CSC148 winter 2014

constant-time access,
review
week 12

Danny Heap
heap@cs.toronto.edu
BA4270 (behind elevators)
http://www.cdf.toronto.edu/~heap/148/W14/
416-978-5899

April 2, 2014

Outline

constant time

ADTs

testing

recursion

exceptions

python idiom

big-oh

constant-time insert, find

trees maintain data in order, and (with certain constraints) with $\lg n$ performance for inserting and finding elements, but Python dictionaries can insert, find in constant time — how?

a Python list has constant access, if the index is known, since the references to elements are stored in consecutive memory locations:

down-side of **list**

Python lists are accessed by index (an integer) — what about other, more meaningful keys?

a string, or a tuple, can be converted into an integer, using **hash** — not the same as **id**!

storage and collisions

once you have a way to convert objects to integers, you can then convert those integers into indices in the appropriate range — think %

how big a list do you need to hold your data? — the answer is a bit surprising

suppose you had 365 "slots" — how soon would you expect a collision?

dictionaries explained

a python dictionary:

- key $\rightarrow$ integer
- integer $\rightarrow$ list position
- collision $\rightarrow$ linked list (chain) or probe the list

Abstract Data Type

- **abstract**: user doesn't want to know how it works (implementation), but how to work it (public interface)
- **data**: record information
- **type**: structure encapsulates a concept that is common to an entire set (type) of instances

use Python classes...

- classes implement your ADTs, to add to built-in ones like **str** and **Turtle**.
- classes encapsulate data, and methods to operate on it, that express an idea — they are plans for an ADT.
- developers can use (instantiate) existing classes without knowing the details of how they work (implementation) — an instance of a class is an actual object, a member of the ADT the class defines.
- developers can recycle existing class definitions, extending and modifying them through inheritance.

ADT example: stacks...

computer stacks have the same strengths and weaknesses as the stacks they imitate in the physical world: sequences of items with easy storage and retrieval from the top; retrieval from other positions in the stack, not so much.

Data: sequence of items

Operations: **push(item)**, **pop()**, **is_empty**

inheritance

Classes allow you to recycle existing code by

- Composition: create a new class that includes an instance of an old class, for example there's an instance of **list** in one of our stack implementations.
- Inheritance: Modify or extend an old class by creating a subclass that inherits some features, extends or modifies others.
- In poly.py and special_poly.py I use all approaches: composition to include a **Turtle** instance, extending the **__init__** method of **Polygon** in **SpecialPolygon**, replacement of **Polygon**'s draw method in **SpecialPolygon**, and inheritance of **get_param**. Use the debugger for intuition.

testing

Use your `docstring` for testing as you develop, but use unit testing to make sure that your particular implementation remains consistent with your ADT's interface. Be sure to:

- import the module `unittest`
- subclass `unittest.Testcase` for your tests, and begin each method that carries out a test with the string `test`
- compose tests **before** and **during** implementation

using exceptions

You can use Exception, and its subclasses in several ways

- If execution stops with an Exception, read the message and reason about the cause.
- You can define a subclass of Exception (or one of Exception's subclasses) with a more meaningful name, and write code to **raise** your exception when you detect a problem.
- You can use **try**...**except** pairs to **try** to execute code, but then execute alternate code under **except**ional circumstances.

going with the (pep) tide

Python is more flexible than the community you are coding in. Try to figure out what the python way is

- don't re-invent the wheel (except for academic exercises), e.g. **sum**, **max**, **set**, **any**, **all**
- use comprehensions when you mean to produce a new list (tuple, dictionary, set, ...)
- use ternary `if` when you want an expression that evalutes in different ways, depending on a condition

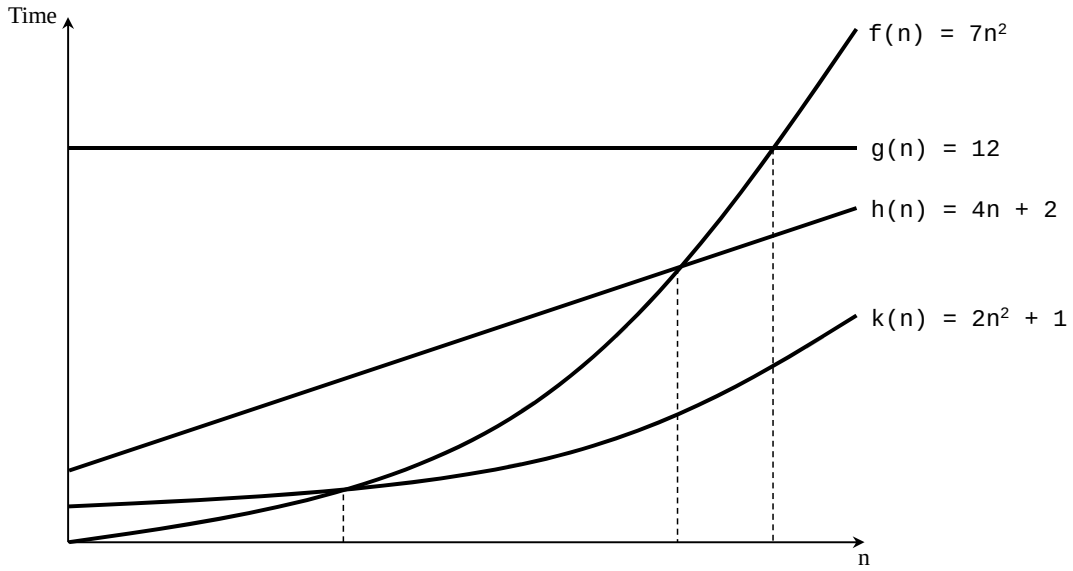
running time analysis

algorithm's behaviour over large input (size **n**) is common way to compare performance — how does performance vary as **n** changes?

constant:	$c \in \mathbb{R}^+$ (some positive number)
logarithmic:	$c \log n$
linear:	cn (probably not the same c)
quadratic:	cn^2
cubic:	cn^3
exponential:	$c2^n$
horrible:	cn^n or $cn!$

Running Time Analysis

- An algorithm is O(g(n)) if any reasonable implementation of the algorithm on any reasonable computer would require O(g(n)) time to solve a large problem of size n.



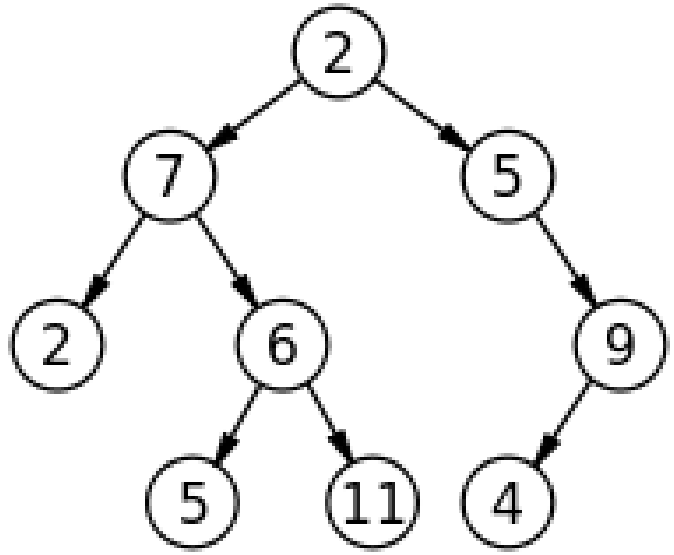
problems that resemble their parts

Suppose n is a non-negative integer stored in your computer, and you'd like to represent n as a decimal (base 10) string. After all, Python has to do something like this with integers that are stored in hardware in binary.

The problem can be broken down into how to get the units digit (use the % operator), and how to get everything but the units digit (use the / operator). Now concatenate the string representing the units digit to the string representing everything but the units digit.

The second part of the solution sounds like a smaller instance of the original problem. Use recursion — call the function itself within its own definition.

recursion, natural and otherwise



terminology

- set of **nodes** (possibly with values or labels), with directed **edges** between some pairs of nodes
- One node is distinguished as **root**
- Each non-root node has exactly one parent.
- A **path** is a sequence of nodes $n_1, n_2, \ldots, n_k$, where there is an edge from n_i to n_{i+1}. The **length** of a path is the number of edges in it
- There is a unique path from the root to each node. In the case of the root itself this is just n_1, if the root is node n_1.
- There are no **cycles** — no paths that form loops.

more terminology

- **leaf**: node with no children
- **internal node**: node with one or more children
- **subtree**: tree formed by any tree node together with its descendants and the edges leading to them.
- **height**: Maximum path length in a tree. A node also defines a height, which is the maximum path length of the tree rooted at that node
- **arity, branching factor**: maximum number of children for any node.

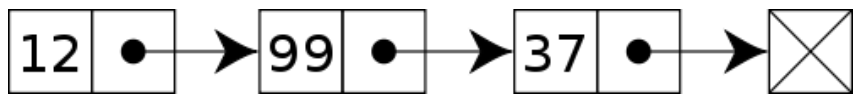
linear trees?

Trees of arity (branching factor) 1 can be thought of as a sequence of lists. Every node has no more than one child, and every node (other than the lone leaf) has no less than one child.

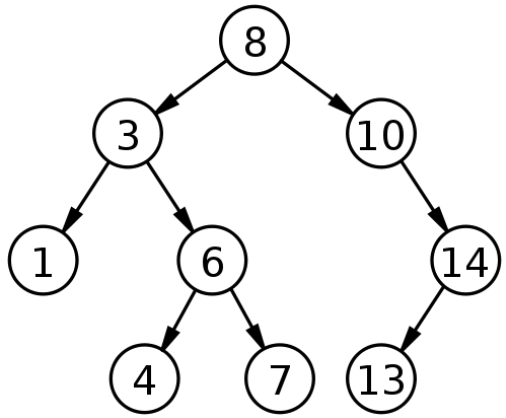
linked lists, conceptually

- **data**: Sequence of nodes, each with a **head** (value) and a reference to **rest** (its successors).

- **operations**: **prepend(value)**, **__contains__(value)**

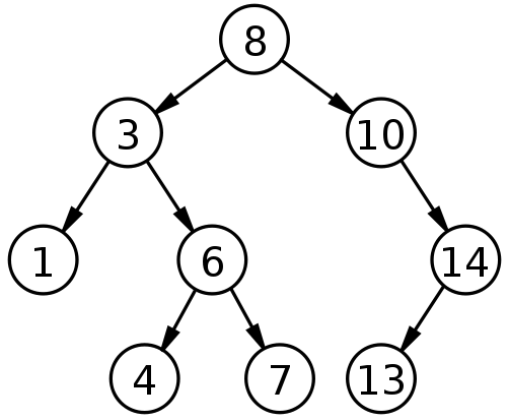


binary search tree



- A binary tree
- left subtree of each node contains elements with values less than that node
- right subtree of each node contains elements with values more than that node

tree surgery



Standard operations such as **find(data)**, **insert(data)**, **height()**, and **delete(data)** can be made efficient in a BST, due to the order property. Since each of **insert** and **delete** may alter the root of the tree, it makes sense to have them return a reference to the root.

quick sort performance

- how many times do we choose the pivot?
- how many steps each time we choose a pivot?

merge sort

idea: divide the list in half, (merge) sort the halves, then merge the sorted results

merge sort performance

- how many times do we split the list in half?
- how many steps each time we split?

scaling:

How well do these various sorts perform as the size of the problem (list length) increases? Time and compare.

TAs...

Tuesday, 9–11: Madina, Larry (BA3175) Brian (BA3185) Sagun (BA3195)

Tuesday 11–1: Abayomi (BA3175) Tong (BA3185) Sunny (BA3195)

Tuesday 1–3: Shobhit (BA3175) Abdi (BA3185) Shems (BA3195)

Wednesday 11-1: Sam (BA3175) Hazem (BA3185) Amirali (BA3195)

Wednesday 1–3: Nahla (BA3175) Jake (BA3185) Carter (BA3195)

Wednesday 3–5: Olessia (BA3175) Madina, Lin (BA3185) Cheng (BA3195)

Tuesday 7–9: Younan (BA3175) Yanshuai (BA3185)

Wednesday 7–9: Michalis (BA3175)