Overview

• Data Structures
  • Stack
  • Queue
  • Deque
  • Lists
Objective

- Learn about different types of data structures and how they can be implemented in Python
- Learn about the interplay between data structure and algorithms
- Be able to implement data structures to enable the efficient performing of algorithms

Linear Data Structures

- Examples: stacks, queues, dequeues, and lists
- Data collections of ordered items
- Order depends on how items are added and removed
- Once added item stays in position
Linear Data Structures - Characteristics

- Two ends (left – right, front – rear)
- Linear structures distinguished by how items are added and removed
- E.g., additions of new items only allowed at the end (queue)
- Appear in many algorithms

Stack

- Ordered collection of items
- Addition/removal of items always takes place at same end (top)
- Base represents bottom and contains the item that has been in the stack the longest
- Most recently added to be removed first (last-in-first-out, LIFO)
- Top: newer items; bottom: lower items
Stack

- Stack of objects
  - What are suitable operations on a stack?
- Examples of stacks?

Applications Using Stacks

- Reverse order (e.g., letters in word)
  - Stack implements LIFO (last-in-first-out)
  - Reverse “algorithm”
Applications Using Stacks

- Check if delimiters are matched
- Matching of opening and closing symbols: {},[],(),
- Check: {{a}[b]{{c}(d(e)f)}(g)} and {{a}b(c)}

Applications Using Stacks

- Solve maze:
  - At intersection: down, left, up, right (but not back)
- Solve:
Stack - Abstract Data Type

- **Stack()** creates a new, empty stack; no parameters and returns an empty stack.
- **push(item)** adds a new item at top of stack; needs the item and returns nothing.
- **pop()** removes top item; needs no parameters, returns item, stack is modified.
- **peek()** returns top item from the stack but doesn’t remove it; needs no parameters, stack is not modified.
- **isEmpty()** test if stack is empty; needs no parameters, returns a boolean value.
- **size()** returns number of items on stack; needs no parameters; returns an integer.

Stack Example

<table>
<thead>
<tr>
<th>Stack Operation</th>
<th>Stack Contents</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.isEmpty()</td>
<td>[]</td>
<td>True</td>
</tr>
<tr>
<td>s.push(4)</td>
<td>[4]</td>
<td></td>
</tr>
<tr>
<td>s.push(&quot;dog&quot;)</td>
<td>[4,&quot;dog&quot;]</td>
<td></td>
</tr>
<tr>
<td>s.peek()</td>
<td>[4,&quot;dog&quot;]</td>
<td>'dog'</td>
</tr>
<tr>
<td>s.push(True)</td>
<td>[4,&quot;dog&quot;,True]</td>
<td></td>
</tr>
<tr>
<td>s.size()</td>
<td>[4,&quot;dog&quot;,True]</td>
<td>3</td>
</tr>
<tr>
<td>s.isEmpty()</td>
<td>[4,&quot;dog&quot;,True]</td>
<td>False</td>
</tr>
<tr>
<td>s.push(8.4)</td>
<td>[4,&quot;dog&quot;,True,8.4]</td>
<td></td>
</tr>
<tr>
<td>s.pop()</td>
<td>[4,&quot;dog&quot;,True]</td>
<td>8.4</td>
</tr>
<tr>
<td>s.pop()</td>
<td>[4,&quot;dog&quot;]</td>
<td>True</td>
</tr>
<tr>
<td>s.size()</td>
<td>[4,&quot;dog&quot;]</td>
<td>2</td>
</tr>
</tbody>
</table>
Implementing Stack in Python

class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items) - 1]

    def size(self):
        return len(self.items)

Stack Class in Action

from Stack import Stack

s = Stack()
print(s.isEmpty())
s.push(4)
s.push('dog')
print(s.peek())
s.push(True)
print(s.size())
print(s.isEmpty())
s.push(8.4)
print(s.pop())
print(s.pop())
print(s.size())
Implementation as a List?

- Implement list where top of stack is beginning of list
- pop() and append() no longer work

Stack as List

class StackList:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.insert(0, item)

    def pop(self):
        return self.items.pop(0)

    def peek(self):
        return self.items[0]

    def size(self):
        return len(self.items)
Stack Analysis

- Abstract data type
  - Change physical implementation while maintaining logical characteristics
- `append()` and `pop()` operations are both O(1)
- `insert(0)` and `pop(0)` operations both require O(n) for a stack of size n
- Very different timings

Queue

- Ordered collection of items
- Add items on one end
- Remove items on the other end
- Item starts at the rear and makes its way to the front => first-in first-out (FIFO)
Queue Examples

- Checkout line
- Printer queue
- Take-off at airport
- Queues in network equipment like switches and routers

Queue - Abstract Data Type

- `Queue()` creates a new, empty queue; no parameters and returns an empty queue.
- `enqueue(item)` adds a new item to rear of queue; needs the item and returns nothing.
- `dequeue()` removes front item; needs no parameters, returns item, queue is modified
- `isEmpty()` test if queue is empty; needs no parameters, returns a boolean value
- `size()` returns number of items in the queue; needs no parameters; returns an integer
Queue Example

<table>
<thead>
<tr>
<th>Queue Operation</th>
<th>Queue Contents</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>q.isEmpty()</td>
<td>[]</td>
<td>True</td>
</tr>
<tr>
<td>q.enqueue(4)</td>
<td>[4]</td>
<td></td>
</tr>
<tr>
<td>q.enqueue('dog')</td>
<td>['dog',4]</td>
<td></td>
</tr>
<tr>
<td>q.enqueue(True)</td>
<td>[True,'dog',4]</td>
<td></td>
</tr>
<tr>
<td>q.size()</td>
<td>[True,'dog',4]</td>
<td>3</td>
</tr>
<tr>
<td>q.isEmpty()</td>
<td>[True,'dog',4]</td>
<td>False</td>
</tr>
<tr>
<td>q.enqueue(8.4)</td>
<td>[8.4,True,'dog',4]</td>
<td></td>
</tr>
<tr>
<td>q.dequeue()</td>
<td>[8.4,True,'dog']</td>
<td>4</td>
</tr>
<tr>
<td>q.dequeue()</td>
<td>[8.4,True]</td>
<td>'dog'</td>
</tr>
<tr>
<td>q.size()</td>
<td>[8.4,True]</td>
<td>2</td>
</tr>
</tbody>
</table>

Implementing Queue in Python

class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        return self.items.pop() 

    def size(self):
        return len(self.items)
Queue Class in Action

```python
from Queue import Queue
q=Queue()
q.enqueue(4)
q.enqueue('dog')
q.enqueue(True)
print(q.size())
```

Hot Potato

After 5 passes, Brad is eliminated

Bill
pass to next person
david
and so on
Kent
until predefined counting constant
X

Jane
Implementation as Queue

```python
from Queue import Queue

def hotPotato(namelist, num):
    simqueue = Queue()
    for name in namelist:
        simqueue.enqueue(name)

    while simqueue.size() > 1:
        for i in range(num):
            simqueue.enqueue(simqueue.dequeue())

        simqueue.dequeue()

    return simqueue.dequeue()

print(hotPotato(['Bill', 'David', 'Susan', 'Jane', 'Kent', 'Brad'], 7))
```

Deque

- Very similar to queue
- BUT, items can be added at front and rear
- Same for removing items
- Does not require FIFO and LIFO!
Deque - Abstract Data Type

- Deque() creates a new, empty deque; no parameters and returns an empty queue.
- addFront(item) adds a new item to front of deque; needs the item and returns nothing.
- addRear(item) adds a new item to rear of deque; needs the item and returns nothing.
- removeFront() removes front item; needs no parameters, returns item, queue is modified.
- removeRear() removes rear item; needs no parameters, returns item, queue is modified.
- isEmpty() test if deque is empty; needs no parameters, returns a boolean value.
- size() returns number of items in the deque; needs no parameters; returns an integer.

Deque Example

<table>
<thead>
<tr>
<th>Deque Operation</th>
<th>Deque Contents</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>d.isEmpty()</td>
<td>[]</td>
<td>True</td>
</tr>
<tr>
<td>d.addRear(4)</td>
<td>[4]</td>
<td></td>
</tr>
<tr>
<td>d.addRear('dog')</td>
<td>['dog',4]</td>
<td></td>
</tr>
<tr>
<td>d.addFront('cat')</td>
<td>['dog',4,'cat']</td>
<td></td>
</tr>
<tr>
<td>d.addFront(True)</td>
<td>['dog',4,'cat',True]</td>
<td></td>
</tr>
<tr>
<td>d.size()</td>
<td>['dog',4,'cat',True]</td>
<td>4</td>
</tr>
<tr>
<td>d.isEmpty()</td>
<td>['dog',4,'cat',True]</td>
<td>False</td>
</tr>
<tr>
<td>d.addRear(8.4)</td>
<td>[8.4,'dog',4,'cat',True]</td>
<td></td>
</tr>
<tr>
<td>d.removeRear()</td>
<td>['dog',4,'cat',True]</td>
<td>8.4</td>
</tr>
<tr>
<td>d.removeFront()</td>
<td>['dog',4,'cat']</td>
<td>True</td>
</tr>
</tbody>
</table>
Implementing Deque in Python

class Deque:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def addFront(self, item):
        self.items.append(item)

    def addRear(self, item):
        self.items.insert(0, item)

    def removeFront(self):
        return self.items.pop()

    def removeRear(self):
        return self.items.pop(0)

    def size(self):
        return len(self.items)

Deque Class in Action

from Deque import Deque
d=Deque()
print(d.isEmpty())
d.addRear(4)
d.addRear('dog')
d.addFront('cat')
d.addFront(True)
print(d.size())
print(d.isEmpty())
d.addRear(8.4)
print(d.removeRear())
print(d.removeFront())
### Palindrome Checker

- String that reads the same forward and backward
  - Radar
  - Hannah
  - Madam, I'm Adam
- Use deque and check letters on both ends
- Remove letters until different or only one left

#### Palindrome Checker Code

```python
from Deque import Deque

def palchecker(aString):
    chardeque = Deque()
    for ch in aString:
        chardeque.addRear(ch)
    stillEqual = True
    while chardeque.size() > 1 and stillEqual:
        first = chardeque.removeFront()
        last = chardeque.removeRear()
        if first != last:
            stillEqual = False
    return stillEqual

print(palchecker("lsdkjfskf"))
print(palchecker("radar"))
```
Lists

- Used Python lists to implement abstract data types
- Not included by all programming languages
- List is collection of items where each holds relative position (first item, second item, third item, etc.)
- Simple unsorted list for scores: 54, 26, 93, 17, 77, 31

Unordered List Abstract Data Type

- `List()` creates a new, empty list; no parameters and returns an empty list.
- `add(item)` adds a new item to the list; needs the item and returns nothing; assume item is not already in list
- `remove(item)` removes item from list; needs item and modifies list; assume item is present in list
- `search(item)` searches for item in list; needs item, returns a boolean value
- `isEmpty()` test if list is empty; needs no parameters, returns a boolean value
- `size()` returns number of items in list; needs no parameters; returns an integer
Unordered List Abstract Data Type

- **append(item)** adds a new item to end of list; needs the item and returns nothing; assume item is not already in list
- **index(item)** returns position of item in list; needs item and returns index; assumes item is in list
- **insert(pos, item)** adds a new item to list at position pos; needs item, returns nothing; assume item is not already in list and enough exiting items to have position pos
- **pop()** removes and returns last item; needs nothing and returns item; assume list has at least one item
- **pop(pos)** removes and returns item at position pos; needs position and returns item; assume item is in the list

Linked List

![Diagram of a linked list](image)
Node Class

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

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self, newdata):
        self.data = newdata

    def setNext(self, newnext):
        self.next = newnext
Unordered List

• Build from collection of nodes
  • Each linked to the next by reference
  • Must contain reference to first node

```python
class UnorderedList:
    def __init__(self):
        self.head = None
```

Unsorted List

```python
>>> mylist = UnorderedList()
```

• Initially

• Eventually
Unordered List

def isEmpty(self):
    return self.head == None

>>> mylist.add(31)
>>> mylist.add(77)
>>> mylist.add(17)
>>> mylist.add(93)
>>> mylist.add(26)
>>> mylist.add(54)

• Since item 31 was entered first, it will be
  last element in list
• 54 will be first

Unordered List Add

def add(self, item):
    temp = Node(item)
    temp.setNext(self.head)
    self.head = temp

• Result when
  lines 3 and 4
  are swapped
**Unsorted List: Linked List Traversal**

```python
def size(self):
    current = self.head
    count = 0
    while current != None:
        count = count + 1
        current = current.getNext()
```

**Unsorted List: Search**

```python
def search(self, item):
    current = self.head
    found = False
    while current != None and not found:
        if current.getData() == item:
            found = True
        else:
            current = current.getNext()
    return found
```
Unsorted List: Search

```python
>>> mylist.search(17)
```

Unsorted List: Remove

- Two step approach
  - Find
  - Remove
- How to remove node?
  - Remove node
  - Use previous and current
def remove(self, item):
    current = self.head
    previous = None
    found = False
    while not found:
        if current.getData() == item:
            found = True
        else:
            previous = current
            current = current.getNext()

    if previous == None:
        self.head = current.getNext()
    else:
        previous.setNext(current.getNext())
**Unsorted List: Remove**

If list used above were ordered: 17, 26, 31, 54, 77, 93

**Ordered List**

- If list used above were ordered: 17, 26, 31, 54, 77, 93
Ordered List

- **OrderedList()** creates a new, empty ordered list; no parameters and returns an empty list.
- **add(item)** adds a new item to the list; preserves order; needs the item and returns nothing; assume item is not already in list
- **remove(item)** removes item from list; needs item and modifies list; assume item is present in list
- **search(item)** searches for item in list; needs item, returns a boolean value
- **isEmpty()** test if list is empty; needs no parameters, returns a boolean value
- **size()** returns number of items in list; needs no parameters; returns an integer

Ordered List

- **index(item)** returns position of item in list; needs item and returns index; assumes item is in list
- **pop()** removes and returns last item; needs nothing and returns item; assume list has at least one item
- **pop(pos)** removes and returns item at position pos; needs position and returns item; assume item is in the list
Ordered List

class OrderedList:
    def __init__(self):
        self.head = None

• isEmpty and size implement as in unordered list
• Likewise remove will work as expected
• search and add require modification

Ordered List: Search

def search(self, item):
    current = self.head
    found = False
    stop = False
    while current != None and not found and not stop:
        if current.getData() == item:
            found = True
        else:
            if current.getData() > item:
                stop = True
            else:
                current = current.getNext()
    return found
Ordered List: Add

```python
def add(self, item):
    current = self.head
    previous = None
    stop = False
    while current != None and not stop:
        if current.getData() > item:
            stop = True
        else:
            previous = current
            current = current.getNext()

    temp = Node(item)
    if previous == None:
        temp.setNext(self.head)
        self.head = temp
    else:
        temp.setNext(current)
        previous.setNext(temp)
```

Ordered List: Add
Binary Search

• Take greater advantage of ordered list with clever comparisons
• Start examining the middle item
  • The one searching for -> done
  • Item searching for $\geq$ middle item -> eliminate lower half and middle item from search
• Continue process with upper half

Example for finding 54
**Binary Search**

```python
def binarySearch(alist, item):
    first = 0
    last = len(alist)-1
    found = False

    while first<=last and not found:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
            found = True
        else:
            if item < alist[midpoint]:
                last = midpoint-1
            else:
                first = midpoint+1

    return found

# Example usage

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42]
print(binarySearch(testlist, 3))
print(binarySearch(testlist, 13))
```

**Binary Search: Analysis**

- Each comparison eliminates ~1/2 of remaining items from consideration
- Start with \( n \) items, \( n/2 \) items left after 1\(^{st}\) comparison (second \( n/4 \), third \( n/8 \), etc.)
- \( l^{th} \) item \( \Rightarrow \frac{n}{2^l} \)
- \( O(\log n) \)
Next Steps

• Next lecture on Thursday
• HW2 will be posted on Thursday