

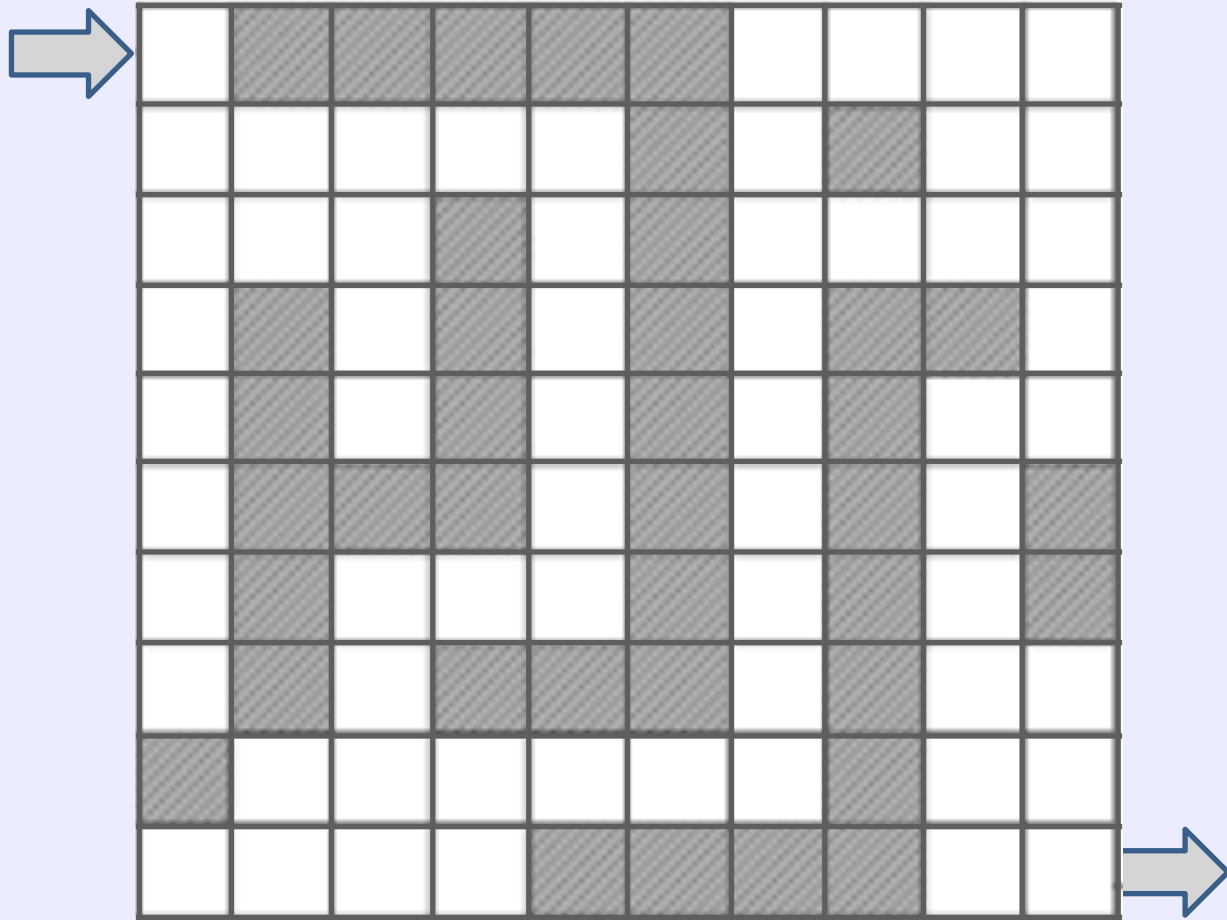
COL106: Data Structures and Algorithms

Lecture 3

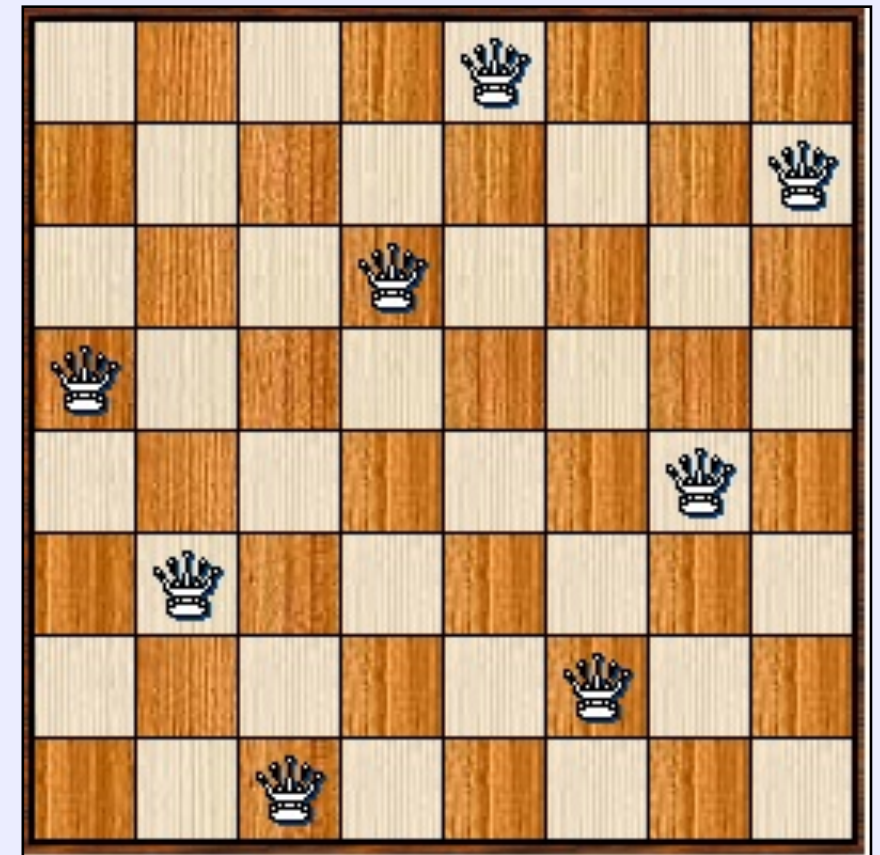
Keerti Choudhary
Department of Computer Science, IIT Delhi

Stack: A new data-structure

A powerful data structure with applications in:



Maze solving



Solving 8x8 Queen Puzzle
(Place 8 queens so that no two
of them attach each other)

$$x = 3 + 6 * (4 - 8 * (9 + 7 * 6) + 2)$$

Expression evaluation

Stack

Recall: List is a data-structure where we can insert/delete/query at any given position.

In Stack: One can insert/delete/query only at top/first index.

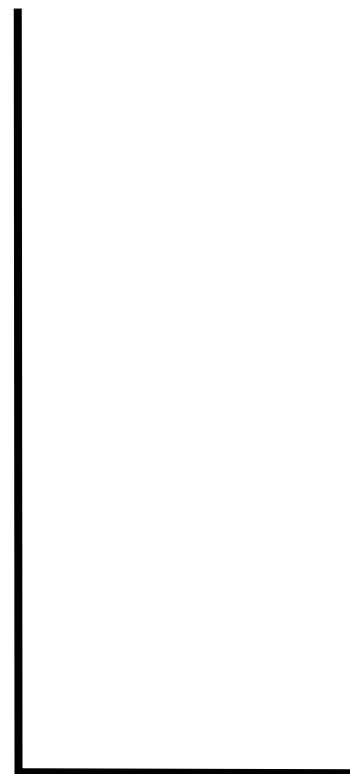
Stack

Recall: List is a data-structure where we can insert/delete/query at any given position.

In Stack: One can insert/delete/query only at top/first index.

Push: Adds element

Pop: Removes element



Example: stack

Stack

Recall: List is a data-structure where we can insert/delete/query at any given position.

In Stack: One can insert/delete/query only at top/first index.

Push: Adds element

Pop: Removes element

Push(10)

Example: stack

Stack

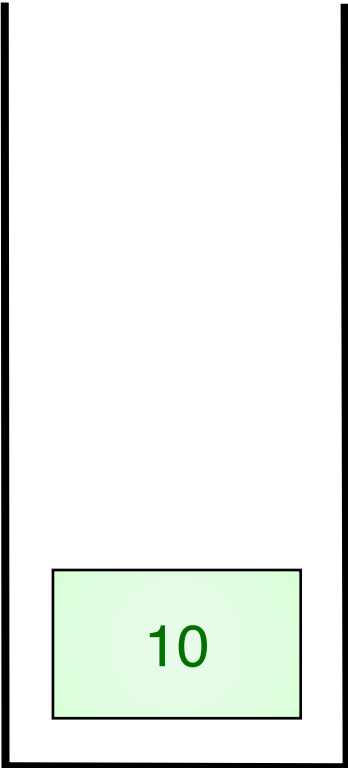
Recall: List is a data-structure where we can insert/delete/query at any given position.

In Stack: One can insert/delete/query only at top/first index.

Push: Adds element

Pop: Removes element

Push(10)

A diagram of a stack. It consists of a tall, narrow rectangle with a black border. Inside this rectangle, at the bottom, is a smaller green rectangle with a black border. The number '10' is written in green text inside the green rectangle.

10

Example: stack

Stack

Recall: List is a data-structure where we can insert/delete/query at any given position.

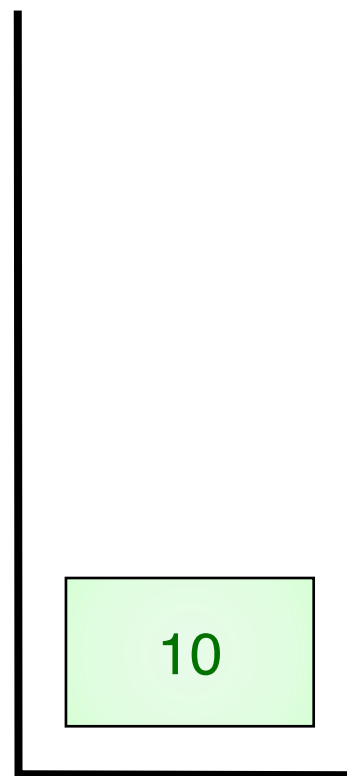
In Stack: One can insert/delete/query only at top/first index.

Push: Adds element

Pop: Removes element

Push(10)

Push(20)



Example: stack

Stack

Recall: List is a data-structure where we can insert/delete/query at any given position.

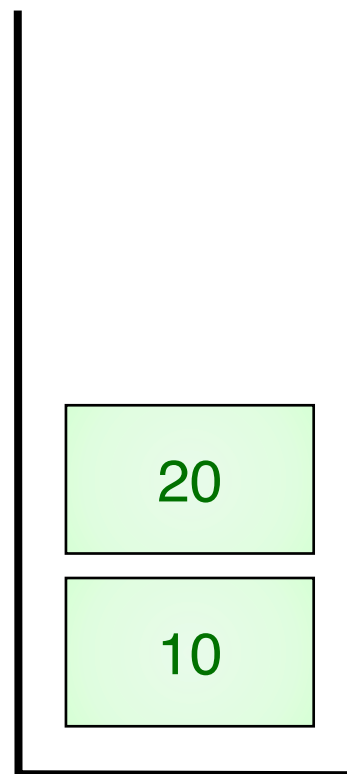
In Stack: One can insert/delete/query only at top/first index.

Push: Adds element

Pop: Removes element

Push(10)

Push(20)



Example: stack

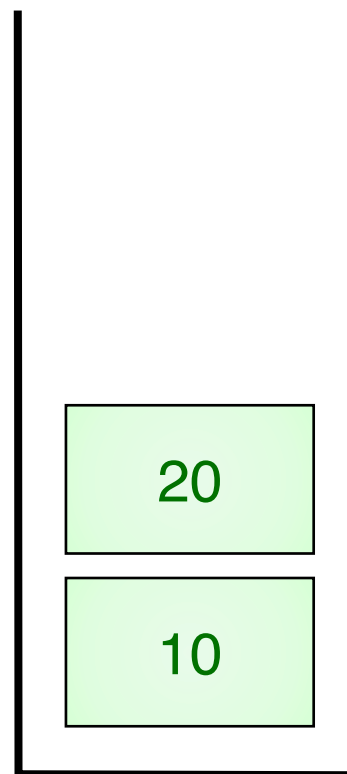
Stack

Recall: List is a data-structure where we can insert/delete/query at any given position.

In Stack: One can insert/delete/query only at top/first index.

Push: Adds element

Pop: Removes element



Push(10)

Push(20)

Push(30)

Example: stack

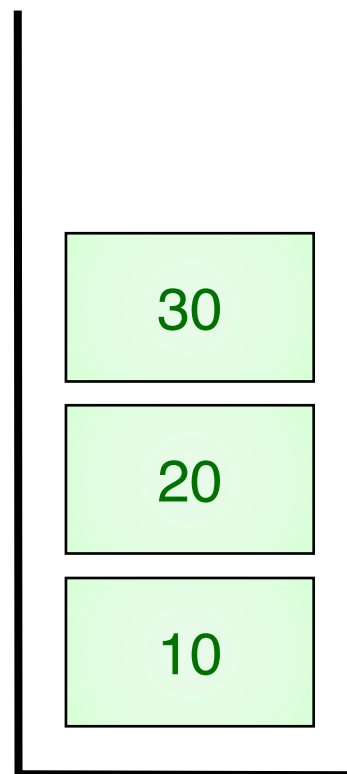
Stack

Recall: List is a data-structure where we can insert/delete/query at any given position.

In Stack: One can insert/delete/query only at top/first index.

Push: Adds element

Pop: Removes element



Push(10)

Push(20)

Push(30)

Example: stack

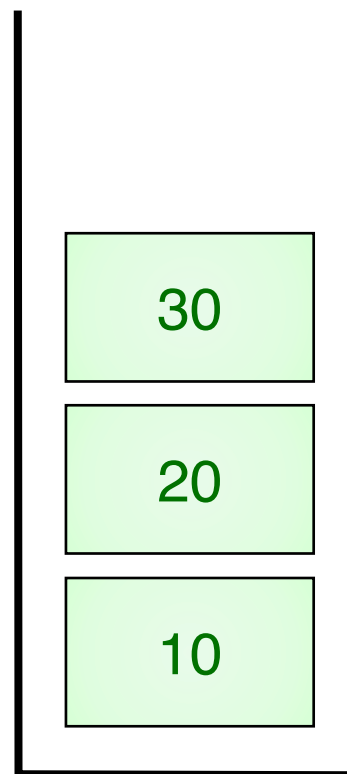
Stack

Recall: List is a data-structure where we can insert/delete/query at any given position.

In Stack: One can insert/delete/query only at top/first index.

Push: Adds element

Pop: Removes element



Push(10)

Push(20)

Push(30)

Pop()

Example: stack

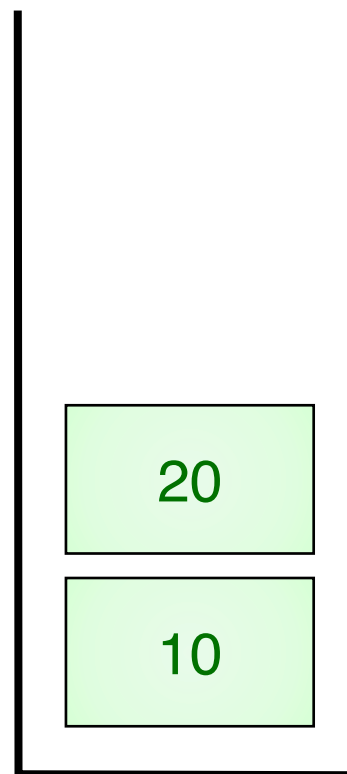
Stack

Recall: List is a data-structure where we can insert/delete/query at any given position.

In Stack: One can insert/delete/query only at top/first index.

Push: Adds element

Pop: Removes element



Push(10)

Push(20)

Push(30)

Pop()

Example: stack

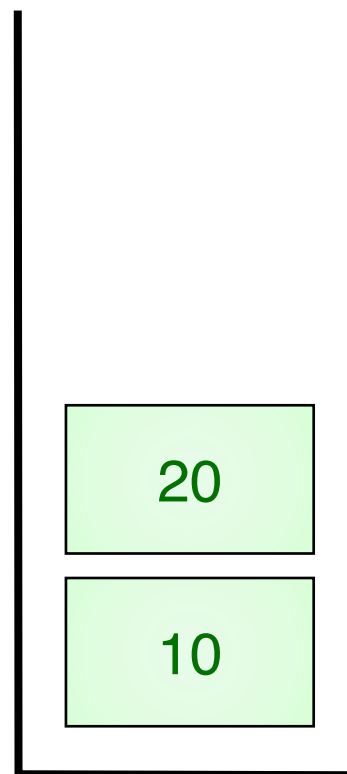
Stack

Recall: List is a data-structure where we can insert/delete/query at any given position.

In Stack: One can insert/delete/query only at top/first index.

Push: Adds element

Pop: Removes element



Push(10)

Push(20)

Push(30)

Pop()

Push(40)

Example: stack

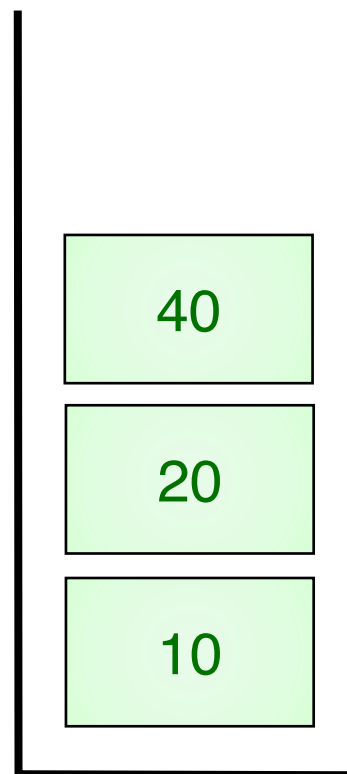
Stack

Recall: List is a data-structure where we can insert/delete/query at any given position.

In Stack: One can insert/delete/query only at top/first index.

Push: Adds element

Pop: Removes element



Push(10)

Push(20)

Push(30)

Pop()

Push(40)

Example: stack

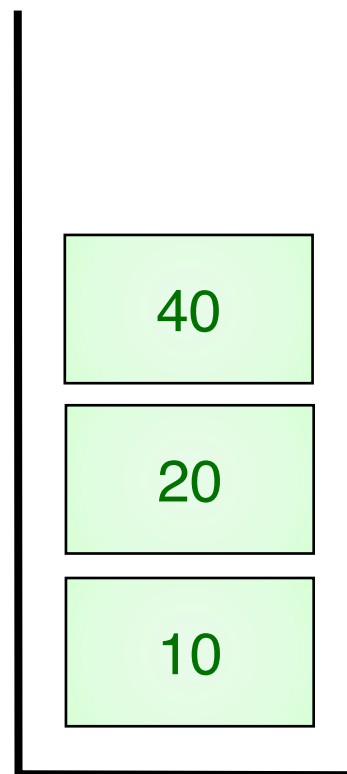
Stack

Recall: List is a data-structure where we can insert/delete/query at any given position.

In Stack: One can insert/delete/query only at top/first index.

Push: Adds element

Pop: Removes element



Push(10)

Push(20)

Push(30)

Pop()

Push(40)

Push(50)

Example: stack

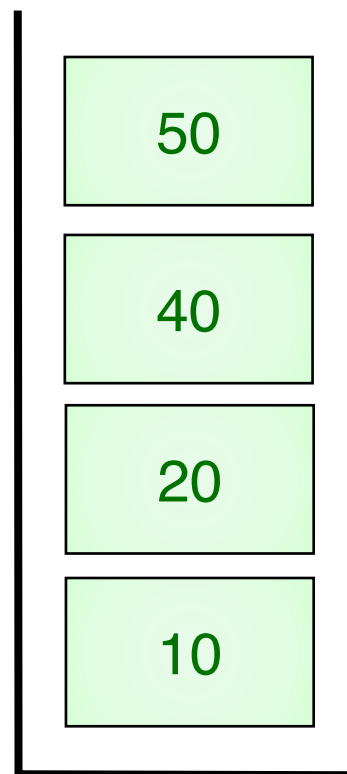
Stack

Recall: List is a data-structure where we can insert/delete/query at any given position.

In Stack: One can insert/delete/query only at top/first index.

Push: Adds element

Pop: Removes element



Push(10)

Push(20)

Push(30)

Pop()

Push(40)

Push(50)

Example: stack

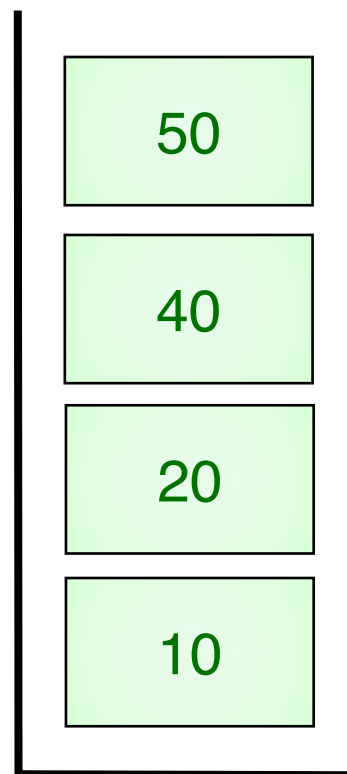
Stack

Recall: List is a data-structure where we can insert/delete/query at any given position.

In Stack: One can insert/delete/query only at top/first index.

Push: Adds element

Pop: Removes element



Push(10)

Push(20)

Push(30)

Pop()

Push(40)

Push(50)

Pop()

Example: stack

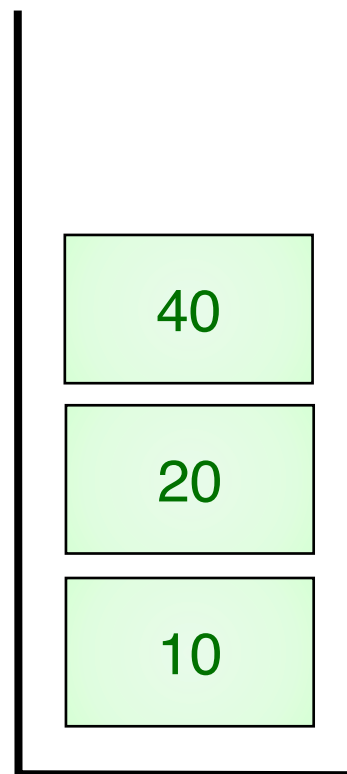
Stack

Recall: List is a data-structure where we can insert/delete/query at any given position.

In Stack: One can insert/delete/query only at top/first index.

Push: Adds element

Pop: Removes element



Example: stack

Push(10)

Push(20)

Push(30)

Pop()

Push(40)

Push(50)

Pop()

Operations on Stack

Query Operations

- **Top(S)**: returns the element at top of stack S.
Example: If $S = (a_1, a_2, \dots, a_n)$, then **Top(S)** return a_1 .
- **IsEmpty(S)**: determine if S is an empty stack.

Update Operations

- **Push(x, S)**: insert x at top of stack S.
Example: If $S = (a_1, \dots, a_n)$, then after **Push(x, S)** stack is $(x, a_1, \dots, a_n)$.
- **Pop(S)**: delete top element from stack S.
Example: If $S = (a_1, a_2, \dots, a_n)$, then after **Pop(S)** stack is $(a_2, \dots, a_n)$.

Application: Balanced Parentheses

Input: Sequence containing characters '(', ')', '{', '}', '[' and ']

Goal: Determine if the Parentheses are balanced.

Eg 1. ({ [] }) [] is balanced

Eg 2. ({ [] }) [] is unbalanced

Application: Balanced Parentheses

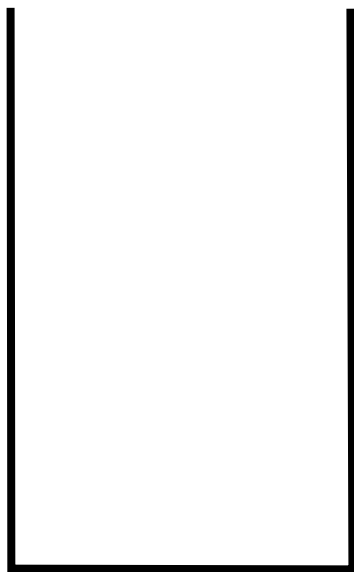
Input: Sequence containing characters '(', ')', '{', '}', '[' and ']

Goal: Determine if the Parentheses are balanced.

Eg 1. ({ [] }) [] is balanced

Eg 2. ({ [] }) [] is unbalanced

Eg 1. ({ [] }) []



Algorithm using stack

Application: Balanced Parentheses

Input: Sequence containing characters '(', ')', '{', '}', '[' and ']

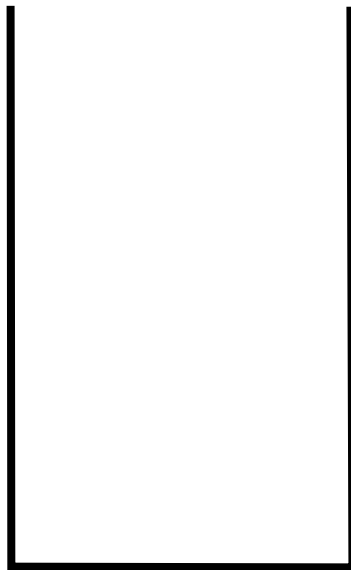
Goal: Determine if the Parentheses are balanced.

Eg 1. ({ [] }) [] is balanced

Eg 2. ({ [] }) [] is unbalanced

Eg 1. ({ [] }) []

Push('(')



Algorithm using stack

Application: Balanced Parentheses

Input: Sequence containing characters '(', ')', '{', '}', '[' and ']

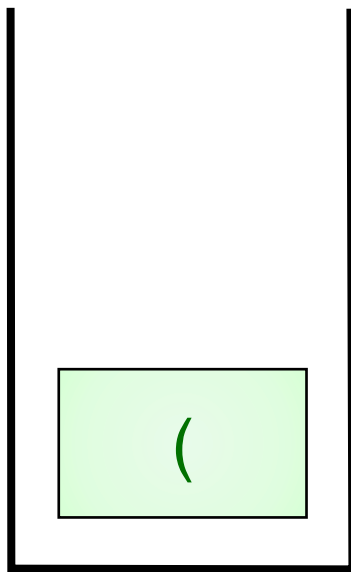
Goal: Determine if the Parentheses are balanced.

Eg 1. ({ [] }) [] is balanced

Eg 2. ({ [] }) [] is unbalanced

Eg 1. ({ [] }) []

Push('(')



Algorithm using stack

Application: Balanced Parentheses

Input: Sequence containing characters '(', ')', '{', '}', '[' and ']

Goal: Determine if the Parentheses are balanced.

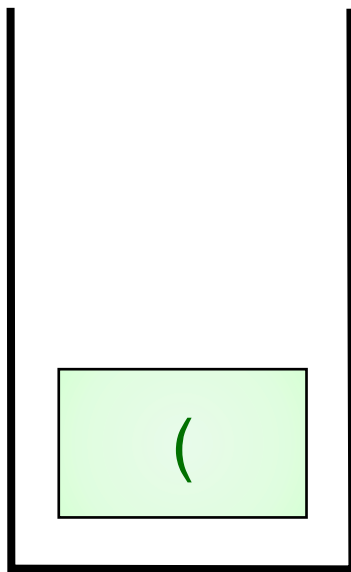
Eg 1. ({ [] }) [] is balanced

Eg 2. ({ [] }) [] is unbalanced

Eg 1. ({ [] }) []

Push('(')

Push('{')



Algorithm using stack

Application: Balanced Parentheses

Input: Sequence containing characters '(', ')', '{', '}', '[' and ']

Goal: Determine if the Parentheses are balanced.

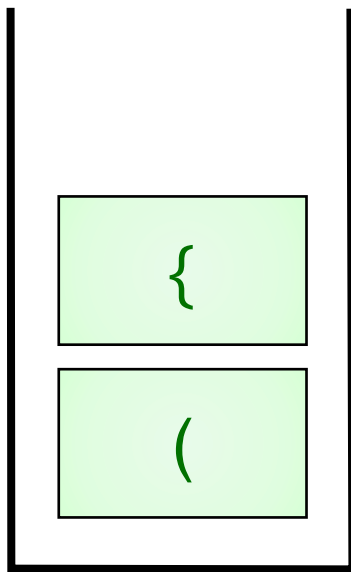
Eg 1. ({ [] }) [] is balanced

Eg 2. ({ [] }) [] is unbalanced

Eg 1. ({ [] }) []

Push('(')

Push('{')



Algorithm using stack

Application: Balanced Parentheses

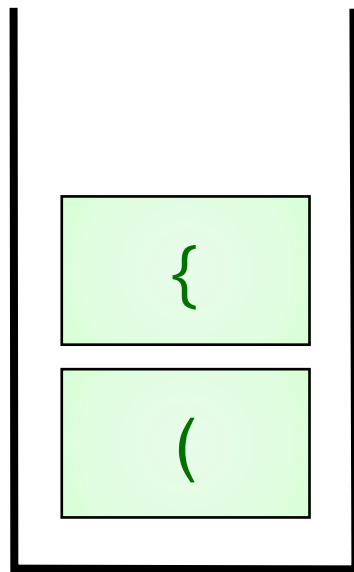
Input: Sequence containing characters '(', ')', '{', '}', '[' and ']

Goal: Determine if the Parentheses are balanced.

Eg 1. ({ [] }) [] is balanced

Eg 2. ({ [] }) [] is unbalanced

Eg 1. ({ [] }) []



Push('(')

Push('{')

Push('[')

Algorithm using stack

Application: Balanced Parentheses

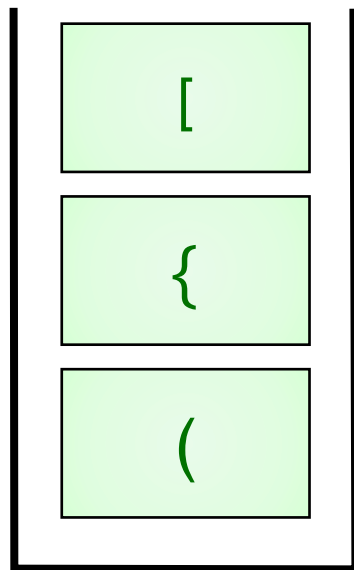
Input: Sequence containing characters '(', ')', '{', '}', '[' and ']

Goal: Determine if the Parentheses are balanced.

Eg 1. ({ [] }) [] is balanced

Eg 2. ({ [] }) [] is unbalanced

Eg 1. ({ [] }) []



Push('(')

Push('{')

Push('[')

Algorithm using stack

Application: Balanced Parentheses

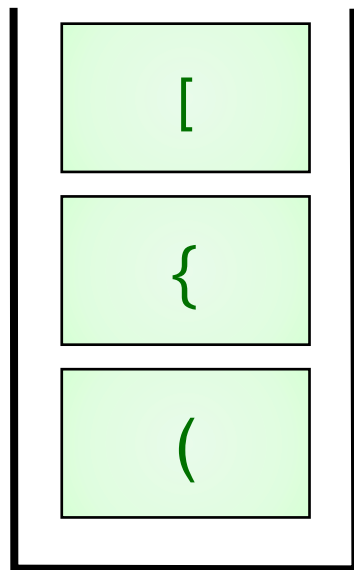
Input: Sequence containing characters '(', ')', '{', '}', '[' and ']

Goal: Determine if the Parentheses are balanced.

Eg 1. ({ [] }) [] is balanced

Eg 2. ({ [] }) [] is unbalanced

Eg 1. ({ [] }) []



Push('(')

Push('{')

Push('[')

if Top() == '[' then Pop() else print(error)

Algorithm using stack

Application: Balanced Parentheses

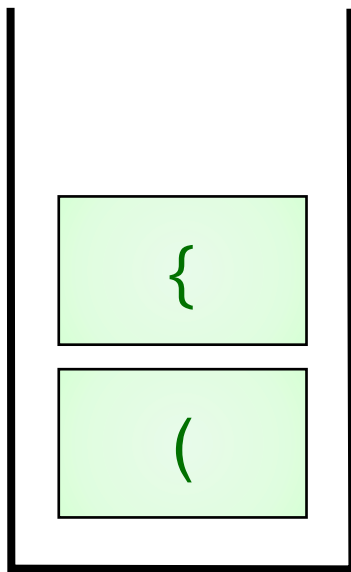
Input: Sequence containing characters '(', ')', '{', '}', '[', and ']'

Goal: Determine if the Parentheses are balanced.

Eg 1. ({ [] }) [] is balanced

Eg 2. ({ [] }) [] is unbalanced

Eg 1. ({ [] }) []



Push('(')

Push('{')

Push('[')

if Top() == '[' then Pop() else print(error)

Algorithm using stack

Application: Balanced Parentheses

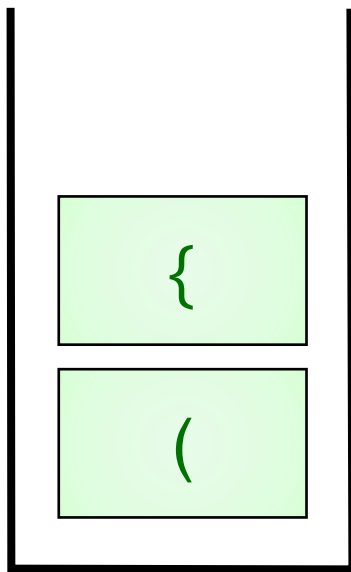
Input: Sequence containing characters '(', ')', '{', '}', '[' and ']

Goal: Determine if the Parentheses are balanced.

Eg 1. ({ [] }) [] is balanced

Eg 2. ({ [] }) [] is unbalanced

Eg 1. ({ [] }) []



Algorithm using stack

Push('(')

Push('{')

Push('[')

if Top() == '[' then Pop() else print(error)

if Top() == '{' then Pop() else print(error)

Application: Balanced Parentheses

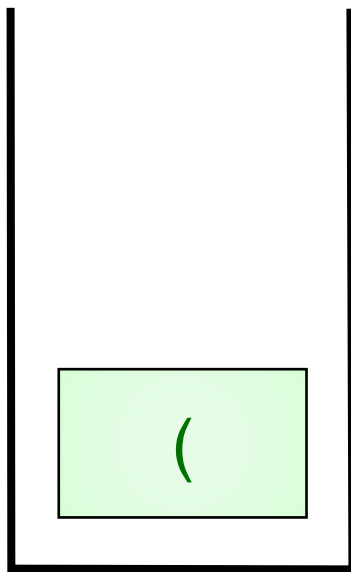
Input: Sequence containing characters '(', ')', '{', '}', '[' and ']'

Goal: Determine if the Parentheses are balanced.

Eg 1. ({ [] }) [] is balanced

Eg 2. ({ [] }) [] is unbalanced

Eg 1. ({ [] }) []



Algorithm using stack

Push('(')

Push('{')

Push('[')

if Top() == '[' then Pop() else print(error)

if Top() == '{' then Pop() else print(error)

Application: Balanced Parentheses

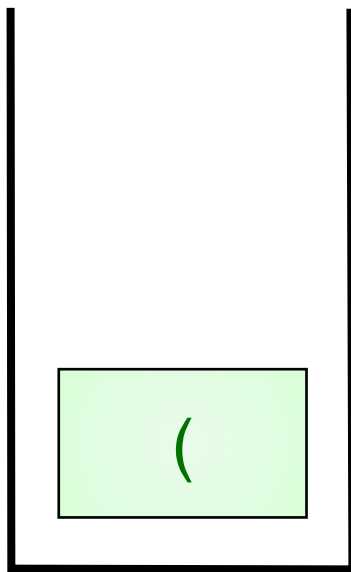
Input: Sequence containing characters '(', ')', '{', '}', '[' and ']

Goal: Determine if the Parentheses are balanced.

Eg 1. ({ [] }) [] is balanced

Eg 2. ({ [] }) [] is unbalanced

Eg 1. ({ [] }) []



Algorithm using stack

Push('(')

Push('{')

Push('[')

if Top() == '[' then Pop() else print(error)

if Top() == '{' then Pop() else print(error)

if Top() == '(' then Pop() else print(error)

Application: Balanced Parentheses

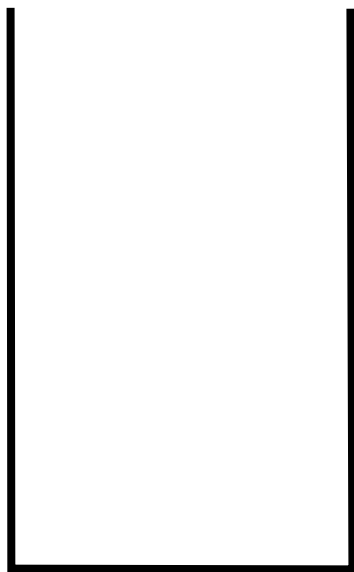
Input: Sequence containing characters '(', ')', '{', '}', '[' and ']'

Goal: Determine if the Parentheses are balanced.

Eg 1. ({ [] }) [] is balanced

Eg 2. ({ [] }) [] is unbalanced

Eg 1. ({ [] }) []



Algorithm using stack

Push('(')

Push('{')

Push('[')

if Top() == '[' then Pop() else print(error)

if Top() == '{' then Pop() else print(error)

if Top() == '(' then Pop() else print(error)

Application: Balanced Parentheses

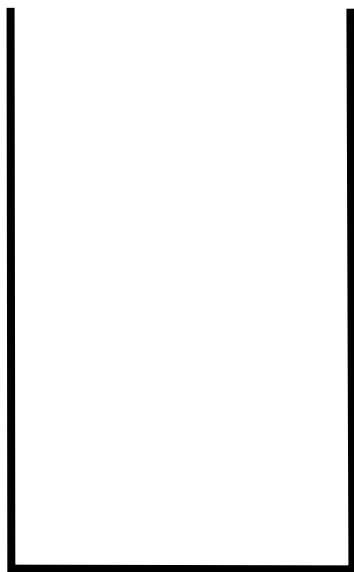
Input: Sequence containing characters '(', ')', '{', '}', '[', and ']'

Goal: Determine if the Parentheses are balanced.

Eg 1. ({ [] }) [] is balanced

Eg 2. ({ [] }) [] is unbalanced

Eg 1. ({ [] }) []



Algorithm using stack

Push('(')

Push('{')

Push('[')

if Top() == '[' then Pop() else print(error)

if Top() == '{' then Pop() else print(error)

if Top() == '(' then Pop() else print(error)

Push('[')

Application: Balanced Parentheses

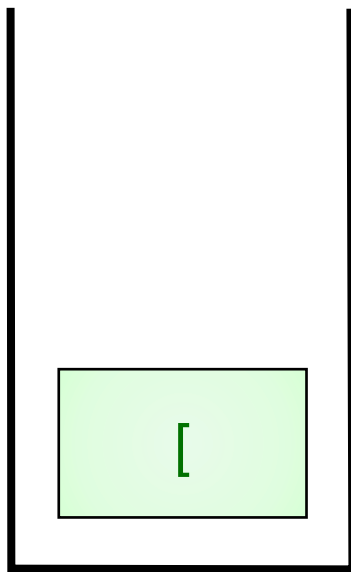
Input: Sequence containing characters '(', ')', '{', '}', '[', and ']'

Goal: Determine if the Parentheses are balanced.

Eg 1. ({ [] }) [] is balanced

Eg 2. ({ [] }) [] is unbalanced

Eg 1. ({ [] }) []



Algorithm using stack

Push('(')

Push('{')

Push('[')

if Top() == '[' then Pop() else print(error)

if Top() == '{' then Pop() else print(error)

if Top() == '(' then Pop() else print(error)

Push('[')

Application: Balanced Parentheses

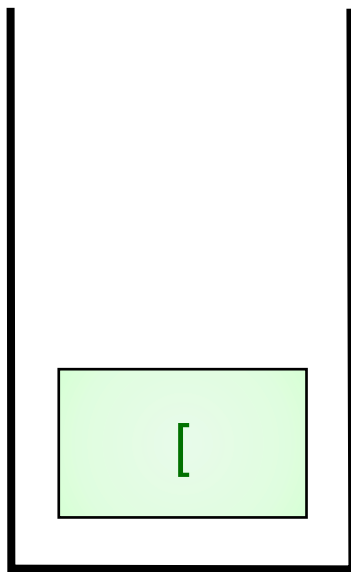
Input: Sequence containing characters '(', ')', '{', '}', '[' and ']

Goal: Determine if the Parentheses are balanced.

Eg 1. ({ [] }) [] is balanced

Eg 2. ({ [] }) [] is unbalanced

Eg 1. ({ [] }) []



Algorithm using stack

Push('(')

Push('{')

Push('[')

if Top() == '[' then Pop() else print(error)

if Top() == '{' then Pop() else print(error)

if Top() == '(' then Pop() else print(error)

Push('[')

if Top() == '[' then Pop() else print(error)

Application: Balanced Parentheses

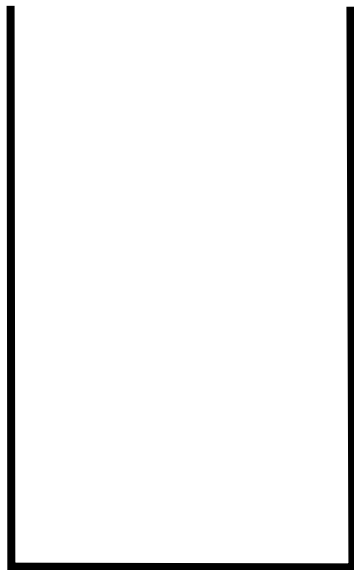
Input: Sequence containing characters '(', ')', '{', '}', '[', and ']'

Goal: Determine if the Parentheses are balanced.

Eg 1. ({ [] }) [] is balanced

Eg 2. ({ [] }) [] is unbalanced

Eg 1. ({ [] }) []



Algorithm using stack

Push('(')

Push('{')

Push('[')

if Top() == '[' then Pop() else print(error)

if Top() == '{' then Pop() else print(error)

if Top() == '(' then Pop() else print(error)

Push('[')

if Top() == '[' then Pop() else print(error)

Balanced Parentheses

Input: An array X of n tokens, each of which is a grouping symbol.

Goal: Output **true** if and only if all the grouping symbols in X match

Let S be an empty stack

for $i = 0$ to $n - 1$ **do**

if $X[i]$ is an opening symbol **then**

$S.push(X[i])$

else if $X[i]$ is a closing symbol **then**

if $S.isEmpty()$ **then**

return false {nothing to match with}

if $S.pop()$ does not match the type of $X[i]$ **then**

return false {wrong type}

if $S.isEmpty()$ **then**

return true {every symbol matched}

else

return false {some symbols were never matched}

Implementing Stack

```
public class Stack{
```

```
    class Node
```

```
    {
```

```
        int data;
```

```
        Node next;
```

```
        public Node(int data)
```

```
        {
```

```
            this.data = data;
```

```
            this.next = null;
```

```
        }
```

```
    }
```

```
    public Node head = null;
```

```
    public void push(int data)
```

```
    {
```

```
        Node a = new Node(data);
```

```
        if(head == null) { head = a; }
```

```
        else
```

```
        {
```

```
            a.next = head;
```

```
            head = a;
```

```
        }
```

```
    }
```

```
    public int top()
```

```
    {
```

```
        return head.data;
```

```
    }
```

```
    public int pop()
```

```
    {
```

```
        int value = head.data;
```

```
        head = head.next;
```

```
        return value;
```

```
    }
```

```
}
```

```
Stack S = new Stack();
```

```
for( int i=1; i<=5; i++) S.push(i);
```

```
for( int i=1; i<=5; i++)
```

```
    System.out.println(S.pop());
```

```
/* Output is : 5 4 3 2 1 */
```

Implementing Stack using Arrays

```
public class Stack{
```

Array size → private int **N**=10;
private int[] X=new int[N];

Stack size → private int **size** = 0;

```
public void push(int data)
{
    X[size]=data;
    size=size+1;
}
```

```
public int top()
{
    return X[size-1];
}
```

```
public int pop()
{
    int value = X[size-1];
    size = size-1;
    return value;
}
```

```
}
```

```
Stack S = new Stack();
for( int i=1; i<=5; i++) S.push( 10+i );
```

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14					

Integer array **X** of size **N=10**
↑
size=5

Limitation: There is a bound on the size of stack.

Stack from Dynamic Arrays

```
public void push(int data)
{
    if(size == N)
    {
        int[ ] temp = X;
        N = 2*N;
        X = new int[N];
        for(int i=0; i< size; i++)
            X[i]=temp[i];
    }

    X[size]=data;
    size=size+1;
}
```

Doubling approach

Stack from Dynamic Arrays

```
public void push(int data)
{
    if(size == N)
    {
        int[ ] temp = X;
        N = 2*N;
        X = new int[N];
        for(int i=0; i< size; i++)
            X[i]=temp[i];
    }

    X[size]=data;
    size=size+1;
}
```

Doubling approach

Claim: The amortized (average) time-complexity of “push” operations is $O(1)$.

Proof:

Suppose N takes the values $k, 2k, 4k, 8k, \dots, 2^\alpha k$

Then the total time to copy elements of array is:

$$k + 2k + 4k + 8k + \dots + 2^\alpha k = O(2^\alpha k)$$

And the number of insert operations is $\geq 2^{\alpha-1}k$.

Thus, amortized time-complexity is $O(1)$.

Stack from Dynamic Arrays

```
public void push(int data)
{
    if(size == N)
    {
        int[ ] temp = X;
        N = 2*N;
        X = new int[N];
        for(int i=0; i< size; i++)
            X[i]=temp[i];
    }

    X[size]=data;
    size=size+1;
}
```

Doubling approach

```
public int pop()
{
    int value = X[size-1];
    size = size-1;

    if(size < N/4 && N>40)
    {
        int[ ] temp = X;
        N = N/2;
        X = new int[N];
        for(int i=0; i< size; i++)
            X[i]=temp[i];
    }

    return value;
}
```

Halving approach

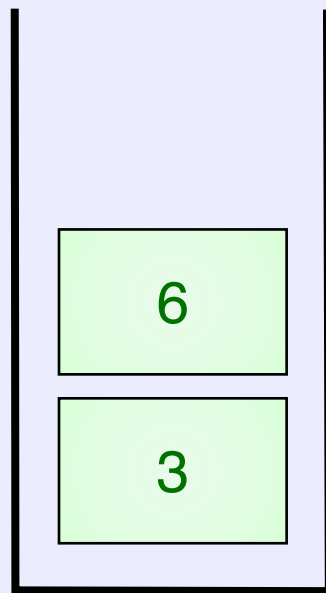
Homework: The amortized (average) time-complexity of “push” and “pop” operations is $O(1)$.

Homework

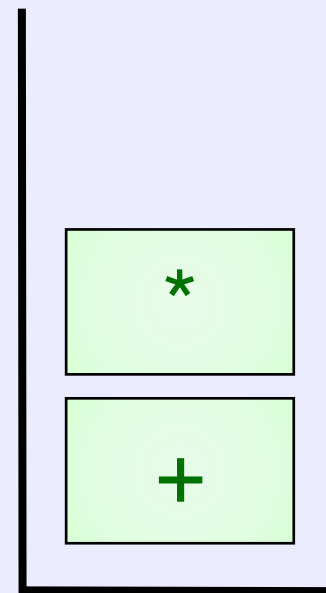
Use stack to design an algorithm for expression evaluation

$$x = 3 + 6 * 4 - 8 * 9 + 72 / 6 + 2$$

Hint:



Number stack

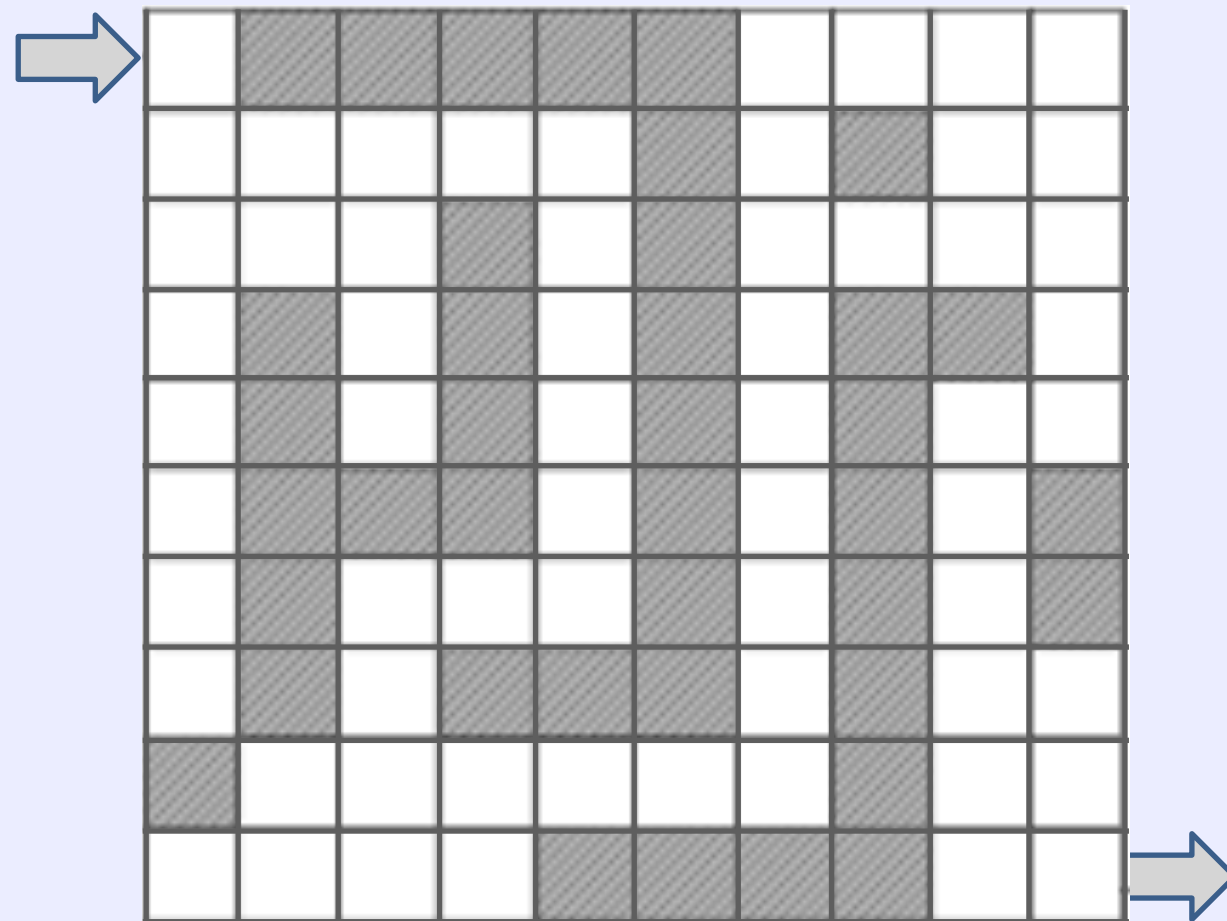


Operations stack

Homework

Use stack to obtain an algorithm for maze solving.

You can assume that maze is represented by a binary matrix M of size $n \times n$.



Maze solving