CSc 120
Introduction to Computer Programming II

15: Stacks and Queues
linear data structures
Linear data structures

A linear data structure is a collection of objects with a straight-line ordering among them

- each object in the collection has a position
- for each object in the collection, there is a notion of the object before it or after it
Data structures we've seen

<table>
<thead>
<tr>
<th>Linear</th>
<th>Not linear</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Python lists (aka arrays)</td>
<td>• Dictionaries</td>
</tr>
<tr>
<td>• Linked lists</td>
<td>• Sets</td>
</tr>
<tr>
<td></td>
<td>• Trees</td>
</tr>
</tbody>
</table>
Today's topic

<table>
<thead>
<tr>
<th>Linear</th>
<th>Not linear</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Python lists (aka arrays)</td>
<td>• Dictionaries</td>
</tr>
<tr>
<td>• Linked lists</td>
<td>• Sets</td>
</tr>
<tr>
<td>• Stacks</td>
<td>• Trees</td>
</tr>
<tr>
<td>• Queues</td>
<td></td>
</tr>
<tr>
<td>• Dequeues</td>
<td></td>
</tr>
</tbody>
</table>

Key property: the way in which objects are added to, and removed from, the collection
stacks
Stacks

A *stack* is a linear data structure where objects are inserted or removed only at one end

- all insertions and deletions happen at one particular end of the data structure
- this end is called the *top* of the stack
- the other end is called the *bottom* of the stack

insertions and deletions happen at one end
Stacks: insertion of values

Insertion of a sequence of values into a stack:

5    17    33    9    43

stack top  None
stack bottom  None
Stacks: insertion of values

Insertion of a sequence of values into a stack:

5 17 33 9 43
Stacks: insertion of values

Insertion of a sequence of values into a stack:

5  17  33  9  43
Stacks: insertion of values

Insertion of a sequence of values into a stack:

5  17  33  9  43
Stacks: insertion of values

Insertion of a sequence of values into a stack:

5 17 33 9 43
Stacks: insertion of values

Insertion of a sequence of values into a stack:

5 17 33 9 43

Stack top
Stack bottom
Stacks: insertion of values

5  17  33  9  43

order in which values were inserted
Stacks: removal of values

5 17 33 9 43

*order in which values were inserted*

Removing values from the stack:
Stacks: removal of values

5  17  33  9  43

order in which values were inserted

Removing values from the stack:

43
Stacks: removal of values

5 17 33 9 43

*order in which values were inserted*

Removing values from the stack:

43 9
Stacks: removal of values

5  17  33  9  43

*order in which values were inserted*

Removing values from the stack:

43  9  33
Stacks: removal of values

5 17 33 9 43

*order in which values were inserted*

Removing values from the stack:

43 9 33 17
Stacks: removal of values

5 17 33 9 43

*order in which values were inserted*

Removing values from the stack:

43 9 33 17 5
Stacks: removal of values

5 17 33 9 43

*order in which values were inserted*

Removing values from the stack:

43 9 33 17 5

*order in which values were removed*
Stacks: LIFO property

5 17 33 9 43

order in which values were inserted

Removing values from the stack:

43 9 33 17 5

order in which values were removed

values are removed in reverse order from the order of insertion

"LIFO order"
Last in, First out
Methods for a Stack class

• Stack() : creates a new empty stack

• push(item) : adds item to the top of the stack
  – returns nothing
  – modifies the stack

• pop() : removes the top item from the stack
  – returns the removed item
  – modifies the stack

• is_empty() : checks whether the stack is empty
  – returns a Boolean
Implementing a Stack class

class Stack:
    # the top of the stack is the last item in the list
    def __init__(self):
        self._items = []
    def push(self, item):
        self._items.append(item)
    def pop(self):
        return self._items.pop()

removes and returns the last item in a list
EXERCISE

```python
>>> s = Stack()
```
```python
>>> s.push(4)
```
```python
>>> s.push(17)
```
```python
>>> s.push(5)
```
```python
>>> x = s.pop()
```
```python
>>> y = s.pop()
```
```
what does the stack s look like here? what are the values of x and y?
```
EXERCISE

```python
>>> s = Stack()
>>> s.push(4)
>>> s.push(17)
>>> s.push(5)
>>> x = s.pop()
>>> y = s.pop()
>>> s.push(x)
>>> s.push(y)
```

↔ what does the stack `s` look like here?
stacks: applications
An application: balancing parens

IDLE (the Python shell) matches up left and right parens ( ), brackets [ ], and braces { }

How does it figure out how far back to highlight?
An application: balancing parens

Basic idea: Match each ] with corresponding [
  - similarly for ( ... ) and { ... } pairs

  - Idea:
    ○ maintain a stack
    ○ on seeing '[' : push
    ○ on seeing ']' : pop the matching symbol

Example: [ 1, 2, [ 3, [ 4 ], 5 , [ 7 ] ] ]

Stack  (empty)
An application: balancing parens

Basic idea: Match each ] with corresponding [
- similarly for ( ... ) and { ... } pairs

- Idea:
  - maintain a stack
  - on seeing '[': push
  - on seeing ']': pop the matching symbol

Example: \[ 1, 2, [ 3, [ 4 ], 5, [ 7 ] ] \]
An application: balancing parens

Basic idea: Match each ] with corresponding [

- similarly for ( ... ) and { ... } pairs

- Idea:
  - maintain a stack
  - on seeing '[' : push
  - on seeing ']' : pop the matching symbol

Example: [ 1, 2, [ 3, [ 4 ], 5, [ 7 ] ] ]
An application: balancing parens

Basic idea: Match each ] with corresponding [
- similarly for ( ... ) and { ... } pairs

- Idea:
  ○ maintain a stack
  ○ on seeing '[' : push
  ○ on seeing ']' : pop the matching symbol

Example: [ 1, 2, [ 3, [ 4 ], 5 , [ 7 ] ] ]
An application: balancing parens

Basic idea: Match each ] with corresponding [
- similarly for ( ... ) and { ... } pairs

- Idea:
  - maintain a stack
  - on seeing '[': push
  - on seeing ']': pop the matching symbol

Example: [ 1, 2, [ 3, [ 4 ], 5 , [ 7 ] ] ]
An application: balancing parens

Basic idea: Match each ] with corresponding [  
  - similarly for ( ... ) and { ... } pairs

- Idea:
  - maintain a stack
  - on seeing '[' : push
  - on seeing ']' : pop the matching symbol

Example:  [ 1, 2, [ 3, [ 4 ], 5 , [ 7 ] ] ]
An application: balancing parens

Basic idea: Match each ] with corresponding [
  - similarly for ( ... ) and { ... } pairs

  - Idea:
    o maintain a stack
    o on seeing '[' : push
    o on seeing ']' : pop the matching symbol

Example:   [ 1, 2, [ 3, [ 4 ], 5 , [ 7 ] ] ]
An application: balancing parens

Basic idea: Match each ] with corresponding [ 
- similarly for ( ... ) and { ... } pairs

- Idea:
  - maintain a stack
  - on seeing ']' : push
  - on seeing ')' : pop the matching symbol

Example: [ 1, 2, [ 3, [ 4 ], 5 , [ 7 ] ] ]
An application: balancing parens

Basic idea: Match each ] with corresponding [
  - similarly for ( ... ) and { ... } pairs

  Idea:
  ○ maintain a stack
  ○ on seeing '[': push
  ○ on seeing ']': pop the matching symbol

Example: [ 1, 2, [ 3, [ 4 ], 5 , [ 7 ] ] ]

Stack
An application: balancing parens

Basic idea: Match each ] with corresponding [ 
- similarly for ( ... ) and { ... } pairs

- Idea:
  - maintain a stack
  - on seeing '[: push
  - on seeing ']' : pop the matching symbol

Example: [ 1, 2, [ 3, [ 4 ], 5 , [ 7 ] ] ]

Elaboration: Have each stack element keep track of the position of its [ ]
Related: Displaying web pages

Web page
Related: Displaying web pages

**Web page**

- main header: large font, bold
- secondary header: medium font, bold
- bold font
- italics font

**Display considerations**

**Question**: how does the web browser figure out how much a given display format should include? E.g., which text is in boldface, how much is in italics, etc.
Related: Displaying web pages

Web page

HTML source
Related: Displaying web pages

Web page

HTML source
Related: Displaying web pages

Web page

HTML source
Related: Displaying web pages

"tags"

<h1> : "open header 1"
</h1> : "close header 1"
<h2> : "open header 2"
</h2> : "close header 2"
<i> : "open italics"
</i> : "close italics"
...

HTML source
Related: Displaying web pages

Web page

Figuring out how to display different parts of the web page requires matching up “open-” and “close-” HTML tags. This is essentially the same problem as balancing parens.

HTML source
EXERCISE

```python
>>> s1 = Stack()
>>> s1.push(4)
>>> s1.push(17)
>>> s2 = Stack()
>>> s2.push(s1.pop())
>>> s2.push(s1.pop())
>>> s1.push(s2.pop())
>>> s1.push(s2.pop())
```

← what does the stack s1 look like here?
queues
Queues

A *queue* is a linear data structure where insertions and deletions happen at different ends

- insertions happen at one end (the queue's "back", or "tail")
- deletions happen at the other end (the queue's "front", or "head")
Insertion of a sequence of values into a queue:

5  17  33  9  43
Queues: insertion of values

Insertion of a sequence of values into a queue: 5 17 33 9 43
Queues: insertion of values

Insertion of a sequence of values into a queue:

5  17  33  9  43
Queues: insertion of values

Insertion of a sequence of values into a queue:

queue back

5 17 33 9 43

queue front
Queues: insertion of values

Insertion of a sequence of values into a queue:

Queue back

Queue front

<table>
<thead>
<tr>
<th>9</th>
<th>33</th>
<th>17</th>
<th>5</th>
</tr>
</thead>
</table>

5 17 33 9 43
Queues: insertion of values

Insertion of a sequence of values into a queue:

```
43  9  33  17  5
```

5  17  33  9  43

queue back
queue front
Queues: insertion of values

order of insertion

5  17  33  9  43

43  9  33  17  5

queue back

queue front
Queues: removal of values

*order of insertion*

Removing values from this queue:

| 43 | 9  | 33 | 17 | 5  |

queue back | queue front
Queues: removal of values

Order of insertion

Removing values from this queue:

5

queue back

queue front
Queues: removal of values

Order of insertion

Removing values from this queue:

5  17

43  9  33  17  5
Queues: removal of values

order of insertion

Removing values from this queue:

5  17  33  9  43

43  9  33  17  5

queue back

queue front
Queues: removal of values

order of insertion

Removing values from this queue:

5  17  33  9  43

43  9  33  17  5

queue back

queue front
Queues: removal of values

Order of insertion

Removing values from this queue:

\[ \text{queue back} \quad \text{None} \quad \text{queue front} \quad \text{None} \]
Queues: removal of values

**order of insertion**

5 17 33 9 43

**order of removal**

5 17 33 9 43
Queues: FIFO property

order of insertion

5  17  33  9  43

order of removal

5  17  33  9  43

values are removed in order in which they are inserted

"FIFO order"
First in, First out
Methods for a queue class

• `Queue()`: creates a new empty queue
• `enqueue(item)`: adds `item` to the back of the queue
  – modifies the queue
  – returns nothing
• `dequeue()`: removes and returns the item at the front of the queue
  – returns the removed item
  – modifies the queue
• `is_empty()`: checks whether the queue is empty
  – returns a Boolean
Implementing a Queue class

class Queue:

    # the front of the queue is the first item in the list

def __init__(self):
    self._items = []

def enqueue(self, item):
    self._items.append(item)

def dequeue(self):
    return self._items.pop(0)
Implementing a Queue class II

```python
class Queue:
    # the front of the queue is the last item in the list
    def __init__(self):
        self._items = []

    def enqueue(self, item):
        self._items.insert(0, item)

    def dequeue(self):
        return self._items.pop()
```

removes and returns the last item in the list

head tail
```python
>>> q = Queue()
>>> q.enqueue(4)
>>> q.enqueue(17)
>>> x = q.dequeue()
>>> q.enqueue(5)
>>> y = q.dequeue()
```

← what are the values of $x$ and $y$?
EXERCISE

>>> q = Queue()
>>> q.enqueue(4)
>>> q.enqueue(17)
>>> x = q.dequeue()
>>> y = q.dequeue()
>>> q.enqueue(y)
>>> q.enqueue(x)
>>> q.enqueue(y)

← what does the queue q look like here?
queues: applications
Application 1: Simulation

• Suppose we are opening a grocery store. How many checkout lines should we put in?
  – too few ⇒ long wait times, unhappy customers
  – too many ⇒ wasted money, space

• Use simulations of the checkout process to guide the decision
  – study existing stores to figure out typical shopping and checkout times
  – estimate no. of customers expected at the new location
  – run simulations to determine customer wait time and checkout line utilization under different scenarios
Discrete event simulation

By varying the parameters of the simulation (arrival and departure rates, no. of servers) we can try out different scenarios.

departure rate distribution

arrival rate distribution
Application 2: game playing

Goal: to write a program to play a 2-person game (e.g., tic-tac-toe, chess, go, ...)

How does this work?
Application 2: game playing

Goal: to write a program to play a 2-person game (e.g., tic-tac-toe, chess, go, ...)

Generate successive levels of board positions
• At each level, pick best move for the player at that level
• Work backwards to find the move that will lead to the best position $n$ moves later

*best position for the computer among all positions at this level*
Application 2: game playing

• For a nontrivial game (e.g., chess, go) the tree is usually too large to build or explore fully
  
  – also, usually there are time constraints on play
  – our previous tree traversal algorithms don't work

• Game-playing algorithms typically explore the tree level by level
  
  – consider the nodes at depth 1, then depth 2, etc.
Level-by-level tree traversal

Level 1

1

2

3

4

5 6 7 8 9 10
Level-by-level tree traversal
Level-by-level tree traversal

This order of traversal is called *breadth-first traversal*
Breadth-first tree traversal

Breadth-first traversal order:
1 2 3 4 5 6 7 8 9 10
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  – enqueue the nodes at that level
  – while queue not empty:
    ◦ node = $q$.dequeue()
    ◦ process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  – enqueue the nodes at that level
  – while queue not empty:
    ○ node = $q$.dequeue()
    ○ process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  - enqueue the nodes at that level
  - while queue not empty:
    ◦ node = $q$.dequeue()
    ◦ process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  - enqueue the nodes at that level
  - while queue not empty:
    ◦ node = $q$.dequeue()
    ◦ process node
Breadth-first tree traversal

Data structure: use a queue \( q \)

Algorithm:

- for each level in the tree:
  - enqueue the nodes at that level
  - while queue not empty:
    - node = \( q \).dequeue()
    - process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  ‒ enqueue the nodes at that level
  ‒ while queue not empty:
    ○ node = $q$.dequeue()
    ○ process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  – enqueue the nodes at that level
  – while queue not empty:
    ◦ node = $q$.dequeue()
    ◦ process node

![Diagram of a tree with nodes and queue](image)

- back
- front

node None
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  - enqueue the nodes at that level
  - while queue not empty:
    ◦ node = $q$.dequeue()
    ◦ process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:
• for each level in the tree:
  – enqueue the nodes at that level
  – while queue not empty:
    ◦ node = $q$.dequeue()
    ◦ process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  − enqueue the nodes at that level
  − while queue not empty:
    ○ node = $q$.dequeue()
    ○ process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  – enqueue the nodes at that level
  – while queue not empty:
    ◦ node = $q$.dequeue()
    ◦ process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:
• for each level in the tree:
  – enqueue the nodes at that level
  – while queue not empty:
    ○ node = $q$.dequeue()
    ○ process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  – enqueue the nodes at that level
  – while queue not empty:
    ○ node = $q$.dequeue()
    ○ process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  - enqueue the nodes at that level
  - while queue not empty:
    ○ node = $q$.dequeue()
    ○ process node
Breadth-first tree traversal

Data structure: use a queue \( q \)

Algorithm:

- for each level in the tree:
  - enqueue the nodes at that level
  - while queue not empty:
    - node = \( q \).dequeue()
    - process node

```
10   ... etc. ...   6
  ^          ^
 back       front
```

node None
Breadth-first vs. Depth-first

• Stacks and queues are closely related structures
• What if we use a stack in our tree traversal?
Breadth-first vs. Depth-first

- Stacks and queues are closely related structures
- What if we use a stack in our tree traversal?
Breadth-first vs. Depth-first

- Stacks and queues are closely related structures
- What if we use a stack in our tree traversal?
Breadth-first vs. Depth-first

• Stacks and queues are closely related structures
• What if we use a stack in our tree traversal?
Breadth-first vs. Depth-first

- Stacks and queues are closely related structures
- What if we use a stack in our tree traversal?
Breadth-first vs. Depth-first

• Stacks and queues are closely related structures
• What if we use a stack in our tree traversal?
Breadth-first vs. Depth-first

• Stacks and queues are closely related structures
• What if we use a stack in our tree traversal?
  – the deeper levels of the tree are explored first
  – this is referred to as depth-first traversal
Summary

• Stacks and queues are *linear* data structures
  – items can be thought of as arranged in a line
  – each item has a position and a before/after relationship with the other items

• They differ in the way items are added and removed
  – stacks: items added and removed at one end
    ○ results in LIFO behavior
  – queues: items added at one end, removed at the other
    ○ results in FIFO behavior

• They find a wide range of applications in computer science
EXERCISE

```python
>>> s = Stack()
>>> q = Queue()
>>> q.enqueue(17)
>>> s.push(5)
>>> q.enqueue(19)
>>> s.push(q.dequeue())
>>> s.push(q.dequeue())
>>> q.enqueue(s.pop())
```

← what do s and q look like here?
Puzzle:

- Given: a stack $s$; a variable $x$; arithmetic on $s, x$ only (no constants); and the input sequence $[2, 3]$

- Construct: the stack top

"input sequence" means the values are consumed as they are read