Stack ADT

CMPT 125
Mo Chen
SFU Computing Science
4/3/2020

Lecture 21

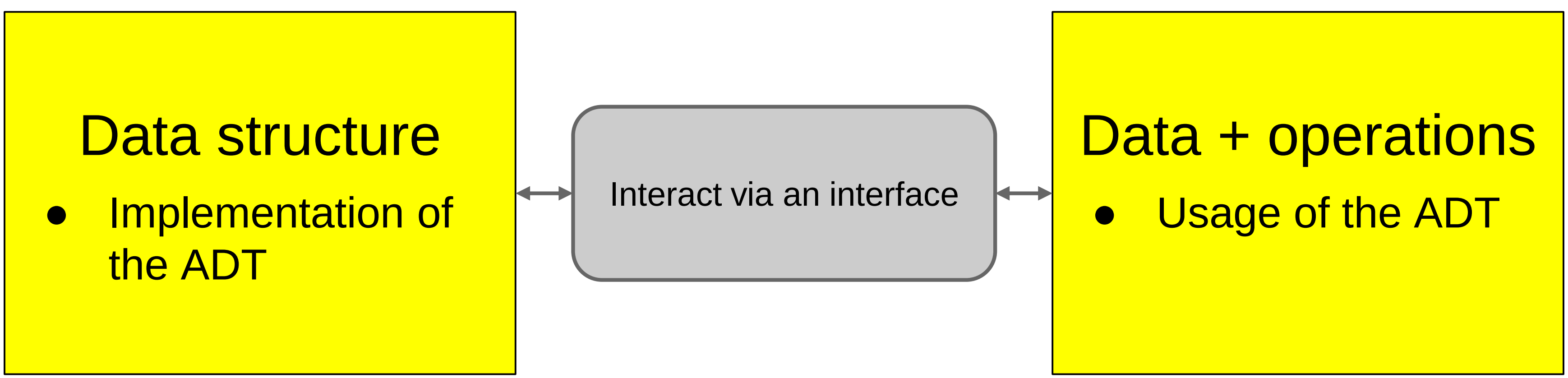
Today:

- Stack ADT
- Algorithms that use a Stack
- Implementing a Stack (with a Linked List)

Abstract Data Types (Review)

Abstract data type (ADT): a collection of data and a set of allowed operations on that data.

- specifies data and operations, not how the data are stored or how operations are carried out
- different from the data structure, which deals with the implementation



Why use interfaces? (Review)

Algorithm that implements a Stack ADT

Stack Interface:

- a sequence of data (LIFO)
- push element
- pop element
- isEmpty?

Algorithm that requires a Stack ADT

Why use interfaces? (Review)

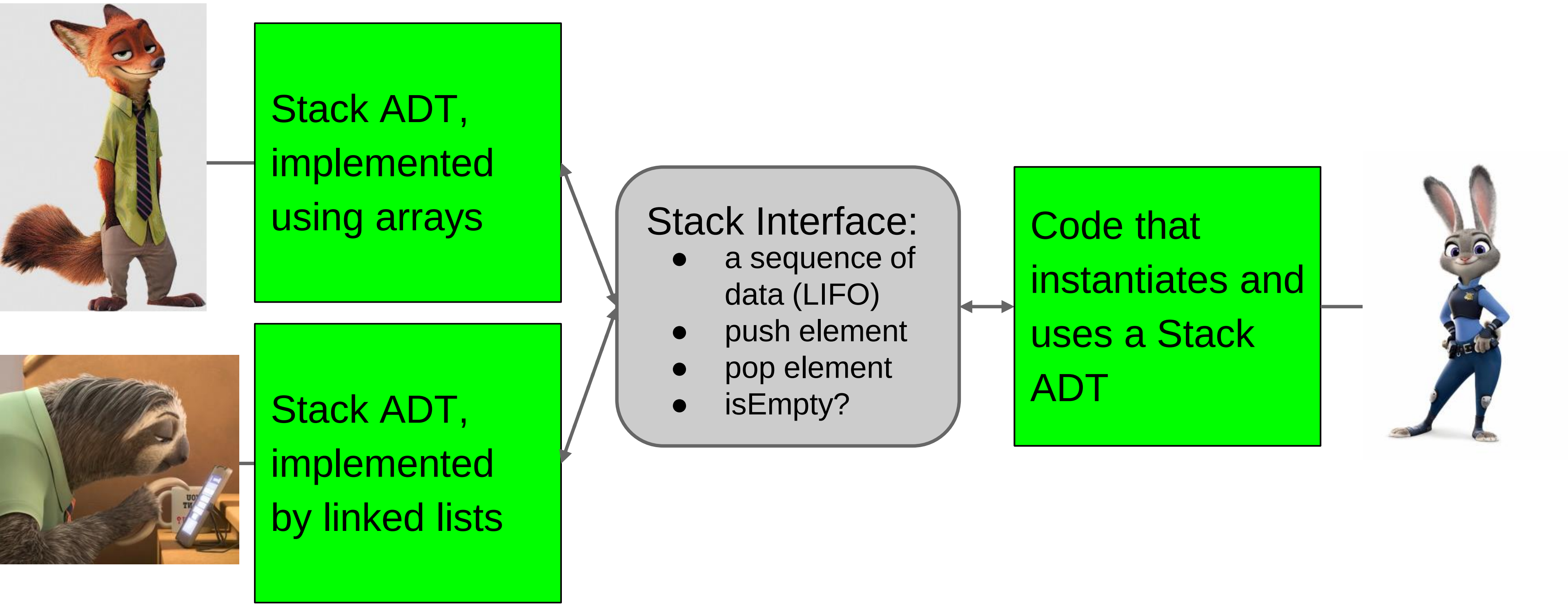
Stack ADT, implemented using arrays

Stack Interface:

- a sequence of data (LIFO)
- push element
- pop element
- isEmpty?

Code that instantiates and uses a Stack ADT

Why use interfaces? (Review)



Code independence

Why use interfaces? (Review)

Stack ADT, implemented using arrays

Stack ADT, implemented by linked lists

Stack Interface:
- a sequence of data (LIFO)
- push element
- pop element
- isEmpty?

Code that instantiates and uses a Stack ADT

Other code that instantiates and uses a Stack ADT

Code independence

Code re-usage

Postfix Calculation

A *postfix* operator comes after its operands

E.g. `24 6 + → 30`   `24 6 * → 144`

`24 6 - → 18`   `24 6 / → 4`

You are accustomed to `24 + 6`, which is *infix*.

No brackets are required in postfix

- operator always refers to last two numbers / results
- E.g. `24 6 * 15 3 - / → (24*6)/(15-3)`

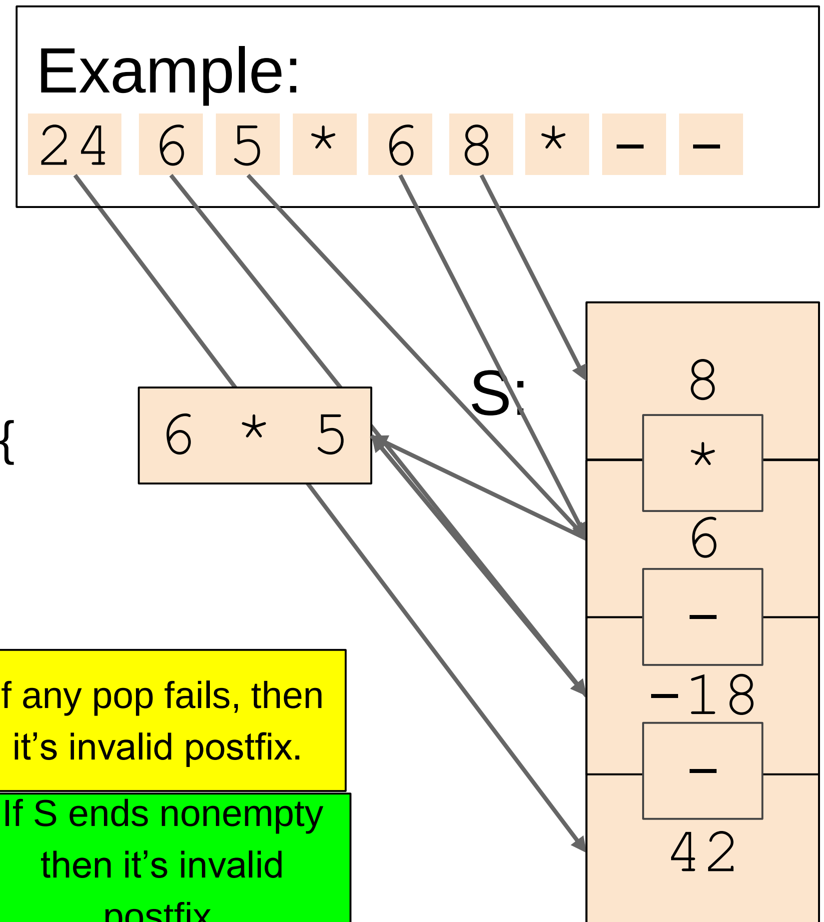
Q. Evaluate: (24((6 5 *)(6 8 *) −) −) → 42

Stack-Based Postfix Calculator

Use a Stack ADT to evaluate postfix.

Algorithm:

```
Create an empty stack S
while there is still input {
        if next input token is a number
                push the number to S
        if next input token is an operator {
                pop from S → b
                pop from S → a
                push (a op b) to S
        }
}
pop from S → result
```



If any pop fails, then it’s invalid postfix.

If S ends nonempty then it’s invalid postfix.

Balancing Brackets

Your compiler needs to be able to match pairs of 3 different types of brackets: `()`, `[]`, `{}`

- Each left one must have a matching right one.
- Nested brackets are OK, but mismatched brackets are disallowed.

E.g. `{[()]}` is acceptable, but `([)]` is not. Neither is `())` nor `{{}`.

Your compiler uses a stack to solve this problem too.

Stack-Based Bracket Balancer

Use a Stack ADT to balance brackets.

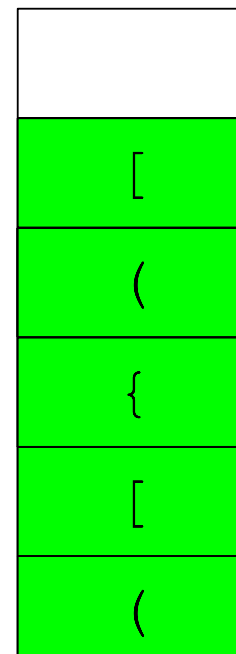
Algorithm:

```
Create an empty stack S
while there is still input {
        if next input token is a left bracket
                push it to S
        if next input token is a right bracket {
                pop from S → left
                if left doesn’t match right or failed pop then error
        }
}
if S not empty then error
```

Example:

([{ [ ] ([ ]) { } } [[ ]]])

S:



Implementation of Stack ADT

ADT implementations are tied to the data structure you choose:

- the faster, the better
- the smaller, the better

Big-O is the measuring stick

For today's implementation of a Stack, we choose linked lists, i.e., 1 Stack ↔ 1 Linked List.

Q. What's the running time of:

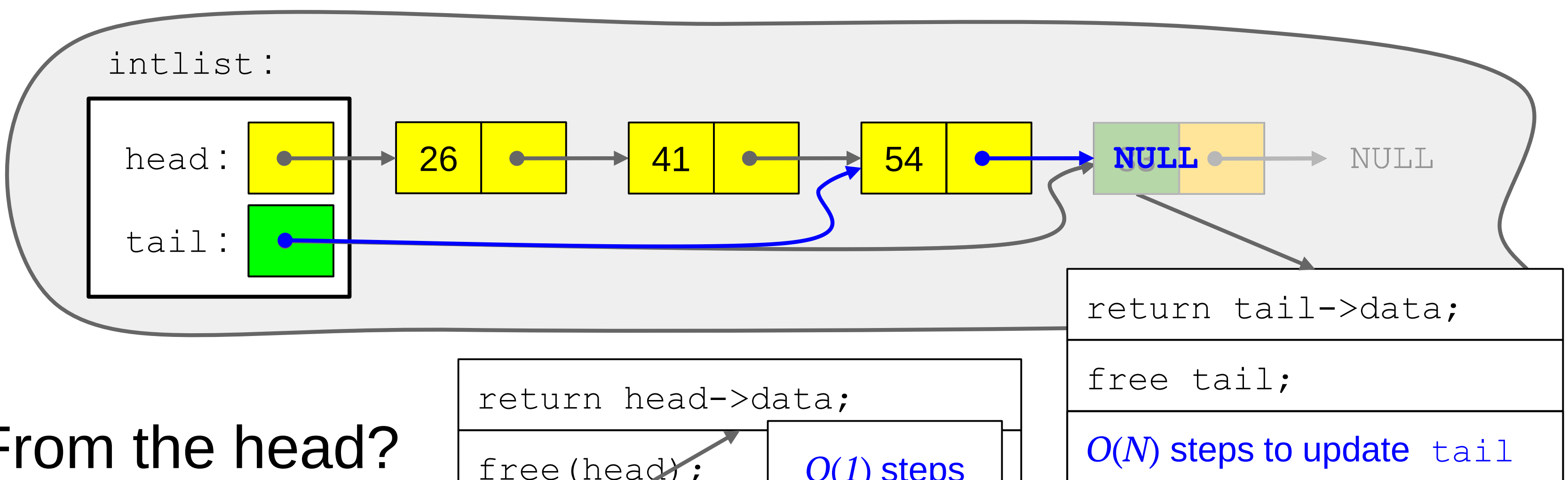
- `create()`?
- `isEmpty(S)`?
- `push(S, x)`?

Two options:
Can `LLappend(x)` to the tail OR can `LLprepend(x)` to the head.
Both are $O(1)$.

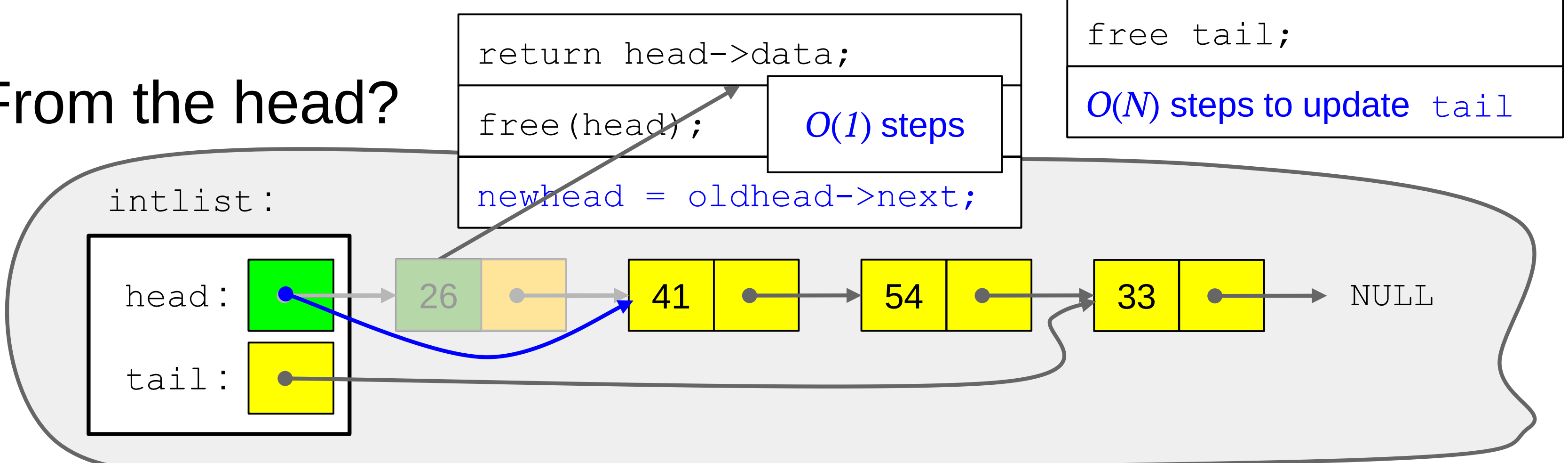
Implementing `pop(S)`

Q. From which end should you remove an item?

From the tail?



From the head?



Stack Implementation: Algorithms

```
create():
      return LLcreate();

isEmpty(S):
      return (S->head == NULL);

pop(S):
      return LLremoveHead(S);

push(S, x):
      LLprepend(S, x);
```