Data Structures and their Use in Elementary Algorithms

Version 1.0

Authors

Dr. Muhammad Nur Yanhaona
Zaber Mohammad
Nazia Afreen
Sifat Tanvir
Rubayat Ahmed Khan
Avijit Biswas
S. M. Farah Al Fahim
Modhumonty Das

Table of Contents

Preface

Data structures deal with the representation and arrangement of data that we use in computer programs. Based on how we represent data, certain features of our program can be highly efficient or inefficient and, often, limits what our programs can do. Therefore, the study of data structures is essential in computer science education. In fact, a part of the reason we can program modern-day computers so easily is because programming languages allow us to define variables and constants the way we humans understand that languages translate to bits of electrical signals that computers can process. You would be surprised to know that in the early days of computers, users had to write binary numbers (that is, just sequences of zeros and ones) to communicate with the computers. Numbers, characters, and even instructions were just sequences of zeros and ones that users needed to tediously write and then send to the machine to let it do any computation. Imagine how hard and error-prone that mode of programming was. So it is a blessing that we can use data structures whose behavior we easily comprehend when writing computer programs.

Just as big structures in the real world such as roads, buildings, cars, and electrical grids are made of building block elements such as concrete, brick blocks, aluminum, and wires; complex data structures that our programs use are made out of elementary data structures. These elementary data structures are so common that they have been standardized among languages and hardware. *They are typically called primitive data types as there is no structure below them*. For example, bit (aka. A true/false value), byte, character, integer, single-precision fractional number (aka, floating points), double precision fractional numbers are supported by virtually all hardware and programming languages. Larger structures are a combination of these smaller structures and most often a collection of them.

In fact, larger data structures are made of multiple layers of intermediate data structures that go back to primitive data types. For example, to represent a 2D rectangle (a data structure) you need four points (an intermediate data structure) each of which has two integers (for horizontal and vertical placement). Now to compute the total land area covered by a region of rectangular land plots, you would need a collection type data structure that holds all rectangles. Suppose in your computer program you need to know the total number of rectangles frequently. Without any additional info in your collection data structure, you would need to count the rectangles every time you need that information. However, if you keep a counter in the collection that you increase by one every time you add a new rectangular plot, then when querying the total number of rectangles, you could quickly use that counter. So you understand depending on how you structure the data some features may take constant time or involve a lot of computations. Consider another program, where you know that all plots are of the same size. Then you do not need a collection data structure at all. You just need the counter and a single rectangle. This saves a lot of space.

I hope you understand from this simple example, why the choice of data structures is so important. A good grasp of data structures is indispensable for any programming related job and even in hardware design. If you want to do research in computer science in the future then you need a strong foundation on data structures even more. That is why the data structures course is in the program core. I would take it further to say that it is among the few courses that are at the very center of the core program curriculum.

You now understand the pivotal role of data structures in program and hardware design. But what does it means to have a strong foundation on data structures? It means you are capable of designing data new structures from existing when needed, pick appropriate data structures

when being asked to choose, correctly assess what is the cost of different operations related to a data structure, and finally describe the composition of complex data structures in terms of their building block parts.

As we are using computers for solving all sorts of problems for so many decades now, the principal building block data structures that we need to solve interesting problems and larger structures are already well-known. The universal realization of the computing community is that these data structures are principally needed for dealing with different types of collections. If you think a little, this makes sense. A computer may be a thousand times faster than a human in doing an addition between two numbers. However, that a normal human being can do that same operation within a few seconds makes the speed advantage unnecessary. It is when we are given a large sequence of numbers then the speed difference becomes prominent. A computer can do in minutes that may take a human a decade.

Now, different collection handling data structures have different advantages and disadvantages. There is no one data structure that fits for all. Furthermore, when we need to capture the interrelation among the elements of a collection along with the capacity to list them, there are some collection types that simply cannot achieve that.

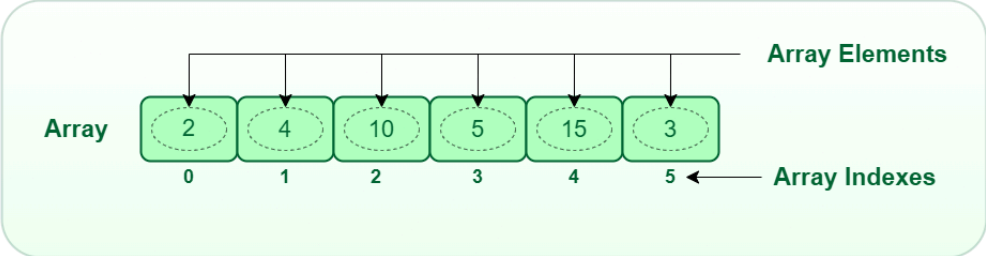
In this course, you will learn the various standard data structures for collection handling. We skip basic data type (often called a class) design as you learned that in your earlier programming courses. You will design these data structures, define useful operations on them, and then use them in common programming problems. As you will go through many data structure studies and implementation exercises in this course, you might feel overwhelmed if you do not study regularly and work sincerely to understand things. So be diligent and committed to your study and attend both theory and lab classes regularly. Remember that, the student life is for learning difficult things for making future life easier.

I hope you will learn a great deal and enjoy this course very much.

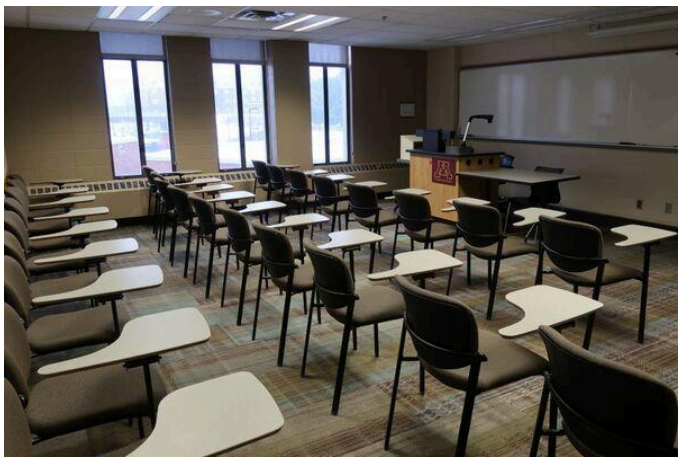
Chapter 1: Array

1.1 Array basics

1.1.1 Introduction



An array is a fixed-dimensional and indexed sequence of fix number of elements of a single type. Every aspect of this definition is important. First, an array has dimensions. So we can create 1D arrays, 2D arrays, 3D arrays, and so on. The simplest form is the one-dimensional array. Second, an array is an indexed collection. Indexed means each element has a designated position. Hence, you can retrieve an element by its position. For example, if you ask what is the value at position 2 in the above picture then the answer is 10. Third, an array has a fixed number of elements along each dimension. This count of elements along a dimension is called the length of the dimension. When people deal with a 1D array, they typically use the term length of the array to indicate the length of that sole dimension. Note that this does not mean you have to use a constant in your code as an array dimension length in your source code. It means that when you create an array in the program, you have to specify the length of each of its dimensions that you cannot change later. Finally, an array can hold elements of only a single type. That is, you cannot mix strings with numbers in an array – not even mix integer numbers with fractional numbers. All these restrictions are essential for efficient operations. In fact, arrays are by far the fastest data structures that computers can process. There is a reason for this efficiency. A computer's own memory is a 1D array of fixed-sized memory cells. Consequently, the notion of a programming language array fits directly with how computer stores and access data from memory.



The sit arrangement in a classroom can be described using a 2D array. In a multidimensional array, each array element has an index for each of the array dimensions. So a 2D array of classroom sits has two indexes, a row index and a column index. The value to store in the index depends on array type. For example, you may be interested to know which sits are occupied and what are not. Then each element can be a Boolean value. However, if you want to know who sit where, then each element of the array will be a student object.

A real life example of a 2D array

Accessing an Array Element: If you consider how the CPU of your computer read/write data item from the computer memory (aka the RAM) for an array, you will understand why array access is so efficient. The CPU is given the starting address of the array in memory and the index of the element program needs to access. As the size of the object an array hold can be larger than the size of a single memory cell, CPU cannot just add the index with the starting address to locate the element in the searched index. However, we know that the elements are all of the same type. So there size is fixed. Therefore, the CPU just multiply the index by the fixed object size and add that result with the starting address to determine the address of the searched element. Then it needs a single memory load/store operation to read/write the element. So accessing element by index from an array is a constant-time operation.

This also tells you why it is important to have the length of each dimension of an array immutable (means you cannot change). Because only then the language can decide how any consecutive memory cells it should reserve from the hardware memory to store the array.

1.2 Operations on Array

The most common operations that you do on an array are writing/reading a data item to/from an index. Suppose you have a 1D array named *student_names*. If you want get the 5th student from the array then you write:

fifth_student = student_names[5]

To change the value at 5th index and set it to 'John Doe', you write:

student_names[5] = 'John Doe'

Typically, you do not shift the positions of elements of an array, unless you are swapping the positions of two elements (often needed for sorting the array in increasing or decreasing order of element values). Similarly inserting a new element at a particular index or removing an element from that index is also atypical as that involves shifting the positions of other elements. You avoid such operations on an array as they are costly. However, it is common to create a new array from an existing array where the new array has a filtered subset of elements from the old array or have the same elements but in a different order. There is another basic data structure that is highly efficient for that, it is called a *List*.

Determining the Length of an Array: it is a curious question, how do you know the dimension lengths of an array. Languages that are focused on the best performance say that the programmers should store the length of each dimension in some other variable. This makes sense as you provide the dimension lengths when you create the array. This means that when you write something like *array[i]* to access the i^{th} element of the array, the language does not check if *i* is crossing the length of the array. In other word, there is no checking if the index you gave is valid or not. If you give an invalid index, you get invalid data. Examples of such languages are C, Fortran, and Golang.

On the other hand, some other languages, particularly object oriented languages, store the length information you supplied in the first memory cell of the array and you cannot access that location from your program. When you write *array[i]* in such a language, the generated code put the array access inside a conditional if-else block. The if-else block check if *i* is greater than equal to 0 and less than the length of the array. If that is not true then you get an error when your program executes the statement containing array[i]. Examples of such languages are Java and C#.

The tradeoff here is that accessing an array element in Java/C# is two to three times slower than accessing an array element in C/Fortran/Golang. However, if you are a careless programmer then Java/C# gives better protection that you will not make errors. This efficiency of array access is one of the main reasons critical system software such as the operating systems and compilers are written in C.

1.3 Linear Array Properties

I. All the elements will be of the same data type
II. All the non-dummy values will be at the left followed by the dummy values (zero/None/null)
III. Length of an array is fixed once declared and cannot be changed

1.3.1. Ways of Creating a Numpy Array

```
1. FUNCTION create_array(size)
2.     array = new Array(size)
3.     FOR i = 0 TO size-1
4.         array[i] = 0
5.     END FOR
6.     RETURN array
7. END FUNCTION
8. arr = create_array(5)
```

1.3.2 Linear Array Operations

I. Iteration
II. Resize
III. Copy (Pass by Value)
IV. Shift: Left and Right
V. Rotate: Left and Right
VI. Reverse: Out-of-place and In-place
VII. Insert: At the end or anywhere else
VIII. Delete: Last element or any other element

I. Iteration

Iteration refers to checking all the values index by index. The main idea is to go to that memory location and check all the values index by index.

```
1. for i = 0 to len(arr1) - 1
```

```
2.    print arr1[i]
3.  end for
```

II. Resize

We can not resize an array because it has a fixed memory location. However, if we ever need to resize an array, we need to create a new array with a new length and then copy the values from the original array. For example, if we have an array [10, 20, 30, 40, 50] whose length is 5 and want to resize the array with length 8. The new array will be [10, 20, 30, 40, 50, None, None, None]. Here None is representing empty values.

```
1.  function resize(arr, new_size)
2.    set arr2 to an array of size new_size, initialized with 0
3.    set i to 0
4.    while i is less than length of arr
5.      set arr2[i] to arr[i]
6.      increment i by 1
7.    end while
8.    return arr2
9.  end function
```

III. Copy (Pass by Value)

Copy array means you initialize a new array with the same length as the given array to copy and then copy the old array's value by value. As the variable where we store the array only stores the memory location, only copying the value is not enough for array copying. For example, if you have an array titled arr = [1, 2, 3, 4] and write a2 = arr. It does not mean you have copied arr to a2.

```
1.  FUNCTION copy_array(arr)
2.      arr2 = create_array(size of arr)
3.      FOR i = 0 TO size of arr - 1
4.          arr2[i] = arr[i]
5.      END FOR
6.      RETURN arr2
7.  END FUNCTION

arr = create_array(5)
copyArray(arr)
```

IV. Shift: Left and Right

Shifting an Array Left: Shifting an entire array left moves each element one (or more, depending how the shift amount) position to the left. Obviously, the first element in the array will fall off at the beginning and be lost forever. The last slot of the array before the shift (ie. the slot where the last element was until the shift) is now unused (we can put a None there to signify that). For example, shifting the array [5, 3, 9, 13, 2] left by one position will result in the array [3, 9, 13, 2, None]. Note how the array[0] element with the value of 5 is now lost, and there is an empty slot at the end.

```
1. FUNCTION shift_left(arr)
2.     FOR i = 1 TO size of arr - 1
3.         arr[i-1] = arr[i]
4.     END FOR
5.     arr[size of arr - 1] = 0
6.     RETURN arr
7. END FUNCTION
```

Shifting an Array Right: Shifting an entire array right moves each element one (or more, depending how the shift amount) position to the right. Obviously, the last element in the array will fall off at the end and be lost forever. The first slot of the array before the shift (ie., the slot where the first element was until the shift) is now unused (we can put a None there to signify that). The size of the array remains the same however because the assumption is that you would something in the now-unused slot. For example, shifting the array [5, 3, 9, 13, 2] right by one position will result in the array [None, 5, 3, 9, 13]. Note how the array[4] element with the value of 2 is now lost, and there is an empty slot at the beginning.

```
1. FUNCTION shift_right(arr)
2.     FOR i = size of arr - 1 DOWNTO 1
3.         arr[i] = arr[i-1]
4.     END FOR
5.     arr[0] = 0
6.     RETURN arr
7. END FUNCTION

arr = create_array(5)
shift_right(arr)
```

V. Rotate: Left and Right

Rotating an Array Left: Rotating an array left is equivalent to shifting a circular or cyclic array left where the 1st element will not be lost, but rather move to the last slot. Rotating the array [5, 3, 9, 13, 2] left by one position will result in the array [3, 9, 13, 2, 5].

```
1. FUNCTION rotate_left(arr)
2.     temp = arr[0]
3.     FOR i = 1 TO size of arr - 1
4.         arr[i-1] = arr[i]
5.     END FOR
6.     arr[size of arr - 1] = temp
7.     RETURN arr
8. END FUNCTION

arr = create_array(5)
```

rotate_left(arr)

Rotating an Array Right: Rotating an array right is equivalent to shifting a circular or cyclic array right where the last element will not be lost, but rather move to the 1st slot. Rotating the array [5, 3, 9, 13, 2] right by one position will result in the array [2, 5, 3, 9, 13].

```
1.  FUNCTION rotate_right(arr)
2.      temp = arr[size of arr - 1]
3.      FOR i = size of arr - 1 DOWNTO 1
4.          arr[i] = arr[i-1]
5.      END FOR
6.      arr[0] = temp
7.      RETURN arr
8.  END FUNCTION

arr = create_array(5)
rotate_right(arr)
```

VI. Reverse: Out-of-place and In-place

Reversing an array can be implemented in two ways. First, we will create a new array with the same size as the original array and then copy the values in reverse order. The method is called out-of-the-place operation.

However, an efficient approach might be to reverse the array in the original array. By this, we will not need to allocate extra spaces. This is known as an in-place operation. To do so we need to start swapping values from the beginning position to the end position. The idea is to swap starting value with the end value, then the second value with the second last value, and so on.

```
1.  FUNCTION reverse_out_of_place(arr)
2.      arr2 = create_array(size of arr)
3.      i = 0
4.      j = size of arr - 1
5.      WHILE i <= size of arr - 1
6.          arr2[i] = arr[j]
7.          i = i + 1
8.          j = j - 1
9.      END WHILE
10.     RETURN arr2
11. END FUNCTION

arr = create_array(5)
reverse_out_of_place(arr)
```

```
1.  FUNCTION reverse_in_place(arr)
2.      j = size of arr - 1
3.      FOR i = 0 TO (size of arr - 1) DIVIDED BY 2
4.          SWAP arr[i] WITH arr[j]
5.          j = j - 1
6.      END FOR
7.      RETURN arr
8.  END FUNCTION

arr = create_array(5)
reverse_in_place(arr)
```

Introducing New Linear Array Property: Size

- Size indicates the number of non-dummy values in an array.
- Size of an array can never exceed the length of that array. Which means size > length is not possible.
- Size cannot be negative.
- Size is used in array insertion or deletion, or any task that involves working with the non-dummy values only.
- If size < length, insertion in an array is possible, otherwise must resize the array.
- If size > 0, at least one element from the array can be deleted, otherwise, there is nothing to delete.

```
1.  FUNCTION print_non_dummies(arr, size)
2.      FOR i = 0 TO size - 1
3.          PRINT arr[i]
4.      END FOR
5.  END FUNCTION

arr = create_array(8)
print_non_dummies(arr, 5) # Suppose it has 5 values
```

VII. Insert: At the end or anywhere else

- At first, check whether insertion is possible or not. If not possible, resize the array
- Valid places of inserting: index 0 to size
- If you are inserting at the end of the array, no shifting is needed
- If you are inserting anywhere else but at the end of an array, then the subsequent elements must be right shifted before insertion
- Increase size after insertion

```
#Inserting at the end
    1.  FUNCTION insert_at_the_end(arr, size, elem)
    2.      IF size >= size of arr
    3.          arr = resize_array(arr, size of arr + 5)
    4.      END IF
    5.      arr[size] = elem
    6.      RETURN arr
    7.  END FUNCTION

arr = create_array(5)
insert_at_the_end(arr, 5, 10)

#Inserting anywhere
    1.  FUNCTION insert_anywhere(arr, size, index, elem)
    2.      IF index < 0 OR index > size
    3.          RETURN "Insertion Not Possible"
    4.      END IF
    5.      IF size >= size of arr
    6.          arr = resize_array(arr, size of arr + 3)
    7.      END IF
    8.      FOR i = size DOWNTO index + 1
    9.          arr[i] = arr[i-1]
    10.     END FOR
    11.     arr[index] = elem
    12.     RETURN arr
    13. END FUNCTION

arr = create_array(5)
insert_anywhere(arr, 5, 2, 10)
```

Note: If you want to insert at the end with this function too, all you have to do is provide the index value equal to the size

VIII. Remove: Last element or any other element

- At first, check whether deletion is possible or not
- Valid places of deletion: index 0 to size-1
- If you are deleting the last element of the array, no shifting is needed
- If you are deleting any other element except for the last element, then the subsequent elements must be left shifted after deletion
- Decrease size after deletion

```
#Deleting last element
    1.  FUNCTION delete_last_element(arr, size)
    2.      IF size = 0
```

```
    3.          RETURN "Deletion Not Possible"
    4.      END IF
    5.      arr[size-1] = 0
    6.      RETURN arr
    7.  END FUNCTION
```

```
arr = create_array(5)
delete_last_element(arr, 5)

#Deleting any element
    1.  FUNCTION delete_any_element(arr, size, index)
    2.      IF size = 0 OR (index < 0 AND index >= size)
    3.          RETURN "Deletion Not Possible"
    4.      END IF
    5.      FOR i = index TO size - 1
    6.          arr[i] = arr[i+1]
    7.      END FOR
    8.      arr[size-1] = 0
    9.      RETURN arr
    10. END FUNCTION

arr = create_array(5)
delete_any_element(arr, 5, 2)
```

Practice Problem 1: Check Palindrome
Look up what a palindrome is. Write a function that checks whether an array is a plaindrome or not.

```
def palindrome(arr, size):
  pass
```

Practice Problem 2: Reverse Print
Write a function Rev_Print that Reverse iterates and prints all the non-dummy values of an array.

Function Call:
arr= np.array([1 3 2 5 0 0])
print(Rev_Print(arr, 4)) #Array, size
Output: 5 2 3 1

Practice Problem 3: Merge Two Arrays
Write a function Merge_Arrays that takes two arrays and the sizes of those two arrays (total 4 parameters) and mergers the two arrays. There will be no dummy value in the merged array.

Function Call:
arr1= np.array([1 5 0 0])

```
arr2= np.array([3 5 2 0])
print(Merge_Arrays(arr1, 2, arr2, 3)) #Array1, Array1 size, Array2, Array2 size
Output: [1 5 3 5 2]
```

1.4 Multidimensional Array

1.4.1 Introduction

You know the basics of arrays and understand why they are important. However, we focused mostly on linear or 1D array in our earlier discussion. Let us focus on multidimensional arrays now. Multidimensional arrays give us a natural way to deal with real-world objects that are multidimensional, and such objects are quite frequent. Consider an image rendering operation to display a picture on the screen. It is convenient to use a 2D array of pixels to represent the picture to decide how to magnify and align the pixels of the image with the points of the 2D screen. For another example, if you do computation related to heat propagation on a 3D object as part of a scientific study of solid materials' heat conductivity; it is natural to use 3D arrays to represent the solids. Take this second example. Suppose you want to read the current temperature at the <x,y,z> point of a 3D plate variable representing a solid, you can simply do this as follows:

cell_temperature = plate[x][y][z]

You see how convenient it is to write computations using multidimensional arrays! Fortunately, an element access from multidimensional arrays is as efficient as it is convenient. To understand the efficiency, we need to see how languages allocate multidimensional arrays and locate the memory address for a particular multidimensional index.

1.4.2 Multidimensional Arrays in Memory

Interestingly – but not surprisingly – programming languages store multidimensional arrays as liner arrays in the RAM. This makes sense, right? As we know the RAM is a linear array of memory cells. Let us see how it works. Remember that an array contains elements of a single type and its number of dimensions and dimension lengths are given when you create it. Therefore, a language can follow the simple technique of placing all elements for increasing values of the index along a single dimension in consecutive locations in the memory while keeping the indexes along all other dimensions fixed. After it is done traversing a dimension, then it increases the value of the index along another dimension and restart from the zero index of the previous dimension. Following this process until the indexes along all dimensions become the maximum possible, translates the whole multidimensional array into a linear array. There is just one restriction; a language must follow a fixed dimension ordering rule to store all multidimensional arrays. There are two standards here:

1. Increasing the final dimension's index first and progressively move to earlier dimensions, this is called the Row-Major Ordering. For example, Java and C take this approach.
2. Increasing the starting dimension's index first and progressively move to later dimensions, this is called the Column-Major Ordering. For example, FORTRAN takes this approach.

The following diagram shows the difference between these two approaches for a 2D matrix of 3 ×3 dimensions. (By the way, row is the vertical and column is the horizontal axis in 2D. You should already know that. Here Dimension 0 represents the row and Dimension 1 represents the column.)

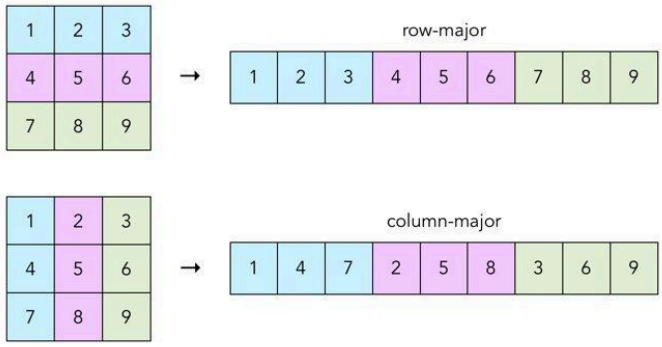


Figure 1: Two Different Ways of Storing Multidimensional Arrays in a Linear Format

There is a great impact of different languages choosing different ordering of dimensions for storing multidimensional arrays. The impact is related to what is efficient in computer hardware and what is not. The way modern computers are built, it is several times more efficient to access data to/from consecutive memory locations than from locations that are distant from one another. As a result, if you iterate the elements of a multidimensional array in an order that is different from how they are stored in memory then your program can be several times slower. Now given the index ordering chosen by Java is radically opposite from how FORTRAN does it, if you simply translate an efficient Java code into an equivalent FORTRAN code then the performance will tank.

The important lesson here is that efficient programming is often language dependent.

The following diagram shows how the row-major approach works for a 3D 3×3×3 cube. From this example, you understand the higher dimension count of the original multidimensional array is not a problem.

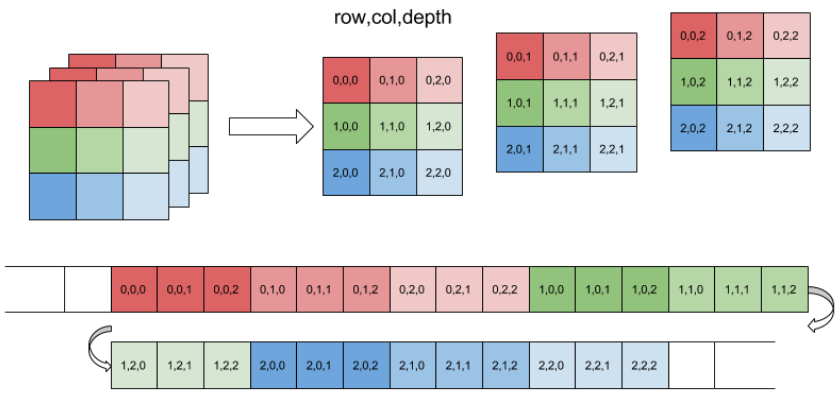


Figure 2: Row Major Ordering of Indexes of a 3D Array

Let us stick to the row-major ordering, as that is more common in application programming. You should be interested to know how to determine the translated linear index of a multidimensional index from the original array. For that, imagine you have a 4D array with dimensions M×N×O×P, named *box,* and the element at the index you are interested in is the *box[w][x][y][z]*. Then the index of that element in the linear array is:

$$w \times (N \times O \times P) + x \times (O \times P) + y \times P + z$$

You understand the general rule, right? Basically multiply the upper dimension indexes by the multiplication of the lengths of lower dimensions then add them all. So easy, isn't it?

Do a practice of the reverse operation. Suppose a linear array of length 128 actually stores a 3D array of dimensions 4×4×8. What are the multidimensional indexes of the element stored at location 111? Remember that indexing starts from 0 in each dimension.

Since it is so simple to represent multidimensional arrays as linear arrays in memory, some languages do not support multidimensional arrays as building block data structures at all. They only support linear arrays. The idea is, since it is such a simple computation to go back and forth between multidimensional and linear arrays, let programmers handle the logic of index conversions and keep the language simple. C is a classic example of such a language that does not allow creation of multidimensional arrays dynamically (which means during program runtime).

You have to admit that this is a valid argument.

Solution of the Example Problem:
The dimension of the given array is [4][4][8] and length is 128. We need to map the dimension of linear index 111.

The equation we get is:

X * (4*8) + Y*8 + Z = 111 where 0 <= X <= 3, 0 <= Y <= 3 and 0 <= Z <= 7.

Now to find the value of X, Y and Z we need to repeatedly divide the remainder of the linear index with the multiplication of the lengths of lower dimensions till you reach the last dimension. So first 111/(4*8) then the quotient is x and then you use the remainder R to do R/8 to get y index. The remainder of that is the z index
Following the process we get,
X = 111//(4*8) = 3 and 111 % (4*8) = 15
Y = 15 // 8 = 1 and 15 % 8 = 7
Z = 7

So, the dimension of linear index 111 is [3][1][7].
Now practice finding the dimension of 96, 107, and 60 of the linear index.

1.4.3 Initialization of a Multidimensional Array
We can use the NumPy library for initializing a multidimensional array. To use the library we have to import the numpy library first. Hence,

```
import numpy as np
```

To create an empty array, we can use the zeros() function

```
m = np.zeros((2,3), dtype = int)
```

Here, (2,3) indicates the dimension [namely 2 rows and 3 columns] and dtype indicates datatype. Default datatype of a numpy array is float. The resulting array, m will be

```
[[0 0 0]
 [0 0 0]]
```

Without explicitly mentioning dtype, the resulting array becomes-

```
[[0. 0. 0.]
 [0. 0. 0.]]
```

Here, '.' indicates floating point numbers.

We can also create a multidimensional NumPy array by using the array() function.

```
m = np.array([[4,3,8],[2,5,1]])
```

The resulting array, m will be

```
[[4 3 8]
 [2 5 1]]
```

For simplicity's sake, we shall consider 2D matrices in this lecture.

Suppose, you initialize a 2D array using
m = [[None]*3]*2
Then you assign, 4 in (0,0) index
m[0][0] = 4
print m and see what the output is!

1.4.4 Creating a 2D Matrix

```
1.  FUNCTION createArray()
2.      DECLARE m: ARRAY [0:1, 0:2] OF INTEGER
3.      DECLARE row, col: INTEGER
4.      row, col = m.DIMENSIONS
5.      FOR i = 0 TO row-1
6.          FOR j = 0 TO col-1
7.              PRINT "Enter element of [" + i + "][" + j + "] index: "
8.              READ m[i][j]
9.          END FOR
10.     END FOR
11.     RETURN m
12. END FUNCTION
```

1.4.5 Iteration of a 2D Matrix

1. **Row Wise Iteration**

```
1.1.    FUNCTION print_row(m: ARRAY)
1.2.        DECLARE row, col: INTEGER
1.3.        row, col = m.DIMENSIONS
1.4.        FOR i = 0 TO row-1
1.5.            FOR j = 0 TO col-1
1.6.                PRINT m[i][j] + " "
1.7.            END FOR
1.8.            PRINT "\n"
1.9.        END FOR
1.10.   END FUNCTION
```

2. **Column Wise Iteration:**

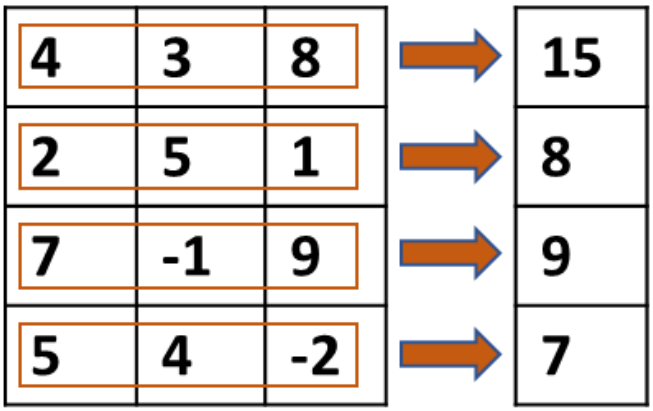
```
2.1.    FUNCTION print_col(m: ARRAY)
2.2.        DECLARE row, col: INTEGER
2.3.        row, col = m.DIMENSIONS
2.4.        FOR i = 0 TO col-1
2.5.            FOR j = 0 TO row-1
2.6.                PRINT m[j][i] + " "
2.7.            END FOR
2.8.            PRINT "\n"
2.9.        END FOR
2.10.   END FUNCTION
```

1.4.6 Summation

1. **Sum of all elements in a 2D matrix:**

```
1.1.  FUNCTION array_sum(m: ARRAY)
1.2.      DECLARE sum, row, col: INTEGER
1.3.      sum = 0
1.4.      row, col = m.DIMENSIONS
1.5.      FOR i = 0 TO row-1
1.6.          FOR j = 0 TO col-1
1.7.              sum = sum + m[i][j]
1.8.          END FOR
1.9.      END FOR
1.10.     RETURN sum
1.11. END FUNCTION
```

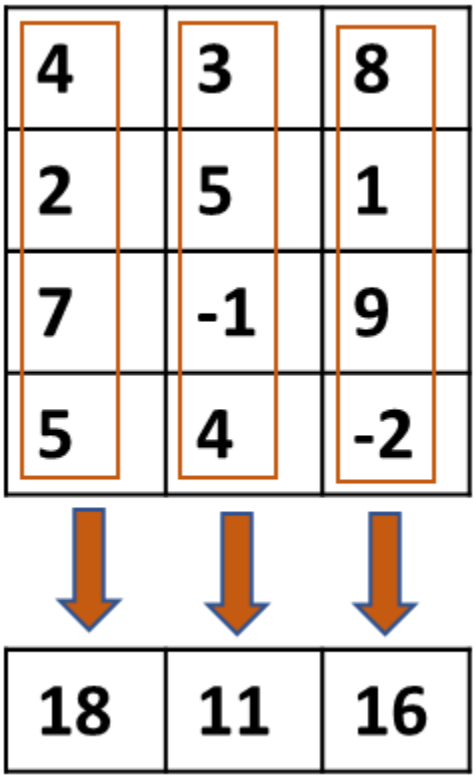
2. **Sum of every row in a given matrix**



Thus, the resulting matrix will always have 1 column to store row wise sum and equal number of rows of the main matrix.

```
1.  FUNCTION row_wise_sum(m: ARRAY)
2.      DECLARE row, col: INTEGER
3.      DECLARE result: ARRAY [0:row-1, 0:0] OF INTEGER
4.      row, col = m.DIMENSIONS
5.      FOR i = 0 TO row-1
6.          FOR j = 0 TO col-1
7.              result[i][0] = result[i][0] + m[i][j]
8.          END FOR
9.      END FOR
10.     RETURN result
11. END FUNCTION
```

3. Sum of every column in a given matrix

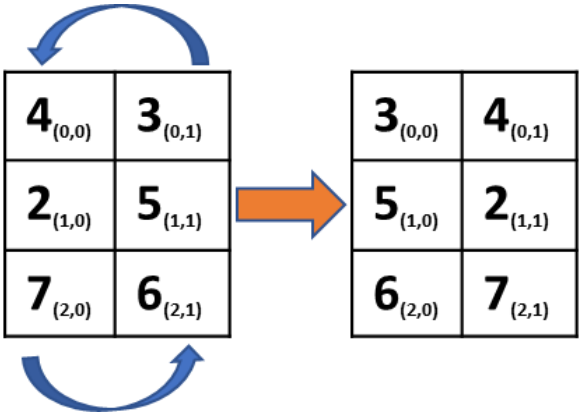


Thus, the resulting matrix will always have 1 row to store column wise sum and equal number of columns of the main matrix.

```
1.  FUNCTION col_wise_sum(m: ARRAY)
2.      DECLARE row, col: INTEGER
3.      DECLARE result: ARRAY [0:0, 0:col-1] OF INTEGER
4.      row, col = m.DIMENSIONS
5.      FOR i = 0 TO col-1
6.          FOR j = 0 TO row-1
7.              result[0][i] = result[0][i] + m[j][i]
8.          END FOR
9.      END FOR
10.     RETURN result
11. END FUNCTION
```

1.4.7 Swapping:

1. **Swap the two columns of a m x 2 matrix**




```
1.  FUNCTION swap_2columns(m: ARRAY)
2.      DECLARE row, col: INTEGER
3.      row, col = m.DIMENSIONS
4.      FOR i = 0 TO row-1
5.          temp = m[i][0]
6.          m[i][0] = m[i][1]
7.          m[i][1] = temp
8.      END FOR
9.      RETURN m
10. END FUNCTION
```

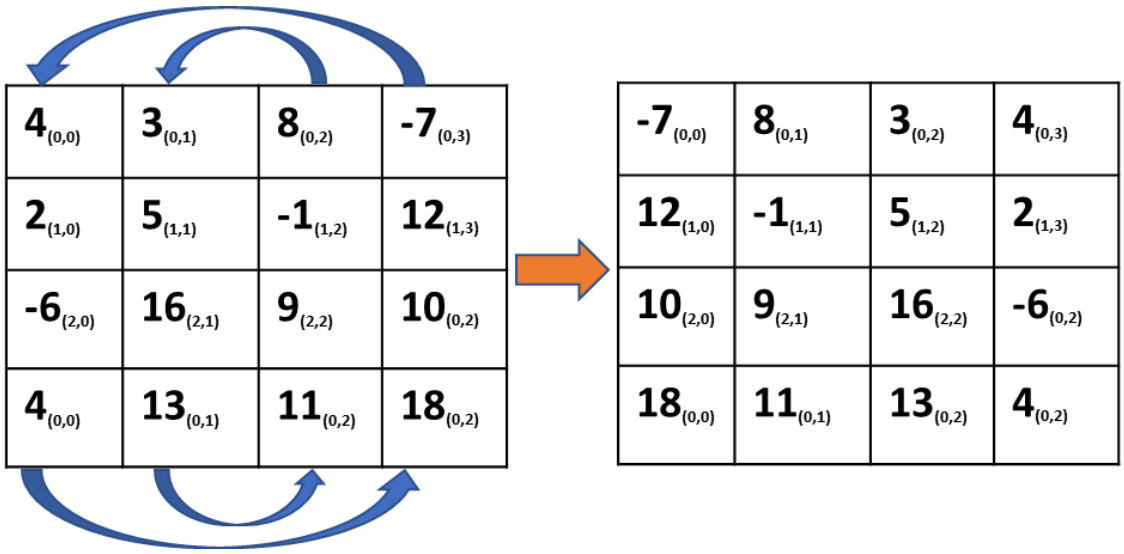
2. Swap the columns of a m x n matrix

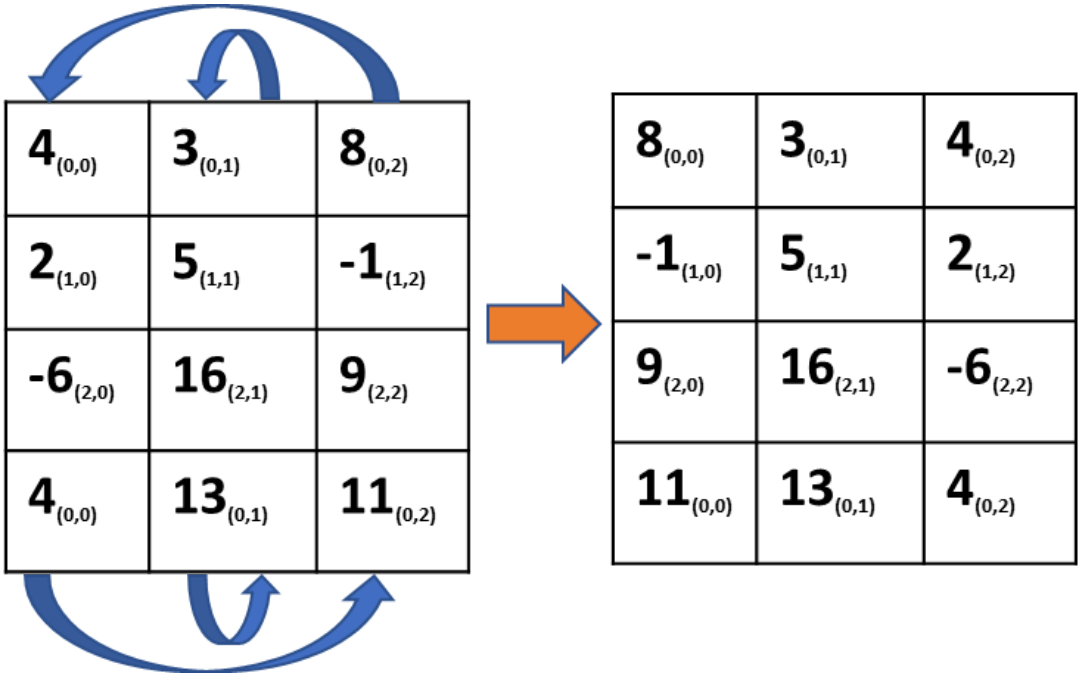
0^{th} column ⇌ $(n-1)^{th}$ column

1^{st} column ⇌ $(n-2)^{th}$ column

2^{nd} column ⇌ $(n-3)^{th}$ column

and so on and so forth

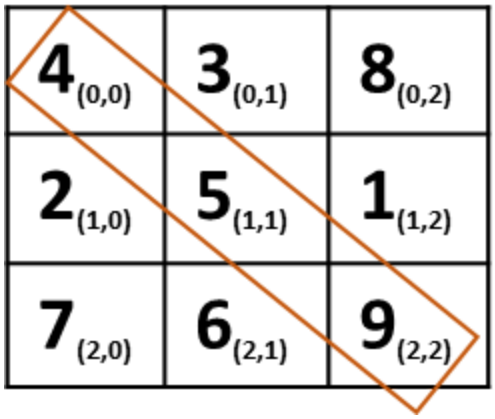





```
1.  FUNCTION swap_columns(m: ARRAY)
2.      DECLARE row, col: INTEGER
3.      row, col = m.DIMENSIONS
4.      FOR i = 0 TO row-1
5.          FOR j = 0 TO col/2
6.              temp = m[i][j]
7.              m[i][j] = m[i][col-1-j]
8.              m[i][col-1-j] = temp
9.          END FOR
10.     END FOR
11.     RETURN m
12. END FUNCTION
```

1.4.8 Addition:

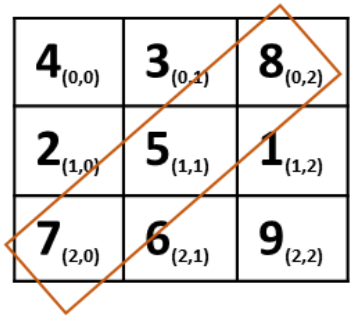
1. **Add the elements of the primary diagonal in a square matrix**



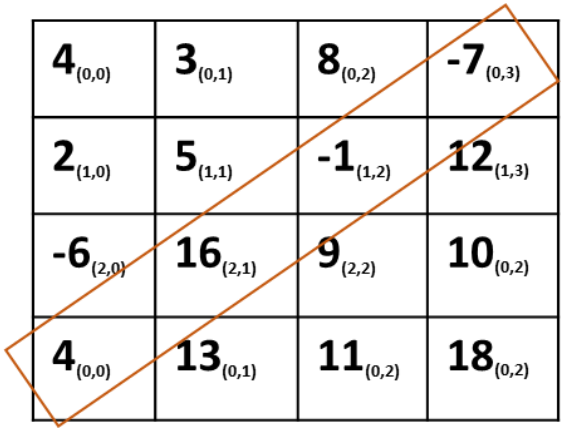
The main diagonal of a square matrix are the elements who's row number and column number are equal, a_{ii}

```
1.  FUNCTION sum_primary_diagonal(m: ARRAY)
2.      DECLARE row, col: INTEGER
3.      row, col = m.DIMENSIONS
4.      ASSERT row = col, "Not a square matrix"
5.      DECLARE sum: INTEGER
6.      sum = 0
7.      FOR i = 0 TO row-1
8.          sum = sum + m[i][i]
9.      END FOR
10.     RETURN sum
11. END FUNCTION
```

2. Add the elements of the secondary diagonal in a square matrix



Secondary diagonal of a 3x3 Matrix



Secondary diagonal of a 4x4 Matrix

For an element a_{ij} in the secondary diagonal, can you find out the j for a particular i? Hint: try i+j for a_{ij} in the secondary diagonal and find out the relation.

3. Add two matrices of same dimension

```
3.1.  FUNCTION add_matrix(m: ARRAY, n: ARRAY)
3.2.      DECLARE r_m, c_m, r_n, c_n: INTEGER
3.3.      r_m, c_m = m.DIMENSIONS
3.4.      r_n, c_n = n.DIMENSIONS
3.5.      ASSERT r_m = r_n AND c_m = c_n, "Dimension mismatch"
3.6.      DECLARE result: ARRAY [0:r_m-1, 0:c_m-1] OF INTEGER
3.7.      FOR i = 0 TO r_m-1
3.8.          FOR j = 0 TO c_m-1
3.9.              result[i][j] = m[i][j] + n[i][j]
3.10.         END FOR
3.11.     END FOR
3.12.     RETURN result
3.13. END FUNCTION
```

1.4.9 Multiply two matrices

```
1.  FUNCTION multiply(m: ARRAY, n: ARRAY)
2.      DECLARE r_m, c_m, r_n, c_n: INTEGER
3.      r_m, c_m = m.DIMENSIONS
4.      r_n, c_n = n.DIMENSIONS
5.      ASSERT c_m = r_n, "Cannot multiply"
6.      DECLARE result: ARRAY [0:r_m-1, 0:c_n-1] OF INTEGER
7.      FOR i = 0 TO r_m-1
8.          FOR j = 0 TO c_n-1
9.              FOR k = 0 TO c_m-1
10.                 result[i][j] = result[i][j] + m[i][k] * n[k][j]
11.             END FOR
12.         END FOR
13.     END FOR
14.     RETURN result
15. END FUNCTION
```

Exercises

1.1: Given two arrays a[] and b[] of size n and m respectively. The task is to find union between these two arrays.
Union of the two arrays can be defined as the set containing distinct elements from both the arrays. If there are repetitions, then only one occurrence of element should be printed in the union.
Example 1:
Input:
5 3
1 2 3 4 5
1 2 3
Output:
5
Explanation:
1, 2, 3, 4 and 5 are the
elements which comes in the union set
of both arrays. So count is 5.
Example 2:
Input:
6 2
85 25 1 32 54 6
85 2
Output:
7
Explanation:
85, 25, 1, 32, 54, 6, and
2 are the elements which comes in the union set of both arrays. So count is 7.

1.2: Given an unsorted array arr[] of size N having both negative and positive integers. The task is place all negative element at the end of array without changing the order of positive element and negative element.
Example 1:
Input :
N = 8
arr = [1, -1, 3, 2, -7, -5, 11, 6]
Output :
1 3 2 11 6 -1 -7 -5
Example 2:
Input :
N=8
arr =[-5, 7, -3, -4, 9, 10, -1, 11]
Output :
7 9 10 11 -5 -3 -4 -1

1.3: Given an unsorted array A of size N that contains only non-negative integers, find a continuous sub-array which adds to a given number S.
In case of multiple subarrays, return the subarray which comes first on moving from left to right.
Example 1:

Input:
N = 5, S = 12
A = [1,2,3,7,5]
Output: 2 4
Explanation: The sum of elements
from 2nd position to 4th position
is 12.
Example 2:
Input:
N = 10, S = 15
A = [1,2,3,4,5,6,7,8,9,10]
Output: 1 5
Explanation: The sum of elements
from 1st position to 5th position
is 15.

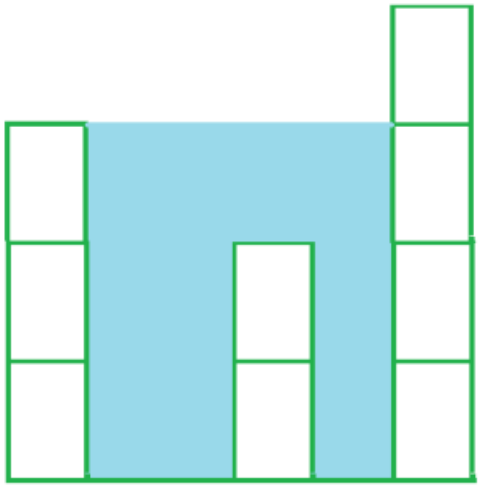
1.4: Given an array of N positive integers and an integer X. The task is to find the frequency of X in the array.
Example 1:
Input:
N = 5
arr = [1, 1, 1, 1, 1]
X = 1
Output:
5
Explanation: The frequency of 1 is 5.

1.5: Given an array arr and an integer K where K is smaller than size of array, the task is to find the Kth smallest element in the given array. It is given that all array elements are distinct.
Example 1:
Input:
N = 6
arr = [7, 10, 4, 3, 20, 15]
K = 3
Output : 7
Explanation :
3rd smallest element in the given
array is 7.
Example 2:
Input:
N = 5
arr = [7, 10, 4, 20, 15]
K = 4
Output : 15
Explanation :
4th smallest element in the given
array is 15.

1.6: Given an array of size N containing only 0s, 1s, and 2s; sort the array in ascending order.
Example 1:

Input:
N = 5
arr = [0, 2, 1, 2, 0]
Output:
0 0 1 2 2
Explanation:
0s 1s and 2s are segregated
into ascending order.
Example 2:
Input:
N = 3
arr = [0, 1, 0]
Output:
0 0 1
Explanation:
0s 1s and 2s are segregated
into ascending order.

1.7: Given an array arr of N non-negative integers representing the height of blocks. If the width of each block is 1, compute how much water can be trapped between the blocks during the rainy season.
Example 1:
Input:
N = 6
arr = [3,0,0,2,0,4]
Output:
10
Explanation:



Bars for input {3, 0, 0, 2, 0, 4}
Total trapped water = 3 + 3 + 1 + 3 = 10

Example 2:
Input:

N = 4
arr = [7,4,0,9]
Output:
10
Explanation:
Water trapped by above
block of height 4 is 3 units and above
block of height 0 is 7 units. So, the
total unit of water trapped is 10 units.
Example 3:
Input:
N = 3
arr = [6,9,9]
Output:
0
Explanation:
No water will be trapped.

1.8: Given a sorted array arr of distinct integers. Sort the array into a wave-like array(In Place). In other words, arrange the elements into a sequence such that arr[1] >= arr[2] <= arr[3] >= arr[4] <= arr[5]..... If there are multiple solutions, find the lexicographically smallest one.
Example 1:
Input:
n = 5
arr = [1,2,3,4,5]
Output: 2 1 4 3 5
Explanation: Array elements after
sorting it in wave form are
2 1 4 3 5.
Example 2:
Input:
n = 6
arr = [2,4,7,8,9,10]
Output: 4 2 8 7 10 9
Explanation: Array elements after
sorting it in wave form are
4 2 8 7 10 9.

1.9: Given an array A of N elements. Find the majority element in the array. A majority element in an array A of size N is an element that appears more than N/2 times in the array.
Example 1:
Input:
N = 3
A = [1,2,3]
Output:
-1
Explanation:
Since, each element in
[1,2,3] appears only once so there
is no majority element.

Example 2:
Input:
N = 5
A = [3,1,3,3,2]
Output:
3
Explanation:
Since, 3 is present more
than N/2 times, so it is
the majority element.

1.10: Given an array of N integers arr where each element represents the max length of the jump that can be made forward from that element. Find the minimum number of jumps to reach the end of the array (starting from the first element). If an element is 0, then you cannot move through that element.
Note: Return -1 if you can't reach the end of the array.
Example 1:
Input:
N = 11
arr = [1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9]
Output: 3
Explanation:
First jump from 1st element to 2nd
element with value 3. Now, from here
we jump to 5th element with value 9,
and from here we will jump to the last.
Example 2:
Input :
N = 6
arr = [1, 4, 3, 2, 6, 7]
Output: 2
Explanation:
First we jump from the 1st to 2nd element
and then jump to the last element.

1.11: Given an array arr denoting heights of N towers and a positive integer K. For each tower, you must perform exactly one of the following operations exactly once.
● Increase the height of the tower by K
● Decrease the height of the tower by K
Find out the minimum possible difference between the height of the shortest and tallest towers after you have modified each tower.
Note: It is compulsory to increase or decrease the height by K for each tower. After the operation, the resultant array should not contain any negative integers.
Example 1:
Input:
K = 2, N = 4
Arr = [1, 5, 8, 10]
Output:
5
Explanation:

The array can be modified as
[3, 3, 6, 8]. The difference between
the largest and the smallest is 8-3 = 5.
Example 2:
Input:
K = 3, N = 5
Arr = [3, 9, 12, 16, 20]
Output:
11
Explanation:
The array can be modified as
[6, 12, 9, 13, 17]. The difference between
the largest and the smallest is 17-6 = 11.

1.12: Given arrival and departure times of all trains that reach a railway station. Find the minimum number of platforms required for the railway station so that no train is kept waiting. Consider that all the trains arrive on the same day and leave on the same day. Arrival and departure time can never be the same for a train but we can have arrival time of one train equal to departure time of the other. At any given instance of time, same platform can not be used for both departure of a train and arrival of another train. In such cases, we need different platforms.
Note: Time intervals are in the 24-hour format(HHMM) , where the first two characters represent hour (between 00 to 23) and the last two characters represent minutes (this may be > 59).
Example 1:
Input: n = 6
arr = [0900, 0940, 0950, 1100, 1500, 1800]
dep = [0910, 1200, 1120, 1130, 1900, 2000]
Output: 3
Explanation:
Minimum 3 platforms are required to
safely arrive and depart all trains.
Example 2:
Input: n = 3
arr = [0900, 1100, 1235]
dep = [1000, 1200, 1240]
Output: 1
Explanation: Only 1 platform is required to
safely manage the arrival and departure
of all trains.

1.13: Given an array A of positive integers of size N, where each value represents the number of chocolates in a packet. Each packet can have a variable number of chocolates. There are M students, the task is to distribute chocolate packets among M students such that :
1. Each student gets exactly one packet.
2. The difference between maximum number of chocolates given to a student and minimum number of chocolates given to a student is minimum.
Example 1:
Input:
N = 8, M = 5
A = [3, 4, 1, 9, 56, 7, 9, 12]
Output: 6

Explanation: The minimum difference between
maximum chocolates and minimum chocolates
is 9 - 3 = 6 by choosing following M packets :
[3, 4, 9, 7, 9].
Example 2:
Input:
N = 7, M = 3
A = [7, 3, 2, 4, 9, 12, 56]
Output: 2
Explanation: The minimum difference between
maximum chocolates and minimum chocolates
is 4 - 2 = 2 by choosing following M packets :
[3, 2, 4].

1.14: Given an array of positive integers. Find the length of the longest sub-sequence such that elements in the subsequence are consecutive integers, the consecutive numbers can be in any order.
Example 1:
Input:
N = 7
a = [2,6,1,9,4,5,3]
Output:
6
Explanation:
The consecutive numbers here
are 1, 2, 3, 4, 5, 6. These 6
numbers form the longest consecutive
subsquence.
Example 2:
Input:
N = 7
a = [1,9,3,10,4,20,2]
Output:
4
Explanation:
1, 2, 3, 4 is the longest
consecutive subsequence.

1.15: Given two sorted arrays nums1 and nums2 of size m and n respectively, print the median of the two sorted arrays.
Example 1:
Input: nums1 = [1,3], nums2 = [2]
Output: 2.00000
Explanation: merged array = [1,2,3] and median is 2.
Example 2:
Input: nums1 = [1,2], nums2 = [3,4]
Output: 2.50000
Explanation: merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5

Chapter 2: Linked List

2.1 Introduction

When studying the array, we were quite annoyed by the limitations of these arrays:

- **Fixed capacity:** Once created, an array cannot be resized. The only way to "resize" is to create a larger new array, copy the elements from the original array into the new one, and then change the reference to the new one.
- **Shifting elements in insert:** Since we do not allow gaps in the array, inserting a new element requires that we shift all the subsequent elements right to create a hole where the new element is placed. In the worst case, we "disturb" every slot in the array when we insert it at the beginning of the array!
- **Shifting elements in removal:** Removing may also require shifting to plug the hole left by the removed element. If the ordering of the elements does not matter, we can avoid shifting by replacing the removed element with the element in the last slot (and then treating the last slot as empty).

The answer to these problems is the **Linked List** data structure. It is another primary data structure which is a sequence of nodes connected by links. The links allow the insertion of new nodes anywhere in the list or to remove of an existing node from the list without having to disturb the rest of the list (the only nodes affected are the ones adjacent to the node being inserted or removed). Since we can extend the list one node at a time, we can also resize a list until we run out of resources. However, we'll find out soon enough what we lose in the process — random access, and space! A linked list is a sequence container that supports sequential access only and requires additional space for at least n references for the links.

We maintain a linked list by referring to the first node in the list, conventionally called the **head** reference. All the other nodes can then be reached by traversing the list, starting from the head. An empty list is represented by setting the head reference to None. Given a head reference to a list, how do you count the number of nodes in the list? You have to iterate over the nodes, starting from the head, and count until you reach the end.

As mentioned earlier, a linked list is a sequence of nodes. To put anything (primitive type or object reference) in the list, we must first put this "thing" in a node, and then put the node in the list. Compare this with an array – we simply put the "thing" (our primitive type or object reference) directly into the array slot, without the need for any extra scaffolding. Now to put the “thing” in our list and to store the next item’s information, we need to design our own data type by designing a **Node** class. Our Node class is quite simple — it contains an element, and a reference to the next node in the list. Below is a code snippet of a node class given.

```
CLASS Node
   elem = NULL
   next = NULL
   CONSTRUCTOR( e, n)

      elem = e
```

```
        next = n

    END CONSTRUCTOR
END CLASS
```

This allows you to create a singly-linked list, and as a result, we can only move forward in the list by following the next links. To be able to move backward as well, we'll have to add another reference to the previous node, increasing the space usage even more. If you are wondering how we can do that, give it a thought (We will cover it later).

2.2 Operations of Linked List

To understand the operations, first, we will need to understand how to create a linked list. After creation, we can perform different operations on it like an array (e.g: iteration, insertion, removal, and so on).

2.2.1 Creation:

In order to create a linked list, we just need to create objects of the Node class and then connect one node to another using that next variable. We will only store the first Node-type object and call it head. Whenever we want to add a new node-type object, we just need to go to the last node and add the new node after that.
In the example code below, we are taking an array and converting it to a Linked List. To keep things easier, we are using the tail variable so that we do not need to go to the end of the list every time we want to add a new item.

```
1.  FUNCTION create_list(arr)
2.      head = Node(arr[0], NULL)
3.      tail = head
4.      FOR i = 1 TO size of arr - 1
5.          n = Node(arr[i], NULL)
6.          tail.next = n
7.          tail = tail.next
8.      END FOR
9.      RETURN head
10. END FUNCTION

create_list(arr)
```

2.2.2 Iteration:

Iteration is one of the most important aspects of every data structure. We will use an iterative approach to traverse a linked list. The important thing is that to do this traversal we are using a variable (temp in this case) to store the location of node and so we are updating this temp for every iteration. Below is an example of an iteration function that takes the head of a linked list as a parameter.

```
1.  FUNCTION iteration(head)
```

```
2.      temp = head
3.      WHILE temp != NULL
4.          PRINT temp.element
5.          temp = temp.next
6.      END WHILE
7.  END FUNCTION
```

2.2.3 Count:

In order to count the number of nodes present in a linked list, we just need to count how many times the iteration process is running during a traversal.

```
1.  FUNCTION count(head)
2.      count = 0
3.      temp = head
4.      WHILE temp != NULL
5.          count = count + 1
6.          temp = temp.next
7.      END WHILE
8.      RETURN count
9.  END FUNCTION
```

2.2.4 Retrieving an element from an index:

In the memory level, a linked list is not indexed as it does not book any memory block. However, the concept of indexing is very beneficial in order to access values, insert a value, and remove a value (we have seen it in the array part). For this reason, we will implement indexing in our linked list where the head will be a value at index 0 and the next node will be 1, and so on (just like an array). Below an example code is shown where the head of the elemAt function is taking a head of the list and an index (idx). The function is returning the corresponding element at that index. If such an index is not found the function will print an error message.

```
1.  FUNCTION elem_at(head, idx)
2.      count = 0
3.      temp = head
4.      WHILE temp != NULL
5.          IF count = idx
6.              RETURN temp.elem
7.          END IF
8.          temp = temp.next
9.          count = count + 1
10.     END WHILE
11.     RETURN NULL
12. END FUNCTION

elem_at(head, idx)
```

2.2.5 Retrieving a node from an index:

Similar to getting the element, we can also get the node of that index. Below the example, nodeAt function is given which is similar to the elemAt.

```
1.  FUNCTION nodeAtt(head, idx)
2.      count = 0
3.      temp = head
4.      WHILE temp != NULL
5.          IF count = idx
6.              RETURN temp
7.          END IF
8.          temp = temp.next
9.          count = count + 1
10.     END WHILE
11.     RETURN NULL
12. END FUNCTION

elem_at(head, idx)
```

2.2.6 Update value of specific index:

By tuning the elemAt function, we can also update the element of a linked list using an index.

```
1.  FUNCTION set(head, idx, elem)
2.      count = 0
3.      temp = head
4.      is_updated = False
5.      WHILE temp != NULL
6.          IF count = idx
7.              temp.elem = elem
8.              is_updated = True
9.              BREAK
10.         END IF
11.         temp = temp.next
12.         count = count + 1
13.     END WHILE
14.     IF is_updated
15.         PRINT "Value successfully updated!!!!"
16.     ELSE
17.         PRINT "Invalid index"
18.     END IF
19. END FUNCTION
set(head, idx, elem)
```

2.2.7 Searching an element in the list:

Searching for an element in a list can be done by sequentially searching through the list. There are two typical variants: return the index of the given element (indexOf), or return true if the element exists

```
1.  FUNCTION index_of(head, elem)
2.      temp = head
3.      count = 0
4.      WHILE temp != NULL
5.          IF elem = temp.elem
6.              RETURN count
7.          END IF
8.          count = count + 1
9.          temp = temp.next
10.     END WHILE
11.     RETURN -1 # Here -1 represents the absence of element in the list
12. END FUNCTION

index_of(head, elem)
```

Second approach:

```
1.  FUNCTION contains(head, elem)
2.      temp = head
3.      WHILE temp != NULL
4.          IF elem = temp.elem
5.              RETURN True
6.          END IF
7.          temp = temp.next
8.      END WHILE
9.      RETURN False
10. END FUNCTION

contains(head, elem)
```

2.2.8 Inserting an element into a list:

There are three places in a list where we can insert a new element: in the beginning ("head" changes), in the middle, and at the end. To insert a new node in the list, you need the reference to the predecessor to link in the new node. There is one "special" case however — inserting a new node at the beginning of the list because the head node does not have a predecessor. Inserting, in the beginning, has an important side effect — it changes the head reference! Inserting in the middle or at the end is the same — we first find the predecessor node and link in the new node with the given element. To find the specific node, we can take the help of the nodeAt function mentioned above. Below is the example code of insertion. However, this code can be made more efficient!!! Try that.

```
1.  FUNCTION insert(head, elem, idx)
2.      total_nodes = count(head)
3.      IF idx = 0 # Inserting at the beginning
4.          n = Node(elem, head)
5.          head = n
6.      ELSE IF idx >= 1 AND idx < total_nodes # Inserting at the middle
7.          n = Node(elem, head)
8.          n1 = node_at(head, idx - 1)
9.          n2 = node_at(head, idx)
10.         n.next = n2
11.         n1.next = n
12.     ELSE IF idx = total_nodes # Inserting at the end
13.         n = Node(elem, NULL)
14.         n1 = node_at(head, total_nodes - 1)
15.         n1.next = n
16.     ELSE
17.         PRINT "Invalid Index"
18.     END IF
19.     RETURN head
20. END FUNCTION
```

insert(head, elem, idx)

2.2.9 Removing an element from a list:

Removing an element from the list is done by removing the node that contains the element. If we only know the element that we want to remove, then we have to sequentially search the list to find the node which contains the element (if it exists in the list that is); or else, we may already have a reference to the node which contains the element to remove; or, we have an index of the element in the list. Just like inserting a new node in a list, removing requires that you have the reference to the predecessor node. And just like in insertion, removing the 1st node in the list is a "special" case — it does not have a predecessor, and removing it has the side-effect of changing the head. You can try to make this more efficient too!!!

```
1.  FUNCTION remove(head, idx)
2.      IF idx = 0 # Removing first element
3.          head = head.next
4.      ELSE IF idx >= 1 AND idx < count(head) # Removing middle element
5.          n1 = node_at(head, idx - 1)
6.          removed_node = n1.next
7.          n1.next = removed_node.next
8.      ELSE
9.          PRINT "Invalid Index"
10.     END IF
11.     RETURN head
12. END FUNCTION
```

remove(head, idx)

2.2.10 Copying a list:

Copying the elements of a source list to a destination list is simply a matter of iterating over the elements of the source list, and inserting these elements at the end of the destination list.

```
1.  FUNCTION copy_list(source)
2.      copy_head = NULL
3.      copy_tail = NULL
4.      temp = source
5.      WHILE temp != NULL
6.          n = Node(temp.elem, NULL)
7.          IF copy_head = NULL
8.              copy_head = n
9.              copy_tail = copy_head
10.         ELSE
11.             copy_tail.next = n
12.             copy_tail = copy_tail.next
13.         END IF
14.         temp = temp.next
15.     END WHILE
16.     RETURN copy_head
17. END FUNCTION
```

2.2.11 Reversing a list:

Since a linked list does not support random access, it is difficult to reverse a list in place without changing the head reference. Instead, we'll create a new list with its own head reference, and copy the elements in the reverse order. This method does not modify the original list, so we can call it an out-of-place method.

```
1.  FUNCTION reverse_out_of_place(head)
2.      new_head = Node(head.elem, NULL)
3.      temp = head.next
4.      WHILE temp != NULL
5.          n = Node(temp.elem, new_head)
6.          new_head = n
7.          temp = temp.next
8.      END WHILE
9.      RETURN new_head
10. END FUNCTION

reverse_out_of_place(head)
```

The problem with this approach is that it creates a copy of the whole list, just in reverse order. We would like an in-place approach instead — re-order the links instead of copying the nodes! That would of course change the original list and would have a new head reference (which is the reference to the tail node in the original list).

```
1.  FUNCTION reverse_in_place(head)
2.      new_head = NULL
3.      temp = head
4.      WHILE temp != NULL
5.          n = temp.next
6.          temp.next = new_head
7.          new_head = temp
8.          temp = n
9.      END WHILE
10.     RETURN new_head
11. END FUNCTION

reverse_in_place(head)
```

2.2.12 Rotating a list left:

Rotating a list left is much simpler than rotating an array — the 2nd node becomes the new head, and the 1st becomes the new tail node. We don't have to actually move the elements, it's just a matter of rearranging a few links

```
1.  FUNCTION rotate_left(head):
2.    new_head = head.next
3.    temp = new_head
4.    WHILE temp.next != None:
5.      temp = temp.next
6.    temp.next = head
7.    head.next = None
8.    head = new_head
9.    RETURN head
10. END FUNCTION
```

2.2.13 Rotating a list right:

Rotating a list right is almost the same as rotating left — the current last node becomes the new head and the current second last node becomes the last node.

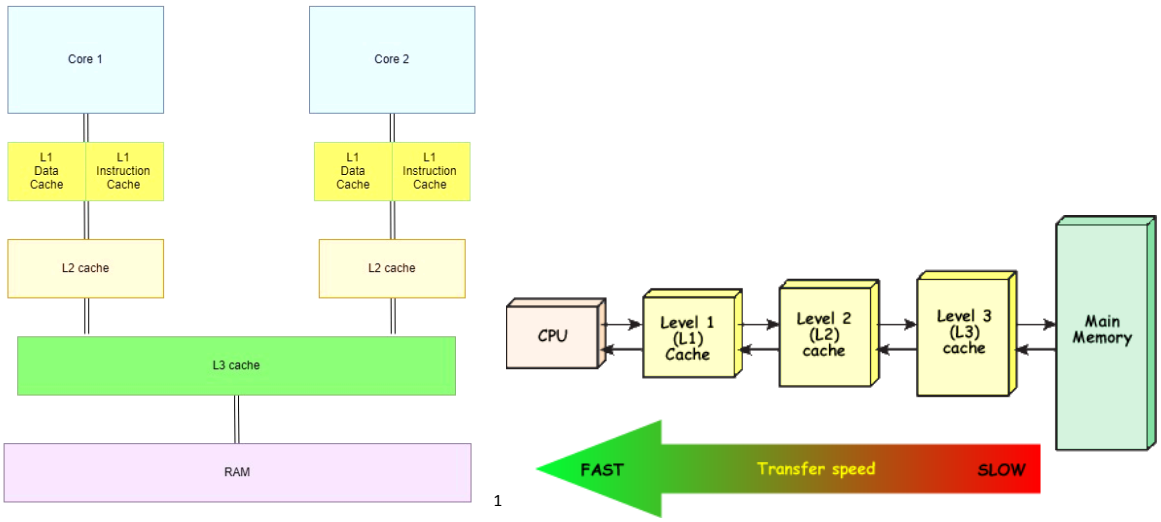
```
1.  FUNCTION rotate_right(head):
2.    last_node = head.next
3.    second_last_node = head
4.    WHILE last_node.next != None:
5.      last_node = last_node.next
6.      second_last_node = second_last_node.next
7.    last_node.next = head
8.    second_last_node.next = None
9.    head = last_node
10.   RETURN head
11. END FUNCTION
```

2.3 Understanding the Hidden Cost of Data Structures

By now you know that traversing all the elements of a linear array and doing that on a linked list have the same computational cost if the number of elements in the array and the list are the same. The cost is some multiple of the number of elements, N. However, what multiple? That is a big question and in a real world setting that matters a lot. To understand the issue, you need a little bit of initiation on how the computer CPU and memory are organized.

From the early days of computers, CPU is many times faster than the RAM (which is the memory) which in turn many times faster than the DISK (which is the computer's data storage). Since, to do any computation in the CPU, you need data; the computer designers applied a simple optimization to avoid hurting computer's performance due to the slowness of data read/write. The idea is that the RAM loads a large block of data at once, which is several KB or MB, from the storage even if the CPU asks for a single byte of data. The reason is, we expect in any typical program, CPU will ask for more data from locations nearby to its first request.

Surprisingly, or not, this assumption happens to be true for more than 98% of time for all data accesses of a program. However, even with this strategy, hardware designer realized that the slowness of the RAM (no longer the slowness of the disk, which has been taken care of) hurts overall program performance greatly. So they just reused the same idea by introducing something called caches. Caches are like RAM but much smaller and faster (their small size is related to their faster speed).



[1]

So when a program runs, if the CPU ask for a single byte of data that is available in the RAM, the hardware actually load multiple consecutive bytes, at the range of 256 bytes or more from the RAM to the cache. The reason is the same; the hardware hopes that the program will make request for data among the additional bytes in near future. At that time, the request can be served from the cache. This idea of loading more data in closer memory works so good that now a days, all computers has multiple levels of caches between the CPU and the RAM.

This strategy for optimization affects the performance of element traversals of an array and a list differently. As you know, array elements are allocated in consecutive memory cells during the

[1] Images: https://www.cybercomputing.co.uk/Languages/Hardware/cacheCPU.html, https://miro.medium.com/max/1030/1*Do424lmt3V06kJcmU1CEgg.png

creation of the array. Therefore, traversal is extremely fast. There are just occasional data transfers between the cache and the RAM. However, space for a list element is allocated when you add the element to the list. The hardware gives whatever the next free memory cells available at that time. As a result, the elements are actually in different places of the RAM and are only connected together by the links of the linked list. So when you traverse a linked list, the hardware has to fetch data from the RAM to caches many times more. Therefore, traversing a linked list is significantly slower than traversing a linear array of the same size. It is like using Google MAP to go from a source location to a destination location. It asks you whether you will walk, bike (bicycle), take a car, or a bus despite the routes (aka intermediate locations you need to cross) being the same. Traversing the array is like taking a bus or a car. Traversing a linked list is like walking or biking.

Why do you not see the performance difference in your assignment programs? The reason it, the arrays and lists you use in your assignments are so small that both typically fit within the amount of data hardware load from the RAM to the cache in a few reads.

2.4 Types of Linked List

Now we have an idea about Linked List and how a linked list works. One important thing to remember is that everything regarding the linked list is created and maintained manually by us. We have designed the node class and also created the linked list. After that, we implemented the idea of indexing along with different operations such as insert, removal, rotation, and so on. If you think about it all these operations are a bit complicated because of our node class design which only has the option of moving from one node to another in a forward manner.

This brings the question can we design our linked list in such a way that we can traverse from one node to another in a forward and backward manner? In addition, can we make our linked list circular and is there any way where we will not need to handle the head explicitly?

To answer all these questions, the types of the linked list have been introduced. Linked Lists type can be determined by considering three different categories. Each category is independent of the other two options. We need to understand first what these options mean.

Category 1	Category 2	Category 3
Non-Dummy Headed	Singly	Linear
Dummy Headed	Doubly	Circular

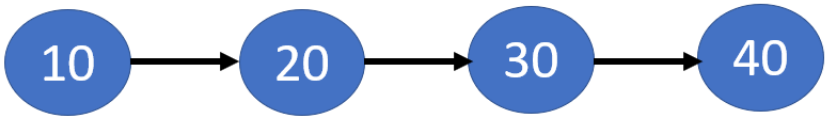
2.4.1 Non-Dummy Headed: It means that the linked list's first node (the head) contains an item along with the information of the next node. For example, the lists we have used till now are Non-Dummy headed.

Example: 10 → 20 → 30 → 40. In this Linked list, the first node stores the value head.

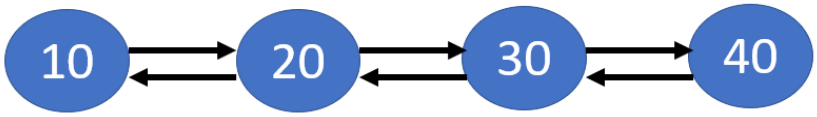
2.4.2 Dummy-Headed: This refers to the linked list where a head node is a node-type object but it does not store any element on its own. Rather it stores the information of the next node where the starting value is stored. The benefit of this is that we do not need to handle the first item of the data carefully now. To illustrate, we can remove or add items at beginning of the list without handling the head delicately. The reason is the first item is stored after the dummy head.

Example: DH → 10 → 20 → 30 → 40. In this linked list, DH refers dummy head which is a node without any element. The first item is stored after the dummy head.

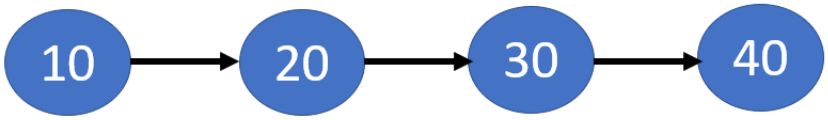
2.4.3 Singly: It means every node has only the information of its next node. The reason is the design of the Node class has only one variable (next variable). Up until now all the list we worked on are Singly Linked List.



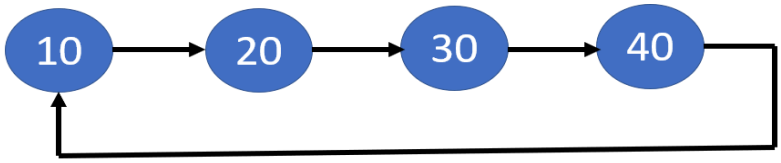
2.4.4 Doubly: It means a node knows its next node and its previous node's information. To achieve this we need to modify the Node class design where we store the previous node's information.



2.4.5 Linear: Linear linked list refers to the linked list where the linked list is linear in structure. To illustrate, the last node of the linked list will refer to None.



2.4.6 Circular: Circular linked list refers to a linked list where the linked list is circular in structure. To illustrate, the last node of the linked list will refer to the starting node.

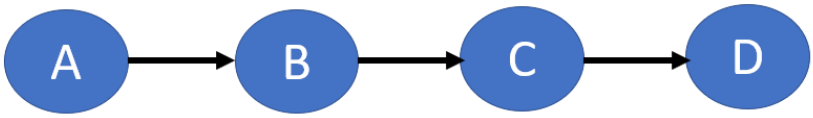


2.4.7 Explanation of the different linked list types

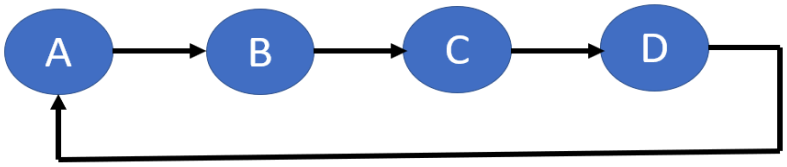
Now that we understand the options of each category, we can easily identify the type of a linked list. The main idea is that a linked list will contain one option from each category mentioned above. The option of one category does not have any impact on other categories.

The linked list we are taught until now is Non-Dummy Headed Singly Linear Linked List. You can think of this as a default linked list. It means if not mentioned, think of the head as Non-Dummy Head, Connection as Singly, and the structure as Linear. Now combining the categories the linked list can be 8 types. DH means dummy head in below diagram

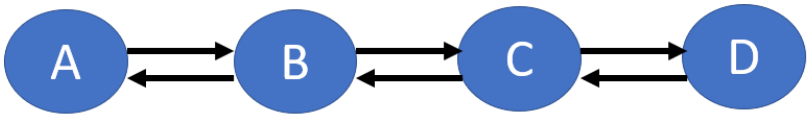
1. **Non-Dummy Headed Singly Linear Linked List:**



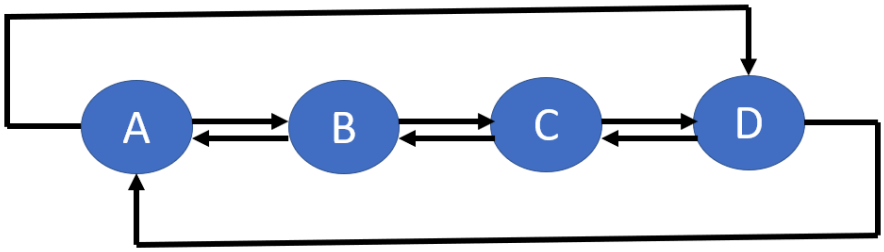
2. **Non-Dummy Headed Singly Circular Linked List:**



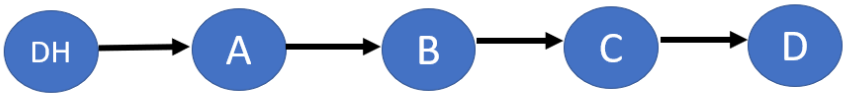
3. **Non-Dummy Headed Doubly Linear Linked List:**



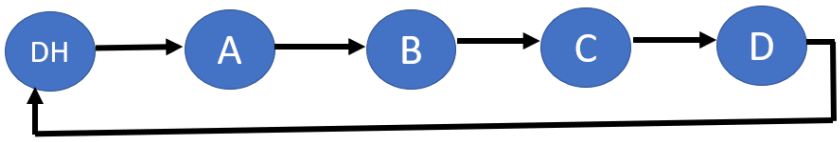
4. **Non-Dummy Headed Doubly Circular Linked List:**



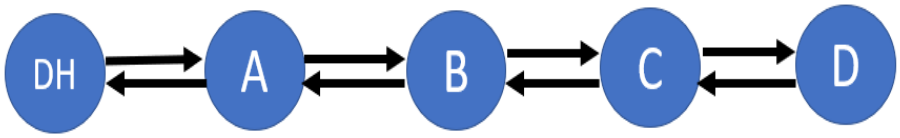
5. **Dummy Headed Singly Linear Linked List:**



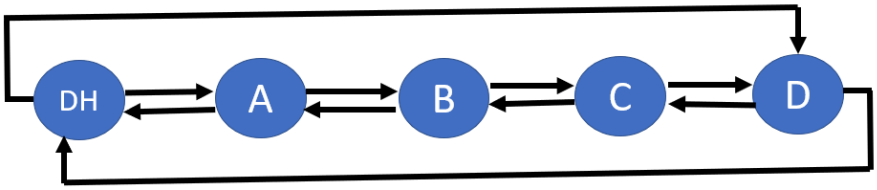
6. **Dummy Headed Singly Circular Linked List:**



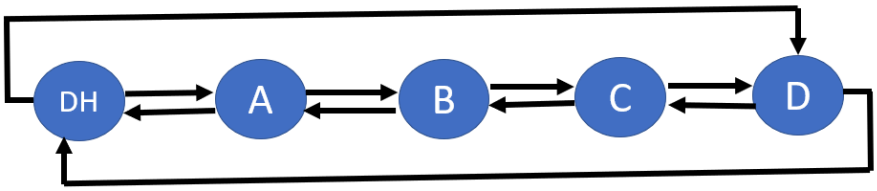
7. **Dummy Headed Doubly Linear Linked List:**



8. **Dummy Headed Doubly Circular Linked List:**



2.5 Dummy Headed Doubly Circular Linked List Operations



To practice we will learn about the dummy-headed doubly circular linked list. As the title suggests, the list will have a dummy head, doubly connection, and circular structure. For this reason, the Node class will be

```
CLASS DoublyNode:
  CONSTRUCTOR (elem, next, prev):
    elem = elem
    next = next # For storing next node's memory address
    prev = prev # For storing previous node's memory address
```

2.5.1 Creation:

It is similar to creating a linked list. The difference is that we need to ensure the connections in a proper way.

The below function takes an array and creates a dummy-headed doubly circular linked list.

```
1.  FUNCTION createList(arr):
2.    dh = DoublyNode(None, None, None)
3.    dh.next = dh
4.    dh.prev = dh
5.    tail = dh
6.
7.    FOR i in range(len(arr)):
8.      n = DoublyNode(arr[i], dh, tail)
9.      tail.next = n
10.     tail = tail.next
11.     dh.prev = tail
12.
13.   RETURN dh
14. END FUNCTION
```

2.5.2 Iteration:

It is similar to the previous linked list. The difference is that it will start from the next item of the dummy head and run till the pointer comes back to the dummy head.

```
1.  FUNCTION iteration(dh):
2.    temp = dh.next
3.    WHILE temp != dh:
4.      PRINT temp.elem
5.      temp = temp.next
6.  END FUNCTION
```

2.5.3 NodeAt:

The nodeAt method is similar to the previous one. The difference is that the dummy head does not represent any index as it does not have any value.

```
1.  FUNCTION nodeAt(dh, idx):
2.    temp = dh.next
3.    c = 0
4.    WHILE temp != dh:
5.      IF c == idx:
6.        RETURN temp
7.      c += 1
8.      temp = temp.next
9.    RETURN None # Invalid Index
10. END FUNCTION
```

Note that, if you want to count the number of total nodes, you should ignore the dummy head too.

2.5.4 Insertion:

To insert a new node in the list, you need the reference to the predecessor to link in the new node. Unlike for a singly-linked linear list, there is no "special" case here, since there is always a valid predecessor node available, thanks to the dummy head.

```
1.  FUNCTION insertion(dh, elem, idx):
2.    # Assuming the idx is valid
3.    node_to_insert = DoublyNode(elem, None, None)
4.    indexed_node = nodeAt(dh, idx) # Retriving the node at that index
5.    prev_node = indexed_node.prev # There will always be a previous node
6.    # Change the connection
7.    # Observe that no special case is needed
8.    node_to_insert.next = indexed_node
9.    node_to_insert.prev = prev_node
10.   prev_node.next = node_to_insert
11.   indexed_node.prev = node_to_insert
12. END FUNCTION
```

2.5.5 Removal:

Removing an element from the list is done by removing the node that contains the element. Just like inserting a new node in a list, removing requires that you have the reference to the predecessor node. Since we're using a doubly-linked list, finding a predecessor of a node is trivial — it's n.prev. And thanks to the dummy head, there is no "special" case here as well.

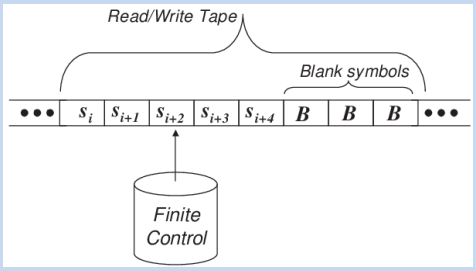
```
1.  FUNCTION removal(dh, idx):
2.    # Assuming the idx is valid
3.    node_to_remove = nodeAt(dh, idx)
4.    prev_node = node_to_remove.prev
5.    next_node = node_to_remove.next
6.    # Change the connection
7.    # No special case is needed
8.    prev_node.next = next_node
9.    next_node.prev = prev_node
10.   node_to_remove.next = None
11.   node_to_remove.prev = None
12.   RETURN node_to_remove.elem # Returning the removed element
13. END FUNCTION
```

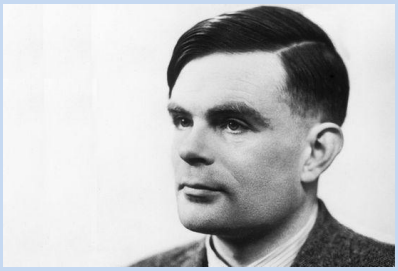
2.6 Reason for Doubly Linked List

By now, you know the difference between arrays and linked lists and their relative advantages and disadvantages. You learned about two types of linked list: single-linked list and double-linked list. Here we will discuss why double linked lists are important.

Although a single-linked list allows us to handle a linear sequence of elements that can be of unknown length and dynamic[2] in our program, there are some limitations in the traversal of the list. Suppose we want to do read/remove/insert near at the end of the list. Then we have to traverse all the way from the head to our position of interest then do the desired operation. When the list is long then this traversal cost is a huge problem. Furthermore, we have to be very careful regarding where to stop the traversal. For example, if you want to insert at position n, then we have to stop traversal at position n – 1. If you want to read then we stop traversal at n. If we make one more wrong step then we cannot backtrack. You have to restart from the head.

This second problem of not being able to move back-and-forth during our traversal is a major concern. In later courses, you will learn that moving backward on a list can be considered as going back in time to investigate the context of future events. This is highly useful for complex language parsing, intelligent agent design, and statistical modeling.





Some of you know about the famous mathematician and computer scientist Alan Turing. He investigated and proved many important properties about computers. One of his most fascinating contributions is the Turing Machine that can simulate any computer. By doing so, the Turing machine allows us to investigate the fundamental limitations and capabilities of all computers ever being built. The machine is surprisingly simple. It has an infinite tape of symbols and a read-write head that can make one-step forward or backward on that tape in each step or change the content of the tape cell where the head is located at that moment. The read-write head, also called the finite control, keeps track of its current state and decides what to do based-on the symbol on the tape in its position and its state.

So what kind of list you need to implement this tape?

[2] means the sequence may change during program execution

Exercises

You can assume your node class is given.

```
class Node:
    def __init__(self, elem, next):
        self.elem = elem
        self.next = next
```

However, you may need to change the node class according to your problem.

2.1: Write a function that moves the last node to the front in a given Singly Linked List.

Examples:

Input: 1 → 2 → 3 → 4 → 5
Output: 5 → 1 → 2 → 3 → 4

Input: 3 → 8 → 1 → 5 → 7 → 12
Output: 12 → 3 → 8 → 1 → 5 → 7

2.2: Given two lists sorted in increasing order, create and return a new list representing the intersection of the two lists. The new list should be made with its own memory — the original lists should not be changed.

Example:

Input:
First linked list: 1 → 2 → 3 → 4 → 6
Second linked list be 2 → 4 → 6 → 8,
Output: 2 → 4 → 6.
The elements 2, 4, 6 are common in
both the list so they appear in the
intersection list.

Input:
First linked list: 1 → 2 → 3 → 4 → 5
Second linked list be 2 → 3 → 4,
Output: 2 → 3 → 4
The elements 2, 3, 4 are common in
both the list so they appear in the
intersection list.

2.3: You are given a linked list that contains N integers. You have performed the following reverse operation on the list:

Select all the subparts of the list that contain only even integers. For example, if the list is {1, 2, 8, 9, 12, 16}, then the selected subparts will be {2, 8}, {12, 16}.
Reverse the selected subparts such as {8, 2} and {16, 12}.
Now, you are required to retrieve the original list.

Input format

First line: N
Next line: N space-separated integers that denote elements of the reverse list
Output format

Print the N elements of the original list.

Sample Input
9
2 18 24 3 5 7 9 6 12

Sample Output
24 18 2 3 5 7 9 12 6

Explanation
In the sample, the original list is {24, 18, 2, 3, 5, 7, 9, 12, 6,} which when reversed according to the operations will result in the list given in the sample input.

2.4: Given a Linked List of integers, write a function to modify the linked list such that all even numbers appear before all the odd numbers in the modified linked list. Also, keep the order of even and odd numbers the same.

Examples:

Input: 17 → 15 → 8 → 12 → 10 → 5 → 4 → 1 → 7 → 6 → None
Output: 8 → 12 → 10 → 4 → 6 → 17 → 15 → 5 → 1 → 7 → None

Input: 8 → 12 → 10 → 5 → 4 → 1 → 6 → None
Output: 8 → 12 → 10 → 4 → 6 → 5 → 1 → None

2.5: Given a Linked List and a number N, write a function that returns the value at the Nth node from the end of the Linked List.

Example:
Input:
10 → 20 → 30 → 40 → 50 → None
N = 2
Output: 40

Input:
35 → 15 → 4 → 20 → 45 → None
N = 4

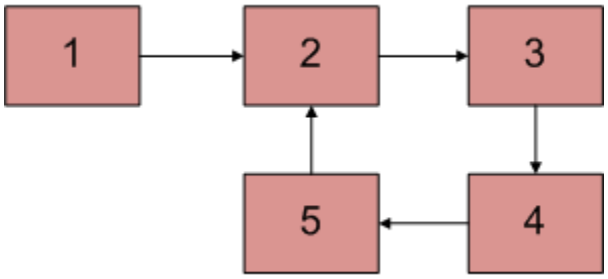
Output: 15

2.6: Given a singly linked list, find the middle of the linked list. If there are even nodes, then there would be two middle nodes, we need to print the second middle element.

Example:
Input: 10 → 20 → 30 → 40 → 50 → None
Output: 30

Input: 1 → 2 → 3 → 4 → 5 → 6 → None
Output: 4

2.7: Given a linked list, check if the linked list has a loop or not. The below diagram shows a linked list with a loop.



2.8: After getting her PhD, Christie has become a celebrity at her university, and her facebook profile is full of friend requests. Being the nice girl she is, Christie has accepted all the requests.

Now Kuldeep is jealous of all the attention she is getting from other guys, so he asks her to delete some of the guys from her friend list.

To avoid a 'scene', Christie decides to remove some friends from her friend list, since she knows the popularity of each of the friend she has, she uses the following algorithm to delete a friend.

```
Algorithm Delete(Friend):
   DeleteFriend=false
   for i = 1 to Friend.length-1
        if (Friend[i].popularity < Friend[i+1].popularity)
           delete i th friend
           DeleteFriend=true
           break
   if(DeleteFriend == false)
      delete the last friend
```

Input:
First line contains a T number of test cases. First line of each test case contains N, the number of friends Christie currently has and K ,the number of friends Christie decides to delete. Next line contains the popularity of her friends separated by space.

Output:

For each test case print N-K numbers which represent popularity of Christie friend's after deleting K friends.
NOTE:
Order of friends after deleting exactly K friends should be maintained as given in input.

Sample Input
3
3 1
3 100 1
5 2
19 12 3 4 17
5 3
23 45 11 77 18

Sample Output
100 1
19 12 17
77 18

2.9: A number is Palindrome if it reads same from front as well as back.For example,
2332 is a palindrome number as it reads the same from both sides.

Linked List can also be palindrome if they have the same order when it traverses from forward as well as backward.

Write a function that will take a linked list as input and return True if the list is a palindrome and return False otherwise.

Sample Input 1:
1 → 2 → 3 → 2 → 1 → None

Sample Output 1:
True

Sample Input 2:
1 → 2 → 3 → 1 → 1 → None

Sample Output 2:
False

2.10: Given a Singly Linked List, starting from the second node delete all alternate nodes of it.

Example:
Input: 10 → 20 → 30 → 40 → 50 → None
Ouput: 10 → 30 → 50 → None

Input: 1 → 2 → 3 → 4 → 5 → 6 → None
Output: 1 → 3 → 5 → None

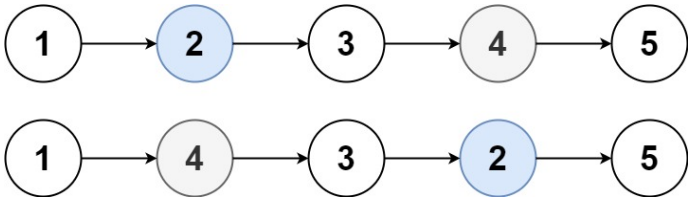
2.11: Two Linked Lists are identical when they have the same data and the arrangement of data is also the same. Write a function to check if the given two linked lists are identical.

Examples:
Input:
a = 1 → 2 → 3 → 4 → 5 → None
b = 1 → 2 → 3 → 4 → 5 → None
Output: Identical

Input:
a = 1 → 2 → 3 → 4 → None
b = 1 → 7 → 4 → 5 → None
Output: Not Identical

2.12: You are given a linked list, and an integer k.

Return the head of the linked list after swapping the values of the kth node from the beginning and the kth node from the end (the list is 1-indexed).



Example:
Input:
a = 1 → 2 → 3 → 4 → 5 → None
k = 2
Output: 1 → 4 → 3 → 2 → 5 → None

Input:
a = 1 → 7 → 4 → 5 → None
k = 2
Output: 1 → 4 → 7 → 5 → None

You can assume your node class is given for Doubly Linked List problems.

```
class Node:
    def __init__(self, elem, next, prev):
        self.elem = elem
        self.next = next
        self.prev = prev
```

However, you may need to change the node class according to your problem.

2.13: Given a non-dummy headed doubly non-circular linked list, write a function that returns true if the given linked list is a palindrome, else false.

Example:
Input: 1 ⇔ 7 ⇔ 7 ⇔ 1 ⇔ None

Output: True

Input: 1 ⇔ 7 ⇔ 4 ⇔ 5 ⇔ None
Output: False

2.14: Given a non-dummy headed doubly non-circular linked list, reverse the list.

Example:
Input: 10 ⇔ 20 ⇔ 30 ⇔ 40 ⇔ 50 ⇔ None
Output: 50 ⇔ 40 ⇔ 30 ⇔ 20 ⇔ 10 ⇔ None

2.15: Given a dummy headed doubly circular linked list, find the largest node in the doubly linked list.

Example:
Input: dummy_head ⇔ 10 ⇔ 70 ⇔ 40 ⇔ 15 ⇔ dummy_head (consider it circular)
Output: 70

2.16: Given a dummy headed doubly circular linked list, rotate it left by k node (where k is a positive integer)

2.17: Given a dummy headed doubly circular linked list, rotate it right by k node (where k is a positive integer)

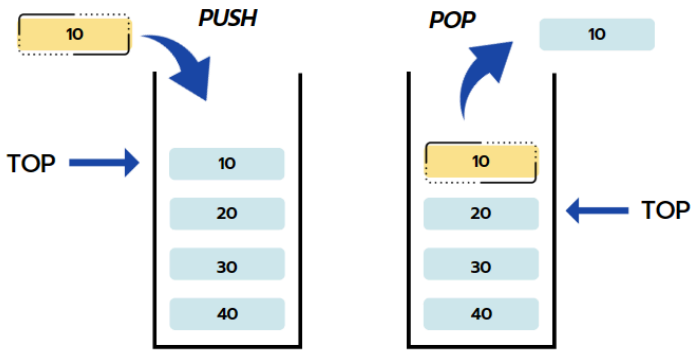
Chapter 3: Stack and Queue

3.1 Stack Introduction

There are some problems in which we want to store information, and we want to access the more recently stored information first. A data structure that allows us to access information in this manner is called a stack. It is so named because it resembles a stack of dishes. When you want a dish, you take it off the stack of dishes. When you clean a dish, you put it back on top of the stack of dishes. The same holds with a stack of data. When you store a new data item, you store it on top of the stack. When you want to retrieve a data item, you retrieve the top item from the stack. Note that this data item will be the one most recently stored on the stack.

Thus, a Stack is a restricted ordered sequence in which we can only add to and remove from one end — the **top** of the stack. Imagine stacking a set of books on top of each other — you can **push** a new book on **top** of the stack, and you can **pop** the book that is currently on the top of the stack. You are not, strictly speaking, allowed to add to the middle of the stack, nor are you allowed to remove a book from the middle of the stack. The only book that can be taken out of the stack is the **most recently added** one; a stack is thus a "last in, first out" (LIFO) data structure.

We use stacks everyday — from finding our way out of a maze, to evaluating postfix expressions, "undoing" the edits in a word-processor, and to implementing recursion in programming language runtime environments.



STACK DATA STRUCTURE

Three basic stack operations are:

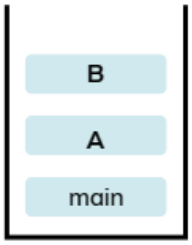
push(obj): adds obj to the top of the stack ("overflow" error if the stack has fixed capacity, and is full)
pop: removes and returns the item from the top of the stack ("underflow" error if the stack is empty)
peek: returns the item that is on the top of the stack, but does not remove it

("underflow" error if the stack is empty)

3.2 Stack Applications

A couple of examples where you want to access information in this manner in the real world are as follows:

3.2.1 Call Stacks: Whenever your C program calls a function, the C compiler creates a so-called *stack record* that contains important information about that function, such as storage for its parameters, its return value, and its local variables. The C compiler pushes this stack record onto something called a *call stack*. The call stack is organized so that the stack record of the most recently called function is on top of the stack, and the stack record of the least recently called function, which in the case of C is main, is at the bottom of the stack. For example, if the main calls A, which in turn calls B, then the call stack would be:



By convention stacks are shown growing upwards. While a function is executing, its stack record will be on top of the stack, and hence its information will be readily available. In the above case, B is currently executing and so we can readily access its parameters and local variables. If B calls a new function C, then a stack record for C is created and pushed onto the call stack. C now has access to its local variables and parameters. When C returns, we want B to resume executing, and thus we want access to B's stack record. By simply popping C's stack record off the call stack, we again have access to B's stack record. Hence a stack is the perfect data structure for implementing a call stack, because we always want access to the most recently called function, and when it returns, we want access to the function that called it.

3.2.2 Reverse: The simplest application of a stack is to reverse a word. You push a given word to stack – letter by letter – and then pop letters from the stack. Here's the trivial algorithm to print a word in reverse:

```
1. begin with an empty stack and an input stream.
2. while there is more characters to read, do:
3.         read the next input character;
4.         push it onto the stack;
5. end while;
6. while the stack is not empty, do:
7.         c = pop the stack;
8.         print c;
9. end while;
```

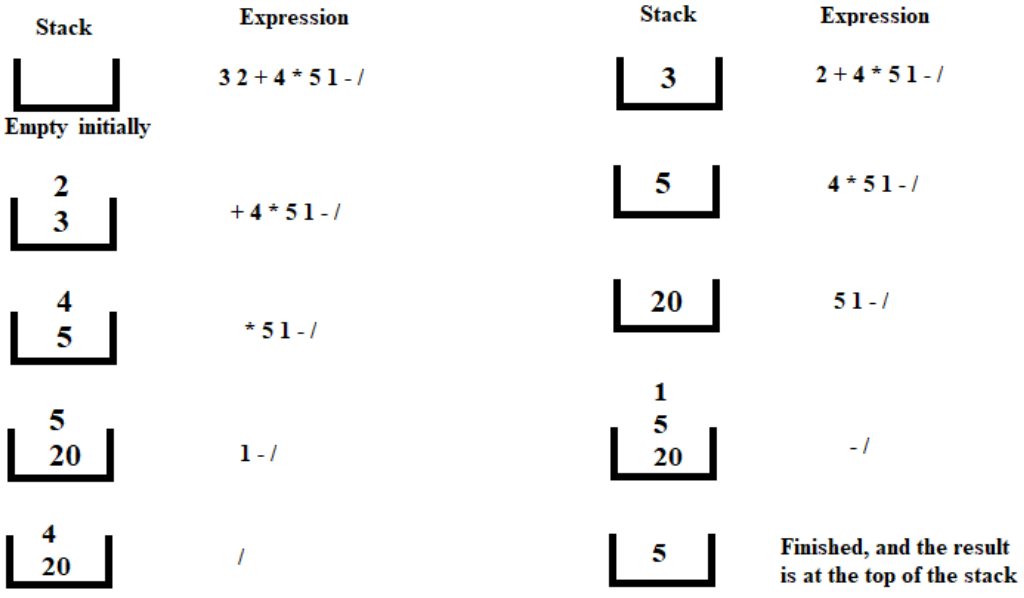
3.2.3 Undo: Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack. Popping the stack is equivalent to "undoing" the last action. A very similar one is going back pages in a browser using the *back* button; this is accomplished by keeping the pages visited in a stack. Clicking on the *back* button is equivalent to going back to the most-recently visited page prior to the current

one.

3.2.4 Expression Evaluation: When an arithmetic expression is presented in the *postfix* form, you can use a stack to evaluate it to get the final value. For example: the expression 3 + 5 * 9 (which is in the usual *infix* form) can be written as 3 5 9 * + in the *postfix*. More interestingly, postfix form removes all parentheses and thus all implicit precedence rules; for example, the infix expression ((3 + 2) * 4) / (5 - 1) is written as the postfix 3 2 + 4 * 5 1 - /. You can now design a calculator for expressions in postfix form using a stack. The algorithm may be written as the following:

```
1.  begin with an empty stack and an input stream (for the expression).
2.  while there is more input to read, do:
3.          read the next input symbol;
4.          if it's an operand,
5.              then push it onto the stack;
6.          if it's an operator
7.              then pop two operands from the stack;
8.              perform the operation on the operands;
9.              push the result;
10. end while;
11. // the answer of the expression is waiting for you in the stack:
12. pop the answer;
```

Let's apply this algorithm to to evaluate the postfix expression 3 2 + 4 * 5 1 - / using a stack:



3.2.5 Parentheses Matching: In many editors, it is common to want to know which left parenthesis (or brace or bracket, etc) will be matched when we type a right parenthesis. Clearly we want the most recent left parenthesis. Therefore, the editor can keep track of parentheses by pushing them onto a stack, and then popping them off as each right parenthesis is encountered. We often have expressions involving "()[]{}" that require that the different types of parentheses are *balanced*. For example, the following are properly balanced: (a (b + c) + d)

[(a b) (c d)]

([a {x y} b])

But the following are not:

(a (b + c) + d

[(a b] (c d))

([a {x y) b])

The algorithm may be written as the following:

```
begin with an empty stack and an input stream (for the expression).
while there is more input to read, do:
      read the next input character;
      if it's an opening parenthesis/brace/bracket ("(" or "{" or "[")
         then push it onto the stack;
      if it's a closing parenthesis/brace/bracket (")" or "}" or "]")
         then pop the opening symbol from stack;
         compare the closing with opening symbol;
         if it matches
            then continue with next input character;
         if it does not match
            then return false;
end while;
// all matched, so return true
return true;
```

3.2.6 Backtracking: This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit?

Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.

3.2.7 Language Processing:

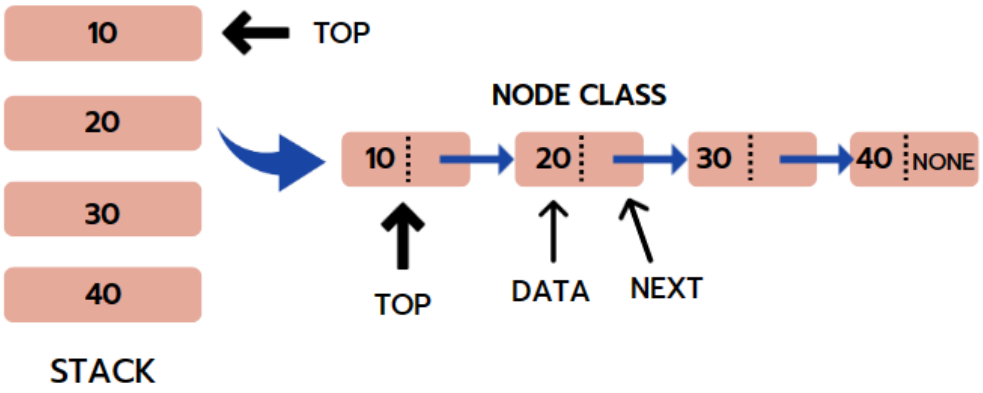
- space for parameters and local variables is created internally using a stack (*activation records*).
- The compiler's syntax check for matching braces is implemented by using stack.
- support for recursion

3.3 Stack Implementation

In the standard library of classes, the data type stack is an *adapter* class, meaning that a stack is built on top of other data structures. The underlying structure for a stack could be an array, a vector, an ArrayList, a linked list. Regardless of the type of the underlying data structure, a ***Stack*** must implement the same functionality. One requirement of a ***Stack*** implementation is that the ***push*** and ***pop*** operations run in *constant time*, that is, the time taken for stack operation is independent of how big or small the stack is.

3.3.1 Linked List-Based Implementation

Linked List-based implementation provides the best (from the efficiency point of view) dynamic stack implementation.



The following shows a partial head-referenced singly-linked based implementation of an *unbounded* stack. In a singly-linked list-based implementation we add the new item being pushed at the beginning of the array (why?), and consequently pop the item from the beginning of the list as well.

First, we create a class node. This is our Linked List node class which will have data in it and a node pointer to store the address of the next node element.

Then, we define the Stack class inside which we have the push(), pop() and peek() methods.

3.3.1.1 Push Operation

Adding a new node in the Stack is termed a push operation. Push operation on stack implementation using linked-list involves several steps:

- Create a node first and allocate memory to it.
- If the stack is empty, then the node is pushed as the first node of the linked list and defined as top. This operation assigns a value to the data part of the node and gives None to the address part of the node.
- If some nodes are already there in the linked list, then we have to add a new node at the beginning of the linked list so that we do not violate Stack's property. For this, assign the previous top to the next address field of the new node and the new node will be the starting node or the current top of the list.

The algorithm may be written as the following:

```
begin
   if stack is empty
      Create a node
      Assign the node to the top variable of the stack
   else
      Create a node and make the top of the stack its next node
      Assign the node to the top variable of the stack

end procedure
```

3.3.1.2 Pop Operation:

Deleting a node from the Stack is known as a pop operation. Pop operation involves the following steps:

- In Stack, the node is removed from the top of the linked list. Therefore, we must delete the value stored in the head/top pointer, and the node must get free. The following link node then becomes the head/top node.
- An underflow condition will occur when we try to pop a node when the Stack is empty.

The algorithm may be written as the following:

```
begin
   if stack is empty
      return underflow exception
   else
      Save the reference of the top of the stack node in a variable
      Make the top node's next node the top of the stack
      Make the previously top node's element and next null using the saved variable

end procedure
```

3.3.1.3 Peek Operation:

In the peek operation, we simply return the data stored in the head/top of the Linked List based Stack. An underflow condition will occur when we try to peek at a value when the Stack is empty.

The algorithm may be written as the following:

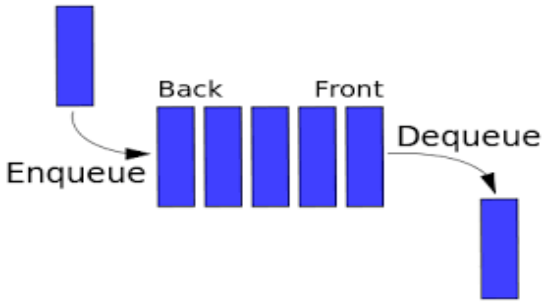
```
Pseudocode for peek:
begin
   if stack is empty
      return underflow exception
   else
      Return the reference of the top of the stack node
end procedure
```

3.4 Queue Introduction

We have started to learn secondary data structures made using primary data structures. The first secondary data structure we have learned is Stack. The idea of Stack was Last In First Out (LIFO) and we implemented that using a linked list. Stack helps us to solve multiple problems more efficiently. Similarly, we can create a new Data Structure that will follow different rules. One example of a new data structure is Queue

A queue is a new data structure that follows **First In First Out (FIFO)** while inserting and retrieving data. We are very familiar with the idea of queue in our daily lives. To illustrate, if you are waiting in line to buy a bus ticket, the counter will serve the customer who has come first and a new customer will wait behind the last person waiting in line. This idea is also important for our computers. One of the main uses of the queue in computers is resource management. For example, think of an office scenario where 10 people use the same printer. So if everyone requests to print simultaneously, the printer will print the first request and store the other requests in a queue. While executing any printing operation if a new print request comes it will store it in the same queue and execute the command in the First In First Out manner. In addition, when we execute 100 programs in our 8-core CPU, the processor also maintains a queue for command execution (Though the scenario is a bit more complex which you will learn in Operating Systems).

3.4.1 Queue Operations:



Queue

A queue has three main operations (like a stack). These operations are:

1. **Enqueue:** It stores a new object in the queue at the end. If the queue is full, this operation throws a QueueOverflow exception.
2. **Dequeue:** It removes the first object from the queue. If the queue is empty, this operation throws a QueueUnderflow exception.
3. **Peek**: This operation shows the first item in the queue. This operation does not remove any items from the queue. If the queue is empty, this operation throws a QueueUnderflow exception.

3.5 Queue Implementation

As we can see, to maintain the queue, we need two pointers, one for item retrieval and one for item insertion. As a result, we will maintain a front and a back for a queue. The idea is whenever we dequeue an item or peek at an item, we will use the front pointer and to enqueue an item we will use the back pointer. An example of a linked list-based queue is given below.

Algorithm for Queue using Linked List:

1. Create a class called `Node` with the following data members:
   - `elem`: to store the value of the node.
   - `next`: to point to the next node in the queue.

2. Initialize two variables: `front` and `rear`, both initially set to `NULL`.

3. Define the following operations:

   a. **Enqueue (Add an element to the queue):**
      - Create a new `Node` with data equal to the value to be enqueued.
      - If `rear` is `NULL`, set both `front` and `rear` to the new node.
      - Otherwise, set `rear->next` to the new node and update `rear` to the new node.

   b. **Dequeue (Remove an element from the queue):**
      - If `front` is `NULL`, return the underflow exception.
      - Store the current `front` in a temporary variable `temp`.
      - Move `front` to the next node (i.e., `front = front->next`).
      - If `front` becomes `NULL`, set `rear` to `NULL`.
      - Delete `temp` from memory.

   c. **Peek (Print all elements of the queue):**
      - If `front` is `NULL`, return the underflow exception.
      - Else return the front variable

4. End procedure.

Note that here, we are maintaining a front (for dequeue and peek) and a back (for enqueue). As the size of the linked list is not fixed, we are not maintaining the QueueOverflow exception in the given code.

Array Based Queue: If we want to create a queue with fixed size then an array based queue is a good approach. Here we will also use a front index and a back index. Initially both will start from the same index. After that, after every enqueue the back index will move one index further. After every dequeue the front index will move one index further. To keep indexing convenient we will use a circular array to implement the queue. To determine the queue overflow exception, we will maintain a size variable. **Array based queues are also known as buffers.** Buffer is used in many applications such as web server request management.

Algorithm for **Array Based Queue**:

1. Initialize an array `queue` of size `n`.
2. Initialize two variables: `front` and `rear`, both set to 0 (indicating an empty queue).
3. Define the following operations:

   a. **Enqueue (Add an element to the queue):**
      - If `rear` is equal to `n`, print "Queue is full" (overflow condition).
      - Otherwise, store the new element at `queue[rear]` and increment `rear` by 1.

   b. **Dequeue (Remove an element from the queue):**
      - If `front` is equal to `rear`, print "Queue is empty" (underflow condition).
      - Otherwise, shift all elements to the left by one position (i.e., `queue[i] = queue[i + 1]`) for `i` from 0 to `rear - 1`.
      - Decrement `rear` by 1.

   c. **Front (Get the front element from the queue):**
      - If `front` is equal to `rear`, print "Queue is empty."
      - Otherwise, print `queue[front]`.

   d. **Display (Print all elements of the queue):**
      - If `front` is equal to `rear`, print "Queue is empty."
      - Otherwise, traverse and print all elements from index `front` to `rear - 1`.

4. End procedure.

3.6 Queue Simulation

You need to do the following operations on a queue:

You need to perform the following operations on an array-based queue where the length of the array is 4 and the starting index of front and back is 3. [Blank means empty space].

enqueue a, enqueue b, dequeue, peek, enqueue c, peek, dequeue.

Index →	0	1	2	3	Front index	Back Index
Initial					3	3
enq (a)				a	3	0
enq (b)	b			a	3	1
deq	b				0	1
peek	b				0	1
enq (c)	b	c			0	2
peek	b	c			0	2
deq		c			1	2

Dequeued values: a, b

Peeked values: b, b

3.7 A Discussion on the Importance of Stacks and Queues

By now you well understand that larger complex data structures are made of smaller building block data structures. For example, multi-dimensional and circular arrays or vectors are made using linear arrays; lists are used to create sets and maps (which is called a dictionary in some languages and associated arrays in some others). However, as building block data structures, stacks and queues are unique in some sense.

For example, a set or a map that you create from a linked list provides additional features over their building block component that is the list. However, stacks and queues are strictly restricted forms of linked list (in rare cases, where you know the maximum number of elements you will put in them, you can use arrays for stacks and queues also). In other words, they do not provide any new features that a linked list does not already have. So it is a natural question why stacks and queues are considered fundamental data structures.

One reason for their importance is that scenarios where you need a linked list behave like a stack or queue is very frequent. For example, you need stacks for language parsing, expression evaluation, function calls, efficient graph traversals, etc. Queues are similarly important for resource scheduling, time sharing of CPU among candidate processes, message routing in network switches and routers, and efficient graph traversal.

Another central reason is that you often do not want to give access to the underlying list data structure that forms a stack or queue to the code that requested an insert or a removal from the stack/queue. If access to the list is provided then the code becomes more insecure. To consider this case, imagine direct access to the function call stack is possible. Then a program can manipulate the stack and jump from the current function to a much earlier function, instead of

the immediate predecessor function that called it. Why might this be a problem? Well, this would not be a problem if all the functions in the stack belong to your own program. However, in real-world, library functions, operating system's service functions interleave with user program's functions in the stack. Unprotected jumps to those functions can crash the system.

A third reason to have stack and queues as restricted forms of linked lists (or arrays) is that their restricted functions provide smaller code that can fit in very short memory or can even be implemented inside hardware devices directly. This feature is particularly important when you need fast response, for example, in routers and switches. In fact, array based stack and queue implementations are most suited in cases like this.

Exercises

3.1: Implement the push, pop and peek functions using an array.

3.2: Implement the push, pop and peek functions using a linked list.

3.3: Reverse a string using stack.

Example:
Input: ‘CSE220’
Output: ‘022ESC”

3.4: Check whether a string is palindrome or not using stack. If it is palindrome, print True, otherwise, print False.

Example:
Input: “MADAM”
Output: True

Input: “CSE220”
Output: False

3.5: The school cafeteria offers circular and square sandwiches at lunch break, referred to by numbers 0 and 1 respectively. All students stand in a queue. Each student either prefers square or circular sandwiches.

The number of sandwiches in the cafeteria is equal to the number of students. The sandwiches are placed in a **stack**. At each step:

- If the student at the front of the queue prefers the sandwich on the top of the stack, they will take it and leave the queue.
- Otherwise, they will leave it and go to the queue's end.

This continues until none of the queue students want to take the top sandwich and are thus unable to eat.

You are given two integer arrays students and sandwiches where sandwiches[i] is the type of the ith sandwich in the stack (i = 0 is the top of the stack) and students[j] is the preference of the jth student in the initial queue (j = 0 is the front of the queue). Return the number of students that are unable to eat.

Example 1:

Input: students = [1,1,0,0], sandwiches = [0,1,0,1]
Output: 0

Explanation:
- Front student leaves the top sandwich and returns to the end of the line making students = [1,0,0,1].

- Front student leaves the top sandwich and returns to the end of the line making students = [0,0,1,1].
- Front student takes the top sandwich and leaves the line making students = [0,1,1] and sandwiches = [1,0,1].
- Front student leaves the top sandwich and returns to the end of the line making students = [1,1,0].
- Front student takes the top sandwich and leaves the line making students = [1,0] and sandwiches = [0,1].
- Front student leaves the top sandwich and returns to the end of the line making students = [0,1].
- Front student takes the top sandwich and leaves the line making students = [1] and sandwiches = [1].
- Front student takes the top sandwich and leaves the line making students = [] and sandwiches = [].

Hence all students are able to eat.

Example 2:
Input: students = [1,1,1,0,0,1], sandwiches = [1,0,0,0,1,1]
Output: 3

3.6: Convert the following infix notation to postfix using stack following the given precedence of the operators. You must show the workings. You do not need to write code.

Consider the following precedence (decreases down the order)
1. * , /, //
2. %
3. + , -
4. == , <=, >=
5. &&

Now convert the following infix to its postfix notations:

1. A + B * C
2. A * B + C
3. A + B * C - D
4. A + B * (C - D / E)
5. A * (B + C) * D
6. A * B + C * D
7. A - B + C - D * E
8. (A - B + C) * (D + E * F)
9. A * (B + C - (D + E / F))
10. ((A + B) - C * (D / E)) + F

3.7: Convert the following postfix notations to its infix using stack. You must show the workings. You do not need to write code.

1. A B -
2. A B + C D + *
3. A B C * + D +
4. A B * C D * +
5. A B C + * D *

6. A B * C D * +
7. A B - C + D E * -
8. A B - C + D E F * + *
9. A B + C D E / * - F +
10. A B C D E / - * +

3.8: Evaluate the postfix expressions using stack. You must show the workings. You do not need to write code.

1. 3 4 * 2 5 * +
2. 2 3 - 4 + 5 6 7 * + *

3.9: Implement the enqueue, dequeue and peek functions using an array.

3.10: Implement the enqueue, dequeue and peek functions using a linked list.

3.11: Given a string s, find the first non-repeating character in it and return its index. If it does not exist, return -1.

Example:
Input: s = "programmingisfun"
Output: 0

Input: s = "pythonispopular"
Output: 1

Input: s = "aabb"
Output: -1

3.12: There are n people in a line queuing to buy tickets, where the 0th person is at the front of the line and the (n - 1)th person is at the back of the line.

You are given a 0-indexed integer array tickets of length n where the number of tickets that the ith person would like to buy is tickets[i].

Each person takes exactly 1 second to buy a ticket. A person can only buy 1 ticket at a time and has to go back to the end of the line (which happens instantaneously) in order to buy more tickets. If a person does not have any tickets left to buy, the person will leave the line.

Return the time taken for the person at position k (0-indexed) to finish buying tickets.

Example 1:
Input: tickets = [2,3,2], k = 2
Output: 6
Explanation:
- In the first pass, everyone in the line buys a ticket and the line becomes [1, 2, 1].
- In the second pass, everyone in the line buys a ticket and the line becomes [0, 1, 0].
The person at position 2 has successfully bought 2 tickets and it took 3 + 3 = 6 seconds.

Example 2:

Input: tickets = [5,1,1,1], k = 0
Output: 8
Explanation:
- In the first pass, everyone in the line buys a ticket and the line becomes [4, 0, 0, 0].
- In the next 4 passes, only the person in position 0 is buying tickets.
The person at position 0 has successfully bought 5 tickets and it took 4 + 1 + 1 + 1 + 1 = 8 seconds.

3.13: At a lemonade stand, each lemonade costs $5. Customers are standing in a queue to buy from you and order one at a time (in the order specified by bills). Each customer will only buy one lemonade and pay with either a $5, $10, or $20 bill. You must provide the correct change to each customer so that the net transaction is that the customer pays $5.

Note that you do not have any change in hand at first.

Given an integer array bills where bills[i] is the bill the ith customer pays, return true if you can provide every customer with the correct change, or false otherwise.

Example 1:
Input: bills = [5,5,5,10,20]
Output: true
Explanation:
From the first 3 customers, we collect three $5 bills in order.
From the fourth customer, we collect a $10 bill and give back a $5.
From the fifth customer, we give a $10 bill and a $5 bill.
Since all customers got correct change, we output true.

Example 2:
Input: bills = [5,5,10,10,20]
Output: false
Explanation:
From the first two customers in order, we collect two $5 bills.
For the next two customers in order, we collect a $10 bill and give back a $5 bill.
For the last customer, we can not give the change of $15 back because we only have two $10 bills.
Since not every customer received the correct change, the answer is false.

3.14: Given two strings s and goal, return true if you can swap two letters in s so the result is equal to goal, otherwise, return false.

Swapping letters is defined as taking two indices i and j (0-indexed) such that i != j and swapping the characters at s[i] and s[j]. For example, swapping at indices 0 and 2 in "abcd" results in "cbad".

Example 1:
Input: s = "ab", goal = "ba"
Output: true
Explanation: You can swap s[0] = 'a' and s[1] = 'b' to get "ba", which is equal to goal.

Example 2:
Input: s = "ab", goal = "ab"
Output: false
Explanation: The only letters you can swap are s[0] = 'a' and s[1] = 'b', which results in "ba" != goal.

Example 3:
Input: s = "aa", goal = "aa"
Output: true
Explanation: You can swap s[0] = 'a' and s[1] = 'a' to get "aa", which is equal to goal.

3.15: You are given an integer array deck. There is a deck of cards where every card has a unique integer. The integer on the ith card is deck[i]. You can order the deck in any order you want. Initially, all the cards start face down (unrevealed) in one deck. You will do the following steps repeatedly until all cards are revealed:

Take the top card of the deck, reveal it, and take it out of the deck.
If there are still cards in the deck then put the next top card of the deck at the bottom of the deck.
If there are still unrevealed cards, go back to step 1. Otherwise, stop.
Return an ordering of the deck that would reveal the cards in increasing order.

Note that the first entry in the answer is considered to be the top of the deck.

Example 1:
Input: deck = [17,13,11,2,3,5,7]
Output: [2,13,3,11,5,17,7]
Explanation:
We get the deck in the order [17,13,11,2,3,5,7] (this order does not matter), and reorder it.
After reordering, the deck starts as [2,13,3,11,5,17,7], where 2 is the top of the deck.
We reveal 2, and move 13 to the bottom. The deck is now [3,11,5,17,7,13].
We reveal 3, and move 11 to the bottom. The deck is now [5,17,7,13,11].
We reveal 5, and move 17 to the bottom. The deck is now [7,13,11,17].
We reveal 7, and move 13 to the bottom. The deck is now [11,17,13].
We reveal 11, and move 17 to the bottom. The deck is now [13,17].
We reveal 13, and move 17 to the bottom. The deck is now [17].
We reveal 17.
Since all the cards revealed are in increasing order, the answer is correct.

Chapter 4: Hashing and Hashtable

4.1 Introduction

We have learned array and linked list as primary data structures and then learned how we can use those primary data structures to create Stack and Queue. We have also learned the merits and demerits of those different data structures. One very important aspect is that in both array and linked list, our data is stored against some index value for our using purpose.

Another important concept of storing data is using **key-value** pairs to store data instead of storing against an index. The idea is to find the data in constant time using that key. You have already seen this type of data structure in the Python dictionary. The concept is also used in JSON (JavaScript Object Notation). Now the question is how we can achieve this. The problem with our existing knowledge is that if we use linear search in the array to find the object it takes linear time O(n) to find it. If we use a sorted array to use the concept of counting sort it also takes linear time (for simplicity) to sort the array which is O(n).

4.2 Structure of Hashtable

The idea of hashing and hash table data structure comes to solve this issue. The idea of hashing is that there will be a function that will map any key as an integer, that integer will be used as an index of an array, and both key and value will be stored in that index. The array where both key and values are stored is known as a hash table. For example, we may want to store “name”: “Naruto Uzumaki” where the key is “name” and the value is “Naruto Uzumaki”. It will be stored in a hash table with length 5 (which means the array’s length is 5). Now the hash function will convert the string “name” and map it to an integer. E.g.: using summation ASCII value of “name” and then mod it by 5 (in order to get a valid index). We get the index (110 + 97 + 109 + 101) = 417 % 5 = 2. Now in that particular index, the array will store both “name” as key and “Naruto Uzumaki” as value. Now you may wonder why we are storing both key and value, the reason is we need to distinguish between different keys which map to the same index. To illustrate, if we have another key “mean”, it will also map to index 2 and so we need to store data in such a way that the values of the “name” key and “mean” key are stored separately.

Now the question comes to how to search an item if multiple keys are mapped into the same index. One approach might be to use a hash function in such a way that all the keys are mapped to different distinguished indexes (one-to-one function). However, the problem with such an approach is that it will waste lots of space because many indexes will remain empty. To deal with this problem, hash functions usually use modulus operation with the array size to get a valid index. This creates the problem of multiple keys getting mapped into the same index as discussed above. The event is known as Collision and the hash table has some mechanism to deal with collision handling.

4.3 Collision Handle

One of the popular methods of collision handling is called Forward Chaining. The idea of Forward Chaining is to store a linked list in the index. As a result, whenever a collision occurs, the value is stored at the beginning of the linked list in that particular index. It is stored at the beginning of the linked list to decrease the insertion time. Whenever we search for an item, it will traverse the list and check the key to find the value. One limitation of this method is that the search time is longer if collisions occur frequently. For this reason, choosing the appropriate hash function to minimize the collision is very important.

Another method for collision handling is to use Linear Probing. It stores the data in the next available index once the collision has occurred. However, this process increases the chance of collision and so another technique is to use double hashing. The idea of this method is to use another hash function if we find any collision. The first hash function determines the key and the second hash function determines the jump from the given key if collision occurs.

4.4 Hashtable Example

Now we will see an example of hash table generation using forward chaining as a collision technique. In this example, we'll use a basic hash function to map keys to their corresponding hash index and use linked lists to handle collisions using forward chaining.

Let's assume we have a hash table of size 5, and the hash function is a simple modulo operation to convert keys to hash indices.

Hash Function: hash(key) = key % 5

Now, let's insert some key-value pairs into the hash table:

Insert (Key: 12, Value: "Apple")
Insert (Key: 5, Value: "Orange")
Insert (Key: 17, Value: "Banana")
Insert (Key: 10, Value: "Grapes")
Insert (Key: 22, Value: "Watermelon")
Insert (Key: 15, Value: "Pineapple")
Here's the resulting hash table after each insertion:

Step 1: Insert (Key: 12, Value: "Apple") index 2

Index	Key-Value Pair
2	(12, "Apple")

Step 2: Insert (Key: 5, Value: "Orange") index 0

Index	Key-Value Pair
2	(12, "Apple")
0	(5, "Orange")

Step 3: Insert (Key: 17, Value: "Banana") Collision at index 2

Index Key-Value Pair
2 (17, "Banana") -> (12, "Apple") #Forward chaining with collision
0 (5, "Orange")

Step 4: Insert (Key: 10, Value: "Grapes") Collision at index 0

Index Key-Value Pair
2 (17, "Banana") -> (12, "Apple")
0 (10, "Grapes") -> (5, "Orange")

Step 5: Insert (Key: 22, Value: "Watermelon") Collision at index 2

Index Key-Value Pair
2 (22, "Watermelon") -> (17, "Banana") -> (12, "Apple")
0 (10, "Grapes") -> (5, "Orange")

Step 6: Insert (Key: 15, Value: "Pineapple") - Collision at index 0

Index Key-Value Pair
2 (22, "Watermelon") -> (17, "Banana") -> (12, "Apple")
0 (15, "Pineapple") -> (10, "Grapes") -> (5, "Orange")

In this example, when a collision occurs, the new key-value pair is added to the linked list at the same index (using forward chaining) instead of overwriting the existing entry. This way, all key-value pairs with the same hash index can be accessed by traversing the linked list.

Forward chaining is a simple and effective way to handle collisions in a hash table, and it ensures that all key-value pairs are retained without overwriting or losing any data.

4.5 Advance Topic

Cryptographic Hashing: Cryptographic hashing, also known as one-way hashing or hash function, is a fundamental concept in computer science and cryptography. It is a mathematical function that takes an input (or "message") of arbitrary size and produces a fixed-size string of characters, typically represented in hexadecimal format. This fixed-size output is often referred to as the hash value, hash code, or simply hash.

Key properties of cryptographic hashing functions are:

Deterministic: For a given input, the hash function always produces the same output. This property is crucial for consistency and predictability.

Irreversibility (One-way): The hash function is designed to be irreversible, meaning it should be computationally infeasible to derive the original input from its hash value. This property ensures that the original data remains secure even if the hash is known.

Collision resistance: It should be extremely difficult to find two different inputs that produce the same hash value. In other words, it should be computationally infeasible to create a collision intentionally.

Cryptographic hashing has numerous applications in information security and computer science, including but not limited to:

1. Password storage: Storing passwords securely by hashing them, preventing direct exposure of user passwords in databases.

2. Data integrity verification: Verifying that data has not been tampered with by comparing the hash of the original data with the computed hash of the received data.

3. Digital signatures: Used in digital signature algorithms to ensure authenticity and integrity of messages or documents.

4. Secure storage and retrieval: Employed in blockchain technology to create secure links between blocks of data.

Common cryptographic hashing algorithms include MD5 (Message Digest Algorithm 5), SHA-1 (Secure Hash Algorithm 1), SHA-256 (part of the Secure Hash Algorithm 2 family), and SHA-3. It's important to note that as computing power increases and new cryptographic vulnerabilities are discovered, older hashing algorithms may become less secure, and it's advisable to use the most up-to-date and robust hashing functions available.

Chapter 5: Tree

5.1 Tree Basics

Tree is a non-linear data structure that allows us to organize, access and update data efficiently by representing non-linear data in a form of hierarchy.

5.1.1 Introduction to Tree Data Structure

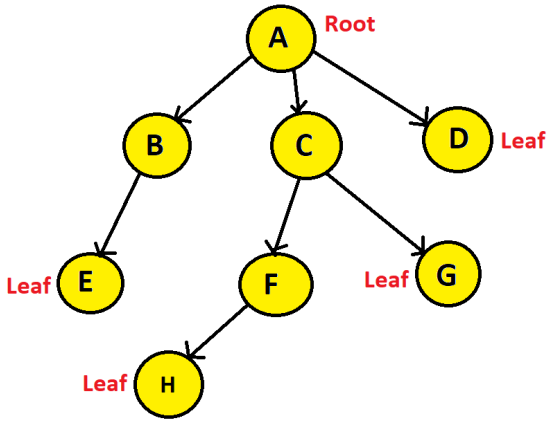
We have covered a number of data structures so far in this course. However, trees are extremely unique as some feats can only be achieved efficiently using trees. Following are some real life applications of trees:

- Continuous Sorting can be achieved using trees. Imagine you have a sorted array of 1 million data. Now you have to add 1000 more data into it. Then comes the responsibility of sorting the entire array again. However, using one variation of trees (Binary Search Tree), we can efficiently insert data later in a sorted manner without having to sort all the data again and again.
- Trees are widely used in folder/file system structure. For example, your desktop folder is the root of the tree and in your desktop you have multiple folders, and in those folders, you have some files. These files are the leaf nodes of the tree.
- Trees are used in electrical circuit designing and electricity transmission. For example, suppose you have a main center point from where all the electricity is generated and from there on it will be spread into branches until it reaches every home, office or industry.
- Router/Computer Network algorithms construct trees of the locations across the network to determine the route that data packets must follow to reach their destination efficiently.

5.1.2 Terminologies of a Tree

5.1.2.1 Root/Leaf/Non-Leaf Node:

Each tree consists of one or more nodes that are interlinked. The topmost node is called the **root** node. Below the root there are one or more nodes. The nodes residing at the bottom, or in other words the nodes that do not lead to any other node are called **leaf** nodes. If we have access to the root node, we have access to the entire tree.



Here we can see that node A is the root node, Nodes E, H, G, and D are the leaf nodes. Nodes A, B, F, and C are non-leaf nodes. The root node and the non-leaf nodes can be considered as internal nodes.

5.1.2.2 Parent/Child:
The direction of the tree is from top to bottom. Which means Node B is the immediate successor of node A, and Node A is the immediate predecessor of node B. Therefore, node B is the child of A, whereas, A is the parent of B.

5.1.2.3 Siblings:
All the nodes having the same parent are known as siblings. Therefore, B, C, and D are siblings, and F and G are siblings.

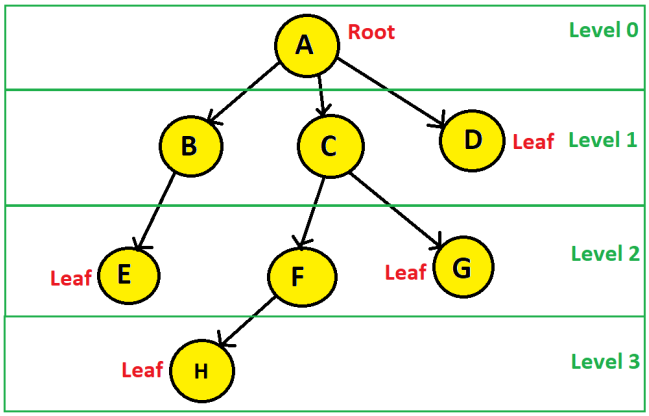
5.1.2.4 Edge/Path:
The link between two nodes is called an edge. A path is known as the consecutive edges from the source node to the destination node. So, if we asked what is the path from node A to E? The answer would be A→B→E. A tree having n number of nodes will have (n-1) number of edges. Here, we have 8 nodes and 7 edges in total.

5.1.2.5 Degree:
The number of children of a node is known as degree. The degree of node A is 3, node B is 1 and Node E is 0.

5.1.2.6 Depth:
The length of the path from a node to the root node is known as the depth. The depth of nodes E, F, and G is 2; depth of B, C, and D is 1; depth of A is 0; depth of H is 3.

5.1.2.7 Height:
The length of the longest path from a node to one of its leaf nodes is known as the height. From node A to the leaf nodes there are four paths: A→B→E, A→C→F→H, A→C→G, and A→D. Of these four paths, A→C→F→H is the longest path. Hence, the height of Node A is 3.

5.1.2.8 Level:
Each hierarchy starting from the root is known as the level.



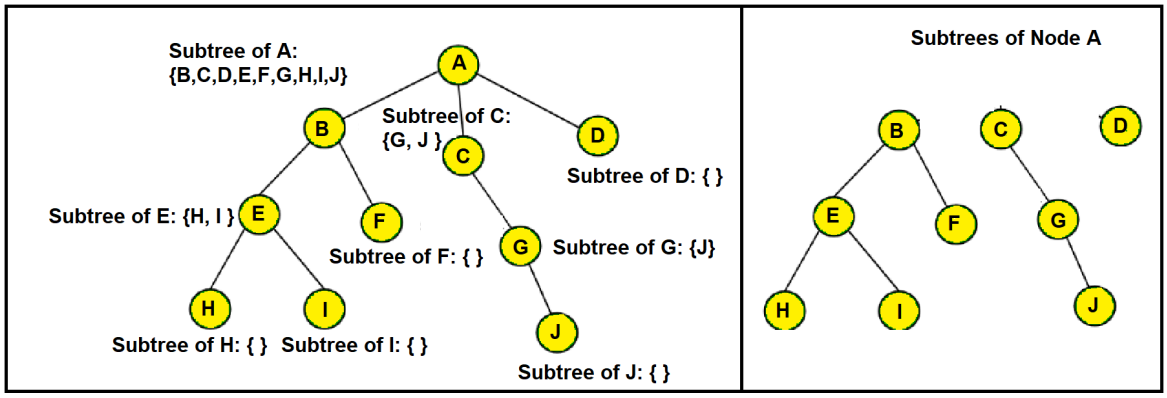
From the above figure, we can see that the level of node A is 0; level of nodes B, C, and D is 1; level of nodes E, F, and G is 2; and level of node H is 3.

Points to remember:

- *The depth and height of a node may not be the same. The depth of A is 0, whereas, the height of A is 3.*
- *Level of a node == Depth of that node.*

5.1.2.9 Subtree:

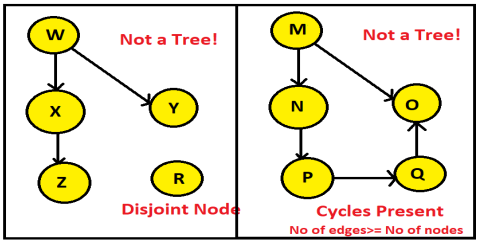
A tree that is the child of a Node.



Any tree can be further divided into subtrees with respect to a particular node. Here node A has three subtrees that are shown on the right side.

5.1.3 Characteristics of a Tree

- A tree must be continuous and not disjoint, which means every single node must be traversable starting from the root node.
- A tree cannot have a cycle. A tree having n number of nodes will have n or more edges if it contains one or more cycles. This means, for a tree, no of edges == no of nodes - 1.



5.1.4 Tree Coding - Tree Construction using Linked List (Dynamic Representation)

A tree can be represented using a linked list (Dynamic Representation) or an array (Sequential Representation). Here we shall see how to dynamically represent trees.

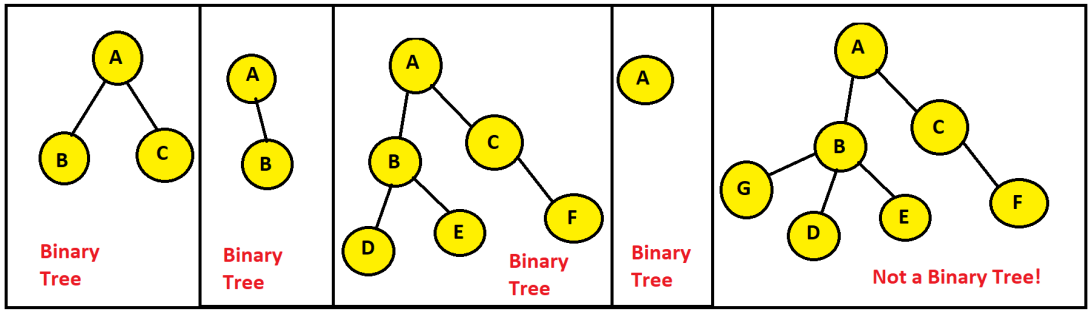
1. Create a class called `Node` with the following data members:
   - `elem`: to store the value of the node.
   - an array named `children`: to store the references of all the children.

2. The first node created becomes the root of the tree

3. After creating each child node, their reference must be stored at their parent node's children array.

5.2 Binary Tree

A tree is a binary tree if every single node of the tree has at most 2 child nodes.



5.3 Characteristics of a Binary Tree

- Each node in a binary tree can have at most two child nodes
- If the number of internal nodes is n, number of external nodes is n+1, number of edges is 2n, number of internal edges is n-1, number of external edges is n+1.
- The maximum number of nodes possible in a binary tree of height 'h' is: $2^{h+1} - 1$
- The maximum number of nodes at level i is: 2^{i}

5.4 Binary Tree Traversal: Pre-order, In-order, Post-order

Pre-order

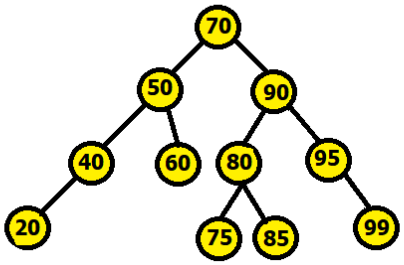
Whenever a node is visited for the **first** time, its element is printed. We start from the root and print its element. Then its left subtree is traversed. If a node does not have a left child, we return to that node After all the nodes of its entire left subtree have been traversed thrice, we head back to the root node for the second time and check if it has any right child, and if it does not have a right child either, we again return to that node After all the nodes of its entire left subtree have been traversed thrice, we head back to the root node for the third time and then go towards its parent node. After all the nodes of its entire left subtree have been traversed twice, we head back to the root node. After that, its right subtree is traversed. The root is printed at first.

In-order

Whenever a node is visited for the **second** time, its element is printed. We start from the root and traverse its left subtree. If a node does not have a left child, we return to that node for the second time and print its element. Then we check if it has any right child, and if it does not have a right child either, we again return to that node for the third time and then go towards its parent node. After all the nodes of its entire left subtree have been traversed thrice, we head back to the root node for the second time and print its element. The root is printed after all the nodes of its left subtree are printed. After that, its right subtree is traversed. After all the nodes of its entire right subtree have been traversed thrice, we head back to the root node for the third time. **In-order traversal also sorts the data in ascending order.**

Post-order

Whenever a node is visited for the **third** time, its element is printed. We start from the root and traverse its left subtree. If a node does not have a left child, we return to that node for the second time and check if it has any right child, and if it does not have a right child either, we again return to that node for the third time, print its element and then go towards its parent node. After all the nodes of its entire left subtree have been traversed thrice, we head back to the root node and then traverse its right subtree. After all the nodes of its entire right subtree have been traversed thrice, we head back to the root node for the third time and print its element. The root is printed at the last.



Pre-Order: 70, 50, 40, 20, 60, 90, 80, 75, 85, 95, 99

In-Order: 20, 40, 50, 60, 70, 75, 80, 85, 90, 95, 99

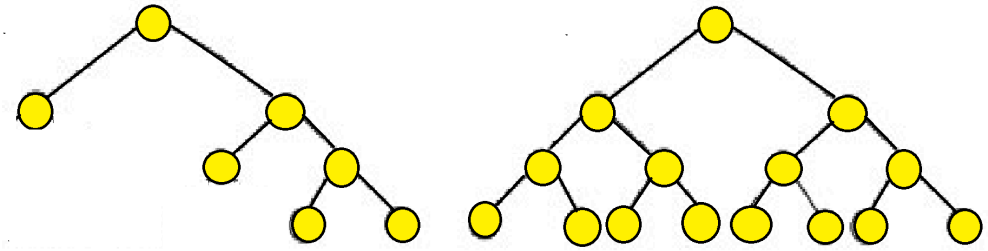
Post-Order: 20, 40, 60, 50, 75, 85, 80, 99, 95, 90, 70

5.5 Types of a Binary Tree

5.5.1 Full/Strict Binary Tree

In a full binary tree, internal nodes (every node except the leaf nodes) have two children. How can we identify a full binary tree? Any binary tree that maintains the following condition is a full binary tree:

No of leaf nodes = no of internal nodes + 1



Full Binary Tree

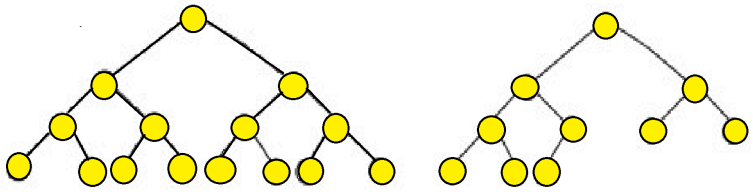
Here, in the leftmost tree, the no of internal nodes is 3 and the no of leaf nodes is 4. Again in the rightmost tree, the no of internal nodes is 7 and the no of leaf nodes is 8.

5.5.2 Complete Binary Tree

In a complete binary tree, all the levels are filled entirely with nodes, except the lowest level of the tree. Also, in the lowest level of this binary tree, every node should possibly reside on the left side.

How can we identify a complete binary tree?

- If all the internal nodes (every node except the leaf nodes) have two childs, then it is a complete binary tree.
- If any of the internal nodes has only one child, the child must reside in the left side and not in the right side.

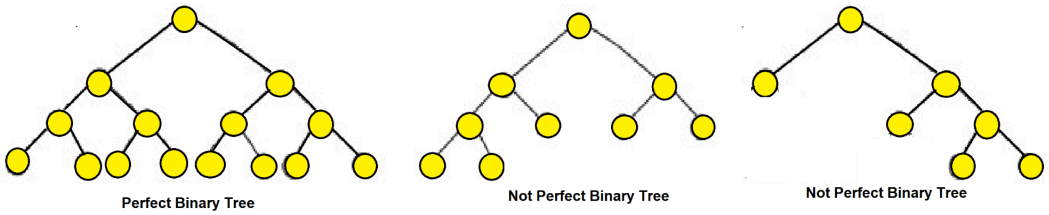


Complete Binary Tree

In the leftmost tree, all internal nodes have two children. Therefore, it is a complete binary tree. In the rightmost tree, all internal nodes except one have two children. The only internal node that has one child, has its child residing on its left side. Therefore, it is also a complete binary tree.

5.5.3 Perfect Binary Tree

In a perfect binary tree, every internal node has exactly two child nodes and all the leaf nodes are at the same level.



Perfect Binary Tree

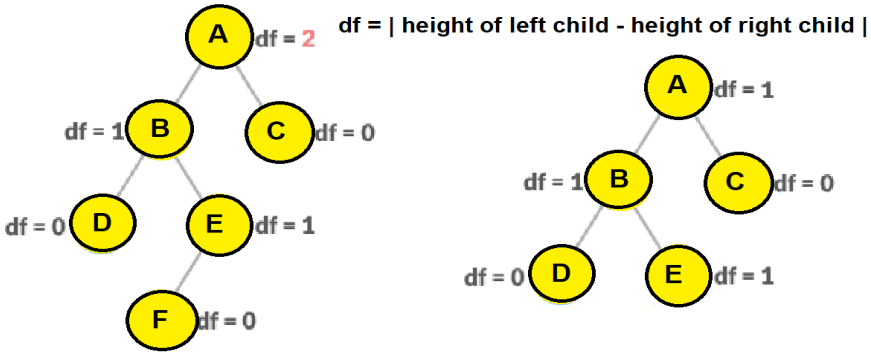
Not Perfect Binary Tree

Not Perfect Binary Tree

In the tree in the middle, all the leaf nodes are not on the same level. In the rightmost tree, not all internal nodes have two children.

5.5.4 Balanced Binary Tree

In a balanced binary tree, the height of the left and right subtree of any node differ by not more than 1. Balanced binary trees are also referred to as a height-balanced binary tree.



Unbalanced Tree

Balanced Tree

In the leftmost tree, the height of Node A's left subtree is 2 and right subtree is 0. Therefore, the difference between these two is 2. On the other hand, in the rightmost tree, no nodes have a height difference of more than 1 in between their left and right subtrees.

Points to remember:

- *Every perfect binary tree is also a full binary tree, but every full binary tree is not a perfect binary tree.*

5.6 Binary Tree Coding

5.6.1 Tree Construction using Array (Sequential Representation)

Have you ever thought about how to represent binary trees in your program? If you can recall, we have been using linked lists so far to represent trees. So, there are two ways to represent binary trees:

1. Dynamic Representation (Using Linked List)
2. Sequential Representation (Using Array)

We have already covered the dynamic representation of trees. Now let us look at how to sequentially represent binary trees.

Sequential Representation (Using Array) Conditions:

I. If the height of the binary tree is h, An array of maximum 2^{h+1} length is required.
II. The root is placed at index 1.
III. Any node that is placed at index i, will have its left child placed at 2i and its right child at 2i+1.

Pseudocode of Array representation of a Binary Tree:

The numbering of nodes can start either from 0 to (n-1) or 1 to n.

Let's assume we're numbering nodes from 1 to n for this example:

1. Initialize an array tree of size n (assuming a maximum of n nodes).
2. Call a method and send the root of the tree, the array and 1 (index) as parameters
3. If the root is not NULL, Set the root node value at the given index of the array.
4. For any node at index i:
    a. Its left child should be at index (2*i). Recursive call the method and send the left child, the array and 2*i
    b. Its right child should be at index (2*i) + 1. Recursive call the method and send the right child, the array and 2*i+1

Pseudocode of Creating a Binary Tree from an Array:

The numbering of nodes can start either from 0 to (n-1) or 1 to n.

Let's assume we're numbering nodes from 1 to n for this example:

1. Call a method and send the array and 1 (index) as parameters

2. While index is not greater than the array length:
    a. If the element in the given index of the array is not NULL, create the root node using the element in that index.
    b. For any node at index i:
        i. Its left child should be at index (2*i). Recursive call the method and send the array and 2*i and assign the function call as the left child.
        ii. Its right child should be at index (2*i) + 1. Recursive call the method and send the array and 2*i+1 and assign the function call as the right child.
    c. Return the root

5.6.2 Level, Height, and Depth Finding

```
Pseudocode of get_height(node):
  if, node is null, return -1
  else, return 1 + maximum of
(recursive call with node's left child + recursive call with node's right child)
```

5.6.3 Number of Nodes Finding

```
Pseudocode of get_no_of_nodes (node):
  if node is null, return 0
  else, return 1 + recursive call with node's left child
+ recursive call with node's right child
```

5.6.4 Identifying Tree Types: Full, Complete and Perfect

Full Binary Tree or Not	Complete Binary Tree or Not
function isFullTree(root) if root is null return true if root has no children return true if root has two children	function isCompleteTree(root) if root is null return true create an empty queue and enqueue root create a flag to indicate if a non-full node is seen while queue is not empty

```
    return isFullTree(root.left) and isFullTree(root.right)
  return false
```

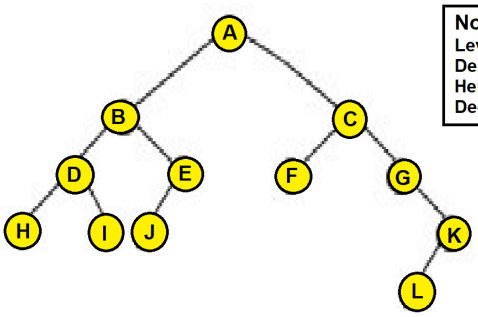
```
    dequeue a node from the queue
    if node has left child
      if flag is true
        return false
      enqueue node's left child
    else
      set flag to true
    if node has right child
      if flag is true
        return false
      enqueue node's right child
    else
      set flag to true
  return true
```

Perfect Binary Tree or Not	Balanced Binary Tree or Not

function isPerfectTree(root)
  if root is null
  return true
  if root has no children
  return true
  if root has one child
  return false
  return isPerfectTree(root.left) and isPerfectTree(root.right) and getDepth(root.left) == getDepth(root.right)
	function isBalanced(root)
	  if root is null
	  return true
	  get the height of left subtree as leftHeight
	  get the height of right subtree as rightHeight
	  if the absolute value of leftHeight - rightHeight is more than one
	  return false
	  return isBalanced(root.left) and isBalanced(root.right)

Exercises

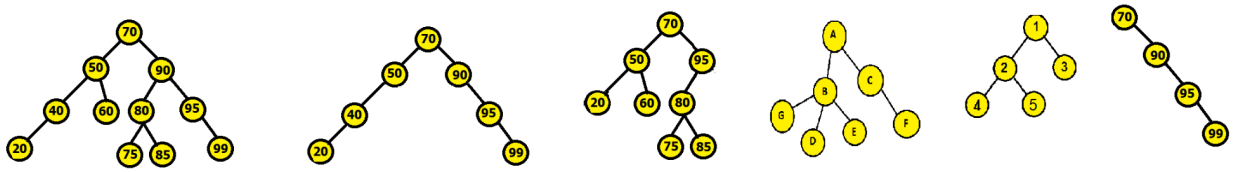
5.1: Find the level, depth, height and degree of the specified nodes of the following tree.



Node A	Node D	Node F	Node C
Level:	Level:	Level:	Level:
Depth:	Depth:	Depth:	Depth:
Height:	Height:	Height:	Height:
Degree:	Degree:	Degree:	Degree:

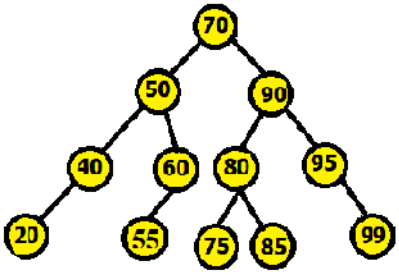
Node E	Node G	Node L
Level:	Level:	Level:
Depth:	Depth:	Depth:
Height:	Height:	Height:
Degree:	Degree:	Degree:

5.2: Identify which of the following trees are full, complete, perfect and balanced.



5.3: Write the code to construct a tree of height 3 and minimum number of 9 nodes. Use your imagination while designing the tree.

5.4: Traverse the following trees in pre-order, in-order and post-order and print the elements. Show both simulation and code.

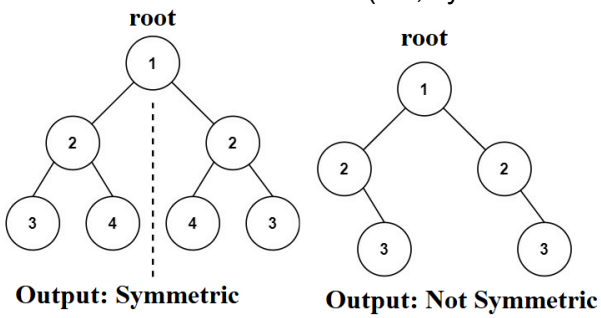


Pre-order:
In-order:
Post-order:

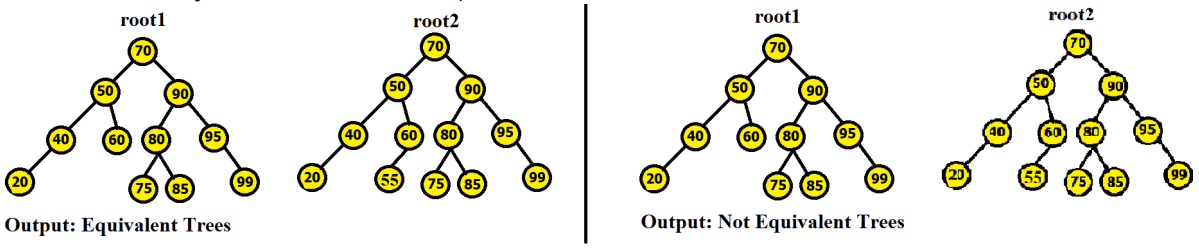
5.5: Consider the following array and convert it into a binary tree. Show simulation and code.

```
[None, 15, 25, 35, 10, 35, 15, 18, None, None, None, 33, None, 5, None, 19, None, None, None, 16]
```

5.6: Write a Python function **isSymmetric(root)** that takes the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).



5.7: Write a Python function **isIdentical(root1, root2)** that takes the roots of two binary trees, check whether they are identical or not).



5.8: Given two binary trees, find if both of them are identical or not.

Input:

```
    1          1
  /   \      /   \
 2     3    2     3
```

Output: Yes

Explanation: There are two trees both having 3 nodes and 2 edges, both trees are identical having the root as 1, left child of 1 is 2 and right child of 1 is 3.

Input:

```
    1        1
  /   \    /   \
 2     3  3     2
```

Output: No

Explanation: There are two trees both having 3 nodes and 2 edges, but both trees are not identical.

5.9: Given a binary tree, convert it into its mirror.

Input:

```
      10
     /  \
    20   30
   /  \
  40  60
```

Output: 30 10 60 20 40

Explanation: The tree is

```
      10                10
    /    \  (mirror)  /    \
   20    30    =>    30    20
  /  \                /  \
 40  60              60   40
```

The inroder traversal of mirror is 30 10 60 20 40.

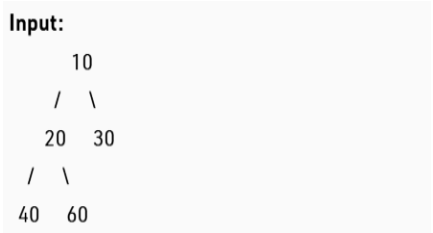
5.10: Given a binary tree, find if it is height balanced or not. A tree is height balanced if the difference between heights of left and right subtrees is not more than one for all nodes of the tree.

Input:

Output: 0

Explanation: The max difference in height of left subtree and right subtree is 2, which is greater than 1. Hence unbalanced

Input:



Output: 1

Explanation: The max difference in height of left subtree and right subtree is 1. Hence balanced.

5.11: Given a binary tree, check whether all of its nodes have the value equal to the sum of their child nodes.

Input:

Output: 1

Explanation: Here, every node is sum of its left and right child.

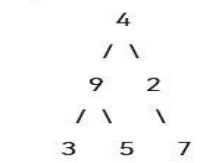
Input:

Output: 0

Explanation: Here, 1 is the root node and 4, 3 are its child nodes. 4 + 3 = 7 which is not equal to the value of root node. Hence, this tree does not satisfy the given conditions.

5.12: Given a binary tree, find the largest value in each level.

Input :



Output : 4 9 7

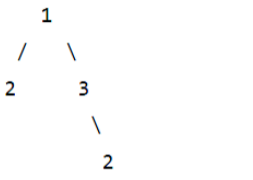
Explanation :

There are three levels in the tree:

1. {4}, max = 4
2. {9, 2}, max = 9
3. {3, 5, 7}, max=7

5.13: Given a binary tree, check if it has duplicate values.

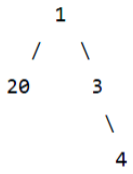
Input : Root of below tree



Output : Yes

Explanation : The duplicate value is 2.

Input : Root of below tree

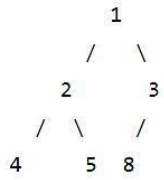


Output : No

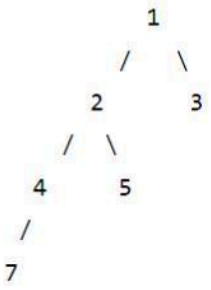
Explanation : There are no duplicates.

5.14: Given a root of a binary tree, and an integer k, print all the nodes which are at k distance from root. Distance is the number of edges in the path from the source node (Root node in our case) to the destination node.

For example, in the below tree, 4, 5 & 8 are at distance 2 from root.

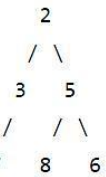


5.15: Given a binary tree and a key, write a function that prints all the ancestors of the node with the key in the given binary tree. For example, if the given tree is following binary Tree and the key is 7, then your function should print 4, 2, 1.



5.16: Given a binary tree, print all the nodes having exactly one child. Print "-1" if no such node exists.

Input:

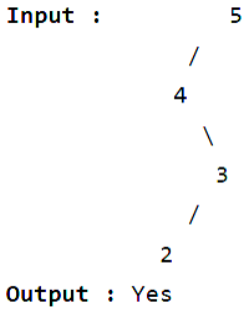


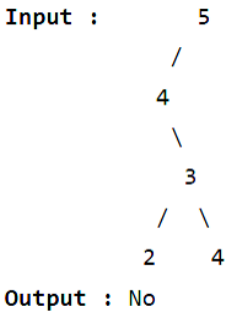
Output: 3

Explanation:

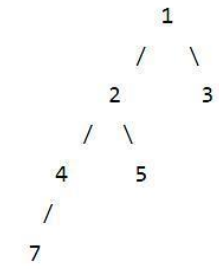
There is only one node having single child that is 3.

5.17: Given a binary tree, check whether it is a skewed binary tree or not. A skewed tree is a tree where each node has only one child node or none.

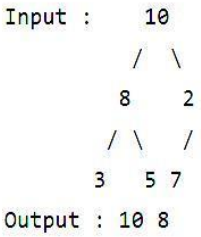




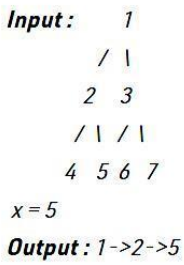
5.18: Given a binary tree, find its minimum depth. The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node. For example, the minimum depth of the below Binary Tree is 2.



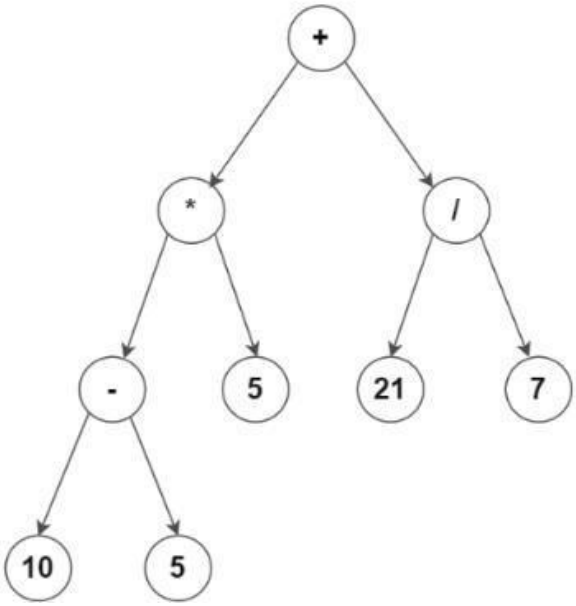
5.19: Given a binary tree, print all nodes that are full nodes. Full nodes are nodes which have both left and right children as non-empty.



5.20: Given a binary tree with distinct nodes(no two nodes have the same data values). The problem is to print the path from root to a given node **x**. If node **x** is not present then print "No Path".



5.21: Evaluate a given binary expression tree representing algebraic expressions. A binary expression tree is a binary tree, where the operators are stored in the tree's internal nodes, and the leaves contain constants. Assume that each node of the binary expression tree has zero or two children. The supported operators are +, -, * and /.
For example, the value of the following expression tree is 28.



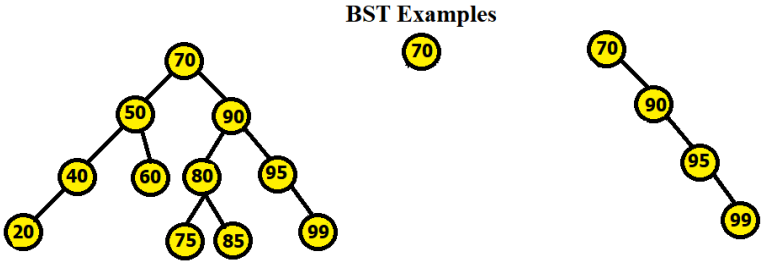
Chapter 6: Binary Search Tree (BST) and Heap

6.1 Characteristics of a BST

Binary Search Tree is a binary tree data with the following fundamental properties:

1. The left subtree of a node contains only nodes with keys lesser than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. The left and right subtree each must also be a binary search tree.
4. Each node must have a distinct key, which means no duplicate values are allowed.

The goal of using BST data structure is to search any element within O(log(n)) time complexity.

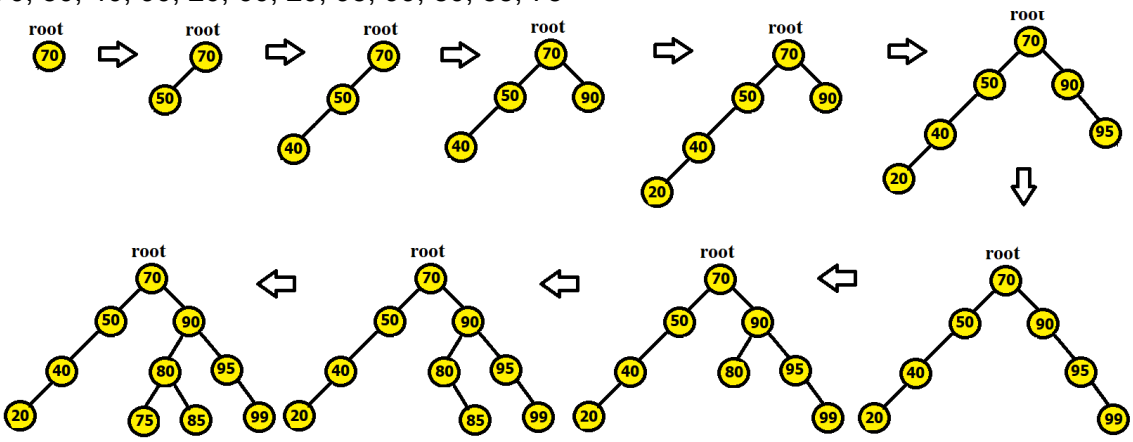


6.2 Basic Operations on a BST

Any operation done in a BST must not violate the fundamental properties of a BST.

6.2.1 Creation of a BST

Draw the BST by inserting the following numbers from left to right:
70, 50, 40, 90, 20, 60, 20, 95, 99, 80, 85, 75



6.2.2 Inserting a Node

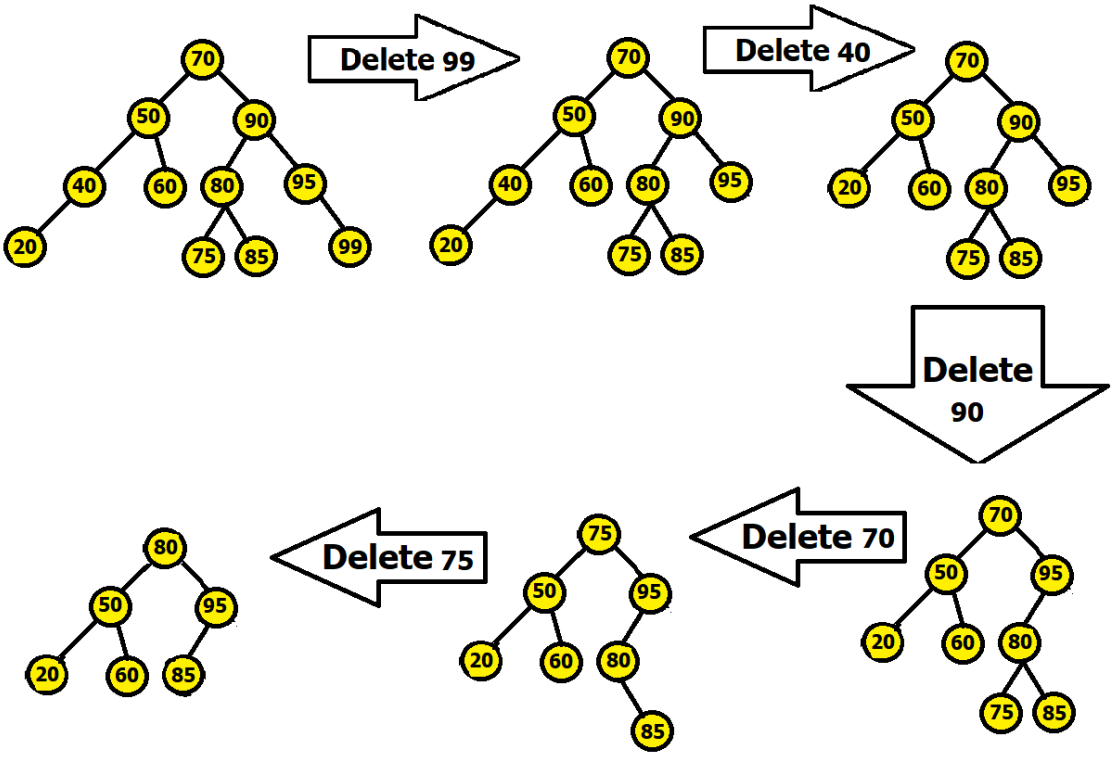
While inserting a node to an existing BST, the process is similar to that of creating a BST. We take the new node data which must not be repetitive or already present in the BST. We start comparing it with the root node, if the new node data is greater we go towards the right subtree of the root, if smaller we go towards the left subtree. We keep on going like this until we find a free space, and then make a node using the new node data and attach the new node at the

vacant place. Note that, after insertion, a balanced BST may become **unbalanced** and therefore any searching operation done on it may take a lot longer than O(log(n)). We have to balance the tree if it becomes unbalanced.

6.2.3 Removing a Node

3 possible cases can occur while deleting a node:

1. *Case 1* | No subtree or children: This one is the easiest one. You can simply just delete the node, without any additional actions required.
2. *Case 2* | One subtree (one child): You have to make sure that after the node is deleted, its child is then connected to the deleted node's parent.
3. *Case 3* | Two subtrees (two children): You have to find and replace the node you want to delete with its leftmost node in the right subtree (inorder successor) or rightmost node in the left subtree (inorder predecessor).



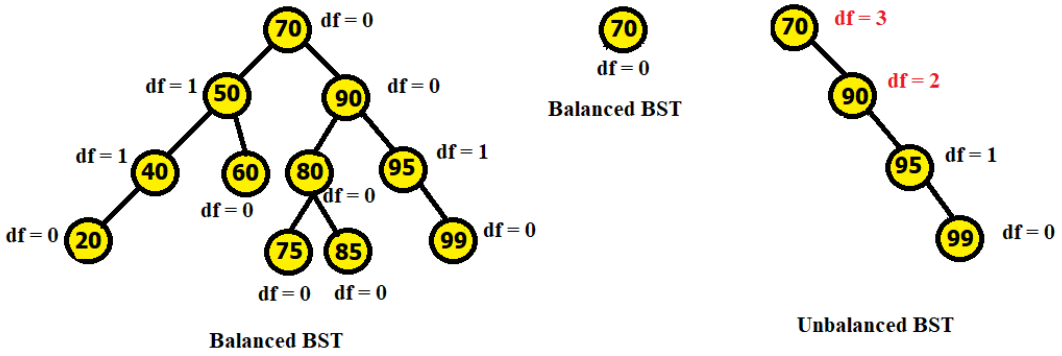
Here deletion of 99 falls under case 1, deletion of 40 falls under case 2, and deletion of 90, 70 and 75 fall under case 3. While deleting 75, we replaced 75 with its leftmost child from its right subtree, 80. After that 85 was put in the 80's previous place.

Note that, after deletion, a balanced BST may become **unbalanced** and therefore any searching operation done on it may take a lot longer than O(log(n)). We have to balance the tree if it becomes unbalanced.

6.3 Balanced vs Unbalanced BST

If the height difference between left and right subtree of any node in the BST is more than one, it is an unbalanced BST. Otherwise, it is a balanced one. In a balanced BST, any searching operation can take upto **O(log(n))** time complexity. In an unbalanced one, it may take upto **O(n)** time complexity, which renders the usage of BST obsolete. Therefore, we should only work with balanced BST and after conducting any operation on a BST, we must first check if it became unbalanced or not. If it does become unbalanced, we have to balance it.

df = | height of left child - height of right child |

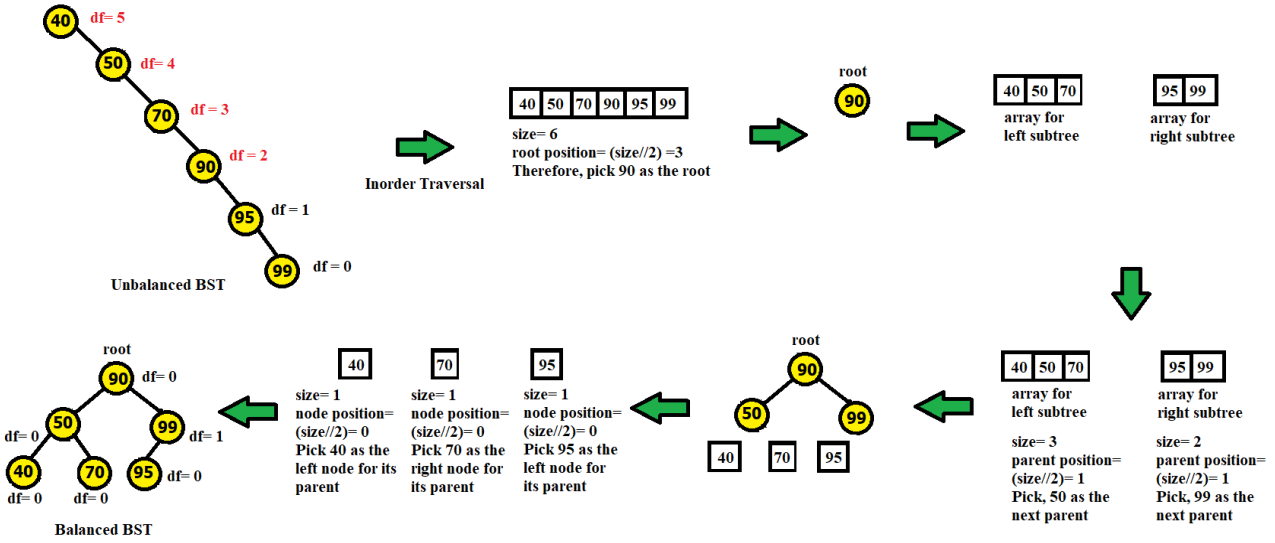


How to convert an unbalanced BST into a balanced BST?

I. Traverse given BST in inorder and store result in an array. This will give us the ascending sorted order of all the data.

II. Now take the data in the (size//2) position in the array and make it the root. Now the left subtree of the root will be all the data residing in from 0 to (size//2)-1 positions of the array. The right subtree of the root will be all the data residing in from (size//2)+1 to (size-1) positions of the array.

III. Now again choose the middlemost values from the left subtree and right subtree and connect these to the root. Keep on repeating the process until all the elements of the array have been taken.



6.4 BST Coding

6.4.1 Creating a BST / Inserting a Node

Pseudocode

```
function createBST(values)
  if values is empty
    return null
  create a new node with values[0] as data
  for i from 1 to values.length - 1
    insert node into BST with values[i] as data
  return node

function insert(node, data)
  if node is null
    return new node with data
  if data < node.data
    node.left = insert(node.left, data)
  else
    node.right = insert(node.right, data)
  return node
```

6.4.2 BST Traversal: Pre-order, In-order, Post-order

Pre-Order	In-Order	Post-Order

function preorder(root)
  if root is null
  return
  print root.data
  preorder(root.left)
  preorder(root.right)
	function inorder(root)
	  if root is null
	  return
	  inorder(root.left)
	  print(root.data)
	  inorder(root.right)
		function postorder(root)
		  if root is null
		  return
		  postorder(root.left)
		  postorder(root.right)
		  print(root.data)

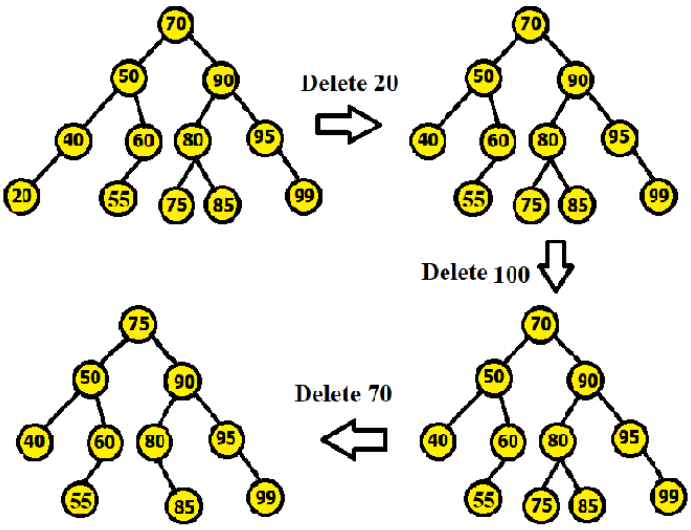
6.4.3 Searching for an element

```
function search(root, item):
    if root.val = item:
        return root
    else if root.val < item:
        return search(root.left, item)
    else:
        return search(root.right, item)
```

6.4.4 Removing a Node

```
function deleteNode(root, key)
  if root is null
    return null
  if key < root.data
    root.left = deleteNode(root.left, key)
  else if key > root.data
    root.right = deleteNode(root.right, key)
  else
    if root has no children
      delete root
      return null
    else if root has one child
      temp = root.left or root.right
      delete root
      return temp
    else
      succ = findMin(root.right)
      root.data = succ.data
      root.right = deleteNode(root.right, succ.data)
  return root

function findMin(node)
  while node.left is not null
    node = node.left
  return node
```

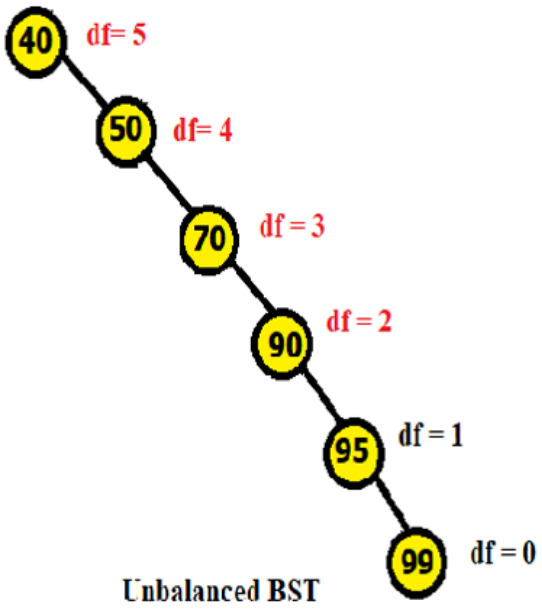


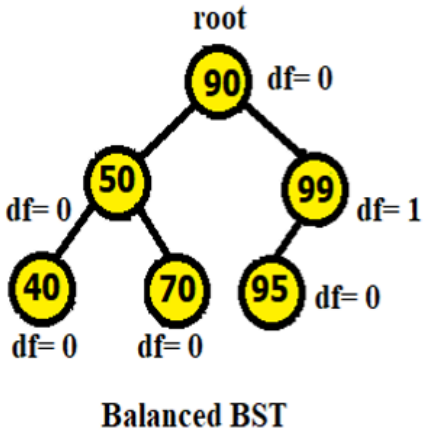
6.4.5 Balancing BST

```
function balanceBST(root)
  if root is null
    return null
  create an empty array arr
  storeInorder(root, arr) // store the nodes in sorted order
  return sortedArrayToBST(arr, 0, arr.length - 1) // build a balanced BST from the array

function storeInorder(node, arr)
  if node is null
    return
  storeInorder(node.left, arr)
  arr.append(node)
  storeInorder(node.right, arr)

function sortedArrayToBST(arr, start, end)
  if start > end
    return null
  mid = (start + end) / 2
  root = arr[mid]
  root.left = sortedArrayToBST(arr, start, mid - 1)
  root.right = sortedArrayToBST(arr, mid + 1, end)
  return root
```



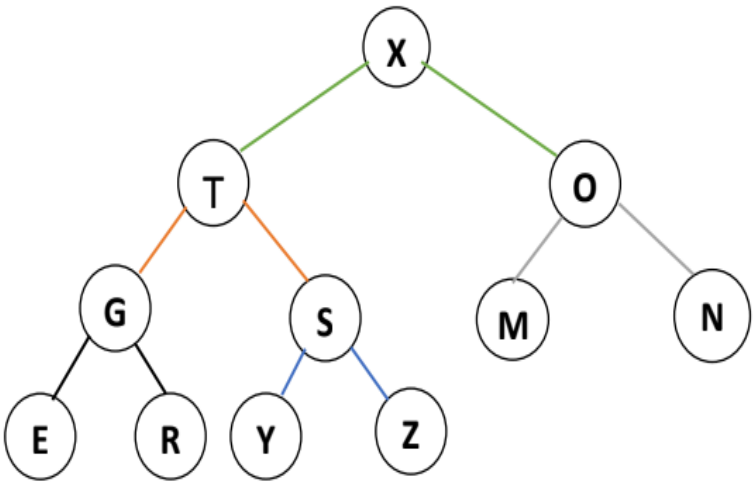


6.5 Heap

Heap is an ADT for storing values. A heap is expressed as a special binary tree pictorially and as its underlying data structure it uses an array. The tree gives heap an advantage to manipulate using a pen and paper quite easily which we will see as we progress. Let me break down the "special tree" as mentioned. A heap has to be a complete binary tree and it must satisfy the heap property.

6.6 Heap Property

The value of the parent must be greater than or equal to the values of the children. (Max heap). or the value of the parent must be smaller than or equal to the values of the children. (Min heap). There are two types of heaps. Max heap is mostly used. A heap can be either a max heap or a min heap but can't be both at the same time.



The above tree is a heap. Below is the array representation of the above heap. Please note that the tree is used for efficient tracing. While programming the data structure is a simple array. Below is the array representation of the above heap. We start the index from 1.

1	2	3	4	5	6	7	8	9	10	11	12
X	T	O	G	S	M	N	E	R	Y	Z	C

For i^{th} NODE, its parent NODE is i/2.
For i^{th} NODE, its children are 2i and 2i+1.
Note: 2i is the LEFT child and 2i+1 is the RIGHT child.

Benefit of using ARRAY for Heap rather than Linked List

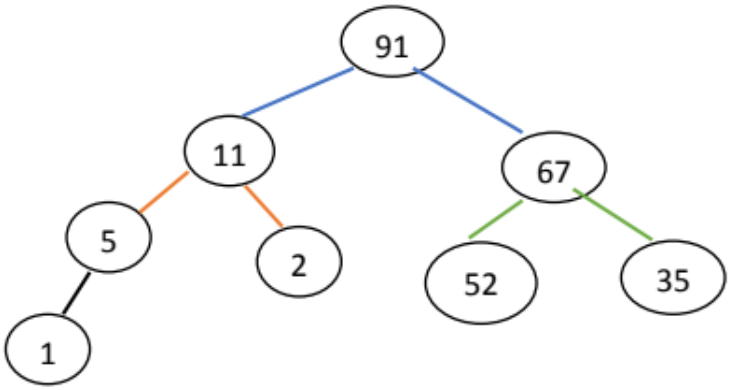
ARRAYS give you random access to its elements by indices. You can just pick any element from the ARRAY by just calling the corresponding index. Finding a parent and its children is trivial. Linked List is sequential. This means you need to keep visiting elements in the linked list unless you find the element you are looking for. Linked List does not allow random access

as ARRAY does. On the other hand each linked list must have three (3) references to traverse the whole Tree (Parent, left, Right).

6.7 Operations on Heap

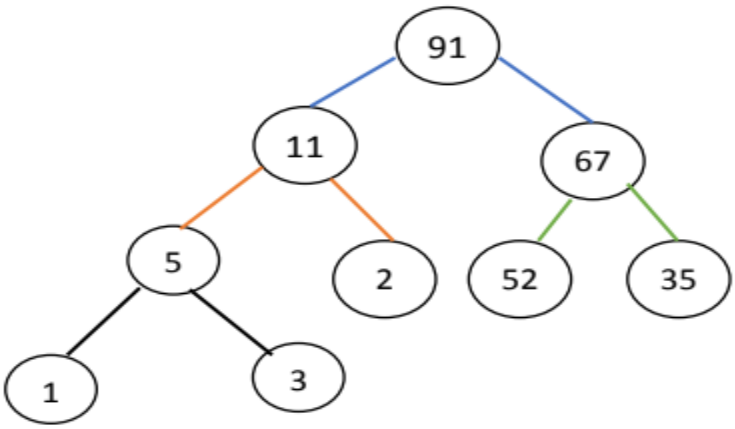
6.7.1 Insert

Inserts an element at the bottom of the Heap. Then we must make sure that the Heap property remains unchanged. When inserting an element in the Heap, we start from the left available position to the right.



Consider the above Heap. If we want to insert an element 3, we start left to right at the bottom level. Therefore 3 will be added as a child of 5.

Then the new Heap will look like this:

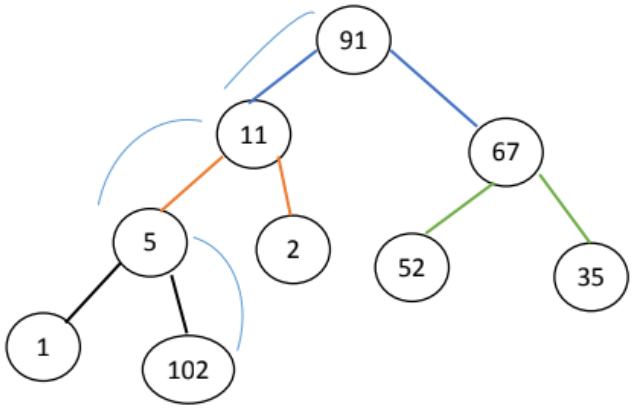


Look carefully five (5) is 3's parent and it is larger. Hence Heap property is kept intact.

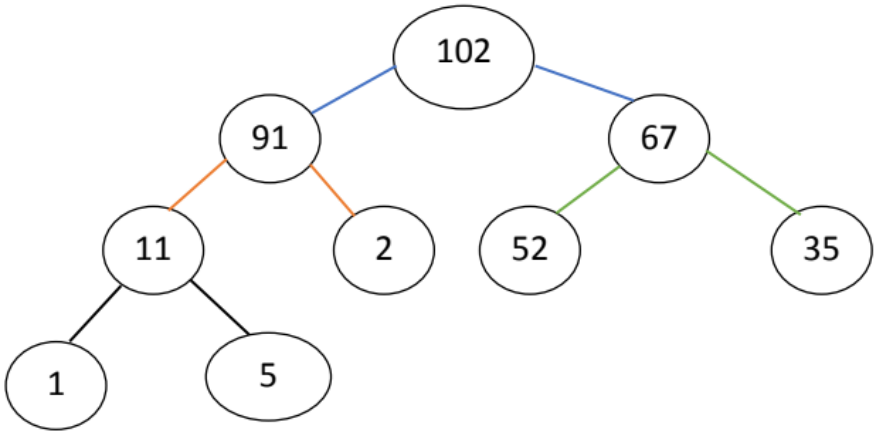
What if we want to insert 102 instead of 3?

Let's say we want to insert 102 at the existing Heap. 102 will be added as a child of 5. Now is the Heap property hold intact? Therefore we need to put 102 in its correct position. How we going to do it? The methodology is called HeapIncreaseKey () or swim().

HeapIncreaseKey ()/swim(): Let the new NODE be 'n' (in this case it is the node that contains 102). Check 'n' with its parent. If the parent is smaller (n > parent) than the node 'n', replace 'n' with the parent. Continue this process until n is its correct position.



After the swim() operation the Heap will look like this:



Time Complexity: Best case O(1) when a key is inserted in the correct position at the first go. Worst case is when the newest node needs to climb up to the root. We have learnt that the distance from the leaf node to the root is lg(n) (height of the tree). Hence this is the worst case complexity. O(1) [insertion] + O(lgn) [swim]

Thus insert is a combination of two functions: insert() and swim().

Pseudocode:

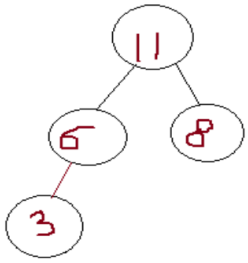
```
insert (H, key){
```

```
        size(H) = size(H) + 1;
        H[size] = key;
        swim (H, size);
}
```

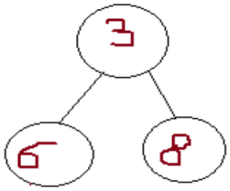
```
swim(H, index){
        if (index < = 1){
                return;
        }else{
                parent = H[index/2];
                if (parent > H[index]){
                        return;
                }else{
                        exchange parent with H[index]
                        swim(H, parent);
                }
}
```

6.7.2 Delete

In heap you cannot just cannot randomly delete an item. Deletion is done by replacing the root with the last element. The Heap property will be broken 100%. Small value will be at the top (root) of the Heap. Therefore we must put it in a right place which is definitely somewhere down the Tree.



Replace the root (11) with the last node (3)



The root is now smaller than its children. So replace 3 with the children with the greater key

This process of putting a node in its correct place by traveling downward is called sink() or MaxHeapify(). The time complexity of sink is lgn as it might have to sink down to the end of the tree.

Delete is a combination of delete() and sink() hence the worst time complexity is lgn too.

Pseudocode:

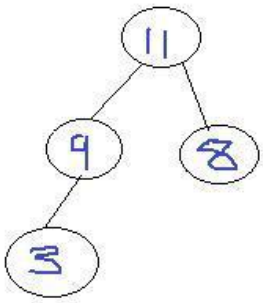
```
delete(H){
        if (size(H)==0){
                return;
        }else{
                exchange H[1] with H[size]
                size --;
                maxHeapify(H, 1)
            }
}
```

```
maxHeapify(H, index){
        if (size(H) ==0){
                return;
        }else{
                left = 2*index;
                right=2*index+1;
                if (left <= size && right<=size){
                        exchange H[1] with Max (H[left], H[right]);
                        maxHeapify(Max (left, right));
                }else{
                        if (left<= size && right>size){
                        exchange H[1] with (H[left]);
                    }

            }
}
```

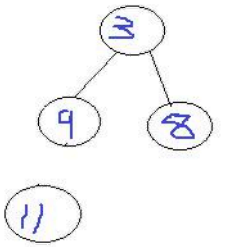
6.8 Heap Sort

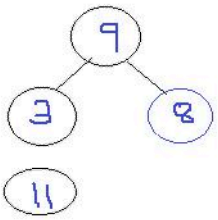
Delete all the nodes of the heap.



Steps:

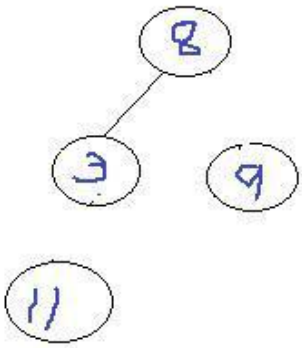
1. Replace the root with the last node, which is basically delete and then sink.





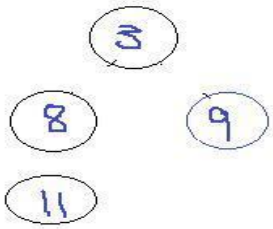
After heapify down

2. Continue deletion until no node is left. Now replace 9 with 8, which is basically delete again.



Heapy down not required because the root is bigger than child

3. Now replace 8 with 3.



After heapify down

This is heap sort. Time complexity O(nlgn). Heap sort is in place, that is, no extra array is required and not stable.

Pseudocode:

```
for all nodes i = 1 to n{
        delete (H);
}
```

Build Max Heap: You are given an arbitrary array and you have been asked to built it a heap. This will take O(nlgn).

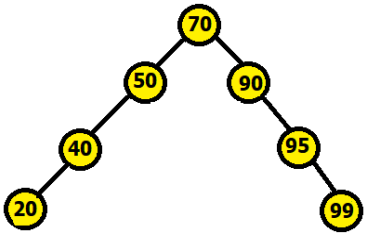
Pseudocode:

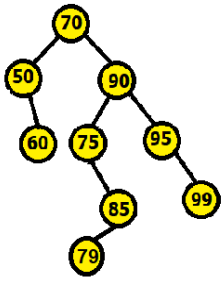
```
for all nodes i = 1 to n{

        swim (H, i);
}
```

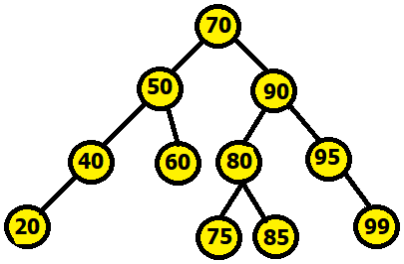
Exercises

6.1: Convert the following unbalanced BSTs into balanced BSTs. Show simulation.

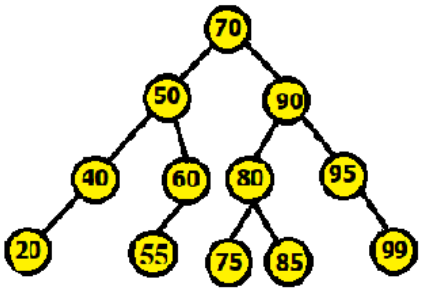




6.2: Insert keys 65, 105, 69 into the following BST and show the steps. Show simulation and code.

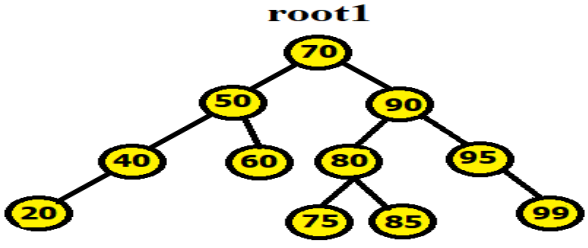


6.3: Delete keys 20, 95, 50, 70, 75 into the following BST and show the steps. Show simulation and code..



6.4: How can you print the contents of a tree in descending order with and without using stack? Solve using code.

6.5: Write a python program that takes the root of a tree and finds its inorder successor and predecessor.



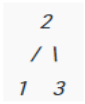
Output: In-order Successor: 75
In-order Predecessor: 60

6.6: Given a sorted array, write a function that creates a Balanced Binary Search Tree using array elements. Follow the steps mentioned below to implement the approach:

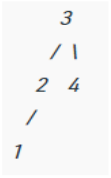
1. Set The middle element of the array as root.
2. Recursively do the same for the left half and right half.
3. Get the middle of the left half and make it the left child of the root created in step 1.
4. Get the middle of the right half and make it the right child of the root created in step 1.
5. Print the preorder of the tree.

Given Array#1: [1, 2, 3]
Output: Pre-order of created BST: 2 1 3

BST#1



BST#2

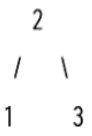


Given Array#2: [1, 2, 3,4]
Output: Pre-order of created BST: 3 2 1 4

6.7: Given the root of a binary tree, check whether it is a BST or not. A BST is defined as follows:

A. The left subtree of a node contains only nodes with keys less than the node's key.
B. The right subtree of a node contains only nodes with keys equal or greater than the node's key.
C. Both the left and right subtrees must also be binary search trees.

Input:



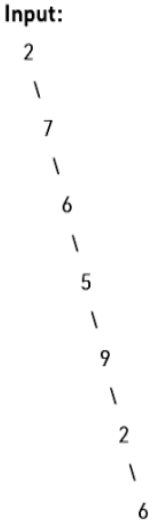
Output: 1

Explanation:

The left subtree of root node contains node with key lesser than the root nodes key and the right subtree of root node contains node with key greater than the root nodes key.

Hence, the tree is a BST.

Input:



Output: 0

Explanation:

Since the node with value 7 has right subtree nodes with keys less than 7, this is not a BST.

6.8: Given a sorted array. Write a function that creates a Balanced Binary Search Tree using array elements. Height balanced BST means a binary tree in which the depth of the left subtree and the right subtree of every node never differ by more than 1

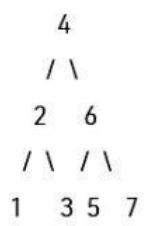
Input: nums = {1,2,3,4,5,6,7}

Ouput: {4,2,1,3,6,5,7}

Explanation:

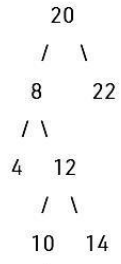
The preorder traversal of the following

BST formed is {4,2,1,3,6,5,7} :



6.9: Given a BST, and a reference to a Node x in the BST. Find the Inorder Successor of the given node in the BST.

Input:



K(data of x) = 8

Output: 10

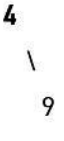
Explanation:

Inorder traversal: 4 8 10 12 14 20 22

Hence, successor of 8 is 10.

6.10: Given a Binary search tree, your task is to complete the function which will return the Kth largest element without doing any modification in the Binary Search Tree.

Input:



k = 2

Output: 4

Appendix A - Recursion

A.1 Introduction

Recursive function is no different than a normal function. The motivation to use recursive functions vs non-recursive is that the recursive solutions are usually easier to read and comprehend. Certain applications, like **tree search, directory traversing** etc. are very well suited for recursion. The drawbacks are that you may need a little time to learn how to use it and you may need a little longer to debug errors. It takes more processing time and more memory. But there can be cases when recursion is the best way.

For example if you need to get the full tree of a directory and store it somewhere. You can write loops but it will be very complicated. And it will be very simple if you use recursion. You'll only get files of root directory, store them and call the same function for each of the subdirectories in root.

Recursion is a way of thinking about problems, and a method for solving problems. The basic idea behind recursion is the following: it's a method that solves a problem by solving smaller (in size) versions of the same problem by breaking it down into smaller subproblems. Recursion is very closely related to mathematical induction.
We'll start with recursive definitions, which will lay the groundwork for recursive programming. We'll then look at a few prototypical examples of using recursion to solve problems. We'll finish by looking at problems and issues with recursion.

A.2 Recursive Definitions

We'll start thinking recursively with a few recursive definitions:
1. Factorial
2. Fibonacci numbers

A.2.1 Factorial:

The factorial of a non-negative integer n is defined to be the product of all positive integers less than or equal to n. For example, 5! = 5 * 4 * 3 * 2 * 1 = 120.

```
1! = 1
2! = 2 * 1
3! = 3 * 2 * 1
4! = 4 * 3 * 2 * 1
5! = 5 * 4 * 3 * 2 * 1
…..
….. And so on
```

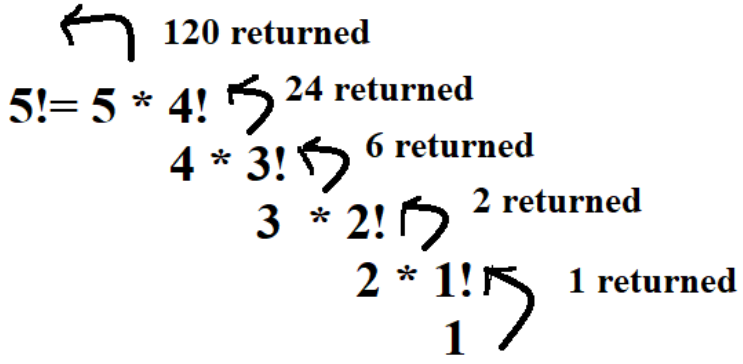
We can easily evaluate n! for any valid value of n by multiplying the values iteratively. However, there is a much more interesting recursive definition quite easily seen from the factorial expressions: 5! is nothing other than 5 * 4!. If we know 4!, we can trivially compute 5!.
4! on the other hand is 4 * 3!, and so on until we have n! = n * (n - 1)!, with 1! = 1 as the base case. The mathematicians have however added the special case of 0! = 1 to make it easier (yes, it does, believe it or not). For the purposes of this discussion, we'll use both 0! = 1 and 1! = 1 as the two base cases.

$$n! = \begin{cases} 1 & \text{if } n=1 \\ 0 & \text{if } n=0 \\ n^* (n-1)! & \text{if } n>1 \end{cases}$$

Now we can expand 5! recursively, stopping at the base condition.

```
5!= 5 * 4!
        4 * 3!
            3  * 2!
                2 * 1!
                    1
```

The recursion tree for 5! shows the values as well.



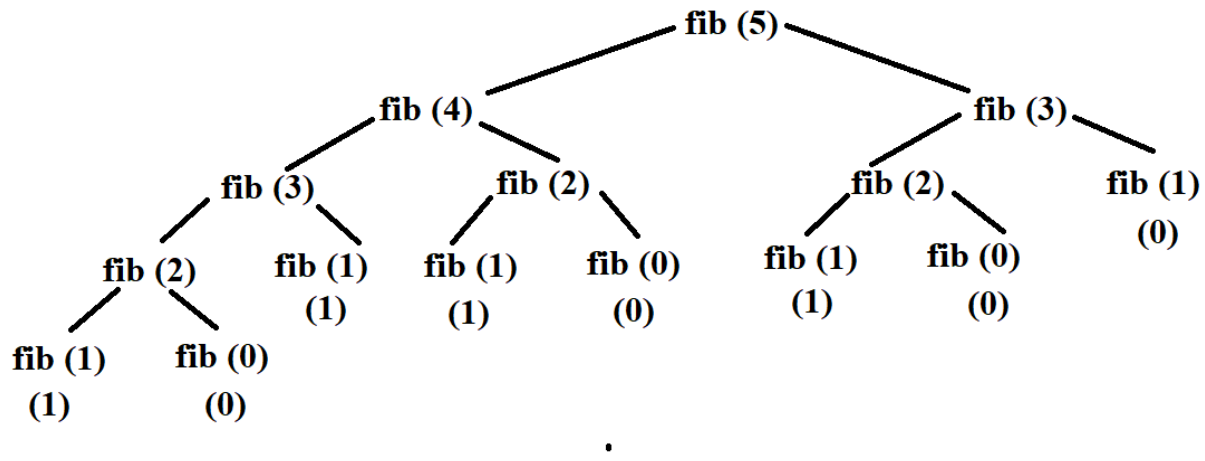
A.2.2 Fibonacci numbers:

The Fibonacci numbers are ‹0, 1, 1, 2, 3, 5, 8, 13, ...› (some define it without the leading 0, which is ok too). If we pay closer attention, we see that each number, except for the first two, is nothing but the sum of the previous two numbers. We can easily compute this iteratively, but let's stick with the recursive method. We already have the recursive part of the recursive definition, so all we need is the non-recursive part or the base case. The mathematicians have decided that the first two numbers in the sequence are 0 and 1, which give us the base cases (notice the two base cases).

Now we can write the recursive definition for any Fibonacci number n >= 0.

$$\text{fib}(n) = \begin{cases} 1 & \text{if } n=1 \\ 0 & \text{if } n=0 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n>1 \end{cases}$$

We can now compute fib(5) using this definition.



Before moving on, you should note how many times we're computing Fibonacci of 3 (fib(3) above), and Fibonacci of 2 (fib(2) above) and so on. This redundancy in computation leads to gross inefficiency, but something we can easily address is Memoization, which is the later topic we study.

A.3 Recursive programming

A recursive function is one that calls itself, since it needs to solve the same problem, but on a smaller sized input. In essence, a recursive function is a function that is defined in terms of itself.

Let's start with computing the factorial. We already have the recursive definition, so the question is how do we convert that to a recursive method that actually computes a factorial of a given non-negative integer. This is an example of functional recursion.

```
function factorial(n)
  if n == 0 or n == 1
    return 1
  else
    return n * factorial(n - 1)
  end if
end function
```

Once you have formulated the recursive definition, the rest is usually quite trivial. This is indeed one of the greatest advantages of recursive programming.

Let's now look at an example of structural recursion, one that uses a recursive data structure. We want to compute the sum of the list of numbers. Linked list = 3-> 8-> 2-> 1-> 13-> 4->None.

The "easiest" way to find the sum of numbers of a linked list iteratively, as shown below.

```
function sumList(head)
  sum = 0 // initialize the sum variable
  ptr = head // initialize a pointer to the head of the list
  while ptr is not null // loop until the end of the list
    sum = sum + ptr.data // add the current node's data to the sum
    ptr = ptr.next // move the pointer to the next node
  end while
  return sum // return the sum of the list
end function
```

But a linked list is a recursive data structure. Hence, by thinking recursively, we note the following:

1. The sum of the numbers in a list is nothing but the sum of the first number plus the sum of the rest of the list. The problem of summing the rest of the list is the same problem as the original,

except that it's smaller by one element! Ah, recursion at play.
2. The shortest list has a single number, which has the sum equal to the number itself. Now we have a base case as well. Note that if we allow our list of numbers to be empty, then the base case will need to be adjusted as well: the sum of an empty list is simply 0.
Note the key difference between the iterative and recursive approaches: for each number in the list vs the rest of the list.

Now we can write the recursive definition of this problem:

$$sum = \begin{cases} n.elem & \text{When n is the only node} \\ 0 & \text{When the linkedlist is empty} \\ n.elem + sum(n.next) & \text{Otherwise} \end{cases}$$

and using recursive definition, we can write the recursive method as shown below:

```
function sumList(head)
  if head is null
    return 0
  else
    return head.data + sumList(head.next)
```

Examples:
Now we will look at the following examples which can be solved by recursive programming:

1. Length of a String
2. Length of a linked list
3. Sequential search in a sequence
4. Binary search in a sorted array
5. Finding the maximum in a sequence (linear version)
6. Finding the maximum in an array (binary version)
7. Selection sort
8. Insertion sort
9. Exponentiation – a^n

A.3.1 Length of a linked list:

A linked list is also a recursive structure: a linked list is either empty, or a node followed by the rest of the list.

We can also compute the length of a list recursively as follows: the length of a linked list is 1 longer than the rest of the list! The empty list has a length of 0, which is our base case.

Recursive definition is given below:

$$\text{length(list\_a)} = \begin{cases} 0 & \text{When the node is Null} \\ 1 + \text{length(list\_a.next)} & \text{Otherwise} \end{cases}$$

Recursive process is shown below:

```
function countList(head)
  if head is null
    return 0
  else
    return 1 + countList(head.next)
```

A.3.2 Sequential search in a sequence:

How would you find something in a linked list? Well, look at the first node and check if the key is in that node. If so, done. Otherwise, check the rest of the linked list for the given key. If you search an empty list for any key, the answer is false, so that's our base case.

This is almost exactly the same, at least in form, as finding the length of a linked list, and also an example of structural recursion.

Recursive definition is given below:

$$\text{contains (n, k)} = \begin{cases} \text{false} & \text{if n in null} \\ \text{true} & \text{if n.elem == k} \\ \text{contains(n.next, k)} & \text{otherwise} \end{cases}$$

Recursive process is shown below:

```
function sequentialSearch(head, key)
  if head is null
    return false
  else if head.data == key
    return true
  else
    return sequentialSearch(head.next, key)
```

What if the sequence is an array? How do we deal with the rest of the array part then? We can handle it like this:

We can maintain a left index, along with a reference to the array, that is used to indicate where the beginning of the array is.

Initially, left = 0, meaning that the array begins at the expected index of 0. Eventually, a value of left = len(array) - 1 means that the rest of the array is simply the last element, and then left = len(array) means that it's an 0-sized array.

Recursive definition:

$$\text{contains}(a, l, k) = \begin{cases} -1 & \text{if length of a is} == l \\ l & \text{if } a[l] == k \\ \text{contains}(a, l+1, k) & \text{Otherwise} \end{cases}$$

Recursive method:

```
function sequentialSearch(array, index, key)
  if index >= array.length
    return -1
  else if array[index] == key
    return index
  else
    return sequentialSearch(array, index + 1, key)
```

We start the search with **contains(arr, 0, key)**, and then at each step, the rest of the array is given by advancing the left boundary(index).

Instead of just a yes/no answer, what if we wanted the position of the key in the array? We can simply **return left** instead of true as the position if found, or use a sentinel -1 instead of false if not.

A.3.3 Binary search in a sorted array:

Given the abysmal performance of sequential search, we obviously want to use binary search whenever possible. Of course, the pre-conditions must be met first:

1. The sequence must support random access (an array that is)
2. The data must be sorted

Now we can write the recursive definition:.

$$\text{binarySearch(array, key, left, right)} = \begin{cases} \text{false} & \text{if } l > r \\ \text{true} & \text{if key} = \text{array[mid]} \\ \text{binarySearch(array, key, left, mid-1)} & \text{if key} < \text{array[mid]} \\ \text{binarySearch(array, key, mid+1, right)} & \text{if key} > \text{array[mid]} \end{cases}$$

Recursive process:

```
function binarySearch(array, key, left, right)
  if left > right
    return -1
  else
    mid = (left + right) / 2
    if array[mid] == key
      return mid
    else if array[mid] > key
      return binarySearch(array, key, left, mid - 1)
    else
      return binarySearch(array, key, mid + 1, right)
```

We start the search with **binarySearch(arr, key, 0, len(arr) - 1)**, and then at each step, the rest of the array is given by one half of the array – left or right, depending on the comparison of the key with the middle element.
Instead of just a yes/no answer, what if we wanted the position of the key in the array? We can simply **return mid** as the position instead of true if found, or use a sentinel -1 instead of false if not.

A.3.4 Finding the maximum in a sequence (linear version):

Given a sequence of keys, our task is to find the maximum key in the sequence. This is of course trivially done iteratively (for a non-empty sequence): take the 1st one as maximum, and

then iterate from the 2nd to the end, exchanging the current with the maximum if the current is larger than the maximum.
Formulating this recursively: the maximum key in a sequence is the larger of the following two:
1. the 1st key in the sequence
2. the maximum key in the rest of the sequence

Once we have (recursively) computed the maximum key in the rest of the sequence, we just have to compare the 1st key with that, and we have our answer! The base case is also trivial (for a non-empty sequence): the maximum key in a single-element sequence is the element itself.
Since the rest of the sequence does not need random access, we can easily do this for a linked list or an array. Let's write it for a linked list first.

```
function findMax(head)
  if head is null
    return -infinity
  else
    return max(head.data, findMax(head.next))
```

We start to find the maximum with **findMax(head)** (where head is the reference to the first node of the list), and then at each step, the rest of the array is given by advancing the head reference.

What if the sequence is an array? Well, then we use the same technique we've used before — use a left (and optionally right) boundary to window into the array.

Recursive process:

```
function findMax(array, n)
  if n == 1
    return array[0]
  else
    return max(array[n-1], findMax(array, n-1))
```

We start to find the maximum with **findMax(arr, 0)**, and then at each step, the rest of the array is given by advancing the left boundary(index).

A.3.5 Finding the maximum in an array (binary version):

If our sequence is an array, we can also find the maximum by formulating the following recursive definition: the maximum key in an array is the larger of the following two:

1. the maximum key in the left half of the array
2. the maximum key in the right half of the array

```
function findMax(array, left, right)
  if left == right
    return array[left]
  else
    mid = (left + right) / 2
    leftMax = findMax(array, left, mid)
    rightMax = findMax(array, mid + 1, right)
    return max(leftMax, rightMax)
```

We start to find the maximum with **findMax(arr, 0, len(array)-1)**, and then at each step, the array is divided into two halves.

A.4 Advance Recursion Part 1

A.4.1 Selection sort:

How about sorting a sequence recursively? Since it does not require random access, we'll look at recursive versions for both linked lists and arrays.
The basic idea behind selection sort is the following: put the 1st minimum in the 1st position, the 2nd minimum in the 2nd position, the 3rd minimum in the 3rd position, and so on until each key is placed in its position according to its rank. To come up with a recursive formulation, the following observation is the key:

Once the 1st minimum in the 1st position, it will never change its position. Now all we have to do is to sort the rest of the sequence (from 2nd position onwards), and we'll have a sorted sequence.

Now we can write the recursive definition for a linked list, and see the pseudocode.

```
function selectionSort(head)
  if head is null or head.next is null
    return head
  else
    min = findMin(head) // find the minimum node in the list
    swap(head, min) // swap the head node with the minimum node
    head.next = selectionSort(head.next) // recursively sort the rest of the list
    return head
```

Here, be careful about one thing: the swap method means we are not exchanging the nodes rather than we are exchanging the element of nodes.

```
Function swap(a, b)
  temp = a.element // store a.element in temp
  a.element = b.element // assign b.element to a.element
  b.element = temp // assign temp to b.element
end Function
```

We sort a list headed by head reference by calling **selectSort(head)**. Note that we're not finding the minimum key, but rather the node that contains the minimum key since we need to exchange the left key with the minimum one. We can write that iterative of course, but a recursive one is simply more fun. This is of course almost identical to finding the maximum in a sequence, with two differences: we find the minimum, and we return the node that contains the minimum, not the actual minimum key.

Recursive process of findMin:

```
function findMin(head)
  if head is null
    return
  else
    return min(head.data, findMin(head.next))
```

If the sequence is an array, then we have to use the left (and optionally right) boundary to window into the array.

```
function selectionSort(array,  i = 0)
  if i == array length - 1 // base case: the array is sorted
    return
  else
    minIndex = i // assume the first element is the minimum
    for j = i + 1 to  array length - 1 // loop through the rest of the array
      if array[j] < array[minIndex] // find the actual minimum
        minIndex = j

    swap(array[i], array[minIndex]) // swap the minimum with the first element
    selectionSort(array,  i + 1) // recursively sort the remaining array
end function
```

Here, be careful about one thing: the swap method means we are not exchanging the indices rather than we are exchanging the element of these indices in the array.

```
Function swap(a, b)
  temp = a // store a in temp
  a = b // assign b to a
```

```
  b = temp // assign temp to b
end function
```

We sort an array(arr) by calling **selectSort(arr, 0)**.

A.4.2 Insertion Sort:

Insertion works by inserting each new key in a sorted array so that it is placed in its rightful position, which now extends the sorted array by the new key. In the beginning, there is a single key in the array, which by definition is sorted. Then the second key arrives, which is then inserted into the already sorted array (of one key at this point), and now the sorted array has two keys. Then the third key arrives, which is inserted into the already sorted array, creating a sorted array of 3 keys. And so on. Iteratively, it's a fairly simple operation. The question is how can we formulate this recursively. The following observation is the key to this recursive formulation:
Given an array of n keys, sort the first n-1 keys and then insert the nth key in the sorted array such that all n keys are now sorted.
Note how the recursive part comes first, and then the nth key is inserted (iteratively) into the sorted array.
Unlike recursive selection sort, we're going from right to left in the recursive version of insertion sort. Note that we don't need random access, but need to be able to iterate in both directions (reverse direction to insert the new key in the sorted partition). So, if we're sorting a linked list using the insertion sort algorithm, the list must be doubly-linked.

Recursive process of insertion sort:

```
function insertionSort(array, n, i = 1)
  if i == n // base case: the array is sorted
    return
  else
    key = array[i] // store the current element
    j = i - 1 // start from the previous element
    while j >= 0 and array[j] > key // loop through the sorted subarray
      array[j + 1] = array[j] // shift the larger elements to the right
      j = j - 1 // move to the next element

    array[j + 1] = key // insert the current element in the correct position
    insertionSort(array, n, i + 1) // recursively sort the remaining array

end function
```

We sort an array(arr) by calling **insertionSort(arr, len(arr)- 1, 1)**.

A.4.3 Exponentiation – a^n:

This is another example of functional recursion. To compute a^n, we can iteratively multiply a n times, and that's that. Thinking recursively, a^n = a * a^{n-1}, and a^{n-1} = a * a^{n-2}, and so on. The recursion stops when the exponent n = 0, since by definition a^0 = 1.

Recursive definition:

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a * a^{n-1} & \text{if } n > 0 \end{cases}$$

Recursive approach:

```
function power(base, exponent)
  if exponent == 0 // base case: any number raised to 0 is 1
    return 1
  else
    return base * power(base, exponent - 1) // recursive case: multiply the base by itself
exponent times
end function
```

As it turns out, there is actually a much more efficient recursive formulation for the exponentiation of a number. We start by noting that $2^8 = 2^4 * 2^4$, and that $2^7 = 2^3 * 2^3 * 2$. We can generalize that with the following recursive definition, and its implementation.

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a^{n/2} * a^{n/2} & \text{if n is even} \\ a^{(n-1)/2} * a^{(n-1)/2} * a & \text{if n is odd} \end{cases}$$

Recursive approach:

```
function power(base, exponent)
  if exponent == 0 // base case: any number raised to 0 is 1
    return 1
```

```
  else
    If exponent is even
      return power(base, exponent/2)*power(base, exponent/2)
    else
      return power(base, (exponent-1)/2)*power(base, (exponent-1)/2) * a
```

But why would we care about this formulation over the more familiar one? If we solve for the running time, both solutions take the same time, so what is the benefit of this approach? Notice how we're computing the following expressions twice:
1. exp(a, n/2)
2. exp(a, (n - 1)/2)

Why not compute it once, and then use the result twice (or as many times as needed)? We can, and as we will find out, that will give us a huge boost when we compute the running time of this algorithm. Here is the modified version.

```
function power(base, exponent)
  if exponent == 0 // base case: any number raised to 0 is 1
    return 1
  else
    If exponent is even
      temp = power(base, exponent/2)
      return temp * temp
    else
      temp = power(base, (exponent-1)/2)
      return temp * temp * base
```

All we are doing is removing the redundancy in computations by saving the intermediate results in temporary variables. This is a simple case of a technique known as **Memoization**, which is the next topic we study. Remember that we have already seen such redundancy in recursive computation — when computing the Fibonacci numbers.

A.5 Issues/problems to watch out for

1. **Inefficient recursion:**
The recursive solution for Fibonacci numbers outlined in these notes shows massive redundancy, leading to very inefficient computation. The 1st recursive solution for exponentiation also shows how redundancy shows up in recursive programs. There are ways to avoid computing the same value more than once by caching the intermediate results, either using Memoization (a top-down technique — see next topic), or Dynamic Programming (a bottom-up technique — survive this semester to enjoy it in the next one).

2. **Space for activation frames:**

Each recursive method call requires that it's activation record be put on the system call stack. As the depth of recursion gets larger and larger, it puts pressure on the system stack, and the stack may potentially run out of space.

3. **Infinite recursion:**
Ever forgot a base case? Or miss one of the base cases? You end up with infinite recursion, since there is nothing stopping your recursion! Whenever you see a Stack Overflow error, check your recursion!

A.6 Advanced Recursion Part 2: Optimizing Recursive Program Memoization

A.6.1 Introduction:

To develop a recursive algorithm or solution, we first have to define the problem (and the recursive definition, often the hard part), and then implement it (often the easy part). This is called the **top-down** solution, since the recursion opens up from the top until it reaches the base case(s) at the bottom.
Once we have a recursive algorithm, the next step is to see if there are **redundancies** in the computation— that is, if the same values are being computed multiple times. If so, we can benefit from **memoizing** the recursion. And in that case, we can create a memoized version and see what savings we get in terms of the running time.
Recursion has certain overhead costs that may be minimized by transforming the memoized recursion into an iterative solution. And, finally, we see if there are further improvements that we can make to improve the time and space complexity.

The steps are as follows:
1. write down the recursion,
2. implement the recursive solution,
3. memoize it,
4. transform into an iterative solution, and finally
5. make further improvements.

A.6.2 Example using the Fibonacci sequence:

To see this in action, let's take Fibonacci numbers as an example. Fortunately for us, the mathematicians have already defined the problem for us – the Fibonacci numbers are ‹ 0, 1, 1, 2, 3, 5, 8, 13,... › (some define it without the leading 0, which is ok too). Each number, except for the first two, is nothing but the sum of the previous two numbers. The first two are by definition 0 and 1. These two facts give us the recursive definition to compute the n^{th} Fibonacci number for some n >= 0.

Let's go through the 5 steps below.

Step 1: Write or formulate the recursive definition of the nth Fibonacci number (defined only for n>=0)

$$fib(n) = \begin{cases} n & \text{if } n<2 \\ fib(n-1) + fib(n-2) & \text{if } n>=2 \end{cases}$$

Step 2: Write the recursive implementation. This usually follows directly from the recursive definition

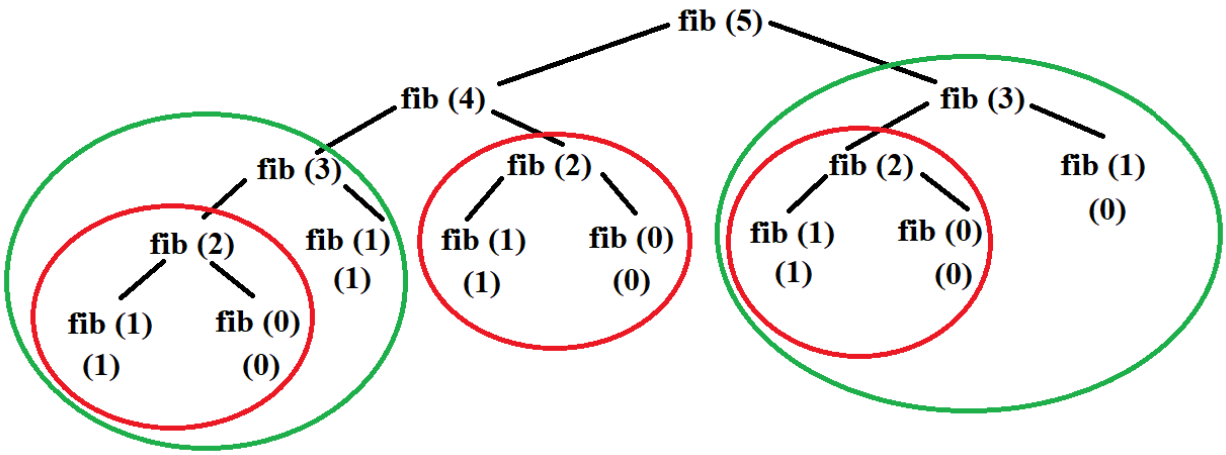
```
function fib(n)
  if n == 0 or n == 1 // base case: the first two Fibonacci numbers are 0 and 1
    return n
  else // recursive case: use the Fibonacci formula
    return fib(n - 1) + fib(n - 2)
end function
```

Step 3: Memoize the recursion

Would this recursion benefit from memoization? Well, let's see by "unrolling" the recursion **fib(5)** a few levels:

Now you should notice something very interesting — we're computing the same fibonacci number quite a few times. We're computing fib(2) 3 times and fib(3) 2 times. Is there any reason why we couldn't simply save the result after computing it the first time, and re-using it each time it's needed afterward?

This is the primary motivation for **memoization** – to avoid re-computing overlapping subproblems. In this example, fib(2) and fib(3) are overlapping subproblems that occur independently in different contexts.



Memoization certainly looks like a good candidate for this particular recursion, so we'll go ahead and memoize it. Of course, the first question is how we save the results of these overlapping subproblems that we want to reuse later on.

The basic idea is very simple — before computing the i^{th} fibonacci number, first check if that has already been solved; if so, just look up the answer and return; if not, compute it and save it before returning the value. We can modify our fib method accordingly (using some pseudocode for now). Since fibonacci is a function of 1 integer parameter, the easiest is to use a 1-dimensional array or table with a capacity of **n + 1** (we need to store fib(0) ... fib(n), which requires n + 1 slots) to store the intermediate results.

What is the cost of memoizing a recursion? It's the space needed to store the intermediate results. Unless there are overlapping subproblems, memoizing a recursion will not buy you anything at all, and in fact, cost you more space for nothing!
Remember this — **memoization trades space for time.**

Let's try out our first memoized version. (M_fib below stands for **"memoized fibonacci"**).

```
function fib(n)
  // assume that we have a “global” array (also called a table) with n+1 capacity


  // check if the nth Fibonacci number is already computed
  if memo[n] is not null
    return memo[n]
  else
    // base case: the first two Fibonacci numbers are 0 and 1
    if n == 0 or n == 1
      memo[n] = n
    else
      // recursive case: use the Fibonacci formula and store the result
```

```
    memo[n] = fib(n - 1) + fib(n - 2)

  return memo[n]

end function
```

Think about why we used a global array instead of declaring arrays inside the function!

This is all that we need to avoid redundant computations of overlapping subproblems. The first time fib(3) is called, it will compute the value and save the answer in memo[3], and then subsequent calls would simply return memo[3] without doing any work at all!

There are a few details we have left out at this point:
1. Where would we create this "table" to store the results?
2. How can we initialize each element of memo to indicate that the value has not been computed/saved yet?(Note the **"is empty"** in the code above).

Let's take these one at a time.

1. The "fib" method is a function of a single parameter — n, so if we wanted to save the intermediate results, all we need is an array that goes from 0 ... n (i.e., of n + 1 capacity). Since the local variables within a method are created afresh each time the method is called, F cannot be a local array. We can either use an instance variable within an object, or create an array in the caller of fib(n), and then pass the array to fib (in which case we will have to modify fib to have another parameter).
2. We need to use a sentinel which will indicate that the value has not been computed. Since it's an array of integers, we can't use null (which is the sentinel used to indicate the absence of an object).
However, we know that the n^{th} fibonacci number is a non-negative integer, so we can use any negative number as the sentinel. **Let us choose -1**. So, let's have an array F of n+1 capacity that holds all the values of the intermediate results we need to compute the n^{th} Fibonacci number.
We can have a wrapper method, which creates this array or table, initializes the table and passes it onto M_fib as a parameter.

First, the wrapper method called **fib**, which basically sets up the table for **M_fib**, and calls it on the user's behalf.

```
function fib(n)
   Create an array (also called a table) with n+1 capacity named memo
   Call M_fib(n,memo)


function M_fib(n, memo)

  // check if the nth Fibonacci number is already computed
  if memo[n] is not null
```

```
    return memo[n]
  else
    // base case: the first two Fibonacci numbers are 0 and 1
    if n == 0 or n == 1
      memo[n] = n
    else
      // recursive case: use the Fibonacci formula and store the result
      memo[n] = M_fib(n - 1) + M_fib(n - 2)

    return memo[n]

end function
```

To compute the 5th fibonacci number, we simply call **fib(5)**, which in turn calls **M_fib(5, F)** to compute and return the value.
Now that we have a memoized fibonacci, the next question is to see if we can improve the space overhead of memoization.

Step 4: Convert the recursion to iteration – the bottom-up solution.

To compute the 5th fibonacci number, we wait for 4th and 3rd, which in turn wait for 2nd, and so on until the base cases of n = 0 and n = 1 return the values which move up the recursion stack. Other than the n = 0 and n = 1 base cases, the first value that is actually computed and saved is n = 2, and then n= 3, and then n = 4 and finally n = 5. Then why not simply compute the solutions for
n = 2, 3, 4, 5 by iterating (using the base cases of course), and fill in the table from left to right? This is called the **bottom-up** solution since the recursion tree starts at the bottom (the base cases) and works its way up to the top of the tree (the initial call). The bottom-up technique is more popularly known as **dynamic programming**, a topic that we will spend quite a bit of time on next semester!

```
function fib(n)
  // create an array to store the Fibonacci numbers
  array f[n + 1]

  // initialize the first two Fibonacci numbers
  f[0] = 0
  f[1] = 1

  // loop from the third Fibonacci number to the nth Fibonacci number
  for i = 2 to n
    // use the Fibonacci formula and store the result in the array
    f[i] = f[i - 1] + f[i - 2]
  end for

  // return the nth Fibonacci number
  return f[n]
```

```
end function
```

You should convince yourself that this is indeed a solution to the problem, only using iteration instead of memoized recursion. Also, that it solves each subproblem (e.g., fib(3) and fib(2)) exactly once, and re-uses the saved answer.

This one avoids the overhead of recursion by using iteration, so tends to run much faster.

Can we improve this any further?

Step 5: improving the space-requirement in the bottom-up version

The n^{th} Fibonacci number depends only on the $(n - 1)^{th}$ and $(n - 2)^{th}$ Fibonacci numbers. However, we are storing ALL the intermediate results from 2 ... n - 1 Fibonacci numbers before computing the n^{th} one. What if we simply store the last two? In that case, instead of having an array of n + 1 capacity, we need just two instance variables (or an array with 2 elements). Here's what the answer may look like.

```
function fib(n)
  // initialize the first two Fibonacci numbers
  f0 = 0
  f1 = 1

  // loop from the first Fibonacci number to the nth Fibonacci number
  for i = 0 to n - 1
    // use the Fibonacci formula and store the result in a temporary variable
    temp = f0 + f1
    // update the previous two Fibonacci numbers
    f0 = f1
    f1 = temp
  end for

  // return the nth Fibonacci number
  return f0
end function
```

Exercises

A.1: Given a non-negative int n, return the sum of its digits recursively (no loops). Note that mod (%) by 10 yields the rightmost digit (126 % 10 is 6), while divide (/) by 10 removes the rightmost digit (126 / 10 is 12).

Example:
Input: sumDigits(126)
Output: 9

Input: sumDigits(49)
Output: 13

Input: sumDigits(12)
Output: 3

A.2: We have bunnies standing in a line, numbered 1, 2, ... The odd bunnies (1, 3, ..) have the normal 2 ears. The even bunnies (2, 4, ..) we'll say have 3 ears, because they each have a raised foot. Recursively return the number of "ears" in the bunny line 1, 2, ... n (without loops or multiplication).

Example:
Input: bunnyEars2(0)
Output: 0

Input: bunnyEars2(1)
Output: 2

Input: bunnyEars2(2)
Output: 5

A.3: Given a non-negative int n, return the count of the occurrences of 7 as a digit, so for example 717 yields 2. (no loops). Note that mod (%) by 10 yields the rightmost digit (126 % 10 is 6), while divide (/) by 10 removes the rightmost digit (126 / 10 is 12).

Example:
Input: count7(717)
Output: 2

Input: count7(7)
Output: 1

Input: count7(123)
Output: 0

A.4: Given a string, compute recursively (no loops) the number of lowercase 'x' chars in the string.

Example:

Input: countX("xxhixx")
Output: 4

Input: countX("xhixhix")
Output: 3

Input: countX("hi")
Output: 0

A.5: Given a string, compute recursively (no loops) a new string where all appearances of "pi" have been replaced by "3.14".

Example:
Input: changePi("xpix")
Output: "x3.14x"

Input: changePi("pipi")
Output: "3.143.14"

Input: changePi("pip")
Output: "3.14p"

A.6: Given an array of ints, compute recursively the number of times that the value 11 appears in the array. We'll use the convention of considering only the part of the array that begins at the given index. In this way, a recursive call can pass index+1 to move down the array. The initial call will pass in index as 0.

Example:
Input: array11([1, 2, 11], 0)
Output: 1

Input: array11([11, 11], 0)
Output: 2

Input: array11([1, 2, 3, 4], 0)
Output: 0

A.7: Given a string, compute recursively a new string where identical chars that are adjacent in the original string are separated from each other by a "*".

Example:
Input: pairStar("hello")
Output: "hel*lo"

Input: pairStar("xxyy")
Output: "x*xy*y"

Input: pairStar("aaaa")
Output: "a*a*a*a"

A.8: Count recursively the total number of "abc" and "aba" substrings that appear in the given string.

Example:
Input: countAbc("abc")
Output: 1

Input: countAbc("abcxxabc")
Output: 2

Input: countAbc("abaxxaba")
Output: 2

A.9: Given a string, compute recursively the number of times lowercase "hi" appears in the string, however do not count "hi" that have an 'x' immedately before them.

Example:
Input: countHi2("ahixhi")
Output: 1

Input: countHi2("ahibhi")
Output: 2

Input: countHi2("xhixhi")
Output: 0

A.10: Given a string and a non-empty substring sub, compute recursively the number of times that sub appears in the string, without the sub strings overlapping.

Example:
Input: strCount("catcowcat", "cat")
Output: 2

Input: strCount("catcowcat", "cow")
Output: 1

Input: strCount("catcowcat", "dog")
Output: 0

A.11: We have a number of bunnies and each bunny has two big floppy ears. We want to compute the total number of ears across all the bunnies recursively (without loops or multiplication).

Example:
Input: bunnyEars(0)
Output: 0

Input: bunnyEars(1)
Output: 2

Input: bunnyEars(2)
Output: 4

A.12: We have triangle made of blocks. The topmost row has 1 block, the next row down has 2 blocks, the next row has 3 blocks, and so on. Compute recursively (no loops or multiplication) the total number of blocks in such a triangle with the given number of rows.

Example:
Input: triangle(0)
Output: 0

Input: triangle(1)
Output: 1

Input: triangle(2)
Output: 3

A.13: Given a string, compute recursively a new string where all the 'x' chars have been removed.

Example:
Input: noX("xaxb")
Output: "ab"

Input: noX("abc")
Output: "abc"

Input: noX("xx")
Output: ""

A.14: Given an array of ints, compute recursively if the array contains somewhere a value followed in the array by that value times 10. We'll use the convention of considering only the part of the array that begins at the given index. In this way, a recursive call can pass index+1 to move down the array. The initial call will pass in index as 0.

Example:
Input: array220([1, 2, 20], 0)
Output: True

Input: array220([3, 30], 0)
Output: True

Input: array220([3], 0)
Output: False

A.15: Given a string, compute recursively a new string where all the lowercase 'x' chars have been moved to the end of the string.

Example:
Input: endX("xxre")

Output: "rexx"

Input: endX("xxhixx")
Output: "hixxxx"

Input: endX("xhixhix")
Output: "hihixxx"

A.16: Given a string, compute recursively (no loops) the number of "11" substrings in the string. The "11" substrings should not overlap.

Example:
Input: count11("11abc11")
Output: 2

Input: count11("abc11x11x11")
Output: 3

Input: count11("111")
Output: 1

A.17: Given a string that contains a single pair of parenthesis, compute recursively a new string made of only of the parenthesis and their contents, so "xyz(abc)123" yields "(abc)".

Example:
Input: parenBit("xyz(abc)123")
Output: "(abc)"

Input: parenBit("x(hello)")
Output: "(hello)"

Input: parenBit("(xy)1")
Output: "(xy)"

A.18: Given a string and a non-empty substring sub, compute recursively if at least n copies of sub appear in the string somewhere, possibly with overlapping. N will be non-negative.

Example:
Input: strCopies("catcowcat", "cat", 2)
Output: True

Input: strCopies("catcowcat", "cow", 2)
Output: False

Input: strCopies("catcowcat", "cow", 1)
Output: True

A.19: Given a string, compute recursively (no loops) a new string where all the lowercase 'x' chars have been changed to 'y' chars.

Example:
Input: changeXY("codex")
Output: "codey"

Input: changeXY("xxhixx")
Output: "yyhiyy"

Input: changeXY("xhixhix")
Output: "yhiyhiy"

A.20: Given an array of ints, compute recursively if the array contains a 6. We'll use the convention of considering only the part of the array that begins at the given index. In this way, a recursive call can pass index+1 to move down the array. The initial call will pass in index as 0.

Example:
Input: array6([1, 6, 4], 0)
Output: True

Input: array6([1, 4], 0)
Output: False

Input: array6([6], 0)
Output: True

A.21: Given a string, compute recursively a new string where all the adjacent chars are now separated by a "*".

Example:
Input: allStar("hello")
Output: "h*e*l*l*o"

Input: allStar("abc")
Output: "a*b*c"

Input: allStar("ab")
Output: "a*b"

A.22: We'll say that a "pair" in a string is two instances of a char separated by a char. So "AxA" the A's make a pair. Pair's can overlap, so "AxAxA" contains 3 pairs -- 2 for A and 1 for x. Recursively compute the number of pairs in the given string.

Example:
Input: countPairs("axa")
Output: 1

Input: countPairs("axax")
Output: 2

Input: countPairs("axbx")
Output: 1

A.23: Given a string, return recursively a "cleaned" string where adjacent chars that are the same have been reduced to a single char. So "yyzzza" yields "yza".

Example:
Input: stringClean("yyzzza")
Output: "yza"

Input: stringClean("abbbcdd")
Output: "abcd"

Input: stringClean("Hello")
Output: "Helo"

A.24: Given a string, return true if it is a nesting of zero or more pairs of parenthesis, like "(())" or "((()))". Suggestion: check the first and last chars, and then recur on what's inside them.

Example:
Input: nestParen("(())")
Output: True

Input: nestParen("((()))")
Output: True

Input: nestParen("(((x))")
Output: False

A.25: Given a string and a non-empty substring sub, compute recursively the largest substring which starts and ends with sub and return its length.

Example:
Input: strDist("catcowcat", "cat")
Output: 9

Input: strDist("catcowcat", "cow")
Output: 3

Input: strDist("cccatcowcatxx", "cat")
Output: 9

A.26: Given an array of ints, is it possible to choose a group of some of the ints, such that the group sums to the given target? This is a classic backtracking recursion problem. Once you understand the recursive backtracking strategy in this problem, you can use the same pattern for many problems to search a space of choices. Rather than looking at the whole array, our convention is to consider the part of the array starting at index start and continuing to the end of the array. The caller can specify the whole array simply by passing start as 0. No loops are needed -- the recursive calls progress down the array.

Example:
Input: groupSum(0, [2, 4, 8], 10)

Output: True

Input: groupSum(0, [2, 4, 8], 14)
Output: True

Input: groupSum(0, [2, 4, 8], 9)
Output: False

A.27: Given an array of ints, is it possible to divide the ints into two groups, so that the sums of the two groups are the same. Every int must be in one group or the other. Write a recursive helper method that takes whatever arguments you like, and make the initial call to your recursive helper from splitArray(). (No loops needed.)

Example:
Input: splitArray([2, 2])
Output: True

Input: splitArray([2, 3])
Output: False

Input: splitArray([5, 2, 3])
Output: True

A.28: Given an array of ints, is it possible to divide the ints into two groups, so that the sum of one group is a multiple of 10, and the sum of the other group is odd. Every int must be in one group or the other. Write a recursive helper method that takes whatever arguments you like, and make the initial call to your recursive helper from splitOdd10(). (No loops needed.)

Example:
Input: splitOdd10([5, 5, 5])
Output: True

Input: splitOdd10([5, 5, 6])
Output: False

Input: splitOdd10([5, 5, 6, 1])
Output: True

A.29: Given an array of ints, is it possible to divide the ints into two groups, so that the sum of the two groups is the same, with these constraints: all the values that are multiple of 5 must be in one group, and all the values that are a multiple of 3 (and not a multiple of 5) must be in the other. (No loops needed.)

Example:
Input: split53([1, 1])
Output: True

Input: split53([1, 1, 1])
Output: False

Input: split53([2, 4, 2])

Output: True

A.30: Given an array of ints, is it possible to choose a group of some of the ints, such that the group sums to the given target with these additional constraints: all multiples of 5 in the array must be included in the group. If the value immediately following a multiple of 5 is 1, it must not be chosen. (No loops needed.)

Example:
Input: groupSum5(0, [2, 5, 10, 4], 19)
Output: True

Input: groupSum5(0, [2, 5, 10, 4], 17)
Output: True

Input: groupSum5(0, [2, 5, 10, 4], 12)
Output: False

Chapter 7 - Graphs

7.1 Introduction

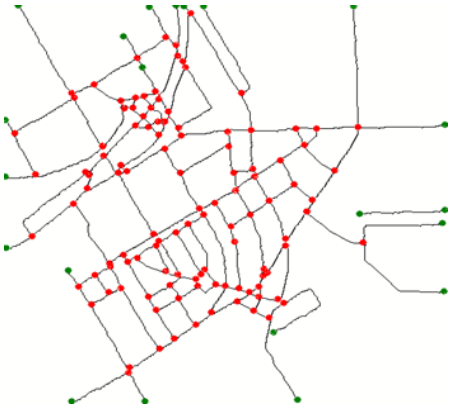


Figure 1: A graph for a road network[3]

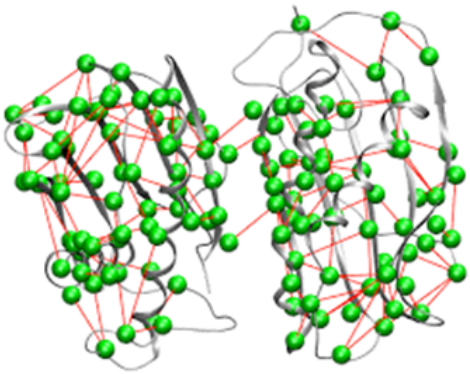


Figure 2: a graph showing a Protein Structure[4]

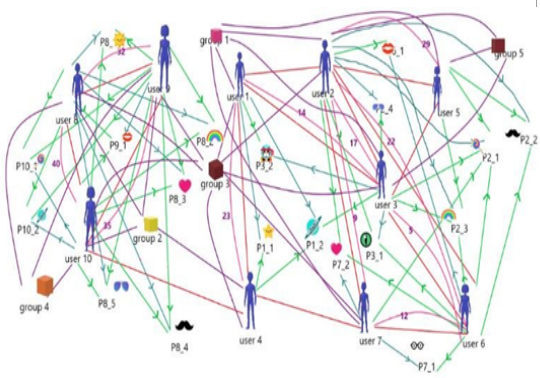


Figure 3: A Facebook network[6]

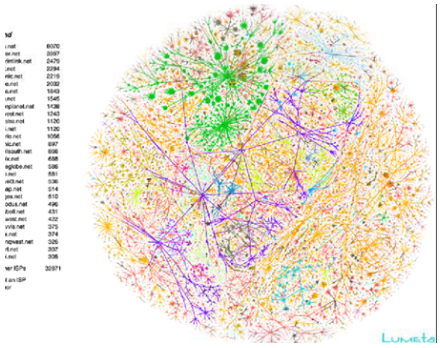


Figure 4: graph visualizing the internet[5]

So far, we have learned about data structures that can store linear sequences (e.g., arrays, lists, stack, queues) and data structures that can represent non-linear hierarchical relationships (i.e., different types of trees). What can be the most generic data structures that can represent arbitrary relationships among entities without any ordering restrictions? The answer is graphs. The graph is the most versatile data structure that can represent any relationship network among a collection of objects. For example, we can use graphs to represent road networks, the

[3] Image source: https://www.researchgate.net/figure/An-example-of-road-network-extraction-and-graph-representation_fig12_279263325

[4] Image source: https://benthamopen.com/contents/supplementary-material/TOBIOIJ-5-53_SD1.pdf

[5] Image source: https://www.researchgate.net/figure/2-A-graph-visualisation-of-the-topology-of-network-connections-of-the-core-of-the_fig2_239550496

[6] Image source: https://www.researchgate.net/publication/340347681_A_Study_on_Graph_Theory_Properties_of_On-line_Social_Networks

internet, structure of molecules and proteins, social networks, evolutionary relationships among species, geographic regions, and so on.

In fact, a single graph can capture multiple types of entities and relationships. Whenever we have a collection of entities and need to model their interactions and relations as a network, we can use graphs. Therefore, being able to store and use graphs in our programs is absolutely essential. Graphs are so important in computer science that graph theory and graph algorithms are considered core computer science courses and many books have been written on them. Even drawing a graph for visualization and human analysis is a vast topic taught as a separate course in many universities! Here we will only learn the basics related to graphs: the terminologies we have to understand, common graph data structure representations, and how to traverse a graph.

7.2 Graph Terminologies

A graph G is represented as a set of vertices (or nodes) V that represent the entities/objects/concepts and a set of edges (or links) E that represent relationship/interaction among those vertices. So, we typically write a graph G as G = (V, E). If a and b are two vertices in V and if there is an edge between them in the graph, we commonly write the edge as (a, b). When there is an edge (a,b) in E, we say vertex a and b are adjacent or neighbors. Note that the relationship a graph captures may be directional or undirected. In the former case we have a directed graph, in the latter case, an undirected graph. Below are examples of a directed and an undirected graph. We typically draw edges in an undirected graph as lines/curves and edges in a directed graph as arrows.

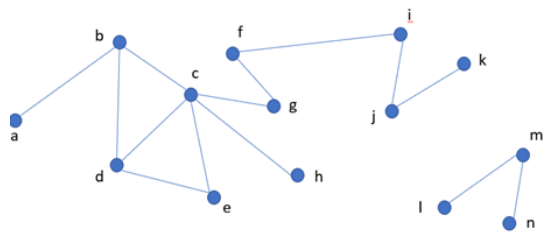


Figure 5: an undirected graph

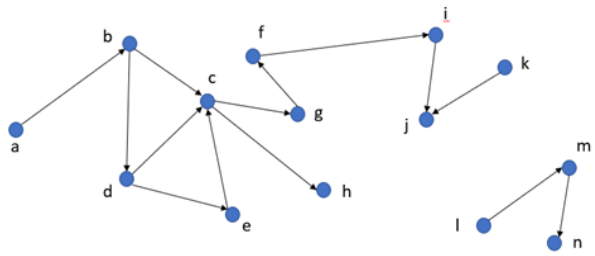


Figure 6: a directed graph

Notice that in a undirected graph having an edge (a,b) in E means the same thing as having the edge (b,a). However, in a directed graph (a,b) means an edge from a to b, while (b,a) means an edge from b to a. We can have none, either, or both in a directed graph. If we have both the edges, then we have to draw two arrows (one from a to b and another from b to a) when drawing the graph.

The **degree** of a vertex in an undirected graph is the number of edges the vertex has. For example, in Figure 5, degree of vertex d is 3. In a directed graph, a vertex has both in-degree

and out-degree. The former represents the number of edges originating from other vertices and ending at that vertex, the latter represents the opposite. So, the in-degree of vertex d in Figure 6 is 1 and out-degree is 2. A directed edge (a,b) in E is an outgoing edge of a and an incoming edge of b.

A **path** between two vertices x and y in a graph is a sequence of edges of the form ($x = u_0$, u_1),(u_1, u_2),(u_2,u_3),…,(u_{n-1}, u_n=y). For example, the sequences of edges (a,b),(b,c),(c,e),(e,d) forms a path between vertex a and d in the graph of Figure 5. The length of the path is the number of edges in it. Note that, there may be zero, one, or more paths between any two vertices in a graph. When there is no path between two vertices u, v in a graph then the graph is called **disconnected**. Then we also say that u and v are unreachable from each other. The subset of vertices that are mutually reachable from one another along with the various edges form a **component** of a disconnected graph. Hence, there are two components in the graph of Figure 5.

Note that in the directed graph of Figure 6, there is no path (a,b),(b,c),(c,e),(e,d) as the edges are directional. However, vertex d is still reachable from a as we have the path (a,b),(b,d). Notice that, a is not reachable from d in this case, this is again because the edges are directional. The **distance** of a vertex v from another vertex u in a graph is the number of edges in the shortest path from u to v.

A **cycle** in a graph is a sequence of edges that starts and ends at the same vertex. For example in Figure 5, (b,d),(d,e),(e,c),(c,b) edge sequence form a cycle. In a directed graph, the cycles are also directional. Hence, there is no directed cycle in the graph of Figure 6.

In both directed and undirected cases, the vertices and/or the edges of the graph can have weights assigned to them. Then we call the graph a weighted directed/undirected graph. The weight assigned to a vertex is used to represent the importance of the entity/object/concept it represents. The weight assigned to an edge typically represents the cost or strength of the relationship among the vertices it connects. For example, in a road network graph, the vertices are road junctions and the edges are road segments. Then the weight of a junction can be the maximum traffic capacity at that junction and the weight of the edge between two junctions can be their distance, aka, the length of the road. Below are two examples of weighted graphs.

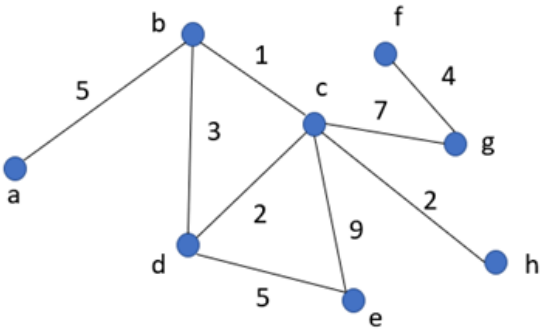


Figure 7: an edge weighted undirected graph

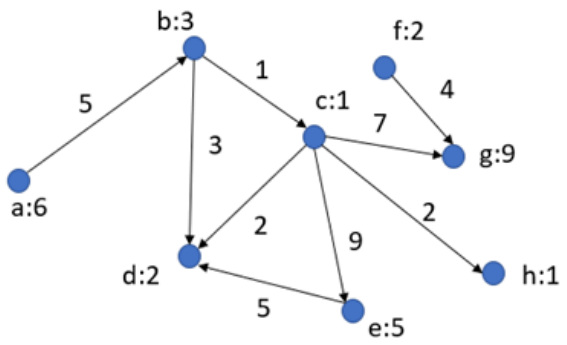


Figure 8: directed graph having both vertex and edge weights

We say a graph is sparse when the number of edges is too low compared to the number of vertices. We call it a dense graph when the number of edges is high. This distinction is important when we choose among various data structure representations of a graph.

There are many more definitions related to graphs. However, for our purpose, knowing up to this much is sufficient. Given an undirected, unweighted graph is the most common case, in the rest of this chapter we will use the term graph to mean an undirected, unweighted graph by default. When we will refer to the other cases, we mention them explicitly.

7.3 Graph Representations

Unlike a tree, a graph generally does not have a root vertex/node that we can use to recursively discover all other vertices. The reason for that is simple. As the graph may be disconnected, it may be impossible to explore the entire graph if we are given only a single starting vertex. Hence, in any typical graph representation we have all the vertices and edges given. The most common representations of a graph are the following:

1. Adjacency List Representation
2. Adjacency matrix representation, and
3. Incidence matrix representation

7.3.1 Adjacency List Representation

In this representation, the vertices of the graph are represented by the indices of an array. Each entry of the array is a linked list. The elements of the list contain indices of the vertices that are adjacent to the vertex owning the index.

Let us consider the graph of Figure 5, and assume vertices a to n are assigned successive increasing indices starting from 0. Then the adjacency list representation of the graph is as follows:

a 0	b 1	c 2	d 3	e 4	f 5	g 6	h 7	I 8	j 9	K 10	l 11	m 12	n 13
1	0 2 3	1 3 4 6 7	1 2 4	2 3	6 8	2 5	2	5 9	8 10	9	12	11 13	12

Table 1: adjacency list representation of graph in Figure 5

It is quite easy to represent a directed graph using this format also. Everything remains the same, we just include the edge in the entry for the vertex where the edge starts from – not where it ends. Therefore, the adjacency list representation of the directed graph of Figure 6 is as follows:

a 0	b 1	c 2	d 3	e 4	f 5	g 6	h 7	I 8	j 9	K 10	l 11	m 12	n 13
1	2 3	6 7	2 4	2	8	5		9		9	12	13	

Table 2: Adjacency list representation of the graph in Figure 6

When we have a weighted graph to represent then this plain array of list of numbers is not sufficient. Let us consider the most generic case, where both the vertices and the edges can have weights. We now have to construct an Edge class to hold all information about an edge and a separate array holding the node weights. Then the Edge class should look as follows:

```
Class Edge {
        int ep1
        int ep2
        Int weight
}
```

Here in the properties of the Edge class, ep1 and ep2 are the indices of the vertices that an edge connects. With this modification, we can represent the weighted directed graph of Figure 8 as follows.

Node Weight Array	a	b	c	d	e	f	g	h
	0	1	2	3	4	5	6	7

	6	3	1	2	5	2	9	1
Adjacency List	a	b	c	d	e	f	g	h
	0	1	2	3	4	5	6	7
	<0,1,5>	<1,2,1> <1,3,3>	<2,3,2> <2,4,9> <2,6,7> <2,7,2>		<4,3,5>	<5,6,4>		

Table 3: Adjacency list representation of the weighted directed graph of Figure 8

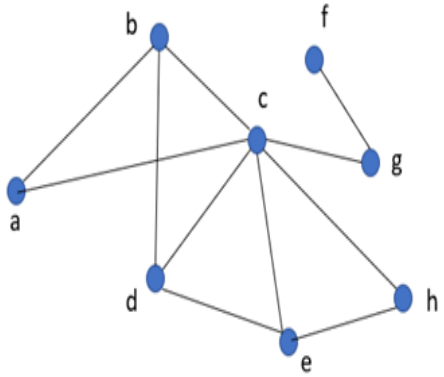
In the above, the tuple <x,y,z> stands for <ep1,ep2,weight> of an edge. Notice that in the directed weighted graph case, the edge class can record only the destination vertex of an edge; the source is not needed as it is the same as vertex owning the index of the array holding the list of edges. However, the way we defined the edge class, we can represent both directed and undirected weighted graphs using that class.

Finally, note that the vertices and edge can have other properties along with their weights in graph representations of real-world scenarios. In that case, we can define a Node class for the vertices and have an array of nodes and include more properties related to the edges in the Edge class. One can further, combine the two arrays by having a node class holding the list of edges along with a vertex's other properties.

Space Complexity: if a graph has N vertices and M edges, then the adjacency matrix representation needs an array of size N and the total number of nodes in various lists in different array indices will be total 2M (in case of undirected graph) or M (in case of a directed graph). Hence the space requirement for the adjacency list representation is proportional to N + M. In the asymptotic notation, we can write the space complexity as $O(N+M)$.

7.3.2 Adjacency Matrix Representation

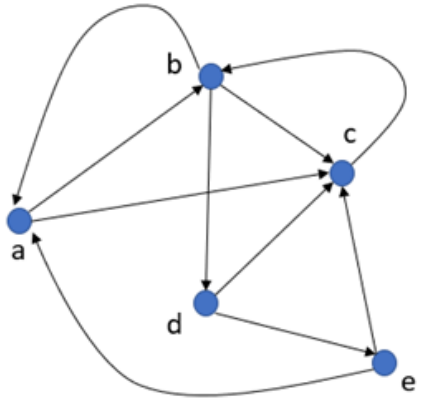
The adjacency matrix representation of a graph with N vertices uses an N×N matrix. As in the case of previous representation, vertices are assigned increasing indices starting from 0. The i^{th} row and column of the matrix are for the i^{th} vertex of the graph. The entry at i^{th} row and j^{th} column of the matrix is 1 if there is an edge between the corresponding vertices of the graph. Below is the drawing and adjacency matrix of a graph as an example.



	0	1	2	3	4	5	6	7
a 0	0	1	1	0	0	0	0	0
b 1	1	0	1	1	0	0	0	0
c 2	1	1	0	1	1	0	1	1
d 3	0	1	1	0	1	0	0	0
e 4	0	0	1	1	0	0	0	1
f 5	0	0	0	0	0	0	1	0
g 6	0	0	1	0	0	1	0	0
h 7	0	0	1	0	1	0	0	0

Table 4: an adjacency matrix representation of an undirected unweighted graph

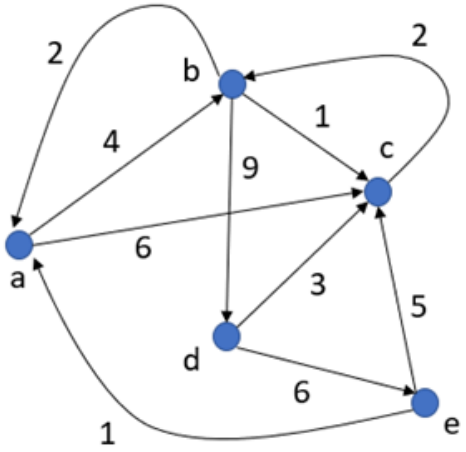
Notice that the matrix is symmetric. That is if we call the matrix A then each A[i][j] entry is the same as A[j][i] entry for all i,j < N. The reason for this is simple, as the edges are undirected, each edge occurs twice in the matrix. The adjacency matrix of a directed graph is not symmetric, unless of course when for each (a,b) edge of the graph there is also a (b,a) edge from vertex b to vertex a. Below is an adjacency matrix representation of a small directed graph.



	0	1	2	3	4
a 0	0	1	1	0	0
b 1	1	0	1	1	0
c 2	0	1	0	0	0
d 3	0	0	1	0	1
e 4	1	0	1	0	0

Table 5: an adjacency matrix representation of a directed graph

If a graph has only weight on its edges, then we can represent it using the adjacency matrix just as easily in both directed and undirected cases. If there is an edge (a,b) with weight w in the graph and the indices of a and b are i and j in the matrix respectively, then we simply write w in the <i,j> cell of the adjacency matrix, instead of writing 1. If the edge is undirected then we do the same in <j,i> cell also. Below is an adjacency representation when the earlier directed graph is edge-weighted (i.e., only the edges have weights).



	0	1	2	3	4
a 0	0	4	6	0	0
b 1	2	0	1	9	0
c 2	0	2	0	0	0
d 3	0	0	3	0	6
e 4	1	0	5	0	0

Table 6: adjacency matrix representation of a weighted directed graph

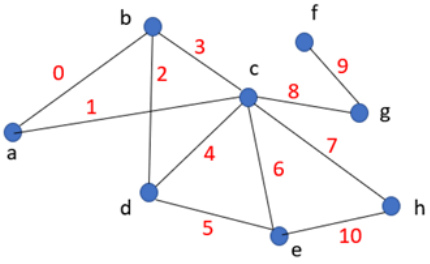
When both vertices and the edges of a graph have weights then we need a node weight array as in Table 3 to represents the weights of the vertices along with the adjacency matrix that captures the edge weights. Write the adjacency matrix representation of the graph of Figure 8 as an exercise of this case.

Space Complexity: for a graph with N vertices and M edges, the adjacency matrix representation will take N×N entries, that is, N^2 entries. Consequently, the asymptotic space complexity in this representation is $O(N^2)$. Apparently, the space cost of adjacency matrix representation is significantly higher than that of adjacency list representation. Then why we use this second representation? The answer is many computations are easier in the matrix representation than in the list representation. We trade space with time when using any representation.

7.3.3 Incidence Matrix Representation

The final commonly used representation for graphs is the incidence matrix representation. This representation is particularly useful and popularized by electrical circuit analysis where edges represent wires with resistance/inductance/capacitance and the vertices represents junctions where the edges connect and where voltage being applied. You would be surprised to know that this representation is more than 150 years old!

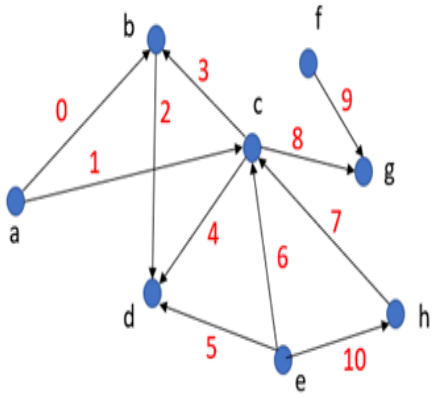
In the incidence matrix representation, the edges and vertices both are numbered. Thus, if there are M edges and N vertices in the graph, edges are numbered from 0 to M – 1 and vertices are from 0 to N – 1. Then a M×N matrix is used where the rows are the edges and the columns are the vertices. We write a 1 in the two columns of a row to indicate that these are the two vertices connected by the edge owning that row. The remaining entries of the row are filled with zeros. Below is an example (here the indices of the edges are shown in red color).



	a:0	b:1	c:2	d:3	e:4	f:5	g:6	h:7
0	1	1	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0
2	0	1	0	1	0	0	0	0
3	0	1	1	0	0	0	0	0
4	0	0	1	1	0	0	0	0
5	0	0	0	1	1	0	0	0
6	0	0	1	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0
9	0	0	0	0	0	1	1	0
10	0	0	0	0	1	0	0	1

Table 7: The incidence matrix representation of an undirected graph

When we have a directed graph, we put a 1 in the source of the edge and a -1 in the destination (or vice versa). Below is an example.



	a:0	b:1	c:2	d:3	e:4	f:5	g:6	h:7
0	1	-1	0	0	0	0	0	0
1	1	0	-1	0	0	0	0	0
2	0	1	0	-1	0	0	0	0
3	0	-1	1	0	0	0	0	0
4	0	0	1	-1	0	0	0	0
5	0	0	0	-1	1	0	0	0
6	0	0	-1	0	1	0	0	0
7	0	0	-1	0	0	0	0	1
8	0	0	1	0	0	0	-1	0
9	0	0	0	0	0	1	-1	0
10	0	0	0	0	1	0	0	-1

Table 8: incidence matrix representation of a directed unweighted graph

You probably already understand how to represent an edge-weighted graph in this representation. Instead of writing 1 and -1, we have to put the positive and negative weight of the edge in the proper columns in a row.

Space Complexity: for a graph with N vertices and M edges, the incidence matrix representation will require M×N entries. Consequently, the asymptotic space complexity in this representation is $O(MN)$. When the underlying graph is sparse, the incidence matrix representation may be cheaper than the adjacency matrix representation or similar in cost. When the opposite is true, the adjacency matrix is better. Electrical circuits are typically sparse graphs, that is why, incidence matrix representation is quite popular in circuit analysis. However, there are algorithms that specifically need the incidence matrix representation for computation efficiency.

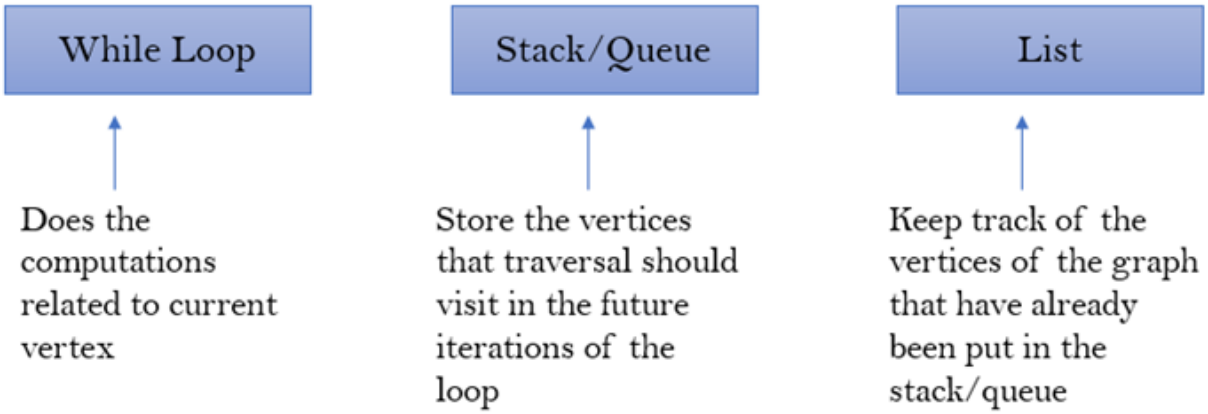
Practice Problems:

1. Given an unweighted graph in the adjacency matrix representation, calculates the number of edges the graph has.
2. Given a directed unweighted graph in the adjacency matrix representation, generate the adjacently list representation of the graph.
3. Solve the following problems for all three types of representation of the input graph.
    a. Given an undirected, unweighted graph as input, write a function to find the vertex with maximum degree and return the degree of that vertex.
    b. Given an undirected, edge-weighted graph as input, write a function to find the vertex whose sum of edge weights is maximum.
    c. Solve Problem #a and #b for directed, edge-weighted graph considering only outgoing edges.

7.4 Graph Traversals

To solve any interesting problem that involves graphs, one needs to know how to traverse a graph. When we traverse a graph, we start from an arbitrary vertex then follow its edges to reach its neighbor vertices, then follow their edges to their neighbors, and so on. Its sounds just like traversing a tree starting from a root node and following the children of a node in each step. Doesn't it? However, there are two important caveats. Since, unlike a tree, graph is not a hierarchical data structure, a recursive traversal can lead us back to an earlier vertex when the graph has cycles. When that happens, we will have an infinite recursion, that will keep rotating within the vertices of the first cycle it gets sucked into. The second concern is that, since a graph may be disconnected, a traversal starting from a particular vertex will only visits the vertices of the component where the starting vertex belongs to. For both these reasons, the

straightforward recursive traversals that we used for trees do not apply for graphs. Rather, the recursive graph traversal is implemented using while loops and stack/queue and a list as supporting data structures to keep track of the progress of traversal. The following diagram explains the role of the different components of a graph traversal.



The general approach is, we insert the starting vertex in the stack/queue and also in the list. Then we enter the while loop that will iterate until the stack/queue is empty. In each iteration of the loop, we will remove the top vertex from the stack/queue using the stack pop or queue dequeue method. We do the necessary computation using that vertex. Then we get the list of neighbors of that vertex and insert only those vertices in the stack/queue that are not in the list. As we insert a vertex in the stack/queue, we also insert that vertex in the list.

The above approach will visit all the vertices of the graph that are reachable from the starting vertex through some path. When dealing with a disconnected graph, we have to do one more thing. That is, after the while loop ends, we need to check if all vertices of the input graph are included in the list where we keep track of the visited vertices. If we find a vertex that is not already in the list, then we start the whole process of the previous paragraph by using that vertex as the starting point.

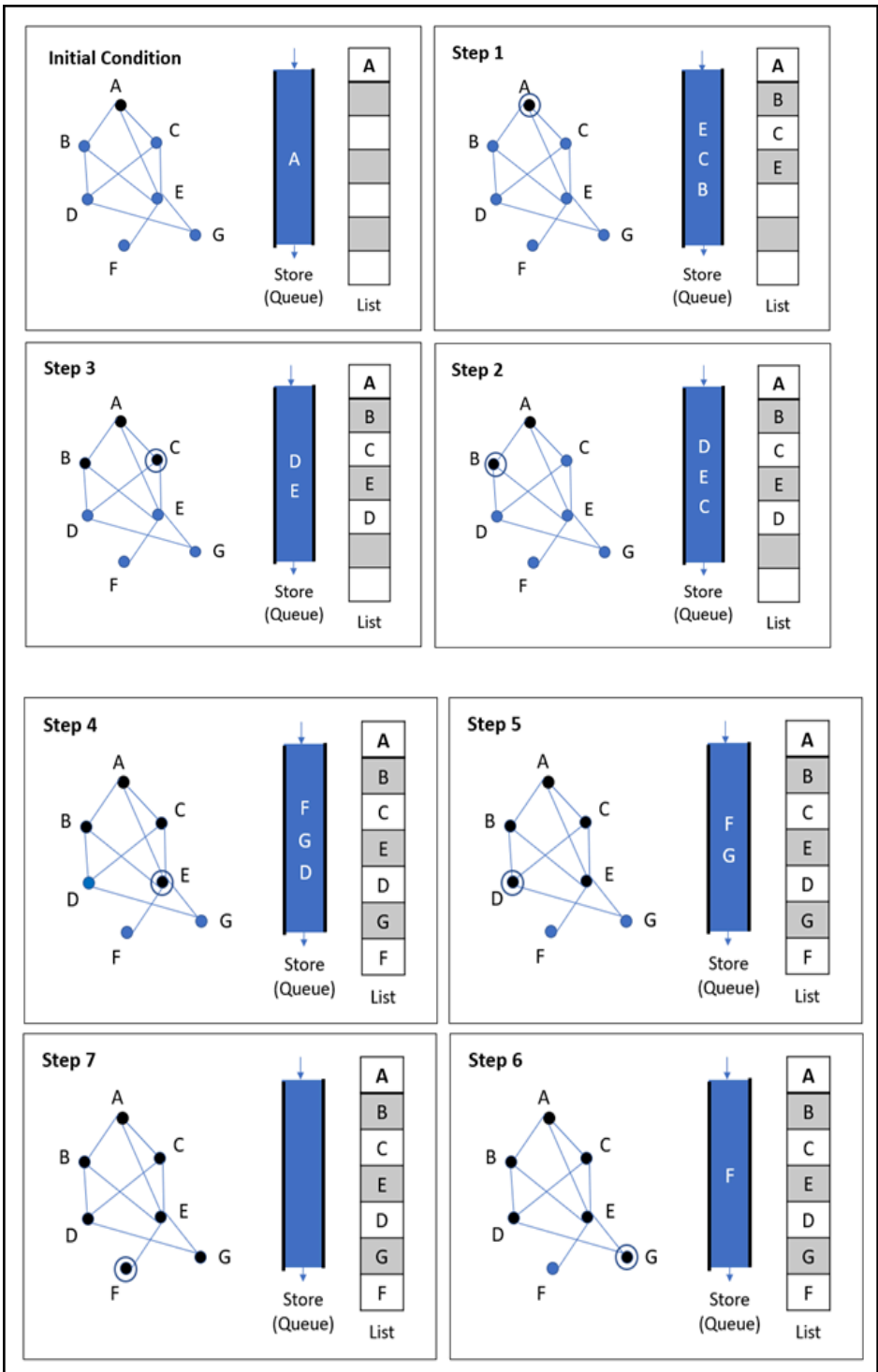
Below is a pseudo-code of the graph traversal process.

```
Algorithm  Traverse-Graph(G)
        visited := new List
        store := new Queue/Stack
        V := G.vertices
        start := V[0]
        while (start != nill) {
        store.insert(start)
        visited.insert(start)
        while(store.isEmpty() == FALSE) {
```

```
                u := store.removeNext()
                // do computation related to u
                edges := u.edges
                foreach (e in edges) {
                        if (e.ep1 == u) {
                                v := e.ep2
                        } else {
                                v := e.ep1
                                }
                                If (visited.contains(v) == FALSE) {
                                        visited.append(v)
                                        store.insert(v)
                                }
                }
        }
        start = nill
        foreach (v in V) {
                If (visited.contains(v) == FALSE) {
                        start := v
                        break
                }
        }
}
```

7.4.2 Breadth First and Depth First Traversals

Interestingly, we have two very different graph exploration patterns depending on whether we used a stack or a queue in the aforementioned traversal algorithms. Let us examine the behavior of the graph traversal for a small graph and a queue as the store of vertices to be visited next, as in Figure 9. For simplicities sake, we use an example of a connected graph.



Notice that in this given a queue is a first come first out data structure, the item that is oldest in the queue comes out when we invoke the remove operation in the traversal algorithm.

As a result, if we start the traversal from vertex A then the next three vertices that are visited are the three neighbors B,C, and E of A.

We can explain the traversal progression as follows: starting from A, all vertices that are at distance 1 from A is visited next (B, C, and E). Then vertices that are at distance 2 from A are visited (D and G). Finally, the vertices that are at distance 3 from A are visited (F).

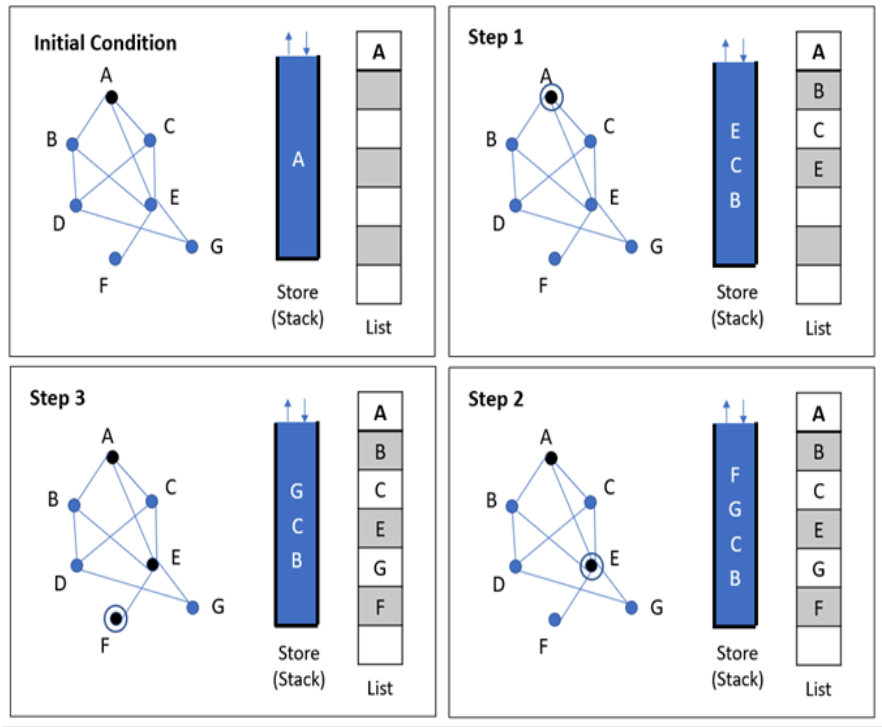
As if we start from the center A and expanding our search within the breadth of successively increasing distances.

This is why this traversal process is called **Breadth First Search** or **BFS**.

Figure 9: Breadth First Graph Traversal of a Graph

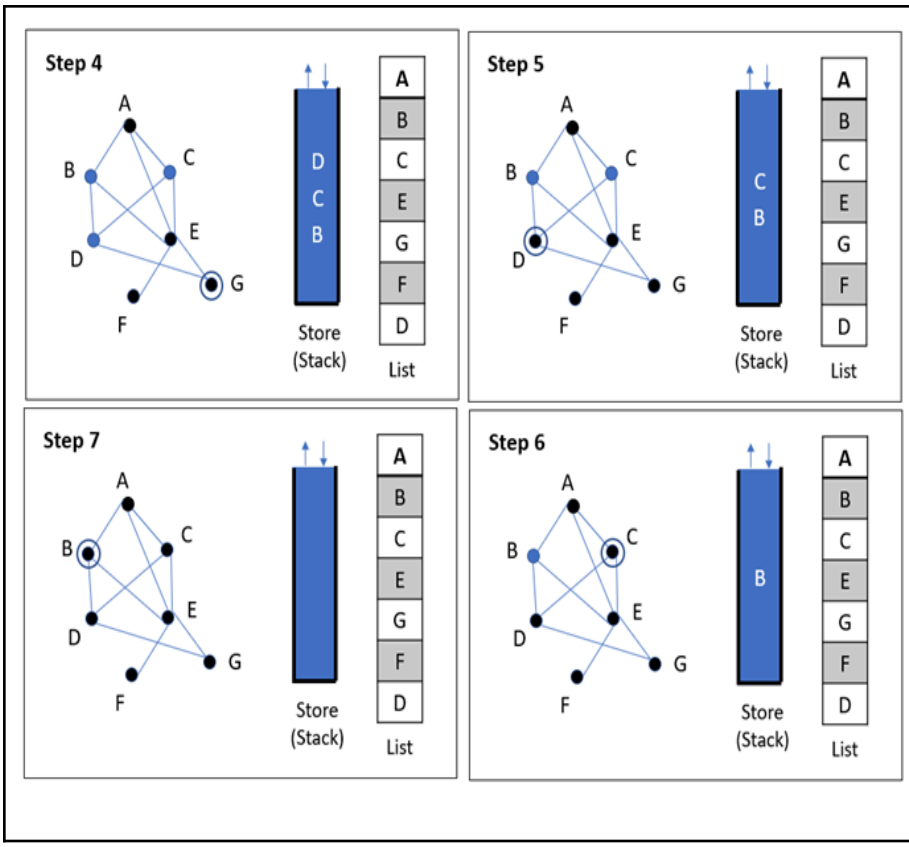
The algorithm for graph traversal visits each vertex only once and when visiting it, it has to evaluate all its edges. For an undirected graph, each edge is evaluated twice as the edge is considered from the both endpoints. Hence, if a graph has n vertices and m edges, the traversal itself takes n + 2m steps. However, in each step, checking whether the other endpoint of an edge is in the visited list take n steps in the worst case. So the running time of the traversal algorithm is $O(n + 2m + n^2) = O(m + n^2)$. However, if we use a Boolean array instead of the visited list for tracking the already considered vertices, then checking if a vertex needs to be put in the store takes a constant time. Then running time of traversal becomes $O(n + m)$.

If we rather use a stack for the store then the exploration pattern would be as shown in Figure 10.



Notice that starting from A, the traversal algorithm inserts the neighbors of A in the store in the order B, C, E as in the BFS case. However, given the store is a stack, which is a last come first serve data structure, the vertex that comes out from the store in the next iteration is E.

Progressing from E, the neighbors G and F are inserted in the store. F comes out from the store in the next iteration. Since F has no neighbor that are not in the visited list, the next iteration pops G from the store. Which puts D in the



store.

So, we see that the traversal process is as such that, in each iteration, it goes one distance farther from the starting vertex following some path. When it cannot move any farther, the traversal backtracks to the previous step and try another path from there.

This is why this traversal process is called **Depth First Search** or **DFS**.

Figure 10: Depth First Traversal of a Graph