Scheme Recursion Processing

Introduction to Data Structures

Kyuseok Shim
ECE, SNU.

SCHEME INTERPRETER PSEUDO CODE (2nd Version)

```
Procedure Main()
begin
1. while (true)
2.     Command := GetCommand()
3.     InitializeTokenizer(command)
4.     root := Read()
5.     result := Eval(root)
6.     PrintResult(result, true)
end
```

SCHEME INTERPRETER PSEUDO CODE (2nd Version)

```
Procedure Preprocessing(newcommand)
begin
1. // newcommand is an empty string when this procedure is first called
2. while (token := GetNextToken()) is not empty
3.    if token is “define”
4.       newcommand := Concatenate(newcommand, “define”)
5.       token := GetNextToken()
6.       if token is “(”
            // (define (square x) ( * x x )) ==>
            // (define square (lambda (x) ( * x x )))
7.          token := GetNextToken()
8.         newcommand := Concatenate(newcommand, token,
9.                            “(lambda(”, Preprocessing(newcommand), “)” )
```

SCHEME INTERPRETER PSEUDO CODE (2nd Version)

```
Procedure Preprocessing()
…
elseif token is "'"
     // '(a b c) ==> (quote (a b c))
     newcommand := Concatenate(newcommand, "(quote")
     number_of_left_paren := 0
     do
        token := GetNextToken()
        newcommand := Concatenate(newcommand, token)
        if token is "("
           number_of_left_paren := number_of_left_paren+1
        elseif token is ")"
           number_of_left_paren := number_of_left_paren−1
     while (number_of_left_paren>0)
     newcommand := Concatenate(newcommand, ")" )
else newcommand := Concatenate(newcommand, token)
return newcommand
end
```

SCHEME INTERPRETER PSEUDO CODE (2nd Version)

```
Procedure Eval(root)
begin
1. tokenindex := GetHashValue(Memory[root].lchild)
2. if (token index = PLUS)
3.    return GetHashValue(GetVal(Eval(Memory[Memory[root].rchild].lchild))
4.                 + GetVal(Eval(Memory[Memory[Memory[root].rchild].rchild].lchild)))
…
11. elseif (token index = isEQ) // eq?
12.    return Eval(Memory[Memory[root].rchild].lchild
13.                = Eval(Memory[Memory[Memory[root].rchild].rchild].lchild)
14. elseif (token index = isEQUAL) // equal?
15.    return CheckStructure(Eval(Memory[Memory[root].rchild].lchild),
16.                          Eval(Memory[Memory[Memory[root].rchild].rchild].lchild))
```

SCHEME INTERPRETER PSEUDO CODE (2nd Version)

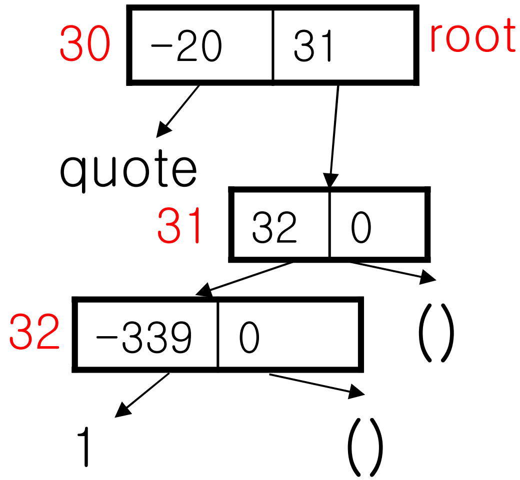
```
Procedure Eval(root)
…
elseif (token index = isNUMBER)
  if IsNumber(Eval(Memory[Memory[root].rchild].lchild)) is true
     return GetHashValue("#t")
  else return GetHashValue("#f")

elseif (token index = isSYMBOL)
  if result := EVAL(Memory[Memory[root].rchild].lchild) is true and IsNumber(result) is false
     return GetHashValue("#t")
  else return GetHashValue("#f")

elseif (token index = isNULL)
  if Memory[root].rchild is NIL or Eval(Memory[root].rchild) is NIL
     return GetHashValue("#t")
  else return GetHashValue("#f")
```

SCHEME INTERPRETER PSEUDO CODE (2nd Version)

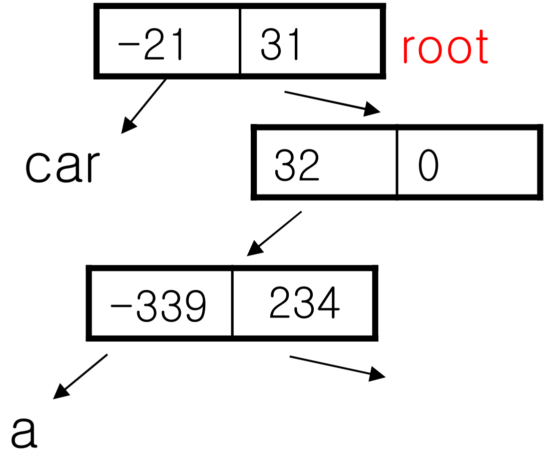
```
Procedure Eval(root)
…
elseif (token index = CONS)
  newmemory := Alloc()
  Memory[newmemory].lchild := Eval(Memory[Memory[root].rchild].lchild)
  Memory[newmemory].rchild := Eval(Memory[Memory[Memory].root].rchild].rchild].lchild)
  return newmemory

elseif (token index = COND)
  while Memory[Memory[root].rchild].rchild is not NIL
    root := Memory[root].rchild
    if (EVAL(Memory[Memory[root].lchild].lchild) = TRUE)
      return EVAL(Memory[Memory[root].lchild].rchild)
 if Memory[Memory[Memory[root].rchild].lchild].lchild is not ELSE
   Error()
 return Eval(Memory[Memory[Memory[Memory[root].rchild].lchild].rchild].lchild)
```

SCHEME INTERPRETER PSEUDO CODE (2nd Version)

```
Procedure Eval(root)
…
elseif (token index = CAR)
  return Memory[EVAL(Memory[Memory[root].rchild].lchild)].lchild

elseif (token index = CDR)
  return Memory[EVAL(Memory[Memory[root].rchild].lchild)].rchild

elseif (token index = DEFINE)
  if function define
     hashTable[Memory[Memory[root].rchild].lchild].pointer :=
                    Eval(Memory[Memory[Memory[root].rchild].rchild].lchild)
  else hashTable[Memory[Memory[root].rchild].lchild].pointer :=
                    EVAL(Memory[Memory[root].rchild].rchild)

  elseif (token index = QUOTE)
     return Memory[Memory[root].rchild].lchild
```



SCHEME INTERPRETER PSEUDO CODE (2nd Version)

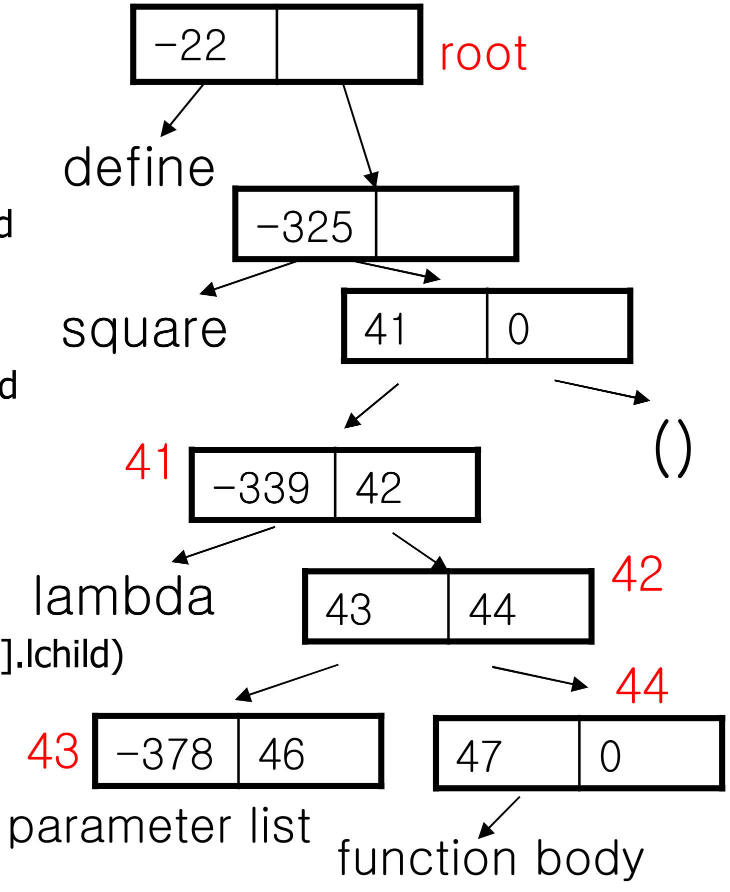



```
Procedure Eval(root)
…
42. elseif (token index = CAR)
43.   return Memory[EVAL(Memory[Memory[root].rchild].lchild)].lchild

44. elseif (token index = CDR)
45.   return Memory[EVAL(Memory[Memory[root].rchild].lchild)].rchild

46. elseif (token index = DEFINE)
47.   if function define
48.     hashTable[Memory[Memory[root].rchild].lchild].pointer :=
49.                  Eval(Memory[Memory[Memory[root].rchild].rchild].lchild)
50.   else hashTable[Memory[Memory[root].rchild].lchild].pointer :=
51.                  EVAL(Memory[Memory[root].rchild].rchild)

52.   elseif (token index = QUOTE)
53.     return Memory[Memory[root].rchild].lchild
```

()

SCHEME INTERPRETER PSEUDO CODE (2nd Version)

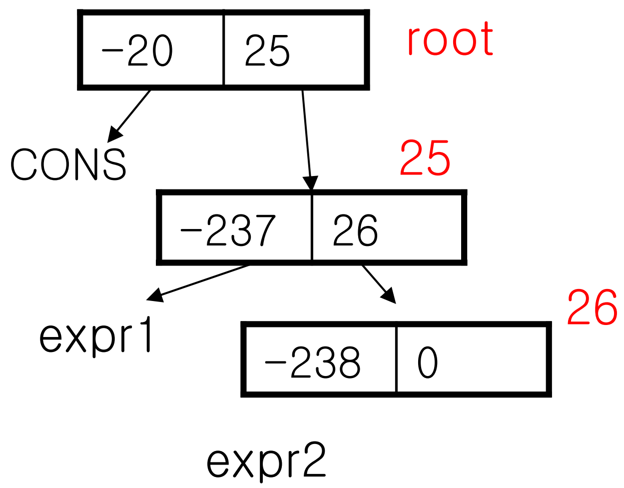
```
Procedure Eval(root)
…
42. elseif (token index = CAR)
43.   return Memory[EVAL(Memory[Memory[root].rchild].lchild)].lchild

44. elseif (token index = CDR)
45.   return Memory[EVAL(Memory[Memory[root].rchild].lchild)].rchild

46. elseif (token index = DEFINE)
47.   if function define
48.      hashTable[Memory[Memory[root].rchild].lchild].pointer :=
49.                     Eval(Memory[Memory[Memory[root].rchild].rchild].lchild)
50.   else hashTable[Memory[Memory[root].rchild].lchild].pointer :=
51.                     EVAL(Memory[Memory[root].rchild].rchild)

52.   elseif (token index = QUOTE)
53.      return Memory[Memory[root].rchild].lchild
```



SCHEME INTERPRETER PSEUDO CODE (2nd Version)

Procedure Eval(root)

```
…
elseif (token index = CONS)
    newmemory := Alloc()
    Memory[newmemory].lchild := Eval(Memory[Memory[root].rchild].lchild)
    Memory[newmemory].rchild := Eval(Memory[Memory[Memory].root].rchild].rchild].lchild)
    return newmemory
```

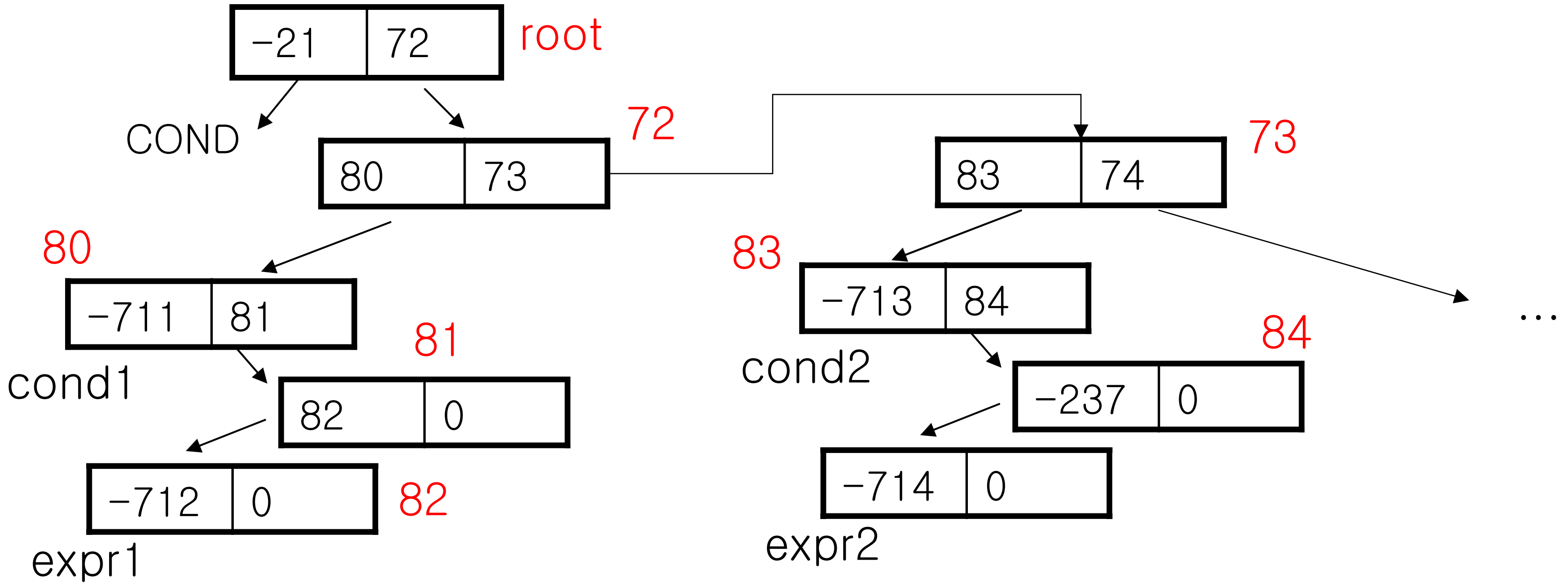


(CONS expr1 expr2)

SCHEME INTERPRETER PSEUDO CODE (2nd Version)

```
elseif (token index = COND)
  while Memory[Memory[root].rchild].rchild is not NIL
    root := Memory[root].rchild
    if (EVAL(Memory[Memory[root].lchild].lchild) = TRUE)
      return EVAL(Memory[Memory[root].lchild].rchild)
if Memory[Memory[Memory[root].rchild].lchild].lchild is not ELSE
   Error()
return Eval(Memory[Memory[Memory[Memory[root].rchild].lchild].rchild].lchild)
```

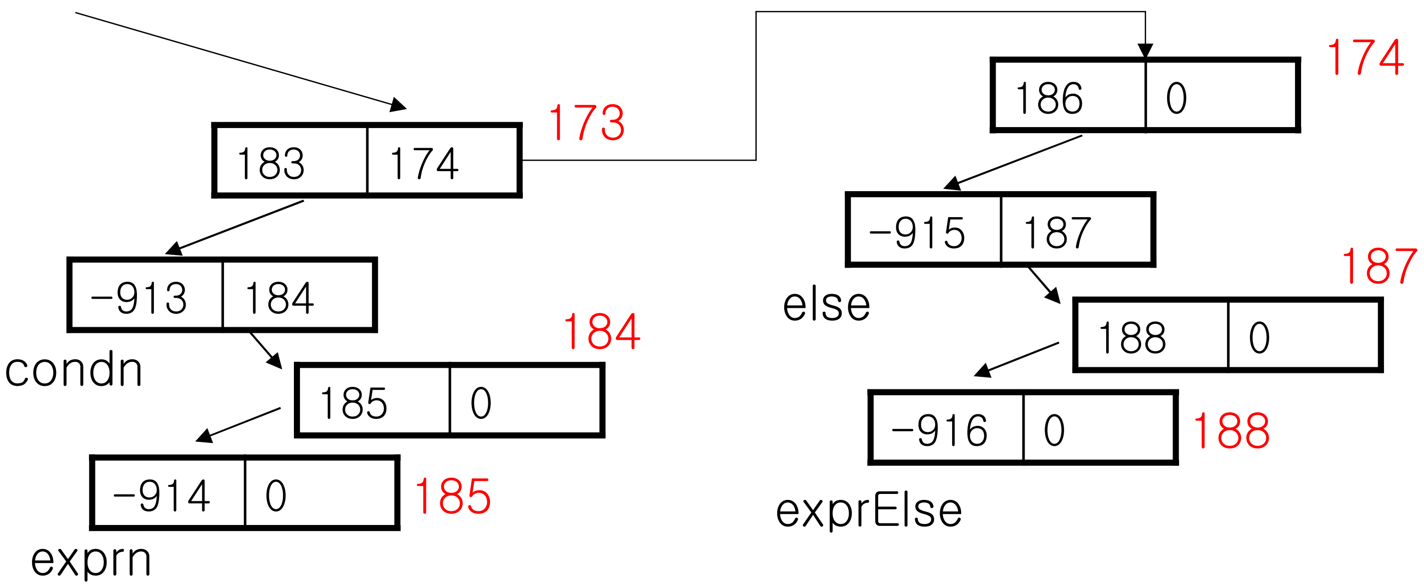
```
(COND ((cond1) (expr1))
      ((cond2) (expr2))
      ...
      ((condn) (exprn))
      ((else) (exprElse)))
```



SCHEME INTERPRETER PSEUDO CODE (2nd Version)

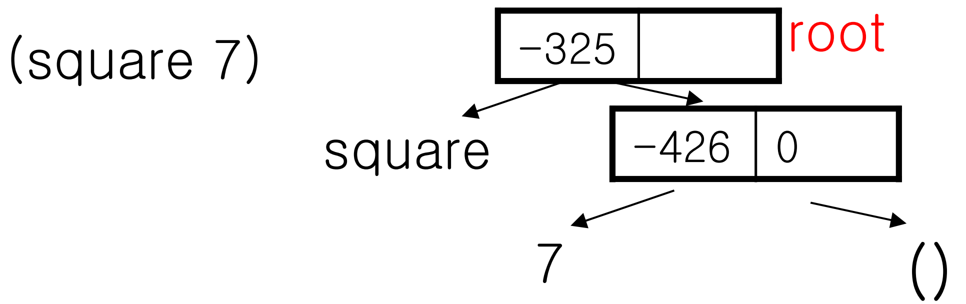
```
elseif (token index = COND)
  while Memory[Memory[root].rchild].rchild is not NIL
    root := Memory[root].rchild
    if (EVAL(Memory[Memory[root].lchild].lchild) = TRUE)
      return EVAL(Memory[Memory[root].lchild].rchild)
if Memory[Memory[Memory[root].rchild].lchild].lchild is not ELSE
    Error()
return Eval(Memory[Memory[Memory[Memory[root].rchild].lchild].rchild].lchild)
```

```
(COND ((cond1) (expr1))
      ((cond2) (expr2))
      …
      ((condn) (exprn))
      ((else) (exprElse)))
```

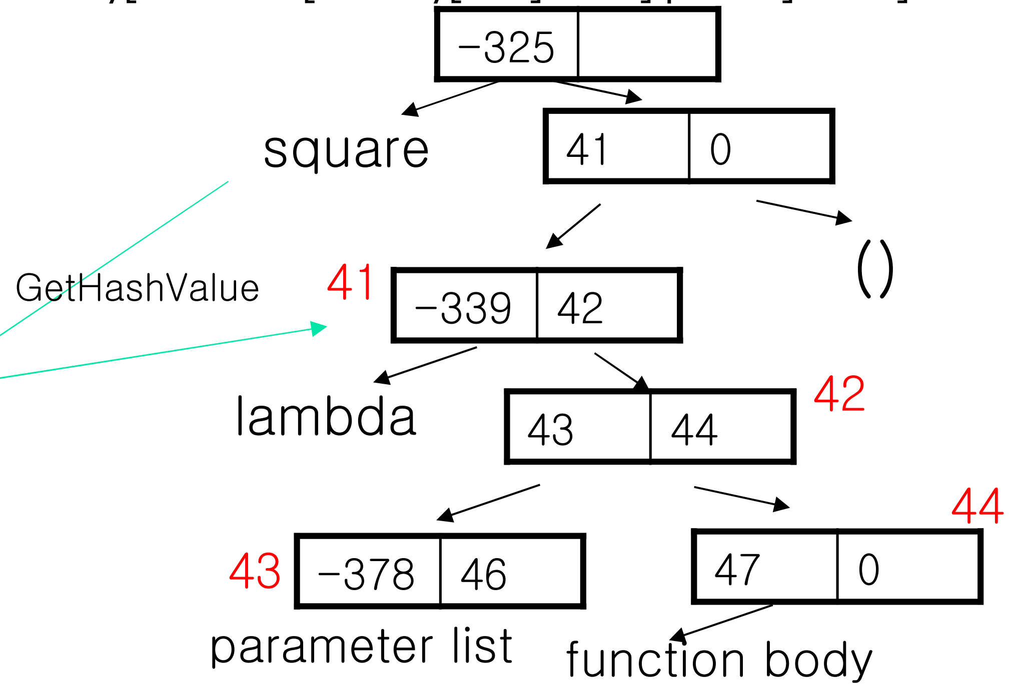


SCHEME INTERPRETER PSEUDO CODE (2nd Version)

```
Procedure Eval(root)
…
elseif token index is user defined function
    push current values to stack
    set parameter by function argument
    result := Eval(Memory[Memory[Memory[hashTable[Memory[root].lchild].pointer].rchild].rchild].lchild)
    pop the values from stack
return result
```



Hash Value	Symbol	Link of Value
…		
…		
-325	square	41
…		
-426	7	0
…		
…		

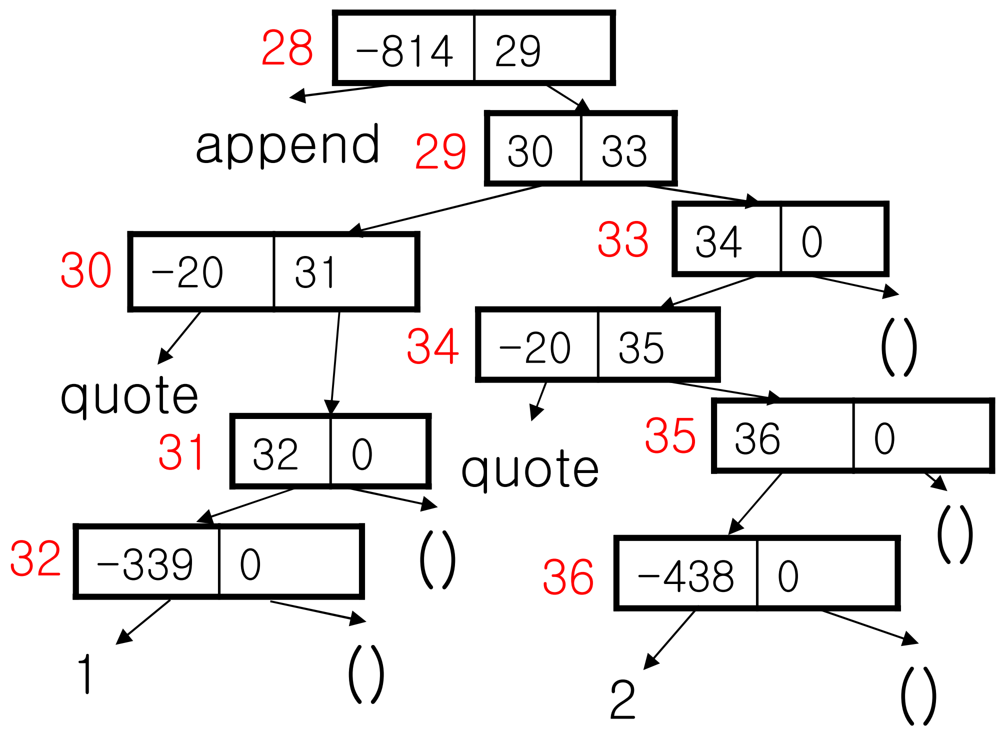


Append a list to another list

```
(define (append L R)
  (cond ((null? L) R)
    (else (cons (car L) (append (cdr L) R)))))
```

- (append '(1) '(2)) : (1 2)

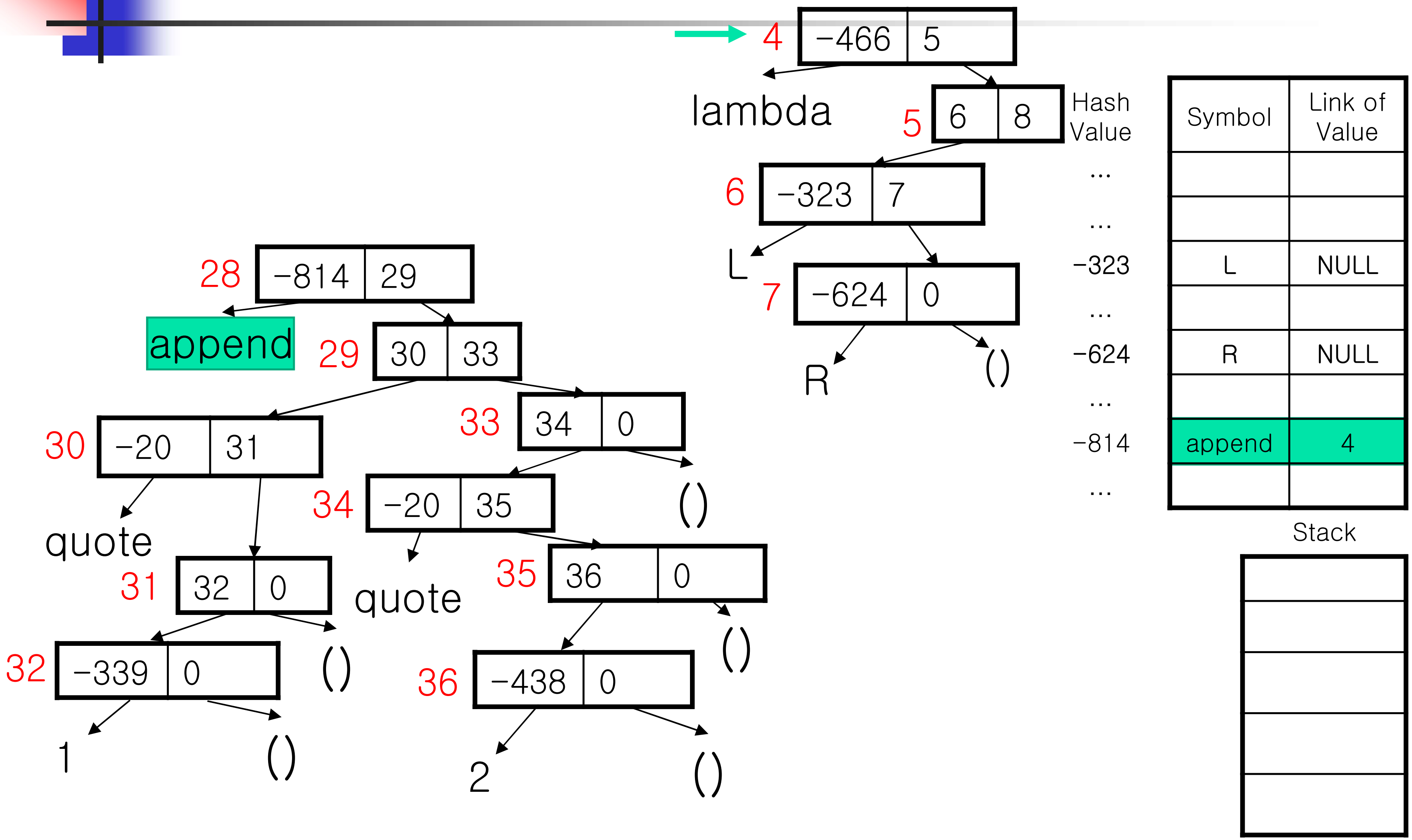
Evaluation of (append '(1) '(2))



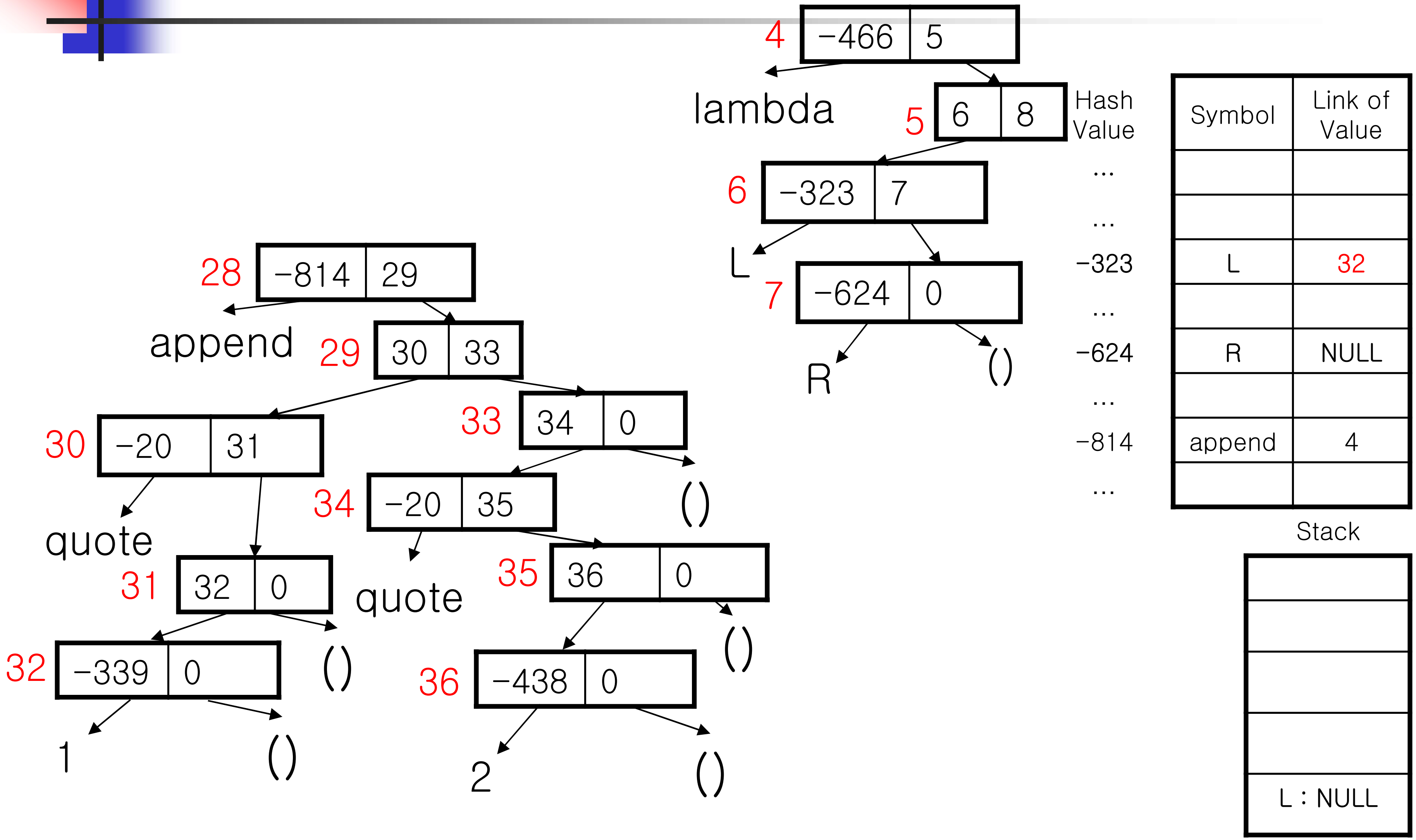
Hash Value	Symbol	Link of Value
…		
…		
-323	L	NULL
…		
-624	R	NULL
…		
-814	append	4
…		

Stack

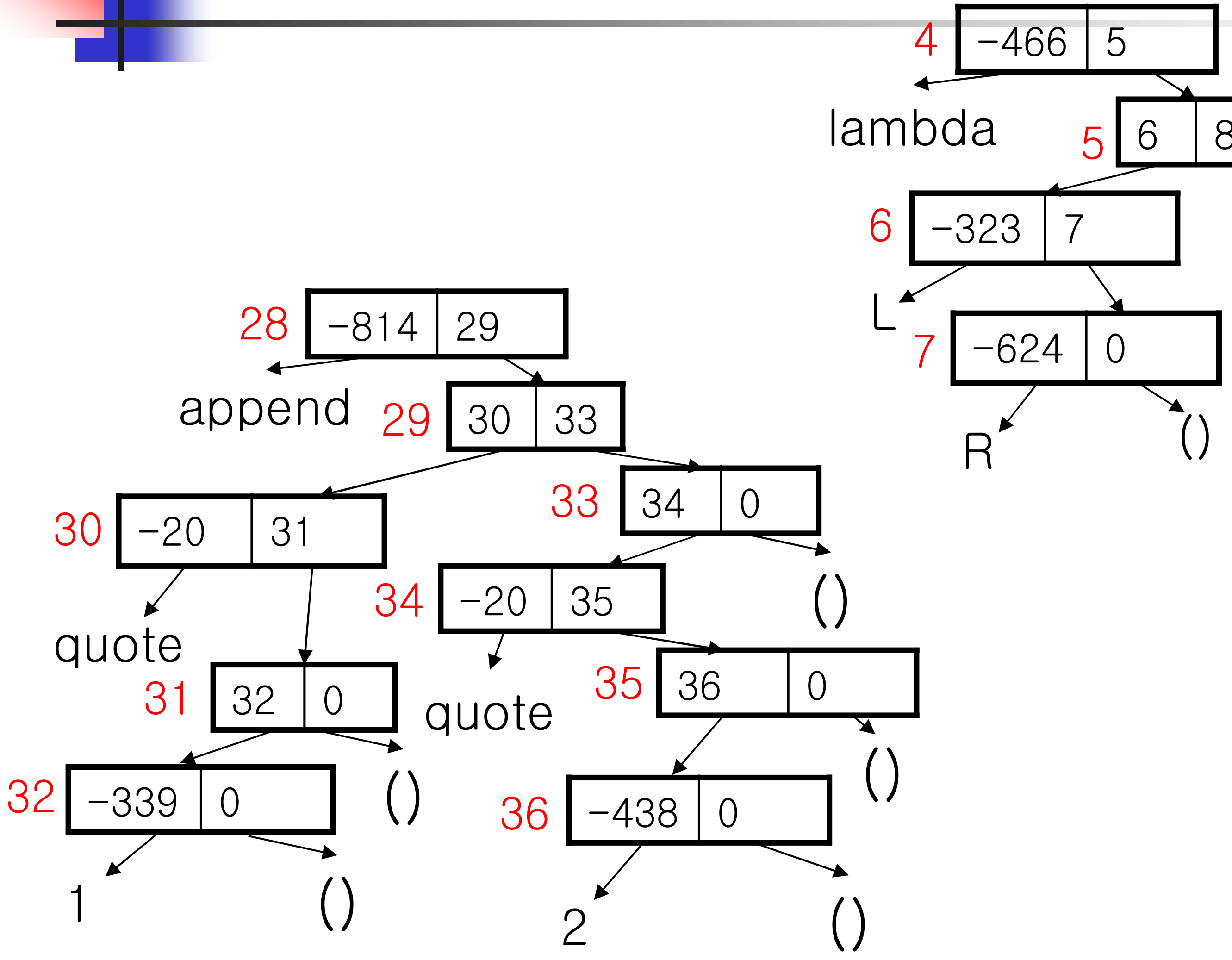
Evaluation of (append '(1) '(2))



Evaluation of (append '(1) '(2))



Evaluation of (append '(1) '(2))



Hash Value	Symbol	Link of Value
…		
…		
-323	L	32
…		
-624	R	36
…		
-814	append	4
…		

Stack

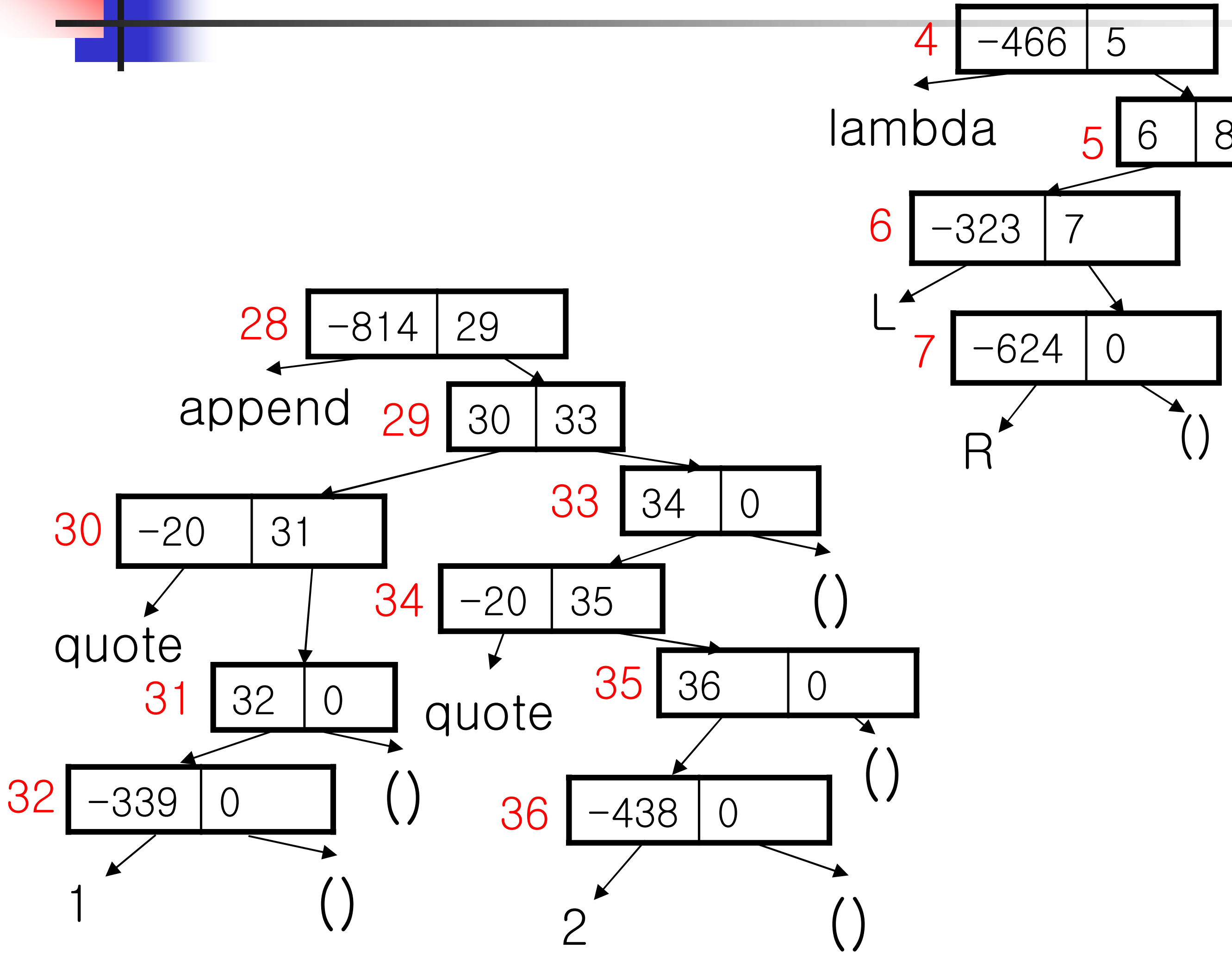
R : NULL
L : NULL

Append a list to another list

```
(define (append L R)
  (cond ((null? L) R)
    (else (cons (car L) (append (cdr L) R)))))
```

- (append '(1) '(2)) : (1 2)

Evaluation of (append '(1) '(2))

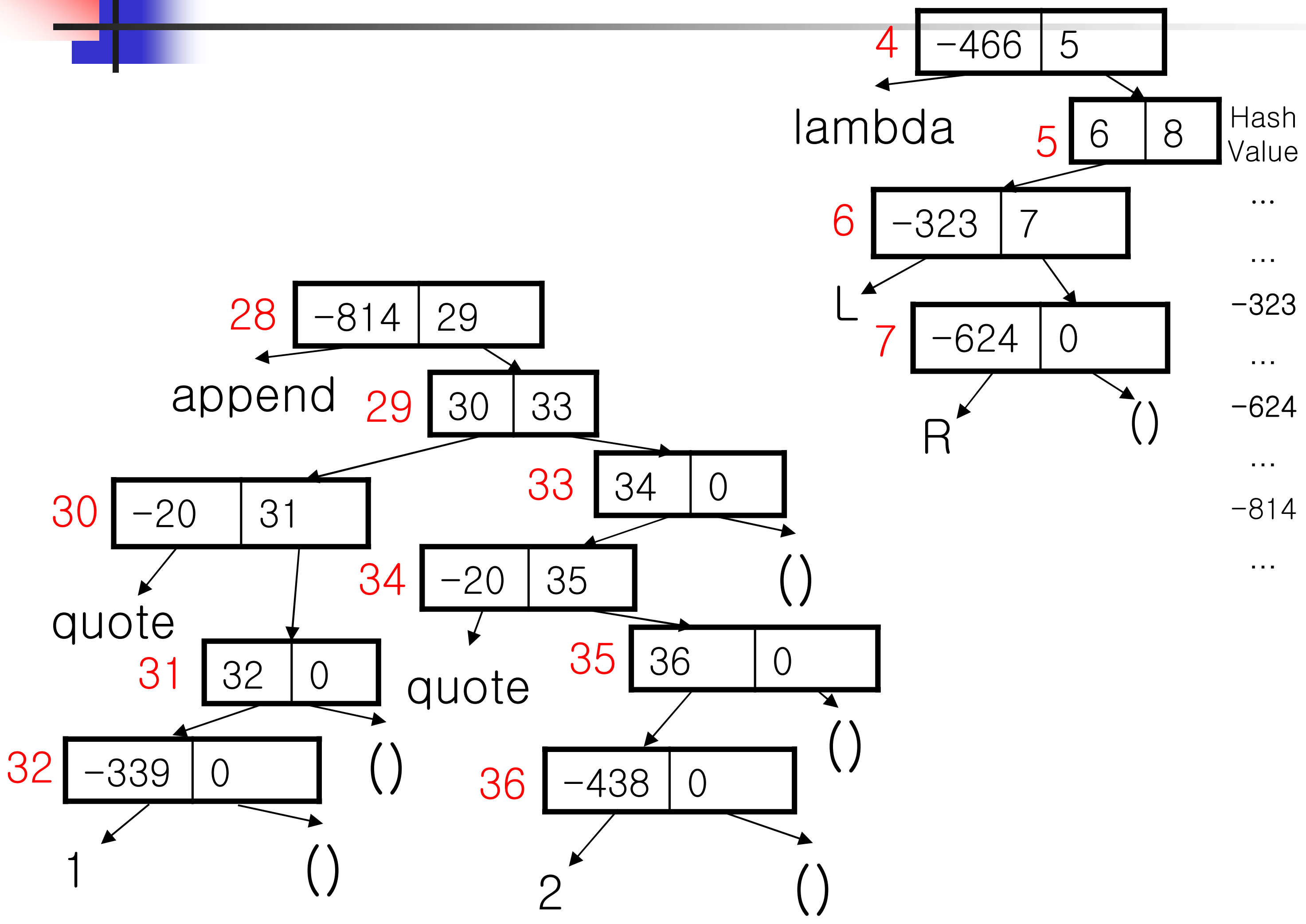


Hash Value	Symbol	Link of Value
...		
...		
-323	L	0
...		
-624	R	36
...		
-814	append	4
...		

Stack

L : 32
R : NULL
L : NULL

Evaluation of (append '(1) '(2))

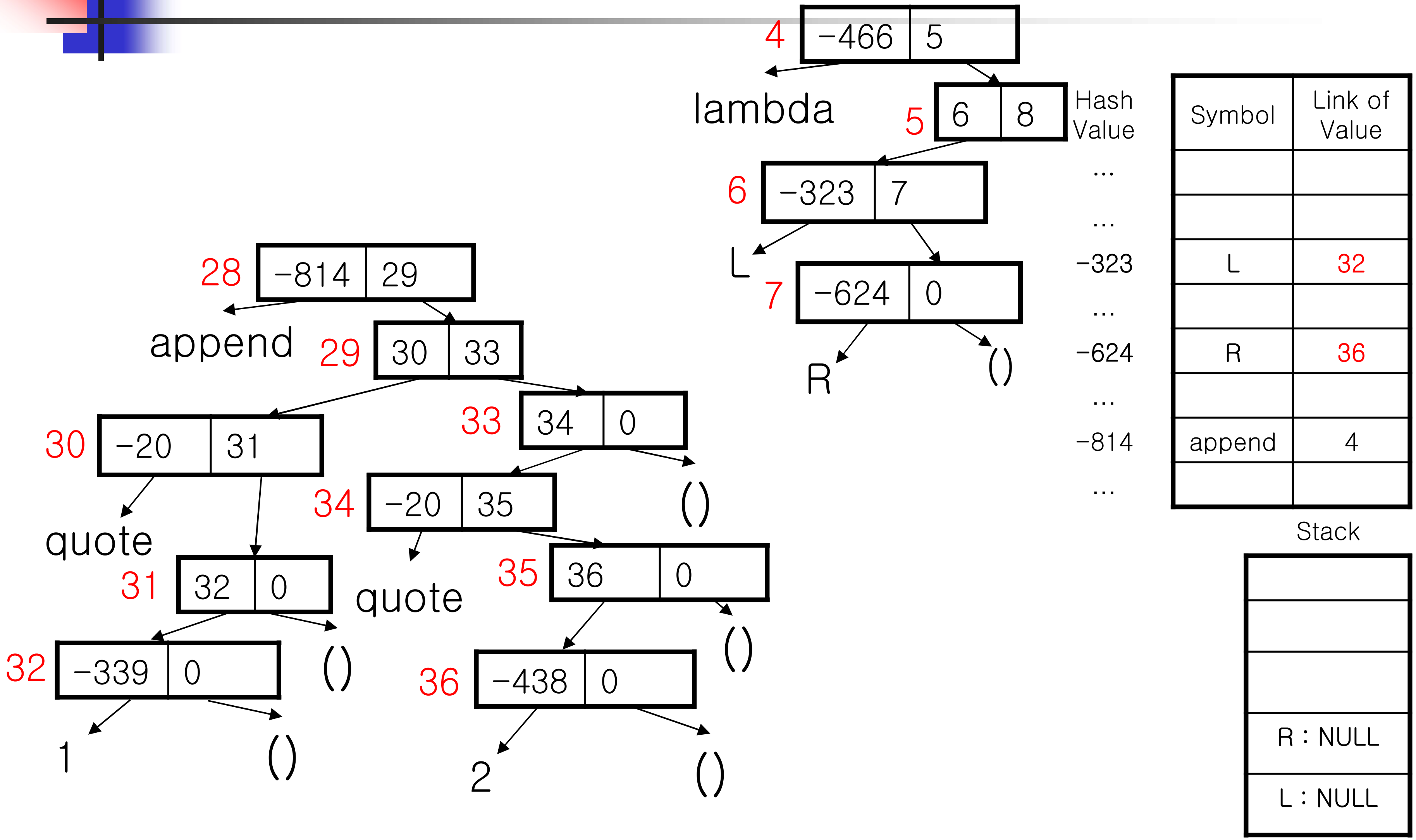


Symbol	Link of Value
L	0
R	36
append	4

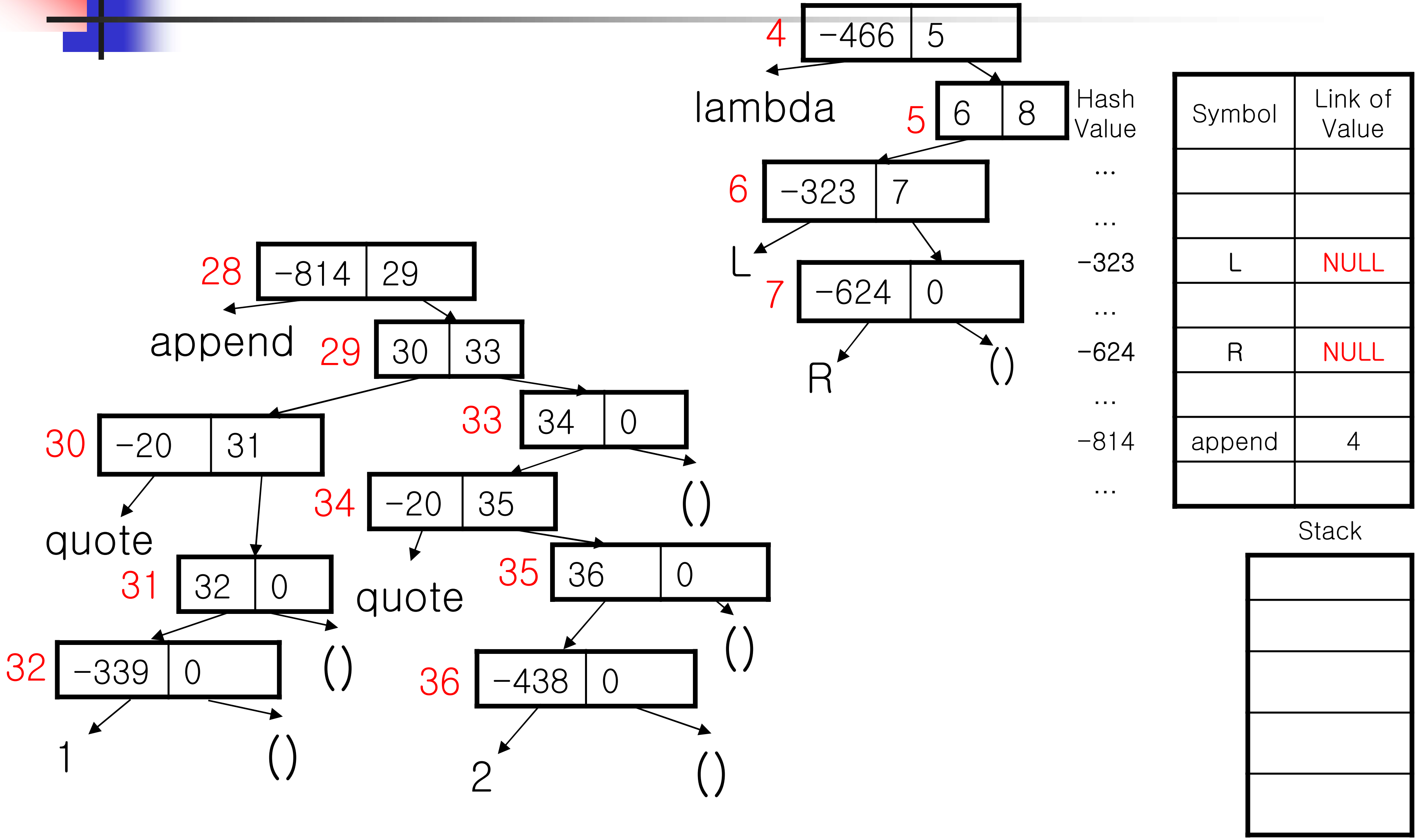
Stack

R : 36
L : 32
R : NULL
L : NULL

Evaluation of (append '(1) '(2))



Evaluation of (append '(1) '(2))



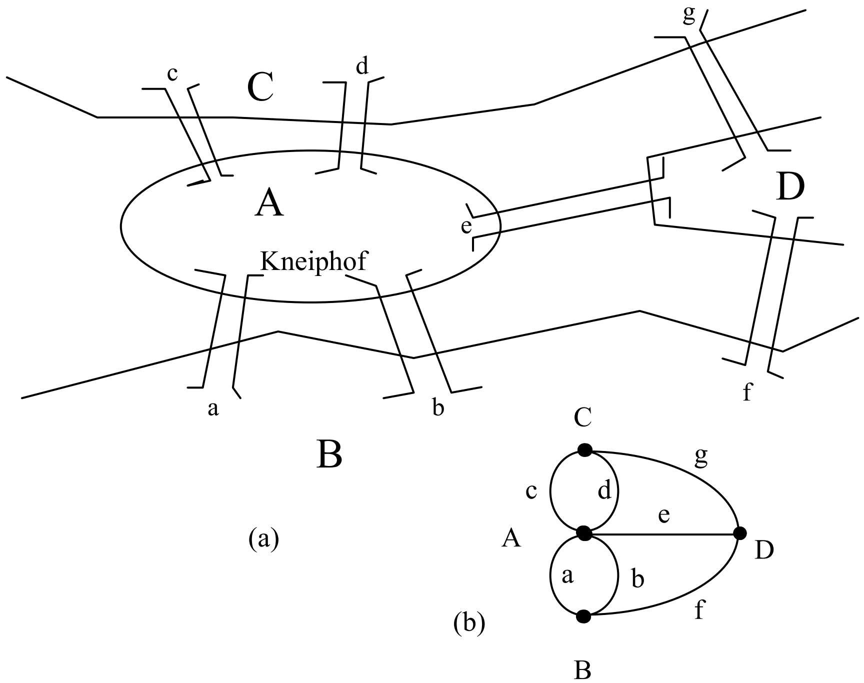
Graphs

Introduction to Data Structures

Kyuseok Shim
ECE, SNU.

Graph Abstract Data Type

- Konigsberg bridge problem



–Eulerian walk
Degree of each vertex is even

Figure 6.1 : (a) Section of the river Pregel in Konigsberg; (b) Euler's graph

Graph Abstract Data Type (Cont.)

- Graph G=(V, E)
  - V is a finite, nonempty set of vertices
  - E is a set of edges
  - An edge is a pair of vertices
  - V(G) is the set of vertices of G
  - E(G) is the set of edges of G

- Undirected (directed) graph
  - The pair of vertices representing an edge is unordered (ordered)

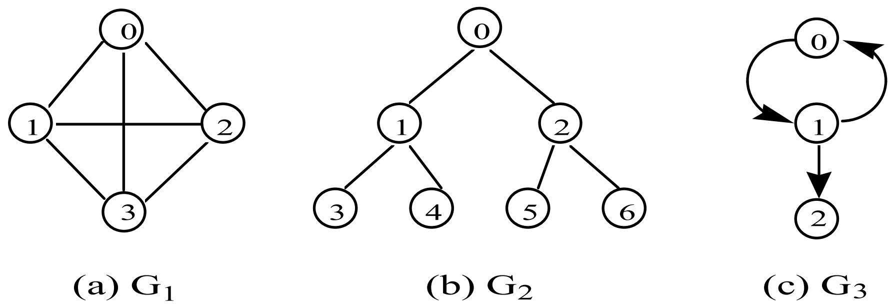
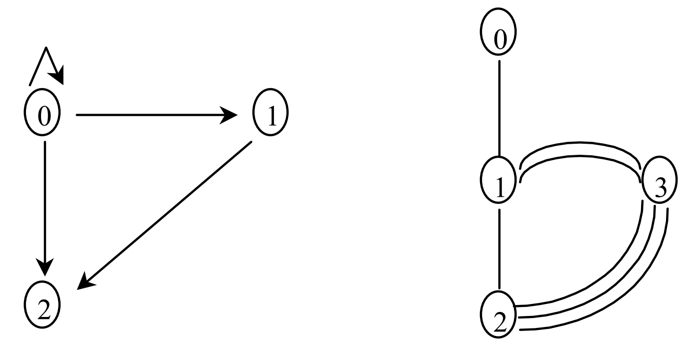


Figure 6.2 : Three sample graphs

Graph Abstract Data Type (Cont.)

- Restriction
  - A graph may not have an edge from a vertex back to itself
  - A graph may not have multiple occurrences of the same edge



(a) Graph with a self edge (b) Multigraph

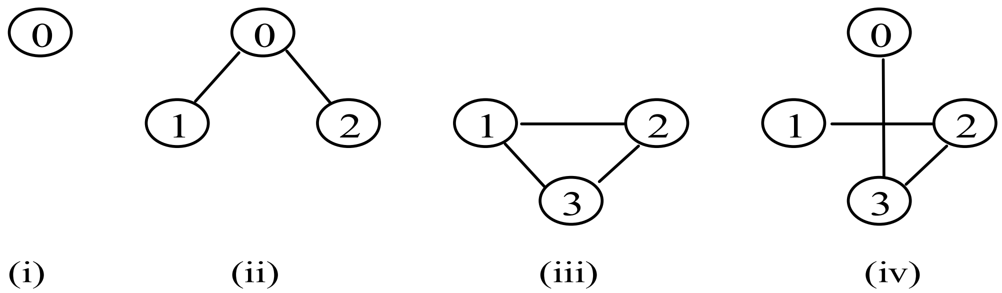
Figure 6.3 : Examles of graphlike structures

Graph Abstract Data Type (Cont.)

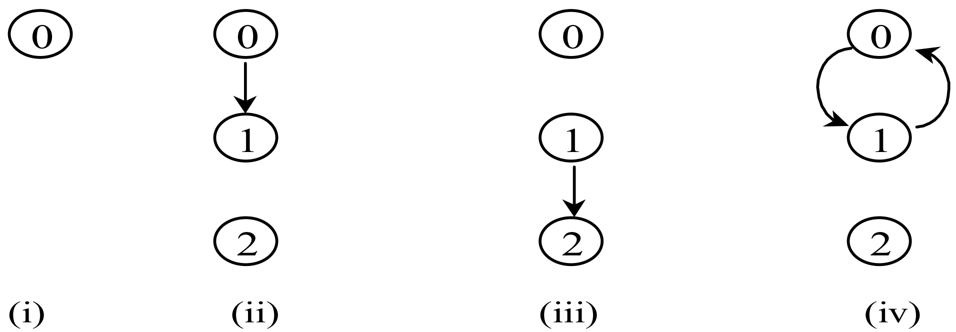
- Complete graph
  - n-vertex, undirected graph with n(n-1)/2 edges

- (u,v) is an edge in E(G)
  - Vertices u and v are adjacent
  - (u,v) is incident on vertices u and v
  - if (u, v) is a directed edge
    - u is adjacent to v
    - v is adjacent from u
- Subgraph of G
  - Graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$

Graph Abstract Data Type (Cont.)



(a) some of the subgraphs of G_1



(b) some of the subgraphs of G_3

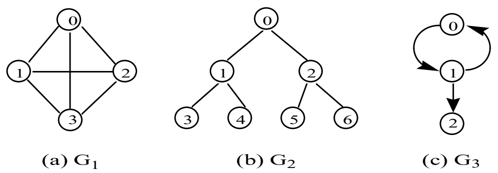


Figure 6.4 : Some subgraphs

Graph Abstract Data Type (Cont.)

- Path from u to v in G
  - A sequence of vertices u, i_1, i_2, ..., i_k, v such that (u,i_1) (i_1, i_2) ...(i_k, v) are edges in E(G)
  - Length of path is number of edges on it
  - Simple path is path in which all vertices except possibly the first and last are distinct
  - Cycle is a simple path in which the first and last vertices are the same

Graph Abstract Data Type (Cont.)

- Vertices u and v are connected in (undirected) graph G, there is a path in G from u to v
- Connected graph
  - For every pair of distinct vertices u and v in V(G) there is a path from u and v
- (connected) Component
  - A maximally connected subgraph
  - Maximal: no more vertices or edges can be added while preserving its connectivity

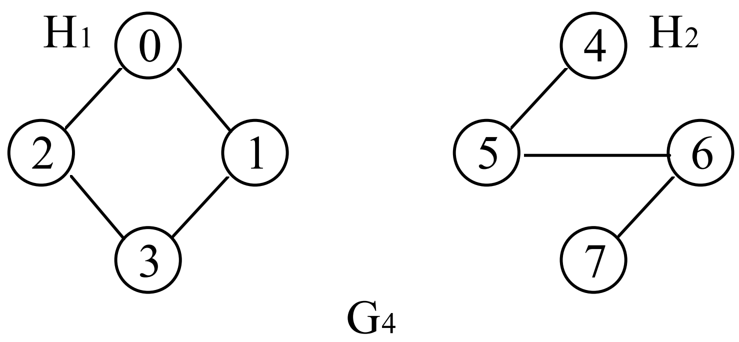


Figure 6.5 : A graph with two connected components

Graph Abstract Data Type (Cont.)

- Tree
  - Connected acyclic graph
- Degree of vertex
  - Number of edges incident to that vertex
- d_i is the degree of vertex i in G with n vertices and e edges

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

Graph Abstract Data Type (Cont.)

```
class Graph
{// objects: A nonempty 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 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 the 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) = 0;
                // delete v and all edges incident to it
        virtual void DeleteEdge(int u, int v) = 0;
                // deelte edge (u,v) from the graph
private:
                int n;          //number of vertices
                int e;          // number of edges
};
```

ADT 6.1 : Abstract data type Graph

Graph Representations

- Adjacency Matrix
- Adjacency Lists

Adjacency Matrix

- Definition
  - G=(V,B) is a graph with n vertices, $n \geq 1$
  - Adjacency matrix A of G
    - two dimensional n x n array
    - A[i][j]=1 iff edge(i, j) is in E(G)

- Properties
  - Space needed is n^2
  - A is symmetric for undirected G
    - Need only the upper or lower triangle of A

Adjacency Matrix (Cont.)

- Degree of vertex i for an undirected graph
  - $$i = \sum_{j=0}^{n-1} A[i][j]$$
- How many edges are in a directed graph?
  - Complexity of operations : $n^2 - n = O(n^2)$ since diagonal entries are zero

	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	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	0
7	0	0	0	0	0	0	1	0

(c) G_4

Figure 6.7 : Adjacency matrices

Adjacency Lists

- Representation
  - One list for each vertex in G
    - Nodes in list i represent vertices that are adjacent from vertex i
    - Each list has a head node
    - Vertices in a list are not ordered
  - Fields of node
    - data : index of vertex adjacent to vertex i
    - link
- Declaration in C++

```
class Graph
{
private:
   List<int> *HeadNodes;
   int n;
public:
   Graph(const int vertices = 0) : n(vertices)
   { HeadNodes = new List<int>[n];};
};
```

Adjacency Lists (Cont.)

- For n vertices and edges
  - Requires n head nodes and 2e list nodes

- Complexity of operations
  - Number of nodes in adjacency list = O(n+e)

Adjacency Lists (Cont.)

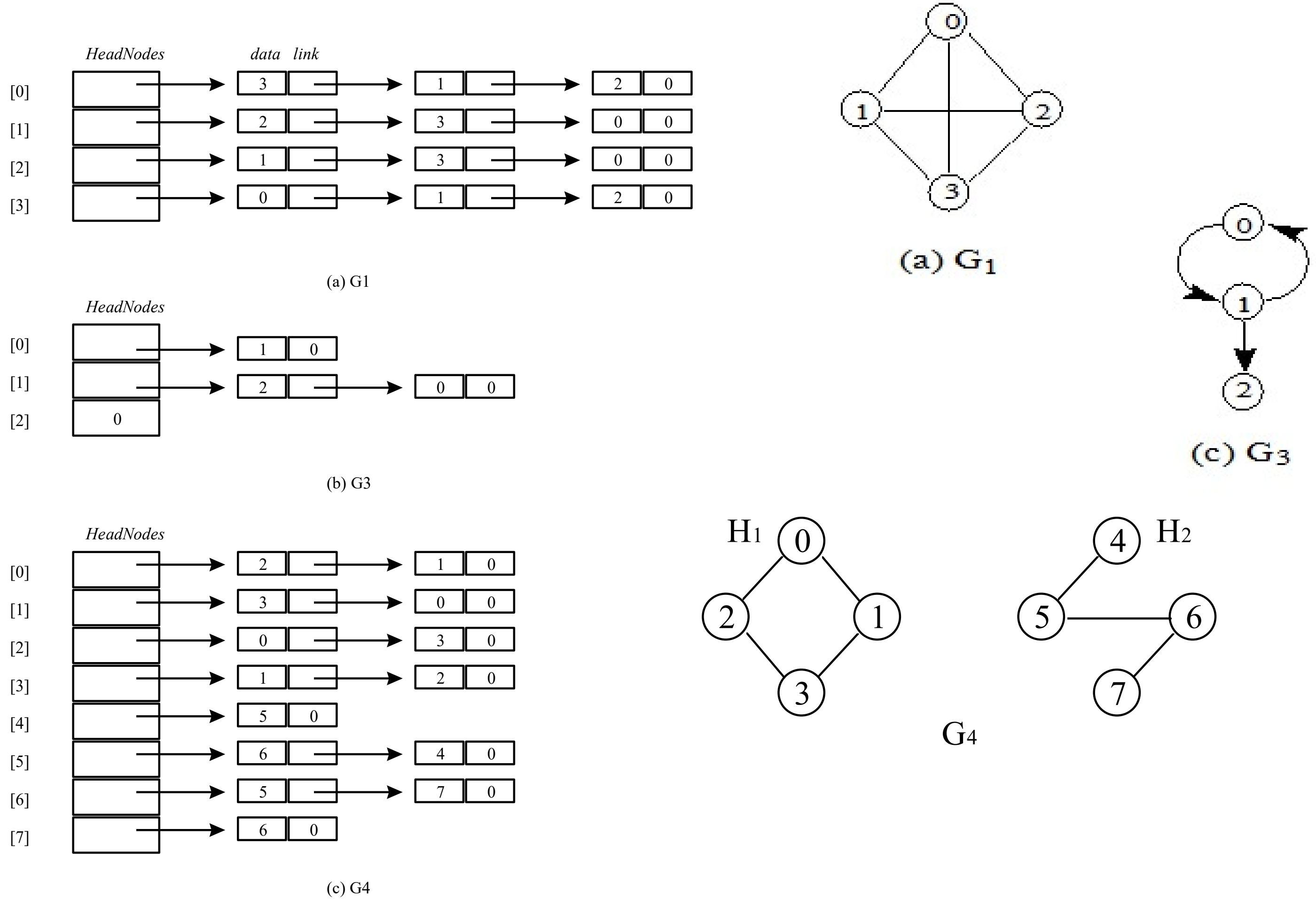


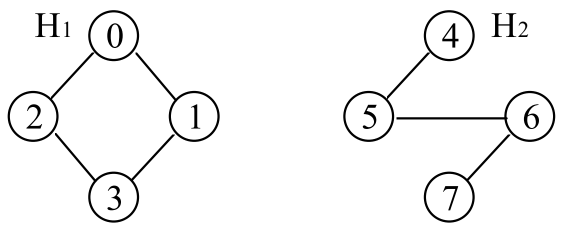
Figure 6.8 : Adjacency lists

Adjacency Lists (Cont.)

- Packing nodes
  - Eliminate pointers
  - node[i] : Starting point of list for vertex i
  - Vertices adjacent from node i :
  - node[node[i]], ..., node[node[i+1]-1]

int $nodes[n + 2 * e + 1];$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	2	1	3	0	0	3	1	2	5	6	4	5	7	6



G4

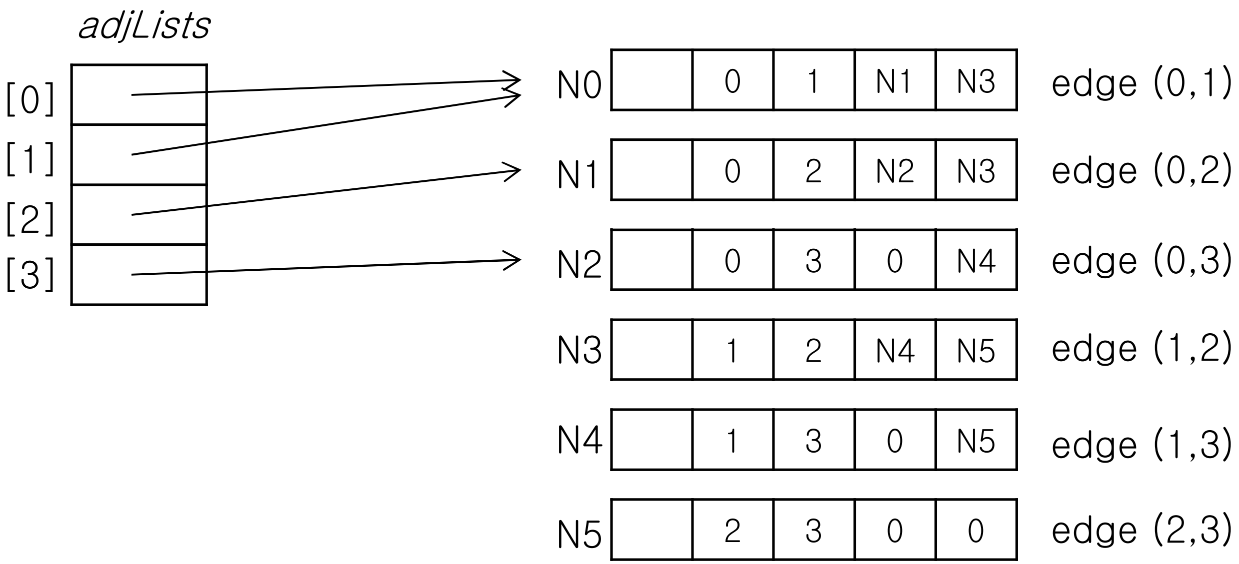
Figure 6.9 : Sequential representation of graph G4

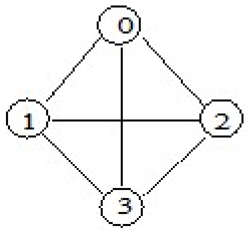
Adjacency Multilists

- Property
  - For each edge, there will be exactly one node
  - But this node will be in two lists
- Boolean mark field *m*
  - Indicate whether or not the edge has been examined
- Storage requirement
  - Same as for normal adjacency lists except for the addition of mark bit
- Node structure

m	*vertex1*	*vertex2*	*link1*	*link2*

Adjacency Multilists (Cont.)





(a) G_1

The lists are

vertex 0: N0 -> N1 -> N2
vertex 1: N0 -> N3 -> N4
vertex 2: N1 -> N3 -> N5
vertex 3: N2 -> N4 -> N5

Figure 6.12:Adjacency multilists for G_1 of Figure 6.2(a)

Weighted Edges

- Network
  - Graph with weighted edges
- Representation
  - Adjacency matrix
    - A[i][j] keeps weight
  - Adjacency list
    - Additional field in list node keeps weight

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

- Procedure
  1. Visit start vertex v
  2. An unvisited vertex w adjacent to v is selected, and initiate DFS from w
  3. When u is reached such that all its adjacent vertices have been visited
     - Back up to the last vertex visited that has an unvisited vertex w adjacent to it
     - Initiate DFS from w
  4. Search terminates when no unvisited vertex can be reached from visited vertices

Depth-First-Search (Cont.)

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

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

Program 6.1 : Depth-first search

Depth-First-Search (Cont.)

- Analysis
  - If adjacency list is used
    - Examines each node in the adjacency lists at most once
    - There are 2e list nodes
    - O(e)
  - If adjacency matrix is used
    - time to determine all adjacent vertices to v : O(n)
    - total time : $O(n^2)$

Breadth-First-Search

- Procedure
  1. Visit start vertex v
  2. Visit all unvisited vertices adjacent to v
  3. Visit unvisited vertices adjacent to the newly Visited vertices

Breadth-First-Search (Cont.)

```
virtual void Graph::BFS() // Driver
{
      visited = new bool[n];
            // visited is declared as a bool* data member of graph
      fill(visited, visited + n, false);
      for(int v=0; v<n; v++)
          if(visited[v] == false)
              BFS(v); // start search at vertex 0
      delete [] visited;
}

virtual void Graph::BFS(int v)
{ // A breadth first search of the graph is carried out beginning at vertex v.
   // visited[i] is set to true when v is visited. The function uses a queue.
      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 loop
}
```

Program 6.2 : Breadth-first search

Breadth-First-Search (Cont.)

- Analysis
  - Adjacency matrix : $O(n^2)$
  - Adjacency list : $O(e)$

Example (DFS and BFS)

- DFS
  - 0 -> 1 -> 3 -> 7 -> 4 -> 5 -> 2 -> 6
- BFS
  - 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7

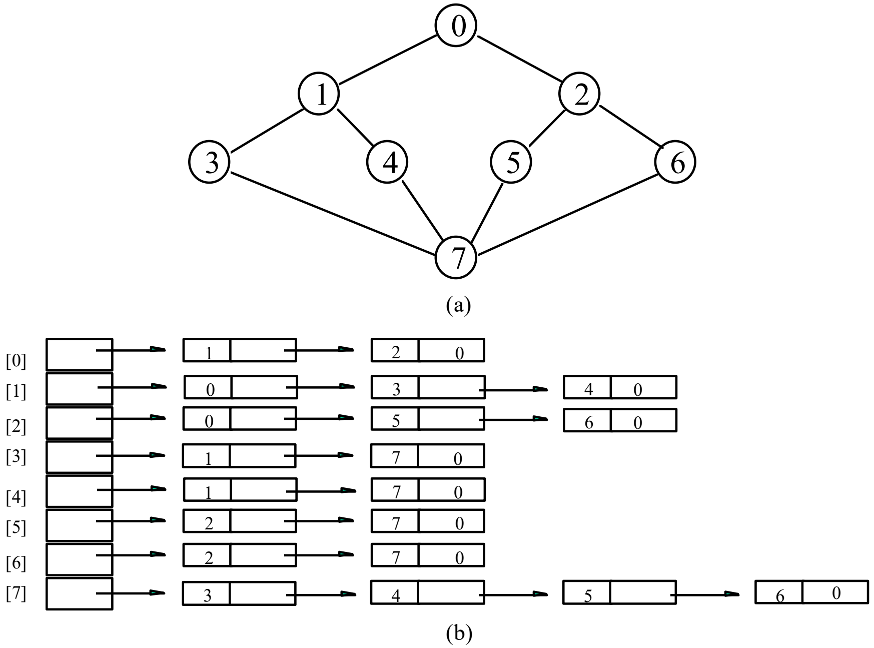


Figure 6.17 : Graph G and its adjacency lists

Connected Components

- If G is an undirected Graph, one can know its connectivity by simply making a call to either DFS or BFS
  - Making a call to either DFS or BFS and then determining if there is any unvisited vertex

- Determining connected components
  - Obtained by making repeated calls to either DFS or BFS
  - Start with a vertex that has not yet been visited

Connected Components (Cont.)

```
virtual void Graph::Components()
{ // Determine the connected components of the graph.
        // visited is assumed to be declared as a bool* data member of Graph
        visited = new bool[n];
        fill(visited, visited + n, false);
        for(i =0; i < n; i++)
                if(!visited[i]) {
                        DFS(i); // find a componenet
                        OutputNewComponents();
                }
        delete [] visited;
}
```

Program 6.3 : Determining connected components

Connected Components (Cont.)

- Analysis
  - Adjacency matrix : $O(n^2)$
  - Adjacency list : $O(n+e)$

Spanning Trees

- Spanning tree
  - Tree consisting of edges in G and including all vertices
  - Depth-First-Spanning tree
  - Breadth-First-Spanning tree

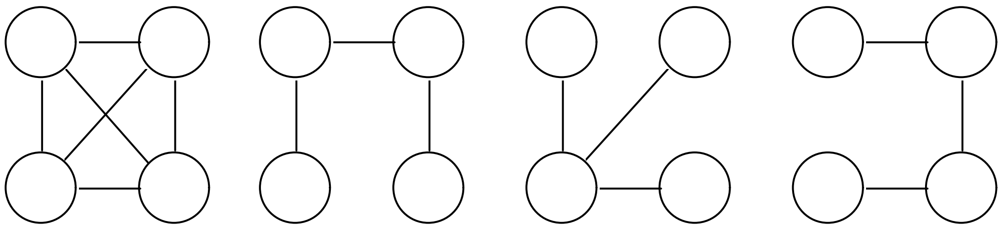
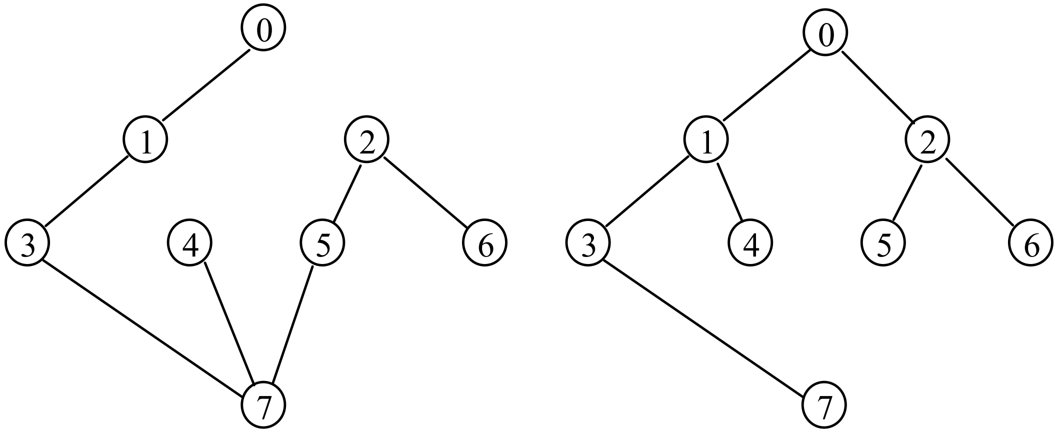


Figure 6.18 : A complete graph and three of its spanning trees

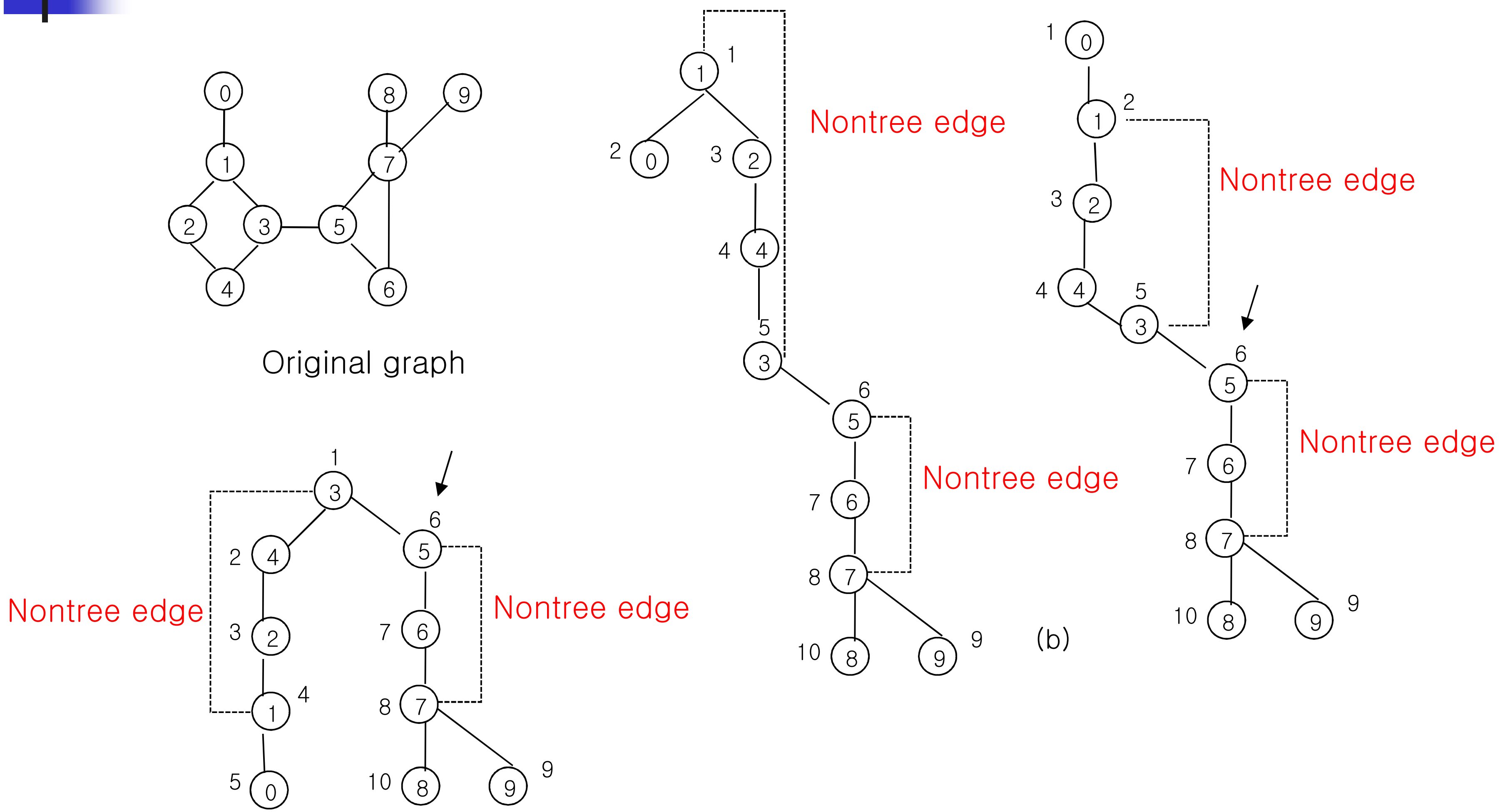
Spanning Trees (Cont.)



(a) *DFS*(0) spanning tree (b) *BFS*(0) spanning tree

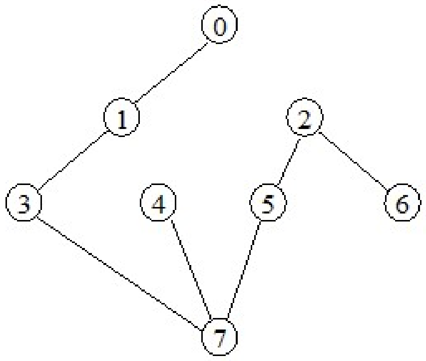
Figure 6.19 : Depth-first and breadth-first spanning trees for graph of Figure 6.17

Depth-first spanning trees



Spanning Trees (Cont.)

- Properties
  - If a non tree edge is introduced into any spanning tree, then a cycle is formed
    - Ex) If (7,6) edge is added to Fig 6.19(a), then the resulting cycle is 7,6,2,5,7
    - Used to obtain an independent set of circuit equations for an electrical network
  - A Spanning tree is a minimal subgraph G' of G such that
    - V(G') = V(G)
    - G' is connected
  - Spanning tree has n-1 edges



(a) *DFS*(0) spanning tree

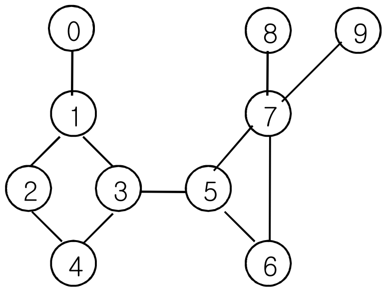
Biconnected Components

- Articulation point
  - A vertex v whose deletion operation leaves behind a graph that has at least two connected components
- Biconnected graph
  - A connected graph that has no articulation points
- Biconnected component
  - Maximal biconnected subgraph
  - The original graph contains no other biconnected subgraph

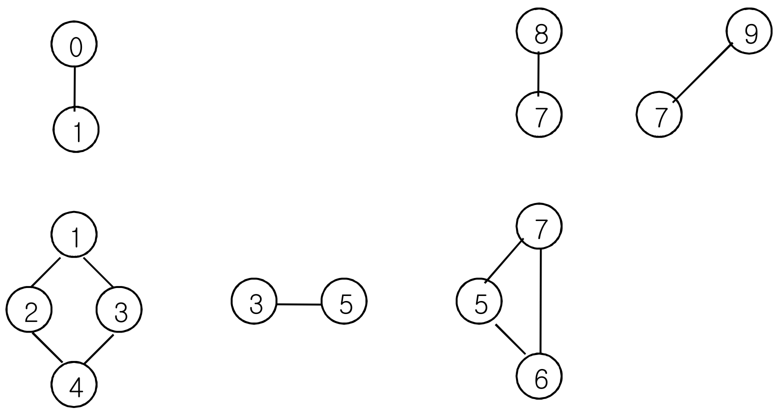
Biconnected Components (Cont.)

- A biconnected graph has just one biconnected component.
- Two biconnected components of the same graph can have at most one vertex in common.
- No edge can be in two or more biconnected components.
- Biconnected components of a graph G partition the edges of G.
- The biconnected components of a connected undirected graph G can be found by using any depth-first spanning tree of G.

Biconnected Components (Cont.)



(a) A connected graph



(b) Its biconnected components

Figure 6.20: A connected graph and its biconnected components

A Depth-first Spanning Tree with Root 0

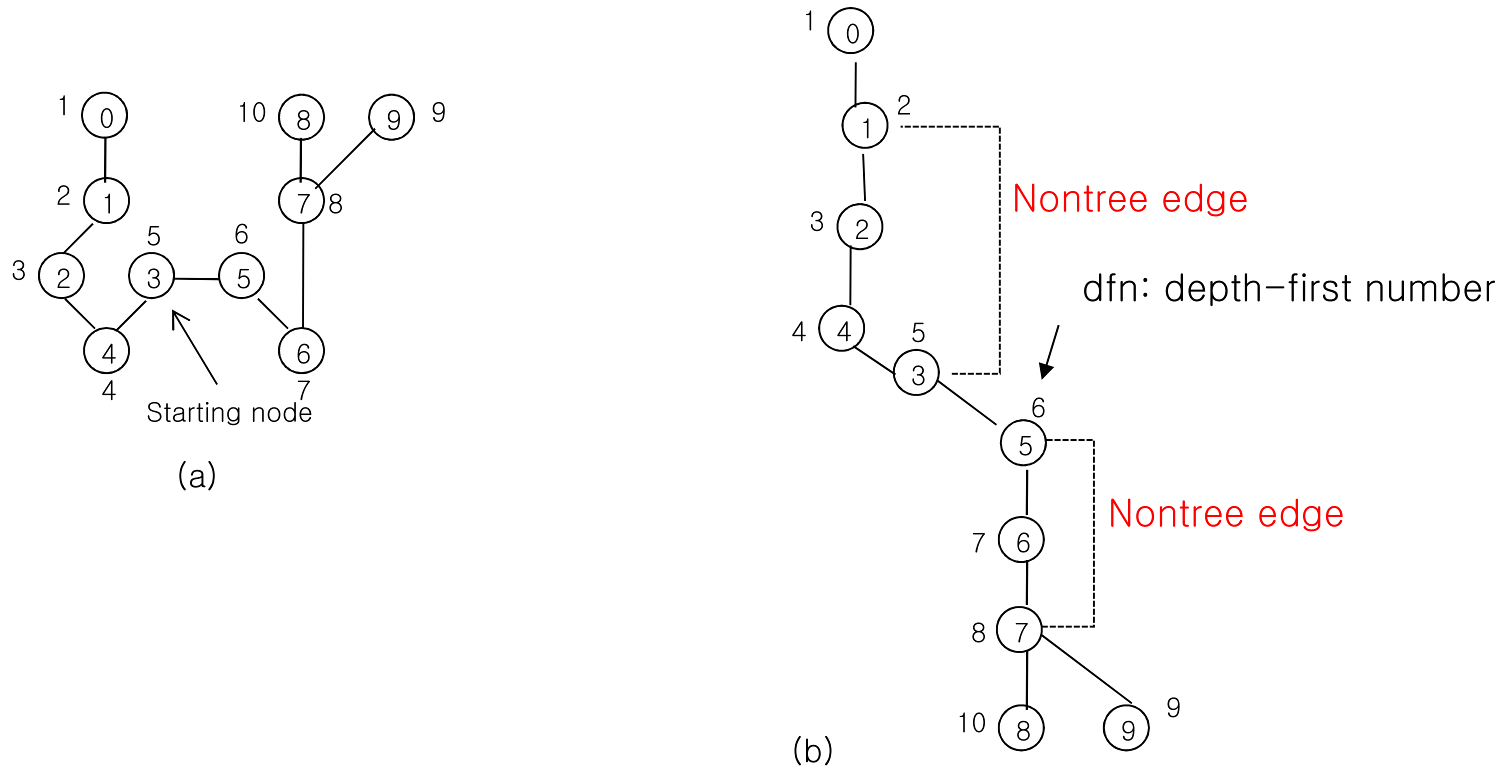
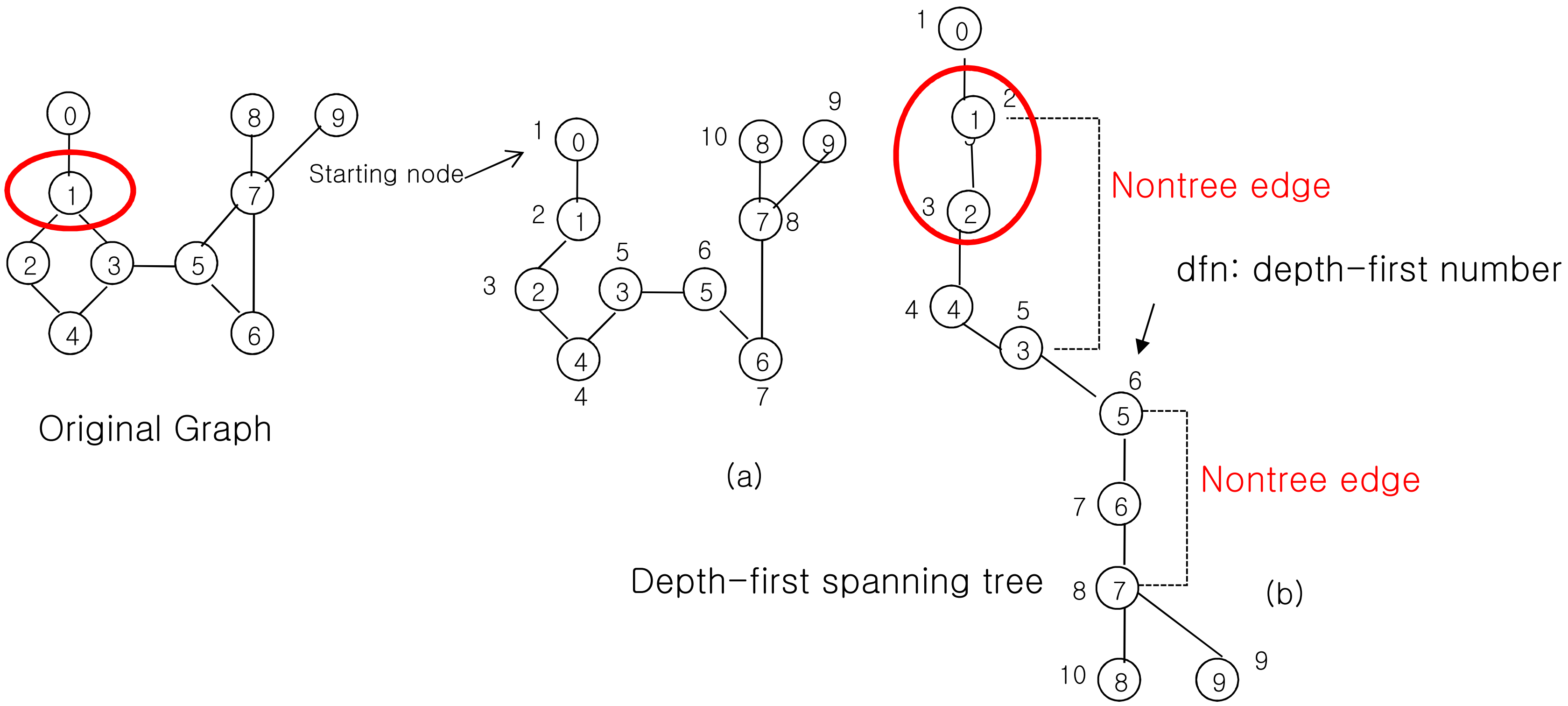


Figure 6.21: Depth-first spanning tree of Figure 6.20(a)

Determining Biconnected Components (Cont.)

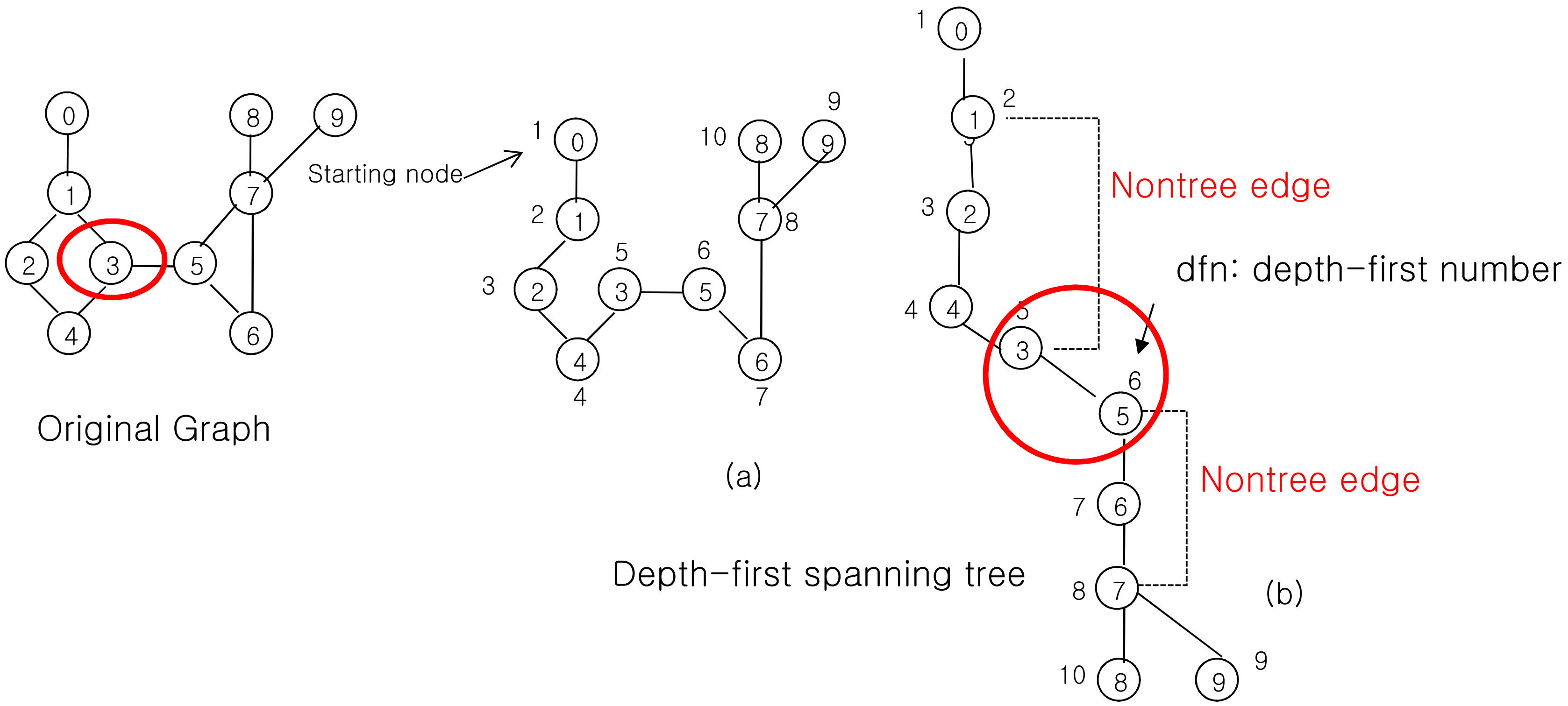


$low(w) \geq dfn(u)$

Vertex	0	1	2	3	4	5	6	7	8	9
dfn	1	2	3	5	4	6	7	8	10	9
low	1	2	2	2	2	6	6	6	10	9

dfn and low values for the spanning tree

Determining Biconnected Components (Cont.)

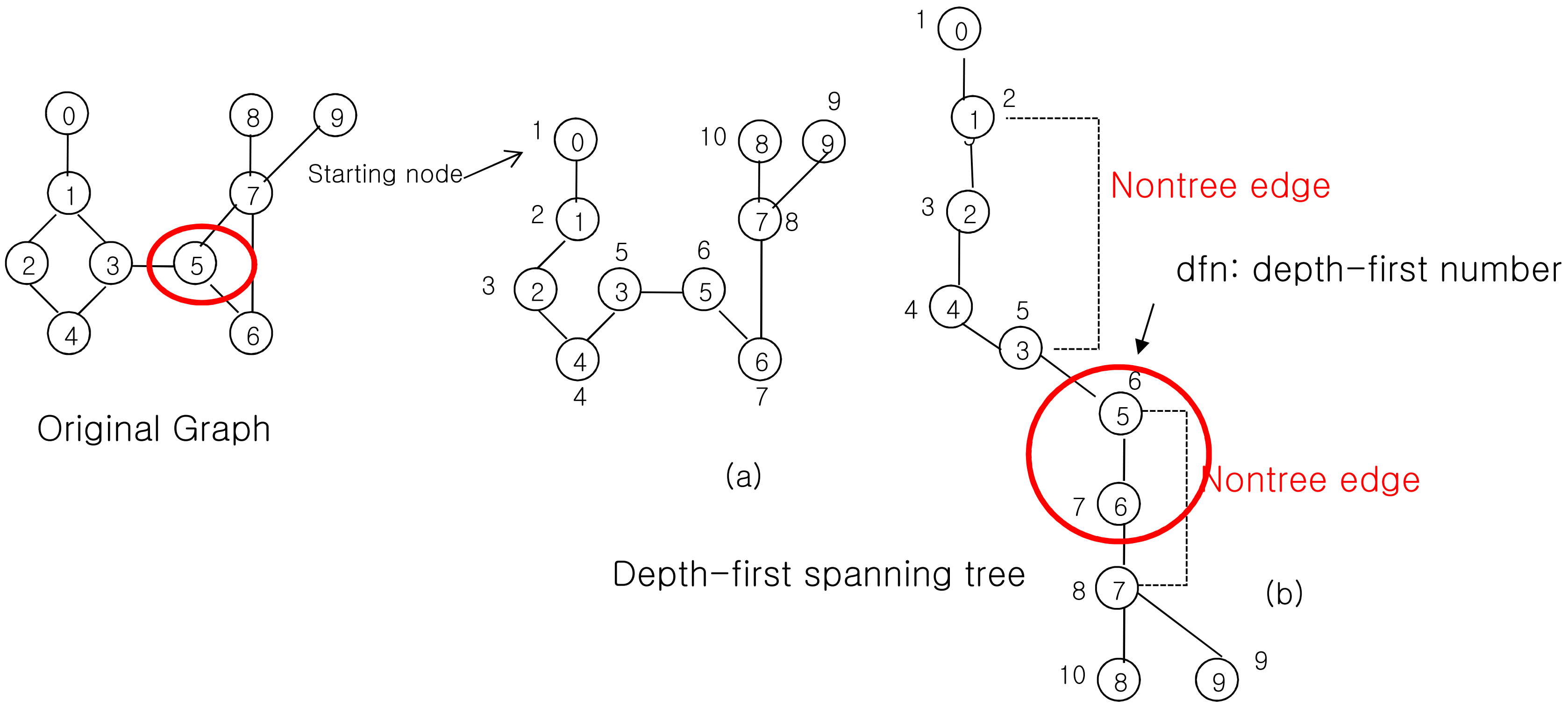


$low(w) \geq dfn(u)$

Vertex	0	1	2	3	4	5	6	7	8	9
dfn	1	2	3	5	4	6	7	8	10	9
low	1	2	2	2	2	6	6	6	10	9

dfn and low values for the spanning tree

Determining Biconnected Components (Cont.)

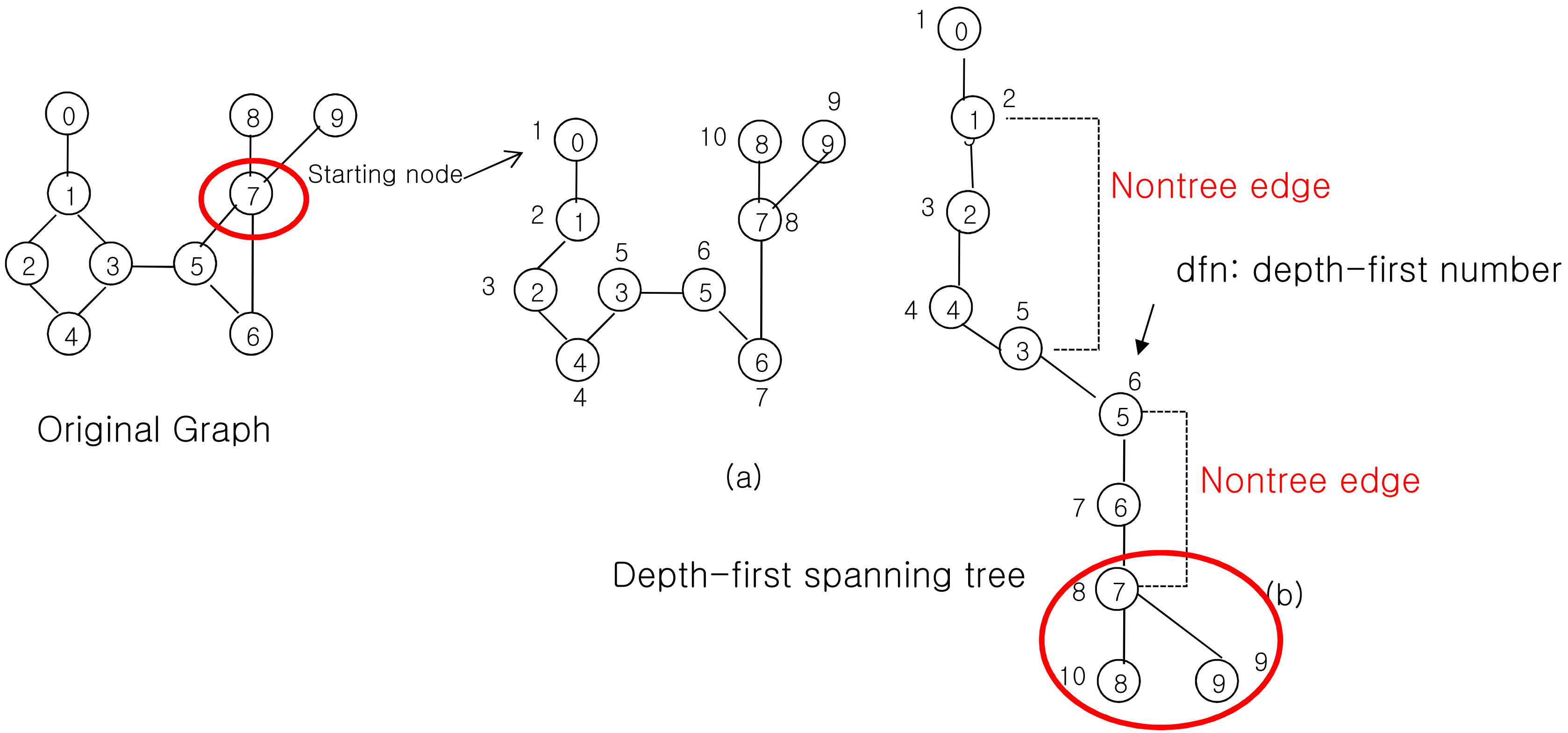


low(w) ≥ dfn(u)

Vertex	0	1	2	3	4	5	6	7	8	9
dfn	1	2	3	5	4	6	7	8	10	9
low	1	2	2	2	2	6	6	6	10	9

dfn and low values for the spanning tree

Determining Biconnected Components (Cont.)

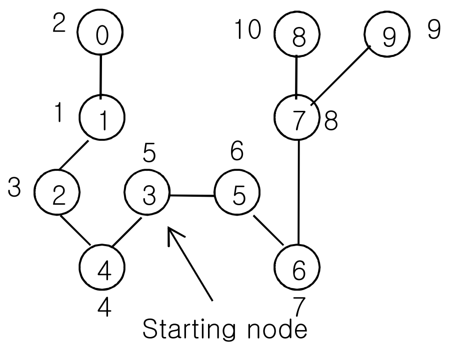


$low(w) \geq dfn(u)$

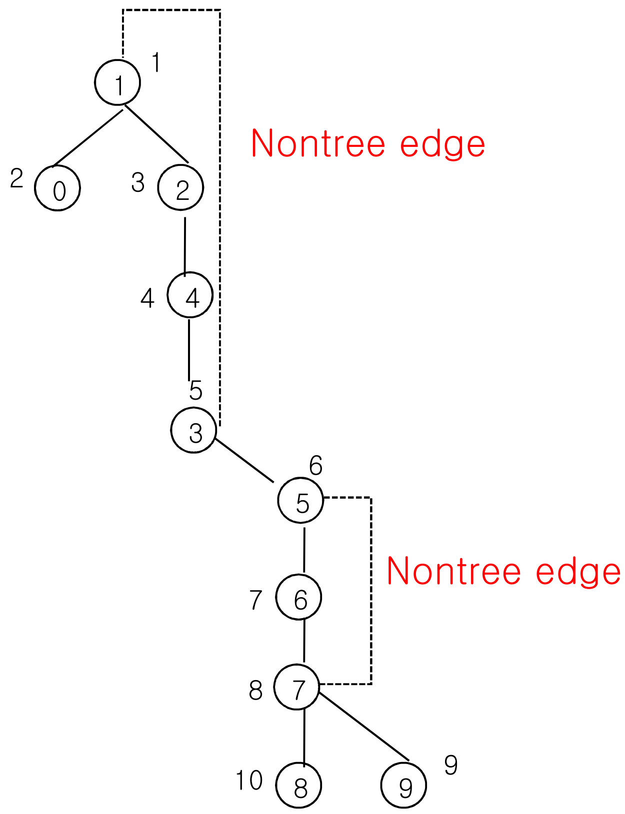
Vertex	0	1	2	3	4	5	6	7	8	9
dfn	1	2	3	5	4	6	7	8	10	9
low	1	2	2	2	2	6	6	6	10	9

dfn and low values for the spanning tree

A Depth-first Spanning Tree with Root 1



(a)



(b)

A Depth-first Spanning Tree with Root 3

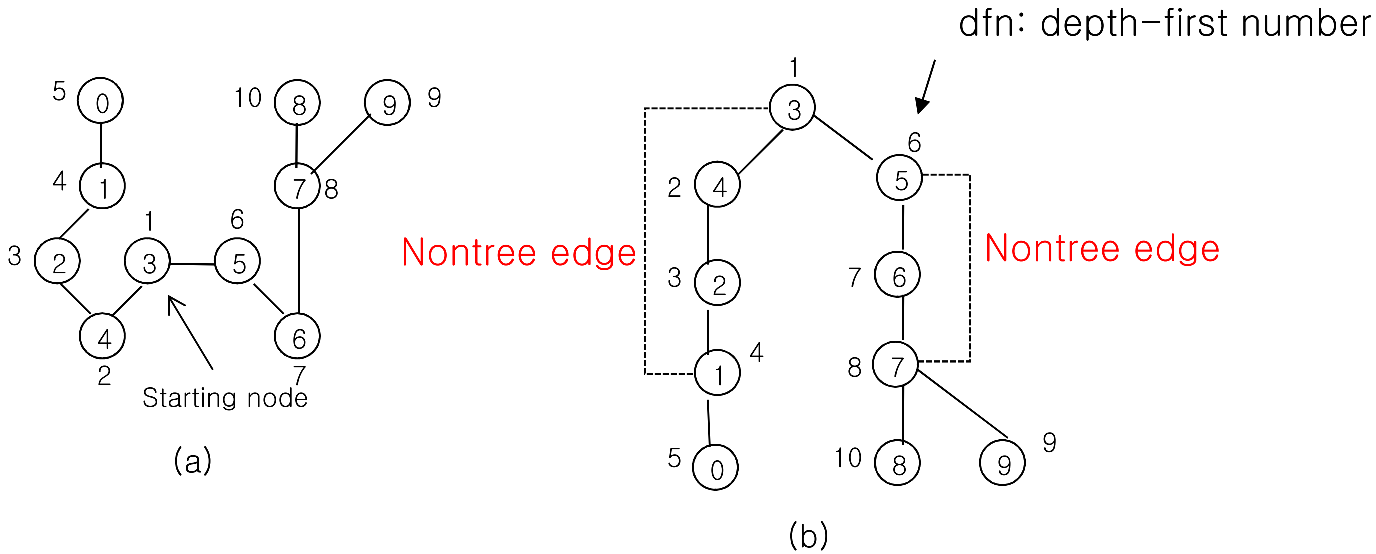


Figure 6.21: Depth-first spanning tree of Figure 6.20(a)

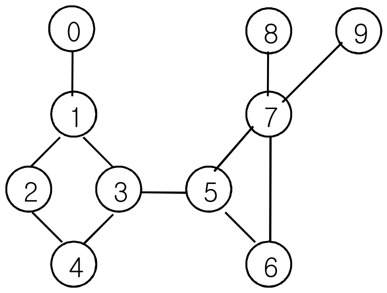
A Depth-first Spanning Tree

- A non-tree 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.
- A nontree edge that is not a back edge is called a cross edge.
- The root node of the depth-first spanning tree is an articulation point iff it has at least two children.
- Any other 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 of w, descendants of w and a single back edge.

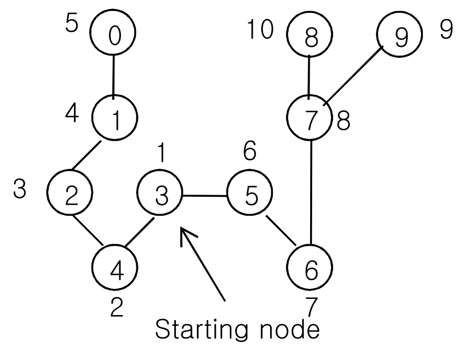
Determining Biconnected Components

- Depth-first Number
  - The sequence in which the vertices are visited during the DFS
- Back edge (u, v)
  - Nontree edge
  - Either u is an ancestor of v or v is an ancestor of u
- low(w) – the lowest depth-first number that can be reached from w using a path of descendants followed by at most one back edge
  - **min{** dfn(w), min{low(x)| x is a child of w},
    min{dfn(x)|(w, x) is a back edge} **}**
- Articulation point (2 cases)
  - vertex u is an articulation point iff
  1. If u is the root of the spanning tree and has two or more children
  2. If u has a child w such that low(w) ≥ dfn(u)

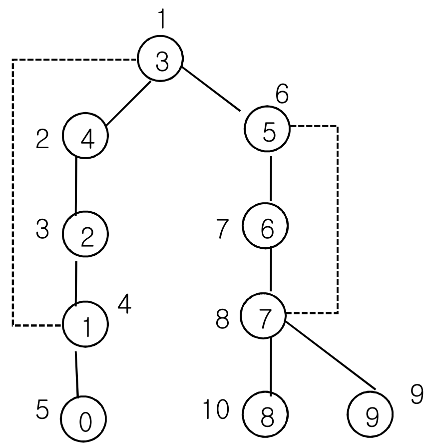
Determining Biconnected Components (Cont.)



Original Graph



(a)



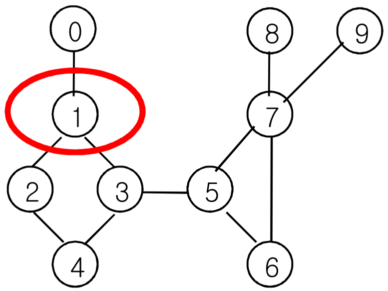
(b)

Depth-first spanning tree

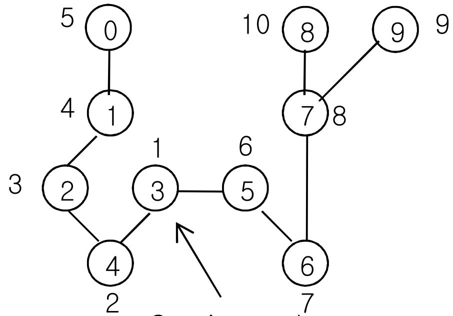
Vertex	0	1	2	3	4	5	6	7	8	9
dfn	5	4	3	1	2	6	7	8	10	9
low	5	1	1	1	1	6	6	6	10	9

Figure 6.22: dfn and low values for the spanning tree of Figure 6.21(b)

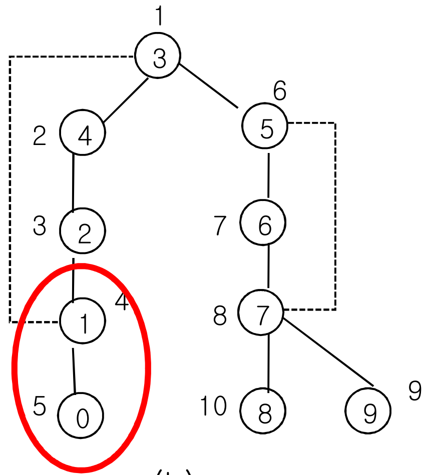
Determining Biconnected Components (Cont.)



Original Graph



(a)



(b)

Depth-first spanning tree

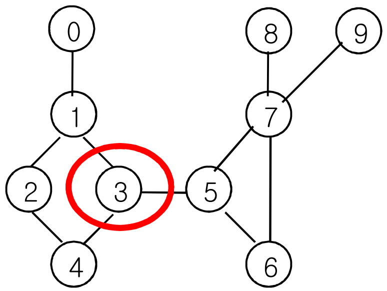
$low(w) \geq dfn(u)$

Vertex	0	1	2	3	4	5	6	7	8	9
dfn	5	4	3	1	2	6	7	8	10	9
low	5	1	1	1	1	6	6	6	10	9

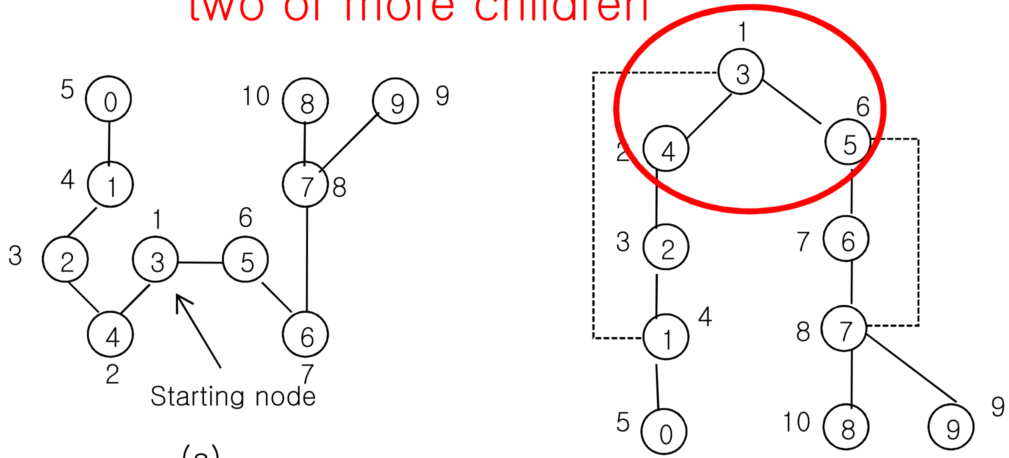
Figure 6.22: dfn and low values for the spanning tree of Figure 6.21(b)

Determining Biconnected Components (Cont.)

root of the spanning tree and has two or more children



Original Graph



(a)

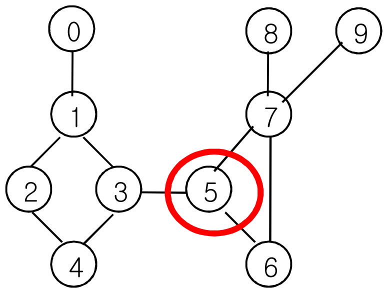
(b)

Depth-first spanning tree

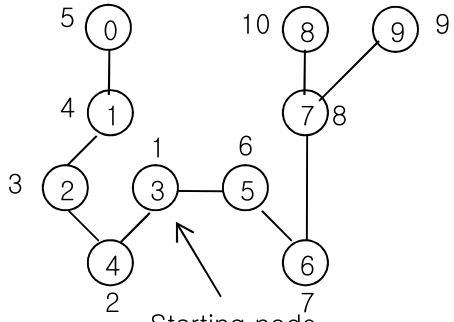
Vertex	0	1	2	3	4	5	6	7	8	9
dfn	5	4	3	1	2	6	7	8	10	9
low	5	1	1	1	1	6	6	6	10	9

Figure 6.22: dfn and low values for the spanning tree of Figure 6.21(b)

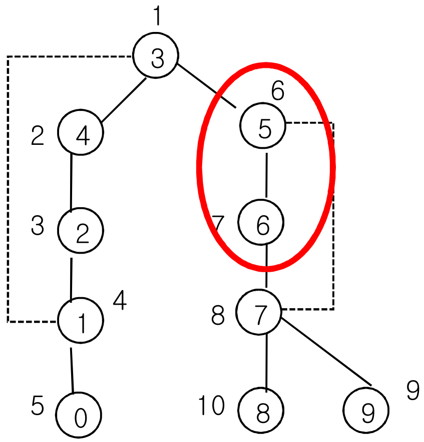
Determining Biconnected Components (Cont.)



Original Graph



(a)



(b)

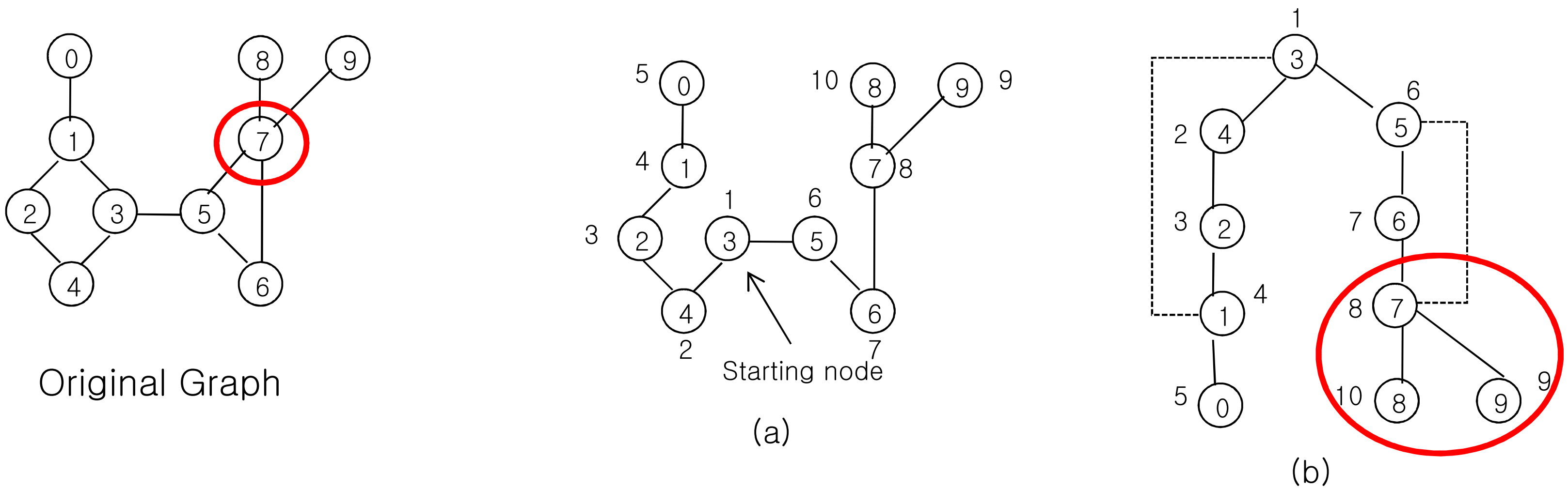
Depth-first spanning tree

low(w) ≥ dfn(u)

Vertex	0	1	2	3	4	5	6	7	8	9
dfn	5	4	3	1	2	6	7	8	10	9
low	5	1	1	1	1	6	6	6	10	9

Figure 6.22: dfn and low values for the spanning tree of Figure 6.21(b)

Determining Biconnected Components (Cont.)



Depth-first spanning tree

low(w) ≥ dfn(u)

Vertex	0	1	2	3	4	5	6	7	8	9
dfn	5	4	3	1	2	6	7	8	10	9
low	5	1	1	1	1	6	6	6	10	9

Figure 6.22: dfn and low values for the spanning tree of Figure 6.21(b)

Determining Biconnected Components (Cont.)

```
virtual void Graph::DfnLow(const int x) // begin DFS at vertex x
{
        num = 1;              // num is an int data member of Graph
        dfs = new int[n];     // dfn is declared as int* in Graph
        low = new int[n];     // low is declared as int* in Graph
        fill(dfn, dfn + n, 0);
        fill(low, low + n, 0);
        DfnLow(x, -1); // start at vertex x
        delete [] dfn;
        delete [] low;
}

void Graph::DfnLow(const int u, const int v)
{ // Compute dfn and low while performing a depth first search beginning at vertex u.
  // v is the parent (if any) of u in the resulting spanning tree.
        dfn[u] = low[u] = num++;
        for(each vertex w adjacent from u) // actual code uses an iterator
                if(dfn[w] == 0) { // w is an unvisited vertex
                        DfnLow(w, u);
                        low[u] = min(low[u], low[w]);
                }
                else if(w != v) low[u] = min(low[u], dfn[w]); // back edge
}
```

Program 6.4: Computing dfn and low

Printing Biconnected Components

- Biconnected components in a graph can be determined by using the previous algorithm with a slight modification.
- That modification is to maintain a stack of edges.
- Keep adding edges to the stack in the order they are visited
- When an articulation point is detected
  - i.e., say a vertex u has a child v such that no vertex in the subtree rooted at v has a back edge ($low[v] \geq dfn[u]$)
  - Pop and print all the edges in the stack till the (u,v) is found, as all those edges including the edge (u,v) will form one biconnected component.

Printing Biconnected Components (Cont.)

```
virtual void Graph::Biconnected()
{
        num = 1;                                   // num is an int datamember of Graph
        dfn = new int[n];     // dfn is declared as int* in Graph
        low = new int[n];     // low is declared as int* in Graph
        fill(dfn, dfn + n, 0);
        fill(low, low + n, 0);
        Biconnected(0,-1); // start at vertex 0
        delete [] dfn;
        delete [] low;
}
virtual void Graph::Biconnected(const int u, const int v)
{ // Compute dfn and low, and output the edges of G by their biconnected components.
  // v is the parent (if any) of u in the resulting spanning tree.
  // s is an initially empty stack declared as a data member of Graph.
        dfn[u] = low[u] = num++;
        for(each vertex w adjacent from u) { // actual code uses an iterator
                if((v != w)&&(dfn[w]<dfn[u])) add (u,w) to stack s;
                if(dfn[w] == 0) { // w is an unvisited vertex
                        Biconnected(w, u);
                        low[u] = min(low[u], low[w]);
                        if(low[w] >= dfn[u]) {
                                cout << "New Biconnected Component: " << end;
                                do {
                                        delete an edge from the 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] = min(low[u], dfn[w]); // back edge
}
```

Program 6.5: Outputting biconected components when n > 1