Bangladesh Army University of Science and Technology
Department of Computer Science & Engineering

Course Outline

Course Title: **Data Structures and Algorithms I** Course Code: **CSE 2103**
Credit: **3.00**

Course Conveners:

Md. Moazzem Hossain
Lecturer, CSE, BAUST
Contact: +8801719363034
Email: moazzem@baust.edu.bd

Course Description: Internal data representation; Abstract data types; Algorithm performance and elementary asymptotic analysis (Introduction to Big-O notation); Elementary data structures: array, linked list, stack, queue, tree and tree traversal, graphs and graph representation, heap, binary search tree; Sorting algorithms; Searching: linear search and binary search; Advanced data Structures: balanced binary search trees, skip list, advanced heaps, Hashing.

Text Book(s):

1. Data Structures – Seymour Lipschutz
2. Data Structures Using C and C++ – Y. Langsam, M. J. Augenstein, A. M. Tenenbaum

Reference Book(s):

1. Data structures and algorithm – Hopcroft, Ullman
2. Introduction to Algorithms – Thomas H. Cormen
3. Data structures and Algorithm- Aho, Hopcroft and Ullmann

Course Objectives:
A successful student will be able to:

1. Describe and implement a variety of basic data structures (array, linked list, stack, queue, binary tree, binary search tree, heap, graphs).
2. Describe and implement a variety of sorting algorithms including Bubble Sort, Insertion Sort.
3. Analyze the space and time complexity of the data structures and sorting algorithms studied in the course.
4. Identify different solutions for a given problem; analyze advantages and disadvantages to different solutions.

Marks Distribution:

Theory Course		**Sessional Course**	
Class Participation / Observation	5%	Class Attendance	10%
Class Attendance	5%	Class Performance	10%
HW/ Assignment/ Quizzes/**Class tests**	20%	Report	10%
Final Examination (3 hours)	70%	Quiz	20%
Total	**100%**	Viva	20%
		Lab Test	30%
		Total	**100%**

Lesson Plan

Week	Day	Topics to be taught	Remarks
01	01	• Introduction and Course Overview ○ Basic Terminologies ○ Data Structures ○ Different Types of Data Structures ○ Data Structures Operations	
	02	• Complexity Theory and Algorithmic Notation ○ Time Complexity ○ Space Complexity	
	03	• Asymptotic Notations	
02	04	• Arrays ○ One-dimensional Array ○ Multidimensional Array	
	05	○ Sparse Arrays	
	06	○ Operations ○ String operations	CT 1
03	07	• Searching + Complexity analysis ○ Linear ○ Binary	Adjunct Professor (4 class)
	08	• Sorting + complexity ○ Selection Sort ○ Bubble Sort ○ Insertion Sort	
	09	○ Merge Sort ○ Radix Sort ○ Bucket Sort ○ Quick Sort	
04	10	• Pointers	CT 2
	11	• Linked List	
	12	○ Single Linked List ○ Doubly link list	
05	13	○ Linked List operations ○ Traversal	
	14	○ Insert ▪ Beginning ▪ Intermediate ▪ End ▪ Chronological	
	15	○ Update ○ Delete	
06	16	• Hashing ○ Hash Function	Adjunct Professor (3 class)
	17	○ Collision Resolution ○ Open Addressing	
	18	○ Channing	

07	19	• Stack ○ Stack implementation ○ Stack operations(Push, Pop)	CT 3
	20	○ Stacks algorithms	
		○ String reversing	
	21	○ Infix to postfix ○ Postfix evaluation	
08	22	• Queue ○ Queue Implementation ○ Enqueue ○ Dequeue	
	23	○ Circular Queue	
	24	○ Deques	
09	25	• Trees ○ Terms	
	26	○ Binary tree properties	
	27	○ Tree traversals(Preorder, Post order, In order)	
10	28	• Graphs ○ Search: DFS, BFS	CT 4
	29	○ Shortest Path: Warshall	
	30	○ Floyed Warshall	
11	31	• Trees (Continued) ○ Binary Search Tree	
	32	○ Balanced BST	
	33	○ Operations on BST	
12	34	• Review on Searching, Sorting	
	35	• Review on Hashing	
	36	• Review on Trees	CT 5
13	37	• Heap ○ Max- Heap	Adjunct Professor (4 class)
	38	○ Min-Heap	
	39	○ Heap Application: ○ Priority Queue ○ Heap-Sort	
14	40	• Review on Complexity Theory and Array	
	41	• Review on Pointer, Linked List and Queue	
	42	• Review on Graph and Heap	

Special Instructions:

1. Students are encouraged to attend classes on time. Latecomers will not be allowed to disrupt the flow of the lecture.
2. After each class, students should review class notes seriously because usually the next topic

relies on the previous topic.

3. No makeup class test.

Lecture 1

Introduction and Course overview

Outlines:

- **Basic Terminologies**
- **Data Structures**
- **Different Types of Data Structures**
- **Data Structures Operations**

Basic terminologies:

Data are simply values or sets of values. A data item refers to a single unit of values. **Data**, in the context of computing, refers to distinct pieces of digital information. Data is usually formatted in a specific way and can exist in a variety of forms, such as numbers, text, etc. When used in the context of transmission media, data refers to information in binary digital format.

Group Item: Data items that are divided into sub items are called Group item. For example employee's names divided into three sub items- First name, Middle name and Last name.

Elementary Item: Data items that are not divided into sub items are called Elementary items. But social security numbers of employee are not divided into sub items.

Attribute/ Field: Attribute/ Filed is a single elementary unit of information representing an attribute of an entity. Example: Name, Age, Gender e.t.c.

Entity/Record: Entity/Record is something that has certain attributes or properties which may be assigned values. The values themselves may be either numeric or nonnumeric. For example, the following are possible attributes and their corresponding values for an entity, an employee of a given organization.

File/Table					
Attributes/Fields:	*Name*	*Age*	*Gender*	*Phone Number*	
Values:	Noman, Sarkar	24	Male	01818….	**Entity/Record**
	Sakil, Ahmed	23	Male	01723…..	
	Mithila, Hossain	22	Female	01615…..	

Data Structures:

Data Structure is the mathematical or logical model of a particular organization of data is called a data structure. **Data Structure** is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have some data which has, player's **name** "Abed" and **age** 26. Here "Abed" is of **String** data type and 26 is of **integer** data type.

We can organize this data as a record like **Player** record, which will have both player's name and age in it. Now we can collect and store player's records in a file or database as a data structure. **For example**: "Dinar" 30, "Gazi" 31, "Tamim" 33.

If you are aware of Object Oriented programming concepts, then a `class` also does the same thing, it collects different type of data under one single entity. The only difference being, data structures provides for techniques to access and manipulate data efficiently.

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily. It represents the knowledge of data to be organized in memory. It should be designed and implemented in such a way that it reduces the complexity and increases the efficiency.

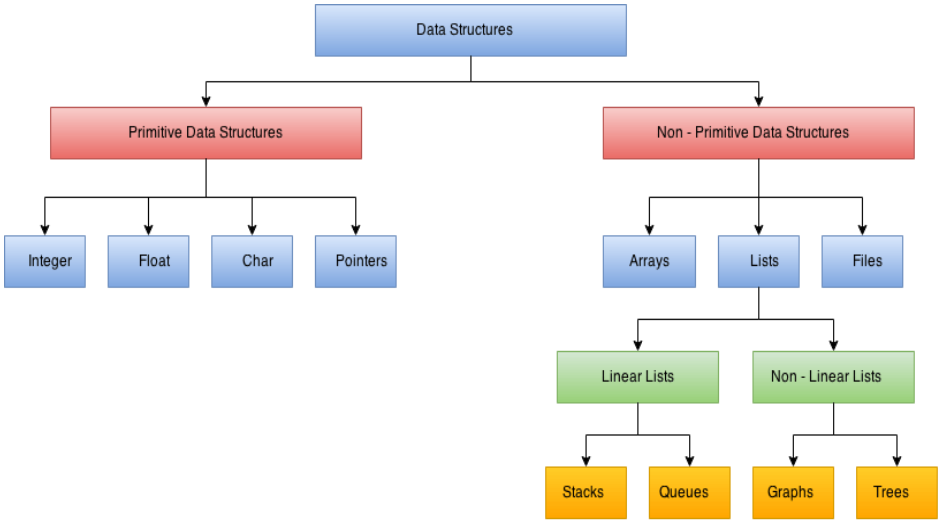
Different Types of Data Structures:

There are different types of Data Structures, like arrays, linked lists, trees etc. This is because some type of data structures are efficient in a particular application than others. Like, say, if we want to store a series of number which are of fixed size, we would use an array. And also, it is important for us to get faster access of data. Hence forth, some data structures have faster access like in case of AVL Trees were we make sure that the tree is balanced so that access time is less. We are more concerned about this 'time complexity' because, in real life situations, accessing big chunks of data would take a long time and hence we need better algorithms (Data Structures) for faster access.

Mainly Data Structures are two types:

i) Primitive Data Structures

ii) Non- Primitive Data Structures



i) Primitive Data Structures: The term "data type" and "primitive data type" are often used interchangeably. Primitive data types are predefined types of data, which are supported by the programming language. For example, integer, character, and string are all primitive data types. Programmers can use these data types when creating variables in their programs. For example, a programmer may create a variable called "last name" and define it as a string data type. The variable will then store data as a string of characters.

Types of Primitive Data Structure:

Data type	Size(bytes)	Range	Format String
char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
short	2	-32,768 to 32,767	%d
unsigned short	2	0 to 65535	%u
int	2	32,768 to 32,767	%d
unsigned int	2	0 to 65535	%u
long	4	-2147483648 to +2147483647	%ld
Unsinged long	4	0 to 4294967295	%lu
float	4	-3.4e-38 to +3.4e-38	%f
double	8	1.7 e-308 to 1.7 e+308	% lf
long double	10	3.4 e-4932 to 1.1 e+4932	%lf

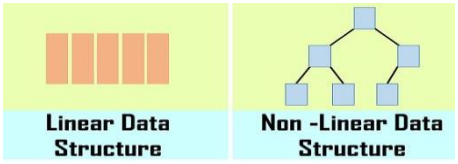
ii) Non- Primitive Data Structures: Non-primitive data types are not defined by the programming language, but are instead created by the programmer. They are sometimes called "reference variables," or "object references," since they reference a memory location, which stores the data. In the Java programming language, non-primitive data types are simply called "objects" because they are created, rather than predefined. While an object may contain any type of data, the information referenced by the object may still be stored as a primitive data type.

Array: an array data structure, or simply an array, is a data structure consisting of a collection of elements

(values or variables), each identified by at least one array index or key. An array is stored such that the position of each element can be computed from its index tuple by a mathematical formula. The simplest type of data structure is a linear array, also called one-dimensional array. For example, an array of 10 32-bit integer variables, with indices 0 through 9, may be stored as 10 words at memory addresses 2000, 2004, 2008, ... 2036, so that the element with index i has the address 2000 + 4 × i.

List: a list or sequence is an abstract data type that represents a countable number of ordered values, where the same value may occur more than once. An instance of a list is a computer representation of the mathematical concept of a finite sequence; the (potentially) infinite analog of a list is a stream. Lists are a basic example of containers, as they contain other values. If the same value occurs multiple times, each occurrence is considered a distinct item.

Linear List: a linear list stores a collection of objects of a certain type, usually denoted as the elements of the list. The elements are ordered within the linear list in a linear sequence. Linear lists are usually simply denoted as lists.



Unlike an array, a list is a data structure allowing insertion and deletion of elements at an arbitrary position of the sequence. If the position in question is given, for example by a reference, such a modification takes only a constant number of operations, that is, no effortful copying of entries is necessary and all insertion and deletion operations take an equally short time. Conversely, however, one cannot access a single element via an (integral) index in constant time, as in the case of an array, without having searched for it before and having received a reference to it. Furthermore, lists are not limited to a certain maximum number of elements from the beginning on (like an array). So they are a dynamic data structure

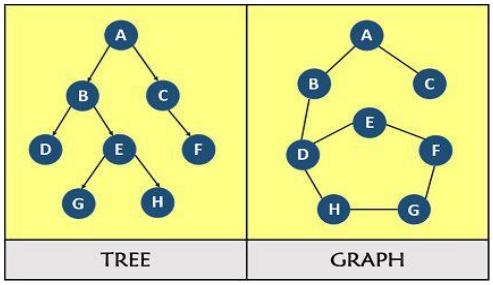
Stack: A stack is a basic data structure that can be logically thought of as a linear structure represented by a real physical stack or pile, a structure where insertion and deletion of items takes place at one end called top of the stack. The basic concept can be illustrated by thinking of your data set as a stack of plates or books where you can only take the top item off the stack in order to remove things from it. This structure is used all throughout programming.

Queue: Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first. A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Non Linear List: It arranges the data in a sorted order and there exists a relationship between the data elements. Traversing of data elements in one go is not possible and complex.

Graph: Graphs evolved from the field of mathematics. They are primarily used to describe a model that shows the route from one location to another location. A graph consists of a set of nodes and a set of edges. An edge is a pair of nodes that are connected. A path is the term used to describe traveling between nodes that share an edge.

Tree: A tree data structure, like a graph, is a collection of nodes. There is a root node. The node can then have children nodes. The children nodes can have their own children nodes called grandchildren nodes. This repeats until all data is represented in the tree data structure. A tree is a graph that has no cycles (a cycle being a path in the graph that starts and ends at the same vertex). A child node can only have one parent. For this reason trees are not a recursive data structure.



File: A collection of data or information that has a name, called the filename. Almost all information stored in a computer must be in a file. There are many different types of files: data files, text files, program files, directory files, and so on. Different types of files store different types of information. For example, program files store programs, whereas text files store text.

Data Structures Operations: Data are processed by means of certain operations which appearing in the data structure. Data has situation on depends largely on the frequency with which specific operations are performed. This section introduces the reader to some of the most frequently used of these operations.

(1) *Traversing:* Accessing each record exactly once so that certain items in the record may be processed.
(2) *Searching:* Finding the location of a particular record with a given key value, or finding the location of all records which satisfy one or more conditions.
(3) *Inserting:* Adding a new record to the structure.
(4) *Deleting:* Removing the record from the structure.
(5) *Sorting:* Managing the data or record in some logical order (Ascending or descending order).
(6) *Merging:* Combining the record in two different sorted files into a single sorted file.

(7) *Updating:* Update any element on the data structure (i.e. replace)

Lecture 2

Complexity Theory and Algorithmic Notation

Outlines:

- **Asymptotic Analysis**
- **Worst-Case and Average-Case Analysis**
- **Order of Growth and Big-O Notation**
- **Comparing Orders of Growth**
- **Shortcomings of asymptotic analysis**
- **Complexity of algorithms from class**

Algorithmic complexity is a very important topic in computer science. Knowing the complexity of algorithms allows you to answer questions such as

- How long will a program run on an input?
- How much space will it take?
- Is the problem solvable?

These are important bases of comparison between different algorithms. An understanding of algorithmic complexity provides programmers with insight into the efficiency of their code. Complexity is also important to several theoretical areas in computer science, including algorithms, data structures, and complexity theory.

Asymptotic Analysis: When analyzing the running time or space usage of programs, we usually try to estimate the time or space as function of the input size. For example, when analyzing the worst case running time of a function that sorts a list of numbers, we will be concerned with how long it takes as a function of the length of the input list. For example, we say the standard insertion sort takes time $T(n)$ where $T(n)= c*n^2+k$ for some constants *c and k*. In contrast, merge sort takes time $T'(n) = c'*n*log_2(n) + k'$.

The **asymptotic** behavior of a function *f(n)* (such as $f(n)=c*n$ or $f(n)=c*n^2$, etc.) refers to the growth of *f(n)* as *n* gets large. We typically ignore small values of *n*, since we are usually interested in estimating how slow the program will be on large inputs. A good rule of thumb is: the slower the asymptotic growth rate, the better the algorithm (although this is often not the whole story).

By this measure, a linear algorithm (*i.e.*, $f(n)=d*n+k$) is always asymptotically better than a quadratic one (*e.g.*, $f(n)=c*n^2+q$). That is because for any given (positive) *c, k, d,* and *q* there is always some *n* at which the magnitude of $c*n^2+q$ overtakes $d*n+k$. For moderate values of *n*, the quadratic algorithm could very well take less time than the linear one, for example if *c* is significantly smaller than *d* and/or *k* is significantly smaller than *q*. However, the linear algorithm will always be better for sufficiently large inputs. Remember to **THINK BIG** when working with asymptotic rates of growth.

Worst-Case and Average-Case Analysis: When we say that an algorithm runs in time *T(n)*, we mean that *T(n)* is an upper bound on the running time that holds for all inputs of size *n*. This is called *worst-case analysis*. The algorithm may very well take less time on some inputs of size *n*, but it doesn't matter. If an algorithm takes $T(n)=c*n^2+k$ steps on only a single input of each size *n* and only *n* steps on the rest, we still say that it is a quadratic algorithm.

A popular alternative to worst-case analysis is *average-case analysis*. Here we do not bound the worst case running time, but try to calculate the expected time spent on a randomly chosen input. This kind of analysis is generally harder, since it involves probabilistic arguments and often requires assumptions about the distribution of inputs that may be difficult to justify. On the other hand, it can be more useful because sometimes the worst-case behavior of an algorithm is misleadingly bad. A good example of this is the popular quicksort algorithm, whose worst-case running time on an input sequence of length *n* is proportional to n^2 but whose expected running time is proportional to *n* log *n*.

Order of Growth and Big-O Notation: In estimating the running time of `insert_sort` (or any other program) we don't know what the constants *c* or *k* are. We know that it is a constant of moderate size, but other than that it is not important; we have enough evidence from the asymptotic analysis to know that a `merge_sort` (see below) is faster than the quadratic `insert_sort`, even though the constants may differ somewhat. (This does not always hold; the constants can sometimes make a difference, but in general it is a very good rule of thumb.)

We may not even be able to measure the constant *c* directly. For example, we may know that a given expression of the language, such as `if`, takes a constant number of machine instructions, but we may not know exactly how many. Moreover, the same sequence of instructions executed on a Pentium IV will take less time than on a Pentium II (although the difference will be roughly a constant factor). So these estimates are usually only accurate up to a constant factor anyway. For these reasons, we usually ignore constant factors in comparing asymptotic running times.

Computer scientists have developed a convenient notation for hiding the constant factor. We write *O(n)* (read: "order *n*") instead of "*cn* for some constant *c*." Thus an algorithm is said to be *O(n)* or *linear time* if there is a fixed constant *c* such that for all sufficiently large *n*, the algorithm takes time at most *cn* on inputs of size *n*. An algorithm is said to be $O(n^2)$ or *quadratic time* if there is a fixed constant *c* such that for all sufficiently large *n*, the algorithm takes time at most cn^2 on inputs of size *n*. *O(1)* means *constant time*.

Polynomial time means $n^{O(1)}$, or n^c for some constant *c*. Thus any constant, linear, quadratic, or cubic ($O(n^3)$) time algorithm is a polynomial-time algorithm.

This is called *big-O notation*. It concisely captures the important differences in the asymptotic growth rates of functions.

One important advantage of big-O notation is that it makes algorithms much easier to analyze, since we can conveniently ignore low-order terms. For example, an algorithm that runs in time

$10n^3 + 24n^2 + 3n \log n + 144$

is still a cubic algorithm, since

$10n^3 + 24n^2 + 3n \log n + 144$
$<= 10n^3 + 24n^3 + 3n^3 + 144n^3$
$<= (10 + 24 + 3 + 144)n^3$
$= O(n^3)$.

Of course, since we are ignoring constant factors, any two linear algorithms will be considered equally good by this measure. There may even be some situations in which the constant is so huge in a linear algorithm that even an exponential algorithm with a small constant may be preferable in practice. This is a valid criticism of asymptotic analysis and big-O notation. However, as a rule of thumb it has served us well. Just be aware that it is *only* a rule of thumb--the asymptotically optimal algorithm is not necessarily the best one.

Some common orders of growth seen often in complexity analysis are

O(1)	constant
O(log n)	logarithmic
O(n)	linear
O(n log n)	"n log n"

$O(n^2)$	quadratic
$O(n^3)$	Cubic
$n^{O(1)}$	Polynomial
$2^{O(n)}$	Exponential

Here *log* means log_2 or the logarithm base 2, although the logarithm base doesn't really matter since logarithms with different bases differ by a constant factor. Note also that $2^{O(n)}$ and $O(2^n)$ are not the same!

Comparing Orders of Growth:

O Let f and g be functions from positive integers to positive integers. We say f is O(g(n)) (read: "f is order g") if g is an upper bound on f: there exists a fixed constant c and a fixed n0 such that for all $n \geqslant n_0$,

$f(n) \leqslant cg(n)$. Equivalently, f is O(g(n)) if the function f(n)/g(n) is bounded above by some constant.

o

We say f is o(g(n)) (read: "f is little-o of g") if for all arbitrarily small real c > 0, for all but perhaps finitely many n,

$f(n) \leqslant cg(n)$.Equivalently, f is o(g) if the function f(n)/g(n) tends to 0 as n tends to infinity. That is, f is small compared to g. If f is o(g) then f is also o(g)

Ω

We say that f is Ω(g(n)) (read: "f is omega of g") if g is a lower bound on f for large n. Formally, f is Ω(g) if there is a fixed constant c and a fixed n0 such that for all n>n0, $cg(n) \leqslant f(n)$. For example, any polynomial whose highest exponent is nk is Ω(nk). If f(n) is Ω(g(n)) then g(n) is O(f(n)). If f(n) is o(g(n)) then f(n) is not Ω(g(n)).

Θ

We say that f is Θ(g(n)) (read: "f is theta of g") if g is an accurate characterization of f for large n: it can be scaled so it is both an upper and a lower bound of f. That is, f is both O(g(n)) and Ω(g(n)). Expanding out the definitions of Ω and O, f is Θ(g(n)) if there are fixed constants c1 and c2 and a fixed n0 such that for all n>n0, $c1g(n) \leqslant f(n) \leqslant c2\ g(n)$

For example, any polynomial whose highest exponent is nk is Θ(nk). If f is Θ(g), then it is O(g) but not o(g). Sometimes people use O(g(n)) a bit informally to mean the stronger property Θ(g(n)); however, the two are different.

Here are some examples:

n + log n is O(n) and Q(n), because for all n > 1, n < n + log n < 2n.

n1000 is o(2n), because n1000/2n tends to 0 as n tends to infinity.

For any fixed but arbitrarily small real number c, n log n is o(n1+c), since n log n / n1+c tends to 0. To see this, take the logarithm

log(n log n / n1+c)

= log(n log n) - log(n1+c)

= log n + log log n - (1+c)log n

= log log n - c log n

and observe that it tends to negative infinity. The meaning of an expression like O(n2) is really a set of functions: all the functions that are O(n2). When we say that f(n) is O(n2), we mean that f(n) is a member of this set. It is also common to write this as f(n) = O(g(n)) although it is not really an equality.

We now introduce some convenient rules for manipulating expressions involving order notation. These rules, which we state without proof, are useful for working with orders of growth. They are really statements about sets of functions. For example, we can read #2 as saying that the product of any two functions in O(f(n)) and O(g(n)) is in O(f(n)g(n)). $cnm = O(nk)$ for any constant c and any $m \leqslant k$.

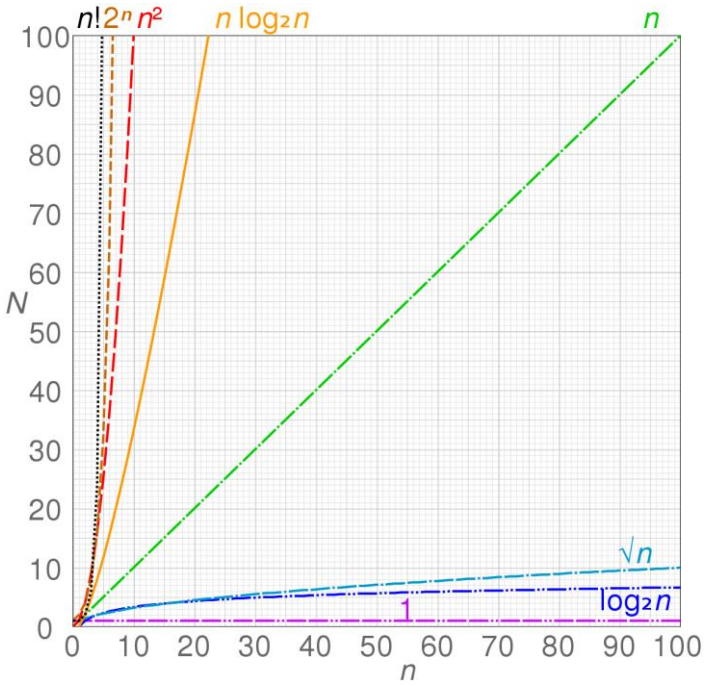
$O(f(n)) + O(g(n)) = O(f(n) + g(n))$.

$O(f(n))O(g(n)) = O(f(n)g(n))$.

$O(cf(n)) = O(f(n))$ for any constant c.

c is $O(1)$ for any constant c.

$\log_b n = O(\log n)$ for any base b.All of these rules (except #1) also hold for Q as well.



Shortcomings of asymptotic analysis:

In practice, other considerations beside asymptotic analysis are important when choosing between algorithms. Sometimes, an algorithm with worse asymptotic behavior is preferable. For the sake of this discussion, let algorithm *A* be asymptotically better than algorithm *B*. Here are some common issues with algorithms that have better asymptotic behavior:

- Implementation complexity: Algorithms with better complexity are often (much) more complicated. This can increase coding time and the constants.
- Small input sizes: Asymptotic analysis ignores small input sizes. At small input sizes, constant factors or low order terms could dominate running time, causing *B* to outperform *A*.
- Worst case versus average performance: If *A* has better worst case performance than *B*, but the average performance of *B* given the expected input is better, then *B* could be a better choice than *A*. Conversely, if the worst case performance of *B* is unacceptable (say for life-threatening or mission-critical reasons), *A* must still be used.

Complexity of algorithms from class:

	Average	Worst case
Binary search tree	--	O(max height of tree)
Red-black tree	--	O(log n)
Splay tree	O(log n)	O(n)
DFS, BFS	--	O(m+n)

	Insert	Minimum	Extract Min	Union	Decrease	Delete
Binary heap	O(log(n))	O(1)	O(log(n))	O(n)	O(log(n))	O(log(n))
Binomial heap	O(log(n))	O(log(n))	O(log(n))	O(1)	O(log(n))	O(log(n))

In designing of Algorithm, complexity analysis of an algorithm is an essential aspect. Mainly, algorithmic complexity is concerned about its performance, how fast or slow it works.

The complexity of an algorithm describes the efficiency of the algorithm in terms of the amount of the memory required to process the data and the processing time.

Complexity of an algorithm is analyzed in two perspectives: **Time** and **Space**.

Time Complexity

It's a function describing the amount of time required to run an algorithm in terms of the size of the input. "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.

Space Complexity

It's a function describing the amount of memory an algorithm takes in terms of the size of input to the algorithm. We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural *butfixed−length* units to measure this.

Space complexity is sometimes ignored because the space used is minimal and/or obvious, however sometimes it becomes as important an issue as time.

Lecture 3

Asymptotic Notations

Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.

Time function of an algorithm is represented by **T*n***, where **n** is the input size.

Different types of asymptotic notations are used to represent the complexity of an algorithm. Following asymptotic notations are used to calculate the running time complexity of an algorithm.

- **O** − Big Oh
- **Ω** − Big omega
- **θ** − Big theta
- **o** − Little Oh

- **ω** − Little omega

O: Asymptotic Upper Bound

'O' *BigOh*

is the most commonly used notation. A function ***fn*** can be represented is the order of ***gn*** that is ***Og(n)***, if there exists a value of positive integer **n** as $\mathbf{n_0}$ and a positive constant **c** such that $-f(n) \leqslant c.g(n)$ for $n > n0$ in all case

Hence, function ***gn*** is an upper bound for function ***fn***, as ***gn*** grows faster than ***fn***.

Example

Let us consider a given function, $f(n)=4.n3+10.n2+5.n+1$

Considering $g(n)=n3$,$f(n) \leqslant 5.g(n)$ for all the values of $n>2$

Hence, the complexity of ***fn*** can be represented as $O(g(n))$, i.e. $O(n3)$

Ω: Asymptotic Lower Bound

We say that $f(n)=\Omega(g(n))$ when there exists constant **c** that $f(n) \geqslant c.g(n)$ for all sufficiently large value of **n**. Here **n** is a positive integer. It means function ***g*** is a lower bound for function ***f***; after a certain value of ***n, f*** will never go below ***g***.

Example

Let us consider a given function, $f(n)=4.n3+10.n2+5.n+1$.

Considering $g(n)=n3$, $f(n) \geqslant 4.g(n)$ for all the values of $n>0$.

Hence, the complexity of ***fn*** can be represented as $\Omega(g(n))$, i.e. $\Omega(n3)$

θ: Asymptotic Tight Bound

We say that $f(n)=\theta(g(n))$

when there exist constants $\mathbf{c_1}$ and $\mathbf{c_2}$ that $c1.g(n) \leqslant f(n) \leqslant c2.g(n)$ for all sufficiently large value of **n**. Here **n** is a positive integer. This means function **g** is a tight bound for function **f**.

Example

Let us consider a given function, $f(n)=4.n3+10.n2+5.n+1$

Considering $g(n)=n3$, $4.g(n) \leqslant f(n) \leqslant 5.g(n)$ for all the large values of **n**.

Hence, the complexity of ***fn*** can be represented as $\theta(g(n))$, i.e. $\theta(n3)$.

O - Notation

The asymptotic upper bound provided by **O-notation** may or may not be asymptotically tight. The bound $2.n2=O(n2)$ is asymptotically tight, but the bound $2.n=O(n2)$ is not.We use **o-notation** to denote an upper bound that is not asymptotically tight.

We formally define ***og(n)*** *little−ohofgofn* as the set ***fn = og(n)*** for any positive constant $c>0$ and there exists a value $n0>0$, such that $0 \leqslant f(n) \leqslant c.g(n)$.

Intuitively, in the **o-notation**, the function ***fn*** becomes insignificant relative to ***gn*** as **n** approaches infinity; that is,

$$\lim n \to \infty (f(n)g(n))=0$$

Example

Let us consider the same function, $f(n)=4.n3+10.n2+5.n+1$

Considering $g(n)=n4$,

$$\lim n \to \infty (4.n3+10.n2+5.n+1n4)=0$$

Hence, the complexity of ***fn*** can be represented as $o(g(n))$, i.e. $o(n4)$.

ω – Notation

We use **ω-notation** to denote a lower bound that is not asymptotically tight. Formally, however, we define ***ωg(n)*** *little−omegaofgofn* as the set ***fn = ωg(n)*** for any positive constant ***C > 0*** and there exists a value $n0>0$, such that $0 \leqslant c.g(n)<f(n)$.

For example, $n22=\omega(n)$, but $n22 \neq \omega(n2)$. The relation $f(n)=\omega(g(n))$ implies that the following limit exists

$$\lim n \to \infty (f(n)g(n))=\infty$$

That is, ***fn*** becomes arbitrarily large relative to ***gn*** as **n** approaches infinity.

Example

Let us consider same function, $f(n)=4.n3+10.n2+5.n+1$

Considering $g(n)=n2$,

$$\lim n \to \infty (4.n3+10.n2+5.n+1n2)=\infty$$

Hence, the complexity of ***fn*** can be represented as $o(g(n))$, i.e. $\omega(n2)$

.

Apriori and Apostiari Analysis

Apriori analysis means, analysis is performed prior to running it on a specific system. This analysis is a stage where a function is defined using some theoretical model. Hence, we determine the time and space complexity of an algorithm by just looking at the algorithm rather than running it on a particular system with a different memory, processor, and compiler.

Apostiari analysis of an algorithm means we perform analysis of an algorithm only after running it on a system. It directly depends on the system and changes from system to system.

In an industry, we cannot perform Apostiari analysis as the software is generally made for an anonymous user, which runs it on a system different from those present in the industry.

In Apriori, it is the reason that we use asymptotic notations to determine time and space complexity as they change from computer to computer; however, asymptotically they are the same.

Lecture 4

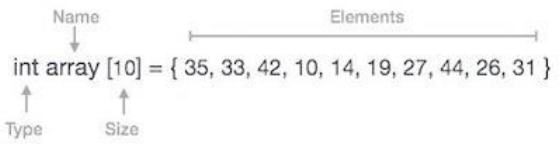
Arrays

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

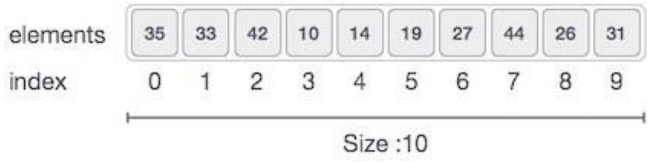
- **Element** − Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

Basic Operations

Following are the basic operations supported by an array.

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.

In C, when an array is initialized with size, then it assigns defaults values to its elements in following order.

Data Type	Default Value
Bool	false
Char	0
Int	0
Float	0.0
double	0.0f
Void	
wchar_t	0

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array −

Algorithm

Let **Array** be a linear unordered array of **MAX** elements.

Example

Result

Let **LA** be a Linear Array *unordered*

with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm where ITEM is inserted into the K^{th} position of LA –

```
1. Start
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
```

```
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>

main() {
   int LA[] = {1,3,5,7,8};
   int item = 10, k = 3, n = 5;
   int i = 0, j = n;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   n = n + 1;

   while( j >= k) {
      LA[j+1] = LA[j];
      j = j - 1;
   }

   LA[k] = item;

   printf("The array elements after insertion :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result –

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after insertion :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 10
LA[4] = 7
LA[5] = 8
```

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to delete an element available at the K^{th} position of LA.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
```

```
6. Set N = N-1
7. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>

void main() {
   int LA[] = {1,3,5,7,8};
   int k = 3, n = 5;
   int i, j;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   j = k;

   while( j < n) {
      LA[j-1] = LA[j];
      j = j + 1;
   }

   n = n -1;

   printf("The array elements after deletion :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after deletion :
LA[0] = 1
LA[1] = 3
LA[2] = 7
LA[3] = 8
```

Search Operation

You can perform a search for an array element based on its value or its index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>

void main() {
   int LA[] = {1,3,5,7,8};
   int item = 5, n = 5;
   int i = 0, j = 0;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   while( j < n){
      if( LA[j] == item ) {
         break;
      }

      j = j + 1;
   }

   printf("Found element %d at position %d\n", item, j+1);
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
Found element 5 at position 3
```

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to update an element available at the K^{th} position of LA.

```
1. Start
2. Set LA[K-1] = ITEM
3. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>

void main() {
   int LA[] = {1,3,5,7,8};
   int k = 3, n = 5, item = 10;
   int i, j;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   LA[k-1] = item;
```

```
   printf("The array elements after updation :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after updation :
LA[0] = 1
LA[1] = 3
LA[2] = 10
LA[3] = 7
LA[4] = 8
```

Lecture 5

Sparse Arrays

In computer science, a **sparse array** is an array in which most of the elements have the same value (known as the default value—usually 0 or null). A naive implementation of an array may allocate space for the entire array, but in the case where there are few non-default values, this implementation is inefficient. Typically the algorithm used instead of an ordinary array is determined by other known features (or statistical features) of the array, for instance if the sparsity is known in advance, or if the elements are arranged according to some function (e.g. occur in blocks).

As an example, a spreadsheet containing 100×100 mostly empty cells might be more efficiently stored as a linked list rather than an array containing ten thousand array elements. A heap memory allocator inside a program might choose to store regions of blank space inside a linked list rather than storing all of the allocated regions in, say a bit array.

What is Sparse Matrix?

In computer programming, a matrix can be defined with a 2-dimensional array. Any array with 'm' columns and 'n' rows represents a m X n matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.

Sparse matrix is a matrix which contains very few non-zero elements.

When a sparse matrix is represented with 2-dimensional array, we waste lot of space to represent that matrix. For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of matrix are filled with zero. That means, totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times. To make it simple we use the following sparse matrix representation.

Sparse Matrix Representations

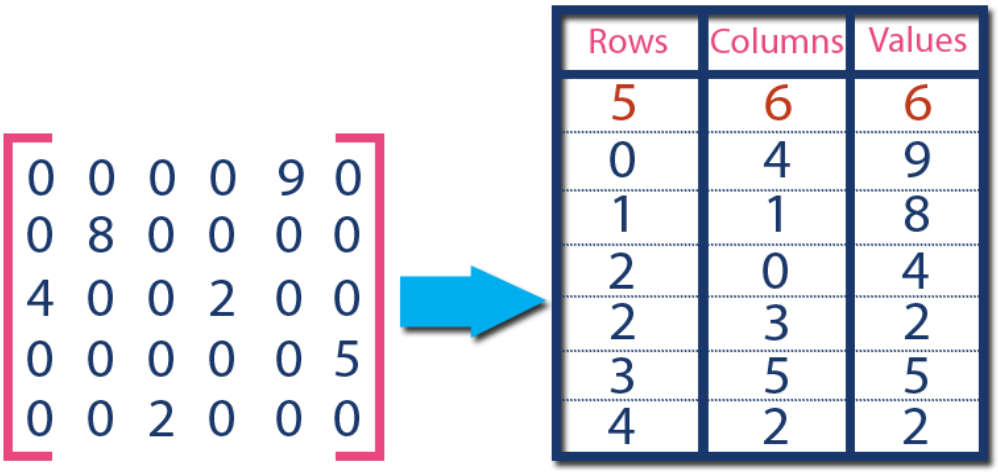
A sparse matrix can be represented by using TWO representations, those are as follows...

1. Triplet Representation (Array Representation)
2. Linked Representation

Triplet Representation (Array Representation)

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0^{th} row stores total number of rows, total number of columns and total number of non-zero values in the sparse matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image…



In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. Second row is filled with 0, 4, & 9 which indicates the non-zero value 9 is at 0th row 4th column in the Sparse matrix. In the same way the remaining non-zero values also follows the similar pattern.

Implementation of Array Representation of Sparse Matrix using C++

```
#include<iostream>

using namespace std;

int main()
{
    // sparse matrix of class 5x6 with 6 non-zero values
    int sparseMatrix[5][6] =
    {
        {0 , 0 , 0 , 0 , 9, 0 },
        {0 , 8 , 0 , 0 , 0, 0 },
        {4 , 0 , 0 , 2 , 0, 0 },
        {0 , 0 , 0 , 0 , 0, 5 },
        {0 , 0 , 2 , 0 , 0, 0 }
    };

    // Finding total non-zero values in the sparse matrix
    int size = 0;
    for (int row = 0; row < 5; row++)
        for (int column = 0; column < 6; column++)
            if (sparseMatrix[row][column] != 0)
                size++;

    // Defining result Matrix
    int resultMatrix[3][size];

    // Generating result matrix
    int k = 0;
    for (int row = 0; row < 5; row++)
        for (int column = 0; column < 6; column++)
            if (sparseMatrix[row][column] != 0)
```

```
            {
                resultMatrix[0][k] = row;
                resultMatrix[1][k] = column;
                resultMatrix[2][k] = sparseMatrix[row][column];
                k++;
            }

    // Displaying result matrix
    cout<<"Triplet Representation : "<<endl;
    for (int row=0; row<3; row++)
    {
        for (int column = 0; column<size; column++)
            cout<<resultMatrix[row][column]<<" ";

        cout<<endl;
    }
    return 0;
}
```

Lecture 6

String Operations

A string is a sequence of characters. In computer science, strings are more often used than numbers. We have all used text editors for editing programs and documents. Some of the Important Operations which are used on strings are: searching for a word, find -and -replace operations, etc.

There are many functions each can be defined on strings. Some important functions are :

1. String length : Determines length of a given string.
2. String concatenation : Concatenation of two or more strings. coping.
3. String copy : Creating another string which is a copy of the original or a copy of a part of the original.
4. String matching : Searching for a query string in given string.

String Algorithms:

STRING LENGTH

- Strings can have an arbitrary but finite length.
- There are two types of string data types:
    - Fixed length strings
    - Variable length strings
- Fixed length strings have a maximum length and all the strings uses same amount of space despite of their actual size.
- Variable length strings uses varying amount of memory depending on their actual size. Throughout of our discussion we strings are of variable length type.
- Variable length string is an array of characters terminated by a special character.
- To find the length of a string we scan through the string from left to right until we find the special symbol and each time i counter to keep track of number of characters scanned so far.

String Length

We assume that the given string STR is terminated by special symbol '\o'.

1. length = 0, i=0; //Indentity starts from '0'.
2. while STR[i] != '\0' //In C '\o' is used as end-of-string markes.

    i++;
    length=i;

3. return length

STRING CONCATENATION

- Appending one string to the end of another string is called string concatenation

Example let STR1= "hello"

STR2= "world"

- If we concatenate STR2 with STR1, then we get the string "helloworld"

Algorithm

```
1. i= 0, j=0;
2. while   STR1[i] != '\o'
   i++;
3. while    STR2[j] != '\o'
            STR1[i]= STR2[j];
            i =i+1
            j = j+1
4. STR1[i]= '\o';
5. Return  STR1;
```

STRING COPY

- By string copy, we mean copying one string to another string character by character.
- The size of the destination string should be greater than equal to the size of the source string.

Algorithm

```
1. Set i=0
2. while STR[i] != '\o'
            {
            STR2[i]=STR1[i];
            i =i+1;
            }
3. Set STR2[i]='\o'
4. Return STR2
```

STRING-MATCHING

- String matching is a most important problem.
- String matching consists of searching a query string (or pattern) *P* in a given text *T*.
- Generally the size of the pattern to be searched is smaller than the given text.
- There may be more than one occurrences of the pattern *P* in the text *T*. Sometimes we have to find all the occurrences of the pattern in the text.
- There are several applications of the string matching. Some of these are
  - Text editors
  - Search engines
  - Biological applications
- Since string-matching algorithms are used extensively, these should be efficient in terms of time and space.
- Let P [1..*m*] is the pattern to be searched and its size is m.
- T [1..*n*] is the given text whose size is *n*

- Assume that the pattern occurs in *T* at position (or shift) *i*. Then the output of the matching algorithm will be the integer i where 1 <= *i* <= *n-m*. If there are multiple occurrences of the pattern in the text, then sometimes it is required to output all the shifts where the pattern occurs.

Let Pattern *P* = CAT
Text = ABABNACATMAN
Then there is a match with the shift 7 in the text *T*

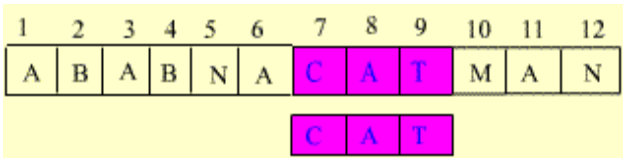


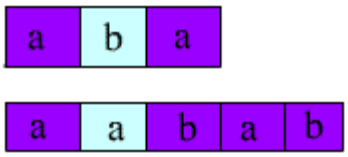
Fig : 1

Brute Force String Matching algorithm.

- This algorithm is a simple and obvious one, in which we compare a given pattern *P* with each of the sub strings of the text *T*, moving from left to right, until a match is found
- Let S_i is the substring of *T*, beginning at the *i* th position and whose length is same as pattern *P*.
- We compare *P*, character by character, with the first substring S_1. If all the corresponding characters are same, then the pattern *P* appears in *T* at shift 1. If some of the characters of S_1 are not matched with the corresponding characters of *P*, then we try for the next substring S_2. This procedure continues till the input text exhausts.
- In this algorithm we have to compare *P* with *n-m+1* substrings of *T*.

Example

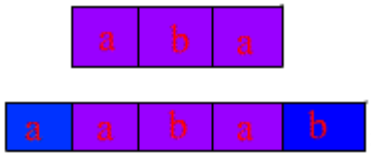
- Let *P* = aba
  T = aabab
- Compare *P* with 1st substring of *T*



Mismatch at the second character of *T*

Fig : 2(a)

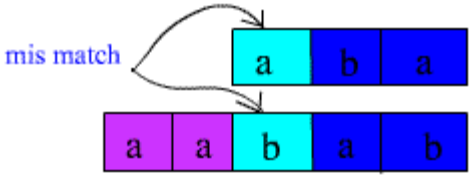
- Compare *P* with 2nd substring of *T*



Scince the corresponding character are same, there is a match at shift 1.

Fig : 2(b)

- Compare *P* with 3rd substring of *T*



Mismatch at the 3rd character of *T*

Fig : 2(c)

Algorithm For Brute-Force String matching

*

1. i = 1; [substring 1]
2. Repeat steps 3 to 5 while i <=n-m+1 do
3. for j= 1 to m [For each character of P]

   If P[j] != T[i+j-1] then
   goto step 5
4. Print "Pattern found at shift i "
5. i= i + 1
6. exit

- The complexity of the brute force string matching algorithm is O(nm)
- On average the inner loop runs fewer than m times to know that there is a mismatch.
- The worst case situation arises when first m character are matched for all substrings S_i. If pattern is of the $a^{m-1}b$ and text is of the form $a^{n-1}b$, where a^{n-1} denotes a repeated *n -1* times. In this case the inner loop runs exactly for m times before knowing that there is a mismatch. In this situation there will be exactly *m*(n-m+1)* number of comparisons.

Lecture 7

Searching algorithms and their complexity analysis

Linear: Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Algorithm

```
Linear Search ( Array A, Value x)

Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

Pseudocode

```
procedure linear_search (list, value)

   for each item in the list
      if match item == value
         return the item's location
      end if
```

```
   end for

end procedure
```

Binary: Binary search is a fast search algorithm with run-time complexity of *Ologn*. This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

How Binary Search Works?

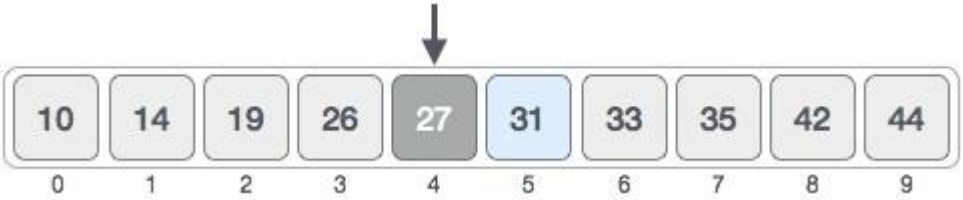
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −

```
mid = low + (high - low) / 2
```

Here it is, 0 + 9−0/ 2 = 4 *integer value of* 4.5

. So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

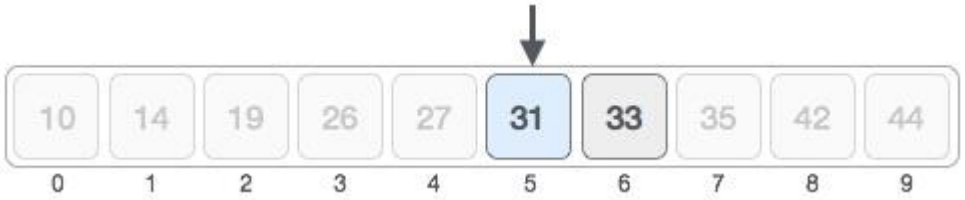
We change our low to mid + 1 and find the new mid value again.

```
low = mid + 1
mid = low + (high - low) / 2
```

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Pseudocode

The pseudocode of binary search algorithms should look like this −

```
Procedure binary_search
   A ← sorted array
   n ← size of array
   x ← value to be searched

   Set lowerBound = 1
   Set upperBound = n

   while x not found
      if upperBound < lowerBound
         EXIT: x does not exists.

      set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

      if A[midPoint] < x
         set lowerBound = midPoint + 1

      if A[midPoint] > x
         set upperBound = midPoint - 1

      if A[midPoint] = x
         EXIT: x found at location midPoint
   end while

end procedure
```

Interpolation search is an improved variant of binary search. This search algorithm works on the probing position of the required value. For this algorithm to work properly, the data collection should be in a sorted form and equally distributed.

Binary search has a huge advantage of time complexity over linear search. Linear search has worst-case complexity of O*n*

whereas binary search has O*logn*

.

There are cases where the location of target data may be known in advance. For example, in case of a telephone directory, if we want to search the telephone number of Morphius. Here, linear search and even binary search will seem slow as we can directly jump to memory space where the names start from 'M' are stored.

Positioning in Binary Search

In binary search, if the desired data is not found then the rest of the list is divided in two parts, lower and higher. The search is carried out in either of them.

Even when the data is sorted, binary search does not take advantage to probe the position of the desired data.

Position Probing in Interpolation Search

Interpolation search finds a particular item by computing the probe position. Initially, the probe position is the position of the middle most item of the collection.

If a match occurs, then the index of the item is returned. To split the list into two parts, we use the following method −

```
mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])

where −
   A    = list
   Lo   = Lowest index of the list
   Hi   = Highest index of the list
   A[n] = Value stored at index n in the list
```

If the middle item is greater than the item, then the probe position is again calculated in the sub-array to the right of the middle item. Otherwise, the item is searched in the subarray to the left of the middle item. This process continues on the sub-array as well until the size of subarray reduces to zero.

Runtime complexity of interpolation search algorithm is **O*log*(*logn***

) as compared to **O*logn***

of BST in favorable situations.

Algorithm

As it is an improvisation of the existing BST algorithm, we are mentioning the steps to search the 'target' data value index, using position probing −

```
Step 1 − Start searching data from middle of the list.
Step 2 − If it is a match, return the index of the item, and exit.
Step 3 − If it is not a match, probe position.
Step 4 − Divide the list using probing formula and find the new midle.
Step 5 − If data is greater than middle, search in higher sub-list.
Step 6 − If data is smaller than middle, search in lower sub-list.
Step 7 − Repeat until match.
```

Pseudocode

```
A → Array list
N → Size of A
X → Target Value

Procedure Interpolation_Search()

   Set Lo  →  0
   Set Mid → -1
   Set Hi  →  N-1

   While X does not match


      if Lo equals to Hi OR A[Lo] equals to A[Hi]
         EXIT: Failure, Target not found
      end if


      Set Mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])

      if A[Mid] = X
         EXIT: Success, Target found at Mid
      else
         if A[Mid] < X
            Set Lo to Mid+1
         else if A[Mid] > X
            Set Hi to Mid-1
         end if
      end if
   End While

End Procedure
```

Lecture 10

Pointers

Pointers:

Pointer is a variable contain the address of another variable. If a variable contains address of another variable than it is said that first variable points to second. All operation perform on pointers are done through two operators '*' and '&'. '&' is a unary operator that returns a memory address of a variable. '*' is complement of '&' and return value stored at a memory location stored in a pointer. '*' can interpreted as statement "at address" while '&' can be interpreted as statement "address of".

Pointer Declaration:

Declaring a pointer variable is quite similar to declaring a normal variable all you have to do is to insert a star '*' operator before it.General form of pointer declaration is -
datatype* name;
where datatype represent the type of data to which pointer thinks it is pointing to.
Multiple pointers of similar type can be declared in one statement but make sure you use * before every one otherwise they will become a variable of that type.
Example:

```
int *p;
float *f1,*f2;
char *ch;
```

Pointer Assignment:

Once we declare a pointer variable we must point it to a value by assigning the address of the variable
Example:

```
int *p;
int x;
p=&x;
```

The value of one pointer can be asssigned to another pointer using assignment operator '=' . In this value of right hand side points to memory address of variable stored in left hand side pointer. As a result both pointers point to same memory location after this expression.

Pointer of similar type can be used in expression easily as shown below but for diffrent type pointers you need to type cast them as shown in next section.

```
#include < stdio.h >
int main ()
{
char ch = 'x';
char *c1, *c2;
c1 = &ch;
c2 = c1; // Pointer Assignement Taking Place
printf (" *c1 = %c And *c2 = %c", *c1,*c2); // Prints 'x' twice
return 0;
}
```

Pointer Conversion

Before concept of pointer conversion you must understand the concept of a void pointer. Void pointer technically is a pointer which is pointing to the unknown. Void pointer has special property that it can be type casted into anyother pointer without any type casting though every other conversion needs an type casting. In dynamic memory allocation function such as malloc ( ) and calloc ( ) returns void pointer which can be easily converted to other types.

Also there is a pointer called null pointer which seems like void pointer but is entirely diffrent. Null pointer is a pointer which points to nothing. Null pointer points to the base address of the CPU register and since register is not addressable usage of a null pointer will lead to crash or at minimum a segmentation fault.

Also be careful while typecasting one pointer to another because even after type casting your pointer can point to anything but it will still think it is pointing to something of it declared type and have properties of the orignal type.

Type conversion is a powerful feature but yet it may lead difficult to remove bugs and crashes,it may also lead to unexpected and unreliable results but program would compile succesfully.

Code below shows a type casting of one pointer into another –

```
#include < stdio.h >
int main ()
{
int x=10;
char *ch;
int *p;
p = &x;
ch = (char *) p; // Type Casting and Pointer Conversion
printf (" *ch = %c And *p = %d", *ch,*p); // Output maybe unexpected depending on the compiler.
return 0;
}
```

Pointer Expressions:

Like normal variables pointer variable can be used in expressions.

Example:consider 2 integer pointers p1,p2

```
int *p1,*p2;
int x,y,z,k;
p1=&x;
p2=&y;
z=*p1**p2;
k=k+*p1;
z=10**p2/*p1;
```

Pointer Arithmetic

Pointer arithemetic is quite diffrent from normal arithemetic. Not all artihemetic operations are defined in pointers. You can increment them, decrement them, add and subtract integer values from them. You even can subtract two pointers.But you cannot add two pointers, mulitply, divide,modulus them. You can not also add or subtract values other than integer.

Address+Number=Address
Address-Number=Address
Address++=Address
Address--=Address

Now consider a pointer X , its current value that address it is pointing to is 1000 (just assuming).We make another assumption about the size of the data types. Size of data type is machine dependent, for example int can be 2,4 byte depending upon the compiler.
Now if this X pointer is char type(assumed 1 Byte) then X++ will have value 1001 and X-- will have value 999. Now if this X pointer is integer type (assumed 2 byte) then X++ will have value 1002 and X-- will have value 998. Again if this X pointer is float type (assumed 4 Byte) than X++ will have value 1004 and X-- will have value 996. Also if this X pointer is double type(assumed 8 Byte) than X++ will have value 1008 and X-- will have value 992.

when you increment a pointer of certain base type it increase it value in such a way that it points to next element of its base type. If you decrement a pointer its value decrease in such a way that it points to previous value of its base type.

You can add or subtract any integer value, in such case value of pointer get increase and decrease by the product of the value to be added or subtract and size of the base type. Pointer of user defined types such as structures and union also increase by the quantity of thier bit values which can be determined using sizeof operator.

Pointer Comparison:

Two pointers can be compared no matter where they point. Comparison can be done using <, >, =, <= and >= operators. Though it is not forcibly implied but comparison of two pointers become sensible only when they are related such as when they are pointing to element of same arrays.Comparison of two unrelated pointers is unpredictable and your code should not rely upon it. All comparison are generally done on basis of memory organization in the host machine.
Following C source code shows pointer comparison in C –

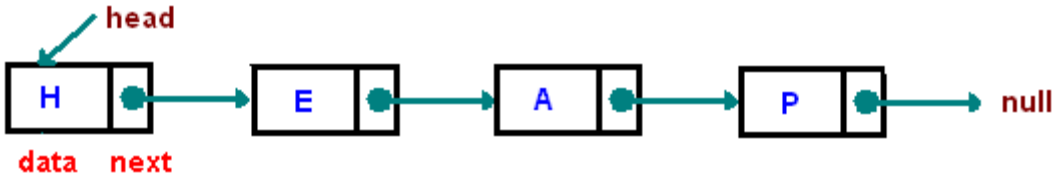
```
#include < stdio.h >
int main ()
{
int data[10],i;
int* p1,*p2;
for (i = 0; i <10;i++)
{
data[i] = i;
}
p1 = &data [1];
p2 = &data [2];
if (p1 > p2)
{
printf ("p1 is greater than p2\n");
}
else
{
printf ("p2 is greater than p1\n");
}
}
output:p2 is greater than p1
```

Lecture 11

Linked List

One disadvantage of using arrays to store data is that arrays are static structures and therefore cannot be easily extended or reduced to fit the data set. Arrays are also expensive to maintain new insertions and deletions. In this chapter we consider another data structure called Linked Lists that addresses some of the limitations of arrays.

A linked list is a linear data structure where each element is a separate object.



Each element (we will call it a **node**) of a list is comprising of two items - the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the **head** of the list. It should be noted that head is not a separate node, but the reference to the first node. If the list is empty then the head is a null reference.

A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list.

One disadvantage of a linked list against an array is that it does not allow direct access to the individual elements. If you want to access a particular item then you have to start at the head and follow the references until you get to that item.

Another disadvantage is that a linked list uses more memory compare with an array - we extra 4 bytes (on 32-bit CPU) to store a reference to the next node.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** − Item navigation is forward only.
- **Doubly Linked List** − Items can be navigated forward and backward.
- **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

Basic Operations

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

Why Linked List?
Arrays can be used to store linear data of similar types, but arrays have following limitations.
1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to be shifted.

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

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

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).
Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

Advantages over arrays
1) Dynamic size
2) Ease of insertion/deletion

Drawbacks:
1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation. Read about it here.
2) Extra memory space for a pointer is required with each element of the list.
3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Representation:
A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL.
Each node in a list consists of at least two parts:
1) data
2) Pointer (Or Reference) to the next node
In C, we can represent a node using structures. Below is an example of a linked list node with an integer data.
In Java, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

filter_none

edit

play_arrow

brightness_4

```
// A linked list node
struct Node
{
  int data;
  struct Node *next;
};
```

Linked list creation:

```
#include<stdio.h>
#include<stdlib.h>

struct Node
{
  int data;
  struct Node *next;
};

// Program to create a simple linked
// list with 3 nodes
int main()
{
  struct Node* head = NULL;
  struct Node* second = NULL;
  struct Node* third = NULL;

  // allocate 3 nodes in the heap
  head = (struct Node*)malloc(sizeof(struct Node));
  second = (struct Node*)malloc(sizeof(struct Node));
  third = (struct Node*)malloc(sizeof(struct Node));

  /* Three blocks have been allocated  dynamically.
     We have pointers to these three blocks as first,
     second and third
       head           second           third
        |               |               |
        |               |               |
    +---+-----+     +----+----+     +----+----+
    | #   | #   |     | #   | #   |     |   # |   # |
```

```
  +---+-----+     +----+----+     +----+----+

 # represents any random value.
 Data is random because we haven't assigned
 anything yet  */

head->data = 1; //assign data in first node
head->next = second; // Link first node with
                     // the second node

/* data has been assigned to data part of first
   block (block pointed by head).  And next
   pointer of first block points to second.
   So they both are linked.

     head          second         third
      |              |              |
      |              |              |
  +---+---+     +----+----+     +-----+----+
  | 1  | o----->| #  | #  |     |  #  | #  |
  +---+---+     +----+----+     +-----+----+
*/

// assign data to second node
second->data = 2;

// Link second node with the third node
second->next = third;

/* data has been assigned to data part of second
   block (block pointed by second). And next
   pointer of the second block points to third
   block. So all three blocks are linked.

     head         second         third
      |             |             |
      |             |             |
  +---+---+     +---+---+     +----+----+
  | 1  | o----->| 2 | o-----> |  # |  # |
  +---+---+     +---+---+     +----+----+      */

third->data = 3; //assign data to third node
third->next = NULL;

/* data has been assigned to data part of third
  block (block pointed by third). And next pointer
  of the third block is made NULL to indicate
  that the linked list is terminated here.

   We have the linked list ready.

        head
          |
          |
     +---+---+     +---+---+       +----+------+
     | 1  | o----->|  2  | o-----> |  3 | NULL |
     +---+---+     +---+---+       +----+------+


  Note that only head is sufficient to represent
  the whole list.  We can traverse the complete
  list by following next pointers.    */
```

```
  return 0;
}
```

How to traverse a linked list

Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.

When temp is `NULL`, we know that we have reached the end of linked list so we get out of the while loop.

```
struct node *temp = head;

printf("\n\nList elements are - \n");

while(temp != NULL)

{

     printf("%d --->",temp->data);

     temp = temp->next;

}
```

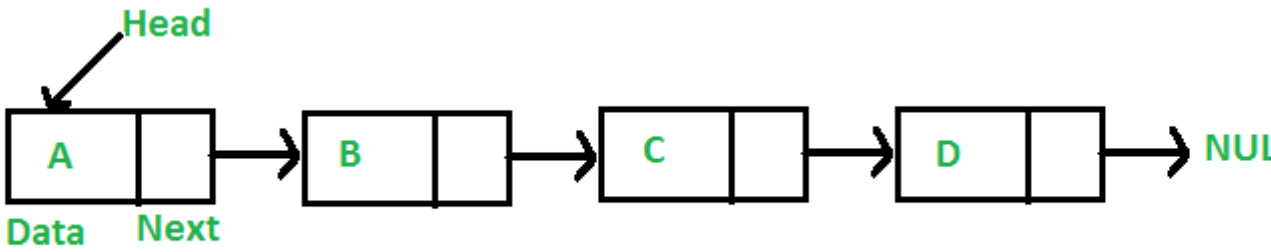
The output of this program will be:

```
List elements are -

1 --->2 --->3 --->
```

Types of Linked Lists

A **singly linked list** is described below:

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:

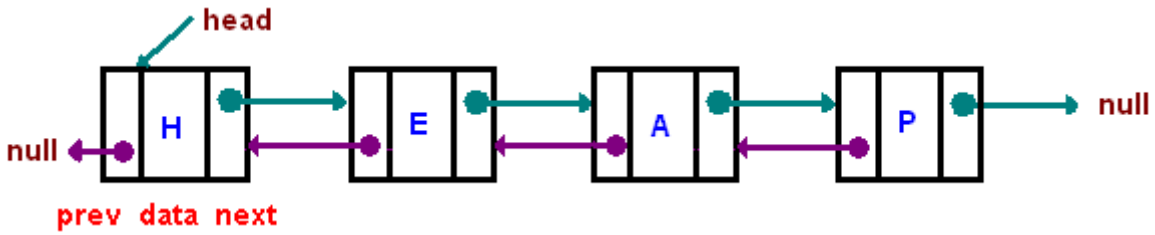


In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

Lecture 12

Doubly Linked List

A **doubly linked list** is a list that has two references, one to the next node and another to previous node.



Another important type of a linked list is called a **circular linked list** where last node of the list points back to the first node (or the head) of the list.

The Node class

In Java you are allowed to define a class (say, B) inside of another class (say, A). The class A is called the outer class, and the class B is called the **inner** class. The purpose of inner classes is purely to be used internally as helper classes. Here is the LinkedList class with the inner Node class

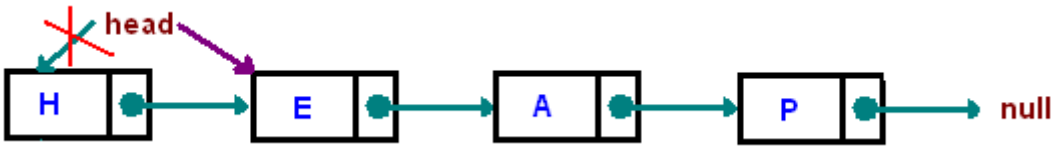
```
private static class Node<AnyType>
{
  private AnyType data;
  private Node<AnyType> next;

  public Node(AnyType data, Node<AnyType> next)
  {
    this.data = data;
    this.next = next;
  }
}
```

An inner class is a member of its enclosing class and has access to other members (inclusing private) of the outer class, And vise versa, the outer class can have a direct access to all members of the inner class. An inner class can be declared private, public, protected, or package private. There are two kind of inner classes: static and non-static. A static inner class cannot refer directly to instance variables or methods defined in its outer class: it can use them only through an object reference.

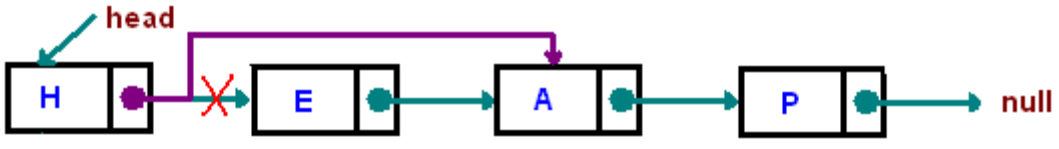
Examples

Let us assume the singly linked list above and trace down the effect of each fragment below. The list is restored to its initial state before each line executes

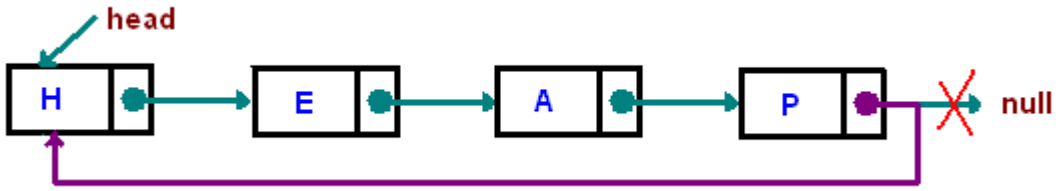
1. head = head.next;

2. 

3. head.next = head.next.next;

4. 

5. head.next.next.next.next = head;

6. 

Lecture 13

Linked List Operations: Inserting

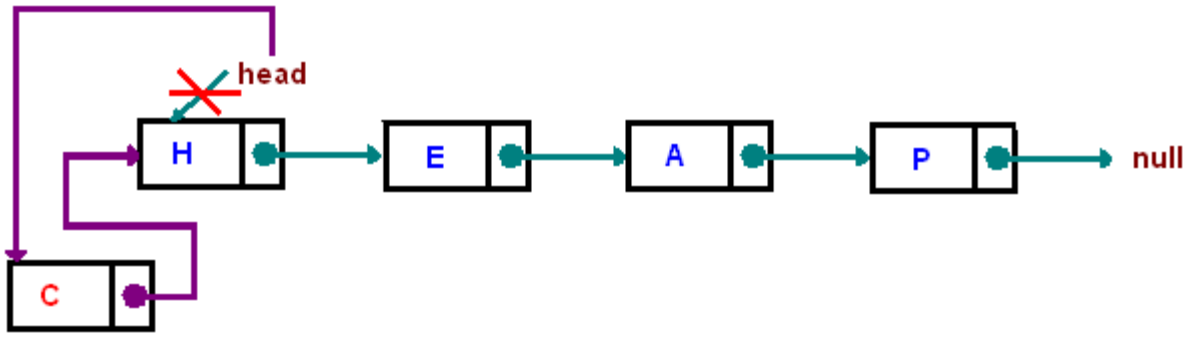
addFirst : The method creates a node and prepends it at the beginning of the list.

You can add elements to either beginning, middle or end of linked list.

Add to beginning

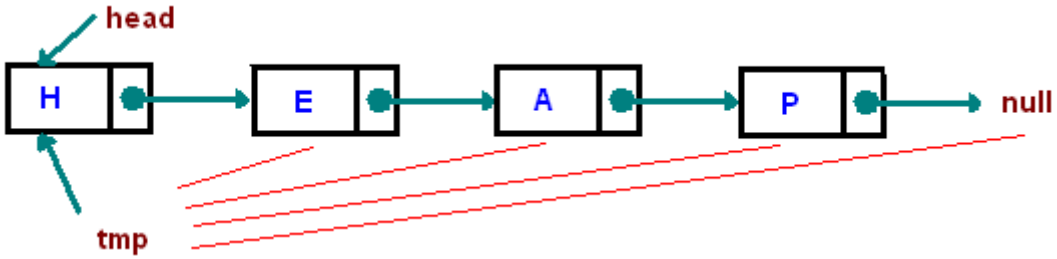
- Allocate memory for new node
- Store data
- Change next of new node to point to head
- Change head to point to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = head;
head = newNode;
```




```
public void addFirst(AnyType item)
{
  head = new Node<AnyType>(item, head);
}
```

Start with the head and access each node until you reach null. Do not change the head reference.



Node tmp = head;

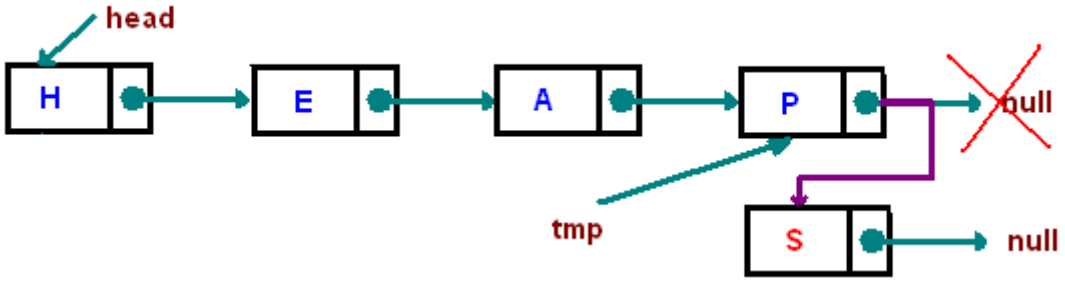
while(tmp != null) tmp = tmp.next;

addLast : The method appends the node to the end of the list. This requires traversing, but make sure you stop at the last node

Add to end

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = NULL;

struct node *temp = head;
while(temp->next != NULL){
  temp = temp->next;
}

temp->next = newNode;
```




```
public void addLast(AnyType item)
{
  if(head == null) addFirst(item);
  else
  {
    Node<AnyType> tmp = head;
    while(tmp.next != null) tmp = tmp.next;
```

```
    tmp.next = new Node<AnyType>(item, null);
  }
}
```

Lecture 14

Linked List Operations: Inserting

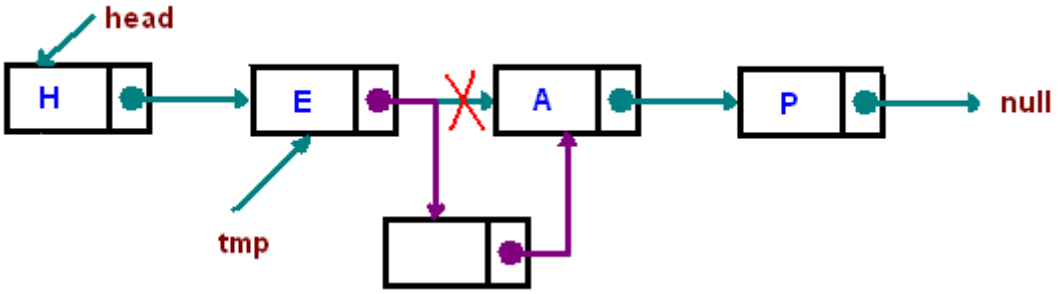
Inserting "after"

Find a node containing "key" and insert a new node after it. In the picture below, we insert a new node after "e":

Add to middle

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between

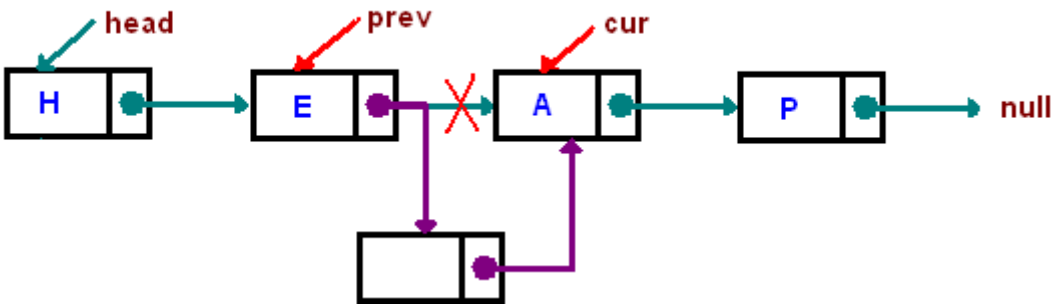
```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;

struct node *temp = head;

for(int i=2; i < position; i++) {
    if(temp->next != NULL) {
        temp = temp->next;
    }
}
newNode->next = temp->next;
temp->next = newNode;
```




```
public void insertAfter(AnyType key, AnyType toInsert)
{
  Node<AnyType> tmp = head;
  while(tmp != null && !tmp.data.equals(key)) tmp = tmp.next;

  if(tmp != null)
    tmp.next = new Node<AnyType>(toInsert, tmp.next);
}
```

Inserting "before"

Find a node containing "key" and insert a new node before that node. In the picture below, we insert a new node before "a":



For the sake of convenience, we maintain two references `prev` and `cur`. When we move along the list we shift these two references, keeping `prev` one step before `cur`. We continue until `cur` reaches the node before which we need to make an insertion. If `cur` reaches null, we don't insert, otherwise we insert a new node between `prev` and `cur`.

Examine this implementation

```
public void insertBefore(AnyType key, AnyType toInsert)
{
  if(head == null) return null;
  if(head.data.equals(key))
  {
    addFirst(toInsert);
    return;
  }

  Node<AnyType> prev = null;
  Node<AnyType> cur = head;

  while(cur != null && !cur.data.equals(key))
  {
    prev = cur;
    cur = cur.next;
  }
  //insert between cur and prev
  if(cur != null) prev.next = new Node<AnyType>(toInsert, cur);
}
```

Lecture 15

Linked List Operations: Deletion and Update

How to delete from a linked list

You can delete either from beginning, end or from a particular position.

Delete from beginning

- Point head to the second node

```
head = head->next;
```

Delete from end

- Traverse to second last element
- Change its next pointer to null

```
struct node* temp = head;
while(temp->next->next!=NULL){
  temp = temp->next;
}
temp->next = NULL;
```

Delete from middle

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

```
for(int i=2; i< position; i++) {
    if(temp->next!=NULL) {
        temp = temp->next;
    }
}

temp->next = temp->next->next;
```

Full Program Linked list:

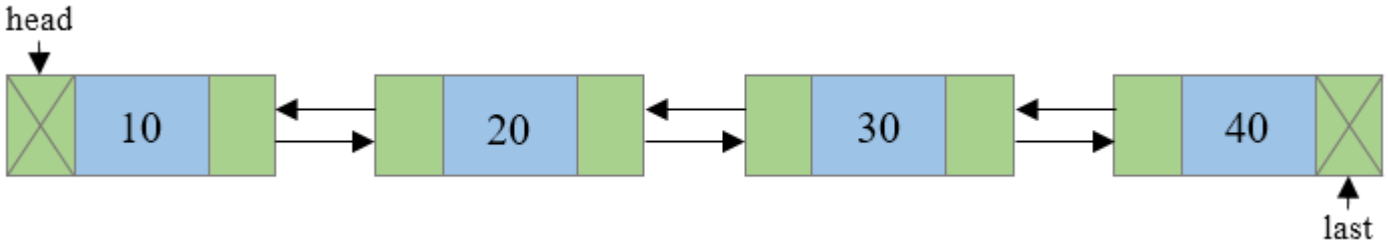
```
#include<stdio.h>
#include<stdlib.h>

struct node
{
  int data;
  struct node *next;
};

void display(struct node* head)
{
      struct node *temp = head;
      printf("\n\nList elements are - \n");
      while(temp != NULL)
      {
        printf("%d --->",temp->data);
        temp = temp->next;
      }
}
void insertAtFront(struct node** headRef, int value) {
```

```
    struct node* head = *headRef;

    struct node *newNode;
    newNode = malloc(sizeof(struct node));
    newNode->data = value;
    newNode->next = head;
    head = newNode;

    *headRef = head;
}
void insertAtEnd(struct node* head, int value){
    struct node *newNode;
    newNode = malloc(sizeof(struct node));
    newNode->data = value;
    newNode->next = NULL;

    struct node *temp = head;
    while(temp->next != NULL){
      temp = temp->next;
    }
    temp->next = newNode;
}
void insertAtMiddle(struct node *head, int position, int value) {

    struct node *newNode;
    newNode = malloc(sizeof(struct node));
    newNode->data = value;

    struct node *temp = head;
    int i;

for(i=2; i < position; i++) {
    if(temp->next != NULL) {
        temp = temp->next;
      }
   }
newNode->next = temp->next;
temp->next = newNode;
```

```
}

void deleteFromFront(struct node** headRef){
    struct node* head =  *headRef;
    head = head->next;
    *headRef = head;
}
void deleteFromEnd(struct node* head){
    struct node* temp = head;
    while(temp->next->next!=NULL){
      temp = temp->next;
    }
    temp->next = NULL;
}
void deleteFromMiddle(struct node* head, int position){
    struct node* temp = head;
    int i;
    for(i=2; i< position; i++) {
      if(temp->next!=NULL) {
        temp = temp->next;
      }
    }

  temp->next = temp->next->next;
}



int main() {
  /* Initialize nodes */
  struct node *head;
  struct node *one = NULL;
  struct node *two = NULL;
  struct node *three = NULL;

  /* Allocate memory */
  one = malloc(sizeof(struct node));
  two = malloc(sizeof(struct node));
  three = malloc(sizeof(struct node));
```

```
  /* Assign data values */
  one->data = 1;
  two->data = 2;
  three->data = 3;

  /* Connect nodes */
  one->next = two;
  two->next = three;
  three->next = NULL;

  /* Save address of first node in head */
  head = one;

  display(head); // 1 --->2 --->3 --->

  insertAtFront(&head, 4);
  display(head); // 4 --->1 --->2 --->3 --->

  insertAtEnd(head, 5);
  display(head); // 4 -->1 --->2 --->3 --->5 --->

  int position = 3;
  insertAtMiddle(head, position, 10);
  display(head); // 1 --->2 --->10 --->3 --->5 --->

  deleteFromFront(&head);
  display(head); // 1 --->10 --->2 --->3 --->5 --->

  deleteFromEnd(head);
  display(head); // 1 --->10 -->2 --->3 --->

  deleteFromMiddle(head, position);
  display(head); // 1 --->10 --->3 --->
}
```

Write a C program to implement Doubly linked list data structure. Write a C program to create a doubly linked list and display all nodes of the created list. How to create and display a doubly linked list in C. Algorithm to create and traverse doubly linked list.



Doubly Linked List

Algorithm to create a Doubly linked list

```
Algorithm to create Doubly Linked list
Begin:
    alloc (head)
    If (head == NULL) then
        write ('Unable to allocate memory')
    End if
    Else then
        read (data)
        head.data ← data;
        head.prev ← NULL;
        head.next ← NULL;
        last ← head;
        write ('List created successfully')
    End else
End
```

Algorithm to traverse or display Doubly linked list from beginning

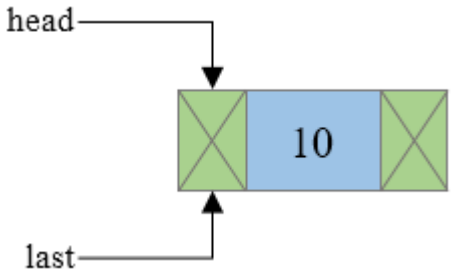
```
Algorithm to traverse Doubly Linked list from beginning
%% Input :  head {Pointer to the first node of the list}
Begin:
    If (head == NULL) then
        write ('List is empty')
    End if
    Else then
        temp ← head;
        While (temp != NULL) do
            write ('Data = ', temp.data)
            temp ← temp.next;
        End while
    End else
End
```

Algorithm to traverse or display Doubly linked list from end

```
Algorithm to traverse Doubly Linked list from end
%% Input :  last {Pointer to the last node of the list}
Begin:
    If (last == NULL) then
        write ('List is empty')
    End if
    Else then
        temp ← last;
        While (temp != NULL) do
            write ('Data = ', temp.data)
            temp ← temp.prev;
        End while
    End else
End
```

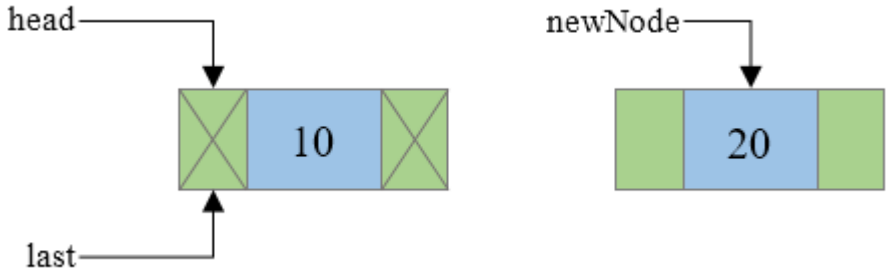
Steps to create Doubly linked list

1. Create a head node and assign some data to its data field.
2. Make sure that the previous and next address field of the head node must point to NULL.
3. Make the head node as last node.

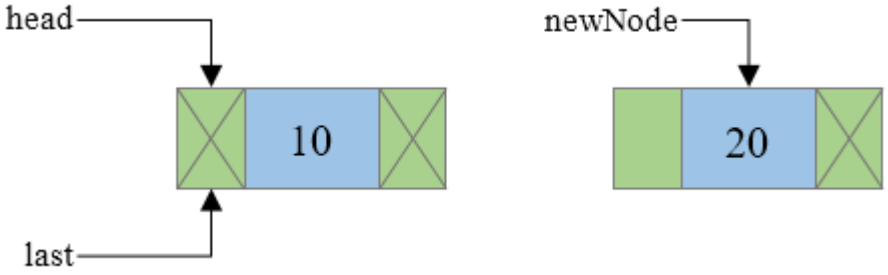


If you want to create more nodes then follow these steps:

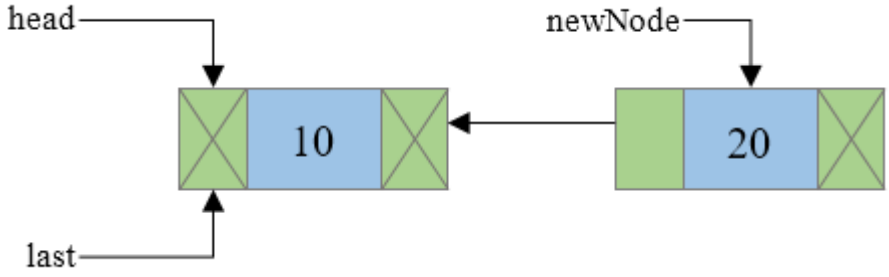
1. Create a new node and assign some data to its data field.



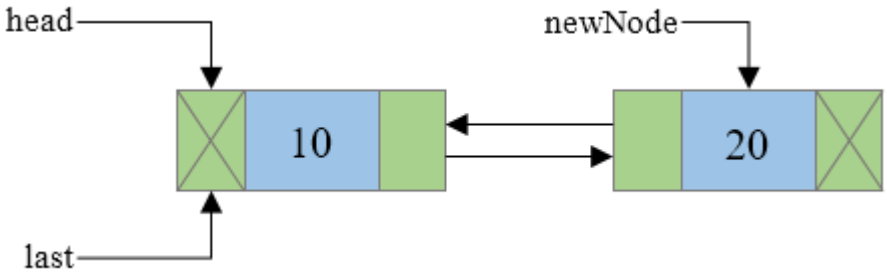
2. Make sure that the next address field of new node must point to NULL.



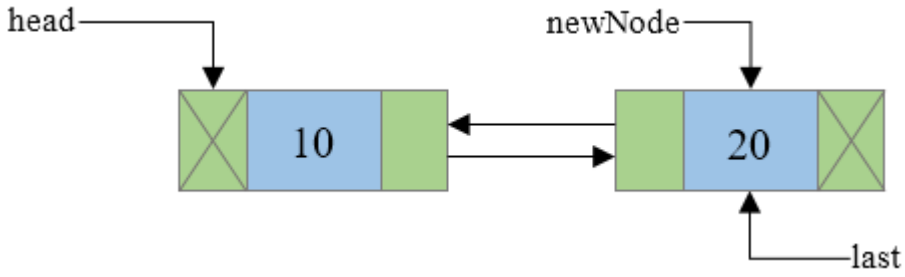
3. Link the new node previous address field with lastNode.



4. Link the lastNode next address field with newNode.



5. Move the lastNode to newNode i.e. last node will now point to new node.



6. Repeat Steps 4-8 if you want to add more nodes to the list.

Program to create and traverse doubly linked list

```
/**
 * C program to create and display Doubly linked list
 */

#include <stdio.h>
#include <stdlib.h>

/*
 * Basic structure of Node
 */
struct node {
    int data;
    struct node * prev;
    struct node * next;
}*head, *last;


/*
 * Function used in this program
 */
void createList(int n);
void displayListFromFirst();
void displayListFromEnd();


int main()
{
    int n, choice;

    head = NULL;
    last = NULL;

    printf("Enter the number of nodes you want to create: ");
    scanf("%d", &n);

    createList(n); // Create list of n nodes
```

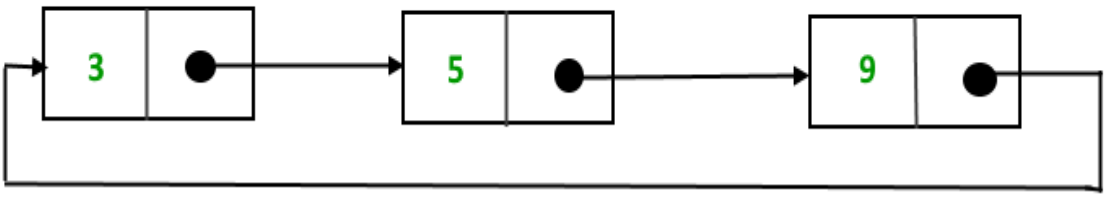
```
    printf("\nPress 1 to display list from First");
    printf("\nPress 2 to display list from End : ");
    scanf("%d", &choice);

    if(choice==1)
    {
        displayListFromFirst();
    }
    else if(choice == 2)
    {
        displayListFromEnd();
    }

    return 0;
}


/**
 * Create a doubly linked list of n nodes.
 * @n Number of nodes to be created
 */
void createList(int n)
{
    int i, data;
    struct node *newNode;

    if(n >= 1)
    {
        head = (struct node *)malloc(sizeof(struct node));

        if(head != NULL)
        {
            printf("Enter data of 1 node: ");
            scanf("%d", &data);

            head->data = data;
            head->prev = NULL;
            head->next = NULL;

            last = head;

            /*
             * Create rest of the n-1 nodes
             */
            for(i=2; i<=n; i++)
            {
                newNode = (struct node *)malloc(sizeof(struct node));

                if(newNode != NULL)
                {
                    printf("Enter data of %d node: ", i);
                    scanf("%d", &data);

                    newNode->data = data;
                    newNode->prev = last; // Link new node with the
previous node
                    newNode->next = NULL;

                    last->next = newNode; // Link previous node with the
new node
                    last = newNode;          // Make new node as
last/previous node
                }
                else
                {
                    printf("Unable to allocate memory.");
```

```
                    break;
                }
            }

            printf("\nDOUBLY LINKED LIST CREATED SUCCESSFULLY\n");
        }
        else
        {
            printf("Unable to allocate memory");
        }
    }
}


/**
 * Displays the content of the list from beginning to end
 */
void displayListFromFirst()
{
    struct node * temp;
    int n = 1;

    if(head == NULL)
    {
        printf("List is empty.");
    }
    else
    {
        temp = head;
        printf("\n\nDATA IN THE LIST:\n");

        while(temp != NULL)
        {
            printf("DATA of %d node = %d\n", n, temp->data);

            n++;

            /* Move the current pointer to next node */
            temp = temp->next;
        }
    }
}


/**
 * Display the content of the list from last to first
 */
void displayListFromEnd()
{
    struct node * temp;
    int n = 0;

    if(last == NULL)
    {
        printf("List is empty.");
    }
    else
    {
        temp = last;
        printf("\n\nDATA IN THE LIST:\n");

        while(temp != NULL)
        {
            printf("DATA of last-%d node = %d\n", n, temp->data);

            n++;
```

```
            /* Move the current pointer to previous node */
            temp = temp->prev;
        }
    }
}
```

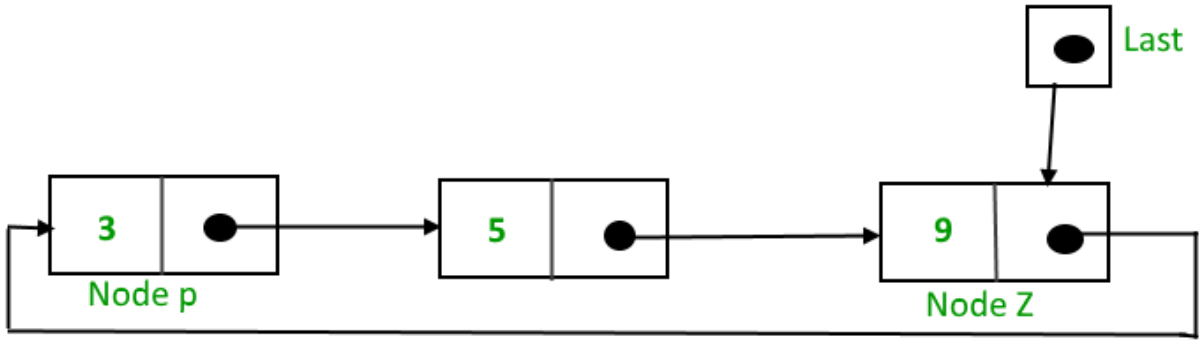
Why Circular? In a singly linked list, for accessing any node of linked list, we start traversing from the first node. If we are at any node in the middle of the list, then it is not possible to access nodes that precede the given node. This problem can be solved by slightly altering the structure of singly linked list. In a singly linked list, next part (pointer to next node) is NULL, if we utilize this link to point to the first node then we can reach preceding nodes. Refer this for more advantages of circular linked lists.

The structure thus formed is circular singly linked list look like this:



In this post, implementation and insertion of a node in a Circular Linked List using singly linked list are explained.

Implementation
To implement a circular singly linked list, we take an external pointer that points to the last node of the list. If we have a pointer last pointing to the last node, then last -> next will point to the first node.



The ponter *last* points to node Z and last -> next points to node P.

Why have we taken a pointer that points to the last node instead of first node ?
For insertion of node in the beginning we need traverse the whole list. Also, for insertion and the end, the whole list has to be traversed. If instead of *start* pointer we take a pointer to the last node then in both the cases there won't be any need to traverse the whole list. So insertion in the begging or at the end takes constant time irrespective of the length of the list.

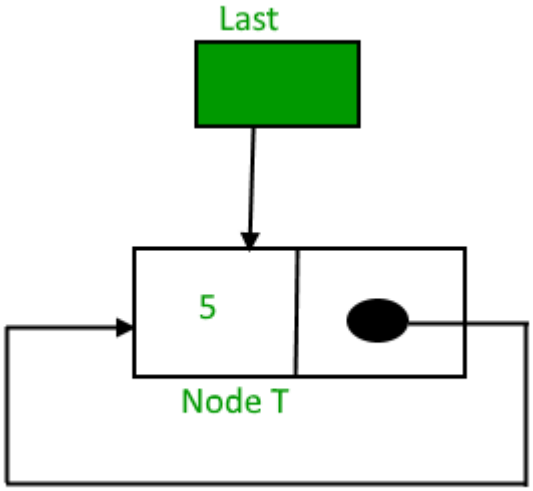
Insertion
A node can be added in three ways:

- Insertion in an empty list

- Insertion at the beginning of the list
- Insertion at the end of the list
- Insertion in between the nodes

Insertion in an empty List
Initially when the list is empty, *last* pointer will be NULL.

After inserting a node T,



After insertion, T is the last node so pointer *last* points to node T. And Node T is first and last node, so T is pointing to itself.
Function to insert node in an empty List,

filter_none

edit

play_arrow

brightness_4

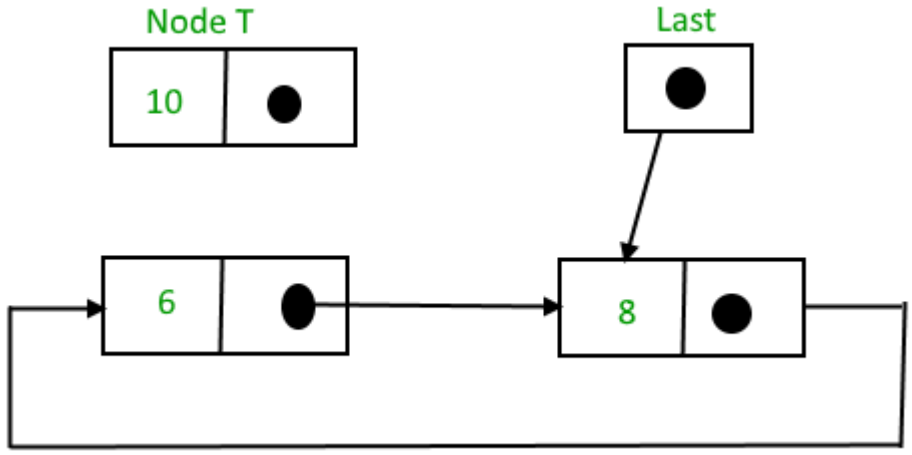
```
struct Node *addToEmpty(struct Node *last, int data)
{
    // This function is only for empty list
    if (last != NULL)
      return last;

    // Creating a node dynamically.
    struct Node *last =
          (struct Node*)malloc(sizeof(struct Node));

    // Assigning the data.
    last -> data = data;

    // Note : list was empty. We link single node
    // to itself.
    last -> next = last;

    return last;
```

```
}
```

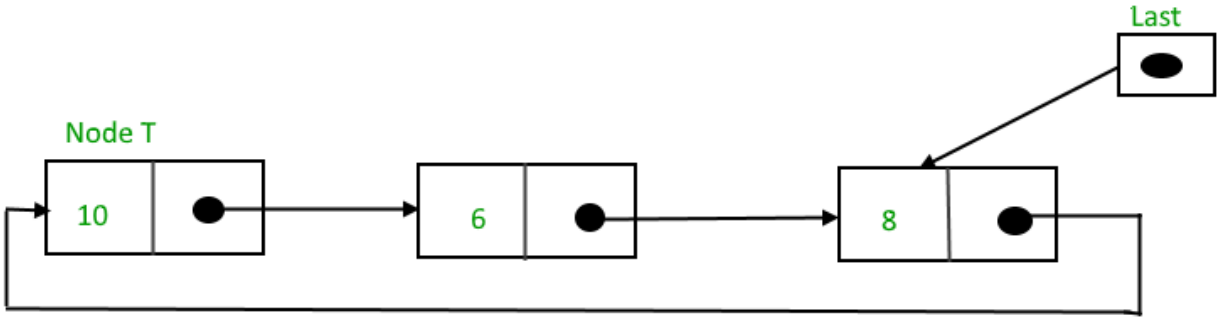
Insertion at the beginning of the list

To Insert a node at the beginning of the list, follow these step:

1. Create a node, say T.
2. Make T -> next = last -> next.
3. last -> next = T.



After insertion,



Function to insert node in the beginning of the List,

filter_none

edit

play_arrow

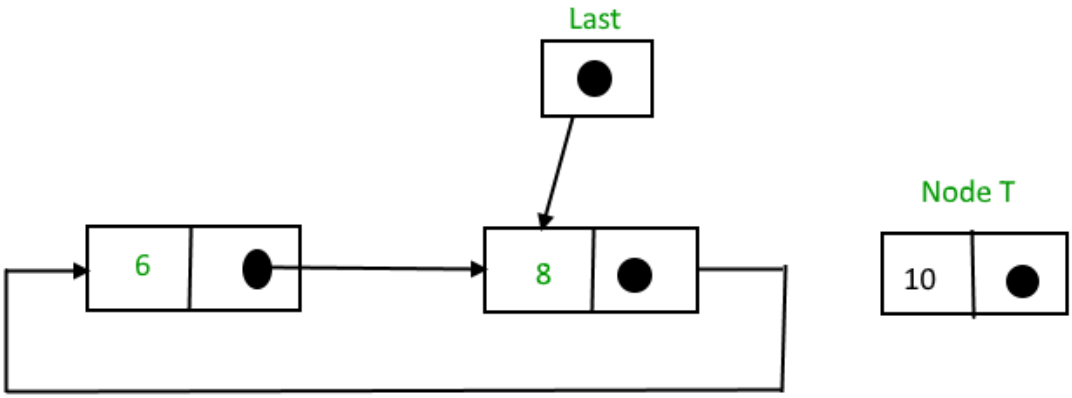
brightness_4

```
struct Node *addBegin(struct Node *last, int data)
{
  if (last == NULL)
     return addToEmpty(last, data);

  // Creating a node dynamically.
  struct Node *temp
        = (struct Node *)malloc(sizeof(struct Node));

  // Assigning the data.
  temp -> data = data;

  // Adjusting the links.
  temp -> next = last -> next;
  last -> next = temp;

  return last;
}
```

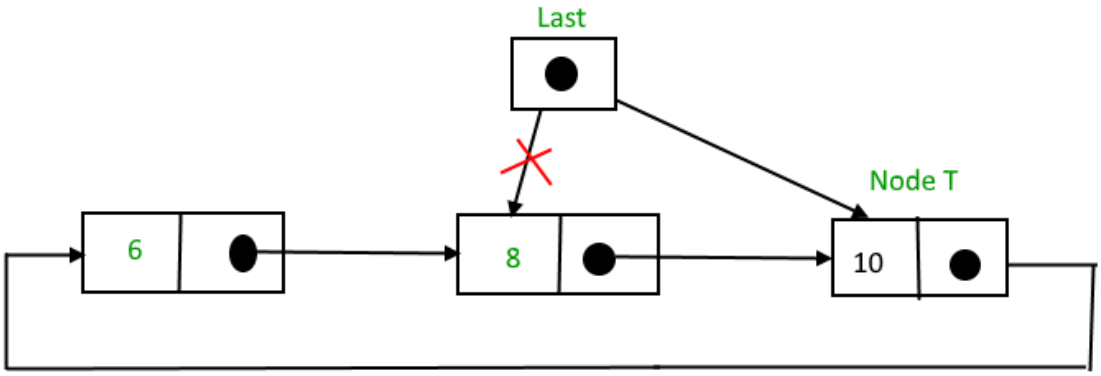
Insertion at the end of the list

To Insert a node at the end of the list, follow these step:

1. Create a node, say T.
2. Make T -> next = last -> next;
3. last -> next = T.
4. last = T.



After insertion,



Function to insert node in the end of the List,

filter_none

edit

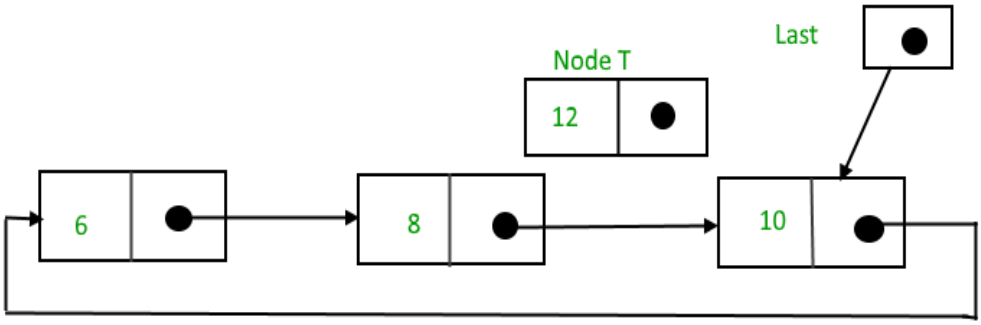
play_arrow

brightness_4

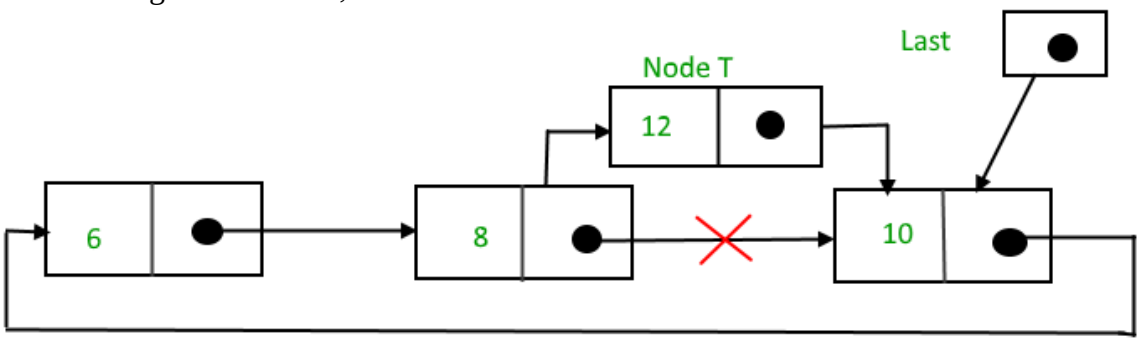
```
struct Node *addEnd(struct Node *last, int data)
{
  if (last == NULL)
     return addToEmpty(last, data);

  // Creating a node dynamically.
  struct Node *temp =
        (struct Node *)malloc(sizeof(struct Node));

  // Assigning the data.
  temp -> data = data;

  // Adjusting the links.
  temp -> next = last -> next;
  last -> next = temp;
  last = temp;

  return last;
}
```

Insertion in between the nodes
To Insert a node at the end of the list, follow these step:
1. Create a node, say T.
2. Search the node after which T need to be insert, say that node be P.
3. Make T -> next = P -> next;
4. P -> next = T.

Suppose 12 need to be insert after node having value 10,



After searching and insertion,



Function to insert node in the end of the List,

filter_none

edit

play_arrow

brightness_4

```
struct Node *addAfter(struct Node *last, int data, int item)
{
    if (last == NULL)
       return NULL;

    struct Node *temp, *p;
    p = last -> next;

    // Searching the item.
    do
    {
        if (p ->data == item)
        {
            // Creating a node dynamically.
            temp = (struct Node *)malloc(sizeof(struct Node));

            // Assigning the data.
            temp -> data = data;

            // Adjusting the links.
```

```
            temp -> next = p -> next;

            // Adding newly allocated node after p.
            p -> next = temp;

            // Checking for the last node.
            if (p == last)
                last = temp;

            return last;
        }
        p = p -> next;
    } while (p != last -> next);

    cout << item << " not present in the list." << endl;
    return last;
}
```

Following is a complete program that uses all of the above methods to create a circular singly linked list.

filter_none

edit

play_arrow

brightness_4

```
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    int data;
    struct Node *next;
};

struct Node *addToEmpty(struct Node *last, int data)
{
    // This function is only for empty list
    if (last != NULL)
      return last;

    // Creating a node dynamically.
    struct Node *temp =
           (struct Node*)malloc(sizeof(struct Node));

    // Assigning the data.
    temp -> data = data;
    last = temp;

    // Creating the link.
    last -> next = last;

    return last;
}

struct Node *addBegin(struct Node *last, int data)
{
    if (last == NULL)
        return addToEmpty(last, data);

    struct Node *temp =
            (struct Node *)malloc(sizeof(struct Node));
```

```
    temp -> data = data;
    temp -> next = last -> next;
    last -> next = temp;

    return last;
}

struct Node *addEnd(struct Node *last, int data)
{
    if (last == NULL)
        return addToEmpty(last, data);

    struct Node *temp =
        (struct Node *)malloc(sizeof(struct Node));

    temp -> data = data;
    temp -> next = last -> next;
    last -> next = temp;
    last = temp;

    return last;
}

struct Node *addAfter(struct Node *last, int data, int item)
{
    if (last == NULL)
        return NULL;

    struct Node *temp, *p;
    p = last -> next;
    do
    {
        if (p ->data == item)
        {
            temp = (struct Node *)malloc(sizeof(struct Node));
            temp -> data = data;
            temp -> next = p -> next;
            p -> next = temp;

            if (p == last)
                last = temp;
            return last;
        }
        p = p -> next;
    }  while(p != last -> next);

    cout << item << " not present in the list." << endl;
    return last;

}

void traverse(struct Node *last)
{
    struct Node *p;

    // If list is empty, return.
    if (last == NULL)
    {
        cout << "List is empty." << endl;
        return;
    }

    // Pointing to first Node of the list.
```

```
    p = last -> next;

    // Traversing the list.
    do
    {
        cout << p -> data << " ";
        p = p -> next;

    }
    while(p != last->next);

}

// Driven Program
int main()
{
    struct Node *last = NULL;

    last = addToEmpty(last, 6);
    last = addBegin(last, 4);
    last = addBegin(last, 2);
    last = addEnd(last, 8);
    last = addEnd(last, 12);
    last = addAfter(last, 10, 8);

    traverse(last);

    return 0;
}
```

Iterator

The whole idea of the iterator is to provide an access to a private aggregated data and at the same moment hiding the underlying representation. An iterator is Java is an object, and therefore it's implementation requires creating a class that implements the *Iterator* interface. Usually such class is implemented as a private inner class. The *Iterator* interface contains the following methods:

- AnyType next() - returns the next element in the container
- boolean hasNext() - checks if there is a next element
- void remove() - (optional operation).removes the element returned by next()

In this section we implement the Iterator in the LinkedList class. First of all we add a new method to the LinkedList class:

```
public Iterator<AnyType> iterator()
{
  return new LinkedListIterator();
}
```

Here `LinkedListIterator` is a private class inside the LinkedList class

```
private class LinkedListIterator implements Iterator<AnyType>
{
  private Node<AnyType> nextNode;

  public LinkedListIterator()
  {
    nextNode = head;
  }
  ...
}
```

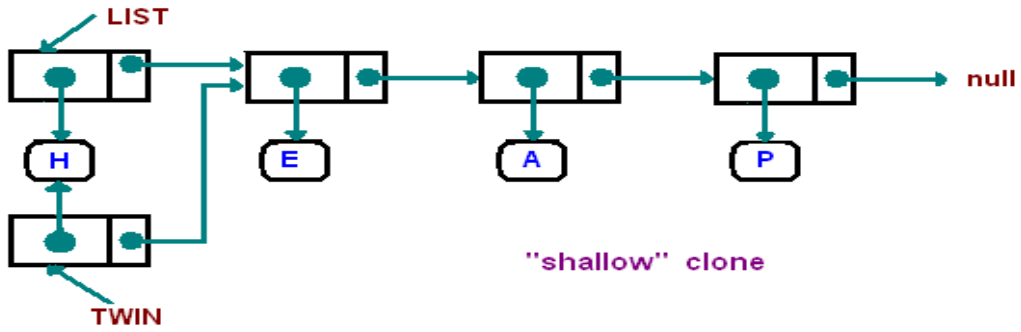
The `LinkedListIterator` class must provide implementations for `next()` and `hasNext()` methods. Here is the `next()` method:

```
public AnyType next()
{
```

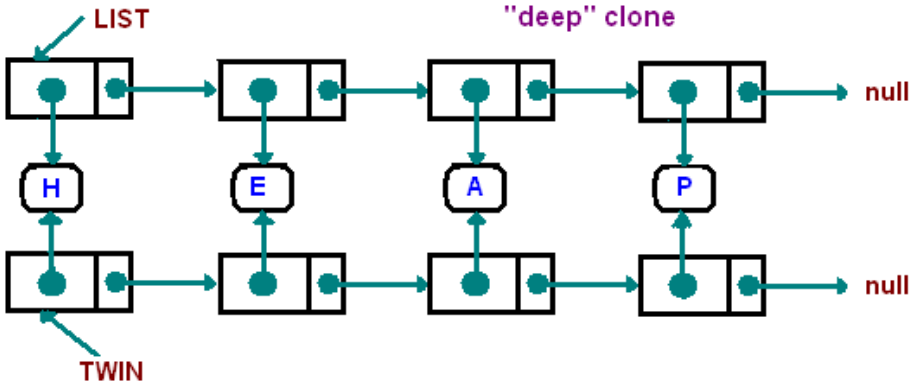
```
    if(!hasNext()) throw new NoSuchElementException();
    AnyType res = nextNode.data;
    nextNode = nextNode.next;
    return res;
  }
```

Cloning

Like for any other objects, we need to learn how to clone linked lists. If we simply use the clone() method from the Object class, we will get the following structure called a "shallow" copy:



The Object's clone() will create a copy of the first node, and share the rest. This is not exactly what we mean by "a copy of the object". What we actually want is a copy represented by the picture below



Since out data is immutable it's ok to have data shared between two linked lists. There are a few ideas to implement linked list copying. The simplest one is to traverse the original list and copy each node by using the addFirst() method. When this is finished, you will have a new list in the reverse order. Finally, we will have to reverse the list:

```
  public Object copy()
  {
    LinkedList<AnyType> twin = new LinkedList<AnyType>();
    Node<AnyType> tmp = head;
    while(tmp != null)
    {
      twin.addFirst( tmp.data );
      tmp = tmp.next;
    }

    return twin.reverse();
  }
```

A better way involves using a tail reference for the new list, adding each new node after the last node.

```
  public LinkedList<AnyType> copy3()
  {
    if(head==null) return null;
    LinkedList<AnyType> twin = new LinkedList<AnyType>();
    Node tmp = head;
    twin.head = new Node<AnyType>(head.data, null);
    Node tmpTwin = twin.head;

    while(tmp.next != null)
    {
      tmp = tmp.next;
```

```
        tmpTwin.next = new Node<AnyType>(tmp.data, null);
        tmpTwin = tmpTwin.next;
      }

      return twin;
    }
```

Applications:

Polynomial Algebra

The biggest integer that we can store in a variable of the type int is $2^{31} - 1$ on 32-but CPU. You can easily verify this by the following operations:

```
int prod=1;
for(int i = 1; i <=; 31; i ++)
        prod *= 2;
System.out.println(prod);
```

This code doesn't produce an error, it produces a result! The printed value is a *negative* integer $-2147483648 = -2^{31}$. If the value becomes too large, Java saves only the low order 32 (or 64 for longs) bits and throws the rest away.

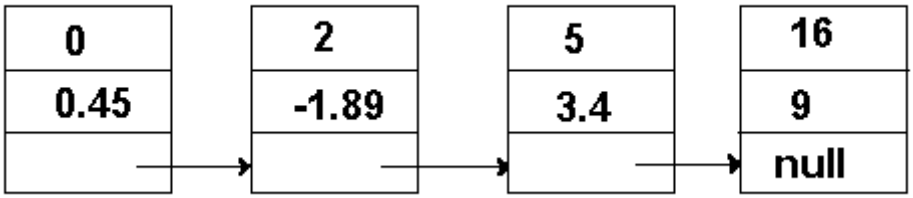
In real life applications we need to deal with integers that are larger than 64 bits (the size of a long). To manipulate with such big numbers, we will be using a linked list data structure. First we observe that each integer can be expressed in the decimal system of notation.

$937 = 9*10^2 + 3*10^1 + 7*10^0$
$2011 = 2*10^3 + 0*10^2 + 1*10^1 + 1*10^0$

Now, if we replace a decimal base 10 by a character, say 'x', we obtain a univariate polynomial, such as

$$0.45 - 1.89\ x^2 + 3.4\ x^5 + 9\ x^{16}$$

We will write an application that manipulates polynomials in one variable with real coefficients.Among many operations on polynomials, we implement addition, multiplication, differentiation and evaluation. A polynomial willbe represented as a linked list, where each node has an integer degree, a double coefficient and a reference to the next term. The final node will have a null reference to indicate the end of the list. Here is a linked link representation for the above polynomial:



Lecture# 19

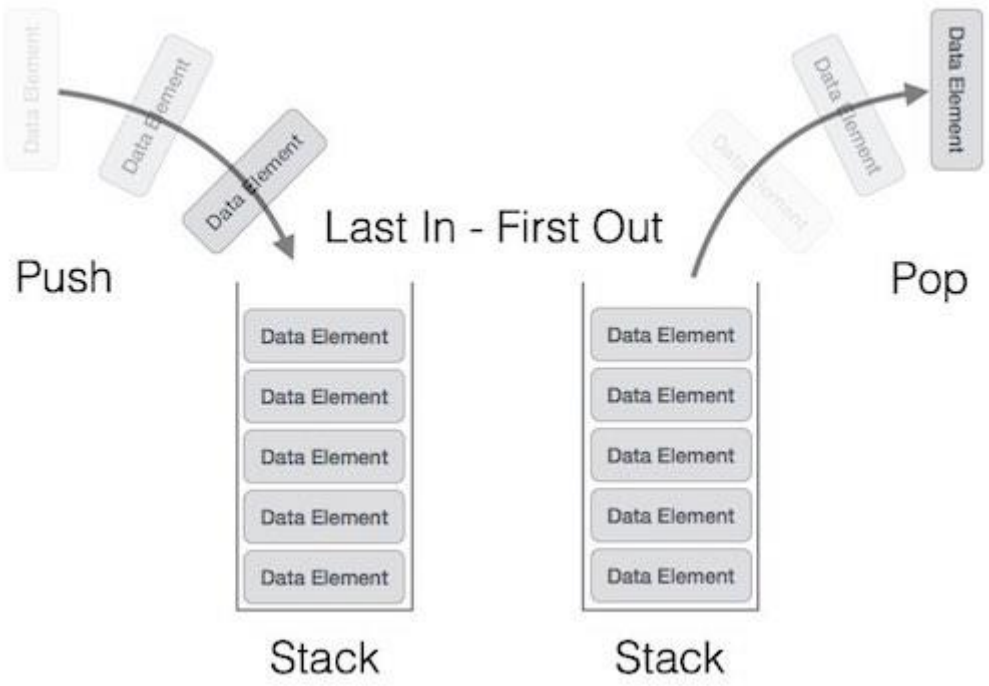
Stack:

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.
This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

peek()

Algorithm of peek() function −

```
begin procedure peek
   return stack[top]
end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek() {
   return stack[top];
}
```

isfull()

Algorithm of isfull() function −

```
begin procedure isfull
  if top equals to MAXSIZE
     return true
   else
     return false
   endif
end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull() {
   if(top == MAXSIZE)
     return true;
   else
     return false;
}
```

isempty()

Algorithm of isempty() function −

```
begin procedure isempty
  if top less than 1
     return true
   else
     return false
   endif
end procedure
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

Example

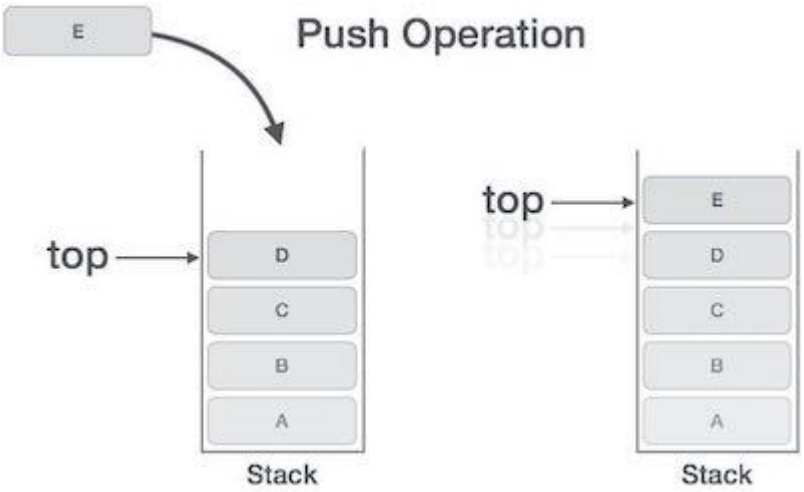
```
bool isempty() {
```

```
  if(top == -1)
    return true;
  else
    return false;
}
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data
  if stack is full
    return null
  endif
  top ← top + 1
  stack[top] ← data
end procedure
```

Implementation of this algorithm in C, is very easy. See the following code −

Example

```
void push(int data) {
  if(!isFull()) {
```

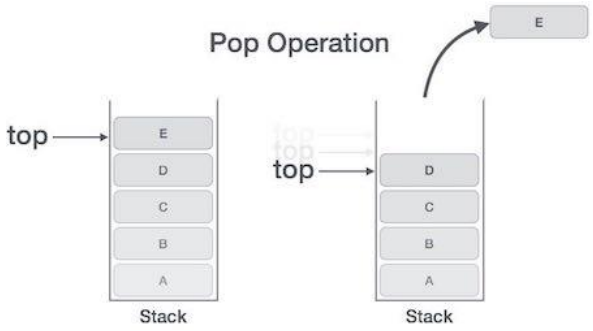
```
    top = top + 1;

    stack[top] = data;

  } else {

    printf("Could not insert data, Stack is full.\n");

  }

}
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

 if stack is empty

    return null

  endif

  data ← stack[top]

  top ← top - 1

  return data

end procedure
```

Implementation of this algorithm in C, is as follows −

Example

```
int pop(int data) {


  if(!isempty()) {
```

```
    data = stack[top];
    top = top - 1;
    return data;
  } else {
    printf("Could not retrieve data, Stack is empty.\n");
  }
}
```

Points to note:

a. A stack is simply another collection of data items and thus it would be possible to use exactly the same specification as the one used for our general collection. However, collections with the LIFO semantics of stacks are so important in computer science that it is appropriate to set up a limited specification appropriate to stacks only.

b. Although a linked list implementation of a stack is possible (adding and deleting from the head of a linked list produces exactly the LIFO semantics of a stack), the most common applications for stacks have a space restraint so that using an array implementation is a natural and efficient one (In most operating systems, allocation and de-allocation of memory is a relatively expensive operation, there is a penalty for the flexibility of linked list implementations.).

Lecture# 20

Stack applications:

Expression Evaluation: Stack is used to evaluate prefix, postfix and infix expressions.

Expression Conversion: An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.

Syntax Parsing: Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

Backtracking: Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.

Parenthesis Checking: Stack is used to check the proper opening and closing of parenthesis.

String Reversal: Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.

Function Call: Stack is used to keep information about the active functions or subroutines.

String reversing:

Reversing string is an operation of Stack by using Stack we can reverse any string, here we implemented a program in C - this will reverse given string using Stack.

The logic behind to implement this program:

1) Create an empty stack.

2) One by one push all characters of string to stack.

3) One by one pop all characters from stack and put them back to string.

```c
/*C program to Reverse String using STACK*/

#include <stdio.h>
#include <string.h>

#define MAX 100       /*maximum no. of characters*/

/*stack variables*/
int top=-1;
int item;
/****************/

/*string declaration*/
char stack_string[MAX];

/*function to push character (item)*/
void pushChar(char item);

/*function to pop character (item)*/
char popChar(void);

/*function to check stack is empty or not*/
int isEmpty(void);

/*function to check stack is full or not*/
int isFull(void);

int main()
{
   char str[MAX];

   int i;

   printf("Input a string: ");
   scanf("%[^\n]s",str); /*read string with spaces*/
   /*gets(str);-can be used to read string with spaces*/

   for(i=0;i<strlen(str);i++)
      pushChar(str[i]);

   for(i=0;i<strlen(str);i++)
      str[i]=popChar();

   printf("Reversed String is: %s\n",str);

   return 0;
}

/*function definition of pushChar*/
void pushChar(char item)
{
   /*check for full*/
   if(isFull())
   {
      printf("\nStack is FULL !!!\n");
      return;
   }

   /*increase top and push item in stack*/
   top=top+1;
   stack_string[top]=item;
}

/*function definition of popChar*/
```

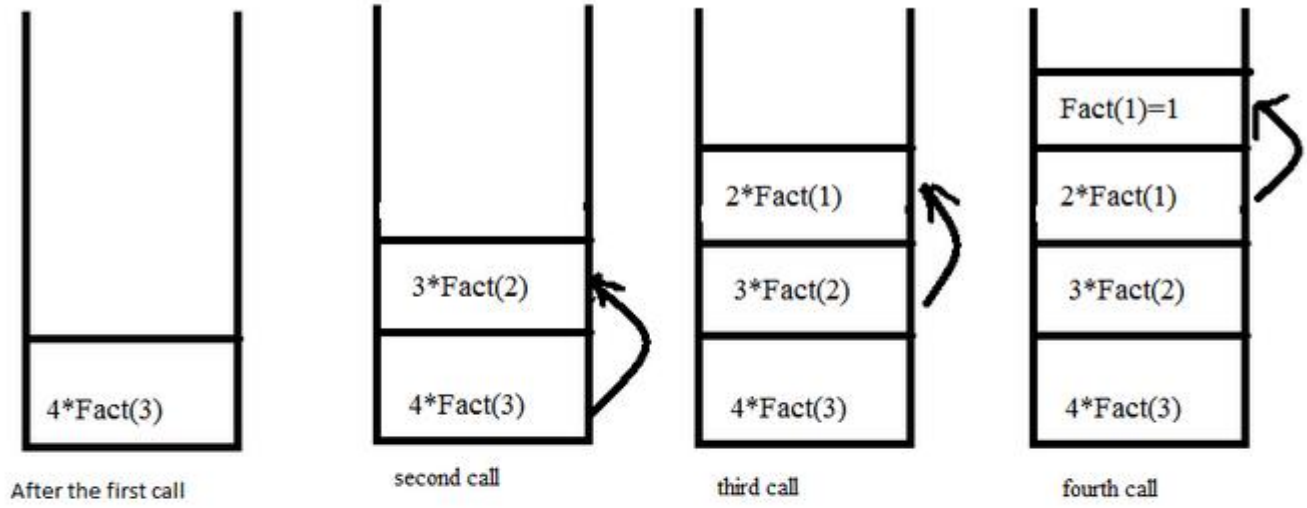
```
 char popChar()
{
   /*check for empty*/
   if(isEmpty())
   {
      printf("\nStack is EMPTY!!!\n");
      return 0;
   }

   /*pop item and decrease top*/
   item = stack_string[top];
   top=top-1;
   return item;
}

/*function definition of isEmpty*/
int isEmpty()
{
   if(top==-1)
      return 1;
   else
      return 0;
}

/*function definition of isFull*/
int isFull()
{
   if(top==MAX-1)
      return 1;
   else
      return 0;
}
```

Recursive Function Calling using stack:

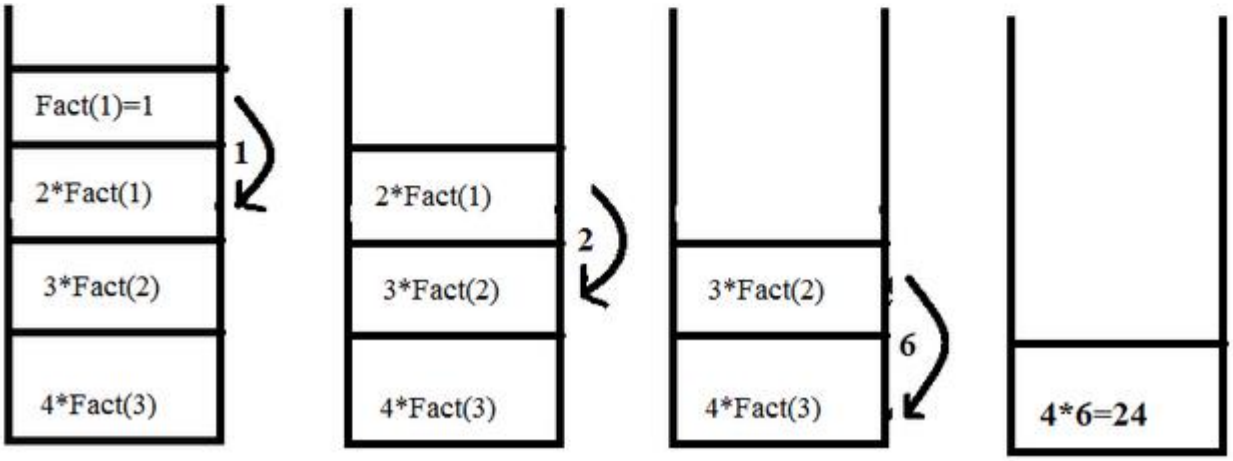
Factorial of a number:

```
#include <stdio.h>
int  Fact(int n){
   int f=1;
   if(n!=1){
   f=n*Fact(n-1);
   }
 return f;
}
int main()
{
   int n=0;
   printf("Type a number to get its factorial\n");
   scanf("%d",&n);
   printf("The factorial of %d is = %d \n",n,Fact(n));
   return 0;
}
```

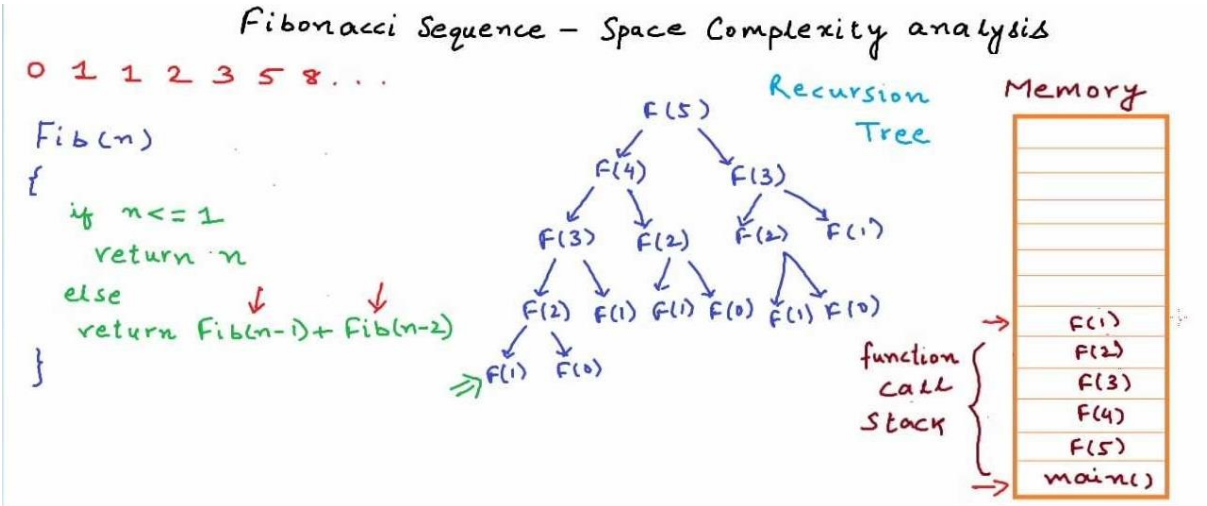
When function call happens previous variables gets stored in stack



Returning values from base case to caller function



Fibonacci sequence:



Program for Tower of Hanoi

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
1) Only one disk can be moved at a time.
2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3) No disk may be placed on top of a smaller disk.

Approach :

Take an example for 2 disks :

Let rod 1 = 'A', rod 2 = 'B', rod 3 = 'C'.

Step 1 : Shift first disk from 'A' to 'B'.

Step 2 : Shift second disk from 'A' to 'C'.

Step 3 : Shift first disk from 'B' to 'C'.

The pattern here is :

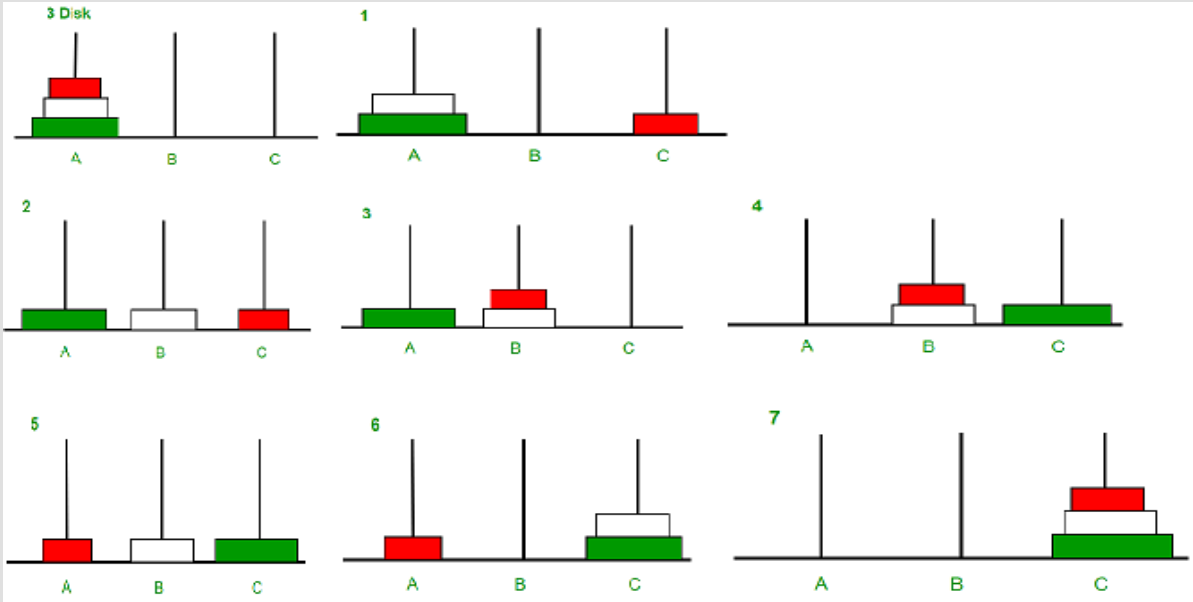
Shift 'n-1' disks from 'A' to 'B'.

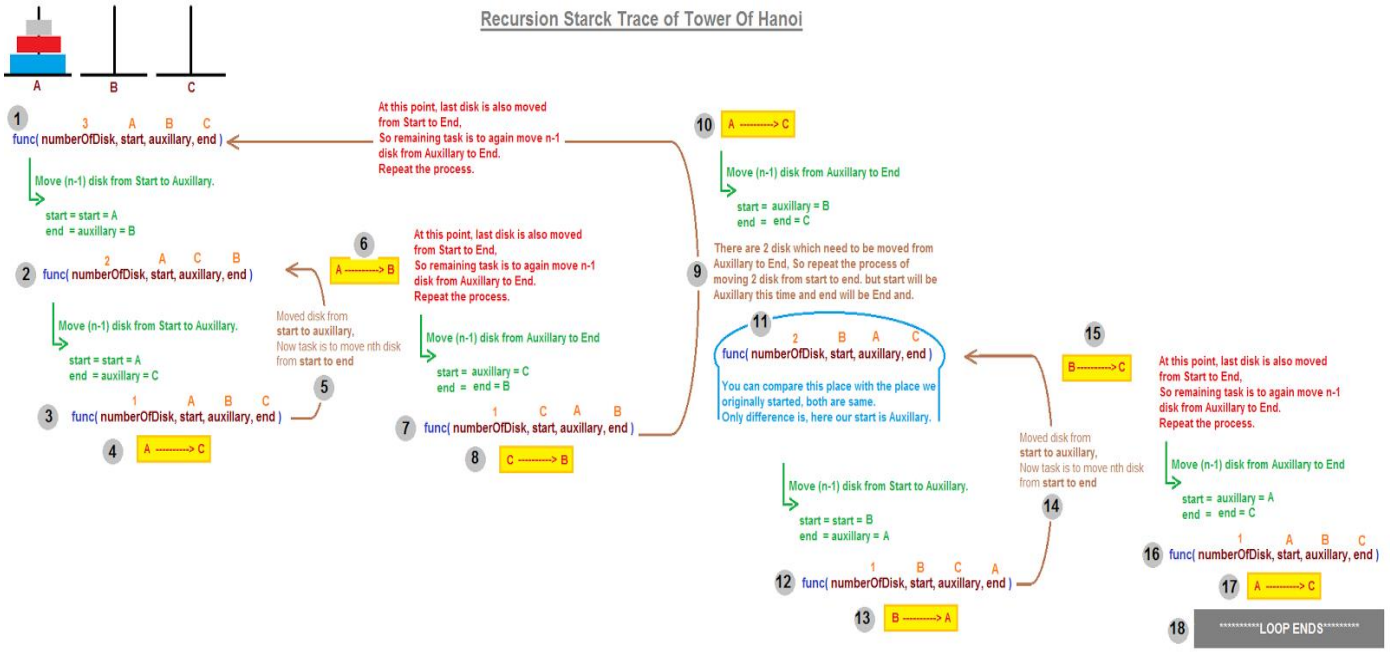
Shift last disk from 'A' to 'C'.

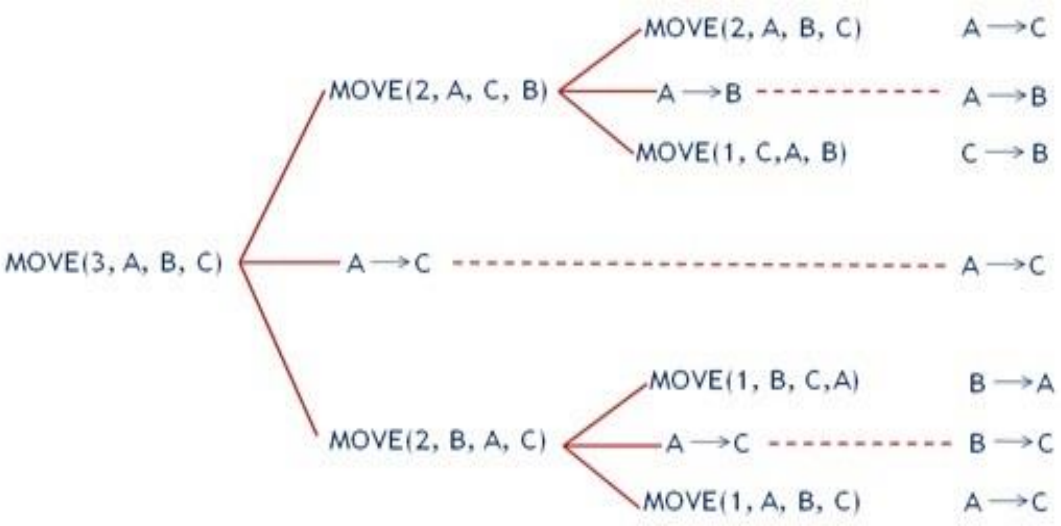
Shift 'n-1' disks from 'B' to 'C'.

```
Procedure Hanoi(disk, source, dest, aux)
  IF disk == 1, THEN
    move disk from source to dest
  ELSE
    Hanoi(disk - 1, source, aux, dest)    // Step 1
    move disk from source to dest         // Step 2
    Hanoi(disk - 1, aux, dest, source)    // Step 3
  END IF
END Procedure
```

Image illustration for 3 disks :







Lecture# 21

Data Structure - Expression Parsing

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

Infix Notation

We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

The following table briefly tries to show the difference in all three notations **-**

Sr. No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) ∗ c	∗ + a b c	a b + c ∗
3	a ∗ (b + c)	∗ a + b c	a b c + ∗
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) ∗ (c + d)	∗ + a b + c d	a b + c d + ∗
6	((a + b) ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

Parsing Expressions

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example −

As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is provided later.

Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as **(a + b) − c**.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) −

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (∗) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example −

In **a + b*c**, the expression part **b*c** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for **a + b** to be evaluated first, like **(a + b)*c**.

Postfix Evaluation Algorithm

We shall now look at the algorithm on how to evaluate postfix notation −

Step 1 − scan the expression from left to right
Step 2 − if it is an operand push it to stack
Step 3 − if it is an operator pull operand from stack and perform operation
Step 4 − store the output of step 3, back to stack
Step 5 − scan the expression until all operands are consumed
Step 6 − pop the stack and perform operation

Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

Infix to postfix expression:

Algorithm to convert Infix To Postfix

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

1. Push "(" onto Stack, and add ")" to the end of X.
2. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
3. If an operand is encountered, add it to Y.
4. If a left parenthesis is encountered, push it onto Stack.
5. If an operator is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
    2. Add operator to Stack.
    [End of If]
6. If a right parenthesis is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
    2. Remove the left Parenthesis.
    [End of If]
    [End of If]
7. END.

Advantage of Postfix Expression over Infix Expression

An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence).Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

Infix expression is: A+(B*C-(D/E^F)*G)*H

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack, that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack, that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Lecture# 22

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Types of Queues in Data Structure

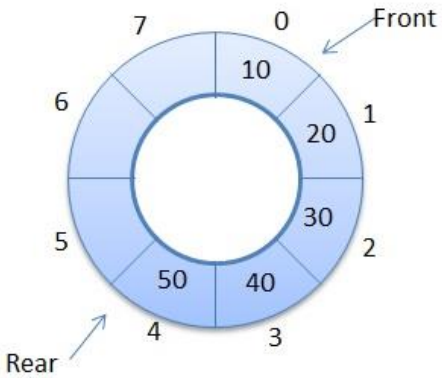
Simple Queue

As is clear from the name itself, simple queue lets us perform the operations simply. i.e., the insertion and deletions are performed likewise. Insertion occurs at the rear (end) of the queue and deletions are performed at the front (beginning) of the queue list.

All nodes are connected to each other in a sequential manner. The pointer of the first node points to the value of the second and so on.

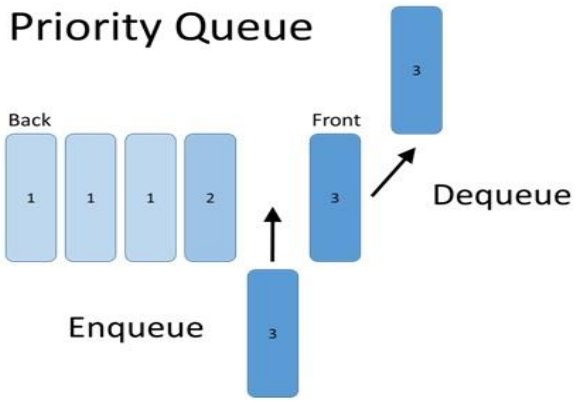
The first node has no pointer pointing towards it whereas the last node has no pointer pointing out from it.

Circular Queue



Unlike the simple queues, in a circular queue each node is connected to the next node in sequence but the last node's pointer is also connected to the first node's address. Hence, the last node and the first node also gets connected making a circular link overall.

Priority Queue



Priority queue makes data retrieval possible only through a predetermined priority number assigned to the data items.

While the deletion is performed in accordance to priority number (the data item with highest priority is removed first), insertion is performed only in the order.

Doubly Ended Queue (Dequeue)

The doubly ended queue or dequeue allows the insert and delete operations from both ends (front and rear) of the queue.

Queues are an important concept of the data structures and understanding their types is very necessary for working appropriately with them.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue −

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows –

Algorithm

```
begin procedure peek
   return queue[front]
end procedure
```

Implementation of peek() function in C programming language –

Example

```
int peek() {

   return queue[front];

}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function –

Algorithm

```
begin procedure isfull



   if rear equals to MAXSIZE

      return true

   else

      return false

   endif



end procedure
```

Implementation of isfull() function in C programming language –

Example

```
bool isfull() {

   if(rear == MAXSIZE - 1)

      return true;

   else

      return false;

}
```

isempty()

Algorithm of isempty() function –

Algorithm

```
begin procedure isempty



   if front is less than MIN  OR front is greater than rear

      return true
```

```
  else
    return false
  endif

end procedure
```

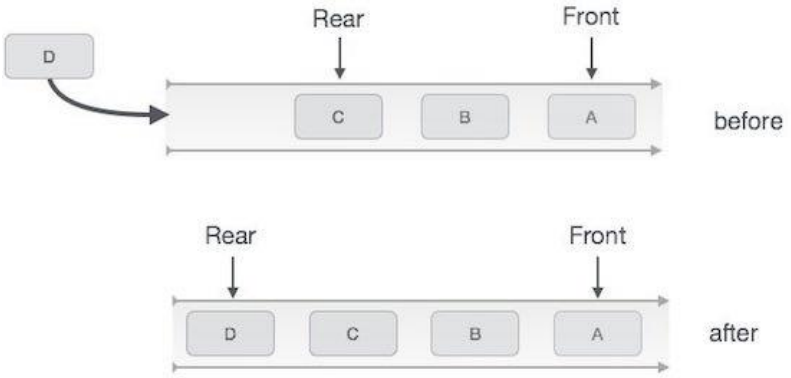
If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.
Here's the C programming code −
Example

```
bool isempty() {
  if(front < 0 || front > rear)
    return true;
  else
    return false;
}
```

Lecture# 23

Enqueue Operation
Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.
The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.
Algorithm for enqueue operation

```
procedure enqueue(data)

  if queue is full
    return overflow
  endif
```

```
   rear ← rear + 1

   queue[rear] ← data

   return true


end procedure
```

Implementation of enqueue() in C programming language −

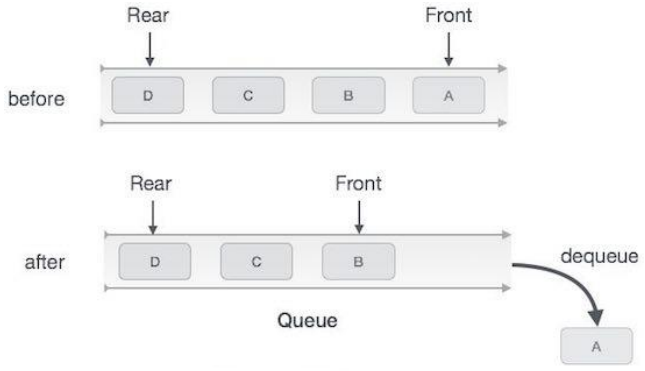
Example

```
int enqueue(int data)

   if(isfull())

      return 0;


   rear = rear + 1;

   queue[rear] = data;


   return 1;

end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue
```

```
   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1
   return true

end procedure
```

Implementation of dequeue() in C programming language −

Example

```
int dequeue() {
   if(isempty())
      return 0;

   int data = queue[front];
   front = front + 1;
   return data;
}
```

Lecture #24

Applications of queue:

Queue is used when things don't have to be processed immediately, but have to be processed in **F**irst **I**n **F**irst **O**ut order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

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.

- Queue is useful in CPU scheduling, Disk Scheduling. When multiple processes require CPU at the same time, various CPU scheduling algorithms are used which are implemented using Queue data structure.
- When data is transferred asynchronously between two processes. Queue is used for synchronization. Examples: IO Buffers, pipes, file IO, etc.
- In print spooling, documents are loaded into a buffer and then the printer pulls them off the buffer at its own rate. Spooling also lets you place a number of print jobs on a queue instead of waiting for each one to finish before specifying the next one.
- Breadth First search in a Graph .It is an algorithm for traversing or searching graph data structures. It starts at some arbitrary node of a graph and explores the neighbor nodes first, before moving to the next level neighbors.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.
- In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.

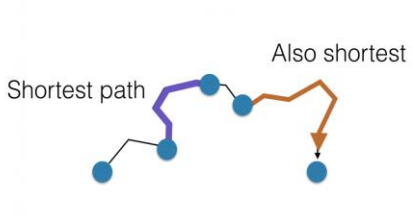
Dijkstra's Algorithm

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

It differs from minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

How Dijkstra's Algorithm works

Dijkstra's Algorithm works on the basis that any subpath `B -> D` of the shortest path `A -> D` between vertices A and D is also the shortest path between vertices B and D.



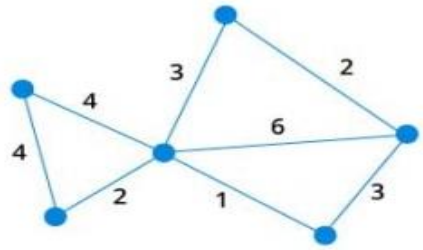
Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbours to find the shortest subpath to those neighbours. The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

Example of Dijkstra's algorithm

It is easier to start with an example and then think about the algorithm.

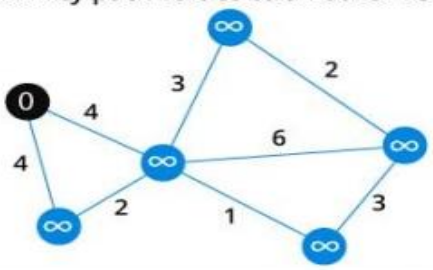
1

Start with a weighted graph



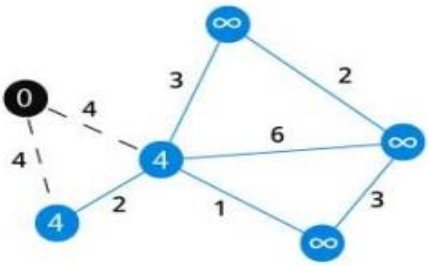
2

Choose a starting vertex and assign infinity path values to all other vertices



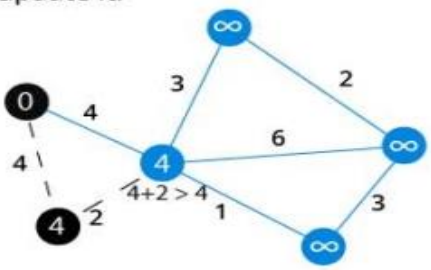
3

Go to each vertex adjacent to this vertex and update its path length



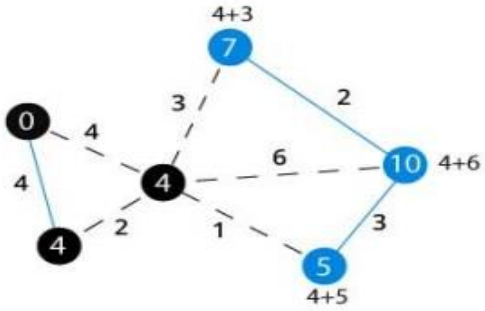
4

If the path length of adjacent vertex is lesser than new path length, don't update it.



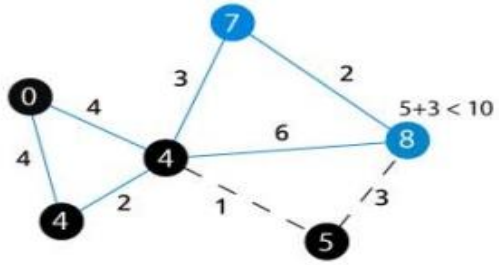
5

Avoid updating path lengths of already visited vertices



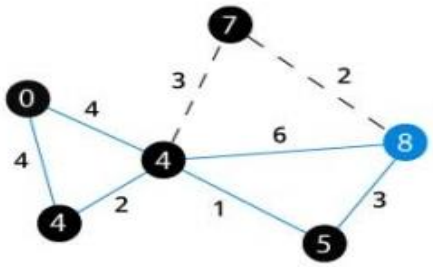
6

After each iteration, we pick the unvisited vertex with least path length. So we chose 5 before 7



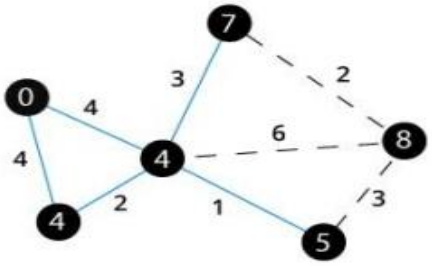
7

Notice how the rightmost vertex has its path length updated twice



8

Repeat until all the vertices have been visited



Djikstra's algorithm pseudocode

We need to maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices.

We also want to able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

A minimum priority queue can be used to efficiently receive the vertex with least path distance.

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

Queue using Stacks:

A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

Method 1 (By making enQueue operation costly) This method makes sure that oldest entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

enQueue(q, x):

- While stack1 is not empty, push everything from stack1 to stack2.
- Push x to stack1 (assuming size of stacks is unlimited).
- Push everything back to stack1.

Here time complexity will be O(n)

deQueue(q):

- If stack1 is empty then error
- Pop an item from stack1 and return it

Here time complexity will be O(1)

Below is the implementation of the above approach:

// CPP program to implement Queue using

// two stacks with costly enQueue()

```
#include <bits/stdc++.h>
using namespace std;
struct Queue {
        stack<int> s1, s2;

        void enQueue(int x)
        {
                // Move all elements from s1 to s2
                while (!s1.empty()) {
                        s2.push(s1.top());
                        s1.pop();
                }

                // Push item into s1
                s1.push(x);

                // Push everything back to s1
                while (!s2.empty()) {
                        s1.push(s2.top());
                        s2.pop();
                }
        }
```

```
// Dequeue an item from the queue
        int deQueue()
        {
                // if first stack is empty
                if (s1.empty()) {
                        cout << "Q is Empty";
                        exit(0);
                }
                // Return top of s1
                int x = s1.top();
                s1.pop();
                return x;
        }
};

// Driver code
int main()
{
        Queue q;
        q.enQueue(1);
        q.enQueue(2);
        q.enQueue(3);
        cout << q.deQueue() << '\n';
        cout << q.deQueue() << '\n';
        cout << q.deQueue() << '\n';
        return 0;
}
```

Method 2 (By making deQueue operation costly) In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

enQueue(q, x)

1) Push x to stack1 (assuming size of stacks is unlimited).

Here time complexity will be O(1)

deQueue(q)

1) If both stacks are empty then error.

2) If stack2 is empty

While stack1 is not empty, push everything from stack1 to stack2.

3) Pop the element from stack2 and return it.

Here time complexity will be O(n)

Method 2 is definitely better than method 1. Because, Method 1 moves all the elements twice in enQueue operation, while method 2 (in deQueue operation) moves the elements once and moves elements only if stack2 empty. So, the amortized complexity of the dequeue operation becomes $\Theta(1)$.

```
// CPP program to implement Queue using
// two stacks with costly deQueue()
#include <bits/stdc++.h>
using namespace std;

struct Queue {
        stack<int> s1, s2;

        // Enqueue an item to the queue
        void enQueue(int x)
        {
                // Push item into the first stack
                s1.push(x);
        }

        // Dequeue an item from the queue
        int deQueue()
        {
                // if both stacks are empty
                if (s1.empty() && s2.empty()) {
                        cout << "Q is empty";
                        exit(0);
                }
```

```
                // if s2 is empty, move
                // elements from s1
                if (s2.empty()) {
                        while (!s1.empty()) {
                                s2.push(s1.top());
                                s1.pop();
                        }
                }

                // return the top item from s2
                int x = s2.top();
                s2.pop();
                return x;
        }
};
// Driver code
int main()
{
        Queue q;
        q.enQueue(1);
        q.enQueue(2);
        q.enQueue(3);

        cout << q.deQueue() << '\n';
        cout << q.deQueue() << '\n';
        cout << q.deQueue() << '\n';

        return 0;
}
```

Queue can also be implemented using one user stack and one Function Call Stack. Below is modified Method 2 where recursion (or Function Call Stack) is used to implement queue using only one user defined stack.

```
enQueue(x):
  1) Push x to stack1.

deQueue:
  1) If stack1 is empty then error.
  2) If stack1 has only one element then return it.
  3) Recursively pop everything from the stack1, store the popped item
     in a variable res,  push the res back to stack1 and return res
```

The step 3 makes sure that the last popped item is always returned and since the recursion stops when there is only one item in *stack1* (step 2), we get the last element of *stack1* in deQueue() and all other items are pushed back in step

3. Implementation of method 2 using Function Call Stack:

```
// CPP program to implement Queue using
// one stack and recursive call stack.
#include <bits/stdc++.h>
using namespace std;

struct Queue {
        stack<int> s;

        // Enqueue an item to the queue
        void enQueue(int x)
        {
                s.push(x);
        }

        // Dequeue an item from the queue
        int deQueue()
        {
                if (s.empty()) {
                        cout << "Q is empty";
                        exit(0);
                }

                // pop an item from the stack
                int x = s.top();
                s.pop();

                // if stack becomes empty, return
                // the popped item
                if (s.empty())
                        return x;
                // recursive call
                int item = deQueue();
        // push popped item back to the stack
                s.push(x);
        // return the result of deQueue() call
                return item;
        }
};
// Driver code
int main()
{
        Queue q;
        q.enQueue(1);
        q.enQueue(2);
        q.enQueue(3);
        cout << q.deQueue() << '\n';
        cout << q.deQueue() << '\n';
        cout << q.deQueue() << '\n';
        return 0;
}
```

Stack using Queue:

We are given a Queue data structure that supports standard operations like enqueue() and dequeue(). We need to implement a Stack data structure using only instances of Queue and queue operations allowed on the instances.

A stack can be implemented using two queues. Let stack to be implemented be ‘s’ and queues used to implement be ‘q1’ and ‘q2’. Stack ‘s’ can be implemented in two ways:

Method 1 (By making push operation costly)
This method makes sure that newly entered element is always at the front of ‘q1’, so that pop operation just dequeues from ‘q1’. ‘q2’ is used to put every new element at front of ‘q1’.

1. **push(s, x)** operation’s step are described below:
    - Enqueue x to q2
    - One by one dequeue everything from q1 and enqueue to q2.
    - Swap the names of q1 and q2
2. **pop(s)** operation’s function are described below:
    - Dequeue an item from q1 and return it.

Below is the implementation of the above approach:

```
/* Program to implement a stack using
two queue */
#include <bits/stdc++.h>
using namespace std;

class Stack {
        // Two inbuilt queues
        queue<int> q1, q2;
        // To maintain current number of
        // elements
        int curr_size;

public:
        Stack()
        {
                curr_size = 0;
        }

        void push(int x)
        {
                curr_size++;

                // Push x first in empty q2
                q2.push(x);

                // Push all the remaining
                // elements in q1 to q2.
                while (!q1.empty()) {
                        q2.push(q1.front());
                        q1.pop();
                }

                // swap the names of two queues
                queue<int> q = q1;
                q1 = q2;
                q2 = q;
        }
```

```
void pop()
        {

                // if no elements are there in q1
                if (q1.empty())
                        return;
                q1.pop();
                curr_size--;
        }

        int top()
        {
                if (q1.empty())
                        return -1;
                return q1.front();
        }

        int size()
        {
                return curr_size;
        }
};

// Driver code
int main()
{
        Stack s;
        s.push(1);
        s.push(2);
        s.push(3);

        cout << "current size: " << s.size()
                << endl;
        cout << s.top() << endl;
        s.pop();
        cout << s.top() << endl;
        s.pop();
        cout << s.top() << endl;

        cout << "current size: " << s.size()
                << endl;
        return 0;
}
```

Method 2 (By making pop operation costly)
In push operation, the new element is always enqueued to q1. In pop() operation, if q2 is empty then all the elements except the last, are moved to q2. Finally the last element is dequeued from q1 and returned.

1. **push(s, x)** operation:
    - Enqueue x to q1 (assuming size of q1 is unlimited).
2. **pop(s)** operation:
    - One by one dequeue everything except the last element from q1 and enqueue to q2.
    - Dequeue the last item of q1, the dequeued item is result, store it.
    - Swap the names of q1 and q2
    - Return the item stored in step 2.

```
/* Program to implement a stack
using two queue */
#include <bits/stdc++.h>
using namespace std;

class Stack {
        queue<int> q1, q2;
        int curr_size;

public:
        Stack()
        {
                curr_size = 0;
        }

        void pop()
        {
                if (q1.empty())
                        return;

                // Leave one element in q1 and
                // push others in q2.
                while (q1.size() != 1) {
                        q2.push(q1.front());
                        q1.pop();
                }

                // Pop the only left element
                // from q1
                q1.pop();
```

```
int top()
        {
                if (q1.empty())
                        return -1;

                while (q1.size() != 1) {
                        q2.push(q1.front());
                        q1.pop();
                }

                // last pushed element
                int temp = q1.front();

        // to empty the auxiliary queue after
                // last operation
                q1.pop();

                // push last element to q2
                q2.push(temp);

                // swap the two queues names
                queue<int> q = q1;
                q1 = q2;
                q2 = q;
                return temp;
        }

        int size()
        {
                return curr_size;
```

```
            curr_size--;

            // swap the names of two queues
            queue<int> q = q1;
            q1 = q2;
            q2 = q;
        }

        void push(int x)
        {
            q1.push(x);
            curr_size++;
        }

        s.push(4);

        cout << "current size: " << s.size() << endl;
        cout << s.top() << endl;
        s.pop();
        cout << s.top() << endl;
        s.pop();
        cout << s.top() << endl;
        cout << "current size: " << s.size()<< endl;
        return 0;
}
```

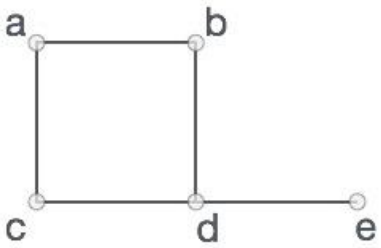
```
        }
};

// Driver code
int main()
{
        Stack s;
        s.push(1);
        s.push(2);
        s.push(3);
```

Lecture #28

Graphs

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph −



In the above graph,

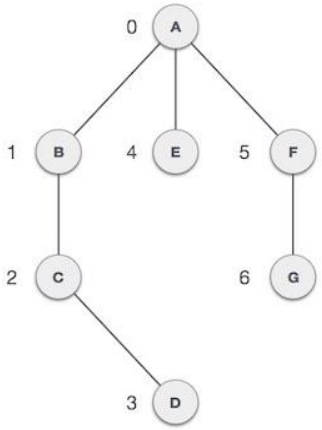
V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

- **Vertex** − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



Basic Operations

Following are basic primary operations of a Graph −

- **Add Vertex** − Adds a vertex to the graph.
- **Add Edge** − Adds an edge between the two vertices of the graph.
- **Display Vertex** − Displays a vertex of the graph.

Graph consists of two following components:
1. Vertices
2. Edges

- Graph is a set of vertices (V) and set of edges (E).
- V is a finite number of vertices also called as nodes.
- E is a set of ordered pair of vertices representing edges.

For example, in Facebook, each person is represented with a vertex or a node. Each node is a structure and contains the information like user id, user name, gender etc.

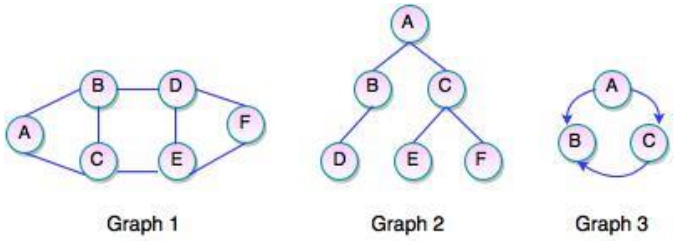


Fig. Graphs

The above figures represent the graphs. The set representation for each of these graphs are as follows:
Graph 1:

V = {A, B, C, D, E, F}
E = {(A, B), (A, C), (B, C), (B, D), (D, E), (D, F), (E, F)}

Graph 2:

V = {A, B, C, D, E, F}
E = {(A, B), (A, C), (B, D), (C, E), (C, F)}

Graph 3:

V = {A, B, C}
E = {(A, B), (A, C), (C, B)}

Directed Graph

- If a graph contains ordered pair of vertices, is said to be a Directed Graph.
- If an edge is represented using a pair of vertices (V_1, V_2), the edge is said to be directed from $V_1$to V_2.
- The first element of the pair V_1 is called the start vertex and the second element of the pair V_2 is called the end vertex.

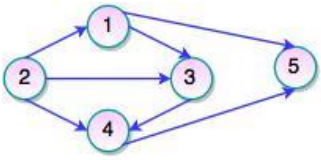


Fig. Directed Graph

Set of Vertices V = {1, 2, 3, 4, 5, 5}
Set of Edges W = {(1, 3), (1, 5), (2, 1), (2, 3), (2, 4), (3, 4), (4, 5)}

Undirected Graph

- If a graph contains unordered pair of vertices, is said to be an Undirected Graph.
- In this graph, pair of vertices represents the same edge.

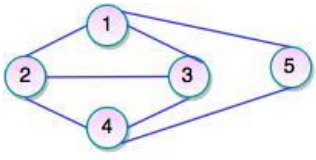


Fig. Undirected Graph

Set of Vertices V = {1, 2, 3, 4, 5}
Set of Edges E = {(1, 2), (1, 3), (1, 5), (2, 1), (2, 3), (2, 4), (3, 4), (4, 5)}

- In an undirected graph, the nodes are connected by undirected arcs.
- It is an edge that has no arrow. Both the ends of an undirected arc are equivalent, there is no head or tail.

Representation of Graphs

Adjacency Matrix

- Adjacency matrix is a way to represent a graph.
- It shows which nodes are adjacent to one another.
- Graph is represented using a square matrix.

Graph can be divided into two categories:

a. Sparse graph contains less number of edges.
b. Dense graph contains number of edges as compared to sparse graph.

- Adjacency matrix is best for dense graph, but for sparse graph, it is not required.
- Adjacency matrix is good solution for dense graph which implies having constant number of vertices.
- Adjacency matrix of an undirected graph is always a symmetric matrix which means an edge (i, j) implies the edge (j, i).

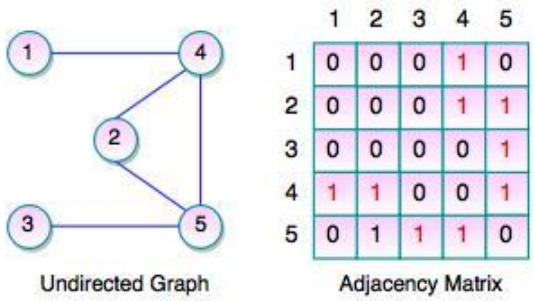
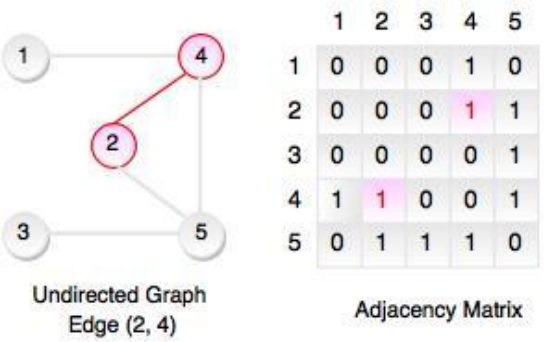


Fig. Adjacency Matrix Representation of Undirected Graph

The above graph represents undirected graph with the adjacency matrix representation. It shows adjacency matrix of undirected graph is symmetric. If there is an edge (2, 4), there is also an edge (4, 2).



Adjacency matrix of a directed graph is never symmetric adj[i][j] = 1, indicated a directed edge from vertex i to vertex j.

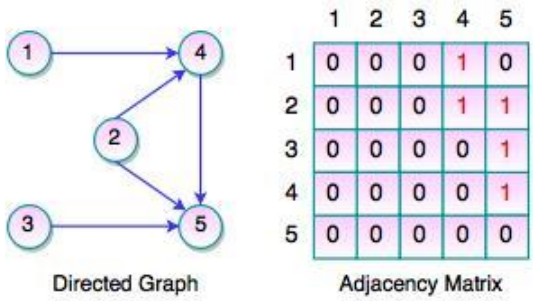


Fig. Adjacency Matrix Representation of Directed Graph

The above graph represents directed graph with the adjacency matrix representation. It shows adjacency matrix of directed graph which is never symmetric. If there is an edge (2, 4), there is not an edge (4, 2). It indicates direct edge from vertex i to vertex j.

Advantages of Adjacency Matrix

- Adjacency matrix representation of graph is very simple to implement.
- Adding or removing time of an edge can be done in O(1) time. Same time is required to check, if there is an edge between two vertices.
- It is very convenient and simple to program.

Disadvantages of Adjacency Matrix

- It consumes huge amount of memory for storing big graphs.
- It requires huge efforts for adding or removing a vertex. If you are constructing a graph in dynamic structure, adjacency matrix is quite slow for big graphs.

Adjacency List

- Adjacency list is another representation of graphs.
- It is a collection of unordered list, used to represent a finite graphs.
- Each list describes the set of neighbors of a vertex in the graph.
- Adjacency list requires less amount of memory.
- For every vertex, adjacency list stores a list of vertices, which are adjacent to the current one.
- In adjacency list, an array of linked list is used. Size of the array is equal to the number of vertices.

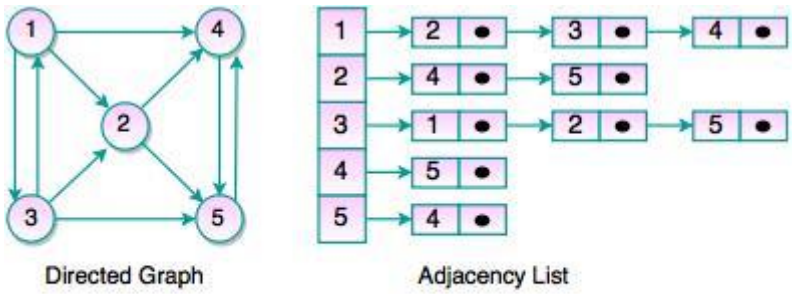


Fig. Adjacency List Representation of Directed Graph

- In adjacency list, an entry array[i] represents the linked list of vertices adjacent to the i^{th} vertex.
- Adjacency list allows to store the graph in more compact form than adjacency matrix.
- It allows to get the list of adjacent vertices in O(1) time.

Disadvantages of Adjacency List

- It is not easy for adding or removing an edge to/from adjacent list.
- It does not allow to make an efficient implementation, if dynamically change of vertices number is required.

Important Note:

Vertex: Each node of the graph is represented as a vertex.
Edge: It represents a path between two vertices or a line between two vertices.
Path: It represents a sequence of edges between the two vertices.
Adjacency: If two nodes or vertices are connected to each other through an edge, it is said to be an adjacency.

Lecture # 29

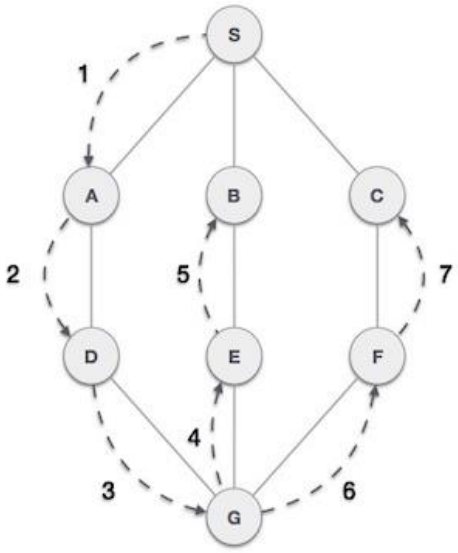
Graph Traversal

- Graph traversal is a process of checking or updating each vertex in a graph.
- It is also known as Graph Search.
- Graph traversal means visiting each and exactly one node.
- Tree traversal is a special case of graph traversal.

There are two techniques used in graph traversal:

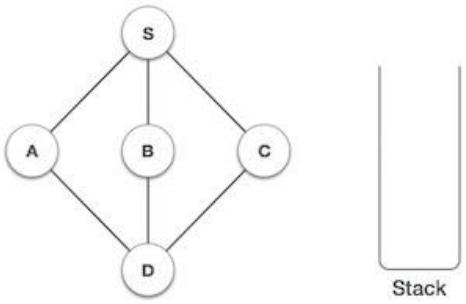
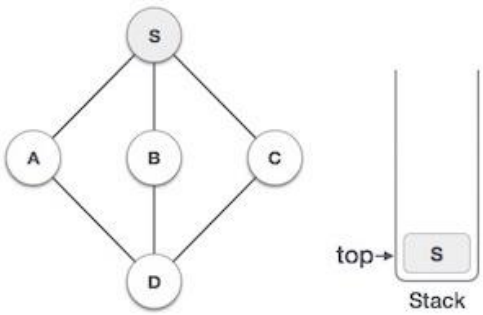
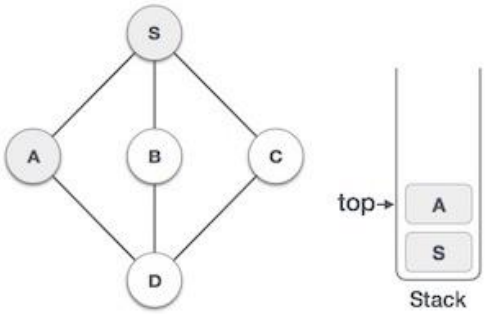
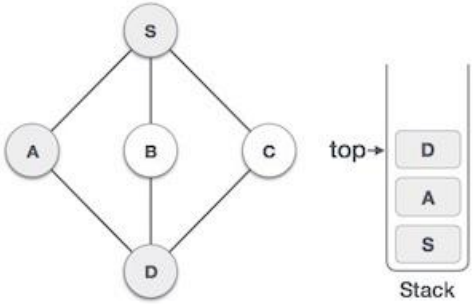
1. Depth First Search
2. Breadth First Search

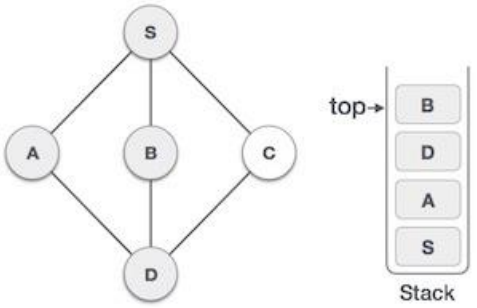
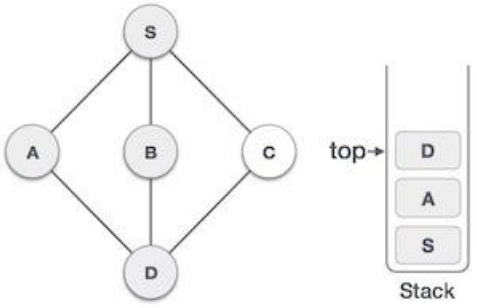
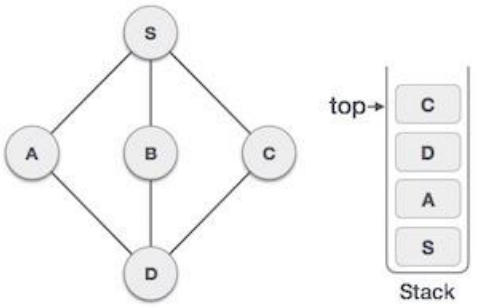
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

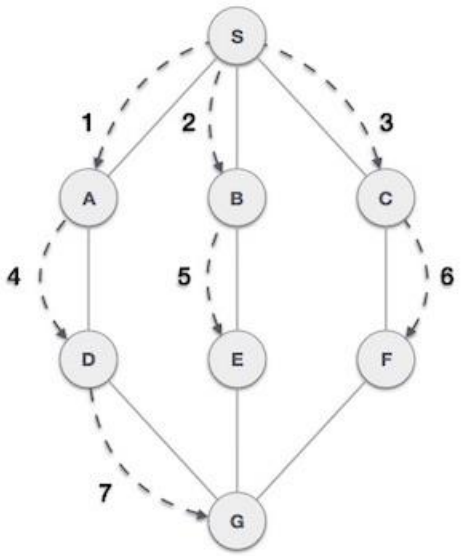
Step	Traversal	Description
1		Initialize the stack.
2		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.

5		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.
6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.
7		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

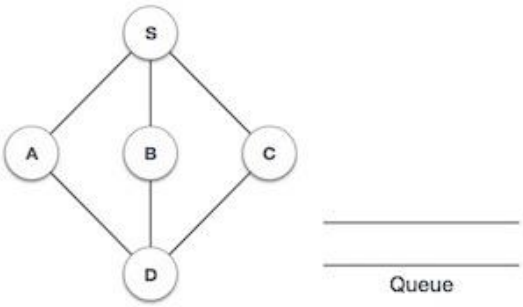
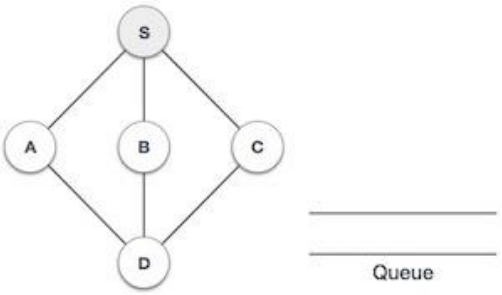
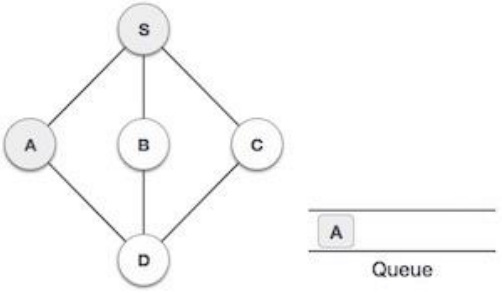
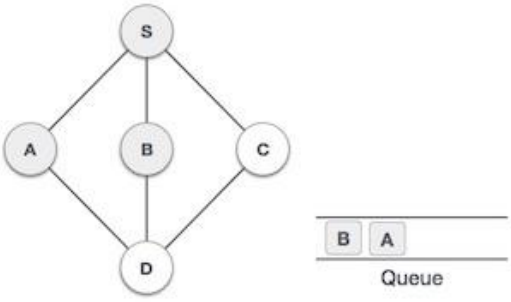
Breadth-First Search

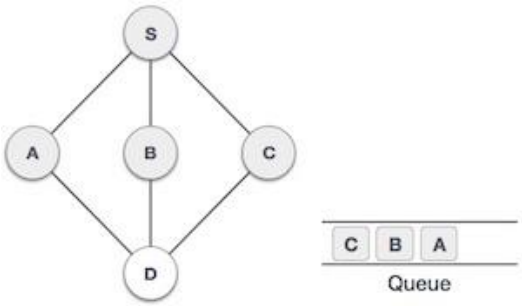
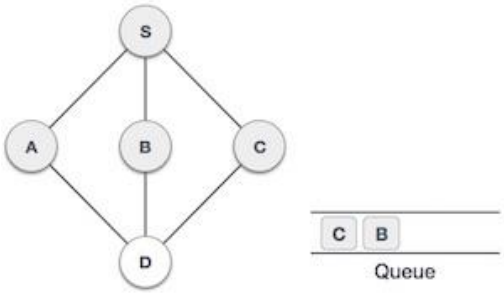
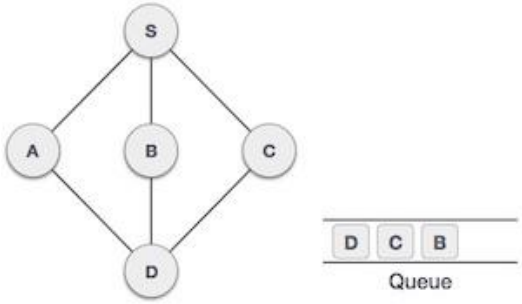
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting **S**(starting node), and mark it as visited.
3		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.

5		Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.
6		Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.
7		From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Breadth-first search: a method to search a graph and an exemplar of many important graph algorithms. Given G = (V, E), chose a vertex, s, to be the *source*. Move out from s, systematically to find all v Î V that are accessible from s, and compute their distances from s. Distance from s to v is the "shortest path".

Words: start at s, find all vertices distance 1 from s, keep going until all vertices have been discovered.

Dye analogy: s is the dye injection point and you watch the progress of the dye through the graph. Each time step a single edge is crossed. In breadth-first search, each vertex is colored:

a. White: not yet discovered

b. Gray: discovered but still some undiscovered adjacent vertices .

c. Black: discovered and all adjacent vertices discovered

Gray vertices are the frontier/interface between discovered/undiscovered.

Constructing a breadth-first tree:

1. Start with s (the root)

2. Scan the adjacency list of all previously discovered nodes. If u was previously discovered and we just found v, then we add v. u is the predecessor/parent of v in the breadth-first tree.

This progresses, iteratively. Note: Start with all nodes white. When you first encounter a white node, you make it gray. When you go through that node's adjacency list, you make it black.

Code: Store

a. Color (white or gray or black)

b. Predecessor p[n]

c. Distance from s d[n]

BFS (G, s)

```
1.  for each vertex u Î V[G] - {s}
2.     do color [u] ¬ white  // white out all but s
3.       d[u] ¬ ¥
4.       p[u] ¬ nil
5.  color [s] ¬ gray // s starts out gray
6.  d[s] ¬ 0
7.  p[s] ¬ nil
8.  Q ¬ {s} // end initialization
9.  while Q ¹ Æ
10.   do u ¬ head[Q]
11.      for each v Adj[u]
12.         do if color[v] = white
13.              then color[v] ¬ gray
14.                 d[v] ¬ d[u] + 1
15.                  p[v] ¬ u
16.                 Enqueue ( Q, v )
17.       Dequeue ( Q ) // Q stores grays
18.       color[u] ¬ black // all of u's adjacents have been seen, so color it black and //
    move on
```

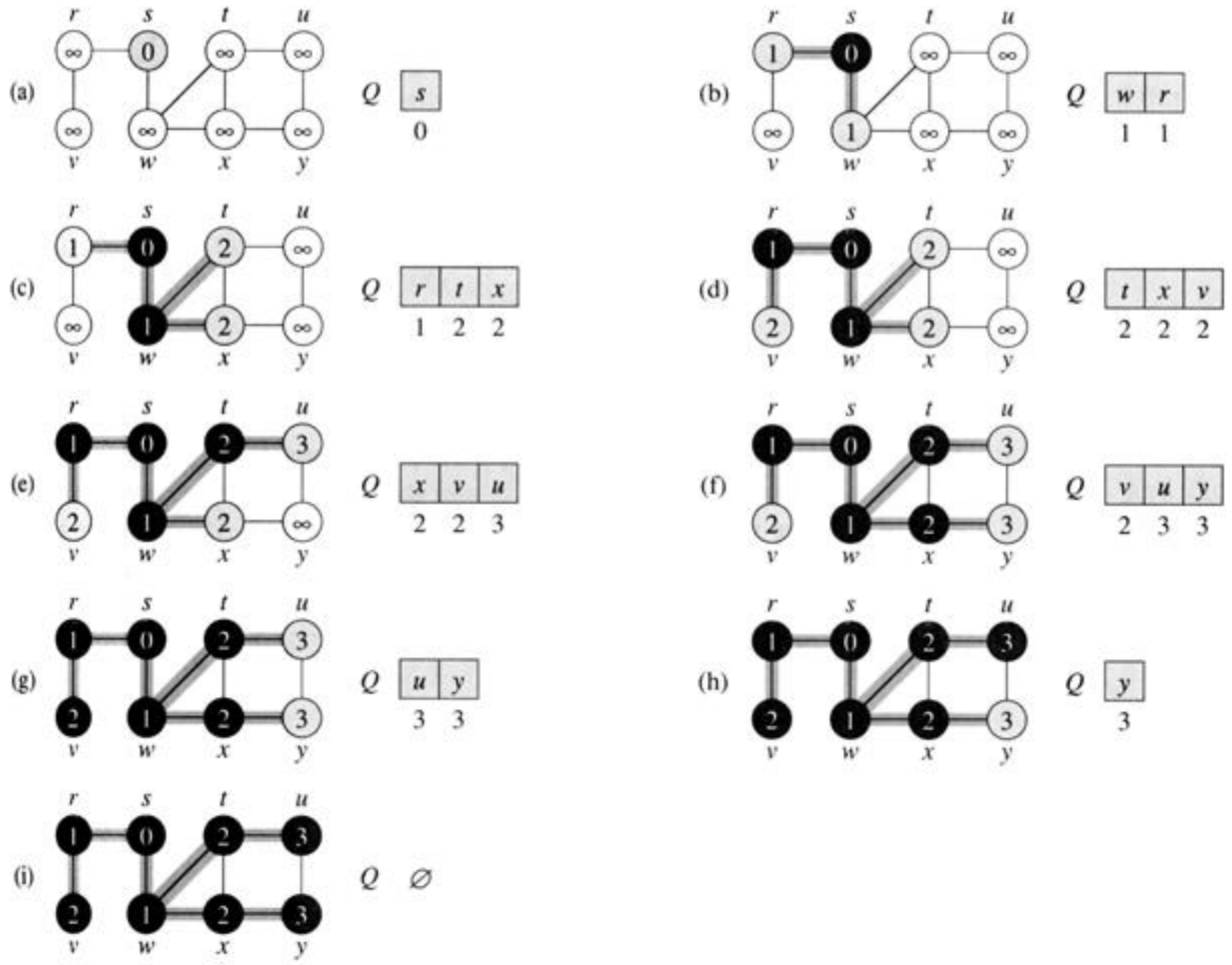


Figure 23.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 9–18. Vertex distances are shown next to vertices in the queue.

Running time of Breadth-First search:

1. Each node is enqueued once (white ® gray)
2. Each node is dequeued once (gray ® black)
3. Each adjacency list is scanned only once.

1. **O** (| V |)
2. **O** (| V |)
3. **O** (| E |)

Overall running time is **O** (| V | + | E |). Clearly better to use the adjacency list representation. (What differs with the matrix representation?)

Shortest-path distance: d (s, v) is the minimum number of edges in any path from s to v, or ¥ if there is no path. A path from s to v with distance d (s, v) is a shortest-path. Note, there may be more than one shortest path.

Lecture #30

Warshall's algorithm

Warshall's algorithm determines whether there is a path between any two nodes in the graph. It does not give the number of the paths between two nodes.

Idea: Compute all paths containing node 1, then all paths containing nodes 1 or 2 or 1 and 2, and so on, until we compute all paths with intermediate nodes selected from the set {1, 2, … n}.

Here we compute a sequence of matrices $P^{(0)}$, $P^{(1)}$,…, $P^{(n)}$ such that $P^{(0)} = A$, $P^{(n)} = P$, the path matrix, and $P^{(r)}$ shows all paths with intermediate nodes selected from the set of nodes {1, 2, 3, …, r}.

The algorithm can be defined recursively:

Let $P^{(0)} = A$.
Let $P^{(r)}$ is a matrix such that:

$p_{ik}^{(r)} = 1$ if and only if there is a path connecting nodes **i** and **k** through one or more of nodes {1, 2, …, r}

$p_{ik}^{(r)} = 0$ otherwise

Let $P^{(r+1)}$ is the matrix where

$p_{ik}^{(r+1)} = 1$ if and only if there is a path connecting nodes **i** and **k** through one or more of nodes {1, 2, …, r, r+1}

$p_{ik}^{(r+1)} = 0$ otherwise

$P^{(r+1)}$ can be obtained from $P^{(r)}$ in the following way:

If $p_{ik}^{(r)} = 1$ then $p_{ik}^{(r+1)} := 1$

Else $p_{ik}^{(r+1)} := p_{i\,r+1}^{(r)} * p_{r+1\,k}^{(r)}$

As a result we have:

$p_{ik}^{(r+1)} = 1$ if and only if there is a path connecting nodes **i** and **k** and containing intermediate nodes selected from the set {1, 2, 3, .. r+1}

$p_{ik}^{(r+1)} = 0$ otherwise.

Correctness of the algorithm

a) Suppose $p_{ik}^{(r+1)} = 1$. This is possible only if

1. $p_{ik}^{(r)} = 1$, which by the definition of $P^{(r)}$ means that there is a path between nodes **i** and **k**, with intermediate nodes selected from {1, 2, … r}, or:

2. Both $p_{i,r+1}^{(r)}$ and $p_{r+1,k}^{(r)}$ are equal to 1.

   $p_{i,r+1}^{(r)} = 1$ means that there is a path from node **i** to node **r+1** containing intermediate nodes selected from the set {1, 2, 3, .. r}

   $p_{r+1,k}^{(r)} = 1$ means that there is a path from node **r+1** to node **k** containing intermediate nodes selected from the set {1, 2, 3, .. r}

   Thus, there is a path from node **i** to node **r+1** and from node **r+1** to node **k**. Hence there is a path from **i** to **k** through **r+1**, i.e. its intermediate nodes are selected from the set {1, 2, 3, .. r+1}

   Therefore, if $p_{i,k}^{(r+1)} = 1$ then there is a path from **i** to **k** with its intermediate nodes selected from the set {1, 2, 3, .. r+1}

b) Suppose now that there is a path from node **i** to node **k** with its intermediate nodes selected from the set {1, 2, 3, .. r+1}

Consider two cases

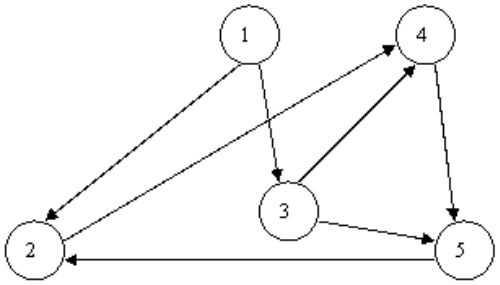
3. r+1 is not in the path connecting **i** and **k**. Then the path is selected from {1,2, .. , r} by the definition of $P^{(r)}$.
   $p_{ik}^{(r)} = 1$ (see (b) above), hence $p_{ik}^{(r+1)} = 1$

4. r+1 is in the path connecting **i**> and **k**. Then there is a path connecting **i** and **r+1**, and a path connecting **r+1** and **k**. Hence $p_{i,r+1}^{(r)}$ and $p_{r+1,k}^{(r)}$ are equal to 1.
   Therefore, $p_{i,r+1}^{(r)} * p_{r+1,k}^{(r)} = 1$
   Therefore $p_{ik}^{(r+1)} = 1$.

Warshall's algorithm requires $O(n^3)$ operations:

$O(n^2)$ to obtain each $P^{(r)}$ and the calculations are done for r = 1, 2, .., n

2. Example



Adjacency matrix A:

	1	**2**	**3**	**4**	**5**
1	0	1	1	0	0
2	0	0	0	1	0
3	0	0	0	1	1
4	0	0	0	0	1
5	0	1	0	0	0

$P^{(0)}$ = A, i.e. this is the matrix showing paths with no intermediate nodes. $P^{(1)}$ is the matrix containing all paths from $P^{(0)}$ plus all paths with 1 as intermediate node.

Its elements are computed in the following way:

$p_{ij}^{(1)} = 1$ if $p_{ij}^{(0)} = 1$

Else $p_{ij}^{(1)} = p_{i1}^{(0)} * p_{1j}^{(0)}$

Thus for all elements that are 0 in $P^{(0)}$, we compute the products of the elements in the first column and the first row of $P^{(0)}$. Since all elements in the first column are 0, the products will be 0, so $P^{(1)}$ will be the same as $P^{(0)}$.

$P^{(1)}$ =

	1	**2**	**3**	**4**	**5**
1	0	1	1	0	0
2	0	0	0	1	0
3	0	0	0	1	1
4	0	0	0	0	1
5	0	1	0	0	0

$P^{(2)}$ will contain all paths computed in $P^{(1)}$ plus all paths that contain 2 as an intermediate node

$p_{ij}^{(2)} = 1$ if $p_{ij}^{(1)} = 1$

Else $p_{ij}^{(2)} = p_{i2}^{(1)} * p_{2j}^{(1)}$

Here we compute the products of the second column and the second row. The second column contains two non-zero elements, and the second row contains one non-zero element, thus the non-zero products will be:

$p_{14}^{(2)} = p_{12}^{(1)} * p_{24}^{(1)}$, $p_{54}^{(2)} = p_{52}^{(1)} * p_{24}^{(1)}$

$P^{(2)}$ =

	1	**2**	**3**	**4**	**5**
1	0	1	1	**1**	0

2	0	0	0	1	0
3	0	0	0	1	1
4	0	0	0	0	1
5	0	1	0	**1**	0

The new paths are the paths connecting nodes 1 and 4 through node 2, and nodes 5 and 4 through node 2.

Next we compute P(3), looking for paths that have intermediate nodes among {1, 2, 3}.

$p_{ij}^{(3)} = 1$ if $p_{ij}^{(2)} = 1$

Else $p_{ij}^{(3)} = p_{i3}^{(2)} * p_{3j}^{(2)}$

Here we look at the products of the elements in the third column and the third row. The non-zero products are:

$p_{14}^{(3)} = p_{13}^{(2)} * p_{34}^{(2)}$, $p_{15}^{(3)} = p_{13}^{(2)} * p_{35}^{(2)}$

Note that p14(2) = 1, which means that we have already found a path between nodes 1 and 2. The new path computed here is the path connecting nodes 1 and 5 through node 3.

$P^{(3)}$ =

	1	**2**	**3**	**4**	**5**
1	0	1	1	1	**1**
2	0	0	0	1	0
3	0	0	0	1	1
4	0	0	0	0	1
5	0	1	0	1	0

Next we compute P(4), looking for paths that have intermediate nodes among {1, 2, 3, 4}.

$p_{ij}^{(4)} = 1$ if $p_{ij}^{(3)} = 1$

Else $p_{ij}^{(4)} = p_{i4}^{(3)} * p_{4j}^{(3)}$

Here we find the products of the elements in the fourth column and fourth row. The non-zero products are:

$p_{15}^{(4)} = p_{14}^{(3)} * p_{45}^{(3)}$
$p_{25}^{(4)} = p_{24}^{(3)} * p_{45}^{(3)}$
$p_{35}^{(4)} = p_{34}^{(3)} * p_{45}^{(3)}$
$p_{55}^{(4)} = p_{54}^{(3)} * p_{45}^{(3)}$

Only $p_{25}^{(4)}$ and $p_{55}^{(4)}$ give new paths, connecting nodes 2 and 5 through node 4, and node 5 to itself through node 4. Thus we have

$P^{(4)}$ =

	1	**2**	**3**	**4**	**5**
1	0	1	1	1	1
2	0	0	0	1	**1**
3	0	0	0	1	1
4	0	0	0	0	1
5	0	1	0	1	**1**

Finally, we compute $P^{(5)}$

$p_{ij}^{(5)} = 1$ if $p_{ij}^{(4)} = 1$

Else $p_{ij}^{(5)} = p_{i5}^{(4)} * p_{5j}^{(4)}$

The fifth column contains five non-zero elements, and the fifth row contains three non-zero elements, hence fifteen products will be not equal to zero. The new paths are given by the elements:

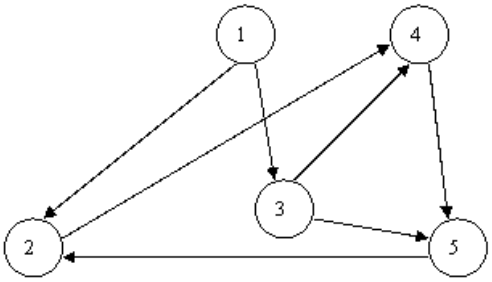
$p_{22}^{(5)}$ - connecting 2 to itself through node 5
$p_{32}^{(5)}$ - connecting 3 and 2 through node 5

$p_{42}^{(5)}$ - connecting 4 and 2 through node 5
$p_{44}^{(5)}$ - connecting 4 to itself through node 5
$P^{(5)}$ =

	1	**2**	**3**	**4**	**5**
1	0	1	1	1	1
2	0	**1**	0	1	1
3	0	**1**	0	1	1
4	0	**1**	0	**1**	1
5	0	1	0	1	1

The matrix shows that no node is connected to node 1. Except node 1, no other node is connected to node 3. Node 1 is connected to all nodes except to itself. All nodes are connected to 2, 4, and 5.

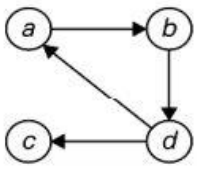
You can find the paths in the graph:



ALGORITHM *Warshall*$(A[1..n, 1..n])$

```
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
R^(0) ← A
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            R^(k)[i, j] ← R^(k−1)[i, j] or (R^(k−1)[i, k] and R^(k−1)[k, j])
return R^(n)
```

Time efficiency: $\Theta(n^3)$



$R^{(0)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

Ones reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$R^{(1)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	**1**	1	0

Ones reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$R^{(2)}$ =

	a	b	c	d
a	0	1	0	**1**
b	0	0	0	1
c	0	0	0	0
d	1	1	1	**1**

Ones reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$R^{(3)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

Ones reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a, b,* and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$R^{(4)}$ =

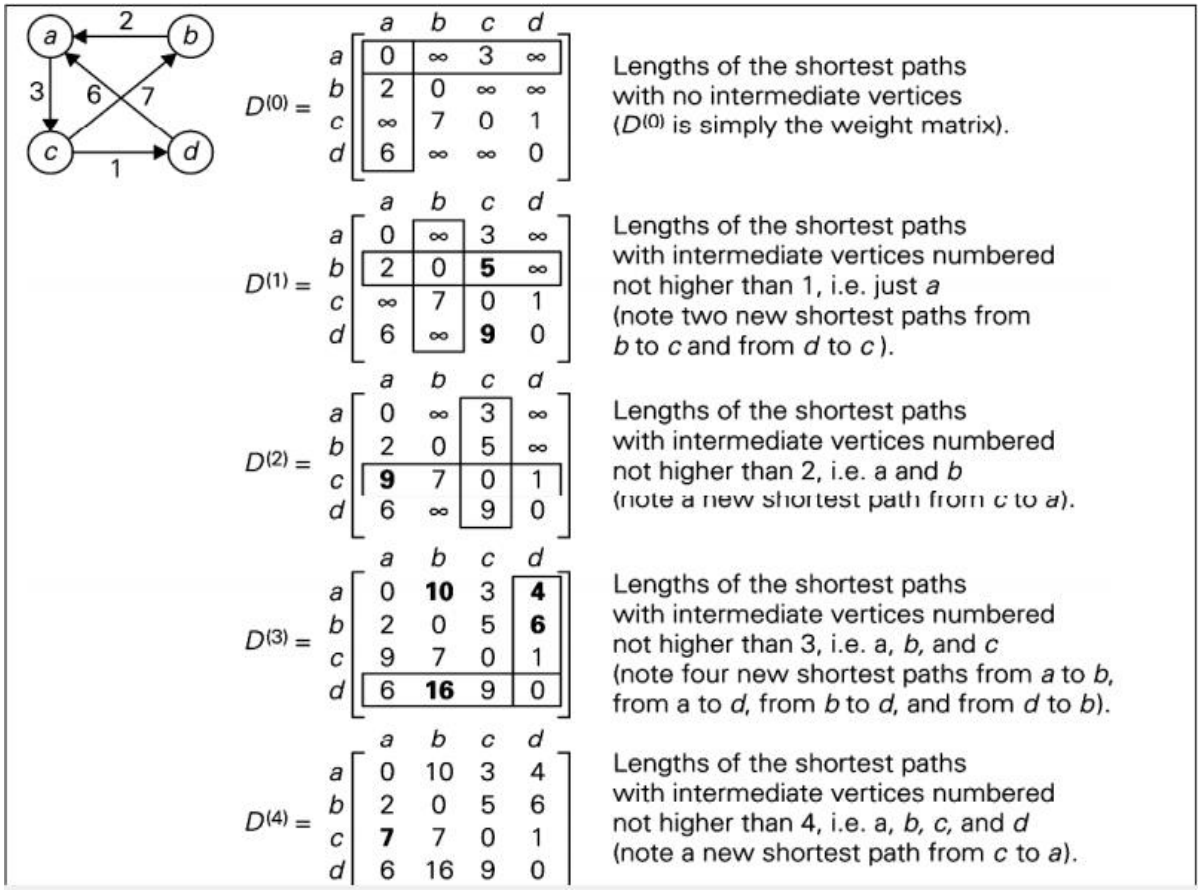
	a	b	c	d
a	**1**	1	**1**	1
b	**1**	**1**	**1**	1
c	0	0	0	0
d	1	1	1	1

Ones reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a, b, c,* and *d* (note five new paths).

ALGORITHM *Floyd*$(W[1..n, 1..n])$

```
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
D ← W //is not necessary if W can be overwritten
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D
```

Time efficiency: $\Theta(n^3)$



Lecture #37 and 38

Heap

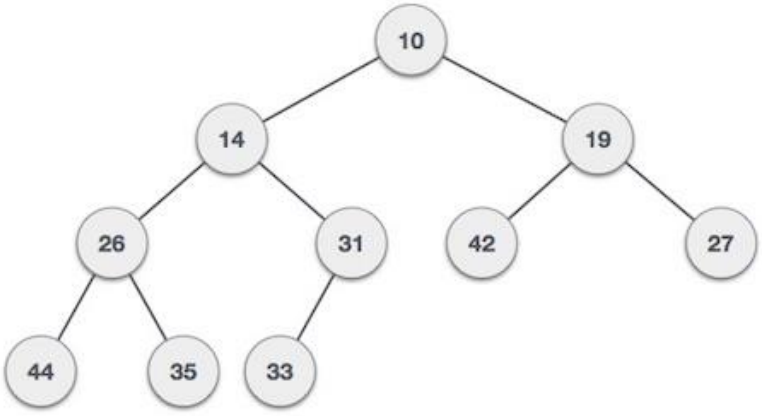
Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If **α** has child node **β** then −

$$\text{key}(\alpha) \geq \text{key}(\beta)$$

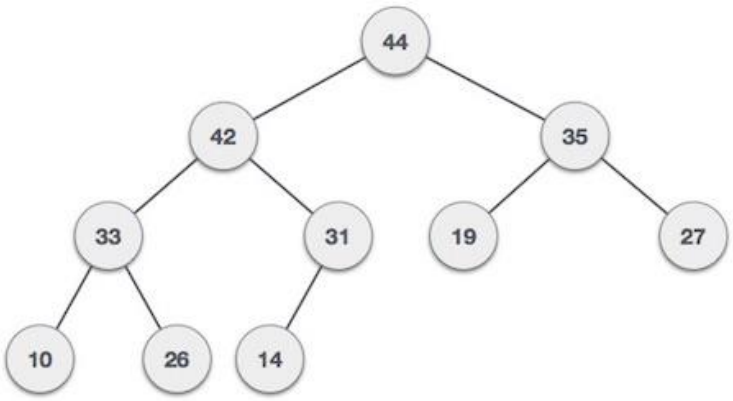
As the value of parent is greater than that of child, this property generates **Max Heap**. Based on this criteria, a heap can be of two types −

For Input → 35 33 42 10 14 19 27 44 26 31

Min-Heap − Where the value of the root node is less than or equal to either of its children.



Max-Heap − Where the value of the root node is greater than or equal to either of its children.



Both trees are constructed using the same input and order of arrival.

Max Heap Construction Algorithm

We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

Step 1 − Create a new node at the end of heap.
Step 2 − Assign new value to the node.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.

Note − In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

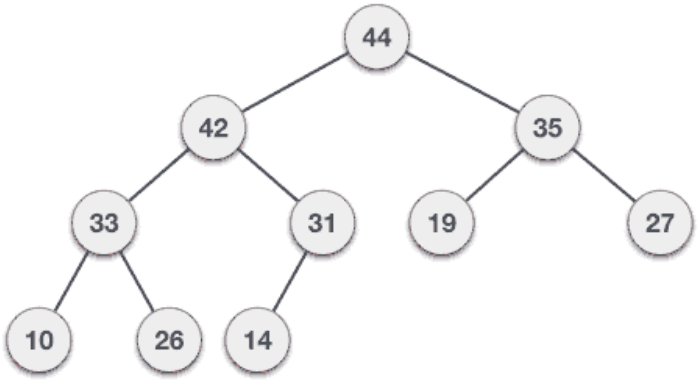
Let's understand Max Heap construction by an animated illustration. We consider the same input sample that we used earlier.

Input 35 33 42 10 14 19 27 44 26 31

Max Heap Deletion Algorithm

Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

Step 1 − Remove root node.
Step 2 − Move the last element of last level to root.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.



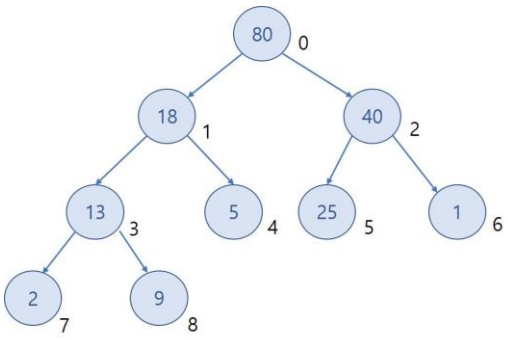
Heap Data Structure Properties

Below are some of the features of the heap data structures

- Unlike binary search trees, the left child of a node does not need to be less than the key of the parent. So naturally, the right child of a node does not have to be greater than that of the parent.
- For each level of the tree, the data is **inserted from left to right**.
- A new level of the tree is NOT started until the current level is filled from left to right.

Let me use the example below to clarify.As you can see, unlike in the binary search tree, the left Child is not necessarily lesser than the right child. As long as the parent is greater than its children, it a valid heap.If we were to insert another node into the heap below, it will be inserted as a left child of node with key of 5.

The tree currently has a depth of four. Level four will continue to be filled and a new level will not be started until the node with key value 1 (index 6) has both a left and a right child.



Maximum Heap Properties

Let's take a look at the properties that define the max heap.

- The key of parent nodes are **greater than or equal to** (>=) that of the children.
- Therefore, the highest key is at the root node.

If this is not sticking, allow me to demonstrate with a picture. Hopefully this will make everything clearer.

Minimum Heap Properties

To put things into perspective, the min heap is the inverted result of the max heap.

Just to be thorough, let us go through each of the key properties of the min heap.

- The key of the parent nodes are less than or equal to (<=) that of the children.
- Therefore, lowest key is the root node.

Heap Implementation Using Arrays

As stated before, the heap can be implemented via two distinct methods. The first is to implement the heap using an array. The other method is the one that you should be familiar with if you have been following my blog: using nodes.

Implementing the Heap using Arrays

Let's take a look at implementing the heap using an array.

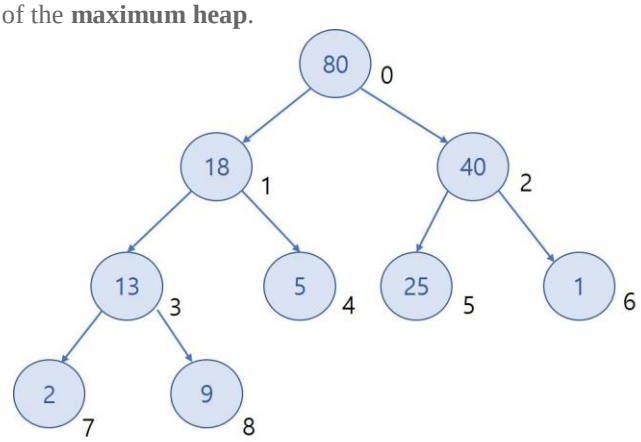
- When inserting data to the heap, after inserting, we need to check whether the heap properties are met.

- If we are working with a minimum heap, we need to ensure that the key of the parent node is greater than the current node.

- Conversely, if we are implementing the maximum heap, we need to make sure that the key of the parent node is lesser than the current node.

If any of the conditions are violated, we need to perform swaps and check recursively to ensure that the heap properties are being upheld throughout the entire tree.

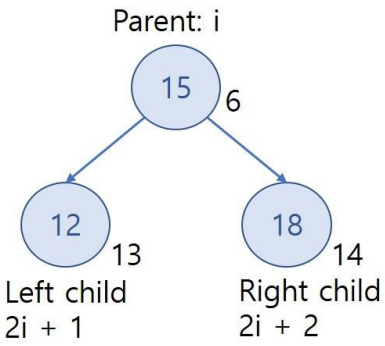
Below is a sample image of the **maximum heap**.



If we look at the underlying array housing the heap displayed above, it will look like this

[80, 18, 40, 13, 5, 25, 1, 2, 9]

When we are building the heap using arrays as the underlying data structure, we need to be aware of the following formula for calculating the index of items in the tree.



In the heap example above, we have the following array.

[80, 18, 40, 13, 5, 25, 1, 2, 9]
Let's take a look at

[80, 18, 40]
The index of 8 is 0. Therefore, i in this case equals zero. No problem understanding this right?
If i is zero, the index of 18 is

```
(2 * 0) + 1

0 + 1

1
```

Working with 40, its index will be

```
(2 * 0) + 2

0 + 2

2
```

The calculations above are simple examples. Before moving on, I would like to issue a challenge to you.

This will help solidify your understanding of how to implement the heap using arrays as the underlying data structure.

Exercise

Try working with all the items inside of the array and see if this property holds. Afterwards, try constructing your own heap using arrays with different values, and see if the property holds.

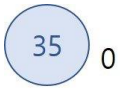
Inserting Data into the Maximum Heap

I am going to break down the insertion process into steps.

For demonstration purposes, I will also provide an image of the data structure after each insertion operation.

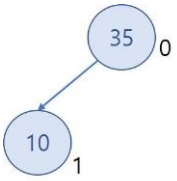
Step 1: Insert 35.

Array: [35]



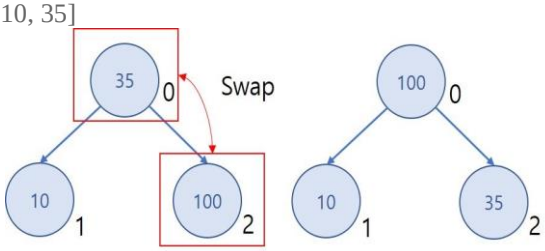
Step 2: Insert 10.

Array: [35, 10]



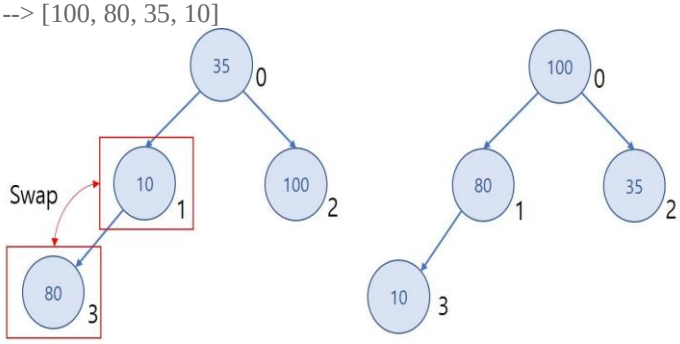
Step 3: Insert 100. Max heap rule is violated: Parent (35) is less than right child (100). Swap 35 and 100.

Array: [35, 10, 100] --> [100, 10, 35]



Step 4: Insert 83. Max heap rule is violated. Parent (10) is less than left child (83) Swap 10 and 83.

Array: [100, 10, 35, 80] --> [100, 80, 35, 10]



Lecture #39

Heap Application:

The Heap Data Structure has NOTHING to do with the Heap Memory
I just wanted to address this common misconceptions to those that read all the way up to this section.

Please, erase this misconception from your mind. The heap data structure and the heap memory in your computer are NOT related.

The heap data structure is not used in the heap memory implementation details.

Parting words
In the next post in this series, we will be discussing the implementation details. If this post helped you out, please share the goodies.

By now, you should have enough knowledge to implement a basic heap data structure in your favorite language.

In the future, I will upload the source code of the heap data structure implementation JavaScript, Java and C++.

Hope that this read was informative. In the near future (probably next week), I will be coming back to this post to add more information and details surrounding the heap data structure, namely more information on deleting data from the heap.

Heap Data Structure is generally taught with Heapsort. Heapsort algorithm has limited uses because Quicksort is better in practice. Nevertheless, the Heap data structure itself is enormously used. 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. Binomoial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also in O(logn) time which is a O(n) operation in Binary Heap. Heap Implemented priority queues are used in Graph algorithms like Prim's Algorithmand Dijkstra's algorithm.
Order statistics: The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array. See method 4 and 6 of this post for details.

Heap Sort

Heaps can be used in sorting an array. In max-heaps, maximum element will always be at the root. Heap Sort uses this property of heap to sort the array.

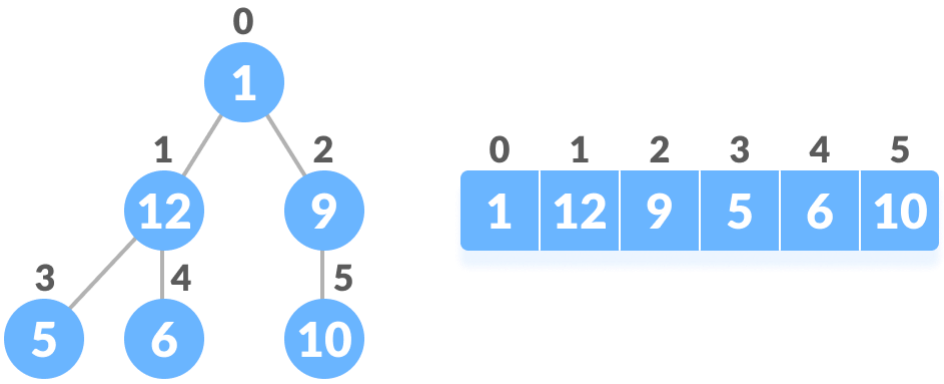
Consider an array Arr which is to be sorted using Heap Sort.

- Initially build a max heap of elements in Arr.
- The root element, that is Arr[1], will contain maximum element of Arr. After that, swap this element with the last element of Arr and heapify the max heap excluding the last element which is already in its correct position and then decrease the length of heap by one.
- Repeat the step 2, until all the elements are in their correct position.

Relationship between Array Indexes and Tree Elements

A complete binary tree has an interesting property that we can use to find the children and parents of any node.

If the index of any element in the array is *i*, the element in the index `2i+1` will become the left child and element in `2i+2` index will become the right child. Also, the parent of any element at index *i* is given by the lower bound of `(i-1)/2`.



Relationship between array and heap indices

Let's test it out,

```
Left child of 1 (index 0)
= element in (2*0+1) index
= element in 1 index
= 12


Right child of 1
= element in (2*0+2) index
= element in 2 index
= 9

Similarly,
Left child of 12 (index 1)
= element in (2*1+1) index
= element in 3 index
= 5

Right child of 12
= element in (2*1+2) index
= element in 4 index
= 6
```

Let us also confirm that the rules hold for finding parent of any node

```
Parent of 9 (position 2)
= (2-1)/2
= ½
= 0.5
~ 0 index
= 1

Parent of 12 (position 1)
= (1-1)/2
= 0 index
= 1
```

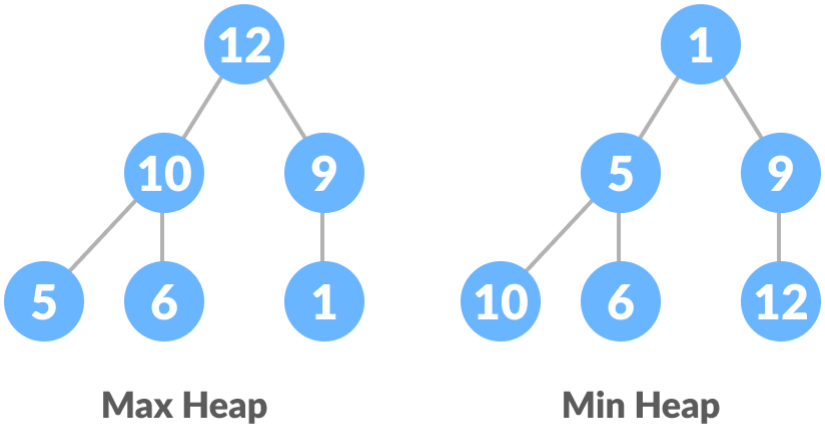
Understanding this mapping of array indexes to tree positions is critical to understanding how the Heap Data Structure works and how it is used to implement Heap Sort.

What is Heap Data Structure?

Heap is a special tree-based data structure. A binary tree is said to follow a heap data structure if

- it is a complete binary tree
- All nodes in the tree follow the property that they are greater than their children i.e. the largest element is at the root and both its children and smaller than the root and so on. Such a heap is called a max-heap. If instead, all nodes are smaller than their children, it is called a min-heap

The following example diagram shows Max-Heap and Min-Heap.



Max Heap and Min Heap

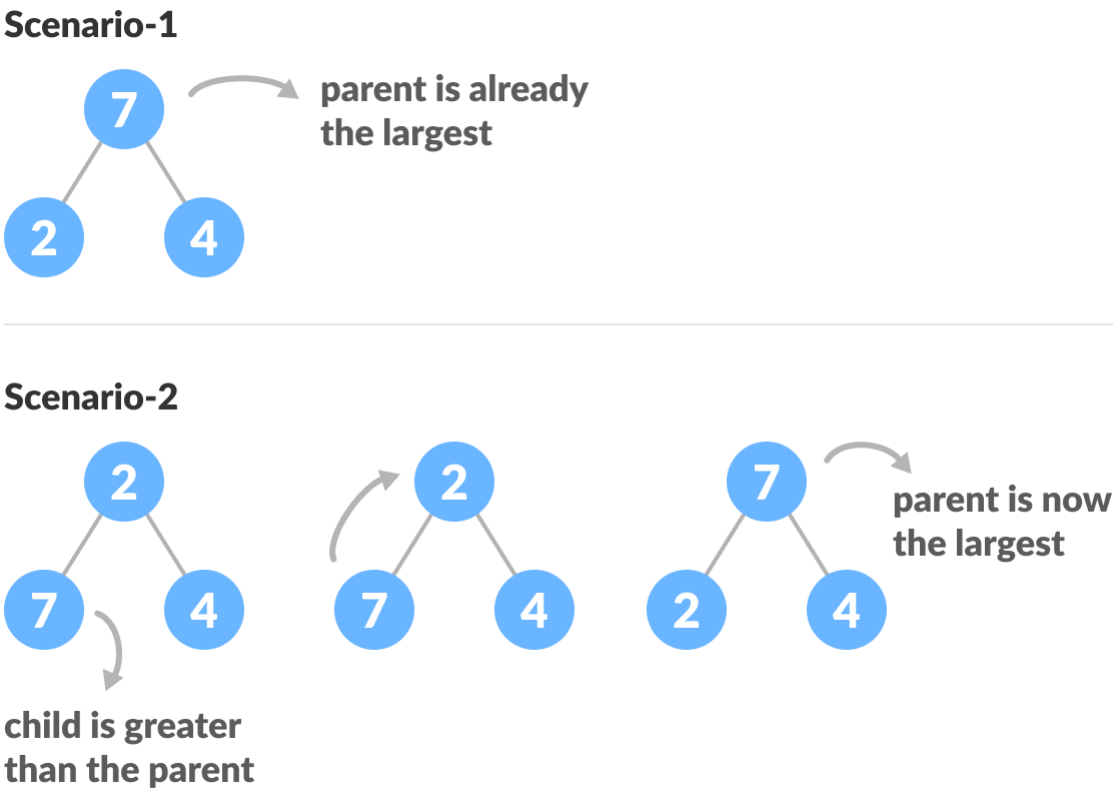
To learn more about it, please visit Heap Data Structure.

How to "heapify" a tree

Starting from a complete binary tree, we can modify it to become a Max-Heap by running a function called heapify on all the non-leaf elements of the heap.

Since heapify uses recursion, it can be difficult to grasp. So let's first think about how you would heapify a tree with just three elements.

```
heapify(array)
    Root = array[0]
    Largest = largest( array[0] , array [2*0 + 1]. array[2*0+2])
    if(Root != Largest)
          Swap(Root, Largest)
```

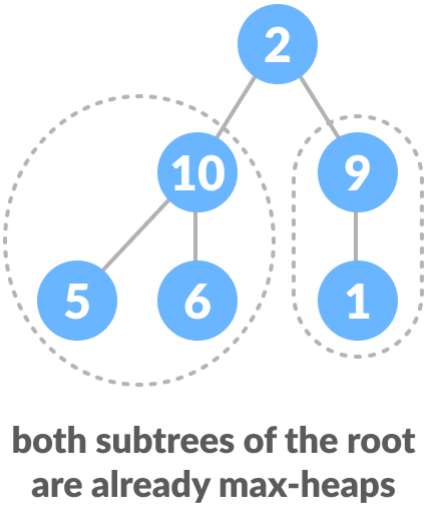


Heapify base cases

The example above shows two scenarios - one in which the root is the largest element and we don't need to do anything. And another in which the root had a larger element as a child and we needed to swap to maintain max-heap property.

If you're worked with recursive algorithms before, you've probably identified that this must be the base case.

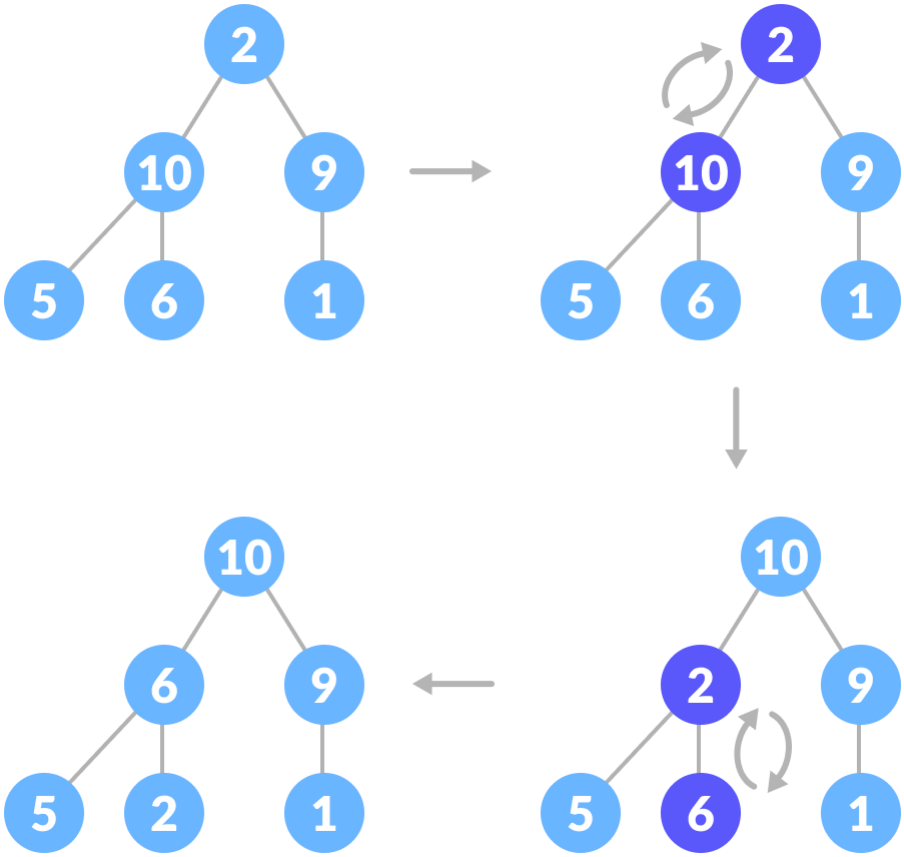
Now let's think of another scenario in which there is more than one level.



How to heapify root element when its subtrees are already max heaps

The top element isn't a max-heap but all the sub-trees are max-heaps.

To maintain the max-heap property for the entire tree, we will have to keep pushing 2 downwards until it reaches its correct position.



How to heapify root element when its subtrees are max-heaps

Thus, to maintain the max-heap property in a tree where both sub-trees are max-heaps, we need to run heapify on the root element repeatedly until it is larger than its children or it becomes a leaf node.

We can combine both these conditions in one heapify function as

```
void heapify(int arr[], int n, int i) {
  // Find largest among root, left child and right child
  int largest = i;
  int left = 2 * i + 1;
  int right = 2 * i + 2;

  if (left < n && arr[left] > arr[largest])
    largest = left;

  if (right < n && arr[right] > arr[largest])
    largest = right;

    // Swap and continue heapifying if root is not largest
    if (largest != i) {
      swap(&arr[i], &arr[largest]);
      heapify(arr, n, largest);
  }
}
```

This function works for both the base case and for a tree of any size. We can thus move the root element to the correct position to maintain the max-heap status for any tree size as long as the sub-trees are max-heaps.

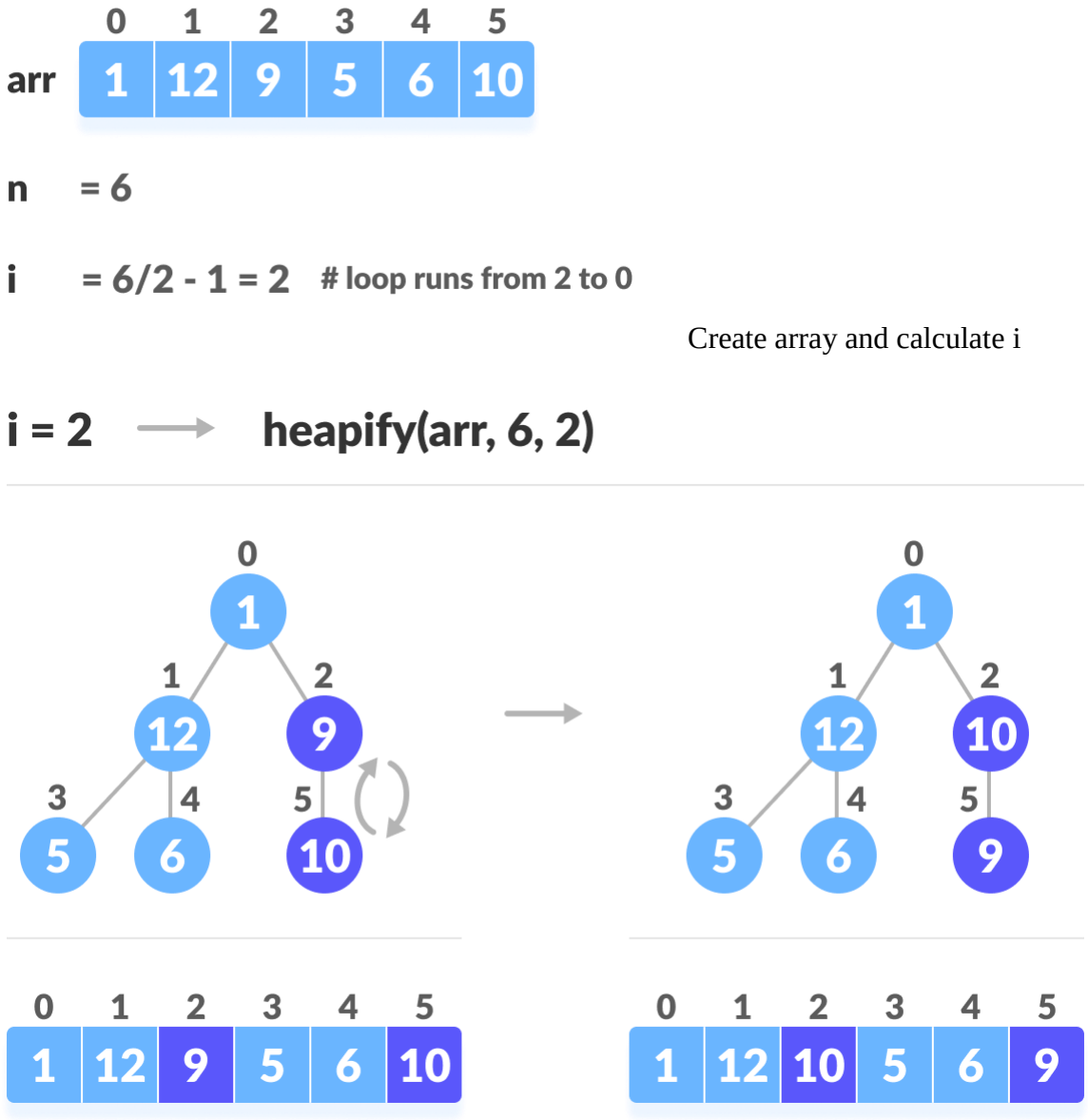
Build max-heap

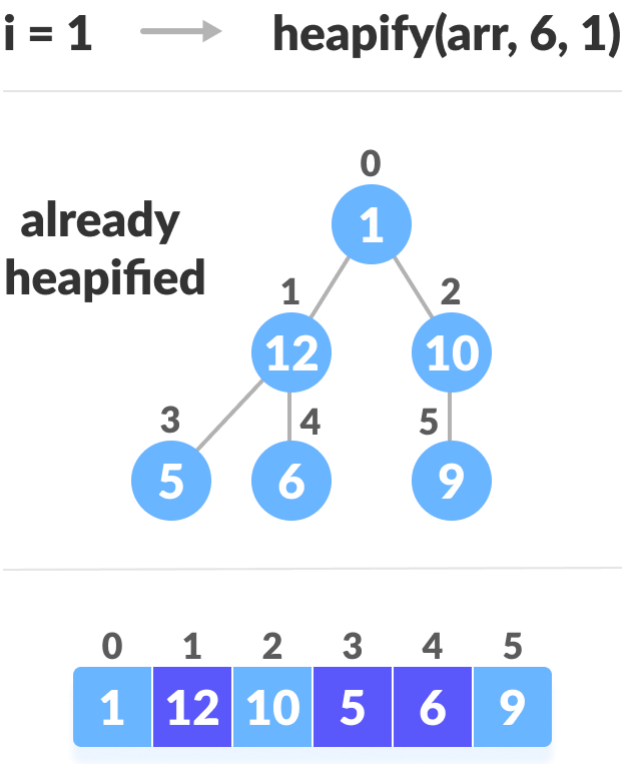
To build a max-heap from any tree, we can thus start heapifying each sub-tree from the bottom up and end up with a max-heap after the function is applied to all the elements including the root element.

In the case of a complete tree, the first index of a non-leaf node is given by `n/2 - 1`. All other nodes after that are leaf-nodes and thus don't need to be heapified.

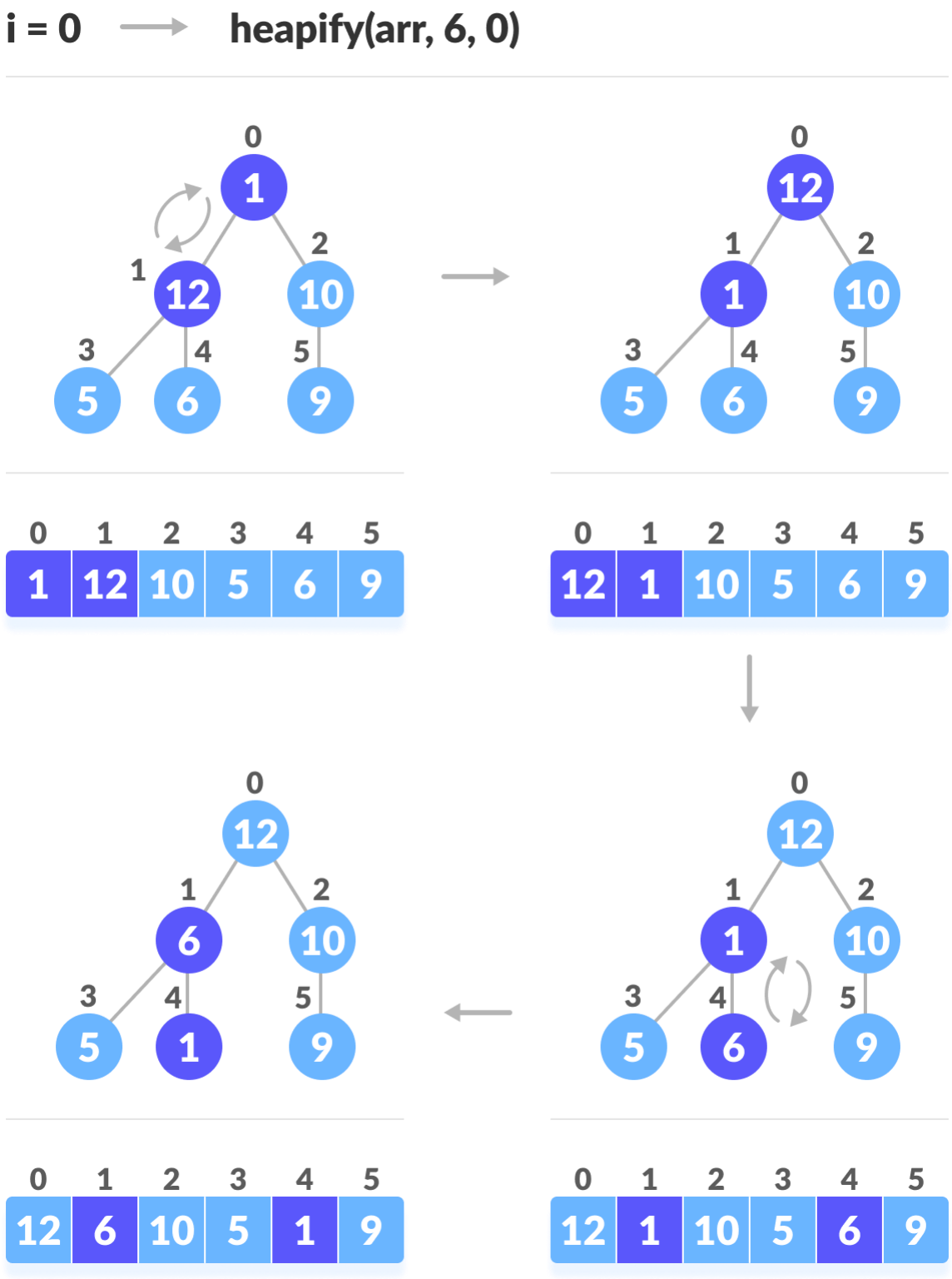
So, we can build a maximum heap as

```
// Build heap (rearrange array)
for (int i = n / 2 - 1; i >= 0; i--)
  heapify(arr, n, i);
```





Steps to build max heap for heap sort
Steps to build max heap for heap sort



Steps to build max heap for heap sort

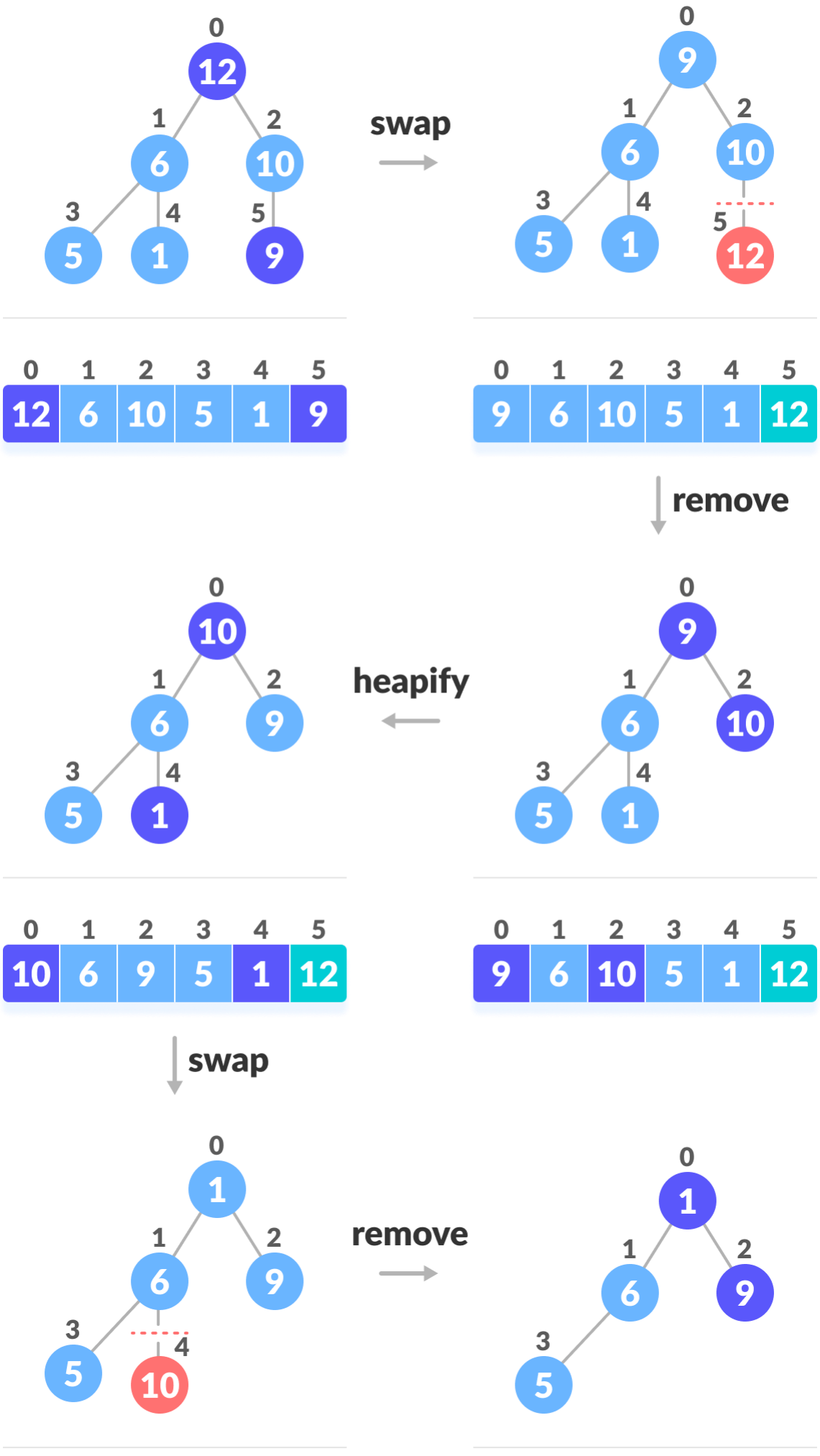
As shown in the above diagram, we start by heapifying the lowest smallest trees and gradually move up until we reach the root element.

If you've understood everything till here, congratulations, you are on your way to mastering the Heap sort.

Working of Heap Sort

1. Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.

2. **Swap:** Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.
3. **Remove:** Reduce the size of the heap by 1.
4. **Heapify:** Heapify the root element again so that we have the highest element at root.
5. The process is repeated until all the items of the list are sorted.


swap
remove
heapify
swap
remove

Swap, Remove, and Heapify

The code below shows the operation.

```
    // Heap sort
    for (int i = n - 1; i >= 0; i--) {
      swap(&arr[0], &arr[i]);

      // Heapify root element to get highest element at root again
      heapify(arr, i, 0);
    }
```

Heap Sort Code in C++

```
// C++ program for implementation of Heap Sort
#include <iostream>

using namespace std;

// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{
        int largest = i; // Initialize largest as root
        int l = 2 * i + 1; // left = 2*i + 1
        int r = 2 * i + 2; // right = 2*i + 2

        // If left child is larger than root
        if (l < n && arr[l] > arr[largest])
                largest = l;

        // If right child is larger than largest so far
        if (r < n && arr[r] > arr[largest])
                largest = r;

        // If largest is not root
        if (largest != i) {
                swap(arr[i], arr[largest]);

                // Recursively heapify the affected sub-tree
                heapify(arr, n, largest);
        }
}

// main function to do heap sort
void heapSort(int arr[], int n)
{
        // Build heap (rearrange array)
        for (int i = n / 2 - 1; i >= 0; i--)
                heapify(arr, n, i);

        // One by one extract an element from heap
        for (int i = n - 1; i > 0; i--) {
                // Move current root to end
                swap(arr[0], arr[i]);

                // call max heapify on the reduced heap
                heapify(arr, i, 0);
        }
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
        for (int i = 0; i < n; ++i)
                cout << arr[i] << " ";
        cout << "\n";
```

```
}

// Driver code
int main()
{
        int arr[] = { 12, 11, 13, 5, 6, 7 };
        int n = sizeof(arr) / sizeof(arr[0]);

        heapSort(arr, n);

        cout << "Sorted array is \n";
        printArray(arr, n);
}
```

Heap Sort Complexity

Time Complexity	
Best	O(nlog n)
Worst	O(nlog n)
Average	O(nlog n)
Space Complexity	O(1)
Stability	No

Heap Sort has `O(nlog n)` time complexities for all the cases (best case, average case, and worst case).

Let us understand the reason why. The height of a complete binary tree containing n elements is `log n`

As we have seen earlier, to fully heapify an element whose subtrees are already max-heaps, we need to keep comparing the element with its left and right children and pushing it downwards until it reaches a point where both its children are smaller than it.

In the worst case scenario, we will need to move an element from the root to the leaf node making a multiple of `log(n)` comparisons and swaps.

During the build_max_heap stage, we do that for `n/2` elements so the worst case complexity of the build_heap step is `n/2*log n ~ nlog n`.

During the sorting step, we exchange the root element with the last element and heapify the root element. For each element, this again takes `log n` worst time because we might have to bring the element all the way from the root to the leaf. Since we repeat this *n* times, the heap_sort step is also `nlog n`.

Also since the `build_max_heap` and `heap_sort` steps are executed one after another, the algorithmic complexity is not multiplied and it remains in the order of `nlog n`.

Also it performs sorting in `O(1)` space complexity. Compared with Quick Sort, it has a better worst case `( O(nlog n) )`. Quick Sort has complexity `O(n^2)` for worst case. But in other cases, Quick Sort is fast. Introsort is an alternative to heapsort that combines quicksort and heapsort to retain advantages of both: worst case speed of heapsort and average speed of quicksort.

Heap Sort Applications

Systems concerned with security and embedded systems such as Linux Kernel use Heap Sort because of the `O(n log n)` upper bound on Heapsort's running time and constant `O(1)` upper bound on its auxiliary storage.

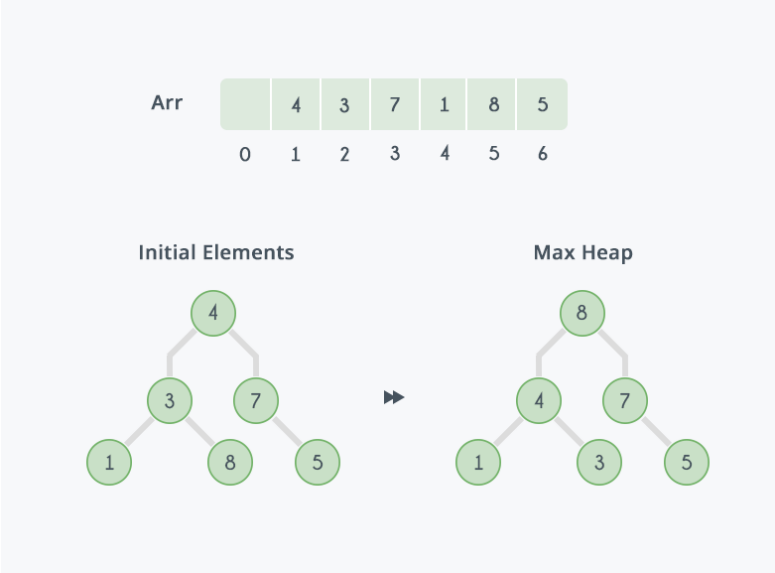
Although Heap Sort has `O(n log n)` time complexity even for the worst case, it doesn't have more applications (compared to other sorting algorithms like Quick Sort, Merge Sort). However, its underlying data structure, heap, can be efficiently used if we want to extract the smallest (or largest) from the list of items without the overhead of keeping the remaining items in the sorted order. For e.g Priority Queues.

Implementation:

```
void heap_sort(int Arr[ ])

{
   int heap_size = N;

   build_maxheap(Arr);
   for(int i = N; i >= 2 ; i-- )
   {
      swap|(Arr[ 1 ], Arr[ i ]);
      heap_size = heap_size - 1;
      max_heapify(Arr, 1, heap_size);
   }
}
```

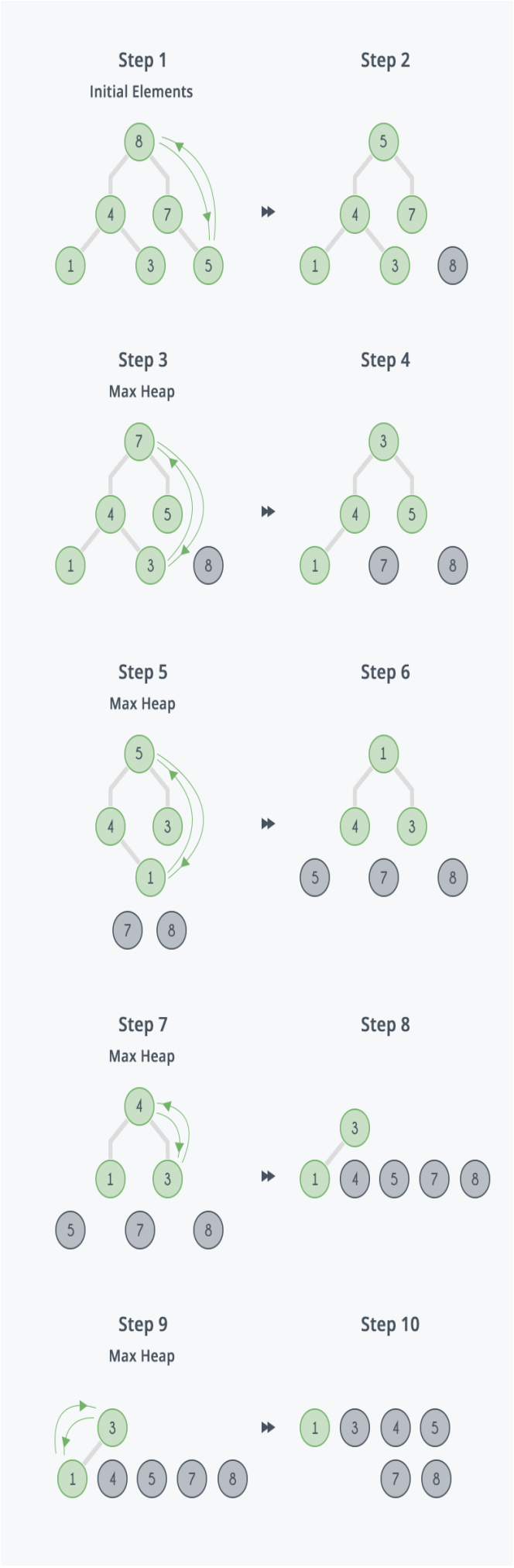
Complexity:
max_heapify has complexity O(logN), build_maxheap has complexity O(N) and we run max_heapify N−1times in heap_sort function, therefore complexity of heap_sort function is O(NlogN).

Example:
In the diagram below,initially there is an unsorted array Arr having 6 elements and then max-heap will be built.

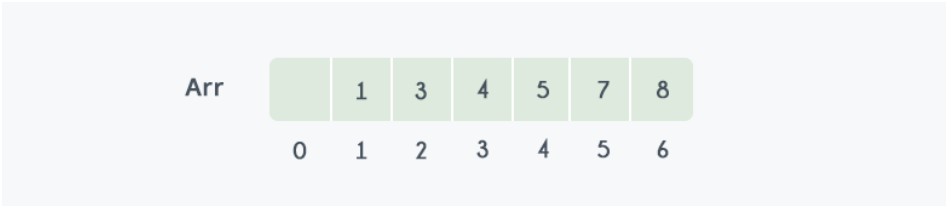


After building max-heap, the elements in the array Arr will be:

Step 1: 8 is swapped with 5.
Step 2: 8 is disconnected from heap as 8 is in correct position now and.
Step 3: Max-heap is created and 7 is swapped with 3.
Step 4: 7 is disconnected from heap.
Step 5: Max heap is created and 5 is swapped with 1.
Step 6: 5 is disconnected from heap.
Step 7: Max heap is created and 4 is swapped with 3.
Step 8: 4 is disconnected from heap.
Step 9: Max heap is created and 3 is swapped with 1.
Step 10: 3 is disconnected.


Step 1
Initial Elements
Step 2
Step 3
Max Heap
Step 4
Step 5
Max Heap
Step 6
Step 7
Max Heap
Step 8
Step 9
Max Heap
Step 10

After all the steps, we will get a sorted array.



Lecture #40

Tree

Introduction:

Trees: Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.

Tree Vocabulary: The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent. For example, ‘a’ is a child of ‘f’, and ‘f’ is the parent of ‘a’. Finally, elements with no children are called leaves.

```
      tree
      ----
       j    <-- root
     /   \
    f      k
  /   \      \
 a     h      z    <-- leaves
```

Why Trees?
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:

```
file system
-----------
     /    <-- root
  /      \
...       home
      /          \
   ugrad         course
    /       /      |      \
  ...      cs101  cs112  cs113
```

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 (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms
6. Form of a multi-stage decision-making (see business chess).

Binary Tree: A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

Binary Tree Representation in C: A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL.
A Tree node contains following parts.
1. Data
2. Pointer to left child
3. Pointer to right child

In C, we can represent a tree node using structures. Below is an example of a tree node with an integer data.

```
struct node
{
  int data;
  struct node *left;
  struct node *right;
};
```

First Simple Tree in C
Let us create a simple tree with 4 nodes in C. The created tree would be as following.

```
      tree
      ----
       1    <-- root
     /   \
    2     3
   /
  4
```

```
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

/* newNode() allocates a new node with the given data and NULL left and
   right pointers. */
struct node* newNode(int data)
{
  // Allocate memory for new node
  struct node* node = (struct node*)malloc(sizeof(struct node));

  // Assign data to this node
  node->data = data;

  // Initialize left and right children as NULL
  node->left = NULL;
  node->right = NULL;
  return(node);
}


int main()
{
  /*create root*/
  struct node *root = newNode(1);
  /* following is the tree after above statement

        1
```

```
      /   \
     NULL  NULL
  */


  root->left        = newNode(2);
  root->right       = newNode(3);
  /* 2 and 3 become left and right children of 1
           1
         /   \
        2      3
      /    \    /  \
    NULL NULL NULL NULL
  */


  root->left->left  = newNode(4);
  /* 4 becomes left child of 2
           1
        /       \
       2          3
     /   \       /  \
    4    NULL  NULL  NULL
  /  \
NULL NULL
*/

  getchar();
  return 0;
}
```

Summary: Tree is a hierarchical data structure. Main uses of trees include maintaining hierarchical data, providing moderate access and insert/delete operations. Binary trees are special cases of tree where every node has at most two children.

Properties of Binary tree:

1) The maximum number of nodes at level 'l' of a binary tree is 2^{l-1}.
Here level is number of nodes on path from root to the node (including root and node). Level of root is 1.
This can be proved by induction.
For root, l = 1, number of nodes = $2^{1-1} = 1$
Assume that maximum number of nodes on level l is 2^{l-1}
Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. $2 * 2^{l-1}$

2) Maximum number of nodes in a binary tree of height 'h' is $2^h – 1$.
Here height of a tree is maximum number of nodes on root to leaf path. Height of a tree with single node is considered as 1.
This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So maximum number of nodes in a binary tree of height h is $1 + 2 + 4 + .. + 2^{h-1}$. This is a simple geometric series with h terms and sum of this series is $2^h – 1$.
In some books, height of the root is considered as 0. In this convention, the above formula becomes $2^{h+1} – 1$

3) In a Binary Tree with N nodes, minimum possible height or minimum number of levels is $Log_2(N+1)$?
This can be directly derived from point 2 above. If we consider the convention where height

of a leaf node is considered as 0, then above formula for minimum possible height becomes $Log_2(N+1) – 1$.

4) A Binary Tree with L leaves has at least (Log$_2$L + 1) levels

A Binary tree has maximum number of leaves (and minimum number of levels) when all levels are fully filled. Let all leaves be at level l, then below is true for number of leaves L.

$L <= 2^{l-1}$ [From Point 1]

$l = Log_2L + 1$

Where l is the minimum number of levels.

5) In Binary tree where every node has 0 or 2 children, number of leaf nodes is always one more than nodes with two children.

L = T + 1

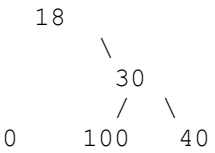
Where L = Number of leaf nodes

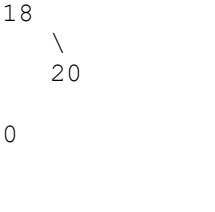
T = Number of internal nodes with two children

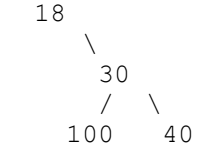
Types of Binary Tree:

Following are common types of Binary Trees.

Full Binary Tree: A Binary Tree is full if every node has 0 or 2 children. Following are examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaves have two children.







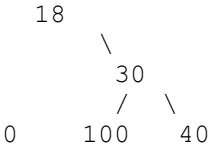
In a Full Binary, number of leaf nodes is number of internal nodes plus 1

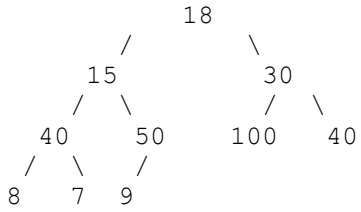
L = I + 1

Where L = Number of leaf nodes, I = Number of internal nodes

Complete Binary Tree: A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible

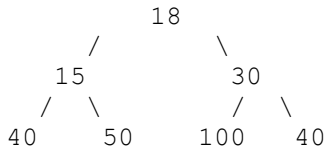
Following are examples of Complete Binary Trees

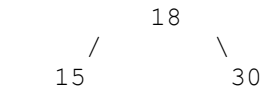




Practical example of Complete Binary Tree is Binary Heap.

Perfect Binary Tree: A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at the same level.
Following are examples of Perfect Binary Trees.





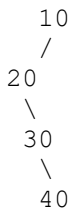
A Perfect Binary Tree of height h (where height is the number of nodes on the path from the root to leaf) has $2^h – 1$ node.

Example of a Perfect binary tree is ancestors in the family. Keep a person at root, parents as children, parents of parents as their children.

Balanced Binary Tree

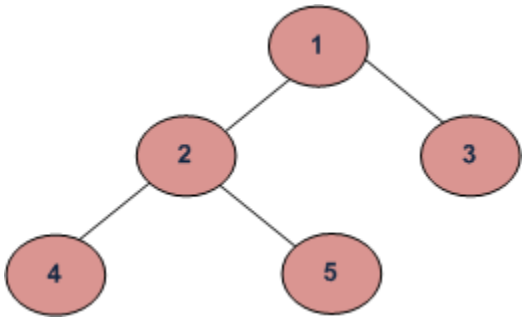
A binary tree is balanced if the height of the tree is O(Log (n)) where n is the number of nodes. For Example, AVL tree maintains O(Log n) height by making sure that the difference between heights of left and right subtrees is atmost 1. Red-Black trees maintain O(Log n) height by making sure that the number of Black nodes on every root to leaf paths are 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.

A degenerate (or pathological) tree: A Tree where every internal node has one child. Such trees are performance-wise same as linked list.



Tree Traversals (Inorder, Preorder and Postorder)

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Example Tree

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
Please see this post for Breadth First Traversal.

Inorder Traversal :

```
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)
```

Uses of Inorder
In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.
Example: Inorder traversal for the above-given figure is 4 2 5 1 3.

Preorder Traversal :

```
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)
```

Uses of Preorder
Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree.
Example: Preorder traversal for the above given figure is 1 2 4 5 3.

Postorder Traversal :

```
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.
```

Uses of Postorder
Postorder traversal is used to delete the tree. Postorder traversal is also useful to get the postfix expression/ Reverse Polish notation of an expression tree.

Example: Postorder traversal for the above given figure is 4 5 2 3 1.

```
// C program for different tree traversals
#include <iostream>
using namespace std;


/* A binary tree node has data, pointer to left child
and a pointer to right child */
```

```
struct Node
{
    int data;
    struct Node* left, *right;
    Node(int data)
    {
        this->data = data;
        left = right = NULL;
    }
};

/* Given a binary tree, print its nodes according to the
"bottom-up" postorder traversal. */
void printPostorder(struct Node* node)
{
    if (node == NULL)
        return;

    // first recur on left subtree
    printPostorder(node->left);

    // then recur on right subtree
    printPostorder(node->right);

    // now deal with the node
    cout << node->data << " ";
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct Node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    cout << node->data << " ";

    /* now recur on right child */
    printInorder(node->right);
}

/* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct Node* node)
{
    if (node == NULL)
        return;

    /* first print data of node */
    cout << node->data << " ";

    /* then recur on left sutree */
    printPreorder(node->left);

    /* now recur on right subtree */
    printPreorder(node->right);
}

/* Driver program to test above functions*/
int main()
{
    struct Node *root = new Node(1);
```

```
    root->left           = new Node(2);
    root->right          = new Node(3);
    root->left->left     = new Node(4);
    root->left->right    = new Node(5);

    cout << "\nPreorder traversal of binary tree is \n";
    printPreorder(root);

    cout << "\nInorder traversal of binary tree is \n";
    printInorder(root);

    cout << "\nPostorder traversal of binary tree is \n";
    printPostorder(root);

    return 0;
}
```

Output:

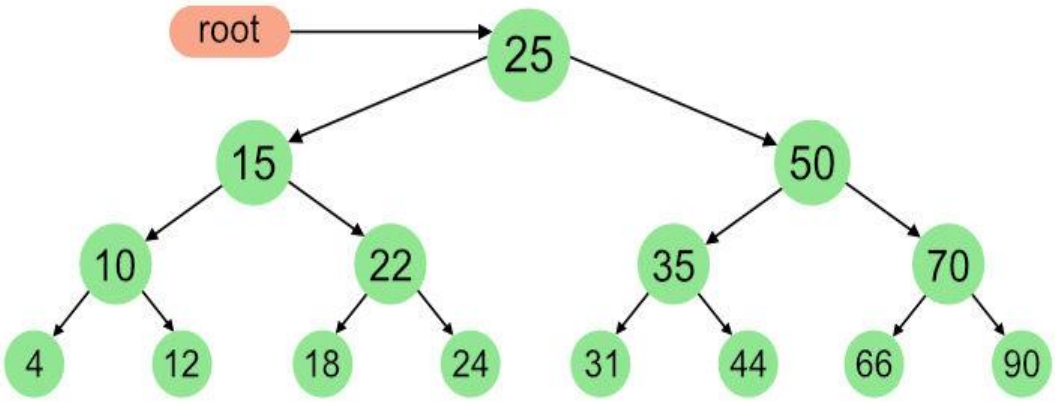
```
Preorder traversal of binary tree is
1 2 4 5 3
Inorder traversal of binary tree is
4 2 5 1 3
Postorder traversal of binary tree is
4 5 2 3 1
```

One more example:

InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Time Complexity: O(n)
Let us see different corner cases.
Complexity function T(n) — for all problem where tree traversal is involved — can be defined as:

$T(n) = T(k) + T(n – k – 1) + c$

Where k is the number of nodes on one side of root and n-k-1 on the other side.

Let's do an analysis of boundary conditions

Case 1: Skewed tree (One of the subtrees is empty and other subtree is non-empty)

k is 0 in this case.
T(n) = T(0) + T(n-1) + c
T(n) = 2T(0) + T(n-2) + 2c
T(n) = 3T(0) + T(n-3) + 3c
T(n) = 4T(0) + T(n-4) + 4c

……
…….
T(n) = (n-1)T(0) + T(1) + (n-1)c
T(n) = nT(0) + (n)c

Value of T(0) will be some constant say d. (traversing a empty tree will take some constants time)

T(n) = n(c+d)
T(n) = Θ(n) (Theta of n)

Case 2: Both left and right subtrees have equal number of nodes.

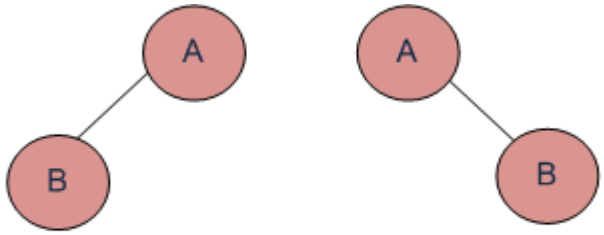
T(n) = 2T(|_n/2_|) + c

This recursive function is in the standard form (T(n) = aT(n/b) + (-)(n)) for master method. If we solve it by master method we get (-)(n).

Auxiliary Space : If we don't consider size of stack for function calls then O(1) otherwise O(n).

If you are given two traversal sequences, can you construct the binary tree?

It depends on what traversals are given. If one of the traversal methods is Inorder then the tree can be constructed, otherwise not.



Trees having Preorder, Postorder and Level-Order and traversals

Therefore, following combination can uniquely identify a tree.

Inorder and Preorder.
Inorder and Postorder.
Inorder and Level-order.

And following do not.
Postorder and Preorder.

Preorder and Level-order.
Postorder and Level-order.

For example, Preorder, Level-order and Postorder traversals are same for the trees given in above diagram.

Preorder Traversal = AB
Postorder Traversal = BA
Level-Order Traversal = AB

So, even if three of them (Pre, Post and Level) are given, the tree can not be constructed.

Construct Full Binary Tree using its Preorder traversal and Preorder traversal of its mirror tree

Given two arrays that represent Preorder traversals of a full binary tree and its mirror tree, we need to write a program to construct the binary tree using these two Preorder traversals.

A **Full Binary Tree** is a binary tree where every node has either 0 or 2 children.

Note: It is not possible to construct a general binary tree using these two traversal. But we can create a full binary tree using the above traversals without any ambiguity. For more details refer to this article.

Examples:

```
Input :  preOrder[] = {1,2,4,5,3,6,7}
         preOrderMirror[] = {1,3,7,6,2,5,4}

Output :           1
                 /    \
               2       3
             /   \   /   \
            4     5 6    7
```

Method 1: Let us consider the two given arrays as preOrder[] = {1, 2, 4, 5, 3, 6, 7} and preOrderMirror[] = {1 ,3 ,7 ,6 ,2 ,5 ,4}

In both preOrder[] and preOrderMirror[], the leftmost element is root of tree. Since the tree is full and array size is more than 1. The value next to 1 in preOrder[], must be left child of root and value next to 1 in preOrderMirror[] must be right child of root. So we know 1 is root and 2 is left child and 3 is the right child. How to find the all nodes in left subtree? We know 2 is root of all nodes in left subtree and 3 is root of all nodes in right subtree. All nodes from and 2 in preOrderMirror[] must be in left subtree of root node 1 and all node after 3 and before 2 in preOrderMirror[] must be in right subtree of root node 1. Now we know 1 is root, elements {2, 5, 4} are in left subtree, and the elements {3, 7, 6} are in the right subtree.

```
          1
        /    \
       /      \
  {2,5,4}   {3,7,6}
```

We will recursively follow the above approach and get the below tree:

```
            1
          /    \
        2       3
      /   \   /   \
     4     5 6    7
```

Below is the implementation of above approach:

```
// C++ program to construct full binary tree
// using its preorder traversal and preorder
// traversal of its mirror tree

#include<bits/stdc++.h>
using namespace std;

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node *temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to print inorder traversal
// of a Binary Tree
void printInorder(Node* node)
{
    if (node == NULL)
        return;

    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

// A recursive function to construct Full binary tree
//  from pre[] and preM[]. preIndex is used to keep
// track of index in pre[]. l is low index and h is high
//index for the current subarray in preM[]
Node* constructBinaryTreeUtil(int pre[], int preM[],
                    int &preIndex, int l,int h,int size)
{
    // Base case
    if (preIndex >= size || l > h)
        return NULL;

    // The first node in preorder traversal is root.
    // So take the node at preIndex from preorder and
    // make it root, and increment preIndex
    Node* root = newNode(pre[preIndex]);
        ++(preIndex);

    // If the current subarry has only one element,
    // no need to recur
    if (l == h)
        return root;

    // Search the next element of pre[] in preM[]
    int i;
    for (i = l; i <= h; ++i)
        if (pre[preIndex] == preM[i])
            break;
```

```
    // construct left and right subtrees recursively
    if (i <= h)
    {
        root->left = constructBinaryTreeUtil (pre, preM,
                                    preIndex, i, h, size);
        root->right = constructBinaryTreeUtil (pre, preM,
                                  preIndex, l+1, i-1, size);
    }

     // return root
    return root;
}

// function to construct full binary tree
// using its preorder traversal and preorder
// traversal of its mirror tree
void constructBinaryTree(Node* root,int pre[],
                         int preMirror[], int size)
{
    int preIndex = 0;
    int preMIndex = 0;

    root =  constructBinaryTreeUtil(pre,preMirror,
                            preIndex,0,size-1,size);

    printInorder(root);
}

// Driver program to test above functions
int main()
{
    int preOrder[] = {1,2,4,5,3,6,7};
    int preOrderMirror[] = {1,3,7,6,2,5,4};

    int size = sizeof(preOrder)/sizeof(preOrder[0]);

    Node* root = new Node;

    constructBinaryTree(root,preOrder,preOrderMirror,size);

    return 0;
}
```

Construct Special Binary Tree from given Inorder traversal

Given Inorder Traversal of a Special Binary Tree in which key of every node is greater than keys in left and right children, construct the Binary Tree and return root.

Examples:

```
Input: inorder[] = {5, 10, 40, 30, 28}
Output: root of following tree
         40
       /   \
      10     30
     /         \
    5          28

Input: inorder[] = {1, 5, 10, 40, 30,
                    15, 28, 20}
Output: root of following tree
          40
        /   \
```

```
       10      30
      /          \
     5            28
    /            /  \
   1          15    20
```

The idea used in Construction of Tree from given Inorder and Preorder traversals can be used here. Let the given array is {1, 5, 10, 40, 30, 15, 28, 20}. The maximum element in given array must be root. The elements on left side of the maximum element are in left subtree and elements on right side are in right subtree.

```
          40
        /    \
   {1,5,10}   {30,15,28,20}
```

We recursively follow above step for left and right subtrees, and finally get the following tree.

```
           40
         /    \
       10      30
      /          \
     5            28
    /            /  \
   1          15    20
```

Algorithm: buildTree()
1) Find index of the maximum element in array. The maximum element must be root of Binary Tree.
2) Create a new tree node 'root' with the data as the maximum value found in step 1.
3) Call buildTree for elements before the maximum element and make the built tree as left subtree of 'root'.
5) Call buildTree for elements after the maximum element and make the built tree as right subtree of 'root'.
6) return 'root'.

Implementation: Following is the implementation of the above algorithm.

filter_none

edit

play_arrow

brightness_4

```
/* C++ program to construct tree
from inorder traversal */
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data,
pointer to left child and
a pointer to right child */
class node
{
    public:
    int data;
    node* left;
    node* right;
};

/* Prototypes of a utility function to get the maximum
value in inorder[start..end] */
int max(int inorder[], int strt, int end);

/* A utility function to allocate memory for a node */
```

```
node* newNode(int data);

/* Recursive function to construct binary of size len from
Inorder traversal inorder[]. Initial values of start and end
should be 0 and len -1. */
node* buildTree (int inorder[], int start, int end)
{
    if (start > end)
        return NULL;

    /* Find index of the maximum element from Binary Tree */
    int i = max (inorder, start, end);

    /* Pick the maximum value and make it root */
    node *root = newNode(inorder[i]);

    /* If this is the only element in inorder[start..end],
    then return it */
    if (start == end)
        return root;

    /* Using index in Inorder traversal, construct left and
    right subtress */
    root->left = buildTree (inorder, start, i - 1);
    root->right = buildTree (inorder, i + 1, end);

    return root;
}

/* UTILITY FUNCTIONS */
/* Function to find index of the maximum value in arr[start...end] */
int max (int arr[], int strt, int end)
{
    int i, max = arr[strt], maxind = strt;
    for(i = strt + 1; i <= end; i++)
    {
        if(arr[i] > max)
        {
            max = arr[i];
            maxind = i;
        }
    }
    return maxind;
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
node* newNode (int data)
{
    node* Node = new node();
    Node->data = data;
    Node->left = NULL;
    Node->right = NULL;

    return Node;
}

/* This funtcion is here just to test buildTree() */
void printInorder (node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder (node->left);
```

```
    /* then print the data of node */
    cout<<node->data<<" ";

    /* now recur on right child */
    printInorder (node->right);
}

/* Driver code*/
int main()
{
    /* Assume that inorder traversal of following tree is given
        40
        / \
        10 30
        /    \
        5    28 */

    int inorder[] = {5, 10, 40, 30, 28};
    int len = sizeof(inorder)/sizeof(inorder[0]);
    node *root = buildTree(inorder, 0, len - 1);

    /* Let us test the built tree by printing Insorder traversal */
    cout << "Inorder traversal of the constructed tree is \n";
    printInorder(root);
    return 0;
}

// This is code is contributed by rathbhupendra
```

Output:

```
 Inorder traversal of the constructed tree is
 5 10 40 30 28
```

Time Complexity: O(n^2)

Construct a Binary Tree from Postorder and Inorder

Given Postorder and Inorder traversals, construct the tree.

Examples:

```
Input :
in[]   = {2, 1, 3}
post[] = {2, 3, 1}

Output : Root of below tree
      1
    /   \
   2     3


Input :
in[]   = {4, 8, 2, 5, 1, 6, 3, 7}
post[] = {8, 4, 5, 2, 6, 7, 3, 1}




Output : Root of below tree
          1
       /     \
```

```
     2         3
   /    \    /   \
  4      5  6     7
    \
      8
```

We have already discussed construction of tree from Inorder and Preorder traversals. The idea is similar.

Let us see the process of constructing tree from in[] = {4, 8, 2, 5, 1, 6, 3, 7} and post[] = {8, 4, 5, 2, 6, 7, 3, 1}

1) We first find the last node in post[]. The last node is "1", we know this value is root as root always appear in the end of postorder traversal.

2) We search "1" in in[] to find left and right subtrees of root. Everything on left of "1" in in[] is in left subtree and everything on right is in right subtree.

```
         1
       /    \
[4, 8, 2, 5]   [6, 3, 7]
```

3) We recur the above process for following two.
….**b)** Recur for in[] = {6, 3, 7} and post[] = {6, 7, 3}
…….Make the created tree as right child of root.
….**a)** Recur for in[] = {4, 8, 2, 5} and post[] = {8, 4, 5, 2}.
…….Make the created tree as left child of root.

Below is the implementation of above idea. One important observation is, we recursively call for right subtree before left subtree as we decrease index of postorder index whenever we create a new node.

```
/* C++ program to construct tree using inorder and
   postorder traversals */
#include <bits/stdc++.h>

using namespace std;

/* A binary tree node has data, pointer to left
   child and a pointer to right child */
struct Node {
    int data;
    Node *left, *right;
};

// Utility function to create a new node
Node* newNode(int data);

/* Prototypes for utility functions */
int search(int arr[], int strt, int end, int value);

/* Recursive function to construct binary of size n
   from  Inorder traversal in[] and Postorder traversal
   post[].  Initial values of inStrt and inEnd should
   be 0 and n -1.  The function doesn't do any error
   checking for cases where inorder and postorder
   do not form a tree */
Node* buildUtil(int in[], int post[], int inStrt,
                int inEnd, int* pIndex)
{
    // Base case
```

```
    if (inStrt > inEnd)
        return NULL;

    /* Pick current node from Postorder traversal using
       postIndex and decrement postIndex */
    Node* node = newNode(post[*pIndex]);
    (*pIndex)--;

    /* If this node has no children then return */
    if (inStrt == inEnd)
        return node;

    /* Else find the index of this node in Inorder
       traversal */
    int iIndex = search(in, inStrt, inEnd, node->data);

    /* Using index in Inorder traversal, construct left and
       right subtress */
    node->right = buildUtil(in, post, iIndex + 1, inEnd, pIndex);
    node->left = buildUtil(in, post, inStrt, iIndex - 1, pIndex);

    return node;
}

// This function mainly initializes index of root
// and calls buildUtil()
Node* buildTree(int in[], int post[], int n)
{
    int pIndex = n - 1;
    return buildUtil(in, post, 0, n - 1, &pIndex);
}

/* Function to find index of value in arr[start...end]
   The function assumes that value is postsent in in[] */
int search(int arr[], int strt, int end, int value)
{
    int i;
    for (i = strt; i <= end; i++) {
        if (arr[i] == value)
            break;
    }
    return i;
}

/* Helper function that allocates a new node */
Node* newNode(int data)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

/* This funtcion is here just to test  */
void preOrder(Node* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->left);
    preOrder(node->right);
}

// Driver code
int main()
```

```
{
    int in[] = { 4, 8, 2, 5, 1, 6, 3, 7 };
    int post[] = { 8, 4, 5, 2, 6, 7, 3, 1 };
    int n = sizeof(in) / sizeof(in[0]);

    Node* root = buildTree(in, post, n);

    cout << "Preorder of the constructed tree : \n";
    preOrder(root);

    return 0;
}
```

Output :

```
Preorder of the constructed tree :
1 2 4 8 5 3 6 7
```

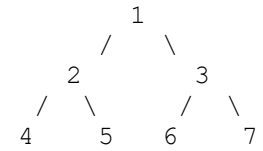
Time Complexity : $O(n^2)$

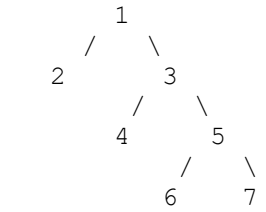
Construct Full Binary Tree from given preorder and postorder traversals

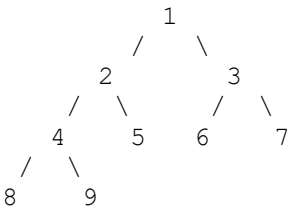
Given two arrays that represent preorder and postorder traversals of a full binary tree, construct the binary tree.

A **Full Binary Tree** is a binary tree where every node has either 0 or 2 children

Following are examples of Full Trees.



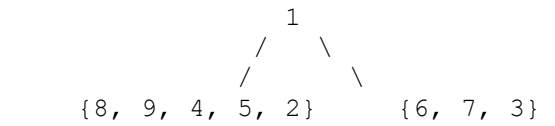




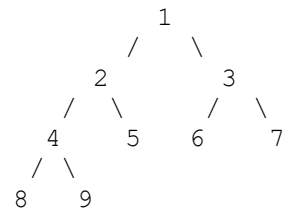
It is not possible to construct a general Binary Tree from preorder and postorder traversals (See this). But if know that the Binary Tree is Full, we can construct the tree without ambiguity. Let us understand this with the help of following example.

Let us consider the two given arrays as pre[] = {1, 2, 4, 8, 9, 5, 3, 6, 7} and post[] = {8, 9, 4, 5, 2, 6, 7, 3, 1};
In pre[], the leftmost element is root of tree. Since the tree is full and array size is more than 1. The value next to 1 in pre[], must be left child of root. So we know 1 is root and 2 is left child. How to find the all nodes in left subtree? We know 2 is root of all nodes in left subtree.

All nodes before 2 in post[] must be in left subtree. Now we know 1 is root, elements {8, 9, 4, 5, 2} are in left subtree, and the elements {6, 7, 3} are in right subtree.



We recursively follow the above approach and get the following tree.




```
/* program for construction of full binary tree */
#include <bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer to left child
and a pointer to right child */
class node
{
    public:
    int data;
    node *left;
    node *right;
};

// A utility function to create a node
node* newNode (int data)
{
    node* temp = new node();

    temp->data = data;
    temp->left = temp->right = NULL;

    return temp;
}

// A recursive function to construct Full from pre[] and post[].
// preIndex is used to keep track of index in pre[].
// l is low index and h is high index for the current subarray in post[]
node* constructTreeUtil (int pre[], int post[], int* preIndex,
                         int l, int h, int size)
{
    // Base case
    if (*preIndex >= size || l > h)
        return NULL;

    // The first node in preorder traversal is root. So take the node at
    // preIndex from preorder and make it root, and increment preIndex
    node* root = newNode ( pre[*preIndex] );
    ++*preIndex;

    // If the current subarry has only one element, no need to recur
    if (l == h)
        return root;

    // Search the next element of pre[] in post[]
    int i;
```

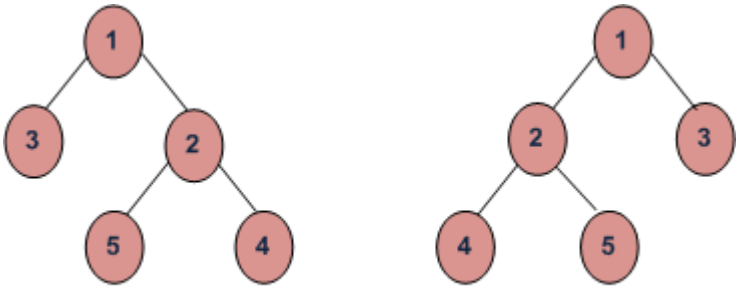
```
    for (i = l; i <= h; ++i)
        if (pre[*preIndex] == post[i])
            break;

    // Use the index of element found in postorder to divide
        // postorder array in two parts. Left subtree and right subtree
    if (i <= h)
    {
        root->left = constructTreeUtil (pre, post, preIndex,
                                                l, i, size);
        root->right = constructTreeUtil (pre, post, preIndex,
                                                 i + 1, h, size);
    }

    return root;
}

// The main function to construct Full Binary Tree from given preorder and
// postorder traversals. This function mainly uses constructTreeUtil()
node *constructTree (int pre[], int post[], int size)
{
    int preIndex = 0;
    return constructTreeUtil (pre, post, &preIndex, 0, size - 1, size);
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder (node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    cout<<node->data<<" ";
    printInorder(node->right);
}

// Driver program to test above functions
int main ()
{
    int pre[] = {1, 2, 4, 8, 9, 5, 3, 6, 7};
    int post[] = {8, 9, 4, 5, 2, 6, 7, 3, 1};
    int size = sizeof( pre ) / sizeof( pre[0] );

    node *root = constructTree(pre, post, size);

    cout<<"Inorder traversal of the constructed tree: \n";
    printInorder(root);

    return 0;
}

//This code is contributed by rathbhupendra
```

Output:

```
Inorder traversal of the constructed tree:
8 4 9 2 5 1 6 3 7
```

Convert a Binary Tree into its Mirror Tree

Mirror of a Tree: Mirror of a Binary Tree T is another Binary Tree M(T) with left and right children of all non-leaf nodes interchanged.



Mirror Trees

Trees in the above figure are mirror of each other

Method 1 (Recursive)

Algorithm – Mirror(tree):

```
(1)  Call Mirror for left-subtree    i.e., Mirror(left-subtree)
(2)  Call Mirror for right-subtree  i.e., Mirror(right-subtree)
(3)  Swap left and right subtrees.
          temp = left-subtree
          left-subtree = right-subtree
          right-subtree = temp
```

filter_none

edit

play_arrow

brightness_4

```
// C++ program to convert a binary tree
// to its mirror
#include<bits/stdc++.h>
using namespace std;

/* A binary tree node has data, pointer
to left child and a pointer to right child */
struct Node
{
    int data;
    struct Node* left;
    struct Node* right;
};

/* Helper function that allocates a new node with
the given data and NULL left and right pointers. */
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*)
                        malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}


/* Change a tree so that the roles of the left and
    right pointers are swapped at every node.

So the tree...
    4
    / \
```

```
    2 5
    / \
1 3

is changed to...
    4
    / \
    5 2
        / \
    3 1
*/
void mirror(struct Node* node)
{
    if (node == NULL)
        return;
    else
    {
        struct Node* temp;

        /* do the subtrees */
        mirror(node->left);
        mirror(node->right);

        /* swap the pointers in this node */
        temp     = node->left;
        node->left = node->right;
        node->right = temp;
    }
}

/* Helper function to print
Inorder traversal.*/
void inOrder(struct Node* node)
{
    if (node == NULL)
        return;

    inOrder(node->left);
    cout << node->data << " ";
    inOrder(node->right);
}


// Driver Code
int main()
{
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    /* Print inorder traversal of the input tree */
    cout << "Inorder traversal of the constructed"
         << " tree is" << endl;
    inOrder(root);

    /* Convert tree to its mirror */
    mirror(root);

    /* Print inorder traversal of the mirror tree */
    cout << "\nInorder traversal of the mirror tree"
         << " is \n";
    inOrder(root);
```

```
    return 0;
}

// This code is contributed by Akanksha Rai
```

Output :

```
Inorder traversal of the constructed tree is
4 2 5 1 3
Inorder traversal of the mirror tree is
3 1 5 2 4
```

Time & Space Complexities: This program is similar to traversal of tree space and time complexities will be same as Tree traversal