Applications for Linear Data Structures

CMPT 145

Bracket matching

- In a mathematical expression, we use brackets to indicate order of operations.
- Every open bracket must have a close bracket.

  Matched: $(3 + 4) \times 5$

  Unmatched: $(3 + 4 \times 5$

  Unmatched: $3 + 4) \times 5$
- We allow brackets to be nested.

  Matched: $((3 + 4) \times 5)$

Bracket Checking Problem

- Given: A string representing a mathematical expression
- Return: True if the brackets match properly, False otherwise.

  True: $(3 + 4) \times 5$

  True: $((3 + 4) \times 5)$

  False: $(3 + 4 \times 5$

  False: $3 + 4) \times 5$

Bracket Checking Algorithm

- Scan the text from the beginning character by character
  - If the current character is '(' push it on the stack.
  - If the current character is ')':
    - If you can pop the stack, do so. The ')' matches your stored '('.
    - If you cannot pop the stack, the ')' is unmatched.
  - If the current character is anything else, ignore it.
- If you reached the end of the text, and the stack is not empty, you have one or more unmatched '('.

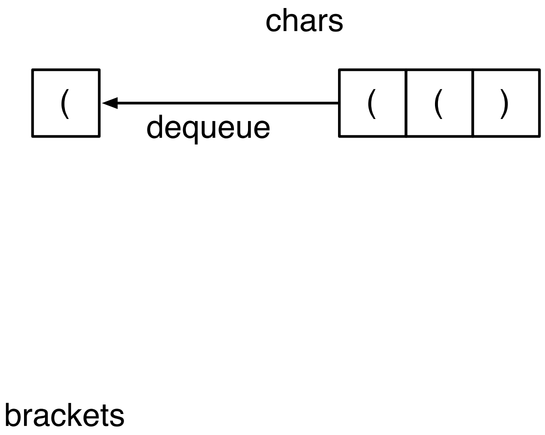
Visualizing the algorithm - Initially

chars

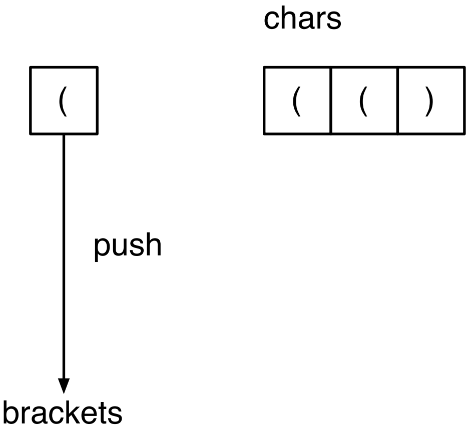
(	(	(	)

brackets

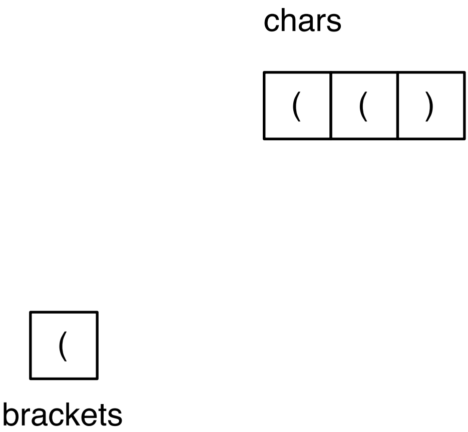
Visualizing the algorithm - First dequeue



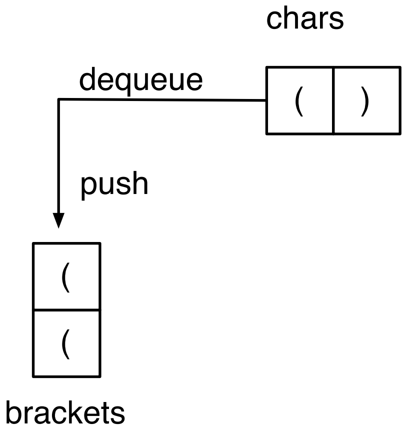
Visualizing the algorithm - First push



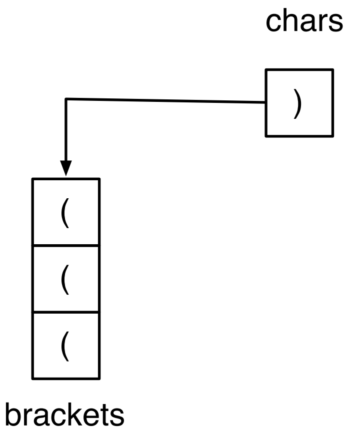
Visualizing the algorithm - After first push



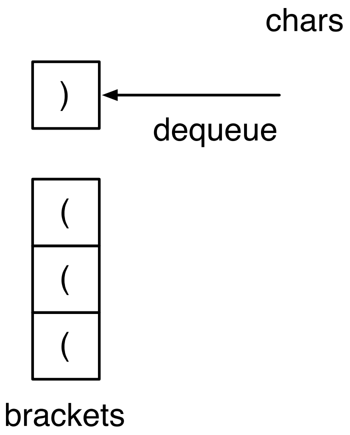
Visualizing the algorithm - Dequeue and push



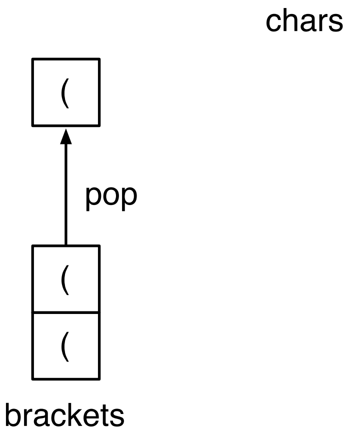
Visualizing the algorithm - Dequeue and push



Visualizing the algorithm - Finding ')'



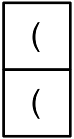
Visualizing the algorithm - Pop



Visualizing the algorithm - Empty queue

chars

(empty)



brackets

Unmatched!

Demo

Thinking about bracket checking

- Why do we use a LIFO stack to store '('?
- Could we use a FIFO queue instead?
- Why do we use a FIFO queue to store ')'?

Doing arithmetic without brackets at all!

- Normally, we write arithmetic expressions like this: $((a + b) \times (c + d)) \times e$.
- We use the brackets to indicate the order of operations.
- We don't need brackets at all, if we use something called *postfix notation*.
- Here's the same expression, using postfix notation.

$$a\ b + c\ d + \times e\ \times$$

- Looks weird, but here's how to read it (left to right):

$$\underbrace{\underbrace{\underbrace{a\ b+}\ \underbrace{c\ d+}\ \times}\ e\ \times}$$

- No brackets needed. Ever.

Post-fix examples

The following expression evaluates to 7:

$$3\ 4\ +$$

The following expression evaluates to 12:

$$3\ 4\ \times$$

The following expression evaluates to 8:

$$12\ 4\ -$$

The following example evaluates to 42:

$$3\ 4\ \times\ 5\ 6\ \times\ +$$

Demo

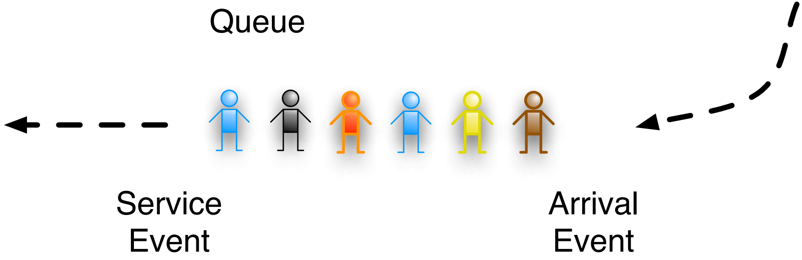
Queueing Simulation

- Assumption: Customers arrive randomly.
- Assumption: Service takes random amount of time.
- Question: How long do customers wait?

Key ideas

- Model the customers' arrival with an average arrival rate (customers per minute).
- Model the service time with an average service rate (customers per minute).
- Keep track of 3 things:
    1. Time of next customer arrival ("arrival event")
    2. Time of next customer service ("service event")
    3. The arrival times of customers who are waiting (queue)
- Time advances to the next event (not by a ticking clock)

Overview



The simulation algorithm

- Schedule the first arrival event
- Schedule the first service event
- Repeat:
    - While an arrival event must happen before a service event:
        - Enqueue the current arrival event
        - Schedule the next arrival event.
    - Handle the service event (e.g., calculate wait time)
    - Schedule the next service event:
        - If there is a customer waiting, start service immediately
        - Otherwise, start after the next customer arrives

Demo

Visualizing the algorithm - Initially

Queue

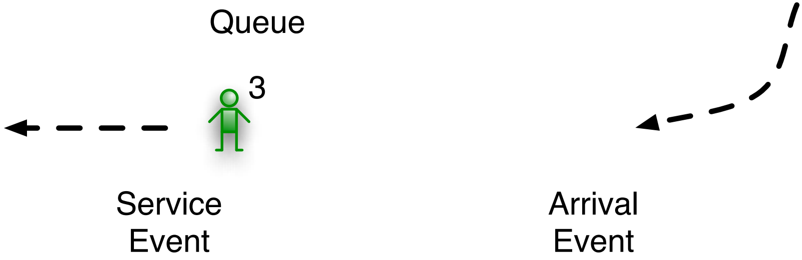
Service Event

Arrival Event

nextService = 3 + 2

nextArrival = 3

First arrival



nextService = 5

nextArrival = 3

Schedule the next arrival

Queue

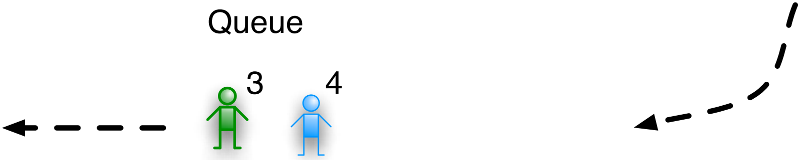
Service Event

Arrival Event

nextService = 5

nextArrival = 3+1

Second arrival



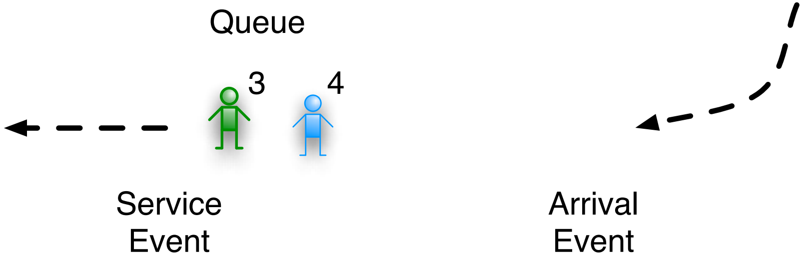
Service Event

Arrival Event

nextService = 5

nextArrival = 4

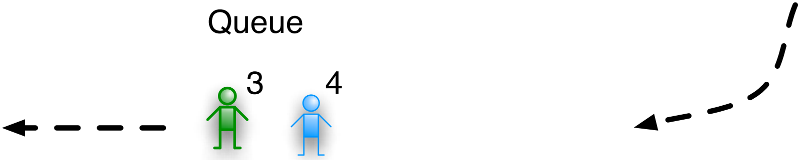
Schedule the next arrival



nextService = 5

nextArrival = 4+2

First service complete



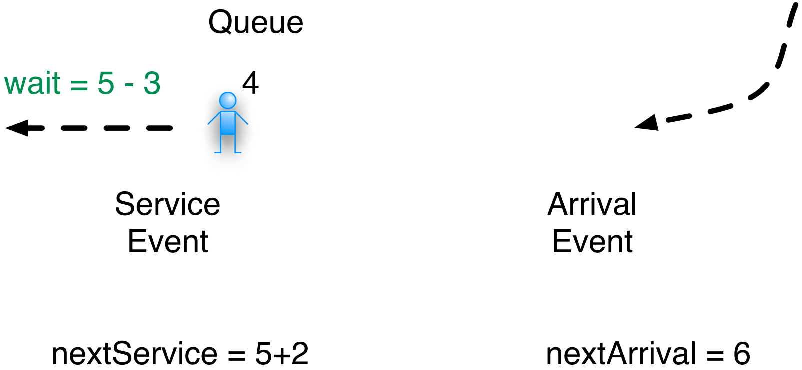
Service Event

Arrival Event

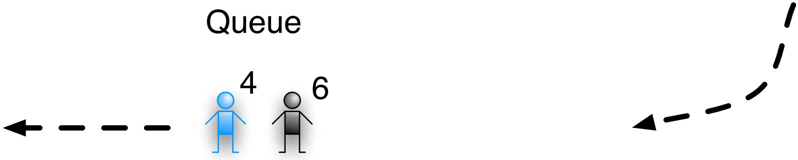
nextService = 5

nextArrival = 6

Schedule the next service



Arrival



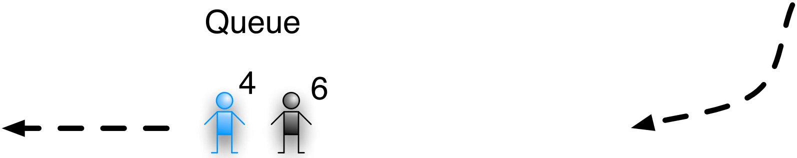
Service Event

Arrival Event

nextService = 7

nextArrival = 6

Schedule the next arrival



Service Event

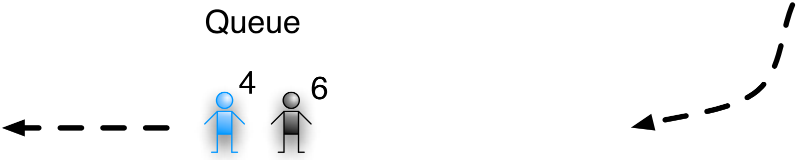
Arrival Event

nextService = 7

nextArrival = 6+1

Service complete

Queue



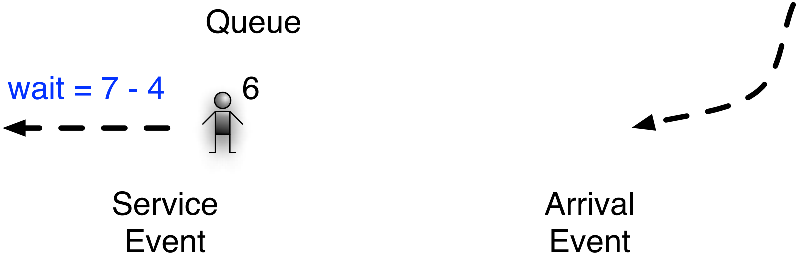
Service
Event

Arrival
Event

nextService = 7

nextArrival = 7

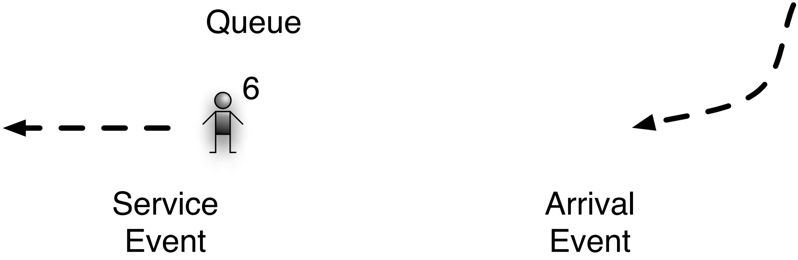
Schedule the next service



nextService = 7 + 5

nextArrival = 7

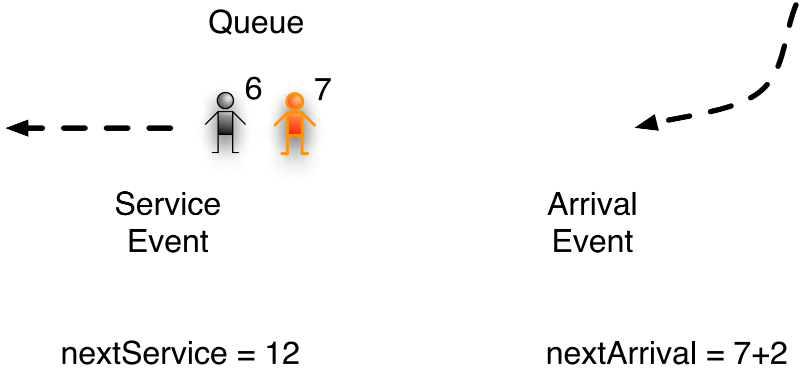
Clock advances to next arrival



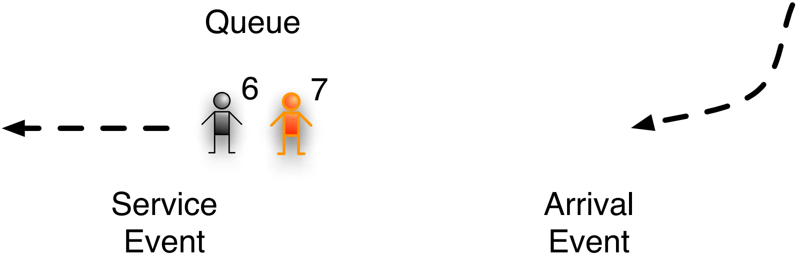
nextService = 12

nextArrival = 7

Arrival and schedule next arrival



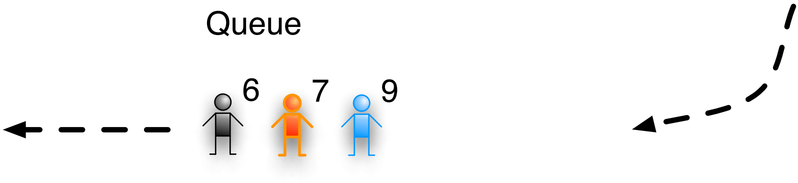
Clock advances to next arrival



nextService = 12

nextArrival = 9

Arrival and schedule next arrival



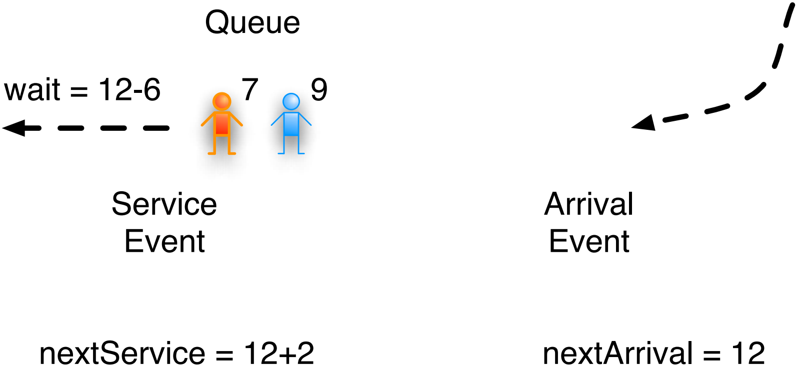
Service Event

Arrival Event

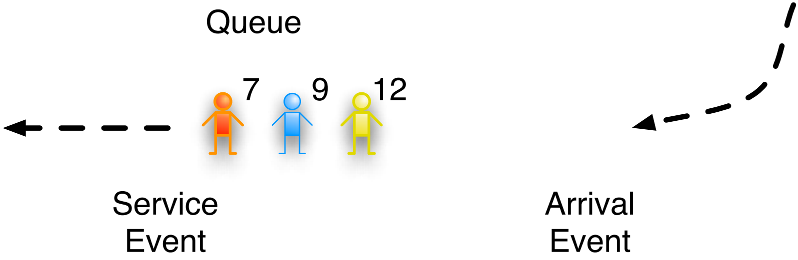
nextService = 12

nextArrival = 9+3

Service complete and schedule next service



Clock advances to next arrival



nextService = 14

nextArrival = 12

Linear ADTs: Queues and Stacks

- Interesting algorithms make use of stacks and queues!
- ADTs provide a useful abstraction to computational concepts
- You could implement all the algorithms without using ADTs, but
  - The ADT helps document the intentions of the program
  - The limited set of operations help prevent errors
  - Resulting code is much clearer