CSE 2101: Data Structures

Lecture 08: Stacks

Md Zahidul Islam
Computer Sciece & Engineering Discipline
zahidcseku@gmail.com

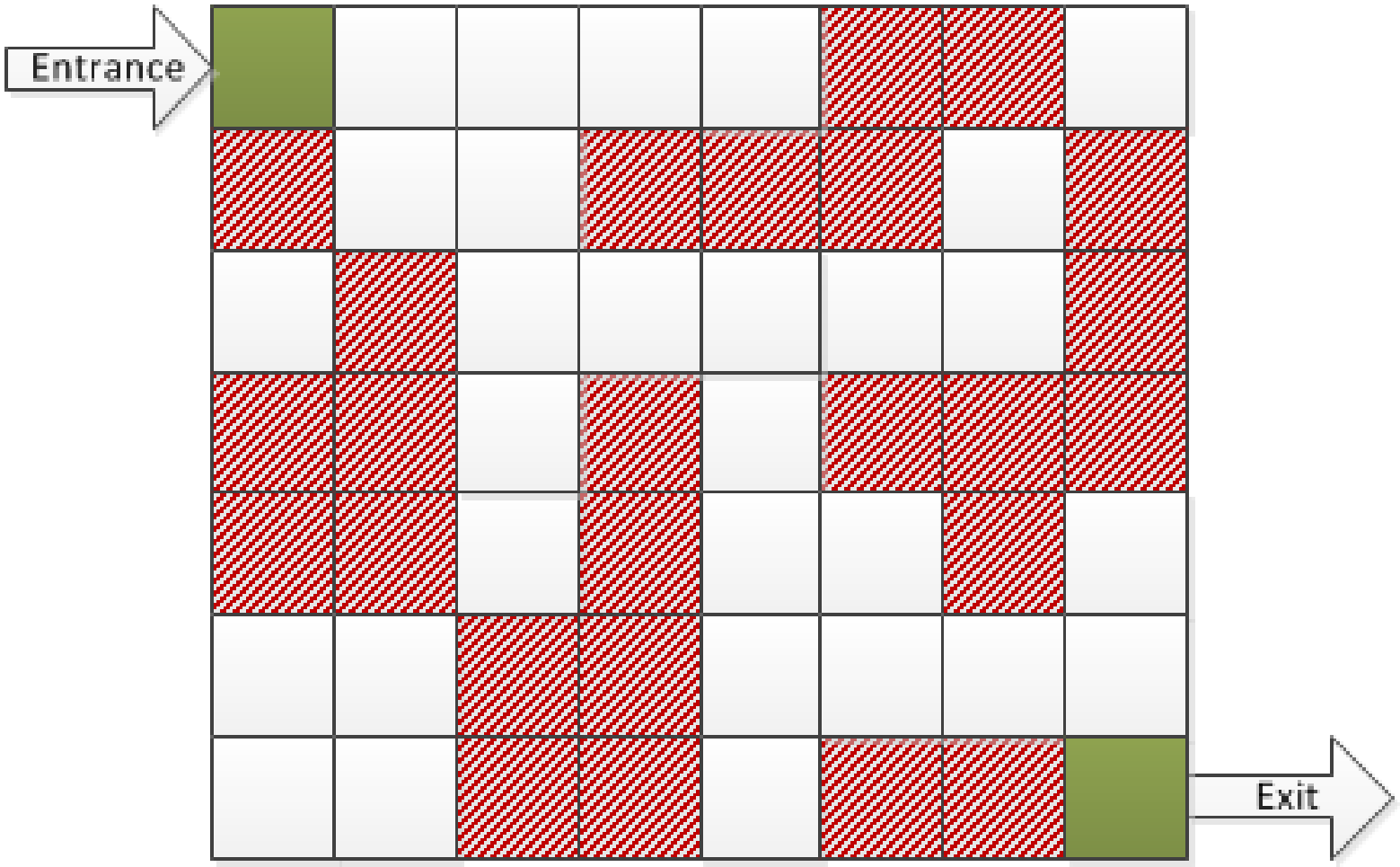
April 07, 2015

Outline

Objectives

- Learn about stacks
- Examine various stack operations
- Learn how to implement a stack as an array
- Learn how to implement a stack as a linked list
- Discover stack applications
- Learn how to use a stack for arithmatic expression evaluation

Maze Solver

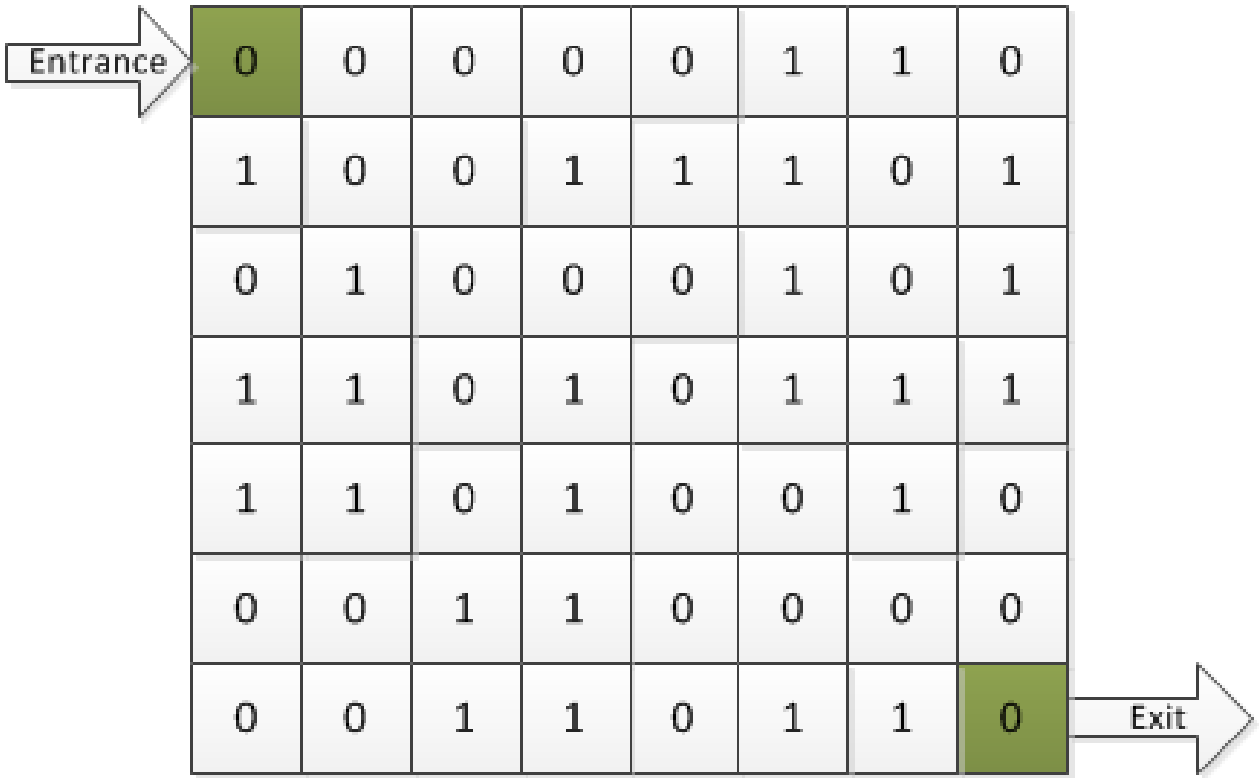


Maze Solver: Issues

- How can we represent the maze?

Maze Solver: Issues

- How can we represent the maze? Use a 2D array. Cell value 0 represents an open block, 0 represents a closed block.

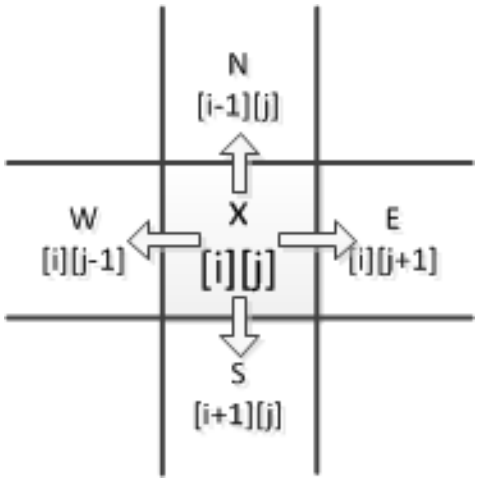


Maze Solver: Issues

- What are the allowable moves?

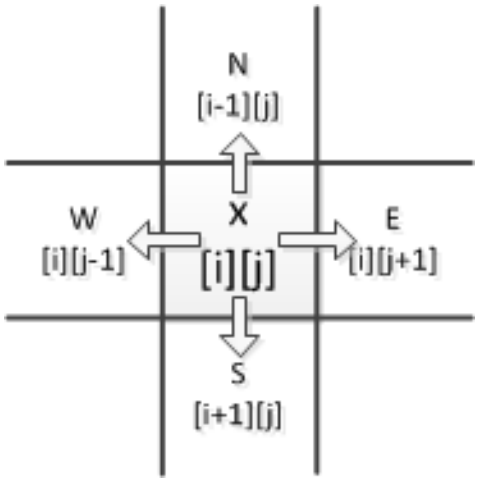
Maze Solver: Issues

- What are the allowable moves? Let us consider form a particular position $x(i, j)$ we can move to or directions (north, south, east, and west) as shown below:



Maze Solver: Issues

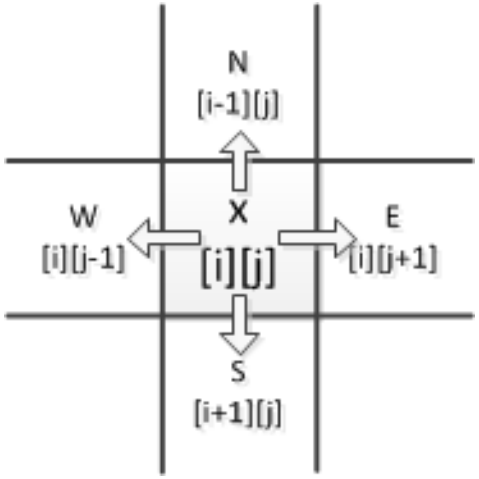
- What are the allowable moves? Let us consider form a particular position $x(i, j)$ we can move to or directions (north, south, east, and west) as shown below:



- How to generate possible moves?

Maze Solver: Issues

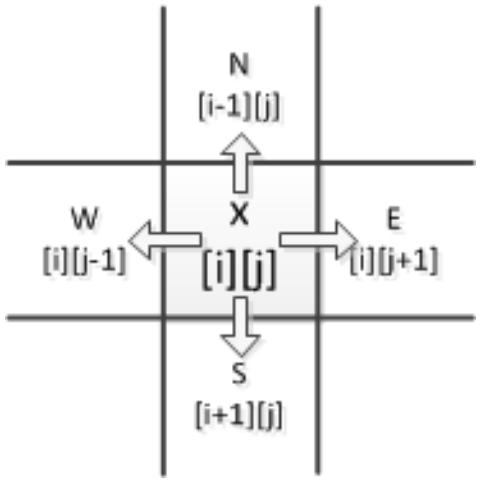
- What are the allowable moves? Let us consider form a particular position $x(i, j)$ we can move to or directions (north, south, east, and west) as shown below:



- How to generate possible moves? Not every position has four neighbours. E.g., the positions at the boundaries.

Maze Solver: Issues

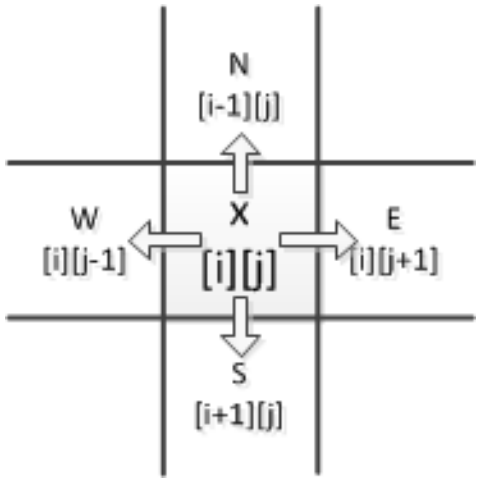
- What are the allowable moves? Let us consider form a particular position $x(i, j)$ we can move to or directions (north, south, east, and west) as shown below:



- How to generate possible moves? Not every position has four neighbours. E.g., the positions at the boundaries.
- How to get the valid moves?

Maze Solver: Issues

- What are the allowable moves? Let us consider form a particular position $x(i, j)$ we can move to or directions (north, south, east, and west) as shown below:



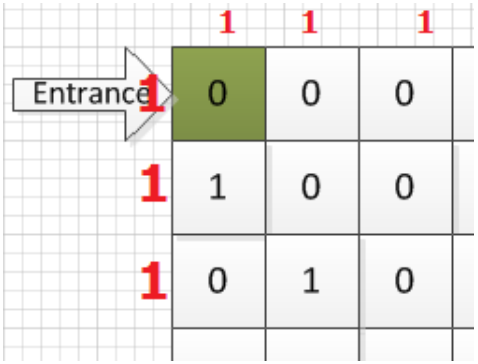
- How to generate possible moves? Not every position has four neighbours. E.g., the positions at the boundaries.
- How to get the valid moves? Check the boundary conditions and the value at the neighbouring position is not 1.

Maze Solver: Issues

- How to get the valid moves?

Maze Solver: Issues

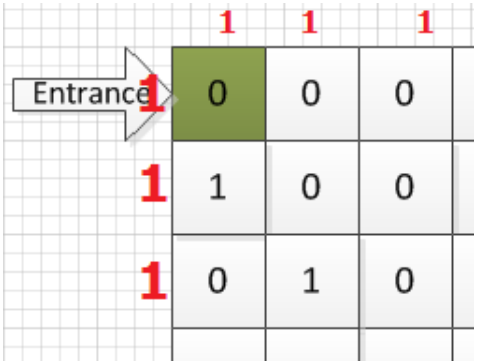
- How to get the valid moves?



We can omit the boundary condition checking by sourrunding the maze with a border of 1.

Maze Solver: Issues

- How to get the valid moves?

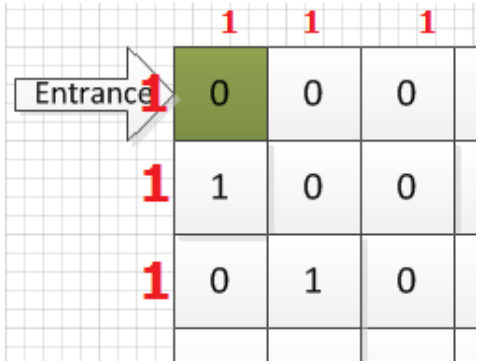


We can omit the boundary condition checking by sourrunding the maze with a border of 1.

- How do we solve it?

Maze Solver: Issues

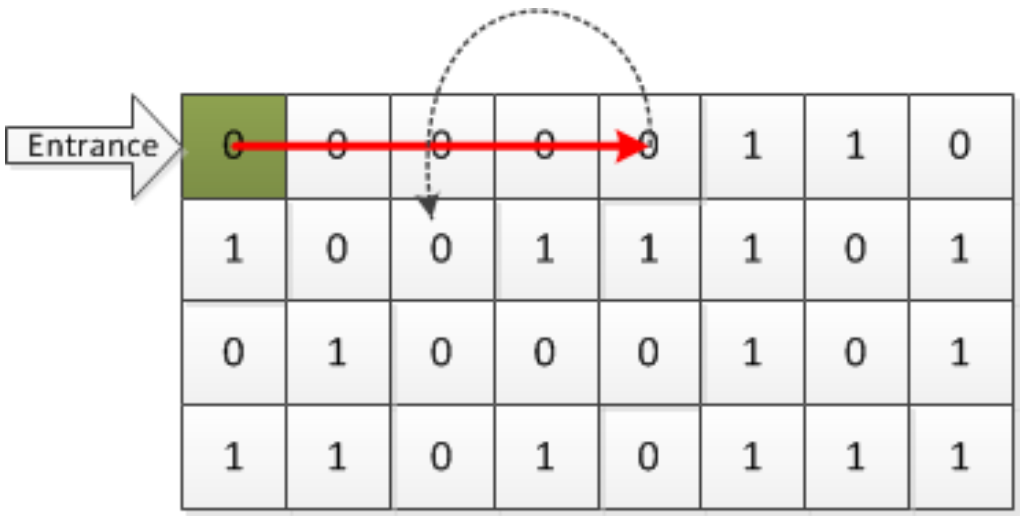
- How to get the valid moves?



We can omit the boundary condition checking by sourrunding the maze with a border of 1.

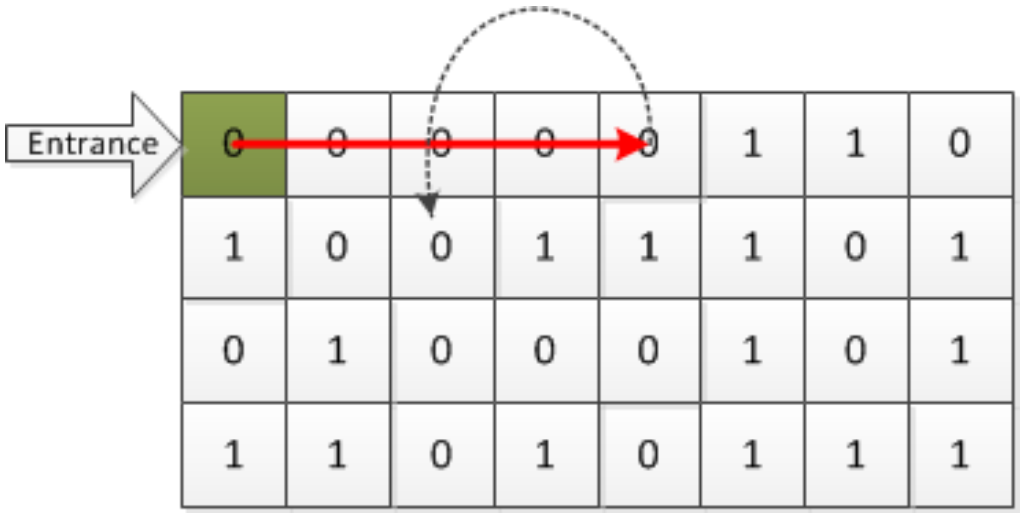
- How do we solve it? We start at location $(0, 0)$, continue to move forward until we hit a block.
If we hit on we try other direction, if there is no possible move
we go back to a previous position where there was the possible move in other direction.

Maze Solver: Issues



- If there were multiple choices in the past, how do we know which has not been explored already?

Maze Solver: Issues



- If there were multiple choices in the past, how do we know which has not been explored already?
  To prevent us from going down the same path we use another array, `mark[][]`, which is initially 0.
  `mark[i][j]` is set to 1 once wqe arrive at position (i, j).

Maze Solver: Issues

- How do we know where to return is case of a blocked move?

Maze Solver: Issues

- How do we know where to return is case of a blocked move? We must put the options somewhere so that we can return to those options once we hit a blocked cell. May be an array...

Maze Solver: Issues

- How do we know where to return is case of a blocked move? We must put the options somewhere so that we can return to those options once we hit a blocked cell. May be an array...
- To avoid re-entering the maze many times, we store the maze in a text file and store the maze from that file.

Maze Solver: Algorithm

Step: 1 Deifne a storage for maze, some required variables and moves.

```
int[][] board; //row X col
int rows, cols;
int [][] moves = {{-1,0}, {0, 1}, {1,0}, {0,-1}};
                          //N      E      S      W
```

Step: 2 Load the maze from a given file name.

```
fileIn = new Scanner (new FileReader(fileName) );
rows = fileIn.nextInt(); cols = fileIn.nextInt();
board = new int[rows+2][cols+2];
for(int i = 0; i < rows+2; i++ ) {
  for (int j = 0; j < cols+2; j++ ) {
    if(i==0||i==rows+1||j==0||j==cols+1)
      board[i][j] = 1;
    else board[i][j] = fileIn.nextInt();
  }
}//end for
```

Maze Solver: Algorithm

Step: 3 initialize the `mark[][]` array.

```
1 for(int i = 0; i < rows+2; i++ )
2   for (int j = 0; j < cols+2; j++ )
3     mark[i][j] = 0;
```

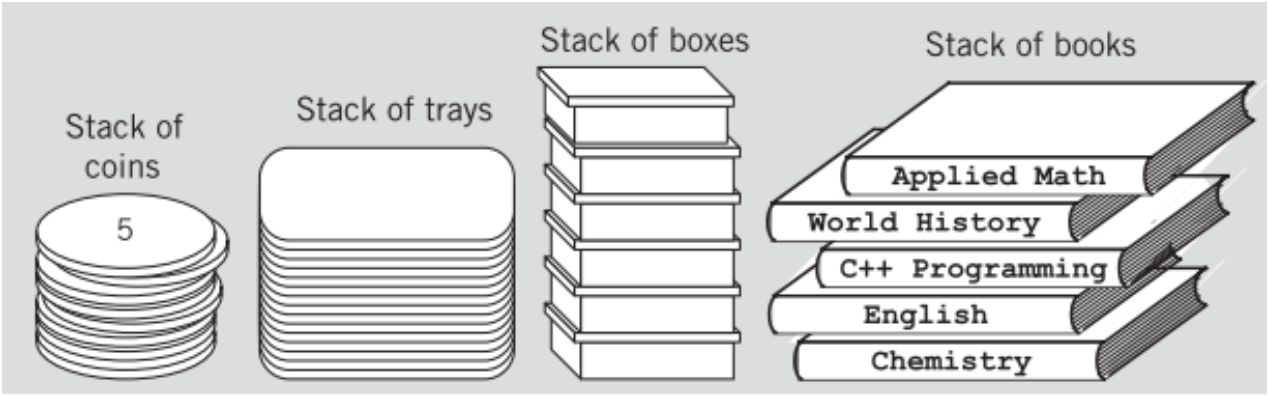
Step: 4 Until the goal is reached or there is no solution continue the following steps:

- Step a: Mark the current position.
- Step b: Generate all the possible moves from the current position.
- Step c: Save them in a temporay storage.
- Step d: Get a new position from the temporary storage for next iteration.

Maze Solver: Step 4

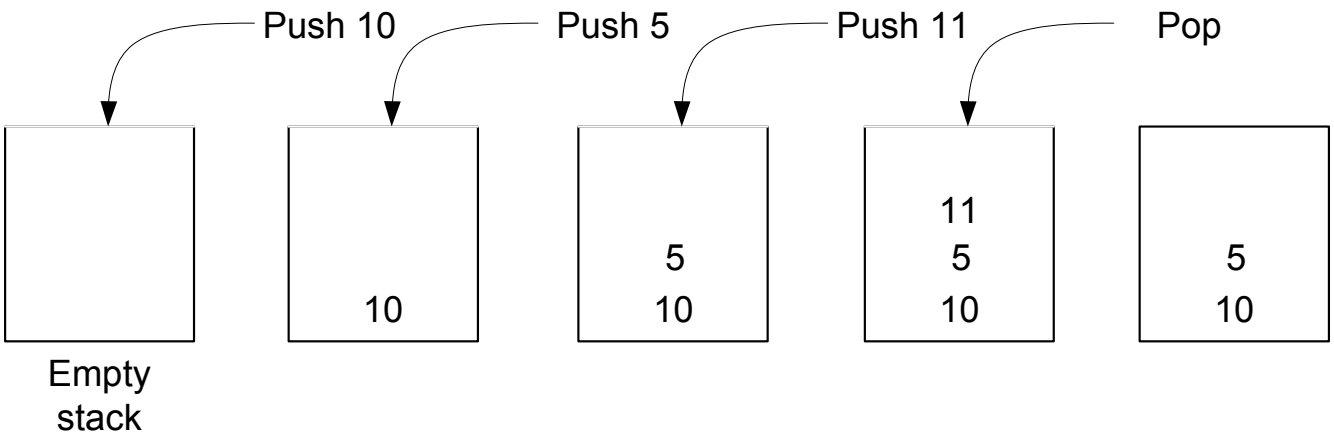
```
1  //load a new position form the storage
2  cx = temp.getX();  cy = temp.getY();
3  mark[cx][cy] = 1;   //mark it
4
5  //generate next moves and save in a temp storage
6  boolean anyValid = false;
7  for ( int i = 0; i < 4; i++ ){
8    newx = cx + moves[i][1];
9    newy = cy + moves[i][0];
10
11   if (board[newx][newy]==0 && mark[newx][newy]==0) {
12     anyValid = true; // a valid move
13     save it;
14   }
15 }//end for
```

Stack

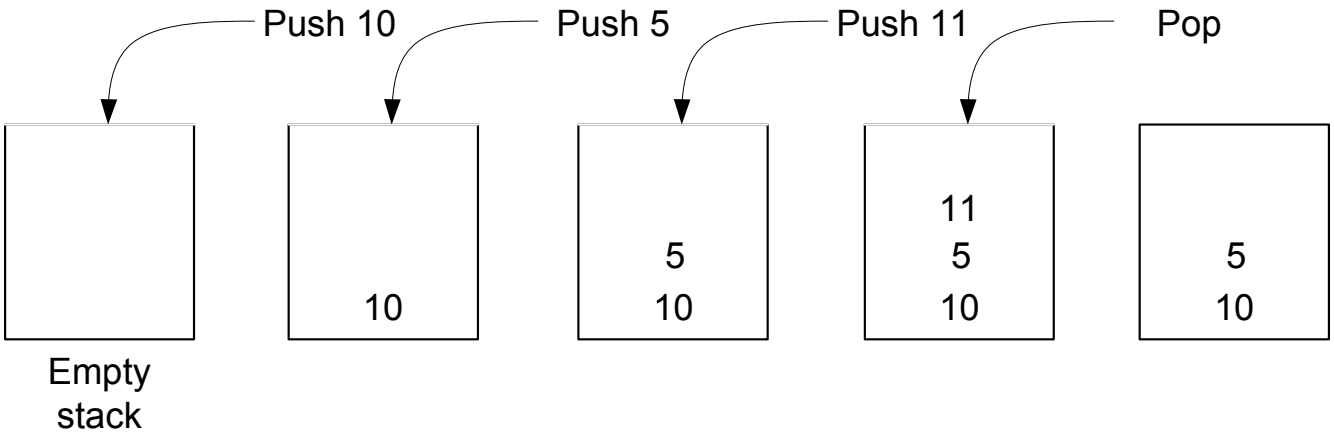


- A stack is a collection of items into which items can be inserted or deleted at one end.
- A stack implements the LIFO protocol
- The end where all the operations occurs is called *top*
- Operations on stack:
  - **Push**: Puts an item onto the stack
  - **Pop**: Removes an item from a stack
  - **Empty**: Checks whether the stack is empty or not
  - **Peek**: Returns the top element of the stack without changing the stack

STACKS

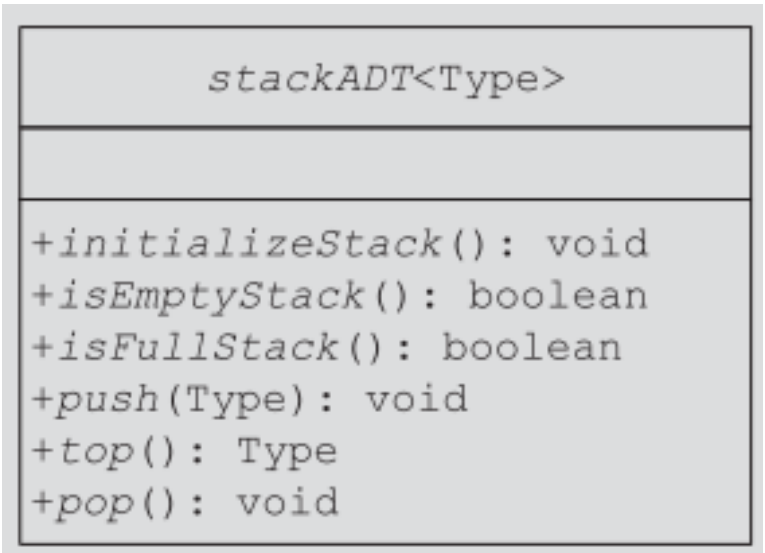


STACKS



- An empty stack allows push operations but not pop operation
- the result of an illegal attempt to pop or access an item from an empty stack is called underflow.
- Underflow can be avoided ensuring that Empty operation returns *false* before attempting the operation **Pop** or **Peek**.

The Stack ADT



Some Applications of Stacks

- Back button of your web browsers.
- Undo button of your text editors.
- Neumerous use in compiler
  - Check for syntax error in your program.
  - Cascading loops.
  - Recursion.
- Evaluating arithmatic expression.
- Many more...

Maze Solver Algorithm

Step: 4 Until the goal is reached or there is no solution continue the following steps:

- Step a: Mark the current position.
- Step b: Generate all the possible moves from the current position.
- Step c: Push them in a stack.
- Step d: If there is no valid move, pop from the stack.
- Step e: Peek a new position from the stack for next iteration.

Stack Implementation

- Stacks implementation in Java:
  1. Using arrays: must define the size of the stack during declaration. Must keep track of the size of the stack to prevent overflow.
  2. Using dynamic variables(linked lists): must make sure that all the links are appropriate
- **Overflow:** occurs if we try to insert an item into an array which is full.
- **Underflow:** occurs when we try to retrieve an item form an empty list.

Array Based Stack

- Things to represent: **stack top**, **stack elementspush(item)**, **pop()**, **peek()**

Array Based Stack

- Things to represent: **stack top, stack elementspush(item), pop(), peek()**

```
1  public class ArrayStack<E> {
2    private E[] data; // generic array used for storage
3    private int t = -1; // index of the top element
4
5    public ArrayStack(int capacity) {
6      data = (E[ ]) new Object[capacity];
7    }
8
9    public void initializeStack() { }
10   public int size() { }
11   public boolean isEmptyStack() { }
12   public boolean isFullStack() { }
13   public void push(E e) throws IllegalStateException { }
14   public E pop() { }
15   public E peek() { }
16 }
```

SIZE(), ISEMPTYSTACK() AND ISFULLSTACK()

- Arrays start at index 0 in Java.
- We initialize stack index to -1 to indicate an empty stack
- When an array has data from data[0] to data[t], it has t+1 elements.

SIZE(), ISEMPTYSTACK() AND ISFULLSTACK()

- Arrays start at index 0 in Java.
- We initialize stack index to -1 to indicate an empty stack
- When an array has data from data[0] to data[t], it has t+1 elements.

```
1 public int size() {
2   return (t+1);
3 }
```

```
1 public boolean isEmptyStack() {
2   return (t == -1);
3 }
```

```
1 public boolean isFullStack() {
2   return (t == capacity);
3 }
```

Push Operation

Steps:

1. If the stack is full, print an error message (overflow)
2. Make place for new item by incrementing top
3. Put the item in place

Push Operation

Steps:

1. If the stack is full, print an error message (overflow)
2. Make place for new item by incrementing top
3. Put the item in place

```
1 public void push(E e) throws IllegalStateException {
2   if (isFullStack()) throw new IllegalStateException(''
      Stack is full'');
3   data[++t] = e;
4 }
```

Pop Operation

Steps:

1. If the stack is empty, print an error message an halt execution (underflow)
2. Remove the top element from the stack
3. Assign null to delted position to help Java GC
4. Decrement top
5. Return the element to the calling program

Pop Operation

Steps:

1. If the stack is empty, print an error message an halt execution (underflow)
2. Remove the top element from the stack
3. Assign null to delted position to help Java GC
4. Decrement top
5. Return the element to the calling program

```
1 public E pop() throws IllegalStateException{
2   if (isEmptyStack() ) throw new IllegaStateException ('
      'Stack is empty!'');
3
4   E answer = data[t];
5   data[t] = null;
6   t--;
7   return answer;
8 }
```

Peek Operation

- Same as pop operation except that the element is not removed from the stack.

Peek Operation

- Same as pop operation except that the element is not removed from the stack.

Steps:

1. If the stack is empty, print an error message an halt execution (underflow)
2. Return the element to the calling program

Peek Operation

- Same as pop operation except that the element is not removed from the stack.

Steps:

1. If the stack is empty, print an error message an halt execution (underflow)
2. Return the element to the calling program

```
1 public E pop() throws IllegalStateException{
2   if (isEmptyStack() ) throw new IllegaStateException ('
      'Stack is empty!'');
3
4   return data[t];
5 }
```

Drawback of Array Based Implementation

- Array-based stack implementation is simple and efficient.
- Problems of array based implementation:
  - The size of the stack must be declared at compile time
  - When we create an object of array stack, memory is allocated to contain the maximum number of elements:
    - if we use fewer elements - memory is wasted
    - if we need more we get a stack overflow warning

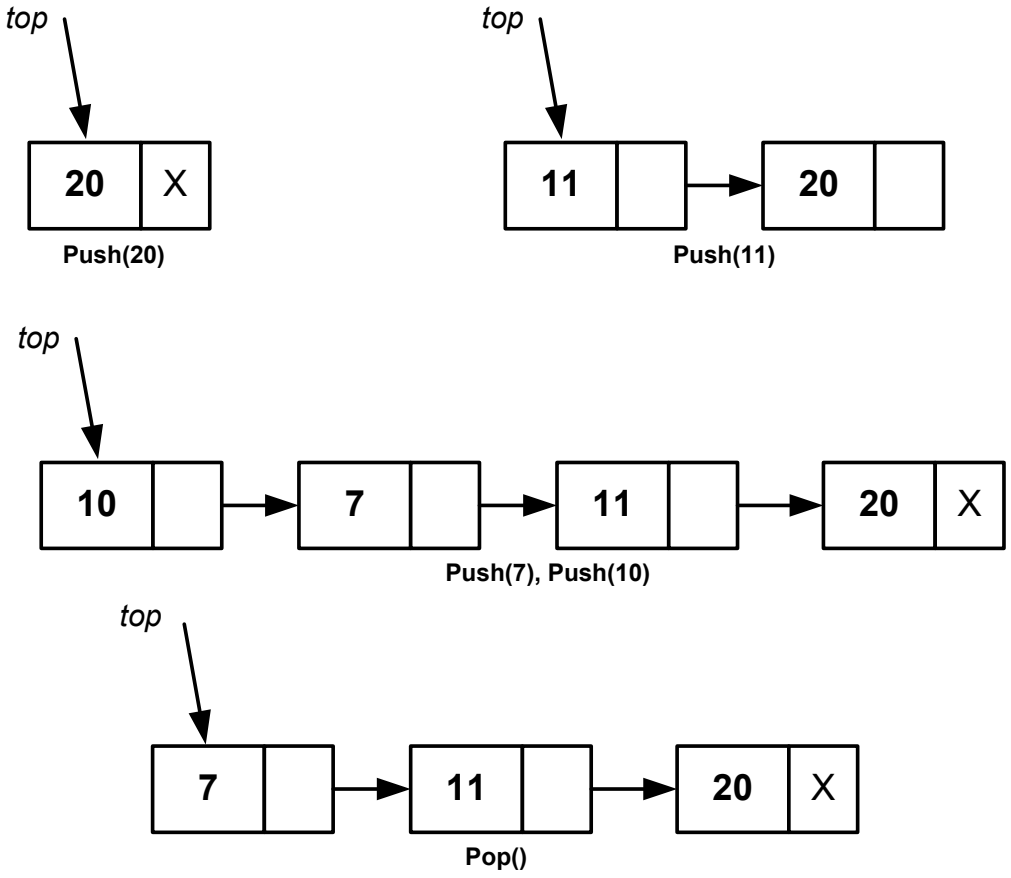
Drawback of Array Based Implementation

- Array-based stack implementation is simple and efficient.
- Problems of array based implementation:
  - The size of the stack must be declared at compile time
  - When we create an object of array stack, memory is allocated to contain the maximum number of elements:
    - if we use fewer elements - memory is wasted
    - if we need more we get a stack overflow warning
- Alternative - use linked list based implementation or expand array mechanism.

Dynamic Variable Based Stack

- The linked implementation only requires space for the number of elements actually on the stack at run time (no wastage).
- In situations where the stack size is unpredictable, the linked implementation is preferable.

Operations In a Dynamic Stack



- Can you find any similarity between this operations and linked list operations??

Checking the Validity of an Expression

1. Start to read the expression from the left, one symbol at a time.
2. Whenever a openning delimiter is encountered, it is pushed on to the stack.
3. Whenever a closing delimiter is encountered, the stack is examined
   - 3.1 If the stack is empty, then the closing delimiter does not have a matching opener and therefore the expression is invalid.
   - 3.2 If the stack is non empty, we pop the stack and check whether the popped item corresponds to the closing delimiter. If a match occur we continue. If does not, then the expression is invalid.
4. When the end of the string is reached, the stack must be empty; otherwise one or more scopes have been opened which have not been closed and the string is invalid.

Checking the Validity of an Expression

```
1  public static boolean isMatched(String expression) {
2    final String opening = "({[";  //opening delimiters
3    final String closing = ")}]";  //respective closing
4
5    Stack<Character> myStack = new LinkedStack<>();
6
7    for (char c : expression.toCharArray()) {
8      if (opening.indexOf(c) != -1)
9        myStack.push(c);
10     else if (closing.indexOf(c) != -1) {
11       if (myStack.isEmpty())
12         return false;
13       if (closing.indexOf(c) != opening.indexOf(myStack.
       pop()))
14         return false; //mismatched delimiter
15     }//end else if
16   }
17   return myStack.isEmptyStack();
18 }
```

Matching Tags in a Markup Language

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine.  The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

Matching Tags in a Markup Language

Similar to dilimiter checking.

1. Separate each `tag`. – i.e., the sub string from '<' to '>'
2. Whenever a openning `tag` is encountered (i.e., not having '/' character), push it on to the stack.
3. Whenever a closing `tag` is encountered, the stack is examined.
   - 3.1 If the stack is empty, then the closing tag does not have a matching opener, so the expression is invalid.
   - 3.2 If the stack is non empty, we pop the stack and check whether the popped `tag` corresponds to the closing `tag`. If a match occur we continue. If does not, then the HTML is invalid.
4. When all the tags are considerd, the stack must be empty; otherwise one or more openning tags have been found without closing tags, hence HTML is not valid.

Ilustration of the Example

Tag	Operation	myStack
body	push("body")	`body`

Ilustration of the Example

Tag	Operation	myStack
body	push("body")	`body`
center	push("center")	`center` `body`

Ilustration of the Example

Tag	Operation	myStack
body	push("body")	`body`
center	push("center")	`center` `body`

h1	push("h1")	`h1`
		`center`
		`body`

Ilustration of the Example

Tag	Operation	myStack
body	push("body")	`body`
center	push("center")	`center` `body`

h1	push("h1")	`h1`
		`center`
		`body`

\h1	pop()	`center` `body`

Ilustration of the Example

Tag	Operation	myStack
body	push("body")	`body`
center	push("center")	`center` `body`

h1	push("h1")	`h1`
		`center`
		`body`

\h1	pop()	`center` `body`
\center	pop()	`body`

Ilustration of the Example

Tag	Operation	myStack
body	push("body")	`body`
center	push("center")	`center` `body`

h1	push("h1")	`h1`
		`center`
		`body`

\h1	pop()	`center` `body`
\center	pop()	`body`
p	push("p")	`p` `body`

Ilustration of the Example

Tag	Operation	myStack
body	push("body")	`body`
center	push("center")	`center` `body`

h1	push("h1")	`h1`
		`center`
		`body`

\h1	pop()	`center` `body`
\center	pop()	`body`
p	push("p")	`p` `body`
\p	pop()	`body`

Ilustration of the Example

Tag	Operation	myStack
body	push("body")	`body`
center	push("center")	`center` `body`

h1	push("h1")	`h1`
		`center`
		`body`

\h1	pop()	`center` `body`
\center	pop()	`body`
p	push("p")	`p` `body`
\p	pop()	`body`
ol	push("ol")	`ol` `body`

Ilustration of the Example

Tag	Operation	myStack
body	push("body")	`body`
center	push("center")	`center` `body`

h1	push("h1")	`h1`
		`center`
		`body`

\h1	pop()	`center` `body`
\center	pop()	`body`
p	push("p")	`p` `body`
\p	pop()	`body`
ol	push("ol")	`ol` `body`

li	push("li")	`li`
		`ol`
		`body`

Ilustration of the Example

Tag	Operation	myStack
body	push("body")	body
center	push("center")	center body
h1	push("h1")	h1 center body
\h1	pop()	center body
\center	pop()	body
p	push("p")	p body
\p	pop()	body
ol	push("ol")	ol body
li	push("li")	li ol body

Tag	Operation	myStack
\li	pop()	ol body

Ilustration of the Example

Tag	Operation	myStack
body	push("body")	body
center	push("center")	center body
h1	push("h1")	h1 center body
\h1	pop()	center body
\center	pop()	body
p	push("p")	p body
\p	pop()	body
ol	push("ol")	ol body
li	push("li")	li ol body

Tag	Operation	myStack
\li	pop()	ol body
li	push("li")	li ol body

Ilustration of the Example

Tag	Operation	myStack
body	push("body")	body
center	push("center")	center body
h1	push("h1")	h1 center body
\h1	pop()	center body
\center	pop()	body
p	push("p")	p body
\p	pop()	body
ol	push("ol")	ol body
li	push("li")	li ol body
\li	pop()	ol body
li	push("li")	li ol body
\li	pop()	ol body

Ilustration of the Example

Tag	Operation	myStack
body	push("body")	body
center	push("center")	center body
h1	push("h1")	h1 center body
\h1	pop()	center body
\center	pop()	body
p	push("p")	p body
\p	pop()	body
ol	push("ol")	ol body
li	push("li")	li ol body

Tag	Operation	myStack
\li	pop()	ol body
li	push("li")	li ol body
\li	pop()	ol body
li	push("li")	li ol body

Ilustration of the Example

Tag	Operation	myStack
body	push("body")	body
center	push("center")	center body
h1	push("h1")	h1 center body
\h1	pop()	center body
\center	pop()	body
p	push("p")	p body
\p	pop()	body
ol	push("ol")	ol body
li	push("li")	li ol body

Tag	Operation	myStack
\li	pop()	ol body
li	push("li")	li ol body
\li	pop()	ol body
li	push("li")	li ol body
\li	pop()	ol body

Ilustration of the Example

Tag	Operation	myStack
body	push("body")	body
center	push("center")	center body
h1	push("h1")	h1 center body
\h1	pop()	center body
\center	pop()	body
p	push("p")	p body
\p	pop()	body
ol	push("ol")	ol body
li	push("li")	li ol body
\li	pop()	ol body
li	push("li")	li ol body
\li	pop()	ol body
\ol	pop()	body

Ilustration of the Example

Tag	Operation	myStack
body	push("body")	`body`
center	push("center")	`center` `body`

h1	push("h1")	`h1`
		`center`
		`body`

\h1	pop()	`center` `body`
\center	pop()	`body`
p	push("p")	`p` `body`
\p	pop()	`body`
ol	push("ol")	`ol` `body`

li	push("li")	`li`
		`ol`
		`body`

\li	pop()	`ol` `body`

li	push("li")	`li`
		`ol`
		`body`

\li	pop()	`ol` `body`
\ol	pop()	`body`
\body	pop()	

*At the end `isEmptyStack()` is *true*, i.e., the HTML is valid.

Thank you!!!

Solve ProblemSet05.