Week 6 - Wednesday

COMP 2100

Last time

- What did we talk about last time?
- More recursion
- Merge sort

Questions?

Assignment 3

Recursion

Project 2

Infix to Postfix Converter

Symbol Tables

Symbol tables

- A symbol table goes by many names:
  - Map
  - Lookup table
  - Dictionary
- The idea is a table that has a two columns, a key and a value
- You can store, lookup, and change the value based on the key

Example

- A symbol table can be applied to almost anything:

Key	Value
Spiderman	Climbing and webs
Wolverine	Super healing
Professor X	Telepathy
Human Torch	Flames and flying
Deadpool	Super healing
Mr. Fantastic	Stretchiness

- The key doesn't have to be a **`String`**
- But it **must** be unique

Key	Value
1500	Introduction to Computer Science
1600	Introduction to Programming
2000	Object-Oriented Design
2100	Data Structures
3100	Software Engineering

Symbol table ADT

- We can define a symbol table ADT with a few essential operations:
  - put(Key key, Value value)
    - Put the key-value pair into the table
  - get(Key key):
    - Retrieve the value associated with key
  - delete(Key key)
    - Remove the value associated with key
  - contains(Key key)
    - See if the table contains a key
  - isEmpty()
  - size()
- It's also useful to be able to iterate over all keys

Ordered symbol tables

- The idea of order in a symbol table is reasonable:
  - You want to iterate over all the keys in some natural order
  - Ordering can give certain kinds of data structures (like a binary search tree) a way to organize

Ordered symbol table ADT

- An ordered symbol table ADT adds the following operations to a regular symbol table ADT:
  - Key min()
    - Get the smallest key
  - Key max()
    - Get the biggest key
  - void deleteMin()
    - Remove the smallest key
  - void deleteMax()
    - Remove the largest key
- Other operations might be useful, like finding keys closest in value to a given key or counting the number of keys in a range between two keys

Implementations

- Like other ADTs, a symbol table can be implemented in a number of different ways:
  - Linked list
  - Sorted array
  - Binary search tree
  - Balanced binary search tree
  - Hash table
- Note that a hash table cannot be used to implement an ordered symbol table
  - And it's inefficient to use a linked list for ordered

Sorted array

- We know how to make a sorted array symbol table
- A search is $\Theta(\log n)$ time, which is great!
- The trouble is that doing an insert takes $\Theta(n)$ time, because we have to move everything in the array around
- A sorted array is a reasonable model for a symbol table where you don't have to add or remove items

Trees

- Trees will allow us to make a sorted symbol table with the following miraculous properties:
  - $\Theta(\log n)$ get
  - $\Theta(\log n)$ put
  - $\Theta(\log n)$ delete
  - $\Theta(n)$ traversal (iterating over everything)
- Unfortunately, only balanced binary search trees will give us this property
- We'll start with binary search trees and build up to balanced ones

Trees

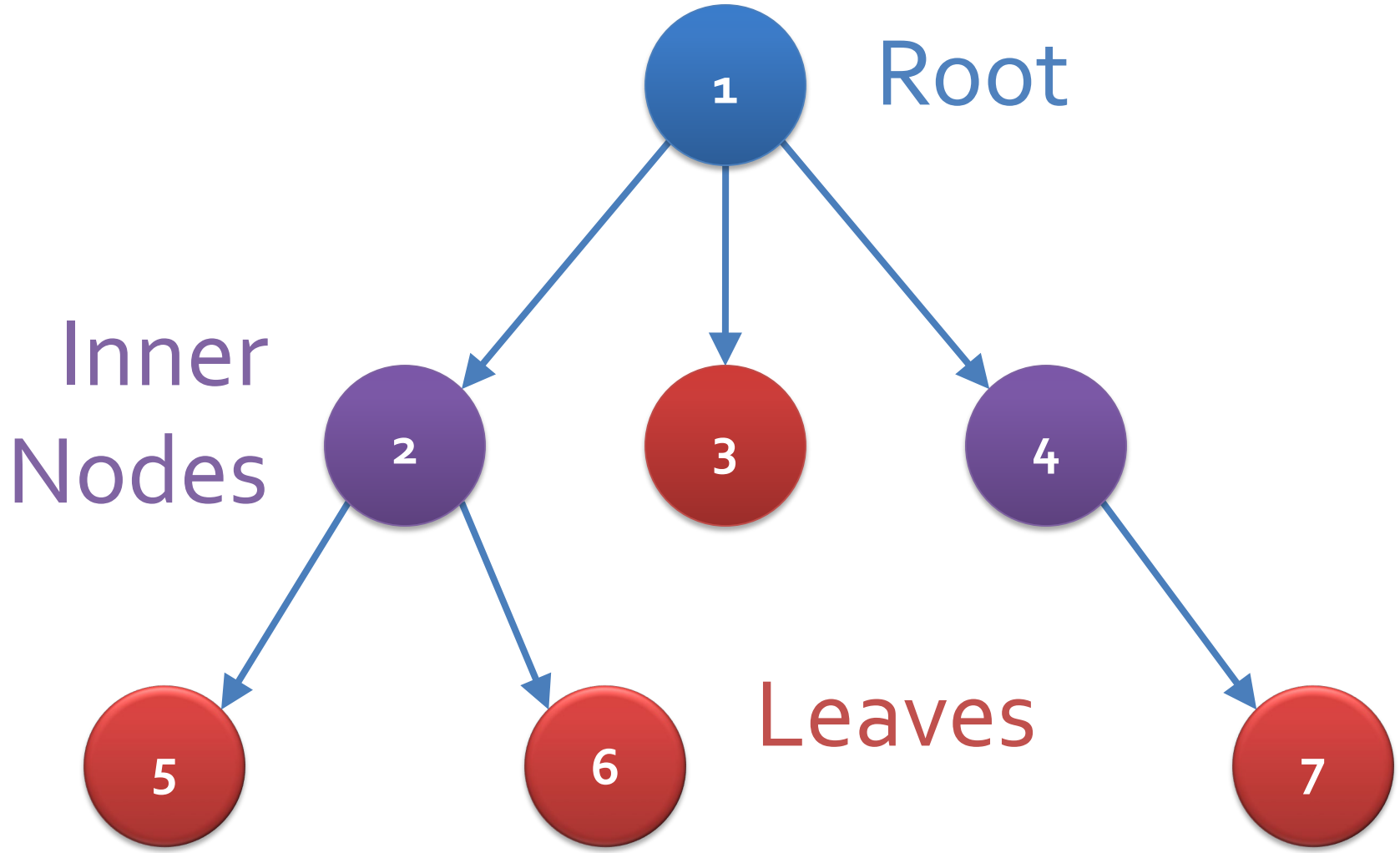
What is a tree?

- A tree is a data structure built out of nodes with children
- A general tree node can have any non-negative number of children
- Every child has exactly one parent node
- There are no loops in a tree
- A tree expressions a hierarchy or a similar relationship

Terminology

- The **root** is the top of the tree, the node which has no parents
- A **leaf** of a tree is a node that has no children
- An **inner node** is a node that does have children
- An **edge** or a **link** connects a node to its children
- The **depth** of a node is the length of the path from the root to the node
  - Note that some definitions add 1 to this definition
- The **height** of the tree is the greatest depth of any node
- A **subtree** is a node in a tree and all of its children

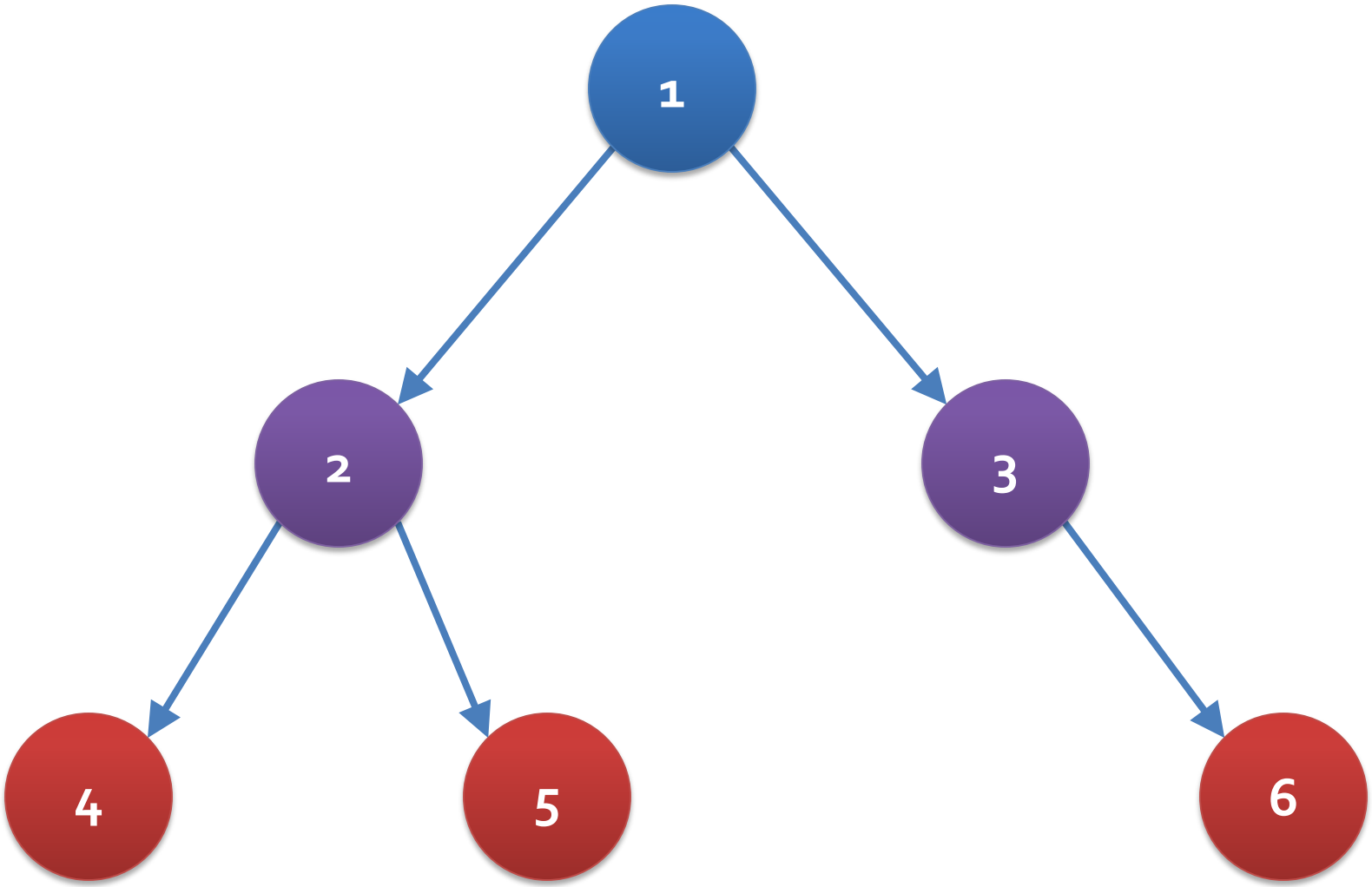
A tree



Binary tree

- A binary tree is a tree such that each node has two or fewer children
- The two children of a node are generally called the **left child** and the **right child**, respectively

Binary tree



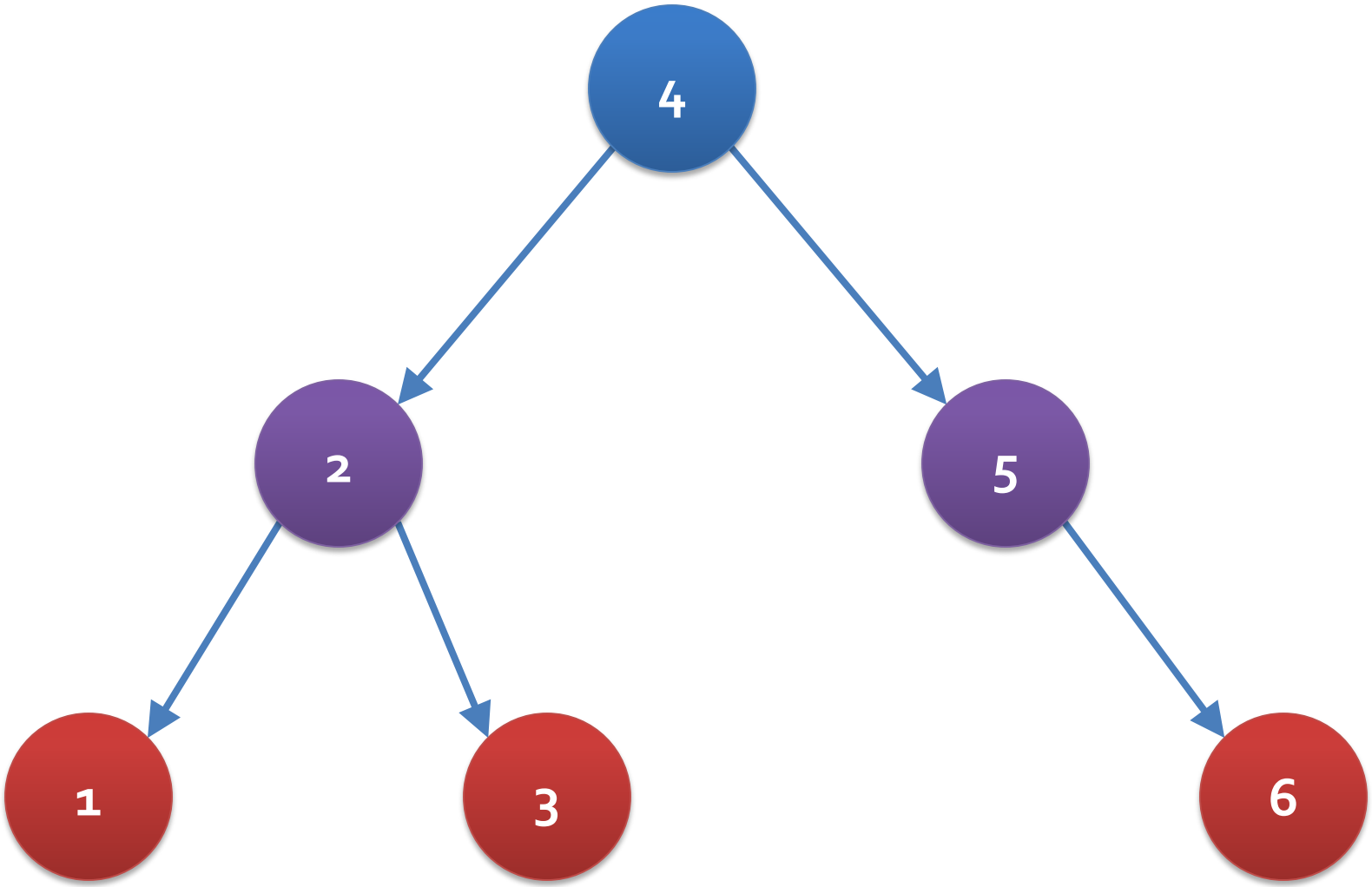
Binary tree terminology

- **Full binary tree:** every node other than the leaves has two children
- **Perfect binary tree:** a full binary tree where all leaves are at the same depth
- **Complete binary tree:** every level, except possibly the last, is completely filled, with all nodes to the left
- **Balanced binary tree:** the depths of all the leaves differ by at most 1

Binary search tree (BST)

- A binary search tree is binary tree with three properties:
  1. The left subtree of the root only contains nodes with keys less than the root's key
  2. The right subtree of the root only contains nodes with keys greater than the root's key
  3. Both the left and the right subtrees are also binary search trees

BST



BST facts

- Let h be the height of a binary tree
- A perfect binary tree has $2^{h+1} - 1$ nodes
- Alternatively, a perfect binary tree has $2L - 1$ nodes, where L is the number of leaves
- A complete binary tree has between 2^h and $2^{h+1} - 1$ (exclusive) nodes
- A binary tree with n nodes has $n + 1$ `null` links

BST performance

- Unbalanced BST:
  - $\Theta(n)$ find
  - $\Theta(n)$ insert
  - $\Theta(n)$ delete
- Balanced BST:
  - $\Theta(\log n)$ find
  - $\Theta(\log n)$ insert
  - $\Theta(\log n)$ delete

Implementation of a BST

Basic BST class

```
public class Tree {
  private static class Node {
    public int key;
    public Object value;
    public Node left;
    public Node right;
  }

  private Node root = null;

  …
}
```

The book uses a generic approach, with keys of type **`Key`** and values of type **`Value`**. The algorithms we'll use are the same, but I use **`int`** keys to simplify comparison.

Calling methods on trees

- Almost all the methods we call on trees will be recursive
- Each will take a **`Node`** reference to the root of the current subtree
- Because the root is private, assume that every recursive method is called by a public, non-recursive proxy method:

  `public Type doSomething()` calls

  `private static Type doSomething( Node root )`

Finding the smallest key

```
private static int min(Node node)
```

- Proxy:

```
public int min() {
   return min( root );
}
```

- What's the code?
- Use recursion!

Finding the largest key

```
private static int max(Node node)
```

- Proxy:

```
public int max() {
   return max( root );
}
```

- What's the code?
- Use recursion!

Getting a value from the key

```
private static Object get(Node node, int key)
```

- Proxy:

```
public Object get(int key) {
   return get( root, key );
}
```

- What's the code?
- Use recursion!

Adding a key-value pair

```
private static Node put(Node node, int key,
  Object value)
```

- Proxy:

```
public void put(int key, Object value) {
   root = put( root, key, value );
}
```

- What's the code?
- Use recursion!

Quiz

Upcoming

Next time…

- Traversals
- Deletion
- Breadth-first search and level-order traversal

Reminders

- Work on Project 2
- Finish Assignment 3
  - Due this Friday
- Keep reading Section 3.2