Python - data structures and algorithms

Table of Contents

Basic ideas

These are notes about data structures and algorithms, using Python. A basic knowledge of Python, and how to write and run Python code, is needed.

Foundational computer science includes a set of classical data structures and algorithms. These notes cover:

- Some classical data structures
- The data structure classes built in to Python
- Defining classes to model other data structures
- Some classical algorithm applications

Variables values and types

For example

```
x=27
print(x, type(x))
x="ABCD"
print(x, type(x))
```

Then x is a variable. A variable has

- a name
- a value (which can change)
- a type (which can change)

In this example, the *name* of the variable is x. The *value* is at first 27, then it is "ABCD". The *type* is class int, then class str.

Because this is Python, all values are objects. That means that they contain pieces of data, and also *methods*, so we can tell them to do things. For example

```
x="ABCD"
print(x.lower())   # abcd
```

so the x object has a method named 'lower', producing a new string with characters switched to lower case.

What is a data structure?

We usually need to handle a *set of data items together,* not just one at a time. For examples

- all the pixels in an image
- all the files in a folder
- all the transactions in a bank account over the last month
- some of the paragraphs in a web page
- product orders on an ecommerce site since last Tuesday

A set of data items together needs to be arranged in some pattern or design or structure or arrangement. We usually think about the structure as a picture. Some possibilities are

- a list, with items in a row or a line
- a table
- a circle

and several others.

The data items in a structure are usually called *elements*, stored in *nodes*.

What is an algorithm?

An algorithm is a method, a problem solution, a recipe, a way of doing something. An algorithm is a sequence of simple steps which a computer can do, without intelligence. We write an algorithm as pseudo-code, to make it as clear and simple as possible, or in some programming language, so a computer can carry out the algorithm at speed.

With a data structure, we often want to:

- add a new node
- find a node
- delete a node
- count the nodes
- add them up
- find the largest, smallest, average

We need algorithms to do all of these. The algorithm usually depends on which structure we are using.

A *good algorithm* is fast and uses as little memory as possible.

Objects and types

In Python, all values are objects. That means they have one or more data parts and pieces of code which do things, known as methods.

A *class* is a type of object. Python has some built-in classes, and we can code our own.

What is an abstract data type?

An abstract data type (ADT) is a data structure treated as *what it does* and *not how it works*.

As an example, a stack ADT is a *last-in, first-out* structure. There are only two actions - to push a value onto a stack, and pop a value off. Th evalue we pop off is always the last value we pushed.

So if we start with an empty stack, and push values 6,2,9 onto it, in that order, then we

pop a value - we get 9

pop a value - get 2

pop a value - get 6

Now the stack is empty and there is nothing more on it to pop.

In actual program code, we must somehow *implement an ADT*. That is, write new code or use built-in language features to make something that works that way.

Usually an ADT can be implemented in different ways. A stack, for example, can be implemented using an array or a linked list - these are explained later.

We will look at the standard ADTs, usual implementations, and their algorithms.

Traversals

We often want to process somehow all the elements of a data structure. For example we might want to find a value, which might mean 'visiting' every node.

To traverse a structure means to visit some or all of the nodes. Different traversal algorithms are used for different ADTs and implementations.

Some common data structures

The study of data structures is an important part of computer science. This section outlines the standard structures. This is language-agnostic - which means it applies to all programming languages, and does not include the details of these structures in Python, or anything else.

The idea is to sketch out some basic ideas, before looking at some Python details.

This just describes these as ADTs - abstract data types, saying *what they do*, not *how they work.*

Arrays

An array is an ADT in which each element has an *index* (a unique integer), and there are two operations

- *Put a data item* at an index position. So for example

```
arr[5]=77
```

puts the data value 77 into the array named arr, at index 5

- *Get a data item* at an index position. For example

```
print ( arr[5] )
```

gets the data at index 5 from the array arr, and prints it.

There might be other operations - such as getting the length of the array, which is the number of data items in it.

In many languages (but not all) the following are true:

- The size of an array is fixed at compile-time. Items cannot be added or removed, just changed.
- The index starts at 0. So if an array contains 10 elements the index ranges from 0 to 9 inclusive.
- All elements have the same data type, and so occupy the same number of bytes
- An array is stored in one *contiguous* memory block. That means one single area of memory, not a set of blocks in different places, somehow linked.
- Array element access is fast.

Stacks

A stack is a *last-in first-out* structure. It has two basic operations:

- Push another item onto the stack
- Pop a value off the stack (and get the most recently pushed value)

There is sometimes an additional peek operation, to get a copy of the stack top item without removing it.

Stacks naturally reverse sequences. For example if we push 1,2,3,4, then pop them, we get 4,3,2,1

Stack underflow is when we try to pop an empty stack.

Stack overflow happens when we try to push more values than will fit into memory assigned for it (which depends on the implementation)

Queues

A queue is a *first-in first-out* structure. It has two basic operations:

- Enqueue another item into the queue
- Dequeue a value from the queue (and get off the value which has been there the longest)

Queues preserve sequences. If we enqueue 1,2,3,4 then dequeue them, we get 1,2,3,4

Maps

A map is an ADT containing a set of data pairs. Each pair has a key, and is paired with some value. The keys are unique - no two pairs can have the same key. There are two basic operations:

- *Put a key-value pair* into the map. Usually if the key is already in the map, the value replaces the current value. For example

myMap.put(254, "abcd")

would add a pair to myMap, with key 254, and value "abcd"

- *Get a value* which is paired with a given key. So

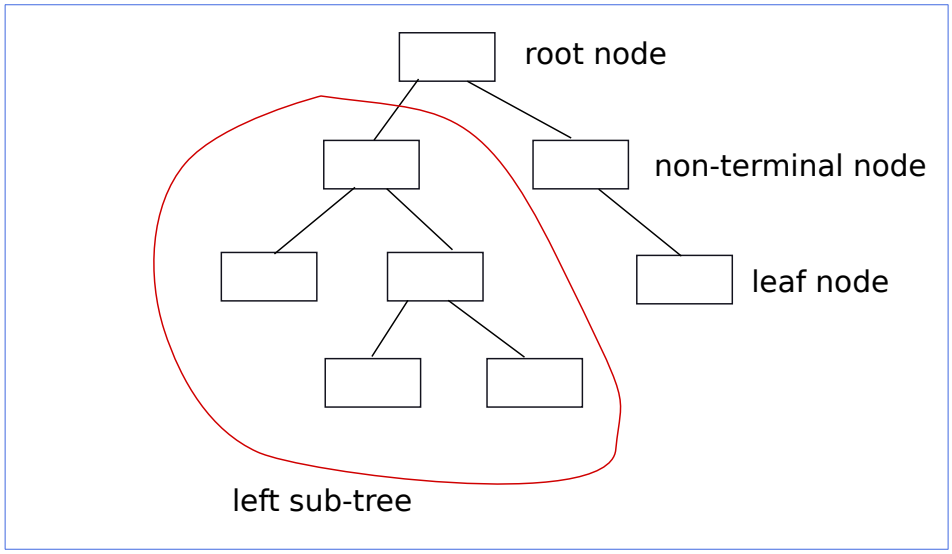
print(myMap.get(254))

would get the value "abcd" paired with key 254 from myMap, and print it.

Maps are sometimes called dictionaries or associative arrays. JavaScript objects, for example, are associative arrays.

Trees

A tree is an ADT as shown.



It contains a set of nodes, each of which are linked to zero or more child nodes. There are no loops - no circular inks.

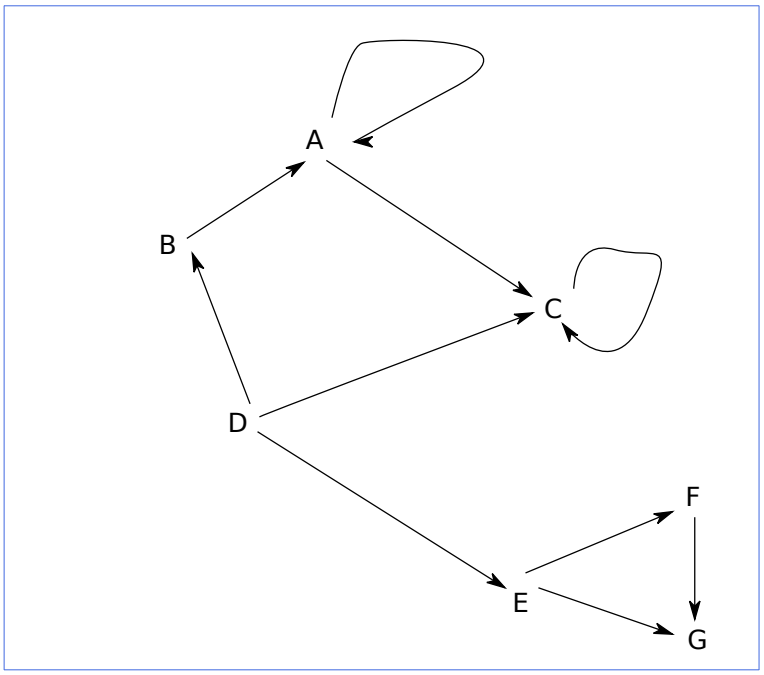
The node at the top is called the root node. A node with no child nodes is called leaf node or terminal node.

A binary tree has 0,1 or 2 children for each node. A non-binary tree is not limited like this. The tree corresponding to a folders and files, for example, might have many files in one folder so would not be binary.

Trees are often treated recursively. For example, a tree is a root node, and left and right sub-trees, which are themselves trees.

The defining aspect is *no loops*. A tree is an *acyclic graph*.

Graphs



A graph is an ADT rather like a road map.

It has a set of nodes (like towns) called *vertexes*, and links between them (like roads) which are called *edges*.

Edges may be one-way, in which case we have a *directed graph*. Or not.

Edges may have a *weighting*, such as the distance of the road, or how heavy the traffic is. Or it might be unweighted.

There might be a route from any node to any other node - in which case it is a *connected* graph. Or not - say for towns on an island separated from the mainland.

The graph might have loop journeys back to a starting point - when it is a *cyclic graph*. Otherwise it is acyclic.

A tree is one kind of graph - an acyclic one.

Built-in Python structures

In Python we have

- lists. This is Python's central data structure. We can use it as an array, a stack, a queue and other structures
- dicts. That is, a dictionary, which is the Python version of a map.
- Other things which the documentation calls types. These include range, tuple, str (for string), set and bytearray
- Structures defined in other modules, especially deque, a double-ended queue, in the collections module
- The ability to define our own classes. Through that we can define our own data structures, either using the above or not.

These are *mutable* (can be changed) or *immutable* (cannot be changed) They mostly have a set of methods called 'common sequence operations', and some mutable operations if appropriate.

Lists

A Python list can be used as a basic data structure, and can be used in different ways. For example

```
x = 28
myList = [5, 2, 47, 'a', 'b', 'c', x] # lists have [ square brackets ]
#  use as array
print(myList[2])  # 47 -  has index 2 (starts at 0)
myList[2] = "Hello" # we over-wrote the 47
print(myList[2])  # elements not all same type
print( myList) #  [5, 2, 'Hello', 'a', 'b', 'c', 28]
myList.reverse() # invoke a method
print( myList) #  [[28, 'c', 'b', 'a', 'Hello', 2, 5]
myList.append(999) # [28, 'c', 'b', 'a', 'Hello', 2, 5, 999]
# append adds an element to 'the end'
another=[1,2,3]
myList.extend(another) # extends joins another list on the end
print( myList) #  [28, 'c', 'b', 'a', 'Hello', 2, 5, 999, 1, 2, 3]

# use as stack : push and pop at index 0
myList.clear() # empty list
myList.insert(0,4) # push - add at index 0
myList.insert(0,6)
myList.insert(0,9)
while len(myList) != 0:
    print(myList.pop(0)) #9 6 4 - popping from index 0 - LIFO
```

```
# use as queue : enqueue at 0, dequeue from end
myList.clear()
myList.insert(0,4)
myList.insert(0,6)
myList.insert(0,9)
while len(myList) != 0:
    print(myList.pop(len(myList)-1)) # 4 6 9 - remove from end
```

Dicts

Python calls a map a *dictionary*, with the structure named *dict*:

```
myMap = dict()
myMap[56]="New York" # insert the key 56 and value New York into the map
myMap[72]="London"
myMap[16]="Paris"
myMap[32]="Rome"
print(myMap[16]) # Paris
try:
    print(myMap[4]) # no pair with key 4
except KeyError:
    print("No such key") # this is output
for k in myMap.keys(): # get all keys
    print(k) # 56 72 16 32
for v in myMap.values(): # and all values
    print(v) # New York London Paris Rome
```

There is no control over the data types of keys or values.

As an example use, suppose we want to count how many times each character is used in a string. We somehow want a count for each character used. We can ignore characters not used - in other words, with a count of zero.

We can do this using a map, using as key a character, and as value, a count of how many times it is used. For each character in the string, we check if it is in teh map. If not, we put it in, with a count of 1. If it is already in, we increase its count by 1:

```
# make frequency table of chars in a string
str="Test test input"
countMap=dict() #empty map
# keys will be the chars in the str,
# and the value, the count of that char
for c in str: # for each character
    val = countMap.get(c) # the count so far
    if  val is None: # not yet in
        countMap[c]=1 # so start at 1
    else: # add 1 to count
        countMap[c]=val+1
for c in countMap.keys(): # output result
    print(c, countMap[c] )
```

Output is

```
s 2
t 4
p 1
u 1
T 1
i 1
e 2
n 1
  2
```

Other built-in types

A Python *tuple* is a read-only list:

```
myTuple=(4,"Hello", 6.6) # round brackets
print( myTuple[0]) #4
print( type(myTuple[1]))  # <class 'str'>
print( 4 in myTuple) # True
```

A *range* is sequence of integers with a fixed common difference - not a data structure. It is usually used with a for loop:

```
myRange = range(3,6)
for x in myRange:
    print(x) # 3 4 5
```

A *set* is a structure with no duplicates. A frozenset is a read-only set.

A *bytearray* is intended to store a large binary object in some format, such as an MP3.

A Python *array*, in the array module, has all elements the same type, with the type being a basic simple one such as a character or a number.

Stack implementations

A stack is a LIFO ADT. This section gives examples of some ways that could be implemented in Python.

Using a Python list

We simply push and pop elements at the 'end', at largest index. We push by using append, and pop using the pop method with no argument, which defaults to removing from the end:

```
myStack=[]  # new empty stack
# push 4,5,6 in that order
myStack.append(4)
myStack.append(5)
myStack.append(6)
while not len(myStack) == 0:
    print(myStack.pop()) # 6 5 4
```

A class wrapping a list

We can define a Python class wrapping a list - in other words containing a list as a field:

```
class Stack:
    def __init__(self):
        self.data=[]
    def push(self,x):
        self.data.append(x)
    def pop(self):
        if self.isEmpty():
            return None
        else:
            return self.data.pop()
    def isEmpty(self):
        return len(self.data)==0


myStack=Stack()
# push 4,5,6 in that order
myStack.push(4)
myStack.push(5)
myStack.push(6)
while not myStack.isEmpty():
    print(myStack.pop()) # 6 5 4
```

This is just *syntactic sugar*. This means the source code looks neater, simpler and clearer, but we are not adding any real functionality, over simply using a list as a stack.

Array as stack

We use an array as a stack. We have a value 'stackTop' which is the index in the array where a new value will be pushed. This is initially 0, since this is where the first value is pushed in an empty array. The push process is

1. store value at stackTop
2. increment stackTop

The pop process is

1. Decrement stackTop
2. Return value at index stackTop

In this version we use a Python list as the array. We initialise this to have SIZE elements, so the stack is full once this number of values have been pushed. In fact Python lists can grow - but this is how it works for usual arrays:

```
class Stack:
    def __init__(self, size):
        self.SIZE=size;
        self.data=[1] # arbitrary first element
        self.data*=size # now have 'size' elements
        self.stackTop=0
    def push(self,x):
        if self.stackTop==self.SIZE:
            return # stack full
        # if not full
        self.data[self.stackTop]=x
        self.stackTop+=1
    def pop(self):
        if self.isEmpty():
            return None
        else:
            self.stackTop-=1
            return self.data[self.stackTop]
    def isEmpty(self):
        return self.stackTop==0


myStack=Stack(100)
# push 4,5,6 in that order
myStack.push(4)
myStack.push(5)
myStack.push(6)
while not myStack.isEmpty():
    print(myStack.pop()) # 6 5 4
```

This corresponds to how stacks are implemented at the processor native code level. A block of memory, the stack segment, is used to hold the stack. There are push and pop

instructions. A register, the *stack pointer*, holds the address of the top of stack. If the memory segment is full, we get a *stack overflow* condition.

Node structures

We often implement an ADT by using linked nodes. That means we set up nodes which have fields for data values, and also pointers for links to other nodes.

A pointer is a way of storing something which is a direction to something else, as a road sign shows the way to another town. How we do this depends on what the language allows:

- One type of pointer is simply an index into an array. For example

```
where = 27 # here points to the array cell with index 27
myData[where] = 88 # store 88 in the array cell pointed at by 'where'
```

- Another type of pointer is the address in RAM where a value is held. This is how pointers are handled in C. These are called raw pointers or exposed pointers. For example

```
int x = 88;  ; standard int variable
int * where; ; where is a pointer to an int
where=&x;    ; where set to the address of x - where points to x
*where = 27; ; write 27 to that location
```

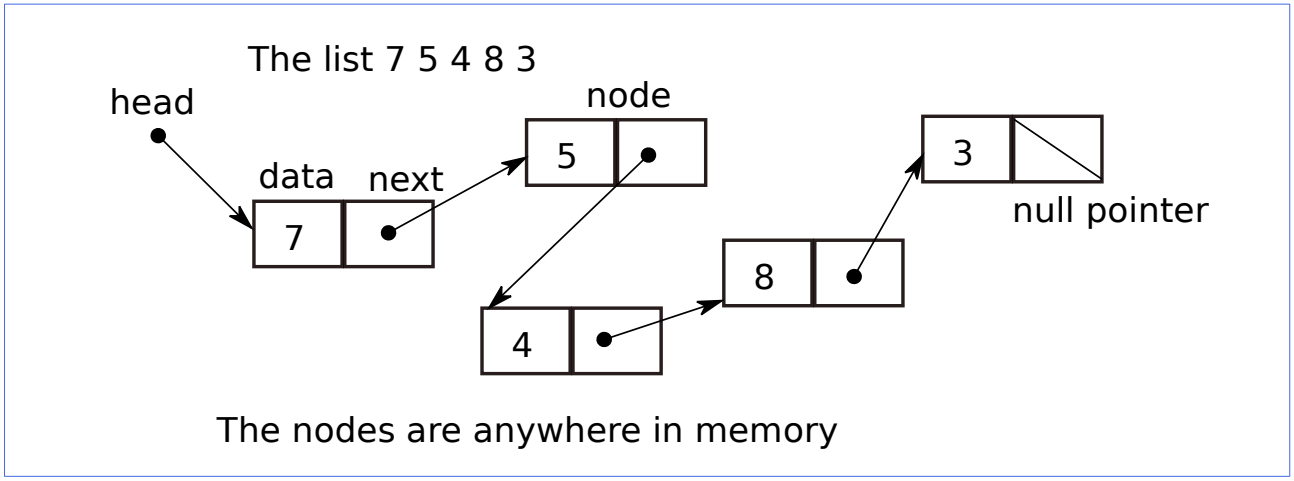
- RAM address pointers are powerful, but require the programmer to manage memory use for themselves. Some languages do memory management for you, and instead offer references. A reference is in effect a RAM address but controlled to avoid many bugs. This is used in Java and Python, for example. If we say in Python

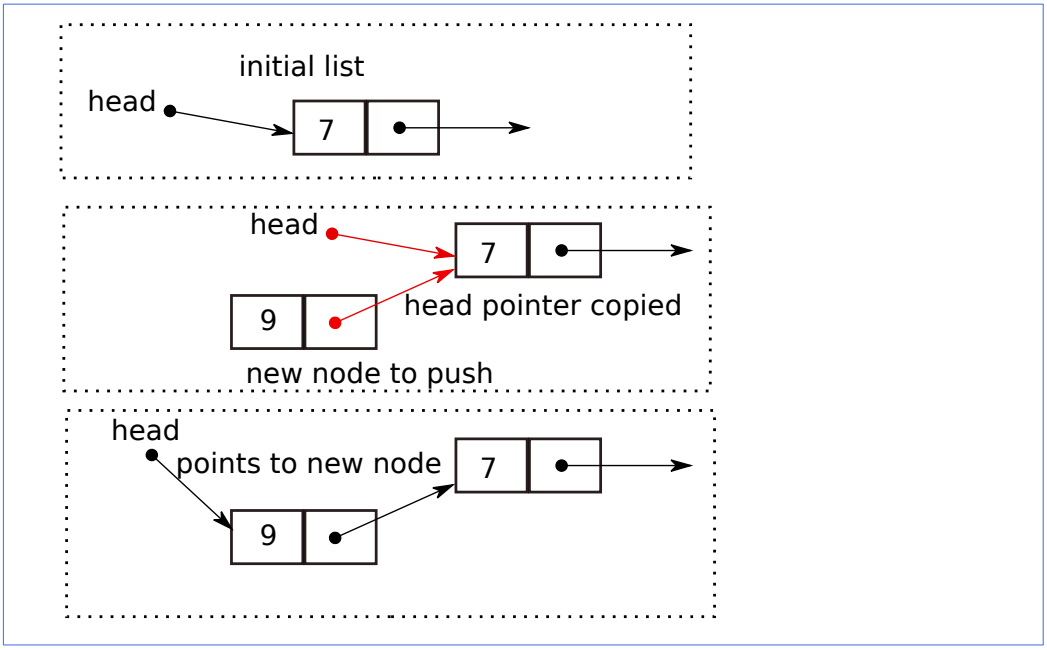
```
x = 27
```

Then x is not simply the value 27. It is a pointer or reference to an object. The object has type int, value 27, and has all the methods of class int.

Linked lists

A linked list is a simple example of using linked nodes to implement a list. The list is a chain of nodes, with each node having a link to the next one. This means the list does not need to be in one contiguous memory block.





Python is an OOP language, and so the appropriate way to do this is by defining suitable classes. We need to define two classes:

1. a node structure, with fields for links , and
2. the data structure made of those nodes

This uses a linked list as a stack:

```
# use a linked list as a stack
class ListNode:  # a node in the linked list
 def __init__(self, data):
   self.data = data
   self.next= None

class LinkedList: # has just one field : head
 def __init__(self):
   self.head=None  # empty list
 def push(self, data): # push a new value at head of list
   node=ListNode(data) # make node
   if self.head is None: # if list empty
     self.head=node # new node is at head
   else: # new node links to current head
     node.next=self.head
     self.head=node # head is now the new node
 def pop(self): # pop value from head
   if self.head is None:
     return None
   val=self.head.data # copy value at current head
   self.head=self.head.next # head is now next one
   return val
 def isEmpty(self):
   return self.head is None
```

```
# try it out
myList=LinkedList()
myList.push(56)
myList.push(48)
myList.push(21)
while not myList.isEmpty():
 print(myList.pop()) # 21 48 56
```

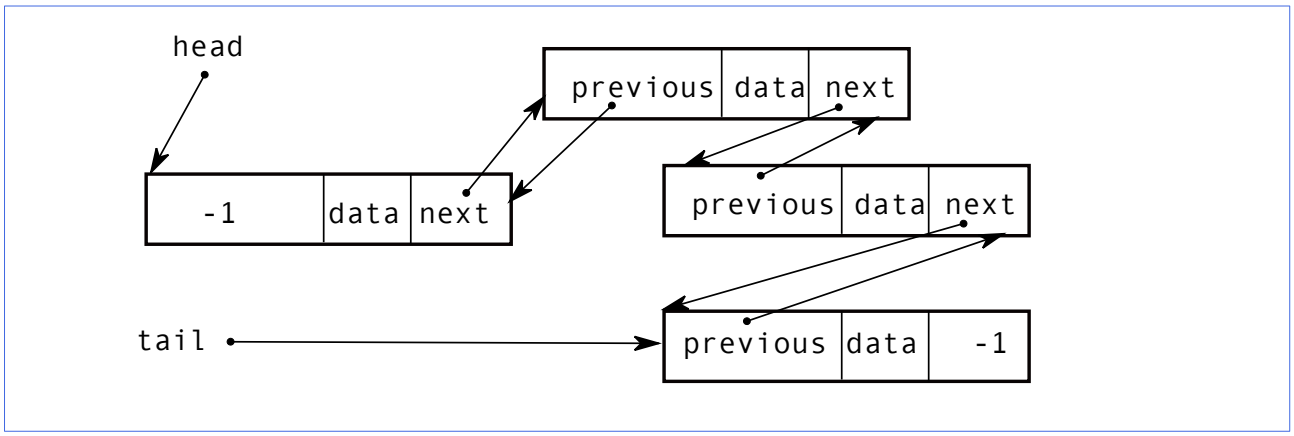
Queue implementations

Doubly-linked lists - queues

A doubly-linked list has nodes with two pointer fields - to the next node, and to the previous node.

We can implement a queue ADT as a singly linked list. We add new nodes at the head, and dequeue nodes as the last node. We can find the last node, by traversing through from the head until we find one whose next field is null. But this is slow for a large list. We also need to know which is the node next to last, because it becomes the new last node.

A faster implementation is to use a doubly-linked list, and maintain pointers to the head and tail:



This diagram uses -1 to mean a 'pointer to nowhere', the null pointer, represented as None in Python.

In Python we can use a list as a queue, but here is an implementation as a doubly-linked list:

```
  # use a doubly-linked list  as a queue
class ListNode:  # a node in the linked list
    def __init__(self, data):
        self.data = data
        self.next= None
        self.previous = None

class Queue: # implement as doubly-linked list
# enqueue at head, dequeue from tail
    def __init__(self):
        self.head=None  # empty list, no nodes
        self.tail=None
    def enQueue(self, data): # place a new value at head of list
        node=ListNode(data) # make node
        if self.head is None: # if list empty
```

```
            self.head=node # new node is at head
            self.tail=node
        else: # new node links to current head
            node.next=self.head
            self.head.previous=node
            self.head=node # head is now the new node

    def deQueue(self): # get value from tail
        if self.tail is None:
            return None
        val=self.tail.data # copy value at current head
        self.tail=self.tail.previous # head is now next one
        if self.tail is None:
            self.head=None
        return val

    def isEmpty(self):
        return self.head is None

# try it out
myList=Queue()
myList.enQueue(56)
myList.enQueue(48)
myList.enQueue(21)
while not myList.isEmpty():
    print(myList.deQueue()) # 56 48 21
```

The queue module

Python has a queue module containing implementations of various types related to queues. In this case we do not know how the queue is being implemented - an example of abstraction:

```
import queue as q

myQ = q.Queue()
myQ.put(23)
myQ.put(24)
myQ.put(25)
while not myQ.empty():
    print(myQ.get()) # 23 24 25
```

Priority queues

A priority queue is an ADT which is an *ordered list* (using list in the general sense, not as a Python list). Nodes have a 'priority' field. There is just one required operation - data removed from the priority queue has lowest (or highest) priority:

```
import queue as q

myPQ = q.PriorityQueue()
```

```
myPQ.put(9)
myPQ.put(4)
myPQ.put(25)
myPQ.put(17)
while not myPQ.empty():
    print(myPQ.get()) # 4 9 17 25
```

Linear searches

This means a search (looking for target data) in a linear data structure, as opposed to a search where the data is not in a line - such as a tree.

We usually want to find *where* the data is. A possible result is that the data is not in the structure.

Python has the built-in list structure, and that has the index() method to find a value. We look at how that might work

Sequential search

A sequential search is simply:

1. Start at the beginning

2. Look at each item in turn, and stop if you get a match

3. If no match, its not there.

In Python:

```
def search(lyst, target):
    for index in range(0,len(lyst)):
        if lyst[index] is target:
            return index
    return None

myList=[7,2,9,6,4,3,8,5]
print( search(myList, 9))  # 2
print( search(myList, 88)) # None
```

Speed of a linear search

The actual speed, in milliseconds, will depend on how fast the device is. A better comparison is counting how many steps are needed.

We make some changes:

1. We count the steps

2. We do it for lists 1000, 2000, 3000.. 110000 elements long

3. Since we do not want to input thousands of values, we use random data:

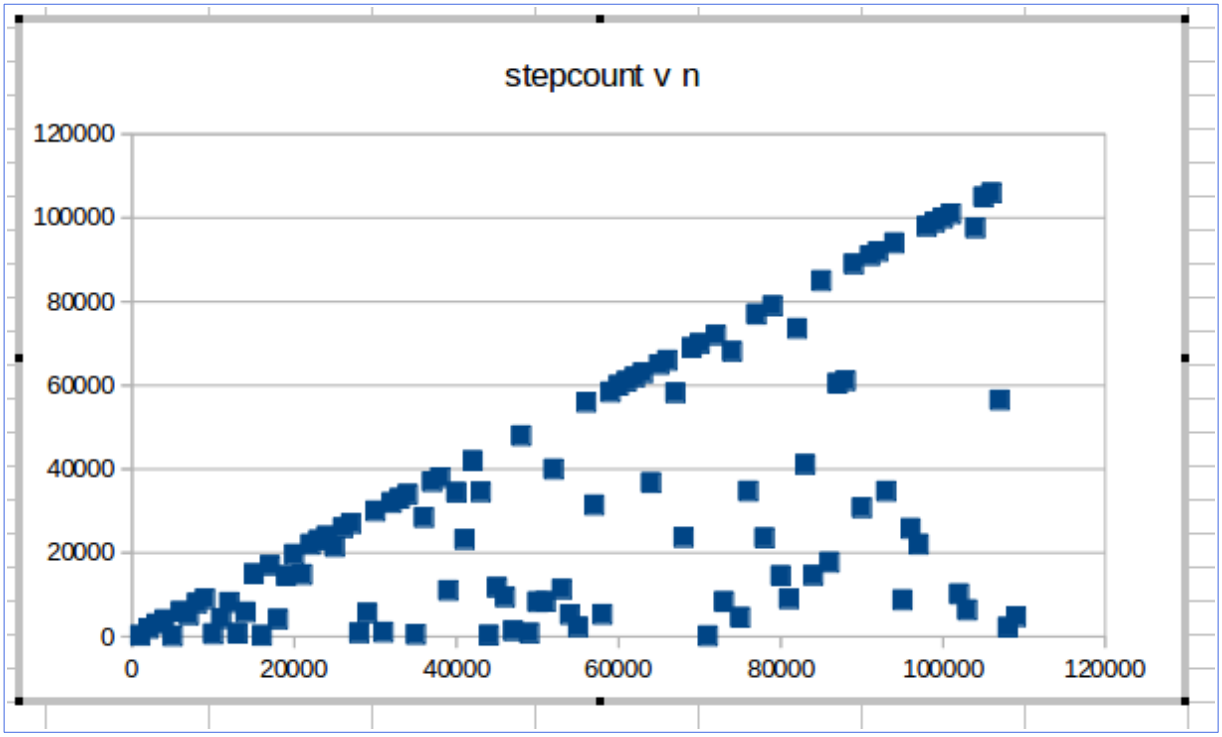
```
# linear search
import random
```

```
def search(lyst, target):
    global stepCount
    for index in range(0,len(lyst)):
        stepCount+=1
        if lyst[index] == target:
            return index
    return None

for n in range(1000, 110000, 1000):
    myList=[]
    for i in range(0,n):
        myList.append(random.randrange(n))
    stepCount=0
    search(myList, random.randrange(n))
    print(n, stepCount)
```

The output of this starts:

```
1000 1
2000 5
3000 1
4000 47
5000 93
6000 19
7000 5
8000 199
```

If we graph the stepcount, against the size of the list, we get:



Often the target is not in the list, so we have n repeats. Sometimes we are lucky, and find the target towards the start of the list. Sometimes it is towards the end. So this ranges from 1 to n. Can we find a faster method?

Bisection search

This only works if the list is in order.

1. We have two pointers, lo and hi, which are indexes into the list. To start with lo=0 and hi=list length-1, so we cover the whole list
2. We find the middle of the section.
3. If this is the target we have found it - stop
4. If hi=lo+1, the section is only 2 elements long, and it is not present.
5. We change to either using the lower section, or the upper. If middle is too big, use the lower section - else the upper
6. Repeat from step 2.

In Python:

```
def search(lyst, target, lo, hi):
    global stepCount
    while True:
        stepCount+=1
        middle=(hi+lo)//2
        print(lo, middle, hi, " section length = ",hi-lo)
        if lyst[middle] == target:
            return middle
        if hi==lo+1:
            return None
        if lyst[middle]>target:
            hi=middle
        else:
            lo=middle



myList=[2,4,5,7,8,9,12,14,18,21,24]
stepCount=0
print(search(myList,7, 0, len(myList)-1))
print(stepCount)
```

This outputs:

```
0 5 10  section length =  10 # search whole list - middle at 5
0 2 5  section length =  5 # use lower 1/2
2 3 5  section length =  3 # upper half - middle at 3
3  # found at index 3
```

```
3  # took 3 steps
```

Or looking for a missing value:

```
myList=[2,4,5,7,8,9,12,14,18,21,24]
stepCount=0
print(search(myList,15, 0, len(myList)-1))
print(stepCount)
```

Output:

```
0 5 10  section length =  10 # whole list
5 7 10  section length =  5 # top 1/2
7 8 10  section length =  3 # top 1/2
7 7 8  section length =  1 # lower 1/2
None # not present
4 # took 4 steps
```

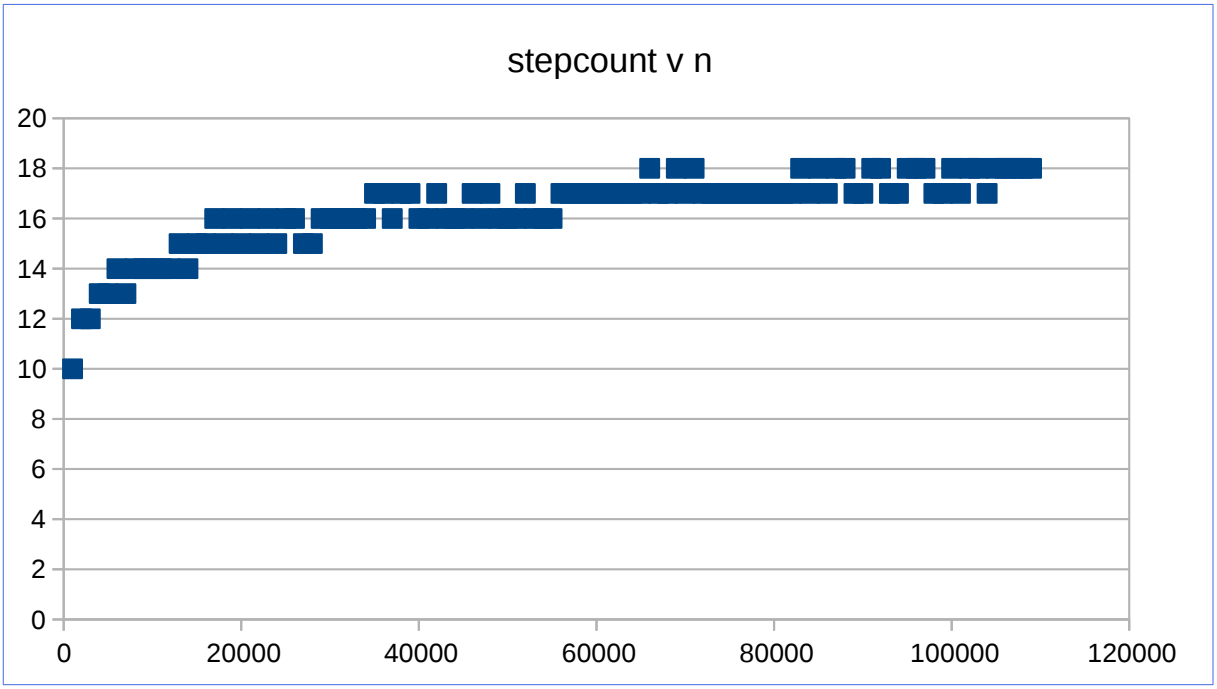
The worst case is if the target is missing. Then the number of steps is how many times we can halve the list length.

Binary search speed

We can modify the binary search code to see the number of steps needed as n changes, as for a sequential search:

```
 def search(lyst, target, lo, hi):
    global stepCount
    while True:
        stepCount+=1
        middle=(hi+lo)//2
        if lyst[middle] == target:
            return middle
        if hi==lo+1:
            return None
        if lyst[middle]>target:
            hi=middle
        else:
            lo=middle



for n in range(1000, 110000, 1000):
    myList=[]
    val=0
    for i in range(0,n):
        val+=random.randrange(n)//n # make sure
        myList.append(val)          # myList is in increasing order
    stepCount=0
    search(myList, random.randrange(n), 0, len(myList)-1 )
    print(n, stepCount)
```

If we plot this, we get:



This is a very different shape from a seqential search. It also takes around 18 steps for about 100,000 items, compared with around 50,000 steps with a sequential search. Why?

Each step in a binary search halves the list section - we reject half of it as too big or too small. How many times can we halve 100,000? Roughly:

50000, 25000, 12000, 6000, 3000, 1500, 750, 400, 200, 100, 50, 25, 12, 6, 3, 1

About 16 times.

Other way round: $2^{16} = 131072$

So the expected step count is how many times we can halve n. Mathematically that is $\log_2 n$ - log to base 2

Time complexity

The time complexity of an algorithm means how the number of steps it takes changes with n, the number of data items, for large n.

Why?

We count the steps, rather than measure the actual time, since this will depend on the clock speed of the device (and many other things, like what else it is doing at the same time, what language it is written in). The step count just depends on the algorithm.

And we only care about very large n. The device has a clock speed of around 1 billion steps a second, so any algorithm will be very fast if n is a few hundred. But it may take hours if n is a million or so.

Mathematically this is called the *asymptotic behaviour*.

We will have best, worst and average cases.

But for a sequential search, we need around n/2 steps, and for a bisection search, around log n.

So

- a sequential search is slow, to be avoided if possible, but
- a bisection search needs data in order - and sorting takes time - see later

This is the *time complexity* of an algorithm - how many steps it needs.

Space complexity is how much memory space it requires. Many algorithms are in place, which means they process a data structure without using an extra memory.

Partition algorithms

A partition algorithm is one that works by splitting data structures into parts.

A bisection search is an example, splitting a list into a a smaller part and a larger part.

Because partition algorithms divide data structures by 2, there time complexity often involves $\log_2 n$.

Linear sorts

Radix sort

Suppose we have a set of 2 digit integers, and we want to put them into increasing order:

12 34 45 66 23 19 21 67 46 17 28 39 41 77 98 81 18 25 31 88 52 55 51

We could separate these into 9 parts, based on the *second digit*:

21 41 81 31 51

12 52

23

34

45 25 55

46

66

67 17 77

28 98 18 88

19 39

each part is in the same sequence as in the original

Next, put these 9 parts together, in the same sequence

21 41 81 31 51 12 52 23 34 45 25 55 46 66 67 17 77 28 98 18 88 19 39

Then put them into 9 piles, based on the *first digit*

12 17 19

21 23 25

34

41 45 46

51 52 55

66 67

77

81 88

98

and put them together

12 17 19 21 23 25 34 1 45 46 51 52 55 66 67 77 81 88 98

This method is known as a *radix sort*.

We could do this with data which was alphabetic, not numeric, by having 26 piles, not 9 or 10.

So what?

The 1880 US Census data took 8 years to process, so a better method was needed for the 1890 Census. A man named Herman Hollerith suggested putting the data onto punched cards. The cards could then be put through a machine to separate them into 26 piles, as above, on the last letter, put them together, separate again on the second letter, put together, and so on for every letter. The result was a set of cards in order.

Hollerith went on to start IBM.

Why name it radix sort? Ordinary numbers are decimal, radix 10. In a computer we have binary data, radix 2. Alphabetic data is radix 26. A radix sort does a separation into a set of piles, for each radix position in the data.

How fast is radix sort? If we have n data items, one run through takes n steps (one per card through the whole deck). How many times do we have to do it? The number of digits in the data - the key length. So overall this is key length X n. This is very fast - faster than quicksort which in n log n, the most common software sort.

Why is it called 'sorting'?

And not 'ordering', which is what it really means?

To 'sort' data means to separate it into different sorts - to classify it into different categories. This is how the original radix sort worked, on punched cards. As a result, ordering has come to be known as 'sorting'.

A linear sort means sorting data in a line - a list. We could sort data using other structures, such as trees for example.

Bubble sort

This is usually the first sort algorithm taught - but it is usually too slow to use.

The basic algorithm is

1. Scan through the list, from the start, comparing each node with the next, and exchange if they are in the wrong order.

2. Step 1 moves an element into its correct place. So repeat 1, for however many items there are:

```
def bubble(list):
    "Do a bubble sort on the list"
    n=len(list)
    global steps
    for count in range(1,n): # do this n-1 times
        print(list)
        for index in range(0, n-1):
            # index goes 0 to n-2 inclusive = n-1 repeats
            steps+=1
            if list[index] > list[index+1]:
                # compare element with the next
                # we access index+1 - so index goes to n-2,
                # and at the end
                # we compare n-2 and n-1  last 2 in the list
                temp=list[index] # do the swap here
                list[index]=list[index+1]
                list[index+1]=temp
    return list
steps=0
myList=[6,2,7,3,4,1,9,5]
myList=bubble(myList)
print(steps)
```

The output, showing the list at the start of each count loop, is:

```
[6, 2, 7, 3, 4, 1, 9, 5] # initial list
[2, 6, 3, 4, 1, 7, 5, 9] # 9 swept to final position.6 moved
[2, 3, 4, 1, 6, 5, 7, 9] # 7 in final position
[2, 3, 1, 4, 5, 6, 7, 9] # 6 7 9
[2, 1, 3, 4, 5, 6, 7, 9]
[1, 2, 3, 4, 5, 6, 7, 9]
[1, 2, 3, 4, 5, 6, 7, 9] # no change
[1, 2, 3, 4, 5, 6, 7, 9] # final order
49
```

This also counts the steps in the algorithm. The data count is n = 8 items. The count loop repeats n-1 times, and each time, the index loop repeats n-1 times. So the total steps is 7 X 7 = 49.

Can we improve this? Each count loop puts one element into its correct place. So the index loops can get shorter - no need to scan all the way through:

```
        for index in range(0, n-count):
```

This brings the step count down to 28.

Other improvements are possible - such using a flag to check if any swaps are made in a scan, and stopping once we find there are none.

Even then a bubble sort on n items needs around n^2 steps, which is too slow for most n (for example sorting a thousand items will require a million steps).

Best, average and worst cases

Suppose we do a bubble sort on data which is already ordered? In other words

```
myList=[1,2,3,4,5,6,7,8]
myList=bubble(myList)
```

In fact as coded here this still requires the same number of steps.

But in general the performance of a sort will depend on the initial order of the data. We can usually find the best (fastest) and worst cases, and the average case, which means random data.

Stable sorts

Suppose we have

```
myList=[1,2,3,4,5,3,7,8]
myList=bubble(myList)
```

Here we have 2 equal values. Will the sort keep the green 3 before the yellow 3? If so, it is called a stable sort. If it (pointlessly) reverses them, it is unstable.

Bubble sort is stable.

Order

A sort arranges the data into order. if the data is numeric, it is clear what 'order' mean. For string data is is usually lexicographic order - as in a dictionary. But if it is some other data type, it may be unclear what we mean by something being 'greater' than something else. Then we need to define that, and supply it to the sort method.

For example, suppose our data type is a pair of integers, x and y, and we order them on x, ignoring y. So (8,2) > (6,3) and (4,0) > (2,2). We can define a class like this with a compare method:

```
class Pair:
    def __init__(self,x,y):
        self.x=x
        self.y=y
    def compare(self, other):
```

```
        return self.x>other.x
    def __str__(self):
        return '('+str(self.x)+','+str(self.y)+')'
```

then sort a list of Pairs:

```
def bubble(list):
    "Do a bubble sort on the list"
    n=len(list)
    for count in range(1,n): # do this n-1 times
        for index in range(0, n-count):
            if list[index].compare(list[index+1]):

                temp=list[index] # do the swap here
                list[index]=list[index+1]
                list[index+1]=temp
    return list
```

used like:

```
myList=[Pair(1,2), Pair(5,3), Pair(2,2), Pair(3,6)]
myList=bubble(myList)
for x in myList: print(x)
```

outputting

```
(1,2)
(2,2)
(3,6)
(5,3)
```

Exchange sort

The algorithm is:

1. Find the smallest, and exchange it with the first

2. Find the next smallest (miss out the first) and exchange with the second

3. Find the third smallest.. and so on

```
def exchangeSort(list):

    n=len(list)
    for start in range(0,n-1): # start at 0, up to n-2
        # find smallest from start to end
        where=start # initialise at start
        smallest=list[start]
        for index in range(start+1,n): # check rest
            if list[index]<smallest:
                smallest=list[index] # found new smallest
                where=index
        # exchaneg smallest with start
```

```
        temp=list[where] # do the swap here
        list[where]=list[start]
        list[start]=temp
    return list

myList=[9,3,8,6,7,2,4]
myList=exchangeSort(myList)
print(myList) # [2, 3, 4, 6, 7, 8, 9]
```

The analysis is similar to bubble sort. Scans get shorter - they start at the length of the whole list (=n) and g down to 1. That averages n/2. We must do this n times, so the number of steps is $n^2/2$.

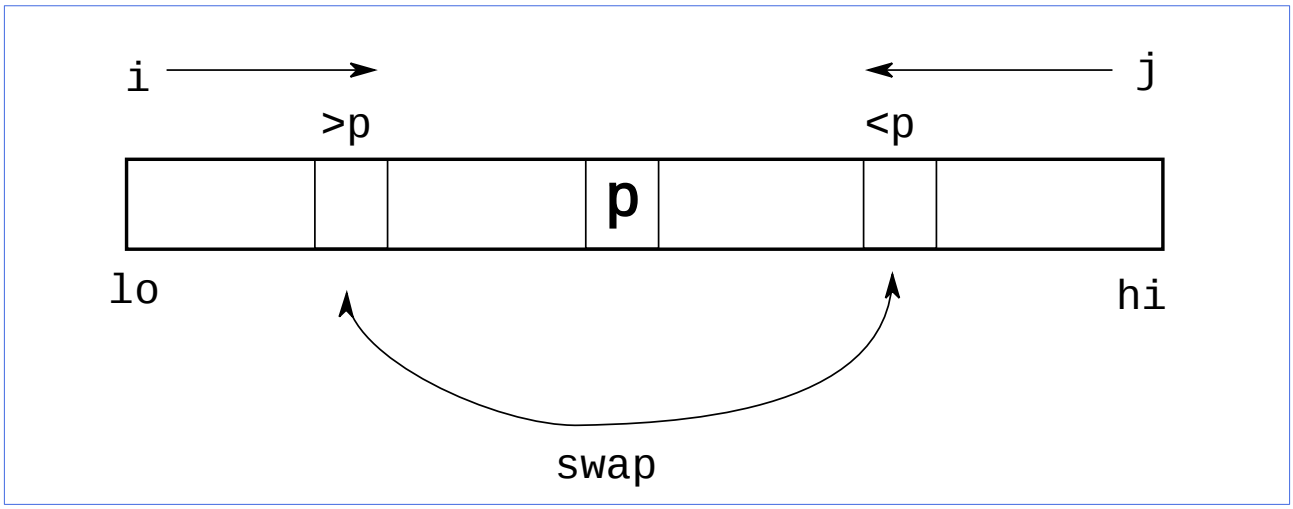
In fact, if we have n-1 elements in the correct place, the last one must also be in its correct place. But that makes little difference - exchange sort is around n^2, and that is too slow for n of more than around 100.

Quicksort

The outline idea is as follows. We pick an element, say the first one. This is usually called the pivot. We have two pointers which start at the ends. Call them i (start next to the pivot) and j (start at the end). We move i up and j down. If the element at i is more than the pivot, and at j is less, we swap the values at i and j. This continues until i and j meet.

This partitions the list into two parts. All values in the first part are less than the pivot, and all in the second bigger.

We do this recursively on the two parts until the parts are only 1 long. Then the list is sorted.



In Python:

```
def quicksort(lyst, lo, hi):
    if lo < hi:
```

```
        p= partition(lyst, lo, hi)
        print("Partition", p)
        quicksort(lyst, lo, p)
        quicksort(lyst, p + 1, hi)

def partition(lyst, lo, hi):

    pivot= lyst[(hi + lo) // 2 ]
    print("lo=",lo," hi=",hi, " pivot=", pivot,  lyst)
    i= lo - 1
    j= hi + 1
    while True:
        while True:
            i+=1
            if lyst[i] >= pivot: break
        while True:
            j-=1
            if lyst[j] <= pivot: break
        if i >= j:
            return j
        temp=lyst[i]
        lyst[i]=lyst[j]
        lyst[j]=temp

myList=[9,3,8,6,7,2,4]
quicksort(myList,0,len(myList) - 1)
print(myList) # [2, 3, 4, 6, 7, 8, 9]
```

```
The output of this, with notes, is:
lo= 0  hi= 6  pivot= 6 [9, 3, 8, 6, 7, 2, 4]start whole list
Partition 3
lo= 0  hi= 3  pivot= 3 [4, 3, 2, 6, 7, 8, 9]lower part. pivot
Partition 1
lo= 0  hi= 1  pivot= 2 [2, 3, 4, 6, 7, 8, 9]
Partition 0
lo= 2  hi= 3  pivot= 4 [2, 3, 4, 6, 7, 8, 9]
Partition 2
lo= 4  hi= 6  pivot= 8 [2, 3, 4, 6, 7, 8, 9]upper part
Partition 5
lo= 4  hi= 5  pivot= 7 [2, 3, 4, 6, 7, 8, 9]
Partition 4
[2, 3, 4, 6, 7, 8, 9]
```

This is a *partition* algorithm. It splits the data into parts, and deals with each part. This is sometimes called a divide-and -conquer solution.

With random data, the number of times the n items can be divided is $\log_2 n$, and on each divide, all n items are scanned. So the speed is n log n - much quicker than bubble or exchange sorts for large n.

Quicksort worst case is when the data is already sorted, since it then the partition only splits of 1 item each time. We get n partitions, and the speed is n^2. Real quicksorts are usually modified to account for this. They also switch to another method for small n.

The C standard library sort is qsort, and uses this method.

Mergesort

To start with, suppose we have a function that merges two sorted lists into a single sorted list. So for example with input

list 1 : 2,4,7,8,9,12,15

list 2 : 3,5,6,10,13

the output is: 2,3,4,5,6,7,8,9,10,13,15

How would this work? We start comparing the heads of the 2 lists - 2 and 3. 2 is less, so put it to the output and remove it from list1. Then compare 4 and 3. 3 is less so output it and remove from list2. Now compare 4 and 5. 4 is less so output.. and so on. Continue until one input list is empty, and just move the rest of the other list to output.

Then, we use this function as follows. The input is an unsorted list - maybe

9,2,5,8,1,3,7,4,3,3,8

Treat this as a sequence of lists, 1 element long, and merge them pairwise.

9	2	5	8	1	3	7	4	3	3	8
pair 1 - merged below		pair 2		pair 3						
2	9	5	8	1	3	4	7	3	3	8

Then we take pairs 2 elements long, and merge them:

2	9	5	8	1	3	4	7	3	3	8
pair 1				pair 2				pair 3		
2	5	8	9	1	3	4	7	3	3	8

Then pairs 4 elements long. In fact, one pair with 4 elements. The other pair has length 3 and length 0

2	5	8	9	1	3	4	7	3	3	8
pair 1								pair 2		
1	2	3	4	5	7	8	9	3	3	8

(pair 4 has zero length). Then pairs 8 elements long. In fact we have one length 8, merging with one length 3

1	2	3	4	5	7	8	9	3	3	2
pair 1										
1	2	2	3	3	3	4	5	7	8	9

One scan through takes n steps, through the whole list. How many scans? The sublist lengths were 1,2,4,8... Going backwards, this halves the list each time, and the number of scans is how often we can divide n by 2 - which is log2 n.

So the total steps is $O[n \log_2 n]$. This is much faster than a bubble sort.

In Python:

```
# mergesort

def merge(list1, list2):
  "Merge 2 sorted lists and return result"
  result = []  # initialise a local variable
  # until one list is emptied..
  while len(list1) != 0 and len(list2) != 0:
    if list1[0] < list2[0]:  # if list1 head is smaller..
      # remove and add to result
      result.append(list1.pop(0))
    else:  # do the same for list2
      result.append(list2.pop(0))
  if len(list1) == 0:  # list1 emptied first
    result.extend(list2)  # copy all list2 to output
  else:  # the other one
    result.extend(list1)
  return result


def mergeSort(list):
  res = []
  listLen = 1
  while listLen < len(list):
    newList = []                      # used to hold merged sublists
    while (len(list) > 0):
      l1 = list[0:listLen]            # get first part of list into l1
      list = list[listLen:]           # and lose it
      l2 = list[0:listLen]            # same into l2
      list = list[listLen:]
      newList.extend(merge(l1, l2))  # put at end of newList
```

```
    list = newList                    # put back to list
    listLen *= 2
  return list


list = [56, 2, 6, 3, 4, 78, 99, 32, 14]
list = mergeSort(list)
print(list)                          # [2, 3, 4, 6, 14, 32, 56, 78, 99]
```

The shunt yard algorithm

This is a classic algorithm using stacks and queues.

The problem is in compilers and interpreters, handling expressions like '2+3*4+5'. We would like to scan this left to right, working it out as we go along. But we must do the 3*4 before the additions, and this is in the middle of the expression. How to work it out?

Operators and operands

An operator stands for some action - here, numeric. So + - * and / are the standard operators.

An operand is a value which an operator works on. So in '3+4' there are 2 operands, 3 and 4. Operands are numbers, or they might be variables representing numbers, like x+y.

Some operators are *binary*. This means they apply to two operands. So * is a binary operator - you multiply 2 numbers.

Some operators are *unary*. They apply to one operand. For example, unary minus. In '-4+2' the '-' is unary. The other common unary operator is +, as in '+62-34'.

Note '-' and '+' might be unary or binary, depending where they are in an expression.

Precedence

Precedence controls the order in which operators are applied. An operator with higher precedence is applied first. So 2+3*4 is 14, not 20.

Infix, pre-fix and post-fix

Evaluating a normal expression, like 2+3*4+5, cannot be done simply scanning left to right. We change to another form which can.

This is about how expressions are written. In standard notation, the operator is written *between* the operands. So for example, 2+3. This is called infix notation.

An alternative is to write the operator *after* the operands. So, 2 3 +. This is called post-fix (or sometimes, reverse Polish).

Or put the operator *before* the operands, like + 2 3. This is pre-fix notation.

We focus here on post-fix. For example infix 2+3*4 is 2 3 4 * + postfix.

Why? The operator comes after the operands. So in 2 3 4 * +, the * applies to the 3 and 4 before it - so it means 3*4. And the + applies to the two things before it - which are 2 and 3*4.

Another example: 1-2+3*5+4 is 1 2 - 3 5 * 4 + in post-fix. Note the order of the operands is unchanged - 1 2 3 5 4.

Post-fix evaluation

We scan through the post-fix, left to right, and use a stack. Operands are transferred to the stack. If we have an operator, we pop values off the stack, apply the operator, and put the result on the stack. At the end we have one value left on the stack, which is the expression value.

For example, with 1+2*3+4, which in postfix is 1 2 3 * + 4 +:

Input	Action with first input item	stack after (top on right)
1 2 3 * + 4 +	start with 1	
2 3 * + 4 +	1 is operand - push on stack	1
3 * + 4 +	push 2	1 2
* + 4 +	push 3	1 2 3
+ 4 +	* is operand. Pop 2 and 3 off, *, push result back	1 6
4 +	+ is operand. Pop 1 and 6 off, +, push result back	7
+	push 4	7 4
	+ is operand. Pop 7 and 4 off, +, push result back	11

So the plan is

1. Convert infix input to postfix

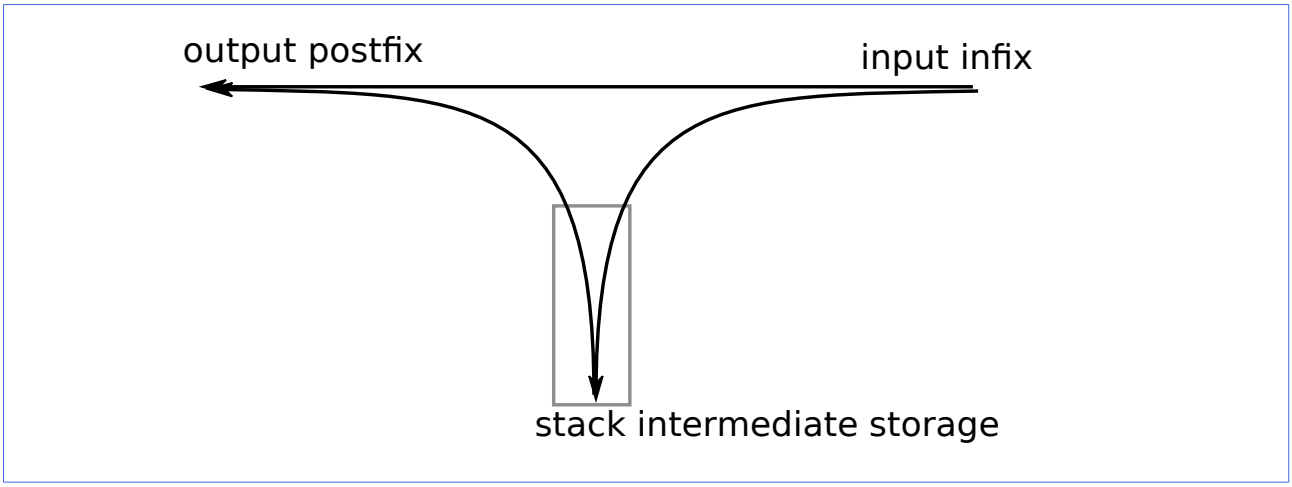
2. Evaluate postfix left to right as here.

The shunt-yard algorithm

This is the standard way to convert an infix to a postfix expression.

As an example, in 2+3*4, we cannot do the + first, even though we meet it first in a left-to-right scan. We must do the * first. But we must do the + later - so we must somehow store it for later use. We use a stack for that storage.

We have an infix input list, and a postfix output list. And a stack for intermediate storage of operators.



The algorithm in outline is to scan the input list left to right:

1. An operand moves straight to the output list

2. An operator is pushed onto the stack, if empty

3. If not empty, if it has higher precedence, it is pushed onto the stack (so will be done first). Else pop old one off stack to output, then push new one onto stack

4. When input empty, pop anything still on stack to output.

For example, on 2+3*4-1

Input	Stack	Output	Notes
2+3*4-1		2	Operand straight to output list
+3*4-1	+		Operator to empty stack
3*4-1	+	2 3	Operand to output
*4-1	+ *	2 3	* has higher precedence than + so push
4-1	+ *	2 3 4	
-1	+ -	2 3 4 *	- has slower precedence than stack top
1	+ -	2 3 4 * 1	
	+ -	2 3 4 * 1 - +	inputy empty - pop stack

In the following code, the input and output are strings. We take the next (leftmost) character from the input as

```
    token = input[0] # get first char
    input = input[1:]# cut off first char
```

The code is

```
# infix to rpn

# set up
prec = {  # dictionary of precedence levels of operators
          '+': 1,
          '-': 1,
          '*': 2,
          '/': 2
}
operators = "+-*/"
stack = []  # main stack - we push and pop at index 0
# so push = stack.insert(0, data)
# pop = stack.pop(0)
output = ""

# start shunt yard algorithm
input = "2+3*4+1"
while len(input) != 0:
    token = input[0]
    input=input[1:]
    if token not in operators:  # its an operand
        output=output+token
    else:  # its an operator
        if len(stack) == 0 or prec.get(token) > prec.get(stack[0]):
            # higher precedence
            stack.insert(0, token)
        else:  # lower
            op = stack.pop(0)
            output+=op
            stack.insert(0, token)
while len(stack) != 0:  # pop all stack to output
    output+=stack.pop(0)
# shunt yard end
print(output) # 234*1++
```

Brackets

If the expression includes brackets, we need to handle something like

..(xxxx)..

We do this by modifying the algorithm:

When we input a (, push it onto stack, and continue with the xxxx

When we input the matching), pop the stack to the output untl we reach the pushed (

A (needs to be given low priority. So the code is:

```
# infix to rpn

# set up
prec = {  # dictionary of precedence levels of operators
          '(': 0,
```

```
          '+': 1,
          '-': 1,
          '*': 2,
          '/': 2
}
operators = "+-*/"
stack = []
output = ""

# start shunt yard algorithm
input = "2*(3+4)-1"
while len(input) != 0:
    token = input[0]
    input=input[1:]
    if token is "(":
        stack.insert(0, token)
        continue
    if token is ")":
        op = stack.pop(0)
        while not (op is "("):
            output+=op
            op = stack.pop(0)
        continue

    if token not in operators:  # its an operand
        output=output+token
    else:  # its an operator
        if len(stack) == 0 or prec.get(token) > prec.get(stack[0]):
            # higher precedence
            stack.insert(0, token)
        else:  # lower
            op = stack.pop(0)
            output+=op
            stack.insert(0, token)
while len(stack) != 0:  # pop all opStack onto stack
    output+=stack.pop(0)
# shunt yard end
print(output) # 234+*1-
```

Syntax errors

To keep it simple, this code assumes the input is correct - so that there is a (for every), for example. Real versions need to check for errors and report them.

Tokenisation

A token is a sequence of input characters that form what a syntactic element. Examples would be

12.34 or 3 or 48 number constants

+ or * or / an operator

x or y or res	a variable
(or)	brackets
sin or cos or log	a function

Some tokens are single characters - like 3 or + or (. But most are not.

For example if the input is

12.5+res

that is 8 characters, in 3 tokens: <number><operator><variable>

Real code must start by tokenising the input. That means working through the input recognising tokens - like 12.5 is a number, and is followed by +, an operator. This needs to refer to the grammar of the language, which defines what is valid. So 12.56 is a valid number, and 12.5.6 is not.

In our example, we are only using single digit numbers, so one character is one token.

Hash Maps

As described above, a map is an ADT made of a set of key-value pairs.

Python has a built-in map implementation, which it calls a dictionary.

We can implement our own using two parallel lists:

```
class SimpleMap:
    def __init__(self):
        self.keys=[] # 2 lists - keys and values
        self.values=[]
    def put(self,key, value):           # insert a key value pair
        if key in self.keys:            # already got this key?
            where=self.keys.index(key)  # if so, where?
            values[where]=value         # replace value there
        else:
            self.keys.append(key)       # else add new key
            self.values.append(value)   # and new value
    def get(self, key):                 # get value matching a key
        if key in self.keys:            # key in map?
            where=self.keys.index(key)  # where is it?
            return self.values[where]   # return corresponding value
        else:
            return None                 #key not in map

myMap=SimpleMap()
myMap.put(28,"New York")
myMap.put(32,"Paris")
myMap.put(17,"London")
print(myMap.get(32)) # Paris
print(myMap.get(18)) # None
```

This shows what a map does, but it is not a very good implementation. It is fragile - keys and values are linked by having the same index in two different lists. If somehow an element as removed from either list, entries would no longer match.

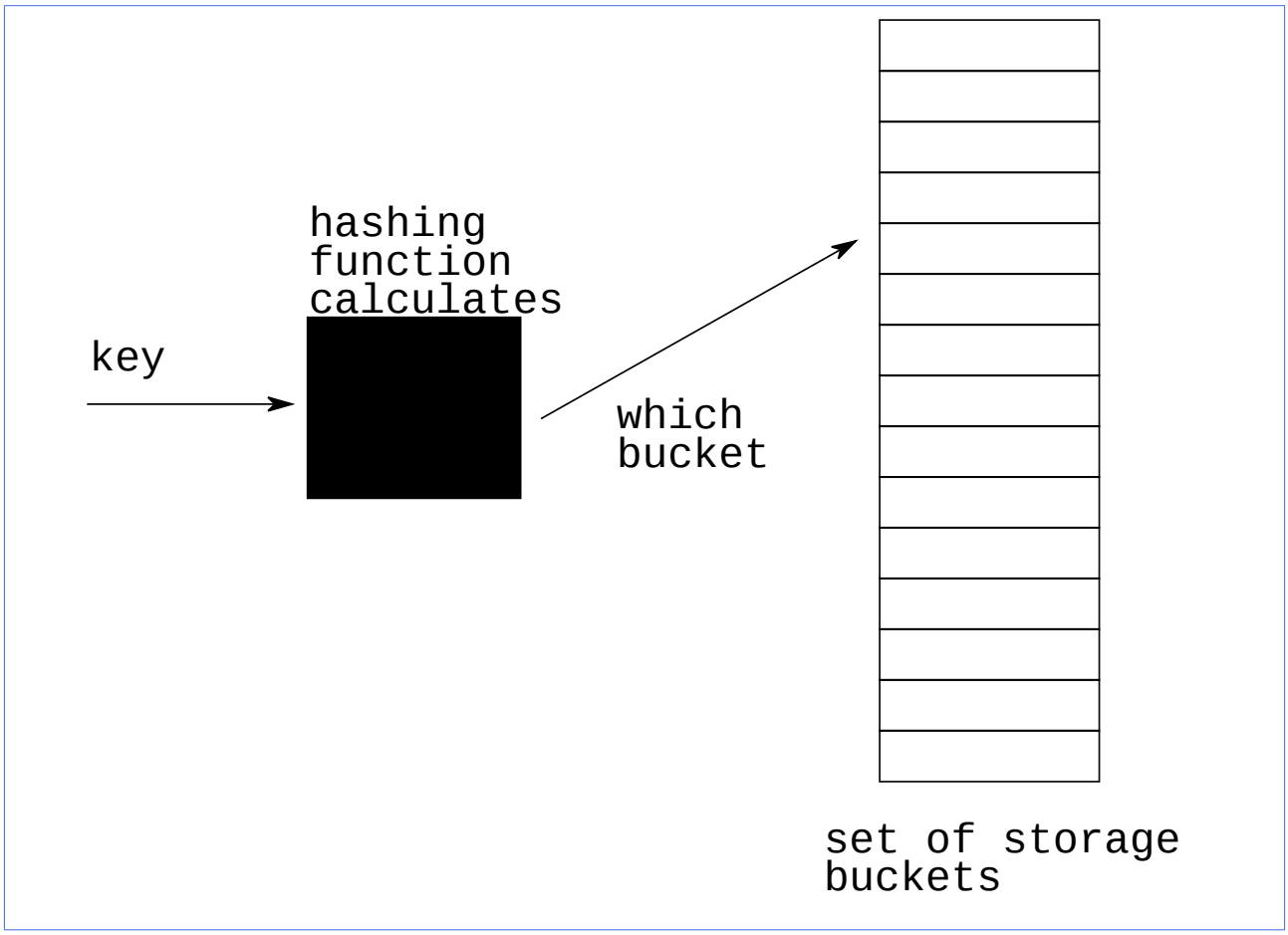
More importantly - it is slow. It uses index() to look for an item in the key list. This is a linear search, and its time complexity is O[n]. We would like a faster way to find a key.

Hashing functions

A hash table is one implementation method of a map ADT.

The aim is to make it very fast to find a key.

The answer is to *calculate* where it is. The hashing function is how that calculation is done.



When we insert a key and value pair, we use the hashing function to calculate which storage bucket it should go in.

When we want to fetch a value, we use the key, repeat the calculation, and just look in that bucket.

In the example listed here, we have 1000 buckets. The keys are just integers, and the hashing function is simply key mod 1000. So key 3458 hashes to bucket 458. Key 5871 hashes to bucket 871.

Collisions

A problem is that 2 different keys might hash to the same bucket. We can reduce the chances by having a very large number of buckets - and so, use a lot of memory. But for any finite bucket count, we will still have collisions.

In our example, 3458 hashes to 458. 7458 also hashes to 458. This is a collision.

Some way of dealing with collisions is needed. One way is to simply use the next available bucket. So to insert 7458, if bucket 458 is used, we look in 459, and store it there if free. If not, store it in 460 if free. And so on.


```
class HashTable:
  def __init__(self,n):
    # table has n buckets. Each bucket is a list with 2
    # elements a key and a value pair
    self.EMPTY=[0,0] # a constant, for an empty bucket
    self.SIZE=n
    self.buckets=[]       # empty list
    for n in range(0,n):  # add n empty buckets
      self.buckets.append(self.EMPTY)


  def hash(self, key): # very simple hash - range 0 to SIZE-1
    return key % self.SIZE

  def put(self, key, value):
    where = self.hash(key)  # initial location
    print(key,"hashes to",where)
    if self.buckets[where]==self.EMPTY or \
       self.buckets[where][0]==key:
      # if empty, put it there, or update current value
      self.buckets[where]=[key, value]
      print("Inserted at",where)
      return # finished
    else:
      print("Collision")
    while self.buckets[where]!=self.EMPTY and \
          self.buckets[where][0]!=key:  # look for
      where = (where+1)% self.SIZE      # next empty slot,
    self.buckets[where]=[key, value]    # wrapping around,
    print("Inserted at",where)          # and put it there
    return

  def get(self, key):
    where = self.hash(key)                # initial location
    if self.buckets[where]==self.EMPTY:   # if empty, must be absent
      return None
    if self.buckets[where][0]==key:       # found it
      return self.buckets[where][1]
    while self.buckets[where][0]!=key:    # or has been collision
      where = (where+1)% self.SIZE        # start linear search
      if self.buckets[where]==self.EMPTY: # to find it
        return None
    return self.buckets[where][1]



myTable=HashTable(1000)

# ordinary put
myTable.put(38677,"one")
myTable.put(747464,"two")
myTable.put(8377623,"three")
# and gets
print(myTable.get(38677)) # one
print(myTable.get(747464)) # two
```

```
# check not present
print(myTable.get(8677))  # None
# make collision
myTable.put(47464,"four")
# and get of collision
print(myTable.get(47464)) # four
# check can over-write key
myTable.put(747464,"new")
print(myTable.get(747464)) # new
```

Load factor

The load factor of a hash table is the fraction of the buckets which are in use.

So if the load factor is low, most buckets are empty.

The load factor affects how fast the hash table is.

We modify the code to measure how much. We change the get method to count how many steps are needed:

```
  def get(self, key):
    global stepCount
    where = self.hash(key)                # initial location
    stepCount=1
    if self.buckets[where]==self.EMPTY:   # if empty, must be absent
      return None
    if self.buckets[where][0]==key:       # found it
      return self.buckets[where][1]
    while self.buckets[where][0]!=key:    # or has been collision
      stepCount+=1
      where = (where+1)% self.SIZE        # start linear search
      if self.buckets[where]==self.EMPTY: # to find it
        return None
    return self.buckets[where][1]
```

Write functions to put and get random keys:

```
def putRandom(table):
  key = random.randrange(10*SIZE)
  value="xxx"
  table.put(key, value)

def getRandom(table):
  key = random.randrange(SIZE)
  table.get(key)
```

and measure the average search:

```
SIZE=1000
LOADFACTOR=0.2
myTable=HashTable(SIZE)
```

```
for count in range(0, int(LOADFACTOR*SIZE)):
  putRandom(myTable) # fil table to load factor
total=0
for go in range(0,100):
  stepCount=0 # find average search count
  getRandom(myTable)
  total+=stepCount
print(total/100)
```

With a load factor of 0.2, the average search count is 1.24.

With a load factor of 0.9, it is 22.28.

It is not hard to see why. If the load factor is high, collisions will be common, and we will have a lot of runs of buckets filled with keys that have hashed to then same location. A get will often hit one of these, requiring a a linear search to find it.

Even if we use a different way to handle collisions, a higher load factor will always slow gets.

The usual solution to a high load factor is

1. Create a new, larger, set of empty buckets

2. Use a different hash function to map to the larger bucket set

3. Move existing data into the new bucket set.

but this will take time and is to be avoided if possible.

Hash tables are best when we first insert a lot of data, then use gets without adding a lot of new data.

Other map implementations

A hash table is only one map implementation. Anything which holds key-value pairs, and has put and get operations, can be used as a map implementation.

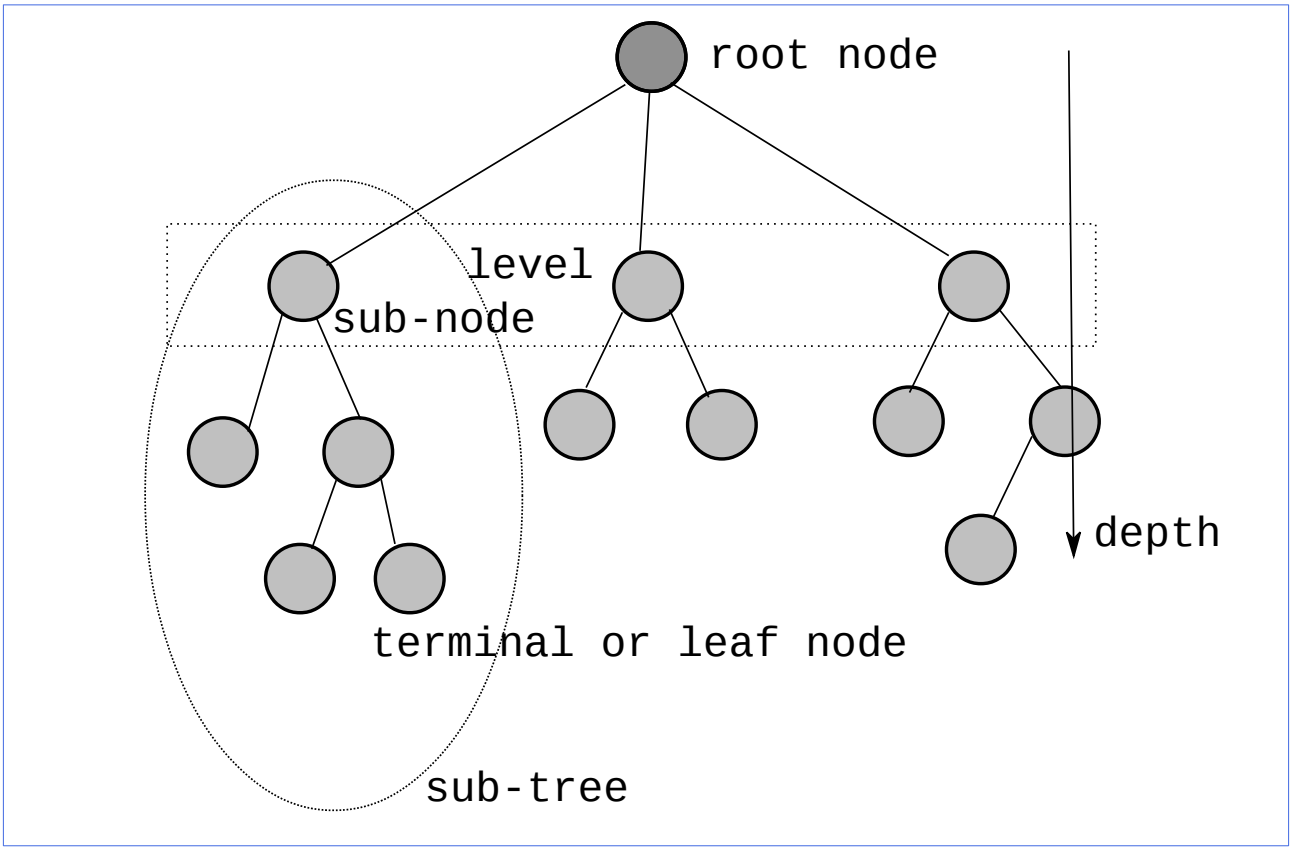
One option is as a binary search tree, as described in the next section.

Trees

In this section

A set of ideas about types of trees

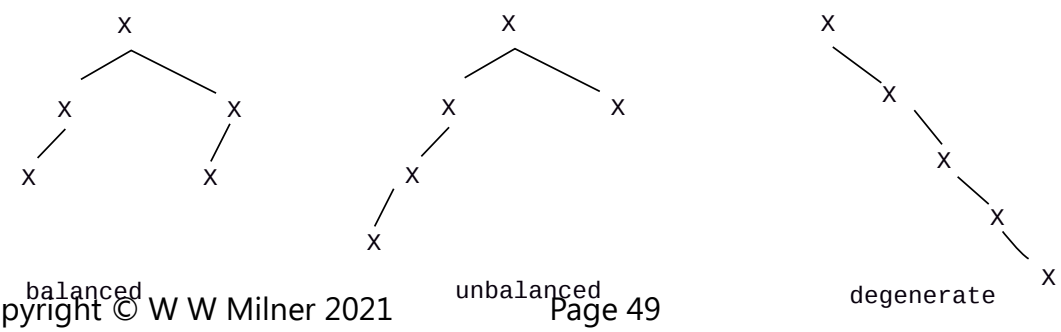
Tree ideas



Trees are upside down - they have the *root* at the top - or at least they are usually drawn that way. There is a set of *nodes*, and each node has 0 or more *sub-nodes* connected to it. As in other data structures, a node has fields for pointers to other nodes, plus data nodes. The data is often a unique key field and other value fields.

A *leaf node* has 0 sub-nodes.

We can think of the set of nodes with a sub-node at the op(as shown) as being a tree in its own right - a *sub-tree*. So trees are recursive, and we often use recursive algorithms.

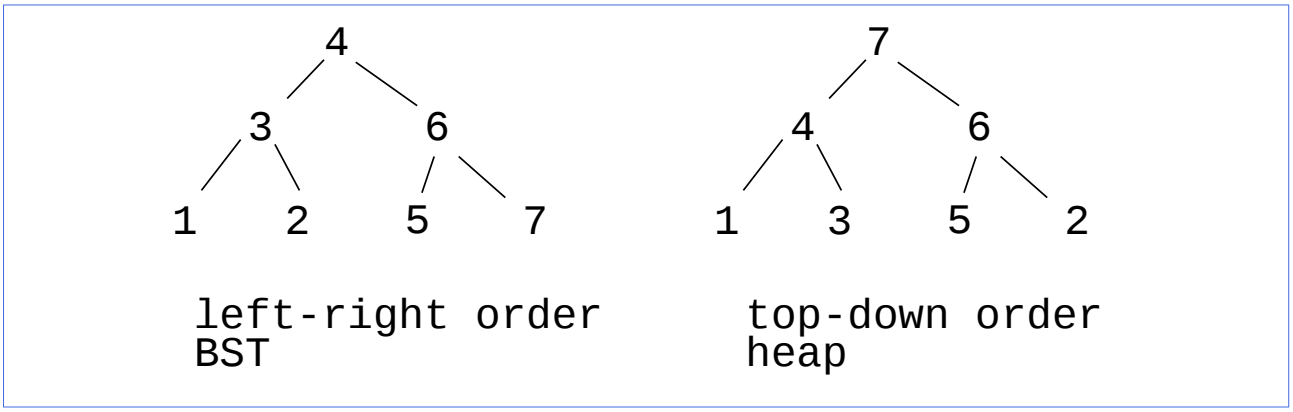


A *balanced tree* has the same number of nodes in all left and right sub-trees. An extreme unbalanced tree has each node having only one sub-node, and this is sometimes called an *degenerate* tree - in effect it is a list.

Some algorithms have speed dependent on the depth of the tree. This is less (faster algorithm) if the tree is balanced. Two common types of balanced tree are *AVL* and *red-black* trees.

A *binary tree* has at most 2 sub-nodes from each node (not a tree represented in binary data - that applies to all of them. The tree drawn above is not binary, since the root node has 3 sub-nodes.

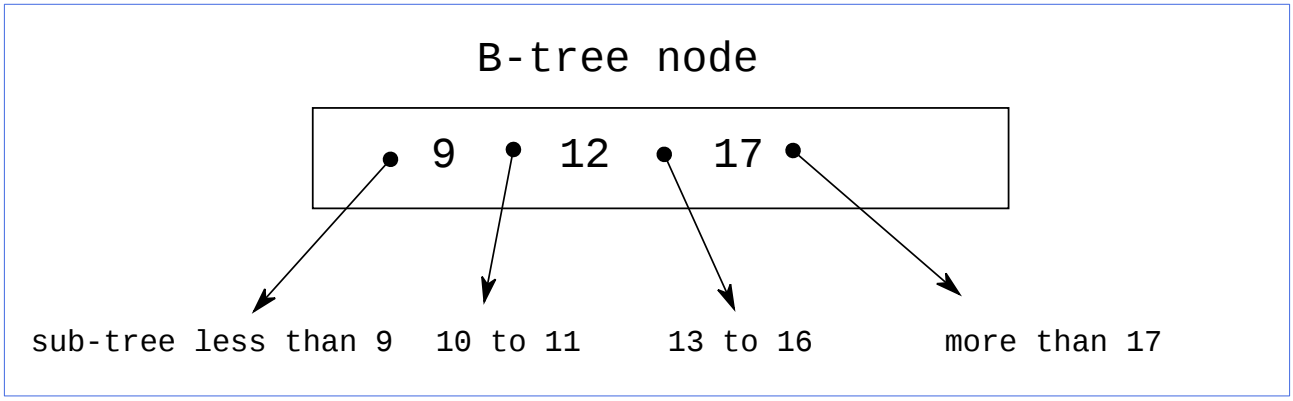
An *ordered tree* has the nodes in some order of key fields.



Different orders are possible. One is left-right. Left sub-nodes are less than node, and right sub-nodes are greater. A left-right ordered binary tree is called a *binary search tree* (BST).

Another order is top-down. Each node is greater than all its sub-nodes, as shown. Note it is still possible for a smaller key (such as 4) to be at a higher level than a larger key (such as 5) if they are in different sub-trees Top-down ordered trees are called *heaps*.

A *B-tree* is a generalization of a BST. In a BST, a node has one key, and divides those below it into 2 sets - those less, and those more. A B-tree node has an ordered list of key values, and has sub-nodes separated by those values. Like this:



Insertion and deletion algorithms make sure the tree remains balanced.

A set of trees which are not connected is called a *forest*.

The defining aspect of a tree is that there are no loops - no cycles. Formally a tree is an *acyclic graph*.

Tree implementations - pointers

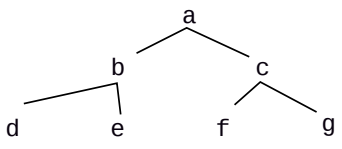
A tree is an ADT. One implementation method is to define a tree node structure, with pointers to other nodes, and a tree, with at least a pointer to the root node. In Python we woudl define these as classes:

```
class TreeNode:
    # A node in a binary tree.

    def __init__(self, keyParam, valueParam):
        self.left=None
        self.right=None
        self.key=keyParam # key and value data fields
        self.value=valueParam

class BST(object):
    #A binary search tree class. Nodes have type TreeNode

    def __init__(self):
        # Construct an new empty tree, no nodes in it
        self.root=None
   ..

myTree=BST() # a new empty tree
```

Array implementation

An alternative implementation is to store the tree in an array. This avoids using pointers.

For example, we have a tree with nodes a,b,c..g as shown. We store these in an array, a at 0, b at 1, c at 2 and so on:



tree

```
a b c d e f g     array contents
0 1 2 3 4 5 6     index
```

array implementation

We can calculate where the sub-nodes of a node are, from its index. For example, the sub-nodes of b, at 1, are at d and e, 3 and 4. In general the left sub-node from index n is a 2n+1, and the right sub-node at 2n+2.

The parent node at index n is at index (n-1) % 2. For example, the parent of g, at 6, is c, at 2.

In Python we can use a list as an array.

```
class Tree:
 def __init__(self, levels):
   self.nodes=list() # a list of nodes, initialy empty
   self.count=0 # how many non-empty nodes we have
   self.levels=levels # how many levels are possible
   self.size=2**levels # how many nodes we could have
   for i in range(0,self.size):
     self.nodes.append(None) # fill list with empty nodes

 def insert(self, value, where):
   # Insert value into tree as a BST
   # This is recursive. The initial call is insert(key,0)
   if where>self.size:
     return # tree is full - do nothing
   if self.nodes[where] is None: # node is empty
     self.nodes[where]=value # store it there
     self.count+=1
     return
   if self.nodes[where]>value:
     self.insert(value, 2*where+1) # go left
   else:
     self.insert(value, 2*where+2) # go right
```

```
 def traverse(self, where):
   if 2*where+1< self.size and not self.nodes[2*where+1] is None:
     self.traverse(2*where+1)
   print(self.nodes[where])
   if 2*where+2< self.size and not self.nodes[2*where+2] is None:
     self.traverse(2*where+2)
```

We can add a method to pretty-print the tree, if it does not have too many levels

```
 def __str__(self):
   # pretty-print tree
   result=''
   i=0 # index of node through list
   width=128 # twice width of first row, for root
   for level in range(0,self.levels):
     width=width>>1 # halve field width
     for offset in range(0, 2**level): ##go through the level
       # get node, stringify, and centre in field width
       element=self.nodes[i].__str__().center(width)
```

```
        result+=element
        i+=1 # next node
      result+='\n'  #next level / output line
    return result
```

So we can say:

```
myTree=Tree(4)
myTree.insert(5,0)
myTree.insert(11,0)
myTree.insert(9,0)
myTree.insert(4,0)
myTree.insert(17,0)
myTree.insert(99,0)

print(myTree)
myTree.traverse(0)
```

and get:

```
                              5
              4                                   11
     None             None              9                17
  None    None    None    None    None    None    None     99

4
5
9
11
17
99
```

Tree traversals - in-order

Traversing a tree (or any data structure) means to 'visit' nodes, in some sequence. What a 'visit' is depends on the situtation. It might be to add up keys, find a maximum, or to serach fro some target, in which case we might stop the traversal when the target is found. In our case, we just print the key and value data fields.

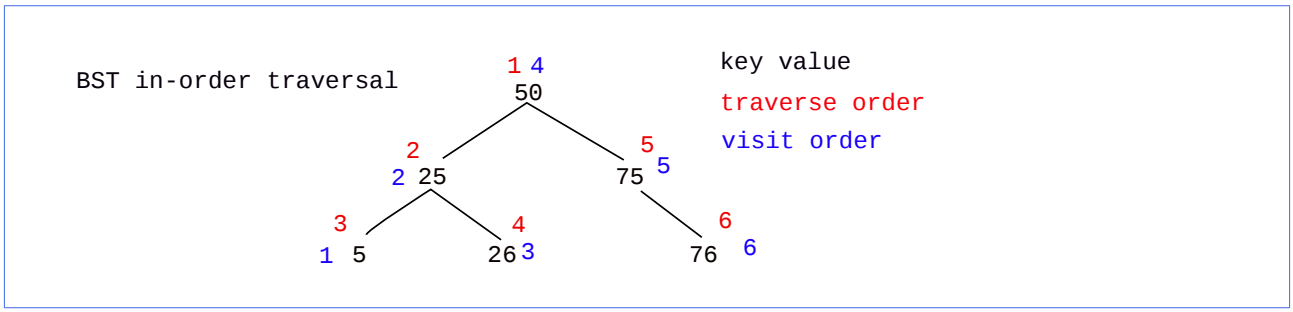
For a binary tree, one traversal algorithm is recursive:

1. if it exists, traverse the left sub-tree

2. visit the node

3. if it exists, traverse the right sub-tree

and we start this by traversing the root. This is called an in-order traversal - left, node, right.



We start *traversing* the root, so 50. This means traversing left, so 25, then right, 75. But traversing 25 means (before the 75) traversing left then right. Going left we reach 5. There is nothing left, so visit the 5, and nothing right. That ends the 5 traversal. Then visit 25 and go left to 26. Nothing left, visit 26, nothing right.That completes the 25 traversal, having visited 5 25 26 in that order.

Next visit 50.

Then we traverse 75 next. Nothing left, so visit 75, then 76.

We do not *visit* a node before traversing left. So we do not visit 50 before traversing 25. And we do not visit that before traversing left, so 5. At 5 there is neither a left or right, so we just visit it, and that call is complete. So 5 is the first node visited.

We add an insert and an in-order traversal to our BST implementation:

```
class TreeNode:
    # A node in a binary tree.
     ..

class BST(object):
    #A binary search tree class. Nodes have type TreeNode

    def __init__(self):
        "Construct an new empty tree"
        self.root=None
    def insert(self, key,value):
        # Insert a key value pair in the tree
        node=TreeNode(key,value) # make a new node with this data
        if self.root==None: # into an empty tree
            self.root=node
            return
        else: # find where to put it
            where=self.root
        while True:
            if where.key < key: # want to go right
                if where.right==None: # found space
                    where.right=node # put it there
                    return # and finish
                else: # go right and continue
                    where=where.right
            else: # want to go left, in same way
```

```
                if where.left==None:
                    where.left=node
                    return
                else:
                    where=where.left
    def inOrderTraverse(self):
        #in-order tree traversal
        self.inOrder(self.root)

    def inOrder(self, where):
        # Visit the left sub-tree, the node, and the right sub-tree
        if where.left!=None:
            self.inOrder(where.left) # recurse left
        print(where.key, where.value) # output data
        if where.right != None: # recurse right
            self.inOrder(where.right)


# End of BST class

myTree=BST()
myTree.insert(50,"Joe")
myTree.insert(25, "Jim")
myTree.insert(75,"Jake")
myTree.insert(5,"Jack")
myTree.insert(26, "Joan")
myTree.insert(76,"June")

myTree.inOrderTraverse()
```

The above code outputs

```
5 Jack
25 Jim
26 Joan
50 Joe
75 Jake
76 June
```

BST search

Suppose we want to search the tree for some target key. We might think:

```
Start at the root
Compare the key with target. If match, found - end
If key too big, go left
else go right
```

'go left' means do this, but starting at the left node. 'go right' means do this starting at the right node.

So this is recursive.

We need to avoid an infinite recursion - it must stop somewhere. Going left or right might be impossible, if there is no left or right sub-tree. This also corresponds to the target not being in the data structure. So we modify the plan:

```
Have we reached a null pointer? If so end - target not present.
Compare the key with target. If match, found - end
If key too big, go left
else go right
```

and start this at the root.

In code, adding to the above BST class:

```
# BST

class TreeNode():
    ..

class BST():
    ..


    def search(self, key, where):
        # Search the tree for this key
        if where==None:
            return "Not present"
        if where.key==key:
            return where.value
        if where.key>key:
            return self.search(key, where.left)
        else:
            return self.search(key, where.right)

# End of BST class
myTree=BST()
myTree.insert(50,"Joe")
myTree.insert(25, "Jim")
myTree.insert(75,"Jake")
myTree.insert(5,"Jack")
myTree.insert(26, "Joan")
myTree.insert(76,"June")

print(myTree.search(26, myTree.root)) # Joan
print(myTree.search(27, myTree.root)) # Not present
```

How fast is this? We start at the root, and decide to go left or right. Either way, we have rejected half the nodes. If there are one million nodes in our tree, we have rejected 500,000 with one comparison.

Each 'if' divides the nodes to check by 2. So the number of 'ifs' is how many times we can halve the node count. This is $\log_2 n$, where n is the number of nodes in the tree.

This is fast. This is why a left-right ordered binary tree is called a binary *search* tree.

Each 'if' partitions the nodes into those which are too big, and those too small. It therefore corresponds to a bisection search on an ordered list.

Pre-order traversal

This is:

```
visit node
traverse left
traverse right
```

For example:

```
class TreeNode(object):
  #A node in a binary tree.

  def __init__(self, keyParam, valueParam):
    ..

class BinaryTree(object):
  def __init__(self):
    "Construct an new empty tree"
    self.root = None

  def insert(self, key,value):
    "Insert a key value pair in the tree"
     .. left-right order insertion



  def traverse(self):
    "An in-order tree traversal"
    self.preOrder(self.root)

  def preOrder(self, where):
    "Visit the node, the left sub-tree,  and the right sub-tree"
    print(where.key, where.value) # output data
    if where.left!=None:
      self.preOrder(where.left) # recurse left
    if where.right != None: # recurse right
      self.preOrder(where.right)


# End of BST class

myTree=BinaryTree()
myTree.insert(50,"Joe")
myTree.insert(25, "Jim")
myTree.insert(75,"Jake")
myTree.insert(5,"Jack")
myTree.insert(26, "Joan")
myTree.insert(76,"June")
myTree.insert(60,"Julie")
```

```
myTree.traverse()
```

and the output is:

```
50 Joe
25 Jim
5 Jack
26 Joan
75 Jake
60 Julie
76 June
```

Pre-order traversals can be used to convert an AST to post-fix, as described later.

Post-order traversal

This is

```
traverse left
traverse right
visit node
```

For example, modifying pre-order:

```
  ..
  def traverse(self):
    "An in-order tree traversal"
    self.postOrder(self.root)

  def postOrder(self, where):
    if where.left!=None:
      self.postOrder(where.left) # recurse left
    if where.right != None: # recurse right
      self.postOrder(where.right)
    print(where.key, where.value) # output data

# End of BST class

myTree=BinaryTree()
myTree.insert(50,"Joe")
myTree.insert(25, "Jim")
myTree.insert(75,"Jake")
myTree.insert(5,"Jack")
myTree.insert(26, "Joan")
myTree.insert(76,"June")
myTree.insert(60,"Julie")

myTree.traverse()
```

outputs

```
5 Jack
26 Joan
25 Jim
60 Julie
76 June
75 Jake
50 Joe
```

Post-order traversal can be used to delete a tree - that is, garbage collect the nodes and return them to free memory. We delete left, then right, then the node:

```
  ..
  def traverse(self):
    if self.root != None:
      self.postOrder(self.root)

  ..

  def deleteTree(self):
    self.delete(self.root)
    self.root=None

  def delete(self,where):
    if where.left!=None:
      self.delete(where.left) # recurse left
      where.left=None
    if where.right != None: # recurse right
      self.delete(where.right)
      where.right=None
    del where


# End of BST class

myTree=BinaryTree()
myTree.insert(50,"Joe")
myTree.insert(25, "Jim")
myTree.insert(75,"Jake")
myTree.insert(5,"Jack")
myTree.insert(26, "Joan")
myTree.insert(76,"June")
myTree.insert(60,"Julie")

myTree.deleteTree()

myTree.traverse()
```

We need to say

```
      where.left=None
```

and

```
      where.right=None
```

because

```
    del where
```

does nothing if there are still references to where - which there would be, from the node above it.

We cannot just say

```
tree.root = None
```

because the left and right sub-trees would still exist.

Breadth-first traversal

Another traversal sequence would be to start at the root, visit nodes next to it, traverse those nodes, and so on. The algorithm uses a queue:

```
start an empty queue
enqueue the root
while the queue is not empty
 dequeue a node
 enqueue its left and right sub-nodes, if they exist
 visit the node
```

Python code would be

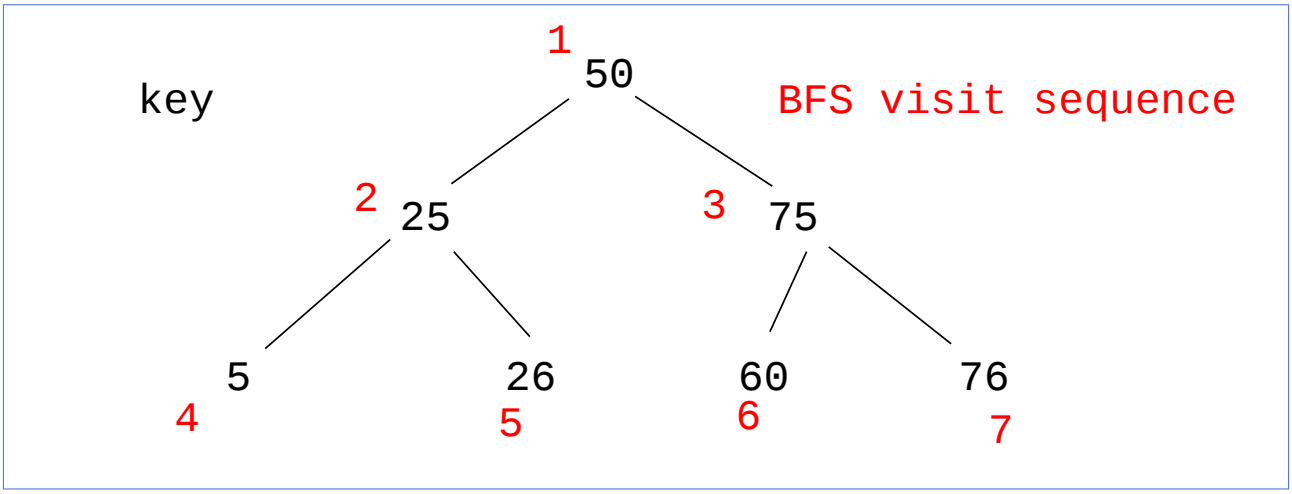
```
class TreeNode(object):
  #A node in a binary tree.

  ..

class BinaryTree(object):
  def __init__(self):
    "Construct an new empty tree"
    self.root = None

  def insert(self, key,value):
    .. left right order insertion


  def traverse(self):
    where=self.root
    q=[]
    q.insert(0, where)
    while len(q)!=0:
      node=q.pop()
      if node.left!=None:
        q.insert(0,node.left)
```

```
      if node.right!=None:
        q.insert(0, node.right)
      print(node.key)

# End of BST class

myTree=BinaryTree()
myTree.insert(50,"Joe")
myTree.insert(25, "Jim")
myTree.insert(75,"Jake")
myTree.insert(5,"Jack")
myTree.insert(26, "Joan")
myTree.insert(76,"June")
myTree.insert(60,"Julie")

myTree.traverse()
```

Output is

```
50
25
75
5
26
60
76
```

The tree and the sequence is:



so this visits the root, all nodes on level 1, all nodes on level 2 and so on. At each level it covers the whole width of the tree, so is called a *breadth-first search*, BFS.

Depth-first traversal

Suppose we use a stack in place of the queue:

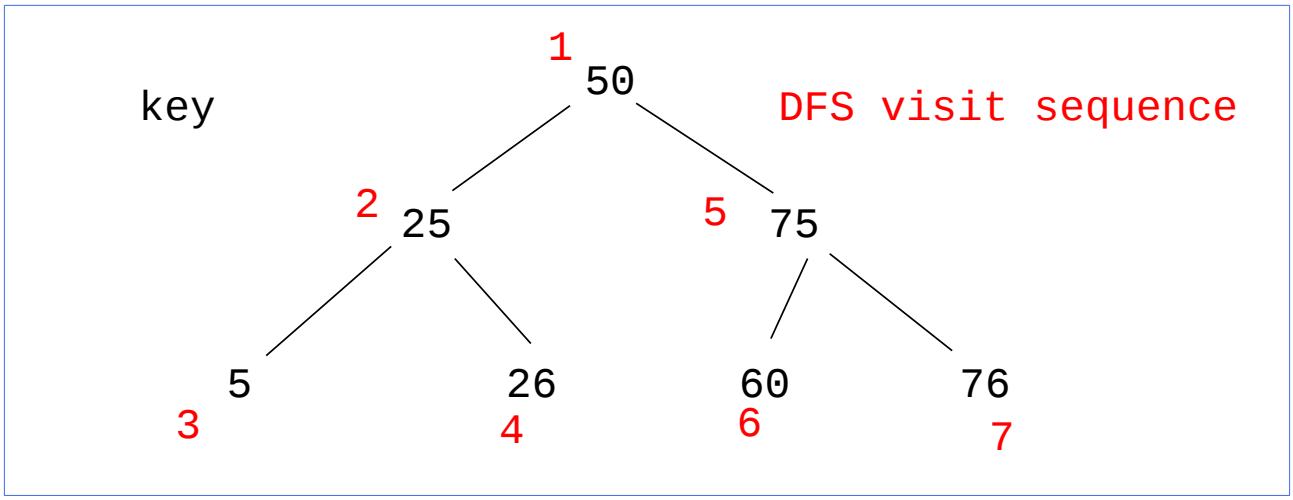
```
  ..
  def traverse(self):
    where=self.root
    stack=[]
    stack.append(where)
    while len(stack)!=0:
      node=stack.pop()
      if node.right!=None:
        stack.append(node.right)
      if node.left!=None:
        stack.append(node.left)

      print(node.key)



# End of BST class

myTree=BinaryTree()
myTree.insert(50,"Joe")
myTree.insert(25, "Jim")
myTree.insert(75,"Jake")
myTree.insert(5,"Jack")
myTree.insert(26, "Joan")
myTree.insert(76,"June")
myTree.insert(60,"Julie")
```



```
myTree.traverse()
```

Output is

```
50
25
5
26
75
60
76
```



This goes from the root as deep as possible.

Tree applications

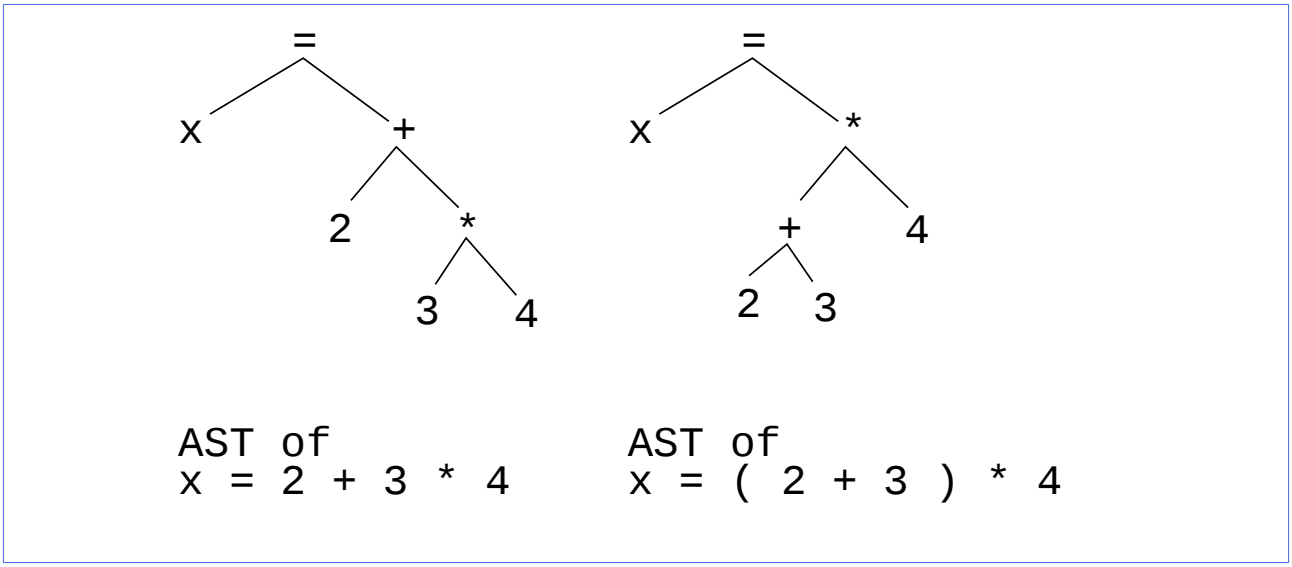
A tree is a basic ADT, and has applications throughout computer science. This section outlines three examples:

- Abstract syntax trees, used in compilers and intepreters
- B-trees, a generalisation of a BST, used for database indexes
- Tries, used for text data compression and predictive text

Abstract syntax trees

This is about code in interpreters and compilers.

An abstract syntax tree (AST) is a tree representing the syntax structure of source code. For example



The structure of the tree matches the syntax of the code. We do not need brackets for (2+3)*4.

Having made an AST, the system can

- Executing it recursively or
- Convert it to post-fix, and execute that

These are described below.

Python has the compile() built-in method, and the ast module, which allow us to use in our code what the compiler usually does. For example:

```
import ast
```

```
myAST=compile('2+3*4','<string>', 'eval', ast.PyCF_ONLY_AST)
print(ast.dump(myAST))
```

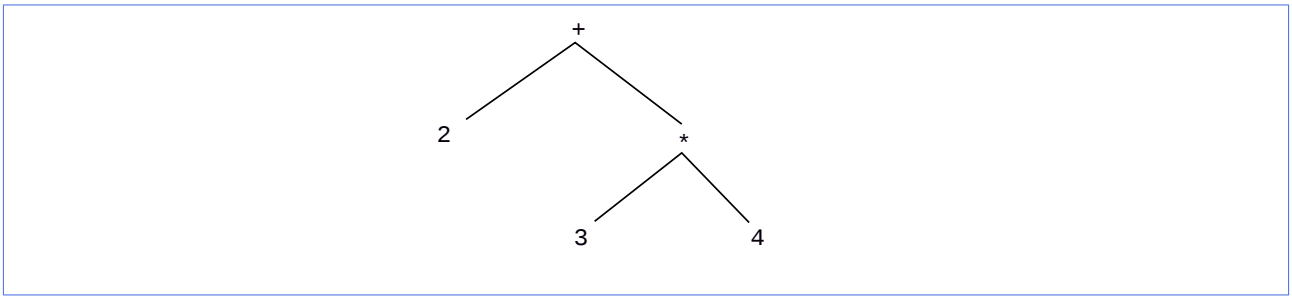
'eval' means to use a single expression - here, '2+3*4'. <string> would be the name of an external source code file, or here, a name to be used by the output object of compile. ast.PyCF_ONLY_AST means Pthon compiler flag to just produce the AST.

The output is:

```
 Expression(body=BinOp(left=Num(n=2), op=Add(),
right=BinOp(left=Num(n=3), op=Mult(), right=Num(n=4))))
```

So this is an expression. The root node is a binary operation, Add. The left node from that is a number, 2, and the right node is another binary operation, Mult. From that, left and right nodes are 3 and 4.

In other words



We can add a method to our Tree class implemented as an array to place values at a given level and position:

```
 def place(self, value, level, across):
   # place value into tree
   # at level (top=0)
   # and across that level (left=0)
   where = 2**level-1+across
   if where<self.size:
     if self.nodes[where]==None:
       self.count+=1
     self.nodes[where]=value
```

and say

```
myTree=Tree(4)
myTree.place('+',0,0)
myTree.place(2,1,0)
myTree.place('*',1,1)
myTree.place(3,2,2)
```

```
myTree.place(4,2,3)

print(myTree)
```

to get

```
                                +
                2                                       *
      None              None                3                   4
  None     None     None     None     None     None     None     None
```

We can evaluate an AST node recursively as follows:

```
if the node is a constant, return its value
else
  evaluate the left node
  evaluate the right node
  apply the operator, and return the value.
```

Then the tree value is the root, evaluated. For example:

```
 def eval(self, where):
   if self.nodes[where] == '+':
     return self.eval(2*where+1) + self.eval(2*where+2)
   if self.nodes[where] == '*':
     return self.eval(2*where+1) * self.eval(2*where+2)
   # else its a constant
   return self.nodes[where]
```

so

```
myTree=Tree(4)
myTree.place('+',0,0)
.. as above

print(myTree.eval(0))
```

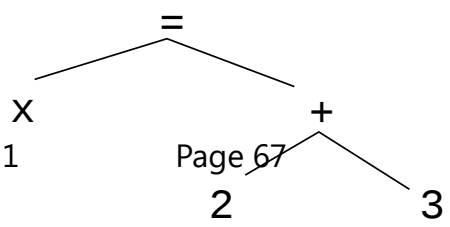
outputs 14.

This only handles + and *, but we can extend it to other operators.

We can extend it to variables as well as constants, but just looking up the variable name in a *symbol table* and replacing it with its value.

We can include =, by treating it as a binary operator. The evaluated value is the right-hand side, but it has the *side-effect* of changing the left-hand side value to that of the right-hand side. For example the statement

x = 2 + 3



turns into this AST. Then that is evaluated as follows

1. Evaluate the RHS as above, get 5
2. Side-effect, make x 5 in the symbol table
3. AST value is 5

Treating = as a binary operator lets us have statements like

y = x = 2+3

which is standard in many languages.

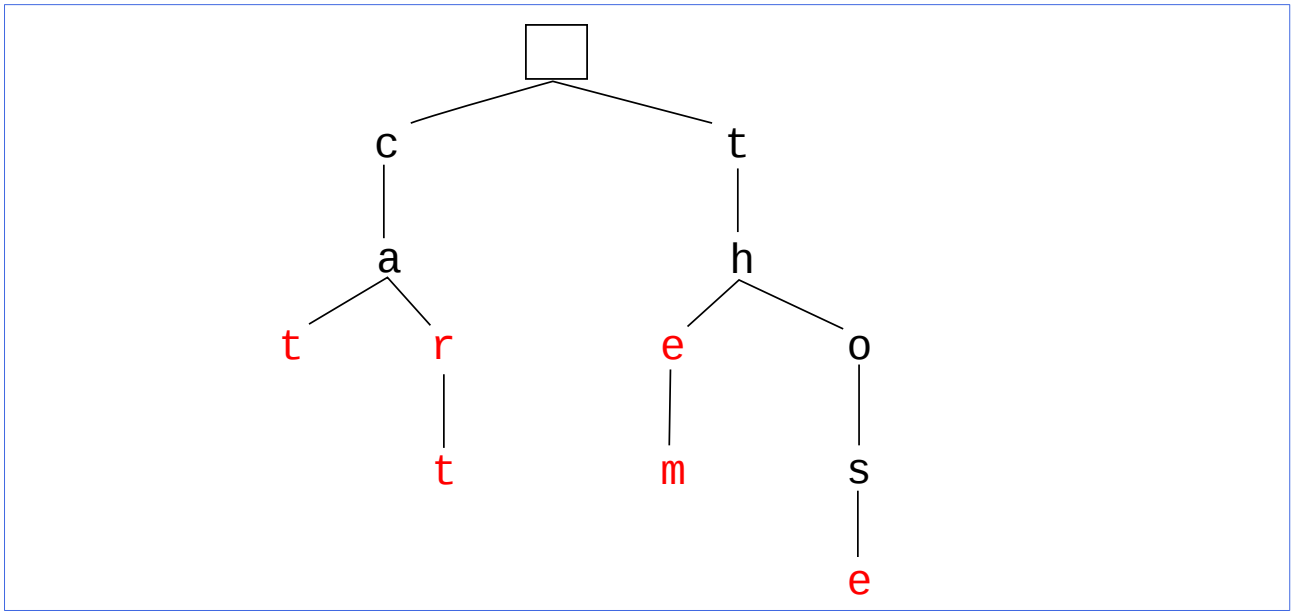
Tries

A trie is a kind of tree, usually used to hold strings of characters. Most trees hold a data item in a node, but a trie holds the data *down a path* from the root.

For example, Suppose we want to hold a very large number of words to use as a dictionary. The word list might include

cart, car, cart, the, them, those

We could store these in a simple list of strings. But that would hold 'c' 3 times, 't' 3 times and so on, wasting space. A trie version would be as shown. Tracing a path down from the root, we get a string.



Some characters are in red because they are possible string ends. For example our trie contains both car and cart.

This is not a binary tree, because a node might have more than 2 sub-nodes.

How to implement this in Python? If we only use characters 'a' to 'z', then from each node, we can only have a maximum of 26 sub-nodes. One way would be to have a list of 26 sub-node pointers from each node. If there is no real sub-node (like no 'b' from 'c' above) the pointer is None. So a trie node could be:

```
class TrieNode:
    def __init__(self, parent):
        self.parent=parent # pointer to parent node
        self.nodes=[] # empty list at first
        self.end=False # not end of a word
        for index in range(0, 26): # 26 null pointers
            self.nodes.append(None)
```

We do not need to actually store which character it is. If this node is the third one in its parent's list, it represents a 'c'.

The trie is:

```
class Trie:
    def __init__(self):
        self.root=TrieNode(None) # root has no parent

    def insert(self, str): # insert a word
        where=self.root # start at the root
        for c in str: # for each character
            index=ord(c)-ord('a') # a is 0, b = 1, ..z is 25
            if where.nodes[index]==None: # no sub-node
                newNode=TrieNode(where) # make one
                where.nodes[index]=newNode # link it
                where=newNode # move there
            else:
                where=where.nodes[index] # just move there
        where.end=True # mark as word end

    def contains(self, str): # does trie contain word?
        where=self.root # start at root
        for c in str:
            index=ord(c)-ord('a')
            if where.nodes[index]==None: # no link
                return False # so no word
            else:
                where=where.nodes[index] # follow link
        return where.end # is it a word end

    def add(self, str): # insert a space-separated word list
        strList=str.split()
        for s in strList:
            self.insert(s)
```

which we can use as:

```
myTrie=Trie()
myTrie.add("cat the car them cart those")
print(myTrie.contains("the")) # true
print(myTrie.contains("car")) # true
print(myTrie.contains("cart")) # true
print(myTrie.contains("teh")) # false
```

Treesort

We simply insert the data into a BST, then an inorder travesal will retreive it inkey order:

```
class TreeNode(object):

    def __init__(self, keyParam, valueParam):
        self.left = None
        self.right = None
        self.key = keyParam
        self.value = valueParam


class BST(object):

    def __init__(self):
        self.root = None

    def insert(self, key, value):
        ..

    def inOrderTraverse(self):
        "An in-order tree traversal"
        self.inOrder(self.root)

    def inOrder(self, where):
        "Visit the left sub-tree, the node, and the right sub-tree"
        if where.left != None:
            self.inOrder(where.left)  # recurse left
        print(where.key, where.value)  # output data
        if where.right != None:  # recurse right
            self.inOrder(where.right)

    ..

# End of BST class

myTree = BST()
for key in [5, 1, 9, 3, 6, 8, 21, 17, 23, 41, 16]:
    myTree.insert(key, None)

myTree.inOrder(myTree.root)
```

Output is:

```
1 None
3 None
```

```
5 None
6 None
8 None
9 None
16 None
17 None
21 None
23 None
41 None
```

For random initial data the time complexity is O(n log n), the time for a BST traversal.

If the initial data is already sorted, the BST is degenerate, and the time is $O(n^2)$. This is like a quicksort on ordered data. We can avoid it by using an AVL or red-black tree so it is balanced.

However it is not in-place, and the space complexity is O(n). We can do it as an in-place sort if we treat the array as a tree, as described above.

Tree Map

We can use a BST as a map of key value pairs:

```
class TreeNode(object):
    ..


class BST(object):
    "A binary search tree class. Nodes have type TreeNode"

    def __init__(self):
        "Construct a new empty tree"
        self.root = None

    def put(self, key, value):
        "Insert a key value pair in the tree"
        node = TreeNode(key, value)  # make a new node with this data
        if self.root == None:
            self.root = node
            return
        else:  # find where to put it
            where = self.root
        while True:
            if where.key < key:  # want to go right
                if where.right == None:  # found space
                    where.right = node  # put it there
                    return  # and finish
                else:  # go right and continue
                    where = where.right
            else:  # want to go left, in same way
```

```
                if where.left == None:
                    where.left = node
                    return
                else:
                    where = where.left

..

    def search(self, key, where):
        # Search the tree for this key
        if where == None:
            return "Not present"
        if where.key == key:
            return where.value
        if where.key > key:
            return self.search(key, where.left)
        else:
            return self.search(key, where.right)

    def get(self, key):
        return self.search(key, self.root)

# End of BST class



myTree = BST()
myTree.put(50, "Joe")
myTree.put(25, "Jim")
myTree.put(75, "Jake")
myTree.put(5, "Jack")
myTree.put(26, "Joan")
myTree.put(76, "June")

print(myTree.get(75))   # Jake
print(myTree.get(6))    # Not present
```

Graphs

As described above, the idea of a graph is like a road map. We have nodes (sometimes called vertexes) which are like towns, linked by edges which are like roads. Edges might be directed (one-way roads) or not. Edges might be weighted (like the road length) or not.

Graphs could be used to represent:

- Road maps, metro maps, railmaps, air flight routes
- Devices on a LAN or on the Internet
- Software packages and their dependencies

These are not the same as graphs of functions using x,y Cartesian co-ordinates.

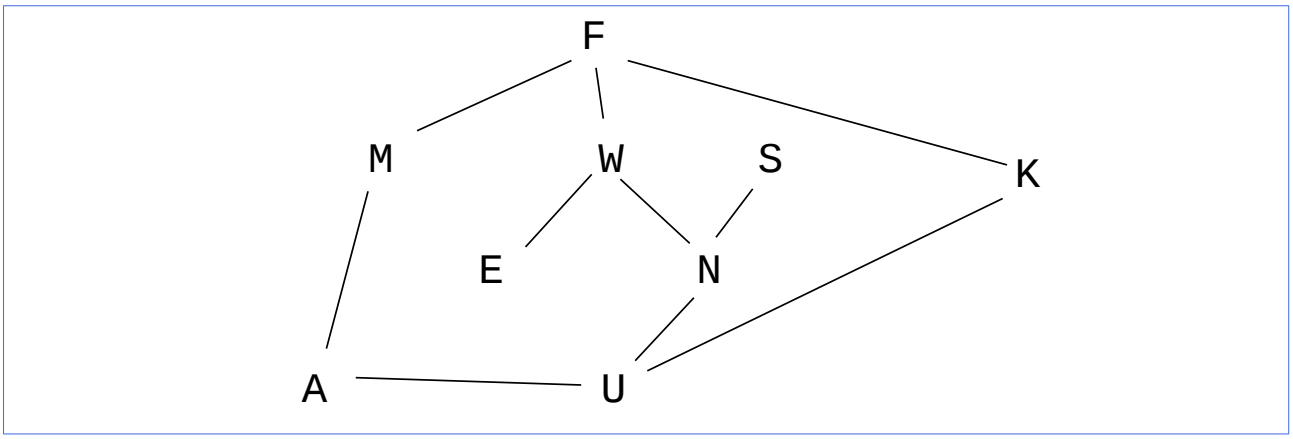
Trees are one type of graph (graphs with no loops of cycles - acyclic graphs).

A graph is an ADT, and can be implemented in different ways. The two common ways are

- A matrix. This is a table, with rows and columns for nodes. The entry in the cell (i,j) is 1 if nodes i and j are linked directly by an edge, and 0 if they are not. That is for an unweighted graph. For a weighted graph, (i,j) is the weight of the edge from i to j.
- An adjacency list. There is a list for every node i. The list contains all nodes linked directly with i.

Adjacency list

This is an example of an undirected unweighted graph:



and this is an adjacency list implementation in Python:

```
rowF = ['M','W', 'K']  # list of nodes F is linked to
rowM = ['F', 'A']      # nodes M is linked to
rowW = ['F', 'E', 'N'] # and so on
rowK = ['F', 'U']
```

```
rowE =['W']
rowN =['W', 'U', 'S']
rowS =['N']
rowA =['M', 'U']
rowU=['A', 'N', 'K']
adjList = { # this is a dictionary - to make access fast
  'F': rowF,
  'M': rowM,
  'W': rowW,
  'K': rowK,
  'E': rowE,
  'N': rowN,
  'S': rowS,
  'A': rowA,
  'U': rowU
}

# show nodes linked to N:
whichNode='N'
list=adjList[whichNode]
print(list) # ['W', 'U', 'S']
```

Breadth First Traversal

To be completed..

