Module 1

Introduction

Data Structures

Def: Data Structure is a way of organizing all the data items. It contains both the elements and the relationship between them.
Example: stack, list, queue

- Affects the structural and functional aspects of programs
- A complete program = algorithm + appropriate data structure

Selection of data structures

Depends on

1. Should be able to mirror actual relationships of data as in real world
2. Simple enough to process data effectively

Classification of Data Structures

Primitive

- Basic data structures.
- Directly operated upon.
- Predefined data structures
- Type and size are predefined.
- Different computer systems = different representations. For example, integer might be 32 bits on some systems and 16 on others.
- Examples:
    a. Integer
    b. Float
    c. Character
    d. Double

Non primitive

- More sophisticated
- Derived from basic data types
- Emphasize on structuring a group of homogeneous/heterogeneous data items
    a. Linear
        i. Traversal is linear
        ii. Data elements are arranged sequentially

iii. Each element is connected with one path to other elements
    1. Array
    2. List
    3. Stack
    4. Queue

b. Non Linear
i. Traversal of nodes is non-linear.
ii. Data elements are not arranged sequentially.
iii. Single level is not involved, we cannot traverse all elements in a single run.
iv. Has no sequence of connecting all elements.
v. Each element can have multiple paths to other elements
vi. All the one-many, many-many, many-one relations are handled using non linear ds
vii. Every data element can have any number of predecessors and successors
viii. Examples:
    1. Trees
    2. Graphs

Arrays

Definition: A contiguous collection of homogeneous data.

Declaration:
int a[n]; where n is the size.

Advantages

1. Simple
2. Supported by almost all programming languages
3. Constant access time
4. Compiler maps array elements to its physical location in the memory

Disadvantages

1. Size is defined at compile time, so it is static
2. Insertion and deletion is time consuming
3. Requires a contiguous block of memory

List

Linked lists are a special type of list.
Stores data elements linked to each other.

Logic ordering is represented by each element pointing to the next.
Each element = node.
Each node has two parts.

1. INFO: stores the information/data
2. Pointer: address of next node (pointer to the next node)

Stack

1. Follows the LIFO principle: last in, first out
2. Definition: A linear data structure that follows a particular order in which operations are performed.
3. Insertion and deletion happens at ONLY ONE END, called the **top** of the stack.

Operations on Stack

1. Push operation
   Insertion of element into stack
2. Pop operation
   Removal of element from the stack

The **top** pointer helps keep track of the last element in the list. Both deletion and insertion alter this top pointer.

Applications of stack

1. Expression evaluation, syntax parsing
2. Infix to postfix and prefix conversion
3. Reversing a string
4. redo/undo
5. Forward backward in web browsers
6. Backtracking
7. Compile time memory management
8. Call stacks
9. Security: Some computing algorithms take stacks to make them vulnerable to security attacks. Understanding stack behavior helps prevent buffer overflows, stack smashing, and similar attacks.
10. Stock span problem (a specific algo)

Backtracking

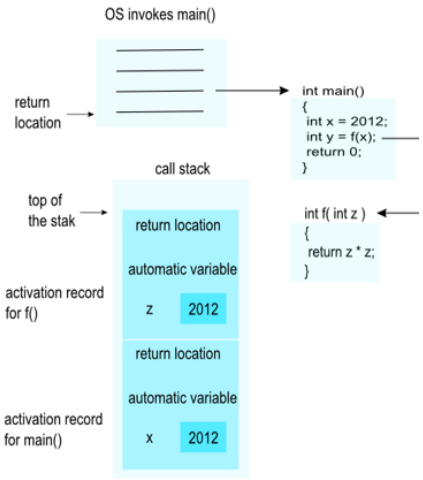
1. Solves problems recursively by building a solution one piece at a time
2. Remove solutions that do not meet the constraints
3. Example: finding the correct path in a maze. With the help of stacks, remember the point you have reached. This is done by pushing the point

to stack. If path was wrong, pop the top element and return to the last point, then go ahead for other points.

4. DFS or depth first search is another application of backtracking. Find all vertices of a graph that can be reached from a starting vertex.

Call stacks

Most programming languages are stack oriented: they take arguments from stack and return values on the stack as well.



Stacks are an important way of supporting nested function calls.
Recall how the factorial program of recursion works. Call factorial 5, then 4, then 3, and each of these are stored on the stack. During printing, the stack reverses, operates, and prints the result.

Call stack: the stack used for the function blocks to receive their parameters and return results.
//diagram: An automatic variable is a temporary local variable stored on the stack.

Queues

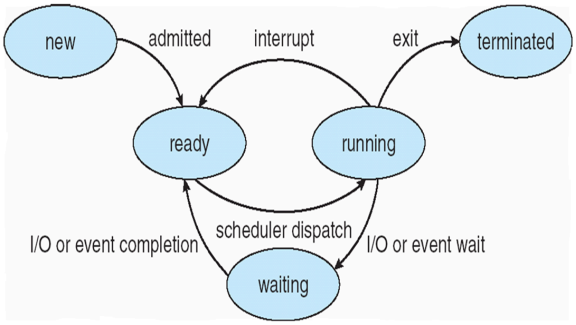
1. Follow FIFO format: first in first out. Like a regular human queue
2. **Definition:** A linear structure that follows a particular order in performing operations.
3. There are two ends. Data is inserted from one end called the rear end and removed from the other, called the front end.

Operations

1. Enqueue
   Add an element to the queue
   Happens at the rear end
2. Dequeue
   Remove an element from the queue
   Happens at the front end

Applications of Queue

1. Single shared resource like printer etc. The person to queue their document first gets it printed first.
2. Disk scheduling algorithms used by OS to decide the order of servicing disk I/O requests:
    a. FCFS → First Come First Serve
    b. SSTF → Shortest Seek Time First
    c. SCAN → Arm moves back and forth like an elevator
    d. C-SCAN → Arm moves in one direction, jumps back
    e. LOOK → Like SCAN, but stops at last request, not disk end
    f. C-LOOK → Like C-SCAN, but jumps back from last request, not disk end
3. CPU scheduling: If many processes need the CPU simultaneously, queue algorithms decide cpu scheduling.



4. Breadth first search
5. Data is transferred asynchronously between two processes.
6. Maintain playlist in media players
7. Accessing hardware like printer etc in public environment

Difference between Stacks and Queues

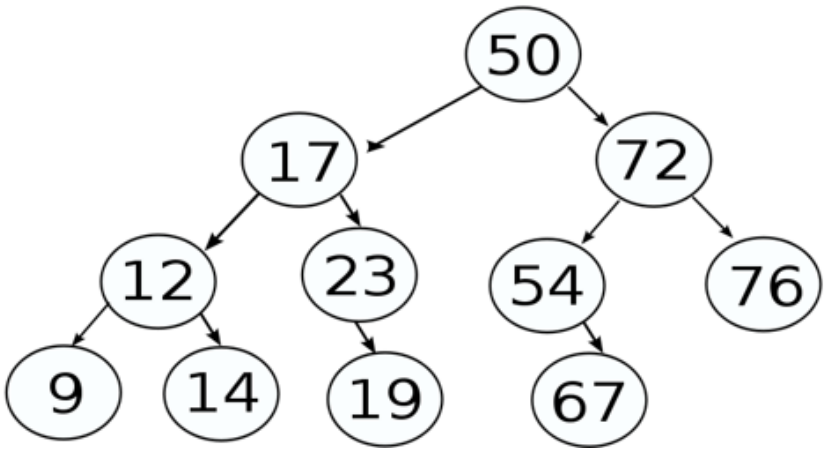
STACKS	QUEUES
Stacks are based on the LIFO principle, i.e., the element inserted at the last, is the first element to come out of the list.	Queues are based on the FIFO principle, i.e., the element inserted at the first, is the first element to come out of the list.
Insertion and deletion in stacks takes place only from one end of the list called the top.	Insertion and deletion in queues takes place from the opposite ends of the list. The insertion takes place at the rear of the list and the deletion takes place from the front of the list.
Insert operation is called push operation.	Insert operation is called enqueue operation.
Delete operation is called pop operation.	Delete operation is called dequeue operation.
In stacks we maintain only one pointer to access the list, called the top, which always points to the last element present in the list.	In queues we maintain two pointers to access the list. The front pointer always points to the first element inserted in the list and is still present, and the rear pointer always points to the last inserted element.
Stack is used in solving problems works on recursion.	Queue is used in solving problems having sequential processing.

Difference between Non Linear and Linear Data Structures

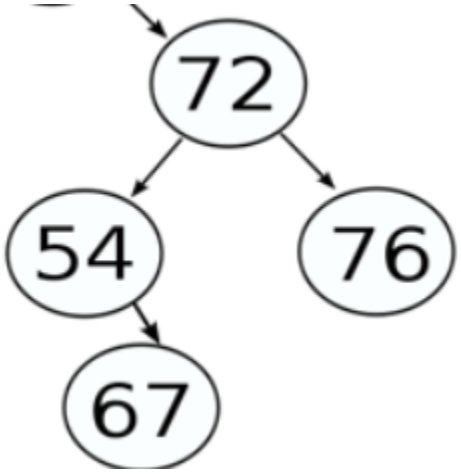
Sr. No.	Key	Linear Data Structures	Non-linear Data Structures
1	Data Element Arrangement	In linear data structure, data elements are sequentially connected and each element is traversable through a single run.	In non-linear data structure, data elements are hierarchically connected and are present at various levels.
2	Levels	In linear data structure, all data elements are present at a single level.	In non-linear data structure, data elements are present at multiple levels.
3	Implementation complexity	Linear data structures are easier to implement.	Non-linear data structures are difficult to understand and implement as compared to linear data structures.
4	Traversal	Linear data structures can be traversed completely in a single run.	Non-linear data structures are not easy to traverse and needs multiple runs to be traversed completely.
5	Memory utilization	Linear data structures are not very memory friendly and are not utilizing memory efficiently.	Non-linear data structures uses memory very efficiently.
6	Time Complexity	Time complexity of linear data structure often increases with increase in size.	Time complexity of non-linear data structure often remain with increase in size.
7	Examples	Array, List, Queue, Stack.	Graph, Map, Tree.

Trees

1. Represent hierarchy
2. Example: organization structure of a corporation: ceo-> managers -> employees
3. **Definition:** A non linear data structure that represents hierarchical relationships between several data items



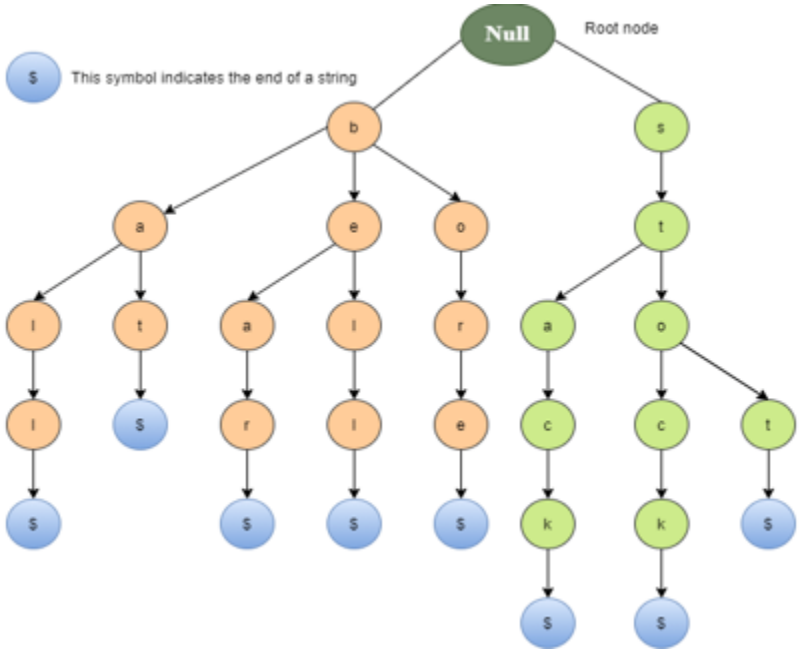
4. Finite set of data items such that there is one root (50 in our example). The remaining data items are partitioned into mutually exclusive subsets. Each subset is itself a tree, and hence called subtrees.
   Example of subtree with root 72:



Applications

1. **Store hierarchical data** like folder structure, organization structure, etc.
2. **Binary search tree** allows fast search, insert, and delete on sorted data.
3. **Heap** is a tree ds implemented using arrays and used to implement Priority Queues.
4. Multiway tree: A multiway tree is defined as a tree that can have more than two children.
5. B and B + tree: Used to implement indexing in Databases
6. Syntax trees are used in compilers

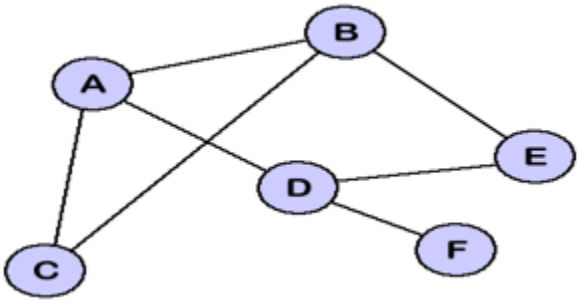
7. Spanning trees and shortest path trees: used in routers and bridges respectively in computer networks.
8. Trie: implement dictionary with prefix lookup



Graphs

Definition: Non linear data structure that represents physical data structures. A graph G(v,e) is a set of vertices and edges.

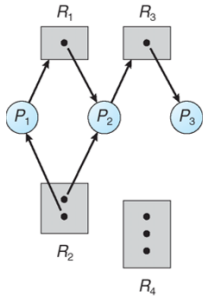
- Edge connects a pair of vertices and might have weight like length, cost etc
- Vertices: points
- Edges: lines joining them
- Edge e = v,w where v and w are vertices that the edge e connects



Applications

1. Google maps: A is connected to B and B is connected to C. So A can be reached by C and vice versa.
    a. Use graphs for building transportation systems
    b. Intersection of roads = vertex
    c. Road = edge
    d. Navigation is based on algo to calculate the shortest path between two vertices.

2. Facebook
    a. User = vertex
    b. Two friends = linked with an edge
    c. Friend suggestion uses graph theory
    d. Undirected graph example (graph without a direction)
3. Closeness centrality: calculates the shortest path bw nodes and each node gets a score of the sum of its shortest path. Helps find good broadcasters or individuals who are best placed and have most influence (connectivity)
4. World wide web
    a. Web pages = vertices
    b. Edge = between page 1 and 2 if they're hyperlinked together
    c. Example of directed graph: can reach p2 from p1 but not otherwise
    d. Basic idea behind google page ranking algorithm
5. Operating systems
    a. Resource allocation graph where vertex = each process/resource
    b. Edge = resource and process
    c. If there is a cycle, a deadlock may occur. Processes and resources are dots, arrows show who wants attention from who. If arrows make a loop, everyone waits forever – that's deadlock risk.



Abstract Data Types: ADT

Definition: a mathematical concept to define the data type.
An adt is a collection of data and associated operations for manipulating that data.
A class of objects jiska logical behaviour is defined by a set of values and operations.
The definition of an ADT mentions what operations are to be performed, not how these are implemented.
A logical description of how data is organized and the operations allowed on it, without focusing on implementation details.

An adt does not specify:
1. Data organization in memory
2. Algorithms
3. space/time complexity
4. Implementation details

Data abstraction: hiding unnecessary information and implementation details and providing only the essential information.

Encapsulation: bundling of data with methods that operate on it. Most OOP languages implement encapsulation via classes and objects.

ADTs support abstraction, encapsulation, and information hiding.

Adt operations

Every collection abstract data type should let you:

- Create that ds
- Add an item
- Remove an item
- Find, retrieve, or delete an item

ADT syntax: value definition

```
//defines the adt itself

abstract typedef <parameterType p1, parameterType p2…> ADTtype
conditions:

//defines the operations associated with that adt

abstract returnType operationName <parameterType p1, parameterType p2…>
Precondition:
Postcondition:
```

Adt: String

Value Definition

```
abstract typedef <<char s>> StringType
condition: none
```

Note: here, **StringType** is a predefined adt name. Abstract and typedef are part of the syntax. <<>> double angle brackets indicate a list of the specified data type, whereas single <> brackets indicate the single datatype.

Examples of ADT Operation Def:

```
abstract Integer StringLength (StringType String)
precondition: none
postcondition: NumberOfCharacters(String)
```

//**Integer** as returntype is capital to show that return type is also abstract
//**StringLength** is a predefined ADT condition
//**String** is a parameter name/variable.

abstract StringType StringConcat(StringType String1, StringType String2)
Precondition: None
Postcondition: StringConcat= String1+String2

abstract Boolean StringCompare(StringType String1, StringType String2)
Precondition: None
Postcondition: false if unequal, true if equal.

abstract StringType StringCopy(StringType String1, StringType String2)
Precondition: None
Postcondition: StringCopy: String1 = String2

ADT: Rational Number

Definition: expressed as fraction of two integers.
Operations: makeRational(), isEqualRational(), multiplyRational(), addRational()

Value Definition

abstract typedef <Integer x, Integer y> RationalType
condition: RationalType[1] != 0 //cannot divide by 0

Operation Definition

abstract RationalType makeRational (Integer x, Integer y)
Precondition: y!=0
Postcondition:
makeRational[0] = x
makeRational[1] = y

abstract RationalType addRational (RationalType a, RationalType b)
Precondition: none
Postcondition:
add[0] = a[0] * b[1] + b[0] * a[1]
add[1] = a[1] * b[1]

abstract RationalType multiplyRational (RationalType a, RationalType b)
Precondition: none

```
Postcondition:
mult[0] = a[0]*b[0]
mult[1] = a[1]*b[1]

abstract boolean isEqualRational (RationalType a, RationalType b)
Precondition: none
Postcondition: equal = true
a[0] * b[1] = b[0] * a[1]
```

ADT: Complex Number

Definition: Expressed as summation of the real and imaginary part.
Operations: makeComplex(), isEqualComplex(), add(), sub(), mul()

Value Definition

```
abstract typedef <Integer x, Integer y> ComplexType
condition: ComplexType[1] !=0
//if i ka part becomes zero, it’s not a complex number anymore.
```

Operation Definition

```
abstract ComplexType makeComplex (Integer x, Integer y)
precondition: y!=0
postcondition:
makeComplex[0]=x
makeComplex[1]=y

abstract ComplexType add (ComplexType x, ComplexType y)
precondition: none
postcondition:
add[0] = x[0] + y[0]
add[1] = x[1] + y[1]

abstract ComplexType sub (ComplexType x, ComplexType y)
precondition: none
postcondition:
sub[0] = x[0] - y[0]
sub[1] = x[1] - y[1]

abstract ComplexType mult (ComplexType x, ComplexType y)
precondition: none
postcondition:
mult[0] = a[0]*b[0] - a[1]*b[1]
```

mult[1] = a[0]*b[1] + a[1]*b[0]

Advantages of ADT

- Hide unnecessary details by building walls around data and operations
- Functionalities don't change very often
- Localize instead of globalizing changes
- Help manage software complexity
- Easier to maintain software

An ADT operation provides an interface to data structure and secure access. Violating Abstraction: User programs should not be able to use the underlying data structure directly, and should not depend on implementation details.

List ADT

This data structure is generally stored in key sequence in a list. It has a head structure, and consists of counts, pointers, and address of the compare function needed to compare the data in the list.

List ADT Functions

- get() – Return an element from the list at any given position.
- insert() – Insert an element at any position of the list.
- remove() – Remove the first occurrence of any element from a non-empty list.
- removeAt() – Remove the element at a specified location from a non-empty list.
- replace() – Replace an element at any position by another element.
- size() – Return the number of elements in the list.
- isEmpty() – Return true if the list is empty, otherwise return false.
- isFull() – Return true if the list is full, otherwise return false.

Stack ADT

Instead of data stored in each node, the pointer to the data is stored. The program allocates memory for data and address both.
The head node and data nodes are encapsulated in the ADT. the calling function only has access to the pointer. The stack head structure contains a pointer to the top and number of elements in the stack.

Stack ADT functions

- push() – Insert an element at one end of the stack called top.

- pop() – Remove and return the element at the top of the stack, if it is not empty.
- peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
- size() – Return the number of elements in the stack.
- isEmpty() – Return true if the stack is empty, otherwise return false.
- isFull() – Return true if the stack is full, otherwise return false.

Queue ADT

The Queue ADT follows the basic design of the stack ADT. each node 1. Data 2. Link pointer to next element in queue.
Operations take place at both ends. Deletion: front. Insertion: back

Queue ADT functions

- enqueue() – Insert an element at the end of the queue.
- dequeue() – Remove and return the first element of the queue, if the queue is not empty.
- peek() – Return the element on top of the queue without removing it, if the queue is not empty.
- size() – Return the number of elements in the queue.
- isEmpty() – Return true if the queue is empty, otherwise return false.
- isFull() – Return true if the queue is full, otherwise return false.

Module 2

Memory Allocation

Compile Time/ Static Allocation

- Allocated to the program at the start of the program
- Allocated for declared variables by the compiler
- Memory allocated is fixe
- Ex: float a[5] is an array of floats of size 5. So memory allocated is 5 times size of float = 5*4 bytes
- **Drawback:** Causes under-utilization of memory if less than mentioned number of elements are used. For example, declared array is size 101, but your array had only 4 elements, so memory is wasted. There is no reusability of allocated memory.
- **Drawback:** Can cause overflow if you try to access memory that was never allocated. For example, if declared array is float a[5], you cannot access a[5] as it is out of bounds. What will you allocate as size if you do not know size beforehand to ensure no overflow or underflow occurs? (nothing)

Runtime/ Dynamic Allocation

- All linked data structures are preferably implemented by runtime allocation.
- Provide flexibility in adding, deleting, and rearranging objects at runtime.
- Managed via these functions in C
    - malloc()
    - calloc()
    - free()
    - realloc()
- Memory is allocated at the time of execution
- Space required by variables is calculated and allocated while execution
- Best suited when we do not know size in advance.
- Helps with efficient use of memory: additional space can be allocated, unwanted space can be released
- Reusability of memory space using free() function

malloc()

- Stands for memory allocation
- Allocates a block of memory in bytes

- User should specifically give block size when needed
- Requests RAM to allocate memory. If granted, returns a pointer to the first block of memory. Else, returns null.
- malloc() gives a **contiguous memory** block. So if there is no space for 5 elements to be stored consecutively in the ram, it returns null. !!!!!!!!
- Type of pointer is void (can be assigned to any pointer type)
- header file alloc.h / stdlib.h
- Does not initialize the allocated memory. If you access it before giving it a value, you get segmentation error/garbage value

Syntax Eg-

```
int size = sizeof(float)*5;
float *ptr;
ptr = (float*) malloc (size);
```

Here, 20 bytes are allocated, void pointer is casted to float and assigned to ptr.

Eg in Linked List

```
struct student;
{
      int roll_no;
      char name[30];
      float percentage;
};
struct student *st_ptr;
st_ptr = (struct student *)malloc(10*sizeof(struct student));
```

calloc()

- Stands for contiguous allocation
- Similar to malloc
- Only difference is that it needs two arguments instead of one.

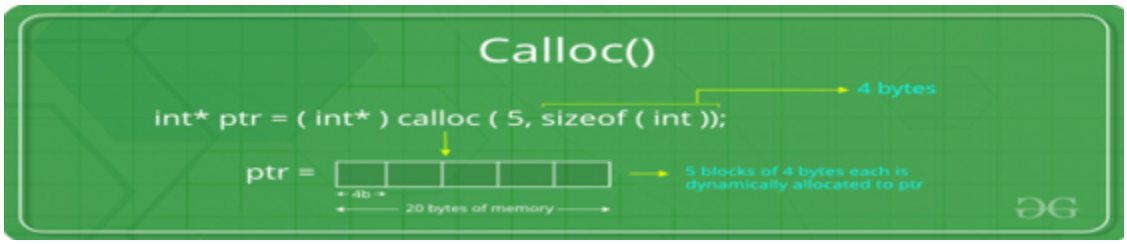
Syntax example:

```
int *ptr;
ptr = (int*) calloc(10,2);
//allocate 10 elements ke liye 2 byte each space. total=20
```

- Same head files as malloc
- Initializes the allocated memory to zero. So if you access it before giving any value, get 0

Difference between calloc and malloc

Gives a single large block of memory of the size required.



Allocates multiple contiguous blocks of memory.

free()

- Used to deallocate the previously allocated memory using calloc or malloc
- Syntax: free(ptr);
- Returns the freed memory to the RAM
- Free is not compulsory to use, but malloc ka allocated memory is not deallocated automatically. So free() helps do just that.
- If free is not explicitly used, the memory allocated by malloc is deallocated automatically only after the program ends.

Why use free?

Programs like digital clock run in the background for a very long time and allocate memory periodically, if memory is not deallocated, this will result in bugs and endless memory usage. This is called memory leak. Eventually, system will run out of memory.

Memory leak: A type of resource leak when a computer incorrectly manages memory allocation.

realloc()

- Resize the size of memory block that was already allocated by malloc
- Used when allocated memory is insufficient or allocated memory is much more than required.
- ptr = realloc(ptr, size);
- Can only resize memory allocated by malloc calloc

Difference between Static and Dynamic Memory Allocation

S. No	Static Memory Allocation	Dynamic Memory Allocation
1	In the static memory allocation, variables get allocated permanently, till the program executes or function call finishes.	In the Dynamic memory allocation, the memory is controlled by the programmer. It gets allocated whenever a malloc() is executed gets deallocated wherever the free() is executed.
2	Static Memory Allocation is done before program execution.	Dynamic Memory Allocation is done during program execution.
3	It uses stack for managing the static allocation of memory	It uses heap (not heap data structure) of memory for managing the dynamic allocation of memory
4	It is less efficient	It is more efficient
5	In Static Memory Allocation, there is no memory re-usability	In Dynamic Memory Allocation, there is memory re-usability and memory can be freed when not required
6	In static memory allocation, once the memory is allocated, the memory size can not change.	In dynamic memory allocation, when memory is allocated the memory size can be changed.

7	In this memory allocation scheme, we cannot reuse the unused memory.	This allows reusing the memory. The user can allocate more memory when required. Also, the user can release the memory when the user needs it.
8	In this memory allocation scheme, execution is faster than dynamic memory allocation.	In this memory allocation scheme, execution is slower than static memory allocation.
9	In this memory is allocated at compile time.	In this memory is allocated at run time.
10	In this allocated memory remains from start to end of the program.	In this allocated memory can be released at any time during the program.
11	**Example:** This static memory allocation is generally used for array.	**Example:** This dynamic memory allocation is generally used for linked list.

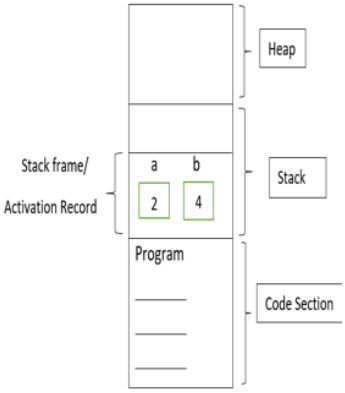
Pointers and Arrays

A pointer can be used as an array. Condition: it should point to enough memory.

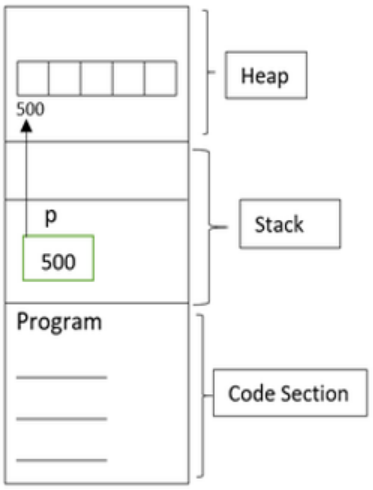
```
Example:
int *ip;
ip = (int *) malloc(sizeof(int)*10 ); // allocate 10 ints for array of size10

ip[6] = 42; // set the 7th element to 42
ip[10] = 99; // WRONG: array only has 10 elements
(this would corrupt the memory!, does not give compilation error but may
access memory out of bounds / crash at runtime)
```

Stack & Heap for Memory Allocation

NOTE: Memory is allocated from stack for static allocation. There is a limit on the size of variables that a stack can store. Memory is allocated from heap for dynamic allocation. Heap is a more free floating region of memory. Does not have a limit unlike stack.



```
void main(){
      int a;
      long b;
}//allocates memory to stack.
```




```
void main(){
   int *p; //2 bytes
   p = (int*) malloc(5*sizeof(int));
}//allocates memory from heap
```

here, p has 500 in the stack as that might have been the address returned by malloc() for the allocated block of heap. pointers store addresses.

When you free p, the heap gets free. Assigning null to a pointer is good practice to avoid a dangling pointer.

Memory is divided into addressable units. These units are called bytes and each byte has an address.
The memory is divided into three sections:

1. Heap
2. Stack
3. Code

Stack and heap are stored in RAM.

To allocate memory on heap, use calloc and malloc.
Deallocate memory on heap, use free.
If you dont use free, it leads to memory leak.

In a stack, you do not have to deallocate explicitly. After termination, memory is freed automatically. It has a very fast access to elements but variables cannot be resized. Local access of variables is allowed only.

In a heap, you’re in charge of allocating and freeing memory. Does not have size restrictions but working is slower because pointers are used. Global access of variables is allowed.

Linked List

Definition: A linear collection of data elements called nodes. The linear order is given by pointers.

Each node has two parts: storing data and storing the address of the next element.

The last node's address field contains NULL pointer as there is no element after it and hence points to nothing. The NULL pointer indicates the end of the list.

Advantages of LL over Arrays

1. Dynamic: grow and shrink at runtime
2. Memory utilization: no preallocation of memory, allocate as needed
3. Easier insertion and deletion: provides flexibility in inserting/deleting a data item. (only pointers need to be manipulated)

Disadvantages

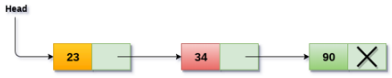
1. In linked lists, accessing an arbitrary element requires traversing from the head, so it's slower than arrays.
2. Instead of just one (like in arrays) two fields per element is needed, so it uses more space.

Types of Linked List

1. Singly linked list
2. Doubly linked list
3. Circular linked list

Singly linked list

- All nodes are linked in sequence
- Linear
- One way: only move forward
- Has a beginning and an end
- **Drawback:** Cannot access predecessor of a node

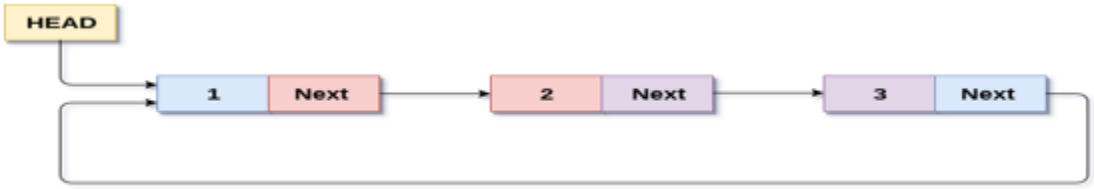


Doubly Linked List

- Holds two pointer fields
- Address of next and address of previous node is linked with each node
- Two way: traverse in both directions

Circular Linked List

- First and last elements are connected to each other
- A linked list can be made circular by storing the address of first node in the pointer field of last node, instead of null pointer



Operations on Linked List

1. Creation
2. Insertion
3. Deletion
4. Traversal
5. Searching

Creation: Singly linked list

```
struct node{
      int info; //stores the value
      struct node *link; // ptr to next node
}
```

Creation of new node:

```
struct node *tmp;
tmp = (struct node *) malloc(sizeof(struct node));
tmp -> info = data;
tmp -> link = NULL;
```

Creating a node and adding it at the end of linked list

Algo:

1. Check if memory is available, if not print error and exit

2. If yes, create a new node
3. Initialize q = start of linked list
4. Traverse the linked list till q reaches the end.
5. Link q ka next to temp
6. Temp ka next is null
7. End

```
create_list(int data)
{
      struct node *q,*tmp;
      tmp= (struct node *) malloc(sizeof(struct node));
      tmp->info=data;
      tmp->link=NULL;
      //adding temp to list,
      //temp will be the last element and hence point to null.

      if(start==NULL) /*If list is empty */
      {
            start=tmp;
      }
      else
      {
            q=start;
            while(q->link!=NULL)
                  {
                        q=q->link;
                  }
            q->link=tmp;
      }
}
```

Traversal

C++code:

```
struct node *q = start;
while(q!=NULL) //while end of list is not encountered
{
      cout << q->info << endl; //print the info at this node
      q = q->link; //move q forward
}
```

Algo:

1. Initialize, set ptr = start

2. Print ptr->info
3. Set ptr = ptr->link
4. Repeat 2 and 3 till q becomes null
5. Exit

Searching

Algo:

1. Traverse
2. Compare info part to search query
3. If found, print result and early exit
4. If not found and end is reached, print not found
5. End

Code:

```
search(int data)
{
	struct node *ptr = start;
	int pos = 1;
	while(ptr!=NULL)
	{
		if(ptr->info==data)
		{
			printf("Item %d found at position %d\n",data,pos);
			return;
		}
		ptr = ptr->link;
		pos++;
	}
	if(ptr == NULL)
		printf("Item %d not found in list\n",data);
}
```

Insertion

At beginning

Algo:

1. Check if memory for new node is available, if not, print overflow else step2
2. Create a new node
3. Connect its link part to the original first node
4. Connect start pointer to the new node

Pseudocode:

```
struct node* temp;
temp = (struct node*) malloc(sizeof(struct node));
if(temp==NULL) exit;
temp->info = data; //input
temp->link = null; //for now

temp->link = start; //temp points to first node.
start = temp; //temp becomes first node
```

In between insertion

Algo:

1. Check if memory is available, if not print error and exit
2. If yes, create a new node
3. Traverse the linked list till the index at which insertion is required.
4. Link temp node to its successor
5. Link its predecessor to it
6. End

Pseudocode:

```
pos = kahan insert karna hai
q=start;
for(i=0;i<pos-1;i++)
      {
            q=q->link;
            if(q==NULL)
            {
                  printf("There are less than %d elements",pos);
                  return;
            }
      }
//now, q has reached the point JUST before we want to insert temp.
temp->info = data;//input
temp->next = q->link;
q->link = temp;
```

Deletion

In beginning

Algo:

1. If start=null, underflow and exit
2. Connect start pointer to second node
3. Delete first node
4. End

Pseudocode:

```
struct node *q = start;
start = start->link; //nothing is pointing to 1st element ab
free(q); //or delete q in cpp
```

In between

Algo:

1. Let x=node to be deleted
2. Connect the previous node of x to next node of x
3. Delete x

Pseudocode:

```
int key=5;//value to delete

while(temp != NULL && temp->info != key) {
    temp = temp->link;
}//find key in list (traversal)

struct node* q= start;
while(q->link!=temp){
      q=q->link;
}
q->link = temp->link
free (temp);
```

At the end

Algo:

1. Find position of the node
2. Traverse till you find q->next = temp
3. Then q->next = null (now nothing points to temp)
4. Delete temp
5. End

Pseudocode:

```
struct node* q = start;
while(q->link->link != null){
      //q ke baad ke baad null nahi aana chahiye
      q = q->link;
}
q->link = null;
```

Circular Linked List

1. In a singly linked list, it is impossible to go backwards
2. By just linking first to last, you can go back in circles.
3. Creation is same as singly linked list, just last node points to first node
4. last -> link = first;

Advantages

1. Can easily traverse to prev node
2. Entire list can be traversed starting from any node.
3. For insertion at end, you dont have to traverse the entire list
4. Insertion in beginning or end takes constant time.

Disadvantages

1. Complex as compared to singly linked list
2. Reversing a circular linked list is complex compared to singly linked lists and doubly linked lists
3. We could end up in an infinite loop
4. Does not support direct access of elements

Insertion

In an empty linked list

```
//create a new node, add data to its info part and set link to null. last and first is provided in link definition.

if (last==NULL)
{
      last=tmp;
      tmp->link=last;
}//points to itself
```

At the end

```
tmp ->link = last->link;  //last pointed to first, now temp also points to first
last->link = tmp; //last now points to temp
last = tmp; //temp becomes last
```

At the beginning

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

In between

Q is before temp. Find it using traversal.

```
temp-> link = q-> link
temp-> info = num;
q->link = temp;
```

Creation

```
struct node*q, *temp;
temp = malloc(sizeof(struct node));
temp->info=num;

if(last==NULL) //circular linked list is empty
{
      last = temp;
      temp->link = last;
} //temp becomes the last element. It points to itself since list was empty

else{
      temp->link = last-> link;
      last->link=temp;
      last = temp;
}
```

Traversal

```
struct node*q;
if(last==NULL)
{
//empty list print
return;
}
while(q!=last)
{
      print q->info;
      q=q->link;
}
print last-> info
```

Deletion

Only one element exists

```
Num = to be deleted

if(last->link == last && last->info==num)
{
      temp = last;
      last = NULL;
      free(temp);
}
```

Delete first node

```
q = last->link;
if(q->info == num)
{
      temp = q;
      last->link = q->link;
      free(temp);
}
```

In between

```
q = last->link; //first element
while(q->link!=last)//infinite
{
      if(q->link->info == num)
      {
            temp = q->link; //this is to be deleted
            q->link = temp->link;
            free(temp);
      }
      q=q->link;
}
//q points to last now
if(q->link->info == num)
{
      temp = q->link;
      q->link = temp->link;
      free(temp);
      last = q;
}
```

```
//deal separately.
```

Doubly Linked List

Drawbacks of SLL and CLL:

1. Single linked list: only one side se traversal
2. Circular: to visit just previous wala node, you have to traverse all n-1 nodes first.

Data structure:

```
struct node
{
      struct node *prev; //pointer to previous element
      int info; //data
      struct node *next; //pointer to next element
}*start; //where it starts
```

Insertion

At the beginning 5 steps

```
Create node temp

temp->next = start
temp->info = data
start->prev= temp
start=temp
temp->prev=NULL
```

In between

```
Traverse to obtain q. This is the node just before our insertion point.

temp->prev = q;
temp->next = q->next;
q->next->prev = temp; //q ke baad wale node ke peeche add temp
q->next = temp;

//order is very important
```

At last

```
Traverse to obtain q such that q->link=NULL
```

```
temp->prev = q;
temp->next = NULL;
q->next = temp;
```

Deletion

At beginning

```
temp = start;
start = start->next;
start->prev = NULL;
free(temp)
```

In between

Reach q after which u want to delete

```
temp = q->next;
q->next = temp->next;
temp->next->prev = q;
free(temp)
```

Last node

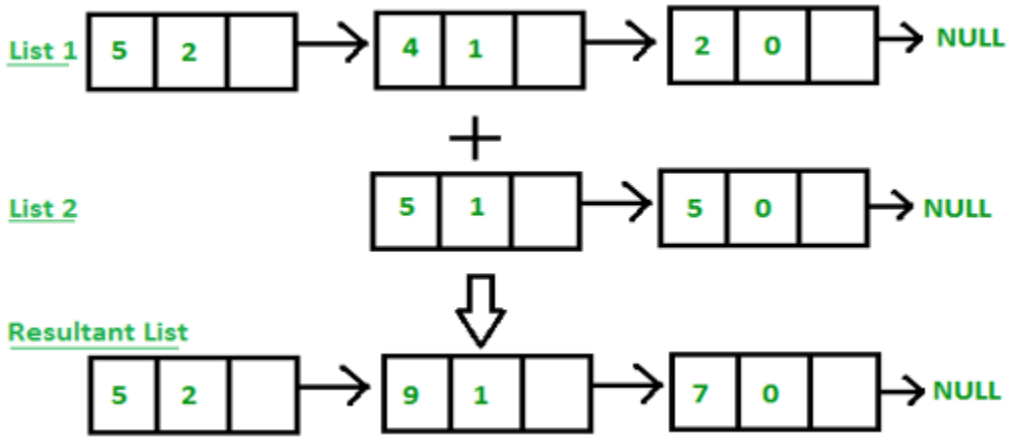
Traverse till 2nd last node.

```
temp = q->next;
q->next = NULL;
free(temp)
```

Application of linked list

1. Polynomial representation and addition
   Each node represents the coefficient of the polynomial. When the end of a list is met, the remaining nodes of the other lists are added to

corresponding node of resultant list.



Stacks

Definition: Linear data structure that follows a particular order of operations. LIFO/FILO

Operations:

Top is initially -1.

Push

Insertion of elements
Top is increased by 1
a[top]=num;

Overflow condition:
if(top==size-1)
Overflow occurs. We cannot have a[5] in array size 5, and top cannot be increased after 4.

Algo:

1. Check Stack Full condition
2. If(top==MAX-1) print stack is full or overflow
3. Otherwise increase the top value by 1
4. Input the value
5. Assign the item at top position stack_arr[top]=pushed_item;
6. End

Pop

Removal of elements
Removal of a[top] happens.
Top is decreased by 1
Underflow:
if(top==-1)
Stack is empty, underflow occurs

Algo:

1. Check Stack Underflow condition
2. If(top== -1) print stack is underflow
3. Otherwise, delete the top position element: Popped_item = stack_arr[top]
4. Decrease the position of top top=top-1
5. Print the popped_item
6. End

Display

Code:

```
display()
{
      int i;
      if(top == -1)
            printf("Stack is empty\n");
      else
      {
            printf("Stack elements :\n");
            for(i = top; i >=0; i--)
                  printf("%d\n", stack_arr[i] );
      }
}
```

Peek

Print topmost element without removal.
In single ended data structures like stack, peek happens at one end.
In doubly ended data structures like dequeues, peek happens at both ends.

Linked List Representation of Stack

The info field of the node holds element of stack
Link holds pointers to the neighboring elements of stack
Start pointer = TOP
Null pointer = END of stack

```
struct node
{
      int info;
      struct node *link;
} *top = NULL;
```

Initially, stack is empty so top points to null.

Push on stack via LL

Pseudocode:

```
tmp = (struct node *)malloc(sizeof(struct node));
tmp->link = top;
top=tmp;
```

Code:

```
push()
{
      struct node *tmp;
      int pushed_item;
      tmp = (struct node *)malloc(sizeof(struct node));
      printf("Input the new value to be pushed on the stack : ");
      scanf("%d",&pushed_item);
      tmp->info=pushed_item;
      tmp->link=top;
      top=tmp;
}
```

Pop on stack using LL

Pseudocode:

```
temp = top;
print(temp->info);
top = temp->next;
free(temp)
```

Code:

```
pop()
{
      struct node *tmp;
      if(top == NULL)
            printf("Stack is empty\n");
      else
      {
            tmp=top;
            printf("Popped item is %d\n",tmp->info);
            top=tmp->link;
            free(tmp);
      }

}
```

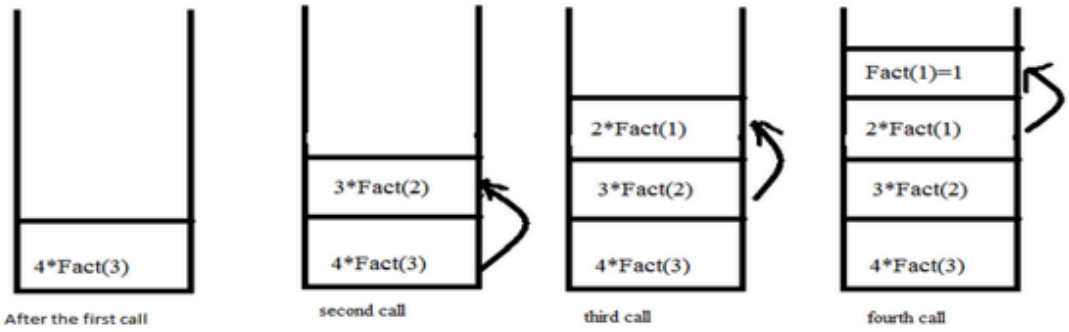
Application of stacks

1. Push each character of the string on the stack. When whole string has been pushed, pop one by one to reverse the string
2. Polish notation: infix, postfix, prefix

//should know manual infix to prefix, postfix to infix bla bla
//should know by stack ka three columns

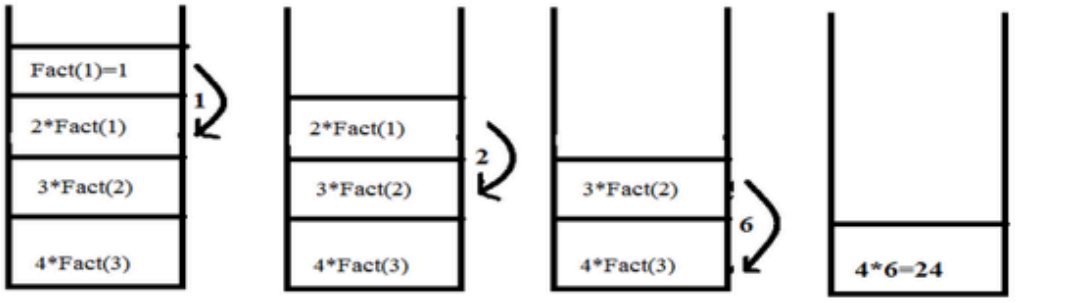
3. Recursion: The called function gets inside the caller function and it forms a stack. The caller has to stop temporarily until all called functions are completely executed. Example of recursion using stacks alr done in mod1 ke notes. This is called a call stack.

When function call happens previous variables gets stored in stack



After the first call | second call | third call | fourth call

Returning values from base case to caller function



Queues

Definition: an ordered list of elements in which addition happens at one end (rear) and deletion at the other(front).

Example:
Drive thru at mcdonalds

Applications of Queues

Ready queue

Operating systems maintain a queue of processes ready to execute.

Buffer

A buffer is a temporary storage area for data between two processes. It holds messages so they can be handled in the order they arrive. Usually implemented as a queue to keep the time order intact.

//should know basic theory of how a queue works, and implementation using array

Queue implementation using arrays

Two variables: rear and front

Overflow

```
if(rear==max-1)
Queue overflowed
```

Underflow

```
if(front==-1 or front > rear)
Underflowed
```

Implementation

```
#define MAX 5
int queue_arr[MAX];
int rear = -1;
```

```
int front = -1;
```

Insertion

Algo:

1. Check for overflow condition. If yes, exit
2. else, if front is -1, this element is the first element, increase front to 0.
3. Increase rear by 1
4. Add the element to rear’th index
5. End.

Code:

```
insert()
{
      int num;
      if (rear==MAX-1)
            printf("Queue Overflow\n");
      else
      {
            if (front==-1)
                  front=0;

            printf("Input the element for adding in queue : ");
            scanf("%d", &num);

            rear=rear+1;
            queue_arr[rear] = num ;
      }
}
```

Deletion

Algo:

1. Check for underflow condition. If met, cant delete, exit.
2. Else, print the front wala element
3. Increase front by 1.
4. End

Code:

```
del()
```

```
{
      if (front == -1 || front > rear)
      {
            printf("Queue Underflow\n");
            return ;
      }
      else
      {
            printf("Element deleted from queue is : %d\n", queue_arr[front]);
            front=front+1;
      }
}
```

Display

Algo:

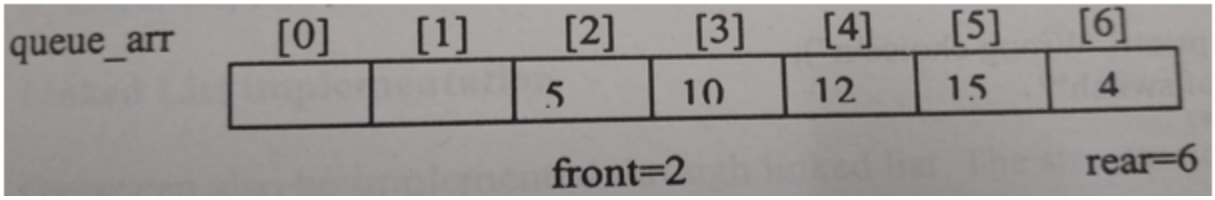
1. Check underflow, exit if yes
2. Print all front elements one by one till underflow cond front>rear
3. Increase front at each instance
4. End

Code:

```
display()
{
      int i;
      if (front == -1)
            printf("Queue is empty\n");
      else
      {
            printf("Queue is :\n");
            for(i=front;  i<= rear;   i++)
                  printf("%d  ",queue_arr[i]);
            printf("\n");
      }
}
```

Limitation

If rear is at the last position of the array and front is not at the 0th position.
If all elements were added, then 2 were removed, front now sits at 2’nd index or third element. Cannot add more elements since rear is at end but technically we have more space since first two indices are free.



We can shift all elements to the left and have rear != capacity
Stupid approach
Smart approach: circular queue

ISRO | ISRO CS 2017 | Question 53
The minimum number of stacks needed to implement a queue is
(A) 3
(B) 1
(C) 2
(D) 4

Ans: C

Linked list representation of queue

Need two pointers: front and rear

Insertion

Always add at the end of linked list

```
insert()
{
      struct node *tmp;
      int num;
      tmp = (struct node *)malloc(sizeof(struct node));

      printf("Input the element for adding in queue : ");
      scanf("%d",&num);

      tmp->info = num;
      tmp->link=NULL; //tmp becomes the last element
      if(front==NULL)
            front=tmp;
            rear=tmp;
      else
            rear->link=tmp;
      rear=tmp;
```

```
}
```

Deletion

```
del()
{
      struct node *tmp;
      if(front == NULL)
            printf("Queue Underflow\n");
      else
      {
            tmp=front;
            printf("Deleted element is  %d\n",tmp->info);
      front=front->link; //element behind front becomes new front.
            free(tmp);
      }
}
```

Display

```
display()
{
      struct node *ptr;
      ptr = front;
      if(front == NULL)
            printf("Queue is empty\n");
      else
      {
            printf("Queue elements :\n");
            while(ptr != NULL)
            {
                  printf("%d ",ptr->info);
                  ptr = ptr->link;
            }
            printf("\n");
      }
}
```

```
Link is:
F>X>X>X>X>R>N

F- front
```

X-other elements
R-rear
N-null

Circular queue

Last element of queue ke baad first element aajaega

Implementation by arrays

```
# define MAX 5
int cqueue_arr[MAX];
int front = -1;
int rear = -1;
```

Overflow:

Front = rear+1
Or front=0 and rear = max-1
Basically when front and rear are adj.

```
if((front == 0 && rear == MAX-1) || (front == rear+1))
    {
        printf("Queue Overflow \n");
        return;
    }
```

If r=n-1, set r=0 //to the front of the queue.
//only if no overflow

Insertion

```
void insert() {
    int num;

    if ((front == 0 && rear == MAX - 1) || (front == rear + 1)) {
        printf("Queue Overflow \n");
        return;
    }

    if (front == -1) {
        front = 0;
        rear = 0;
```

```
    } else if (rear == MAX - 1) {
        rear = 0;
    } else {
        rear++;
    }

    printf("Input the element for insertion in queue: ");
    scanf("%d", &num);
    queue[rear] = num;
}
```

Deletion

```
front=-1, queue is empty
front =n-1, delete it and set front = 0
If front == rear and rear!=-1, that means only one element in queue. On
deletion, front and rear=-1

void del() {
    if (front == -1) {
        printf("Queue Underflow\n");
        return;
    }

    printf("Element deleted from queue is: %d\n", cqueue_arr[front]);

    if (front == rear) {
        front = -1;
        rear = -1;
    } else if (front == MAX - 1) {
        front = 0;
    } else {
        front++;
    }
}
```

Display

```
int front_pos = front,rear_pos = rear;
      if(front == -1)
      {
            printf("Queue is empty\n");
            return;
      }
```

```
printf("Queue elements :\n");
      if( front_pos <= rear_pos )//normal queue
            while(front_pos <= rear_pos)
            {
                  printf("%d ",cqueue_arr[front_pos]);
                  front_pos++;
            }
      else //case of circular
      {
            while(front_pos <= MAX-1)
            {
                  printf("%d ",cqueue_arr[front_pos]);
                  front_pos++;
            }
//print front se leke end tak. Then make front 0, print wahan se rear tak.
            front_pos = 0;
            while(front_pos <= rear_pos)
            {
                  printf("%d ",cqueue_arr[front_pos]);
                  front_pos++;
            }
      }
```

Doubly ended queue/DEqueue

- Can add and delete on both sides
- Can be of two types: input restricted and output restricted
- Need two pointers: left and right
- left-> left position of queue
- right-> right position of queue

Input restricted

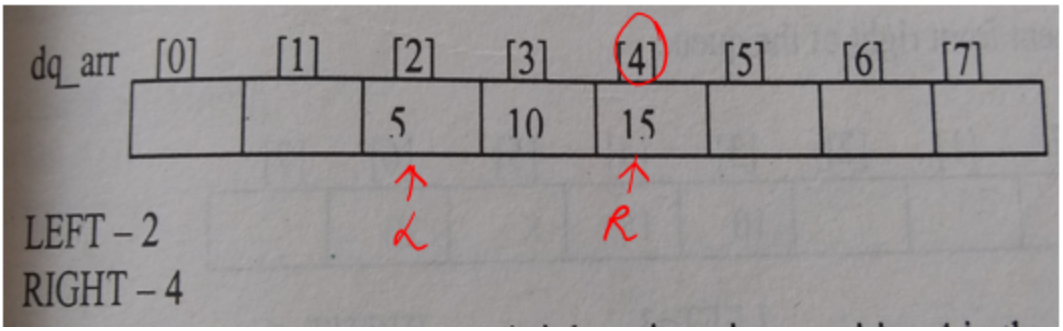
- Element is added at one end, but removed on both
- Two types: left input restriction and right (obvious definition)

Output restricted

- Element is removed at one end only, but added on both
- Two types: left output restriction and right (obvious definition

//should know working

Right input restricted dequeue mein can add only on right side but delete on both ends.



Here if you add an element, it gets added only at a[5]
Removal dono se hosakta

Insertion

Overflow:

```
if((left == 0 && right == MAX-1) || (left == right+1))
      {
            printf("Queue Overflow\n");
            return;
      }
```

left=right+1 condition is for circular array implementation

Initially empty

```
if (left == -1)  /* if queue is initially empty */
      {
            left = 0;
            right = 0;
      }
```

Circular effect:

If right == MAX-1 , then we set the values of right =0 add the element at the 0th position of the array. otherwise element will be added same as in simple queue, right=right+1.

Insertion at right

```
insert_right()
{
      int added_item;
```

```
	if((left == 0 && right == MAX-1) || (left == right+1))
	{
		printf("Queue Overflow\n");
		return;
	}
	if (left == -1)  /* if queue is initially empty */
	{
		left = 0;
		right = 0;
	}
	else
	if(right == MAX-1)  /*right is at last position of queue */
		right = 0;
	else
		right = right+1;
	printf("Input the element for adding in queue : ");
	scanf("%d", &added_item);
	deque_arr[right] = added_item ;
}
```

Insertion at left: circular effect

If left == 0 , then we set the values of left =MAX-1, add the element at the MAX-1 th position of the array. otherwise element will be added same as in left=left-1.

```
insert_left()
{
	int added_item;
	if((left == 0 && right == MAX-1) || (left == right+1))
	{
		printf("Queue Overflow \n");
		return;
	}
	if (left == -1)/*If queue is initially empty*/
	{
		left = 0;
		right = 0;
	}
	else
	{
	if(left== 0)
		left=MAX-1;
```

```
      else
            left=left-1;
      printf("Input the element for adding in queue : ");
      scanf("%d", &added_item);
      deque_arr[left] = added_item ;
}
}
```

Deletion at left

```
if(left==-1) underflow
if(left==right) only one element, delete and initialise left=right=-1
if left=max-1, set left=0
Else, delete and increase left.

Code for delete at left;

delete_left()
{
      if (left == -1)
      {
            printf("Queue Underflow\n");
            return ;
      }
      printf("Element deleted from queue is : %d\n",deque_arr[left]);

      if(left == right) /*Queue has only one element */
      {
            left = -1;
            right=-1;
      }
      else
            if(left == MAX-1)
                  left = 0;
            else
                  left = left+1;
}
```

Deletion at right

```
if(right==0)delete, r=n-1
if(r==l)r=-1,l=-1
Else r--;
```

Display

1. Check for underflow: if l=-1
2. Else run from front till rear if regular queue jaisa hai, no circular (front<=rear)
3. Else run from front to end, then 0 to rear.

Priority queue

Definition: Used for storing a set of elements based on a key value. The key value denotes priority of the element.

Determines the order of elements to exit the queue.
Higher priority = removed before
Fifo is tiebreaker if same priority

Generally,
Small number = high priority
Large number = low priority

Applications of Priority queue

- Used in CPU scheduling
  Higher priority = faster allocation
- Djikstra's algo
- All queue applications where priority is involved

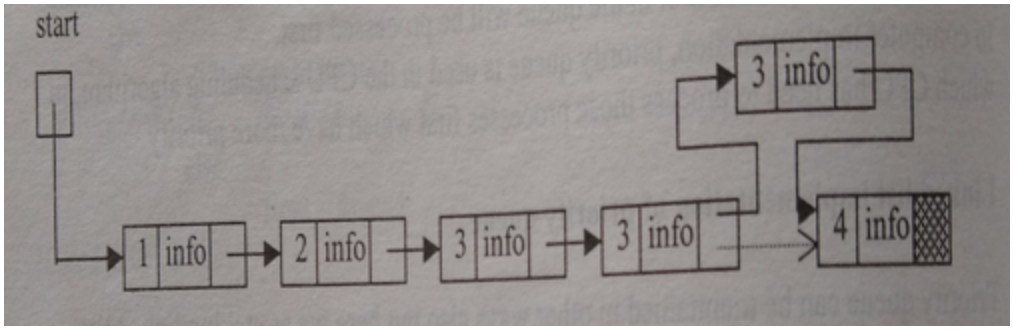
Application of PQ by Linked List

```
struct pq{
      int priority;
      int data;
      struct pq *link;
}
```

Operations

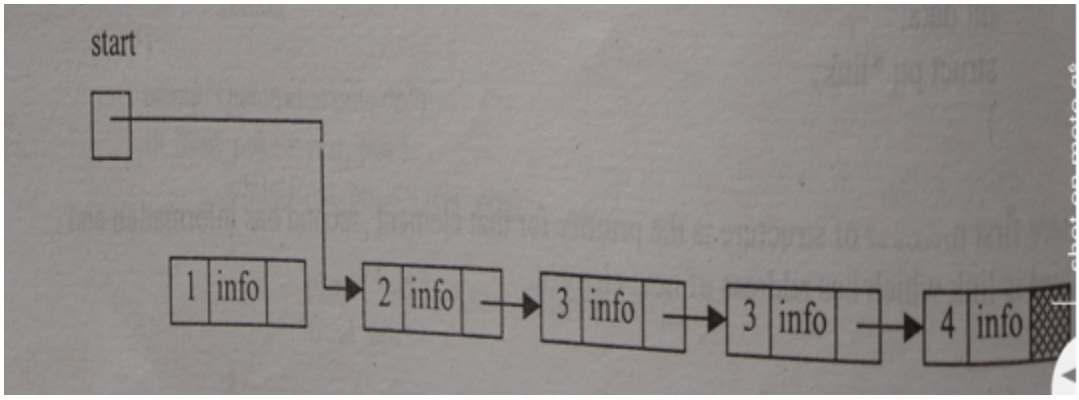
Add

Same as insertion in linked list.
Insert new element based on priority. Traverse till priority <= my priority, then add when a higher is found.



Delete

Delete the first element always. Always has higher priority.



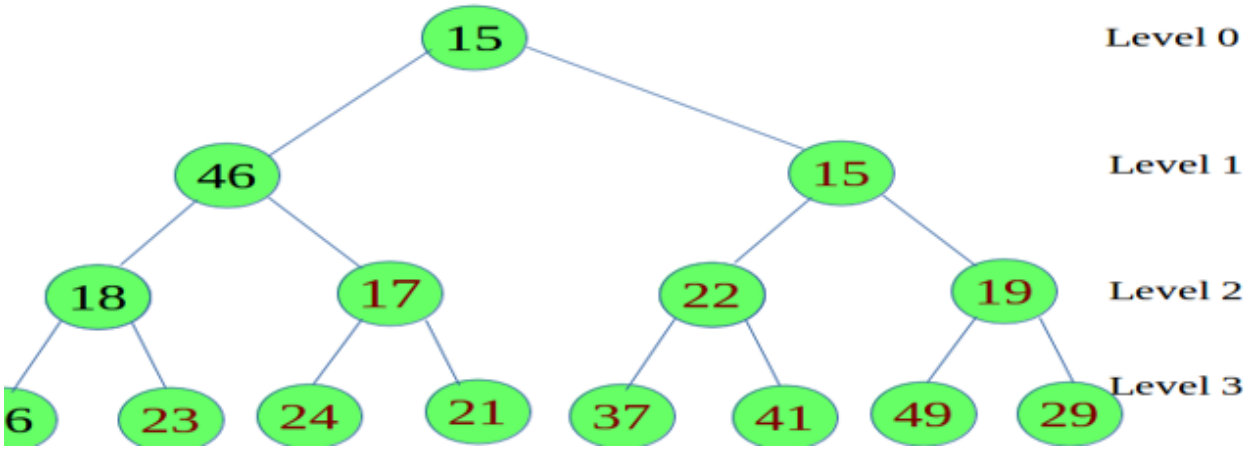
—----------over.

Module 3

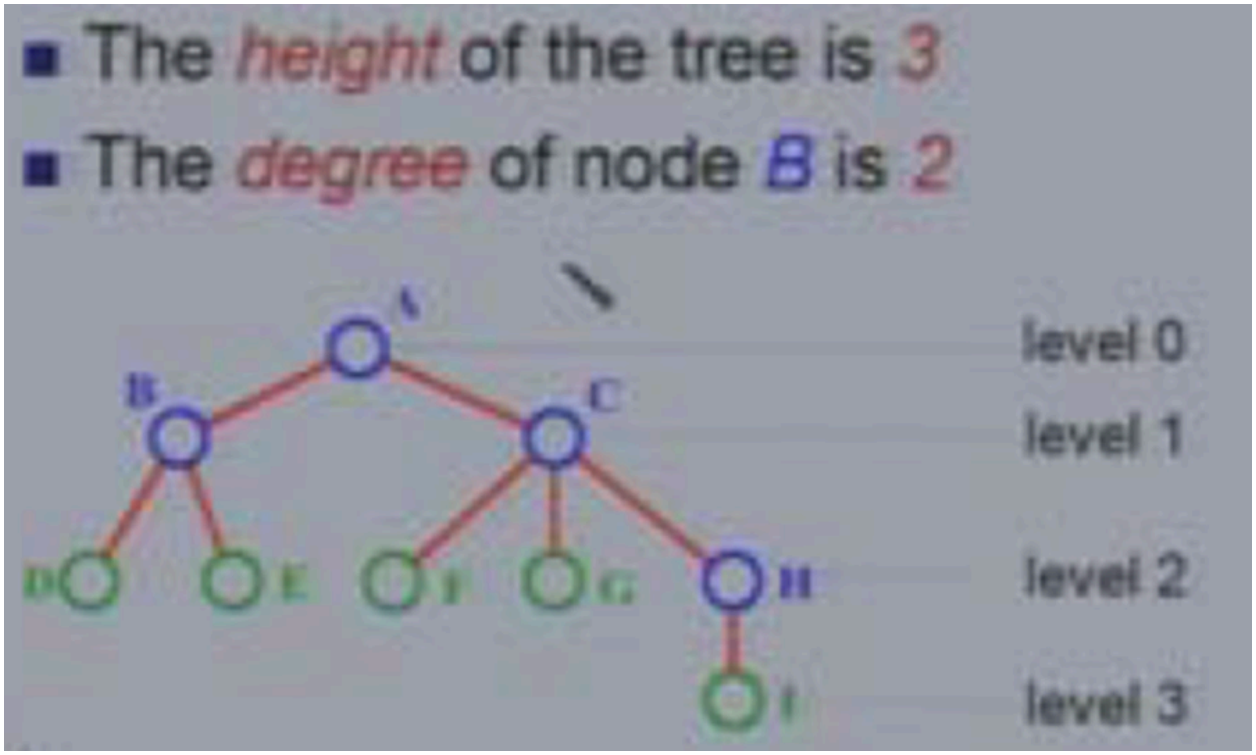
Trees

Level-

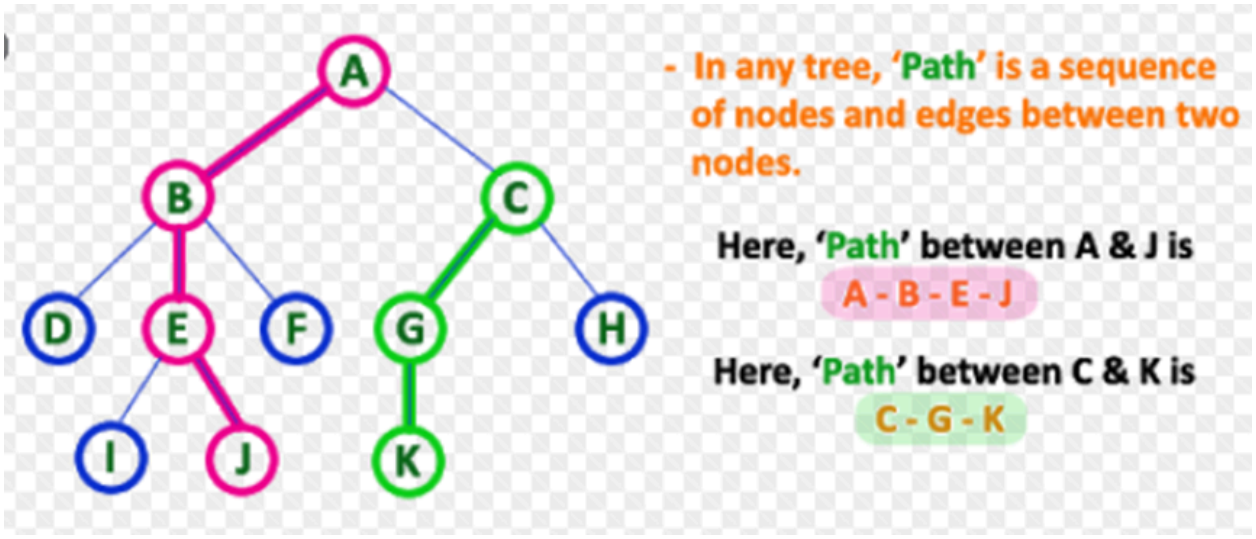
Root Node is always at Level 0
Its immediate successors are Level 1,
Their Successor at level 2 and so on
If a node is at Level n then its children will be at Level n+1

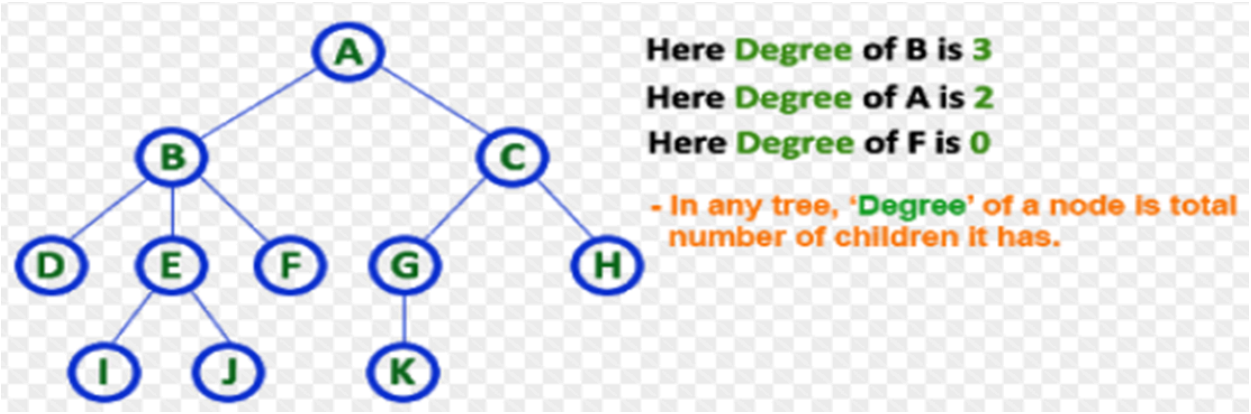


Height and degree



Degree = max number of children a node has
Height = max level





Degree of entire tree = highest degree among all its nodes

Terminal Node/Leaf

Any node whose degree is 0

Non-Terminal Node

Any node whose degree is Non-Zero

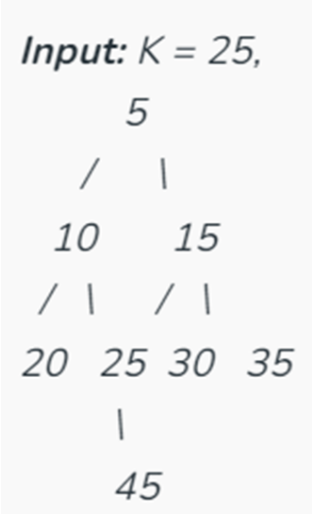
Forest

Set of Disjoint trees. If you remove the Root node, it becomes forest

Height

Height of a node: neeche se max kitne nodes hain (in a single line) excluding itself.
Height of 10: 2
Height of 15 = 1
Height of 30 = 0
Height of 5 = 3

In other words, longest path kya ho sakta hai for a node if you can only go down.



Depth

depth of a node: upar se max kitne nodes hain excluding itself.

depth of 10: 1
depth of 15 = 1
depth of 30 = 2
depth of 5 = 0
In other words, longest path kya ho sakta hai for a node if you can only go up?

Construction
Traversal

Types of Binary Trees

Strictly Binary Tree

1. Can ONLY have 2 children or no children. Single children not allowed
2. Also called full binary tree

Complete Binary Tree

1. All nodes have exactly 2 children
2. No of nodes at level x = 2^{x+1}-1

Structure of Binary tree

Linked list

```
struct node
{
      int num;
      struct node *left;
      struct node *right;
};
```

Preorder traversal

```
void preorder(struct node *tree)
{
      if(tree!=NULL)
        {
            printf("%d\n",tree->num);  //print node
            preorder(tree->left); // print left
            preorder(tree->right); // print right
        }
}
```

Inorder

```
void inorder(struct node *tree)
{
      if(tree!=NULL)
          {
            inorder(tree->left);
            printf("%d\n",tree->num);
            inorder(tree->right);
          }
}
```

Postorder

```
void postorder(struct node *tree)
{
      if(tree!=NULL)
        {
            postorder(tree->left);
            postorder(tree->right);
            printf("%d\n",tree->num);
        }

}
```

Binary Search Tree

Characteristics

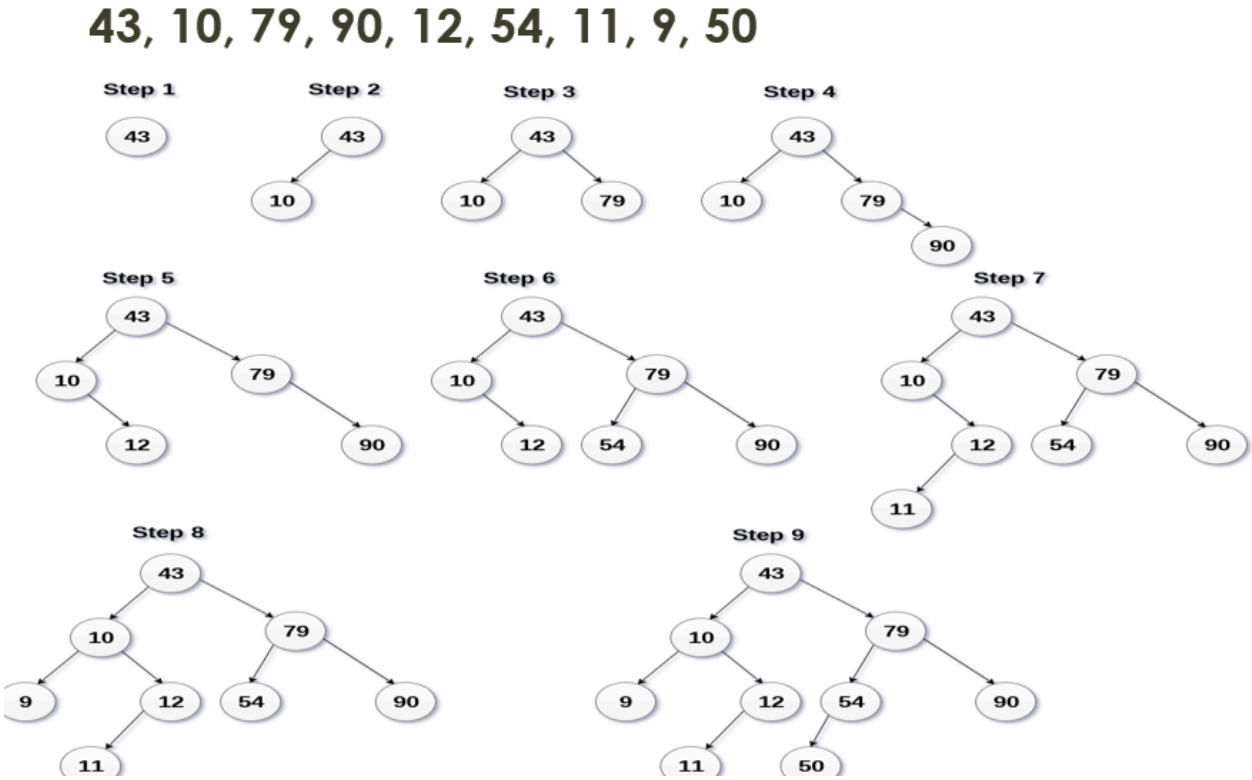
1. Value in left child is less than root
2. Value in right child is more than root
3. No duplicate nodes

4. In order traversal always gives sorted list
5. Traversal in BST = traversal in binary tree

Why do we need

Fast search, minimum, maximum element

Construction



Deletion in BST

Case 1:
Node is a leaf node

Algo:

1. Assign null to its parents' left pointer if the node is less than the parent
2. Assign null to its parents' right pointer if the node is more than the parent
3. Delete/free the node

Case 2:
Node has one child

1. To be deleted node=temp
2. Parent of temp -> child = temp -> child
3. Free temp

Case 3:
Node has 2 children
Let this node=q, predecessor=p

1. Find the predecessor of this element
2. Copy p->info into q->info.
3. Replace p with its child and delete p
   q-> child = p-> child
4. end

Implementation

```
struct node
{
      int num;
      struct node *left;
      struct node *right;
};
```

Insertion

```
struct node *insert(struct node *tree,int digit)
{
      if(tree==NULL) //tree is empty
        {
            tree = (struct node *)malloc(sizeof(struct node));
            tree->left = tree->right = NULL;
            tree->num = digit;
        }
     else
        if(digit < tree->num)
            tree->left=insert(tree->left,digit);
     else
        if(digit>tree->num)
            tree->right=insert(tree->right,digit);
     else if(digit==tree->num)
          {
             printf("Duplicate node:program exited");
             exit(0);
          }
```

```
          return(tree);
}
```

Algorithm:

1. Start
2. If the tree is empty (the current node is a leaf node)
   a. Allocate memory to tree
   b. Set its left and right pointers to NULL
   c. Set its value to value entered
3. If the number to be entered is greater than the current node, call the function again by passing only the right subtree of this node.
4. If the number to be entered is less than the current node, call the function again by passing only the left subtree of this node.
5. If the number is equal to current node, we have a duplicate, which is not allowed. Exit the program.
6. End

Search

```
void search(struct node *tree,int digit)
{
      if(tree==NULL)
         printf("The number does not exist.\n");
   else
      if(digit==tree->num)
      printf("Number=%d\n" ,digit);
   else
      if(digit<tree->num)
         search(tree->left,digit);
   else
        search(tree->right,digit);
}
```

Algorithm:

1. If the tree is null (we reached a leaf node), return exit statement.
2. If the search value is MORE than the current node, we have to check its right subtree. Call the function again passing only the right subtree.
3. If the search value is LESS than current node, we have to search its left subtree. Call the function again by passing only the left subtree.
4. If the value is equal to search value, print a positive statement and exit.

Graphs

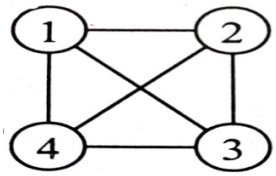
What:

1. Collection of two sets: set of vertices and set of edges
2. Edge: line connecting two nodes / vertices

Types:

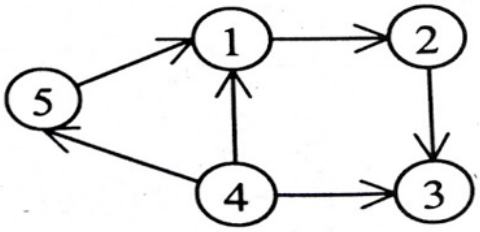
1. Directed
2. Undirected

Undirected



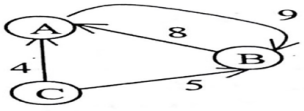
1. Edges don't have a direction
2. Graph has an unordered pair of vertices

Directed



1. Edges have a direction
2. Graph has ordered pair of vertices
3. Also called diagraph

Weighted graph



1. When edges have a non-negative value associated, the graph becomes weighted
2. A weighted graph is called a network

Terminologies

Adjacent nodes

- Nodes are adjacent if they share a edge
- In an undirected graph, (v0,v1) is an edge then v0 is adjacent ***to*** v1 and v1 is adjacent ***to*** v0.
- In a digraph if <v0,v1> is an edge then v0 is adjacent ***to*** v1 and v1 is adjacent ***from*** v0.

Incidence

- edge(a,b) is incident on nodes a,b in an undirected graph
- edge<a,b> is incident from a and incident to b in a digraph

Path

- A path from node a to b is a sequence of all nodes such that each consecutive node is adjacent to each other.
- Length of path: total number of nodes in that path

Closed path

- If last node = first node in a path, the path is closed

- Simple path: if all nodes occur just once in the path (except the first and last nodes, which can be repeated in case of a closed path), the path is simple
- Cycle: a simple, closed path

Cyclic graph

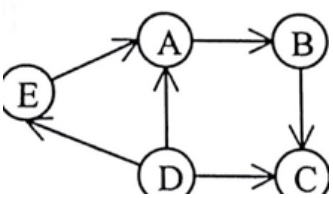
A graph that has cycles = cyclic graph

Acyclic graph

A graph that has no cycle = acyclic graph

Dag: directed acyclic graph

A directed graph that does not have any cycle. No node can reach itself using any possible path.



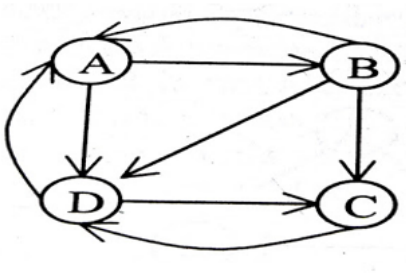
Degree:

- Number of edges in a node
- In the above graph, degree of A is three and degree of E is 2.
- In an undirected graph, degree is simply number of edges
- A directed graph has two degrees: indegree and outdegree
  Indegree: number of edges coming into that node
  Outdegree: number of edges leaving that node
  In the above example, indegree of A=2 and outdegree=1, total=3

Representation

Adjacency matrix

An adjacency matrix helps us keep track of adjacent nodes in a graph
Arr[i][j] = 1 If there is an edge from node i to node j, 0 if there's no edge from node i to node j



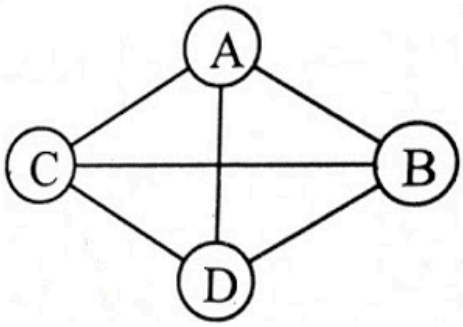
	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	0	0	1
D	1	0	1	0

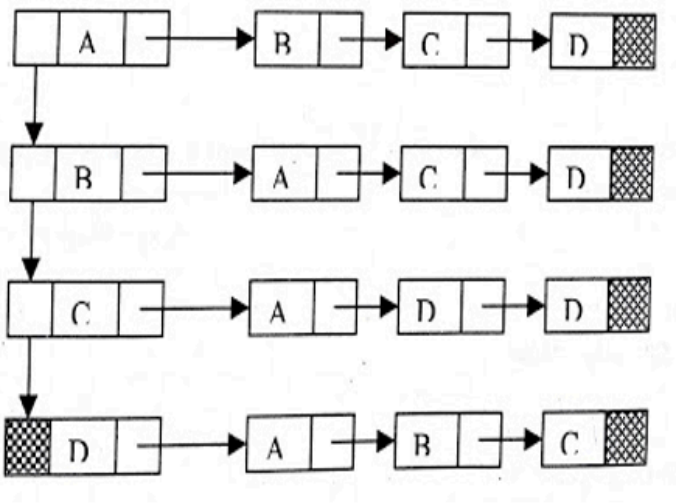
In a weighted graph, Arr[i][j] = weight of the edge connecting nodes i and j if edge exists, 0 otherwise

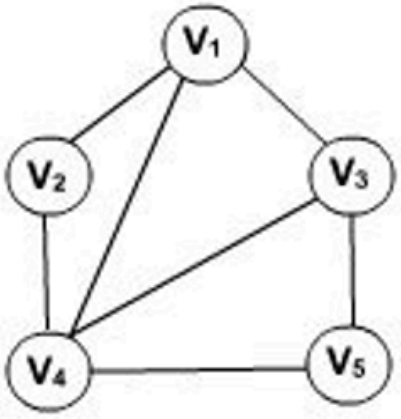
Adjacency list

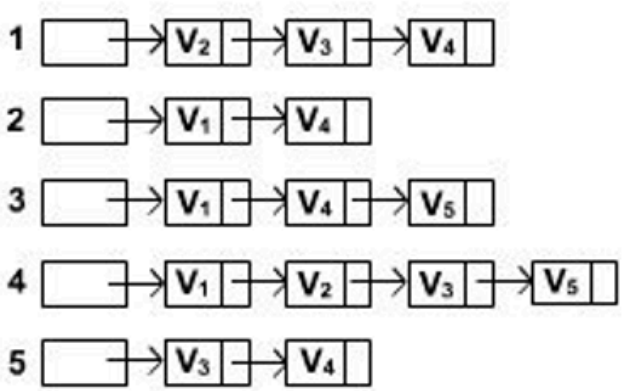
1. Two lists
2. One to hold all nodes
3. One to hold list of adjacent nodes of all nodes

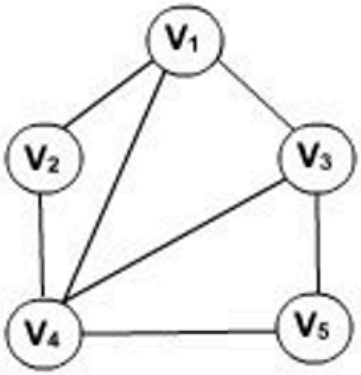
Suppose there are n nodes.
Create one list which will keep information of all nodes in the graph.
Create n lists, where each list will keep information of all adjacent nodes of that particular node.
Each list has a header node, which will be the corresponding node in the first list.











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

Implementation

Header node structure

```
struct node{
	struct node *next;
	char name;
	struct edge *adj;
};
```

Edge structure

```
struct edge{
	struct edge *link;
	char dest;
```

};

Insertion in Adj Matrix

Node

1. Insert one row and one column
2. Make all its entries=0

Edge

1. Change value from 0 to 1 for all adjacent nodes

Deletion

Node

1. Just delete that element’s corresponding row and column

Edge

1. Change value from 1 to 0 for all adjacent nodes

Insertion in Adj List

Node

1. Insert that node in the header nodes of the adjacency list.

Edge

1. Requires add operation in the list of the starting node of edge.
2. For example, we need to add an edge from B to C, go to B in the header nodes and add C to B’s list’s end.

Deletion

Node

1. Deletion of the entire list associated with this header node

Edge

1. Go to the header node in the header list
2. Then traverse the node’s list to find the 2nd node associated and delete

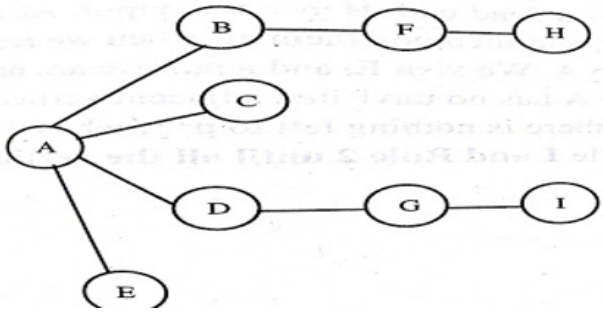
Graph traversal

Depth First Search

Distance of a node from starting node is called depth
Algorithm:

1. Pick the starting node, mark it visited and push it into the stack. Add the traversal in a list or array.
2. While possible, visit unvisited adjacent node, add to list, mark visited, push into stack.
3. If all nodes are visited, pop this vertex from stack
4. Repeat 2 and 3

IMPORTANT: dfs backtracks only when met with deadend



Event	Stack	DFS
Visit A	A	A
Visit B	AB	AB
Visit F	ABF	ABF
Visit H	ABFH	ABFH
Pop H	ABF	
Pop F	AB	
Pop B	A	
Visit C	AC	ABFHC
Pop C	A	
Visit D	AD	ABFHCD
Visit G	ADG	ABFHCDG
Visit I	ADGI	ABFHCDGI
Pop I	ADG	
Pop G	AD	
Pop D	A	
Visit E	AE	AB F HCDGIE
Pop E	A	
Pop A		
Done		

Traversal when starting at A: ABFHCDGIE
Working:

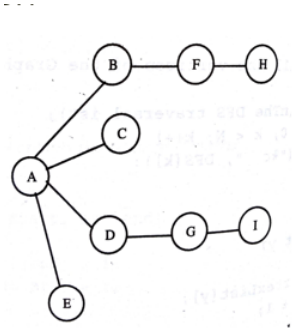
- Visit each element starting from the root node OUTWARDS.
- Example: Starting A, go to node B and travel outwards till end: H
  Since there are no nodes, come back one node at a time and see if there are more branches to it
- Since there are no branches till A, come back till A and go to next branch
- Repeat for all branches of A.

Breadth first search

Algorithm:

1. Pick starting node, mark it visited and enqueue it. Add the traversal in a list or array.
2. If possible, visit all of the front of queue's unvisited adj nodes, add traversal and enqueue
3. If all nodes are visited, dequeue

4. Repeat 2 and 3

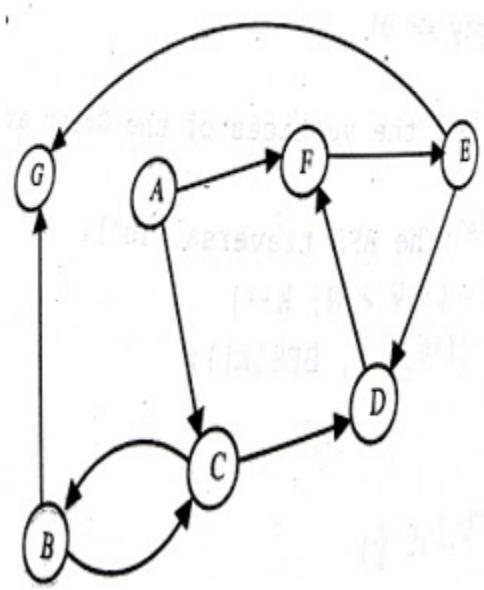


Traversal: ABCDEFHGI

Event	Queue (Front to Rear)	BFS
Visit A		A
Visit B	B	AB
Visit C	B C	ABC
Visit D	BCD	ABCD
Visit E	BCDE	ABCDE
Remove B	CDE	
Visit F	CDEF	ABCDEF
Remove C	DEF	
Remove D	EF	
Visit G	EFG	ABCDEFG
Remove E	FG	
Remove F	G	
Visit H	GH	ABCDEFGH
Remove G	H	
Visit I	HI	ABCDEFGHI
Remove H	I	
Remove I		
Done		

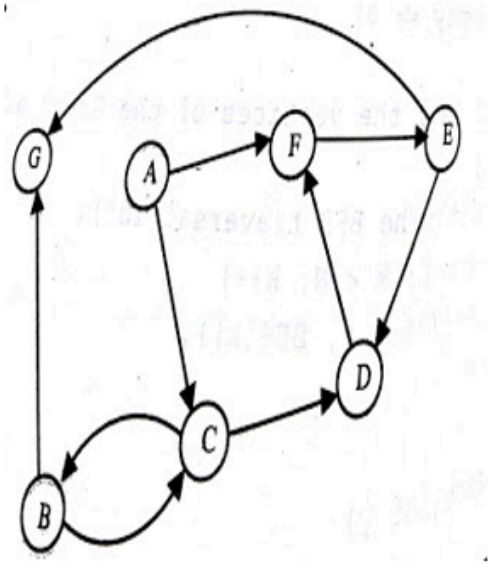
In a directed graph or digraph, a node is included in the traversal iff it goes from parent to child.

DFS



Event	Stack	DFS
Visit A	A	A
Visit C	AC	AC
Visit B	ACB	ACB
Visit G	ACBG	ACBG
Pop G	ACB	
Pop B	AC	
Visit D	ACD	ACBGD
Visit F	ACDF	ACBGDF
Visit E	ACDFE	ACBGDFE
Pop E	ACDF	
Pop F	ACD	
Pop D	AC	
Pop C	A	
Pop A	NULL	

BFS:



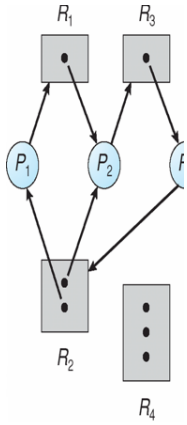
Event	Queue (Front to Rear)	BFS
Visit A		A
Visit C	C	AC
Visit F	CF	ACF
Remove C	F	
Visit B	FB	ACFB
Visit D	FBD	ACFBD
Remove F	BD	
Visit E	BDE	ACFBDE
Remove B	DE	
Visit G	DEG	ACFBDEG
Remove D	EG	
Remove E	G	
Remove G		
Done		

Applications

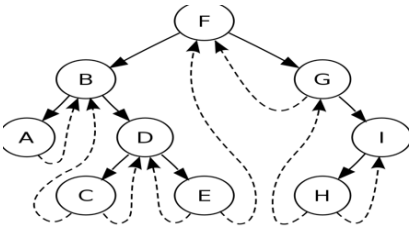
1. Roadmap
2. Map of airlines
3. Layout of games
4. Links between webpages
5. Diagrams of flow capacities and relationships in algos and networks

In google maps, places are vertices and roads are edges. The shortest distance is calculated by their navigation algorithm.

If there is a cycle in a graph, a deadlock is created. Here, two cycles are created and thus, there are two deadlocks.



Threaded binary tree



In a regular binary three, the leaf nodes have NULL pointers as they have no children. In a threaded binary tree, the leaf nodes' null pointers point to the node just one level higher to itself on both sides.
These pointers are now called threads.

A binary tree with threads is called a threaded binary tree.

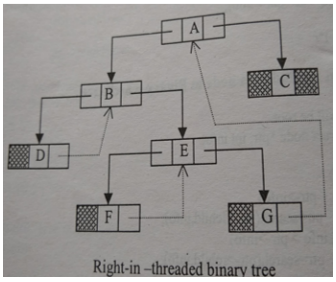
In-order threading

One way

Each right NULL pointer is changed to a thread to the node's inorder successor

Two way

Each right NULL pointer is changed to a thread to the node's inorder successor
And each left NULL pointer is changed to a thread to the node's inorder predecessor



Right-in –threaded binary tree

Right in-threaded tree:

1. Each null pointer should connect to that node's inorder successor

Traversal for image: DBFEGAC

Left in-threaded tree:

1. Each null pointer should connect to that node's inorder predecessor
2. In the diagram attached, F's left null pointer should go to B for left-in threaded binary tree.

Fully in-threaded tree:

1. The left thread of a node is connected to its in-order predecessor and right thread is connected to the successor node.
   In the diagram, F's left null ptr goes to B and right goes to E

Preorder Threading

Everything is the same, just predecessor and successor is decided based on preorder traversal.

Structure

```
struct node
    {
    node *left_ptr;
    bool left;
    int info;
    node *right_ptr;
    bool right;
};
```

The boolean left and right are to differentiate between thread and link.

The variable “left”

- Stores 0 if left_ptr is a normal left child.
- Stores 1 if left_ptr is a thread (points to inorder predecessor).

The variable “right”

- Stores 0 if right_ptr is a normal right child.
- Stores 1 if right_ptr is a thread (points to inorder successor).

So:
left = 1 → left_ptr is a thread
right = 1 → right_ptr is a thread.

Enumeration

Data type consisting of a set of named values called elements, members or enumerators of the type.
Example

```
enum Direction { EAST, NORTH, WEST, SOUTH };
```

Here, EAST=0, NORTH=1 and so on

```
enum Direction { EAST, NORTH, WEST = 5, SOUTH };
```

But here, since we allocated west=5, south becomes 6 and so on.

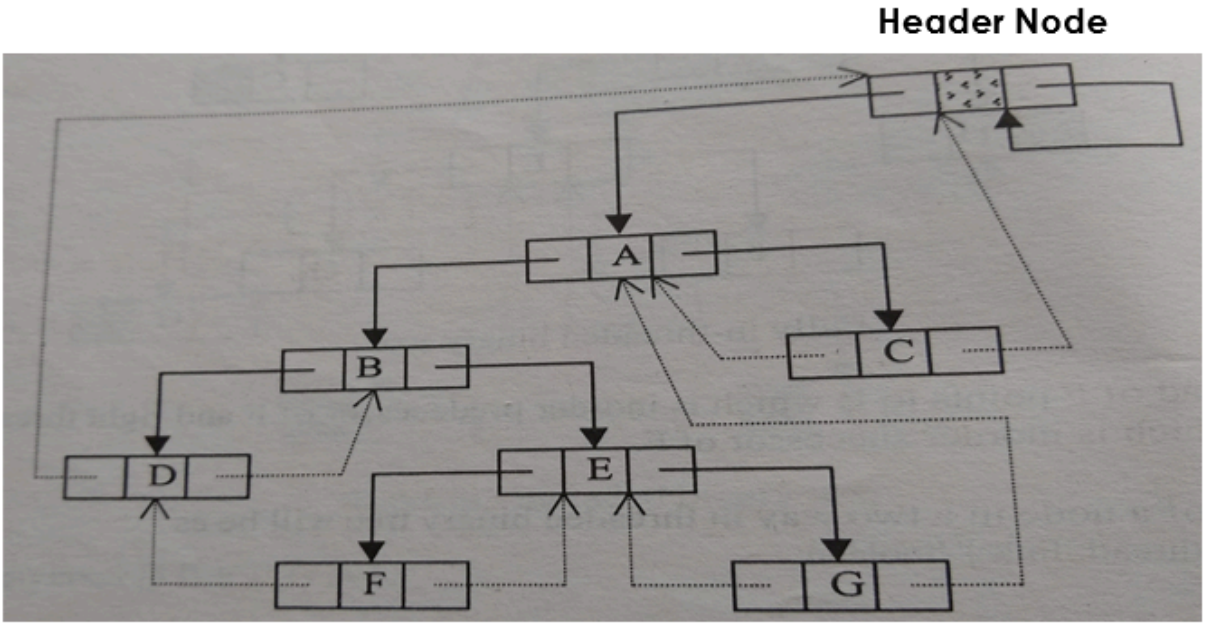
Example of Implementation:

```
#include <stdio.h>
#include <iostream>
using namespace std;
```

```
enum day {sunday, monday, tuesday, wednesday, thursday, friday, saturday};

int main()
{
    enum day d = thursday;
    cout << "The day number stored in d is " << d << endl;
    return 0;
}
```

Output will be 4. (enum is 0 indexed)
Here, elements sunday, monday etc are called enumerators.
enum is the keyword.
day is the enum variable.
d is the object created for day.

However, in threaded binary trees, the firstmost element still has no predecessor and the lastmost node has no successor. This implies there are still NULL values. The solution is, in-threaded trees with header nodes. A dummy node is taken. To this node's left, we create our tree. Left pointer of header node points to our tree's root. And the leftmost/rightmost nodes in traversal will not point to null, but to this node.



Condition for empty In-threaded Tree with Header Node-
head->lchild=head

AVL Trees

A binary search tree where height of left and right subtree of any node will be with maximum difference 1

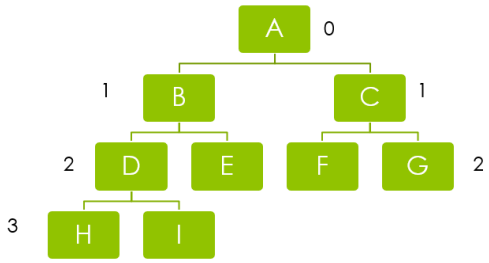
Each node has a balance factor:
Balance Factor = Height of Left subtree-Height of Right Subtree

Right heavy node:
1. Height of right subtree > height of left subtree

Left heavy node:
1. Height of left subtree > height of right subtree

Balanced
1. Equal height on both subtrees

For A, balance factor is 3-2=1
It is left heavy



Insertion

1. Insert the node at the proper place using same procedure as in BST
2. Calculate the balance factors of all the nodes on the path starting from the inserted node to the root node.
3. If the tree is balanced then exit.
4. If absolute value of balanced factor of any node in this path >1 then the tree becomes unbalanced.
5. The node which is nearest to the inserted node & has absolute value of balance factor >1 is marked as Pivot node
6. We perform rotations about the pivot node.

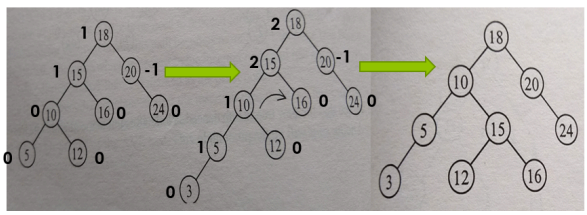
https://visualgo.net/en/bst for visualization.

Avl rotations: C to S (child to subtree)

Pivot node: the first node where imbalance occurs.

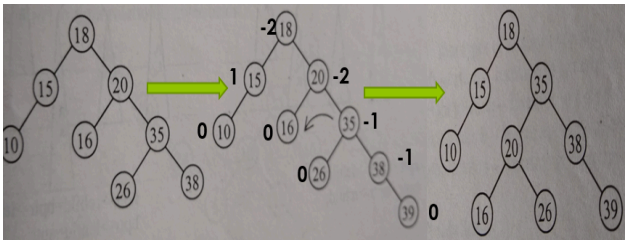
Left to left

Inserted: 3



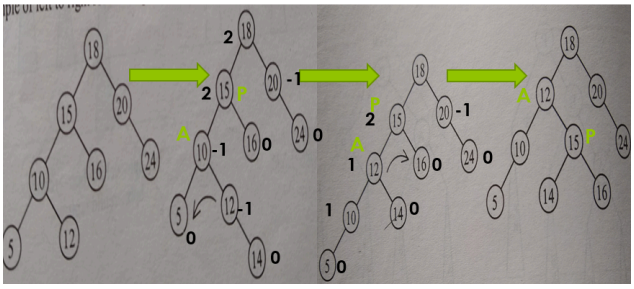
When new node is inserted in left subtree of the left child of pivot node.
After inserting 3, the balance factor increased. Thus, CW rotation about 10

Right to right



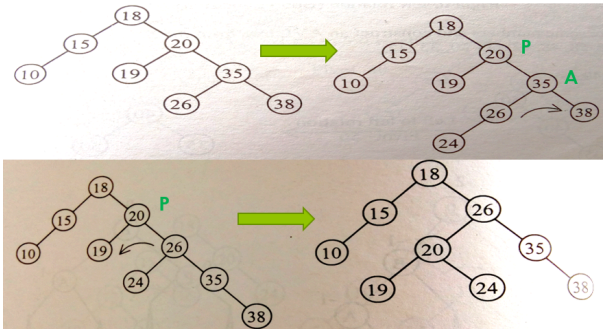
Inserted: 39
When new node is inserted in right subtree of the right child of pivot node.
Rotate ACW
Here, after inserting 39, we have an imbalance
ACW rotation about 35 fixes the balance factor.

Left to right



Inserted: 14
When the new node is inserted in the right subtree of the left child of the pivot node.
ROTATE ACW FIRST, THEN CW
Inserted: 14
15 is pivot, notice how 14 is added to the right of its left subtree?

Right to left

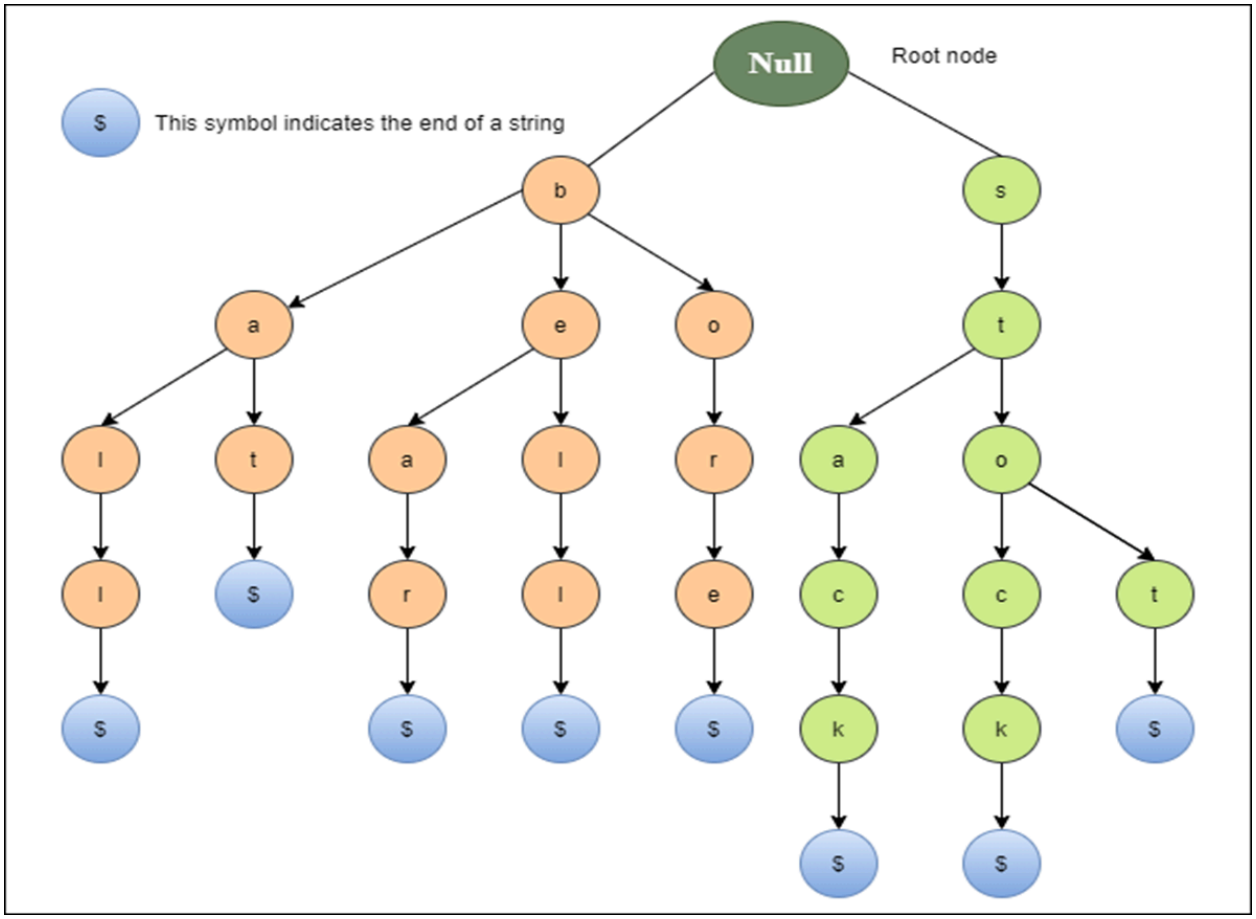


Inserted: 24
When the new node is inserted in the left of the right subtree of the pivot node.
Rotate CW FIRST THEN ACW
Pivot = 20
24 is inserted to 20’s right child’s left subtree.

Trie

Trie is a sorted tree-based data-structure that stores the set of strings
Root is always null
Each child is sorted alphabetically
Each node can have max 26 children

Example



The dollar sign $ acts as the end of the word

Applications

1. Spell checker
2. Autocomplete
3. Browser history (used to complete URL on your browser based on history)

As you start typing, the system starts searching and traversing alphabetically and then gives suggestions based on the branches

Advantage

1. Find strings in o(n) time
2. Faster than BST
3. Can easily print all words in sorted order
4. Efficient prefix searching

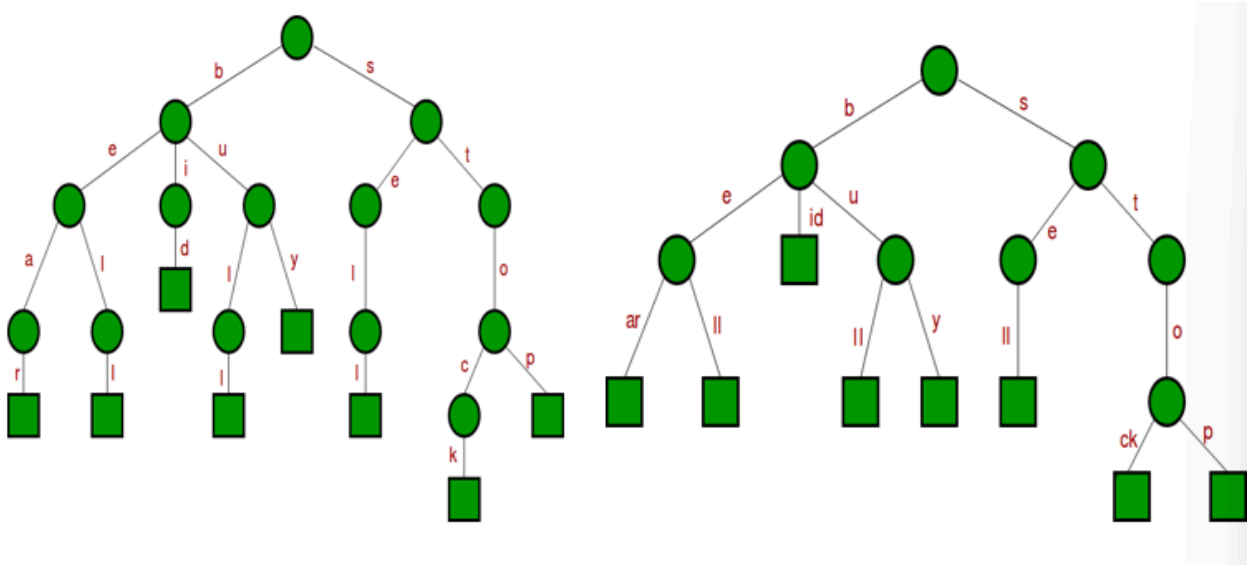
Disadvantage

1. Need a lot of memory because of so many pointers

Suffix tree

A compressed trie for all suffixes of a text
Example:
bo can have several suffixes:
Boy, box, bot, bother, bottom, boston etc

A suffix tree compress trie is obtained from a standard trie by joining chains of nodes. When there's no branching from a node down, we can compress them to be denoted in a single node.



Josephus problem: https://www.youtube.com/watch?v=uCsD3ZGzMgE
Application of Queue in josephus problem

- Store people in a circular queue.
- For each count: Dequeue front person.
- Enqueue back if not eliminated (for first k−1 steps).
- At the k-th step, simply dequeue and discard.

Module 4

Map

Definition: Any data structure that groups a dynamic number of key-value pairs.

Declaration:
map<char, int> mp; [here, char is the datatype of key and int is the datatype of value]

Helps us
1. Retrieve values by key
2. Insert new pairs
3. Update value associated with key

It works like the array. The only difference is that in an array, we can only access elements by its integer index but in maps, the index can be any datatype.

map[char]=string etc

Here, char is the key type and string is the value type. map[‘c’]= “charisma”, here ‘c’ is the key and value is charisma

Two values cannot have the same key. One map cannot have duplicate keys.

Functions

- begin() – Returns an iterator to the first element in the map
  Usage: auto it = mp.begin();
- end() – Returns an iterator to the pointer **AFTER** the last element in the map
  Usage: auto it = mp.end();
- size() – Returns the number of elements in the map
  int s = mp.size();
- max_size() – Returns the maximum number of elements that the map can hold
  int max = mp.max_size();
- empty() – Returns whether the map is empty
  bool isEmpty = mp.empty();
  or while(!mp.empty()){ //do something };
- pair insert(keyvalue, mapvalue) – Adds a new element to the map
  mp.insert({key,value});

- erase(iterator position) – Removes the element at the position pointed by the iterator
  auto it = mp.begin()+5; //5th element in the map
  mp.erase(it); //erase value at it
- clear() – Removes all the elements from the map
  mp.clear();

Accessing elements:

```
#include <stdio.h>
#include <iostream>
#include <map>
using namespace std;


int main()
{
    map<char, int> mp;
    for(int i=0;i<26;i++){
        mp.insert({'a'+i,i+1}); //stores alphabet plus its index
    }

    mp.insert({'a',34}); //this does nothing since a already has a
value

    for(auto [key,value]:mp){
        cout << key << " " << value << endl; //print elements
    }

    for(auto it:mp){
        cout << it.first << " " << it.second << endl;
    }
    return 0;
}
```

```
a 1
b 2
c 3
d 4
e 5
f 6
g 7
h 8
i 9
j 10
k 11
l 12
m 13
n 14
```

mp.size() for this = 26
mp.erase('a') will erase the value of a and remove it from the map.

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

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


int main()
{
    map<char, int> mp;
    for(int i=0;i<5;i++){
        mp.insert({'a'+i,i+1}); //stores alphabet plus its index
    }

    mp.erase('a'); //value erased
    mp.insert({'a',34}); //this gets run now
    for(auto [key,value]:mp){
        cout << key << " " << value << endl; //print elements
    }
}
```

```
a 34
b 2
c 3
d 4
e 5
```

Map implementation

1. Array
2. Linked list of pairs (slow o(n) and insufficient)
3. Hash table(fast o(1))
   Uses a hash table as internal container
4. Binary search tree (fast o(log n), but keys will be ordered)

Sets, Maps, Dictionary

Set

Collection of elements without duplicates

Set adt
Fundamental functions on set S:

1. union(a) = replace S with union of S and A
2. intersect(a) = replace S with intersection of S and A
3. subtract(a) = replace S with difference of S and A
4. insert(e) = add element e to S
5. find(e) = if S contains e, return iterator pointing to it else return end
6. erase(e)= find e in S and remove it
7. begin() = return iterator to beginning of set
8. end() = return iterator to after the ending of set

STL functions

1. end()
2. begin()
3. size(): returns number of elements in set
4. max_size(): returns max number of elements set can hold
5. empty()
6. rbegin(): returns reverse iterator pointing to last container in set
7. rend(): returns reverse iterator pointing to first container in set
8. crbegin(): returns constant iterator pointing to last element.
9. crend(): returns constant iterator pointing to first element.

…

Disjoint sets and partitions

A and B are disjoint partitions of S if

1. A union B = S
2. A intersection B = NULL
3. Example: S={1,2,3,4} A = {1,2} and B={3,4}
   If A = {1} or A={1,2,3}, then it is not a valid disjoint partition

Union creates disjoint subsets and find checks if they’re connected

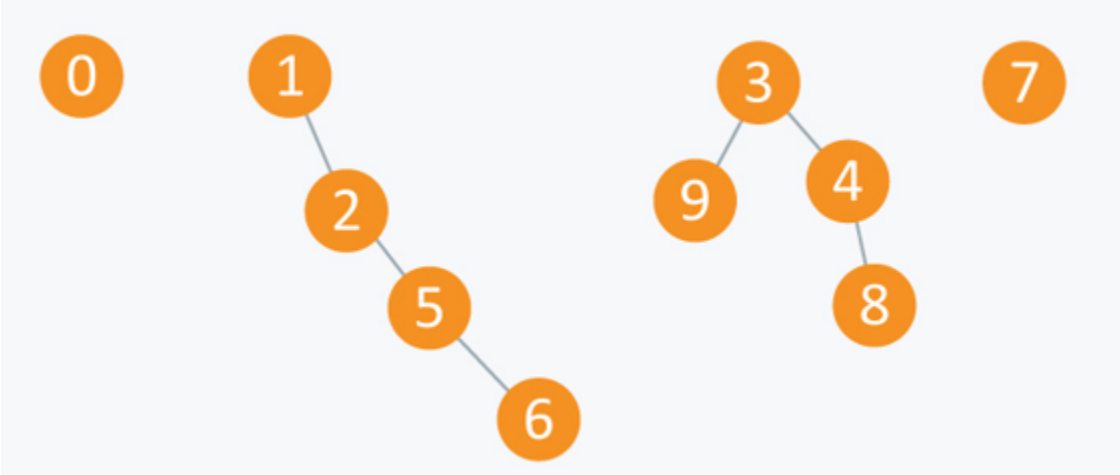
Example
S = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

N = 10
Originally, visualise set like this

Perform:

1) Union(2, 1)
2) Union(4, 3)
3) Union(8, 4)
4) Union(9, 3)
5) Union(6, 5)
6) Union(5, 2)

After all perform operations, set looks like this



Perform:

1) Find(6,1) : are 6 and 1 connected to same root? Yes. return true
2) Find(8,9): return true
3) Find(7,1): return false

Applications

1. Elections
2. Classification
3. Pattern matching
4. Mutually exclusive processes

Dictionary

Definition: A dictionary allows for keys and values to be of any

object type.
Unlike Map, a dictionary allows for multiple entries to have the same key.

Adt

- size(): returns size
- empty()
- find(k): returns first instance of k, else return end
- findAll(k): return a pair of iterators of all instances of k
- insert(k,v): insert k:v pair
- erase(k): erase by key
- erase(p): erase by iterator
- Begin and end

Implementations

1. Unordered list
2. Hashtable with separate chaining
3. Ordered search table

Module 5

Hashing

Searching

Searching is the process of retrieving values associated with our search keys.
There are two main types of searching techniques

1. Linear search: Also called sequential search. Simple, o(n) time.
   Begins at one end.
   Goes thru each element one by one until we find element / reach end
   If found, it returns location else null

   Algo:
   - Set pos=-1
   - Set i=1
   - Go from i=1 to n
     - If a[i]= search val, set pos=i+1 and return pos
   - If pos=-1 after loop, print unavailable
   - End

   Linear search is used mainly when dataset is unordered

2. Binary search
   Works efficiently with sorted list
   At each iteration, searching space gets reduced by half

   Algo
   - Set low=0
   - Set high =n
   - While low<=high
     - Set mid = low+high/2
     - If a[mid] = val, pos=mid return pos
     - If a[mid] < low, low = mid+1
     - Else high = mid-1
   - If after loop, pos=-1, print unavailable
   - End

Hashing

- Allows retrieval and storage in O(1)
- Uses a hash function to map one value with a special key for fast access.

- Uses key to-address transformation
- For storing: key -> generate index -> store
- Hash table uses a random access data structure like an array and a map function called a hash function for implementation.

Hash function

1. Should be computable
2. Should uniformly distribute keys
3. Generate minimum collision waale address
4. A big number/string is mapped to a small integer that can be used as a key in the table
5. If h is the hash function and k is the key, h(k) is the index at which the record with key value k is stored.

If each key is mapped with a unique address, we get an ideal situation.
If the same hash table address is generated for different keys, we get a collision

A good function = minimum collisions

Truncation method

Uses a part of the key as address

Example

2358173, 218414, 128484
If size=100, we take rightmost 2 digits for address:
73, 14, and 84

Mid-square method

Squares the key, take some digits down the middle

Example
1337, 1273, 1391
Square: 1787569, 1620529, 1934881
Take 4,5 digit: 75, 05, 48 as key

Folding

Break the key in pieces, add to get address
Example
1378471, 2148628

137+84+71 = 292. Thus, key is 292
And so on

Suppose the size of the hashtable is 100, we truncate the key received. New key: 92

Modular method

Perform modulus operation, remainder is the address
Table size= prime number

Example:
82394561%97=45
87139465%%97=0
45 and 0 are the addresses

Closed Hashing

- Uses an array
- Techniques: linear, quadratic, double, rehashing

Linear probing

For insertion:

1. If two keys get the same address, send the second one to the next available address.
2. Hash table is used as a circular array: if the last cell receives collision, look for empty address from 1st index
3. If table size is N, after n-1, value is entered at 0

Example
H(29)=29%11=7
H(18)=18%11=7
H(18)=7+1=8
H(43)=43%11=10
H(10)=10%11=10
H(10)=10+1=11%11=0
H(36)=36%11=3
H(25)=25%11=3
H(25)=3+1=4
H(46)=46%11=2

0	10
1	
2	46
3	36
4	25
5	
6	
7	29
8	18
9	
10	43

For searching:

1. Check the hash address position in table
2. If not found, search linearly from this address

Disadvantage:

1. Clustering occurs. Clustering is when a long range of cells are occupied in a sequence.
2. In case of clusters, we need to traverse a longer range to find an empty position to insert/ to search

Quadratic Probing

Insertion:

1. If address is h and is occupied, look for location $(h+i^2)$ % size for i=1,2,3,4…
   i is updated for every cell checked for the same original address h.
2. Decreases clustering
3. Cannot search all locations

0	10
1	
2	46
3	54
4	
5	
6	
7	29
8	18
9	
10	43

Example:
H(29)=29%11=7
H(18)=18%11=7, collision h+(1*1)%size = 8
H(43)=43%11=10
H(10)=10%11=10, collision h+ 1*1 %size =11%size=11%11=0
H(46)=46%11=2
H(54)=54%11=10, collision h + (2*2) = h+4 = 14%11 = 3

Double Hashing

Insertion:

1. Choose a 2nd hash function
   H = key%size
   H’ = prime2 - key%prime2
   Prime2 is a prime LESS THAN THE TABLE SIZE
2. h =hash value from 1st function
3. h’=hash value from 2nd function
4. In case of collision, search h%size, h+h’ % size, h+2h’%size and so on

The two time calculation of function complicates the process
Searching is slower than linear and quadratic functions

0	
1	
2	
3	55
4	
5	
6	
7	
8	8
9	48
10	
11	
12	68

Example
Elements=8, 55, 48, 68
Table size=13

H(8)=8%13=8
H(55)=55%13=3
H(43)=48%13=9

H(10)=68%13=3, collision
Next address=
=(68%13)+(11-(68%11))%13
=(3+(11-2))%13
=(3+9)%13
=12%13
=12
Position for 68=12

Rehashing

- There's a chance of insertion failure when the table is full.
- Create a new hash table with the size double as the previous table
- Use the new hash function and insert the elements again into the new table with updated size.

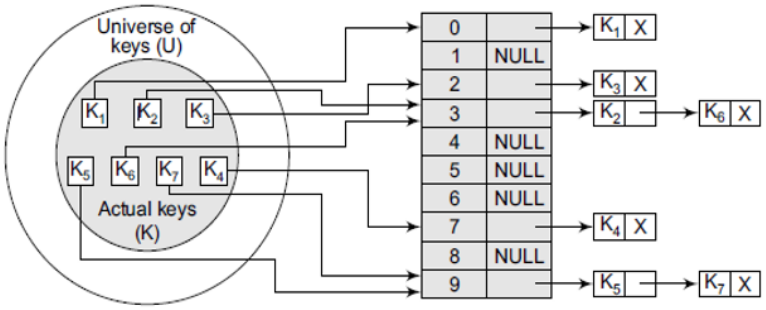
Open Hashing

Open hashing uses linked list for resolving collision.
Technique: **Separate chaining**
Maintains a chain of elements with same hash address
Maintain a linked list corresponding to the repeated address, store elements with same address in this linked list

Insertion

1. Get hash value thru hash function
2. Map hash value into the hash table position. Insert element to linked list
3. Same for searching

Separate Chaining



Advantage:

1. Simple
2. Never fills up
3. Less sensitive to hash func
4. Used when number of keys are unknown

Disadvantage:

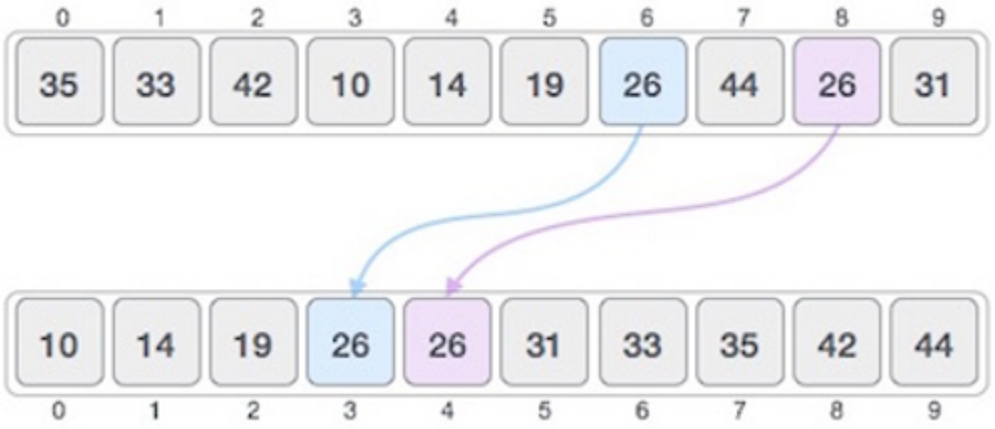
1. Cache performance is not as good as keys are stored in linked list
2. Space wastage
3. Long chain -> o(n) search time
4. Extra space needed for pointer

S.NO.	SEPARATE CHAINING	OPEN ADDRESSING
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care for to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

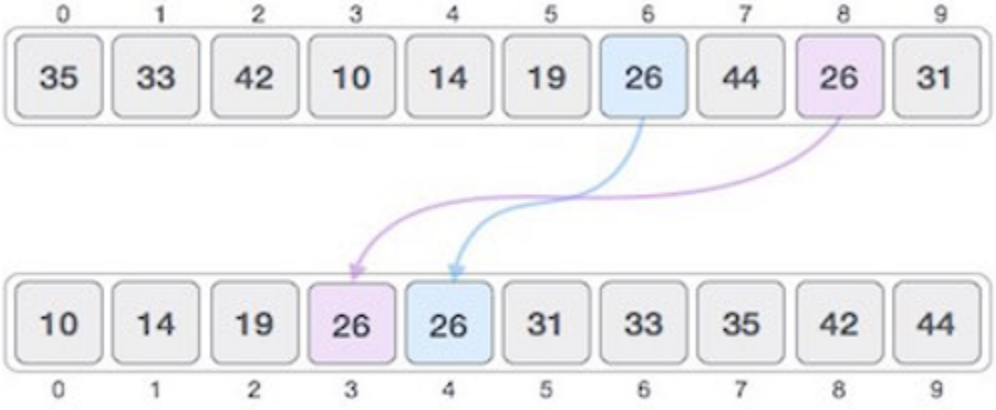
Sorting

- Arrange data in some order
- Sorting algorithm: specifies the way to arrange
- Makes searching easy and optimized
- Makes data readable
- Example: telephone directory, dictionary

Stable sorting:
When the same elements' relative order is same before and after sorting



When the same elements' relative order changes.



Stability is mainly important when we have key value pairs with duplicate keys possible (like people names as keys and their details as values). And we wish to sort these objects by keys.

Inplace and Not-Inplace

An in-place sorting does not need an extra space and produces an output in the same memory that contains the data by transforming the input 'in-place'.

A not-in-place sorting takes O(n) extra space and produces output there.

Which Sorting Algorithms are In-Place and which are not?
In Place: Bubble sort, Selection Sort, Insertion Sort, Heapsort.
Not In-Place: Merge Sort. Note that merge sort requires O(n) extra space

Bubble sort

- Compares adjacent element
- Swaps them if not intended order

Algorithm:

1. Pass 1:
    a. Compare a[1] and a[2]
    b. If intended order, continue
    c. Else exchange
    d. Similarly compare till a[n-1] and a[n]
2. Pass 2:
    a. Repeat the same starting a[1] but only till a[n-1]
    b. With each pass, the last index reduces

Not suitable.
Complexity: $O(n^2)$

Shell sort

//assume ascending
With each iteration, working space reduces.
Consider:

[33,31,40,8,12,17,25,42]
n=8

- Take k=2, interval: n/k = 4
- Compare 0th element to 4th, swap if needed
- Continue for 1,5; 2,6 and so on
- Array: [12,17,25,8,33,31,40,42]

Increase k, k=4

- Compare 0th and 2nd
- 1 and 3 and so on
- Swap if needed
- Array: [12,8,25,17,33,31,40,42]

Increase k=8,

- Compare 0 and 1, 1, 2 and 2, 3 and so on
- Array: [8, 12, 17, 25, 31, 33, 40, 42]

Practice Questions

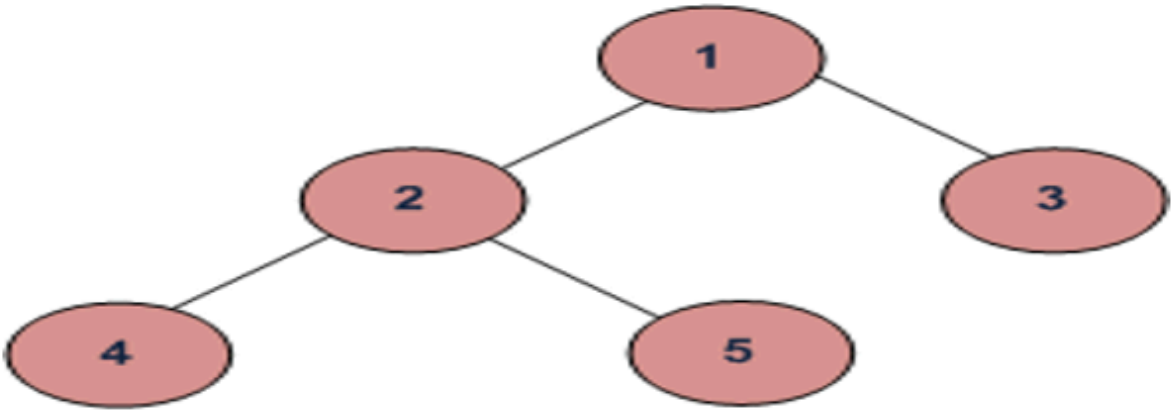
Pseudo/Algo

1. DFS
2. BFS
3. Insertion/deletion/search/traverse
    a. In BST
    b. In doubly linked list, circular linked list
    c. In double ended queue, in queue (both implementations)
    d. In priority queue (both implementations)
    e. In stack (both implementations)
    f. In AVL (algo only)

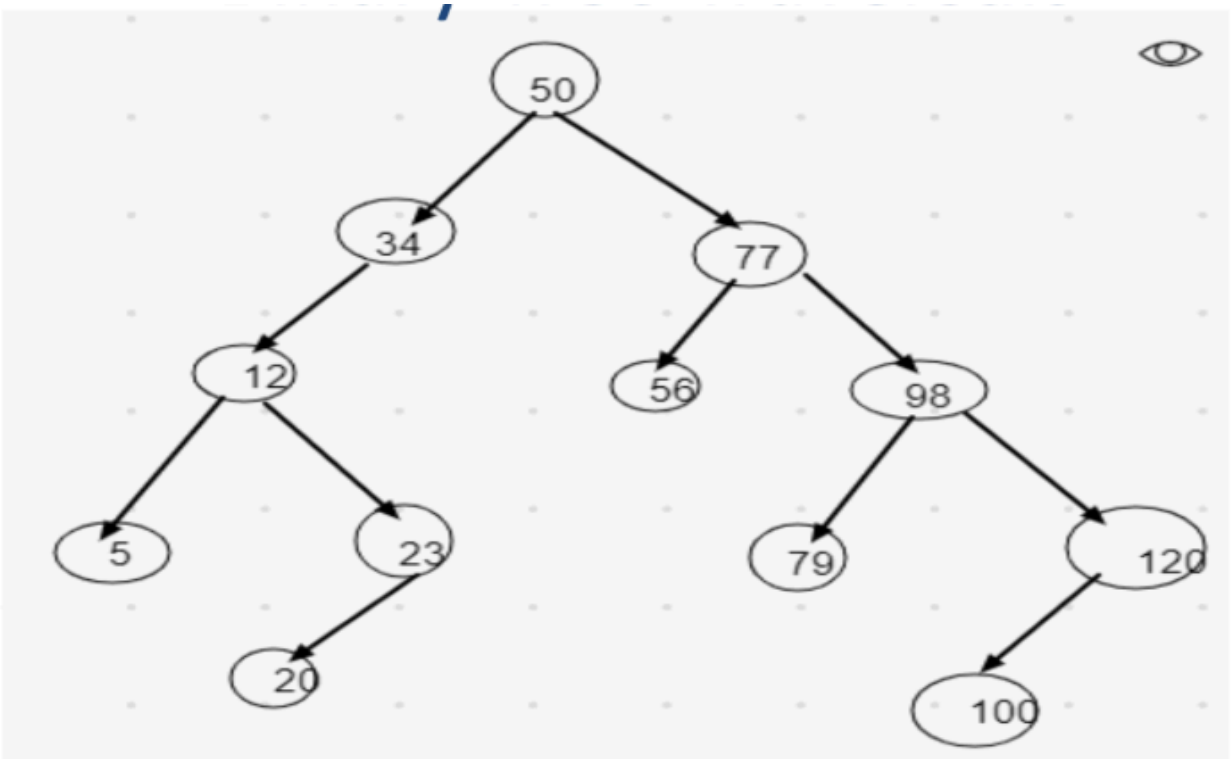
Numericals

1. Tree traversal/construction
2. BFS using queue
3. DFS using stack
4. Prefix/postfix to infix
5. Infix to prefix/postfix
6. AVL construction
7. Binary tree construction
8. B-tree (if in course) construction
9. Hashing
    a. Linear
    b. Quadratic
    c. Separate chaining
    d. Double !!
10. Dictionary/maps function returns what
11. Threaded binary trees
12. Performing pre/post fix operations using stack
13. union/find operations on set
14. Shell sort/bubble sort visualisation

Traversal/Construction Practice

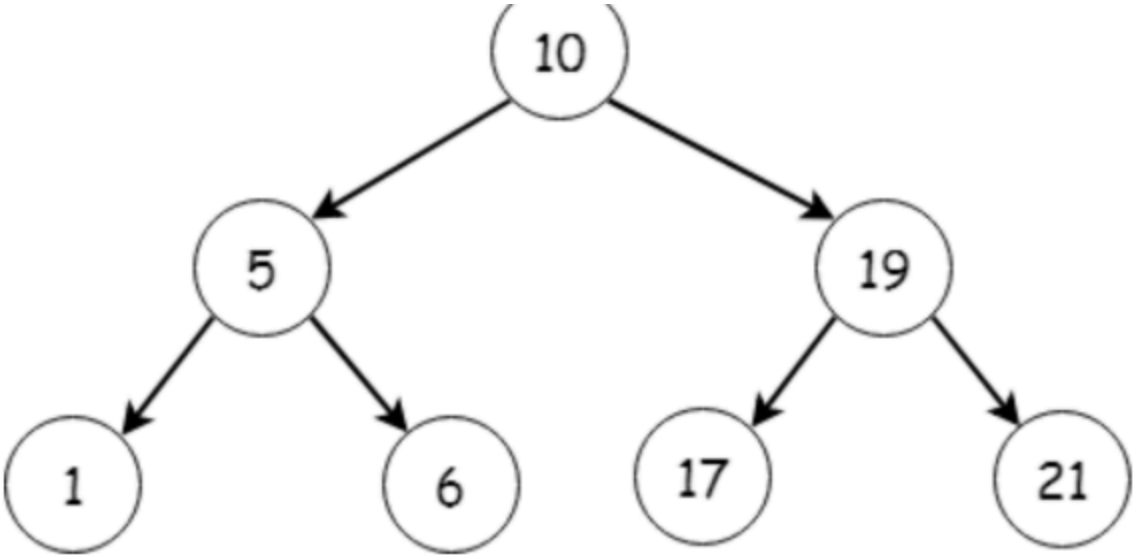


Inorder (Left, Root, Right) : 4 2 5 1 3
Preorder (Root, Left, Right) : 1 2 4 5 3
Postorder (Left, Right, Root) : 4 5 2 3 1

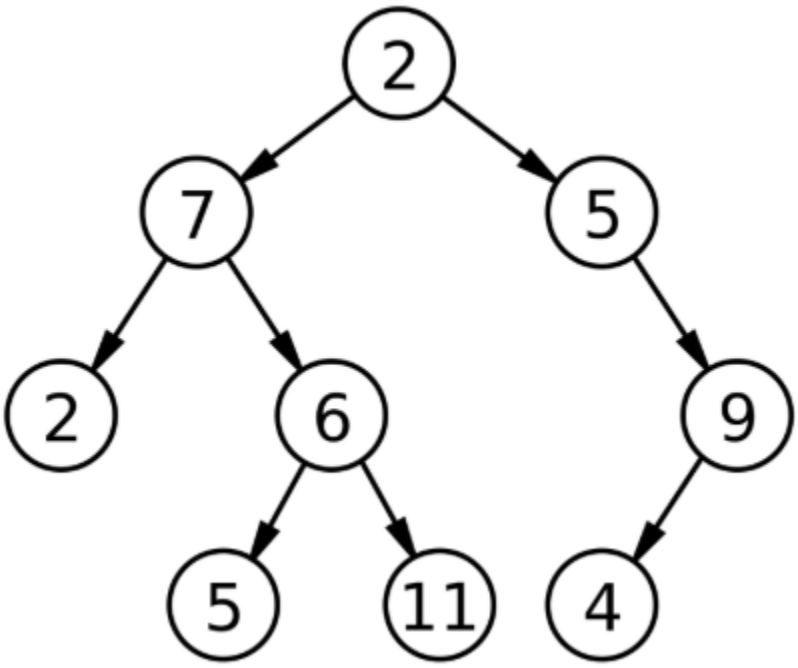


Preorder- 50, 34, 12,5, 23, 20,77, 56,98 79,120, 100,

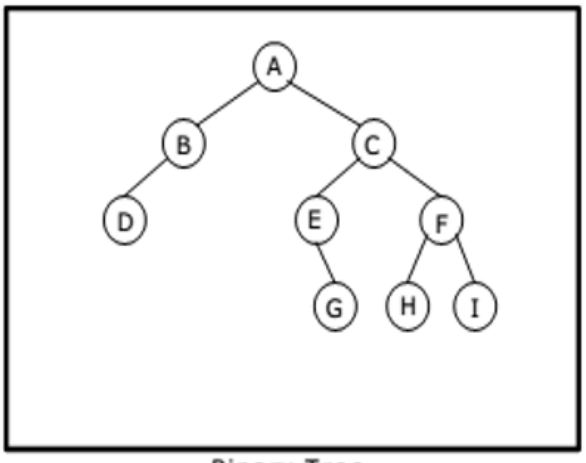
Postorder- 5, 20, 23, 12, 34, 56, 79, 100, 120, 98, 77, 50
Inorder- 5, 12, 20, 23, 34, 50, 56, 77,79, 98, 100, 120



- Inorder: 1-5-6-10-17-19-21
- Preorder: 10-5-1-6-19-17-21
- Postorder: 1-6-5-17-21-19-10



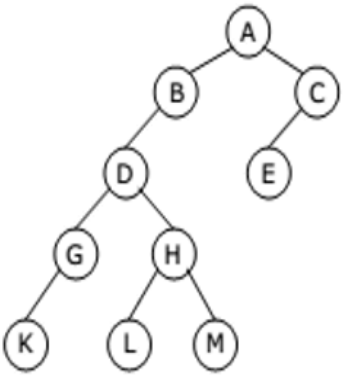
Inorder: 2,7,5,6,11,2,5,4,9
Post-order: 2,5,11,6,7,4,9,5,2
Pre: 2,7,2,6,5,11,5,9,4



Binary Tree

- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
  D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
  D, B, A, E, G, C, H, F, I
- Level order traversal yields:
  A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing



Binary Tree

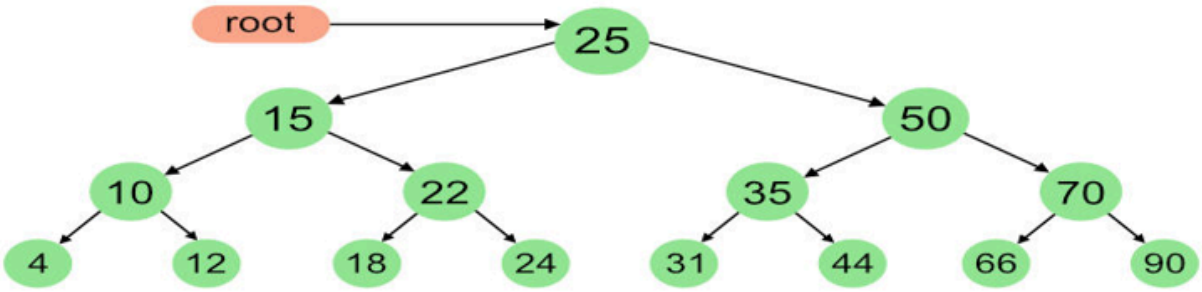
- Preorder traversal yields:
  A, B, D, G, K, H, L, M, C, E
- Postorder travarsal yields:
  K, G, L, M, H, D, B, E, C, A
- Inorder travarsal yields:
  K, G, D, L, H, M, B, A, E, C

Pre, Post and Inorder Traversing

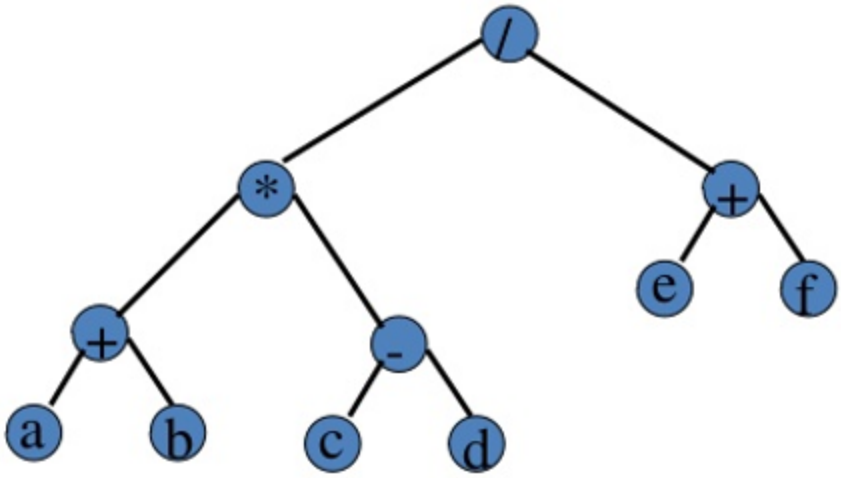
InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

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

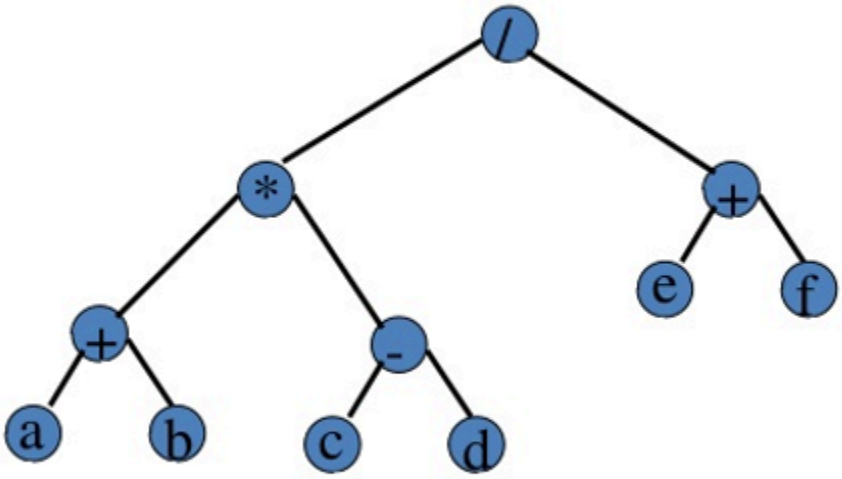


Preorder Of Expression Tree



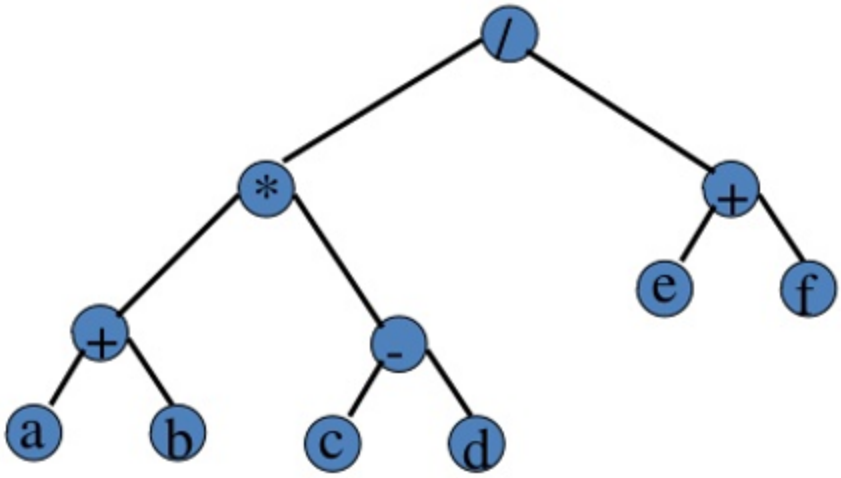
/ * + a b - c d + e f

Inorder Of Expression Tree

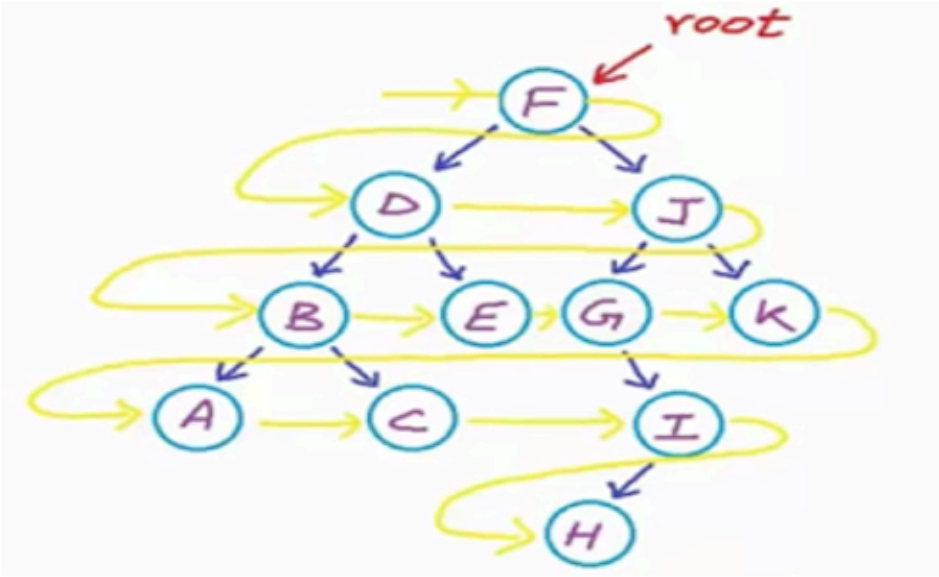


a + b * c - d / e + f

Postorder Of Expression Tree



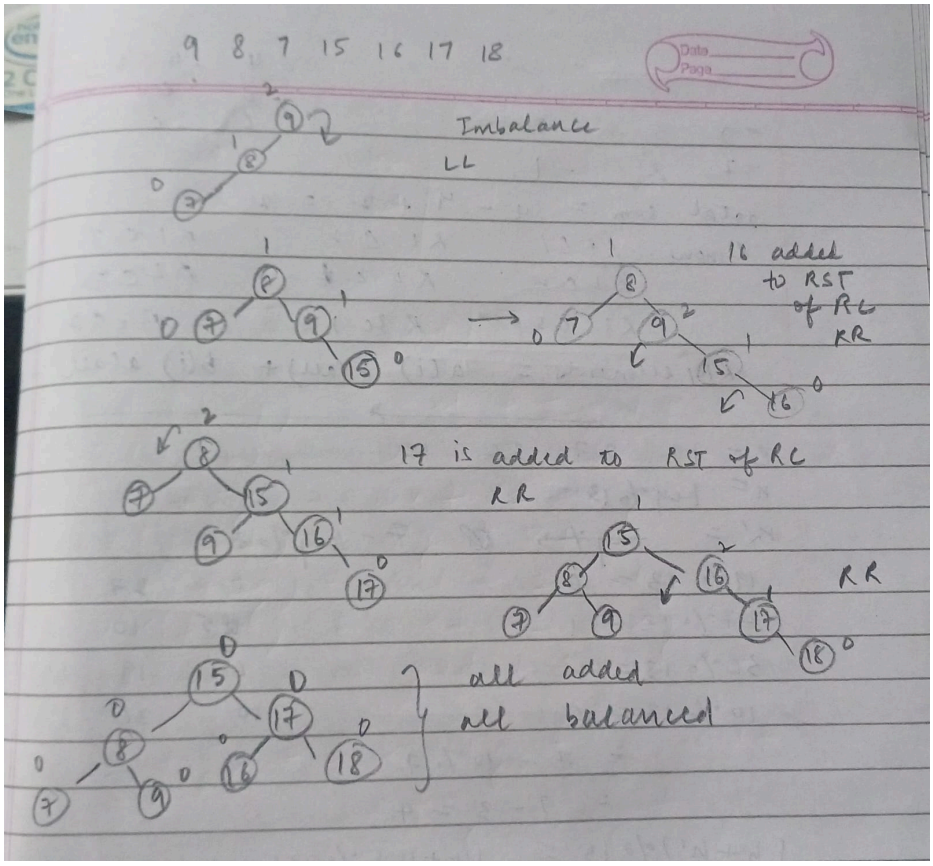
a b + c d - * e f + /



Level Order Traversal=F, D, J, B, E, G, K, A, C I, H

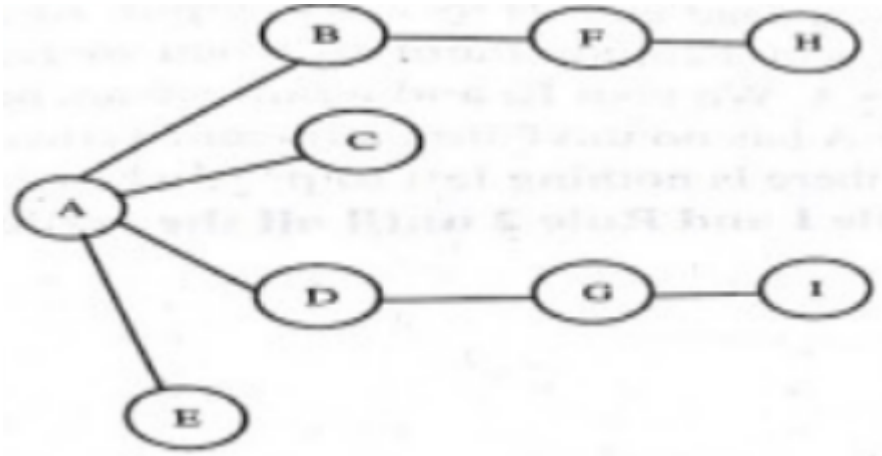
Avl : 15, 20, 24, 10, 13, 7, 30, 36

Construct an AVL Tree for the following sequence 9,8,7,15,16,17,18
Further mention the type of Rotation and the balance factors in each intermediated step



Construct a trie for the words-and,ant,dad,do
Construct the trie for the words-to,a, tea,ted,ten,innt

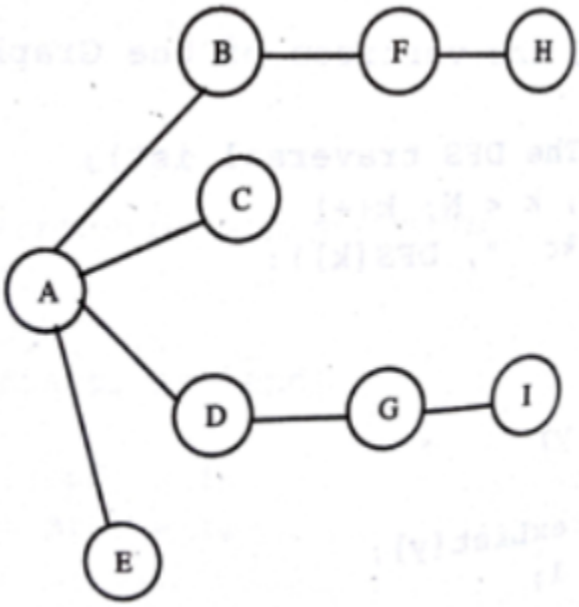
Perform DFS:



Answer; ABFHCDGIE
Working:

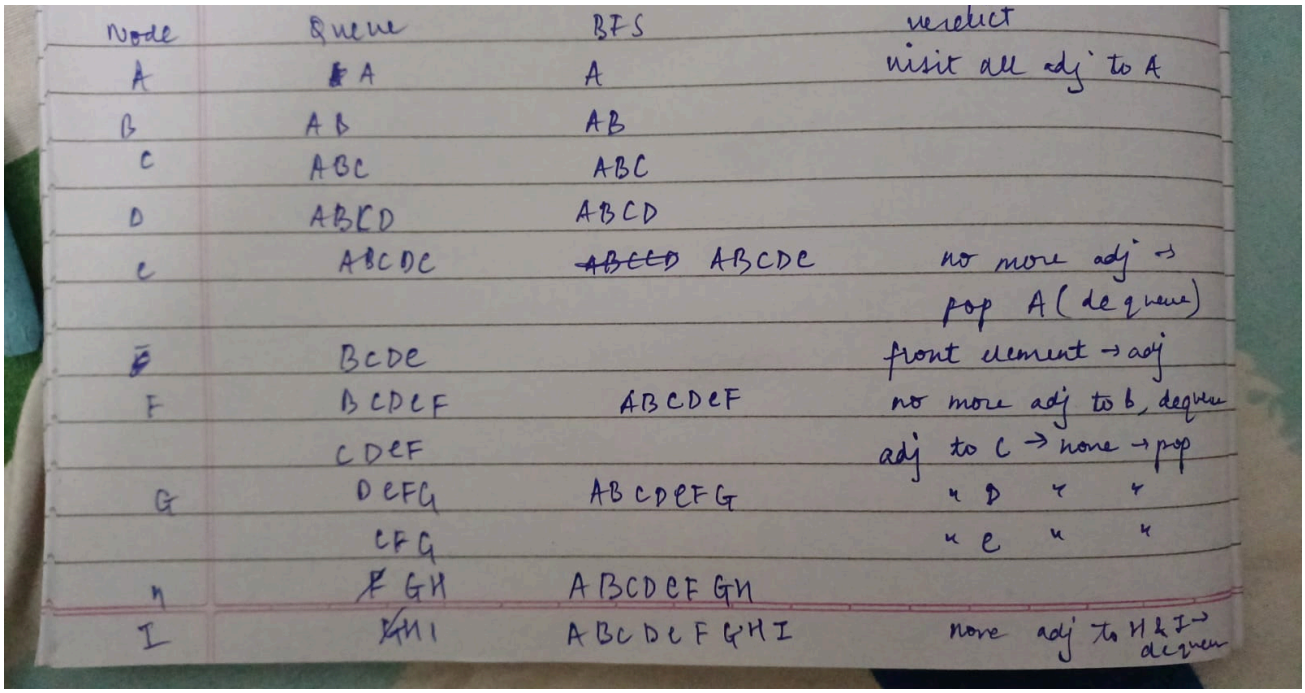
Curr Node	Stack	DFS order	Verdict
A	A	A	
B	AB	AB	
F	ABF	ABF	
H	ABFH	ABFH	No node further
	ABF		pop (no node → pop)
	AB		pop
B	A		
C	AC	ABFHC	pop
	A		
D	AD	ABFHCD	
G	ADG	ABFHCDG	
I	ADGI	ABFHCDGI	pop (no node further)
	ADG		
	AD		
	A		
E	AE	ABFHCDGIE	
	A		pop
	* [ ]		pop . end

Perform BFS:



answer: ABCDEFGHI

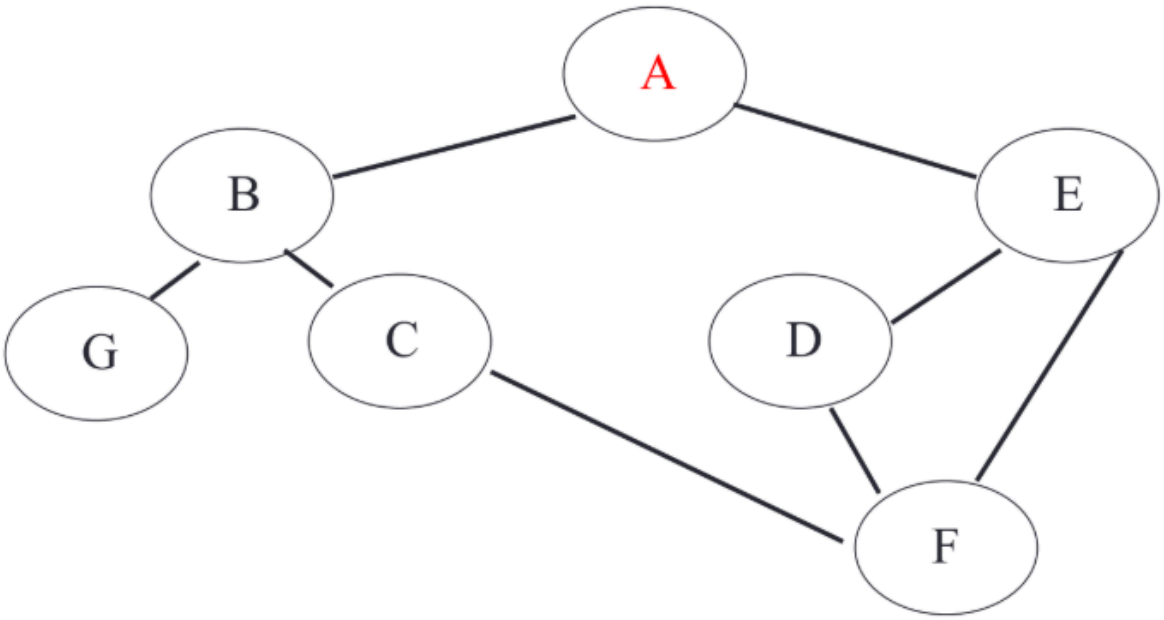
Working: <visit root -> enqueue -> visit all elements adjacent to front of queue -> when no elements, dequeue -> repeat till end>



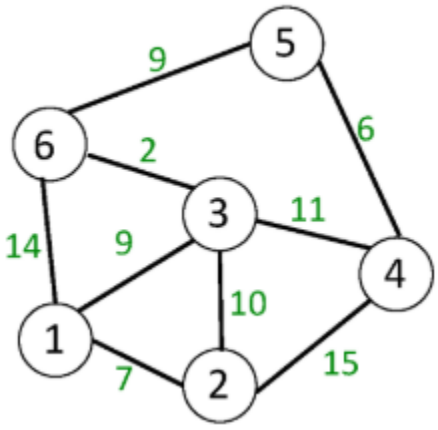
Node	Queue	BFS	reselect
A	A	A	visit all adj to A
B	AB	AB	
C	ABC	ABC	
D	ABCD	ABCD	
E	ABCDE	~~ABCED~~ ABCDE	no more adj → pop A (dequeue)
~~B~~	BCDE		front element → adj
F	BCDEF	ABCDEF	no more adj to b, dequeue
	CDEF		adj to C → none → pop
G	DEFG	ABCDEFG	" D " "
	EFG		" E " "
H	~~F~~ GH	ABCDEFGH	
I	~~GH~~ I	ABCDEFGHI	more adj to H & I → dequeue

Bfs

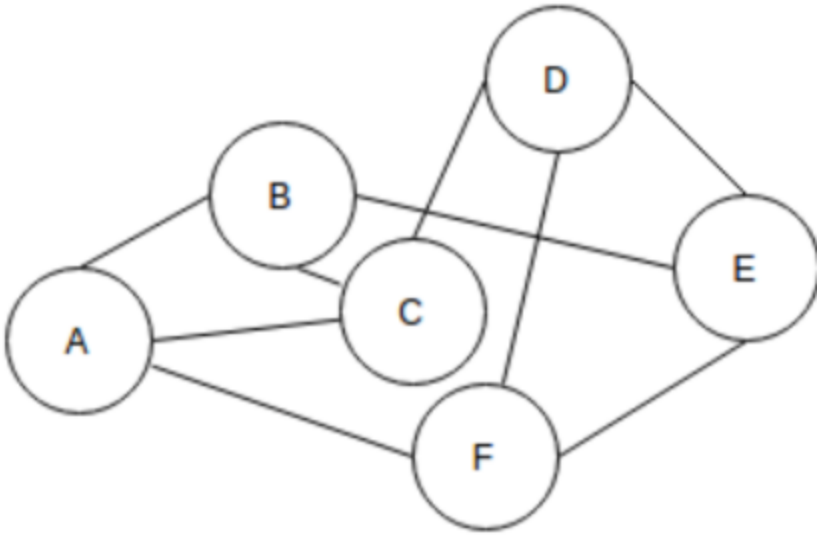
3FS: Graph



A B E G C D F

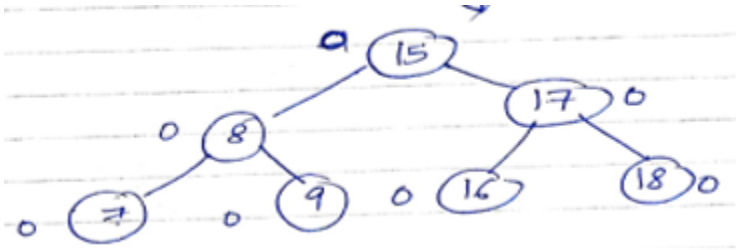


BFS: 1,2,3,6,4,5



ABEDCF / ACDFEB / ABCDEF / ABCDFE / ABEFDC /and so on: DFS

Construct an AVL Tree for the following sequence
9,8,7,15,16,17,18



Construct an AVL tree by inserting the following numbers in the given sequence in an initially empty AVL tree.
15, 20, 24, 10, 13, 7, 30, 36

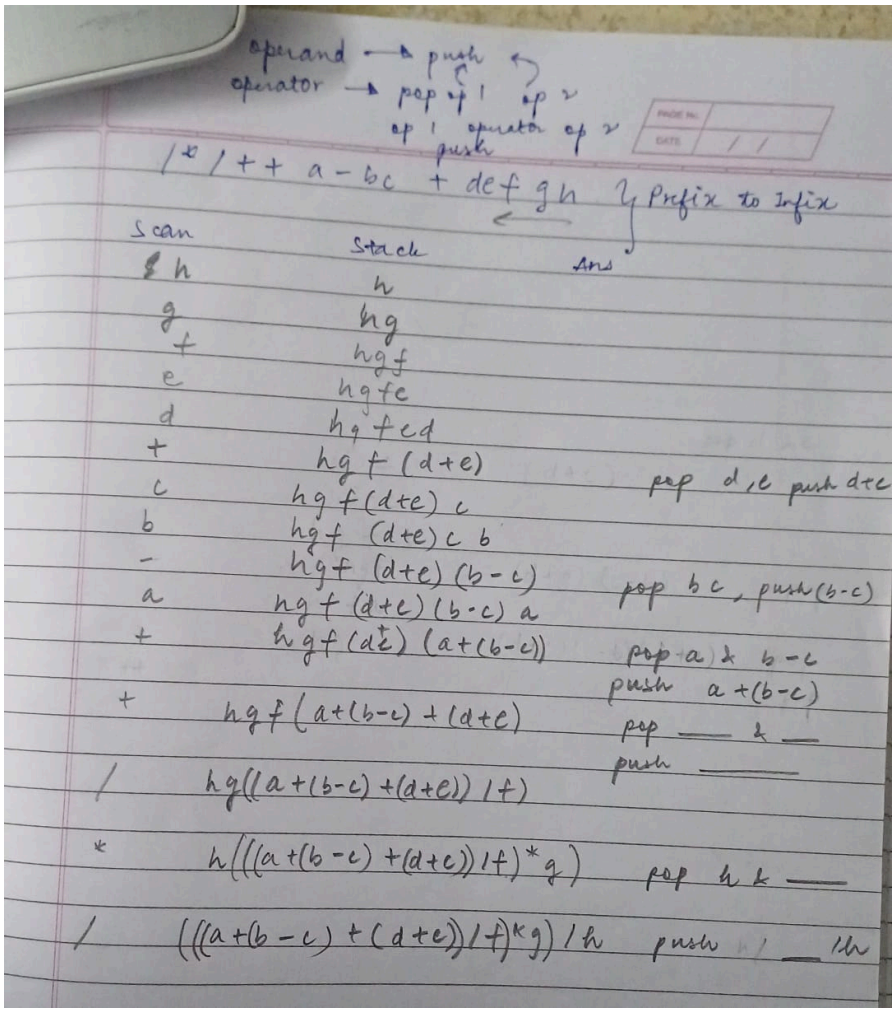
Let us take elements 19, 27,36,10
Perform Double Hashing using the following functions

Lets say, Hash1 (key) = key % 13

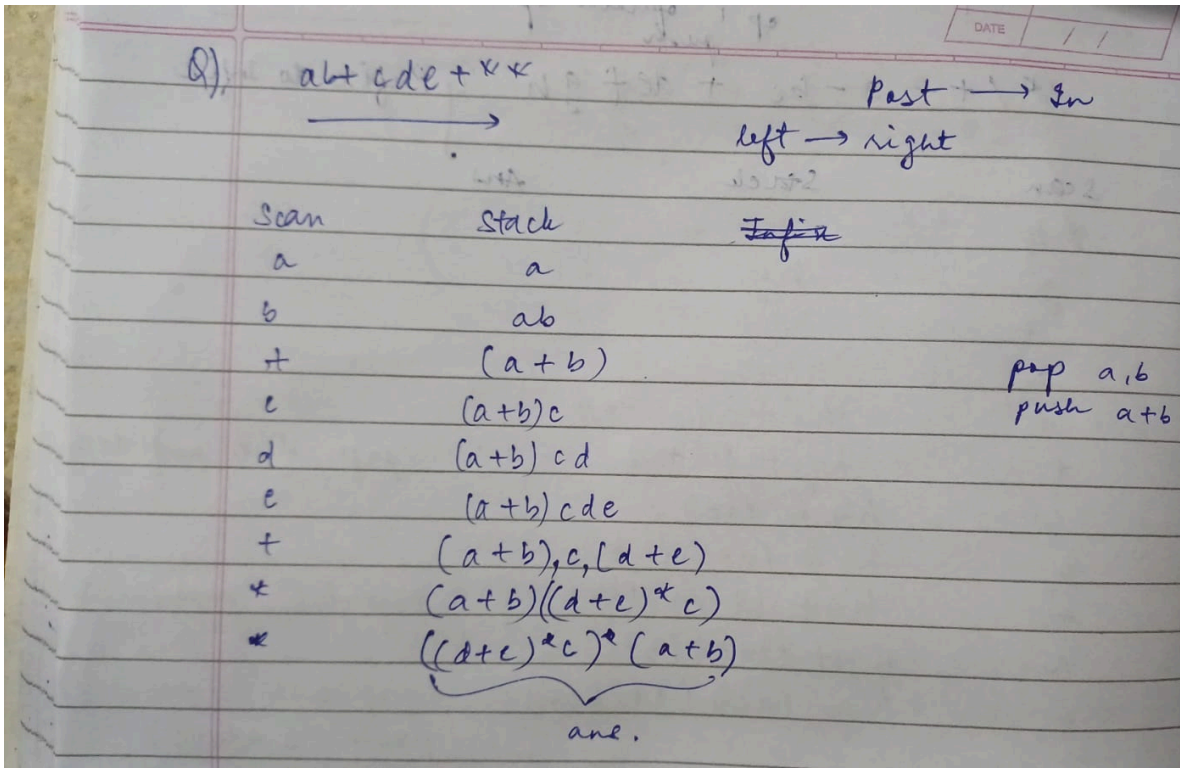
Hash2 (key) = 7 – (key % 7)

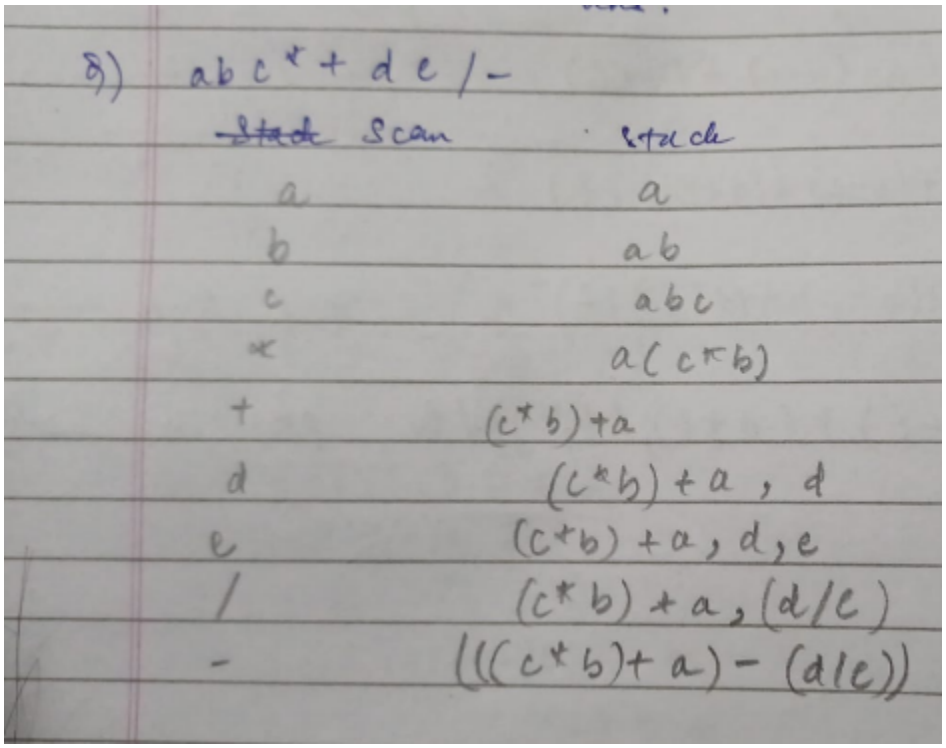
Ans: 27 -> 1, 19 -> 6, 36 -> 10, 10-> 5

Prefix to infix using stack:
Ques: /*/++a-bc+defgh



POSTFIX TO INFIX USING STACK:



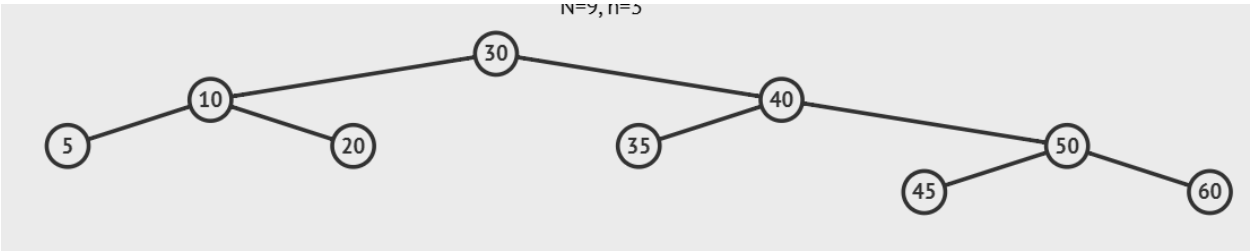


Implement shell sort: [22, 34, 25, 12, 64, 11, 90, 88, 45]
After 1st gap: [22, 11, 25, 12, 45, 34, 90, 88, 64]
After 2nd gap: [22, 11, 25, 12, 45, 34, 64, 88, 90]

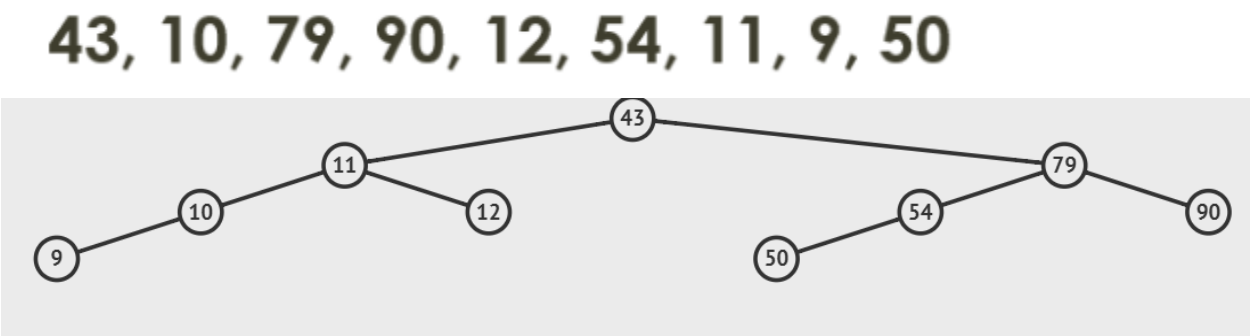
After 3rd: [11,12,22,25,34,45,64,84,90]

Implement shell sort:
[9,8,3,7,5,6,4,1]
1st gap: [5,6,3,1,9,8,4,7]
2nd: [3,1,4,6,5,7,9,8]
3rd: [1,3,4,5,6,7,8,9]

Construct AVL: 30, 20, 40, 10, 5, 35, 50, 45, 60



Construct AVL



Pyq:
https://docs.google.com/document/d/1eI2tm7AgUuxJo6rGhtuZTuENM9N8Tss1/edit a