COMP108
Data Structures and Algorithms

Data structures - Queues and Stacks

Professor Prudence Wong

pwong@liverpool.ac.uk

2022-23

Outline

Queues, Stacks

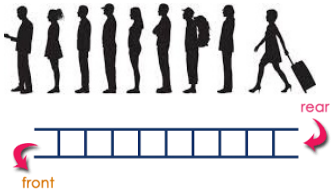
- What are they?
- How to operate data on them?

Learning outcome:

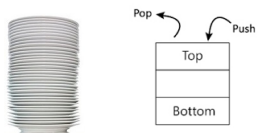
- Understand what queues, stacks are and their associated algorithms

Using arrays to build other data structures

- Arrays allow access to any item with one access
- Sometimes data is accessed in particular order
  - Queue: First-In-First-Out (First-Come-First-Served)

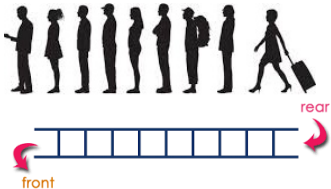


  - Stack: First-In-Last-Out

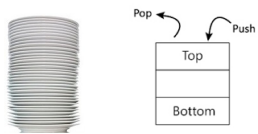


Using arrays to build other data structures

- Arrays allow access to any item with one access
- Sometimes data is accessed in particular order
  - Queue: First-In-First-Out (First-Come-First-Served)



  - Stack: First-In-Last-Out



- *Question: Are orders in restaurants handled by queue or stack?*

Queues . . .

Applications of queues

- Waiting in line for service at a bank

  - Customers arriving get in the end of a line for a teller
  - When a teller is available, customer at the front of the teller is served
  - There may be separate queues for different types of tellers

- Network router (very simplified view)

  - Router can process one packet at a time
  - Packets arriving at router get in a queue
  - Scheduler processes next packet in queue
  - May exercise Active Queue Management to drop packets if full

Queues

Queues: FIFO (first-in-first-out)

- Enqueue: insert element to the tail
- Dequeue: delete element from the head

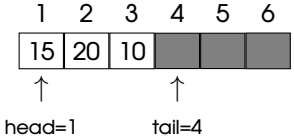
Queues

Queues: FIFO (first-in-first-out)

- Enqueue: insert element to the tail
- Dequeue: delete element from the head

Using an array `Q[ ]`, with two pointer variables **`head`** and **`tail`**

Q contains 3 elements



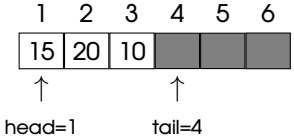
Queues

Queues: FIFO (first-in-first-out)

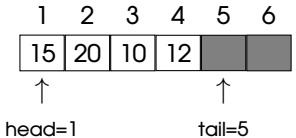
- Enqueue: insert element to the tail
- Dequeue: delete element from the head

Using an array `Q[ ]`, with two pointer variables **`head`** and **`tail`**

Q contains 3 elements



Enqueue(Q, 12)




```
Q[tail] <- 12
tail <- tail+1
```

Queues

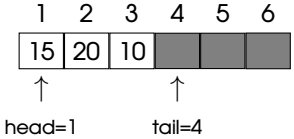
Queues: FIFO (first-in-first-out)

- Enqueue: insert element to the tail
- Dequeue: delete element from the head

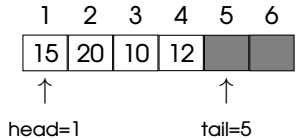
Using an array `Q[ ]`, with two pointer variables **`head`** and **`tail`**

```
return Q[head]
head <- head+1
```

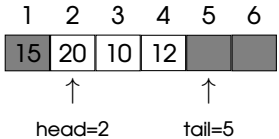
Q contains 3 elements



Enqueue(Q, 12)



Dequeue(Q)



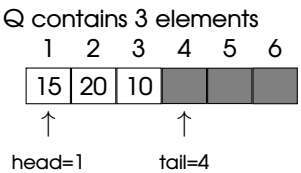
Queues

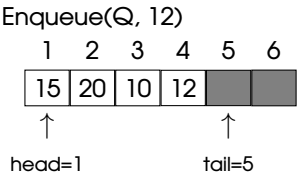
Queues: FIFO (first-in-first-out)

- Enqueue: insert element to the tail
- Dequeue: delete element from the head

Using an array `Q[ ]`, with two pointer variables **`head`** and **`tail`**

- Enqueue: save the data to `Q[tail]` and increment **`tail`** by **`1`**
- Dequeue: retrieve data from `Q[head]` and increment **`head`** by **`1`**
- Note: need to check if queue is empty before Dequeue (full before Enqueue)





Queues - Pseudo code - assuming array of infinite size

- Enqueue(Q, x)
  $Q[tail] \leftarrow x$
  tail $\leftarrow$ tail+1

Queues - Pseudo code - assuming array of infinite size

- Enqueue(Q, x)
  $Q[tail] \leftarrow x$
  tail $\leftarrow$ tail+1

- Dequeue(Q)

  $x \leftarrow Q[head]$
  head $\leftarrow$ head+1
  return x

Queues - Pseudo code - assuming array of infinite size

- Enqueue(Q, x)
  $Q[tail] \leftarrow x$
  tail $\leftarrow$ tail+1

- Dequeue(Q)
  if then
  Queue is EMPTY
  else begin
  x $\leftarrow$ $Q[head]$
  head $\leftarrow$ head+1
  return x
  end

Queues - Pseudo code - assuming array of infinite size

- Enqueue(Q, x)
  $Q[tail] \leftarrow x$
  tail $\leftarrow$ tail+1

- Dequeue(Q)
  if head == tail then
      Queue is EMPTY
  else begin
      x $\leftarrow$ $Q[head]$
      head $\leftarrow$ head+1
      return x
  end

Queues - Pseudo code - what if array is of limited size?

- Let **SIZE** be the size of the array we use.
- Enqueue(Q, x)

```
if              then
      Queue is FULL
else begin
      Q[tail] ← x
      tail ← tail+1
end
```

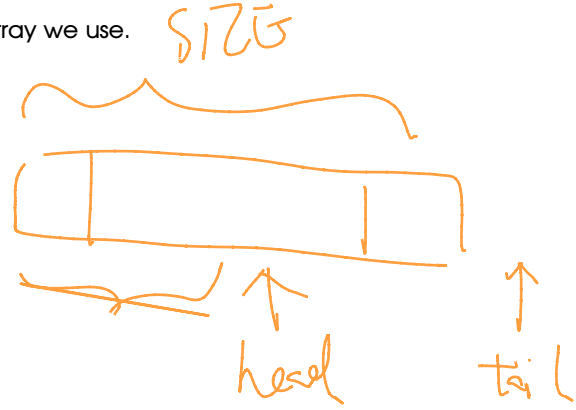
Queues - Pseudo code - what if array is of limited size?

- Let **SIZE** be the size of the array we use.
- Enqueue(Q, x)
  if tail > SIZE then
      Queue is FULL
  else begin
      $Q[tail] \leftarrow x$
      tail $\leftarrow$ tail+1
  end

Queues - Pseudo code - what if array is of limited size?

- Let **`SIZE`** be the size of the array we use.

- Enqueue(Q, x)
  if tail > SIZE then
      Queue is FULL
  else begin
      $Q[tail] \leftarrow x$
      tail $\leftarrow$ tail+1
  end



see **`SampleSimpleQueue.java`** on Canvas

Queues - More implementation details

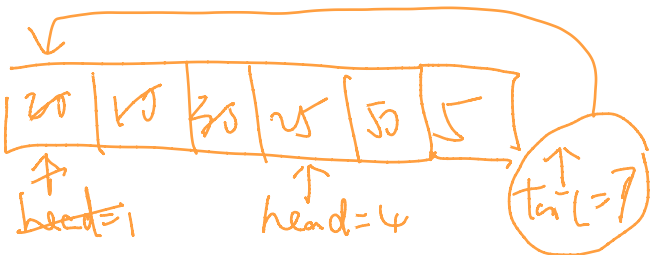
- When elements are dequeue from the queue, its space is wasted.
- Can we reuse those spaces? How?

Queues - More implementation details

- When elements are dequeue from the queue, its space is wasted.
- Can we reuse those spaces? How?
- Wrap around the indices when exceeds the limit.

Queues - More implementation details

- When elements are dequeue from the queue, its space is wasted.
- Can we reuse those spaces? How?
- Wrap around the indices when exceeds the limit.
- Enqueue:
  - after incrementing **`tail`**, if it exceeds **`SIZE`**, then reset it to **`1`**
  - similarly for checking FULL or not
  - if indices are from **`0`** to **`SIZE-1`**, can use **`%`**



Queues - More implementation details

- When elements are dequeue from the queue, its space is wasted.
- Can we reuse those spaces? How?
- Wrap around the indices when exceeds the limit.
- Enqueue:
  - after incrementing **`tail`**, if it exceeds **`SIZE`**, then reset it to **`1`**
  - similarly for checking FULL or not
  - if indices are from **`0`** to **`SIZE-1`**, can use `%`
- Dequeue:
  - after incrementing **`head`**, if it exceeds **`SIZE`**, then reset it to **`1`**
  - if indices are from **`0`** to **`SIZE-1`**, can use `%`

Queues - More implementation details

- When elements are dequeue from the queue, its space is wasted.
- Can we reuse those spaces? How?
- Wrap around the indices when exceeds the limit.
- Enqueue:
  - after incrementing **`tail`**, if it exceeds **`SIZE`**, then reset it to **`1`**
  - similarly for checking FULL or not
  - if indices are from **`0`** to **`SIZE-1`**, can use **`%`**
- Dequeue:
  - after incrementing **`head`**, if it exceeds **`SIZE`**, then reset it to **`1`**
  - if indices are from **`0`** to **`SIZE-1`**, can use **`%`**

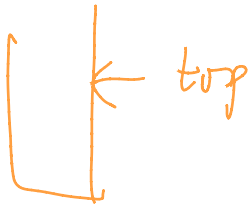
see **`SampleQueue.java`** on Canvas

Stacks . . .

Stacks

Stacks: LIFO (last-in-first-out)

- Push: insert element to the location 1 beyond top
- Pop: delete element from the top



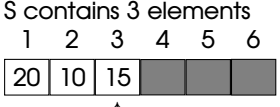
Stacks

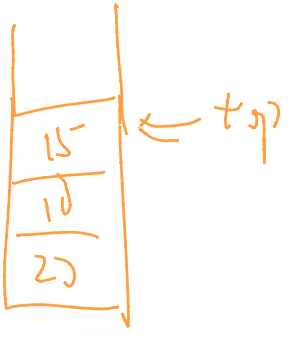
Stacks: LIFO (last-in-first-out)

- Push: insert element to the location 1 beyond top
- Pop: delete element from the top

Using an array **`S[ ]`**, with a pointer variable **`top`**

S contains 3 elements





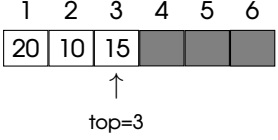
Stacks

Stacks: LIFO (last-in-first-out)

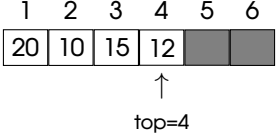
- Push: insert element to the location 1 beyond top
- Pop: delete element from the top

Using an array **`S[ ]`**, with a pointer variable **`top`**

S contains 3 elements



Push(S, 12)



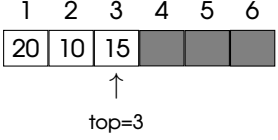
Stacks

Stacks: LIFO (last-in-first-out)

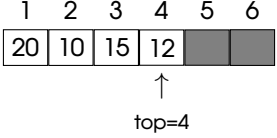
- Push: insert element to the location 1 beyond top
- Pop: delete element from the top

Using an array **`S[ ]`**, with a pointer variable **`top`**

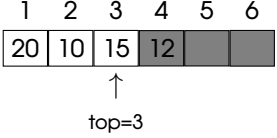
S contains 3 elements



Push(S, 12)



Pop(Q)



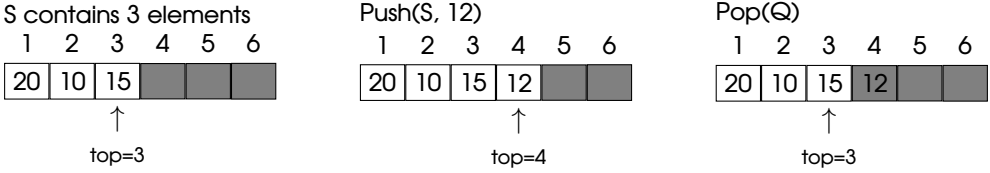
Stacks

Stacks: LIFO (last-in-first-out)

- Push: insert element to the location 1 beyond top
- Pop: delete element from the top

Using an array **`S[ ]`**, with a pointer variable **`top`**

- Push: increment **`top`** by **`1`** and save the data to **`S[top]`**
- Pop: retrieve data from **`S[top]`** and decrement **`top`** by **`1`**
- Note: need to check if stack is empty before Pop



Stacks - Pseudo code - assuming array of infinite size

- Push(S,x)
  top $\leftarrow$ top+1
  $S[top] \leftarrow x$

Stacks - Pseudo code - assuming array of infinite size

- Push(S,x)
  top $\leftarrow$ top+1
  $S[top] \leftarrow x$

- Pop(S)

  $x \leftarrow S[top]$
  top $\leftarrow$ top-1
  return x

Stacks - Pseudo code - assuming array of infinite size

- Push(S,x)
  top $\leftarrow$ top+1
  $S[top] \leftarrow x$

- Pop(S)
  if then
    stack is EMPTY
  else begin
    $x \leftarrow S[top]$
    top $\leftarrow$ top-1
    return x
  end

Stacks - Pseudo code - assuming array of infinite size

- Push(S,x)
  top $\leftarrow$ top+1
  $S[top] \leftarrow x$

- Pop(S)
  if top == 0 then
      stack is EMPTY
  else begin
      $x \leftarrow S[top]$
      top $\leftarrow$ top-1
      return x
  end

Stacks - Pseudo code - what if array is of limited size?

- Let **`SIZE`** be the size of the array we use.
- Push(Q, x)

```
if                then
      Stack is FULL
else begin
      top ← top+1
      S[top] ← x
end
```

Stacks - Pseudo code - what if array is of limited size?

- Let **`SIZE`** be the size of the array we use.

- Push(Q, x)
  if top == SIZE then
      Stack is FULL
  else begin
      top $\leftarrow$ top+1
      $S[top] \leftarrow x$
  end

Stacks - Pseudo code - what if array is of limited size?

- Let **`SIZE`** be the size of the array we use.
- Push(Q, x)
  if top == SIZE then
      Stack is FULL
  else begin
      top $\leftarrow$ top+1
      $S[top] \leftarrow x$
  end
- No need to worry about reusing space

Applications of stacks - Infix vs Postfix representation of mathematical expressions

- Mathematical expressions
  - **Operators:** +, -, *, /, etc.
  - **Operands:** numbers, variables
- Infix: (10 - 3 * 2) * 5
  - operator **in between** operands, **3 * 2** means multiply 3 and 2
  - evaluation needs to look inside parentheses first and follow precedence rules (*,/ before +,-)

Applications of stacks - Infix vs Postfix representation of mathematical expressions

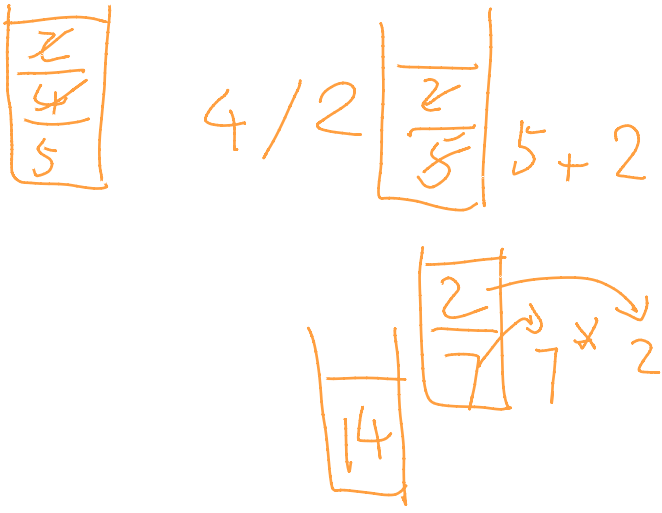
- Mathematical expressions
  - **Operators:** +, -, *, /, etc.
  - **Operands:** numbers, variables

10 6 −

- Infix: (10 - 3 * 2) * 5
  - operator **in between** operands, **3 * 2** means multiply 3 and 2
  - evaluation needs to look inside parentheses first and follow precedence rules (*,/ before +,-)
- Postfix: 10 3 2 * - 5 *
  - operator **after** operands, **3 2 *** means multiply 3 and 2
  - interpret as ((10 (3 2 *) -) 5 *), and equivalent to ((10 - (3 * 2)) * 5)
  - parentheses **not** needed: when we see an operator, the operands are already there to be evaluated
  - precedence is **implicit**
  - ***Easy to process by a program using a stack***

Applications of stacks - Infix vs Postfix representation of mathematical expressions #2

- Examples:
  - Infix: (5 + 4 / 2) * 2
    Postfix: 5 4 2 / + 2 *
  - Infix: (4 - 6 / 3) * 3
    Postfix: 4 6 3 / - 3 *



Applications of stacks - Infix vs Postfix representation of mathematical expressions #2

- Examples:
  - Infix: (5 + 4 / 2) * 2
    Postfix: 5 4 2 / + 2 *
  - Infix: (4 - 6 / 3) * 3
    Postfix: 4 6 3 / - 3 *
- $\exists$ an algorithm using stack to evaluate postfix evaluation.
  $\exists$ an algorithm using stack to convert infix to postfix expression.

Applications of stacks - Evaluating postfix expressions using stack

Algorithm (focusing on binary operators)

1. Read the next token from the expression which may be operator or operand
2. If token is operand, PUSH it to stack
3. If token is operator,
   - 3.1 POP two operands from stack
   - 3.2 operate on the two operands
   - 3.3 PUSH result back to stack
4. If expression is not exhausted, go back Step 1
5. If expression is exhuasted, evaluation finishes and answer is on top of stack

Applications of stacks - Evaluating postfix expressions using stack

Algorithm (focusing on binary operators)

1. Read the next token from the expression which may be operator or operand
2. If token is operand, PUSH it to stack
3. If token is operator,
   - 3.1 POP two operands from stack
   - 3.2 operate on the two operands
   - 3.3 PUSH result back to stack
4. If expression is not exhausted, go back Step 1
5. If expression is exhuasted, evaluation finishes and answer is on top of stack

***Evaluate* 10 3 2 * - 5 ***

Read 10, PUSH(S, 10)
Read 3, PUSH(S, 3)
Read 2, PUSH(S, 2)

Applications of stacks - Evaluating postfix expressions using stack

Algorithm (focusing on binary operators)

1. Read the next token from the expression which may be operator or operand
2. If token is operand, PUSH it to stack
3. If token is operator,
   - 3.1 POP two operands from stack
   - 3.2 operate on the two operands
   - 3.3 PUSH result back to stack
4. If expression is not exhausted, go back Step 1
5. If expression is exhuasted, evaluation finishes and answer is on top of stack

Evaluate **10 3 2 * - 5 ***

Read 10, PUSH(S, 10)
Read 3, PUSH(S, 3)
Read 2, PUSH(S, 2)

Read *,
POP(S) to get 2, POP(S) to get 3
compute 3 * 2 to get 6
PUSH(S, 6)

Applications of stacks - Evaluating postfix expressions using stack

Algorithm (focusing on binary operators)

1. Read the next token from the expression which may be operator or operand
2. If token is operand, PUSH it to stack
3. If token is operator,
   - 3.1 POP two operands from stack
   - 3.2 operate on the two operands
   - 3.3 PUSH result back to stack
4. If expression is not exhausted, go back Step 1
5. If expression is exhuasted, evaluation finishes and answer is on top of stack

***Evaluate* 10 3 2 * - 5 ***

Read 10, PUSH(S, 10)
Read 3, PUSH(S, 3)
Read 2, PUSH(S, 2)
Read *,
  POP(S) to get 2, POP(S) to get 3
  compute 3 * 2 to get 6
  PUSH(S, 6)

Read -,
  POP(S) to get 6, POP(S) to get 10
  compute 10 - 6 to get 4
  PUSH(S, 4)

Applications of stacks - Evaluating postfix expressions using stack

Algorithm (focusing on binary operators)

1. Read the next token from the expression which may be operator or operand
2. If token is operand, PUSH it to stack
3. If token is operator,
   - 3.1 POP two operands from stack
   - 3.2 operate on the two operands
   - 3.3 PUSH result back to stack
4. If expression is not exhausted, go back Step 1
5. If expression is exhuasted, evaluation finishes and answer is on top of stack

***Evaluate* 10 3 2 * - 5 ***

Read 10, PUSH(S, 10)
Read 3, PUSH(S, 3)
Read 2, PUSH(S, 2)
Read *,
  POP(S) to get 2, POP(S) to get 3
  compute 3 * 2 to get 6
  PUSH(S, 6)
Read -,
  POP(S) to get 6, POP(S) to get 10
  compute 10 - 6 to get 4
  PUSH(S, 4)
Read 5, PUSH(S, 5)

Applications of stacks - Evaluating postfix expressions using stack

Algorithm (focusing on binary operators)

1. Read the next token from the expression which may be operator or operand
2. If token is operand, PUSH it to stack
3. If token is operator,
   - 3.1 POP two operands from stack
   - 3.2 operate on the two operands
   - 3.3 PUSH result back to stack
4. If expression is not exhausted, go back Step 1
5. If expression is exhuasted, evaluation finishes and answer is on top of stack

***Evaluate* 10 3 2 * - 5 ***

Read 10, PUSH(S, 10)
Read 3, PUSH(S, 3)
Read 2, PUSH(S, 2)
Read *,
  POP(S) to get 2, POP(S) to get 3
  compute 3 * 2 to get 6
  PUSH(S, 6)
Read -,
  POP(S) to get 6, POP(S) to get 10
  compute 10 - 6 to get 4
  PUSH(S, 4)
Read 5, PUSH(S, 5)
Read *,
  POP(S) to get 5, POP(S) to get 4
  compute 4 * 5 to get 20
  PUSH(S, 20)

Applications of stacks - Evaluating postfix expressions using stack

Algorithm (focusing on binary operators)

1. Read the next token from the expression which may be operator or operand
2. If token is operand, PUSH it to stack
3. If token is operator,
   - 3.1 POP two operands from stack
   - 3.2 operate on the two operands
   - 3.3 PUSH result back to stack
4. If expression is not exhausted, go back Step 1
5. If expression is exhuasted, evaluation finishes and answer is on top of stack

***Evaluate* 10 3 2 * - 5 ***

Read 10, PUSH(S, 10)
Read 3, PUSH(S, 3)
Read 2, PUSH(S, 2)
Read *,
  POP(S) to get 2, POP(S) to get 3
  compute 3 * 2 to get 6
  PUSH(S, 6)
Read -,
  POP(S) to get 6, POP(S) to get 10
  compute 10 - 6 to get 4
  PUSH(S, 4)
Read 5, PUSH(S, 5)
Read *,
  POP(S) to get 5, POP(S) to get 4
  compute 4 * 5 to get 20
  PUSH(S, 20)
End, POP(S) to get 20

Converting infix to postfix expressions using stack (SELF STUDY)

Algorithm (focusing on binary operators)

* Whenever an operator is pop, OUTPUT it

1. Read the next token from the expression which may be operator, operand, (, or)
2. If token is operand, OUTPUT it
3. If token is (, PUSH it on stack
4. If token is), POP from stack until matching (
5. If token is operator, POP from stack until seeing one of the followings
   - 5.1 an operator with same/lower precedence
   - 5.2 opening (
   - 5.3 stack empty

   PUSH the operator on stack
6. If expression is not exhausted, go back Step 1
7. If expression is exhausted, POP rest of stack

***Convert* (10 - 3 * 2) * 5**

Read (, PUSH(S, ``('')
Read 10, OUTPUT 10
Read -, PUSH(S, ``-'')
Read 3, OUTPUT 3
Read *, PUSH(S, ``*'')
Read 2, OUTPUT 2
Read),
  POP(S) and OUTPUT *
  POP(S) and OUTPUT -
  POP(S) to get matching (
Read *, PUSH(S, ``*'')
Read 5, OUTPUT 5
End, POP rest of S, OUTPUT *
OUTPUT: 10 3 2 * - 5 *

***Compare with* (10 * 3 - 2) * 5**

OUTPUT: ~~10 3 * 2 - 5 *~~

10 3 2 * - 5 -

Summary: Queues & Stacks

Next: Linked Lists

For note taking