Dynamic Data Structures: Trees

ENGF0002: Design and Professional Skills

Mark Handley, University College London, UK

Term 1, 2018

The story so far

Python lists (aka arrays):

- **Fast append** to end
- **Slow addition** anywhere other than end
- **Slow deletion** anywhere other than end
- **Fast access** to data **by index** (ie position)
- **Slow access** to data **by value**
- If we need fast access by value, can sort the data, and then do **binary search.**

Linked lists

- **Fast append** to end
- **Fast addition** anywhere, if you know the **previous node**
- **Fast deletion** anywhere, if you know the **previous node**
- **Slow access** to data **by index**
- **Slow access** to data **by value**
- **Can't do binary search**, even on sorted data.

Storing and retrieving data by key

Motivation: keeping indexes for dynamic data structures.

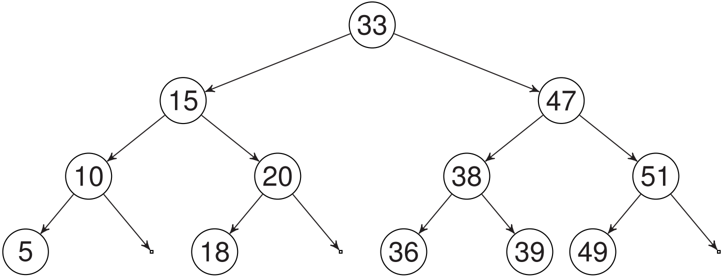
What next?

- Searching is faster in sorted structures. Binary search is $\mathcal{O}(\log N)$.
- However, the cost of sorting is $\mathcal{O}(N \cdot \log N)$.
- What to do when **adding or removing elements?** Sort again? No.
- Efficient data structures to maintain sorted sequences, and search in them

Key example: **binary sorted tree**, allowing $\mathcal{O}(\log N)$ **insert, remove and lookup**.

Trees and sorted trees.

A tree is composed of a set of nodes. Each contains a key (and possibly a value), and links to left and right child nodes.



Invariant of a sorted binary tree: at every node, **left child's key is less than or equal to the key of its parent**. Similarly, the right child's key is larger.

Representing a TreeNode as a Python class.

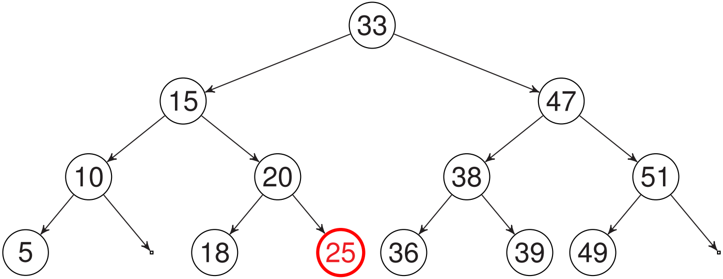
Each TreeNode is an instance of a class, containing four attributes.

```python
class TreeNode():
    def __init__(self, key, value):
        self.left = None
        self.right = None
        self.key = key
        self.value = value
```

By convention we represent a missing child as `None`.
We'll keep the tree sorted by **key**, while **value** can be any data we wish to find quickly using the key.

How to add an element to the Tree.

How to insert an item with key 25 to the tree?



Navigate from the root of the tree to the position where the item should be inserted, at a leaf, and place a new branch there with the item.

Adding a node

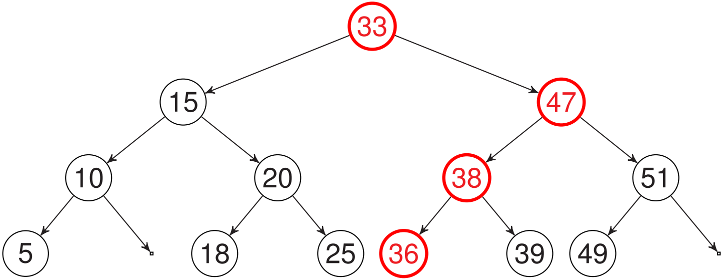
Follow the correct left or right branch according to the invariant. When it finds the first empty leaf (`None`) insert a new branch with the item.

```python
def add(self, node):
    if node.key < self.key:
        if self.left is None:
            self.left = node
        else:
            self.left.add(node)
    else:
        if self.right is None:
            self.right = node
        else:
            self.right.add(node)
```

Note the use of recursion; and the mutation of the tree.

Finding an item

How to find node with key 36?



Navigate from the root of the tree, going left if the key is less than the current node's key, right if it's greater.

Finding items.

A recursive **divide-and-conquer** algorithm for finding a node by key.

```python
    def find(self, key):
        if key == self.key:
            return self
        elif key < self.key:
            if self.left is None:
                return None
            else:
                return self.left.find(key)
        else:
            if self.right is None:
                return None
            else:
                return self.right.find(key)
```

Hiding the internal implementation from users

```python
class BinaryTree():
    def __init__(self):
        self.root = None
```

```python
    def add(self, key, value):
        node = TreeNode(key, value)
        if self.root is None:
            self.root = node
        else:
            self.root.add(node)
        self.count += 1
```

```python
    def get(self, key):
        if self.root is None:
            return None
        node = self.root.find(key)
        if node is None:
            return None
        return node.value
```

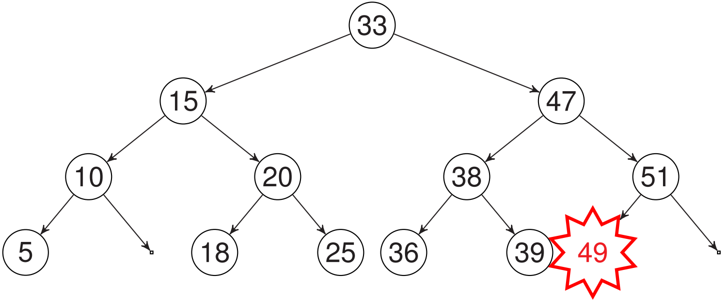
Hiding the internal implementation from users

Users don’t want to see how a binary tree is implemented. They just want an API that is simple to use:

```
tree = BinaryTree()
tree.add("Karp", ("x30406", "MPEB 6.20"))
tree.add("Handley", ("x37679", "MPEB 6.21"))
tree.add("Vissichio", ("x31397", "MPEB 6.19"))
phone, office = tree.get("Handley")
assert phone == "x37679"
```

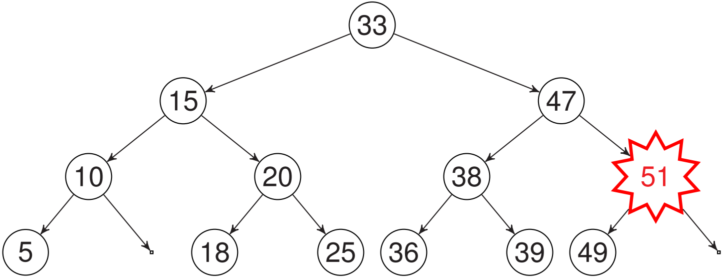
Removing items — Case 1: removing a leaf.

When removing a leaf (node with no children, eg. 49) we can simply replace it with `None` in the parent.



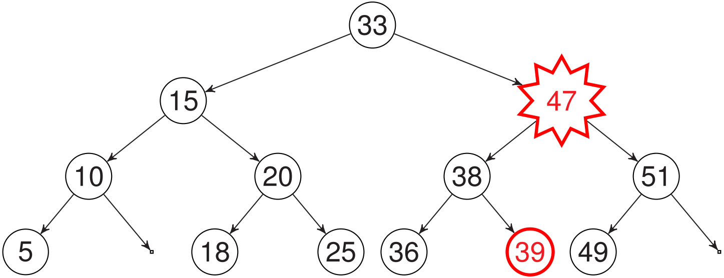
Case 2 & 3: remove an incomplete branch.

When the node to be removed has only one child (eg. 51), we can simply ‘promote up’ that child.



Case 4: a full branch

Removing a node with two children (eg. 47) leaves a ‘hole’ in the tree. We need to substitute the node with another one. A good candidate for a substitude is the node in the left subtree with the largest key (eg. 39).



The remove API

```
        assert(tree.get(2) is None)
        assert(tree.get(3) is None)
        tree.delete(1)  # delete something that's no there
        assert(tree.get(1) is None)
        assert(tree.get(2) is None)
        assert(tree.get(3) is None)

    def test_demo():
        tree = BinaryTree()
        tree.add("Karp", ("x30406", "MPEB 6.20"))
        tree.add("Handley", ("x37679", "MPEB 6.21"))
```

The BinaryTree implementation

```
class BinaryTree():
    def __init__(self):
        self.root = None
```

```
    def delete(self, key):
        if self.root is None:
            return
        self.root = self.root.delete(key)
```

Almost all the work happens in the TreeNode’s `delete()` method.

TreeNode’s delete() implementation.

```
    def delete(self, key):
        if key == self.key: # this node is being deleted
            if self.left is None and self.right is None:
                return None          # Case 1: it's a leaf
            elif self.left is None:
                return self.right    # Case 2: no left child
            elif self.right is None:
                return self.left     # Case 3: no right child
            else:                    # Case 4: two children
                maxkey = self.left.max()
                maxnode = self.left.find(maxkey)
                self.left = self.left.delete(maxkey)
                self.key = maxnode.key
                self.value = maxnode.value
        elif key < self.key:
            if self.left is not None:
                self.left = self.left.delete(key)
        else:
            if self.right is not None:
                self.right = self.right.delete(key)
        return self
```

Depth-first traversal involves returning items, by first returning all nodes in the left branch, then the item, and all items in the right branch. As the tree is sorted, items are returned in order.

In BinaryTree class:

```
def walk(self):
    if self.root is None:
        return
    yield from self.root.walk()
```

In TreeNode class:

```
def walk(self):
    if self.left is not None:
        yield from self.left.walk()
    yield (self.key, self.value)
    if self.right is not None:
        yield from self.right.walk()
```

This is a **generator** defined through keyword `yield`. The keyword `yield from` returns items from another generator until it's **exhausted**, then continues the execution. See `https://wiki.python.org/moin/Generators`.

Testing add and remove

We test at the level of the interface, not internal representation.

```python
def test_add_delete_len(two_lists):
    items = list(range(100))
    random.shuffle(items)   # randomly shuffled list from 0 to 99
    # Test init and add
    tree = BinaryTree()
    for i in items:
        tree.add(i,i)                    # Test add
    assert tree.root.max() == 99         # Test max

    # Test remove, get, len
    random.shuffle(items)
    count = len(items)
    for i in items:     # Helper iterator
        assert len(tree) == count   # Test len
        assert tree.get(i) is not None  # Test isin
        tree.delete(i)                  # Test remove
        assert tree.get(i) is None      # Test get
        count -= 1
```

Testing the iterator

We ensure that iterating returns elements in order.

```python
def test_walk(two_lists):
    items = list(range(100))        # [0, ..., 99]
    random.shuffle(items)           # shuffle

    tree = BinaryTree()
    for i in items:
        tree.add(i,str(i))

    assert len(items) == len(tree)    # Test len

    # Check that iterator returns elements in order
    count = 0
    for key,value in tree.walk():
        assert key == count  # Check the order is the same
        assert value == str(count)
        count += 1
```

Computational complexity.

Average case operation:

- **add**, **remove**, **get**: $\mathcal{O}(\log N)$
- **walk** walks the full tree: $\mathcal{O}(N)$

Advanced data structures: Balanced trees

The average case performance is only attained if the tree is ‘balanced’. However trees might get unbalanced under dynamic additions and deletions (consider adding elements in order). Balanced binary trees are needed to ensure balance is maintained for $\mathcal{O}(\log N)$ operations.

The builtin `dict` type.

The Python dictionary type provides a very efficient key-value store.

- the `dict` function takes a sequence of tuples and returns a dictionary mapping them as key values.
- There is a shorthand for dict literals, eg. `{k1:v1, k2:v2}`
- Get and set operations are as in minimap, eg. `m[k] = v` for set, and `m[k]` for get.
- the `len` function returns the number of items, and an iterator returns an unordered sequence of keys.

For a full list of operations see `https://docs.python.org/3.7/library/stdtypes.html#mapping-types-dict`.