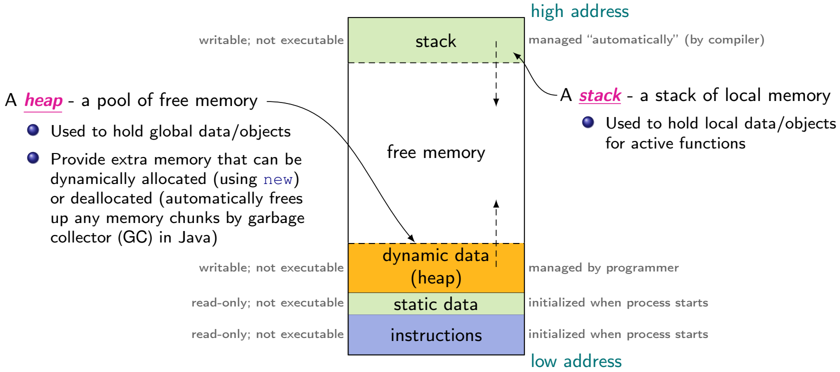
Memory model

What are pointers/references?

- A basic data type
- An address or a chunk of memory where data can be stored



A ***heap*** - a pool of free memory

- Used to hold global data/objects
- Provide extra memory that can be dynamically allocated (using `new`) or deallocated (automatically frees up any memory chunks by garbage collector (GC) in Java)

A ***stack*** - a stack of local memory

- Used to hold local data/objects for active functions

General memory layout

Note: Copying the address does not copy the chunk of memory

However, how does the stack (and other memory in computer) actually operate?

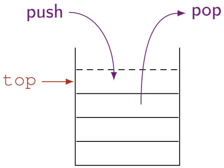
CPT108 Data Structures and Algorithms

Lecture 15

Stack and Queue

Stack

- An abstract data type (ADT)
- A stack is a *list* in which *insertion* (push) and *deletion* (pop) take place at the same end
- Also known as last-in, first-out (LIFO) lists
  - The last element will be the first to be retrieved from the list
- A *pointer*, `top`, is used to point to the top element of the stack

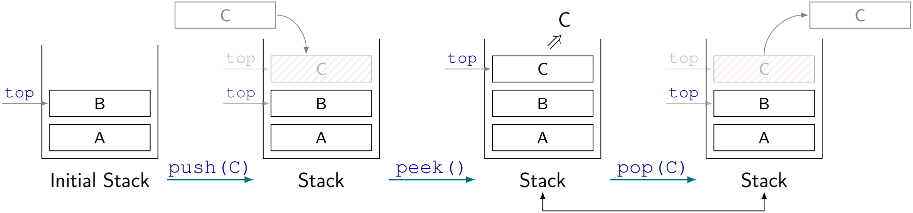


Stack

Stack

- Desirable stack operations;
  - Push a node on top of the stack (*insertion*)
  - Top (or peek) the top node of stack
  - Pop and return the top node on stack (*deletion*)

  - Size of the stack
  - Whether stack is empty
  - Whether stack is full (if stack is bounded)



Notice the different between the two!

Stack: Implementation – Interface

```
public interface Stack {
  int push(Node node);
  Node pop();
  Node peek();
  boolean isEmpty();
  boolean isFull();
  int size();
  boolean contains(Node node);
  void clear();
}
```

Stack

Applications

- Storage of function calls information
  (recall the “factorial” example in recursion lecture)
- The “Undo” command that discards the last changes to the file in most applications.

Stack

Applications (cont.)

Symbols matching

- Check the corresponding opening and closing symbols of:
    - Mathematical expression, such as:
        - Parentheses: “(” and “)”
        - Braces “{” and “}”
        - Brackets: “[” and “]”
    - HTML and XML tags:
        - `<tag_name>` and `</tag_name>`

```
1 Create an empty stack
2 Read in the next token until end of file
    1 If the token is an opening symbol, push it onto the stack
    2 If the token is a closing symbol:
        1 Report an error if stack is empty
        2 Pop the stack.  Report an error if the popped is not the corresponding
          opening symbol.
3 At end of file, report an error if the stack is not empty
```

Stack

Applications (cont.)

Postfix expression evaluation

- Expressions that have the operator put after the operands
- Used in some calculators and in compilers
- E.g.

Infix form	Postfix form
5 * 6	5 6 *
5 * (6 + 1)	5 6 1 + *
(5 * 6) - 10	5 6 * 10 -
4 + ((5 * 6) / 3)	4 5 6 * 3 / +

```
1 Create an empty stack
2 Read in the next token from the expression
3 While the next token is not empty, do
    (a) If the token is an integer, push its value onto the stack;
    (b) Otherwise // the token is an operator
        (i) Pop the top two elements out of stack
        (ii) Apply the operator to the values
        (iii) Push the result onto stack
4 Pop the top value out of stack
5 If the stack is not empty, print error message
6 Otherwise, print the value as result
```

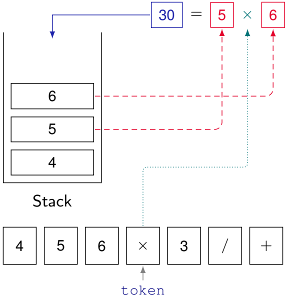
Postfix expression evaluation

- Postfix expression evaluation:

  Evaluate: Infix form : 4 + ((5 × 6) / 3)

  Postfix form: **4 5 6 × 3 / +**

- Read in the next token from the *expression*
- If the token is an integer, push its value onto the stack;
- Otherwise // the token is an operator
  - (i) Pop the top two elements out of stack
  - (ii) Apply the operator to the values
  - (iii) Push the result onto stack
- Pop the top value out of stack



Stack

Applications (cont.)

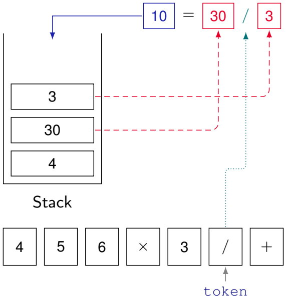
Postfix expression evaluation

- Postfix expression evaluation:

  Evaluate: Infix form : 4 + ((5 × 6) / 3)

  Postfix form: **4 5 6 × 3 / +**

```
• Read in the next token from the
  expression
• If the token is an integer, push its
  value onto the stack;
• Otherwise // the token is an
  operator
    (i)   Pop the top two elements out of
          stack
    (ii)  Apply the operator to the values
    (iii) Push the result onto stack
• Pop the top value out of stack
```



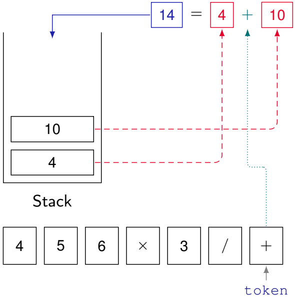
Postfix expression evaluation

- Postfix expression evaluation:

  Evaluate: Infix form : 4 + ((5 × 6) / 3)

  Postfix form: **4 5 6 × 3 / +**

```
- Read in the next token from the expression
- If the token is an integer, push its value onto the stack;
- Otherwise // the token is an operator
    (i) Pop the top two elements out of stack
    (ii) Apply the operator to the values
    (iii) Push the result onto stack
- Pop the top value out of stack
```



Stack

Applications (cont.)

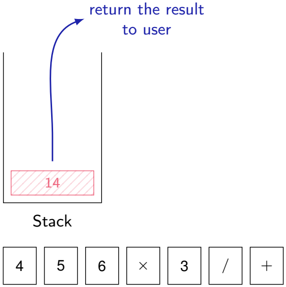
Postfix expression evaluation

- Postfix expression evaluation:

  Evaluate: Infix form : 4 + ((5 × 6) / 3)

  Postfix form: **4 5 6** × **3 / +**

```
• Read in the next token from the
  expression
• If the token is an integer, push its
  value onto the stack;
• Otherwise // the token is an
  operator
    (i)   Pop the top two elements out of
          stack
    (ii)  Apply the operator to the values
    (iii) Push the result onto stack
• Pop the top value out of stack
```



Stack

Implementation

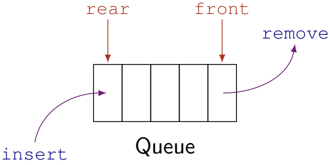
Can be implemented using:

- Array
  - Static: size of the stack is given initially
  - Set `top=0` initially, and increase (decrease) when new element is added to (removed from) the stack
  - Always needs to check for the size of the stack to prevent the array from overflow
- Linked list
  - Dynamic: size of the stack is not bounded
  - Always add/remove element to/from the `head`
- In both implementations, operation can be completed in constant time
  - For array implementation,
    - ★ the operations are performed in very fast constant time
    - ★ however, you may have to resize the array when it is full!
- Leave it to *you* as an exercise!

  (*You can re-use some of the code in the linked list!*)

Queue

- Also an abstract data type (ADT)
- A queue is a *list* in which:
  - *insertion* (enqueue, or offer) is done at one end, and
  - *deletion* (dequeue, or removal) is performed at another end
- Also known as First-in, First-out (FIFO) lists
  - Elements will leave the queue according to the order that they enter the queue.



- Two pointers, `front` and `rear`, are used to point to the *first* and *last* elements of the queue, respectively

Queue (cont.)

Desirable operations

- Insert a node in queue (i.e., queue up)
- Remove a node from queue
- Get the first node in queue
- Get size of the queue (i.e., queue length)
- Whether queue is empty
- Whether queue is full (if queue is bounded)

Queue: Implementation – Interface

```
public interface Queue {

  int insert(Node node);

  Node remove();

  Node peek();

  boolean isEmpty();

  boolean isFull();

  int size();

  boolean contains(Node node);

  void clear();

}
```

Queue: Applications

- Maintaining Queue of Customers or Services
  - Customers queue up at check out counter in supermarket for service
  - Print spooling queue
  - I/O event queue
- Traffic Simulation
  - Buses and private cars queue up at road junction, waiting to enter the highway
- Network Traffic with Bounded Buffer
  - Network messages (like emails) are routing through computers in the network. Each computer allocates fixed amount of memory (i.e., buffer) to hold the messages in transition

Queue: Implementation (cont.)

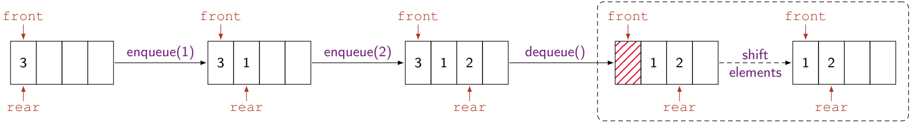
Similar to stack, a queue can be implemented using:

- Linked list
  - ▸ Dynamic: size of the queue is not bounded
  - ▸ Always enqueue at the `rear` and dequeue at the `front`
  - ⇒ Leave it to *you* as an exercise.
    (*You can re-use some of the code in the linked list!*)
- Array
  - ▸ Static: size of the stack is given initially
  - ▸ However, how to implement the enqueue and dequeue operations are a bit tricky

Queue: Implementation (cont.)

Consider the scenario below.

- When enqueuing, the pointer `front` is fixed at index 0 while the pointer `rear` moves forward in the array
- When dequeuing, element at the front of the queue is removed.
  - Therefore, all the elements after it need to be moved by one position.
  - Results in $O(n)$ running time for the `dequeue` method

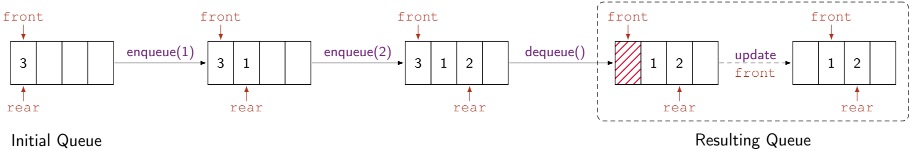


Initial Queue

Resulting Queue

Queue: Implementation (cont.)

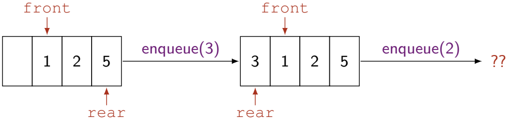
- To resolve the issue,
  - A better way:
    - ★ When enqueue, the pointer `rear` moves forward by one element
    - ★ When dequeue, the pointer `front` also moves forward by one element



- However, the problem here is that the pointer `rear` cannot move beyond the last element in the array

Queue: Implementation (cont.)

- We can improve the situation by using a circular array



Initial Queue

- However, there remain a challenge with the revised approach:
  - ★ How to detect an empty or full queue?
  - ⇒ Use a counter to count the number of elements in the queue.

```
boolean isFull() {
  return (avail+1) % max_size == front ;
}
```

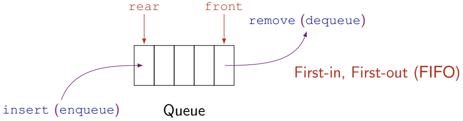
Does the same problem appear in stack?

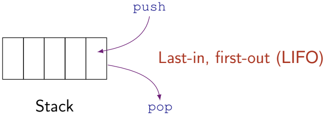
Complexities: Queue vs Stack

- If linked list is used to implement the stack and the queue, we have:

Operation	Stack	Queue
add (insert or push)	$O(1)$	$O(1)$
remove (remove or pop)	$O(1)$	$O(1)$
peek	$O(1)$	$O(1)$
contains	$O(n)$	$O(n)$
size	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
isFull	$O(1)$	$O(1)$

Questions





- Is it possible for us to implement a queue using instances of stack data structure (with `push` and `pop` operations), and operations on them?
- Is it possible for us to implement a stack using instances of queue data structure (with `insert` and `remove` operations), and operations on them?

Reading

- Chapter 10 and p.250, pp. 254-255, 256-257, 449-458, Cormen (2022)