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.
- Use agent to design and implement basic 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. In Python, we can implement a data structure using a `class`, which defines a type of object with attributes (data) and methods (behavior). Attributes can be created dynamically, for example, in the `__init__` method.

The `@dataclass` decorator provides a concise way to define data-oriented classes in Python. By declaring fields in the class body, it automatically generates methods such as `__init__`, `__repr__`, and `__eq__`, reducing boilerplate code.

```python
from dataclasses import dataclass

@dataclass
class Point:
    x: int
    y: int

    def other_methods(self):
        ...



    #
```

Code 5.1: Point defined with `@dataclass`

```python
class Point:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y

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

    def __eq__(self, other):
        ...

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

Code 5.2: Methods generated by `@dataclass`

Figure 5.1: A `@dataclass` definition and its generated methods

Code 5.1 shows how to define a data structure, namely `Point`, using Python' s `@dataclass` decorator. We declare the fields `x` and `y`, along with any additional methods such as `other_methods`. The decorator automatically generates methods such as the constructor (`__init__`), representation (`__repr__`), and equality comparison (`__eq__`), eliminating the need to implement them manually. Note that each field must have a type annotation, but the annotation can be `object` to allow values of any type. You may also write `x = 1`, but without a type annotation, `x` is treated as a class attribute rather than a dataclass field, and is not included in the generated method.

Based on how elements are organized, data structures can be classified into two main categories: linear and non-linear. We will introduce each category in detail in the following sections.

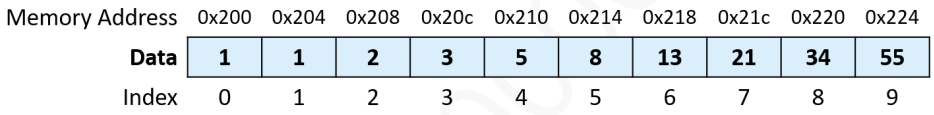
5.2 Linear Data Structures

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

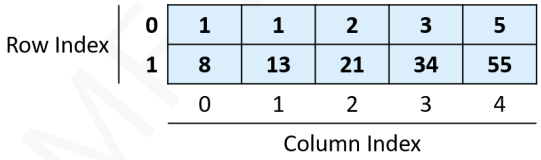
5.2.1 Array

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) Abstract representation of a two-dimensional array.

Figure 5.2: The memory layout and abstract representations of arrays.

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:

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

Pros and Cons Accessing an array element is extremely fast, taking constant time, or (O(1)), regardless of the element's position in the array. 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.

MT2026

Using Arrays in Python Python provides the built-in `list` type for storing sequences of elements. However, a Python list is not a traditional array of elements stored contiguously in memory; instead, it stores references to Python objects. For applications that require efficient numerical computation, the `NumPy` library provides the `ndarray` type, which supports multi-dimensional arrays and efficient storage of numerical data. It also supports element-wise operations, such as addition and subtraction, making it suitable for numerical computing.

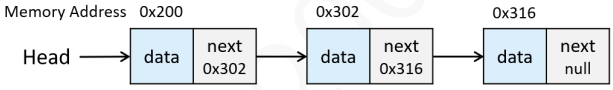
```
import numpy as np

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
```

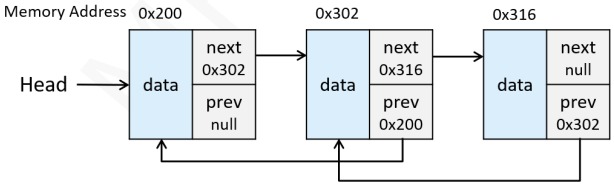
Code 5.3: Using array in Python via NumPy

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.

Pros and Cons 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.

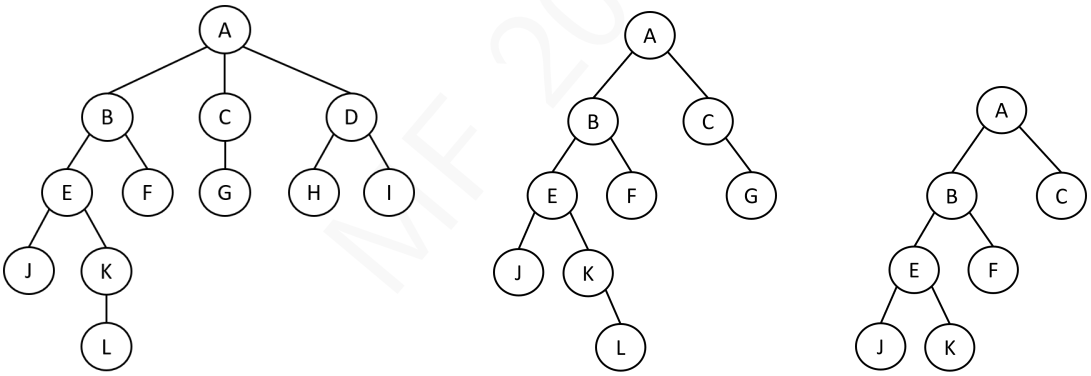
5.3 Graphs

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.
- *Parent* refers to the node immediately above a given node.
- *Children* are nodes directly connected to a parent 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`.

Binary Tree 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.

NF 2006

Traversal Order 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.

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

A binary tree can use either `object` or a generic type parameter to specify the type of each node's value. These two approaches differ in how they express type constraints.

```
from dataclasses import dataclass
from typing import Optional



@dataclass
class Node:
    value: object
    left: Optional["Node"] = None
    right: Optional["Node"] = None

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

Code 5.5: Binary tree using `object`

```
from dataclasses import dataclass
from typing import Generic, Optional, TypeVar

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 5.6: Generic binary tree using `T`

Figure 5.5: Binary trees using `object` and generic types.

Using `object` allows different nodes in the same tree to store values of different types, such as integers, strings, and user-defined objects. In contrast, a generic type uses a type parameter, such as `T`, to specify the type of values stored in the tree, so all nodes are expected to store values of the same type.

Unlike a binary tree, each node in a general tree can have an arbitrary number of child nodes. Code 5.7 shows an example, where the `children` field is a list of child nodes and the `value` field uses `object`, allowing different nodes to store values of different types.

```python
from __future__ import annotations
from dataclasses import dataclass, field
from typing import List

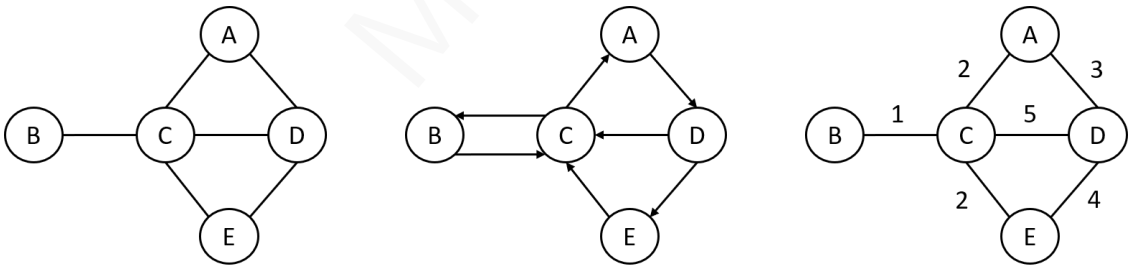
@dataclass
class TreeNode:
    value: object
    children: List[TreeNode] = field(default_factory=list)
```

Code 5.7: Tree node for a general tree

The list is initialized using `field(default_factory=list)`, which tells the data class to call `list()` whenever a new `TreeNode` instance is created without an explicit value for `children`. We cannot simply use `children: List[TreeNode] = []`, because the same mutable list would be shared across all instances. Using `default_factory` ensures that each instance gets its own empty list, preventing unintended side effects when modifying the children of different nodes.

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 5.6: Demonstration of graphs.

Directed vs. Undirected If the edges have no direction, the graph is called an *undirected graph*. An edge between vertices A and B 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 and is represented as an ordered pair (A, B), meaning that the edge goes from A to B. Figure 5.6a and Figure 5.6b illustrate an undirected graph and a directed graph, respectively.

Weighted vs. Unweighted Independently of whether a graph is directed or undirected, its edges may have weights. A *weighted graph* associates each edge with a numerical value, called a *weight*, which may

represent distance or cost. A graph whose edges have no associated weights is called an *unweighted graph*. Both directed and undirected graphs can be weighted or unweighted. Figure 5.6c shows an example of a weighted undirected graph.

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

- *DFS*: Explore as far as possible along each branch before backtracking, *e.g.,* `A-D-C-B-E` for Figure 5.6b.
- *BFS*: Visit all neighboring vertices at the current level before moving to the next level, *e.g.,* `A-D-C-E-B` for Figure 5.6b.

The traversal order is not unique when a vertex has multiple neighbors to visit, as the order in which these neighbors are explored can vary.

Representation with Adjacency Matrix A graph can be represented using an *adjacency matrix*, which is a two-dimensional array in which each row and column correspond to a vertex. If there is an edge from vertex i to vertex j, the element at position (i, j) is set to 1 for an unweighted graph or to the edge weight for a weighted graph. Otherwise, it is set to 0 or ∞, respectively, to indicate the absence of an edge.

_	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. 5.6a

_	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. 5.6b

_	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. 5.6c

Figure 5.7: Three adjacency matrices side by side.

Figure 5.7 illustrates examples of adjacency matrix representations corresponding to the graphs shown in Figure 5.6. For an undirected graph, the adjacency matrix is symmetric about the main diagonal. For a directed graph, it is not necessarily symmetric, because an edge from vertex i to vertex j does not imply an edge from vertex j to vertex i.

5.3.4 Implementing Graphs in Python

Graphs can be implemented in Python in various ways depending on the intended use case.

```python
@dataclass(frozen=True)
class Node:
    id: int


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

Code 5.8: Example of a weighted graph using nested dictionaries

Code 5.8 shows a weighted graph represented using an adjacency list implemented with nested dictionaries. The outer dictionary maps each node to a dictionary of its neighbors and the corresponding edge weights. Dictionaries support efficient lookups and updates. Since nodes are used as dictionary keys, they must be hashable. The `@dataclass` decorator generates the `__eq__` method by default, which compares instances based on their field values. However, mutable `Node` instances are not hashable by default. Setting `frozen=True` prevents fields from being reassigned after instance creation and allows a hash method to be generated when the fields are hashable. Therefore, `@dataclass(frozen=True)` allows `Node` instances to serve as dictionary keys. Two nodes with the same `id` are considered equal.

An alternative approach is to represent a weighted graph using an adjacency list stored directly in each node, as shown in Code 5.9.

```python
from dataclasses import dataclass, field
from typing import List, Tuple

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

@dataclass
class Graph:
    node: Node  # A single entry node, if all nodes are reachable from it.
    # Alternatively, store all nodes to include disconnected components.
    # nodes: List[Node] = field(default_factory=list)

n1 = Node(1)
n2 = Node(2)
n3 = Node(3)

n1.edges = [(n2, 5), (n3, 10)]
graph = Graph(n1)
```

Code 5.9: Implementing a weighted graph using adjacency lists

In this representation, each `Node` maintains a list of tuples, where each tuple contains a neighboring node and the corresponding edge weight. For example, `n1.edges = [(n2, 5), (n3, 10)]` means that node 1 has edges to nodes 2 and 3, with weights 5 and 10, respectively. The `Graph` class stores a single entry node if all nodes are reachable from it. If the graph contains disconnected components, it can instead store a list of all nodes. This representation is often more memory-efficient for sparse graphs and makes it convenient to iterate over a node's neighbors.

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 for these data structures.

Exercise

1. Given an adjacency matrix, how can we determine whether it represents a tree or a general graph?
2. Can every finite, undirected, simple graph in which every vertex has degree 2 be drawn in the plane without any edge crossings? Prove your answer.
3. Implement an undirected/directed graph with the following features and test the correctness of the generated code.
    - Add: insert nodes and edges into the graph.
    - Remove: delete nodes and edges from the graph.
    - Traversal: visit all nodes using depth-first or breadth-first order.
    - Search: check for the existence of nodes, edges, or paths.
4. Design a weighted graph in which each vertex has a 2D coordinate, and the weight of each edge is the Euclidean distance between its two endpoints.

MF 20006