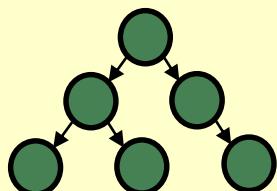
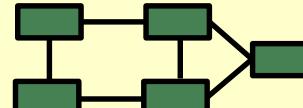
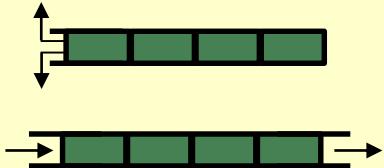


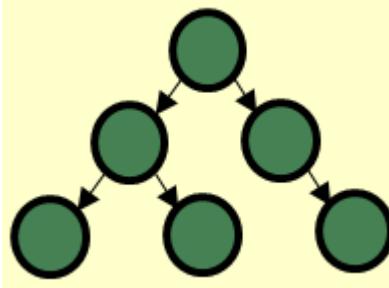
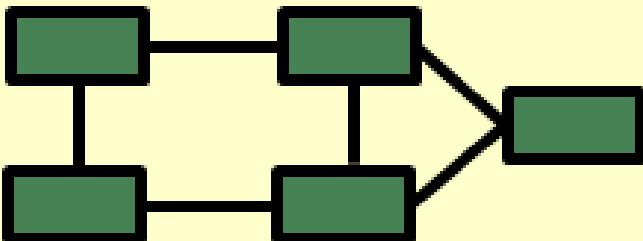


Data Structures





Introduction to Data Structures & Algorithms





Why DSA?

We are surrounded by information which is to be stored in form of data !

Storing is not only important. Fetching / Retrieving and Processing has to be efficient

Data has to be properly structured

List, Stack, Tree, Graph

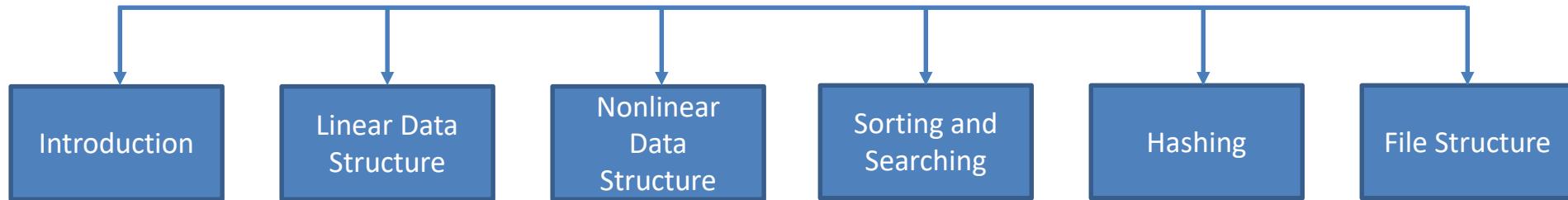
Implementation in C, C++, Java etc

GATE, Job Interview

Big data, AI, Machine Learning

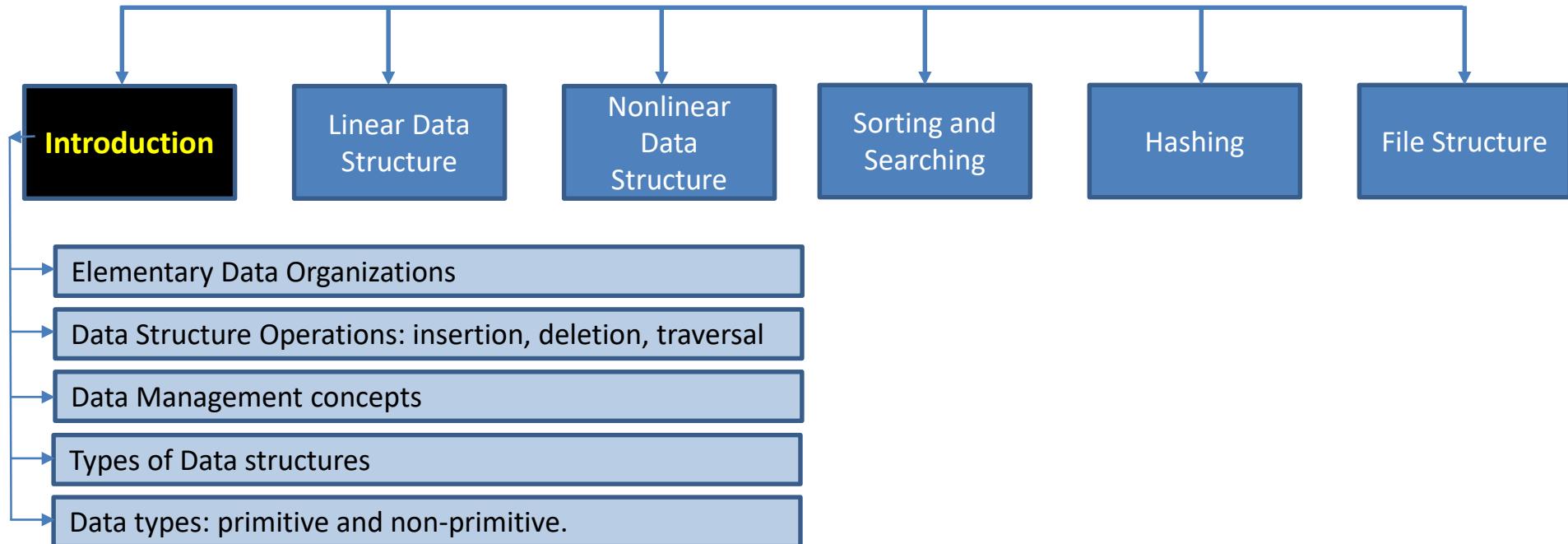
“Data Structure” and “Algorithms” are two different words

Data Structures & Algorithms (DSA)





DSA: Introduction





DSA: Introduction

What is a data structure?

Structuring the data

Organizing the data

Why do I need to organize data?

To efficiently manage the data

What do you mean by efficiently manage data?

So that the data can be stored, retrieved and updated in a faster way



DSA: Introduction

Source: Wikipedia

What is a data structure?

In computer science, a data structure is a data organization, management, and storage format that enables efficient access and modification.

More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

It makes the algorithm (and in turn code) easy to understand and manage.

Concerns

Efficiency

Scalability



DSA: Introduction

Source: Wikipedia

Abstract Data Types (ADT)

In computer science, an abstract data type (ADT) is a mathematical model for data types.

ADT	Implementation in DS
List	Array, Linked List
Queue	Array, Linked List
Stack	Array, Linked List
Map	Tree, Graph



DSA: Introduction

Computational Complexity

- To understand the performance of the data structure or algorithm
- How much **time** does an algorithm need to complete?
- How much memory (**space**) does an algorithm require for its computation?

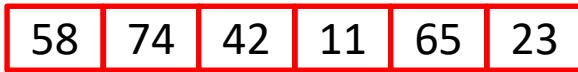
Upper bound and Lower bound

- Big-O Notation (O) – Worst case scenario
- Omega Notation (Ω) – Best case scenario
- Theta Notation (Θ) – Average case scenario

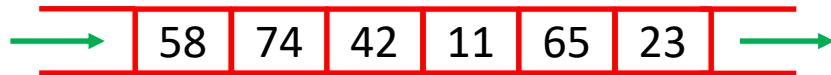


Thank You

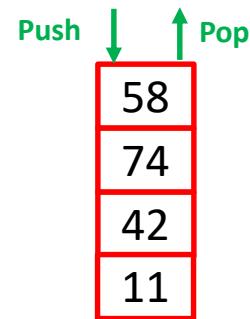
Next: Types of Data Structure



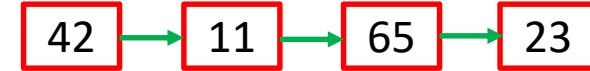
Array



Queue



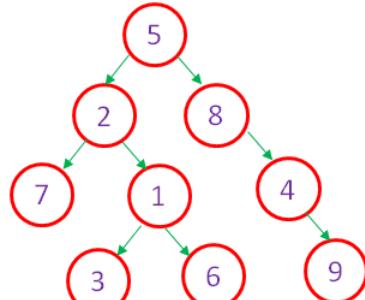
Stack



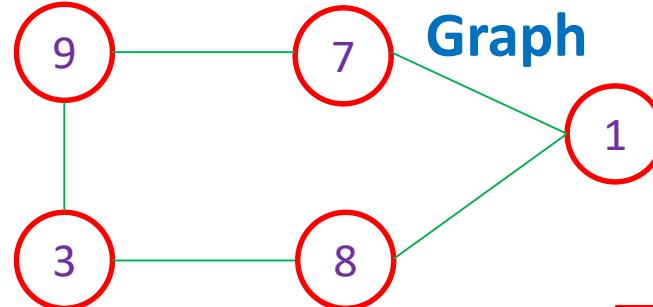
Linked List

Types of Data Structures

Tree



Graph





Types of Data Structures

Types of Data Structures

- Primitive Vs. Non-primitive data structures
- Linear Vs. Non-linear data structures
- Static Vs. Dynamic data structure



Primitive Vs. Non-primitive Data Structures

Primitive Data Structure

- Fundamental data structure available for programming
- Example: int, float, char, pointer
- Can store only single data value

Non-primitive Data Structure

- User defined data structure
- Classified into: linear and non-linear data structure
- Example of linear data structure: array, stack, queue, linked list
- Example of non-linear data structure: tree, graph
- Can store multiple data values

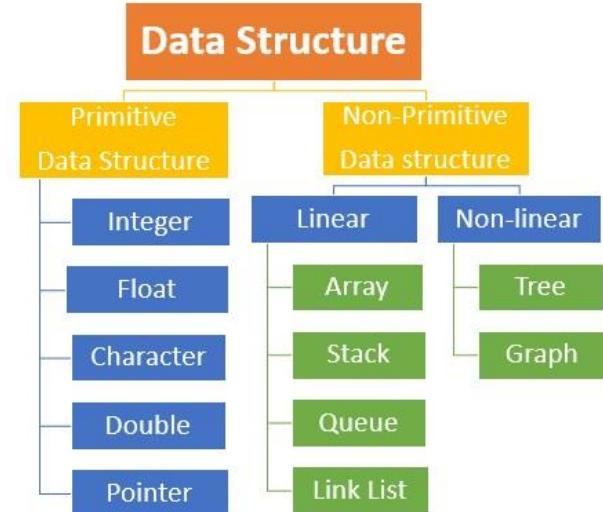


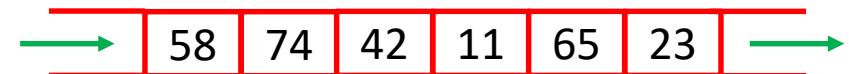
Image source: <https://msatechnosoft.in/>



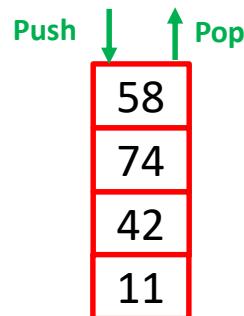
Linear Data Structures



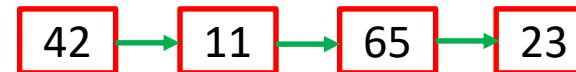
Array



Queue



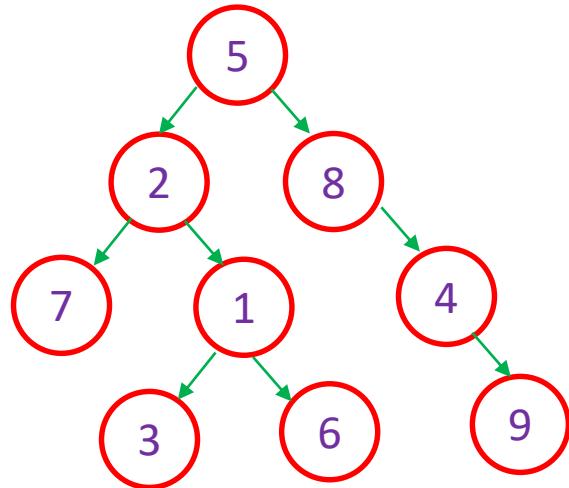
Stack



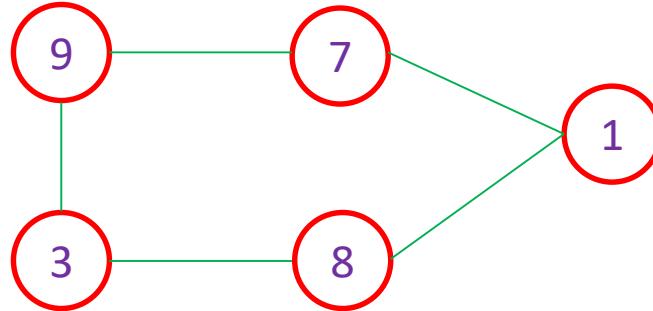
Linked List



Non-linear Data Structures



Tree



Graph



Linear Vs. Non-Linear DS

Linear Data Structure

- It arranges the data in orderly manner where the elements are attached adjacently
- Level: Single
- Implementation: Easier
- Example: Array, Stack, Linked List, Queue

Non-linear Data Structure

- It arranges the data in sorted order, creating a relationship among the data element
- Level: Multiple
- Implementation: Difficult
- Example: Tree, Graph



Static Vs Dynamic DS

Static Data Structure

- Memory is allocated at compile time (before running/execution).
- Maximum size is fixed
- Example: Array
- **Advantage:** Fast access. **Disadvantage:** Slower insertion/deletion, memory waste

Dynamic Data Structure

- Memory is allocated at run time
- Maximum size is flexible (can be increased/reduced)
- Example: Linked list
- **Advantage:** Faster insertion/deletion, Memory save **Disadvantage:** Slow access, Computation overhead



Operations on DS

- Search
- Sort
- Insert
- Delete
- Update

42	23	74	11	65	58
----	----	----	----	----	----

11?

11	23	42	58	65	74
----	----	----	----	----	----

49?

11	23	42	58	65	74
----	----	----	----	----	----

23

11	23	42	58	65	74
----	----	----	----	----	----

58->59

11	23	42	58	65	74
----	----	----	----	----	----

11	23	42	49	58	65	74
----	----	----	----	----	----	----

11	42	58	65	74
----	----	----	----	----

11	23	42	59	65	74
----	----	----	----	----	----

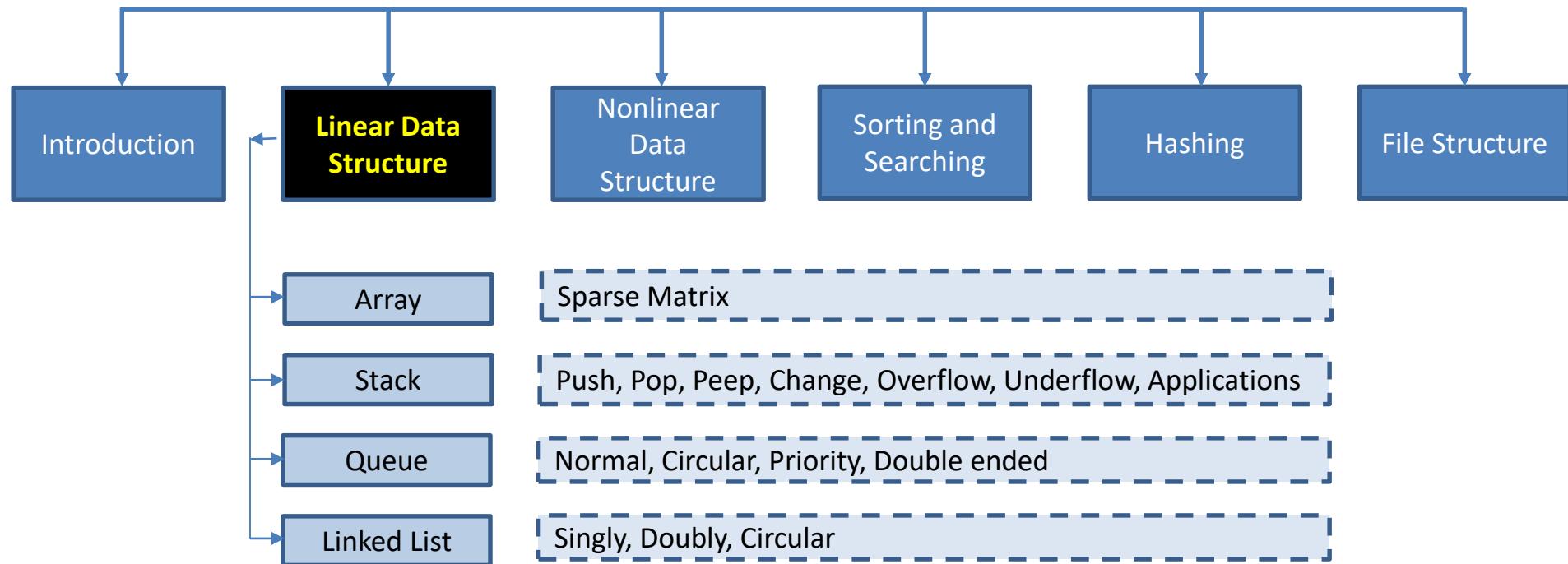


Thank You

Next: Linear Data Structure (Sparse Matrix)



DSA: Linear Data Structure



Sparse Matrix

$$\begin{bmatrix} 0 & 0 & 5 & 2 \\ 0 & 0 & 0 & 6 \\ 9 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 \end{bmatrix} \rightarrow$$

Row	Column	Value
0	2	5
0	3	2
1	3	6
2	0	9
3	0	7





Sparse Matrix

- Majority of elements are ZERO

$$\begin{bmatrix} 0 & 0 & 5 & 2 \\ 0 & 0 & 0 & 6 \\ 9 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 \end{bmatrix}$$

- How do we implement/represent sparse matrix?
 - Arrays
 - Linked List

- Why Sparse matrix?

- Storage [Only non-zero elements are stored in a triple <row, col, value>]
- Computation power [Only non-zero elements are traversed]



Sparse Matrix

- **Array representation of a sparse matrix**

- 2D array can be represented using three rows. First row indicating row of non-zero element, second row representing column of non-zero element and third row depicting the non-zero value.

- **Example**

0	0	5	2
0	0	0	6
9	0	0	0
7	0	0	0

Row	0	0	1	2	3
Column	2	3	3	0	0
Value	5	2	6	9	7

Row	Column	Value
0	2	5
0	3	2
1	3	6
2	0	9
3	0	7



Sparse Matrix

- Assignment 1: Write a program to check whether the given matrix is sparse matrix or not.
- Hint: Number of ZEROS should be more than half of the total elements in the matrix

```
int isSparse(int A[row][col])
{
int i,j, count=0;
for(i=0; i<row; ++i)
{for(j=0; j<col; ++j)
    if(A[i][j]==0) count++;
}
if(count>=(row*col)/2) return count;
else return -1;
}

/* sparse matrix */
#include<stdio.h>
#define row 4
#define col 3
int main()
{
    int S[row][col]={0,5,0,0,0,8,0,0,0,0,1,0,0},status;
    status = isSparse(S);
    if(status== -1)printf("Not a sparse matrix");
    else printf("Sparse matrix: With %d zero elements
from total %d elements", status, row*col);
    return 0;
}
```

OUTPUT

Sparse matrix: With 9 zero elements from total 12 elements



Sparse Matrix

- Assignment 2: Write a program to convert the given matrix into sparse matrix.

```
/* sparse matrix */
#include<stdio.h>
#define row 4
#define col 3
#define max 10
void main()
{ int original[row][col]={0,5,0,0,0,0,8,0,0,0,1,0,0}, sparse[max][3], i,j,count=0;
  for(i=0; i<row; ++i)
  {for(j=0; j<col; ++j)
    {if(original[i][j]!=0)
      {sparse[count][0]=i;
       sparse[count][1]=j;
       sparse[count][2]=original[i][j];
       count++;}
    }
  }
  for(i=0; i<count; ++i) printf("%d\t%d\t%d\n", sparse[i][0], sparse[i][1], sparse[i][2]);
}
```

<https://github.com/hbpatel1976/Data-Structure/blob/master/sparse2.c>

OUTPUT

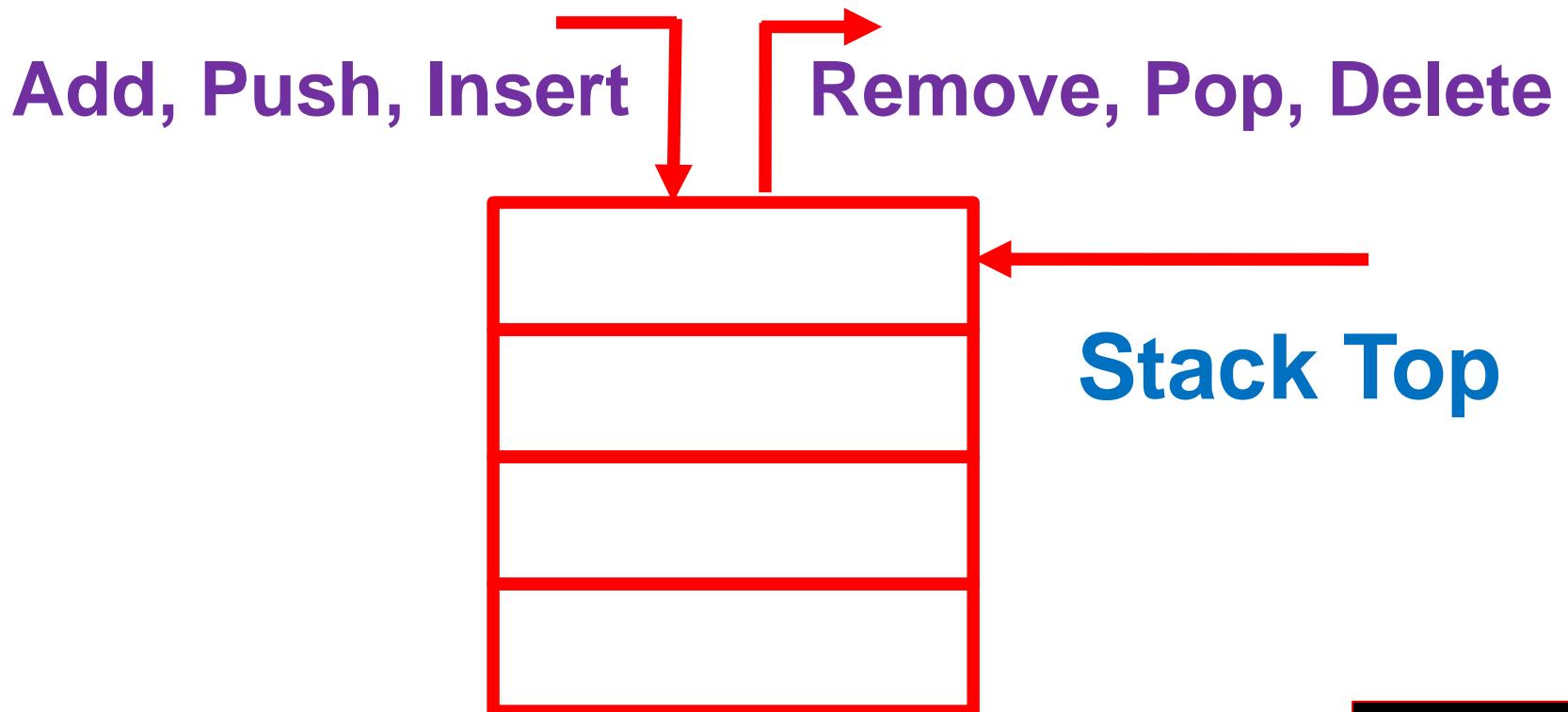
0	1	5
1	2	8
3	0	1



Thank You

Next: Linear Data Structure (Stack)

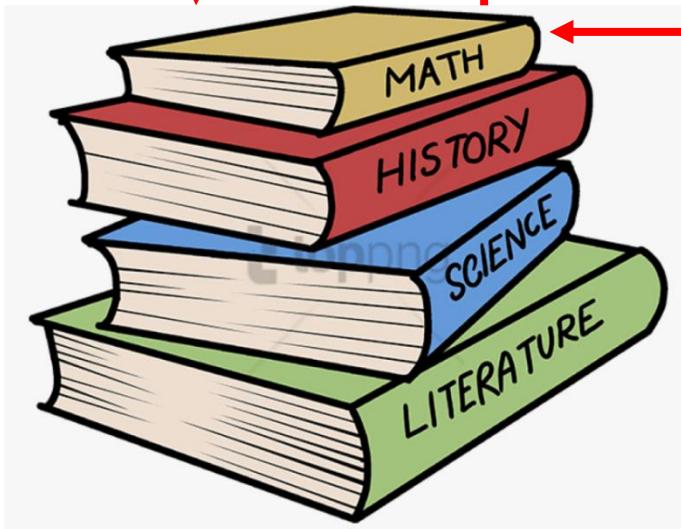
Stack





Stack

Add Book
(Push / Insertion)



Remove Book (Pop / Deletion)

Stack Top



Another Examples



Stack

Stack is a linear data structure (having similar data types)

Insertion (**pushing** an element) and deletion (**poping** an element) is possible from one end only, which is nothing but **top of the stack**.

Last in first out (**LIFO**) [Or First in last out - FILO]

Last in first out (**LIFO**) [Or First in last out - FILO]

Stack is an ordered list

`peep ()` – Return top of the stack without removing the element

`isEmpty ()` - Is the stack empty?

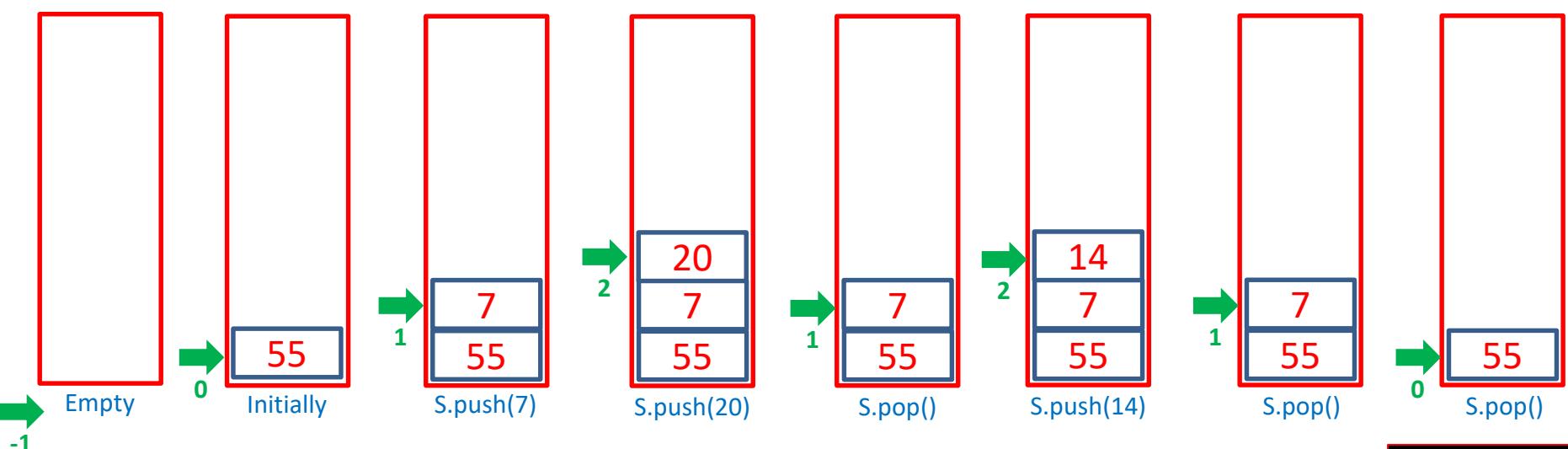
`isFull ()` - Is the stack full?



Stack: Example

Let S is an instance of stack. Consider the following sequence of operations performed on S which initially contains element with 55 as top most elements. What will be the element at the top of the stack after the following sequence of operations?

S.push(7), S.push(20), S.pop(), S.push(14), S.pop(), S.pop()





Stack: Implementation

```
#include<stdio.h>
#define size 5
int S [size], top=-1;

int main()
{
    push(50);
    push(7);
    push(20);
    pop();
    push(14);
    pop();
    pop();
    printf("Stack top = %d",top);
    return 0;
}

void push(int x)
{
    if(top==size-1)
        {printf("Stack Overflow"); return;}
    else
    {
        top++;
        S[top]=x;
        printf("Element pushed : %d\n",S[top]);
    }
}

void pop(void)
{
    if(top==-1)
        {printf("Stack Underflow"); return;}
    else
        {printf("Element popped : %d\n",S[top]);top--}
}
```

OUTPUT

```
Element pushed : 50
Element pushed : 7
Element pushed : 20
Element popped : 20
Element pushed : 14
Element popped : 14
Element popped : 7
Stack top = 0
```



Stack: Other Terminology

peep() – get the top data element of the stack, without removing it.

```
int peep ()  
{  
return S[top];  
}
```

isFull() – check if stack is full.

```
int isFull ()  
{  
if (top == size-1) return 1;  
else return 0;  
}
```

isEmpty() – check if stack is empty.

```
int isEmpty ()  
{  
if (top == -1) return 1;  
else return 0;  
}
```



Stack: Assignment

Write a complete menu driven program that incorporates all the functionalities of the stack viz. push, pop, peep, isFull, isEmpty.

<https://github.com/hbpatel1976/Data-Structure/blob/master/stackfull.c>



Stack: Application

Check whether the expression parenthesis in right order or not.

```
#include <stdio.h>
#define size 10
int stack[size],top=-1;
void main()
{
char str[20],ch;
int i=0,flag=0;
void push(char);
char pop(void);
printf("Enter the expression : ");
scanf("%s",str);
while(str[i]!='\0')
{
if(str[i]=='{' || str[i]=='(' || str[i]=='<' || str[i]=='[') push(str[i]);
else if(str[i]=='}' || str[i]==')' || str[i]==')' || str[i]==']')
{
ch=pop();
if(isPair(ch,str[i])==0){flag=1;break;}
}
else {flag=2;break;}
i++;
}
if(flag==0)printf("Balanced Brackets\n");
else if(flag==1)printf("Imbalanced Brackets\n");
else if(flag==2)printf("Invalid Input\n");
}
```



Stack: Application

```
void push(char c)
{
    stack[++top]=c;
}

char pop(void)
{
    return stack[top--];
}

int isPair(char ch1, char ch2)
{
    if((ch1=='<' && ch2== '>') || (ch2=='<' && ch1== '>'))return 1;
    if((ch1=='(' && ch2== ')') || (ch2=='(' && ch1== ')'))return 1;
    if((ch1=='{' && ch2== '}') || (ch2=='{' && ch1== '}'))return 1;
    if((ch1=='[' && ch2== ']') || (ch2=='[' && ch1== ']'))return 1;
    return 0;
}
```

OUTPUT

Enter the expression : [{ (<>) }]
Balanced Brackets

OUTPUT

Enter the expression : [{ (<) >) }]
Imbalanced Brackets

OUTPUT

Enter the expression : [{ (<1>2) }]
Invalid Input



Thank You

Next: Infix, Prefix & Postfix Notation

Infix : a + b

Prefix : + a b

Postfix : a b +





Infix Expressions

Expression: <operand> <operator> <operand>

Example: a + 5

Example: (b * 8) + 5

Infix notation or infix operation. Operator is there between operands

We need rules to evaluate infix expression. Why?

$$5 + 1 \times 6 \quad \begin{matrix} 5 + 1 = 6, & 6 \times 6 = 36 \\ 1 \times 6 = 6, & 5 + 6 = 11 \end{matrix}$$



Infix Expressions

Rules: Precedence and associativity rules

Priority	Operators	Associativity
HIGH-1	() {} []	
2	\wedge	Right to Left
3	$*$ /	Left to Right
LOW-4	+ -	Left to Right

$$5 + 1 \times 6 \\ 1 \times 6 = 6, \quad 5 + 6 = 11$$

$$(5 + 1) \times 6 \\ 5 + 1 = 6, \quad 6 \times 6 = 36$$



Infix Expressions

Rules: Precedence and associativity rules

Priority	Operators	Associativity
HIGH-1	() { } []	
2	\wedge	Right to Left
3	$*$ /	Left to Right
LOW-4	+ -	Left to Right

Complex Expressions

$$3+5*7-30/6$$

$$3+35-30/6$$

$$3+35-5$$

$$3+35-5$$

$$38-5$$

$$33$$

$$2 \wedge 3 \wedge 2$$

$$2 \wedge 9$$

$$512$$



Prefix Expressions

Also known as **Polish Notation**

Expression: <operator> <operand> <operand>

Infix to Prefix

8 + 5 → + 8 5

A * B - C → * A B-C → -* A B C



Postfix Expressions

Also known as **Reverse Polish Notation**

Expression: <operand> <operand> <operator>

Infix to Prefix

8 + 5 → 8 5 +

A * B - C → A B * - C → A B * C -



Thank You

Next: Reverse Polish Notation

Reverse Polish Notation

Infix $(a + b)$ => Postfix $(a\ b\ +)$



Rules: Converting Infix to Postfix

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
 -3.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '(', push it.)
 -3.2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop the stack and and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

$(a+b^c^d) * (e+f/d)$ [Scan from left to right]

Input Expression	Stack	Output Expression
((
a	(a
+	(+	a
b	(+	a b
^	(+ ^	a b
c	(+ ^	a b c
^	(+ ^	a b c
d	(+ ^^	a b c d
)	*	a b c d^^+
*	*	a b c d^^+
(*(a b c d^^+
e	*(a b c d^^+e
+	*(+	a b c d^^+e
f	*(+	a b c d^^+ef
/	*(+ /	a b c d^^+ef
d	*(+ /	a b c d^^+efd
)		a b c d^^+efd/+*

Infix to Postfix Conversion using Stack

Convert given infix string to
postfix notation using stack
 $(a+b^c^d) * (e+f/d)$

Priority	Operators	Associativity
HIGH-1	() {} []	
2	^	Right to Left
3	* /	Left to Right
LOW-4	+ -	Left to Right

K + L - M * N + (O ^ P) * W / U / V * T + Q

Input Expression	Stack	Output Expression
K		K
+	+	K
L	+	KL
-	-	KL+
M	-	KL+M
*	- *	KL+M
N	- *	KL+MN
+	+	KL+MN*-
(+()	KL+MN*-
O	+()	KL+MN*-O
^	+(^)	KL+MN*-O
P	+(^)	KL+MN*-OP
)	+	KL+MN*-OP
*	+*	KL+MN*-OP
W	+*	KL+MN*-OP^W
/	+/	KL+MN*-OP^W*
U	+/	KL+MN*-OP^W*U
/	+/	KL+MN*-OP^W*U/
V	+/	KL+MN*-OP^W*U/V
*	+*	KL+MN*-OP^W*U/V/
T	+*	KL+MN*-OP^W*U/V/T
+	+	KL+MN*-OP^W*U/V/T*+
Q	+	KL+MN*-OP^W*U/V/T*+Q
		KL+MN*-OP^W*U/V/T*+Q+

Infix to Postfix Conversion using Stack

Priority	Operators	Associativity
HIGH-1	() {} []	
2	^	Right to Left
3	* /	Left to Right
LOW-4	+ -	Left to Right



Online Infix to Postfix Conversion

<https://www.mathblog.dk/tools/infix-postfix-converter/>

Infix to postfix:

$(a+b^c^d)*(e+f/d)$

CONVERT

a b c d ^ ^ + e f d / + *

Infix to postfix:

$K + L - M * N + (O ^ P) * W / U / V * T + Q$

CONVERT

K L + M N * - O P ^ W * U / V / T * + Q +



Implementation: Infix to Postfix Conversion

```
#include<stdio.h>
char stack[20];
int top = -1;
void push(char x)
{
    stack[++top] = x;
}

char pop()
{
    if(top == -1) return -1;
    else return stack[top--];
}

int priority(char x)
{
    if(x == '(') return 0;
    if(x == '+' || x == '-') return 1;
    if(x == '*' || x == '/') return 2;
}
```

```
main()
{
    char exp[20],x;
    int i=0;
    printf("Enter the expression :: ");
    scanf("%s",exp);
    while(exp[i] != '\0')
    {
        if(isalnum(exp[i])) printf("%c",exp[i]);
        else if(exp[i] == '(') push(exp[i]);
        else if(exp[i] == ')')
        {while((x = pop()) != '(') printf("%c", x);}
        else
        {while(priority(stack[top])>=priority(exp[i]))
            printf("%c",pop());
            push(exp[i]);}
        i++;
    }
    while(top != -1){printf("%c",pop());}
}
```



Thank You

Next: Polish Notation

Postfix to Infix Conversion

Postfix (a b +) => Infix (a + b)





Postfix to Infix Conversion

A B - C D E + * + F G + -

[Scan from left to right]

Input Expression	Stack	Output Expression
A	A	
B	A B	
-		
C	C	A-B
D	C D	A-B
E	C D E	A-B
+	C	A-B
*		A-B D+E
+		A-B C*(D+E)
		A-B + C*(D+E)
F	F	A-B + C*(D+E)
G	F G	A-B + C*(D+E)
+		A-B + (D+E)*C
-		F+G
		A-B + (D+E)*C F+G
		A-B + (D+E)*C - (F+G)



Postfix to Infix Conversion

12, 7, 3, -, /, 2, 1, 5, +, *, +

[Scan from left to right]

Input Expression	Stack	Output Expression
12	12	
7	12 7	
3	12 7 3	
-	12	
/		7-3
2	2	12 / (7-3)
1	2 1	
5	2 1 5	
+	2	
*		12 / (7-3) (1+5)
+		(7-3) / 12 2*(5+1)
		(7-3) / 12 + 2*(5+1)



Thank You

Next: Recursion

Recursion

```
int fact(int n) {  
    if (n<=1)  
        return 1;  
    else  
        return (n * fact(n-1));  
}
```

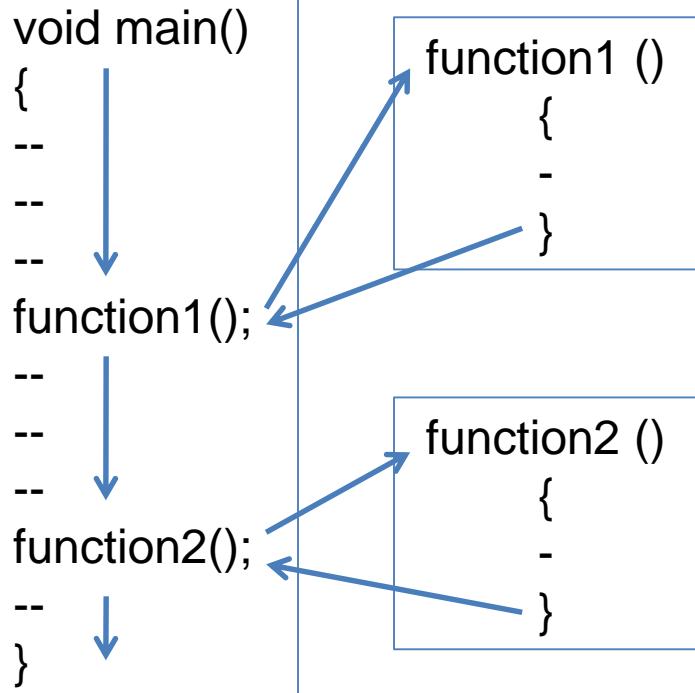
Terminating Condition → **fact(n)**



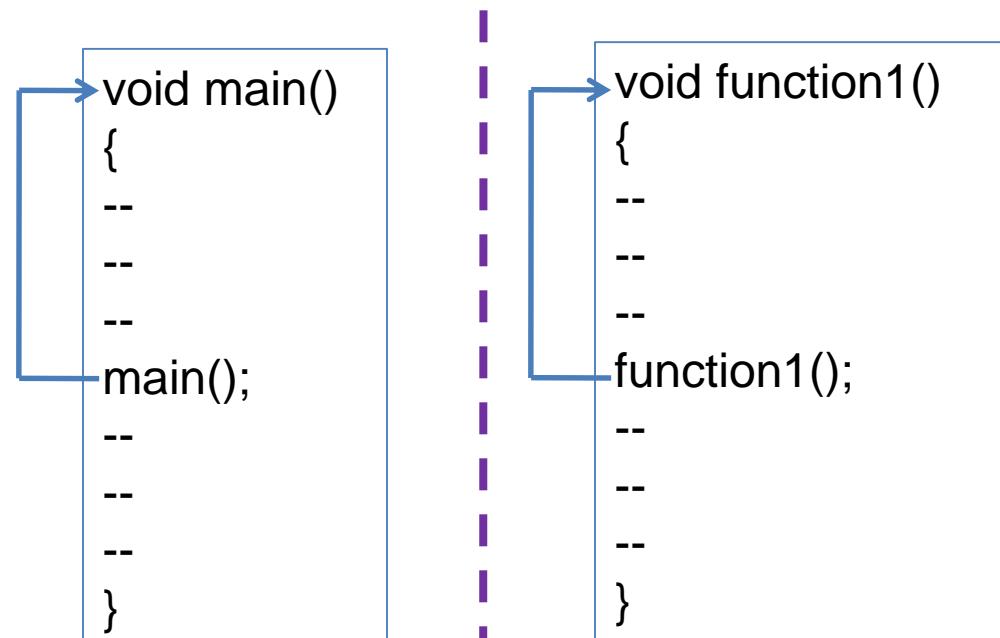


Recursion

Normal Function Call



Recursive Function Call





Recursion

- Function calls itself
- Terminating condition?
- Not applicable to all problems
- Shorter code
- Complex to understand



Recursion Implementation

- Example 1: Factorial [$n! = n * (n-1)!$]
- Example 2: Power $x^y=x * x^{y-1}$
- Example 3: Fibonacci Series

Next Term = Addition of Previous Two Terms



Recursion : Example 1 (factorial)

$$n! = n \times (n-1)!$$

$$5! = 5 \times 4!$$

$$5! = 5 \times 4 \times 3!$$

$$5! = 5 \times 4 \times 3 \times 2!$$

$$5! = 5 \times 4 \times 3 \times 2 \times 1!$$

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

$$5! = 120$$

Terminating Condition



Recursion : Example 1 (factorial)

www.hbpatel.in

```
#include <stdio.h>
void main()
{
    int fact(int);
    int x=5,ans;
    ans=fact(x);
    printf("Factorial = %d",ans);
}
int fact(int n)
{
    if (n<=1) return 1;
    else return (n * fact(n-1));
}

main()
{
    - ans=fact(5);
    - printf ans
    fact(n=5)
    {
        - return 5 x fact(4);
        fact(n=4)
        {
            - return 4 x fact(3);
            fact(n=3)
            {
                - return 3 x fact(2);
                fact(n=2)
                {
                    - return 2 x fact(1);
                    fact(n=1)
                    {
                        - return 1;
                    }
                }
            }
        }
    }
}
```

OUTPUT
Factorial = 120



Recursion : Example 2 ($\text{base}^{\text{exponent}}$)

$$b^e = b \times b^{e-1}$$

$$2^5 = 2 \times 2^4$$

$$2^5 = 2 \times 2 \times 2^3$$

$$2^5 = 2 \times 2 \times 2 \times 2^2$$

$$2^5 = 2 \times 2 \times 2 \times 2 \times 2^1$$

$$2^5 = 2 \times 2 \times 2 \times 2 \times 2 \times 2^0$$

$$2^5 = 2 \times 2 \times 2 \times 2 \times 2 \times 1$$

$$2^5 = 32$$

Terminating Condition

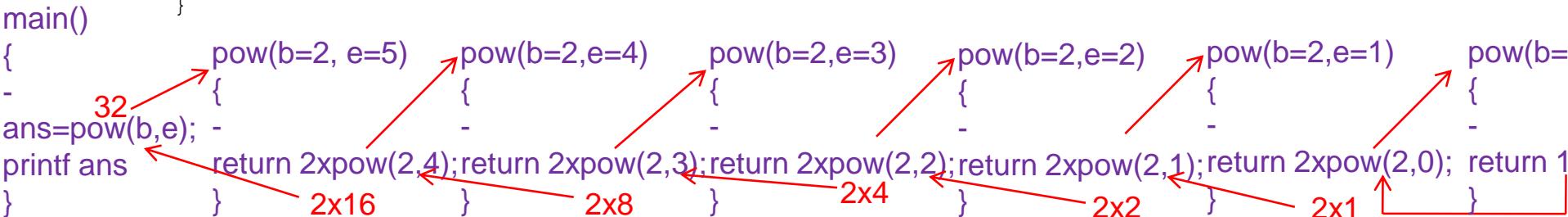


Recursion : Example 2 (base^{exponent})

www.hbpatel.in

```
#include <stdio.h>
void main()
{
    int pow(int,int);
    int b=2,e=5,ans;
    ans=pow(b,e);
    printf("%d power %d = %d",b,e,ans);
}
```

```
int pow(int b, int e)
{
    if(e<=0) return 1;
    else return (b * power(b,e-1));
}
```





Recursion : Example 3 (Fibonacci)

Last term (T_1) = 1 , Second Last Term (T_2) = 0

Next term (T_3) = $T_1 + T_2 = 1 + 0 = 1$

$T_1 \leftarrow T_2$ ($T_1=0$), $T_2 \leftarrow T_3$ ($T_2=1$)

Next term (T_3) = $T_1 + T_2 = 0 + 1 = 1$

$T_1 \leftarrow T_2$ ($T_1=1$), $T_2 \leftarrow T_3$ ($T_2=1$)

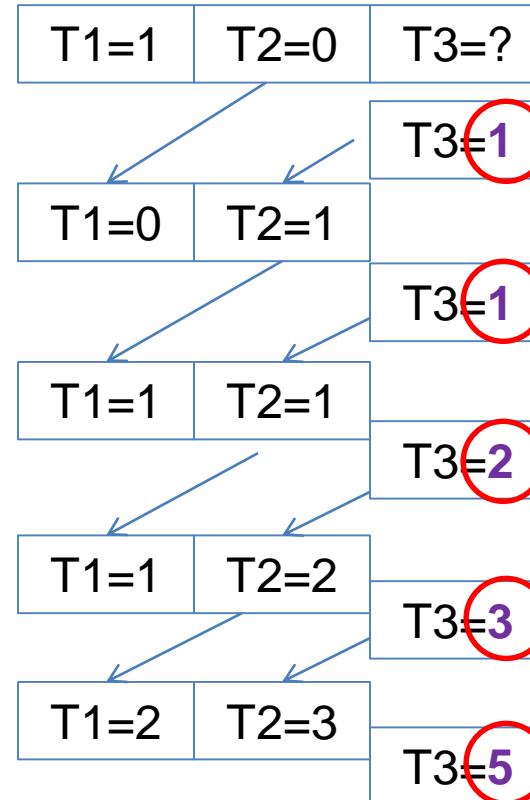
Next term (T_3) = $T_1 + T_2 = 1 + 1 = 2$

$T_1 \leftarrow T_2$ ($T_1=1$), $T_2 \leftarrow T_3$ ($T_2=2$)

Next term (T_3) = $T_1 + T_2 = 1 + 2 = 3$

$T_1 \leftarrow T_2$ ($T_1=2$), $T_2 \leftarrow T_3$ ($T_2=3$)

Next term (T_3) = $T_1 + T_2 = 2 + 3 = 5$

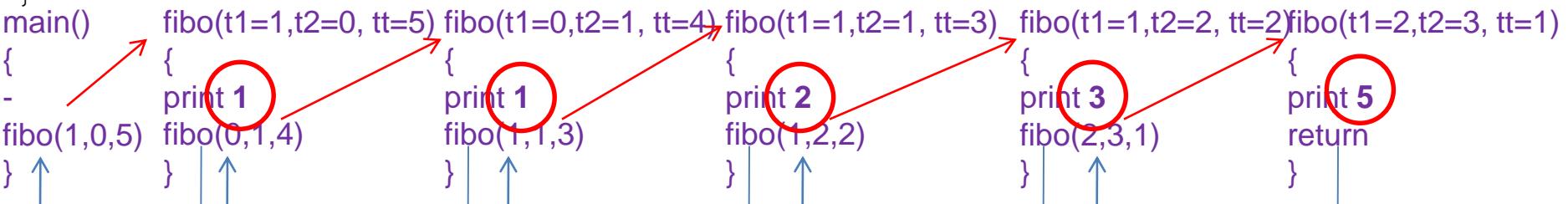




Recursion : Example 3 (Fibonacci)

```
#include <stdio.h>
void main()
{void fibo(int,int,int);
fibo(1,0,5); /* First term, second term, total terms */
}
```

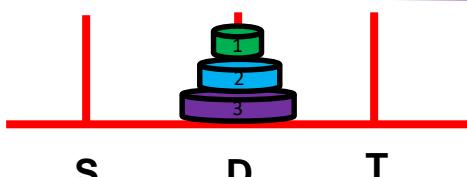
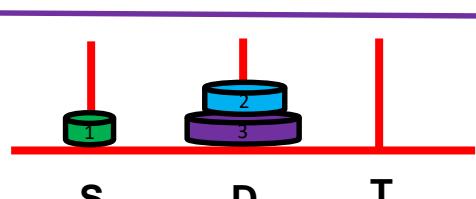
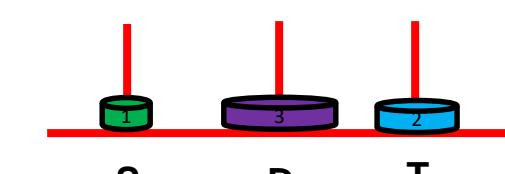
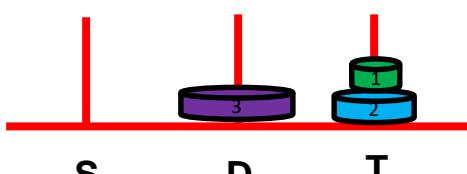
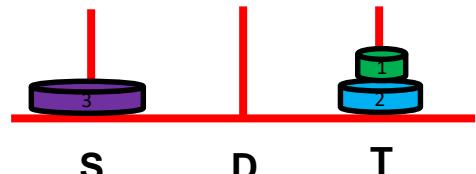
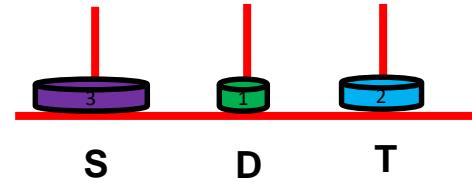
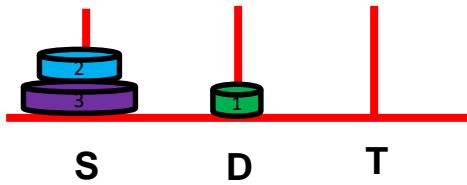
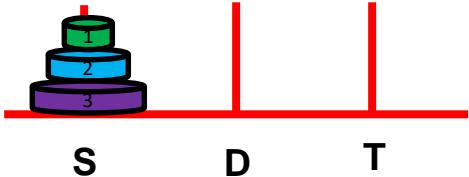
```
void fibo(int t1, int t2, int totalterms)
{
printf("%d\t",t1+t2);
if(totalterms<=1) {return;}
else {fibo(t2,t1+t2,totalterms-1);}
}
```



OUTPUT				
1	1	2	3	5



Recursion : Example 4 (Tower of Hanoi)



Move disk #2 from T to D

Move disk #1 from S to D

Recursion : Example 4 (Tower of Hanoi)

```
#include<stdio.h>
int main()
{  int n=3;
   TOH(n, 'S', 'D', 'T');
}
void TOH(int n, char source, char temp, char dest)
{  if(n>0)
   {
      TOH(n-1,source,dest,temp);
      printf("move disk # %d from %c to %c\n",n,source,temp);
      TOH(n-1,dest,temp,source);
   }
}
```

TOH (3, S, D, T)

TOH (2, S, T, D)

TOH (2, T, D, S)

TOH (1, S, D, T)

TOH (1, D, T, S)

TOH (1, T, S, D)

TOH (1, S, D, T)

(0, S, T, D) (0, T, D, S)

(0, D, S, T) (0, S, T, D)

(0, T, D, S) (0, D, S, T)

(0, S, T, D) (0, T, D, S)

OUTPUT

Move disk # 1 from S to D
Move disk # 2 from S to T
Move disk # 1 from D to T
Move disk # 3 from S to D
Move disk # 1 from T to S
Move disk # 2 from T to D
Move disk # 1 from S to D



Thank You

Next: Queue

Queue



Insertion

Rear / Tail



Deletion

Head / Front



Image source: alerttech.net





Queue

Linear data structure

Abstract data type

It is an ordered list

First In First Out (FIFO) – First Come First Serve (FCFS) - LILO

Insertion is known as Enqueue and Deletion is known as Dequeue



Stack Vs. Queue

	Stack	Queue
Data Structure	Linear	Linear
List	Ordered	Ordered
Type	LIFO	FIFO
End	One	Two
Insertion	Push	Enqueue
Deletion	Pop	Dequeue

Operations on Queue

Enqueue (x)

Dequeue

front() / peep ()

isFull()

isEmpty()

Size = 5

0 1 2 3 4

Front End



Rear End

0 1 2 3 4

front = rear = -1



Enqueue(2)



Enqueue(13)



Enqueue(-4)



Dequeue()

Enqueue(19)



Enqueue(10)



Enqueue(5)



Overflow

if front = rear = -1 then queue is empty
else if rear = size-1 then queue is full
else if front = rear then Only one element in the queue



Applications Queue

To buffer the requests received for a common shared resource (E.g. a printer shared in a network)

Instructions waiting for their turn to get executed by the processor

Queue Implementation (Using Arrays)

```
void enqueue(int element)
{
if(rear==size-1)
{
printf("Overflow\n");
}
else if(front==-1 && rear==-1)
{
front=rear=0;
queue[rear]=element;
}
else
{
rear++;
queue[rear]=element;
}
}
```

Queue Implementation (Using Arrays)

```
void dequeue(void)
{
if(front== -1 && rear== -1)
{
printf("Underflow\n");
}
else if(front==rear)
{
printf("Element deleted=%d\n",queue[front]);
front=rear=-1;
}
else
{
printf("Element deleted=%d\n",queue[front]);
front++;
}
}
```

Queue Implementation (Using Arrays)

```
void peep (void)
{
if(front== -1 && rear== -1)
    printf("Queue Empty\n");
else
    printf("Data at front=%d\n",queue[front]);
}
```

Queue Implementation (Using Arrays)

```
void display(void)
{
int i;
if(front== -1 && rear== -1)
printf("Queue Empty\n");
else
{
for(i=front; i<=rear; ++i){printf("%d\t",queue[i]);}
printf("\n");
}
}
```

Queue Implementation (Using Arrays)

```
#include <stdio.h>
#define size 5
int queue[size];
int front=-1, rear=-1;

int main()
{
    enqueue(10);
    enqueue(20);
    enqueue(-5);
    display();
    peep();
    dequeue();
    peep();
    display();
    return 0;
}

void enqueue(int element)
{
    if(rear==size-1){printf("Overflow\n");}
    else if(front==-1 && rear==-1){front=rear=0;queue[rear]=element;}
    else {rear++;queue[rear]=element;}
}

void dequeue(void)
{
    if(front==-1 && rear==-1){printf("Underflow\n");}
    else if(front==rear){printf("Element deleted=%d\n",queue[front]); front=rear=-1;}
    else {printf("Element deleted=%d\n",queue[front]); front++; }
}

void display(void)
{
    int i;
    if(front==-1 && rear==-1)printf("Queue Empty\n");
    else {for(i=front; i<=rear; ++i)printf("%d\t",queue[i]);printf("\n");}
}

void peep (void)
{
    if(front==-1 && rear==-1)printf("Queue Empty\n");
    else printf("Data at front=%d\n",queue[front]);
}
```

OUTPUT

10	20	-5
Data at front=10		
Element deleted=10		
Data at front=20		
20	-5	

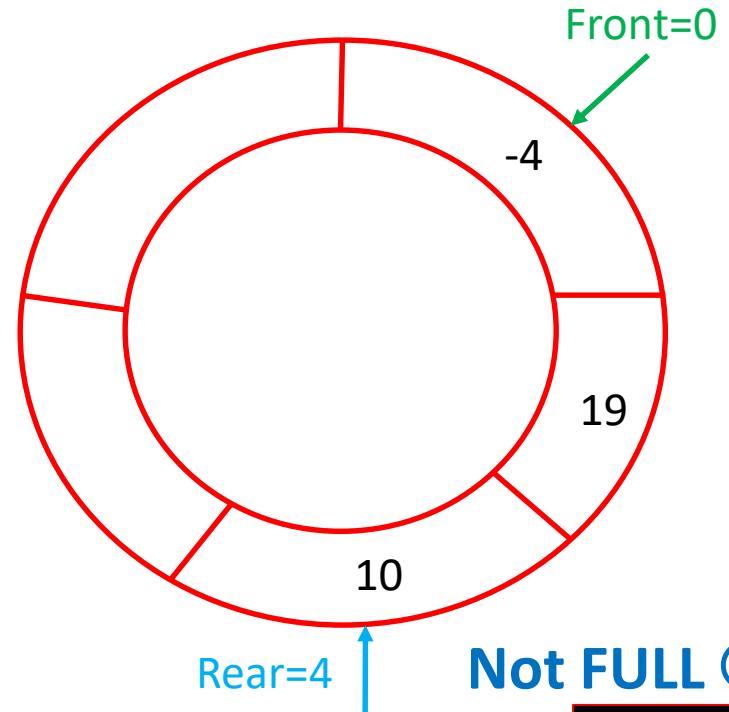
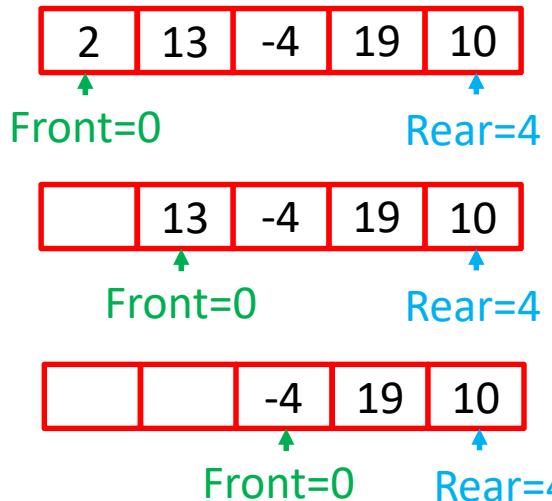


Thank You

Next: Circular Queue

Circular Queue

Normal Queue



Queue FULL 😞



Circular Queue

Drawback of regular linear queue is...

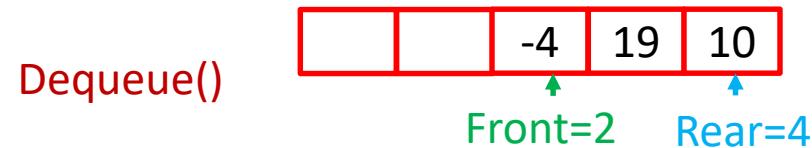


So, this queue is full !

```
void enqueue(int element)
{
    if(rear==size-1)
    {
        printf("Overflow\n");
    }
    ---
}
```



Still the queue is full !



Queue is full 😞



Circular Queue

If we can modify the condition little bit such that.....



size = 5

```
if ((rear+1)%size == front)
{
    --
}
```

What could be the condition?



Circular Queue

Previous

```
void enqueue(int element)
{
if(rear==size-1)
{
printf("Overflow\n");
}
else if(front===-1 && rear===-1)
{
front=rear=0;
queue[rear]=element;
}
else
{
rear++;
queue[rear]=element;
}
}
```

New

```
void enqueue(int element)
{
if((rear+1)%size==front)
{
printf("Overflow\n");
}
else if(front===-1 && rear===-1)
{
front=rear=0;
queue[rear]=element;
}
else
{
rear=(rear+1)%size;
queue[rear]=element;
}
}
```



Circular Queue

Previous

```
void dequeue(void)
{
if(front===-1 && rear===-1)
{
printf("Underflow\n");
}
else if(front==rear)
{
printf("Element deleted=%d\n",queue[front]);
front=rear=-1;
}
else
{
printf("Element deleted=%d\n",queue[front]);
front++;
}
}
```

New

```
void dequeue(void)
{
if(front===-1 && rear===-1)
{
printf("Underflow\n");
}
else if(front==rear)
{
printf("Element deleted=%d\n",queue[front]);
front=rear=-1;
}
else
{
printf("Element deleted=%d\n",queue[front]);
front=(front+1)%size;
}
}
```



Circular Queue

Previous

```
void display(void)
{
int i;
if(front===-1 && rear===-1)
    printf("Queue Empty\n");
else
{
    for(i=front; i<=rear; ++i)
        {
            printf("%d\t",queue[i]);
        }
    printf("\n");
}
}
```

New

```
void display(void)
{
int i=front;
if(front===-1 && rear===-1)
    printf("Queue Empty\n");
else
{
    while(i != rear)
        {
            printf("%d\t",queue[i]);
            i=(i+1)%size;
        }
    printf("\n");
}
}
```

Circular Queue Implementation (Using Arrays)

```
#include <stdio.h>
#define size 5
int queue[size];
int front=-1, rear=-1;

int main()
{
    enqueue(10);
    enqueue(20);
    enqueue(-5);
    display();
    peep();
    dequeue();
    peep();
    display();
    return 0;
}

void enqueue(int element)
{
    ----
}

void dequeue(void)
{
    ----
}

void display(void)
{
    ----
}

void peep (void)
{
    ----
}
```

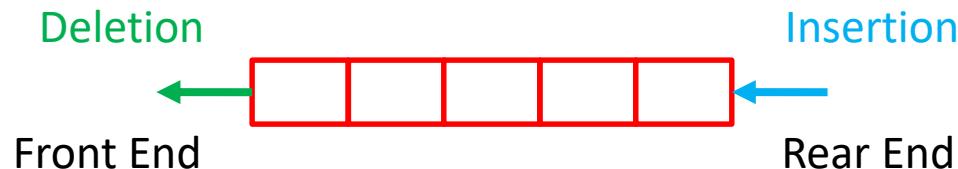


Thank You

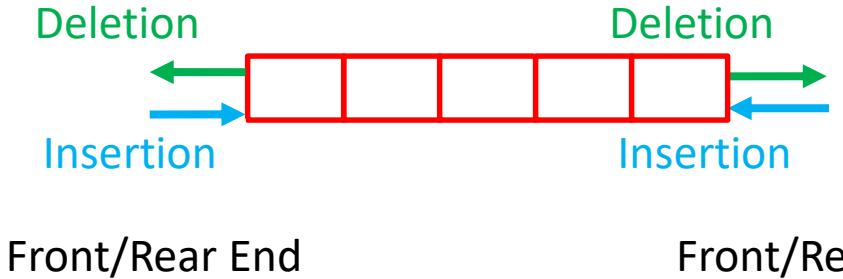
Next: Double Ended Queue (DQ)

Double Ended Queue

Normal Queue



Double Ended Queue



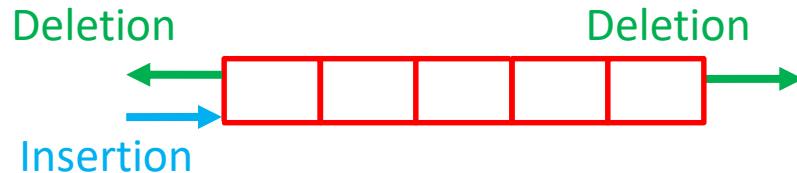
Double Ended Queue (Dequeue)

Properties of Dequeue

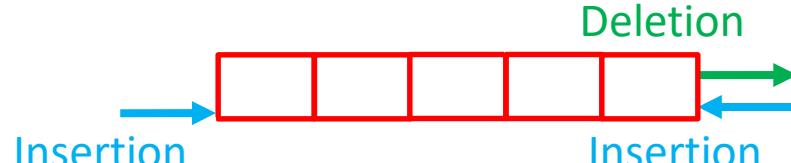
It is having properties of both, stack (LIFO) and queue (FIFO)

Two types of Dequeue: (a) Input restricted and (b) Output restricted

(a) Input restricted: Insertion is allowed from only one end, but deletion from both the ends



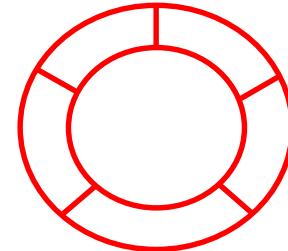
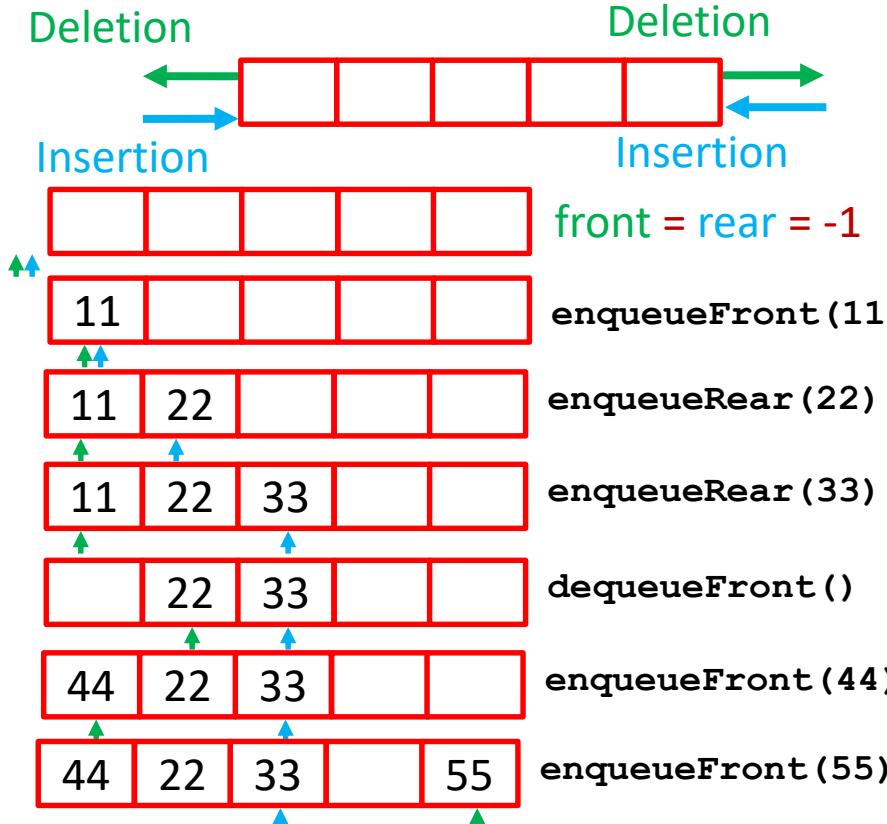
(b) Output restricted: Insertion is allowed from both end, but deletion from one end only



Operations on Dequeue: (i) Insert at front (ii) Delete from front (iii) Insert at rear and (iv) Delete from rear (v) getFront (vi) getRear



Dequeue Implementation



- enqueueFront ()
 - enqueueRear ()
 - dequeueFront ()
 - dequeueRear ()
- Operations:
- 44 22 33 **dequeueFront ()**
 - 44 22 **dequeueRear ()**
 - 44 22 66 **enqueueFront (66)**
 - 44 66 **dequeueRear ()**
 - 66 **dequeueRear ()**
 - dequeueFront ()**



Dequeue Implementation

```
void enqueueFront(int element)
{
if((front==0 && rear==size-1) || (front==rear+1)) {printf("Queue is full\n");}
else if(front== -1 && rear== -1) {front=rear=0; dequeue[front]=element;}
else if(front==0) {front=size-1; dequeue[front]=element;}
else {front--; dequeue[front]=element;}
}
```



Dequeue Implementation

```
void enqueueRear(int element)
{
    if((front==0 && rear==size-1) || (front==rear+1)) {printf("Queue is full\n");}
    else if(front==-1 && rear==-1) {front=rear=0; dequeue[rear]=element;}
    else if(rear==size-1) {rear=0; dequeue[rear]=element;}
    else {rear++; dequeue[rear]=element;}
}
```



Dequeue Implementation

```
void dequeueFront(void)
{
    if(front===-1 && rear===-1){printf("Queue is empty\n");}
    else if(front==rear) {printf("Element deleted = %d\n",dequeue[front]); front=rear=-1;}
    else if (front==size-1) {printf("Element deleted = %d\n",dequeue[front]); front=0;}
    else {printf("Element deleted = %d\n",dequeue[front]); front++;}
}
```



Dequeue Implementation

```
void dequeueRear(void)
{
    if(front== -1 && rear == -1){printf("Queue is empty\n");}
    else if(front==rear) {printf("Element deleted = %d\n",dequeue[rear]); front=rear=-1;}
    else if (rear==0) {printf("Element deleted = %d\n",dequeue[front]); rear=size-1;}
    else {printf("Element deleted = %d\n",dequeue[rear]); rear--;}
}
```



Dequeue Implementation

```
void display(void)
{
int i=front;
printf("Dequeue -> \t");
while (i != rear)
{
    printf("%d\t", dequeue[i]);
    i=(i+1)%size;
}
printf("%d\n", dequeue[rear]);
}
```

```
void getFront (void)
{
printf("Front Element = %d\n", dequeue[front]);
}
```

```
void getRear (void)
{
printf("Rear Element = %d\n", dequeue[rear]);
}
```



Dequeue Implementation

```
#include <stdio.h>
#define size 5
int dequeue[size];
int front=-1, rear=-1;
int main()
{
enqueueFront(11);
enqueueRear(22);
enqueueRear(33);
display();
dequeueFront();
enqueueFront(44);
enqueueFront(55);
getFront();
display();
dequeueFront();
dequeueRear();
dequeueRear();
enqueueFront(66);
display();
getRear();
dequeueRear();
dequeueFront();
display();
return 0;
}
```

OUTPUT

```
Dequeue ->      11      22      33
Element deleted = 11
Front Element = 55
Dequeue ->      55      44      22      33
Element deleted = 55
Element deleted = 33
Element deleted = 22
Dequeue ->      66      44
Rear Element = 44
Element deleted = 44
Element deleted = 66
Dequeue ->      0
```



Thank You

Next: Priority Queue



Priority Queue

Elements	Priority
A	2
B	4
C	1
D	5
E	3

Highest
Lowest

Without Priority



With Priority



Implementation

1. Arrays
2. Linked List
3. Heap



Array Implementation of PQ

Elements	A	B	C	D	E
Priority	2	4	1	5	3

Insertion

A					
2					

C	A	B	D	
1	2	4	5	

A	B				
2	4				

C	A	E	B	D
1	2	3	4	5

C	A	B			
1	2	4			

Deletion (E, 3)

C	A	E	B	D
1	2	3	4	5

C	A	B	D	
1	2	4	5	



Array Implementation of PQ

```
void PQInsert(char newElement, int newPriority)
{
    int count=0, k;
    while (count < totalElement)
    {
        if(newPriority>=PQPriority[count]) count++;
        else break;
    }
    if(count!=totalElement)
        for(k=totalElement; k>count; --k)
        {
            PQElement[k]=PQElement[k-1];
            PQPriority[k]=PQPriority[k-1];
        }
    PQElement[count]=newElement;
    PQPriority[count]=newPriority;
    totalElement++;
}
```



Array Implementation of PQ

```
void PQDelete(char deleteElement, int deletePriority)
{
    int count=0,i;
    while (count < totalElement)
    {
        if(PQElement[count]==deleteElement&&PQPRIORITY[count]==deletePriority)break;
        else count++;
    }
    if(count==totalElement)printf("The element to delete did not find in the list\n");
    else
    {
        for(i=count; i<totalElement; ++i)
        {
            PQElement[i]=PQElement[i+1];
            PQPRIORITY[i]=PQPRIORITY[i+1];
        }
        totalElement--;
        printf("Element %c deleted\n",deleteElement);
    }
}
```



Array Implementation of PQ

```
#include <stdio.h>
#define size 5
char PQElement[size*2];
int PQPriority[size*2];
int totalElement = 0;
int main()
{
    void PQInsert(char, int);
    void PQDelete(char, int);
    void display (void);
    PQInsert('A',2);
    PQInsert('B',4);
    PQInsert('C',1);
    PQInsert('D',5);
    PQInsert('E',3);
    PQInsert('F',3);
    display();
    PQDelete('C',1);
    display();
    return 0;
}
```

```
void display(void)
{
    int i;
    for(i=0; i<totalElement; ++i)
    {
        printf("Element: %c Priority : %d\n", PQElement[i],PQPriority[i]);
    }
}
```

OUTPUT

```
Element: C Priority: 1
Element: A Priority: 2
Element: E Priority: 3
Element: F Priority: 3
Element: B Priority: 4
Element: D Priority: 5
Element C deleted
Element: A Priority: 2
Element: E Priority: 3
Element: F Priority: 3
Element: B Priority: 4
Element: D Priority: 5
```



Thank You

Next: Linked List



Linked List

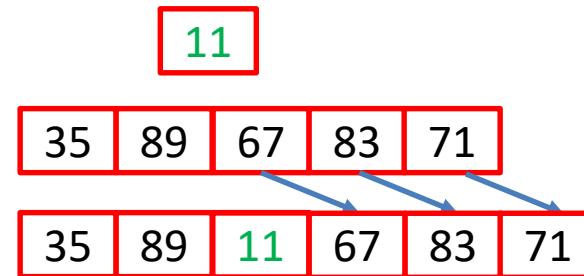
Static Memory Allocation

```
int marks [5] = {35, 89, 67, 83, 71};
```

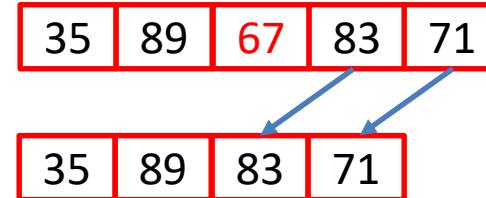
marks	35	89	67	83	71
	0	1	2	3	4

marks[0]	35	1000
marks[1]	89	1002
marks[2]	67	1004
marks[3]	83	1006
marks[4]	71	1008

Insertion



Deletion





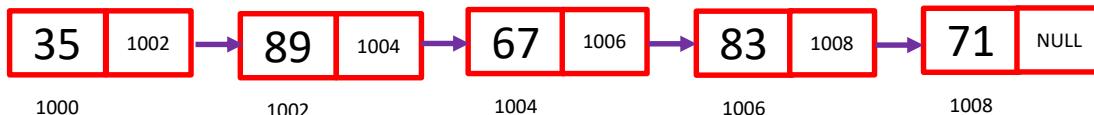
Linked List

A linked list is a linear data structure consisting of a group of nodes where each node point to the next node by means of a pointer. Each node is composed of data and a reference (in other words, a link) to the next node in the sequence.



Linked List

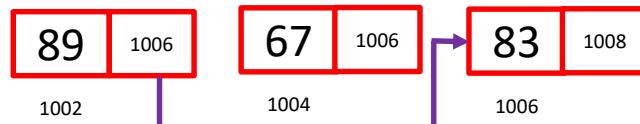
Dynamic Memory Allocation



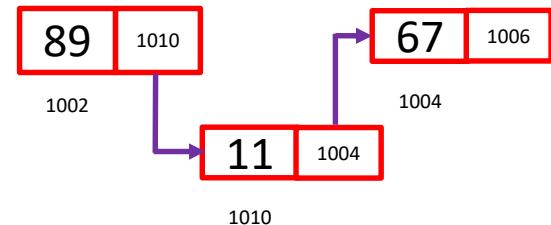
node

Data	Address of Next Node
------	-------------------------

Deletion



Insertion





Linked List

Static Memory Allocation

- Memory is allocated at compile time (before running/execution).
- Maximum size is fixed
- Example: Array
- **Advantage:** Fast (random) access. **Disadvantage:** Slower insertion/deletion, memory waste

Dynamic Memory Allocation

- Memory is allocated at run time
- Maximum size is flexible (can be increased/reduced)
- Example: Linked list
- **Advantage:** Faster insertion/deletion, Memory save **Disadvantage:** Slow (linear) access, Computation overhead

Linked List

Singly Linked List



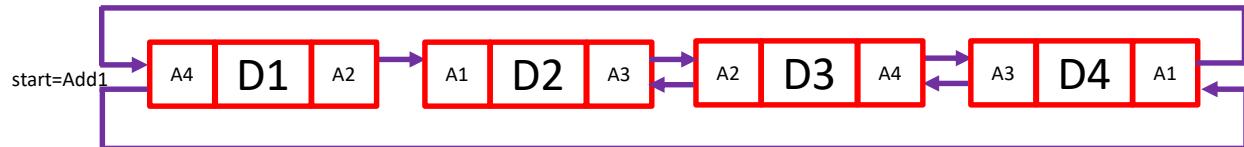
Circular Linked List



Doubly Linked List



Doubly Circular Linked List



Linked List

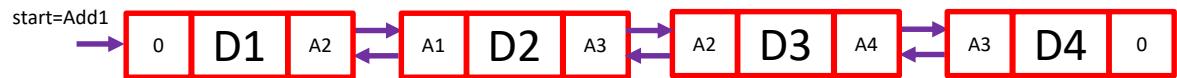
Singly Linked List



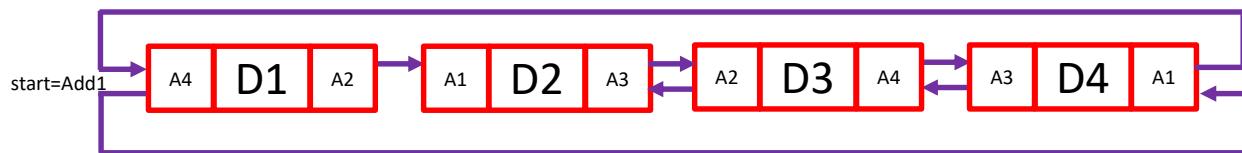
Circular Linked List



Doubly Linked List



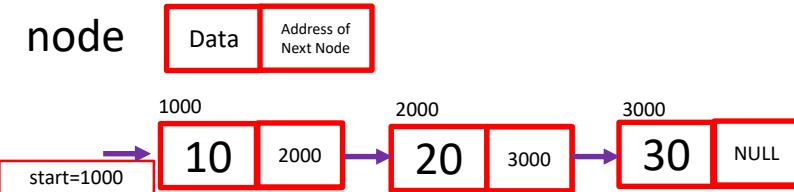
Doubly Circular Linked List





How to create/access structure members?

node



struct node

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

```
struct node *start=NULL, *newnode, *temp;
newnode = (struct node*)malloc (sizeof(struct node));
printf("Enter data : ");
scanf("%d", &newnode->data);
newnode->next=NULL;
```



Singly Linked List (Initialization)

```
#include <stdio.h>
struct node
{
    int data;
    struct node *next;
};

int main ()
{
    struct node *start=NULL, *newnode, *temp;
    int choice;
```



Singly Linked List (Creation)

```
do
{
newnode = (struct node*)malloc (sizeof(struct node));
printf("Enter data : ");
scanf("%d",&newnode->data);
newnode->next=NULL;
if(start==NULL) {start = temp = newnode;}
else
{
    temp->next = newnode;
    temp=newnode;
}
printf("Do you wish to continue? (1 For Yes / 0 For No) : ");
scanf("%d",&choice);
}
while(choice != 0);
```



Singly Linked List (Display)

```
temp = start;
while (temp != NULL)
{
    printf("%d\t",temp->data);
    temp=temp->next;
}
}
```



Singly Linked List (Whole Program)

```
#include <stdio.h>
struct node
{
    int data;
    struct node *next;
};

int main ()
{
    struct node *start=NULL, *newnode, *temp;
    int choice;
    do
    {
        newnode = (struct node*)malloc (sizeof(struct node));
        printf("Enter data : ");
        scanf("%d",&newnode->data);
        newnode->next=NULL;
        if(start==NULL) {start = temp = newnode;}
        else
        {
            temp->next = newnode;
            temp=newnode;
        }
        printf("Do you wish to continue? (1 For Yes / 0 For No) : ");
        scanf("%d",&choice);
    }while(choice != 0);
```

```
temp = start;
while (temp != NULL)
{
    printf("%d\t",temp->data);
    temp=temp->next;
}
```

```
Enter data : 10
Do you wish to continue? (1 For Yes / 0 For No) : 1
Enter data : 20
Do you wish to continue? (1 For Yes / 0 For No) : 1
Enter data : 30
Do you wish to continue? (1 For Yes / 0 For No) : 0
10      20      30
```



Thank You

Next: Linked List (Insertion)



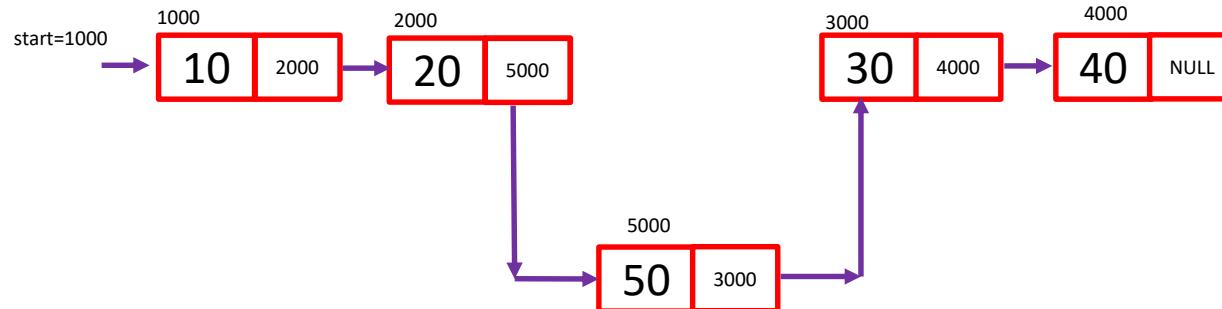
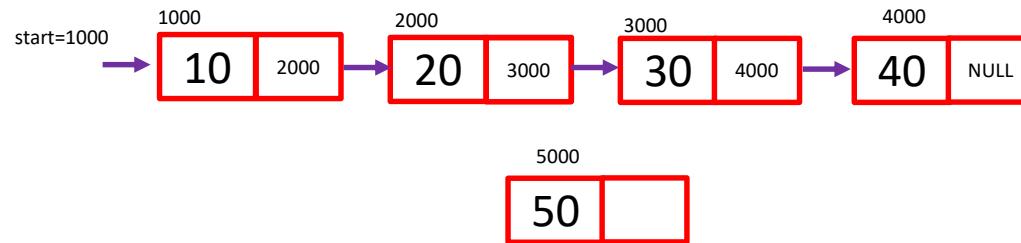
Insertion in Singly Linked List

Insertion in the Beginning

Insertion at the End

Insertion in between (after *pos*)

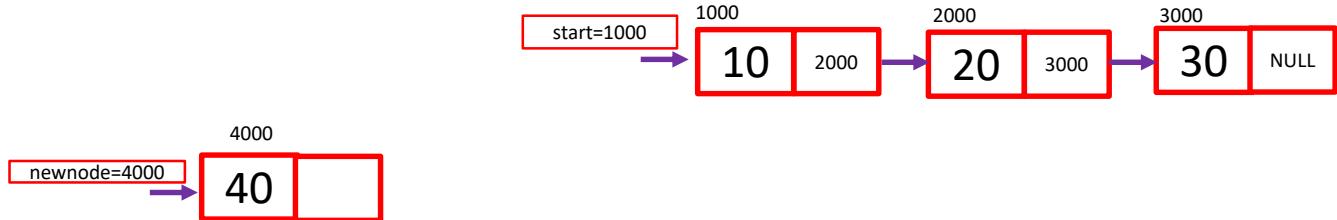
Insertion in Singly Linked List





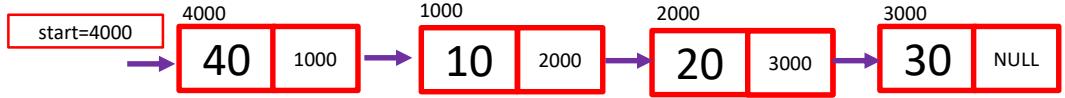
Insertion in Singly Linked List

Insertion in the Beginning



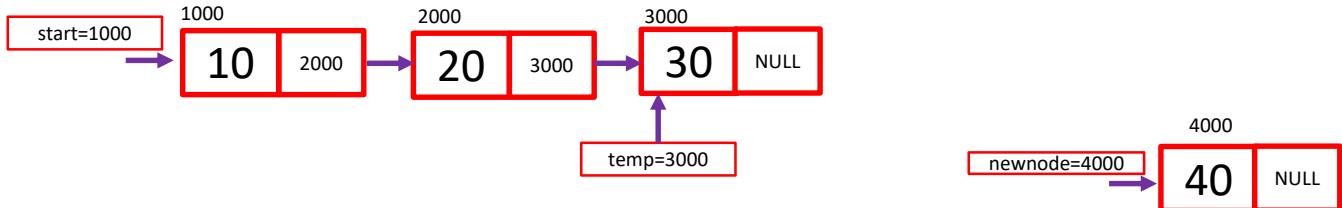
```
newnode = (struct node*) malloc (sizeof (struct node));
```

```
void insertBegin(void)
{
    newnode->next=start;
    start=newnode;
}
```



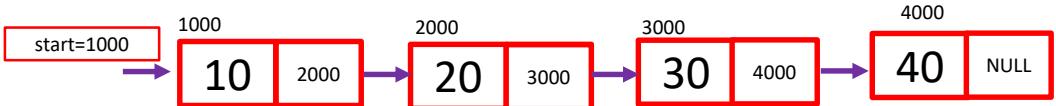
Insertion in Singly Linked List

Insertion at the End



```
newnode = (struct node*) malloc (sizeof (struct node));
```

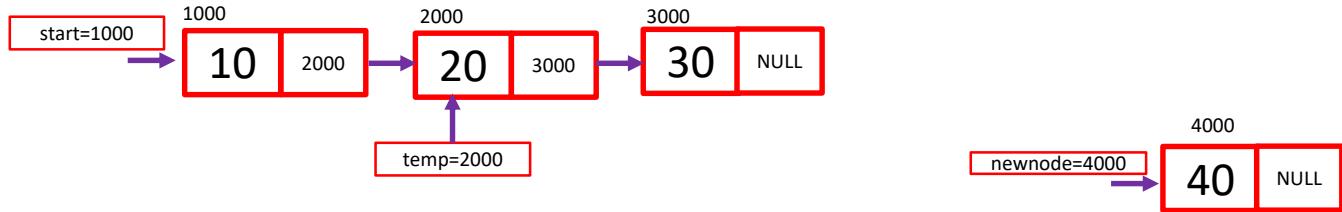
```
void insertEnd(void)
{
temp = start;
while (temp->next != NULL)
{
    temp=temp->next;
}
temp->next = newnode;
}
```





Insertion in Singly Linked List

Insertion in between (after pos)



```
void insertPos(int p)
{
int count=1;
temp=start;
while (count < p)
{
    {
        temp=temp->next;
        count++;
    }
newnode->next = temp->next;
temp->next = newnode;
}
```

```
newnode = (struct node*) malloc (sizeof (struct node));
pos=2;
```





Insertion in Singly Linked List

```
#include <stdio.h>
struct node
{
    int data;
    struct node *next;
};

int totalNode=0;
struct node *start=NULL, *newnode, *temp;
int main ()
{
    int choice, pos;

    do
    {
        printf("\n1: Insert in beginning\n");
        printf("2: Insert at end\n");
        printf("3: Insert after a position\n");
        printf("4: Display\n");
        printf("5: Exit\n");
        printf("Enter your choice : ");
        scanf("%d", &choice);
        if(choice==1 || choice==2 || choice==3)
        {
            newnode = (struct node*) malloc (sizeof (struct node));
            printf("Enter data : ");
            scanf("%d",&newnode->data);
            newnode->next=NULL;
        }
        switch(choice)
        {
            case 1:      insertBegin();break;
            case 2:      insertEnd();break;
            case 3:      printf("Enter position: ");
                          scanf("%d",&pos);
                          insertPos(pos);break;
            case 4:      display();break;
            case 5:      break;
            default:     printf("Invalid Choice...Please try again...\n");
        }
    }while(choice != 5);
}
```



Insertion in Singly Linked List

```
void insertBegin(void)
{
newnode->next=start;
start=newnode;
}
```

```
void insertPos(int p)
{
int count=1;
temp=start;
while (count < p)
{
    temp=temp->next;
    count++;
}
newnode->next = temp->next;
temp->next = newnode;
}
```

```
void insertEnd(void)
{
temp = start;
while (temp->next != NULL)
{
    temp=temp->next;
}
temp->next = newnode;
}
```

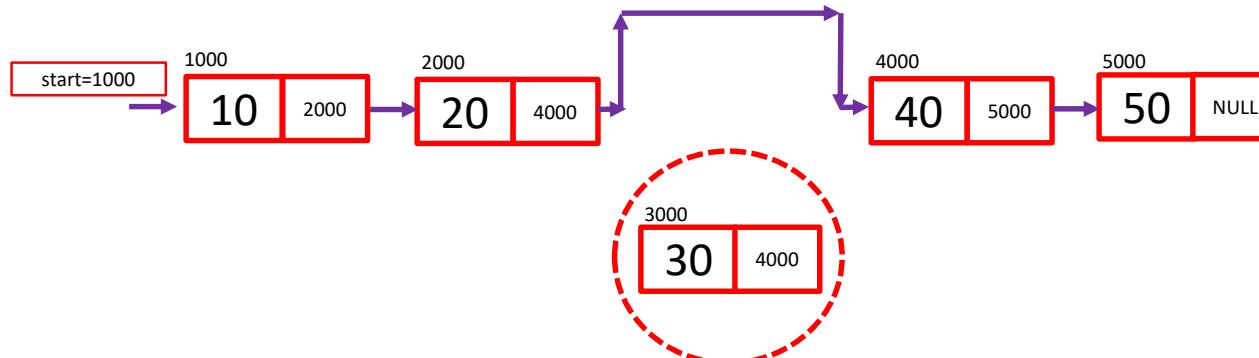
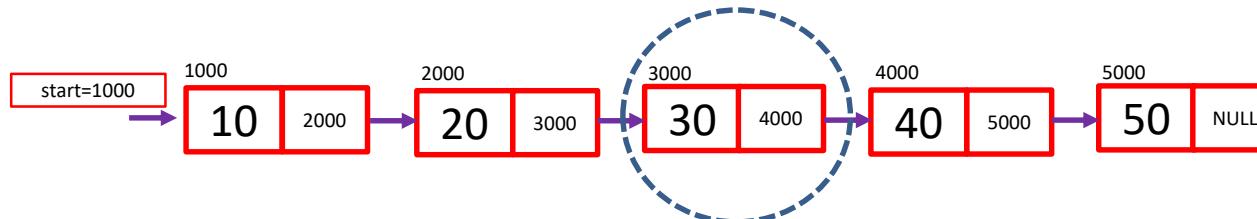
```
void display(void)
{
temp=start;
while(temp!=NULL)
{
    printf("%d\t",temp->data);
    temp=temp->next;
}
}
```



Thank You

Next: Linked List (Deletion)

Deletion from Singly Linked List





Deletion from Singly Linked List

Deletion from the Beginning

Deletion from the End

Deletion from specific *pos*



Deletion from Singly Linked List

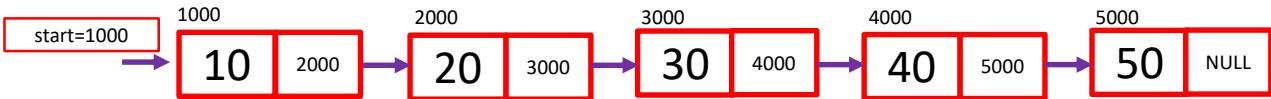
Insert at the End

```
void insertEnd(int element)
{
newnode = (struct node *) malloc (sizeof(struct node));
newnode -> data = element;
newnode -> next = NULL;
if(start==NULL)
{
    start=newnode;
}
else
{
    temp = start;
    while (temp->next != NULL)
    {
        temp=temp->next;
    }
    temp->next = newnode;
}
}
```



Deletion from Singly Linked List

Deletion from the Beginning



```
void deleteBeginning ()  
{  
if(start == NULL)  
{  
    printf("No Element in the List\n");  
}  
else  
{  
    temp = start;  
    start = start -> next;  
    free (temp);  
}  
}
```

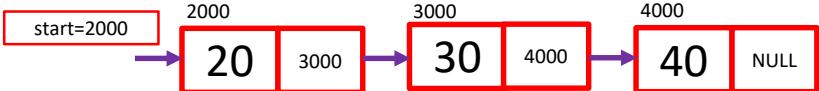
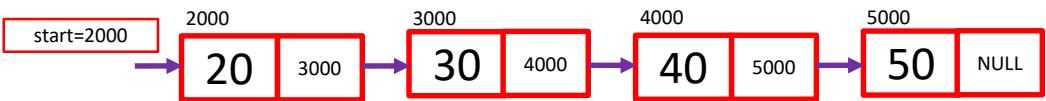




Deletion from Singly Linked List

Deletion from the End

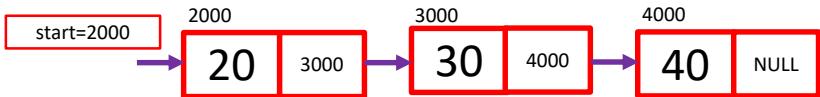
```
void deleteEnd ()  
{  
temp = start;  
if (temp->next == NULL)  
{  
    start = NULL;  
    free (temp);  
}  
else  
{  
    while (temp -> next -> next != NULL)  
    {  
        temp = temp -> next;  
    }  
    free (temp->next);  
    temp->next=NULL;  
}  
}
```





Deletion from Singly Linked List

Deletion from specific pos



```
void deletePosition (int p)
{
    temp = start;
    int i;
    for(i=1; i<p-1; ++i)
    {
        temp = temp -> next;
    }
    nextnode = temp -> next;
    temp -> next = nextnode -> next;
    free (nextnode);
}
```





Deletion from Singly Linked List

```
#include <stdio.h>
struct node
{
    int data;
    struct node *next;
};
struct node *start=NULL, *newnode,
int main ()
{
    insertEnd(10);
    insertEnd(20);
    insertEnd(30);
    insertEnd(40);
    insertEnd(50);
    display();
    deleteBeginning();
    display();
    deleteEnd();
    display();
    deletePosition(2);
    display();
}
```

```
void insertEnd(int element)
```

```
{
---
```

C:\TURBOC3\BIN\ldelete.exe

10	20	30	40	50
20	30	40	50	
20	30	40		
20	40			

```
{
---
```

```
}
```

```
void display(void)
```

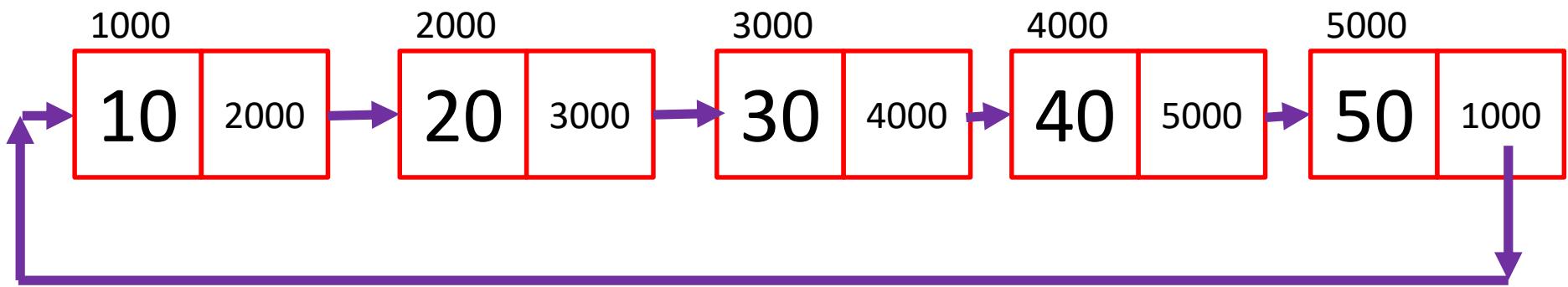
```
{
---
```



Thank You

Next: Circular Linked List

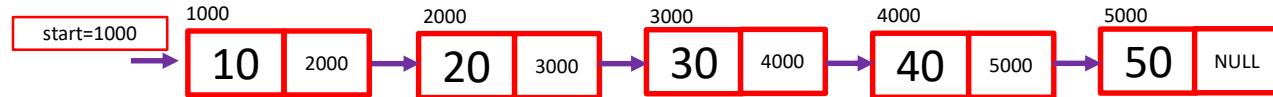
Circular Linked List



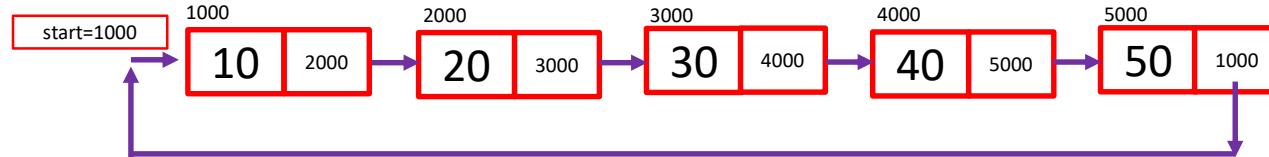


Circular Linked List

Singly Linked List



Circular Linked List





Circular Linked List

```
void createCircularLinkedList(void)
{
    struct node * newnode;
    int choice=1;
    while (choice != 0)
    {
        newnode = (struct node *)malloc(sizeof(struct node));
        printf("Enter data to insert : ");
        scanf("%d",&newnode->data);
        newnode->next=NULL;
        if (start==NULL)start=temp=newnode;
        else
        {
            temp->next=newnode;
            temp=newnode;
        }
        temp->next=start;
        printf("Continue? 1: Yes, 0: No => ");
        scanf("%d",&choice);
    }
}
```



Circular Linked List

```
#include <stdio.h>
struct node {
    int data;
    struct node *next;
};
struct node *start=NULL, *temp;
void main ()
{
    void createCircularLinkedList(void);
    void display(void);
    createCircularLinkedList();
    display();
}
void display(void)
{
    temp=start;
    do
    {
        printf("%d\t",temp->data);
        temp=temp->next;
    }while(temp != start);
    printf("\n");
}
void createCircularLinkedList(void)
{--}
```

```
D:\Personal\MyLectures\DSA\Programs\llcircular.exe
Enter data to insert : 10
Continue? 1: Yes, 0: No => 1
Enter data to insert : 20
Continue? 1: Yes, 0: No => 1
Enter data to insert : 30
Continue? 1: Yes, 0: No => 1
Enter data to insert : 40
Continue? 1: Yes, 0: No => 0
10      20      30      40
```



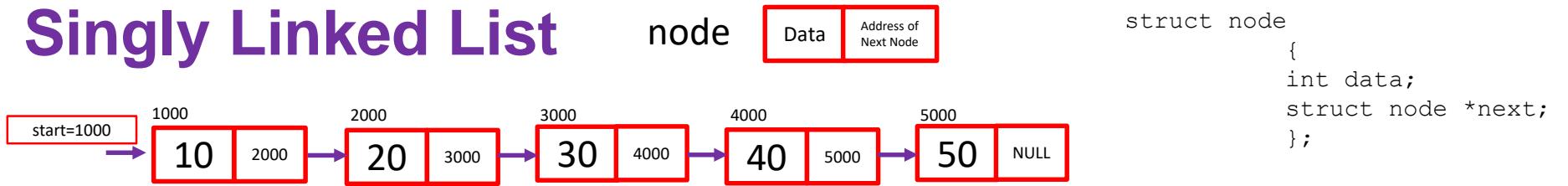
Thank You

Next: Doubly Linked List

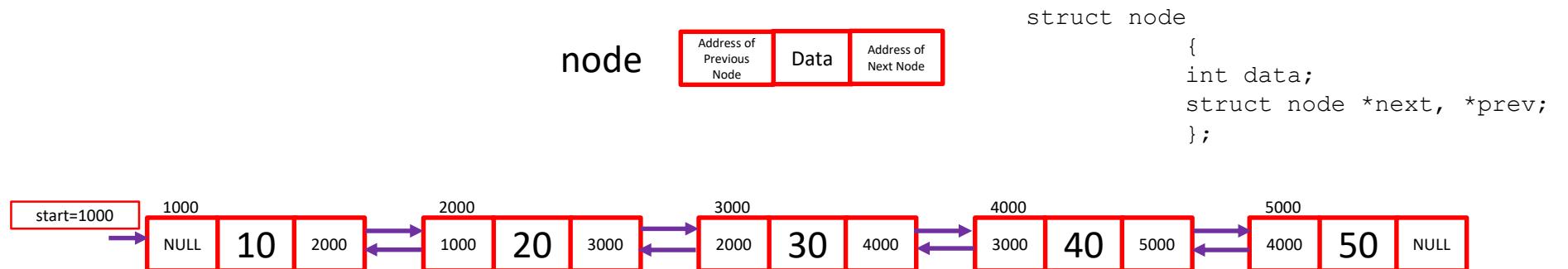


Doubly Linked List

Singly Linked List



Doubly Linked List





Doubly Linked List

```
void createDoublyLinkedList(void)
{
struct node * newnode;
int choice=1;
while (choice != 0)
{
    newnode = (struct node *)malloc(sizeof(struct node));
    printf("Enter data to insert : ");
    scanf("%d",&newnode->data);
    newnode->next=newnode->prev=NULL;
    if (start==NULL)start=temp=newnode;
    else
    {
        temp->next=newnode;
        newnode->prev=temp;
        temp=newnode;
    }
    printf("Continue? 1: Yes, 0: No => ");
    scanf("%d",&choice);
}
```



Doubly Linked List

```
#include <stdio.h>
struct node {
    int data;
    struct node *next, *prev;
};
struct node *start=NULL, *temp;
void main ()
{
    createDoublyLinkedList();
    display();
}
void createDoublyLinkedList(void)
{--}
void display(void)
{
    temp=start;
    do
    {
        printf("%d\t",temp->data);
        temp=temp->next;
    }while(temp != NULL);
    printf("\n");
}
```

```
D:\Personal\MyLectures\DSA\Programs\lldoubly.exe
Enter data to insert : 10
Continue? 1: Yes, 0: No => 1
Enter data to insert : 20
Continue? 1: Yes, 0: No => 1
Enter data to insert : 30
Continue? 1: Yes, 0: No => 1
Enter data to insert : 40
Continue? 1: Yes, 0: No => 0
10      20      30      40
```



Thank You

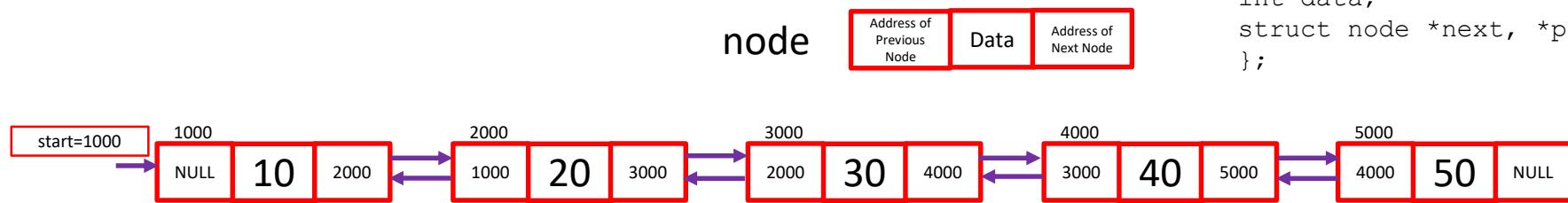
Next: Doubly Circular Linked List



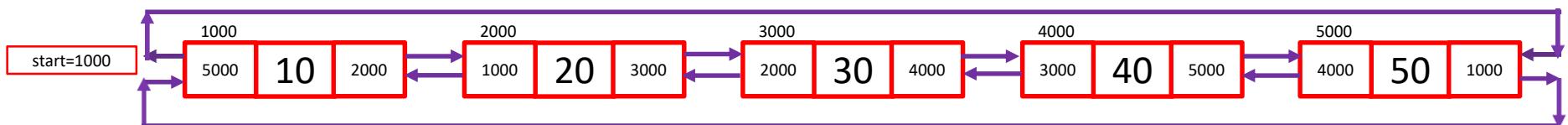
Doubly Circular Linked List

Doubly Linked List

```
struct node
{
    int data;
    struct node *next, *prev;
};
```



Doubly Circular Linked List





Doubly Circular Linked List

```
#include <stdio.h>
struct node
{
    int data;
    struct node *next, *prev;
};
struct node *start=NULL, *end=NULL, *temp;
void main ()
{
    createDoublyCircularLinkedList();
    displayForward();
    displayBackward();
}
```



Doubly Circular Linked List

```
void createDoublyCircularLinkedList(void)
{struct node * newnode;
int choice=1;
while (choice != 0)
{
    newnode = (struct node *)malloc(sizeof(struct node));
    printf("Enter data to insert : ");
    scanf("%d",&newnode->data);
    newnode->next=newnode->prev=NULL;
    if (start==NULL){start=end=newnode;
                      start->next=start;
                      start->prev=start;
                      }
    else           {end->next=newnode;
                    newnode->prev=end;
                    newnode->next=start;
                    start->prev=newnode;
                    end=newnode;
                    }
    printf("Continue? 1: Yes, 0: No => ");
    scanf("%d",&choice);
}
```

D:\Personal\MyLectures\DSA\Programs\lldblcirc.exe

```
Enter data to insert : 10
Continue? 1: Yes, 0: No => 1
Enter data to insert : 20
Continue? 1: Yes, 0: No => 1
Enter data to insert : 30
Continue? 1: Yes, 0: No => 1
Enter data to insert : 40
Continue? 1: Yes, 0: No => 0
10      20      30      40
40      30      20      10
```



Doubly Circular Linked List

```
void displayForward(void)
{
    temp=start;
    do
    {
        printf("%d\t",temp->data);
        temp=temp->next;
    }while(temp != start);
    printf("\n");
}

void displayBackward(void)
{
    temp=end;
    do
    {
        printf("%d\t",temp->data);
        temp=temp->prev;
    }while(temp != end);
    printf("\n");
}
```

D:\Personal\MyLectures\DSA\Programs\lldblcirc.exe

```
Enter data to insert : 10
Continue? 1: Yes, 0: No => 1
Enter data to insert : 20
Continue? 1: Yes, 0: No => 1
Enter data to insert : 30
Continue? 1: Yes, 0: No => 1
Enter data to insert : 40
Continue? 1: Yes, 0: No => 0
10      20      30      40
40      30      20      10
```



Thank You

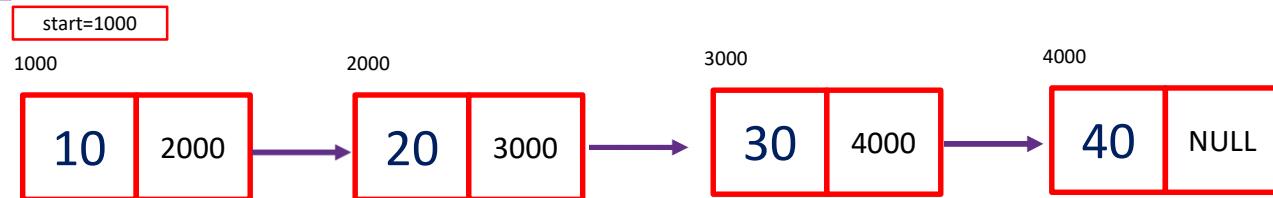
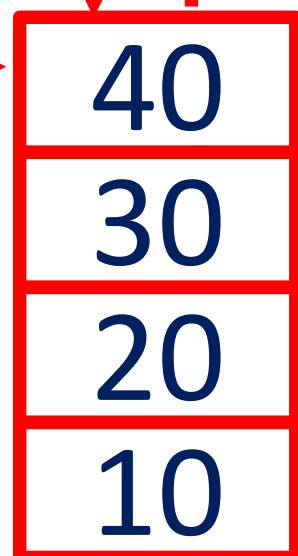
**Next: Stack Implementation
using Linked List**



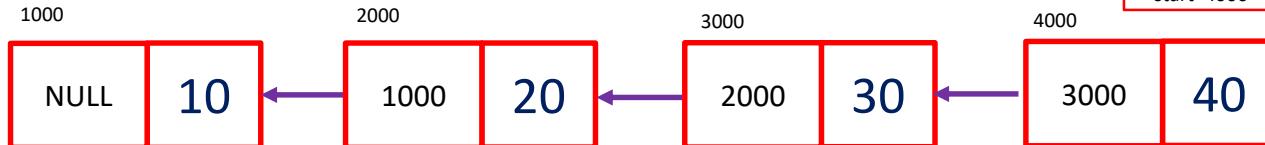
Stack Implementation using Linked List

Push Pop

Stack Top

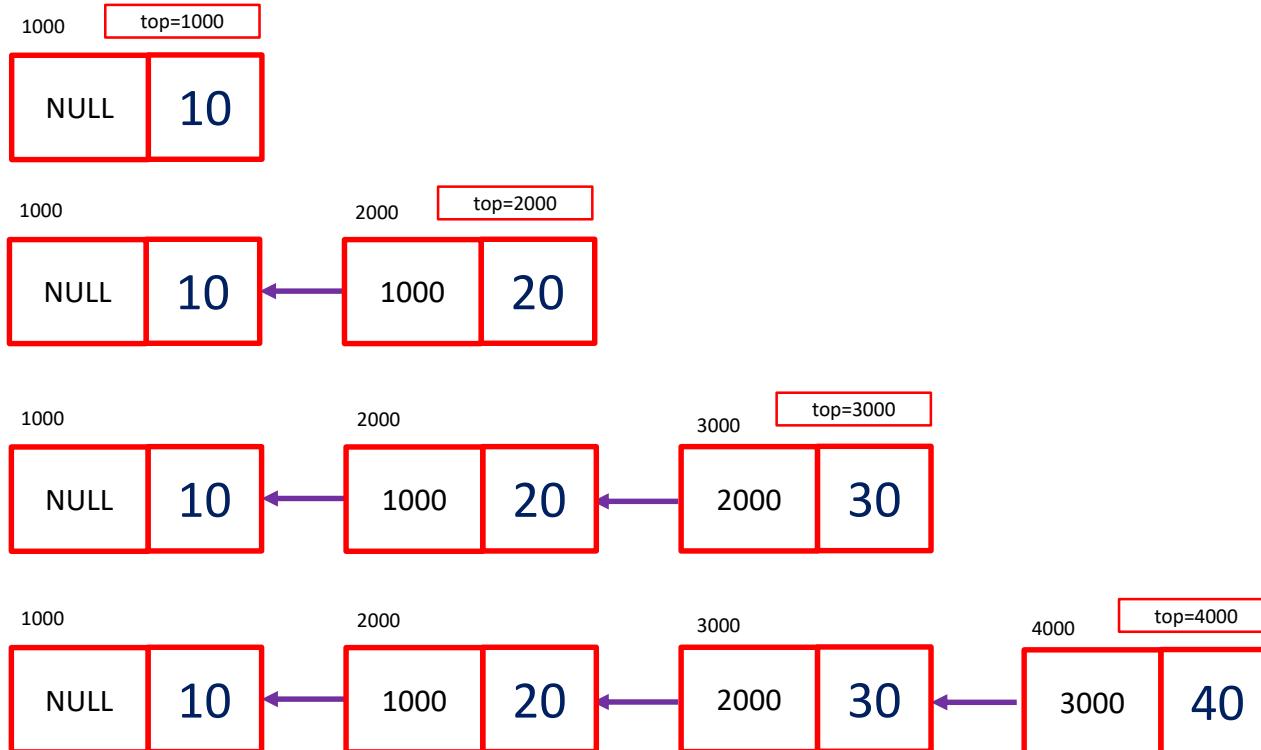


What we want !





Stack Implementation using Linked List





Stack Implementation using Linked List

```
#include <stdio.h>
struct node
{
    int data;
    struct node *prev;
};
struct node *top=NULL;
void main ()
{
    push(10);
    push(20);
    push(30);
    push(40);
    display();
    pop();
    display();
    peep();
}
```

```
D:\Personal\MyLectures\DSA\Programs\stackll.exe
40      30      20      10
30      20      10
Element on the top = 30
30      20      10
```



Stack Implementation using Linked List

```
void push(int item)
{
    struct node *newnode = (struct node *)malloc(sizeof(struct node));
    newnode->data=item;
    newnode->prev=top;
    top=newnode;
}
```



Stack Implementation using Linked List

```
void pop(void)
{
    struct node *temp=top;
    if(top==NULL){printf("Stack Empty"); return;}
    else {top=top->prev;free(temp);}
}

void peep(void)
{
    if(top==NULL){printf("Stack Empty"); return;}
    else {printf("Element on the top = %d\n",top->data);}
}
```



Stack Implementation using Linked List

```
void display(void)
{
    struct node *temp=top;
    while(temp!=NULL)
    {
        printf("%d\t",temp->data);
        temp=temp->prev;
    }
    printf("\n");
}
```

D:\Personal\MyLectures\DSA\Programs\stackll.exe

```
40      30      20      10
30      20      10
Element on the top = 30
30      20      10
```



Thank You

**Next: Queue Implementation
using Linked List**



Queue Implementation using Linked List

Insertion

Rear / Tail

Deletion

Head / Front

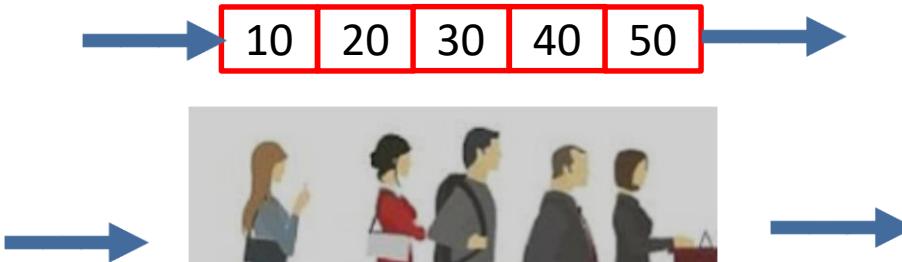


Image source: alerttech.net





Queue Implementation using Linked List

```
struct node  
{int data;  
struct node *next;  
};  
struct node *front=NULL, *rear=NULL;  
  
void enqueue(int item)  
{  
    struct node *newnode = (struct node *)malloc(sizeof(struct node));  
    newnode->data=item;  
    newnode->next=NULL;  
    if(front==NULL && rear==NULL)front=rear=newnode;  
    else {rear->next=newnode; rear=newnode;}  
}
```



Queue Implementation using Linked List

```
void dequeue(void)
{
    struct node *temp=front;
    if(front==NULL){printf("Queue Empty"); return;}
    else {front=front->next;free(temp);}
}
```



Queue Implementation using Linked List

```
void peep(void)
{
    if(front==NULL){printf("Queue Empty"); return;}
    else
    {
        printf("Element at the front = %d\n",front->data);
        printf("Element at the end = %d\n",rear->data);
    }
}
```



Queue Implementation using Linked List

```
void display(void)
{
    struct node *temp=front;
    while(temp!=NULL)
    {
        printf("%d\t",temp->data);
        temp=temp->next;
    }
    printf("\n");
}
```



Queue Implementation using Linked List

```
void main ()  
{  
    enqueue(10);  
    enqueue(20);  
    enqueue(30);  
    enqueue(40);  
    display();  
    dequeue();  
    display();  
    peep();  
    display();  
}
```

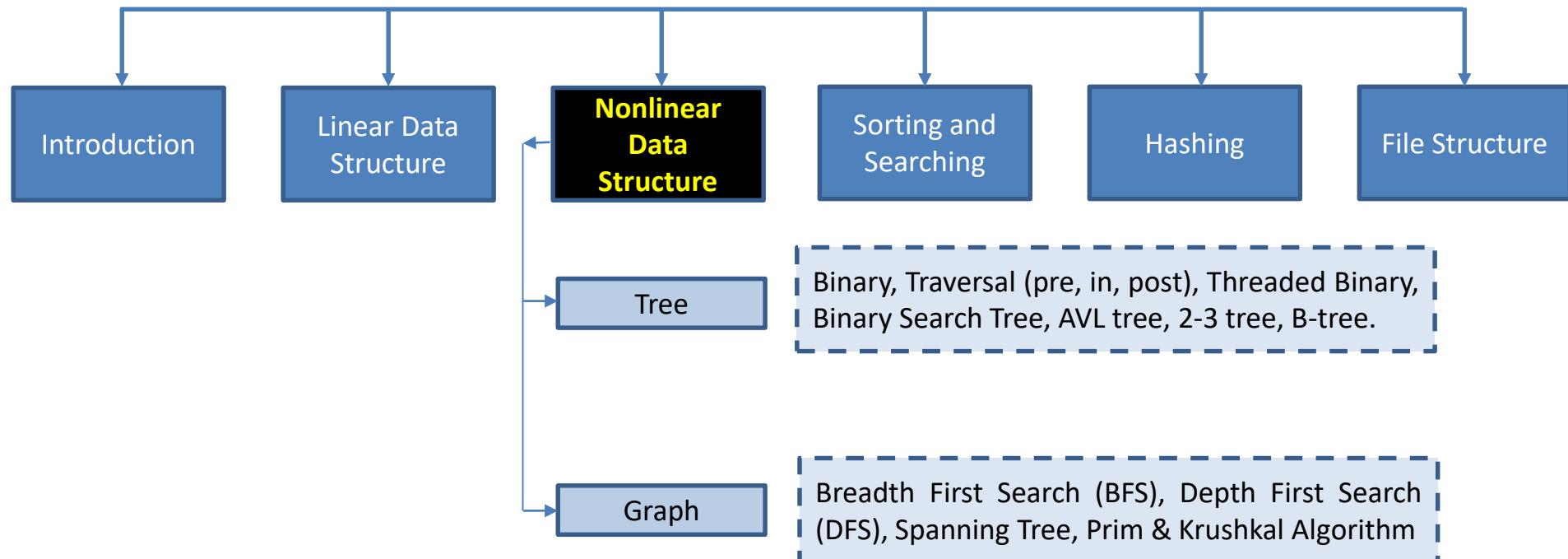
```
D:\Personal\MyLectures\DSA\Programs\queuell.exe  
10      20      30      40  
20      30      40  
Element at the front = 20  
Element at the end = 40  
20      30      40
```



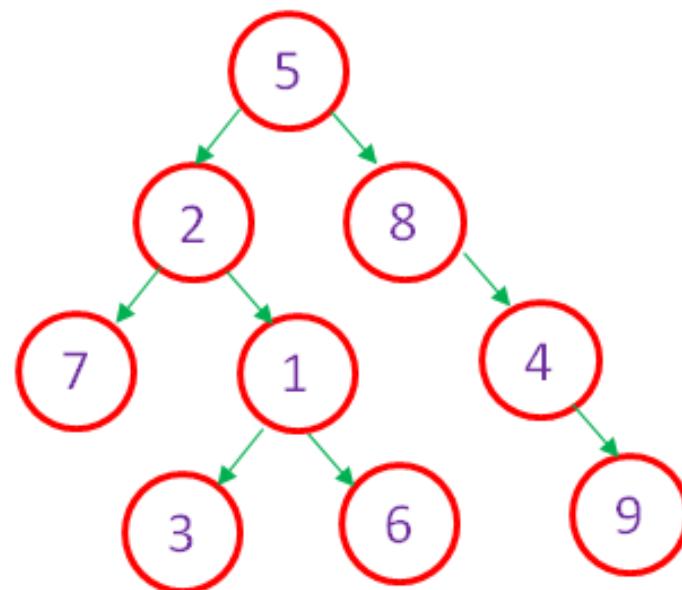
Thank You

Next: Introduction to Tree Data Structures

DSA: Nonlinear Data Structure



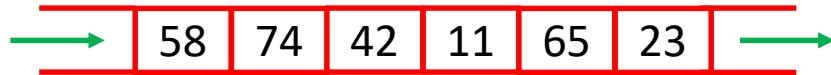
TREE Data Structures



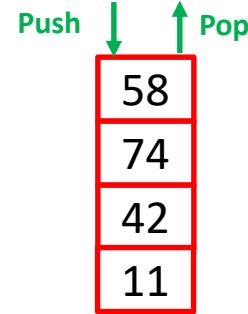
Linear Data Structures



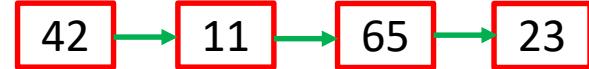
Array



Queue



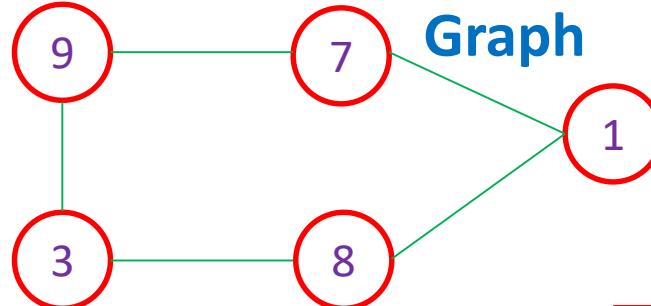
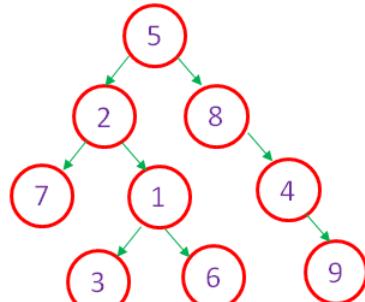
Stack



Linked List

Nonlinear Data Structures

Tree



Graph





Linear Vs. Non-Linear DS

Linear Data Structure

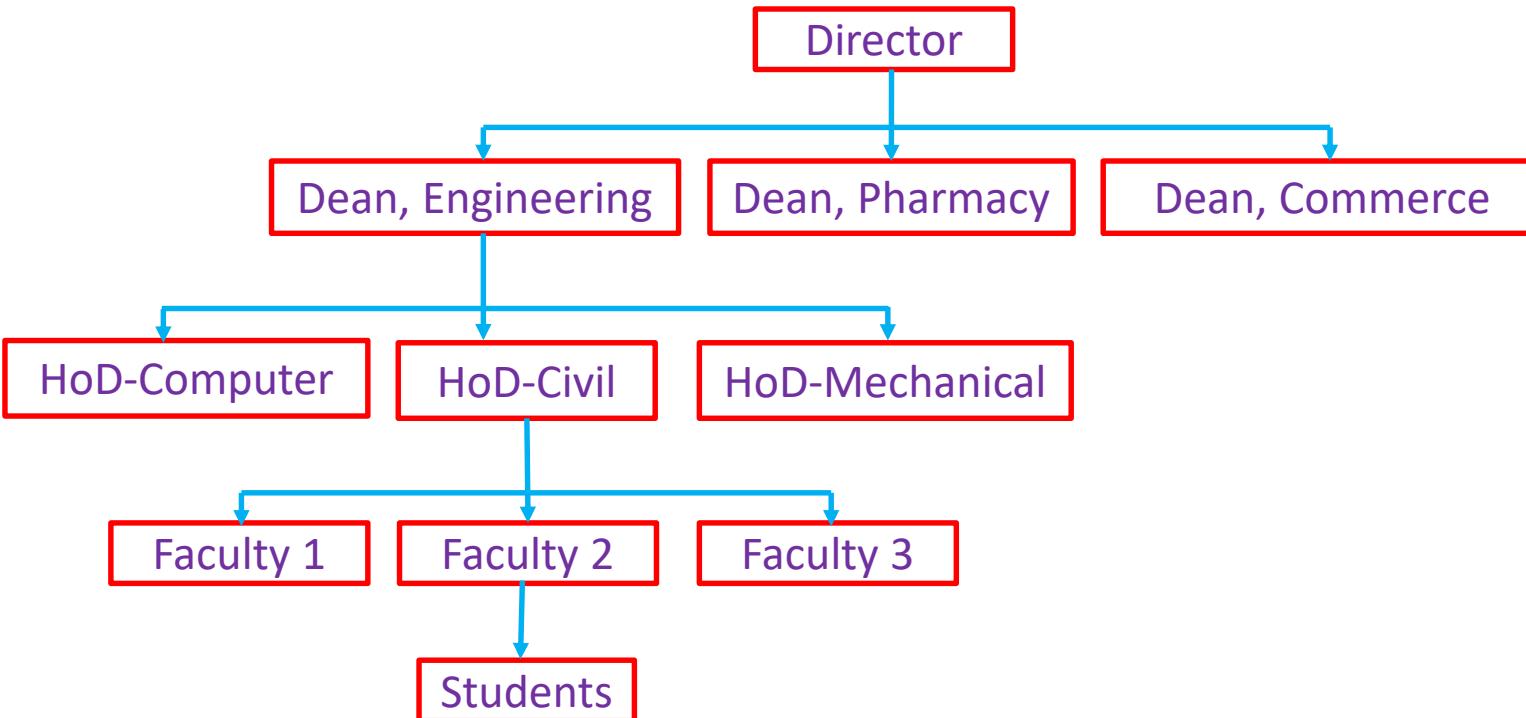
- It arranges the data in orderly manner where the elements are attached adjacently
- Level: Single
- Implementation: Easier
- Example: Array, Stack, Linked List, Queue

Non-linear Data Structure

- It arranges the data in sorted order, creating a relationship among the data element
- Level: Multiple
- Implementation: Difficult
- Example: Tree, Graph

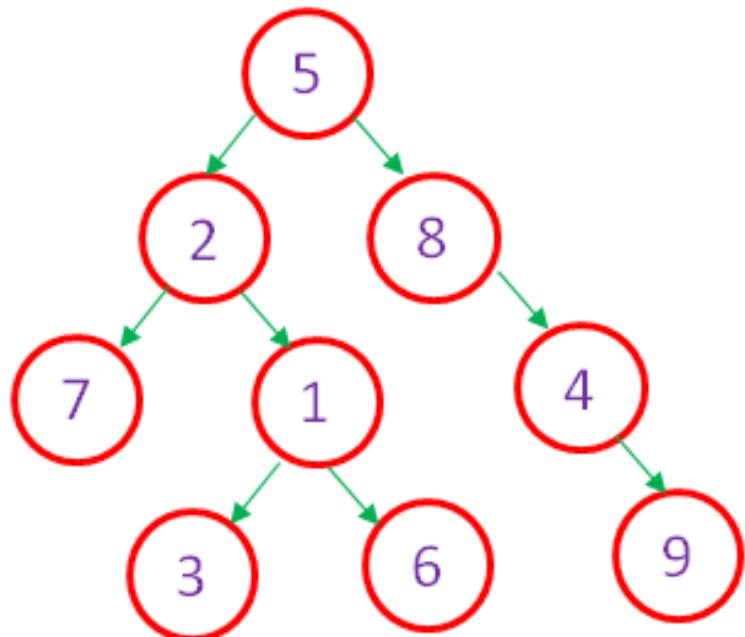


Tree: Example





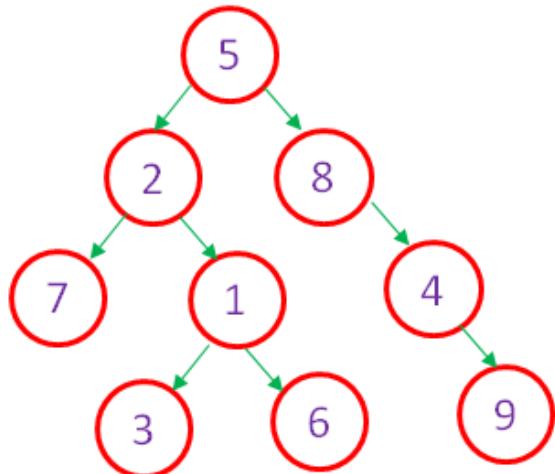
Tree



A tree is a widely used abstract data type that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.



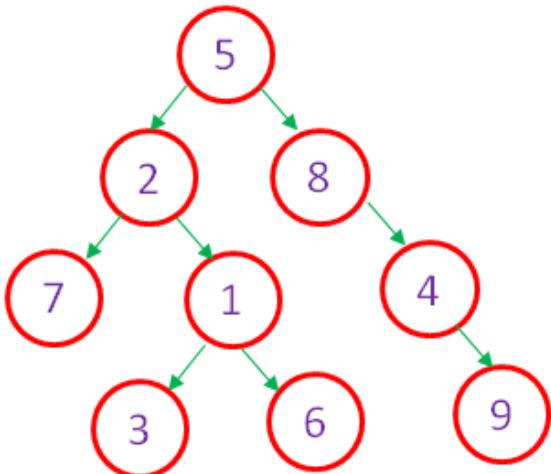
Tree Terminology



A **node** is a structure which may contain a value or condition. The topmost node in a tree is called the **root** node. Each node in a tree has zero or more **child** nodes, which are below it in the tree. A node that has a child is called the child's **parent** node (or superior). Child nodes with the same parent are **sibling** nodes.



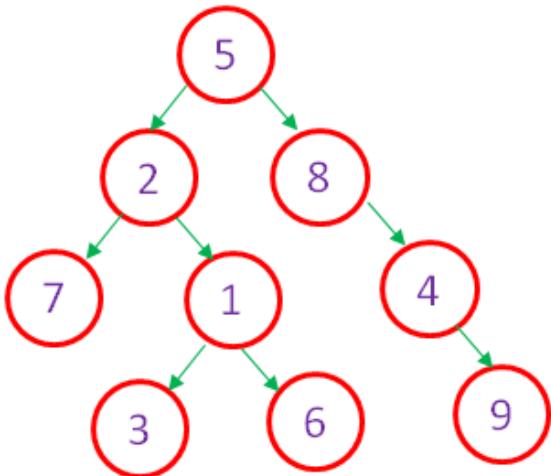
Tree Terminology



Path refers to the sequence of nodes along the edges of a tree. The node which does not have any child node is called the **leaf** node. The **depth of a node** is the number of edges from the root to the node. The **height of a node** is the number of edges from the node to the deepest leaf. **Degree of a node** is the total number of children of that node. **Degree of a tree** is the highest degree of a node among all the nodes in the tree.



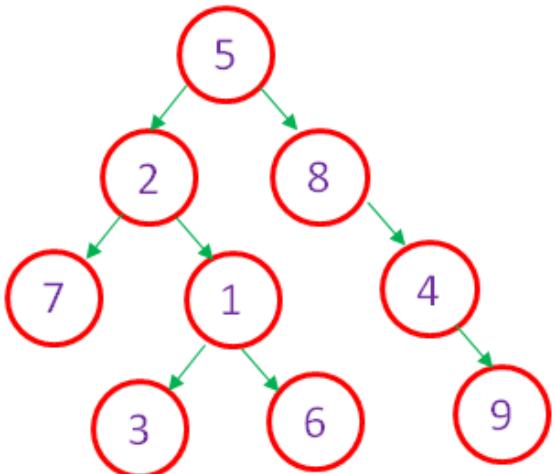
Tree Terminology



A **subtree** of a tree T is a tree S consisting of a node in T and all of its descendants in T. The subtree corresponding to the root node is the **entire tree**; the subtree corresponding to any other node is called a **proper subtree**.



Tree Terminology



Height of a tree is height of a root node. **Level of a node** is the number of edges/distance from root to that node.

Level of a tree is **height of a tree** but **level of a node** may or may not be same as **height of a node**.

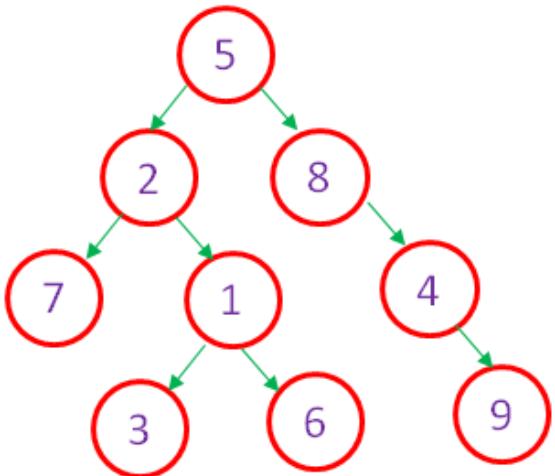


Thank You

Next: Binary Tree



Binary Tree



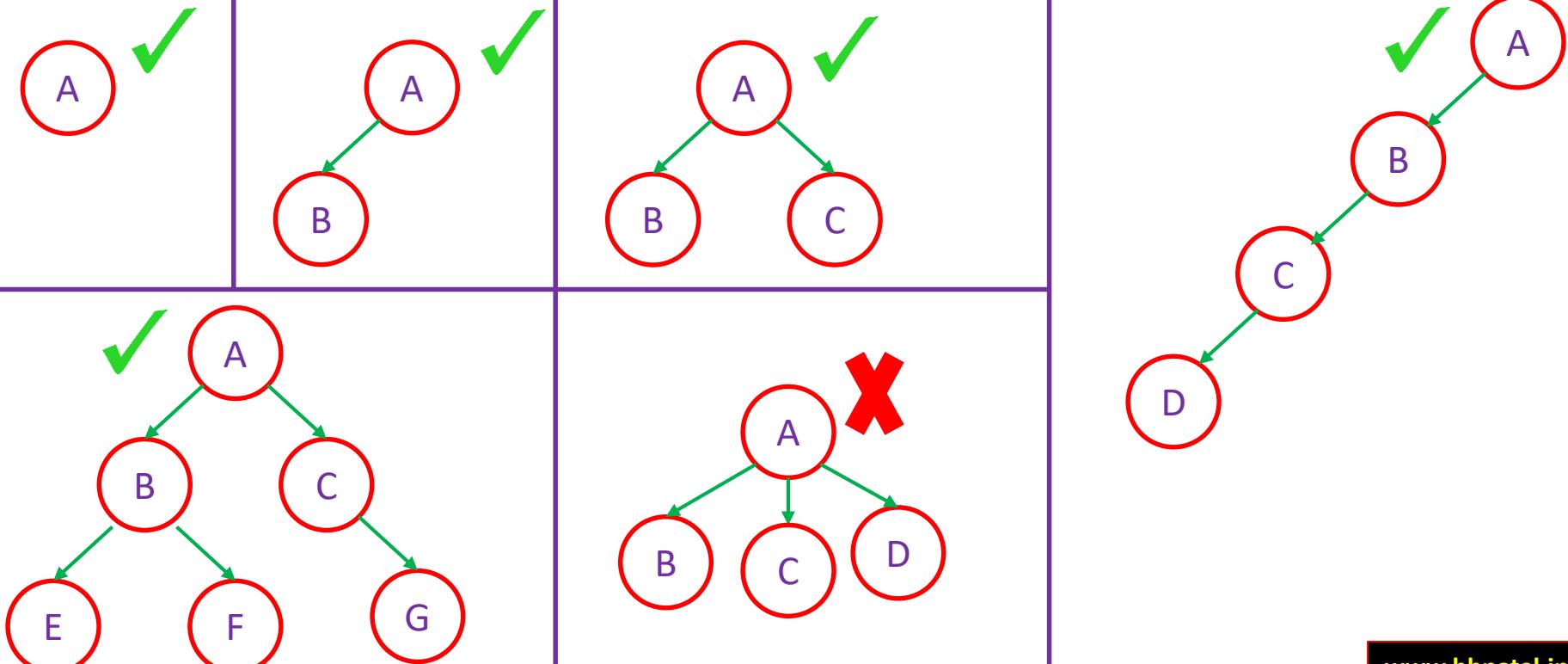
A **binary tree** is a hierarchical data structure in which each node has **at most two children** generally referred as left child and right child. Each node contains three components: Pointer to left subtree. Pointer to right subtree. Data element.



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

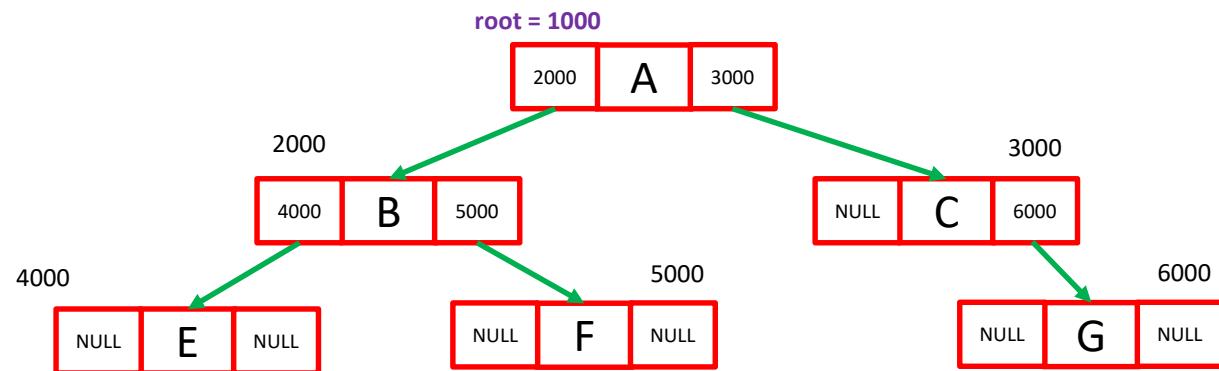
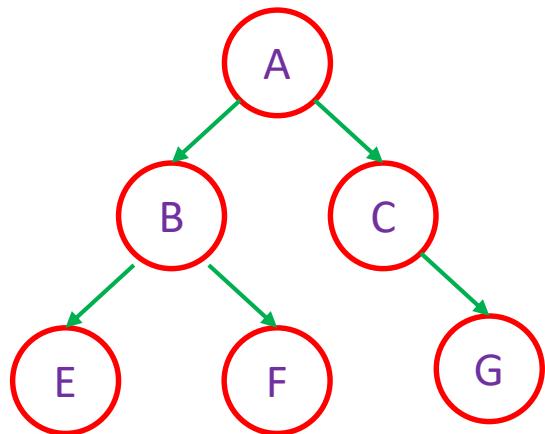


Binary Tree





Binary Tree



Implementation of Binary Tree



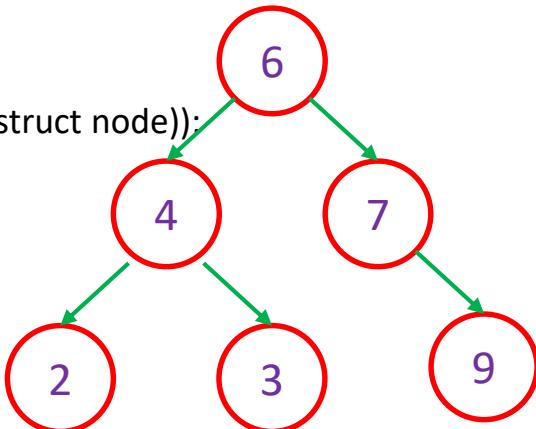
```
#include <stdio.h>
struct node
{
    int data;
    struct node *right, *left;
};

void main()
{
    struct node *createBinTree(void);
    struct node *root=NULL;
    root=createBinTree();
}
```

Implementation of Binary Tree



```
struct node *createBinTree(void)
{
int d;
struct node *newnode;
printf("Enter data : (-1 to exit) : ");
scanf("%d",&d);
if(d==-1)return NULL;
newnode = (struct node*)malloc(sizeof(struct node));
newnode->data=d;
printf("Enter left node of %d :\n",d);
newnode->left = createBinTree();
printf("Enter right node of %d :\n",d);
newnode->right = createBinTree();
return newnode;
}
```



```
D:\Personal\MyLectures\DSA\Programs\bintree.c
Enter data : (-1 to exit) : 6
Enter left node of 6 :
Enter data : (-1 to exit) : 4
Enter left node of 4 :
Enter data : (-1 to exit) : 2
Enter left node of 2 :
Enter data : (-1 to exit) : -1
Enter right node of 2 :
Enter data : (-1 to exit) : -1
Enter right node of 4 :
Enter data : (-1 to exit) : 3
Enter left node of 3 :
Enter data : (-1 to exit) : -1
Enter right node of 3 :
Enter data : (-1 to exit) : -1
Enter right node of 6 :
Enter data : (-1 to exit) : 7
Enter left node of 7 :
Enter data : (-1 to exit) : -1
Enter right node of 7 :
Enter data : (-1 to exit) : 9
Enter left node of 9 :
Enter data : (-1 to exit) : -1
Enter right node of 9 :
Enter data : (-1 to exit) : -1
```



Binary Tree: Properties

- Each node can have maximum 2 children
- At any level, maximum number of nodes can be 2^n , where n is the level number
- Maximum number of nodes in a tree having height h = $n^{h+1}-1$
- Minimum number of nodes in a tree having height h = h+1



Thank You

Next: Tree Traversal

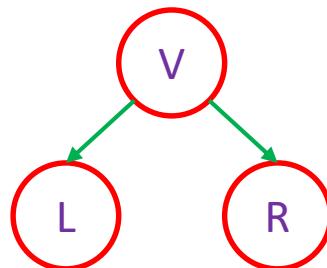
Tree Traversal

Inorder

$L \rightarrow V \rightarrow R$

Preorder

$V \rightarrow L \rightarrow R$



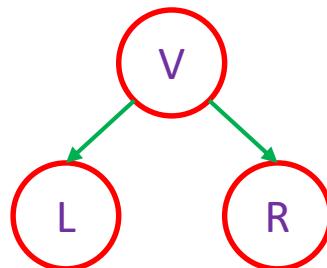
Postorder

$L \rightarrow R \rightarrow V$

Tree Traversal

Inorder

$L \rightarrow V \rightarrow R$



Preorder

$V \rightarrow L \rightarrow R$

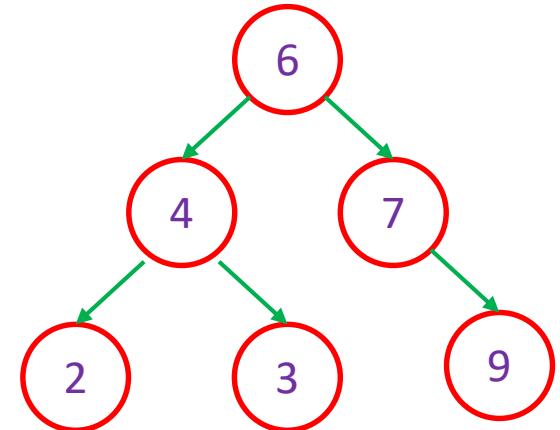
Postorder

$L \rightarrow R \rightarrow V$





Tree Traversal



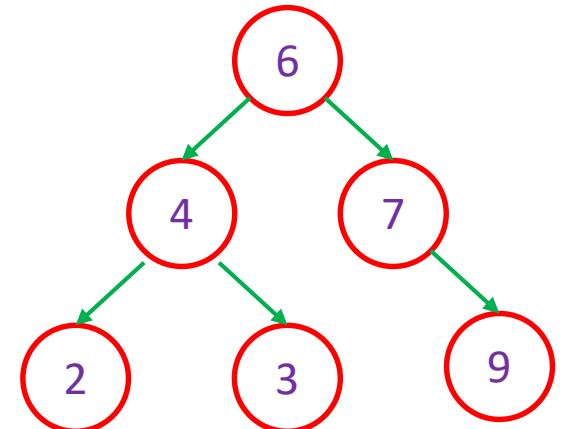
In-order Traversal: 2→4→3→6→7→9

Pre-order Traversal: 6→4→2→3→7→9

Post-order Traversal: 2→3→4→9→7→6



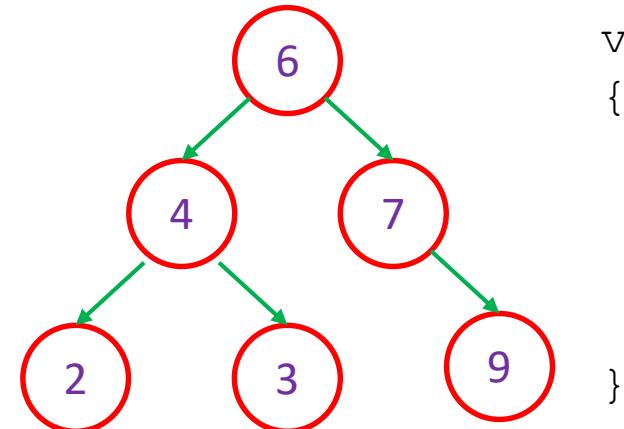
Pre-Order Tree Traversal



```
void preOrderTraversal (struct node *ptr)
{
    if(ptr==NULL) return;
    printf("%d ",ptr->data);
    preOrderTraversal (ptr->left) ;
    preOrderTraversal (ptr->right) ;
}
```



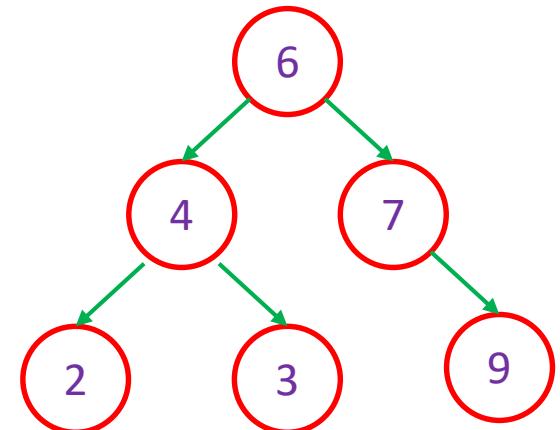
Post-Order Tree Traversal



```
void postOrderTraversal (struct node *ptr)
{
    if(ptr==NULL) return;
    preOrderTraversal (ptr->left);
    preOrderTraversal (ptr->right);
    printf ("%d ",ptr->data);
}
```



In-Order Tree Traversal



```
void inOrderTraversal (struct node *ptr)
{
    if(ptr==NULL) return;
    preOrderTraversal(ptr->left);
    printf("%d ",ptr->data);
    preOrderTraversal(ptr->right);
}
```



Complete Tree Traversal

```
#include <stdio.h>
struct node
{
    int data;
    struct node *right, *left;
};
void main()
{
    struct node *createBinTree(void);
    void preOrderTraversal (struct node *ptr);
    void postOrderTraversal (struct node *ptr);
    void inOrderTraversal (struct node *ptr);

    struct node *root=NULL;
    root=createBinTree();

    printf("\n Pre Order Tranversal : ");preOrderTraversal(root);
    printf("\n Post Order Tranversal : ");postOrderTraversal(root);
    printf("\n In Order Tranversal : ");inOrderTraversal(root);
}
```

```
struct node *createBinTree(void)
{
    int d;
    struct node *newnode = (struct node*)malloc(sizeof(struct node));
    printf("Enter data : (-1 to exit) : ");
    scanf("%d",&d);
    if(d==-1)return NULL;
    newnode->data=d;
    newnode->right=NULL;
    newnode->left=NULL;
    printf("Enter left node of %d :\n",d);
    newnode->left = createBinTree();
    printf("Enter right node of %d :\n",d);
    newnode->right = createBinTree();
    return newnode;
}
```



Complete Tree Traversal

```
void preOrderTraversal (struct node *ptr)
{
if(ptr==NULL)return;
printf("%d ",ptr->data);
preOrderTraversal(ptr->left);
preOrderTraversal(ptr->right);
}

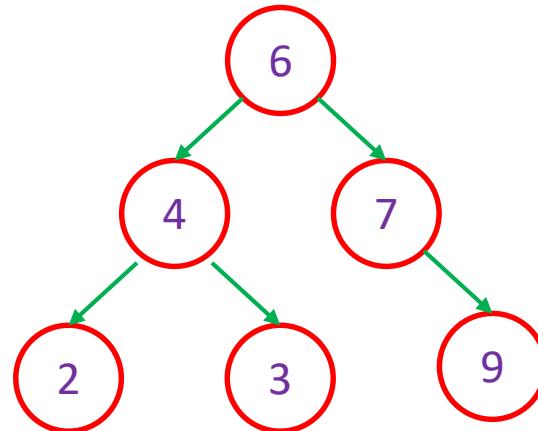
void postOrderTraversal (struct node *ptr)
{
if(ptr==NULL)return;
postOrderTraversal(ptr->left);
postOrderTraversal(ptr->right);
printf("%d ",ptr->data);
}

void inOrderTraversal (struct node *ptr)
{
if(ptr==NULL)return;
inOrderTraversal(ptr->left);
printf("%d ",ptr->data);
inOrderTraversal(ptr->right);
}
```

D:\Personal\MyLectures\DSA\Programs\btreeTraverse.e

```
Enter data : (-1 to exit) : 6
Enter left node of 6 :
Enter data : (-1 to exit) : 4
Enter left node of 4 :
Enter data : (-1 to exit) : 2
Enter left node of 2 :
Enter data : (-1 to exit) : -1
Enter right node of 2 :
Enter data : (-1 to exit) : -1
Enter right node of 4 :
Enter data : (-1 to exit) : 3
Enter left node of 3 :
Enter data : (-1 to exit) : -1
Enter right node of 3 :
Enter data : (-1 to exit) : -1
Enter right node of 6 :
Enter data : (-1 to exit) : 7
Enter left node of 7 :
Enter data : (-1 to exit) : -1
Enter right node of 7 :
Enter data : (-1 to exit) : 9
Enter left node of 9 :
Enter data : (-1 to exit) : -1
Enter right node of 9 :
Enter data : (-1 to exit) : -1

Pre Order Tranversal : 6 4 2 3 7 9
Post Order Tranversal : 2 3 4 9 7 6
In Order Tranversal : 2 4 3 6 7 9
```

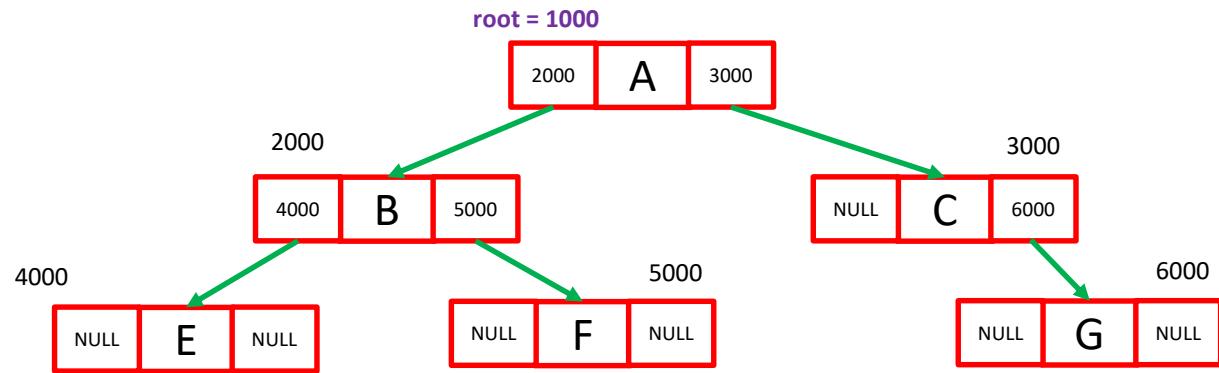
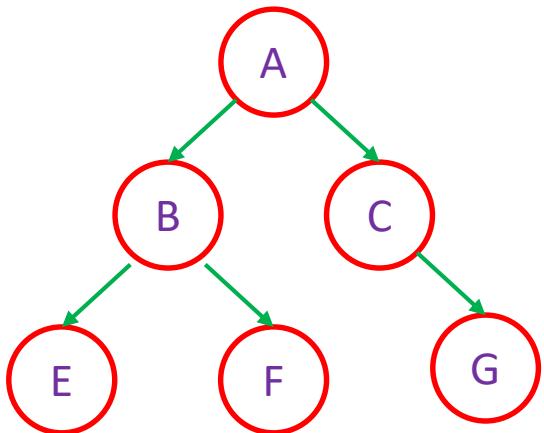




Thank You

Next: Threaded Binary Tree

Threaded Binary Tree (TBT)



Can I use the pointers, where NULL is stored, to store some other address?

Threaded Binary Tree

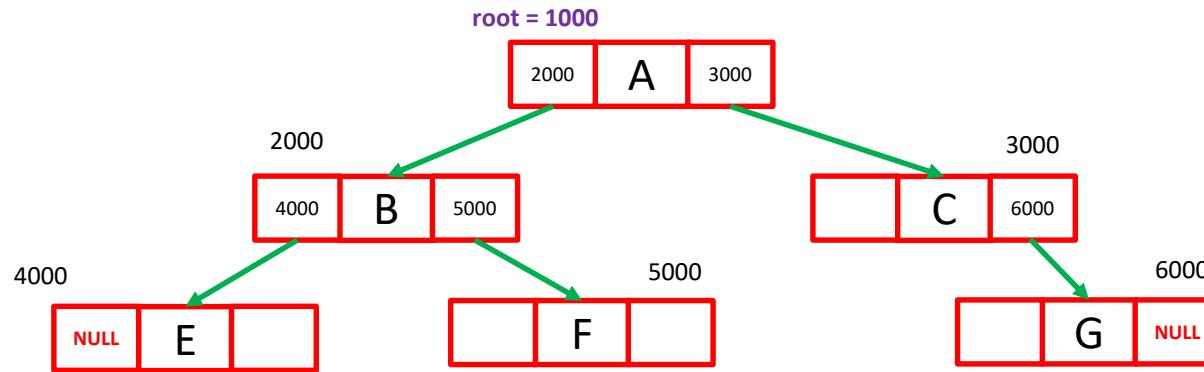
Can I use the pointers, where NULL is stored, to store some other address?

Step:

1. Keep the LEFT most and RIGHT most NULL pointers as NULL
2. Change all other NULL pointers as
 - 2.1 Left Pointer = Inorder Predecessor
 - 2.2 Right Pointer = Inorder Successor



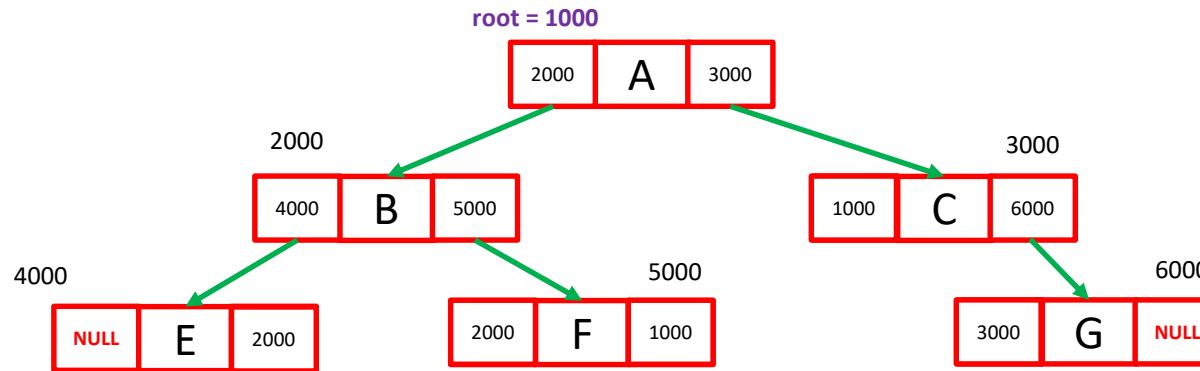
Threaded Binary Tree



1. Keep the LEFT most and RIGHT most NULL pointers as NULL



Threaded Binary Tree



- 2.1 Left Pointer = Inorder Predecessor
2.2 Right Pointer = Inorder Successor

Inorder Traversal

E → B → F → A → C → G

NULL → E → B → F → A → C → G → NULL



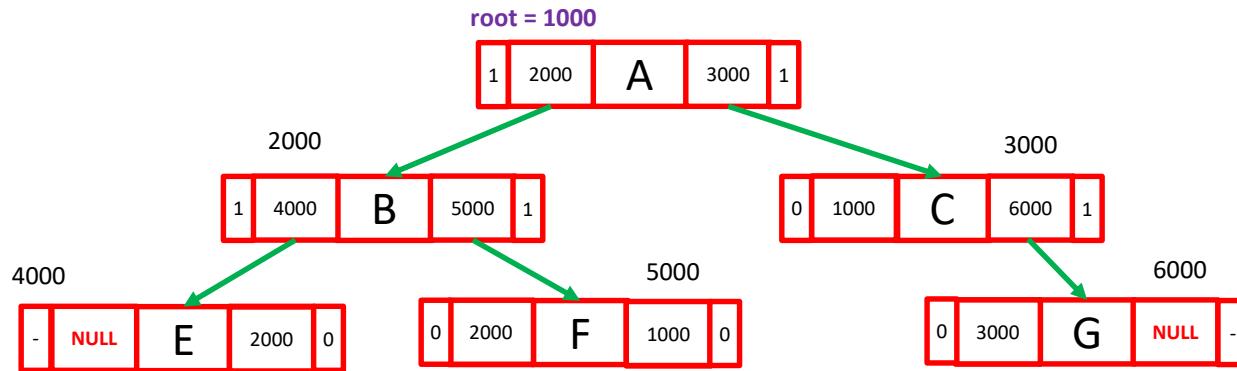
Threaded Binary Tree

How do I know that the pointer points to the parent node or some other node?

```
Struct node
{
    char data;
    int left_flag,right_flag;
    struct node *left, *right;
};
```



Threaded Binary Tree

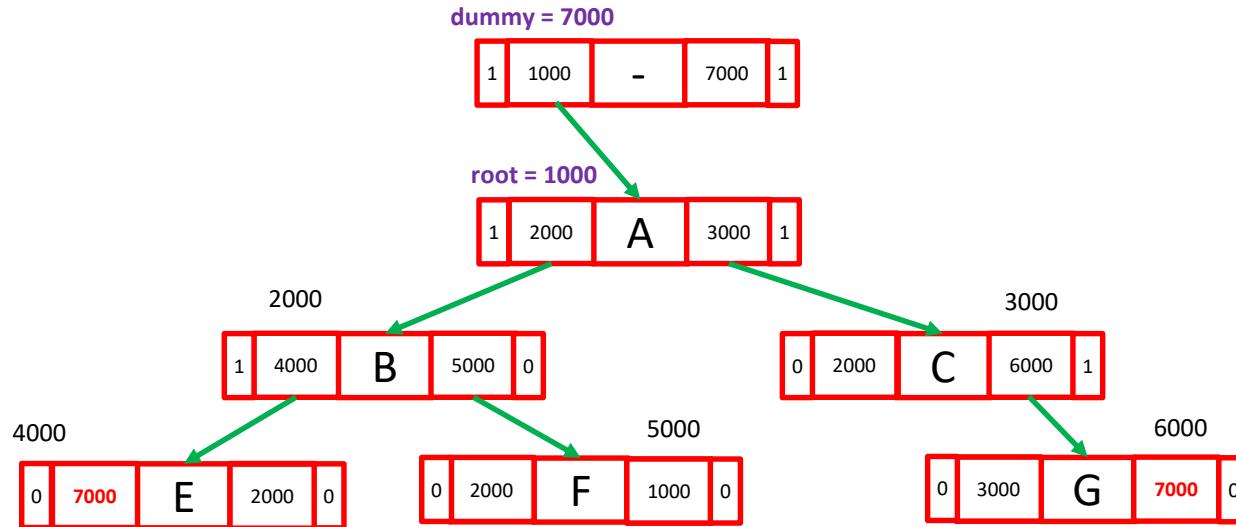


Flag = 1, the pointer points to the child node

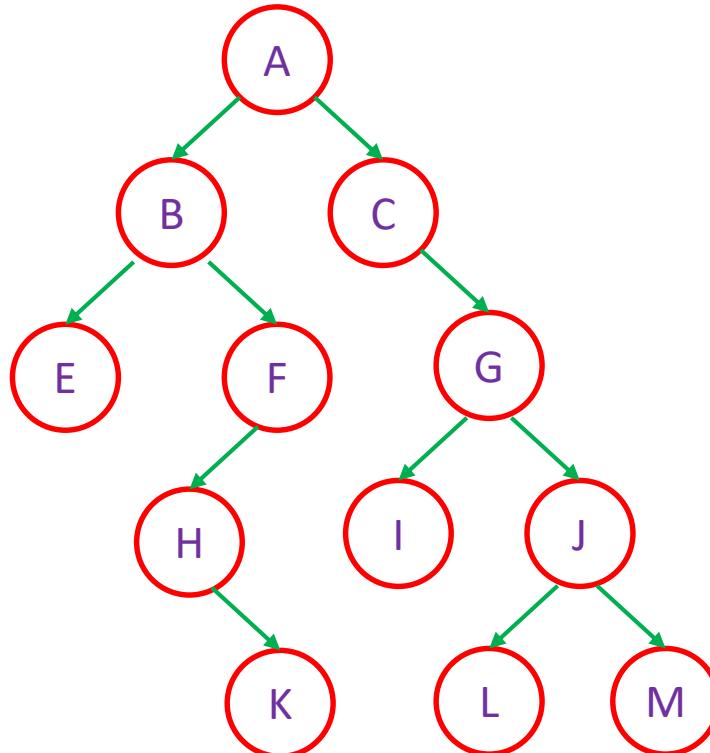
Flag = 0, the pointer points to the parent/ancestor node



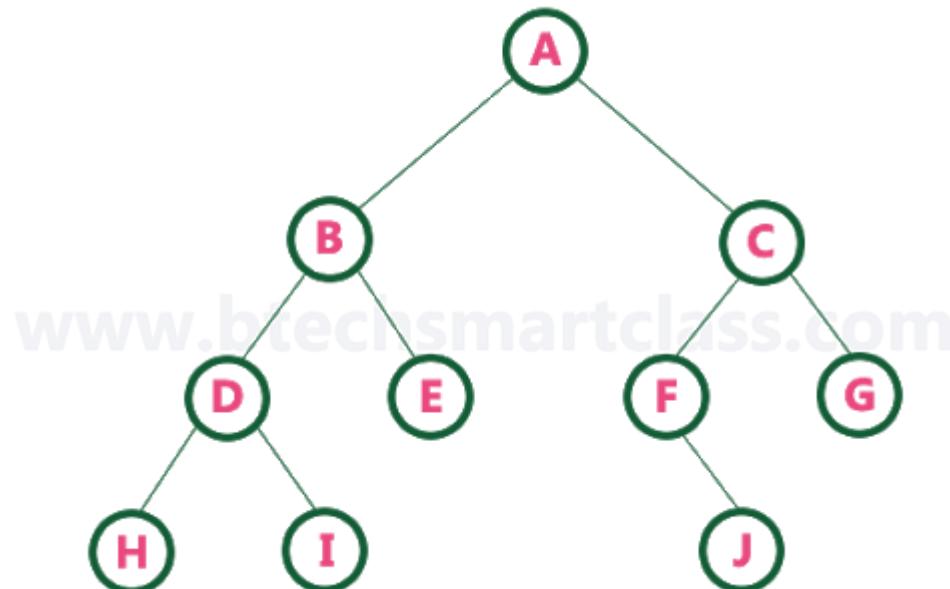
Threaded Binary Tree



Laboratory Exercise: TBT



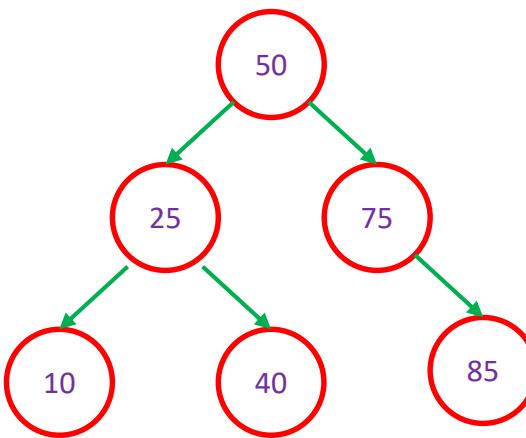
Laboratory Exercise: TBT





Thank You

Next: Binary Search Tree

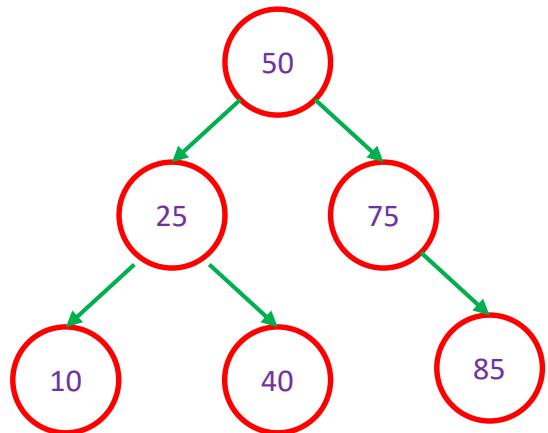


Binary Search Tree (BST)

Condition 1: Maximum Number of Children = 2

Condition 2: $L < V < R$

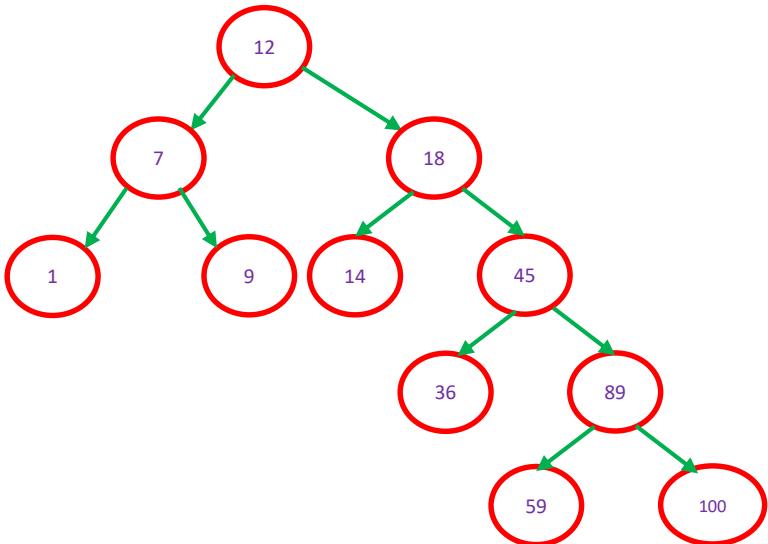
Binary Search Tree (BST)



Left subtree should contain values less than the vertex and right subtree should contain values greater than the vertex.

Binary Search Tree (BST)

Create binary search tree for following sequence of numbers:
12, 7, 9, 18, 14, 1, 45, 89, 100, 36 59





BST - Implementation

```
#include <stdio.h>
struct node
{
    int data;
    struct node *left, *right;
};

struct node* newnode(int item)
{
    struct node* temp = (struct node*)malloc(sizeof(struct node));
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}
```



BST - Implementation

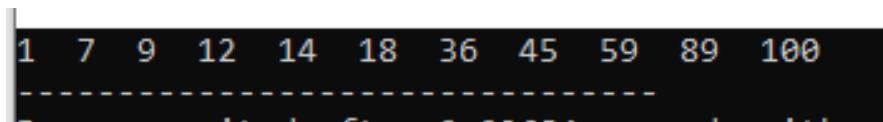
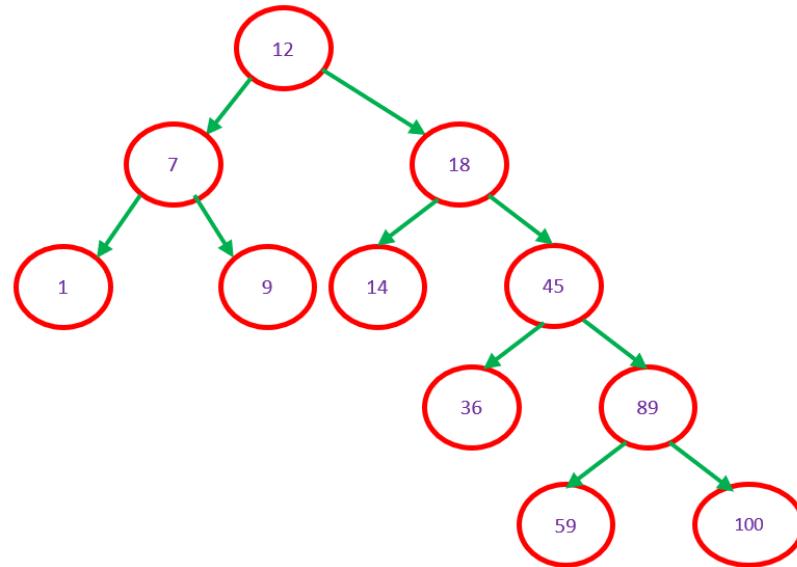
```
struct node* insert(struct node* node, int data)
{
    if (node == NULL) return newnode(data);
    if (data < node->data) node->left = insert(node->left, data);
    else if (data > node->data) node->right = insert(node->right, data);
    return node;
}

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



BST - Implementation

```
int main()
{
    struct node* root = NULL;
    root = insert(root, 12);
    insert(root, 7);
    insert(root, 9);
    insert(root, 18);
    insert(root, 14);
    insert(root, 1);
    insert(root, 45);
    insert(root, 89);
    insert(root, 100);
    insert(root, 36);
    insert(root, 59);
    inorder(root);
    return 0;
}
```



Binary Search Tree - Delete

Deleting a data/node from BST

The node to be deleted may have 0, 1 or 2 child node(s).

Case 1: Node to be deleted has 0 child

Simply delete the child and set the parent/successor link to NULL

Case 2: Node to be deleted has 1 child

Its successor (left/right) is to be connected with its parent node.

Case 3: Node to be deleted has 2 children

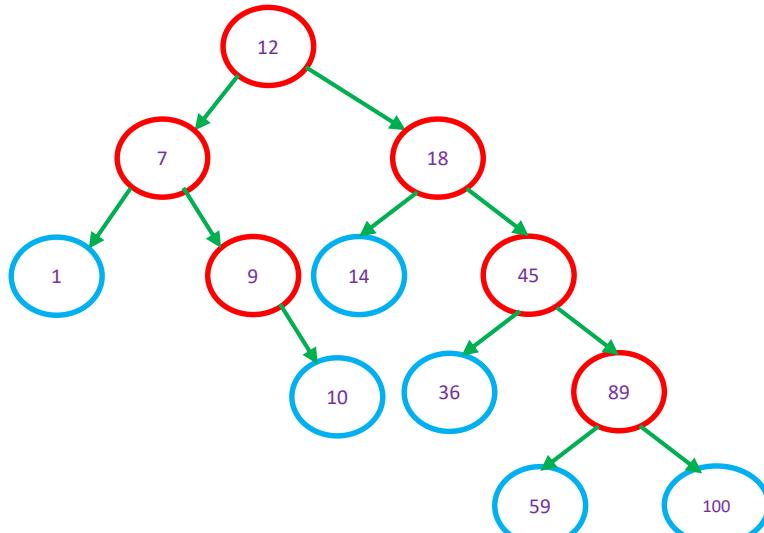
The node could be deleted (or replaced with) its (a) inorder predecessor OR
(b) inorder successor



Binary Search Tree - Delete

Case 1: Node to be deleted has 0 child

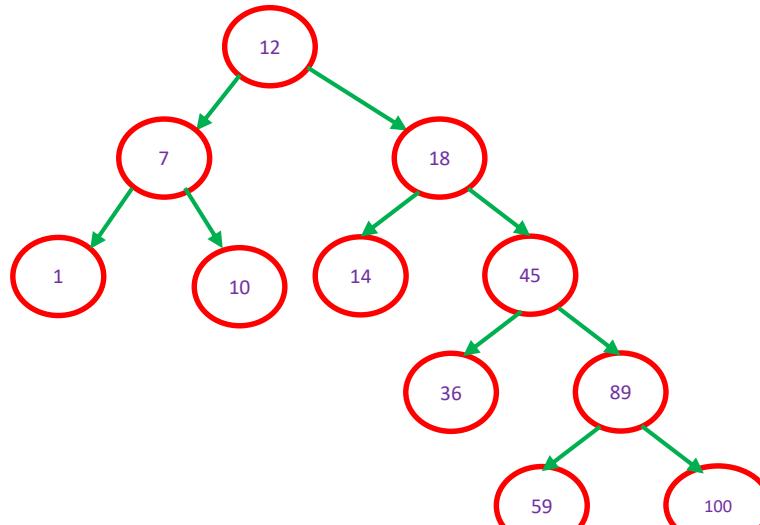
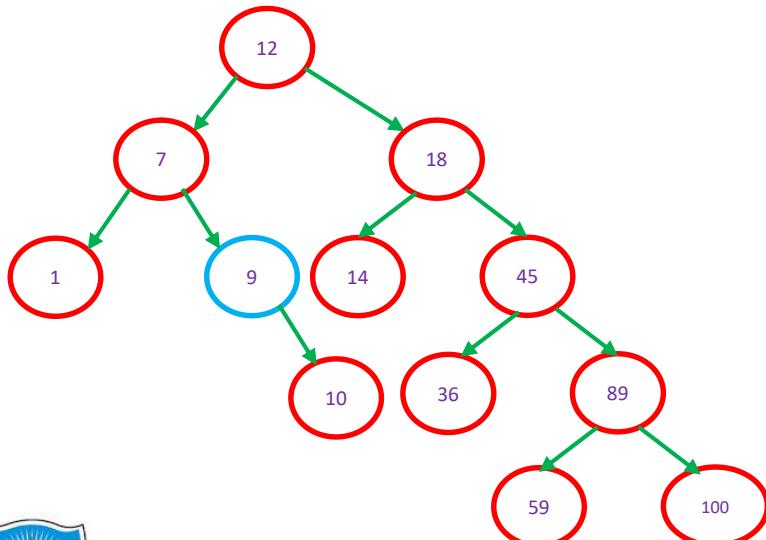
Simply delete the child and set the parent/successor link to NULL



Binary Search Tree - Delete

Case 2: Node to be deleted has 1 child

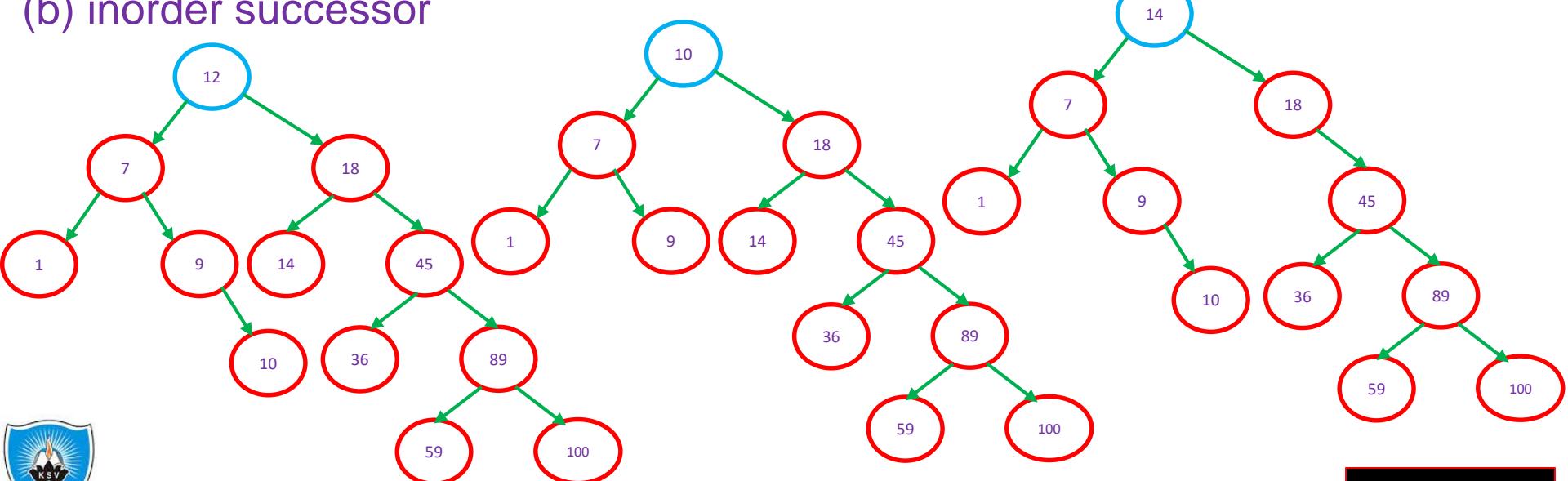
Its successor (left/right) is to be connected with its parent node.



Binary Search Tree - Delete

Case 3: Node to be deleted has 2 children

The node could be deleted (or replaced with) its (a) inorder predecessor OR
(b) inorder successor



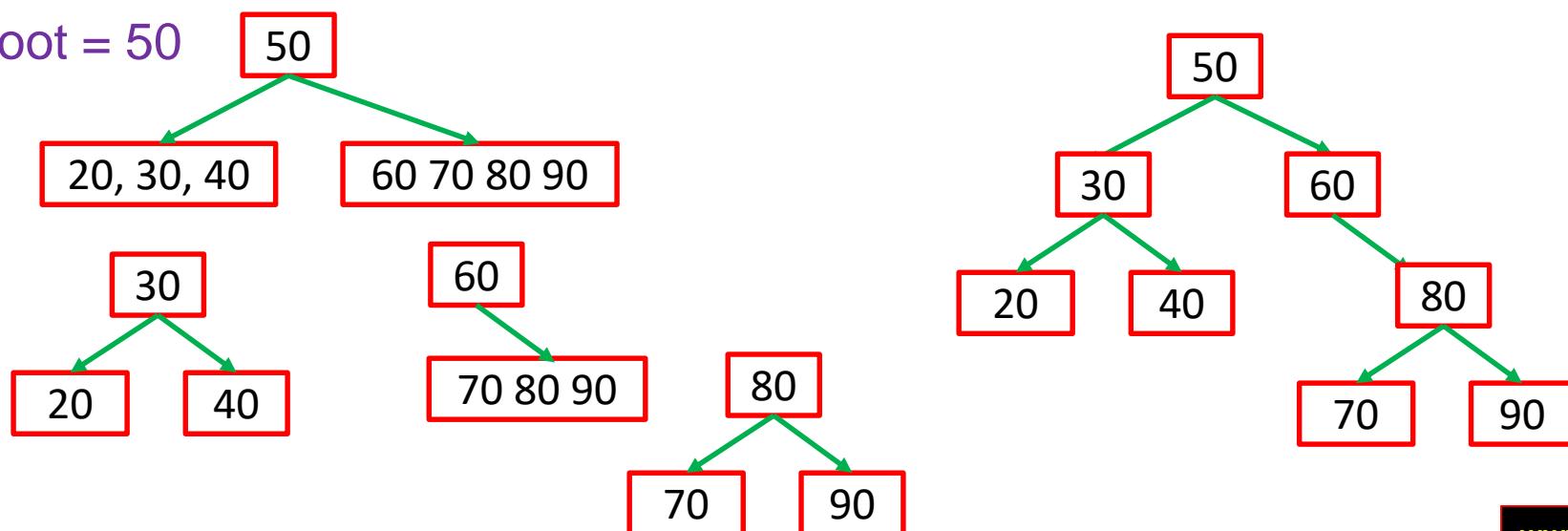


Constructing BST from Preorder Data

Given: Preorder (VLR): 50 30 20 40 60 80 70 90

Find: Inorder (LVR - Sorted): 20 30 40 50 60 70 80 90

Root = 50



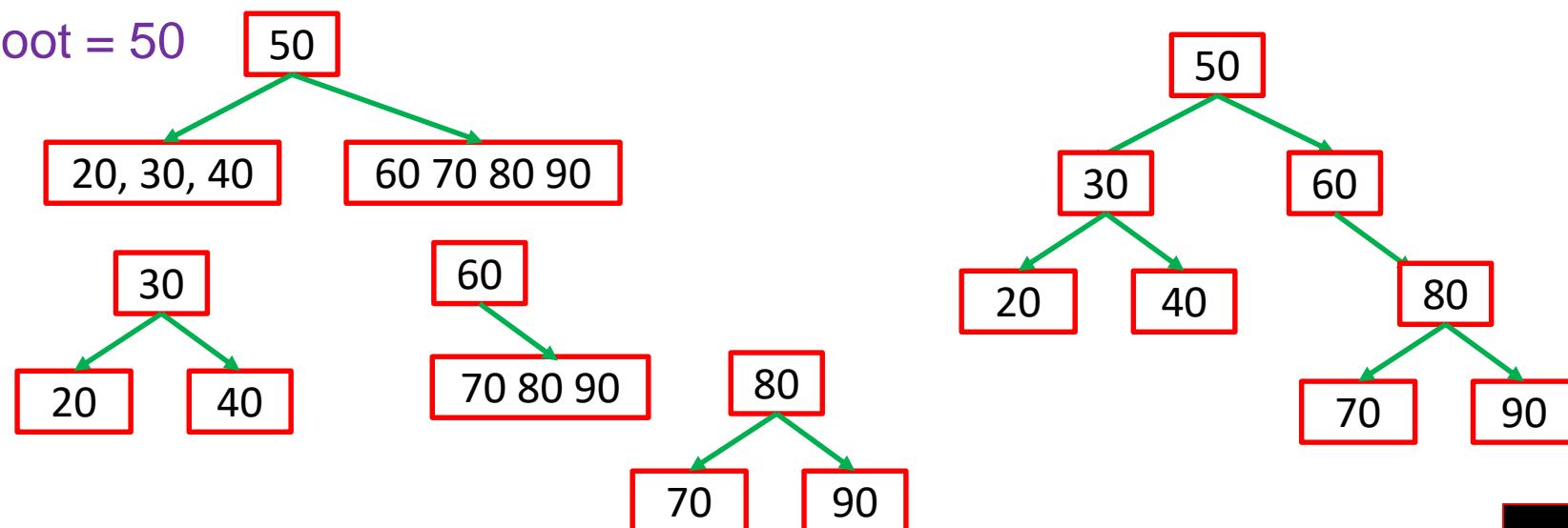


Constructing BST from Postorder Data

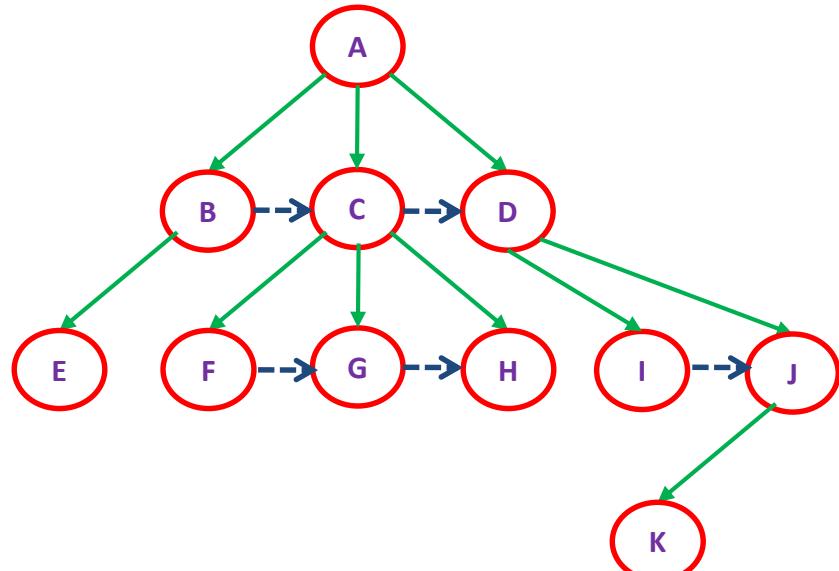
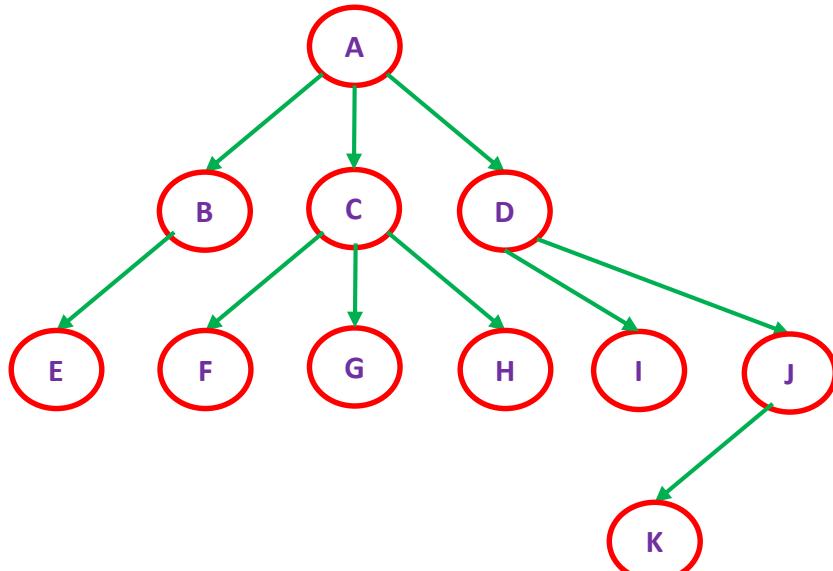
Given: Postorder (LRV): 20 40 30 70 90 80 60 50

Find: Inorder (LVR - Sorted): 20 30 40 50 60 70 80 90

Root = 50



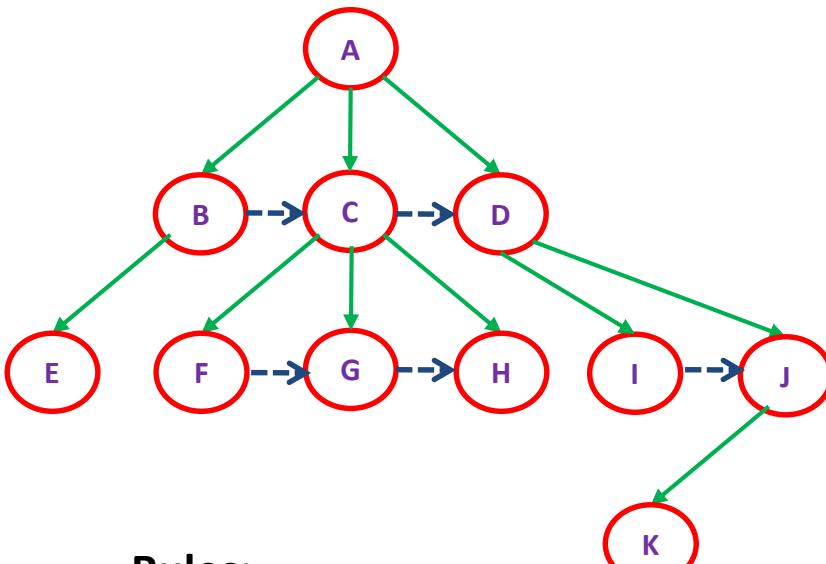
Constructing Binary Tree from General Tree



Rules:

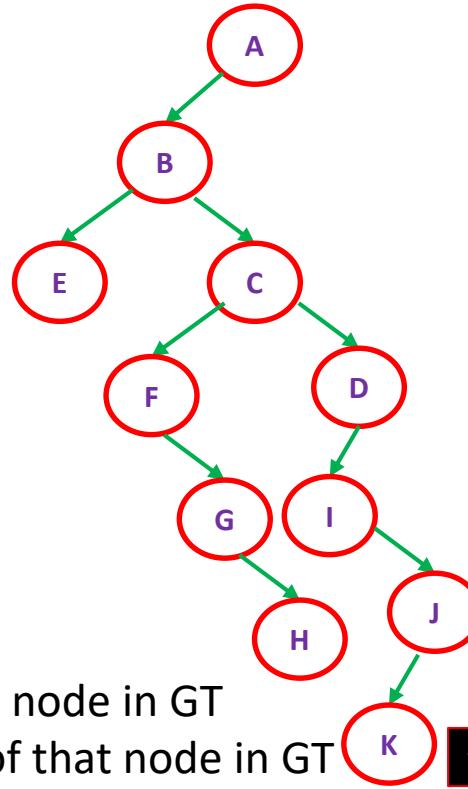
1. Root node of BT = Root node of GT
2. Left child of the node in BT = Left most child of the node in GT
3. Right child of the node in BT = Next (right) sibling of that node in GT

Constructing Binary Tree from General Tree



Rules:

1. Root node of BT = Root node of GT
2. Left child of the node in BT = Left most child of the node in GT
3. Right child of the node in BT = Next (right) sibling of that node in GT



Laboratory Exercise: BST

Create BST For:

50, 25, 75, 12, 100, 30, 28, 90, 150, 112, 80, 83

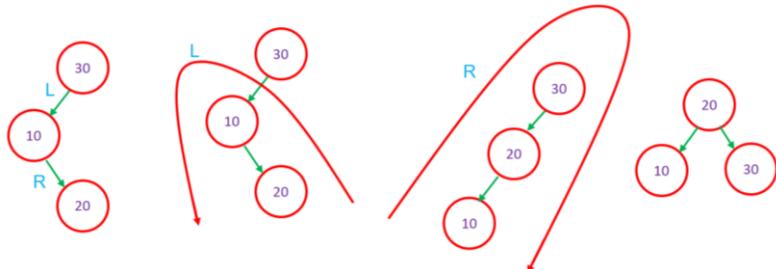
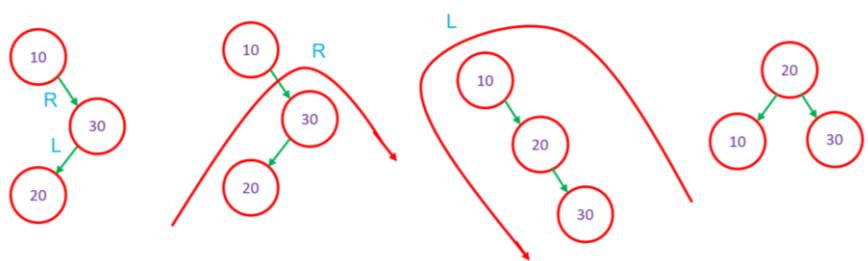
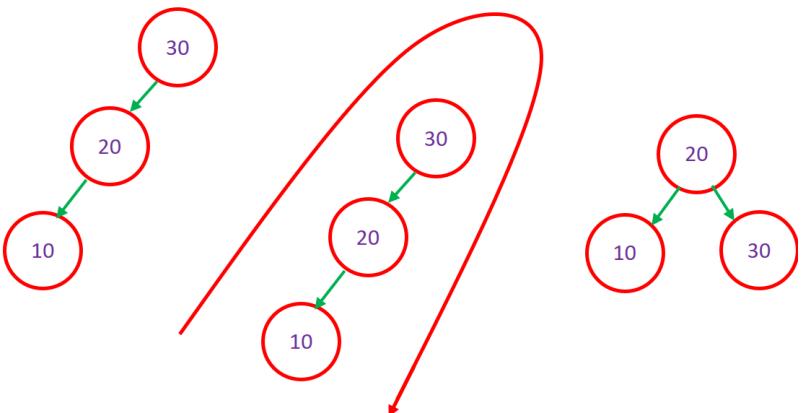
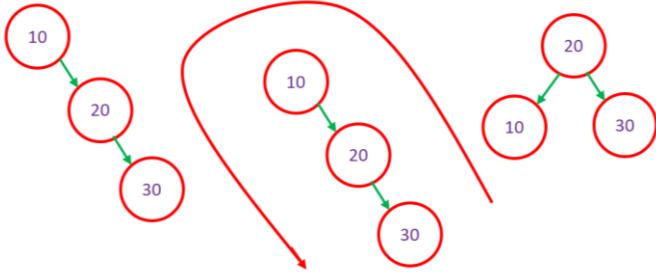




Thank You

Next: AVL Tree

AVL Tree



AVL Tree

An AVL tree (named after inventors **Adelson-Velsky** and **Landis**) is a self-balancing binary search tree.

1. It is a BST
2. Height of Left Subtree – Height of Right Subtree = {-1, 0, 1}

Balance Factor = Height of Left Subtree – Height of Right Subtree = {-1, 0, 1}



AVL Tree

After every insertion in AVL, we calculate balance factor, and if it happens to be outside $\{-1, 0, 1\}$, we have to balance out the tree in such a way that the balance factor lies in $\{-1, 0, 1\}$

How to balance out a tree?

There are 4 different cases



Balancing AVL Tree

Case 1:

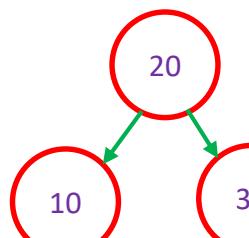
Left Rotation / Anti-clockwise turn



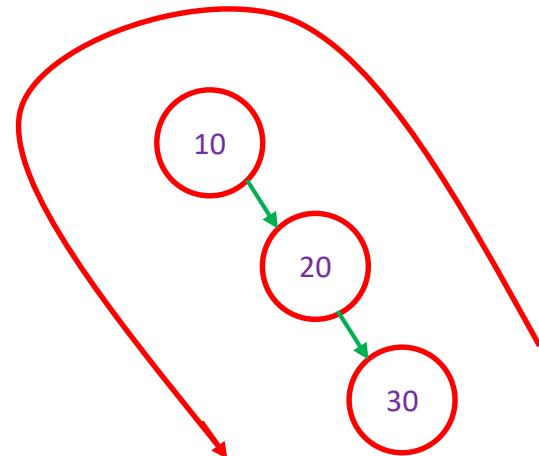
Balance factor of "20" = 0

Result

Balance factor of "10" = 0



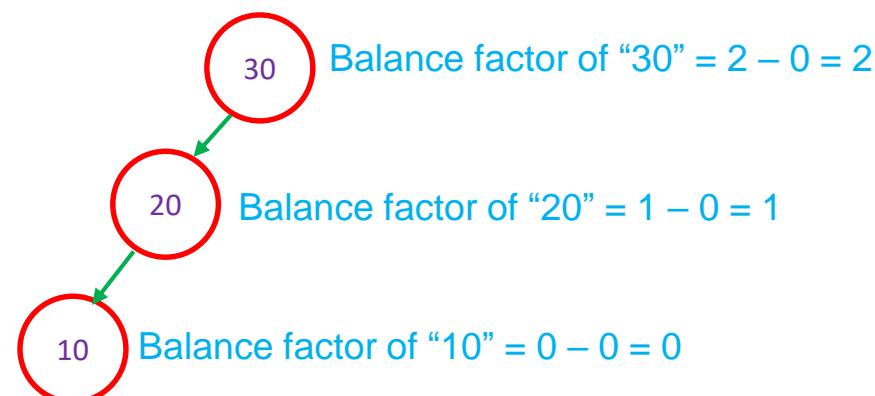
Balance factor of "30" = 0



Balancing AVL Tree

Case 2:

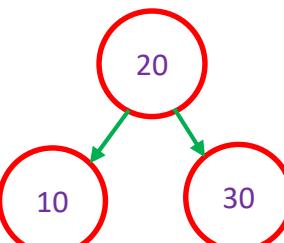
Right Rotation / Clockwise turn



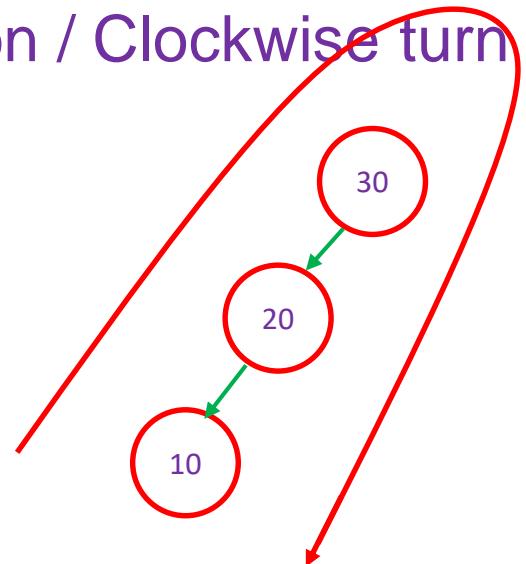
Balance factor of "20" = 0

Result

Balance factor of "10" = 0

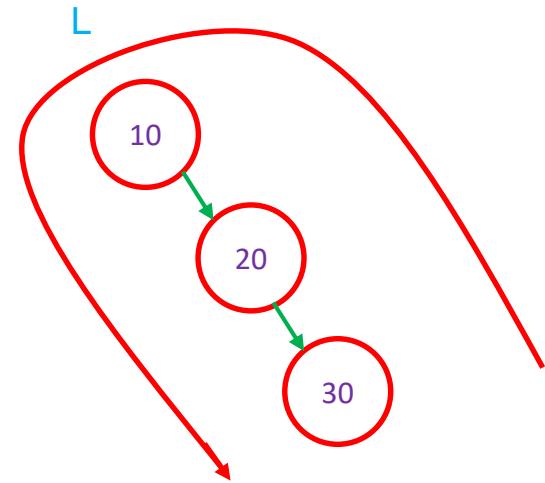
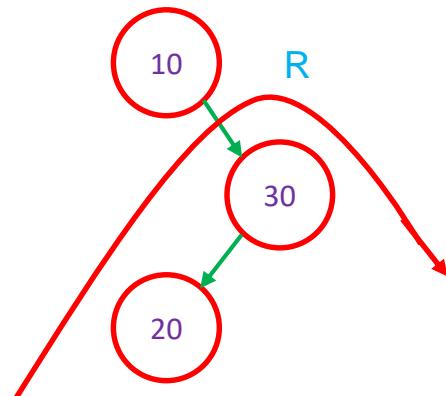
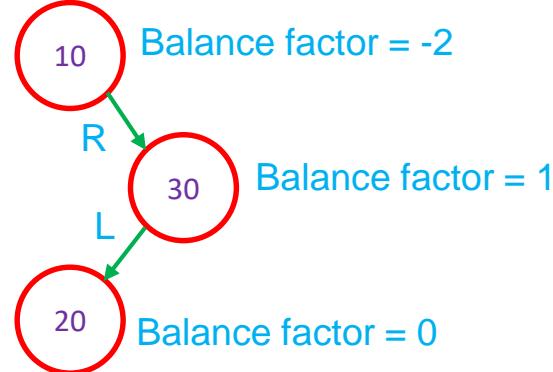


Balance factor of "30" = 0



Balancing AVL Tree

Case 3:

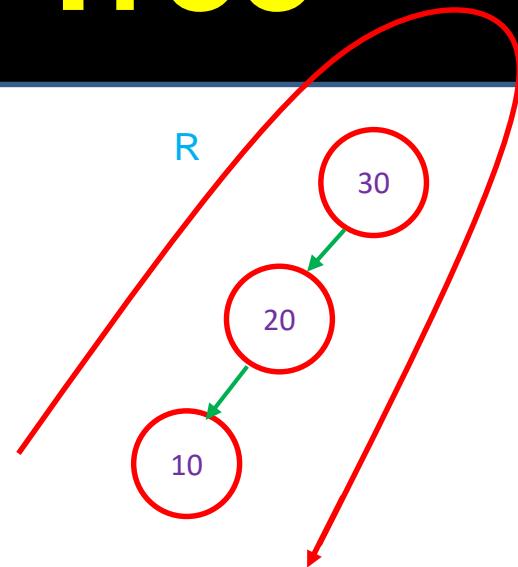
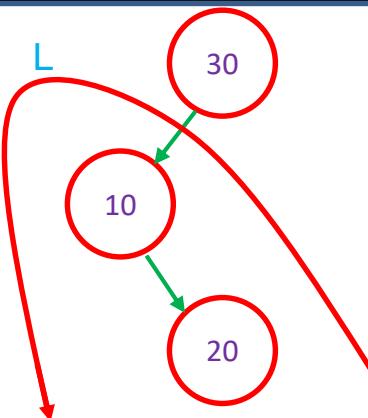
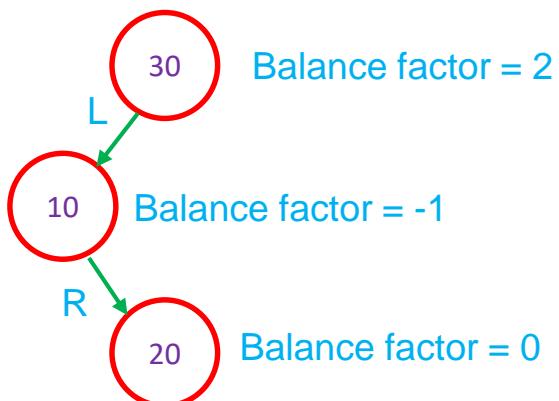


Result

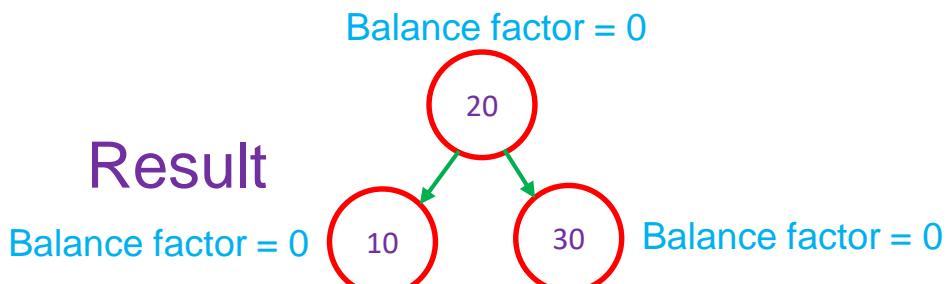


Balancing AVL Tree

Case 4:



Result



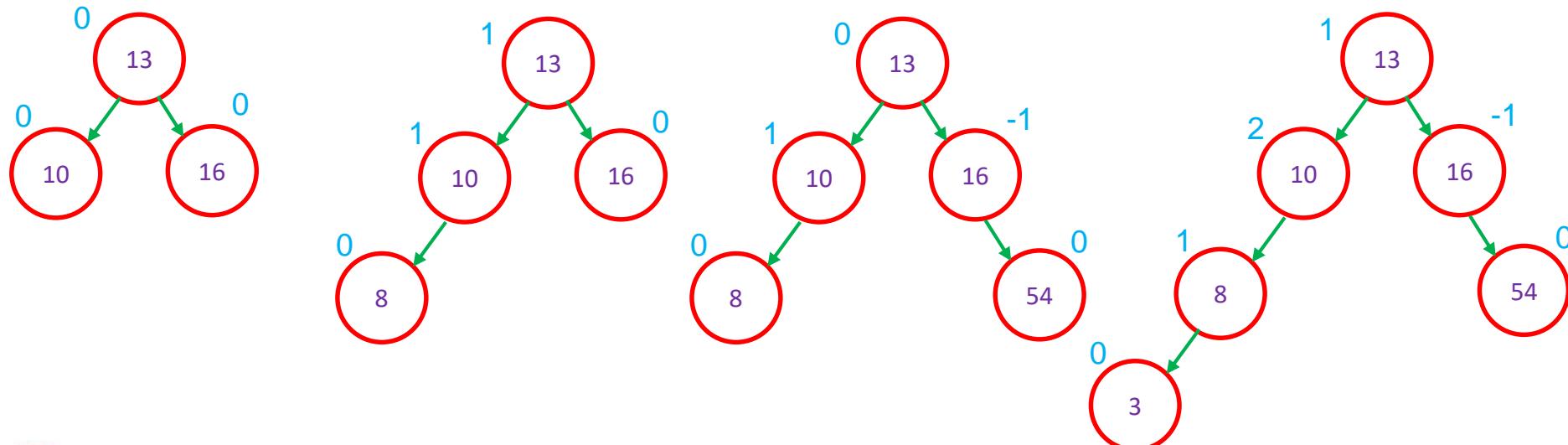
Balancing AVL Tree



Source: https://upload.wikimedia.org/wikipedia/commons/f/fd/AVL_Tree_Example.gif

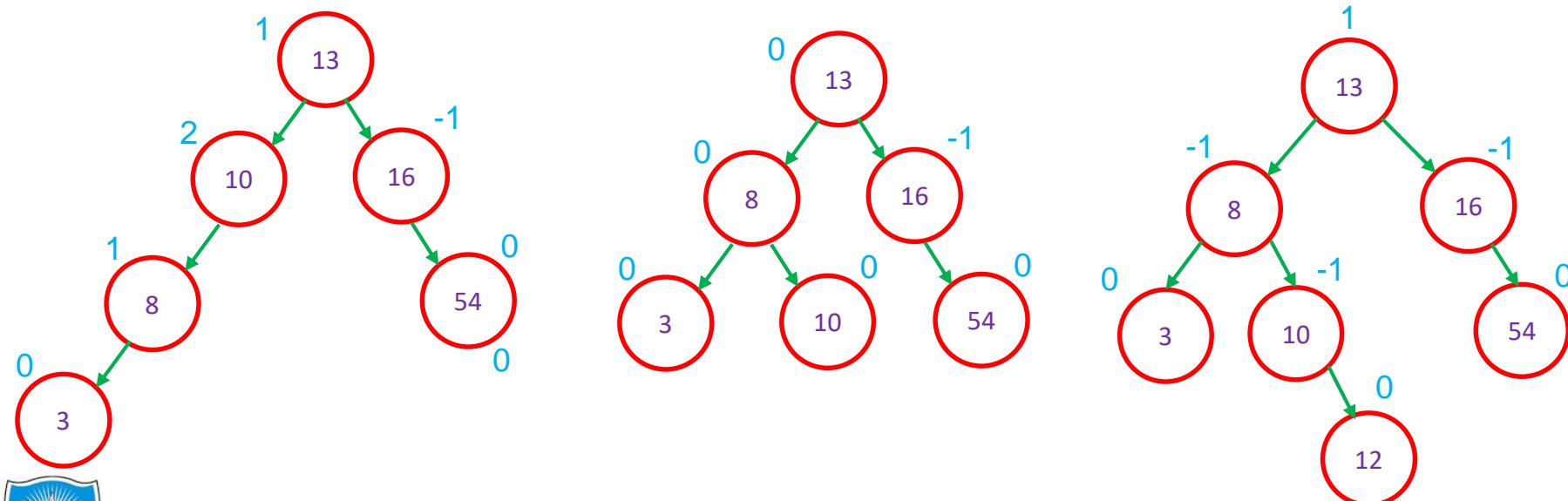
Assignment: Construct AVL Tree

13, 16, 10, 8, 54, ,3, 12, 11, 9, 59, 20, 15, 21



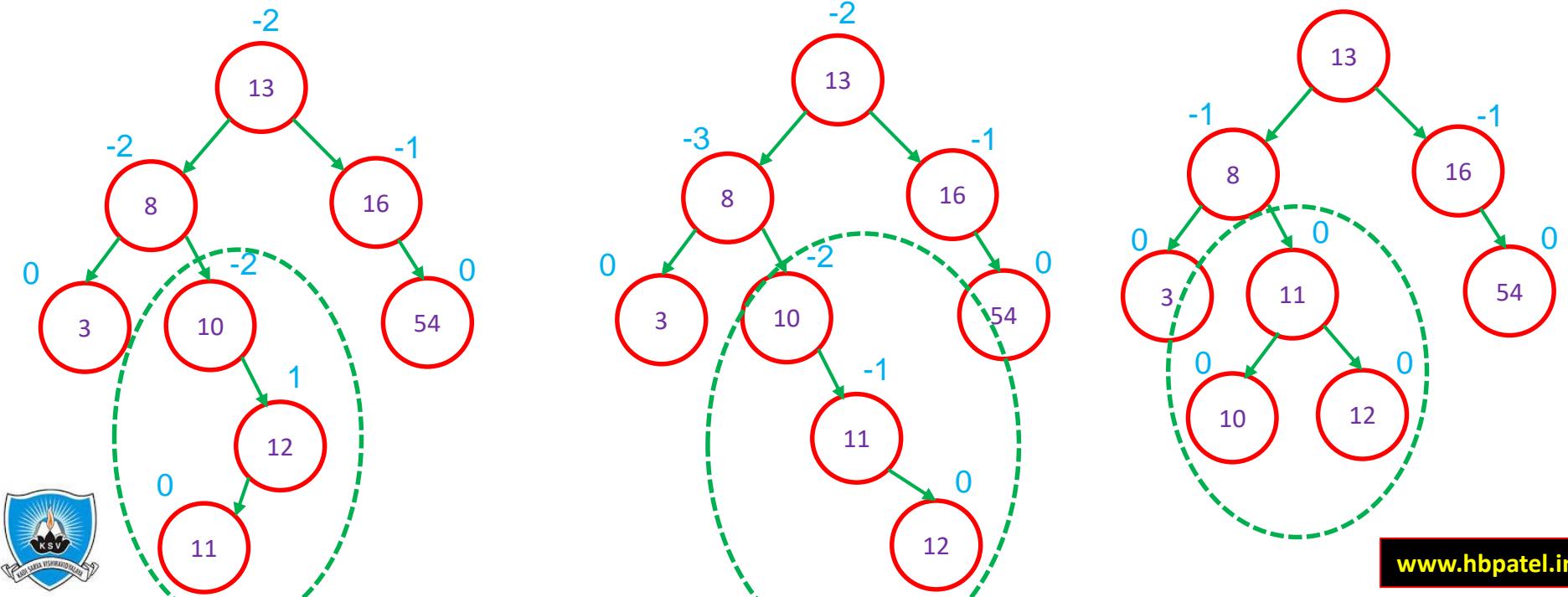
Assignment: Construct AVL Tree

13, 16, 10, 8, 54, ,3, 12, 11, 9, 59, 20, 15, 21



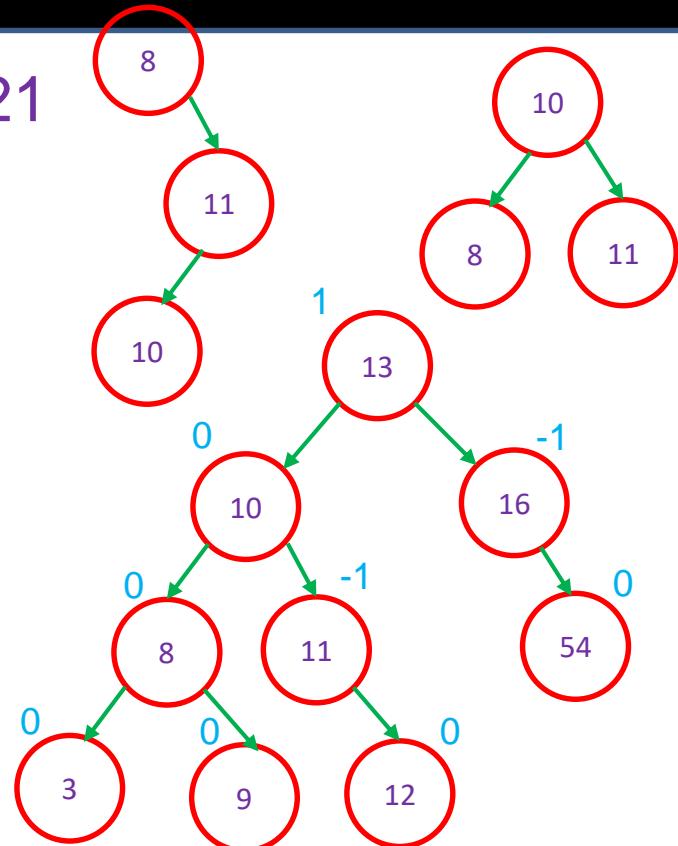
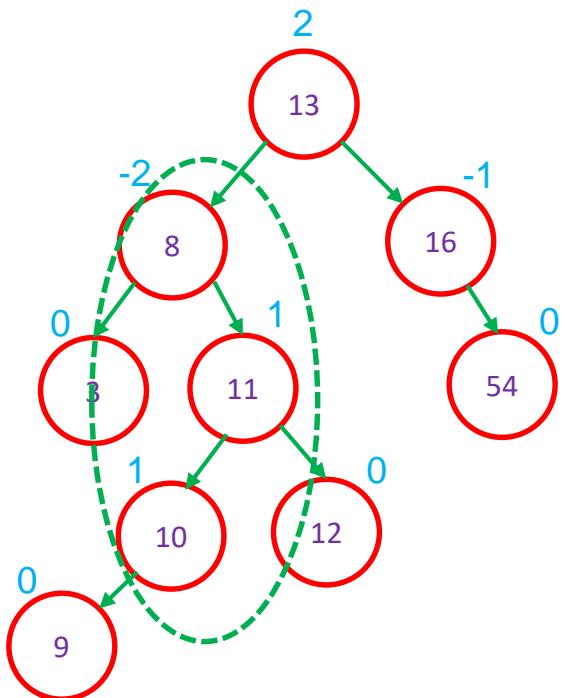
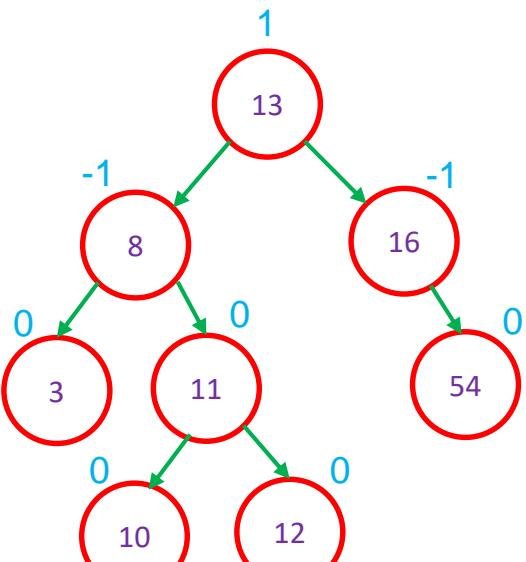
Assignment: Construct AVL Tree

13, 16, 10, 8, 54, ,3, 12, 11, 9, 59, 20, 15, 21



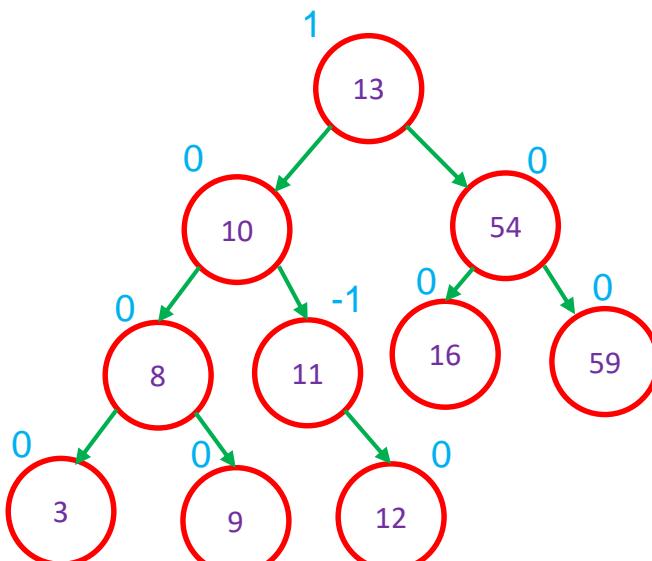
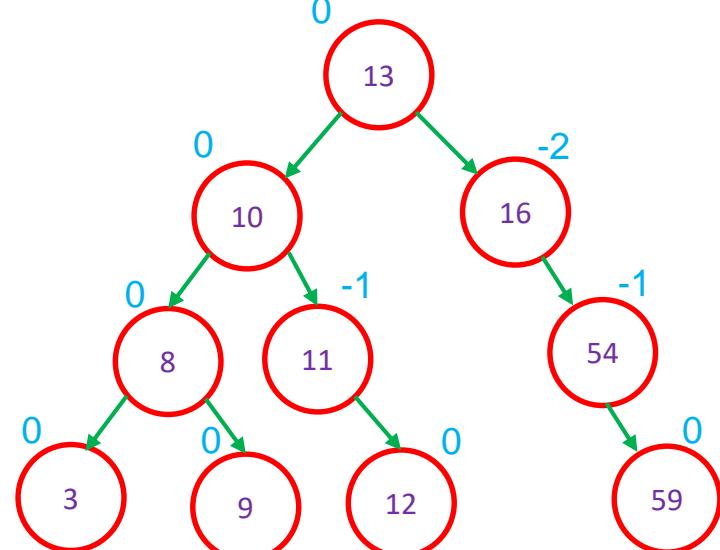
Assignment: Construct AVL Tree

13, 16, 10, 8, 54, ,3, 12, 11, 9, 59, 20, 15, 21



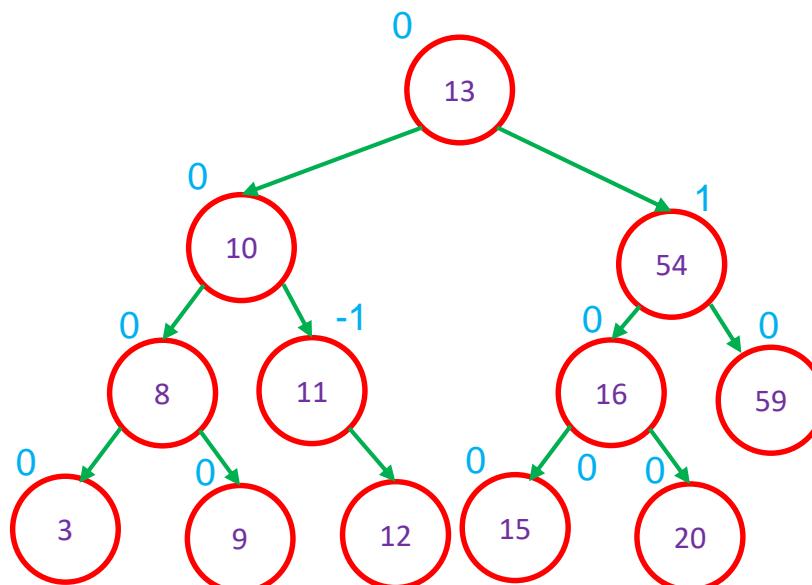
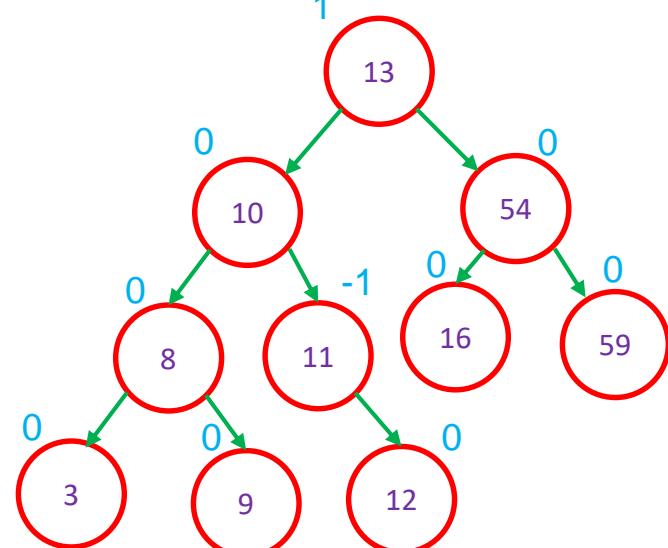
Assignment: Construct AVL Tree

13, 16, 10, 8, 54, ,3, 12, 11, 9, 59, 20, 15, 21



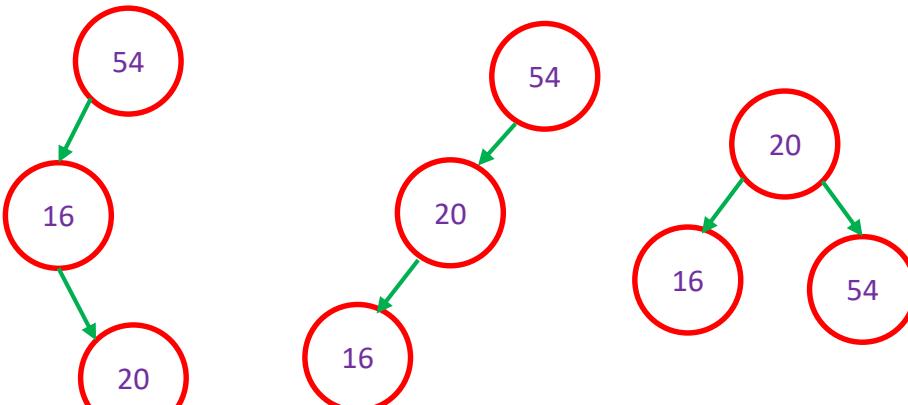
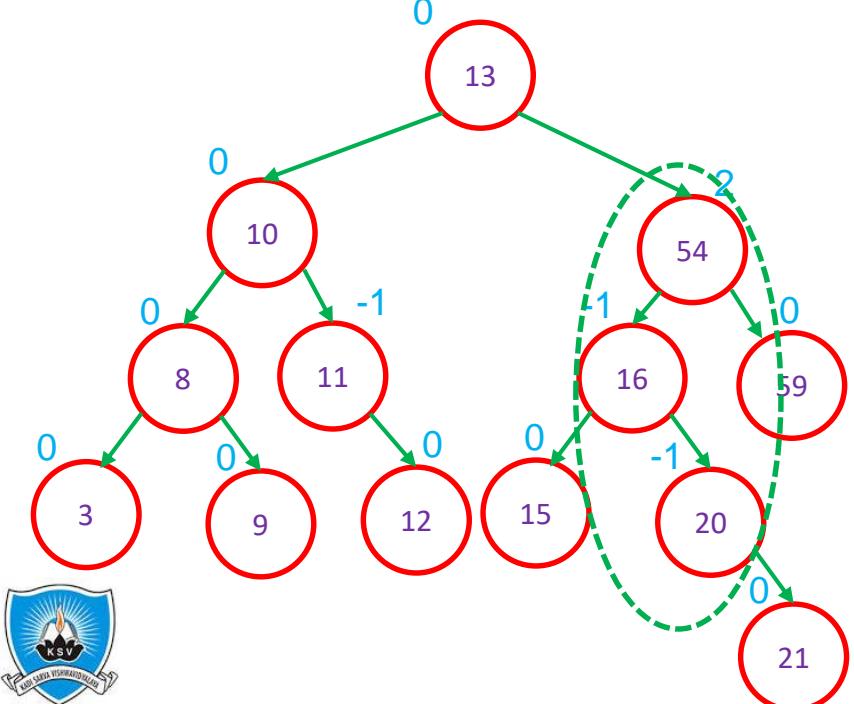
Assignment: Construct AVL Tree

13, 16, 10, 8, 54, ,3, 12, 11, 9, 59, 20, 15, 21



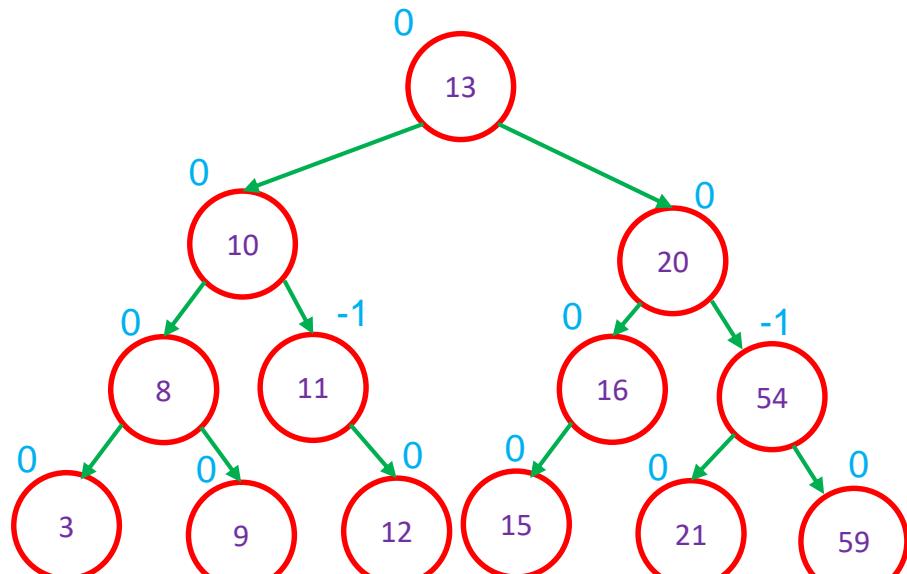
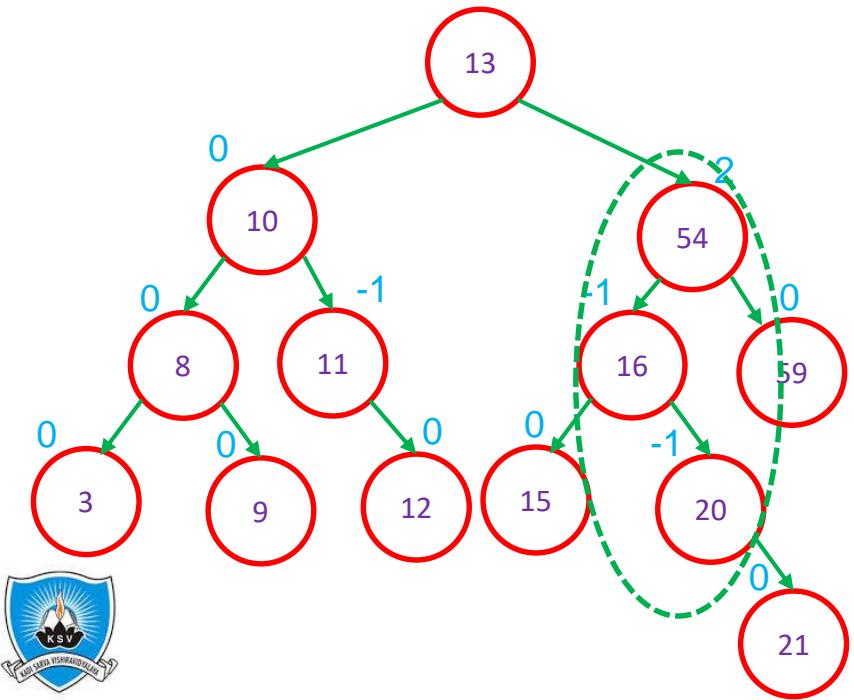
Assignment: Construct AVL Tree

13, 16, 10, 8, 54, ,3, 12, 11, 9, 59, 20, 15, 21



Assignment: Construct AVL Tree

13, 16, 10, 8, 54, ,3, 12, 11, 9, 59, 20, 15, 21





AVL Tree: Implementation

```
struct node
{
int data;
struct node *left;
struct node *right;
int height;
};

int max(int a, int b)
{
return (a > b)? a : b;
}

int height(struct node *N)
{
if (N == NULL) return 0;
return N->height;
}

struct node* newnode(int data)
{
struct node* node = (struct node*) malloc(sizeof(struct node));
node->data = data;
node->left = NULL;
node->right = NULL;
node->height = 1;
return(node);
}
```



AVL Tree: Implementation

```
struct node *rightRotate(struct node *t1)
{
    struct node *t2 = t1->left;
    struct node *t3 = t2->right;
    t2->right = t1;
    t1->left = t3;
    t1->height = max(height(t1->left), height(t1->right))+1;
    t2->height = max(height(t2->left), height(t2->right))+1;
    return t2;
}
struct node *leftRotate(struct node *t1)
{
    struct node *t2 = t1->right;
    struct node *t3 = t2->left;
    t2->left = t1;
    t1->right = t3;
    t1->height = max(height(t1->left), height(t1->right))+1;
    t2->height = max(height(t2->left), height(t2->right))+1;
    return t2;
}
int getBalanceFactor(struct node *N)
{
    if (N == NULL) return 0;
    return height(N->left) - height(N->right);
}
```



AVL Tree: Implementation

```
struct node* insert(struct node* node, int data)
{
if (node == NULL) return(newnode(data));
if (data < node->data) node->left = insert(node->left, data);
else if (data > node->data) node->right = insert(node->right, data);
else return node;

node->height = 1 + max(height(node->left), height(node->right));

int balance = getBalanceFactor(node);

/* Left Left Case */
if (balance > 1 && data < node->left->data) return rightRotate(node);
/* Right Right Case */
if (balance < -1 && data > node->right->data) return leftRotate(node);
/* Left Right Case */
if (balance > 1 && data > node->left->data)
{node->left = leftRotate(node->left); return rightRotate(node);}
/* Right Left Case */
if (balance < -1 && data < node->right->data)
{node->right = rightRotate(node->right); return leftRotate(node);}
return node;
}
```



AVL Tree: Implementation

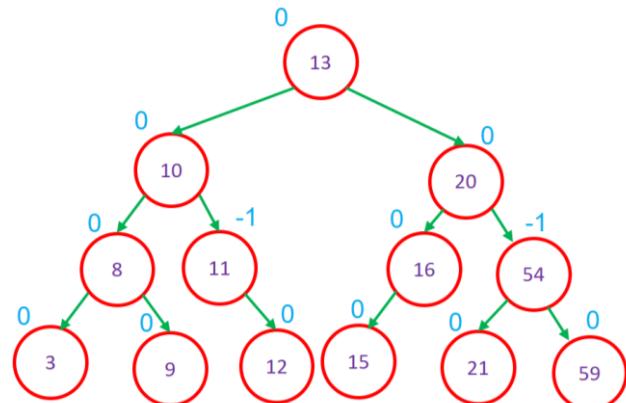
```
void preOrder(struct node *root)
{
    if(root != NULL)
    {
        printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
}
void inOrder(struct node *root)
{
    if(root != NULL)
    {
        inOrder(root->left);
        printf("%d ", root->data);
        inOrder(root->right);
    }
}
void postOrder(struct node *root)
{
    if(root != NULL)
    {
        postOrder(root->left);
        postOrder(root->right);
        printf("%d ", root->data);
    }
}
```



AVL Tree: Implementation

```
int main()
{
    struct node *root = NULL;
    root = insert(root, 13);
    root = insert(root, 16);
    root = insert(root, 10);
    root = insert(root, 8);
    root = insert(root, 54);
    root = insert(root, 3);
    root = insert(root, 12);
    root = insert(root, 11);
    root = insert(root, 9);
    root = insert(root, 59);
    root = insert(root, 20);
    root = insert(root, 15);
    root = insert(root, 21);
    printf("\nPreorder traversal of the constructed AVL tree is\n");
    preOrder(root);
    printf("\nInorder traversal of the constructed AVL tree is\n");
    inOrder(root);
    printf("\nPostorder traversal of the constructed AVL tree is\n");
    postOrder(root);
    return 0;
}
```

```
Preorder traversal of the constructed AVL tree is
13 10 8 3 9 11 12 20 16 15 54 21 59
Inorder traversal of the constructed AVL tree is
3 8 9 10 11 12 13 15 16 20 21 54 59
Postorder traversal of the constructed AVL tree is
3 9 8 12 11 10 15 16 21 59 54 20 13
```



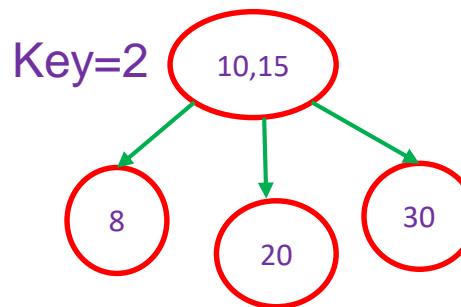
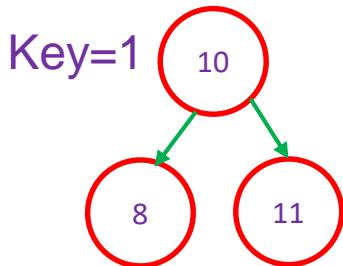


Thank You

Next: B-Tree

B-Tree

- Balanced m-way tree (where m is order of the tree)
- Generalization of BST in which a node can have more than two children and more than one key



B-Tree

- Maintains data in sorted order
- All the leaf nodes must be at the same level
- B-tree of order ‘m’ has following properties:

Children

- Every node has maximum ‘m’ children
- For leaf, maximum children = 0
- For root, maximum children = 2
- For internal node, minimum children = $[m/2]$ (upper ceiling)

Keys

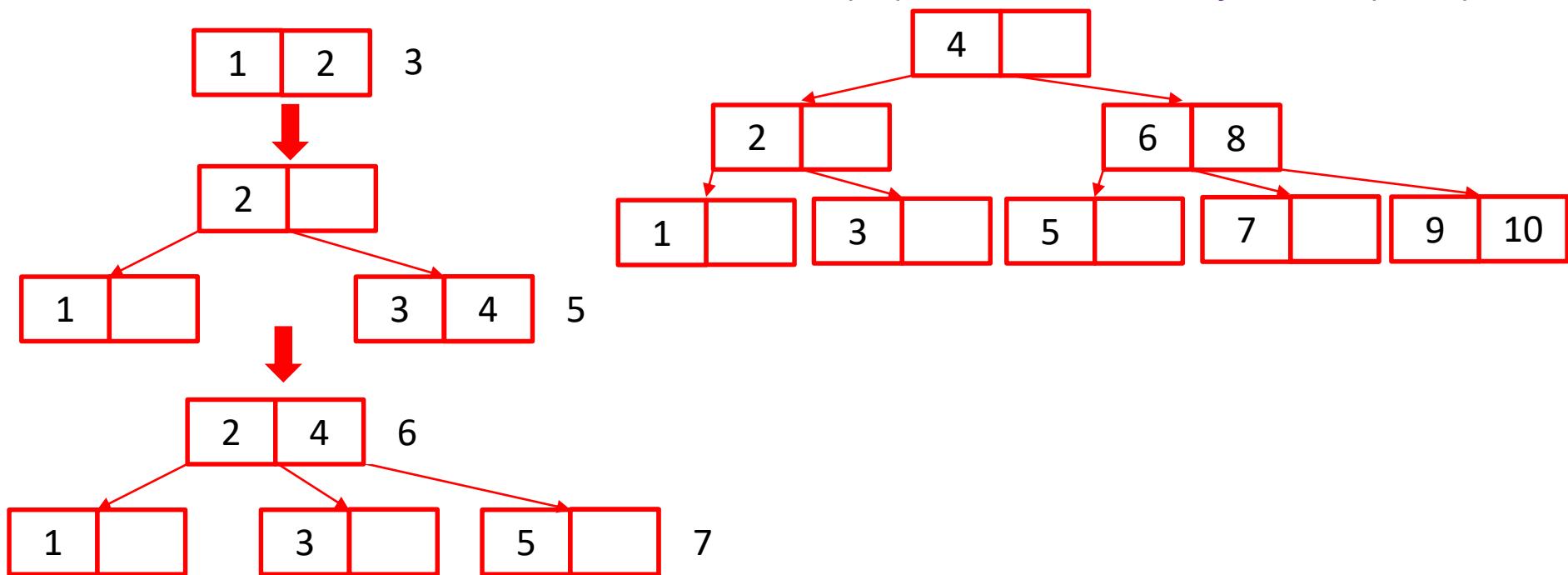
- Every node has maximum $(m-1)$ keys
- For root node, minimum key = 1
- For all other nodes, minimum number of keys = $[m/2]-1$ (upper ceiling)





B-Tree

- Create a B-Tree of order 3 by inserting values from 1 to 10
- Order $m=3$, maximum children = 3 (m), maximum keys = 2 ($m-1$)





B-Tree: Exercise

- Example: 10, 20, 40, 50, 60, 70, 80, 30, 35, 5, 15
- Order m=4, maximum children = 4 (m), maximum keys = 3 (m-1)



B-Tree: Exercise

- Example: 5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8
- Order m=4, maximum children = 4 (m), maximum keys = 3 (m-1)



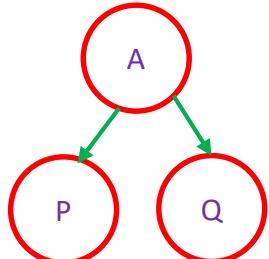
Thank You

Next: 2-3 Tree

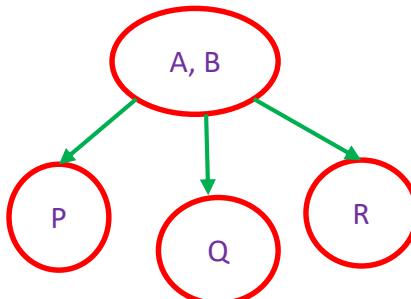
2-3 Tree : Introduction

- Similar to BST where $L < V < R$
- It contains 3 different kinds of nodes
 - (1) leaf node (2) 2-node (3) 3-node

2-node: 2 Children + 1 data element



3-node: 3 Children + 2 data element



2-3 Tree : Properties

Properties of 2-3 Tree are

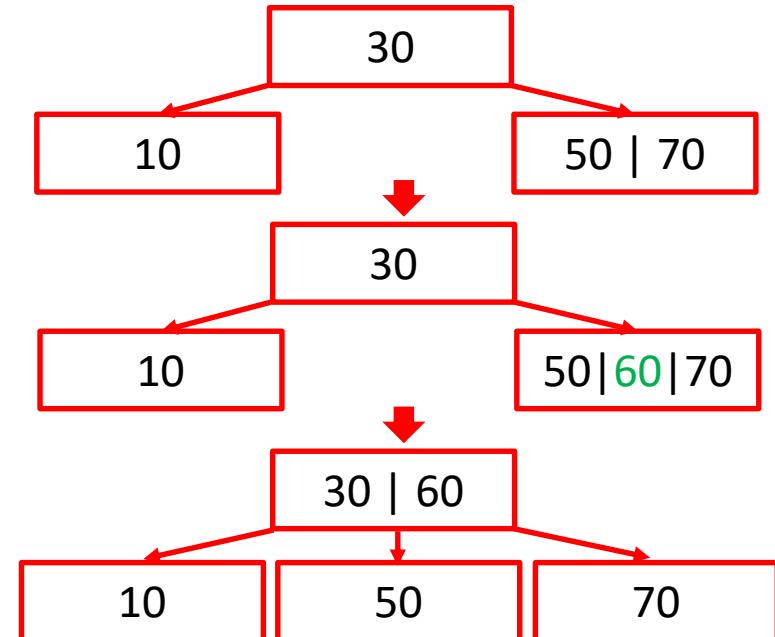
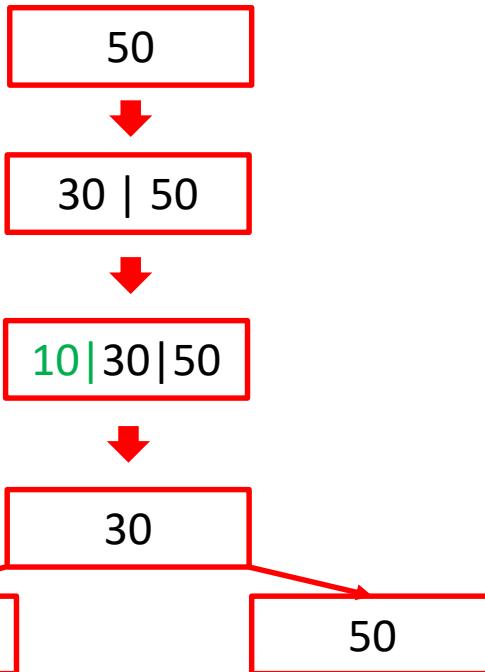
- It is a B-tree with order 3 (Max 2 keys and 3 children)
- Balanced tree
- Every internal node is either 2-node or 3-node
- All leaves are at same level
- Data is stored in sorted manner





2-3 Tree : Example

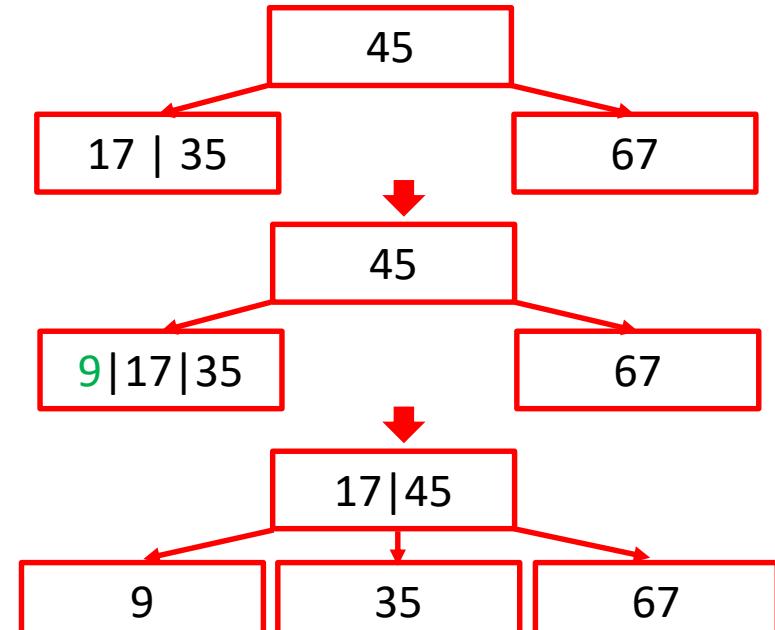
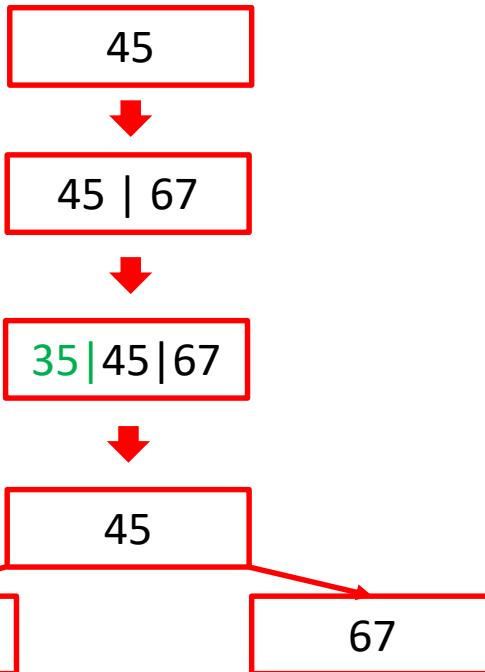
50, 30, 10, 70, 60





2-3 Tree : Example

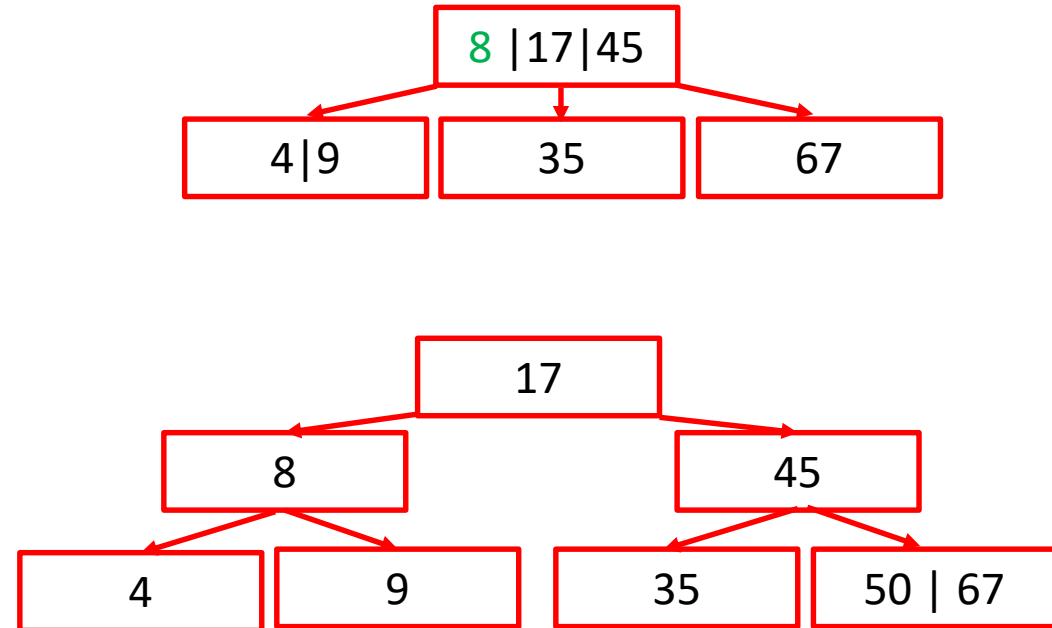
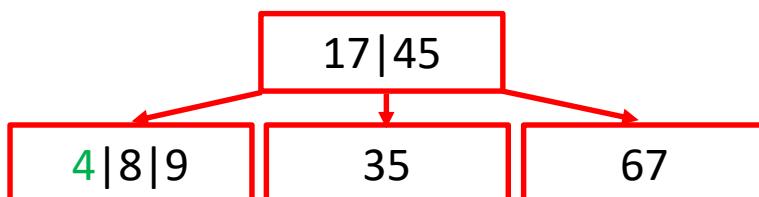
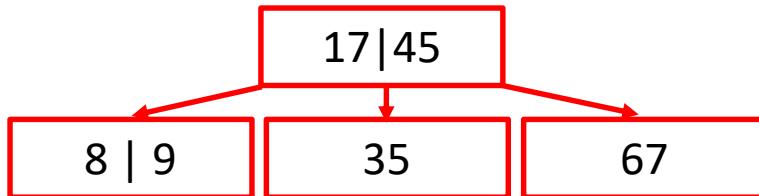
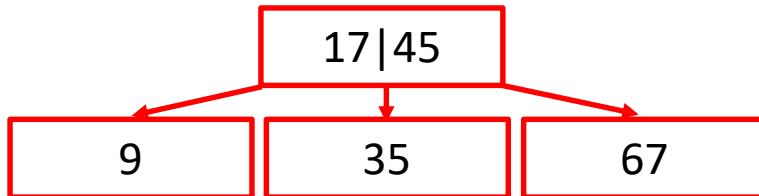
45, 67, 35, 17, 9, 8 ,4, 50





2-3 Tree : Example

45, 67, 35, 17, 9, 8 ,4, 50





2-3 Tree : Algorithm

Step 1: If the tree is empty, create a node and put the value into it

Step 2: Find the appropriate leaf node for the value

Step 3: If the leaf node has only one value, put the value into node

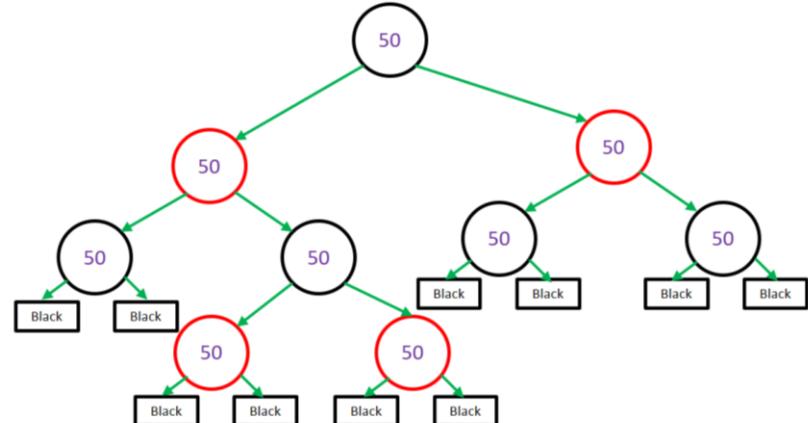
Step 4: If the leaf node has more than two values, split the node and promote the median to the parent node

Step 5: If the parent node has more than two values, continue splitting & promoting for the parent node by forming a new node if necessary

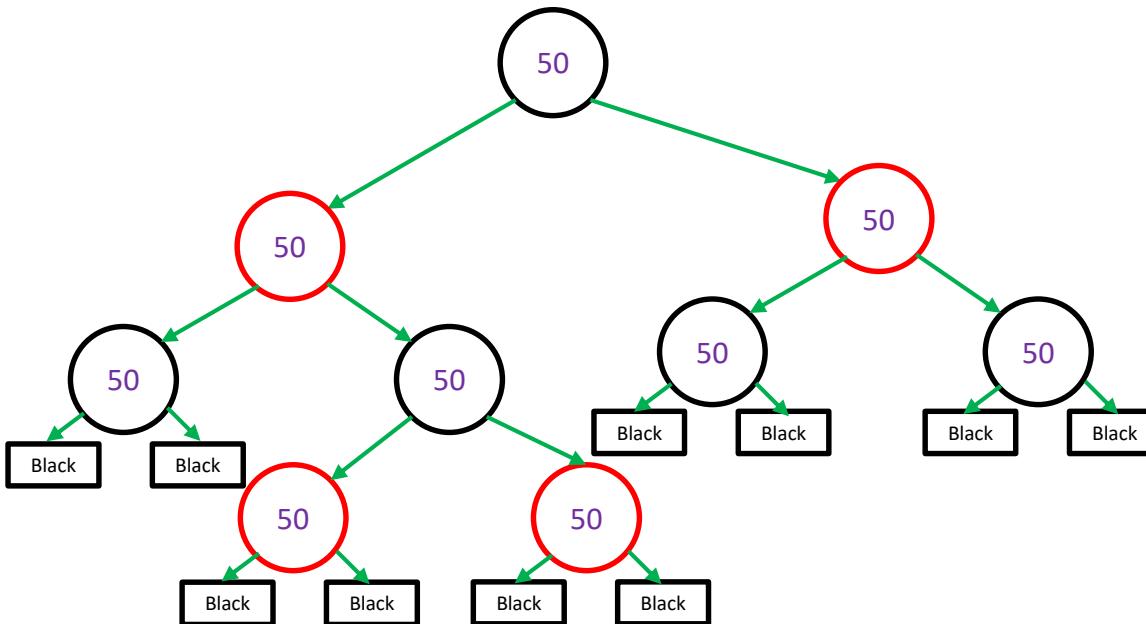
Red-Black Tree

Red-Black Tree is

- Binary Search Tree (BST)
[$L < V < R$, No. of Children ≤ 2]
- Self-balancing (E.g. AVL)
- Every node is either red or black
- Root is always black
- Every leaf which is NIL is black
- If node is red, then its children are black (no red-red parent-child relation)
- Every path from a node to any of its descendant NIL node has same number of black nodes



Red-Black Tree





Thank You

Next: Graph

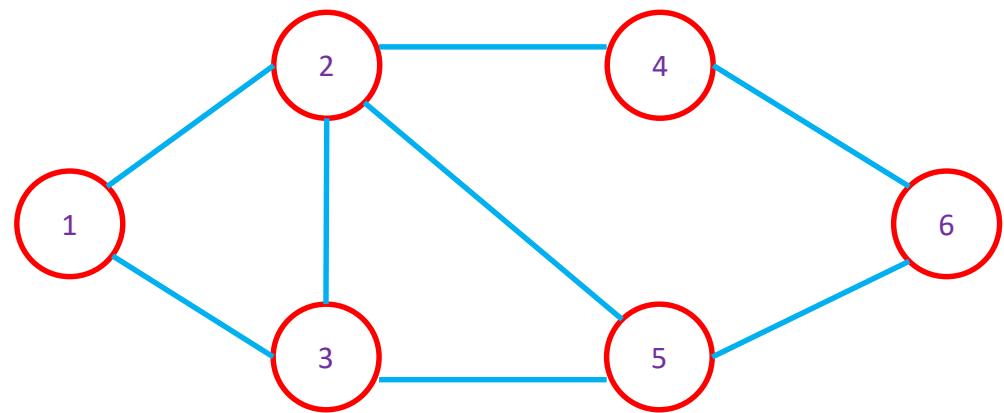
Graph

A Graph is a non-linear data structure consisting of nodes (vertices) and edges (lines or arcs)

$$G = \{V, E\}$$

$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{E_{12}, E_{13}, E_{23}, E_{24}, E_{25}, E_{35}, E_{56}, E_{46}\}$$

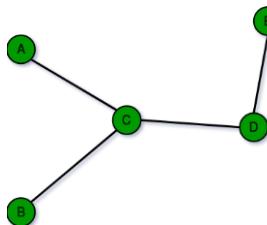




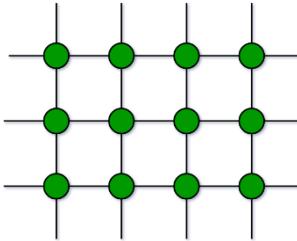
Types of Graph

www.hbpatel.in

Finite Graphs: A graph is said to be finite if it has finite number of vertices and finite number of edges



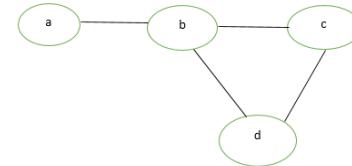
Infinite Graph: A graph is said to be infinite if it has infinite number of vertices as well as infinite number of edges.



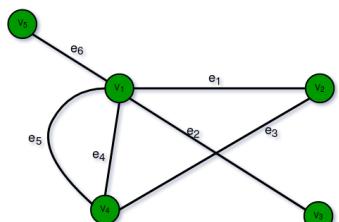
Trivial Graph: A graph is said to be trivial if a finite graph contains only one vertex and no edge.



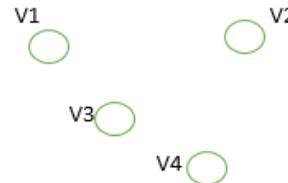
Simple Graph: A simple graph is a graph which does not contain more than one edge between the pair of vertices.



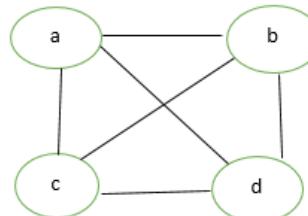
Multi Graph: Any graph which contains some parallel edges but doesn't contain any self-loop is called multi graph.



Null Graph: A graph of order n and size zero that is a graph which contains n number of vertices but do not contain any edge.



Complete/Full Graph: A simple graph with n vertices is called a complete graph if the degree of each vertex is $n-1$, that is, one vertex is attached with $n-1$ edges.



Source: www.geeksforgeeks.org

Applications of Graph

Flow of computation

Networks of communication

Data organization

Graph theory is used to find shortest path in road or a network

In Google Maps, various locations are represented as vertices or nodes and the roads are represented as edges and graph theory is used to find the shortest path between two nodes

Facebook's Graph API is perhaps the best example of application of graphs to real life problems. The Graph API is a revolution in large-scale data provision.

Flight Networks

GPS Navigation Systems

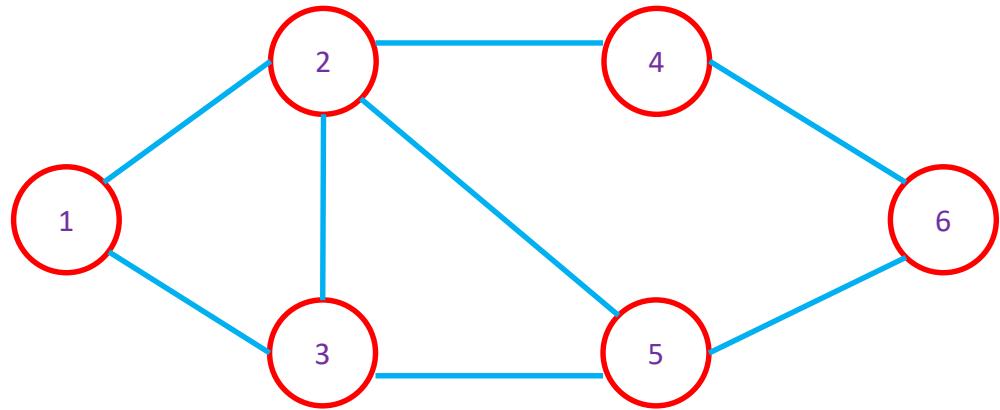


Graph Representation

- Adjacency Matrix
- Adjacency List

Adjacency Matrix

	1	2	3	4	5	6
1						



Adjacency Matrix is a matrix $A[n][n]$ ($n=\text{number of vertices}$) where,

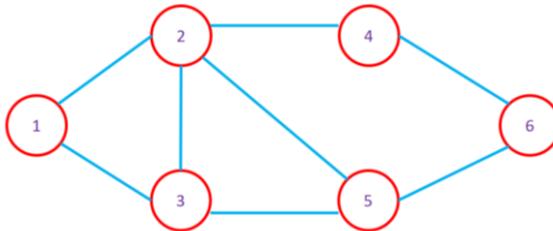
$A[i][j]=1$ if i and j are adjacent

$A[i][j]=0$ otherwise

Space Complexity = $\Theta(n^2)$

Graph Representation: Adjacency Matrix

```
#include <stdio.h>
#define size 6
void main()
{
int i,j;
int matrix[size][size]=
{
{0,1,1,0,0,0},
{1,0,1,1,1,0},
{1,1,0,0,1,0},
{0,1,0,0,0,1},
{0,1,1,0,0,1},
{0,0,0,1,1,0}
};
```



```
printf("ADJACENCY MATRIX REPRESENTATION OF A GRAPH\n");
for(i=0; i<size; ++i)
{
    printf("Neighbours of Node # %d : ",i+1);
    for(j=0; j<size; ++j)
    {
        if(matrix[i][j]==1)printf("%d ",j+1);
    }
    printf("\n");
}
```

OUTPUT

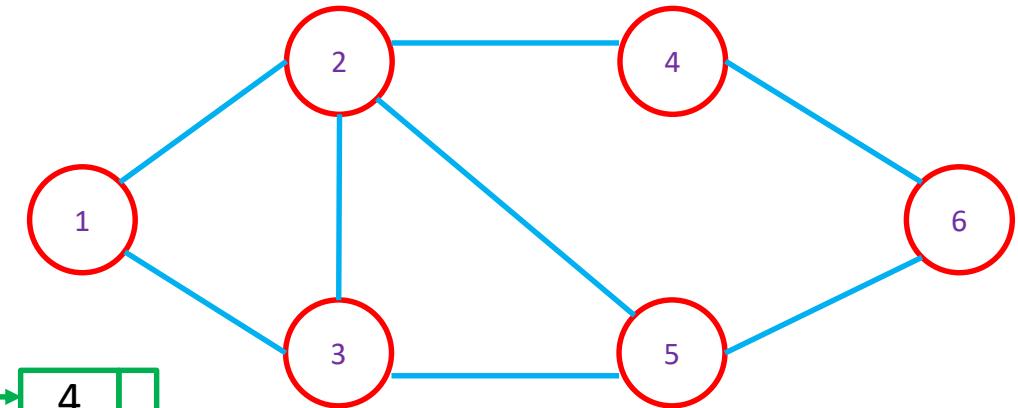
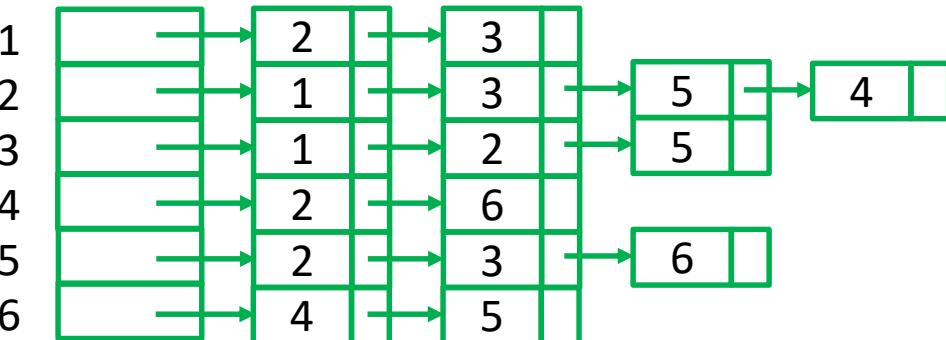
Neighbours of Node # 1 : 2 3
Neighbours of Node # 2 : 1 3 4 5
Neighbours of Node # 3 : 1 2 5
Neighbours of Node # 4 : 2 6
Neighbours of Node # 5 : 2 3 6
Neighbours of Node # 6 : 4 5



Graph Representation

- Adjacency Matrix
- Adjacency List

Adjacency List



Space Complexity = $\Theta(n + 2e)$

Graph Representation: Adjacency List

```
#include <stdio.h>
#define size 6
struct node
{
    int node;
    struct node * next;
};

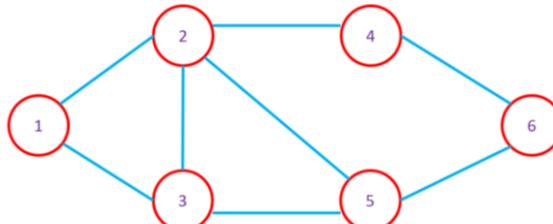
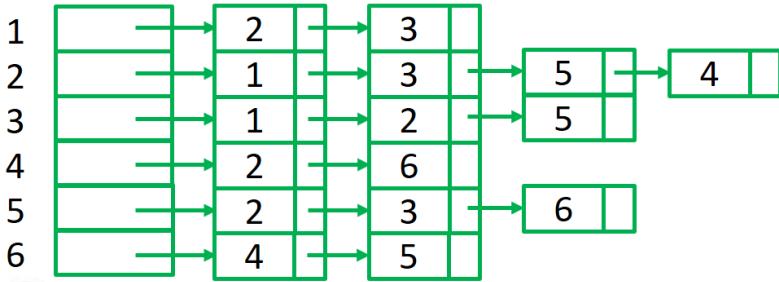
void main()
{
    int i,j;
    int matrix[size][size]=
    {
        {0,1,1,0,0,0},
        {1,0,1,1,1,0},
        {1,1,0,0,1,0},
        {0,1,0,0,0,1},
        {0,1,1,0,0,1},
        {0,0,0,1,1,0}
    };
    struct node *list[size],*newnode,*temp;
    for(i=0; i<size; ++i)list[i]=NULL;
    for(i=0; i<size; ++i)
        for(j=0; j<size; ++j)
            if(matrix[i][j]==1)
                {
                    newnode = (struct node*) malloc (sizeof(struct node));
                    newnode->node=j+1;
                    newnode->next=NULL;
                    if(list[i]==NULL)list[i]=newnode;
                    else
                    {
                        temp=list[i];
                        while(temp->next != NULL) temp=temp->next;
                        temp->next=newnode;
                    }
                }
    printf("\n");
}
```



Graph Representation: Adjacency List

```
for(i=0; i<size; ++i)
{
temp=list[i];
printf("Neighbours of Node # %d : ",i+1);
while(temp!=NULL)
{
    printf("%d\t",temp->node);
    temp=temp->next;
}

printf("\n");
}
```



OUTPUT

```
Neighbours of Node # 1 : 2 3
Neighbours of Node # 2 : 1 3 4 5
Neighbours of Node # 3 : 1 2 5
Neighbours of Node # 4 : 2 6
Neighbours of Node # 5 : 2 3 6
Neighbours of Node # 6 : 4 5
```



Graph Representation

Dense Graph → Adjacency Matrix

Sparse Graph → Adjacency List

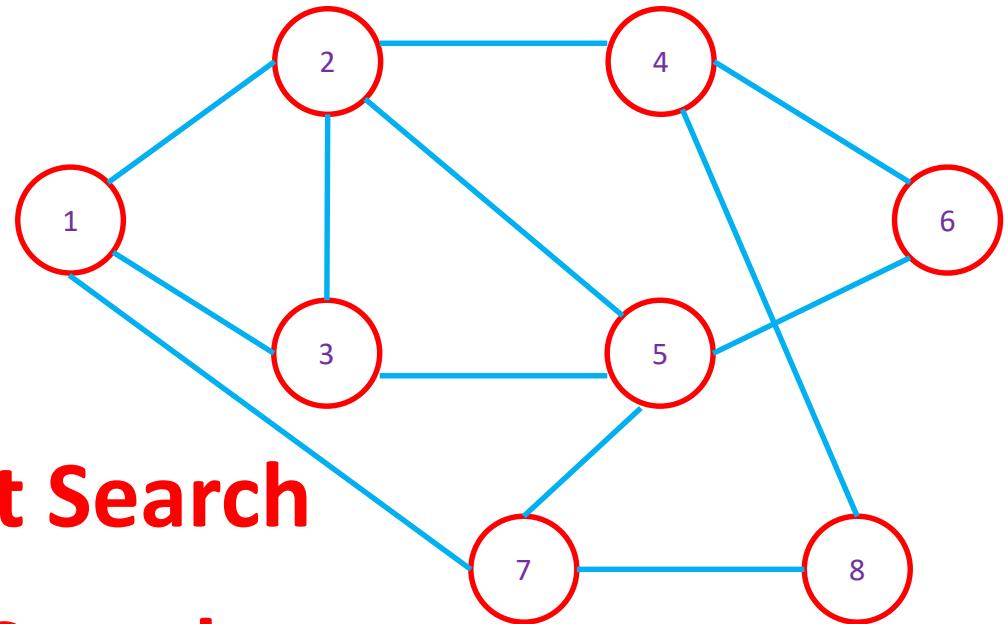




Thank You

Next: Graph Traversal (BFS/DFS)

Graph Traversal

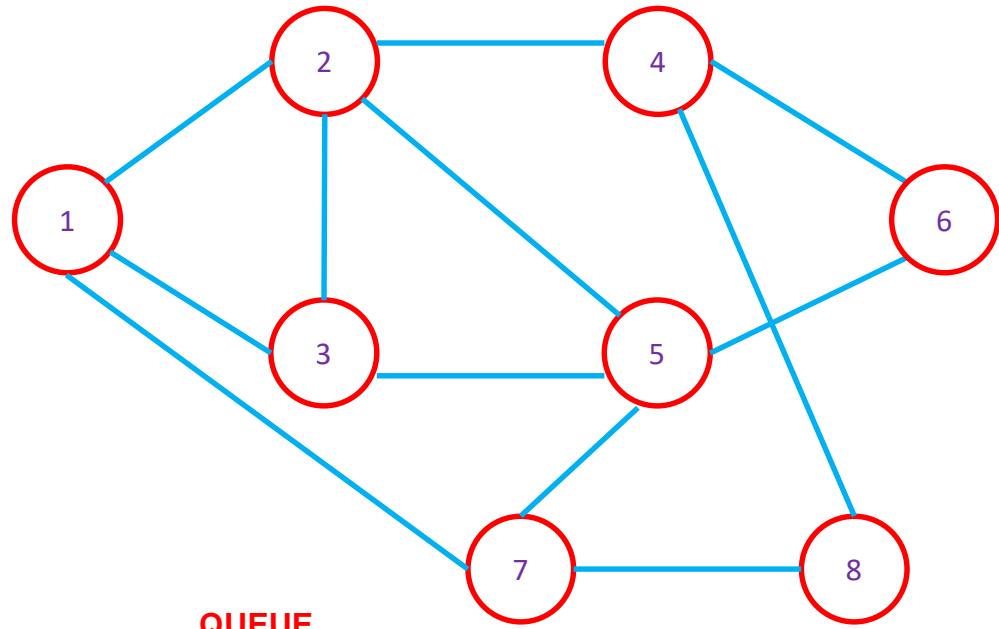


BFS : Breadth First Search

DFS : Depth First Search



BFS : Breadth First Search



Level Order Search

You can start traversing from any node. (E.g. 1 in our case)

Data Structure Used: QUEUE (FIFO)

1	2	3	7	4	5	8	6
---	---	---	---	---	---	---	---

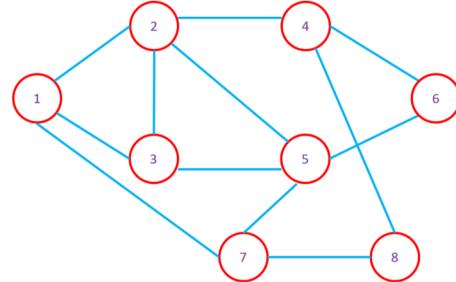
1	2	3	7	4	5	8	6
---	---	---	---	---	---	---	---

BFS : Breadth First Search

```
#include <stdio.h>
#define size 8
int output[size], outputPtr=0,
queue[size], front=-1, rear=-1;
void main()
{int matrix[size][size]=
{
{0,1,1,0,0,0,1,0},
{1,0,1,1,1,0,0,0},
{1,1,0,0,1,0,0,0},
{0,1,0,0,0,1,0,1},
{0,1,1,0,0,1,1,0},
{0,0,0,1,1,0,0,0},
{1,0,0,0,1,0,0,1},
{0,0,0,1,0,0,1,0}
};

int i, j, k, start=5,temp,isthere;
void enqueue(int);
```

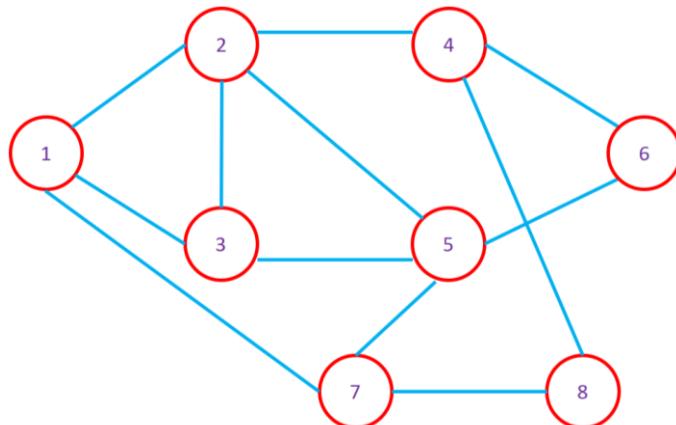
```
enqueue(start);
while(!(front== -1 && rear== -1))
{
temp=dequeue();
printf("%d\t",temp);
output[outputPtr++]=temp;
if(outputPtr>=size)break;
for(j=0; j<size; ++j)
{
if(matrix[temp-1][j]==1)
{
isthere=0;
for(k=0; k<outputPtr; ++k)
{
if(output[k]==j+1)isthere=1;
}
if(!!(front== -1 && rear== -1))
for(k=front; k<=rear; ++k)
{
if(queue[k]==j+1)isthere=1;
}
if(isthere==0)enqueue(j+1);
}
}
```



BFS : Breadth First Search

```
void enqueue(int node)
{
if(front== -1 && rear== -1) front=rear=0;
else rear++;
queue[rear]=node;
}

int dequeue(void)
{
int temp=queue[front];
if(front==0 && rear==0){front=rear=-1;}
else front++;
return temp;
}
```

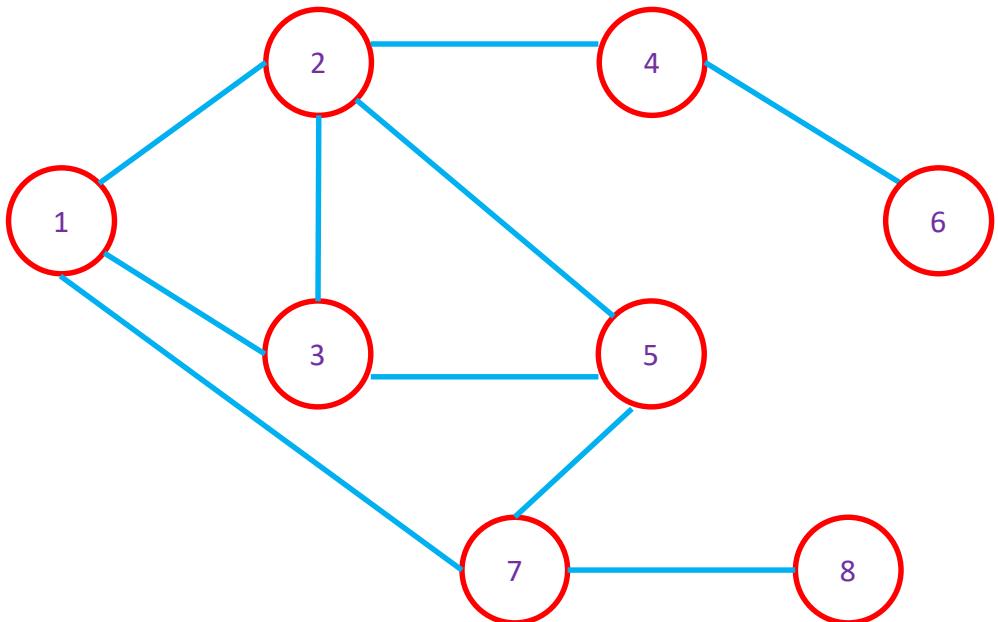


OUTPUT

```
5 2 3 6 7 1 4 8
```



DFS : Depth First Search



Select any one unvisited adjacent vertex
Back tracking

You can start traversing from any node. (E.g. 1 in our case)

Data Structure Used: STACK (LIFO)

OUTPUT

1	2	3	5	7	8	4	6
---	---	---	---	---	---	---	---



STACK



DFS : Depth First Search

```
#include <stdio.h>
#define size 8
int output[size], outputPtr=0, stack[size], top=-1;
void main()
{int matrix[size][size]={{0,1,1,0,0,0,1,0},
{1,0,1,1,1,0,0,0},
{1,1,0,0,1,0,0,0},
{0,1,0,0,0,1,0,1},
{0,1,1,0,0,1,1,0},
{0,0,0,1,1,0,0,0},
{1,0,0,0,1,0,0,1},
{0,0,0,1,0,0,1,0}
};

int i, j, k, start=1,temp,isthere,alreadyAnElementPushed;
void push(int);
push(start);

while(top!=-1)
{temp=pop();
printf("%d\t",temp);
output[outputPtr++]=temp;
if(outputPtr>=size)break;
for(j=0; j<size; ++j)
{alreadyAnElementPushed=0;
if(matrix[temp-1][j]==1)
{isthere=0;
for(k=0; k<outputPtr; ++k)
{if(output[k]==j+1)isthere=1;}
if(top!=-1)
for(k=0; k<=top; ++k)
{if(stack[k]==j+1)isthere=1;}
if(isthere==0)
{push(j+1);
alreadyAnElementPushed=1;
}
}
if(alreadyAnElementPushed==1)break;
}
}
}

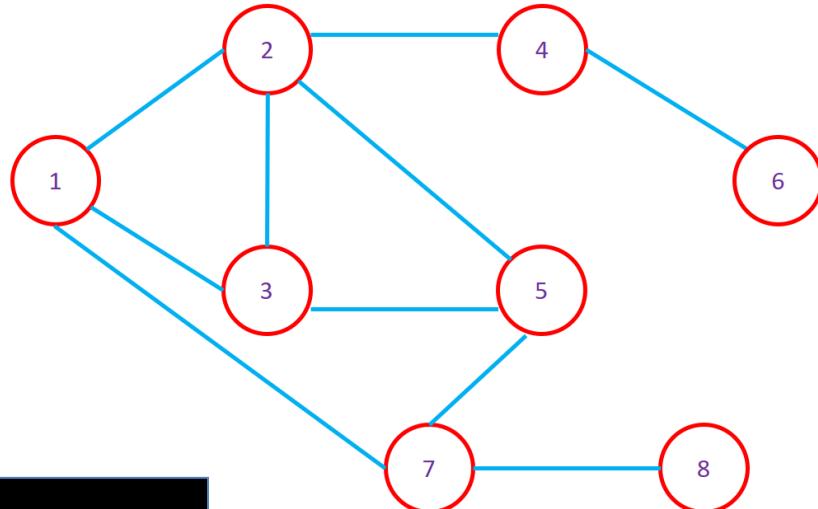
Source code: https://github.com/hbpatel1976/Data-Structure/blob/master/GRPH\_DFS.C
```



DFS : Depth First Search

```
void push(int node)
{
    stack[++top]=node;
}

int pop(void)
{
    return stack[top--];
}
```



OUTPUT

```
1 2 3 5 6 4 8 7
```



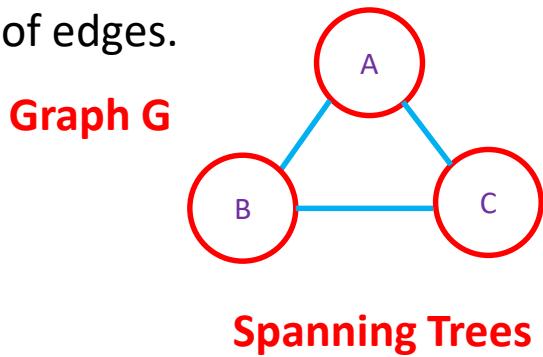


Thank You

Next: Spanning Tree

Spanning Tree

Spanning Tree: A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges.



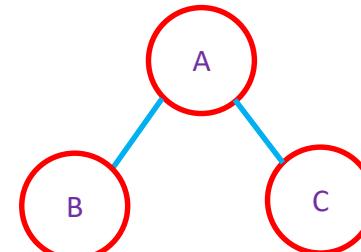
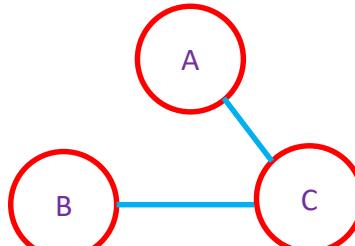
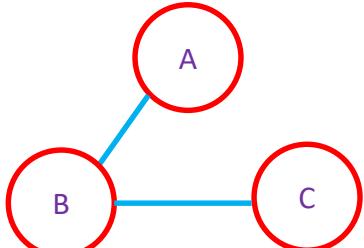
Graph G = (V, E)

Spanning Tree G' = (V', E')

$V' = V$

$E' \subset E$

$E' = |V| - 1$



Spanning Tree

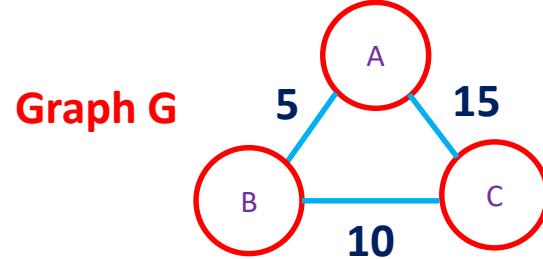
General Properties of Spanning Tree:

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.
- Maximum number of spanning tree = n^{n-2} (n = number of vertices)
- From a complete graph, by removing max $(e-n+1)$ edges, we can construct spanning a tree.

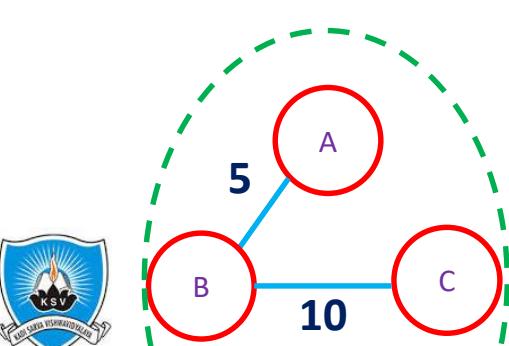


Minimum Spanning Tree

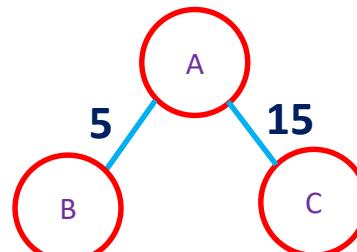
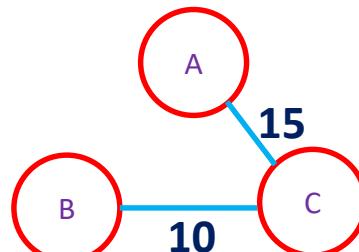
In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph.



MST



Spanning Trees



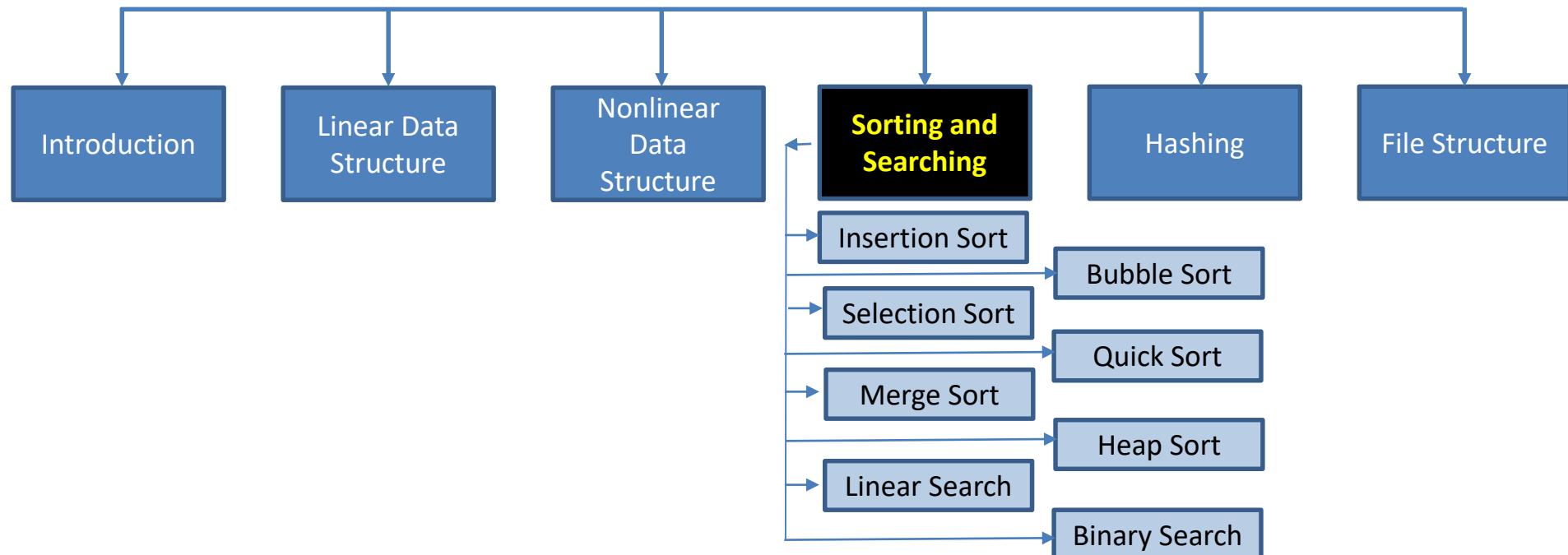


Thank You

Next: Sorting & Searching



DSA: Sorting and Searching





Why Sorting?



Image source: wikipedia.org

Udaipur, Rajasthan, India

Sort by

Featured ⓘ

Star rating

Distance

Guest rating

Price

Price (high to low)

Price (low to high)

Image source: in.hotels.com

Home > Home Entert... > Televisions

Televisions (Showing 1 – 24 products of 459 products)

Sort By

Popularity

Price – Low to High

Price – High to Low

Newest First

Discount

Source image: flipkart.com



Source image: <https://images.app.goo.gl/TtGsxPnaaNqUYon96>

Teams	Mat	Won	Lost	Tied	NR	Pts	NRR
India	9	7	1	0	1	15	+0.809
Australia	9	7	2	0	0	14	+0.868
England	9	6	3	0	0	12	+1.152
New Zealand	9	5	3	0	1	11	+0.175
Pakistan	9	5	3	0	1	11	-0.430
Sri Lanka	9	3	4	0	2	8	-0.919
South Africa	9	3	5	0	1	7	-0.030
Bangladesh	9	3	5	0	1	7	-0.410
West Indies	9	2	6	0	1	6	-0.225
Afghanistan	9	0	9	0	0	0	-1.322

Source image: cricbuzz.com

return Flight
Bengaluru to New Delhi | Tue, 07 Jul | Sorted By: Price

Price	Low to High	<input checked="" type="checkbox"/>
AirAsia I5-722	00:05	02 hrs 40 mins
Go Air G8-116	09:35	02 hrs 40 mins

Duration Low to High

Departure (BLR) Early Late

Arrival (DEL) Early Late

Source Image: makemytrip.com



Source image: thycotic.com

www.hbpatel.in

Data Structures and Algorithms

Selection Sort

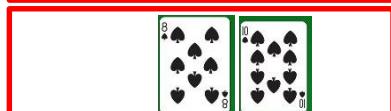
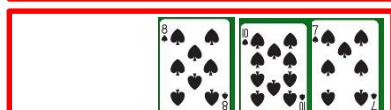
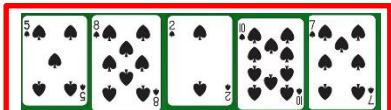




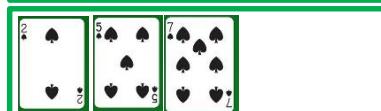
Selection Sort

- Simplest sorting algorithm

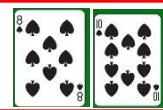
Unsorted (Left Hand)



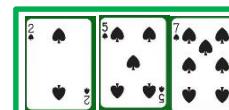
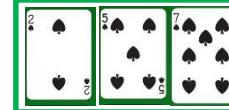
Sorted (Right Hand)



Unsorted (Left Hand)



Sorted (Right Hand)





Selection Sort

42	23	74	11	65	58
----	----	----	----	----	----

42	23	74	11	65	58
----	----	----	----	----	----

42	23	74	11	65	58
----	----	----	----	----	----

42	23	74	11	65	58
----	----	----	----	----	----

42	23	74	11	65	58
----	----	----	----	----	----

42	23	74	11	65	58
----	----	----	----	----	----

42	23	74	11	65	58
----	----	----	----	----	----

--	--	--	--	--	--

11					
----	--	--	--	--	--

11	23				
----	----	--	--	--	--

11	23	42			
----	----	----	--	--	--

11	23	42	58		
----	----	----	----	--	--

11	23	42	58	65	
----	----	----	----	----	--

11	23	42	58	65	74
----	----	----	----	----	----

COPY



11	23	42	58	65	74
----	----	----	----	----	----

11	23	42	58	65	74
----	----	----	----	----	----



Selection Sort

Find the minimum and exchange with appropriate position

42	23	74	11	65	58
----	----	----	----	----	----

11	23	74	42	65	58
----	----	----	----	----	----

11	23	42	58	65	74
----	----	----	----	----	----

42	23	74	11	65	58
----	----	----	----	----	----

11	23	74	42	65	58
----	----	----	----	----	----

11	23	42	58	65	74
----	----	----	----	----	----

11	23	74	42	65	58
----	----	----	----	----	----

11	23	42	74	65	58
----	----	----	----	----	----

11	23	42	58	65	74
----	----	----	----	----	----

11	23	74	42	65	58
----	----	----	----	----	----

11	23	42	74	65	58
----	----	----	----	----	----

11	23	42	58	65	74
----	----	----	----	----	----



Selection Sort: Algorithm

```
procedure selectionSort(A:array, n: size)
for i = 0 to size(A) - 2 do:
    iMin <- I

    for j = i+1 to size(A) - 1 do:
        if (A[j] > A[iMin])
            then iMin <- j
    end for

    swap (A[i], A[iMin])
end for
end procedure
```



Selection Sort: Program

```
#include<stdio.h>
int main()
{
int i, j, iMin, temp, size=6;
int A[25]={42, 23, 74, 11, 65, 58};

for(i=0;i<=size-2;i++)
{
    iMin = i;
    for(j=i+1; j<=size-1; ++j)
        {
            if(A[j]<A[iMin]) iMin = j;
        }
    temp=A[i];
    A[i]=A[iMin];
    A[iMin]=temp;
}
printf("Order of Sorted elements: ");
for(i=0;i<size;i++) printf(" %d",A[i]);
return 0;
}
```

i	j	iMin	A[j]	A[iMin]	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
					42	23	74	11	65	58
0	1	0	23	42						
0	2	1	74	23						
0	3	3	11	23						
0	4	3	65	11						
0	5	3	58	11						
					11	23	74	42	65	58
1	2	1	74	23						
1	3	1	42	23						
1	4	1	65	23						
1	5	1	58	23						
					11	23	74	42	65	58
2	3	2	42	74						
2	4	3	42	65						
2	5	3	42	58						
					11	23	42	74	65	58
3	4	3	65	75						
3	5	4	65	58						
		5			11	23	42	58	65	74
4	5	4	58	75						
					11	23	42	58	65	74

OUTPUT

Order of Sorted elements: 11 23 42 58 65 74



Thank You

Next: Bubble Sort

Data Structures and Algorithms

Bubble Sort





Bubble Sort

Compare two adjacent elements and swap them if required

42	23	74	11	65	58
42	23	74	11	65	58

23	42	11	65	58	74
23	42	11	65	58	74

11	23	42	58	65	74
11	23	42	58	65	74

23	42	74	11	65	58
23	42	74	11	65	58

23	11	42	65	58	74
23	11	42	65	58	74

11	23	42	58	65	74
11	23	42	58	65	74

23	42	11	74	65	58
23	42	11	74	65	58

23	11	42	65	58	74
23	11	42	65	58	74

11	23	42	58	65	74
11	23	42	58	65	74

23	42	11	74	65	58
23	42	11	74	65	58

23	11	42	58	65	74
23	11	42	58	65	74

11	23	42	58	65	74
11	23	42	58	65	74

23	42	11	65	74	58
23	42	11	65	74	58

23	11	42	58	65	74
23	11	42	58	65	74

11	23	42	58	65	74
11	23	42	58	65	74

23	42	11	65	74	58
23	42	11	65	74	58

23	11	42	58	65	74
23	11	42	58	65	74

11	23	42	58	65	74
11	23	42	58	65	74

23	42	11	65	58	74
23	42	11	65	58	74

23	11	42	58	65	74
23	11	42	58	65	74

11	23	42	58	65	74
11	23	42	58	65	74

23	42	11	65	58	74
23	42	11	65	58	74

11	23	42	58	65	74
11	23	42	58	65	74



Bubble Sort: Algorithm

```
procedure bubbleSort(A : array of items, n: size of
array)
for i = 0 to size(A) - 2 do:

    for j = 0 to size(A) - i - 2 do:
        if (A[j] > A[j+1])
            swap (A[j], A[j+1])

    end for

end for
end procedure
```



Bubble Sort: Algorithm

```
procedure bubbleSort(A : array of items, n: size of
array)
for i = 0 to size(A) - 2 do:
    flag <- 0
    for j = 0 to size(A) - i - 2 do:
        if (A[j] > A[j+1])
            swap (A[j], A[j+1])
            flag <- 1
    end for
    if flag = 0 break
end for
end procedure
```



Bubble Sort: Program

```
#include<stdio.h>
int main()
{
int i, j, temp, count=6;
int A[25]={42, 23, 74, 11, 65, 58};

for(i=0;i<count-1;i++)
{
    for(j=0; j<count-i-1; ++j)
        {
            if(A[j]>A[j+1])
                {
                    temp=A[j];
                    A[j]=A[j+1];
                    A[j+1]=temp;
                }
        }

printf("Order of Sorted elements: ");
for(i=0;i<count;i++) printf(" %d",A[i]);
return 0;
}
```

i	j	j+1	A[j]	A[j+1]	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
					42	23	74	11	65	58
0	0	1	42	23	23	42				
0	1	2	42	74		42	74			
0	2	3	74	11			11	74		
0	3	4	74	65				65	74	
0	4	5	74	58					58	74
					23	42	11	65	58	74
1	0	1	23	42						
1	1	2	42	11		11	42			
1	2	3	42	65					58	65
1	3	4	65	58					65	74
					23	11	42	58	65	74
2	0	1	23	11	11	23				
2	1	2	23	42						
2	2	3	42	58						
					11	23	42	58	65	74
3	0	1	11	23						
3	1	2	23	42						
					11	23	42	58	65	74
4	0	1	11	12						
					11	23	42	58	65	74



Bubble Sort: Assignment

Explain the trace of bubble sort on following data. 42, 23, 74, 11, 65, 58, 94, 36, 99, 87.

42	23	74	11	65	58	94	36	99	87
42	23	74	11	65	58	94	36	99	87
23	42	74	11	65	58	94	36	99	87
23	42	74	11	65	58	94	36	99	87
23	42	11	74	65	58	94	36	99	87
23	42	11	65	74	58	94	36	99	87
23	42	11	65	58	74	94	36	99	87
23	42	11	65	58	74	94	36	99	87
23	42	11	65	58	74	94	36	99	87
23	42	11	65	58	74	94	99	87	
23	42	11	65	58	74	36	94	99	87
23	42	11	65	58	74	36	94	99	87
23	42	11	65	58	74	36	94	99	87
23	42	11	65	58	74	36	94	87	99

23	42	11	65	58	74	36	94	87	99
23	42	11	65	58	74	36	94	87	99
23	11	42	65	58	74	36	94	87	99
23	11	42	65	58	74	36	94	87	99
23	11	42	58	65	74	36	94	87	99
23	11	42	58	65	74	36	94	87	99
23	11	42	58	65	74	36	94	87	99
23	11	42	58	65	74	94	87	99	
23	11	42	58	65	36	74	94	87	99
23	11	42	58	65	36	74	94	87	99
23	11	42	58	65	36	74	87	94	99
23	11	42	58	65	36	74	87	94	99

Continue on next slide..

www.hbpatel.in

Bubble Sort: Assignment



...continued from previous slide

23	11	42	58	65	36	74	87	94	99
11	23	42	58	65	36	74	87	94	99
11	23	42	58	65	36	74	87	94	99
11	23	42	58	65	36	74	87	94	99
11	23	42	58	65	36	74	87	94	99
11	23	42	58	65	36	74	87	94	99
11	23	42	58	36	65	74	87	94	99
11	23	42	58	36	65	74	87	94	99
11	23	42	58	36	65	74	87	94	99

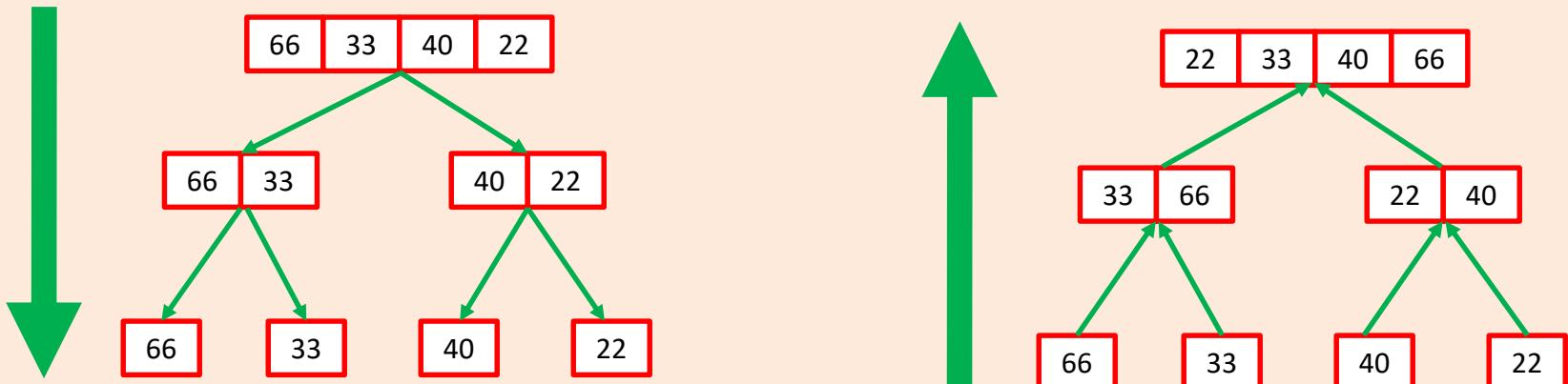
11	23	42	58	36	65	74	87	94	99
11	23	42	58	36	65	74	87	94	99
11	23	42	58	36	65	74	87	94	99
11	23	42	58	36	65	74	87	94	99
11	23	42	36	58	65	74	87	94	99



Thank You

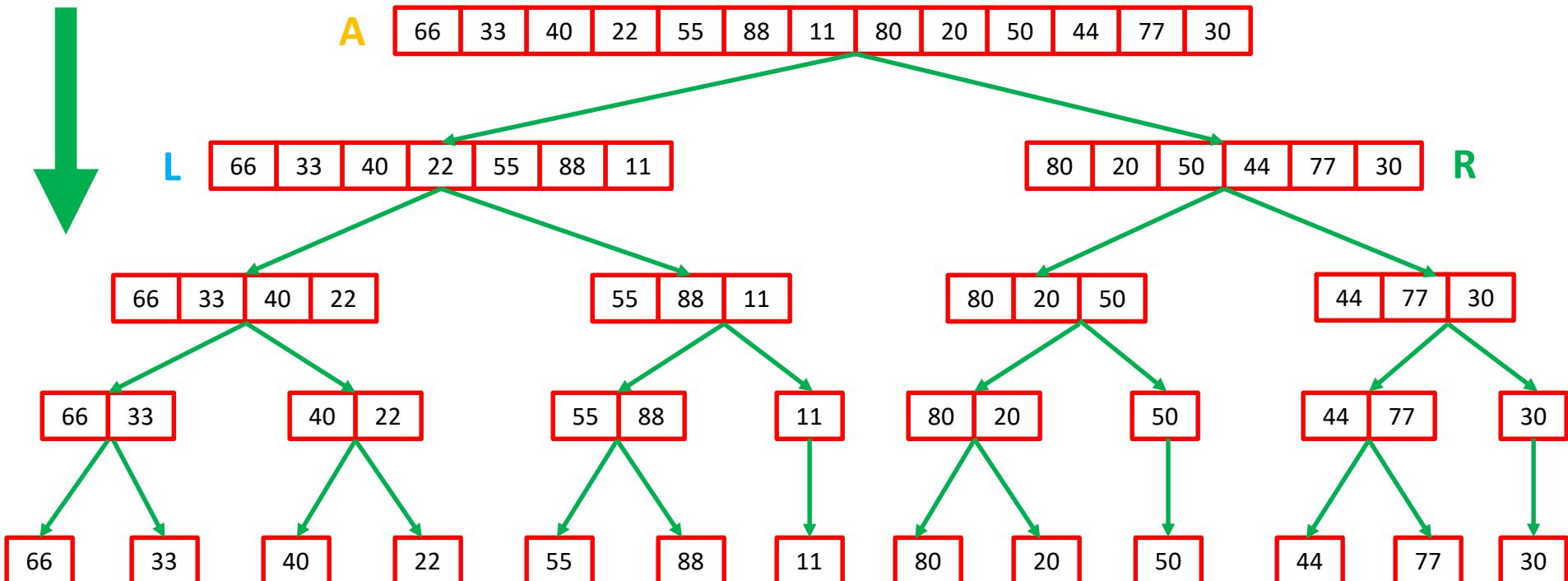
Next: Merge Sort

Merge Sort



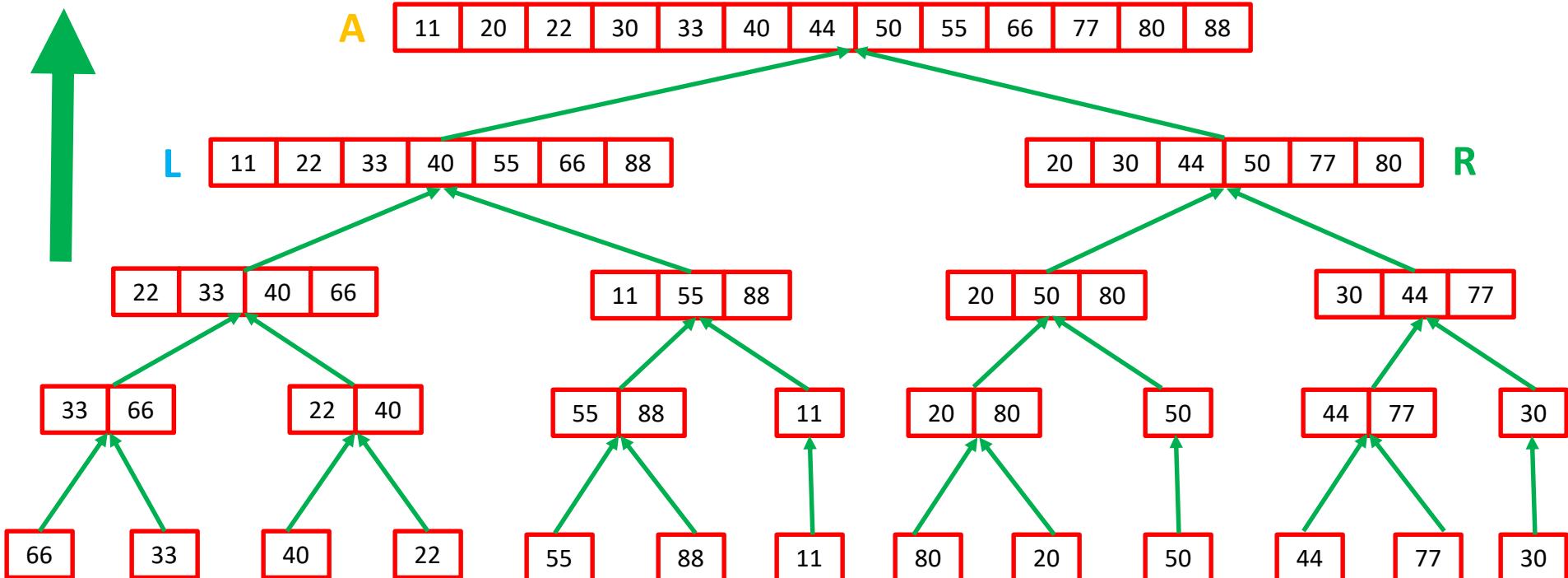


Merge Sort



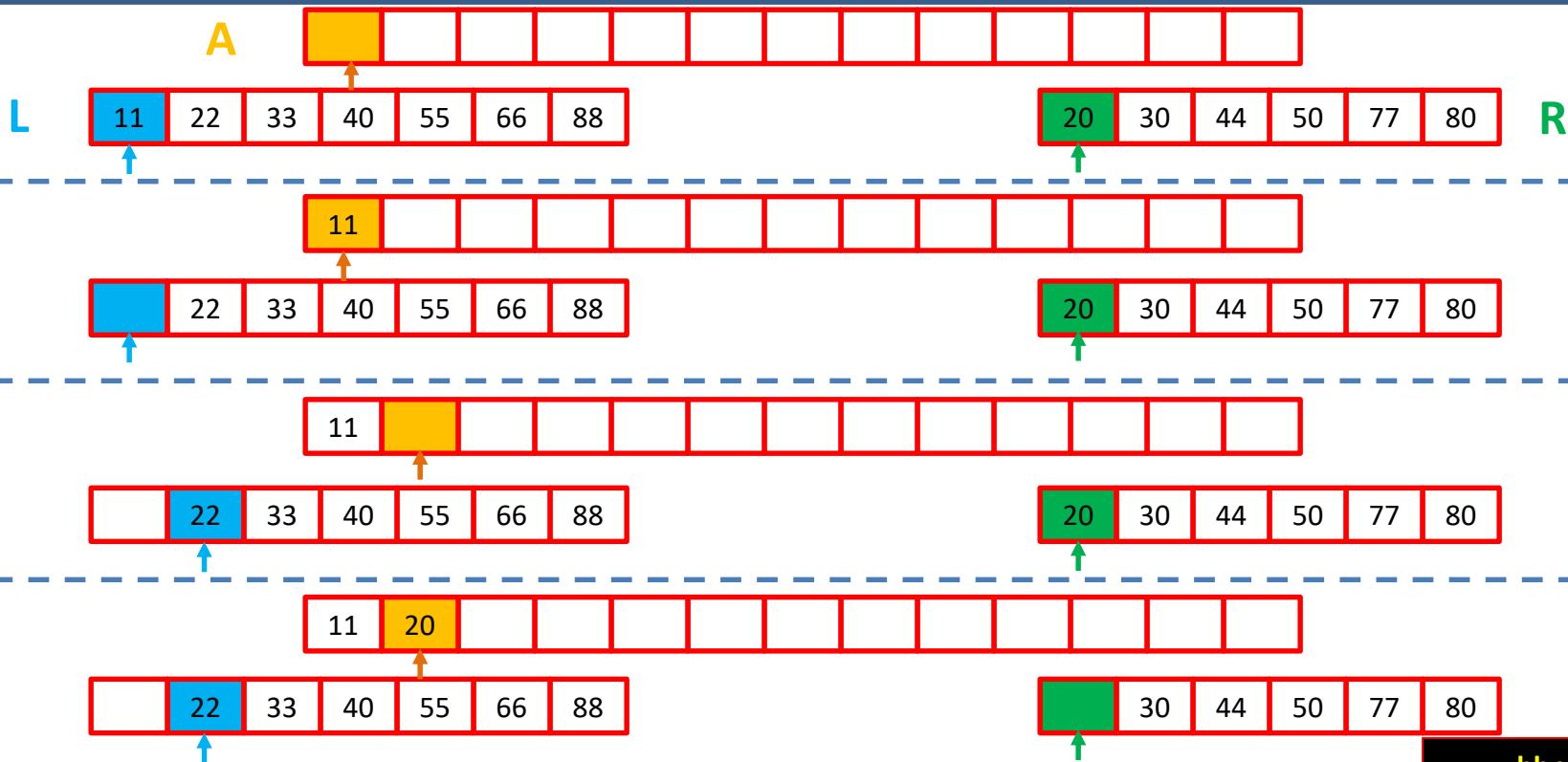


Merge Sort



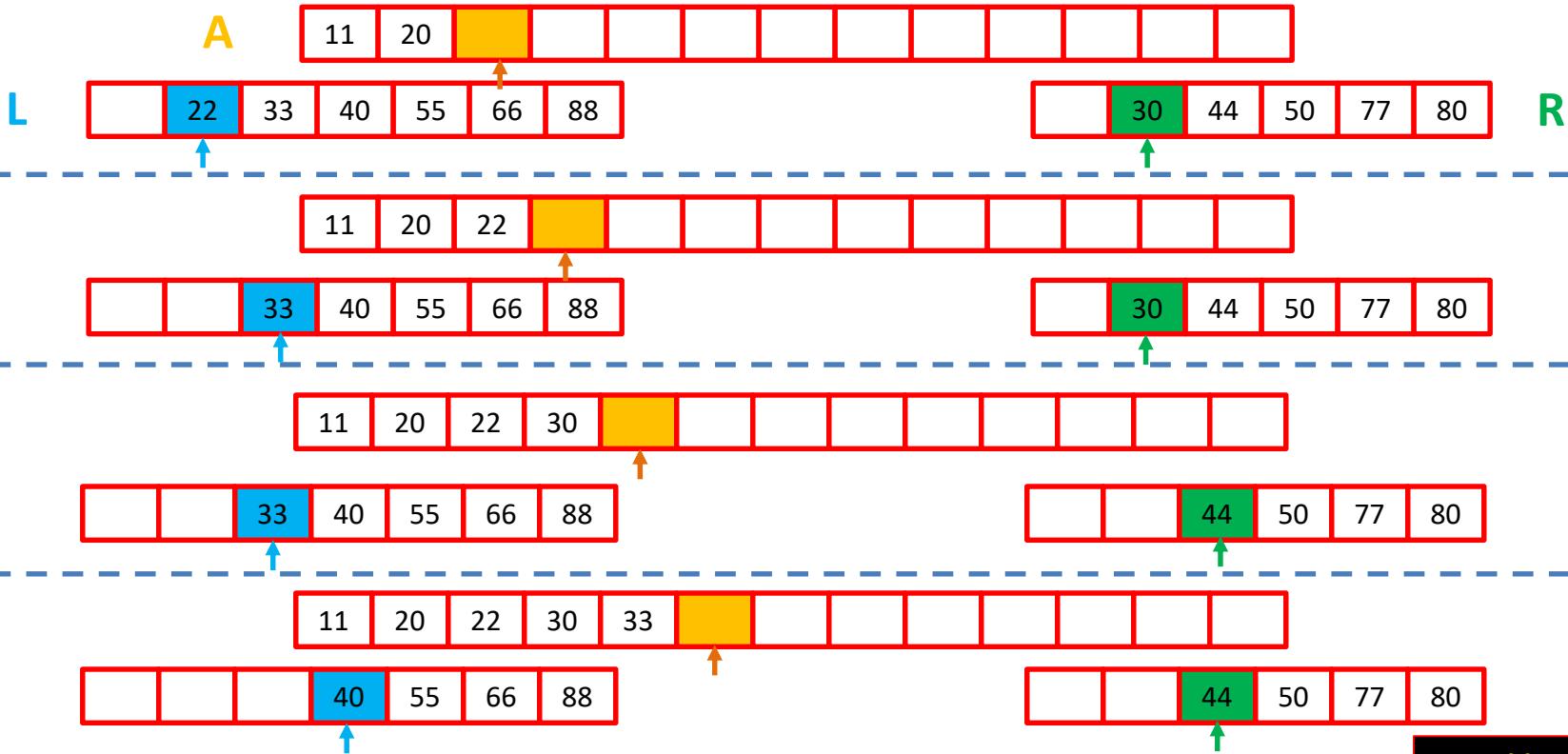


Merge Sort



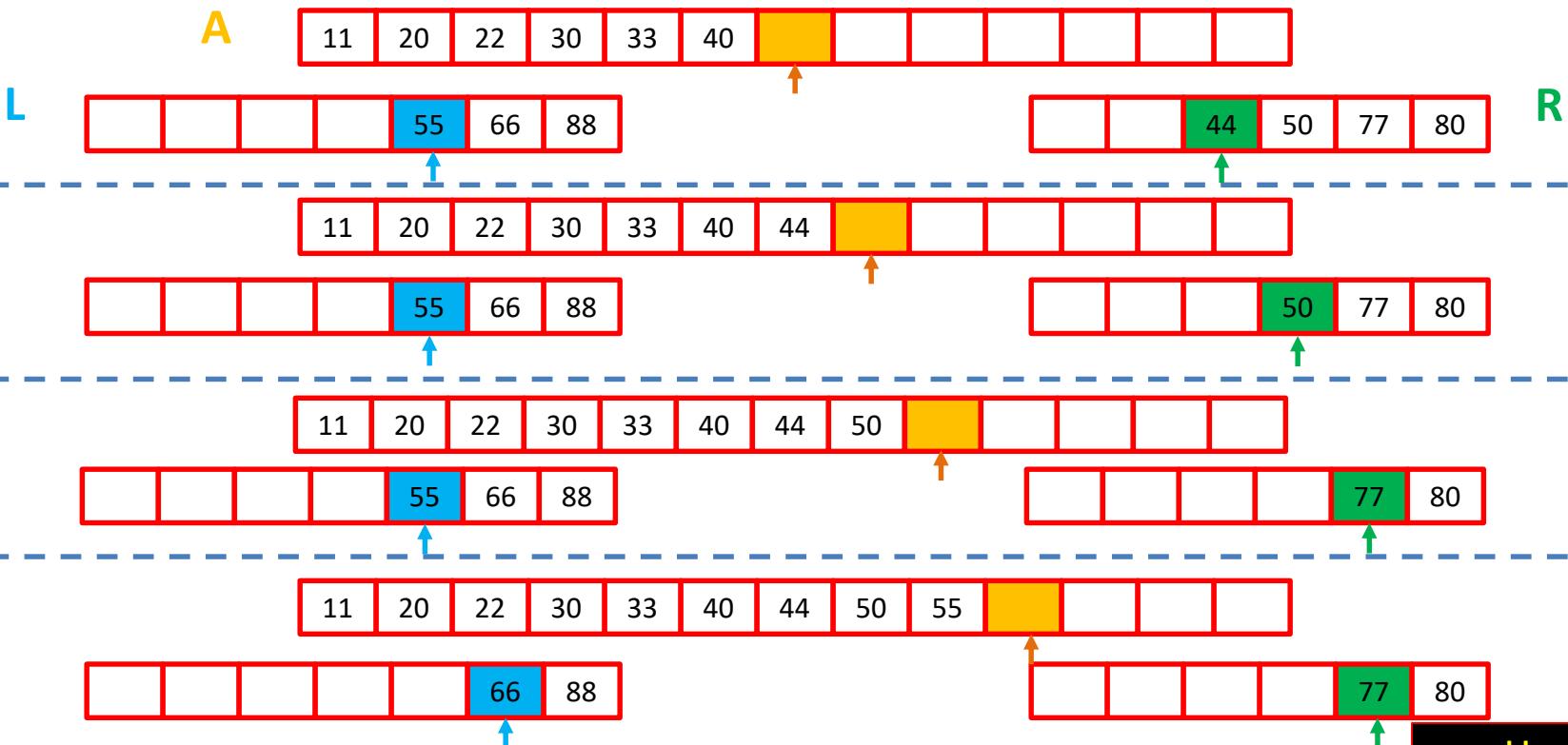


Merge Sort



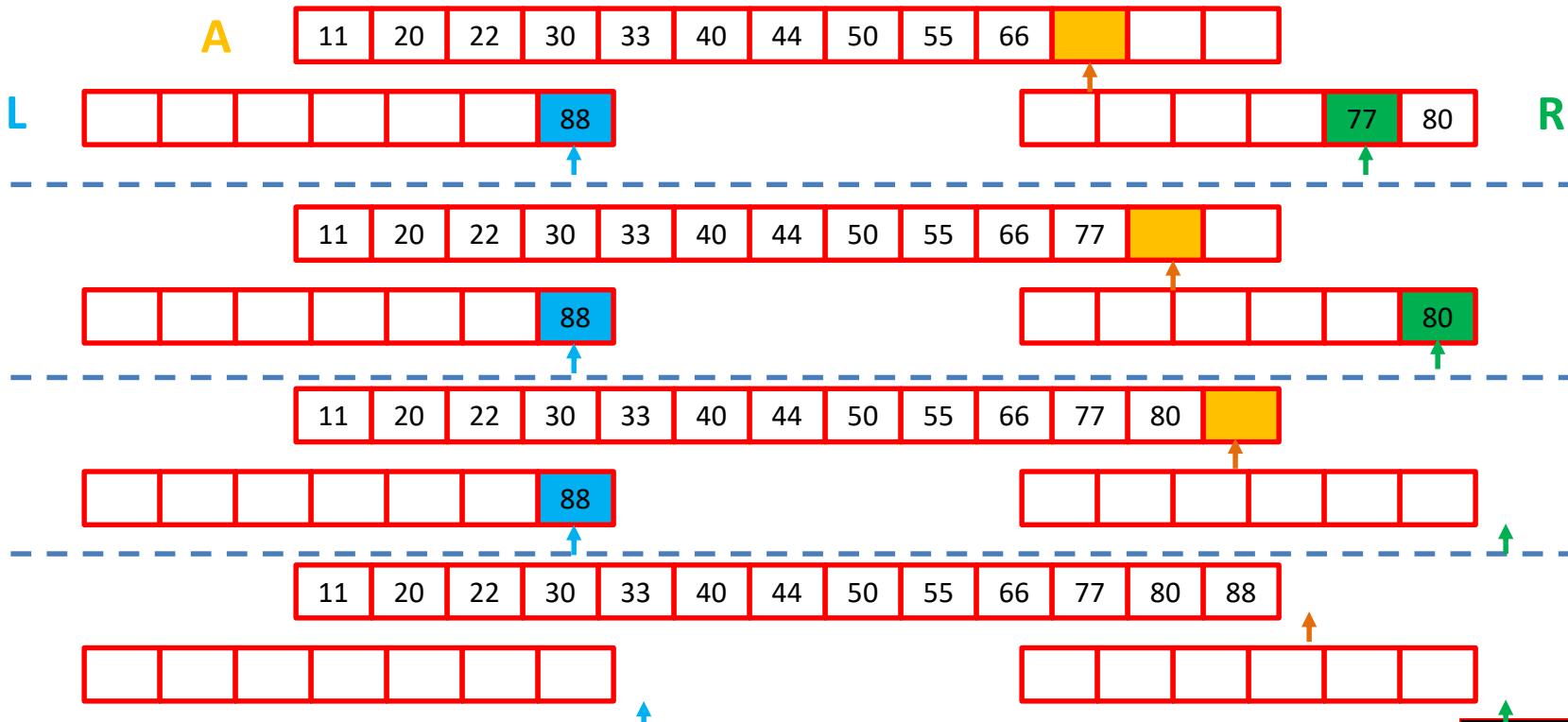


Merge Sort





Merge Sort





Merge Sort: Algorithm

```
mergeSort (A)
```

```
n <- size(A)
if n < 2 return
mid <- n / 2
Create left array L of size(mid)
Create right array R of size (n -
mid)
```

```
for i <- 0 to mid-1
    L [i] <- A[i]
```

```
for i < mid to n - 1
    R [i-mid] <- A[i]
```

```
MergeSort (L)
```

```
MergeSort (R)
```

```
Merge (L, R, A)
```



Merge Sort: Algorithm

```
merge (L, R, A)
    nL <- size(L)
    nR <- size(R)
    i <- j <- k <- 0
    while i < nL AND j < nR do
        if L[i] <= R[j]
            copy L[i] into A[k]
            increment i
        else
            copy R[j] into A[k]
            increment j
        end if
        increment k
    end while

    while i < nL do
        copy L[i] into A[k]
        increment i and k
    end while
    while j < nR do
        copy R[j] into A[k]
        increment j and k
    end while
```



Merge Sort: Program

```
void mergeSort(int A[], int left, int right)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;
        mergeSort(A, left, mid);
        mergeSort(A, mid + 1, right);
        merge(A, left, mid, right);
    }
}
```

```
int main()
{
int i, A[] = {66, 33, 40, 22, 55, 88, 11, 80,
20, 50, 44, 77, 30}, size=13;

mergeSort(A, 0, size - 1);

for(i=0; i<size; ++i)printf("%d ",A[i]);
return 0;
}
```



Merge Sort: Program

```
void merge(int A[], int left, int mid, int right)
{
    int i, j, k;
    int nL = mid - left + 1;
    int nR = right - mid;
    int L[nL], R[nR];
    for (i = 0; i < nL; i++) L[i] = A[left + i];
    for (j = 0; j < nR; j++) R[j] = A[mid + 1 + j];

    i = 0; j = 0; k = left;
    while (i < nL && j < nR)
    {
        if (L[i] <= R[j]) {A[k] = L[i]; i++;}
        else {A[k] = R[j]; j++;}
        k++;
    }
    while (i < nL) {A[k] = L[i]; i++; k++;}
    while (j < nR) {A[k] = R[j]; j++; k++;}
}
```



Thank You

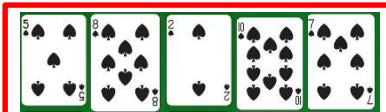
Next: Insertion Sort



Insertion Sort

- Not a best sorting algorithm
- But, better than selection and bubble sort
- Let us try to understand insertion sort using some cards

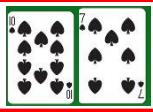
Unsorted (Left Hand)



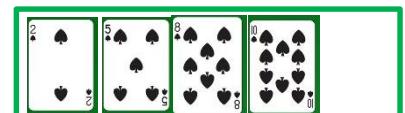
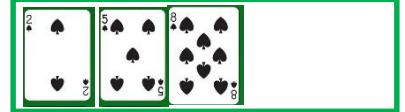
Sorted (Right Hand)



Unsorted (Left Hand)

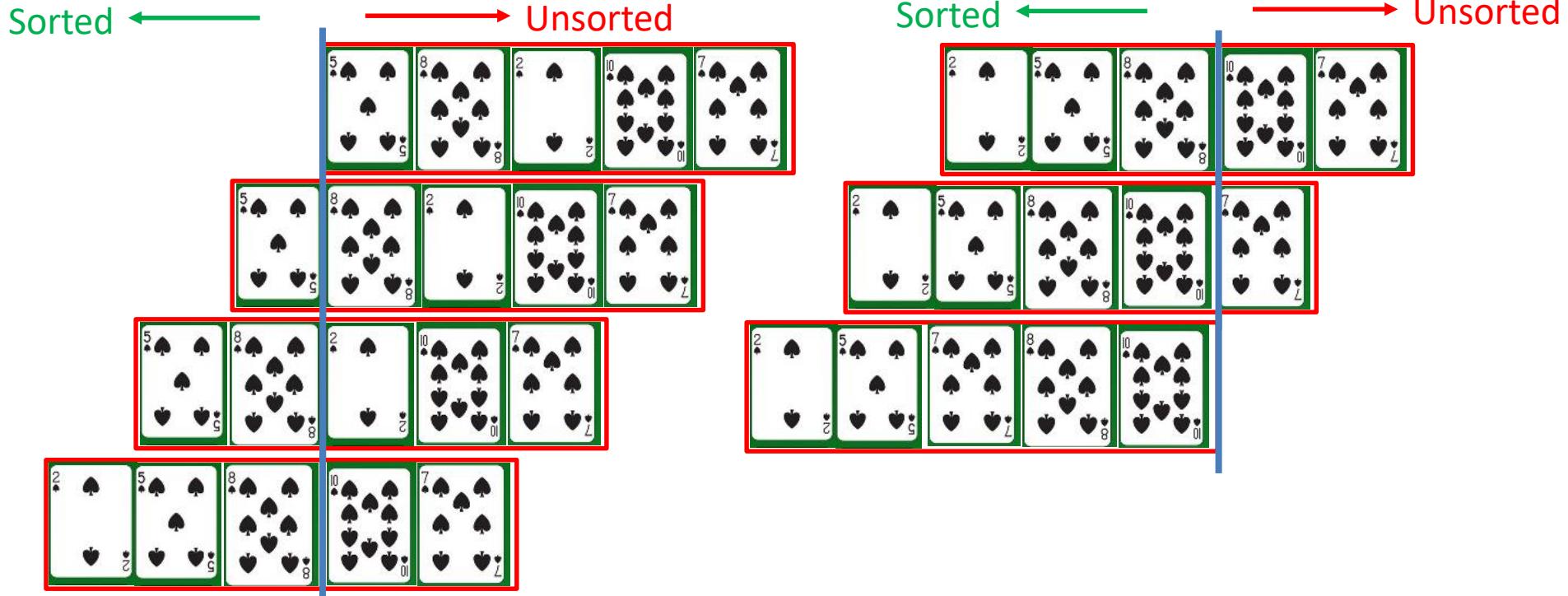


Sorted (Right Hand)





Insertion Sort





Insertion Sort

42	23	74	11	65	58
42	23	74	11	65	58
42		74	11	65	58
42		74	11	65	58
23	42	74	11	65	58
23	42	74	11	65	58
23	42		11	65	58
23	42	74	11	65	58
23	42	74		65	58
23	42	74		65	58
23	42		74	65	58
23		42	74	65	58
23		42	74	65	58

23	42	74	11	65	58
11	23	42	74		
11	23	42	74	65	58
11	23	42	74		
11	23	42		74	58
11	23	42	65	74	58
11	23	42	65		
11	23	42	65	74	
11	23	42		65	74
11	23	42	58	65	74



Insertion Sort: Psudocode

Step 1: If it is the first element, it is already sorted.

return 1;

Step 2: Pick next element

Step 3: Compare with all elements in the sorted sub-list

Step 4: Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5: Insert the value

Step 6: Repeat until list is sorted



Insertion Sort: Algorithm

```
procedure insertionSort( A : array of items )
int hole
int value
for i = 1 to size(A) do:
    /* select value to be inserted */
    value = A[i]
    hole = i

    /*locate hole position for the element to be inserted */
    while hole > 0 and A[hole-1] > value do:
        A[hole] = A[hole-1]
        hole = hole -1
    end while

    /* insert the A at hole position */
    A[hole] = value
end for
end procedure
```



Insertion Sort: Program

```
#include<stdio.h>
int main()
{int i, hole, size=6, value;
int A[25]={42,23,74,11,65,58};
for(i=1; i<size; i++)
{
    value=A[i];
    hole=i-1;
    while((value<A[hole]) && (hole>=0))
        {A[hole+1]=A[hole];
         hole=hole-1;
        }
    A[hole+1]=value;
}
printf("Order of Sorted elements: ");
for(i=0;i<size;i++) printf(" %d",A[i]);
return 0;
}
```

OUTPUT

Order of Sorted elements: 11 23 42 58 65 74

i	Value= A[i]	hole	A[hole]	Value<A[hole]	hole>=0	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
						42	23	74	11	65	58
1	23	0	42	(23<42?) True	True		42				
1		-1			False	23					
						23	42	74	11	65	58
2	74	1	42	(74<42?) False	True	23	42	74	11	65	58
3	11	2	74	(11<74?) True	True					74	
3		1	42	(11<42?) True	True				42		
3		0	23	(11<23?) True	True		23				
3		-1			False	11					
						11	23	42	74	65	58
4	65	3	74	(65<74?) True	True						74
4		2	42	(65<42) False	True					65	
						11	23	42	65	74	58
5	58	4	74	(58<75?) True	True						74
5		3	65	(58<65?) True	True						65
5		2	42	(58<42?) False	True				58		
						11	23	42	58	65	74



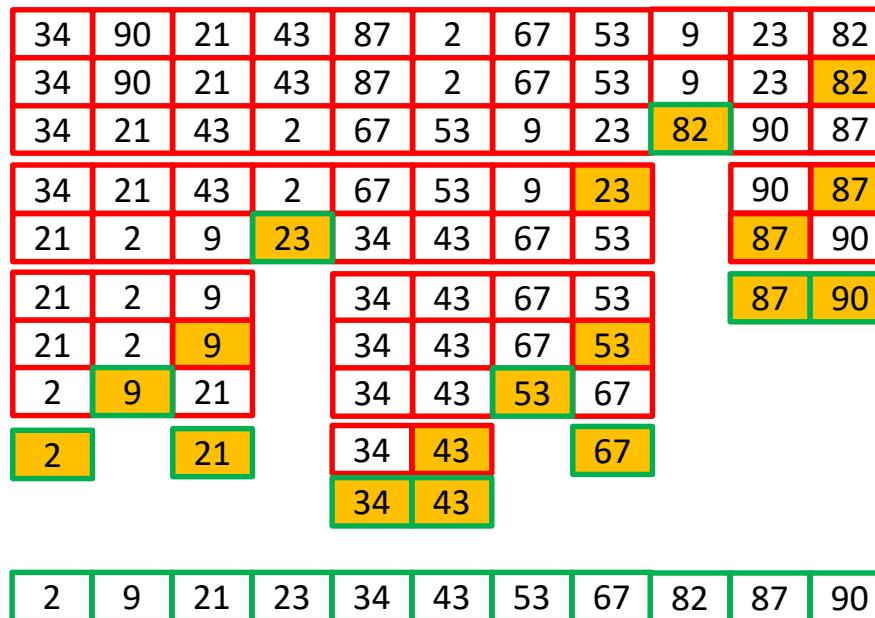
Thank You

Next: Quick Sort



Quick Sort

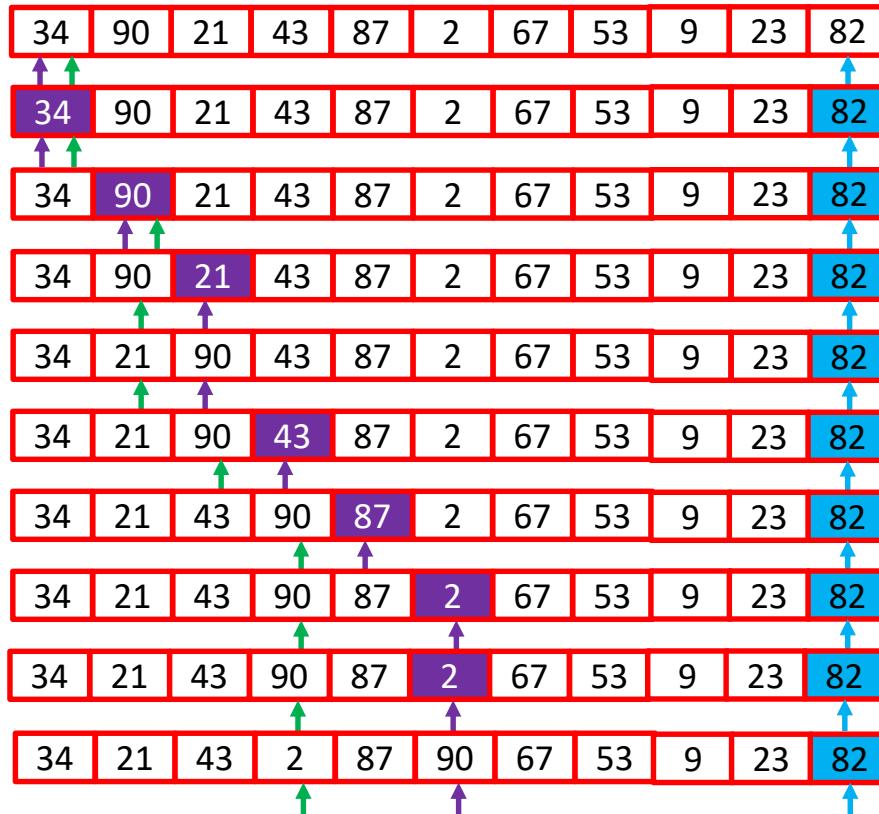
- Divide and conquer strategy, In-place sorting algorithm, Fast and efficient in practical situation



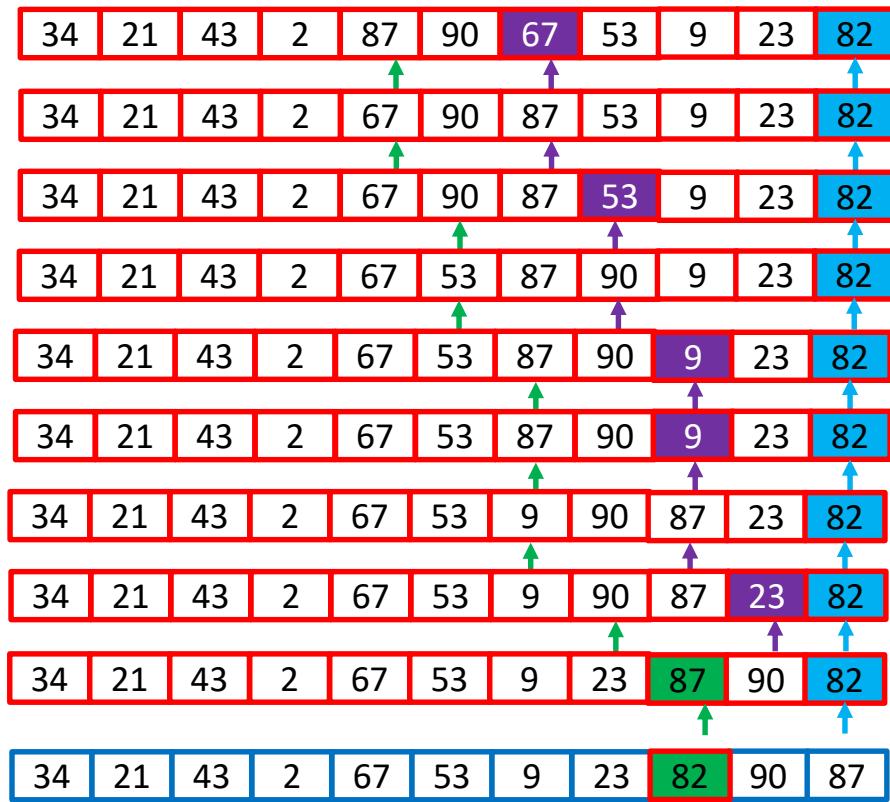


Quick Sort: Partitioning

↑ Partition Index ← Start ↑ Pivot ← A[end] ↑ i index



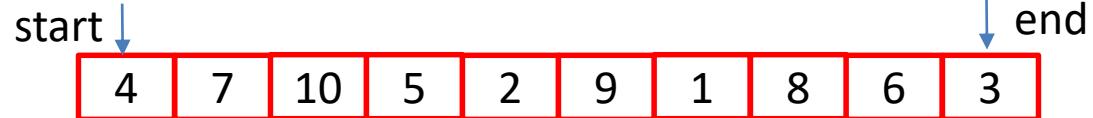
if($A[i] < \text{pivot}$) then swap $A[i]$ and $A[\text{PartitionIndex}]$





Quick Sort: Algorithm

```
QuickSort (A, start, end)
{
if (start < end)
{
    pIndex <- Partition (A,start, end)
    QuickSort (A, start, pIndex-1)
    QuickSort (A, pIndex+1, end)
}
}
```



```
Partition (A, start, end)
{
pivot <- A[end]
pIndex <- start
for i <- start to end-1 do
    if A[i] <= pivot
        swap A[i] with A[partisionIndex]
        increment pIndex
    end if
end for
}
```



Quick Sort: Program

```
int QuickSort (int *A, int start, int end)
{if(start < end)
    {int pIndex = Partition (A, start, end);
    QuickSort (A, start, pIndex-1);
    QuickSort (A, pIndex+1, end);
    }
    return 0;
}

int Partition (int *A, int start, int end)
{int i, temp, pivot = A[end], pIndex = start;
for (i=start; i<end; ++i)
    {if(A[i]<pivot)
        {temp = A[i]; A[i] = A[pIndex];A[pIndex] = temp;
        pIndex++;
        }
    }
temp = A[end]; A[end] = A[pIndex]; A[pIndex] = temp;
return pIndex;
}

int main()
{
int A[] = {5,7,10,5,2,9,1,8,6,3}, i;
QuickSort (A, 0, 9);
for(i=0; i<10; ++i) printf("%d ",A[i]);
return 0;
}
```



Thank You

Next: Heap Sort

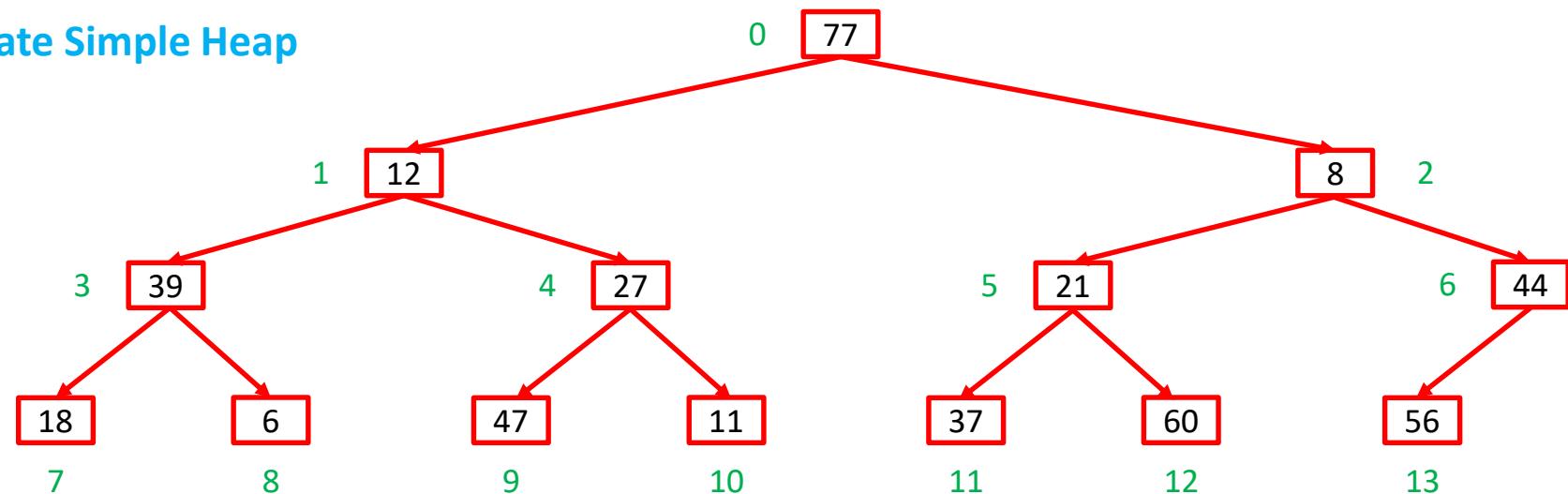


Heap Sort

77	12	8	39	27	21	44	18	6	47	11	37	60	56
0	1	2	3	4	5	6	7	8	9	10	11	12	13

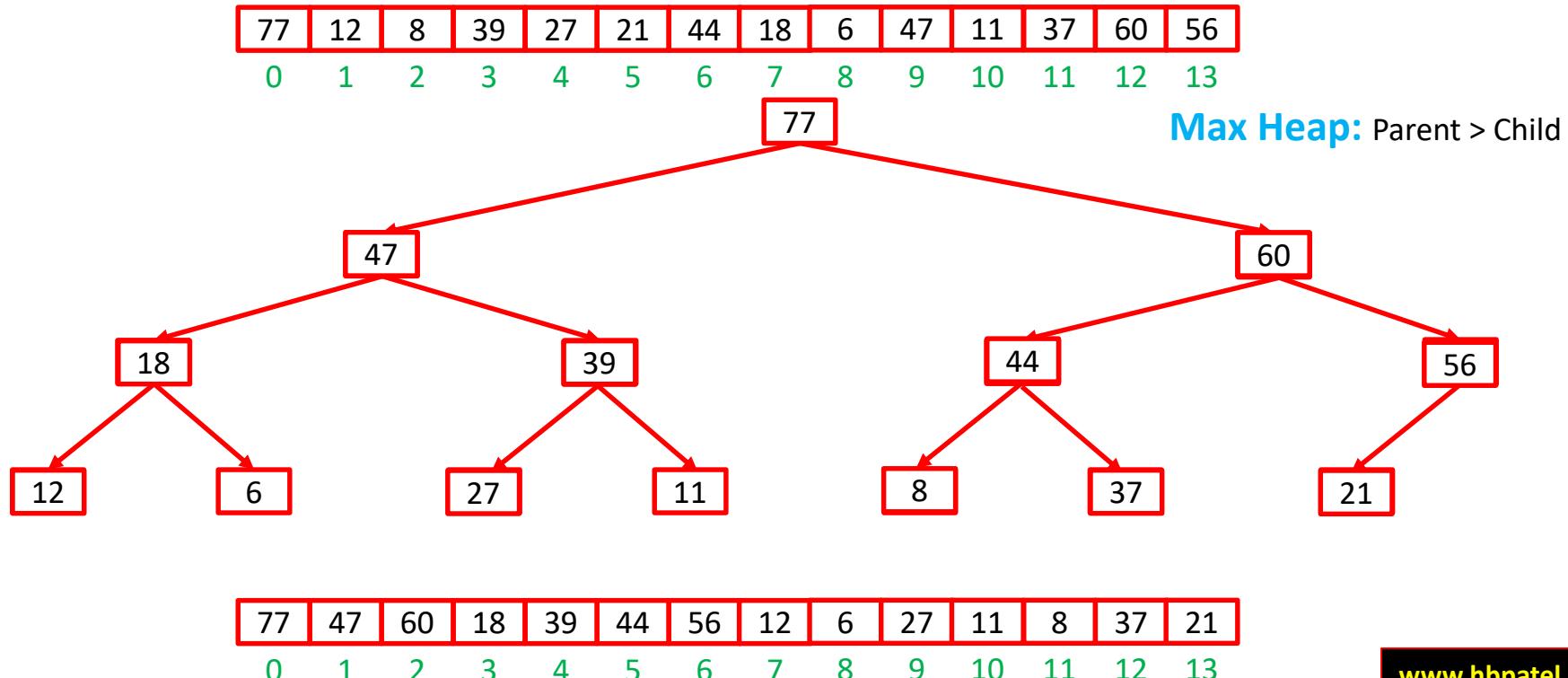
Heap: Ordered binary tree

Create Simple Heap



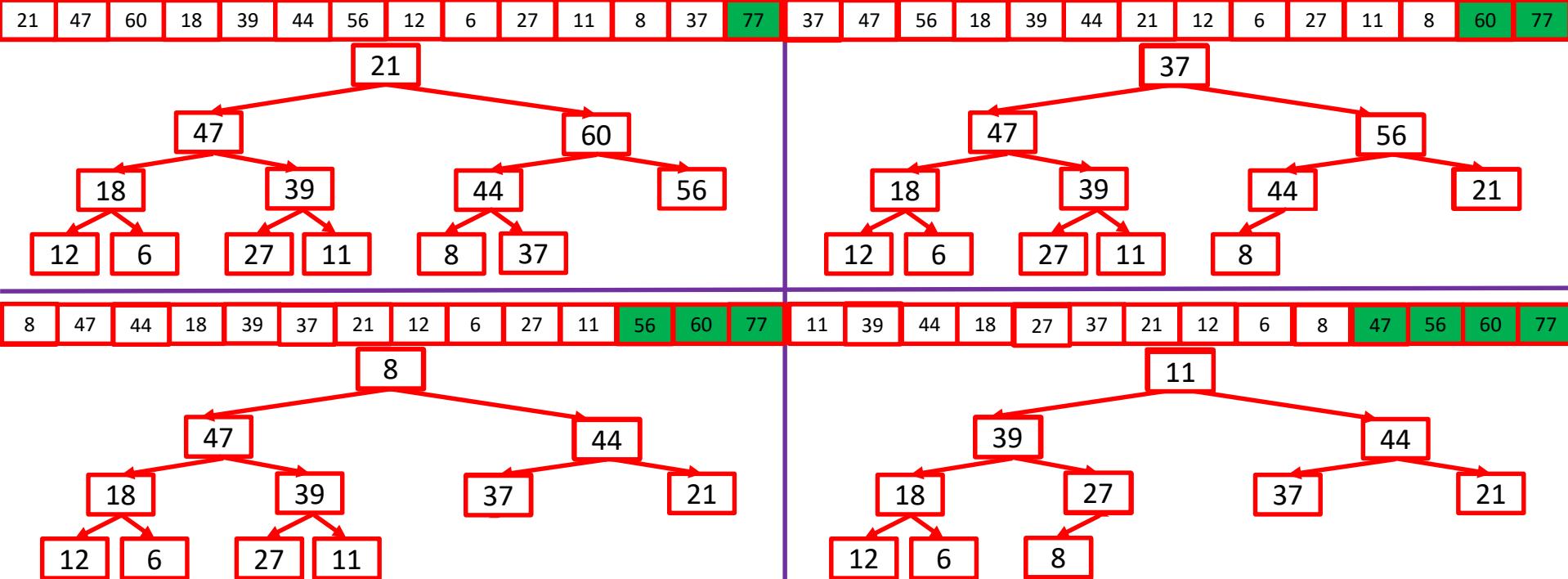


Create Max Heap



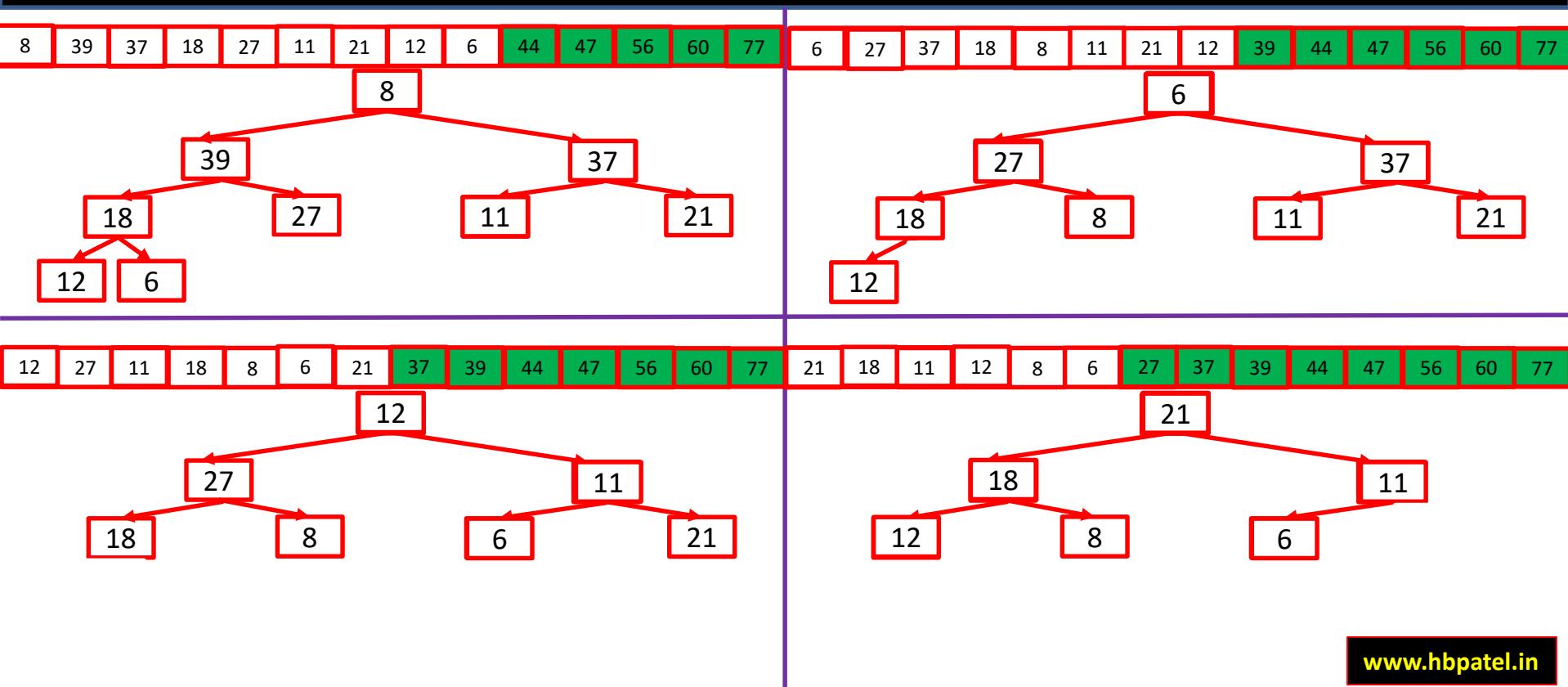
Heapify

- parent > child
- Swap maximum element (root) with the last element
- Remove the maximum element



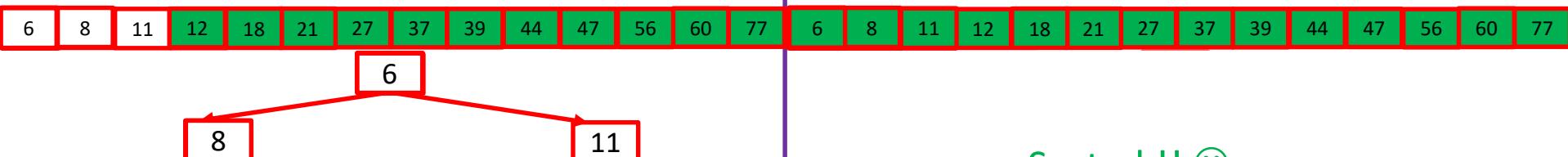
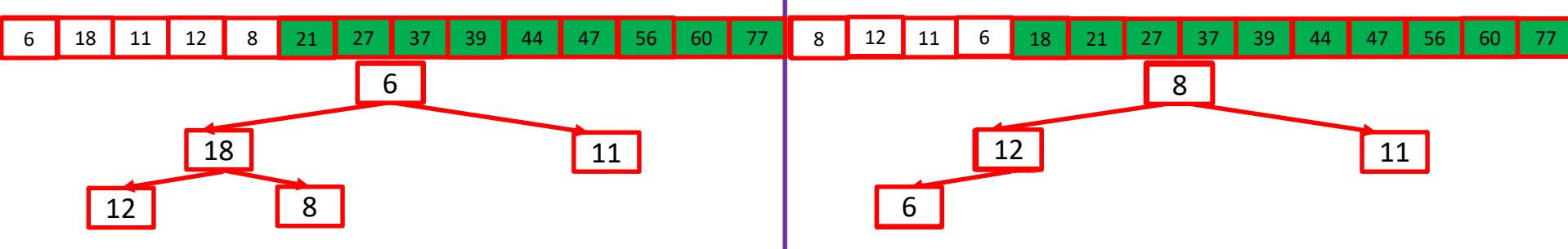
Heapify

- parent > child
- Swap maximum element (root) with the last element
- Remove the maximum element



Heapify

- parent > child
- Swap maximum element (root) with the last element
- Remove the maximum element



Sorted !! 😊



Linear Search

11	58	6	77	18	21	39	37	27	44	47	8	60	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Does exist in array? Result? At which position?

21	Yes	5
41	No	-
60	Yes	12
88	No	-

How do we search? Simplest option is
Linear Search !

searchLinear

Best case: Find the x at first place, hence only 1 comparison

Worst case: Find the x at last place (or x does not exist), hence n comparisons

Array does not need to be sorted

Linear Search: Algorithm & Program

```
#include<stdio.h>

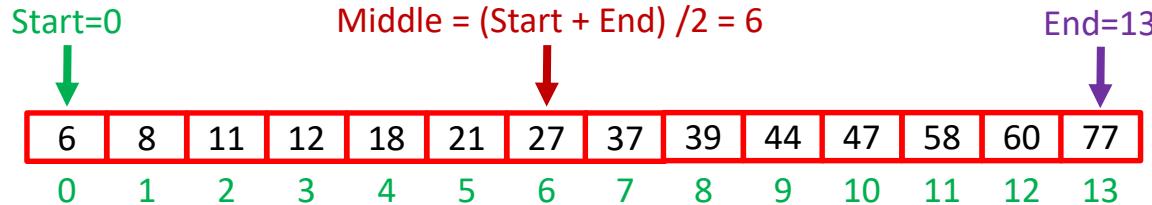
int main()
{
int A[]={77,12,8,39,27,21,44,18,6,47,11,37,60,56}
int n=14, x=39;
int searchLinear (int[],int,int);
int searchResult = searchLinear (A,n,x);
if (searchResult == -1)printf("Element does not exist");
else printf("Element %d found at position %d",x,searchResult);
return 0;
}

int searchLinear (int A[], int n, int x)
{
int i;
for(i=0; i<n; ++i){if (A[i]==x) return i;}
return -1;
}
```

```
searchLinear (A, n, x)
for each i from 0 to n-1 do
    if A[i] is equal to x
        then return i
    end if
end for
return -1
```



Binary Search



Let us try to find out whether the number $x=39$ exists in the array or not !

We compare x with $A[Middle]$ and there are three possibilities

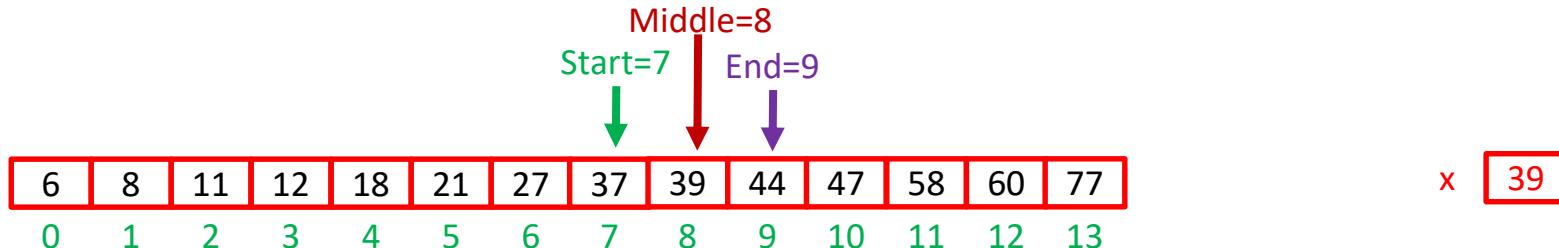
Possibility 1: x is equal to $A[Middle]$. Our search is over as we have found the element.

Possibility 2: x is less than $A[Middle]$. We need to search between $Start$ and $Middle-1$. That means End becomes $Middle-1$.

Possibility 3: x is greater than $A[Middle]$. We need to search between $Middle+1$ to End . That means $Start$ becomes $Middle+1$.



Binary Search



X=39 A[Middle]=27 Compare x with A[Middle] Start=0 End=13 Middle = 6

39 27 39 > 27 7 13 10

39 47 39 < 47 7 9 8

39 39 39 = 39 FOUND ! ☺



Binary Search: Algorithm

```
searchBinary (A, n, x)
start <- 0
end <- n-1
while start <= end do
    middle <- (start + end) / 2
    if A[Middle] is equal to x then
        return Middle
    else if A[Middle] is greater than x then
        end <- Middle - 1
    else
        start <- Middle + 1
    end if
end while
return -1
```



Binary Search: Program

```
/* binary search */
#include<stdio.h>
int main()
{
    int A[]={6,8,11,12,18,21,27,37,39,44,47,58,60,77}, n=14, x=39;
    int searchBinary (int[],int,int);
    int searchResult = searchBinary (A,n,x);
    if (searchResult == -1)printf("Element does not exist");
    else printf("Element %d found at position %d",x,searchResult);
    return 0;
}
int searchBinary (int A[], int n, int x)
{
    int i, start=0, end=n-1, middle;
    while (start <= end)
    {
        middle = (start + end)/2;
        if (A[middle]==x) return middle;
        else if (A[middle]>x) end=middle-1;
        else start=middle+1;
    }
    return -1;
}
```

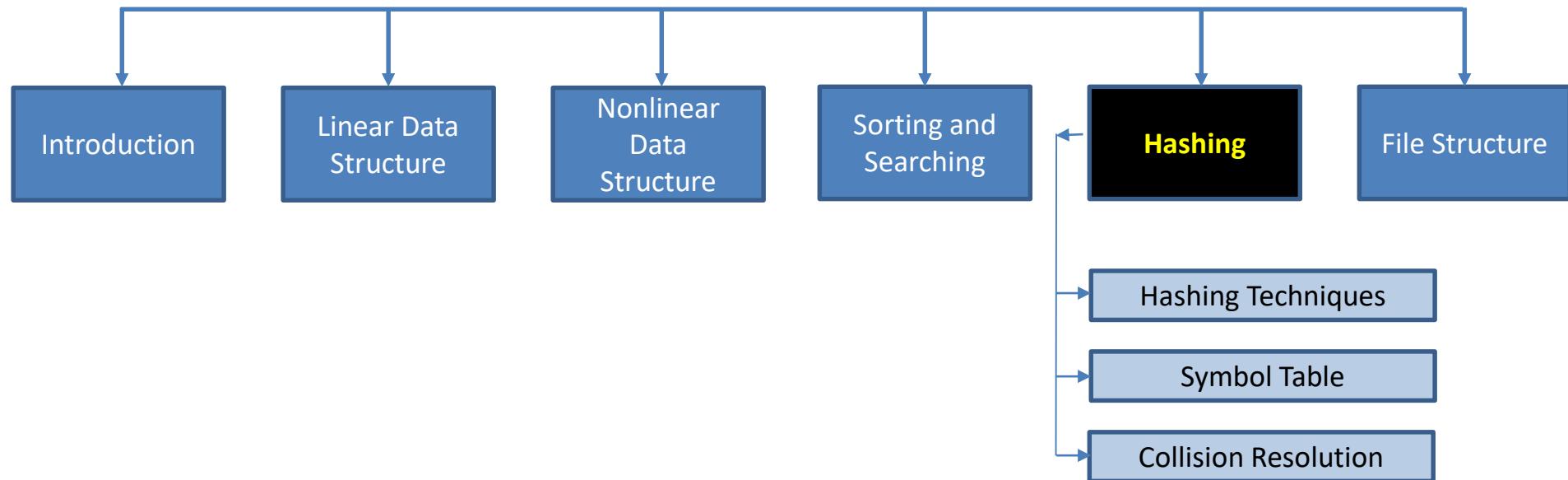


Thank You

Next: Hashing



DSA: Hashing





Hashing

Hashing is a technique that is used to uniquely identify/search/retrieve a specific object from a group of similar objects.

For example, using Enrolment number of a student, one can fetch other information about him/her.

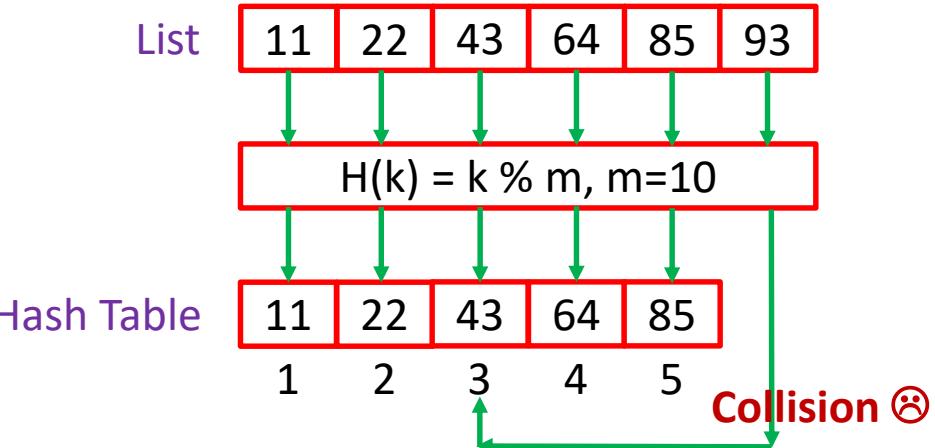
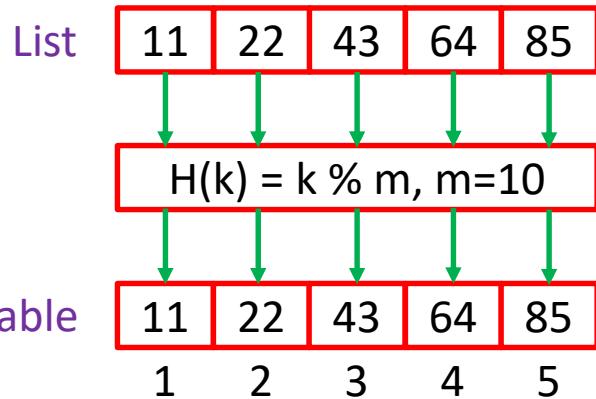
As an another example, using accession number on a book, one can find the exact location of the book in library.



Hashing

Let $H(k)$ be a hash function that maps the value V at the index $k \% m$, where k is the key and m is the size of hash table.

For example, we have a list $L = \{11, 22, 43, 64, 85\}$, it will be stored at position $\{1,2,3,4,5\}$ in an array or hash table, respectively.



Collision Resolution

- (1) Open Hashing (Closed Addressing)
- (2) Closed Hashing (Open Addressing)

- (1) Open Hashing (Closed Addressing)

✓ Chaining

- (2) Closed Hashing (Open Addressing)

- Linear Probing
- Quadratic Probing
- Double Hashing

Collision can not be avoided but can be minimized.



Collision Resolution: Chaining

(1) Open Hashing (Closed Addressing) - Chaining

List $L(k) = \{3, 2, 9, 6, 11, 13, 7, 12\}$

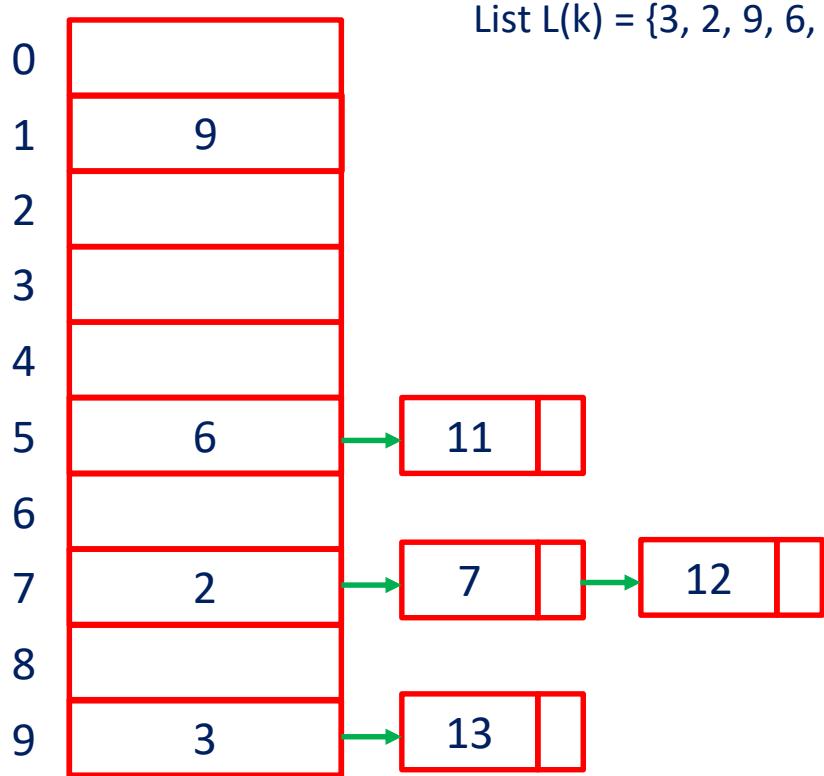
Hash Function $H(k) = 2k + 3$

Size of hash table = $m = 10$

Collision Resolution Technique to be used = Division Method
AND Open Hashing (Closed Addressing) - Chaining



Collision Resolution: Chaining



k	Location= $(2k+3)\%m$
3	$(2 \times 3 + 3) \% 10 \rightarrow 9$
2	$(2 \times 2 + 3) \% 10 \rightarrow 7$
9	$(2 \times 9 + 3) \% 10 \rightarrow 1$
6	$(2 \times 6 + 3) \% 10 \rightarrow 5$
11	$(2 \times 11 + 3) \% 10 \rightarrow 5$
13	$(2 \times 13 + 3) \% 10 \rightarrow 9$
7	$(2 \times 7 + 3) \% 10 \rightarrow 7$
12	$(2 \times 12 + 3) \% 10 \rightarrow 7$

Collision Resolution: Chaining

```
#include <stdio.h>
#define m 10
#define size 20
struct hashing
{
    int data;
    struct hashing *next;
};
void main()
{
    struct hashing *hash[m], *newnode, *temp;
    int list[size]={23,54,10,55,67,32,44,77,2,9,53,94,30,15,167,352,434,767,27,98}, i, pos;

    for(i=0; i<m; ++i) hash[i]=NULL;
```



Collision Resolution: Chaining

```
for (i=0; i<size; ++i)
{
    pos = (2 * list[i] + 3) % m;
    if(hash[pos]==NULL)
    {
        newnode = (struct hashing *) malloc (sizeof(newnode));
        newnode->data = list[i];
        newnode->next = NULL;
        hash[pos]=newnode;
    }
    else
    {---}
}
```



Collision Resolution: Chaining

```
for (i=0; i<size; ++i)
{
    pos = (2 * list[i] + 3) % m;
    if(hash[pos]==NULL)           {--}
    else
    {
        temp=hash[pos];
        while(temp->next!=NULL) {temp=temp->next;}
        newnode = (struct hashing *) malloc (sizeof(newnode));
        newnode->data = list[i];
        newnode->next = NULL;
        temp->next=newnode;
    }
}
```



Collision Resolution: Chaining

```
for(i=0; i<m; ++i)
{
    temp=hash[i];
    printf("Hash Index : %d ->", i);
    while(temp!=NULL)
    {
        printf("%d\t",temp->data);
        temp=temp->next;
    }
    printf("\n");
}
```



Collision Resolution: Chaining

Hash Index : 0 ->								
Hash Index : 1 ->	54	44	9	94	434			
Hash Index : 2 ->								
Hash Index : 3 ->	10	55	30	15				
Hash Index : 4 ->								
Hash Index : 5 ->								
Hash Index : 6 ->								
Hash Index : 7 ->	67	32	77	2	167	352	767	27
Hash Index : 8 ->								
Hash Index : 9 ->	23	53	98					





Collision Resolution

- (1) Open Hashing (Closed Addressing)
- (2) Closed Hashing (Open Addressing)

- (1) Open Hashing (Closed Addressing)

- Chaining

- (2) Closed Hashing (Open Addressing)

- Linear Probing

- Quadratic Probing

- Double Hashing

Collision Resolution: Linear Probing

(2) Closed Hashing (Open Addressing) – Linear Probing

List $L(k) = \{3, 2, 9, 6, 11, 13, 7, 12\}$

Hash Function $H(k) = 2k + 3$

Size of hash table = $m = 10$

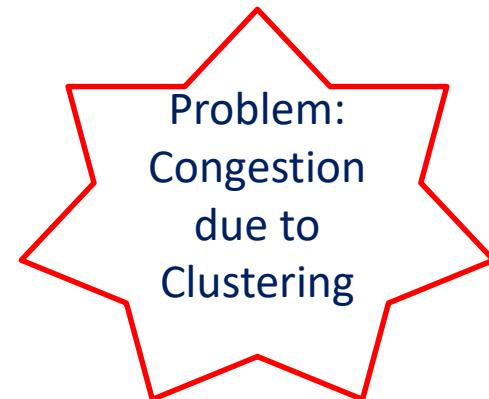
Collision Resolution Technique to be used = Division Method
AND Closed Hashing (Open Addressing) – Linear Probing



Collision Resolution: Linear Probing

Insert K_i at first free location from $(loc + i) \% m$ where $i=0$ to $m-1$

0	13
1	9
2	12
3	
4	
5	6
6	11
7	2
8	7
9	3



k	Location= $(2k+3)\%m$	Probes
3	$(2 \times 3 + 3) \% 10 \rightarrow 9$	1
2	$(2 \times 2 + 3) \% 10 \rightarrow 7$	1
9	$(2 \times 9 + 3) \% 10 \rightarrow 1$	1
6	$(2 \times 6 + 3) \% 10 \rightarrow 5$	1
11	$(2 \times 11 + 3) \% 10 \rightarrow 5$	2
13	$(2 \times 13 + 3) \% 10 \rightarrow 9$	2
7	$(2 \times 7 + 3) \% 10 \rightarrow 7$	2
12	$(2 \times 12 + 3) \% 10 \rightarrow 7$	6

Order of Elements: 13, 9, 12, -, -, 6, 11, 2, 7, 3



Collision Resolution

- (1) Open Hashing (Closed Addressing)
- (2) Closed Hashing (Open Addressing)

- (1) Open Hashing (Closed Addressing)

- Chaining

- (2) Closed Hashing (Open Addressing)

- Linear Probing

-  Quadratic Probing

- Double Hashing

Collision Resolution: Quadratic Probing

(3) Closed Hashing (Open Addressing)

List $L(k) = \{3, 2, 9, 6, 11, 13, 7, 12\}$

Hash Function $H(k) = 2k + 3$

Size of hash table = $m = 10$

Collision Resolution Technique to be used = Division Method
AND Closed Hashing (Open Addressing)- Quadratic Probing



Collision Resolution: Quadratic Probing

Insert K_i at first free location from $(loc + i^2) \% m$ where $i=0$ to $m-1$

0	13
1	9
2	
3	12
4	
5	6
6	11
7	2
8	7
9	3

k	Location= $(2k+3)\%m$	Probes
3	$(2 \times 3 + 3)\%10 \rightarrow 9$	1
2	$(2 \times 2 + 3)\%10 \rightarrow 7$	1
9	$(2 \times 9 + 3)\%10 \rightarrow 1$	1
6	$(2 \times 6 + 3)\%10 \rightarrow 5$	1
11	$(2 \times 11 + 3)\%10 \rightarrow 5$	2
13	$(2 \times 13 + 3)\%10 \rightarrow 9$	2
7	$(2 \times 7 + 3)\%10 \rightarrow 7$	2
12	$(2 \times 12 + 3)\%10 \rightarrow 7$	5

Order of Elements: 13, 9, -, 12, -, 6, 11, 2, 7, 3





Collision Resolution

- (1) Open Hashing (Closed Addressing)
- (2) Closed Hashing (Open Addressing)

- (1) Open Hashing (Closed Addressing)

- Chaining

- (2) Closed Hashing (Open Addressing)

- Linear Probing

- Quadratic Probing

- ✓ Double Hashing

Collision Resolution: Double Hashing

(4) Closed Hashing (Open Addressing)

List $L(k) = \{3, 2, 9, 6, 11, 13, 7, 12\}$

Hash Function $H(k) = 2k + 3$

Size of hash table = $m = 10$

Collision Resolution Technique to be used = Division Method
AND Closed Hashing (Open Addressing)- Double Hashing



Collision Resolution: Double Hashing

$$H_1(k) = 2k + 3, H_2(k) = 3k + 1$$

Insert K_i at first free place from $(loc1 + loc2 \times i) \% m$, where $i=\{0 \text{ to } m-1\}$

0	
1	9
2	
3	11
4	12
5	6
6	
7	2
8	
9	3

k	Loc1=(2k+3)%m	Loc2=(3k+1)%m	Probes
3	$(2 \times 3 + 3) \% 10 \rightarrow 9$	-	1
2	$(2 \times 2 + 3) \% 10 \rightarrow 7$	-	1
9	$(2 \times 9 + 3) \% 10 \rightarrow 1$	-	1
6	$(2 \times 6 + 3) \% 10 \rightarrow 5$	-	1
11	$(2 \times 11 + 3) \% 10 \rightarrow 5$	$(3 \times 11 + 1) \% 10 \rightarrow 4$	3
13	$(2 \times 13 + 3) \% 10 \rightarrow 9$	$(3 \times 13 + 1) \% 10 \rightarrow 0$	m^*
7	$(2 \times 7 + 3) \% 10 \rightarrow 7$	$(3 \times 7 + 1) \% 10 \rightarrow 2$	m^*
12	$(2 \times 12 + 3) \% 10 \rightarrow 7$	$(3 \times 12 + 1) \% 10 \rightarrow 7$	2

Order of Elements: -9, -11, 12, 6, -2, -, 3

* Poor hash function



Symbol Table

```
int a;
int fun1()
{
int b, c;
{
    int d, e;
}
int f, g;
{
    int h, i;
}
}
void fun2()
{
int j, k;
{
    int l, m;
}
int n;
}
```

Symbol Table (Global)

Name	Type	Data type
a	var	int
fun1	fun	int
fun2	fun	void

Symbol Table (fun1)

Name	Type	Data type
b	var	int
c	var	int
f	var	int
g	var	int

Symbol Table (fun2)

Name	Type	Data type
i	var	int
k	var	int
n	var	int

Symbol Table (Inner Scope - I)

Name	Type	Data type
d	var	int
e	var	int

Symbol Table (Inner Scope - II)

Name	Type	Data type
h	var	int
i	var	int

Symbol Table (Inner Scope - III)

Name	Type	Data type
l	var	int
m	var	int



Symbol Table

It is an important **data structure** created and maintained by the compiler in order to keep track of semantics of variable i.e. it stores information about scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

Items stored in Symbol table:

- Variable names and constants
- Procedure and function names
- Literal constants and strings
- Compiler generated temporaries
- Labels in source languages

Information used by compiler from Symbol table:

- Data type and name
- Declaring procedures
- Offset in storage
- If structure or record then, pointer to structure table.
- For parameters, whether parameter passing by value or by reference
- Number and type of arguments passed to function
- Base Address



Symbol Table

Following are commonly used data structure for implementing symbol table :

- List / Arrays
- Linked List
- Hash Table
- Binary Search Tree

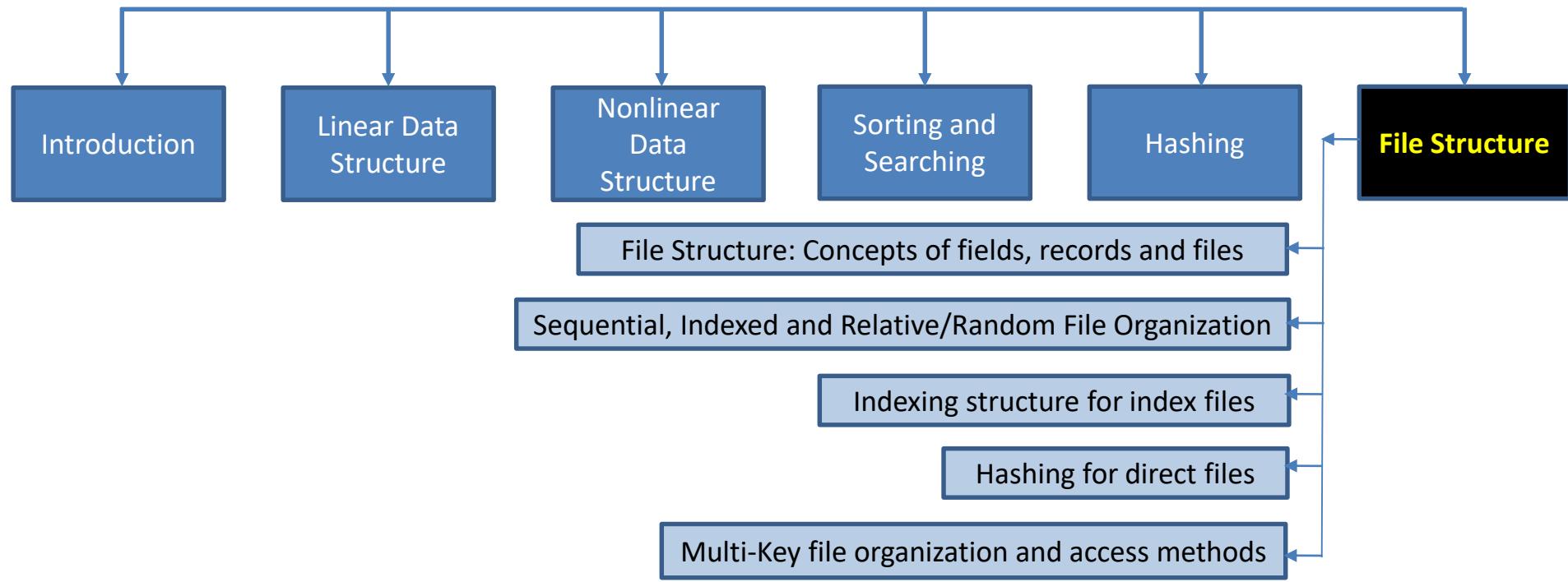


Thank You

Next: File Structures



DSA: File Structure



File Structures



File organization ensures that records are available for processing. It is used to determine an efficient file organization for each base relation.

For example, if we want to retrieve students records in alphabetical order of name. Sorting the file by student name is a good option for file organization. However, if we want to retrieve all students whose CPI are in a certain range, a file is ordered by student name would not be a good file organization.

File organization indicates how the records are organized in a file.



File Structures

Basic Terminology

- A file is a collection of data.
- The data is subdivided into records
- Each record contains a number of fields
- One (or more) field is the key field

Database: collection of related files

File: collection of related records

Record: collection of related data fields

Key Fields: It uniquely identifies the record

Enrolment No	Name	CPI	Address
19CE201	Jaivik	8.9	Kadi
19IT301	Yash	7.8	Gandhinagar
19CSE401	Priyam	8.7	Ahmedabad

Storing the files in certain order is called file Organization

File Structures



Issue: How to organize the records such that it provides convenient access to us?

Various methods have been introduced to Organize files.

- Sequential File Organization
- Direct Access File Organization
- Indexed File Organization
- Relative File Organization
- Random File Organization

Access Methods:

- Sequential
 - Sequential file
- Random
 - Direct Access file
 - Indexed file
 - Hashed file

Sequential File Organization



Records are conceptually organized in a sequential list and can only be accessed sequentially

- The actual storage may/may-not be sequential. For example, on a tape, mostly it is sequential, but on hard disk, it may be distributed across sectors/track
- Suitable for applications that requires sequential processing of the file
- Originally designed to operate for magnetic tapes
- Records can only be accesses sequentially, one after another, from beginning to end
- It is not possible to add a record in the middle of the file without rewriting the file.



Image Source: archive.indianexpress.com



Image Source: www.123rf.com

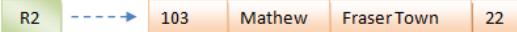
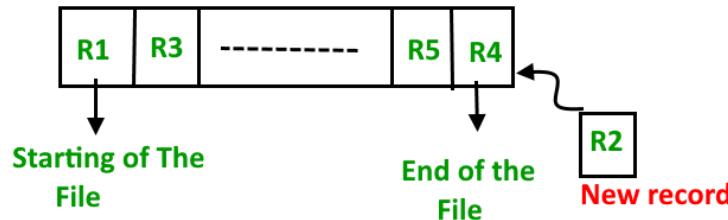


Sequential File Organization

1. Pile File Method



Image source: www.geeksforgeeks.org



2. Sorted File Method

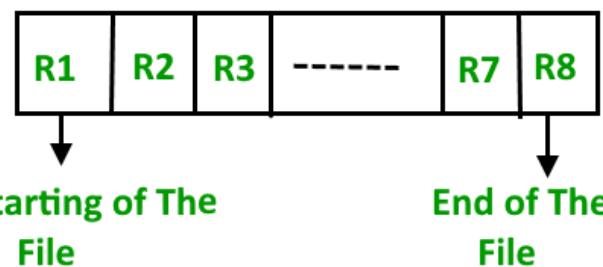
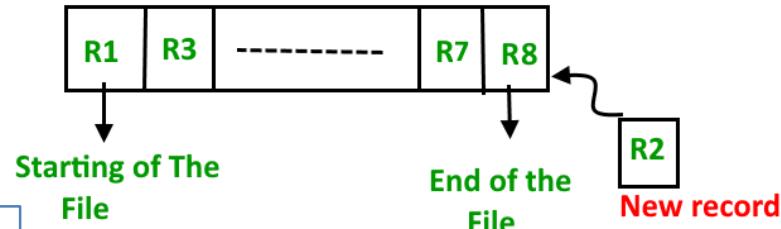


Image source: www.tutorialcup.com

Sequential File Organization



Advantage:

- Simple design. Easy to store the data.
- Fast and efficient for large data for similar type of tasks. (E.g. Generating marksheets, Salary slips etc.)
- Good for report generation or statistical calculations.
- E.g. magnetic tapes

Disadvantage:

- Sorting/Arranging records is cumbersome (time & space consumption).
- Becomes slow as it tries to sort the data after every insert/delete/update.
- Accessing a random/specific data

Direct Access File Organization



- Direct access file is also known as random access or relative file organization.
- In direct access file, all records are stored in direct access storage device (DASD), such as hard disk. The records are randomly placed throughout the file.
- The records does not need to be in sequence because they are updated directly and rewritten back in the same location.
- This file organization is useful for immediate access to large amount of information. It is used in accessing large databases.
- It is also called as hashing.

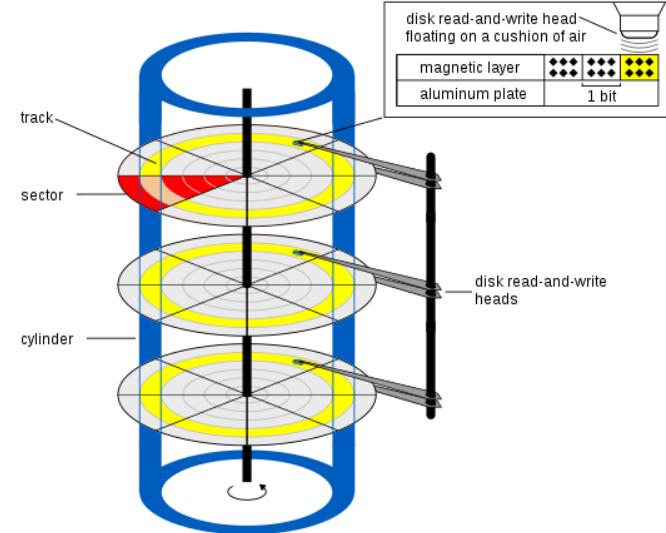


Image Source: www.wikipedia.org

Direct Access File Organization



Advantages:

- Direct access file helps in online transaction processing system (OLTP) like online railway reservation system.
- In direct access file, sorting of the records are not required.
- It accesses the desired records immediately.
- It updates several files quickly.
- It has better control over record allocation.

Disadvantages:

- It is expensive.



Indexed Sequential File Organization

- Indexed sequential access file combines both **sequential** file and **direct access** file organization.
- It consists of two parts:
Data File contains records in sequential scheme.
Index File contains the primary key and its address in the data file.

Block No	Name	Maintenance	Mobile
A-101	--	--	--
--	--	--	--
--	--	--	--
--	--	--	--
--	--	--	--
D-101	--	--	--
D-102	--	--	--
--	--	--	--
--	--	--	--
--	--	--	--
--	--	--	--
--	--	--	--
--	--	--	--
--	--	--	--
G-101	--	--	--
G-102	--	--	--
--	--	--	--

Indexed Sequential File Organization



- In indexed sequential access file, records are stored randomly on a direct access device such as magnetic disk by a primary key.
- This file have multiple keys. These keys can be alphanumeric in which the records are ordered is called primary key.
- The data can be access either sequentially or randomly using the index. The index is stored in a file and read into memory when the file is opened.

Indexed Sequential File Organization



Advantage:

- In indexed sequential access file, sequential file and random file access is possible.
- It accesses the records very fast if the index table is properly organized.
- The records can be inserted in the middle of the file.
- It provides quick access for sequential and direct processing.
- It reduces the degree of the sequential search.

Disadvantage:

- Indexed sequential access file requires unique keys and periodic reorganization.
- Indexed sequential access file takes longer time to search the index for the data access or retrieval.
- It requires more storage space.
- It is expensive because it requires special software.
- It is less efficient in the use of storage space as compared to other file organizations.

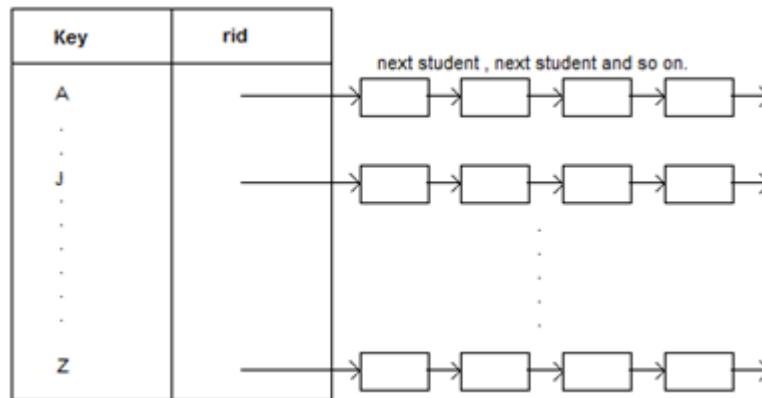
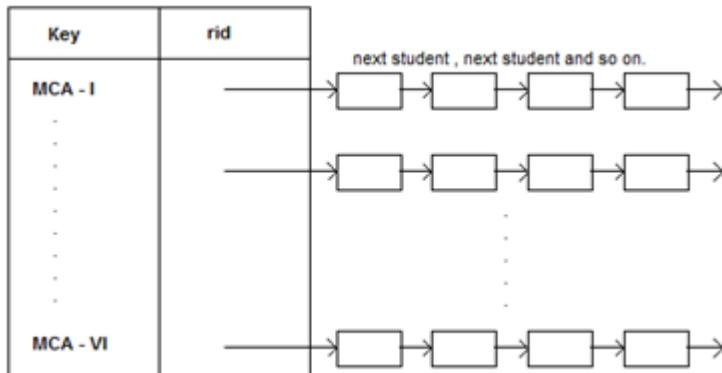


Multi-Key File Organization

Why?

- When you need to access a file by multiple keys
- For example, I wish to access the student data using (i) enrolment number (ii) name (iii) city etc

Example





Thank You