What We Will See Today:

- Linked lists
- Graphs
- Trees
- Iterators and Generators
Linked Lists

- Separate the logical order of items from their physical order in memory
- Each item points to the location of the next item
- Dynamically expands list as needed
- Linked lists may also be used to implement other data structures, like queues and stacks
- Always remember that when a link to an item is destroyed – the item can not be reached!
The two most common mistakes regarding lists are:

- Disconnecting without keeping a pointer to the head/tail of the list.
- Trying to get the next of the trailing None.
class Node:
    def __init__(self, data=None, next=None):
        self.__data = data
        self.__next = next

    def __str__(self):
        return str(self.__data)

    def get_data(self):
        return self.__data

    def get_next(self):
        return self.__next

    def set_data(self, data):
        self.__data = data

    def set_next(self, next):
        self.__next = next
class LinkedList:
    def __init__(self, head=None):
        self.__head = head

    def get_head(self):
        return self.__head

    def is_empty(self):
        return self.__head == None

    def add(self, new_head):
        new_head.set_next(self.__head)
        self.__head = new_head
class LinkedList:

def __len__(self):
    current = self.__head
    count = 0
    while current is not None:
        count = count + 1
        current = current.get_next()
    return count

def index(self, item):
    ix = 0
    current = self.__head
    while current != None:
        if current.get_data() == item:
            return ix
        else:
            ix = ix+1
        current = current.get_next()
    return -1
class LinkedList:

def rev_list1(self):
    current = self.__head
    self.__head = None
    while current is not None:
        next = current.get_next()
        self.add(current)
        cur = next
Can we do any better?

Improved implementation of linked lists

Add:

- a length variable
  - in init: self.__length = 0
  - update in add/remove
- append?
  - hold a pointer to the tail
- going backwards?
  - doubly linked list
Doubly-linked list with a tail

- Sometimes it is convenient that we can add and remove items from both ends of the linked-list (e.g. if we want to use it both as a stack and a queue)
- To support this methods we will need a doubly-linked list with both head and a tail
A doubly-linked list with a tail

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

class DoublyLinkedList:
    def __init__(self):
        self.__head = self.__tail = None
        self.__length = 0
def add_first(self, node):
    if self.__head is None:
        # list was empty
        self.__tail = node
    else:
        # connect old head to new node
        self.__head.prev = node;
        node.next = self.__head

    # update head
    self.__head = node
    self.__length +=1
def add_last(self, node):
    if self.__tail is None:
        # list was empty
        self.__head = node
    else:
        # connect old tail to new node
        self.__tail.next = node;
        node.prev = self.__tail
        # update tail
        self.__tail = node
    self.__length+=1
A doubly-linked list with a tail

def remove_first(self):
    d = self.__head.data
    self.__head = self.__head.next
    if self.__head is None: # list is now empty
        self.__tail = None
    else: # disconnect old head
        self.__head.prev.next = None
        self.__head.prev = None
    self.__length -=1
    return d
A doubly-linked list with a tail

```python
def remove_last(self):
    d = self.__tail.data
    self.__tail = self.__tail.prev
    if self.__tail is None:  # list is now empty
        self.__head = None
    else:  # disconnect old tail
        self.__tail.next.prev = None
        self.__tail.next = None
    self.__length -= 1
    return d;
```

What does this method assume?

That the list is not empty
Stacks and Queues

- A stack is a last in, first out (LIFO) data structure
  - Items are removed from a stack in the reverse order from the way they were inserted

- A queue is a first in, first out (FIFO) data structure
  - Items are removed from a queue in the same order as they were inserted
Queues (LIFO): What would be the API?

- what are the functions required in order to use it?

```python
class MyQueue:
    def __init__(self):
        # Initialize the queue

    def enqueue(self, item):
        # Add an item to the queue

    def dequeue(self):
        # Remove an item from the queue

    def get_size(self):
        # Get the size of the queue
```

```python
class MyQueue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)
        else:
            return None

    def get_size(self):
        return len(self.queue)
```
Implementing Queue

class MyQueue:
    def __init__(self):
        self.__head = None
        self.__tail = None
        self.__size = 0

    def get_size(self):
        return self.__size

    def is_empty(self):
        if self.__size > 0:
            return False
        else:
            return True
def enqueue(self, item):
    if self.__tail == None:
        self.__head = Node(item)
        self.__tail = self.__head
    else:
        # adding new item to the end of the list
        self.__tail.next = Node(item)
        self.__tail = self.__tail.next
    self.__size += 1
Removing an item from the queue

```python
def dequeue(self, item):
    result = self.__head.data
    self.__head == self.__head.next
    if self.__head == None:
        self.__tail = None
        self.__size -= 1
    return result
```
Question from the exam 2015:

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

נתונה المتحלקה Node המ (_) ייצגת איבר ברשימה מקושרת (סacock).

לתרשים שנוי אובייקטים מטיפוס Node, שנמצאים ב X-1, Y-1, שהם ראשית של שתי רשתות מקושרות.

האודות: נאמר כי רשתות מתלכדות אם כל איברי.Meshauptim בשני הרשתות.

לדוחה, הרשתות ביאור מספר 1 מתחברותлежаי האיברים האחוזים שייכים לקבוצת X-1, Y-1.

שימו לב כי רשתות מתלכדות, לח ה어서ור האישורים המשותפים, כל האיברים הבודדים貝 הם המשותפים.

.node_x is node_y מ- X-1, Y-ו X-1, Y-ו node_x מ- X-1, Y-ו Meshauptim, X-1, Y-ו Meshauptim.

לתרשים: אם x ייצג X-1, Y-ו x ייצג X-1, Y-ו Meshauptim.
Q from the exam

```python
# 2.1

def len_list(head):
    length = 0
    while head is not None:
        length += 1
        head = head.next

    return length
```
Q from the exam

```python
def node_i(head, idx):
    # 2.2
    if idx < 0:  # If the index is negative then idx = len+idx
        idx = len_list(head) + idx
        if idx < 0:
            return None

    while idx > 0 and head is not None:
        head = head.next
        idx -= 1
    return head
```
Q from the exam

ב (7 נקודות) המדריך: ראשית המ推動 X היא דב של רישמה מקושטרה(dst)ックスת אם כל האיברים של Y

Moshe ב-X, כי ישיבר הרישמה Y במילים ייבא יibi הרישמה המקושטרה ב-Y.

באיור מספר 2 המזרכים Y היא דב של X.

איור מספר 2

הערה: ישמש לב כرشימה ריקה (None) (יאינה דב של רישמה אירת.

.head_y-1.head_x(head_y) לאריש של שטח רישמה מקושטרה

 mmc את הפונקציה (is_tail) שמקבלת שני ממצאים של שטח רישמה מקושטרה

.head_y(head_x) (יאובידס מטיפוס (Node) (יאבידס את הרישמה השכיחה (Y) כי דב של רישמה הרשונה

.Ahead (False-0, head_x, head_y) היא דב של head_y (יאמ True) (יאמ Head)

def is_tail(head_x, head_y):

# 2.3

```python
def is_tail(head_x, head_y):
    if not head_x or not head_y:
        return False

    # Run until we reached the end of x or found a shared Node
    # We note that it's enough that we found one joint node to declare tail
    while head_x is not None and head_x != head_y:
        head_x = head_x.next

    if not head_x:  # We reached the end of x
        return False

    return True
```

Q from the exam
Q from the exam

def find_joint_node(head_x, head_y):

2. (6 נקודות) ממשו את הפונקציה head_y-1 head_x שמקבלת שני مضיים x מ(mutexים find_joint_node שעון מ下面是小 השקה של רשימה (Node באובייקטים מ Geoffrey mezger) (Node ובוודקה אם הרשימות מתלכד הזירה בהודרה של הרשימות מתלכד הזירה בשאלת None אם הרשימות מתלכד הזירה על הפונקציה שהזריז המספר לעיניו למצב שלっていう הרשימה (Node אובייקט מitty) (Node cracked לעיל זהיר), אחות עליהلاحירי.

המשתמש שיתף הרשימות באובייקטים מקבילים (כאשר האיברים שונים 모습יים פעמים),

במדות שיתף הרשימות מתלכד באובייקטים שונים במקביל (כאשר האיברים שונים 모습יים פעמים). במושעשתمفיעל האיבריםakhirונ (שהוא Huckabee זה, אם הרשימות מתלכד ההזירה בשתייה.

def find_joint_node(head_x, head_y):

```python
# 2.4

```
# Linked Lists vs. Python’s List

<table>
<thead>
<tr>
<th>Operation</th>
<th>Linked List</th>
<th>Python’s list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert at head</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Insert at middle</td>
<td>O(n), O(1) given a reference</td>
<td>O(n)</td>
</tr>
<tr>
<td>Remove from head</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Remove from middle</td>
<td>O(n), O(1) given a reference</td>
<td>O(n)</td>
</tr>
<tr>
<td>Find k-th element</td>
<td>O(k)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Search unsorted</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Search sorted</td>
<td>O(n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>
A graph in mathematics and computer sciences consists of nodes (also known as vertices). Nodes may or may not be connected with one another by edges.
A *directed graph* is a graph where the edges have a direction.

For example, one can go from node A to node B directly, but has no way to go from node B to node A.
Finding Path in a Graph

- Lets say we want to get from one node to another.
- Each pair of connected nodes have a given distance between them.
- We want to know the total distance of the path.

For example here, one path from node a to node e:

a \rightarrow c \rightarrow e

And the distance is 6.

Note that this is not the only path from a to e.
Let's think...

- We start with an empty list of the path.
- Each time we get to next neighboring node, we can add it to the path, consider the distance we’ve made and continue from there.
- Recursion!
Implementation

```python
class Node:

    def __init__(self, data=None):
        self.__data = data
        self.__neighbors = set()

    def __str__(self):
        return self.__data

    def get_neighbors(self):
        return self.__neighbors

    def add_edge(self, other, distance):
        self.__neighbors.add(((other, distance)))
```

Each item in neighbors is the neighboring node and the distance between it and the current node.
def print_path(path):
    print('[', end=' ')
    for i, node in enumerate(path):
        if i == len(path) - 1:
            end_mark = ''
        else:
            end_mark = ' -> '
        print(node, end=end_mark)
    print(']')

def add_mutual_edge(node_a, node_b, distance):
    node_a.add_edge(node_b, distance)
    node_b.add_edge(node_a, distance)
```python
def find_path(start, end, path=None, distance=0):
    if not path:
        path = [start]
    else:
        path = path + [start]
    if start == end:
        return path, distance
    for node, curr_distance in start.get_neighbors():
        # check if we haven't visited that node already
        if node not in path:
            new_path, new_distance = find_path(node, end, path,
                                               distance + curr_distance)
            if new_path:
                return new_path, new_distance
    return None, 0
```
Building the graph and finding a path:

```python
a = Node('a')
b = Node('b')
c = Node('c')
d = Node('d')
e = Node('e')
f = Node('f')

add_mutual_edge(a, b, 2)
add_mutual_edge(a, c, 4)
add_mutual_edge(b, c, 3)
add_mutual_edge(b, f, 3)
add_mutual_edge(c, e, 2)
add_mutual_edge(c, d, 1)
add_mutual_edge(d, e, 1)
add_mutual_edge(e, f, 3)

path, dist = find_path(a, f)
print_path(path)
print("distance: ", dist)
```

[ a -> c -> d -> e -> f ]

distance: 9
Trees

- A Tree is a graph that has no cycles: there is only one path to get from one node to another.
- Directed trees have an hierarchical structure
Trees – Some Common Terminologies

- **Root** - the top node of the tree (no incoming edges)
- **A Parent** of a node is a node that has an edge from it to the node
- **Siblings** – Nodes with the same parent.
- **Leaf** – A node with no children.
- **Level** - The level of a node is defined by $1 + \text{(the number of connections between the node and the root)}$.
- **Depth** – The depth of a node is the number of edges from the node to the tree's root node.
A binary tree is a tree with the additional properties:

- Each node may have maximum 2 subtrees as its children
- All values in the left subtree of a node are smaller than the value of the node itself
- All values in the right subtree of a node are greater than the value of the node itself
- Each value in the tree can appear only once
This is another way of handling sorted data
class TreeNode:
    def __init__(self, value, left=None, right=None):
        self._value = value
        self._left = left
        self._right = right
def search(self, value):
    # returns True if value is in the tree
    if self.__value == value:
        return True
    else:
        if value < self.__value:
            if self.__left:
                return self.__left.search(value)
            else:
                return False
        else:
            if self.__right:
                return self.__right.search(value)
            else:
                return False
Searching — Runtime Analysis

How long does it take to search for an item?

The first node we are checking is the root

- Best case: the value we are looking for is at the root
  => $O(1)$

- Worst case: the value is at the leaf and all values in the tree are smaller or larger than the root
  => $O(n)$

- On a balanced tree (i.e., the depth of its leaves differ by maximum 1)
  => $O(\log(n))$
def insert(self, item):
    if self.__value == item:
        # we do nothing because the item is already here
        return
    else:
        if item < self.__value:
            if self.__left:
                self.__left.insert(item)
            else:
                self.__left = TreeNode(item)
        else:
            if self.__right:
                self.__right.insert(item)
            else:
                self.__right = TreeNode(item)
Same as in searching, we are actually searching for the right insertion position:

- Best case: $O(1)$
- Worst case: $O(n)$
- On a balanced tree: $O(\log(n))$
Iterators
Iteration motivation

- We already know strings, tuples, lists, sets and dictionaries.
- Those are all objects that may contain several items; those are all containers.
- So there are many types of objects which can be used with a for/while loop. These are called iterable objects.
- As the for loop is the most common use case for both iterators and generators, we will use that as our guideline.
What Is An Iterator?

- Iterator is an **object** that allows to traverse a **container**, and should support two main operations:
  - Pointing to a specific item in the collection.
  - Change its view by pointing to the next item in the collection.

- There are several types of iterators, but you should note that modifying the container while iterating on it is **dangerous**!
Iterators

- The build-in function `iter` takes an iterable object and returns an iterator:

  ```python
  >>> my_iter = iter([1,2,3])
  >>> my_iter
  <list_iterator object at 0x00000000024CC940>
  ```

- We can use `next()` method to get the next element

  ```python
  >>> next(my_iter)
  1
  >>> next(my_iter)
  2
  >>> next(my_iter)
  3
  ```
Iterators

- So we went over all items in the list. What will happen if we call `next` now?

```python
>>> next(my_iter)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- it raises a `StopIteration` - Be careful of such behaviors, iterators can be used only one time!
Making an iterable object on range

class RangeIter:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __iter__(self):
        return self

    def next(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
From the previous slides it seems that you can only iterate on built-in objects, or newly created objects (which is still a mystery).

Lets say we want to calculate the $n$’th prime, this could be done simply in a for loop right?

But what if we want to print every prime until the $n$’th one? Or we would like to do some more calculations on those numbers before printing them?

An iterator would be nice here, but is kind of an overkill.
Generators

- Until now, every function you created either returned a result or raised an exception, and thus in both cases – finished.
- Wouldn’t it be nice if we could tell the function to return intermediate values?
  - In our prime example, return every intermediate prime you see, and wait until called again to return the next one.
- Well, this is possible, and that is the power of a generator.
The “yield” keyword

- To create a generator function you simply need to use the *yield* keyword, instead of *return*.
- What happens in the background is that when using *yield* the state of the function is saved and a new value is generated.
  - Allowing to “remember” the current state of:
    - Variables (values)
    - Line of code (where we stopped)
  - Not like *return* where you cannot “remember” where you were in the function.
A simple generator example

```python
def integers():
    i = 1
    while True:
        yield i
        i += 1

def squares():
    for i in integers():
        yield i * i

def take(n, seq):
    seq = iter(seq)
    result = []
    try:
        for i in range(n):
            result.append(next(seq))
    except StopIteration:
        pass
    return result

print(take(5, squares()))

[1, 4, 9, 16, 25]
```

A call for those functions creates generator objects

This is a generator
A Fibonacci number is a number that corresponds to the formula:

\[ F_n = F_{n-2} + F_{n-1} \]

We would like to create a generator function which accepts two parameters (first and second numbers) and returns the next Fibonacci number.
Fibonacci example - 1

```python
def fibo(fibN_1, fibN_2):
    while True:
        yield fibN_1
        fibN_1, fibN_2 =
          fibN_2, fibN_1 + fibN_2
```

× But invoking the function gives this result:

```python
>>> fibo(0,1)
<generator object fibo at 0x031E9710>
```

× As we mentioned before, generators are used to generate results inside a for loop.

× So using this will work!

```python
for fib in fibo(0,1):
    print(fib)
```
What could go wrong with our example?

```python
def fibo(fibN_1,fibN_2):
    while True:
        yield fibN_1
        fibN_1,fibN_2 =
            fibN_2,fibN_1 + fibN_2

for fib in fibo(0,1):
    print(fib)
```

Well that’s an infinite loop! As the generator won’t stop generating numbers, the for loop will never stop.
To produce a stopping generator we ask for a maximal number to check:

```python
def fibo(fibN_1, fibN_2, fib_max):
    while True:
        if fibN_1 >= fib_max:
            raise StopIteration
        yield fibN_1
    fibN_1, fibN_2 =
    fibN_2, fibN_1 + fibN_2
```

Raising a `StopIteration` exception stops the generator.

- We could have also used:
  ```python
  while fibN_1 < fib_max:
  ```
Generators — a memory saver?

- In almost all cases, using a generator to generate data is more efficient, memory wise, than using a list or non-generator expressions.
- That is because when using a generator, each value is only generated upon demand.
  - Remember that generators are most frequently used in a for loop, to generate the data.
  - Meaning you don’t need to create the list you will work on in advance, you can do it on the fly!