Abstract Data Types and Stacks

CSE 2320 – Algorithms and Data Structures
Vassilis Athitsos
University of Texas at Arlington

Abstract vs. Specific Data Types

- Specific data types: lists, arrays, strings.
  - Types for which we can refer to a SPECIFIC implementation.
- Abstract data types: sequences, trees, forests, graphs.
- Where have we used forests in this course?

- What is the difference between an abstract data type such as sequences, and a specific data type such as lists and arrays?

Abstract vs. Specific Data Types

- Specific data types: lists, arrays, strings.
  - Types for which we can refer to a SPECIFIC implementation.
- Abstract data types: sequences, trees, forests, graphs.
- Where have we used forests in this course?
  - In the Union-Find problem (forests are sets of trees).
- What is the difference between an abstract data type such as sequences, and a specific data type such as lists and arrays?
  - An abstract data type can be implemented in multiple ways.
  - Each of these ways can offer different trade-offs in performance, that may be desirable in different cases.

Regarding Lists

- Are lists a "specific" data type or an "abstract" data type?

Regarding Lists

- Are lists a "specific" data type or an "abstract" data type?
- If we refer to a **specific** implementation of lists, then we refer to a "specific" data type.
- If we are not referring to a "specific" implementation, then there is a certain level of abstraction.
  - Different choices lead to different performance:
  - Single links or double links?
  - Pointer to first element only, or also to last element?

Examples of Abstraction

- What are some examples of abstract data types, for which we have seen multiple specific implementations?

Examples of Abstraction

- Union-Find:
  - Representing sets, performing unions and performing finds can be done in different ways, with very different performance characteristics.
- Sequences:
  - They can be represented as arrays or lists.
- Graphs:
  - We saw two implementations (adjacency matrices and adjacency lists) that provide the same functionality, but are different in terms of time and space complexity.
  - Alternative implementations also exist, that have their own pros and cons.

Why Use Abstract Data Types

- Using abstract data types allows us to focus on high-level algorithm design, and not on low-level details of specific data types.
  - The data types should fit the algorithm, not the other way around.
- Designing an algorithm using abstract data types, we oftentimes get multiple algorithms for the price of one.
  - The high-level algorithm can be implemented in multiple ways, depending on our choice of specific data types.
- Each choice may give different performance trade-offs.
  - Choosing a specific data type may yield better space complexity but worse time complexity.

Why Use Abstract Data Types

- Example (that we will see later in the course): search algorithms.
  - Used for navigation, game playing, problem solving…
- Several search algorithms, with vastly different properties, can be described with the same algorithm.
- The only thing that changes is one data type choice: what type of queue to use.

Generalized Queues

- A generalized queue is an abstract data type that stores a set of objects.
  - Let's use **Item** to denote the data type of each object.
- The fundamental operations that such a queue must support are:

  void insert(Queue q, Item x): adds object **x** to set **q**.

  Item delete(Queue q): choose an object **x**, remove that object from **q**, and return it to the calling function.

Generalized Queues

- Basic operations:
  - **void insert(Queue q, Item x)**
  - **Item delete(Queue q)**
- The meaning of **insert** is clear in all cases: we want to add an item to an existing set.
- However, we note that **delete** does NOT take as an argument the item we want to delete, so the function itself must choose.

Generalized Queues - Delete

- How can **delete** choose which item to delete?
  - Choose the item that was inserted last.
  - Choose the item that was inserted first.
  - Choose a random item.
  - If each item contains a **key** field: remove the item whose key is the smallest.
- You may be surprised as you find out, in this course, how important this issue is.
- We will spend significant time studying solutions corresponding to different choices.

The Pushdown Stack

- The pushdown stack behaves like the desk of a busy (and disorganized) professor.
  - Work piles up in a stack.
  - Whenever the professor has time, he picks up whatever is on top and deals with it.
- We call this model a LIFO (last-in, first-out) queue.
  - The object that leaves the stack is always **the object that was inserted last** (among all objects still in the stack).
- Most of the times, instead of saying "pushdown stack" we simply say "stack".
  - By default, a "stack" is a pushdown stack.

Push and Pop

- The pushdown stack supports **insert** and **delete** as follows:
  - ~~**insert**~~ **push:** This is what we call the insert operation when we talk about pushdown stacks. It puts an item "on top of the stack".
  - ~~**delete**~~ **pop**: This is what we call the delete operation when we talk about pushdown stacks. It removes the item that was on top of the stack (the last item to be pushed, among all items still on the stack).

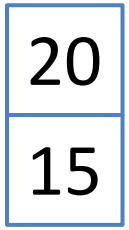
Examples of Push and Pop

- **push(15)**
- push(20)
- pop()
- push(30)
- push(7)
- push(25)
- pop()
- push(12)
- pop()
- pop()

15

Examples of Push and Pop

- push(15)
- **push(20)**
- pop()
- push(30)
- push(7)
- push(25)
- pop()
- push(12)
- pop()
- pop()



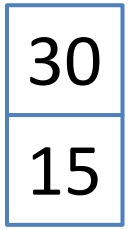
Examples of Push and Pop

- push(15)
- push(20)
- **pop()** – returns 20
- push(30)
- push(7)
- push(25)
- pop()
- push(12)
- pop()
- pop()

15

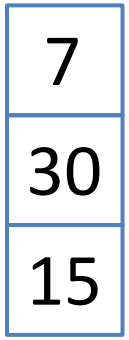
Examples of Push and Pop

- push(15)
- push(20)
- pop()
- **push(30)**
- push(7)
- push(25)
- pop()
- push(12)
- pop()
- pop()



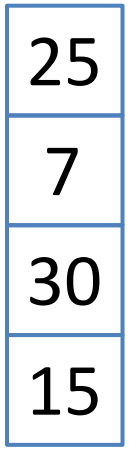
Examples of Push and Pop

- push(15)
- push(20)
- pop()
- push(30)
- **push(7)**
- push(25)
- pop()
- push(12)
- pop()
- pop()



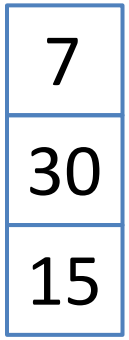
Examples of Push and Pop

- push(15)
- push(20)
- pop()
- push(30)
- push(7)
- **push(25)**
- pop()
- push(12)
- pop()
- pop()



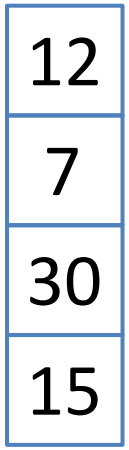
Examples of Push and Pop

- push(15)
- push(20)
- pop()
- push(30)
- push(7)
- push(25)
- **pop()** – returns 25
- push(12)
- pop()
- pop()



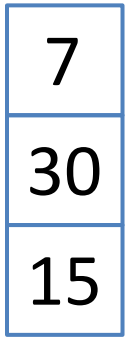
Examples of Push and Pop

- push(15)
- push(20)
- pop()
- push(30)
- push(7)
- push(25)
- pop()
- **push(12)**
- pop()
- pop()



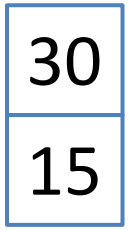
Examples of Push and Pop

- push(15)
- push(20)
- pop()
- push(30)
- push(7)
- push(25)
- pop()
- push(12)
- **pop()** – returns 12
- pop()



Examples of Push and Pop

- push(15)
- push(20)
- pop()
- push(30)
- push(7)
- push(25)
- pop()
- push(12)
- pop()
- **pop()** – returns 7



Implementation Details: Later

- We temporarily postpone discussing how stacks are implemented.
- We will first talk about how stacks can be **used**.
- Why? This is a good exercise for getting used to separating these two issues:
  - How a data type is implemented.
  - How a data type is used.
- Knowing that stacks support **push** and **pop** is sufficient to allow us to design programs using stacks.

Uses of Stacks

- Modeling a busy professor's desk is NOT the killer app for stacks.
- Examples of important stack applications:

Uses of Stacks

- Modeling a busy professor's desk is NOT the killer app for stacks.
- Examples of important stack applications:
  - Function execution in computer programs: when a function is called, it enters the **calling stack**. The function that leaves the calling stack is always the last one that entered (among functions still in the stack).
  - Interpretation and evaluation of symbolic expressions: stacks are used to evaluate things like (5+2)*(12-3), or to parse C code (as a first step in the compilation process).
  - Search methods. Search is a fundamental algorithmic topic, with applications in navigation, game playing, problem solving… We will see more later in the course.

Stacks and Calculators

- Consider this expression:
  - 5 * (((9 + 8) * (4 * 6)) + 7)
- This calculation involves saving intermediate results.
- First we calculate (9 + 8).
- We save 17 (push it to a stack).
- Then we calculate (4 * 6) and save 24 (push to a stack).
- Then we pop 17 and 24, multiply them, save the result.
- And so on…

Infix and Postfix Notation

- The standard notation we use for writing mathematical expressions is called **infix notation**.
- Why? Because the operators are between the operands.
- There are two alternative notations:
  - **prefix notation**: the operator comes before the operands.
  - **postfix notation**: the operator comes after the operands.
- Example:
  - **infix:** 5 * (((9 + 8) * (4 * 6)) + 7)
  - **prefix:** (* 5 (+ (* (+ 9 8) (* 4 6)) 7))
  - **postfix:** 5 9 8 + 4 6 * * 7 + *

Postfix Notation

- Example:
  - **infix:** 5 * (((9 + 8) * (4 * 6)) + 7)
  - **postfix:** 5 9 8 + 4 6 * * 7 + *
- Postfix notation does not need any parentheses.
- It is pretty easy to write code to evaluate postfix expressions (we will).

Processing a Symbolic Expression

- How do we process an expression such as:
  - 5 * (((9 + 8) * (4 * 6)) + 7)
  - postfix: 5 9 8 + 4 6 * * 7 + *
- One approach is textbook Program 4.2.
  - Only few lines of code, but dense and hard to read.
- Second approach: think of the input as a **stream of tokens**.
- A **token** is a **logical unit** of input, such as:
  - A number
  - An operator
  - A parenthesis.

Tokens

- A **token** is a **logical unit** of input, such as:
  - A number
  - An operator
  - A parenthesis.
- What are the tokens in:
  - 51 * (((19 + 8) * (4 - 6)) + 7)
- Answer: 51, *, (, (, (, 19, +, 8,), *, (, 4, -, 6,),), +, 7,)
  - 19 tokens.

Tokens

- A **token** is a **logical unit** of input, such as:
  - A number
  - An operator
  - A parenthesis.
- We need a data type for a token.
- We need to write functions that can read data (from a string or from a file) **one token at a time**.
  - See files tokens.h and tokens.c on the course website.
- Using tokens: it is a bit harder to get started with the code (compared to Programs 4.2 and 4.3), but much easier to extend the code to more complicated tasks.

Processing Postfix: Pseudocode

- **input:** a stream of tokens in infix order.
  - What do we mean by **stream**?

Processing Postfix: Pseudocode

- **input:** a stream of tokens in infix order.
  - What do we mean by **stream**?
  - A **stream** is any source of data from which we can read data one unit at a time.
  - Examples of streams: a file, a string, a network connection.

Processing Postfix: Pseudocode

- **input:** a stream of tokens in infix order.
- **output:** the result of the calculation (a number).
- while(input remains to be processed)

Processing Postfix: Pseudocode

- **input:** a stream of tokens in infix order.
- **output:** the result of the calculation (a number).
- while(input remains to be processed)
  - T = next token (number or operator) from the input
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- final_result = ???

Processing Postfix: Pseudocode

- **input:** a stream of tokens in infix order.
- **output:** the result of the calculation (a number).
- while(input remains to be processed)
  - T = next token (number or operator) from the input
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- final_result = pop(stack)

Example: Postfix Notation Evaluation

- Example: we will see how to use this pseudocode to evaluate this **postfix** expression:
  - 5 9 8 + 4 6 * * 7 + *
- In **infix**, the equivalent expression is:
  - 5 * (((9 + 8) * (4 * 6)) + 7)

Example: Postfix Notation Evaluation

- Input: 5 9 8 + 4 6 * * 7 + *
- while(input remains to be processed)
  - T = next token
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- final_result = pop(stack)

- T = 5

5

Example: Postfix Notation Evaluation

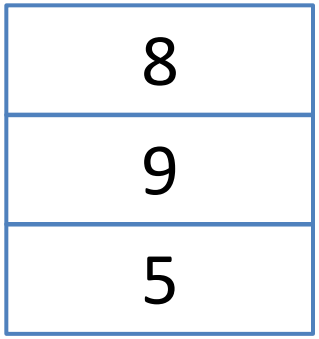
- Input: 5 9 8 + 4 6 * * 7 + *
- while(input remains to be processed)
  - T = next token
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- final_result = pop(stack)

- T = 9

Example: Postfix Notation Evaluation

- Input: 5 9 8 + 4 6 * * 7 + *
- while(input remains to be processed)
  - T = next token
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- final_result = pop(stack)

- T = 8



Example: Postfix Notation Evaluation

- Input: 5 9 8 + 4 6 * * 7 + *
- while(input remains to be processed)
  - T = next token
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- final_result = pop(stack)

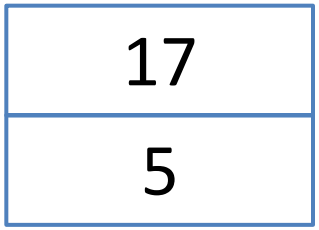
- T = +
- A = 8
- B = 9

5

Example: Postfix Notation Evaluation

- Input: 5 9 8 + 4 6 * * 7 + *
- while(input remains to be processed)
  - T = next token
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- final_result = pop(stack)

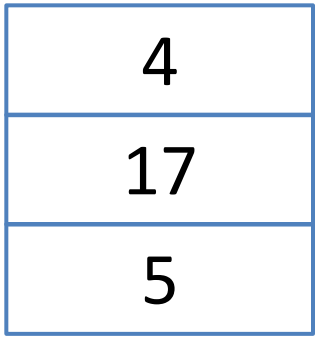
- T = +
- A = 8
- B = 9
- C = 17



Example: Postfix Notation Evaluation

- Input: 5 9 8 + 4 6 * * 7 + *
- while(input remains to be processed)
  - T = next token
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- final_result = pop(stack)

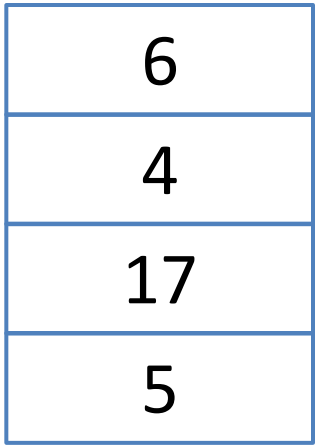
- T = 4



Example: Postfix Notation Evaluation

- Input: 5 9 8 + 4 6 * * 7 + *
- while(input remains to be processed)
  - T = next token
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- final_result = pop(stack)

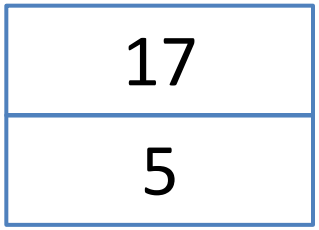
- T = 6



Example: Postfix Notation Evaluation

- Input: 5 9 8 + 4 6 * * 7 + *
- while(input remains to be processed)
  - T = next token
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- final_result = pop(stack)

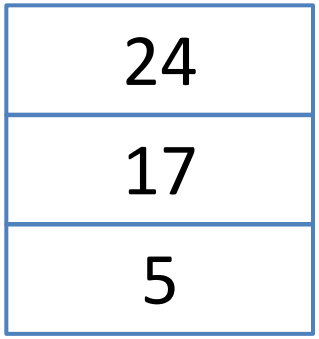
- T = *
- A = 6
- B = 4



Example: Postfix Notation Evaluation

- Input: 5 9 8 + 4 6 * * 7 + *
- while(input remains to be processed)
  - T = next token
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- final_result = pop(stack)

- T = *
- A = 6
- B = 4
- C = 24



Example: Postfix Notation Evaluation

- Input: 5 9 8 + 4 6 * * 7 + *
- while(input remains to be processed)
  - T = next token
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- final_result = pop(stack)

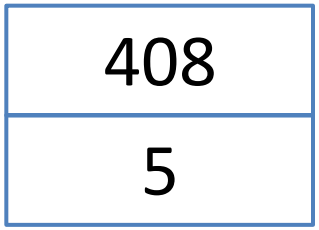
- T = *
- A = 24
- B = 17

5

Example: Postfix Notation Evaluation

- Input: 5 9 8 + 4 6 * * 7 + *
- while(input remains to be processed)
  - T = next token
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- final_result = pop(stack)

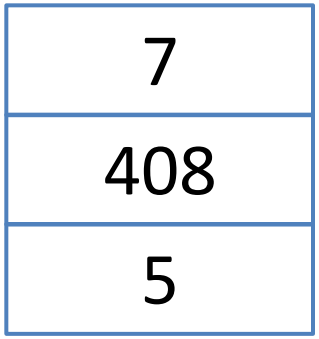
- T = *
- A = 24
- B = 17
- C = 408



Example: Postfix Notation Evaluation

- Input: 5 9 8 + 4 6 * * 7 + *
- while(input remains to be processed)
  - T = next token
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- final_result = pop(stack)

- T = 7



Example: Postfix Notation Evaluation

- Input: 5 9 8 + 4 6 * * 7 + *
- while(input remains to be processed)
  - T = next token
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- final_result = pop(stack)

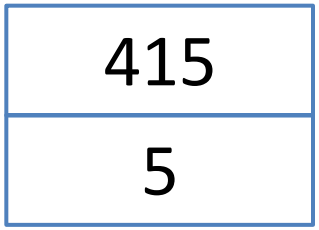
- T = +
- A = 7
- B = 408

5

Example: Postfix Notation Evaluation

- Input: 5 9 8 + 4 6 * * 7 + *
- while(input remains to be processed)
  - T = next token
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- final_result = pop(stack)

- T = +
- A = 7
- B = 408
- C = 415



Example: Postfix Notation Evaluation

- Input: 5 9 8 + 4 6 * * 7 + *
- while(input remains to be processed)
  - T = next token
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- final_result = pop(stack)

- T = *
- A = 415
- B = 5

Example: Postfix Notation Evaluation

- Input: 5 9 8 + 4 6 * * 7 + *
- while(input remains to be processed)
  - T = next token
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- final_result = pop(stack)

- T = *
- A = 415
- B = 5
- C = 2075

2075

Example: Postfix Notation Evaluation

- Input: 5 9 8 + 4 6 * * 7 + *
- while(input remains to be processed)
  - T = next token
  - If T is a number, push(stack, T).
  - If T is an operator:
    - A = pop(stack)
    - B = pop(stack)
    - C = apply operator T on A and B.
    - push(stack, C)
- **final_result = pop(stack)**

- T = *
- A = 415
- B = 5
- C = 2075

final_result = 2075

Converting Infix to Postfix

- Another example of using stacks is converting infix notation to postfix notation.
- We already saw how to evaluate postfix expressions.
- By converting infix to postfix, we will be able to evaluate infix expressions as well.
- **input:** a stream of tokens in infix order.
- **output**: a list of tokens in postfix order.

Converting Infix to Postfix

- **input:** a stream of tokens in infix order.
  - Assumption 1: the input is fully parenthesized. That is, every operation (that contains an operator and its two operands) is enclosed in parentheses.
    - 3 + 5 NOT ALLOWED.
    - (3 + 5) ALLOWED.
  - Assumption 2: Each operator has two operands.
    - (2 + 4 + 5) NOT ALLOWED.
    - (2 + (4 + 5)) ALLOWED.
  - Writing code that does not need these assumptions is great (but optional) practice for you.
- **output**: a list of tokens in postfix order.

Converting Infix to Postfix

- **input:** a stream of tokens in infix order.
- **output**: a list of tokens in postfix order.
- while(the input stream is not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator, push(op_stack, T).
  - If T is a number, insertAtEnd(result, T)

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T =
- op_stack = [ ] empty stack
- result = [ ] (empty list)

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T = (
- op_stack = [ ] empty stack
- result = [ ] (empty list)

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T = 5
- op_stack = [ ] empty stack
- result = [5]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T = *
- op_stack = [*]
- result = [5]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T = (
- op_stack = [*]
- result = [5]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T = (
- op_stack = [*]
- result = [5]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T = (
- op_stack = [*]
- result = [5]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T = 9
- op_stack = [*]
- result = [5 9]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T = +
- op_stack = [* +]
- result = [5 9]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T = 8
- op_stack = [* +]
- result = [5 9 8]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T =)
- op_stack = [*]
- result = [5 9 8 +]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T = *
- op_stack = [* *]
- result = [5 9 8 +]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T = (
- op_stack = [* *]
- result = [5 9 8 +]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T = 4
- op_stack = [* *]
- result = [5 9 8 + 4]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T = *
- op_stack = [* * *]
- result = [5 9 8 + 4]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T =)
- op_stack = [* * *]
- result = [5 9 8 + 4 6]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T =)
- op_stack = [* *]
- result = [5 9 8 + 4 6 *]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T =)
- op_stack = [*]
- result = [5 9 8 + 4 6 * *]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T = +
- op_stack = [* +]
- result = [5 9 8 + 4 6 * *]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T = 7
- op_stack = [* +]
- result = [5 9 8 + 4 6 * * 7]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T =)
- op_stack = [*]
- result = [5 9 8 + 4 6 * * 7 +]

Example: Infix to Postfix

- Input: (5 * (((9 + 8) * (4 * 6)) + 7))
- while(input stream not empty)
  - T = next token
  - If T is left parenthesis, ignore.
  - If T is right parenthesis:
    - op = pop(op_stack)
    - insertAtEnd(result, op)
  - If T is an operator: push(op_stack, T).
  - If T is a number: insertAtEnd(result, T)

- T =)
- op_stack = [ ]
- result = [5 9 8 + 4 6 * * 7 + *]

The Stack Interface (Textbook Version)

- The textbook defines a stack interface that supports four specific functions:
- **void STACKinit(int max_size):**
  - Initialize the stack. Argument **max_size** declares the maximum possible size for the stack.
- **int STACKempty():**
  - Returns 1 if the stack is empty, 0 otherwise.
- **void STACKpush(Item item):**
  - Pushes the item on top of the stack.
- **Item STACKpop():**
  - Removes from the stack the item that was on top, and returns that item.

Problems With Textbook Interface?

Problems With Textbook Interface?

- These functions do not refer to any specific stack object.
- What is the consequence of that?

Problems With Textbook Interface?

- These functions do not refer to any specific stack object.
- What is the consequence of that?
- This interface can only support a single stack. If we need to use simultaneously two or more stacks, we need to extend the interface.

Problems With Textbook Interface?

- Suppose that we fix the first problem, and we can have multiple stacks.
- Can we have a stack A that contains integers, and a stack B that contains strings?

Problems With Textbook Interface?

- Suppose that we fix the first problem, and we can have multiple stacks.
- Can we have a stack A that contains integers, and a stack B that contains strings?
- No. Each stack contains values of type **Item**.
- While **Item** can be set (using **typedef**) to any type we want, all stacks in the code must contain values of that one type.
- We have the exact same problem with **lists**.
- Let's see how to solve this problem, for both lists and stacks.

Links Revisited

old definition

```
struct node
{
  Item item;
  link next;
};
typedef struct node * link;
```

new definition

```
struct node
{
  void * item;
  link next;
};
typedef struct node * link;
```

- **void ***:

Links Revisited

old definition

```
struct node
{
  Item item;
  link next;
};
typedef struct node * link;
```

new definition

```
struct node
{
  void * item;
  link next;
};
typedef struct node * link;
```

- **void ***: this is a special type in C.
  - It means **pointer to anything.**
- If item is of type void*, it can be set equal to a pointer to any object:
  - int *, char *, double *, pointers to structures, …

Example: Adding an **int** to a List

- To insert integer 7 to a list **my_list**:

```
struct node
{   void * item;
    link next;
};

link newLink(void * content)
{  link result = malloc(sizeof(struct node));
   result->item = content;
   result->next = NULL;
}
```

Example: Adding an **int** to a List

- To insert integer 7 to a list **my_list**:
  - Allocate memory for a pointer to an int.
  - Set the contents of the pointer equal to 7.
  - Add to the list a new link, whose item is the pointer.

```
struct node
{   void * item;
    link next;
};

link newLink(void * content)
{  link result = malloc(sizeof(struct node));
   result->item = content;
   result->next = NULL;
}
```

Example: Adding an **int** to a List

- To insert integer 7 to a list **my_list**:
  - Allocate memory for a pointer to an int.
  - Set the contents of the pointer equal to 7.
  - Add to the list a new link, whose item is the pointer.

```
struct node
{   void * item;
    link next;
};

link newLink(void * content)
{  link result = malloc(sizeof(struct node));
   result->item = content;
   result->next = NULL;
}

int * content = malloc(sizeof(int));
*content = 7;
insertAtEnd(my_list, newLink(content));
```

Example: Adding an **int** to a List

- Note: instead of **insertAtEnd**, we could have used any other insertion function.

```
struct node
{   void * item;
    link next;
};

link newLink(void * content)
{  link result = malloc(sizeof(struct node));
   result->item = content;
   result->next = NULL;
}

int * content = malloc(sizeof(int));
*content = 7;
insertAtEnd(my_list, newLink(content));
```

Example: Accessing an **int** in a Link

```
struct node
{   void * item;
    link next;
};

link n = … // put any code here that produces a link.
```

- To access the value of an int stored in link **n**:

Example: Accessing an **int** in a Link

```
struct node
{   void * item;
    link next;
};

link n = … // put any code here that produces a link.
```

- To access the value of an int stored in link **n**:
  - Call **linkItem** to get the item stored at **n**.
  - Store the result of **linkItem** to a variable of type int* (use casting).
  - Dereference that pointer to get the number.

Example: Accessing an **int** in a Link

```
struct node
{   void * item;
    link next;
};

link n = … // put any code here that produces a link.
int * content = (int *) linkItem(n);     // note the casting.
my_number = *content;      // pointer dereferencing.
```

- To access the value of an int stored in link **n**:
  - Call **linkItem** to get the item stored at **n**.
  - Store the result of **linkItem** to a variable of type int* (use casting).
  - Dereference that pointer to get the number.

Example: Removing an **int** from a List

- To remove a link containing an int:

```
struct node
{   void * item;
    link next;
};

void * linkItem(link the_link)
{
   return the_link->item;
}
```

Example: Removing an **int** from a List

- To remove a link containing an int:
  - Remove the link from the list.
  - Get the pointer that is stored as the link's item.
  - **Dereference** the pointer to get the int.
  - Free the link (this we were doing before, too).
  - Free the int * pointer (this we didn't have to do before).

```
struct node
{   void * item;
    link next;
};

void * linkItem(link the_link)
{
   return the_link->item;
}
```

Example: Removing an **int** from a List

- To remove a link containing an int:
  - Remove the link from the list.
  - Get the pointer that is stored as the link's item.
  - **Dereference** the pointer to get the int.
  - Free the link (this we were doing before, too).
  - Free the int * pointer (this we didn't have to do before).

```
struct node
{   void * item;
    link next;
};

void * linkItem(link the_link)
{
   return the_link->item;
}

link a = deleteAtBeginning(my_list);
int * content = (int *) linkItem(a);
int my_number = *content;
free(a); free(content);
```

Example: Removing an **int** from a List

- Note: instead of **deleteAtBeginning** we could have used any other function that deletes links from a list.

```
struct node
{   void * item;
    link next;
};

void * linkItem(link the_link)
{
   return the_link->item;
}

link a = deleteAtBeginning(my_list);
int * content = (int *) linkItem(a);
int my_number = *content;
free(a); free(content);
```

Storing Objects of Any Type in a List

- The previous examples can be adapted to accommodate objects of any other type that we want to store in a list.

- To store an object **X** of type **T** in a link **L**:

```
T* P = malloc(sizeof(T));
*P = X;
link L = newLink(P);
```

- To access an object **X** of type **T** stored in a link **L**:

```
T* P = linkItem(L);
T value = *P;
```

The New List Interface

- See files **lists.h** and **lists.c** posted on the course website.

- NOTE: the new interface allows us to store objects of different types even **on the same list**.
- Also, the new implementation makes it efficient to insert at the end of the list, which will be useful later (in FIFO queues).

Implementing Stacks

- A stack can be implemented using either lists or arrays.
- Both implementations are fairly straightforward.
- List-based implementation:
  - What is a stack?
  - **push(stack, item)** is implemented how?
  - **pop(stack)** is implemented how?

Implementing Stacks

- A stack can be implemented using either lists or arrays.
- Both implementations are fairly straightforward.
- List-based implementation:
  - What is a stack? A stack is essentially a list.
  - **push(stack, item)** inserts that item at the beginning of the list.
  - **pop(stack)** removes (and returns) the item at the beginning of the list.
  - What is the time complexity of these operations?

Implementing Stacks

- A stack can be implemented using either lists or arrays.
- Both implementations are fairly straightforward.
- List-based implementation:
  - What is a stack? A stack is essentially a list.
  - **push(stack, item)** inserts that item at the beginning of the list.
  - **pop(stack)** removes (and returns) the item at the beginning of the list.
  - Both operations take O(1) time.
- See **stacks_lists.c** on course website.

Implementation Code

- See files posted on course website:
  - **stacks.h:** defines the public interface.
  - **stacks_lists.c:** defines stacks using lists.
  - **stacks_arrays.c:** defines stacks using arrays.

The Stack Interface

- See file **stacks.h** posted on the course website.

```
??? newStack(???);
??? destroyStack(???);
??? push(Stack stack, void * content);
??? * pop(Stack stack);
??? stackEmpty(Stack stack);

??? popInt(???);
??? pushInt(???);
??? printIntStack(???);
```

The Stack Interface

- See file **stacks.h** posted on the course website.

```
Stack newStack();
void destroyStack(Stack stack);
void push(Stack stack, void * content);
void * pop(Stack stack);
int stackEmpty(Stack stack);

int popInt(Stack stack);
void pushInt(Stack stack, int value);
void printIntStack(Stack stack);
```

Defining Stacks

```
typedef struct stack_struct * Stack;

struct stack_struct
{
    ???;
};
```

Defining Stacks

```
typedef struct stack_struct * Stack;

struct stack_struct
{
    list items;
};
```

Creating a New Stack

```
typedef struct stack_struct * Stack;
struct stack_struct
{
   list items;
};

Stack newStack()
{
   ???
}
```

Creating a New Stack

```
typedef struct stack_struct * Stack;
struct stack_struct
{
    list items;
};

Stack newStack()
{
    Stack result = malloc(sizeof(*result));
    result->items = newList();
    return result;
}
```

Destroying a Stack

```
typedef struct stack_struct * Stack;
struct stack_struct
{
   list items;
};

void destroyStack(Stack stack)
{
   ???
}
```

Destroying a Stack

```
typedef struct stack_struct * Stack;
struct stack_struct
{
    list items;
};

void destroyStack(Stack stack)
{
    destroyList(stack->items);
    free(stack);
}
```

Pushing an Item

```
typedef struct stack_struct * Stack;
struct stack_struct
{
   list items;
};

void push(Stack stack, void * content)
{
   ???
}
```

Pushing an Item

```
typedef struct stack_struct * Stack;
struct stack_struct
{
   list items;
};

void push(Stack stack, void * content)
{
   link L = newLink(content);
   insertAtBeginning(stack->items, L);
}
```

Popping an Item

```
typedef struct stack_struct * Stack;
struct stack_struct
{
   list items;
};

void * pop(Stack stack)
{
   ???
}
```

Popping an Item

```
typedef struct stack_struct * Stack;
struct stack_struct
{
    list items;
};

void * pop(Stack stack)
{
    link top = deleteAtBeginning(stack->items);
    return linkItem(top);
}
```

What is wrong with this definition of **pop**?

Popping an Item

```
typedef struct stack_struct * Stack;
struct stack_struct
{
    list items;
};

void * pop(Stack stack)
{
    link top = deleteAtBeginning(stack->items);
    return linkItem(top);
}
```

What is wrong with this definition of **pop**? Memory leak!!!

Popping an Item

```
typedef struct stack_struct * Stack;
struct stack_struct
{
   list items;
};

void * pop(Stack stack)
{
   if (stackEmpty(stack))
      ERROR!!!
   link top = deleteAtBeginning(stack->items);
   void * item = linkItem(top);
   free(top);
   return item;
}
```

Pushing an Int

- This is an example of a convenience function.
- If we use stacks of ints (or any other type) a lot, it makes sense to write functions that simplify dealing with such stacks.

```
void pushInt(Stack stack, int value)
{
    ???
}
```

Pushing an Int

- This is an example of a convenience function.
- If we use stacks of ints (or any other type) a lot, it makes sense to write functions that simplify dealing with such stacks.

```
void pushInt(Stack stack, int value)
{
    int * content = malloc(sizeof(int));
    *content = value;
    push(stack, content);
}
```

Popping an Int

- This is another example of a convenience function, that simplifies dealing with stacks of ints.
- Similar functions can be written as needed, to support stacks of other types.

```
int popInt(Stack stack)
{
    ???
}
```

Popping an Int

- This is another example of a convenience function, that simplifies dealing with stacks of ints.
- Similar functions can be written as needed, to support stacks of other types.

```
int popInt(Stack stack)
{
    int * top = (int *) pop(stack);
    return *top;
}
```

What is wrong with this definition of **popInt**?

Popping an Int

- This is another example of a convenience function, that simplifies dealing with stacks of ints.
- Similar functions can be written as needed, to support stacks of other types.

```
int popInt(Stack stack)
{
    int * top = (int *) pop(stack);
    return *top;
}
```

What is wrong with this definition of **popInt**? Memory leak!!!

Popping an Int

- This is another example of a convenience function, that simplifies dealing with stacks of ints.
- Similar functions can be written as needed, to support stacks of other types.

```
int popInt(Stack stack)
{
    int * top = (int *) pop(stack);
    int result = *top;
    free(top);
    return result;
}
```

Array-Based Implementation

- A stack can also be implemented using arrays.
- **push(stack, item):** ???
- **pop(stack):** ???

Array-Based Implementation

- A stack can also be implemented using arrays.
- A stack is essentially an array.
- **push(stack, item)** inserts that item at the end of the array.
- **pop(stack)** removes (and returns) the item at the end of the array.
- What is the time complexity of these two operations?

Array-Based Implementation

- A stack can also be implemented using arrays.
- A stack is essentially an array.
- **push(stack, item)** inserts that item at the end of the array.
- **pop(stack)** removes (and returns) the item at the end of the array.
- Both operations take O(1) time.
- See **stacks_arrays.c** on course website.

Defining Stacks Using Arrays

```
typedef struct stack_struct * Stack;

struct stack_struct
{
    ???
};
```

Defining Stacks Using Arrays

```
typedef struct stack_struct * Stack;

struct stack_struct
{
    int max_size;
    int top_index;
    void ** items;
};
```

Creating a New Stack

```
struct stack_struct
{   int max_size;
    int top_index;
    void ** items;
};

Stack newStack(???)
{
    ???
}
```

Creating a New Stack

```
struct stack_struct
{   int max_size;
    int top_index;
    void ** items;
};

Stack newStack1(int max_size)
{
    Stack result = malloc(sizeof(*result));
    result->items = malloc(max_size * sizeof(void*));
    result->max_size = max_size;
    result->top_index = -1;
    return result;
}
```

Destroying a Stack

```
struct stack_struct
{   int max_size;
    int top_index;
    void ** items;
};

void destroyStack(Stack stack)
{
    ???
}
```

Destroying a Stack

```
struct stack_struct
{   int max_size;
    int top_index;
    void ** items;
};

void destroyStack(Stack stack)
{
    int i;
    for (i = 0; i <= stack->top_index; i++)
        free(stack->items[i]);
    free(stack->items);
    free(stack);
}
```

Pushing an Item

```
struct stack_struct
{   int max_size;
    int top_index;
    void ** items;
};

void push(Stack stack, ???)
{
    ???
}
```

Pushing an Item

```
struct stack_struct
{  int max_size;
   int top_index;
   void ** items;
};

void push(Stack stack, void * content)
{
   if (stack->top_index == stack->max_size - 1)
      ERROR!!!

   stack->top_index++;
   stack->items[stack->top_index] = content;
}
```

Popping an Item

```
struct stack_struct
{   int max_size;
    int top_index;
    void ** items;
};

??? pop(Stack stack)
{
    ???
}
```

Popping an Item

```
struct stack_struct
{  int max_size;
   int top_index;
   void ** items;
};

void * pop(Stack stack)
{
   if (stackEmpty(stack))
      ERROR!!!
   void * item = stack->items[stack->top_index];
   stack->top_index--;
   return item;
}
```