CSc 120
Introduction to Computer Programming II

Adapted from slides
by Dr. Saumya Debray

12: Abstract Data Types
Stacks and Queues
Abstract Data Types
Abstract Data Types

An abstract data type (ADT) describes a set of data values and associated operations that are specified independent of any particular implementation.

An ADT is a logical description of how we view the data and the operations allowed on that data.

- describes what the data represents
- not how is the data represented

The data is encapsulated.
Abstract Data Types

Because the data is *encapsulated* we can change the underlying implementation without affecting the way the ADT behaves.

- the logical description remains the same
- the operations remain the same

Example:

- lists
  - Python built-in lists
  - linked lists implementation
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

- Linear
  - Python lists
  - Linked lists

- Not linear
  - Dictionaries
  - Sets
Today's topic

**Linear**
- Python lists
- Linked lists
- Stacks
- Queues
- Dequeues

**Not linear**
- Dictionaries
- Sets

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

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
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:

\[
\begin{array}{c}
5 & 17 & 33 & 9 & 43 \\
\end{array}
\]
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

5 17 33 9 43

order in which values were inserted
Stacks: removal of values

Removing values from the stack:

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:

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

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

order in which values were inserted

Removing values from the stack:

order in which values were removed

"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()
EXERCISE

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

← what does the stack `s` look like here? what are the values of `x` and `y`?
EXERCISE

>>> 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 { }

>>> x = [1, 2, [3, 4, [5], 7], 8]

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:  
  o maintain a stack  
  o on seeing '[' : push  
  o 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 ] ] ]
```

Stack

33
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:
    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:
  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:
  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:
  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:
  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:
    o maintain a stack
    o on seeing '[: push
    o 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:
    o maintain a stack
    o on seeing '[' : push
    o on seeing ']' : pop the matching symbol

Example:  [ 1, 2, [ 3, [ 4 ], 5 , [ 7 ] ]]  Stack
  matches: pop
  top
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:
  o maintain a stack
  o on seeing '[' : push
  o 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 [}
Write a function `balanced(s)` that returns `True` if the string `s` is balanced with respect to `‘[‘` and `’]’` and `False` otherwise.

class Stack:
    def __init__(self):
        self._items = []
    def push(self, item):
        self._items.append(item)
    def pop(self):
        return self._items.pop()
    def is_empty(self):
        return self._items == []
CSc 120: Phylogenetic Trees

This problem brings together many different programming and trees. It is one of the most technically challenging problems.

Background

An evolutionary tree (also called a phylogenetic tree) is a

This program involves writing code to construct phylogenetic trees. For example, since programs are sequences of characters, we

Expected Behavior

Write a Python program, in a file phylo.py, that behaves

1. Read in the input parameters:
   - Read in the name of an input file using input
   - Read in an integer value N using input

2. Read in the input file. The file format is specified un
Related: Displaying web pages

Web page

Main header: large font, bold

Secondary header: medium font, bold

Expected Behavior

Background

Bold font

Italics font

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

CSc 120: Phylogenetic Trees

This problem brings together many different programming constructs we covered over the course of the semester including: manipulation, (Python) lists, dictionaries, tuples, classes, list comprehensions, and trees. It is one of the most technically challenging programs assigned in this class this semester. I think it's also one of the most interesting.

<code>
<h1>CSc 120: Phylogenetic Trees</h1>
This problem brings together many different programming constructs we covered over the course of the semester including: manipulation, (Python) lists, dictionaries, tuples, classes, list comprehensions, and trees. It is one of the most technically challenging programs assigned in this class this semester. I think it's also one of the most interesting.
</code>

Expected Behavior

Write a Python program, in a file `phylo.py`, that behaves as specified below:

1. Read in the input parameters:
   - Read in the name of an input file using `input`
   - Read in an integer value `N` using `input` for `n-gram`

2. Read in the input file. The file format is specified un...
Related: Displaying web pages

Web page

The University of Arizona
Department of Computer Science

CSc 120: Phylogenetic Trees

This problem brings together many different programming constructs and techniques we covered over the course of the semester including: manipulation, (Python) lists, dictionaries, tuples, classes, list comprehensions, and trees. It is one of the most technically challenging programs assigned in this class and I think it’s also one of the most interesting.

Background

An evolutionary tree (also called a phylogenetic tree) is a tree that expresses evolutionary relationships between a set of organisms.

Expected Behavior

Write a Python program, in a file `phylo.py`, that behaves:

1. **Read in the input parameters.**
   - Read in the name of an input file using `input`
   - Read in an integer value `N` using `input`

2. **Read in the input file.** The file format is specified in

```html
<br>
<h1>CSc 120: Phylogenetic Trees</h1>

This problem brings together many different programming constructs and techniques we covered over the course of the semester including: manipulation, (Python) lists, dictionaries, tuples, classes, list comprehensions, and trees. It is one of the most technically challenging programs assigned in this class and I think it’s also one of the most interesting.

<h2>Background</h2>

An evolutionary tree (also called a phylogenetic tree) is a tree that expresses evolutionary relationships between a set of organisms.

<h2>Expected Behavior</h2>

Write a Python program, in a file `phylo.py`, that behaves:

1. **Read in the input parameters.**
   - Read in the name of an input file using `input`
   - Read in an integer value `N` using `input`

2. **Read in the input file.** The file format is specified in
CSc 120: Phylogenetic Trees

This problem brings together many different programming constructs we covered over the course of the semester including: manipulation, (Python) lists, dictionaries, tuples, classes, list comprehensions, and trees. It is one of the most technically challenging programs assigned in this class and so I think it’s also one of the most interesting.

An evolutionary tree (also called a phylogenetic tree) is a tree that expresses evolutionary relationships between a set of organisms.

This program involves writing code to construct phylogenetic trees from the genome sequences of a set of organisms. (Of course, there is inherent genetic about the techniques we use and the code we write; example, since programs are sequences of characters, we could just apply this approach to sets of programs.)

Write a Python program, in a file phylo.py, that behaves as specified below.

```html
<h2>Expected Behavior</h2>
Write a Python program, in a file <tt>phylo.py</tt>, that behaves as specified below.
```
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"
...
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.
EXERCISE

>>> s1 = Stack()
>>> s1.push(4)
>>> s1.push(17)

```python
>>> s2 = Stack()
>>> s2.push(s1.pop())
>>> s2.push(s1.pop())
```  
```python
>>> s1.push(s2.pop())
>>> s1.push(s2.pop())
```  
← what does the stack s1 look like here?
Hypothetical: Python 7 has just been released and built-in lists are inefficient. In fact, all operations are $O(n^2)$.

Avoid these inefficiencies by implementing the Stack class using LinkedLists.
queues
A Queue ADT

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")
Queues: insertion of values

Insertion of a sequence of values into a queue:

queue back None

queue front None

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:

5 17 33 9 43
## Queues: insertion of values

Insertion of a sequence of values into a queue:

| 9 | 33 | 17 | 5 |

---

queue back

queue front
Queues: insertion of values

Insertion of a sequence of values into a queue:

| 43 | 9  | 33 | 17 | 5  |

queue back

queue front
Queues: insertion of values

order of insertion → 5 17 33 9 43
Queues: removal of values

*order of insertion* → 5 17 33 9 43

Removing values from this queue:

<table>
<thead>
<tr>
<th>43</th>
<th>9</th>
<th>33</th>
<th>17</th>
<th>5</th>
</tr>
</thead>
</table>

queue back

queue front
Queues: removal of values

order of insertion → 5 17 33 9 43

Removing values from this queue: 5

| 43 | 9 | 33 | 17 | 5 |

queue back

queue front
Queues: removal of values

*order of insertion* 5 17 33 9 43

Removing values from this queue: 5 17

| 43 | 9 | 33 | 17 | 5 |

queue back

queue front
Queues: removal of values

order of insertion → 5 17 33 9 43

Removing values from this queue: 5 17 33
Queues: removal of values

*order of insertion* → 5 17 33 9 43

Removing values from this queue: 5 17 33 9

| 43 | 9 | 33 | 17 | 5 |

queue back

queue front
Queues: removal of values

*order of insertion*  
5  17  33  9  43

Removing values from this queue:  
5  17  33  9  43

| queue back | None |
| queue front | 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
• `size()`: returns the size of the queue
  – returns an integer
Implementing a queue class

• Use a built-in list for the internal representation
  – Python lists can be added at to the front or at the end

• First implementation:
  – the head is the 0\textsuperscript{th} element
  – the tail is the n\textsuperscript{th} element

• Second implementation
  – the head is the n\textsuperscript{th} element
  – the tail is the 0\textsuperscript{th} element
Implementing a Queue class I

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

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()
EXERCISE

```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

```python
>>> 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

• Typical applications simulate problems that require data to be managed in a FIFO manner
  – Hot potato
    o Kids stand in a circle and pass a “hot potato” around until told to stop. The person holding the potato is taken out of the circle. The process is repeated until only one person remains.
  – Generalized: Given $n$ elements, eliminate every $k$th element repeatedly until only 1 element is left. What was the original position of the remaining element?

• Use a *simulation* to determine which element remains.
Problem: Given $n$ elements, eliminate every $k$th element repeatedly until only 1 element is left. What was the original position of the remaining element?

use a queue to simulate the circle

$n$ is the number of elements to put into the queue while there is more than one element in the queue

eliminate every $k$th element

What operations take an element from the front of the queue and place it at the back of the queue?
General solution for k=2

• Given n elements, eliminate every kth element repeatedly until only 1 element is left. What was the original position of the remaining element?
• When k = 2, the original position can be derived from the binary representation of n.

  Take the first digit of the binary representation.
  Move it to the end
  The result is the original position.

Ex: n = 41, k=2

In binary
  n = 101001
Therefore, the original position (in binary) is
  010011
  and 010011 = 2^4 + 2^1 + 2^0 = 19

Application 2 : 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.
Summary

- Stacks and queues are abstract data types (ADTs)
  - similar in that they are both 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
    o results in LIFO behavior
  - queues: items added at one end, removed at the other
    o results in FIFO behavior
- They find a wide range of applications in computer science
A Deque ADT

A *deque* is a linear data structure where insertions and deletions happen at both ends.