Fundamental Data Structures

Madiba Hudson-Quansah

Contents

Chapter 1

Introduction to Data Structures

1.1 Data Structures

Definition 1.1.1: Data Type

Describes the set of values that a variable can hold and the types of operations that can be performed on it.

Definition 1.1.2: Data Structure

A way of storing and organizing data in computer memory / The logical mathematical model of a particular organization of data.

In choosing a data structure, we consider the following in the following order:

1. The relationships of the data in the real world
2. The type and amount of data to be stored
3. Cost (time complexity) of operations
4. Memory occupations
5. Ease of implementation

1.1.1 Types of Data Structures

Data structures can be broadly classified into two types:

- Primitive Data Structures
- Non-Primitive / Composite Data Structures

With data structures under the primitive category being:

- Variables
  - Integer
  - Float
  - Character
  - Boolean
  - Enumerated
  - Reference / Pointer

And data structures under the non-primitive category being:

- Arrays
- Structure / Record
- Union
- Class
- Abstract Data Type
  - Linear
    - List
    - Stack
    - Queue
  - Non-Linear / Associative
    - Tree
    - Graph
    - Hash Table

Data structures can also more simply be classified in two ways:

- Linear / Sequential Data Structures
  - Linked List
  - Array
  - Stack
  - Queue
  - Set
- Non-Linear / Associative Data Structures
  - Graph
  - Tree
  - Hash Table

In this way data structures are defined by an implementation of a particular abstract data type.

1.1.1.1 Primitive Data Structures

These are the basic data structures that are directly operated on by machine / CPU instructions. They are the atomic data type, i.e. they cannot be divided further

1.1.1.2 Non-Primitive Data Structures

These are data structures composed of or derived from primitive data structures.

1.1.1.3 Linear Data Structures

These data structures are characterized by the fact their elements can be accessed in a sequential / linear manner. This means every element, excluding the first and last elements which only have a successor and predecessor respectively, has both a predecessor and successor, allowing bi-directional traversal.

1.1.1.4 Non-Linear Data Structures

These data structures are characterized by the fact that their elements are accessed based on some relationship / association between elements, therefore non-linearly.

1.2 Non-Primitive Data Structures

1.2.1 Arrays

Definition 1.2.1: Array

An indexed collection of a fixed number of homogeneous data elements. Elements are stored in contiguous memory, i.e. sequentially and can be accessed by their index.

1.3 Generics

Definition 1.3.1: Type Parameter

A placeholder for a type that is used in the definition of a generic type.

Definition 1.3.2: Generic Type

A type that is defined with one or more type parameters, i.e., `<T>`. These type parameters are used to define the type of the data that the generic type can store.

Definition 1.3.3: Generics Class and Method

A class ,interface or method that is defined with one or more type parameters.

Generics enable classes and functions to be type safe yet reusable for multiple data types. Generic types also resolve type-casting problems, for example when a class is defined to store integers, it cannot store any other data type. However, with generics, the class can be defined to store any data type.

Definition 1.3.4: Type Safety

Prevents the program from compiling if the data type of the variable is not compatible with the data type of the object or if an unsupported operation is performed on the object. / Guarantees that the data type of the variable is compatible with the data type of the object.

1.3.1 Wildcards

Definition 1.3.5: Wildcard Character

The wildcard character, `<?>` , is used to denote that a generic type can be of any type, with the usual restrictions of parametrized types still applying.

A parametrized wildcard type on its own is called an unbounded wildcard type and essentially means the same as the upper bounded wildcard type `<? extends Object>`. When unbounded wildcard types in methods act similar to an equivalent generic method, for example:

```
public void print(ArrayList<?> list) {
  for (Object obj : list) System.out.println(obj);
}

public <T> void print(ArrayList<T> list) {
  for (T t: list) System.out.println(t);
}
```

These two methods perform exactly the same and work with any parametrized type.

```
public void printZip(ArrayList<?> first, ArrayList<?> second) {
    for (int i = 0; i < first.size(); i++) {
        System.out.println(first.get(i) + " " + second.get(i));
    }
}

public <T> void printZip(ArrayList<T> first, ArrayList<T> second) {
    for (int i = 0; i < first.size(); i++) {
        System.out.println(first.get(i) + " " + second.get(i));
    }
}
```

This listing shows the main difference in the use of normal parametrized types and wildcard types. The wildcard implementation can accept two difference types for each of the ArrayLists but the same type `T` is used for both lists in the generic method implementation.

1.3.1.1 Upper Bounded Wildcards

Definition 1.3.6: Upper Bounded Wildcard

A wildcard parametrized type that only accepts types of the bounded type or any subclass of the bounded type, denoted by:

`<? extends T>`

```
public double sum(ArrayList<? extends Number> numbers) {
    double sum = 0;
    for (Number number : numbers) sum += number.doubleValue();
    return sum;
}
```

In this listing the method `sum` only accepts an ArrayList of any subclass of the `Number` class, i.e. `Integer`, `Double`, `Float`, etc.

1.3.1.2 Lower Bounded Wildcards

Definition 1.3.7: Lower Bounded Wildcard

A wildcard parametrized type that only accepts types of the bounded type or any superclass of the bounded type, denoted by:

`<? super T>`

```
public int sum(ArrayList<? super Integer> numbers) {
    int sum = 0;
    for (int i = 0; i < 10; i++) {
        sum += numbers.get(i);
    }
    return sum;
}
```

In this listing the method `sum` on accepts an ArrayList of type `Integer`, or superclasses, `Number` and `Object`.

Chapter 2

Abstract Data Types

Definition 2.0.1: Abstract Data Type

Defined by the operations that can be performed on it, its representation and the values that can be stored in it.

Example 2.0.1

An Integer ADT is defined below:

- Operations: Addition, Subtraction, Multiplication, Division
- Representation: Binary, Decimal, Hexadecimal
- Values: -∞ to ∞

Algorithm 1 Is-Empty (A, n)

```
1: for i:= 1 to n do
2:     if A_i ≠ NULL then
3:         return false
4:     end if
5:     return true
6: end for
```

Chapter 3

Lists

3.1 Introduction

A list is a linearly ordered data structure that stores elements sequentially. The locations of elements in this data structure can be described by their index, which is an integer value that starts at 0. By this definition the first element in an list is at the 0th index and the last element is at the $n-1$ index, where n is the number of elements in the list.

3.2 Abstract Data Type

3.2.1 Operations

The List ADT is defined by the following operations:

Size - Returns the number of elements in the list

IsEmpty - Returns true if the list is empty, false otherwise

Get - Returns the element at the specified index

Set - Replaces the element at the specified index with the specified element

Add - Adds the specified element to the end of the list or at a specified index

Remove - Removes the element at the specified index, moving all subsequent elements one index to the left to fill the gap left by the removed element.

3.2.2 Representation

A list can be represented in two ways:

- Array Based
- Linked List Based

With linked lists discussed in chapter 4.

3.3 Array List

In representing a list using an array a common notion is to use an array A, where $A[i]$, stores a reference to the element with index i in the list. This representation assumes that the array is fixed in size, i.e. the array has a fixed capacity. This means that the array has a maximum number of elements that it can store, and if the number of elements in the list exceeds this capacity, a new array is created with a larger capacity and the elements are copied over to the new array. This process is known as resizing the array, and would be a $O(n)$ time and space complexity operation as all the elements in the list would have to be copied over to the new array.

This means that the elements in the list are stored in contiguous memory making the *Set* and *Get* operations operate in $O(1)$ time complexity due to the random access nature of arrays.

This also means that the *Add* and *Remove* operations operate in $O(n)$ time complexity worst case. In the worst case for the *Add* operation a new element is added at index 0. This requires that all the elements currently in the array would have to be shifted one index to the right to make space for the new element, i.e. taking n operations. Although in the best case, adding to the end of the list, the *Add* operation operates in $O(1)$ as the new element is added to the end of the list and no shifting is required. Similarly the *Remove* operation operates in $O(n)$ time complexity in the worst case where the first element is removed, requiring all the elements to be shifted one index to the left to fill the gap left by the removed element, and also $O(1)$ time complexity in the best case where the last element is removed.

The algorithms for the operations of the array list are shown below:

Algorithm 2 Get (A, n, i)

```
1: if i ⩽ 0 or i > n then                    ▷ Check if index is out of bounds
2:     return "Index out of bounds"
3: end if
4: return A_i
```

Algorithm 3 Set (A, n, nE, i)

```
1: if i ⩽ 0 or i > n then                    ▷ Check if index is out of bounds
2:     return "Index out of bounds"
3: end if
4: result = A_i
5: A_i = nE
6: return result
```

Algorithm 4 Add (A, n, e, i)

```
1: for k = n down to i do                    ▷ Shift elements to the right of index i to the right
       A_{k+1} = A_k                         ▷ Shift elements to the right
2: end for
3: A_i = e                                   ▷ Add element e at index i
4: n = n + 1
```

Algorithm 5 Remove (A, n, i)

```
1: if i ⩽ 0 or i > n then                    ▷ Check if index is out of bounds
2:     return "Index out of bounds"
3: end if
4: result = A_i
5: for k = i to n do
       A_k = A_{k+1}
6: end for
       A_n = null
7: n = n − 1
8: return result
```

Chapter 4

Linked Lists

Chapter 5

Stacks

Chapter 6

Queues

Chapter 7

Trees

Chapter 8

Graphs

8.1 Introduction

Definition 8.1.1: Graph

A non-linear data stricture that consists of a finite set of vertices and a set of edges that connect the vertices. A graph G is an ordered pair of a set of vertices V and a set of edges or arcs E.
Each edge can be an ordered / directed or unordered / undirected pair of vertices. An ordered pair is indicated as (u, v), and an unordered pair is indicated as $\{u, v\}$, where u and v are vertices.

Graph theory is further discussed in the Discrete Math topic in the same computer science topic.

8.1.1 Applications of Graphs

A graph can be used to model any collection with pairwise relationships.

- Social Networks
- Links between web pages
- Road Connections

8.1.2 Terminology

Definition 8.1.2: Path

A sequence of vertices, where each adjacent pair is connected by an edge. In a directed graph the direction must be aligned.
A simple path is a path in which no vertices and thus edges are repeated. The term walk is usually used to denote paths in general while path is used to indicate simple paths. Thus a walk in which no vertices are repeated is called a path and one where vertices are repeated is called a **Trail**

Definition 8.1.3: Strength

A directed graph is *strongly connected* if there is a path from any vertex to any other vertex.
If a graph is undirected we simply call it a *connected* graph.
If a directed graph is not strongly connected but can be turned into a connected graph by treating all edges as undirected it is termed a *weakly connected* graph

8.2 Abstract Data Type

Number of Vertices Returns the number of vertices in a graph

Vertices Returns an iteration of all the vertices in a graph

Number of Edges Returns the number of edges in a graph

Edges Returns an iteration of all the edges in a graph

Get Edge Returns the edge from a vertex u to a vertex v.

End Vertices Returns an iteration containing the two endpoint vertices of an edge e.

Opposite For an edge e incident to a vertex v, returns the other vertex of the edge

Out Degree Returns the number of outgoing edges from a vertex v

In Degree Returns the number of incoming edges from a vertex v

Out Going Edges Returns an iteration containing all the outgoing edges from a vertex v

Incoming Edges Returns an iteration containing all the incoming edges from a vertex v

Insert Vertex Inserts a vertex v into the Graph

Insert Edge Inserts an edge e between two vertices u and v

8.3 Representation