Solution for Home Assignment 3 (Theoretical part)
Danis Alukaev BS19-02

3.1. Constructing B-tree.
Given a set of keys $K = \{9, 31, 42, 69, 2, 26, 34, 57, 76, 27, 58\}$. Our goal is to store these keys in the B-tree with a minimum degree $t = 2$, which can be defined as follows:
1. Every node other than the root must contain at least 1 key.
2. Every internal node other than the root has at least 2 children.
3. Every node can contain at most 3 keys.
4. Every internal node can have at most 4 children.
The resultant B-tree shown in Figure 1. Since the maximum degree (branch factor B) is even, we can implement an insert with a single pass down the tree (Preemptive splitting).

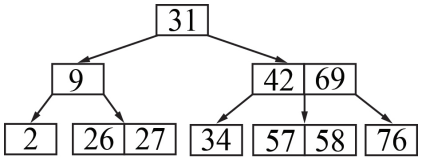

Figure 1: Resultant B-tree.

Figure 2 provides the structures of the B-tree after each insertion indexed from 1 to 11 for a given set of keys.

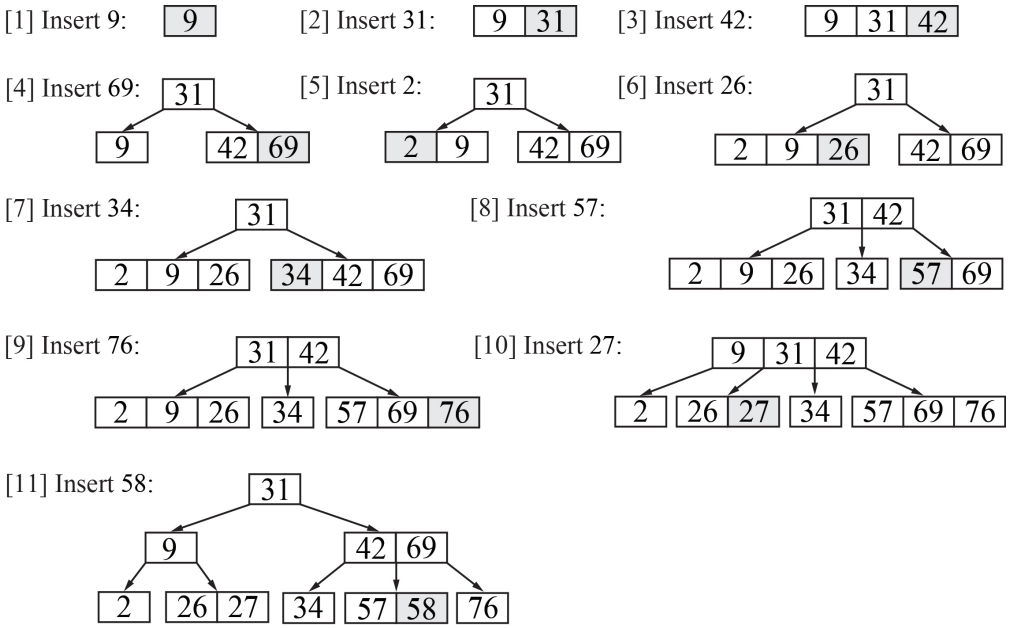

Figure 2: Inserting in B-tree.

3.2. The square graph.

The square of a directed unweighted graph $G = (V, E)$ is the graph $G_2 = (V, E_2)$ such that $(u, w) \in E_2$ iff there exists $v \in V$ such that $(u, v) \in E$ and $(v, w) \in E$; i.e., there is a path of exactly two edges from u to w. Our goal is to propose algorithms and their respective time complexities using the Big-O notation for constructing the square unweighted graphs.

The idea is to invoke the BFS and discover all vertices that are at distance of 2 from the starting vertex. Each of these vertices should be added to the adjacency list of starting vertex (or the connectivity marked in an adjacency matrix). Distance can be found using predecessor field π of vertex (see Algorithm 1, 2). Proposed procedure repeated for all vertices of the graph.

The pseudocode of an algorithm for constructing the square graph using adjacency list shown in table Algorithm 1 (page 3). During the breadth-first search of an input graph $G(V, E)$, each vertex is enqueued and dequeued at most once. These operations take $O(1)$ time, so the total time devoted to process all vertices is $O(|V|)$. When vertex is dequeued an algorithm scans the corresponding adjacency list. The total sum of the lengths of all adjacency lists is $\Theta(|E|)$, so BFS requires $O(|E|)$ time to process all adjacency lists. Thus, the total running time of the BFS procedure is $O(|V|) + O(|E|) = O(|V| + |E|)$. The BFS is invoked for all vertices (to explore adjacent vertices at the distance of 2). Therefore, the time complexity of the proposed algorithm is $O(|V|(|V| + |E|)) = O(|V|^2 + |V||E|) = O(|V||E|)$. It is important to note that the worst case is $O(|V|^3)$, because the number of edges in dense graphs is $E = \Theta(|V|^2)$.

The pseudocode of an algorithm for constructing the square graph using adjacency matrix shown in table Algorithm 2 (page 4). The total time devoted to enqueue-dequeue all vertices is the same with the Algorithm 1 and equals $O(|V|)$. However, when the vertex is dequeued an algorithm scans **all** the vertices (except dequeued one) of the graph to find adjacent vertices. The total number of these scans (all possible edges) is $\Theta(|V|^2)$, so BFS requires $O(|V|^2)$ time to determine all possible adjacent vertices. Thus, the total running time of the BFS procedure is $O(|V|) + O(|V|^2) = O(|V| + |V|^2) = O(|V|^2)$. We repeat this procedure on all vertices of the graph, so the time complexity of the proposed algorithm is $O(|V|(|V|^2)) = O(|V|^3)$.

Algorithm 1 Algorithm for constructing the square graph (using adjacency list).

```
1: procedure GET(G.AdjList, v)
2:     return list of adjacent to a vertex v vertices in a graph G

3: procedure APPEND(L, v)
4:     add the item v to the end of the list L

5: procedure SQUARE-MATRIX(G)
6:     new graph G_2                                          ▷ create new graph G_2
7:     G_2.Vertices ← G.Vertices                              ▷ copy the set of vertices
8:     G_2.AdjList ← G.AdjList                                ▷ copy adjacency list
9:     for each vertex s ∈ G.Vertices do
10:        terminateBFS ← False
11:        for each vertex u ∈ G.Vertices – {s} do             ▷ for all vertices except s
12:            u.state ← NOT VISITED                          ▷ set states and predecessors
13:            u.π ← NIL
14:        s.state ← OPENED                                   ▷ mark s as opened
15:        s.π ← NIL                                          ▷ mark predecessor as undefined
16:        Q ← ∅                                              ▷ create queue for BFS
17:        ENQUEUE(Q, s)
18:        while Q ≠ ∅ and not terminateBFS do
19:            u ← DEQUEUE(Q)
20:            for each v ∈ GET(G.AdjList, u) do              ▷ for all adjacent to u vertices
21:                if v.state == NOT VISITED then
22:                    v.state ← OPENED
23:                    v.π ← u
24:                    ENQUEUE(Q, v)
25:                    if v.π.π == s then                     ▷ v is at distance of 2 from starting s
26:                        APPEND(GET(G_2.AdjList, s), v)     ▷ insert new edge
27:                    else if v.π.π.π == s then              ▷ v is at distance of 3 from s
28:                        terminateBFS ← True                ▷ all vertices at distance of 2 are visited
29:                        break                              ▷ terminate BFS for s
30:                u.state ← EXITED                           ▷ mark vertex u as exited
31:    return G_2
```

Algorithm 2 Algorithm for constructing the square graph (using adjacency list).

```
1: procedure GetIndex(G, v)
2:     return index of vertex v of a graph G in adjacency matrix

3: procedure GetVertex(G, i)
4:     return vertex with index i in adjacency matrix of a graph G

5: procedure Square-Matrix(G)
6:     new graph G_2                                   ▷ create new graph G_2
7:     G_2.Vertices ← G.Vertices                       ▷ copy the set of vertices
8:     G_2.AdjMatrix ← G.AdjMatrix                     ▷ copy adjacency matrix
9:     for each vertex s ∈ G.Vertices do
10:        terminateBFS ← False
11:        for each vertex u ∈ G.Vertices – {s} do     ▷ for all vertices except s
12:            u.state ← NOT VISITED                   ▷ set states and predecessors
13:            u.π ← NIL
14:        s.state ← OPENED                            ▷ mark s as opened
15:        s.π ← NIL                                   ▷ mark predecessor as undefined
16:        Q ← ∅                                       ▷ create queue for BFS
17:        Enqueue(Q, s)
18:        index_s ← GetIndex(G, s)                    ▷ find the index of s in adjacency matrix
19:        while Q ≠ ∅ and not terminateBFS do
20:            u ← Dequeue(Q)
21:            index_u ← GetIndex(G, u)                ▷ find index of u in adjacency matrix
22:            for i = 0 to n do                       ▷ treat all indexes of vertices in adjacency matrix
23:                if G.AdjMatrix[index_u][i] == 1 then   ▷ check presence of edge
24:                    v ← GetVertex(G, i)
25:                    if v.state == NOT VISITED then
26:                        v.state ← OPENED
27:                        v.π ← u
28:                        Enqueue(Q, v)
29:                        if v.π.π == s then          ▷ v is at distance of 2 from starting s
30:                            G_2.AdjMatrix[index_s][i] ← 1   ▷ insert new edge
31:                        else if v.π.π.π == s then   ▷ v is at distance of 3 from s
32:                            terminateBFS ← True     ▷ all vertices at distance of 2 are visited
33:                            break                   ▷ terminate BFS for s
34:                u.state ← EXITED                    ▷ mark vertex u as exited
35:    return G_2
```