Data Structures & Algorithms

Week 2

Collections

Lecturer: Dr. Msagha J Mbogholi, PhD

Flashback from Lesson 1

- Variables that are associated with programming include constants, global, instance, local and class variables.
- A class variable is one that is available to the whole class in which it is declared. In Java it is normally declared under the class name, outside of all methods and constructors; further only one copy of the class variable per class. The same is true in Python
- An abstract data type (or ADT) is a programmer-defined data type that specifies a set of data values and a collection of well-defined operations that can be performed on those values.
- All data structures store a collection of values, but differ in how they organize the individual data items and by what operations can be applied to manage the collection. The choice of a particular data structure depends on the ADT and the problem at hand.
- In the traditional study of algorithms, there are two main criteria for judging the merits of algorithms: correctness (does the algorithm give solution to the problem in a finite number of steps?) and efficiency (how much resources (in terms of memory and time) does it take to execute them).
- Three types of analysis of algorithms in terms of time: worst case, average case, and best case. These are all expressed in mathematical notation based on the algorithm under investigation.

Content

- Introduction
- Linear collections
- Non-linear collections

Part 1

Introduction

Introduction

- Supposing you have built (or bought) your ideal home in a neighborhood you are content with. Your home will have a number of bedrooms, living room, kitchen, dining room, and so on. One space I doubt you would not miss to have is a store or shed.
- In the store you will probably keep stuff that you can't keep anywhere else in the house. This might include tools (for DIY chores), crates for holding drinks, parts, and so on. How do you go about arranging these in the store? What criteria do you use?
- I don't know about you, but one criteria that most people use is ease of retrieval. That is to say, you would wish to arrange items in a manner that will allow for you to reach and retrieve them with minimal effort; not so?
- This is one of the key training areas for people who work in commercial stores (including supermarkets). There has to be a way in which to arrange goods in the store area.

Introduction

- Let's begin with how you would go about this in your personal space (store in your home). You would want stuff that you use on a regular basis to be at the front (if space is limited), while what you don't use so regularly will be at the back.
- In the case of drinks you would like crates with full bottles be at the top of the stack, while those with empty bottles to be at the bottom. You would also like to find some way of arranging like things (like tools) in one area, and so on.
- This is the same scenario with commercial stores (yours truly was a stores manager once upon a time you know). There are two ways items can be organized here; according to use, or according to manufacturer. The former appears to make more sense though. For example in a manufacturing plant, consumables would be in one shelf area, while a separate area would be designated for paint, another area for obsolete items (those that have lost value since they haven't been used over the years), and so on.
- In supermarkets the logic is also to arrange according to use; thus, you will find kitchen cooking oils in one shelf (area), beverages in another, washing detergents in another, and so on. Further it can be noticed that anything to do with bathroom consumables will also be in one area (bathing soap, oils, hair oils, shaving cream, and so on).

Introduction

- This is more so since the human mind is always processing in logical order, and would associate certain items with others (such as bathing soap and shaving cream), and thus arranging in this way makes it easier for the shoppers. That's one of the many factors that makes shoppers prefer one supermarket chain to another (ease of use).
- Ok! How does this relate to our course you wonder?
- This is exactly the logic behind data structures, as introduced in lesson 1. It makes a lot of sense to logically group data together for the computer as well. The arrangement of logic used by applications is also based on human intuition. We like to have our data organized in some kind of order so that it makes sense to us and other users.
- In this lesson we introduce the term 'collection', and go on to describe the different classes of collections; of course these will be discussed in greater detail in lessons to come. However, in this lesson we would like to discuss the types and how they are related to each other. In later lessons we shall tackle each one individually.

Introduction

- Many authors do not differentiate between a collection and a data structure; thus the two terms are used interchangeably. However, in the context of a language such as Java the distinction is clear; this is the case with other programming languages as well (admin, 2016):"
- A data structure is how information is represented – the structure the data has with your machine's memory. As opposed, the term 'collection' identifies just how the data could be accessed.
- This distinction is essential since the choices you will be making about data storage may affect the performance of your application. Sorting a linked list, by way of example, will likely be slower than sorting an array list. That's why it's essential to find the right data structure for the task.
- Collection representations can be used to access your data – you can use an index, or step throughout the data with all the getFirst, getPrev, or some other commands, as an example."
- Let us now examine the different types of collections in the remainder of this lesson.

Part 2

Linear collections

2.1 Introduction

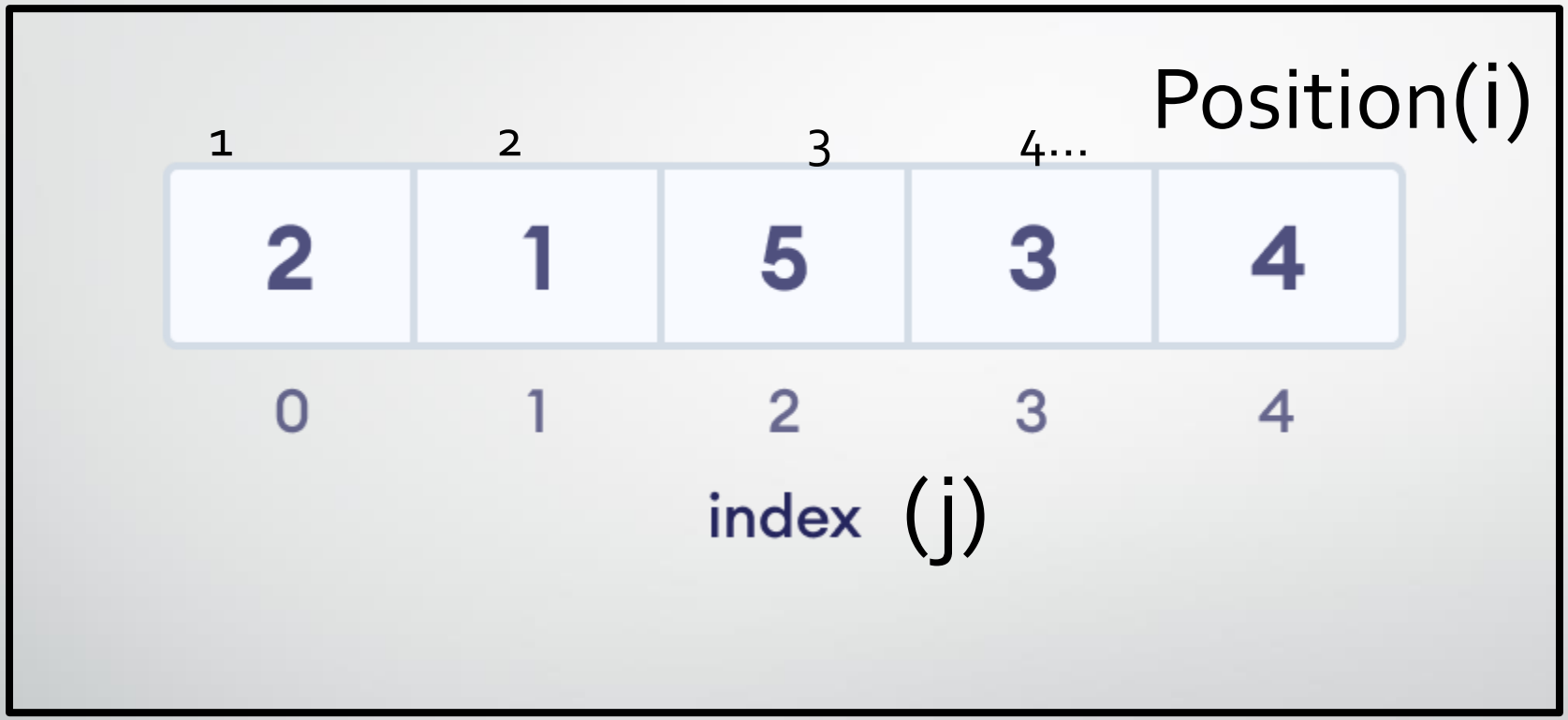
- "A **linear data structure** has data elements connected to each other so that elements are arranged in a sequential manner and each element is connected to the element in front of it and behind it. This way, the structure can be traversed in a single run. Linear data structures can be implemented easily as computer memory is also arranged in a linear manner." (Samad, n.d.).
- There are four linear data structures that are identified currently:
  - Array
  - Linked list
  - Stack
  - Queue

2.2 Array

- In an array, elements in memory are arranged in continuous memory. All the elements of an array are of the same type. And, the type of elements that can be stored in the form of arrays is determined by the programming language. (Programiz, n.d.)
- Arrays are declared differently depending on the language syntax; however, they are normally accessed in the same manner in all languages. The use of arrays works well depending on the size of the data and complexity of operations of the program.
- When the operations required become complex then a different collection will have to be used.
- The array elements are referred to using the index value starting from index 0 (first element) to index j (the last element which we can call i), but the position is referred to starting from the 1st to the ith element. Thus the ith element is at index j; thus mathematically we can see that i = (j – 1), or j = (i + 1). For example the element at position 1 (i) of the array is in index 0 (j). Figure 1 illustrates this in a simple manner.

Figure 1

Representation of elements in an array



Adapted from (Programiz, n.d.)

Table 1

Python operations on arrays

Method	Description
array(data type, value list)	Used to create an array with data type and value list specified in its arguments.
append()	Used to add the value mentioned in its arguments at the end of the array.
insert(i, x)	Used to add the value(x) at 'i' position.
pop()	Removes the element at the position mentioned in its argument and returns it.
remove()	Used to remove the first occurrence of the value mentioned in its arguments.
index()	Returns the index of the first occurrence of value mentioned in its arguments.
reverse()	Reverses the array.

(From Samad, nd)

Table 2

Java operations on arrays

S/No	Method & Description
1	**public static int binarySearch(Object[] a, Object key)** Searches the specified array of Object (Byte, Int , double, etc.) for the specified value using the binary search algorithm. The array must be sorted prior to making this call. This returns index of the search key, if it is contained in the list; otherwise, it returns (– (insertion point + 1)).
2	**public static boolean equals(long[] a, long[] a2)** Returns true if the two specified arrays of longs are equal to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. This returns true if the two arrays are equal. Same method could be used by all other primitive data types (Byte, short, Int, etc.)
3	**public static void fill(int[] a, int val)** Assigns the specified int value to each element of the specified array of ints. The same method could be used by all other primitive data types (Byte, short, Int, etc.)
4	**public static void sort(Object[] a)** Sorts the specified array of objects into an ascending order, according to the natural ordering of its elements. The same method could be used by all other primitive data types (Byte, short, Int, etc.)

(From (*Java - Arrays - Tutorialspoint*, n.d.)

2.2 Arrays (cont'd)

- Table 1 shows the different operations that can be performed on arrays in the Python programming language. Let us examine a small snippet of code:
- import array
- arr=array.array('i',[1,2,3])
- print(arr)
- This code allows us to import the array library; we then create an array called 'i' with elements 1, 2, and 3 respectively. Lastly we print the array. The output will be as follows:
- array('i', [1, 2, 3])

Also:

```
print("The new created array is : ", end=" ")
for i in range(0, 3):
    print(a[i], end=" ")
print()
```

This will print the array elements in order. Not to worry, we shall look at more code in lesson 4.

2.2 Arrays (cont'd)

- Table 2 describes some of the operations that can be performed in Java on arrays. Let us examine a small program snippet implementing these:
- public static void main(String[] args) {
- double[] myList = {1.9, 2.9, 3.4, 3.5};
- // Print all the array elements
- for (int i = 0; i < myList.length; i++) {
- System.out.println(myList[i] + " ");
- This code creates the array myList and prints out all the array elements…more of this in lesson 4.

2.2 Arrays

- How else can we use an array in a practical situation? Smit(2016) describes an ideal situation: "..let us say, we want to store marks of all students in a class, we can use an array to store them. This helps in reducing the use of a number of variables as we don't need to create a separate variable for marks of every subject. All marks can be accessed by simply traversing the array. "
- As we have not yet covered complexity in detail it is just worth mentioning some parameters of interest regarding arrays; however, these will be revisited once the topic has been covered in lesson 3:
- If an array is of size n, then Accessing Time is O(1) [This is possible because elements are stored at contiguous locations]. Search Time is O(n) for sequential search; O(log n) for binary search if the array is sorted; insertion time is O(n) [The worst case occurs when insertion happens at the Beginning of an array and requires shifting all of the elements]; deletion time is O(n) [The worst case occurs when deletion happens at the beginning of an array and requires shifting all of the elements] (Smit, 2016)

2.2.1 Advantages

- Smit (2016) also describes the advantages of arrays:
- Constant-time Access: Arrays allow for constant-time access to elements by using their index, making it a good choice for implementing algorithms that need fast access to elements.
- Memory Allocation: Arrays are stored in contiguous memory locations, which makes the memory allocation efficient.
- Easy to Implement: Arrays are easy to implement and can be used with basic programming constructs like loops and conditionals.

2.2.1 Advantages

- Prasanna (2023) adds the following advantages:
- Code Optimization: An array allows storing and access of a large number of values by writing a small piece of code instead of declaring each variable separately.
- Functionality: Arrays are one of the most basic data structures and are used for processing many algorithms like searching and sorting, maximum and minimum values, reversing, etc. in simple and easy ways.
- Multi-dimensional: Arrays can handle complex data structures by storing elements of a matrix in 2-dimensional arrays.
- Multiple Uses: The basic data structure of arrays can be used to form different data structures like stacks, queues, graphs, trees, etc.

2.2.2 Disadvantages

- Prasanna (2023) describes the following disadvantages of arrays:
- Size is fixed: An array is static in the sense that the size of an array is fixed. The memory allocated to an array cannot be expanded or shrunk. This does not allow storing extra data in case of any requirement. This may sometimes lead to loss of data if the allocated memory to an array is less than required.
- The problem in expansion: If array size is required to be increased during the development process at a later stage, then the only option is to discard the present array and create a new array of an enlarged size that meets the requirement. But the possibility of changing the size again still exists as the application grows.
- Memory wastage: The size of an array is declared at the beginning based on some requirement that can't be changed. So the memory is allocated accordingly. But if at a later stage the size requirement becomes less then memory wastage happens. For example, for a declared array size of 50, if we get only 38 elements to store, there is wastage of 12 elements storage space.

2.2.2 Disadvantages

- Limitation of type of data: A single array is homogeneous which means only one type of data can be stored in one array but cannot store values of different data types. But in real-life scenarios, we may require storing of elements of different types such as string (student name) integer (roll number), etc. which is not possible.
- Operational limitation: As arrays store data in contiguous memory locations, it becomes difficult to carry out the deletion and insertion operations in arrays. It also involves a lot of time as we need to shift other elements one position ahead or back respectively.
- Memory space: At the beginning of the setting of arrays, developers often declare large array sizes to be on the safe side to avoid any problem arising out of any future data expansion. This leads to the creation of large arrays occupying much space.
- Index bound checking: In C language, if any code is written that is outside the range of index values of an array, the compiler does not perform index bound checking or signal any error. But at the time of access to the data the compiler shows a run time error or gives garbage value.

2.3 Linked lists

- The second type of linear data structure is a linked list.
- "A linked list is a linear data structure (like arrays) where each element is a separate object. A linked list is made up of two items that are data and a reference to the next node. A reference to the next node is given with the help of pointers and data is the value of a node. Each node contains data and links to the other nodes. It is an ordered collection of data elements called a node and the linear order is maintained by pointers. It has an upper hand over the array as the number of nodes i.e. the size of the linked list is not fixed and can grow and shrink as and when required, unlike arrays." (Smit, 2016)
- Figure 2 and Figure 3 show how elements in a linked list are connected: Nodes represent the data elements, and links or pointers connect each node. The head is the beginning of the linked list, while the last node contains a NULL value in the pointer field as it doesn't point to anything.

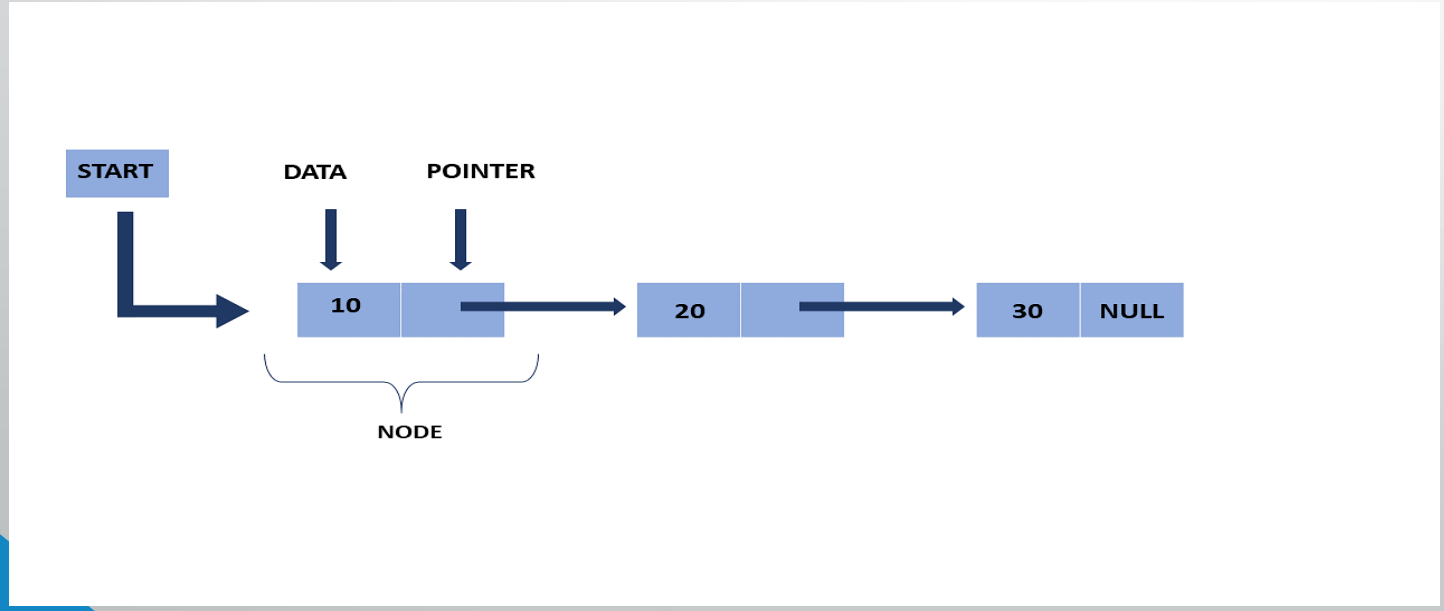
Figure 2

Elements of a linked list

(From Programiz, n.d.)

Figure 3

Data and pointer of a linked list



(From Ravikiran, 2023)

2.3 Linked lists

- Let us examine some Java code that creates a linked list and populates it with four elements (code source: https://www.javatpoint.com/java-linkedlist) :

```
import java.util.*;
public class LinkedList1{
 public static void main(String args[]){
   LinkedList<String> al=new LinkedList<String>();
 al.add("Ravi");   al.add("Vijay");   al.add("Ravi");   al.add("Ajay");
 Iterator<String> itr=al.iterator();
 while(itr.hasNext()){
  System.out.println(itr.next());   }  }   }
```

2.3 Linked lists

- There are three types of linked lists:
  - Singly linked list
  - Doubly linked list
  - Circular linked list
- Let us describe each of these so we can understand the differences and how they are used.

2.3.1 Singly linked list

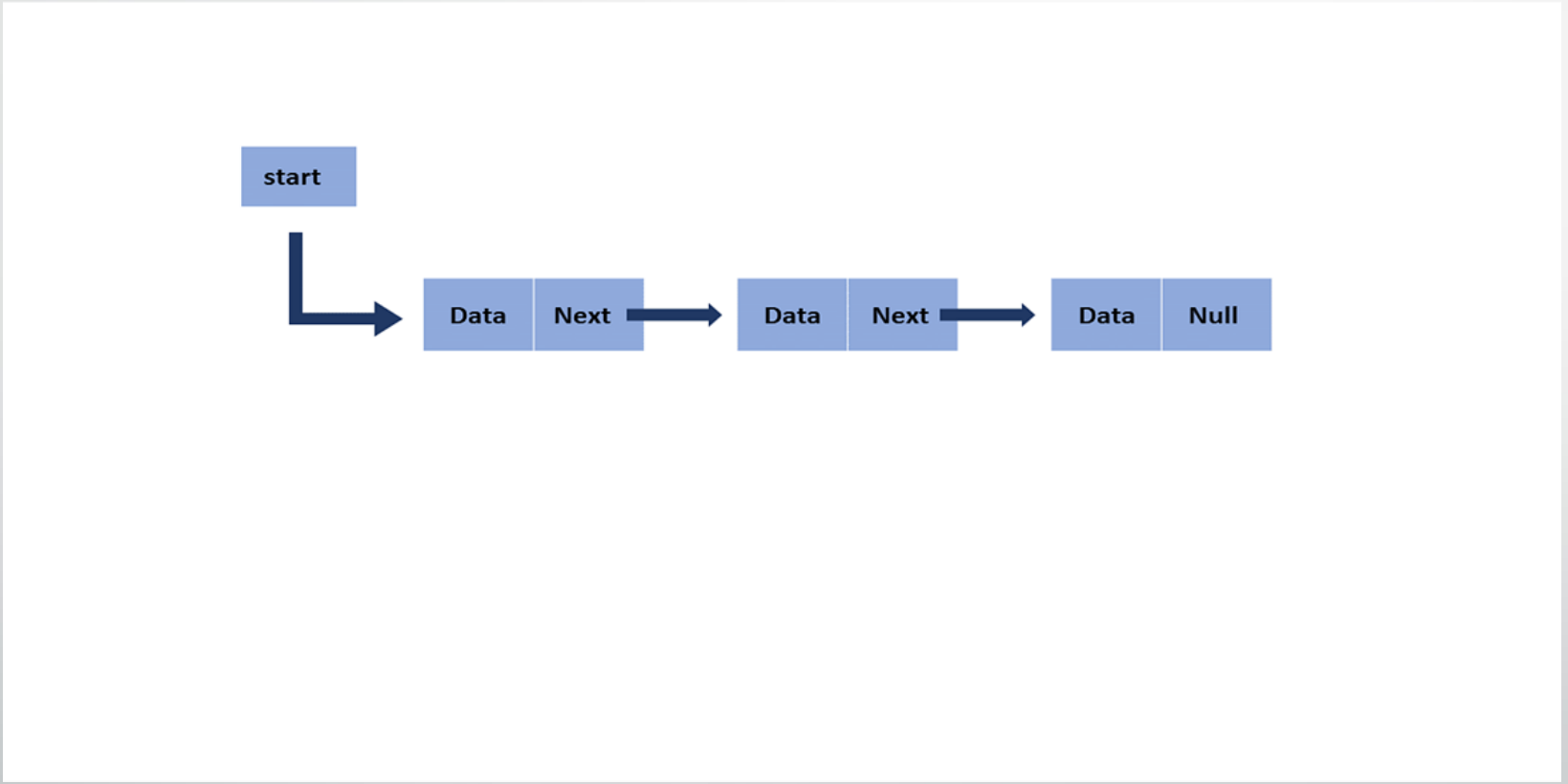
- This is the most used of the three; in this case a node stores the address of the next node in the list and it goes on from the first node storing the address of the second node, the second node storing the address of the third node, and so on. What happens with the last node, what address does it store? Well it stores NULL as the address of the next node. Figure 4 illustrates this simply. Advantages and disadvantages of singly linked lists have been described by (Course, n.d.):
- Advantages:
- **Dynamic size**: The size of a linked list can be changed dynamically, unlike arrays, which have a fixed size.
- **Easy insertion and deletion**: Singly Linked List allows easy insertion and deletion of nodes, as only the pointers need to be modified.
- **Efficient memory utilization**: Singly Linked List allows efficient memory utilization as nodes can be allocated as needed.

2.3.1 Singly linked list

- Disadvantages:
- **No random access**: Singly Linked List does not allow for random access of elements like arrays, as it requires sequential traversal from the head node.
- **Extra memory overhead**: Singly Linked List requires extra memory to store the link field for each node.

Figure 4

Single linked list



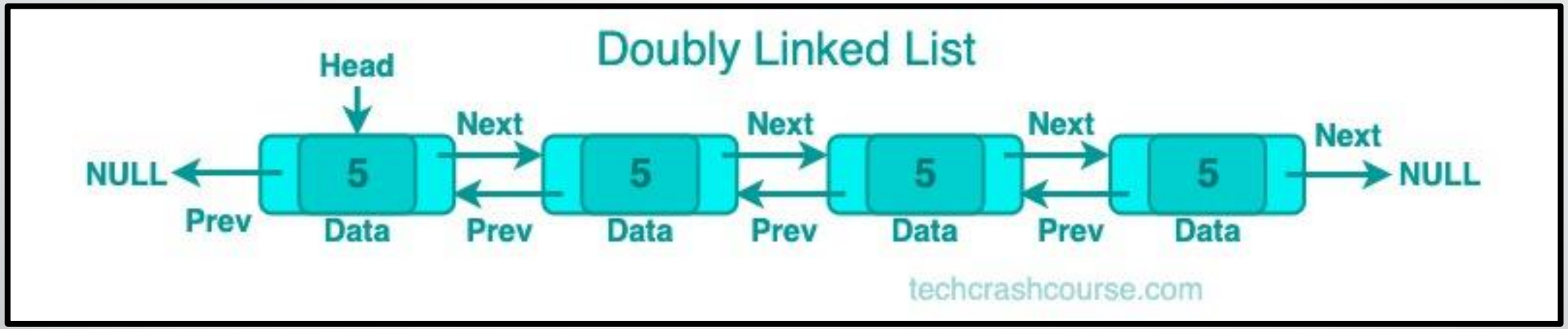
(From Ravikiran, 2023)

2.3.2 Doubly linked list

- "In this type of Linked list, there are two references associated with each node, One of the reference points to the next node and one to the previous node. The advantage of this data structure is that we can traverse in both directions and for deletion, we don't need to have explicit access to the previous node." (Smit, 2016)
- In a doubly linked list each node contains 3 fields as demonstrated in figure 5. the advantages of the doubly linked list apart from that described by (Smit, 2016), are efficiency in insertion and deletion of nodes, and they can be used to implement stack, queue, and deque data structures (Course, n.d.).
- On the other hand they present the following disadvantages (Course, n.d.): Extra space is required for the previous pointers, increasing the memory requirements; and, implementation is more complex than a singly linked list.

Figure 5

Doubly linked list



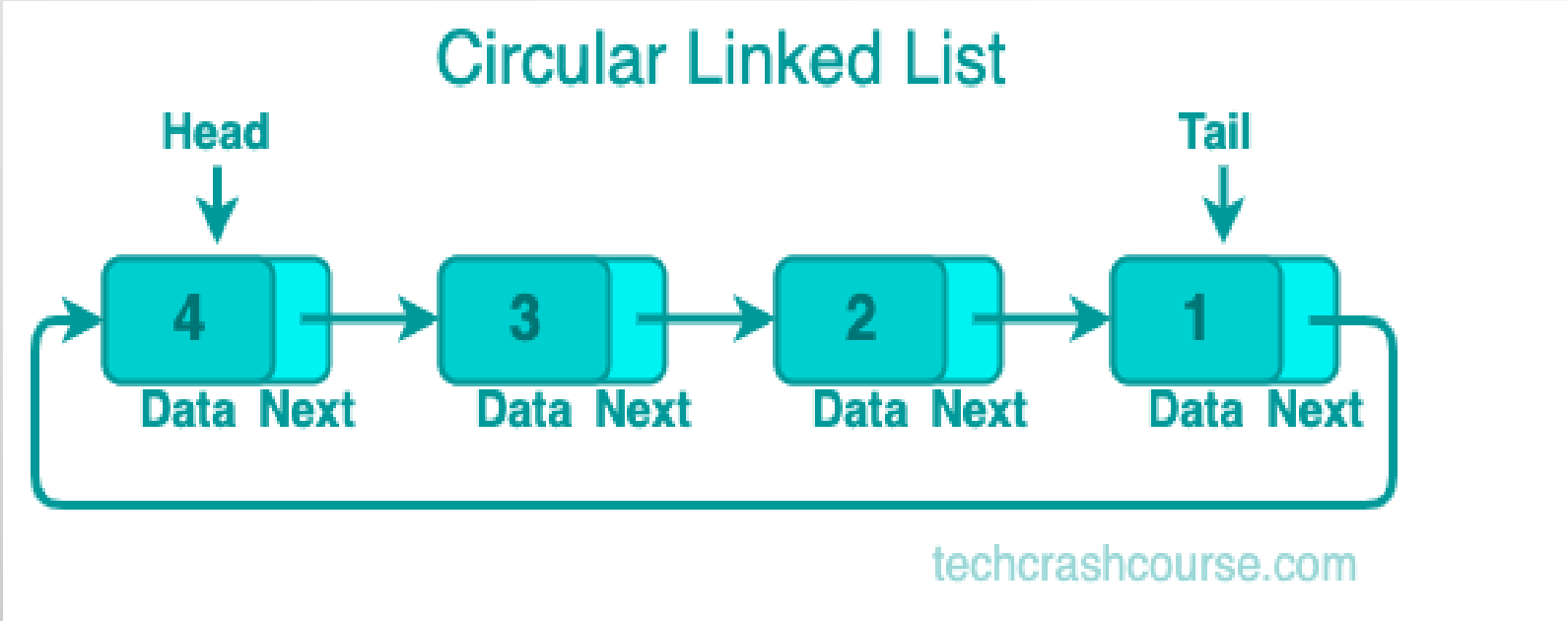
(From Course, nd)

2.3.3 Circular linked list

- A circular linked list is one where all nodes are connected to form a circle. There is no NULL pointer at the end. Thus, each node contains a reference to the next node in the list, and the last node in the list points back to the first node. This creates a circular loop that can be traversed infinitely.
- A circular linked list can emanate from a single linked list or a doubly linked list; these respectively will form a circular single linked list or a circular doubly linked list, respectively.
- According to Course (n.d.) the advantages of the circular linked list are: A circular linked list can be traversed indefinitely, and, insertion and deletion operations can be performed efficiently at any position in the list.
- Similarly, according to Course (n.d.) disadvantages of the circular linked list are: It is more complex to implement than a singly linked list; and, it requires extra memory to store the reference from the last node back to the first node.
- Figure 6 illustrates a circular linked list.

Figure 6

Circular linked list



(From Course, nd)

2.4 Stack

- A stack or LIFO (last in, first out) is an abstract data type that serves as a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the last element that was added. In stack both the operations of push and pop take place at the same end that is top of the stack. It can be implemented by using both array and linked list. (Smit, 2016)
- Further there are two types of stacks (Smit, 2016):
- **Register Stack:** This type of stack is also a memory element present in the memory unit and can handle a small amount of data only. The height of the register stack is always limited as the size of the register stack is very small compared to the memory.
- **Memory Stack:** This type of stack can handle a large amount of memory data. The height of the memory stack is flexible as it occupies a large amount of memory data.

2.4 Stack

- The operations that are commonly performed on the stack include (*Stack*, n.d.):
- **Push**: Add an element to the top of a stack
- **Pop**: Remove an element from the top of a stack
- **IsEmpty**: Check if the stack is empty
- **IsFull**: Check if the stack is full
- **Peek**: Get the value of the top element without removing it.

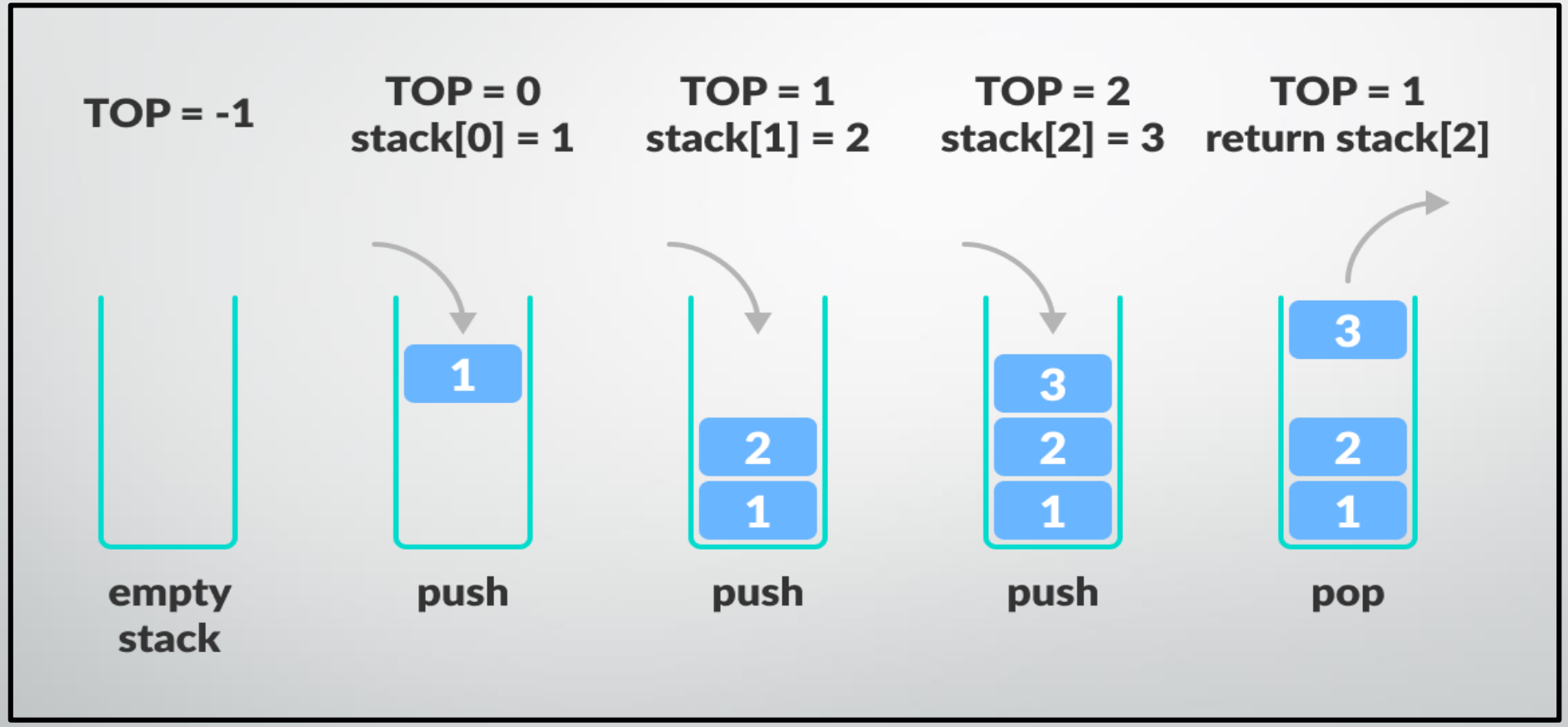
2.4 Stack

- The working of the stack data structure can be described as (Stack, n.d.), and illustrated in Figure 7:

A pointer called TOP is used to keep track of the top element in the stack.
2.When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.
3.On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
4.On popping an element, we return the element pointed to by TOP and reduce its value.
5.Before pushing, we check if the stack is already full
6.Before popping, we check if the stack is already empty.

Figure 7

Stack operation



(From Stack, n.d.)

2.4.1 Advantages

- LIFO (Last-In, First-Out) Order: Stacks allow for elements to be stored and retrieved in a LIFO order, which is useful for implementing algorithms like depth-first search.
- Efficient Operations: Stacks provide efficient push-and-pop operations, as only the top element needs to be updated.
- Easy to Implement: Stacks can be easily implemented using arrays or linked lists, making them a simple data structure to understand and use.
- (Smit, 2016)

2.4.2 Disadvantages

- Fixed Size: Stacks have a fixed size, so they can suffer from overflow if too many elements are added or underflow if too many elements are removed.
- Limited Operations: Stacks only allow for push, pop, and peek (accessing the top element) operations, so they are not suitable for implementing algorithms that require constant-time access to elements or efficient insertion and deletion operations.
- Unbalanced Operations: Stacks can become unbalanced if push and pop operations are performed unevenly, leading to overflow or underflow.
- In summary, stacks are a good choice for problems where LIFO order and efficient push and pop operations are important, but their disadvantages should be considered for problems that require dynamic resizing, constant-time access to elements, or more complex operations.
- (Smit, 2016)

2.4.2 Application areas

- Stacks are used in different areas. (Course, n.d.-b) describes the following:
- **Function call stack**: When a function is called, its return address and parameters are pushed onto the stack. When the function returns, these values are popped off the stack.
- **Expression evaluation**: In computer science, expressions are usually evaluated using a stack. For example, to evaluate the expression 2 * (3 + 4), we push 2 onto the stack, then push 3 and 4 onto the stack, then pop 3 and 4 off the stack and add them together, then multiply the result by 2.
- **Backtracking**: Stacks can be used to implement backtracking algorithms. In a backtracking algorithm, we keep track of the choices we have made so far on the stack. If we reach a dead end, we pop the choices off the stack until we find a choice that leads to a solution.
- **Undo/Redo operations**: Stacks can be used to implement undo and redo operations in software applications. Each operation is pushed onto the stack, and undoing an operation involves popping it off the stack.

2.4.2 Application areas

- (Programiz, n.d.) also describe some more application areas:

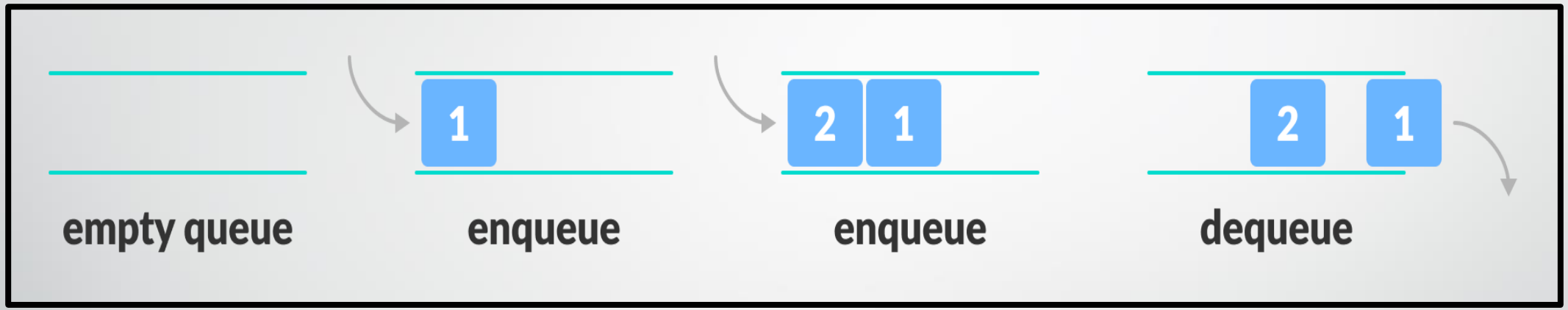
- To reverse a word - Put all the letters in a stack and pop them out. Because of the LIFO order of stack, you will get the letters in reverse order.
- In compilers - Compilers use the stack to calculate the value of expressions like 2 + 4 / 5 * (7 - 9) by converting the expression to prefix or postfix form.
- In browsers - The back button in a browser saves all the URLs you have visited previously in a stack. Each time you visit a new page, it is added on top of the stack. When you press the back button, the current URL is removed from the stack, and the previous URL is accessed.

2.5 Queue

- A queue is a linear data structure that can be explained using the simple analogy of queuing for food at a cafeteria for example. The first person in the queue gets to be served, and they also get to be the first one to leave it; (FIFO – First In First Out) to put it simply.
- Putting items in the queue is called enqueuing while removing items from the queue is known as dequeuing. Consequently, a queue is a data structure that is open at both ends (for enqueuing and dequeuing, respectively).
- In the computer queuing is used for operations such as disk scheduling, CPU scheduling, and asynchronous data transfer between processes such as file I/O.
- Queuing operations (apart from enqueue and dequeue) include IsEmpty() (check if the queue is empty), IsFull(check if the queue is full), and Peek (check value at the front of the queue without removing it. Figure 8 illustrates the enqueuing and dequeuing process.

Figure 8

Queueing operations



(From Programiz, n.d.)

2.5.1 Types

- Smit(2016) describes the different types of queues:
- **Simple Queue:** Simple queue also known as a linear queue is the most basic version of a queue. Here, insertion of an element i.e. the Enqueue operation takes place at the rear end and removal of an element i.e. the Dequeue operation takes place at the front end.
- **Circular Queue:** In a circular queue, the element of the queue act as a circular ring. The working of a circular queue is similar to the linear queue except for the fact that the last element is connected to the first element. Its advantage is that the memory is utilized in a better way. This is because if there is an empty space i.e. if no element is present at a certain position in the queue, then an element can be easily added at that position.

2.5.1 Types

- **Priority Queue:** This queue is a special type of queue. Its specialty is that it arranges the elements in a queue based on some priority. The priority can be something where the element with the highest value has the priority so it creates a queue with decreasing order of values. The priority can also be such that the element with the lowest value gets the highest priority so in turn it creates a queue with increasing order of values.
- **Dequeue:** Dequeue is also known as Double Ended Queue. As the name suggests double ended, it means that an element can be inserted or removed from both the ends of the queue unlike the other queues in which it can be done only from one end. Because of this property it may not obey the First In First Out property.

2.5.2 Advantages

- FIFO (First-In, First-Out) Order: Queues allow for elements to be stored and retrieved in a FIFO order, which is useful for implementing algorithms like breadth-first search.
- Efficient Operations: Queues provide efficient enqueue and dequeue operations, as only the front and rear of the queue need to be updated.
- Dynamic Size: Queues can grow dynamically, so they can be used in situations where the number of elements is unknown or can change over time.
- (Smit, 2016)

2.5.3 Disadvantages

- Limited Operations: Queues only allow for enqueue, dequeue, and peek (accessing the front element) operations, so they are not suitable for implementing algorithms that require constant-time access to elements or efficient insertion and deletion operations.
- Slow Random Access: Queues do not allow for constant-time access to elements by index, so accessing an element in the middle of the queue can be slow.
- Cache Unfriendly: Queues can be cache-unfriendly, as elements are retrieved in a different order than they are stored, which can lead to poor cache utilization and performance.
- (Smit, 2016)

Part 3

Non-Linear collections

3.1 Introduction

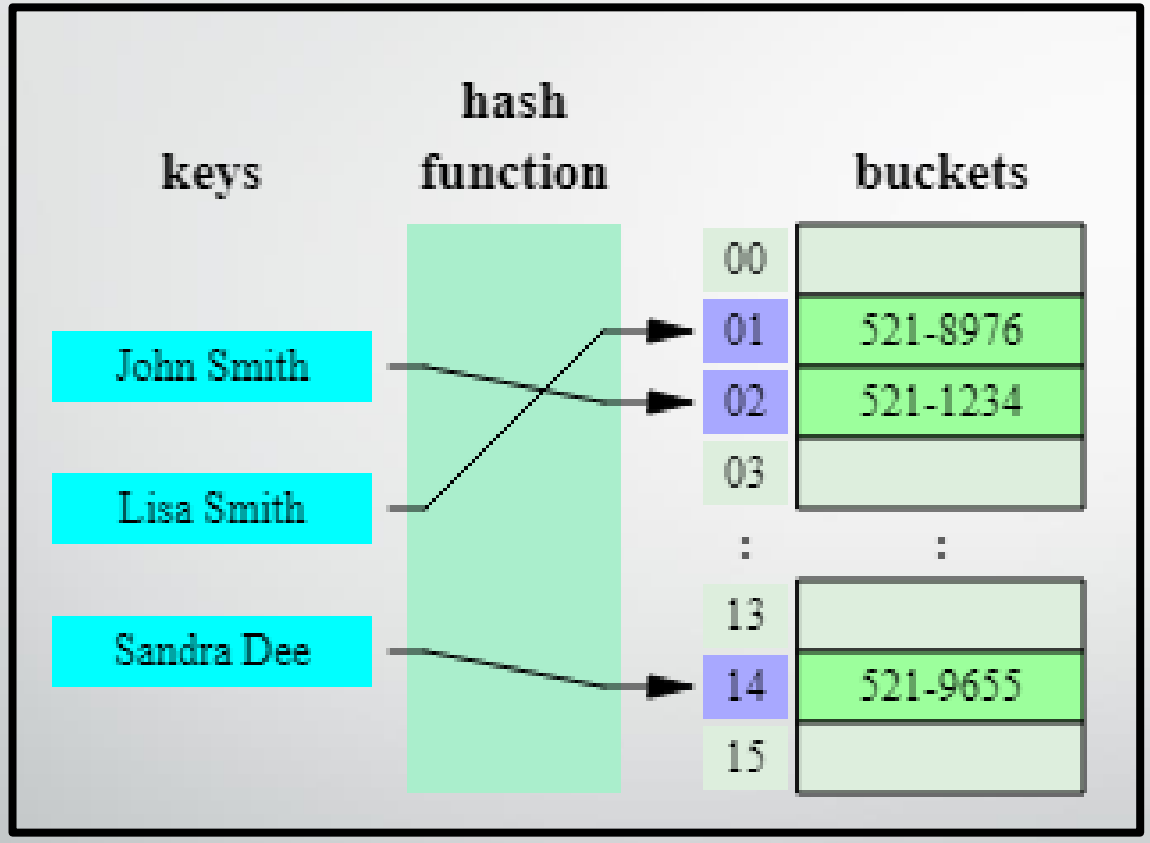
- We have discussed the linear data structures and explained why they are so called. There is a special data structure known as a hash table which can be implemented as linear or non-linear. This is described briefly under non-linear data structures. The non-linear data structures generally fall under two categories:
  - Trees
  - Graphs

3.2 Hash table

- (*Hash Tables*, n.d.) describes hash table as follows: "A hash table is a data structure where data is stored in an **associative** manner. The data is mapped to array positions by a **hash function** that generates a **unique** value from each key…The value stored in a hash table can be searched in **O(1)** time, by using the same hash function which generates an address from the key. The process of **mapping the keys to appropriate locations** (or indices) in a hash table is called **hashing**."
- Hashing offers the distinct advantage of speed over other data structures (note the O(1) which is faster than all the other data structures); meaning lookups are done very fast. These tables work best in situations where entries can be predicted in advance.
- Figure 9 demonstrates a simple hashing operation.

Figure 9

Hash table



(From Hash Tables, n.d.)

3.3 Trees

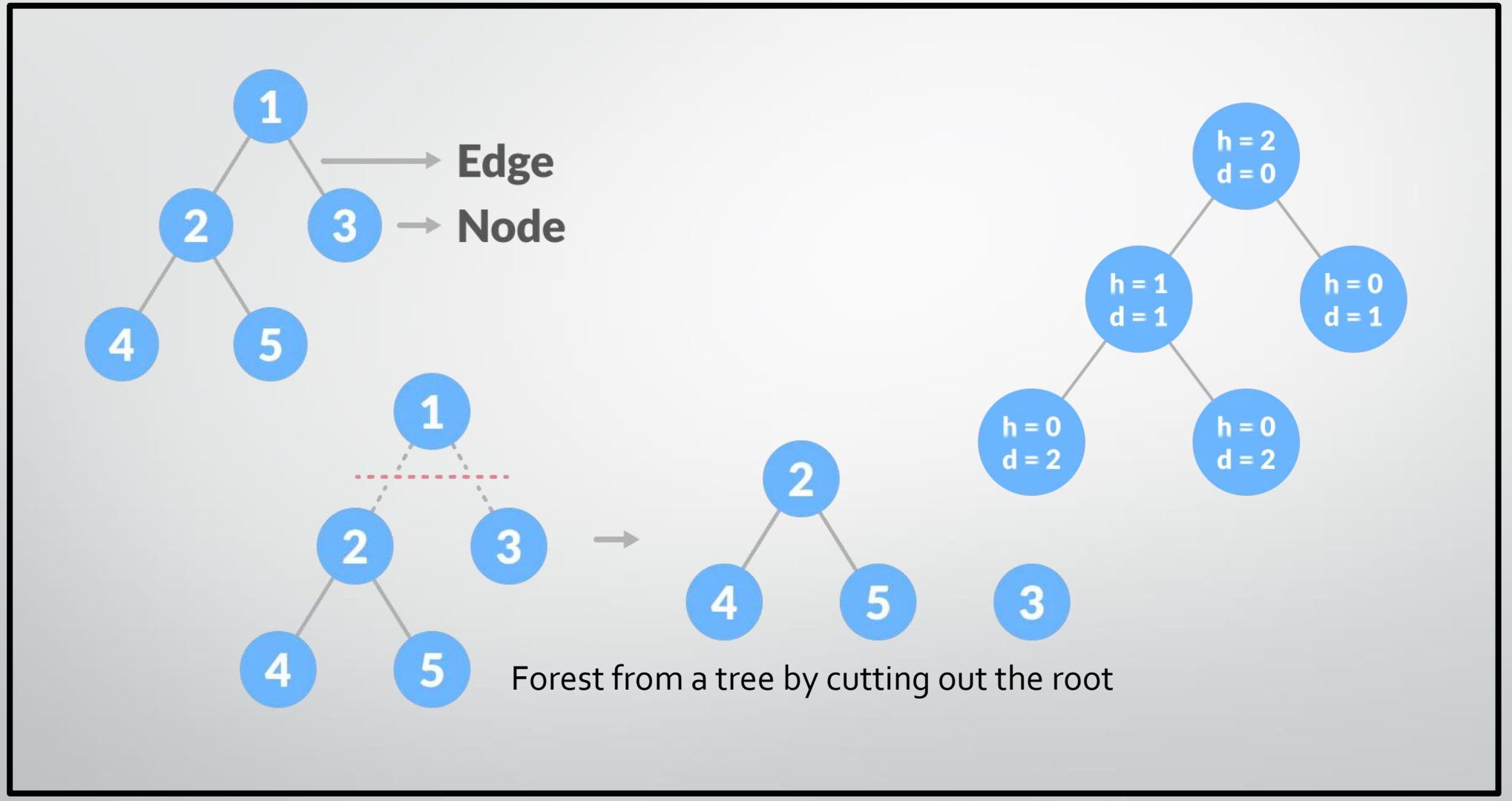
- A tree is a non-linear hierarchical data structure that uses nodes that are connected by edges. Terminologies associated with trees are (*Tree Data Structure*, n.d.):
- **Node :** A node is an entity that contains a key or value and pointers to its child nodes. The last nodes of each path are called **leaf nodes or external nodes** that do not contain a link/pointer to child nodes.
- The node having at least a child node is called an **internal node**.
- **Edge:** It is the link between any two nodes.
- **Root:** It is the topmost node of a tree.

3.3 Trees

- **Height of a Node:** The height of a node is the number of edges from the node to the deepest leaf (i.e., the longest path from the node to a leaf node).
- **Depth of a Node:** The depth of a node is the number of edges from the root to the node.
- **Height of a Tree:** The height of a Tree is the height of the root node or the depth of the deepest node.
- **Degree of a Node:** The degree of a node is the total number of branches of that node.
- **Forest:** A collection of disjoint trees is called a forest. You can create a forest by cutting out the root of a tree.
- The detailed operations on trees and their types will be discussed in detail in lesson 10.
- Figure 10 shows some of these terminologies under one 'roof'.

Figure 10

Tree terminologies



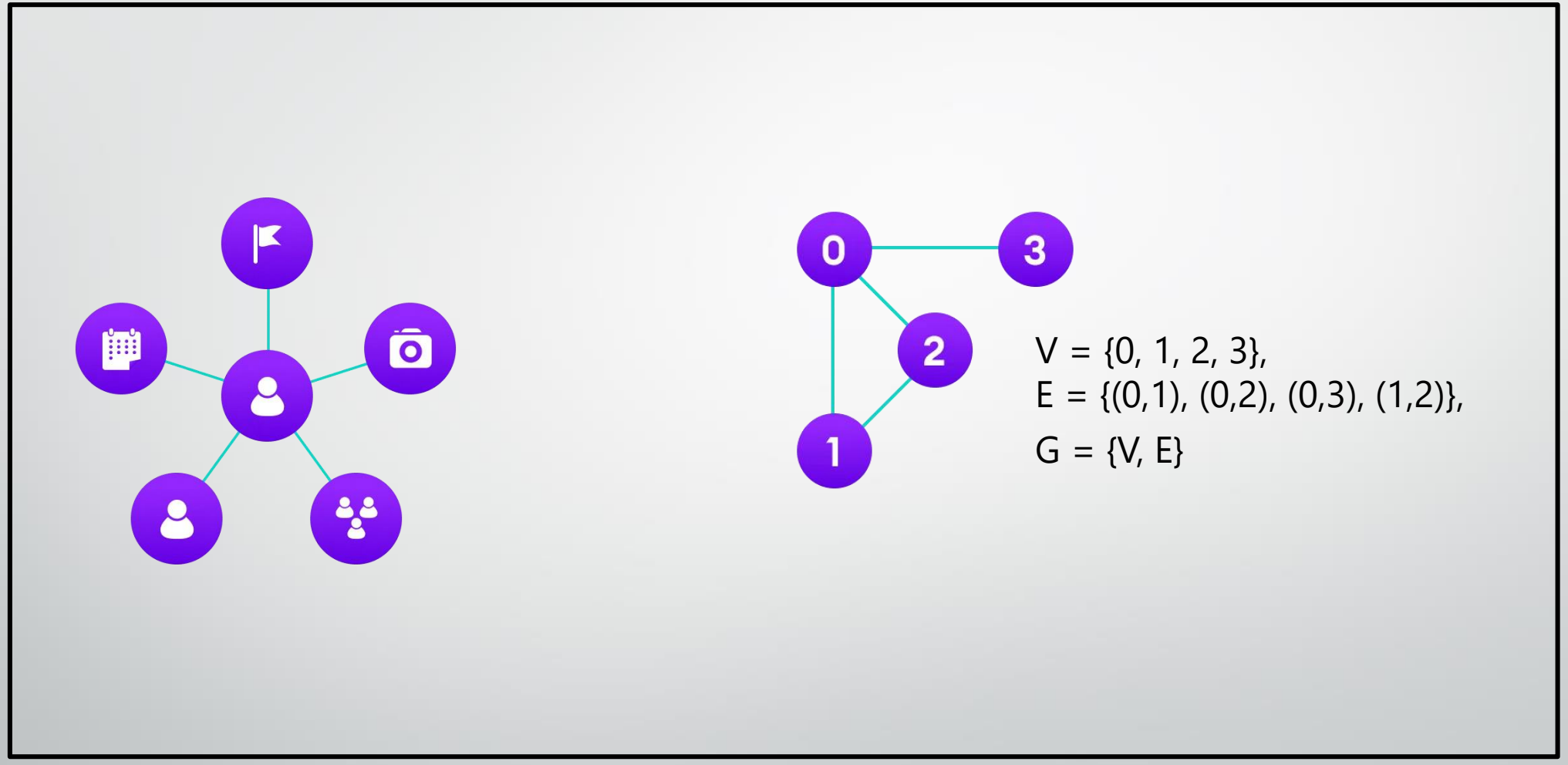
(From Tree Data Structure, n.d)

3.4 Graphs

- (*Graph Data Structure*, n.d.) provides a sufficient description of the graph data structure:
- "A graph data structure is a collection of nodes that have data and are connected to other nodes. Let's try to understand this through an example. On facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, Note…anything that has data is a node.
- Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page, etc., a new edge is created for that relationship.
- All of Facebook is then a collection of these nodes and edges. This is because Facebook uses a graph data structure to store its data.
- More precisely, a graph is a data structure (V, E) that consists of
  - A collection of vertices V
  - A collection of edges E, represented as ordered pairs of vertices (u,v)
- Figure 10 illustrates a graph data structure.

Figure 10

Graph data structure



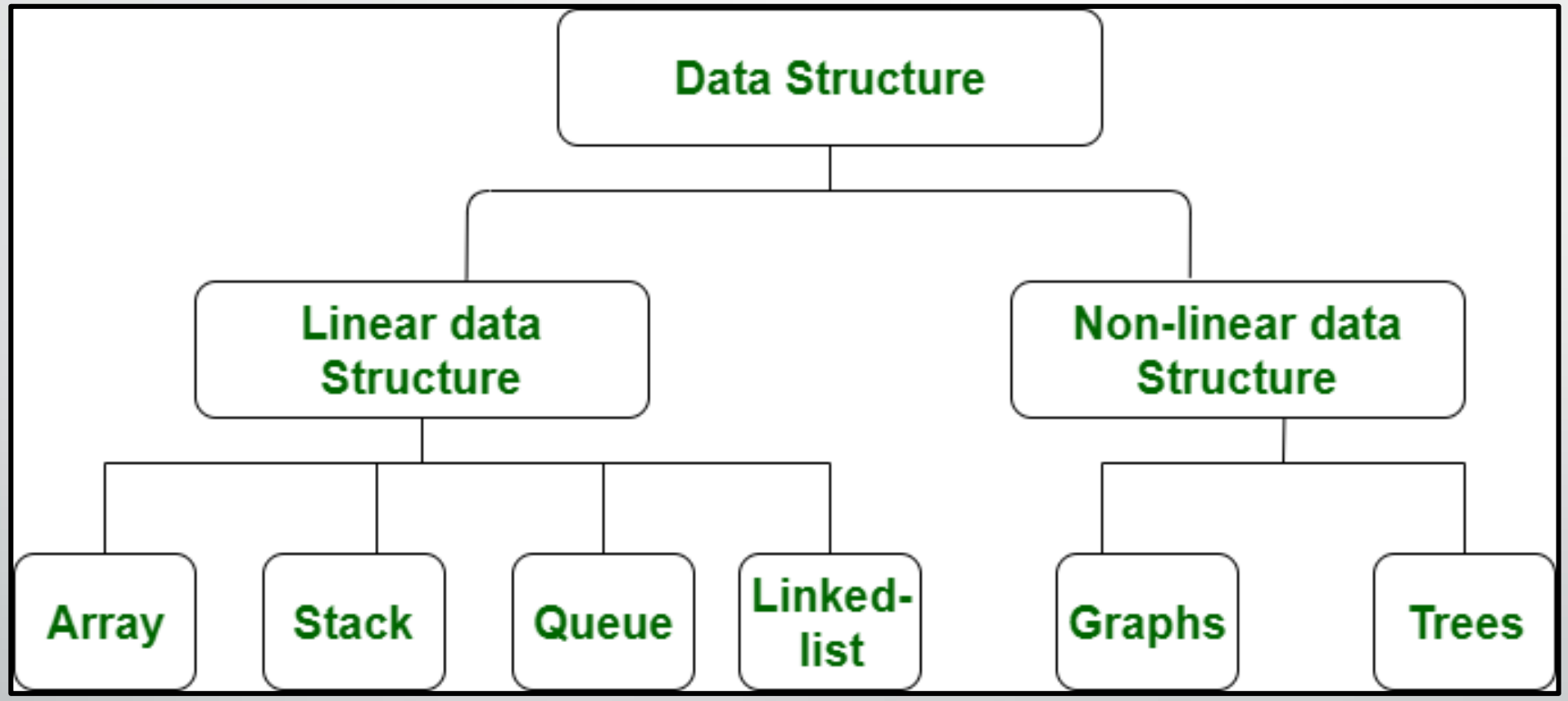
(From *Graph Data Structure*, n.d.)

3.4.1 Terminologies

- **Adjacency**: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
- **Path**: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
- **Directed Graph**: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.
- (*Graph Data Structure*, n.d.)
- Graph operations will be discussed in detail in lesson 12.
- Figure 11 summarizes the linear and non-linear data structures while Table 3 offers a comparison of the two.

Figure 11

Linear and non-linear data structures



(From *Difference between Linear and Non-Linear Data Structures*, 2019)

Table 3

Linear and non-linear data structures

S.NO	Linear Data Structure	Non-linear Data Structure
1.	In a linear data structure, data elements are arranged in a linear order where each and every element is attached to its previous and next adjacent.	In a non-linear data structure, data elements are attached in hierarchically manner.
2.	In linear data structure, single level is involved.	Whereas in non-linear data structure, multiple levels are involved.
3.	Its implementation is easy in comparison to non-linear data structure.	While its implementation is complex in comparison to linear data structure.
4.	In linear data structure, data elements can be traversed in a single run only.	While in non-linear data structure, data elements can't be traversed in a single run only.
5.	In a linear data structure, memory is not utilized in an efficient way.	While in a non-linear data structure, memory is utilized in an efficient way.
6.	Its examples are: array, stack, queue, linked list, etc.	While its examples are: trees and graphs.
7.	Applications of linear data structures are mainly in application software development.	Applications of non-linear data structures are in Artificial Intelligence and image processing.
8.	Linear data structures are useful for simple data storage and manipulation.	Non-linear data structures are useful for representing complex relationships and data hierarchies, such as in social networks, file systems, or computer networks.
9.	Performance is usually good for simple operations like adding or removing at the ends, but slower for operations like searching or removing elements in the middle.	Performance can vary depending on the structure and the operation, but can be optimized for specific operations.

(From *Difference between Linear and Non-Linear Data Structures*, 2019)

Summary

- Data structures can be classified as being either linear or non-linear.
- Data structure where data elements are arranged sequentially or linearly where each and every element is attached to its previous and next adjacent is called a linear data structure.
- Arrays, stacks, queues, and linked lists are examples of linear data structures.
- Data structures where data elements are not arranged sequentially or linearly are called non-linear data structures.
- Trees and graphs are examples of non-linear data structures.
- The hash table is a special data structure that can be classified as either linear or non-linear.

Reference List

- admin. (2016, August 18). *Javas Data Structures and Collections Explained*. On Hoops. http://www.onhoops.com/javas-data-structures-and-collections-explained/
- Course, T. C. (n.d.-a). *Singly Linked List Data Structure*. Https://Www.techcrashcourse.com/. Retrieved September 23, 2023, from https://www.techcrashcourse.com/2023/03/singly-linked-list-data-structure.html
- Course, T. C. (n.d.-b). *Stack Data Structure*. Techcrashcourse.com. Retrieved September 23, 2023, from https://www.techcrashcourse.com/2023/03/stack-data-structure.html
- *Difference between Linear and Non-linear Data Structures*. (2019, October 11). GeeksforGeeks. https://www.geeksforgeeks.org/difference-between-linear-and-non-linear-data-structures/
- *Graph Data Structure*. (n.d.). Www.programiz.com. Retrieved September 24, 2023, from https://www.programiz.com/dsa/graph

Reference List

- *Hash Tables*. (n.d.). Data Structures Handbook. Retrieved September 24, 2023, from https://www.thedshandbook.com/hash-tables/
- *Tree Data Structure*. (n.d.). Www.programiz.com. Retrieved September 24, 2023, from https://www.programiz.com/dsa/trees
- *Java - Arrays - Tutorialspoint*. (n.d.). Www.tutorialspoint.com. Retrieved September 23, 2023, from https://www.tutorialspoint.com/java/java_arrays.htm
- Prasanna. (2023, March 24). *Array Advantages And Disadvantages | What are Array? Advantages and Disadvantages of Array*. A plus Topper. https://www.aplustopper.com/array-advantages-and-disadvantages/#Advantages_of_Array
- Programiz. (n.d.). *Data Structure and Types*. Www.programiz.com; Parewa Labs Pvt. Ltd. https://www.programiz.com/dsa/data-structure-types

Reference List

- Ravikiran, A. S. (2023, May 9). *Linked List in a Data Structure: All You Need to Know*. Simplilearn.com. https://www.simplilearn.com/tutorials/data-structure-tutorial/linked-list-in-data-structure
- Samad, A. (n.d.). *What are linear data structures?* Educative: Interactive Courses for Software Developers. Retrieved September 23, 2023, from https://www.educative.io/answers/what-are-linear-data-structures
- Smit, A. (2016, February 7). *Introduction to Linear Data Structures*. GeeksforGeeks. https://www.geeksforgeeks.org/introduction-to-linear-data-structures/
- *Stack*. (n.d.). Www.programiz.com. Retrieved September 23, 2023, from https://www.programiz.com/dsa/stack