UIT2201 Programming and Data Structures

Stacks and Queues

Chandrabose Aravindan
<AravindanC@ssn.edu.in>

Professor of Information Technology
SSN College of Engineering

June 21, 2022

Stack ADT

- As the name suggests, Stack is a linear collection where objects are "stacked" one above the other
- Insertion and deletion can only happen at the "top"
- It is a special kind of a list where insertion and deletion happen at one end only
- There is only one recognized position called "top"
- It is also referred to as Last-in-First-out (LIFO) structure

Stack Operations

- stk = Stack() — creates and returns a new Stack structure
- stk.isEmpty() — returns "TRUE" if *stk* is empty and returns "FALSE" otherwise
- stk.clear() — removes all the objects in *stk* and resets it to an empty stack

Core Stack Operations

- stk.push(x) — inserts object x at the top of the stack
  - $\langle\rangle$ —> $\langle x\rangle$
  - $\langle e_n, e_{n-1}, \cdots, e_2, e_1\rangle$ —> $\langle x, e_n, e_{n-1}, \cdots, e_2, e_1\rangle$

Core Stack Operations

- stk.push(x) — inserts object x at the top of the stack
  - $\langle\rangle \longrightarrow \langle x \rangle$
  - $\langle e_n, e_{n-1}, \cdots, e_2, e_1 \rangle \longrightarrow \langle x, e_n, e_{n-1}, \cdots, e_2, e_1 \rangle$
- stk.top() — returns the object at the top of the stack *stk* (*stk* is not modified!)
  - $\langle e_n, e_{n-1}, \cdots, e_2, e_1 \rangle \longrightarrow e_n$
  - $\langle\rangle \longrightarrow$???

Core Stack Operations

- stk.push(x) — inserts object x at the top of the stack
  - $\langle\rangle$ —> $\langle x\rangle$
  - $\langle e_n, e_{n-1}, \cdots, e_2, e_1\rangle$ —> $\langle x, e_n, e_{n-1}, \cdots, e_2, e_1\rangle$
- stk.top() — returns the object at the top of the stack *stk* (*stk* is not modified!)
  - $\langle e_n, e_{n-1}, \cdots, e_2, e_1\rangle$ —> e_n
  - $\langle\rangle$ —> ???
- stk.pop() — removes the object at the top of the stack *stk*
  - $\langle e_n, e_{n-1}, \cdots, e_2, e_1\rangle$ —> $\langle e_{n-1}, e_{n-2}, \cdots, e_2, e_1\rangle$
  - $\langle x\rangle$ —> $\langle\rangle$
  - $\langle\rangle$ —> ???

Core Stack Operations

- stk.push(x) — inserts object x at the top of the stack
  - $\langle\rangle \longrightarrow\!\!> \langle x\rangle$
  - $\langle e_n, e_{n-1}, \cdots, e_2, e_1\rangle \longrightarrow\!\!> \langle x, e_n, e_{n-1}, \cdots, e_2, e_1\rangle$
- stk.top() — returns the object at the top of the stack *stk* (*stk* is not modified!)
  - $\langle e_n, e_{n-1}, \cdots, e_2, e_1\rangle \longrightarrow\!\!> e_n$
  - $\langle\rangle \longrightarrow\!\!>$???
- stk.pop() — removes the object at the top of the stack *stk*
  - $\langle e_n, e_{n-1}, \cdots, e_2, e_1\rangle \longrightarrow\!\!> \langle e_{n-1}, e_{n-2}, \cdots, e_2, e_1\rangle$
  - $\langle x\rangle \longrightarrow\!\!> \langle\rangle$
  - $\langle\rangle \longrightarrow\!\!>$???
- Like in the case of a queue, "top()" and "pop()" may be combined into a single operation

Stack as a wrapper around List

- Stack ADT may be easily implemented as a wrapper around a List
- stk.push(x) —> lst.insert(x, lst.begin())
- stk.top() —> lst.retrieve(lst.begin())
- stk.pop() —> lst.delete(lst.begin())
- However, we will later learn about direct implementation of Stack ADT

Adapter Pattern

- Adapter pattern is where an abstraction is realized as a wrapper around existing structure

Adapter Pattern

- Adapter pattern is where an abstraction is realized as a wrapper around existing structure
- Stack abstraction can be easily implemented as a wrapper around python list

Adapter Pattern

- Adapter pattern is where an abstraction is realized as a wrapper around existing structure
- Stack abstraction can be easily implemented as a wrapper around python list
- Why don't we directly use a list? Why build a wrapper?

Adapter Pattern

- Adapter pattern is where an abstraction is realized as a wrapper around existing structure
- Stack abstraction can be easily implemented as a wrapper around python list
- Why don't we directly use a list? Why build a wrapper?
- Python list supports adding a new element at one end ('append') and deleting an element at the same end ('pop'), which is ideal for the stack abstraction

Simple Stack Implementation

- Following the adapter pattern, a stack internally maintains a list

Simple Stack Implementation

- Following the adapter pattern, a stack internally maintains a list

```
class AdapterStack:

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

Simple Stack Implementation

- It should be straightforward to implement the basic stack methods: 'len', 'str', 'push', 'pop', 'top', 'isEmpty'

- It should be straightforward to implement the basic stack methods: 'len', 'str', 'push', 'pop', 'top', 'isEmpty'

```
class AdapterStack:

    def __len__(self):
        return ( len(self._items) )

    def __str__(self):
        return ( str(self._items) )
```

Simple Stack Implementation

- It should be straightforward to implement the basic stack methods: 'len', 'str', 'push', 'pop', 'top', 'isEmpty'

- It should be straightforward to implement the basic stack methods: 'len', 'str', 'push', 'pop', 'top', 'isEmpty'

```
class AdapterStack:

    def isEmpty(self):
        return ( len(self._items) == 0 )

    def push(self, ele):
        self._items.append(ele)
```

Simple Stack Implementation

- It should be straightforward to implement the basic stack methods: 'len', 'str', 'push', 'pop', 'top', 'isEmpty'

Simple Stack Implementation

- It should be straightforward to implement the basic stack methods: 'len', 'str', 'push', 'pop', 'top', 'isEmpty'

```
class Empty(Exception):
    pass

class AdapterStack:

    def top(self):
        if ( len(self._items) == 0 ):
            raise Empty("Stack is empty!")
        return self._items[-1]

    def pop(self):
        if ( len(self._items) == 0 ):
            raise Empty("Stack is empty!")
        return self._items.pop()
```

Array-based Stack Implementation

Array-based Stack Implementation

- Array-based stack implementation is very similar to that of list implementation

Array-based Stack Implementation

- Array-based stack implementation is very similar to that of list implementation

```
import ctypes

class ArrayStack:

    def __init__(self, cap=16):
        self._top = 0
        self._capacity = cap
        self._items = (ctypes.py_object * cap)()
```

Array-based Stack Implementation

- It should be straightforward to implement the basic stack methods: 'len', 'str', 'push', 'pop', 'top', 'isEmpty'

Array-based Stack Implementation

- It should be straightforward to implement the basic stack methods: 'len', 'str', 'push', 'pop', 'top', 'isEmpty'

```
class ArrayStack:

    def __len__(self):
        return self._top

    def isEmpty(self):
        return (self._top == 0 )
```

Array-based Stack Implementation

- It should be straightforward to implement the basic stack methods: 'len', 'str', 'push', 'pop', 'top', 'isEmpty'

Array-based Stack Implementation

- It should be straightforward to implement the basic stack methods: 'len', 'str', 'push', 'pop', 'top', 'isEmpty'

```
class ArrayStack:

    def push(self, item):
        if (self._top == self._capacity):
            self._resize(2 * self._capacity)
        self._items[self._top] = item
        self._top += 1
```

Array-based Stack Implementation

- It should be straightforward to implement the basic stack methods: 'len', 'str', 'push', 'pop', 'top', 'isEmpty'

Array-based Stack Implementation

- It should be straightforward to implement the basic stack methods: 'len', 'str', 'push', 'pop', 'top', 'isEmpty'

```
class ArrayStack:

    def top(self):
        if ( self._top == 0 ):
            raise Empty("Stack is empty!")
        return self._items[self._top - 1]
```

Array-based Stack Implementation

- It should be straightforward to implement the basic stack methods: 'len', 'str', 'push', 'pop', 'top', 'isEmpty'

Array-based Stack Implementation

- It should be straightforward to implement the basic stack methods: 'len', 'str', 'push', 'pop', 'top', 'isEmpty'

```
class ArrayStack:

    def pop(self):
        if ( self._top == 0 ):
            raise Empty("Stack is empty!")
        item = self._items[self._top - 1]
        self._top -= 1
        self._items[self._top] = None
        if (self._top < (self._capacity // 4)):
            self._resize(self._capacity // 2)
        return item
```

Queue

- Queue, as the name suggests, is a special kind of a list where insertions happen only at the end and deletions happen only at the front
- It has only two recognized positions namely “front” and “last”
- It is also referred to as First-in-First-out (FIFO) structure

Queue Operations

- q = Queue() — creates and returns a new Queue structure
- q.isEmpty() — returns "TRUE" if q is empty and returns "FALSE" otherwise
- q.clear(q) — removes all the objects in q and resets it to an empty queue

Enqueue

- q.enqueue(x) — add the object x to the Queue q
  - $\langle\rangle$ —> $\langle x \rangle$
  - $\langle e_1, e_2, \cdots, e_n \rangle$ —> $\langle e_1, e_2, \cdots, e_n, x \rangle$

Enqueue

- q.enqueue(x) — add the object x to the Queue q
  - $\langle\rangle$ —> $\langle x \rangle$
  - $\langle e_1, e_2, \cdots, e_n \rangle$ —> $\langle e_1, e_2, \cdots, e_n, x \rangle$
- There may be a capacity limit for a queue and in that case, enqueue may not always succeed

Front and Dequeue

- q.front() — returns the first object in the queue q (q is NOT modified!)
  - $\langle e_1, e_2, \cdots, e_n \rangle$ —> e_1
  - $\langle \rangle$ —> ???

Front and Dequeue

- q.front() — returns the first object in the queue q (q is NOT modified!)
    - $\langle e_1, e_2, \cdots, e_n \rangle$ —> e_1
    - $\langle\rangle$ —> ???
- q.dequeue() — removes the first object from the queue q
    - $\langle e_1, e_2, \cdots, e_n \rangle$ —> $\langle e_2, \cdots, e_n \rangle$
    - $\langle x \rangle$ —> $\langle\rangle$
    - $\langle\rangle$ —> ???

Front and Dequeue

- q.front() — returns the first object in the queue q (q is NOT modified!)
  - $\langle e_1, e_2, \cdots, e_n \rangle$ —> e_1
  - $\langle \rangle$ —> ???
- q.dequeue() — removes the first object from the queue q
  - $\langle e_1, e_2, \cdots, e_n \rangle$ —> $\langle e_2, \cdots, e_n \rangle$
  - $\langle x \rangle$ —> $\langle \rangle$
  - $\langle \rangle$ —> ???
- It is also possible to combine both Front and Dequeue into a single operation
  - q.dequeue() — removes and returns the first object from the q

Queue as a wrapper around List

- Queue may be easily implemented as a wrapper around a List

Queue as a wrapper around List

- Queue may be easily implemented as a wrapper around a List
- q.enqueue(x) —> lst.insert(x, lst.end())

Queue as a wrapper around List

- Queue may be easily implemented as a wrapper around a List
- q.enqueue(x) —> lst.insert(x, lst.end())
- q.front() —> lst.retrieve(lst.begin())

Queue as a wrapper around List

- Queue may be easily implemented as a wrapper around a List
- q.enqueue(x) —> lst.insert(x, lst.end())
- q.front() —> lst.retrieve(lst.begin())
- q.dequeue() —> lst.delete(lst.begin())

Adapter Pattern

- Like stack, queue abstraction can also be implemented using the adapter pattern — as a wrapper around python list

Adapter Pattern

- Like stack, queue abstraction can also be implemented using the adapter pattern — as a wrapper around python list
- But, is it a good idea to do so?

Circular Array

Queue Full and Queue Empty Conditions

Array-based Queue Implementation

- Queue abstraction can be implemented by viewing the underlying array as a ‘circular array’

Array-based Queue Implementation

- Queue abstraction can be implemented by viewing the underlying array as a ‘circular array’

```
import ctypes

class CircArrayQueue:

    def __init__(self, cap=256):
        self._capacity = cap
        self._front = 0
        self._rear = 0
        self._items = (ctypes.py_object * cap)()
```

Array-based Queue Implementation

- We need a private method to find the 'next position', when the array is viewed as a 'circular array'

Array-based Queue Implementation

- We need a private method to find the 'next position', when the array is viewed as a 'circular array'

```
class CircArrayQueue:

    def _next(pos):
        return ( (pos + 1) % self._capacity )
```

Array-based Queue Implementation

- As discussed, it should be easy to check if the queue is empty or full

Array-based Queue Implementation

- As discussed, it should be easy to check if the queue is empty or full

```
class CircArrayQueue:

    def isFull(self):
        return ( self._front == self._next(self._rear)

    def isEmpty(self):
        return ( self._front == self._rear )
```

Array-based Queue Implementation

- The core queue operations should also be easy to implement

Array-based Queue Implementation

- The core queue operations should also be easy to implement

```
class CircArrayQueue:

    def enqueue(self, item):
        if ( self._front == self._next(self._rear) ):
            raise Full("The queue is already full!")
        self._items[self._rear] = item
        self._rear = self._next(self._rear)
```

Array-based Queue Implementation

- The core queue operations should also be easy to implement

Array-based Queue Implementation

- The core queue operations should also be easy to implement

```
class CircArrayQueue:

    def front(self):
        if ( self._front == self._rear ):
            raise Empty("The queue is empty!")
        return self._items[self._front]
```

Array-based Queue Implementation

- The core queue operations should also be easy to implement

Array-based Queue Implementation

- The core queue operations should also be easy to implement

```
class CircArrayQueue:

    def dequeue(self):
        if ( self._front == self._rear ):
            raise Empty("The␣queue␣is␣empty!")
        item = self._items[self._front]
        self._items[self._front] = None
        self._front = self._next(self._front)
        return item
```

Array-based Queue Implementation

- How do we find the length of the current queue?

Array-based Queue Implementation

- How do we find the length of the current queue?
- If ‘rear’ is ahead of 'front', then the length is $\mathrm{rear} - \mathrm{front}$

Array-based Queue Implementation

- How do we find the length of the current queue?
- If ‘rear’ is ahead of ’front’, then the length is $\mathrm{rear} - \mathrm{front}$
- Else, length is $\mathrm{capacity} - (\mathrm{front} - \mathrm{rear})$

Array-based Queue Implementation

- How do we find the length of the current queue?
- If 'rear' is ahead of 'front', then the length is $\text{rear} - \text{front}$
- Else, length is $\text{capacity} - (\text{front} - \text{rear})$

```
class CircArrayQueue:

    def __len__(self):
        if ( self._front <= self._rear ):
            return (self._rear - self._front)
        else:
            return self._capacity - (self._front - sel
```

Double-Ended Queues

- Double-ended Queue (pronounced as "deck") is an abstraction where enqueue and dequeue operations are possible at both the ends

Double-Ended Queues

- Double-ended Queue (pronounced as "deck") is an abstraction where enqueue and dequeue operations are possible at both the ends
- It is a generalization of both stack and queue abstractions, since it can also be viewed as permitting push and pop operations at both the ends

Double-Ended Queues

- deq.addFirst(item)
- deq.addLast(item)
- deq.deleteFirst()
- deq.deleteLast()
- deq.first()
- deq.last()
- deq.isEmpty()
- deq.isFull() [This is required only when the capacity is fixed]
- len(deq)

Double-Ended Queues

- Implementation of double-ended queue is left as an exercise!

Summary

- We have discussed array based implementations of Stack and Queue ADTs

What next?

- We will explore link-based implementations of List, Stack, and Queue ADTs
- We will look at some of the applications of Lists, Stacks, and Queues