# Ch21.Backtracking

SNU Internet DataBase Lab.

# Bird's-Eye View

- A surefire way to solve a problem is to make a list of all candidate answers and check them
  - If the problem size is big, we can not get the answer in reasonable time using this approach
  - List all possible cases? ➔ exponential cases

- By a systematic examination of the candidate list, we can find the answer without examining every candidate answer
  - *Backtracking* and *Branch and Bound* are most popular systematic algorithms

** Surefire = 확실한, 틀림없는

SNU Internet DataBase Lab.

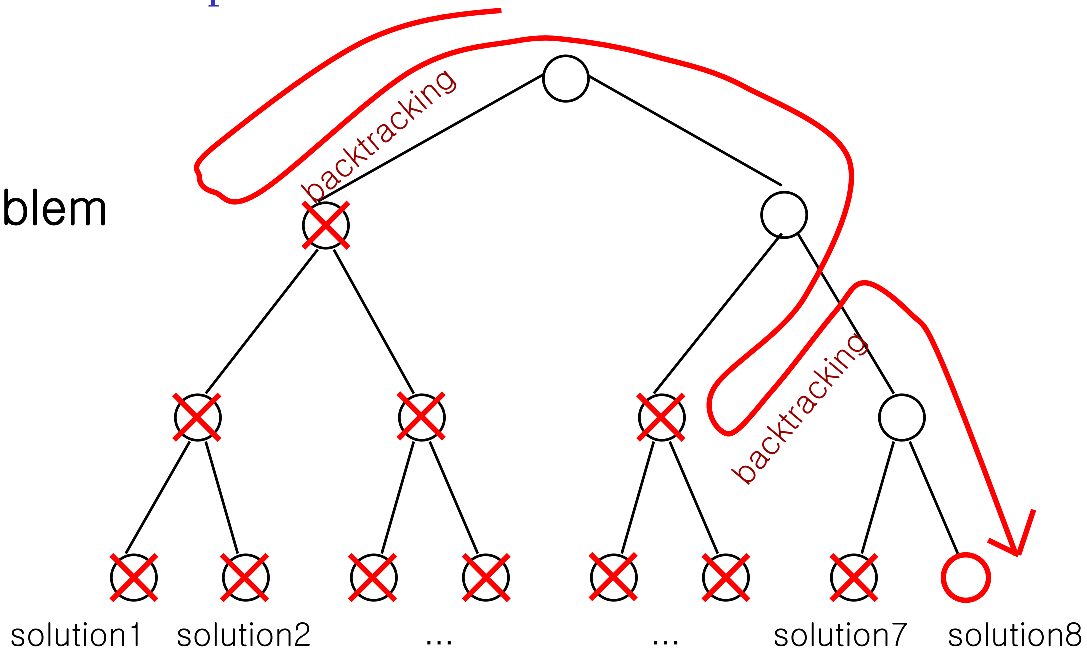
# Table of Contents

- The Backtracking Method
- Application
  - Rat in a Maze
  - Container Loading

# Backtracking

- A systematic way to search for the solution to a problem
- No need to check all possible choices → Better than the brute-force approach
- Three steps of backtracking:
  - Define a solution space
  - Construct a graph or a tree representing the solution space
  - Search the graph or the tree in a depth-first manner to find a solution

Tree representation of a problem



# Backtracking Steps (1 & 2)

- Step 1: Define a solution space
  - Solution space is a space of possible choices including at least one solution
    - In the case of the rat-in-a-maze problem, the solution space consists of all paths from the entrance to the exit
    - In the case of chess, the solution space consists of all possible locations of checkers

- Step 2: Construct a graph or a tree representing the solution space
  - Solution space can be represented either by a tree or by a graph, depending on the characteristic of the problem
    - In the case of the rat-in-a-maze problem, the solution space can be represented by a graph
    - The solution space for container loading is a tree

# Backtracking Step (3)

- Step 3: Search the graph or the tree in a depth-first manner to find a solution
  - Two nodes
    - a live node (node from which we can reach to the solution)
    - an E-node (node representing the current state)

  - We start from the start node (node representing initial state)
    - Initially, the start node is both a live node and an E-node
  - Try to move to a new node (node representing a new state we have never seen)
    - Success → Push current node into the stack if it is live, and make the new node a live node & E-node
    - Fail → Current node *dies* (i.e. it is no longer live) and we move back (*backtrack*) to the most recently seen live node in the stack
  - The search terminates when
    - we have found the answer, or
    - we run out of live nodes to back up to

SNU Internet DataBase Lab.

# Table of Contents



SNU Internet DataBase Lab.

# Rat in a Maze

- 3 x 3 rat-in-a-maze instance (Example 21.1)

entrance

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

exit

0 : road

1 : obstacle

- A maze is a tour puzzle in the form of a complex branching passage through which the solver must find a route
  - A maze is a graph
  - So, we can traverse a maze using DFS / BFS
- Backtracking ➔ Finding solution using DFS
  - Worst-case time complexity of finding path to the exit of n*n maze is $O(n^2)$

SNU Internet DataBase Lab.

# Backtracking in “Rat in a Maze”

1. Prepare an empty stack S and an empty 2D array
2. Initialize array elements with 1 where obstacles are, 0 elsewhere
3. Start at the upper left corner
4. Set the array value of current position to 1
5. Check adjacent (up, right, down and left) cell whose value is zero
   - If we found such cell, push current position into the stack and move to there
   - If we couldn’t find such cell, pop a position from the stack and move to there
6. If we haven't reach to the goal, repeat from 4

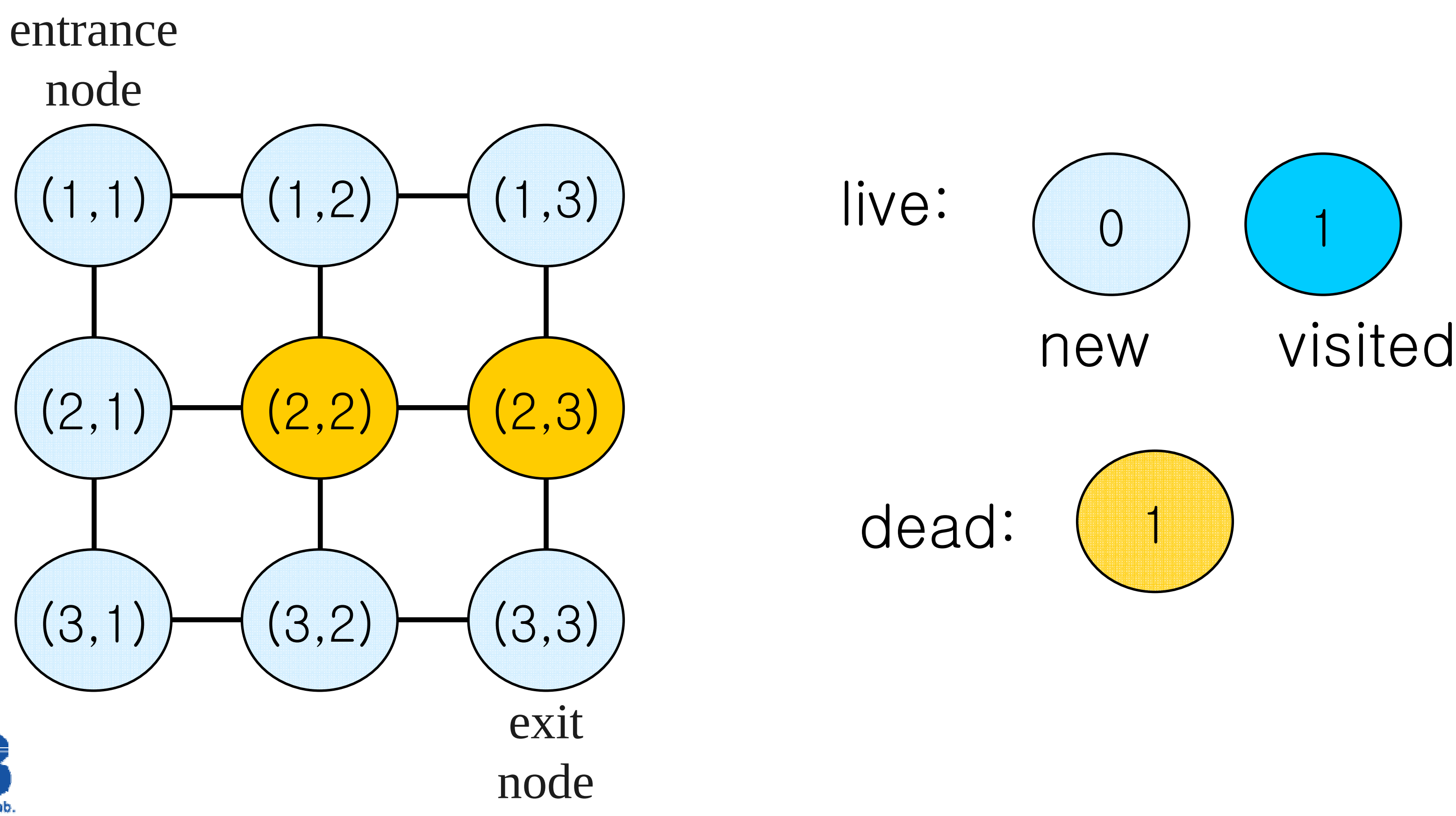
# Rat in a Maze Code

```
Prepare an empty stack and an empty 2D array
Initialize array elements with 1 where obstacles are, 0 elsewhere
i ← 1
j ← 1
Repeat until reach to the goal {
     a[i][j] ← 1;
     if (a[i][j+1]==0) {          put (i,j) into the stack
                                  j++; }
     else if(a[i+1][j]==0) {      put (i,j) into the stack
                                  i++; }
     else if (a[i][j-1]==0) {     put (i,j) into the stack
                                  j--; }
     else if (a[i-1][j]==0) {     put (i,j) into the stack
                                  i--; }
     else pop (i,j) from the stack;
}
```

SNU Internet DataBase Lab.

# Rat in a Maze Example (1)

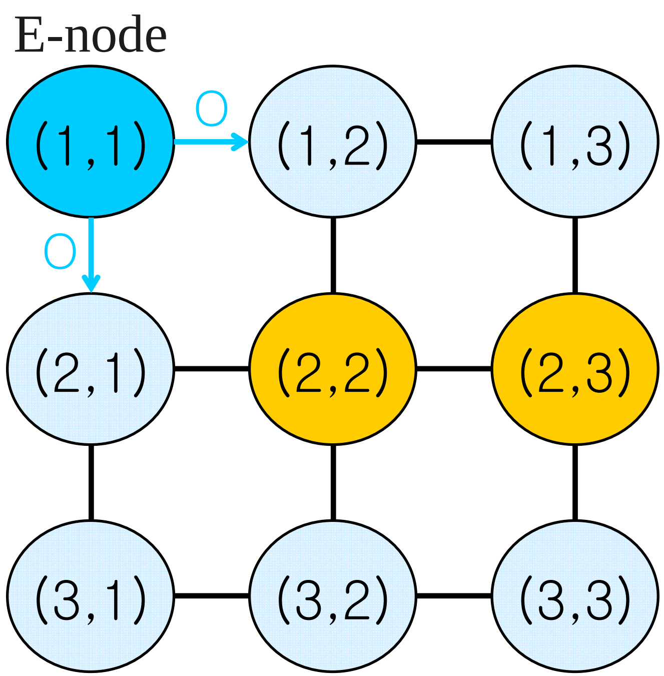
- Organize the solution space




SNU Internet DataBase Lab.

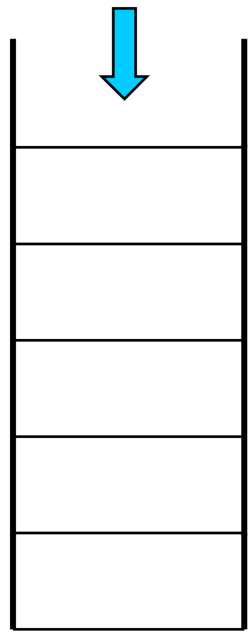
# Rat in a Maze Example (2)

- Search the graph in a depth-first manner to find a solution



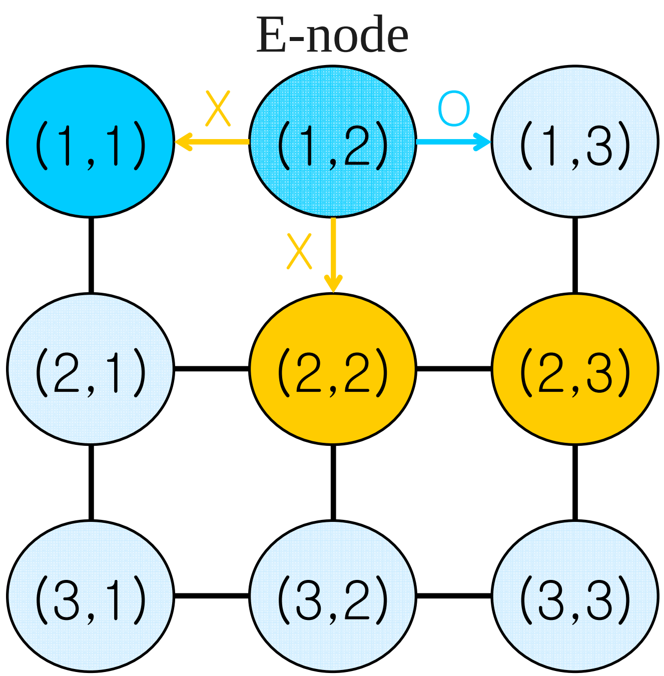

Push (1,1) &
Move to (1,2)

Live node stack

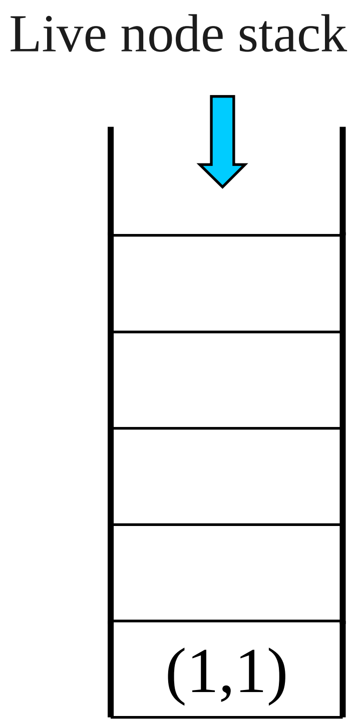


# Rat in a Maze Example (3)

- Search the graph in a depth-first manner to find a solution



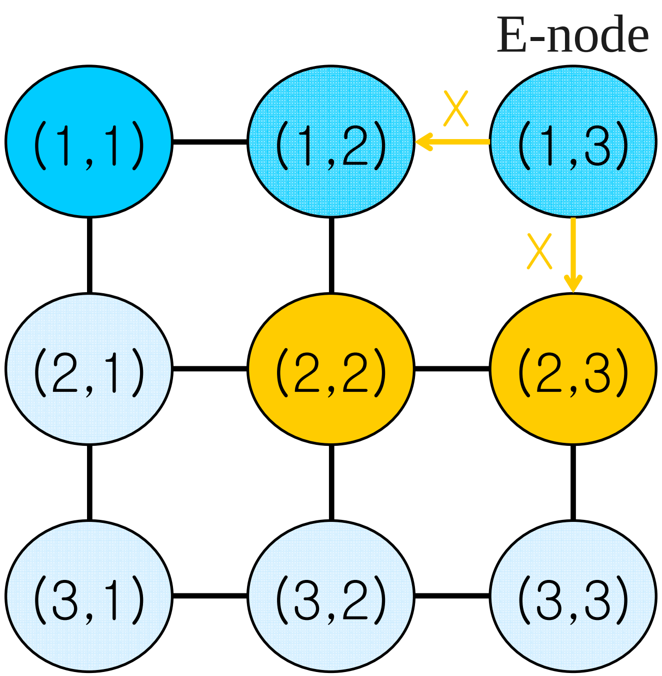

Push (1,2) &
Move to (1,3)



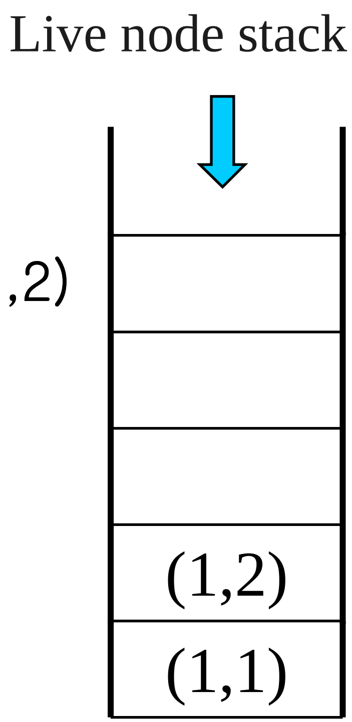

SNU Internet DataBase Lab.

# Rat in a Maze Example (4)

- Search the graph in a depth-first manner to find a solution

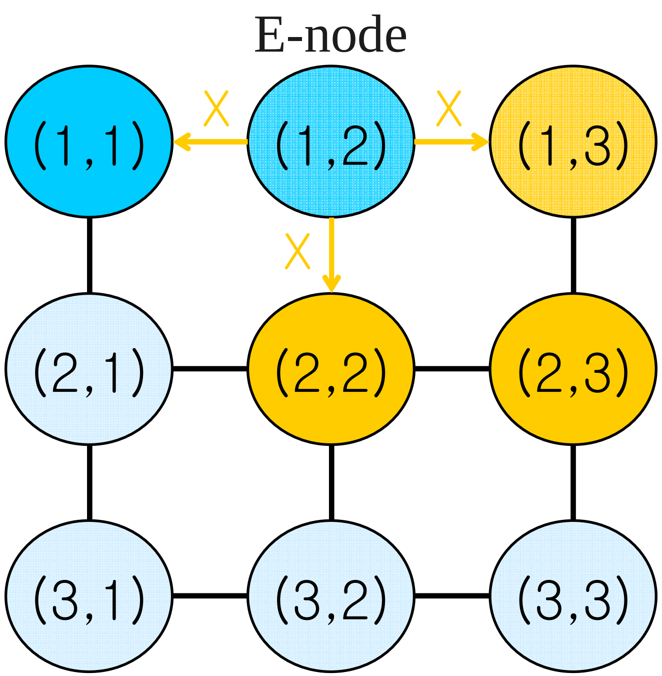



Pop (1,2) &
Backtrack to (1,2)

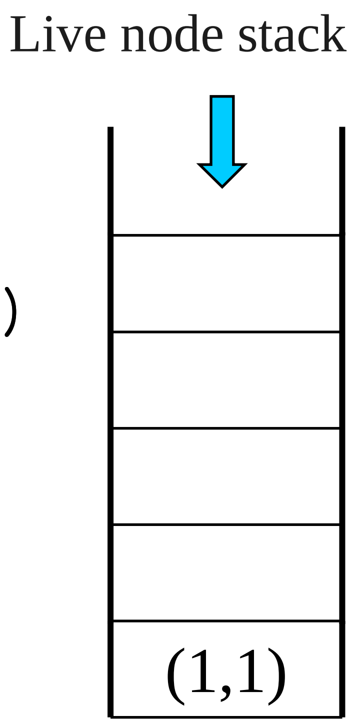


# Rat in a Maze Example (5)

- Search the graph in a depth-first manner to find a solution



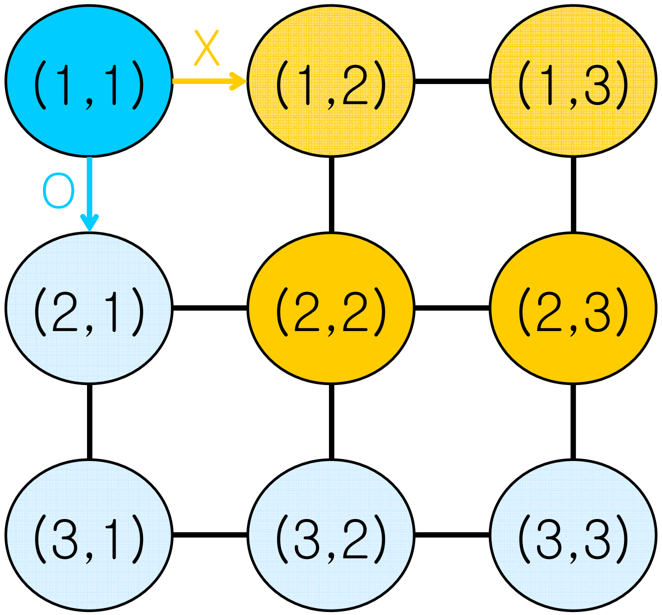

Pop (1,1) &
Backtrack (1,1)



# Rat in a Maze Example (6)

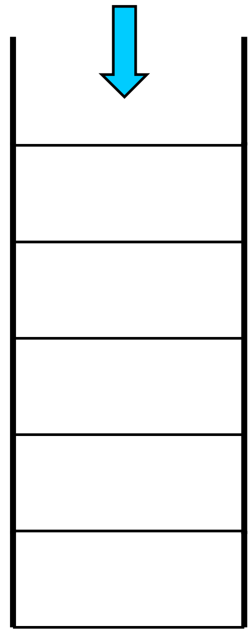
- Search the graph in a depth-first manner to find a solution

E-node



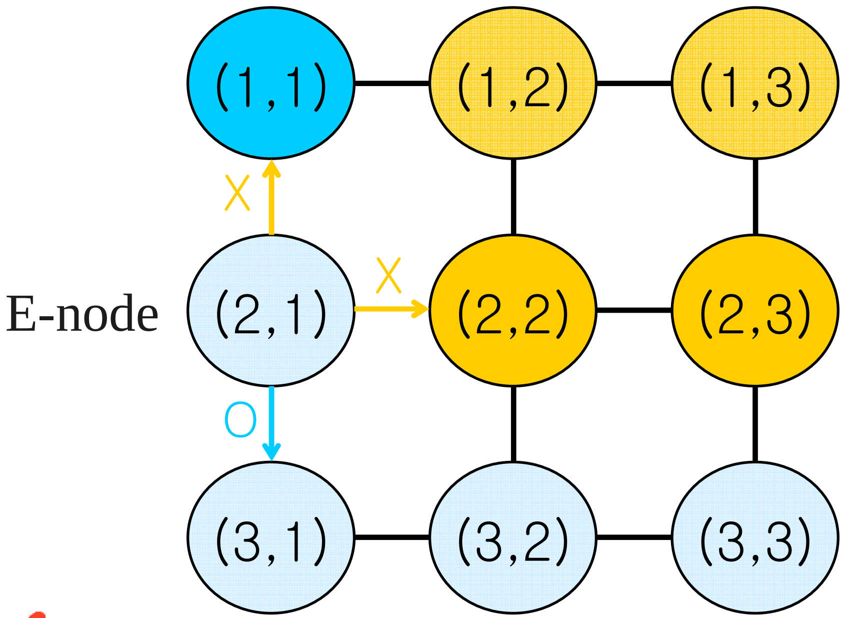

Push (1,1) &
Move to (2,1)

Live node stack

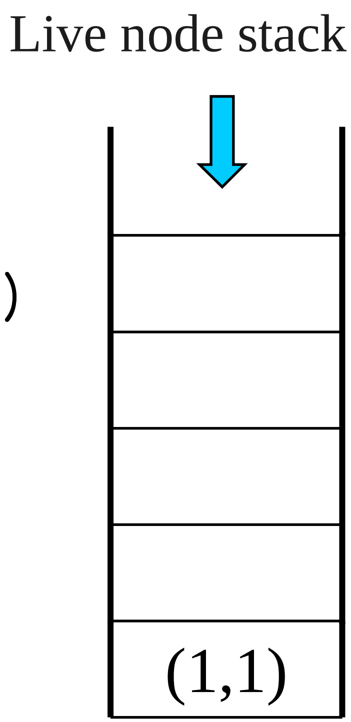


# Rat in a Maze Example (7)

- Search the graph in a depth-first manner to find a solution

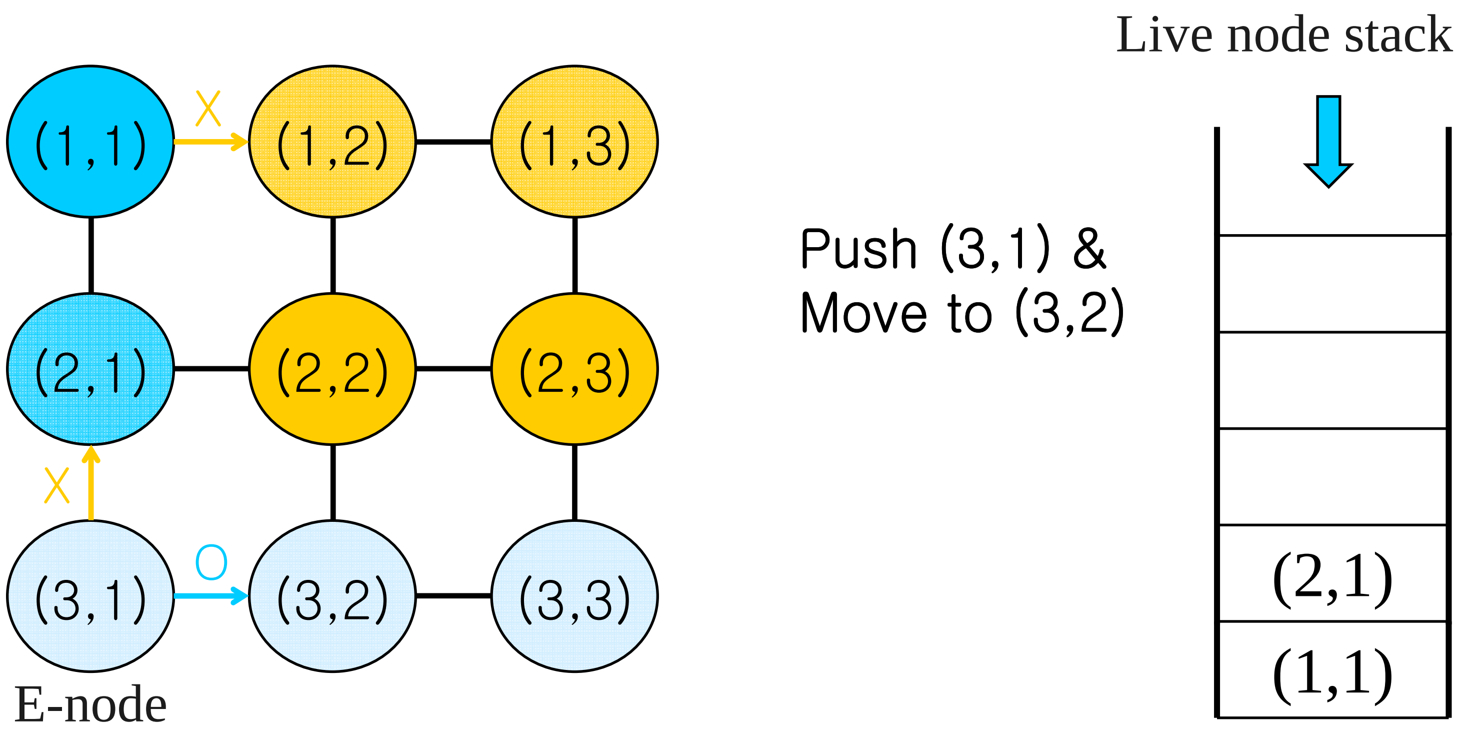



Push (2,1) &
Move to (3,1)



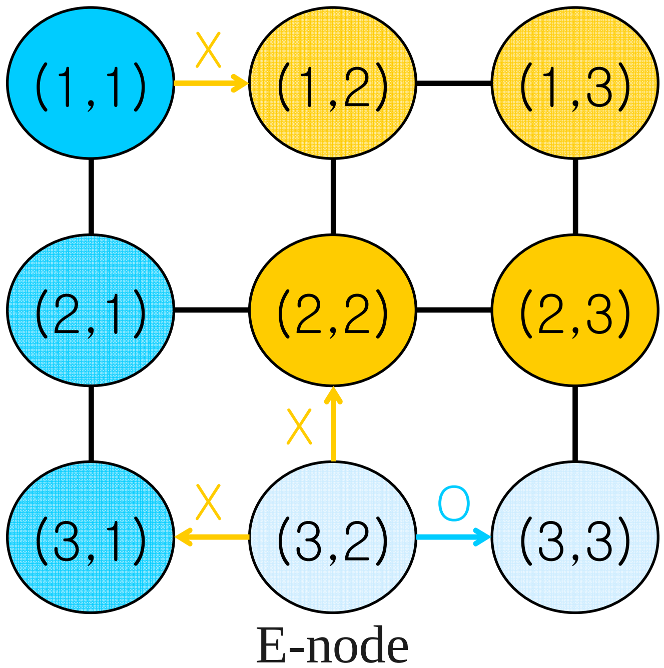
# Rat in a Maze Example (8)

- Search the graph in a depth-first manner to find a solution

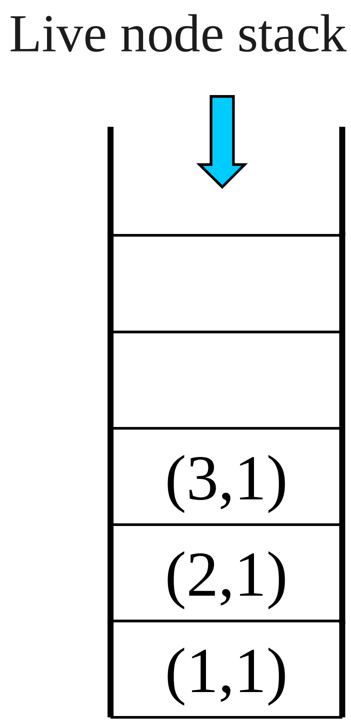


# Rat in a Maze Example (9)

- Search the graph in a depth-first manner to find a solution

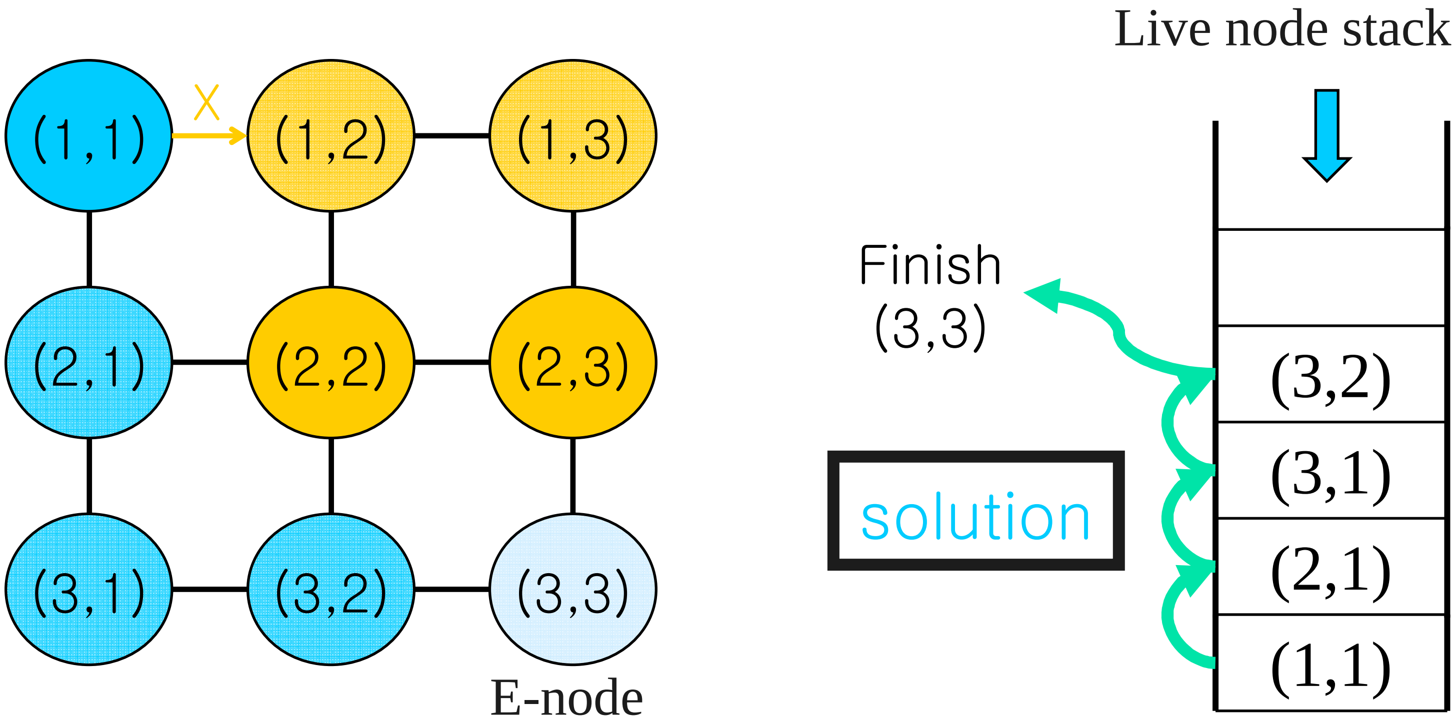



Push (3,2) &
Move to (3,3)



# Rat in a Maze Example (10)

- Search the graph in a depth-first manner to find a solution



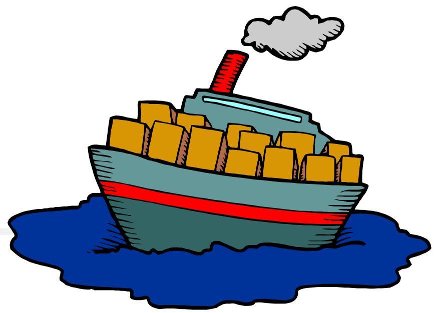

- Observation
  - Backtracking solution may not be a shortest path
  - Nodes in the stack represent the solution

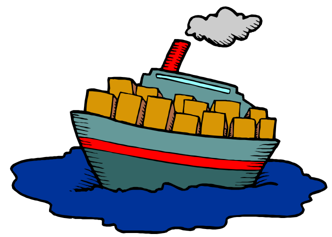
# Table of Contents

- The Method
- Application
  - Rat in a Maze
  - Container Loading

SNU Internet DataBase Lab.

# Container Loading





- Container Loading Problem (Example 21.4)
  - 2 ships and *n* containers
  - Ship capacity: $c_1$, $c_2$
  - The weight of container *i:* $w_i$
  - $$\sum_{i=1}^{n} w_i \le c_1 + c_2$$
  - Is there a way to load all *n* containers?

- Container Loading Instance
  - n = 4
  - $c_1 = 12$, $c_2 = 9$
  - w = [8, 6, 2, 3]
- Find a subset of the weights with sum as close to $c_1$ as possible

# Considering only One Ship

- Original problem: Is there any way to load n containers with

$$\sum_{i \text{ belongs to } ship_1} w_i \le c_1, \quad \sum_{i \text{ belongs to } ship_2} w_i \le c_2$$

- Because $\sum_{i \text{ belongs to } ship_1} w_i + \sum_{i \text{ belongs to } ship_2} w_i = \sum_{i=1}^{n} w_i$ is constant,

$$\max\Big(\sum_{i \text{ belongs to } ship_1} w_i\Big) = \min\Big(\sum_{i \text{ belongs to } ship_2} w_i\Big)$$

- So, all we need to do is trying to load containers at ship 1 as much as possible and check if the sum of weights of remaining containers is less than or equal to $c_2$

SNU Internet DataBase Lab.

# Solving without Backtracking

- We can find a solution with brute-force search

```
1. Generate n random numbers x1, x2, …, xn
   where xi = 0 or 1  (i = 1,…,n)
2. If xi = 1, we put i-th container into ship 1
   If xi = 0, we put i-th container into ship 2
3. Check if sum of weights in both ships are less
   than their maximum capacity
3-1.    If so, we found a solution!
3-2.    Otherwise, repeat from 1
```

- Above method are too naïve and not duplicate-free

➔ Backtracking provides a systematic way to search feasible solutions (still NP-complete, though)

SNU Internet DataBase Lab.

# Container Loading and Backtracking

- Container loading is one of NP-complete problems
  - There are $2^n$ possible partitionings
- If we represent the decision of location of each container with a *branch*, we can represent container loading problem with a *tree*
- Organize the solution space
  - Solution space is represented as a binary tree
  - Every node has a label, which is an identifier

- So, we can traverse the tree using DFS / BFS
- Backtracking = Finding solution using DFS
- Worst-case time complexity is $O(2^n)$ if there are n containers

SNU Internet DataBase Lab.

# Backtracking in Container Loading

1. Prepare an empty stack S and a complete binary tree T with depth *n*
2. Initialize the *max* to zero
3. Start from root of T
4. Let *t* as current node
5. If we haven't visit left child and have space to load $w_{depth(t)}$,
   then load it, push *t* into S and move to left child
   else if we haven't visit right child, push t into S and move to right child
6. If we failed to move to the child, check if the stack is empty
   1. If the stack is not empty, pop a node from the stack and move to there
7. If current sum of weights is greater than *max*, update *max*
8. Repeat from 4 until we have checked all nodes

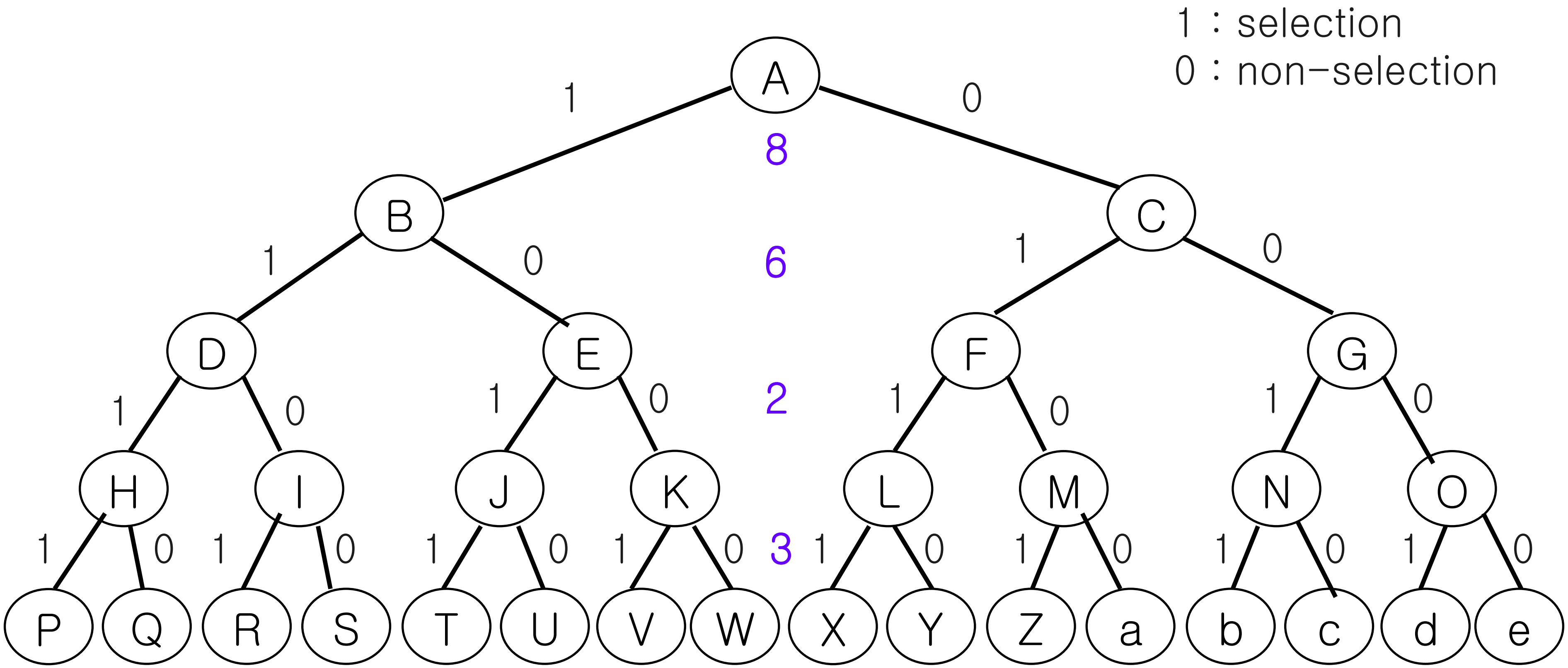
SNU Internet DataBase Lab.

# Container Loading Code

```
Consider n, c1, c2, w are given
Construct a complete binary tree with depth n &  Prepare an empty stack
max ← 0;        sum ← 0;       depth ← 0;    x ← root node of the tree;
While (true) {
    if (depth < n && !x.visitedLeft && c1 – sum ≥ w[depth]) {
        sum ← sum + w[depth]
        if (sum > max) max = sum;
        Put (x,sum) into the stack
        x.visitedLeft ← true;
        x ← x.leftChild;
        depth++; }
    else if  (depth < n && !x.visitedRight) {
            Put (x,sum) into the stack
            x.visitedRight ← true;
            x ← x.rightChild;
            depth++; }
        else  { if  (the stack is empty)  {
                        If sum(w) – max <= c2, max is the optimal weight
                        Otherwise, it is impossible to load all containers
                        Quit the program     }
                Pop (x,sum) from the stack;
                depth--;}
}
```

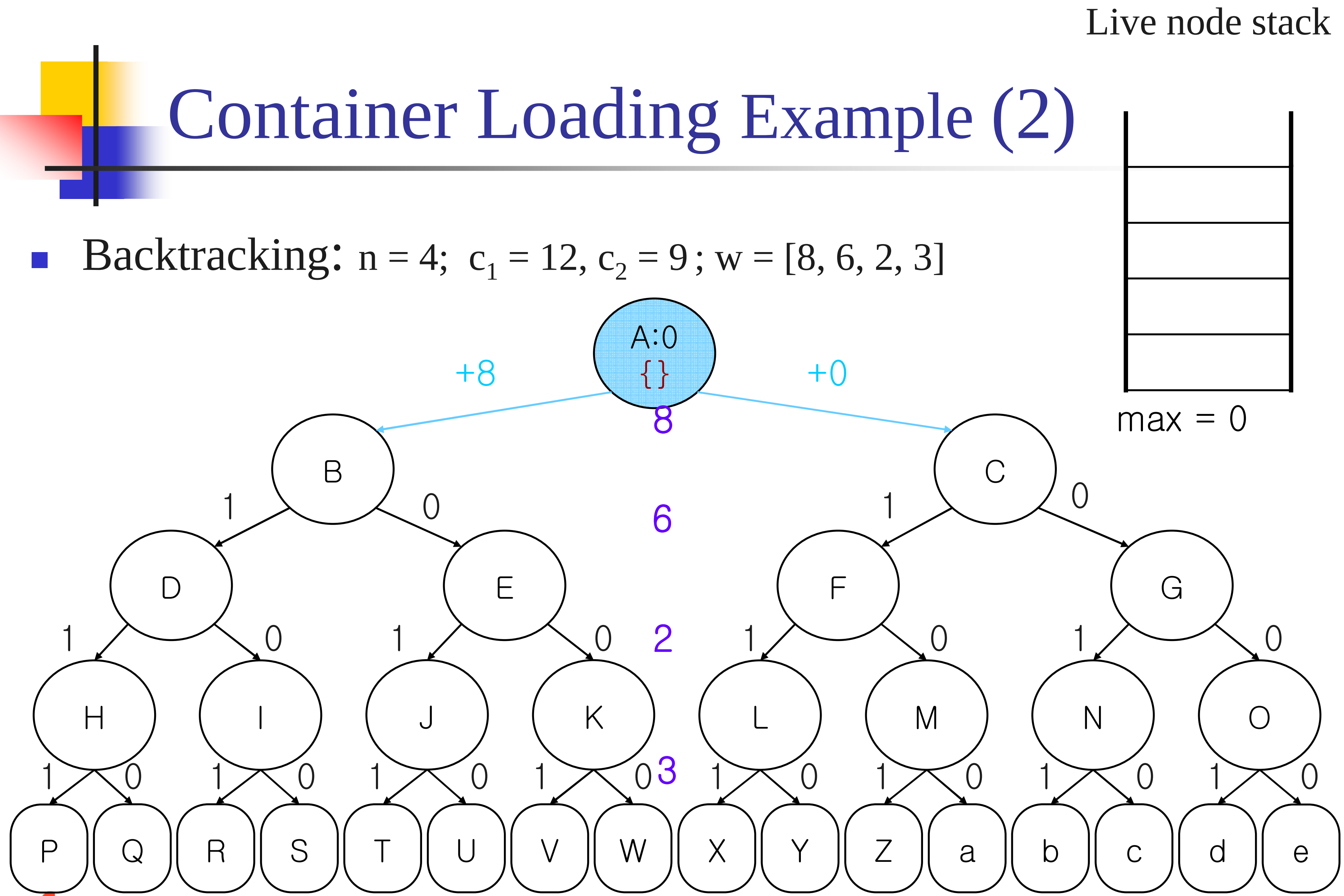
# Container Loading Example (1)

- Organize the solution space: n = 4; $c_1 = 12$, $c_2 = 9$ ; w = [8, 6, 2, 3]



# Container Loading Example (2)

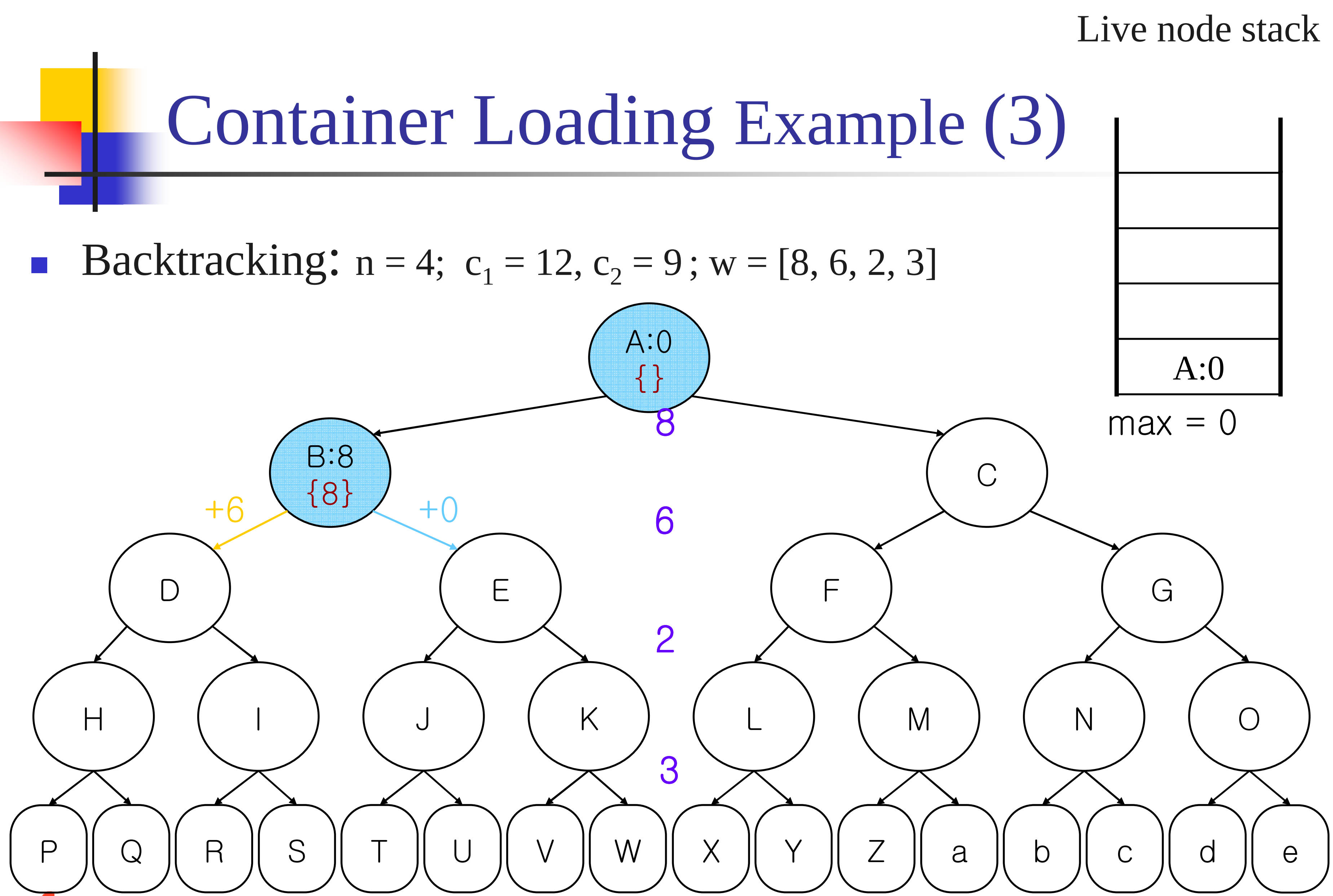
- Backtracking: $n = 4$; $c_1 = 12$, $c_2 = 9$ ; $w = [8, 6, 2, 3]$




Push A:0 and Move to B

# Container Loading Example (3)

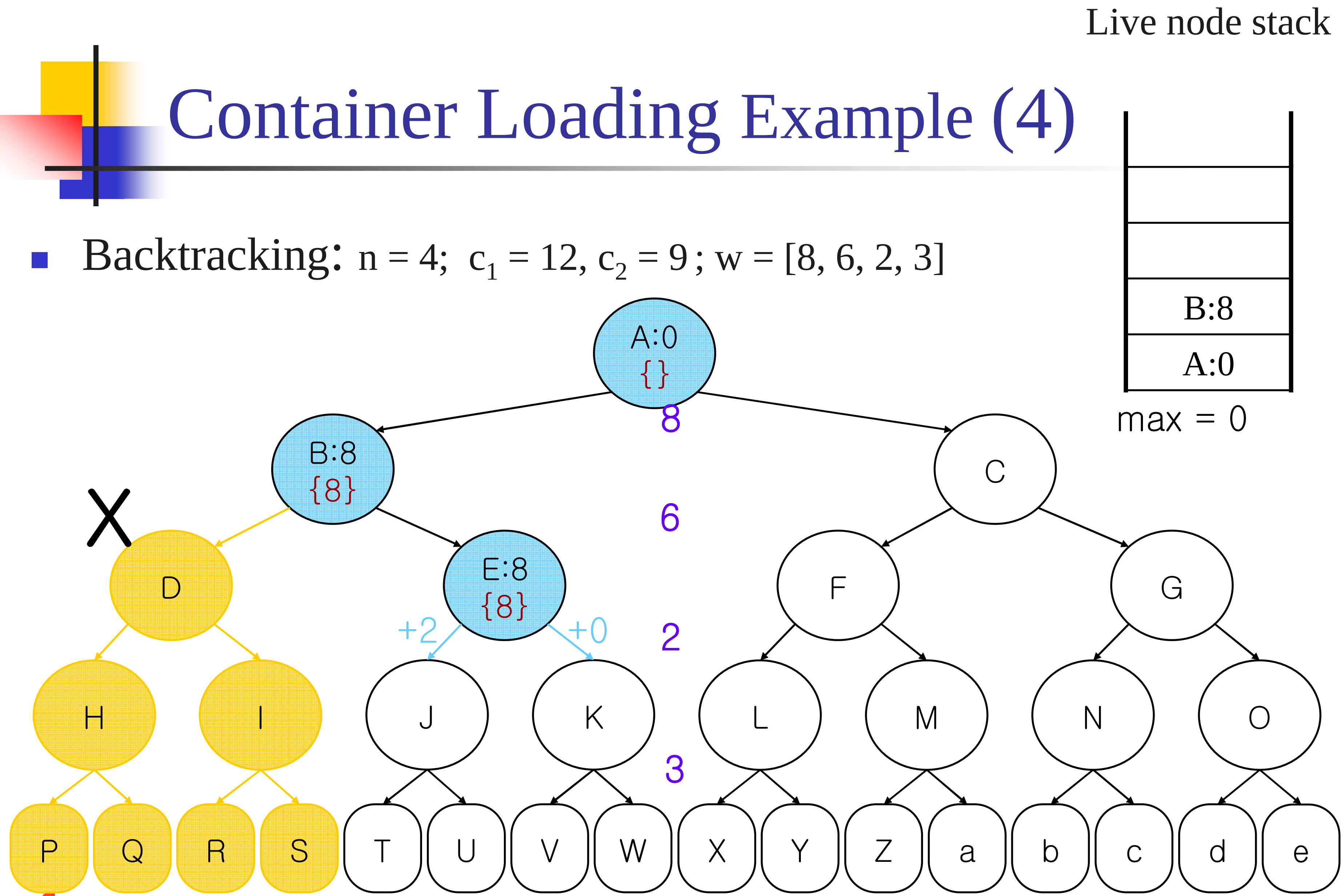
- Backtracking: $n = 4$; $c_1 = 12$, $c_2 = 9$ ; $w = [8, 6, 2, 3]$




Push B:8 and Move to E

# Container Loading Example (4)

- Backtracking: $n = 4$; $c_1 = 12$, $c_2 = 9$ ; $w = [8, 6, 2, 3]$

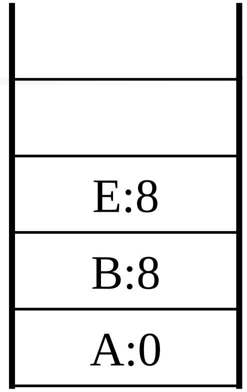



Push E:8 and Move to J

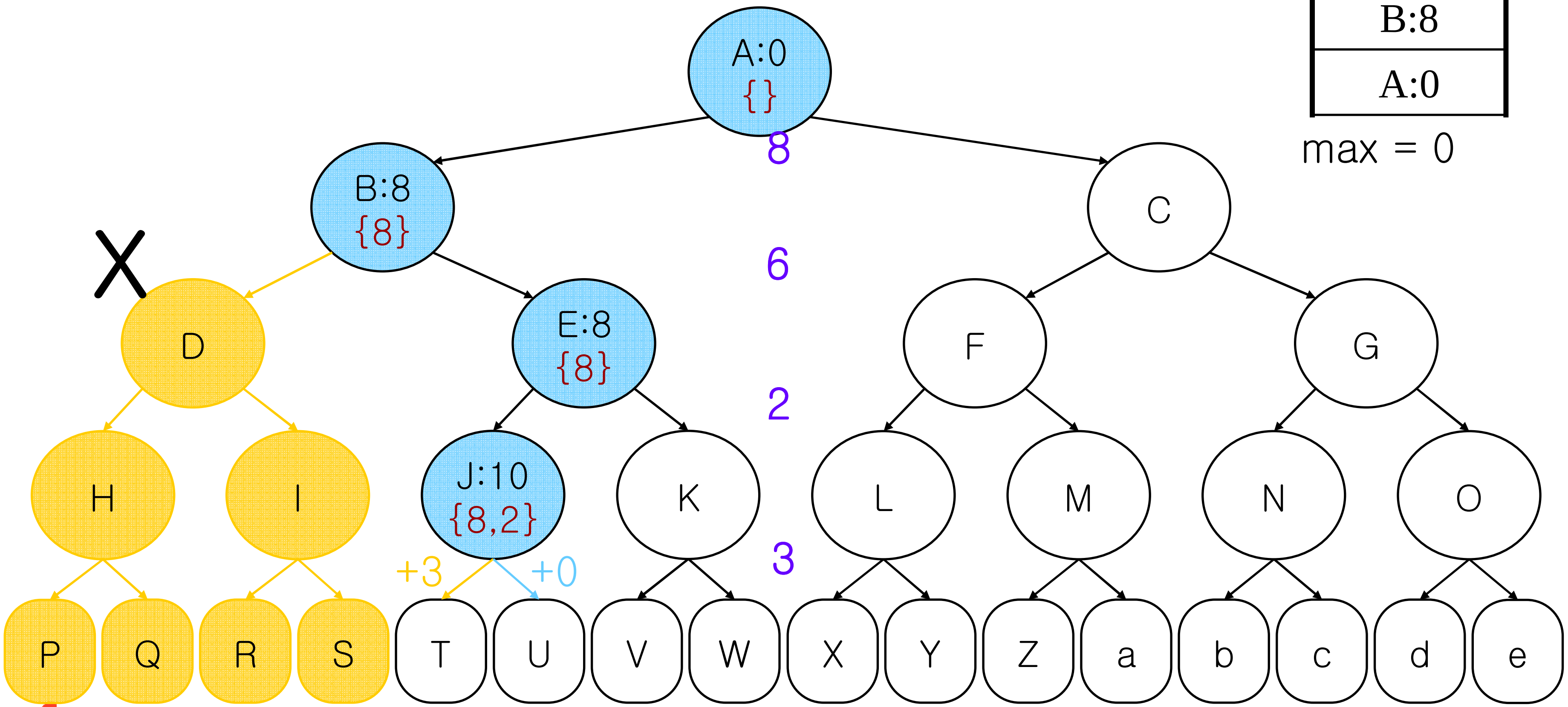
# Container Loading Example (5)

- Backtracking: n = 4;  $c_1$ = 12, $c_2$ = 9 ; w = [8, 6, 2, 3]

Live node stack



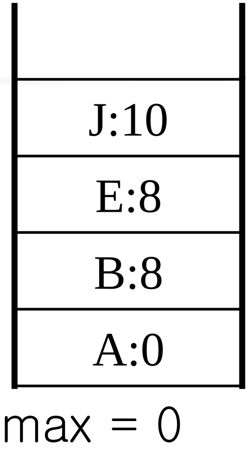

max = 0



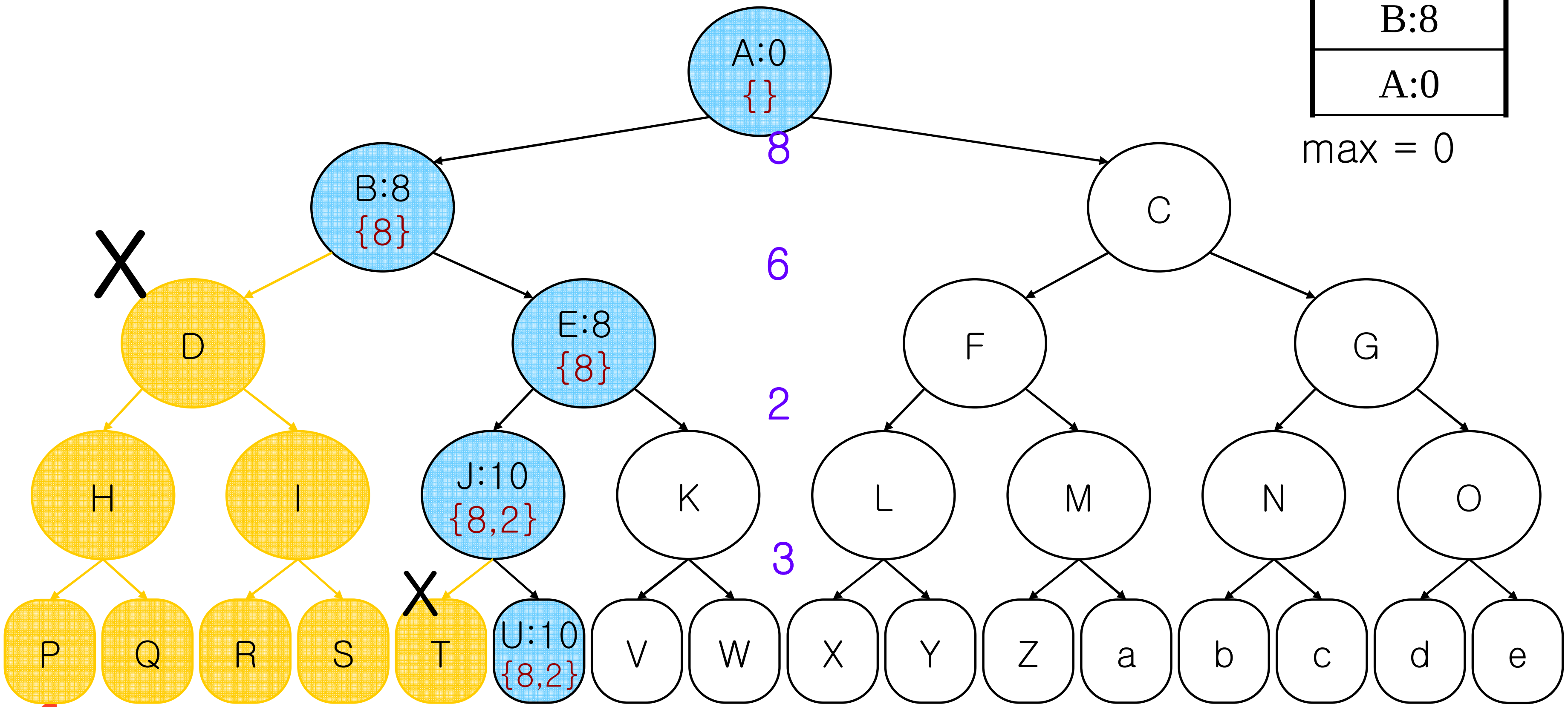

Push J:10 and Move to U

Live node stack

# Container Loading Example (6)



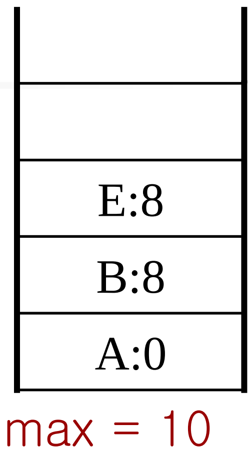

- Backtracking: n = 4; $c_1 = 12$, $c_2 = 9$ ; w = [8, 6, 2, 3]



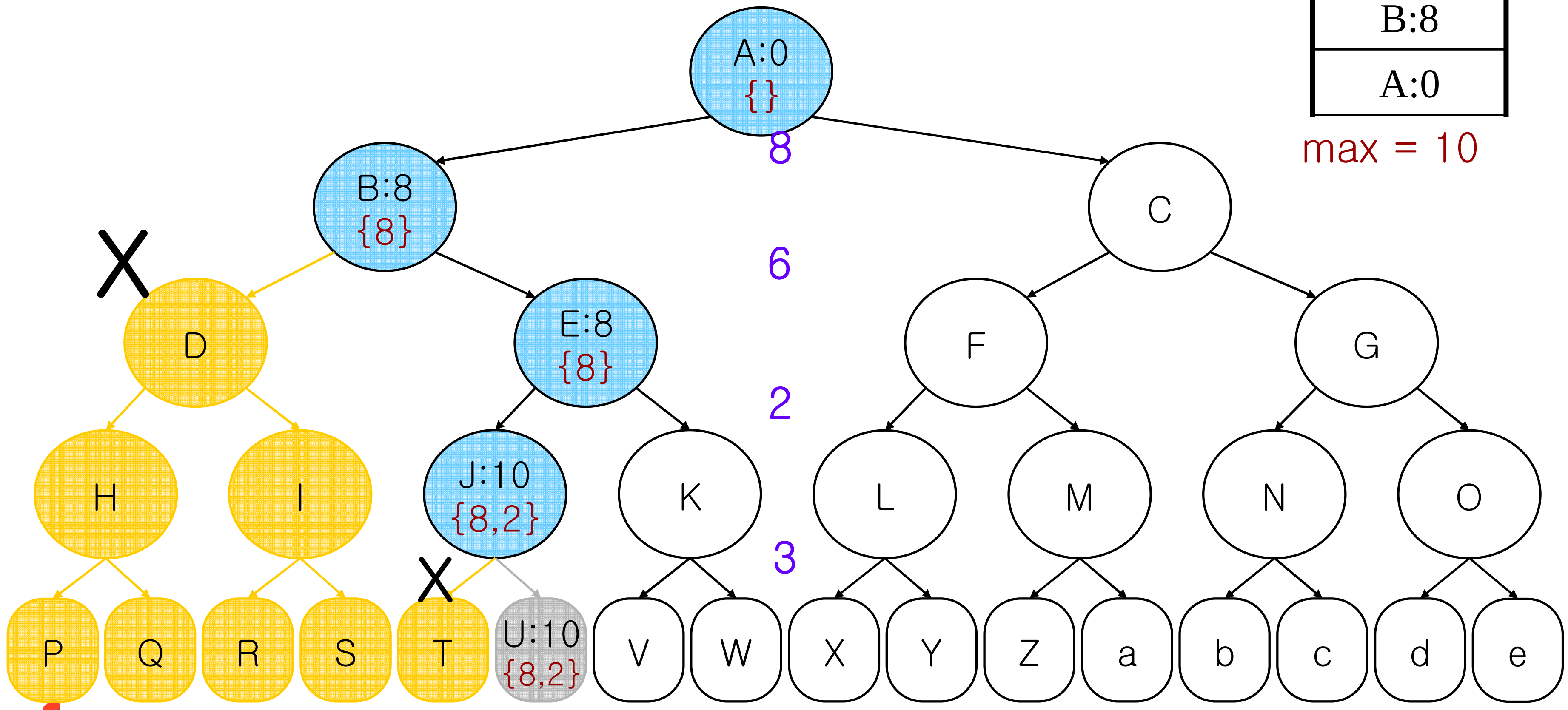

Set max ←10, Pop J:10 and Backtrack to J

Live node stack

# Container Loading Example (7)



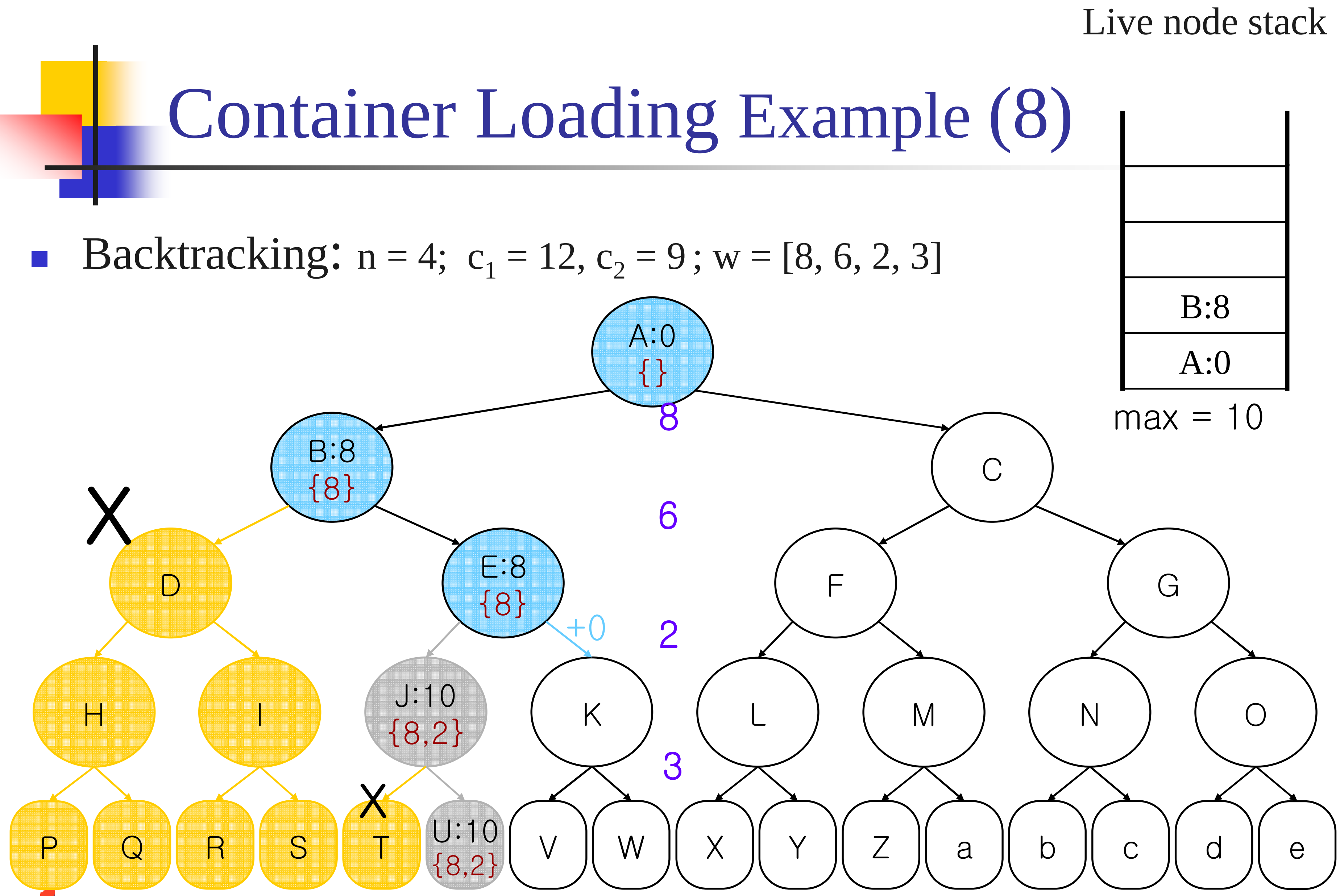

- Backtracking: $n = 4$; $c_1 = 12$, $c_2 = 9$ ; $w = [8, 6, 2, 3]$




Pop E:8 and Backtrack to E

# Container Loading Example (8)

- Backtracking: $n = 4$; $c_1 = 12$, $c_2 = 9$ ; $w = [8, 6, 2, 3]$

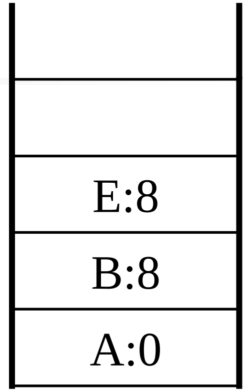



Push E:8 and move to K

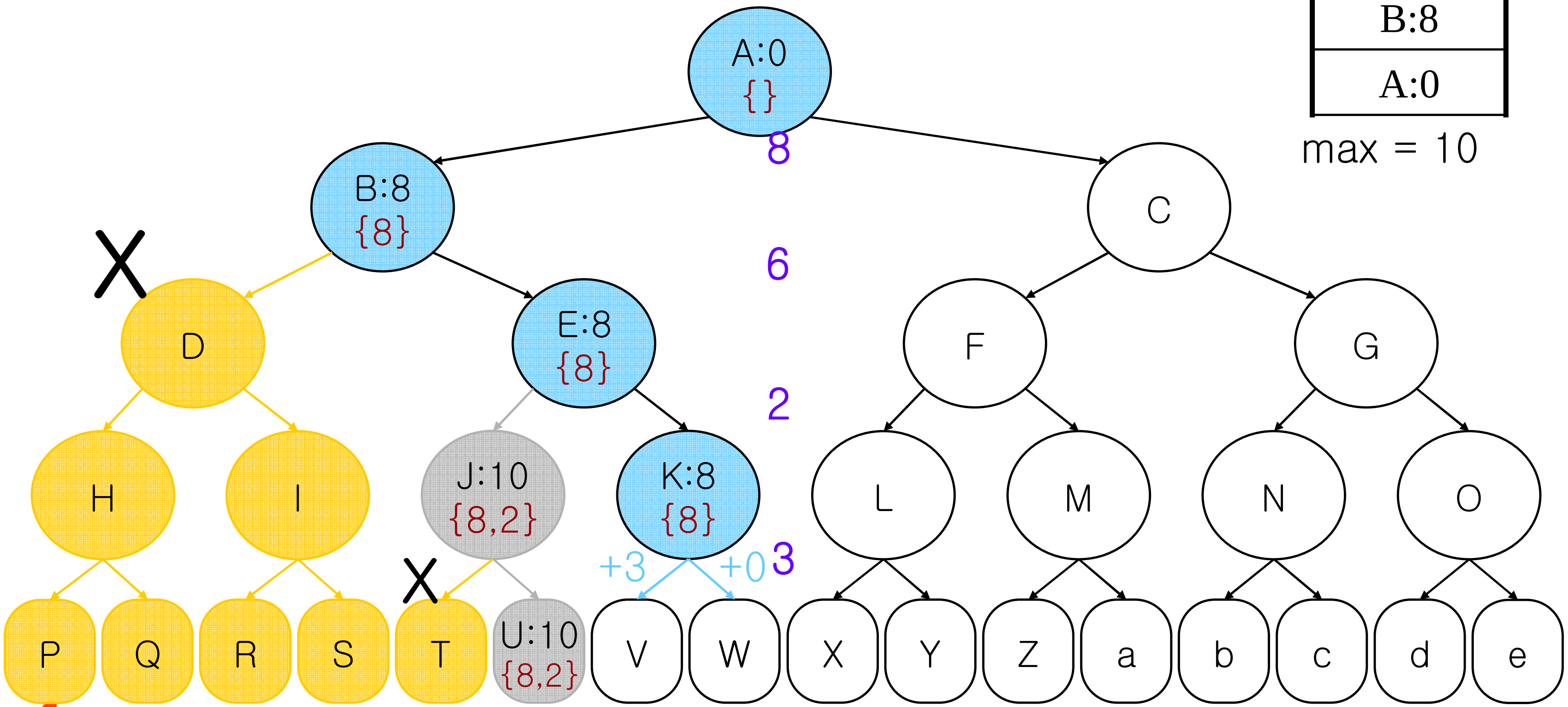
# Container Loading Example (9)

- Backtracking: n = 4;  $c_1 = 12$, $c_2 = 9$ ; w = [8, 6, 2, 3]

Live node stack



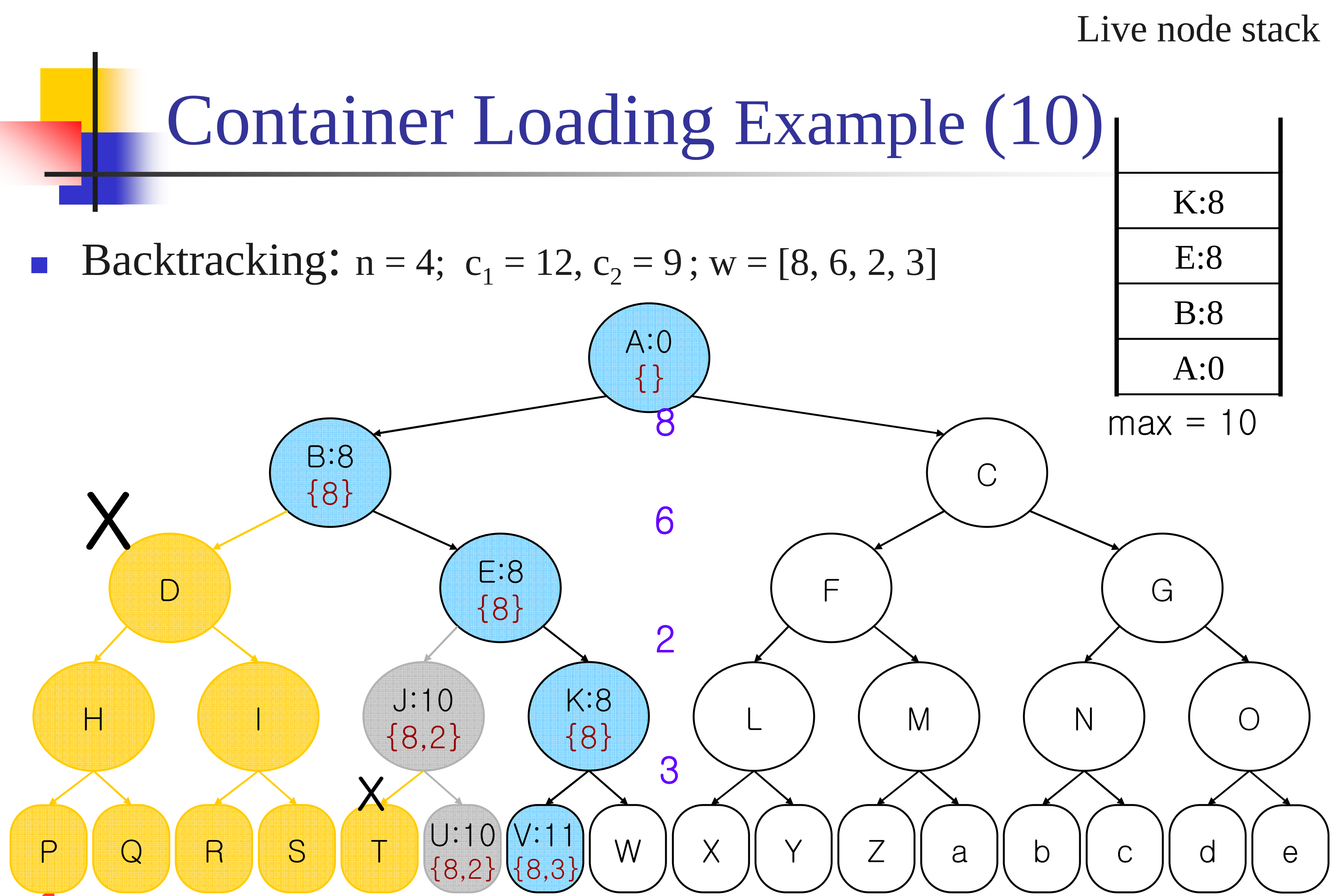

max = 10




Push K:8 and move to V

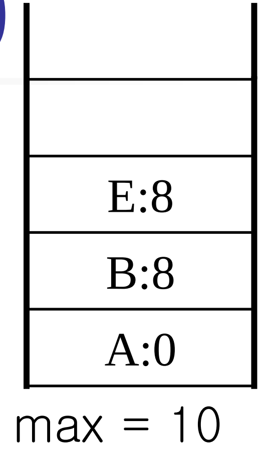
# Container Loading Example (10)

- Backtracking: $n = 4$; $c_1 = 12$, $c_2 = 9$; $w = [8, 6, 2, 3]$



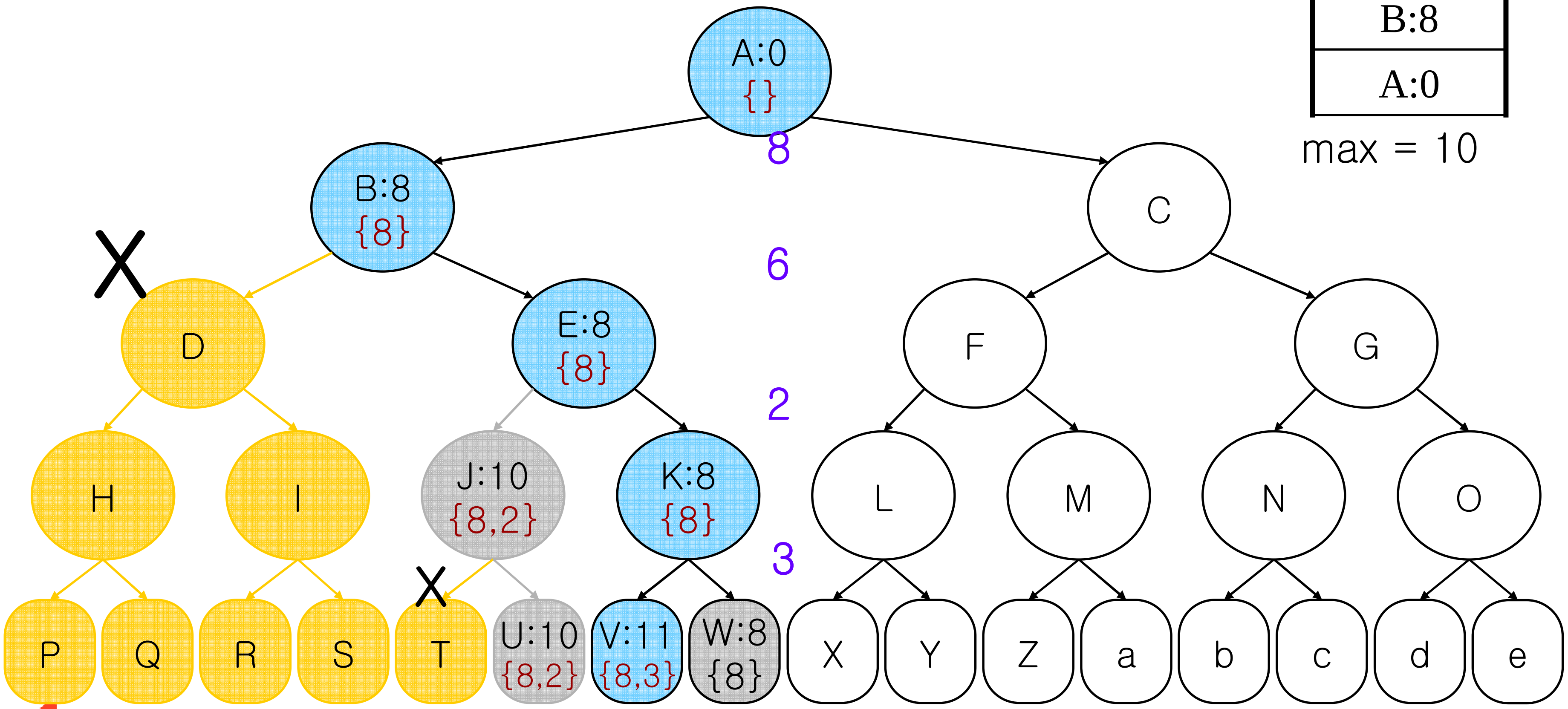

Set max←11, pop K:8 and Backtrack to K

Live node stack




# Container Loading Example (11)

- Backtracking: n = 4; $c_1 = 12$, $c_2 = 9$ ; w = [8, 6, 2, 3]

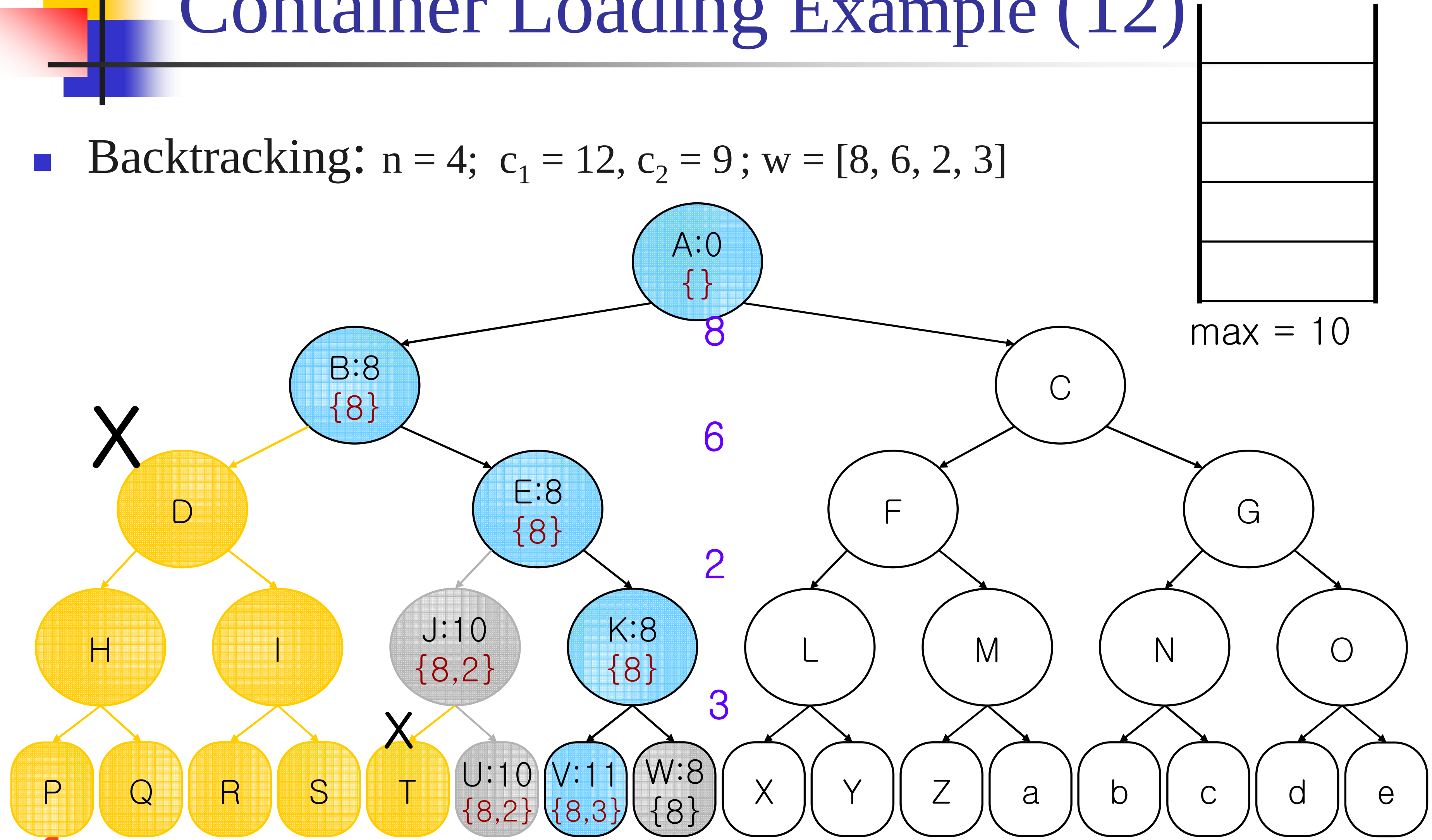



pop E:8 & Backtrack to E; pop B:8 & Backtrack to B; pop A:8 & Backtrack to A;

Live node stack

# Container Loading Example (12)

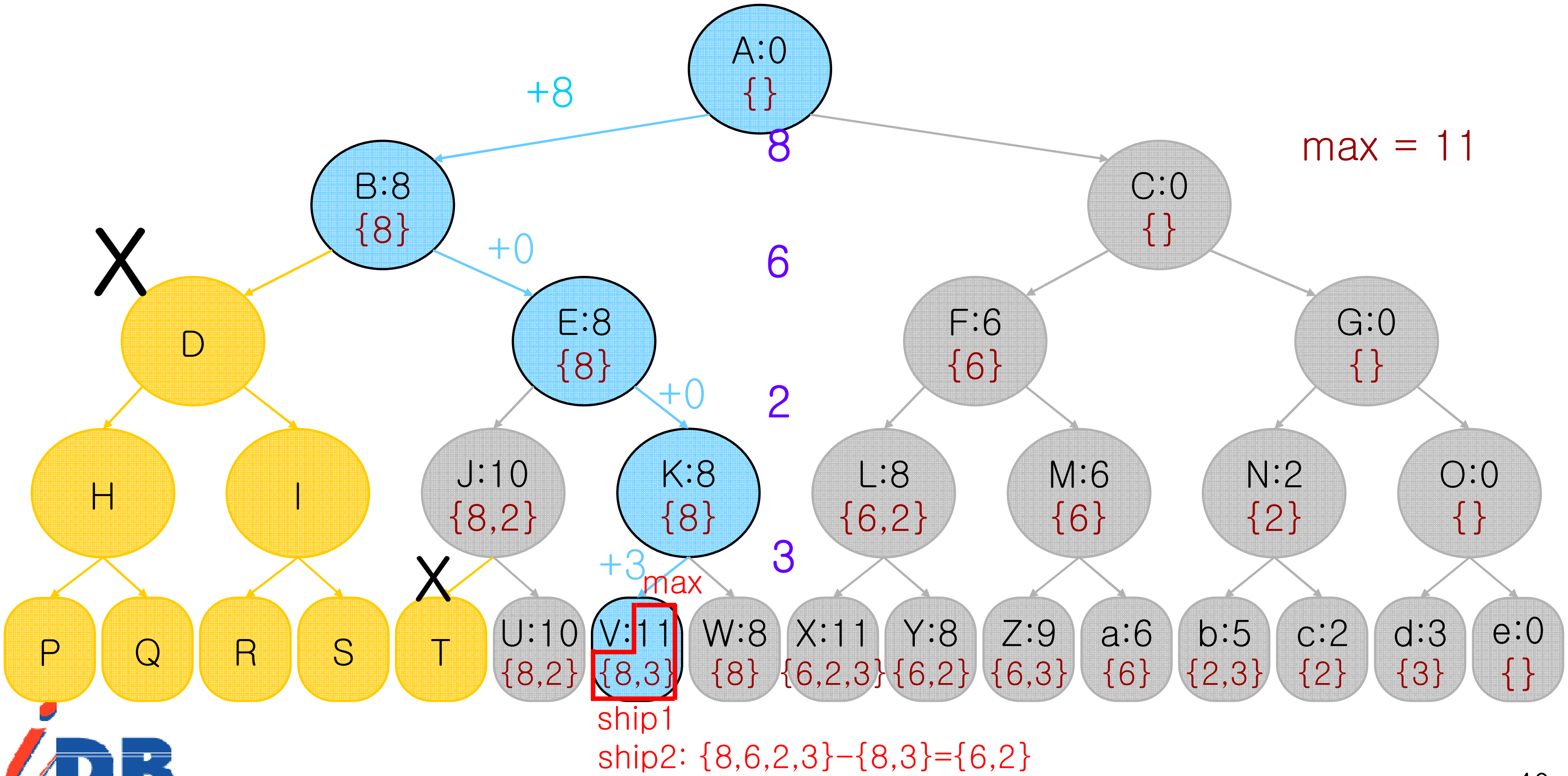
- Backtracking: $n = 4$; $c_1 = 12$, $c_2 = 9$ ; $w = [8, 6, 2, 3]$




Process the right subtree of A node with the same previous manner

# Container Loading Example (13)

- Backtracking: $n = 4$; $c_1 = 12$, $c_2 = 9$ ; $w = [8, 6, 2, 3]$




SNU Internet DataBase Lab.

# Bird's-Eye View

- A surefire way to solve a problem is to make a list of all candidate answers and check them
  - If the problem size is big, we can not get the answer in reasonable time using this approach
  - List all possible cases ➔ exponential cases
- By a systematic examination of the candidate list, we can find the answer without examining every candidate answer
  - *Backtracking* and *Branch and Bound* are most popular systematic algorithms

** Surefire = 확실한, 틀림없는

SNU Internet DataBase Lab.

# Table of Contents

- The Backtracking Method
- Application
  - Rat in a Maze
  - Container Loading

SNU Internet DataBase Lab.