Ceng 111 – Fall 2015
Week 13b

ADT

Credit: Some slides are from the “Invitation to Computer Science” book by G. M. Schneider, J. L. Gersting and some from the “Digital Design” book by M. M. Mano and M. D. Ciletti.
Queues

- **FIFO:**
  - First In First Out
  - The item that was inserted first is removed first.

- **Main operations:**
  - Add
  - Remove
Queues (cont’d)

Operations:
1. Add
2. Remove
3. Front/Peak
4. Is-Empty
5. Length
## Queues in Python

<table>
<thead>
<tr>
<th>Queue Operation</th>
<th>Corresponding Python Op.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>L.append(item)</td>
</tr>
<tr>
<td>Remove</td>
<td>L.pop(0)</td>
</tr>
<tr>
<td>Front/Peek</td>
<td>L[0]</td>
</tr>
<tr>
<td>Is-Empty</td>
<td>L == []</td>
</tr>
<tr>
<td>Length</td>
<td>len(L)</td>
</tr>
</tbody>
</table>
Queues: Formal Definition

\[ \text{add}(\text{item}, \text{queue}) \rightarrow \text{item} \in \text{queue} \]

- \( \text{new}() \rightarrow \emptyset \)
- \( \text{front}(\xi \in \emptyset) \rightarrow \xi \)
- \( \text{front}(\xi \in Q) \rightarrow \text{front}(Q) \)
- \( \text{remove}(\xi \in \emptyset) \rightarrow \emptyset \)
- \( \text{remove}(\xi \in Q) \rightarrow \xi \in \text{remove}(Q) \)
- \( \text{isempty}(\emptyset) \rightarrow \text{TRUE} \)
- \( \text{isempty}(\xi \in Q) \rightarrow \text{FALSE} \)
Priority Queue

- Similar to Queue except that the items in a queue has a priority value based on which they are kept in order!

- Operations:
  - `insert(item, priority)` → Push item with the given priority
  - `Highest()` → The item in the queue that has the highest priority
  - `Deletehighest()` → Delete the item that has the highest priority
  - `Is-Empty`
  - `Length`
Priority Queues in Python

Priority Queue Operation
- Insert
- Highest
- Delete highest
- Is-Empty
- Length

Corresponding Python Op.
- L.append(((item, priority)))
- Write a function that finds the max
- Write a function that finds the max and deletes it
- L == []
- len(L)
\[
\text{insert}(\text{item}, \text{PQ}) \quad \text{item} \sim \text{PQ}
\]

- \text{new()} \rightarrow \emptyset
- \text{highest}(\xi \sim \emptyset) \rightarrow \xi
- \text{highest}(\xi \sim \text{PQ}) \rightarrow
  \begin{cases} 
  \xi & \text{if priority}(\xi) > \text{priority}(\text{highest}(\text{PQ})) \\
  \text{highest}(\text{PQ}) & \text{else}
  \end{cases}
- \text{deletehighest}(\xi \sim \emptyset) \rightarrow \emptyset
- \text{deletehighest}(\xi \sim \text{PQ}) \rightarrow
  \begin{cases} 
  \text{PQ} & \text{if priority}(\xi) > \text{priority}(\text{highest}(\text{PQ})) \\
  \xi \sim \text{deletehighest}(\text{PQ}) & \text{else}
  \end{cases}
- \text{isempty}(\emptyset) \rightarrow \text{TRUE}
- \text{isempty}(\xi \sim \text{PQ}) \rightarrow \text{FALSE}
Trees
Example for Trees: Decision/Game Tree
Properties of Trees

- A tree is composed of nodes.
- A node can have either no branches, two branches or more than two branches.
- Binary tree: a tree where nodes have two branches.
- The depth of a tree:
  - The number of levels in the tree.
Binary Search Tree

- The nodes in the left branch of a node have less value than the node.
- The nodes in the right branch of a node have more value than the node.
How can we represent Trees in Python?

- Nested Lists vs. Tuples / Lists
- Nested Tuples vs. Dictionaries
Today

- Abstract Data Types
  - Trees
Let us open here a parenthesis for dictionaries

Python has `dict` data type for storing a set of key-value pairs:

```
{key_1:value_1, key_2:value_2, ..., key_N:value_N}
```

Key-value pairs are separated by comma and within curly braces.
Dictionaries: Accessing Elements

```python
person = {'age': 20, 'name': 'Veli'}

>>> person = {'age': 30, 'name': 'Veli'}
>>> person
{'age': 30, 'name': 'Veli'}
>>> person['ssn']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'ssn'
>>> person['ssn'] = 124
>>> person
{'age': 30, 'name': 'Veli', 'ssn': 124}
```
Dictionaries: Accessing Elements

```python
>>> person
{'age': 30, 'name': 'Veli', 'ssn': 124}
>>> person.keys()
['deli', 'age', 'name']
>>> person.values()
[30, 30, 'Veli']
```
Creating Dictionaries

- Enclosing a set of key-value pairs within curly braces, like 
  \{’age’:10, ’name’:'Ali’\}.

- Supplying as argument to the \texttt{dict()} function a nested list or a
  nested tuple of items where each item has two items. For example,
  \texttt{dict([[20, 30], [30, 40], [40, 50]])} creates the dictionary
  \{40: 50, 20: 30, 30: 40\}.

- Using the \texttt{input()} function as follows:

  \begin{verbatim}
  >>> d = input(’Give me a tuple:’)
  Give me a tuple:{’age’:20}
  >>> d
  {’age’: 20}
  \end{verbatim}
Looping Over Dictionaries

We can get:
- Keys
- Values
- Keys & Values

```python
P = {'a': 10, 'b': 20, 'c': 0}

# Take keys only
for k in P:  # P.keys() is also possible
    print k + " has value " + str(P[k])

# Take values
for v in P.values():
    print str(v) + " is a value"

# Take keys & values
for k, v in P.items():
    print k + " has value " + str(v)
```

Exercise: Find the key with the lowest value.
Modifying Dictionaries

```python
>>> person = {'age': 20, 'name': 'Ali'}
>>> del person['age']
>>> person
{'name': 'Ali'}
```

```
Dict['anahtar'] = <new_value>.
```
Now, let us see how we can represent Trees in Python

Using Lists:
- [10, [5, [3, [], []], [8, [], []]], [30, [], []]].
- [10, [5, [3, '#', '#'], [8, '#', '#']]], [30, '#', '#']], where the empty branches are marked with '#'.
- [10, [5, [3], [8]], [30]].
Now, let us see how we can represent Trees in Python

- Using dictionaries

```
Tree = {
    'value': 10,
    'left': {
        'value': 5,
        'left': {
            'value': 3,
            'left': {},
            'right': {}},
        'right': {
            'value': 8,
            'left': {},
            'right': {}}},
    'right': {
        'value': 30,
        'left': {},
        'right': {}}
}
```
Tree operations

- **datum()**
- **isempty()**
- **left()**
- **right()**
- **createNode()**

```python
# Return the value stored in the node
def datum(T):
    return T[0] # Assume nested list rep.

# Check whether the Tree is empty
def isempty(T):
    return T == [] # Assume nested list rep.

# Get the left branch
def left(T):
    #TODO: Throw exception if the tree is empty
    return T[1] # Assume nested list rep.

# Get the right branch
def right(T):
    #TODO: Throw exception if the tree is empty

# Create a node
def createNode(datum, left=[], right=[]):
    return [datum, left, right]
```
1. **Pre-order Traversal**

![Tree Diagram]

```
[10, [5, [3, [], []], [8, [], []]], [30, [], []]].
```

```python
1       def preorder_traverse(T):
2          if isempty(T):
3              return
4          print datum(T)
5          preorder_traverse(left(T))
6          preorder_traverse(right(T))
```
Traversing Trees

2. In-order Traversal

```python
def inorder_traverse(T):
    if isempty(T):
        return

    inorder_traverse(left(T))
    print datum(T)
    inorder_traverse(right(T))
```

3 5 8 10 30
Traversing Trees

3. Post-order Traversal

```
def postorder_traverse(T):
    if isempty(T):
        return
    postorder_traverse(left(T))
    postorder_traverse(right(T))
    print datum(T)
```

3 8 5 30 10
Yet another way to represent trees

Can you think of a one-dimensional representation where from the position of a node (call it n), the position of its parent (call it p) and children (call it c) can be calculated?

- In other words:
  - \( p = f(n) \) and \( c = g(n) \).

Heap:
- \( p = \text{round}(n/2) \)
- \( c = 2n \)
Binary **Search** Trees

```
   10
  /   \\
 5     30
/     /
3     8
```
def insert_node(T, value):
    '''Insert a node with value to the binary search tree'''
    if isempty(T):
        T.extend(createNode(value))
    elif datum(T) == value: #duplicate return
    elif value < datum(T):
        insert_node(left(T), value)
    else:
        insert_node(right(T), value)

# The following can construct the tree on the right
Tree = []
insert_node(Tree, 10)
insert_node(Tree, 30)
insert_node(Tree, 5)
insert_node(Tree, 3)
insert_node(Tree, 8)
Binary Search Trees: An example

```python
def search_tree(T, value):
    '''Search 'value' in binary search tree
    if isempty(T):
        return False
    elif datum(T) == value:
        return True
    elif value < datum(T):
        search_tree(left(T), value)
    else:
        search_tree(right(T), value)
```
Exercises

1. Write a function to determine the height of a binary tree.

2. Write a function to determine whether a binary tree is balanced (a tree $T$ is balanced if $|\text{height(left}(T)) - \text{height(right}(T))| \leq 1$ for every node in $T$)

3. Write a function to swap left and right branches of every node.

4. Write a function to count the leaves of a binary tree.