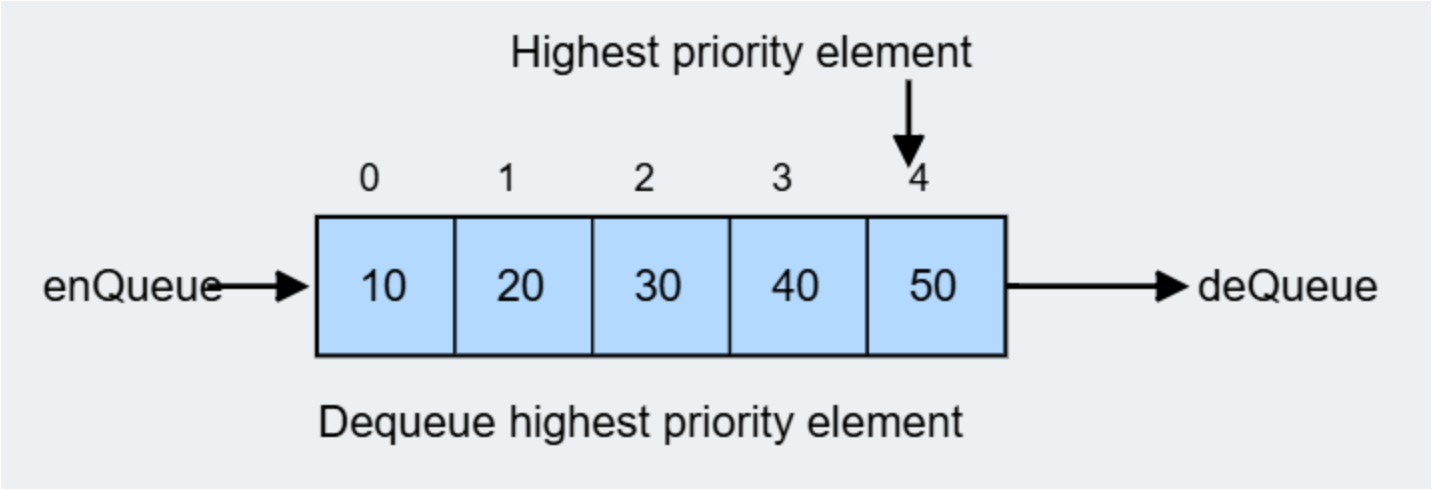
Priority Queue

Priority Queue

- **Definition:**
  - The **priority queue** is **an abstract data-type similar to a regular queue or stack** data structure in which each element additionally has a "**priority**" associated with it. In a priority queue, an element with high priority is served before an element with low priority.
  - A **priority queue ADT** is a data structure that supports the operations Insert and Delete Min (which returns and removes the minimum element) or Delete Max (which returns and removes the maximum element) and peeking the element(find min or find max).

Priority Queue

- **Difference between Priority Queue and Normal Queue:**
  - In a queue, the **first-in-first-out rule** is implemented whereas, in a priority queue, the values are removed **on the basis of priority**. The element with the highest priority is removed first.



Characteristics of a Priority Queue

- **A queue is termed as a priority queue if it has the following characteristics:**
  - Each item has some **priority** associated with it.
  - An item with the **highest priority** is moved at the **front** and **process first (e.g. delete)**.
  - If two elements share the **same priority value**, then the priority queue follows the **first-in-first-out principle** for dequeue operation.

Characteristics of a Priority Queue

- **A queue is termed as a priority queue if it has the following characteristics:**
  - Each item has some priority associated with it.
  - An item with the highest priority is moved at the front and deleted first.
  - If two elements share the same priority value, then the priority queue follows the first-in-first-out principle for dequeue operation.
- **Implementation of the Priority Queue in Data Structure:**
  - You can implement the priority queues in one of the following ways:
    1. Arrays.
       1. Unordered Array
       2. Ordered Array
    2. **Linked list**
    3. Binary heap
       1. Min heap
       2. Max heap
    4. Binary search tree

Implementing Priority Queue using Linked List

```
class PriorityNode:
    def __init__(self, data, priority):
        self.data = data
        self.priority = priority
        self.next = None
```

```
class PriorityQueue:
    def __init__(self):
        self.head = None
        self.size = 0
```

Stack and Queues: Applications

Stack and Queues: Applications

Stack

- Balanced Parentheses Problem (Lab)
- Algebraic expression conversion (infix, prefix and postfix)
- Recursive functions to Iterative functions

Queue

- Palindromes (Lab)
- Scheduling in multitasking, network, job, mailbox etc.

Application: Using a Stack to Ensure Brackets and Parentheses Are Balanced

- ([ ] ({ ( ) } [( )])) is balanced;
  ([ ] ({ ( ) } [( ))]) is not

- Simple counting is not enough to check the balance

- You can do it with a stack: going left to right,
  - If you see a **(**, **[**, or **{**, **push** it on the stack
  - If you see a **)**, **]**, or **}**, **pop** the stack and check whether you popped the corresponding (, [, or {
  - When you reach the end, check that the **stack is empty**

Infix

Operators	Precedence
*, /, %	Highest
+, -	
<<, >>	
&&	
=	Lowest

- While writing an arithmetic expression using **Infix** notation, the operator is placed between the operands.
  - For example, *A+B;* here, plus operator is placed between the two operands A and B.
- **A * (B + C) / D** means: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."
- Information is needed about operator precedence, associativity rules, and brackets which overrides these rules.
- Although it is easy to write expressions using infix notation, computers find it difficult to parse as they need a lot of information to evaluate the expression.

Postfix Notation

- Postfix notation which is better known as Reverse Polish Notation or RPN.
- In Postfix notation, the operator is placed after the operands. For example, if an expression is written as *A+B* in **Infix** notation, the same expression can be written as *AB+* in **Postfix** notation.
- A postfix operation does not follow the rules of **operator precedence**. The operator which occurs first in the expression is operated first on the operands.
- For example, given a postfix notation **AB+C***. While evaluation, addition will be performed prior to multiplication.
- The order of evaluation of a postfix expression is always from left to right.

Prefix Notation

- In a **Prefix notation**, the operator is placed before the operands.
- For example, if A+B is an expression in **Infix notation**, then the corresponding expression in prefix notation is given by +AB.
- While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator.
- Prefix expressions also do not follow the rules of operator precedence and associativity.
- The expression (A + B) * C is written as: *+ABC in the prefix notation
- So, computers work more efficiently with expressions written using Prefix and Postfix notations.

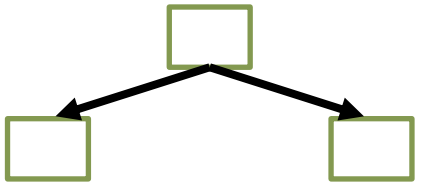
Binary Tree

Binary Tree

- The operations for a binary tree :
  - Insert a node
  - Delete a node
  - Search a node
  - Traversal (display all nodes in a certain order)
    - Inorder
    - Preorder
    - Postorder
    - Level-by-level
    - How do we systematically travel each node once in a tree?

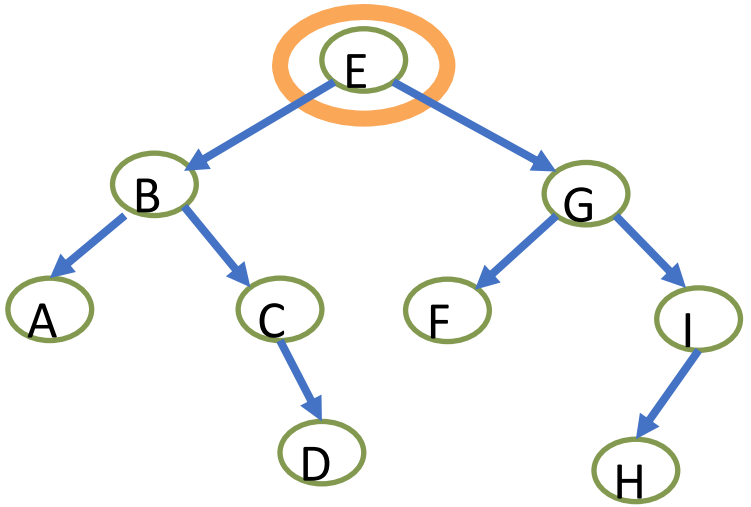
Binary Tree Node Class

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



Pre-order, In-order and Post-order Depth First Traversal

- **Pre-order**
  - **Process the current node's data**
  - Visit the left child subtree
  - Visit the right child subtree
- **In-order**
  - Visit the left child subtree
  - **Process the current node's data**
  - Visit the right child subtree
- **Post-order**
  - Visit the left child subtree
  - Visit the right child subtree
  - **Process the current node's data**



Summary of Depth First Traversal

Pre-Order Traversal

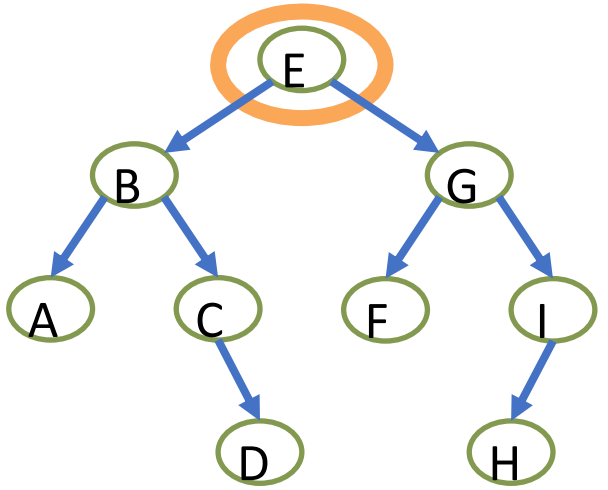
```
E  B  A  C  D  G  F  I  H
```

In-Order Traversal

```
A  B  C  D  E  F  G  H  I
```

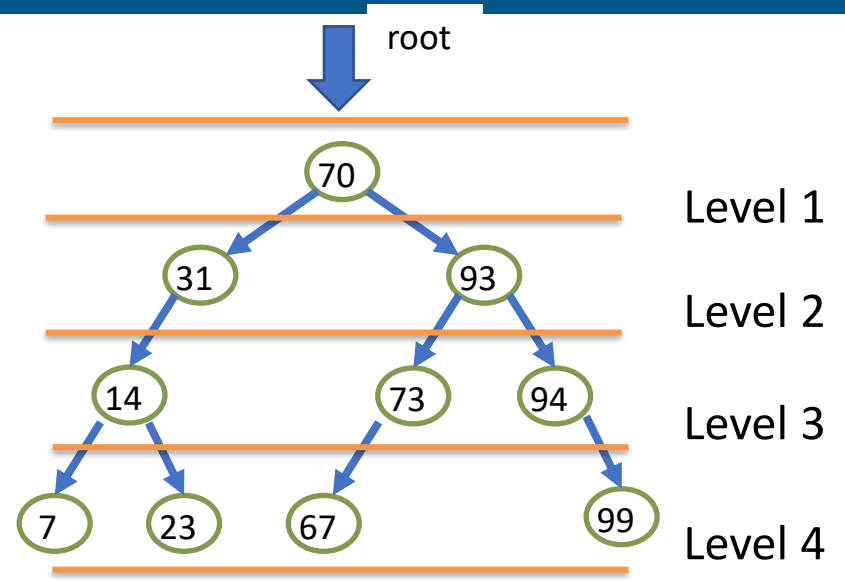
Post-Order Traversal

```
A  D  C  B  F  H  I  G  E
```



Breath-first Traversal: Level-by-level

1. **Initialize the queue:**
   - Create an instance of the Queue class.
   - Add the root node to the queue using **queue.enqueue(self.root).**
2. **Start the traversal loop:**
   - While the queue is not empty:
     - Dequeue the front node using **queue.dequeue()** and print its data.
     - Enqueue the left child if it exists.
     - Enqueue the right child if it exists.
3. **Finish traversal:**
   - Repeat the process until the queue is empty, ensuring level-by-level traversal.




```
def level_order_traversal(self):
    if not self.root:
        return

    queue = Queue()
    queue.enqueue(self.root)

    while not queue.is_empty():
        current_node = queue.dequeue()
        print(current_node.data, end=" ")

        if current_node.left:
            queue.enqueue(current_node.left)
        if current_node.right:
            queue.enqueue(current_node.right)
```

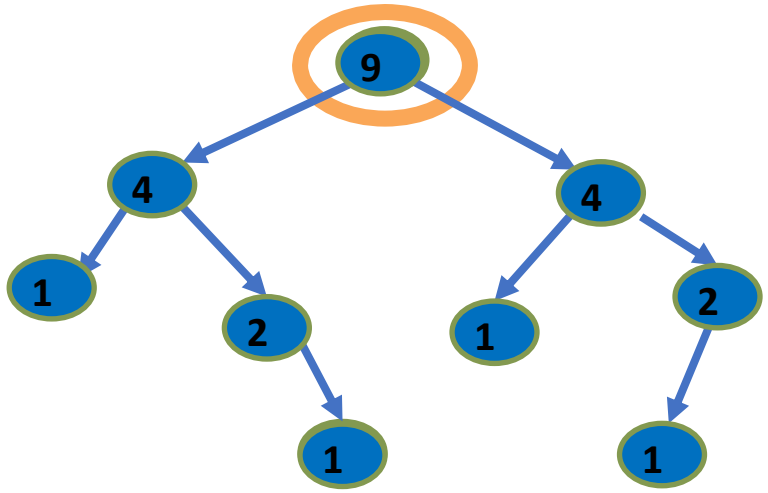
Count Nodes in a Binary Tree (SIZE)

- Recursive definition:
  - Number of nodes in a tree

    = 1

    + number of nodes in left subtree

    + number of nodes in right subtree

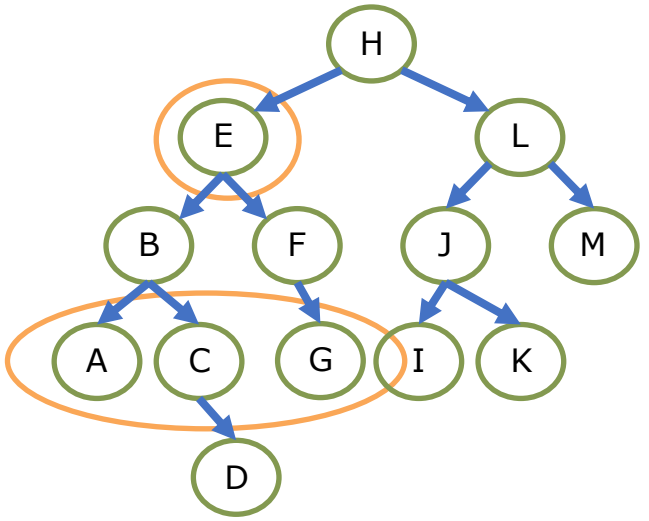


- Each node returns the number of nodes in its subtree

```
def count_nodes(self, node):
    if not node:
        return 0
    return 1 + self.count_nodes(node.left) + self.count_nodes(node.right)
```

Find the Kth-level Descendant Nodes

- Given a node X, find all the descendants of X
- Given node E and K=2, we should return its grandchild nodes A, C, and G
- What if we want to find K^{th}-level descendants?
  - **Need a way to keep track of how many levels down we've gone**



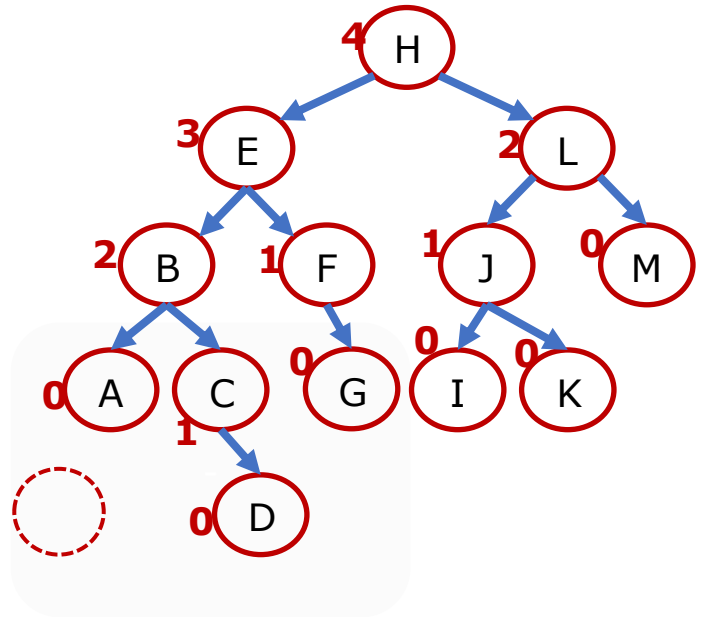

```
    def kth_level_descendants(self, node, k):
        if node is None:
            return []

        if k == 0:
            return [node.data]

        left_descendants = self.kth_level_descendants(node.left, k - 1)
        right_descendants = self.kth_level_descendants(node.right, k - 1)

        return left_descendants + right_descendants
…
```

```
print(bt.kth_level_descendants(bt.root, 2))
```

Height of A Node in A Binary Tree

- The height of a tree: The number of edges on the longest path from the root to a leaf
  - Leaf node returns 0
  - **None node (empty) returns -1**

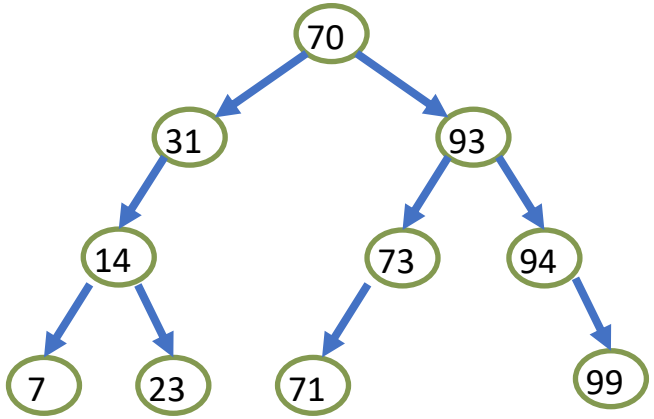
```
def calculate_height(self, node):
        if node is None:
            return -1

        left_height = self.calculate_height(node.left)
        right_height = self.calculate_height(node.right)

        return 1 + max(left_height, right_height)
```

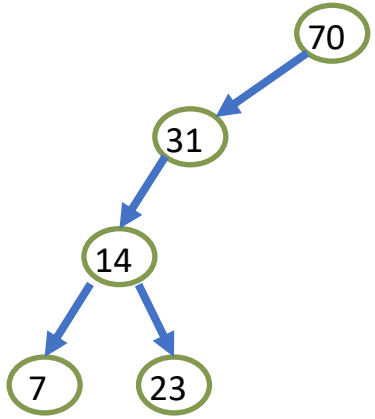


Binary Search Tree

Binary Search Tree

- If the given binary tree is a binary search tree (BST), then each node in the tree must satisfy the following properties:
  - Node's value is greater than all values in its left subtree.
  - Node's value is less than all values in its right subtree.
  - Both subtrees of the node are also binary search trees.

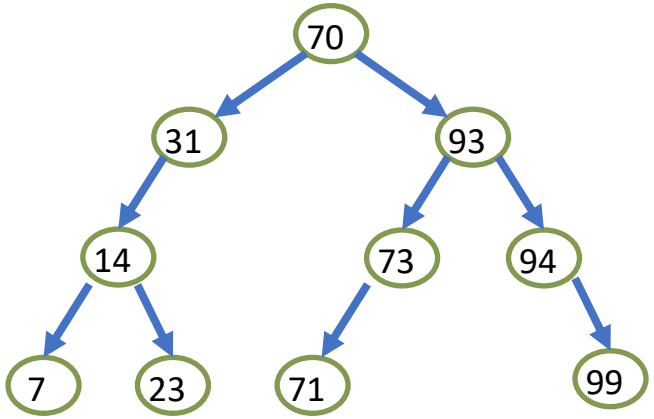




Binary Search Tree: Searching

- The approach is a decrease-and-conquer approach
- A problem is divided into two smaller and similar sub-problems, one of which does not even have to be solved
- This approach uses the information of the order to reduce the search space.

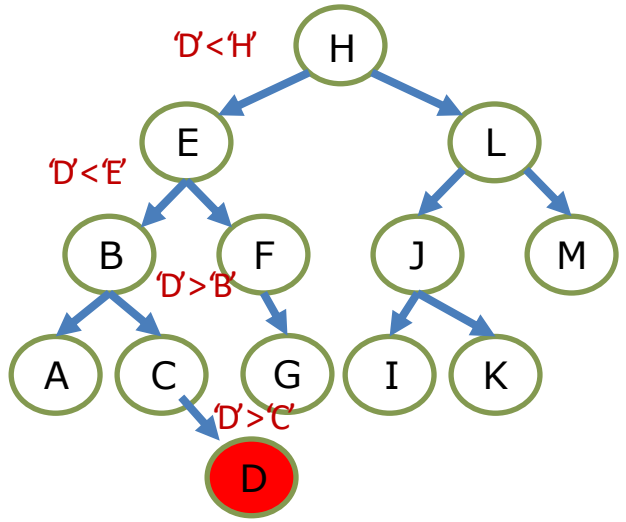
```
def _search(self, data, current_node):
        if current_node is None:
            return False
        elif data == current_node.data:
            return True
        elif data < current_node.data:
            return self._search(data, current_node.left)
        else:
            return self._search(data, current_node.right)
```



Binary Search Tree: Insertion

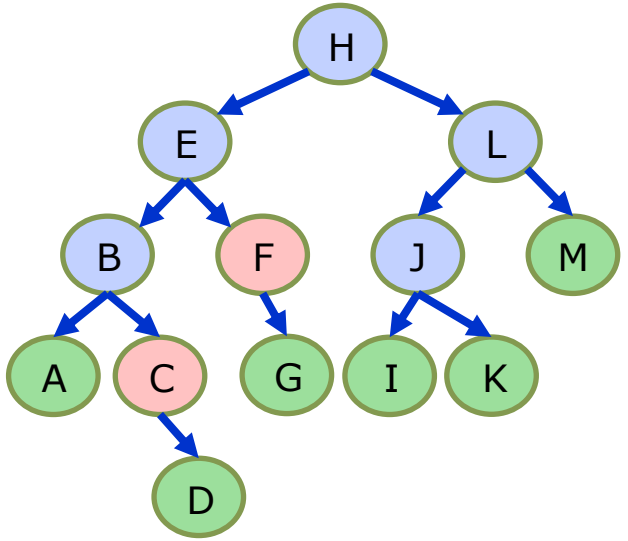
- After insert a node, the BST must remain as a BST
- A duplicate node is not allowed for insertion
- A unique position of the given BST will be given to the node

```
def _insert(self, data, current_node):
        if data < current_node.data:
            if current_node.left is None:
                current_node.left = Node(data)
            else:
                self._insert(data, current_node.left)
        else:
            if current_node.right is None:
                current_node.right = Node(data)
            else:
                self._insert(data, current_node.right)
```



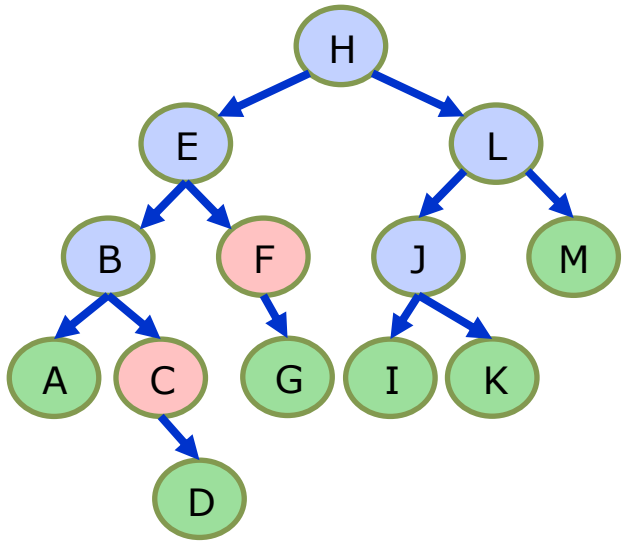
Deleting nodes in binary search trees

- Deleting nodes is the most complicated of the four basic operations
- After remove a node X, the BST must remain as a BST
- The procedure for deleting the node depends on its children:
  1. X has no children
  2. X has one child
  3. X has two children



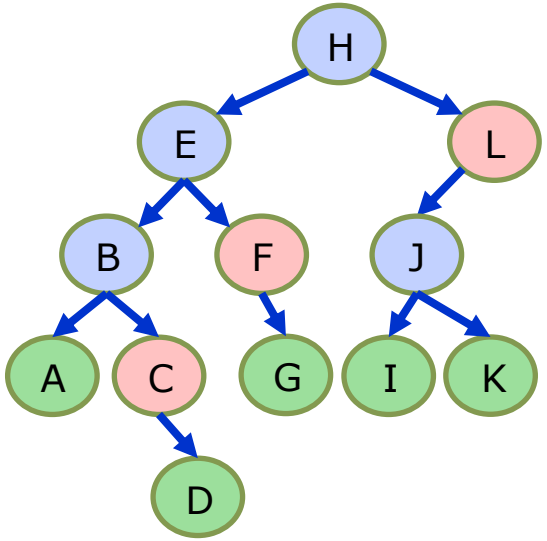
Deleting nodes in binary search trees

- Deleting nodes is the most complicated of the four basic operations
- After remove a node X, the BST must remain as a BST
- The procedure for deleting the node depends on its children:
  1. **X has no children**
     - Remove **X**



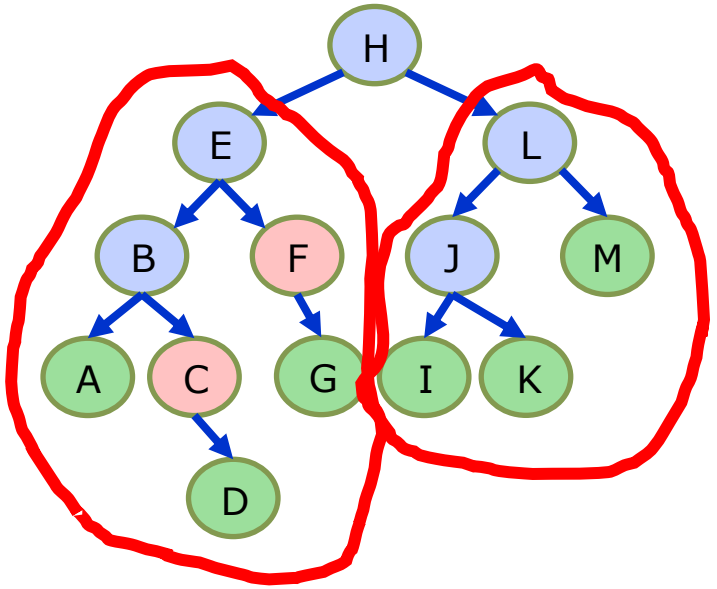
Binary Search Tree: Deletion

- Deleting nodes is the most complicated of the four basic operations
- After remove a node X, the BST must remain as a BST
- The procedure for deleting the node depends on its children:
  2. **X has one child**
     - Replace X with Y
     - Remove X



Binary Search Tree: Deletion

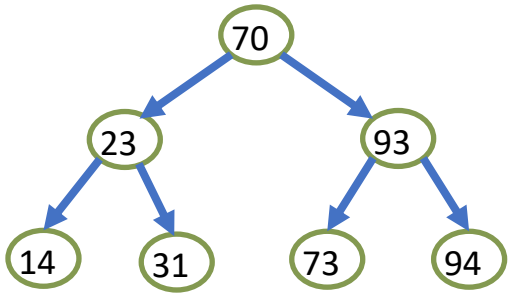
- Deleting nodes is the most complicated of the four basic operations
- After remove a node X, the BST must remain as a BST
- The procedure for deleting the node depends on its children:
  3. **X** has two children
     - **Swap x with successor**
       - the (largest) rightmost node in left subtree
       - the (smallest) leftmost node in right subtree
     - **Perform case 1 or 2 to remove it**

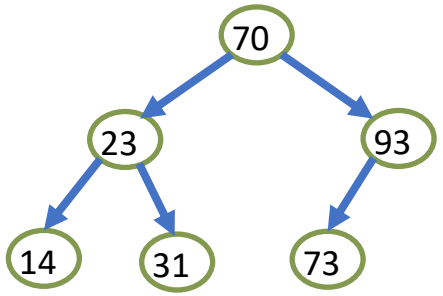


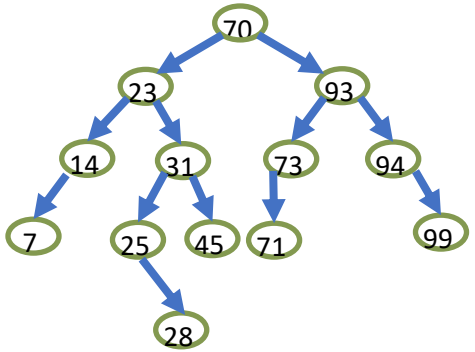
Balance Trees

Terminology

- **Height of a tree**: The number of **edges** on the longest path from the root to a leaf
- **Depth/Level of a node**: The number of edges from the node to the root of its tree.
- **Empty Binary Tree**: A binary tree with no nodes. It is still considered as a tree.
- **Perfect Binary Tree**: A binary tree of height *H* with no missing nodes. All leaves are at level *H* and all other nodes each have two children
- **Complete Binary Tree**: A binary tree of height H that is full to level H-1 and has level H filled in from left to right
- **Balanced Binary Tree**: A binary tree in which the left and right subtrees of any node have heights that differ by at most 1





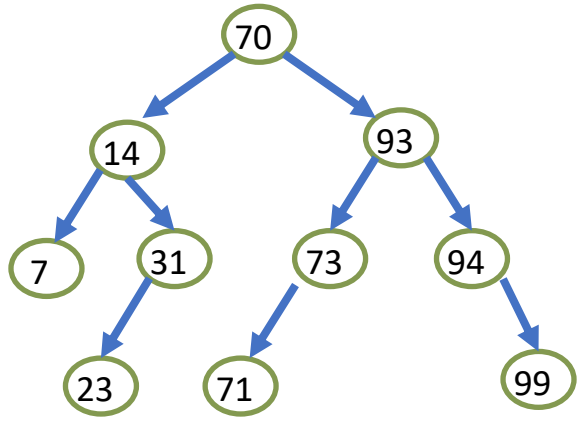


Balance Trees

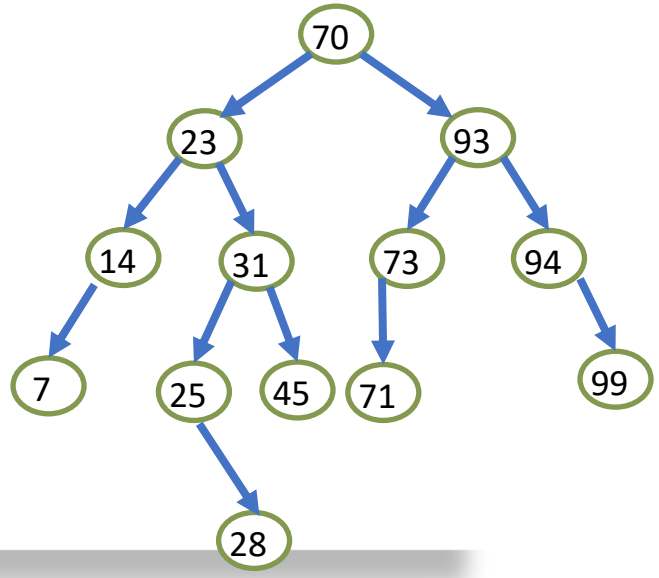
- Most Balanced BST: Each tree node has exactly two child nodes except for the bottom 2 levels
- How do we balance a binary search tree?

1. **Sort all the data in an array and reconstruct the tree**
2. **The AVL Tree:**
   - It is a locally balanced tree:
   - Heights of left vs right subtrees differ by at most 1
   - invented by Adel'son-Velskii and Landis in 1962

Most Balanced BST



AVL Tree



Sort all data in a list and reconstruct the tree

1. In-order traversal visits every node in the given BST. We obtain the sorted data:

   7,14,23,31,70,71,73,93,94,99

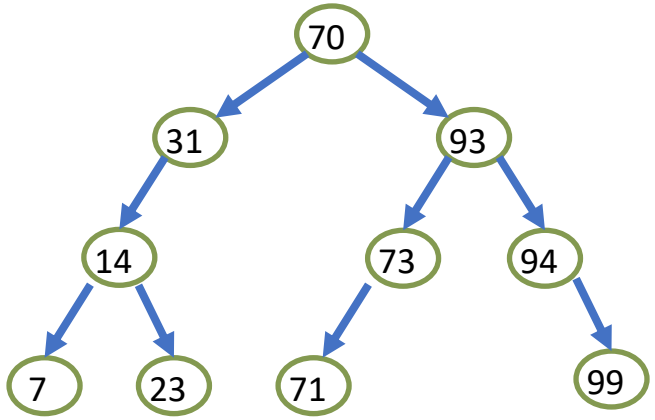
2. Storage it in a list
3. Take the middle element of the array as the root of the tree: 70
4. The first half of the array is used to build the left subtree of 70

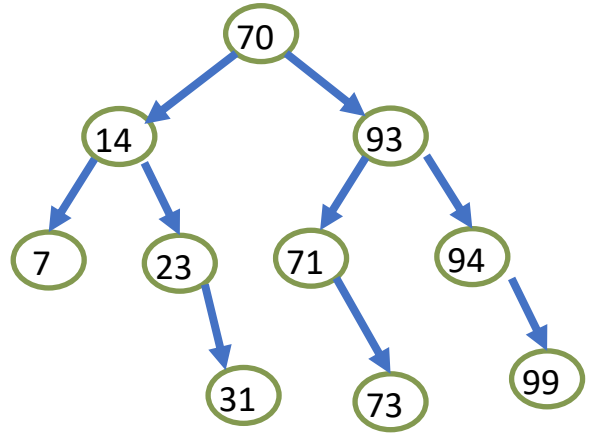
   7,14,23,31

5. The second half of the array is used to build the right subtree

   71,73,93,94,99

6. Step 4 and Step 5 recursively repeat the step 3-5.

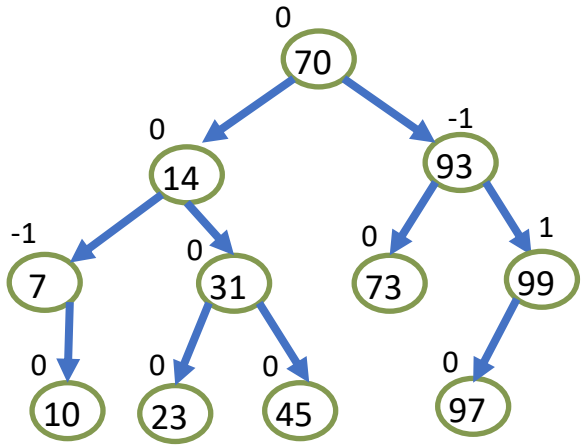




Balance Factor

- **Balance Factor: Height of Left Subtree – Height of Right Subtree**
- All the leave have 0 Balance Factor

```
class AVLNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.height = 0
```

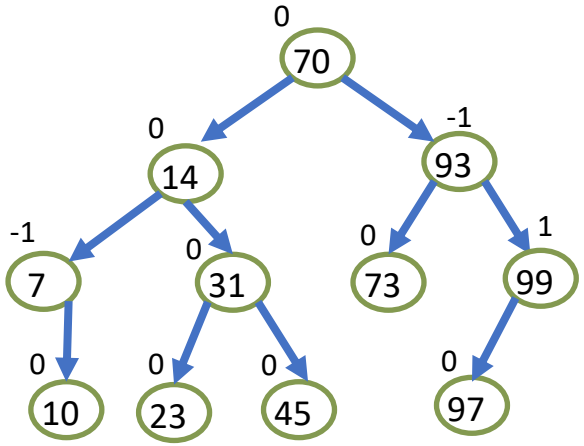


- **An AVL tree**: The **heights** of left and right subtrees of every node differ by at **most one**.
  - **Balance Factor** of each node in an AVL tree can only be **-1, 0** or **1.**
  - Node insertion or node removal from the tree may change the balance factor of its ancestors (from parent, grandparent, grand-grandparent etc. to the root of the tree)

Balance Factor

- **Balance Factor: Height of Left Subtree – Height of Right Subtree**

```
def _get_height(self, node):
        if not node:
            return -1
        return node.height # return 1 + max(left_height, right_height)
```




```
def _get_balance(self, node):
        if not node:
            return 0
        return self._get_height(node.left) - self._get_height(node.right)
```

Height of a tree: The number of **edges** on the longest path from the root to a leaf

Node Insertion

- **Case 1**: Balance factor of nodes along the insertion path from the root to the expectant parent node is zero.
- **Case 2**: Balance factor of nodes along the insertion path from the root to the expectant parent node is non-zero (1 or -1) but a new node is inserted at the shorter subtree.
- **Case 3**: Balance factor of nodes along the insertion path from the root to the expectant parent node is non-zero (1 or -1) and a new node is inserted at the higher subtree. The new node is inserted at the non-zero balance factor node's
    a) Left child's Left subtree (**LL Case**)
    b) Right child's Right subtree (**RR Case**)
    c) Left child's Right subtree (**LR Case**)
    d) Right child's Left subtree (**RL Case**)