# DYNAMIC PROGRAMMING

- **P**roblems like knapsack problem, shortest path can be solved by greedy method in which optimal decisions can be made one at a time.
- For many problems, it is not possible to make stepwise decision in such a manner that the sequence of decisions made is optimal.

# DP Idea

- D*ynamic Programming* is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems
- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS
- "Programming" here means "planning"
- Main idea:
    - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
- \- solve smaller instances once
    - record solutions in a table
    - extract solution to the initial instance from that table

# Contd…

- The Cause of inefficiency in divide-and-conquer
- After division …
  - Smaller instances are unrelated, e.g., *mergesort*
  - Smaller instances are related, e.g., *fibonacci*
    - repeatedly solve common instances

# Contd…

- Dynamic programming
  - bottom-up approach
  - use an array (table) to save solutions to small instances

# Fibonacci

- The same Fibonacci series algorithm in Dynamic programming is as follows:
- Dynamic programming Algorithm *n*th Fibonacci Term (Iterative)
  - Problem: Determine the *n*th term in the Fibonacci sequence.
  - Inputs: a nonnegative integer *n*.
  - Outputs : *fib*2, the *n*th term in the Fibonacci sequence.

```
int fib 2 (int n) {
    index i;
    int f[0 .. n]; // array to store Fibonacci values
    f[ 0 ] = 0;
    if (n > 0){
            f[ 1 ] = 1;
            for (i = 2; i<= n; i++)
                    f[ i ] = f[i - 1] + f [i -2 ]; }
    return f[ n ];
    }
```

# Binomial coefficent

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \qquad for \quad 0 \le k \le n$$

# The binomial coefficient

- Definition

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \qquad for \quad 0 \le k \le n$$

- Recursive definition

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 \quad k = 0 \quad or \quad k = n \end{cases}$$

# The algorithm

- Algorithm 3.1: Binomial Coefficient Using Divide-and-Conquer
  - Problem: Compute the binomial coefficient.
  - Inputs: nonnegative integers $n$ and $k$, where $k \le n$.
  - Outputs: *bin*, the binomial coefficient .

**int** *bin* (**int** *n*, **int** *k*) {

    **if** ( $k == 0 \| n == k$)

        **return 1;**

    **else**

        **return** *bin* (*n*-1, *k* - 1)+*bin* (*n* - 1, *k*);

    }

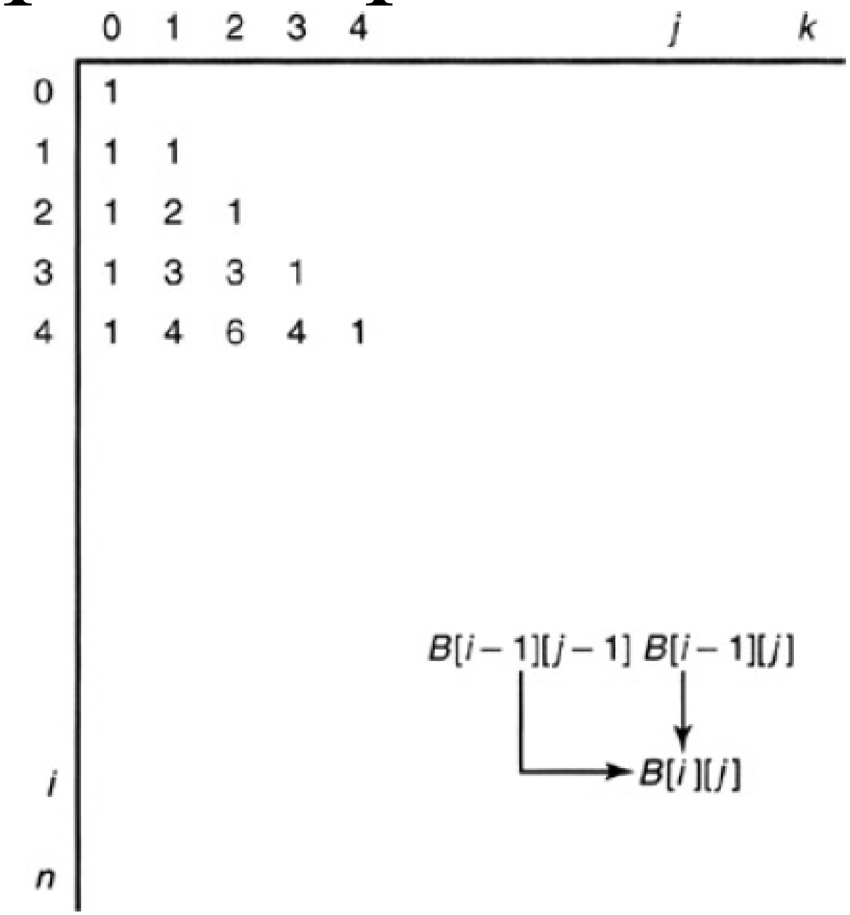
# Using dynamic programming

- Using an array *B* to store coefficients
- Steps:
  - Establish a recursive property:

$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & 0 < j < i \\ 1 \quad j = 0 \quad or \quad j = i \end{cases}$$

  - Solve an instance of the problem in a bottom-up fashion by computing the rows in B in sequence starting with the first row

# Compute sequence of rows




- Let's compute $B[4][2]$

# The algorithm

- Algorithm 3.2: Binomial Coefficient Using Dynamic Programming
  - Problem: Compute the binomial coefficient.
  - Inputs: nonnegative integers *n* and *k*, where $k \leq n$.
  - Outputs: *bin* 2, the binomial coefficient

```
int bin2 (int n, int k) {
      index i, j;
      int B[0..n] [0..k];
      for (i = 0; i <= n; i++)
            for (j = 0; j <= minimum (i, k); j++)
                  if (j = = 0 || j = = i)
                        B[i][ j] = 1;
                  else
                        B[i][j] = B[i - 1][j - 1] + B[i - 1] [j];
      return B[n][k];
}
```

# DYNAMIC PROGRAMMING (Contd..)

***Example***:

- Suppose a shortest path from vertex i to vertex j is to be found.
- Let Ai be vertices adjacent from vertex i. which of the vertices of Ai should be the second vertex on the path?
- One way to solve the problem is to enumerate all decision sequences and pick out the best.
- In dynamic programming the principle of optimality is used to reduce the decision sequences.

# DYNAMIC PROGRAMMING (Contd..)

***Principle of optimality***:

- An optimal sequence of decisions has the property that whatever the initial state and decisions are, the remaining decisions must constitute an optional decision sequence with regard to the state resulting from the first decision.
- In the greedy method only one decision sequence is generated.
- In dynamic programming many decision sequences may be generated.

# Contd…

- An optimal solution to an instance of a problem always contains optimal solutions to all subinstances
- ensures that an optimal solution to an instance can be obtained by combining optimal solutions to subinstances
- It is necessary to show that the principle applies before using dynamic programming to obtain the solution

# DYNAMIC PROGRAMMING (Contd..)

***Example***:

- [0/1 knapsack problem]:the xi’s in 0/1 knapsack problem is restricted to either 0 or 1.
- Using KNAP(l,j,y) to represent the problem

$$\text{Maximize} \sum_{l \le i \le j} p_i x_i$$

$$\text{subject to} \sum_{l \le i \le j} w_i x_i \le y, \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots(1)$$

$$x_i = 0 \text{ or } 1 \quad l \le i \le j$$

The 0/1 knapsack problem is KNAP(l,n,M).

# DYNAMIC PROGRAMMING (Contd..)

- Let $y_1, y_2, \ldots\ldots, y_n$ be an optimal sequence of 0/l values for $x_1, x_2 \ldots\ldots, x_n$ respectively.
- If $y_1 = 0$ then $y_2, y_3, \ldots\ldots, y_n$ must constitute an optimal sequence for KNAP (2,n,M).
- If it does not, then $y_1, y_2, \ldots\ldots, y_n$ is not an optimal sequence for KNAP (1, n, M). If y1=1, then $y_1, y_2, \ldots\ldots, y_n$ is an optimal sequence for KNAP(2,n,M-wi ).
- If it is not there is another 0/l sequence $z_1, z_2, \ldots, z_n$ such that ∑pizi has greater value.
- Thus the principal of optimality holds.

# DYNAMIC PROGRAMMING (Contd..)

- Let $g_j(y)$ be the value of an optimal solution to KNAP (j+1,n,y).
- Then $g_0(M)$ is the value of an optimal solution to KNAP(l,n,M).
- From the principal of optimality

  $g_0(M) = \max\{g_1(M), g_1(M-W_1) + P_1)\}$

- $g_0(M)$ is the maximum profit which is the value of the optimal solution .

# DYNAMIC PROGRAMMING (Contd..)

- The principal of optimality can be equally applied to intermediate states and decisions as has been applied to initial states and decisions.
- Let $y_1,y_2,\ldots..y_n$ be an optimal solution to KNAP(l,n,M).
- Then for each j $l \le j \le n$ , $y_i,..,y_j$ and $y_{j+1},\ldots.,y_n$ must be optimal solutions to the problems $KNAP(1,j,\sum_{l \le i \le j} w_i y_i)$

  and $KNAP(j+1,n,M-\sum_{l \le i \le j} wiyi)$ respectively.

# DYNAMIC PROGRAMMING (Contd..)

- Then $g_i(y) = \max\{g_{i+1}(y)$ (xi +1= 0 case),
  $g_{i+1}(y-w_{i+1}) + p_{i+1}\}$…(1) (xi +1= 1case),
- Equation (1) is known as recurrence relation.
- Dynamic programming algorithms solve the relation to obtain a solution to the given problems.
- To solve 0/1 knapsack problem , we use the knowledge $g_n(y) = 0$ for all y, because $g_n(y)$ is on optimal solution (profit) to the problem

KNAP(n+1, n, y) which is obviously zero for any y.

# DYNAMIC PROGRAMMING (Contd..)

- Substituting $i = n - 1$ and using $g_n(y) = 0$ in the above relation(1), we obtain $g_{n-1}(y)$.
- Then using $g_{n-1}(y)$, we obtain $g_{n-2}(y)$ and so on till we get $g_0(M)$ (with $i=0$) which is the solution of the knapsack problem.
- There are two approaches to solving the recurrence relation 1
- (1) Forward approach and (2) Backward approach

# DYNAMIC PROGRAMMING (Contd..)

- In the forward approach ,decision $\underline{x_i \text{ is made in terms}}$ of optimal decision Sequences for $x_{i+1}$........$x_n$ (i.e we look ahead).
- In the backward approach, decision $\underline{x_i \text{ is in terms}}$ of optimal decision sequences for $x_1$......$x_{i-1}$(i.e we look backward).

# DYNAMIC PROGRAMMING (Contd..)

- For the 0/l knapsack problems

  $G_i(y)=\max\{g_{i+1}(y), g_{i+1}(y-w_{i+1})+P_{i+1}\}$ ………(1)
- Is the forward approach as $g_{n-1}(y)$ is obtained using $g_n(y)$.
- $f_i(y) = \max\{f_{j-1}(y), f_{j-1}(y-w_i) + p_j\}$ …………..(2)
- is the backward approach, $f_j(y)$ is the value of optimal solution to Knap(i,j,Y). (2) may be solved by beginning with

  $f_i(y) = 0$ for all $y \geq 0$ and $f_i(y) =$ -infinity for $y < 0$.

# Example

- Consider 0/1 knapsack problem which has 3 objects n=3, their weights are w1=2, w2=3, w3=4, their profits are p1=1, p2=2, p3=5 and knapsack capacity m=6. compute $g_0(6)$.

# solution

$g_0(6) = \max\{g_1(6), g_1(6-W_1) + P_1)\}$
$= \max\{g_1(6), g_1(6-2) + 1)\}$
$g_0(6) = \max\{g_1(6), g_1(4) + 1)\}$

$g_1(6) = \max\{g_2(6), g_2(6-W_2) + P_2)\}$
$g_1(6) = \max\{g_2(6), g_2(3) + 2)\}$

$g_2(6) = \max\{g_3(6), g_3(6-W_3) + P_3)\}$
$g_2(6) = \max\{0, g_3(2) + 5)\} = \max\{0, 5)\} = 5.$

$g_2(3) = \max\{g_3(3), g_3(3-W_3) + P_3)\}$
$= \max\{0, g_3(3-4) + 5)\} = \max\{0, -\text{infinity})\} = 0$

# Contd…

$g_1(4) = \max\{g_2(4), g_2(4\text{-}W_2) + P_2)\}$
$= \max\{g_2(4), g_2(4\text{-}3) + 2)\}$
$= \max\{g_2(4), g_2(1) + 2)\}$

$g_2(4) = \max\{g3(4), g_3(4\text{-}4) + 5)\} = \max\{0, 5)\} = 5$
$g_1(4) = \max\{5, g_2(1) + 2)\}$

# Contd…

$g_2(1) = \max\{g_3(1), g_3(1-W_3) + P_3)\}$

$g_2(6) = \max\{0, g_3(1-4) + 5)\} = \max\{0, -\text{infinity} + 5)\} = 0.$

$G1(4) = \max\{5, 0+2\} = 5.$

$G0(6) = \max\{ 5, 5 +1\} = 6.$

# FEATURES OF DYNAMIC PROGRAMMING SOLUTIONS

- It is easier to obtain the recurrence relation using backward approach.
- **Dynamic programming** algorithms often have polynomial complexity.
- Optimal solution to sub problems are retained so as to avoid recomputing their values.

# OPTIMAL BINARY SEARCH TREES

- Definition: **binary search tree (BST)** A binary search tree is a binary tree; either it is empty or each node contains an identifier and

(i) all identifiers in the left sub tree of T are less than the identifiers in the root node T.

(ii) all the identifiers the right sub tree are greater than the identifier in the root node T.

(iii) the right and left sub tree are also BSTs.

# Optimal Binary Search Trees

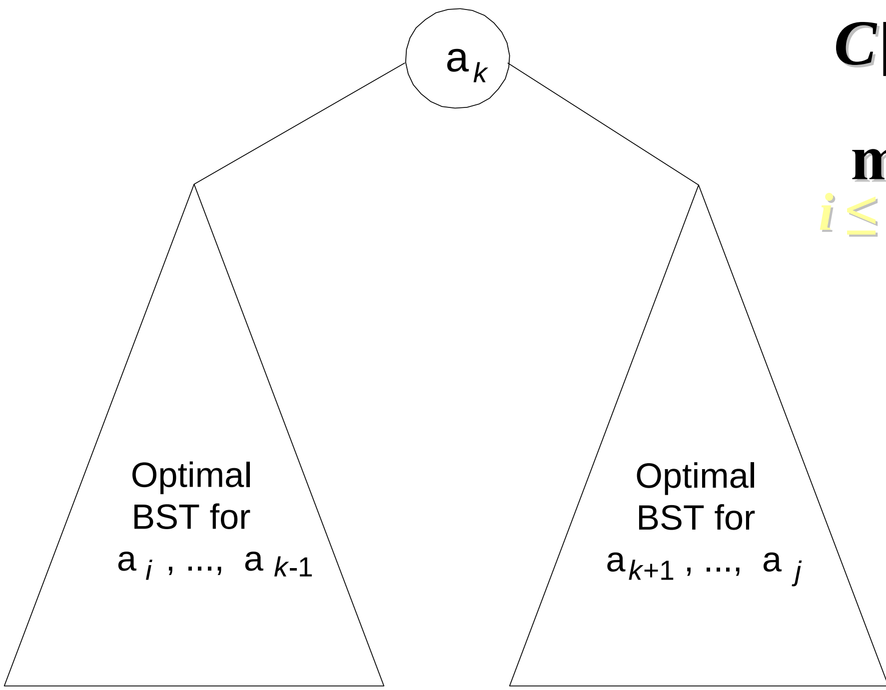
**Problem: Given $n$ keys $a_1 < \ldots < a_n$ and probabilities $p_1 \leq \ldots \leq p_n$ searching for them, find a BST with a minimum average number of comparisons in successful search.**

**Since total number of BSTs with $n$ nodes is given by $C(2n,n)/(n+1)$, which grows exponentially, brute force is hopeless.**

**Example: What is an optimal BST for keys *A*, *B*, *C*, and *D* with search probabilities 0.1, 0.2, 0.4, and 0.3, respectively?**

# DP for Optimal BST Problem

**Let $C[i,j]$ be minimum average number of comparisons made in T[$i,j$], optimal BST for keys $a_i < \ldots < a_j$ , where $1 \le i \le j \le n$. Consider optimal BST among all BSTs with some $a_k$ ($i \le k \le j$ ) as their root; T[$i,j$] is the best among them.**




$$C[i,j] = \min_{i \le k \le j} \{p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \text{ (level } a_s \text{ in T}[i,k\text{-}1] +1) + \sum_{s=k+1}^{j} p_s \text{ (level } a_s \text{ in T}[k+1,j] +1)\}$$

Example: key $\quad A \quad B \quad C \quad D$

probability $\quad 0.1 \quad 0.2 \quad 0.4 \quad 0.3$

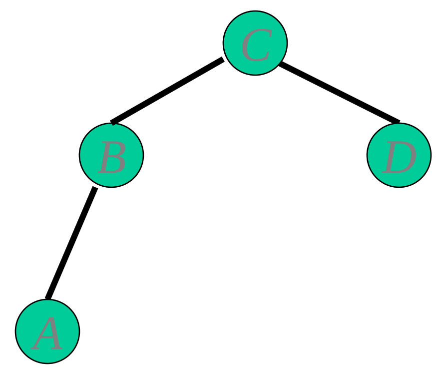
**The tables below are filled diagonal by diagonal: the left one is filled using the recurrence**

$$\boldsymbol{C[i,j] = \min_{i \le k \le j} \{C[i,k\text{-}1] + C[k+1,j]\} + \sum_{s=i}^{j} p_s, \quad C[i,i] = p_i\,;}$$

**the right one, for trees' roots, records *k*'s values giving the minima**

| $i$ \ $j$ | | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 1 | 0 | .1 | .4 | 1. | 1. |
| 2 | 0 | 0 | .2 | 1 | 7 1. |
| 3 | | | 0 | .8 | 4 1. |
| 4 | | | | .4 0 | 0 .3 |
| 5 | | | | | 0 |

| $i$ \ $j$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 1 | | 1 | 2 | 3 | 3 |
| 2 | | | 2 | 3 | 3 |
| 3 | | | | 3 | 3 |
| 4 | | | | | 4 |
| 5 | | | | | |




**optimal BST**

# Optimal Binary Search Trees

**ALGORITHM** *OptimalBST*($P[1..n]$)

```
//Finds an optimal binary search tree by dynamic programming
//Input: An array P[1..n] of search probabilities for a sorted list of n keys
//Output: Average number of comparisons in successful searches in the
//          optimal BST and table R of subtrees' roots in the optimal BST
for i ← 1 to n do
    C[i, i − 1] ← 0
    C[i, i] ← P[i]
    R[i, i] ← i
C[n + 1, n] ← 0
for d ← 1 to n − 1 do //diagonal count
    for i ← 1 to n − d do
        j ← i + d
        minval ← ∞
        for k ← i to j do
            if C[i, k − 1] + C[k + 1, j] < minval
                minval ← C[i, k − 1] + C[k + 1, j]; kmin ← k
        R[i, j] ← kmin
        sum ← P[i]; for s ← i + 1 to j do sum ← sum + P[s]
        C[i, j] ← minval + sum
return C[1, n], R
```

# Analysis DP for Optimal BST Problem

**Time efficiency: $\Theta(n^3)$ but can be reduced to $\Theta(n^2)$ by taking advantage of monotonicity of entries in the root table, i.e., $R[i,j]$ is always in the range between $R[i,j-1]$ and $R[i+1,j]$**

**Space efficiency: $\Theta(n^2)$**

**Method can be expended to include unsuccessful searches**

# ALGORITHM TO SEARCH FOR AN IDENTIFIER IN THE TREE 'T'.

Procedure SEARCH (T X I)

// Search T for X, each node had fields LCHILD, IDENT, RCHILD//

// Return address i pointing to the identifier X// //Initially T is pointing to tree.

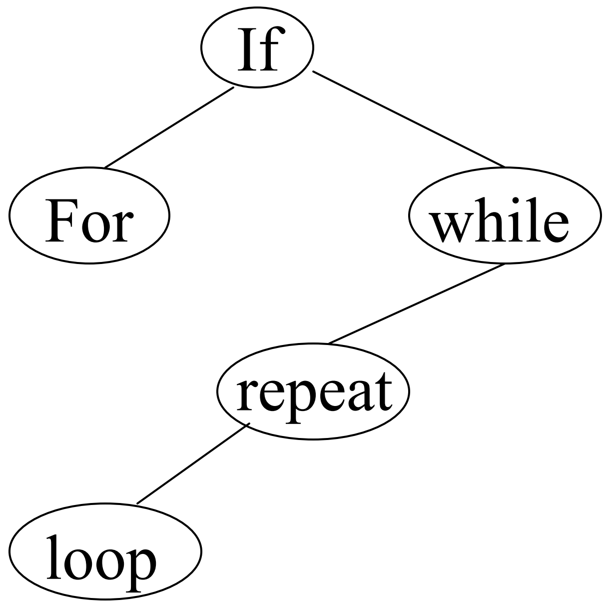
//ident(i)=X or i=0 //

i $\leftarrow$ T

## Algorithm to search for an identifier in the tree 'T'(Contd..)

```
While i ≠ 0 do
   case : X < Ident(i) : i ←LCHILD(i)
            : X = IDENT(i) : RETURN i
            : X > IDENT(i) : I ← RCHILD(i)
   endcase
repeat
end SEARCH
```

# Optimal Binary Search trees - Example




if each identifier is searched with equal probability the average number of comparisons for the above tree are $\frac{1+2+2+3+4}{5} = 12/5$.

# OPTIMAL BINARY SEARCH TREES (Contd..)

- Let us assume that the given set of identifiers are $\{a_1, a_2, \ldots\ldots.. a_n\}$ with $a_1 < a_2 < \ldots\ldots. < a_n$.
- Let $P_i$ be the probability with which we are searching for $a_i$.
- Let $Q_i$ be the probability that identifier x being searched for is such that $a_i < x < a_{i+1}$ $0 \le i \le n$, and $a_0 = -\infty$ and $a_{n+1} = +\infty$.

# OPTIMAL BINARY SEARCH TREES (Contd..)

- Then $\sum Q_i$ is the probability of an unsuccessful search.

  $0 \leq i \leq n$

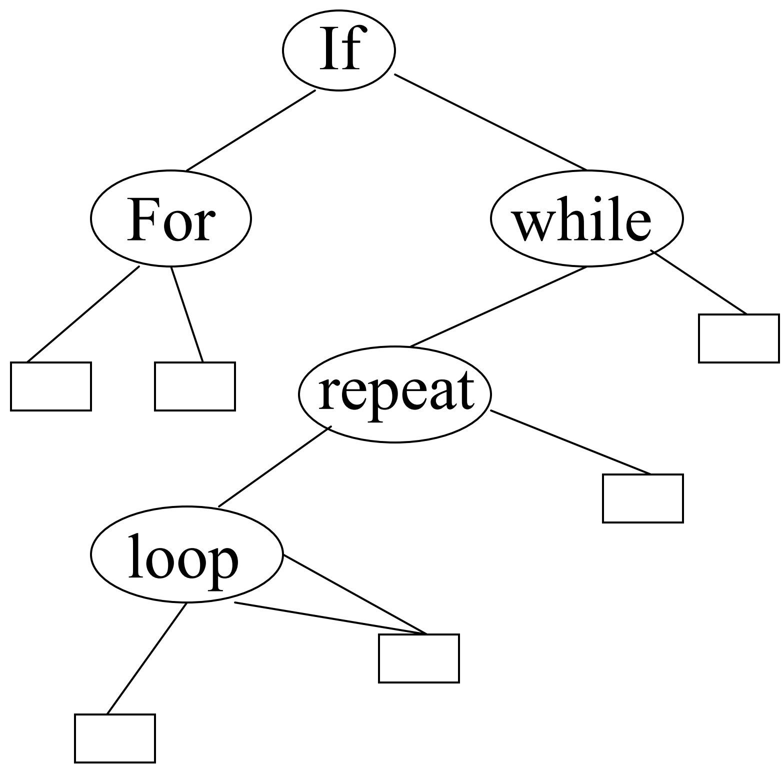
  $\sum P(i) + \sum Q(i) = 1$. Given the data,

  $1 \leq i \leq n \quad 0 \leq i \leq n$

let us construct one optimal binary search tree for ($a_1$……….$a_n$).

- In place of empty sub tree, we add external nodes denoted with squares.
- Internal nodes are denoted as circles.

# OPTIMAL BINARY SEARCH TREES (Contd..)



# Construction of optimal binary search trees

- A BST with n identifiers will have n internal nodes and n+1 external nodes.
- Successful search terminates at internal nodes unsuccessful search terminates at external nodes.
- If a successful search terminates at an internal node at level L, then L iterations of the loop in the algorithm are needed.
- Hence the expected cost contribution from the internal nodes for $a_i$ is P(i) * level($a_i$).

# OPTIMAL BINARY SEARCH TREES (Contd..)

- Unsuccessful searche terminates at external nodes i.e. at i = 0.
- The identifiers not in the binary search tree may be partitioned into n+1 equivalent classes

  $E_i \quad 0 \leq i \leq n$.

  $E_o$ contains all X such that $X \leq a_i$

  $E_i$ contains all X such that $a < X <= a_{i+1} \quad 1 \leq i \leq n$

  $E_n$ contains all X such that $X > a_n$

# OPTIMAL BINARY SEARCH TREES (Contd..)

- For identifiers in the same class $E_i$, the search terminate at the same external node.
- If the failure node for $E_i$ is at level L, then only L-1 iterations of the while loop are made

  $\therefore$ The cost contribution of the failure node for $E_i$ is Q(i) * level ($E_i$ )   -1)

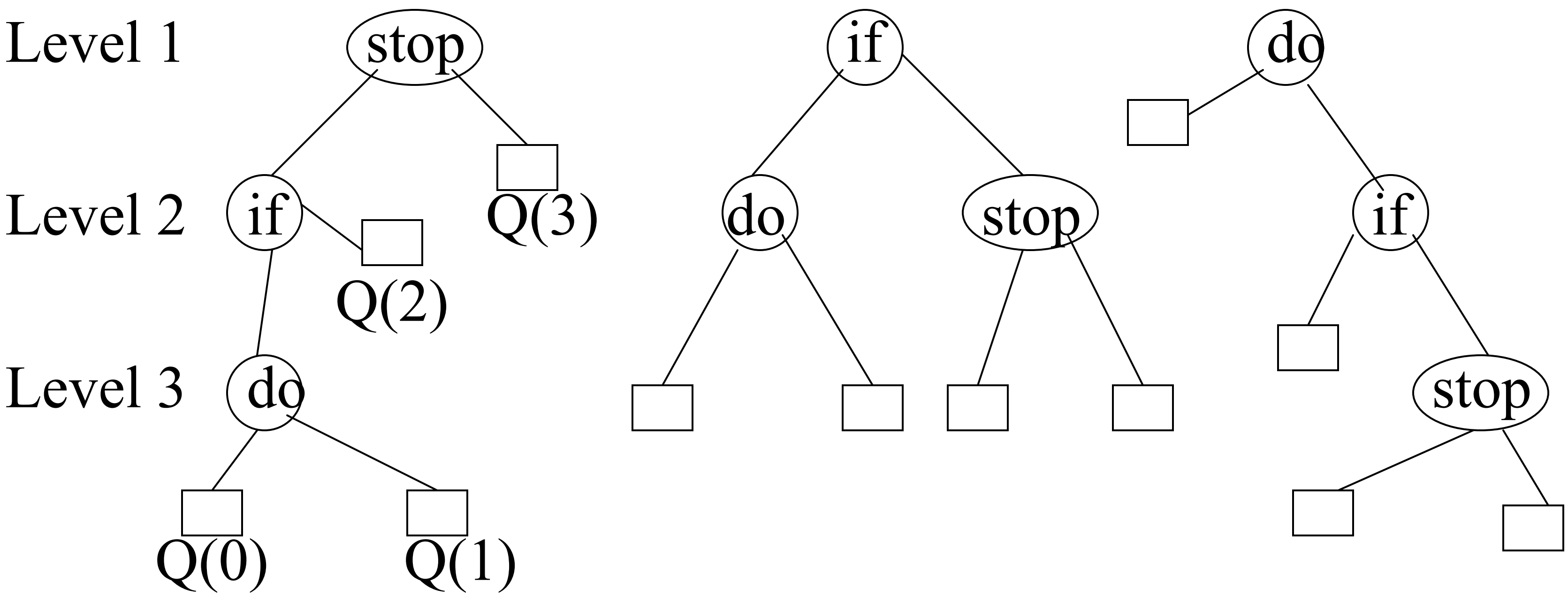
# OPTIMAL BINARY SEARCH TREES (Contd..)

- Thus the expected cost of a binary search tree is:

  $$\sum_{1 \le i \le n} P(i) * \text{level}(a_i) + \sum_{0 \le i \le n} Q(i) * \text{level}(E_i) - 1) \quad \ldots\ldots(2)$$

- An optimal binary search tree for $\{a_1 \ldots\ldots, a_n\}$ is a BST for which (2) is minimum .
- Example: Let $\{a_1, a_2, a_3\} = \{\text{do, if, stop}\}$

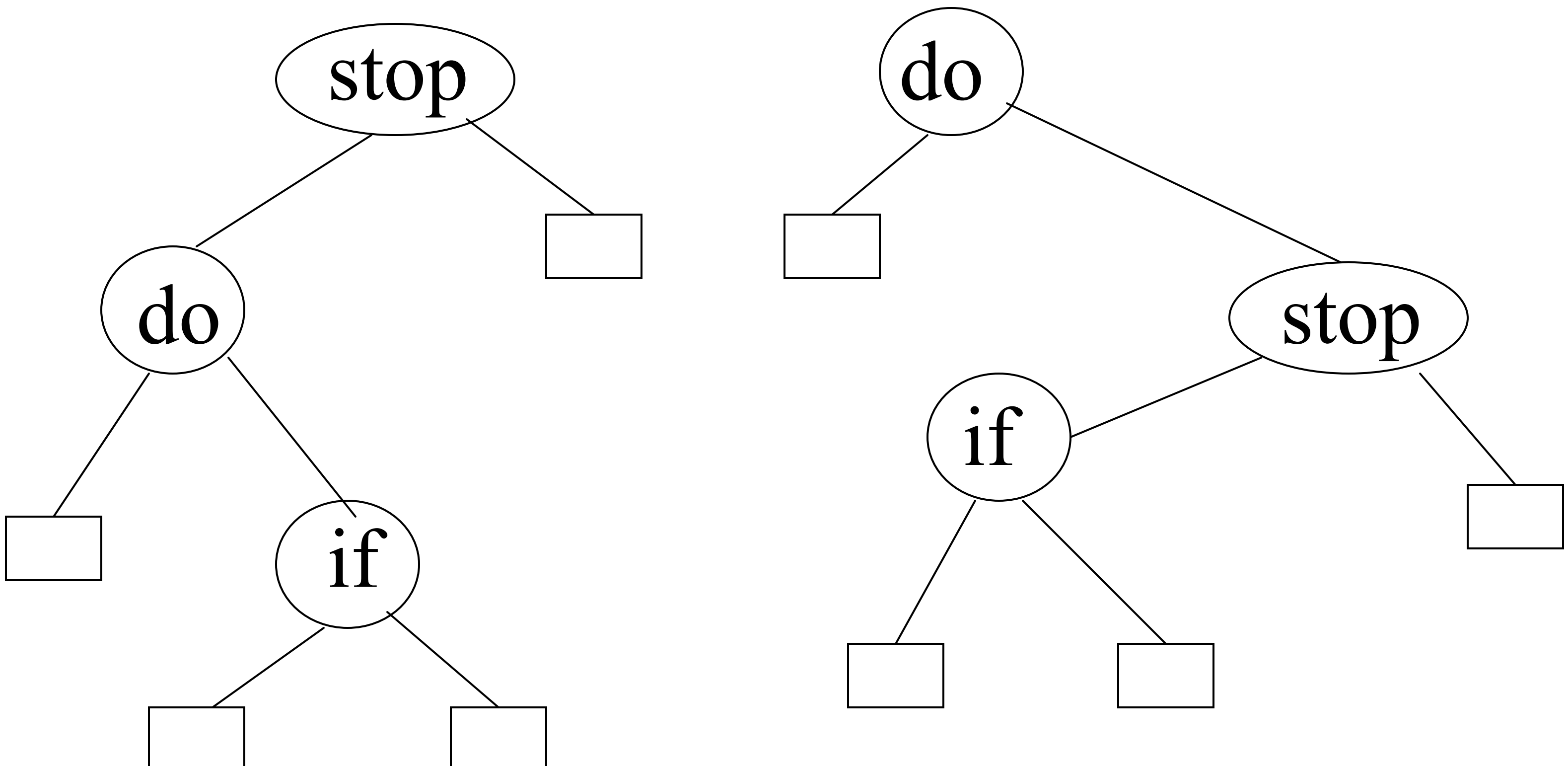
# OPTIMAL BINARY SEARCH TREES (Contd..)




(a) (b) (c)

{a1,a2,a3}={do,if,stop}

# OPTIMAL BINARY SEARCH TREES (Contd..)




(d) (c)

# OPTIMAL BINARY SEARCH TREES (Contd..)

- With equal probability P(i)=Q(i)=1/7.
- Let us find an OBST out of these.
- Cost(tree a)=$\sum_{1 \le i \le n}$ P(i)*level a(i) +$\sum_{0 \le i \le n}$ Q(i)*level (Ei) -1

$$(2\text{-}1)\quad(3\text{-}1)\ (4\text{-}1)\ (4\text{-}1)$$

$$=1/7[1+2+3 + 1 + 2 + 3 + 3] = 15/7$$

- Cost (tree b) =17[1+2+2+2+2+2+2] =13/7
- Cost (tree c) =cost (tree d) =cost (tree e) =15/7

∴ tree b is optimal.

# OPTIMAL BINARY SEARCH TREES (Contd..)

- If P(1) =0.5 ,P(2) =0.1, P(3) =0.05 , Q(0) =.15 , Q(1) =.1, Q(2) =.05 and Q(3) =.05 find the OBST.
- Cost (tree a) = .5 x 3 +.1 x 2 +.05 x 3 +.15x3 +.1x3 +.05x2 +.05x1 = 2.65
- Cost (tree b) =1.9 , Cost (tree c) =1.5 ,Cost (tree d) =2.05 ,
- Cost (tree e) =1.6 **Hence tree C is optimal**.

# OPTIMAL BINARY SEARCH TREES (Contd..)

- To obtain a OBST using Dynamic programming we need to take a sequence of decisions regard. The construction of tree.
- First decision is which of $a_i$ is be as root.
- Let us choose $a_k$ as the root . Then the internal nodes for $a_1,\ldots\ldots,a_{k-1}$ and the external nodes for classes $E_o,E_1,\ldots\ldots,E_{k-1}$ will lie in the left subtree L of the root.
- The remaining nodes will be in the right subtree R.

# OPTIMAL BINARY SEARCH TREES (Contd..)

**Define**

$$\text{Cost}(L) = \sum_{1 \le i \le k} P(i) * \text{level}(ai) + \sum_{0 \le i \le k} Q(i) * (\text{level}(E_i) - 1)$$

$$\text{Cost}(R) = \sum_{k \le i \le n} P(i) * \text{level}(ai) + \sum_{k \le i \le n} Q(i) * (\text{level}(E_i) - 1)$$

- Tij be the tree with nodes $a_{i+1}, \ldots, a_j$ and nodes corresponding to $E_i, E_{i+1}, \ldots, E_{j.}$
- Let W(i,j) represents the weight of tree $T_{ij.}$

# OPTIMAL BINARY SEARCH TREES (Contd..)

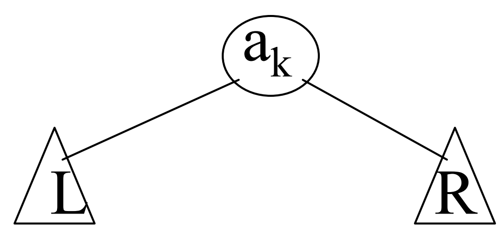
$$W(i,j)=P(i+1)+\ldots+P(j)+Q(i)+Q(i+1)\ldots Q(j)=Q(i)+\sum_{l=i+1}^{j}[Q(l)+P(l)]$$

- The expected cost of the search tree in (a) is (let us call it T) is

  $P(k)+cost(l)+cost(r)+W(0,k-1)+W(k,n)$

  $W(0,k-1)$ is the sum of probabilities corresponding to nodes and nodes belonging to equivalent classes to the left of $a_k$.

  $W(k,n)$ is the sum of the probabilities corresponding to those on the right of $a_k$.




(a) OBST with root $a_k$