CSC148 winter 2014

constant-time access,
review
week 12

Danny Heap / Dustin Wehr
heap@cs.toronto.edu / dustin.wehr@utoronto.ca
BA4270 / SF4306D
http://www.cdf.toronto.edu/~heap/148/F13/

April 2, 2014

Outline

- constant time
- ADTs
- testing
- recursion
- exceptions
- big-oh

constant-time insert and retrieval

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

A Python list has constant-time access for looking up the object at a given index, since:

- the references to elements of the list are stored in consecutive memory locations.
- your computer's hardware, namely its RAM (random-access memory), allows constant-time retrieval of the data stored at a given memory address.

Address of list element i is roughly (address of list) $+ i$.

down-side of **list**

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

a string, or a tuple, can be converted into an integer, using a **hash function**. python has a built-in function called **hash** that converts immutable objects to 32 or 64-bit integers — not the same as **id**!

ideal situation

where you don't have to do anything fancy like Python does with the built-in `dict` datatype

- you know the set of keys K that you'll want to use in your dictionary
- it's possible –*and you have time to*– write a fast (linear time in the length of the key) *injective* "**hash function**" f that maps K to integers in some range $\{0, \ldots, N\}$, and you can afford to allocate $\mathcal{O}(N)$ bits of memory for this dictionary ($64N$ or $32N$ depending on your computer).
- then you have a dictionary just by composing f and a `list` L.
  $L[f(k)] = v$ to store the mapping $k \mapsto v$
  $L[f(k)]$ to retrieve the value that k maps to.

usual situation...

If we want to use regex strings as keys, are we in the ideal situation?

Probably not.

Attempt: Since regexs are strings from an alphabet of size 8, there are $\leq 8^n$ valid regexs of length 8. Can very quickly map a length-n regex to an integer in $\{0, \ldots, 8^n\}$:

```
replace character with the digit below it
( ) * | . 0 1 2
0 1 2 3 4 5 6 7
```

But 8^n is way too large; even for $n = 11$ the list wouldn't fit in your computer's memory.

Hope for solution: Vast majority of those 8^n strings aren't even valid length-n regexs, and we'll probably only ever be using $\leq$ 1 million regex string keys at any one time.

usual situation...

If we want to use regex strings as keys, are we in the ideal situation?

Probably not.

Attempt: Since regexs are strings from an alphabet of size 8, there are $\leq 8^n$ valid regexs of length 8. Can very quickly map a length-n regex to an integer in $\{0, \ldots, 8^n\}$:

```
replace character with the digit below it
( ) * | . 0 1 2
0 1 2 3 4 5 6 7
```

But 8^n is way too large; even for $n = 11$ the list wouldn't fit in your computer's memory.

Hope for solution: Vast majority of those 8^n strings aren't even valid length-n regexs, and we'll probably only ever be using $\leq$ 1 million regex string keys at any one time.

usual situation...

If we want to use regex strings as keys, are we in the ideal situation?
Probably not.

Attempt: Since regexs are strings from an alphabet of size 8, there are $\leq 8^n$ valid regexs of length 8. Can very quickly map a length-n regex to an integer in $\{0, \ldots, 8^n\}$:

```
replace character with the digit below it
( ) * | . 0 1 2
0 1 2 3 4 5 6 7
```

But 8^n is way too large; even for $n = 11$ the list wouldn't fit in your computer's memory.

Hope for solution: Vast majority of those 8^n strings aren't even valid length-n regexs, and we'll probably only ever be using $\leq$ 1 million regex string keys at any one time.

usual situation...

If we want to use regex strings as keys, are we in the ideal situation?
Probably not.

Attempt: Since regexs are strings from an alphabet of size 8, there are $\leq 8^n$ valid regexs of length 8. Can very quickly map a length-n regex to an integer in $\{0, \ldots, 8^n\}$:

```
replace character with the digit below it
( ) * | . 0 1 2
0 1 2 3 4 5 6 7
```

But 8^n is way too large; even for $n = 11$ the list wouldn't fit in your computer's memory.

Hope for solution: Vast majority of those 8^n strings aren't even valid length-n regexs, and we'll probably only ever be using $\leq$ 1 million regex string keys at any one time.

usual situation...

If we want to use regex strings as keys, are we in the ideal situation?
Probably not.

Attempt: Since regexs are strings from an alphabet of size 8, there are $\leq 8^n$ valid regexs of length 8. Can very quickly map a length-n regex to an integer in $\{0, \ldots, 8^n\}$:

```
replace character with the digit below it
( ) * | . 0 1 2
0 1 2 3 4 5 6 7
```

But 8^n is way too large; even for $n = 11$ the list wouldn't fit in your computer's memory.

Hope for solution: Vast majority of those 8^n strings aren't even valid length-n regexs, and we'll probably only ever be using $\leq$ 1 million regex string keys at any one time.

storage and collisions

Solution: drop the **injective** requirement for the hash function.

Say we're willing to use $\mathcal{O}(N)$ memory for our dictionary.

Let f be the function from regex strings to integers from the previous slide.

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

Try to store regex string s at list index $f(x)//N$.

If the values of $f(x)$ are well spread out, then the values of the hash function $f(x)//N$ will be too, which means we will minimize the number of **collisions** - a crucial property for hash functions.

storage and collisions

Solution: drop the **injective** requirement for the hash function.

Say we're willing to use $\mathcal{O}(N)$ memory for our dictionary.

Let f be the function from regex strings to integers from the previous slide.

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

Try to store regex string s at list index $f(x)//N$.

If the values of $f(x)$ are well spread out, then the values of the hash function $f(x)//N$ will be too, which means we will minimize the number of **collisions** - a crucial property for hash functions.

storage and collisions

Solution: drop the **injective** requirement for the hash function.

Say we're willing to use $\mathcal{O}(N)$ memory for our dictionary.

Let f be the function from regex strings to integers from the previous slide.

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

Try to store regex string s at list index $f(x)//N$.

If the values of $f(x)$ are well spread out, then the values of the hash function $f(x)//N$ will be too, which means we will minimize the number of **collisions** - a crucial property for hash functions.

storage and collisions

Solution: drop the **injective** requirement for the hash function.

Say we're willing to use $\mathcal{O}(N)$ memory for our dictionary.

Let f be the function from regex strings to integers from the previous slide.

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

Try to store regex string s at list index $f(x)//N$.

If the values of $f(x)$ are well spread out, then the values of the hash function $f(x)//N$ will be too, which means we will minimize the number of **collisions** - a crucial property for hash functions.

storage and collisions

Solution: drop the **injective** requirement for the hash function.

Say we're willing to use $\mathcal{O}(N)$ memory for our dictionary.

Let f be the function from regex strings to integers from the previous slide.

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

Try to store regex string s at list index $f(x)//N$.

If the values of $f(x)$ are well spread out, then the values of the hash function $f(x)//N$ will be too, which means we will minimize the number of **collisions** - a crucial property for hash functions.

storage and collisions

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

Suppose you had 366 "slots" — how soon would you expect a collision?

- For sure once we have 367 keys.
- 50% chance once have 23 keys.
- 99.9% chance once have 70 keys.

dictionaries explained

a python dictionary:

- key $\rightarrow$ integer
- integer $\rightarrow$ list position
- collision $\rightarrow$ linked list (chain), or re-hash the key, or resize the dictionary

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** and **area**. 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

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.

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.

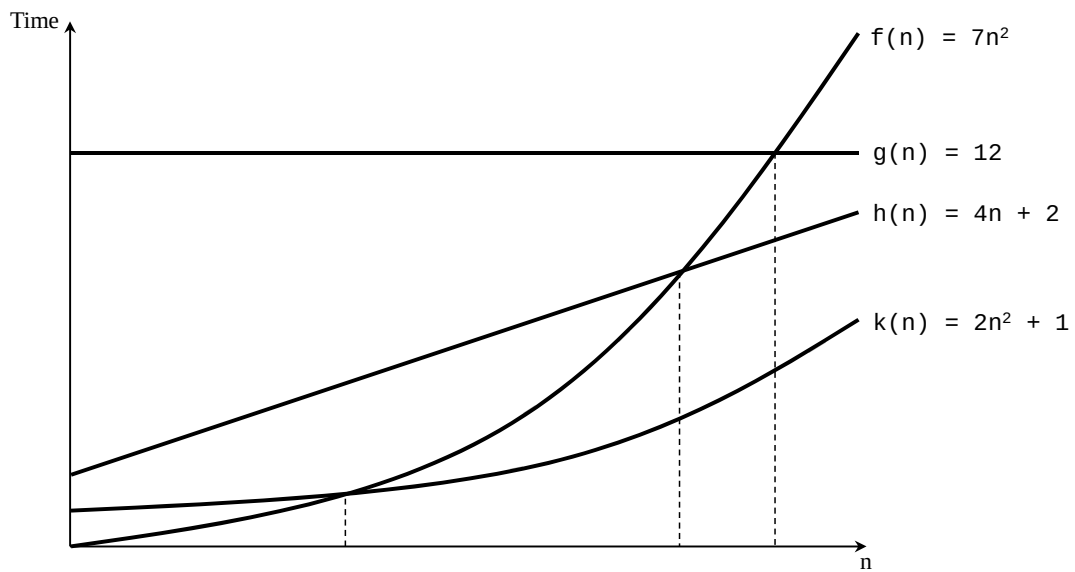
running time analysis

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.



terminology

- set of **nodes** (possibly with values or labels), with directed **edges** between some pairs of nodes
- One node is distinguished as **root** – the only node in the tree with no parent.
- Each non-root node has exactly one parent. u is a **child of** v if and only if v is u's 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** or **branching factor**: maximum number of children for any node.

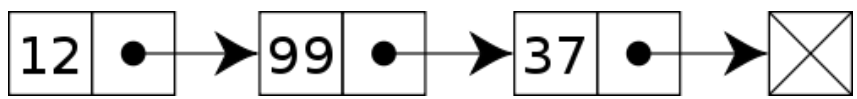
linear trees?

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.

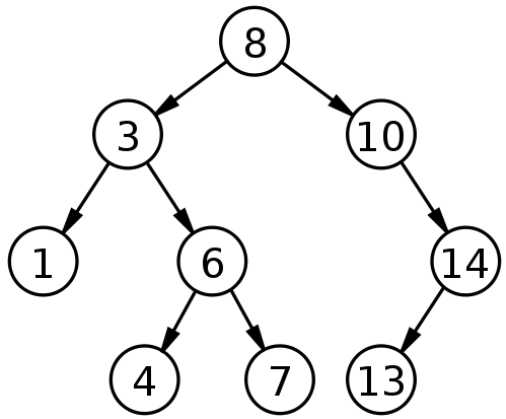
linked lists, conceptually

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

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

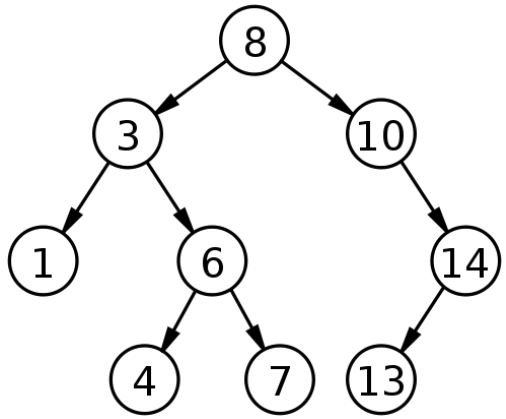


binary search tree



- 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.

merge sort

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

merge sort runtime

Suppose we implement merge sort, and in examining the program we see that

- At most 6 operations are used when $n = 1$.
- When $n \geq 2$, at most $6n$ operations are used for just the splitting and merging parts of the program.

Let $T(n)$ be the number of operations used on lists of length n.

Then $T(n) \leq 2\,T(n/2) + 6n$ from the previous observations.

Claim: Follows that $T(n) \leq 6n \log n + 6n$, so merge sort is $\mathcal{O}(n \log n)$.
(you're not responsible for being able to prove this)

merge sort runtime

Let $T(n) \leq 2T(n/2) + 6n$ be the total number of operations used on lists of length n.

$$
\begin{aligned}
&T(n) \leq 2T(n/2) + 6n \\
&= 2[6(n/2)\log(n/2) + 6(n/2)] + 6n \\
&= 6n\log(n/2) + 6n + 6n \\
&= 6n[\log n - \log 2] + 12n \\
&= 6n[\log n - 1] + 12n \\
&= 6n\log n - 6n + 12n
\end{aligned}
$$

So $T(n) \leq 6n\log n + 6n$.

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)