Introduction to programming

Lecture 12:

Lecturers: Giovanni Casini (giovanni.casini@uni.lu) and Xavier Parent (xavier.parent@uni.lu)

Revised version of material from Clément Guérin

Trees

In computer science, a tree is a **data structure** that is modeled after nature. The tree data structure is upside down: the root of the tree is on top. A tree consists of **nodes** and its connections are called **edges**. The bottom nodes are also named **leaf nodes**. A tree does not have cycles.

A node can contain a value or any kind of data.

Any tree is formalised using two attributes: a **node** and a **list of subtrees** which is a list of trees.

If a tree has an empty list of subtrees then it is a **leaf**.

The very definition of a tree is **recursive** and this explains why almost every definition or function dealing with trees is recursive.

As a data structure, the **content** of a tree is the union of the contents of all its subtrees with its node.

The **depth** of a tree is 1 if the tree is a leaf and is $1+$ the maximum of the depth of its subtrees else.

The **width** of a tree is 1 if the tree is a leaf and is the sum of the widths of its subtrees else.

In [1]:

```
from turtle import *  # I want to draw

class Tree():
    'A recursive class, two attributes : nod and sub'

    def __init__(self,node=None,listtrees=None):
        self.node=node
        if listtrees==None: #Little trick to avoid the use of a mutable keyword
 'constant' argument
            self.sub=[]
        else:
            self.sub=listtrees

    #Answers if the tree is a leaf
    def isleaf(self):
        if len(self.sub)==0:
            return True
        else:
            return False

    #Computes the number of nodes in a tree.
    def __len__(self):
        if self.isleaf():
            return 1
        else:
            s=1
            for x in self.sub:
                s+=__len__(x)
            return s


    #Computes the depth of a tree
    def depth(self):
        if self.isleaf():
            return 1
        else:
            he=0
            for x in self.sub:
                he2=x.depth()
                if he<he2:
                    he=he2
                else: pass
            return he+1

    #Computes the width of a tree (number of leaves)
    def width(self):
        if self.isleaf():
            return 1
        else:
            s=0
            for x in self.sub:
                s+=x.width()
            return s



    # This method answers wether x is a node of  self or not.
    def isanode(self,x):
        if x==self.node():
```

```
            return True
        elif self.isleaf():
            return False
        else:
            for t in self.sub:
                if isanode(self.t):
                    return True
            return False


    def draw(self,xmin,xmax,ymin,ymax,size=10,drawnode=True): #The drawing of se
lf will be centered between
                                                #xmin and xmax (in x coordinates)
                                                #ymin and ymax (in y coordinates)
                                                #size is the size of the characters
                                                #drawnode is a boolean saying if th
e nodes have to be drawn or not
        if drawnode:
            if self.isleaf():
                penup()
                #Write the node of the tree
                setposition((xmax+xmin)/2.,ymax)
                if self.node!=None:
                    write("{}".format(self.node), True, align="center",font=( "A
rial", size, "normal"))


            else:
                    penup()
                        #Write the node of the tree
                    setposition((xmax+xmin)/2.,ymax)
                    if self.node!=None:
                        write("{}".format(self.node), True, align="center",font=
( "Arial", size, "normal"))

                #Define the depth
                    h=self.depth()

                # Define the new  ymax
                    nymax=ymax-(ymax-ymin)/(h-1)
                    L=list(map(lambda x: x.width(),self.sub))
                    s=0
                    for x in L:
                        s+=x

                #the succesive umin and umax are the xmin and xmax of all subtre
es of self
                    umin=xmin
                    for t in self.sub:
                        umax=umin+(xmax-xmin)*t.width()*1.0/s
                        setposition((xmax+xmin)/2.,ymax)
                        pendown()
                        setposition((umax+umin)/2.,nymax+size)
                        penup()

                    #We have to recompute ymin
                        t.draw(umin,umax,ymin+(1-t.depth()/(h-1))*(ymax-ymin),ny
max,size)

                        umin=umax
```

```
        else:
            if self.isleaf():
                penup()
                #Write the node of the tree
                setposition((xmax+xmin)/2.,ymax)
            else:
                penup()
                #Write the node of the tree
                setposition((xmax+xmin)/2.,ymax)


                #Define the depth
                h=self.depth()

                # Define the new  ymax
                nymax=ymax-(ymax-ymin)/(h-1)

                L=list(map(lambda x: x.width(),self.sub))
                s=0
                for x in L:
                    s+=x

                #the succesive umin and umax are the xmin and xmax of all subtre
es of self
                umin=xmin
                for t in self.sub:
                    umax=umin+(xmax-xmin)*t.width()*1.0/s
                    setposition((xmax+xmin)/2.,ymax)
                    pendown()
                    setposition((umax+umin)/2.,nymax)
                    penup()

                    #We have to recompute ymin
                    t.draw(umin,umax,ymin+(1-t.depth()/(h-1))*(ymax-ymin),nymax,
size,False)

                    umin=umax
```

In [4]:

```
#Here a few trees are defined


tree1=Tree(1)
tree2=Tree(2,[tree1])
tree3=Tree(3,[tree1,tree2,tree1,tree2])
tree7=Tree(7,[tree3])
tree8=Tree(8,[tree7])
tree9=Tree(listtrees=[tree8])
tree10=Tree(10,[tree9])
tree4=Tree(4,[tree3,tree3,tree10,tree3])
tree5=Tree(5,[tree3,tree10,tree2,tree10])
tree6=Tree(6,[tree4,tree3,tree4,tree5,tree1])


reset()
title("tree2 and tree4")
speed(0)
hideturtle()
tree2.draw(-200,-100,-200,250,10)
tree5.draw(-50,200,-100,250,10)

print("depth of tree5",tree5.depth())
print("width of tree5",tree5.width())
```

```
depth of tree5 8
width of tree5 13
```

In [7]:

```
#Now we draw a big tree


tracer(1)
reset()
title("Example of a tree with and without nodes")
speed(0)
hideturtle()


tree6.draw(-300,300,0,300,10,False)
tree6.draw(-300,300,-300,-20,10,True)

penup()
setpos(-200,-51)
setpos(300,-51)
penup()
tracer(1)
```

Reverse Polish Notation (RPN)

Principle

The **Reverse Polish Notation** (RPN) also known as **Polish postfix notation** is a way to write a computation without needing any parenthesis to describe the priority. As we shall see, it can be described using a **tree**. By the way, the traditional way to write an arithmetic expression is called the **infix notation**.

We will call **operator** the function which accepts n arguments. n is the **arity** of the operator.

The n arguments of the operators are the **operands**.

The process is called **operation** which will give a **result**.

For instance, in $2 + 3$,

- $+$ is an **operator** of **arity** 2,
- 2 and 3 are **operands**,
- The **result** of the **operation** is 5.

The qualities of the RPN :

- the syntax is more compact than the infix notation one because you don't need parenthesis
- it is closer to the machine language

The drawbacks of the RPN :

- learning arithmetic as a child is harder with the RPN rather than with infix notation.
- An operator has a fixed arity in RPN whereas there are exeptions in infix notation. FOr example, $-$ can both mean do the difference of two numbers or take the opposite of one number.

For simplicity, in this lecture, in the RPN notation.

- Operands will only be integers non-negative floating numbers or integers
- Operands will be separated by one single space
- The operators of arity 2 we shall use are $+$, $\times$ (noted as the letter x) , $-$, / (the quotient) and 'p' (the power)
- $3\ 2-$ means $3 - 2$ and not $2 - 3$.
- $3\ 4/$ means $3/4$ and not $4/3$.
- $3\ 2p$ means 3^2 and not 2^3.
- The operators of arity 1 we shall use will be $\sin$ and $\cos$ denoted as s and c

Examples and algorithm of evaluations

The RPN conversion of '$2 + 3$' is '$2\ 3+$'.

The RPN conversion of '$2 + 3 \times 2$' is '$2\ 3\ 2\times+$'.

The RPN conversion of '$(2 + 3) \times 2$' is '$2\ 3 + 2\times$'.

For any real numbers a, b, c $abc \times \times = acb \times \times$ is the definition of the associativity in RPN notation.

The RPN conversion of $(5 - 3)^8$ is '$5\ 3 - 8p$'

The algorithm to convert an RPN formula to goes as follow. You begin with an empty list (the **stack**) which will contain all operands and a reading index (**rind**) at 0.

If you read an integer or a point, it means that you are supposed to read an operand and you keep on reading the number (meaning increase **rind** by 1) until you reach a space (meaning that you are changing of operands) or an operator. Then you append the number to the **stack**.

If you read an operator, then you apply the operator (of given arity k) to the last k-elements respecting the order and append the result to the list.

You stop when there is not enough operands in the stack to apply an operator (in which the formula is badly written) or when you are at the end of the formula.

If you have two elements in the stack or more there is a problem, else return the single element of the list.

The algorithm to convert an RPN formula to goes as follow. You begin with an empty list (the **stack**) and a reading index (**rind**) at 0.

First you want to find an operand. You keep on reading (meaning increase **rind** by 1) until you reach a space (meaning that you are changing of operands) or an operator. Then you append the number to the **stack**.

If you read an operator, then you apply the operator (of given arity k) to the last k-elements respecting the order and append the result to the list.

You stop when there is not enough operands in the stack to apply an operator (in which the formula is badly written) or when you are at the end of the formula.

If you have two elements in the stack or more there is a problem, else return the single element of the list.

In [2]:

```
#I use cos and sin
import math

#list of possible operators
oparity1=['c','s']
oparity2=['+','x','-','/','p']


#Algorithm to compute a formula written in RPN
def RPNcomp(formula):

    #Initialization
    stack=[]
    rind=0


    while rind<len(formula):

        #index to read the operand
        rind2=rind
        while not(formula[rind2] in oparity1 or  formula[rind2] in oparity2 or f
ormula[rind2]==' '):
                rind2+=1

        #read the whole real number, error if it is badly written


        #By adding this if loop I add the possibility to use as many blank space
s as I want in my formula
        if rind2!=rind:
            num=int(formula[rind:rind2])
            stack.append(int(num))


        #Add the number to your LIFO stack
        #Begin the reading
        rind=rind2

        #If you stop reading th number because of an operator
        if formula[rind]=='c':
            x=stack.pop()
            stack.append(math.cos(x))
        elif formula[rind]=='s':
            x=stack.pop()
            stack.append(math.sin(x))
        elif formula[rind]=='+':
            x=stack.pop()
            y=stack.pop()
            stack.append(y+x)
        elif formula[rind]=='x':
            x=stack.pop()
            y=stack.pop()
            stack.append(y*x)
        elif formula[rind]=='-':
            x=stack.pop()
            y=stack.pop()
            stack.append(y-x)
        elif formula[rind]=='/':
            x=stack.pop()
```

```
            y=stack.pop()
            stack.append(y/x)
        elif formula[rind]=='p':
            x=stack.pop()
            y=stack.pop()
            stack.append(y**x)
        #Else you either have a space
        elif formula[rind]==' ':pass
        #Or some character I don't know
        else:
            raise ValueError('{} is not an admissible character'.format(formula[
rind]))
        rind+=1


    #After the while loop the stack is complete

    if len(stack)>1:
        raise ValueError('the formula is not valid because the stack is too big'
)
    else:
        return stack[0]
```

In [9]:

```
RPNcomp('5 3-8  p  ')
```

Out[9]:

```
256
```

RPN to trees/trees to RPN

The RPN notation admits a nice representation as a tree of operators.

If we consider an operand as an operator of arity 0 (a constant) then we can represent an operation as a tree with:

- All the nodes that are not leaves represents an operator that has as many subtrees as its arity;
- All the leaves represents operands (numbers)

In [10]:

```
#For example, what is the RPN associated to this tree?
RPNtree2=Tree('2')
RPNtree3=Tree('3')
RPNtree6=Tree('6')
RPNtreex=Tree('x',[RPNtree2,RPNtree3])
RPNtreepl=Tree('+',[RPNtreex,RPNtree6])
RPNtree=Tree('x',[RPNtreepl,RPNtreepl])
reset()
hideturtle()
speed(0)
RPNtree.draw(-200,200,-200,200,40)
```

Trees to RPN

In this case, to convert a tree, it suffices to convert a leaf to its node, recursively concatenate the RPN notation of the subtrees and write the node operator at the end. You have to take care of spaces, but as long as you add a space right after a leaf node you are fine.

In [3]:

```
def treetoRPN(tree):
    if tree.isleaf():
        return('{} '.format(str(tree.node)))
    else:
        res=''
        for t in tree.sub:
            res+='{}'.format(treetoRPN(t))
        res+=tree.node
        return res
```

In [12]:

```
print(treetoRPN(RPNtree))
RPNcomp(treetoRPN(RPNtree))
```

```
2 3 x6 +2 3 x6 +x
```

Out[12]:

```
144
```

In [ ]:

```
# Exercise :
#1 Write an evaluation function that 'evaluates' an RPN-tree
#2 Write a script that shows the evolution of the RPN-tree (replace a tree by it
s value when it is actually computed)
```

RPN to trees

The algorithm to convert an RPN formula to a tree is looking like the algorithm evaluating a RPN formula.

We put **leafs** instead of floating numbers in the stack and instead of evaluating when we come across an operator we just **build** the tree with the corresponding arity operator.

In [13]:

```
def RPNtotree(formula):

    #Initialization
    stack=[]
    rind=0


    while rind<len(formula):

        #index to read the operand
        rind2=rind
        while not(formula[rind2] in oparity1 or  formula[rind2] in oparity2 or f
ormula[rind2]==' '):
                  rind2+=1

        #read the whole real number, error if it is badly written


        #By adding this if loop I add the possibility to use as many blank space
s as I want in my formula
        if rind2!=rind:
            num=Tree(int(formula[rind:rind2]))
            stack.append(num)


        #Add the number to your LIFO stack
        #Begin the reading
        rind=rind2

        #If you stop reading th number because of an operator
        if formula[rind] in oparity1:
            x=stack.pop()
            stack.append(Tree(formula[rind],[x]))
        elif formula[rind] in oparity2:
            x=stack.pop()
            y=stack.pop()
            stack.append(Tree(formula[rind],[y,x]))
        #Else you either have a space
        elif formula[rind]==' ':pass
        #Or some character I don't know
        else:
            raise ValueError('{} is not an admissible character'.format(formula[
rind]))
        rind+=1


    #After the while loop the stack is complete

    if len(stack)>1:
        raise ValueError('the formula is not valid because the stack is too big'
)
    else:
        return stack[0]
```

In [14]:

```
t=RPNtotree(treetoRPN(RPNtree))
clear()
RPNtree.draw(-300,-10,-100,100,30)
t.draw(10,300,-100,100,30)
```

Binary trees

A **binary tree** is a tree where the list of subtrees is of length at most 2, and all subtrees are themselves binary. It is clearly a subclass of the class of trees.

For implementation reasons it is generally a better thing to consider that a tree has either 0 or 2 subtrees, and if it has two then one of them may be void (a binary tree is void if it is a leaf with a 'None' node).

In a binary tree, every node which is not a leaf has a left branch, a right branch or both.

In most cases, one imposes another relation between the node n of a tree, the node of the left subtree n_L and the node of the right subtree n_R : $n_L \leq n \leq n_R$.

The reason for this condition is that if you are **looking for an element** x in a tree t and $x \neq n$, where n is the node of t, then if $x < n$ you just have to look for x in the left subtree of t and if $x > n$ you just have to look for x in the right subtree of t.

In [15]:

```
class BinTree(Tree):

    def __init__(self,node=None,listtrees=None):
        self.node=node
        if listtrees==None: #Little trick to avoid the use of a mutable keyword
 'constant' argument
            self.sub=[]
        elif len(listtrees)>2:
            raise ValueError("Not a binary tree")
        else:
            self.sub=listtrees


    def __len__(self):
        if self.node!=None:
            s=1
        else:
            s=0
        if self.isleaf():
            return s
        else:
            for x in self.sub:
                s+=len(x)
            return s

    def width(self):
        if self.isleaf():
            return 1
        else:
            s=0
            for x in self.sub:
                s+=x.width()
            return s

    def isin(self,x):
        if self.node==x:
            return True
        else:
            if self.isleaf():
                return False
            elif x<self.node():
                return (self.sub[0]).isin(x)
            else:
                return (self.sub[1]).isin(x)

    def drawb(self,xmin,xmax,ymin,ymax,size=10,drawnode=True): #The drawing of s
elf will be centered between
                                            #xmin and xmax (in x coordinates)
                                            #ymin and ymax (in y coordinates)
                                            #size is the size of the characters
                                            #drawnode is a boolean saying if th
e nodes have to be drawn or not
        if drawnode:

            if self.isleaf():
                penup()
                #Write the node of the tree
                setposition((xmax+xmin)/2.,ymax)
                if self.node!=None:
```

```
                    write("{}".format(self.node), True, align="center",font=( "A
rial", size, "normal"))


            else:
                    penup()
                        #Write the node of the tree
                    setposition((xmax+xmin)/2.,ymax)
                    if self.node!=None:
                        write("{}".format(self.node), True, align="center",font=
( "Arial", size, "normal"))

                #Define the depth
                    h=self.depth()

                # Define the new  ymax
                    nymax=ymax-(ymax-ymin)/(h-1)
                    L=list(map(lambda x: x.width(),self.sub))
                    s=0
                    for x in L:
                        s+=x

                #the succesive umin and umax are the xmin and xmax of all subtre
es of self
                    umin=xmin
                    for t in self.sub:

                        umax=umin+(xmax-xmin)*t.width()*1.0/s
                        if t.node!=None:
                            setposition((xmax+xmin)/2.,ymax)
                            pendown()
                            setposition((umax+umin)/2.,nymax+size)
                            penup()

                    #We have to recompute ymin
                        t.drawb(umin,umax,ymin+(1-t.depth()/(h-1))*(ymax-ymin),n
ymax,size)

                        umin=umax

        else:
            if self.isleaf():
                penup()
                #Write the node of the tree
                setposition((xmax+xmin)/2.,ymax)



            else:
                    penup()
                        #Write the node of the tree
                    setposition((xmax+xmin)/2.,ymax)


                #Define the depth
                    h=self.depth()

                # Define the new  ymax
                    nymax=ymax-(ymax-ymin)/(h-1)
                    L=list(map(lambda x: x.width(),self.sub))
                    s=0
```

```
                    for x in L:
                        s+=x

                #the succesive umin and umax are the xmin and xmax of all subtre
es of self
                    umin=xmin
                    for t in self.sub:

                        umax=umin+(xmax-xmin)*t.width()*1.0/s
                        if t.node!=None:
                            setposition((xmax+xmin)/2.,ymax)
                            pendown()
                            setposition((umax+umin)/2.,nymax)
                            penup()

                        #We have to recompute ymin
                        t.drawb(umin,umax,ymin+(1-t.depth()/(h-1))*(ymax-ymin),n
ymax,size,False)

                        umin=umax
```

Sorting a list with binary trees

A binary tree is a very efficient data structure to sort lists.

Let us say you want to sort the list L. The algorithm is quite simple, you take the first element x from the list, it is the node of your tree t. Then you compare every other elements of the list L to x and construct two lists Lm and Lp of elements of L which are respectively smaller than x and greater than x. Then the left subtree of t is the tree associated to Lm and the right subtree is the tree associated to Lp.

In [16]:

```
def listtobinTree(L):
    if len(L)==0:
        return BinTree(None)
    if len(L)==1:
        return BinTree(L[0])
    else:


        #Construction of Lm and Lp
        Lm,Lp=[],[]
        for x in L[1:len(L)]:
            if x<L[0]:
                Lm.append(x)
            else:
                Lp.append(x)

        return BinTree(L[0],[listtobinTree(Lm),listtobinTree(Lp)])
```

In [3]:

```
#First example
L=[1,3,2,7,4,10,5,-2,6,-5]
tracer(1)
reset()
speed(3)
hideturtle()
T=listtobinTree(L)
type(T)
T.drawb(-300,300,-300,300,20)
```

```
---------------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-3-c1b92733c106> in <module>
      5 speed(3)
      6 hideturtle()
----> 7 T=listtobinTree(L)
      8 type(T)
      9 T.drawb(-300,300,-300,300,20)

NameError: name 'listtobinTree' is not defined
```

In [18]:

```
sentence='Sed condimentum elit a dui auctor, sed accumsan ipsum laoreet. Maecena
s nec malesuada dolor. Fusce dictum tortor vitae interdum malesuada. Vestibulum
 id diam quis leo porta dictum. Aliquam sit amet gravida nibh. Duis at tellus id
ligula efficitur accumsan eu a erat. Pellentesque consectetur odio sed ligula la
oreet sagittis. Phasellus maximus tristique leo, vitae laoreet nunc congue eu. A
liquam erat volutpat. Orci varius natoque penatibus et magnis dis parturient mon
tes, nascetur ridiculus mus. Fusce at pulvinar felis. Sed mattis ligula vitae ri
sus placerat auctor. Class aptent taciti sociosqu ad litora torquent per conubia
nostra, per inceptos himenaeos. Vestibulum iaculis nunc at neque ullamcorper mol
estie. Cras quis egestas metus, id maximus velit. Nulla ac congue eros \
Proin varius sodales mauris, vitae vestibulum arcu tincidunt eu. In hac habitass
e platea dictumst. Maecenas urna eros, mollis eget elit non, facilisis euismod n
ibh. Pellentesque sed pellentesque turpis, eget facilisis diam. Mauris vel tempu
s augue. Praesent lacinia elit sed lobortis porttitor. Suspendisse potenti. Maur
is ultrices rhoncus iaculis. Vestibulum elementum mauris lorem, quis consequat j
usto iaculis nec. Praesent posuere, purus sed posuere facilisis, nisl justo laor
eet nisi, feugiat interdum ex odio pretium dui.\
Quisque sit amet dignissim dui, vitae sodales est. Curabitur quis ornare mi, a r
utrum metus. Cras ultrices suscipit mi vel euismod. Nunc suscipit imperdiet ligu
la, at posuere dui ornare quis. Quisque eleifend augue sed accumsan consectetur.
Vivamus maximus dignissim ipsum vel rutrum. Aenean elit nulla, volutpat eget leo
sit amet, semper lobortis est. Vivamus auctor in mi non molestie. Etiam risus er
os, vestibulum ac fringilla eget, rhoncus gravida turpis. Sed eu viverra quam.\
Vestibulum varius nisl purus, quis pretium orci dictum nec. Donec mollis, augue
 quis venenatis molestie, purus augue aliquet purus, vitae sodales nulla est ac
 neque. Pellentesque posuere dui in lorem tempor hendrerit. Curabitur a porttito
r metus, sit amet malesuada orci. Nunc ac magna sit amet sapien posuere viverra.
Pellentesque hendrerit tristique lacus quis maximus. Sed id pulvinar sapien. Cra
s vel pretium felis. Nullam lectus nisl, sagittis et cursus eget, consectetur ne
c mi. Aenean fermentum iaculis sapien, eu vulputate purus pretium eu. Praesent n
on cursus lacus, non dictum eros.\
Quisque at augue aliquet, pellentesque dolor nec, elementum nibh. Aenean imperdi
et arcu vitae gravida elementum. Phasellus vulputate, mauris a semper scelerisqu
e, tortor sem vehicula justo, a faucibus tellus ex vitae magna. Sed pharetra sem
at luctus pellentesque. Nulla eu erat sed metus egestas interdum. Sed volutpat v
elit condimentum, dignissim quam id, aliquet nisi. Sed enim nibh, sodales et era
t vel, posuere lacinia ex. Ut bibendum orci commodo eros consectetur, eu consect
etur nulla dapibus. Nulla faucibus quis mauris vitae consequat. Cras tincidunt a
nte at purus lobortis imperdiet. Quisque hendrerit suscipit felis, vel congue di
am cursus nec. Sed et turpis vehicula, convallis neque ut, porta diam. Vestibulu
m porta sed tellus quis congue. Interdum et malesuada fames ac ante ipsum primis
in faucibus. Mauris in condimentum urna.\
Curabitur velit lacus, accumsan sit amet orci aliquam, rhoncus eleifend orci. Nu
llam aliquet sagittis quam, sit amet volutpat justo dignissim vel. Nullam sed ti
ncidunt nibh, et tincidunt ante. Vestibulum facilisis venenatis ante a porta. Vi
vamus posuere mi ut nibh pulvinar, scelerisque sodales orci faucibus. Nullam con
sectetur nunc vitae pretium sodales. Duis tincidunt justo eget nunc scelerisque
 bibendum. Vestibulum sagittis, diam vitae ullamcorper finibus, justo dolor grav
ida massa, id pretium odio libero accumsan nisl. Donec placerat aliquam odio in
 facilisis. Nam lacinia odio eros, a fringilla velit placerat sit amet. Nam rutr
um eget nisi quis venenatis. Mauris a libero nunc. Nunc sed posuere nunc. Fusce
 eget massa et arcu efficitur molestie quis eu nisl. Curabitur id elit erat. Don
ec lacinia ullamcorper dui, ac varius urna luctus non.'
```

In [19]:

```
#The algorithm also works for list of words
T=listtobinTree(sentence.split())
```

In [20]:

```
reset()
speed(0)
hideturtle()

tracer(10)
T.drawb(-300,300,-300,300,20,False)
tracer(0)
```

The resulting tree changes drastically when you randomize the choice of the pivot x.

In [21]:

```
import random
```

In [22]:

```
def listtobinTreerandom(L):
    if len(L)==0:
        return BinTree(None)
    if len(L)==1:
        return BinTree(L[0])
    else:
        Lm,Lp=[],[]

        #Here is the maine change
        indexpivot=random.randrange(0,len(L))
        for index in range(0,len(L)):
            if index!=indexpivot:
                x=L[index]
                if x<L[indexpivot]:
                    Lm.append(x)
                else:
                    Lp.append(x)
        if len(Lm)==0:
            return BinTree(L[0],[listtobinTree(Lp)])
        elif len(Lp)==0:
            return BinTree(L[0],[listtobinTree(Lm)])
        else:
            return BinTree(L[0],[listtobinTree(Lm),listtobinTree(Lp)])
```

In [23]:

```
T=listtobinTreerandom(sentence.split())
```

In [24]:

```
reset()
speed(0)
hideturtle()
screensize(1200,800)
tracer(10000)
T.draw(-500,500,-300,300,3,False)
tracer(1)
```

In [25]:

```
N=1000
S=0
# Here we make a little comparison for a list of numbers between 0 and N-1
L=list(range(0,N))
random.shuffle(L)



for i in range(0,25):
    T=listtobinTreerandom(L)
    S+=T.depth()
    reset()
    hideturtle()
    screensize(1500,1000)
    tracer(10000)
    T.draw(-500,500,-300,300,3,False)
    tracer(1)
    speed(1)
    left(100)
    right(100)
    speed(0)
```

In [ ]:

```
reset()
hideturtle()

tracer(100)
T=listtobinTree(L).draw(-300,300,-300,300,3,False)
tracer(1)
```