Today’s Outline

Announcements
- Homework 1 due TODAY at 10:59pm 😊
- Homework 2 out
  - Due online next Friday 10:59 pm

Today’s Topics
• Finish Asymptotic Analysis
• Dictionary ADT (a.k.a. Map): associate keys with values
  – Extremely common
• Binary Trees
Summary of Asymptotic Analysis

Analysis can be about:

• The problem or the algorithm (usually algorithm)
• Time or space (usually time)
• Best-, worst-, or average-case (usually worst)
• Upper-, lower-, or tight-bound (usually upper)

• The most common thing we will do is give an O upper bound to the worst-case running time of an algorithm.
Addendum: Timing vs. Big-O Summary

• Big-O
  – Examine the algorithm itself, not the implementation
  – Reason about performance as a function of $n$
  – For small $n$, an algorithm with worse asymptotic complexity might be faster

• Timing
  – Compare implementations
  – Focus on data sets other than worst case
  – Determine what the constants actually are
Let’s take a breath

• So far we’ve covered
  – Simple ADTs: stacks, queues, lists
  – Some math (proof by induction)
  – Algorithm analysis
  – Asymptotic notation (Big-Oh)

• Coming up….  
  – Many more ADTs!
    • Starting with dictionaries
The Dictionary (a.k.a. Map) ADT

• Data:
  – set of (key, value) pairs
  – keys must be comparable

• Operations:
  – insert(key, value)
  – find(key)
  – delete(key)
  – ...

  \[\text{insert(lauren, ...)}\]
  \[\text{find(mert)}\]
  \[\text{Mert Saglam, ...}\]

• Lauren Milne
  OH: Mon 1.30-2.30

• Mert Saglam
  OH: Wed 4-5

• Mauricio Hernandez
  OH: Fri 12:00-1:00
A Modest Few Uses

Used to store information with some key and retrieve it efficiently
  – Lots of programs do that!

- Search: phone directories
- Networks: router tables
- Operating systems: page tables
- Compilers: symbol tables
- Databases: dictionaries with other nice properties
- Biology: genome maps
- …

Possibly the most widely used ADT
Simple implementations

For dictionary with $n$ key/value pairs

- Unsorted linked-list
  - $O(1)\ast$ insert
  - $O(n)$ find
  - $O(n)$ delete

- Unsorted array
  - $O(1)\ast$ insert
  - $O(n)$ find
  - $O(n)$ delete

- Sorted linked list
  - $O(n)$ insert
  - $O(n)$ find
  - $O(n)$ delete

- Sorted array
  - $O(n)$ insert
  - $O(\log n)$ find
  - $O(n)$ delete

* Unless we need to check for duplicates

We’ll see a Binary Search Tree (BST) probably does better but not in the worst case (unless we keep it balanced)
Lazy Deletion

A general technique for making delete as fast as find:
  – Instead of actually removing the item just mark it deleted

Pros:
  – Simpler
  – Can do removals later in batches
  – If re-added soon thereafter, just unmark the deletion

Cons:
  – Extra space for the “is-it-deleted” flag
  – Data structure full of deleted nodes wastes space
  – May complicate other operations
Better dictionary data structures

There are many good data structures for (large) dictionaries

1. Binary trees
2. AVL trees
   - Binary search trees with guaranteed balancing
3. B-Trees
   - Also always balanced, but different and shallower
   - B-Trees are not the same as Binary Trees
     • B-Trees generally have large branching factor
4. Hashtables
   - Not tree-like at all

Skipping: Other balanced trees (e.g., red-black, splay)
Tree terms

- Root (tree)
- Leaves (tree)
- Children (node)
- Parent (node)
- Siblings (node)
- Ancestors (node)
- Descendents (node)
- Subtree (node)

- Depth (node)
- Height (tree)
- Degree (node)
- Branching factor (tree)
More tree terms

• There are many kinds of trees
  – Binary trees, linked lists, etc…

• There are many kinds of binary trees
  – binary search tree, binary heaps

• A tree can be balanced or not
  – A balanced tree with $n$ nodes has a height of $O(\log n)$
  – Use different “balance conditions” to achieve this
Kinds of trees

Certain terms define trees with specific structure

- **Binary tree**: Each node has at most 2 children (branching factor 2)
- **$n$-ary tree**: Each node has at most $n$ children (branching factor $n$)
- **Perfect tree**: Each row completely full
- **Complete tree**: Each row completely full except maybe the bottom row, which is filled from left to right

What is the height of a **perfect binary** tree with $n$ nodes?
A **complete 14-ary** tree?
Binary Trees

- **Binary tree**: Each node has at most 2 children (branching factor 2)

- Binary tree is
  - A root *(with data)*
  - A left subtree *(may be empty)*
  - A right subtree *(may be empty)*

- Representation:

<table>
<thead>
<tr>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>left pointer</td>
</tr>
</tbody>
</table>

- For a dictionary, data will include a key and a value
Binary Tree Representation
Binary Trees: Some Numbers

Recall: height of a tree = longest path from root to leaf (count edges)

For binary tree of height $h$:
- max # of nodes: $2^{h+1} - 1$
- max # of leaves: $2^h$
- min # of leaves: 1
- min # of nodes: $h + 1$

For $n$ nodes, we cannot do better than $O(\log n)$ height and we want to avoid $O(n)$ height
Calculating height

What is the height of a tree with root \texttt{root}?

\begin{verbatim}
int treeHeight(Node root) {
    ???
}
\end{verbatim}
Calculating height

What is the height of a tree with root \( \text{root} \)?

```java
int treeHeight(Node root) {
    if (root == null)
        return -1;
    return 1 + max(treeHeight(root.left),
                    treeHeight(root.right));
}
```

Running time for tree with \( n \) nodes:
\( O(n) \) – single pass over tree

Note: non-recursive is painful – need your own stack of pending nodes; much easier to use recursion’s call stack
Tree Traversals

A traversal is an order for visiting all the nodes of a tree

- **Pre-order:** root, left subtree, right subtree
- **In-order:** left subtree, root, right subtree
- **Post-order:** left subtree, right subtree, root

(an expression tree)
More on traversals

```java
void inOrderTraversal(Node t) {
    if (t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}
```

= current node  = processing (on the call stack)
= completed node  ✓ = element has been processed
More on traversals

```java
void inOrderTraversal(Node t) {
    if (t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}
```

![Diagram of tree traversal]

- **A** = current node
- **A** = processing (on the call stack)
- **A** = completed node
- ✓ = element has been processed
More on traversals

```java
void inOrderTraversal(Node t) {
    if (t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}
```

- **A** = current node
- **A** = processing (on the call stack)
- **A** = completed node
- ✓ = element has been processed
More on traversals

```java
void inOrderTraversal(Node t) {
    if (t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}
```

- **A** = current node
- **A** = processing (on the call stack)
- **A** = completed node
- ✓ = element has been processed
More on traversals

```java
void inOrderTraversal(Node t) {
    if (t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}
```

= current node  = processing (on the call stack)

= completed node  ✓ = element has been processed
void inOrderTraversal(Node t) {
    if (t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}
More on traversals

```java
void inOrderTraversal(Node t) {
    if (t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}
```

= current node  = processing (on the call stack)

= completed node ✓ = element has been processed
More on traversals

```java
void inOrderTraversal(Node t) {
    if(t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}
```

- **A** = current node
- **A** = processing (on the call stack)
- **A** = completed node
- ✓ = element has been processed
More on traversals

```java
void inOrderTraversal(Node t) {
    if (t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}
```

- **A** = current node
- **A** = processing (on the call stack)
- **A** = completed node
- ✓ = element has been processed
More on traversals

```java
void inOrderTraversal(Node t) {
  if (t != null) {
    inOrderTraversal(t.left);
    process(t.element);
    inOrderTraversal(t.right);
  }
}
```

- A = current node
- A = processing (on the call stack)
- A = completed node
- ✓ = element has been processed
**Tree Traversals**

A *traversal* is an order for visiting all the nodes of a tree

- **Pre-order:** root, left subtree, right subtree
  
  \[+ \times 2 \, 4 \, 5\]

- **In-order:** left subtree, root, right subtree
  
  \[2 \times 4 \, + \, 5\]

- **Post-order:** left subtree, right subtree, root
  
  \[2 \, 4 \times 5 \, +\]

(an expression tree)