Data Structure and Algorithm

Priority	High
Status	In Progress
Type	Backend Software Engineering

Algorithm 1 (Princeton University)

Coursera | Online Courses & Credentials From Top Educators. Join for Free | Coursera

Learn online and earn valuable credentials from top universities like Yale, Michigan, Stanford, and leading companies like Google and IBM. Join Coursera for free and transform your career with degrees, certificates, Specializations, & MOOCs in data science, computer science, business, and

https://www.coursera.org/learn/algorithms-part1/home/welcome

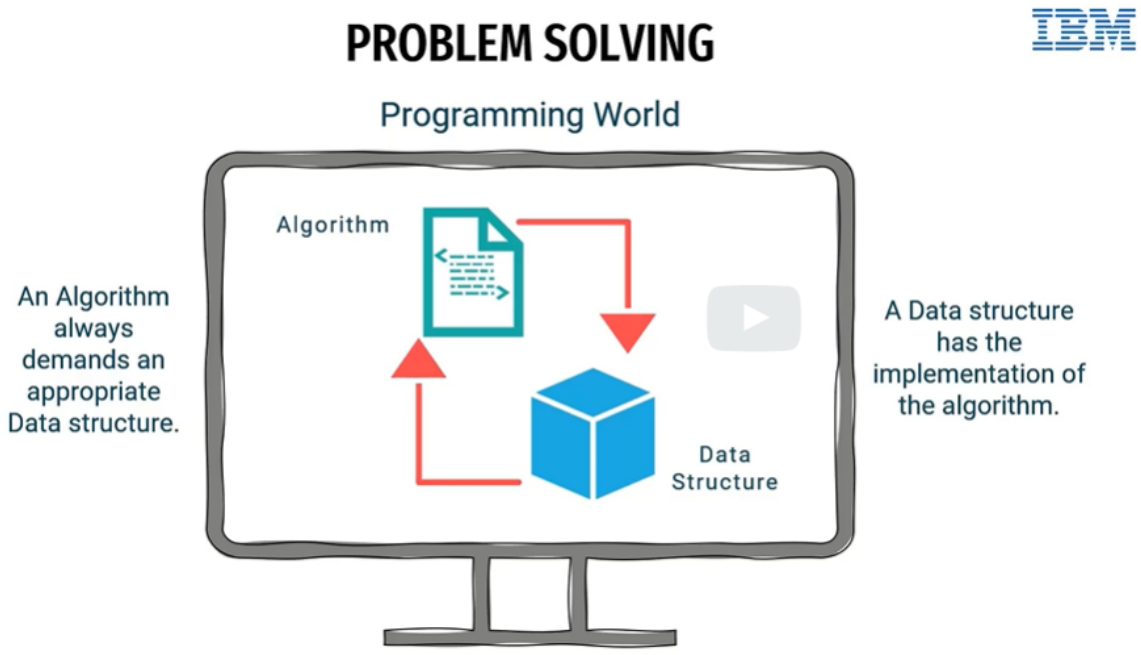
Curriculum

Algorithhms 4th Edition by Robert Sedgewick, Kevin Wayne.pdf

https://drive.google.com/file/d/17MObZ0XQy9pUI-IBslyp9bW5R5v0pqKI/view?usp=sharing

topic	data structures and algorithms
data types	stack, queue, bag, union-find, priority queue
sorting	quicksort, mergesort, heapsort
searching	BST, red-black BST, hash table
graphs	BFS, DFS, Prim, Kruskal, Dijkstra
strings	radix sorts, tries, KMP, regexps, data compression
advanced	B-tree, suffix array, maxflow

▼ Data Structures and Algorithms Core Concepts



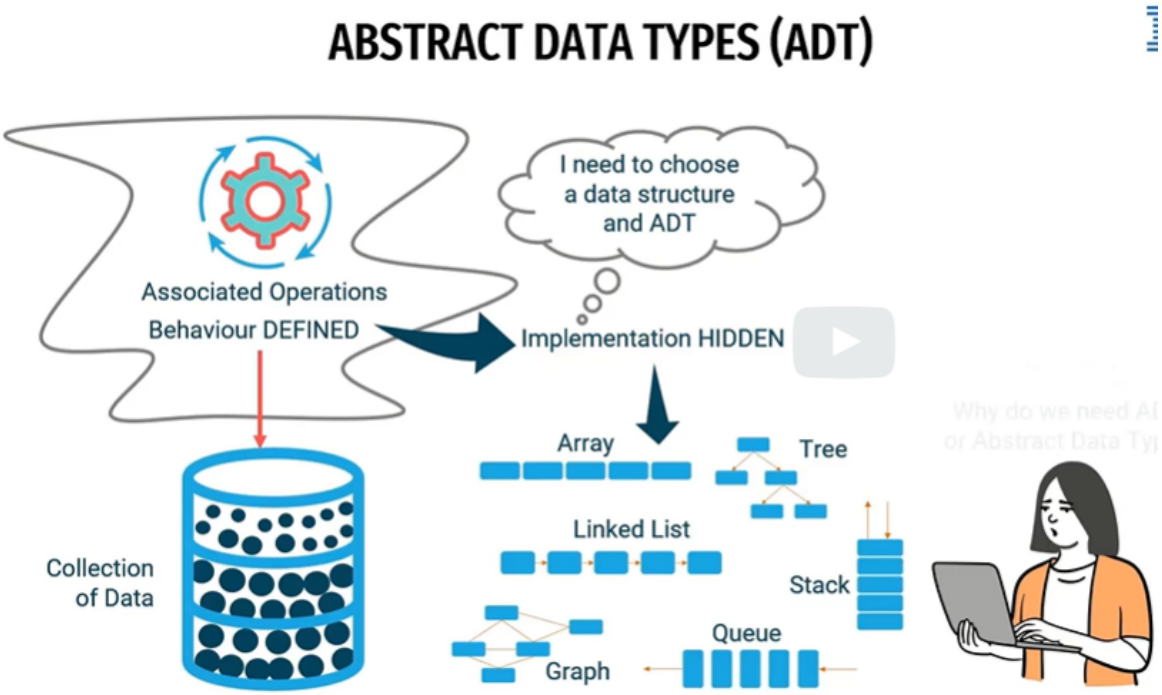
Algorithm

- Problem-solving procedure
- Step-by-step procedure for performing a task within a specified time.
- Operates on a collection of data with a definite beginning, a definite end, and finite steps.

Data Structure

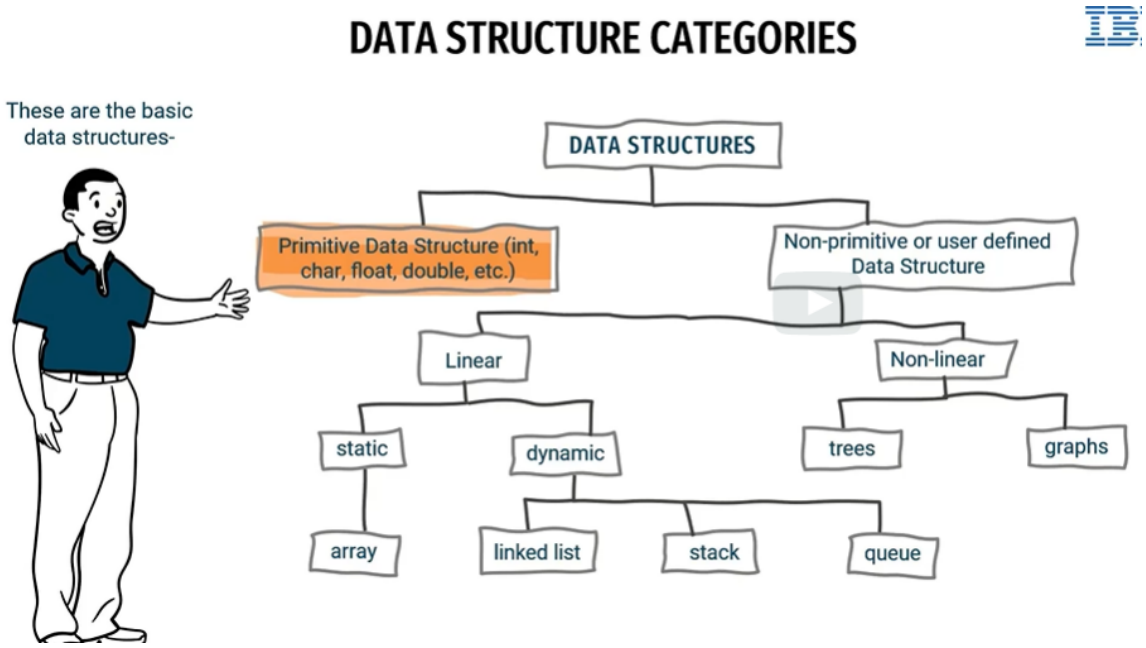
- The way we store and organize the data
- there are different ways to store data for example Array, Stack, LinkedList, Queue, Hash, Graph, Tree, etc.
- We need to choose the correct data structure based on our data nature, to be resulting an efficient program.

Abstract Data Types (ADT)



Represent the collection of data and set of operations performed on the data.

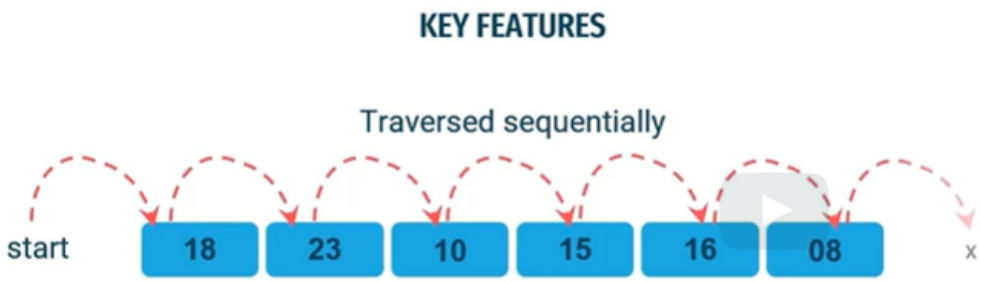
▼ Data Structure Categories

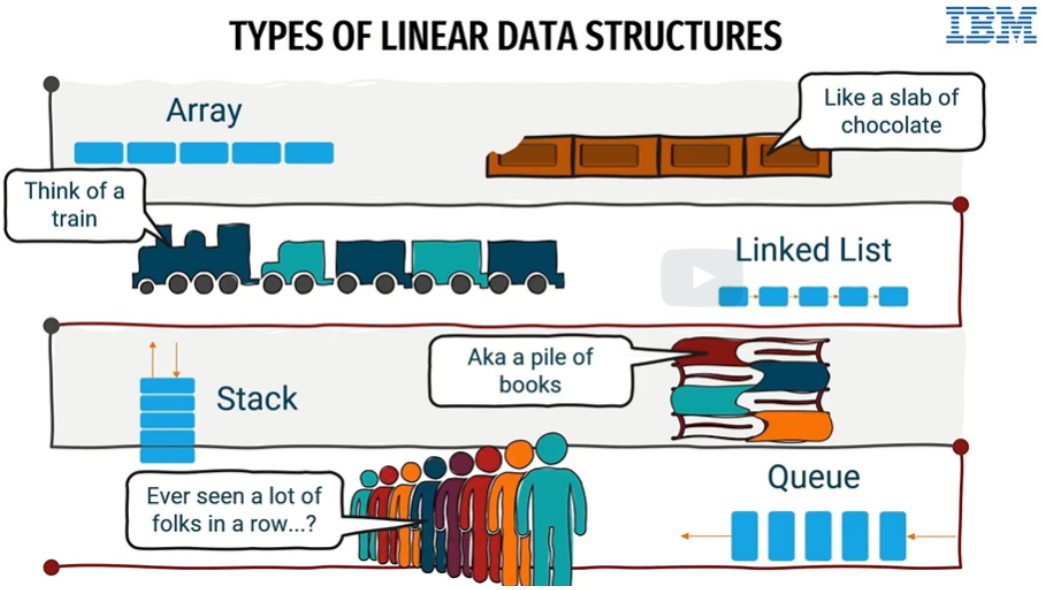


Linear Data Structure

- Can be constructed as a continuous arrangement of data elements in the memory.

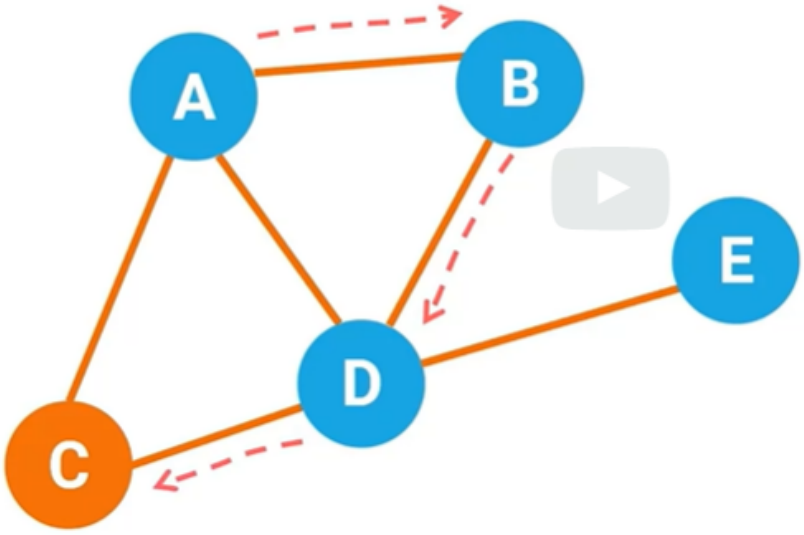
- The elements are traversed sequentially to access the data.



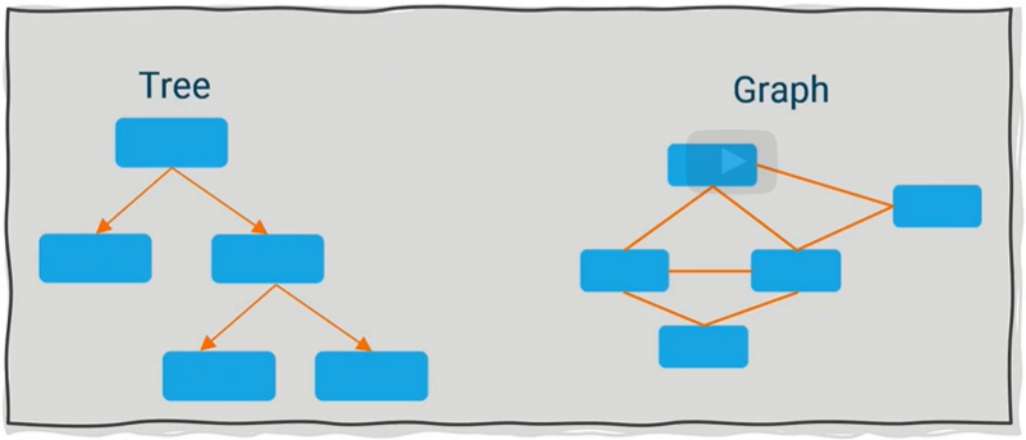


Non-Linear Data Structure

- Data is not stored in a linear or sequential order.
- A data structure can be constructed as a collection of a randomly distributed set of data items joined together by using a reference pointer.



- Examples of Non-Linear Data structures are Tree and Graph



Summaries

- Algorithms and data structures are two main components of a problem solution
- An algorithm is a step-by-step procedure for performing a task within the specified time
- A data structure is a way to store and organize data
- A data structure's major operations include insertion, modification, deletion, search, sort, merge, and traversal of data.

- Abstract Data Structure (ADT) represents the type of data and the operations without revealing the way of implementation
- Data structures are of linear types and non-linear types
- In a linear data structure, data is arranged in a linear or sequential order whereas, in a non-linear data structure, data is stored in no-sequential order

▼ Linear Data Structures - Arrays and Linked Lists

There are two types of Linear Data Structures which are

- Static Data Structure
  - Array
- Dynamic Data Structure
  - LinkedList
  - Queue
  - Stack

Array

- Data structure implementation **helps to store homogeneous data** elements in contiguous memory locations.
- It can be one dimensional or multidimensional implementation
- The biggest drawback of array implementation is the size of the array. Since the array is a static data structure, it will be a problem when we need to grow the size or if we are going to delete some elements from a particular array.

Linked List

- The primary advantage of a linked list is we are able to insert elements in any place within the chain
- The size is able to grow as the data come in
- It will be able to shrink itself as the data is getting deleted.

▼ Core Concepts

Linear List

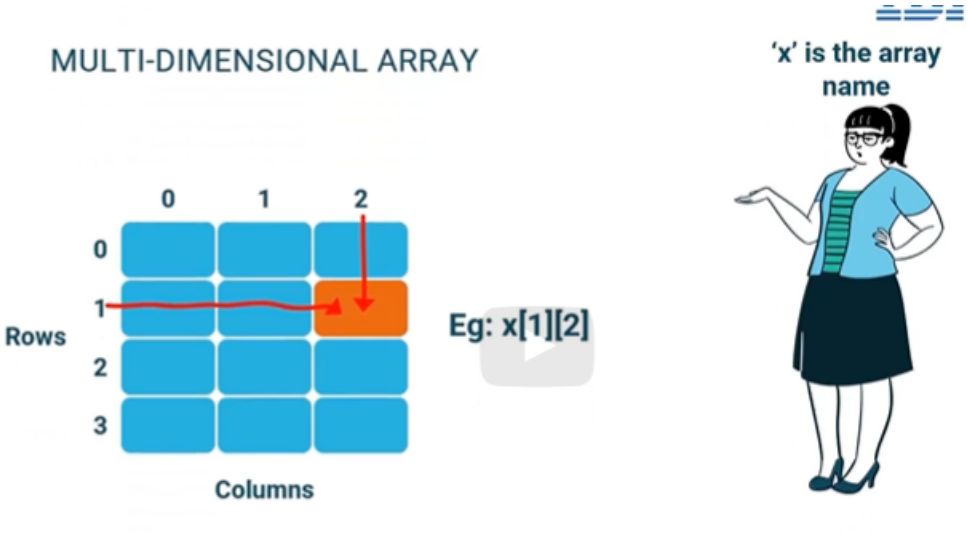
Generally, a list means a collection of sequentially connected items. For example, a to-do list, an item list, etc. In Computer science, a list is an ordered collection of values, items, entries, elements, etc. The implementation of a linear list can be Array or Linked List.

▼ Array

- An array is a type of linear data structure used to store homogeneous data in contiguous memory locations.
- It is something like a box with equally spaced slots into which data can be stored.
- Array have fixed sizes.
- Arrays can be classified as single-dimensional arrays and multi-dimensional arrays.

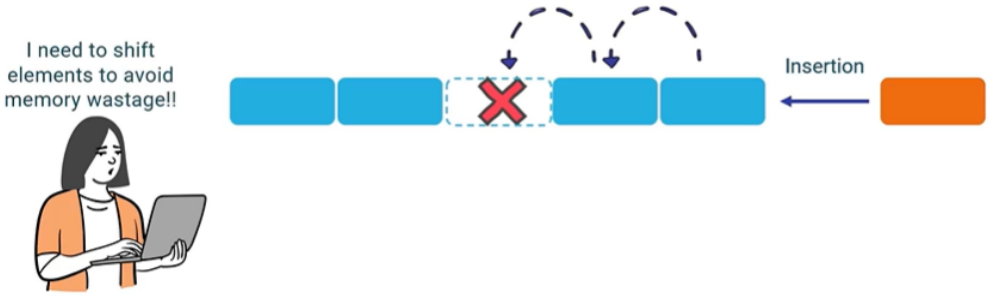
1. Single-Dimensional arrays

2. Multi-Dimensional Array

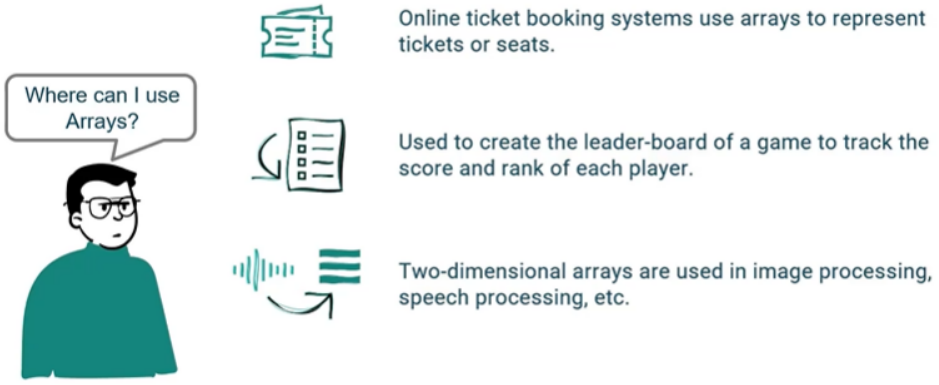


- When do we need arrays?
    1. When we need to index or access randomly more frequently.
    2. When memory is a concern because arrays use up less memory.
- Problems in Arrays
    1. Arrays have fixed size

    2. Complicated Insertion and Deletion



- Application of Array



- Implementation of Array using C++

One-Dimensional Array

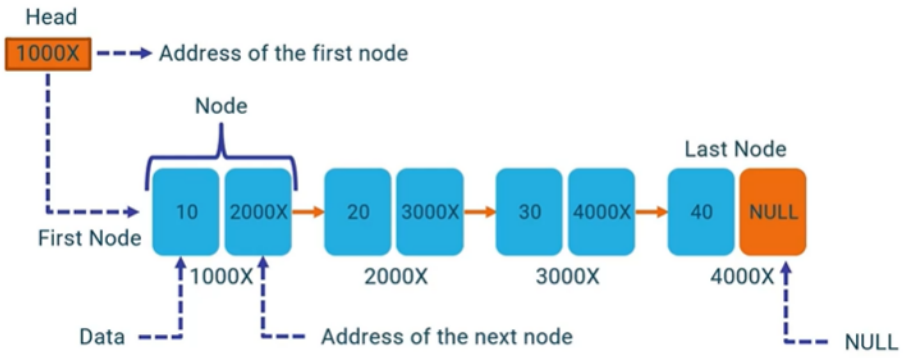
```
#include<iostream>
using namespace std;

int main()  {
  int arr[50],n,idx ;
  cout<<"Enter the size of array:"<<endl;
  cin>>n;
  cout<<"Enter the elements of array:"<<endl;
  for(idx=0;idx<n;idx++) {
     cin>>arr[idx];
  }
  cout<<"Array elements are:"<<endl;
  for(idx=0;idx<n;idx++) {
      cout<<arr[idx]<<" ";
  }
   return 0;
}
```

Two-Dimensional Array

```
#include<iostream>
using namespace std;
int main() {
  int arr[3][2];
  cout<<"Enter the elements of array:"<<endl;
  for(int row=0;row<3;row++) {
    for (int col = 0; col < 2;col++) {
      cin>>arr[row][col];
    } }

cout << "Elements in the array are:\n";
  for (int row = 0; row < 3; row++) {
    for (int col = 0; col < 2; col++) {
      cout <<arr[row][col]<<"  ";
    }  }
return 0;
}
```

▼ Linked List

- A linked list is a collection of elements called nodes that are arranged in a linear sequence.
- A linked list allocates space for each element separately in its own block of memory called a node.



Advantages of Linked List

1. It can grow or shrink dynamically to any size

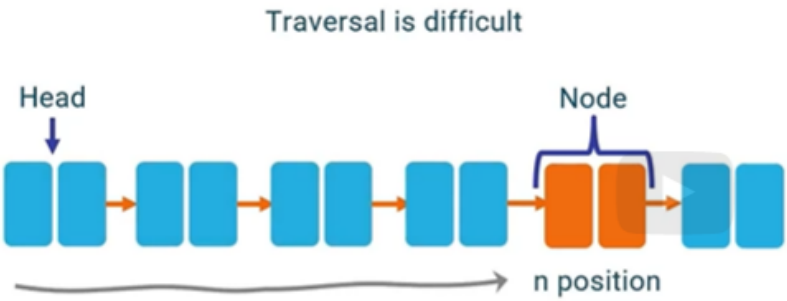
2. More EFFICIENT Insert and Delete operations

   It will be easy to insert or delete an element at the beginning, in the middle, or in the end of the list. Since this data structure is using memory address for connecting the elements.

Disadvantages

1. Traversal is difficult

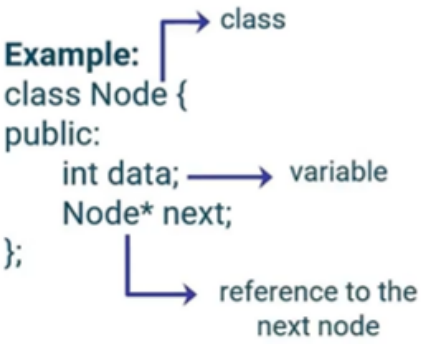
   There is no index in a linked list, which means we can not access the element randomly. If we want to traverse a node and be in the n position, then we have to traverse all nodes that come before n.

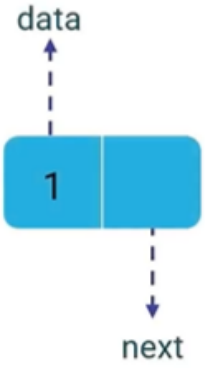


Linked List Implementation

A linked list node **may contain one or more data and a reference** to the next node. In this given example, a linked list node is implemented using a class called Node, which contains data as a member as a variable of integer type and next as a reference to the next node.

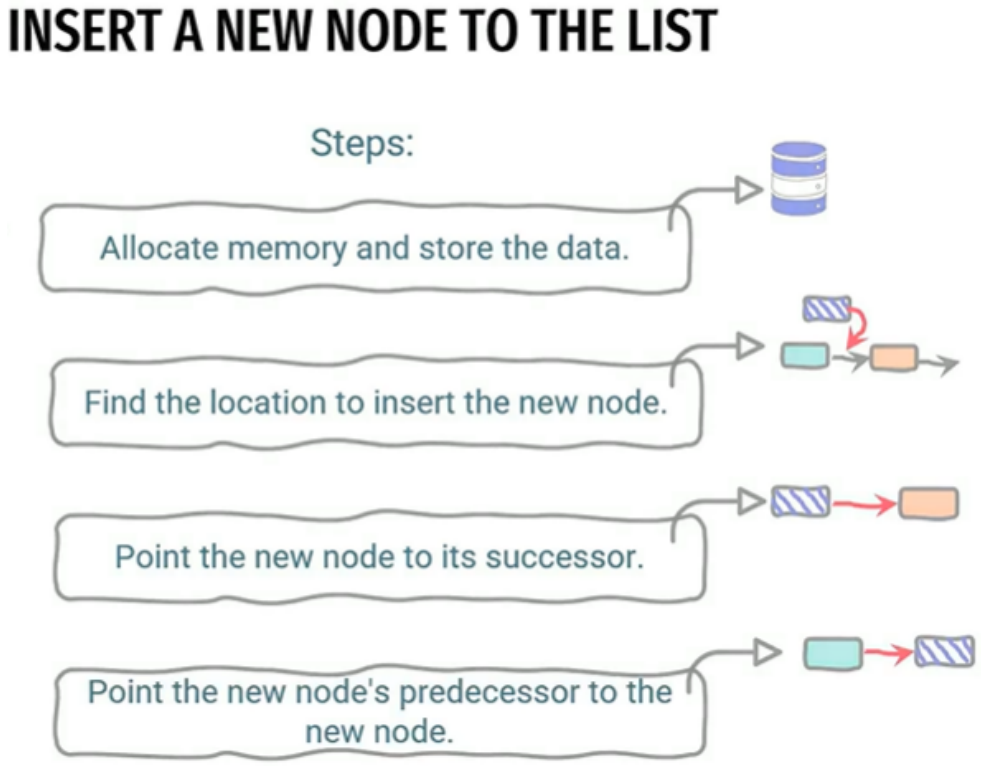
```
Syntax:
class class-name{
  variable declaration;
  reference to next object;
}
```





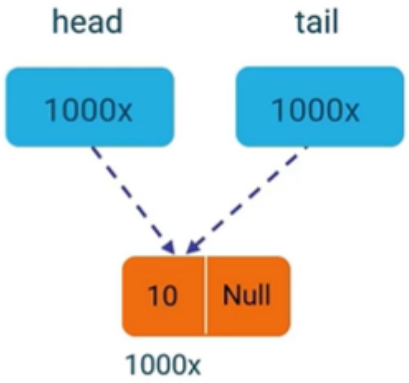
▼ **Operations on Linked List**

1. Insert

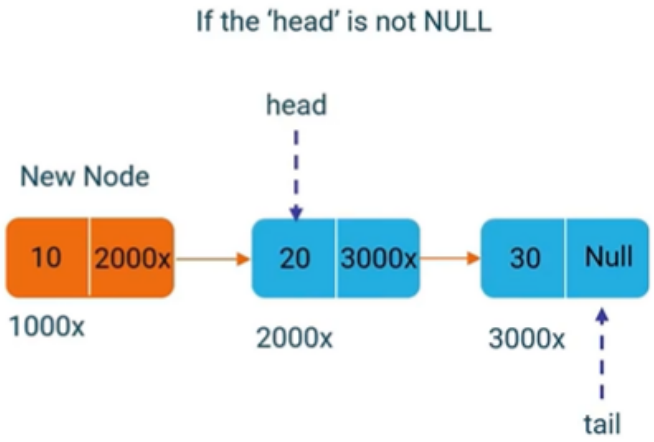


▼ **A). Insert At Front**

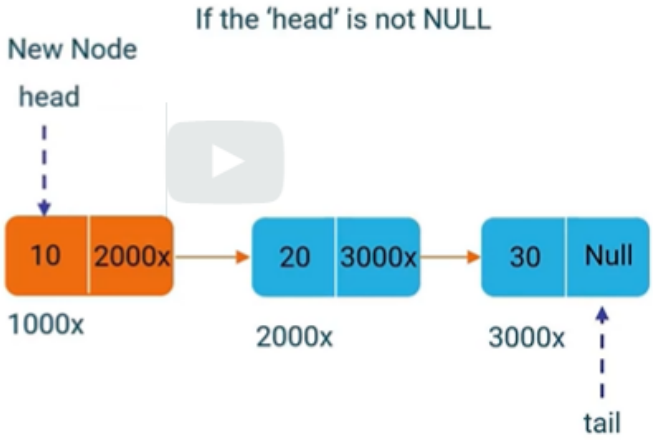
1. Create a new Node using the node class and store the value as its parameter to the ***data*** attribute and store NULL to the **next** attribute.
2. Check whether the head of the list is null or not
    - if it is null, set the new created as the head and the tail because this is the only element in the list



- if the head is not null, then store the head node in the next of the new node.



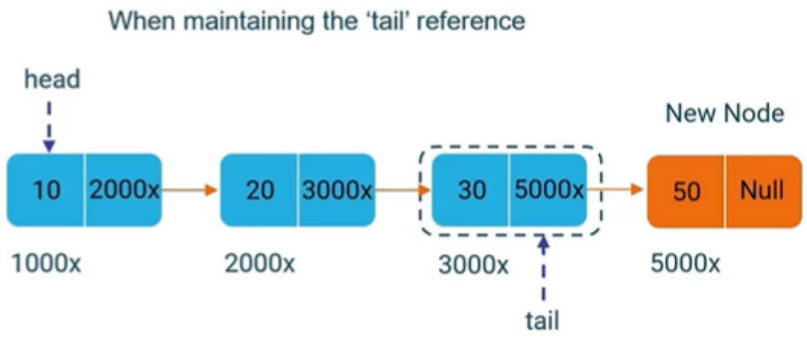
- then set the new node as the head of the list



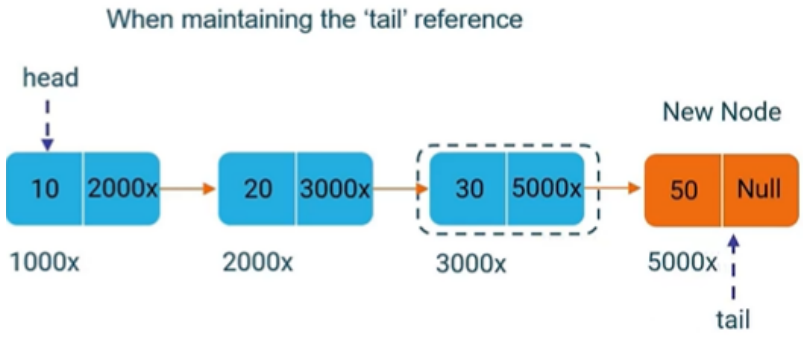
▼ **B). Insert a New Node at The End**

1. Create a new Node using the node class and store the value as its parameter to the ***data*** attribute and store NULL to the **next** attribute.
2. Check whether the head of the list is null or not
   - if it is null, set the new created as the head and the tail because this is the only element in the list (will be same with insert at the front)
   - if it is not:

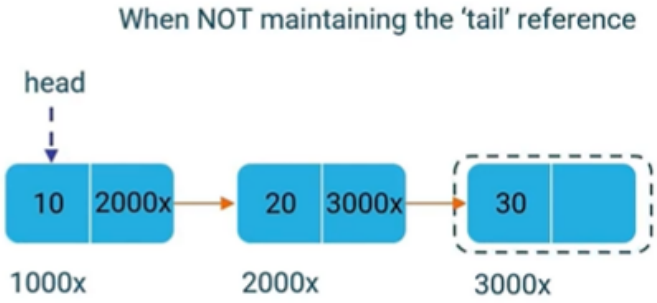
- if the list is maintaining the ‘tail’ reference:
    1. **Set the tail→next:newNode**



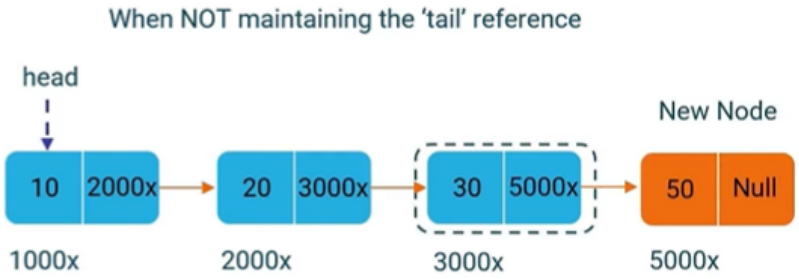
    2. **and tail = tail→next (tail=newNode)**



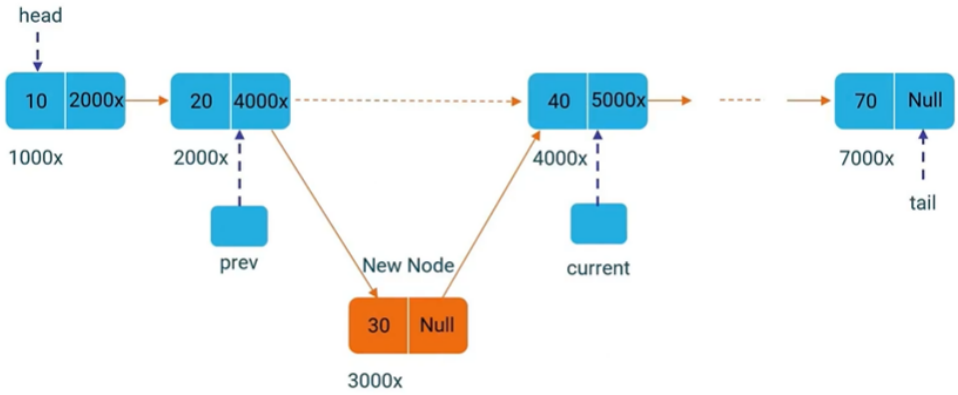
- if the list is not maintaining the ‘tail’ reference:
    1. iterate through the list till the last node



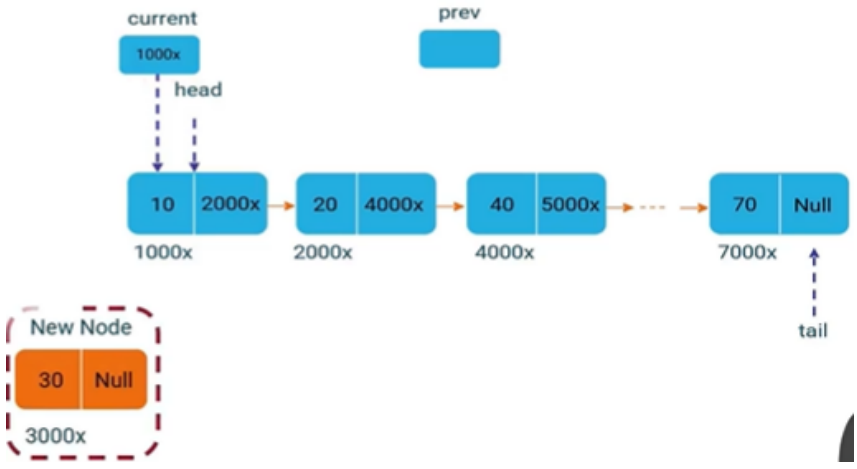
    2. Point the **next** of the **last node to the new node**



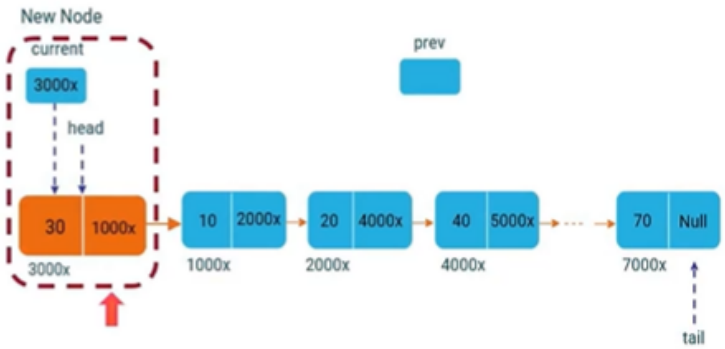
▼ **C). Insert a Node at a Specific Position**



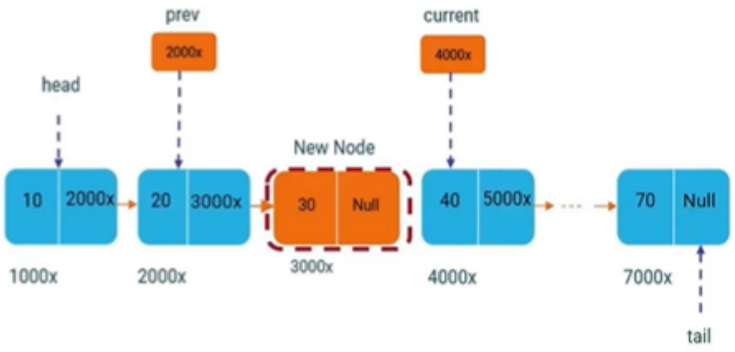
1. Declare two reference variables of node type **(prev and current)**
2. Set the head as the current reference variable
3. create the new node and set the value to the **data** attribute



4. Check the given position for its validity, and make sure it is not less than 1: if it's in less than one, it is invalid and the insert can not proceed.
5. Check whether the given position is one or not, if it is one, store the head in the new node's next variable and set the new node as head.



6. Otherwise, it has to traverse to the position in the list. The variable prev tracks the previous note and the current tracks the current node
   - When it reaches the specified position, the loop exists and stores the new node into the previous node's next and the current node into the new node's next respectively.

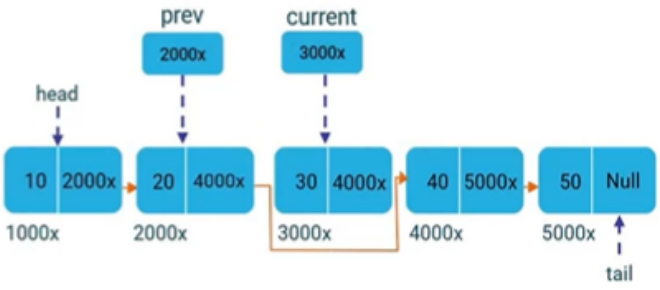


   - If the entered position is not found on the entire traversal of the list, display the appropriate message for the invalid position.

2. Search and Display Element

- To get a certain element in the list, we need to iterate each and every node and check the data part of each node for the search key value.
- Steps:
    1. Create a reference variable for starting the traversal from the first node.
    2. Create a loop for traversal until the current becomes null.
    3. If any node is equal to the search key value then stop the search, display the successful result and return from the function
    4. if not, move to the next node and continue the same process.

3. Delete
    - make a flag variable (boolean)
    - make **current** and **previous** variables for the reference variable
    - point the **current** and **previous** to **head**
    - iterate through the list element till the **current** is not null.
    - check if the current data is equal to the value and if it is also the head node
        - set the head to the current→ next
        - set **flag** to true
        - break the looping
    - check if the current data is equal to the value and if it is not the head
        - point the previous→next to the current→next
        - if (current is tail)
            - set tail as the previous
        - set **flag** to true
        - break the looping



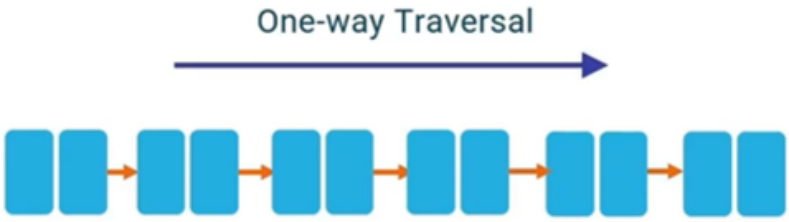
4. Modify

▼ **Linked List Variations**

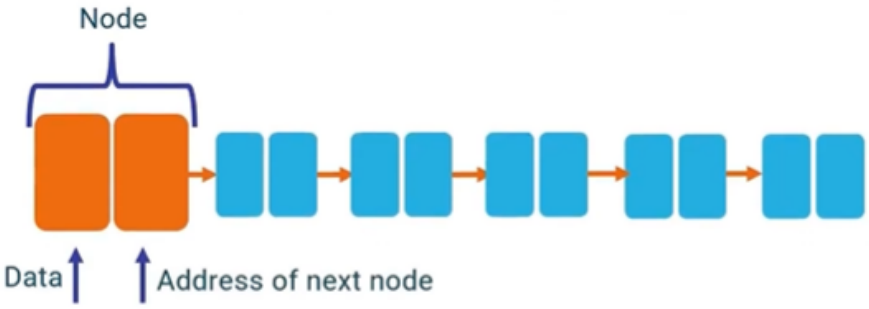
▼ 1. Singly Linked List

- It is a collection of nodes with a head and a tail

- Since all the nodes are linked together in a sequential way, only a one-way traversal is possible

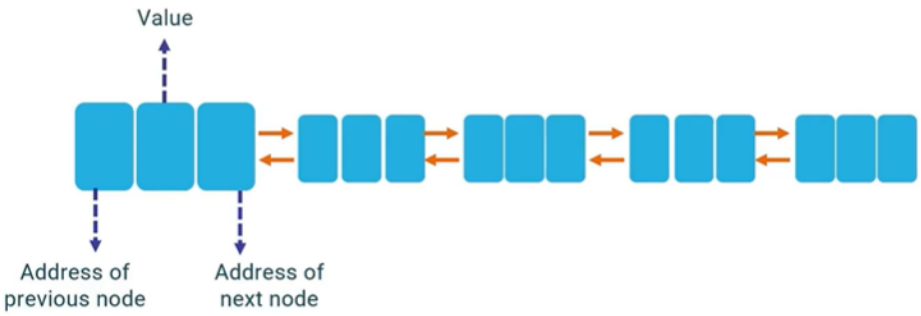


- Each node of the singly linked list contains one or more data fields and an address field that contains the reference to the next node.

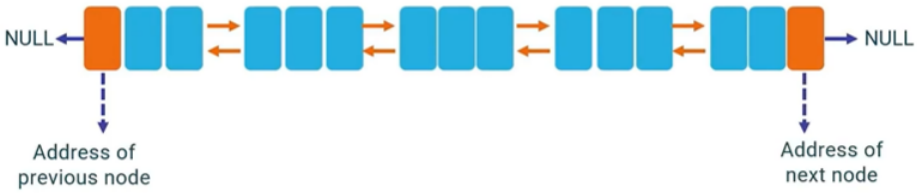


▼ 2. Doubly Linked List

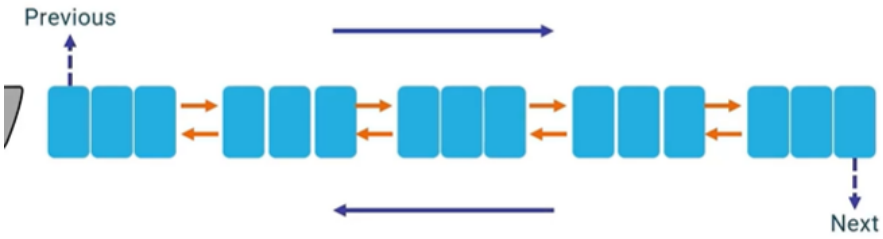
- Each node in a doubly linked list contains an extra memory to store the address of the previous node, together with the address of the next node and data.



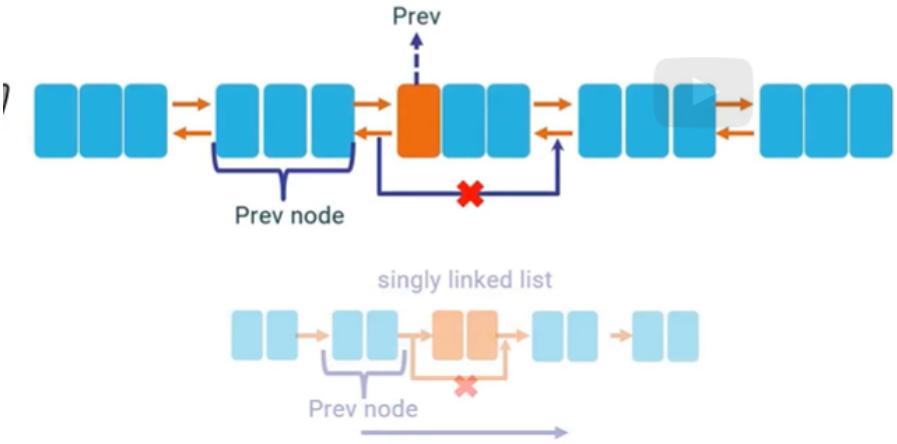
- There are two null values in a doubly linked list, they are located at the first node's previous field and the last node's next field



- Advantages of Doubly Linked List
  1. **Ease of traversal:** A doubly linked list can be traversed in both the forward and backward direction because it has both previous and next reference variables.

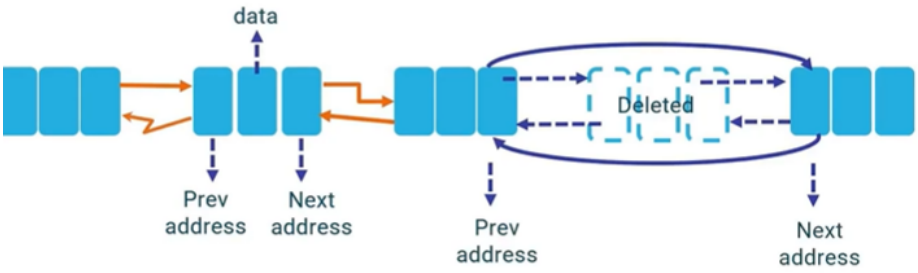


  2. **More Efficient Insert, Delete and Search operations:** It will be easier to delete, insert or search in a doubly linked list since we can get the previous node using the previous reference variable.



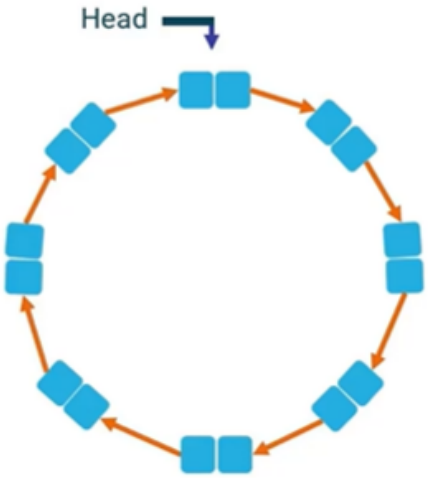
- Disadvantages of doubly linked list
    1. **More Memory Space:** A doubly linked list requires more space because here we need to store the address of the previous node along with the next node's address.

    2. **Deletion, Insertion - The number of modifications increased:** It is because we would need to modify the previous and next variables of the nodes

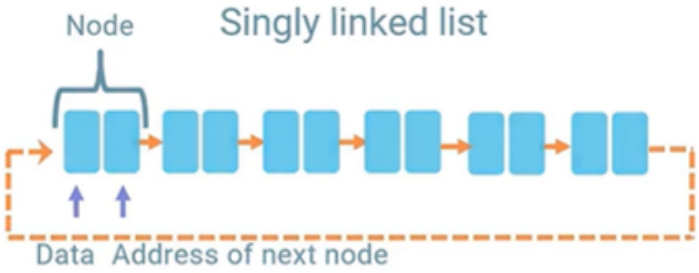


▼ 3. Circular Linked List

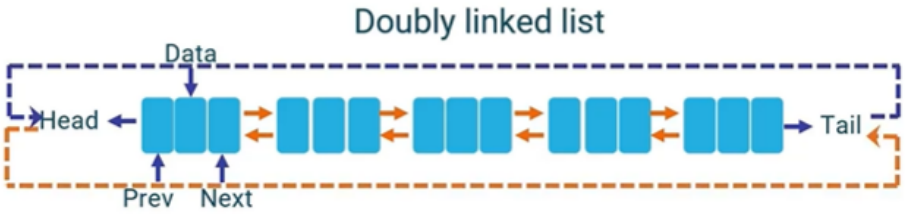
- A circular list is a list in which the link field that is the Next of the last node is made to refer to the first node of the list.



- A circular linked list does not contain null values.
- We can implement the Circular linked list either using A singly or doubly linked list.
- If we implement this circular linked list using a singly linked list, the next of the last node will contain the address of the first node.

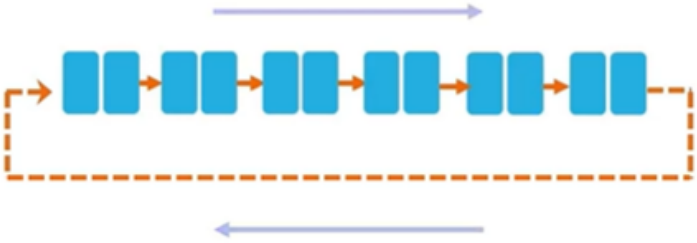


- If the implementation is done using a doubly linked list, the next of the last node will contain the address of the first node and the previous reference of the first node will contain the address of the last node.

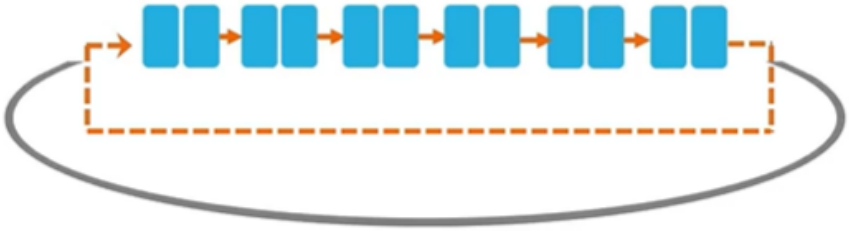


- Advantages of Circular Linked List

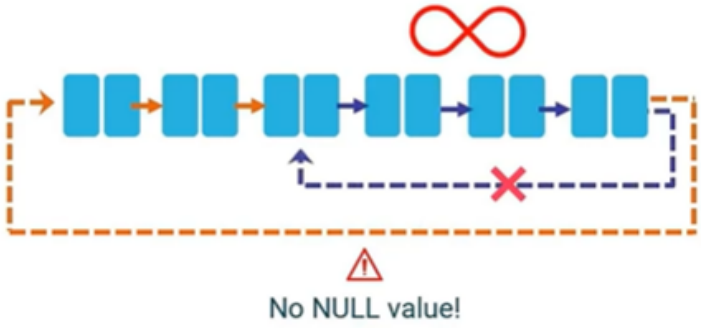
1. Easier traversal than Singly Linked Lists



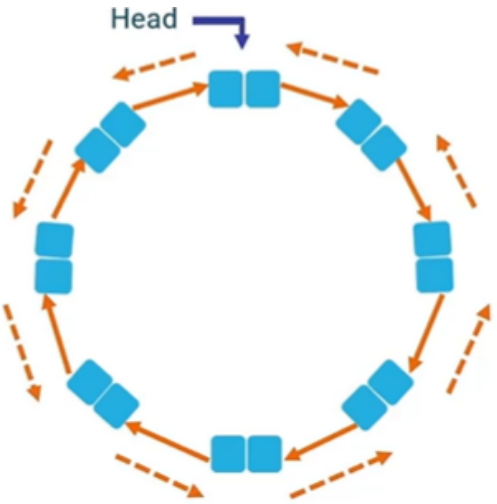
2. Useful when accessed in a loop



- Disadvantages
  1. It has a potential to be trapped in an infinite loop because we wouldn't have any null value to stop the traversal



  2. Operations such as reversing a circular linked list are complex as compared to doing the same in a singly linked list or a doubly linked list.



▼ **Applications of Linked List**

Not sure about the number of elements.

Used in browser cache with BACK-FORWARD visited pages, images, traverse browser history etc..

The undo and redo functionalities in Word, Paint, and other software.

To track various information in a circular fashion.
E.g.: To keep track of player turns in multi player games.

▼ **Summary**

1. A linear list is an ordered collection of values. Arrays and linear lists are linear lists.
2. An array is a collection of homogeneous elements stored continuously and its size is fixed.
3. Insertion and deletion operations are a bit difficult in arrays because of shifting elements.
4. A linked list is a collection of connected elements called nodes, arranged in a linear sequence.
5. The size of a linked list is not fixed. It can grow or shrink dynamically

6. A singly linked list stores data and the address of the next node.
7. A doubly linked list contains a memory area to store the data, the address of the previous node, and the address of the next node.
8. A circular list is a list in which the link field (next) of the last node refers to the first node of the list

▼ Demo

Chocolate Boxes

‘Camp Wonder’ is a summer camp program organized for kids for 5 days. The organizers have planned to conduct various games for the kids each day. On the last day, they planned a different game, where each kid is given an N number of boxes. Each box contains a bunch of chocolate. The kids will get the box one after another and whenever they get a box with an odd number of chocolates, then they have to destroy the first box they got. After getting all the boxes, they have to display the number of chocolates in each box they have. How will you implement the above scenario using a C++ data structure?

input Format

The first input is the number of boxes which should be >0 and ≤10

The second input is the number of chocolates in each box.

Output Format

The output should print the number of chocolates in the box available. If the number of boxes ≤0 or >10, then print ‘Invalid input’.

If the first box contains an odd number or negative value, then print ‘**Sorry!! The First box always should contain positive even no. of chocolates**

▼ Dynamic and Static Arry

1. When and where is a static array used?

   1). Storing and accessing sequential data.

   2). Temporarily storing objects

   3). Used by Io routines as buffers

   4). Lookup tables and inverse lookup tables

   5). Can be used to return multiple values from a function

6). Used in dynamic programming to cache answers to subproblems

Example of the static array:

A=[44,33,22,33,12,67,78,89,0]

2. Dynamic Arry → The dynamic array can grow and shrink in size

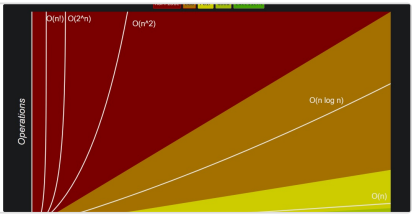
▼ Big O Notation

▼ What is Big O Notation

Silaturahmi Dengan Big O Notation

Big O Notation biasa digunakan untuk menghitung kompleksitas algoritma. Fungsinya untuk mengukur kompleksitas dari sebuah algoritma. Dalam pemrograman dan struktur data, ini sering dijadikan pedoman sebelum

https://ferry.vercel.app/blog/silaturahmi-dengan-big-o-notation



Time Complexity

▼ **O(1) Constant Time - Excellent**

The fastest algorithm. Regardless of how big the input is, the time for running is will be the same. Examples of O(1) complexity:

1. Find if a number is even or odd.
2. Check if an item on an array is null.
3. Find a value on a map
4. Access an element by index in an array.
5. and many similar things

▼ **O(n) Linear time**

Linear time complexity o(n) means that as the input grows, the algorithms take proportionally longer. A function with a linear time complexity has a growth rate. Linear time running implies visiting every element from the input in the worst-case scenario. Examples of O(n) linear times algorithms:

1. Get the max/ min value in an array.
2. Find a given element in a collection
3. Print all the values in a list.

▼ **O(log(n)) Logarithmic Time**

The logarithmic algorithm means the process is not as much as the input size or n. *when an algorithm has O(log n) running time, it means that as the input size grows, the number of operations grows very slowly.* ***Example: binary search***

▼ **O (n log(n) Linierithmic Algorithm**

Linearithmic time complexity it’s slightly slower than a linear algorithm but still much better than a quadratic algorithm. Example:

1. merge sort,
2. quicksort
3. and other efficient sorting algorithms

▼ **O(n^2) Quadratic**

Exponential (base 2) running time means that the calculations performed by an algorithm double every time as the input grows. Examples:

1. Power Set: finding all the subsets on a set.
2. bubble sort
3. selection sort
4. insertion sort.

▼ Course Introduction

▼ Material

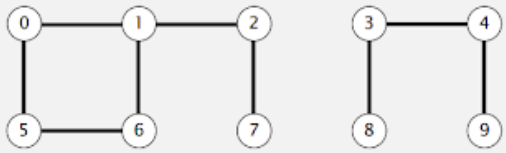
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/86cd4cbf-92d6-48d1-8a7d-aebeee7a6256/_b65e7611894ba175de27bd14793f894a_15UnionFind.pdf

- Algorithm → Method for solving problem
- Data Structure → Method to store information

▼ Union-Find

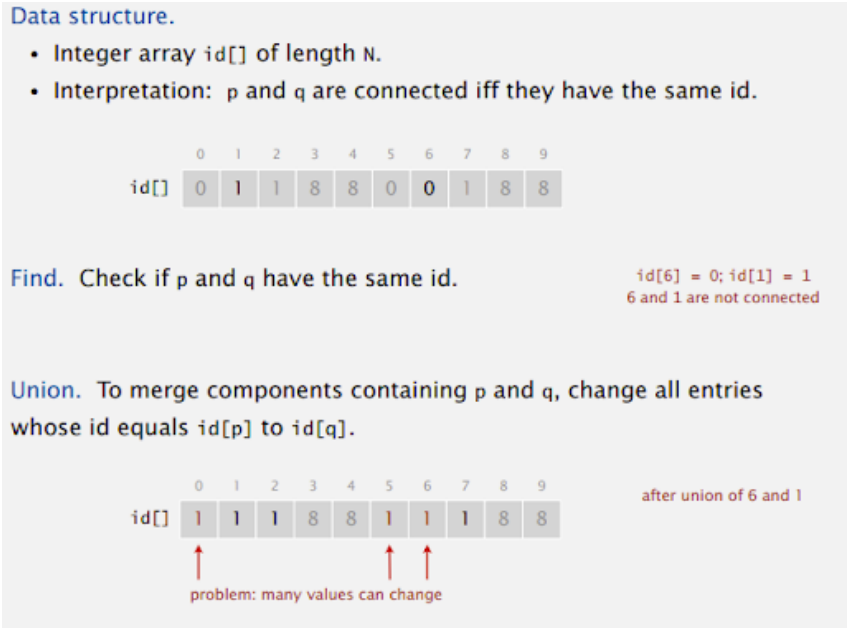
- Dynamic Connectivity
  1. Union → Conncect two objects
  2. Find/connected query → is thare a path connectiong the two objects.

```
union(4, 3)
union(3, 8)
union(6, 5)
union(9, 4)
union(2, 1)
connected(0, 7)  ×
connected(8, 9)  ✔
union(5, 0)
union(7, 2)
union(6, 1)
union(1, 0)
connected(0, 7)  ✔
```

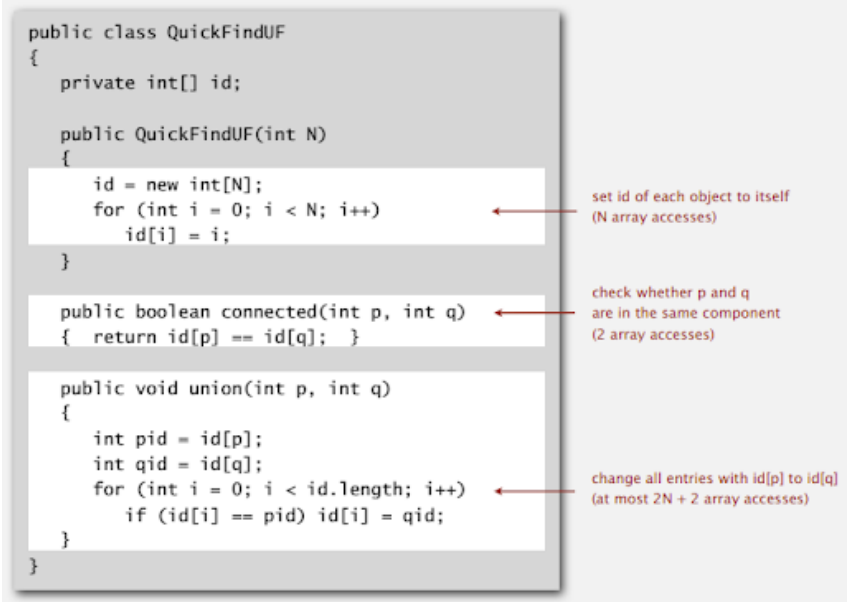


▼ Quick-Find

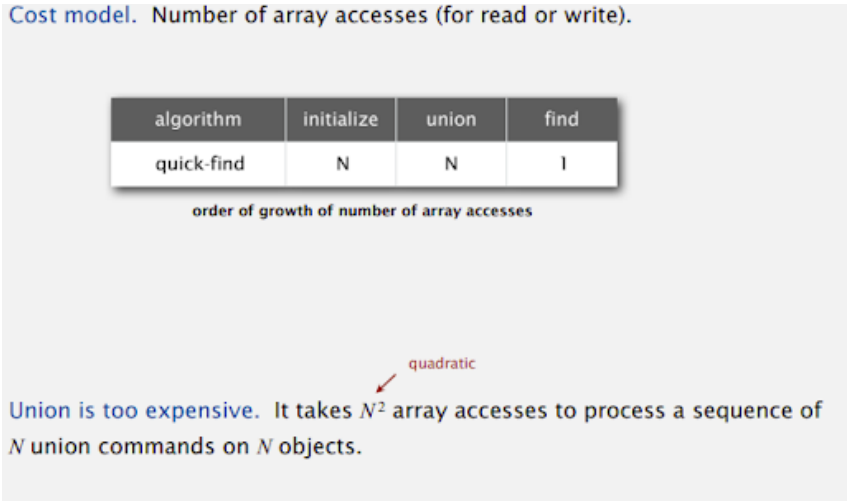
- Quick Find [eager approach]



- Quick Find Java Implementation

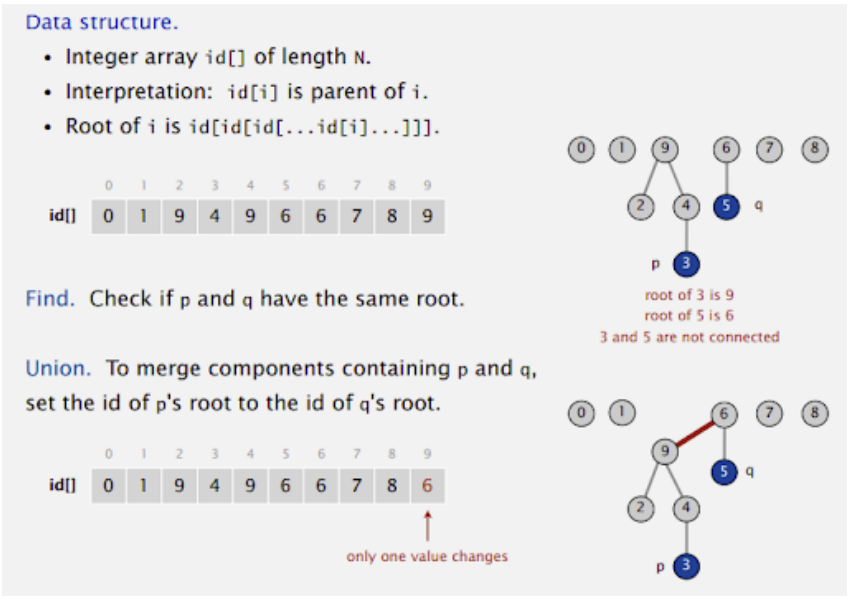


- Quick- Find is too Slow

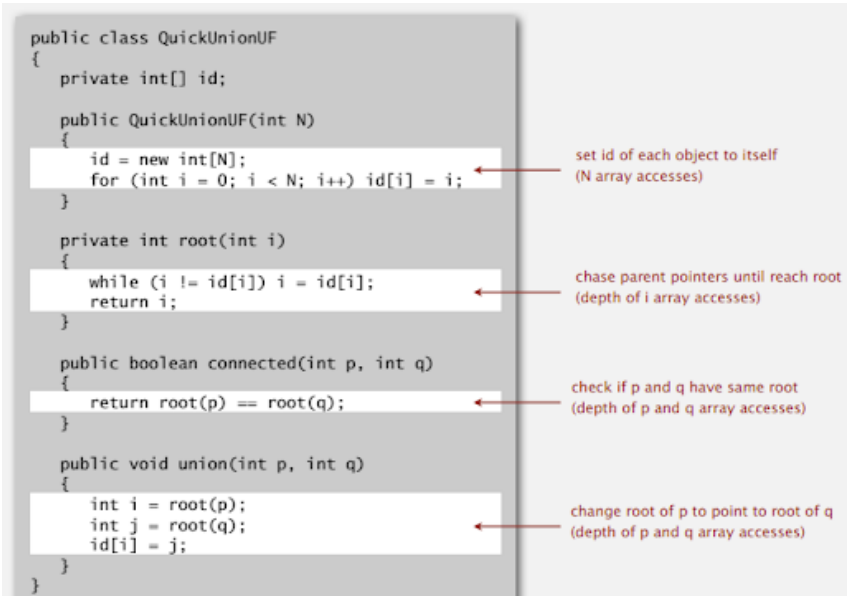


▼ Quick-Union

- lazy approach



- Java Implementation of Quick-Union



▼ Analysis of Algorithms

▼ Materials

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/3d405d7b-905a-41c7-9b14-e9f8c042e04e/_b65e7611894ba175de27bd14793f894a_15UnionFind.pdf

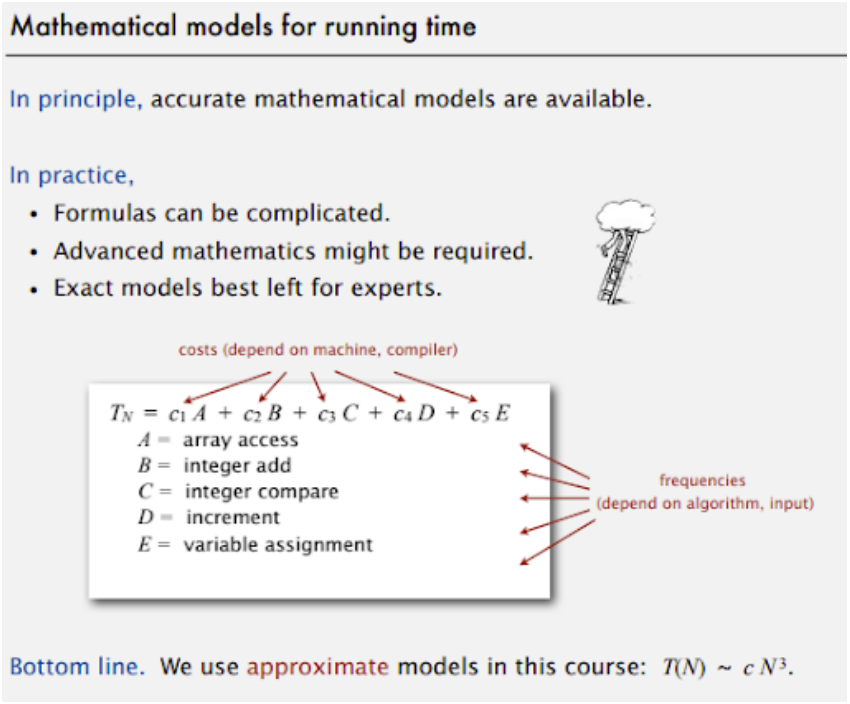
Reason to Analyze algorithms

- Predict performance
- Compare algorithms
- Provide guarantees
- Understand theoritical basis
- avoid performance bugs

Algorithm Analysis Method

▼ Observation

▼ Mathematical Models



▼ Order-of-Growth Classifications

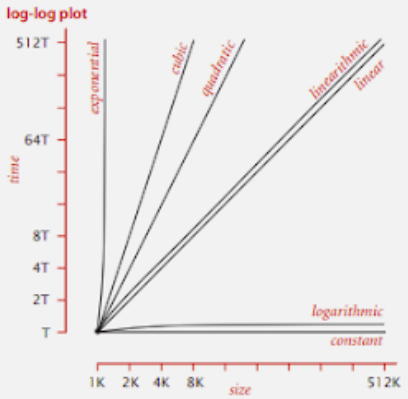
Common order-of-growth classifications

Good news. the small set of functions

order of growth discards leading coefficient

$1,\ \log N,\ N,\ N \log N,\ N^2,\ N^3,$ and 2^N

suffices to describe order-of-growth of typical algorithms.



Typical orders of growth

Common order-of-growth classifications

order of growth	name	typical code framework	description	example	T(2N) / T(N)
1	constant	`a = b + c;`	statement	add two numbers	1
log N	logarithmic	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search	~ 1
N	linear	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum	2
N log N	linearithmic	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	quadratic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }`	double loop	check all pairs	4
N^3	cubic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }`	triple loop	check all triples	8
2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	T(N)

Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ ⋮	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$ ⋮	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$ ⋮	develop lower bounds

▼ Memory

Typical memory usage for primitive types and arrays

type	bytes
boolean	1
byte	1
char	2
int	4
float	4
long	8
double	8

for primitive types

type	bytes
char[]	2N + 24
int[]	4N + 24
double[]	8N + 24

for one-dimensional arrays

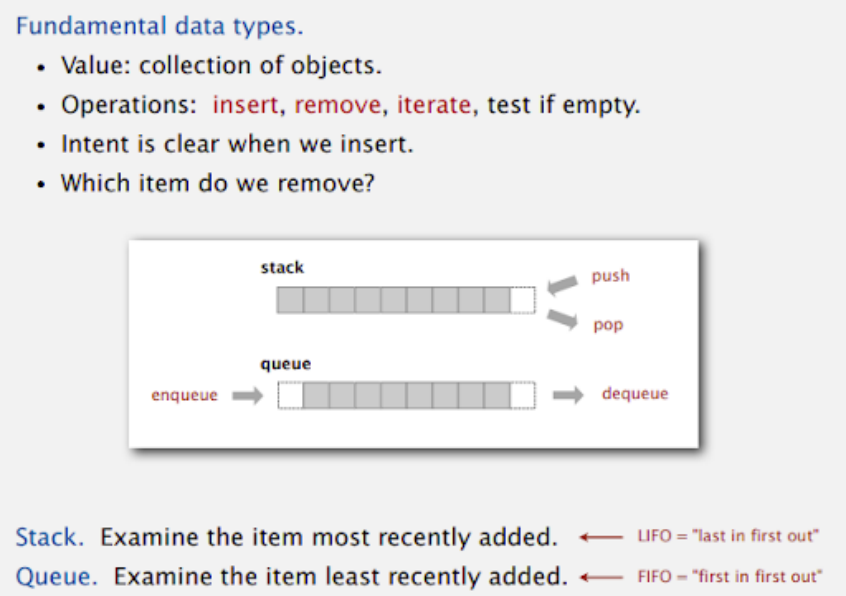
type	bytes
char[][]	~ 2 M N
int[][]	~ 4 M N
double[][]	~ 8 M N

for two-dimensional arrays

▼ Stack and Queues

▼ Stack API

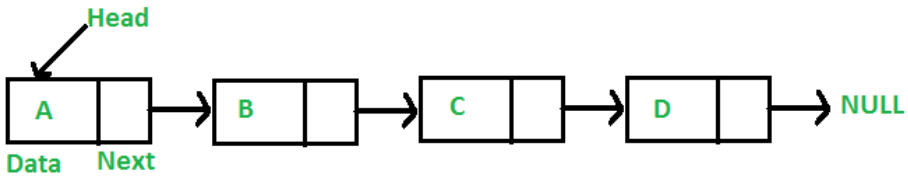
Stack and Queues



▼ Stack: linked-list Implementation in Java

▼ Pre-Materials (Fundamental of Linked List)

Unlike arrays, linked list elements are not stored at a contiguous location; elements are linked using pointers.



Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have the following limitations.

1) *The size of the arrays is fixed:* So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.

2) *Inserting a new element in an array of elements is expensive* because the room has to be created for the new elements and to create room existing elements have to be shifted.

For example, in a system, if we maintain a sorted list IDs in an array id[ ].

id[ ]=[1000, 1010, 1050, 2000, 2040]

and we want ot insert a new ID 1005, then to maintain the shorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[ ], everything after 1010 has to be moved.

Advantages of Linked List over arrays:

1. Dynamic size
2. Ease of insertion/deletion

Drawbacks of LinkedList

- Random access is not allowed. We nedd to access elements sequentially starting form the first node. We can no do binary search with linkedlist efficiently with default implementation.
- Extra memory space for a pointer is required with each element of the list.
- Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Representation:

A linked list is representated by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head is NULL.

Each node in a list consist of at least two parts:

1) Data

2) Pointer (Or Reference) to the next node.

In Java or C#, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

▼ Stack pop : linked-list implementation

inner class

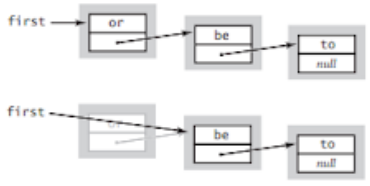
```
private class Node
{
   String item;
   Node next;
}
```

save item to return

```
String item = first.item;
```

delete first node

```
first = first.next;
```



return saved item

```
return item;
```

▼ Stack push: linked-list Implementation

inner class

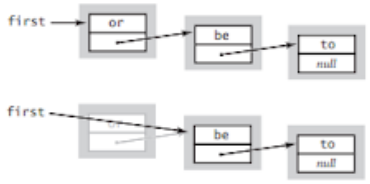
```
private class Node
{
   String item;
   Node next;
}
```

save item to return

```
String item = first.item;
```

delete first node

```
first = first.next;
```



return saved item

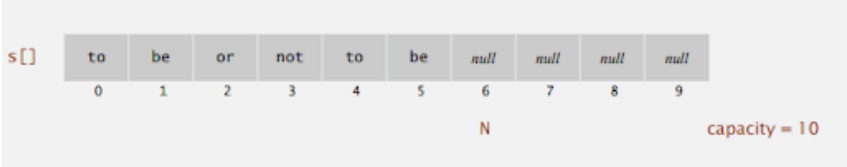
```
return item;
```

```
public class LinkedStackOfStrings {
    private Node first=null;
    private class Node{
        String item;
        Node next;
    }
    public boolean isEmpty(){
        return  first==null;
    }
    public void push (String item){
        Node oldfirst= first;
        first = new Node();
```

```
        first.item=item;
        first.next=oldfirst;
    }
    public String pop(){
        String item = first.item;
        first= first.next;
        return  item;
    }
}

public class Main {
    public static void main(String[] args) {
        LinkedStackOfStrings  linklist= new LinkedStackOfStrings();
        System.out.println(linklist.isEmpty());
        linklist.push("Mario");
        linklist.push("Tiara");
        System.out.println(linklist.pop());
    }
}
//true
//Tiara
```

▼ Stack: Array Implementation

- Use array s[] to store N items on stack.
- push () : add new item at s[N]
- pop(): remove item from s[N-1]




```
public class FixedCapacityStackOfStrings {
    private  String [] s;
    private  int N=0;
    public  FixedCapacityStackOfStrings(int capacity){
        s = new String[capacity];
    }

    public  boolean isEmpty(){
        return N == 0;
    }

    public void push (String item){
        s[N++]=item;
    }

    public String pop(){
        return s[--N];
    }
}
```

▼ Resizing Arrays

Stack : resizing-array implementation

1. How to grow an Array

   → If array is full, create a new array of twice the size, and copy items

```
public class ResizingArrayStackOfString {
    private  String [] s ;
    private  int N=0;
    public  ResizingArrayStackOfString(){
        s = new String[1];
    }

    public void push ( String item){
        if (N==s.length)
            resize(2* s.length);
        s[N++]= item;
    }

    private  void resize ( int capacity){
        String [] copy = new String[capacity];
        for (int i=0;i<N; i++){
            copy[i]= s[i];
        }
        s= copy;
    }
}
```

2. How to shrink an Array

   push (): double size array s[] when array is full

   pop() : halve size of array s[] when array is one-quarter full

```
public String pop(){
        String item = s[--N];
        s[N]=null;
        if (N> 0 && N==s.length/4)
            resize(s.length/2);
        return  item;

    }
```

Stack implementations: resizing array vs. linked list

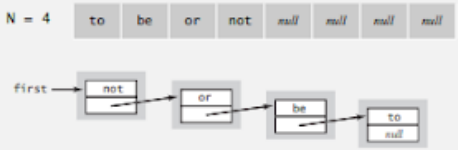
Tradeoffs. Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

Linked-list implementation.

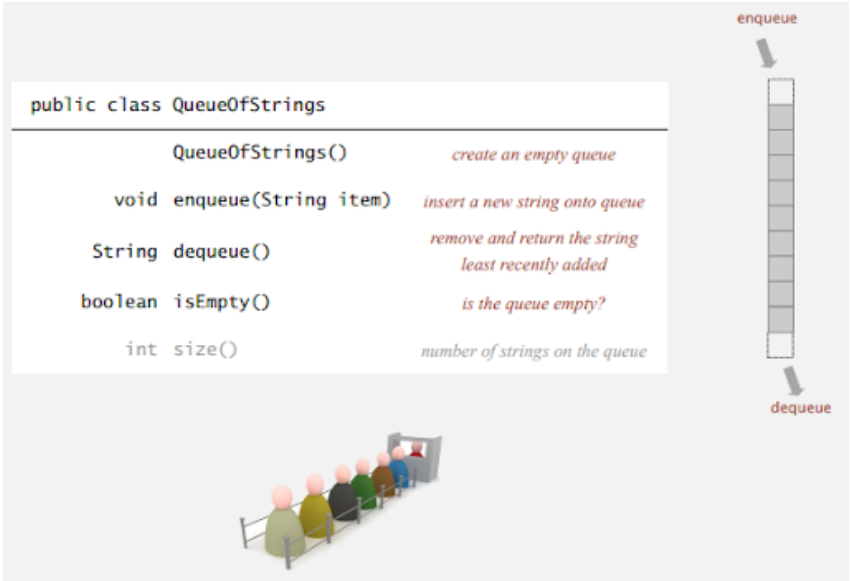
- Every operation takes constant time in the worst case.
- Uses extra time and space to deal with the links.

Resizing-array implementation.

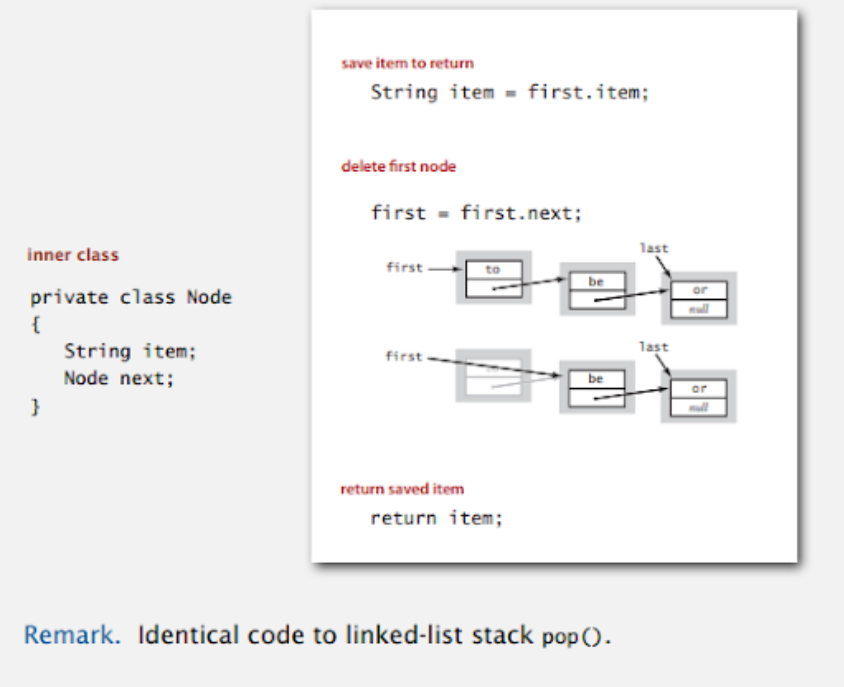
- Every operation takes constant amortized time.
- Less wasted space.



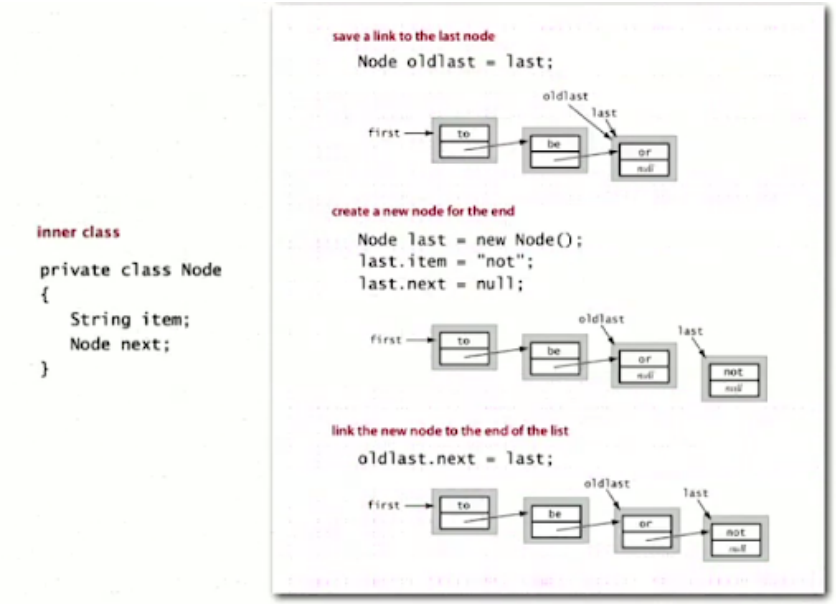
▼ Queue API



▼ Queue dequeue: linked-list implementation



▼ Queue enqueue: linked-list implementation



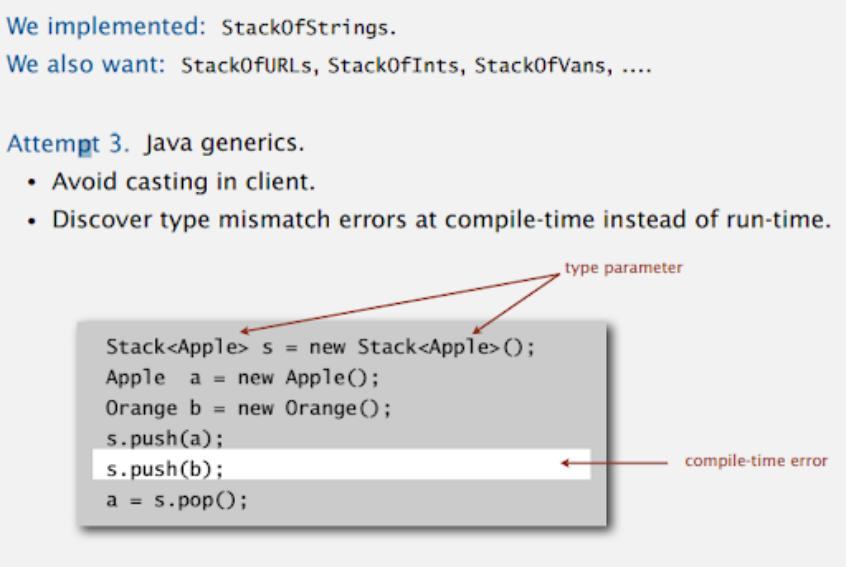
▼ Queue: linked-list implementation in Java

```
public class LinkedQueueOfStrings {
    private Node first, last;
    private  class Node{
        String item;
       Node next;
    }
```

```
    public boolean isEmpty(){
        return first==null;
    }
    public void enqueue (String item){
        Node oldlast= last;
        last = new Node();
        last.item=item;
        last.next=null;
        if (isEmpty()) first=last;
        else            oldlast.next=last;
    }

    public String dequeue(){
        String item= first.item;
        first = first.next;
        if (isEmpty()) last= null;
        return item;
    }
}
```

▼ Generic

Parameteried stack

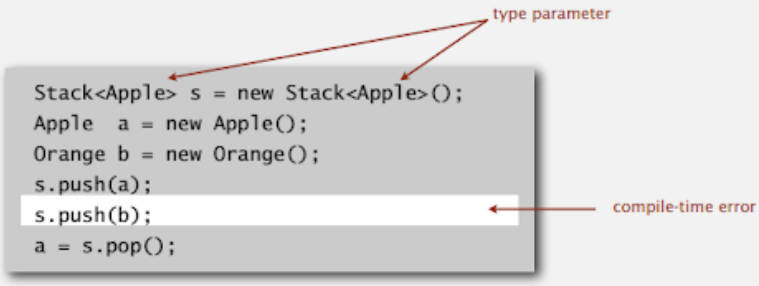


Generic stack: Array Implementation

We implemented: `StackOfStrings`.
We also want: `StackOfURLs`, `StackOfInts`, `StackOfVans`, ...

Attempt 3. Java generics.

- Avoid casting in client.
- Discover type mismatch errors at compile-time instead of run-time.



Generic Data Types : Autoboxing

Q. What to do about primitive types?

Wrapper type.

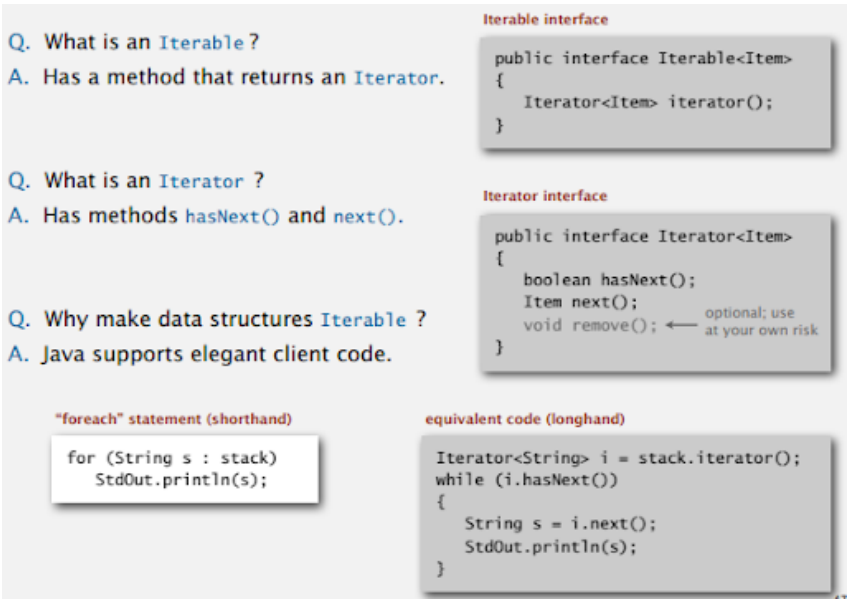
- Each primitive type has a wrapper object type.
- Ex: `Integer` is wrapper type for `int`.

Autoboxing. Automatic cast between a primitive type and its wrapper.

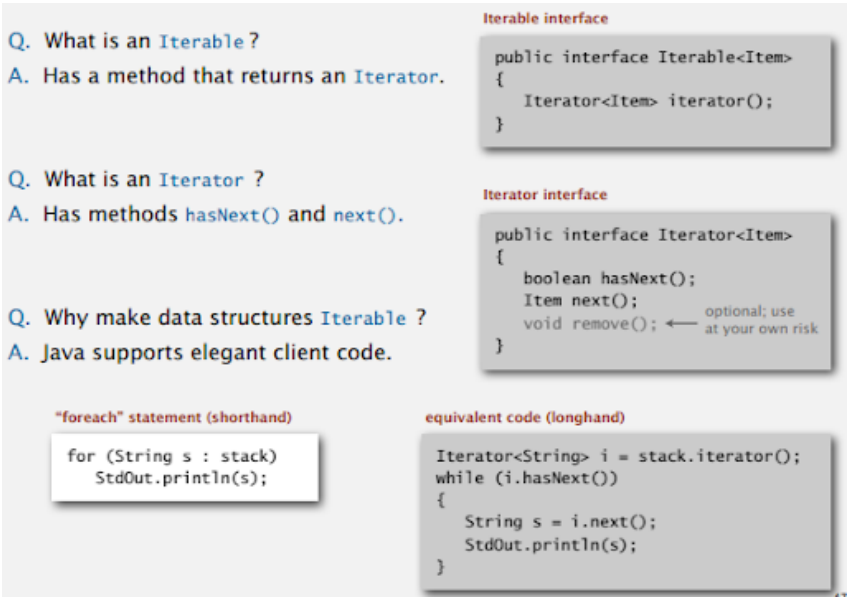
```
Stack<Integer> s = new Stack<Integer>();
s.push(17);        // s.push(Integer.valueOf(17));
int a = s.pop();   // int a = s.pop().intValue();
```

▼ Iterators

▼ **Definition**



▼ **Stack iterator: linked-list implementation**



▼ **Stack Iterator: Array implementation**

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
   …

   public Iterator<Item> iterator()
   { return new ReverseArrayIterator(); }

   private class ReverseArrayIterator implements Iterator<Item>
   {
      private int i = N;

      public boolean hasNext() {  return i > 0;        }
      public void remove()     {  /* not supported */  }
      public Item next()       {  return s[--i];       }
   }
}
```

▼ Stack and Queue Applications

▼ Java Collections Library

Java actually has

List interface. java.util.List which is API for sequence of items. This API provide libararies like:

```
public interface List<Item> implements Iterable<Item>

                 List()                    create an empty list
      boolean    isEmpty()                 is the list empty?
          int    size()                    number of items
         void    add(Item item)            append item to the end
         Item    get(int index)            return item at given index
         Item    remove(int index)         return and delete item at given index
      boolean    contains(Item item)       does the list contain the given item?
 Iterator<Item>  iterator()                iterator over all items in the list
          ...
```

Implementations.

1. Java.util.ArrayList uses resizing array.
2. Java.util.LinkedList uses linked list.'

Java.util.stack

support push(), pop () and ietarion

Extends java.util.Vector, which impelements java.util.List interface, including, get() and remove.

> But we better to make our own stack, Queue and Bag implementations. It because API from java library has to broad or bloated functions. So it's not a good idea to have lots of operations in the same API. And We can't know much about the performance.

▼ Sorting

Two Classic sorting Algorithm

1. QuickSort → Java Sort for primitive types
2. MergeSort → Java Sort For objects.

▼ MergeSort

▼ Material

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/10e2171a-f823-44c0-9a02-c8db03e8761a/_96af337808d8f90939cd4d97c25898e4_22Mergesort.pdf

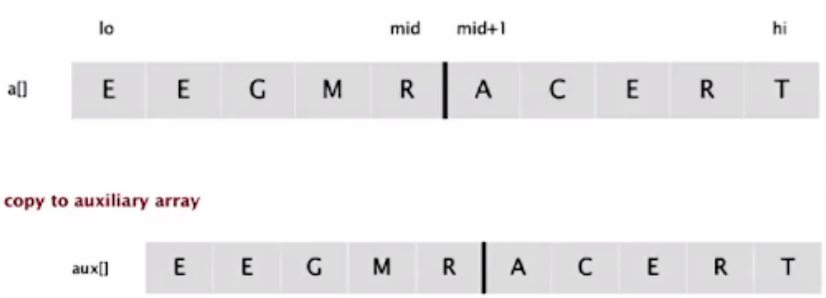
▼ Basic Plan of MergeSort

- Divide array into two helves
- Recursively sort each half.
- Merge two halves.

Mergesort overview

▼ Abstract in-place merge sort

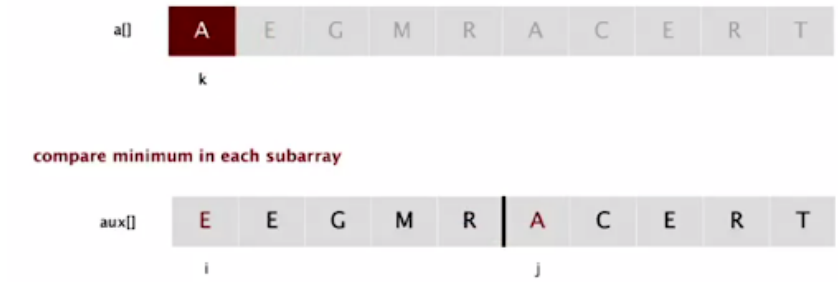
Goal: Given two sorted subbarrays a[low] to a [mid] and a[mid+1] to [high], replace with sorted subarray a[low] to [high]



i and j is the element that will be compered, and k is target element that will be replaced by comparison

▼ Loop 1:

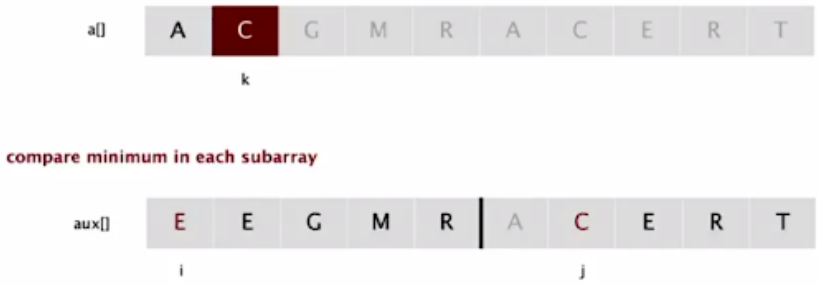
aux[i] comapre to aux[j] ⇒ aux[i]>aux[j] ⇒ a[k] replaces by aux[j]



▼ Loop 2:

Because of at the loop 1, aux [i] > aux [j] the, at this loop :

aux[i] compare to aux[j+1] ⇒ aux[i] > aux[j+1] ⇒ a[k+1] replaced by aux[j+1]



▼ Loop 3:

Because of at the loop 1, aux [i] > aux [j+1] :

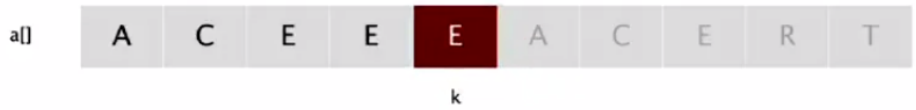
aux[i] compare to aux[j+2] ⇒ aux[i] == [j+] ⇒ a[k+2] replaced by aux[i]

▼ Loop 4:

aux[i+1] compare to aux[j+2] ⇒ aux[i+1] == aux[j+2] ⇒ a[k+3] replaces by aux[i+1]

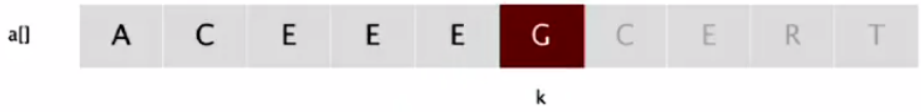
▼ Loop 5:

aux[i+2] compare to aux[j+2] ⇒ aux[i+2]> aux [j+2] ⇒ a[k+4] replaced by aux [j+2]



▼ Loop 6:

aux[i+2[compare to aux [j+3] ⇒ aux[i+2] < aux[j+3] ⇒ a[k+5] replaced by aux[i+2]

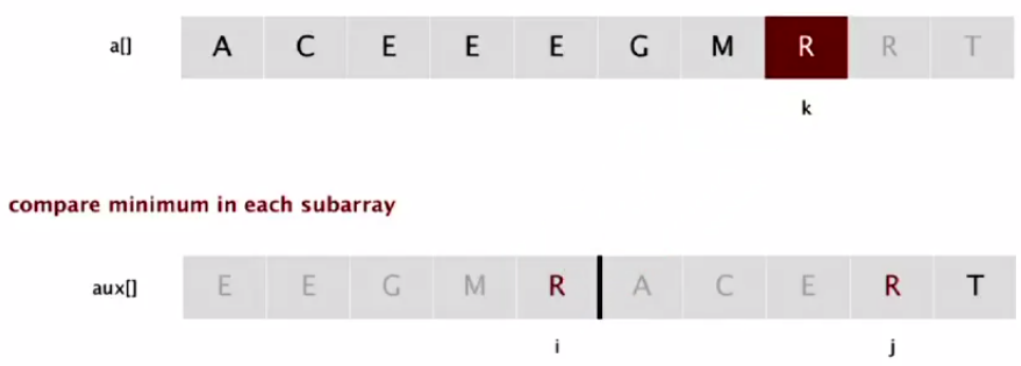


▼ Loop 7:

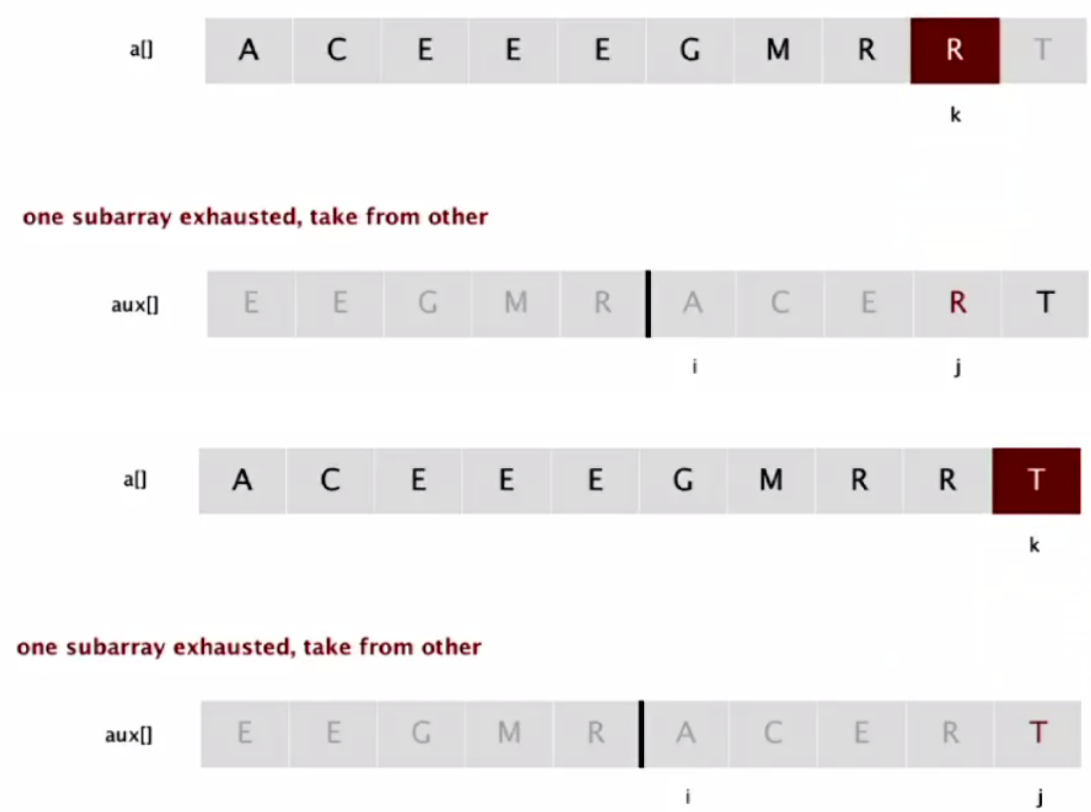
aux[i+3] compare to aux[j+3] ⇒ aux[i+3]<aux[j+3] ⇒ a[k+6] replaced by aux[i+3]

▼ Loop 8:

aux [i+4] compare to aux [j+3] ⇒ aux[i+4]==aux[j+3] ⇒ a[k+7] replaced by aux[i+4]



After loop 8, one subarray exhausted, take from other



▼ Java Implementation

```
package App;

import java.util.Arrays;
public class MergeSort {
    private  static  Comparable[] aux;
    private  static  void merge(Comparable [] a, Comparable [] aux, int lo, int mid, int hi){
        for (int k=lo; k<=hi; k++){
            aux[k]=a[k];
        }
```

```
        int i=lo, j=mid+1;
        for (int k=lo; k<=hi; k++){
            if (i>mid)      a[k]=aux[j++];
            else if (j>hi)  a[k]=aux[i++];
            else if (less(aux[j], aux[i])) a[k]=aux[j++];
            else                           a[k]=aux[i++];
        }

    }

    private static boolean less(Comparable aux, Comparable aux1) {
        return aux.compareTo(aux1) <0;
    }

    private  static void sort (Comparable[] a, Comparable[] aux, int lo, int hi){
        if (hi<=lo)     return;
        int mid= lo + (hi-lo)/2;
        sort(a, aux, lo, mid);
        sort(a, aux, mid+1, hi);
        merge(a,aux,lo,mid,hi);
    }

    public  static  void sort (Comparable[] a){
        aux= new Comparable [a.length];
        sort(a, aux, 0, a.length-1);

    }
}
```

Test:

```
public class MainApp {
    public static void main(String[] args) {
        Comparable [] a={4,6,8,3,1,5,7,9,0};
        MergeSort.sort(a);
        for (var i:a){
            System.out.print(i);
        }
    }
}

Output=> 013456789
```

▼ Mergesort empirical analysis

Running time estimates:

- Laptop executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort (N log N)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Bottom line. Good algorithms are better than supercomputers.

▼ Mergesort analysis: Memory

Running time estimates:

- Laptop executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort (N log N)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Bottom line. Good algorithms are better than supercomputers.

▼ MergeSort Practical improvements

▼ Use insertion for small subarrays

Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 7 items.

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo + CUTOFF - 1)
   {
      Insertion.sort(a, lo, hi);
      return;
   }
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   merge(a, aux, lo, mid, hi);
}
```

▼ Stop if already sorted

Stop if already sorted.

- Is biggest item in first half ≤ smallest item in second half?
- Helps for partially-ordered arrays.


```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   if (!less(a[mid+1], a[mid])) return;
   merge(a, aux, lo, mid, hi);
}
```

▼ Eliminate the copy to the auxiliary array

Eliminate the copy to the auxiliary array. Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
   {
      if      (i > mid)           aux[k] = a[j++];
      else if (j > hi)            aux[k] = a[i++];
      else if (less(a[j], a[i])) aux[k] = a[j++];
      else                        aux[k] = a[i++];
   }
}

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   sort (aux, a, lo, mid);
   sort (aux, a, mid+1, hi);
   merge(a, aux, lo, mid, hi);
}
```

merge from a[] to aux[]

Note: sort(a) initializes aux[] and sets aux[i] = a[i] for each i.

switch roles of aux[] and a[]

▼ QuickSort

▼ Material

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/e969983f-cc5f-428f-a8e9-74ed03327596/_b65e7611894ba175de27bd14793f894a_15UnionFind.pdf

https://www.youtube.com/watch?v=ZHVk2blR45Q

It is s sorting algorithm based on the divide and conquers approach where:

An array is divided into subarrays by selecting a pivot element (element selected from the array). There are many different versions of quickSort that pick pivot in different ways:

- Always pick the first element as a pivot.
- Always pick the last element as the pivot (implemented below)
- Pick a random element as a pivot.
- Pick median as a pivot.

The key process in quickSort is partition(). The target of partitions is:

1. Given an array and an element x of the array as a pivot.
2. Put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x.
3. Put all greater elements (greater than x) after x. All this should be done in linear time.
4. All this should be done in linear time.

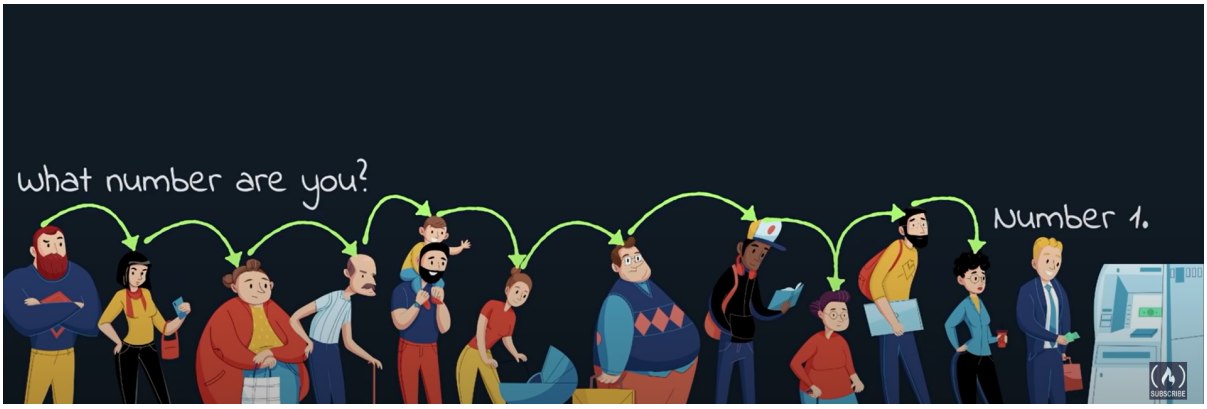
▼ **PseudoCode of QuickSort**

```
quickSort(arr []  low, high){
    if (low<high){
        pi=partition(arr, low, high);
        quickSort (arr, low,pi-1); //Before pi
        quickSort(arr, pi+1, high);
    }
}

partition (arr[], low, high){
    pivot= arr[high];
    i=(low-1);
    for (arr[j]< pivot){
        swap arr[i] and arr[j]
    }

    swap arr[i+1] and arr[high]
```

```
    return (i+1)
}
```

▼ Recursive

The best analogy for recursive is by imagining people who waiting in line to use an ATM. In this situation, the last person in ques asks “How many people are in front of me?” What this person will do is ask the next person on its next, then the person who asked will do the same to the next person. It will be repetitive until the question reaches the person who is the 1st in the queue.

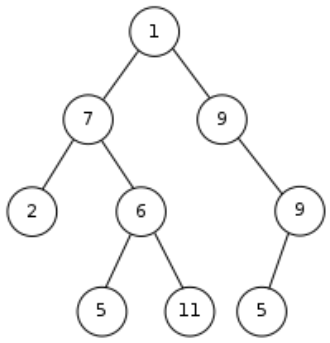


Then, when the first person in the queue gets the question, it will return 1. Then, the 2nd person in the queue will return 1 + 1, the 3rd will return 2+1, and so on.

▼ Trees

https://www.youtube.com/watch?v=fAAZixBzIAI&t=1698s

A binary tree is a tree **data structure in which each node has no more than two children,** which are referred to as the left child and the right child.



Traversing

There are two groups of traversing methods in Tree data structure which are **Depth First Search (DFS) and Breadth First Search (BFS)**

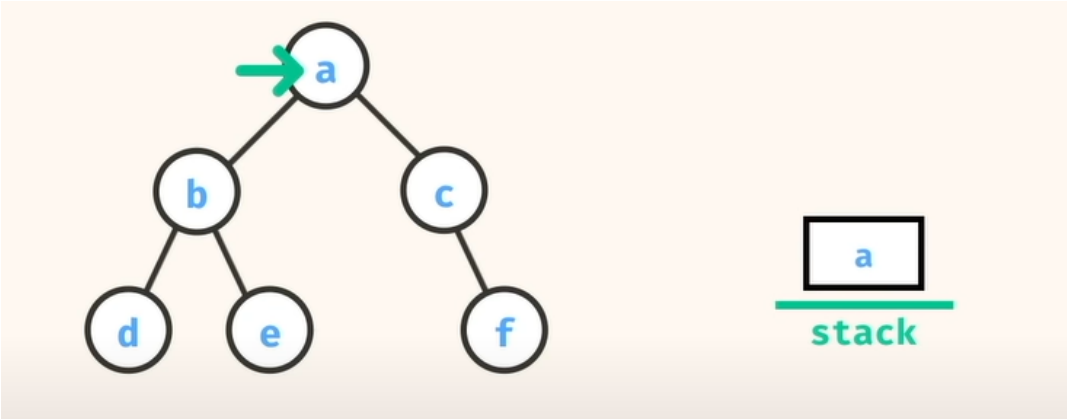
1. **Depth First Search (DFS)**

   The algorithm starts at the root node and explores as far as possible along each branch before backtracking. The purpose of the algorithm is to mark each vertex as visited while avoiding cycles. The DFS algorithm works as follows:

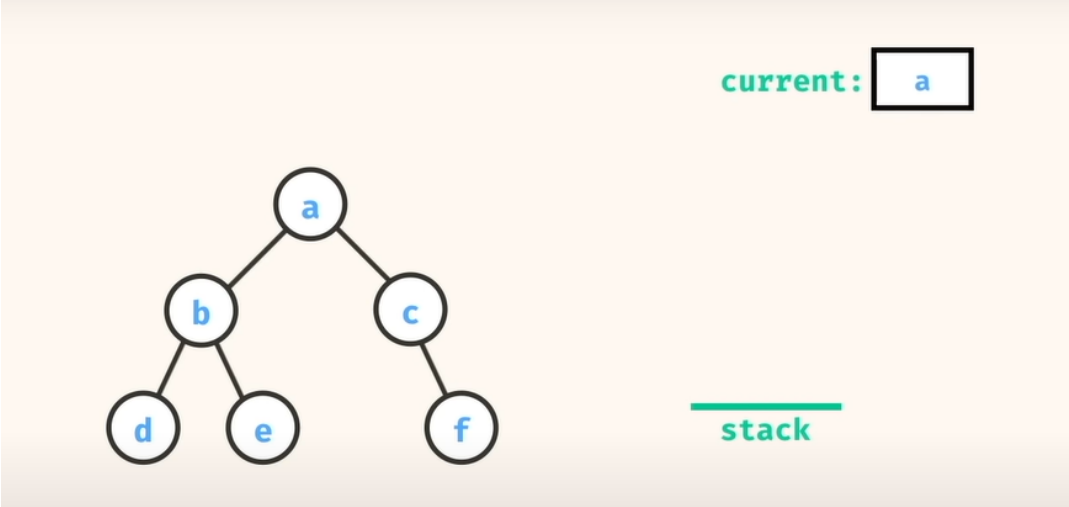
   1. Start by putting any one of the graph's vertices on top of a stack.
   2. Take the top item of the stack and add it to the visited list
   3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
   4. Keep repeating steps 2 and 3 until the stack is empty.

▼ How do implement the step in Programming?

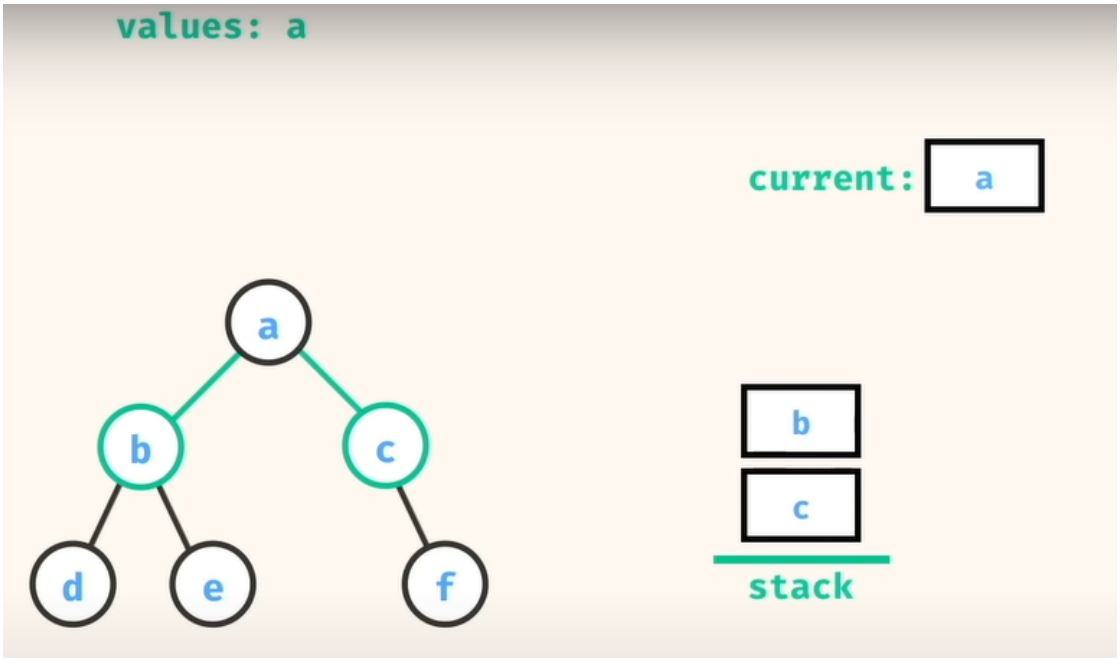
1. Start with the root node which is **a,** and store that node to stack.



2. Remove the a from the stack, and label it as the current node, and now a will be noted as the visited node

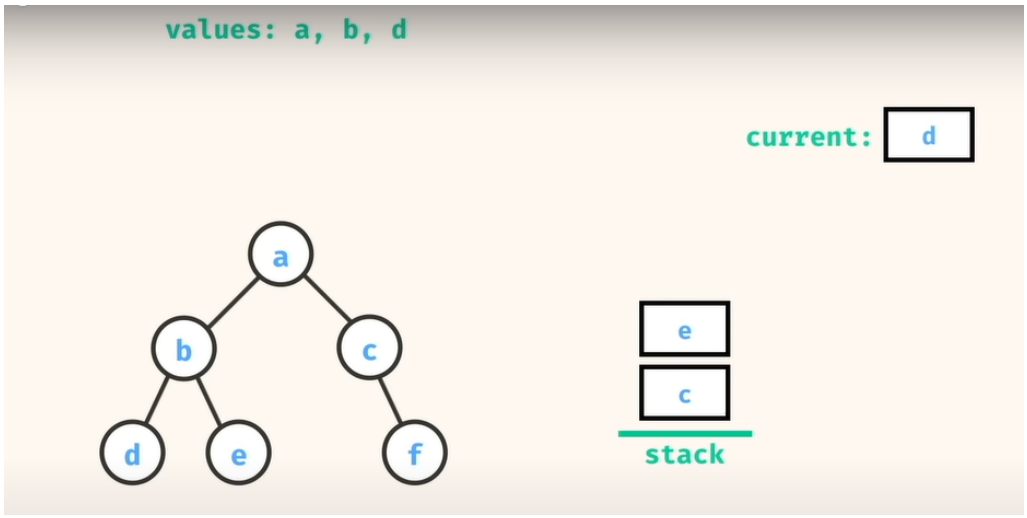


3. From step 2, we can check whether the **“a” node** has children or not. If it has, then we put the children into the stack, in this case, the children are **node c** and **node b.**

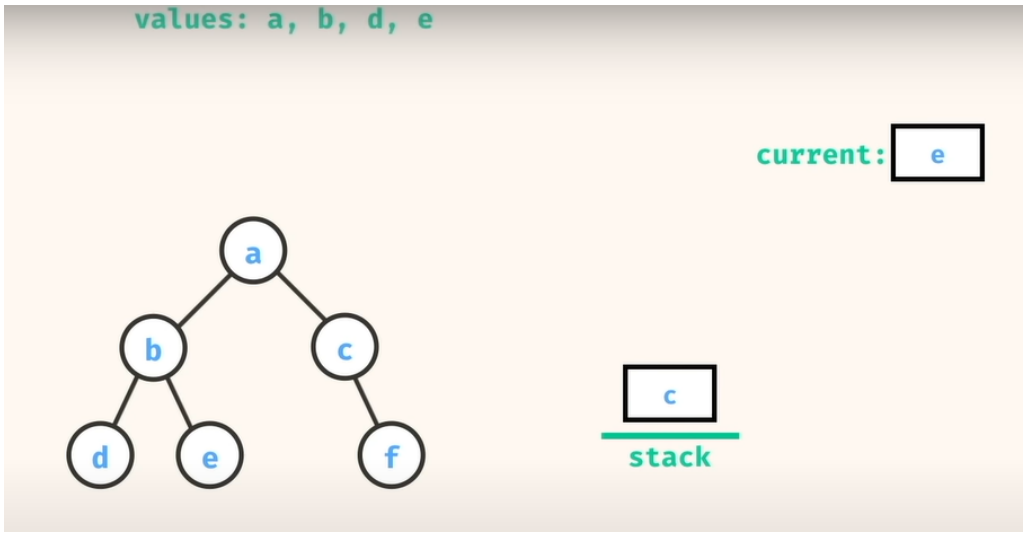


4. **Afterward**, the step will be repeated until the stack is empty.

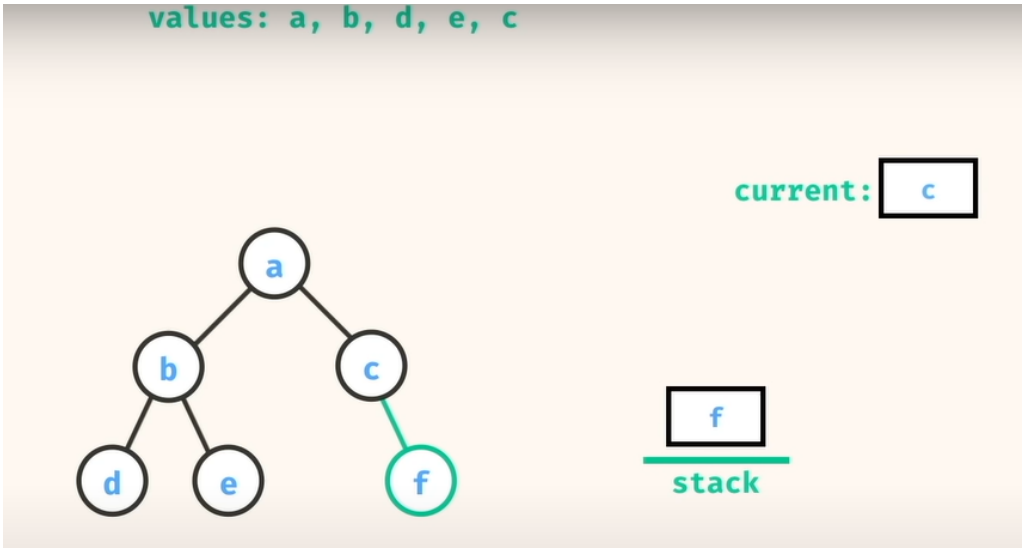
   ▼ Node d


values: a, b, d
current:
d
a
b
c
d
e
f
e
c
stack

▼ Node e

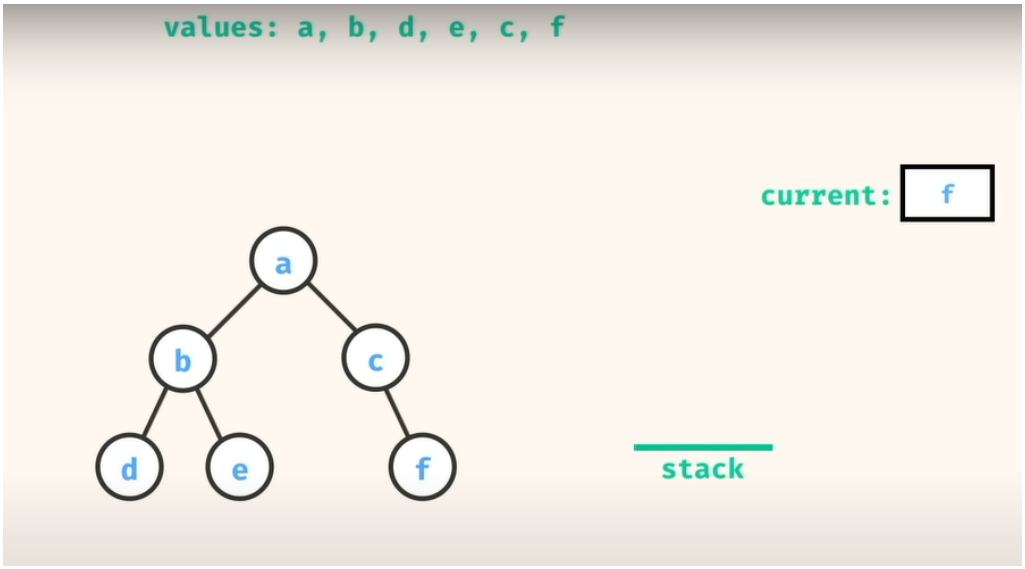

values: a, b, d, e
current:
e
a
b
c
d
e
f
c
stack

▼ Node c



▼ Node f

At this point, the stack will be empty because Node f doesn't have any children.



There are three types of DFS traversals in Binary Trees:

- **In-Order**

left then **root** then **right** ⇒ **A→B→C**

```
public void printInorder(){
    if (left!=null){
       left.printInorder();
    }
    Console.WriteLine(data);
    if (right!=null){
       right.printInorder();
    }
}
```

- **Pre-Order**

  root, then **left** then **right** ⇒ **B →A → C**

```
public void printPreOrder(){
    Console.WriteLine(data);
    if (left!=null){
       left.printPreOrder();
    }
    if (right!=null){
       right.printPreOrder();
    }
}
```

- **Post-Order**

  left then **right** then **root** ⇒ **A→C→ B**

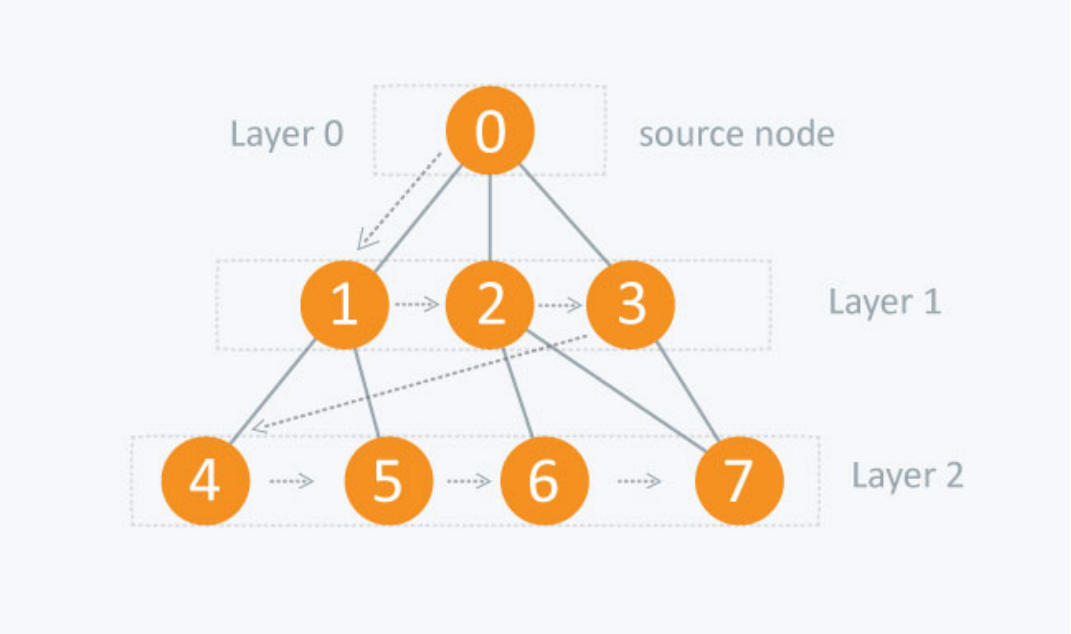
```
public void printPostOrder(){
    if (left!=null){
       left.printPostOrder();
    }
    if (right!=null){
       right.printPostOrder();
    }
    Console.WriteLine(data);
}
```

2. **Breadth First Search (BFS)**

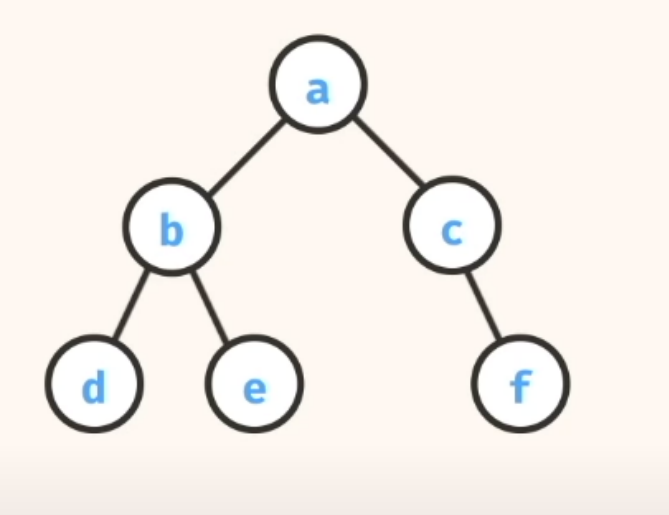
   BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbor nodes (nodes that are directly connected to the source node). We must then move toward the next-level neighbor nodes.

   Following steps for BFS

   1. First, move horizontal and visit all the nodes of the current layer
   2. move to the next layer

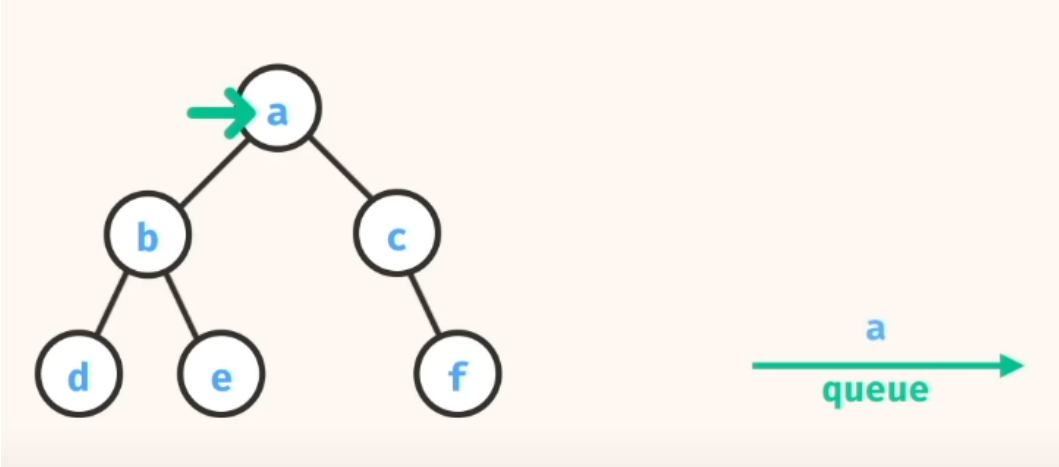


▼ **How do implement the step in Programming?**

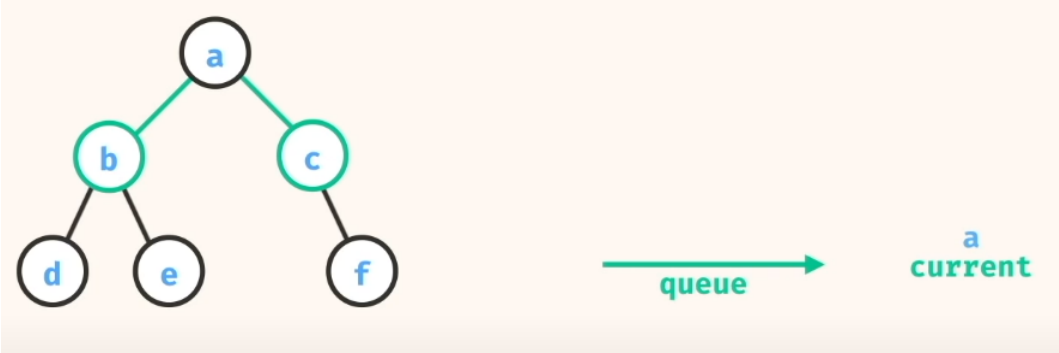


To implement BFS, we can use Que

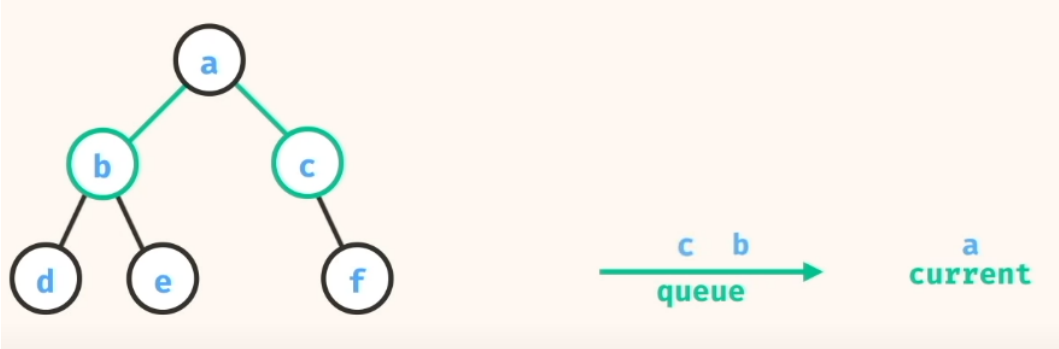
1. First, we insert the starting node into Que.



2. Afterward, enqueue the node as the current node.



3. Then, check the a’s children and insert them into the Queue



4. Repeat the steps until the queue is empty.

▼ **BFS Code**

1. **Iterative**

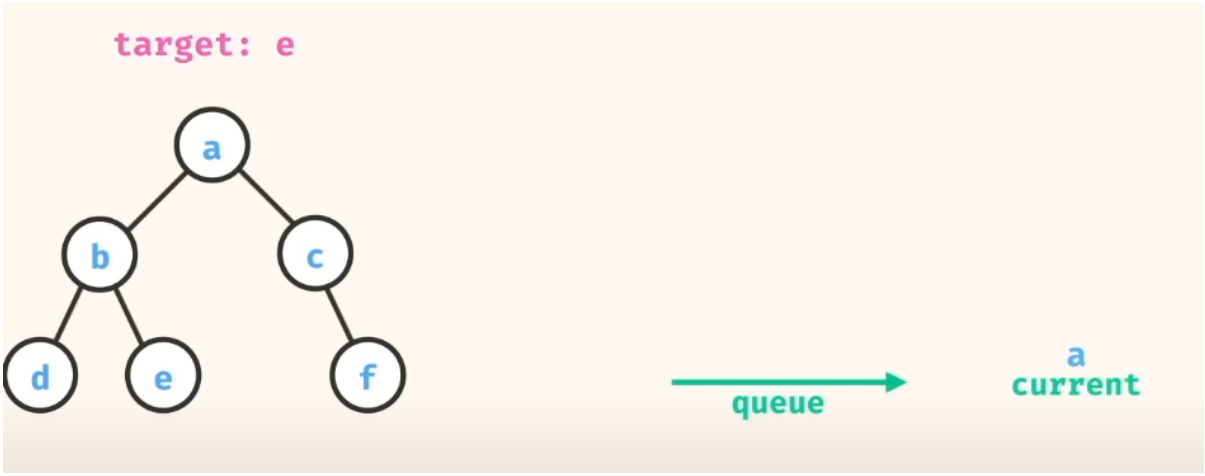
```
public void BreadtFirstTravers(Node start){
            Queue<Node> queue= new Queue<Node>();
            queue.Enqueue(start);
            while(queue.Count>0){
                Node current=queue.Dequeue();
                Console.WriteLine(current.value);
                if (current.left!=null){
                    queue.Enqueue(current.left);
                }
                if (current.right!=null){
                    queue.Enqueue(current.right);
                }
            }
        }
```

2. **Recursive**

```
public Queue<Node> queue= new Queue<Node>();
public void RBreadtFirstTravers(Node start){
    queue.Enqueue(start);
    if (queue.Count<=0){
        return;
    }
    else{
        Node current = queue.Dequeue();
        Console.WriteLine(current.value);
        if (current.left!=null){
            RBreadtFirstTravers(current.left);
        }
        if (current.right!=null){
            RBreadtFirstTravers(current.right);
        }
    }
}
```

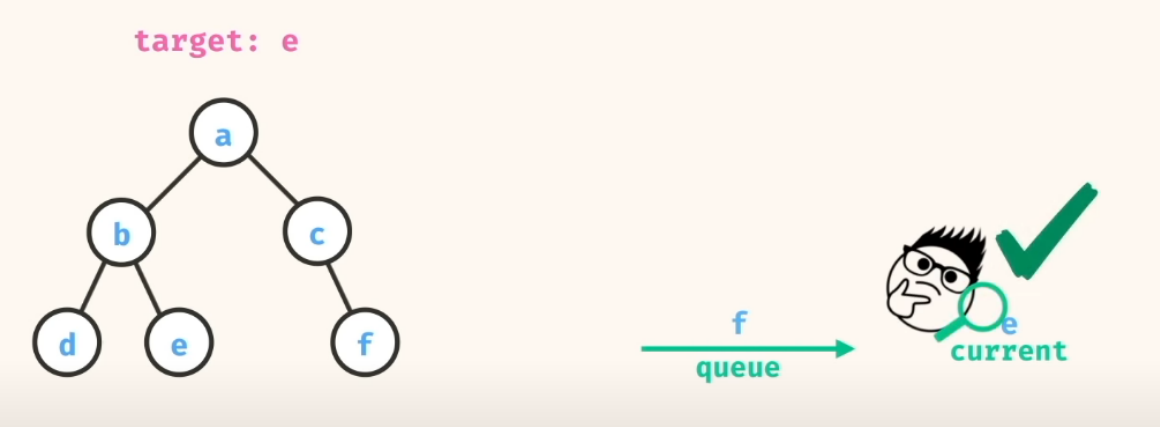
Tree Include

The tree includes aims to check whether a target value exists or not in a tree.



A. Iterative solution Using BFS

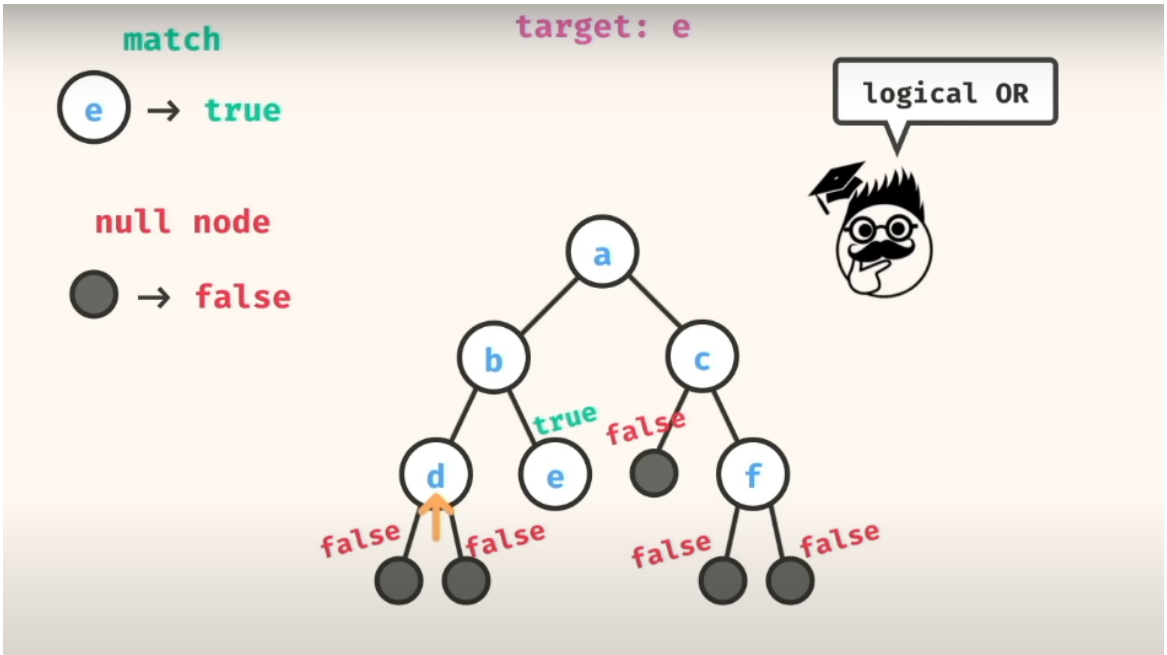
We can use BFS to run through the node and check if the node is equal to the target.



Code implementation using BFS

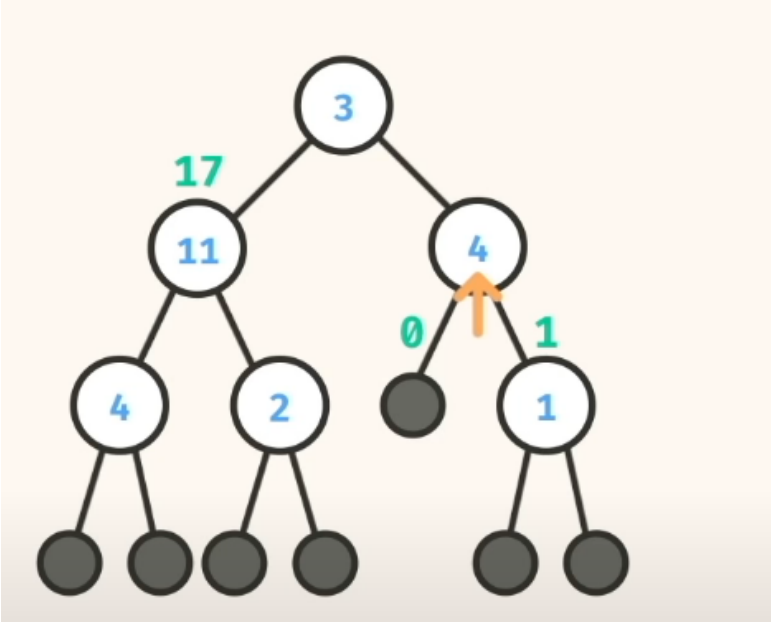
```
public bool Include (Node root, Char target){
Queue<Node> queue = new Queue<Node>();
queue.Enqueue(root);
while(queue.Count>0){
    Node current = queue.Dequeue();
    if (current.value==target){
        return true;
    }
    if (current.left!=null) queue.Enqueue(current.left);
    if (current.right!=null) queue.Enqueue(current.right);
}

return false;
}
```

B. Recursive Solution Using DFS



Tree Sum

We can use DFS or BFS to sum all node in a Tree



Code for Iterative solution **Depth First**

```
public int Sum (Node root){
int sum=0;
Stack<Node> stack = new Stack<Node>();
stack.Push(root);
while(stack.Count>0){
```

```
    Node current = stack.Pop();
    sum+=current.data;
    if (current.left!=null) stack.Push(current.left);
    if (current.right!=null) stack.Push(current.right);
}

return sum;
}
```

Code for Recursive Solution

```
public int RSum (Node root){
    if (root ==null) return 0;
    return root.data + RSum(root.left) + RSum(root.right);
}
```

Tree Min and Max value

1. Min Iterative

```
public int Minvalue(Node root){
  int min=int.MaxValue;
  Stack<Node> stack = new Stack<Node>();
  stack.Push(root);
  while(stack.Count>0){
      Node current = stack.Pop();
      if (current.data<min) min=current.data;
      if (current.left!=null) stack.Push(current.left);
      if (current.right!=null) stack.Push(current.right);
  }
  return min;
}
```

2. Min recursive

```
public int RMinvalue(Node root){
    if (root ==null) return int.MaxValue;
    int leftMin= RMinvalue(root.left);
    int rigtMin= RMinvalue(root.right);
    return Math.Min(root.data, Math.Min(leftMin,rigtMin));
}
```

3. Max recursive

```
public int RMaxnvalue(Node root){
    if (root ==null) return int.MinValue;
    int leftMin= RMaxnvalue(root.left);
    int rigtMin= RMaxnvalue(root.right);
    return Math.Max(root.data, Math.Max(leftMin,rigtMin));
}
```

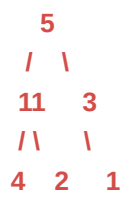
Max Root to Leaf Path Sum

Write a function, *maxPathSum*, that takes in the root of a binary tree that contains number values. The function should return the maximum sum of any root to leaf path within the tree.

```
const a = new Node(3);
const b = new Node(11);
const c = new Node(4);
const d = new Node(4);
const e = new Node(-2);
const f = new Node(1);

a.left = b;
a.right = c;
b.left = d;
b.right = e;
c.right = f;

//       3
//    /    \
//   11     4
//  / \      \
// 4   -2     1

maxPathSum(a); // -> 18
```

Max Path sum aim is to find the path that has maximum sum value. For example the max path sum of the following tree is 18 from (4+11+3) path.

or the following tree has 20 as the **Max Path Sum**

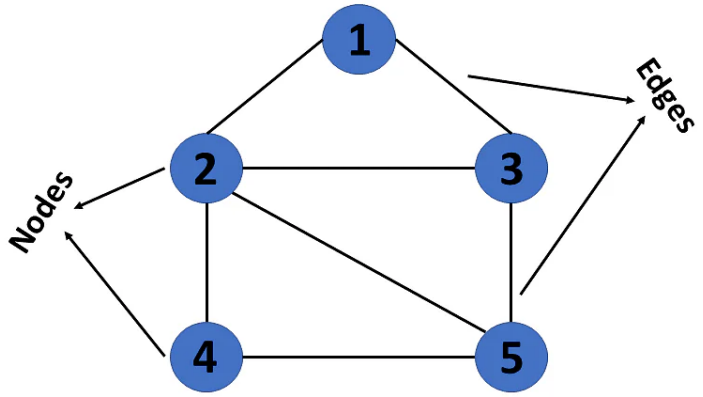


▼ Graph

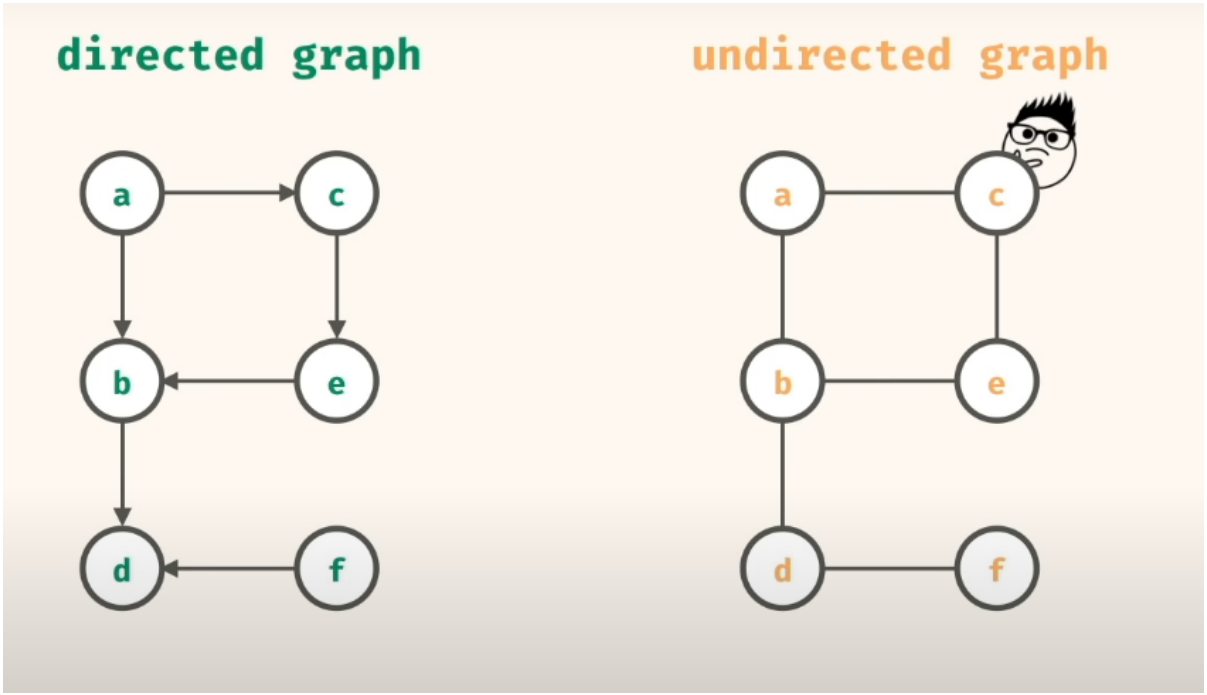
https://www.youtube.com/watch?v=tWVWeAqZ0WU&t=3964s

A graph is a non-linear kind of data structure made up of nodes or vertices and edges. The edges connect any two nodes in the graph, and the nodes are also known as vertices.

Graph = Nodes + Edges



There are two type of graph, which are **directed graph** and **undirected graph**



Directed Graph → a graph in which teh edges have a direction. This is usually indictaed with an arrow on the edge. More formally from the graph above, from node A we are able to travel to c and b, but we can not travel from c to a or b to a.

Undirected Graph → in an undirected graph, the edges are bidirectional, with no direction associated with them. Hence, the graph can be traversed in either direction. The absence of an arrow tells us that the graph is undirected. In the graph above, we are able to travel from a to c and otherwise, we are also able to travel from c to a.