IE5600 Applied Programming | Week 6

Search and Its Applications

Andrew Lim

Group Assignment for Group Projects

	TAN RAYEN	e0175998@u.nus.edu						
	TAN SHUWEN	e0384826@u.nus.edu		CHANG LIANG	e0014813@u.nus.edu		LEOW WEE SHENT JORD	e0176071@u.nus.edu
A	EOW CHENG ZHENG	e0344189@u.nus.edu		NI XUE	e0006939@u.nus.edu		GU RUIXUE	e0251049@u.nus.edu
	YASMIN BINTE AHMAD	e0384222@u.nus.edu	D	THIRUGNANA SAMBANI	e0010871@u.nus.edu	G	CHONG WOONKIAT	e0452659@u.nus.edu
	LEE MING HOR	e0452894@u.nus.edu		NGOH SISUI	e0452722@u.nus.edu		KHOO EE QING	e0384226@u.nus.edu
	XIE XINGCHEN	e0384332@u.nus.edu		PENG JINGMING	e0383707@u.nus.edu		IRNA BINTI JUMAHAT	e0343996@u.nus.edu
	LIM WEI YANG DYLAN	e0176039@u.nus.edu		LIM ZHEN WEI ZAN	e0176128@u.nus.edu		WU TONG	e0546064@u.nus.edu
	SEAH CHOON KONG	e0030914@u.nus.edu		HU XINPING	e0344056@u.nus.edu		WANG HAI	e0008442@u.nus.edu
B	WU ZHUOCHUN	e0546183@u.nus.edu	E	HUANG XUAN	e0384269@u.nus.edu	H	LEI HAO	e0014898@u.nus.edu
	CHIN KAR KAY	e0385170@u.nus.edu		LV JISHAODONG	e0341657@u.nus.edu		TANG CHOR THENG	e0384417@u.nus.edu
	LOO WENG HENG	e0450255@u.nus.edu		TAMILARASAN SO TEYG	e0319471@u.nus.edu		SOH LI JING	e0452786@u.nus.edu
	LAI JUN GUO SCOTT	e0175273@u.nus.edu		HOONG AN SHENG SAM	e0175262@u.nus.edu		WANG XIN	e0408701@u.nus.edu
	JIAO YANG	e0450317@u.nus.edu		HUO TIANMING	e0007875@u.nus.edu		ZHOU LINLI	e0452923@u.nus.edu
C	WANG YONGJIE	e0516177@u.nus.edu	F	TOH HAN WEI	e0344017@u.nus.edu	I	LOW SIN FOU	e0344083@u.nus.edu
	KWOK JEFFERY	e0384210@u.nus.edu		DONG ZHENG	e0450318@u.nus.edu		SUTAVERAYA SURATAN	e0176486@u.nus.edu
	SHI HANG	e0450191@u.nus.edu		DOMINIQUE YEO ZONG	e0452886@u.nus.edu		KEN LIM JUNE KUANG	e0344074@u.nus.edu

Agenda Today

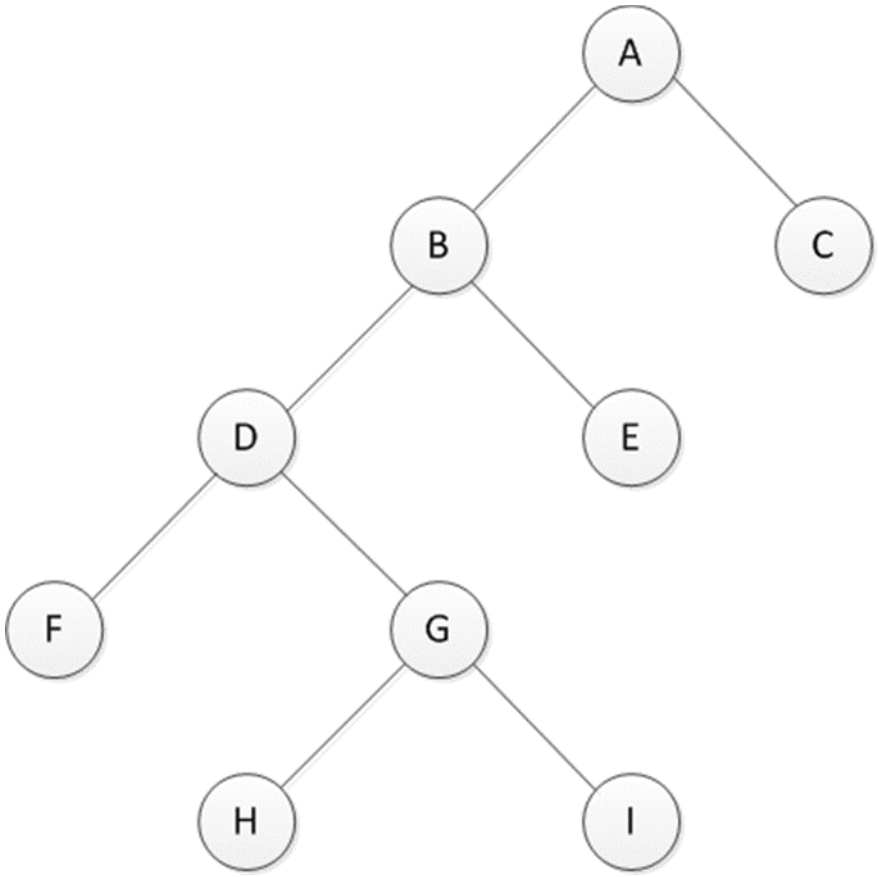
- Will update the course website during the break
- Homework will be released in a few days

- Group Project 1 – Presentation
  - Submit your scores – Based on the interestingness, challenges, innovation, and quality of the presentation

- Time Dependent VRP problems

- Lecture : Search and Its Applications

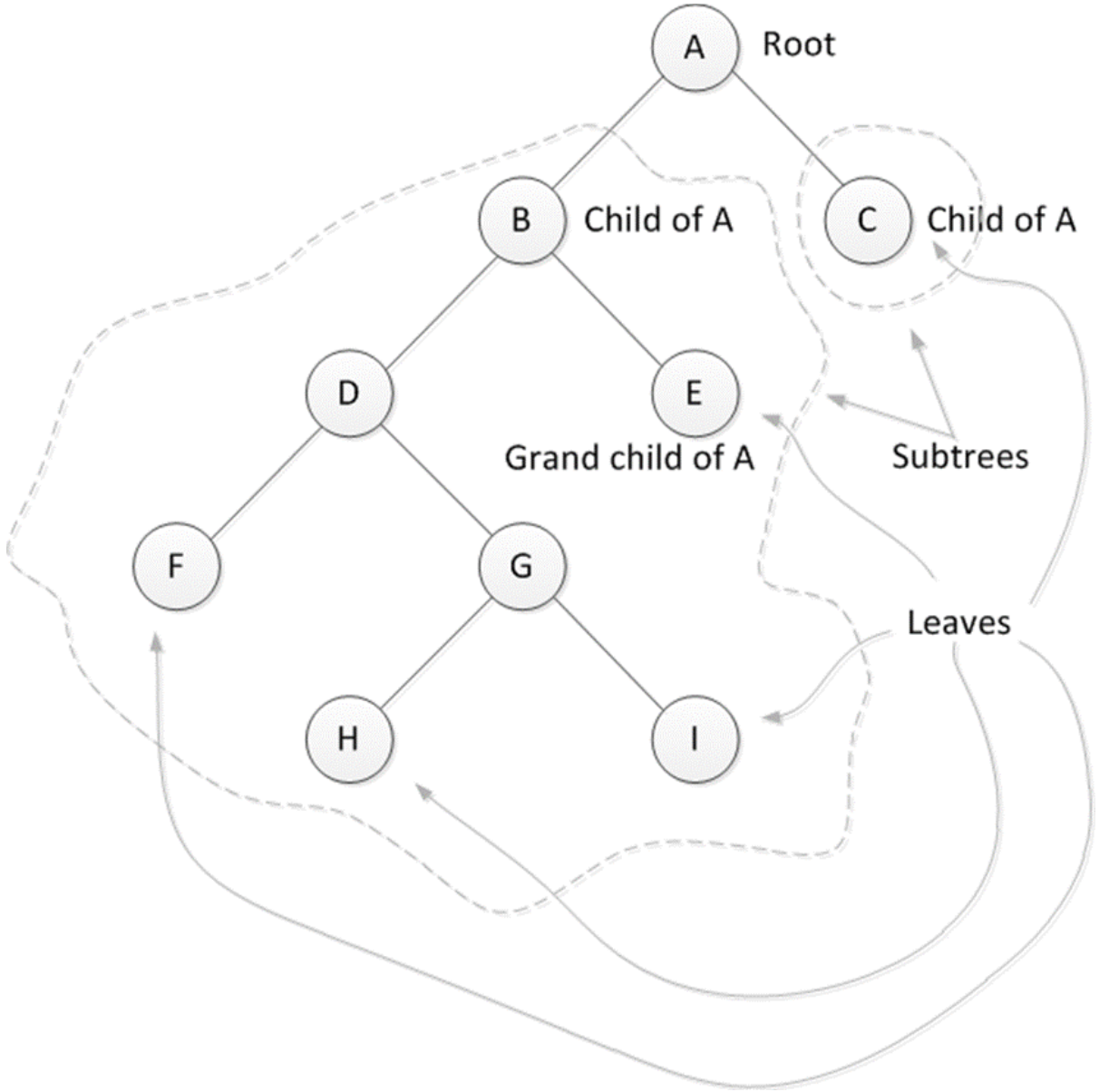
Tree Traversal – A Binary Tree



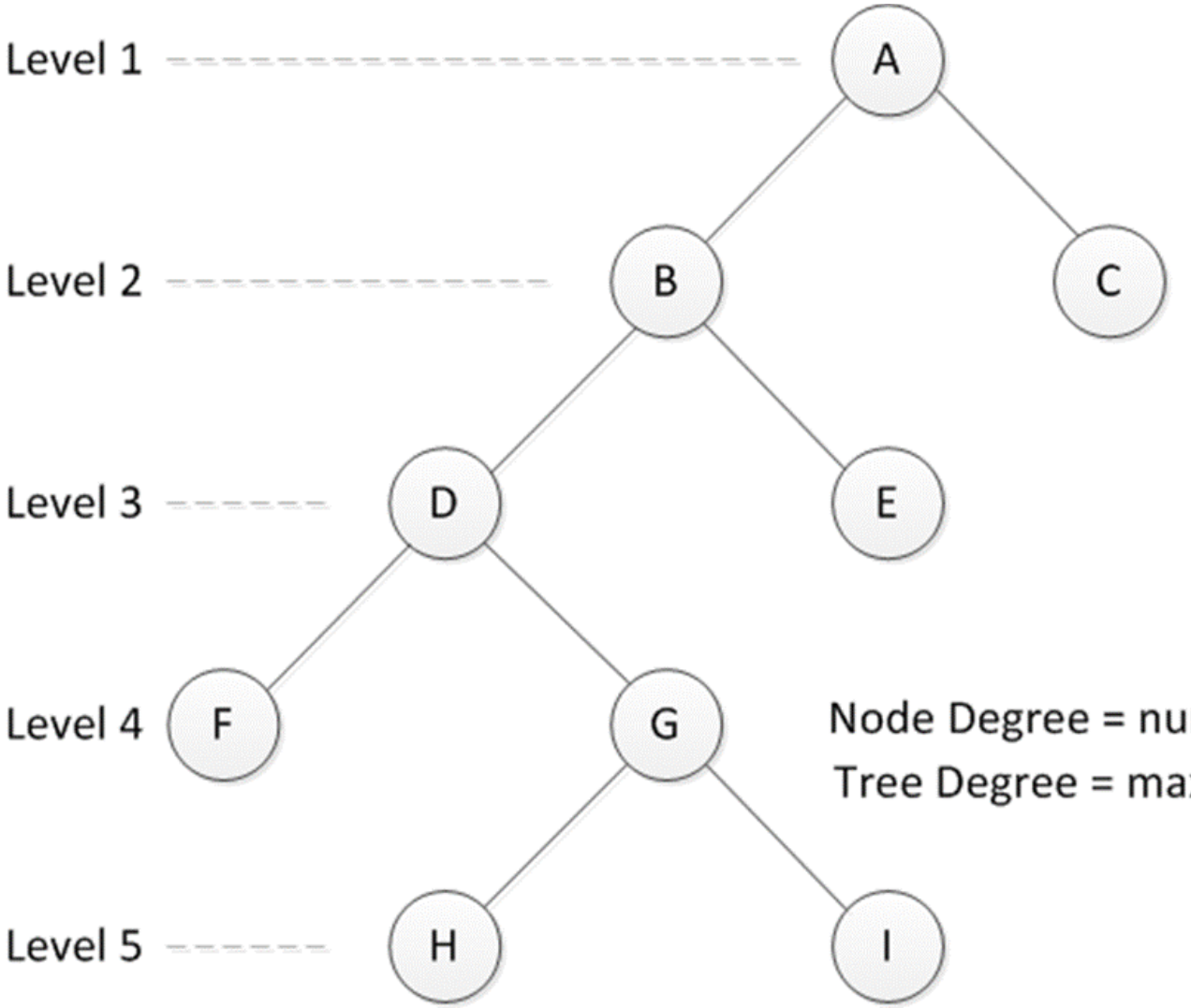

```
class node:
    def __init__(self, data, left=None, right=None):
        self.left = left
        self.right = right
        self.data = data
```

```
# Creating a binary tree
h=node("H")
i=node("I")
f=node("F")
g=node("G", h, i)
d=node("D", f, g)
e=node("E")
b=node("B", d, e)
c=node("C")
a=node("A", b, c)
```

Root, Children, leaves, subtrees



Levels, Node Degree, Tree Degree



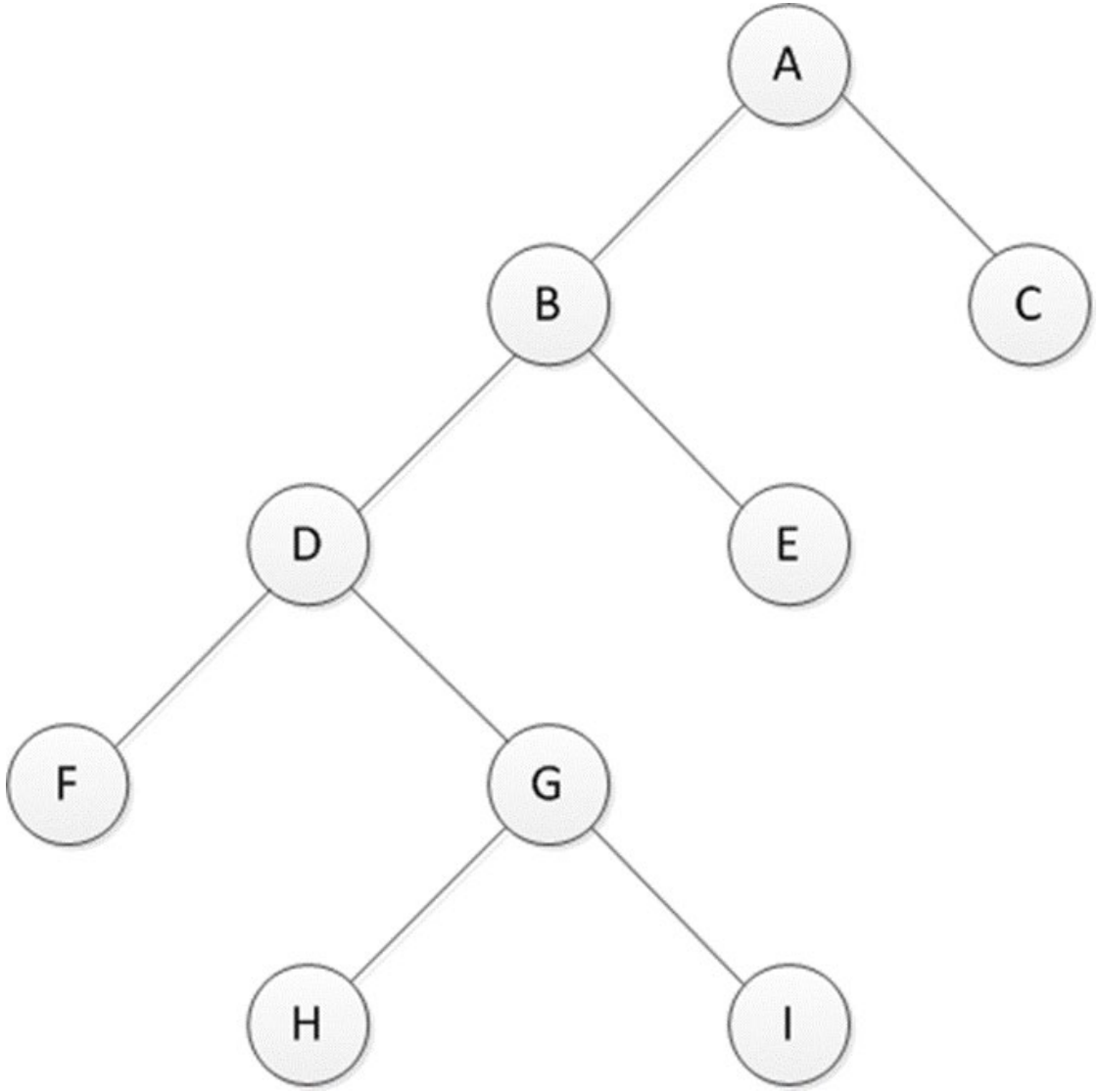
Node Degree = number of children

Tree Degree = max node degree

Inorder Traversal


```
# Inorder Traversal
def inorder(self):
    if self.left:
        self.left.inorder()
    print(self.data) # Action
    if self.right:
        self.right.inorder()
```

```
Inorder - Traversal
F
D
H
G
I
B
E
A
C
```



Binary Search Tree


```
def insert(self, data):
    # Compare the new value with the parent node
    if self.data:
        if data < self.data:
            if self.left is None:
                self.left = node(data)
            else:
                self.left.insert(data)
        elif data > self.data:
            if self.right is None:
                self.right = node(data)
            else:
                self.right.insert(data)
    else:
        self.data = data
```

```
# Inorder Traversal
def inorder(self):
    if self.left:
        self.left.inorder()
    print(self.data) # Action
    if self.right:
        self.right.inorder()
```

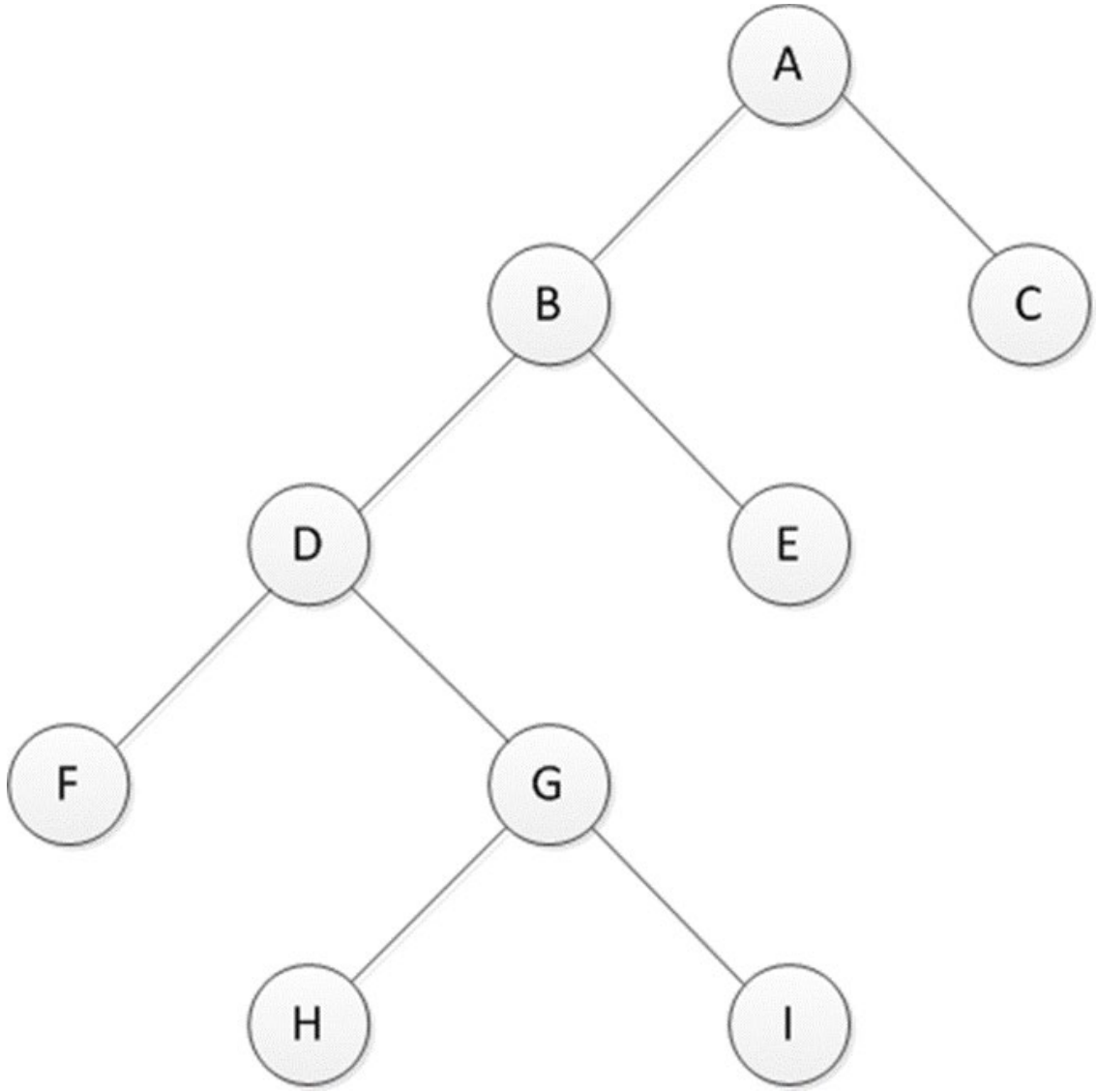
```
t=node(6)
t.insert(4)
t.insert(3)
t.insert(10)
t.insert(8)
t.insert(11)
t.insert(5)
```

```
Inorder - Traversal
3
4
5
6
8
10
11
```

Preorder Traversal


```
# Preorder Traversal
def preorder(self):
    print(self.data) # Action
    if self.left:
        self.left.preorder()
    if self.right:
        self.right.preorder()
```

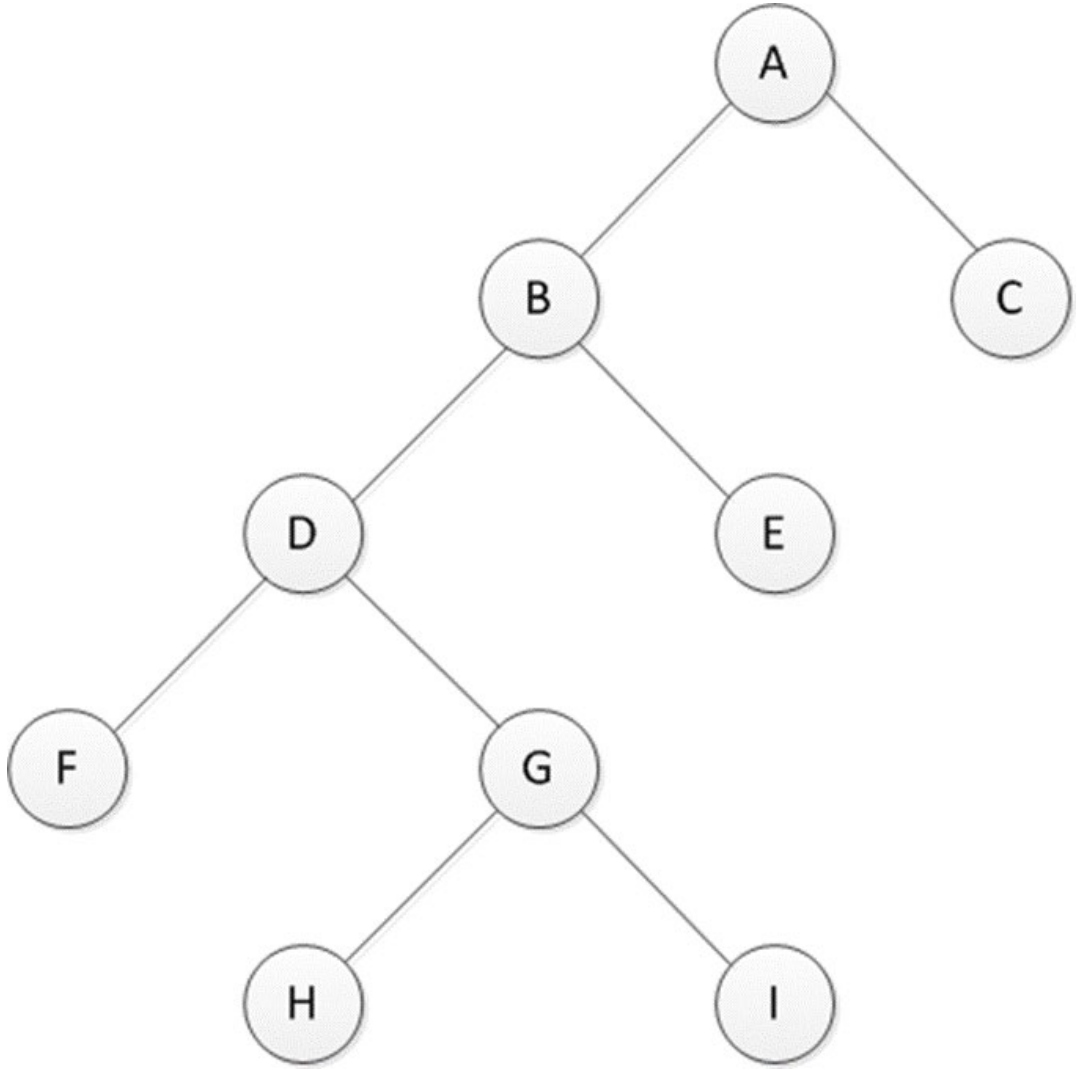
```
Preorder - Traversal
A
B
D
F
G
H
I
E
C
```



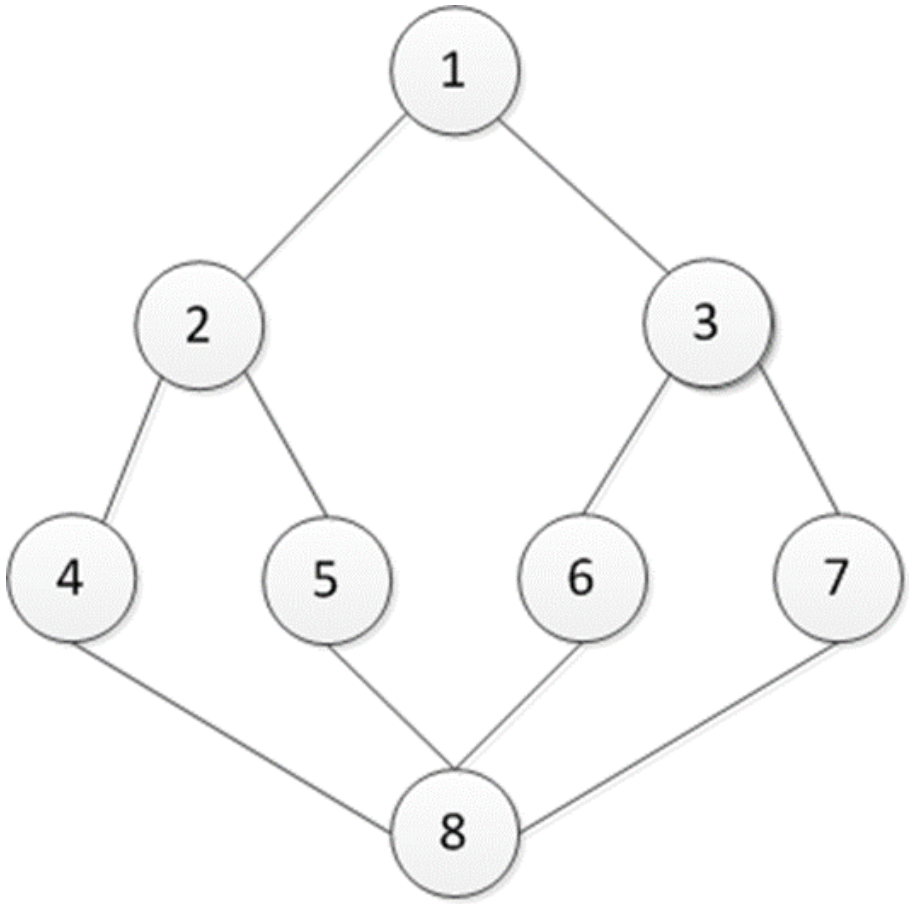
Postorder Traversal


```
#Postorder Traversal
def postorder(self):
    if self.left:
        self.left.postorder()
    if self.right:
        self.right.postorder()
    print(self.data) # Action
```

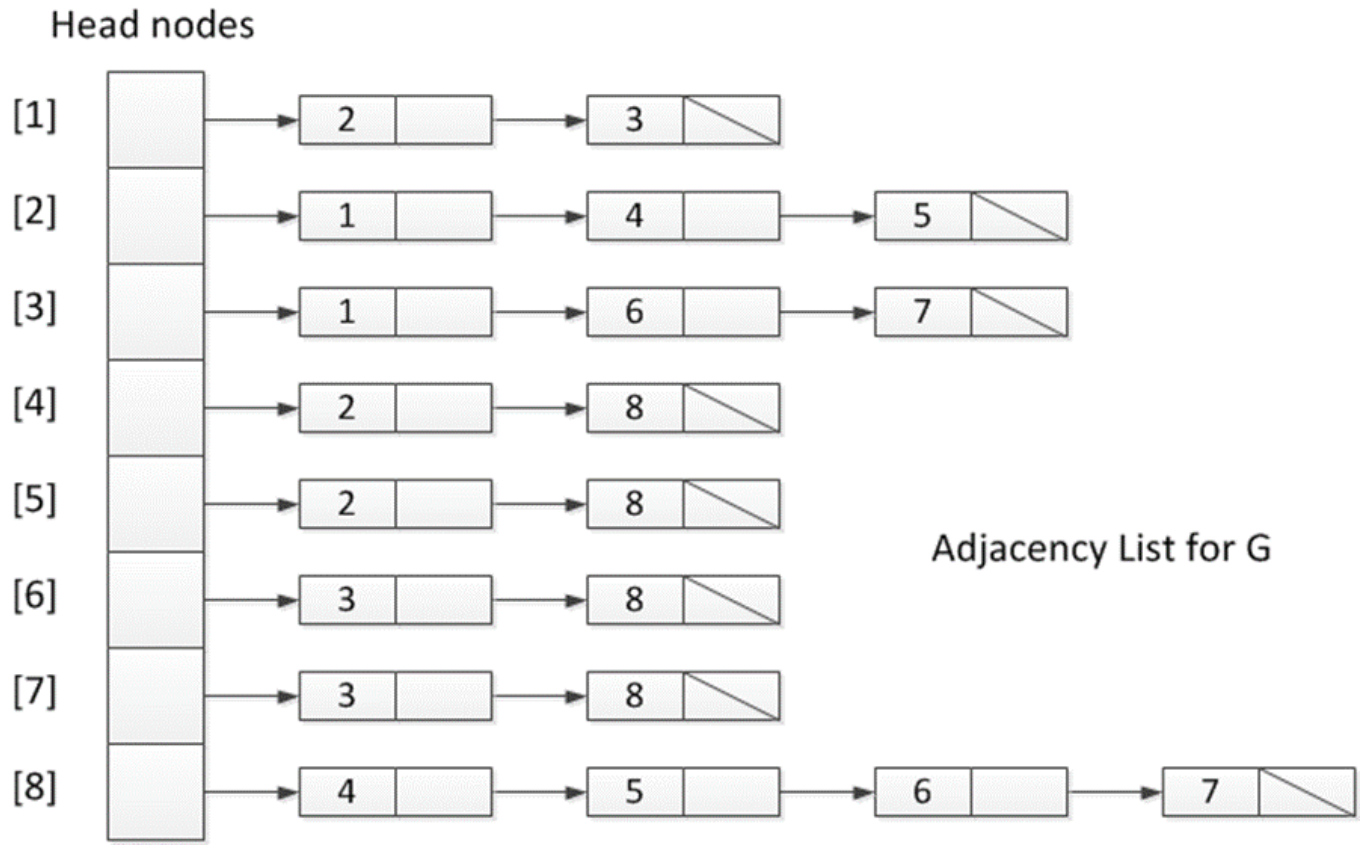
```
Postorder - Traversal
F
H
I
G
D
E
B
C
A
```



Graph Representations – Adjacency Matrix or Adjcency List



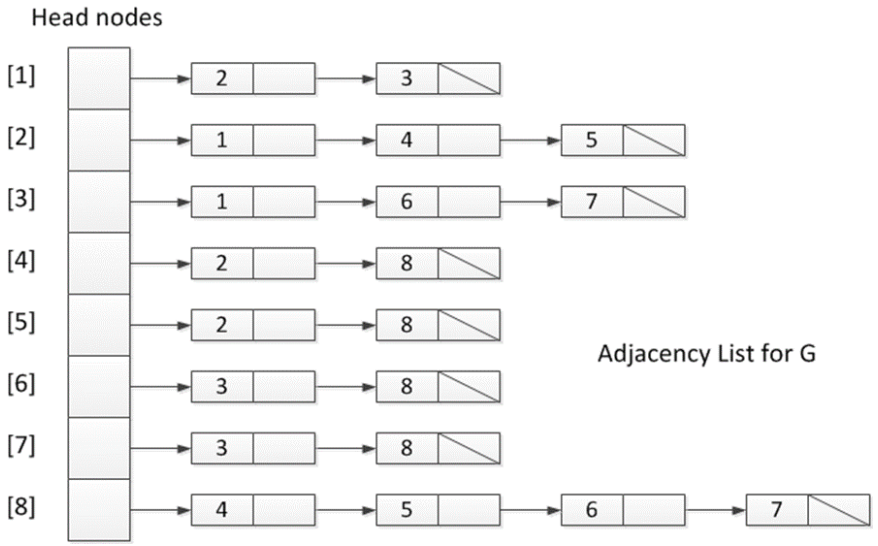
Undirected Graph G



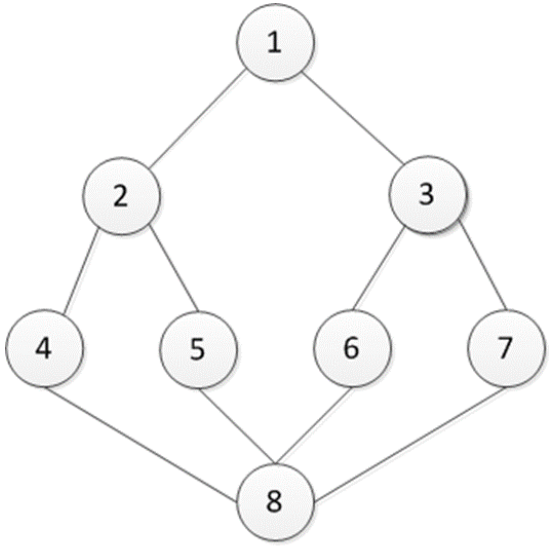
Adjacency List for G

Graph Representations –Adjacency List or Matrix


```
# Use Dictionary to represent a Graph
mygraph = { "1" : ["2","3"],
            "2" : ["1", "4", "5"],
            "3" : ["1", "6", "7"],
            "4" : ["2", "8"],
            "5" : ["2", "8"],
            "6" : ["3", "8"],
            "7" : ["3", "8"],
            "8" : ["4", "5", "6", "7"]
          }

# Print graph (Dictionary)
print(mygraph)
```




```
{'1': ['2', '3'], '2': ['1', '4', '5'], '3': ['1', '6', '7'], '4': ['2', '8'], '5': ['2', '8'], '6': ['3', '8'], '7': ['3',
'8'], '8': ['4', '5', '6', '7']}
```



Undirected Graph G

A Python Graph Class


```
class graph:
    def __init__(self,gdict=None):
        if gdict is None:
            gdict = []
        self.gdict = gdict

# Add the vertex as a key
    def addVertex(self, vrtx):
       if vrtx not in self.gdict:
            self.gdict[vrtx] = []

# Get the keys of the dictionary
    def getVertices(self):
        return list(self.gdict.keys())

    def edges(self):
        return self.findedges()

# Find the distinct list of edges
    def getEdges(self):
        edgename = []
        for vrtx in self.gdict:
            for nxtvrtx in self.gdict[vrtx]:
                if {nxtvrtx, vrtx} not in edgename:
                    edgename.append({vrtx, nxtvrtx})
        return edgename
```

```

# Add the new edge
    def addEdge(self, edge):
        edge = set(edge)
        (vrtx1, vrtx2) = tuple(edge)
        if vrtx1 in self.gdict:
            self.gdict[vrtx1].append(vrtx2)
        else:
            self.gdict[vrtx1] = [vrtx2]

g=graph(mygraph)
g.addVertex("10")
g.getVertices()
```

```
['1', '2', '3', '4', '5', '6', '7', '8', '10']
```

Creating an instance of the graph

In [9]:
```
g1=graph({})
g1.addVertex("1")
g1.addVertex("2")
g1.addVertex("3")
g1.addVertex("4")
g1.addVertex("5")
g1.addVertex("6")
g1.addVertex("7")
g1.addVertex("8")
```

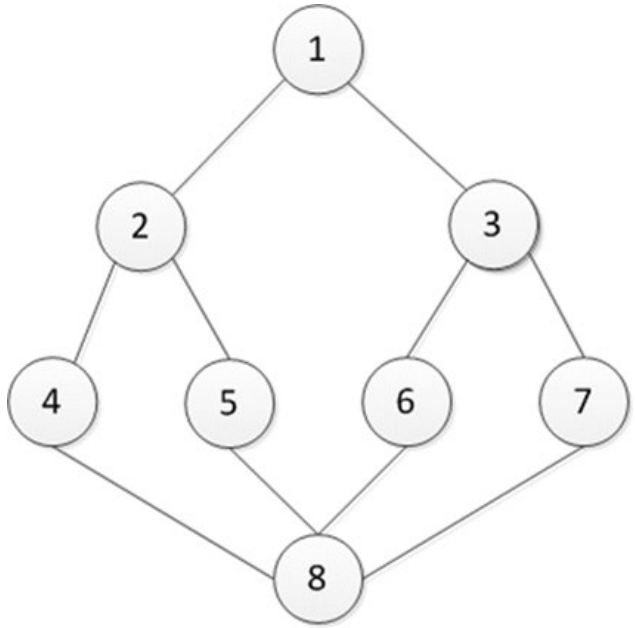
In [10]:
```
g1.getVertices()
```

Out[10]: ['1', '2', '3', '4', '5', '6', '7', '8']

In [11]:
```
g1.addEdge({"1","2"})
g1.addEdge({"1","3"})
g1.addEdge({"2","4"})
g1.addEdge({"2","5"})
g1.addEdge({"3","6"})
g1.addEdge({"3","7"})
g1.addEdge({"4","8"})
g1.addEdge({"5","8"})
g1.addEdge({"6","8"})
g1.addEdge({"7","8"})
```

In [12]:
```
g1.getEdges()
```

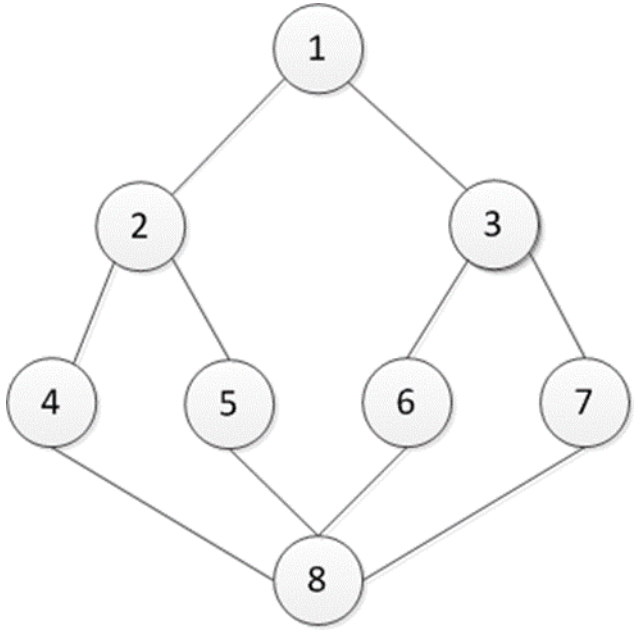
Out[12]:
```
[{'1', '2'},
 {'1', '3'},
 {'3', '6'},
 {'3', '7'},
 {'2', '4'},
 {'2', '5'},
 {'5', '8'},
 {'6', '8'},
 {'4', '8'},
 {'7', '8'}]
```



Undirected Graph G

Depth First Search (DFS) Pseudocode

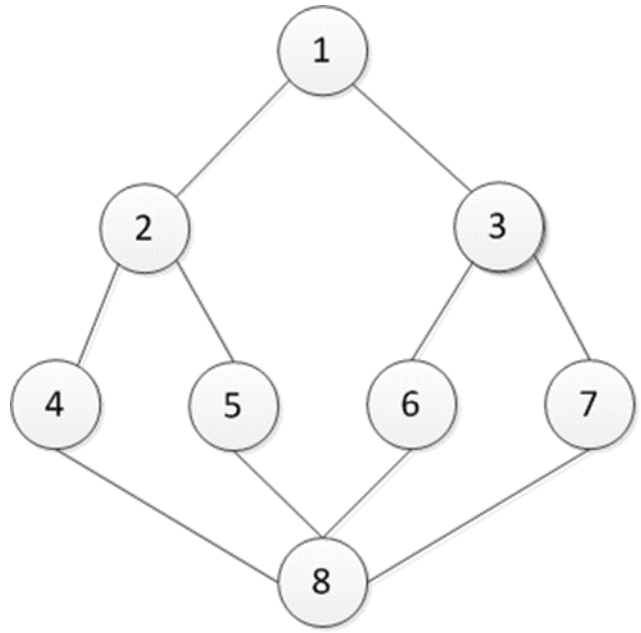

```
void DFS(int v)
// A depth first search of G is carried out beginning
// at vertex v. For any node i, visited[i]==1 if i has
// already been visited. The graph G and array visited[]
// are global; visited[] is initialized to zero.
{
   visited[v]=1;
   for each vertex w adjacent from v
   {
       if (!visited[w]) DFS(w);
   }
}
```



Undirected Graph G

Breadth First Search (BFS) Pseudocode

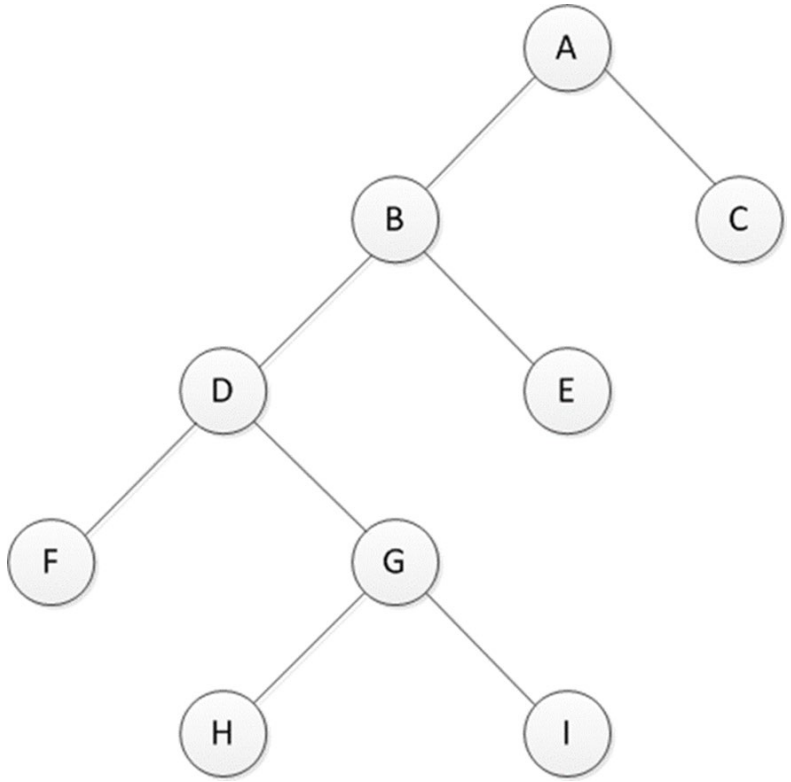

```
void BFS(int v)
// A breadth first search of G is carried out beginning
// at vertex v. For any node i, visited[i]==1 if i  has
// already been visited. The graph G and array visited[]
// are global; visited[] is initialized to zero.
{
  int u=v; Queue q(size);
  //q is a queue of unexplored vertices
  visited[v]=1;
  do{
     for all vertices w adjacent from u {
        if (visited[w]==0){
           q.AddQ(w); // w is unexplored.
           visited[w]=1;
        }
     }
     if (q.Qempty()) return; //no unexplored vertex
     q.DeleteQ(u); // Get the first unexplored vertex
  } while (1);
 }
```



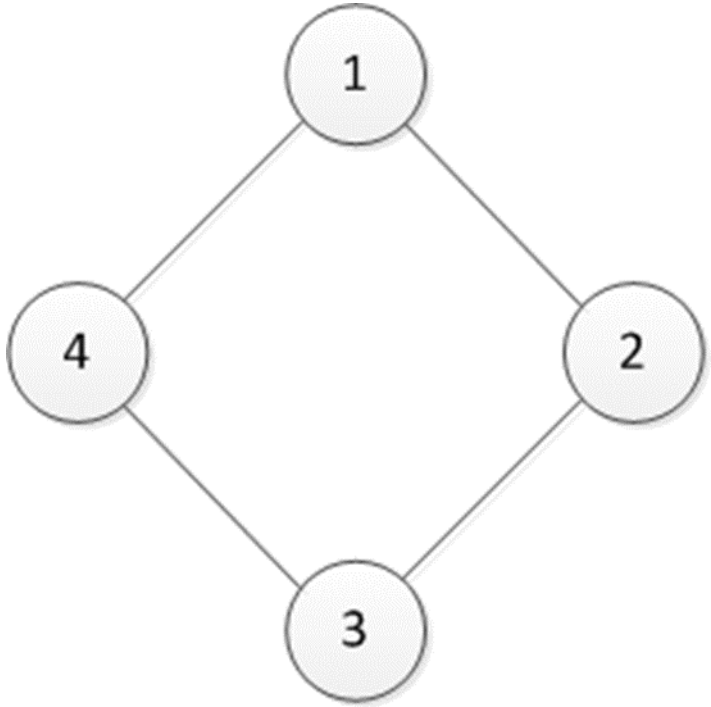
Undirected Graph G

Level Traversal for Trees - BFS


```
# An inefficient way

def height(node):
    if node is None:
        return 0
    else :
        # Compute the height of each subtree
        lheight = height(node.left)
        rheight = height(node.right)

        #Use the larger one
        if lheight > rheight :
            return lheight+1
        else: # Print nodes at a given level
            return rheight+1

# Print nodes at a given level
def printGivenLevel(node, level):
    if node is None:
        return
    if level == 1:
        print(node.data)
    elif level > 1 :
        printGivenLevel(node.left , level-1)
        printGivenLevel(node.right , level-1)

#Level Traversal
def printLevelOrder(node):
    h = height(node)
    for i in range(1, h+1):
        printGivenLevel(node, i)
```

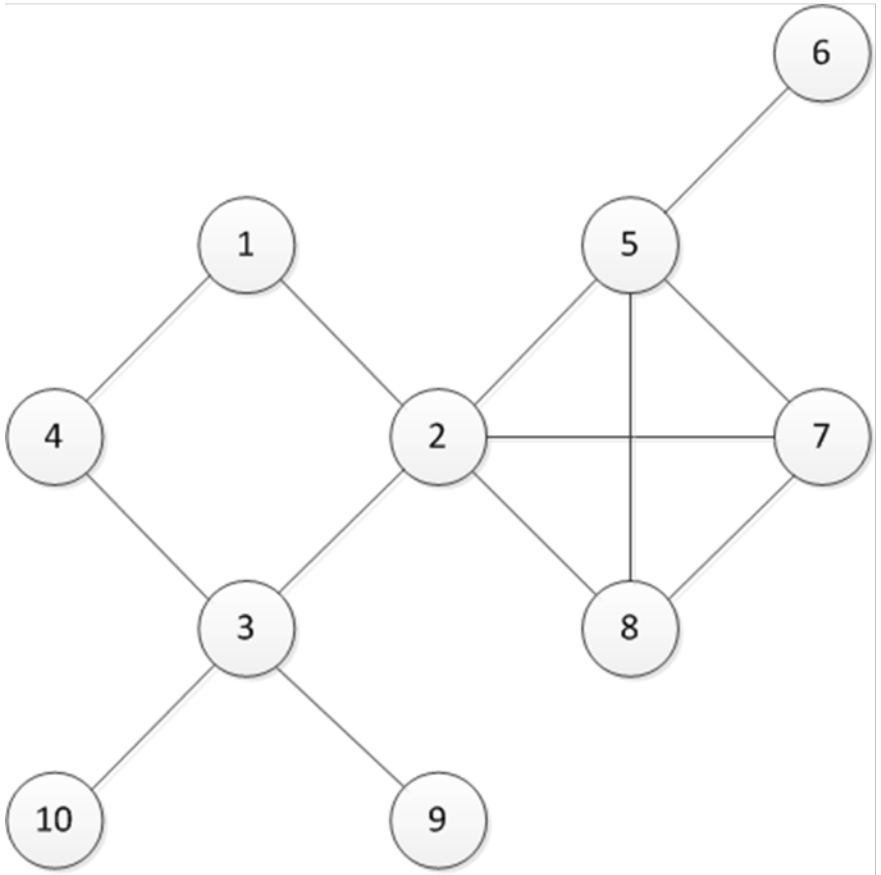


A Biconnected Graph

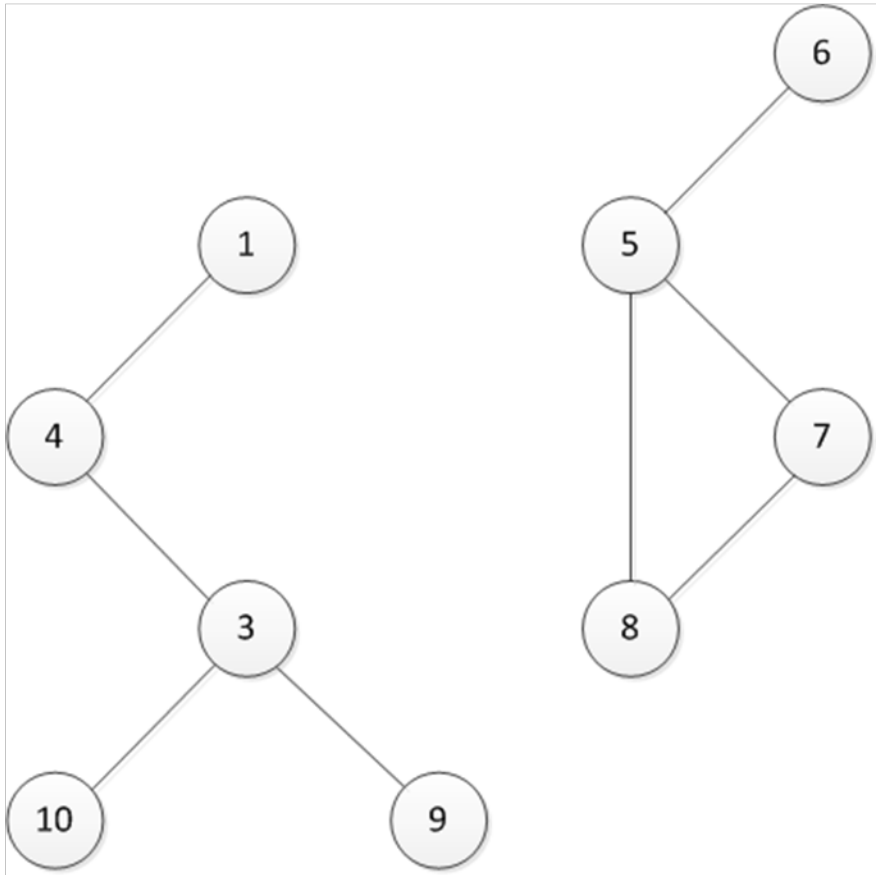


A Biconnected Graph

Articulation Point

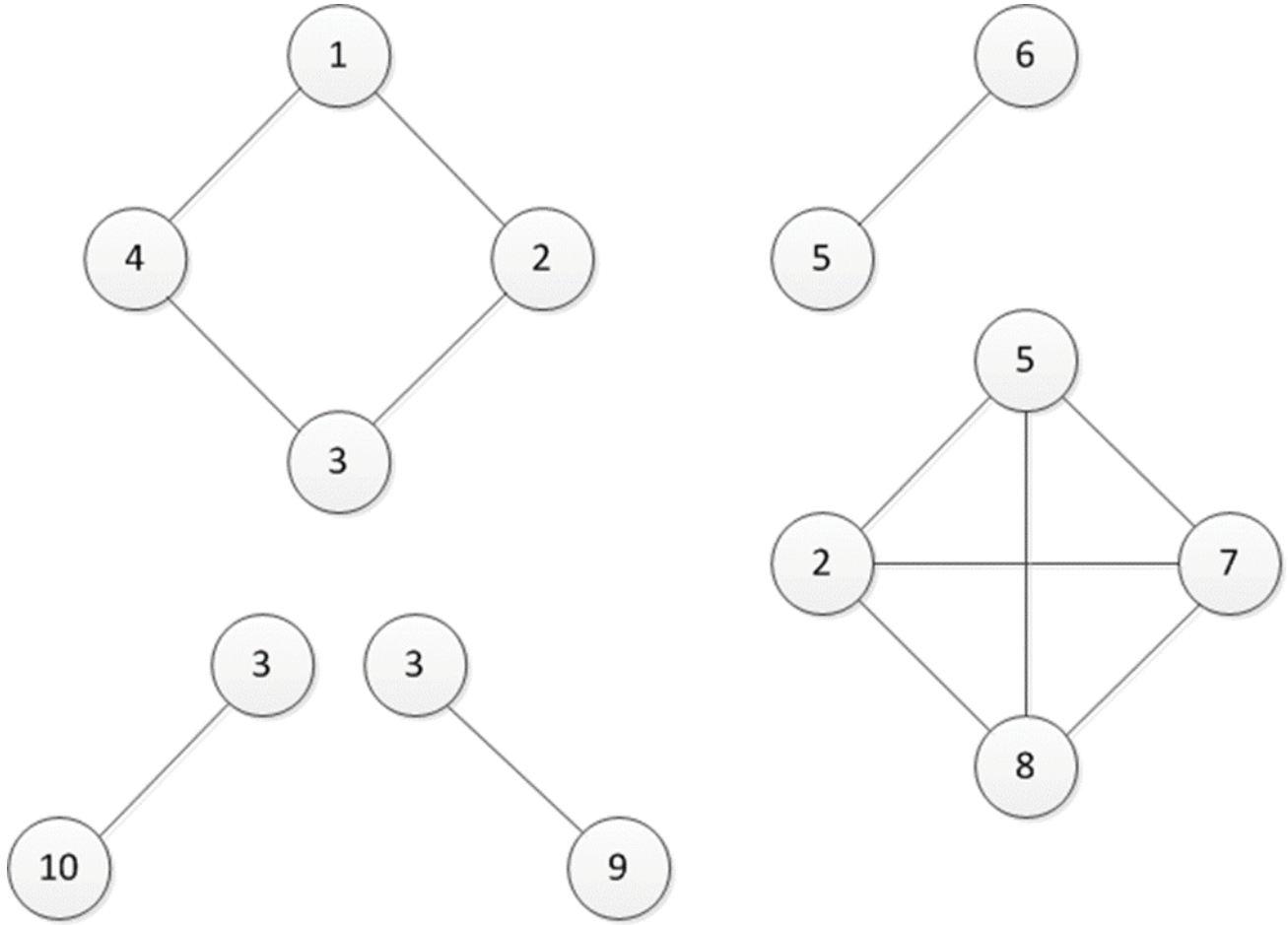


Graph G



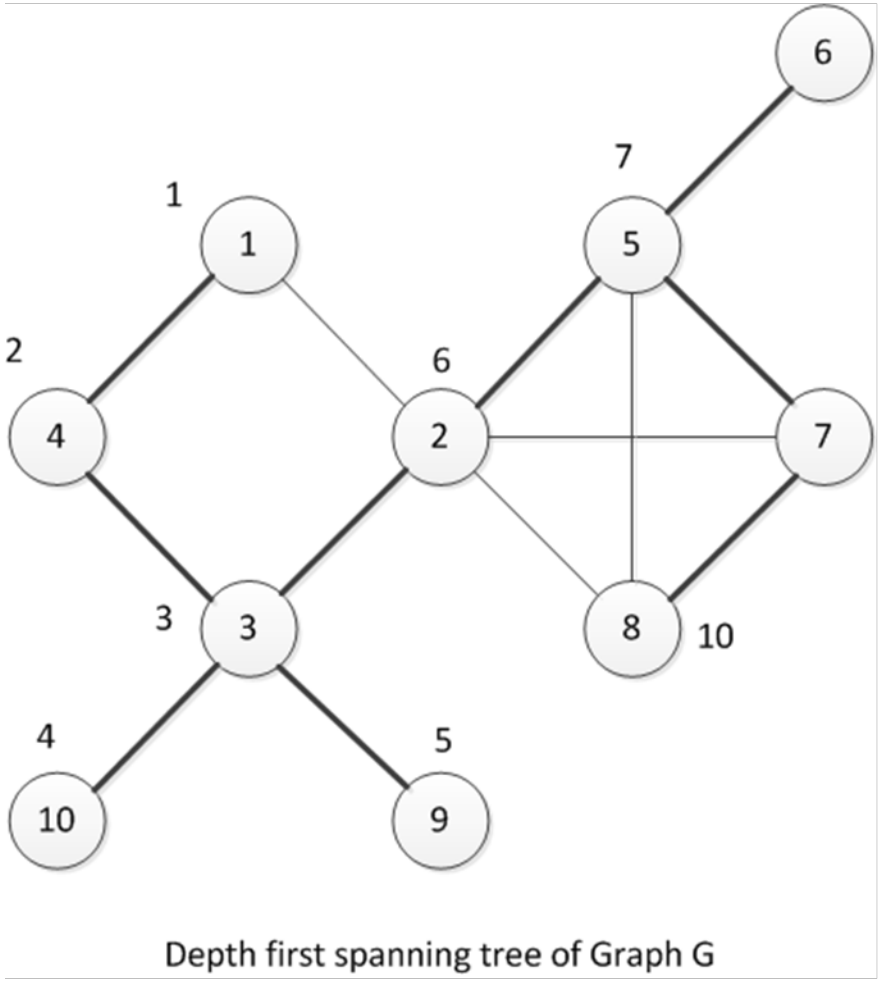
After Deleting vertex 2

Biconnected Components of G

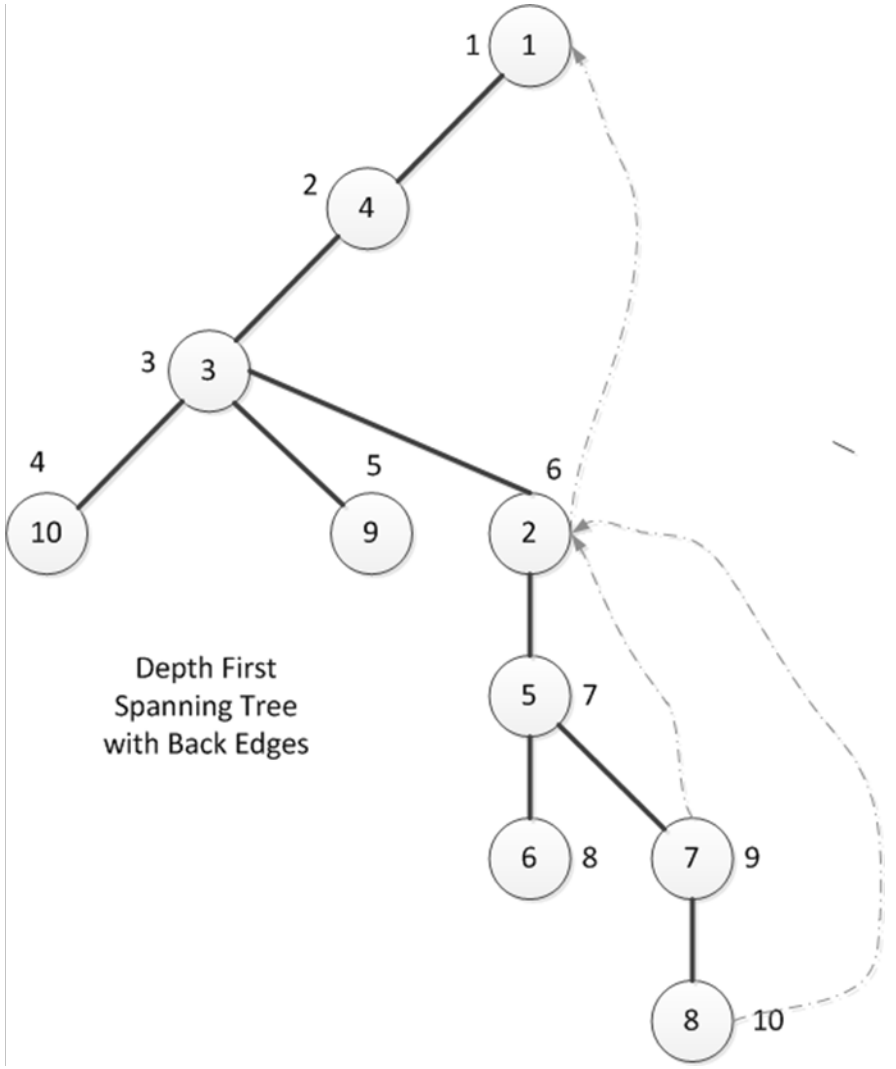


Biconnected components of Graph G

DF Spanning Tree with Back Edge



Depth first spanning tree of Graph G

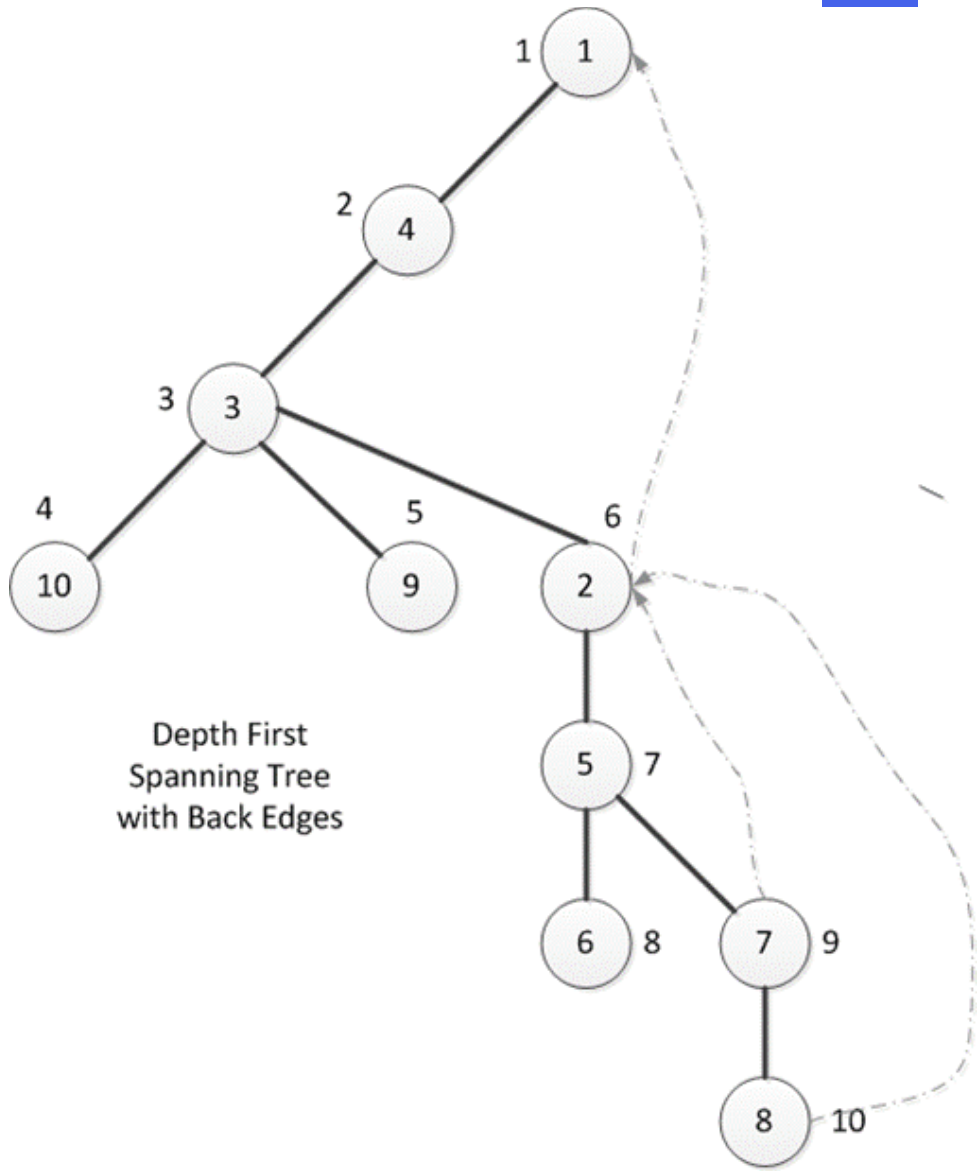


Depth First
Spanning Tree
with Back Edges

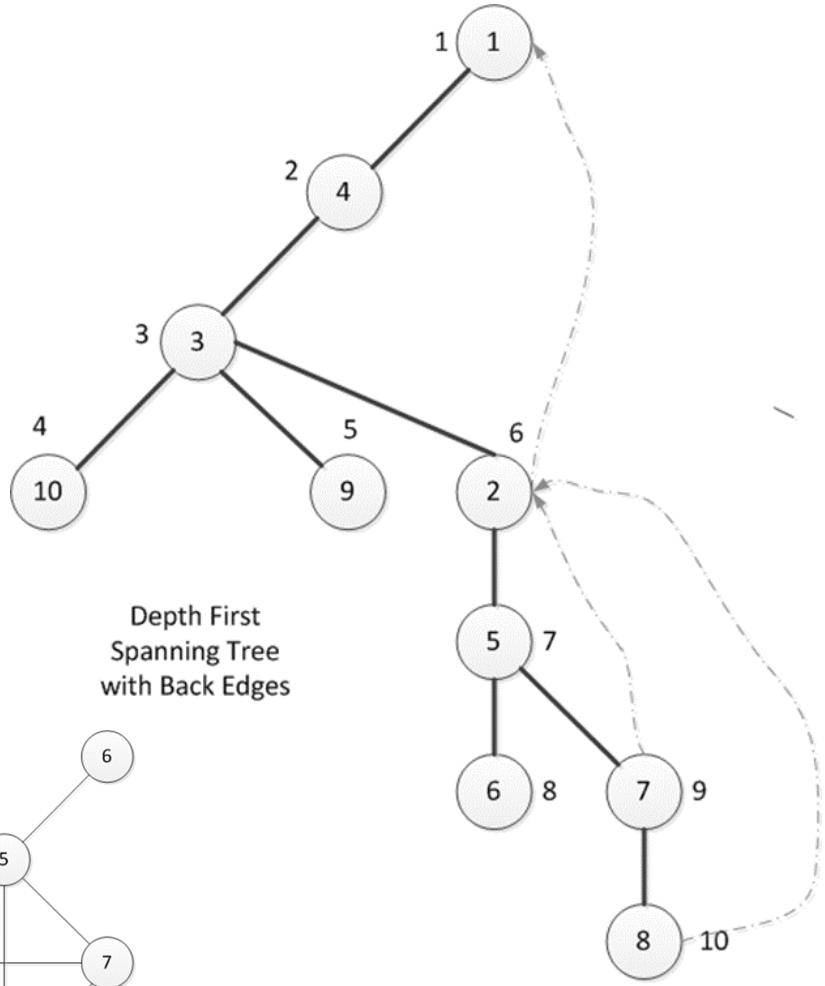
DF Spanning Tree with Back Edge

Lemma The root node of a depth first spanning tree is an articulation point iff it has at least two children. Furthermore, if u is any other vertex, then it is not an articulation point iff from every child w of u it is possible to reach an ancestor of u using only a path made up of descendents of w and a back edge.

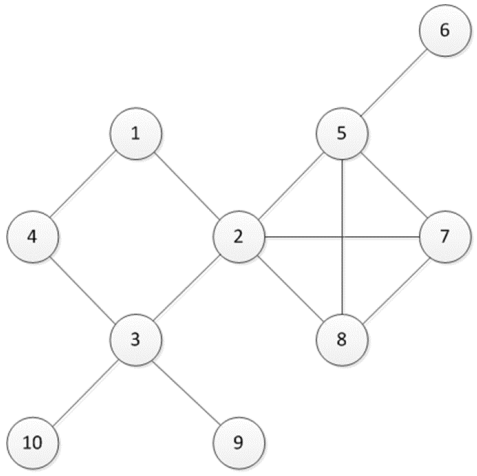
$$L[u] = \min\{ \quad dfn[u],$$
$$\min\{L[w] | w \text{ is a child of u}\},$$
$$\min\{dfn[w] | (u, w) \text{ is a back edge}\}\}$$



dfn[u] and L[u]



Depth First
Spanning Tree
with Back Edges



Graph G

$$L[u] = \min\{\quad dfn[u],$$
$$\min\{L[w] | w \text{ is a child of u}\},$$
$$\min\{dfn[w] | (u,w) \text{ is a back edge}\}\}$$

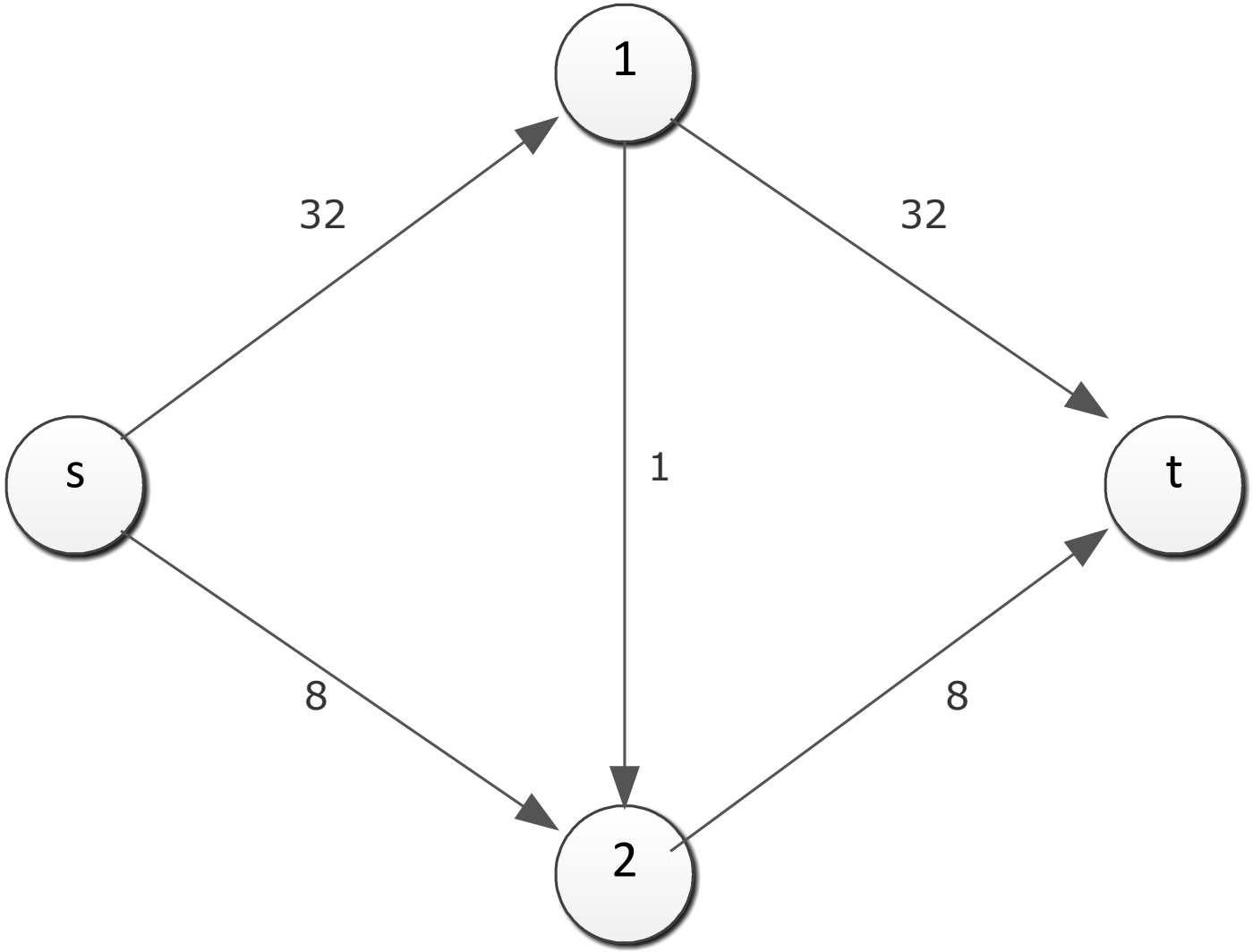
Vertex u	1	2	3	4	5	6	7	8	9	10
$dfn(u)$	1	6	3	2	7	8	9	10	5	4
$L(u)$	1	1	1	1	6	8	6	6	5	4

u is an articulation point iff u has a child w such that $L[w] \geq dfn[u]$.

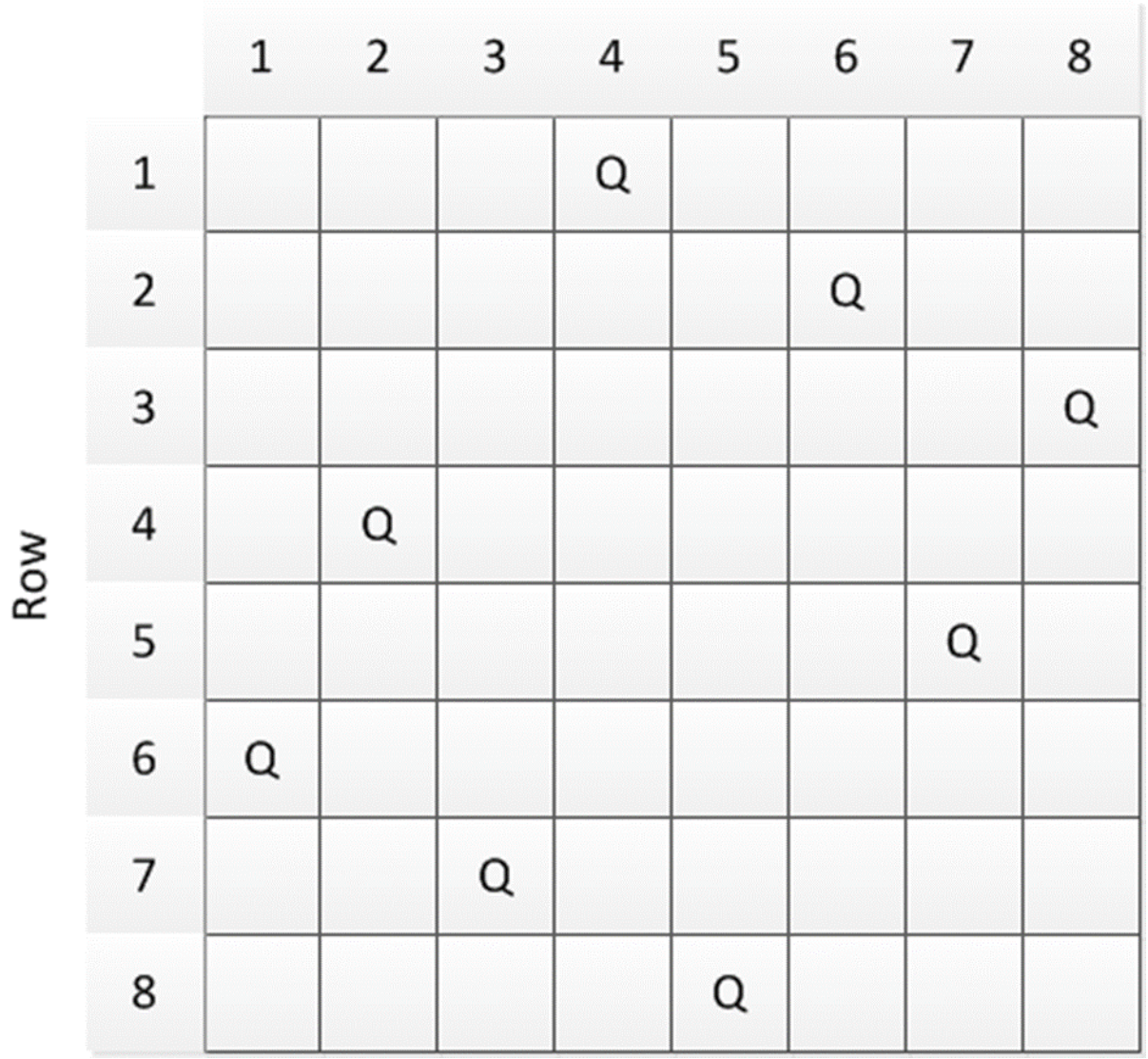
Algorithm to find Articulation Point


```
void Art(int u, int v)
// u is a start vertex for DFS. v is its parent if any in
// the depth first spanning tree. It is assumed that
// the global array dfn is initialized to zero and the
// global variable num is initialized to 1. n is the
// number of vertices in G.
{
  extern int dfn[], L[], num, n;
  dfn[u] = num; L[u]=num; num++;
  for each vertex w adjacent from u{
     if (!dfn[w]){
        Art(w,u); // w is unvisited
        L[u] = min(L[u],L[w]);
     }
    else if (w!=v) L[u]=min(L[u],dfn[w]);
}
```

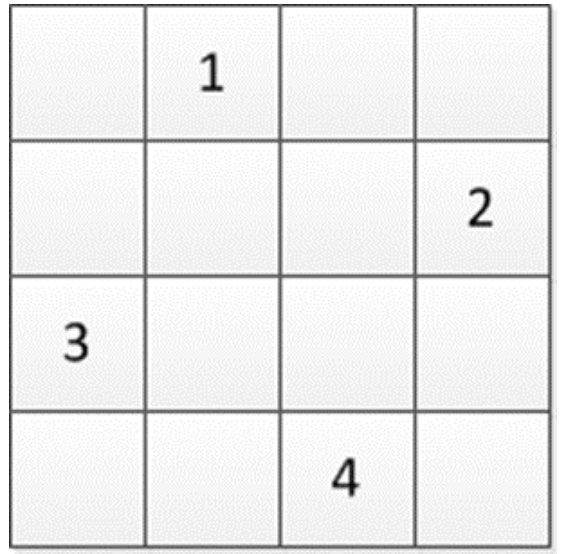
Maximum Flow – Ford and Fulkerson and others

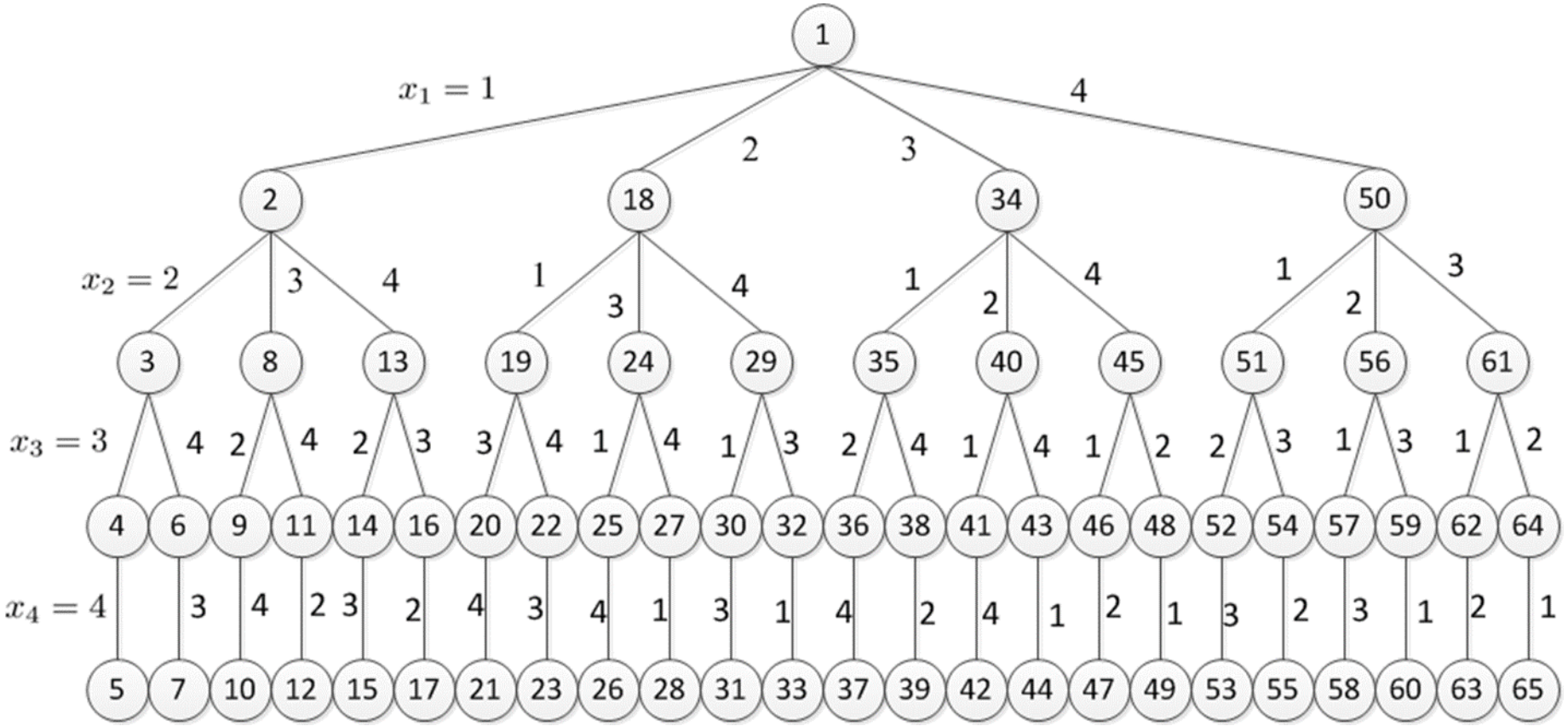


Eight-Queens

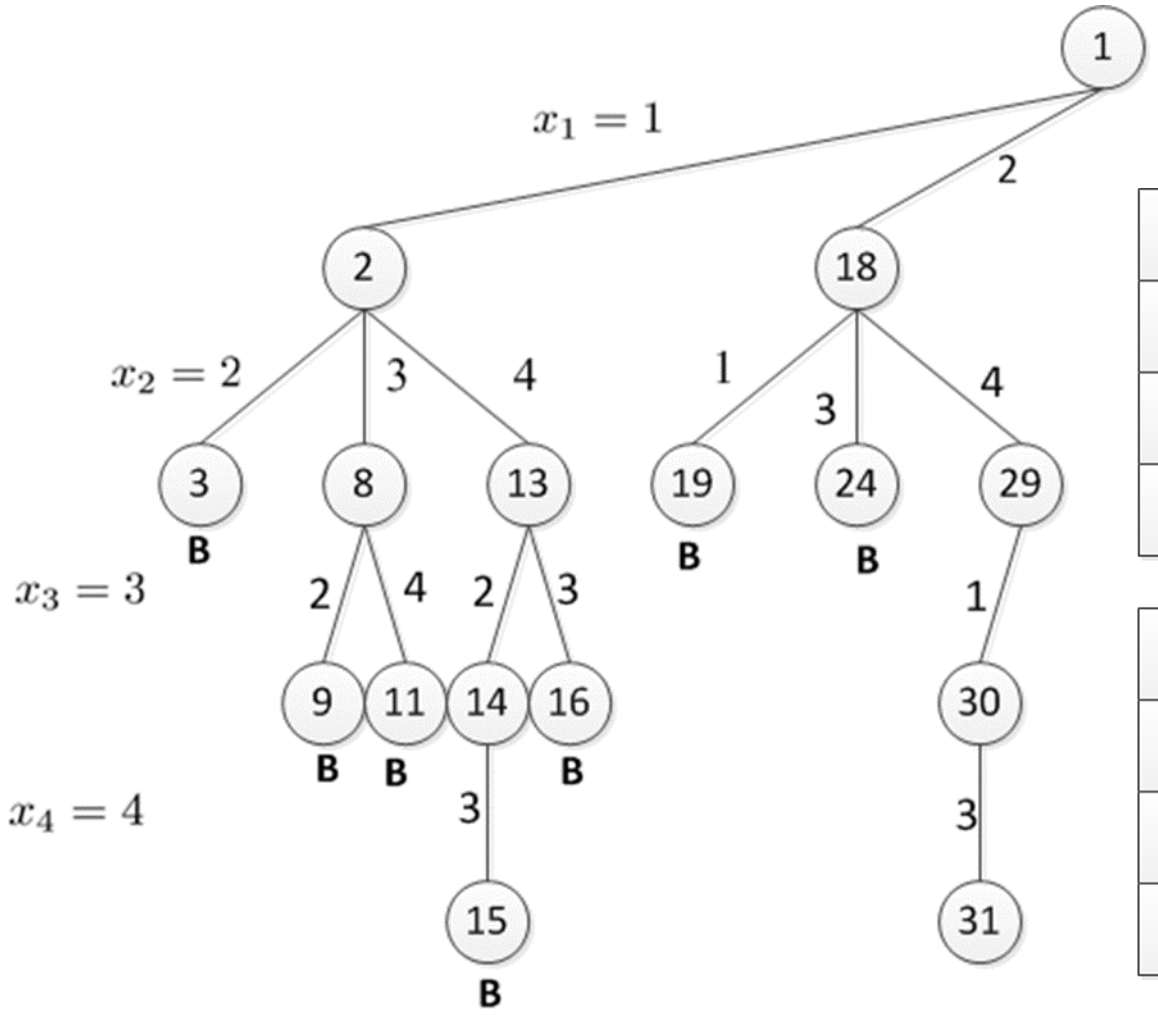


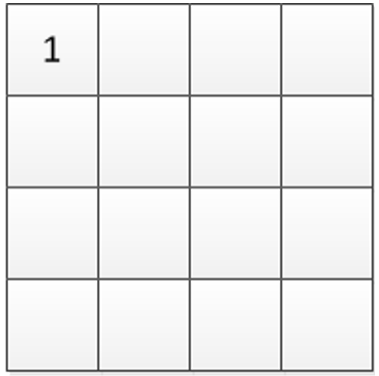
Solution Space Representation – Four Queens

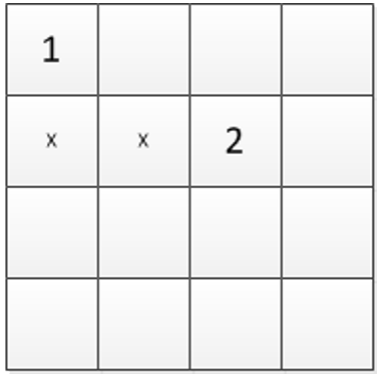


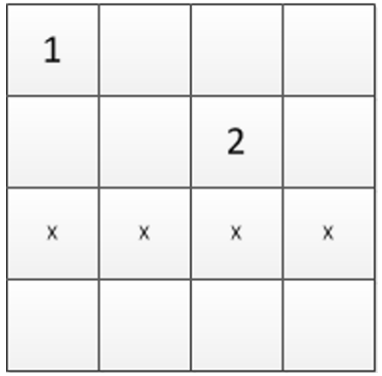


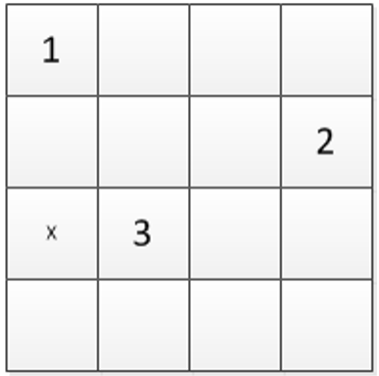
A Better Representation

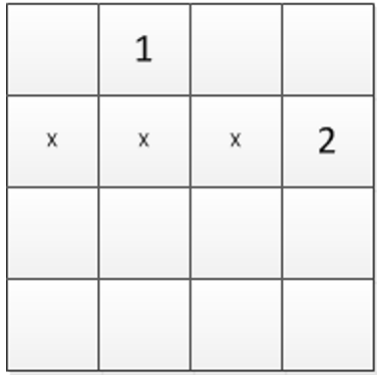


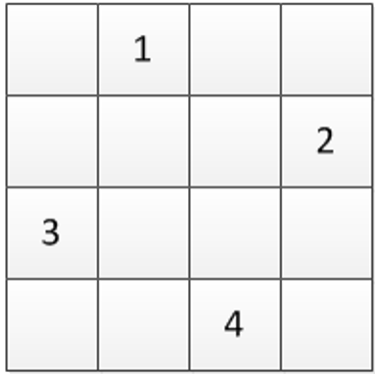












Recursive Backtracking Framework


```
void Backtrack(int k)
//T(x[1], ... , x[k-1]) is the set of possible values for
// x[k] given x[1], x[2], ... , x[k-1].
// B(x[1], ... , x[k]) is false, this partial solution cannot
// lead to a solution else B is true.
{
  for (each x[k] such that x[k] in T(x[1],...,x[k-1]){
     if(B(x[1], ... , x[k])){
       if (x[1], x[2], ... , x[k] is a path to an answer node)
          output x[1:k];
          // insert return; if only one return is needed
       if (k<n) Backtrack(k+1);
     }
  }
}
```

Iterative Backtracking Framework


```
void IBacktrack(int k)
{
 int k=1;
 while (k){
   if (there remains an untried x[k] such that x[k] is in
      T(x[1],...,x[k-1]) and B(x[1], ... , x[k]) is true {
       if (x[1], x[2], ... , x[k] is a path to an answer node)
       output x[1:k];
       k++; //next set
    }
    else k--; // backtrack to the previous set
  }
}
```

C++ Code for Eight Queen


```
bool Place(int k, int i)
// Returns true if queen can be placed in kth row and and ith column
// Otherwise it returns false. x[] is a global arrany whose first (k-1)
// values have been set.
// abs(r) returns the absolute value of r.
{
 for (int j=1; j<k; j++)
  if ((x[j]==i) || // in same column
     (abs(x[j]-i)==abs(j-k)))
      // same diagonal
     return false;
  return true;
}
```

```
void Backtrack(int k)
//T(x[1], ... , x[k-1]) is the set of possible values for
// x[k] given x[1], x[2], ... , x[k-1].
// B(x[1], ... , x[k]) is false, this partial solution cannot
// lead to a solution else B is true.
{
  for (each x[k] such that x[k] in T(x[1],...,x[k-1]){
     if(B(x[1], ... , x[k])){
       if (x[1], x[2], ... , x[k] is a path to an answer node)
          output x[1:k];
          // insert return; if only one return is needed
       if (k<n) Backtrack(k+1);
     }
  }
}
```

```
void NQueens(int k, int n)
// Using Backtracking, this procedure prints all
// possible placements of  n queens on an nxn
// chessboard so that they are non-attacking
{
 for (int i=1; i<=n; i++){
    if (Place(k,i)){
       x[k]=i;
       if (k==n) {
          for (int j=1; j<=n; j++){
             cout << x[j] << ' ';
             cout << endl;
          }
          else NQueens(k+1,n);
       }
    }
}}
```

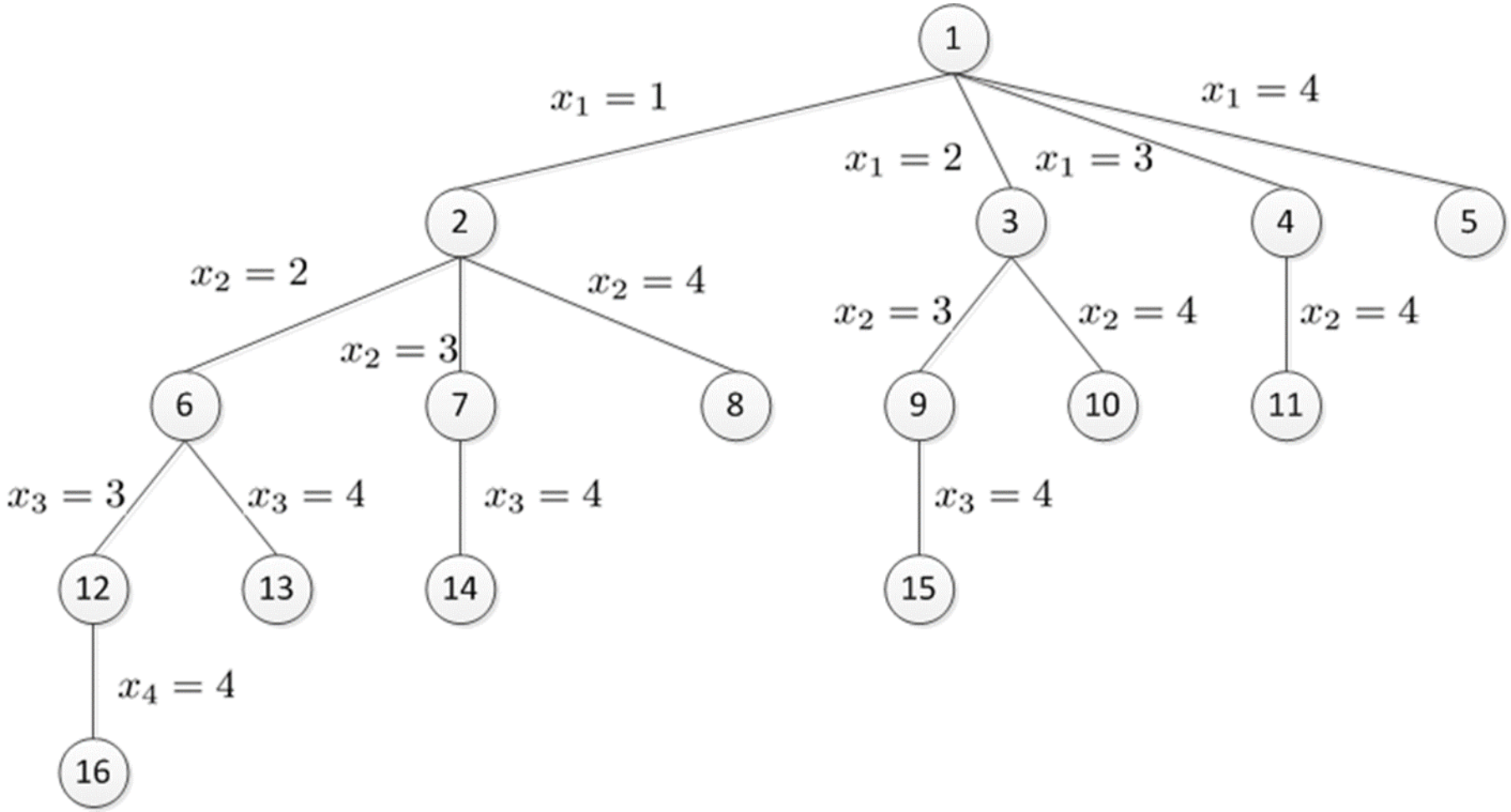
Subset Sum

Given positive numbers w_i, $1 \leq i \leq n$, and m, find all subsets of the w_i whose sums are m.

For example, if $n = 4$, $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$ and $m = 31$, then the desired subsets are $(11, 13, 7)$ and $(24, 7)$. We can represent the solution vector by giving the indices of these w_i.

Now the solutions are described by the vector $(1, 2, 4)$ and $(3, 4)$

Subset Sum – A Possible Solution Space Representation



Subset Sum

Let's assume we use the fixed tuple strategy. Childen of any node can be easily generated. For a node at level i, the left child correponds to $x_i = 1$ and the right to $x_i = 0$.

A simple choice for the bounding functions is $B_k(x_1, \ldots, x_k) =$ true iff

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

Subset Sum – Bounding Functions

Let's assume we use the fixed tuple strategy. Childen of any node can be easily generated. For a node at level i, the left child correponds to $x_i = 1$ and the right to $x_i = 0$.

A simple choice for the bounding functions is $B_k(x_1, \ldots, x_k) =$ true iff

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

Subset Sum – Bounding Functions

$x_1, \ldots, x_n$ cannot lead to an answer node if this condition is not satisfied. The bounding function can be further strengthen by making sure that w_i's are initially in nondecreasing order. In this case, $x_1, \ldots, x_k$ cannot lead to an answer node if $\sum_{i=1}^{k} w_i x_i + w_{k+1} > m$

Therefore $B_k(x_1, \ldots, x_k)$ is true iff

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

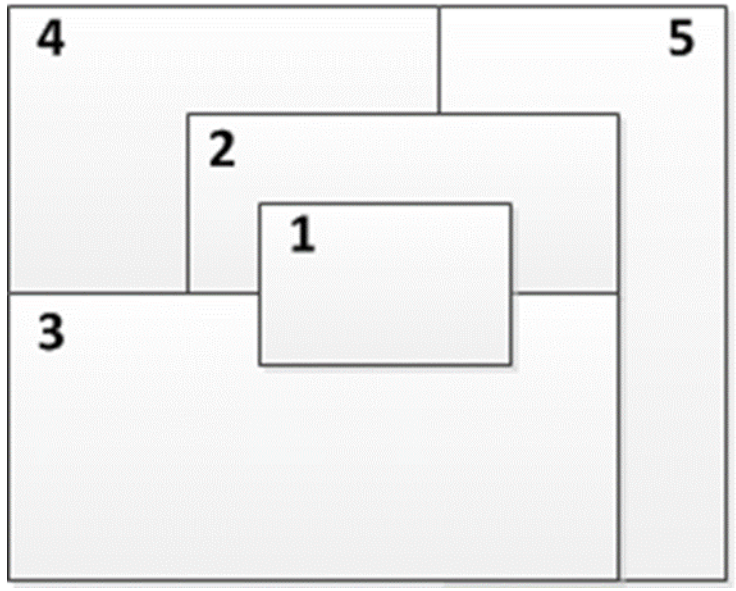
and $\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m.$

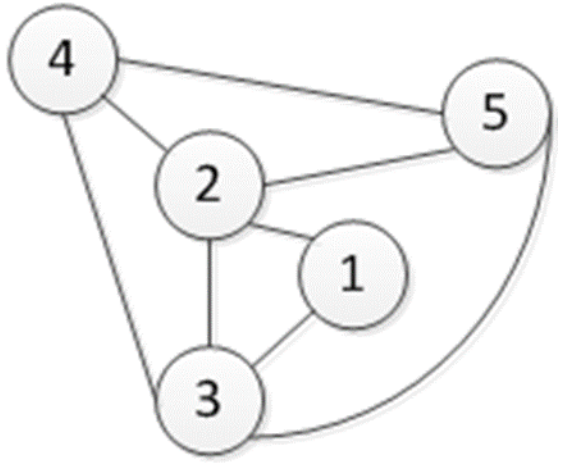
The Code

//Find all subsets of w[1:n] that sum to m. The values of
//x[j], $1 \leq i \leq n$ have already been determined.
// $s = \sum_{j=1}^{k-1} w[j]*x[j]$ and $r = \sum_{j=k}^{n} w[j]$.
// w[j]'s are in nondecreasing order.
// It is assumed that $w[1] \leq m$ and $\sum_{i=1}^{n} w[i] \geq m$.

```
void SumofSub(float s, int k, float r){
  x[k]=1;
  if (s+w[k]==m){//subset found
     for (int j=1; j<=k; j++) cout <<x[j] << ' ';
     cout << endl;
  }
  else if (s+w[k]+w[k+1]<=m)
     SumofSub(s+w[k],k+1,r-w[k]);
  if ((s+r-w[k]>=m) && (s+w[k+1]<=m)){
     x[k]=0;
     SumofSub(s,k+1,r-w[k]);
  }
}
```

Find all m-coloring of a graph





To assign a color from $1, \ldots, m$ to each vertex such that no two adjacent vertices have the same color.

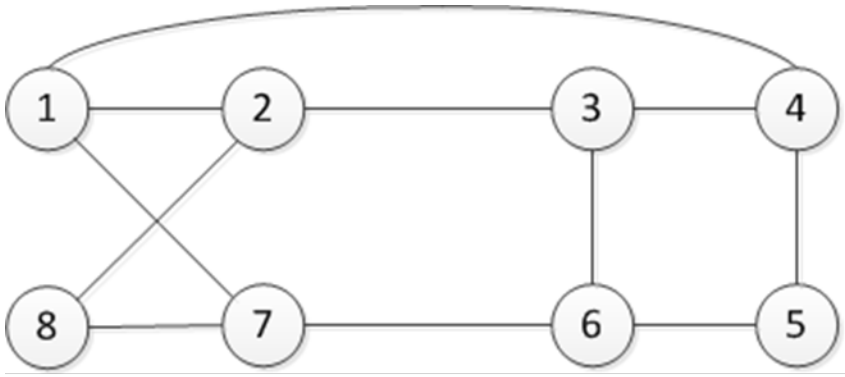
Find all m-coloring of a graph


```
void mcoloring(int k)
// k is the index of the next vertex to color
{
 do{//Generate all legal assignments for x[k]
    NextValue(k); // Assign to x[k] a legal color
    if (!x[k]) return; // no new color possible
    if (k==n) { //At most m colors have been used
                        //to color the n vertices
        for (int i=1; i<=n; i++) cout << x[i] << ' ';
        cout <<endl;
    else mcoloring(k+1);
   } while (1);
}
```

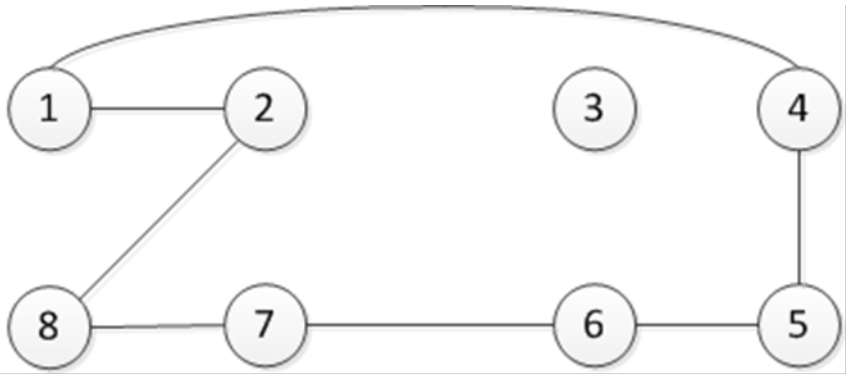
Find all m-coloring of a graph


```
void NextValue(int k)
// k is the index of the next vertex to color
// x[1] ... x[k-1] have been assigned integer values in
// the range of [1,m] such that adjacent vertices have
// distince integers. A value for x[k] is determined in the
// range [0,m]. x[k] is initially assigned to 0. If no such
// color exists, then x[k] will remain as 0
{
 do{
    x[k]=(x[k]+1) % (m+1); //find the next highest color
    if (!x[k]) return; //All colors have been used
    for (int j=1; j<=n; j++){ //check if there is conflict
          if (G[k][j] && (x[k]==x[j]))
             break;
    }
    if (j==n+1) return; //new color found
  }while (1); //Otherwise find another color;
}
```

Hamiltonian Cycles



$G = (V, E)$



Hamiltonian Cycle of G

Let $G = (V, E)$ be a connected graph with n vertices. A Hamiltonian cycle is a round trip path along n edges of G that visits every vertex once and returns to its starting point.

Find all Hamiltonian Cycles


```
void Hamiltonian(int k)
// k is the index of the next vertex to color.
// start at node 1
{
 do{//Generate all legal assignments for x[k]
    NextValue(k); // Assign legal next value to x[k]
    if (!x[k]) return;
    if (k==n) {
       for (int i=1; i<=n; i++) cout << x[i] << ' ';
        cout <<"1"<<endl;
    else Hamiltonian(k+1);
   } while (1);
}
```

NextValue for Hamiltonian Cycle

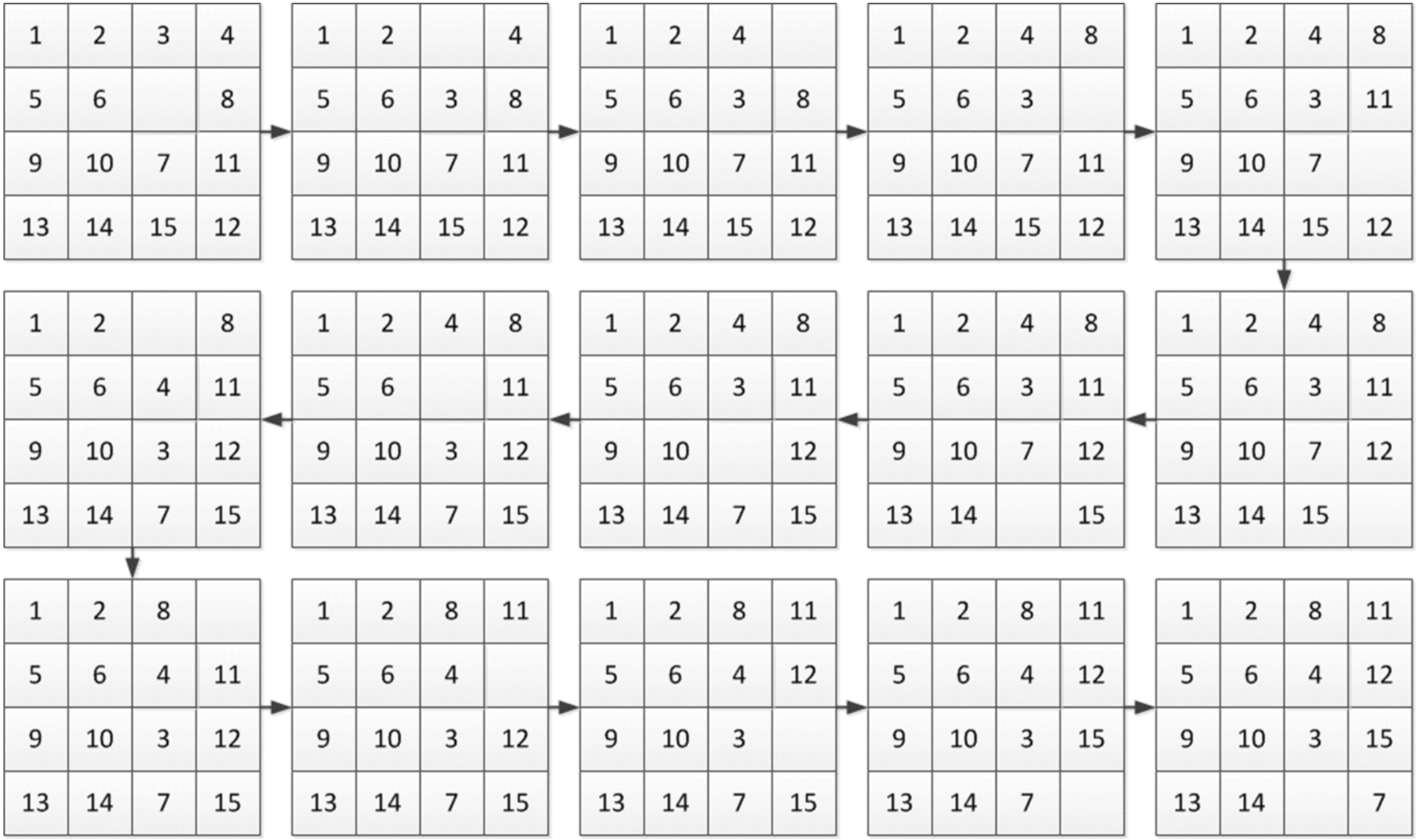

```
void NextValue(int k){
// x[1],...,x[k-1] is a path of k-1 distinct vertices. If
// x[k]==0, then no vertex has as yet been assigned to x[k].
// After execution, x[k] is assigned to the next highest
// numbered vertex which has not appeared in
// x[1],...,x[k-1] and is connected by an edge to x[k-1].
// Otherwise, x[k]==0. If k==n, then in addition x[k]
// must also be connected to x[1]
 do{
    x[k]=(x[k]+1) % (n+1); //next vertex
    if (!x[k]) return;
    if (G[x[k-1]][x[k]] {// Is there an edge?
       for (int j=1; j<=k-1; j++) if (x[j]==x[k]) break;
       if (j==k)// If true then the vertex is distinct
           if ((k<n) || ((k==n) && G[x[n]][x[1]]))
               return;
  }while(1)
}
```

15-Puzzle

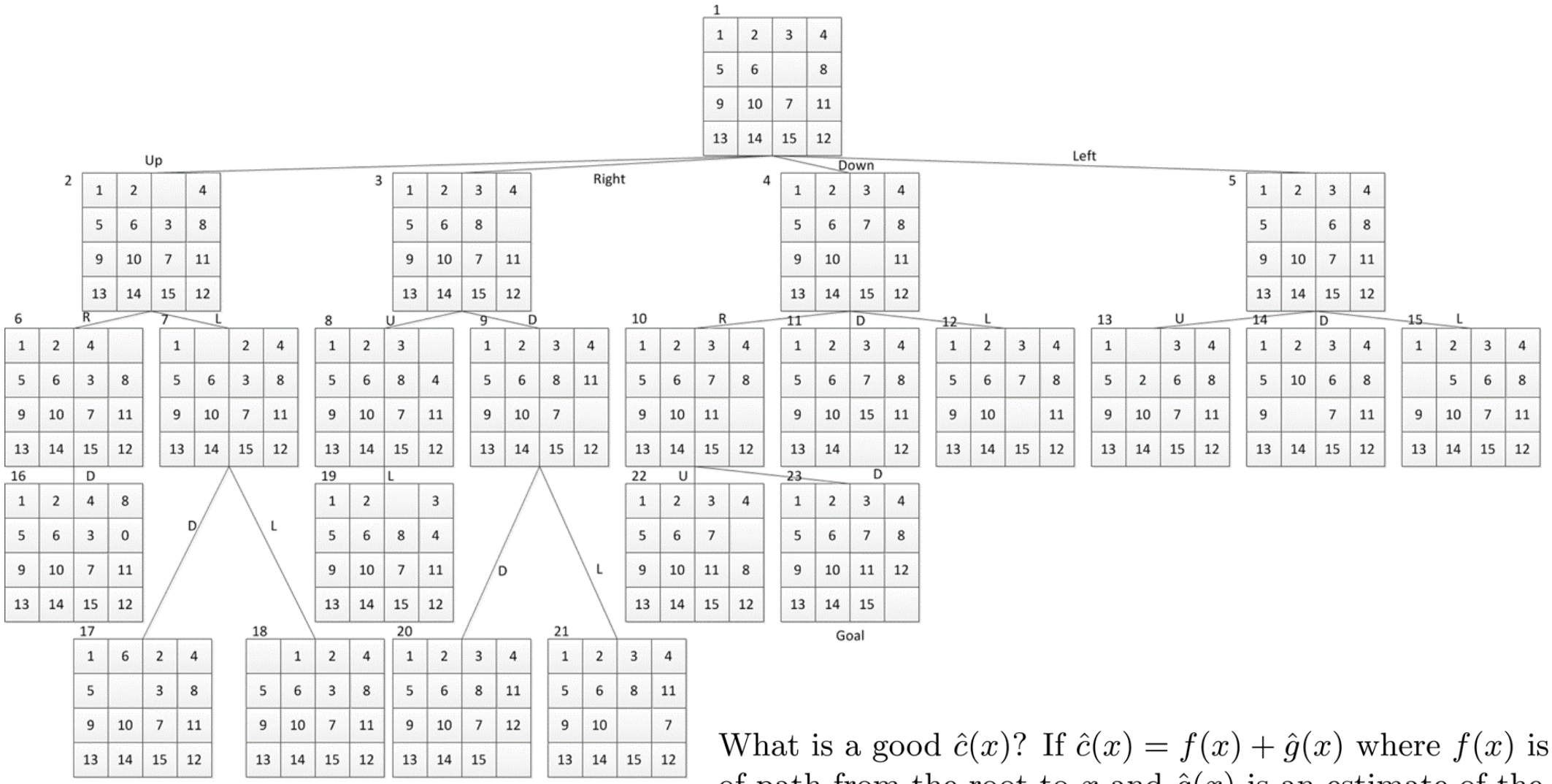
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Up Right Down Left Sequence - DFS



BFS – Least Cost Search



What is a good $\hat{c}(x)$? If $\hat{c}(x) = f(x) + \hat{g}(x)$ where $f(x)$ is the length of path from the root to x and $\hat{g}(x)$ is an estimate of the length of a shortest path from x to a goal node. One possibility is $\hat{g}(x)$ is the number of nonblank tiles not in their goal position.