UIT2201 Programming and Data Structures

Link-based Stack and Queue

Chandrabose Aravindan
<AravindanC@ssn.edu.in>

Professor of Information Technology
SSN College of Engineering

June 29, 2022

Linked List

- Just-in-time allocation of memory of an object added to the list and release of memory when the object is removed
- Allocated memory may not be contiguous! So, along with each object we should store reference to the next (and previous) object!
- We need a light-weight object (often referred to as a Node) to store an object along with other book-keeping information
- A list can be implemented as a sequence of linked nodes

Node Structure

- The simplest possible node structure contains two fields: a reference to an object (stored in that node); and a reference to the 'next' node in the sequence
- The 'next' field will be 'None' where there is no next node

Node Class

- It should be easy to implement a Python class for nodes with single (next) link

Node Class

- It should be easy to implement a Python class for nodes with single (next) link

```
class Node(object):

    __slots__ = ['_itemNode', '_nextNode']

    def __init__(self, item=None, next=None):
        self._itemNode = item
        self._nextNode = next
```

Link-based Stack

Implementation of link-based stack

- We will implement our constructor based on our design decisions

Implementation of link-based stack

- We will implement our constructor based on our design decisions

```
from Node import Node

class LinkedStack(object):

    __slots__ = ['_top']

    def __init__(self):
        self._top = Node()
```

isEmpty?

- It is easy to check if a stack is empty!

isEmpty?

- It is easy to check if a stack is empty!

```
class LinkedStack(object):

    def isEmpty(self):
        return (self._top._nextNode is None)
```

Top operation

- How do we peek into our stack (return the top element)?

Top operation

- How do we peek into our stack (return the top element)?

```
class Empty(Exception):
    pass

class LinkedStack(object):

    def top(self):
        if self._top._nextNode is None:
            raise Empty("Can't␣peek␣in␣to␣an␣empty␣sta
        return self._top._nextNode._itemNode
```

Pop operation

- How do we pop from our stack?

Pop operation

- How do we pop from our stack?

```
class Empty(Exception):
    pass

class LinkedStack(object):

    def pop(self):
        if self._top._nextNode is None:
            raise Empty("Can't␣pop␣from␣an␣empty␣stack
        item = self._top._nextNode._itemNode
        self._top = self._top._nextNode
        return item
```

Push operation

- How do we push into our stack?

Push operation

- How do we push into our stack?

```
class LinkedStack(object):

    def push(self, item):
        self._top._nextNode =
          Node(item, self._top._nextNode)
```

Link-based Queue

Implementation of link-based queue

- We will implement our constructor based on our design decisions

Implementation of link-based queue

- We will implement our constructor based on our design decisions

```
from Node import Node

class LinkedQueue(object):

    __slots__ = ['_front', '_rear', '_size']

    def __init__(self):
        self._front = Node()
        self._rear = self._front
        self._size = 0
```

isEmpty?

- It is easy to find the length of a queue and check if a queue is empty!

isEmpty?

- It is easy to find the length of a queue and check if a queue is empty!

```
class LinkedQueue(object):

    def __len__(self):
        return self._size


    def isEmpty(self):
        return self._front._nextNode is None
```

Front operation

- How do we peek into our queue (return the first element)?

Front operation

- How do we peek into our queue (return the first element)?

```
class Empty(Exception):
    pass

class LinkedQueue(object):

    def front(self):
        if self._front._nextNode is None:
            raise Empty("Queue␣is␣empty!")
        return self._front._nextNode._itemNode
```

Dequeue operation

- How do we dequeue (delete and return) the first element?

Dequeue operation

- How do we dequeue (delete and return) the first element?

```
class Empty(Exception):
    pass

class LinkedQueue(object):

    def dequeue(self):
        if self._front._nextNode is None:
            raise Empty("Queue␣is␣empty!")
        item = self._front._nextNode._itemNode
        self._front = self._front._nextNode
        self._size -= 1
        return item
```

Enqueue operation

- How do we enqueue an element to a queue?

Enqueue operation

- How do we enqueue an element to a queue?

```
class LinkedQueue(object):

    def enqueue(self, item):
        self._rear._nextNode = Node(item, None)
        self._rear = self._rear._nextNode
        self._size += 1
```

Link-based Double Ended Queue

- Is it straightforward to implement double ended queue (“deck”) using this structure?

Link-based Double Ended Queue

- Is it straightforward to implement double ended queue (“deck”) using this structure?
- We have already seen deleting from the front of the queue, and it should be easy to insert at the front as well

Link-based Double Ended Queue

- Is it straightforward to implement double ended queue (“deck”) using this structure?
- We have already seen deleting from the front of the queue, and it should be easy to insert at the front as well
- Similarly, we have already seen inserting at the rear of the queue

Link-based Double Ended Queue

- Is it straightforward to implement double ended queue (“deck”) using this structure?
- We have already seen deleting from the front of the queue, and it should be easy to insert at the front as well
- Similarly, we have already seen inserting at the rear of the queue
- But, is it easy to delete the element at the rear?

Doubly-linked Structure

Doubly-linked node structure

- We need to introduce a ‘prev’ field to our node structure

Doubly-linked node structure

- We need to introduce a ‘prev’ field to our node structure

```
class DNode(Node):

    __slots__ = ['_prevNode']

    # Constructor with optional parameters
    def __init__(self, itemNode=None
                     nextNode=None, prevNode=None):
        super().__init__(itemNode, nextNode)
        self._prevNode = prevNode
```

Implementation of doubly-linked dequeue

- We will implement our constructor based on our design decisions

Implementation of doubly-linked dequeue

- We will implement our constructor based on our design decisions

```
from Node import DNode

class LinkedDequeue(object):

    __slots__ = ['_front', '_rear', '_size']

    def __init__(self):
        self._front = DNode()
        self._rear = DNode()
        self._front._nextNode = self._rear
        self._rear._prevNode = self._front
        self._size = 0
```

Implementing “deck” operations

- Implementation of basic operations of “deck” is left as an exercise!

Another use-case

- Consider a need for a queue that holds a sequence of jobs to executed

Another use-case

- Consider a need for a queue that holds a sequence of jobs to executed
- Suppose round-robin scheduling is followed: first job is dequeued, run for specific amount of time, and enqueued at the end

Another use-case

- Consider a need for a queue that holds a sequence of jobs to executed
- Suppose round-robin scheduling is followed: first job is dequeued, run for specific amount of time, and enqueued at the end
- This can be done with our link-based queue, but special “rotate()” method may be more efficient

Another use-case

- Consider a need for a queue that holds a sequence of jobs to executed
- Suppose round-robin scheduling is followed: first job is dequeued, run for specific amount of time, and enqueued at the end
- This can be done with our link-based queue, but special “rotate()” method may be more efficient
- We get the front of the queue, process the job, and ‘rotate’ whereby first element becomes the last element of the queue

Circularly linked structure

Implementation of circularly-linked queue

- We will implement our constructor based on our design decisions

Implementation of circularly-linked queue

- We will implement our constructor based on our design decisions

```
from Node import Node

class CircularlyLinkedQueue(object):

    __slots__ = ['_rear', '_size']

    def __init__(self):
        self._rear = None  # NO dummy header
        self._size = 0
```

Implementation of circularly-linked queue

- The required ‘rotate’ operation should be easy to implement!

Implementation of circularly-linked queue

- The required ‘rotate’ operation should be easy to implement!

```
class CircularlyLinkedQueue(object):

    def rotate(self):
        if (self._size > 0):
            self._rear = self._rear._nextNode
```

Implementation of circularly-linke Queue

- Implementation of other basic operations of queue is left as an exercise!

Implementation of circularly-linke Queue

- Implementation of other basic operations of queue is left as an exercise!
- Note that there is no dummy header, and so we should check for different cases while adding and deleting elements

Summary

- We have discussed link-based implementations of ADT Stack and ADT Queue
- Certain variations of linked-lists, namely circularly linked lists and doubly linked lists have also been discussed

What next?

- We will look at some of the applications of Lists, Stacks, and Queues
- We will start exploring non-linear structures — first the Tree ADT and then Graph ADT