Data Structures and Algorithms

(CS210A)

Semester I – **2014-15**

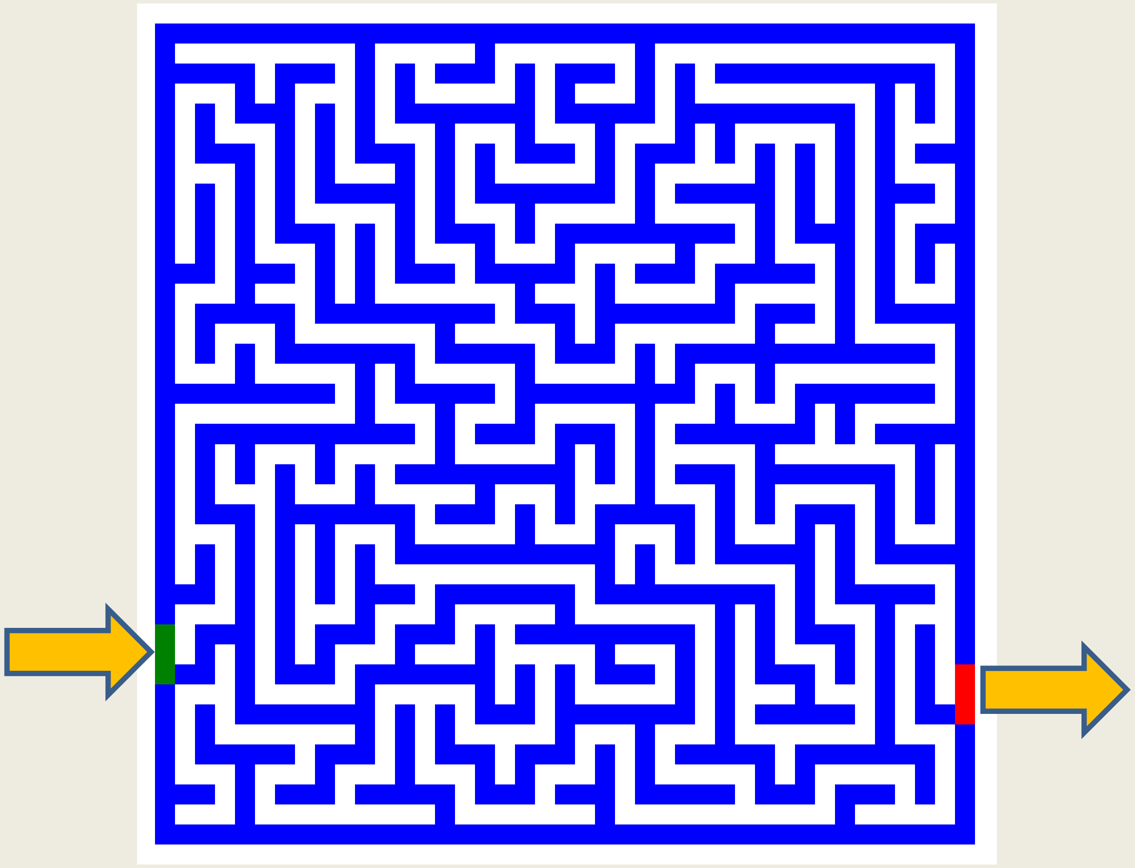
Lecture 9:

- **Stack:** A new data structure
- **Proof of correctness** : Binary search

Motivating Examples

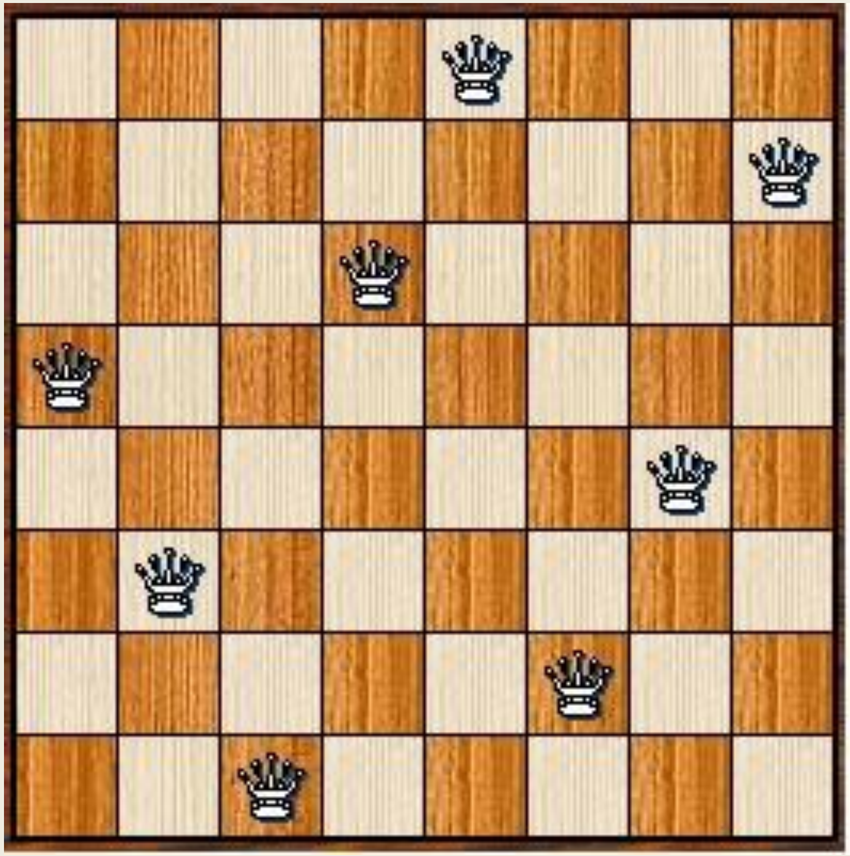
Finding path in a maze

Problem : How to design an algorithm for finding a path in a maze ?



8-Queens Problem

Problem: How to efficiently find a way to place **8 queens on a chess board** so that no two of them attack each other ?



Expression Evaluation

- **x** = **3+4*(5-6*(8+9^2)+3)**

Problem:

Can you write a program to evaluate any arithmetic expression ?

Stack: a data structure

Stack

Data Structure Stack:

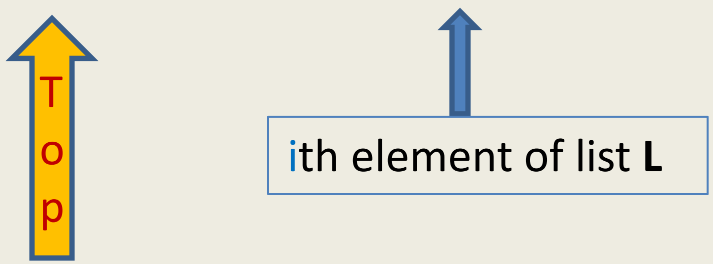
- **Mathematical Modeling of Stack**
- **Implementation of Stack** ← **will be left as an exercise**

Revisiting List

List is modeled as a sequence of elements.

we can insert/delete/query element <u>at any arbitrary position</u> in the list.

L: $a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n$

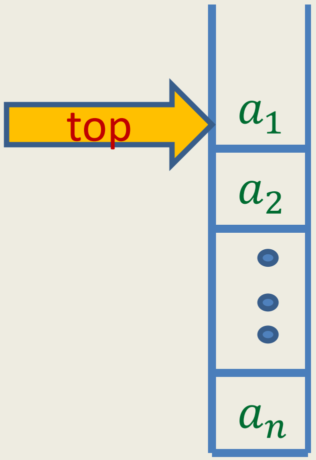


What if we restrict all these operations to take place <u>only at one end</u> of the list ?

Stack: a new data structure

A special kind of list

where all operations (insertion, deletion, query) take place at one end only, called the **top**.



Operations on a Stack

Query Operations

- **IsEmpty(S)**: determine if **S** is an empty stack.
- **Top**(**S**): returns the element at the top of the stack.

  Example: If **S** is a_1, a_2, …, a_n , **then Top**(**S**) returns a_1 .

Update Operations

- **CreateEmptyStack**(**S**): Create an empty stack.
- **Push**(**x,S**): push **x** at the top of the stack **S**.

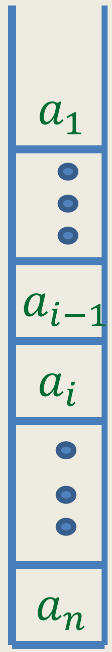
  Example: If **S** is a_1, a_2,…, a_n, then after **Push**(**x,S**), stack **S** becomes

  x, a_1, a_2 ,…, a_n
- **Pop**(**S**): Delete element from top of the stack **S.**

  Example: If **S** is a_1, a_2,…, a_n, then after **Pop**(**S**), stack **S** becomes

  a_2 ,…, a_n

An Important point about stack:
How to access ith element from the top ?



- To access ith element, we **must** pop (hence delete) one by one the top i-1 elements from the stack.

A puzzling question/confusion

- **Why do we restrict** the functionality of a list ?
- **What will be the use** of such restriction ?

How to evaluate an arithmetic expression

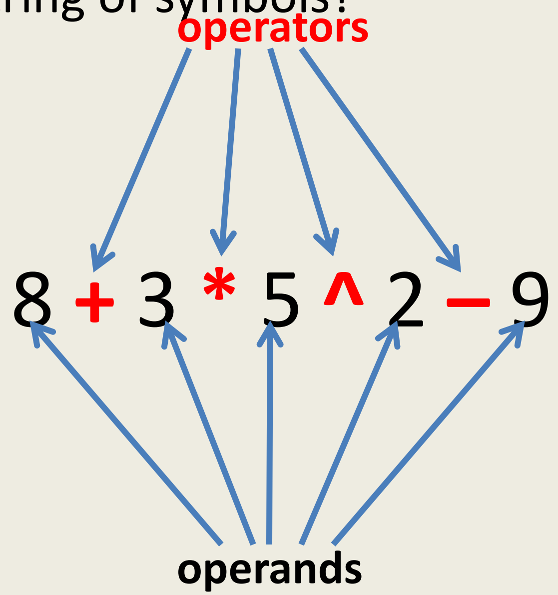
Evaluation of an arithmetic expression

Question: How does a computer/calculator evaluate an arithmetic expression given in the form of a string of symbols ?

$$8 + 3 * 5 \wedge 2 - 9$$

Evaluation of an arithmetic expression

Question: How does a computer/calculator evaluate an arithmetic expression given in the form of a string of symbols?



First it splits the string into **tokens** which are operators or operands (numbers). This is not difficult. But how does it evaluate it finally ???

Precedence of operators

Precedence: "priority" among different operators

- Operator **+** has same precedence as **–**.
- Operator ***** (as well as **/**) has higher precedence than **+**.
- Operator ***** has same precedence as **/**.
- Operator **^** has higher precedence than ***** and **/**.

Associativity of operators

What is 2^3^2 ?

What is 3-4-2 ?

What is 4/2/2 ?

Associativity: "How to group operators of same type ?"

A ● B ● C = ??

(A ● B) ● C **or** A ● (B ● C)

Left associative

Right associative

A trivial way to evaluate an arithmetic expression

8 + 3 * 5 ^ 2 - 9

- First perform all ^ operations.
- Then perform all * and / operations.
- Then perform all + and - operations.

Disadvantages:

1. An ugly and case analysis based algorithm
2. Multiple scans of the expression (one for each operator).
3. What about expressions involving parentheses: 3+4*(5-6/(8+9^2)+33)
4. What about associativity of the operators:
   - 2^3^2 = 512 **and** not 64
   - 16/4/2 = 2 **and** not 8.

Overview of our solution

1. **Focusing on a simpler version of the problem:**
    1. Expressions without parentheses
    2. Every operator is left associative

2. **Solving the simpler version**

3. **Transforming the solution of simpler version to generic**

Step 1

Focusing on a simpler version of the problem

Incorporating precedence of operators through priority number

Operator	Priority
+ , -	1
* , /	2
^	3

Insight into the problem

Let o_i : the operator at position *i* in the expression.

Aim: To determine an order in which to execute the operators.

8 **+** 3 ***** 5 **^** 2 **–** 9 ***** 67

Position of an operator **<u>does</u>** matter

Question: Under what conditions can we execute operator o_i immediately?

Answer: if

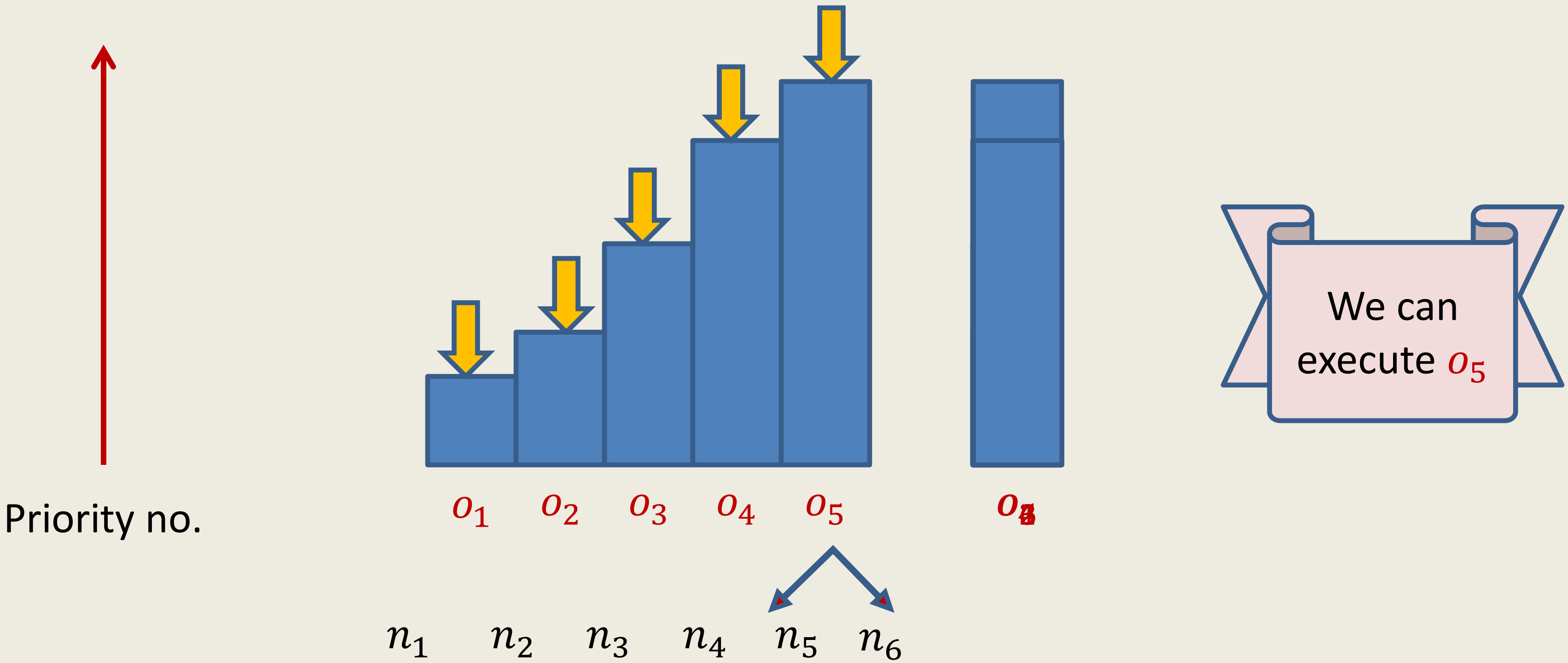
- priority(o_i) > priority(o_{i-1})
- priority(o_i) ≥ priority(o_{i+1})

Give reasons for ≥ instead of >

Question:

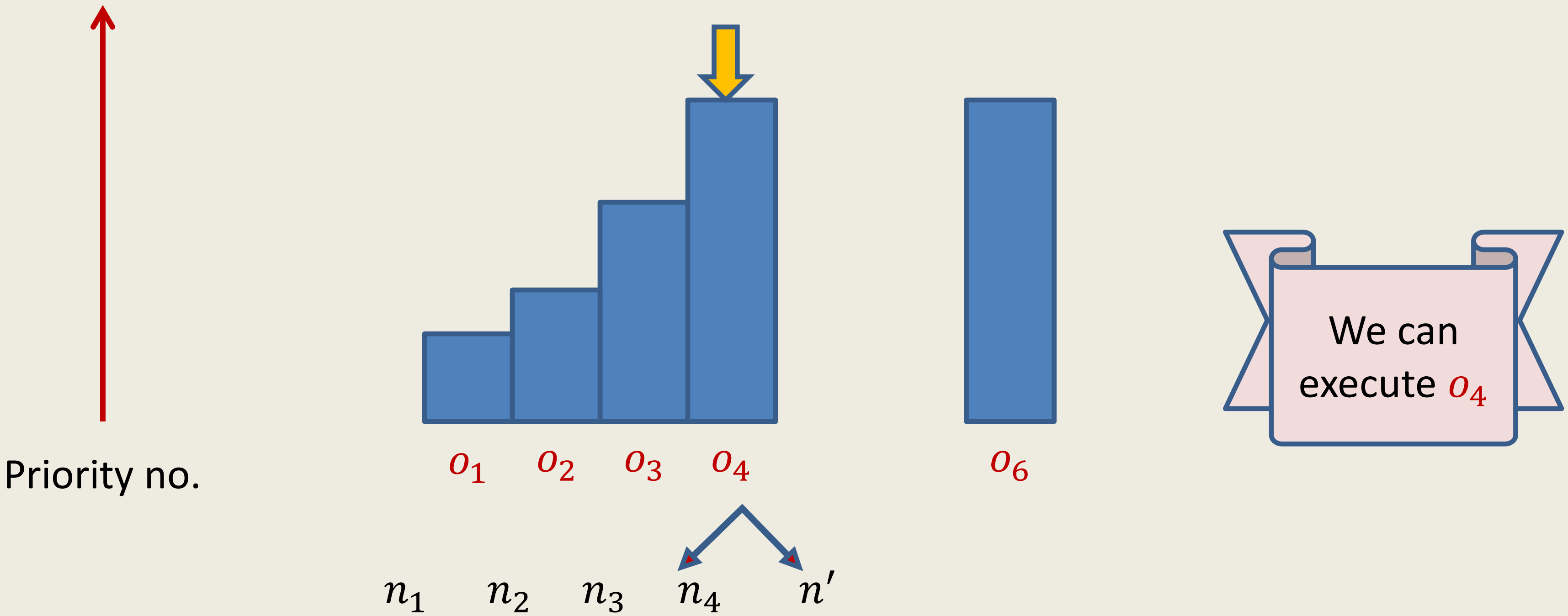
How to evaluate expression in a single scan ?

Expression: $n_1 o_1\ n_2 o_2\ n_3\ o_3\ n_4\ o_4\ n_5\ o_5$ …



Question:
How to evaluate expression in a single scan ?

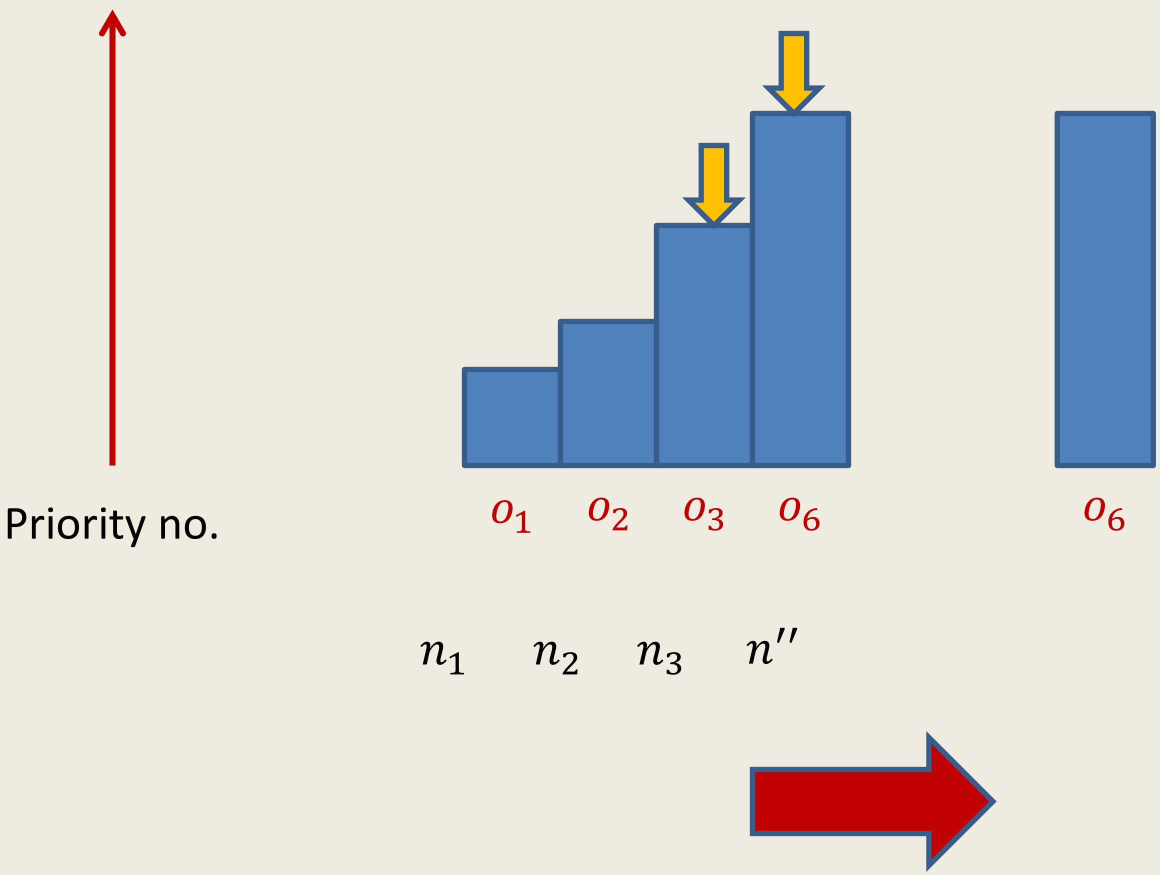
Expression: $n_1 o_1\ n_2 o_2\ n_3\ o_3\ n_4\ o_4\ n_5\ o_5\ \ldots$



Question:

How to evaluate expression in a single scan ?

Expression: $n_1 o_1\ n_2 o_2\ n_3\ o_3\ n_4\ o_4\ n_5\ o_5\ \ldots$



Homework:

Spend sometime to design an algorithm for evaluation of arithmetic expression based on the insight we developed in the last slides.

(hint: use 2 stacks.)

Proof of correctness : Binary search

Binary Search

```
Binary-Search(A[0…n − 1], x)
L ← 0;
R ← n − 1;
Found ← false;
While ( L ≤ R and Found = false )
{     mid ← (L+R)/2;
      If (A[mid] = x) Found ← true;
      else if (A[mid] < x) L ← mid + 1 ;
               else R ← mid - 1 ;
}
if Found return true;
else return false;
```

Observation: If the code returns true, then indeed **output** is correct.

So all we need to prove is that whenever code returns false , then indeed x is not present in **A**[].

This is because Found is set to true only when x is indeed found.

We proved it interactively in the class.