```
#define swap16_const(x)  (((x) >> 8) | (((x) << 8) & 0xff00))


#define swap32(x)  (((x) >> 24) | (((x) >> 8) & 0xff00) |  \
    (((x) << 8) & 0xff0000) | (((x) << 24) & 0xff000000))

#define nswap64(val)    ((((u64bits)(nswap32(val)))<<32) |
((u64bits)(nswap32((val)>>32))))


uint64_t swap64_var(uint64_t x)
{
    return ((x >> 56) | ((x >> 40) & 0xff00) | ((x >> 24) & 0xff0000) |
        ((x >> 8) & 0xff000000) | ((x << 8) & ((uint64_t)0xff << 32)) |
        ((x << 24) & ((uint64_t)0xff << 40)) |
        ((x << 40) & ((uint64_t)0xff << 48)) | ((x << 56)));
}


/*
 * converts 32bit numeric value to string format and stores it in the
 * input buffer
 */

/* convert IP value to string format, input should be big endian format */
#define ip2s(val, sp) \
{ \
    if ((val) >= 100) \
    { \
        *(sp) = ((val) / 100) + '0'; (val) = (val) % 100; (sp)++; \
        *(sp) = ((val) / 10) + '0'; (val) = (val) % 10; (sp)++; \
    } \
    else if ((val) >= 10) \
    { \
        *(sp) = ((val) / 10) + '0'; (val) = (val) % 10; (sp)++; \
    } \
    *(sp) = (val) + '0'; (sp)++; \
}

u32bits
nsbe_ip_ntoa_buf(u32bits ip, char *buf)
{
    char *sp = buf;
    uint ipv;
    ipv = (ip & 0xFF); ip2s(ipv, sp); *sp='.'; sp++;
    ipv = ((ip>>8) & 0xFF); ip2s(ipv, sp); *sp='.'; sp++;
    ipv = ((ip>>16) & 0xFF); ip2s(ipv, sp); *sp='.'; sp++;
    ipv = ((ip>>24) & 0xFF); ip2s(ipv, sp); *sp=0;
    return (sp - buf);
}

char buf[48];
nsbe_ip_ntoa_buf(ip,buf);


int str2ip(char *str)
{
    int ip=0;
    unsigned char oct=0;
    while (*str && *str != '\n' && *str != '\r' && !NS_ISSPACE(*str) && *str !=
',' && *str != ';') {
       if (*str >= 0x30 && *str <= 0x39) {
            oct = oct *10 +(*str -0x30);
       }else if (*str == '.'){
            ip  = ip << 8;
            ip += oct; oct =0;
```

```
        }
        else return 0;
        ++str;
    }
    ip = ip << 8; ip += oct;
    return(ip);
}
```

or

```
int str2ip(char *str)
{
    int ip=0;
    unsigned char val=0;
    while (*str != '\0')  // any string should end with NULL ch.
    {
        if (*str >= '0' && *str <= '9') {
            val = val *10 +(*str - '0');
        }else if (*str == '.'){
            if(val > 255)  return -1;
            ip  = ip << 8;
            ip += val;
            val = 0;
            count++;
            if(count > 3)  return -1;
        } else
            return -1;  // only digits or '.' is acceptable
        ++str;
    }
    if(count != 3)  return -1;
    if(val > 255)   return -1;
    ip = ip << 8;
    ip += val;
    return(ip);
}
```

ip = a.b.c.d = 4 bytes

$(a \times 256^3) + (b \times 256^2) + (c \times 256) + d$

16777216 + 65536 + 256 + 1

every << 8 multiplies by $256 = 2^8$

count the no of bits

1) Build the lookup table

ex:- calculate for 4 byte value.

```
uchar lt[256];
for(int i=0; i<256; i++)
    lt[i] = (i&1) + lt[i/2];
```

→ find for variable x

```
uchar *p = (uchar *)&x;
result = lt[p[0]] + lt[p[1]] +
         lt[p[2]] + lt[p[3]];
```

a.b.c.d.

no. of 1s + no. of 1s + no. of 1s + no. of 1s

1) convert ~~integer~~ decimal to binary.

```
void dec_2_bin(int x)
{  int mask = 1<<((sizeof(int)*8)-1);
   printf("Binary =");
   for(; mask; mask = mask>>1)
       putchar( (x & mask)? '1' : '0');
}
```

getchar() → gets a character from screen
putchar() → puts a character on screen

mask = 100000...

---

2) convert decimal to hex.

→ depends if it's Big E or little E

```
char * dec2hex(int x)
{
static char hex[] = "0123456789ABCDEF";
  static char val[9];
  val[8] = '\0';
  for(int i=0; i<8; i++)
  {   val[7-i] = hex[x & 0xF];
      x = x>>4;
  }
  return val;
}
```

```
int main()  int x;
{ printf("Enter no: \n");
  scanf("%d", &x);
  printf("Hex = %s\n",
        dec2hex(x));
  return 0;
}
```

---

3) Big endian or little endian

```
#define IS_BIG_ENDIAN (*(unsigned short *) "\0x\ff" < 0x100)
```

should be char
or char?

---

3 ways to access 2-d arrays →

1) using single pointer

```
int *a; int row, int col.
a = (int*)malloc(row * col * sizeof(int));
for(int i=0; i<row; i++)
   for(int j=0; j<col; j++)
      *(a+(i*col)+j) = value;
```

2) using array of pointers

```
int *a[row]; int col.
for(int i=0; i<row; i++)
  a[i] = (int *)malloc(col * sizeof(int));
for(int i=0; i<row; i++)
   for(int j=0; j<col; j++)
      a[i][j] = value
```

3) using double pointers:

```
int **a; int row; int col
a = (int **)malloc(row *
                sizeof(int*));
for(int i=0; i<row; i++)
a[i] = (int *)malloc(col *sizeof(int))
for(i=0; i<row; i++)
for(int j=0; j<col; j++)
    a[i][j] = value.
```

STRING OPERATIONS

string = null terminated character array whose memory comes from data segment

1) strlen :-

```
int strlen (const char * str)        → normally std api returns size_t.
{   if (!str) return -1;
    const char *ptr = str;
    while (*ptr++);                  → will stop when *ptr contains null character
    return (ptr - str - 1);          → size - null character
}
```

→ Note - its not safe since we may hit bus error if string is malformed i.e if there is no null termination. Hence take some max upper value.

2) strcpy :-

```
void strcpy (const char *src, char *dst)
{   while (*dst++ = *src++)
    ;
    return;
}
```

→ if (!src || !dst) return;
if (src == dst) return;

3) strncpy :-

```
void strncpy (char * dst, const char *src, int len)
{   if (dst == src) return;
    if (!dst || !src) return;
    if ((src < dst) && (dst < src+len)) {
        while (len--)
            *(dst + len) = *(src+len);
    } else {
        while (len--)
            *dst++ = *src++;
    }
}
```

doubt is the cond wrong - to check for overlap.
if (dst > (src + len))
→ overlap ? → to still copy we do it but here src gets overwritten ∴ don't use const

4) checking 2 strings are anagrams of each other. e.g. army ⟺ mary.
take ascii or extended ascii x[128] or x[256].

```
bodean anagram (const char *a, const char *b) char a[];
{   if (!a || !b) return false;
    if (strlen(a) != strlen(b))
        return false;
```

```
while (*a) x[*a++] += 1;
while (*b) x[*b++] -= 1;
for (int i=0; i<128 or 256; i++)
{   if (x[i]) return false;
}.
```

x[*a] += 1; a++;
x[*b] -= 1; b++;

another method is to sort both strings as per their ascii value & check.

5) check if str1 is a rotation of str2.

Ex: str2 = "abcde", str1 = "deabc".

here concatenate str1 with str1 & search for substring str2 in it.

deabc + deabc = deabcdeabc
(str1)

6) Remove duplicate characters

- we cannot from a string since we cannot modify a string. copy to an array & then remove from array.
- consider only ascii → arr[128]. str is the array string.

```
void rem_dup (char *str)
{   char *trav = str;
    do{
        if(arr[*str]) continue;
        arr[*str] = 1;
        if(trav != str)
            *trav++ = *str;
    }while(*str++);
}
```

7) asci to integer (atoi);

```
int atoi(const char *str) int ret=0;
{   boolean neg = FALSE;   if(!str || !*str) return ret;
    if(*str == '-')
    {   *str++;
        neg = TRUE;
    }
    while(*str >= '0' && *str <= '9')
    {   ret = (ret*10) + *str++ - '0';
    }
    if(neg) return (-ret);
    else
        return(ret);
}
```

for ascii to integer - have a running value.
1) mult this value by 10 & then add the value of the character.

Bitwise

① swap bits at i & j in 64bit integer.
- if bits differ then only swap. Swapping means flipping them

```
long swap(int x, int i, int j)
{
    if (((x >> i) & 1) != ((x >> j) & 1))
    {
        x ^= (1L << i) | (1L << j);
    }
}
```

x = 01100001

say i=3, j=7.
take 8bits
00001000 i<<3.
10000000 j<<7
10001000.
^01100001
11101001

② find the closest integer with the same weight (i.e same no of bits)

```
int func(int x)
{
    for(int i=0; i<63; i++)   → why is it 63 & not 32 ?
    {
        if (((x >> i) & 1) ^ ((x >> (i+1)) & 1))
        {
            x ^= (1UL << i) | (1UL << (i+1));   → swap bits i & i+1
        }
    return (x);
```

③ Reverse digits. 8x:- for 42 ans is 24, -314 → -413.

```
int reverse(int x)
{
    bool val = 0;   int res = 0;
    if (x < 0)  val = 1;
    while (x)
    {
        res = res * 10 + x % 10;
        x = x / 10;
    }
    if (val)
        return (-res);
    else return (res);
```

x = -314.

1) res = 0 * 10 + 314 % 10 = 4.
   x = 314/10 = 31.

2) res = 4 * 10 + 31 % 10 = 40 + 1
   x = 31/10 = 3.

3) res = (41 * 10) + 3 % 10 = 410 + 3
   x = 3/10 = 0.

∴ res = 413.

④ given a no say if it is a palindrome or not.
- use (3) - reverse the no & check if the reversed no is same as the given no.

Bit fields in structure - allows prgmer to divide a ~~byte/2bytes/4bytes~~ byte into named block of bits - can individually access the bit fields rather than doing bitwise operations on them.

```
struct bitfield {
    unsigned x : 1;
    unsigned y : 2;
    unsigned z : 3;
};
```

sizeof(struct bitfield) = 1 (byte itself.)
- doesn't save memory ~~+ bits padding.~~
→ padding.

access: bitfield a;
a.x = 1;

5) power of 2: if((x & (x-1)) == 0) → then x is a pow of 2

6) clear last set bit: x & (x-1). (from ~~left~~ right side)

7) clear last ~~set~~ bit: x & ~(x-1) (from ~~right~~ left side) × ???
clears last bit

8) count no of bits set:
```
for (x; x; x = x & (x-1))
{  count++;
}
```

9) – without branching. → track it very carefully.
```
int count (int x)
{
  x = (x & 0x55555555) + ((x>>1) & 0x55555555);   ⇒ (x & 01010101)
  x = (x & 0x33333333) + ((x>>2) & 0x33333333);   ⇒ (x & 00110011)
  x = (x & 0x0f0f0f0f) + ((x>>4) & 0x0f0f0f0f);   ⇒ (x & 00000111)
  x = (x & 0x00ff00ff) + ((x>>8) & 0x00ff00ff);   ⇒ (x & 0000 1111 / 1byte  1111 / 1byte)
  x = (x & 0x0000ffff) + ((x>>16) & 0x0000ffff);  ⇒ (x & 0000 1111 / 2bytes  1111 / 2bytes)
```

9) isolate highest set bit: x & ~(x>>1).

10) Reverse an integer.
```
int reverse (int x)
{  x = ((x & 0xffff0000) >> 16) | ((x & 0x0000ffff) << 16);
  x = (((x & 0xff00ff00) >> 8) | ((x & 0x00ff00ff) << 8);
  x = (((x & 0xf0f0f0f0) >> 4) | ((x & 0x0f0f0f0f) << 4);
  x = (((x & 0xcccccccc) >> 2) | ((x & 0x33333333) << 2);
  x = (((x & 0xaaaaaaaa) >> 1) | ((x & 0x55555555) << 1);
```

11) ~~swap~~ #define swap16(x) ((x>>8) | (x<<8))

12) #define swap32(x) ((x>>24) | ((x>>8) & 0xff00) | ((x<<8) & 0xff0000) | (x<<24)).
imp imp.

13) #define swap64(x) ((swap32(x) << 32) | (swap32(x) >> 32))

Note:- swapping is not same as reversing in swap a.b.c.d ⇒ d.c.b.a.
but bits within each octet are retained as is.

Arrays =>

1. find max sub sequence in an array.

- 3 possibilities - either the max sub seq can be in left ½, or right ½ or ~~the 2nd~~ ½ in left half & right half both

LBS = left border sum
RBS = right border sum
LS = left sum
RS = right sum

```
static MSS (int a[], int left, int Right)
{   int maxLBS, maxRBS, maxLS, maxRS;
    int LBS, RBS;
    int centre, i;
    if (left == right)
    {   if (A[left] > 0)
            return (A[left]);
        else
            return 0;
    }
    centre = (left + right)/2;
    maxLS
    ~~maxLBS~~ = mss (a, left, centre);
    ~~maxRBS~~ = mss (a, centre+1, right);
    maxRS
    ~~for (i=centre~~  maxLBS = LBS = 0; -> maxLBS should be initialised.
    for (i = centre; i >= centre; i--)
    {       LBS += a[i];
        if (LBS > maxLBS)
            maxLBS = LBS;
    }
    maxRBS = RBS = 0;
    for (i = centre+1; i <= right; i++)
    {       RBS += a[i];
        if (RBS > maxRBS)
            maxRBS = RBS
    }
    return max3 (maxLS, maxRS, (maxLBS + maxRBS));
}
```

→ calling ~~fn~~ MSS (a, 0, n-1) where n = no of elements.

2-D array allocation.

1) using a single pointer:

`int *arr;` int rows, int cols.

arr = (int *) malloc (rows * cols * sizeof (int));

access for (int i=0; i<rows; i++)
for (int j=0; j<cols; j++)
*(arr + (i * cols) + j) =

Note: have to make linear access - skip across an entire row to reach that column.

2) using an array of pointers.

`int * arr[rows];` int cols;

for (int i=0; i<rows; i++)
arr[i] = (int *) malloc (cols * sizeof (int));

access:- for (i=0; i<rows; i++)
for (j=0; j<cols; j++)
arr[i][j] =

3) using double pointers:-

`int **arr;` int rows, int cols;

arr = (int **) malloc (rows * sizeof (int *));
for (int i=0; i<rows; i++)
arr[i] = (int *) malloc (cols * sizeof (int));

access:- for (i=0; i<rows; i++)
for (j=0; j<cols; j++)
arr[i][j] =

1) find sq root of an no :- 2)- fastest way is to pre-compute & store in a table if repeated calls are made.

$\sqrt{32\text{bit no}} = 16\text{bit no}$ ∴ 2^{16} array elements required.

2) use binary search.

```
int sqrt (int x)                    → end cannot be greater than x
{   int beg = 0, end = 2^16 - 1; int mid;
  while (beg + 1 < end)
  {  mid = beg + (end - beg) / 2 ;
     int sq = mid * mid;
     if (x == sq) return sq;
     else if (~~mid~~ sq > x) {
              end = mid;
     }else{
          beg = mid;
     }
  }
  return ~~beg~~ beg;
}
```

---

(2) Tail recursion :-

- optimised way. if any recursive fn has its last statement in the routine as a recursive call - called tail recursion.

- compiler can do optimisation since it does not need to store the current stack frame values since the last call is recursion. - there are no further statements after this.

Ex:- non-tail recursive.

```
int fact (int n)
{   if (n==0) return 1;
    return (n * fact (n-1));
}
```

the compiler has to store stack frame of current recursive call since it has to do multiplication.

tail recursive

```
int fact (int n, int result)
{  if (n==0) return 1;
   return fact (n-1, n* ~~a~~ res);
}
```

→ call it ~~as~~ via

```
fact (5, 1);
```

---

3) BST - lookup of element - always do iterative lookup since recursion eats up stack.

```
NODE *  ~~int~~ BST_lookup (int x, NODE *root)
{  while (root)
   {  if (root->key == x)
      ~~{~~     return root;
      else if (root->key < x) {
            root = root->left;
      } else
            root = root->right;
   }
}
```

4) Quick sort & merge sort both sort in $O(n \log n)$ but Quicksort is better since it sorts in place & unlike merge sort does not require temp. array/buffer.

5)

graphs - set of nodes = vertices.
set of connections = edges.

memory representation
- adj lists - each vertex is represented by a linked list containing all vertices directly connected to it
- adj matrix - $v \times v$ matrix where $v[i,j]$ = represents a path/edge from i to j.

DFS - easy to handle - in recursive DFS the path with all vertices would be present in the call stack. So compiler implicitly handles the stack.

BFS - more difficult since we have to build the queue..

Extra info:- DFS - helps to detect cycles.
BFS - helps to compute distance from a given vertex to the starting vertex.

Adj matrix:-

```
struct Graph {
    int v; // no of vertices
    int E; // no of edges - required??
    int **adj;
};
```

Allocation:->

```
struct Graph *G;
G = (struct Graph *) malloc (sizeof (struct Graph));
printf ("Enter no of vertices:");
scanf ("%d", &G->v);
G->adj = (int **) malloc (G->v * sizeof (int *));
for (int i=0; i<G->v; i++)
    G->adj[i] = (int *) malloc (G->v * sizeof (int));
printf ("Enter the edges");
for (i=0; i<G->E; i++)
    {
    printf ("Reading the edge (u,v):");
    scanf ("%d, %d", u, v)
    } adj[u][v] = 1;
```

printf ("Enter the no of vertices");
scanf ("%d", &G->E);

DFS traversal :-

- mark all vertices as unvisited. Starting at vertex 'u', traverse all edges emanating from 'u' to other vertices & the moment we encounter a vertex which is visited, we come back (backtrack) to the started vertex. If it results in unvisited vertex, then start processing from that vertex as the start vertex.

```
int main()
{   → Allocate adj matrix with
    for(int i=0; i< G->V; i++)
        visited[i] = 0;
    for (i=0; i< G->V; i++)
    {   if(!visited[i])
            DFS(G, i);
    }
} return 0;

void DFS(struct Graph * G, int V)
{   visited[v] = 1;   printf("%d", v);
    for(int i=0; i< G->V; i++)
    {   if(!visited[i] && G->adj[v][i])
            DFS(G, i);
    }
}
```

```
struct Graph {
    int V;
    int E;
    int **adj;
};
int **adj;   int visited[v];
int main()
{   initialise adj. v = no of vertices.
    DFS(0);   → populate each [i][j]
}                  as
void DFS(int i)
{   if(visited[i]) printf("vertex= i")
    visited[i] = 1
    for(int j=0; j< n; j++)
    {   if(!visited[j] && adj[i][j])
            DFS(j);
    }
}
```

BFS traversal :

- similar to level order traversal - starts at vertex 0 & visits all vertices of that level before moving to next level

```
void BFS(struct Graph * G, int v)
{   int u;
    struct Queue * Q = created(G->V);
    enqueue(Q, v);
    while(!isempty(Q))
    {   u = dequeue(Q);
        visited[u] = 1;
        printf("%d", u);
        for(int i=0; i< G->V; i++)
        {   if(!visited[i] && G->adj[u][i])
                enqueue(Q, i);
        }
    }   deletequeue(Q);
}
```

```
int main()
{   - initialise
    for(int i=0; i< G->V; i++)
        visited[i] = 0;
    for(int i=0; i< G->V; i++)
        BFS(G, i);
    return;
}
```

1) find element in BT.

```
NODE *find (NODE *root, int elem)
{   NODE *temp = null;
    if(!root) return null;
    else{
        if(root->data == elem)
            return (root);
        else{
            temp = find (root->left, elem);
            if(temp)
                return(temp);
            else
                return(find (root->right, elem));
        }
    }
    return null;
}
```

2) find max in BT.

```
#define int_min (-2^16-1)
NODE *findmax (NODE *root)
{   int max = int_min;
    if(!root)
    { int root_val = root->data;
      int left = findmax (root->left);
      int right = findmax (root->right);
      if( left > right)
      {  max = left;
      } else
            max = right;
      if (root_val > max)
            max = root_val;
    } return max;
}
```

3) Insert element into BT.

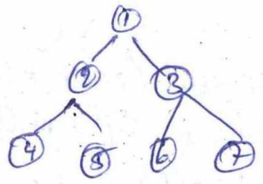
this= traverse in level order & insert under the node whose left or right child is null. - ie insert anywhere below it.

```
void insert (NODE *root, int elem)
{   NODE *temp, *node;
    node = createnew (elem);   → malloc & init left & right ptrs to null.
    struct queue Q = createQ (capa);
    if(!root) return (node);
    else  enque (Q, root);
    while(!isempty (Q))
    {
        temp = dequeue (Q);
        if ( temp->left)
            enqueue (Q, temp->left);
        else {
            temp->left = node;
            deleteQ (Q);
            return;
        }
        if(temp->right)
            enqueue (Q, temp->right);
        else {
            temp->right = node;
            deleteQ (Q);
        }   } return;
}                                  } return;
```

4) size of BT :-

```
int size (NODE *root)
{  if(!root) return 0;
   else
   return (size(root->left) +
           size(root->right) + 1);
}
```

5) Print level order traversal in reverse.



Print = 4, 5, 6, 7, 2, 3, 1

```
void print (NODE *root)
{   if(!root) return;
    Q = createQ();
    enqueue (Q, root);
    while(!isemptyQ(Q))
    {   NODE *temp = dequeue(Q);
        if(temp->left)
            enqueue(Q, temp->left);
        if(temp->right)
            enqueue(Q, temp->right);
        Push(S, temp);
    }
    while(!isemptystack(S))
        printf("%d", pop(S));
}
```

```
S = createstack();
```

---

6) find the deepest node in a BT.

Ans: Simple. In level order traversal, the last 'temp' node you have dequeued is the deepest node. There can be more nodes also - same level.

7) Delete element from BT.

→ Lookup the node to be deleted.
→ find the deepest node of tree

} Replace the data from deepest node. Delete the deepest node

8) find the diameter of a BT.

```
int dia(NODE *root, int *ptr)
{   if(!root) return 0;
    int lft = dia(root->left, ptr);
    int right = dia(root->right, ptr)
    if(left + right > *ptr)
        *ptr = left + right;
    return (max(left, right) + 1);
}
```

9) find level with max sum.

```
int findlevel (NODE *root)
{   NODE *temp;
    int level, maxlevel = 0;
    int cursum, maxsum = 0;
    struct queue Q = createQ();
    if(!root) return 0;
    enque(Q, root);
    enqueue(Q, NULL);
while(!isempty(Q))
{   temp = dequeue(Q);
    if(temp == null)
    {   if(cursum > maxsum){
            maxsum = cursum;
            maxlevel = level;
        }
        cursum = 0;
    if(!isempty(Q)
        enqueue(Q, NULL);
    level++;
    } else {  cursum += temp->data;
        if(temp->left)
            enqueue(Q, temp->left);
        if(temp->right)
            enqueue(Q, temp->right);
    }
return maxlevel;
}
```

10) Search sub string in string.

```
void search(char *txt, char *pat)
{
    int n = strlen(txt);
    int m = strlen(pat);
    for(int i=0; i<=n-m; i++)
    {
        for(int j=0; j<m; j++)
        {
            if(txt[i+j] != pat[j])
                break;
            if(j==m)   → if(j==(m-1))
            { printf("Pat found at %d", i); }
        }
    }
}
```

Boyer Moore

MAX_CHAR = 256;

```
int BC[MAX_CHAR];
void fillBC(char *pat, int m)
{
    for(int i=0; i<MAX_CHAR; i++)
        BC[i] = -1;
    for(i=0; i<m; i++)
        BC[(int) pat[i]] = i;
}
int max(int a, int b)
{ Return ((a>b)? a: b); }

void search(char *txt, char *pat)
{
    int n = strlen(txt);
    int m = strlen(pat);
    fillBC(pat, m);
    int s = 0;
    while(s<=(n-m))
    {
        int j = m-1;
        while(j>=0 && pat[j]==txt[s+j])
            j--;
        if(j<0)
        {
            printf("Pat found at %d", s);
            s += (s+m<n)? m-BC[txt[s+m]] : 1;
                ↳ move over to next pos after finding one instance of pattern
        }
        else
        {
            s += max(1, j-BC[txt[s+j]]);
        }
    }
```

8) reverse the characters of a string.

```
void reverse (char *str)
{
    if (!str || !*str) return;
    char *end = str;
    while (*end)
    {
        end++;
    }
    end--;          // set 1 char back since last char is null.
    while (str < end)
    {
        char temp = *str;
        *str++ = *end;
        *end-- = temp;
    }
}
```

9) Reverse a sentence. - take care of extra white spaces.

```
void reverseS(char *str)
{
    char *ptr = str;
    char *wb = NULL; //word begin.
    while (*ptr)
    {
        if (!wb && *ptr != ' ')
        {
            wb = ptr;
        }
        if (wb && ((*(ptr+1) == ' ') || (*(ptr+1) == '\0')))
        {
            reverse (wb, ptr);
            wb = NULL;
        }
        ptr++;
    }
    reverse (str, ptr-1);
}
```

```
void reverse (char *beg, char *end)
{
    char ch;
    while (beg < end)
    {
        ch = *beg;
        *beg++ = *end;
        *end-- = ch;
    }
}
```

→ or should it be
`if (wb && (*ptr != ' ' && *(ptr+1) != ' ' && *(ptr+1) != '\0'))`

→ Logic:- initialise wb to NULL → just a tracking field.

if (wb is NULL & 1st char is not a space)
{
    initialise wb to 1st character.
}
if (wb is non-null && 2nd char is space or end of string)
{
    reverse the word → this means end of word
    & initialise wb to null.
}

10) Print path from root to leaf nodes. (

```
void PrintPathRecurse (NODE *root, int p[], int len)
{   if(!root) return;
    p[len++] = root->data;
    if(!root->left && !root->right)
        printpath (p, len);
    else {
        printpathRecurse (root->left, p, len);
        printpathRecurse (root->right, p, len);
    }
}
```

```
void printpath (int p[], int len)
{ for(int i=0; i< len; i++)
    printf("%d", p[i]);
}
```

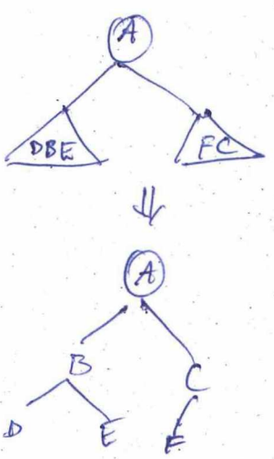
11) Construct a BT from given inorder & pre-order.

in pre-order the 1st element would be the root. Search the root element in inorder & elements to left of root = left subtree.

Right of root is right subtree.

inorder: DBEAFC.

preorder: ABDECF




```
struct NODE * BuildT (int in[], int pre[], int
                      int start, int end)
{   static int index1 = 0;
    NODE * newNd = createnode(pre[index1]);
    index1++;
    if(start == end)
        Return (newNd);
    int index2 = search(in, start, end, newNd->data);
    newNd->left = BuildT(in, pre, start, index2-1);
    newNd->right = BuildT (in, pre, index2+1, end);
    Return (newNd);
}
```