بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۰۱)

طراحی الگوریتم‌ها

حسین فلسفین

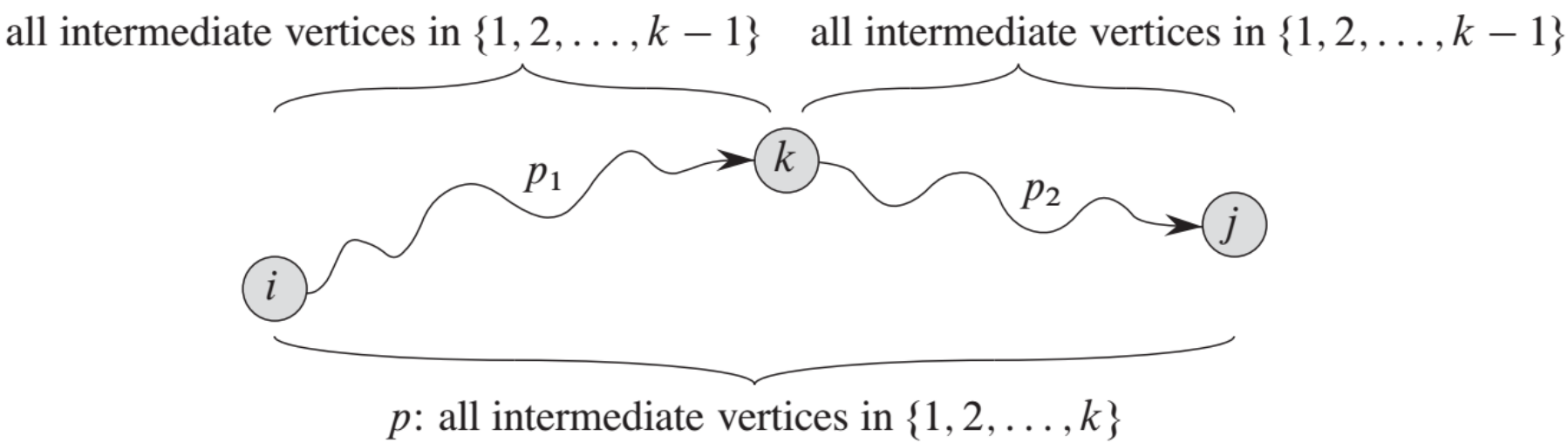
We shall use a different dynamic-programming formulation to solve the all-pairs shortest-paths problem on a weighted, directed graph $G = (V, E)$. The resulting algorithm, known as the Floyd-Warshall algorithm, runs in $\Theta(n^3)$ time. Negative-weight edges may be present, but we assume that there are no negative-weight cycles.

The Floyd-Warshall algorithm considers the intermediate vertices of a shortest path, where an intermediate vertex of a simple path $p = \langle v_1, v_2, \ldots, v_l \rangle$ is any vertex of p other than v_1 or v_l, that is, any vertex in the set $\{v_2, v_3, \ldots, v_{l-1}\}$.

Under our assumption that the vertices of G are $V = \{1, 2, \ldots, n\}$, let us consider a subset $\{1, 2, \ldots, k\}$ of vertices for some k. For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, \ldots, k\}$, and let p be a minimum-weight path from among them. (Path p is simple.) The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The relationship depends on whether or not k is an intermediate vertex of path p.

☞ *If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set $\{1, 2, \ldots, k-1\}$. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.*

☞ *If k is an intermediate vertex of path p, then we decompose p into $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$. p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k\}$. In fact, we can make a slightly stronger statement. Because vertex k is not an intermediate vertex of path p_1, all intermediate vertices of p_1 are in the set $\{1, 2, \ldots, k-1\}$. Therefore, p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Similarly, p_2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$.*


all intermediate vertices in $\{1, 2, \ldots, k-1\}$
all intermediate vertices in $\{1, 2, \ldots, k-1\}$
p_1
p_2
k
i
j
p: all intermediate vertices in $\{1, 2, \ldots, k\}$

A recursive solution to the all-pairs shortest-paths problem

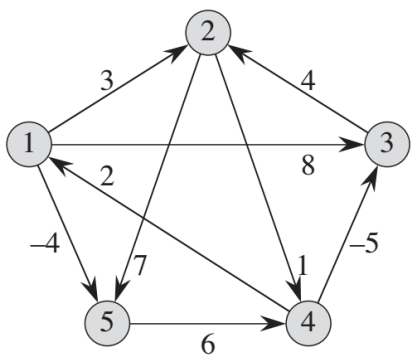
Let $d_{ij}^{(k)}$ ***be the weight of a shortest path from vertex*** i ***to vertex*** j ***for which all intermediate vertices are in the set*** $\{1, 2, \ldots, k\}$. ***When*** $k = 0$, ***a path from vertex*** i ***to vertex*** j ***with no intermediate vertex numbered higher than*** 0 ***has no intermediate vertices at all. Such a path has at most one edge, and hence*** $d_{ij}^{(0)} = w_{ij}$. ***Following the above discussion, we define*** $d_{ij}^{(k)}$ ***recursively by***

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1. \end{cases}$$

Because for any path, all intermediate vertices are in the set $\{1, 2, \ldots, n\}$***, the matrix*** $D^{(n)} = \left(d_{ij}^{(n)}\right)$ ***gives the final answer:*** $d_{ij}^{(n)} = \delta(i, j)$ ***for all*** $i, j \in V$**.**

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4       let D^(k) = (d_ij^(k)) be a new n × n matrix
5       for i = 1 to n
6            for j = 1 to n
7                 d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3-7. Because each execution of line 7 takes $O(1)$ time, the algorithm runs in time $\Theta(n^3)$.



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

We can perform our calculations using only one array D because the values in the kth row and the kth column are not changed during the kth iteration of the loop. That is, in the kth iteration the algorithm assigns

$$d_{ik}^{(k)} = \min\left(d_{ik}^{(k-1)}, d_{ik}^{(k-1)} + d_{kk}^{(k-1)}\right)$$

which clearly equals $d_{ik}^{(k-1)}$, and

$$d_{kj}^{(k)} = \min\left(d_{kj}^{(k-1)}, d_{kk}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

which clearly equals $d_{kj}^{(k-1)}$. During the kth iteration, $d_{ij}^{(k)}$ is computed from only its own value and values in the kth row and the kth column. Because these values have maintained their values from the $(k-1)$st iteration, they are the values we want.

As mentioned before, sometimes after developing a dynamic programming algorithm, it is possible to revise the algorithm to make it more efficient in terms of space.

```
void floyd (int n
            const number W[][],
            number D[][])

  {
    index i, j, k;
    D = W;
    for (k = 1; k <= n; k++)
        for (i = 1; i <= n; i++)
            for (j = 1; j <= n; j++)
                D[i][j] = minimum(D[i][j], D[i][k] + D[k][j]);
  }
```

Transitive closure of a directed graph

Given a directed graph $G = (V, E)$ with vertex set $\{1, 2, \ldots, n\}$, we might wish to determine whether G contains a path from i to j for all vertex pairs $i, j \in V$. We define the transitive closure of G as the graph $G^* = (V, E^*)$, where

$$E^* = \{(i, j) : \text{there is a path from vertex } i \text{ to vertex } j \text{ in } G\}.$$

One way to compute the transitive closure of a graph in $\Theta(n^3)$ time is to assign a weight of 1 to each edge of E and run the Floyd-Warshall algorithm. If there is a path from vertex i to vertex j, we get $d_{ij} < n$. Otherwise, we get $d_{ij} = \infty$.

There is another, similar way to compute the transitive closure of G in $\Theta(n^3)$ time that can save time and space in practice. This method substitutes the logical operations $\vee$ (logical OR) and $\wedge$ (logical AND) for the arithmetic operations $\min$ and $+$ in the Floyd-Warshall algorithm. For $i, j, k = 1, 2, \ldots, n$, we define $t_{ij}^{(k)}$ to be 1 if there exists a path in graph G from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$, and 0 otherwise. We construct the transitive closure $G^* = (V, E^*)$ by putting edge (i, j) into E^* if and only if $t_{ij}^{(n)} = 1$.

A recursive definition of $t_{ij}^{(k)}$ *is*

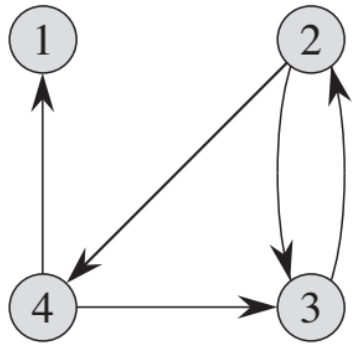
$$t_{ij}^{(0)} = \begin{cases} 0, & \text{, if } i \neq j \text{ and } (i,j) \notin E \\ 1, & \text{, if } i = j \text{ or } (i,j) \in E \end{cases}$$

and for $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$$

As in the Floyd-Warshall algorithm, we compute the matrices $T^{(k)} = (t_{ij}^{(k)})$ *in order of increasing* k.

```
TRANSITIVE-CLOSURE(G)
 1  n = |G.V|
 2  let T^(0) = (t_ij^(0)) be a new n × n matrix
 3  for i = 1 to n
 4      for j = 1 to n
 5          if i == j or (i, j) ∈ G.E
 6              t_ij^(0) = 1
 7          else t_ij^(0) = 0
 8  for k = 1 to n
 9      let T^(k) = (t_ij^(k)) be a new n × n matrix
10      for i = 1 to n
11          for j = 1 to n
12              t_ij^(k) = t_ij^(k−1) ∨ (t_ik^(k−1) ∧ t_kj^(k−1))
13  return T^(n)
```



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

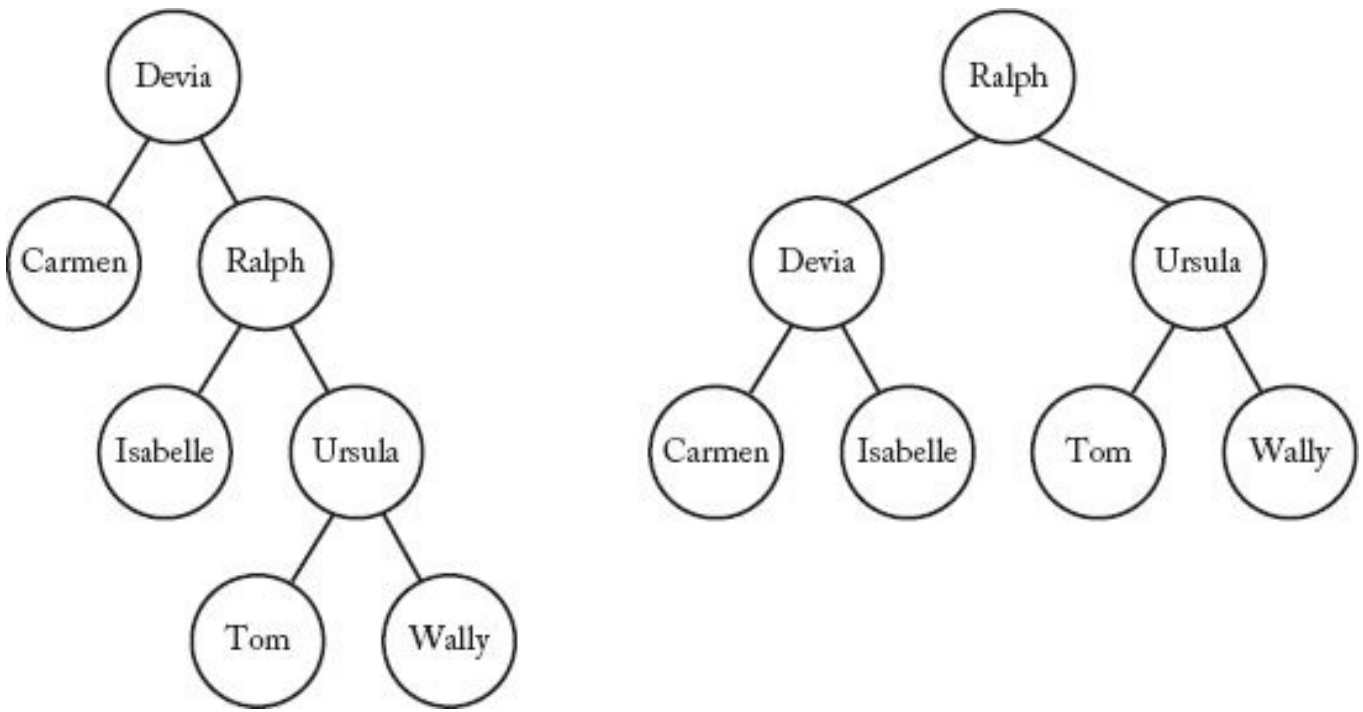
Optimal Binary Search Trees

Definition: *A binary search tree is a binary tree of items (ordinarily called keys), that come from an ordered set, such that*

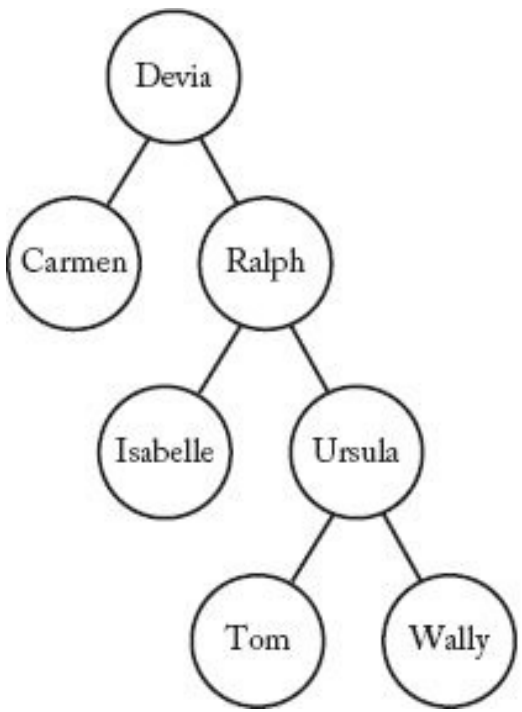
1. *Each node contains one key.*
2. *The keys in the left subtree of a given node are less than or equal to the key in that node.*
3. *The keys in the right subtree of a given node are greater than or equal to the key in that node.*

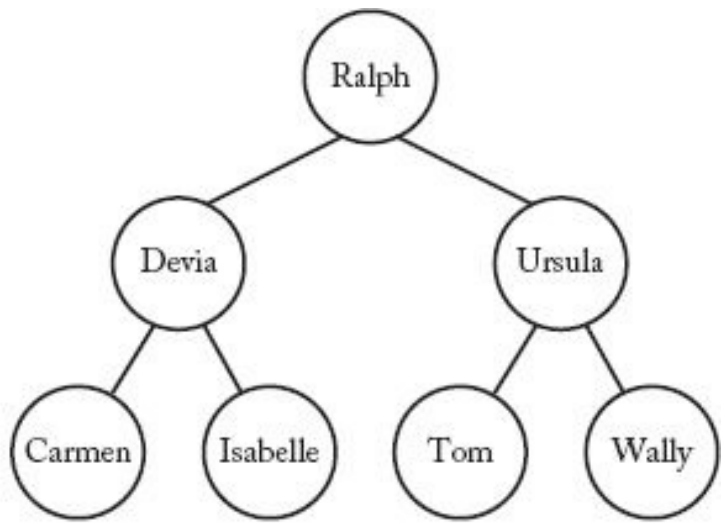
Although, in general, a key can occur more than once in a binary search tree, for simplicity we assume that the keys are ***distinct***.

The depth of a node in a tree is the number of edges in the unique path from the root to the node. This is also called the level of the node in the tree. The depth of a tree is the maximum depth of all nodes in the tree. The tree on the left has a depth of 3, whereas the tree on the right has a depth of 2.



A binary tree is called balanced if the depth of the two subtrees of every node never differ by more than 1. The tree on the left is not balanced because the left subtree of the root has a depth of 0, and the right subtree has a depth of 2. The tree on the right is balanced.





Ordinarily, a binary search tree contains records that are retrieved according to the values of the keys. Our goal is to organize the keys in a binary search tree so that the average time it takes to locate a key is minimized. A tree that is organized in this fashion is called optimal. It is not hard to see that, if all keys have the same probability of being the search key, the tree on the right is optimal (in the previous page). We are concerned with the case where the keys do not have the same probability.

Search Binary Tree

Problem: Determine the node containing a key in a binary search tree. It is assumed that the key is in the tree.

Inputs: a pointer $tree$ to a binary search tree and a key $keyin$.

Outputs: a pointer p to the node containing the key.

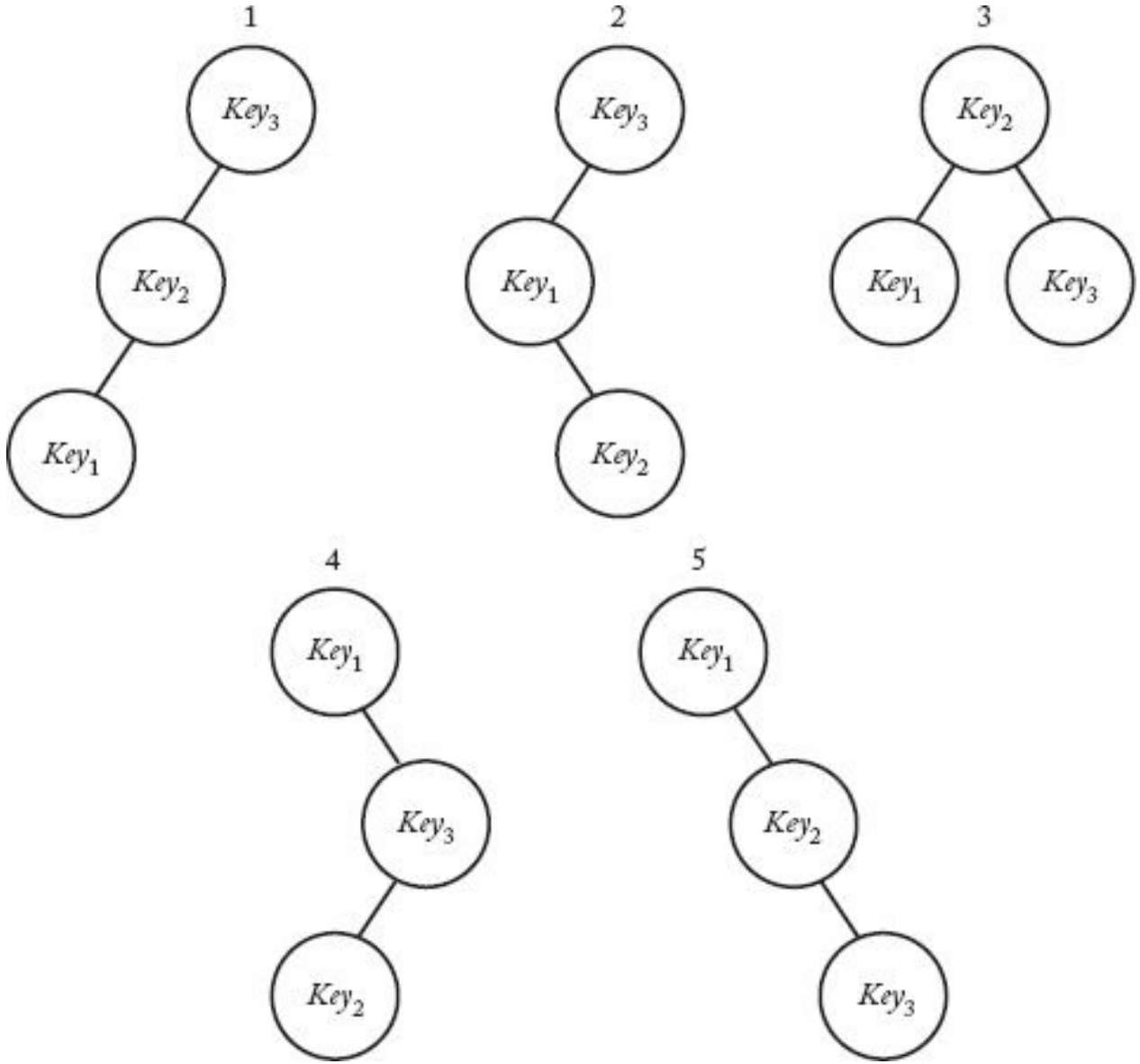
```
void search (node_pointer tree,
             keytype keyin,
             node_pointer& p)
{
  bool found;

  p = tree;
  found = false;
  while (! found)
    if(p->key == keyin)
       found = true;
    else if (keyin < p-> key);
       p = p-> left;                    // Advance to left child.
     else
       p = p-> right;                   // Advance to right child.
}
```

The number of comparisons done by procedure search to locate a key is called the search time. Our goal is to determine a tree for which the average search time is minimal. The search time for a given key is $depth(key) + 1$*, where* $depth(key)$ *is the depth of the node containing the key. For example, because the depth of the node containing "Ursula" is 2 in the left tree, the search time for "Ursula" is* $depth(\text{Ursula}) + 1 = 2 + 1 = 3$.

Let $Key_1, Key_2, \ldots, Key_n$ be the n keys in order, and let p_i be the probability that Key_i is the search key. If c_i is the number of comparisons needed to find Key_i in a given tree, the average search time for that tree is

$$\sum_{i=1}^{n} c_i p_i$$

This is the value we want to minimize.


1
Key_3
Key_2
Key_1
2
Key_3
Key_1
Key_2
3
Key_2
Key_1
Key_3
4
Key_1
Key_3
Key_2
5
Key_1
Key_2
Key_3

The above figure shows the five different trees when $n = 3$. The actual values of the keys are not important. The only requirement is that they be ordered. If

$$p_1 = 0.7, \quad p_2 = 0.2, \quad \textbf{and } p_3 = 0.1,$$

the average search times for the trees are:

1. $3(0.7) + 2(0.2) + 1(0.1) = 2.6$
2. $2(0.7) + 3(0.2) + 1(0.1) = 2.1$
3. $2(0.7) + 1(0.2) + 2(0.1) = 1.8$
4. $1(0.7) + 3(0.2) + 2(0.1) = 1.5$
5. $1(0.7) + 2(0.2) + 3(0.1) = 1.4$

The fifth tree is optimal.

In general, we cannot find an optimal binary search tree by considering all binary search trees because the number of such trees is at least exponential in n. We prove this by showing that if we just consider all binary search trees with a depth of $n-1$, we have an exponential number of trees. In a binary search tree with a depth of $n-1$, the single node at each of the $n-1$ levels beyond the root can be either to the left or to the right of its parent, which means there are two possibilities at each of those levels. This means that the number of different binary search trees with a depth of $n-1$ is 2^{n-1}.

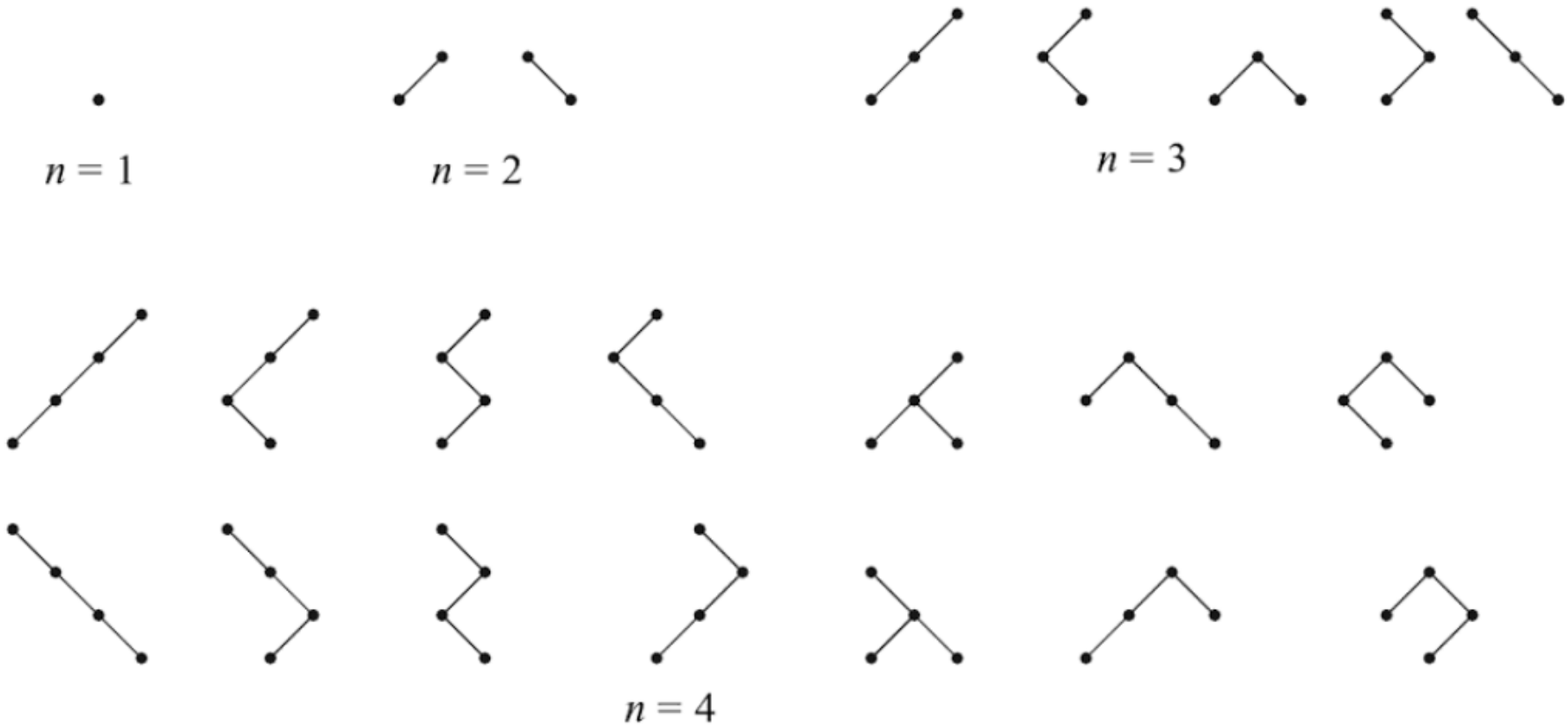
دوباره اعداد کاتالان

Let b_n denote the number of BSTs with n nodes. Since the only binary tree on one node is a root, with no children, it follows that $b_1 = 1$. For $n > 1$, a binary tree T on n vertices has a left subtree with j vertices and a right subtree with $n-1-j$ vertices, for some j between 0 and $n-1$. Pairing the b_j possible left subtrees with the b_{n-1-j} possible right subtrees results in $b_j \times b_{n-1-j}$ different combinations of left and right subtrees for T. Hence,

$$b_n = b_0 b_{n-1} + b_1 b_{n-2} + b_2 b_{n-3} + \cdots + b_{n-1} b_0.$$

This recurrence relation is known as the Catalan recursion, and the quantity b_n is called the nth Catalan number:

$$b_n = C_n = \frac{1}{n+1}\binom{2n}{n}.$$


$n = 1$
$n = 2$
$n = 3$
$n = 4$

حال که دیدیم الگوریتم ***brute-force*** بسیار ناکارآمد عمل می‌کند، سعی می‌کنیم تا از راهبرد برنامه‌ریزی پویا استفاده کنیم:

Suppose that keys Key_i ***through*** Key_j ***are arranged in a tree that minimizes***

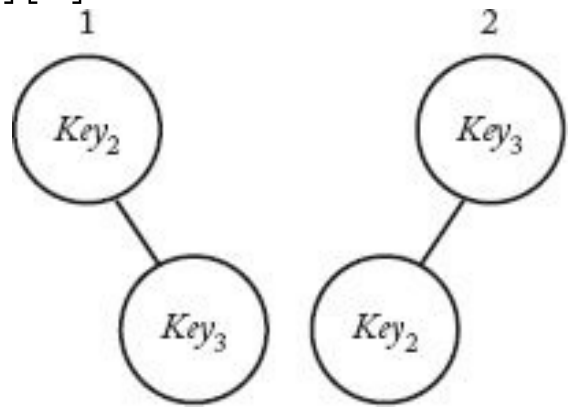
$$\sum_{m=i}^{j} c_m p_m,$$

where c_m ***is the number of comparisons needed to locate*** Key_m ***in the tree. We will call such a tree optimal for those keys and denote the optimal value by*** $A[i][j]$***. Because it takes one comparison to locate a key in a tree containing one key,*** $A[i][i] = p_i$***.***

Example: ***Suppose we have three keys. If***

$$p_1 = 0.7, \quad p_2 = 0.2, \quad p_3 = 0.1,$$

then, to determine $A[2][3]$ ***we must consider following two trees:***



For these two trees we have the following:

1. $1(p_2) + 2(p_3) = 1(0.2) + 2(0.1) = 0.4$
2. $2(p_2) + 1(p_3) = 2(0.2) + 1(0.1) = 0.5$

The first tree is optimal, and $A[2][3] = 0.4$.