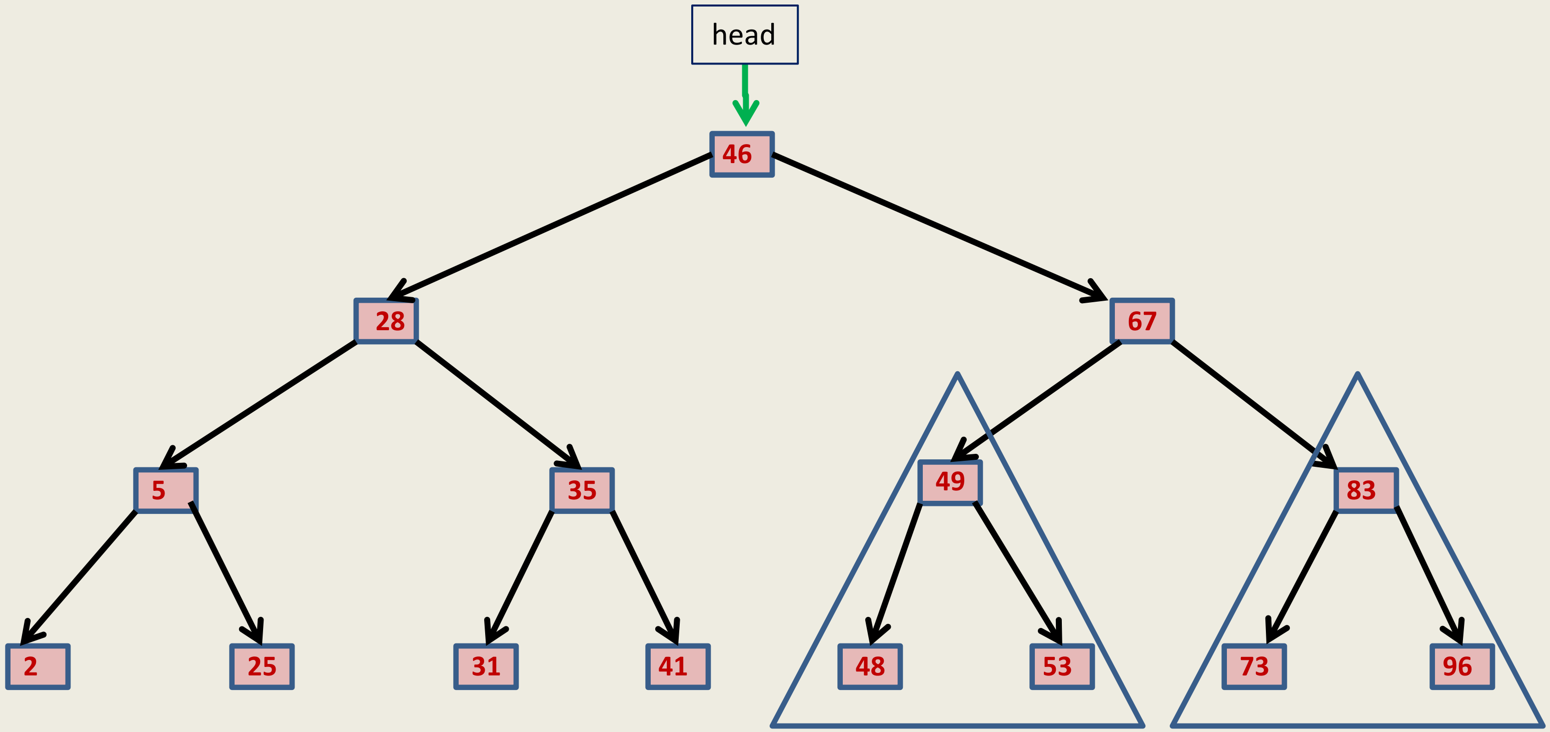
Data Structures and Algorithms

(ESO207)

Lecture 10:

- Exploring nearly balanced BST for the directory problem
- Stack: a new data structure

Binary Search Tree (BST)



Definition: A Binary Tree **T** storing values is said to be **Binary Search Tree**

if for each node **v** in **T**

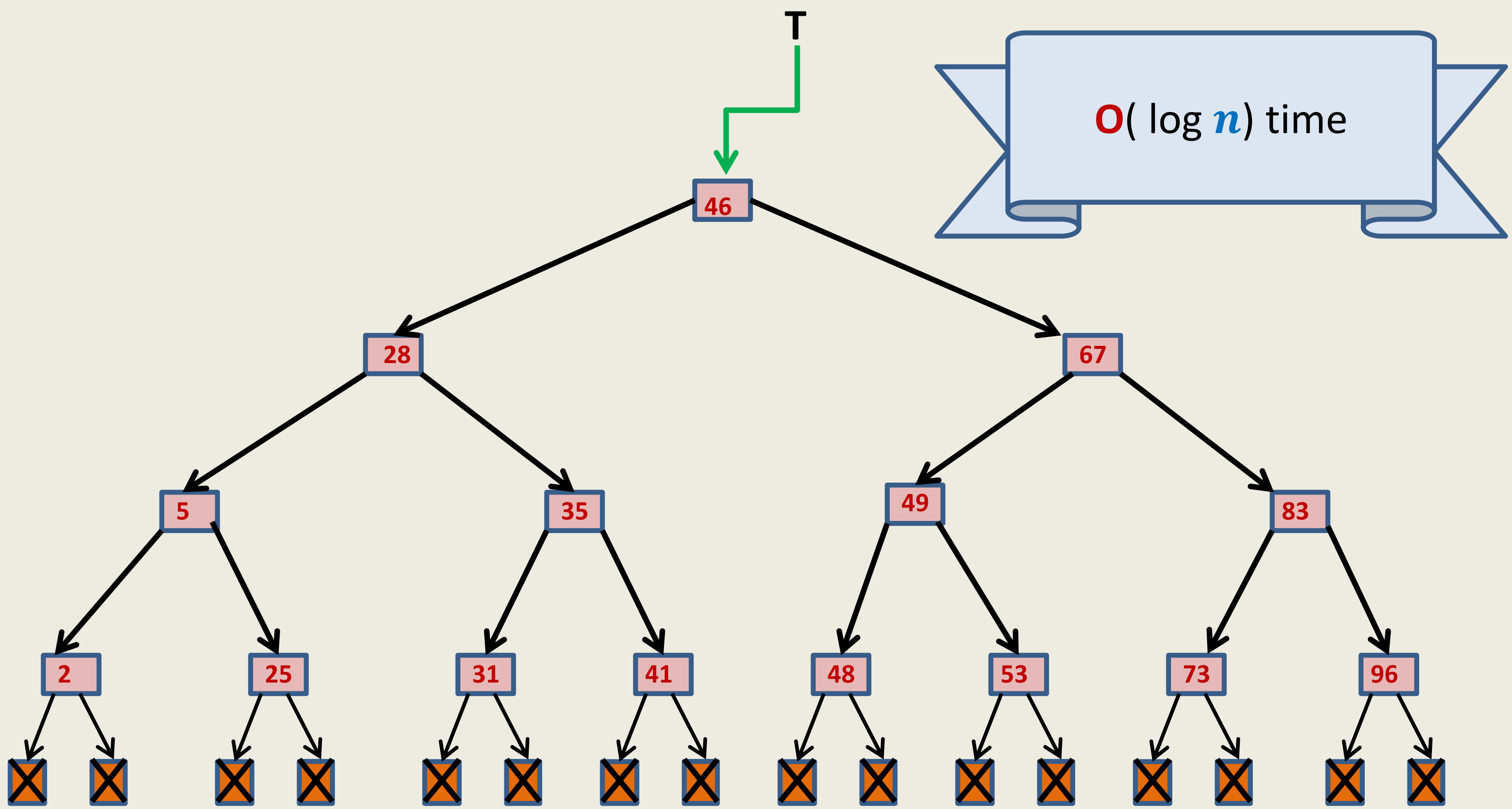
- If **left**(**v**) <> NULL, then **value**(**v**) > **value** of every node in **subtree**(**left**(**v**)).
- If **right**(**v**)<>NULL, then **value**(**v**) < **value** of every node in **subtree(right**(**v**)).

A question

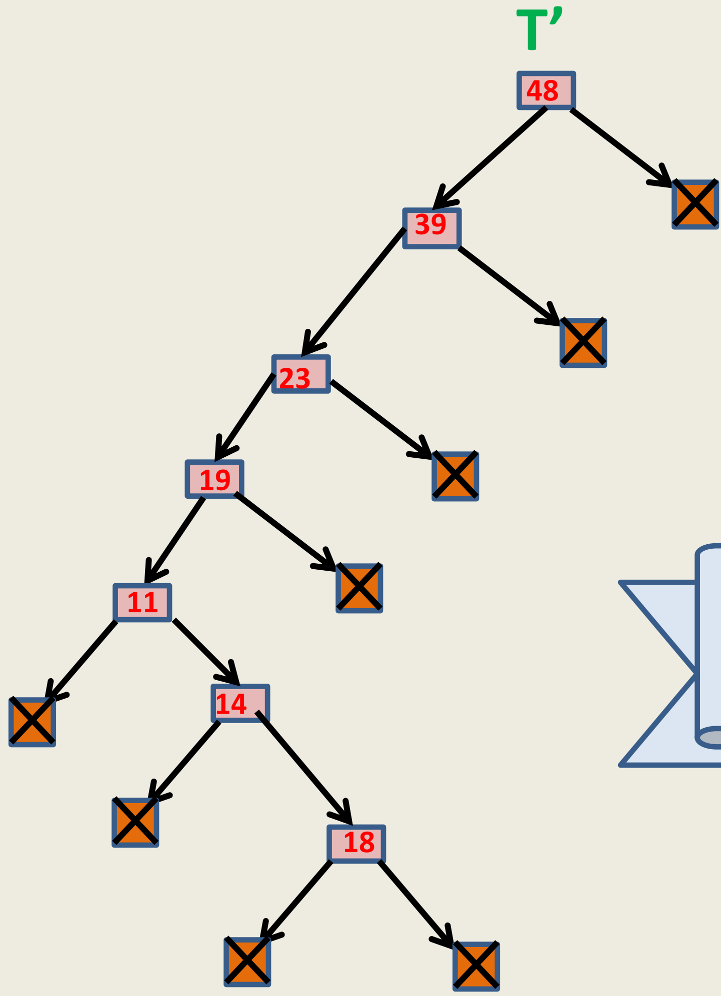
Time complexity of

Search(T,x) and **Insert**(T,x) in a Binary Search Tree **T** = **O(Height(T))**

Time complexity of any search and any single insertion in a perfectly balanced Binary Search Tree on n nodes



Time complexity of any search and any single insertion in a sqewed Binary Search Tree on n nodes



$O(n)$ time !☹

Our Original Problem

Maintain a telephone directory

Operations:

- Search the phone # of a person with **ID** no. **x**
- Insert a new record (**ID** no., phone #,…)

Array based solution	Linked list based solution
Log n	**O(n)**
O(n)	**Log n**

Solution : We may keep **perfectly balanced** BST.

Hurdle: What if we insert records in increasing order of **ID** ?

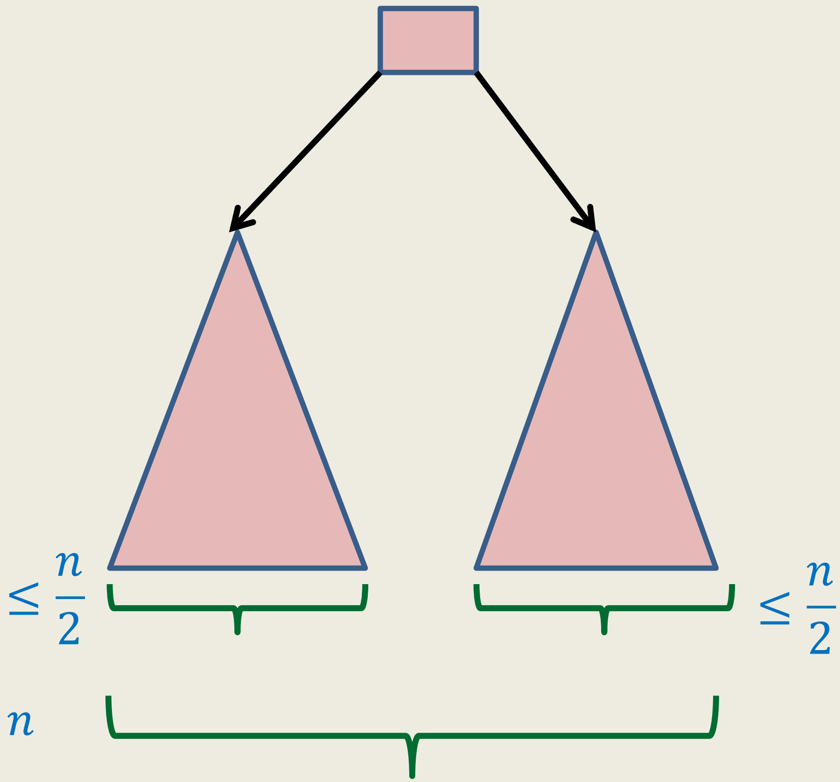
➔ BST will be skewed ☹

BST data structure that we invented looks very elegant,

let us try to find a way to overcome the hurdle.

- Let us try to find a way of achieving **Log** n search time.
- Perfectly balanced BST achieve **Log** n search time.
- But the definition of Perfectly balanced BST looks **too restrictive**.
- Let us investigate : How crucial is perfect balance of a BST ?

How crucial is perfect balance of a BST ?

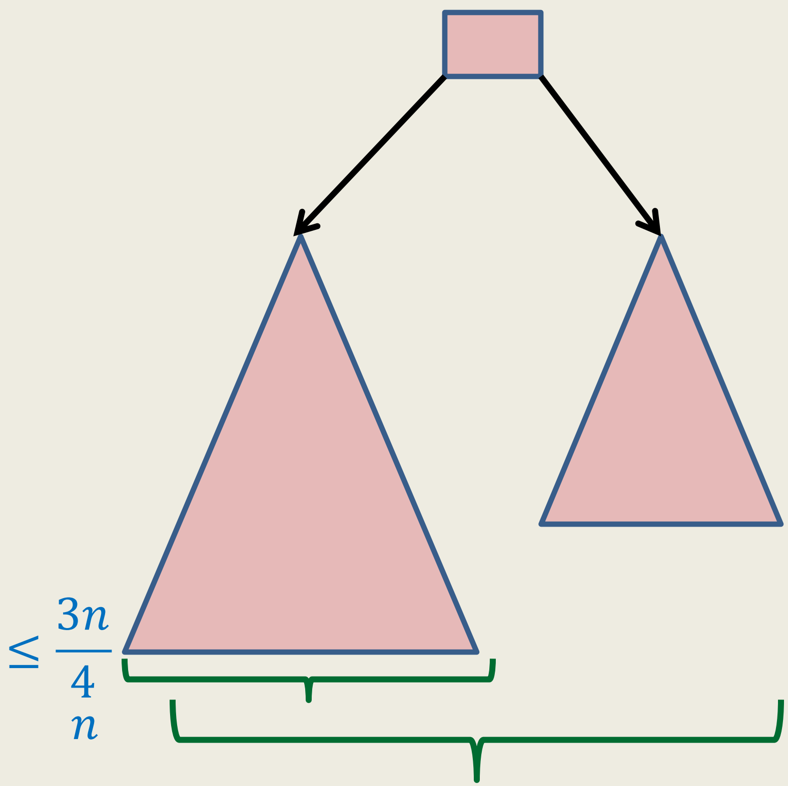


$H(1) = 0$

$H(n) = \leq 1 + H\left(\frac{n}{2}\right)$

Let us change this recurrence slightly.

How crucial is perfect balance of a BST ?



$H(1) = 0$

$$H(n) \leq \mathbf{1} + H\left(\frac{3n}{4}\right)$$

$$\leq \mathbf{1} + \mathbf{1} + H\left(\left(\frac{3}{4}\right)^2 n\right)$$

$$\leq \mathbf{1} + \mathbf{1} + \cdots + H\left(\left(\frac{3}{4}\right)^i n\right)$$

$$\leq \mathbf{log}_{4/3}\, n$$

What lesson did you get from this recurrence ? Think for a while before going further …

Lesson learnt :
We may as well work with **nearly** balanced BST

Nearly balanced Binary Search Tree

Terminology:

size of a binary tree is the number of nodes present in it.

Definition: A binary search tree **T** is said to be **nearly balanced** at node **v**, if

size(**left**(**v**)) $\leq \frac{3}{4}$ **size**(**v**)

and

size(**right**(**v**)) $\leq \frac{3}{4}$ **size**(**v**)

Definition: A binary search tree **T** is said to be **nearly balanced** if it is **nearly balanced** at each node.

Nearly balanced Binary Search Tree

Think of ways of using **nearly balanced BST** for solving our dictionary problem.

You might find the following **observations/tools** helpful :

- If a node **v** is **perfectly balanced**, it requires **many insertions** till **v** ceases to remain **nearly balanced**.

- Any arbitrary **BST** of size n can be converted into a **perfectly balanced BST** in **O**(n) time.

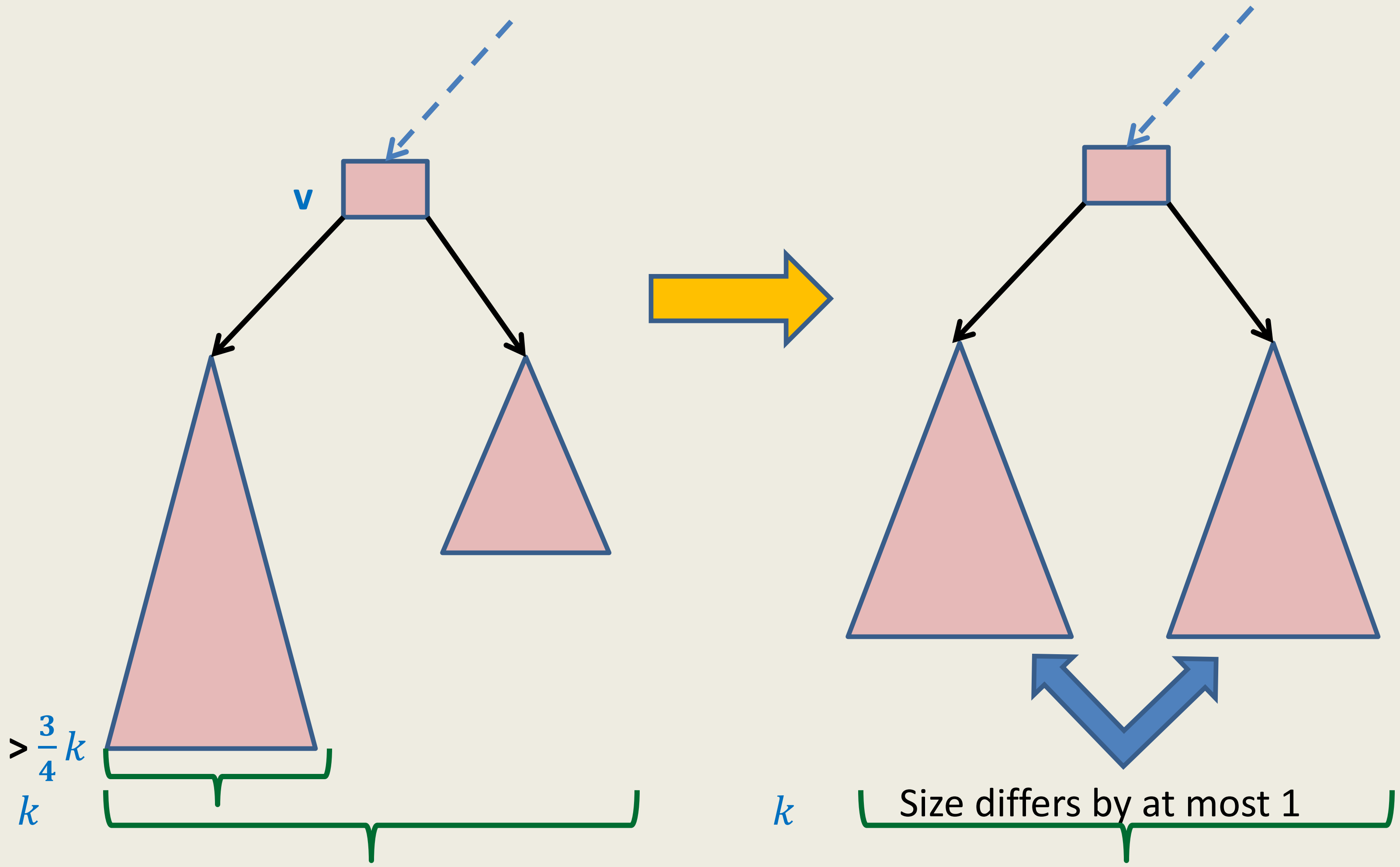
Solving our dictionary problem
Preserving O(log n) height after each operation

- Each node v in T maintains an additional field **size**(v) which is the number of nodes in the **subtree**(v).

- Keep **Search**(**T**,x) operation unchanged.

- Modify **Insert**(**T**,x) operation as follows:
  - Carry out normal insert and update the **size** fields of nodes traversed.
  - If BST **T** is ceases to be **nearly balanced** at any node **v**, transform **subtree(v)** into **perfectly balanced BST**.

“Perfectly Balancing” subtree at a node v



What can we say about this data structure ?

It is elegant and reasonably simple to implement.

Yes, there will be huge computation for *some* insertion operations.

But the number of such operations will be rare.

So, at least intuitively, the data structure appears to be efficient.

Indeed, this data structure achieve the following goals:

- For any arbitrary sequence of n **operations**, total time will be **O(n log n)**.
- Worst case search time: **O(log n)**

How can we justify these claims ?

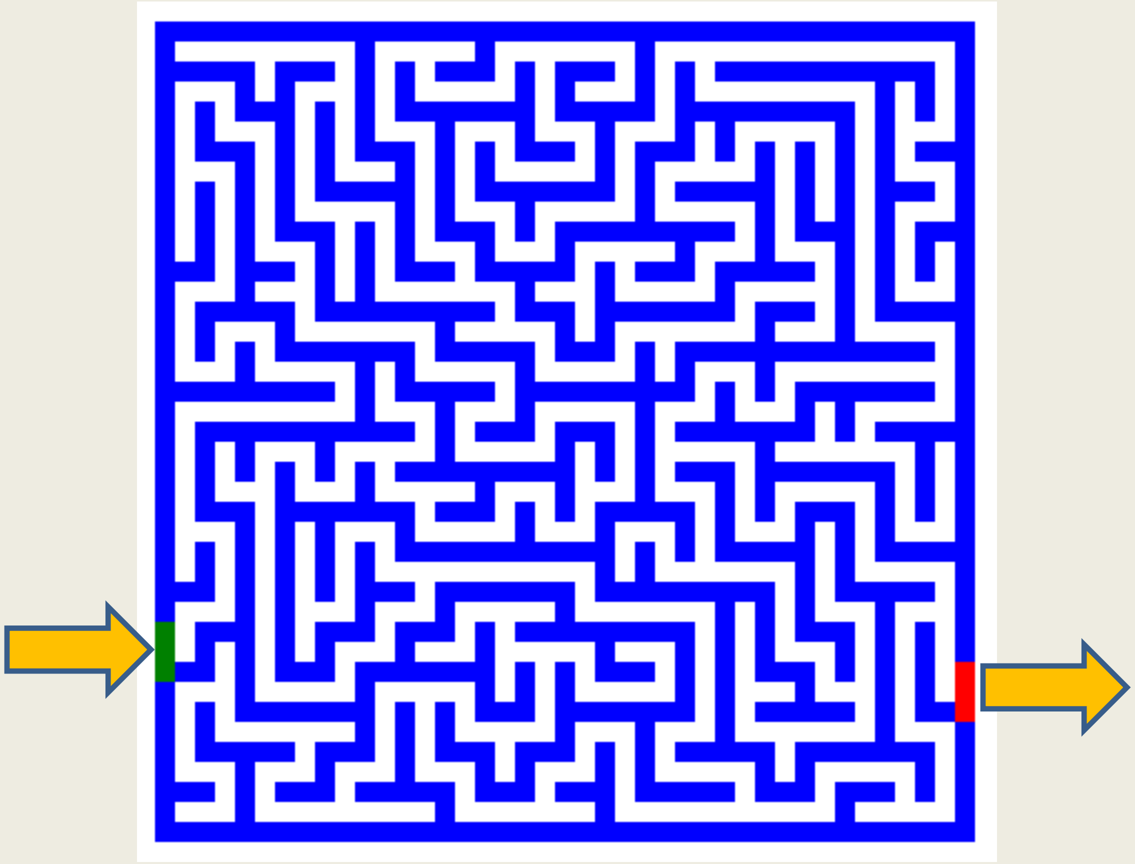
Keep thinking till we do it in a few weeks ☺.

Stack: a data structure

A few motivating examples

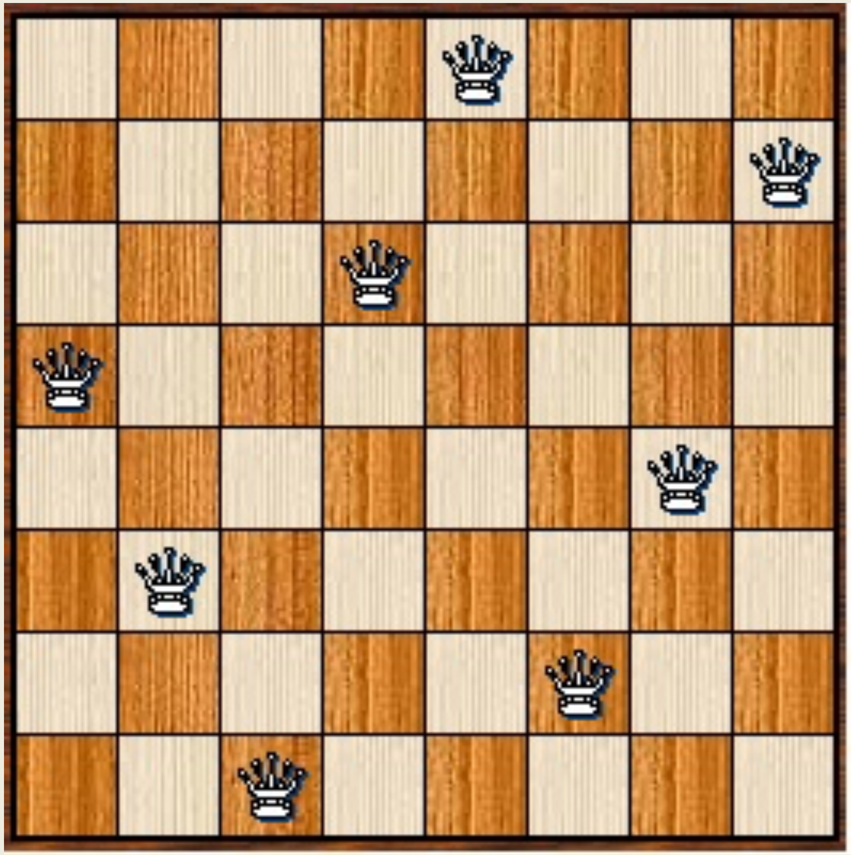
Finding path in a maze

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



8-Queens Problem

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



Expression Evaluation

- $\boldsymbol{x} = \mathbf{3 + 4 * (5 - 6 * (8 + 9^\wedge 2) + 3)}$

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 at any arbitrary position in the list.

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

Top

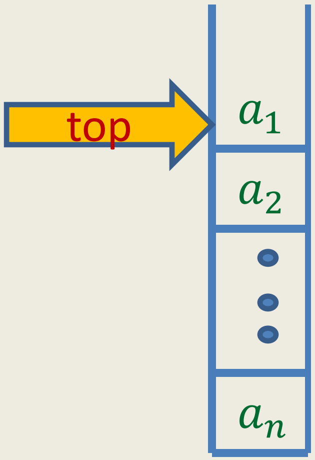
ith element of list **L**

What if we restrict all these operations to take place only at one end 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 $\boxed{a_1}$.

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

  $$\boxed{\mathbf{x}, a_1, a_2, \ldots, a_n}$$

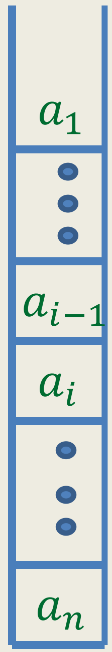
- **Pop**(**S**): Delete element from top of the stack **S**

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

  $$\boxed{a_2, \ldots, a_n}$$

An Important point about stack

How to access ith element from the top ?



- To access ith element, we must

  pop (hence <u>delete</u>) **<u>one by one</u>** 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 ?

operators

8 **+** 3 ***** 5 **^** 2 **–** 9

operands

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 <u>same</u> 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 **does** matter

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

Answer: if

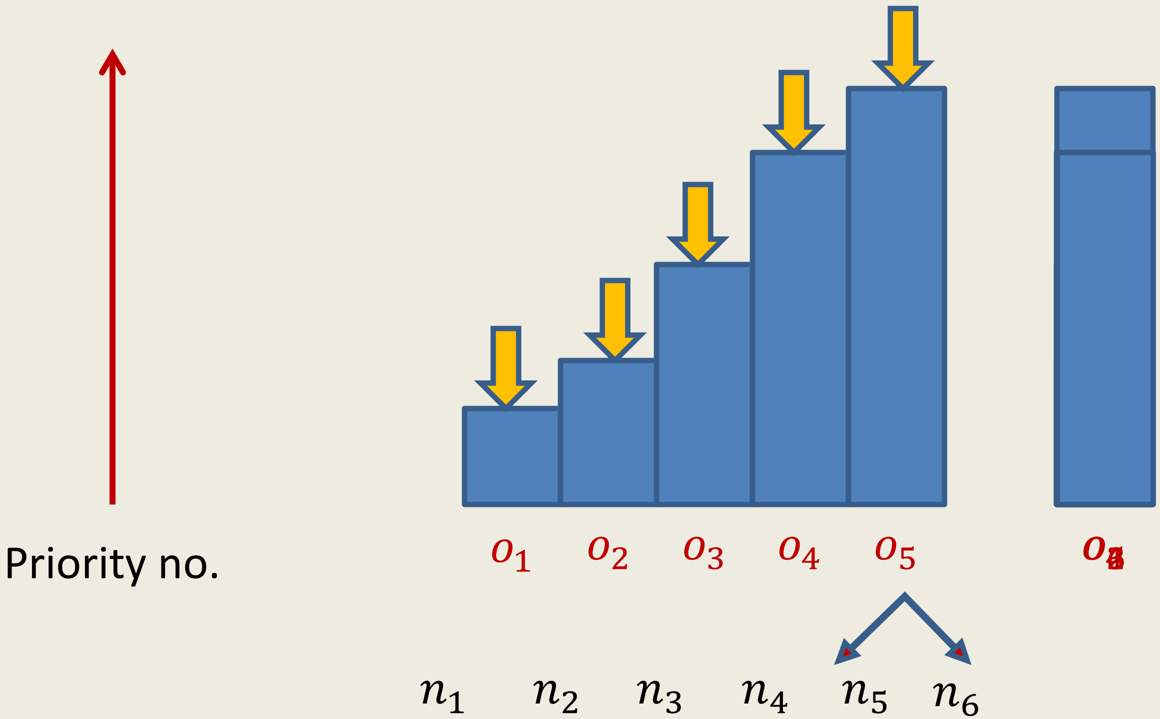
- priority(o_i) > priority(o_{i-1})
- priority(o_i) ≥ priority(o_{i+1})

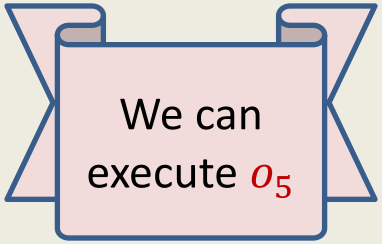
Give reasons for ≥ instead of >

Question:

How to evaluate expression in a single scan ?

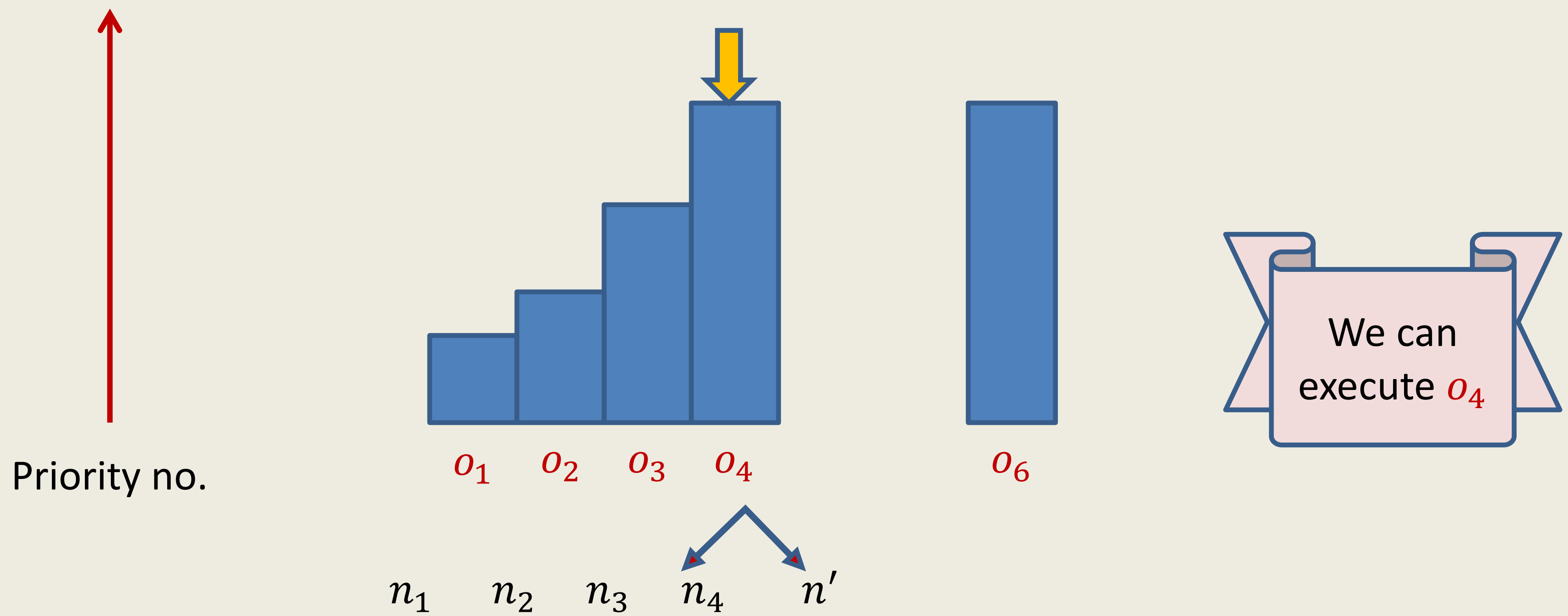
Expression: $n_1 o_1\ n_2 o_2\ n_3\ o_3\ n_4\ o_4\ n_5\ o_5\ n_6\ o_6$ …





Question:
How to evaluate expression in a single scan ?

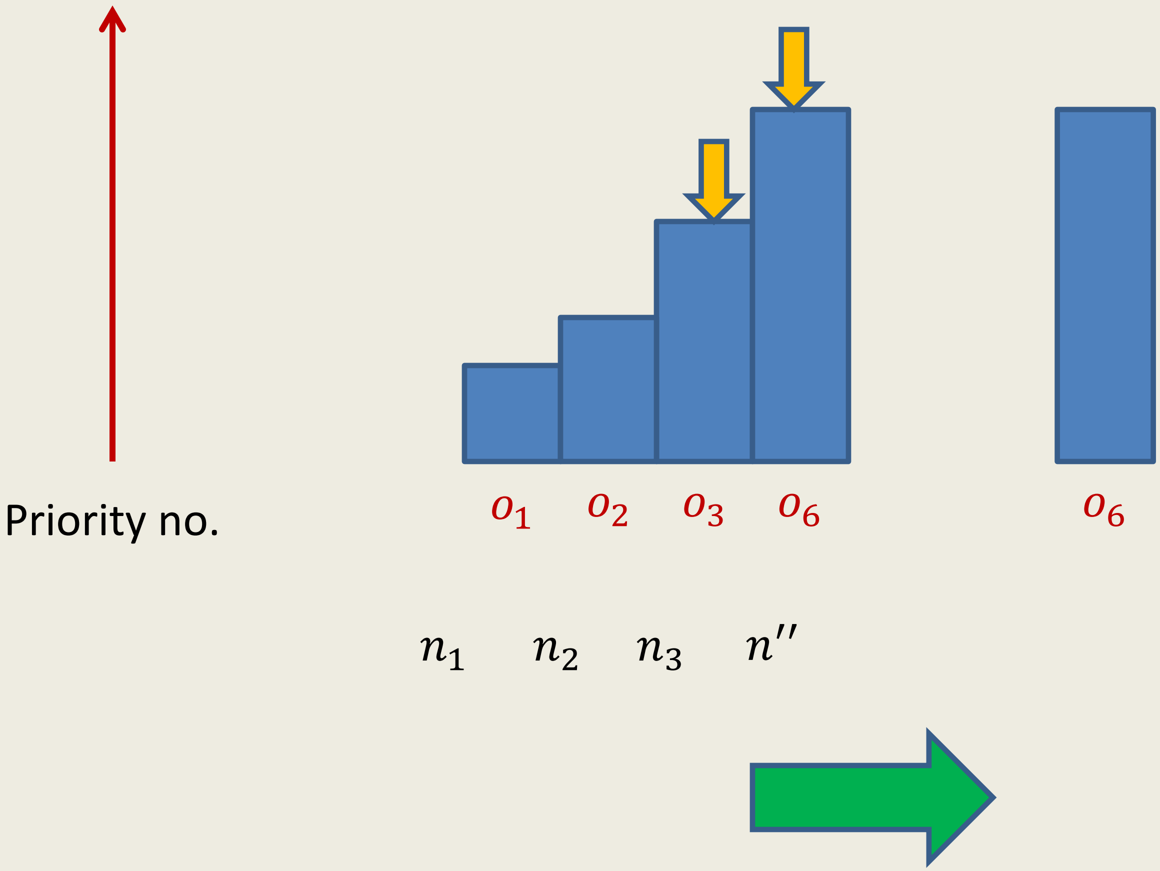
Expression: $n_1 o_1\ n_2 o_2\ n_3\ o_3\ n_4\ o_4\ n_5\ o_5\ n_6\ o_6$ …



Question:

How to evaluate expression in a single scan ?

Expression: $n_1 o_1\ n_2 o_2\ n_3\ o_3\ n_4\ o_4\ n_5\ o_5\ n_6\ o_6$ …



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.)