24/2/23

1) BFS

```
> #include <stdio.h>
#include <stdlib.h>
# define  SIZE 40

 struct queue {
       int items[size];
       int front;
       int rear;
  };

 struct   queue  *createQueue();
void   enqueue (struct  queue *q , int);
int   dequeue (struct   queue  *q);
 void  display (struct   queue  *q);
 int    isEmpty (struct   queue  *q);
void  printQueue (struct  queue *q);

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

struct   node  *createNode(int);

 struct   Graph {
       int    numVertices;
       struct    node  **adjLists;
         int  *visited;
  };
```

26/2/24

1) BFS

```
-> #include <stdio.h>
   #include <stdlib.h>
   # define  SIZE 40

   struct queue {
          int items[size];
          int front;
          int rear;
    };

   struct   queue   *createQueue();
   void   enqueue (struct  queue  *q, int);
   int   dequeue (struct   queue  *q);
   void   display (struct   queue  *q);
   int   isEmpty (struct   queue  *q);
   void   printQueue (struct  queue  *q);

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

   struct   node   *createNode (int);

   struct   ~~g~~ Graph {
          int   numVertices;
          struct   node   **adjLists;
             int   *visited;
     };
```

```
void  bfs ( struct  Graph  *graph , int  startVertex)
{
   struct   queue  *q  =  createQueue ();
     graph -> visited [startVertex] = 1;
     enqueue (q, startVertex);

   while (!isEmpty(q)) {
         printQueue (q);
     int  currentVertex  =  dequeue (q);
     printf (" Visited  %d\n", currentVertex);

      struct  node  *temp  =  graph -> adjLists [currentVertex]

            while (temp) {

     int  adjVertex  =  temp -> vertex;

     if (graph -> visited [adjVertex] == 0) {
            graph -> visited [adjVertex] = 1
             enqueue (q, adjVertex);
        }

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

```
      struct  node  *CreateNode (int v) {
            struct  node  * newnode ;
         newnode = (struct node *) malloc (sizeof (struct node));
        newNode  -> vertex = v;
        newNode  -> next  = NULL;
            ~~reeb~~  return newNode;
      }

      struct   Graph  *createGraph (int vertices) {
         struct  Graph  *graph ;
         graph  = (struct Graph *) malloc (sizeof (struct Graph));
        graph ->numVertices  =   vertices ;
        graph -> adjList  =  malloc (vertices *(sizeof (int)));
         int  i;
        for (i=0 ; i< vertices ; i++) {
            graph -> adjLists [i] = NULL;
            graph -> visited [i] = 0;
        }

        return  graph;
   }


   void  addEdge (struct Graph *graph , int src,
      int dest ) {

      struct   node    * ~~CreateQueue (~~ newNode ;
            newnode =  createNode (dest);
  newnode -> next  =  graph -> adjLists [src] ;
  graph -> adjList [src] =  newnode ;

  (newNode  =  CreateNode (src);
  newNode -> next  = graph -> adjList [dest];
```

```
        graph->adj[dest] = newnode; }

    struct   queue  *createQueue(){
        struct queue *q = malloc(sizeof(struct queue));
            q->front =-1 ;
            q->rear =-1 ;
            ~~requ~~ return q;
        }

    int isEmpty(struct queue *q){
            if (q->rear ==-1){
                return 1;
            else
                return 0;
      }

    void enqueue(struct queue  *q, int value){
            if(q->rear == Size -1)
              printf("Queue is Full);
              else{
                  if (q->front ==-1)
                      q->front = 0;
                  q->rear++ ;
                  q->item[q->rear] = value;
            }
        }
```

```
int dequeue(struct queue *q) {
    int item;
    if (isEmpty(q)) {
        printf("Queue is empty");
        item = -1;
        }

    else {
    q item = queue -> items[q->front];
        q-> front++;
        if (q->front > q->rear) {
        printf("Resetting queue");
          q->front = q->rear = -1;
    }
  }
  return item;
}


void printQueue(struct queue *q) {
    int i = q-> front;
  if (isEmpty (q)) {
      printf ("Queue is Empty");
  } else {
    printf (" Queue contains");
     for (int = q->front; i < q->rear+1; i++) {
  printf ("%d" q->items[i]);
 }
 }
};
```

```
int main(){
    struct Graph *grap = createGraph(6);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 4);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);

    bfs(graph, 0);
    return 0;
}
```

Output

```
Queue contains
0 Resetting queue Visited 0

Queue contains
2 1 Visited 2

Queue contains
1 4 visited 1

Queue contains
4 3 visited 4

Queue contains
3 Resetting queue Visited 3
```

DFS

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

void   DFS ( struct Graph * graph , int vertex)
    {
     struct   node  * adjlist = graph -> adjlist [vertex];
    struct   node  * temp = adjlist ;

      graph -> visited [vertex] = 1;
    printf ("Visited  %d", vertex);

     while ( temp != NULL ) {
    int     connectedVertex = temp -> vertex ;
    if ( graph -> visited [connectedVertex] == 0) {
             DFS( graph , connectedVertex );
     }
     temp = temp -> next ;
   }
  }


void  printGraph ( struct Graph * graph) {
        int  v ;
  for ( v = 0  ; v < graphVertices ; v++ ) {
           struct  node  * temp =  graph -> adjList[v] ;
    printf (" Adjacency  list of   vertex  %d ", v);
           while( temp) {
         printf (" %d ", temp -> vertex);
         temp = temp -> next;
    }  printf (" \n"); }}
```

```
int main(){
  struct Graph * graph(4);
     addEdge ( graph ,0, 1);
     addEdge ( graph ,0,2);
     addEdge ( graph, 1, 2 );
     addEdge (graph ,2,3);

    printGraph(graph);

     DFS(graph ,2);
     return 0;
}
```

<u>output :</u>

```
Adjacency List of vertex 0
 2 → 1

Adjacency list of vertex 1
 2 → 0

Adjacency list of vertex 2
 3 → 1 → 0

Adjacency list of vertex 3
 2
Visited 2
Visited 3
Visited 1
Visited 0
```

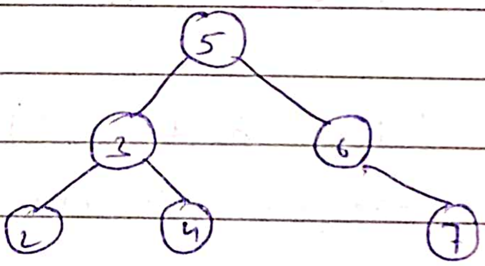
4) Delete Node in BST (leet code)

```
-> struct TreeNode *smallest (struct TreeNode *root)
   {
       struct TreeNode * cur = root;
       while (cur -> left != NULL)
        {
           cur = cur -> left;
        }
       return cur;
   }

   struct TreeNode *deleteNode (struct TreeNode *root,
        int key)
   {   if (root == NULL)
          { return root;
         }

        if (key < root -> val)
        {
           root -> left = deleteNode(root -> left, key);
        }
    else if (key > root -> val) {
         root -> right = deleteNode (root -> right, key);
        }

    else
    {

        if (root -> left = NULL){
           struct TreeNode * temp = root -> right;
           free(root)
           return temp;
        }
else if (root->right == NULL){
        struct TreeNode *temp = root->left;
        free(root);
        return temp;
    }
    struct TreeNode *temp = smallest(root->right);
    root->val = temp->val;
      root->right = deleteNode(root->right,
                                         root->val);
  }
  return root;
}
```

Test cases

Case 1:

root = [5, 3, 6, 2, 4, null, 7]

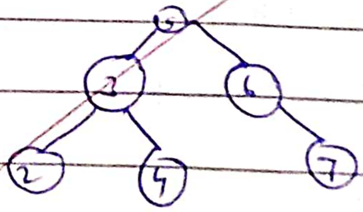


output: [5,4,6,2,
null, null,7]

key = 3

Case 2:

root = [5, 3, 6, 2, 4, null, 7



Key=0

Output: [5,3,6,
null,7]

Case 3 :

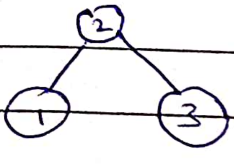
root = [ ]
key = 0
output = [ ]

Find the bottom left tree value (leet code)

```
void find( struct TreeNode *root, int
                *maxdepth , int depth , int *val ) {
    if(!root) return;
      if (*maxdepth < depth) {
           *maxdepth = depth;
           * val = root->val;
      }
    find (root->left, maxdepth, depth+1, val);
    find (root->right, maxdepth, depth +1, val);
}
int findBottomLeftValue (struct Treenode * root ) {
      int maxdepth = -1;
      int val = 0;
      find( root, & maxdepth, 0, & val);
      return val;
~~int find Bottom~~ }
```

Test cases

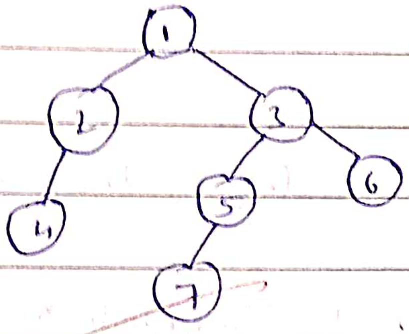
case 1
root = [2,1,3]



output = 1

Case 2

root = [1, 2, 3, 4, null, 5, 6, null, null, 7]



Output = 7

26.02.24