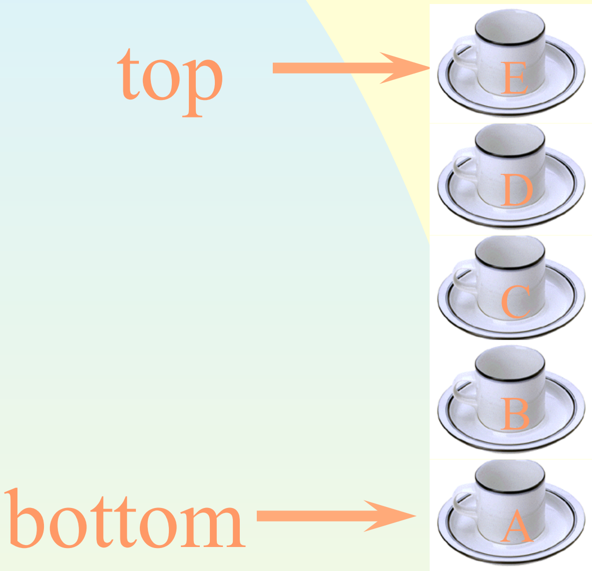
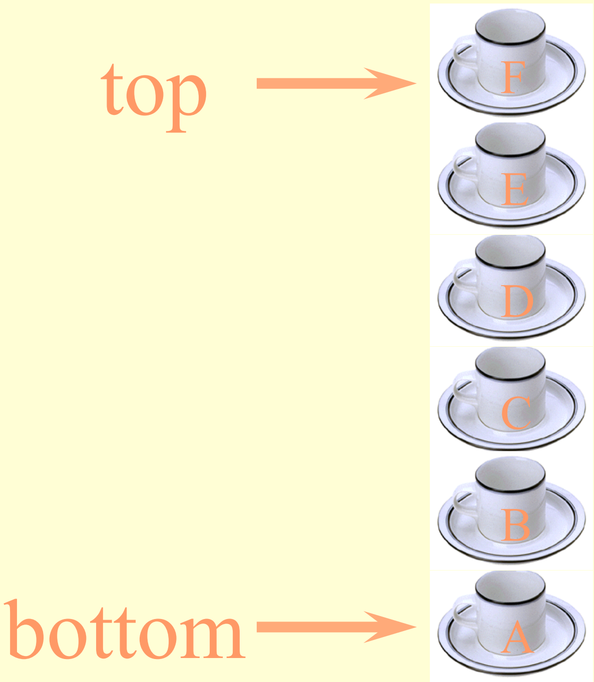
Stacks and Queues

The Stack Abstract Data type

- Linear list.
- One end is called top.
- Other end is called bottom.
- Additions to and removals from the top end only.

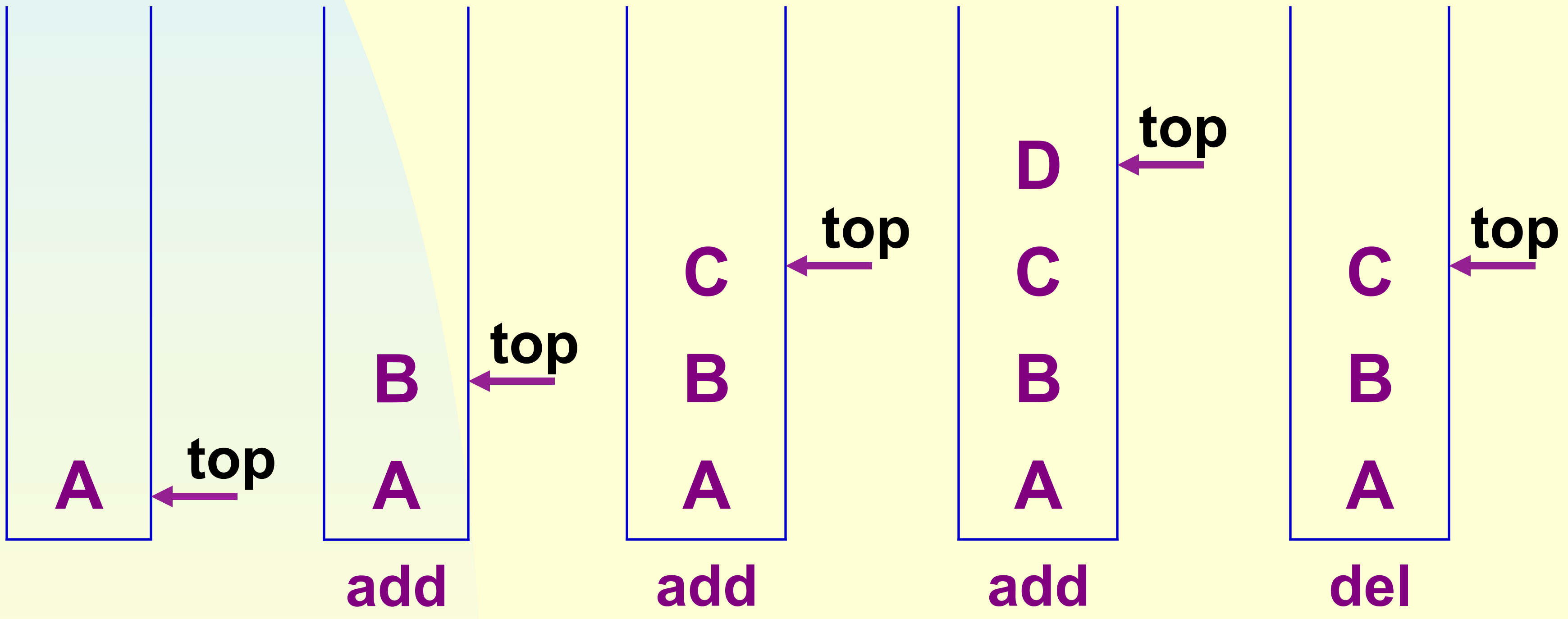
Stack Of Cups





- Add a cup to the stack.
- Remove a cup from new stack.
- A stack is a LIFO list.

Inserting and deleting elements in a stack:



ADT 3.1 Stack

```
template <class T>
class Stack
 { // A finite ordered list with zero or more elements.
 public:
     Stack (int stackCapacity = 10);
    //Creates an empty stack with initial capacity of stackCapacity

     bool IsEmpty() const;
     //If number of elements in the stack is 0, true else false

     T& Top() const;
     // Return the top element of stack

     void Push(const T& item);
     // Insert item into the top of the stack
     void Pop();
     // Delete the top element of the stack.
};
```

To implement STACK ADT, we can use

- **an array**
- **a variable top**

Initially top is set to

–1.

So we have the following data members of Stack:

```
private:
    T* stack;
    int top;
    int capacity;
```

```
template <class T>
Stack<T>::Stack(int stackCapacity): capacity(stackCapacity)
{
   if (capacity < 1) throw "Stack capacity must be > 0";
   stack = new T[capacity];
   top = -1;
}

template <class T>
Inline bool Stack<T>::IsEmpty() const
{
   return(top == -1);
}
```

```
template <class T>
inline T& Stack<T>::Top()
{
   if (IsEmpty) throw "Stack is Empty";
   return stack[top];
}

template <class T>
void Stack<T>::Push(const T& x)
{
   if (top == capacity - 1)
   {
      ChangeSize1D(stack, capacity, 2*capacity);
      capacity *= 2;
    }
    stack[++top] = x;
}
```

The template function ChangeSize1D changes the size of a 1-Dimensional array of type T from oldSize to newSize:

```
template <class T>
void ChangeSize1D(T* a, const int oldSize, const int newSize)
{
   if (newSize < 0) throw "New length must be >= 0";
   T* temp = new T[newSize];
   int number = min(oldSize, newSize);
   copy(a, a + number, temp);
   delete [] a;
   a = temp;
}
```

```
template <class T>
void Stack<T>::Pop()
{ // Delete top element of stack.
    if (IsEmpty()) throw "Stack is empty, cannot delete.";
    stack[top--].~T();  // destructor for T
}
```

Exercises: P138-1, 2

Bus Stop Queue

Bus
Stop

front rear rear rear rear

Bus Stop Queue

Bus
Stop

front rear

Bus Stop Queue

Bus
Stop

front rear

Bus Stop Queue

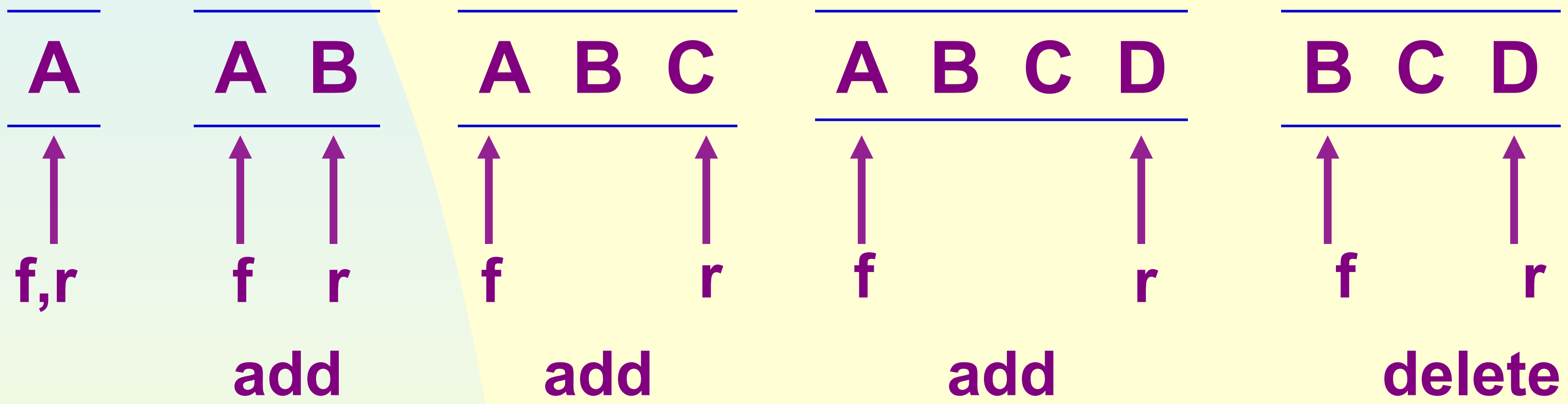
Bus
Stop

front rear rear

3.3 The Queue Abstract Data Type

- Linear list.
- One end is called front.
- Other end is called rear.
- Additions are done at the rear only.
- Removals are made from the front only.

3.3 The Queue Abstract Data Type



f = queue front r = queue rear

ADT 3.2 Queue

```
template <class T>
class Queue
 { // A finite ordered list with zero or more elements.
 public:
    Queue (int queueCapacity = 10);
    // Creates an empty queue with initial capacity of
    // queueCapacity

    bool IsEmpty() const;

    T& Front() const; //Return the front element of the queue.

    T& Rear() const; //Return the rear element of the queue.

    void Push(const T& item);
     //Insert item at the rear of the queue.
    void Pop();
    // Delete the front element of the queue.
};
```

To implement this QUEUE ADT, we can use

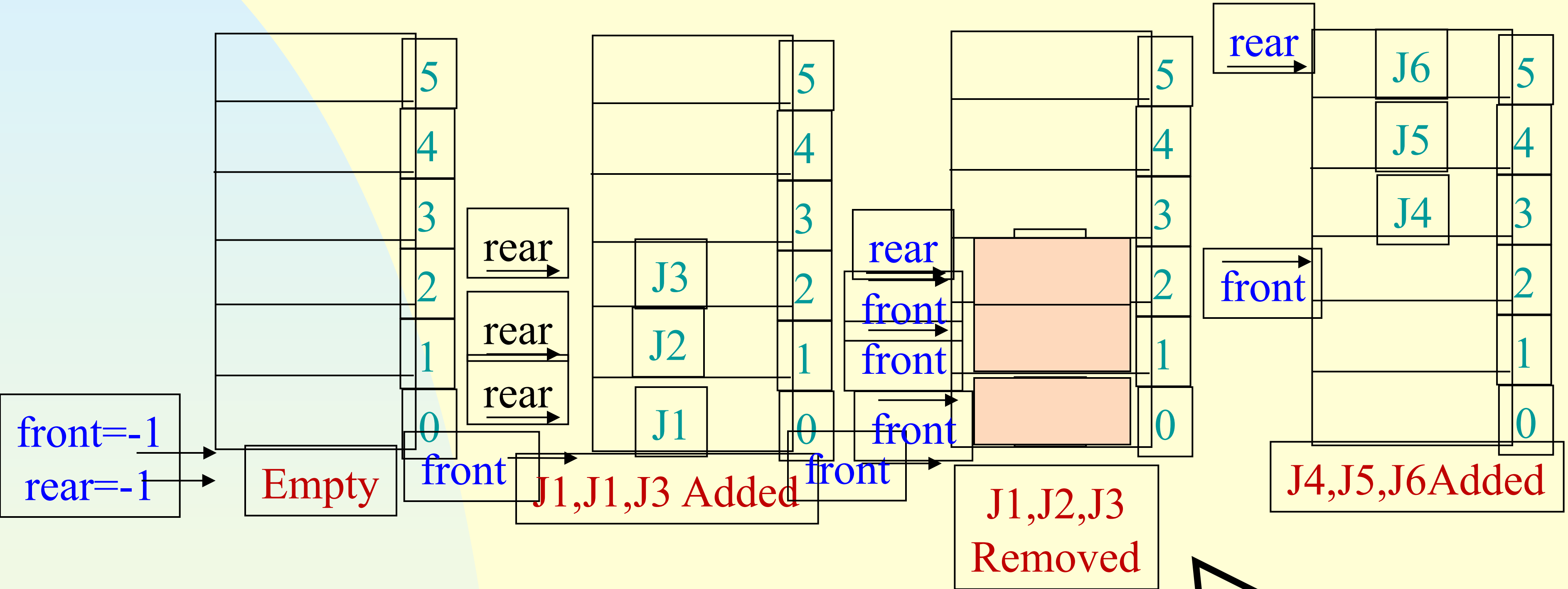
an array

two variable front and rear

front being one less than the position of the first element

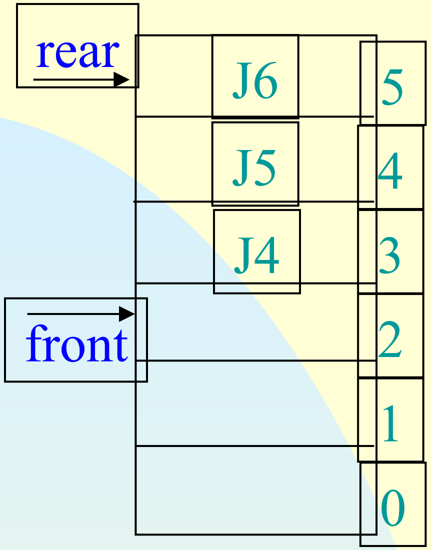
So we have the following data members of Queue:

```
private:
    T* queue;
    int front,
        rear,
        capacity;
```



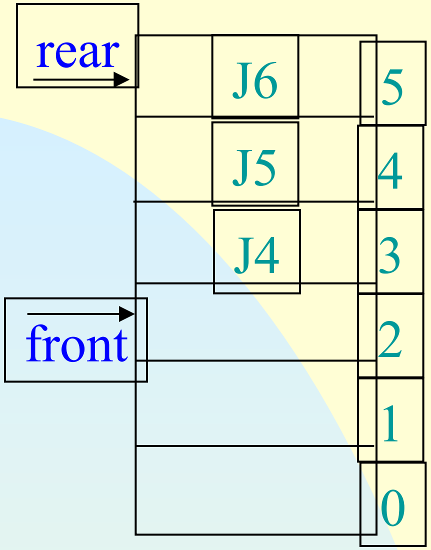
front=rear=-1

Empty： front==rear
EnQ： sq[++rear]=x;
DeQ： x=sq[++front];



Problem

- EnQueue: Add an element
  - Overflow!
  - Space Available! →
    - False Overflow
- Solution?
  - Elements movement
  - ?



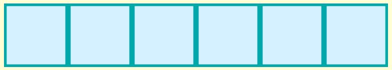
Problem

- False Overflow
- Solution?
  - 6 → 2
  - 6 → 1
  - 6 → 0
- Capacity = 6
- 6 % 6 = 0

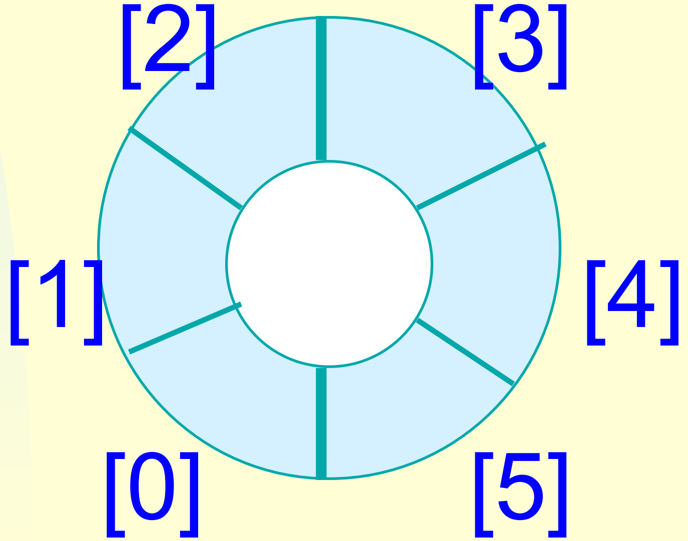
Array Queue

- Use a 1D array queue.

queue[]

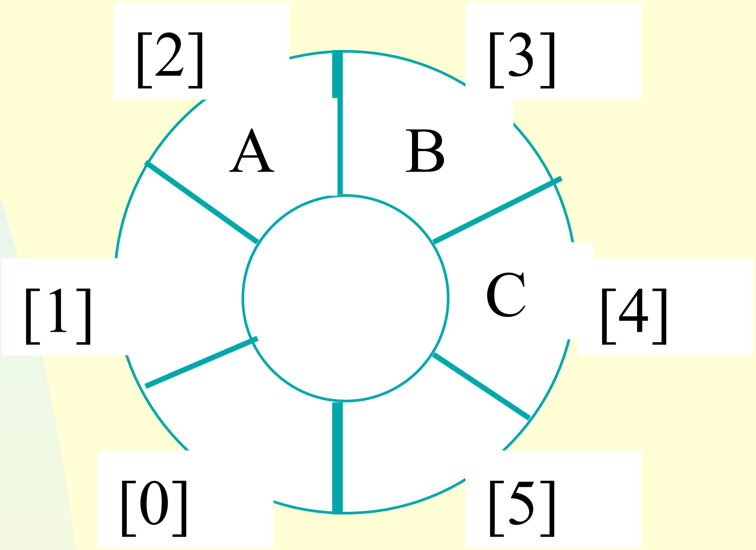


- Circular view of array.



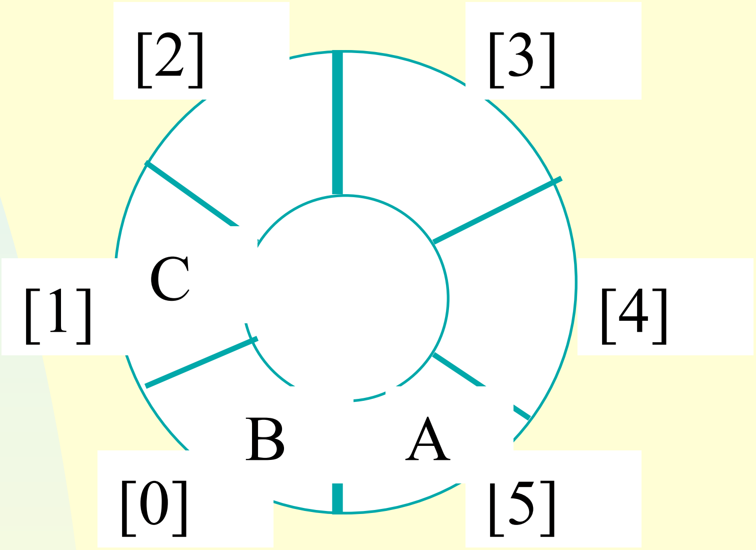
Array Queue

- Possible configuration with 3 elements.



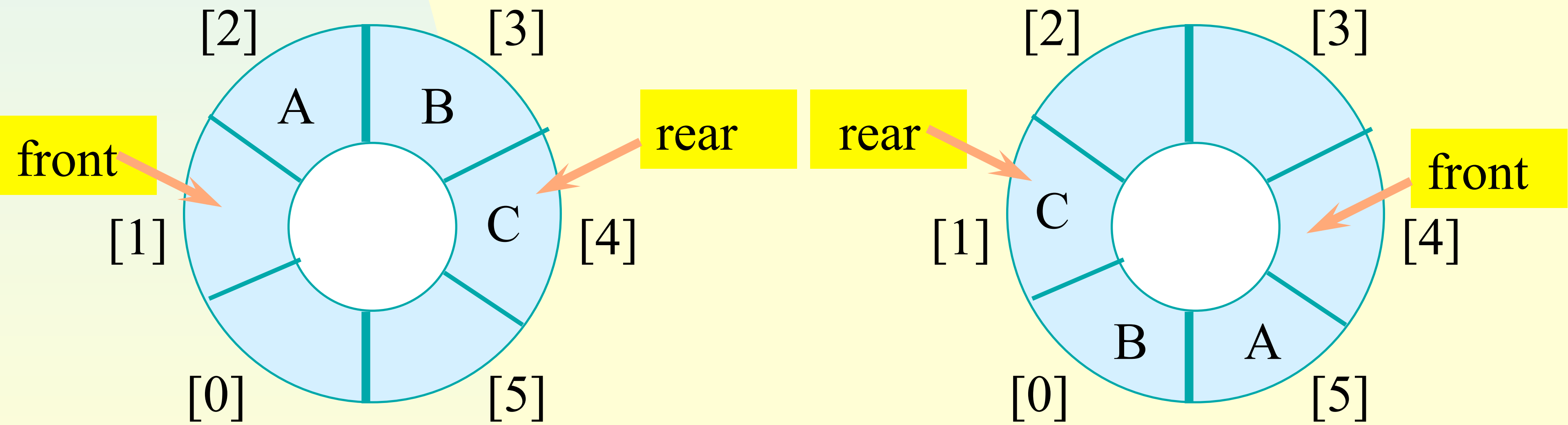
Array Queue

- Another possible configuration with 3 elements.



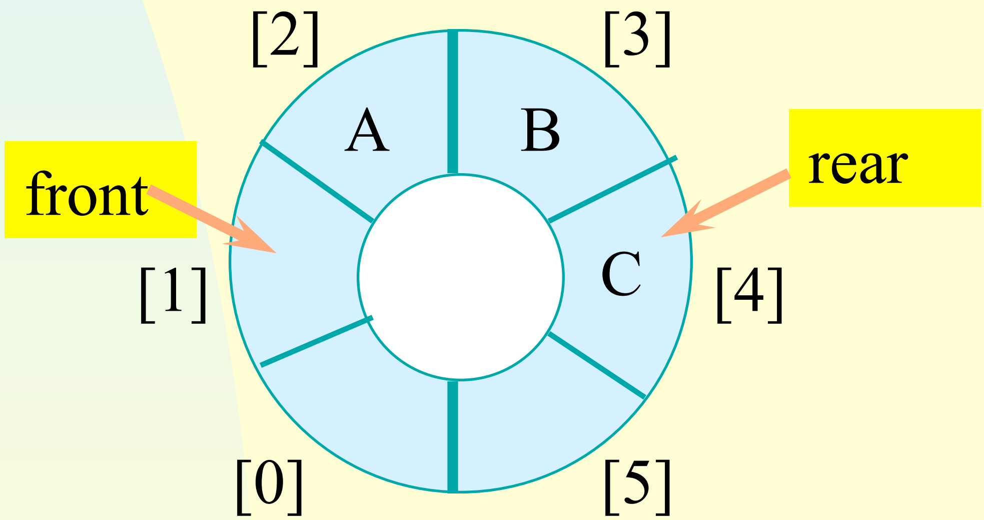
Array Queue

- Use integer variables front and rear.
  - front is one position counterclockwise from first element
  - rear gives position of last element



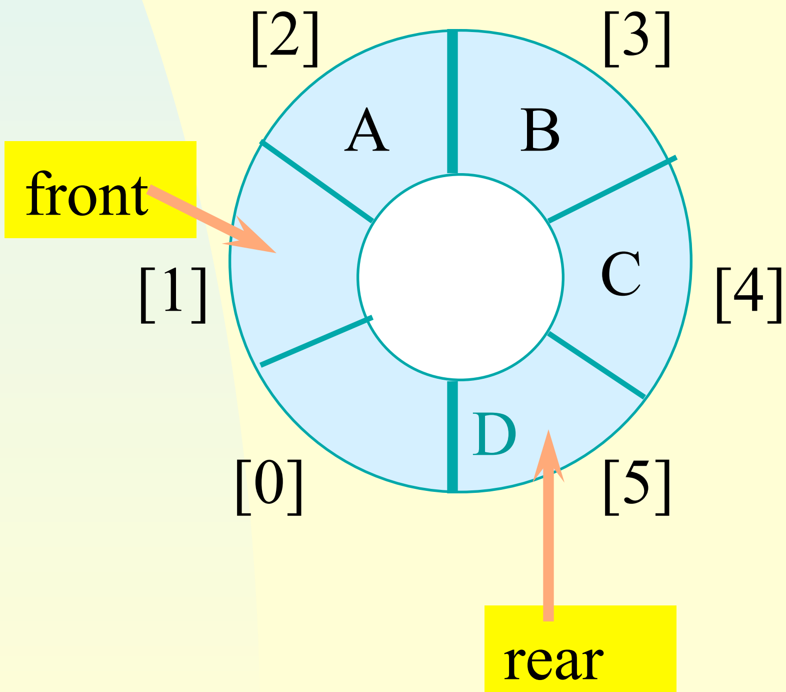
Add An Element

- Move rear one clockwise.



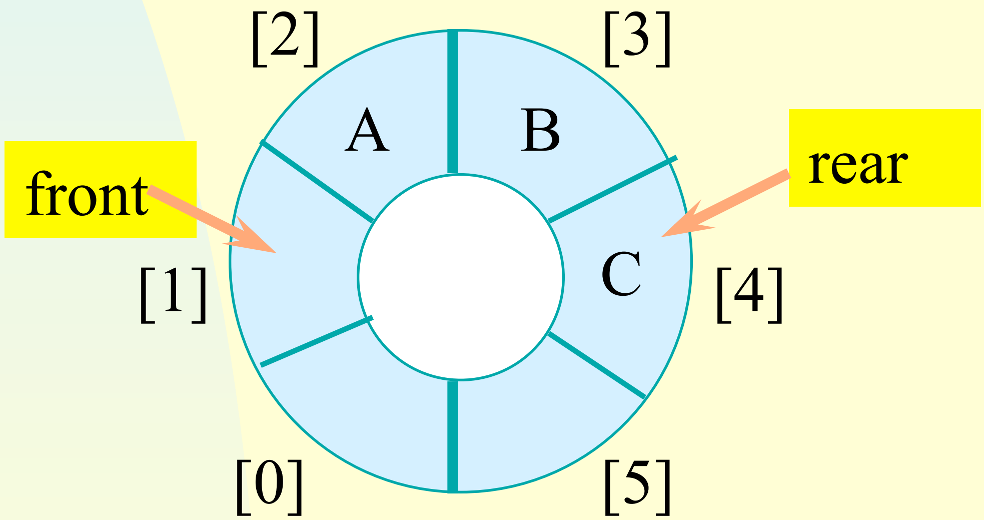
Add An Element

- Move rear one clockwise.
- Then put into queue[rear].



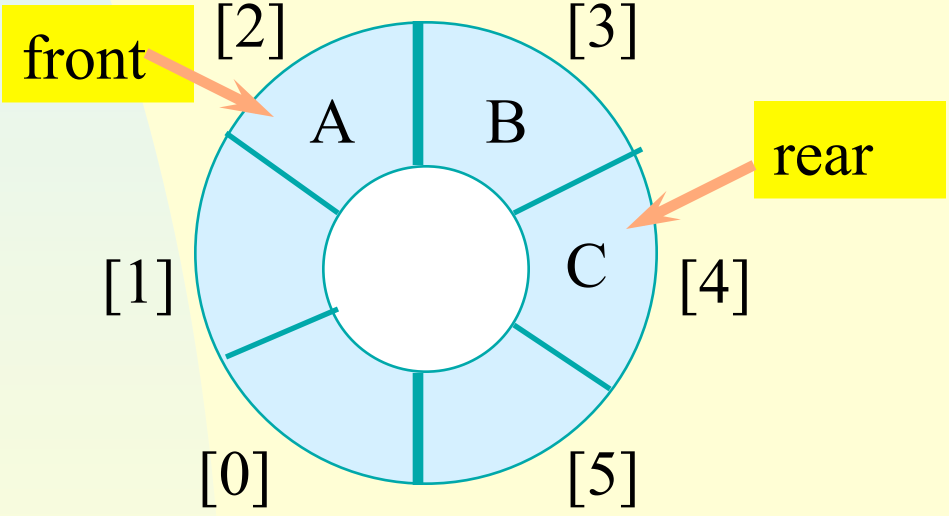
Remove An Element

- Move front one clockwise.



Remove An Element

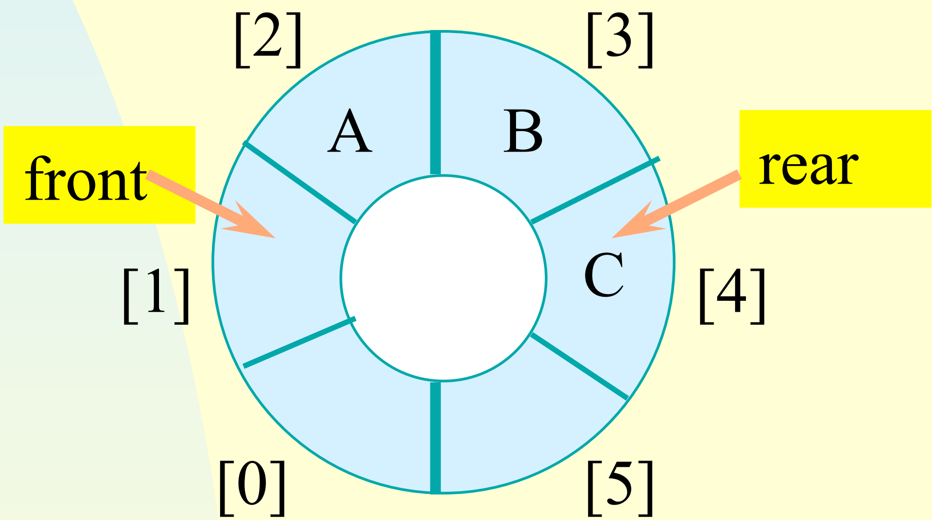
- Move front one clockwise.
- Then extract from queue[front].



Moving rear Clockwise

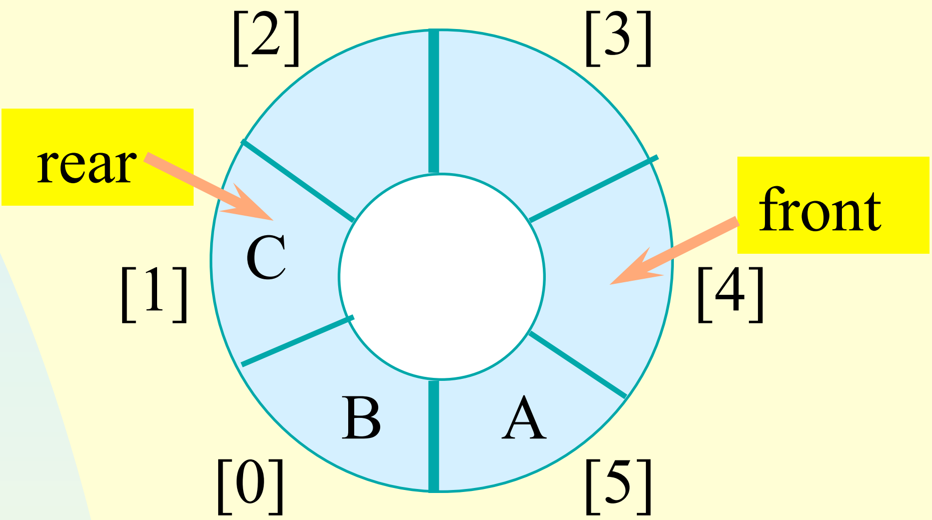
- rear++;

  if (rear = = queue.length) rear = 0;

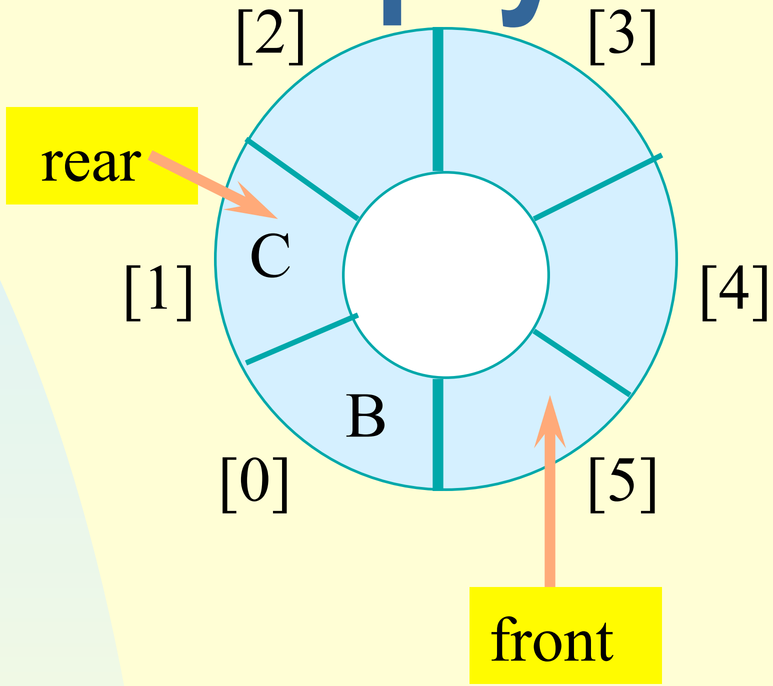


- rear = (rear + 1) % queue.length;

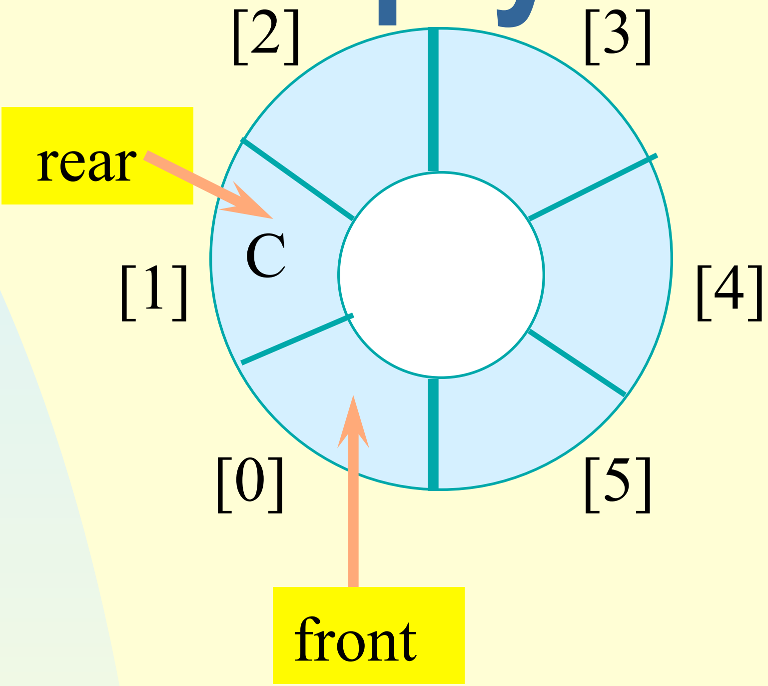
Empty That Queue



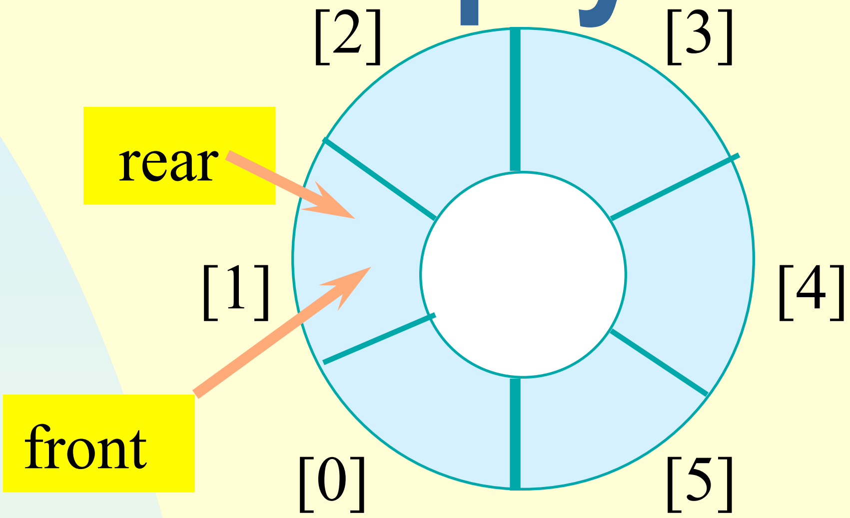
Empty That Queue



Empty That Queue

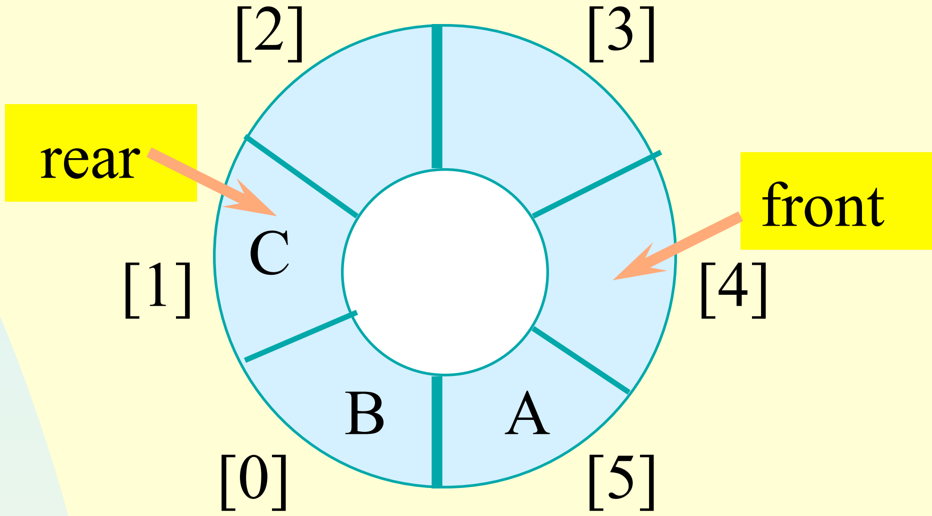


Empty That Queue

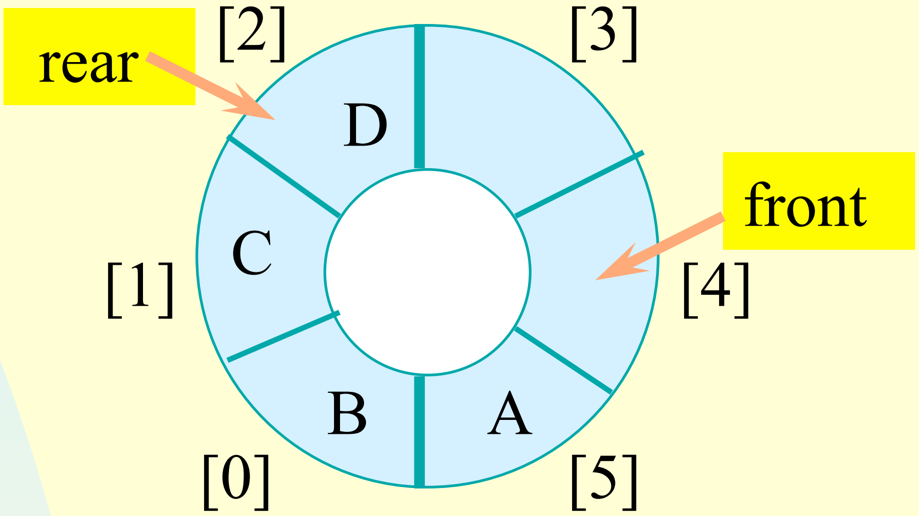


- When a series of removes causes the queue to become empty, front = rear.
- When a queue is constructed, it is empty.
- So initialize front = rear = 0.

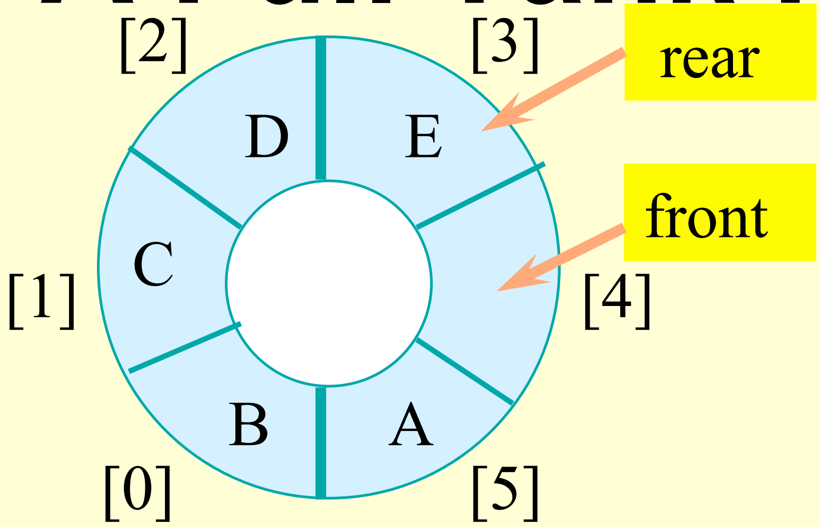
A Full Tank Please



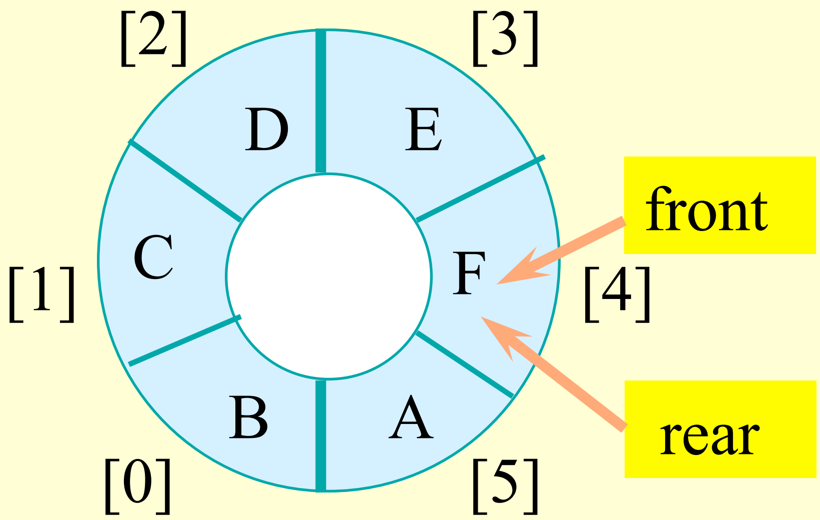
A Full Tank Please



A Full Tank Please



A Full Tank Please



- When a series of adds causes the queue to become full, front = rear.
- So we cannot distinguish between a full queue and an empty queue!

Ouch!!!!!

- Remedies.
  - Don't let the queue get full.
    - When the addition of an element will cause the queue to be full, increase array size.
    - This is what the text does.
  - Define a boolean variable lastOperationIsPut.
    - Following each put set this variable to true.
    - Following each remove set to false.
    - Queue is empty iff (front == rear) && ! lastOperationIsPut
    - Queue is full iff (front == rear) && lastOperationIsPut

Ouch!!!!!

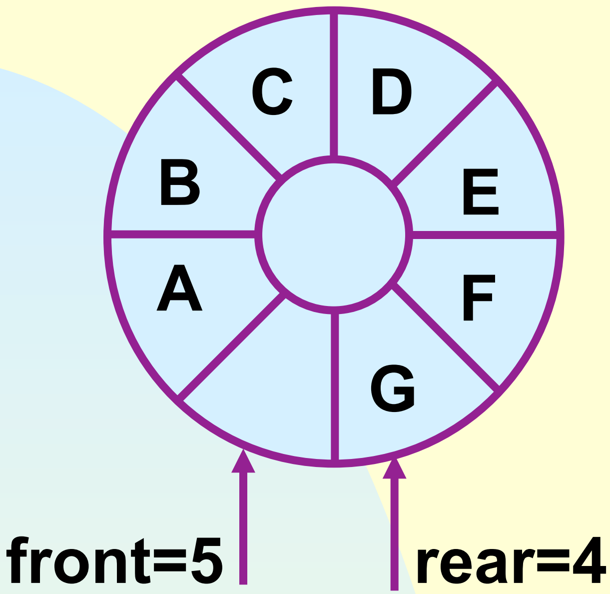
- Remedies (continued).
  - Define an integer variable size.
    - Following each put do size++.
    - Following each remove do size--.
    - Queue is empty iff (size == 0)
    - Queue is full iff (size == queue.length)

```
template <class T>
Queue<Type>::Queue(int queueCapacity):
                                        capacity(queueCapacity)
{
   if (capacity < 1) throw "Queue capacity must > 0";
   queue = new T[capacity];
   front = rear = 0;
}
```

```
template <class T>
Inline bool Queue<T>::IsEmpty()
{ return front==rear };

template <class T>
inline T& Queue<T>::Front()
{
    if (IsEmpty()) throw “Queue is empty. No front element”;
    return queue[(front+1)%capacity];
}

template <class T>
inline T& Queue<T>::Rear()
{
    if (IsEmpty()) throw “Queue is empty. No rear element”;
    return queue[rear];
}
```

```
template <class T>
void Queue<T>::Push(const T& x)
{ // add x at rear of queue
   if ((rear+1)%capacity == front)
   { // queue full, double capacity
     // code to double queue capacity comes here
   }
   rear = (rear+1)%capacity;
   queue[rear] = x;
}
```

We can double the capacity of queue in the way as shown in the next slide:



queue	0	1	2	3	4	5	6	7
	C	D	E	F	G		A	B

front=5, rear=4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	F	G									

front=15, rear=6

This configuration may be obtained as follows:

(1)Create a new array newQueue of twice the capacity.

(2)Copy the second segment to positions in newQueue beginning at 0.

(3)Copy the first segment to positions in newQueue beginning at capacity-front-1.

The code is in the next slide:

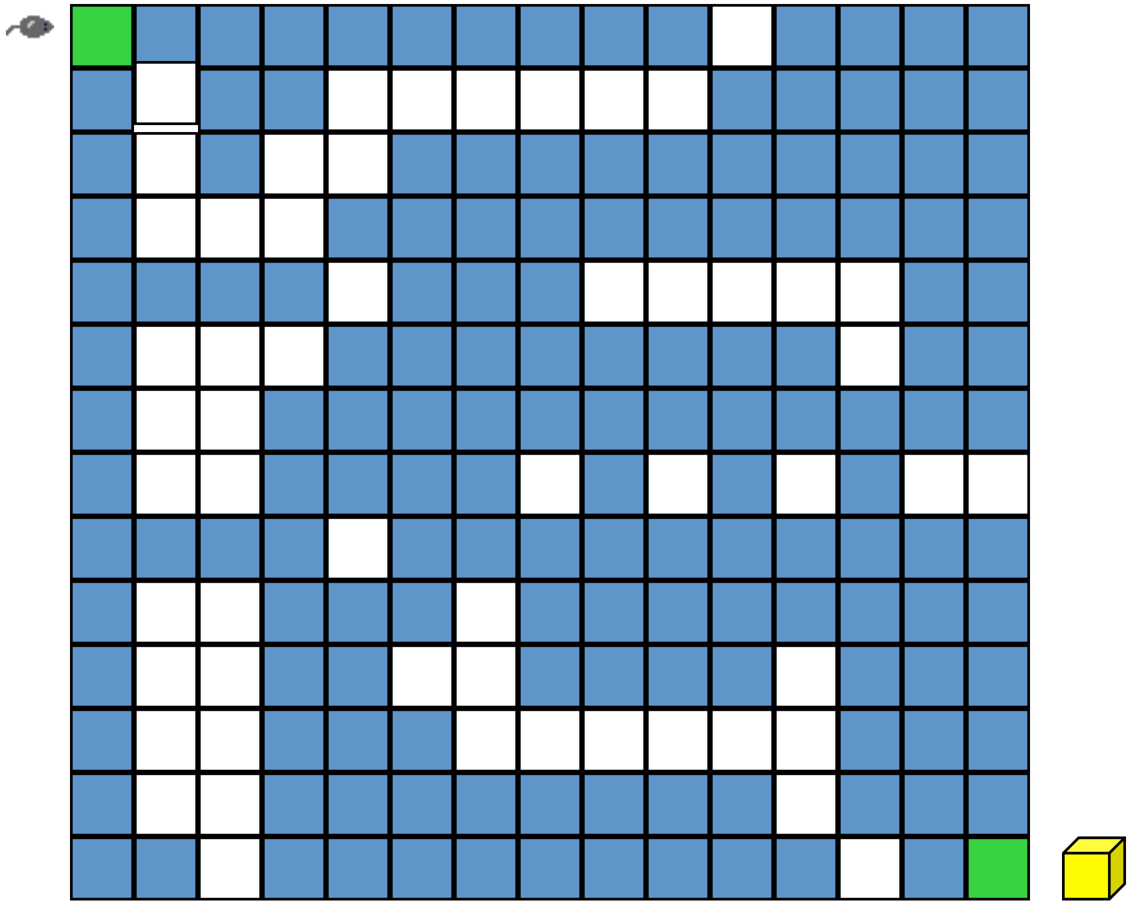
```
// allocate an array with twice the capacity
T*  newQueue = new T[2*capacity];

// copy from queue to newQueue
int start = (front+1)%capacity;
if (start < 2)
    // no wrap around
    copy(queue+start, queue+start+capacity-1, newQueue);
else
{ // queue wraps around
    copy(queue+start, queue+capacity, newQueue);
    copy(queue, queue+rear+1, newQueue+capacity-start);
}

// switch to newQueue
front = 2*capacity-1;  rear =  capacity-2; capacity *= 2;
delete [] queue;
queue = newQueue;
```

```
template <class T>
void Queue<T>::Pop()
{  // Delete front elemnet from queue
   if (IsEmpty()) throw "Queue is empty. Cannot delete";
   front = (front+1)%capacity;
   queue[front].~T;
}
```

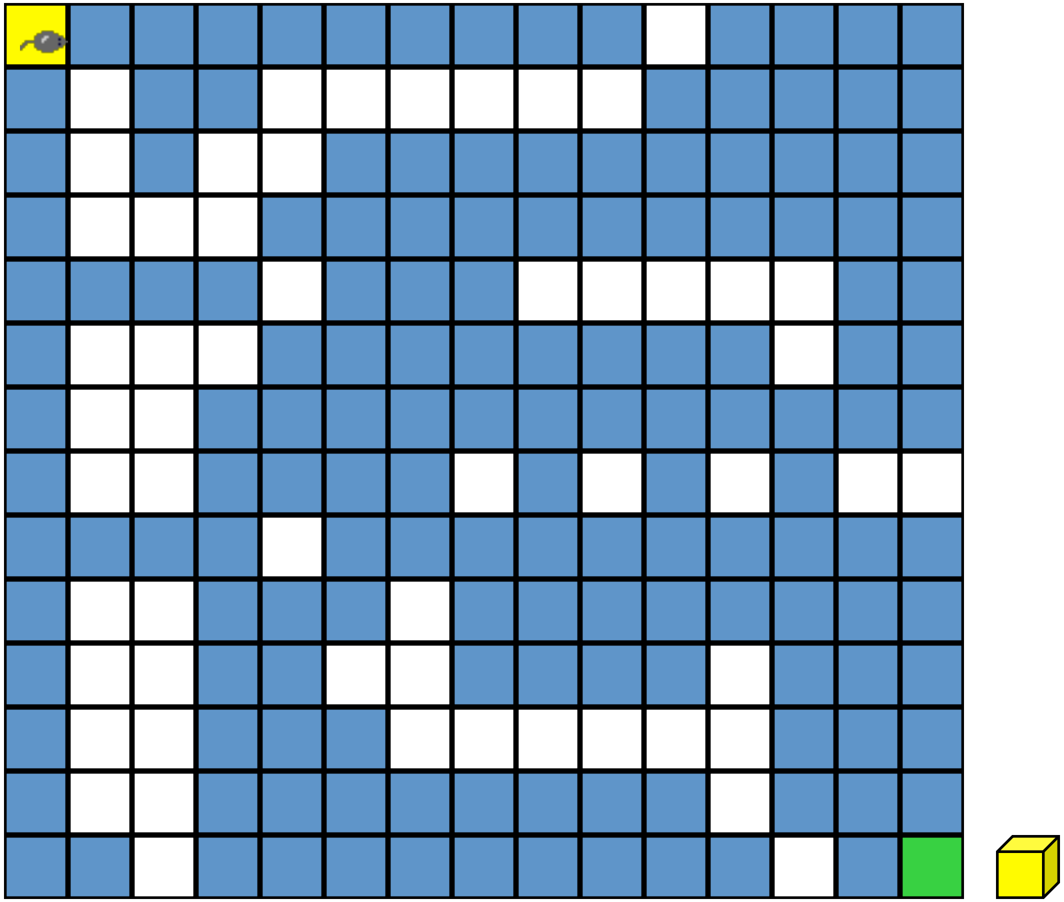
For the circular representation, the worst-case add and delete times (assuming no array resizing is needed) are O(1).

Exercises: **P147-1, 3.**

Rat In A Maze

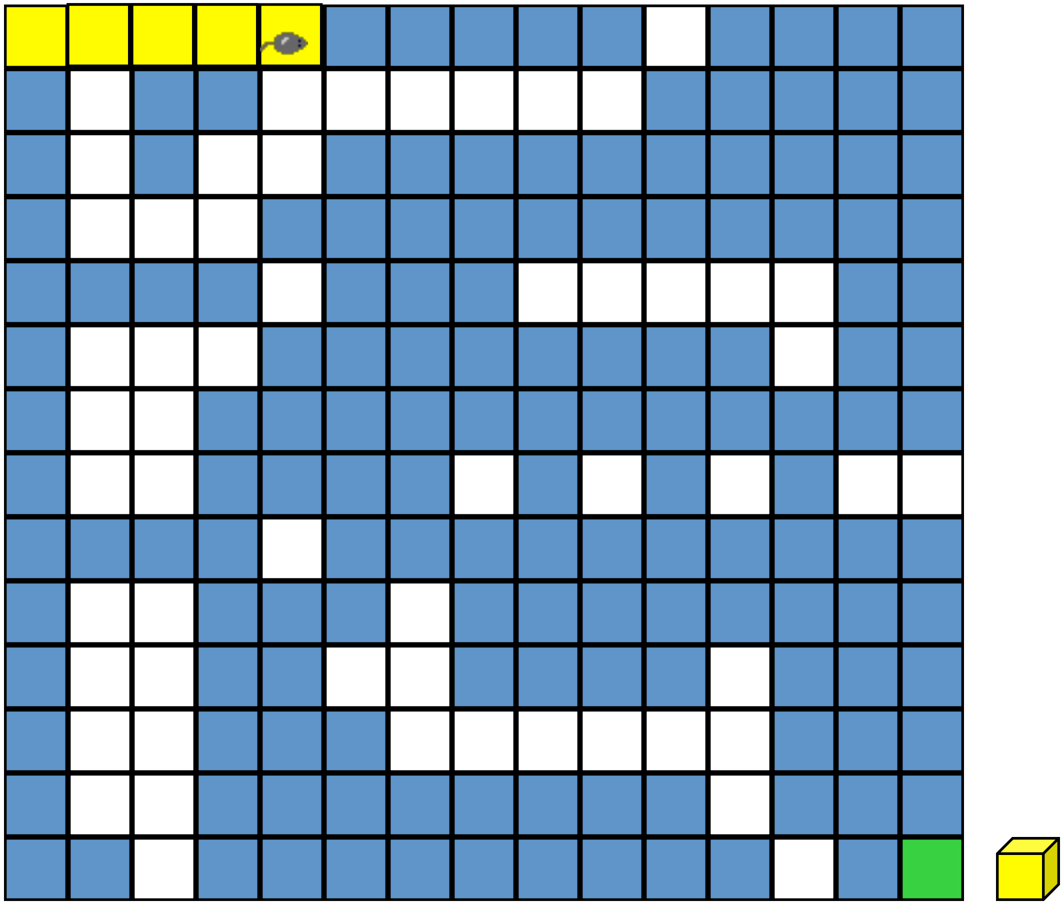


Rat In A Maze



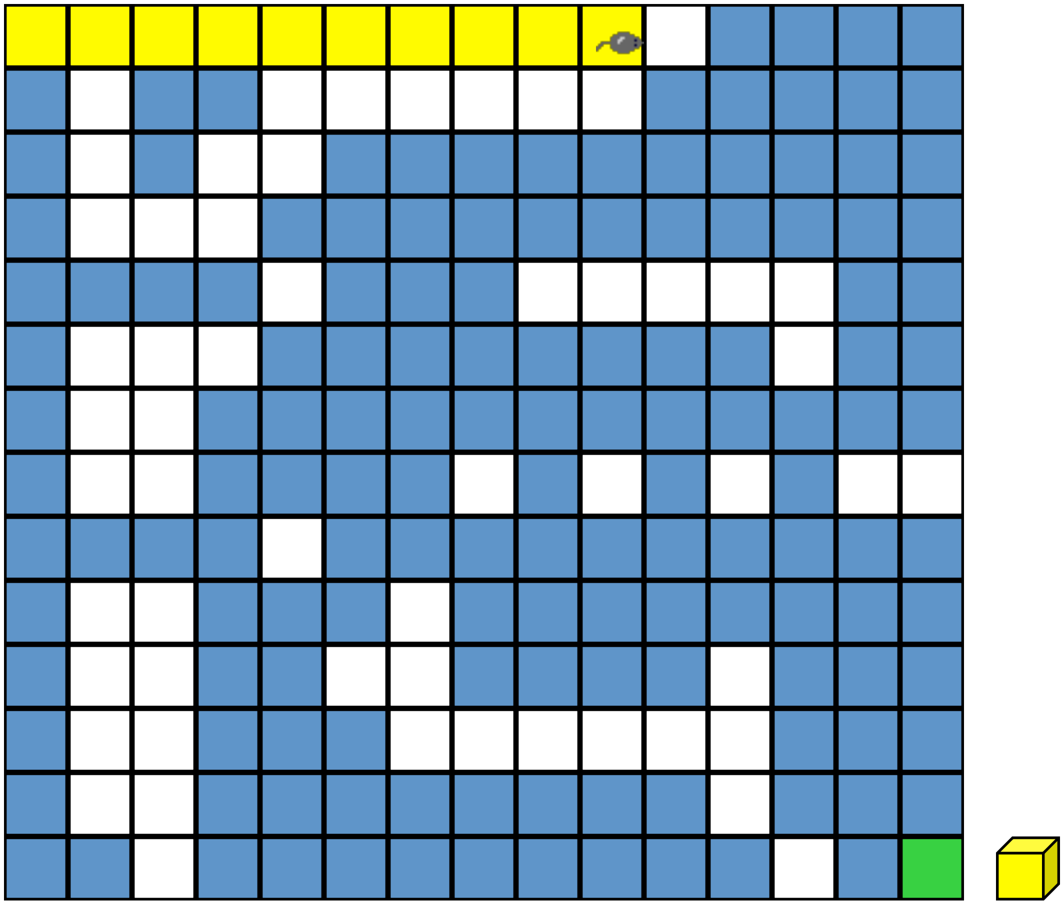
- Move order is: right, down, left, up
- Block positions to avoid revisit.

Rat In A Maze



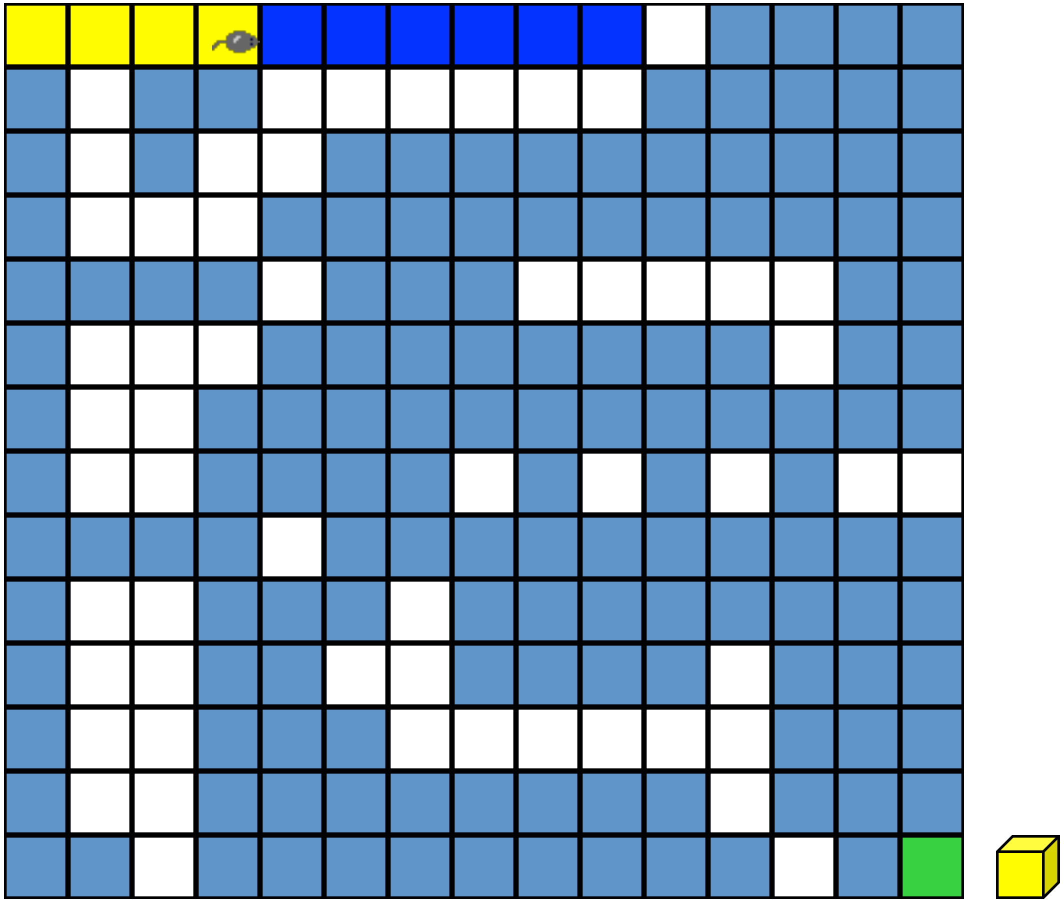
- Move order is: right, down, left, up
- Block positions to avoid revisit.

Rat In A Maze



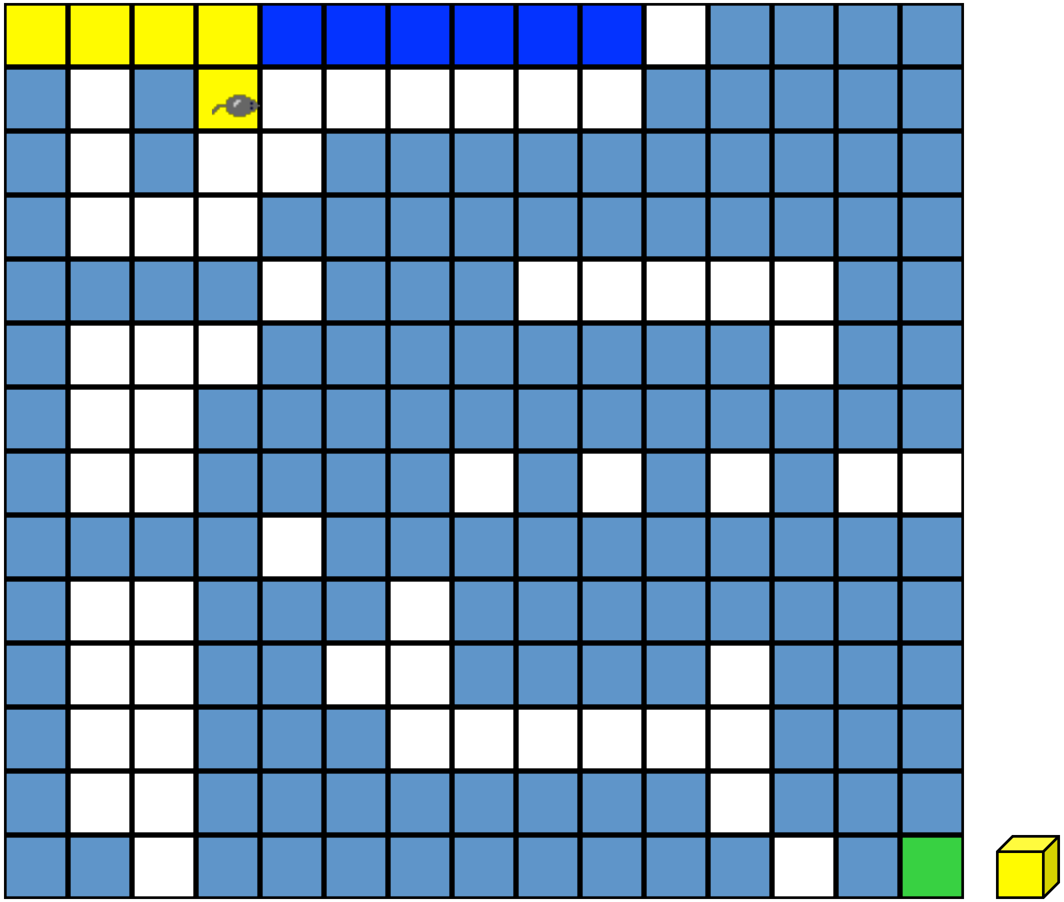
- Move backward until we reach a square from which a forward move is possible.

Rat In A Maze



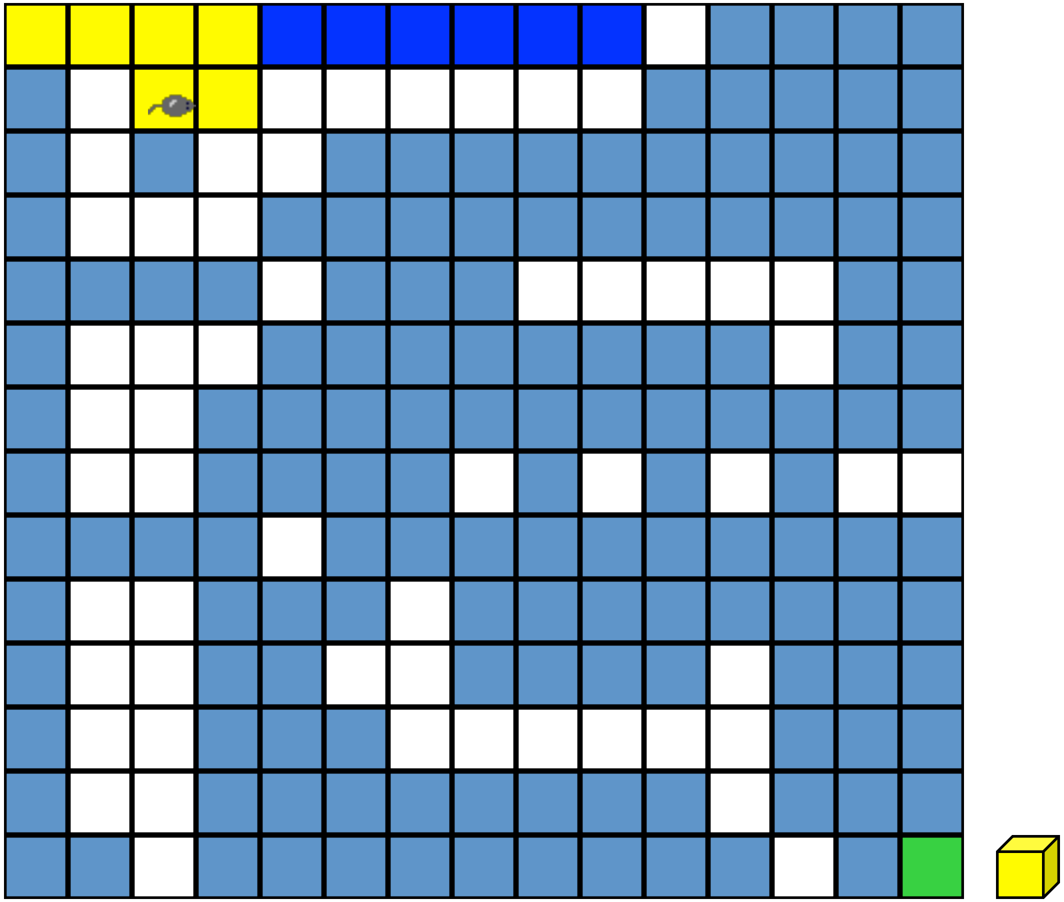
- Move down.

Rat In A Maze



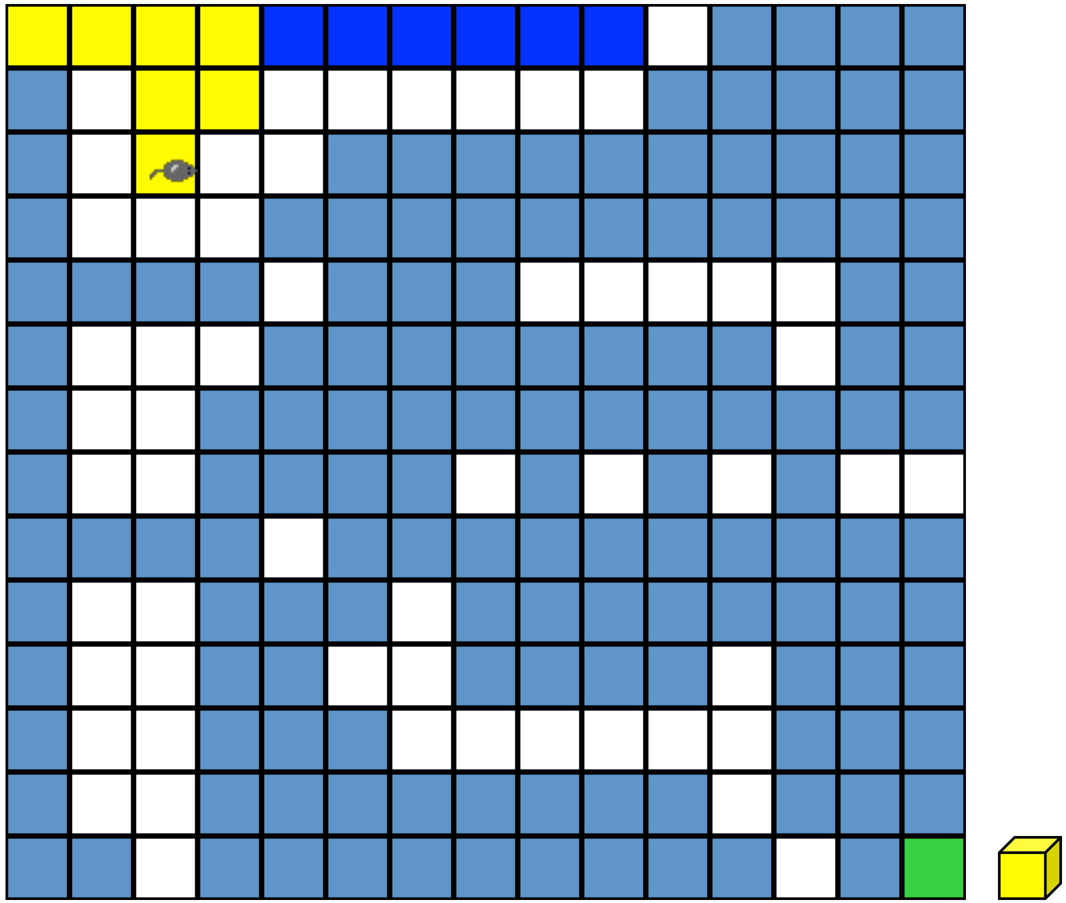
- Move left.

Rat In A Maze



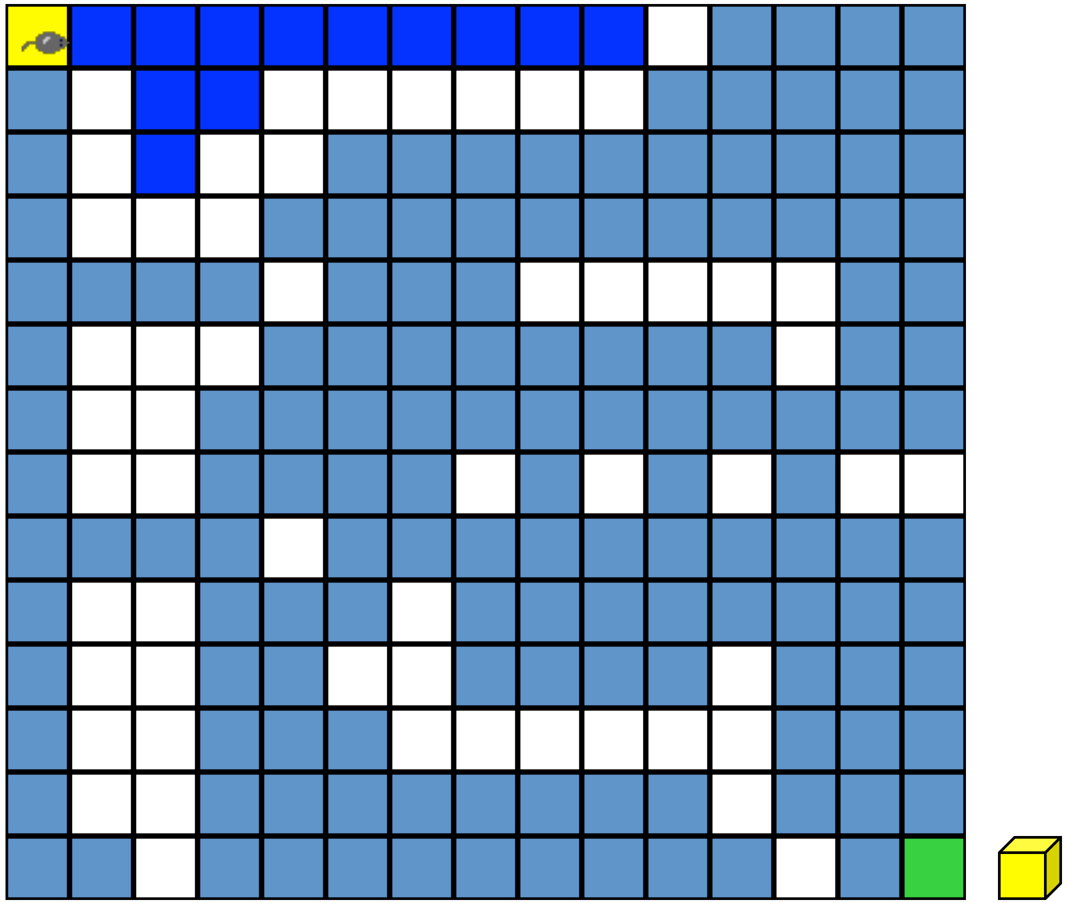
- Move down.

Rat In A Maze



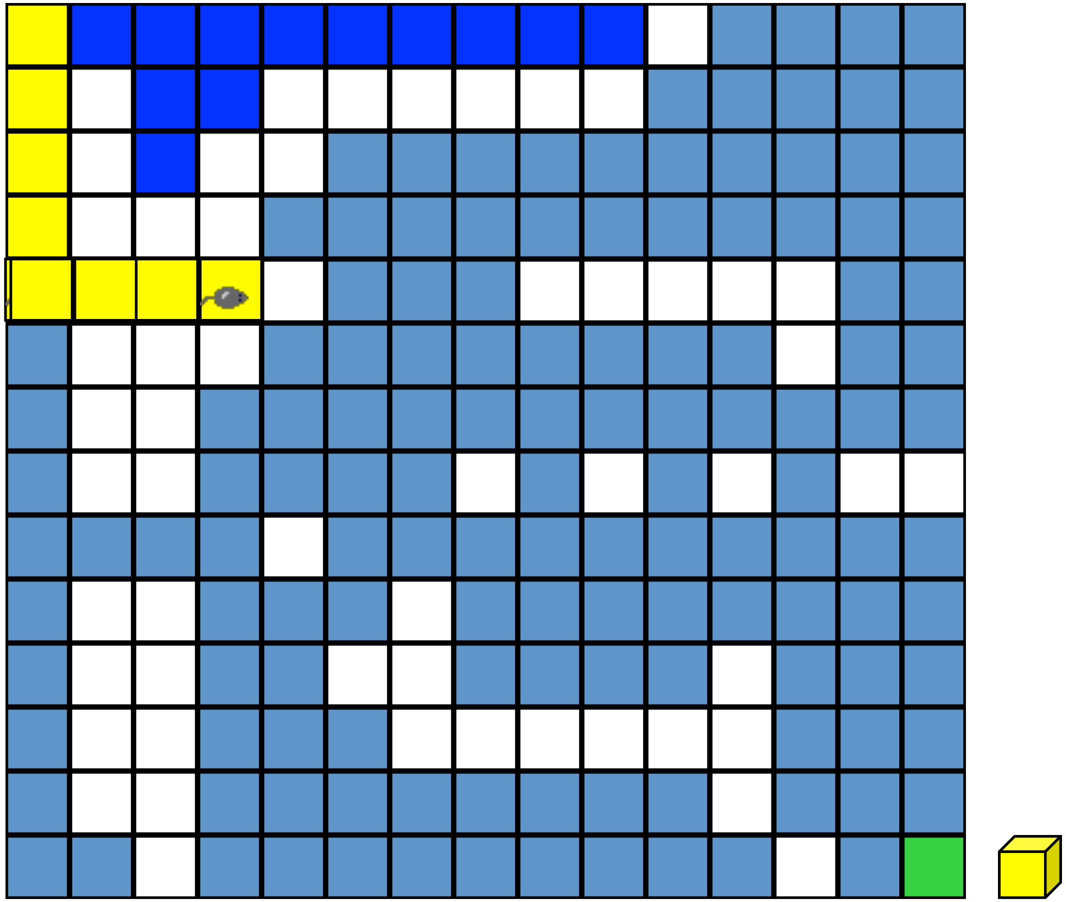
- Move backward until we reach a square from which a forward move is possible.

Rat In A Maze



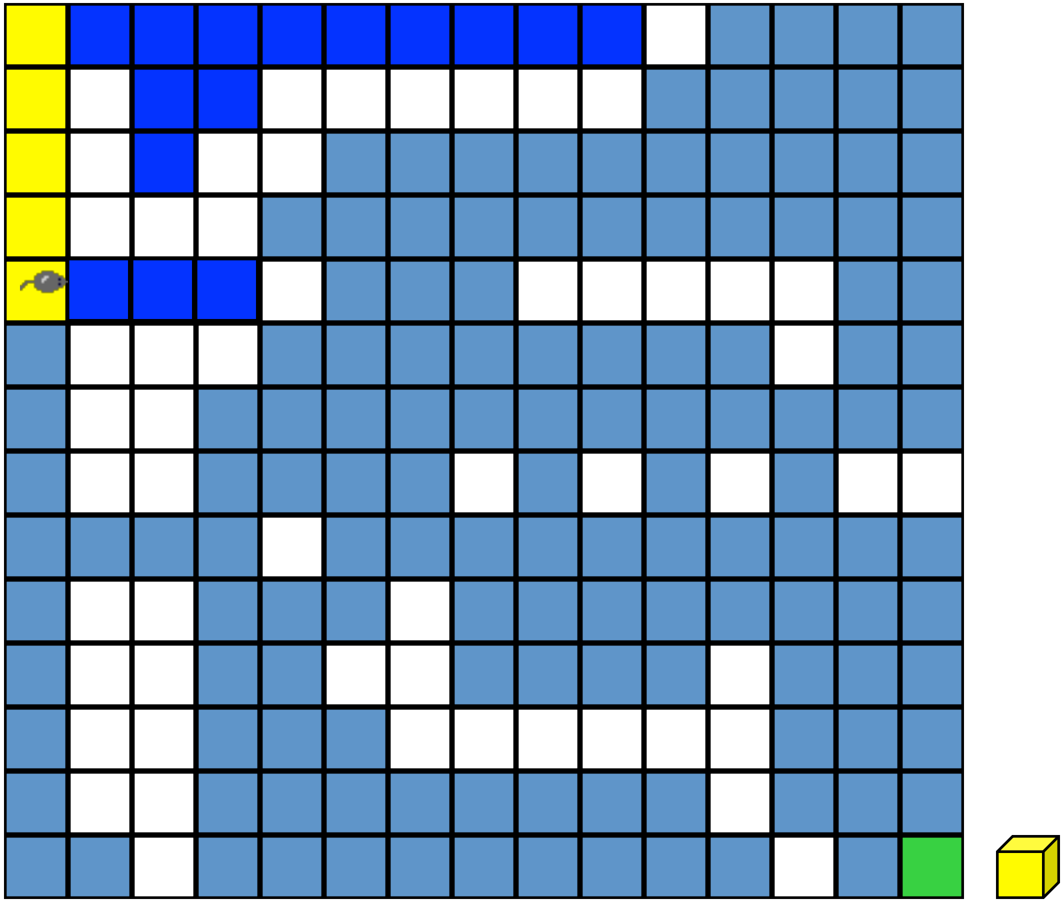
- Move backward until we reach a square from which a forward move is possible.
- Move downward.

Rat In A Maze



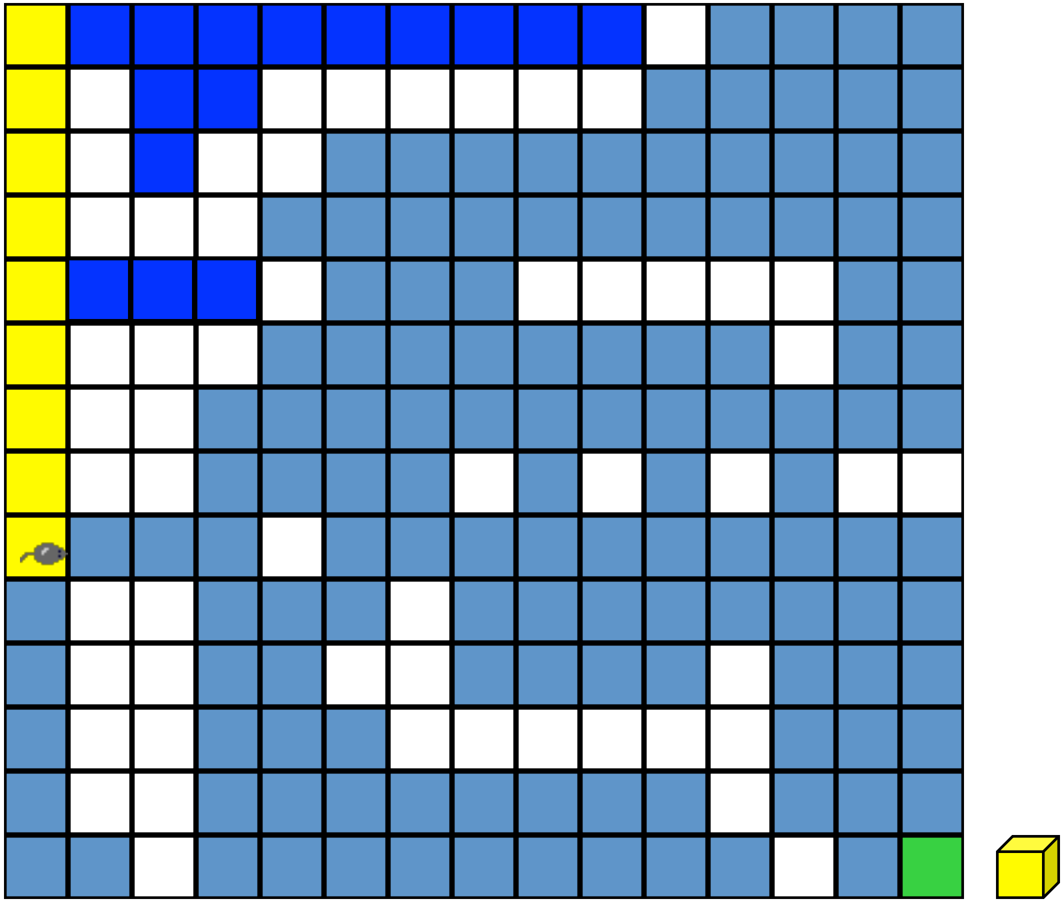
- Move right.
- Backtrack.

Rat In A Maze



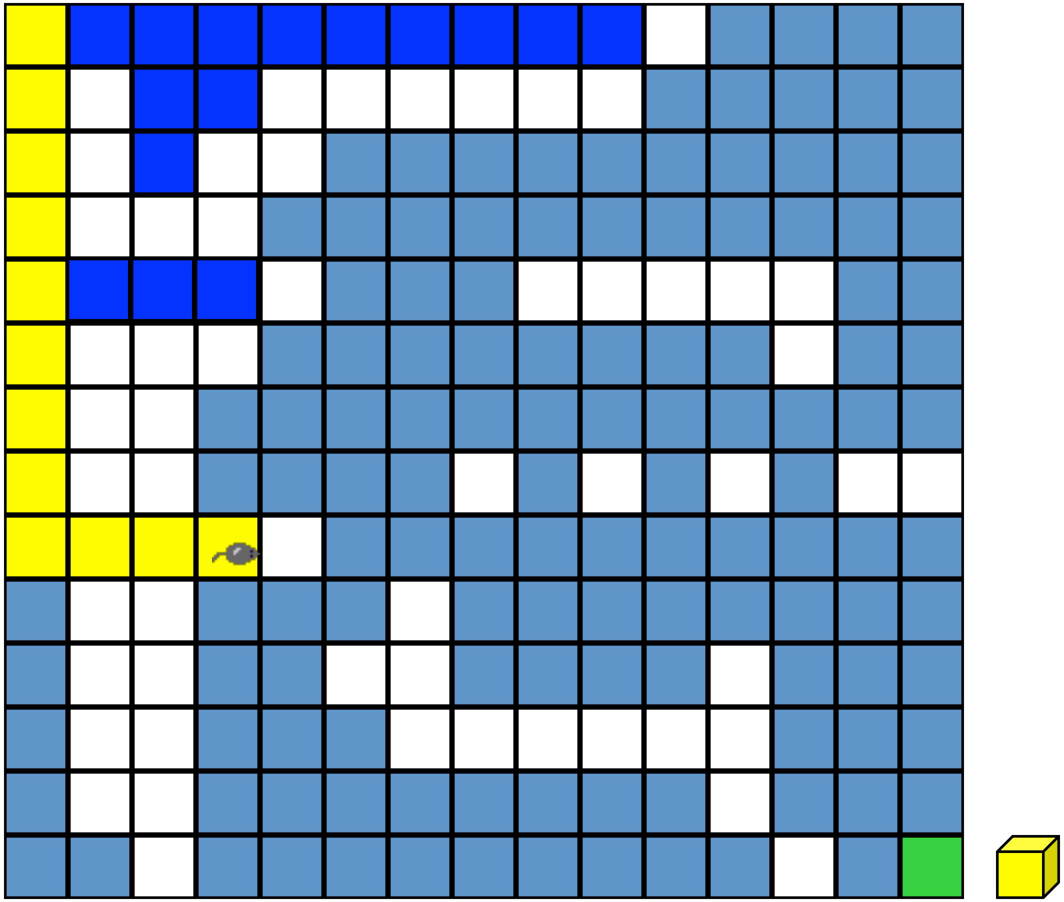
- Move downward.

Rat In A Maze



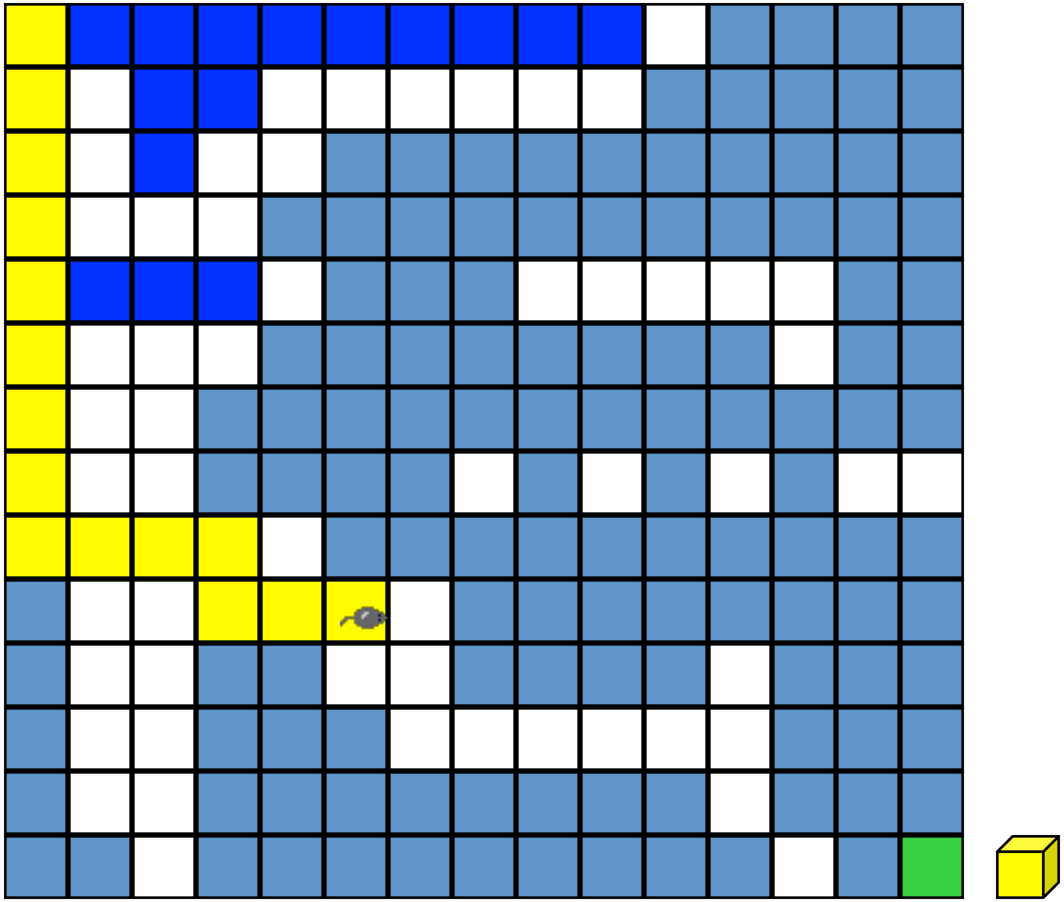
- Move right.

Rat In A Maze



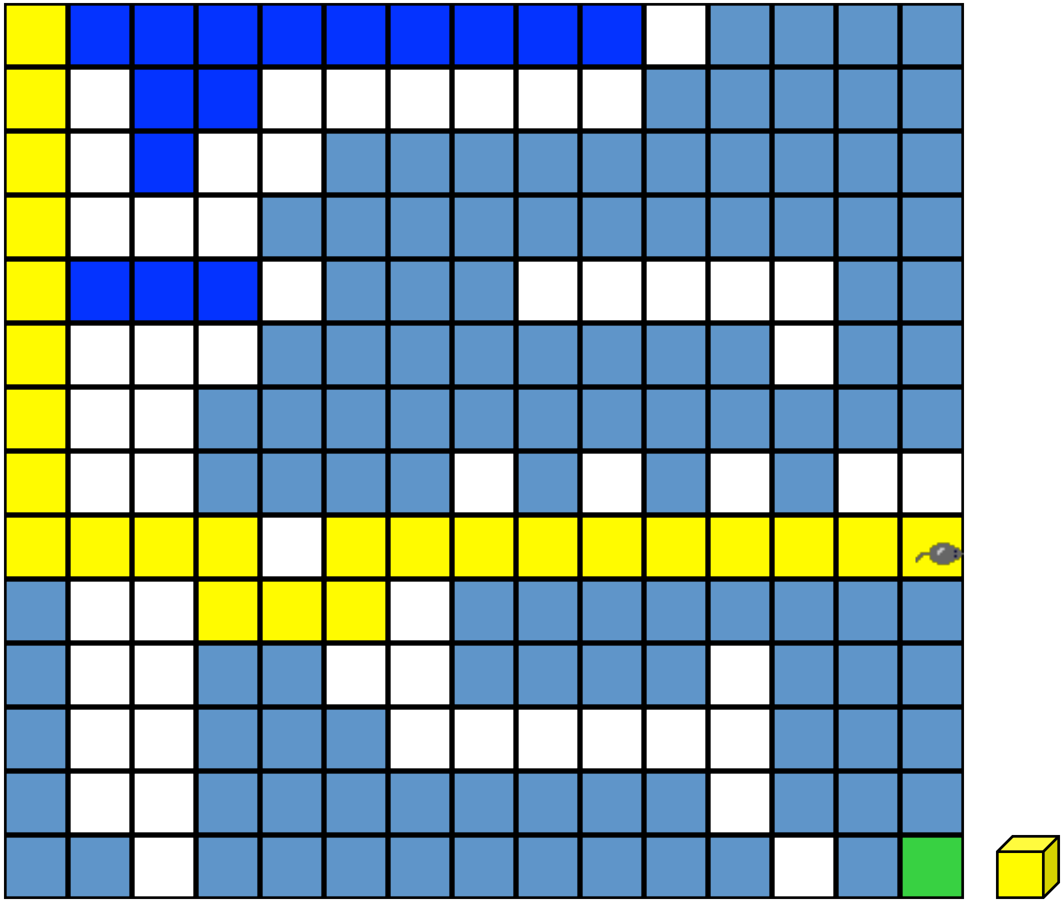
- Move one down and then right.

Rat In A Maze



- Move one up and then right.

Rat In A Maze



- Move down to exit and eat cheese.

Standing… Wondering…

- Move forward whenever **possible**
  - No wall & not visited
- Move back ---- HOW?
  - Remember the footprints
  - OR …… Better?
  - NEXT possible move from previous position
- Storage?
  - STACK

Path from maze entry to current position operates as a stack!

It’s a LONG life …

- How to put an end to this misery? RIP
  - God bless it!
  - Dame it!
    - Whenever exist a possible move from previous positions
    - Whenever the stack is not empty

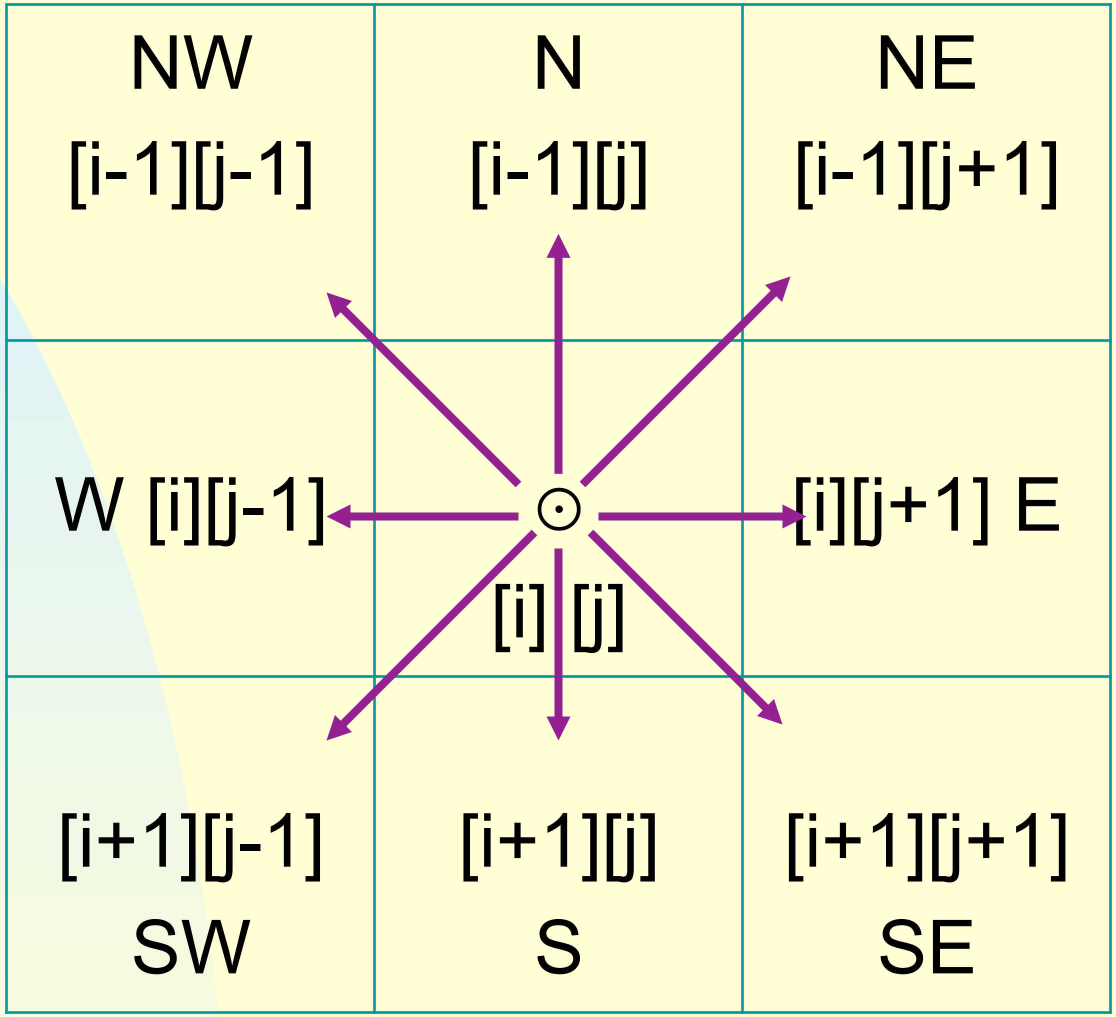
To Do: A Mazing Problem

Problem: find a path from the entrance to the exit of a maze.

entrance	**0**	1	0	0	1	1	0	1	1	
	1	**0**	**0**	1	0	0	1	1	1	
	0	1	1	**0**	1	1	1	0	1	
	1	1	0	**0**	1	**0**	**0**	1	0	
	1	0	0	1	**0**	1	1	**0**	1	
	0	0	1	1	0	1	**0**	1	1	
	0	1	0	0	1	1	**0**	**0**	**0**	**exit**

Representation:

- **maze[i][j], 1 ≤ i ≤ m, 1 ≤ j ≤ p.**
- **1--- blocked, 0 --- open.**
- **the entrance: maze[1][1], the exit: maze[m][p].**
- **current point: [i][j].**
- **boarder of 1's, so a maze[m+2][p+2].**
- **8 possible moves: N, NE, E, SE, S, SW, W and NW.**


NW
[i-1][j-1]
N
[i-1][j]
NE
[i-1][j+1]
W [i][j-1]
[i] [j]
[i][j+1] E
[i+1][j-1]
SW
[i+1][j]
S
[i+1][j+1]
SE

To predefine the 8 moves:

```
struct offsets
{
   int a,b;
};
enum directions {N, NE, E, SE, S, SW, W, NW};
offsets move[8];
```

q	move[q].a	move[q].b
N	-1	0
NE	-1	1
E	0	1
SE	1	1
S	1	0
SW	1	-1
W	0	-1
NW	-1	-1

Table of moves

Thus, from [i][j] to [g][h] in SW direction:

g=i+move[SW].a;

h=j+move[SW].b;

The basic idea:

Given current position [i][j] and 8 directions to go, we pick one direction d, get the new position [g][h].

If [g][h] is the goal, success.

If [g][h] is a legal position, save [i][j] and d+1 in a stack in case we take a false path and need to try another direction, and [g][h] becomes the new current position.

Repeat until either success or every possibility is tried.

In order to prevent us from going down the same path twice:

use another array mark[m+2][p+2], which is initially 0.

Mark[i][j] is set to 1 once the position is visited.

First pass:

Initialize stack to the maze entrance coordinates and direction east**;**

while (stack is not empty)

{

(i, j, dir)=coordinates and direction from top of stack**;**

pop the stack**;**

while (there are more moves from (i, j))

{

(g, h)= coordinates of next move **;**

if ((g==m) **&&** (h==p)) success**;**

```
    if ((!maze[g][h]) && (!mark[g][h])) // legal and not visited
    {
        mark[g][h]=1;
        dir=next direction to try;
        push (i, j, dir) to stack;
        (i, j, dir) = (g, h, N);
    }
  }
}
cout << “No path in maze.”<< endl;
```

We need a stack of items:

```
struct Items {
        int x, y, dir;
};
```

Also, to avoid doubling array capacity during stack pushing, we can set the size of stack to m*p.

Now a precise maze algorithm.

```
void path(const int m, const int p)
{ //Output a path (if any) in the maze; maze[0][i] = maze[m+1][i]
  // = maze[j][0] = maze[j][p+1] = 1, 0 ≤ i ≤ p+1, 0 ≤ j ≤ m+1.
    // start at (1,1)
    mark[1][1]=1;
    Stack<Items> stack(m*p);
    Items temp(1, 1, E);
    stack.Push(temp);
    while (!stack.IsEmpty())
    {
         temp= stack.Top();
        Stack.Pop();
        int i=temp.x; int j=temp.y; int d=temp.dir;
```

```
while (d<8)
{
  int g=i+move[d].a; int h=j+move[d].b;
  if ((g==m) && (h==p)) { // reached exit
     // output path
     cout <<stack;
     cout << i<<" "<< j<<" "<<d<< endl; // last two
     cout << m<<" "<< p<< endl;         // points
     return;
    }
```

```
        if ((!maze[g][h]) && (!mark[g][h])) { //new position
            mark[g][h]=1;
            temp.x=i; temp.y=j; temp.dir=d+1;
            stack.Push(temp);
            i=g ; j=h ; d=N;   // move to (g, h)
        }
        else d++;  // try next direction
      }
   }
 cout << "No path in maze."<< endl;
}
```

The operator << is overloaded for both Stack and Items as:

```
template <class T>
ostream& operator<<(ostream& os, Stack<T>& s)
{
   os << “top=“<<s.top<< endl;
   for (int i=0;i<=s.top;i++);
      os<<i<<“:”<<s.stack[i]<< endl;
   return os;
}
```

We assume << can access the private data member of Stack through the friend declaration.

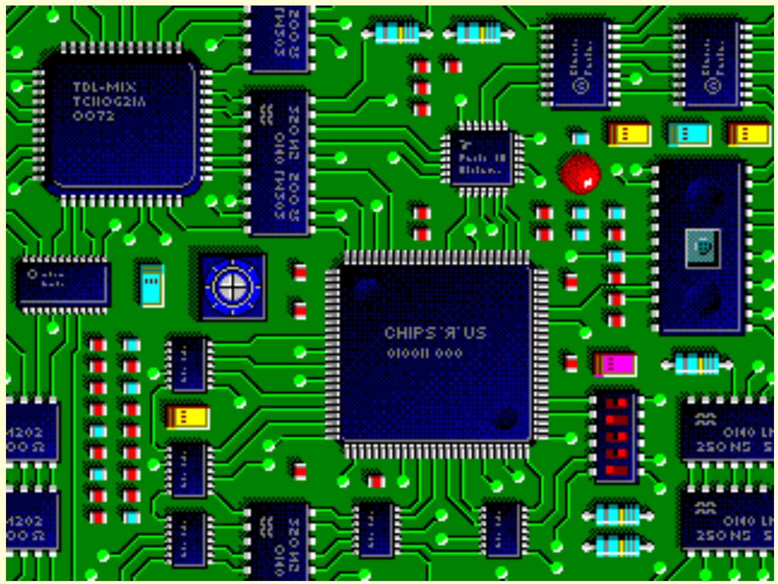
```
ostream& operator<<(ostream& os,Items& item)
{
    return os<<item.x<<“,”<<item.y<<“,”<<item.dir-1;
    // note item.dir is the next direction to go so the current
    // direction is item.dir-1.
}
```

Since no position is visited twice, the worst case computing time is O(m*p).

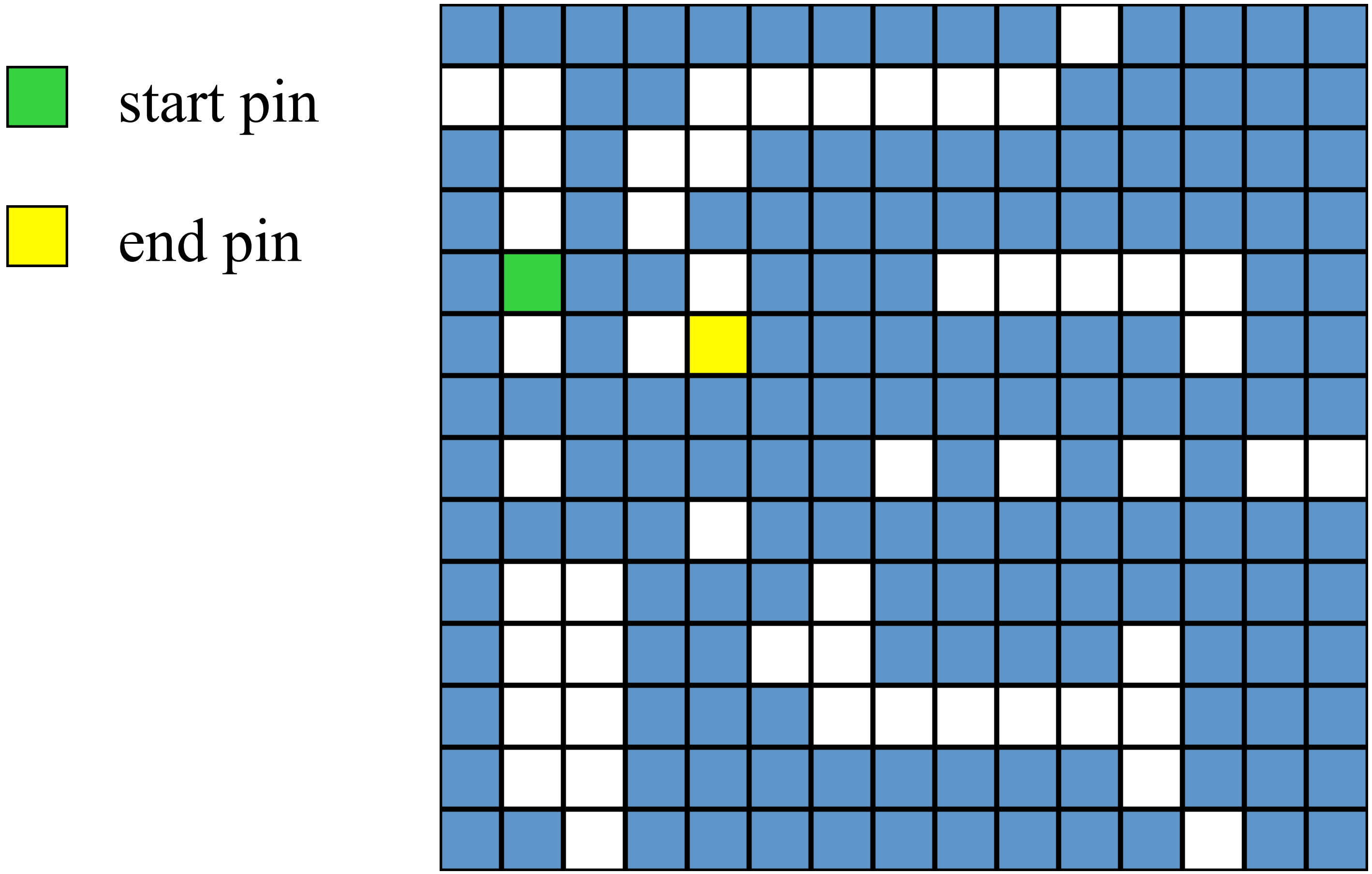
Exercises: P157-2, 3

Queue instead of Stack?

Wire Routing

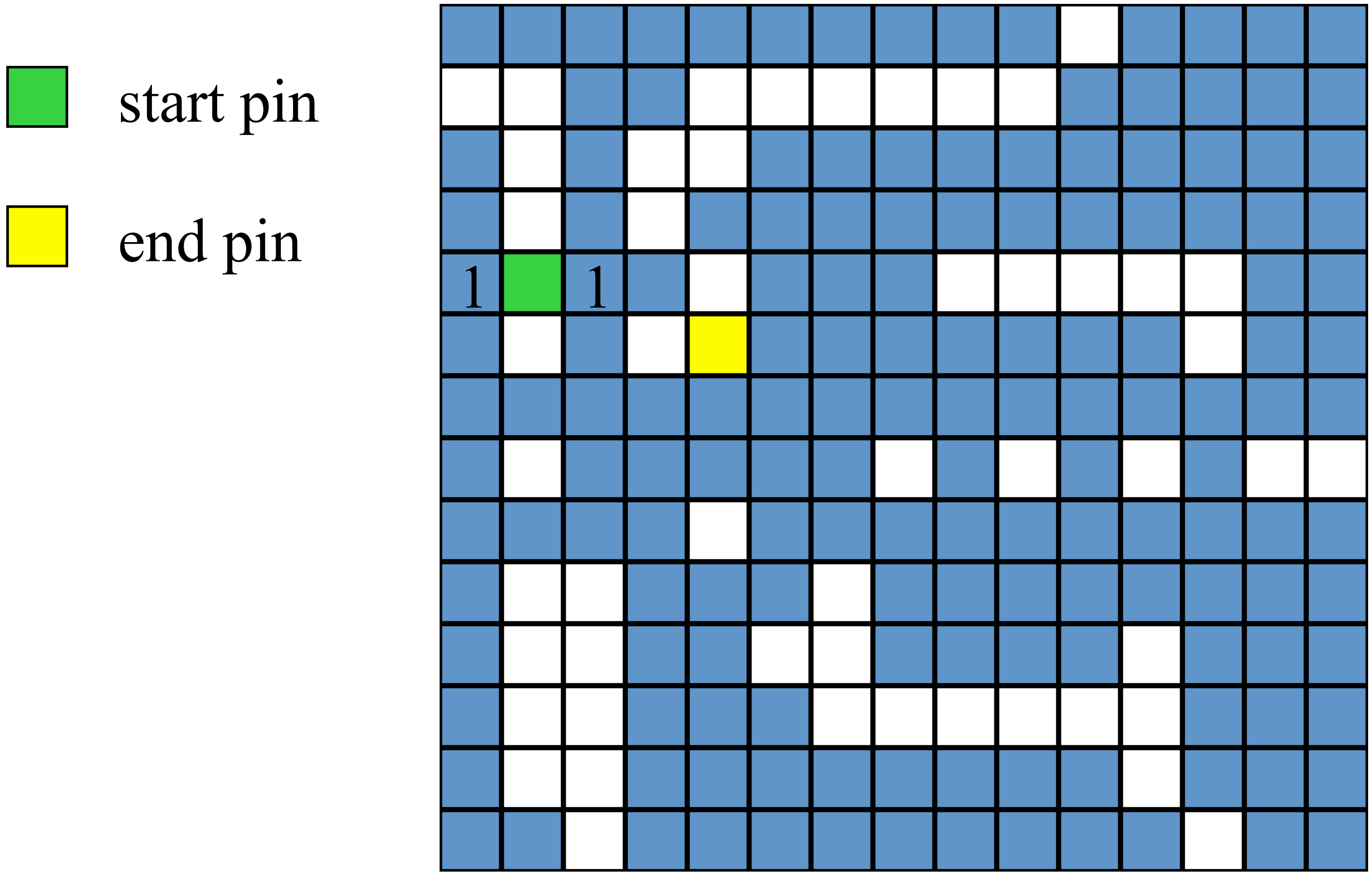


Lee's Wire Router



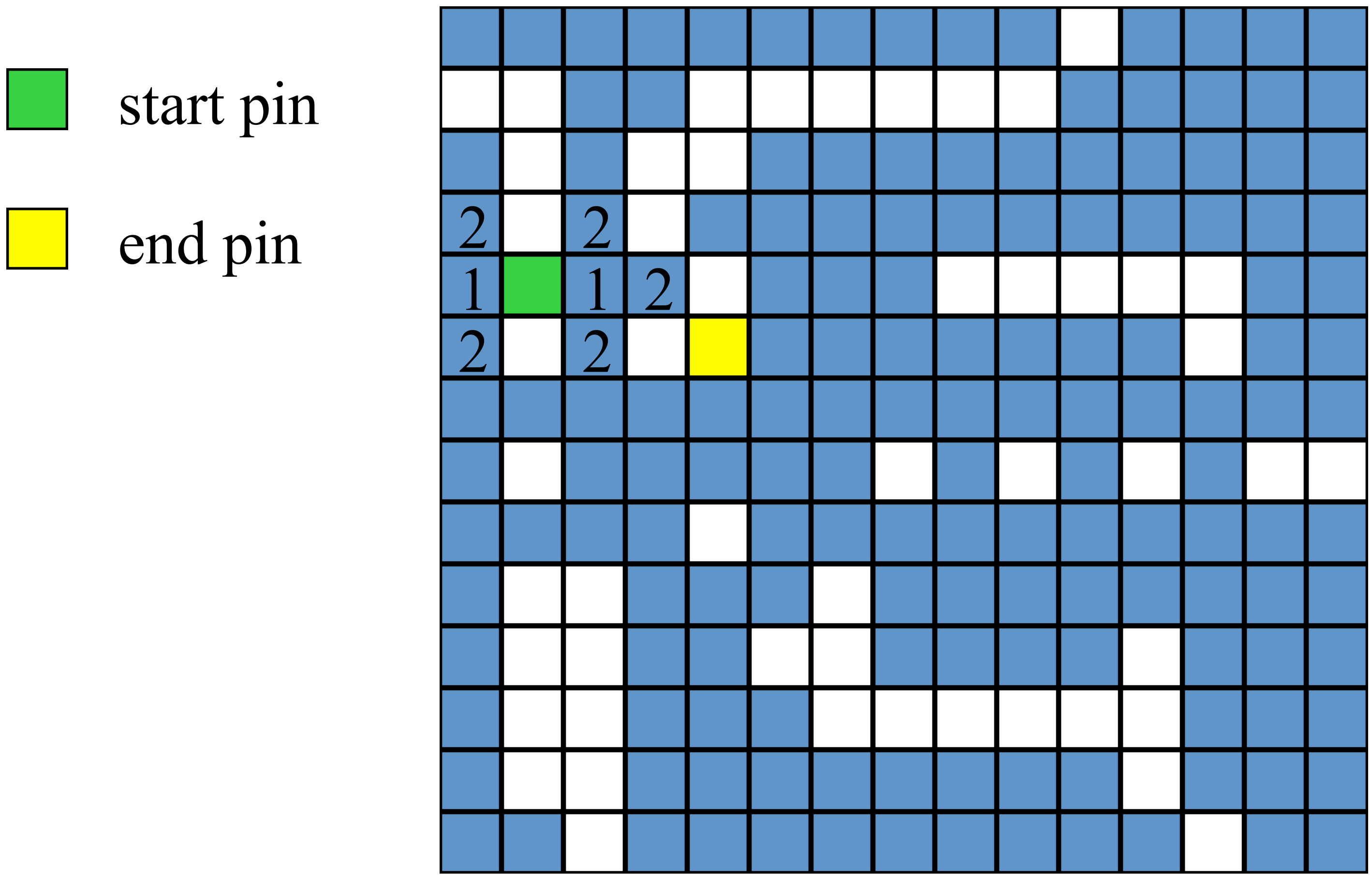
Label all reachable squares 1 unit from start.

Lee's Wire Router



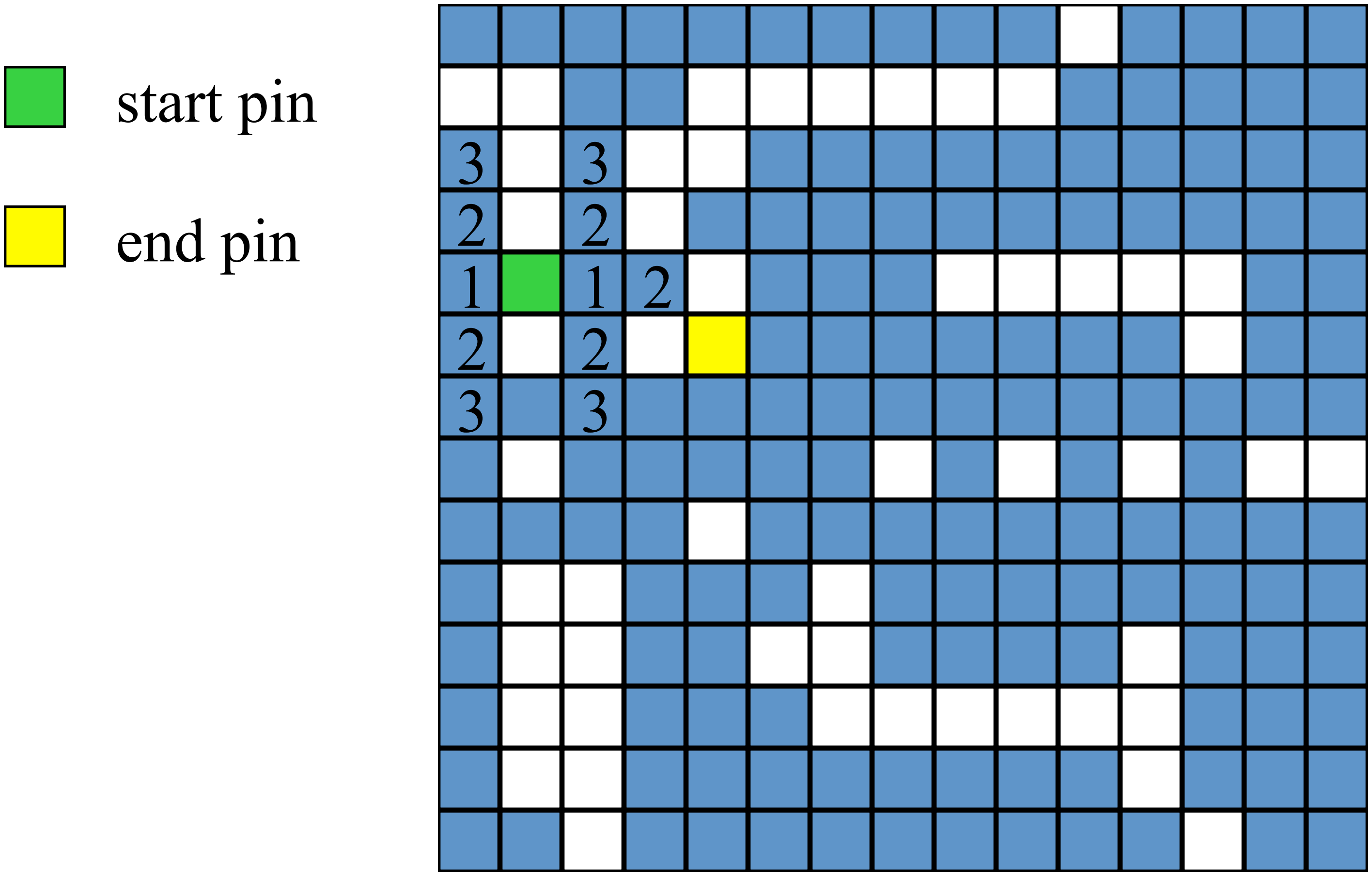
Label all reachable unlabeled squares 2 units from start.

Lee's Wire Router



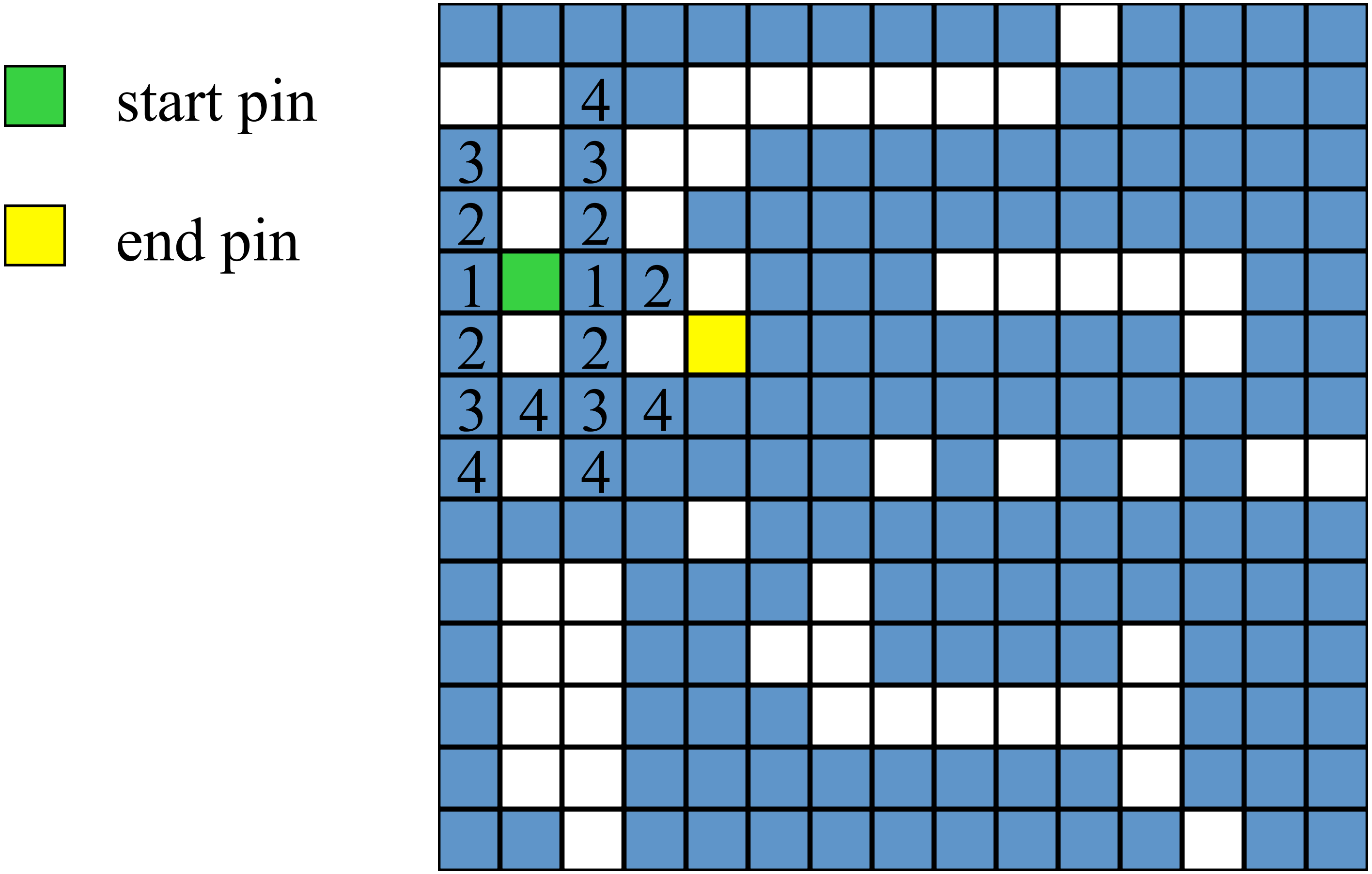
Label all reachable unlabeled squares 3 units from start.

Lee's Wire Router



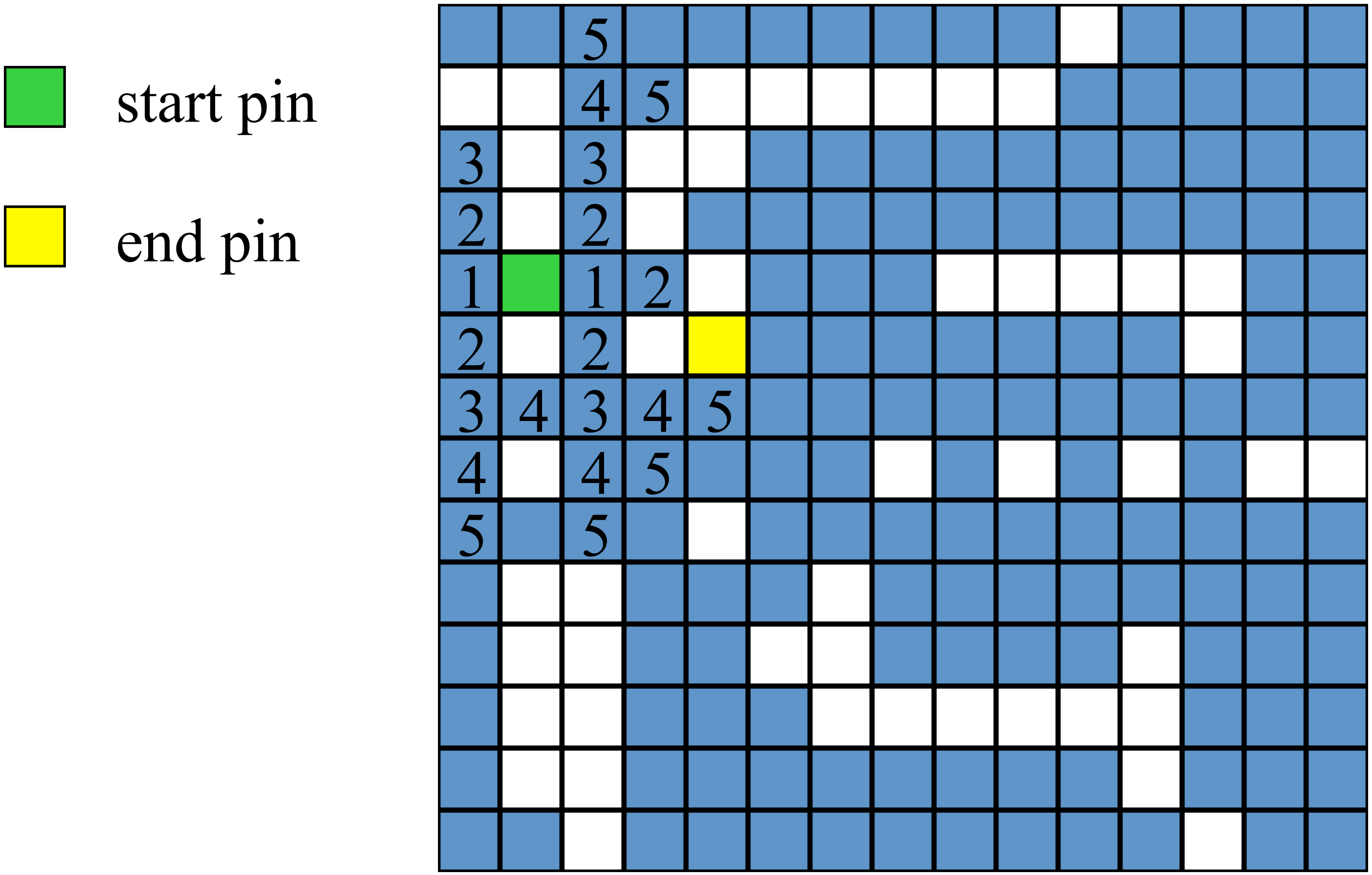
Label all reachable unlabeled squares 4 units from start.

Lee’s Wire Router



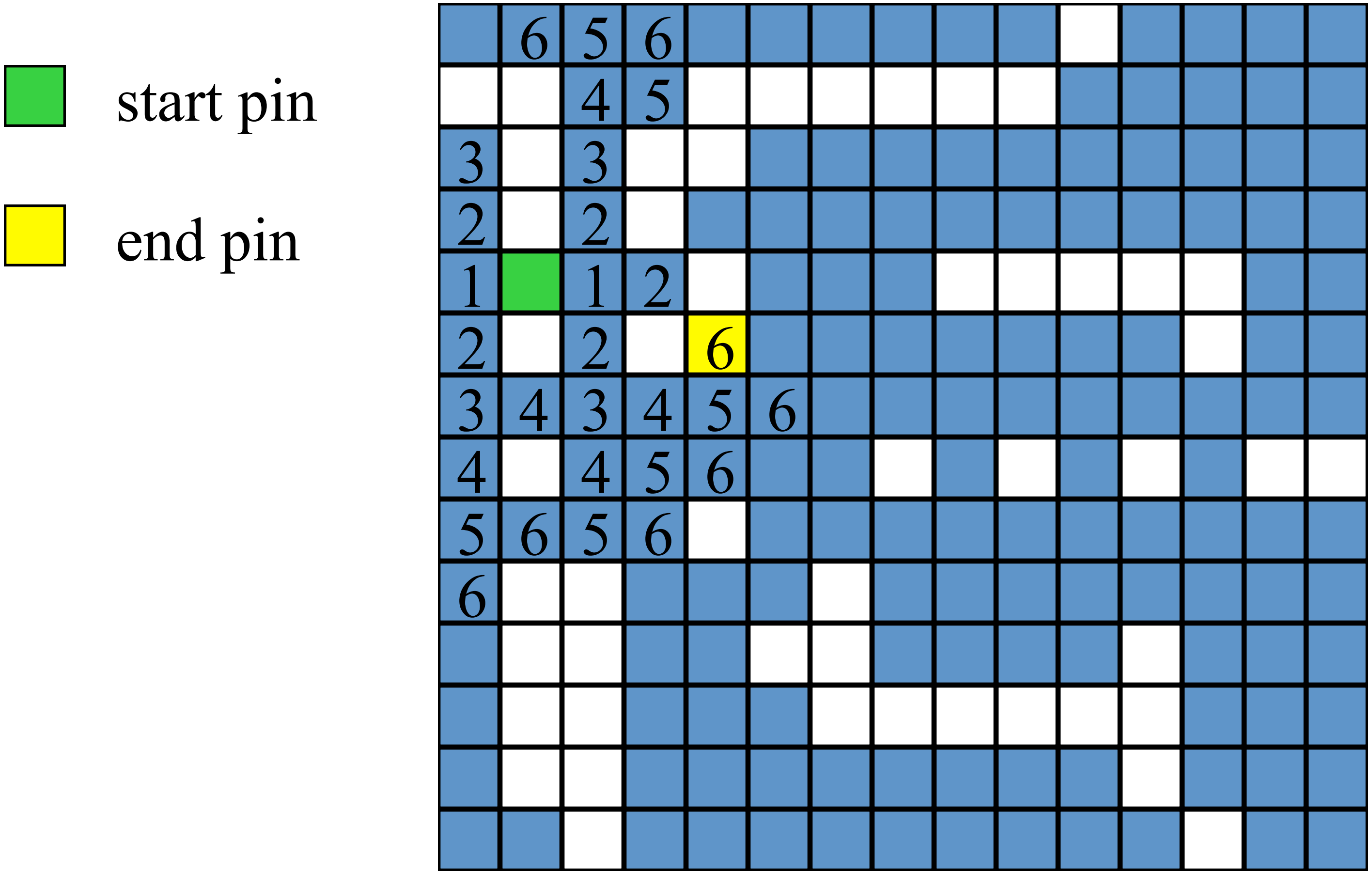
Label all reachable unlabeled squares 5 units from start.

Lee’s Wire Router



Label all reachable unlabeled squares 6 units from start.

Lee’s Wire Router



End pin reached. Traceback.

Evaluation of Expressions

Expressions

A expression is made of operands, operators, and delimiters. For instance,

infix: A / B – C + D * E – A * C

postfix: A B / C – D E * + A C * –

Infix: operators come in-between operands (unary operators precede their operand).

Postfix: each operator appears after its operands.

- **the order in which the operations are carried out must be uniquely defined.**
- **to fix the order, each operator is assigned a priority.**
- **within any pair of parentheses, operators with highest priority will be evaluated first.**
- **evaluation of operators of the same priority will proceed left to right.**
- **Innermost parenthesized expression will be evaluated first.**

The next slide shows a set of sample priorities from C ++.

priority	operator
1	unary minus, !
2	*, /, %
3	+, -
4	<, <=, >=, >
5	==, !=
6	&&
7	||

Problem:

how to evaluate an expression?

postfix: A B / C – D E * + A C * –

Every time we compute a value, we store it in the temporary location T_i, $i \geq 1$. Read the postfix left to right to evaluate it:

A B / C – D E * + A C * –

operation	postfix

T_6 is the result.

Virtues of postfix:

- **no need for parentheses**
- **the priority of the operators is no longer relevant**

Idea:

✓ **make a left to right scan**

✓ **store operands**

✓ **evaluate operators whenever occurred**

What data structure should be used?

- STACK

```
void  Eval(Expression e)
{ // evaluate the postfix expression e. It is assumed that the
  // last token in e is '#'. A function NextToken is used to get
  // the next token from e. Use stack.
    Stack<Token> stack;  //initialize stack
    for  (Token x = NextToken(e); x!='#'; x=NextToken(e))
      if (x is an operand) stack.Push(x);
      else {  // operator
          remove the correct number of operands for operator x
          from stack; perform the operation x and store the result
          (if any) onto the stack;
       }
}
```

Problem: how to evaluate an infix expression?

Solution:

1.Translate from infix to post fix;

2.Evaluate the postfix.

Infix to Postfix

Idea: note the order of the operands in both infix and postfix

infix: A / B – C + D * E – A * C

postfix: A B / C – D E * + A C * –

the same!

immediately passing any operands to the output

store the operators in a stack until the right time.

e.g.

A*(B+C)*D → ABC+*D*

A*(B+C)*D

→ABC+*D*

Next token	stack	output

Attention

From the example, we can see the left parenthesis behaves as an operator with high priority when its not in the stack, whereas once it get in, it behaves as one with low priority.

- **isp (in-stack priority)**
- **icp (in-coming priority)**
- **the isp and icp of all operators in Fig. 3.15 remain unchanged**
- **isp('(')=8, icp('(')=0, isp('#')=8**

Hence the rule:

Operators are taken out of stack as long as their isp is numerically less than or equal to the icp of the new operator.

```
void Postfix (Expression e)
{ // output the postfix of the infix expression e. It is assumed
  // that the last token in e is '#'. Also, '#' is used at the bottom
  // of the stack.
    Stack<Token> stack;  //initialize stack
    stack.Push('#');
```

```
for  (Token x=NextToken(e); x!='#'; x=NextToken(e))
   if (x is an operand) cout<<x;
   else if (x==')')
       { // unstack until '('
         for  (; stackTop()!='('; stack.Pop())
              cout<<stack.Top();
         stack.Pop();  // unstack '('
         }
   else { // x is an operator
        for (; isp(stack.Top()) <= icp(x); stack.Pop())
            cout<<stack.Top();
        stack.Push(x);
    }
 // end of expression, empty the stack
 for (; !stack.IsEmpty()); cout<<stack.Top(), stack.Pop());
 cout << endl;
}
```

Analysis:

- **Computing time: one pass across the input with n tokens, O(n).**
- **The stack will not be deeper than 1 (‘#’) + the number of operators in e.**

Exercises: P165-1,2

Can we evaluate infix expressions directly?

infix: A / B – C + D * E – A * C

Exercise

R [7][7]

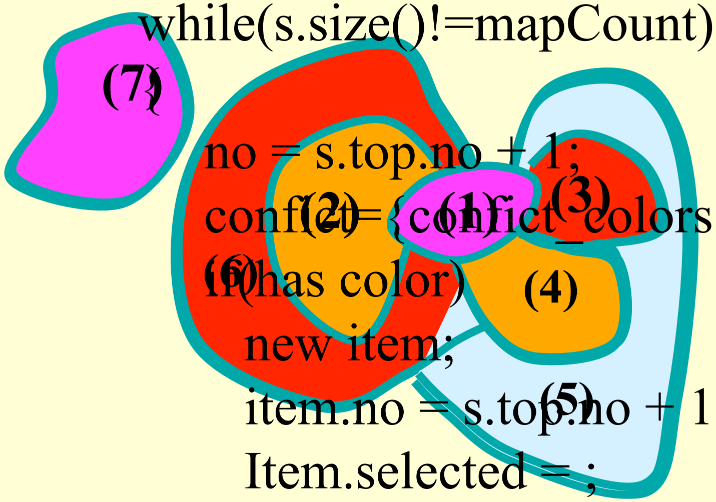
	1	2	3	4	5	6	7
1	0	1	1	1	1	1	0
2	1	0	0	0	0	1	0
3	1	0	0	1	1	0	0
4	1	0	1	0	1	1	0
5	1	0	1	1	0	1	0
6	1	1	0	1	1	0	0
7	0	0	0	0	0	0	0

1# Purple

2# Yellow

3# Red

4# Green




```
Item.no = 1;
Item.seleted = 1;
Item.candidate={2,3,4};
s.push(item);
while(s.size()!=mapCount)
{
  no = s.top.no + 1;
  conflict={conflict_colors}
  if(has color)
    new item;
    item.no = s.top.no + 1;
    Item.selected = ;
    Item.candidate =;
    s.push(item);
  else
    while(!s.top.candidate)
      s.pop();
    s.top.selected =;
    s.top.candidate =;
}
```

1	2	3	4	5	6	7
1	2	3	2	4	3	1

```
void  MapColor  ( int    R[ n ][ n ],   int    s[ n ] )  {
     s [ 0 ] =1;                    //  00  区染 1 色
     i = 1 ;      j = 1 ;           //  i 为区域号，j 为染色号
     while  ( i < n )  {
        while  (( j ≤ 4 )  &&  ( i < n ))  {
           k = 0 ;                  //  k 指示已染色区域号
            while  (( k < i )  &&  (s [ k ] *  R [ i ][ k ] != j ))  k++ ;
           // 判相邻区是否已染色且不同色
            if  ( k < i )  j ++ ;           //   用 j+1 色继续试探
            else  {
              s [ i++ ] = j ;  j = 1 ;
            }  //  该区染色成功，结果进栈，继续染色下一区
        }
        if  ( j > 4 )  { j = s [ - - i ] + 1;  } ;
        //    （回溯）变更栈顶区域的染色色数，用新颜色重试
     }
}
```

Backtracking(回溯)

A method to try all possibilities using recursion.

When there are several possibilities,

- take one and go on;
- go back to the most recent choice, and try another possibility when a dead end is reached.