Linked Lists

Linked Structures

Eric Roberts
CSCI 121
December 2, 2019

The Beacons of Gondor

For answer Gandalf cried aloud to his horse: “On, Shadowfax! We must hurry. Time is short. See! The beacons of Gondor are alight, calling for aid. War is kindled. See, there is the fire on Amon Din, and flame on Eilenach; and there they go spreading west: Nardol, Erelas, Min-Rimmon, Calenhad, and the Halifirien on the borders of Rohan.”
—J. R. R. Tolkien, The Return of the King, 1955

In this scene so brilliantly captured in Peter Jackson’s film adaptation of The Return of the King, Rohan is alerted to the danger to Gondor by a succession of signal fires moving from mountain top to mountain top. This scene is a perfect illustration of the idea of message passing in a linked list.

Message Passing in Linked Structures

• Linked structures of this form can be represented using classes such as the one on the right.
• You can then create a chain of SignalTower objects like this:

```
Minas Tirith
Amon Dîn
Eilenach
Nardol
Erelas
Min-Rimmon
Calenhad
Halifirien
Rohan
```

• Calling signal on the first tower sends a message down the chain.

Using Links to Correct Errors

• Suppose you’re Thomas Jefferson and that you’re writing timeless prose to mark our nation’s founding. Of the British outrages, you’ve written that “our repeated petitions have been answered by repeated injury.”

• Now someone wants you to add “only” after “answered” in the finished text. What do you do?
Using Links to Specify Order

- If a word-processing application stores the text of a document as one long string, making insertions in the middle raises a similar problem to the one that Jefferson faced when inserting the word only into the Declaration of Independence. Adding a word forces the application to shift the characters in the rest of the string to make room for the insertion.
- An alternative strategy is to store the text of the document as a linked list of words:

  - This representation allows you to insert a new word without changing the position of any of the other words in memory.

The Stack Metaphor

- A stack is an abstract data collection in which elements must be extracted in a last-in-first-out order.
- The fundamental operations on a stack are push, which adds a new value to the top of the stack, and pop, which removes and returns the top value.
- One of the most common metaphors for the stack concept is a spring-loaded storage tray for dishes. Adding a new dish to the stack pushes any previous dishes downward. Taking the top dish away allows the dishes to pop back up.

Methods in the Minimal Stack Class

```
stack = Stack()
    Creates an empty stack.

stack.isEmpty()
    Returns True if the stack is empty.

stack.push(value)
    Pushes a new value onto the stack.

stack.pop()
    Removes and returns the top value from the stack.
```

The Queue Metaphor

- A queue is an abstract data collection in which elements are extracted in a first-in-first-out order.
- The fundamental operations on a queue (which have a much greater diversity of names than their stack counterparts) are enqueue, which adds a new value to the end of the queue, and dequeue, which removes and returns the first value.
- The standard metaphor for a queue is a waiting line in which customers are served in the order in which they arrived.

Methods in the Queue Class

```
queue = Queue()
    Creates an empty queue.

queue.isEmpty()
    Returns True if the queue is empty.

queue.enqueue(v)
    Adds the value v to the end of the queue (which is called its tail).

queue.dequeue()
    Removes and returns the first value in the queue (which is called its head).
```

Comparing Stacks and Queues

Stack:
```
push
pop
```

Queue:
```
enqueue
decqueue
```
A List-Based Implementation

- Both stacks and queues are easy to implement using Python’s predefined list class, as shown on the next two slides.
- The fundamental operations are in fact so easy to implement that it often makes sense to use the list class directly, without implementing a new class. If you adopt this strategy, the standard methods look like this:

  ```
  isEmpty() → len(list) == 0
  push(v) → list.append(v)
  pop() → list.pop()
  enqueue(v) → list.append(v)
  dequeue() → list.pop(0)
  ```

Code for the List-Based Stack

```python
class Stack:
    """This class implements a minimal stack ADT."""
    def __init__(self):
        """Creates an empty stack."""
        self._list = []
    def isEmpty(self):
        """Returns True if the stack is empty."""
        return len(self._list) == 0
    def push(self, v):
        """Puts the value v on the stack."""
        self._list.append(v)
    def pop(self):
        """Removes the most recent value pushed."""
        if self.isEmpty():
            raise IndexError("Stack is empty")
        return self._list.pop()
```

Code for the List-Based Queue

```python
class Queue:
    """This class implements a minimal queue ADT."""
    def __init__(self):
        """Creates an empty queue."""
        self._list = []
    def isEmpty(self):
        """Returns True if the queue is empty."""
        return len(self._list) == 0
    def enqueue(self, v):
        """Adds the value v to the end of the queue."""
        self._list.append(v)
    def dequeue(self):
        """Removes and returns the first value in the queue."""
        if self.isEmpty():
            raise IndexError("Queue is empty")
        return self._list.pop(0)
```

Implementing a Linked-List Stack

- The code on the next two slides implements the Stack class using a linked list.
- Important features to note about the implementation include:
  - The private class_Cell is used to hold each linked-list cell.
  - The attributes of_Cell are_data and_link.
  - The linked list is constructed so the top of the stack is at the beginning, which ensures O(1) performance for push and pop.

Code for the Linked-List Stack

```python
class _Cell:
    def __init__(self, data, link=None):
        self.data = data
        self.link = link

def pop(self):
    """Pops the most recent value pushed."""
    if self.isEmpty():
        raise IndexError("Stack is empty")
    v = self._start.data
    self._start = self._start.link
    return v

def push(self, v):
    """Puts the value v on the stack."""
    self._start = _Cell(v, self._start)
```
Implementing a Linked-List Queue

- In the linked-list implementation, the private data structure for the `Queue` class requires two references to `Cell` objects: a `head` reference that points to the first cell in the chain, and a `tail` reference that points to the last cell.
- The `Cell` structure is the same as the one for the linked-list stack.

Code for the Linked-List Queue

```python
class Queue:
    """This class implements a minimal queue ADT."""
    def __init__(self):
        self._head = None
        self._tail = None
    def is_empty(self):
        """Returns True if the queue is empty."""
        return self._head is None
    def enqueue(self, v):
        """Adds the value v to the end of the queue."""
        cell = Queue.Cell(v)
        if self._head is None:
            self._head = cell
        else:
            self._tail._link = cell
            self._tail = cell
```

Tracing the Linked-List Queue

- `q = Queue()`
  `q.enqueue('A')`  `q.enqueue('B')`  `q.enqueue('C')`  `q.dequeue()`  `q.dequeue()`  `q.dequeue()`
Tracing the Linked-List Queue

```python
def dequeue(self):
    if self._head is None:
        raise IndexError("Empty")
    v = self._head._data
    self._head = self._head._link
    if self._head is None:
        self._tail = None
    return v
```

```python
def enqueue(self, v):
    cell = Queue._Cell(v)
    if self._head is None:
        self._head = cell
        self._tail = cell
    else:
        self._tail._link = cell
        self._tail = cell
```

Tracing the Linked-List Queue

```python
def dequeue(self):
    if self._head is None:
        raise IndexError("Empty")
    v = self._head._data
    self._head = self._head._link
    if self._head is None:
        self._tail = None
    return v
```

```python
def enqueue(self, v):
    cell = Queue._Cell(v)
    if self._head is None:
        self._head = cell
        self._tail = cell
    else:
        self._tail._link = cell
        self._tail = cell
```

Tracing the Linked-List Queue

```python
def dequeue(self):
    if self._head is None:
        raise IndexError("Empty")
    v = self._head._data
    self._head = self._head._link
    if self._head is None:
        self._tail = None
    return v
```

```python
def enqueue(self, v):
    cell = Queue._Cell(v)
    if self._head is None:
        self._head = cell
        self._tail = cell
    else:
        self._tail._link = cell
        self._tail = cell
```