RECURSION

- A recursive function is a function that calls itself until a particular condition (termination) is met.

— Applications

(i) Product a*b

$a*b = a$ if $b == 1$
$a*b = a*(b-1) + a$ if $b > 1$

$5 \times 4 =$ $5 \times 3 + 5 \rightarrow 20$
$5 \times 2 + 5 \rightarrow 15$
$5 \times 1 + 5 \rightarrow 10$
5

(ii) Factorial

$5! = 5 \times 4! = 120$
$= 4 \times 3! = 24$
$= 3 \times 2! = 6$
$= 2 \times 1! = 2$
$= 1 * 0! = 1$

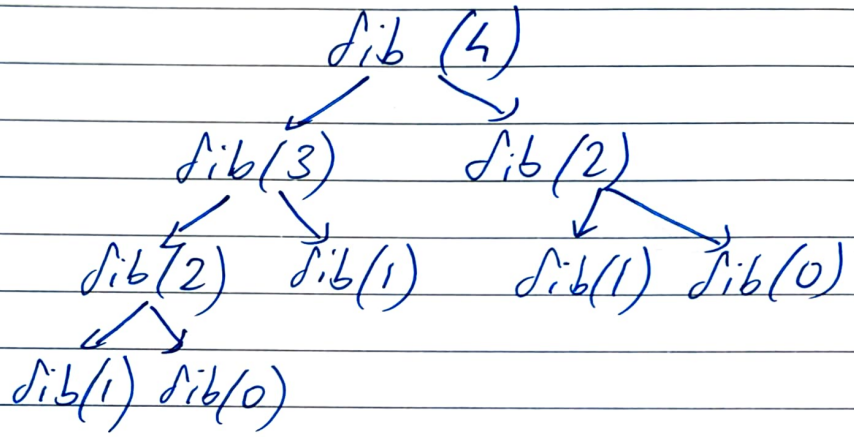
$n! = 1$ if $n == 0$
$n! = n(n-1)(n-2)\ldots.1$ if $n > 0$

__/__/____

(iii) Fibonacci sequence

$fib(n) = n$ if $n=0$ or $n=1$

$fib(n) = fib(n-2) + fib(n-1)$ for $n >= 2$



(iv) Binary Search

- Element is compared with middle element in array
- If element < middle element, first half is to be searched
- If element > middle element, second half is to be searched.

— Properties

- Recursive algorithm should terminate at some point, otherwise it will not end.

__/__/____

☆ Enumerated Types (enum)

- User-defined type based on standard integer type, where each integer value is given an identifier called enumerated constant.

enum typeName {identifier list};
↳ constants.

Ex:- enum colors {RED, BLUE, GREEN, WHITE};
enum colors aColor;

OR

typedef enum {RED, BLUE, GREEN, WHITE} COLORS;
COLORS aColor;

- Assigning values: enum color {RED, BLUE, GREEN};
enum color x;
x = BLUE;

- Comparing: if (x == BLUE)...

- Switch-case statements:

enum months {JAN, ...}
enum months dateMonth;
switch (dateMonth) {
case JAN: ....

__/__/__

☆ Structure

- Collection of related elements of (field) same or diff. types having same name.

— **Declaration**

(i) struct {field-list} variable_identifier ;

Ex:
```
struct {
    int x;
    int y;
} values
```

(ii) (Tagged) struct tag {field-list} variable identifier ;

Ex:
```
struct values
{
    int x;
    int y;
};
```
→ `struct values coordinates;` (tag)
`void func(struct values v);`

(iii) typedef struct {field-list} TYPE-ID;

Ex:
```
typedef struct {
        char id[10];
        char name[20];
} STUDENT;
STUDENT astudent;

void func (STUDENT stu)
```

__/__/____

— Initialization

```
typedef struct {
        int x;
        int y;
        float t;
        char u;
} SAMPLE;
```

→ `SAMPLE sam1 = {2, 5, 3.2, 'A'}`

`SAMPLE sam2 = {7, 3}`

↓ rest are filled with null.

— Accessing

Here, `SAMPLE values;`

`values.x` (or) `values.u` → (dot-operator / member)

- Member operator (.) >>> (&) operator.
- Copy : `sam2 = sam1`

— Pointers

Here, `SAMPLE *ptr;`
`ptr = &sam1;`

Now, `sam1.x` is equal to `(*ptr).x`

Note `*ptr.x` (w/o ~~pared~~ parenthesis) is invalid.

- Selection operator (->) and member operator (.) are same.

`(*ptr).x` ⟺ `ptr->x`

__/__/__

– Complex Structures

(i) Nested

```
typedef struct {
        int month;
        int day;
        int year;
} DATE
typedef struct {
        int hour;
        int min;
        int sec;
} TIME;
```

```
typedef struct {
        DATE date;
        TIME time;
} STAMP
```

(Seperating is preferred)
(Outer structure is declared after inner structure)

→ Now, to access, STAMP stamp; ~~= {{8, 18, 2014}, {}}~~

```
stamp.date.day;
stamp.time.hour;
```

- `STAMP stamp = {{8, 18, 2014}, {08, 40, 50}};`

(ii) Arrays within structures

```
STUDENT student = {"name1", {10, 20, 30}, 40};
```

(iii) Pointers

PTO →

__/__/____

```
typedef struct {
        char *month;
        int day;
        int year;
} DATE;
char jan[] = "January";
char deb[] = "February";
.
.
stamp.date.month = jan;
```

(iv) Array of Structures

→ Finding average of final marks.

```
#define SIZE 10
typedef struct {
        char name[25];
        int midterm[3];
        int final;
} STUDENT;
STUDENT stuary[10];
int i, sum = 0;
float average;

STUDENT *pwalk;
STUDENT *plast;
plast = stuary + SIZE - 1;
for (pwalk = stuary; pwalk <= plast; pwalk++)
        sum = sum + pwalk -> final;

average = sum / (float) SIZE
```

(v) Structions & Function

- By sending individual members.

res.numerator = multiply (fr1.numerator, fr2.numerator);

- By sending whole structure:

res = multiply (fr1, fr2);

It is possible to return more than one element using structures.

- By passing structure through pointers:

```
void multiply (FRACTION *fr1, FRACTION *fr2,
                                FRACTION *res) {
{
    res -> numerator = fr1 -> numerator * fr2 -> numerator
    res -> denominator = fr1 -> denominator * fr2 -> deno.
    return;
}
int main () {
        FRACTION fr1, fr2, res;
        getFr (&fr1);
        getFr (&fr2);
        multiply (&fr1, &fr2, &res);
        printFr (&res);
}
```

__/__/____

```
getFr (FRACTION *pFr) {
        printf("Write fraction in form x/y: ");
        scanf("%d/%d", &Fr->numerator, &Fr->
                                        denominator);
}
```

Note: . > *(ptr) → > &

__/__/____

☆ Pointers

- The various methods for manipulating data on a computer are:

(i) Indirect Approach (Identifiers): We use memory location addresses symbolically by assigning identifiers to data and manipulating their content thru identifiers.

(ii) Direct Approach (Pointers): Using data addresses directly with the ease and flexibility of symbolic names.

- Pointer is a derived datatype whose value is any of the addresses available for storing and accessing data.
- Unlike character constants, value of pointer constants cannot be changed (address)
- Address of a variable is the address of the first byte occupied by that variable.
- To access value of a pointer p, we use the ~~dereference~~ indirection operator * (*p)

Ex: a = 123
p = &a
*p = 123

Note: While performing arithmetic operations on pointers, make sure that *p is enclosed in parenthesis.

Note: $(\&)^{-1} = *$, $(*)^{-1} = \&$

__/__/____

- General form : type * identifier

Ex
```
int main (void) {
    int a;
    int *p;
    a = 14
    p = &a
    printf("%d %p \n", a, &a);
    printf("%p %d %d \n", p, *p, a);
    return 0;
}
```

O/P → 14 00135760
00135760 14 14

- During definition, pointer can be set to NULL as:
  int *p = NULL;

Ex (Adding two numbers) (using pointers)

```
int main (void) {
    int a, b, r;
    int *pa = &a;
    int *pb = &b;
    int *pr = &r;
    printf("Enter first number : ");
    scanf("%d", pa);
    printf("Enter second number : ");
    scanf("%d", pb);
    *pr = *pa + *pb
    printf("In Sum = %d \n", *pr);
    return 0;
}
```

__/__/__

— Pointers & Functions

Ex:-
```
void exchange (int *, int *);
int main (void){
    int a =5;
    int b= 7;
    exchange (&a, &b);
    printf ("%d %d \n", a,b);
    return 0;
}
void exchange (int *px, int *py){
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
    return;
}
```

— Pointers to Pointers

- To refer to a variable a using the pointer p, we have to dereference it once again;
  **q → *p → a

— Applications

- Dynamic Memory Allocation (calloc, malloc, realloc)
- Passing parameters by reference
- Function Pointers

__/__/____

Arrays and Pointers

- The name of the array is a pointer constant to the first element, whose value cannot be changed.
- Address of first element and name of array both represent same location in memory.

Ex: For an array a, a[0] ← *a

~~a[1] ← *a~~

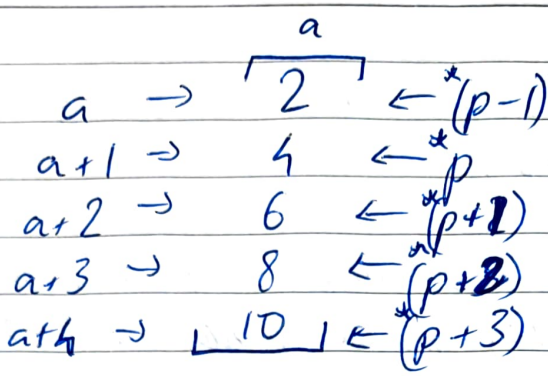
```
int a[5] = {2, 4, 6, 8, 10};
int *p = a;
int i = 0
printf("%d %d \n", a[i], *p);   [2, 2]
```

Here a points to first element a[0] and assigns it to pointer p.

Ex:
```
int a[5] = {2, 4, 6, 8, 10};
int *p;
p = &a[2];
```

Here, a → { 2, 4, 6, 8, 10 }
[0] [1] [2] [3] [4]
↑ ↑ ↑ ↑ ↑
p → [-2] [-1] [0] [1] [2] (relative index)

- Pointer arithmetic is used to manipulate the addresses in pointers for moving element by element in an array.

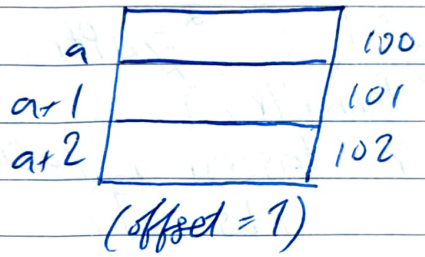


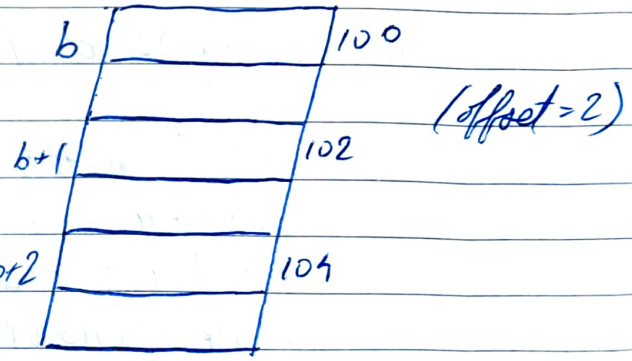
Here n (offset) is 1. (size of element = 1)

- Adding an integer n to a pointer value gives a new value that corresponds to an index location n elements away.

- If offset is more than 1,

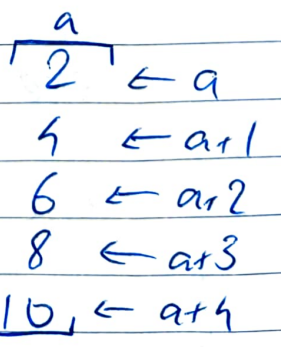
address = pointer + (offset * size of element)





Note:
sizeof(char) = 1 (101)
sizeof(int) = 4 (104)
sizeof(float) = 6 (106)

- a[0] / *(a+0)
  a[1] / *(a+1)
  a[2] / *(a+2)
  a[3] / *(a+3)
  a[4] / *(a+4)



- Addition can be used when one operand is pointer and other is integer.
- Subtraction can be used only when both operands are pointers or when one is pointer and other is index integer.

Exa: p+5, 5+p, p-5, p1-p2, p++, --p.

- p1 >= p2 , p1 != p2
  if (ptr != NULL) == if (ptr)
  if (ptr == NULL) == if (!ptr)

Exa: (Binary Search)

```
int binarySearch (int list[], int *endPtr, int target,
                  int **locnPtr) {
    int *firstPtr, midPtr, lastPtr; (all pointers)
    firstPtr = list;              (list[0])
    lastPtr = endPtr;
    while (firstPtr <= lastPtr) {
        midPtr = firstPtr + (lastPtr - firstPtr) / 2;
        if (target > *midPtr)
            firstPtr = midPtr + 1;
        else if (target < *midPtr)
            lastPtr = midPtr - 1;
        else
            firstPtr = lastPtr + 1;
    }
    *locnPtr = midPtr;
    return (target == *midPtr);
}
```

- Dereferencing of array name of a 2D array is a a pointer to a 1D array
- *(*(table)) refers to first element of first row.

table[i][j] ≡ *(*(table + i) + j)

Ex: (Multiply array elements by 2)

```
void multiply (int *pAry, int size);
int main (void) {
    int ary [SIZE];
    int *pLast;
    int *pWalk;
    pLast = ary + SIZE - 1;
    for (pWalk = ary; pWalk <= pLast; pWalk++) {
        printf ("Enter an integer:");
        scanf ("%d", pWalk);
    }
    multiply (ary, SIZE);
    printf ("Doubled value: \n");  //print-
}
void multiply (int *pAry, int size) {
    int *pWalk;
    int *pLast;
    pLast = pAry + size - 1;
    for (pWalk = pAry; pWalk <= pLast; pWalk++)
        *pWalk = (*pWalk) * 2;
}
```

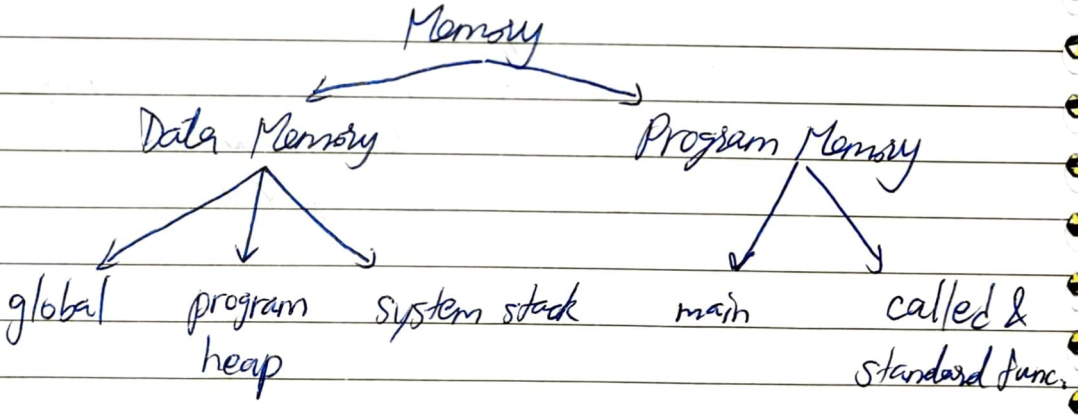
☆ Memory Allocation Functions

(i) Static memory allocation

- Requires declaration and definition of memory to be fully specified in source program.

(ii) Dynamic memory allocation

- Uses predefined functions to allocate and release memory for data while running program.
- Can be used with standard datatypes or derived data types.



- **Program Memory**
- Consists of memory used for main and all called functions.
- main() must be in memory all the time.
- Each called function must be in memory when active.

Data Memory

- Consists of permanent definitions such as global data and constants, local definitions and dynamic data memory.
- Although program code for a function may be in memory all the time, local variables are available only when active.
- In recursion, multiple copies of a local variable are allocated even though only one copy of function is present.
- The memory facilities for these capabilities is known as stack.
- Heap is unused memory allocated to program, available to be assigned during execution.

malloc

- Allocates a block of memory that contains number of bytes specified as parameter.
- Pointer returned by malloc is shown to be casted as integer.

```
pInt = (int *)malloc(sizeof(int));
```

- Successful call → returns address of first byte
  Unsuccessful call → returns NULL pointer.
  Overflow → attempt to allocate memory from heap when insufficient memory

__/__/__

- **calloc** (contiguous memory allocation)
  - To allocate memory for arrays
  - Parameters: No. of elements to be allocated, size of each element.
  - Returns a pointer to first element of array.
  - `(int *) calloc (200, sizeof (int))`
- **realloc** (reallocation)
  - Given a pointer to a previously allocated block of memory, realloc changes size of block by deleting/extending memory at end of block.
  - If extending is not possible, realloc allocates new block.
  - Copies existing → new alloc and deletes old alloc.

  `ptr = (int *) realloc (ptr, 15 * sizeof (int));`
- **free**
  - Can be used when memory allocated by malloc, calloc, realloc are no longer needed.
  - Error for null pointers, pointers of diff type, released memory, pointers other than first element of block.

__/__/____

- Releasing memory does not change value in pointer, it will still contain address in the heap.
- After freeing memory, clear pointer → NULL

```
void free (void *ptr);
```

☆ Array of Pointers (Ragged)

- Rows contain different no. of elements (2D array)

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int rowNum, colNum, i, j;
    int **table;
    printf("\n enter no. of rows: \n");
    scanf("%d", &rowNum);
    table = (int **) calloc (rowNum + 1, sizeof(int*));
    for (i=0; i<rowNum; i++) {
        printf(" Enter size of %d row", i+1);
        scanf("%d", &colNum);
        table[i] = (int *) calloc (colNum + 1, sizeof(int));
        printf("\n enter %d row elements:", i+1);
        for (j=1; j<=colNum; j++) {
            scanf("%d", &table[i][j]);
        }
        table[i][0] = colNum;
        printf("size of row number [%d] = %d" i+1,
                                    table[i][0]);
}
```

```
    table[i] = NULL;
    for (i=0; i<rowNum; i++){
        printf("displaying %d row elements \n", i+1);
        for (j=0; j<=*table[i]; j++)
            printf("%5d", table[i][j]);
        printf("\n");
    }
    return 0;
}
```

O/P
```
3 10  11  12
2 13  14
5 15  16  17  18  19
4 20  21  22  23
7 24  25  26  27  28  29  30
```

☆ Sparse Matrix

- Matrix is a 2D data object made of m rows and n columns, having total m*n values.
- If most of elements of matrix have value of 0, it is a sparse matrix.

Why?
- Less non-zero elements than zeros, thus, less memory used to store, also saving computing time.
- Two common representations of sparse matrix are:

__/__/____

- Using Arrays

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 9 \\ 3 & 0 & 0 \end{bmatrix} \rightarrow$$

Row →	0	1	2
Column →	1	2	0
Value →	1	9	3

- Using linked lists

→
```
#define MAX_TERMS 101
typedef struct {
        int col;
        int row;
        int value;
        } term;
term a[MAX_TERMS];
```

→

	row	col	value
a[0]	6	6	5
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	2	3
[5]	2	5	28

- Here a[0].row → no. of rows
  a[0].col → no. of columns
  a[0].value → total no. of non-zero entries.

__/__/____

MODULE - 2

✱ Data Type

- A datatype defines a set of values and a set of operations that can be applied on those values, which have a particular representation, 1's complement, 2's complement or sign magnitude.

— Abstract Data Type (ADT)

- It is a data declaration packaged together with operations meaningful ~~to~~ on the datatype.
- We encapsulate the data and operations on data and hide them from user. Eg:- push(), pop(), etc.

— Arrays

- Ordered set which consist of fixed no. of objects
- Operations : Creation of array of fixed size, store and retrieve elements, destroy array.

— List

- Ordered set consisting of variable no. of objects.
- Operations : Creation, insertion, deletion of elements, destroy list, store and retrieve elements.

__/__/____

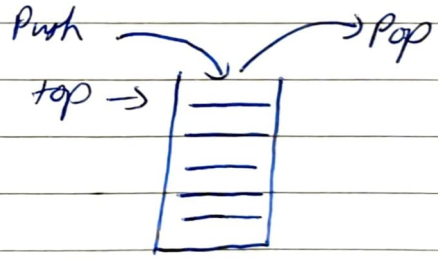
— Array as ADT

- Objects: Set of pairs of <index, value>
  For each value of index, there is a value from item set.
  Index is a finite ordered set of one or more dimensions.

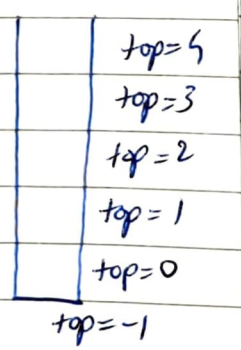
- Functions: For all A ∈ Array, i ∈ Index, x ∈ item, j ∈ int
  Array Create (j, list) → return array of j dimension
  list is j-tuple where i^{th} element is size.
  Item Retrieve (A, i) → retrieve element A[i] from array
  Array Store (A, i, x) → store value x at A[i] in array A.

☆ Stack (aka Last-In-First-Out List)

- Ordered list in which insertion and deletion are made at one end called top.




```
#include <stdio.h>
#include <stdbool.h>
#define MAX_SIZE 3

struct Stack {
        char data[MAX_SIZE];
        int top;
};
void initialize (struct Stack *stack){
        stack->top = -1;
}
```



__/__/____

```
bool isEmpty (struct Stack *stack){
    return stack -> top = -1;
}
bool isFull (struct Stack *stack){
    return (stack -> top == MAX_SIZE - 1);
}
void push (struct Stack *stack, char value){
    if (isFull(stack)){
        printf("Stack overflow! Cannot push \n");
        return;
    }
    stack -> data[++ stack -> top] = value;
}
char pop (struct Stack *stack){
    if (isEmpty(stack)){
        printf("Stack underflow! Cannot pop \n");
        return '\0';
    }
    return stack -> data[stack -> top --];
}
void display (struct Stack *stack){
    if (isEmpty(stack)){
        printf("Stack is empty. \n");
        return;
    }
    printf("Stack contents : \n");
    for (int i = stack -> top; i >= 0; --i){
        printf("| %c | \n", stack_data[i]);
    }
    printf("|____|");
}
```

Same

Same

```
int main () {
    Struct Stack myStack;
    initialize (&myStack);
    int choice;
    char value;
    while (1) {
        printf ("\n Menu : \n");
        printf ("1. Push \n");
        printf ("2. Pop \n");
        printf ("3. Display \n");
        printf ("4. Exit \n");
        printf ("Enter your choice: ");
        scanf ("%d", &choice);
        switch (choice) {
            case 1:
                printf ("Enter a character to push: ");
                scanf ("%c", &value);
                push (&myStack, value);
                break;
            case 2:
                ~~printf~~
                value = pop (&myStack);
                if (value != '\0')
                    printf ("Popped element: %c \n", value);
                break;
            case 3:
                display (&myStack);
                break;
            case 4:
                printf ("Exiting program \n");
                return 0;
            default: printf ("Invalid choice. Try again. \n");
    }
```

__/__/____

```
return 0;
}
```

Infix, Prefix & Postfix Expression

- In an expression if $A\ (op)\ B$ → infix
  $(op)\ A\ B$ → prefix
  $A\ B\ (op)$ → postfix
  where A, B are operands, op → operator.

- During evaluation, infix expression is precedence-dependant while prefix and postfix are independent of precedence.

- Precedence: (), [], →, ++, --, !, sizeof, (type), *, /, %, +, -, >>, <<, >, >=, <, <=, ==, !=, &, ^, |, &&, ||, ?: (if then else)

- Infix → Prefix : ↻ + $S_2\ S_1\ (op)$ + ↻
- Infix → Postfix : (→ stack, Alnum → expression
  if operator precedence < top of stack
  ↳stack ↳exp
  if), from stack S_1, S_2 → exp.
- Prefix → Postfix : ↻ + $S_1\ S_2\ (op)$
- Postfix → Prefix : $(op)\ S_2\ S_1$
- Prefix → Infix : ↻ + $S_1\ (op)\ S_2$
- Postfix → Infix : $S_2\ (op)\ S_1$

S_1
S_2

__/__/____

Ex: (Infix → Postfix)

$((a + (b * c)) + (d * e)) \rightarrow abc * + de * +$

$(a/(b-c) + d)) * (e-a) * c$

$abc - d + / ea - c * *$

$(((a/b) - c) + (d * e)) - (a * c))$

$ab / c - de * + ac * -$

Ex: $A + (B * C - (D / E \uparrow F) * G) * H$ (Infix)

Symbol	Stack	Expression (Postfix)
A		A
+	+	A
(	+ (	A
B	+ (	AB
*	+ (*	AB
C	+ (*	ABC
-	+ (-	ABC *
(	+ (- (	ABC *
D	+ (- (	ABC * D
/	+ (- (/	ABC * D
E	+ (- (/	ABC * DE
↑	+ (- (/ ↑	ABC * DE
F	+ (- (/ ↑	ABC * DEF
)	+ (-	ABC * DEF ↑ /
*	+ (- *	ABC * DEF ↑ /
G	+ (- *	ABC * DEF ↑ / G
)	+	ABC * DEF ↑ / G * -
*	+ *	ABC * DEF ↑ / G * -
H	+ *	ABC * DEF ↑ / G * - H

__/__/____

∴ Postfix → $ABC^{\wedge}DEF\uparrow/G*-H^{\wedge}+$

Exa (Prefix → Postfix)

$*+AB-CD$ (prefix)

$DC-BA+*$ (↻)

Input	Stack Op	Exp (Postfix)
D	Push	D
C	Push	DC
−	Pop	CD−
B	Push	~~CD−~~ B
A	Push	~~CD−~~ BA
+	Pop	AB+
*	Pop	AB+CD−*

Exa (Postfix → Infix)

$ABC/-AK/L-*$ (Postfix)

Input	Exp (Infix)
A	A
B	AB
C	ABC
/	A(B/C)
−	(A−(B/C))
A	(A−(B/C))A
K	(A−(B/C))AK
/	(A−(B/C))(A/K)
L	(A−(B/C))(A/K)L
−	(A−(B/C))(A/K)−L)
*	(A−(B/C))*(A/K)−L)

__/__/____

Exa (Postfix → Prefix)

ABC/-AK/L-* (Postfix)

I/P	Exp
A	A
B	AB
C	ABC
/	A(/BC)
-	(-A(/BC))
A	(-A(/BC))A
K	(-A(/BC))AK
/	(-A(/BC))(/AK)
L	(-A(/BC))(/AK)L
-	(-A(/BC))(-(/AK)L)
*	(*(-A(/BC))(-(/AK)L))

⇒ *-A/BC-/AKL (Prefix)

Exa (Prefix → Infix)

+*AB-CD (Prefix) ⇄ DC-BA*+

I/P	Exp
D	D
C	DC
-	(C-D)
B	(C-D)B
A	(C-D)BA
*	(C-D)(A*B)
+	(A*B)+(C-D)

= A*B+C-D

__/__/____

(Infix → Prefix)

Exa — (A - B / C) * (A / K - L) (Infix)

(L - K / A) * (C / B - A)

I/P	Exp
L	L
K	LK
A	LKA
/	L(KA/)
-	LKA/-
C	C
B	CB
/	CB/
A	CB/A
-	CB/A-

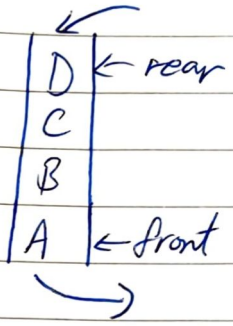
→ LKA/- CB/A- *

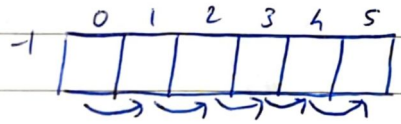
→ *-A/BC-/AKL

Queues (aka First-In-First-Out List)

- It is an ordered list in which insertion takes place at rear end and deletion takes place at front end.

- Used by operating system (OS) to create job queues. If OS does not use priorities, then jobs are processed in the order they enter the system.






```
#include <stdio.h>
#define MAX 5

typedef struct {
    int x[MAX];
    int front;
    int rear;
} queue;
void insertq (queue *q, int x) {
    if (q->rear == MAX-1)
        printf("\n Overflow \n");
    else {
        q->x[++q->rear] = x;
        if (q->front == -1) {
            q->front = 0;
        }
    }
}
```

```
int deleteq (queue *q){
    int x;
    if (q->front == -1){
        printf ("\n Underflow!! \n");
        return -1;
    } else if (q->front = q->rear){
        x = q->x[q->front];
        return x;
    }else{
        return q->x[q->front++];
    }
}
```

Same

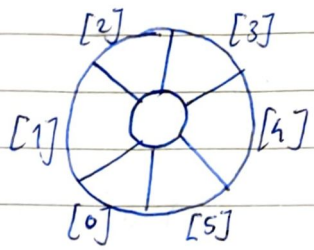
```
void displayq (queue q){
    int i;
    if (q.front == -1){
        printf ("\n Queue is Empty ! \n");
    }else{
        printf ("\n Queue is : \n");
        for (i = q.front; i <=q.rear; i++){
            printf ("%d \n", q.x[i]);
        }
    }
}
int main (){
    queue q;
    q.front = -1
    q.rear = -1;
    int ch, x;
    while (1){
```

[SAME LIKE STACK]

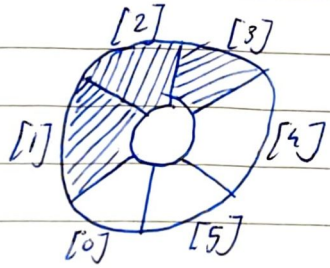
```
}
```

— Circular Queues

- Here, front → one position anti-clockwise from first element.
  rear → current end.



front = 0, rear = 0



front = 0, rear = 3

Note: Linear increment = i = i+1 ~~(1 2 3 4 5)~~
Circular increment = i = (i+1) % SIZE
(for rear) (and front)

```
                                        , char *str
void insertcq (CircularQueue *cq){
    if((cq->rear + 1) % MAX == cq->front){
        printf("Queue is full \n");
        return;
    }
    if (cq->front == -1){
        cq->front = 0;
    }
    cq->rear = (cq->rear + 1) % MAX;
    strcpy(cq->queue[cq->rear], str);
    printf("Inserted %s \n", str);
}
```

```
void deleteCq (CircularQueue *cq) {
    if (cq->front == -1) {
        printf("Queue is empty\n");
        return;
    }
    printf("Deleted %s\n", cq->queue[cq->front]);
    if (cq->front == cq->rear)
        cq->front = cq->rear = -1;
    else {
        cq->front = (cq->front + 1) % MAX;
    }
}
void displayCq (CircularQueue *cq) {
    if (cq->front == -1) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements : \n");
    int i = cq->front;
    while (cq->front != cq->rear) {
        printf("%s", cq->queue[cq->front]);
        cq->front = (cq->front + 1) % MAX;
    }
    printf("%s\n", cq->queue[cq->rear]);
}
```

1 5 2
1 2 5

__/__/__

Priority Queue

- It is a data structure in which the intrinsic ordering of elements determines the results of basic operations.
- Ascending priority queue is a collection of items in which smallest item can be removed. (Queue)
- Descending priority queue is a collection of items in which largest item can be removed. (Stack)

→
```
typedef struct {
    int data[MAX];
    int size;        (No. of elements)
} PriorityQueue (Ascending)

void pqinsert(PriorityQueue *pq, int x) {
    if (full(pq)) {
        printf("Queue is full \n");
        return;
    }
    int i = pq->size - 1;        (previous element)(index)
    while (i >=0 && pq->data[i] > x) {
        pq->data[i+1] = pq->data[i];
        i--;
    }
    pq->data[i+1] = x;
    pq->size++;
}
```
(Ascending)

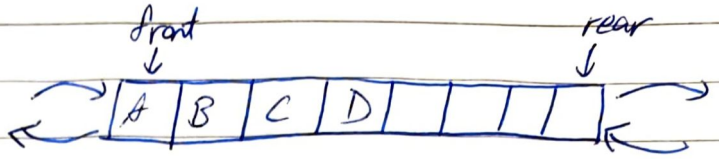
```
int pqmindelete (PriorityQueue *pq) {
~~void~~ if (empty (pq)){
        printf("Priority Queue is empty \n");
        exit(1);
    }
    int min = pq -> data[0];
    for(int i=1; i<pq->size; i++){
        pq->data[i-1] = pq->data[i];
    }
    pq->size--;
    return min;
}
```

→ For descending (delete ~~largest~~ element),

```
void pqinsert (PriorityQueue *pq, int x){
    int i = pq->size-1;
    while (i>=0 && pq->data[i]<x){
        pq->data[i+1] = pq->data[i];
        i--;
    }
    pq->data[i+1] = x;
    pq->size++;
}
```

0. 0.

__/__/__



— Dequeue

- Here elements can be inserted / deleted from both front and rear of queue.

- It is full iff $(dq \to rear + 1) \% MAX == dq \to front$.

```
void insertRear(Deque *dq, char *str){
    if(full(dq)){
        printf("Deque is full\n");
        return;
    }
    if(empty(dq)){ dp->front = dp->rear = 0; }
    else{
        dq->rear = (dq->rear+1) % MAX;
    }
    strcpy(dq->data[dq->rear], str);
}
void insertFront(Deque *dq, char *str){
        :
    else{
  dq->front=(dq->front - 1 + MAX) % MAX.
strcpy(dq->data[dq->front], str);
```

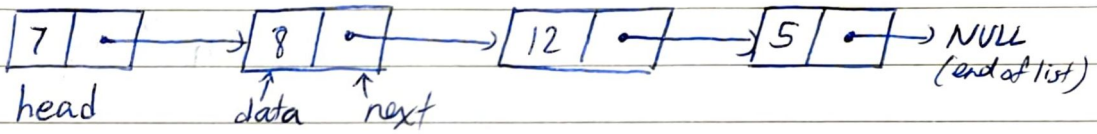
```
~~void insertFr~~
char * deleteFront(Deque *dq){
    if(empty(dq)){
        printf("Deque is empty!\n");
        exit(1);
    }
    char *str = dq->data[dq->front];
```

```
if (dq->front == dq->rear){
    initialize(dq);
}else{
    dq->front = (dq->front + 1) % MAX;
}
return str;
}
char *deleteRear(Deque *dq){
    :
    if (dq->front == dq->rear){
        initialize(dq);
    }else{
        dq->rear = (dq->rear - 1 + MAX) % MAX;
    }
    return str;
}
```

MODULE - 3

☆ Linked Lists

- In arrays, successive items are located at fixed distance apart. Majo disadvantage ~~of~~ were movement of data during insertion and deletion and wastage of space in storing n ordered lists of varying size.




```
struct Node {
    int data;
    struct Node *next;
};
int main() {
    struct Node *head;
    head = (struct Node*) malloc (sizeof (struct Node));
    struct Node *second;
    struct Node *third;
    second = (struct Node*) malloc (sizeof (struct Node));
    third = (struct Node*) malloc (sizeof (struct Node));
    head -> data = 7;
    head -> next = second;
    second -> data = 8;
    second -> next = third;
    third -> data = 12;
    third -> next = ~~NULL~~ fourth;
    ~~fourth~~
    struct Node *fourth;
```

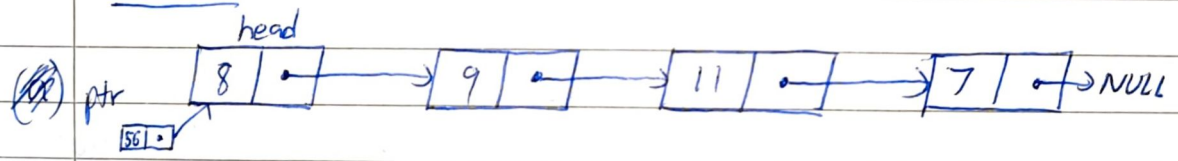
```
fourth = (struct Node *) malloc (sizeof (struct Node));
fourth -> data = 5;
fourth -> next = NULL;
}
```

- Traversing a Linked List

```
void linkedListTraversal (struct Node *ptr){
    while (ptr != NULL){
        printf (" Element: %d \n ", ptr -> data);
        ptr = ptr -> next;
    }
}
```

In main function, linkedListTraversal(head);

- Insertion



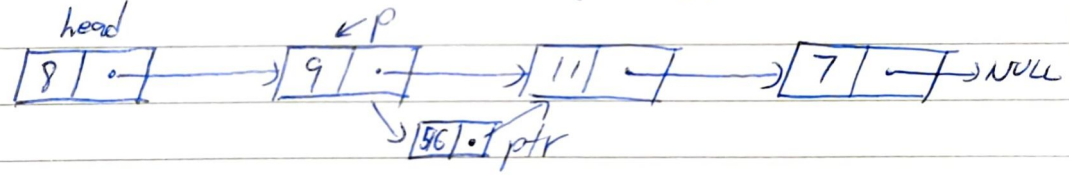
(a)
```
struct Node * insertAtFirst (struct Node *head, int data){
    struct Node *ptr = (struct Node *) malloc (sizeof(
                                    struct Node));
    ptr -> next = head;
    ptr -> data = data;
    return ptr;
}
```

In main function, head = insertAtFirst (head, 56);
linkedListTraversal(head);

(b) struct Node *insertAtIndex (struct Node *head, int data, int index) {

```
struct Node *ptr = (struct Node *)malloc(sizeof(struct Node));
int i=0;
struct Node *p = head;
while (i!=index-1){
    p = p->next;
    i++;
}
ptr->next = p->next;
p->next = ptr;
ptr->data = data;
return head;
}
```

In main func., head = insertAtIndex(head, 56, 2)
linkedListTraversal(head);



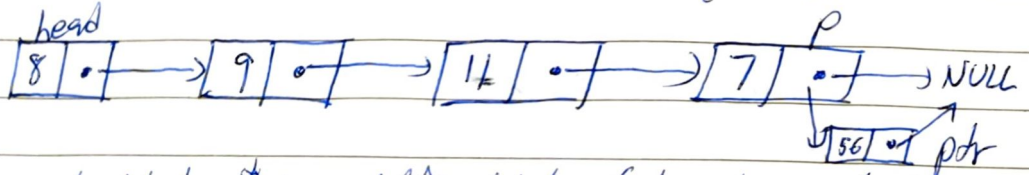
(c) ~~struct Node *insertAtEnd (struct Node *)malloc (sizeof(struct Node));~~

(c) struct Node *insertAtEnd (struct Node *head, int data){

```
struct Node *ptr = (struct Node *)malloc(sizeof(
struct Node *p = head;          struct Node));
ptr->data = data.
while (p->next!=NULL)
    p = p->next;
}
p->next = ptr;
```

```
    ptr -> next = NULL;
    return head;
}
```

In main func., head = insertAtEnd(head, 56);
linkedListTraversal(head);



(d) struct Node *insertAfterNode(struct Node *head, struct Node *prevNode, int data) {

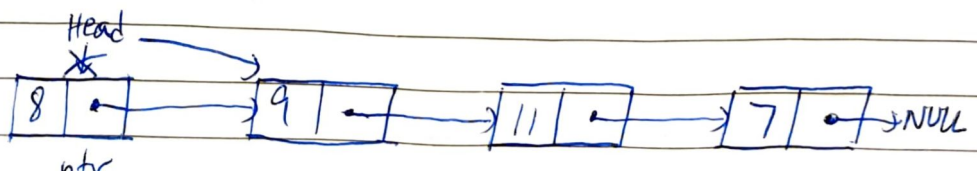
```
    struct Node *ptr = (struct Node *) malloc(sizeof(
                                    struct Node));
    ptr -> next = prevNode -> next;
    ptr -> data = data;
    prevNode -> next = ptr;
    return head;
}
```

In main func., head = insertAfterNode(head, second, 45);
linkedListTraversal(head);

— Deletion

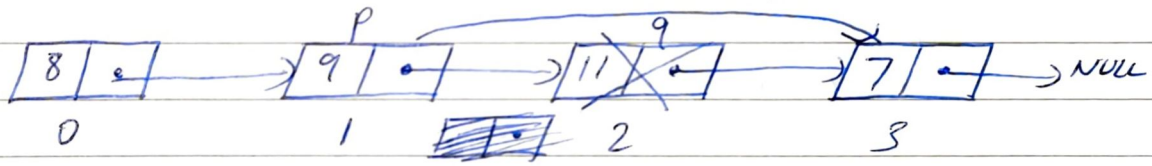
(a) struct Node *deleteAtFirst(struct Node *head, ~~int data~~) {

```
    ~~struct Node *ptr = (struct Node *) malloc(sizeof(
                                    struct Node));~~
    struct Node *ptr = head;
    head = head -> next;
    free(ptr);
    return (head);
}
```

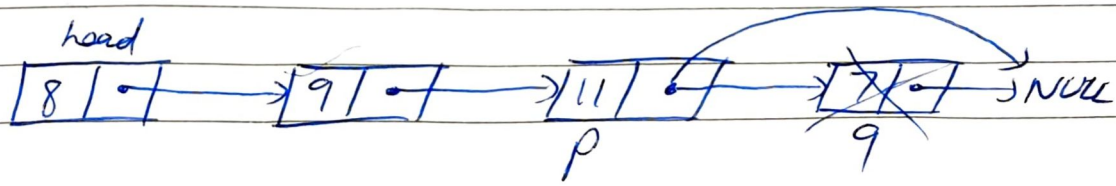


__/__/____

(b)
```
struct Node *deleteAtIndex(struct Node *head, int index){
    struct Node *p = head; int i=0;
    while (i != index-1){
        p = p->next; i++;
    }
    struct Node *q = p->next;
    p->next = q->next;
    free(q);
    return head;
}
```
In main func, `head = deleteAtIndex(head, 2);`
`linkedListTraversal(head);`

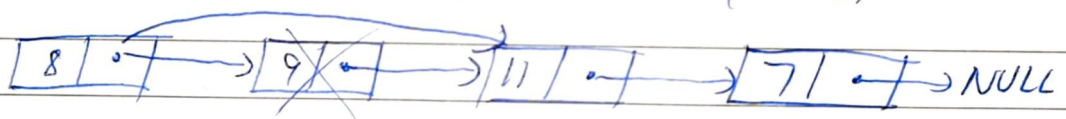


(c)
```
struct Node *deleteAtEnd(struct Node *head){
    struct Node *p = head;
    while (p->next != NULL){
        p = p->next;
    }
    struct Node *q = head->next;
    while (q->next != NULL){
        p = p->next;
        q = q->next;
    }
    p->next = NULL;
    free(q);
    return head;
}
```

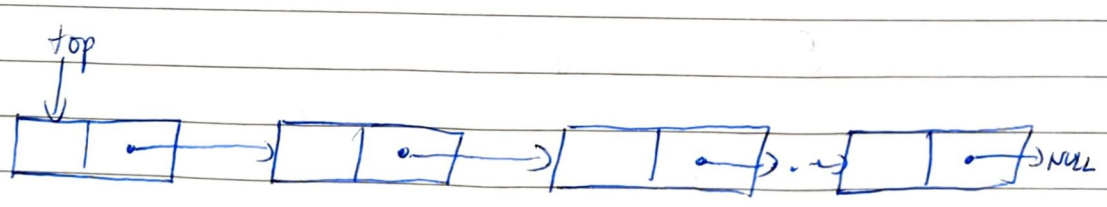


__/__/____

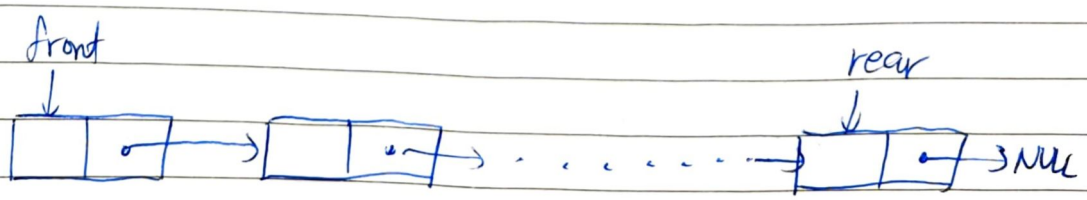
(d)
```
struct Node *deleteValue (struct Node *head, int value)
    struct Node *p = head;
    struct Node *q = head->next;
    while (q->data != value && q->next != NULL){
        p = p->next;
        q = q->next;
    }
    if (q->data == value){
        p->next = q->next;
        free (q);
    }
    return head;
}
In main func,  head = deleteValue (head, 9);
               linkedListTraversal (head);
```



- Linked Stack :

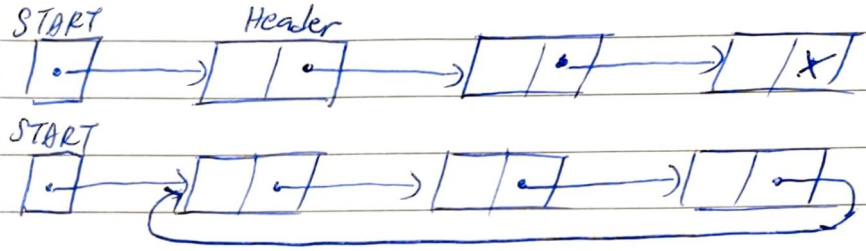


- Linked Queue



__/__/___

- A grounded header list is where last node contains a null pointer
- A circular header list is where last node points back to header node



— Polynomials

$$A(x) = a_{m-1}x^{e_{m-1}} + a_{m-2}x^{e_{m-2}} + \ldots\ldots + a_0 x^{e_0}$$

— Stack using linked list

```
struct Node {
        int data;
        struct Node * next;
};
struct Node * top = NULL;   (Global variable)
int isEmpty (struct Node * top){
    if (top == NULL){
        return 1;
    } else {
        return 0;
    }
}
```

__/__/____

```
int isFull (struct Node *top){
    struct Node *p = (struct Node *) malloc (sizeof(
                                    struct Node));
    if (p == NULL){
        return 1;
    }else{
        return 0;
    }
}

struct Node *push (struct Node *top, int x){
    if (isFull (top)){
        printf ("Stack Overflow\n");
    }
    else {
        struct Node *n =(struct Node *) malloc (sizeof(
                                    struct Node));
        n -> data = x
        n -> next = top;
        top = n;
        return top;
    }
}
```

(In main func, top = push (top, 78);
display (top);)

```
void display (struct Node *ptr){
    while (ptr != NULL){
        printf ("Element : %d \n", ptr -> data);
        ptr = ptr -> next;
    }
}
```

(Harry Bhai Edition)

```
int pop (struct Node *ptr) {        (local variable)
    if (isEmpty(ptr)) {
        printf("Stack Underflow \n");
    }
    else {
        struct Node *n = ptr;
        top = ptr->next;
        int x = n->data;
        free(n);
        return x;
    }
}
```

In main func, `int element = pop(top);`
`printf("Popped element = %d \n", element);`

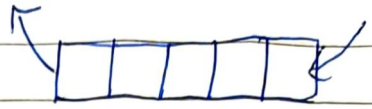
OR

Instead of making top a global variable:

```
struct Node *pop (struct Node *top) {
    if (isEmpty(top)) {
        printf("Stack Underflow \n");
    } else {
        struct Node *n = top;
        top = top->next;
        int x = n->data;
        free(n);
        return x;
    }
}
```

__/__/____

Queue using linked list



```
struct Node {
    int data;
    struct Node *next;
};
struct Node *front = NULL;
struct Node *rear = NULL;          (Global)

void display (struct Node *ptr){
    printf("Queue : \n");
    while (ptr != NULL){
        printf("Element : %d \n", ptr->data);
        ptr = ptr->next;
    }
}
void enqueue (int val) {
    struct Node *n = (struct Node *)malloc(sizeof(struct
                                                  Node));
    if (n == NULL){
        printf("Queue is full");
    }
    else {
        n->data = val;
        n->next = NULL;
        if (front == NULL){                (empty)
            front = rear = n;
        }
        else {
            rear->next = n;
            rear = n;
        }
    }
}
```

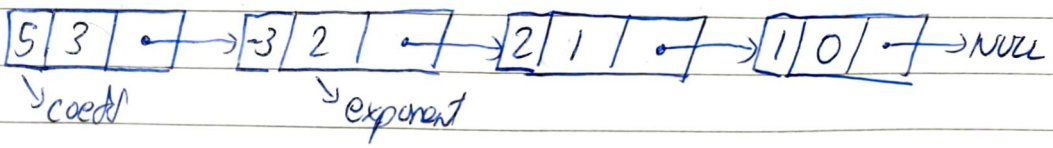
In main func, enqueue(7);
display(front);

```
int
~~void~~ dequeue(){
    int val = -1;
    struct Node *n = front;
    if (front == NULL)
        printf("Queue is empty");
    else {
        front = front->next
        val = n->data
        free(n);
    }
    return val;
}
```

In main func, printf("Removed %d \n", dequeue());

- Polynomial (Contd.)

$5x^3 - 3x^2 + 2x + 1$



- Here, exponents should be in decending order. (sorted)

→
```
struct Node {
    int coeff;
    int exp;
    ~~struct~~ struct Node *~~next~~ link;
}
```

→
```
struct Node *create(struct Node *head){
    int n;
    printf("Enter number of terms: ");
    scanf("%d", &n);
    int i;
    int coeff, exp;
    for(i=0; i<n; i++){
        printf("Enter coefficient of term %d: ", i+1);
        scanf("%d", &coeff);
        printf("Enter the exponent for term %d: ", i+1);
        scanf("%d", &exp);
        head = insert(head, coeff, exp);
    }
}
```

→
```
struct Node *insert(struct Node *head int coeff, int exp)
    struct Node *newP = malloc(sizeof(struct Node));
    struct node *temp;
    newP -> coeff = coeff;
    newP -> exp = exp;
    newP -> link = NULL;
    if (head == NULL || exp > head -> exp){
        newP -> link = head;
        head = newP;
    }
    else{
        temp = head;
        while (temp -> link != NULL && temp -> link -> exp > exp)
            temp = temp -> link;
        newP -> link = temp -> link;
        temp -> link = newP;
    }
    return head;
}
```

__/__/____

```
void print(struct Node *head){
    if (head == NULL)
        printf("No polynomial");
    else {
        struct Node *temp = head;
        while (temp != NULL){
            printf("(%dx^%d)", temp->coeff, temp->exp);
            temp = temp->link;
            if (temp != NULL)
                printf(" + ");
            else
                printf("\n");
        }
    }
}
```

In main func, `struct Node *head = NULL;`
`head = create(head);`

— Addition of Two Polynomials

```
void polyAdd(struct Node *head1, struct Node *head2){
    struct Node *ptr1 = head1;
    struct Node *ptr2 = head2;
    struct Node *head3 = NULL;
    while (ptr1 != NULL && ptr2 != NULL){
        if (ptr1->exp == ptr2->exp){
            head3 = insert(head3, ptr1->coeff + ptr2->coeff,
                           ptr1->exp);
            ptr1 = ptr1->link;
            ptr2 = ptr2->link;
        }
```

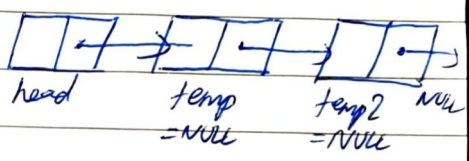
```
        else if (ptr1->exp > ptr2->exp){
            head3 = insert(head3, ptr1->coeff, ptr1->exp);
            ptr1 = ptr1->link;
        }
        else if (ptr2->exp > ptr1->exp){
            head = insert(head3, ptr2->coeff, ptr2->exp);
            ptr2 = ptr2->link;
        }
    }
    while (ptr1 != NULL){
        head3 = insert(head3, ptr1->coff, ptr2->exp);
        ptr1 = ptr1->link;
    while (ptr2 != NULL){
        head3 = insert(head3, ptr2->coff, ptr2->exp);
        ptr2 = ptr2->link;
    }
    printf("Added polynomial : ");
    print(head3);      <- call print function.
}
```

- Multiplication of Polynomials

```
void polyMult (struct Node * head1, struct Node *head2)
    struct Node *ptr1 = head1;
    struct Node *ptr2 = head2;
    struct Node *head3 = NULL;
    if (head1 == NULL || head2 == NULL){
        printf("Zero polynomial \n");
        return;
    }
```

```
while (ptr1 != NULL){
    while (ptr2 != NULL){
        head3 = insert(head3, ptr1->coeff * ptr2->coeff,
                              ptr1->exp + ptr2->exp);
        ptr2 = ptr2->exp;
    }
    ptr1 = ptr1->link;
    ptr2 = head2;
}
print(head3);
}
```

— Inverting a linked list




```
while (head != NULL){
    temp2 = head->link;
    head->link = temp;        (NULL at first)
    temp = head
    head = temp2;
}
head = temp.
return head;
```

— Merge Two linked list

```
void merge (struct Node *ptr1, struct Node *ptr2){
    struct Node *temp;
    if (isEmpty(ptr1))
        return ptr2;
    else {
        if (!isEmpty(ptr2)) {
```

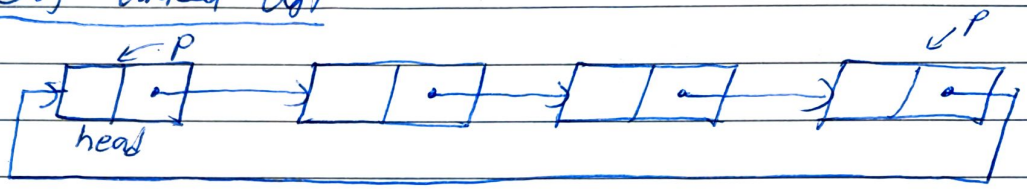
__/__/____

```
for (temp = ptr1; temp->link != NULL; temp=
                                    temp->link);
    temp->link = ptr2;
}
return ptr1;
}
}
```

__/__/____

```
for (temp = ptr1; temp->link != NULL; temp = temp->link);
    temp->link = ptr2;
}
return ptr1;
}
```

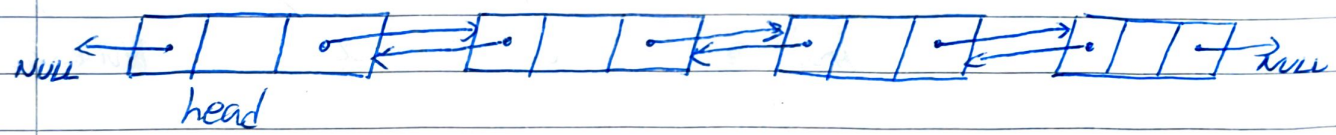
- Circular linked list




```
~~void LinkedListTraversal~~
void CircularLinkedListTraversal (struct Node *~~ptr~~ head){
    struct Node * ptr = head;
    ~~while (ptr != head){~~
        ~~ptr = ptr->next;~~
        ~~printf("Element~~
    do {
        printf("Element is %d \n", ptr->data);
        ptr = ptr->next;
    } while (ptr != head);
}
~~void~~
struct Node * insertAtFirst (struct Node * head, int data){
    Struct Node * ptr = (struct Node *) malloc (sizeof(
                                    struct Node));
    ptr->data = data;
    struct Node *p = head->next;
    while (p->next != head){
        p = p->next;                    (get last element)
}
```

```
    p->next = ptr;
    ptr->next = head;
    head = ptr;
    return head;
}
struct Node *insertAtEnd(struct Node *head, int data){
    struct Node *ptr = (struct Node *)malloc(sizeof(struct Node));
    ptr->data = data;
    struct Node *p = head->next;
    while (p->next != head)
        p = p->next;
    p->next = ptr;
    ptr->next = head;
    return head;
}
```

— Doubly Linked list




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

In main function, struct Node *head = malloc.....

```
head->prev = NULL;
head->data = 10;
head->next = NULL;
```

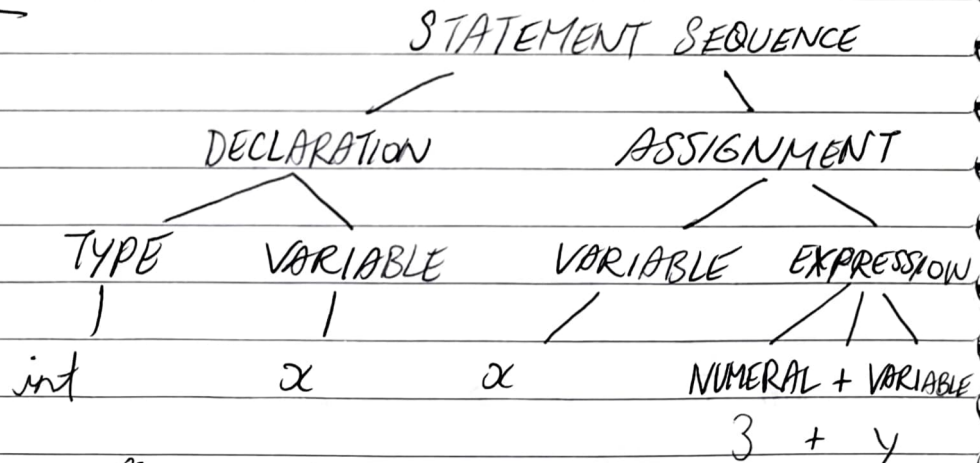
__/__/__

Trees

- Natural structures for representing certain kinds of hierarchial data, allowing us to associate a parent-child relationship between various pieces of data
- Binary Search Trees help to order the elements in such a way that searching takes less time compared to other data structures. (linked list is a linear data structure).

— Parse Tree

int x;
x = 3+y

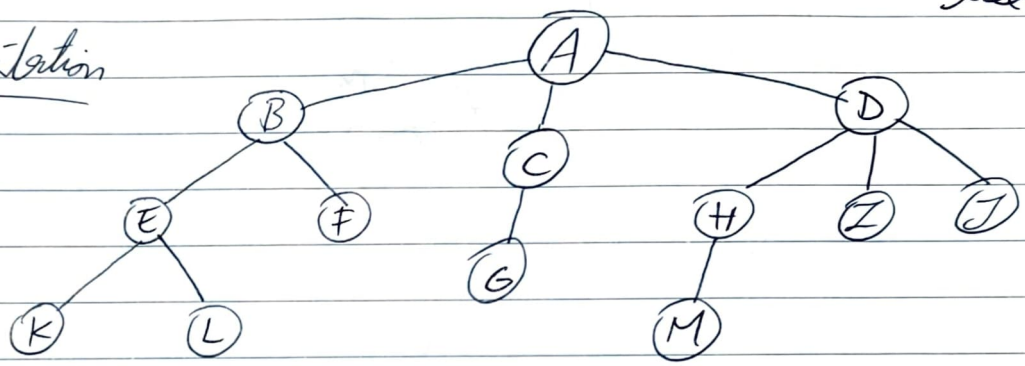


- A tree is a finite set of one or more nodes such that there is a specially designated node called root. and the remaining nodes are partitioned into disjoint sets where each is a tree (subtrees)
- Root is always at the top of the tree, and a node stands for the item of information + branches to other nodes.
- Degree of a node = number of subtrees of a node.
- Node with degree = 0 is called leaf / terminal node.

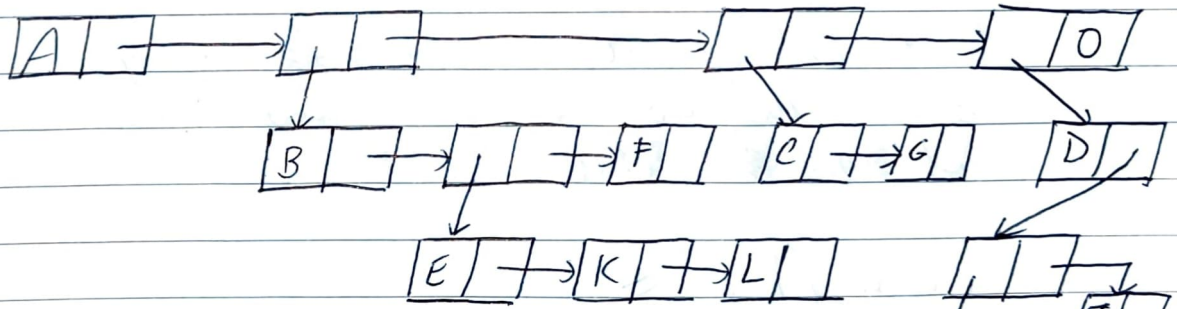
__/__/____

- Degree of tree = maximum degree of nodes in the tree
- Level of node = level of node's parent + 1
- Height / Depth of a tree = maximum level of any node in tree.

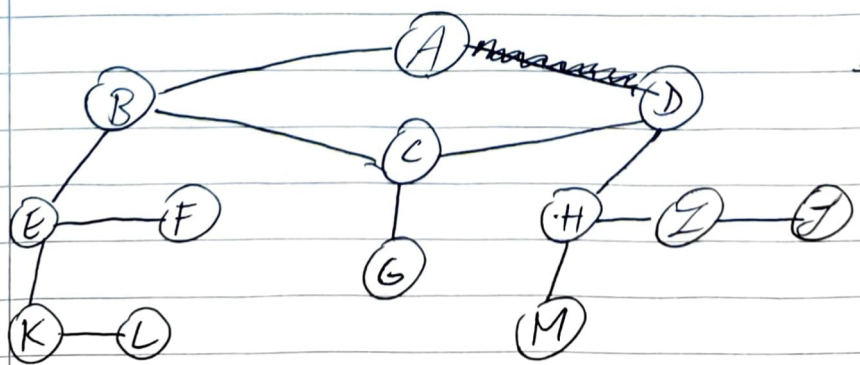
☆ Representation



→ A(B(E(K,L),F),C(G),D(H(M),I,J)))

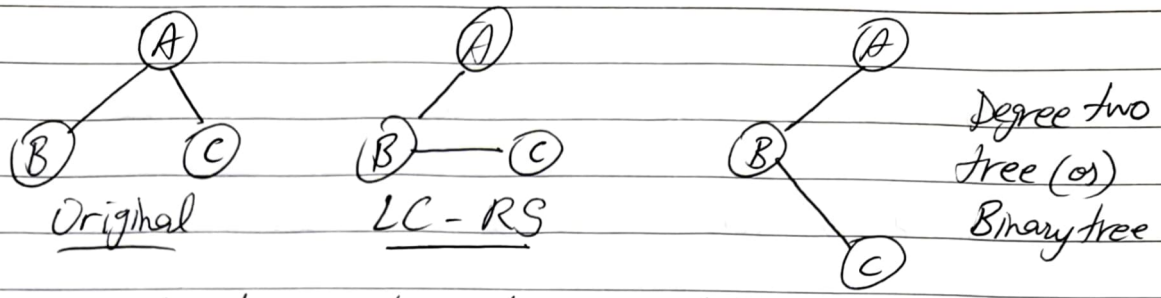


→ Left Child - Right Sibling

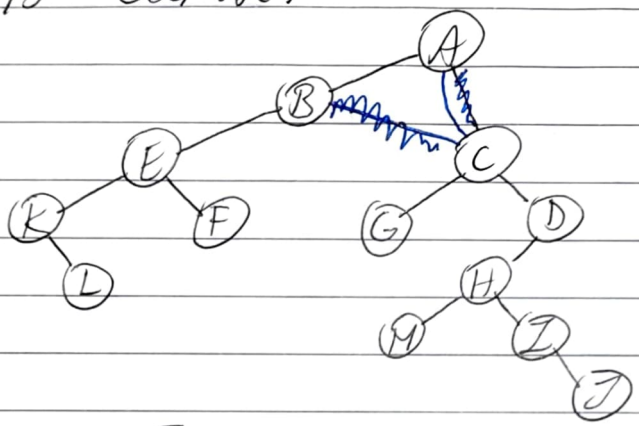


$13 = 5 + 4$ (0, 1, 2)

$13 = 4 + 5 + 4$ (n_1, n_0, n_2)



Note: To get degree-two tree, rotate LC-RS tree by 45° clockwise.



Binary Tree

- Finite set of nodes that is either empty or consists of root + two disjoint binary trees called left subtree and right subtree.

- Degree (BT) ≤ 2

- For level i of BT, max no. of nodes $= 2^{i-1}$
- For BT of depth k, max no. of nodes $= 2^k - 1$

- For a non-empty binary tree, if n_0 is no. of leaf nodes and n_2 is no. of nodes of degree 2.

$$n_0 = n_2 + 1$$

$n_0 \rightarrow$ degree 0 (leaf)
$n_1 \rightarrow$ degree 1
$n_2 \rightarrow$ degree 2

Proof: $n = n_0 + n_1 + n_2$
(total node count)

$12 = 4 + 2(4)$
$\downarrow \quad \downarrow$
$n_1 \quad n_2$

__/__/____

(vertices = edges + 1)

If B = no. of branches in binary tree = $n-1$

$n = B + 1$, $B = n_1 + 2n_2$

↳ 2 branches / children

$n = 1 + n_1 + 2n_2$

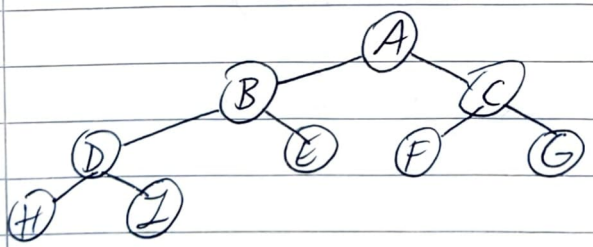
$n_0 = n_2 + 1$

- A full binary tree of depth k is one that has $(2^k - 1)$ nodes (max). Except leaves, every node has two children.

- A binary tree is said to be complete if every level (except the last) is completely filled and all nodes are as far left as possible.

— Array Representation

- parent(i) → $\lfloor i/2 \rfloor$ (if $i=1$, $i \rightarrow$ root)
left_child(i) → $[2i]$ iff $2i \leq n$ (else no left child)
right_child(i) → $[2i+1]$ iff $2i+1 \leq n$ (else no right child)



A	[1]
B	[2]
C	[3]
D	[4]
E	[5]
F	[6]
G	[7]
H	[8]
I	[9]

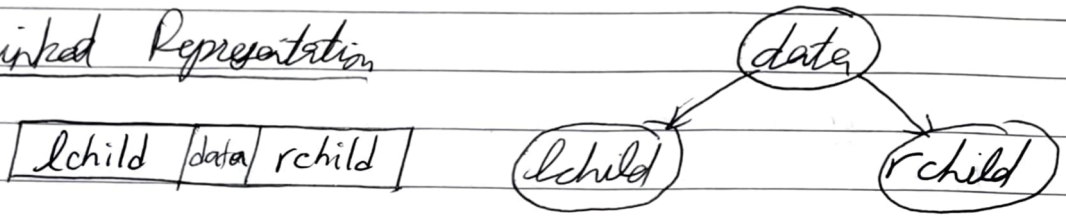
→ Pros

- Easy to understand
- Best representation of full & complete BT
- Easy to program
- Easy to move from child ⇄ parent

→ Cons

- Lots of memory wastage
- Insert & delete of nodes requires lots of data movement.

— Linked Representation



→ Pros

- Insertions & deletions can be made directly at any location w/o data movements, and flexible since system takes care of memory allocation & freeing of nodes.
- Best for any type of trees.

→ Cons

- Difficult to understand, and not easy to access nodes
- Additional memory needed for storing pointers.

```
Node createBinaryTree(int item){
    int x;
    if(item != -1){
        Node temp = getnode();
        temp->data = item;
        printf("Enter lchild of %d:", item);
        scanf("%d", &x);
        temp->lchild = createBinaryTree(x);

        printf("Enter rchild of %d:", item);
        scanf("%d", &x);
```

```
        temp -> rchild = createBinaryTree(x);
        return temp;
    }
    return NULL;
}
```

☆ Tree Traversal

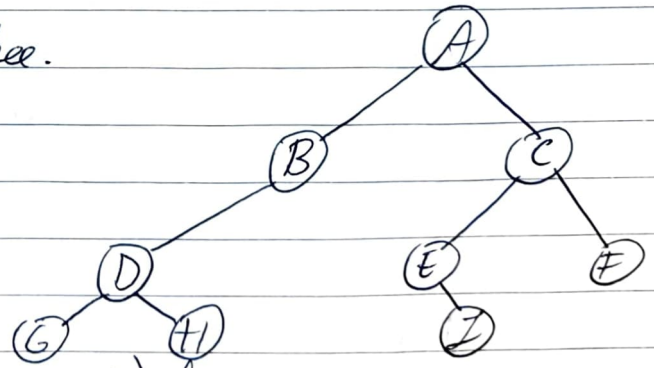
- Let L → moving left
  V → visiting the node
  R → moving right
- LVR → inorder
  LRV → postorder
  VLR → preorder.

— Inorder Traversal

- Traverse the left subtree (till leaf node) and display.
- Root (process)
- Traverse right subtree.

GDHBA

EICF

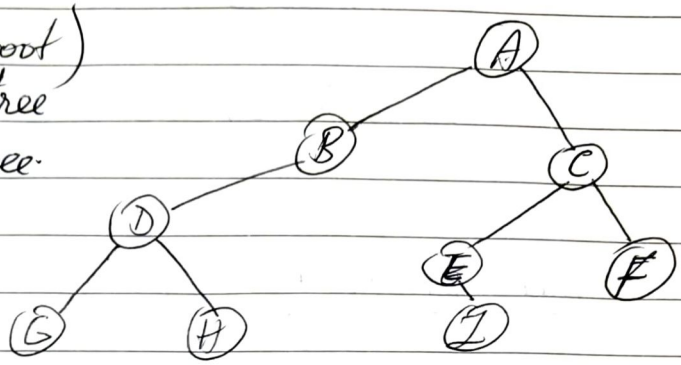



```
void inorder(Node root){
    if(root){
        inorder(root->lchild);
        printf("%d", root->data);
        inorder(root->rchild);
    }
}
```

__/__/____

Preorder Traversal

- Process the node (root)
- Traverse left subtree
- Traverse right subtree.



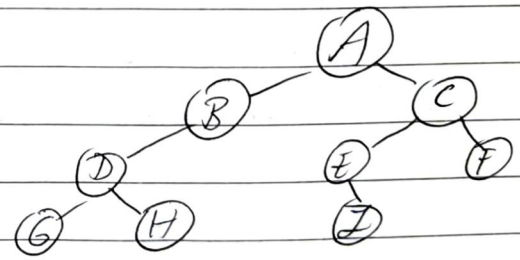
A B D G H

C E I F

```
void preorder (Node root){
    if (root) {
        printf("%d", root -> data);
        preorder(root -> lchild);
        preorder (root -> rchild);
    }
}
```

Postorder Traversal

- Traverse the left subtree
- Traverse the right subtree
- Process the node (root)



G H D B

I E F C A

```
void postorder (Node root) {
    if (root){
        postorder (root -> lchild);
        postorder (root -> rchild);
        printf("%d", root -> data);
    }
}
```

DS (Tree Contd.)

- For arithmetic expressions:

Infix : A / B * C * D + E (Inorder)

~~Prefix~~ ~~Preorder~~ :

Preorder / Prefix : + * * / ABCDE

Postorder / Postfix : AB / C * D * E +

☆ Insertion into BT

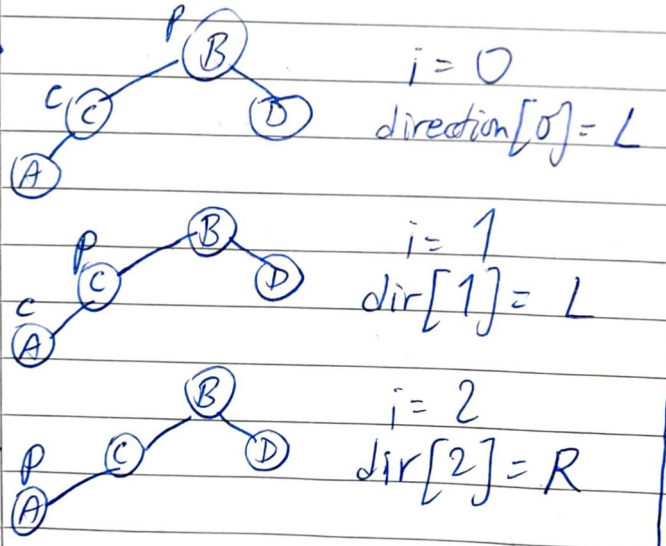
- User has to specify where to insert item, by specifying direction in the form of a string. (Ex:- L LR)

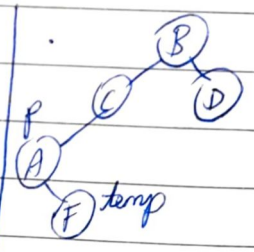
```
void insert (Node root, char direction[], int ele) {
    int i;
    Node temp, cur, parent;
    temp = getNode();
    temp -> data = ele;
    temp -> lchild = temp -> rchild = NULL;

    parent = NULL;
    cur = root;
    i = 0;
    while (cur && direction[i]) {
        parent = cur;
        if (direction[i] == 'L' || direction[i] == 'l')
            cur = cur -> lchild;
        else
            cur = cur -> rchild;
    i++;
```

node already present → wrong direction string ↗

```
if ((cur != NULL || (direction[i] != '\0')){
    printf("Insertion Not Possible \n");
    free (temp);
    return ;
}
if (direction[i-1] == 'L' || direction[i-1] == 'l')
    parent -> lchild = temp;      (new node)
else
    parent -> rchild = temp;
}
```

Ex: Direction → LLR (length = 3)
cur = root , parent = NULL





cur = NULL
i = 3
dir[i-1] = R

Searching a BT (using preorder traversal)

```
int Search (Node root, int ele){
    static int i = 0;
    if (root){
        if (root->data == ele){
            i++;
            return i;
        }
    }
```

```
        if (i==0) Search (root->lchild, ele);
        if (i==0) Search (root->rchild, ele);
    }
}
```

Copy of BT

```
Node copy (Node root){
    Node temp; temp = getNode();
    if (root == NULL)
        return NULL;
    temp->data = root->data;
    temp->lchild = copy (root->lchild);
    temp->rchild = copy (root->rchild);
    return temp;
}
```

Counting No. of Nodes in a tree

```
int count_nodes (Node root){
    if (root == NULL)
        return 0;
    return 1 + count_nodes (root->lchild) + count_nodes
                                            (root->rchild);
}
```

Counting Leaf Nodes in a tree

```
int count_leafnodes (Node root){
    if (root == NULL)
        return 0;
    if (root->lchild == NULL && root->rchild == NULL)
        return 1;
    return count_leafnodes (root->lchild) + count_leafnodes
                                            (root->rchild);
}
```

__/__/____

Height of Tree

```
int height(Node root){
    if (root == NULL)
        return 0;
    return 1 + max(height(root->lchild), height(root->rchild));
}
int max(int a, int b){
    if (a>b)
        return a;
    else
        return b;
}
```

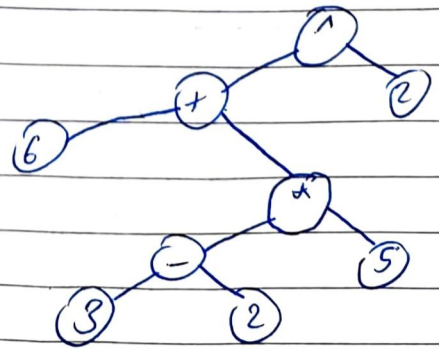
Equal Trees

```
int equal(Node root1, Node root2){
    return ((!root1 && !root2) || (root1 && root2 &&
            (root1->data == root2->data) &&
            equal(root1->lchild, root2->lchild) &&
            equal(root1->rchild, root2->rchild)));
```

Conversion of Expressions

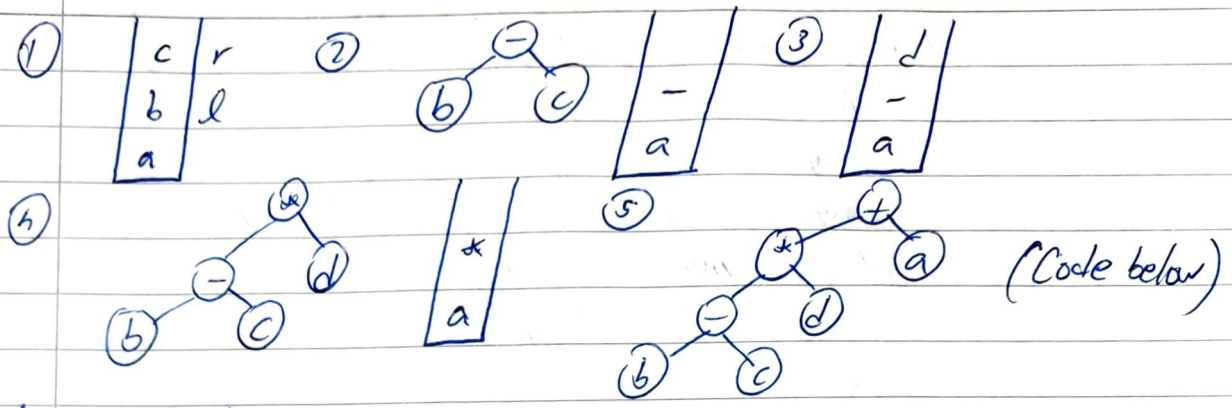
- For infix expressions, innermost parenthesis is considered first.

Ex: $((6 + (3-2)*5)\ ^\wedge\ 2)$



- For postfix expressions:

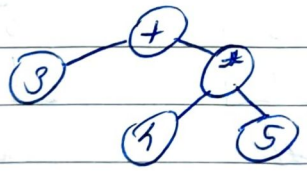
$abc - d^* +$



Evaluation

```
int eval(Node root){
    switch(root->data){
        case '+': return eval(root->lchild)+eval(
                                root->rchild);
        case '-': return eval(root->lchild)-eval(root->rchild);
        case '*': return eval(root->lchild)*eval(root->rchild);
        case '/': return eval(root->lchild)/eval(root->rchild);
        case '^': return pow(eval(root->lchild), eval(root->rchild));
        default: return (root->data - '0');
    }
}
```

Ex:



$\Rightarrow 23$

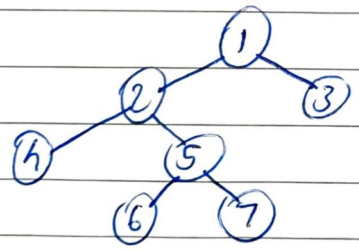
Create BT for Postfix

```
Node createTree(char postfix[]){
    Node temp;
    STACK *s, s1;
    s = &s1;
    int i=0; k=0;
    char symbol;
```

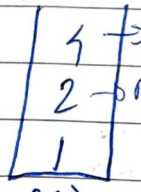
```
while((symbol = postfix[i++]) != '\0'){
    temp = getNode();
    temp -> info = symbol;
    temp -> llink = temp -> rlink = NULL;
    if (isalnum(symbol))
        push(s, temp);
    else {
        temp -> rlink = pop(s);
        temp -> llink = pop(s);
        push(s, temp);
    }
}
return (pop(s));
}
```

☆ Iterative Traversal (w/o Recursion)

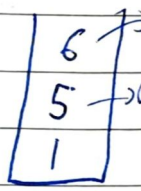
– Inorder (Left - Root - Right)



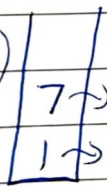
(i)



(ii)



(iii)



(left)
(right = NULL)

(NULL)

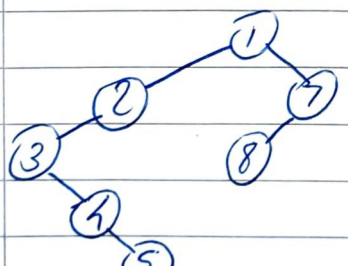
(NULL)

(4 2 6 5 7 1 3)

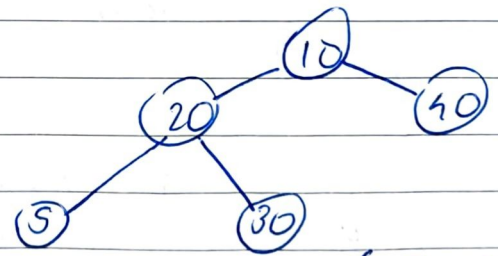
```
void iterative_inorder(Node root){
    Node cur;
    int done = false;

    Stack *s;
    s -> top = -1;
```

```
if (root == NULL) {
    printf("Empty Tree \n");
    return;
}
cur = root;
while (!done) {
    while (cur != NULL) {
        push(s, cur);
        cur = cur -> lchild;
    }
    if (!isEmpty(s)) {
        cur = pop(s);
        printf("%d", cur -> data);
        cur = cur -> rchild;
    }
    else { done = true; }
}
}
```

— Postorder (Left-Right-Root)



(6 5 4 3 2 8 7 1)



(5 30 20 40 10)

```
void postorder (Node root) {
    struct Stack {
        Node node;
        int flags;
    };
```

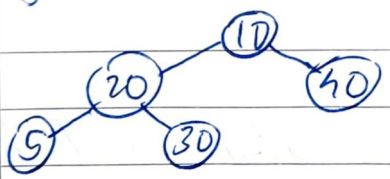
```
int done = false;
Node cur;
struct stack s[20];
int top = -1;
if (root == NULL){
    printf("Tree is empty");
    return;
}
cur = root;
while (!done){
    while (cur != NULL){
        s[++top].node = cur
        s[top].flag = 1;
        cur = cur -> lchild;
    }
    while(s[top].flag < 0){
        cur = s[top--].node;
        printf("%d", cur -> data);
        if (isEmpty(s)){
            done = true;
            return;
        }
    }
    cur = s[top].node;
    cur = cur -> rchild;
    s[top].flag = -1;
}
}
```

__/__/____

- Preorder (Root - Left - Right)

```
void preorder (Node root){
    if (root == NULL)
        return;
    Node stack[100];
    int top = -1;
    stack[++top] = root;                    (push)
    while (top >= 0){
        Node node = stack[top--];           (pop)
        printf("%d", node->data);
        if (node->rchild != NULL)
            stack[++top] = node->rchild;    (push)
        if (node->lchild != NULL)
            stack[++top] = node->lchild;    (push)
    }
}
```

Ex:-



(10 20 5 30 40)

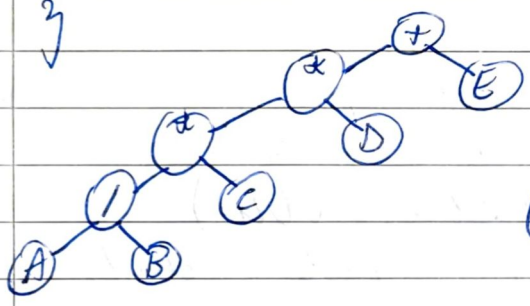
☆ Level Order Traversal

```
void LevelOrder (Node root){
    Queue *q;
    q->front = 1;
    q->rear = -1;
    if (root == NULL){
        printf("\n Empty Tree \n");
        return;
    }
    insertQ(q, root);
```

__/__/____

```
while (!isEmpty(q)){
    Node temp = deleteQ(q);
    printf("%d", temp->data);
    if (temp->lchild != NULL)
        insertQ(q, temp->lchild);
    if (temp->rchild != NULL)
        insertQ(q, temp->rchild);
}
```

Ex:-



(+ * E * D / C A B)
(goes through every level)

☆ Threaded Binary Tree

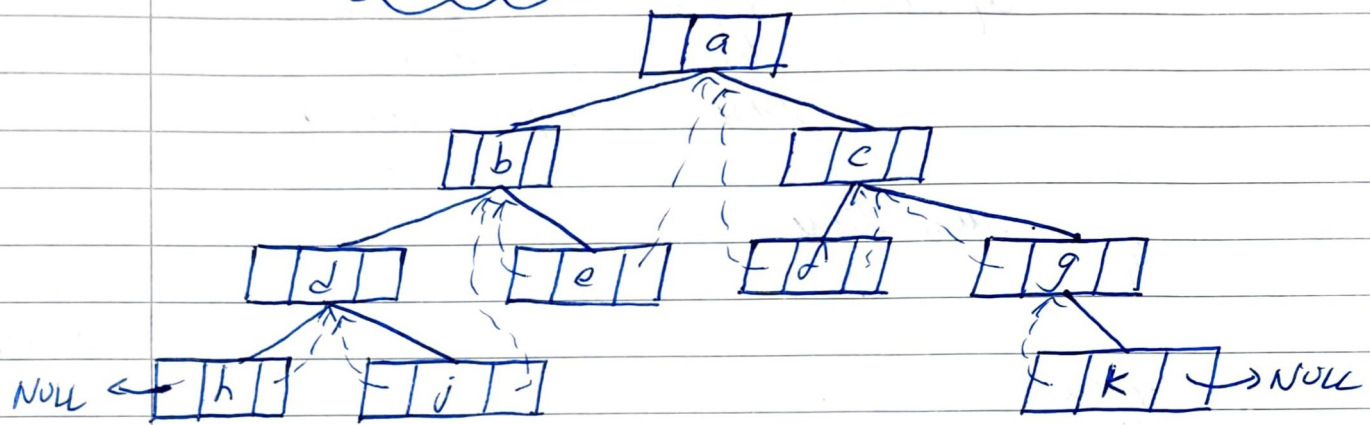
- In a binary tree with n nodes, there are (n+1) NULL links.
- Here, they are replaced by pointers, called threads, to other nodes in the tree.

```
struct threadedTree {
    int leftThread;
    Node lchild;
    char data;
    Node rchild;
    int rightThread;
};
```

leftThread	lchild	data	rchild	rightThread

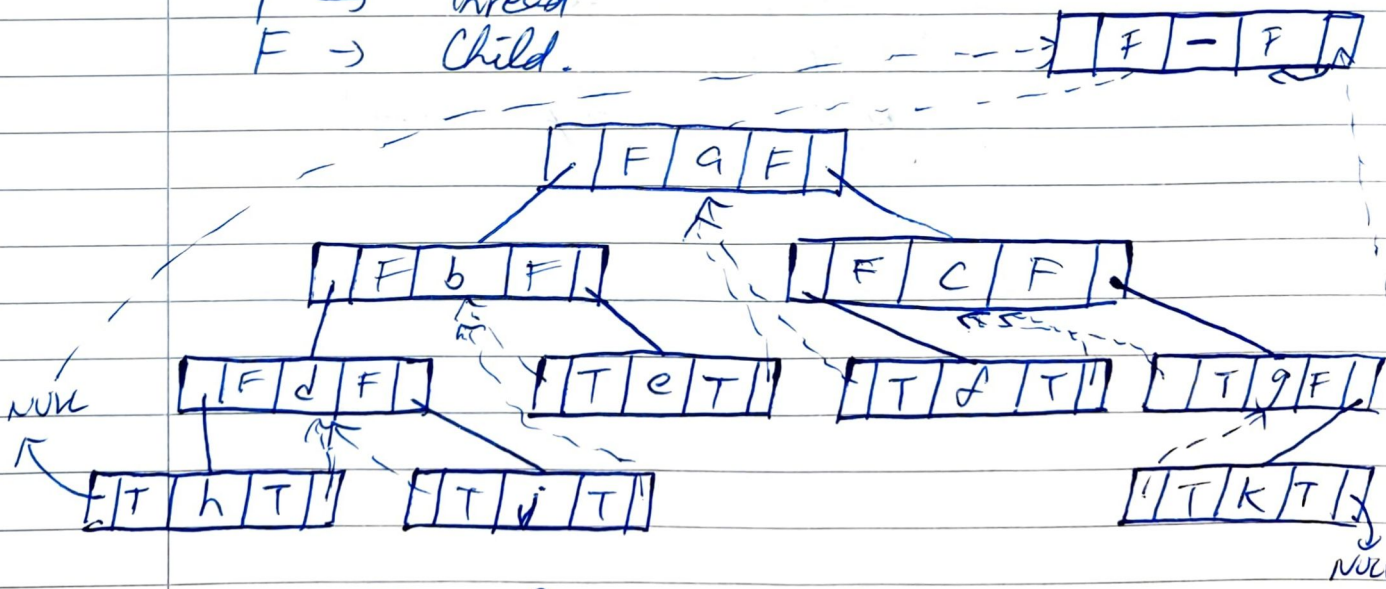
__/__/____

- Keep leftmost & rightmost NULL pointers as NULL. For the rest, Left Pointer ⇒ Inorder Predecessor
  Right Pointer ⇒ Inorder Successor



Inorder: NULL ← h d j b e a f c g k → NULL

- To distinguish between threads and children..
  T → Thread
  F → Child.



— Finding Inorder Successor w/o Stack

P TO →

```
Node inorder_successor (Node node) {
    Node temp = node -> rchild;
    if (node -> rightThread == FALSE) {
        while (temp -> leftThread == FALSE)
            temp = temp -> lchild;
    }
    return temp;
}
```

(Special case)

Inorder Traversal of TBT

(leftmost node)

(dummy node)

```
void thread_inorder (Node treehead) {
    Node temp = treehead;
    while (1) {
        temp = inorder_successor(temp);
        if (temp == treehead)
            break;
        printf("%c", temp -> data);
    }
}
```

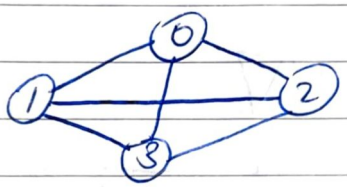
GRAPHS

✱ Graph Representation

— Adjacency Matrix

- Let $G = (V, E)$ be a graph with n vertices.
- If edge $(v_i, v_j) \in E(G) \Rightarrow$ adj_mat[i][j] = 1
  If edge $(v_i, v_j) \notin E(G) \Rightarrow$ adj_mat[i][j] = 0

Ex:



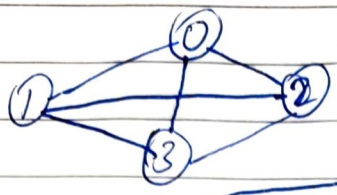
	0	1	2	3
0	0	1	1	1
1	1	0	1	1
2	1	1	0	1
3	1	1	1	0

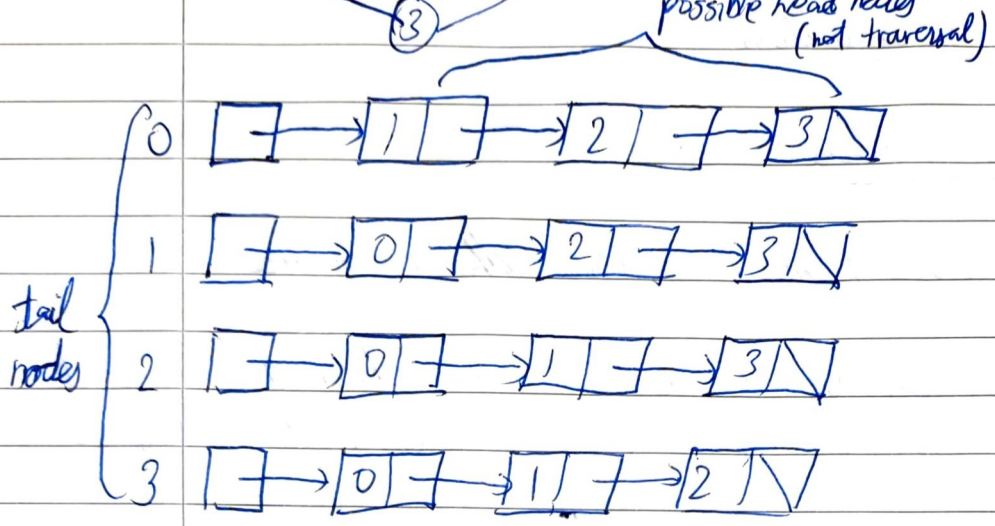
	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

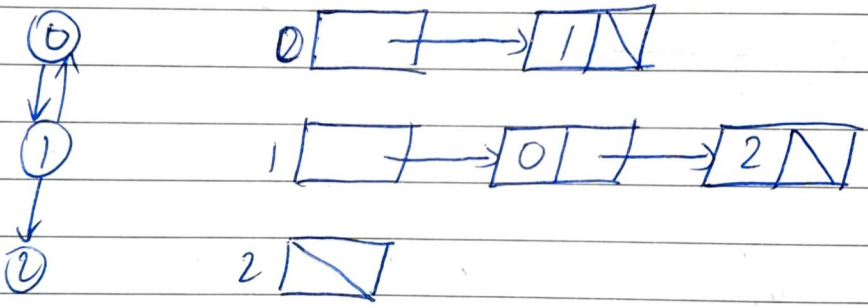
- Degree of vertex = $\sum_{j=0}^{n-1} adj\_mat[i][j]$
- For digraph, row sum = out-degree = $\sum_{j=0}^{n-1} A[i,j]$
  column sum = in-degree = $\sum_{j=0}^{n-1} A[j,i]$
- Each row in adjacency matrix is represented as an adjacency list.

PTO →

$n = 8$
$e = 7$







— Compact Representation

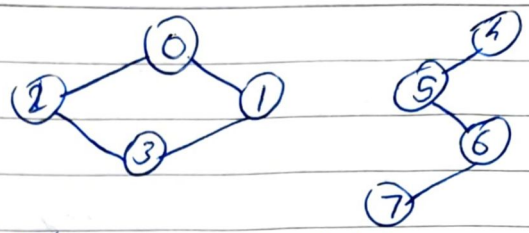
[0]	9	→ location of tail
[1]	11	
[2]	13	
[3]	15	
[4]	17	
[5]	18	
[6]	20	
[7]	22	

[8]	23	→ n+2e+1
[9]	1	} 0
[10]	2	
[11]	0	} 1
[12]	3	
[13]	0	} 2
[14]	3	
[15]	1	} 3

[16]	2	} 3
[17]	5	} 4
[18]	4	} 5
[19]	6	
[20]	5	} 6
[21]	7	
[22]	6	} 7

Here, node [0] .... node [n-1] → tail / starting point of edge
node [n] : n + 2e + 1
node [n+1] .... node [n+2e] → head node of edge.

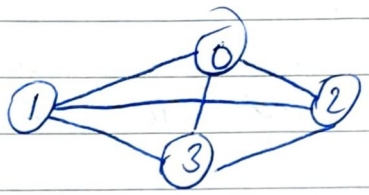
__/__/____



— Adjacency Multilist ⇒

vertex 1	vertex 2	path 1	path 2

- path 1 is chosen based on first occurence of vertex 1
  path 2 is chosen based on first occurrence of vertex 2. (in vertex 1)



N1	0	1	N2	N4
N2	0	2	N3	N4
N3	0	3		N5
N4	1	2	N5	N6
N5	1	3		N6
N6	2	3		

```
typedef struct edge *edge_pointer;
typedef struct edge {
        int vertex1, vertex2;
        edge_pointer path1, path2;
};
edge_pointer graph[MAX_VERTICES];
```

★ Graph Search Methods

- This methods starts at a given vertex v and visits/labels every vertex that is reachable from v.
- In Depth-First Search, all vertices ~~~~ reachable from the start vertex are visited.

__/__/____

- Graph is connected iff all n vertices are visited.
- Time Complexity: $O(n^2)$ when adjacency matrix used
  $O(n+e)$ when adjacency list used
  ($e \rightarrow$ no. of edges)
- On the other hand, Breadth-First Search visits start vertex and puts it in FIFO queue.
- Repeatedly remove a vertex from queue and visit its unvisited adjacent vertices and insert them into queue.
- Time Complexity: $O(n)$ if adjacency matrix used.
  $O(\text{vertex degree})$ if adjacent list used.
  $= O(n+e)$ →

__/__/____

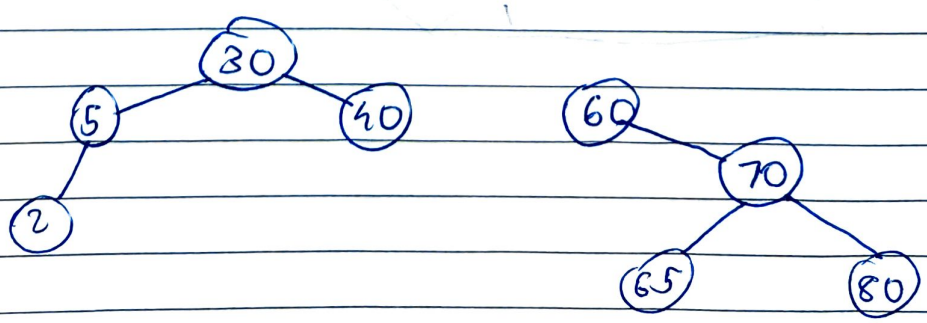
DS (Trees) (Contd.)

☆ Binary Search Tree

- It is basically a tree with no repeating elements and left subtree < root and right subtree > root.

– Create a BST

```
Node CreateBST (Node *root, int item){
    if (root == NULL){
        root = getNode();
        root->data = item;
        root->lchild = root->rchild = NULL;
        return root;
    }
    else{
        if (item < root->data)
            root->lchild = CreateBST(root->lchild, item);
        else if (item > root->data)
            root->rchild = CreateBST(root->rchild, item);
        else
            printf("Duplicates are not allowed\n");
    } return root;
}
```



- Recursive Search

```
Node search (Node *root, int key) {
    if (root == NULL)
        return NULL;
    if (key == root->data)
        return root;
    if (key < root->data)
        return search (root->lchild, key);
    return (search(root->rchild, key));
}
```

- Iterative Search

```
Node search (Node *root, int key) {
    while (root) {
        if (key == root->data)
            return root;
        if (key < root->data)
            root = root->lchild;
        else
            root = root->rchild;
    } return NULL;
}
```

Note! Maximum of BST = rightmost node in BST
Minimum of BST = leftmost node in BST

- Insertion into BST

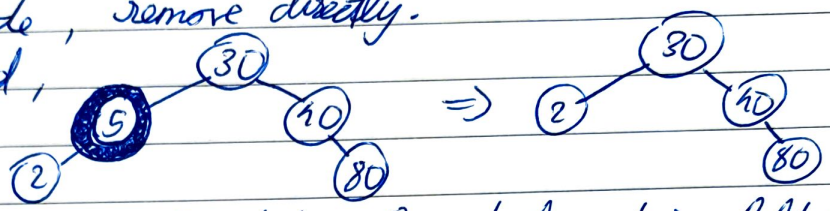
```
void insert (Node *root, int item) {
    Node temp = getNode();
    temp->data = item;
```

```
temp -> lchild = NULL;
temp -> rchild = NULL;
if (*root == NULL) {
    *root = temp;
    return;
}
Node parent, cur;
parent = NULL;
cur = *root;
while (cur) {
    parent = cur;
    if (item == cur -> data) {
        printf("Duplicates not allowed");
        free(temp);
        return;
    }
    else if (item < cur -> data)
        cur = cur -> lchild;
        else
            cur = cur -> rchild;
}
if (item < parent -> data)
    parent -> lchild = temp;
else
    parent -> rchild = temp;
return;
}
```

__/__/____

— Deleting from BST

- If leaf node, remove directly.
  If one child,



- If two children, replaced by largest element in left subtree
  (08)
  replaced by smallest element in right subtree.