Overview

- Linear Search
- Binary Search
- Binary Search Tree
**Objective**

- Understand the structure of trees
- Understand the principles and be able to implement binary search trees
- Be able to implement underlying data structures

**Searching**

- Algorithmic process of finding a particular item in a collection of items
- Returns **True** or **False**, and (sometimes) location of item

```python
>>> 15 in [3,5,2,4,1]
False
>>> 3 in [3,5,2,4,1]
True
```
Linear (Sequential) Search

• When data stored in collection such as list => linear or sequential relationship

• Each data item stored in position relative to others

• In Python: relative positions are index values of individual items

• Start at first item, move from item to item until:
  • item is found
  • run out of items => item NOT in list
Linear Search Implementation

```python
def sequentialSearch(alist, item):
    pos = 0
    found = False
    while pos < len(alist) and not found:
        if alist[pos] == item:
            found = True
        else:
            pos = pos + 1
    return found
```

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

Linear Search Analysis

- If item is not in list => $n$ comparisons
- If item is in list:
  - Best case: find item in first place
  - Worst case: $n$ comparisons
  - On average $n/2$ comparisons
    However for large $n$ => $O(n)$
Linear Search Analysis: Ordered

- So far items were randomly placed in list
- What happens if list is ordered?
  - No advantage if element is in list
  - If element is NOT in list search can abort after item is larger than the one searched for (e.g., search for “49”)

```
def orderedSequentialSearch(alist, item):
    pos = 0
    found = False
    stop = False
    while pos < len(alist) and not found and not stop:
        if alist[pos] == item:
            found = True
        else:
            if alist[pos] > item:
                stop = True
            else:
                pos = pos + 1
    return found
```

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42, 1]
print(orderedSequentialSearch(testlist, 3))
print(orderedSequentialSearch(testlist, 13))
### Linear Search Analysis: Ordered

<table>
<thead>
<tr>
<th>Case</th>
<th>Best Case</th>
<th>Worst Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item is present</td>
<td>1</td>
<td>n</td>
<td>n/2</td>
</tr>
<tr>
<td>Item is not present</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case</th>
<th>Best Case</th>
<th>Worst Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item is present</td>
<td>1</td>
<td>n</td>
<td>n/2</td>
</tr>
<tr>
<td>Item is not present</td>
<td>1</td>
<td>n</td>
<td>n/2</td>
</tr>
</tbody>
</table>

#### Binary Search

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

• Example for finding 54

```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

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
prompt(binarySearch(testlist, 3))
prompt(binarySearch(testlist, 13))
```
**Binary Search: Analysis**

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

**Tree Data Structures**

- New data structure
  - “2-dimensional” structure
  - Easy access (similar to binary search in array)
  - Easy insert and removal (similar to linked list)
- Trees
  - Consist of nodes and links
  - Are graphs without loops
  - Typically have a root node
Structure of Trees

- Nodes
  - Special node at top: root

- Links
  - Connect nodes
  - Zero or more nodes connected to a node

- Nodes can store information

Tree Terminology

- Root: top node
- Parent: node “above”
  - Every node (except root) has exactly one parent
- Child: node “below”
  - Nodes may have zero or more children
  - Binary trees have at most two children
- Leaf: node without children
- Subtree: tree below a given node
  - That node becomes root of the subtree
- Level: distance from root
Binary Trees

- Binary tree
  - Every node has at most two children
    - Left child
    - Right child
- Binary search tree
  - Nodes are arranged in a particular fashion
    - Left subtree has values smaller than node
    - Right subtree has values larger than node

Binary Search Tree

- Mapping from key to value
  - Binary search on list
  - Search on hash tables
- Binary search tree
  - Yet another way to map from key to value
  - Not interested in exact placement of item
  - Using binary tree structure for efficient searching
Map: Abstract Data Type

- **Map()** creates a new, empty map; returns an empty map collection.
- **put(key, val)** adds new key-value pair; if key already in map, replace old with new value
- **get(key)** returns value stored in map or **none** otherwise
- **del** delete key-value pair using statement \( \text{del map[key]} \)
- **len()** returns number of key-value pairs stored in map
- **in** returns **True** for statement \( \text{key in map} \), **False** otherwise

Map

- Benefit: given key look up associated data quickly
- Implementation that supports efficient search
- Hash table potentially \( O(1) \) performance
Search Tree Implementation

- BST property:
  - Key smaller than parent => left
  - Key larger than parent => right
- BST property will guide implementation based on Map ADT

BST Implementation

- Two classes: `BinarySearchTree` and `TreeNode`
- `BinarySearchTree` has reference to `TreeNode` that is root of tree
**BST Implementation**

- BST constructor and miscellaneous functions

```python
class BinarySearchTree:
    def __init__(self):
        self.root = None
        self.size = 0
    def length(self):
        return self.size
    def __len__(self):
        return self.size
    def __iter__(self):
        return self.root.__iter__()
```

**Tree Implementation**

```python
class TreeNode:
    def __init__(self, key, val, left=None, right=None, parent=None):
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent
    def hasLeftChild(self):
        return self.leftChild
    def hasRightChild(self):
        return self.rightChild
```

- Look at remainder in PyCharm
BST – Add Node

If a root node already in place, put calls private, recursive, helper function _put to search tree:

- Starting at the root of the tree, search the binary tree comparing the new key to the key in the current node. If the new key is less than the current node, search the left subtree. If the new key is greater than the current node, search the right subtree.
- When there is no left (or right) child to search, we have found the position in the tree where the new node should be installed.
- To add a node to the tree, create a new TreeNode object and insert the object at the point discovered in the previous step.

```python
def put(self, key, val):
    if self.root:
        self._put(key, val, self.root)
    else:
        self.root = TreeNode(key, val)
        self.size = self.size + 1

def _put(self, key, val, currentNode):
    if key < currentNode.key:
        if currentNode.hasLeftChild():
            self._put(key, val, currentNode.leftChild)
        else:
            currentNode.leftChild = TreeNode(key, val, parent=currentNode)
    else:
        if currentNode.hasRightChild():
            self._put(key, val, currentNode.rightChild)
        else:
            currentNode.rightChild = TreeNode(key, val, parent=currentNode)
```

BST – Put Method

- With the `put` method defined:
  - easily overload the `[ ]` operator for assignment by having the `__setitem__` method call the `put` method
  - allows to write Python statements like `myZipTree['Plymouth'] = 55446`, just like a Python dictionary.

```python
def __setitem__(self, k, v):
    self.put(k, v)
```

BST – Put Example

![BST Tree Diagram]

17

5

2 16

35

29 38

19 33
**BST – Get Method**

```python
def get(self, key):
    if self.root:
        res = self._get(key, self.root)
        if res:
            return res.payload
        else:
            return None
    else:
        return None

def _get(self, key, currentNode):
    if not currentNode:
        return None
    elif currentNode.key == key:
        return currentNode
    elif key < currentNode.key:
        return self._get(key, currentNode.leftChild)
    else:
        return self._get(key, currentNode.rightChild)
```

• With `__getitem__` can write a Python statement that looks just like accessing a dictionary, when in fact we are using a binary search tree, for example `ez = myZipTree['Fargo']`.  

```python
def __getitem__(self, key):
    return self.get(key)
```
**BST – Contains Method**

- Using `get`, implement the `in` operation by writing `__contains__` method.

```python
def __contains__(self, key):
    if self._get(key, self.root):
        return True
    else:
        return False
```

```python
if 'Northfield' in myZipTree:
    print("oom ya ya")
```

**BST – Delete**

- Most challenging method in BST
  - First find node in tree

```python
def delete(self, key):
    if self.size > 1:
        nodeToRemove = self._get(key, self.root)
        if nodeToRemove:
            self.remove(nodeToRemove)
            self.size = self.size - 1
        else:
            raise KeyError('Error, key not in tree')
    elif self.size == 1 and self.root.key == key:
        self.root = None
        self.size = self.size - 1
    else:
        raise KeyError('Error, key not in tree')

def __delitem__(self, key):
    self.delete(key)
```
BST – Delete

- Once key is found, three cases to consider:
  1. Node to be deleted has **no** children
  2. Node to be deleted has only **one** child
  3. Node to be deleted has **two** children

---

**BST – Delete, no Children**
BST – Delete, no Children

• Delete node and remove reference to it

```python
if currentNode.isLeaf():  # leaf
    if currentNode == currentNode.parent.leftChild:
        currentNode.parent.leftChild = None
    else:
        currentNode.parent.rightChild = None
```

BST – Delete, one Child

• Promote child to take place of parent

• Six cases two consider (basically three since they are symmetric):

1. If the current node is left child, update parent reference of left child to point to parent of current node, then update left child reference of parent to point to current node’s left child.

2. If the current node is right child, update parent reference of left child to point to parent of current node, then update right child reference of parent to point to current node’s left child.

3. If the current node has no parent, it must be the root.
else: # this node has one child
    if currentNode.hasLeftChild():
        if currentNode.isLeftChild():
            currentNode.leftChild.parent = currentNode.parent
            currentNode.parent.leftChild = currentNode.leftChild
        elif currentNode.isRightChild():
            currentNode.leftChild.parent = currentNode.parent
            currentNode.parent.rightChild = currentNode.leftChild
    else:
        currentNode.replaceNodeData(
            currentNode.leftChild.key,
            currentNode.leftChild.payload,
            currentNode.leftChild.leftChild,
            currentNode.leftChild.rightChild)

else:
    if currentNode.isLeftChild():
        currentNode.rightChild.parent = currentNode.parent
        currentNode.parent.leftChild = currentNode.rightChild
    elif currentNode.isRightChild():
        currentNode.rightChild.parent = currentNode.parent
        currentNode.parent.rightChild = currentNode.rightChild
    else:
        currentNode.replaceNodeData(
            currentNode.rightChild.key,
            currentNode.rightChild.payload,
            currentNode.rightChild.leftChild,
            currentNode.rightChild.rightChild)
BST – Delete, two Children

• Most difficult of the three cases
• If node has two children
  • Very unlikely that one can be promoted to take node’s place
• Search tree for node that can be used for replacement
  • Find node that preserves binary tree characteristics
  • Find node that has next largest key in tree => successor
• Successor is guaranteed to have not more than one child
BST – Delete, two Children

```python
elif currentNode.hasBothChildren():  # interior
    succ = currentNode.findSuccessor()
    succ.spliceOut()
    currentNode.key = succ.key
    currentNode.payload = succ.payload
```

Three cases to consider:

1. If node has a right child, then the successor is the smallest key in the right subtree.
2. If node has no right child and is left child of its parent, then parent is the successor.
3. If node is right child of its parent, and itself has no right child, then successor to this node is successor of its parent, excluding this node.

```python
def findSuccessor(self):
    succ = None
    if self.hasRightChild():
        succ = self.rightChild.findMin()
    else:
        if self.parent:
            if self.isLeftChild():
                succ = self.parent
            else:
                self.parent.rightChild = None
                succ = self.parent.findSuccessor()
                self.parent.rightChild = self
    return succ
```
**BST – Delete, two Children**

- `findMin` is called to find minimum key in subtree
  - Leftmost child of tree
  - Simply follows `leftChild` references

```python
def findMin(self):
    current = self
    while current.hasLeftChild():
        current = current.leftChild
    return current
```

**Next Steps**

- Next lecture on Thursday
- Discussion on Thursday
- Exam 1 is on 10/01, 7 pm – 9 pm