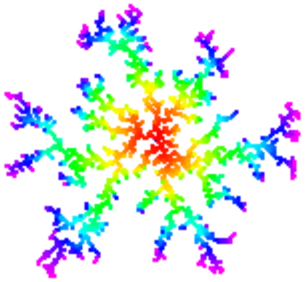
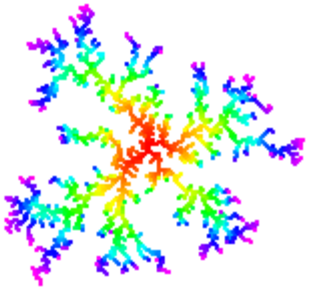
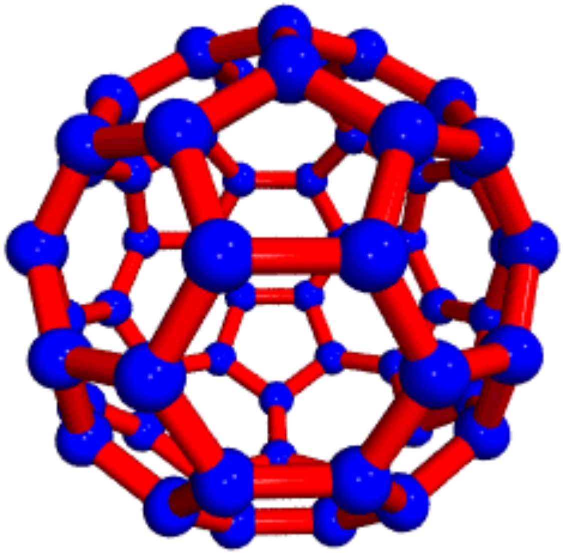
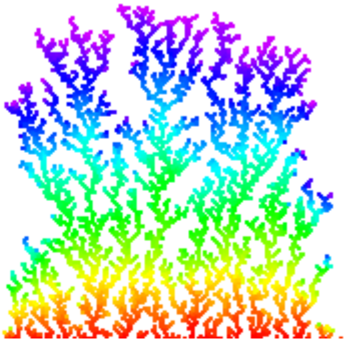


Data Structures and Algorithms

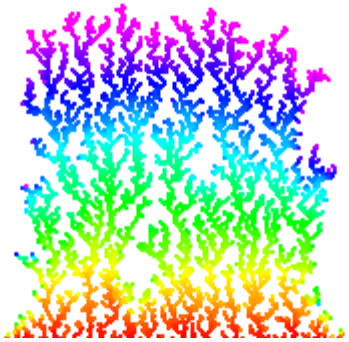






Elementary Data Structures:

Stacks and Queues

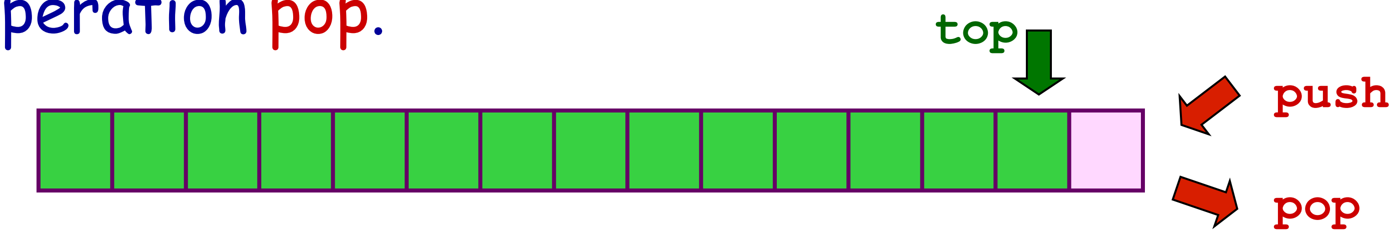


Stack and Queue ADTs

- Fundamental data types
  - Set of operations (add, remove, test if empty) on generic data.
  - Intent is clear when we insert.
  - Which object do we remove?
- Stack ("last in first out" or LIFO)
  - Remove the object most recently added.
  - Analogy: cafeteria trays, surfing Web.
- Queue ("first in first out" or FIFO)
  - Remove the object least recently added.
  - Analogy: Registrar's line.
- MultiSet
  - Remove any object.
  - Law professor calls on arbitrary student.

Stack

- A stack supports the insert and remove operations using a LIFO discipline. We name insert operation as push and the remove operation pop.



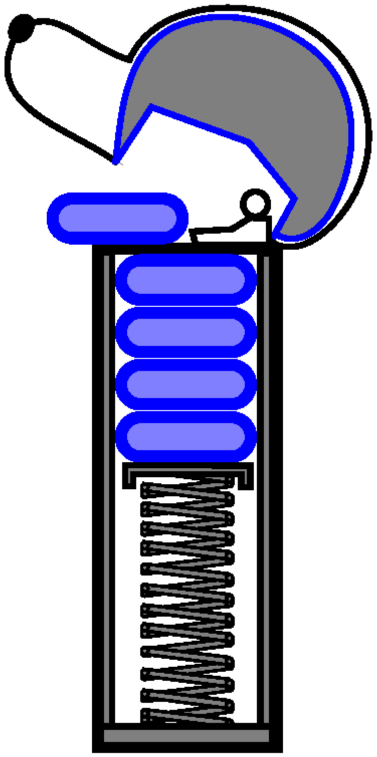
- We also include a method isEmpty to test whether or not the stack is empty. Each element in the collection can be any Java Object. Below we consider two different implementation of the following Stack interface:

```
public interface Stack {
   public boolean isEmpty();
   public Object peek();
   public void push(Object value);
   public Object pop();
}
```

A simple application of stack

- Before we describe how to implement a stack, we give a simple client program that reads in an arbitrary sequence of values and prints them in reverse order.

```
public static void main(String[] args) {
   Stack stack = new StackArray();

   while (StdIn.hasNextString()) {
      String s = StdIn.nextString();
      stack.push(s);
   }

   while (!stack.isEmpty()) {
      String s = (String) stack.pop();
      StdOut.println(s);
   }
}
```

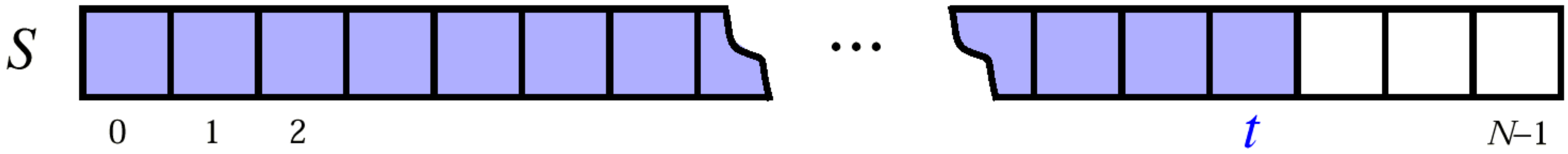


A Stack Interface in Java

```
public interface StackInterface {
   // accessor methods
   public boolean isEmpty();          // see if the stack
                                      // is empty

   public Object peek()               // return the top element
      throws StackEmptyException;     // exception if called on
                                      // an empty stack

   // update methods
   public void push (Object element); // push an element
                                      // onto the stack

   public Object pop()                // remove and return the
      throws StackEmptyException;     // top element
                                      // exception if called on
                                      // an empty stack
}
```

An Array-Based Stack Implementation

- Create a stack using an array by specifying maximum size N for our stack, e.g. N = 1,000.
- The stack consists of an N-element array S and an integer variable t, the index of the top element in array S.



- Array indices start at 0, so we initialize t to -1
- The pseudo-code follows on the next slide

Pseudo-code

```
Algorithm isEmpty( ) :
    return ($t < 0$)


Algorithm pop( ) :
    if isEmpty( ) then
        throw a StackEmptyException
    $e \leftarrow S[t]$
    $S[t] \leftarrow$ null
    $t \leftarrow t - 1$
    return $e$
```

```
Algorithm push($o$):
    if size( ) = $N$ then
        throw a StackFullException
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$


Algorithm peek( ) :
    if isEmpty( ) then
        throw a StackEmptyException
    return $S[t]$
```

An Array-Based Stack

- Each of the above methods runs in constant time
  - i.e. (O(1))

- The array implementation is simple and efficient.
- There is an upper bound, N, on the size of the stack.
  - The arbitrary value N may be too small for a given application,
  - or a waste of memory in other cases.

Array-Based Stack: a Java Implementation

```
public class StackArray implements StackInterface {
   // Implementation of the Stack interface
   // using an array.

   // default capacity of the stack
   public static final int CAPACITY = 1000;

   // maximum capacity of the stack.
   private int capacity;

   // S holds the elements of the stack
   private Object S[];

   // the top element of the stack.
   private int top = -1;
```

Array-Based Stack: a Java Implementation

```
public StackArray(int cap) {
   // Initialize the stack with given capacity
   capacity = cap;
   S = new Object[capacity];
}

public StackArray() {
   // Initialize the stack with default capacity
   this(CAPACITY);
}

public boolean isEmpty() {
   // Return true iff the stack is empty
   return (top < 0);
}
```

Array-Based Stack: a Java Implementation

```
public void push(Object obj) throws StackFullException
{
   // Push a new object on the stack
   if (size() == capacity)
      throw new StackFullException(“Stack overflow.”);
   S[++top] = obj;         // ++top or top++
}

public Object peek() throws StackEmptyException
{
   // Return the top stack element
   if (isEmpty())
   throw new StackEmptyException(“Stack is empty.”);
   return S[top];
}
```

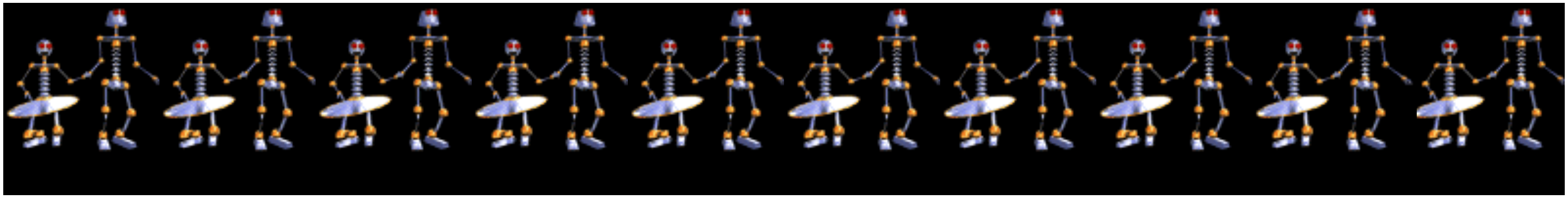
Array-Based Stack: a Java Implementation

```
   public Object pop()
      // Pop off the stack element
      throws StackEmptyException {
      Object elem;
      if (empty())
         throw new StackEmptyException(“Stack is Empty”);
      elem = S[top];
      // Dereference S[top] and decrement top
      S[top--] = null;
      return elem;
   }
}
```

Performance

500,000 pop, push, and peek operations

Class	initial capacity 10	 500,000
ArrayStack	0.44s	0.22s
DerivedArrayStack	0.60s	0.38s
DerivedArrayStackWithCatch	0.55s	0.33s
java.util.Stack	1.15s	-
DerivedLinkedStack	3.20s	3.20s
LinkedStack	2.96s	2.96s



Evaluation

- Code developed from scratch will run faster but will take more time (cost) to develop.
- Tradeoff between software development cost and performance.
- Tradeoff between time to market and performance.
- Could develop easy code first and later refine it to improve performance.

Stack Implementation with Arrays

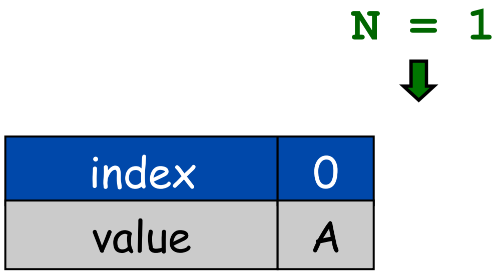
- `Stack stack = new StackArray();`

N = 0 elements on stack

index	0
value	?

Stack Implementation with Arrays

```
   Stack stack = new StackArray();
➡  stack.push("A");
```



N = 1

index	0
value	A

Stack Implementation with Arrays

```
   Stack stack = new StackArray();
   stack.push("A");
➡  stack.push("B");
```

N = 2

⬇

index	0	1
value	A	B

Double size of array

Stack Implementation with Arrays

```
   Stack stack = new StackArray();
   stack.push("A");
   stack.push("B");
➡  stack.push("C");
```

N = 3

⬇

index	0	1	2	3
value	A	B	C	?

Double size of array

Stack Implementation with Arrays

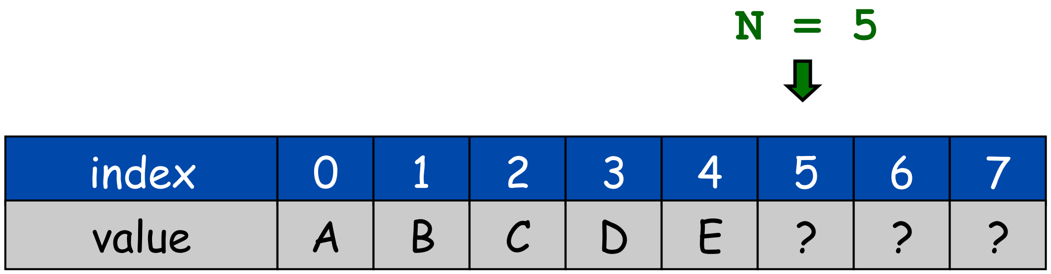
```
  Stack stack = new StackArray();
  stack.push("A");
  stack.push("B");
  stack.push("C");
⇨ stack.push("D");
```

N = 4 ⇩

index	0	1	2	3
value	A	B	C	D

Stack Implementation with Arrays

```
  Stack stack = new StackArray();
  stack.push("A");
  stack.push("B");
  stack.push("C");
  stack.push("D");
⇨ stack.push("E");
```



N = 5

index	0	1	2	3	4	5	6	7
value	A	B	C	D	E	?	?	?

Double size of array

Stack Implementation with Arrays

```
  Stack stack = new StackArray();
  stack.push("A");
  stack.push("B");
  stack.push("C");
  stack.push("D");
  stack.push("E");
➡ stack.push("F");
```

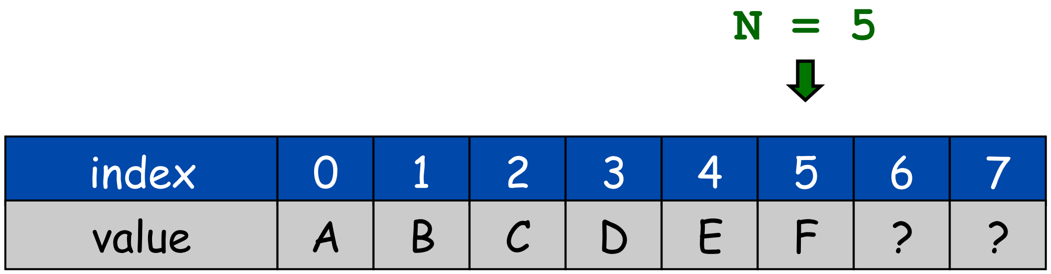
N = 6

⬇

index	0	1	2	3	4	5	6	7
value	A	B	C	D	E	F	?	?

Stack Implementation with Arrays

```
Stack stack = new StackArray();
stack.push("A");
stack.push("B");
stack.push("C");
stack.push("D");
stack.push("E");
stack.push("F");
System.out.println(stack.pop());
```



N = 5

index	0	1	2	3	4	5	6	7
value	A	B	C	D	E	F	?	?

```
% java StackArray
F
```

Stack Implementation with Arrays

```
  Stack stack = new StackArray();
  stack.push("A");
  stack.push("B");
  stack.push("C");
  stack.push("D");
  stack.push("E");
  stack.push("F");
  System.out.println(stack.pop());
➡ System.out.println(stack.pop());
```

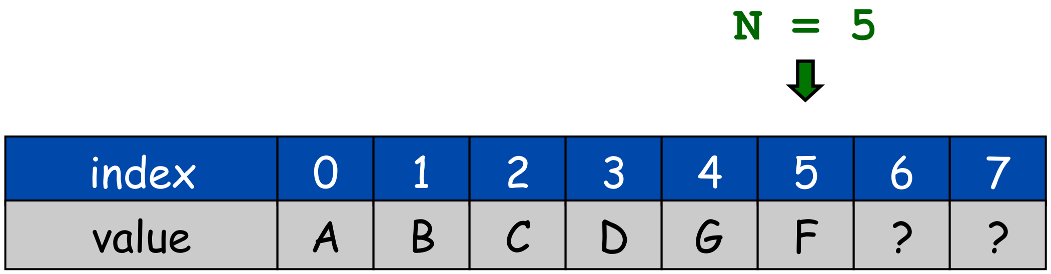
N = 4

index	0	1	2	3	4	5	6	7
value	A	B	C	D	E	F	?	?

```
% java StackArray
F
E
```

Stack Implementation with Arrays

```
   Stack stack = new StackArray();
   stack.push("A");
   stack.push("B");
   stack.push("C");
   stack.push("D");
   stack.push("E");
   stack.push("F");
   System.out.println(stack.pop());
   System.out.println(stack.pop());
➡  stack.push("G");
```



N = 5

index	0	1	2	3	4	5	6	7
value	A	B	C	D	G	F	?	?

```
% java StackArray
F
E
```

Stack Implementation with Arrays

```
  Stack stack = new StackArray();
  stack.push("A");
  stack.push("B");
  stack.push("C");
  stack.push("D");
  stack.push("E");
  stack.push("F");
  System.out.println(stack.pop());
  System.out.println(stack.pop());
  stack.push("G");
➡ System.out.println(stack.pop());
```

N = 4 ⬇

index	0	1	2	3	4	5	6	7
value	A	B	C	D	G	F	?	?

```
% java StackArray
F
E
G
```

Stack Implementation with Arrays

```
  Stack stack = new StackArray();
  stack.push("A");
  stack.push("B");
  stack.push("C");
  stack.push("D");
  stack.push("E");
  stack.push("F");
  System.out.println(stack.pop());
  System.out.println(stack.pop());
  stack.push("G");
  System.out.println(stack.pop());
➡ System.out.println(stack.pop());
```

N = 3 ⬇

index	0	1	2	3	4	5	6	7
value	A	B	C	D	G	F	?	?

```
% java StackArray
F
E
G
D
```

Stack Implementation with Arrays

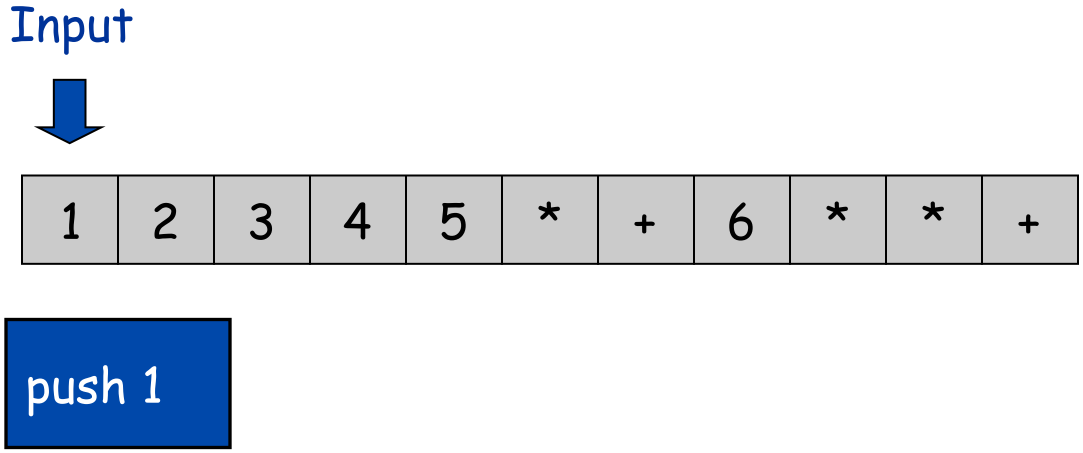
```
   Stack stack = new StackArray();
   stack.push("A");
   stack.push("B");
   stack.push("C");
   stack.push("D");
   stack.push("E");
   stack.push("F");
   System.out.println(stack.pop());
   System.out.println(stack.pop());
   stack.push("G");
   System.out.println(stack.pop());
   System.out.println(stack.pop());
➡  System.out.println(stack.pop());
```

N = 2

index	0	1	2	3	4	5	6	7
value	A	B	C	D	G	F	?	?

```
% java StackArray
F
E
G
D
C
```

Postfix Demo

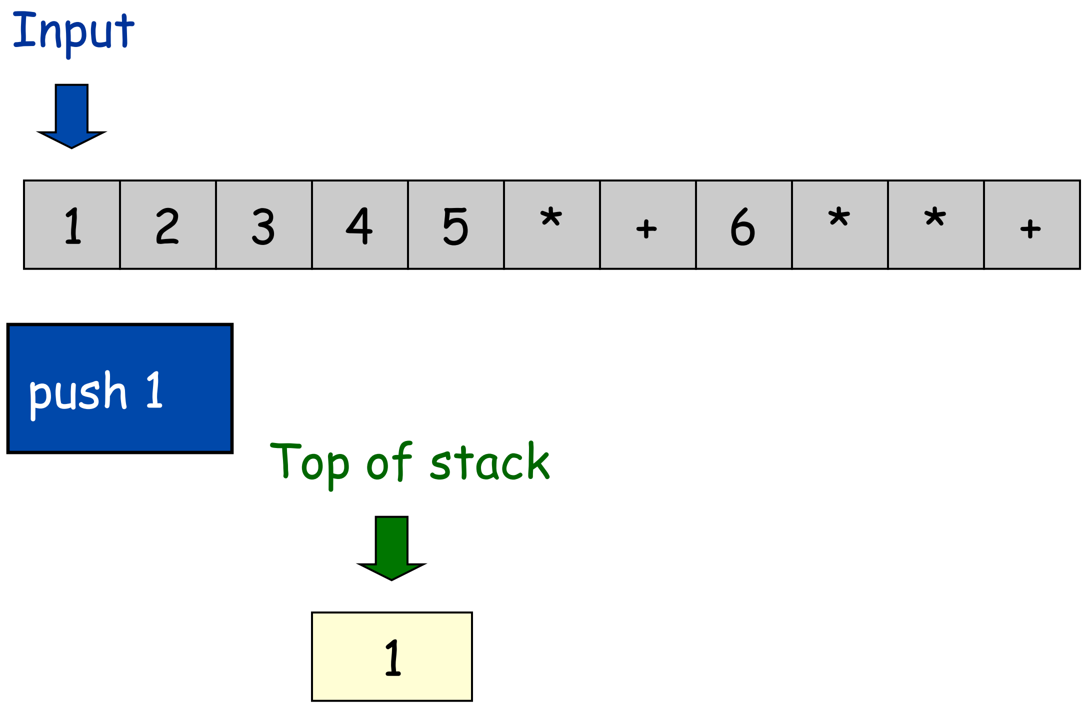


Input

1	2	3	4	5	*	+	6	*	*	+

push 1

Postfix Demo



Input

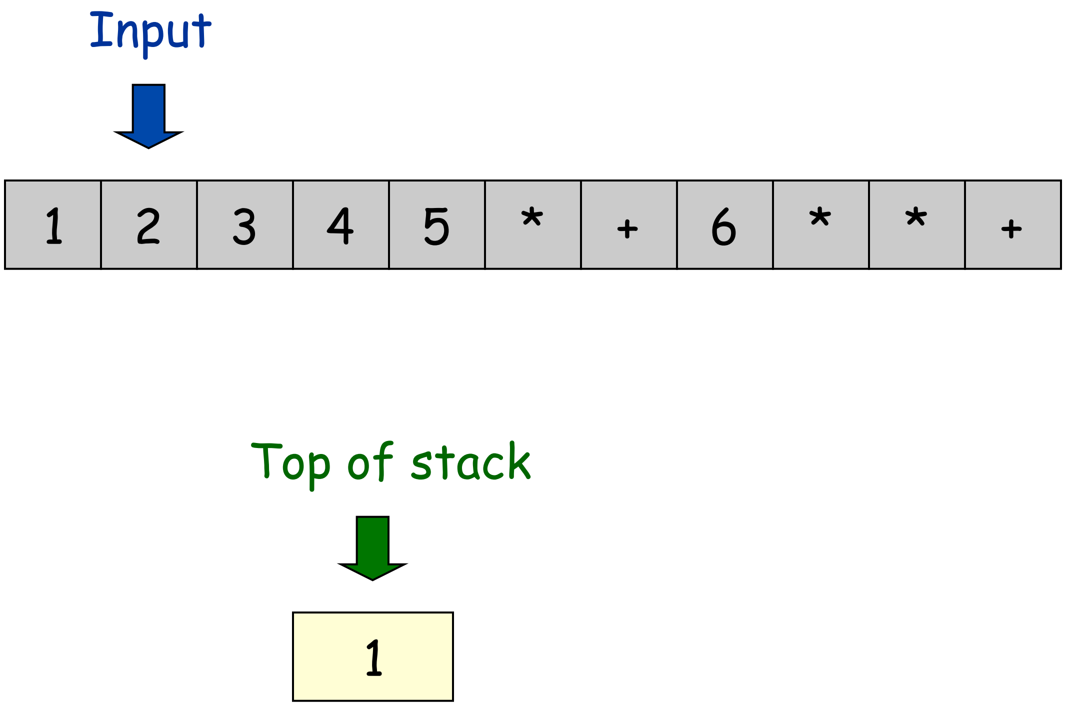
1	2	3	4	5	*	+	6	*	*	+

push 1

Top of stack

1

Postfix Demo



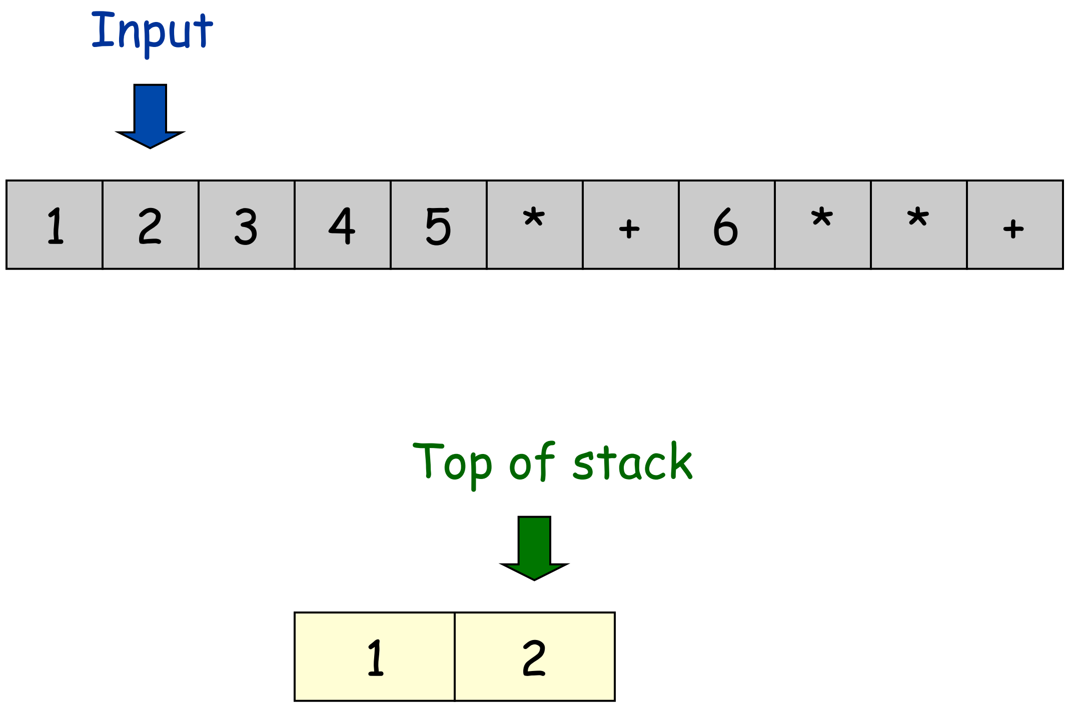
Input

1	2	3	4	5	*	+	6	*	*	+

Top of stack

1

Postfix Demo



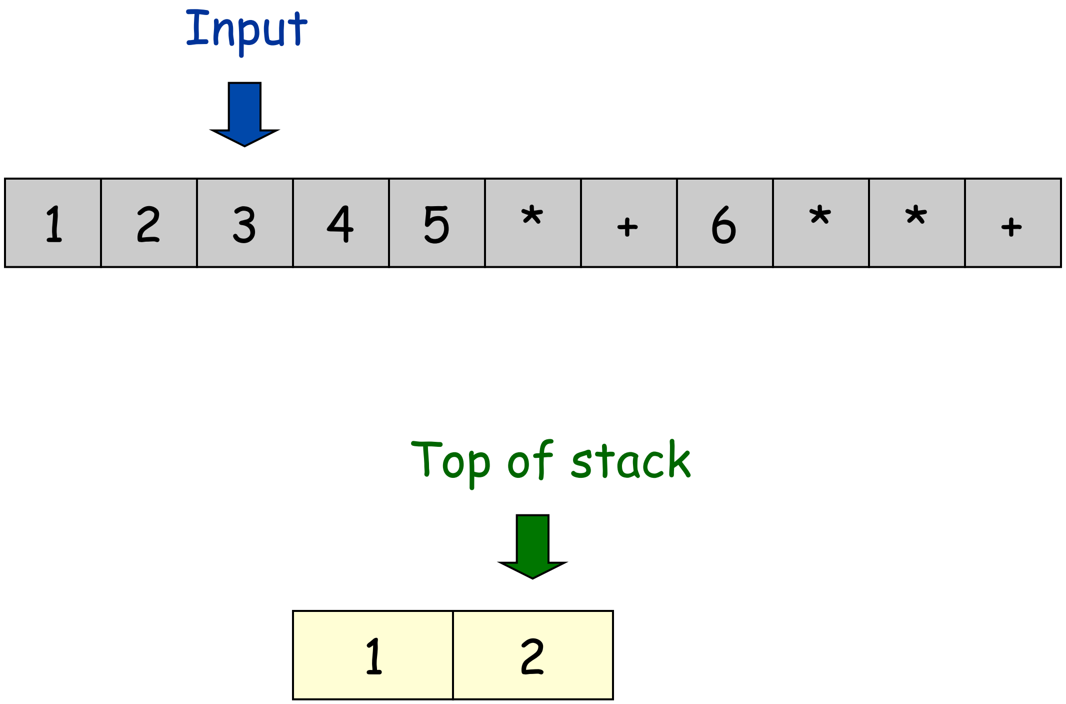
Input

1	2	3	4	5	*	+	6	*	*	+

Top of stack

1	2

Postfix Demo



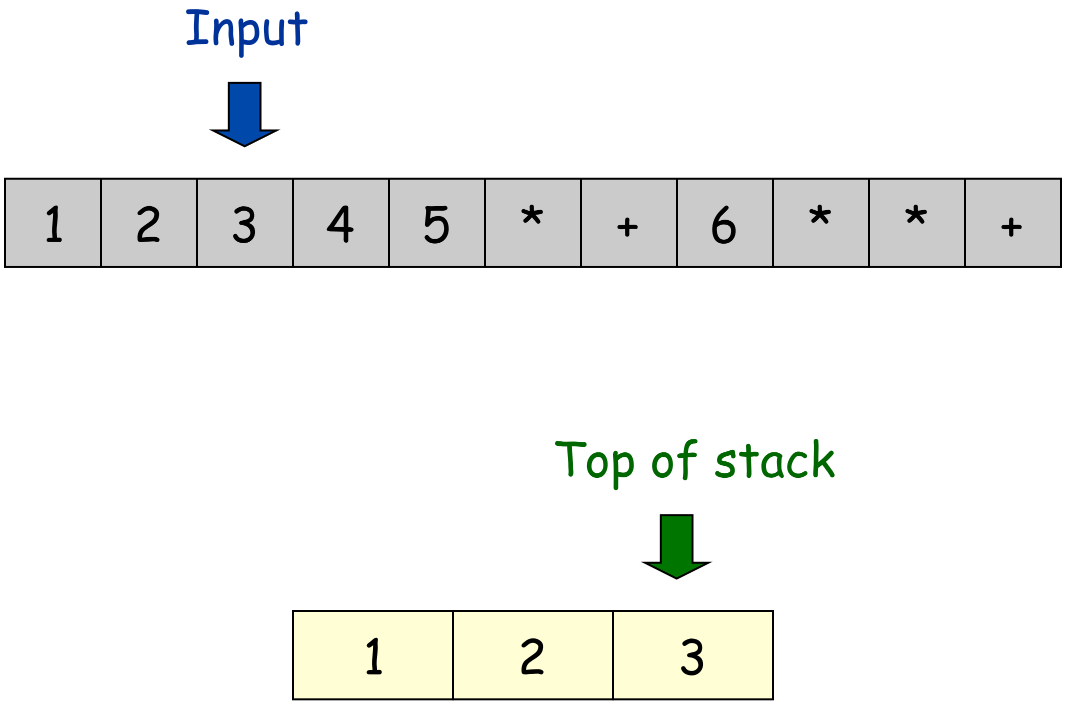
Input

1	2	3	4	5	*	+	6	*	*	+

Top of stack

1	2

Postfix Demo



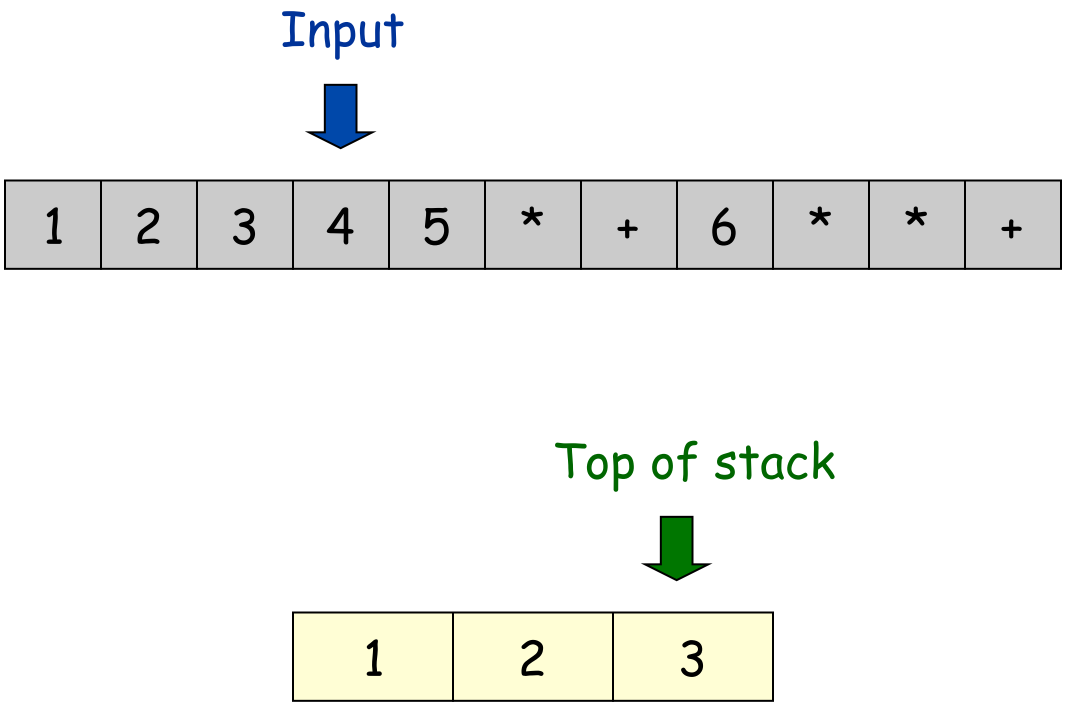
Input

1	2	3	4	5	*	+	6	*	*	+

Top of stack

1	2	3

Postfix Demo



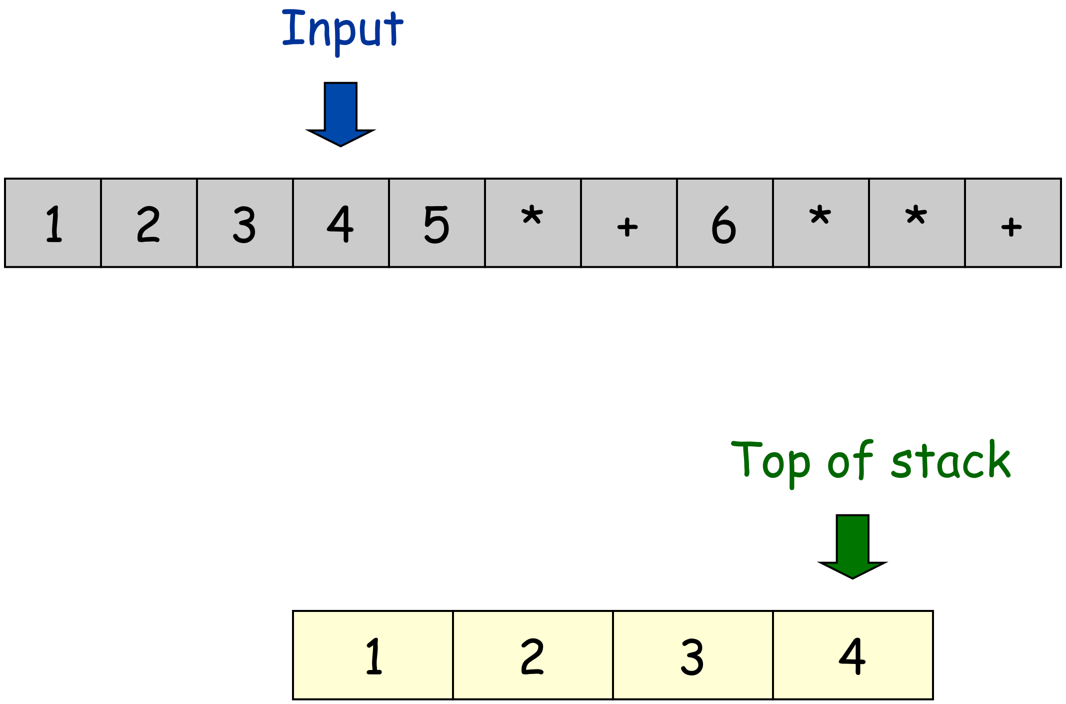
Input

1	2	3	4	5	*	+	6	*	*	+

Top of stack

1	2	3

Postfix Demo



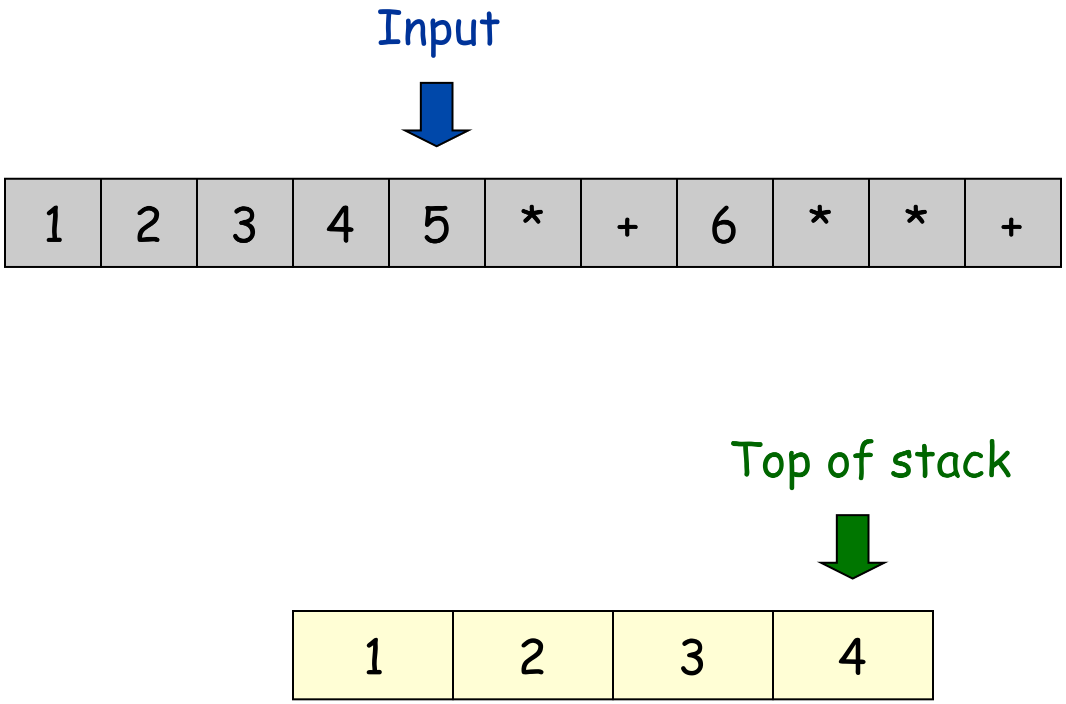
Input

1	2	3	4	5	*	+	6	*	*	+

Top of stack

1	2	3	4

Postfix Demo



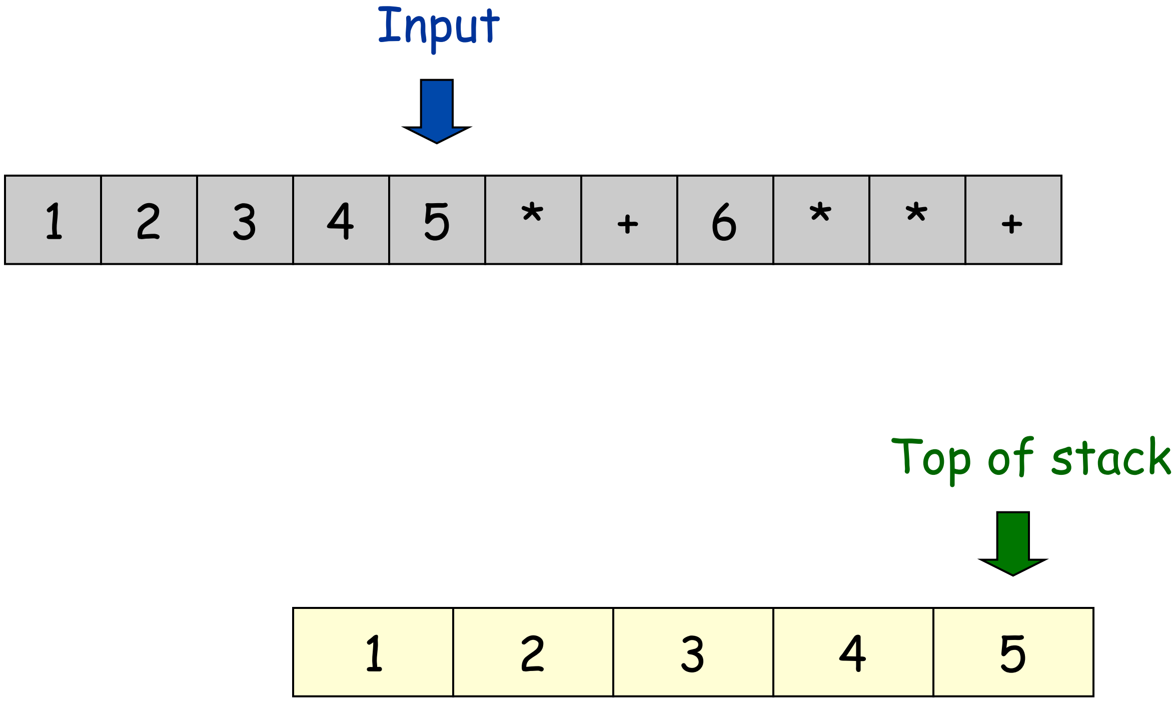
Input

1	2	3	4	5	*	+	6	*	*	+

Top of stack

1	2	3	4

Postfix Demo



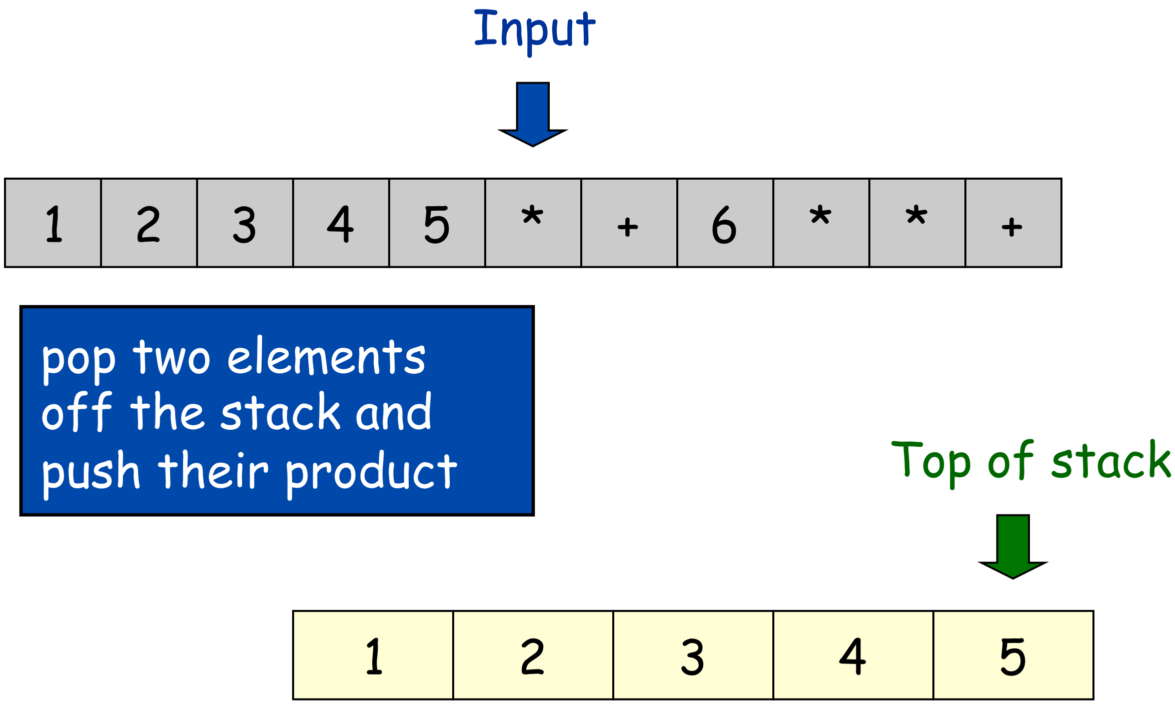
Input

1	2	3	4	5	*	+	6	*	*	+

Top of stack

1	2	3	4	5

Postfix Demo



Input

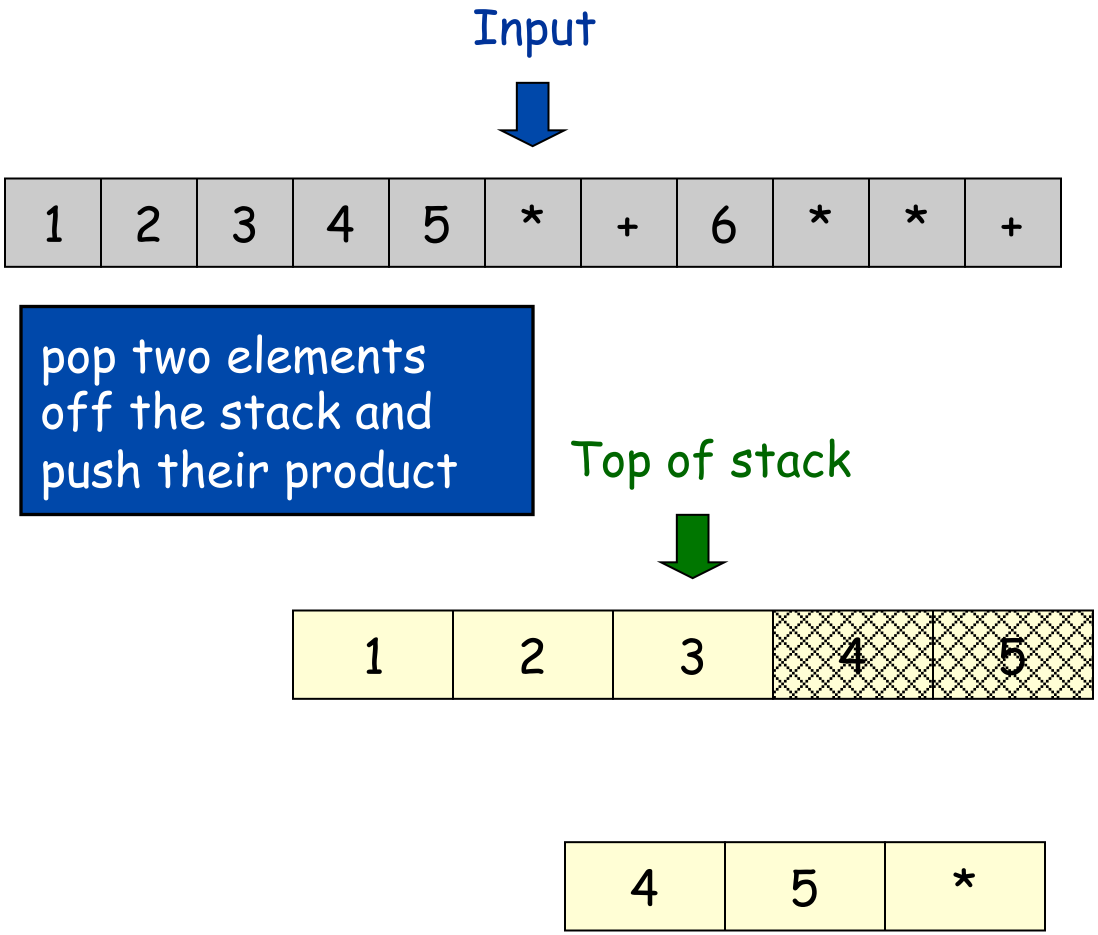
1	2	3	4	5	*	+	6	*	*	+

pop two elements off the stack and push their product

Top of stack

1	2	3	4	5

Postfix Demo



Input

1	2	3	4	5	*	+	6	*	*	+

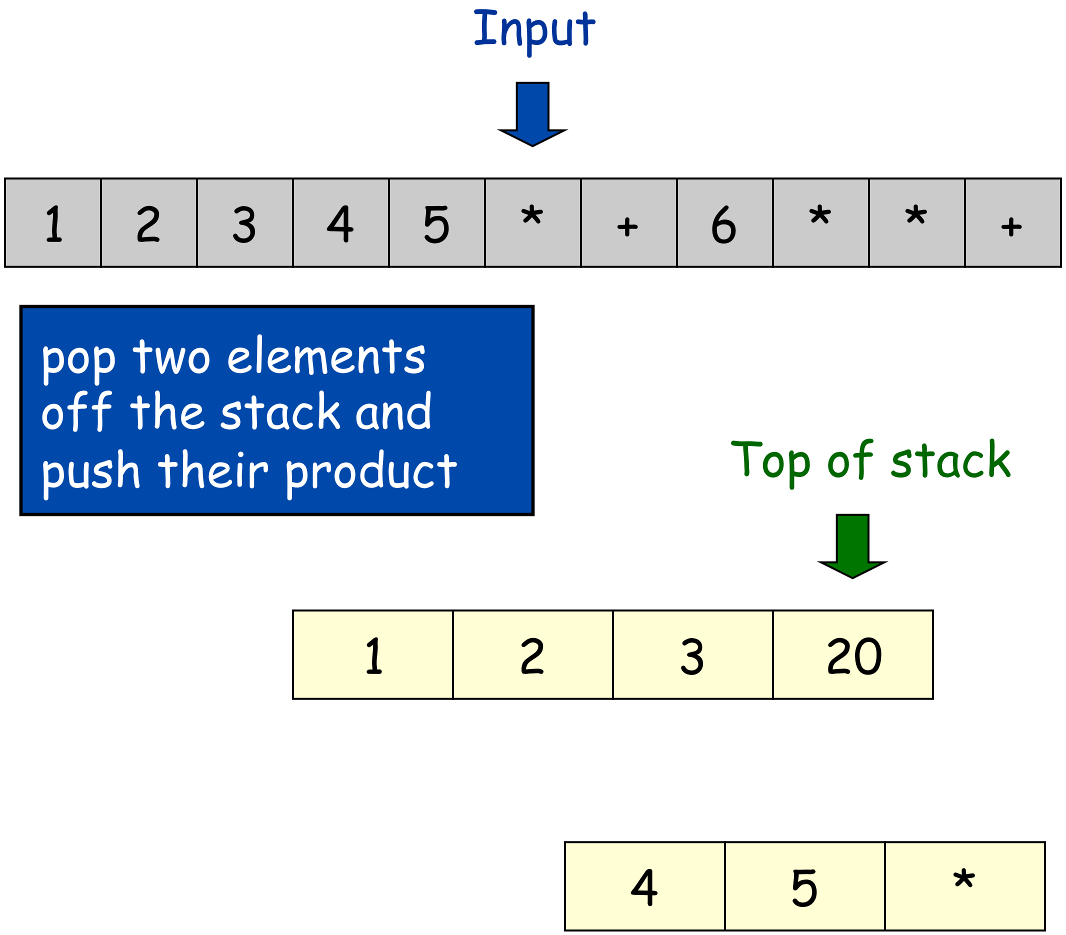
pop two elements off the stack and push their product

Top of stack

1	2	3	4	5

4	5	*

Postfix Demo



Input

1	2	3	4	5	*	+	6	*	*	+

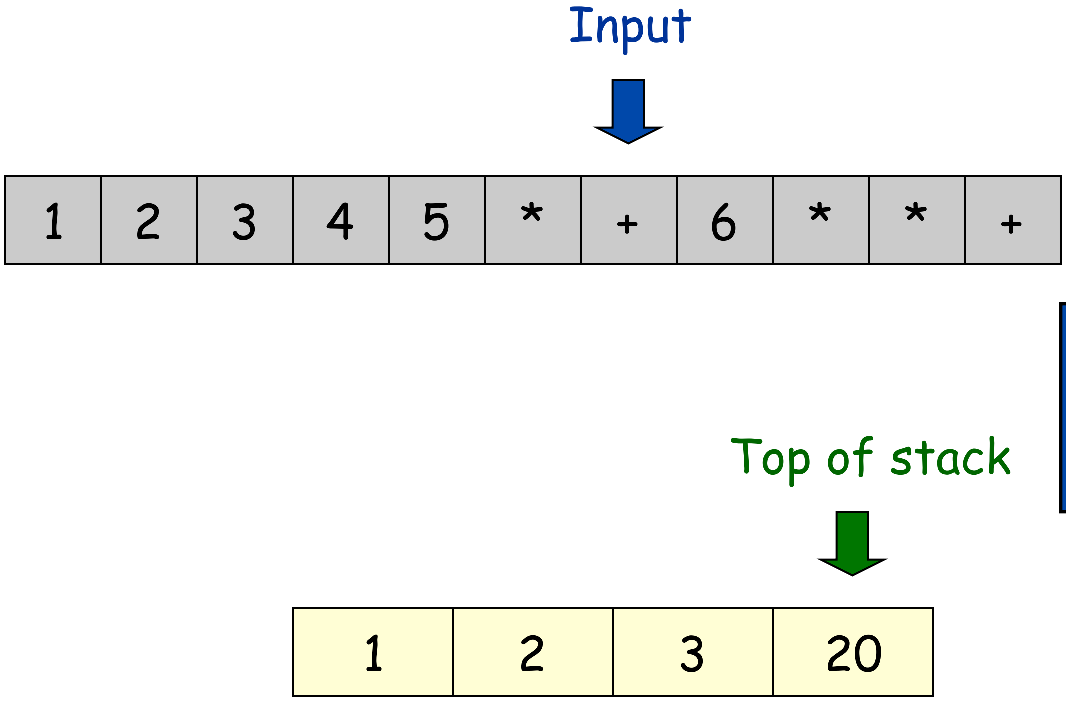
pop two elements off the stack and push their product

Top of stack

1	2	3	20

4	5	*

Postfix Demo



Input

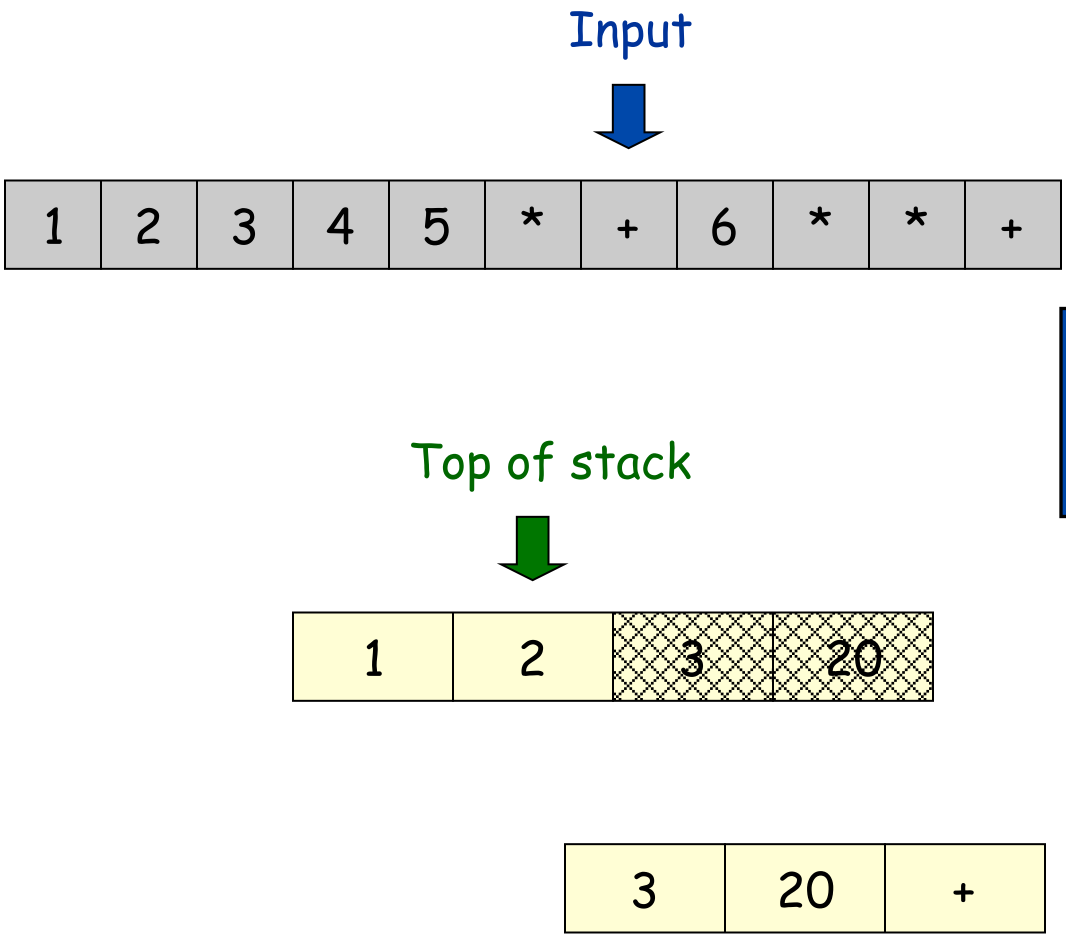
1	2	3	4	5	*	+	6	*	*	+

pop two elements off the stack and push their sum

Top of stack

1	2	3	20

Postfix Demo



Input

1	2	3	4	5	*	+	6	*	*	+

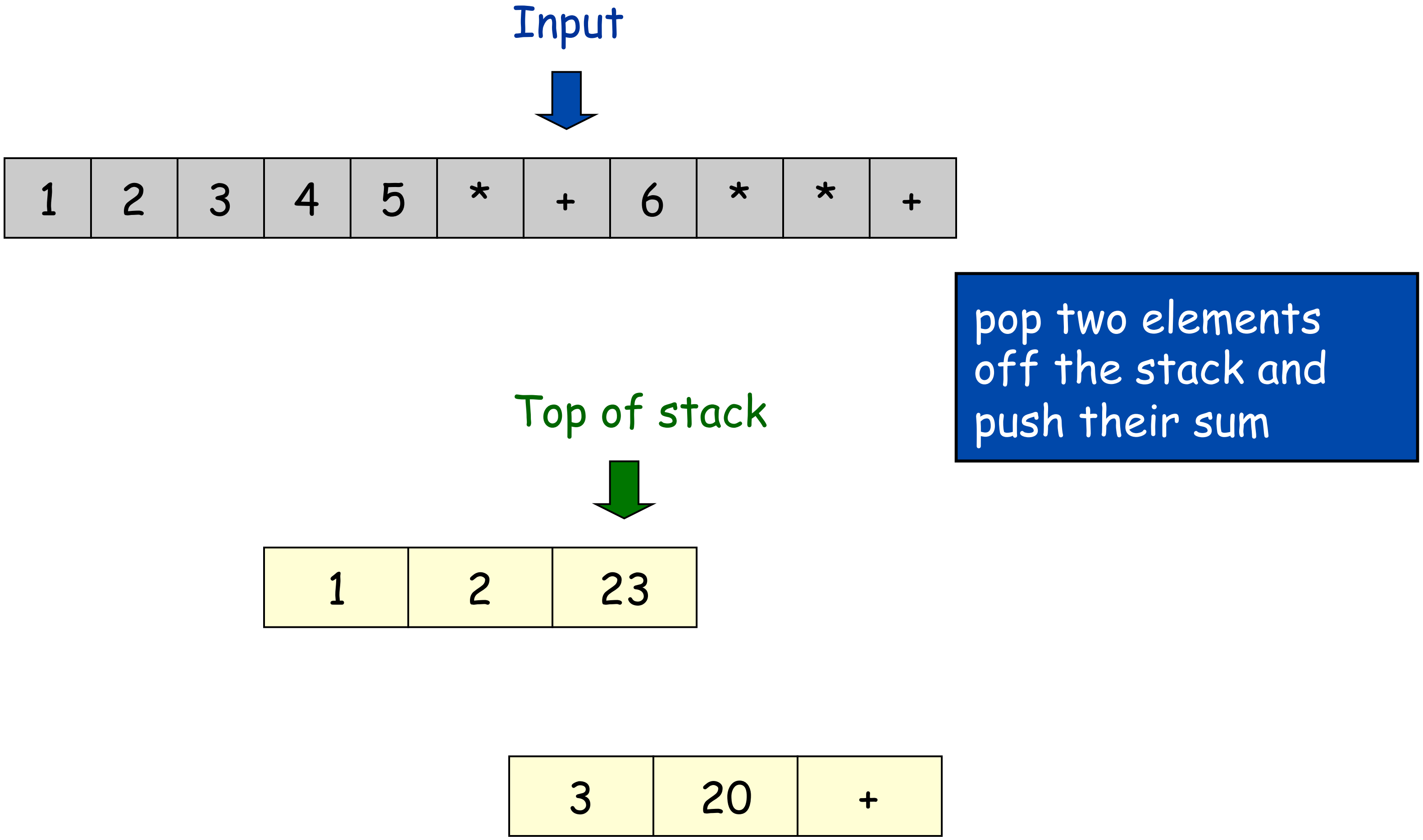
Top of stack

1	2	3	20

3	20	+

pop two elements off the stack and push their sum

Postfix Demo



Input

1	2	3	4	5	*	+	6	*	*	+

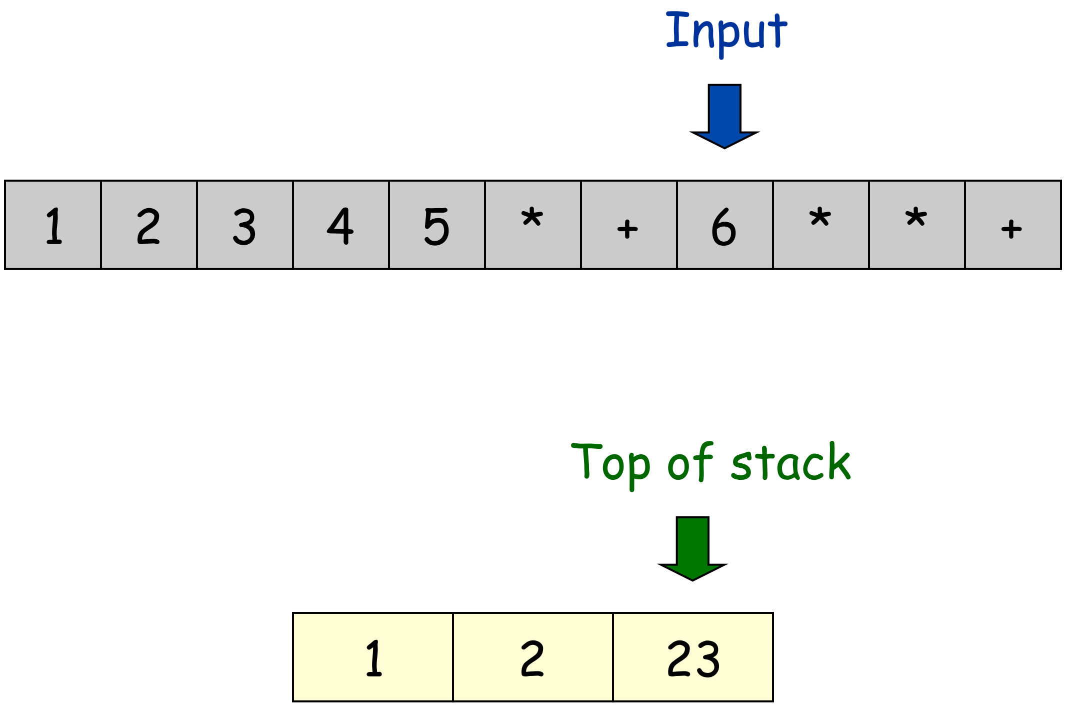
pop two elements off the stack and push their sum

Top of stack

1	2	23

3	20	+

Postfix Demo



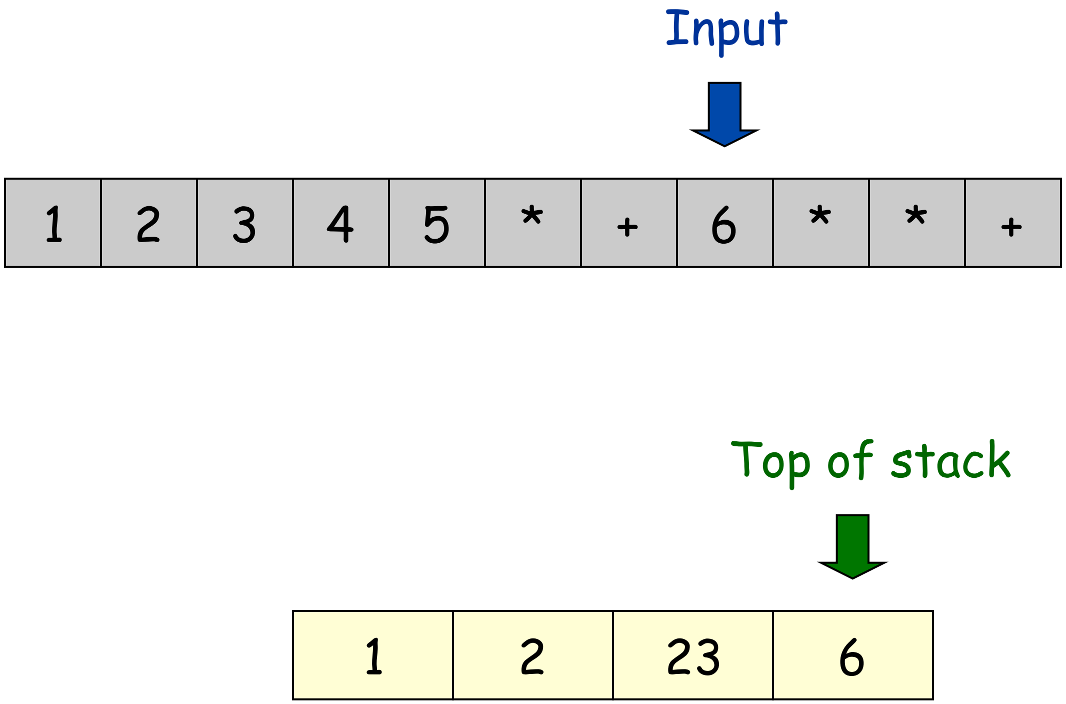
Input

1	2	3	4	5	*	+	6	*	*	+

Top of stack

1	2	23

Postfix Demo



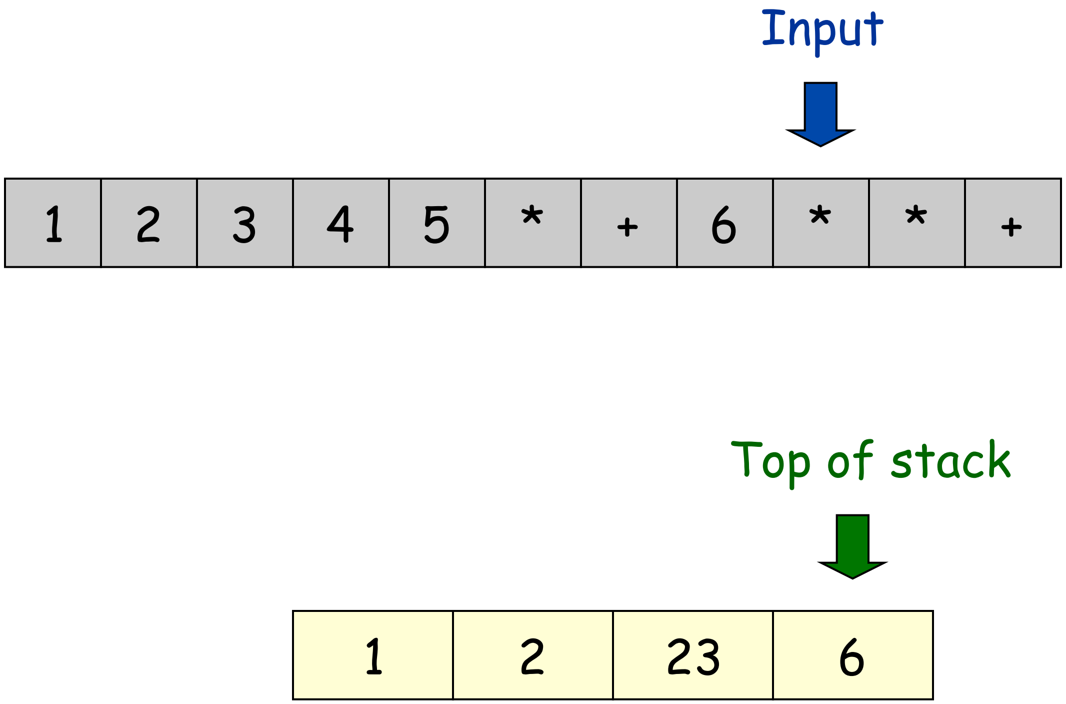
Input

1	2	3	4	5	*	+	6	*	*	+

Top of stack

1	2	23	6

Postfix Demo



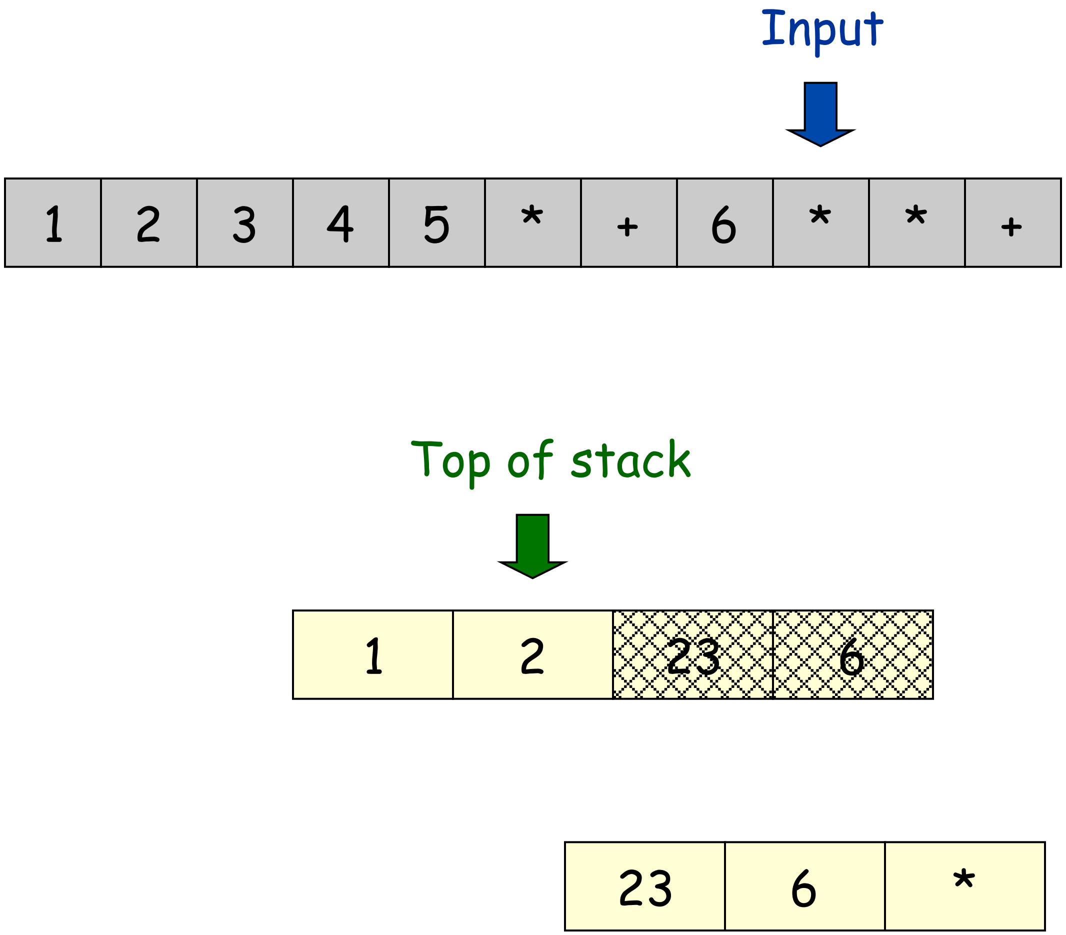
Input

1	2	3	4	5	*	+	6	*	*	+

Top of stack

1	2	23	6

Postfix Demo



Input

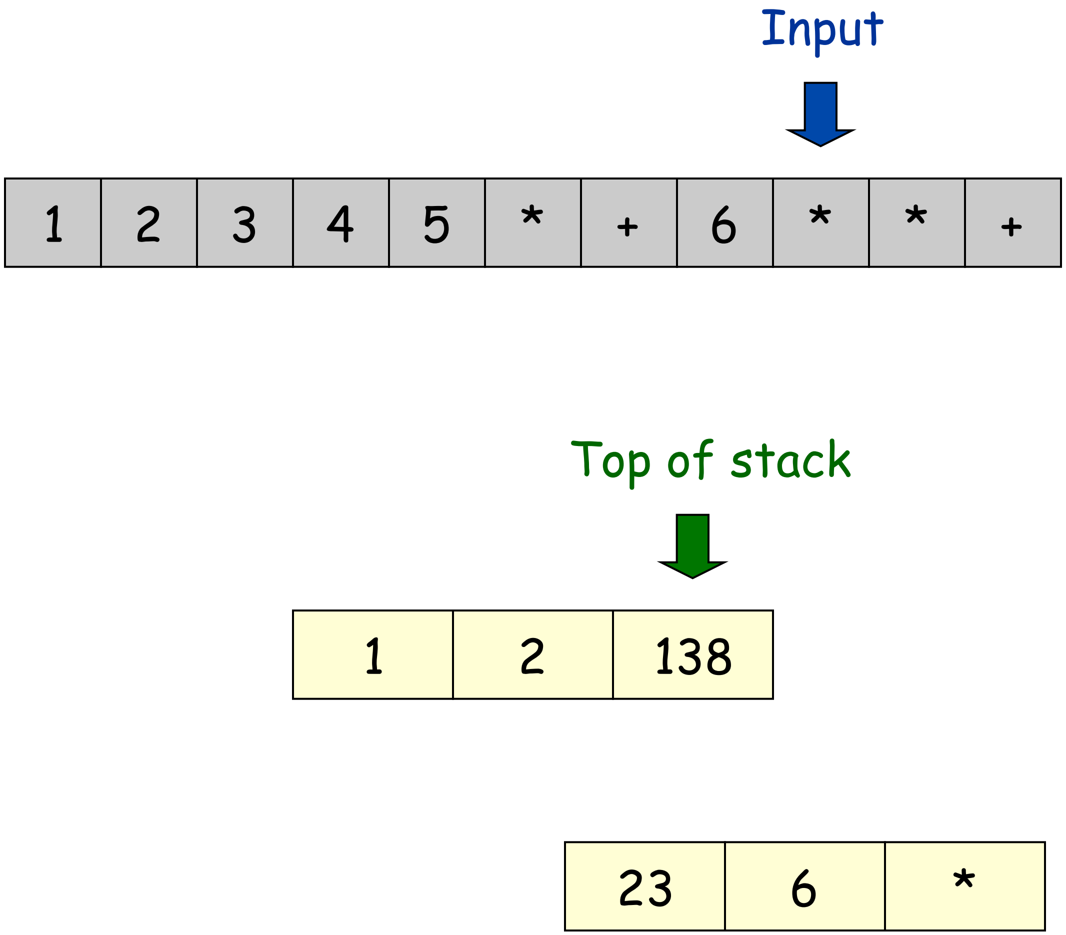
1	2	3	4	5	*	+	6	*	*	+

Top of stack

1	2	23	6

23	6	*

Postfix Demo



Input

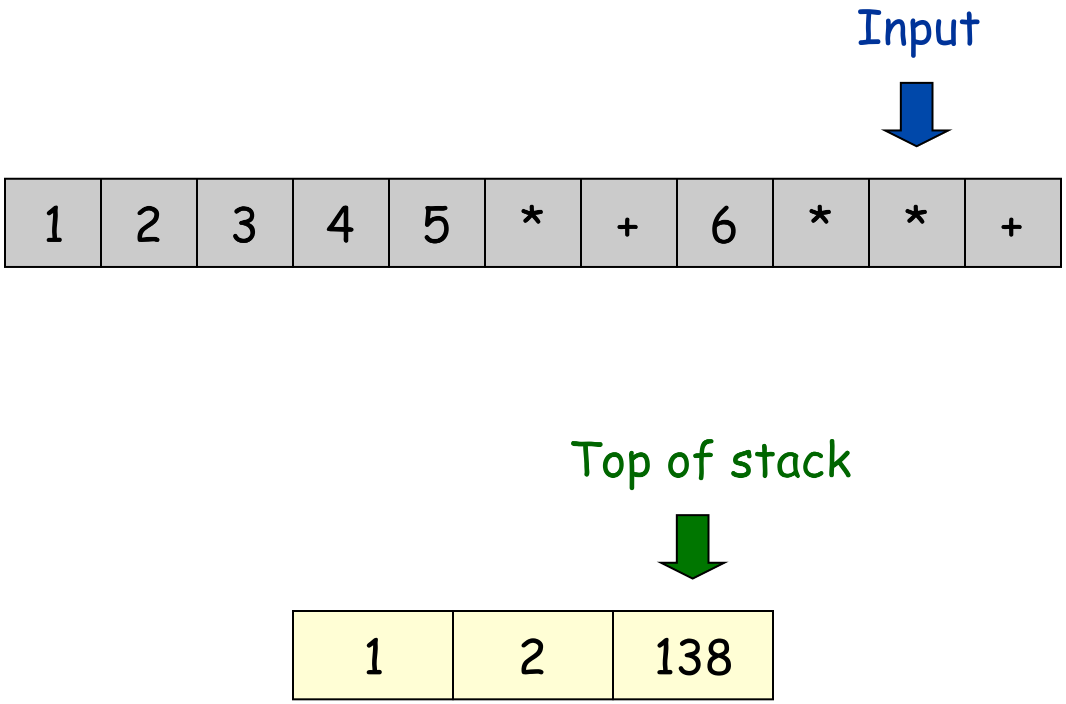
1	2	3	4	5	*	+	6	*	*	+

Top of stack

1	2	138

23	6	*

Postfix Demo



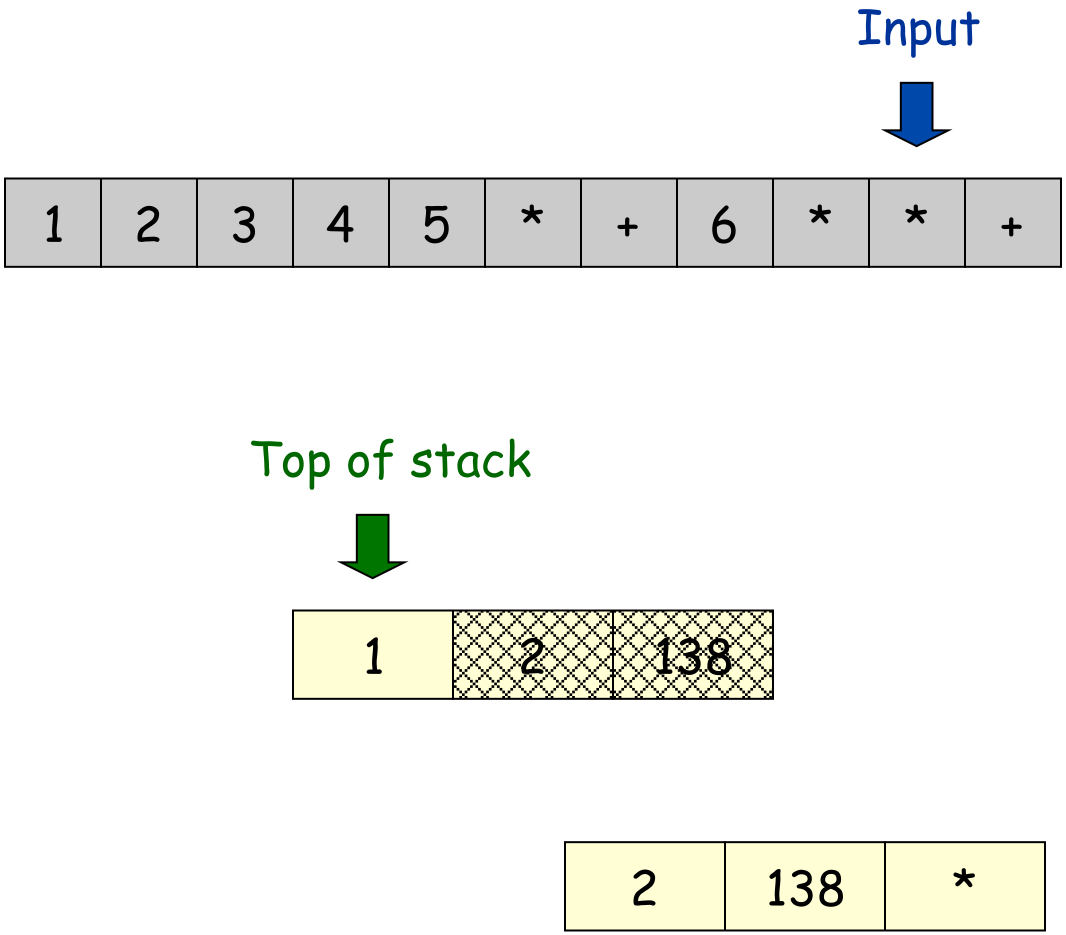
Input

1	2	3	4	5	*	+	6	*	*	+

Top of stack

1	2	138

Postfix Demo



Input

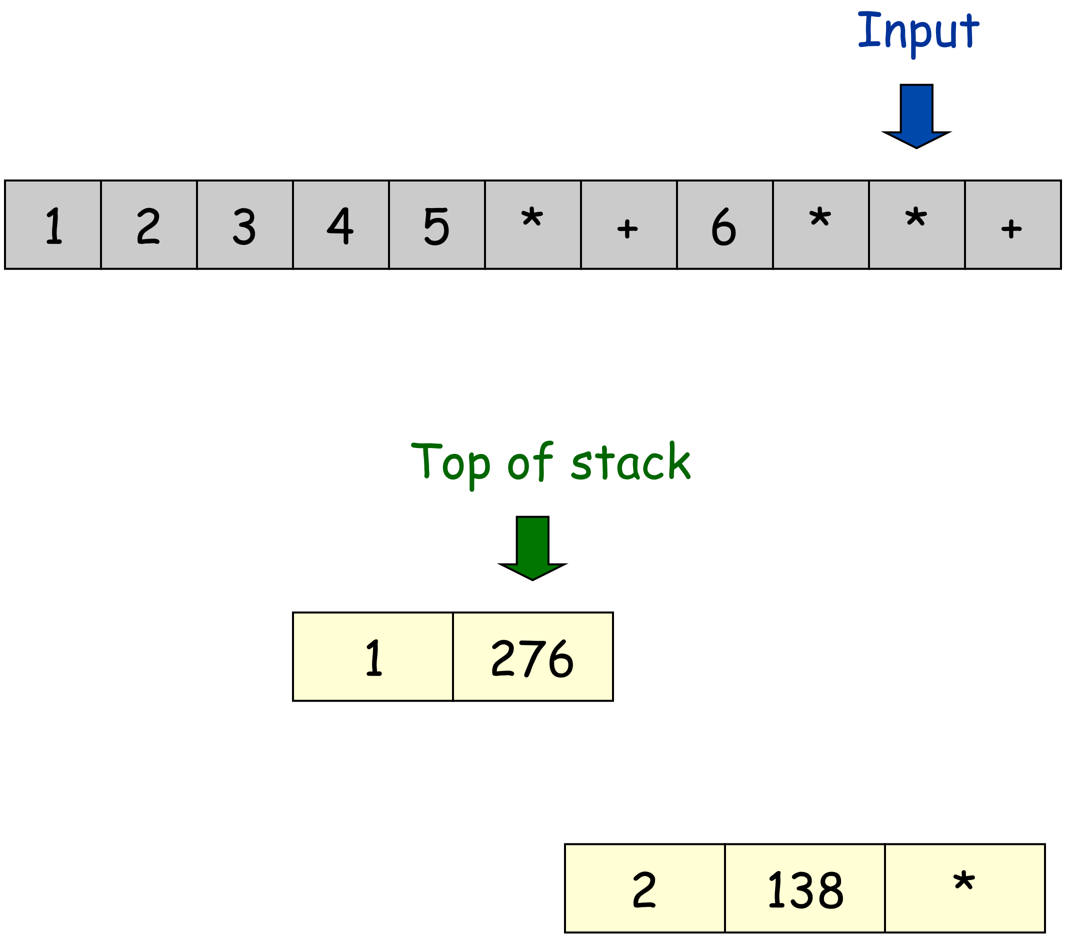
1	2	3	4	5	*	+	6	*	*	+

Top of stack

1	▨ 2	▨ 138

2	138	*

Postfix Demo



Input

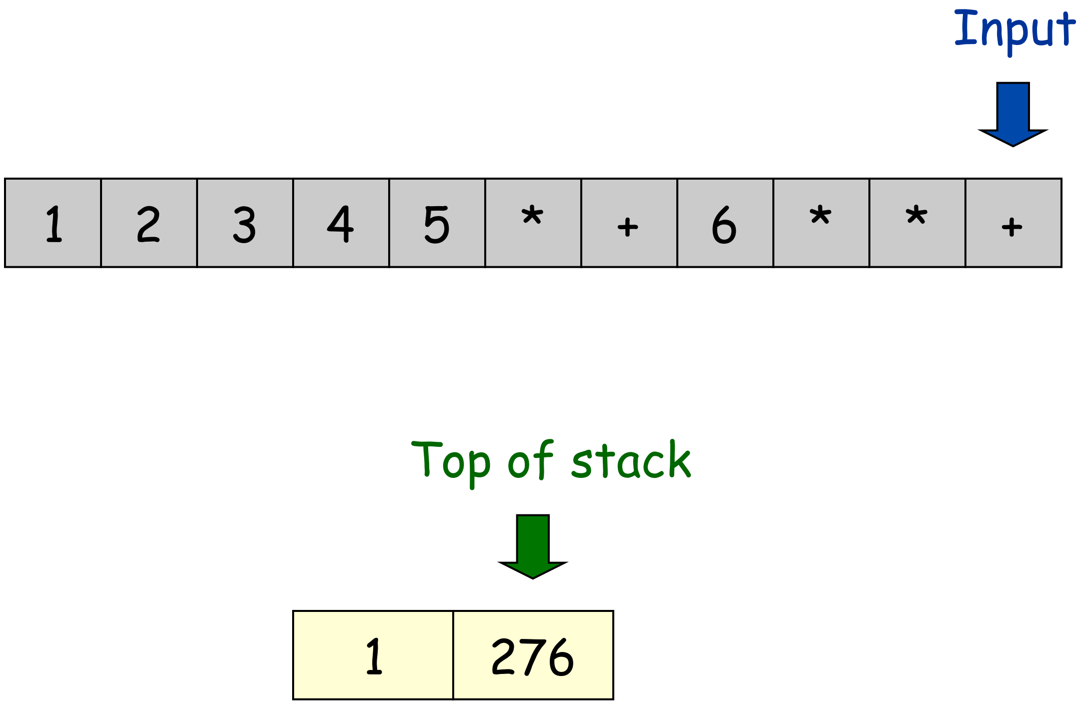
1	2	3	4	5	*	+	6	*	*	+

Top of stack

1	276

2	138	*

Postfix Demo



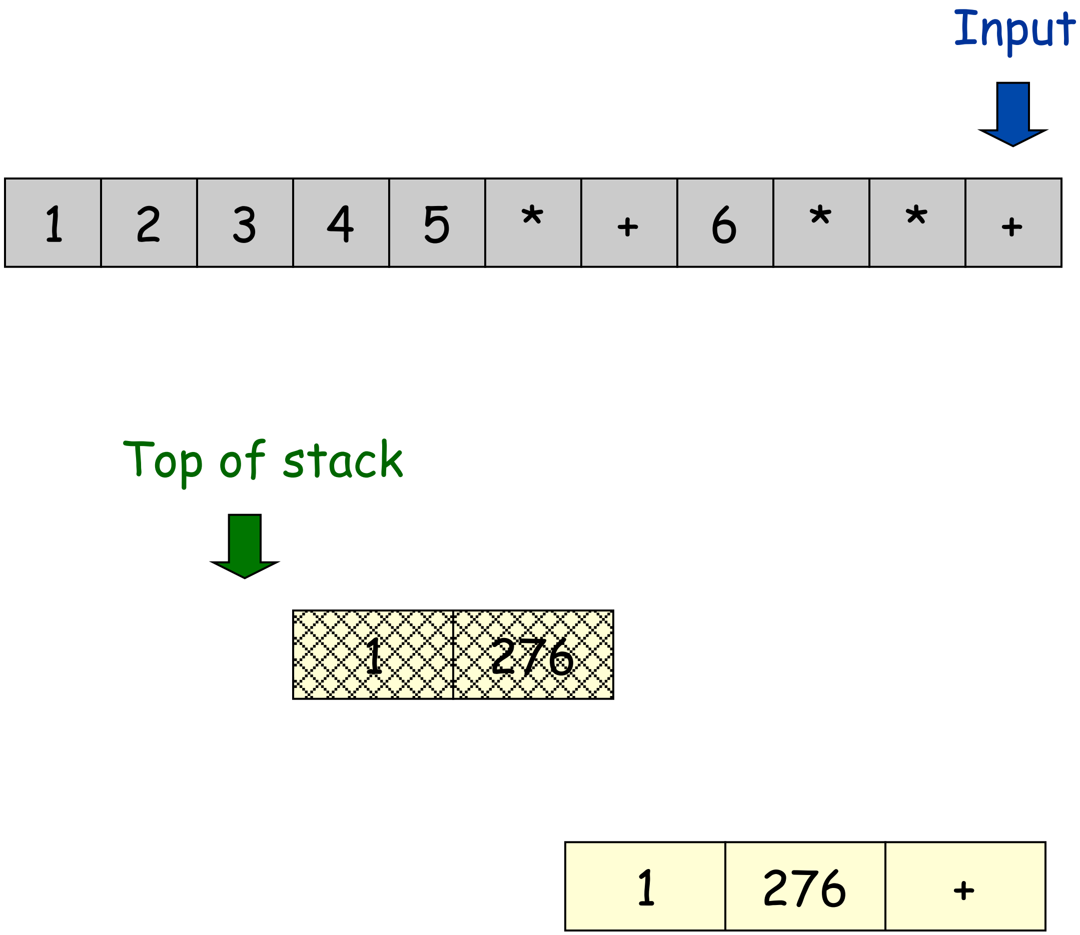
Input

1	2	3	4	5	*	+	6	*	*	+

Top of stack

1	276

Postfix Demo



Input

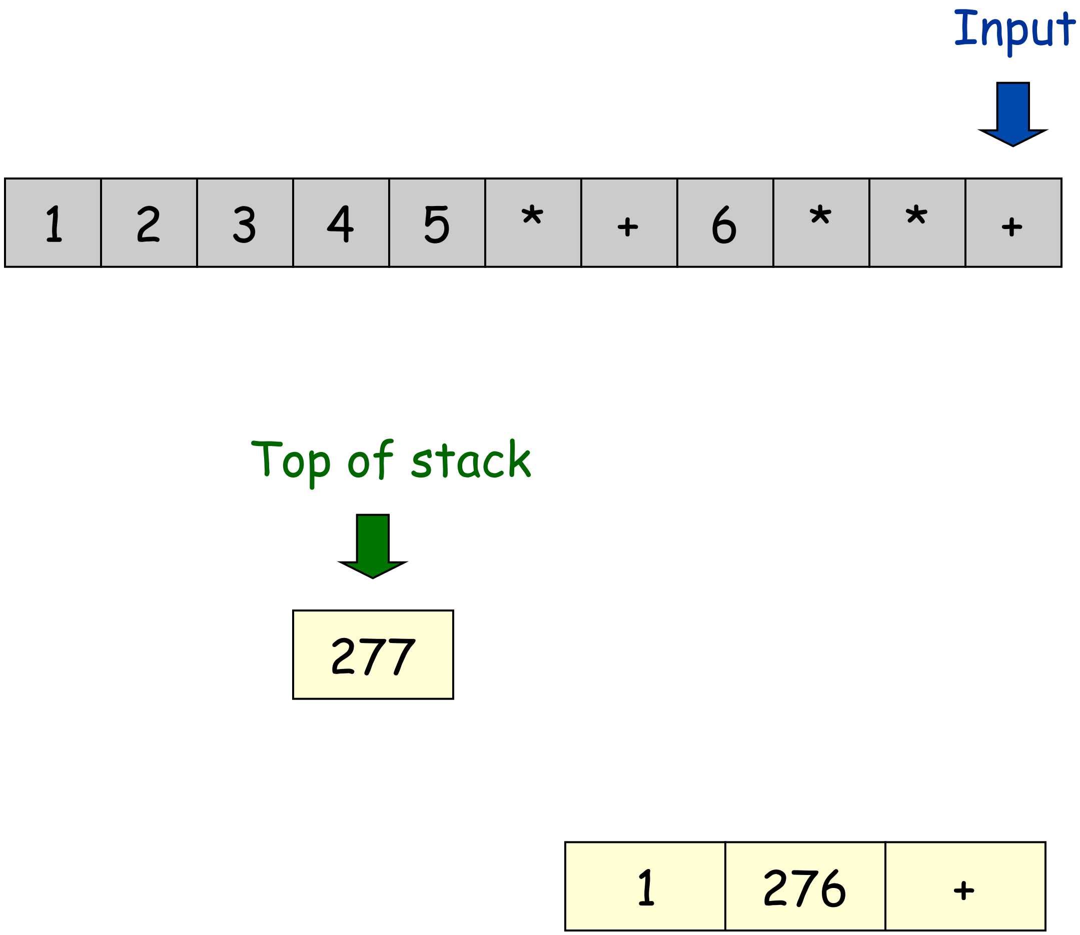
1	2	3	4	5	*	+	6	*	*	+

Top of stack

1	276

1	276	+

Postfix Demo



Input

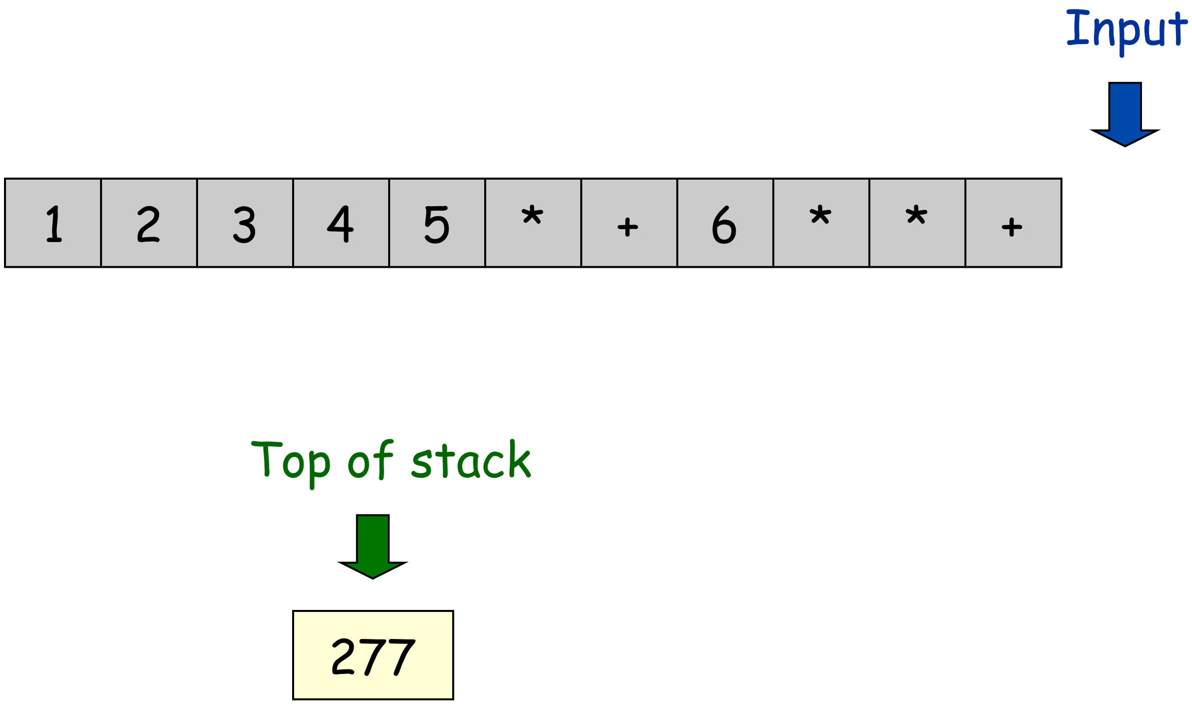
1	2	3	4	5	*	+	6	*	*	+

Top of stack

277

1	276	+

Postfix Demo



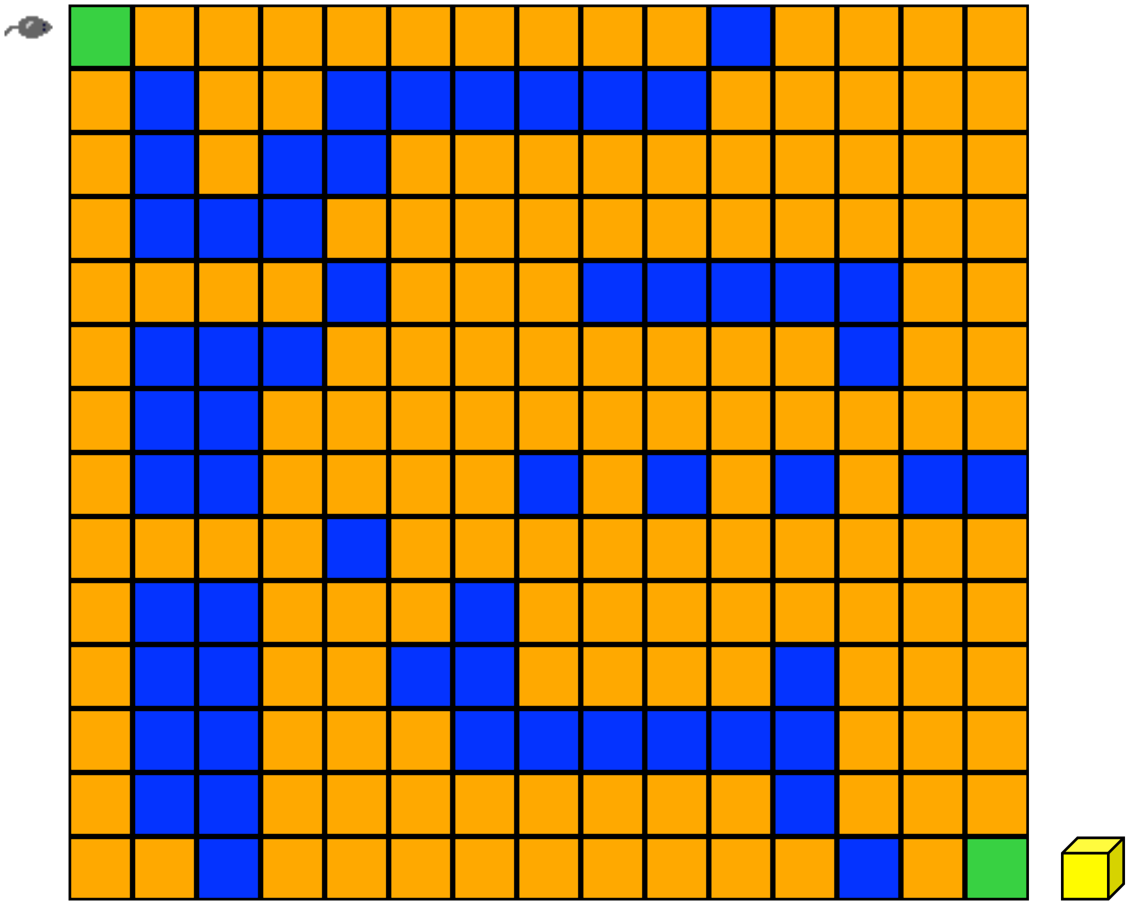
Input

1	2	3	4	5	*	+	6	*	*	+

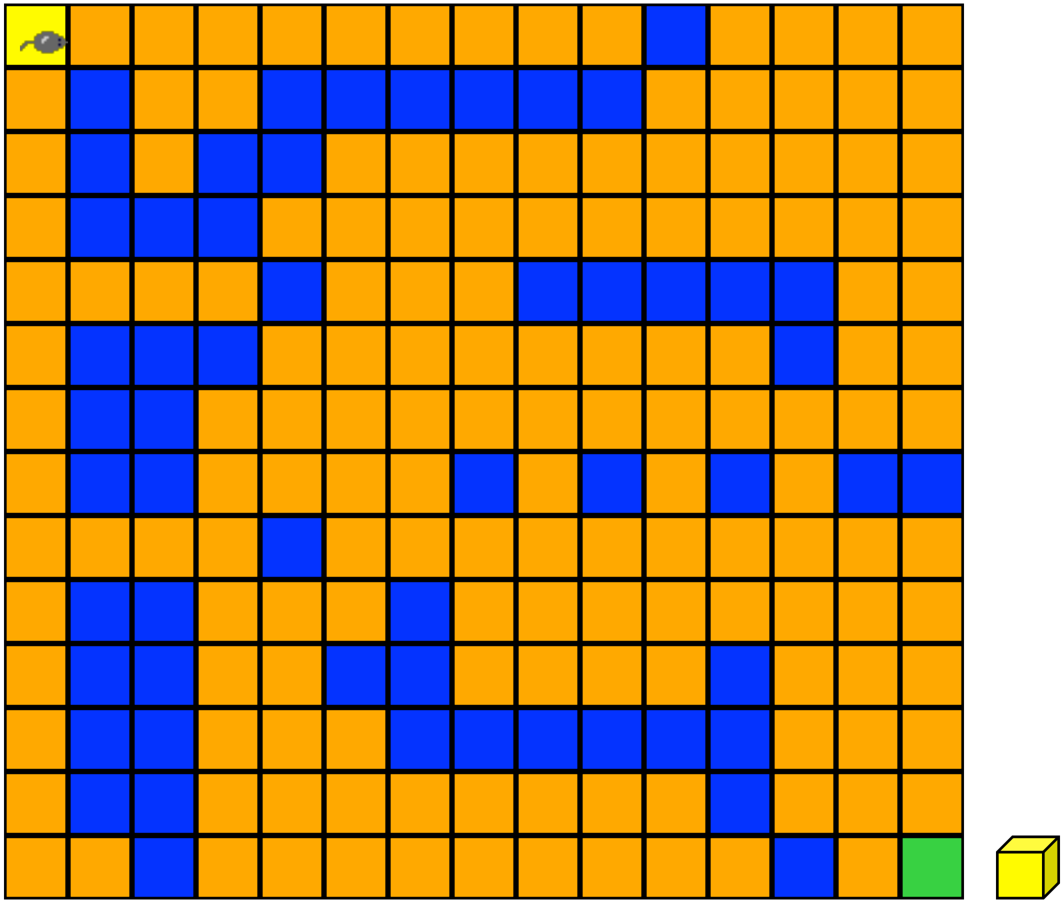
Top of stack

277

Rat In A Maze Demo

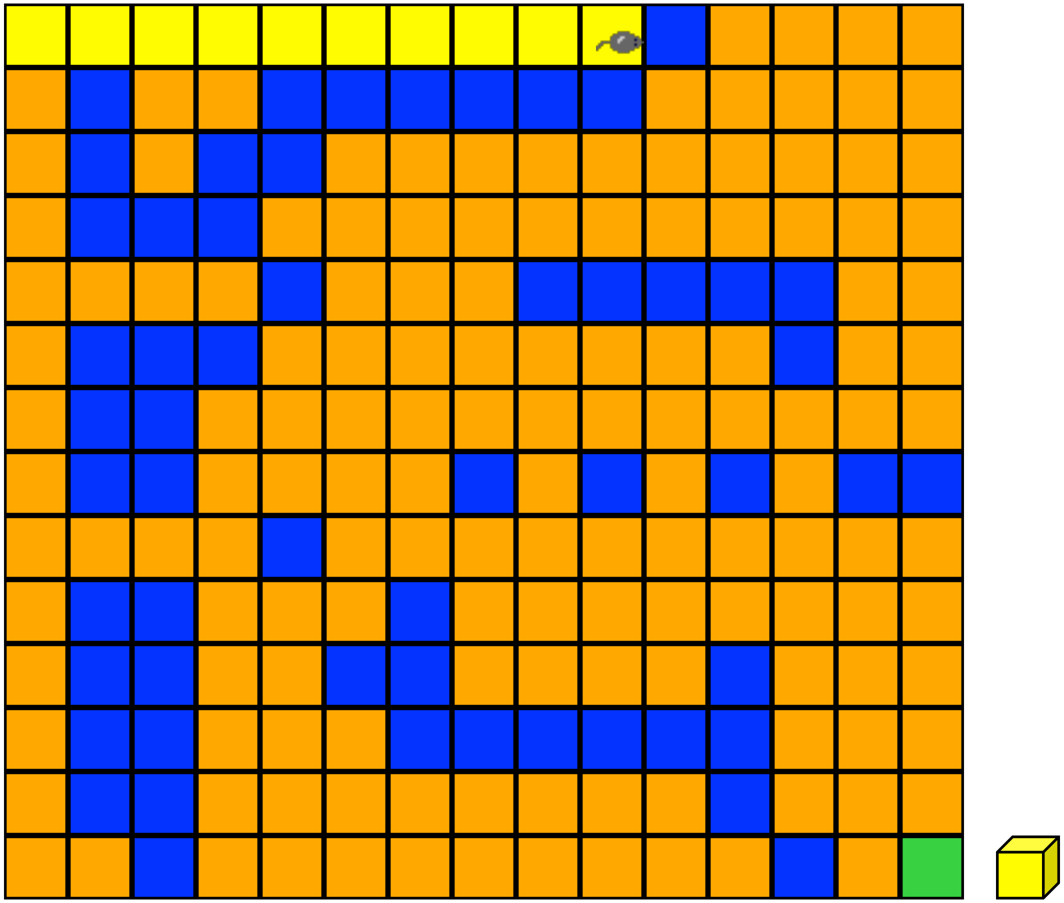


Rat In A Maze Demo



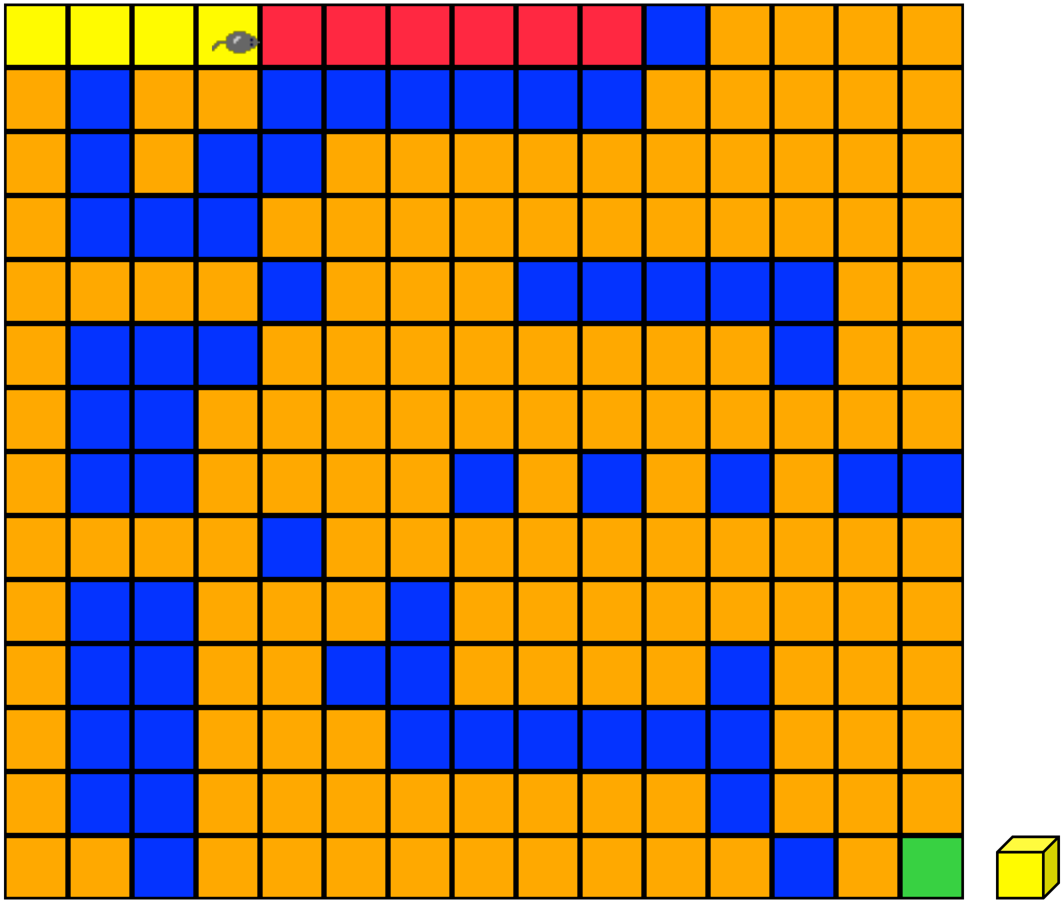
- Move order is: right, down, left, up
- Block positions to avoid revisit.

Rat In A Maze Demo



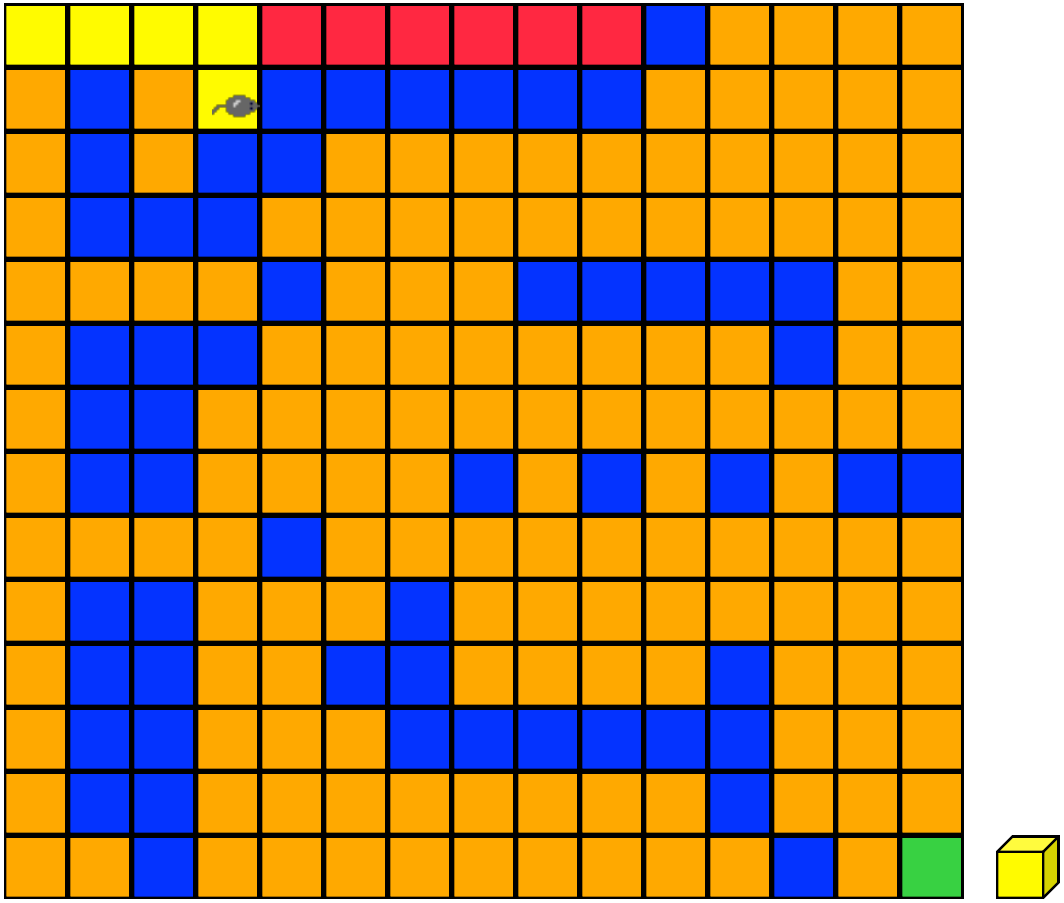
- Move backward until we reach a square from which a forward move is possible.

Rat In A Maze Demo



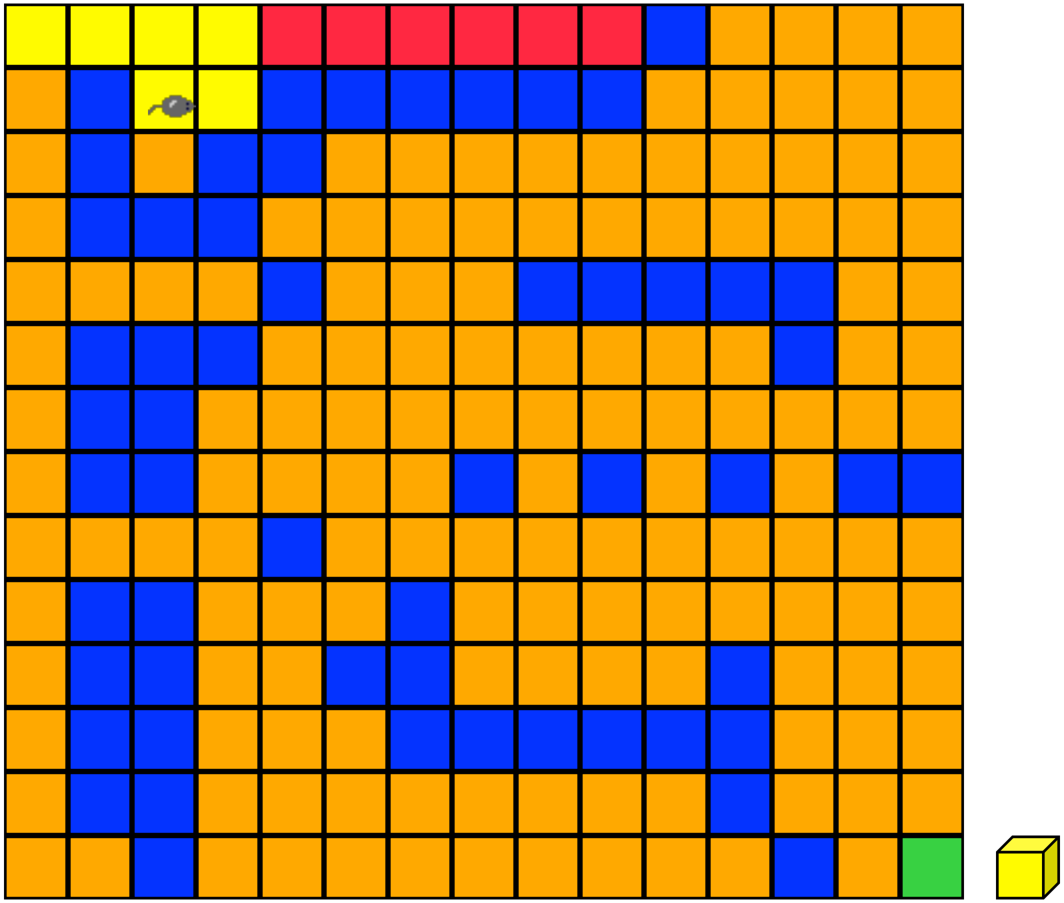
- Move down.

Rat In A Maze Demo



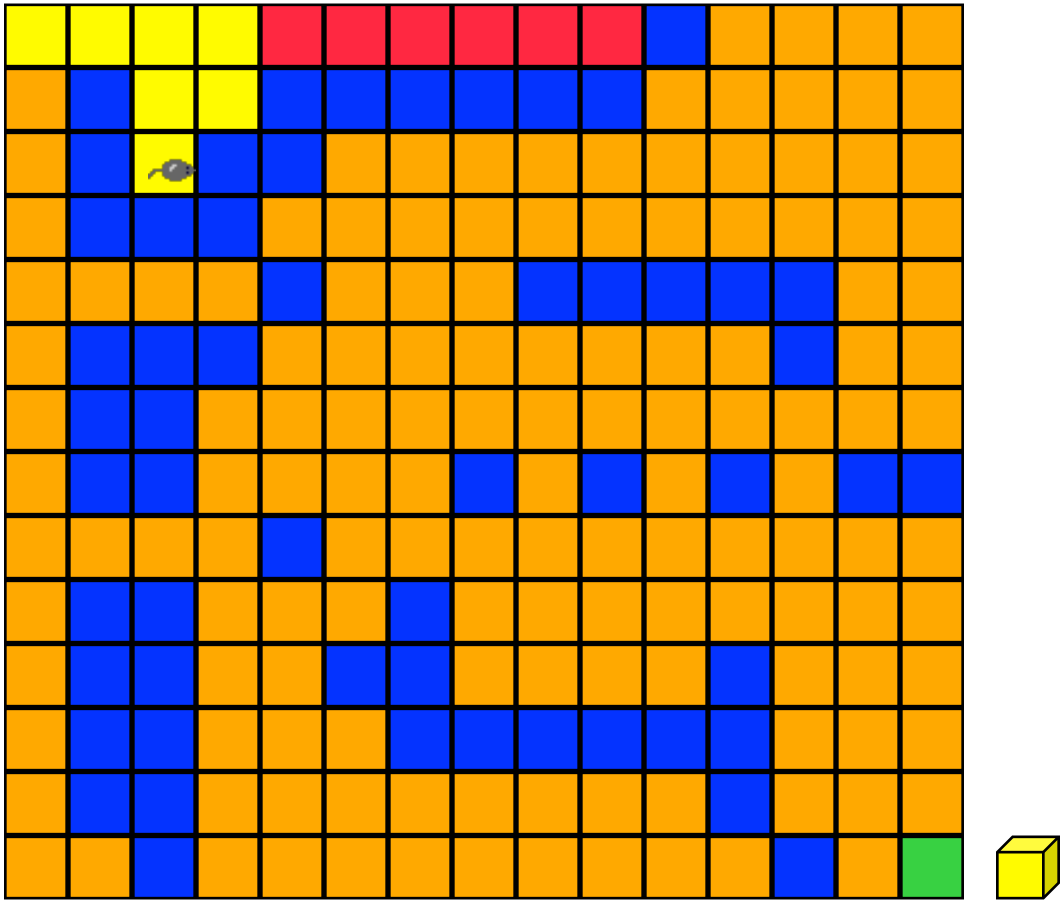
- Move left.

Rat In A Maze Demo



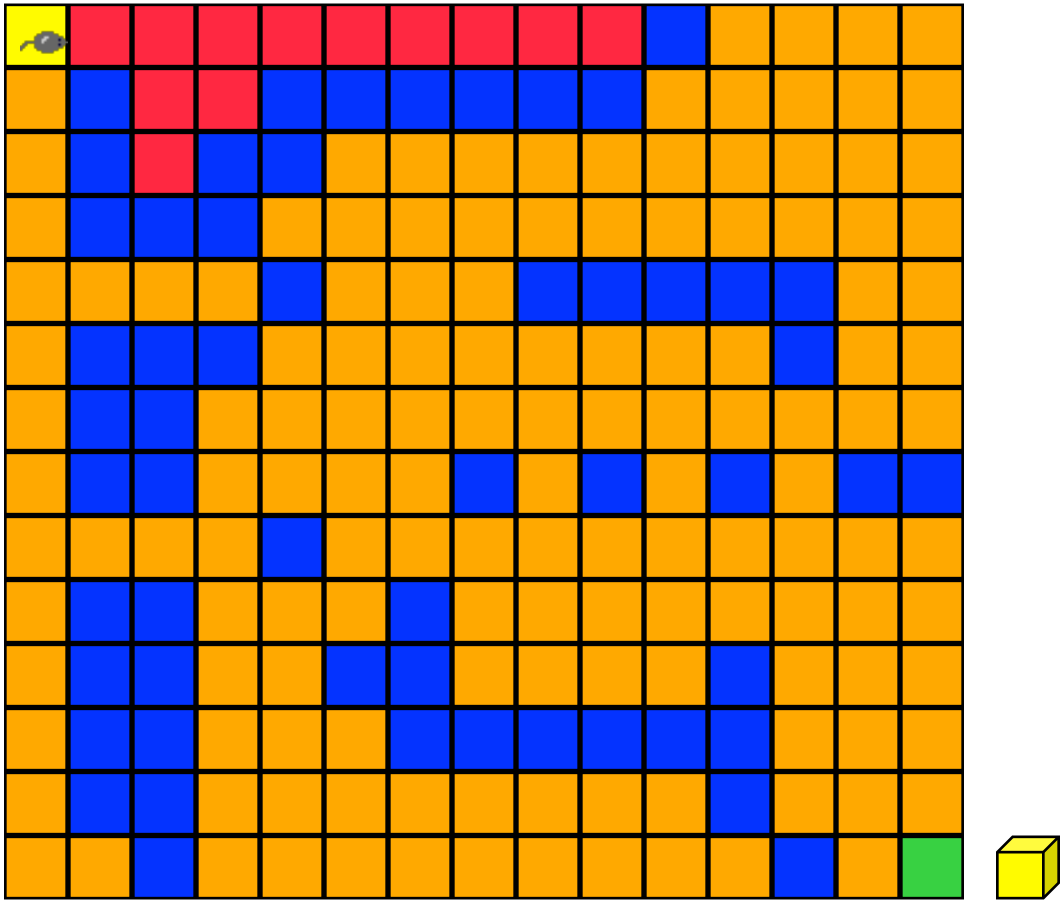
- Move down.

Rat In A Maze Demo



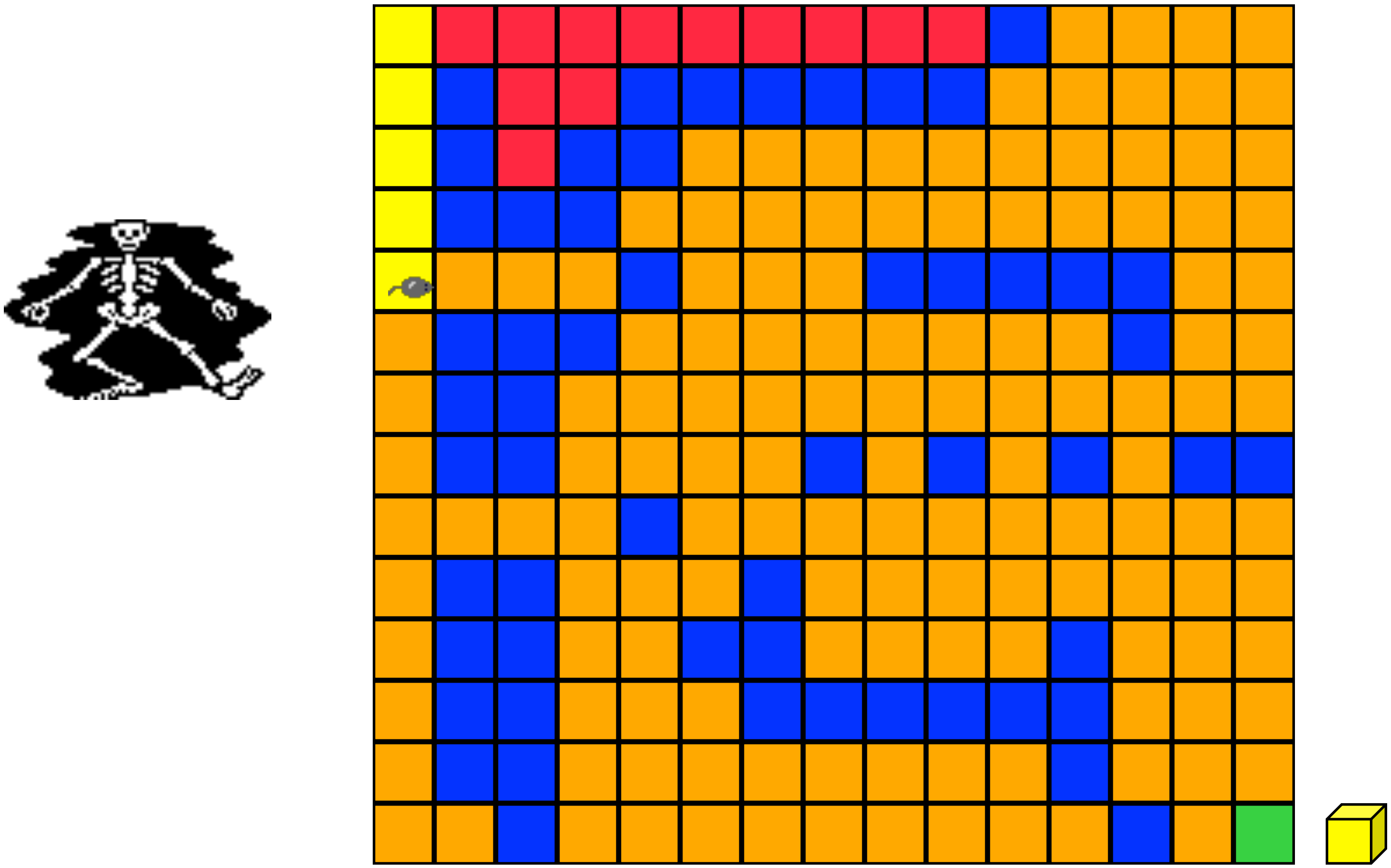
- Move backward until we reach a square from which a forward move is possible.

Rat In A Maze Demo

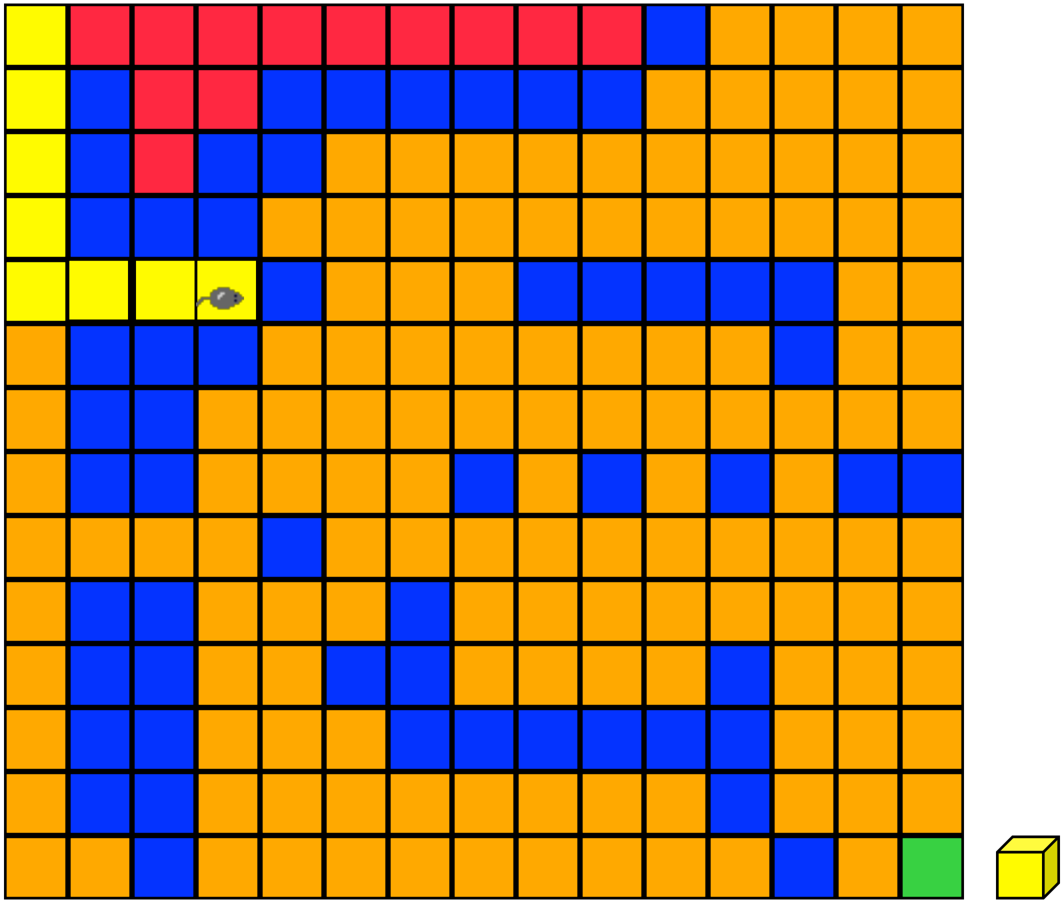


- Move backward until we reach a square from which a forward move is possible.
- Move downward

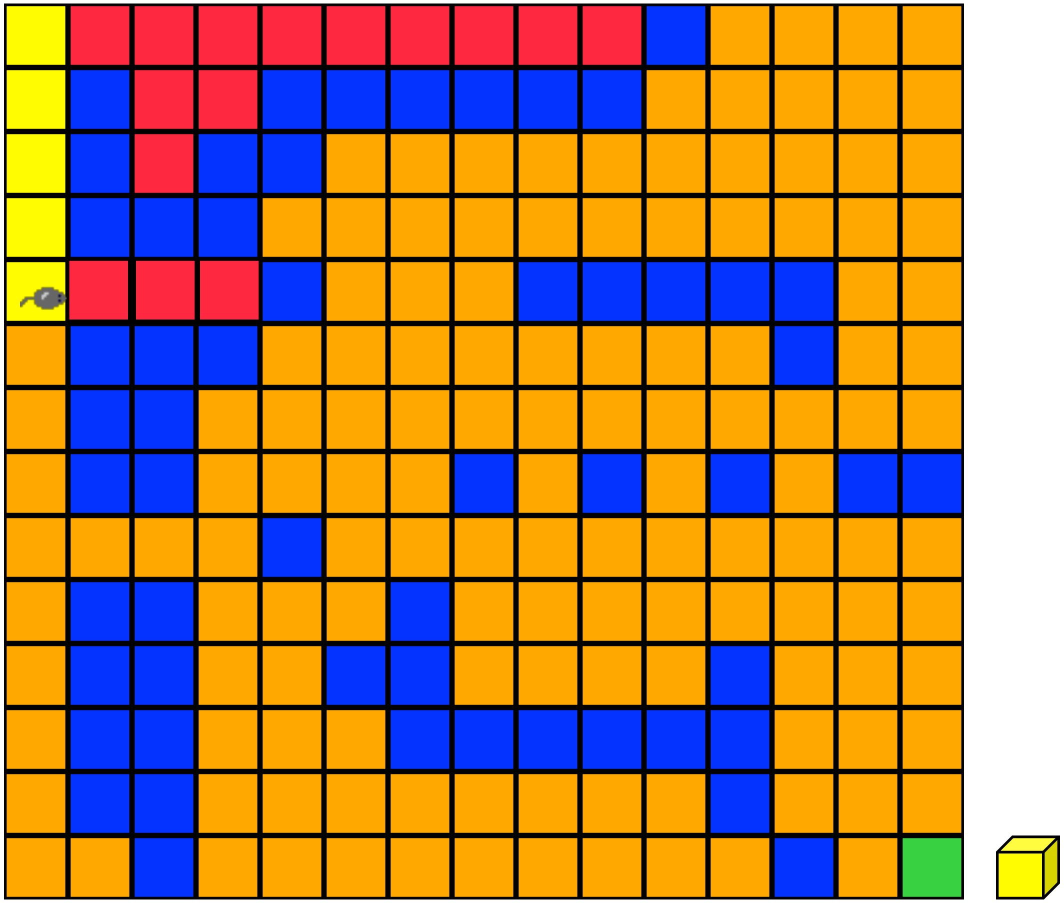
Rat In A Maze Demo



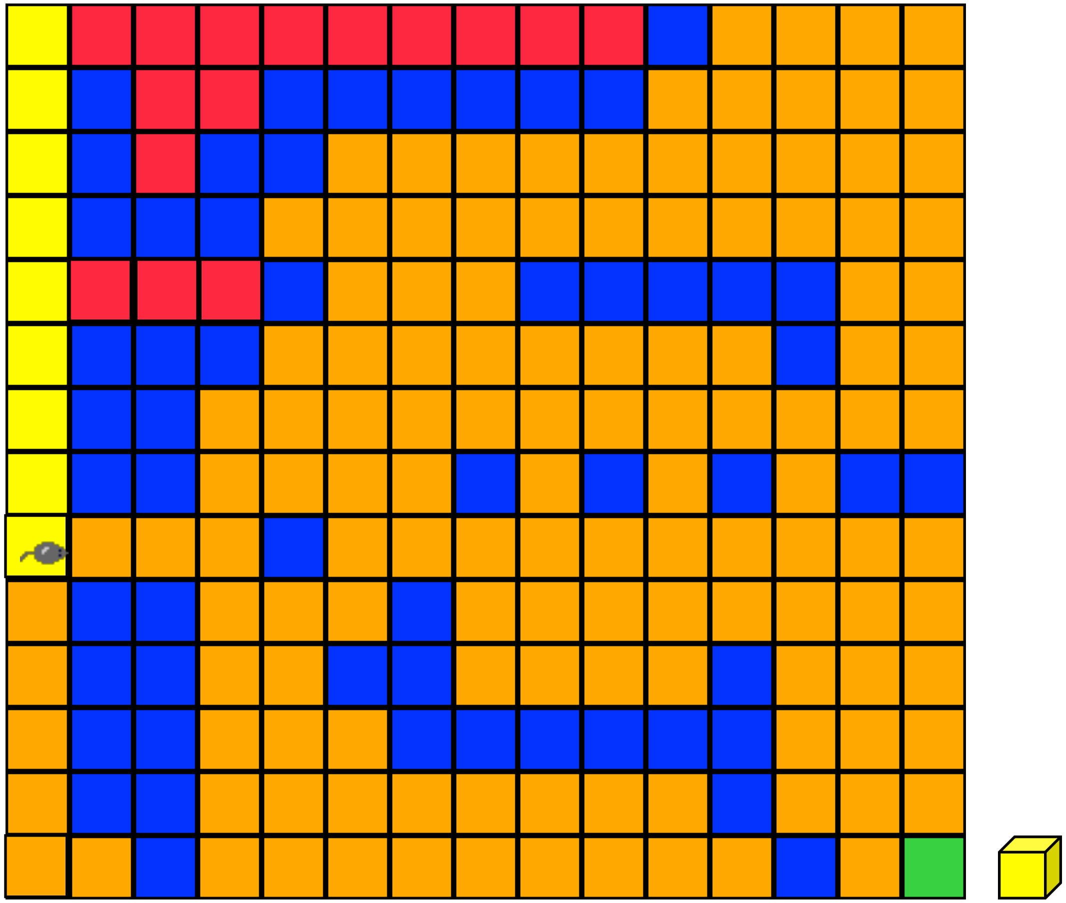
Rat In A Maze Demo



Rat In A Maze Demo

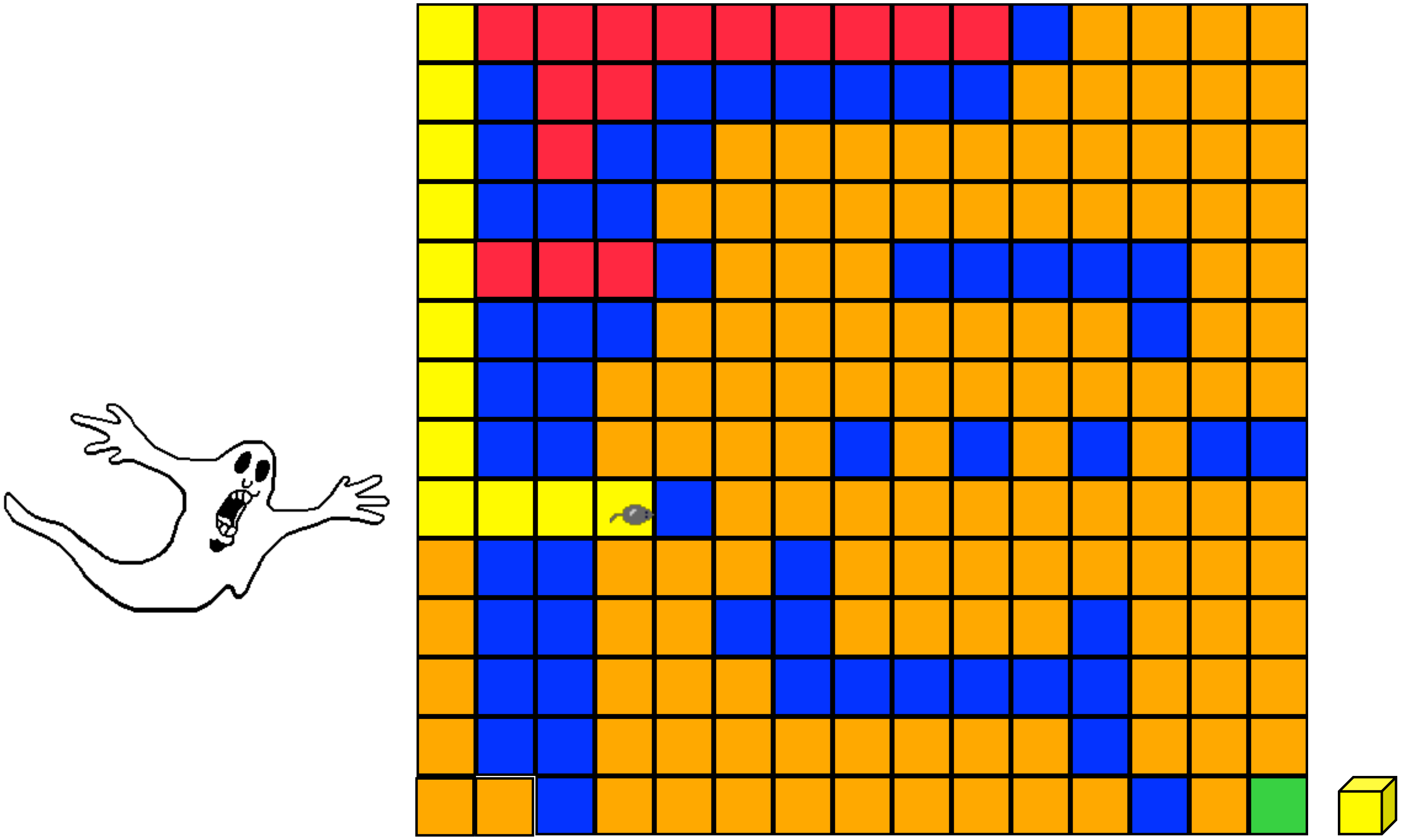


Rat In A Maze Demo



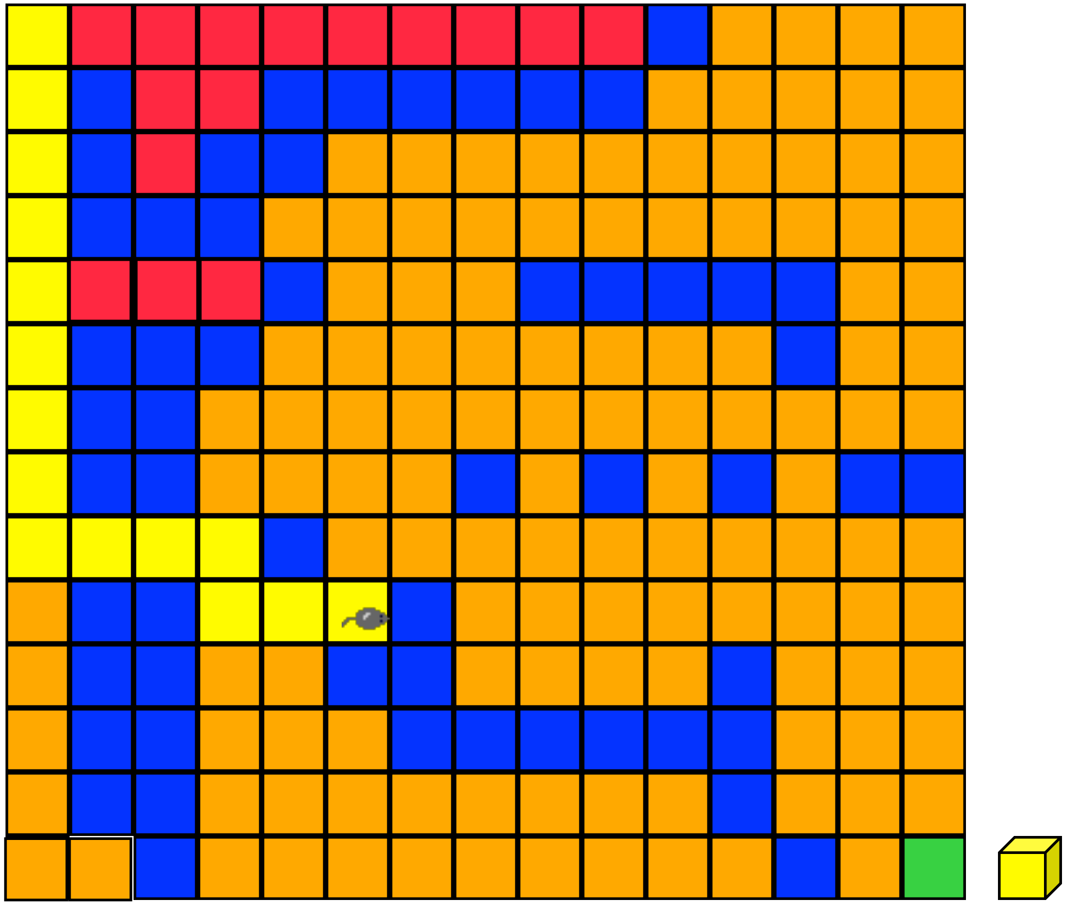
- Move right.
- Backtrack.

Rat In A Maze Demo



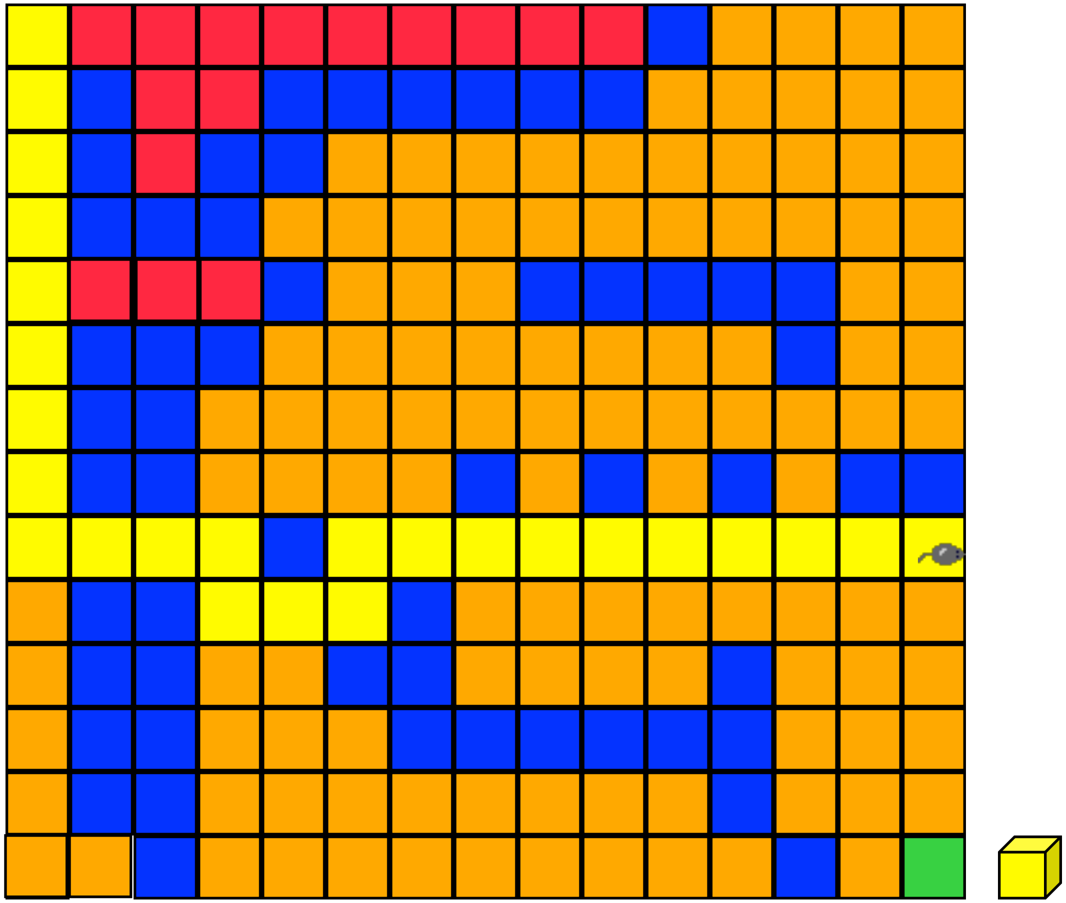
- Move one down and then right.

Rat In A Maze Demo



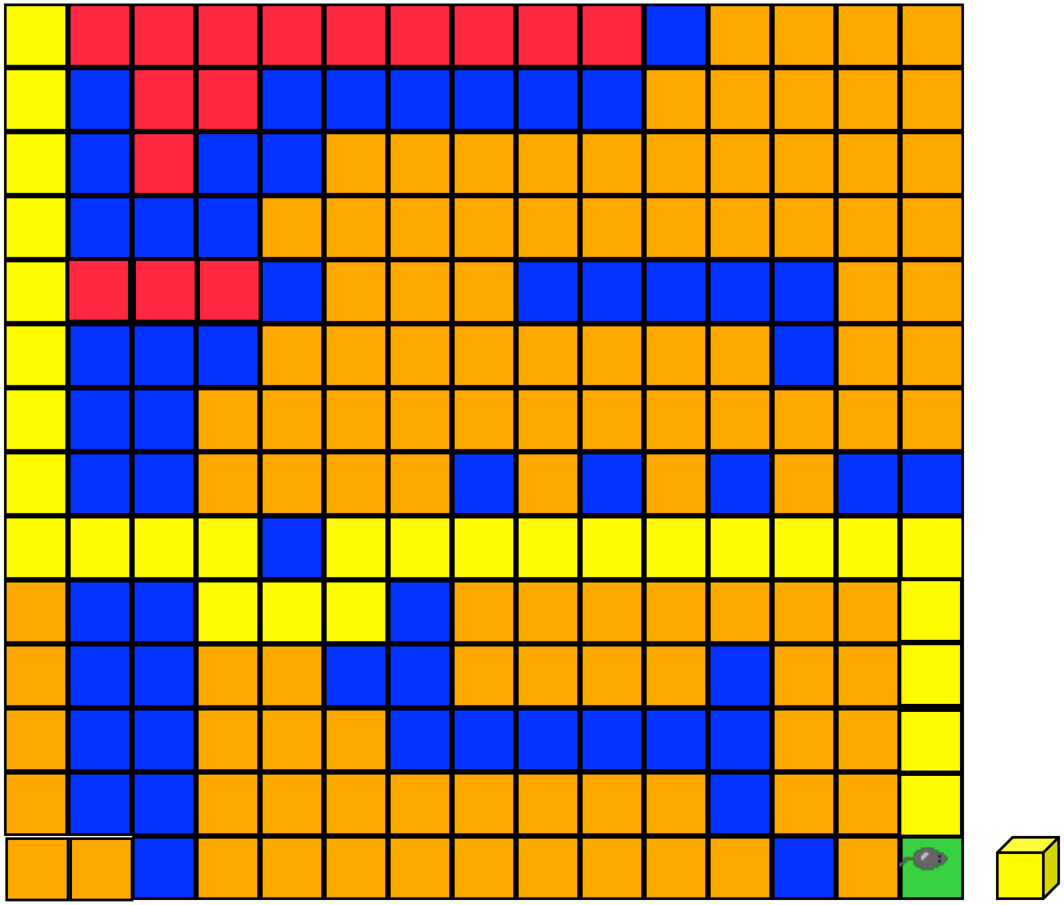
- Move one up and then right.

Rat In A Maze Demo

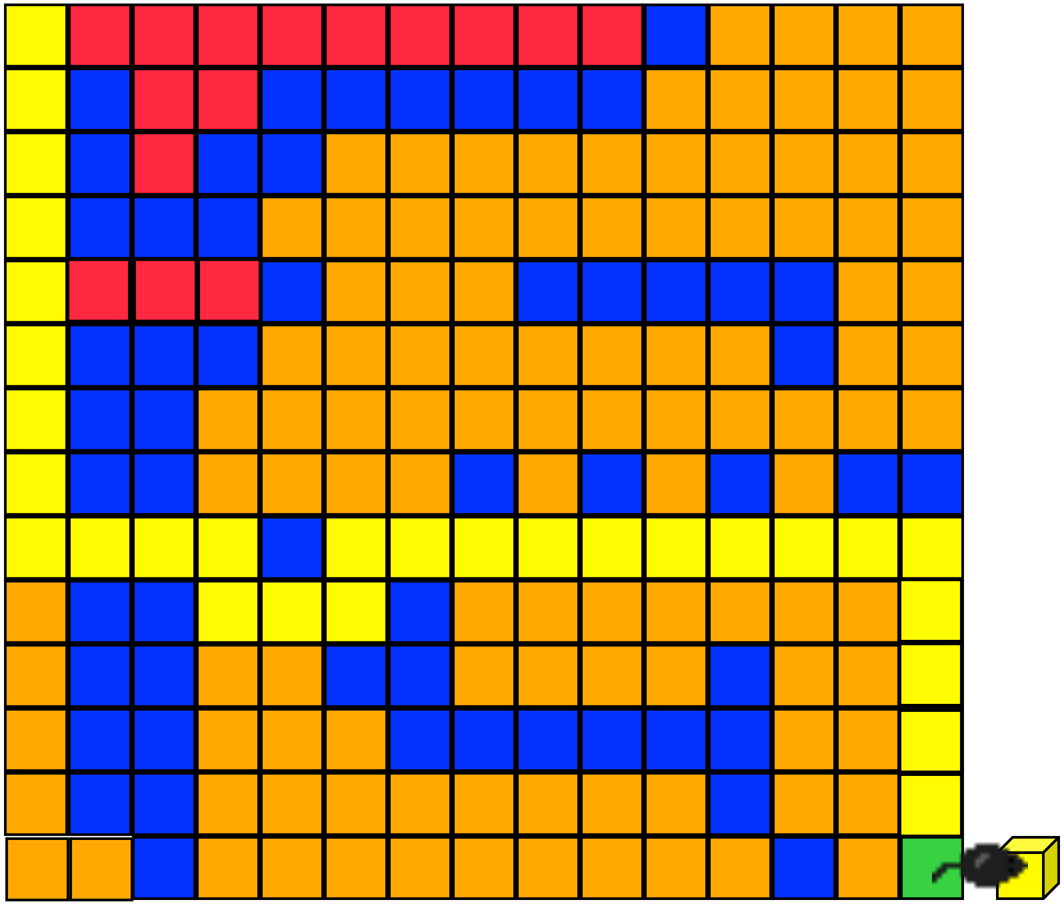


- Move down to exit and eat cheese.

Rat In A Maze Demo



Rat In A Maze Demo



- Path from maze entry to current position operates as a stack.

Rat In A Maze - Java code (1)

```
/** find a path from (1,1) to the exit (size, size)
    * @return true if successful, false if impossible */
  private static boolean findPath()
  {
     path = new StackArray();

     // initialize offsets
     Position [] offset = new Position [4];
     offset[0] = new Position(0, 1);   // right
     offset[1] = new Position(1, 0);   // down
     offset[2] = new Position(0, -1);  // left
     offset[3] = new Position(-1, 0);  // up

     // initialize wall of obstacles around maze
     for (int i = 0; i <= size + 1; i++)
     {
        maze[0][i] = maze[size + 1][i] = 1; // bottom and top
        maze[i][0] = maze[i][size + 1] = 1; // left and right
     }
```

Rat In A Maze - Java code (2)

```
Position here = new Position(1, 1);
maze[1][1] = 1; // prevent return to entrance
int option = 0; // next move
int lastOption = 3;

// search for a path
while (here.row != size || here.col != size)
{   // not at exit
    // find a neighbor to move to
    // won't compile without explicit initialization
    int r = 0, c = 0;   // row and column of neighbor
    while (option <= lastOption)
    {
       r = here.row + offset[option].row;
       c = here.col + offset[option].col;
       if (maze[r][c] == 0) break;
       option++; // next option
    }
```

Rat In A Maze - Java code (3)

```
       // was a neighbor found?
       if (option <= lastOption)  // yes
       {  // move to maze[r][c]
          path.push(here);
          here = new Position(r, c);
          // set to 1 to prevent revisit
          maze[r][c] = 1;
          option = 0;
       }
       else
       {  // no neighbor to move to, back up
          if (path.isEmpty()) return false;  // no place to
                                             // back up to
          Position next = (Position) path.pop();
          if (next.row == here.row)
             option = 2 + next.col - here.col;
          else
             option = 3 + next.row - here.row;
          here = next;
       }
       return true;  // at exit
    }
```

Bus Stop Queue

Bus Stop

front

rear

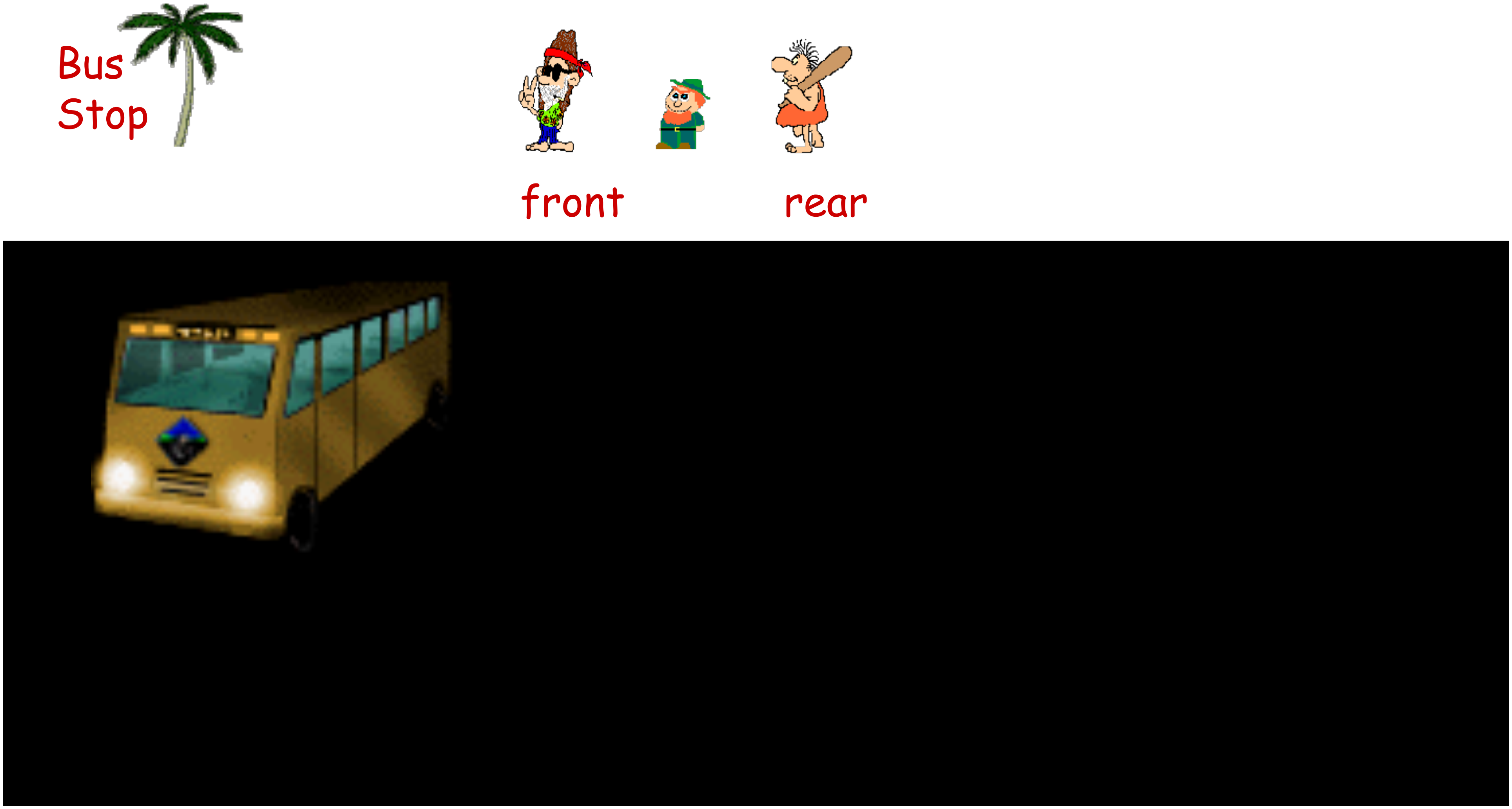
Bus Stop Queue

Bus Stop

front

rear

Bus Stop Queue

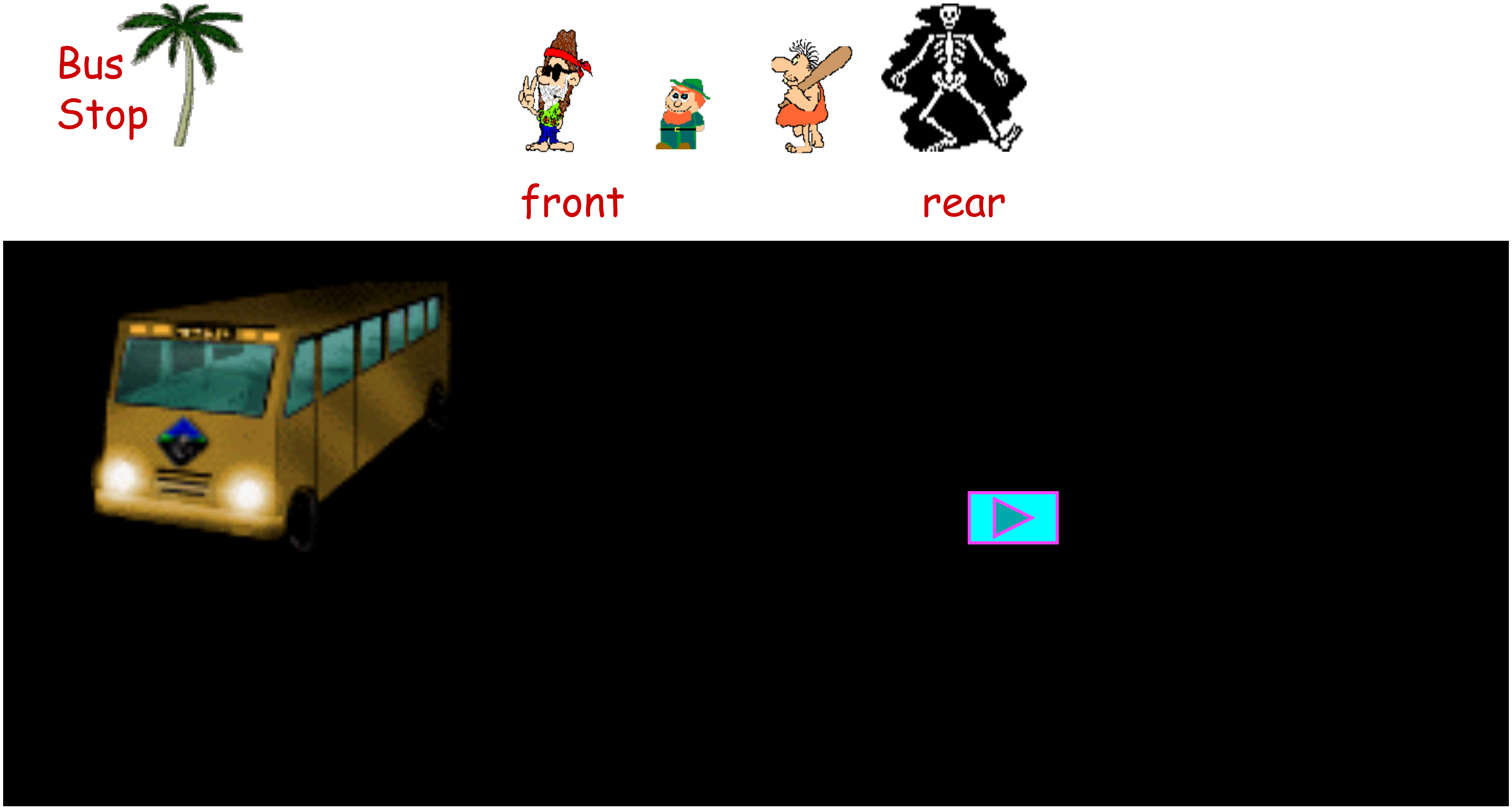


Bus Stop

front

rear

Bus Stop Queue



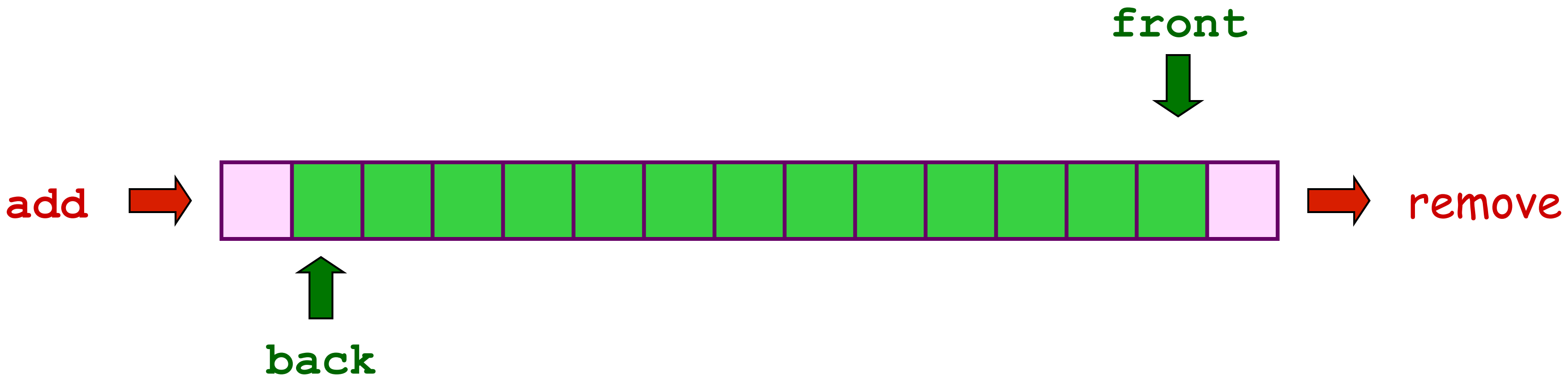
Bus Stop

front

rear

Queue

- Queue operations.
  - add (enqueue) Insert a new object onto queue.
  - remove (dequeue) Delete and return the object least recently added.
  - isEmpty Is the queue empty?

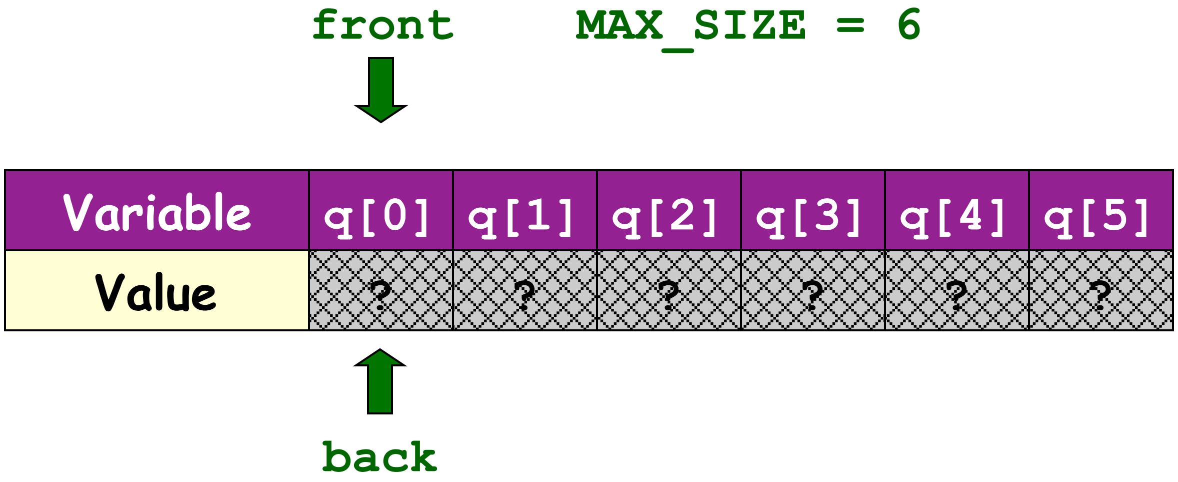


Queue Implementation with Arrays - Demo

```
q.init();
```

Queue Implementation with Arrays - Demo

```
q.init();
```



front

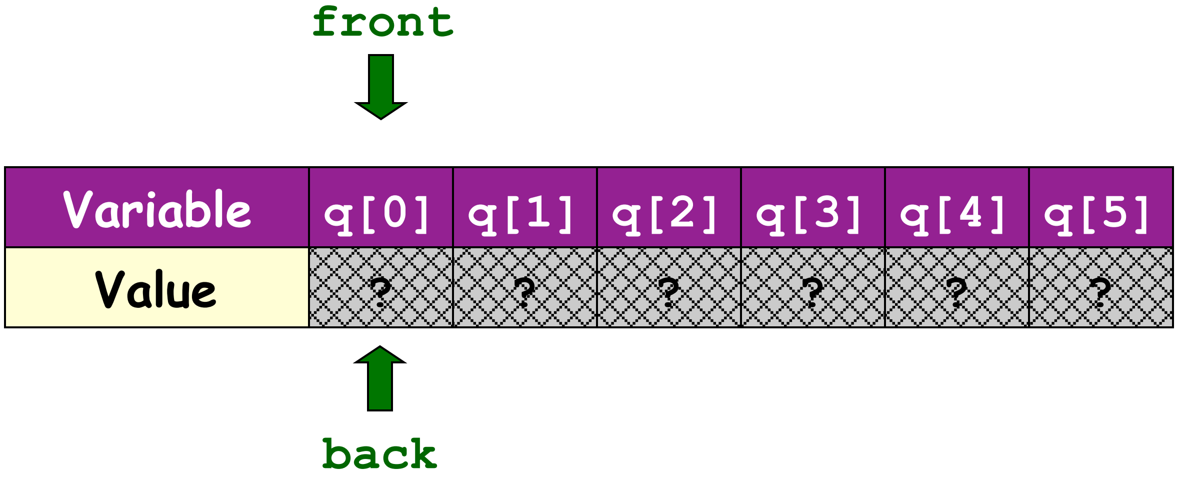
MAX_SIZE = 6

Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	?	?	?	?	?	?

back

Queue Implementation with Arrays - Demo

```
q.init();
q.add('A');
```



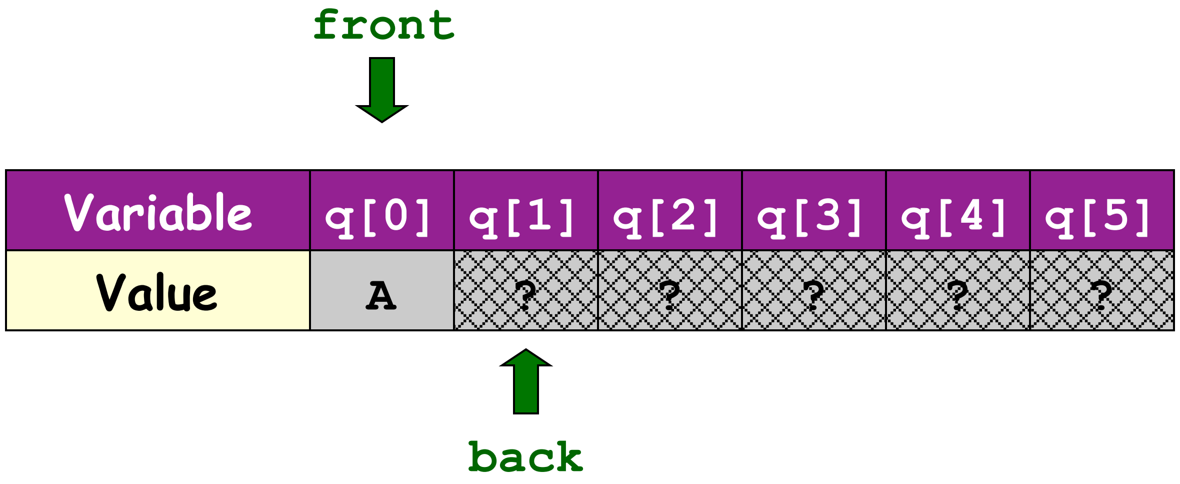
front

Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	?	?	?	?	?	?

back

Queue Implementation with Arrays - Demo

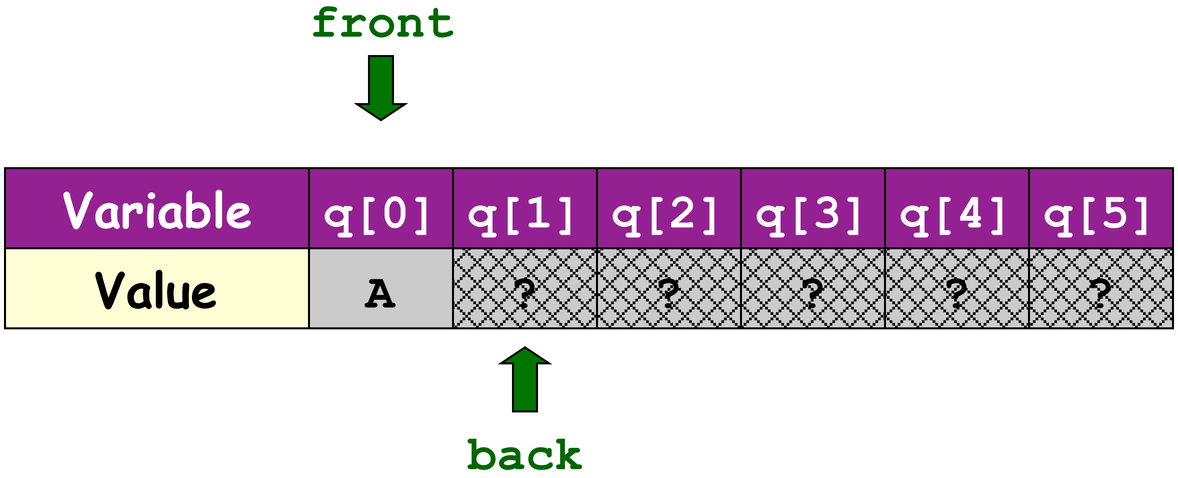
```
q.init();
q.add('A');
```



Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	A	?	?	?	?	?

Queue Implementation with Arrays - Demo

```
q.init();
q.add('A');
q.add('D');
```



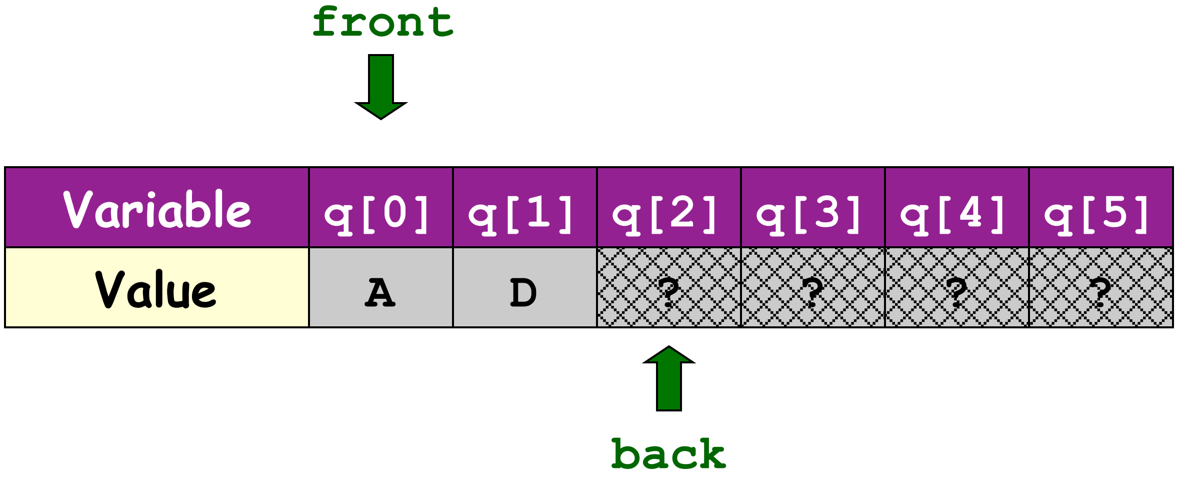
front

Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	A	?	?	?	?	?

back

Queue Implementation with Arrays – Demo

```
q.init();
q.add('A');
q.add('D');
```



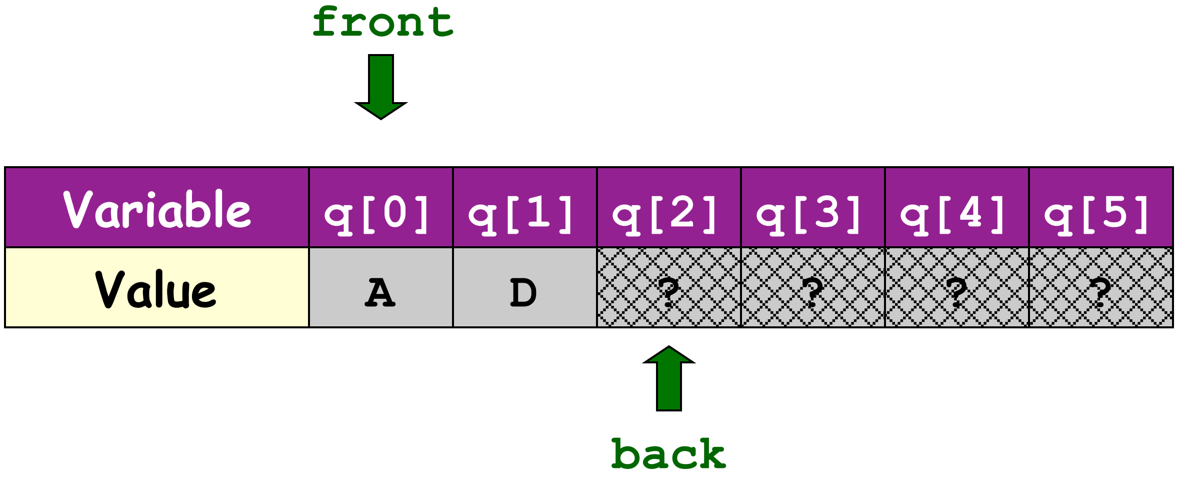
front

Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	A	D	?	?	?	?

back

Queue Implementation with Arrays - Demo

```
q.init();
q.add('A');
q.add('D');
q.add('T');
```



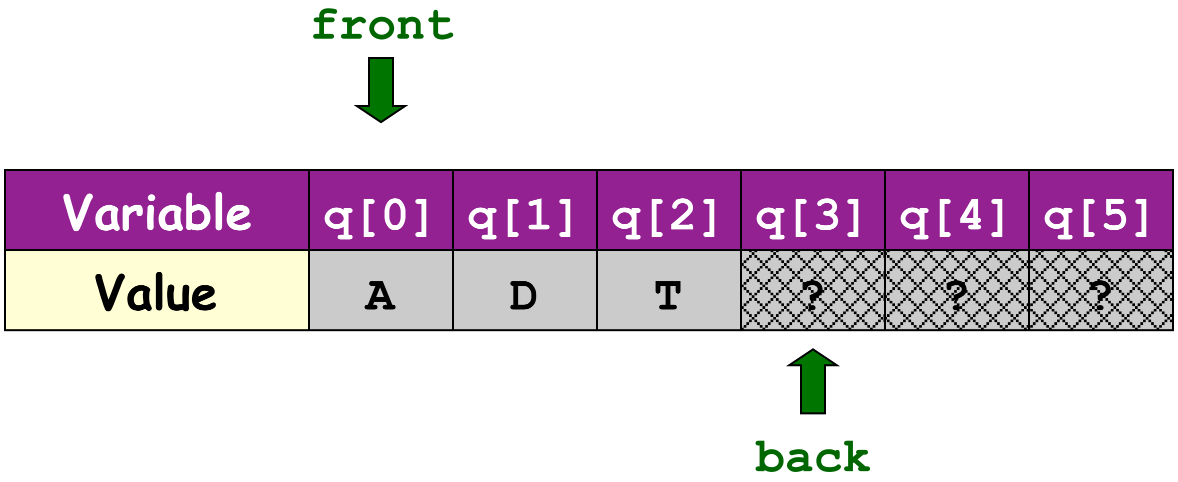
front

Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	A	D	?	?	?	?

back

Queue Implementation with Arrays - Demo

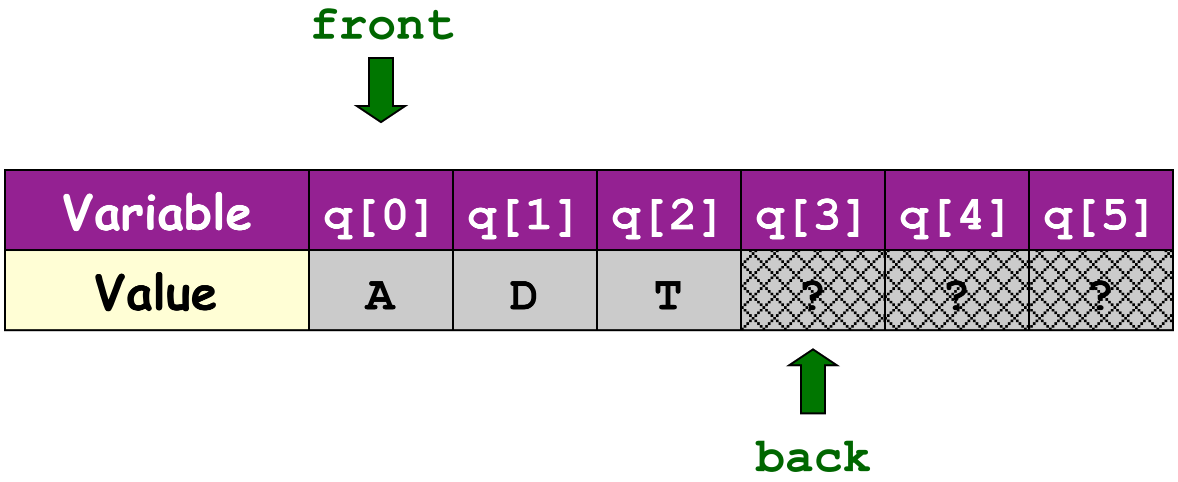
```
q.init();
q.add('A');
q.add('D');
q.add('T');
```



Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	A	D	T	?	?	?

Queue Implementation with Arrays - Demo

```
q.init();
q.add('A');
q.add('D');
q.add('T');
q.add('E');
```



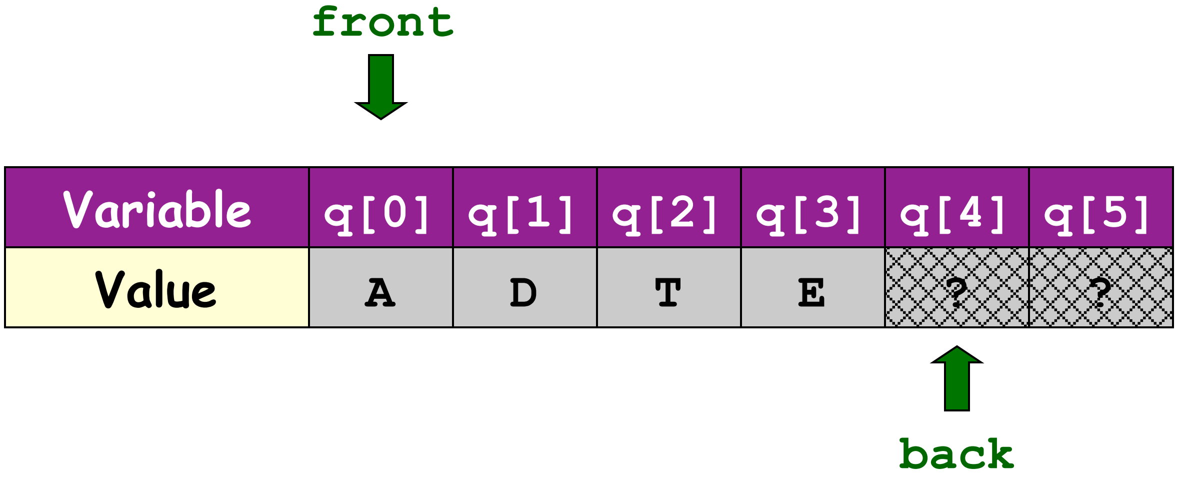
front

Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	A	D	T	?	?	?

back

Queue Implementation with Arrays - Demo

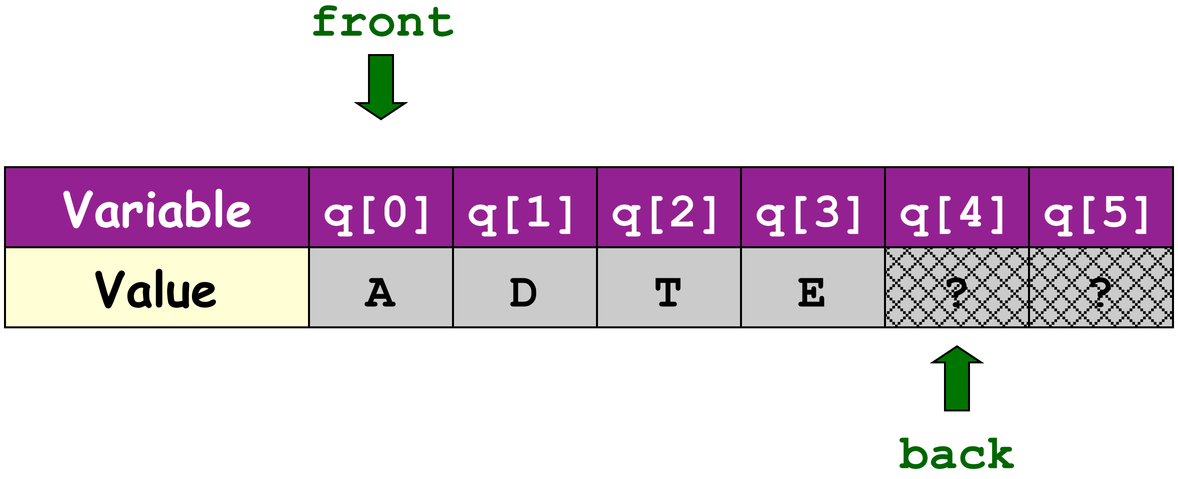
```
q.init();
q.add('A');
q.add('D');
q.add('T');
q.add('E');
```



Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	A	D	T	E	?	?

Queue Implementation with Arrays - Demo

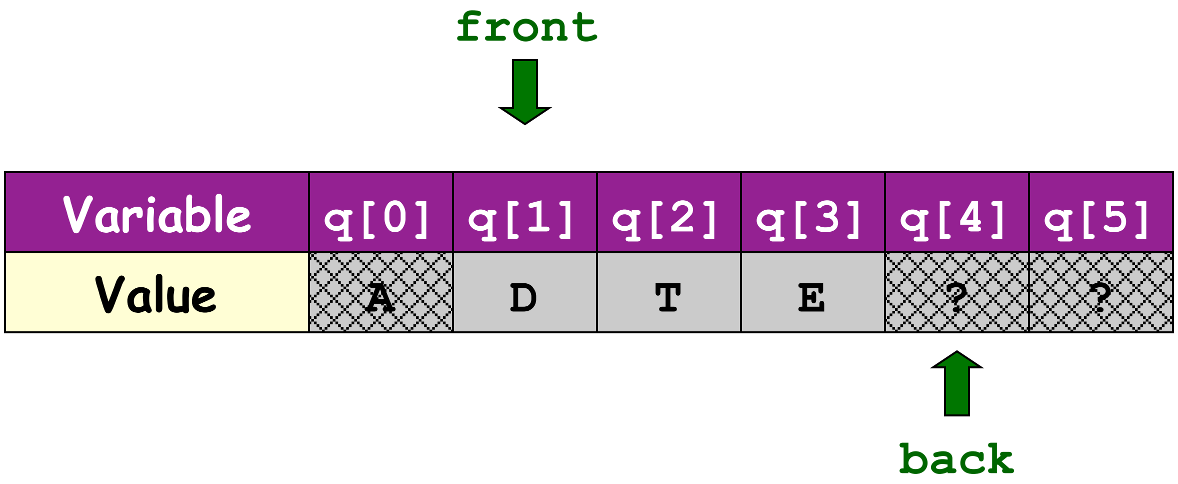
```
q.init();
q.add('A');
q.add('D');
q.add('T');
q.add('E');
q.add();
```



Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	A	D	T	E	?	?

Queue Implementation with Arrays - Demo

```
q.init();
q.add('A');
q.add('D');
q.add('T');
q.add('E');
q.remove();
```

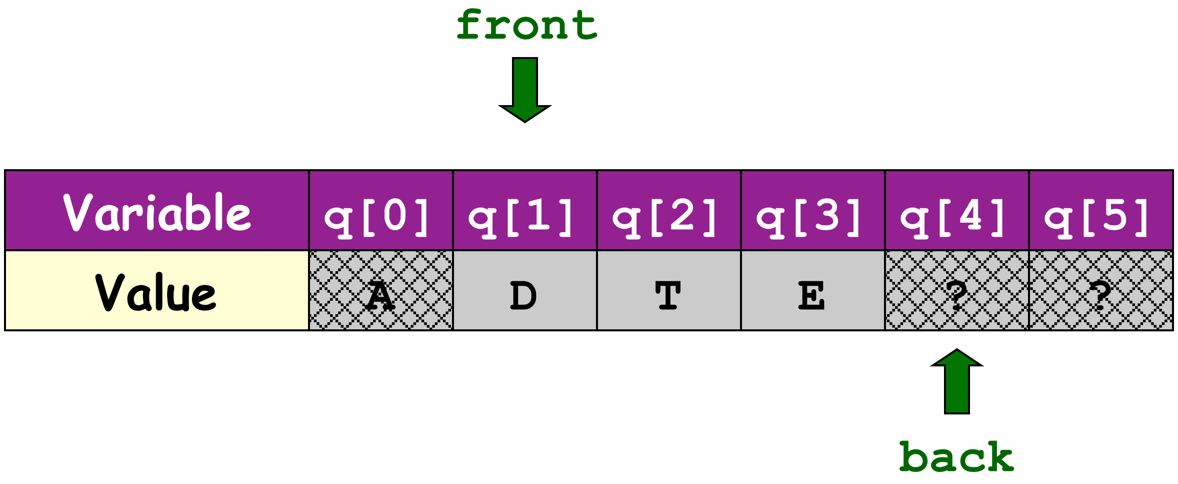


Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	A	D	T	E	?	?

```
Items dequeued: A
```

Queue Implementation with Arrays - Demo

```
q.init();
q.add('A');
q.add('D');
q.add('T');
q.add('E');
q.remove();
q.remove();
```

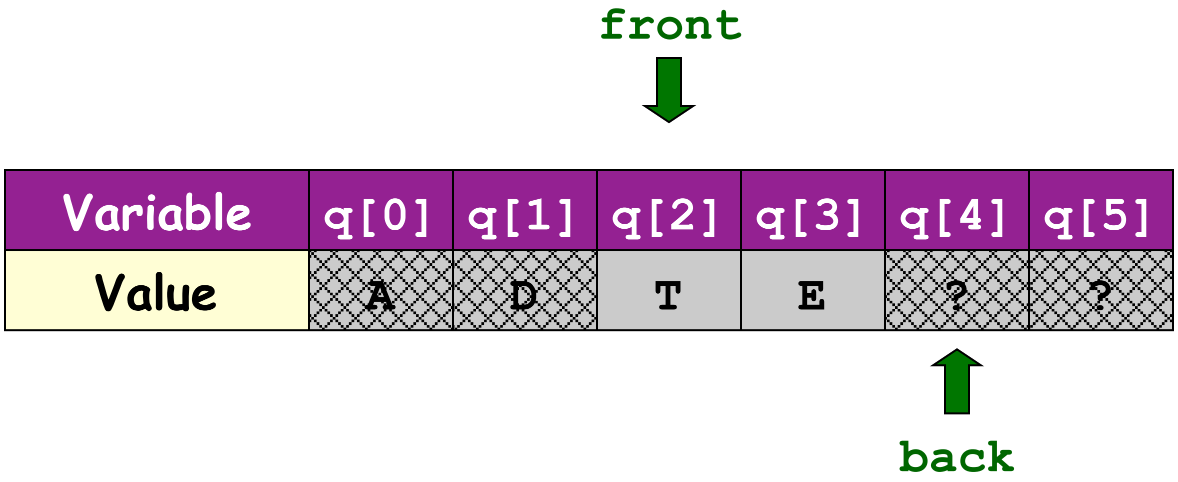


Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	A	D	T	E	?	?

```
Items dequeued: A
```

Queue Implementation with Arrays - Demo

```
q.init();
q.add('A');
q.add('D');
q.add('T');
q.add('E');
q.remove();
q.remove();
```

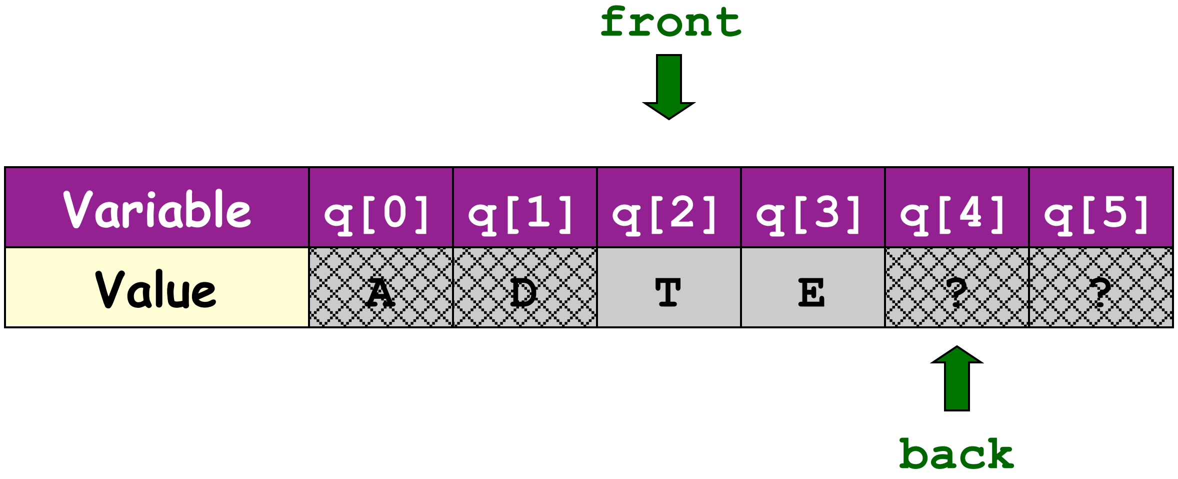


Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	A	D	T	E	?	?

Items dequeued: A D

Queue Implementation with Arrays - Demo

```
q.init();
q.add('A');
q.add('D');
q.add('T');
q.add('E');
q.remove();
q.remove();
q.remove();
```



front

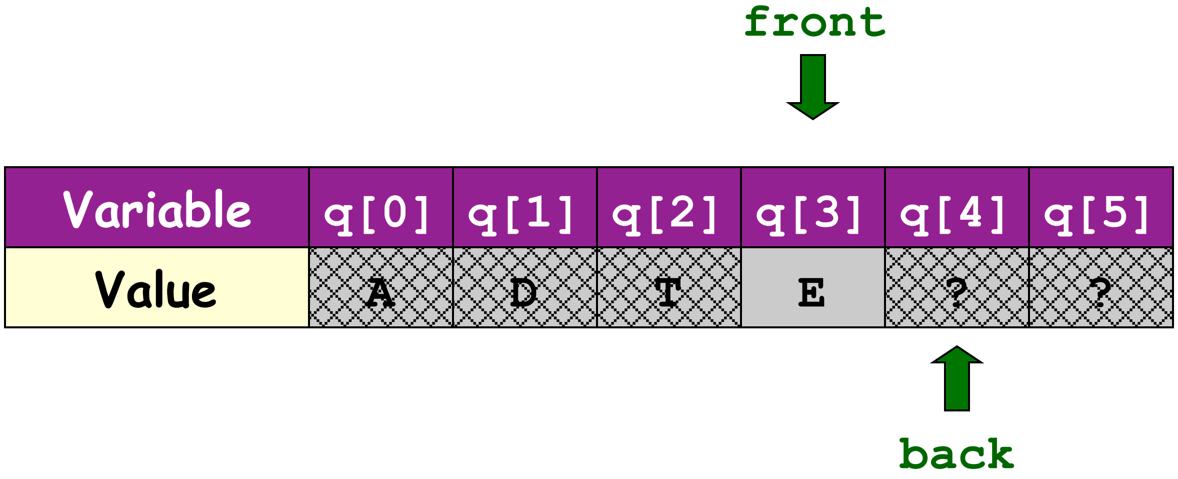
Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	A	D	T	E	?	?

back

```
Items dequeued: A D
```

Queue Implementation with Arrays - Demo

```
q.init();
q.add('A');
q.add('D');
q.add('T');
q.add('E');
q.remove();
q.remove();
q.remove();
```



front

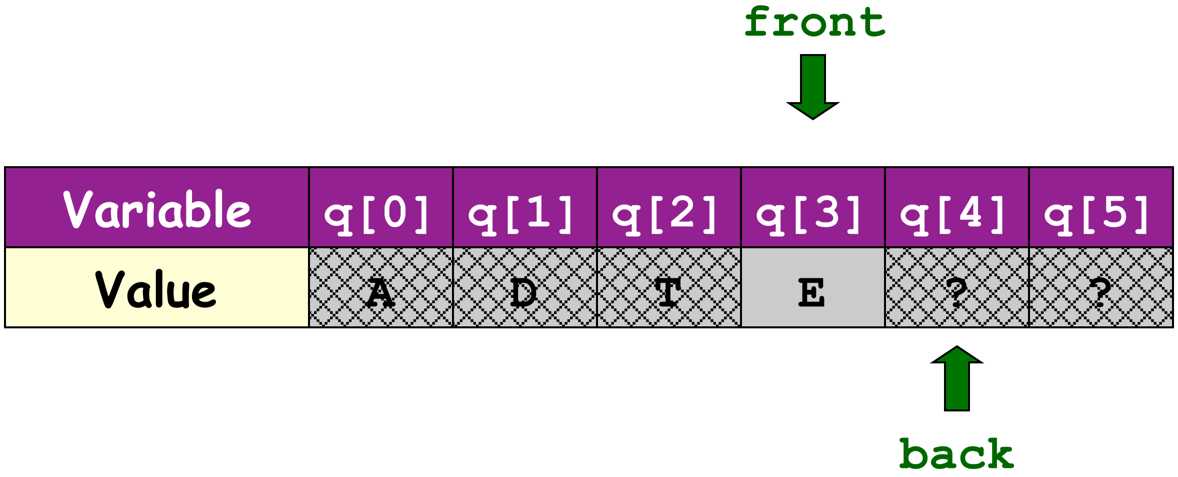
Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	▨ A	▨ D	▨ T	E	▨ ?	▨ ?

back

```
Items dequeued: A D T
```

Queue Implementation with Arrays - Demo

```
q.init();
q.add('A');
q.add('D');
q.add('T');
q.add('E');
q.remove();
q.remove();
q.remove();
q.add('X');
```



front

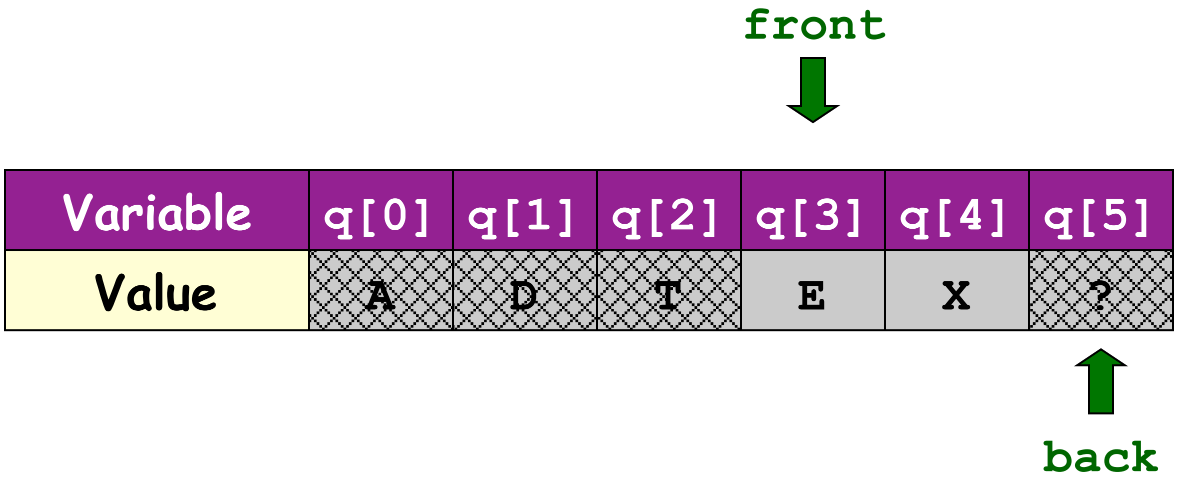
Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	A	D	T	E	?	?

back

```
Items dequeued: A D T
```

Queue Implementation with Arrays - Demo

```
q.init();
q.add('A');
q.add('D');
q.add('T');
q.add('E');
q.remove();
q.remove();
q.remove();
q.add('X');
```

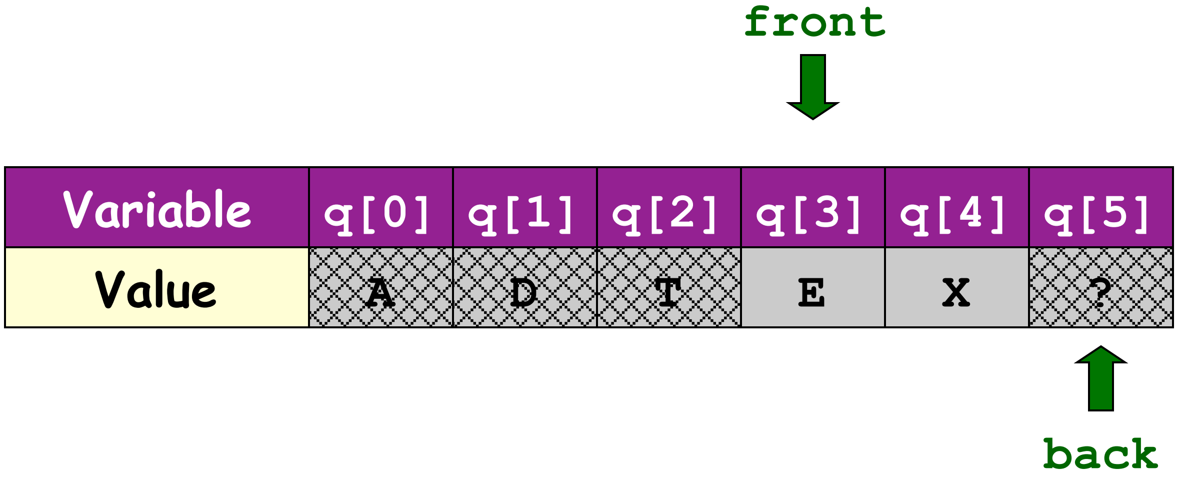


Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	A	D	T	E	X	?

Items dequeued: A D T

Queue Implementation with Arrays - Demo

```
q.init();
q.add('A');
q.add('D');
q.add('T');
q.add('E');
q.remove();
q.remove();
q.remove();
q.add('X');
q.add('A');
```



front

Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	A	D	T	E	X	?

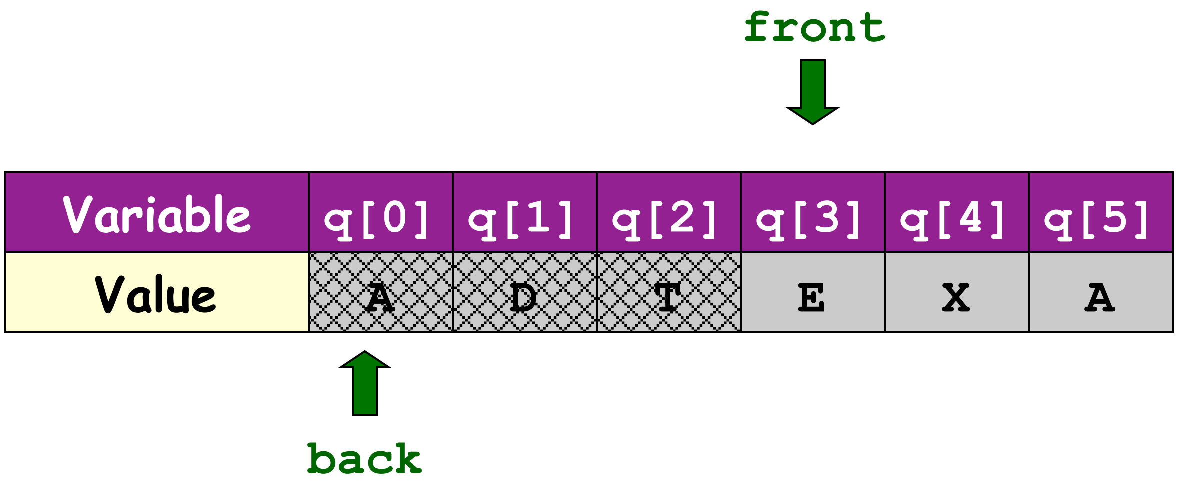
back

```
Items dequeued: A D T
```

Queue Implementation with Arrays - Demo

```
q.init();
q.add('A');
q.add('D');
q.add('T');
q.add('E');
q.remove();
q.remove();
q.remove();
q.add('X');
q.add('A');
```

```
// modulo function takes care of
// wrapping around front & back
back = (back + 1) % q.size
```

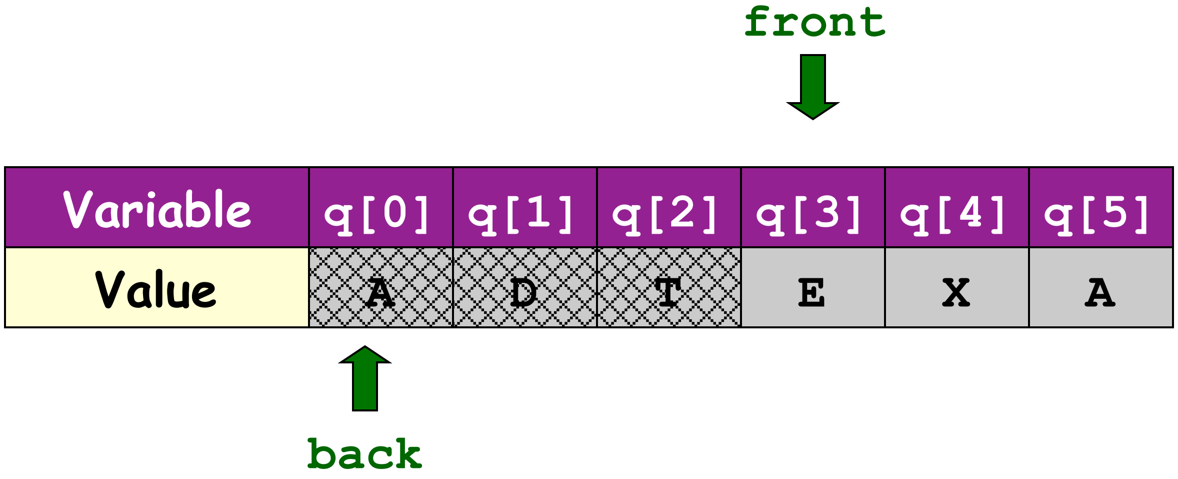


Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	A	D	T	E	X	A

```
Items dequeued: A D T
```

Queue Implementation with Arrays - Demo

```
q.init();
q.add('A');
q.add('D');
q.add('T');
q.add('E');
q.remove();
q.remove();
q.remove();
q.add('X');
q.add('A');
q.add('M');
```



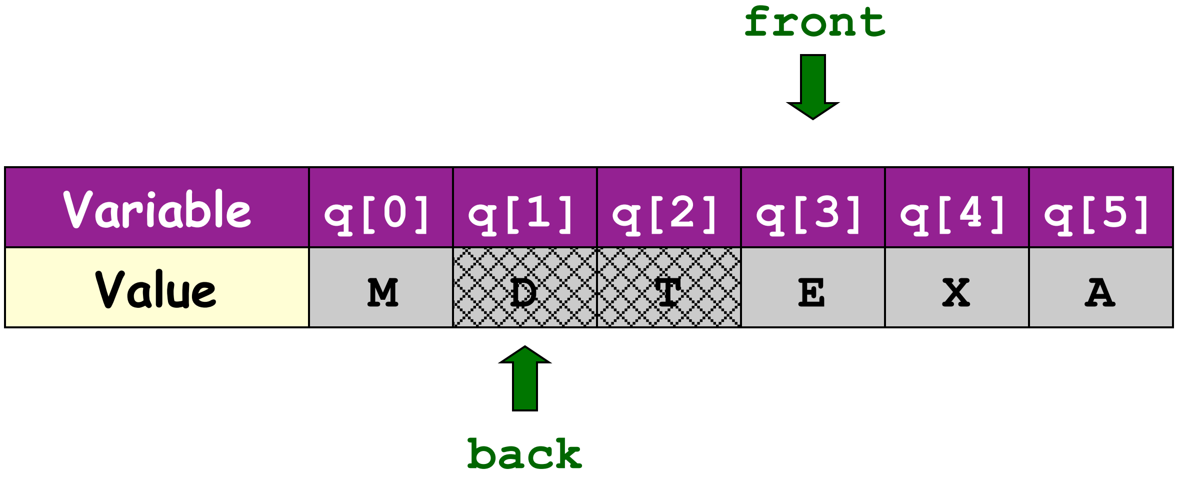
Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	A	D	T	E	X	A

Items dequeued: A D T

Queue Implementation with Arrays - Demo

```
q.init();
q.add('A');
q.add('D');
q.add('T');
q.add('E');
q.remove();
q.remove();
q.remove();
q.add('X');
q.add('A');
q.add('M');
```

wrap-around

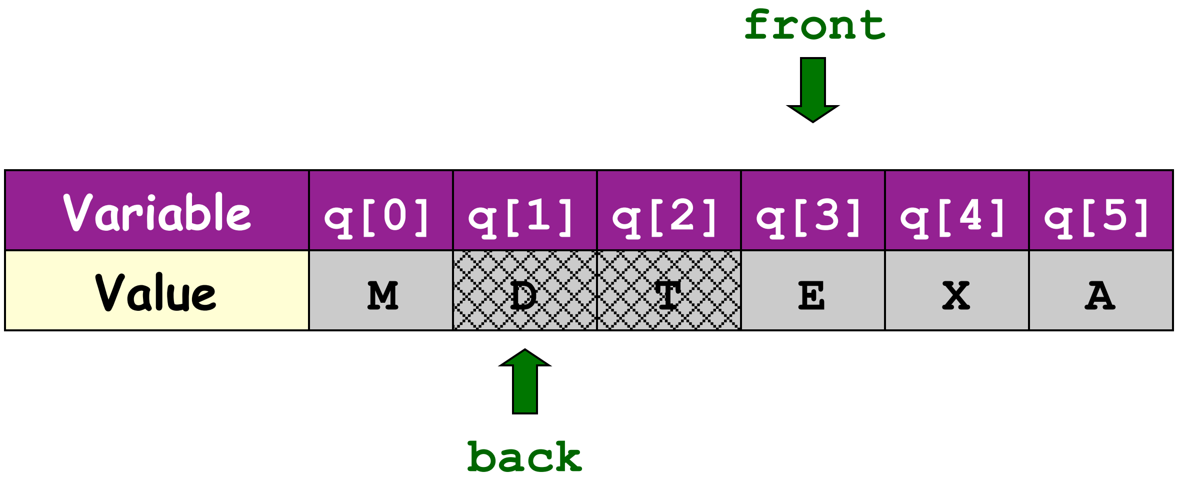


Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	M	D	T	E	X	A

```
Items dequeued: A D T
```

Queue Implementation with Arrays - Demo

```
q.init();
q.add('A');
q.add('D');
q.add('T');
q.add('E');
q.remove();
q.remove();
q.remove();
q.add('X');
q.add('A');
q.add('M');
q.remove();
```

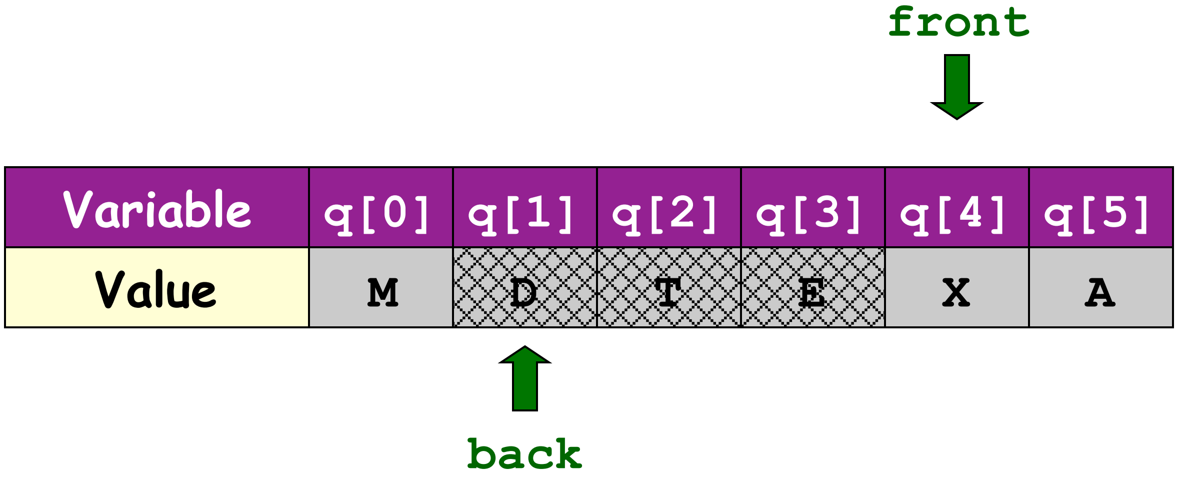


Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	M	D	T	E	X	A

Items dequeued: A D T

Queue Implementation with Arrays - Demo

```
q.init();
q.add('A');
q.add('D');
q.add('T');
q.add('E');
q.remove();
q.remove();
q.remove();
q.add('X');
q.add('A');
q.add('M');
q.remove();
```



front

Variable	q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
Value	M	D	T	E	X	A

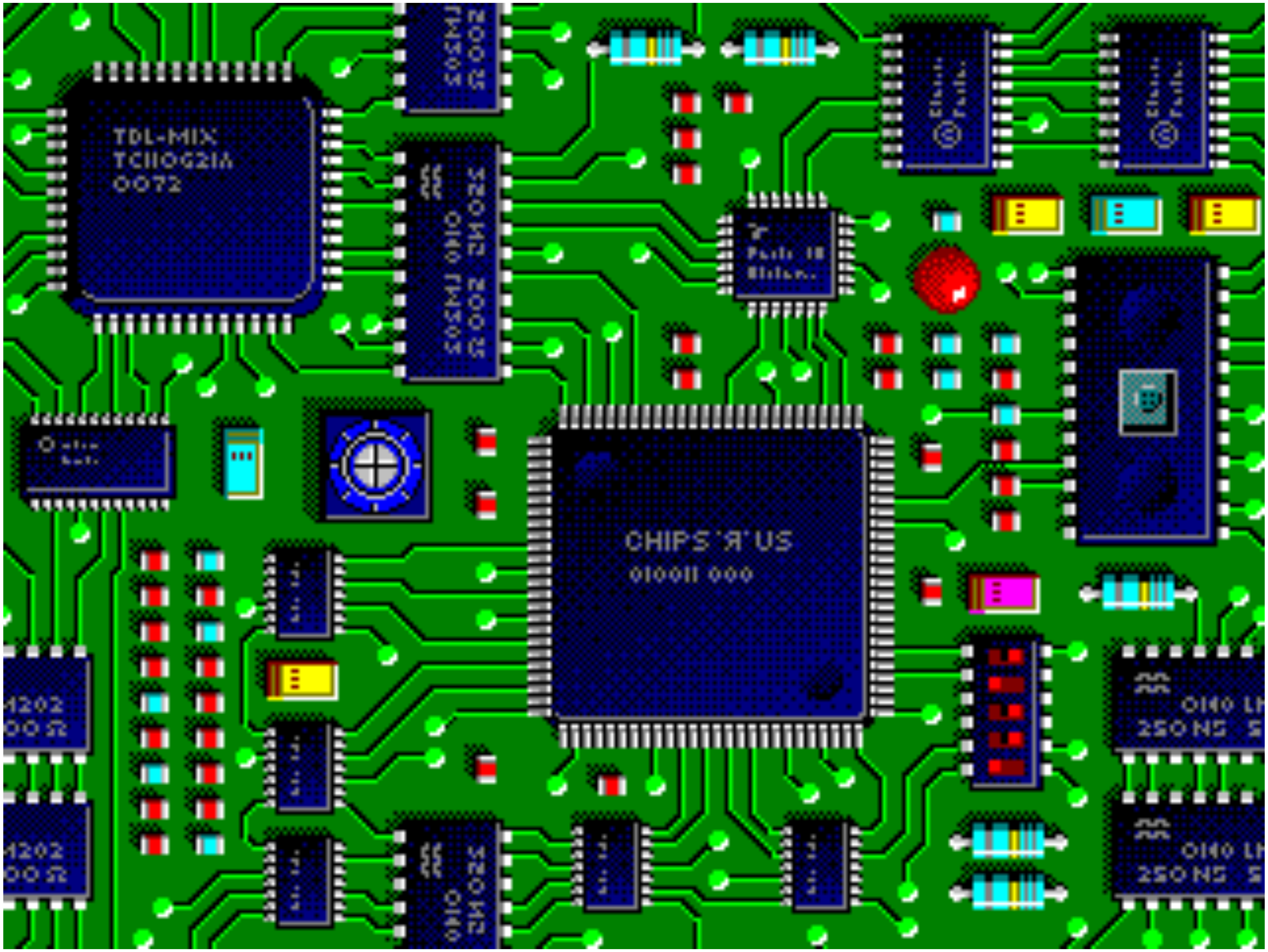
back

Items dequeued: A D T E

Queue: Linked List Implementation

```
public class QueueList implements QueueInterface {
   private List first;          ⇦ reference to first element of queue
   private List last;           ⇦ reference to last element of queue

   private class List { String item; List next; }   ⇦ nested class

   public boolean isEmpty() { return (first == null); }

   public void add(String anItem) {
      List x = new List();
      x.item = anItem;
      x.next = null;
      if (isEmpty()) { head = x; last = x; }
      else           { last.next = x; last = x; }
   }

   public String remove() {
      String val = first.item;
      first = first.next;       ⇦ delete first element
      return val;
   }
}
```

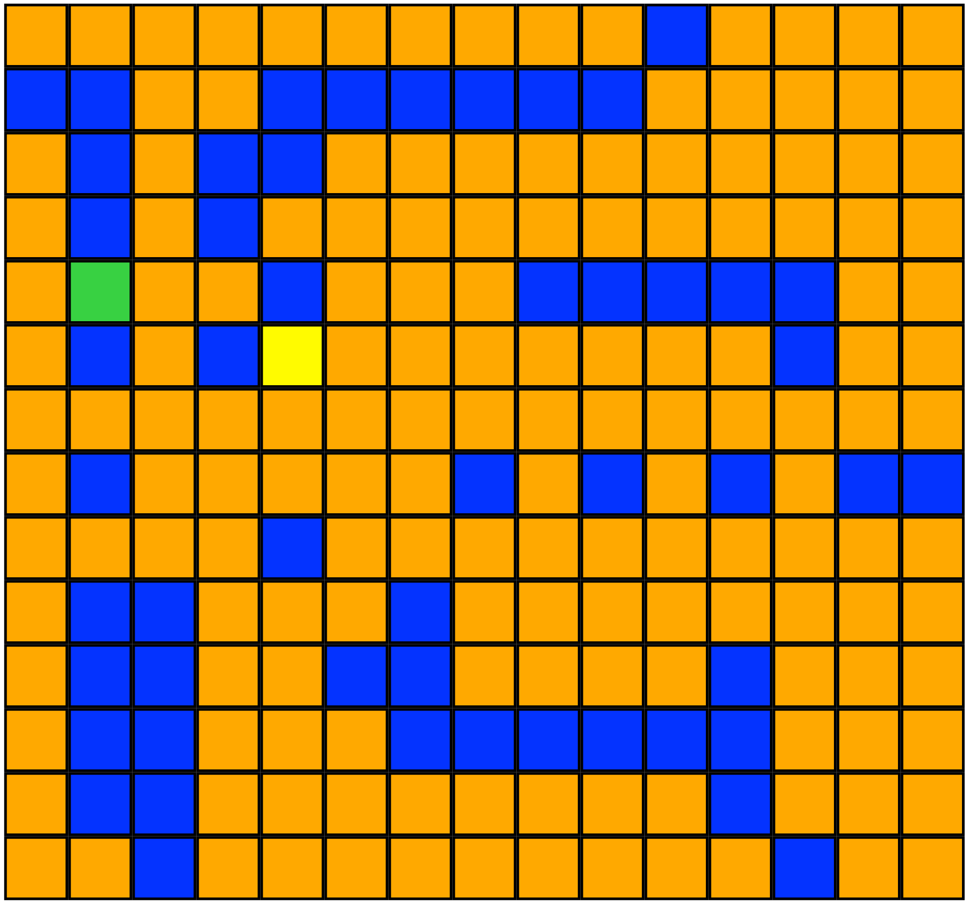
Lee's Wire Router



Lee's Wire Router

🟩 start pin

🟨 end pin

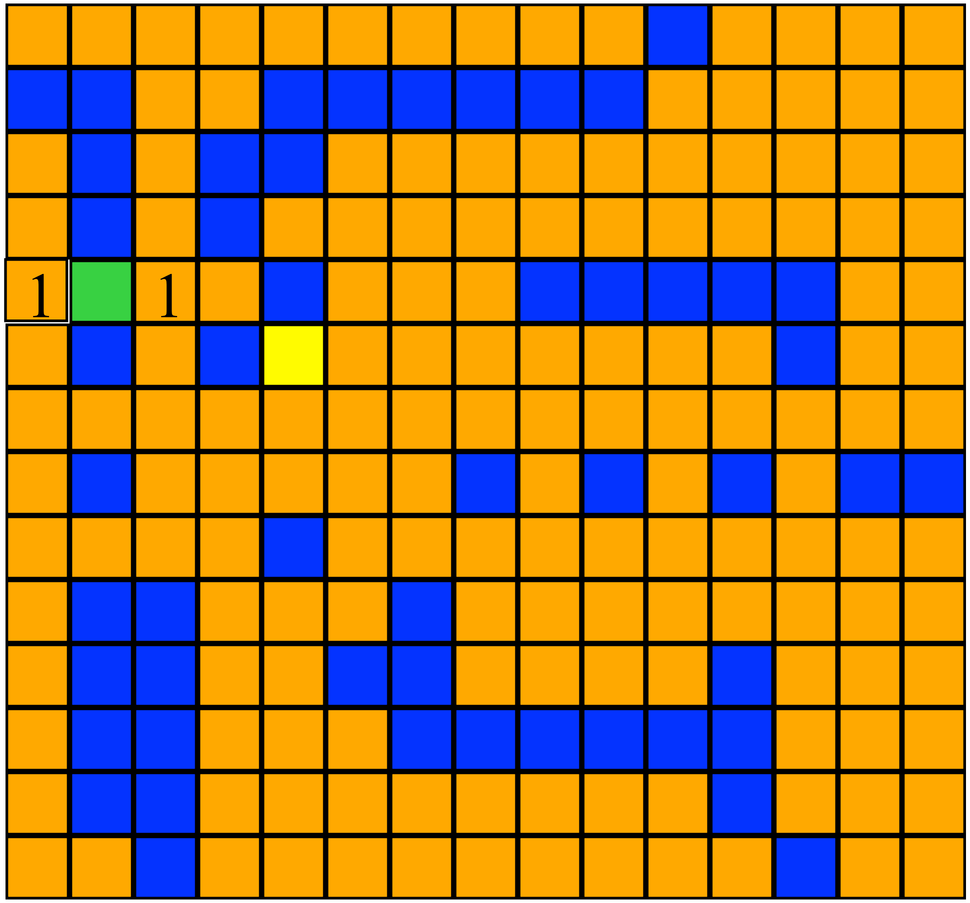


Label all reachable squares 1 unit from start.

Lee's Wire Router

🟩 start pin

🟨 end pin

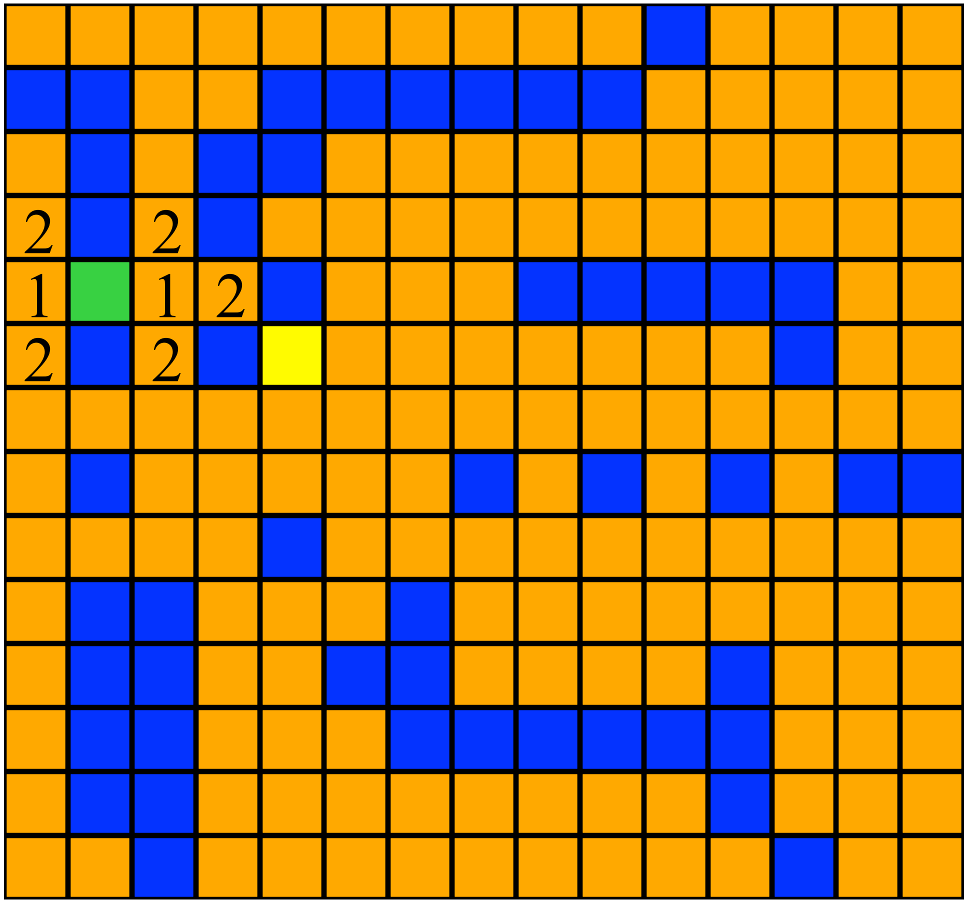


Label all reachable unlabeled squares 2 units from start.

Lee's Wire Router

🟩 start pin

🟨 end pin

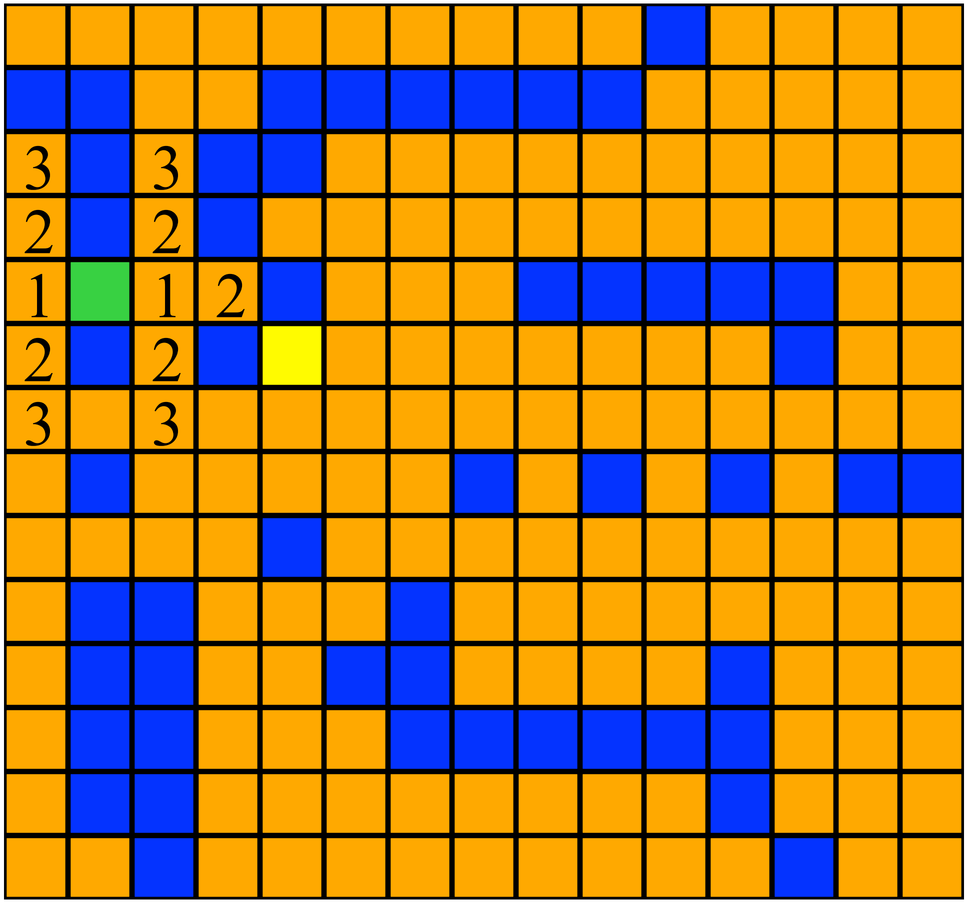


Label all reachable unlabeled squares 3 units from start.

Lee's Wire Router

🟩 start pin

🟨 end pin



🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟦	🟧	🟧	🟧	🟧
🟦	🟦	🟧	🟧	🟦	🟦	🟦	🟦	🟦	🟦	🟧	🟧	🟧	🟧	🟧
3	🟦	3	🟦	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
2	🟦	2	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
1	🟩	1	2	🟦	🟧	🟧	🟧	🟦	🟦	🟦	🟦	🟦	🟧	🟧
2	🟦	2	🟦	🟨	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟦	🟧	🟧
3	🟧	3	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
🟧	🟦	🟧	🟧	🟧	🟧	🟧	🟦	🟧	🟦	🟧	🟦	🟧	🟦	🟦
🟧	🟧	🟧	🟧	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
🟧	🟦	🟦	🟧	🟧	🟧	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
🟧	🟦	🟦	🟧	🟧	🟦	🟦	🟧	🟧	🟧	🟧	🟦	🟧	🟧	🟧
🟧	🟦	🟦	🟧	🟧	🟧	🟦	🟦	🟦	🟦	🟦	🟦	🟧	🟧	🟧
🟧	🟦	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟦	🟧	🟧	🟧
🟧	🟧	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟦	🟧	🟧

Label all reachable unlabeled squares 4 units from start.

Lee's Wire Router

🟩 start pin

🟨 end pin

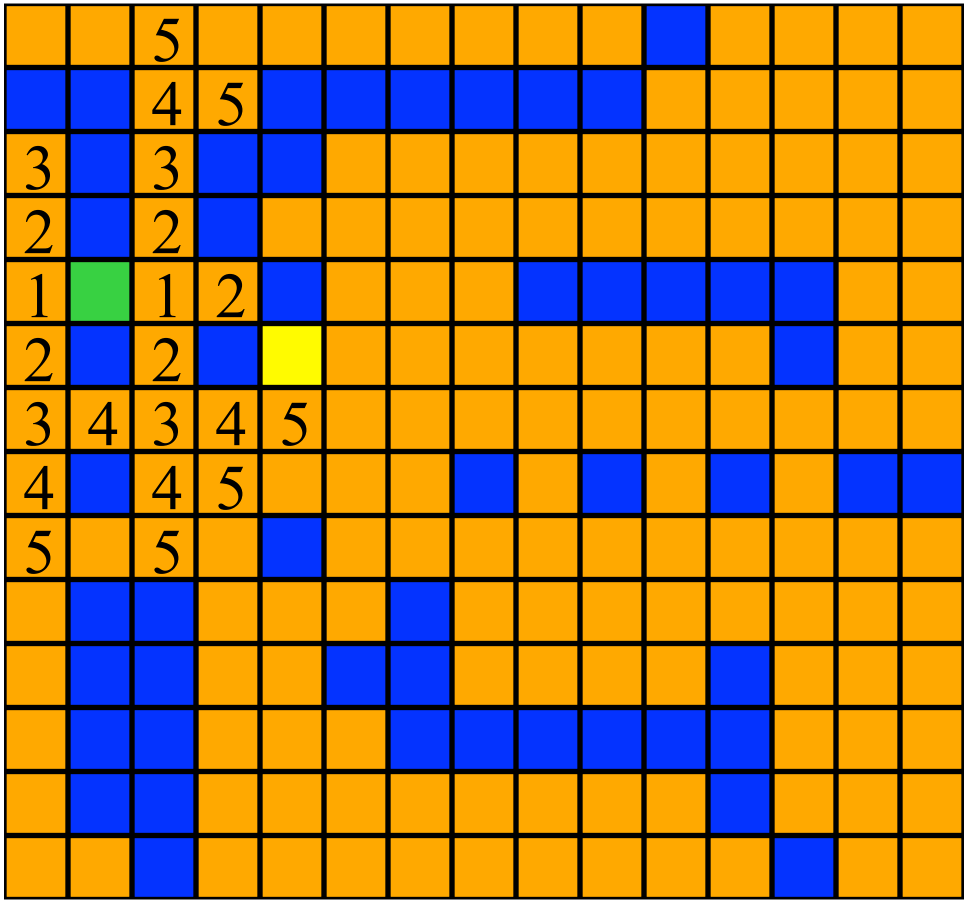
🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟦	🟧	🟧	🟧	🟧
🟦	🟦	4	🟧	🟦	🟦	🟦	🟦	🟦	🟦	🟧	🟧	🟧	🟧	🟧
3	🟦	3	🟦	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
2	🟦	2	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
1	🟩	1	2	🟦	🟧	🟧	🟧	🟦	🟦	🟦	🟦	🟦	🟧	🟧
2	🟦	2	🟦	🟨	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟦	🟧	🟧
3	4	3	4	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
4	🟦	4	🟧	🟧	🟧	🟧	🟦	🟧	🟦	🟧	🟦	🟧	🟦	🟦
🟧	🟧	🟧	🟧	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
🟧	🟦	🟦	🟧	🟧	🟧	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
🟧	🟦	🟦	🟧	🟧	🟦	🟦	🟧	🟧	🟧	🟧	🟦	🟧	🟧	🟧
🟧	🟦	🟦	🟧	🟧	🟧	🟦	🟦	🟦	🟦	🟦	🟦	🟧	🟧	🟧
🟧	🟦	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟦	🟧	🟧	🟧
🟧	🟧	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟦	🟧	🟧

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

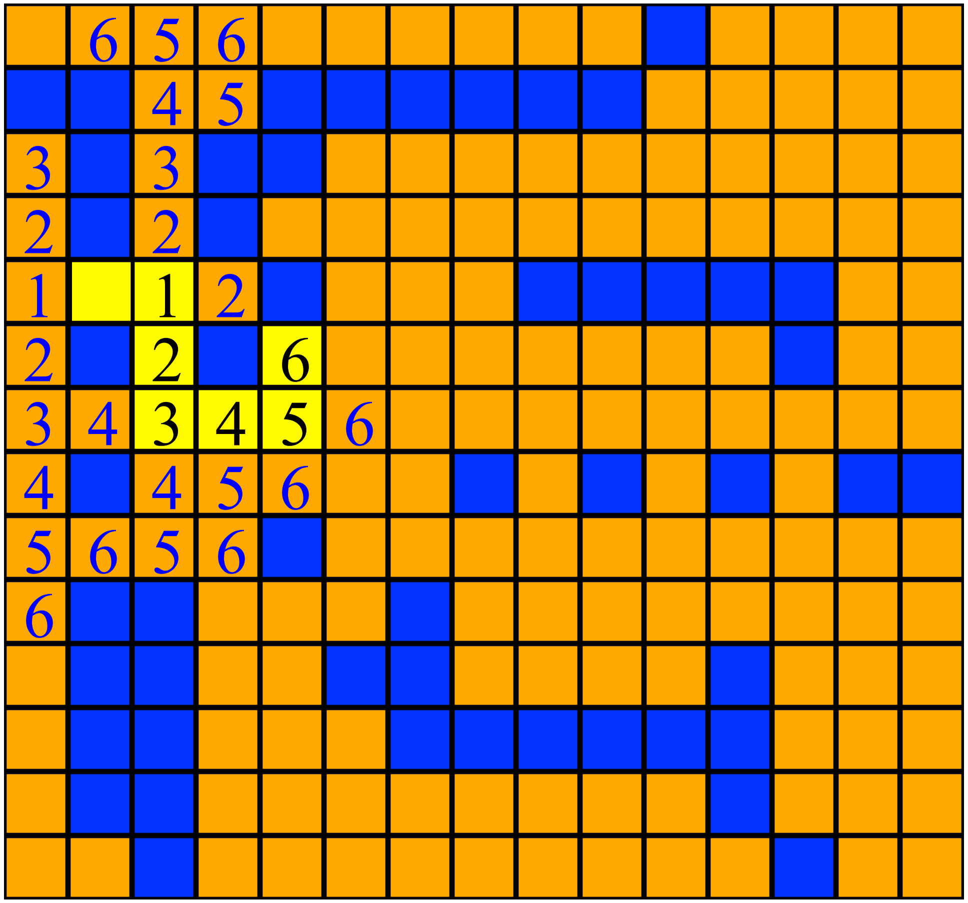
🟧	6	5	6	🟧	🟧	🟧	🟧	🟧	🟧	🟦	🟧	🟧	🟧	🟧
🟦	🟦	4	5	🟦	🟦	🟦	🟦	🟦	🟦	🟧	🟧	🟧	🟧	🟧
3	🟦	3	🟦	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
2	🟦	2	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
1	🟩	1	2	🟦	🟧	🟧	🟧	🟦	🟦	🟦	🟦	🟦	🟧	🟧
2	🟦	2	🟦	🟨 6	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟦	🟧	🟧
3	4	3	4	5	6	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
4	🟦	4	5	6	🟧	🟧	🟦	🟧	🟦	🟧	🟦	🟧	🟦	🟦
5	6	5	6	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
6	🟦	🟦	🟧	🟧	🟧	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
🟧	🟦	🟦	🟧	🟧	🟦	🟦	🟧	🟧	🟧	🟧	🟦	🟧	🟧	🟧
🟧	🟦	🟦	🟧	🟧	🟧	🟦	🟦	🟦	🟦	🟦	🟦	🟧	🟧	🟧
🟧	🟦	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟦	🟧	🟧	🟧
🟧	🟧	🟦	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟦	🟧	🟧

End pin reached. Traceback.

Lee's Wire Router

🟩 start pin

🟨 end pin



	6	5	6							🟦				
🟦	🟦	4	5	🟦	🟦	🟦	🟦	🟦	🟦					
3	🟦	3	🟦	🟦										
2	🟦	2	🟦											
1	🟨	1	2	🟦				🟦	🟦	🟦	🟦	🟦		
2	🟦	2	🟦	6								🟦		
3	4	3	4	5	6									
4	🟦	4	5	6			🟦		🟦		🟦		🟦	🟦
5	6	5	6	🟦										
6	🟦	🟦				🟦								
	🟦	🟦			🟦	🟦					🟦			
	🟦	🟦				🟦	🟦	🟦	🟦	🟦	🟦			
	🟦	🟦									🟦			
		🟦										🟦		

End pin reached. Traceback.

Find a Wire Route - Java Code (1)

```java
/** find a shortest path from start to finish
    * @return true if successful, false if impossible */
  private static boolean findPath()
  {
     if ((start.row == finish.row) && (start.col == finish.col))
     {   // start == finish
         pathLength = 0;
         return true;
     }

     // initialize offsets
     Position [] offset = new Position [4];
     offset[0] = new Position(0, 1);   // right
     offset[1] = new Position(1, 0);   // down
     offset[2] = new Position(0, -1);  // left
     offset[3] = new Position(-1, 0);  // up
```

Find a Wire Route - Java Code (2)

```
// initialize wall of blocks around the grid
     for (int i = 0; i <= size + 1; i++)
     {
        grid[0][i] = grid[size + 1][i] = 1; // bottom and top
        grid[i][0] = grid[i][size + 1] = 1; // left and right
     }

     Position here = new Position(start.row, start.col);
     grid[start.row][start.col] = 2; // block
     int numOfNbrs = 4; // neighbors of a grid position
```

Find a Wire Route - Java Code (3)

```
// label reachable grid positions
     QueueList q = new QueueList();
     Position nbr = new Position(0, 0);
     do
     { // label neighbors of 'here'
         for (int i = 0; i < numOfNbrs; i++)
         { // check out neighbors of 'here'
            nbr.row = here.row + offset[i].row;
            nbr.col = here.col + offset[i].col;
            if (grid[nbr.row][nbr.col] == 0)
            {  // unlabeled nbr, label it
               grid[nbr.row][nbr.col]
                  = grid[here.row][here.col] + 1;
               if ((nbr.row == finish.row) &&
                   (nbr.col == finish.col)) break; // done
               // enqueue - put on queue for later expansion
              q.add(new Position(nbr.row, nbr.col));
            }
         }
```

Find a Wire Route - Java Code (4)

```
// have we reached finish?
        if ((nbr.row == finish.row) &&
            (nbr.col == finish.col)) break;     // done

        // finish not reached, can we move to a nbr?
        if ( q.isEmpty() ) return false;          // no path

        // dequeue next position
        here = (Position) q.remove();
     } while(true);


     // construct path
     pathLength = grid[finish.row][finish.col] - 2;
     path = new Position [pathLength];
```

Find a Wire Route - Java Code (5)

```
// trace backwards from finish
     here = finish;
     for (int j = pathLength - 1; j >= 0; j--)
     {
        path[j] = here;
        // find predecessor position
        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 + 2) break;
        }
        // move to predecessor
        here = new Position(nbr.row, nbr.col);
     }

     return true;
  }
```