Lists

Intro2CS – week 8b
Stack ADT (Abstract Data Type)

• A container with 3 basic actions:
  – push(item)
  – pop()
  – is_empty()

• Semantics:
  – Push inserts an item into the stack
  – Pop will remove and return an item (if the stack isn’t empty). The item that is returned: the last item inserted (that was not already removed)
  – Is_empty will return “True” if every item that has been inserted, was also removed. “False” otherwise

• LIFO (last-in-first-out)
Example

push(1) → [1]
pop() → 1 → []
push(2) → [2]
push(3) → [3,2]
pop() → 3 → [2]
push(4) → [4,2]
push(5) → [5,4,2]
pop() → 5 → [4,2]
pop() → 4 → [2]
pop() → 2 → []
A simple implementation

class Stack1:
    def __init__(self):
        self.__lst = []

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

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

    def is_empty(self):
        return len(self.__lst) == 0

stack = Stack1()
stack.push(1)
stack.pop()
stack.push(2)
print(stack.is_empty())
Using stacks

- Stacks are useful data structures. They can be used to solve interesting algorithmic problems.

- For example, to turn recursive questions into loops
Linked Lists
Python lists in memory

• Python lists are represented as continuous blocks in memory.

• If we want to make the block size larger, we have to re-allocate memory and copy the contents.

• Many times Python does this for us behind the scenes
Linked Lists

- Have many blocks instead of one
- Chain them: remember address of the next one
- Now we can easily add blocks
Linked Lists

class Node:
def __init__(self, data=None, next=None):
    self.data = data
    self.next = next
How to work with linked lists

• To access the data in the linked list:

  head.data
  head.next.data
  head.next.next.data
  ...

• But this is cumbersome.
How to work with linked lists

• Instead, we usually work with a moving reference

```python
cur = head
print(cur.data)
cur = cur.next
print(cur.data)
...
```

![Diagram of linked list with cur and head references]
Iterating over lists

• So in order to print a linked list:

```python
cur = head
while cur!=None:
    print(cur.data)
    cur = cur.next
```
Inserting into a list

- Inserting before the head:

```python
new_node = Node("new val", head)
head = new_node
```

Or simply write:
```
head = Node("new val", head)
```
Inserting into a list

- Inserting at position `cur`:

```
cur.next = Node("some val", cur.next)
```

- Notice that insertion takes $O(1)$, while in Python lists (arrays) it takes $O(n)$
Removing from a list

• Removing the first item:

```
head = head.next
```

![Diagram showing the removal of the first item from a list](image)
Removing from a list

- Removing the node after position cur:

  \[
  \text{cur.next} = \text{cur.next.next}
  \]

- What if we want to remove the node at position cur?
Some list operations are more costly.

- How do we get to the node before the current one?

- How quickly can we get to the k’th node in the list?

- How quickly can we print the list in reverse?
  - What if we only have $O(1)$ memory to use?
Doubly-linked lists

• We can also create lists with pointers leading both ways

• Easier to walk one step back
• Operations like insert and remove (a bit) more complex
• Takes up a little more space in memory
A stack using a linked list

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

class Stack:
    def __init__(self):
        self.__head = None

    def push(self, item):
        self.__head = Node(item, self.__head)

    def pop(self):
        result = self.__head.data
        self.__head = self.__head.next
        return result

    def is_empty(self):
        return self.__head == None
Improving the stack class

• To easily convert the stack into strings:

```python
def __str__(self):
    cur = self.__head
    result = ""
    while cur!=None:
        result = result+str(cur.data)
        if cur.next!=None:
            result = result+","
        cur = cur.next
    return "["+ result +"]"
```
Recursion and linked lists

• Recursion is often very natural on linked lists.
  • They have a “recursive” structure: If you remove the head of a linked list, you have a shorter list.

• Example:
  – Let’s compute the size of our stack recursively.

```python
def get_size(self):
    def recursive_size(list_head):
        if list_head == None:
            return 0
        else:
            return recursive_size(list_head.next)+1
    return recursive_size(self.__head)
```
Finding an item in the list

• E.g., in our stack

```python
def find_ver1(self, item):
    cur = self.__head
    while cur:
        if cur.data == item:
            return True
        cur = cur.next
    return False

def find_ver2(self, item):
    return _find_helper(self.__head, item)
```
Finding an item (version 2)

• A recursive way:

```python
def find_ver2(self, item):
    return _find_helper(self.__head, item)

def _find_helper(node, item):
    if not node:
        return False

    if node.data == item:
        return True

    else:
        return _find_helper(node.next, item)
```