# Programming, Data Structures, and Algorithms in Python: Week 7

This document covers the key topics from Week 7, focusing on the principles of abstraction and the creation of user-defined data types using object-oriented programming. We will explore the concepts of abstract data types (ADTs), classes, and objects, and apply them to implement our own versions of lists and binary search trees.

## 1. Abstract Data Types (ADTs)

A data structure is more than just a way to store data; it's an organization of information defined by a clear set of rules for interaction. This concept is formalized as an **Abstract Data Type (ADT)**, which emphasizes the separation between a data type's logical properties and its physical implementation.

* **Interface vs. Implementation:** An ADT rigorously separates the "what" from the "how."
  + **Interface (Public):** The interface defines the *behavior* of the data type through a specific, public set of allowed operations. This serves as a formal "contract" between the data type and the code that uses it. For example, a stack's interface contract promises that pop() will always return the last item added via push(). The user's code is written to rely solely on this contract.
  + **Implementation (Private):** The implementation consists of the internal details: the choice of underlying data structure (e.g., a list, an array, a dictionary) and the code that makes the interface operations work. This part is hidden from the user, a principle known as **encapsulation**. For instance, a priority queue could be implemented using a simple sorted list (inefficient) or a more complex but highly efficient heap structure.
* **Black Box View:** An ADT can be thought of as a "black box." You don't need to know its internal wiring or the complex mechanics inside. You are given a set of designated buttons (the interface functions) to interact with it, a slot for input, and a display for output. No other manipulation is allowed; you cannot pry open the box and tamper with the internal state directly. This perspective enforces discipline and prevents accidental corruption of the data's internal state.
* **Benefits of Abstraction:**
  + **Simplicity and Safety:** Users can work with a complex data structure through a simple, well-defined interface. This reduces cognitive load and prevents common errors that might arise from directly manipulating a complex internal state.
  + **Flexibility and Maintainability:** The implementation can be completely changed or optimized without affecting the external code that uses the data type. As long as the interface's contract is upheld, switching a priority queue from a list-based implementation (O(n) insertion) to a heap-based one (O(logn) insertion) requires no changes to the user's code but can yield significant, game-changing performance improvements.

## 2. Object-Oriented Programming: Classes and Objects

Object-Oriented Programming (OOP) is a programming paradigm that provides a natural and powerful way to create and manage ADTs. It bundles data and the methods that operate on that data into single units.

* **Class:** A blueprint or template for creating a data type. It defines:
  + The **attributes** (internal data variables) that each instance of the class will possess. For a Point class, this might be x and y coordinates.
  + The **methods** (functions belonging to the class) that define the public interface for creating, inspecting, and manipulating that data.
* **Object (Instance):** A concrete instance created from a class. If Stack is a class, then s1 = Stack() and s2 = Stack() create two separate, independent stack objects. Each object holds its own internal data (its own set of items) but shares the method definitions provided by the class.

### Defining Classes in Python

* **class Keyword:** Defines a new class, e.g., class Point:.
* **The Constructor (\_\_init\_\_)**: A special method called automatically whenever a new object is created (e.g., when Point(3, 4) is called). Its primary purpose is to initialize the object's attributes, setting its initial state.
* **The self Parameter:** The first parameter of every method in a class must be self. self is a reference to the specific object instance the method is being called on. It's the mechanism that allows a method to access and modify the object's own unique attributes (e.g., self.value, self.left). When you call a method like p.translate(dx, dy), Python automatically passes the object p as the first self argument to the translate method.
* **Attributes:** Data associated with a specific object is stored in attributes, which are accessed using dot notation (e.g., self.x). These attributes make up the object's state.
* **Special Methods:** Python uses special methods with double underscores (often called "dunder" methods) to allow user-defined objects to integrate seamlessly with Python's built-in syntax and functions.
  + \_\_str\_\_(self): Called by str() and print(). Should return a user-friendly string representation of the object. Defining this method allows for intuitive debugging.
  + \_\_add\_\_(self, other): Implements the + operator. When Python sees p1 + p2, it translates it to p1.\_\_add\_\_(p2).
  + \_\_lt\_\_(self, other): Implements the < (less than) comparison operator, allowing objects to be sorted or compared naturally.

## 3. User-Defined Linked List

Let's apply these OOP concepts to build our own version of a **linked list**. A linked list is a fundamental data structure composed of a sequence of nodes, where each node contains a value and a pointer (or reference) to the next node. This differs from a standard Python list, which is internally an array (a contiguous block of memory).

### Node Class Implementation

We can define a Node class where each Node object represents a single element but also recursively serves as the head of the rest of the list.

* **Attributes:** Each Node will have self.value and self.next.
* **Empty List Representation:** A single Node object where self.value is None represents an empty list. This is necessary so that even an "empty" list is still a valid object of type Node that can have methods called on it.
* **Singleton List:** A Node where self.value holds data but self.next points to an empty node.

#### Key Methods:

* **append(v):** To add a value v to the end of the list. The implementation must traverse the entire list until it finds the last node, making this an O(n) operation.
  + **Base Cases:**
    - If the list is empty (self.value is None), we simply set self.value = v, turning the empty list into a singleton.
    - If the current node is the last node (self.next.isempty()), we create a new Node(v) and set self.next to point to it.
  + **Recursive Step:** If not at the end, we pass the task down the chain: self.next.append(v).
* **insert(v):** To add a value v to the beginning of the list. This is surprisingly tricky because in Python, we cannot simply reassign the head of the list (self) to a new node from within a method.
  + **The Trick:** The identity of the head node object must be preserved. We achieve this by manipulating the *contents* of the nodes. We create a new node that temporarily holds the old head's value and next pointer. Then, we update the current head's value to the new value v and make it point to this newly created node (which is now the second element). This effectively inserts v at the front while the external variable pointing to the head of the list remains unchanged.
* **delete(v):** To remove the first occurrence of value v.
  + **Deleting the Head:** If self.value == v, we use a similar trick to insert. We copy the value and next reference from the second node (self.next) into the current head node. This overwrites the head's data and makes it point past the second node, effectively deleting it.
  + **Deleting Elsewhere:** To delete a node further down the list, we must modify the next pointer of the *preceding* node. This requires looking one step ahead. If the value to be deleted is in self.next, we bypass it by setting self.next = self.next.next. Otherwise, we make a recursive call on the rest of the list: self.next.delete(v).

## 4. Binary Search Trees (BST)

A binary search tree is a node-based binary tree data structure that maintains its keys in sorted order, making it highly efficient for searching. It is defined by the following properties:

* The left subtree of a node contains only nodes with values **lesser** than the node’s value.
* The right subtree of a node contains only nodes with values **greater** than the node’s value.
* Both the left and right subtrees must also be binary search trees.
* There must be **no duplicate values**.

This structure enables fast lookup, insertion, and deletion of items, serving as a powerful tool for managing dynamic sorted data.

### BST Class Implementation

Similar to our linked list, we'll define a Tree class where each object is a node.

* **Attributes:** self.value, self.left (a reference to the left child Tree object), self.right (a reference to the right child Tree object).
* **Empty Tree:** An empty tree is represented by a Tree object where self.value is None.
* **Leaf Node:** A node with a value, whose left and right children are empty trees. This recursive definition simplifies the implementation of methods.

#### Key Methods:

* **find(v):** Searches for a value v. It's a natural generalization of binary search for a tree structure.
  + If the current node is empty, the search has failed, and v is not in the tree.
  + If v == self.value, v has been found.
  + If v < self.value, the value must be in the left subtree, so recursively search there: self.left.find(v).
  + If v > self.value, the value must be in the right subtree, so recursively search there: self.right.find(v).
* **insert(v):** Adds a value v to the tree while maintaining the BST property.
  + It follows the exact same search path as find.
  + When the search path leads to an empty node, that is the correct position for v. The empty node is then converted into a new leaf node containing v (and two new empty children).
  + If v is already found during the search, the operation terminates to prevent duplicates.
* **delete(v):** The most complex operation. After finding the node N with value v:
  + **Case 1: N is a leaf.** It has no children, so it can be removed by setting its parent's pointer to it to None (or making it an empty node).
  + **Case 2: N has one child.** Promote the child by linking N's parent directly to N's child, bypassing N.
  + **Case 3: N has two children.** This is the most complex case. To fill the "hole" left by deleting N, its value must be replaced while preserving the BST property. We replace it with either:
    - The largest value from its left subtree (its **inorder predecessor**). This value is guaranteed to be greater than all other values in the left subtree and less than all values in the right subtree.
    - The smallest value from its right subtree (its inorder successor).  
      After copying the value, we then recursively delete the node from which the replacement value was taken (which is guaranteed to be an easier Case 1 or Case 2 deletion).

### Complexity and Balanced Trees

For all primary BST operations (find, insert, delete), the time complexity is proportional to the **height of the tree,** O(h).

* **Balanced Tree:** If the tree is reasonably balanced (the left and right subtrees of every node have roughly the same height), the height h is approximately O(logn) for n nodes. In this state, all operations are highly efficient.
* **Unbalanced Tree:** In the worst-case scenario, such as inserting items that are already in sorted order (e.g., [10, 20, 30, 40]), the tree can degenerate into a long chain with no left children. It essentially becomes a linked list. The height becomes O(n), and the performance of all operations degrades to linear time, losing the primary advantage of the tree structure.

To prevent this degeneration and guarantee O(logn) performance, more advanced **self-balancing BSTs** like **AVL trees** or **Red-Black trees** perform "rotations"—local pointer adjustments—during insertions and deletions to automatically maintain the tree's balance.