Introduction To Data Structure

DS&A: Data Structure Basics

REVATURE

A data structure is a way to store & organize data in a computer system in order to use it effectively.

Different types of data structures are used for different kinds of applications.

The data structure is not any programming language like C, C++, java, etc.
It is a set of algorithms that we can use in any programming language to structure the data in the memory.

Why to learn data structures & algorithm?

REVATURE

As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

- ***Data Search –*** *Consider an inventory of 1 million(106) items of a store. If the application is to search an item, it has to search an item in 1 million(106) items every time slowing down the search. As data grows, search will become slower.*

- ***Processor Speed*** *– Processor speed although being very high, falls limited if the data grows to billion records.*

- ***Multiple Requests*** *– As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.*

- To solve the above-mentioned problems, data structures come to rescue.
- Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.
- Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc.
- Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artifical intelligence, Graphics and many more.
- Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way.
- It plays a vital role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

REVATURE

1. Variable:

x+2y=10

2. Data Type:

Primitive Data Type (System Defined Data Type):

Non-Primitive Data Type (User Defined Data Type)

Primitive Data Type (System Defined Data Type):

REVATURE

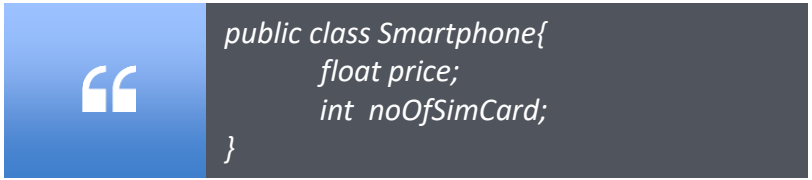
- *Data types that are defined by system are called primitive data types.*
- *The primitive data types provided by many programming languages are: int, float, char, double, bool, etc.*
- *The number of bits allocated for each primitive data type depends on the programming languages, the compiler and the operating system.*
- *For the same primitive data type, different languages may use different sizes.*
- *Depending on the size of the data types, the total available values (domain) will also change.*

For example, "int" may take 2 bytes or 4 bytes. If it takes 2 bytes (16 bits), then the total possible values are minus 32,768 to plus 32,767 (-2^15 to 2^15-1).
If it takes 4 bytes (32 bits), then the possible values are between -2,147,483,648 and +2,147,483,647 (-2^31 to 2^31 -1).
The same is the case with other data types.

Non-Primitive Data Type (User Defined Data Type)

REVATURE

- *If the system-defined data types are not enough, then most programming languages allow the users to define their own data types, called user – defined data types.*
- *Good examples of user defined data types is classes in Java.*
- *For example, in the snippet below, we are combining many system-defined data types and calling the user defined data type by the name "Smartphone".*
- *This gives more flexibility and comfort in dealing with computer memory.*



```
public class Smartphone{
        float price;
        int  noOfSimCard;
}
```

Abstract Data Types (ADTs)

REVATURE

- *We all know that, by default, all primitive data types (int, float, etc.) support basic operations such as addition and subtraction.*
- *The system provides the implementations for the primitive data types.*
- *For user-defined data types we also need to define operations.*
- *The implementation for these operations can be done when we want to actually use them.*
- *That means, in general, user defined data types are defined along with their operations.*
- *To simplify the process of solving problems, we combine the data structures with their operations and we call this Abstract Data Types (ADTs).*
- *An ADT consists of two parts:*
    - ❑ *Declaration of data*
    - ❑ *Declaration of operations*

Abstract Data Types (ADTs) Cont..

REVATURE

- *Commonly used ADTs include-> Linked Lists, Stacks, Queues, Priority Queues, Binary Trees, Dictionaries, Disjoint Sets (Union and Find), Hash Tables, Graphs, and many others.*
- *For example, **Stack** uses LIFO (Last-In-First-Out) mechanism while storing the data in data structures.*
- *The last element inserted into the stack is the first element that gets deleted.*
- *Common operations of it are:*
    - *creating the stack,*
    - *pushing an element onto the stack,*
    - *popping an element from stack,*
    - *finding the current top of the stack,*
    - *finding number of elements in the stack, etc.*
- *Different kinds of ADTs are suited to different kinds of applications, and some are highly specialized to specific tasks.*

Abstract Data Types (ADTs) Cont..

REVATURE

- *A data structure is a way of organizing the data so that it can be used efficiently.*
- *Here, we have used the word efficiently, which in terms of both the space and time.*
- *For example, a stack is an ADT (Abstract data type) which uses either arrays or linked list data structure for the implementation.*
- *Therefore, we conclude that we require some data structure to implement a particular ADT.*
- *An ADT tells what is to be done and data structure tells how it is to be done.*
- *In other words, we can say that ADT gives us the blueprint while data structure provides the implementation part.*
- *Now the question arises: how can one get to know which data structure to be used for a particular ADT?.*
- *As the different data structures can be implemented in a particular ADT, but the different implementations are compared for time and space.*
- *For example, the Stack ADT can be implemented by both Arrays and linked list.*
- *Suppose the array is providing time efficiency while the linked list is providing space efficiency, so the one which is the best suited for the current user's requirements will be selected.*

Classification of Data structures

REVATURE

Data structures can also be classified as:

1) ***Static data structure:*** *It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.*
2) ***Dynamic data structure:*** *It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.*

Type of Data structures

REVATURE

- *Depending on the organization of the elements, data structures are classified into two types:*
  1) ***Linear data structures:*** *Elements are accessed in a sequential order, but it is not compulsory to store all elements sequentially. Examples: Linked Lists, Stacks and Queues.*
  2) ***Non – linear data structures:*** *Elements of this data structure are stored/accessed in a non-linear order. Examples: Trees and graphs.*

Interface & Implementation of Data structures

REVATURE

Data Structure is a systematic way to organize data in order to use it efficiently. Following terms are the foundation terms of a data structure.

1) ***Interface*** *– Each data structure has an interface. Interface represents the set of operations that a data structure supports. An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations.*
2) ***Implementation*** *– Implementation provides the internal representation of a data structure. Implementation also provides the definition of the algorithms used in the operations of the data structure..*

Characteristics of a Data Structure

REVATURE

Following are basic characteristic of data structure.

1) ***Correctness –*** *Data structure implementation should implement its interface correctly.*
2) ***Time Complexity –*** *Running time or the execution time of operations of data structure must be as small as possible.*
3) ***Space Complexity –*** *Memory usage of a data structure operation should be as little as possible.*

Advantages of Data structures

REVATURE

The following are the advantages of a data structure:

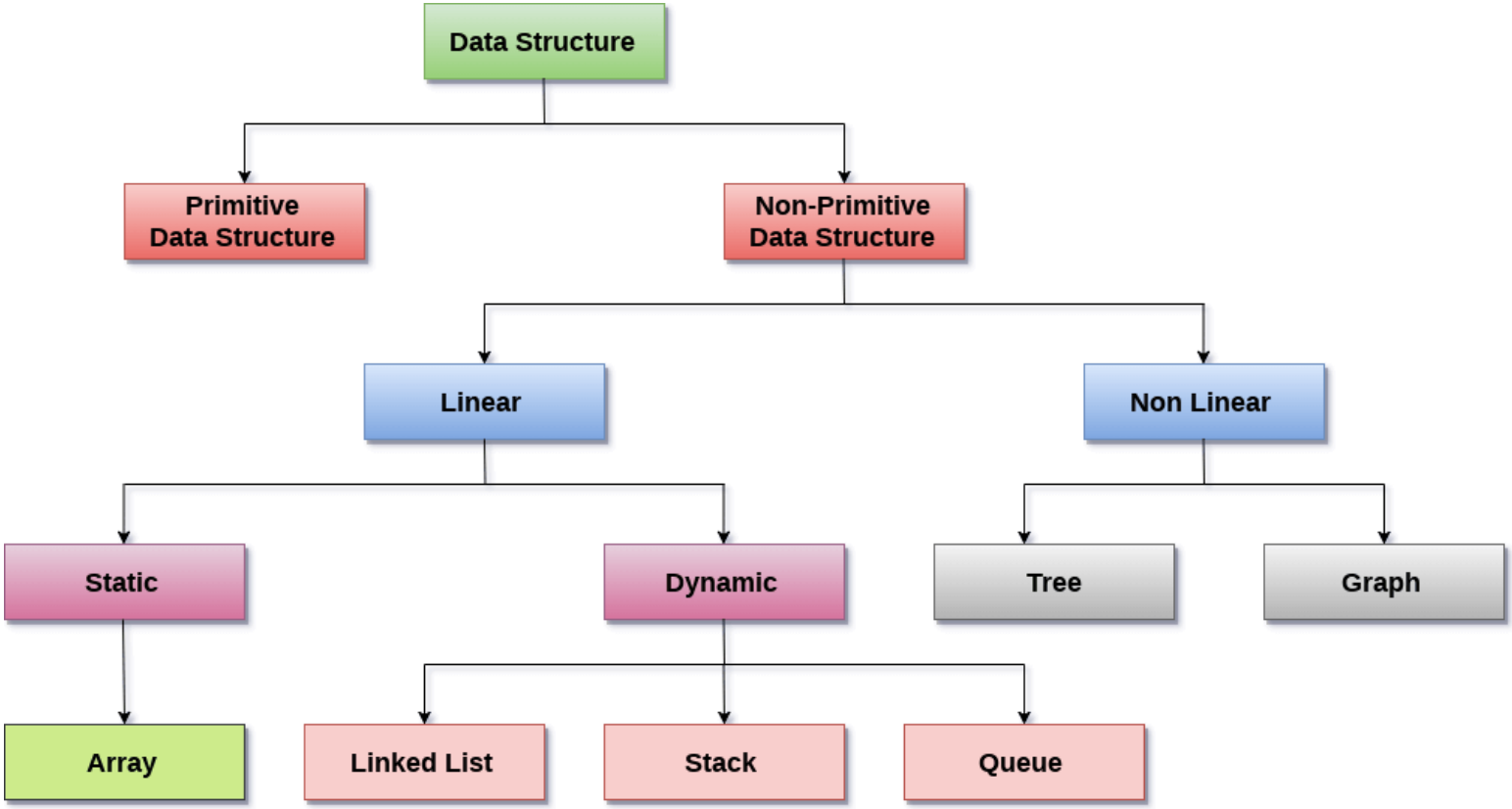
1) ***Efficiency:*** *If the choice of a data structure for implementing a particular ADT is proper, it makes the program very efficient in terms of time and space.*
2) ***Reusability:*** *The data structure provides reusability means that multiple client programs can use the data structure.*
3) ***Abstraction:*** *The data structure specified by an ADT also provides the level of abstraction. The client cannot see the internal working of the data structure, so it does not have to worry about the implementation part. The client can only see the interface.*

Type of Data structures

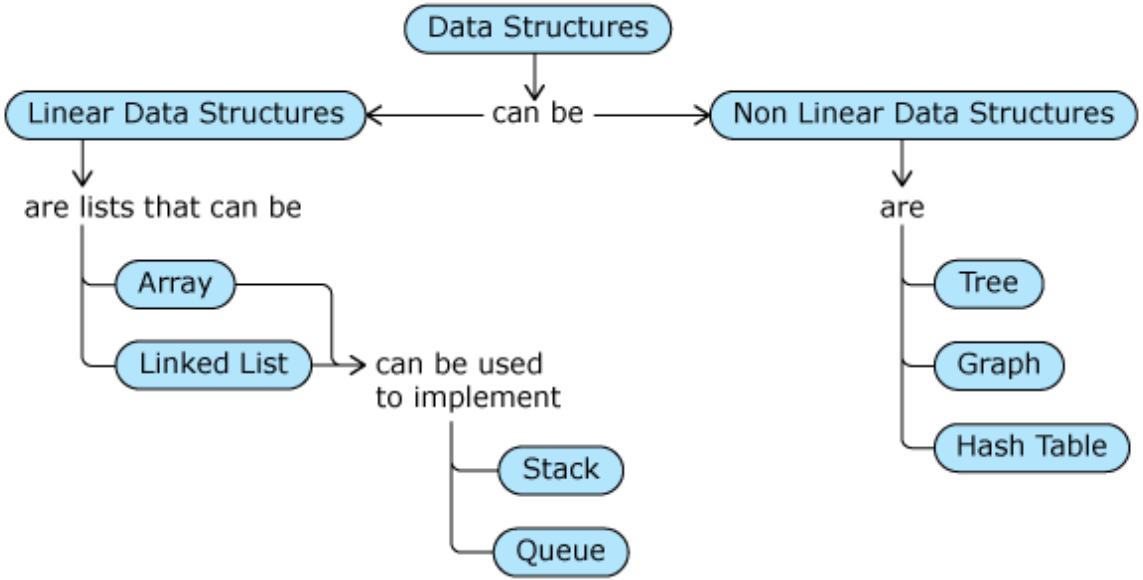
REVATURE

- *Depending on the organization of the elements, data structures are classified into two types:*
  1) ***Linear data structures:*** *Elements are accessed in a sequential order, but it is not compulsory to store all elements sequentially. Examples: Arrays, Linked Lists, Stacks and Queues.*
     In these data structures, one element is connected to only one another element in a linear form.
  1) ***Non – linear data structures:*** *When one element is connected to the 'n' number of elements known as a non-linear data structure. In this case, the elements are arranged in a random manner. Elements of this data structure are stored/accessed in a non-linear order. Examples: Trees and graphs.*

Type of Data structures



Type of Data structures



Operation on Data structures

REVATURE

The major or the common operations that can be performed on the data structures are:

1) ***Traversal:*** *Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.*

Example: If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will devide that sum by the number of subjects i.e. 6, in order to find the average.

2) ***Insertion:*** *Insertion can be defined as the process of adding the elements to the data structure at any location.*

If the size of data structure is n then we can only insert n-1 data elements into it.

3) ***Deletion:*** *The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.*

If we try to delete an element from an empty data structure then underflow occurs.

Operation on Data structures

REVATURE

The major or the common operations that can be performed on the data structures are:

4) ***Updation:*** *We can also update the element, i.e., we can replace the element with another element.*
5) ***Searching:*** *The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later.*
6) ***Sorting:*** *The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.*
7) ***Merging:*** *When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging*

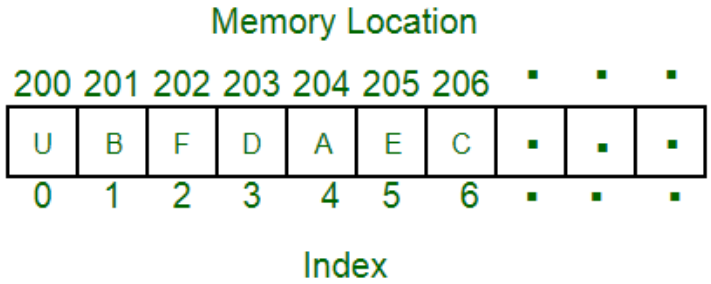
Linear Data Structures

Array

A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

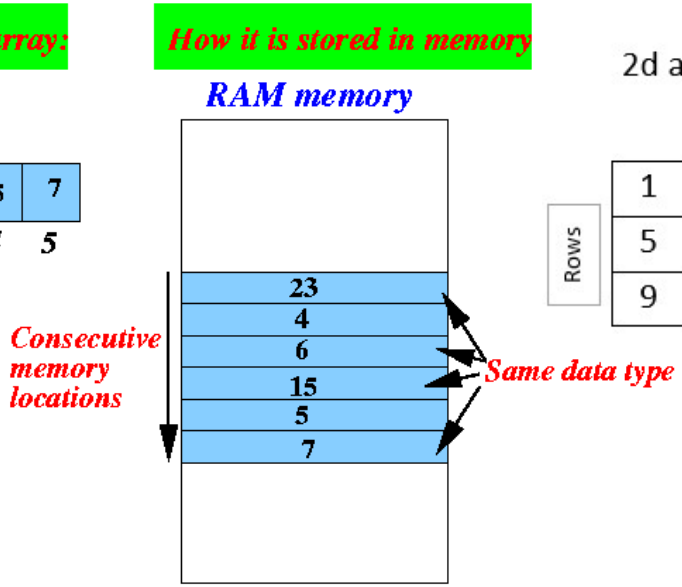
Arrays:

An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.
The elements of array share the same variable name but each one carries a different index number known as subscript.



Array - Dimensions

One Dimension, Two Dimension, Three Dimension



2d array diagram

	Columns			
Rows	1	2	3	4
	5	6	7	8
	9	10	11	12

index position of 2d array

	Columns			
Rows	0,0	0,1	0,2	0,3
	1,0	1,1	1,2	1,3
	2,0	2,1	2,2	2,3

ArrayList

REVATURE

What, Why, How?
Code Demo

Some Applications of the Array are

REVATURE

1. *Arrangement of leader-board of a game can be done simply through arrays to store the score and arrange them in descending order to clearly make out the rank of each player in the game.*
2. *A simple question Paper is an array of numbered questions with each of them assigned to some marks.*
3. *2D arrays, commonly known as, matrix, are used in image processing.*
4. *It is also used in speech processing, in which each speech signal is an array.*

String Class in Java

REVATURE

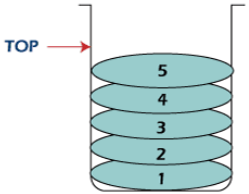
Recap - What, Why, How?
Code Demo

Stack & Its Operations

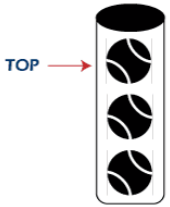
Stack is a linear list in which insertion and deletions are allowed only at one end, called top.

A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

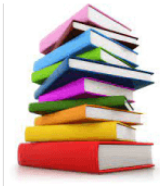
Stack of Coins



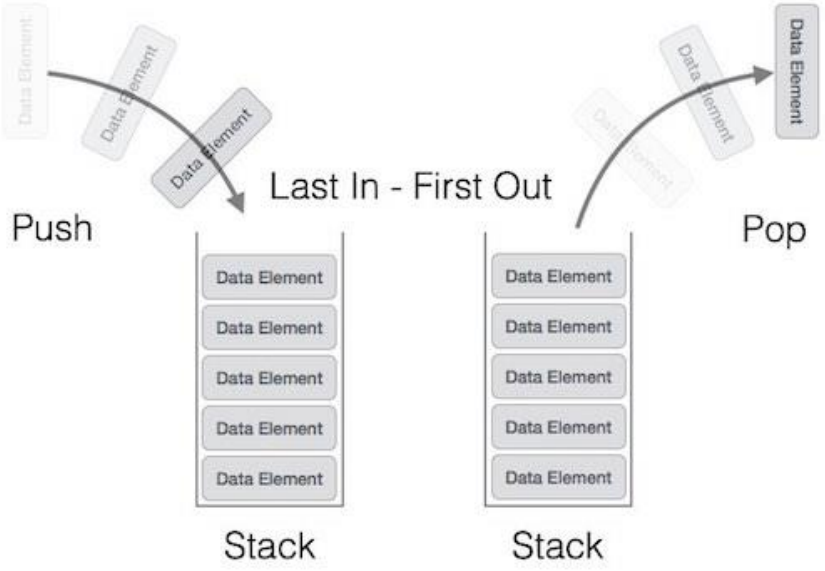
Stack of Plates



Can of Tennis Balls



Stack of Books



Some Applications of a stack are

REVATURE

1. *Check for Balanced Brackets in an expression (well-formedness) using Stack*
2. *Converting infix to postfix expressions.*
3. *Redo-undo features at many places like editors, photoshop.*
4. *Syntaxes in languages are parsed using stacks.*
5. *It is used in many virtual machines like JVM.*
6. *Forward – backward surfing in browser*
7. *History of visited websites*
8. *Message logs and all messages you get are arranged in stack*
9. *Call logs, E-mails, Google photos' any gallery , YouTube downloads, Notifications (latest appears first)*
10. *Scratch card's earned after Google pay transaction*
11. *Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.*
12. *Backtracking is one of the algorithm designing techniques. Some examples of backtracking are the Knight-Tour problem, N-Queen problem, find your way through a maze, and game-like chess or checkers in all these problems we dive into someway if that way is not efficient we come back to the previous state and go into some another path. To get back from a current state we need to store the previous state for that purpose we need a stack.*
13. *In Memory management, any modern computer uses a stack as the primary management for a running purpose. Each program that is running in a computer system has its own memory allocations*
14. *String reversal is also another application of stack. Here one by one each character gets inserted into the stack. So the first character of the string is on the bottom of the stack and the last element of a string is on the top of the stack. After Performing the pop operations on the stack we get a string in reverse order.*

Implement Stack

REVATURE

Demo using Array
Demo using LinkedList

Application of Stack

REVATURE

Convert Infix vs Postfix Expression-
Infix Expression: 3 − 4 + 5 in conventional notation would be written
Postfix(Or Reverse Polish notation): 3 4 − 5 +
Note: 4 is first subtracted from 3, then 5 is added to it.
An advantage of reverse Polish notation is that it removes the need for parentheses that are required by infix notation.

Infix	Infix Evaluation	Postfix	Postfix Evaluation
3 - 4 + 5	4	3 4 - 5 +	(3 - 4) 5 + -1 + 5 4
3 - (4 * 5)	-17	3 4 5 * -	3 (4 5 *) - 3 20 - -17
(3 - 4) * 5	-5	3 4 - 5 *	(3 - 4) 5 * -1 * 5 -5

Queue & Its Operations

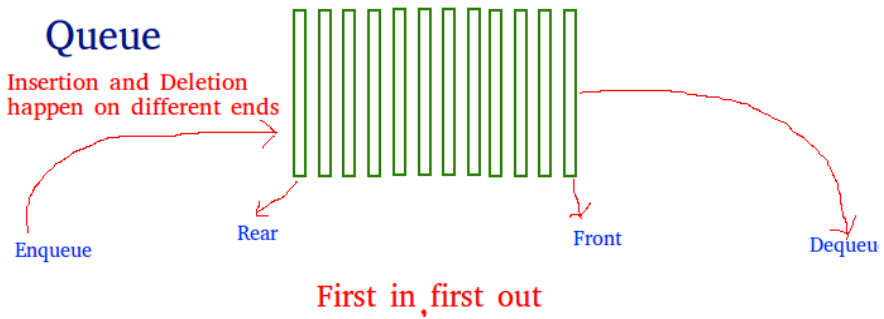
REVATURE

Queue is used when things don't have to be processed immediately but have to be processed in First In First Out order.
A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.
The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Queue & Its Operations Conti..

REVATURE

Operations on Queue:
Mainly the following four basic operations are performed on queue:
***Enqueue**: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.*
***Dequeue**: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.*
***Front**: Get the front item from queue.*
***Rear**: Get the last item from queue.*



Some Applications of a Queue are

REVATURE

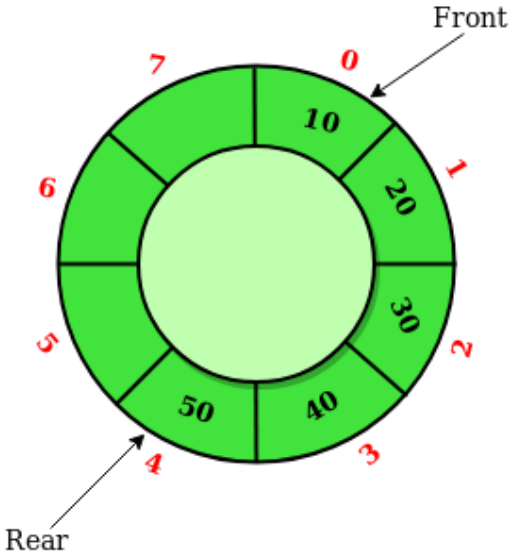
1. *When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.*
2. *When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.*
3. *In Operating systems:*
    - *a) Semaphores*
    - *b) FCFS (first come first serve) scheduling, example: FIFO queue*
    - *c) Spooling in printers*
    - *d) Buffer for devices like keyboard*
4. *In Networks:*
    - *a) Queues in routers/ switches*
    - *b) Mail Queues*

Circular Queue & Its Operations

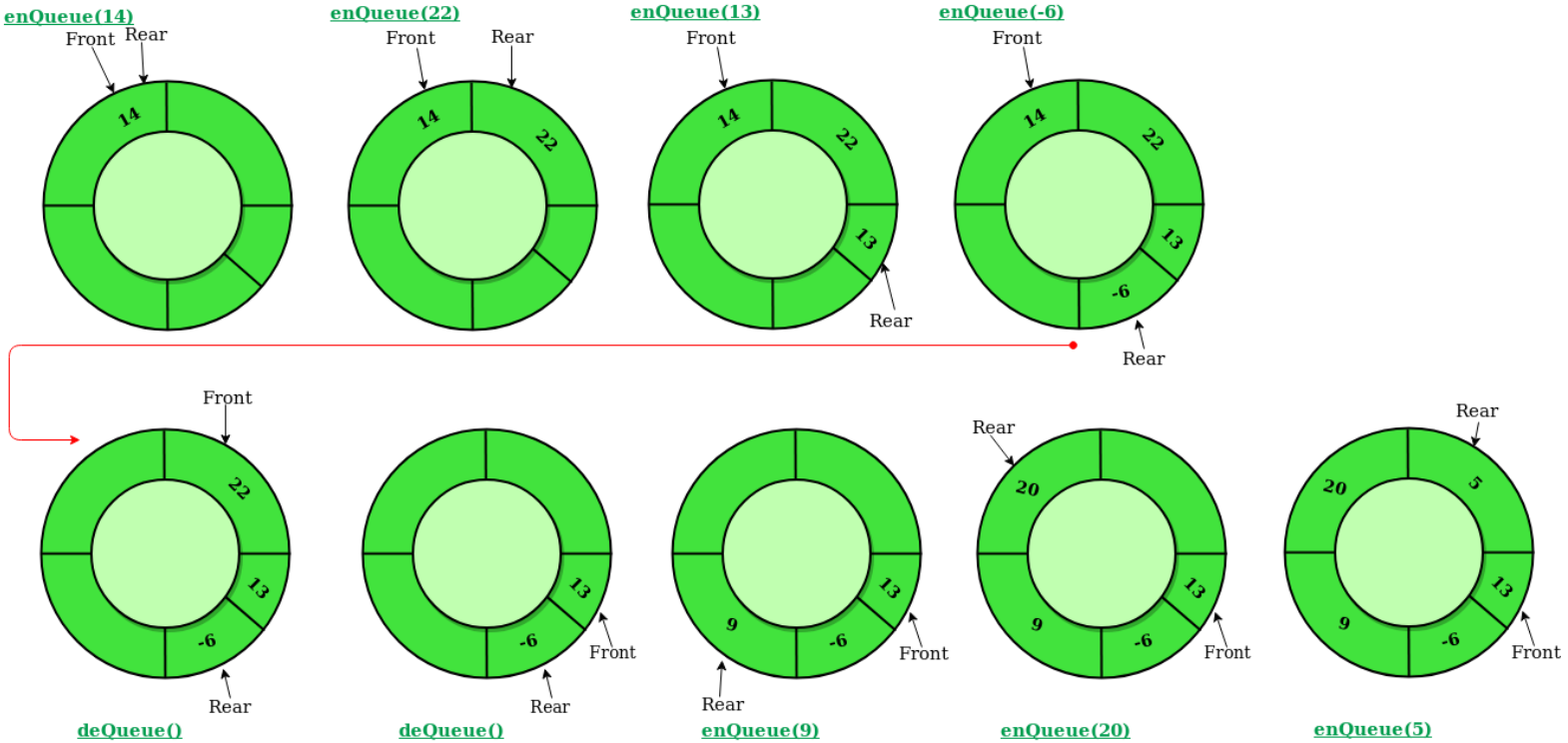
REVATURE

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called 'Ring Buffer'.

In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.



Circular Queue & Its Operations

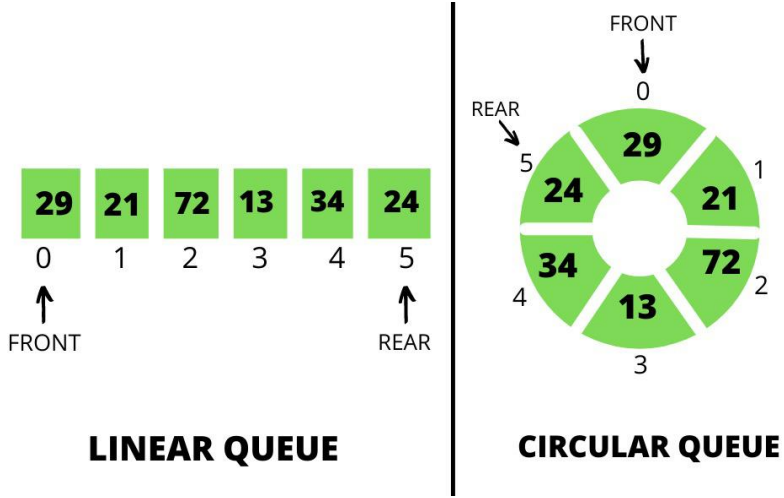


Advantages of circular queue over linear queue

REVATURE

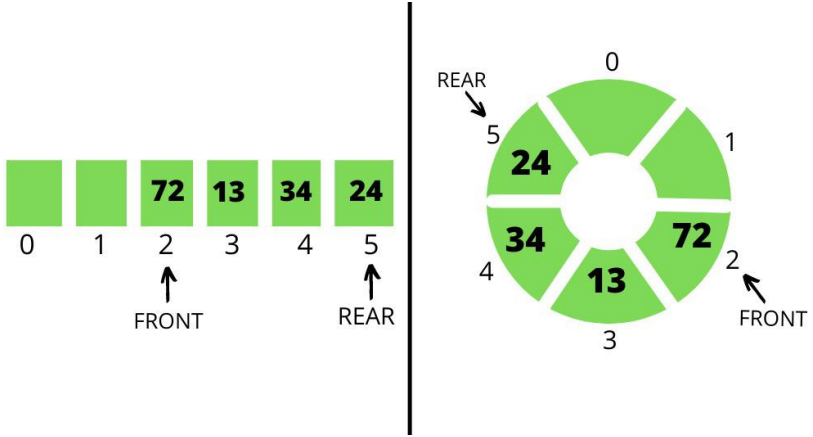
Circular Queue: Circular Queue is just a variation of the linear queue in which front and rear-end are connected to each other to optimize the space wastage of the Linear queue and make it efficient.

Below are the operations that will illustrate that how is Circular Queue is better than Linear Queue:
When Enqueue operation is performed on both the queues: Let the queue is of size 6 having elements {29, 21, 72, 13, 34, 24}. In both the queues the front points at the first element 29 and the rear points at the last element 24 as illustrated below:



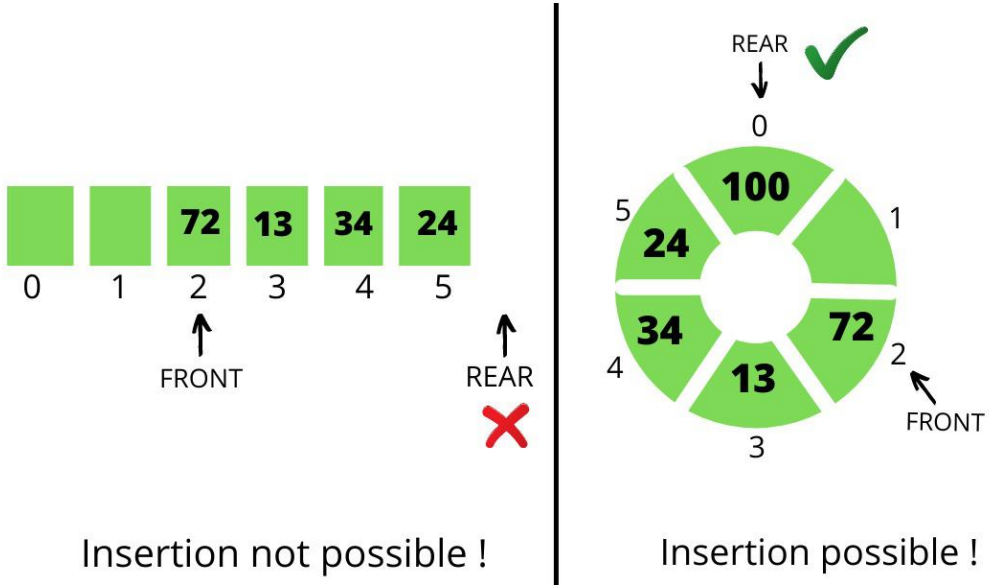
Advantages of circular queue over linear queue

When the Dequeue operation is performed on both the queues: Consider the first 2 elements that are deleted from both the queues. In both the queues the front points at element 72 and the rear points at element 24 as illustrated below:



Advantages of circular queue over linear queue

Now again enqueue operation is performed: Consider an element with a value of 100 is inserted in both the queues. The insertion of element 100 is not possible in Linear Queue but in the Circular Queue, the element with a value of 100 is possible as illustrated below



Advantages of circular queue over linear queue

REVATURE

Explanation

As the insertion in the queue is from the rear end and in the case of Linear Queue of fixed size insertion is not possible when rear reaches the end of the queue.

But in the case of Circular Queue, the rear end moves from the last position to the front position circularly.

Conclusion: *The circular queue has more advantages than a linear queue. Other advantages of circular queue are:*

- ***Easier for insertion-deletion:*** *In the circular queue, elements can be inserted easily if there are vacant locations until it is not fully occupied, whereas in the case of a linear queue insertion is not possible once the rear reaches the last index even if there are empty locations present in the queue.*
- ***Efficient utilization of memory:*** *In the circular queue, there is no wastage of memory as it uses the unoccupied space, and memory is used properly in a valuable and effective manner as compared to a linear queue.*
- ***Ease of performing operations:*** *In the linear queue,* ***FIFO*** *is followed, so the element inserted first is the element to be deleted first. This is not the scenario in the case of the circular queue as the rear and front are not fixed so the order of insertion-deletion can be changed, which is very useful.*

Circular Queue Operations

REVATURE

Operations on Circular Queue:
***Front**: Get the front item from queue.*
***Rear**: Get the last item from queue.*
***enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.*
Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).
If it is full, then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.
***deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.*
Check whether queue is Empty means check (front==-1).
If it is empty, then display Queue is empty. If queue is not empty, then step 3
Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

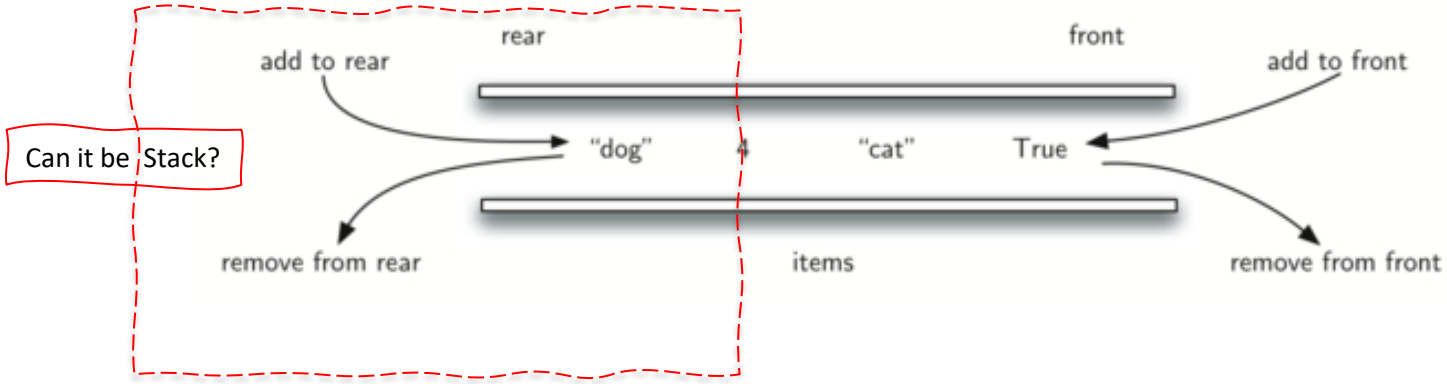
Some Applications of a Circular Queue are

REVATURE

1. ***Memory Management:*** *The unused memory locations in the case of ordinary queues can be utilized in circular queues.*
2. ***Traffic system:*** *In computer controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.*
3. ***CPU Scheduling:*** *Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.*

Dequeue & Its Operations

Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends. A deque, is an ordered collection of items similar to the queue.
It has two ends, a front and a rear, and the items remain positioned in the collection.
What makes a deque different is the unrestrictive nature of adding and removing items.
New items can be added at either the front or the rear.
Likewise, existing items can be removed from either end. In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure.
It is important to note that even though the deque can assume many of the characteristics of stacks and queues, it does not require the LIFO and FIFO orderings that are enforced by those data structures.
It is up to you to make consistent use of the addition and removal operations.



Dequeue & Its Operations Conti..

REVATURE

Operations on Dequeue:
Mainly the following four basic operations are performed on queue:
insertFront(): *Adds an item at the front of Deque.*
insertLast(): *Adds an item at the rear of Deque.*
deleteFront(): *Deletes an item from front of Deque.*
deleteLast(): *Deletes an item from rear of Deque.*

In addition to above operations, following operations are also supported
getFront(): *Gets the front item from queue.*
getRear(): *Gets the last item from queue.*
isEmpty(): *Checks whether Deque is empty or not.*
isFull(): *Checks whether Deque is full or not.*

Implementation: A Deque can be implemented either using a doubly linked list or circular array.

Some Applications of a Dequeue are

REVATURE

1. *Since Deque supports both stack and queue operations, it can be used as both.*
2. *The Deque data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications.*
3. *Also, the problems where elements need to be removed and or added both ends can be efficiently solved using Deque.*
4. *When modeling any kind of real-world waiting line: entities (bits, people, cars, words, particles, whatever) arrive with a certain frequency to the end of the line and are serviced at a different frequency at the beginning of the line. While waiting some entities may decide to leave the line.... etc. The point is that you need "fast access" to insert/deletes at both ends of the line, hence a deque.*

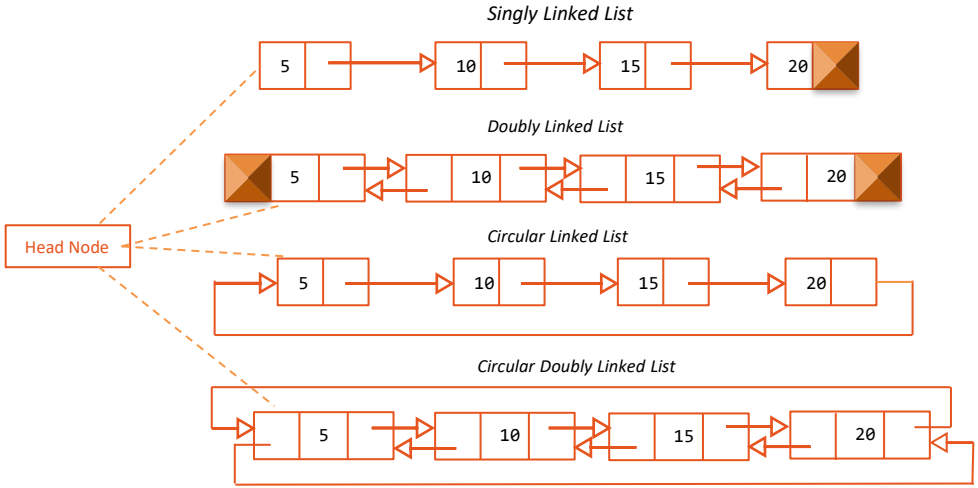
Implement Queue

REVATURE

Demo using ArrayDeque
Demo using PriorityQueue

LinkedList

LinkedList is a linear data structure where each element is an object.
Unlike Array, LinkedList is doesn't have a contiguous memory structure. Each element is linked to the next through a pointer.
Each element in the LinkedList is called Node. Each node contains a key (the data of interest) and an additional pointer for pointing the next element in the list. Initial pointer in a LinkedList is called Head. It is necessary to mention that "Head" is not another node but a pointer to the first element of the LinkedList. For an empty LinkedList value of Head will be null.



Implement LinkedList

REVATURE

Demo using LinkedList

Hash Table & Its Operations

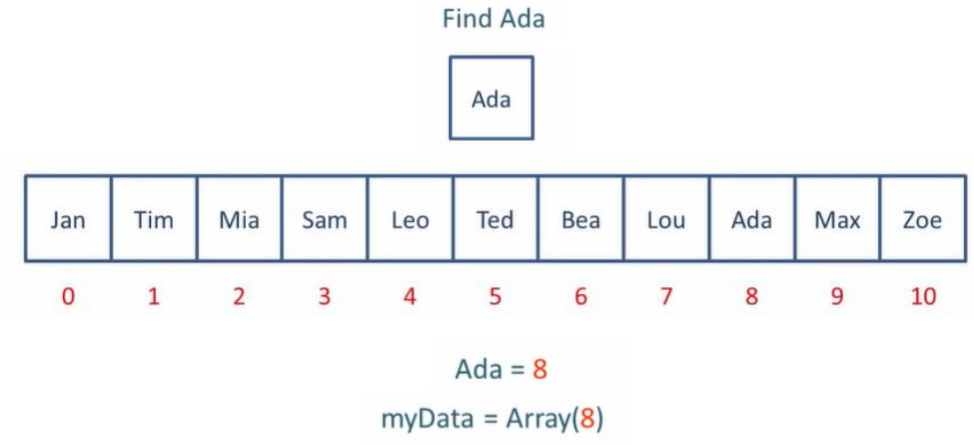
REVATURE

Hash table is a data structure that allows very fast retrieval of data, no matter how much data there is.
Application of Hash Table-

- *Database indexing*
- *Caching*
- *Program Compilation*
- *Password Authentication*
- *Error Checking etc.*

Why Hash Table?

Problem: Find Ada?
Solution 1: Use linear search (Brute force approach).
Drawback: Slow approach if array is of big size since need to check each value one by one.



Solution 2: What if we know the index? We can directly locate the value instead of searching sequentially, irrespective of how much big array or which position its located

Solution 2, But How?

Solution 2: What if we know the index? We can directly locate the value instead of searching sequentially, irrespective of how much big array or which position its located

Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Jan	J	74	a	97	n	110	281	6
Ada	A	65	d	100	a	97	262	9
Leo	L	76	e	101	o	111	288	2
Sam	S	83	a	97	m	109	289	3
Lou	L	76	o	111	u	117	304	7
Max	M	77	a	97	x	120	294	8
Ted	T	84	e	101	d	100	285	10

Index number = *sum ASCII codes* Mod *size of array*

Find Ada Ada = (65 + 100 + 97) = 262

Find Ada 262 Mod 11 = 9

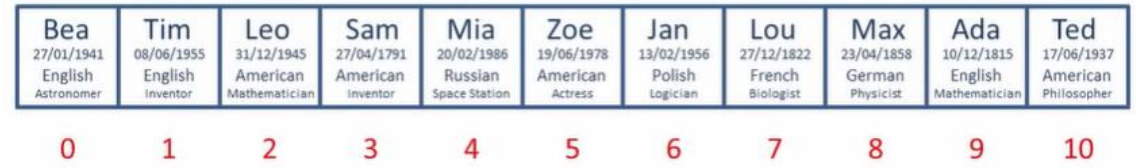
myData = Array(9)

Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Ada	Ted
0	1	2	3	4	5	6	7	8	9	10

Hash Table Structure

REVATURE

Rather than just storing and individual item data, hash tables are often used to store Key, Value pairs.
That time Hash Table are also called as Hash Map
e.g. Key-> Name and Value-> Date of birth



Key → Name
Value → Object

Hashing Algorithm (Hashing Function)

REVATURE

Hashing Algorithm is the calculation applied to a key (which may be a very large number or very large string), to transform it to relatively small index number that correspond to the position in a hash table.
This index number is effectively a memory address.
The beauty of such hash algorithm is that if you feed the same input into it again and again, it will give the same index each time.
If Key is numeric value - Divide the key by number of available addresses, n, and take the remainder.

address=key Mod n

If Key is alphanumeric value - Divide the sum of ASCII codes in a key by the number of available addresses, n, and take the remainder.

*Another method which can be applied is called **folding method** divides key into equal parts then adds the parts together*
e.g.

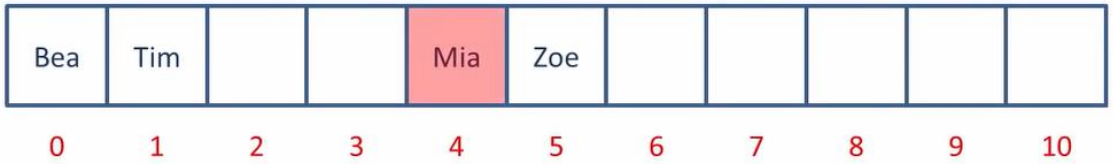
Telephone no: 01452 8345654 becomes 01+45+28+34+56+54=218
If the size of hash table is n then divide by n and take the remainder.

There are lots of different Hashing Algorithms to choose from, some more appropriate than other depending upon nature of your data.

Collisions

Hashing Algorithm might generate same index for two data elements which results into situation called collision.

Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4



Solution:

1. Open Addressing Techniques
   - Linear Probing
   - Double Hashing
2. Closed Addressing Techniques
   - Chaining

Open Addressing Techniques – Linear Probing

Clash at position 4 we check for next available position for Sue since Zoe took it Sue need to be placed inside 6th position.
For Rae we did not find ay position till 7th index hence it needs to be placed in 8th index.

Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4
Len	L	76	e	101	n	110	287	1
Moe	M	77	o	111	e	101	289	3
Lou	L	76	o	111	u	117	304	7
Rae	R	82	a	97	e	101	280	5
Max	M	77	a	97	x	120	294	8
Tod	T	84	o	111	d	100	295	9

Bea	Tim	Len	Moe	Mia	Zoe	Sue	Lou	Rae	Max	Tod
0	1	2	3	4	5	6	7	8	9	10

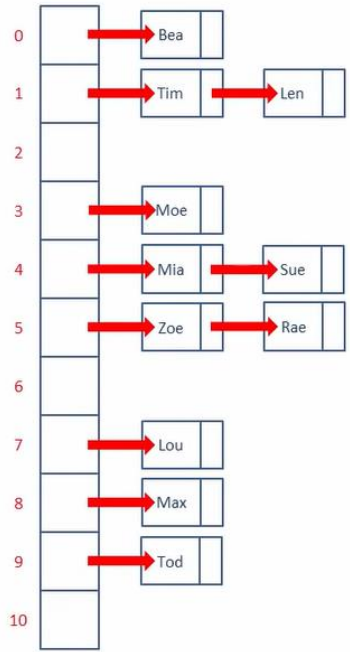
Open Addressing Techniques – Linear Probing

REVATURE

- *If calculated address is occupied, then the linear search is used to find the next available slot.*
- *If linear probing gets till end of array and it still cannot find the free space it might cycle around at the beginning of the array & continue searching from there.*
- *Now when you want to search any item in hash table the hashing function is applied again & if there are collisions, and some items are not at their calculated positions then finding the item will also involve linear probing i.e a linear search.*
- *The more items are there in hash table the more likely you are to get collisions when you insert even more data.*
- *One way to deal with this is to make hash table bigger than needed for the total amount of data you will be expecting, perhaps such that only 70% of the hash table is ever occupied.*
- *The ratio between the number of items stored and the size of the data array is known as the* ***Load Factor.***
- *If the hash table is implemented as a resizable dynamic data structure it could be made to increase in size automatically when the load factor reaches a certain threshold****.***
- *In an ideal world every item will be stored in the hash table according to its calculated index in this best-case scenario the time taken to find any particular item is always the same but you can imagine the worst case scenario depending upon the nature of the data used to calculate the index values and depending upon the appropriateness of the hash algorithm some items may require a linear search of the whole table in order to be found.*
- *As long as the load factor is reasonably low open addressing with linear probing should work reasonably well.*

Chaining

For Collision values separate linked list is maintain and on searching the same is iterated till the item is located.
Remember the lookup is quicker here but traversing Linked list also comes with cost.
If the load factor is low better choose open addressing



Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4
Len	L	76	e	101	n	110	287	1
Moe	M	77	o	111	e	101	289	3
Lou	L	76	o	111	u	117	304	7
Rae	R	82	a	97	e	101	280	5
Max	M	77	a	97	x	120	294	8
Tod	T	84	o	111	d	100	295	9

Find Rae
280 Mod 11 = 5
myData = Array(5)

Objective of Hash Function

REVATURE

If you know all of the keys in advanced, then its theoretically possible to come up with perfect hash function.
One that will produce unique index for each and every data item.
In fact if you know the data in advanced you can come up with perfect hash function that uses all of the available spaces in the array.
Objective of hash function-

1. *It should minimize the collisions*
2. *Uniform distribution of hash values*
3. *Easy to calculate*
4. *Should include suitable method to resolve any collisions that do occur.*

Hash Tables are used to index large amount of data
Address of each key is calculated using the key itself.
Collisions are resolved either with Open or closed addressing.

Implement Hashtable/Hashmap

REVATURE

Demo using Hashtable - how java uses chaining technique for collisions and organize the data in Hashtable
Demo using HashMap - how java uses chaining technique for collisions and organize the data in HashMap

#	Category	Hashtable	HashMap
1	Default Capacity	11	16
2	Default Load Factor	0.75	0.75
3	Capacity increased as	int newCapacity = (11 << 1) + 1;	int newCap = oldCap << 1;
4	Collision chaining sequence	Old->Old->New	New->Old->Old
5	Synchronized	Yes	No
6	Null Key/Value	No	Only 1 null Key, multiple null values
7	Performance	Slow	Fast
8	Legacy	Old One	New One

REVATURE

Algorithm

DS&A: Algorithm Basic

REVATURE

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.

Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

It is not the complete program or code.

It is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudocode.

DS&A: Algorithm Basic Cont..

REVATURE

Problem – Design an algorithm to add two numbers and display the result.
Solution – Multiple

Step 1 – START
Step 2 – Declare three integers a, b & c
Step 3 – Define values of a & b
Step 4 – Add values of a & b
Step 5 – Store output of step 4 to c
Step 6 – Print c
Step 7 – STOP

Step 1 – START
Step 2 – Get values of a & b
Step 3 – c ← a + b
Step 4 – Display c
Step 5 – STOP

Need of Algorithm

REVATURE

1. To understand the basic idea of the problem.
2. To find an approach to solve the problem.
3. To improve the efficiency of existing techniques.
4. To understand the basic principles of designing the algorithms.
5. To compare the performance of the algorithm with respect to other techniques.
6. It is the best method of description without describing the implementation detail.
7. The Algorithm gives a clear description of requirements and goal of the problem to the designer.
8. A good design can produce a good solution.
9. To understand the flow of the problem.
10. To measure the behavior (or performance) of the methods in all cases (best cases, worst cases, average cases)
11. With the help of an algorithm, we can also identify the resources (memory, input-output) cycles required by the algorithm.
12. With the help of algorithm, we convert art into a science.
13. To understand the principle of designing.
14. We can measure and analyze the complexity (time and space) of the problems concerning input size without implementing and running it; it will reduce the cost of design.

Analysis of algorithm

REVATURE

The analysis is a process of estimating the efficiency of an algorithm. There are two fundamental parameters based on which we can analysis the algorithm:
Space Complexity: *The space complexity can be understood as the amount of space required by an algorithm to run to completion.*
Time Complexity: *Time complexity is a function of input size n that refers to the amount of time needed by an algorithm to run to completion.*

Since there can be multiple algorithms for same problem, It would be best to analyze every algorithm in terms of
Time - *that relates to which one could execute faster and*
Memory - *corresponding to which one will take less memory.*

Time vs Memory Complexity Which One to Choose

REVATURE

- *We will be focusing more on time rather than space because time is a more limiting parameter in terms of the hardware.*
- *It is not easy to take a computer and change its speed. So, if we are running an algorithm on a particular platform, we are more or less stuck with the performance that platform can give us in terms of speed.*
- *However, on the other hand, memory is relatively more flexible.*
- *We can increase the memory as when required by simply adding a memory card. So, we will focus on time than that of the space.*
- *The running time is measured in terms of a particular piece of hardware, not a robust measure.*
- *When we run the same algorithm on a different computer or use different programming languages, we will encounter that the same algorithm takes a different time.*

Different system configuration both H/W & S/W like mac, windows, processors, ram e.g. will take different times to execute the problem. Hence denoting/expressing's time complexity is difficult.

Three types of Time Analysis

REVATURE

Generally, we make three types of analysis, which is as follows:

- ***Worst-case time complexity:***
  - *For 'n' input size, the worst-case time complexity can be defined as the maximum amount of time needed by an algorithm to complete its execution.*
  - *Thus, it is nothing, but a function defined by the maximum number of steps performed on an instance having an input size of n.*
- ***Average case time complexity**:*
  - *For 'n' input size, the average-case time complexity can be defined as the average amount of time needed by an algorithm to complete its execution.*
  - *Thus, it is nothing, but a function defined by the average number of steps performed on an instance having an input size of n.*
- ***Best case time complexity**:*
  - *For 'n' input size, the best-case time complexity can be defined as the minimum amount of time needed by an algorithm to complete its execution.*
  - *Thus, it is nothing, but a function defined by the minimum number of steps performed on an instance having an input size of n.*

Asymptotic Analysis

REVATURE

In mathematical analysis, asymptotic analysis, also known as asymptotics, is a method of describing limiting behavior.
The methodology has the applications across science.
It can be used to analyze the performance of an algorithm for some large data set.

As an illustration, suppose that we are interested in the properties of a function $f(n)$ *as n becomes very large. If*

$f(n) = n^2 + 3n$, *then as n becomes very large, the term* $3n$ *becomes insignificant compared to* n^2.

The function $f(n)$ *is said to be "asymptotically equivalent to* n^2, *as* $n \to \infty$".
This is often written symbolically as $f(n) \sim n^2$, *which is read as*

"$f(n)$ is asymptotic to n^2"

https://www.javatpoint.com/data-structure-asymptotic-analysis

Asymptotic Notation

REVATURE

- *Asymptotic Notation is used to describe the running time of an algorithm - how much time an algorithm takes with a given input, n.*
- *Asymptotic notations are used to write fastest and slowest possible running time for an algorithm.*
- *These are also referred to as 'best case' and 'worst case' scenarios respectively.*
- *Why is Asymptotic Notation Important?*
    1. *They give simple characteristics of an algorithm's efficiency.*
    2. *They allow the comparisons of the performances of various algorithms.*

"In asymptotic notations, we derive the complexity concerning the size of the input. (Example in terms of n)"
"These notations are important because without expending the cost of running the algorithm, we can estimate the complexity of the algorithms."

Asymptotic Notation Types

REVATURE

- *Asymptotic Notation is a way of comparing function that ignores constant factors and small input sizes.*
- *Three notations are used to calculate the running time complexity of an algorithm:*
    1. *Big-oh (**O**) notation - describes the upper bound of the complexity.*
    2. *Omega (**Ω**) Notation - describes the lower bound of the complexity.*
    3. *Theta (**θ**) Notation - describes the exact bound of the complexity.*

Big-oh (O) Notation

REVATURE

- *The Big-O notation describes the worst-case running time of a program.*
- *We compute the Big-O of an algorithm by counting how many iterations an algorithm will take in the worst-case scenario with an input of N.*
- *We typically consult the Big-O because we must always plan for the worst case. For example, O(log n) describes the Big-O of a binary search algorithm (explore more on* ***Arrays.sort()****,* **Arrays.binarySearch() & Collections.binarysearch()** methods*.*

$O(Log\ n)$ *but what is the* $Log_x Y$

$Log_2 8$ *What is Logarithm base 2 of 8?*

Let $Log_2 8 = x$

i.e. $2^x = 8$

i.e. $2^x = 2^3$

i.e. $x = 3$

Answer =3

Big-oh (O) Notation Rules

REVATURE

There are few basic rules for calculating algorithm's Big O Notation:

1. *If an algorithm performs a certain sequence of steps f(N) times for a mathematical function f, it takes O(f(N)) steps.*
2. *If an algorithm performs an operation that takes f(N) steps and then performs another operation that takes g(N) steps for function f and g, the algorithm's total performance is is O(g(N) + f(N)).*
3. *If an algorithm takes O(g(N) + f(N)) steps and the function f(N) is bigger than g(N), algorithm's performance can be simplified to O(f(N)).*
4. *If an algorithm performs an operation that takes f(N) steps, and for every step performs another operation that takes g(N) steps, algorithm's total performance is O(f(N)×g(N)).*

Rule 1 : Big-oh (O) Notation

REVATURE

"If an algorithm performs a certain sequence of steps f(N)times for a mathematical function f, it takes O(f(N)) steps."

```
1   function findBiggestNumber(array) {
2       let biggest = array[0];
3
4       for (let i = 0; i < array.length; i++) {
5           if (array[i] > biggest) {
6               biggest = array[i];
7           }
8       }
9
10      return biggest;
11  }
```

- *This algorithm takes an array as an argument and returns the biggest number in that array.*
- *We define the biggest variable equal to the first value in the array.*
- *Then we loop through the array and find the biggest value in it.*
- *After the loop is finished, we return the biggest variable.*
- *This algorithm examines each of the N items once (where N is an array length), so it's performance O(N).*

Rule 2 : Big-oh (O) Notation

REVATURE

"If an algorithm performs an operation that takes f(N) steps and then performs another operation that takes g(N) steps for function f and g, the algorithm's total performance is O(g(N) + f(N))."

```
1   function findBiggestNumber(array) {
2       let biggest = array[0];         // O(1)
3
4       for (let i = 0; i < array.length; i++) {     //O(N)
5           if (array[i] > biggest) {
6               biggest = array[i];
7           }
8       }
9
10      return biggest;                 // O(1)
11  }
```

- *If you look again at the findBiggestNumber shown at the previous section, you'll see that there are few operations outside the loop(line 2 and 10).*
- *Each outer operation takes constant amount of time, so both of them has performance O(1)*
- *Then the total runtime of the algorithm is O(1 + N + 1). You can use algebra rules inside of Big O Notation, so final algorithm's performance is O(N + 2).*

Rule 3 : Big-oh (O) Notation

REVATURE

"If an algorithm takes O(f(N) + g(N)) steps and the function f(N) is bigger than g(N), algorithm's performance can be simplified to O(f(N))."

```
1   function findBiggestNumber(array) {
2       let biggest = array[0];         // O(1)
3
4       for (let i = 0; i < array.length; i++) {     //O(N)
5           if (array[i] > biggest) {
6               biggest = array[i];
7           }
8       }
9
10      return biggest;                 // O(1)
11  }
```

- *The previous findBiggestNumber algorithm has O(N + 2) runtime.*
- *When N grows large, the function N is larger than our constant value 2, so algorithm's runtime can be simplified to O(N).*
- *Ignoring smaller functions helps you to concentrate on the algorithm's behavior as N size becomes large.*

Rule 4 : Big-oh (O) Notation

REVATURE

"If an algorithm performs an operation that takes f(N)steps, and for every step performs another operation that takes g(N) steps, algorithm's total performance is O(f(N)×g(N))."

- *These algorithm takes an array as the only one argument and returns if the array contains duplicate values. Algorithm has two nested loops.*
- *The outer loop iterates through all items in array, so it takes O(N) steps.*
- *For each iteration of the outer loop the inner loop also iterates over all items in array, so it takes O(N) steps too.*
- *Because of the fact that one loop is nested inside the other we can combine their performances — $O(N \times N)$ or $O(N^2)$.*

```
1   function containsDuplicates(array) {
2       for (let i = 0; i < array.length; i++) {
3           for (let r = 0; r < array.length; r++) {
4               if (i === r) {
5                   continue;
6               }
7               if (array[i] === array[r]) {
8                   return true;
9               }
10          }
11      }
12
13      return false;
14  }
```

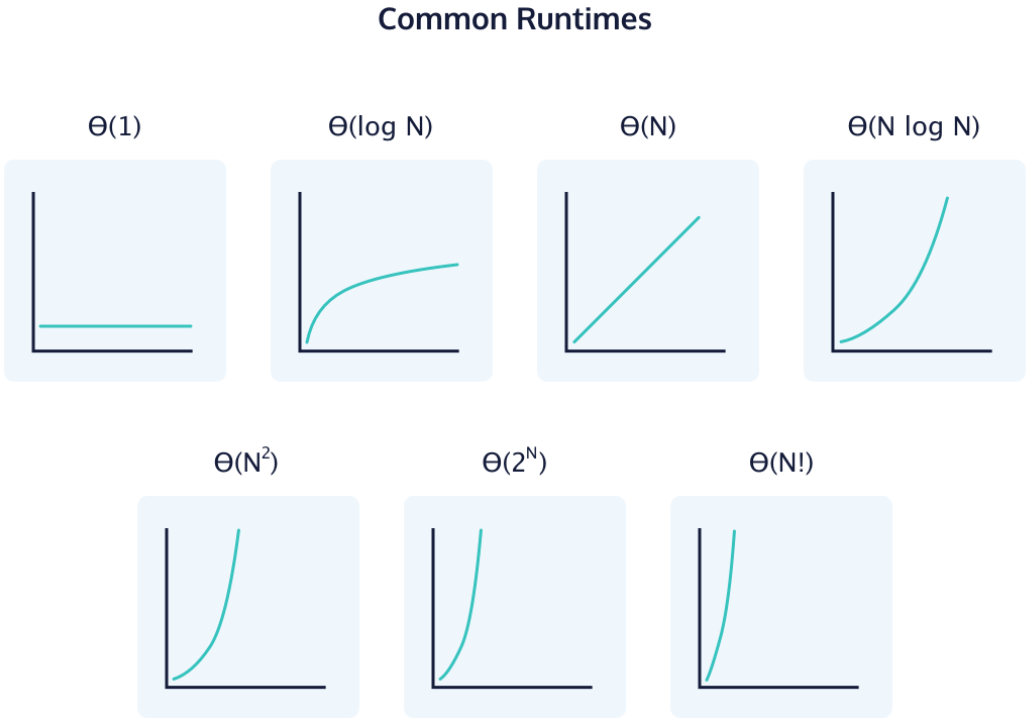
Big-oh (O) Notation Algorithmic Common Runtimes

REVATURE

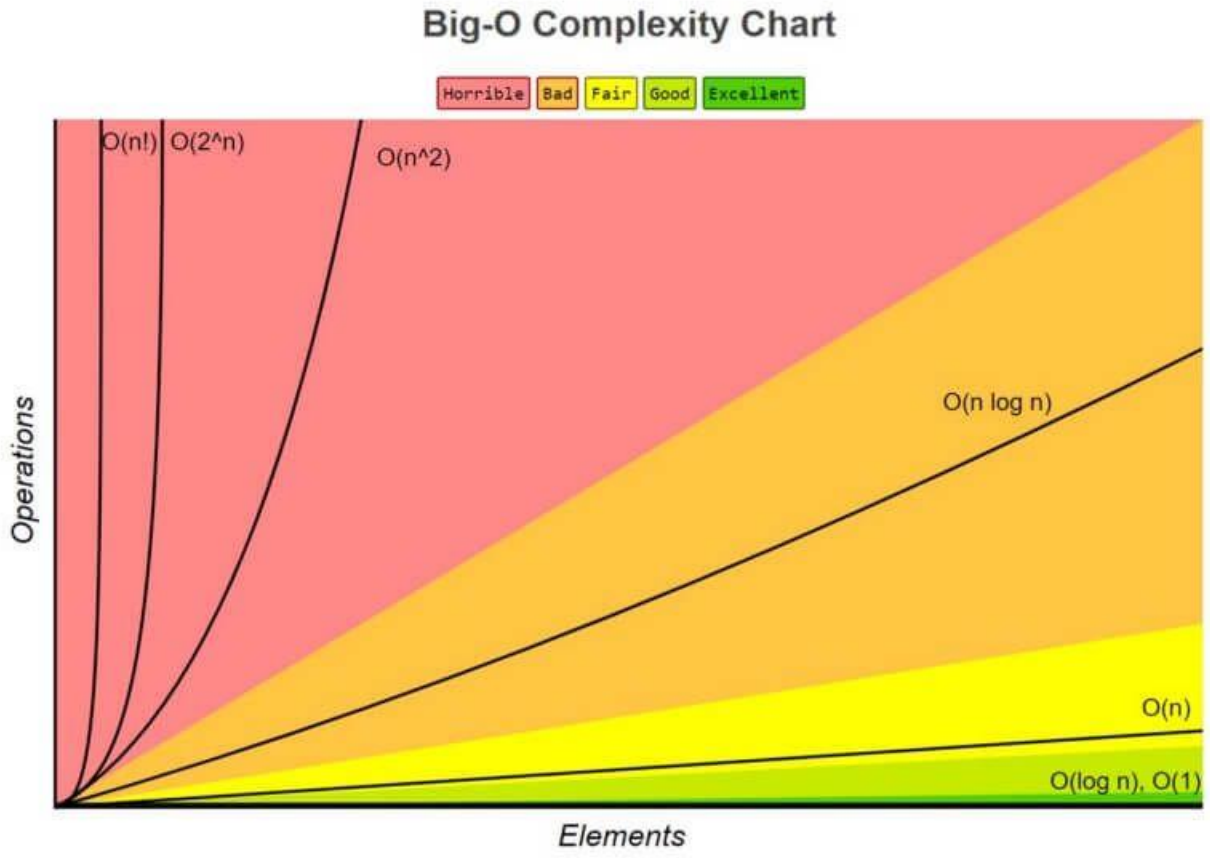
The common algorithmic runtimes from fastest to slowest are:

1. *constant: Θ(1)*
2. *logarithmic: Θ(log N)*
3. *linear: Θ(N)*
4. *polynomial: $\Theta(N^2)$*
5. *exponential: $\Theta(2^N)$*
6. *factorial: Θ(N!)*

x –axis : input of n values
y –axis : execution time of given input

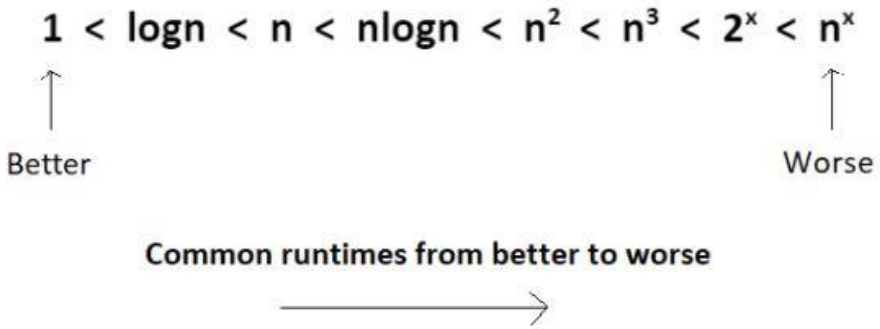


Big-oh (O) Complexity Chart



Increasing order of common runtimes

Below mentioned are some common runtimes which you will come across in your coding career



Brute Force Algorithm Implementations

REVATURE

Brute force allgorithm is an intuitive, direct, and straightforward technique of problem-solving in which all the possible ways or all the possible solutions to a given problem are enumerated.

Pros:

- *The brute force approach is a guaranteed way to find the correct solution by listing all the possible candidate solutions for the problem.*
- *It is a generic method and not limited to any specific domain of problems.*
- *The brute force method is ideal for solving small and simpler problems.*
- *It is known for its simplicity and can serve as a comparison benchmark.*

Cons:

- *The brute force approach is inefficient. For real-time problems, algorithm analysis often goes above the O(N!) order of growth.*
- *This method relies more on compromising the power of a computer system for solving a problem than on a good algorithm design.*
- *Brute force algorithms are slow.*
- *Brute force algorithms are not constructive or creative compared to algorithms that are constructed using some other design paradigms.*

Brute force algorithm is a technique that guarantees solutions for problems of any domain helps in solving the simpler problems and also provides a solution that can serve as a benchmark for evaluating other design techniques, but takes a lot of run time and inefficient.

Brute Force Algorithm Implementations

REVATURE

For Example:
If there is a lock of 4-digit PIN. The digits to be chosen from 0-9 then the brute force will be trying all possible combinations one by one like 0001, 0002, 0003, 0004, and so on until we get the right PIN.
In the worst case, it will take 10,000 tries to find the right combination.

Brute Force Algorithm Demo

REVATURE

Problem 1: Given a array A (not sorted, non duplicate values), having N integers, find all pairs of elements (A[i], A[j]) such that their sum is equal to X.

Solution:
We can use Two Pointers Technique or HashMap- Two pointers is really an easy and effective technique that is typically used for searching pairs in a sorted array.
Check the solution code in shared eclipse project.

Problem 2: Solve the Hackrank question from shared list.

Amortized Analysis

REVATURE

https://www.geeksforgeeks.org/analysis-algorithm-set-5-amortized-analysis-introduction/

Recursive Algorithm

REVATURE

Demo using Fibonacci series
Demo using Factorial of given number

Conversion Algorithm

REVATURE

Demo using infix-postfix, infix-prefix
Demo using evaluation of postfix/prefix expression

Divide and Conquer Algorithm

REVATURE

This technique can be divided into the following three parts:

***Divide**: This involves dividing the problem into smaller sub-problems.*

***Conquer**: Solve sub-problems by calling recursively until solved.*

***Combine**: Combine the sub-problems to get the final solution of the whole problem.*

The divide-and-conquer paradigm is often used to find an optimal solution of a problem. Its basic idea is to decompose a given problem into two or more similar, but simpler, subproblems, to solve them in turn, and to compose their solutions to solve the given problem.

The divide-and-conquer technique is the basis of efficient algorithms for many problems, such as

1. *Binary Search: The binary search algorithm is a searching algorithm, which is also called a half-interval search or logarithmic search.*
2. *Sorting (e.g. merge sort),*
3. *finding the closest pair of points- It is a problem of computational geometry. This algorithm emphasizes finding out the closest pair of points in a metric space, given n points, such that the distance between the pair of points should be minimal.*
4. *syntactic analysis (e.g., top-down parsers)*

Divide and Conquer Algorithm

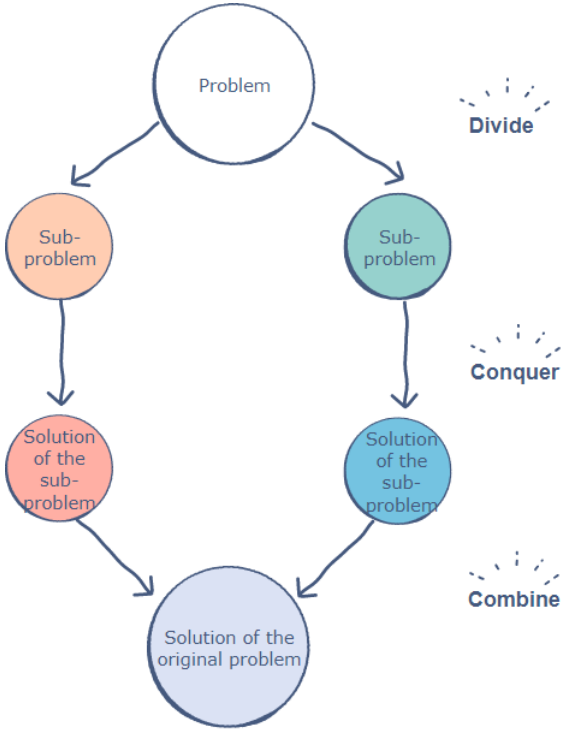
REVATURE

Advantage

1. *Solves difficult problems with less time complexity than its brute-force counterpart.*
2. *Since the sub-problems are independent, they can be computed in parallel.*

Disadvantage

1. *Includes recursion which consumes more space.*
2. *It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.*



Searching Algorithm

REVATURE

Searching is the process of finding some particular element in the list.
If the element is present in the list, then the process is called successful, and the process returns the location of that element; otherwise, the search is called unsuccessful.

Two popular search methods are Linear Search and Binary Search.

- ***Linear Search***
- ***Binary Search***

They can be implemented using

- *Iterative method i.e. looping constructs*
- *Recursive method i.e using recursion logic*

Linear Searching Algorithm

REVATURE

- *Linear search is also called as sequential search algorithm.*
- *It is the simplest searching algorithm.*
- *In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found.*
- *If the match is found, then the location of the item is returned; otherwise, the algorithm returns -1 (or null).*
- *It is used to search an element from the unordered list, i.e., the list in which items are not sorted.*
- *Time complexity:*
    1. ***Best Case Time Complexity** - In Linear search, best case occurs when the element we are finding is at the first position of the array. The best-case time complexity of linear search is **O(1) or Ω(1)** .*
    2. ***Average Case Time Complexity** - The average case time complexity of linear search is **O(n) or θ(n)**.*
    3. ***Worst Case Time Complexity** - In Linear search, the worst case occurs when the element we are looking is present at the end of the array. The worst-case in linear search could be when the target element is not present in the given array, and we have to traverse the entire array. The worst-case time complexity of linear search is **O(n)**.*

NOTE:

- *It is practical to implement linear search in the situations mentioned in When the list has only a few elements and When performing a single search in an unordered list, but for larger elements the complexity becomes larger and it makes sense to sort the list and employ binary search or hashing.*
- *Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster-searching comparison to Linear search.*

Binary Searching Algorithm

REVATURE

- *Binary search is the search technique that works efficiently on sorted lists.*
- *Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted, if the list elements are not arranged in a sorted manner, we must first need to sort them.*
- *Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list.*
- *If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.*
- *Time complexity:*
    1. ***Best Case Time Complexity** - In Binary search, best case occurs when the element to search is found in first comparison, i.e., when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is **O(1)**.*
    2. ***Average Case Time Complexity** - The average case time complexity of Binary search is **O(log n)**.*
    3. ***Worst Case Time Complexity** - In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is **O(log n)**.*

Searching Algorithm

Demo Linear Search
Demo Binary Search

Sorting Algorithm (Divide & Conquer)

REVATURE

When to use which sorting algorithm : https://www.geeksforgeeks.org/when-to-use-each-sorting-algorithms/
Bubble Sort: https://www.javatpoint.com/bubble-sort
Merge Sort: https://www.javatpoint.com/merge-sort
To sort a given list of n natural numbers, split it into two lists of about n/2 numbers each, sort each of them in turn, and interleave both results appropriately to obtain the sorted version of the given list (see the picture). This approach is known as the merge sort algorithm.
Complexity Comparison:
https://en.wikipedia.org/wiki/Sorting_algorithm
https://www.javatpoint.com/time-complexity-of-sorting-algorithms
https://www.interviewbit.com/tutorial/sorting-algorithms/

Time Complexities of Sorting Algorithms:

Algorithm	Best	Average	Worst
Quick Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)
Merge Sort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)

Sorting Algorithm Demo

REVATURE

https://www.baeldung.com/cs/choose-sorting-algorithm

Demo on Bubble Sort (Non-recursive Approach)
Demo on Merge Sort (Using Recursive and Non-recursive Approach)

Exercise:
Given two sorted arrays of size m and n respectively, you are tasked with finding the element that would be at the k'th position of the final sorted array.

Examples:

Input : Array 1 - 2 3 6 7 9
Array 2 - 1 4 8 10
k = 5
Output : 6
Explanation: The final sorted array would be -
1, 2, 3, 4, 6, 7, 8, 9, 10
The 5th element of this array is 6.

Input : Array 1 - 100 112 256 349 770
Array 2 - 72 86 113 119 265 445 892
k = 7
Output : 256
Explanation: Final sorted array is -
72, 86, 100, 112, 113, 119, 256, 265, 349, 445, 770, 892
7th element of this array is 256.

Sorting Algorithm

REVATURE

Quick Sort	Compare to all other sorting algorithms quick sort is fastest and no additional memory is required. It gives worst case response time for the critical applications which require guaranteed response time. Applications such as life monitoring in medical sector, aircraft controlling, monitoring of dangerous materials on industrial plants etc.
Merge Sort	In most of the e-commerce applications Merge Sort is used.
Bubble Sort	Efficient for sorted input data
Insertion Sort, Selection Sort, Bubble Sort	Not require extra memory space for sorting process.
Quick Sort	To make excellent usage of the memory hierarchy like virtual memory or caches. Well suited to modern computer architectures.
Counting Sort	Counting sort is more efficient if the range of input data is smaller (1 to k) than the number of data (1 to n) to be sorted. i.e. k<=n

Sorting Algorithm

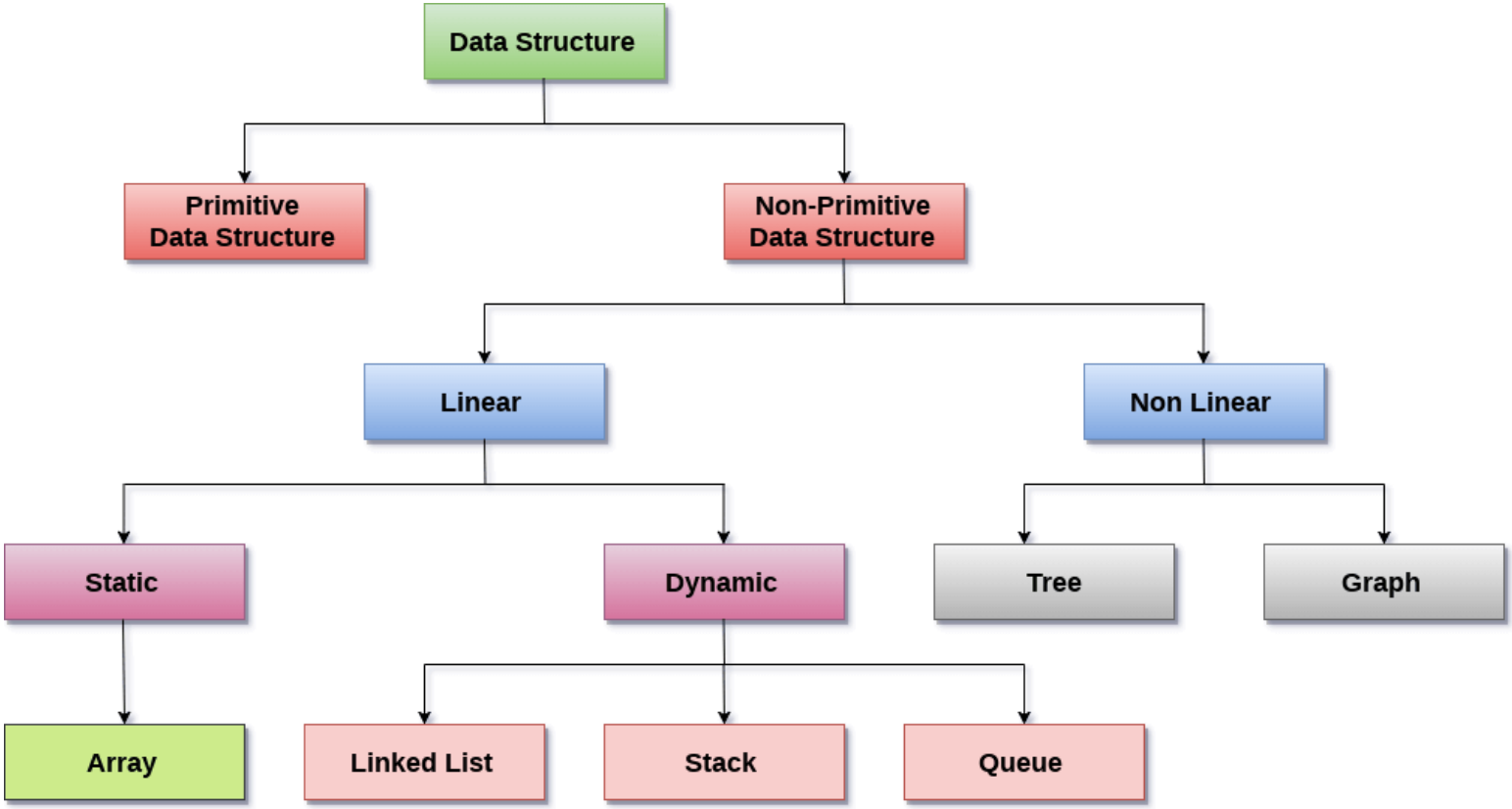
REVATURE

TABLE 1: PERFORMANCE CHARACTERISTICS OF SORTING TECHNIQUES [1, 2, 3, 8, 10]

Technique	Approach	Data Structure	Time Complexity n : number of input elements			Worst case Space Complexity	Stable	In place
			Best Case	Average Case	Worst Case			
Bubble Sort	Comparison Based	Array	O(n2)	O(n2)	O(n2)	O(1) auxiliary	Yes	Yes
Modified Bubble Sort	Comparison Based	Array	O(n)	O(n2)	O(n2)	O(1) auxiliary	Yes	Yes
Selection Sort	Comparison Based	Array	O(n2)	O(n2)	O(n2)	O(n) total, O(1) auxiliary	Yes	Yes
Enhanced Selection Sort	Comparison Based	Array	O(n2)	O(n2)	O(n2)	O(n) total, O(1) auxiliary	Yes	Yes
Insertion sort	Comparison Based	Array	O(n)	O(n2)	O(n2)	O(n) total, O(1) auxiliary	Yes	Yes
Merge Sort	Divide and Conquer	Array	O(nlogn)	O(nlogn)	O(nlogn)	O(n) total, O(n) auxiliary	Yes	No
Quick sort	Divide and Conquer	Array	O(nlogn)	O(nlogn)	O(n2)	O(n) auxiliary	Yes	Yes
Parallel Quick sort	Divide and Conquer	Array	O(nlogn)	O(nlogn)	O(nlogn)	O(n) auxiliary	Yes	Yes
Randomized Quick Sort	Divide and Conquer	Array	O(nlogn)	O(nlogn)	O(nlogn)	O(n) auxiliary	Yes	Yes
Hyper Quick sort	Divide and Conquer	Array	O(nlogn)	O(nlogn)	O(nlogn)	O(n) auxiliary	Yes	Yes
MQ sort (Combination of Merge and Quick Sort)	Divide and Conquer	Array	O(nlogn)	O(nlogn)	O(nlogn)	O(n) auxiliary	Yes	Yes
Radix Sort k : range of data elements	Non Comparison Based	Array, Queue	O(kN)	O(kN)	O(kN)	O(n+ k)	No	No
Counting Sort k : range of data elements	Non Comparison Based	Array	O(n+k) ≈ O(n)	O(n+k) ≈ O(n)	O(n+k) ≈ O(n)	O(n+k)	Yes	No
Heap Sort		Array, Tree	O(nlogn)	O(nlogn)	O(nlogn)	O(1) auxiliary	NO	Yes
Bucket Sort k : number of buckets	Non Comparison Based	Array	O(n+k)	O(n+k)	O(n2)	O(n.k)	Yes	No

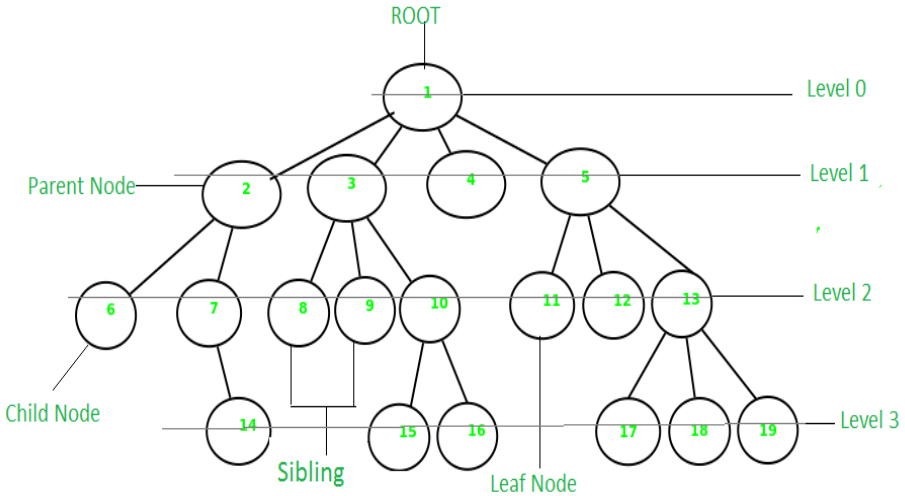
Non-Linear Data structures

REVATURE



Tree

A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value, a list of references to nodes (the "children")
The tree data structure is a kind of hierarchal data arranged in a tree-like structure.
It consists of a central node, structural nodes, and sub nodes, which are connected via edges.
We can also say that tree data structure has roots, branches, and leaves connected with one another.



Basic Terminology In Tree Data Structure

REVATURE

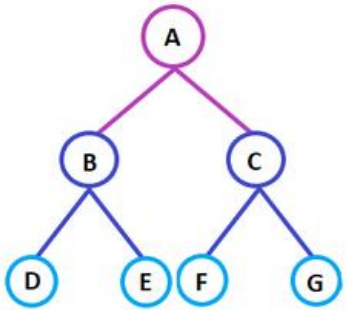
- ***Parent Node**: The node which is a predecessor of a node is called the parent node of that node. {2} is the parent node of {6, 7}.*
- ***Child Node**: The node which is the immediate successor of a node is called the child node of that node. Examples: {6, 7} are the child nodes of {2}.*
- ***Root Node**: The topmost node of a tree or the node which does not have any parent node is called the root node. {1} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.*
- ***Degree of a Node**: The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be 0. The degree of a tree is the degree of its root. The degree of the node {3} is 3.*
- ***Leaf Node or External Node**: The nodes which do not have any child nodes are called leaf nodes. {6, 14, 8, 9, 15, 16, 4, 11, 12, 17, 18, 19} are the leaf nodes of the tree.*
- ***Ancestor of a Node**: Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {1, 2} are the parent nodes of the node {7}*
- ***Descendant**: Any successor node on the path from the leaf node to that node. {7, 14} are the descendants of the node. {2}.*
- ***Sibling**: Children of the same parent node are called siblings. {8, 9, 10} are called siblings.*
- ***Depth of a node**: The count of edges from the root to the node. Depth of node {14} is 3.*
- ***Height of a node**: The number of edges on the longest path from that node to a leaf. Height of node {3} is 2.*
- ***Height of a tree**: The height of a tree is the height of the root node i.e the count of edges from the root to the deepest node. The height of the above tree is 3.*
- ***Level of a node**: The count of edges on the path from the root node to that node. The root node has level 0.*
- ***Internal node**: A node with at least one child is called Internal Node.*
- ***Neighbour of a Node**: Parent or child nodes of that node are called neighbors of that node.*
- ***Subtree**: Any node of the tree along with its descendant*

Basic Terminology In Tree Data Structure

REVATURE

Here,

- *Node A is the root node*
- *B is the parent of D and E*
- *D and E are the siblings*
- *D, E, F and G are the leaf nodes*
- *A and B are the ancestors of E*



Why Trees?

REVATURE

1. *One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer*
2. *Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).*
3. *Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).*
4. *Like Linked Lists and unlike Arrays, Trees don't have an upper limit on number of nodes as nodes are linked using pointers.*

Main applications of trees include:

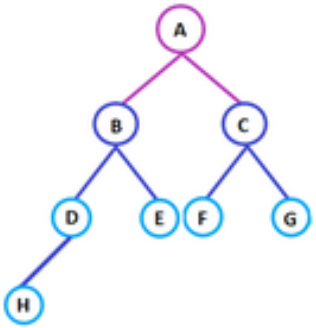
1. *Manipulate hierarchical data.*
2. *Make information easy to search (tree traversal).*
3. *Manipulate sorted lists of data.*
4. *Router algorithms*
5. *Form of a multi-stage decision-making*

Types of Tree data structures

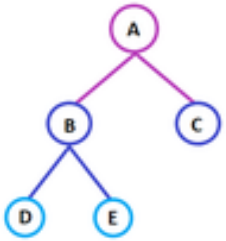
REVATURE

The different types of tree data structures are as follows:

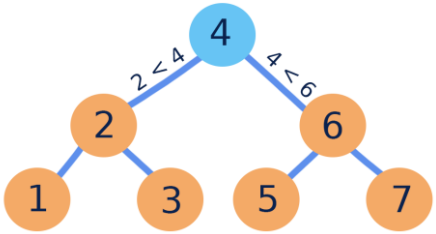
1. *General tree: A general tree data structure has no restriction on the number of nodes. It means that a parent node can have any number of child nodes.*
2. *Binary tree: A node of a binary tree can have maximum number of two child nodes. In the given tree diagram, node B, D, and F are left children, while E, C, and G are the right children.*
3. *Balanced tree: If the height of the left sub-tree and the right sub-tree are equal or differs at most by 1, the tree is known as balanced tree data structure.*
4. *Binary Search Tree: As the name implies, binary search trees are used for various searching and sorting algorithms. It shows that the value of the left node is less than its parent, while the value of right node is greater than its parent.*



Balanced tree



Unbalanced tree



BinaryTree Data Structure

REVATURE

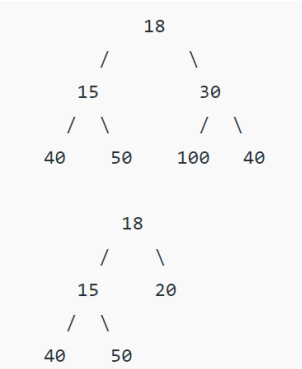
```
1.  /* Class containing left and right child of current node and key value*/
2.  class Node
3.  {
4.      int key;
5.      Node left, right;

6.      public Node(int item)
7.      {
8.          key = item;
9.          left = right = null;
10.     }
11. }
```

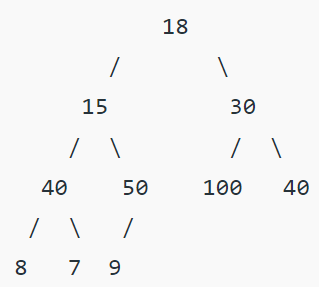
BinaryTree Types

The following are common types of Binary Trees.

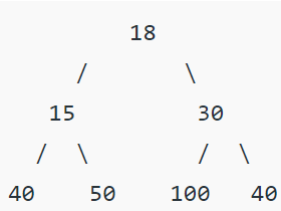
1. ***Full Binary Tree:*** *A Binary Tree is a full binary tree if every node has 0 or 2 children. The following are the examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.*
2. ***Complete Binary Tree****: A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible*
3. ***Perfect Binary Tree:*** *A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.*
4. ***A degenerate (or pathological) tree:*** *A Tree where every internal node has one child. Such trees are performance-wise same as linked list.*



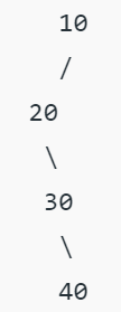
1



2



3



4

BinaryTree Types

REVATURE

5. *Binary Search tree (BST)*
6. *Balanced Binary Tree:*
    - *A binary tree is balanced if the height of the tree is O(Log n) where n is the number of nodes.*
    - *For Example, the AVL tree maintains O(Log n) height by making sure that the difference between the heights of the left and right subtrees is at most 1.*
    - *Red-Black trees maintain O(Log n) height by making sure that the number of Black nodes on every root to leaf paths is the same and there are no adjacent red nodes.*
    - *Balanced Binary Search trees are performance-wise good as they provide O(log n) time for search, insert and delete.*
7. *Height Balance tree (AVL) - AVL tree is a self-balancing Binary Search Tree (BST)*
   ***AVL** is Adelson-Velskii and Landis*
5. *Red-Black tree*

https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm
https://www.geeksforgeeks.org/avl-tree-set-1-insertion/
https://favtutor.com/blogs/avl-tree-python#:~:text=The%20full%20form%20of%20an,path%20of%20its%20right%20subtree

Tree Applications

REVATURE

The applications of tree data structures are as follows:

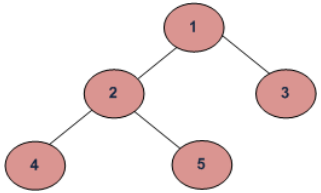
- *Spanning trees- It is the shortest path tree used in the routers to direct the packets to the destination.*
- *Binary Search Tree - It is a type of tree data structure that helps in maintaining a sorted stream of data.*
- *Storing hierarchical data - Tree data structures are used to store the hierarchical data, which means data arranged in the form of order.*
- *Syntax tree- The syntax tree represents the structure of the program's source code, which is used in compilers.*
- *Trie- It is the fast and efficient way for dynamic spell checking. It is also used for locating specific keys from within a set.*
- *Heap- It is also a tree data structure that can be represented in a form of array. It is used to implement priority queues.*

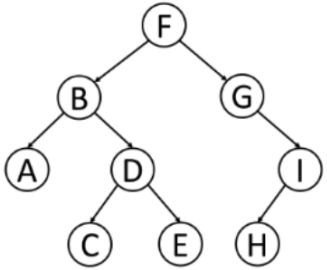
Tree Traversals (Inorder, Preorder and Postorder)

REVATURE

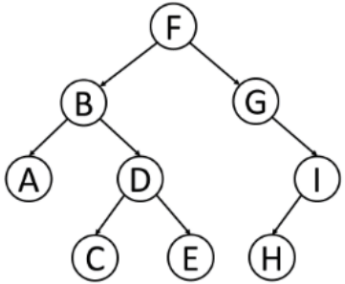
Depth First Traversals:
(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth-First or Level Order Traversal: 1 2 3 4 5

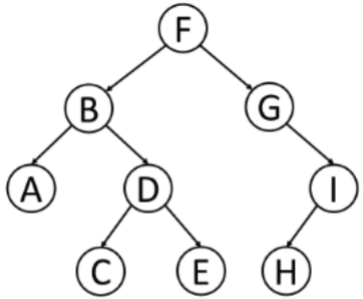




Preorder:



Inorder:



Postorder:

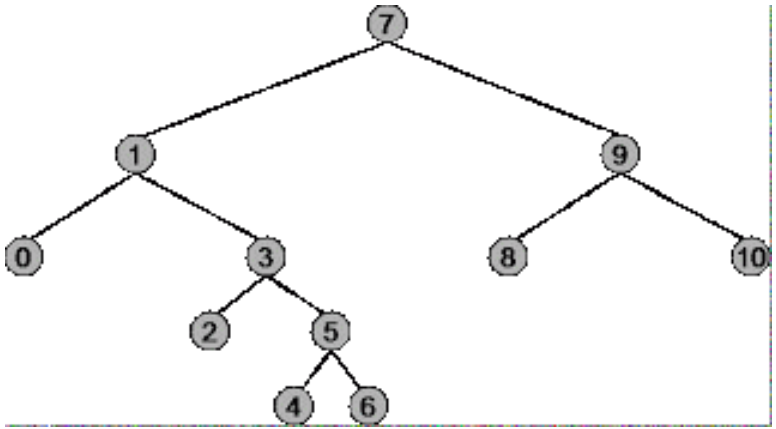
Tree Traversals (Inorder, Preorder and Postorder)

REVATURE

Pre-order traversal:
Summary: Begins at the root (7), ends at the right-most node (10)
Traversal sequence: 7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10

In-order traversal:
Summary: Begins at the left-most node (0), ends at the rightmost node (10)
Traversal Sequence: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Post-order traversal:
Summary: Begins with the left-most node (0), ends with the root (7)
Traversal sequence: 0, 2, 4, 6, 5, 3, 1, 8, 10, 9, 7



Algorithm Pre-Order, In-order or Post-Order?

REVATURE

Algorithm Inorder(tree)

1. *Traverse the left subtree, i.e., call Inorder(left-subtree)*
2. *Visit the root.*
3. *Traverse the right subtree, i.e., call Inorder(right-subtree)*

Algorithm Preorder(tree)

1. *Visit the root.*
2. *Traverse the left subtree, i.e., call Preorder(left-subtree)*
3. *Traverse the right subtree, i.e., call Preorder(right-subtree)*

Algorithm Postorder(tree)

1. *Traverse the left subtree, i.e., call Postorder(left-subtree)*
2. *Traverse the right subtree, i.e., call Postorder(right-subtree)*
3. *Visit the root.*

When to use Pre-Order, In-order or Post-Order?

REVATURE

The traversal strategy the programmer selects depends on the specific needs of the algorithm being designed. The goal is speed, so pick the strategy that brings you the nodes you require the fastest.

pre-order:

- *If you know you need to explore the roots before inspecting any leaves, you pick pre-order because you will encounter all the roots before all of the leaves.*
- *Used to create a copy of a tree. For example, if you want to create a replica of a tree, put the nodes in an array with a pre-order traversal. Then perform an Insert operation on a new tree for each value in the array. You will end up with a copy of your original tree.*

post-order:

- *If you know you need to explore all the leaves before any nodes, you select post-order because you don't waste any time inspecting roots in search for leaves.*
- *Used to delete a tree from leaf to root*

in-order:

- *If you know that the tree has an inherent sequence in the nodes, and you want to flatten the tree back into its original sequence, than an in-order traversal should be used. The tree would be flattened in the same way it was created. A pre-order or post-order traversal might not unwind the tree back into the sequence which was used to create it.*
- *Used to get the values of the nodes in non-decreasing (sorted/ascending) order in a BST.*

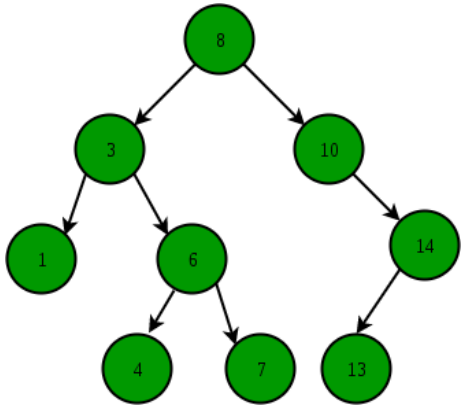
Binary Search Tree (BST)

REVATURE

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- *The left subtree of a node contains only nodes with keys lesser than the node's key.*
- *The right subtree of a node contains only nodes with keys greater than the node's key.*
- *The left and right subtree each must also be a binary search tree. There must be no duplicate nodes.*

The above properties of Binary Search Tree provides an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search for a given key.



Binary Search Tree (BST)

REVATURE

Demo code

AVL Trees

REVATURE

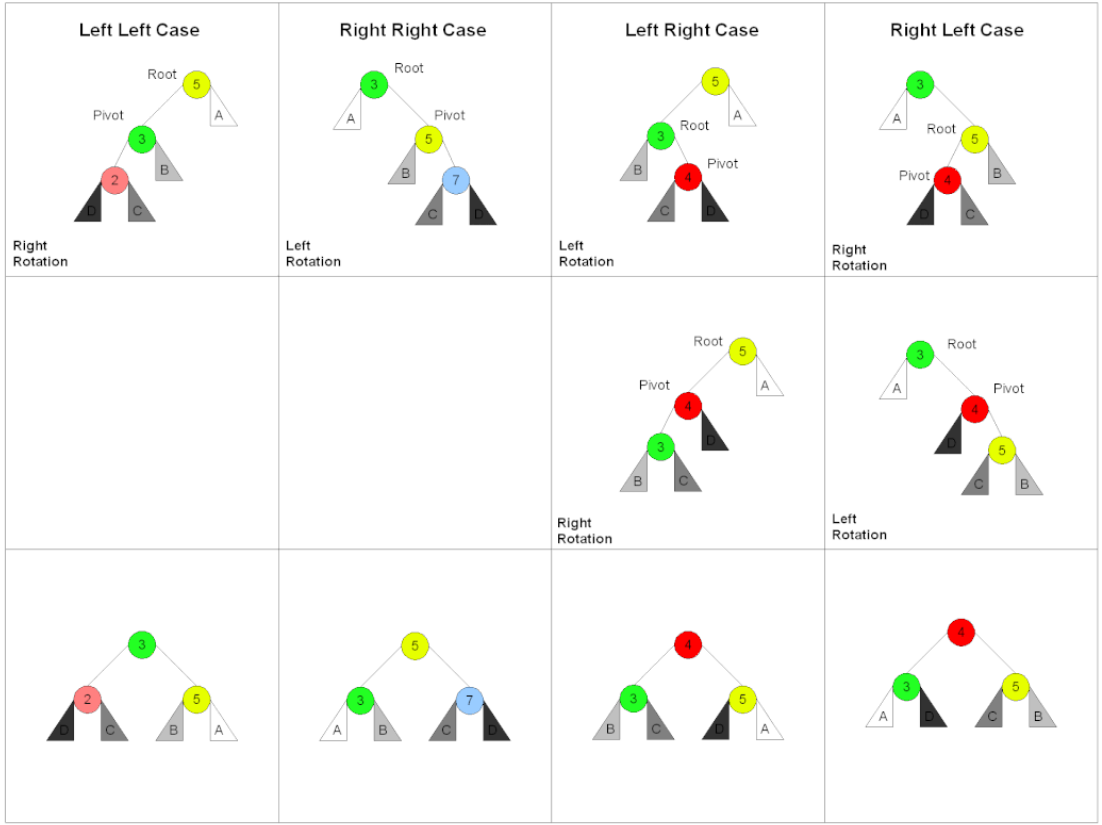
- *The name AVL tree is derived after its two creators, i.e. G.M. Abelson-Velvety and E.M. Landis. AVL tree is a height-balanced binary tree where a balance factor balances each node. A balancing factor is a difference between the height of the left subtree and the right subtree. For a node to be balanced, it should be -1, 0, or 1.*
- *The performance of the lookup time depends on the shape of the tree. In the case of improper organization of nodes, forming a list, the process of searching for a given value can be the O(n) operation. With a correctly arranged tree, the performance can be significantly improved to O(log n).*

- *https://www.javatpoint.com/avl-tree*
- *https://www.simplilearn.com/tutorials/data-structure-tutorial/avl-tree-in-data-structure*

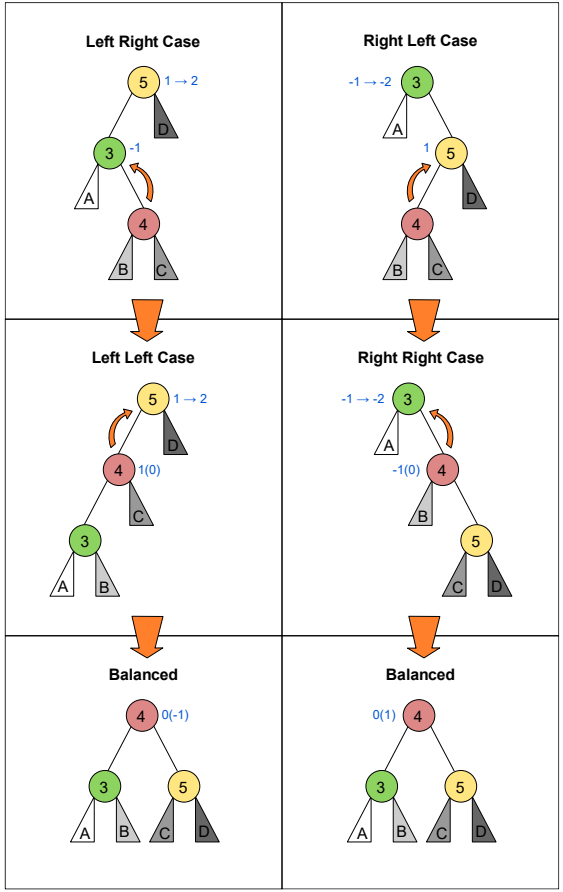
AVL Rotations

There are 4 cases in all, choosing which one is made by seeing the direction of the first 2 nodes from the unbalanced node to the newly inserted node and matching them to the top most row.

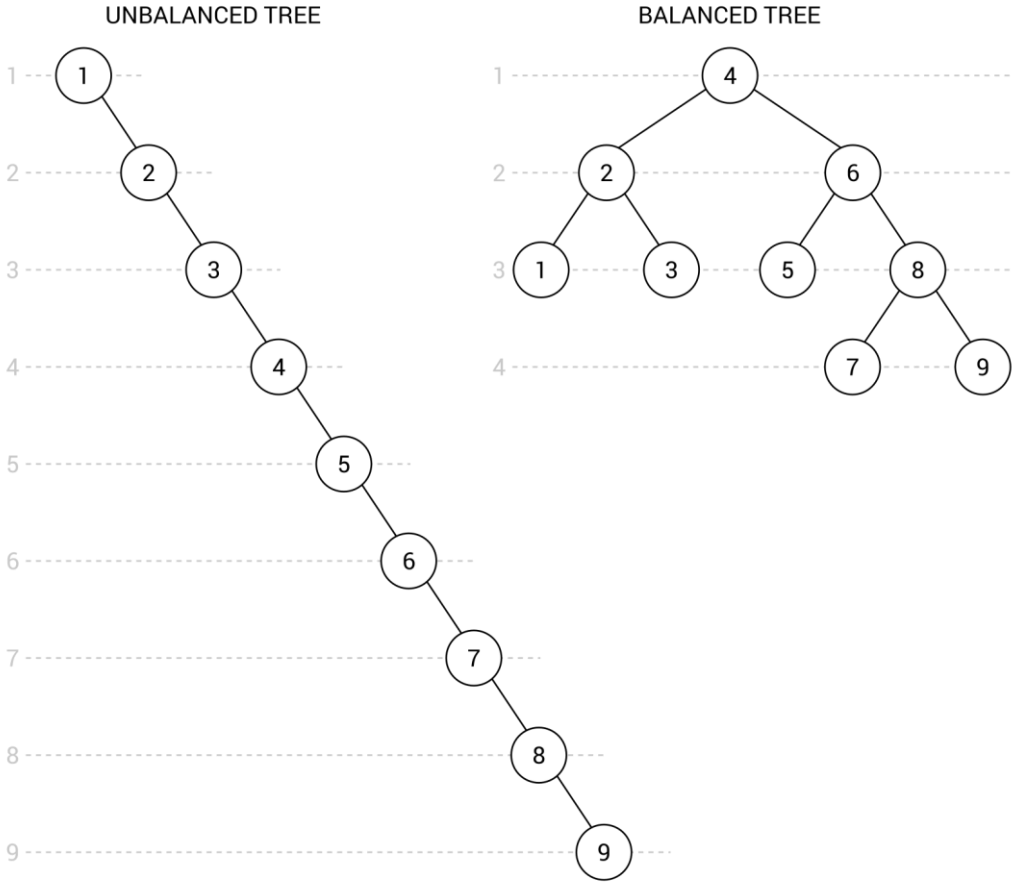
Root is the initial parent before a rotation and **Pivot** is the child to take the root's place.



AVL Rotations



Unbalanced vs Balanced Tree



Balanced Tree

REVATURE

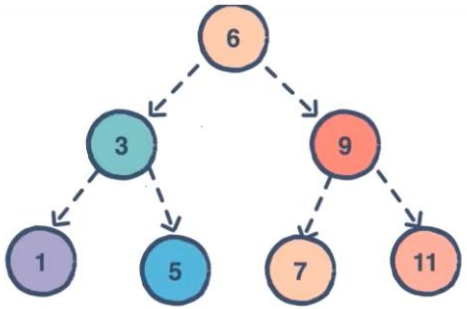
Height Balanced Tree Code

Advantages of BST over Hash Table

REVATURE

- *Hash Table supports following operations in Θ(1) time.*
    - *1) Search*
    - *2) Insert*
    - *3) Delete*
- *The time complexity of above operations in a self-balancing Binary Search Tree (BST) (like Red-Black Tree, AVL Tree, Splay Tree, etc) is O(Logn).*
- *So Hash Table seems to beating BST in all common operations. When should we prefer BST over Hash Tables, what are advantages. Following are some important points in favor of BSTs.*
    1. *We can get all keys in sorted order by just doing Inorder Traversal of BST. This is not a natural operation in Hash Tables and requires extra efforts.*
    2. *Doing order statistics, finding closest lower and greater elements, doing range queries are easy to do with BSTs. Like sorting, these operations are not a natural operation with Hash Tables.*
    3. *BSTs are easy to implement compared to hashing, we can easily implement our own customized BST. To implement Hashing, we generally rely on libraries provided by programming languages.*
    4. *With Self-Balancing BSTs, all operations are guaranteed to work in O(Logn) time. But with Hashing, Θ(1) is average time and some particular operations may be costly, especially when table resizing happens.*

Java HashMap 1.8 Changes



An example of a balanced tree

As we know now that in case of hash collision entry objects are stored as a node in a linked-list and equals() method is used to compare keys. That comparison to find the correct key with in a linked-list is a linear operation so in a worst case scenario the complexity becomes O(n).

JDK 8 HashMap Changes

To address this issue, Java 8 hash elements use balanced trees instead of linked lists after a certain threshold is reached. Which means HashMap starts with storing Entry objects in linked list but after the number of items in a hash becomes larger than a certain threshold, the hash will change from using a linked list to a balanced tree, which will improve the worst case performance from O(n) to O(log n).

`LinkedList` `Balanced BST`

- **Important Points**

1. Time complexity is almost constant for put and get method until rehashing is not done.
2. In case of collision, i.e. index of two or more nodes are same, nodes are joined by link list i.e. second node is referenced by first node and third by second and so on.
3. If key given already exist in HashMap, the value is replaced with new value.
4. hash code of null key is 0.
5. When getting an object with its key, the linked list is traversed until the key matches or null is found on next field.

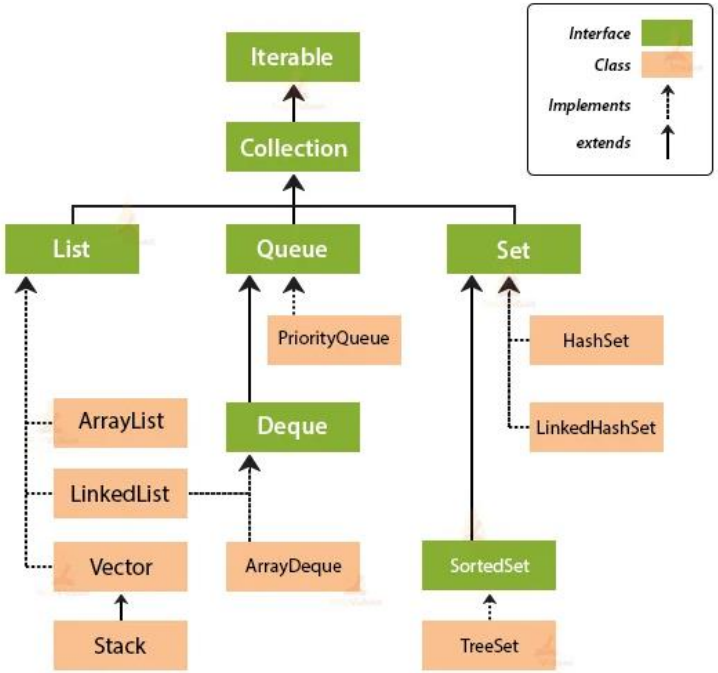
Performance Improvement for HashMap in Java 8

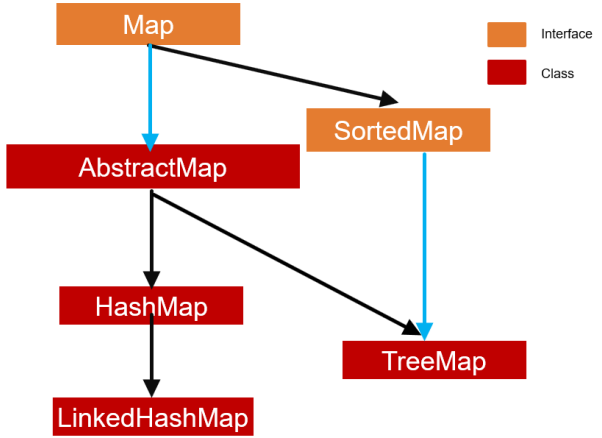
REVATURE

https://www.nagarro.com/en/blog/post/24/performance-improvement-for-hashmap-in-java-8#:~:text=In%20Java%208%2C%20HashMap%20replaces,used%20as%20a%20branching%20variable.

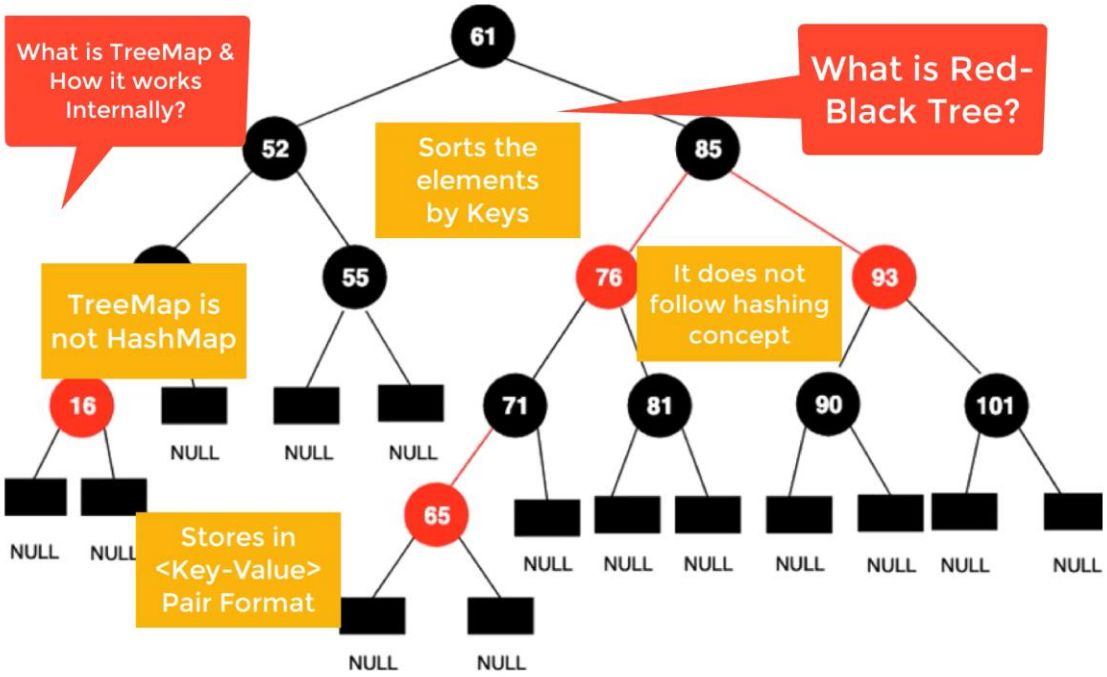
After 8 nodes into same index of LinkedList (i.e. 8 collisions) it will be converted to Balanced BST

Java TreeMap

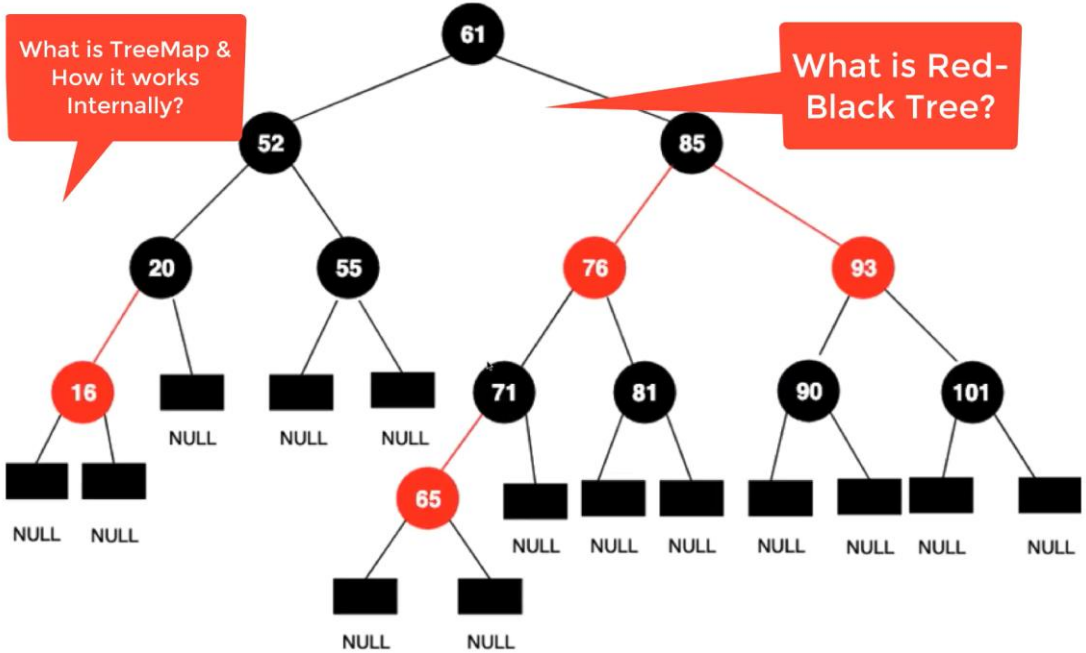




Java TreeMap

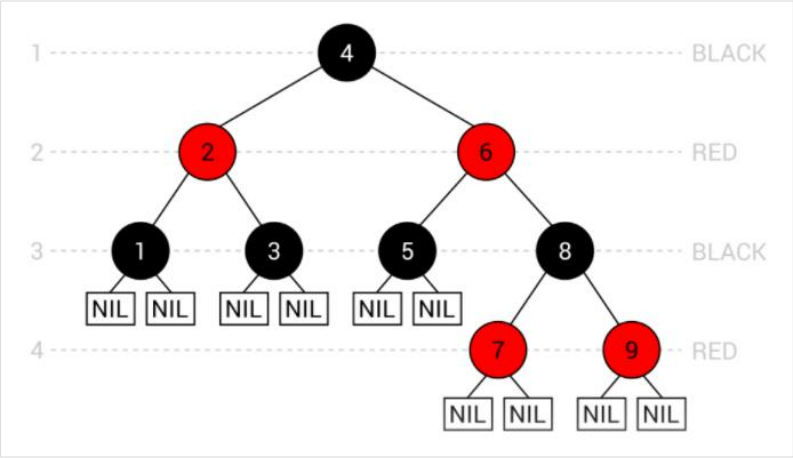


Java TreeMap



Red-Black Trees

- *A Red-black tree, also referred to as an RBT, is the next variant of the self-balancing binary search trees. As a variant of BSTs, this data structure requires that the standard BST rules be maintained. Moreover, the following rules must be taken into account:*



red-black tree

1. A node is either red or black.
2. The root and leaves (NIL) are black.
3. If a node is red, then its children are black.
4. All paths from a node to its NIL descendants contain the same number of black nodes.

"The inserted new node color is always black if its root we make it black and if unbalanced we change color to ensure above 4 rules are met."

Implementation Code:

- *TreeMap in jdk*
- *https://www.javatpoint.com/red-black-tree-java*
- *https://www.happycoders.eu/algorithms/red-black-tree-java/*

Red-Black Trees vs AVL Trees

REVATURE

Difference:

- *AVL trees provide faster lookups than Red Black Trees because they are more strictly balanced.*
- *Red Black Trees provide faster insertion and removal operations than AVL trees as fewer rotations are done due to relatively relaxed balancing.*
- *AVL trees store balance factors or heights with each node, thus requires storage for an integer per node whereas Red Black Tree requires only 1 bit of information per node.*
- *Red Black Trees are used in most of the language libraries like map, multimap, multiset in C++ whereas AVL trees are used in databases where faster retrievals are required.*

Number of Rotations in AVL > Number of Rotations in Red-Black Tree

Where to use:

- *AVL Tree – Where no much Insertion/Deletion*
- *RED BLACK – Where we have more Insertion/Deletion*

Graph

REVATURE

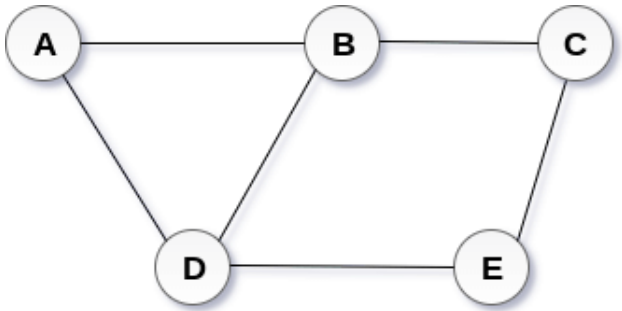
- *A graph can be defined as group of vertices and edges that are used to connect these vertices.*
- *A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.*

- *A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.*
- *A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.*

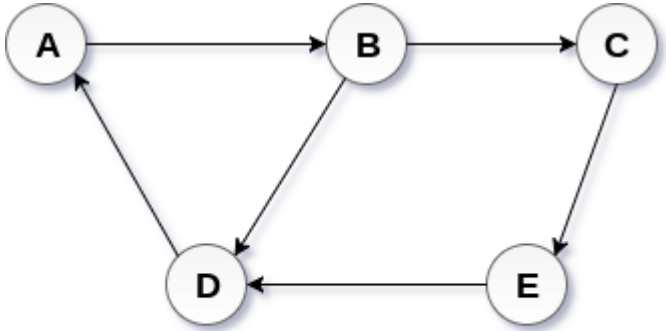
Graph Types

REVATURE

- *In Undirected graph If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.*
- *In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node*



Undirected Graph

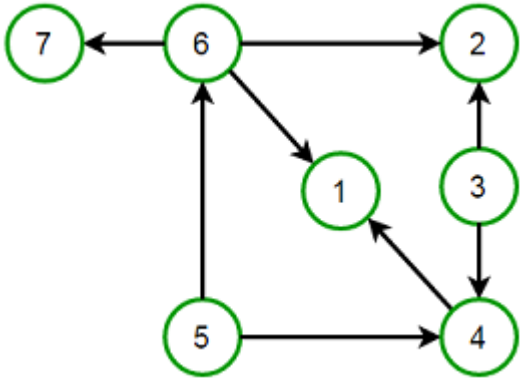


Directed Graph

Directed Acyclic Graph (DAG)

REVATURE

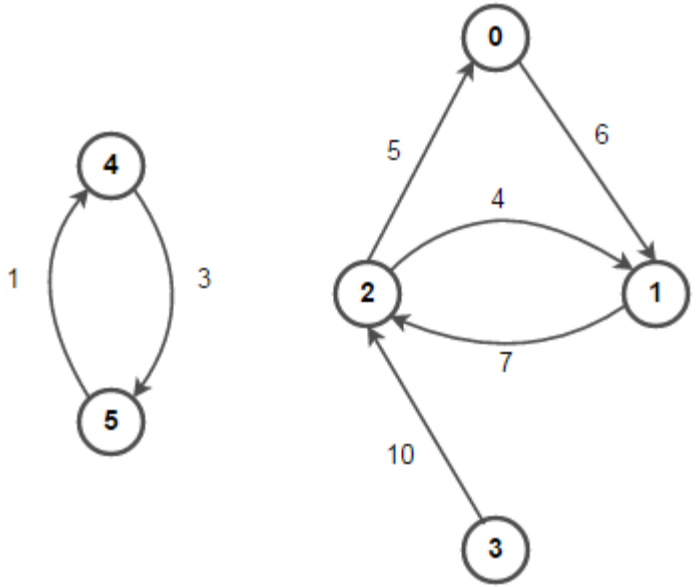
- *A Directed Acyclic Graph (DAG) is a directed graph that contains no cycles.*



Weighted and Unweighted graph

REVATURE

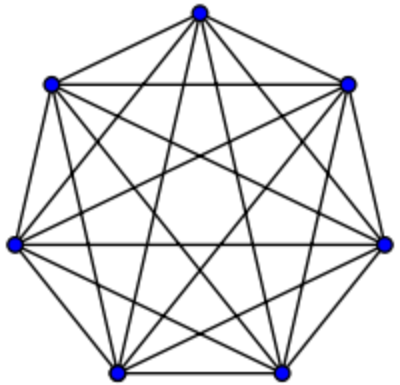
- *A weighted graph associates a value (weight) with every edge in the graph. We can also use words cost or length instead of weight.*
- Unless specified otherwise, all graphs are assumed to be unweighted by default.



Complete graph

REVATURE

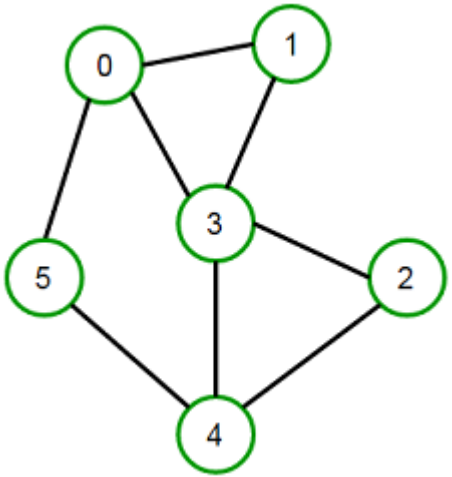
- *A complete graph is one in which every two vertices are adjacent: all edges that could exist are present.*



Connected graph

REVATURE

- *A Connected graph has a path between every pair of vertices. In other words, there are no unreachable vertices. A disconnected graph is a graph that is not connected.*



Graph Representations

REVATURE

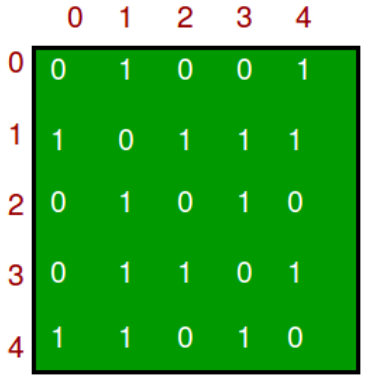
The following two are the most commonly used representations of a graph.

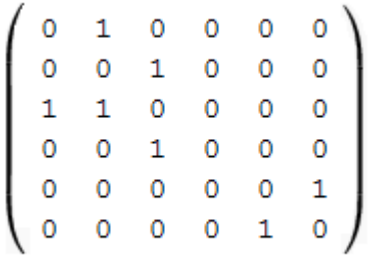
1. *Adjacency Matrix*
2. *Adjacency List*

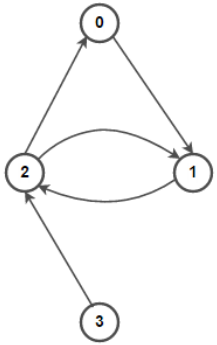
Adjacency Matrix:

- *Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.*

***Pros**: Representation is easier to implement and follow.*
***Cons**: Consumes more space O(V^2).*





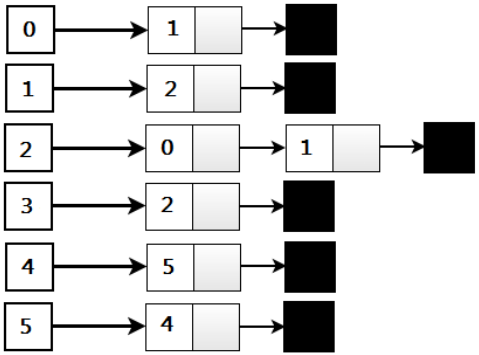


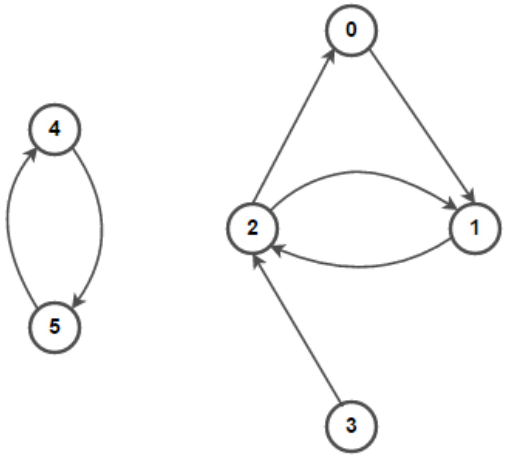
Graph Representations

REVATURE

2. Adjacency List:

An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.





Most commonly used terms in Graphs

REVATURE

- *An **edge** is (together with vertices) one of the two basic units out of which graphs are constructed. Each edge has two vertices to which it is attached, called its endpoints.*
- *Two vertices are called **adjacent** if they are endpoints of the same edge.*
- ***Outgoing edges** of a vertex are directed edges that the vertex is the origin.*
- ***Incoming edges** of a vertex are directed edges that the vertex is the destination.*
- *In a directed graph, the **out-degree** of a vertex is the total number of outgoing edges, and the **in-degree** is the total number of incoming edges.*
- *A vertex with in-degree zero is called a **source vertex**, while a vertex with out-degree zero is called a **sink vertex**.*
- *An **isolated vertex** is a vertex with degree zero, which is not an endpoint of an edge.*
- *An edge that is associated with the similar end points can be called as **Loop**.*
- *If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or **adjacent nodes**.*
- *A **degree of a node** is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.*

Directed Graph (Digraph) Implementation

REVATURE

Demo Code

BFS & DFS Demo

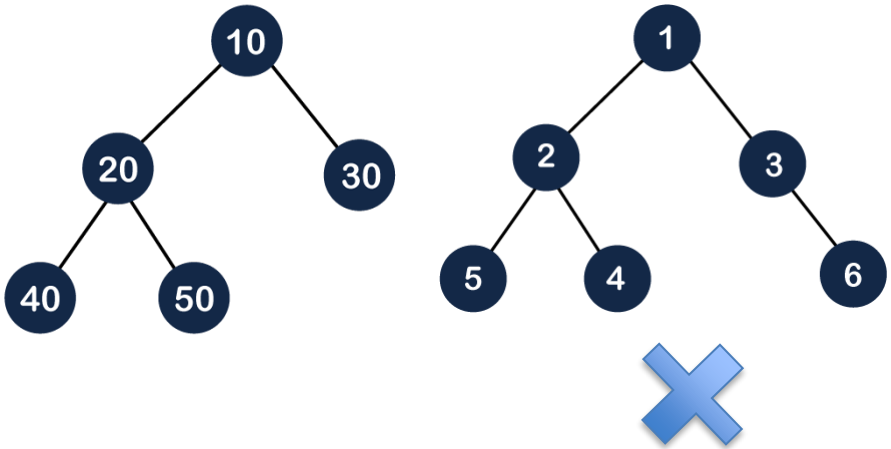
REVATURE

Demo Code
https://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal.htm

Binary Heap Data Structure

REVATURE

A heap is a complete binary tree, and the binary tree is a tree in which the node can have utmost two children.
A Binary Heap is a Binary Tree with following properties.
1) It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.
The below right figure shows that all the internal nodes are completely filled except the leaf node, but the leaf nodes are added at the right part; therefore, the above tree is not a complete binary tree.



What is a heap?

- Always keep the thing we are most interested in close to the top (and fast to access).
- Like a binary search tree, but less structured.
- No relationship between keys at the same level (unlike BST).

How is Binary Heap Represented?

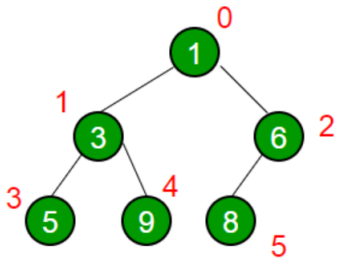
REVATURE

A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as an array.

- *The root element will be at Arr[0].*
- *Below table shows indexes of other nodes for the ith node, i.e., Arr[i]:*

Arr[(i-1)/2]	Returns the parent node
Arr[(2*i)+1]	Returns the left child node
Arr[(2*i)+2]	Returns the right child node

- *The traversal method use to achieve Array representation is Level Order*



1	3	6	5	9	8
0	1	2	3	4	5

Binary Heap Types

REVATURE

There are two types of the heap:

- *Min Heap*
- *Max heap*

Min Heap

REVATURE

Min Heap:*In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree.*

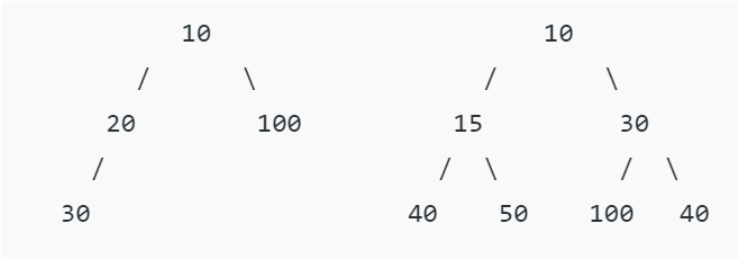
Or

Min heap has the following properties:
Root node value is always smaller in comparison to the other nodes of the heap.
Each internal node has a key value that is always smaller or equal to its children.
In the below figure, 11 is the root node, and the value of the root node is less than the value of all the other nodes (left child or a right child).

We can perform the following three operations in Min heap:

- ***insertNode()***
- ***extractMin()*** *-It is one of the most important operations which we perform to remove the minimum value node, i.e., the root node of the heap. After removing the root node, we have to make sure that heap property should be maintained*
- ***getMin()***

The getMin() operation is used to get the root node of the heap.



Max Heap

REVATURE

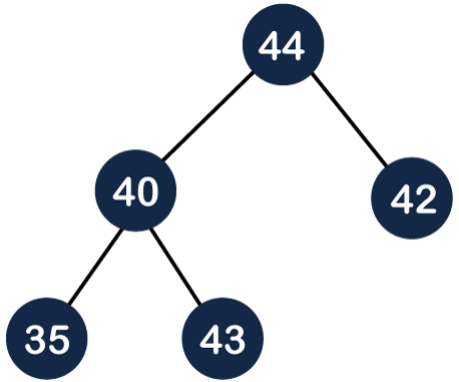
***Max Heap**: The value of the parent node is greater than or equal to its children. Or*

Max heap has the following properties:
Root node value is always greater in comparison to the other nodes of the heap.
Each internal node has a key value that is always greater or equal to its children.

We can perform the following three operations in Min heap:

- ***insertNode()***
- ***extractMax()** -It is one of the most important operations which we perform to remove the maximum value node, i.e., the root node of the heap. After removing the root node, we have to make sure that heap property should be maintained.*
- ***getMax()***

The getMax() operation is used to get the root node of the heap.



Applications of Heap

REVATURE

Following are some uses other than Heapsort.

- *Priority Queues: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time.*
- *Heap Implemented priority queues are used in Graph algorithms like Prim's Algorithm and Dijkstra's algorithm.*
- *Order statistics: The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array.*

Min-Heapify vs Max-Heapify

REVATURE

A very common operation on a heap is heapify, which rearranges a heap in order to maintain its property.
It property will be to maintain min heap or max heap.

Demo code

Greedy Algorithm

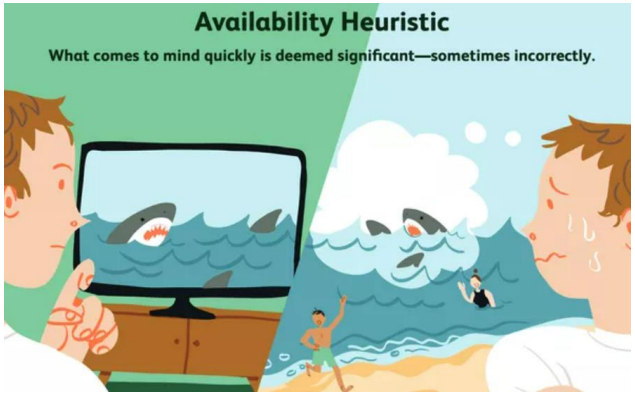
REVATURE

- *Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.*
- *Greedy algorithms are used for optimization problems.*
- *An optimization problem can be solved using Greedy if the problem has the following property:*
    - *At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem.*
- ***If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem as the Greedy algorithms are in general more efficient than other techniques like Dynamic Programming**.*

Greedy Algorithm

REVATURE

- *Among all the algorithmic approaches, the simplest and straightforward approach is the Greedy method.*
- *In this approach, the decision is taken on the basis of current available information without worrying about the effect of the current decision in future.*
- *Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit.*
- *This approach never reconsiders the choices taken previously.*
- *This approach is mainly used to solve optimization problems. Greedy method is easy to implement and quite efficient in most of the cases.*
- *Hence, we can say that Greedy algorithm is an algorithmic paradigm based on ***heuristic** that follows local optimal choice at each step with the hope of finding global optimal solution.*
- *In many problems, it does not produce an optimal solution though it gives an approximate (near optimal) solution in a reasonable time.*



***Heuristic**:*

- *To learn through doing and discovering things themselves rather than telling them about things*
- *A heuristic, or heuristic technique, is any approach to problem solving or self-discovery that employs a practical method that is not guaranteed to be optimal, perfect, or rational, but is nevertheless sufficient for reaching an immediate, short-term goal or approximation.*

Limitations of Greedy Algorithm

REVATURE

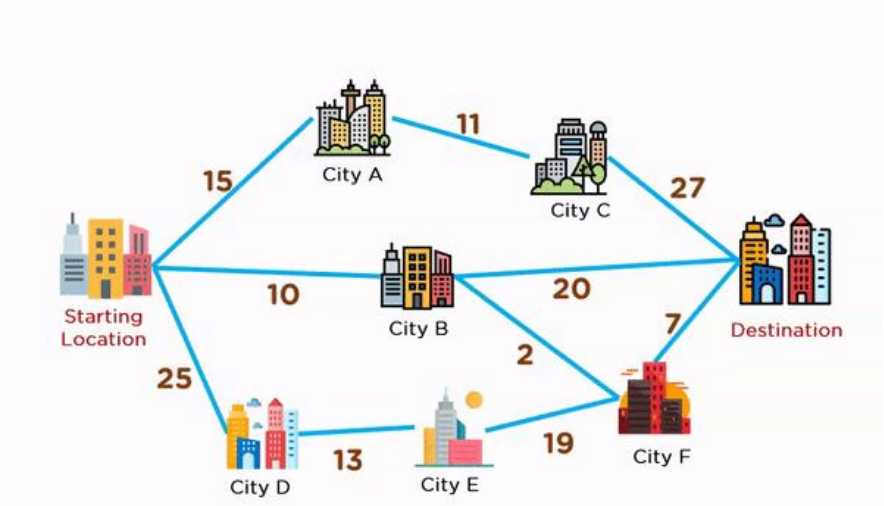
Factors listed below are the limitations of a greedy algorithm:

- *The greedy algorithm makes judgments based on the information at each iteration without considering the broader problem; hence it does not produce the best answer for every problem.*
- *The problematic part for a greedy algorithm is analyzing its accuracy. Even with the proper solution, it is difficult to demonstrate why it is accurate.*
- *Optimization problems (Dijkstra's Algorithm) with negative graph edges cannot be solved using a greedy algorithm.*

Applications of Greedy Algorithm

Following are few applications of the greedy algorithm:

- *Used for Constructing Minimum Spanning Trees: Prim's and Kruskal's Algorithms used to construct minimum spanning trees are greedy algorithms.*
- *Used to Implement Huffman Encoding: A greedy algorithm is utilized to build a Huffman tree that compresses a given image, spreadsheet, or video into a lossless compressed file.*
- *Used to Solve Optimization Problems: Graph - Map Coloring, Graph - Vertex Cover, Knapsack Problem, Job Scheduling Problem, and activity selection problem are classic optimization problems solved using a greedy algorithmic paradigm.*



Where to use Dynamic Programming?

REVATURE

- *Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.*
- *The two main properties of a problem that suggests that the given problem can be solved using Dynamic programming.*
    1. *Overlapping Subproblems :*
        - *Similar to Divide-and-Conquer approach, Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don't have to be re-computed. Hence, this technique is needed where overlapping sub-problem exists.*
        - *For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.*
    2. *Optimal Substructure:*
        - *A given problem has Optimal Substructure Property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.*
        - *For example, the Shortest Path problem has the following optimal substructure property –*
        - *If a node x lies in the shortest path from a source node **S** to destination node **D**, then the shortest path from **S** to **D** is the combination of the shortest path from **S** to **X**, and the shortest path from **X** to **D**.*

Steps of Dynamic Programming Approach

REVATURE

Dynamic Programming algorithm is designed using the following four steps –

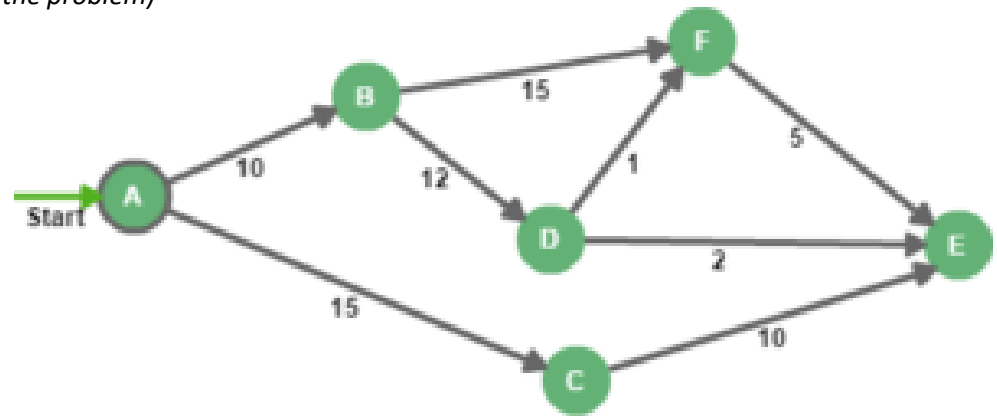
1. *Characterize the structure of an optimal solution.*
2. *Recursively define the value of an optimal solution.*
3. *Compute the value of an optimal solution, typically in a bottom-up fashion.*
4. *Construct an optimal solution from the computed information.*

Example: Graph Shortest Path problem

REVATURE

Dynamic Programming vs Greedy Algorithm:

- *DP algorithms uses the fact that (for some problems) - an optimal solution to a problem of size n is composed of an optimal solution to a problem of size n'<n, and uses this to build the solution bottom-up, from the smallest problem to the required size.*
- *Greedy algorithms are looking at a local point and doing some choice with the data at this point. For some problems (shortest path without negative weights for example) - this local choice will lead to an optimal solution.*
- *A good example of the differences between the two approaches is for the shortest path problem:*
- ***Dijsktra's Algorithm** is a Greedy Approach (At each step, chose the node that the path to it is currently minimized - the choice is done greedily based on the local state of the algorithm).*
- ***Floyd-Warshall and Bellman-Ford Algorithm** is a DP solution ("relax" The standard All Pair Shortest Path algorithms where all edges effectively reducing the problem)*



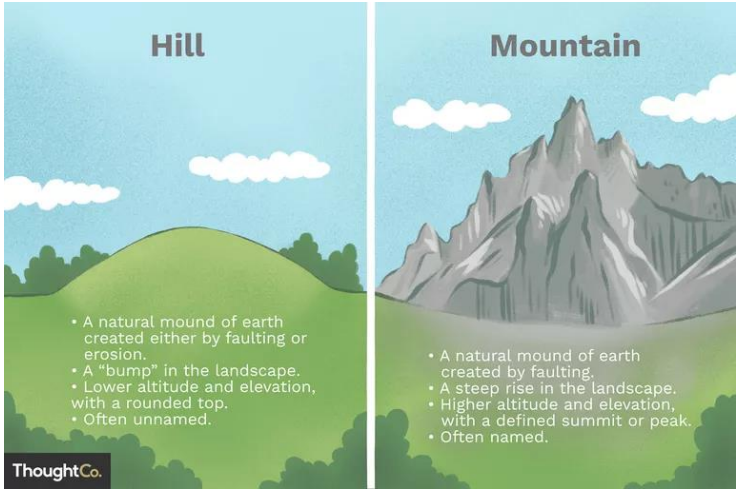
https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/?ref=lbp

Dynamic Programming Vs Greedy Algorithm Example

REVATURE

The difference between DP and greedy is DP will look for the global optimal at each subproblem, but greedy will only look for the local optimal.

- *Suppose you are climbing a mountain, and you want to climb as high as possible.*
- *The road on the mountain has several branches, and at each intersection you need to decide which branch to take, which is the subproblem of this climbing problem (the goal is the same, only the start point is different)*
- *For the greedy algorithm, you will always choose whichever one seems more steeply up.*
- *This is a local optimal decision and is not guaranteed to lead to the best result*
- *For the DP, at each intersection, you should already know the highest altitude each branch will lead you to (suppose your evaluation order is reversed, from end points to the starting point) and choose the one with the largest altitude.*
- *This decision is built upon the global optimum of the future subproblems and there for will be globally optimum for this subproblem.*



Dynamic Programming Vs Greedy Algorithm

REVATURE

- ***Greedy Algorithm:***
    1. *Greedy method focuses on expanding partially constructed solutions.*
    2. *It provides many results such as feasible solution.*
    3. *More efficient*
- ***Dynamic programming:***
    1. *Focuses on principle of optimality.*
    2. *It provides specific answers.*
    3. *Less efficient*

- *The difference between dynamic programming and greedy algorithms is that with dynamic programming, there are overlapping subproblems, and those subproblems are solved using memoization. "Memoization" is the technique whereby solutions to subproblems are used to solve other subproblems more quickly.*
- *The difference is that dynamic programming requires you to remember the answer for the smaller states, while a greedy algorithm is local in the sense that all the information needed is in the current state. Of course, there is some intersection.*
- *The key distinction is that greedy algorithms compose solutions "statically" in the sense that each local choice in the solution can be finalized without needing to know anything about the other local choices made. Dynamic algorithms, however, create sets of possible solutions to sub-problems and only generate a single solution to the global problem when all sub-problems have been considered.*

Dynamic Programming Vs Greedy Algorithm

REVATURE

- *The choice made by a greedy algorithm may depend on choices made so far but not on future choices or all the solutions to the subproblem.*
- *It iteratively makes one greedy choice after another, reducing each given problem into a smaller one.*
- *In other words, a greedy algorithm never reconsiders its choices.*
- *This is the main difference from dynamic programming, which is exhaustive and is guaranteed to find the solution.*
- *After every stage, dynamic programming makes decisions based on all the decisions made in the previous stage, and may reconsider the previous stage's algorithmic path to solution.*

Additional References:
https://www.simplilearn.com/tutorials/data-structure-tutorial/greedy-algorithm
https://www.simplilearn.com/tutorials/data-structure-tutorial/what-is-dynamic-programming

Guess the Technique

Question

"How many possible ways are there to make change for a dollar?"

Answer

There are exactly 293 ways to express a dollar in currently-circulating coins, and by that we mean these coins:

https://www.maa.org/frank-morgans-math-chat-293-ways-to-make-change-for-a-dollar

Guess the Technique -Solution

REVATURE

- *Consider the problem "Making change with dollars, nickels, and pennies." This is a **Greedy Problem**. It exhibits optimal substructure because you can solve for the number of dollars. Then, solve for the number of nickels. Then the number of pennies. You can then combine the solutions to these subproblems efficiently. It does not really exhibit overlapping subproblems since solving each subproblem doesn't help much with the others (maybe a little bit).*

- *Consider the problem "Fibonnaci numbers." It exhibits optimal substructure because you can solve F(10) from F(9) and F(8) efficiently (by addition). These subproblems overlap because they both share F(7). If you memoize the result of F(7) when you're solving F(8), you can solve F(9) more quickly.*

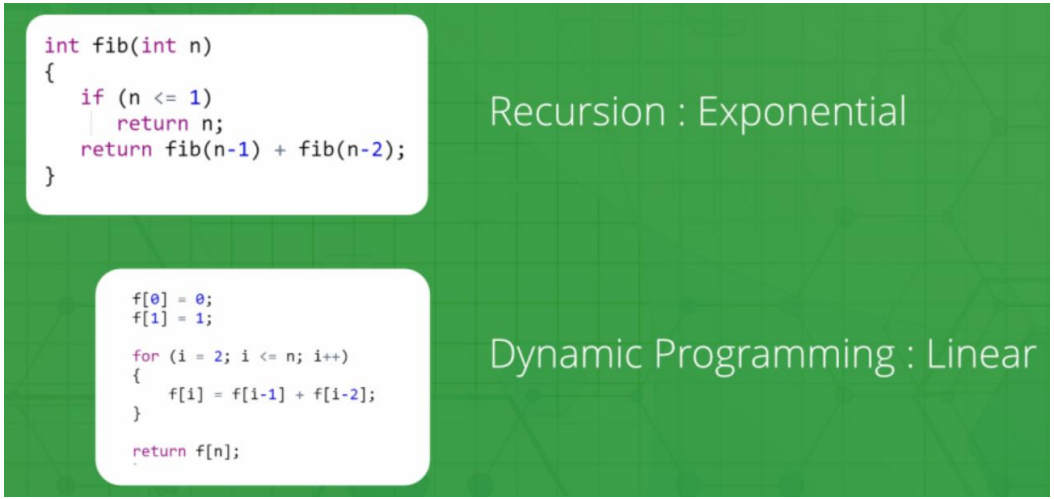
Demo Code

Greedy Algorithm to find Minimum number of Coins:
https://www.geeksforgeeks.org/greedy-algorithm-to-find-minimum-number-of-coins/
Find minimum number of coins that make a given value:
https://www.geeksforgeeks.org/find-minimum-number-of-coins-that-make-a-change/

Example Dynamic Programming

REVATURE

- *Dynamic Programming is mainly an optimization over plain recursion.*
- *Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming*
- *The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later.*
- *This simple optimization reduces time complexities from exponential to polynomial.*
- *For example, if we write simple recursive solution for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.*



Solution

REVATURE

- *Remember the results.*
- *Do not solve the problem again, just recall it from memory*

How?

- *Two methods of storing results in memory-*
    1. *Top-Down (Memoization) – We store the results in the memory whenever we solve a problem for the first time and next time onwards we simply do a lookup. Avoids multiple lookups, thus, saves function call overhead time.*
    2. *Bottom-Up (Tabulation) – We precompute the solutions in a linear fashion and store it in a table.*

Top-Down (Memoization)

REVATURE

- *The memoized program for a problem is similar to the recursive version with a small modification that looks into a lookup table before computing solutions.*
- *We initialize a lookup array with all initial values as NIL.*
- *Whenever we need the solution to a subproblem, we first look into the lookup table.*
- *If the precomputed value is there then we return that value, otherwise, we calculate the value and put the result in the lookup table so that it can be reused later.*

Bottom-Up (Tabulation)

REVATURE

- *The tabulated program for a given problem builds a table in bottom-up fashion and returns the last entry from the table.*
- *For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3), and so on. So literally, we are building the solutions of subproblems bottom-up.*

Key Differences: Top-Down vs Bottom-Up Approach

REVATURE

Top-Down Approach	Bottom-Up Approach
Uses memorization technique	*Uses tabulation technique*
Recursive nature	*Iterative nature*
Structured programming languages such as COBOL, Fortran, C, and others mostly use this technique	*Object-oriented programming languages such as C++, C#, and Python mostly use this technique*
This approach uses decomposition to formulate a solution	*This approach uses composition to develop a solution*
A lookup table is maintained and checked before computation of any subproblem	*The solution is built from a bottom-most case using iteration*

Solution Performance

Time taken to calculate the 40th Fibonacci number (1023344155)

1. *Recursive: 14 Seconds*
2. *Memoization: 0.17 Seconds*
3. *Tabulation: 0.30 Seconds*

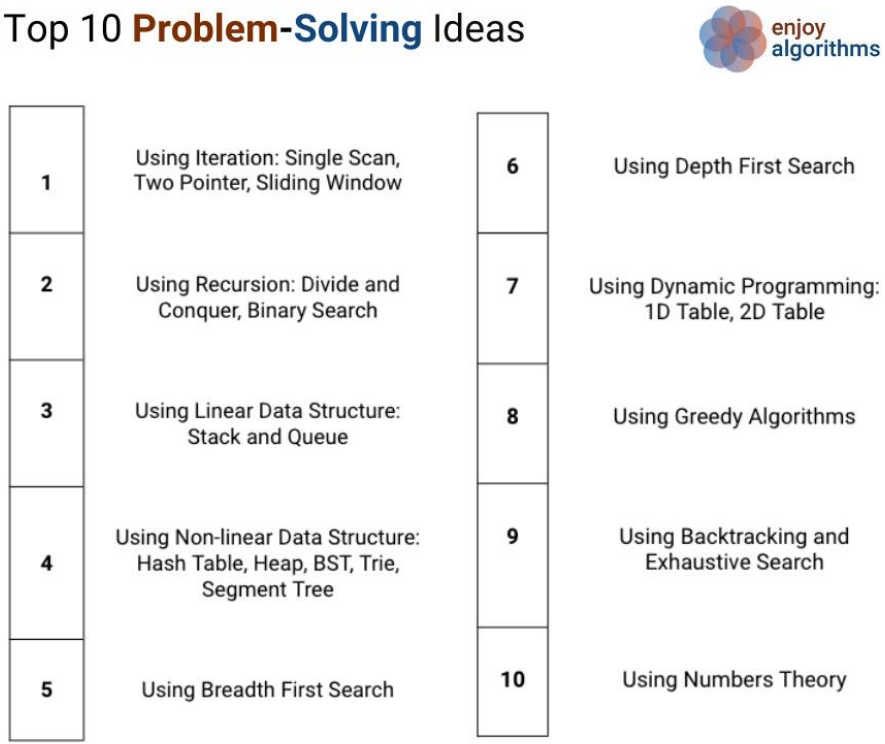
Demo Code

Dynamic Programming Vs Divide and Conquer

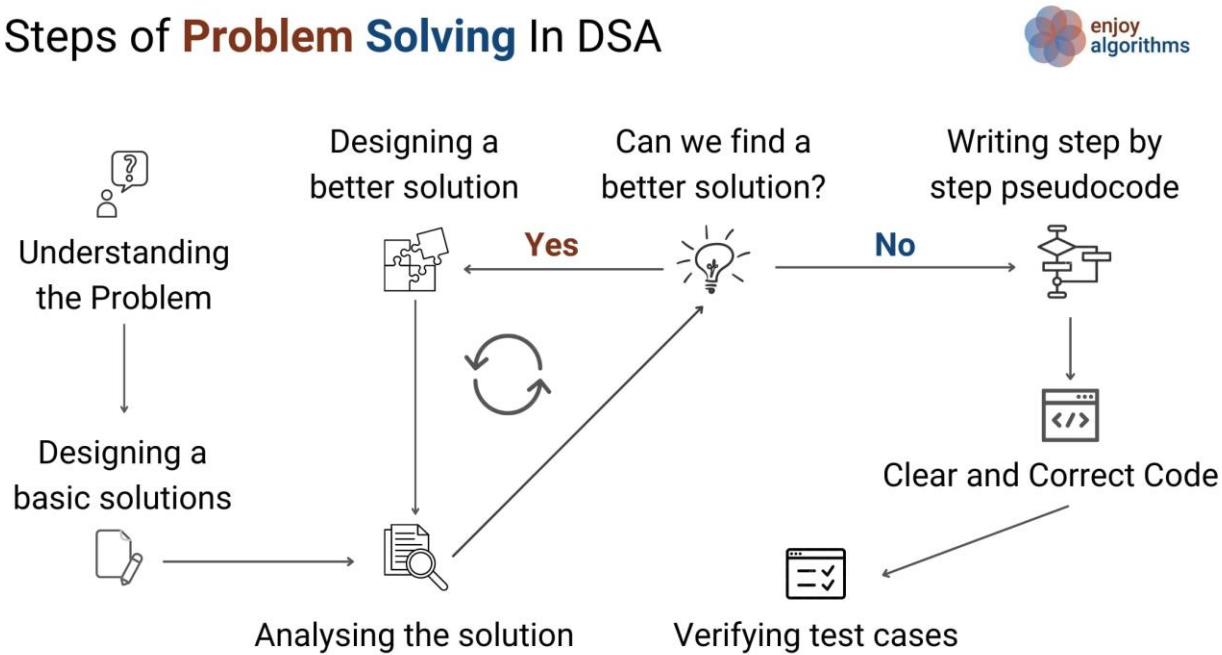
REVATURE

- ***Similarities***- *Both Paradigms work by combining solutions to sub-problems.*
- ***Differences***- *Dynamic programming is mainly used when the overlapping subproblems property is satisfied.*
- ***Example-***
    1. *Binary Search - Divide and Conquer*
    2. *Fibonacci Series - Dynamic Programming*

Problem Solving Ideas



Problem Solving Steps



Pattern Matching Algorithms

REVATURE

Pattern searching is an important problem in computer science.
When we do search for a string in a notepad/word file, browser, or database, pattern searching algorithms are used to show the search results.
Pattern Searching algorithms are used to find a pattern or substring from another bigger string.
The main goal to design these type of algorithms to reduce the time complexity.
The traditional approach may take lots of time to complete the pattern searching task for a longer text.

Boyer Moore Algorithm for Pattern Searching

REVATURE

- Construct 'Bad Match Table'.
- Compare pattern to the text, starting from the **rightmost** character in pattern.
- When the mismatch occurs you shift the pattern to the right corresponding to the value in 'Bad Match Table'.

Constructing Bad Match Table

- Construct **Bad Match Table**.

Value = length – index – 1

(every remaining letter = length)

Constructing Bad Match Table - Boyer Moore Algorithm Example 1

REVATURE

Example 1:

- Pattern – 'teammast'
- Text – 'welcometoteammast'

T E A M M A S T Length = 8

Index : 0 1 2 3 4 5 6 7

Letter	T	E	A	M	S	*
Value						

T = 8 - 0 - 1

(Value = length – index – 1)

T E A M M A S T Length = 8

Index : 0 1 2 3 4 5 6 7

Letter	T	E	A	M	S	*
Value	7	6	2	3	1	

S = 8 - 5 - 1

T E A M M A S T Length = 8

Index : 0 1 2 3 4 5 6 7

Letter	T	E	A	M	S	*
Value	7					

Constructing Bad Match Table - Boyer Moore Algorithm Example 1

REVATURE

Example 1:

- Pattern – 'teammast'
- Text – 'welcometoteammast'

T E A M M A S T Length = 8

Index : 0 1 2 3 4 5 6 7

Letter	T	E	A	M	S	*
Value						

T E A M M A S T Length = 8

Index : 0 1 2 3 4 5 6 7

Letter	T	E	A	M	S	*
Value	7					

T = 8 - 0 - 1

(Value = length – index – 1)

T E A M M A S T Length = 8

Index : 0 1 2 3 4 5 6 7

Letter	T	E	A	M	S	*
Value	7	6	2	3	1	

S = 8 - 5 - 1

T E A M M A S T Length = 8

Index : 0 1 2 3 4 5 6 7

Letter	T	E	A	M	S	*
Value	8	6	2	3	1	

T = 8 Last letter = length, if not already defined.

Since Last T char is duplicate update 1st value

T E A M M A S T Length = 8

Index : 0 1 2 3 4 5 6 7

Letter	T	E	A	M	S	*
Value	8	6	2	3	1	8

Also as mentioned earlier value of any other letter represented by '*' is taken equal to length of string i.e here = 8.

For all non-matching char we use * as last value

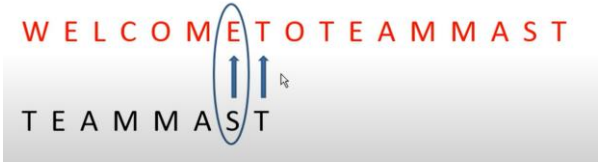
Boyer Moore Algorithm Example 1

REVATURE

Letter	T	E	A	M	S	*
Value	8	6	2	3	1	8

Start from right and compare till non-matching found

Letter	T	E	A	M	S	*
Value	8	6	2	3	1	8

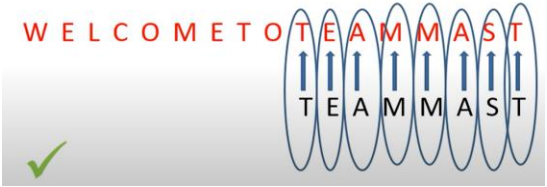


Mismatch, shift right

Letter	T	E	A	M	S	*
Value	8	6	2	3	1	8

Move 8 character next as we started from T

Letter	T	E	A	M	S	*
Value	8	6	2	3	1	8



Finally found

Boyer Moore Algorithm Example 2

REVATURE

Example 2:

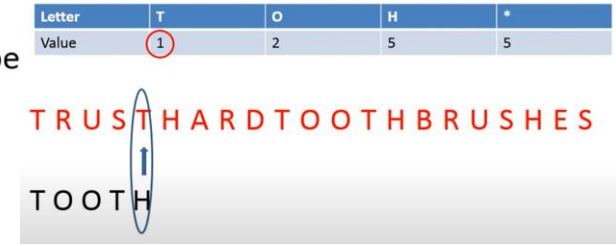
- Pattern – 'tooth'
- Text – 'trusthardtoothbrushes'

T O O T H Length = 5

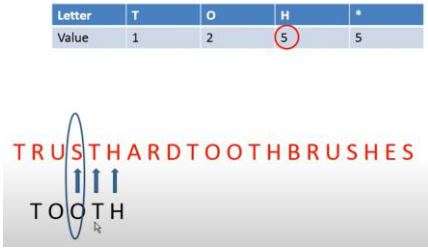
Index : 0 1 2 3 4

Solving, it's Bad Match Table comes out to be

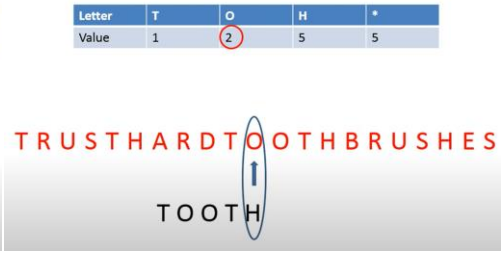
Letter	T	O	H	*
Value	1	2	5	5



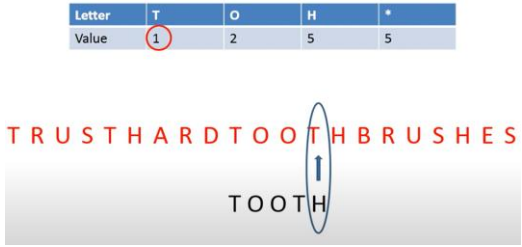
Mismatch, shift right by 1 char



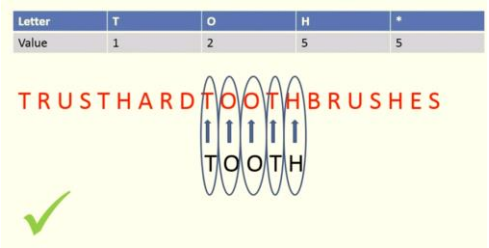
Mismatch, shift right by 5 char



Mismatch, shift right by 2 char

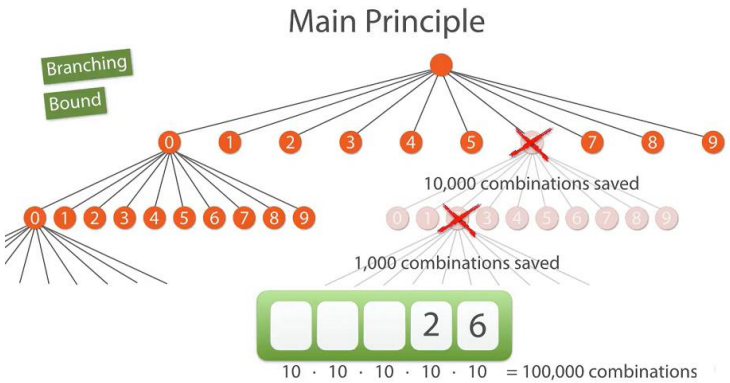


Mismatch, shift right by 1 char



Branch and Bound Technique

In computer science, there is a large number of optimization problems which has a finite but extensive number of feasible solutions.
Branch and bound algorithms are used to find the optimal solution for combinatory, discrete, and general mathematical optimization problems.
Branch and bound algorithm explores the entire search space of possible solutions and provides an optimal solution.
A branch and bound algorithm consist of stepwise enumeration of possible candidate solutions by exploring the entire search space.
With all the possible solutions, we first build a rooted decision tree.
Optimization Problem– Either minimization or maximization problems are solved using Branch and bound technique



Travelling Salesman Problem

REVATURE

In Java, Travelling Salesman Problem is a problem in which we need to find the shortest route that covers each city exactly once and returns to the starting point.

Steps-

1. *Create State Space Tree (Decision Tree)*
2. *Bound Function - which Node will be generate and which not using below types-*
    1. *Upper bound – for maximization problem*
    2. *Lower bound – for minimization problem*

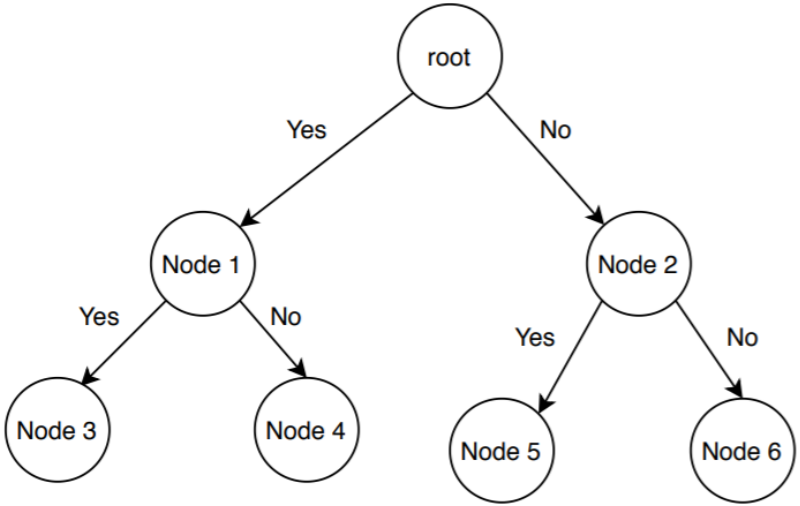
1. *Branch & Bound follows - Breadth First Search*

https://www.techiedelight.com/travelling-salesman-problem-using-branch-and-bound/
https://www.gatevidyalay.com/travelling-salesman-problem-using-branch-and-bound-approach/

Branch and Bound Technique

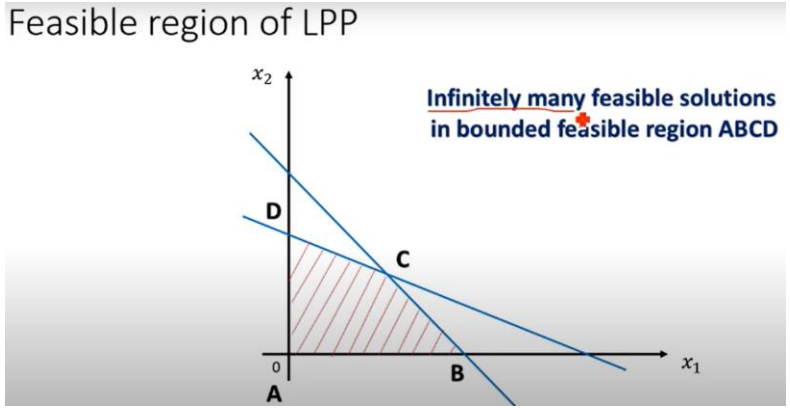
REVATURE

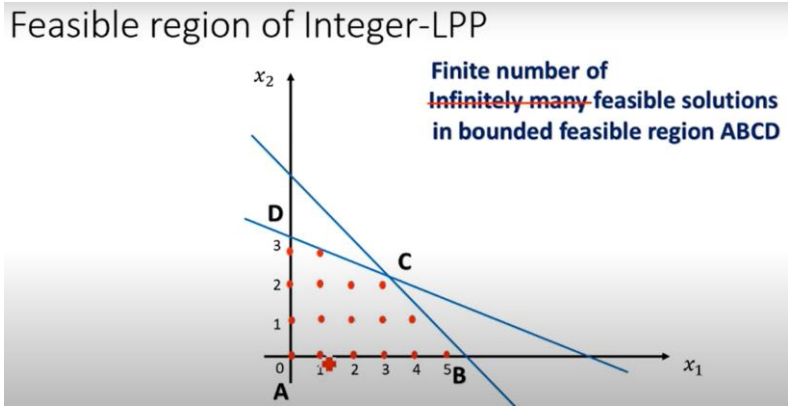
Here, each child node is a partial solution and part of the solution set. Before constructing the rooted decision tree, we set an upper and lower bound for a given problem based on the optimal solution. At each level, we need to make a decision about which node to include in the solution set. At each level, we explore the node with the best bound. In this way, we can find the best and optimal solution fast.



Branch and Bound Technique

Used to solve Integer Linear Programming Problems





In graph there can be infinite feasible solution for LPP but finite solutions for ILP