Chap 6. Graph (2)

Contents

6.2 Elementary Graph Operations

- Graph traversal
  - given G=(V, E) and a vertex *v* in V(G)
  - visit all vertices reachable from *v*

- *Depth First Search*
  - similar to a preorder tree traversal
  - uses *stack* or *recursion*

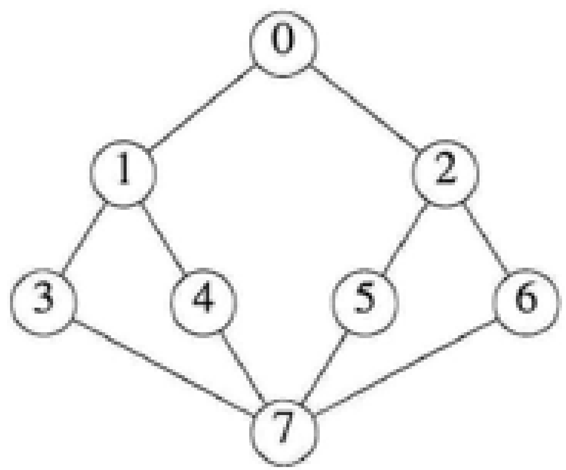
- *Breadth First Search*
  - similar to a level order tree traversal
  - uses *queue*

- We shall assumes that
  - the linked adjacency list for graph is used

6.2.1 Depth First Search

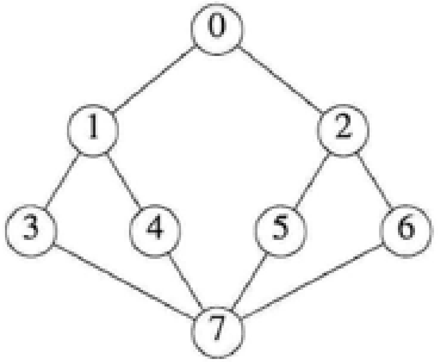
- Procedure

dfs(v){
  Label vertex v as reached.
  for (each unreached vertex u adjacent from v)
    dfs(u);
}



```c
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
```

```c
void dfs(int v)
{/* depth first search of a graph beginning at v */
   nodePointer w;
   visited[v] = TRUE;
   printf("%5d",v);
   for (w = graph[v]; w; w = w→link)
      if (!visited[w→vertex])
         dfs(w→vertex);
}
```



Program 6.1: Depth first search

Example 6.1

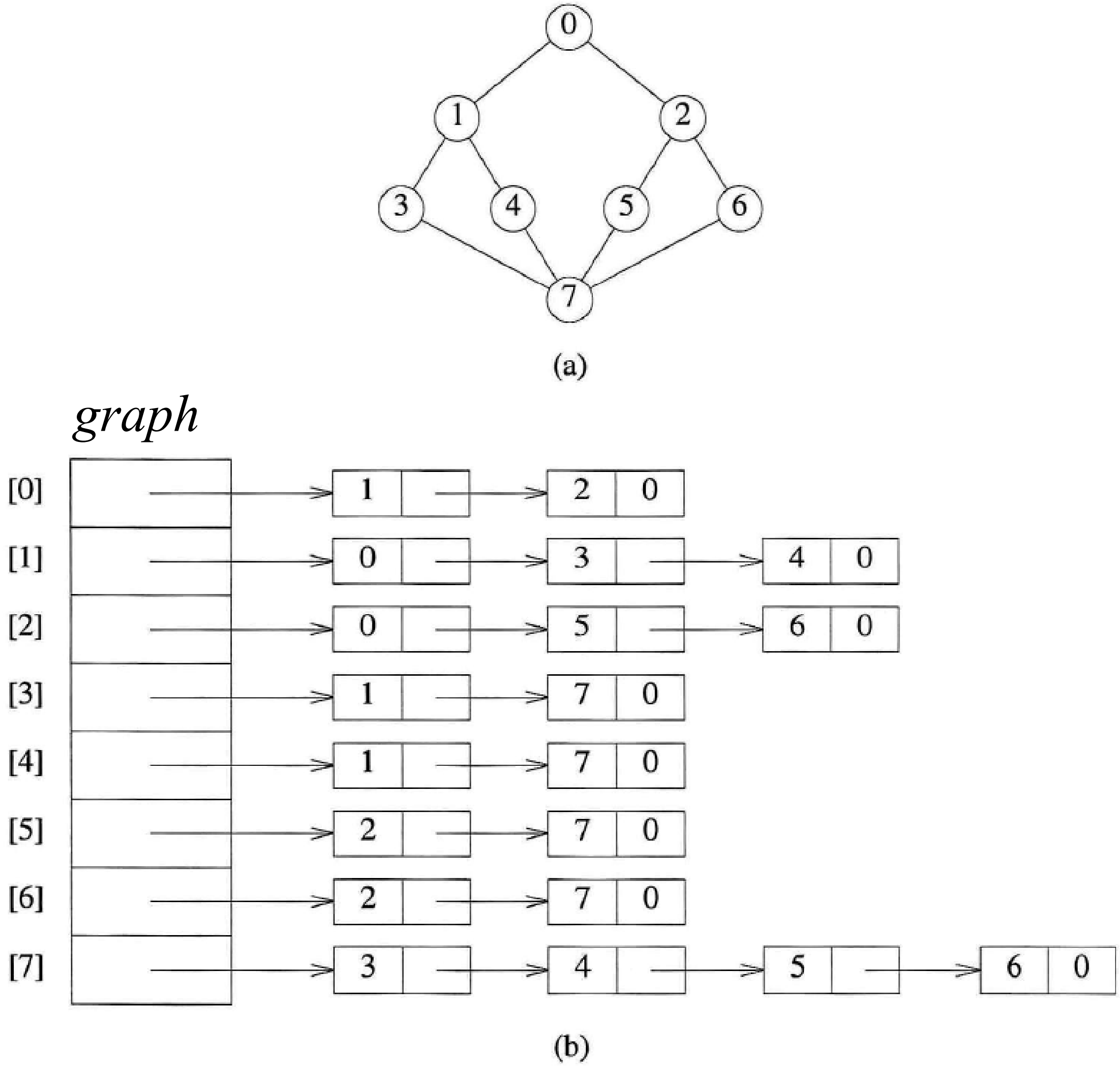
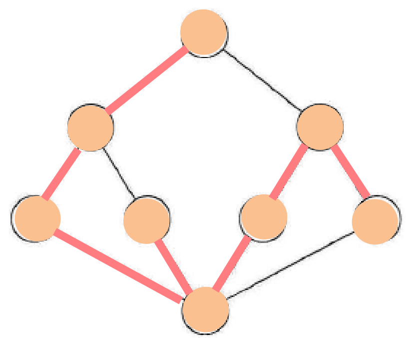


Figure 6.16: Graph *G* and its adjacency lists

dfs(0)

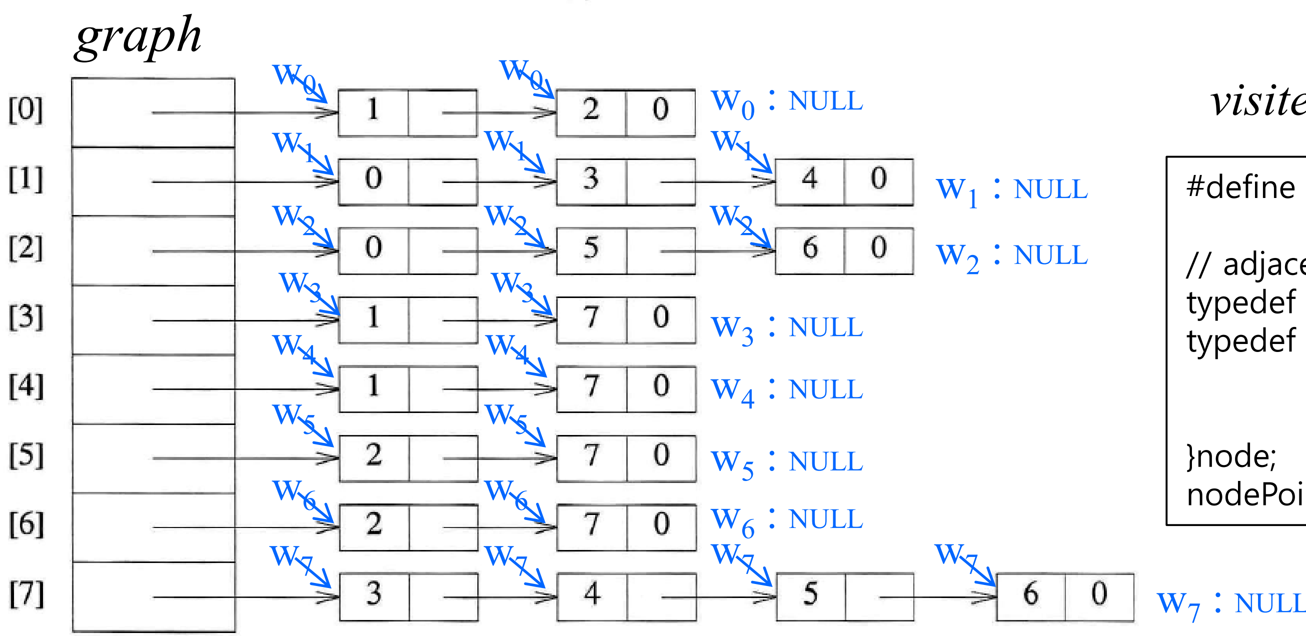


(a)

```
void dfs(int v)
{/* depth first search of a graph beginning at v */
   nodePointer w;
   visited[v] = TRUE;
   printf("%5d",v);
   for (w = graph[v]; w; w = w→link)
      if (!visited[w→vertex])
         dfs(w→vertex);
}
```

output *0 1 3 7 4 5 2 6*

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
visited	*T*	*T*	*T*	*T*	*T*	*T*	*T*	*T*



(b)

Figure 6.16: Graph *G* and its adjacency lists

```
#define MAX 100

// adjacency list of an undirected graph
typedef struct node* nodePointer;
typedef struct node{
        int vertex;
        nodePointer link;
}node;
nodePointer graph[MAX] = { NULL };
```

<system stack>

※ push/pop of the activation record of dfs()

- Analysis of *dfs*
  - if adjacency list is used
    - search for adjacent vertices : $O(e)$
  - if adjacency matrix is used
    - time to determine all adjacent vertices to v : $O(n)$
    - total time : $O(n^2)$

0	1	1	0	0	0	0	0
1	0	0	1	1	0	0	0
1	0	0	0	0	1	1	0
0	1	0	0	0	0	0	1
0	1	0	0	0	0	0	1
0	0	1	0	0	0	0	1
0	0	1	0	0	0	0	1
0	0	0	1	1	1	1	0

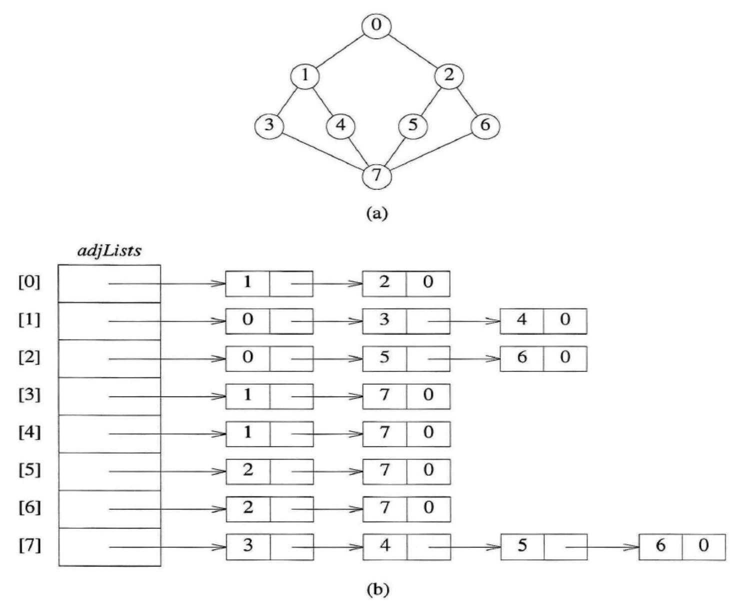


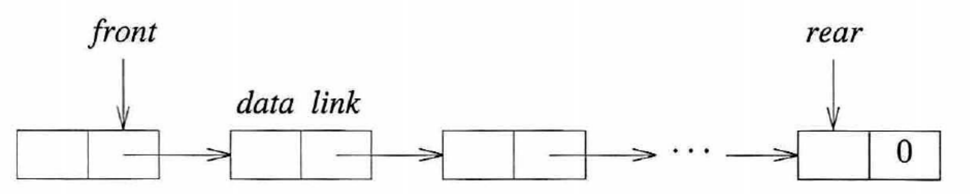
Figure 6.16: Graph *G* and its adjacency lists

6.2.2 Breadth First Search

- Procedure
  - visit start vertex and put into a FIFO *queue*.
  - repeatedly remove a vertex from the queue,
    visit its unvisited adjacent vertices,
    put newly visited vertices into the queue.

The queue definition and the function prototypes used by *bfs* are:

```
typedef struct queue *queuePointer;
typedef struct queue {
        int vertex;
        queuePointer link;
        };
queuePointer front, rear;
void addq(int);
int deleteq();
```

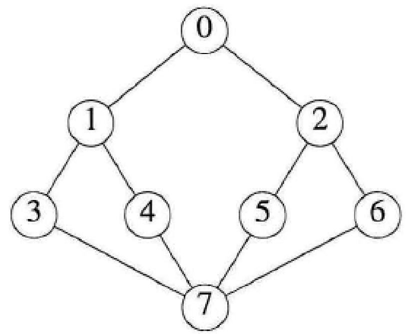


(b) Linked queue

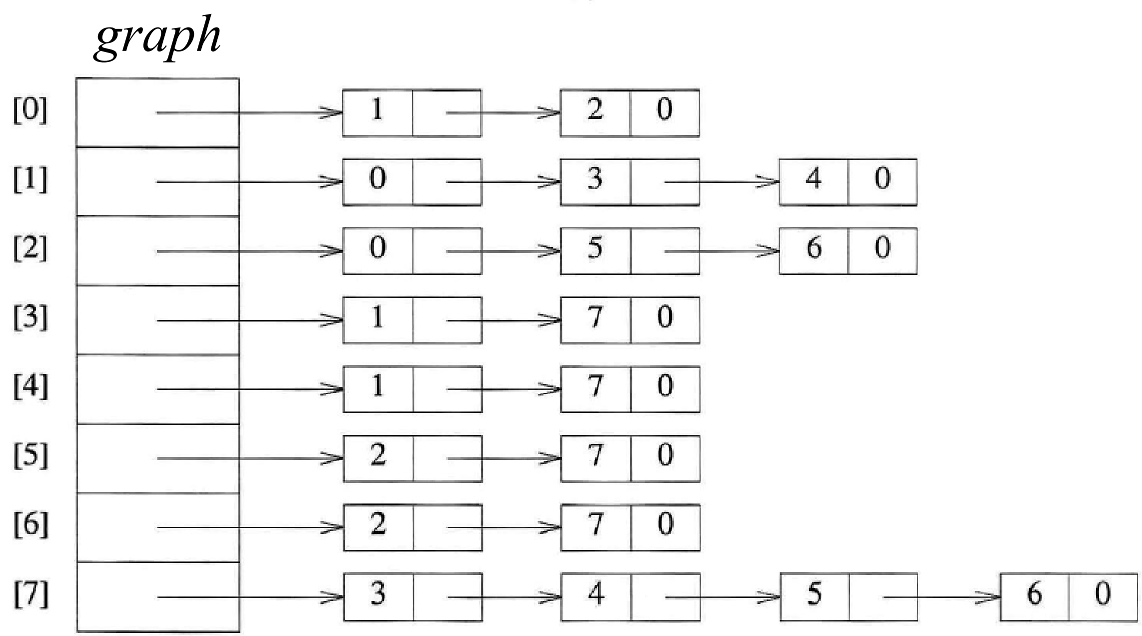
– use a *dynamically linked queue* as in Chapter 4
  - Program 4.7, 4.8
    – replace all reference to *element* with int.

```
void bfs(int v)
{/* breadth first traversal of a graph, starting at v
    the global array visited is initialized to 0, the queue
    operations are similar to those described in
    Chapter 4, front and rear are global */
  nodePointer w;
  front = rear = NULL; /* initialize queue */
  printf("%5d",v);
  visited[v] = TRUE;
  addq(v);
  while (front) { // non-empty queue
    v = deleteq();
    for (w = graph[v]; w; w = w→link)
      if (!visited[w→vertex]) {
        printf("%5d", w→vertex);
        addq(w→vertex);
        visited[w→vertex] = TRUE;
      }
    }
}
```

Program 6.2: Breadth first search of a graph



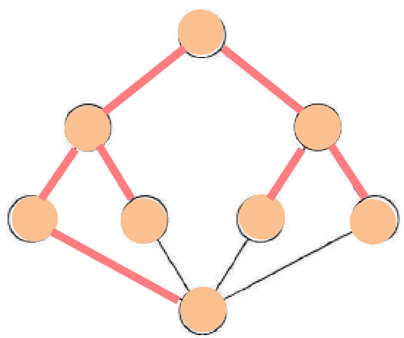
(a)



(b)

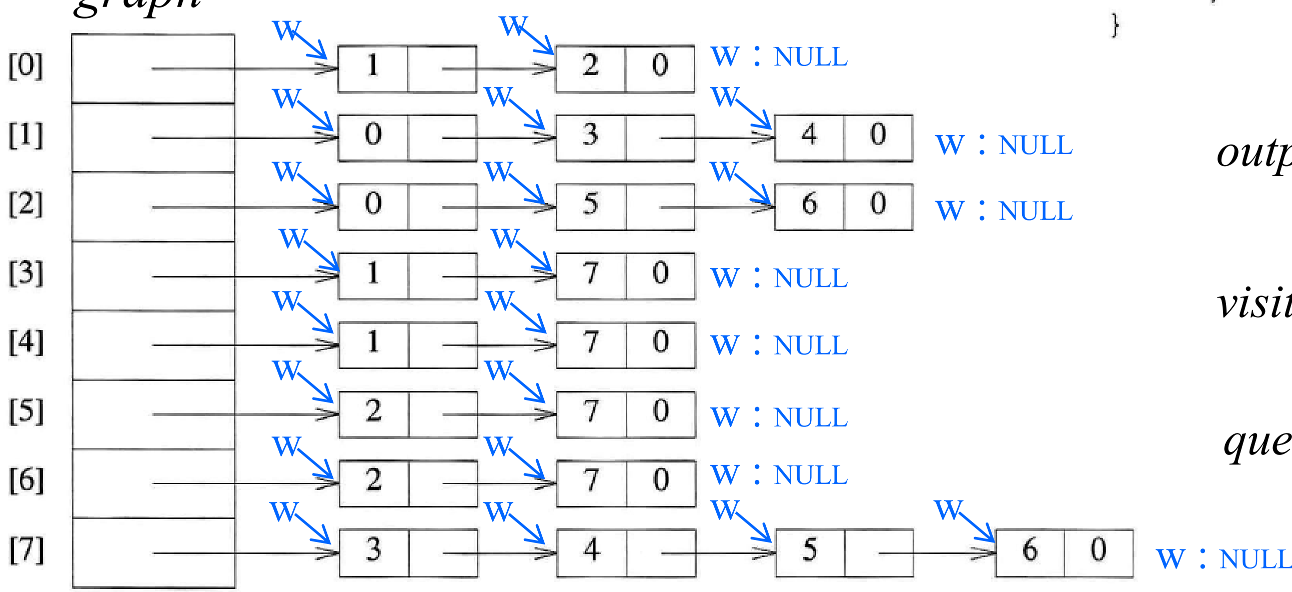
Figure 6.16: Graph G and its adjacency lists

bfs(0)



(a)

graph



(b)

Figure 6.16: Graph *G* and its adjacency lists

```
void bfs(int v)
{  nodePointer w;
   front = rear = NULL;
   printf("%5d",v);
   visited[v] = TRUE;
   addq(v);
   while (front) {
      v = deleteq();
      for (w = graph[v]; w; w = w→link)
         if (!visited[w→vertex]) {
            printf("%5d", w→vertex);
            addq(w→vertex);
            visited[w→vertex] = TRUE;
         }
      }
}
```

output *0 1 2 3 4 5 6 7*

visited

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
T	*T*	*T*	*T*	*T*	*T*	*T*	*T*

queue

- Analysis of *bfs*
  - adjacency list : $O(e)$
  - adjacency matrix : $O(n^2)$

0	1	1	0	0	0	0	0
1	0	0	1	1	0	0	0
1	0	0	0	0	1	1	0
0	1	0	0	0	0	0	1
0	1	0	0	0	0	0	1
0	0	1	0	0	0	0	1
0	0	1	0	0	0	0	1
0	0	0	1	1	1	1	0

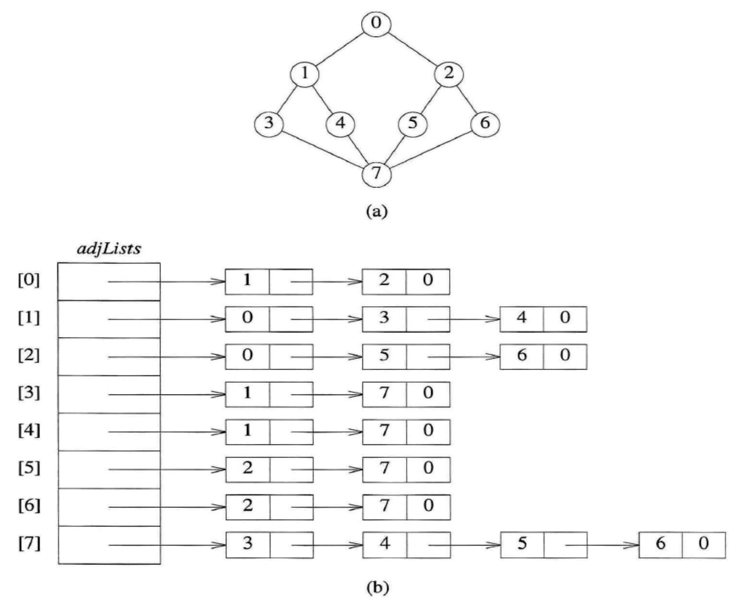


Figure 6.16: Graph *G* and its adjacency lists

6.2.3 Connected Components

- determining if an undirected graph is connected
  - calling *dfs*(0) or *bfs*(0) and then determining if there are any unvisited vertices
    - O(n+e) for adjacency list

- listing the connected components of a graph
  - making repeated calls to either *dfs*(v) or *bfs*(v) where v is an unvisited vertex. (Program 6.3)
    - O(n+e) for adjacency list
    - O(n^2) for adjacency matrix

```
void connected(void)
{/* determine the connected components of a graph */
int i;
for (i = 0; i < n; i++)
   if(!visited[i]) {
      dfs(i);
      printf("\n");
   }
}
```

Program 6.3: Connected components

6.2.4 Spanning Trees

- *Spanning tree*
  - any tree that consists solely of edges in G and that including all vertices

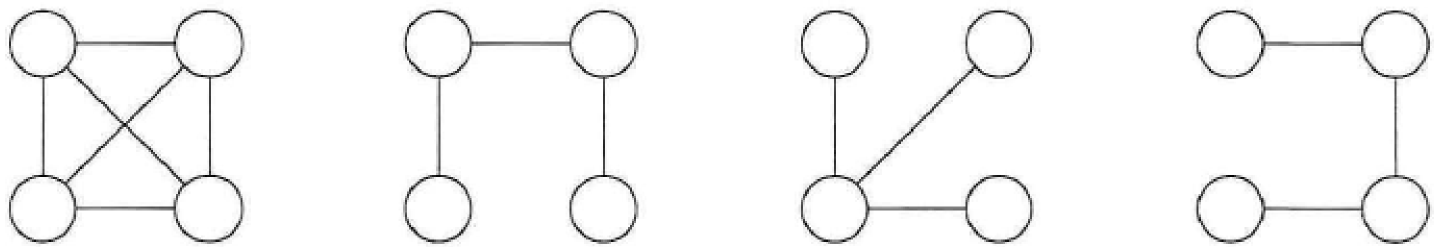
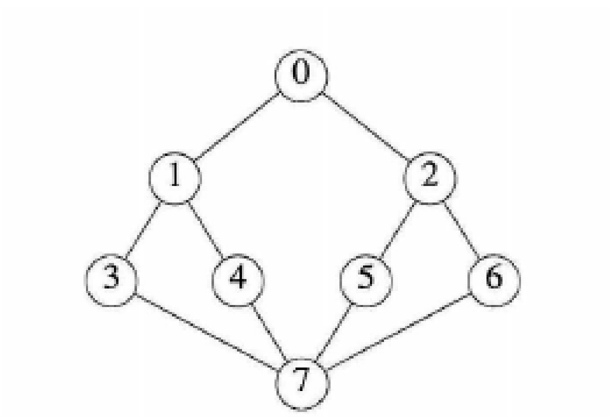
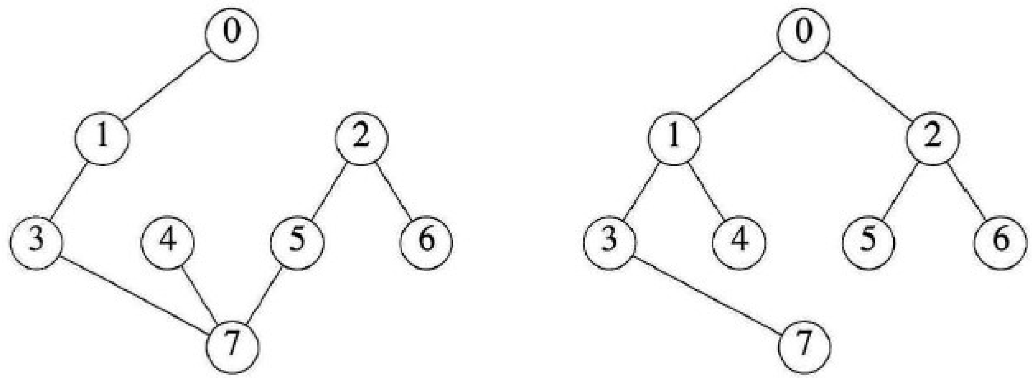


Figure 6.17: A complete graph and three of its spanning trees

- We may use *dfs* or *bfs* to create a spanning tree.



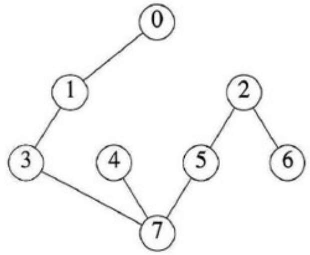


(a) *DFS* (0) spanning tree

(b) *BFS* (0) spanning tree

Figure 6.18: Depth-first and breadth-first spanning trees for graph of Figure 6.16

spanning tree



- Properties
  - If we add a nontree edge, (v,w), into any spanning tree, T, the result is a cycle that consists of the edge (v,w) and all the edges on the path from w to v in T.
  - A spanning tree is **a *minimal subgraph* G' of G** such that V(G') = V(G) and G' is connected.
    - A *minimal subgraph* is defined as one with the fewest number of edges
  - A spanning tree with n vertices has n-1 edges.