5 Data Structures

Hui Xu, xuh@fudan.edu.cn

Learning Objective:

- Understand fundamental linear data structures, including arrays and linked lists.
- Understand non-linear data structures such as trees and graphs.
- Learn how to design and implement tree and graph data structures in Python.

5.1 Overview

A data structure is a specialized format for organizing and storing data, designed to support efficient access and modification through well-defined operations.

The `@dataclass` decorator simplifies defining structured data types in Python. It automatically provides initialization, representation (so class instances can be printed in a readable form), and comparison methods (so class instances can be compared directly), allowing you to focus on designing the structure and operations.

```python
from dataclasses import dataclass

@dataclass
class Point():
    x: int
    y: int

    def other_methods(self):
        ...
```

Code 5.1: Define Point with dataclass

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return f"Point(x={self.x}, y={self.y})"

    def other_methods(self):
        ...
```

Code 5.2: Real class of Point

Figure 5.1: Comparison of `@dataclass` definition vs. expanded class

The above example shows how to define a data structure with Python’ s `@dataclass`. On the left, we define a Point class using `@dataclass`. We only declare the attributes x and y, plus any additional methods like `other_methods`. We don’ t need to write the constructor, representation, or comparison methods manually as `@dataclass` generates them automatically. On the right, we see what the class looks like after the dataclass decorator has been processed. It dramatically simplifies class definitions while still providing all the essential functionality.

Based on the organization of elements, data structures can be categorized into two main types: linear and non-linear. Next, we will introduce each of them in detail.

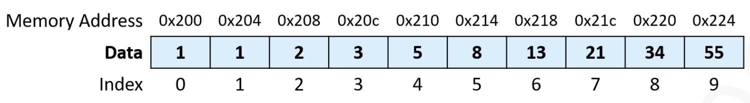
5.2 Linear Data Structures

Linear data structures arrange elements in a sequential order. From the perspective of data distribution, linear data structures includes array and linked list.

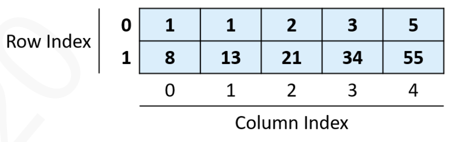
5.2.1 Array

An array is one of the simplest and most fundamental linear data structures. It consists of a fixed number of elements of the same type stored in contiguous memory locations. This organization allows direct access to any element by its index, since the memory address of each element can be computed efficiently as an offset from the base address. Figure 5.2a demonstrates an example with 10 elements of 4-byte integers. If `a[0]` is the first element stored at memory address `0x200`, the address of the ith element `a[i]` can be calculated as:

$$\text{Address of } a[i] = 0x200 + i \times 4.$$



(a) Array elements in memory.



(b) Two dimensional array.

Figure 5.2: Example of array.

FISF130020

A multi-dimensional array is also a linear data structure in memory. For example, Figure 5.2b shows a two-dimensional array with the same 10 elements, where each element can be accessed through a row index and a column index as `a[i][j]`. The layout of these elements in memory is still linear, as shown in Figure 5.2a. Assuming each element occupies 4 bytes and the array has n columns, the address of `a[i][j]` can be computed as:

$$\text{Address of } a[i][j] = 0x200 + (i \times n + j) \times 4.$$

Accessing an array element is extremely fast, with constant-time or $O(1)$ complexity. Despite this efficiency, arrays have limitations. Their size or memory occupancy must be defined in advance, and inserting or deleting elements in the middle of an array requires shifting other elements to maintain order, resulting in a time complexity of $O(n)$, *i.e.,* linear in the number of elements.

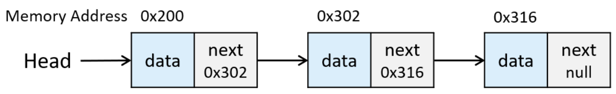
Since Python is an interpreted language without direct control over memory allocation, developers cannot create low-level arrays in the same way as languages like C or C++. However, Python provides the `list` type for general-purpose sequences and the `NumPy` library, which offers the `ndarray` type. `ndarray` supports multi-dimensional arrays with efficient memory storage and allows element-wise operations such as addition, subtraction, and more, similar to arrays in compiled languages.

```python
import numpy as np
a = np.array([[1, 2], [3, 4]]) # Create a 2x2 array from a 2x2 list
b = np.array([[5, 6], [7, 8]])
c = a + b
```

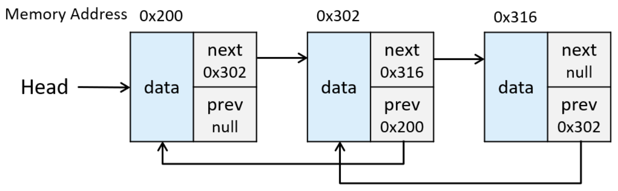
Code 5.3: Using array in Python via NumPy

5.2.2 Linked List

A linked list is another fundamental linear data structure. Unlike an array, the elements of a list are not stored in contiguous memory locations. Instead, each element, called a *node*, contains both the data and a pointer to the next node in the sequence. This structure allows dynamic memory allocation, meaning the list can easily grow or shrink during program execution without the need to reallocate or shift elements.



(a) Single-linked list.



(b) Double-linked list.

Figure 5.3: Example of linked lists.

Figure 5.3a illustrates the layout of a *single-linked list*, where each node contains a value and a pointer to the next node. The last node points to `null`, indicating the end of the list.

A *double-linked list* extends the single-linked list by adding a second pointer in each node that refers to the previous node, as shown in Figure 5.3b. This bidirectional linkage allows traversal in both forward and backward directions. While it requires additional memory to store the extra pointer, it provides greater flexibility, particularly for operations such as reverse traversal or deletion of a node when only a reference to that node is available.

Accessing an element in a linked list requires sequential traversal starting from the head node, giving a time complexity of $O(n)$. However, insertion and deletion can be performed efficiently once the target position is located, typically in constant time $O(1)$, since only pointer references need to be updated rather than shifting elements as in arrays.

FISF130020

5.3 Graphs

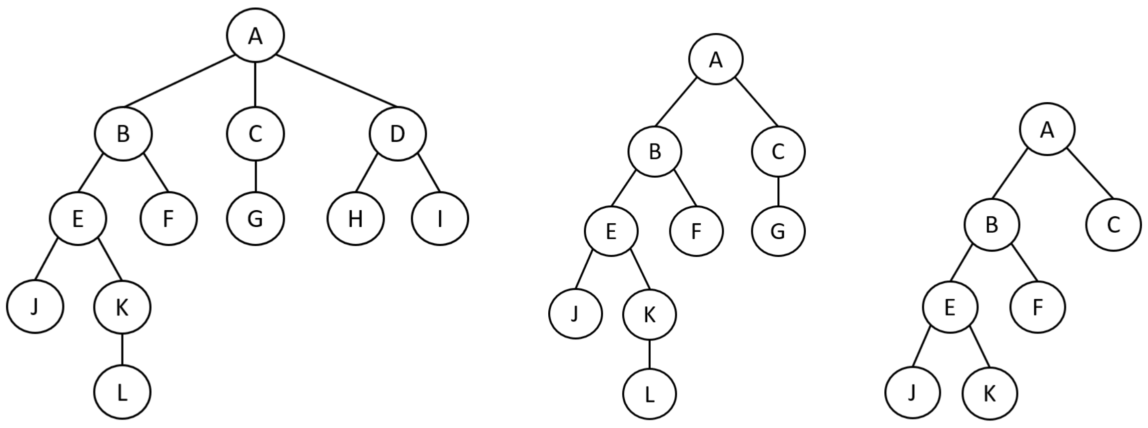
This section introduces non-linear data structures, including trees, which are a special kind of graph, and general graphs.

5.3.1 Trees

A tree is a widely used non-linear data structure that simulates a hierarchical organization of elements. It consists of nodes connected by edges, with the following key concepts:

- The *root* is the topmost node of the tree, serving as the starting point for traversal.
- A *leaf* is a node with no children.
- *Children* are nodes directly connected to a parent node.
- *Parent* refers to the node immediately above a given node.
- *Ancestor* is any node along the path from a node to the root, including the parent.
- The *degree* of a node is the number of its children.

- The *height* of a node is the length of the longest path from the node to a leaf.
- The *height of the tree* is the height of the root node.



(a) Tree. (b) Binary tree. (c) Full binary tree.

Figure 5.4: Demonstration of trees.

Figure 5.4a demonstrates an example tree with a root `A` and six leaf nodes {`J`,`L`,`F`,`G`,`H`,`I`}. In general, a tree allows each node to have an arbitrary number of children. The maximum degree of a node on the tree is three, because node `A` has three children. The height of the tree is four, as the longest path is `A-B-E-K-L`.

A *binary tree* is a special type of tree in which each node has at most two children, commonly referred to as the *left child* and *right child*. If we further restrict that each non-leaf node must have exactly two children, the tree becomes a *full binary tree*. Figure 5.4b and Figure 5.4c illustrate examples of a binary tree and a full binary tree, respectively.

A binary tree can be traversed in different orders. The three commonly used traversal orders differ in when the parent node is visited relative to its left and right children.

- *Pre-order traversal*: The root node is visited first, followed by the left and right subtrees, *e.g.,* `A-B-E-J-K-F-C` for Figure 5.4c.
- *In-order traversal*: The left subtree is visited first, then the root, and finally the right subtree, *e.g.,* `J-E-K-B-F-A-C` for Figure 5.4c.
- *Post-order traversal*: The left and right subtrees are visited before the root node, *e.g.,* `J-K-E-F-B-C-A` for Figure 5.4c.

FSF 2020

5.3.2 Implementing Trees in Python

Code 5.4 defines a binary tree where each node stores an integer value and has two child nodes. In Python, the `Optional` type is used to indicate that a variable may either hold a value of a given type or be `None`. Since a node in a binary tree may have only one child or none at all, each child node is therefore typed as `Optional` and initialized to `None`.

```
@dataclass
class Node():
    value: int
    left: Optional[Node[int]] = None
    right: Optional[Node[int]] = None

@dataclass
class BinaryTree():
    root: Optional[Node[int]] = None
```

Code 5.4: Example of binary tree in PyThon

In addition to `Optional`, Python also supports generic types, which allow a class or data structure to operate with values of different types while preserving type safety. A generic type is defined using a type parameter, such as `T`, which acts as a placeholder for any concrete type specified later. By defining the node’ s value as a generic type, the binary tree becomes more flexible. Code 5.5 illustrates an example, where `TreeNode[T]` can store integers, strings, or even user-defined objects, depending on the type provided when creating the tree instance.

```
@dataclass
T = TypeVar("T")

@dataclass
class Node(Generic[T]):
    value: T
    left: Optional[Node[T]] = None
    right: Optional[Node[T]] = None

@dataclass
class BinaryTree(Generic[T]):
    root: Optional[Node[T]] = None
```

Code 5.5: Example of binary tree with a generic value for each node

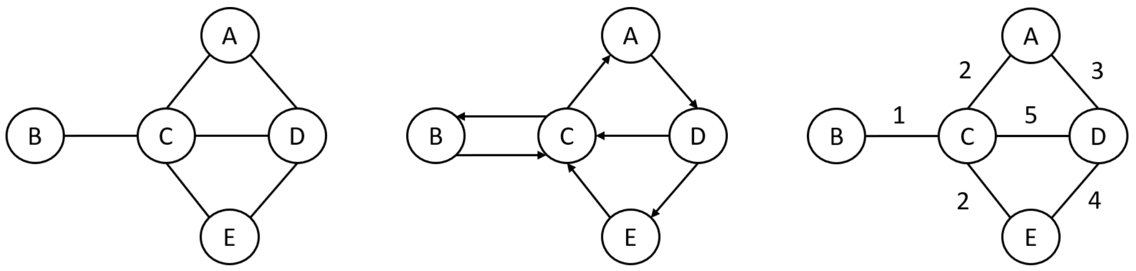
Unlike a binary tree, each node in a general tree can have an arbitrary number of child nodes. Code 5.6 shows an example, where the `children` field is a list of child nodes. The list is initialized using `field(default_factory=list)`, which ensures that each instance of `TreeNode` gets its own separate empty list by default. Note that we cannot simply use `children: List[TreeNode[T]] = []` because mutable default arguments are shared across all instances of the class. Using `default_factory` avoids this issue by creating a new list for each instance, preventing unintended side effects when modifying the children of different nodes.

```
@dataclass
from __future__ import annotations
from dataclasses import dataclass, field

T = TypeVar("T")

@dataclass
class TreeNode(Generic[T]):
    value: T
    children: List[TreeNode[T]] = field(default_factory=list)
```

Code 5.6: Tree node for a general tree

5.3.3 Graph

A graph is a non-linear data structure consisting of a set of *vertices* (also called nodes) and a set of *edges* that connect pairs of vertices. Graphs are used to model relationships between objects, such as networks or social connections.



(a) Undirected graph. (b) Directed graph. (c) Weighted graph.

Figure 5.5: Demonstration of graphs.

If the edges have no direction, the graph is called an *undirected graph.* Having no direction means that if there is an edge between vertex A and vertex B, it can be traversed in both directions. Formally, the edge (A, B) is identical to (B, A). In contrast, in a *directed graph*, each edge has a direction, represented as an ordered pair (A, B), meaning that traversal is only allowed from A to B. Figure 5.5a and Figure 5.5b illustrate an undirected graph and a directed graph, respectively.

Orthogonally, an edge may also carry a weight. A *weighted graph* associates each edge with a numerical value, or *weight*, which may represent distance, cost, or time. Both directed and undirected graphs can have weights on their edges. Figure 5.5c shows an example of a weighted undirected graph.

Graphs can be traversed using depth-first search (DFS) or breadth-first search (BFS).

- *DFS*: Starting from a node, we explore as far along each branch as possible before backtracking, *e.g.,* `A-D-C-B-E` for Figure 5.5b.
- *BFS*: Visit all neighbors of a node before moving to the next level, *e.g.,* `A-D-C-E-B` for Figure 5.5b.

FIST 2020

_	A	B	C	D	E
A	0	0	1	1	0
B	0	0	1	0	0
C	1	1	0	1	1
D	1	0	1	0	1
E	0	0	1	1	0

Matrix for Fig. 5.5a

_	A	B	C	D	E
A	0	0	0	1	0
B	0	0	1	0	0
C	1	1	0	0	0
D	0	0	1	0	1
E	0	0	1	0	0

Matrix for Fig. 5.5b

_	A	B	C	D	E
A	∞	∞	2	3	∞
B	∞	∞	1	∞	∞
C	2	1	∞	5	2
D	3	∞	5	∞	4
E	∞	∞	2	4	∞

Matrix for Fig. 5.5c

Figure 5.6: Three adjacency matrices side by side.

A graph can be represented using an *adjacency matrix*, which is a two-dimensional array where each row and column correspond to the vertices of the graph. If there is an edge from vertex i to vertex j, the element at position (i, j) is set to 1 (or to the edge weight in the case of a weighted graph); otherwise, it is set to 0 or ∞ to indicate no connection. For an undirected graph, the adjacency matrix is symmetric along the main diagonal, while for a directed graph, it is usually asymmetric. Figure 5.6 illustrates examples of adjacency matrix representations corresponding to the graphs shown in Figure 5.5.

5.3.4 Implementing Graphs in Python

Graphs can be implemented in Python in various ways depending on the intended use case. Code 5.7 shows an example of a weighted graph represented using an adjacency matrix. Here, each node is represented by an instance of the `Node` class, and the `Graph` class contains a dictionary that maps each node to another dictionary of its neighbors and the corresponding edge weights. Dictionaries are used because they prevent duplicate entries and allow efficient lookups and updates. Using `@dataclass(eq=True, frozen=True)` for nodes ensures that each node is hashable and can safely serve as a dictionary key. This design is particularly efficient for checking the existence of a connection between two nodes or looking up the weight of a specific edge.

```python
@dataclass
class Node():
    id: int

@dataclass(eq=True, frozen=True)
class Graph():
    adj: Dict[Node, Dict[Node, int]] = field(default_factory=dict)
```

Code 5.7: Example of weighted graph in PyThon

An alternative approach is to represent a weighted graph using an adjacency list, as shown in Code 5.8. In this design, each `Node` maintains a list of tuples (neighbor, weight), storing its outgoing edges and their weights. The `Graph` class contains a list of nodes, which can serve as entry points for traversals; additional nodes may be reachable only through edges in the lists. This approach is often more memory-efficient for sparse graphs and is convenient when iterating over all neighbors of a node.

```python
@dataclass
class Node():
    id: int
    List[Tuple[Node, int]] = field(default_factory=list)

@dataclass
class Graph():
    nodes: List[Node] = field(default_factory=list)
```

Code 5.8: Example of weighted graph in PyThon

So far, we have discussed common approaches to designing data structures. In the next few lectures, we will focus on defining and implementing operations and algorithms on these structures.

Exercise

1. Given an adjacency matrix, how can we determine whether it represents a tree or a general graph?