

Stack Applications

- **Reversing data items**
Ex.: Reverse a list.
Convert Decimal to Binary.
- **Parsing**
Ex.: Brackets Parse.
- **Postponement of processing data items**
Ex.: Infix to Postfix Transformation.
Evaluate a Postfix Expression.
- **Backtracking**
Ex.: Goal Seeking Problem.
Knight's Tour.
Exiting a Maze.
Eight Queens Problem.

Reverse a list

PROBLEM: Read n numbers, print the list in reverse order.

Algorithm **ReverseList**

Pre User supplies numbers.

Post The numbers are printed in reverse order.

Uses Stack ADT.

1. loop (stack is not full and there is more number)

1. read a number
2. push the number into the stack

2. loop (stack is not empty)

1. top the number from the stack
2. pop stack
3. write the number

end ReverseList

Reverse a list

Algorithm **ReverseList()**

1. `stackObj <Stack>`
 2. `stackObj.Create()`
 3. **loop** (not `stackObj.isFull()` and there is more number)
 1. read (number)
 2. `stackObj.Push(number)`
 4. **loop** (not `stackObj.isEmpty()`)
 1. `stackObj.Top(number)`
 2. `stackObj.Pop()`
 3. write (number)
 5. `stackObj.Clear()`
- end ReverseList

Usage of an ADT's Object

In some compilers,

- When an object is declared, it's **default constructor** (constructor without parameters) is called to make it empty.
- Before going out of the scope, the object's **destructor** is called to make it empty.

```
stackObj <Stack>
```

```
stackObj.Create()
```

*(use stackObj in
application's
algorithm)*

```
stackObj.Clear()
```

In our later pseudocode, in order to use an ADT's object, we just declare like that

```
Obj <ObjType>
```

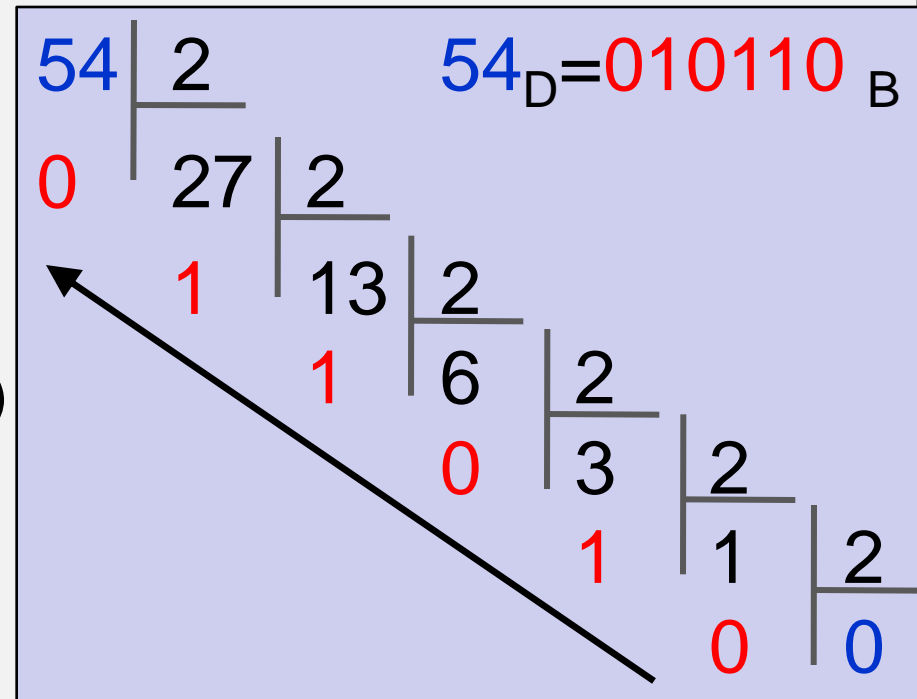
In building an ADT library, we must consider that task: **making an object empty before it's using and before it's going out of the scope** by writing its default constructor and destructor.

Convert Decimal to Binary

<ErrorCode> **Convert()**

PROBLEM: Read a decimal number and convert it to binary.

1. `stackObj` <Stack>
2. read (number)
3. **loop** (not `stackObj.isFull()` and number >0)
 1. digit = number modulo 2
 2. `stackObj.Push(digit)`
 3. number = number / 2
4. **if** (number > 0)
 1. return *overflow*
5. **else**
 1. **loop**(not(`stackObj.isEmpty()`))
 1. `stackObj.Top(digit)`
 2. `stackObj.Pop()`
 3. write(digit)
 2. return *success*



Parsing

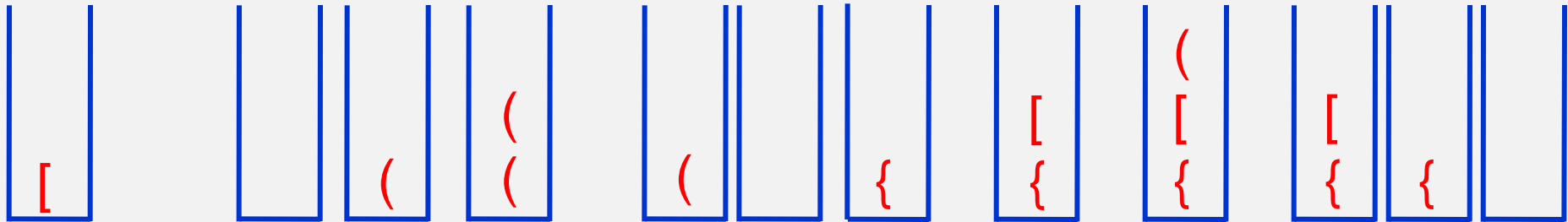
- ✓ **Parsing** is any logic that breaks data into independent pieces for further processing.
- ✓ Ex. : A compiler must parse the program into individual parts such as keywords, names, and other tokens.

Parsing

BracketParse:

Check the brackets are correctly matched or not.

[A + B] / (C * (D + E)) - { M + [A - (B + D)] }



Parsing

<ErrorCode> **BracketParse()**

Check the brackets are correctly matched or not.

Pre None.

Post Print the results of bracket-matched checking:

(1) Unmatched closing bracket detected.

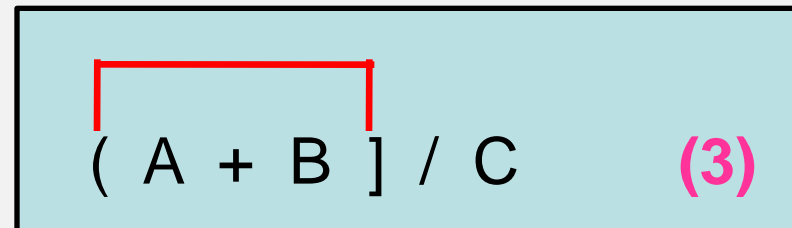
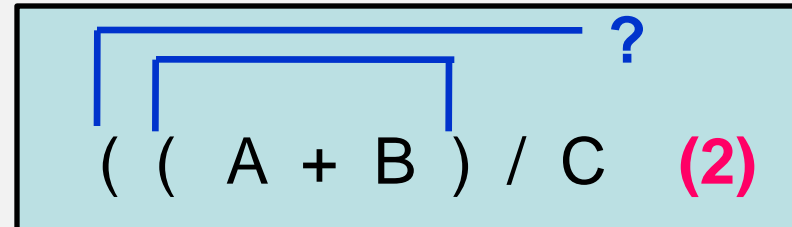
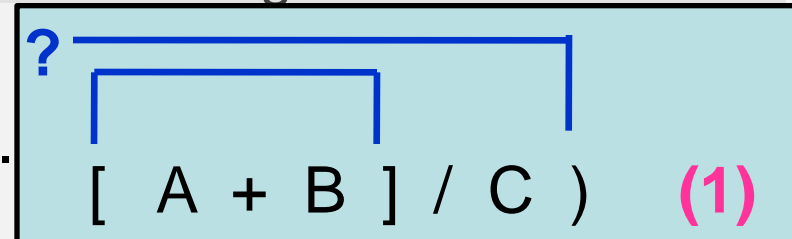
(2) Unmatched opening bracket detected.

(3) Bad match symbol.

(4) Stack is overflow.

Return *failed* or *success*.

Uses Stack ADT, function **isMatched**.



isMatched Function

<boolean> **isMatched** (**opening** <character>, **closing** <character>)

Checks the brackets are matched or not.

1. **Pre** **opening** and **closing** is one of the brackets: (, [, {,),], }.
2. **Post** Return TRUE if both opening and closing are paired off, FALSE otherwise.
3. **Return** **TRUE** or **FALSE**

Parsing

<ErrorCode> **BracketParse()**

1. **stackObj** <Stack>
2. **loop** (more data)
 1. read (character)
 2. **if** (character is an opening bracket)
 1. **if** (**stackObj.isFull()**)
 1. write (**stack overflow**)
 2. return *failed*
 2. **else**
 1. **stackObj.Push**(character)
 3. **else**

Parsing

3. **else** *// character is not an opening bracket*
 1. **if** (character is a closing bracket)
 1. **if** (`stackObj.isEmpty()`)
 1. write (**Unmatched closing bracket detected**)
 2. return *failed*
 2. **else**
 1. `stackObj.Top`(opening bracket)
 2. `stackObj.Pop`()
 3. **if** (not `isMatched` (opening bracket, character))
 1. write (**Bad matched symbol**)
 2. return *failed*
 1. **if** (not `stackObj.isEmpty()`)
 1. write (**Unmatched opening bracket detected**)
 2. return *success*

Postponement

Postponement: The usage of data is deferred until some later point.

Ex.: $5 \ 2 \ *$ \longrightarrow $5 * 2 = 10$

Evaluate a Postfix Expression: all operands will not be processed until their operator appears.

Ex.: $a \ * \ b$ \longrightarrow $a \ b \ *$
 $a \ * \ b \ + \ c$ \longrightarrow $a \ b \ * \ c \ +$
 $a \ + \ b \ * \ c$ \longrightarrow $a \ b \ c \ * \ +$

Infix to Postfix: writing the operator to the output needs to postpone until it's operands have been processed.

Evaluate a Postfix Expression

Postfix	
2 4 6 + * 5 -	
2 4 6 + * 5 -	2
2 4 6 + * 5 -	4 2
2 4 6 + * 5 -	6 4 2

Postfix	
2 4 6 + * 5 -	10 2
2 4 6 + * 5 -	20
2 4 6 + * 5 -	5 20
2 4 6 + * 5 -	15

$4 + 6 = 10$

$10 * 2 = 20$

$20 - 5 = 15$

Infix to Postfix Transformation

<ErrorCode> **InfixToPostfix** (val **infix** <text>, ref **postfix** <text>)

Transforms an infix expression to postfix.

Pre **infix** is a valid infix expression with operators associated from left to right (+, -, *, /).

Post **postfix** has received valid postfix expression.

Return *success* or *failed* (*failed* when the stack is *overflow*).

Uses Stack ADT and function **Process**.

1. **stackObj** <Stack>
2. **loop** (more symbol in **infix**)
 1. read (symbol)
 2. errorCode = **Process** (symbol, **postfix**, **stackObj**)
 3. **if** (errorCode = *overflow*)
 1. return *failed*.
3. Pop the stack until it is empty, put all elements into **postfix**.
return *success*.

Infix		Postfix
a +b*c-(d*e / f)*g		a
a+ + b*c-(d*e / f)*g	+	a
a+b* c -(d*e / f)*g	+	ab
a+b*c- (d*e / f)*g	* +	ab
a+b*c-(d* e / f)*g	* +	abc

Infix		Postfix
a+b*c- - (d*e / f)*g	-	abc* +
a+b*c- (d*e / f)*g	(-	abc*+
a+b*c-(d* e / f)*g	(-	abc*+ d
a+b*c-(d*e / f)*g	* (-	abc*+d
a+b*c-(d*e / f)* g	* (-	abc*+de

Infix		Postfix
a+b*c- (d*e / f)*g	<div>/</div> <div>(</div> <div>-</div>	abc*+de*
a+b*c- (d*e / f)*g	<div>/</div> <div>(</div> <div>-</div>	abc*+de*f
a+b*c- (d*e / f)*g	<div>-</div>	abc*+de*f/
a+b*c- (d*e / f)*g	<div>*</div> <div>-</div>	abc*+de*f/
a+b*c- (d*e / f)*g	<div>*</div> <div>-</div>	abc*+de*f/g*

	Postfix
<div>-</div>	abc*+de*f/g*-

Process Function

```
<ErrorCode> Process(val symbol <char>,  
                        ref output <text>,  
                        ref stackObj <Stack>)
```

Processes the symbol depend on it's type.

Pre *symbol* is one of valid symbols in an expression (operand, operator (+, -, *, /), parenthesis symbol)

Post *output* and *stackObj* have been updated appropriately.

1. Case (*symbol*) of:
 1. **Left parenthesis**: push into *stackObj*.
 2. **Right parenthesis**: pop *stackObj*, put all elements into *output* until encounter a (corresponding) left parenthesis, which is popped but not output.
 3. **Operand**: put into *output*.

Process Function (cont.)

```
<ErrorCode> Process(val symbol <char>,  
                      ref output <text>,  
                      ref stackObj <Stack>)
```

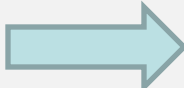
Case (symbol) of: (*cont.*)

4. Operator:

1. If the **priority of the new operator** is **higher than** the **priority of the operator at the top of stackObj**, push it into **stackObj**.
2. Otherwise, **all operator at the top of stackObj**, having priority **higher than** or **equal** the **new operator's priority**, need to be pop and put into **output** before pushing the new operator into **stackObj**.

Return **overflow** if **stackObj** is **overflow**, **success** otherwise.

Priority of operators

- Priority of the operators associated from left to right:
 - Priority 2: * /
 - Priority 1: + -
 - Priority 0: (
 - Operators associated from right to left:
 - Exponent
 - Logarithm
-  The algorithm must be changed.

Backtracking

Common idea of backtracking:

- In solving some problems, from a given position, there are some available valid paths to go.
- Only one path may be try at a time.
- Others are the backtracking points to try later.
- If one valid path is ended without desired solution, backtracking allows trying through another paths systematically.
- Backtracking is very suitable for problems need to find out all solutions.
- Every time one solution is found, it's saved somewhere, and backtracking allows continuing for the rest.

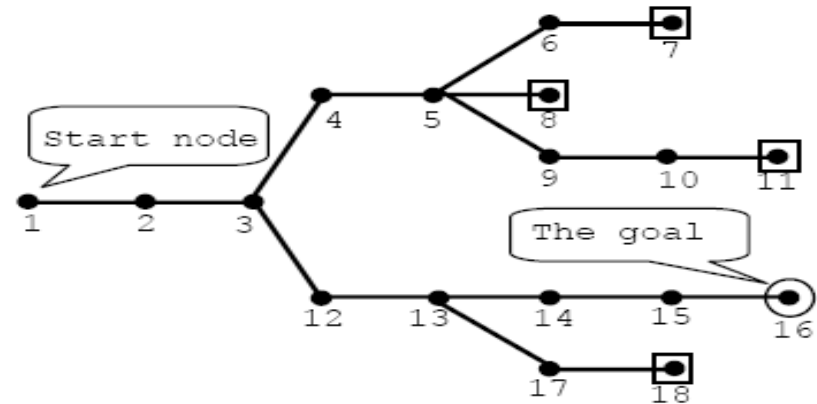
Goal Seeking

➤ Goal seeking problem:

Find the path from the start node to the destination.

➤ Various complexity and extension of goal seeking problem:

- Having only one start node and one destination.
- Having one start node and some destinations.
- Need to determine whether the path exists or not.
- If the path exists, show the nodes in it.
- Need to determine the cost of the path.
- The cost of the path will answer the problem not the specific destinations
- Find only one result if exists.
- Find out all results if exist.
- The graph representing the ways is acyclic or not.
- ...



Goal Seeking (cont.)

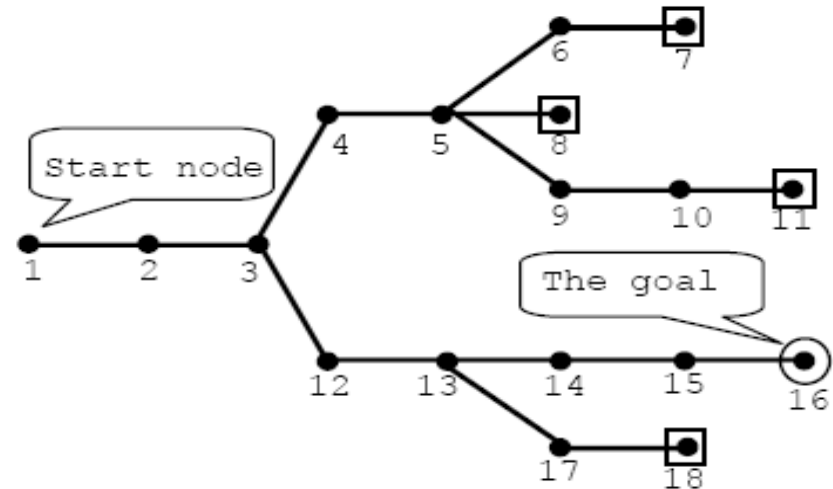
Simplest goal seeking problem:

Acyclic graph has only one start node and one destination.

Determine whether the path from start node to destination exists or not

<ErrorCode> **GoalSeeking1**

```
(val StartNode <NodeType>,  
  val Destination <NodeType>,  
  val Graph <GraphType>)
```



Pre Acyclic **Graph** has **StartNode** and **Destination**.

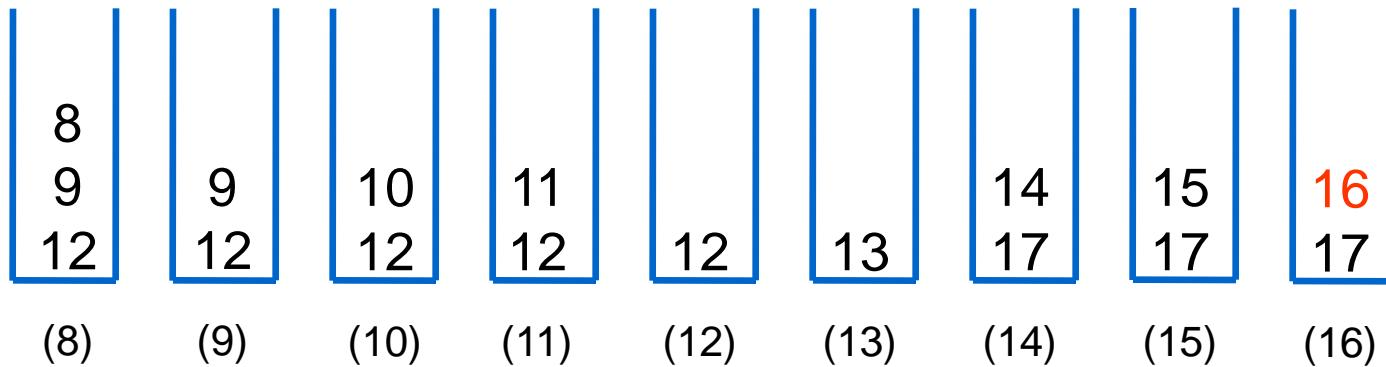
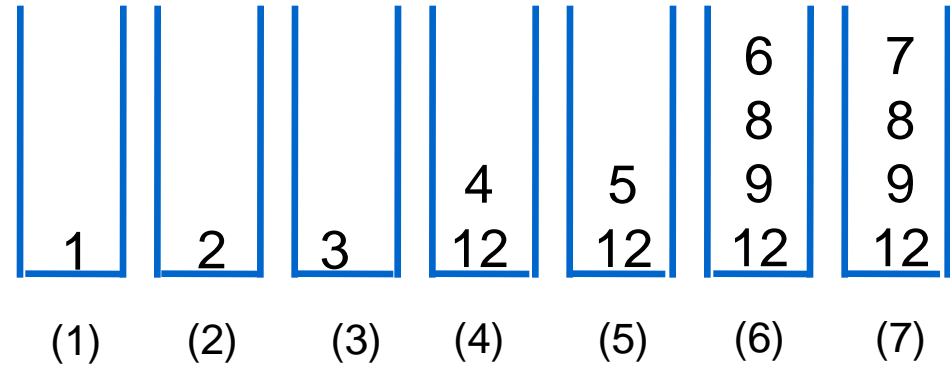
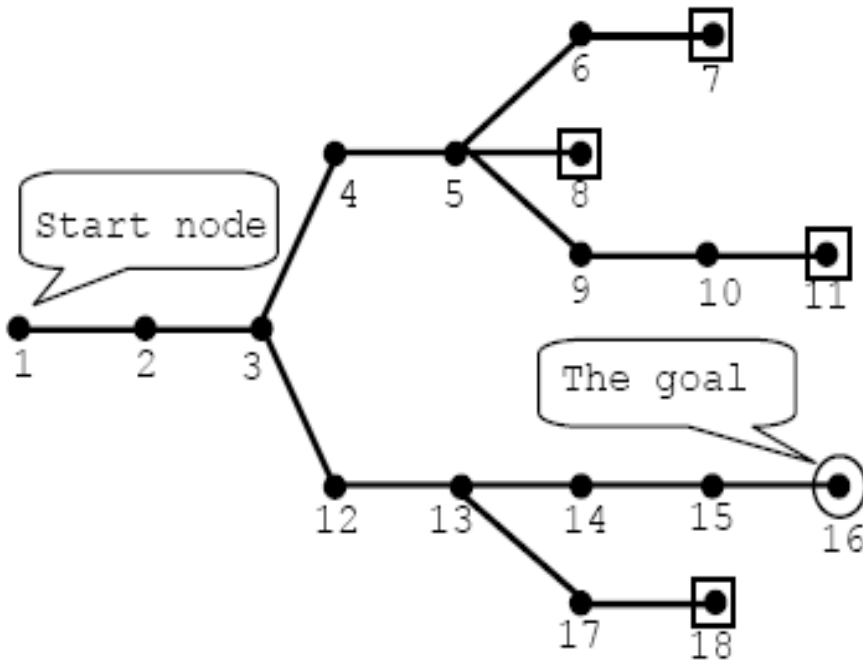
Post Determine whether the path from **StartNode** to **Destination** exists or not.

Return *overflow*, *success* or *failed*.

Uses Stack ADT.

Goal Seeking (cont.)

Algorithm **GoalSeeking1**



Destination
is found,
the path
exists.

Goal Seeking (cont.)

<ErrorCode> **GoalSeeking1** (...)

1. **stackObj** <Stack>
 2. **stackObj.Push**(**StartNode**)
 3. **loop** ((not **stackObj.isEmpty()**) and (**Destination** is not found))
 1. **stackObj.Top**(node)
 2. **stackObj.Pop**()
 3. **if** (node is not **Destination**)
 1. Push into **stackObj** all node's adjacents, if **stackObj** is overflow, return *overflow*.
 4. **if** (**Destination** is found)
 1. return *success*
 5. **else**
 1. return *failed*
- end GoalSeeking1

Goal Seeking (cont.)

Another goal seeking problem:

Acyclic graph has only one start node and one destination. If the path exists, show the nodes in it.

```
<ErrorCode> GoalSeeking2 (val StartNode <NodeType>,  
                             val Destination <NodeType>,  
                             val Graph <GraphType>,  
                             ref ListOfNode <List>)
```

Pre Acyclic graph has StartNode and Destination.

Post If the path from StartNode to Destination exists, ListOfNode contains the nodes in it, otherwise ListOfNode is empty.

Return *overflow*, *success* or *failed*.

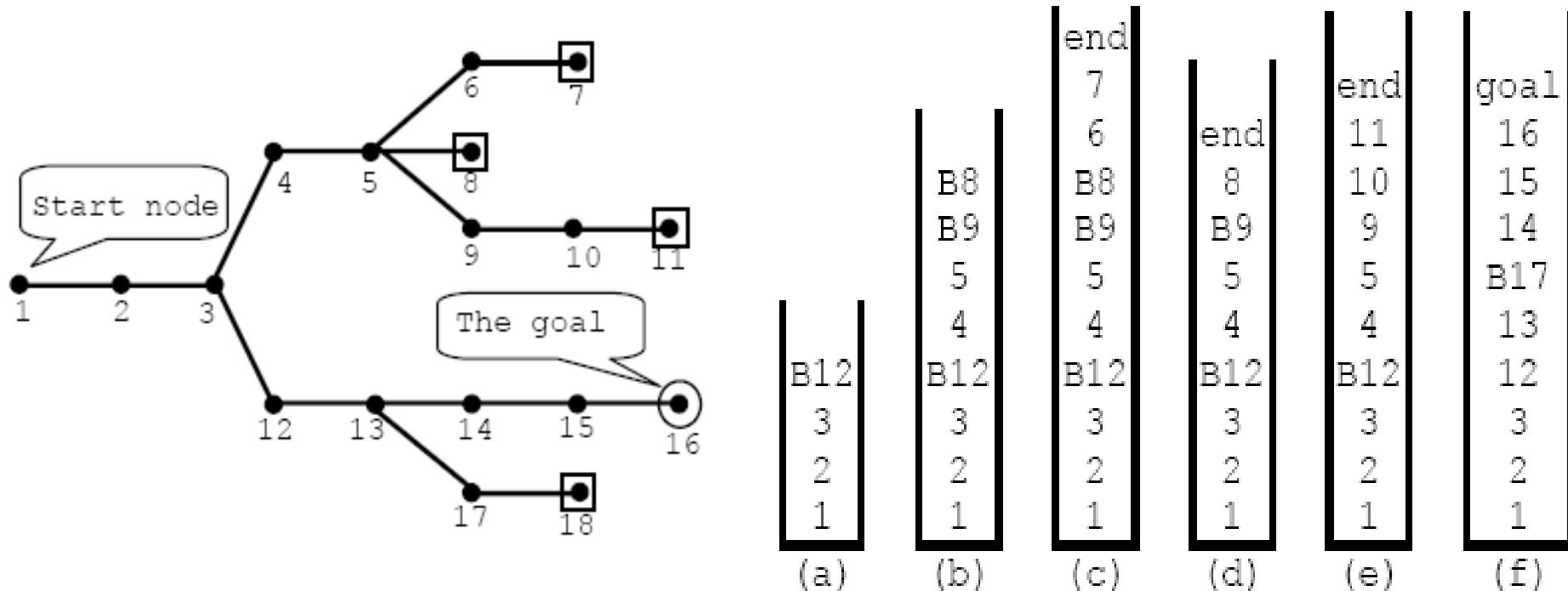
Uses Stack ADT.

Goal Seeking (cont.)

Algorithm **GoalSeeking2**

There are two different types of elements to push into the stack:

- The node in the valid path.
- The backtracking point (with “B” flag).



Goal Seeking (cont.)

<ErrorCode> **GoalSeeking2** (...)

1. **stackObj** <Stack>
 2. **stackObj.Push(StartNode)**
 3. **loop** ((not **stackObj.isEmpty()**) and (**Destination** is not found))
 1. **stackObj.Top(node)**
 2. **if** (node is not **Destination**)
 1. If node having flag “B”
 1. **stackObj.Pop()**
 2. **stackObj.Push(node without the flag “B”)**
 2. If (node has n adjacents) // (n>1)
 1. Push into **stackObj** (n-1) node’s adjacents **with the flag “B” to make the backtracking point.**
 3. Push into **stackObj** the only or the last node’s adjacents **without the flag “B”**
- // If any Push operation is failed as **stackObj** is overflow, return **overflow**.

Goal Seeking (cont.)

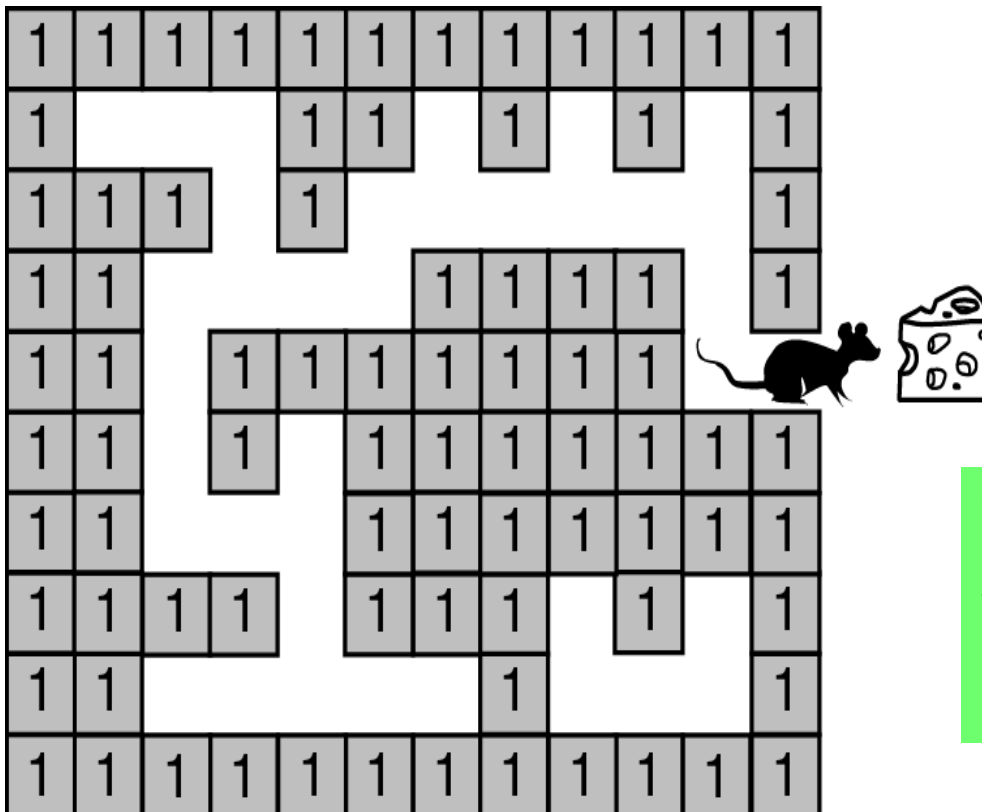
```
<ErrorCode> GoalSeeking2 (...) (cont.)  
4.  if (Destination is found)  
    1.  loop (not stackObj.isEmpty())  
        1.  stackObj.Top(node)  
        2.  stackObj.Pop()  
        3.  if (node without flag "B")  
            1.  ListOfNode.Insert(node, 0) // If Insert operation is failed  
                as ListOfNode is full, return overflow.  
        2.  return success  
5.  else  
    1.  return failed  
end GoalSeeking2
```

Goal Seeking (cont.)

- Tasks depend on each goal seeking problem:
 - Determine what kind of data included in graph (format for nodes and branches, with or without cost), directed or undirected, cyclic or acyclic graph.
 - Determine main goal.
 - Specify input and output.
- Necessary function for all goal seeking problems:
 - Determine all available valid paths from a given position.
- If stack is used in algorithm, determine what kind of data need to be push into the stack which will be used by that function.

Exiting a Maze

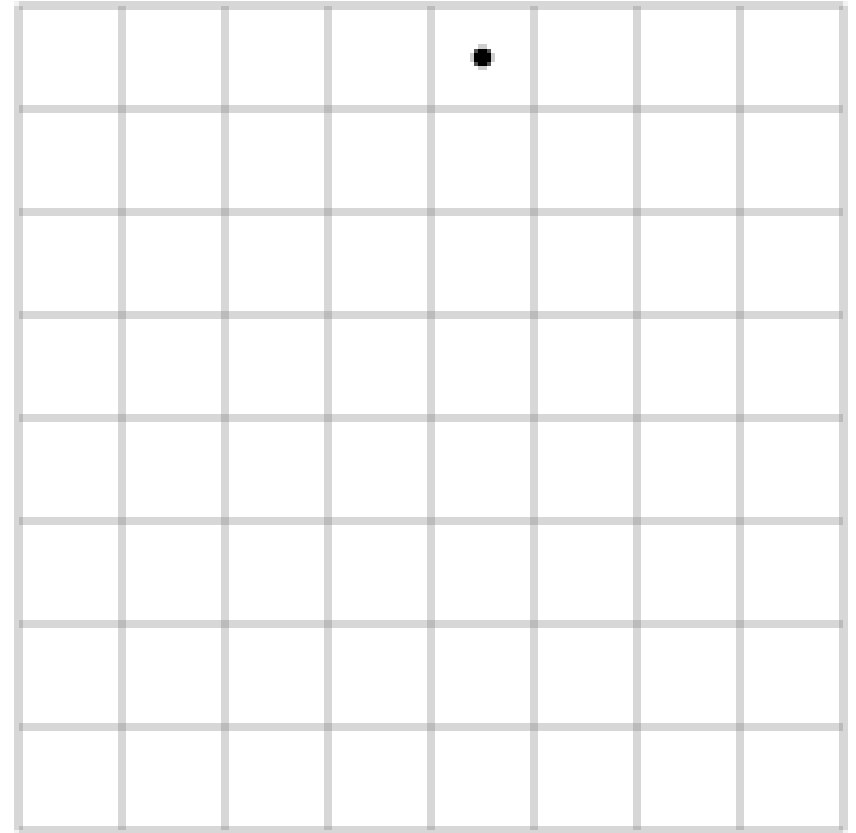
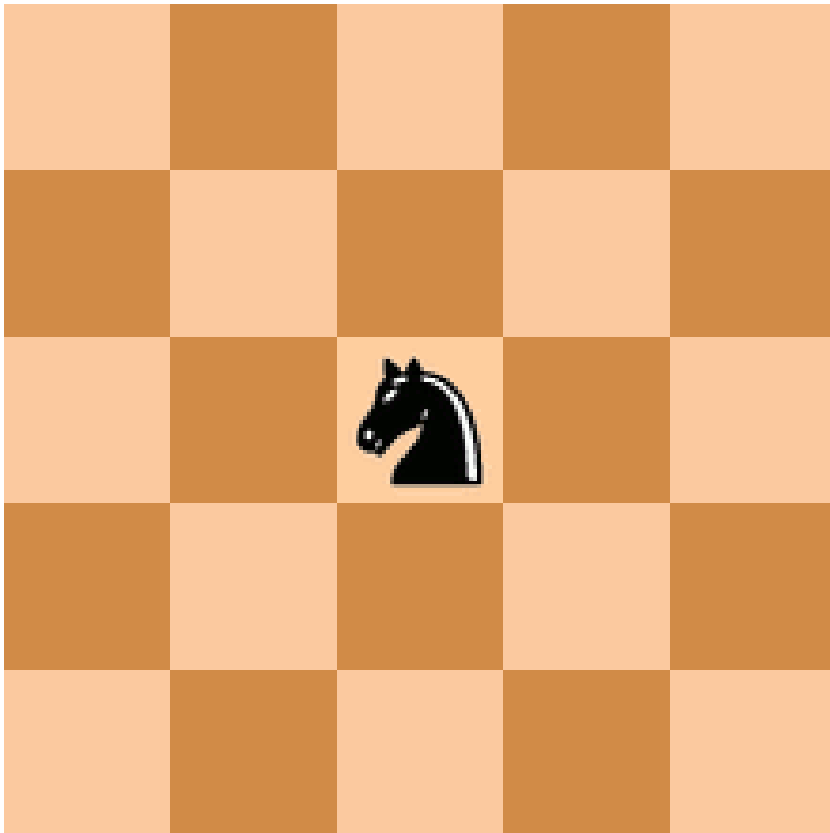
- Graph is cyclic, each node contains co-ordinates of cell, no cost.
- Need to mark for visited cell.
- One or more destination.
- Input is one start cell. Output is any solution or all solutions if exists.



What kind of data, and which data need to be push into the stack?

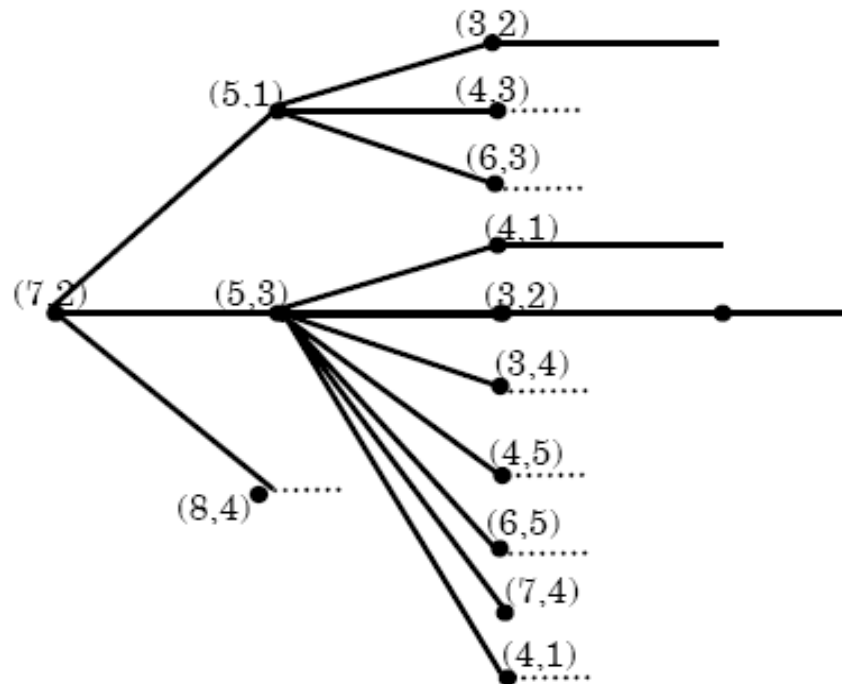
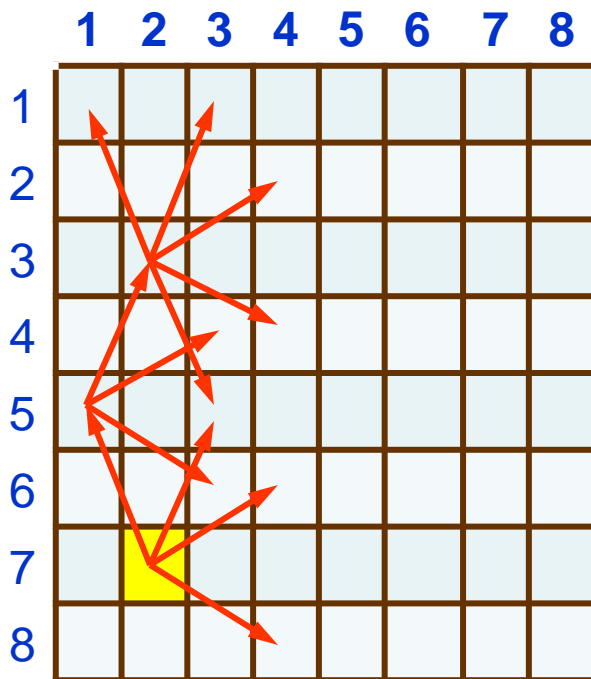
Knight's tour Problem

The knight is placed on the empty board and, moving according to the rules of chess, must visit each square exactly once.



Knight's tour Problem

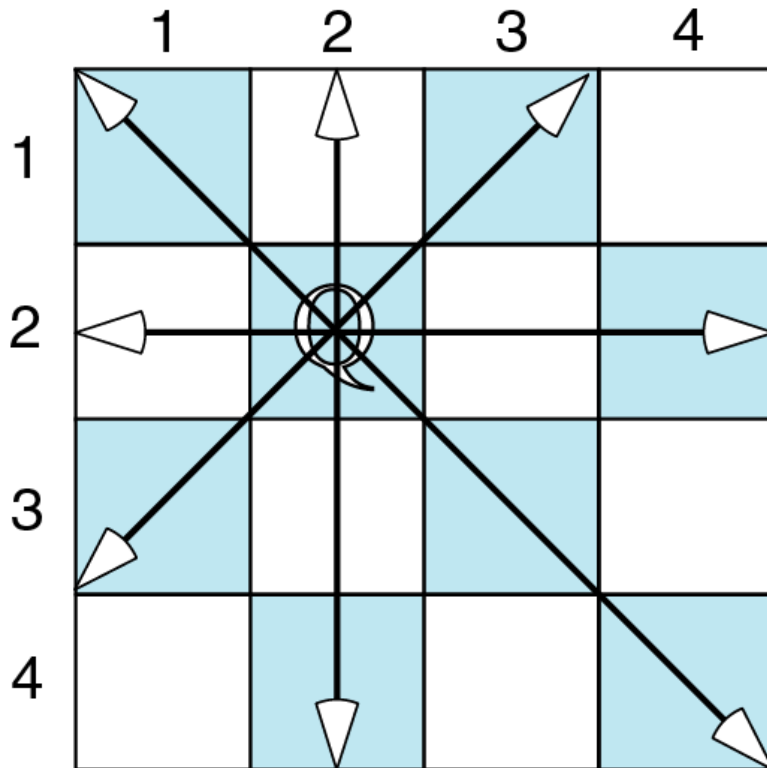
- Graph is cyclic, each node contains co-ordinates of the cell.
- Knight's way is cyclic, need to mark for visited cells.
- Goal is the path having $n*n$ node (n is the size of chess board).
- Output may be any solution or all solutions, if exists.



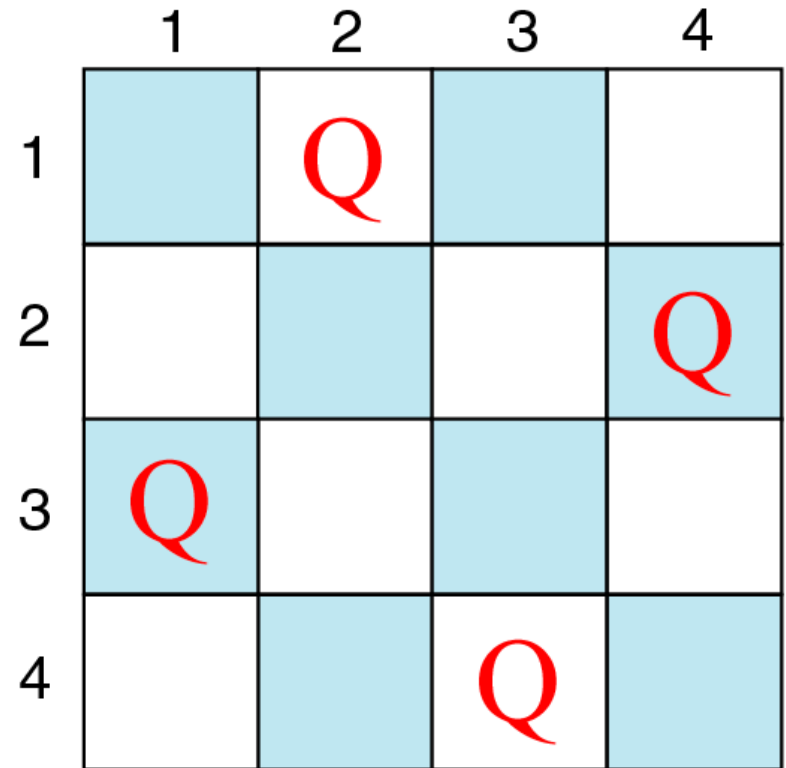
What kind of data, and which data need to be push into the stack?

Queens problem

Determine how to place the Queens on the chessboard so that no queen can take another.



(a) Queen capture rules



(b) First four queens solution 33

Four Queens problem

♔	?	?	?
X	X	♔	?
X	X	X	X

Dead end

(a) (1,1)

♔	?	?	?
X	X	X	♔
X	♔	X	X
X	X	X	X

Dead end

(b)

X	♔	?	?
X	X	X	♔
♔	X	X	X
X	X	♔	X

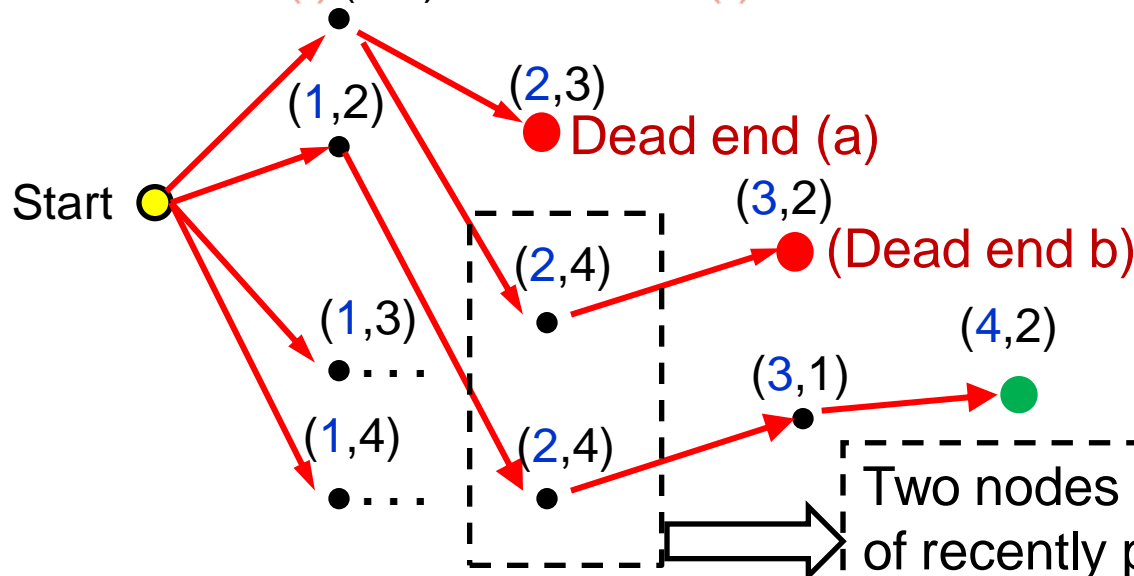
Solution

(c)

		♔	
♔			
			♔
	♔		

Solution

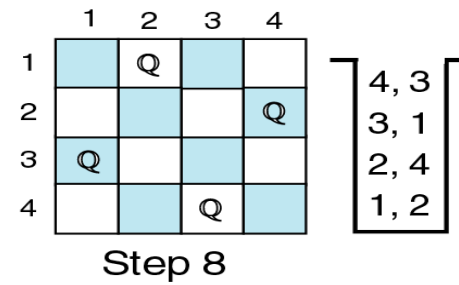
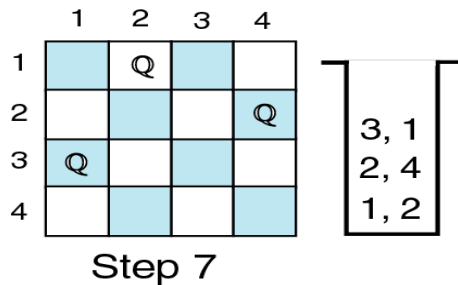
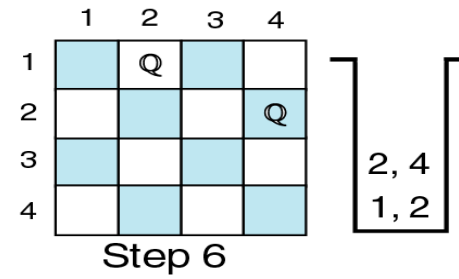
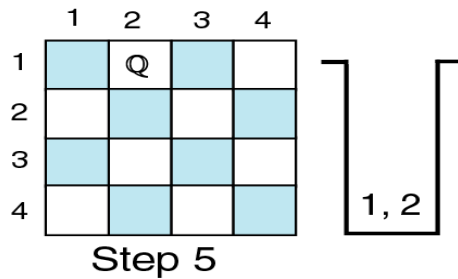
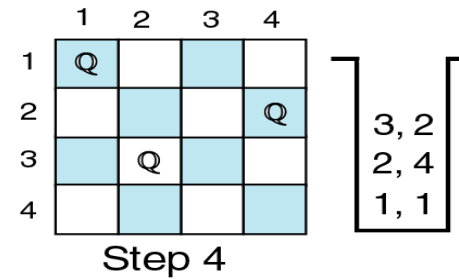
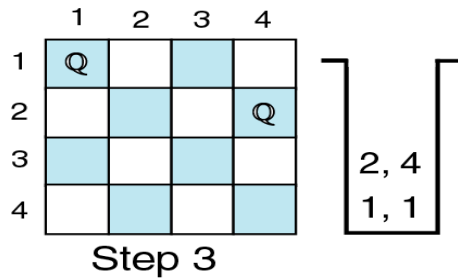
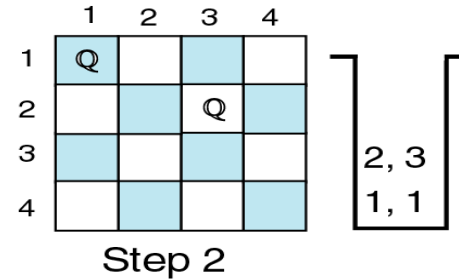
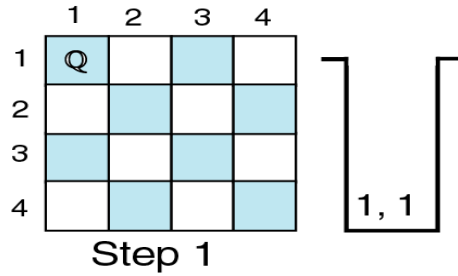
(d)



- Start node.
- End of unsuccessful path.
- One solution is found (path contains 4 nodes).

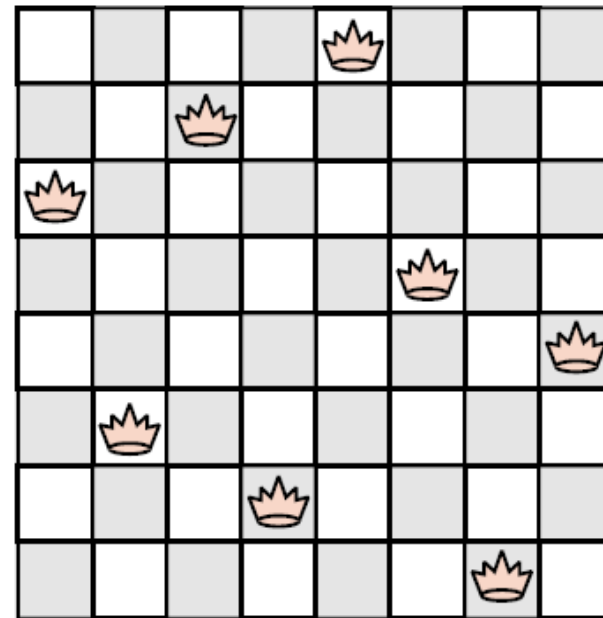
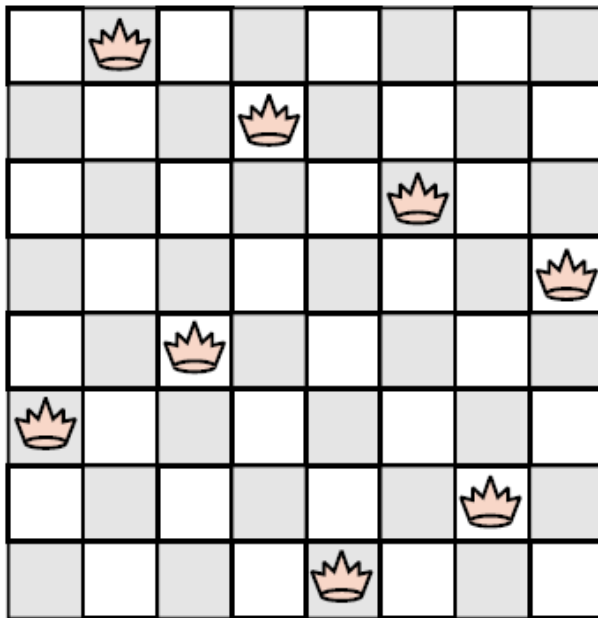
Two nodes contain the same co-ordinates of recently processed Queen, but represent different status of board (figures (b) and (c)).

Four Queens problem (cont.)



Eight Queens problem

- Graph is acyclic, each node contains the current status of the board, not the co-ordinates of the recently processed Queen.
- No specified destination node, goal is the path having n node (n is the size of chess board).
- Output may be any solution or all solutions, if exists.



Two of 92 solutions of Eight Queens Problem

What kind of data, and which data need to be push into the stack?

We will see a lot of interesting problems involved backtracking and usage of Stack ADT while studying recursion, trees, and graphs.