

Chapter 6

Graphs

6.1 The Graph Abstract Data Type

6.1.1 introduction

Graphs have been widely used in:

- **analysis of electrical circuits**
- **finding shortest routes**
- **project planning**

- **identification of chemical compounds**
- **statistical mechanics**
- **genetics**
- **cybernetics**
- **linguistics**
- **social science**
- **...**

6.1.2 Definitions

- graph $G = (V, E)$
- vertices $V(G) \neq \emptyset$
- edges $E(G)$
- undirected graph: $(u, v) = (v, u)$
- directed graph: $\langle u, v \rangle$, u ---tail, v ---head,
 $\langle u, v \rangle \neq \langle v, u \rangle$

Three graphs:

G_1 :

$$V(G_1) = \{0, 1, 2, 3\}$$

$$E(G_1) = \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)\}$$

G_2 :

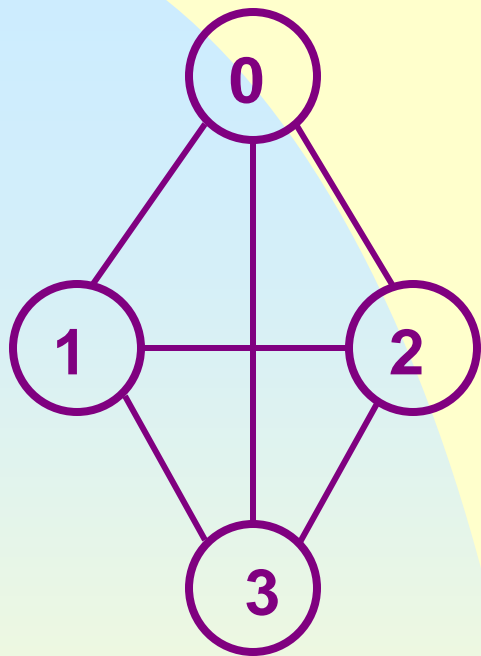
$$V(G_2) = \{0, 1, 2, 3, 4, 5, 6\}$$

$$E(G_2) = \{(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)\}$$

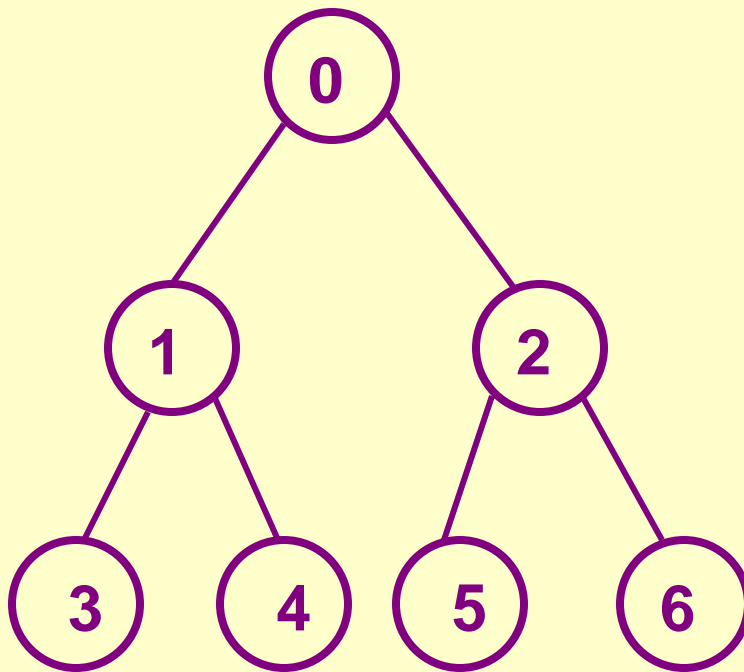
G_3 :

$$V(G_3) = \{0, 1, 2\}$$

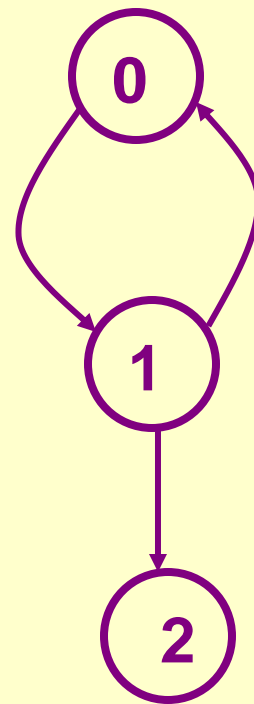
$$E(G_3) = \{<0, 1>, <1, 0>, <1, 2>\} \text{ (directed)}$$



(a) G_1



(b) G_2



(c) G_3

Restrictions:

(1) (v, v) or $\langle v, v \rangle$ is not legal, such edges are known as **self edges**.

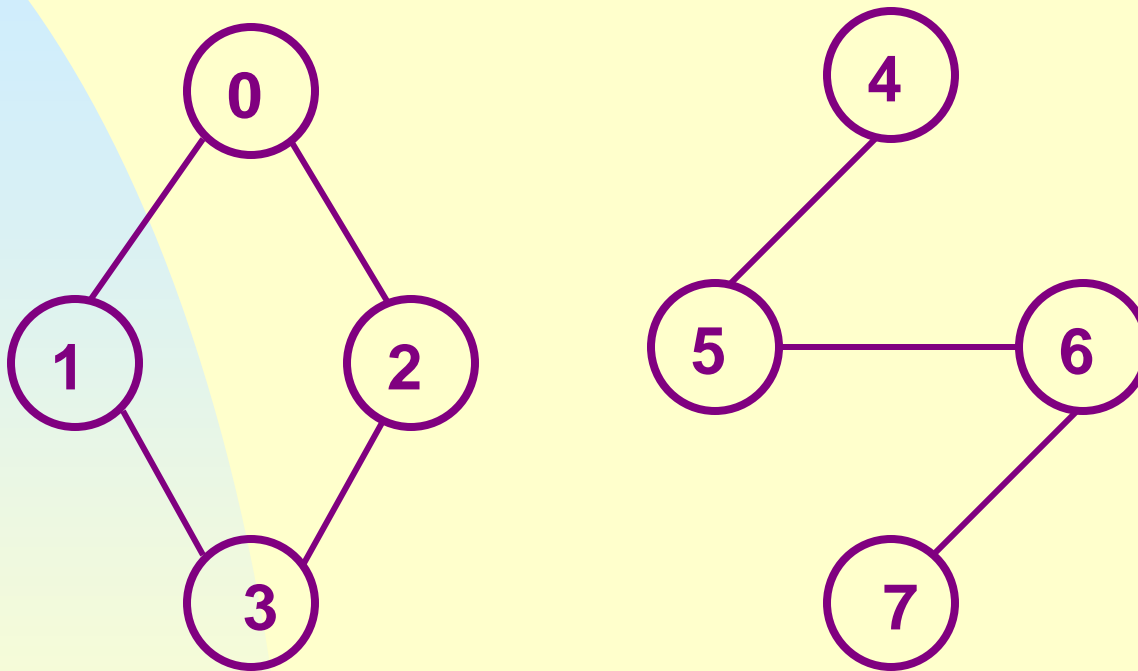
(2) multiple occurrences of the same edges are not allowed. If allowed, we get a **multigraph**.

- The maximum number of edges in any n -vertex, undirected graph is $n(n-1)/2$, and in directed graph is $n(n-1)$.
- An n -vertex undirected graph with $n(n-1)/2$ edges is said to be **complete**.

- If $(u, v) \in E(G)$, we say u and v are **adjacent** and edge (u, v) is **incident** on vertices u and v . If $\langle u, v \rangle$ is a directed edge, then vertex u is **adjacent to** v , and v is **adjacent from** u , $\langle u, v \rangle$ is **incident to** u and v .
- A **subgraph** of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.
- A **path** from u to v in G is a sequence of vertices $u, i_1, i_2, \dots, i_k, v$ such that $(u, i_1), (i_1, i_2), \dots, (i_k, v)$ are edges in $E(G)$. If G' is directed, then $\langle u, i_1 \rangle, \langle i_1, i_2 \rangle, \dots, \langle i_k, v \rangle$ are edges in $E(G')$.
- The **length** of a path is the number of edges on it.

- A **simple path** is a path in which all vertices except possibly the first and last are distinct.
- A **cycle** is a simple path in which the first and last vertices are the same.
- For directed graph, we have directed paths and cycles.
- In an undirected G , u and v are **connected** iff there is a path in G from u to v (also from v to u).
- An undirected G is **connected** iff for every pair of distinct u and v in $V(G)$, there is a path from u to v .

- A **connected component** is a maximal connected subgraph.



G_4 A graph with two connected components

- A **tree** is a connected acyclic graph.
- A directed G is **strongly connected** iff for every pair of distinct u and v in $V(G)$, there is a directed path from u to v and also from v to u .
- A **strongly connected component** is a maximal subgraph that is strongly connected.
- The **degree** of a vertex is the number of edges incident to it.

- For directed G , the **in-degree** of $v \in V(G)$ is the number of edges for which v is the head. The **out-degree** is the number of edges for which v is the tail.
- If d_i is the degree of vertex i in G with n vertices and e edges, then

$$e = \left(\sum_{i=0}^{n-1} d_i \right) / 2$$

- We'll refer to a directed graph as **digraph**, and a undirected graph as **graph**.

ADT 6.1 Graph

class Graph

{ // A non empty set of vertices and a set of undirected
// edges, where each edge is a pair of vertices.

public:

virtual ~Graph(){ };

// virtual destructor

bool IsEmpty() **const** {**return** n==0;};

// **return true** iff graph has no vertices

int NumberOfVertices() **const** {**return** n;};

// **return** the number of vertices in the graph

int NumberofEdges() **const** {**return** e;};

// **return** number of edges in the graph

virtual int Degree(**int** u) **const** =0;

// **return** number of edges incident to vertex u

```
virtual bool ExistsEdge(int u, int v) const =0;  
    // return true iff graph has edge (u, v)  
virtual void InsertVertex (int v) =0;  
    // insert vertex v into graph, v has no incident edges  
virtual void InsertEdge (int u, int v) =0;  
    // insert edge (u, v) into graph  
virtual void DeleteVertex (int v);  
    // delete v and all edges incident to it  
virtual void DeleteEdge (int u, int v) =0;  
    // delete edge (u, v) from the graph  
private:  
    int n;    // number of vertices  
    int e;    // number of edges  
};
```

6.1.3 Graph Representations

Three most commonly used representations:

- (1) Adjacency matrices**
- (2) Adjacency lists**
- (3) Adjacency multilists**

The actual choice depends on application.

6.1.3.1 Adjacency Matrix

The adjacency matrix of G is a $n \times n$ array, say a , such that:

$$a[i,j] = \begin{cases} 1 & \text{iff } (i, j) \in E(G) \text{ (or } \langle i, j \rangle \in E(G) \text{)} \\ 0 & \text{otherwise} \end{cases}$$

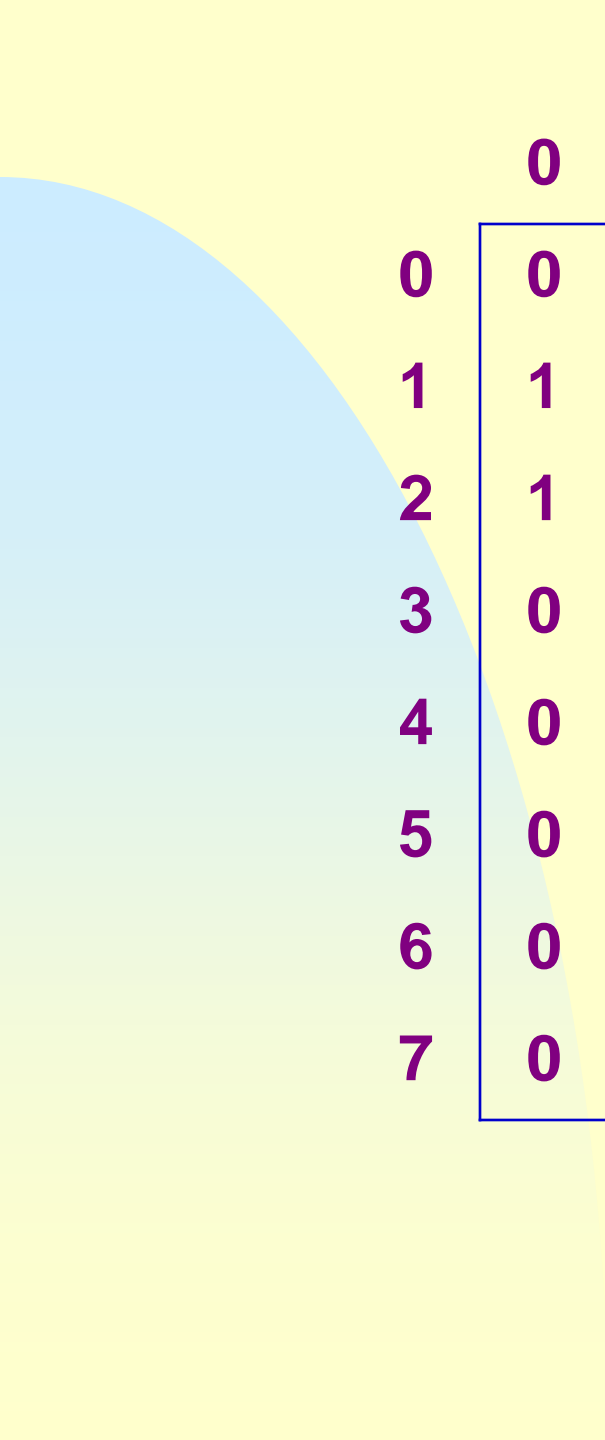
The next slide show adjacency matrices of G_1 , G_3 and G_4 .

	0	1	2	3
0	0	1	1	1
1	1	0	1	1
2	1	1	0	1
3	1	1	1	0

(a) G_1

	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

(b) G_3



	0	1	2	3	4	5	6	7
0	0	1	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0
2	1	0	0	1	0	0	0	0
3	0	1	1	0	0	0	0	0
4	0	0	0	0	0	1	0	0
5	0	0	0	0	1	0	1	0
6	0	0	0	0	0	1	0	1
7	0	0	0	0	0	0	1	0

G_4

For an graph, a is symmetric, and

$$d_i = \sum_{j=0}^{n-1} a[i][j]$$

For a digraph, a may not be symmetric, and

$$\text{out-}d_i = \sum_{j=0}^{n-1} a[i][j]$$

$$\text{in-}d_j = \sum_{i=0}^{n-1} a[i][j]$$

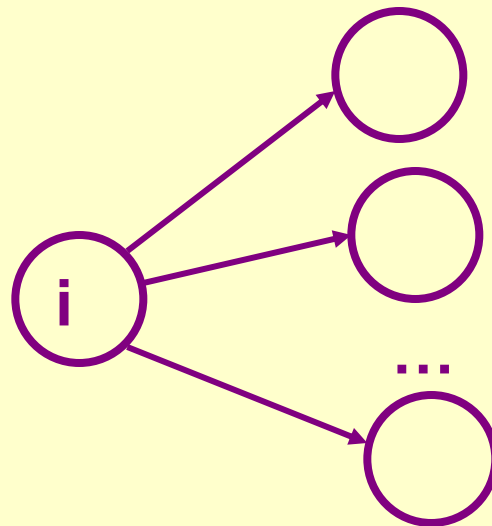
Problem: how many edges are there in G ?

Using a, we need $O(n^2)$. When $e \ll n^2 / 2$, and if we explicitly represent only edges in G , then we can solve the above problem in $O(e + n)$.

This lead to:

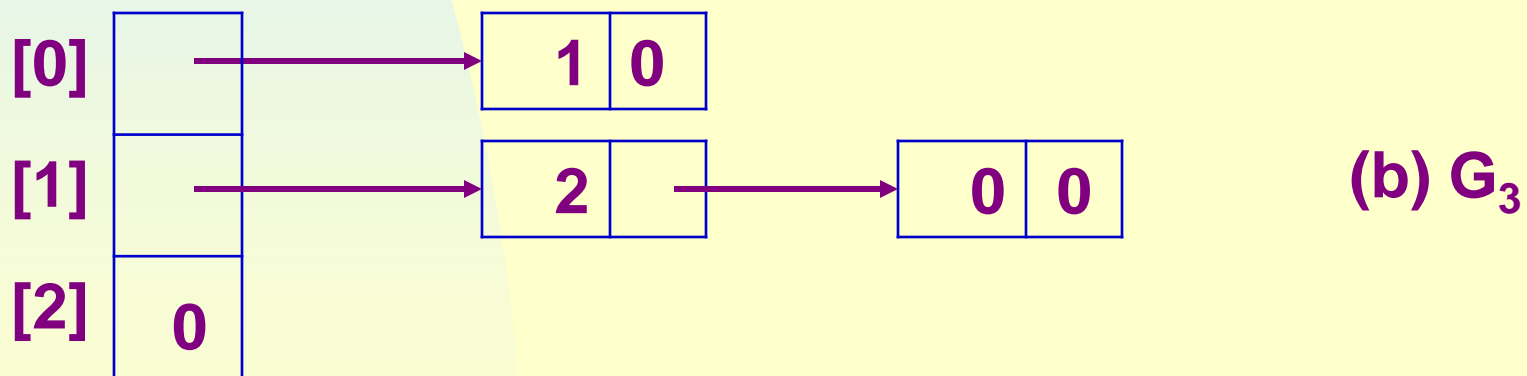
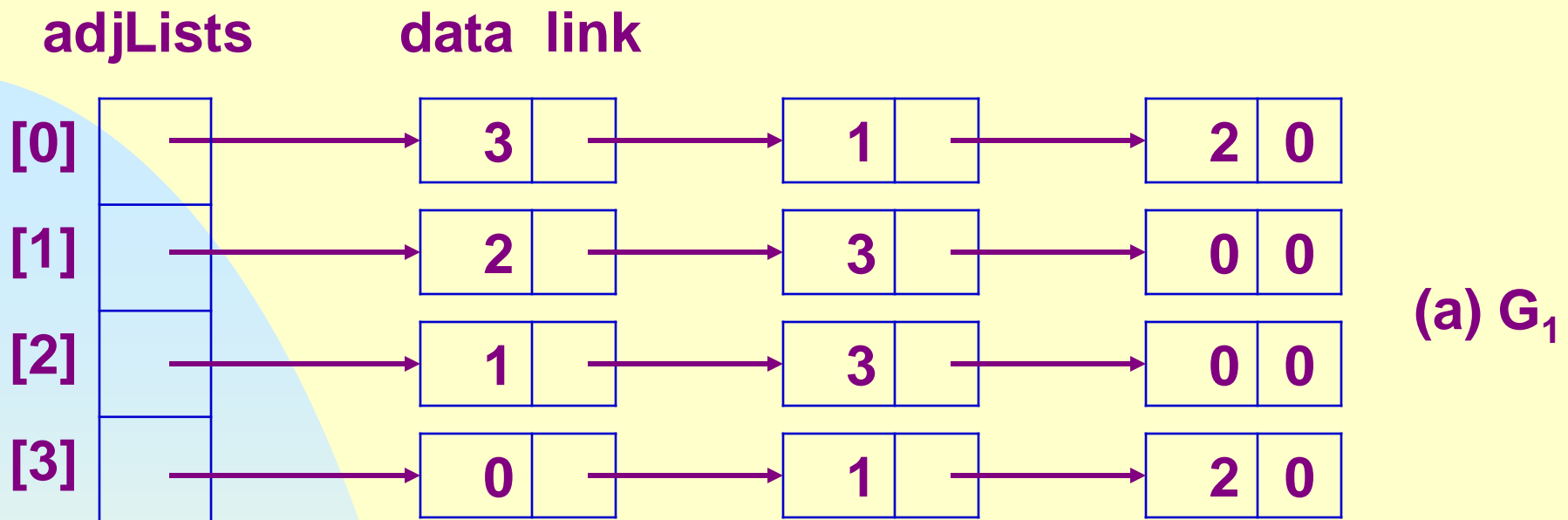
6.1.3.2 Adjacency Lists

- The n rows of the adjacency matrix are represented as n chains.
- The nodes in chain i represent the vertices adjacent from i .

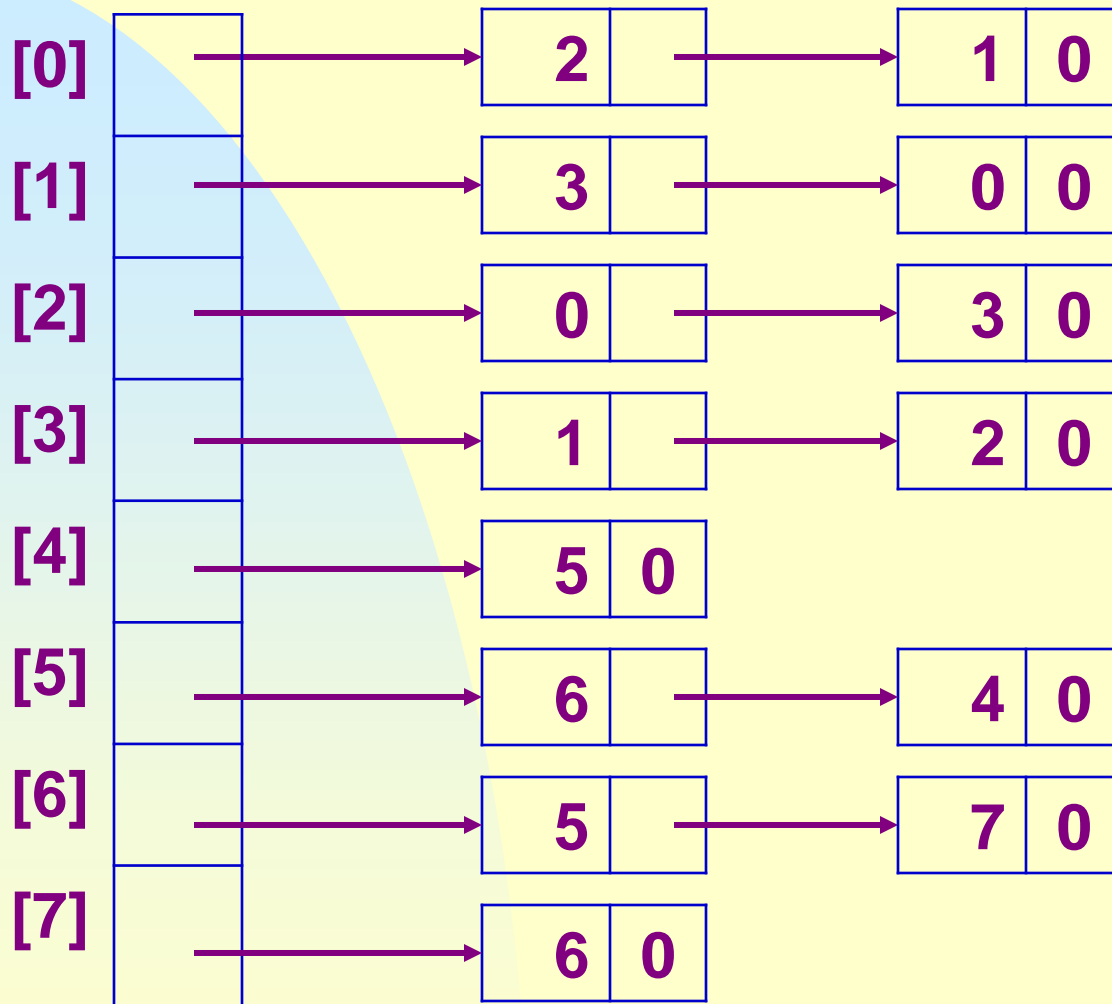


- The **data** field of a chain node store the index of an adjacent vertex.
- The vertices in each chain are not required to be ordered.
- An array **adjLists** is used for accessing any chain in $O(1)$.

```
class LinkedGraph
{
public:
    LinkedGraph (const int vertices): e(0)
    {
        if (vertices < 1) throw "Number of vertices must be > 0";
        n = vertices;
        adjLists = new Chain<int>[n];
        Chain<int> c; // set c.first to 0
        fill(adjLists, adjLists+n,c);
    };
private:
    Chain<int>* adjLists;
    int n;
    int e;
};
```



adjLists



(c) G_4

For an undirected graph with n vertices and e edges, this representation requires an array of size n and $2e$ chain nodes.

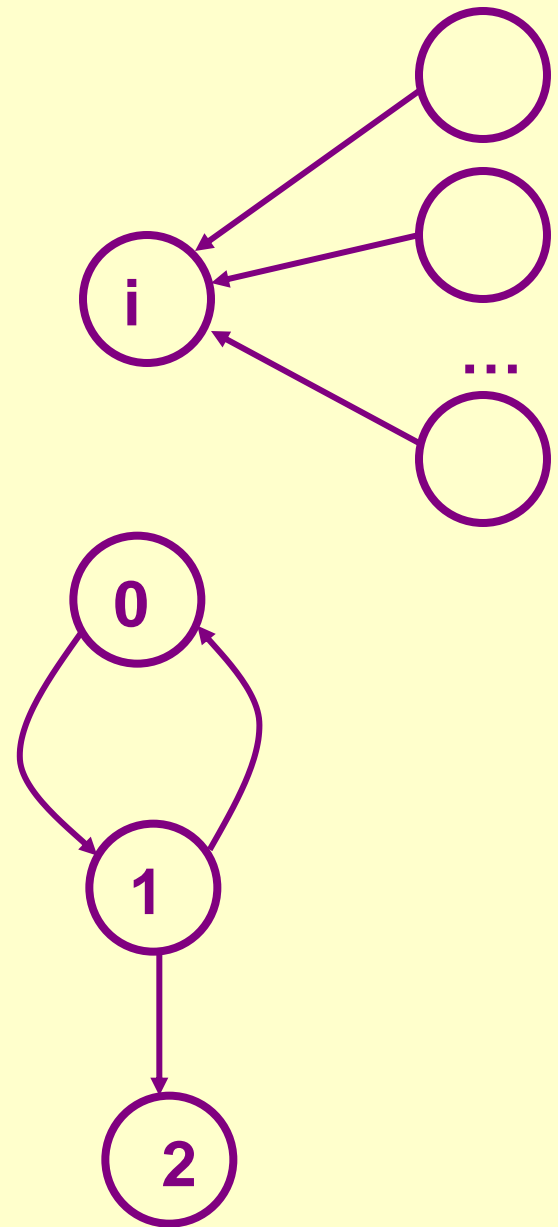
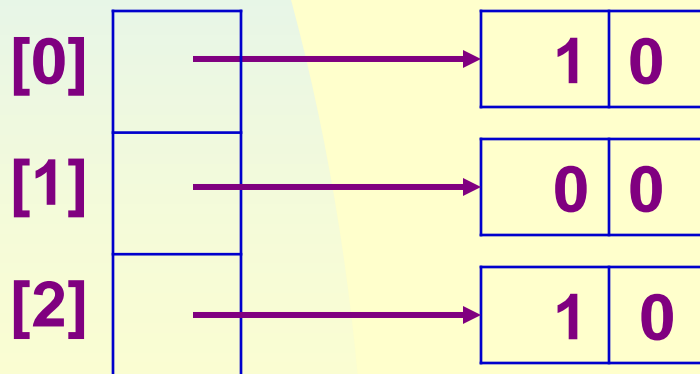
Now it is possible to determine the total number of edges in G in $O(e+n)$.

In case of digraph, the in-degree of a vertex is a little more complex to determine --- use a vector $c[n]$, when traverse from i to j , $c[j]++$.

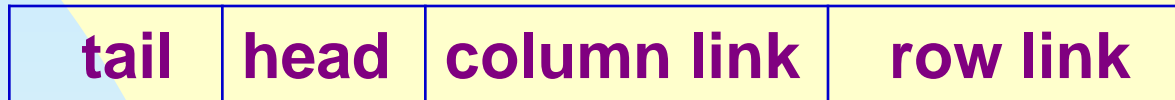
Inverse adjacency lists:

- one list for each vertex
- the nodes in list i represent the vertices adjacent to vertex i .

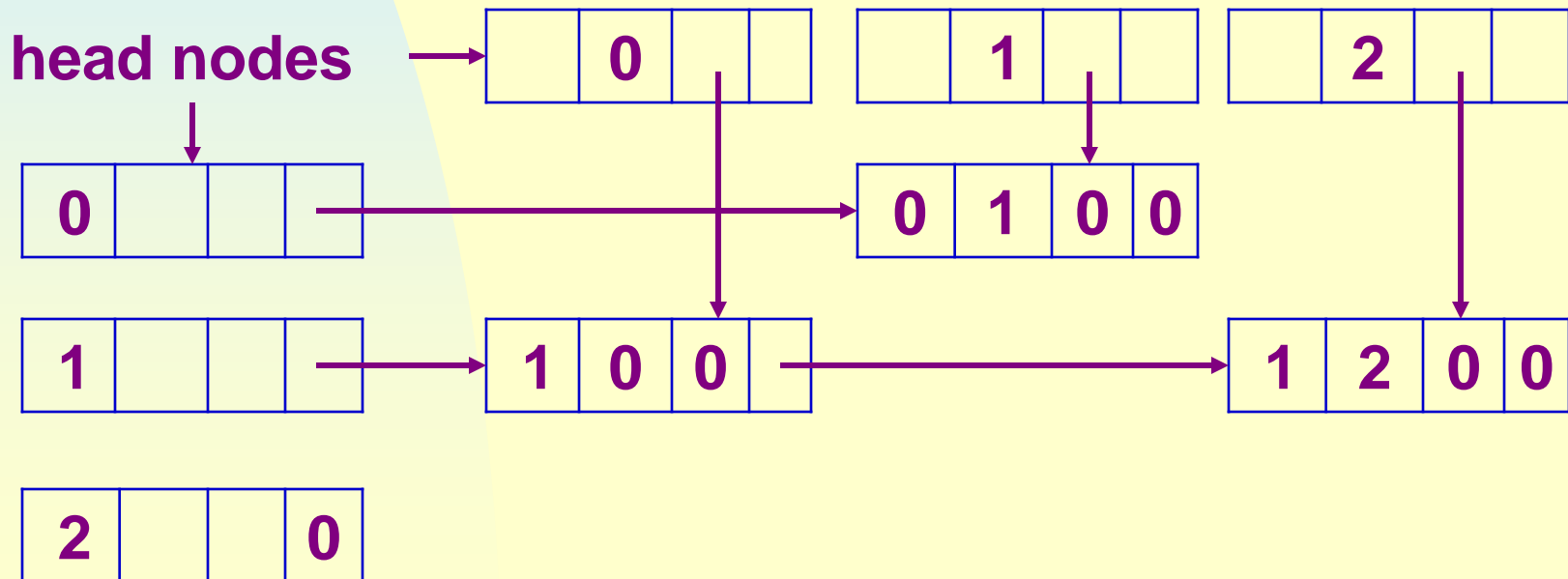
The following is the inverse adjacency lists for G_3 :



Combining nodes for adjacency lists and inverse adjacency lists together forms an **orthogonal list node**:



Here is the orthogonal representation for G_3 :



6.1.3.3 Adjacency Multilists

In the adjacency lists of an undirected graph, each (u, v) is represented by 2 entries.

In some situations, it is necessary to be able to determine the second entry for a particular edge and mark that edge as used.

So we need **multilists**: for each edge there is exactly one node, but it is in two lists.

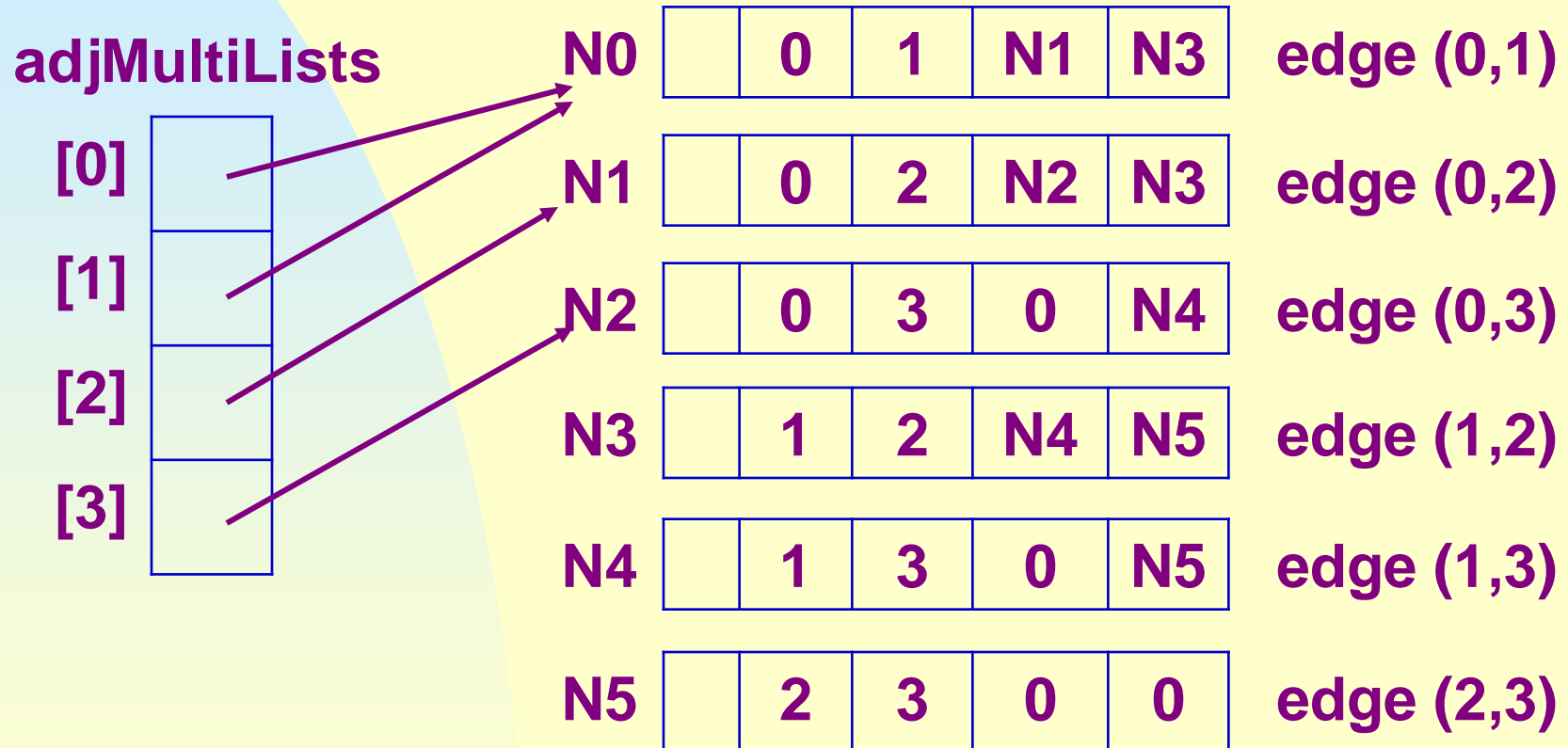
m	vertex1	vertex2	path1	path2
---	---------	---------	-------	-------

Note an edge can be marked from either path1 or path2.

```
class MGraph;
class MGraphEdge {
friend MGraph;
private:
    bool m;
    int vertex1, vertex2;
    MGraphEdge *path1, *path2;
};
typedef MGraphEdge *EdgePtr ;
class MGraph {
public:
    MGraph(const int);
private:
    EdgePtr *adjMultiLists;
    int n;
    int e;
};
```

```
MGraph::MGraph(const int vertices) : e(0)
{
    if (vertices < 1) throw "Number of vertices must be > 0";
    n = vertices;
    adjMultiLists = new EdgePtr[n];
    fill(adjMultiLists, adjMultiLists+n,0);
}
```

Here is the adjacency multilists for G_1 :



If p points to an `MGraphEdge` representing (u, v) , and given u , to get v we need the following test:

```
if (p→vertex1 == u) v = p→vertex2; else v = p→vertex1;
```

And we can insert an edge in $O(1)$:

```
void MGraph::InsertEdge(int u, int v) {  
    MGraphEdge *p = new MGraphEdge;  
    p→m = false; p→vertex1 = u; p→vertex2 = v;  
    p→path1 = adjMultiLists[u]; p→path2 = adjMultiLists[v];  
    adjMultiLists[u] = adjMultiLists[v] = p;  
}
```


6.1.3.4 weighted Edges

Edges may have **weight**.

- In the case of adjacency matrix, $A[i][j]$ may keep this information.
- In the case of adjacency lists, we need a **weight** field in the list node.
- A graph with weighted edges is called a **network**.

Exercises: P340-5, 9

6.2 Elementary Graph Operations

Given $G = (V, E)$, and v in $V(G)$, we wish to visit all vertices in G that are reachable from v .

In the following methods, we assume the graphs are undirected, although they work on the directed as well.

6.2.1 Depth-First Search

Idea:

Visit the start **v** first, next an unvisited **w** adjacent to **v** is selected and a depth-first search from **w** initiated. When **u** is reached such that all its adjacent vertices has been visited, we back up to the **last vertex** visited that has an unvisited vertex **w** adjacent to it and initiate a depth-first search from **w**. The search terminates when no unvisited vertex can be reached from any of the visited vertices.

```
virtual void Graph::DFS() // Driver
{
    visited=new bool[n];
    // visited is declared as a bool* data member of Graph
    fill(visited, visted + n, false);
    DFS (0);    // start search at vertex 0.
    delete [ ] visited;
}
```

```
virtual void Graph::DFS ( const int v) // workhorse
{
    //visit all previously unvisited vertices reachable from v
    visited[v] = true;
    for (each vertex w adjacent to v) // actual code uses an
                                    // iterator
        if ( !visited[w] ) DFS ( w);
}
```

Example 6.1

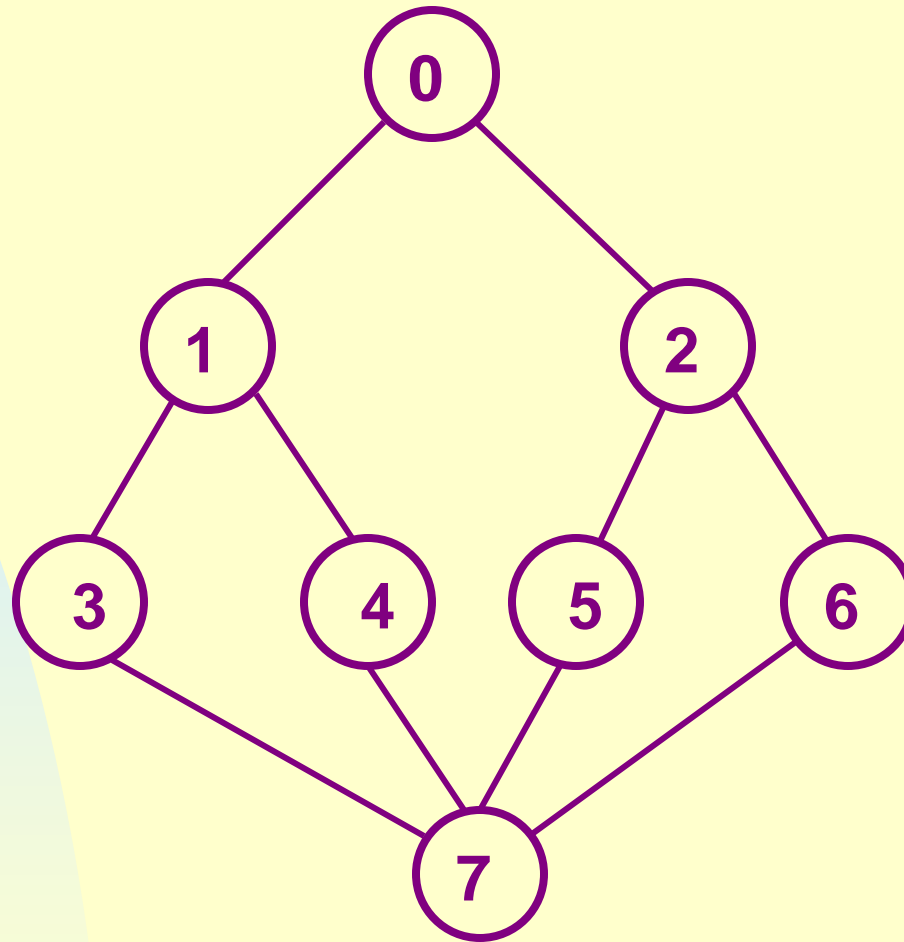


Fig. 6.17 (a) Graph G

adjLists

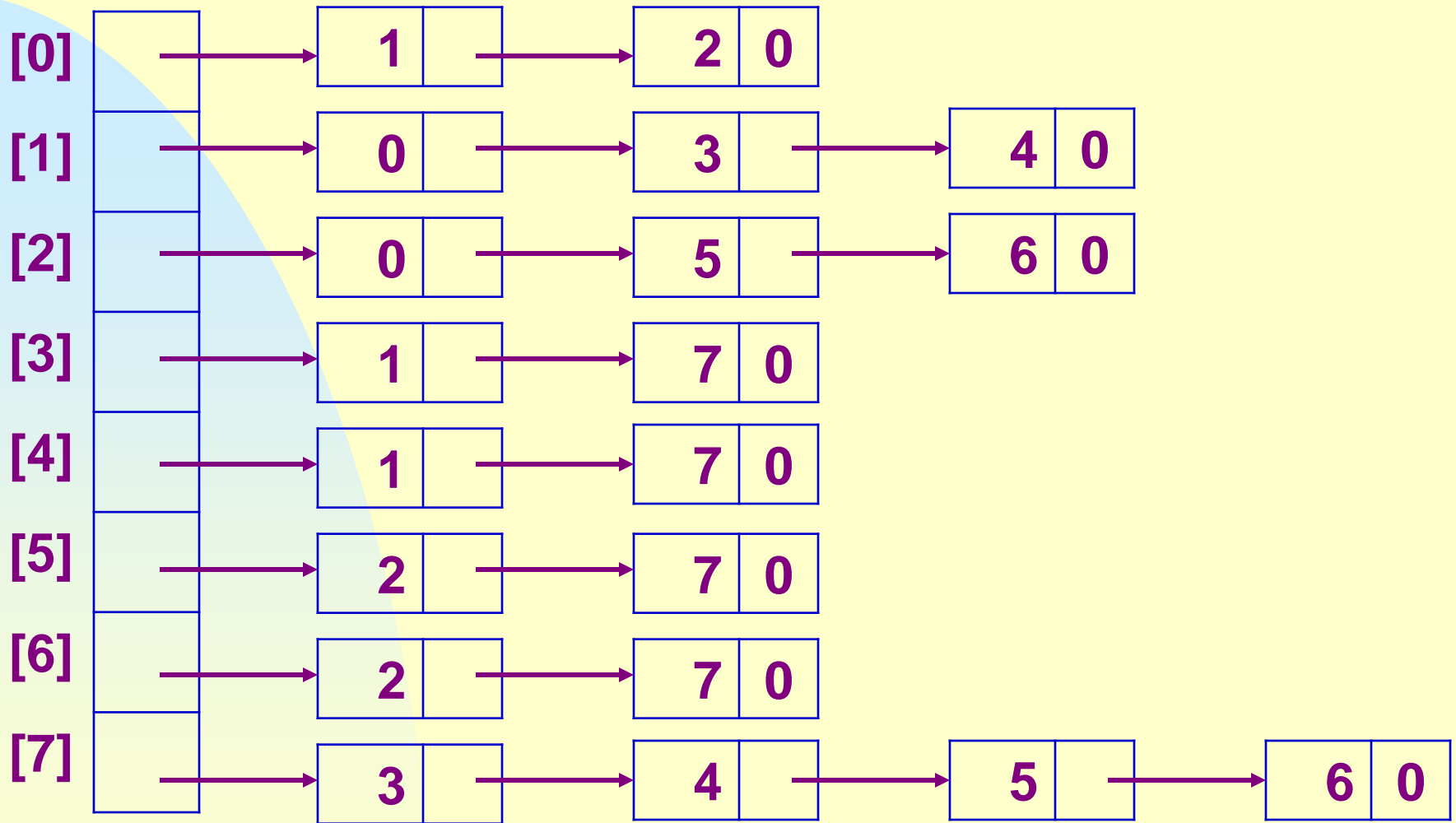


Fig 6.17 (b) Adjacency lists of G

Started from vertex 0, the vertices in G are visited in the order: 0, 1, (0), 3, (1), 7, (3), 4, (1), (7), (back to 7), 5, 2, (0), (5), 6, (2), (7), (back to 2), (back to 5), (7), (back to 7), (6), (back to 3), (back to 1), (4), (back to 0), (2).

Analysis of DFS:

Initiating visited needs $O(n)$. When G is represented by its adjacency lists, DFS examines each node at most once, so the time is $O(e+n)$.

If adjacency matrix is used, the time to determine all vertices adjacent to v is $O(n)$, there are n vertices, the total time is $O(n^2)$.

6.2.2 Breadth-First Search

In a breadth-first search, we begin from the start vertex **v**. Next, all unvisited vertices adjacent to **v** are visited. Unvisited vertices adjacent to these newly visited vertices are then visited, and so on.

```
virtual void Graph::BFS (int v)
{ // breadth-first search: begin at v, use a queue.
    visited = new bool[n];
    fill(visited, visited + n, false);
    visited[v] = true;
    Queue<int> q;
    q.Push(v);
```



```
while ( !q.IsEmpty()) {  
    v = q.Front();  
    q.Pop();  
    for (all vertices w adjacent to v) // actual code uses an  
                                        // iterator  
        if (!visited[w]) {  
            q.Push(w);  
            visited[w] = true;  
        }  
    } // end of while  
    delete [ ] visited;  
}
```

Example 6.2:

Started from vertex 0, the vertices in G of Fig. 6.17 are visited by breadth-first search in the order:

0, 1, 2, 3, 4, 5, 6, 7.

Analysis of BFS:

Each visited vertex enters the queue exactly once, the while loop is iterated at most n times.

If adjacency matrix is used, the loop takes $O(n)$ for each node visited, the total time is $O(n^2)$.

If adjacency lists are used, the loop has a total cost of $d_0 + \dots + d_{n-1} = O(e)$, the total time is $O(n+e)$.

6.2.3 Connected Components

To obtain all the connected components of a undirected graph, we can make repeated calls to either DFS(v) or BFS(v) for unvisited v .

This leads to function **Components**.

Function **OutputNewComponent** output all vertices visited in the most recent invocation of DFS, together with all edges incident on them.

```
virtual void Graph::Components()  
{ // Determine the connected components of the graph.  
    visited = new bool[n];  
    fill(visited, visited+n, false);  
    for (int i=0; i<n; i++)  
        if (!visited[i]) {  
            DFS (i); // find a component  
            OutputNewComponent();  
        }  
    delete [ ] visited;  
}
```

Analysis of Components:

If adjacency lists are used, the total time taken by DFS is $O(e+n)$. The output can be completed in $O(e+n)$ if DFS keeps a list of all newly visited vertices. The for loops take $O(n)$. The total time is $O(e+n)$.

If adjacency matrix is used, the total time is $O(n^2)$.

6.2.4 Spanning Trees

If G is connected, in DFS or BFS, all vertices are visited, the edges of G are partitioned into 2 sets:

- T --- Tree edges
- N --- Nontree edges

T may be obtained by inserting “ $T = T \cup \{(v, w)\}$ ” in the **if** clauses of DFS or BFS.

Any tree consisting solely of edges in G and including all vertices in G is called a **spanning tree.**

- **Depth-first spanning tree** --- the spanning tree resulting from a depth-first search
- **Breadth-first spanning tree** --- the spanning tree resulting from a breadth-first search

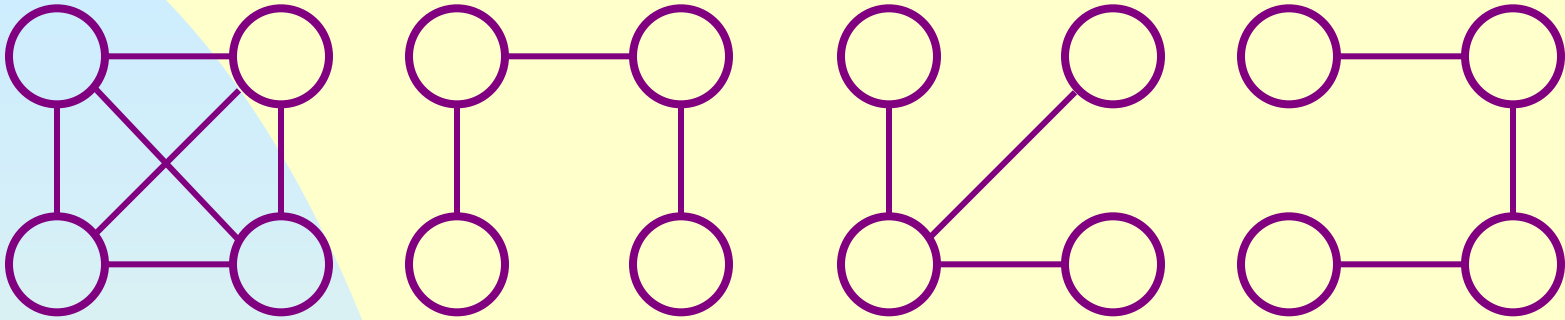


Fig. 6.18 A complete graph and three of its spanning trees

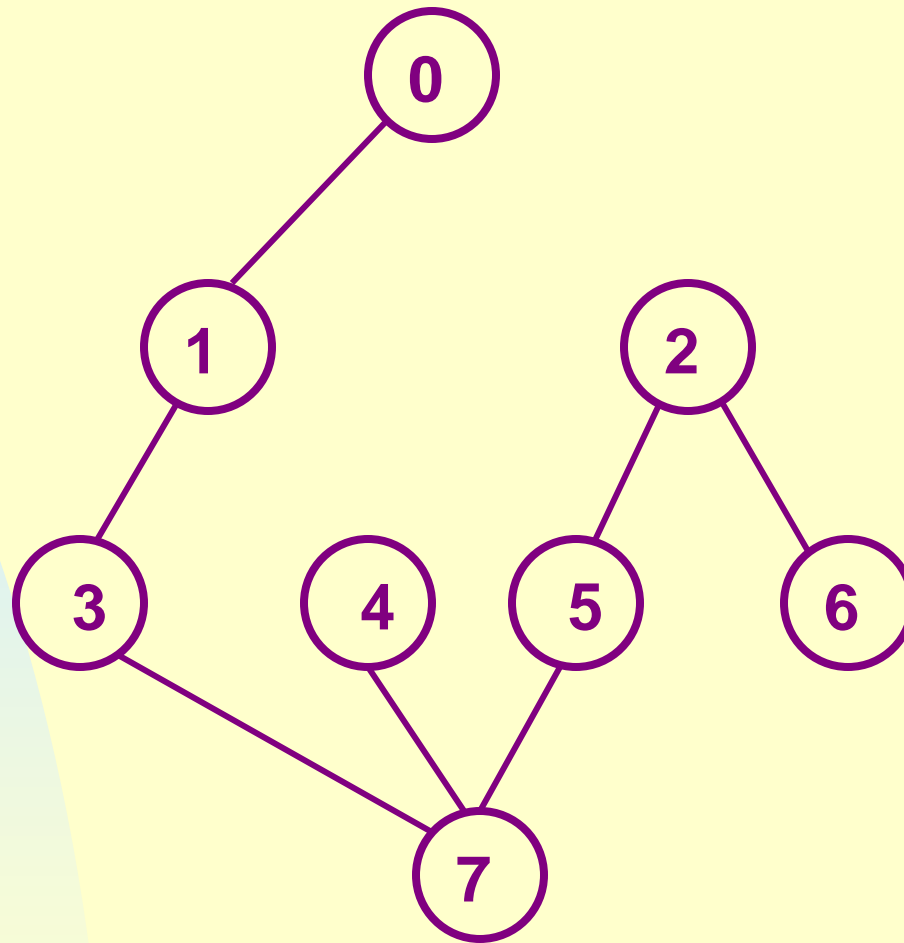


Fig. 6.19 (a) DFS(0) spanning tree for graph of Fig.6.17

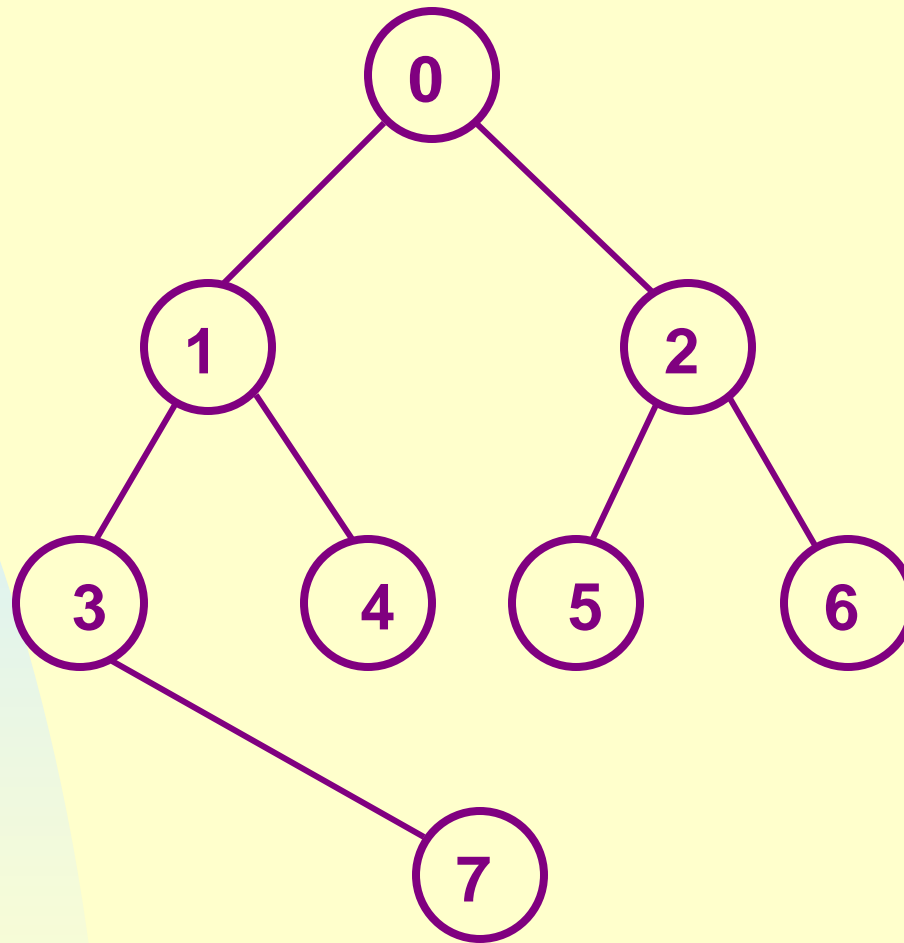


Fig. 6.19 (b) BFS(0) spanning tree for graph of Fig.6.17

If a nontree edge (v, w) is put into any spanning tree T , then a cycle is formed.

Example applications of spanning tree:

(1) Obtain an independent set of circuit equations for an electrical network by introducing nontree edges into the spanning tree one at a time.

(2) Connecting n cities with $n-1$ communication links.

In practical situation, edges will have weights (or cost) assigned to them. The **cost** of a spanning tree is the sum of the costs of the edges in that tree.

Exercises: P352-3, 5, 6

6.2.5 Biconnected Components

Assume that G is an **undirected, connected** graph.

Definition: A vertex v of G is an articulation point iff the deletion of v , together with the the deletion of all edges incident to v , leaves behind a graph that has at least two connected components.

Vertices 1 and 7 are the articulation points of the connected graph of Fig. 6.19(a).

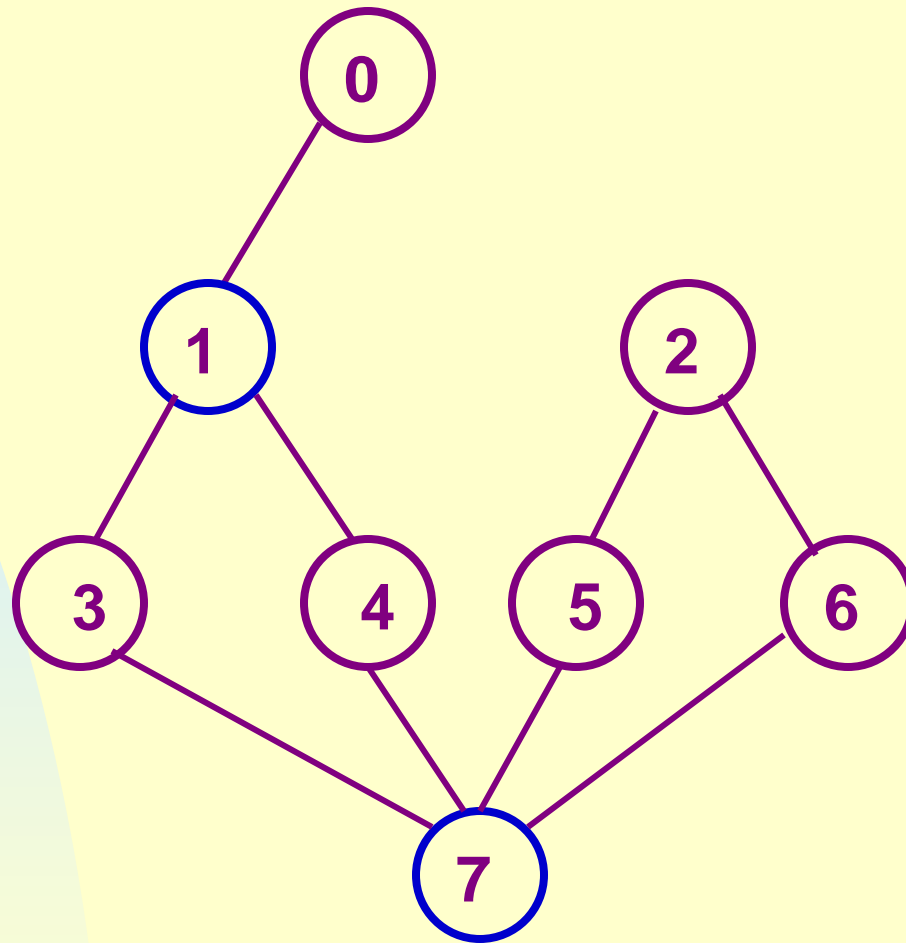


Fig. 6.19 (a) A connected graph

Definition: A biconnected graph is a connected graph that has no articulation points.

The graph of Fig. 6.19(a) is not biconnected.

Articulation points are undesirable in communication network.

Definition: A biconnected component of a connected graph G is a maximal biconnected subgraph H of G .

The graph of Fig. 6.19(a) contains three biconnected components, as shown in Fig. 6.19(b).

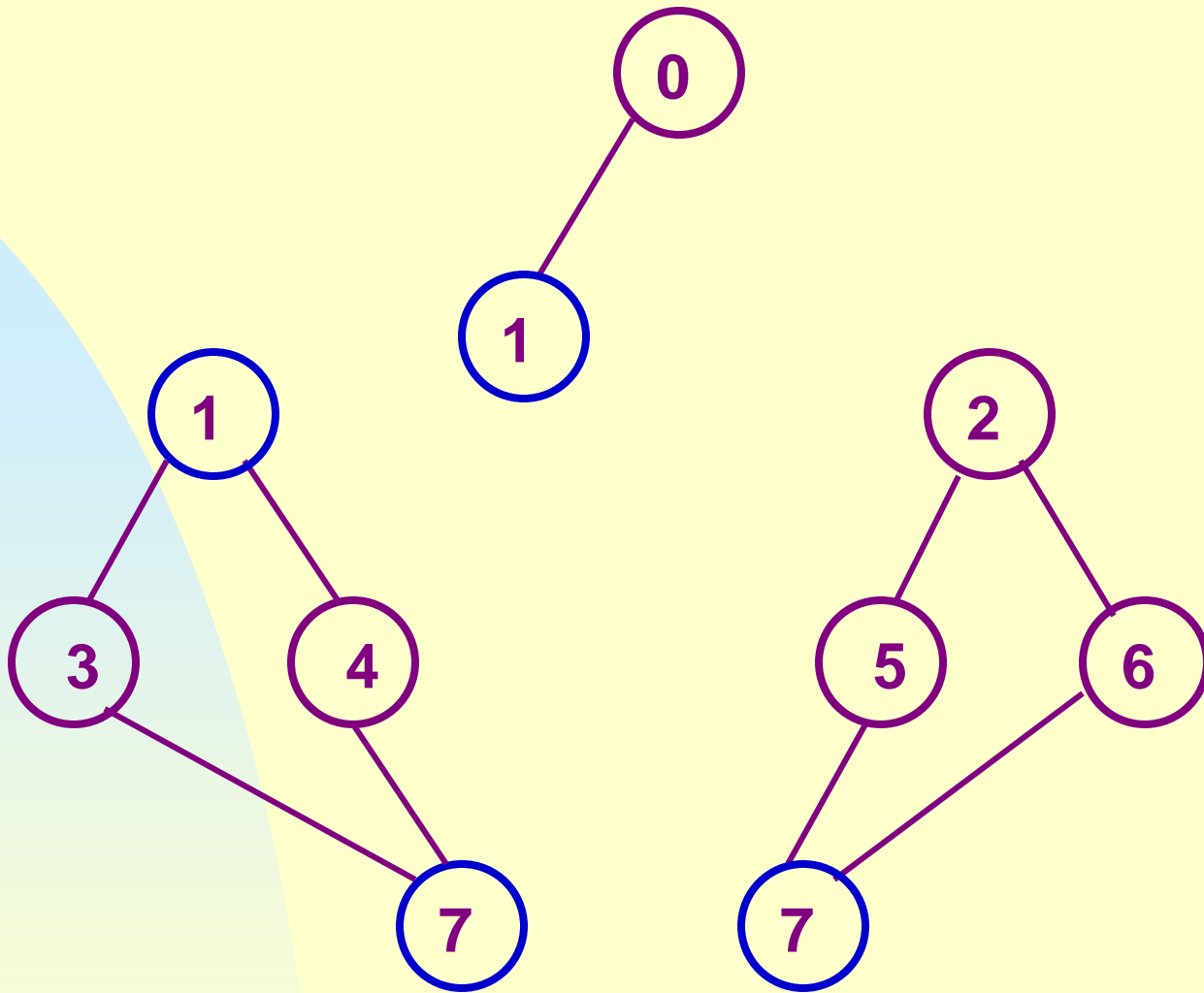


Fig. 6.19 (b) its biconnected components

Of the same graph G :

Two biconnected components can have at most one vertex in common.

No edge can be in two or more biconnected components, hence the biconnected components partition $E(G)$.

To find the biconnected components of G , we can use any depth-first tree of it. For the graph of Fig. 6.19(a), a depth-first tree with root 0 is shown in Fig. 6.20 with the nontree edges shown by broken lines.

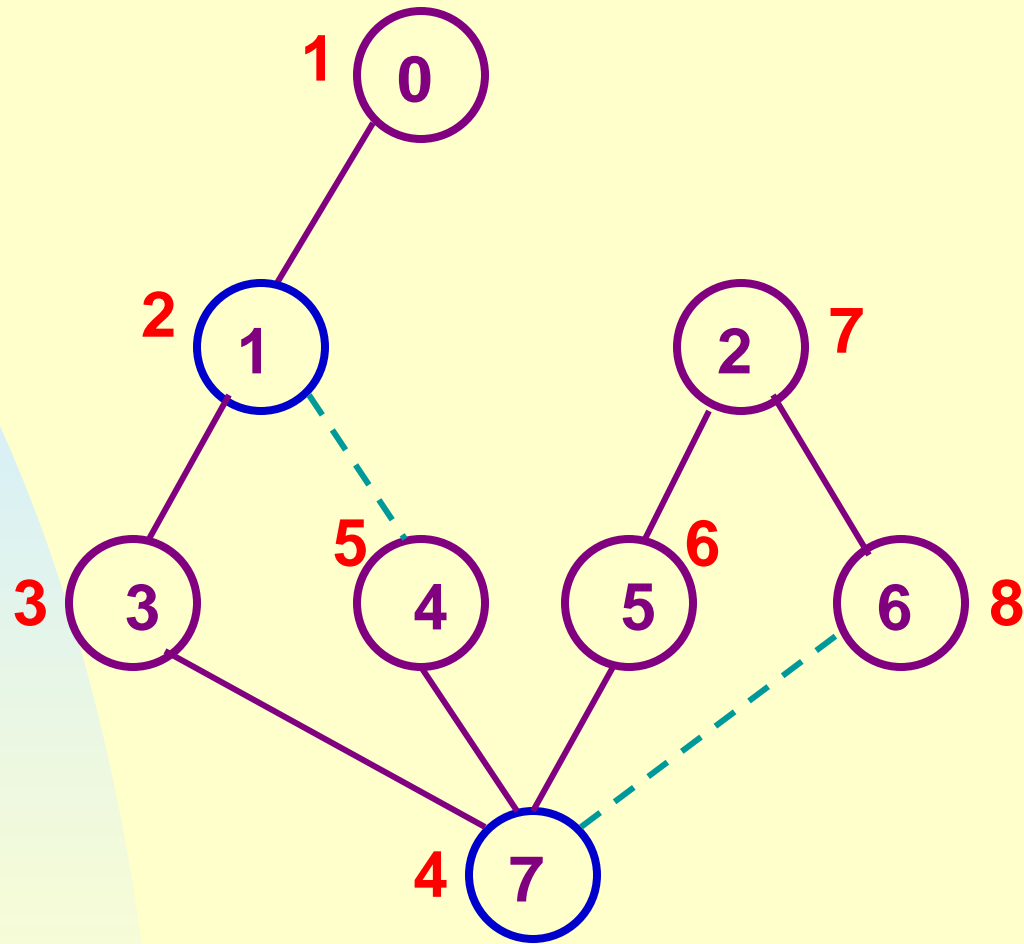


Fig. 6.20 A depth-first spanning tree of Fig. 6.19(a)

The numbers outside the vertices gives the sequence in which the vertices are visited during the depth-first search.

The number is called the **depth-first number**, dfn , of the vertex. For example, $\text{dfn}(1)=2$, $\text{dfn}(7)=4$, $\text{dfn}(6)=8$.

Note that if u is an ancestor of v in the depth-first tree, then $\text{dfn}(u) < \text{dfn}(v)$.

A nontree edge (u, v) is a **back edge** with respect to a spanning tree T iff either u is an ancestor of v or v is an ancestor of u . $(1, 4)$ and $(6, 7)$ are back edges.

A nontree edge that is not a back edge is called a **cross edge**. No graph can have cross edges with respect to any its depth-first spanning trees.

So the root of the depth-first spanning tree is an articulation point iff it has at least two children.

Further, any vertex u is an articulation point iff it has **at least one** child, w , such that it is not possible to reach an ancestor of u using a path composed solely of w , descendants of w , and a single back edge.

To describe these, we define a value **low** for each $w \in V(G)$, as:

$$\text{low}(w) = \min \{ \text{dfn}(w), \\ \min \{ \text{low}(x) \mid x \text{ is a child of } w \}, \\ \min \{ \text{dfn}(x) \mid (w, x) \text{ is a back edge} \} \\ \}$$

Thus u is an articulation point iff u is either the root of the spanning tree and has two or more children or u is not the root and u has a child w such that $\text{low}(w) \geq \text{dfn}(u)$.

vertex	0	1	2	3	4	5	6	7
dfn	1	2	7	3	5	6	8	4
low	1	2	4	2	2	4	4	2

Fig.6.21 dfn and low values for the spanning tree of Fig.6.20

Now we are ready to modify DFS to compute dfn and low as in the function **DfnLow:**

```
void Graph::DfnLow (const int x ) // begin DFS at vertex x
{
    num = 1;           // num is an int data member of Graph
    dfn = new int[n];  // dfn is declared as int * in Graph
    low = new int[n];  // low is declared as int * in Graph
    for ( int i = 0; i < n; i++ ) { dfn[i] = low[i] = 0; }
    DfnLow ( x, -1 ); // x as the root, its parent is dummy -1
    delete [ ] dfn;
    delete [ ] low;
}
```

```

void Graph::DfnLow ( int u, int v )
// compute dfn and low while performing a depth-first search
// beginning at u. v is the parent of u in the resulting spanning
// tree.
{
    dfn[u] = low[u] = num++;
    for ( each vertex w adjacent from u )
    {
        if ( dfn[w] == 0 ) { // w is an unvisited vertex
            DfnLow ( w, u );
            low[u] = min2 ( low[u], low[w] );
            // min2(x,y) return the smaller of x and y.
        }
        else if ( w != v ) low[u] = min2 ( low[u], dfn[w] );
        // back edge. note (v, u) is not a back edge.
    }
}

```


Note that following the return from $\text{DfnLow}(w, u)$,

- $\text{low}[w]$ has been computed.**
- If $\text{low}[w] \geq \text{dfn}[u]$, then u is an articulation point and a new biconnected component has been identified.**
- By using a stack to save edges when they are first encountered, we can output all edges in a biconnected component.**

```
void Graph::Biconnected ( )  
{  
    num = 1;  
    dfn = new int[n];  
    low = new int[n];  
    for ( int i = 0; i < n; i++ ) { dfn[i] = low[i] = 0; }  
    Biconnected ( 0, -1 ); // start at vertex 0  
    delete [ ] dfn;  
    delete [ ] low;  
}
```

```
void Graph::Biconnected ( int u, int v )  
// compute dfn and low, and output the edges of the graph by  
// its biconnected components. v is the parent of u in the  
// resulting spanning tree. S is an initially empty stack declared  
// as a data member of Graph. The function works only for  $n > 1$ .  
// when  $n == 1$ , no edges, but the only vertex should be a  
// biconnected component.  
{  
    dfn[u] = low[u] = num++;  
    for ( each vertex w adjacent from u ) {  
        if (v != w && dfn[w] < dfn[u])  
            // w is not a direct parent of u, see comments later  
            add (u,w) to stack S;
```

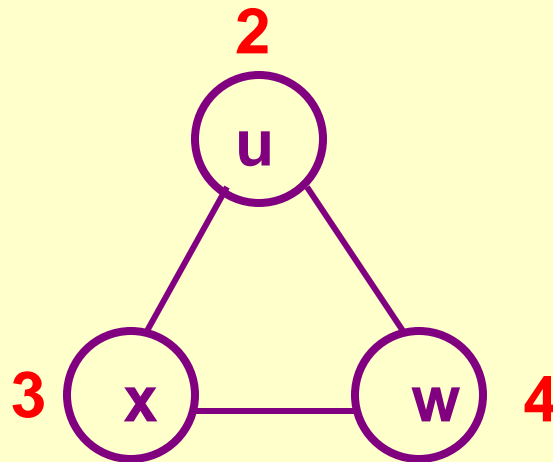
```

if ( dfn[w] == 0 ) { // w is an unvisited vertex
    Biconnected (w, u);
    low[u] = min2 ( low[u], low[w] );
    if ( low[w] >= dfn[u] ) {
        cout<<"New Biconnected Component: " <<endl;
        do {
            delete an edge from stack S;
            let this edge be (x, y);
            cout << x << ", " << y <<endl;
        } while ( (x, y) and (u, w) are not the same edge );
    }
}
else if ( w != v ) low[u] = min2 ( low[u], dfn[w] ); //back edge
}
}

```

Comment:

If ($\text{dfn}[w] > \text{dfn}[u]$) the edge (u, w) must have been added to the stack S as a back edge, as shown below:



When at w , (w, u) has been added to the stack.
When back to u , (u, w) should not be added to the stack.

The time complexity of Biconnected is $O(n+e)$.

Exercises: P356-6, 17, 19

6.3 Minimum-Cost Spanning Trees

The cost of a spanning tree of a weighted, undirected graph is the sum of the costs (weights) of the edges in the spanning tree.

A minimum-cost spanning tree is a spanning tree of least cost.

A design strategy called **greedy method** can be used to obtain a minimum-cost spanning tree.

In the greedy method,

- construct an optimal solution in stages;
- at each stage, make a decision (using some criterion) that appears to be the best (local optimum);
- since the decision can't be changed later, make sure it will result in a feasible solution, i.e., satisfying the constraints.

At the end, if the local optimum is equal to the global optimum, then the algorithm is correct; otherwise, it has produced a **suboptimal** solution.

Fortunately, in the case of constructing minimum-cost spanning tree, the method is correct.

To construct minimum-cost spanning tree, we use a least-cost criterion and the constraints are:

(1) must use only edges within G .

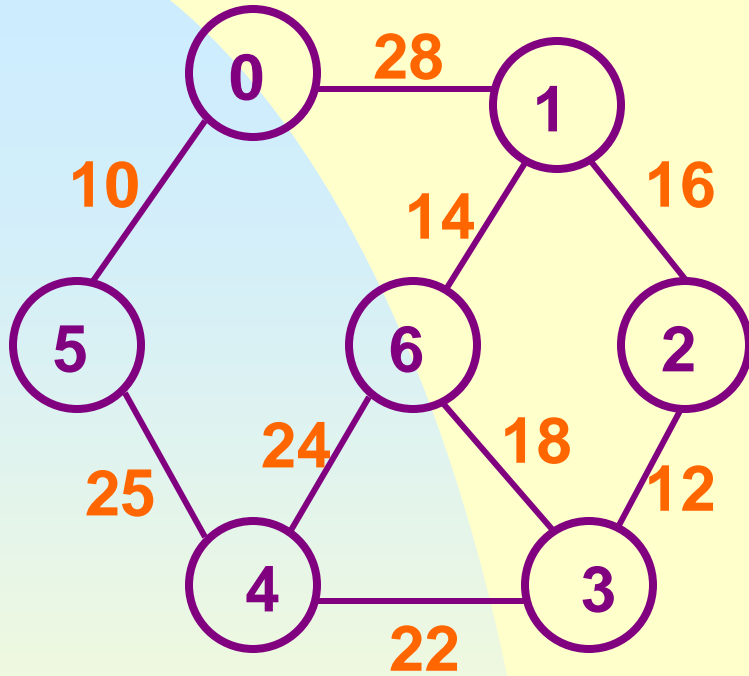
(2) must use exactly $n-1$ edges.

(3) may not use edges that produce a cycle.

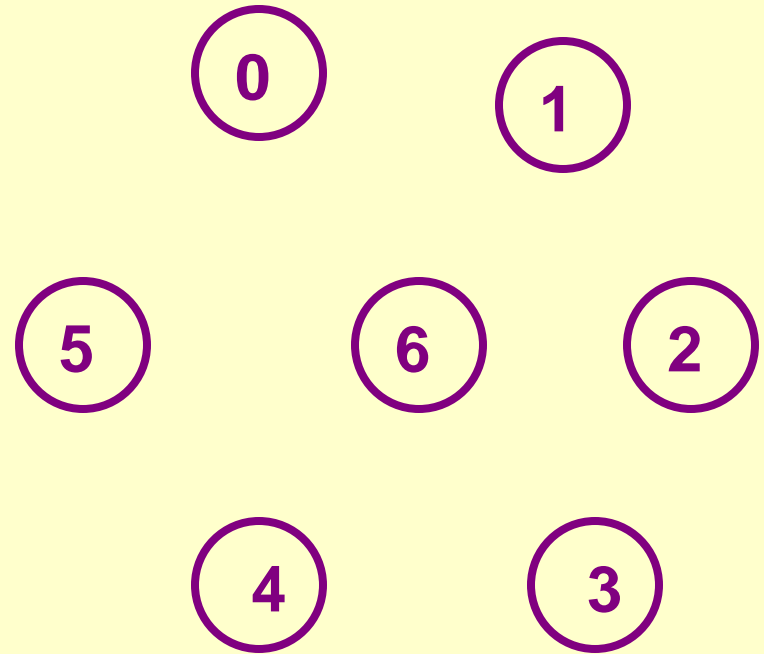
6.3.1 Kruskal Algorithm

- The minimum-cost spanning tree T is built by adding edges to T one at a time.
- Edges are selected for inclusion in T in non-decreasing order of their cost.
- An edge is added to T if it does not form a cycle with the edges already in T .
- Exactly $n-1$ edges will be selected into T .

Example 6.4:

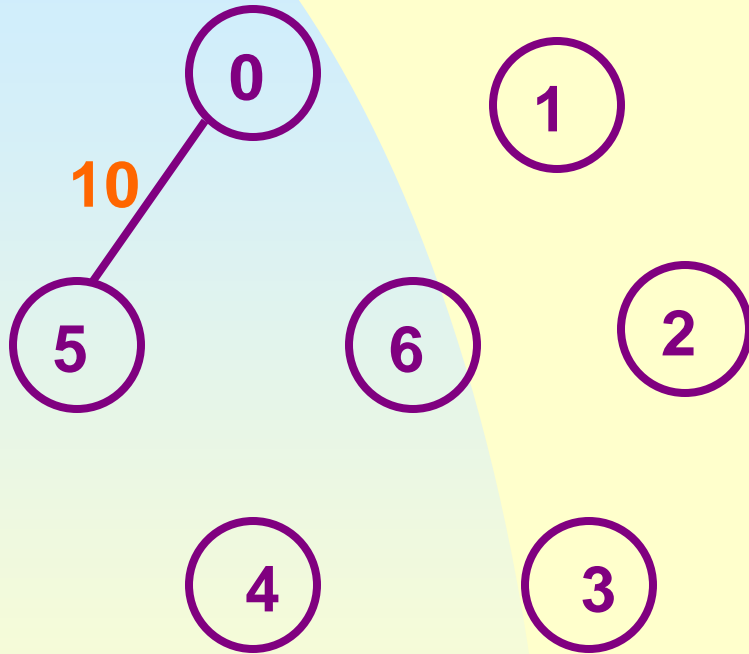


(a) G

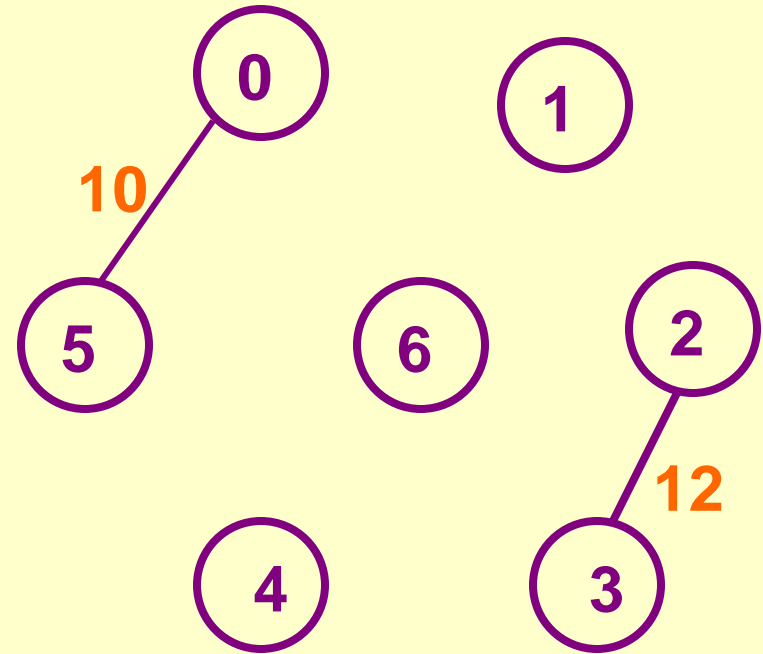


(b)

The cost of remaining edges: **10, 12**, 14, 16, 18, 22, 24, 25, 28

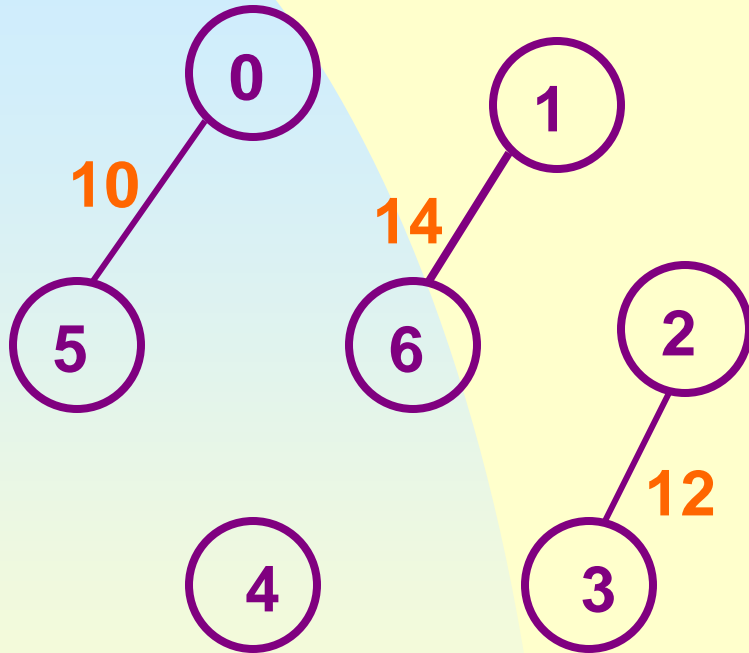


(c)

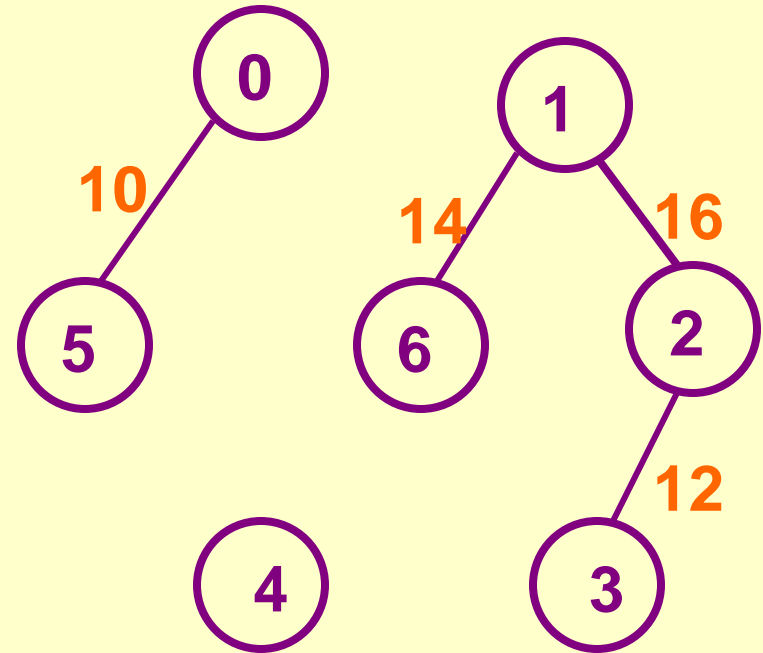


(d)

The cost of remaining edges: 14, 16, 18, 22, 24, 25, 28

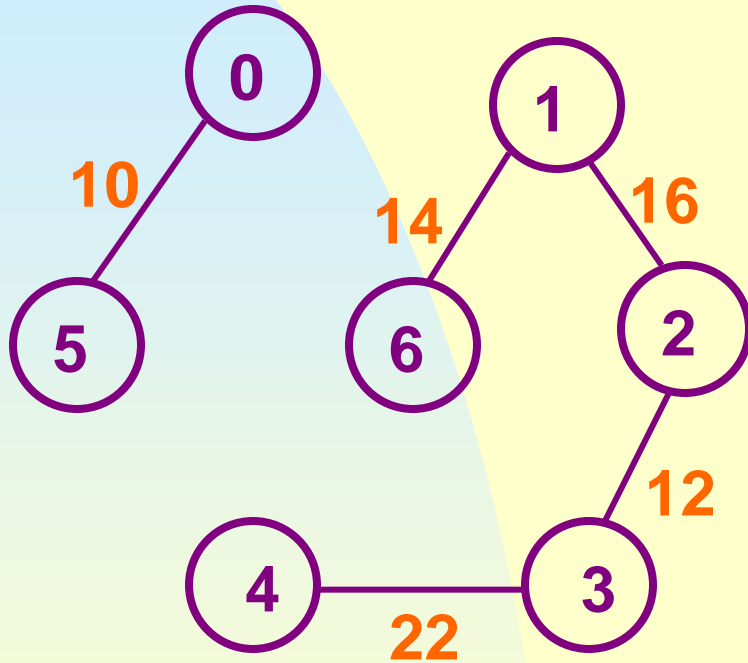


(e)

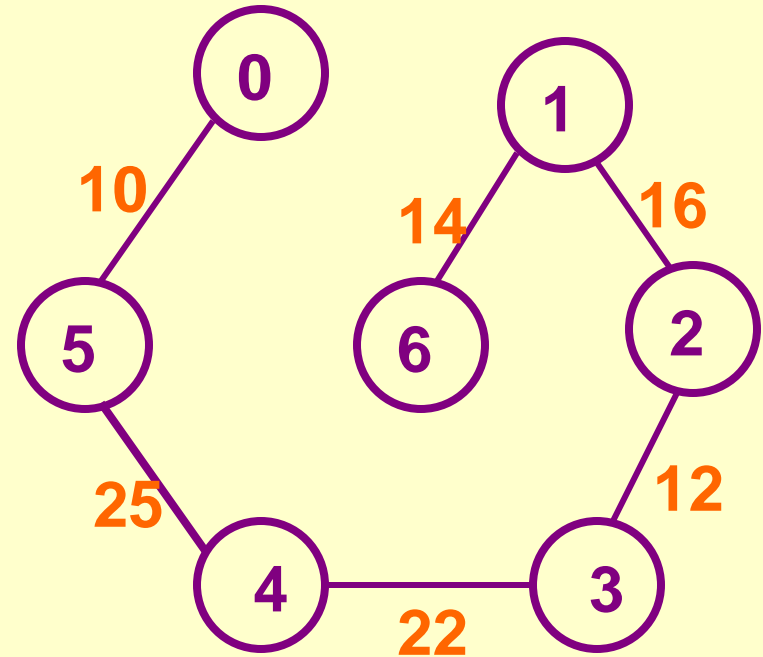


(f)

Remaining edges: 18—(3,6), 22, 24—(4,6), 25, 28



(g) (3,6) discarded



(h) (4, 6) discarded

Given $G = (V, E)$, we have the algorithm:

```
1   $T = \emptyset$ ;  
2  while (( $T$  contains less than  $n-1$  edges) && ( $E$  not Empty)) {  
3      choose an edge  $(v, w)$  from  $E$  of lowest cost;  
4      delete  $(v, w)$  from  $E$ ;  
5      if ( $(v, w)$  does not create a cycle in  $T$ ) add  $(v, w)$  to  $T$ ;  
6      else discard  $(v, w)$ ;  
7  }  
8  if ( $T$  contains fewer than  $n-1$  edges)  
    cout<<"no spanning tree"<<endl;
```


Analysis:

- To perform lines 3 and 4, E can be organized as a min heap, so the next edge can be chosen and deleted in $O(\log e)$. Initialization of the heap takes $O(e)$ (ref. 7.6).
- To perform line 5, vertices in T can be placed into a set using **Union-Find**. The total cost is $O(e \cdot \alpha(e))$.

The total cost: $O(e \log e)$.

Correctness:

Theorem 6.1: Let G be any undirected, connected graph, Kruskal's algorithm generates a minimum-cost spanning tree.

The proof is left as a self-study exercise.

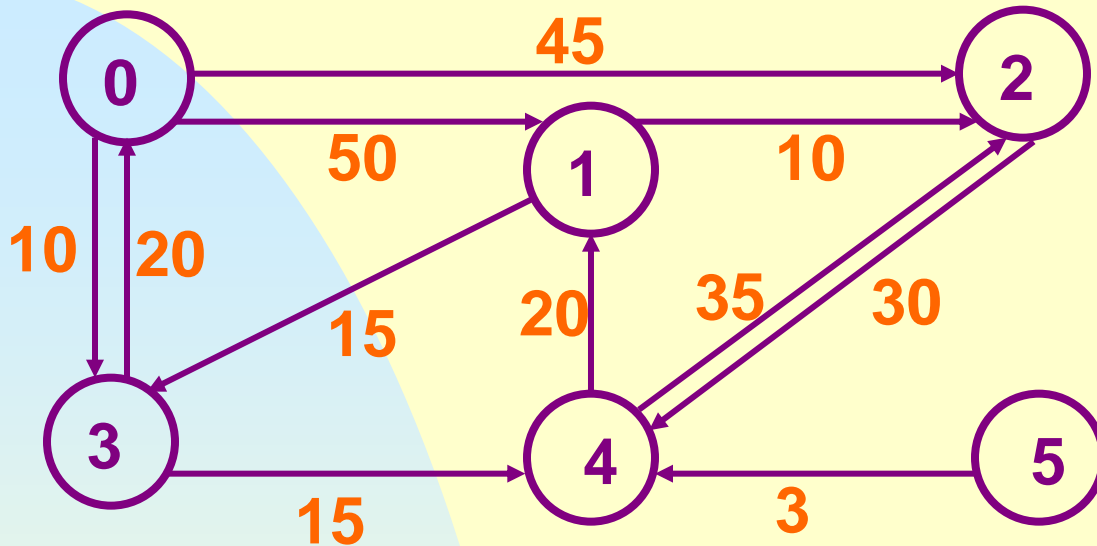
Exercises: P359-1

6.4 Shortest Paths

6.4.1 Single Source/All Destinations: Nonnegative Edge Cost

Problem: given a digraph $G=(V, E)$, a length function $\text{length}(i, j) \geq 0$ for $\langle i, j \rangle \in E(G)$, and a source vertex v , to determine the shortest path from v to all remaining vertices of G .

Example:



(a) G

	path	Length
1)	0, 3	10
2)	0, 3, 4	25
3)	0, 3, 4, 1	45
4)	0, 2	45

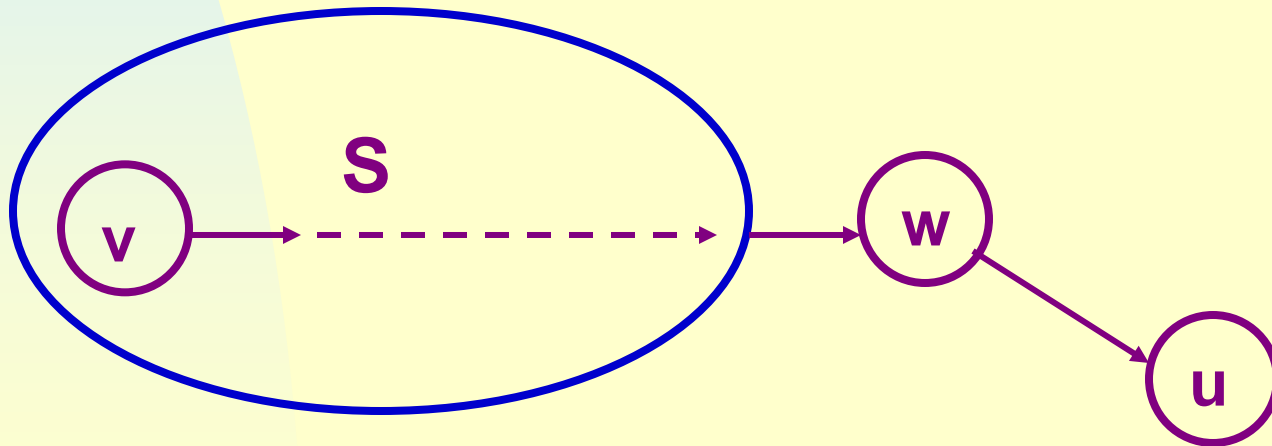
(b) Shortest paths from 0

Let S --- the set of vertices (including v) to which the shortest paths have been found.

For $w \notin S$, $\text{dist}[w]$ --- the length of the shortest path starting from v , going through only the vertices in S , and ending at w .

Assume paths are generated in non-decreasing order of length, observe:

(1) If the next shortest path is to u , then it begins at v , ends at u , and goes through only vertices in S . To prove, assume w on this path is not in S , then the v to u path contains a path from v to w as shown below:

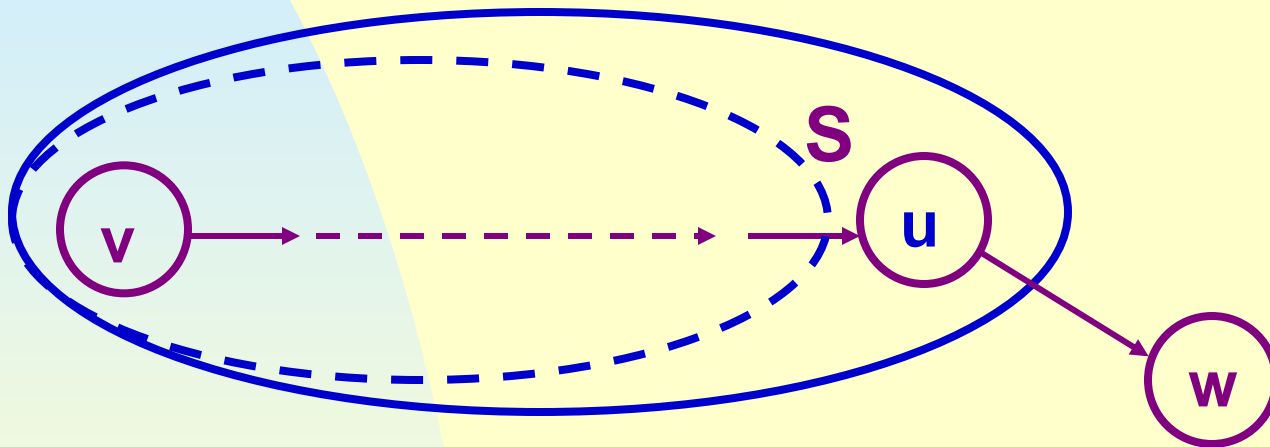


As $\text{length}(w, u) \geq 0$, $\text{length}(v \rightarrow w)$ must be less than $\text{length}(v \rightarrow u)$, otherwise we don't need to go through w . By assumption, the shorter path $(v \rightarrow w)$ has been generated already. Hence there is no intermediate vertex that is not in S .

(2) The destination of the next path generated must be u , such that

$$\text{dist}[u] = \min_{v \notin S} \{ \text{dist}[v] \}$$

(3) Having selected a vertex u as in (2), u becomes a member of S . Now $\text{dist}[w]$, $w \notin S$, may change. If it does, it must be due to a shorter path from v to u and then to w .

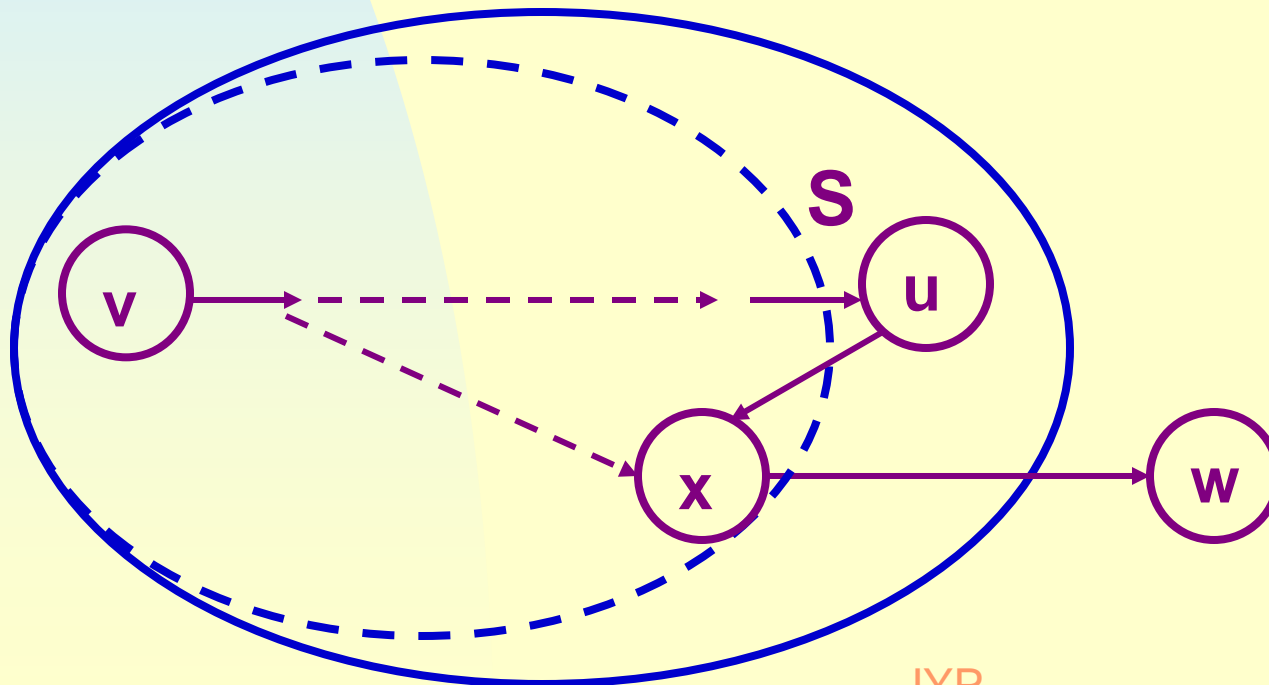


The v to u path must be the shortest v to u path, and the u to w path is just the edge $\langle u, w \rangle$. So if $\text{dist}[w]$ is to decrease, it is because the current $\text{dist}[w] > \text{dist}[u] + \text{length}(u, w)$.

Note the u to w path with any intermediate vertex $x \in S$ cannot affect $\text{dist}[w]$. Because:

- ① $\text{length}(v-u-x) > \text{dist}[x]$
- ② $\text{length}(v-u-x-w) > \text{length}(v-x-w) \geq \text{dist}[w]$

As shown below:



Assume:

- n vertices numbered $0, 1, \dots, n-1$.
- $s[i]=\text{false}$ if $i \notin S$, $s[i]=\text{true}$ if $i \in S$.
- $\text{length}[i][j]$. If $\langle i, j \rangle \notin E(G)$ and $i \neq j$, set $\text{length}[i][j]$ to some large number.

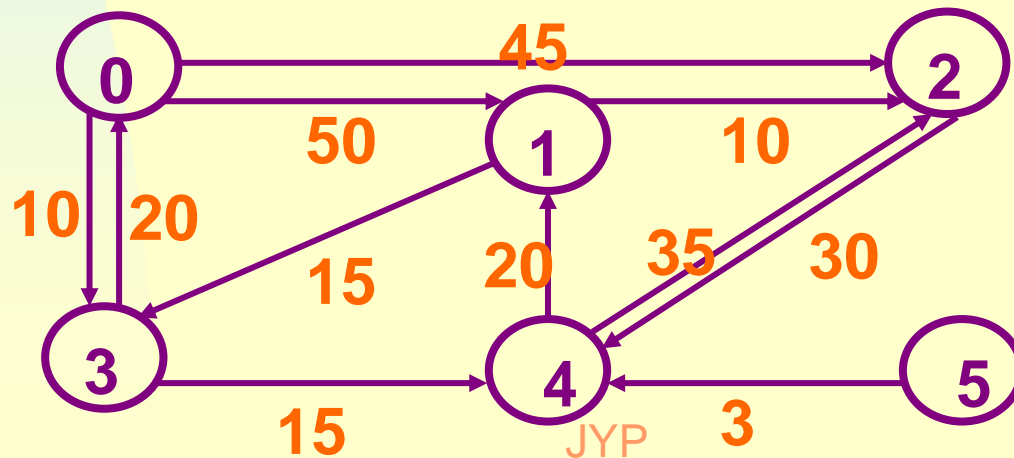
```
class MatrixWDigraph {  
private:  
    double length[NMAX][NMAX]; // NMAX is a constant  
    double *dist;  
    bool *s;  
public:  
    void ShortestPath(const int, const int);  
    int choose(const int);  
};
```

And we assume the constructor will apply space for array **dist** and **s**.

```
1 void MatrixDigraph::ShortestPath(const int n, const int v)
2 { //dist[j], 0 ≤ j < n, is set to the length of the shortest path(v-j)
3 //in a digraph G with n vertices and edge lengths in length[i][j]
4   for (int i=0; i<n; i++) { s[i]=false; dist[i]=length[v][i];}
5   s[v]=true;
6   dist[v]=0;
7   for (i=0; i<n-2; i++) { //determine n-1 paths from v
8     int u=choose(n); //choose returns a value u such that
9       //dist[u]=minimum dist[w], where s[w]=false
10    s[u]= true;
11    for ( int w=0; w<n; w++)
12      if (!s[w] && dist[u]+length[u][w]<dist[w])
13        dist[w]=dist[u]+length[u][w];
14  }
15}
```

The running status of ShortestPath for the example:

vertex	dist					
	0	1	2	3	4	5
0	0	50	45	10	$+\infty$	$+\infty$
3	0	50	45	10	25	$+\infty$
4	0	45	45	10	25	$+\infty$
1	0	45	45	10	25	$+\infty$
2	0	45	45	10	25	$+\infty$

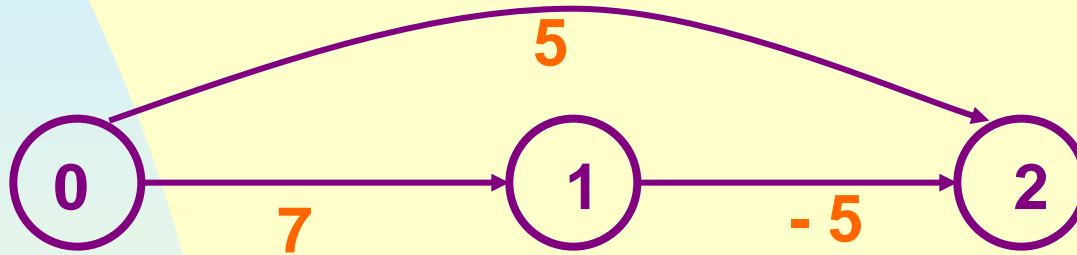


Analysis of ShortestPath:

The time for loop of line 4 is $O(n)$. The for loop of line 7 is executed $n-2$ times, each requires $O(n)$ at line 8 and at line 11 to 13. So the total time is (n^2) .

Even if adjacency lists are used, the overall time for line 11 to 13 can be brought down to $O(e)$, but for line 8 remains $O(n^2)$.

Note the algorithm does not work when some edges may have negative length, as shown below:



6.4.3 All-Pairs Shortest Paths

Problem: find the shortest paths between all pairs of vertices u and v , $u \neq v$.

One solution:

For each vertex in G , takes it as the source, apply ShortestPath, all n times, in $O(n^3)$.

Using the **dynamic programming** approach, we can obtain a conceptually simpler algorithm that has complexity of $O(n^3)$ and works even when G has edges with negative length so long as G has **no cycles with negative length**.

$A^k[i][j]$ --- the length of shortest path from i to j going through no intermediate vertex of index greater than k .

$A^{n-1}[i][j]$ --- the length of the shortest i -- j path in G .

$A^{-1}[i][j]$ --- $\text{length}[i][j]$.

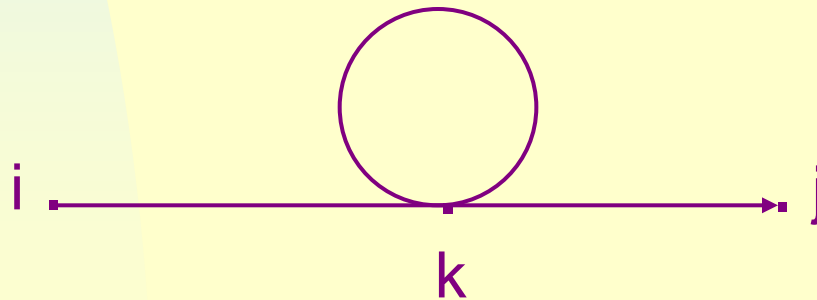
Now the basic idea of the algorithm:

Successively generate $A^{-1}, A^0, A^1, \dots, A^{n-1}$. Given A^{k-1} , we can get A^k by realizing for every pair of i and j , either of the following 2 cases applies:

(1) The shortest path from i to j going through no vertex with index greater than k does not go through k , so its length is $A^{k-1}[i][j]$.

(2) The shortest path does go through k . The path consists of a subpath from i to k and another one from k to j . These must be the shortest paths from i to k and from k to j going through no vertex with index greater than $k-1$, so their lengths are $A^{k-1}[i][k]$ and $A^{k-1}[k][j]$.

Note the above is true only if G has no negative cycle containing k .



From (1) and (2), we have:

$$A^k[i][j] = \min\{A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j]\}, k \geq 0.$$

Assume:

- **G** is represented by length-adjacency matrix as for ShortestPath.
- **double a[NMAX][NMAX]** is data member of **MatrixDigraph**.

```

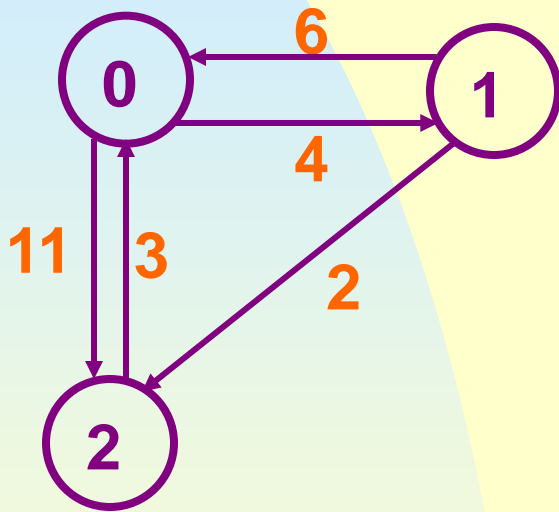
1 void MatrixDigraph::AllLengths (const int n)
2 { //length[n][n] is the adjacency matrix of G with n vertices.
3   //a[i][j] is the length of shortest path from i to j
4   for (int i=0; i<n; i++)
5     for (int j=0; j<n; j++)
6       a[i][j] = length[i][j]; // copy length into a
7   for ( int k=0; k<n; k++) // for a path with highest index k
8     for (i=0; i<n; i++) // for all pairs
9       for (j=0; j<n; j++)
10        if (a[i][k]+a[k][j]<a[i][j]) a[i][j]=a[i][k]+a[k][j];
11}

```

The computation is done **in place** using **a**,
because $A^k[i][k] = A^{k-1}[i][k]$ and $A^k[k][j] = A^{k-1}[k][j]$.

The time is obviously $O(n^3)$.

Example 6.7:



A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

A^0	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

A^1	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

A^2	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

Exercises: P372-1, P373-2, 5, P375-17

6.5 Activity Networks

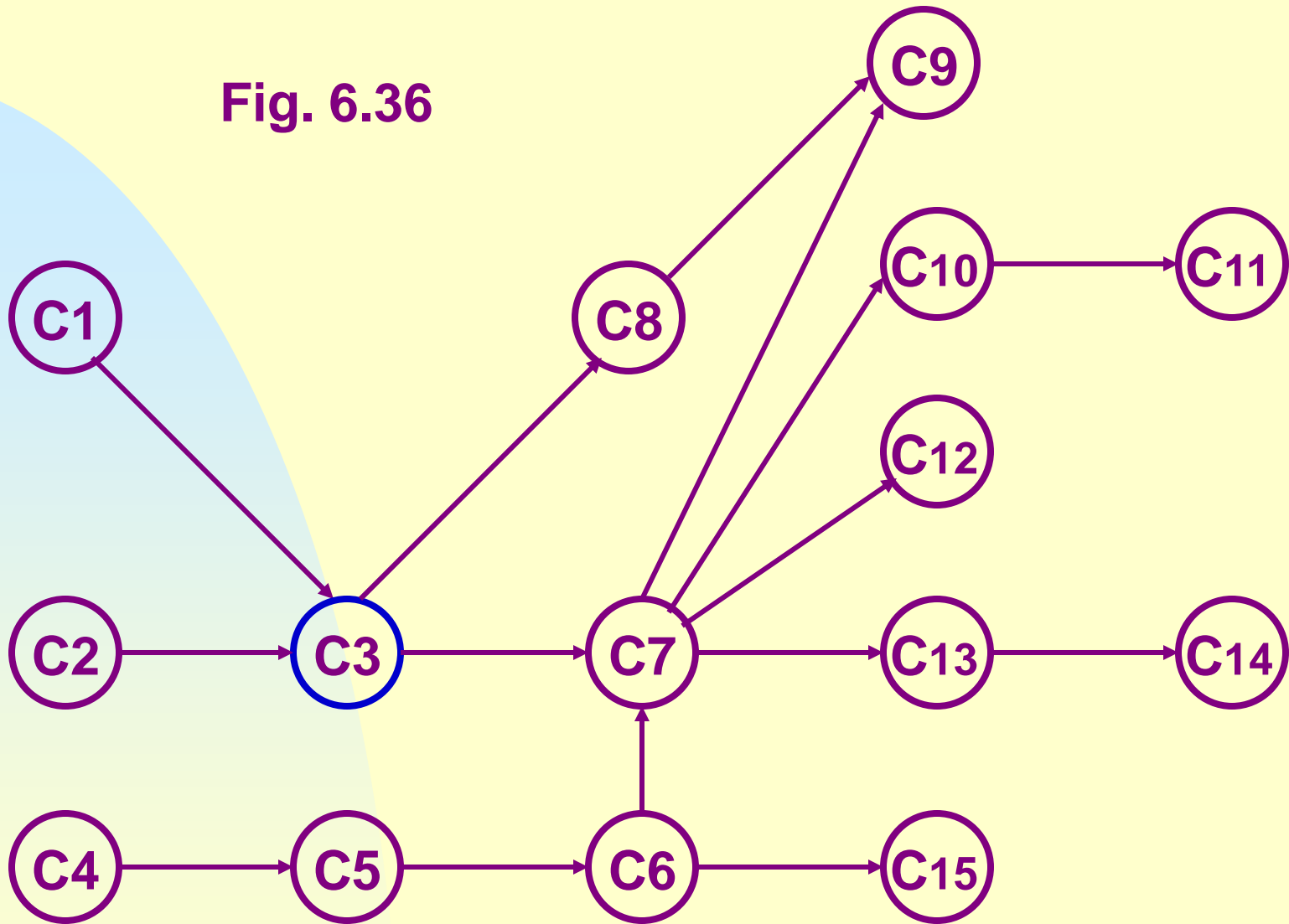
6.5.1 Activity-on-Vertex (AOV) Networks

Definition: A directed graph G in which the vertices represent tasks or activities and the edges represent precedence relations between tasks is an **activity-on-vertex network** or AOV network.

In the following, we'll see the AOV network corresponding to the courses of the next slide.

Course-No.	Course-Name	Prerequisites
C1	Programming I	None
C2	Discrete Mathematics	none
C3	Data Structures	C1, C2
C4	Calculus I	none
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Language	C3
C9	Operating System	C7, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithm	C13
C15	Numerical Analysis	C5

Fig. 6.36



Definition: Vertex i in an AOV network G is a predecessor of j iff there is a directed path from i to j . If $\langle i, j \rangle$ is an edge in G then i is an immediate predecessor of j and j immediate successor of i .

Definition: A precedence relation that is both transitive and irreflexive is a partial order.

A directed graph with no cycle is an acyclic graph.

Given a AOV network G , we are concerned with determining whether or not it is irreflexive, i.e., acyclic.

And we need to generate the topological order of vertices in G .

Definition: A topological order is a linear ordering of vertices of a graph such that, for any two vertices i and j , if i is a predecessor of j in the network, then i precedes j in the linear ordering.

Two possible topological orders of Fig. 6.36 are:
 $C_1, C_2, C_4, C_5, C_3, C_6, C_8, C_7, C_{10}, C_{13}, C_{12}, C_{14}, C_{15}, C_{11}, C_9$
and

$C_4, C_5, C_2, C_1, C_6, C_3, C_8, C_{15}, C_7, C_9, C_{10}, C_{11}, C_{12}, C_{13}, C_{14}$

Topological sorting algorithm

Idea: list a vertex with no predecessor, then delete this vertex together with all edges leading out of it from the network. Repeat the above until all vertices have been listed or all remaining vertices have predecessors.

Formally,

```
1  Input the AOV network, let n be the number of vertices;
2  for (int i=0; i<n; i++) // output the vertices
3  {
4      if (every vertex has a predecessor) return;
5      // network has a cycle and is infeasible.
6      pick a vertex v that has no predecessors;
7      cout << v;
8      delete v and all edges leading out of v from the network;
9  }
```


Example:

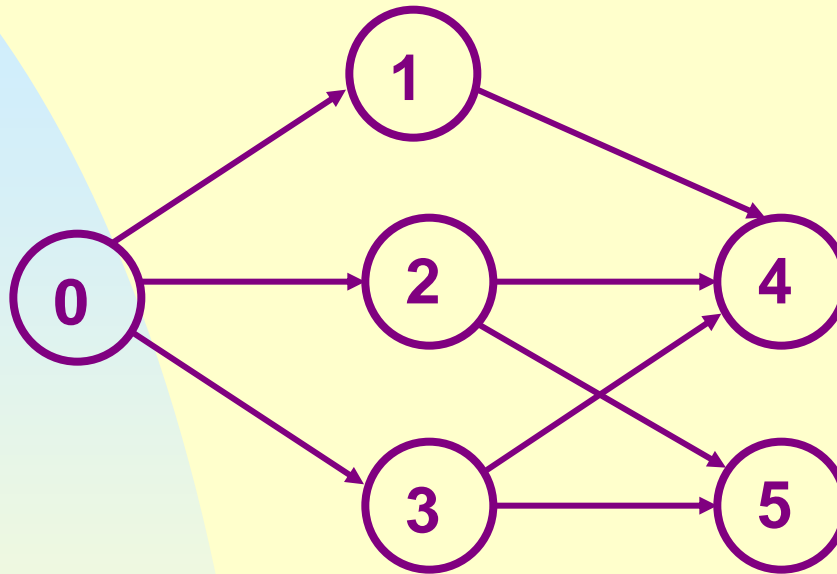
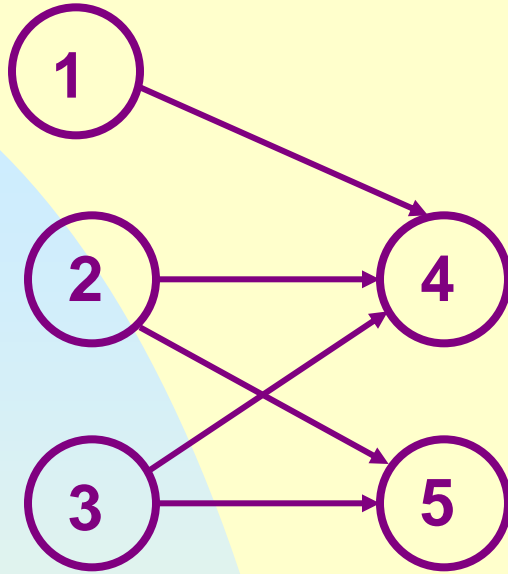
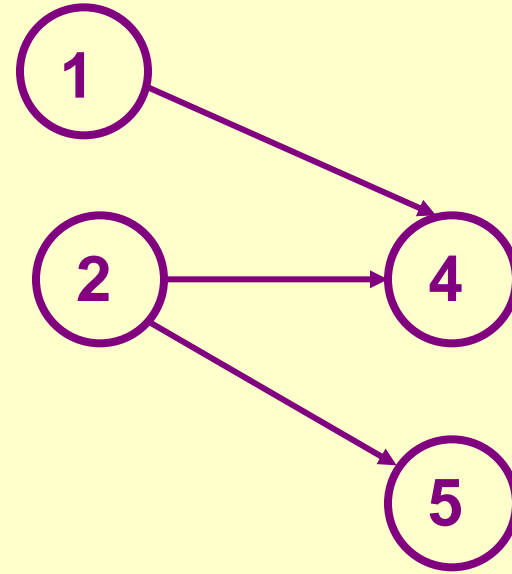


Fig. 6.37
(a) Initial

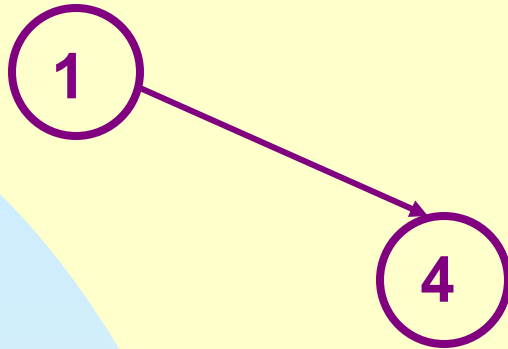


(b) Vertex 0 deleted



(c) Vertex 3 deleted

Topological order generated: **0, 3,**



(d) Vertex 2 deleted



(e) Vertex 5 deleted



(f) Vertex 1 deleted

Topological order generated: 0, 3, 2, 5, 1, 4

The data representation depends on:

(1) Whether a vertex has a predecessor?--- maintain a **count** of the number of immediate predecessors of each vertex.

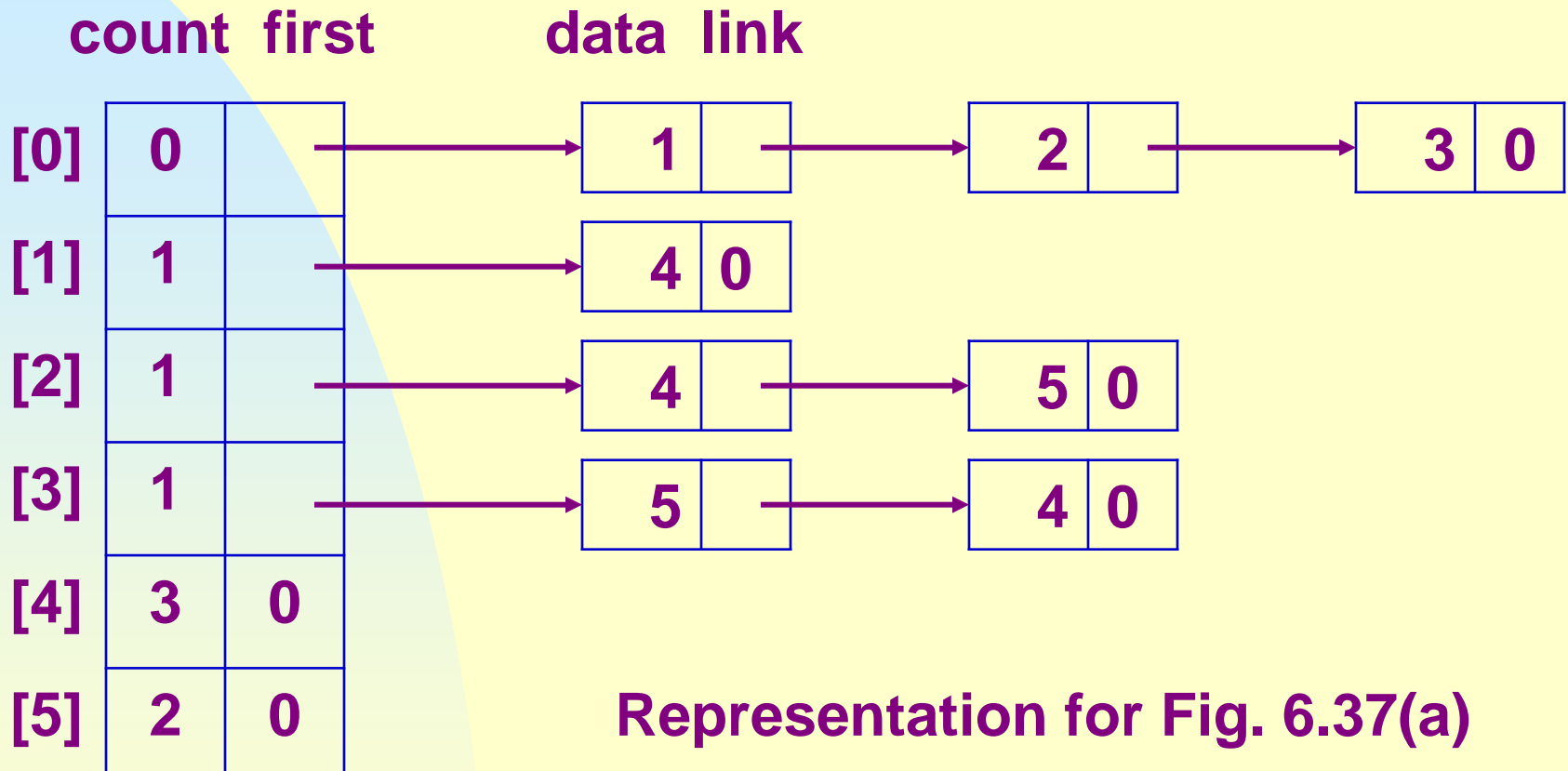
(2) How to delete a vertex together with all its incident edges?--- adjacency lists, and **decrease** the count of all vertices on its adjacency list.

Whenever the count of a vertex drops to 0, that vertex can be placed onto a list of vertices with 0 count.

Hence, we use adjacency lists representation, and assume:

- **int *count** is defined as a data member, and its space is allocated in the constructor.
- **count[i]**, $0 \leq i < n$, has been initialized to the in-degree of vertex *i*. When $\langle i, j \rangle$ is input, the count of *j* is increased by 1.
- the list of vertices with 0 count is maintained as a custom stack linked through the **count** field since it is of no use after the count has become 0.

- **int *t** is defined as a data member, and its space is allocated in the constructor. Vertices are stored in t in topological order for future use.



```
1 void LinkedGraph::TopologicalOrder()
2 { //The n vertices of a network are listed in topological order
3   int top = -1, pos = 0;
4   for (int i=0; i<n; i++) //create a linked stack of vertices with
5     if (count[i]==0) { count[i]=top; top=i;} //no predecessors
6   for (i=0; i<n; i++)
7     if (top== -1) throw "network has a cycle.";
8     int j=top; top=count[top]; //unstack a vertex
9     t[pos++] = j; // store vertex j in topological order
10    Chain<int>::ChainIterator ji=adjLists[j].begin();
11    while (ji != adjLists[j].end()) { // decrease the count of
12      count[*ji]--; // the successor vertices of j
13      if (count[*ji]==0) {count[*ji]=top; top=*ji;} //add to stack
14      ji++; // next successor
15    }
16}
```


Analysis of TopologicalOrder:

Very efficient because of a judicious choice of data structure.

- Lines 4-5 loop: $O(n)$
- Lines 6-10 totally $O(n)$
- The **while** loop of 11-15 takes $O(d_i)$ for each vertex i , where d_i is the out-degree of i . The total time for this part is $O((\sum_{i=0}^{n-1} d_i) + n) = O(e + n)$.

The total time: $O(e + n)$.

6.5.2 Activity-on-Edge (AOE) Networks

The activity-on-edge (AOE) network:

- directed edges --- tasks to be performed
- vertices --- events, signaling the completion of certain activities.
- activities represented by edges leaving a vertex cannot be started until the event at that vertex has occurred.
- an event occurs only when all activities entering it have been completed.

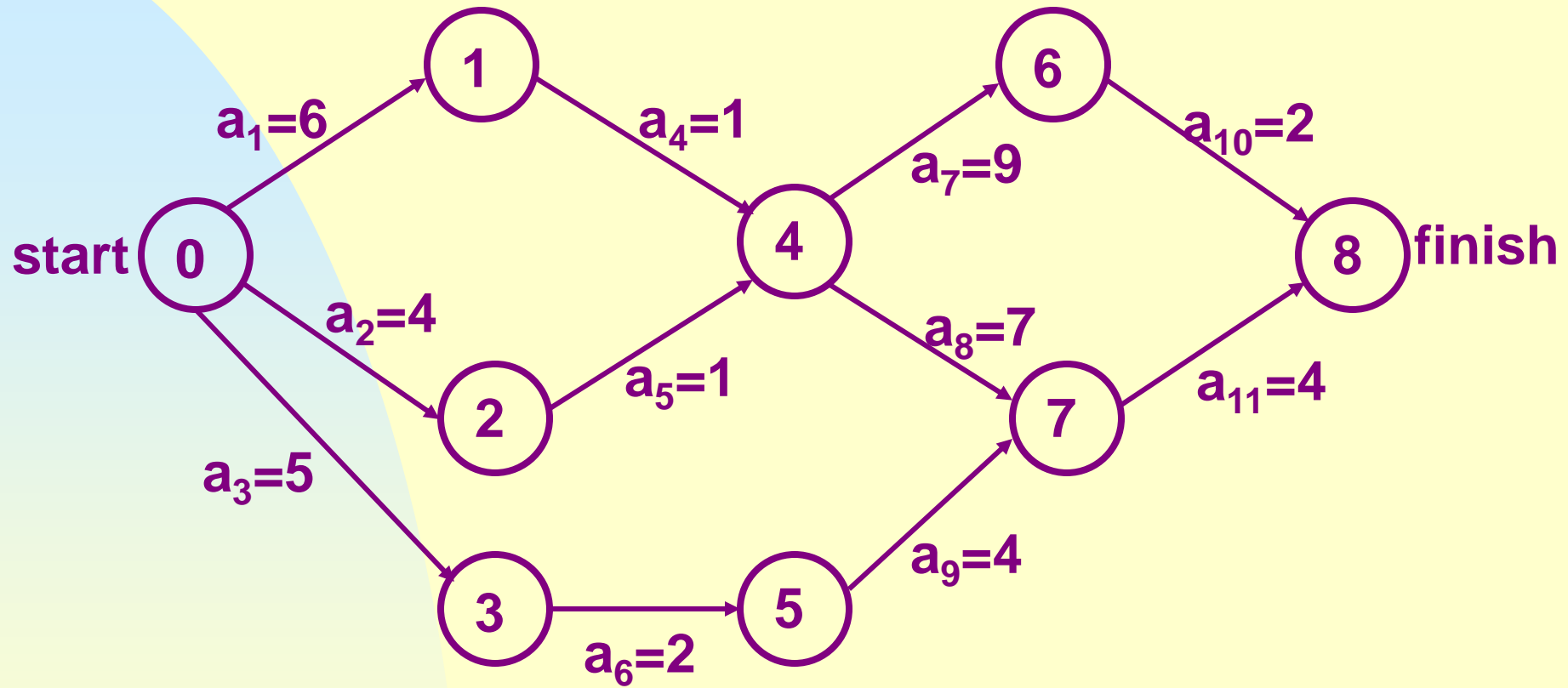


Fig. 6.39

- Since activities in an AOE network can be carried out in parallel, the minimum time to complete the project is the length of the **longest** path from the start to the finish.
- A path of longest length is a **critical path**. e.g., the paths 0, 1, 4, 6, 8 (length 18) and 0, 1, 4, 7, 8.

- $e(i)$ --- the earliest start time for activity a_i .
- $l(i)$ --- the latest start time for activity a_i without increasing the project duration.
- If $e(i) = l(i)$, a_i is called a **critical activity**. $l(i) - e(i)$ is a measure of the criticality of a_i .

- Speed up a critical activity will not necessarily result in a reduced project length unless it is on all critical paths. e.g., speed up a_{11} will not reduce the project length, but a_1 will.
- Once we get $e(i)$ and $l(i)$, critical activities can be easily identified.

6.5.2.1 Calculation of Early Activity Times

How to obtain $e(i)$ and $l(i)$? It is easy to first get:

$ee[j]$ --- the earliest time for event j .

$le[j]$ --- the latest time for event j .

If a_i is edge $\langle k, l \rangle$, then

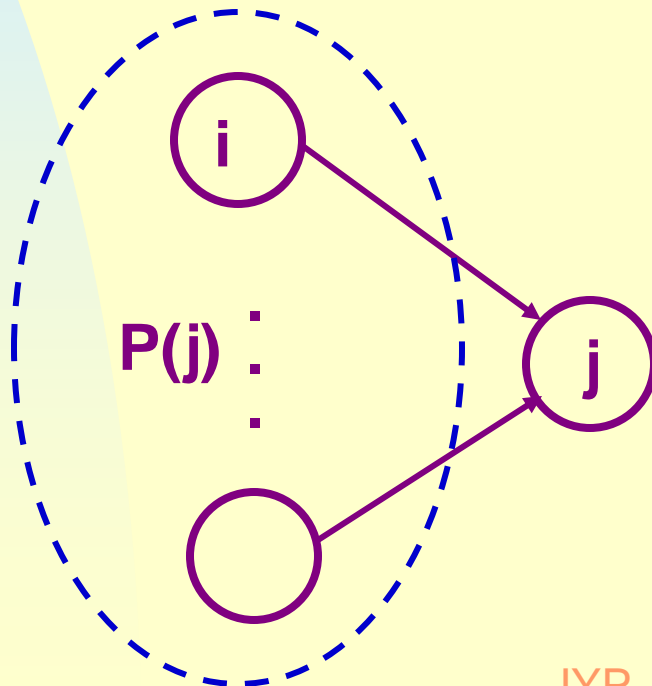
$$\left. \begin{array}{l} e(i) = ee[k] \quad \text{and} \\ l(i) = le[l] - \text{duration of activity } a_i \end{array} \right\} (6.1)$$

$ee[j]$ and $le[j]$ are computed in 2 stages: a forward stage and a backward stage.

The forward stage:

$$\left. \begin{array}{l} ee[0]=0 \quad (\text{suppose } 0 \text{ is the start}) \\ ee[j]= \max_{i \in P(j)} \{ee[i] + \text{duration of } \langle i, j \rangle\} \end{array} \right\} \quad (6.2)$$

where $P(j)$ is the set of all vertices adjacent to j .



To carry the information of **duration**, we use the structure:

```
struct Pair
{
    int vertex;
    int dur;    //activity duration
};
```

```
class LinkedGraph {
private:
    Chain<Pair> *adjLists;
    int *count, *t, *ee, *le;
    int n;
```

public:

```
    LinkedGraph (const int vertices) : {  
        if (vertices < 1) throw "Number of vertices must be > 0";  
        n = vertices;  
        adjLists = new Chain<Pair>[n];  
        count = new int[n]; t = new int[n];  
        ee = new int[n]; le = new int[n];  
    };  
void TopologicalOrder();  
void EarliestEventTime();  
void LatestEventTime();  
void CriticalActivities();  
};
```

If we initialize array ee to 0, and compute in topological order, then the early start times of all predecessors of j would have been computed prior to the computation of $ee[j]$.

Thus we have:

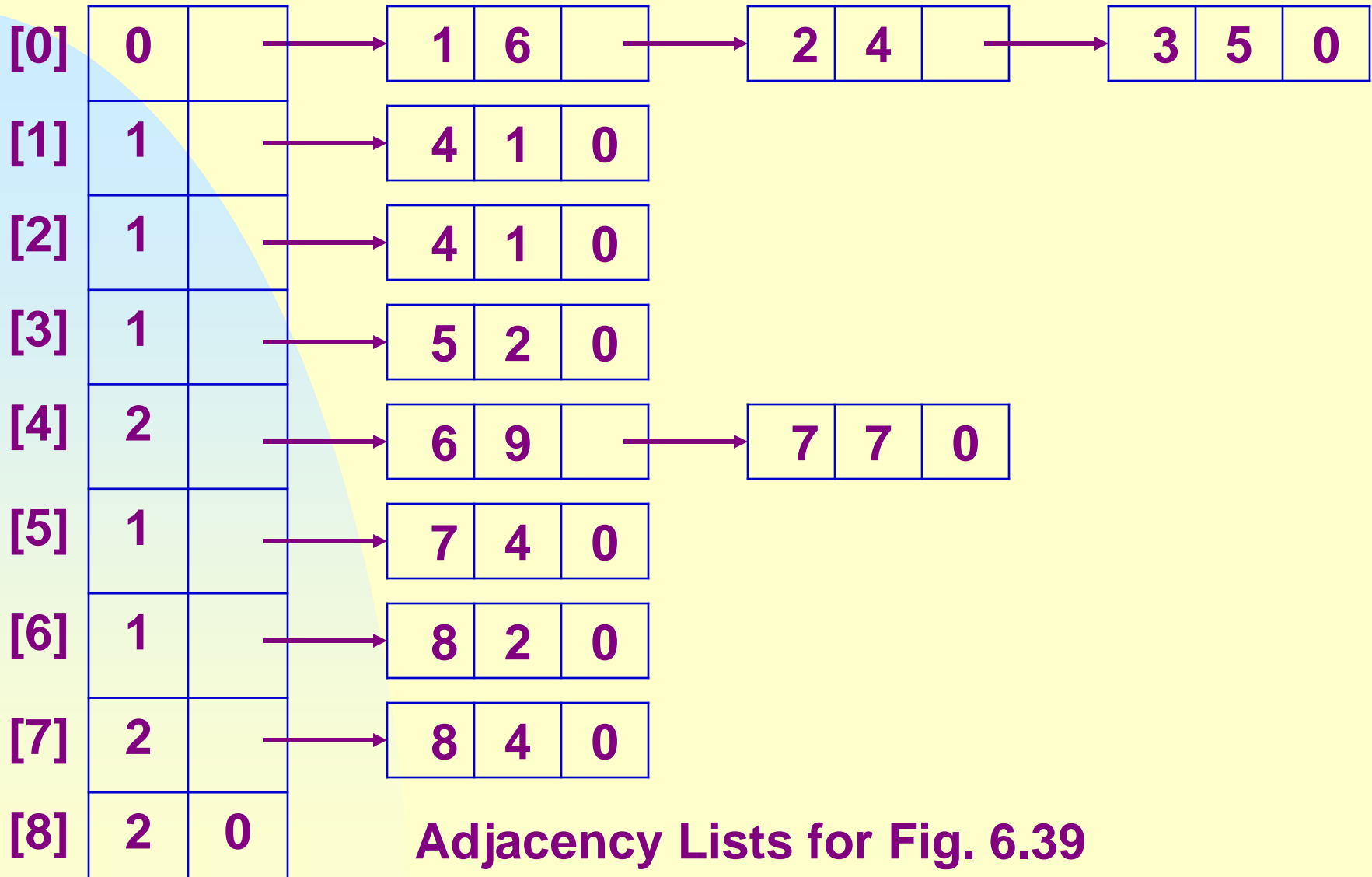
```

void LinkedGraph::EarliestEventTime()
{ // assume a topological order has already been in t,
  // compute ee[j] according to t
  fill(ee, ee+n, 0); // initialize ee
  for (i=0; i<n-1; i++) {
    int j=t[i];
    Chain<Pair>::ChainIterator ji=adjLists[j].begin();
    while (ji!=adjLists[j].end()) {
      int k=ji->vertex; //k is successor of j
      if (ee[k]<ee[j]+ji->dur) ee[k]=ee[j]+ji->dur;
      ji++;
    }
  }
}

```

To illustrate, let's try it out on the network of Fig. 6.39.

count first vertex dur link



Adjacency Lists for Fig. 6.39

ee	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
initial	0	0	0	0	0	0	0	0	0
do 0	0	6	4	5	0	0	0	0	0
do 3	0	6	4	5	0	7	0	0	0
do 5	0	6	4	5	0	7	0	11	0
do 2	0	6	4	5	5	7	0	11	0
do 1	0	6	4	5	7	7	0	11	0
do 4	0	6	4	5	7	7	16	14	0
do 7	0	6	4	5	7	7	16	14	18
do 6	0	6	4	5	7	7	16	14	18

Fig. 6.40

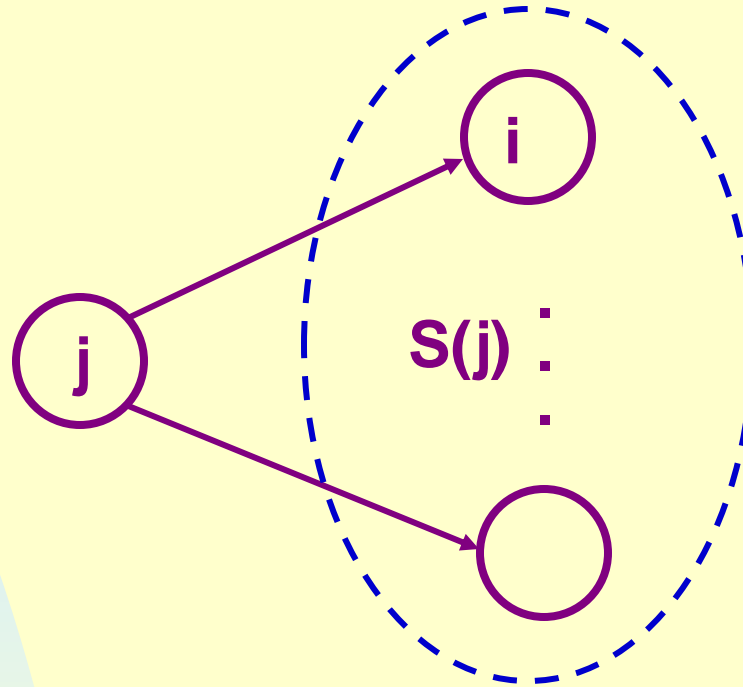
The computing time is obviously $O(n+e)$.

6.5.2.2 Calculation of Late Activity Times

The backward stage:

$$\left. \begin{array}{l} le[n-1]=ee[n-1] \quad (\text{suppose } n-1 \text{ is the finish}) \\ le[j]= \min_{i \in S(j)} \{le[i]-\text{duration of } \langle j, i \rangle\} \end{array} \right\} (6.3)$$

where $S(j)$ is the set of all vertices adjacent from j .



If the forward stage has already been done, and a topological order get, then $le[i]$, $0 \leq i < n$, can be computed directly, using (6.3), by performing the computations in the **reverse** topological order.

Hence we have:

```
void LinkedGraph::LatestEventTime()
{ // assume a topological order has already been in t, ee has
  // been computed, compute le[j] in the reverse order of t
  fill(le, le+n, ee[n-1]); // initialize le
  for (i=n-2; i>=0; i--) {
    int j=t[i];
    Chain<Pair>::ChainIterator ji=adjLists[j].begin();
    while (ji!=adjLists[j].end()) {
      int k=ji->vertex; //k is successor of j
      if (le[k]-ji->dur<le[j]) le[j]=le[k]-ji->dur;
      ji++;
    }
  }
}
```

Example:

given the topological order of Fig. 6.39:

0, 3, 5, 2, 1, 4, 7, 6, 8

we may compute $le[i]$ for $i=8, 6, 7, 4, 1, 2, 5, 3, 0$.

$$le[8] = ee[8] = 18$$

$$le[6] = \min \{le[8] - 2\} = 16$$

$$le[7] = \min \{le[8] - 4\} = 14$$

$$le[4] = \min \{le[6] - 9, le[7] - 7\} = 7$$

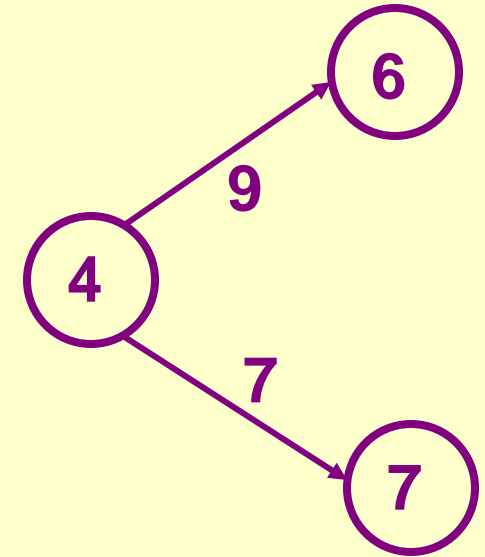
$$le[1] = \min \{le[4] - 1\} = 6$$

$$le[2] = \min \{le[4] - 1\} = 6$$

$$le[5] = \min \{le[7] - 4\} = 10$$

$$le[3] = \min \{le[5] - 2\} = 8$$

$$le[0] = \min \{le[1] - 6, le[2] - 4, le[3] - 6\} = 0$$



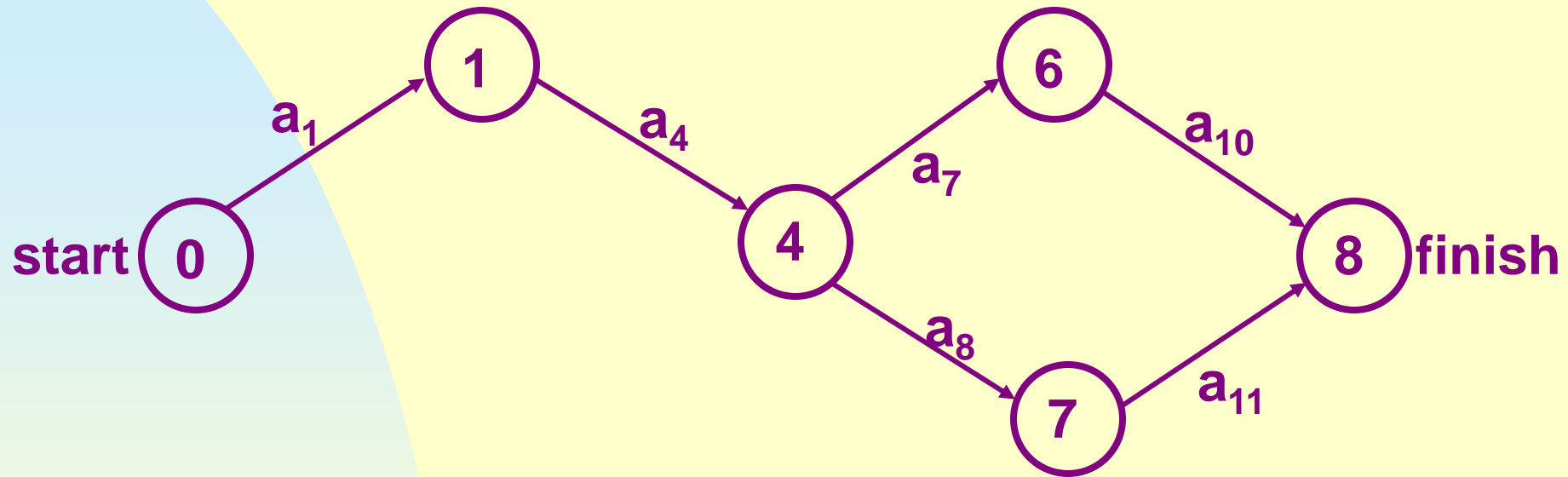
Now we can use (6.1) to output critical activities:

```
void LinkedGraph::CriticalActivities()
{ // compute e[i] and l[i], output critical activities
    int i=1; // the numbering counter for activities
    int u, v, e, l; // e, l are the earliest, latest start time of <u, v>
    for (u=0; u<n; u++) { // scan the adjacency lists.
        Chain<Pair>::ChainIterator ui=adjLists[u].begin();
        while (ui!=adjLists[u].end()) {
            int v=ui->vertex; // <u, v> is an edge numbered i
            e=ee[u]; l=le[v]-ui->dur;
            if (l==e) cout <<"a"<<i<<" "<<u<<" "<<v<<" "<<endl;
            ui++; i++;
        }
    }
}
```

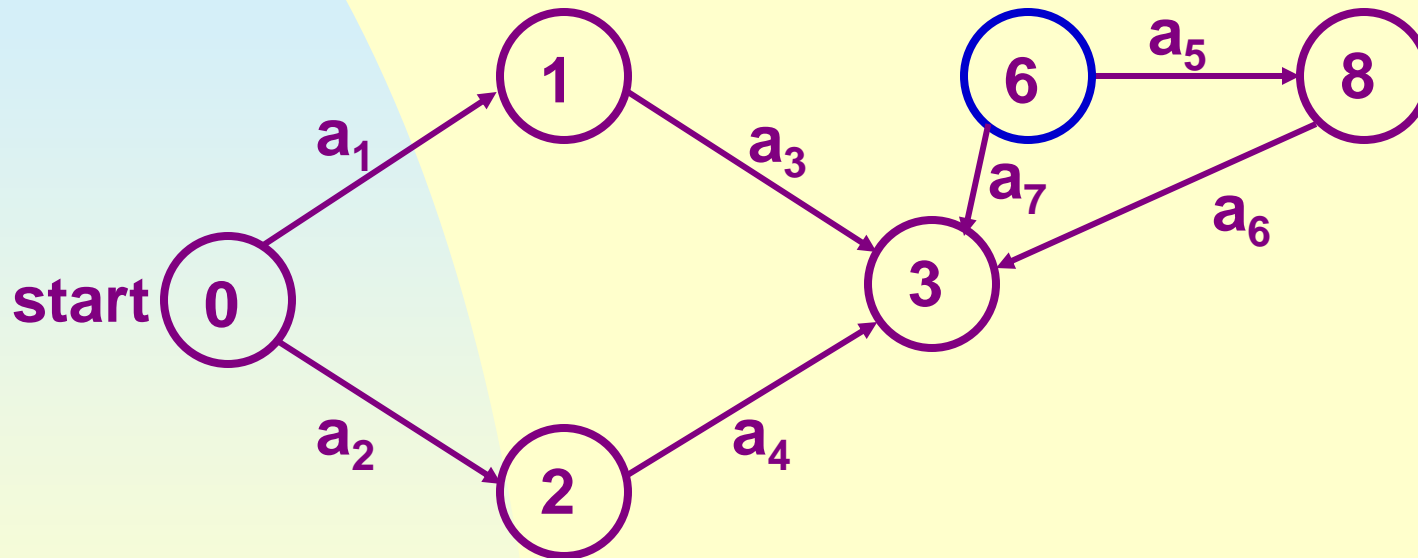
For Fig. 6.39, the $e(i)$ and $l(i)$ are:

activity	e	l	$l-e$	$l-e=0$
$a_1 <0, 1>$	0	0	0	Y
$a_2 <0, 2>$	0	2	2	N
$a_3 <0, 3>$	0	3	3	N
$a_4 <1, 4>$	6	6	0	Y
$a_5 <2, 4>$	4	6	2	N
$a_6 <3, 5>$	5	8	3	N
$a_7 <4, 6>$	7	7	0	Y
$a_8 <4, 7>$	7	7	0	Y
$a_9 <5, 7>$	7	10	3	N
$a_{10} <6, 8>$	16	16	0	Y
$a_{11} <7, 8>$	14	14	0	Y

The critical activities: a_1 , a_4 , a_7 , a_8 , a_{10} , and a_{11} .
Deleting non-critical ones we get:



Suppose vertex 0 is the start, since all activity times are assumed >0 , if finally $ee[i]==0$ ($i \neq 0$) then we can determine that vertex i is not reachable from the start.



Exercises: P389-2, p390-5