ALGORITHM EFFICIENCY

LECTURE 12-1

JIM FIX, REED COLLEGE CSCI 121

DEPARTMENT INFO

- **Talk today!!**
  -

LECTURE 12-

DEPARTM

- Talk toda
  -

REED COLLEGE
COMPUTER SCIENCE DEPARTMENT
JOB TALK

JENNA WISE

Carnegie Mellon University

Gradual Verification: Assuring Program Correctness Incrementally

While software is becoming more ubiquitous in our everyday lives, so are unintended bugs. In response, static verification techniques were introduced to prove or disprove the absence of bugs in code. Unfortunately, current techniques burden users by requiring them to write inductively complete specifications involving many extraneous details. To overcome this limitation, I introduce the idea of gradual verification, which handles complete, partial, or missing specifications by soundly combining static and dynamic checking. As a result, gradual verification allows users to specify and verify only the properties and components of their system that they care about and increase the scope of verification gradually—which is poorly supported by existing tools.

In this presentation, I outline the formal foundations of gradual verification for recursive heap data structures (like lists, trees, and graphs), and the design of a gradual verifier derived from my formal work, called Gradual C0. Gradual C0 is implemented on top of the Viper static verifier and supports the C0 programming language—which is a safer, smaller subset of C taught at CMU. Additionally, I present the results of quantitatively evaluating Gradual C0's static and dynamic performance characteristics for thousands of partial specifications. Gradual C0 on average decreases run-time overhead by 50-90% compared to dynamic verification alone and sources of overhead correspond to predictions in prior work. Qualitatively, Gradual C0 exhibits earlier error detection for incorrect specifications than static verification. I end with my planned new lines of work in gradual verification and its application to other formal methods.

MONDAY, NOVEMBER 21, 2022
4:40PM
ELIOT 314

BINARY SEARCH TREES

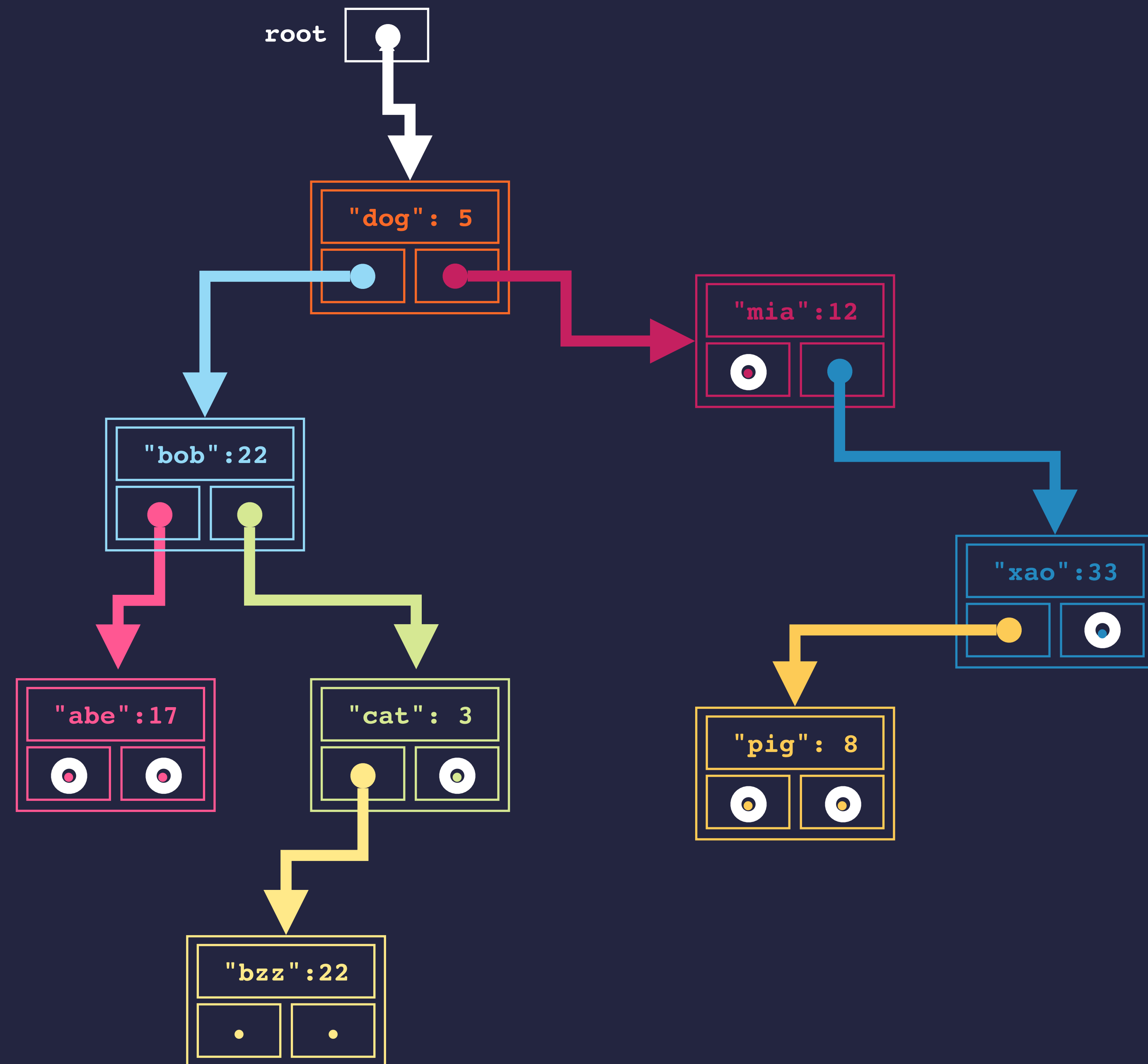
- Binary search trees are a way of keeping track of a ***sorted*** collection.
- Here, we are using them as an *ordered dictionary*.
- For our dictionaries, there is at most one entry per key.
- The link structure sorts the entries; maintains a sorted order.
  - The keys are usually organized alphabetically when strings.
  - The keys are usually sorted smaller/larger if numbers.
- (Generally, in binary search trees, keys might appear more than once; have multiple entries.)
- (Generally, in binary search trees, the nodes might only contain keys without associated values.)

A BST CLASS

- **Operations**:
  - Searching for an entry by key.
  - Adding or updating an entry, ordered according to key, storing the value.
  - Removing an entry.
  - Visiting all the entries in sorted order.
- The first three operations rely on a *search*.
  - This works from the root, moving left or right.

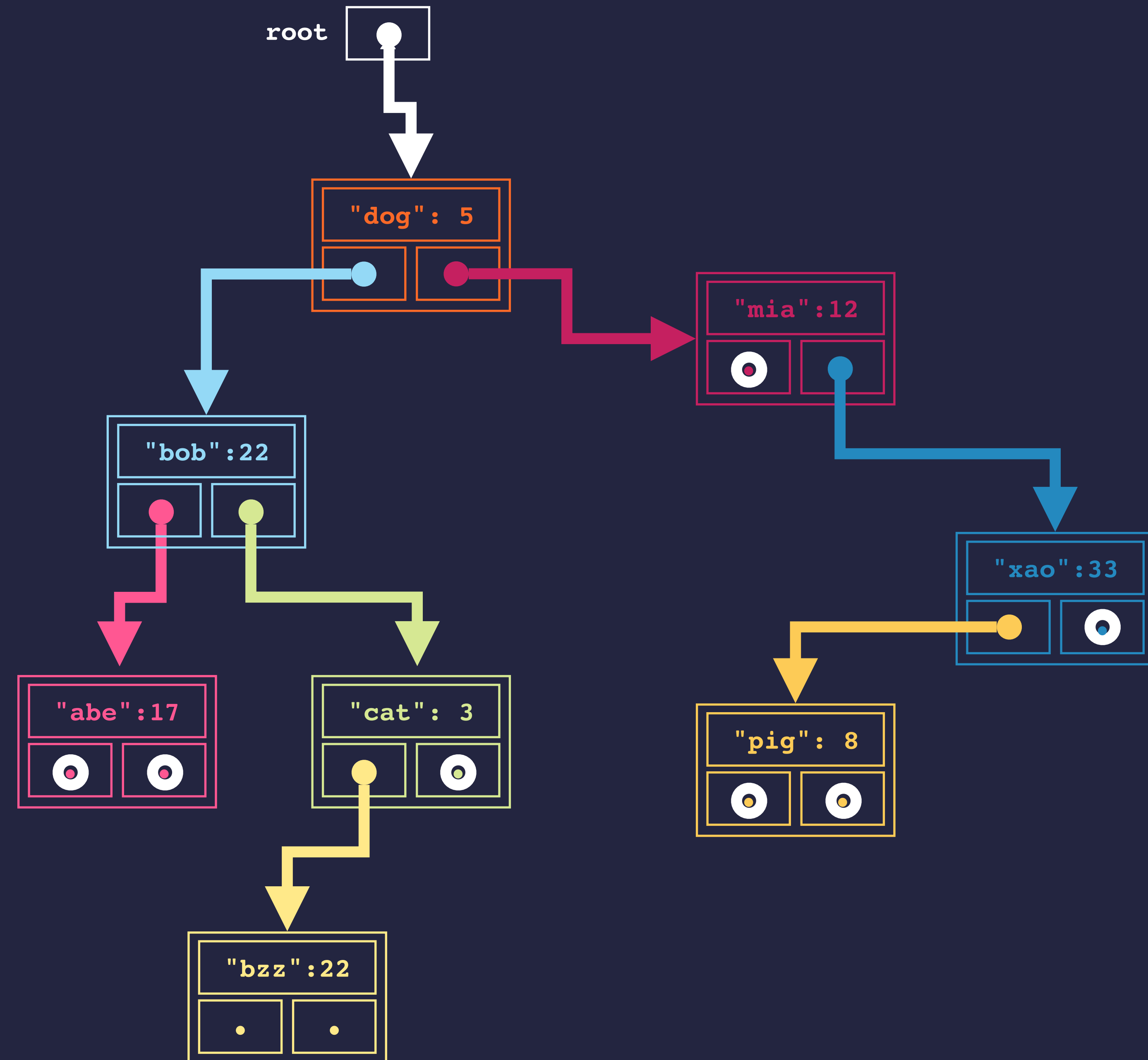
SEARCHING FOR AN ENTRY IN A BST

```
class BSTNode:
    def __init__(self,k,v):
        self.key = k
        self.value = v
        self.left = None
        self.right = None

class BSTree:
    def __init__(self):
        self.root = None

    def contains(self,k):
        curr = self.root
        while curr is not None:
            if k == curr.key:
                return True
            if k < curr.key:
                curr = curr.left
            if k > curr.key:
                curr = curr.right
        return False
```



ADDING AN ENTRY TO A BST

```
class BSTree:
    ...
    def update(self,k,v):
        parent = None
        curr = self.root
        while curr is not None:
            parent = curr
            if k == curr.key:
                curr.value = v
                return
            if k < curr.key:
                curr = curr.left
            if k > curr.key:
                curr = curr.right
        newNode = BSTNode(k,v)
        if parent is None:
            self.root = newNode
        elif k < prnt:
            parent.left = newNode
        else:
            parent.right = newNode
```

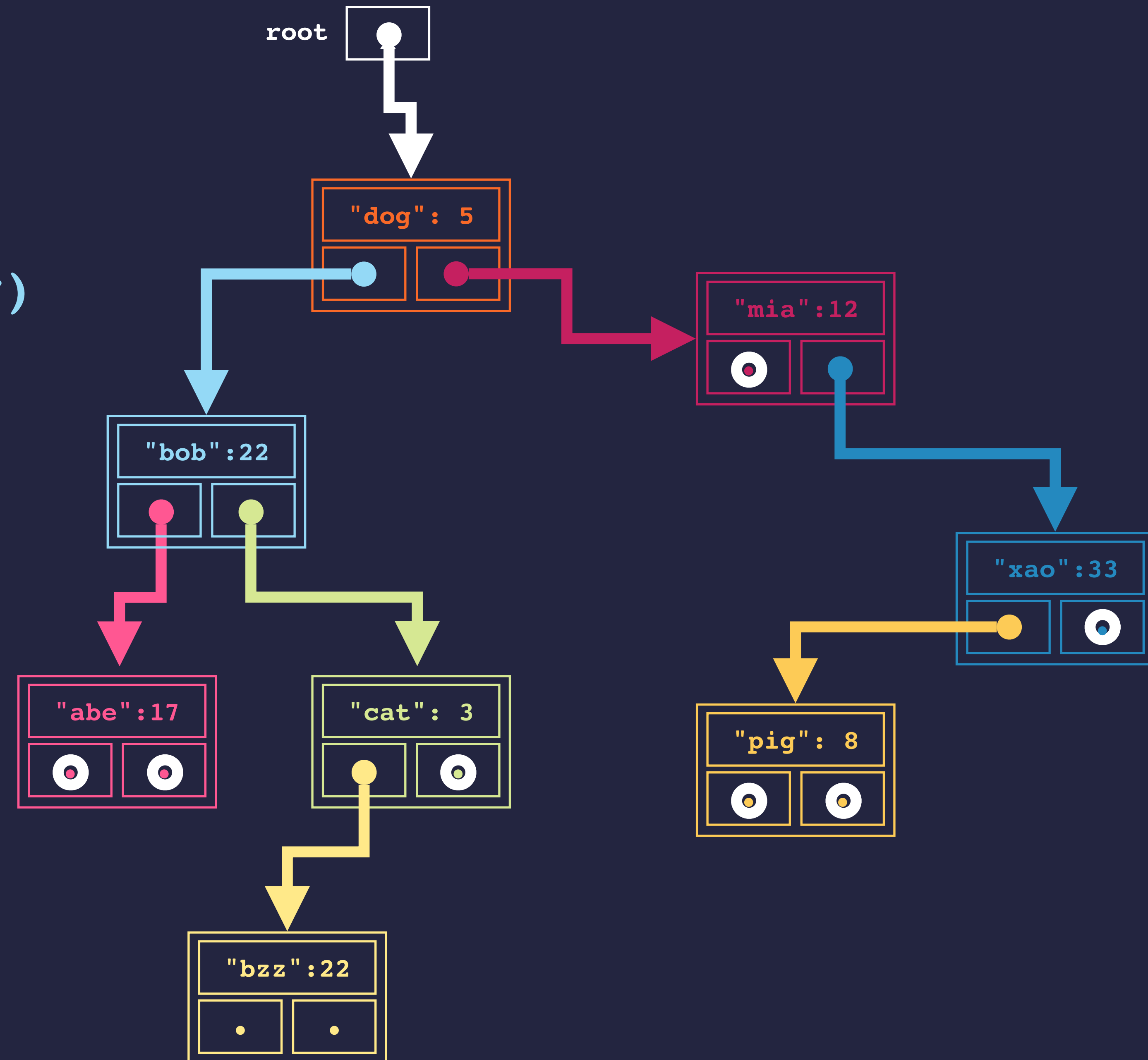


A BST CLASS

- **Operations**:
  - ➡ Searching for an entry by key.
  - ➡ Adding or updating an entry, ordered according to key, storing the value.
  - ➡ Removing an entry.
  - ➡ Visiting all the entries in sorted order.
- The first three operations rely on a *search*.
  - ➡ This works from the root, moving left or right.
- The last is *in-order traversal*. It is a recursive. **Example:** printing all the entries
  - ➡ You print all of the entries left of the root entry.
  - ➡ Then you print the root entry.
  - ➡ And then you print all of the entries right of the root entry.

TRAVERSING A BST

```
def printBST(node):
    if node is not None:
        printBst(node.left)
        entry = str(k)+":"+str(v)
        print(entry,end=' ')

class BSTree:
    ...
    def output(self):
        printBST(self.root)
        print()

>>> t.print()
abe:17 bob:22 bzz:22 cat:3 dog:5
mia:12 pig:8 xao:33
>>>
```



A TOUR OF THE BST CLASS CODE

- **Look at `BSTree.py`**

WRITING BETTER CODE

- As you become a more sophisticated programmer, you'll be driven to write good code. Some measures of "goodness":
  - Is it ***correct***?
  - Is it ***readable***?
  - Is it ***maintainable***?
  - And sometimes writing ***efficient*** code is important, too.

EFFICIENT CODE

- Efficient code is code that uses fewer resources when run. Examples:
  - ➡ It makes fewer calculations and/or takes fewer steps.
  - ➡ It uses less memory with its data structures.
- For both of these, a program will typically compute its answer faster.
  - It runs faster.

- Today we'll focus on ***running time***.

MEASURING RUNNING TIME

- Suppose you have two programs that compute the same result:
  - **program A** and **program B**.
  - **Q**: How do we determine which one is faster?
  - **A:** Run the code on typical inputs, measure the time it takes.

```
>>> import timeit
>>> i = 'import pow2'
>>> s = 'pow2.pow2(20)'
>>> timeit.timeit(stmt=s,setup=i,number=100)
0.0002275099977850914
```

- This will time 100 evaluations of `pow2(20)` then report the elapsed time in seconds.

MEASURING RUNNING TIME

- Suppose you have two programs that compute the same result:
  - ➡ **program A** and **program B**.
- But maybe…
  - You don’t have an exact sense of the typical inputs.
  - The size of typical inputs increases over the lifetime of the algorithms’ use.
  - The size and typicality of inputs might vary widely, depending on the application of the algorithm.
  - The computer might get upgraded in some near future. Or the programs might be rewritten for some unknown system.

MEASURING RUNNING TIME

- Suppose you have two programs that compute the same result:
  - ➡ **program A** and **program B**.
- But maybe…
  - You don't have an exact sense of the typical inputs.
  - The size of typical inputs increases over the lifetime of the algorithms' use.
  - The size and typicality of inputs might vary widely, depending on the application of the algorithm.
  - The computer might get upgraded in some near future. Or the programs might be rewritten for some unknown system.

- We then also work to ***estimate*** running times.
  - ➡ We use ***running time analysis***.

RUNNING TIME ANALYSIS

- Typical major concerns of running time estimation:

  - How does the running time scale (roughly) with input complexity?
    E.g. searching for an item in a list of size `n`
    - We will estimate "limiting" or *asymptotic* running time.

  - For a particular input sizer, what are the trickiest inputs the code will face?
    E.g. the search might have to scan the whole list.
    - We sometimes give *bounds* on the *worst cases*.

RUNNING TIME ANALYSIS

- Typical lesser concerns of running time estimation:
  - E.g. Something that runs 11% faster on one machine over another.
  - E.g. **Program A** runs a little slower on small inputs well on large inputs (0.2sec versus 0.15sec for **Program B**), even though **Program A** runs must faster on large inputs (20sec versus 1000sec for **Program B**).

ASYMPTOTIC EQUIVALENCE

- Let's formalize some of these ideas:
  - ➡ Two algorithms' running times are ***asymptotically equal*** if, for large inputs, which algorithm is faster *depends on the relative speed of their executing computers*.

- Example scenario:
  - Suppose **algorithm A** takes $n^3 - 4n^2$ steps on an n-bit input.
  - Suppose **algorithm B** takes $10n^3+15$steps on an n-bit input.
    - ➡ If **A** and **B** run on the same computer, A runs faster.
    - ➡ If **B** runs on a 100x speedier machine, it beats **A** on large inputs.

ASYMPTOTIC EFFICIENCY

- Let's formalize some of these ideas:
  - Two algorithms' running times are ***asymptotically equal*** if, for large inputs, which algorithm is faster depends on the relative speed of their executing computers.

- We define $\Theta(g(n))$, the set of functions asymptotically equal to g, with:

Definition: $f(n)$ is in the set $\Theta(g(n))$ whenever there exist positive constants L and U, and a positive constant m where

$$L\,g(n) \leq f(n) \leq U\,g(n)$$

for all $n \geq m$.

BIG THETA

Definition: $f(n)$ is in the class $\Theta(g(n))$ whenever there exist positive constants L and U, and a positive constant m where

$$L\,g(n) \leq f(n) \leq U\,g(n)$$

for all $n \geq m$.

Examples:

$n^3 - 4n^2$ is in the class $\Theta(10n^3+15)$

$10n^3+15$ is in the class $\Theta(n^3 - 4n^2)$

$n^3 - 4n^2$ is in the class $\Theta(n^3)$

$10n^3+15$ is in the class $\Theta(n^3)$

NOTE: All these functions grow as ***cubic functions*** of n.

BIG THETA

Definition: $f(n)$ is in the class $\Theta(g(n))$ whenever there exist positive constants L and U, and a positive constant m where

$$L\,g(n) \leq f(n) \leq U\,g(n)$$

for all $n \geq m$.

Examples from the last lecture:

Searching... an entire list of length n takes $\Theta(n)$ time.

...a balanced BST of size n to discover that a key is missing is $\Theta(\log_2(n))$ time.

A nested pair of loops that sum the products $i*j$ takes $\Theta(n^2)$ time.

Computing `pow2(n)` using repeated squaring takes $\Theta(\log_2(n))$ time.

Computing `pow2(n)` by multiplying 2 of n times takes $\Theta(n)$ time.

Computing `pow2(n)` by summing *1*s takes $\Theta(2^n)$ time.

BIG OH

Definition: $f(n)$ is in the class $\mathbf{O}(g(n))$ whenever there are positive U and m such that

$$0 \leq f(n) \leq U\, g(n)$$

for all $n \geq m$.

Examples:

$n^3 - 4n^2$ is in the class $\mathbf{O}(10n^3+15)$

$10n^3+15$ is in the class $\mathbf{O}(n^3 - 4n^2)$

n^2 is in the class $\mathbf{O}(n^3)$

$100000n+987987987$ is in the class $\mathbf{O}(n)$

- We use "big Oh" to say “asymptotically grows no faster than…”

BIG OH

Definition: $f(n)$ is in the class $\mathbf{O}(g(n))$ whenever there are positive U and m such that

$$0 \leq f(n) \leq U\, g(n)$$

for all $n \geq m$.

Examples:

Searching a list of length n takes $\mathbf{O}(n)$ time.

Searching a balanced BST of size n takes $\mathbf{O}(\log_2(n))$ time.

Searching a BST of size n takes $\mathbf{O}(n)$ time.

- We use "big Oh" to say “asymptotically grows no faster than…”

A CASE STUDY: SEARCHING A LIST

SEARCHING A LIST

```
def search(item, someList):
    i, n = 0, len(someList)
    while i < n:
        if someList[i] == item: return True
        i += 1
    return False
```

SEARCHING A SORTED LIST

Can we do better if a list is sorted?

- Suppose that

```
someList[0] ≤ someList[1] ≤ … ≤ someList[n-1]
```

SEARCHING A SORTED LIST

```
def binarySearch(item,someList):
    left, right = 0, len(someList)-1
    while left <= right:
        middle = (left + right) // 2
        if item == someList[middle]:
            return True
        elif item < someList[middle]:
            right = middle-1
        else:
            left = middle+1
    return False
```

SEARCHING A SORTED LIST

```
def binarySearch(item,someList):
    left, right = 0, len(someList)-1
    while left <= right:
        middle = (left + right) // 2
        if item == someList[middle]:
            return True
        elif item < someList[middle]:
            right = middle-1
        else:
            left = middle+1
    return False
```

- With each `someList[middle]` check, we eliminate half the undetermined list items from consideration.
- This means we inspect the list $\mathbf{O}(\log_2(n))$ times.

ANOTHER CASE STUDY: SORTING A LIST

BUBBLE SORT

- With bubble sort we make several left-to-right scans over the list.
  - We swap out-of-order values at neighboring locations
  - This "bubbles up" larger values so they "rise" to the right.

```
def bubbleSort(aList):
    n = len(aList)
    for scan in range(1,n):
        i = 0
        while i < n - scan:
            if aList[i+1] < aList[i]: # Out of order? Swap!
                aList[i],aList[i+1] = aList[i+1],aList[i]
            i += 1
```

BUBBLE SORT

- With bubble sort we make several left-to-right scans over the list.
  - We swap out-of-order values at neighboring locations
  - This “bubbles up” larger values so they “rise” to the right.

```
def bubbleSort(aList):
    n = len(aList)
    for scan in range(1,n):
        i = 0
        while i < n - scan:
            if aList[i+1] < aList[i]: #swap?
                aList[i],aList[i+1] = aList[i+1],aList[i]
            i += 1
```

- This means we only need to make *n* -1 scans.

BUBBLE SORT

- With bubble sort we make several left-to-right scans over the list.
  - We swap out-of-order values at neighboring locations
  - This “bubbles up” larger values so they “rise” to the right.

```
def bubbleSort(aList):
    n = len(aList)
    for scan in range(1,n):
        i = 0
        while i < n - scan:
            if aList[i+1] < aList[i]: #swap?
                aList[i],aList[i+1] = aList[i+1],aList[i]
            i += 1
```

- This means we only need to make *n* -1 scans.
- This means we can stop the scan earlier for later passes.

BUBBLE SORT ANALYSIS

- What is the running time of bubble sort?

```
def bubbleSort(aList):
    n = len(aList)
    for scan in range(1,n):
        i = 0
        while i < n - scan:
            if aList[i+1] < aList[i]:
                aList[i],aList[i+1] = aList[i+1],aList[i]
            i += 1
```

The if statement runs n - 1 times on the first scan, then n - 2 times on the second scan, then n - 3 times on the third scan, …

BUBBLE SORT ANALYSIS

- What is the running time of bubble sort?

```
def bubbleSort(aList):
    n = len(aList)
    for scan in range(1,n):
        i = 0
        while i < n - scan:
            if aList[i+1] < aList[i]:
                aList[i],aList[i+1] = aList[i+1],aList[i]
            i += 1
```

The if statement runs $n - 1$ times on the first scan, then $n - 2$ times on the second scan, then $n - 3$ times on the third scan, …

➡ The total number of swaps is

$$n(n-1)/2 = (n-1) + (n-2) + \ldots + 3 + 2 + 1$$

- Its running time scales ***quadratically*** with n.

SUMMARY

- In running time analysis use asymptotic notation to describe efficiency.
  - We use **Big Theta** for asymptotic equivalence.
  - We use **Big Oh** for asymptotic guarantees, i.e. *upper bounds*.
- Two classic searching and sorting algorithms:
  - Binary search is a ***logarithmic time*** algorithm. It works on sorted lists.
  - Bubble sort is a ***quadratic time*** algorithm. It sorts a list.
- Can we sort faster than in quadratic time?