CSC 148: Introduction to Computer Science

Week 10

Efficiency considerations

(revisited)

Comparison: search efficiency of

lists / LinkedLists / Trees / BSTs, etc..

Efficiency of BST operations... (we assume you did lab9!)

University of Toronto Mississauga,

Department of Mathematical and Computational Sciences

contains in a list

- Suppose v refers to a number: How efficient is the following statement in its use of time?

```
v in [97, 36, 48, 73, 156, 947, 56, 236]
```

- Roughly how much longer would the statement take if the list were
  - 10 times longer?
  - 1,000 times longer?
  - 1,000,000,000 times longer?
- Does it matter whether we used a built-in Python list or our implementation of LinkedList?

Speed of Search

- With either a Python list or a linked list of n elements
  - We must look at elements one by one
  - Each time we eliminate only one element from consideration
  - In the worst case, we look at all n elements
- Either way, it takes time proportional to n
- Can we make search in a Python list faster?

Ordering

- Suppose we know the list is sorted in ascending (or non-descending …) order
  - What strategy would you use to get to the value (if it exists) in a lot less steps than linear search?

- How does the running time scale up as we make the list 2, 4, 8, 16, 32, times longer?

log n

- Key insight: the number of times I repeatedly divide *n* in half, before we are down to 1 element, is the same as the number of times I double 1 before I reach (or exceed) *n*.
  - $log_2 n$, often known in CS as *log n*

- For an *n*-element list, it takes time proportional to ***n*** steps to decide whether the list contains a value, but **only** time proportional to ***log(n)*** to do the same thing on an **ordered** list
- What does that mean if n is 1,000,000? What about 1,000,000,000?

Aside: logarithms

- Recall:
  - $\log_a x = y \iff a^y = x$
  - Example: $2^5 = 32 \iff \log_2 32 = 5$

  - $log_2\ n$, is often known in CS as *log n*
    - After all, base 2 is our favorite base in CS .. :)

The Multiset ADT (search, insert, delete)

- Conclusion: For a sorted list with *n* items…
  - search is fast: O(log *n*) worst case, because of binary search
  - insert and delete can be slow, if inserting/removing from the *front* of the list – O(*n*) in the worst case

Tree Search?

- How efficient is **_contains_** on each of the following:
  - our general Tree class?
  - our BinarySearchTree class?
- As you'll see, for the latter the answer is: “it depends...”

The Multiset ADT (search, insert, delete)

- For a general Tree with *n* items…

```
for subtree in self._subtrees:
  if item in subtree:  # if subtree.__contains__(item):
    return True
return False
```

Search Speed in Tree or BinaryTree

- Strategy – similar to lists:
  - We must look at elements one by one
  - Each time we eliminate only one element from consideration
  - In the worst case, we look at all n elements
- Either way, it takes time proportional to n

The Multiset ADT (search, insert, delete)

- Bottom line: For a general tree with n items…
  - insert can be fast, if you insert as a child of the root – O(1)
  - search and delete can be slow, since you might need to check every item in the tree – O(n) in the worst case

Worst case running times so far…

Better?

operation	Sorted List	Tree	Binary Search Tree
search	O(log n)	O(n)	
insert	O(n)	O(1)	
delete	O(n)	O(n)	

Investigating BST efficiency

- Worksheet ...

Search efficiency in a BST?

- Recall the BST property
  - Exploit the ordering property to go either left or right when searching, but not both
  - => Search is narrowed down to only half of the yet-to-be-considered parts of the tree
  - If value is not in the tree, search all the way to leaf
  - => Worst case, maximum number of steps is equal to the max height of the tree
    - How big is the height in relation to the number of nodes?

Max number of values for height h

- What is the maximum number of values in a BST of height h:
  - 0?
  - 1?
  - 2?
  - 3?
  - 4?
  - 148?
  - h?

Height h based on number of nodes n

- Let's figure out the height h for a given number of nodes n
- We know that $n \leq 2^h - 1$

  $\Rightarrow n + 1 \leq 2^h$

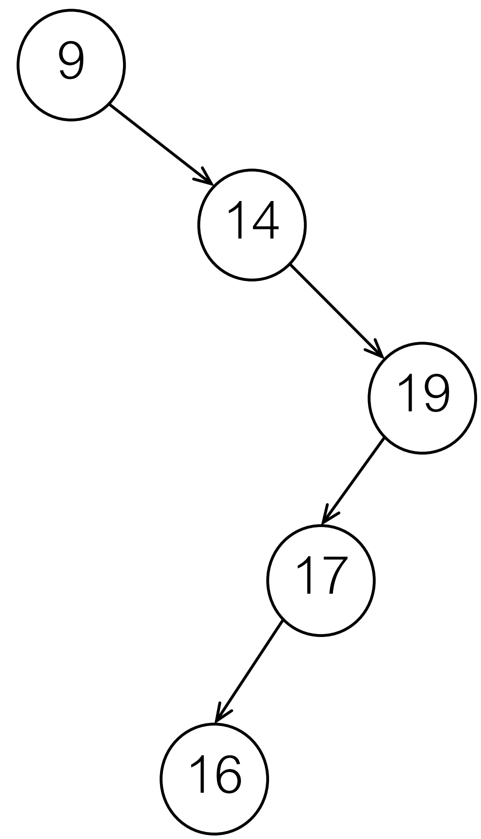
  $\Rightarrow \log_2 (n + 1) \leq h$

  $\Rightarrow h \geq \log_2 (n + 1)$

- So, time to go down a path in the BST will be proportional to *log n*
  - *Or will it?*

Search speed in BST

- Time will be proportional to *log n*, only if the tree is balanced!
- Example (imbalanced tree):
  - Time takes proportional to *n* in this case



- We say that a BST is balanced if its left and right subtrees have roughly equal heights, and these subtrees are also balanced.
- Balanced BSTs have height ≈ log *n*.

Conclusion: Search efficiency in a BST

- Searches that are directed along a single path are efficient:
  - a BST with 1 node has height 1
  - a BST with 3 nodes may have height 2
  - a BST with 7 nodes may have height 3
  - a BST with 15 nodes may have height 4
  - a BST with n nodes may have height $\lceil log\ n \rceil$.

1,000,000 nodes => height < 20!

- **If the BST is "balanced"**, then search takes $log\ n$ node accesses.

Efficiency of BST operations

- In a binary search tree, each Multiset operation's worst-case running time is proportional to the height h of the tree (where $\log n \leq h \leq n$).

operation	Sorted List	Tree	Binary Search Tree
search	$O(\log n)$	$O(n)$	**$O(h)$**
insert	$O(n)$	$O(1)$	**$O(h)$**
delete	$O(n)$	$O(n)$	**$O(h)$**

Efficiency: to be continued in later courses...

- In later courses, you will learn about balanced trees (AVL trees, red-black trees, etc.)

AVL trees ...

operation	Sorted List	Tree	BST	Balanced BST
search	O(log *n*)	O(*n*)	O(*h*): O(*n*)	O(*h*): **O(log *n*)**
insert	O(*n*)	O(1)	O(*h*): O(*n*)	O(*h*): **O(log *n*)**
delete	O(*n*)	O(*n*)	O(*h*): O(*n*)	O(*h*): **O(log *n*)**

Other Trees

- If you have enormous amounts of data, binary trees won't cut it (getting to a leaf is still expensive)
- Databases are such examples (large volumes of data, must fit in memory)
  - Increase the arity/branching factor!
  - Make heavy use of B-trees..
  - You will see this in later courses (CSC343, CSC443)