Answer to the question no.1

B tree property, the maximum number of keys that a node can hold is 2t-1. The maximum number of children that a node can have is 2t.

If the depth is 0 then number of nodes is 1, depth is 1 number of nodes is 2t, depth is 2 then number of nodes is $4t^2$.

Total number of nodes in the tree=$1+2t+4t^2+8t^3+\ldots\ldots + 2^h t^h = (2t)^{h+1}-1/2t-1$.

The maximum number of keys that each can have is 2t-1.

The total number keys= $\{(2t)^{h+1}-1/2t-1\}\text{X } 2t-1=(2t)^{h+1}-1$.

Therefore, the maximum number of keys that can be saved in a B-tree of height h is $(2t)^{h+1}-1$.

Answer to the question no.2

Let us assume the attributes for set objects and link objects are:

x.next: pointer to the next element of the list, null if x is the last element.

x.first: pointer to the set representative, that is, to the first element of the list.

x.last: if x is the first element of a list, then this field points to the last element.

x.size: if x is the first element, then this field contains the size of the list.

```
Make-set(x){
    x.next=null;
    x.first=x;
    x.last=x;
    x.size=1
}

Find-set(x){
    return x.first;
}
Union(x,y){
    If (x.first.size>y.first.size){
        append(x.first,y.first)
    }else{
        append(y.first,x.first)
    }
}


Append(x,y){
    x.last.next=y;
    x.size=x.size+y.size;
    z=y;
    while(z!=NULL){
        do z.first=x;
        z=z.next;
    }


}
```

Answer to the question no.3

Transpose of a graph has all the edges reversed in the original graph.

Adjacency list is of a graph G=(V,E) is an array of length |V|, adj[u]. adj[u] contains the list of nodes incident from the node u. Where, |V| is the number of vertices in the graph G.|E| is the total number of edges in the graph. Sum of the nodes in the each list is equals to the number of edges in the graph. That is, |E|.

```
Transpose_adjacency_list(G){
for(u=1 to |V|){
    for each vertex v in the list G.adj[u]
    add vertex u to the list $G^T$.adj[V]
    }
return $G^T$.adj;
}
```

Explanation:

The algorithm scans through each list in the adjacency list. Adds the vertexes to new adjacency lists such that the new list has reversed edges corresponding to the edges in the original graph. The graph is directed so each edge is appeared once in the array of the list.

Hence, total time complexity of algorithm will be O(|V|+|E|).

```
Transpose_adj_matrix(G,A){
    for i=1 to |V|{
        for(j=1 to |V|){
            b[j][i]=A[i][j]
        }
    }
    return b;
}
```

Explanation:

The algorithm repeats two loops to scan through the adjacency matrix. The adjacency matrix of the transpose is calculated. The new adjacency matrix is the transpose of the adjacency matrix of the original graph. The total time complexity is O(V^2).

Answer to the question no.4

```
computeAdjListOfEquivalentGraph( Graph array ){

            array2;  //Stores new undirected graph

            Exist[ V, V ];

            for i =0 to V {

                Exist[ i , i ] = 1;
            }

            for i=0 to array.V { //Traversing each nodes of
                int j=0;
                while( j <= array.adj[ i ]; ){
                    if ( Exist[ j , i ] !=1)
                        Exist[ j , i ] = 1;
                    Exist [ i , j ] =1;

                          add( array2.adj[i] , j ] );

                    // add vertex i to the neighbouring list vertex j
                    add ( array2.adj[j] , i );

            }
      }
      return array2;
}
```

Time complexity analysis:
The time complexity depends on the iteration of the nested loops, that is, the `for` loop running for the number of vertices- V, and the `while` loop inside it running for number of edges- E. Each node and edge is visited only a single time therefore the running time cumulates to O(|V| + |E|).

Answer to the question no.5

```
BFS( G,s){
      for each vertex u ∈ G.V – {s}{
         u.color = WHITE;
         u.d = ∞
         u. π = NIL
      }
      s.color = GRAY
      s.d = 0
      s. π = NIL
      Q = Ø
      Enqueue( Q , s)
      while Q ≠ Ø{
         u= Dequeue ( Q )
         for v=0 to |V| {
            if(v.d == ∞ && G.Adj[ u, v ] ==1 ){
                        v.color == GRAY
                        v.d = u.d +1
                        v.π = u
                        Enqueue ( Q ,v)
               }
         }
         u.color = BLACK
      }
}
```

//Modification of the book functions are highlighted in bold.

Time Complexity analysis:
Dequeuing and enqueuing takes O(1) time and also the vertices visited for either enqueue or dequeue is at most once, so the cumulative time to do this operation is O(V). One whole row of adjacency matrix is iterated over in the for loop, so one iteration of this loop takes O(V) time. And as this for loop runs – V time, we can say that the scanning procedure also may take O(V^2) time. Total time complexity is O|V^2|.

Answer to the question no.6

```
 time;
DFS ( G ) {
    for each vertex x ∈ G.V
                x.π = NIL
            x.color = WHITE

    time = 0
    for each vertex x ∈ G.V
            if (x.color ==WHITE)
                    DFS_VISIT_USING_STACK( G, x )
}
DFS_VISIT_USING_STACK( G, u ){
        STACK = 0
    u.color = GRAY
    push( STACK, u )

    while( isEmpty(STACK) != true ){
            t= pop ( STACK )

            if( t=color == WHITE)
                    time++
                    t.color = GRAY
                    t.d = time

                    //Boolean to check if all the adjacent vertices are
                    //visited or not
                    AdjacentVerticesVisited == false

                    For v ∈ G.adj [ t ]
                            If( v.color == WHITE)
                                    v.π = t
                                    push(STACK,v)
                       AdjacentVerticesVisited = true

                    if(AdjacentVerticesVisited == false )
                            time++
                                t= pop(STACK)
                            t.color = BLACK
                            t.f = time
    }
```