# 600-152 Informatics 2 Lecture Notes. Recursion.

Bernie Pope. 10/5/09.

### **Introduction**

To iterate is human, to recurse divine. (L. Peter Deutsch)

Perhaps the single greatest function of electronic computers is their ability to execute a sequence of instructions quickly and repeatedly. A fancy word for repetition is *iteration*, and we employ iteration as a programming technique in Python every time we use a for/while-loop, such as:

```
def sum(list):
    result = 0
    for x in list:
       result += x
    return result
```

We say that the above function *iterates* over the values in the variable called list, and returns their sum.

Recursion is a kind of iteration, such that the operation being performed is defined (partly) in terms of itself. Such an operation is said to be *recursive*.

Here is a recursive definition of the sum() function:

```
def sum(list):
    if list == []:
        return 0
    else:
        return list[0] + sum(list[1:])
```

This computes the same thing as the previous definition, but it goes about it in a different way. The first thing to note is that it does not use a for-loop. The second thing to note is that the sum() function calls itself. That is to say, sum() is defined in terms of itself; it is recursive.

How does it work? The easiest way to see how it works is to pretend to be a Python interpreter, and follow the rules (without thinking too hard). Suppose we want to evaluate sum([1,2,3]).

```
sum([1,2,3])
= 1 + sum([2,3])
= 1 + (2 + sum([3]))
= 1 + (2 + (3 + sum([])))
= 1 + (2 + (3 + 0))
= 1 + (2 + 3)
= 1 + 5
= 6
```

In the above evaluation there are two interesting cases to consider:

- 1. When the argument list is empty. That is: sum([]), which evaluates to 0. This is called a *base case*. A base case is a stopping point for the iteration.
- 2. When the argument list is not empty. That is sum(list) (where len(list) > 0), which evaluates to list[0] + sum(list[1:]). This is called a *recursive case*. A recursive case is a repetition point in the iteration.

Recursion is computationally adequate. In rough terms: recursion can simulate for/while-loops (and *vice versa*). In some programming languages (such as <u>Haskell</u>), recursion is the only way that you can iterate (Haskell does not have for-loops).

We introduce recursion in this subject for two reasons:

- 1. It gives rise to the *divide and conquer* problem solving strategy.
- 2. It provides a more natural way to program with recursive data structures (such as Binary Search Trees), where for/while-loops are cumbersome.

# **Divide and conquer**

Way back in the very first lecture of this subject we introduced the merge sort algorithm:

- [Base case] If the list has length <= 1, then it is already sorted (nothing more to do).
- [Recursive case] If the list has length > 1:
  - Split it into two halves (each half is a new list).
  - Apply the merge sort algorithm to each half separately (sort them).
  - Merge the two sorted halves back into a single sorted list.

In the recursive case you can see that the algorithm is applied to each of the list halves; thus it is a recursive algorithm.

The merge sort algorithm, as described above, is a classic example of the *divide and conquer* problem solving strategy. We take a large problem (sorting a list) and we break it up into smaller instances of the same problem (sorting halves of the list). We solve the subproblems by further dividing and conquering (recursion). Finally, we combine the solutions to the sub-problems (merging sorted lists), yielding a solution to the original large problem.

Here is the same algorithm implemented in Python code:

```
def msort(list):
    if len(list) <= 1:
        return list
    else:
        top,bottom = split list(list)
        return merge(msort(top), msort(bottom))
def split list(list):
    half length = len(list) / 2
    return (list[:half length], list[half length:])
def merge(list1, list2):
    if list1 == []:
        return list2
    elif list2 == []:
       return list1
    else:
        e1 = list1[0]
        e2 = list2[0]
        if e1 <= e2:
            return [e1] + merge(list1[1:], list2)
        else:
            return [e2] + merge(list1, list2[1:])
```

The  $split_list()$  function splits a list into two halves (as close as possible). The merge() function takes two sorted lists as its arguments and returns the sorted combination of them as its result. Note that the merge() function is itself recursive.

The divide and conquer strategy is particularly amenable to parallel evaluation. That is, if we have more than one computing device available (such as multiple CPUs, or multiple CPUcores), then we can assign the evaluation of sub-problems to different devices concurrently. This can give significant performance improvements to the execution time of programs, and is an active area of research in Computer Science. Google have famously applied this technique for indexing web pages (and other computationally intensive tasks) in a system they call MapReduce.

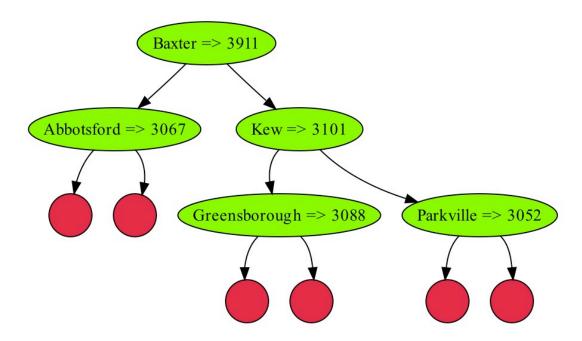
#### **Recursive data structures**

Some data structures are naturally defined recursively. A classic example is <u>Binary Search Trees (BSTs)</u>.

A BST is either an empty node or a branch node. An empty node contains nothing. A branch node contains a key and a value, and two children nodes, which are themselves BSTs.

Notice that the definition of a BST relies on the definition of a branch node, and a branch node relies on the definition of a BST; thus the definition is recursive.

Here is a diagram of an example BST:



In this week's workshop you are given the task of defining an XML format for representing BSTs. The recursive nature of BSTs becomes obvious if you try to write a DTD for such an XML format.

Here is a recursive DTD for such a format (workshop spoiler):

```
<!DOCTYPE bst
[
    <!ELEMENT bst (branch|empty)>
    <!ELEMENT branch ((branch|empty), (branch|empty))>
    <!ELEMENT empty EMPTY>
```

```
<!ATTLIST branch key CDATA #REQUIRED>
<!ATTLIST branch value CDATA #REQUIRED>
]>
```

Notice that the definition of the branch element is recursive.

Here is an XML document which is valid against that DTD, representing the BST in the diagram above:

```
<bst>
    <branch key="Baxter" value="3911">
        <branch key="Abbotsford" value="3067">
            <empty/>
            <empty/>
        </branch>
        <branch key="Kew" value="3101">
            <branch key="Greensborough" value="3088">
                <empty/>
                <empty/>
            </branch>
            <branch key="Parkville" value="3052">
                <empty/>
                <empty/>
            </branch>
        </branch>
    </branch>
</bst>
```

Can you verify that the XML document does accurately depict the BST?

## **Recursive functions over BSTs**

The recursive nature of BSTs makes them ideally suited to being processed recursively.

Recall from the lecture on BSTs that we represented them in Python as follows:

- Empty nodes are represented by empty lists.
- Branch nodes are represented by lists of length 4. The second and third elements of the list store the key and value of a node. The first and last elements store the left and right child of the node (which are themselves BSTs).

In this week's workshop you are provided with some example <u>Python code</u> which implements BSTs. In that code you will find the following function for performing an in-order traversal of a BST:

```
def stack_left_branch (bst, stack):
    while not is_empty (bst):
        stack.append(bst)
        bst = bst[left_child_index]

def in_order(bst):
    list = []
    stack = deque()
    stack_left_branch (bst, stack)
    while len (stack) > 0:
        node = stack.pop()
        list.append (key_value_of(node))
        stack_left_branch (node[right_child_index], stack)
    return list
```

An in-order traversal visits the nodes in the BST in key-sorted order. For instance, the function returns the following list when applied to the example BST from the diagram above:

```
[('Abbotsford', 3067), ('Baxter', 3911), ('Greensborough', 3088), ('Kew',
3101), ('Parkville', 3052)]
```

The result is a list of key-value pairs, such that the pairs are sorted in (lexicographic) order of the keys.

Unfortunately the above definition of  $in\_order()$  is awkward because it is written with a while-loop, which does not reflect the inherent recursive nature of BSTs. The while-loop necessitates the use of an stack, which makes the code hard to understand. A much more natural way to write this function is to use recursion:

```
def in_order(bst):
    if is_empty(bst):
        return []
    else:
        lefts = in_order(left_child_of(bst))
        rights = in_order(right_child_of(bst))
        return lefts + [key value of(bst)] + rights
```

As you can see the code is much more concise. The recursive structure of the code reflects the recursive definition of BSTs.

Below are more examples of recursive functions over BSTs. You should note that the function definitions all follow a similar pattern, corresponding to the natural base case and recursive cases that arise from the structure of BSTs.

This function counts the number of branch nodes in a BST:

```
def size(node):
    if is_empty(node):
        return 0
    else:
        left_size = size(left_child_of(node))
        right_size = size(right_child_of(node))
    return 1 + left size + right size
```

This function calculates the maximum number of steps needed to get from the root node to a leaf node in a BST (this is the depth of the tree):

```
def depth(node):
    if is_empty(node):
        return 0
    else:
        left_depth = depth(left_child_of(node))
        right_depth = depth(right_child_of(node))
        return 1 + max(left_depth, right_depth)
```

This function searches for a key in a BST and returns its value or None if it was not found:

```
def search_rec(node, target_key):
    if is_empty(node):
        return None
    elif key_of(node) == target_key:
        return value_of(bst)
    elif target_key <= key_of(node):
        return search_rec(left_child_of(node), target_key)
    else:
        return search_rec(right_child_of(node), target_key)</pre>
```

This function tests of a tree is a valid BST. That means for every branch node: the key of the node is greater than every key in its left child, and less than every key in its right child. We assume that the keys are unique.

```
def is_bst(node):
    if is_empty(node):
        return True
    else:
        this_key = key_of(node)
        left_child = left_child_of(node)
        # check if the key is > the the root of the left child
        if key_greater_than_node(this_key, left_child):
            right_child = right_child_of(node)
            # check if the key is < the root of the right child
            if key_less_than_node(this_key, right_child):
                 # check if the left and right children are also BSTs</pre>
```

```
return is_bst(left_child) and is_bst(right_child)
            else:
               return False
        else:
            return False
# check if a key is > the the root of a node
def key greater than node(key, node):
    if is empty(node):
        return True
    else:
        return (key > key of(node))
# check if a key is < the the root of a node
def key less than node(key, node):
    if is empty(node):
        return True
    else:
        return (key < key_of(node))</pre>
```

#### This function collects a set of all the keys in a BST:

```
def bst_keys(node):
    if is_empty(node):
        return set()
    else:
        left_keys = bst_keys(left_child_of(node))
        right_keys = bst_keys(right_child_of(node))
        all_keys = left_keys.union(right_keys)
        all_keys.add(key_of(node))
        return all_keys
```