Stacks and Queues

3.2 The Stack Abstract Data type

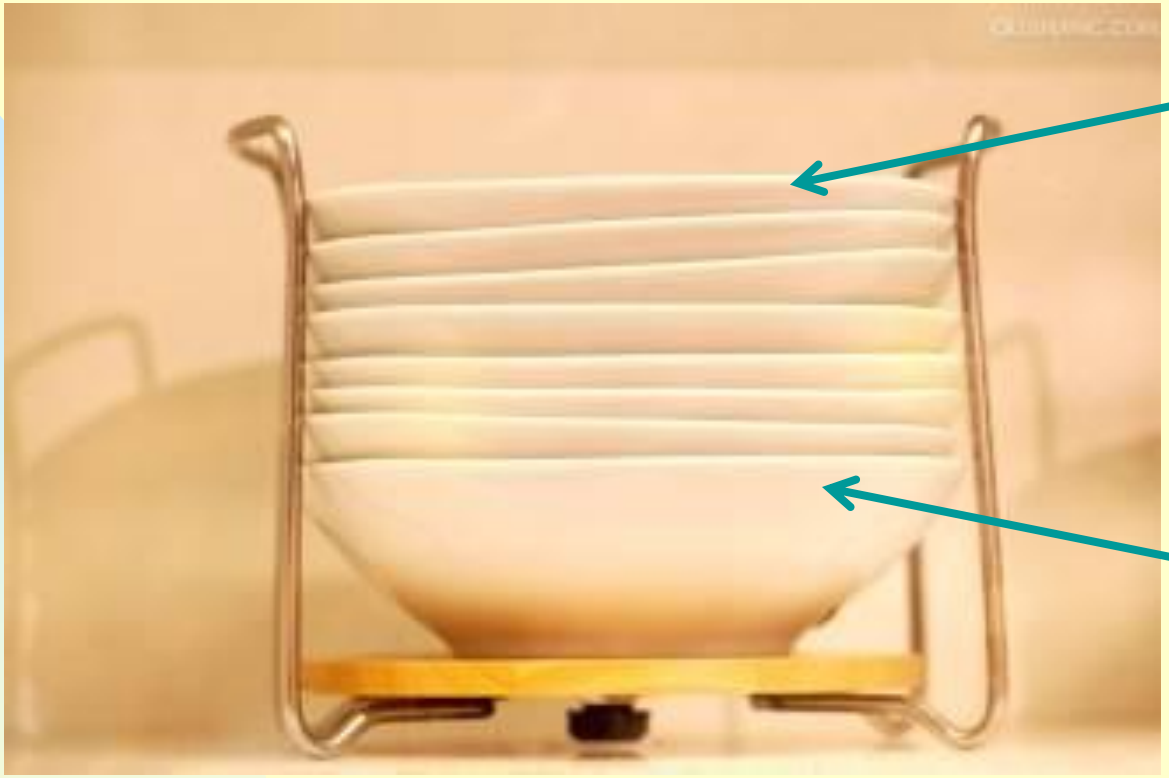
Ordered (Linear) list.

One end is called top.

Other end is called bottom.

Pushes(insertions) and Pops(removals) are made at the top end only.

Stack Of Cups



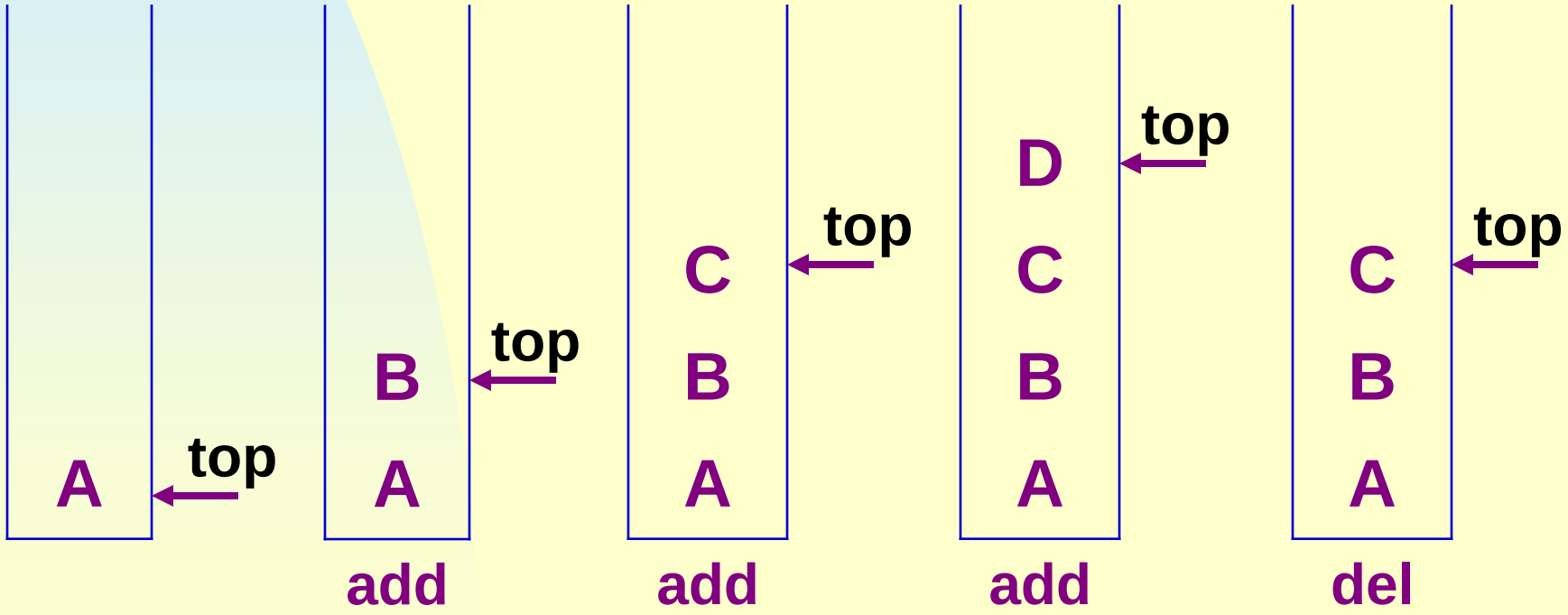
Add a cup to the stack.

Remove a cup from new stack.

A stack is a LIFO(Last In First Out) list.

Inserting and deleting elements in a stack

Given a stack $S=(a_0,a_1,\ldots a_{n-1})$, we say that a_0 is the bottom element, a_{n-1} is the top element.



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() const
{
    if (IsEmpty()) throw “Stack is Empty”;
    return stack[top];
}

template <class T>
void Stack<T>::Push(const T& x)
{
    if (top == capacity - 1) //if the stack is full
    {
        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

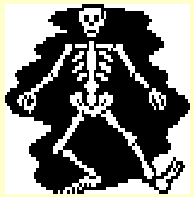
Bus Stop

front

rear

Bus Stop Queue

Bus Stop



front rear rear

3.3 The Queue Abstract Data Type

Linear list.

One end is called front.

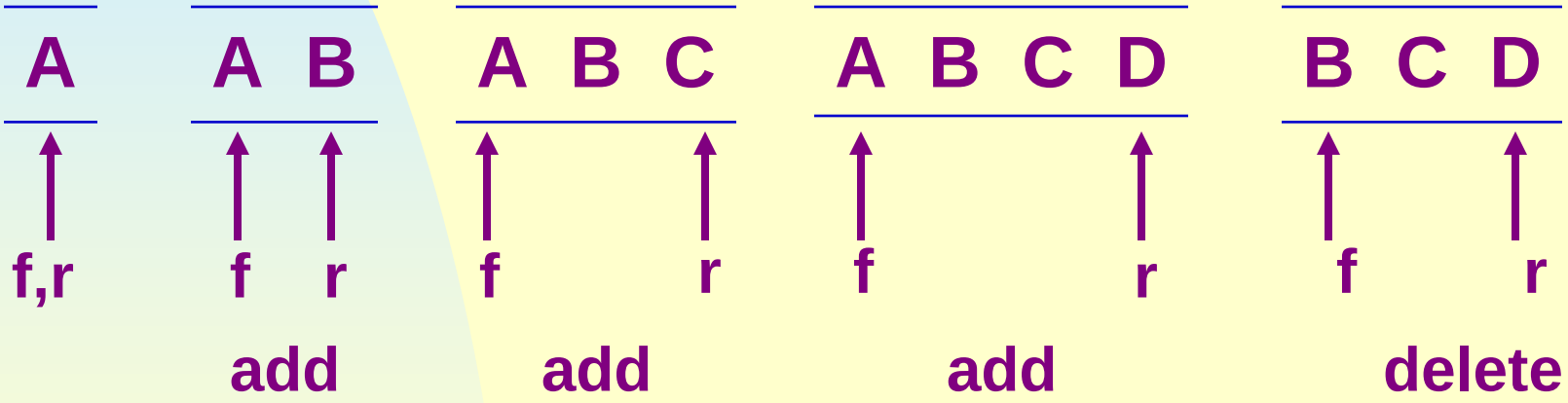
Other end is called rear.

Additions are done at the rear only.

Removals are made from the front only.

A queue is a FIFO(First-In-First-Out) list

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

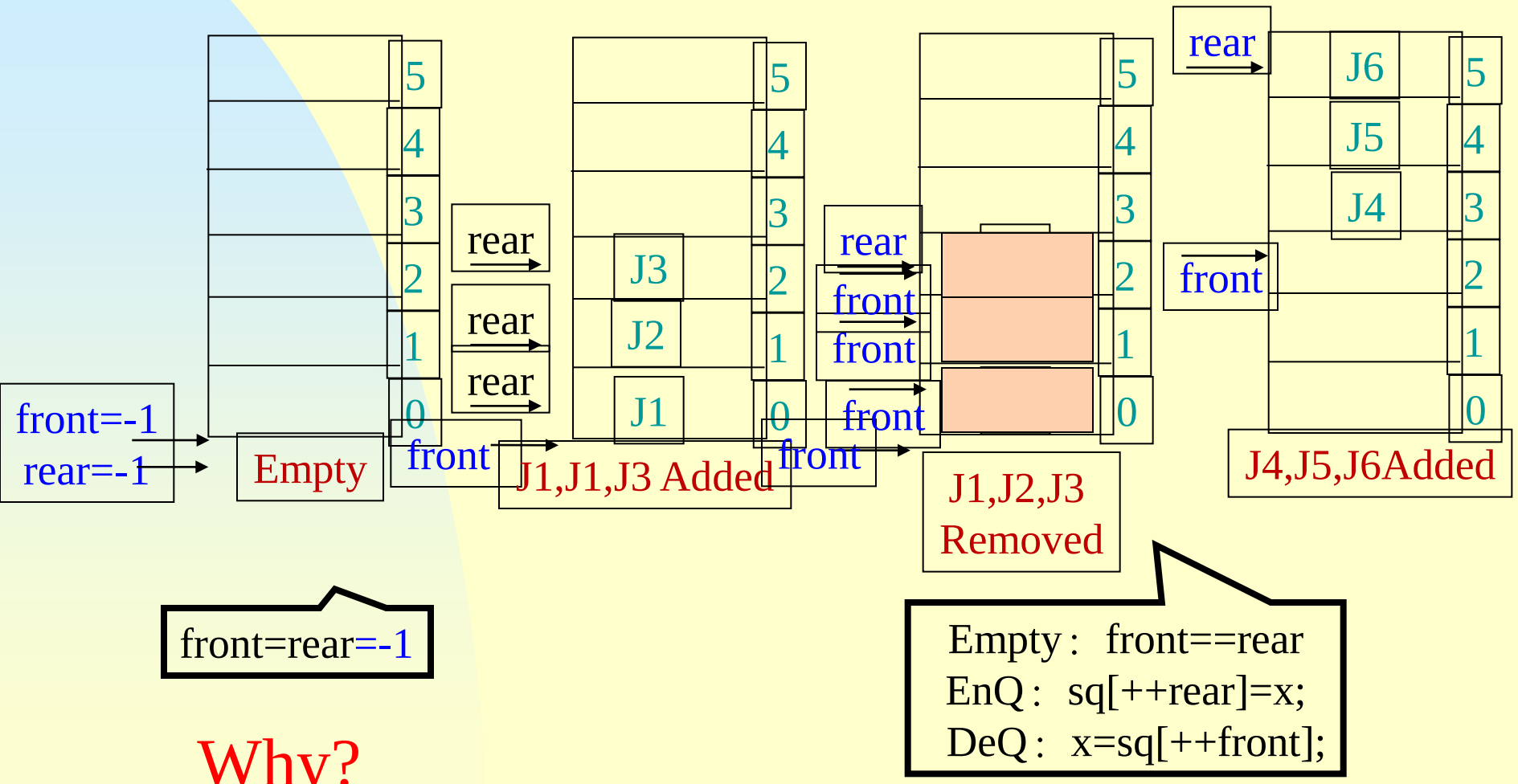
two variable front and rear

front always being one less than the position of the first element(different from the Textbook)

So we have the following data members of Queue:

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

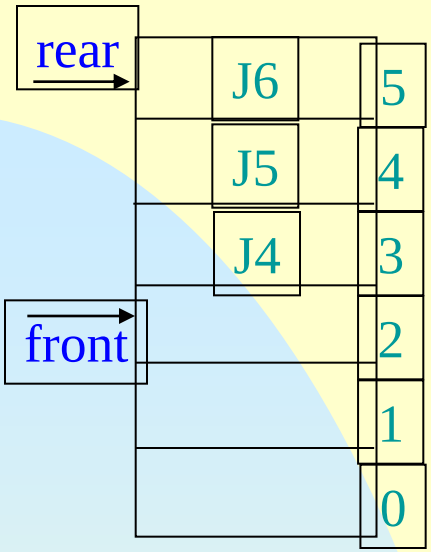
Use a 1D array (positions in a straight line).



front=rear=-1

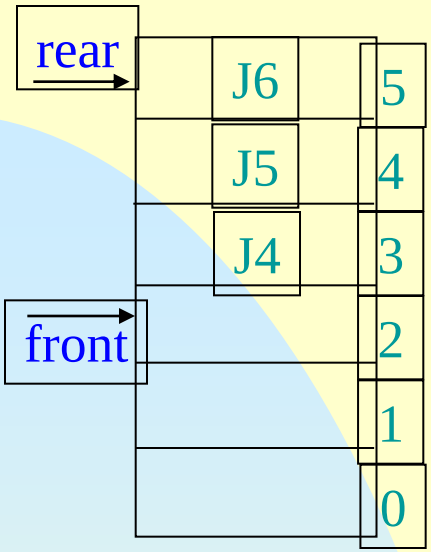
Empty : front==rear
EnQ : sq[++rear]=x;
DeQ : x=sq[++front];

Why?



Problem

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



Problem

False Overflow

Possible solution?

J4→position 0

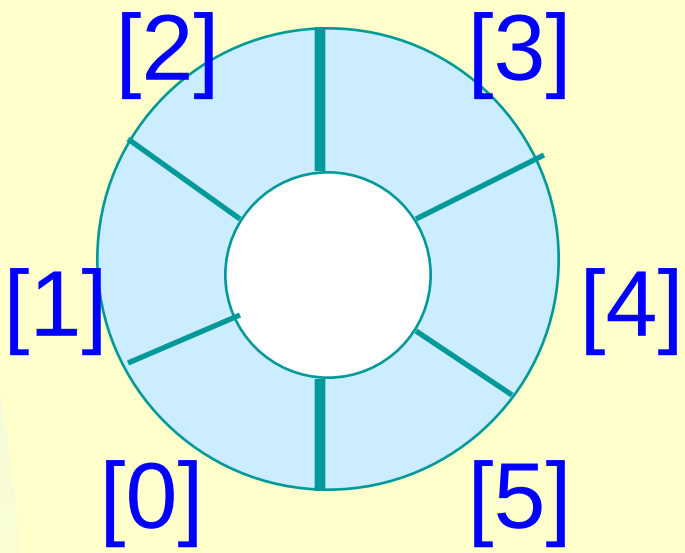
J5→position 1

J6→positon 2

Time complexity is proportional to the numbers of elements(size)

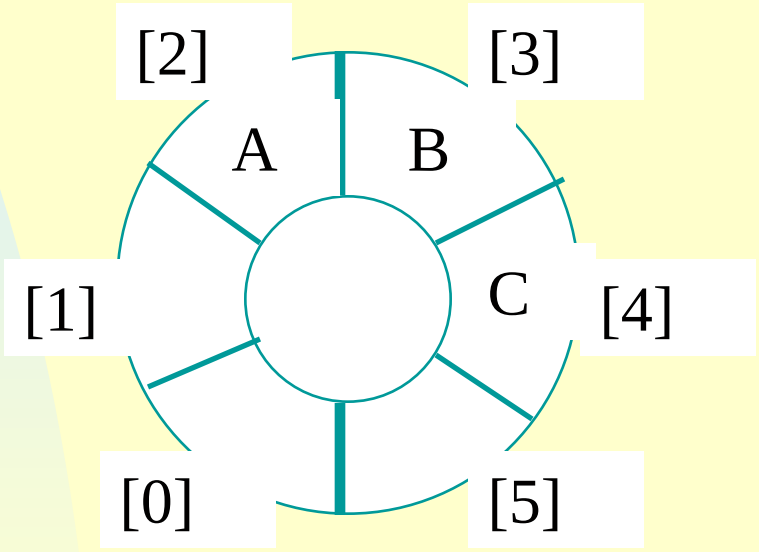
Circular Queue

- Circular view of array.



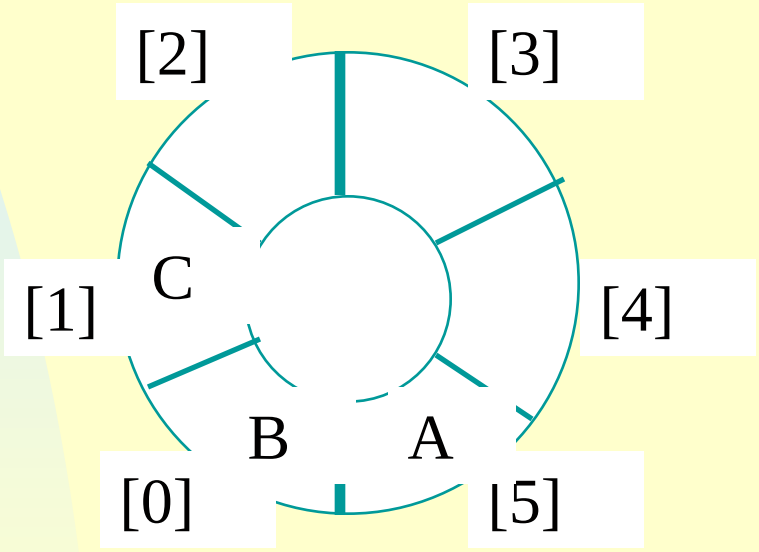
Circular Queue

- Possible configuration with 3 elements.



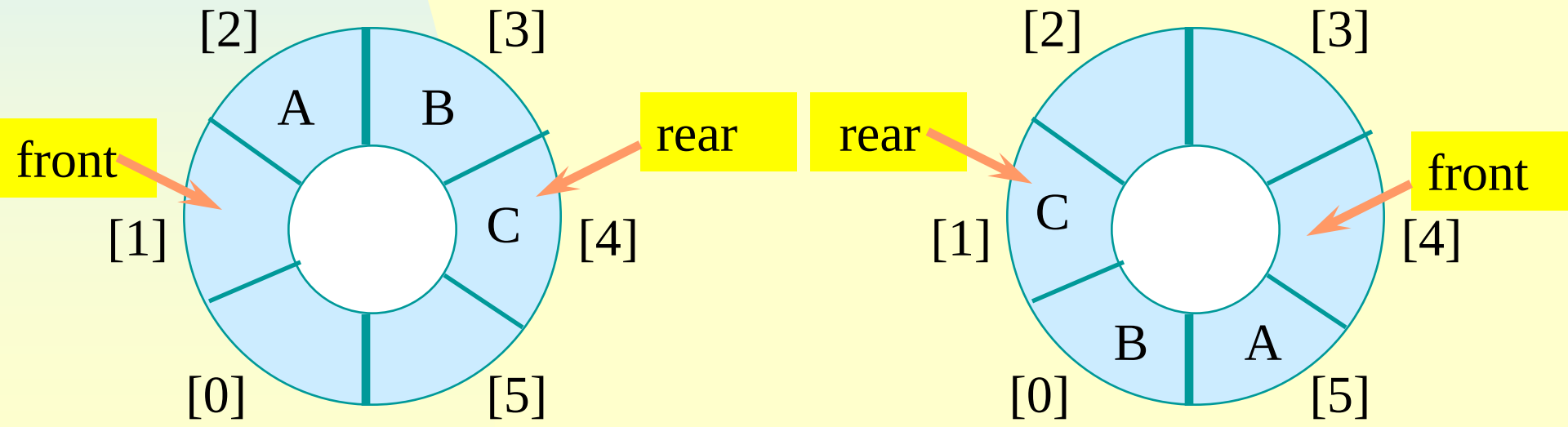
Circular Queue

- Another possible configuration with 3 elements.



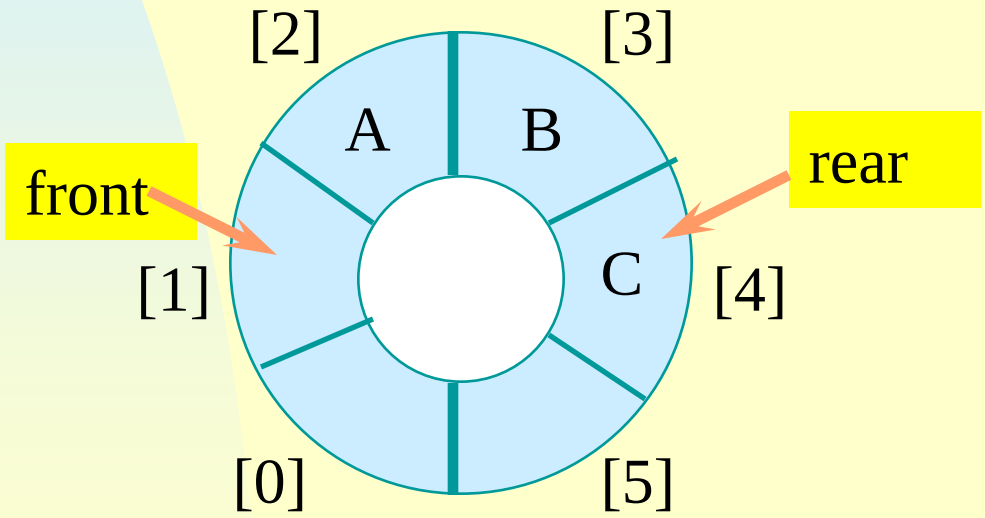
Circular Queue

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



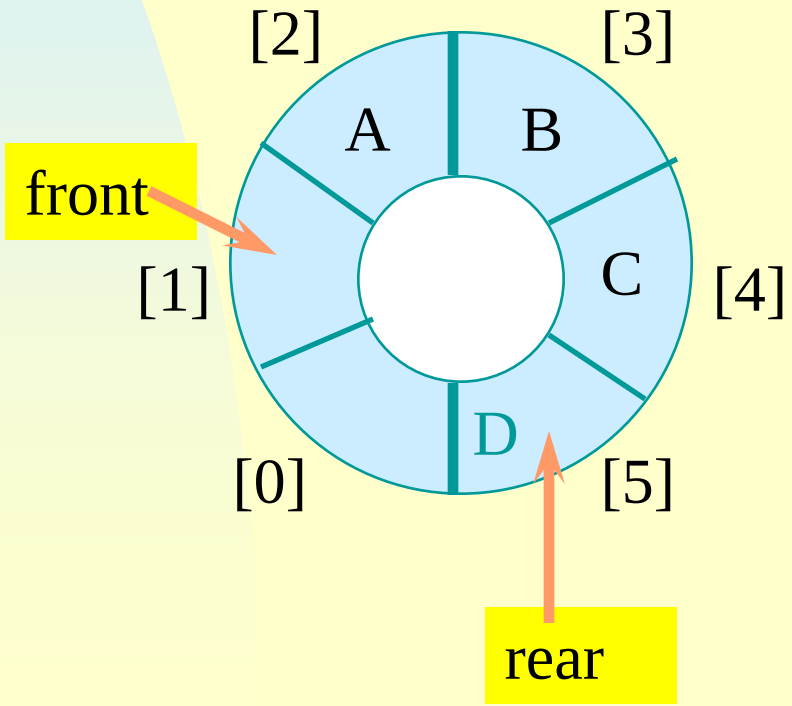
Add An Element

- Move rear one position clockwise.



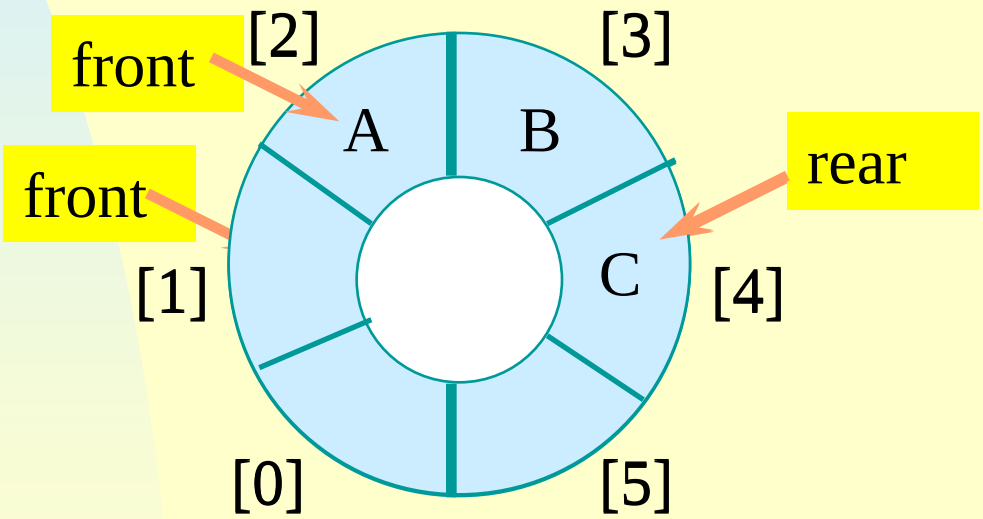
Add An Element

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



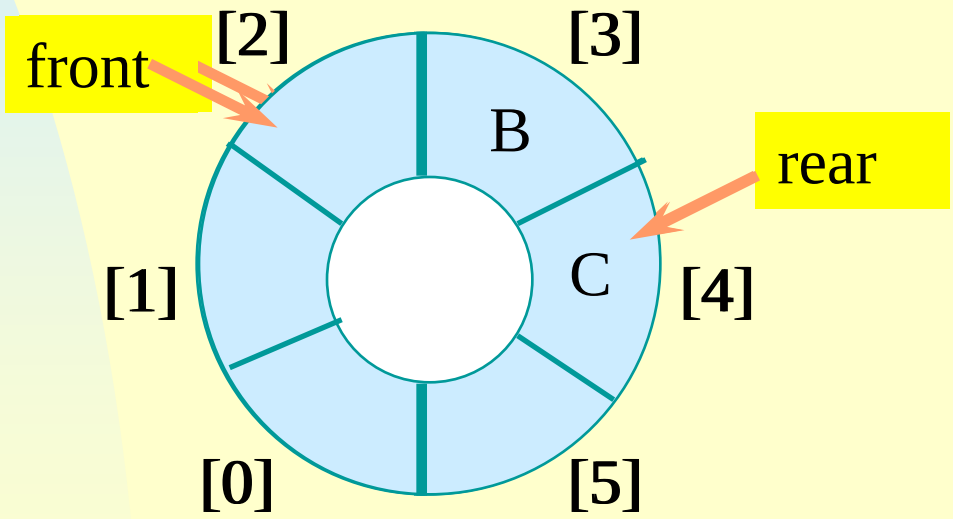
Remove An Element

- Move front one position clockwise.



Remove An Element

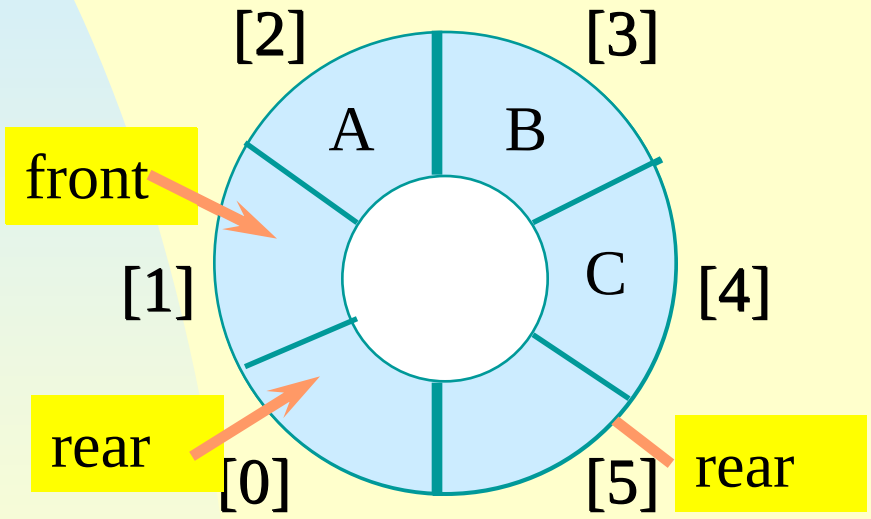
- Move front one position clockwise.
- Then extract one element from queue[front].



Moving rear Clockwise

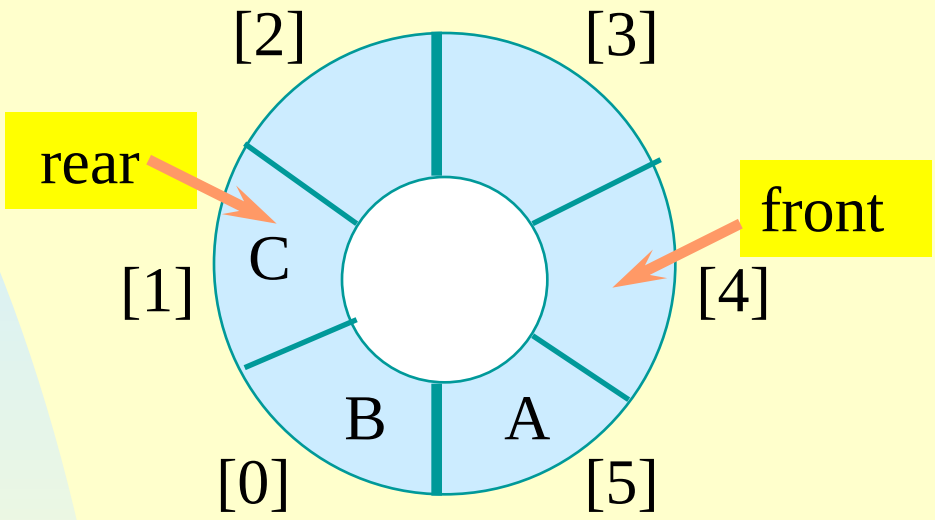
- rear++;

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

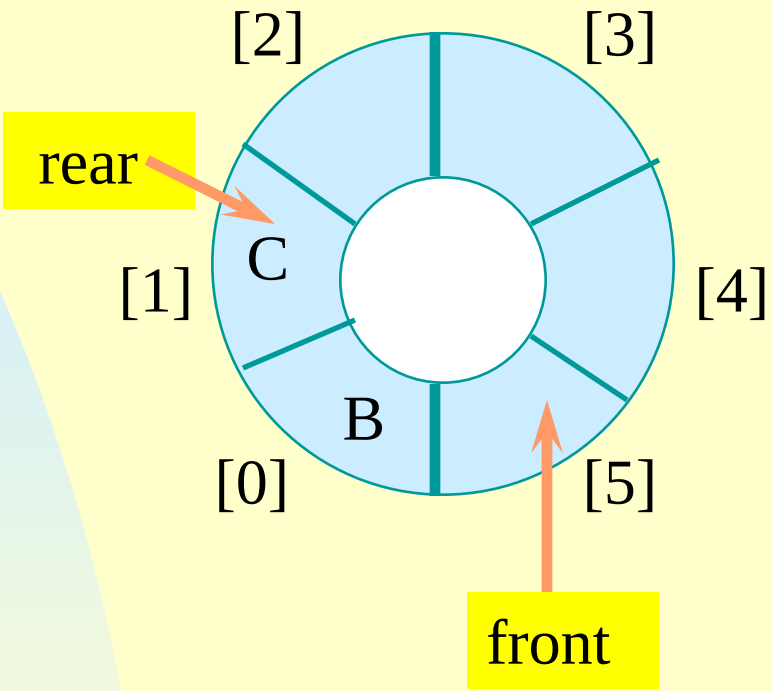


- rear = (rear + 1) % queue.length;

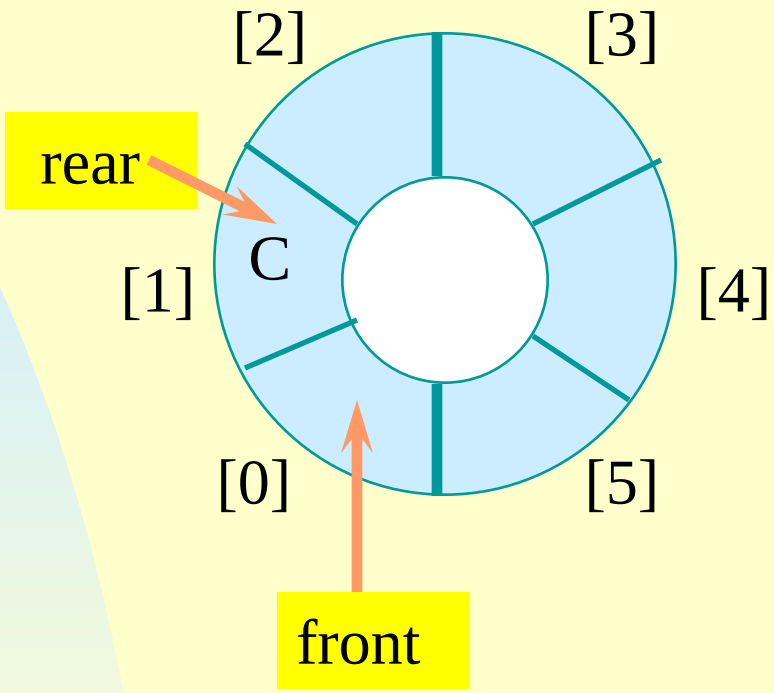
Empty That Queue



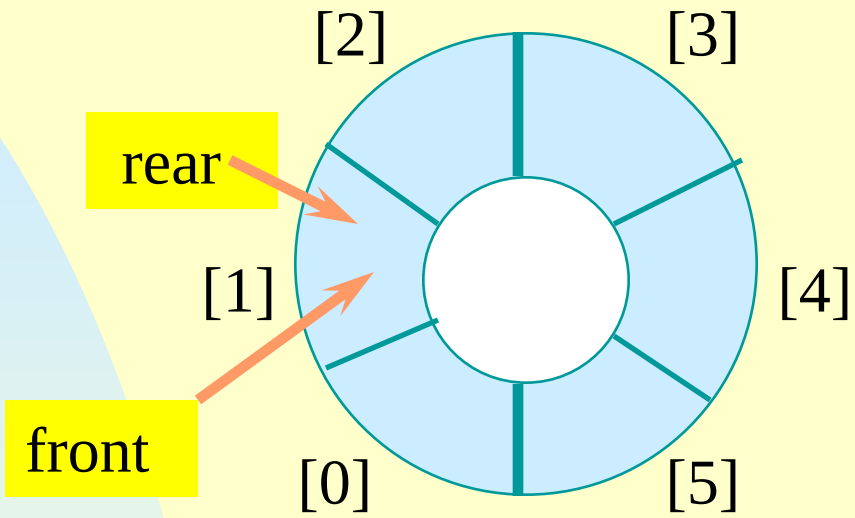
Empty That Queue



Empty That Queue



Empty That Queue

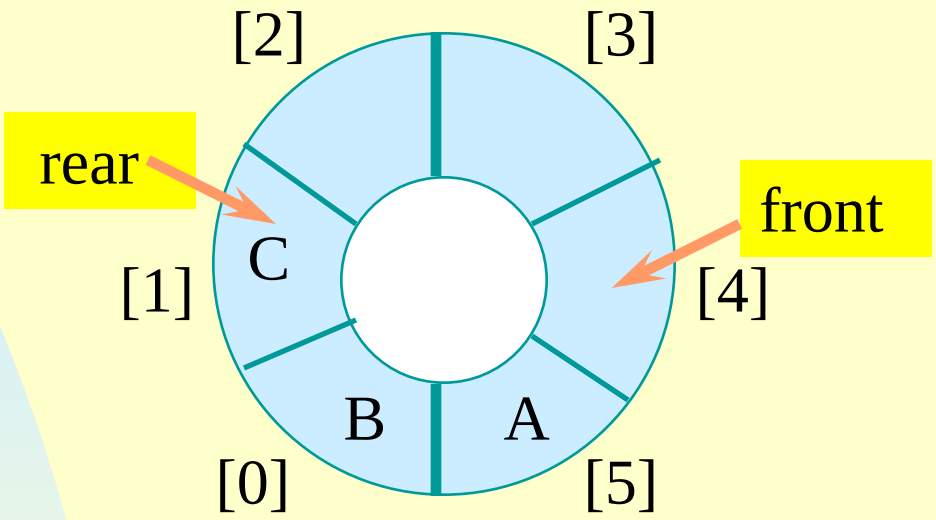


When a series of removes causes the queue to become empty, front = rear.

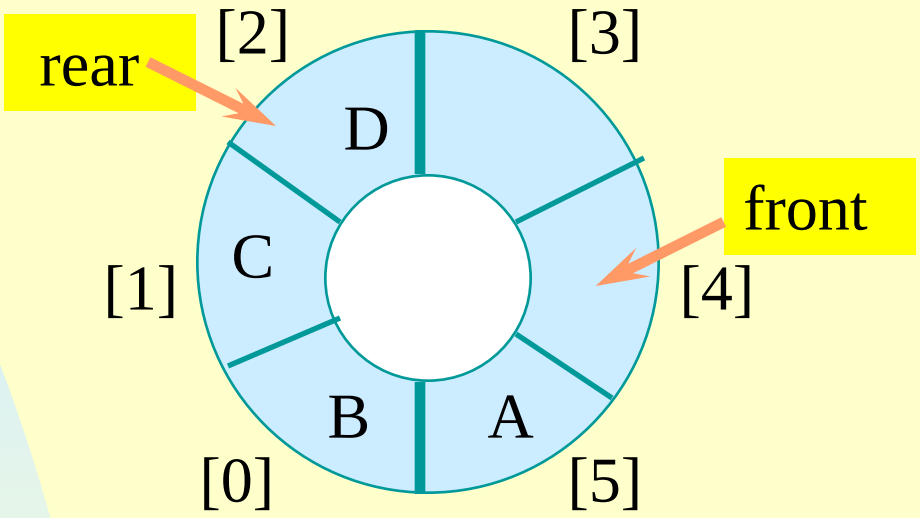
When a queue is constructed, it is empty.

So initialize front = rear = 0.

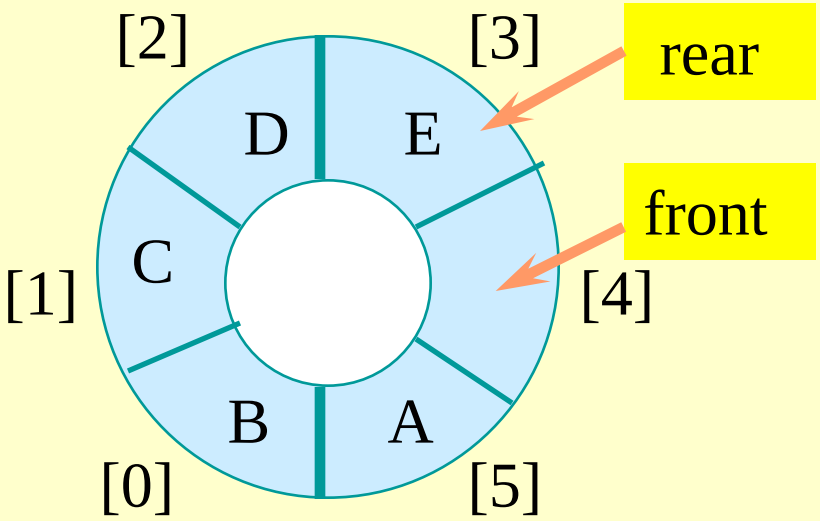
A Full Tank Please



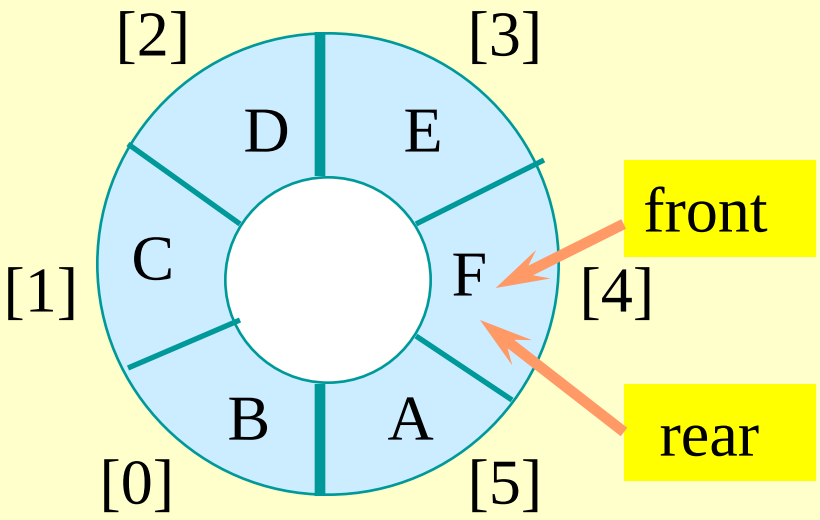
A Full Tank Please



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)

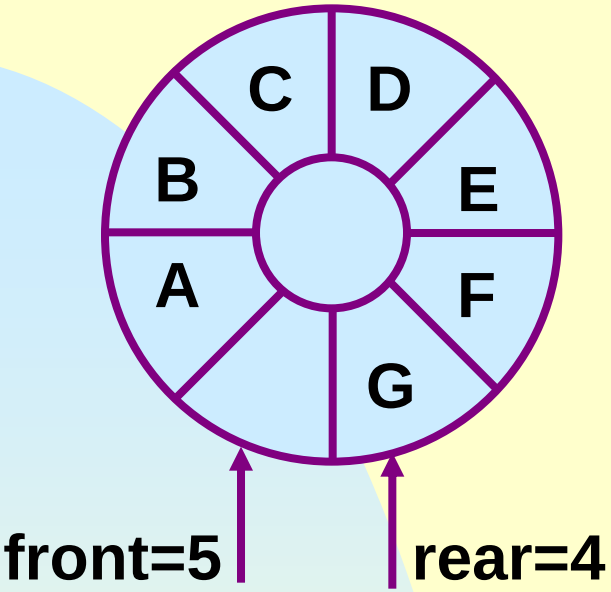
Representation of Circular Queue

```
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

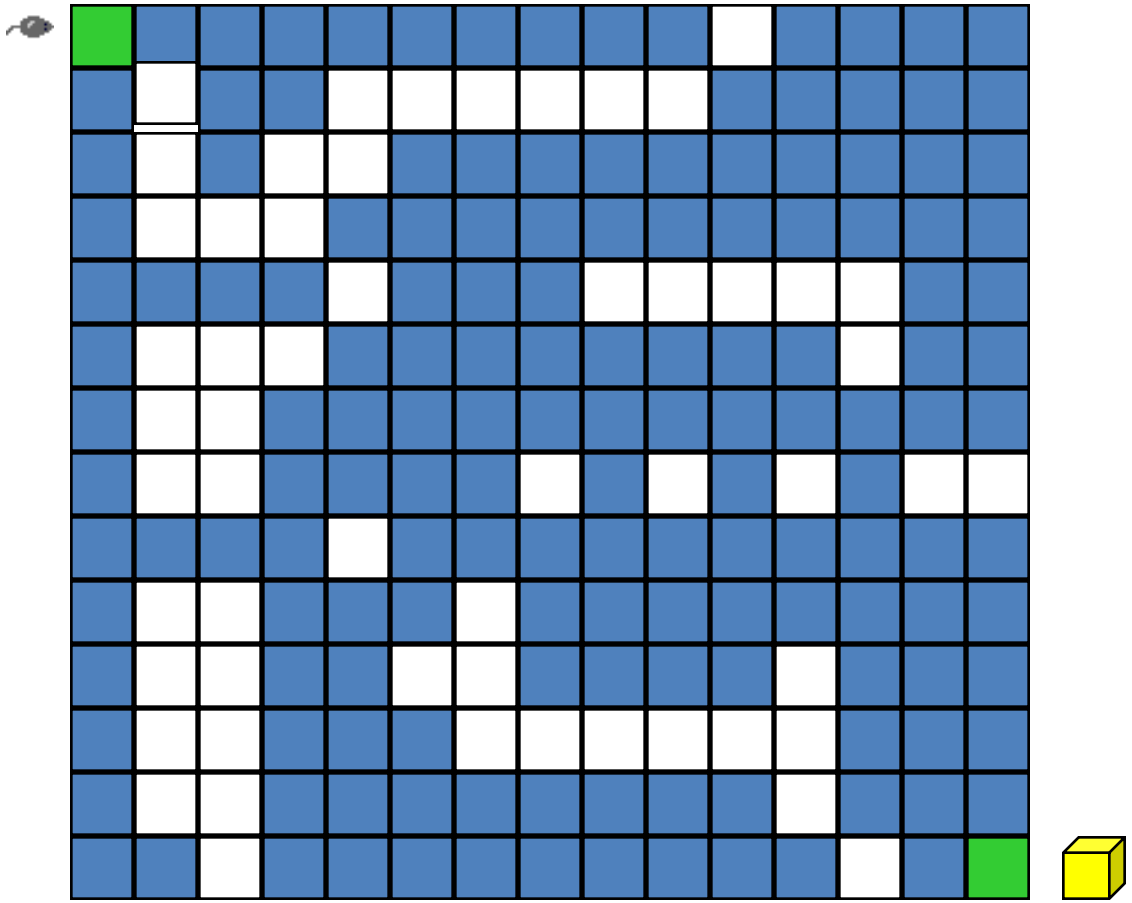
```
// allocate an array with twice the capacity
T*  newQueue = new T[2*capacity];

// 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();
}
```

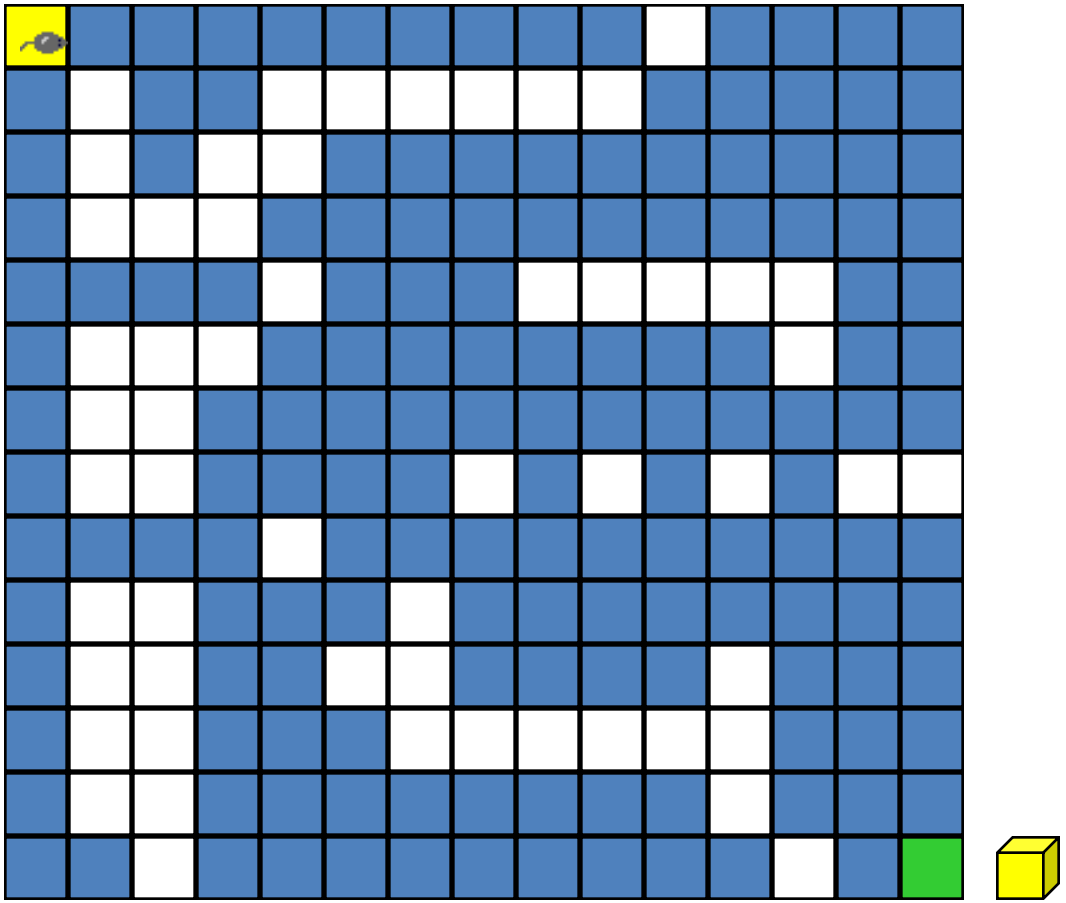
For the circular representation, the worst-case add and delete times (assuming no array resizing is needed) are O(1).

Exercises: P147-1, 3.

Rat In A Maze

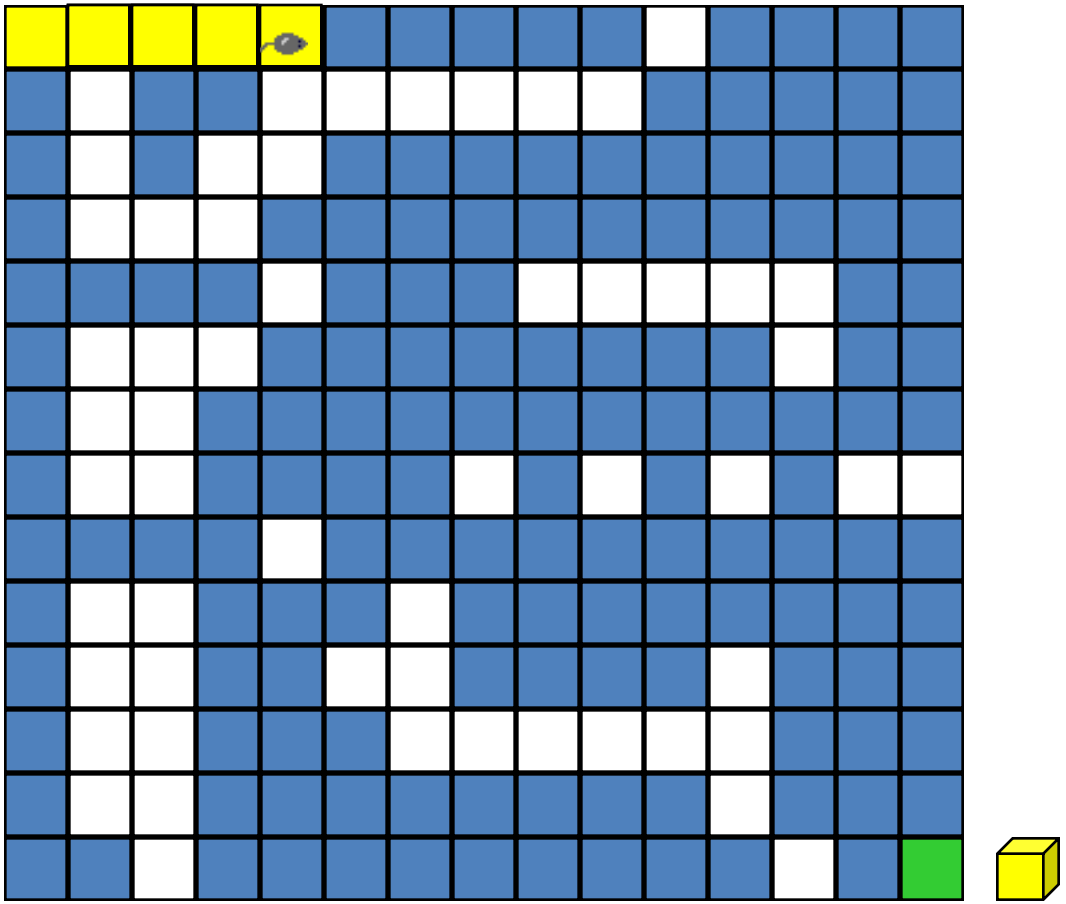


Rat In A Maze



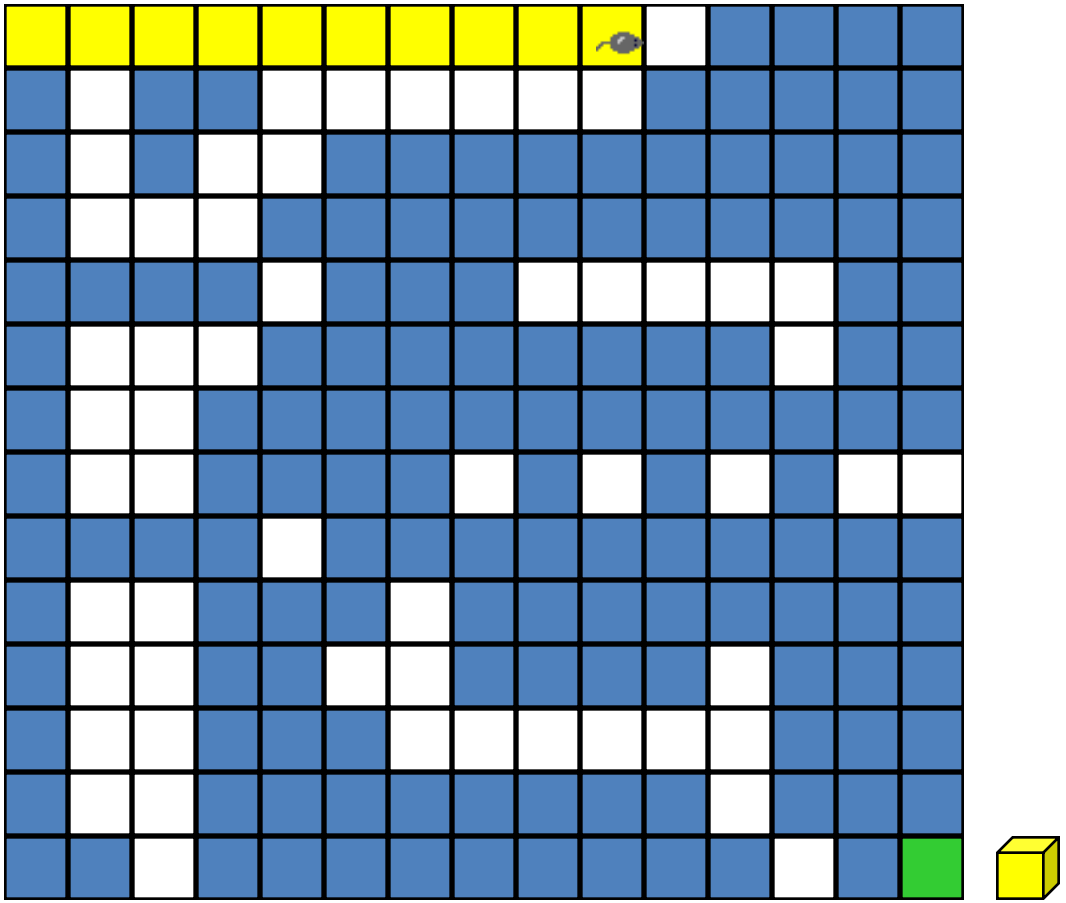
- Move order is: right, down, left, up
- Block positions to avoid revisit.

Rat In A Maze



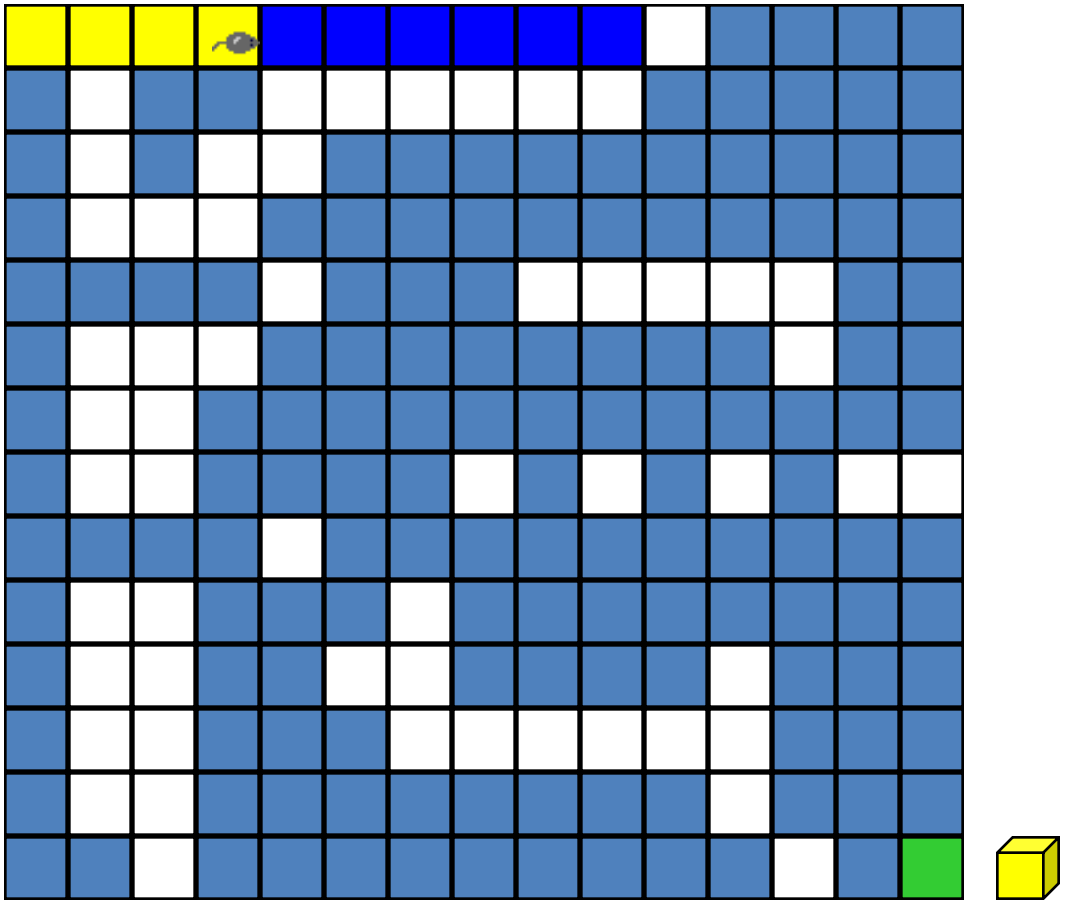
- Move order is: right, down, left, up
- Block positions to avoid revisit.

Rat In A Maze



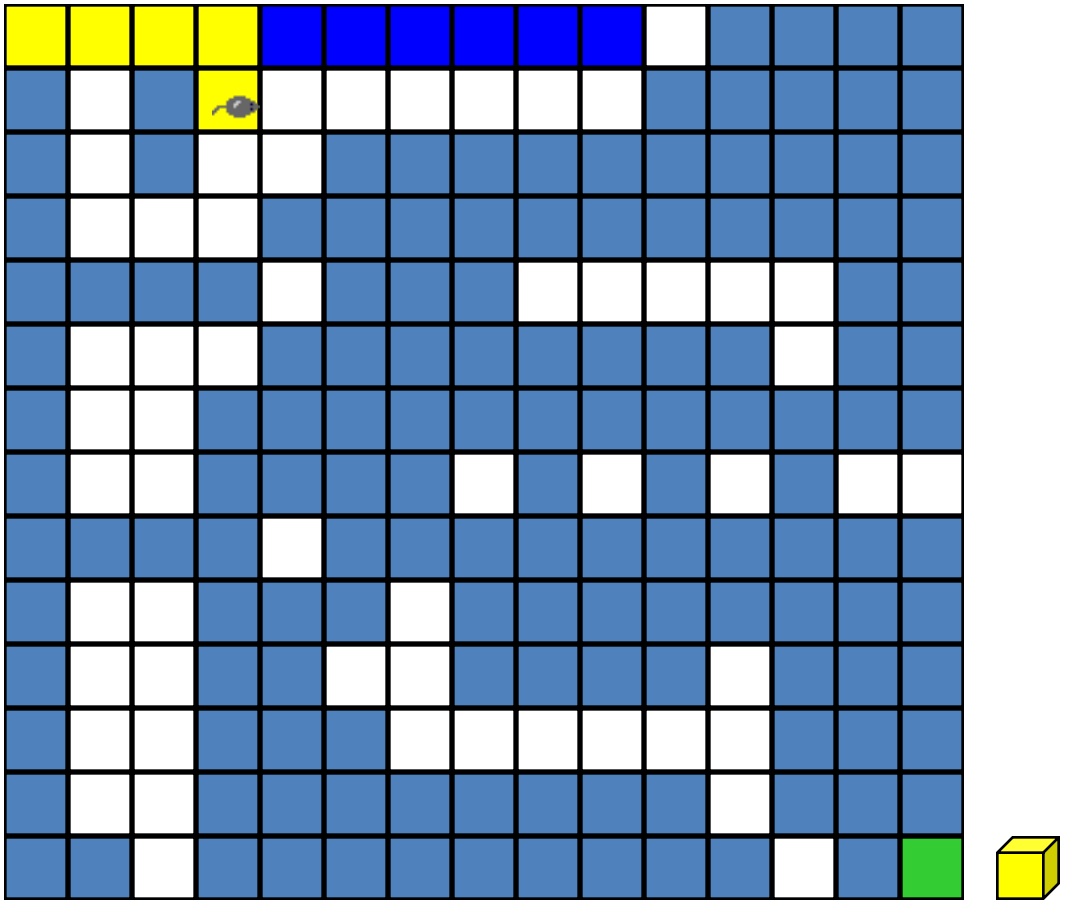
- Move backward until we reach a square from which a forward move is possible.

Rat In A Maze



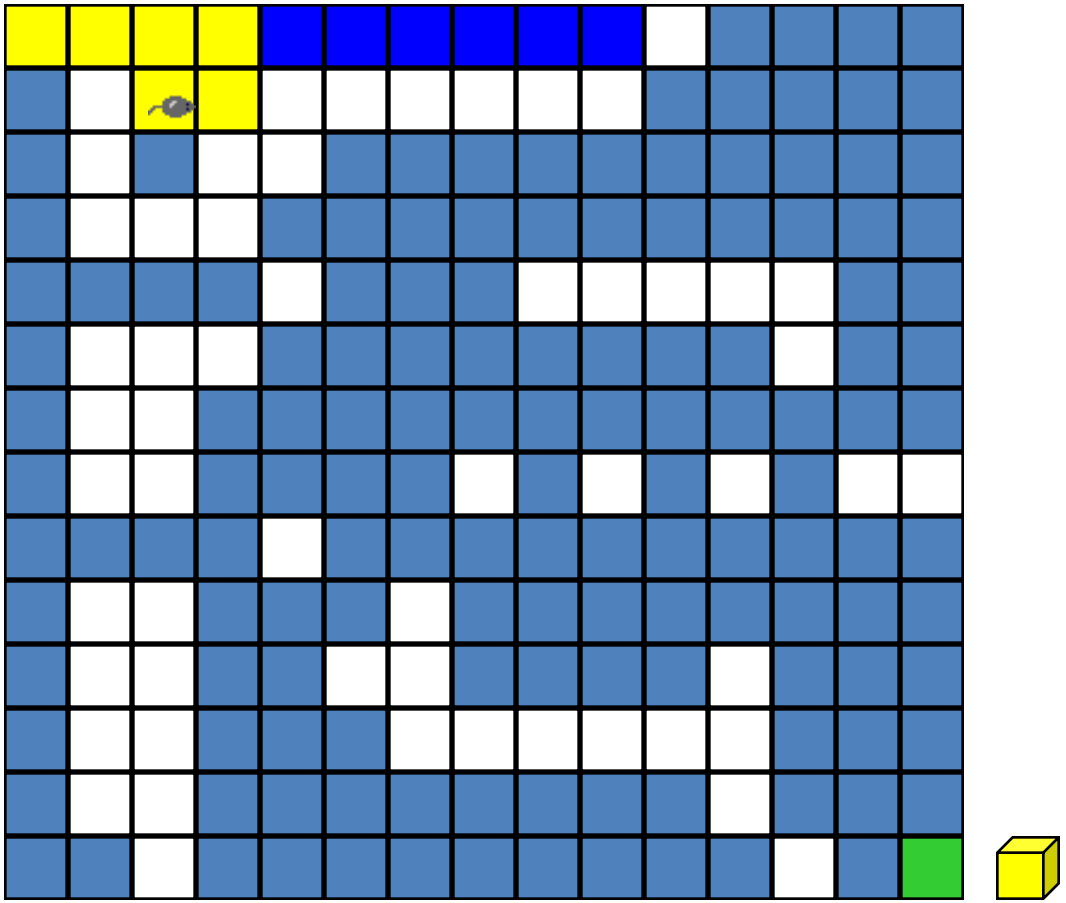
- Move down.

Rat In A Maze



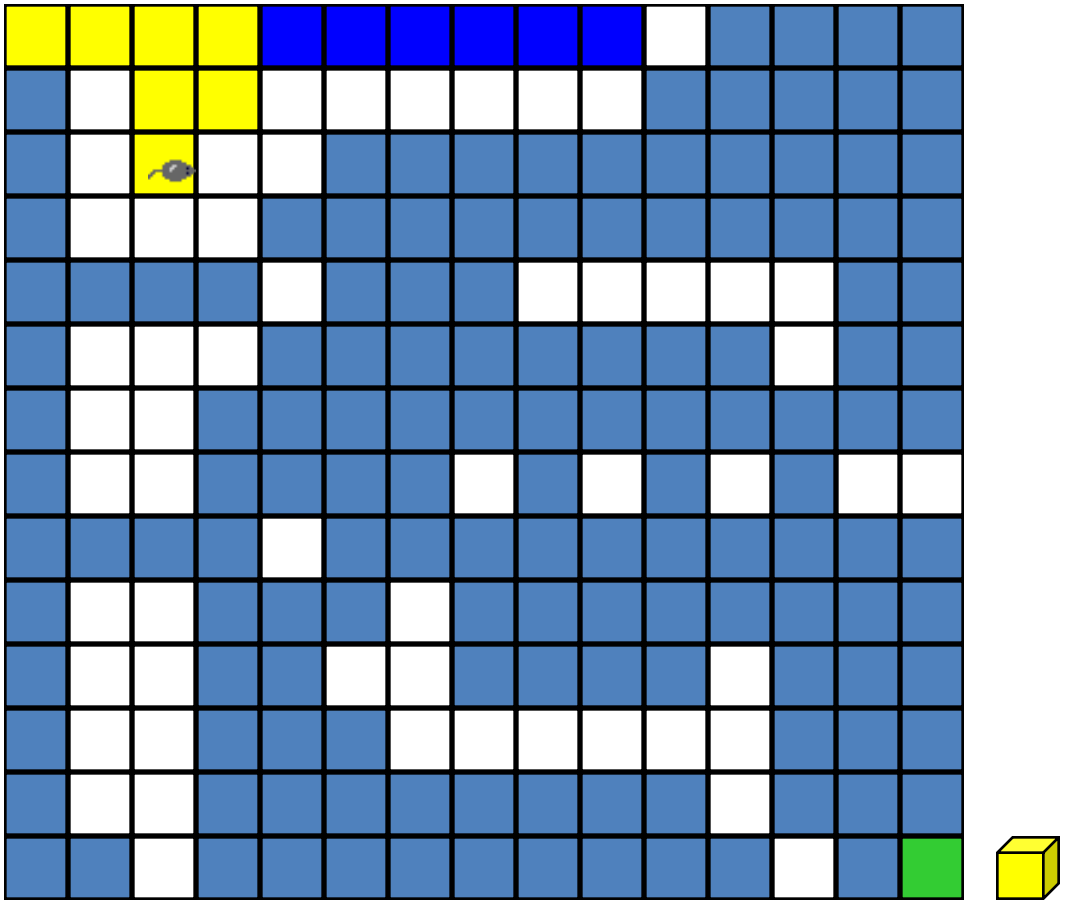
- Move left.

Rat In A Maze



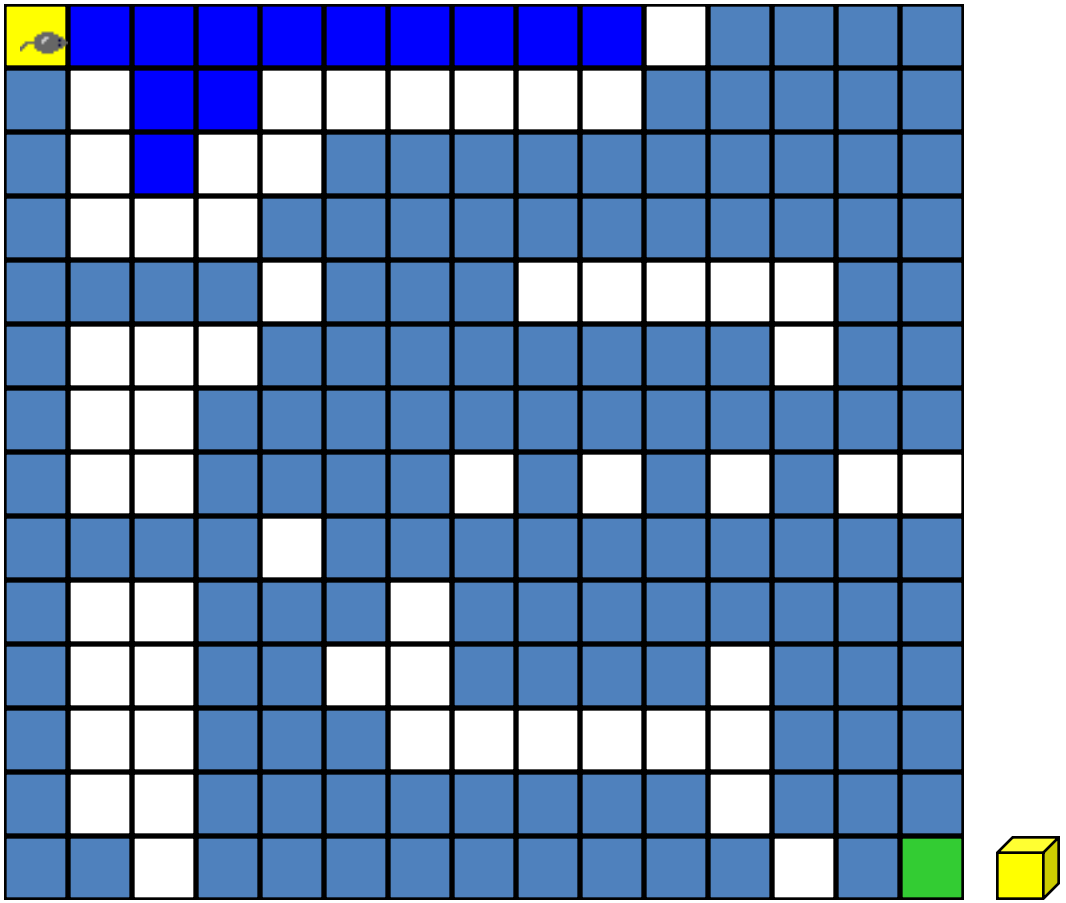
- Move down.

Rat In A Maze



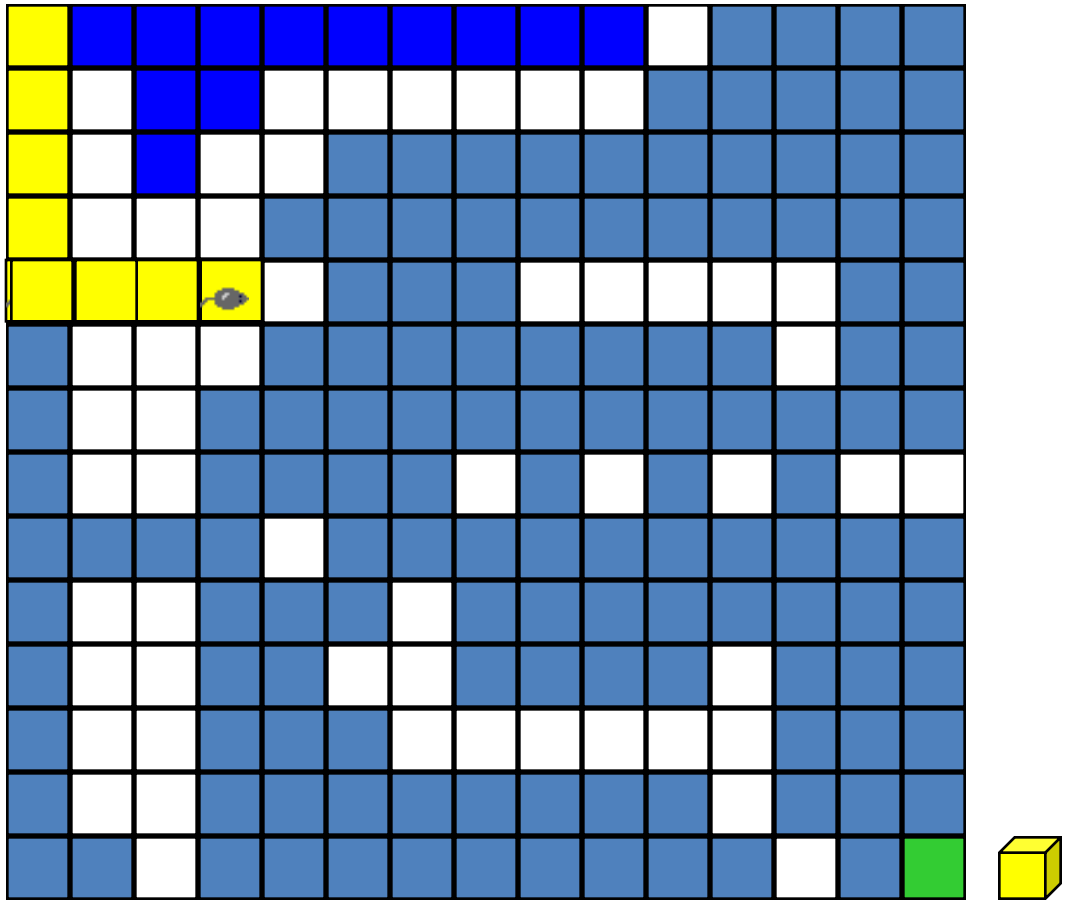
- Move backward until we reach a square from which a forward move is possible.

Rat In A Maze



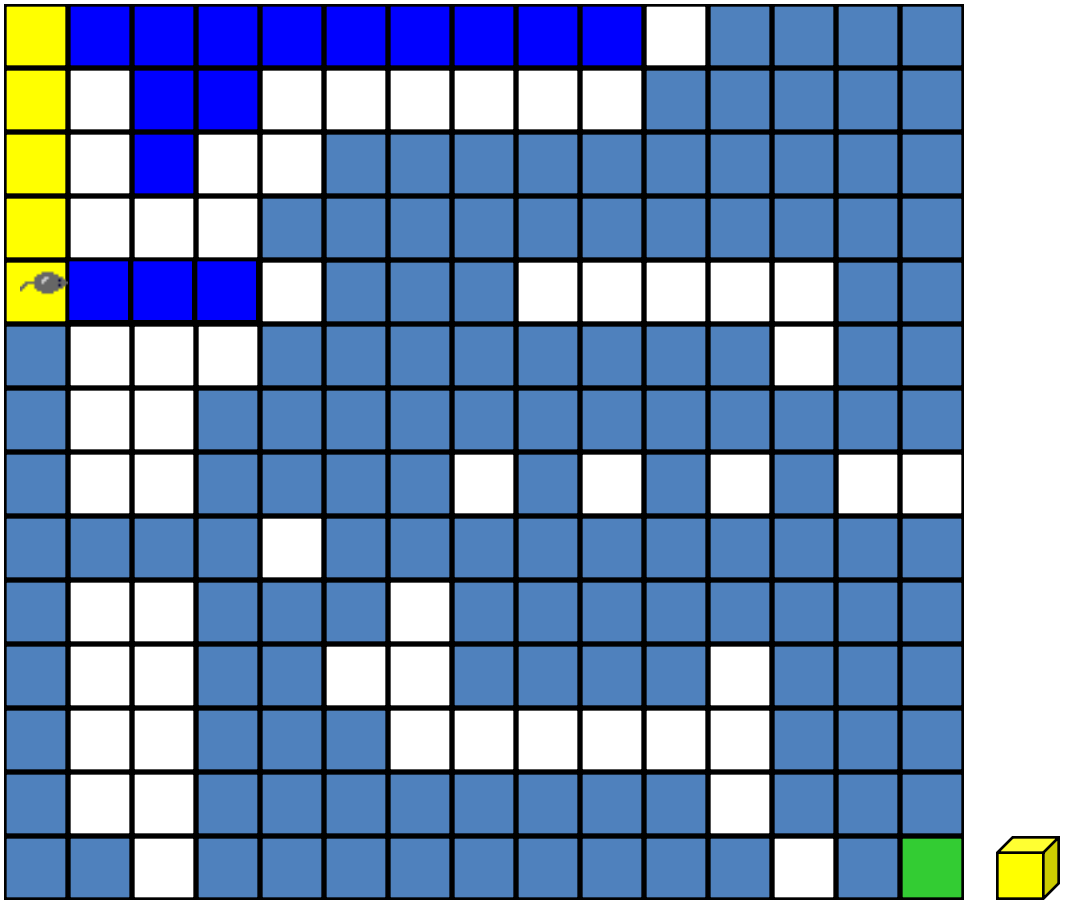
- Move backward until we reach a square from which a forward move is possible.
- Move downward.

Rat In A Maze



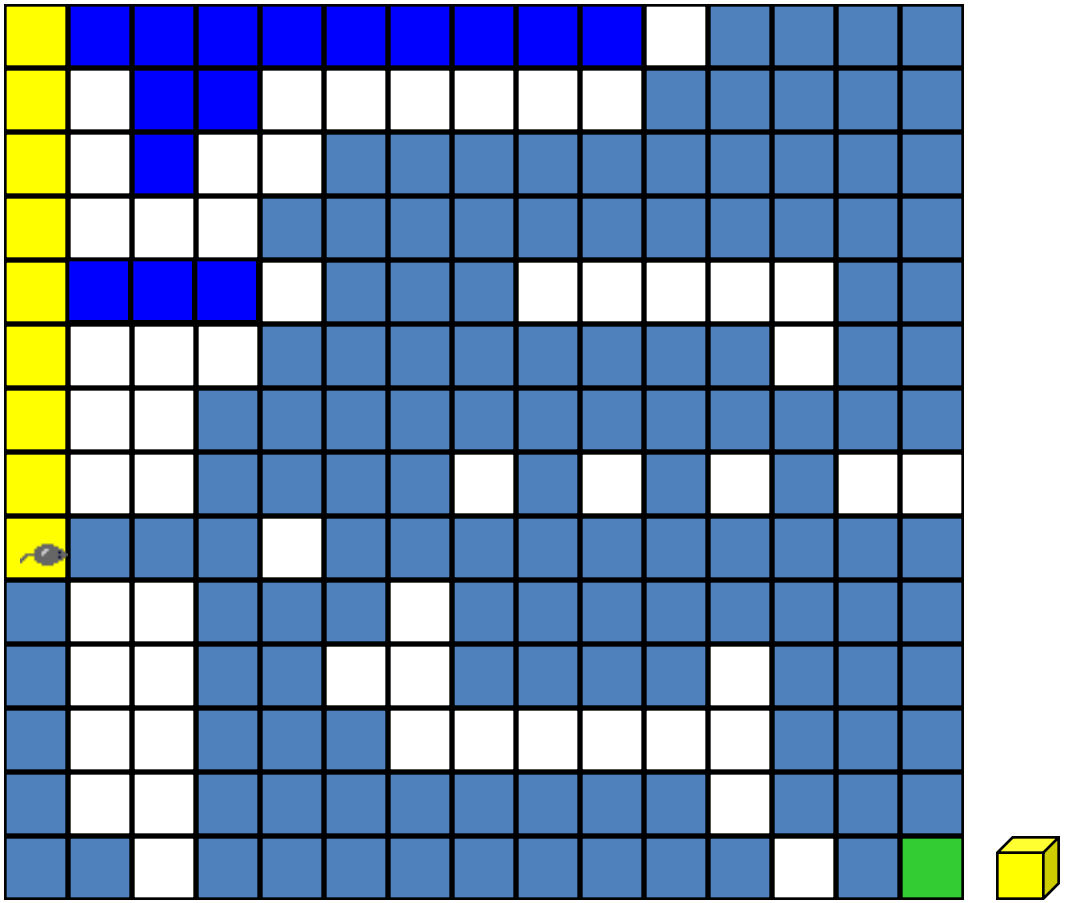
- Move right.
- Backtrack.

Rat In A Maze



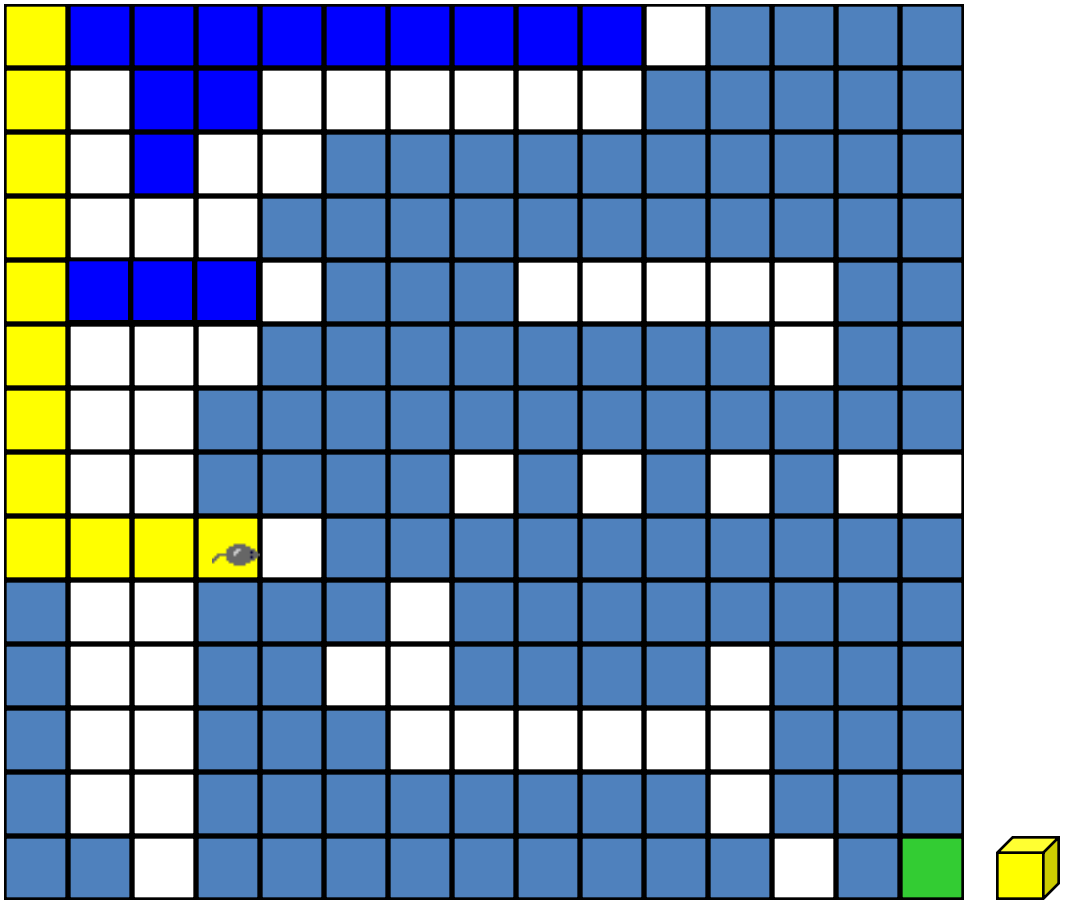
- Move downward.

Rat In A Maze



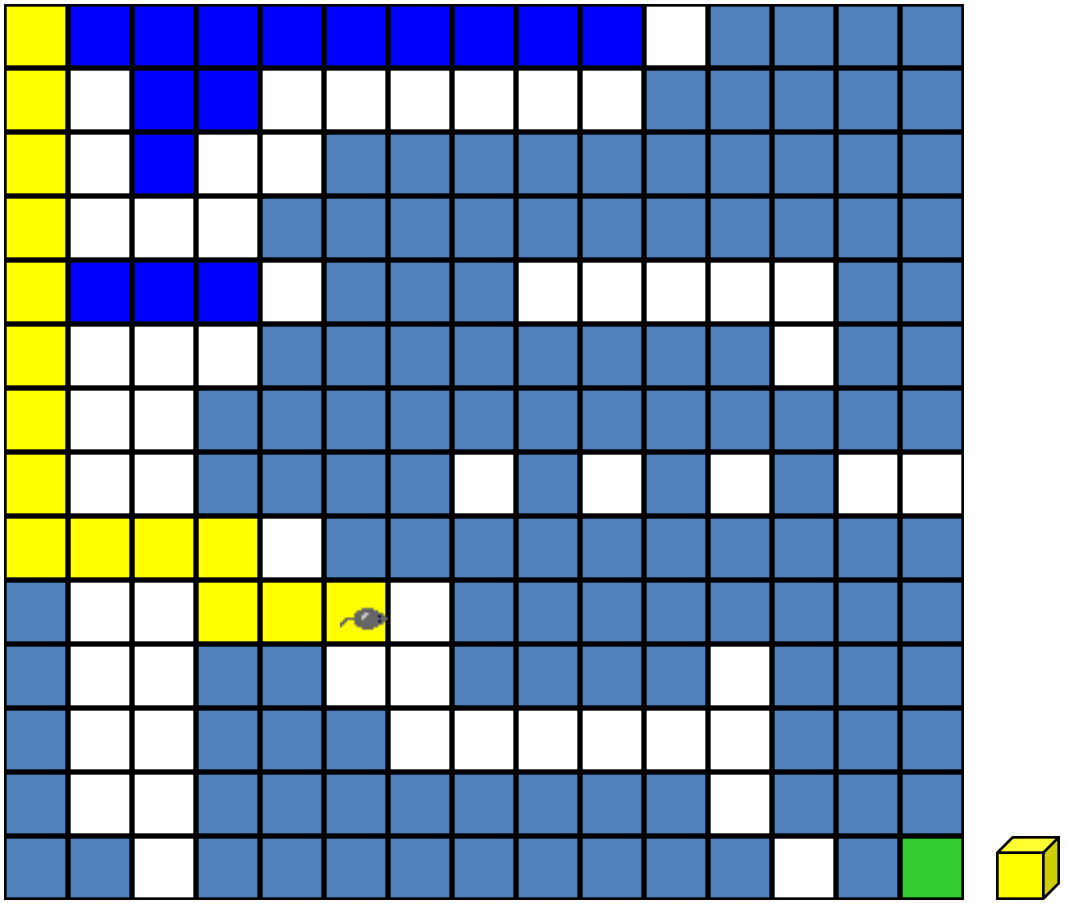
- Move right.

Rat In A Maze



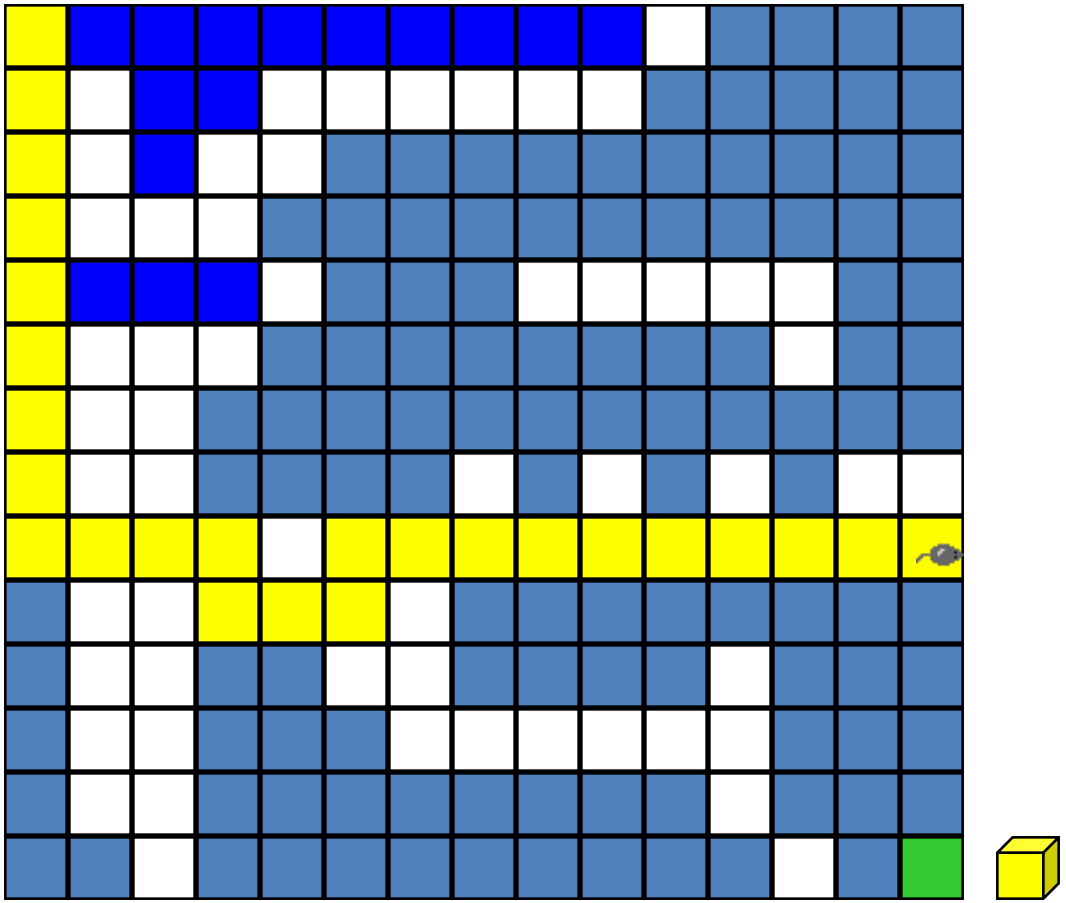
- Move one down and then right.

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
  - NEXT possible move from previous position
- Storage?
  - STACK

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

To Do: A Mazing Problem

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

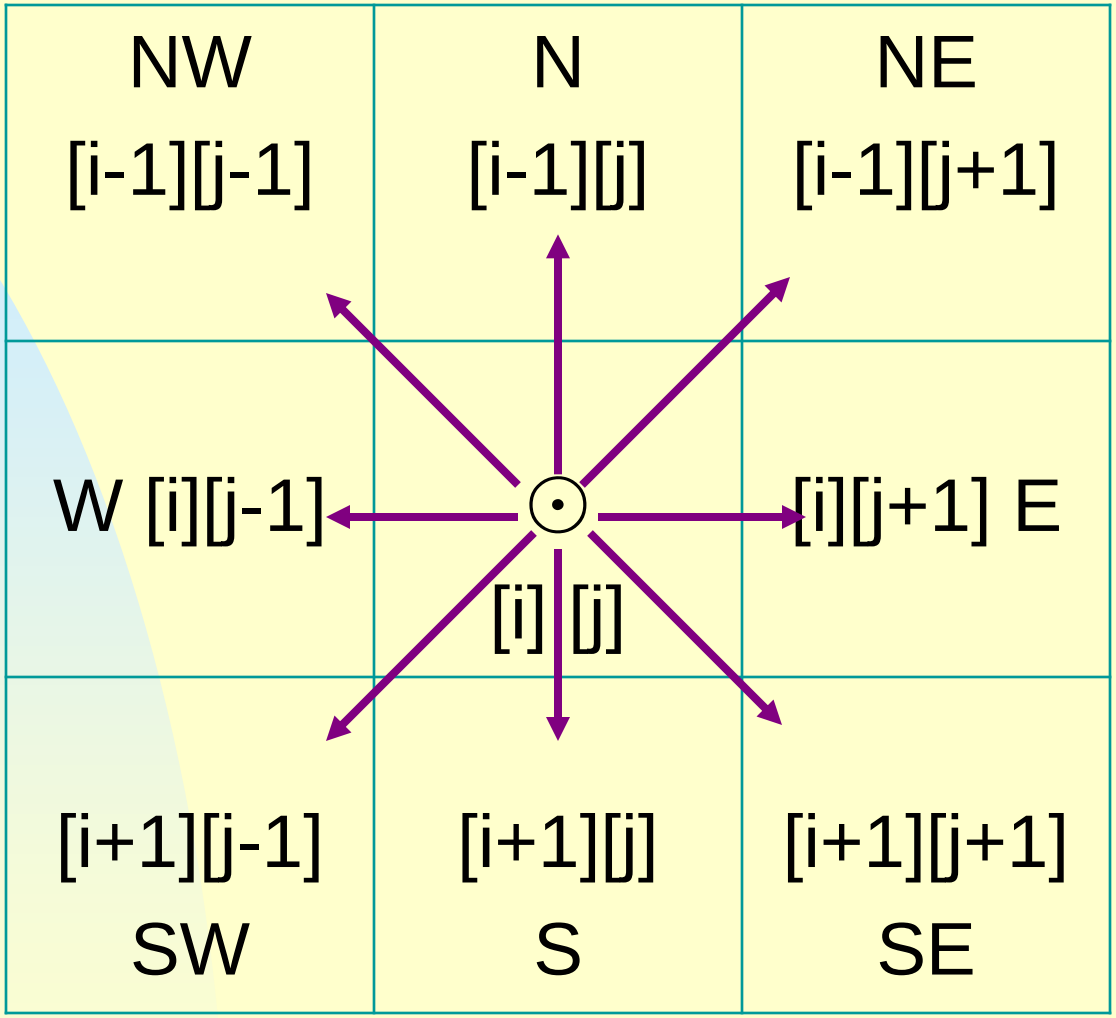
entrance

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], $1 \le i \le m$, $1 \le j \le p$.**
- **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 an accessible position, save [i][j] and d+1 in a stack , and make [g][h] be the new current position

If [g][h] is not an accessible position (take a false path) , need to try another direction. If all the directions are tried and failed, top and pop an element.

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; //output the path

```
        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.

If array capacity during stack pushing

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

Arithmetic Expressions

(a + b) * (c + d) + e – f/g*h + 3.25

Expressions comprise three kinds of entities.

- Operators (+, -, /, *).
- Operands (a, b, c, d, e, f, g, h, 3.25, (a + b), (c + d), etc.).
- Delimiters ((,)).

Operator Degree

Number of operands that the operator requires.

Binary operator requires two operands.

a + b

c / d

e - f

Unary operator requires one operand.

+ g

- h

Infix Form

Normal way to write an expression.

Binary operators come in between their left and right operands.

a * b

a + b * c

a * b / c

(a + b) * (c + d) + e – f/g*h + 3.25

Operator Priorities

How do you figure out the operands of an operator?

a + b * c

a * b + c / d

This is done by assigning operator priorities.

priority(*) = priority(/) > priority(+) = priority(-)

When an operand lies between two operators, the operand associates with the operator that has higher priority.

Tie Breaker

When an operand lies between two operators that have the same priority, the operand associates with the operator on the left.

a + b - c

a * b / c / d

Delimiters

Subexpression within delimiters is treated as a single operand, independent from the remainder of the expression.

(a + b) * (c – d) / (e – f)

Infix Expression Is Hard To Parse

Need operator priorities, tie breaker, and delimiters.

This makes computer evaluation more difficult than is necessary.

Postfix and prefix expression forms do not rely on operator priorities, a tie breaker, or delimiters.

So it is easier for a computer to evaluate expressions that are in these forms.

Postfix Form

The postfix form of a variable or constant is the same as its infix form.

a, b, 3.25

The relative order of operands is the same in infix and postfix forms.

Operators come immediately after the postfix form of their operands.

Infix = a + b

Postfix = ab+

Postfix Examples

Infix = a + b * c

Postfix = a b c * +

- Infix = a * b + c

  Postfix = a b * c +

- Infix = (a + b) * (c – d) / (e + f)
  - Postfix = a b + c d - * e f + /

Unary Operators

Replace with new symbols.

+ a => a @

+ a + b => a @ b +

- a => a ?

- a-b => a ? b -

Problem:

how to evaluate an expression?

postfix: A B / C – D E * + A C * –

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 * –

immediately passing any operands to the output

store the operators somewhere until the right time.

A*(B+C)*D → ABC+*D*

A*(B+C)*D

→ABC+*D*

Next token	storage	output

Attention

What Storage?

From the example, we can see the left parenthesis when its not in the stack behaves as an operator with high priority, whereas once it get in, it behaves as one with low priority (no operator other than the matching right parenthesis should cause it to get unstacked).

isp (in-stack priority)

icp (in-coming priority)

the isp and icp of all operators in Fig. 3.15(P.160) 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:

The function makes only a left-to-right pass across the input. The time spent on each operand is O(1).

Each operator is stacked and unstacked at most once.

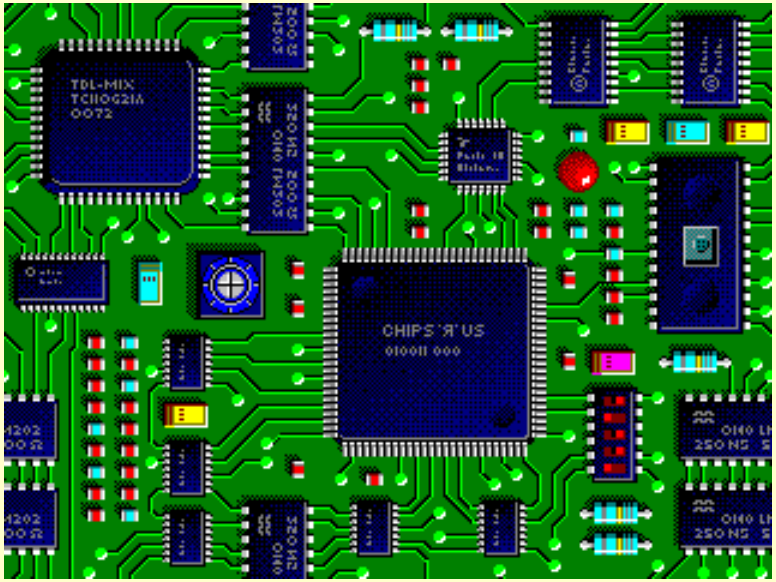
Hence, the time spent on each operator is also O(1).

So the complexity is Θ(n)

The stack will not be deeper than 1 (‘#’) + the number of operators in e.

Exercises: P165-1,2

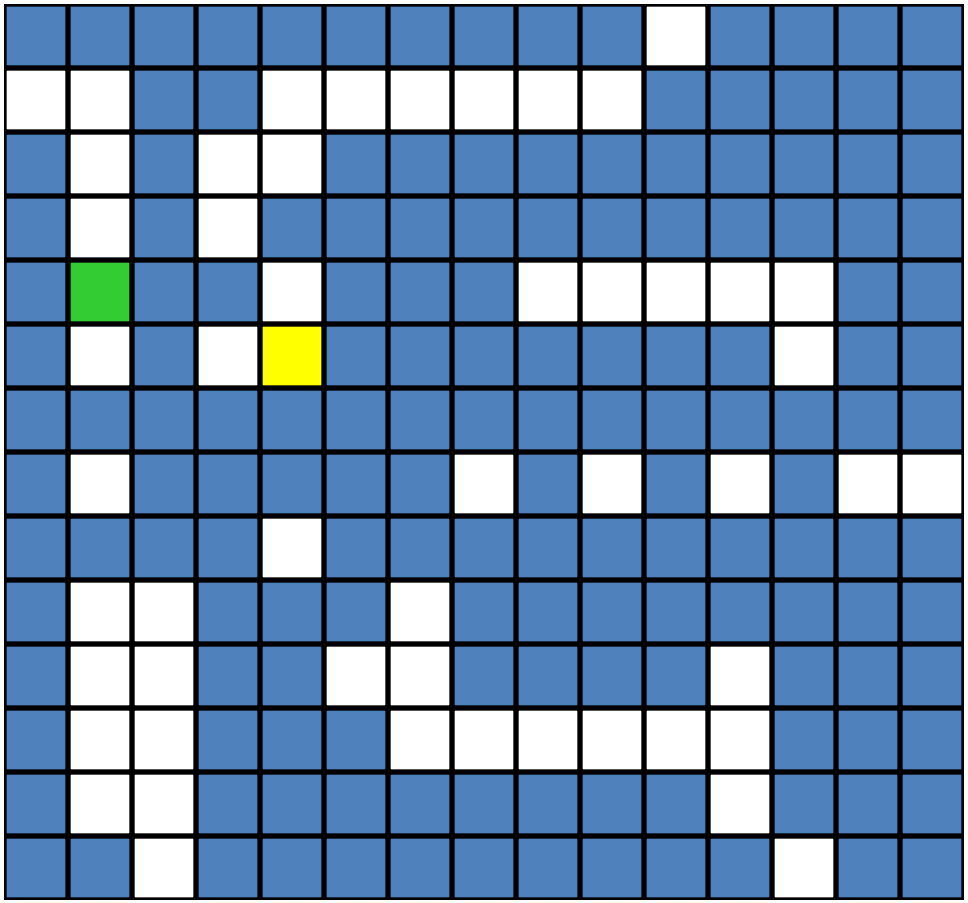
Wire Routing



Lee's Wire Router

start pin

end pin

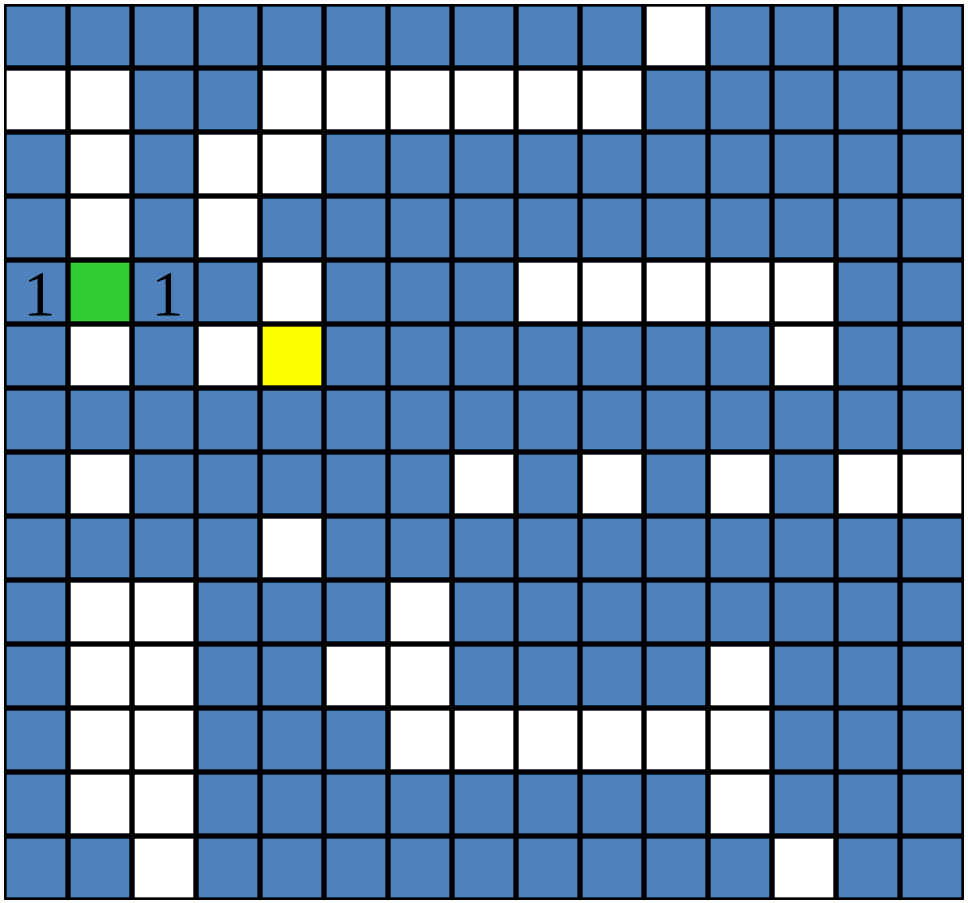


Label all reachable squares 1 unit from start.

Lee's Wire Router

start pin

end pin

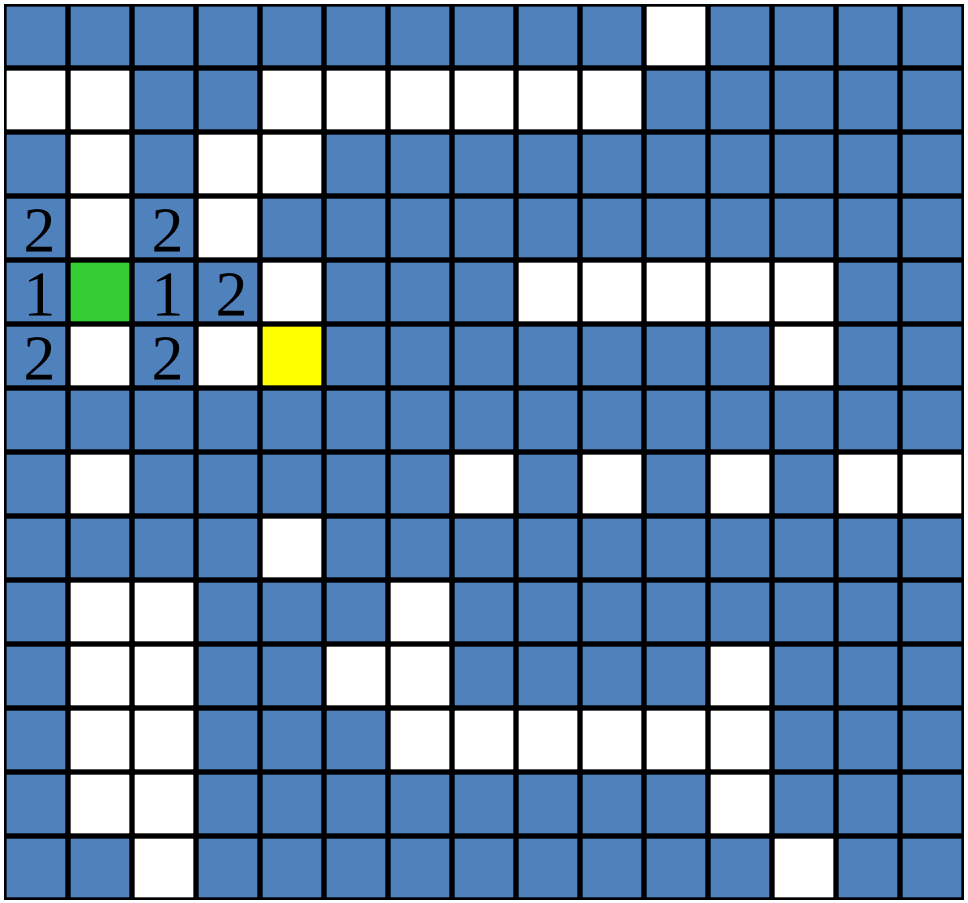


Label all reachable unlabeled squares 2 units from start.

Lee's Wire Router

start pin

end pin

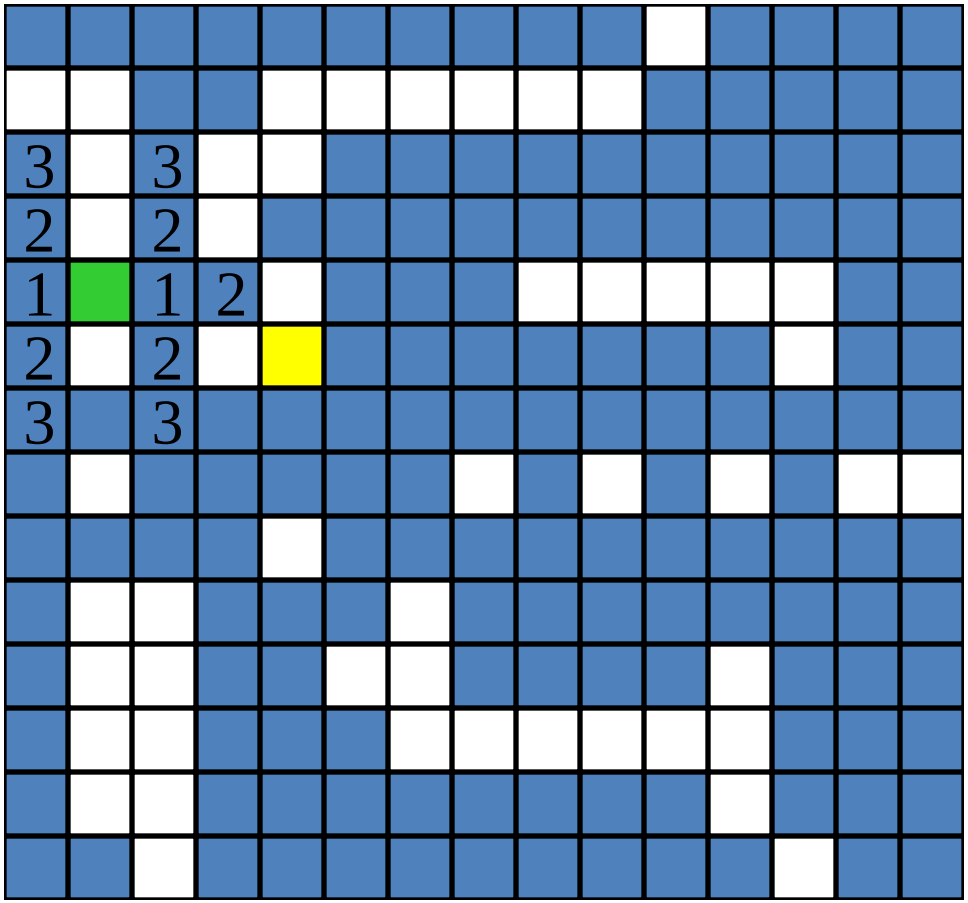


Label all reachable unlabeled squares 3 units from start.

Lee's Wire Router

start pin

end pin

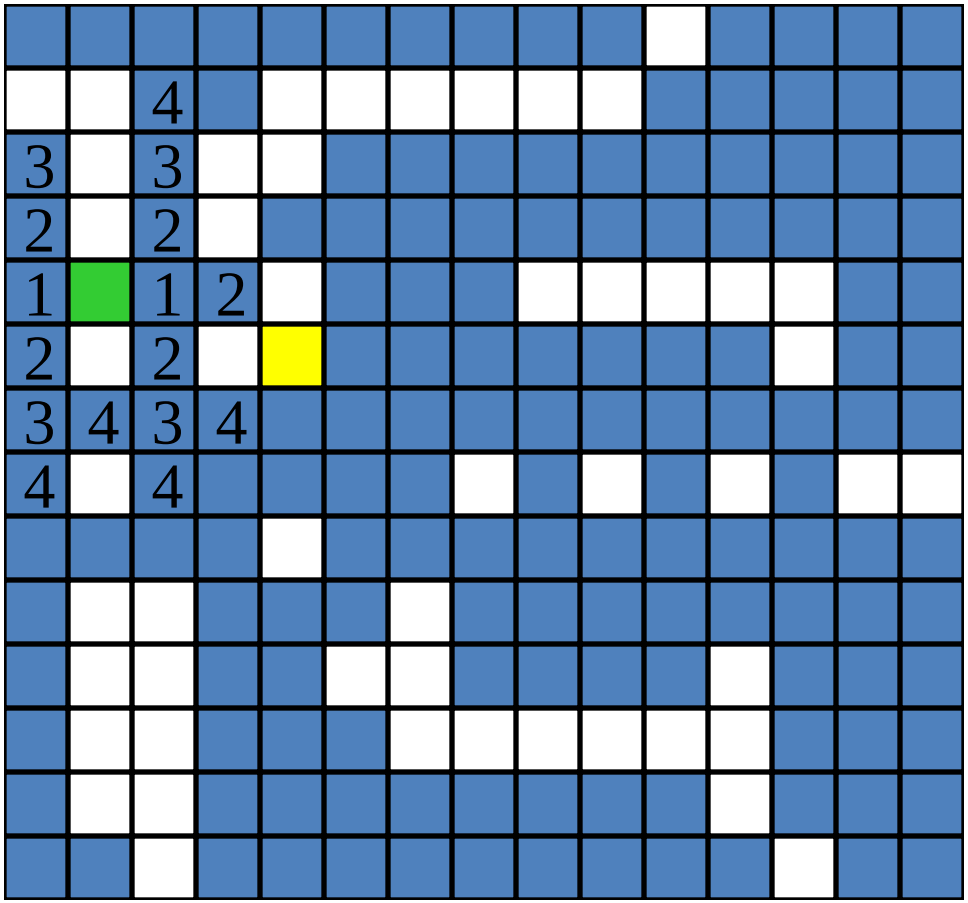


Label all reachable unlabeled squares 4 units from start.

Lee's Wire Router

start pin

end pin

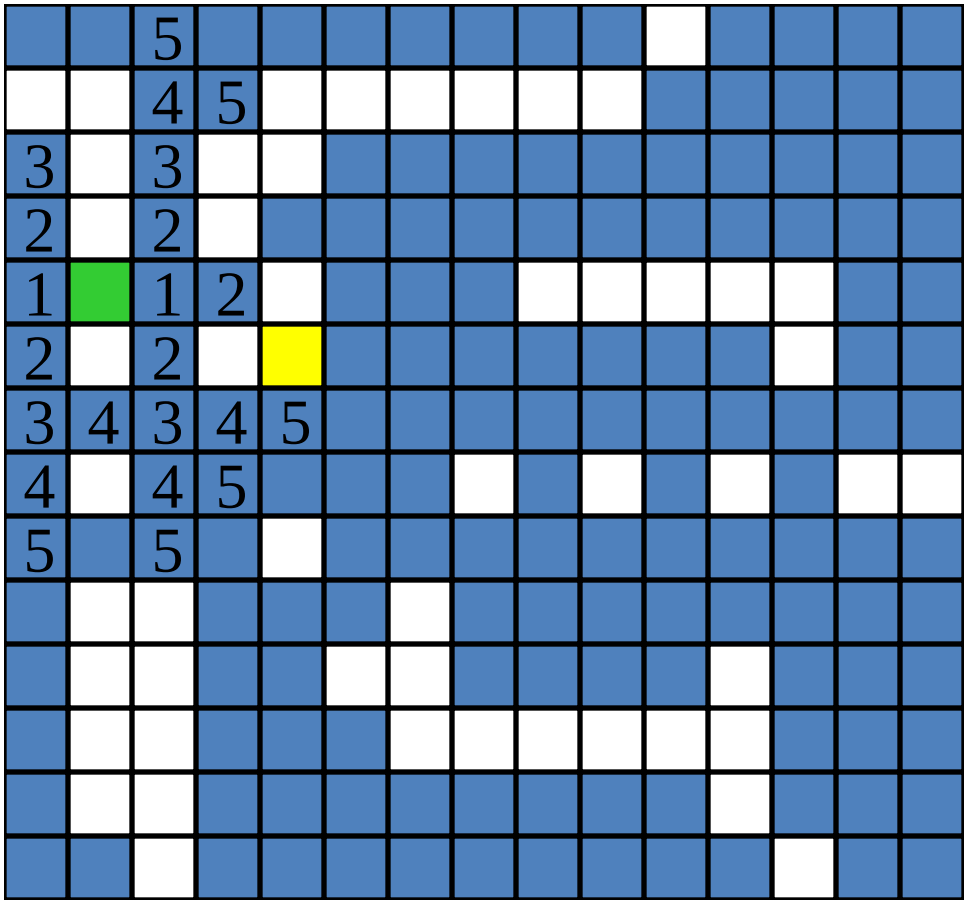


Label all reachable unlabeled squares 5 units from start.

Lee's Wire Router

start pin

end pin

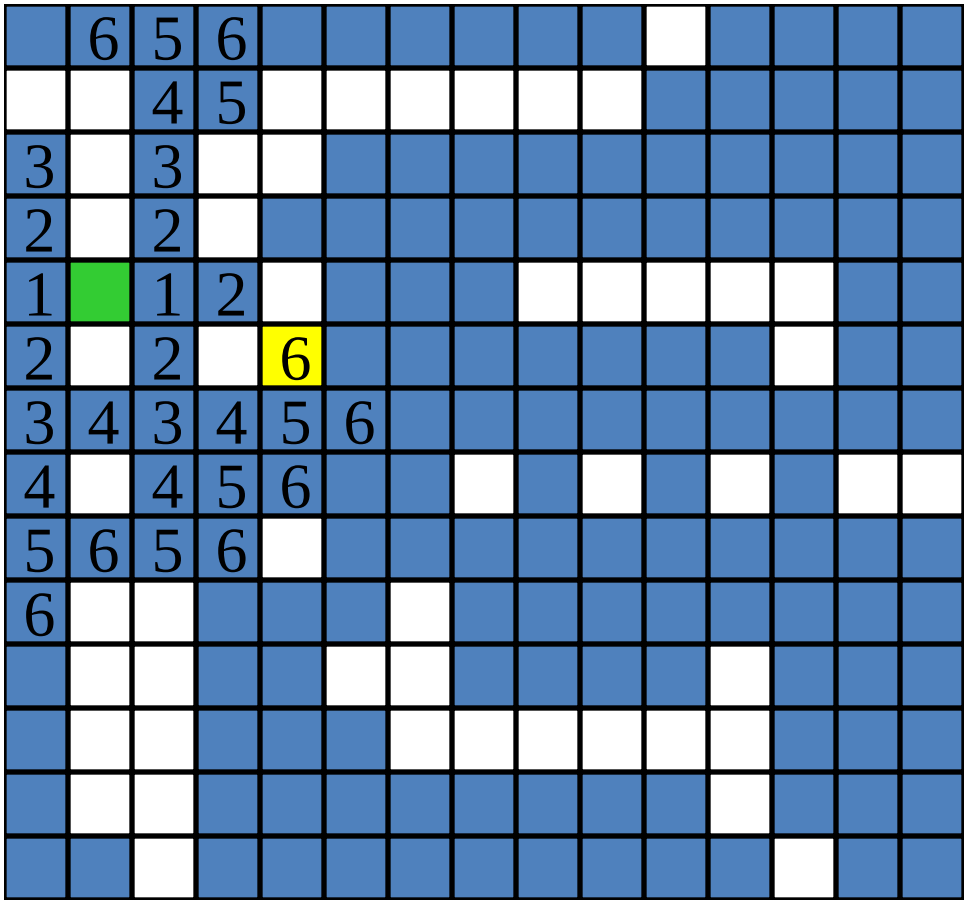


Label all reachable unlabeled squares 6 units from start.

Lee's Wire Router

start pin

end pin



End pin reached. Traceback.

```
typedef struct {
    int row;
    int col;
} Position;
Position offset[4];
offset[0].row = 0;      offset[0].col = 1;     //right
offset[1].row = 1;      offset[1].col = 0;     //down
offset[2].row = 0;      offset[2].col = -1;    //left
offset[3].row = -1;     offset[3].col = 0;     //up

//set the border of grid
for(int i=0; i <= n+1; i++)//blocked
grid[0][i] = grid[m+1][i] = -2;
for(int i=0; i <= m+1; i++)//blocked
grid[i][0] = grid[i][n+1] = -2;
```

```
bool findPath(Position start, Position finish, int &pathLen, Position*
&path, int** grid, unsigned int m, unsigned int n)
{
   if((start.row==finish.row) && (start.col==finish.col))
      {
         pathLen = 0;
         return true;
      }
   int numOfNbrs = 4;
   Position here, nbr;
   here.row = start.row;
   here.col = start.col;
   grid[start.row][start.col] = 0;
   queue<Position> Q;
```

```
do {
    for(int i=0; i < numOfNbrs; i++) {
        nbr.row = here.row+offset[i].row;
        nbr.col = here.col+offset[i].col;
        if(grid[nbr.row][nbr.col] == -1) {
            grid[nbr.row][nbr.col] = grid[here.row][here.col]+1;
            Q.push(nbr);
        }
        if((nbr.row==finish.row) && (nbr.col==finish.col))
            break;
    }
    if((nbr.row==finish.row) && (nbr.col==finish.col))
            break;//finished
    if(Q.empty()) return false;
    here = Q.front();
    Q.pop();
} while(true);
```

```
   //traceback
   pathLen = grid[finish.row][finish.col];
   path = new Position[pathLen];
   here = finish;
   for(int j=pathLen-1; j >= 0; j--) {
       path[j] = here;
       for(int i=0; i < numOfNbrs; i++) {
            nbr.row = here.row+offset[i].row;
            nbr.col = here.col+offset[i].col;
            if(grid[nbr.row][nbr.col] == j)
                 break;
       }
       here = nbr;
    }
    return true;
}//end of findPath
```

Analysis of findpath

- Any pin only be pushed and poped at most once
- In traceback, the time is O(the length from end to start).
- so the time complexity is O(mn)