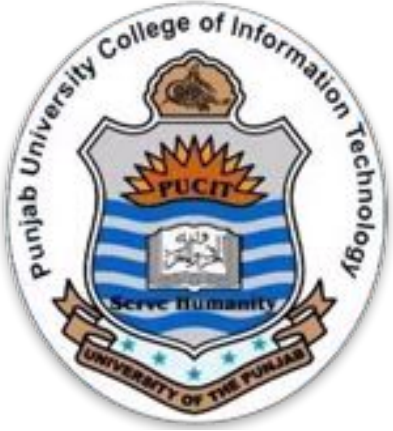
STACKS AND QUEUES

Data Structures and Algorithms
Waheed Iqbal

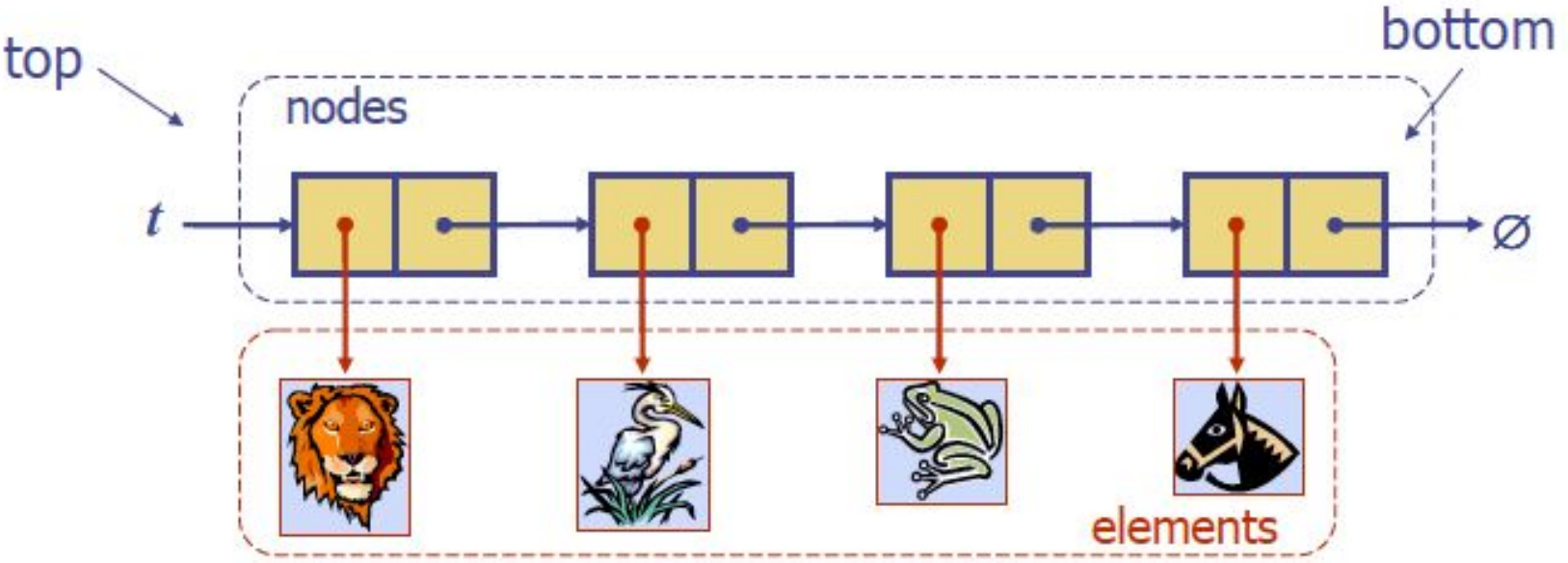


Punjab University College of Information Technology (PUCIT)
University of the Punjab, Lahore, Pakistan.

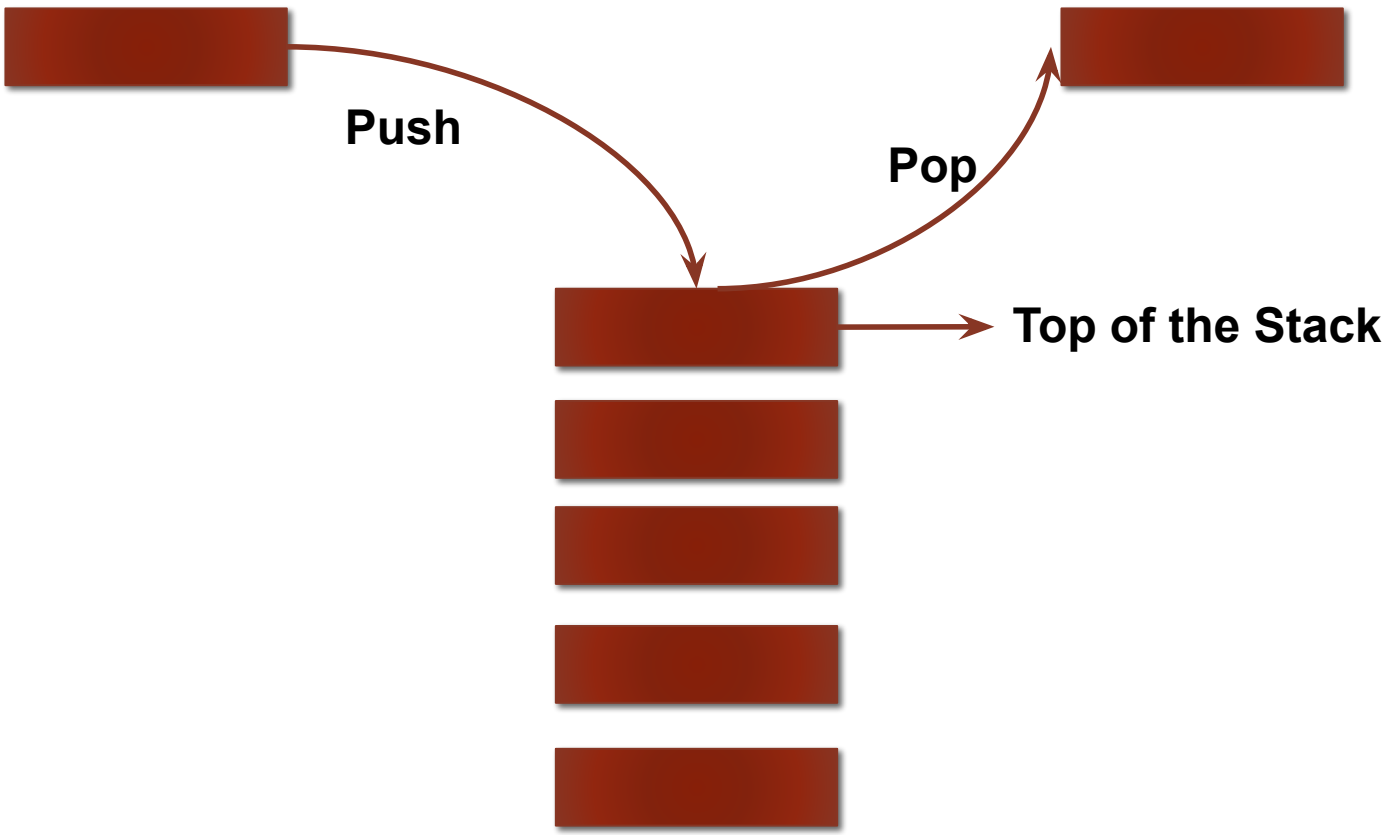
Stack

- Stack is a data structure that allows access to items in a last in first out (LIFO) style
- Main Stack operation:
  - push(object): insert an element to the stack
  - pop(): return the last inserted element and remove it
- Auxiliary stack operations:
  - top() / peek(): return the element on top of the stack (last inserted element)
  - size(): return the number of elements stored
  - isEmpty(): return a boolean value indicating elements are store or not in the stack

Stack Implementation Using Linked List



Stack (Cont.)



Stack Example

Operation	output	stack
push(8)	-	(8)
push(3)	-	(3, 8)
pop()	3	(8)
push(2)	-	(2, 8)
push(5)	-	(5, 2, 8)
top()	5	(5, 2, 8)
pop()	5	(2, 8)
pop()	2	(8)
pop()	8	()
pop()	"error"	()
push(9)	-	(9)
push(1)	-	(1, 9)

Python list can be used as stack!

```
stack = []

# append() function to push
# element in the stack
stack.append('a')
stack.append('b')
stack.append('c')

print('Initial stack')
print(stack)

# pop() function to pop
# element from stack in
# LIFO order
print('\nElements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('\nStack after elements are popped:')
print(stack)
```

Python list can be used as stack!

```
stack = []

# append() function to push
# element in the stack
stack.append('a')
stack.append('b')
stack.append('c')

print('Initial stack')
print(stack)

# pop() function to pop
# element from stack in
# LIFO order
print('\nElements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('\nStack after elements are popped:')
print(stack)
```

Python needs to reallocate memory to grow the underlying list for accepting new items, these operations are slower and can become ***O*(*n*)**

Stack Implementation Using Linked List

Consider the following classes and implement the functions:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class StackLinkedList:
    def __init__(self):
        self.top = None

    def is_empty(self):

    def push(self, data):

    def pop(self):

     def peek(self):
```

```
stack = StackLinkedList()
stack.push(1)
stack.push(2)
stack.push(3)

print("Top of the stack:", stack.peek())
print("Popped item:", stack.pop())
print("Top of the stack after pop:", stack.peek())
```

Stack Implementation Using Linked List

Consider the following classes and implement the functions:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class StackLinkedList:
    def __init__(self):
        self.top = None

    def is_empty(self):

    def push(self, data):

    def pop(self):

     def peek(self):
```

```
def is_empty(self):
    return self.top is None

def push(self, data):
    new_node = Node(data)
    new_node.next = self.top
    self.top = new_node

def pop(self):
    if self.is_empty():
        return "Stack Underflow
    data = self.top.data
    self.top = self.top.next
    return data

def peek(self):
    if self.is_empty():
        return "Stack is empty"
    return self.top.data
```

Stack Implementation Using Linked List

Consider the following classes and implement the functions:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class StackLinkedList:
    def __init__(self):
        self.top = None

    def is_empty(self):

    def push(self, data):

    def pop(self):

     def peek(self):
```

Write a function which returns the size of the stack?

Applications of Stack

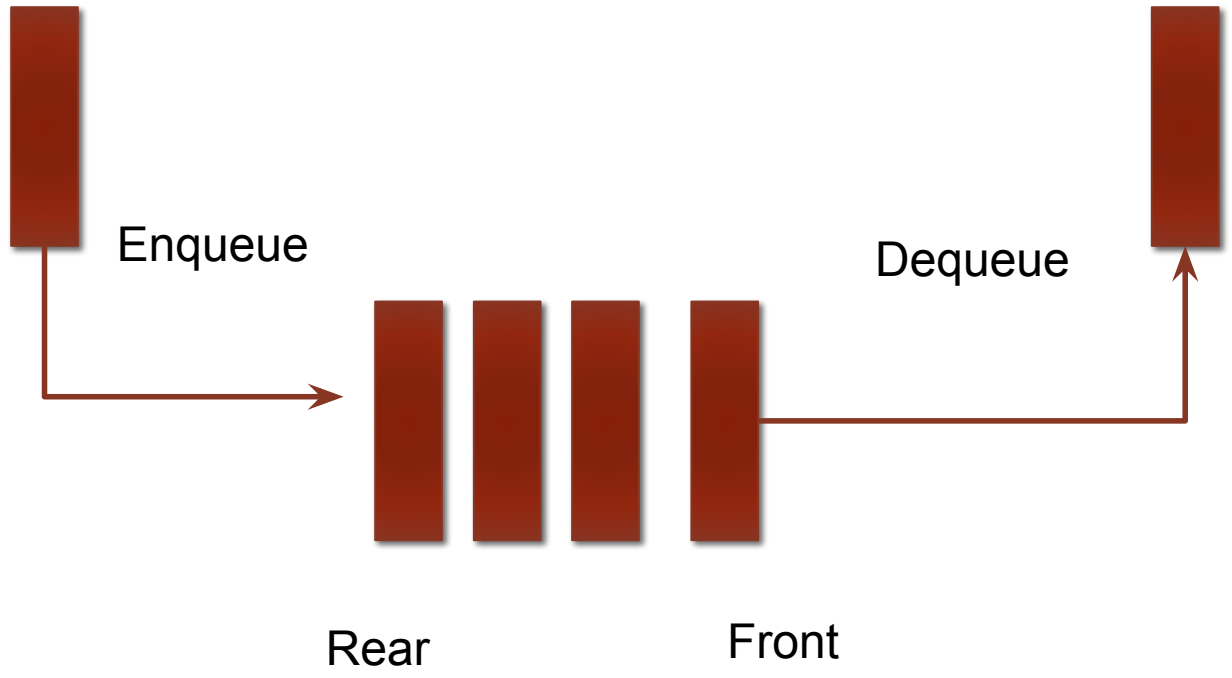
- Reversing data
- Detecting unmatched parentheses
- Page-visited history in a Web browser
- Undo sequence in a text editor
- Implementing recursion

Many other you may need to explore!

Queue

- Queue is a data structure that allows access to items in a first in, first out style (FIFO)
- Main Operations:
  - **enqueue (item)**: add to the queue)
  - **dequeue ():** remove the *oldest* item in the queue
- Auxiliary Operations:
  - **front()**: returns the element at the front without removing it
  - **size():** returns the number of elements stored
  - **isEmpty():** returns a Boolean value indicating whether no elements are stored

Queue (Cont.)



Queue Example

Operation	output	queue
enqueue(5)	-	(5)
enqueue(3)	-	(5, 3)
dequeue()	5	(3)
enqueue(7)	-	(3, 7)
dequeue()	3	(7)
front()	7	(7)
dequeue()	7	()
dequeue()	"error"	()
isEmpty()	true	()
enqueue(9)	-	(9)
size()	1	(9)

Application of Queue

- Waiting lists e.g., customer checkout on a point of sale counter

- Access to shared resources e.g., printer

- Queue in Python can be implemented using **deque** class from the collections module.

- **Deque** is preferred over list in the cases where we need quicker **append** and **pop** operations, as deque provides an **O(1)** time complexity for **append** and **pop** operations as compared to **list which provides O(n)** time complexity, whenever resizing is done!

- Instead of enqueue and deque, append() and popleft() functions are used.

```
from collections import deque
q = deque()
q.append('a')
q.append('b')
q.append('c')
print("Initial queue")
print(q)
print("\nElements dequeued from the queue")
print(q.popleft())
print(q.popleft())
print(q.popleft())

print("\nQueue after removing elements")
print(q)
```

Queue using Linked List

```python
class QueueNode:
    def __init__(self, data):
        self.data = data
        self.next = None

class QueueLinkedList:
    def __init__(self):
        self.front = None
        self.rear = None

    def is_empty(self):
        return self.front is None

    def enqueue(self, data):
        new_node = QueueNode(data)
        if self.is_empty():
            self.front = new_node
            self.rear = new_node
        else:
            self.rear.next = new_node
            self.rear = new_node

    def dequeue(self):
        if self.is_empty():
            return "Queue Underflow"
        data = self.front.data
        self.front = self.front.next
        if self.front is None:
            self.rear = None
        return data

    def peek(self):
        if self.is_empty():
            return "Queue is empty"
        return self.front.data

# Example usage:
queue = QueueLinkedList()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)

print("Front of the queue:", queue.peek())
print("Dequeued item:", queue.dequeue())
```

Palindromes

Palindromes are words which can be read same from forward and reverse. Few examples are:

- Radar
- Mom
- Dad
- Stats
- Madam
- Wassamassaw

How we may use Stack and Queue to determine a given word is palindrome?

Some Problems Solve Easily Using Stack/Queue

- Reverse a String: Reverse a given string.
- Valid Parentheses: Determine if a given string of parentheses is valid.
- Implementing Cache: Implement a cache with a fixed size using a queue.