Lecture 14:
Data Structures: Linked Lists and Binary Trees

Instructors: Amiram Yehudai, Amir Rubinstein
Teaching Assistants: Michal Kleinbort, Yael Baran

School of Computer Science
Tel-Aviv University
Fall Semester 2016
http://tau-cs1001-py.wikidot.com
Lecture 13 : Highlights

• OOP
Lecture 14 - Plan

Data Structures

1. Linked Lists
2. Binary Search Trees

(next time: 3. hash tables)
Data Structures

- A **data structure** is a way to **organize** data in memory, to support various operations.

- **Operations** are divided into two types:
  - **Queries**, like search, minimum, etc.
  - **Mutations**, like insertion, deletion, modification.

- We have seen some built-in structures: strings, tuples, lists, dictionaries.
  In fact, "atomic" types, such as int or float, may also be considered structures, albeit **primitive** ones.

- The choice of data structures for a particular problem depends on desired operations and complexity constraints.

- The term **Abstract Data Type (ADT)** emphasizes the point that the user (client) needs to know what **operations** may be used, but not how they are **implemented**.
Data Structures (cont.)

- We will next implement a new data structure, called linked list, and compare it to Python's built-in list structure.

- Then we will discuss another linked structure, Binary Search Trees.

- Later in the course we will see additional non-built-in structures implemented by us (and you): hash tables, matrices.
Reminder: Representing Lists in python

• We have extensively used Python's built-in list object.

• “Under the hood", Python lists employ C's array. This means it uses a contiguous array of pointers: references to (addresses of) other objects.

• Python keeps the address of this array in memory, and its length in a list head structure.
• This makes accessing/modifying a list element, a[i], an operation whose cost is O(1) - independent of the size of the list or the value of the index:
  if the address in memory of lst[0] is \( a \), then the address in memory of lst[i] is simply \( a+i \). The fact that the list stores pointers, and not the elements themselves, enables Python's lists to contain objects of heterogeneous types.
Reminder: Representing Lists in Python, cont.

• However, the contiguous storage of addresses must be maintained when the list evolves.

• In particular if we want to insert an item at location i, all items from location i onwards must be “pushed” forward (O(n) operations in the worst case for lists with n elements).

• Moreover, if we use up all of the memory block allocated for the list, we may need to move items to get a block of greater size (maybe starting in a different location).

• Some cleverness is applied to improve the performance of appending items repeatedly; when the array must be grown, some extra space is allocated so the next few times do not require an actual resize.

• Source (with minor changes): How are lists implemented?
Linked Lists

• An alternative to using a contiguous block of memory, is to specify, for each item, the memory location of the next item in the list.

• We can represent this graphically using a boxes-and-pointers diagram:
Linked Lists Representation

• We introduce two classes. One for nodes in the list, and another one to represent a list.
• The class Node is very simple, just holding two fields, as illustrated in the diagram.

```python
class Node():
    def __init__(self, val):
        self.value = val
        self.next = None

    def __repr__(self):
        return str(self.value)
        # return "[" + str(self.value) + "," + str(id(self.next)) + "]"
        # THIS SHOWS POINTERS AS WELL - FOR DEBUGGING
```
class Linked_list ():
    def __init__ (self):
        self.next = None  # using same field name as in Node

    def __repr__ (self):
        out = ""
        p = self.next
        while p != None :
            out += str(p) + " "  # str envokes __repr__ of class Node
            p = p.next
        return out

More methods will be presented in the next slides.
Linked List Operations:

length

def length(self):
    p = self.next
    i=0
    while p != None:
        i+=1
        p = p.next
    return i

• The time complexity is $O(n)$, for a list of length $n$.

• Alternatively, we could keep another field, size, initialize it to 0 in __init__, and updated when inserting / deleting elements. Then we'd achieve $O(1)$ time for this operation.
Linked List Operations:
Insertion at the Start

def add_at_start (self , val ):
    p= self
    tmp =p. next
    p. next = Node (val)
    p. next . next = tmp

• Note: The time complexity is $O(1)$ in the worst case!
Linked List Operations:

**Insertion at a Given Location**

```python
def insert(self, val, loc):
    p = self
    for i in range(0, loc):
        p = p.next
    tmp = p.next
    newNode = Node(val)
    p.next = newNode
    newNode.next = tmp
```

- The argument `loc` must be between 0 and the length of the list (otherwise a run time error will occur).
- When `loc` is 0, we get the same effect as `add_at_first`

- Time complexity: \(O(\text{loc})\). In the worst case \(\text{loc} = n\).
Linked List Operations: Access

```python
def get_item (self, loc):
    p = self.next
    for i in range(0, loc):
        p = p.next
    return p.value
```

- The argument loc must be between 0 and the length of the list (otherwise a run time error will occur).

- Time complexity: $O(\text{loc})$. In the worst case $\text{loc} = n$. 
Linked List Operations: Find

def find (self, val):
    p = self.next
    #loc = 0  # in case we want to return the location
    while p != None:
        if p. value == val:
            return p
        else:
            p = p.next
            #loc = loc +1 # in case we want to return the location
    return None

• Time complexity: worst case $O(n)$, best case $O(1)$
Linked List Operations: **Delete**

```python
def delete (self, loc):
    ''' delete element at location loc '''
    p = self
    for i in range(0, loc):
        p = p.next
    if p!= None and p.next != None:
        p.next = p.next.next
```

- The argument `loc` must be between 0 and the length.
- Time complexity: $O(loc)$. In the worst case $loc = n$.

- The **Garbage collector** will “remove" the deleted item (assuming there is no other reference to it) from memory.
- Note: In some languages (e.g. C,C++) the programmer is responsible to explicitly ask for the memory to be freed.
Linked List Operations: Delete

• How would you delete an item with a given value (not location)?

• Searching and then deleting the found item presents a technical inconvenience: in order to delete an item, we need access to the item before it.

• A possible solution would be to keep a 2-directional linked list, aka doubly linked list (each node points both to the next node and to the previous one).

• This requires, however, O(n) additional memory (compared to a 1-directional linked list).

• You may encounter this issue again in the next HW assignment.
Using a linked list

Example:
Search for a certain item, and if found, increment it:

```python
x = lst.find(3)
if x!=None:
    x.value += 1
```
**Linked Lists vs. Python Lists:**

**Operations Complexity**

- **Insertion** after a given item requires $O(1)$ time, in contrast to $O(n)$ for Python lists.

- **Deletion** of a given item requires $O(1)$ time, (assuming we have access to the previous item) in contrast to $O(n)$ for Python lists.

- **Accessing** the $i$-th item requires $O(i)$ time, in contrast to $O(1)$ for Python lists.

- So far we assumed the lists are unordered. We will now consider **sorted** linked lists. What would be improved this way? What would not?
Sorted Linked Lists

- We can maintain an ordered linked list, by always inserting an item in its correct location. This version allows duplicates.

```python
def insert_ordered(self, val):
    p = self
    q = self.next
    while q != None and q.value < val:
        p = q
        q = q.next
    newNode = Node(val)
    p.next = newNode
    newNode.next = q
```
Searching in an Ordered linked list

• We cannot use binary search to look for an element in an ordered list.
• This is because random access to the i’th element is not possible in constant time in linked lists (as opposed to arrays such as Python’s lists).

```python
def find_ordered(self, val):
    p = self.next
    while p != None and p.value < val:
        p = p.next
    if p != None and p.value == val:
        return p
    else:
        return None
```

• We leave it to the reader to write a delete method for ordered lists.
From str to Linked_list

Here is an easy way to build a linked list from a given string:

```python
def string_to_linked_list(str):
    L = Linked_list()
    for ch in str[::-1]:
        L.add_at_start(ch)
    return L
```

```python
>>> string_to_linked_list("abcde")
[a,37371184] [b,37371152] [c,37371120] [d,37371088] [e,505280748]
```
Perils of Linked Lists

• With linked lists, we are in charge of memory management, and we may introduce cycles:
  >>> L = Linked_list()
  >>> L.next = L
  >>> L  #What do you expect to happen?

• Can we check if a given list includes a cycle?

• Here we assume a cycle may only occur due to the next pointer pointing to an element that appears closer to the head of the structure. But cycles may occur also due to the “content” field
Detecting Cycles: First Variant

```python
def detect_cycle1(lst):
    s = set()  # like dict, but only keys
    p = lst
    while True:
        if p == None:
            return False
        if p in s:
            return True
        s.add(p)
        p = p.next
```

• Note that we are adding the whole list element ("box") to the dictionary, and not just its contents.

• Can we do it more efficiently?

• In the worst case we may have to traverse the whole list to detect a cycle, so \(O(n)\) time in the worst case is inherent. But can we detect cycles using just \(O(1)\) additional memory?
Detecting cycles: Bob Floyd’s Tortoise and the Hare Algorithm (1967)

The hare moves twice as quickly as the tortoise. Eventually they will both be inside the cycle. The distance between them will then decrease by 1 at each additional step. When this distance becomes 0, they are on the same point on the cycle.

See demo on board.
Detecting cycles:
The Tortoise and the Hare Algorithm

def detect_cycle2(lst):
    # The hare moves twice as quickly as the tortoise
    # Eventually they will both be inside the cycle
    # and the distance between them will increase by 1 until
    # it is divisible by the length of the cycle.
    slow = fast = lst
    while True :
        if slow == None or fast == None:
            return False
        if fast.next == None:
            return False
        slow = slow.next
        fast = fast.next.next
        if slow is fast:
            return True

The same idea is used in Pollard's algorithm for factoring integers.
Testing the cycle algorithms
The python file contains a function introduces a cycle in a list.

```python
>>> lst = string_to_linked_list("abcde")
>>> lst
a b c d e

>>> detect_cycle1(lst)
False

>>> create_cycle(lst, 2, 4)

>>> detect_cycle1(lst)
True
```

```python
>>> detect_cycle2(lst)
True
```
Cycles in “Regular" Python Lists

As in linked lists, mutations may introduce cycles in Python's lists as well. In this example, either append or assign do the trick.

```python
>>> lst = ['a', 'b', 'c', 'd', 'e']
>>> lst.append(lst)
>>> lst
['a', 'b', 'c', 'd', 'e', [...]]
>>> lst = ['a', 'b', 'c', 'd', 'e']
>>> lst[3] = lst
>>> lst
['a', 'b', 'c', 'd', 'e']
>>> lst[1] = lst
>>> lst
['a', 'b', 'c', 'd', 'e']

We see that Python recognizes such cyclical lists and [...] is printed to indicate the fact.
Linked lists: additional issues

• Note that items of **multiple types** can appear in the same list.

• Some programming languages require homogenous lists (namely all elements should be of the same type).
Linked data structures

• Linked lists are just the simplest form of linked data structures, we can use pointers to create structures of arbitrary form.

• For example doubly-linked lists, whose nodes include a pointer from each item to the preceding one, in addition to the pointer to the next item.

• Another linked structure is binary trees, where each node points to its left and right child. We will see it now and also how it may be used as search trees.

• Another linked structure is graphs (probably not in this course).
Trees

- **Trees** are useful models for representing different physical or abstract structures.

- Trees may be defined as a special case of **graphs**. The properties of graphs and trees are discussed in the course Discrete Mathematics (and used in many courses).

- We will only discuss a common form of so called **(rooted) trees**.
Graphs

- A graph is a structure with **Nodes** (or vertices) and **Edges**. An edge connects two nodes.
- In **directed graphs**, edges have a direction (go from one node to another).
- In **undirected graphs**, the edges have no direction.

Example: undirected graph.
Drawing from wikipedia
(Rooted) Trees – Basic Notions

• A **directed edge** refers to the edge from the **parent** to the **child** (the arrows in the picture of the tree)
• The **root node** of a tree is the (unique) **node** with no parents (usually drawn on top).
• A **leaf node** has no children.
• Non leaf nodes are called **internal nodes**.

• The **depth** (or **height**) of a tree is the length of the longest path from the root to a node in the tree. A (rooted) tree with only one node (the root) has a depth of zero.

• A node $p$ is an **ancestor** of a node $q$ if $p$ exists on the path from the root node to node $q$.
• The node $q$ is then termed as a **descendant** of $p$.
• The **out-degree** of a node is the number of edges leaving that node.
• All the leaf nodes have an out-degree of 0.

Adapted from wikipedia
Example Tree

• Note that nodes are often labeled (for example by numbers or strings). Sometimes edges are also labeled.

• Here the root is labeled 2. The depth of the tree is 3. Node 11 is a descendent of 7, but not of (either of the two nodes labeled) 5.

• This is a binary tree: the maximal out-degree is 2.

Drawing from wikipedia
(Rooted) Binary trees

• (Rooted) binary trees are a special case of (rooted) trees, in which each node has at most 2 children (out-degree at most 2).

• We can also define binary trees recursively (as in the Cormen, Leiserson, Rivest, *Algorithms* book):

• A binary tree is a structure defined on a finite set of nodes that either
  - contains no nodes, or
  - is comprised of three disjoint sets of nodes:
    ➢ a root node,
    ➢ a binary tree called its left subtree, and
    ➢ a binary tree called its right subtree
Properties of Binary Trees

• The number of nodes in a binary tree of depth $h$ is at least $h+1$ and at most $2^{h+1}-1$.

• We will see the two extreme cases in the next slides.
Totally unbalanced binary tree

Each node has only one non empty subtree. The depth of totally unbalanced tree with \( n \) nodes is \( h = n - 1 \)

Complete (Rooted) Binary Trees

All the leaves are in the same level, and all internal nodes have degree exactly 2. A complete binary tree of depth $h$ has $2^h$ leaves and $2^h - 1$ internal nodes. So $n = 2^{h+1} - 1$
Applications of Trees

- Trees may be used to represent:
  - arithmetic expressions
  - programs in a given programming language
  - evolutionary history
  - and more.

- They can also represent the execution of programs: the nodes represent individual function executions, and the edges show the calls invoked from a given execution. A recursive function will appear multiple times in the same graph. It is a useful tool for visualizing algorithms, and also useful in the context of compilers.
Example: **Call Tree** for Computing Fibonacci $f(n)$

Drawing from http://i.stack.imgur.com/oPTFd.png
Binary Search Trees

- Binary search trees are data structures used to represent collections of data items. They support operations like insert, search, delete, etc.

- Each node in a binary search contains a single data record. As before, we will assume the record consists of a key and value. A node will also include pointers to its left and right subtrees.

- The keys in the binary search tree are organized so that every node satisfies the property shown in the next slide.
Binary search property

In each node, all the keys in the left/right subtrees are smaller/larger than the key in the current node, respectively.

Left subtree, all keys < node.key
Right subtree, all keys > node.key
Example binary search trees

Only the keys (and the left and right pointers) are shown. The nodes also contain the value associated with the key, but these are not shown here.
(for example, keys are IDs, and values are names.)

Drawing from wikipedia
Binary Search Tree: python code
A tree node will be represented by a class Tree_node:

class Tree_node():
    def __init__(self, key, val):
        self.key = key
        self.val = val
        self.left = None
        self.right = None

    def __repr__(self):
        return "[" + str(self.left) + "
        " (" + str(self.key) + "," + str(self.val) + ") " \n        + str(self.right) + "]"
Binary Search Tree: *lookup*

In this version, a tree will be represented simply by a *pointer to the root*, and the operations on the tree will be written as functions. Note that the functions are *recursive*.

```python
def lookup(root, key):
    if root == None:
        return None
    elif key == root.key:
        return root.val
    elif key < root.key:
        return lookup(root.left, key)
    else:
        return lookup(root.right, key)
```
Binary Search Tree: \textit{insert}

We first look for the insert location (similar to lookup), and then hang the new node as a leaf.

For example:

Inserting into empty tree:

\begin{itemize}
  \item \textit{root} = \text{None}
  \item \text{Insert} 26
  \item \text{root}
\end{itemize}
Binary Search Tree: `insert`

- Insert is written as a *recursive* function. The return value is used to update the left or right pointer that was None.
- If the user inserts an element whose key is already in the tree, we assume that it should *replace* the one in the tree.

```python
def insert(root, key, val):
    if root == None:
        root = Tree_node(key, val)  # create a new leaf
    elif key == root.key:
        root.val = val  # update the val for this key
    elif key < root.key:
        root.left = insert(root.left, key, val)
    elif key > root.key:
        root.right = insert(root.right, key, val)
    return root
```
Binary Search Tree: Height

• To analyze the time complexity of tree operations, we should first consider the **shape of the tree**.

• Since most operations have to traverse a **path** from the root to some node(s), it is important to know how long such a path may be. So the **height of the tree** is an important factor in its performance.

• In particular, given the **size** of a tree $n$ (the number of elements stored in the tree == the number of nodes), the question is what is the **height** $h$ of the tree as a function of $n$. The smaller the better.

  • **Worst case**: when the tree is totally unbalanced, $h = n-1 = O(n)$.
  • **Best case**: we have seen that when the tree is perfectly balanced $n = 2^{h+1} - 1$, so $h = O(\log n)$. The height is $O(\log n)$ for trees that are not perfectly balanced, but are close to it. We call them **balanced trees** (you will meet them again in the Data Structures course).
Binary Search Tree: lookup time complexity

- The lookup algorithm follows a path from the root to the node where the element is found, or (when the element is not found) to a leaf.

- The time complexity of lookup is the length of the path from the root to the element we are looking for.

- The **best case** occurs when the element we are looking for is in the root. So the best case time complexity is \(O(1)\) (and is not dependent on the shape of the tree).

- The **worst case** occurs when we have to traverse a path from the root to a leaf. So the time complexity is the depth of the tree, and in the worst case (when the tree is totally unbalanced), it is \(O(n)\).

- The worst time complexity of lookup in balanced trees is \(O(\log n)\).
Binary Search Tree: insert time complexity

The insert algorithm is similar to lookup.

The best case is $O(1)$, for example when the element to be inserted is found in the root of the tree.

In a balanced tree, the worst case time complexity of insert is $O(\log n)$.

In arbitrary trees, the worst case time complexity of insert is $O(n)$. 
Binary Search Tree: Execution

```python
>>> bin_tree = None  # represents an empty tree
>>> bin_tree = insert(None, "b", 5)  # not in place
>>> bin_tree
[None (b,5) None]
>>> insert(bin_tree,"d",7)  # in-place
[None (b,5) [None (d,7) None]]
>>> lookup(bin_tree,"d")
7
>>> lookup(bin_tree,"e")
>>>                    # nothing returned
```

Note that when inserting to an empty tree, we must assign the result back to the variable pointing to the tree. This is not needed in later insertions, since the function mutates the relevant nodes. This unpleasant inconsistency can be removed using OOP (possible HW).

Note the way the tree is printed (look at Node.__repr__ )
Binary Search Tree: \texttt{min}

To compute the \textit{minimum key} in a tree, we need to go all the way to the left. For this, we maintain two pointers. (Alternatively, we could write a recursive function)

\begin{verbatim}
def min(node):
    if node == None:
        return None
    next = node.left
    while next != None:
        node = next
        next = node.left
    return node.key
\end{verbatim}

\textbf{complexity?}
Binary Search Tree: time complexity of min

The time complexity of min is the length of the path from the root to the leftmost node.

The best case occurs when the left subtree is empty (the left pointer in the root is None). In this case, the smallest item is at the root. The best case time complexity is $O(1)$.

The worst case occurs in a totally unbalanced tree in which all right subtrees are empty, (the tree is a “left chain”) so the length of path to the minimum is $n-1$, so the time complexity is $O(n)$.

The worst case in a balanced tree is $O(\log n)$. 
Binary Search Tree: depth

To compute the depth, we use a recursive function. Note that this is a non-linear recursion (two recursive calls). A tree with just a root node has depth 0, so that by convention an empty tree has depth -1.

```python
def depth(node):
    if node == None:
        return -1  # by convention
    else:
        return 1 + max(depth(node.left), depth(node.right))
```

Time complexity = size of the tree $O(n)$. This follows from the observation that every node is visited once. We can also write a recurrence relation: $T(n) = 1 + T(n_1) + T(n_2)$ where $n_1$ and $n_2$ are the sizes of the left and right trees, so that $n_1+n_2+1 = n$. $T(0)=0$. 
Binary Search Tree: size

Computing the size is similar to the depth. Again a recursive function.

```python
def size(node):
    if node == None:
        return 0
    else:
        return 1 + size(node.left) + size(node.right)
```

Time complexity = $O(n)$, same as depth.
Binary Search Tree: Another execution

```python
>>> T=Tree_node(3,"benny")

>>> insert(T,5,"amir")
[None (3,benny) [None (5,amir) None]]

>>> T
[None (3,benny) [None (5,amir) None]]

>>> insert(T,6,"samir")
[None (3,benny) [None (5,amir) [None (6,samir) None]]]  # unbalanced

>>> size(T)
3
>>> depth(T)
2
>>> lookup(T,2)  # nothing returned
>>> lookup(T,7)  # nothing returned
>>> lookup(T,3)
'benny'
```
Binary Search Tree: Concluding Remarks

• A function to delete a node is a little harder to write, and is omitted here.
• We can ask what is the average case time complexity of lookup and insert – this will be dealt with in the Data Structures course.
• If we are able to insure that the tree is always balanced, we will have an efficient way to store and search data. But we can observe that the shape of the tree depends on the sequence of inserts that generated the tree.
• It turns out that there are several variations of balanced binary search trees, such as AVL trees, and Red and Black trees, that insure that the tree remains balanced, by performing balancing operations each time an element is inserted (or deleted). This will also be taught in the Data Structures course.