iOS Dev Accelerator Week 6 Day 4

- CoreData/Testing in Swift
- Version Migration
- iCloud
- Multiple Contexts
- Binary Search Trees

Testing with CoreData

Writing tests in swift

- Writing tests in swift still follows the same principles we covered yesterday (XCTest, F.I.R.S.T), but there a few gotchas
- Swift has namespacing, which is why we dont have to #import <ClassName>.h in our swift files. Everything in your Swift app's application target, or module, knows about everything else.
- This is big improvement over Objective-C, but for writing tests it adds an extra hurdle
- You will need to do 2 things to write tests in swift projects

Writing tests in swift

1. At the top of your test file, import your Swift application's module. This will look like:
    - import Name_of_your_project
        - No quotes or .swift necessary. It probably wont autofill which sucks. If you have spaces in your project name, use underscores.
2. Every custom class you are going to test, you must add to the test target as well.

Demo

Swift and CoreData

Swift and CoreData

- Swift and CoreData works pretty much exactly like Objective-C and CoreData.
- Same stack, same MOM, same methods
- The only weirdness comes when generating your custom subclasses with Xcode.

Swift and CoreData

- After generating your subclasses using the auto generate feature in Xcode, you must modify the ‘Class’ entry in the attribute inspector of each entity in your MOM file.
- You simply need to add the name of your project and then a dot, and then the name of the entity.
- So if your project is called CoreDataHotel and your entity is called Hotel, it should say CoreDataHotel.Hotel.
- If you forget to do this, your app will crash when you try to use a custom subclass object and print this error message to the console:
- “Unable to load class named ‘ClassName’ for entity ‘EntityName’, using default NSManagedObject instead.

Demo

Versioning & Migration

Migrating with Core Data

- Its pretty common for your Managed Object Model to change during the development of your app
- Especially between releases, as you are adding or removing new features
- So the question is, how do we account for user's data when they update their app, and the MOM has changed? Their old entities, attributes, and relationships probably wont match the new MOM layout, which is a big problem. How can we properly accommodate our users?
- With Migration.

Migrating with Core Data

- A rule of thumb: Anytime you make changes to your data model, you need to migrate
- However, certain situations can avoid migrations all together. For example, if you are using core data as an offline cache, you can simply clear the old cache out and start new.
- When migration is needed, you need to **a)create a new version of the data model, and b)provide a migration path**

Making the decision

- When you first add a store to your persistent store coordinator, Core Data does a few things under the hood:
- Core Data compares the store’s model version with the coordinators model version.
- If the versions don't match, Core Data will perform a migration (if enabled)

Starting the process

- When the migration process starts, Core Data looks at both the original data model and the destination/new model.
- It use these two models to create a mapping model for the migration.
- Core Data uses this mapping model to convert data in the original store so it can be used in the new model

Under the hood

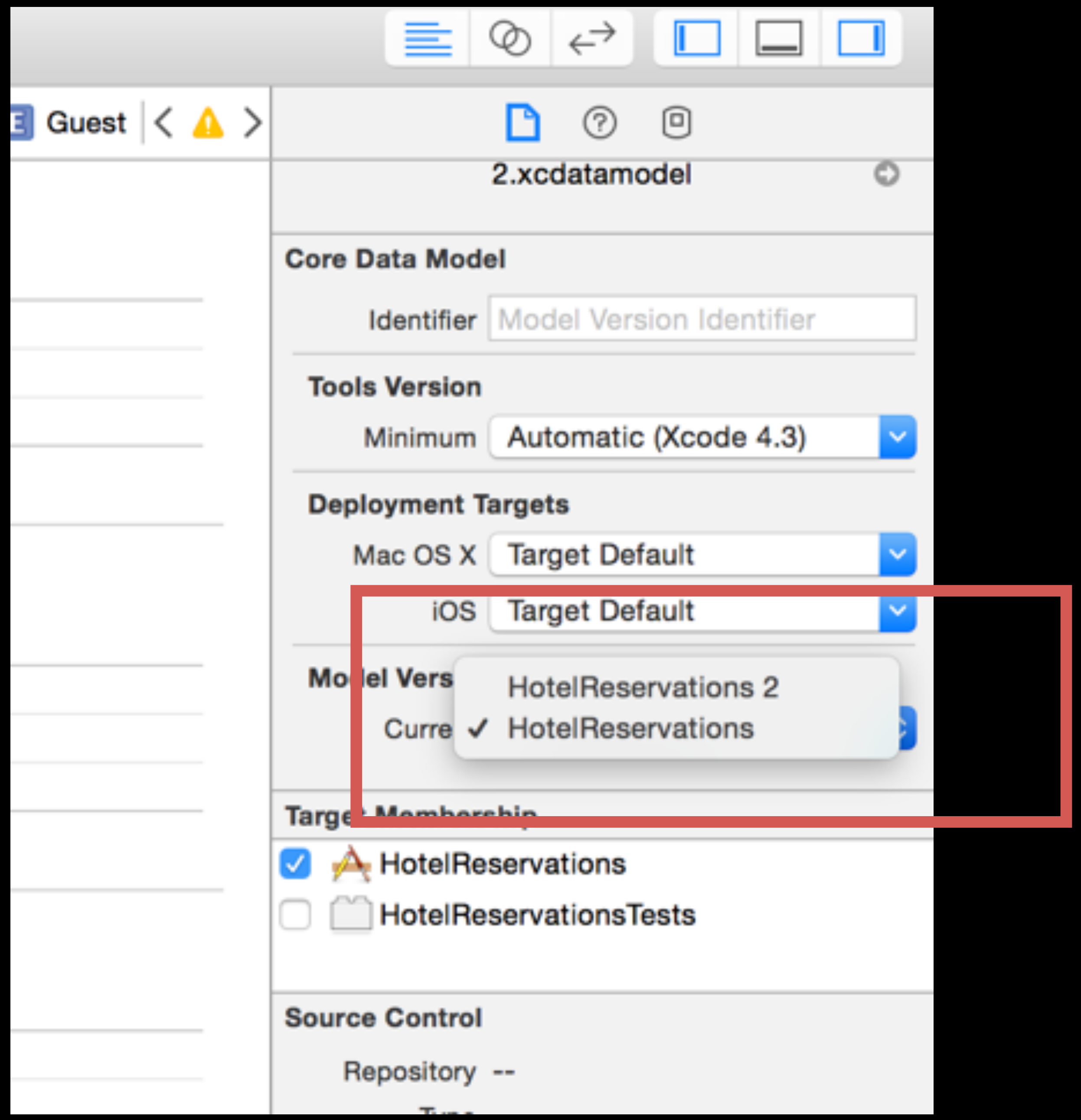
- Under the hood, the migration has 3 steps:
    1. Core Data copies all of the objects from the old data store to the next store
    2. Core Data connects all the objects according to the relationships
    3. Core Data enforces any data validation in the destination model
- If something goes wrong, core data reverts to the old data store. The old data is never deleted until a successful migration is achieved.

The different types of a Core Data Migration

- Officially, Apple only recognizes two types of migrations: lightweight and not lightweight migrations
- But really there is a few different types of migrations, ordered from simplest to most complex:
- **Lightweight migration**: least amount of work involved for you. Its as simple as enabling a couple of flags when you setup your stack. But there are limitations.
- **Manual migration:** More work for you. You have to manually specify how the old data will map to the new data model. GUI tools are provided in Xcode, similar to how you setup your MOM
- **Custom Manual Migration:** You use the GUI mapping model from manual migration, but also add custom code to specific any custom transformation logic.
- **Fully Manual Migration:** Custom version detection logic and custom handling of the migration process

Lightweight Migration

- Lightweight Migration is pretty straight forward process:
  1. With your MOM file open, click Editor>Add Model Version and give your new model a name
  2. Change the version of your MOM to the new version:
  3. Make your changes to your MOM
  4. Generate new subclasses
  5. Modify your persistent store to contain special options to enable automatic migration



Guest

2.xcdatamodel

Core Data Model
Identifier: Model Version Identifier

Tools Version
Minimum: Automatic (Xcode 4.3)

Deployment Targets
Mac OS X: Target Default
iOS: Target Default

Mo el Vers
Curre
HotelReservations 2
✓ HotelReservations

Targe Membership
✓ HotelReservations
HotelReservationsTests

Source Control
Repository --

Enabling Automatic Migration

- In order for Core Data to automatically migrate between your MOM versions, you need to include a specific set of options when adding your persistent store to your PSC
- The options parameter of addPersistentStoreWithType:configuration:URL:options:error: takes in a dictionary of options.
- The two key-value pairings you need are:
  - NSMigratePersistentStoreAutomaticallyOption : true (enables migration in general)
  - NSInferMappingModelAutomaticallyOption : true (tells core data to automatically merge if its capable, set to false if you are doing a manual merge)

Lightweight Limitations

- Lightweight Migrations have some limitations. They can do pretty much any obvious migration pattern:
    - Deleting entities, attributes, relationships
    - Renaming entities, attributes, relationships
    - Adding new attributes
    - Changing optional attributes to required or vice versa
    - Changing entity hierarchy
    - changing a relationship from one to many
    - changing a to many relationship to ordered or unordered
- As you can see, lightweight migrations are pretty powerful! They should cover 90% of your migration scenarios. You should always strive to keep your migrations lightweight.

Demo

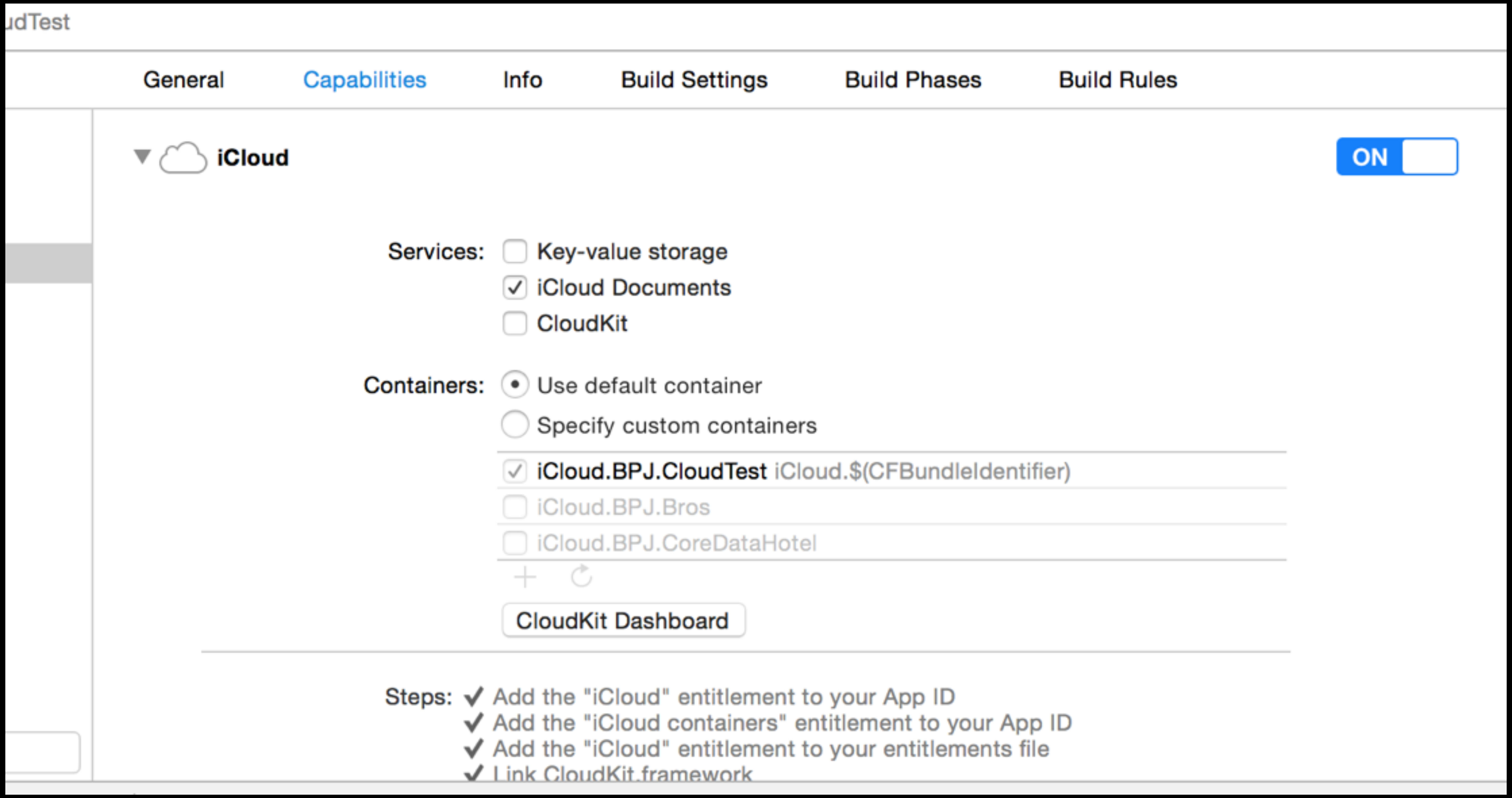
iCloud

iCloud

- “iCloud is a cloud service that gives your users a consistent and seamless experience across all of their iCloud-enabled devices”
- iCloud works with things called ‘**ubiquity** containers’, which are special folders that your app uses to store data in the cloud.
- Whenever you make a change in your ubiquity container, the system uploads the changes to the cloud.
- iCloud is closely integrated with CoreData to help persist your managed objects to the cloud. This is another incentive to use core data to manage your model layer!
- Prior to iOS7, iCloud was extremely hard to setup, unreliable, and impossible to debug. With iOS7, it is now easy to setup, reliable, but still difficult to debug (especially on simulator).

Enabling iCloud in your app

- To enable use of iCloud in your app, you have to enable the iCloud capability in your app's capability screen in Xcode.
- Navigate to your app's target in the project settings in Xcode, and select the capabilities pane. Switch the iCloud option on and follow the configuration prompts.
- Put checks next to the cloud functionality you want to use (for core data, its iCloud documents)


udTest

General | Capabilities | Info | Build Settings | Build Phases | Build Rules

iCloud [✓ ON]

Services:
- [ ] Key-value storage
- [✓] iCloud Documents
- [ ] CloudKit

Containers: [✓ Use default container] Specify custom containers
- [✓] iCloud.BPJ.CloudTest iCloud.$(CFBundleIdentifier)
- [ ] iCloud.BPJ.Bros
- [ ] iCloud.BPJ.CoreDataHotel

CloudKit Dashboard

Steps:
✓ Add the "iCloud" entitlement to your App ID
✓ Add the "iCloud containers" entitlement to your App ID
✓ Add the "iCloud" entitlement to your entitlements file
✓ Link CloudKit.framework

Ubiquity Containers

- Once you enable iCloud in the capabilities tab, your app can now read and write to one or more ubiquity containers.
- In the ubiquity container, core data will store transaction logs, aka records of changes made to your persistence store.
- Core data uses the ‘multi-master replication’ pattern to sync data between your devices.
- The persistent store itself is not stored in the cloud. Rather it is stored on each device, in a special Core Data managed directory called CoreDataUbiquitySupport inside your app sandbox.
- As users change iCloud accounts (rare but it happens) Core Data will manage multiple stores inside this directory
- Each transaction log is simply a plist that keeps track of all insertions, deletions, and updates to your entities in Core Data. This is the meat of how iCloud w/ coredata works

Setting up your core data stack to work with iCloud

- iOS 7 made enabling iCloud in your core data stack as simple as setting up your persistent store with a dictionary containing a few special options:
    - NSPersistentStoreUbiquitousContentNameKey - **required**, option to specify that a persistent store has a given name in ubiquity (No periods allowed!)
    - NSPersistentStoreUbiquitousContentURLKey - **optional in iOS7, but you should do it or else stuff wont work right**, specifies the subdirectory path for the transaction logs
    - Theres a few more optional iCloud related keys, look in the docs.
- You will also want to include a flag for merging options. Apple recommends NSMergeByPropertyObjectTrumpMergPolicy, which will merge conflicts and give priority to those in memory vs disk.

Persistent Store Setup

- The method addPersistentStoreWithType:configuration:URL:options:error: with iCloud options passed in returns a store almost immediately.
- It does this by first returning a fallback, local store that serves as a placeholder.
- Changed made in the fallback store will be automatically migrated to the iCloud store when its ready to use.
- “Using local storage: 1” is printed to the console when the fallback store is setup and being used.
- Once the iCloud store is ready to use you will see “Using local storage: 0”, which means the iCloud store is now ready for use and you should begin seeing your data download down (if there is any)

Demo

iCloud notifications

- There are 3 primary notifications you can observe to react to iCloud Events :
    - NSPersistentStoreCoordinatorStoresWillChangeNotification
    - NSPersistentStoreCoordinatorStoresDidChangeNotification
    - NSPersistentStoreDidImportUbiquitousContentChangeNotification

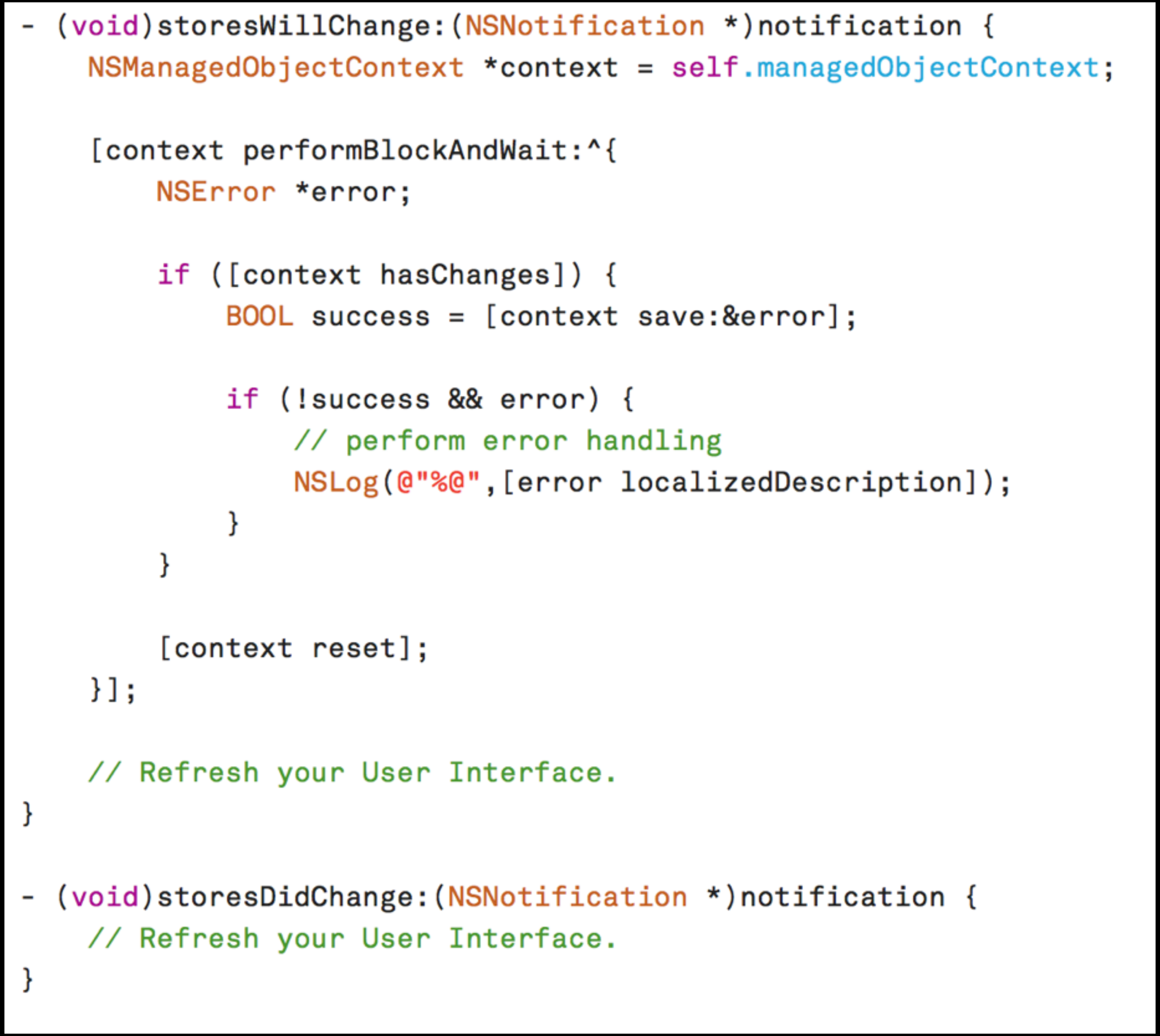
DidChangeNotification

- NSPersistentStoreCoordinatorStoresDidChangeNotification
    - You will receive this notification to let you know the persistent store has been configured to use.
    - The Notification’s user info dictionary contains the created NSPersistentStore

`WillChangeNotification`

- NSPersistentStoreCoordinatorStoresWillChangeNotification
  - Will change happens before Did change
  - Will change is fired right before core data is about to swap out data from temporary local store to the ubiquity store.
  - In the handler for this notification you need to save if there are changes in the context that is currently hooked up to the temp store (so those last changes are moved to the new store), and then reset the context.

WillChange and DidChange Notification Handlers

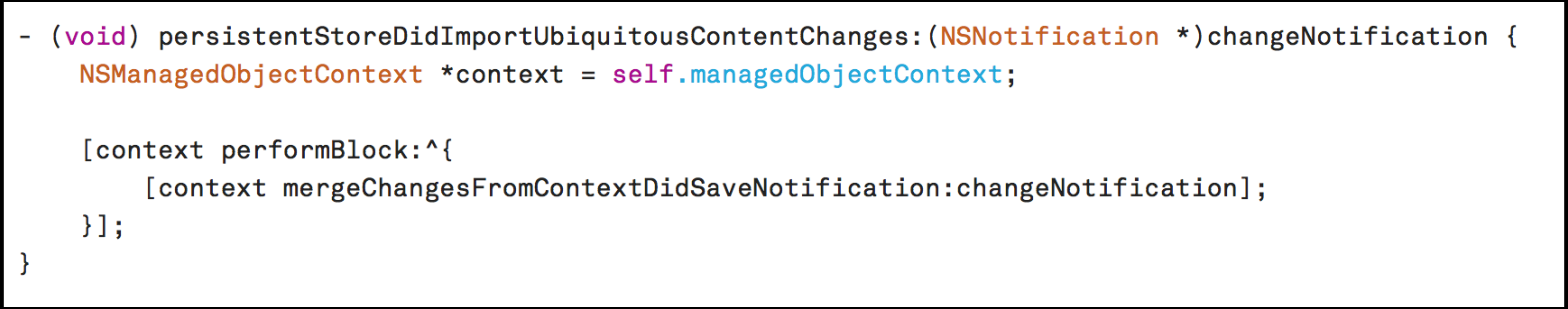


```
- (void)storesWillChange:(NSNotification *)notification {
    NSManagedObjectContext *context = self.managedObjectContext;

    [context performBlockAndWait:^{
        NSError *error;

        if ([context hasChanges]) {
            BOOL success = [context save:&error];

            if (!success && error) {
                // perform error handling
                NSLog(@"%@",[error localizedDescription]);
            }
        }

        [context reset];
    }];

    // Refresh your User Interface.
}

- (void)storesDidChange:(NSNotification *)notification {
    // Refresh your User Interface.
}
```

DidImportUbiqutiousContentChange

- NSPersistentStoreDidImportUbiquituousContentChangeNotification
  - Not only does Core Data import changes in the Cloud during the first time setup, it also does it while your app is running!
  - When your ubiquity container receives changes from iCloud, Core Data posts this notification
  - The user info of the notification contains all the data you need to merge in the changes coming from the cloud

DidImportUbiqutiousContentChange



```
- (void) persistentStoreDidImportUbiquitousContentChanges:(NSNotification *)changeNotification {
    NSManagedObjectContext *context = self.managedObjectContext;

    [context performBlock:^{
        [context mergeChangesFromContextDidSaveNotification:changeNotification];
    }];
}
```

After the merge, you will probably want to refresh your UI (use notification center) and also run reduplication code (if that matters for your app)

Demo

Multiple Context

Multiple Contexts

- So far in our Core Data implementation, we have only operated with one context.
- One context works great for most apps, and its generally the recommended way to implement core data
- But sometimes you need to use multiple contexts. 2 primary benefits:
    - Can harness multithreading by creating a context to be used off the main queue
    - Can treat a 2nd context as a sort of draft-scratch pad, separate from the main context. Calling save on the main context won’t cause whatever changes/things you have created on the 2nd context to save.

How do you create multiple contexts?

- 2 distinct ways to create another context:
  - Use the built initializer on NSManagedObjectContext that takes in a concurrency type, and then assign it a peristentStoreCoordinator
  - Create a child context based off of your already existing primary context

NSManagedObjectContext & Concurrency Type

- When using the NSManagedObjectContext initializer that takes in a concurrency type, the parameter is an enum of type NSManagedObjectContextConcurrencyType:
    - ConfinementConcurrencyType: Treat this as deprecated, don't use it. Uses the old thread confinement pattern.
    - MainQueueConcurrencyType: Creates a context associated with the main queue
    - PrivateQueueConcurrencyType: Create a context associated with a private dispatch queue (not the main thread!)

NSManagedObjectContext & Concurrency Type

- After using the initializer with a concurrency type, you will need to assign the new context a PSC:

```
let backgroundContext = NSManagedObjectContext(concurrencyType:
  NSManagedObjectContextConcurrencyType.PrivateQueueConcurrencyType)

backgroundContext.persistentStoreCoordinator = self.
  persistentStoreCoordinator
```

- This is all the setup you need to do for your background-queue-ready context

Special Considerations For Concurrency and CoreData

- There are certain things you need to keep in mind you while you code concurrency into your core data implementation:
- NSManagedObject's are not thread safe. NSManagedObjectID's are thread safe. So when you need to pass managed object(s) between threads, **you have to pass them by their managedObjectID's (or refetch from desired context)**
- If a managedObjectContext is created with a private queue concurrency type, you can never have it execute code on the main thread. When you need to work with your concurrent context, call performBlock or performBlockAndWait:
- If you try to work with your main queue managed object context on a background thread, bad stuff can happen. Don't do it!

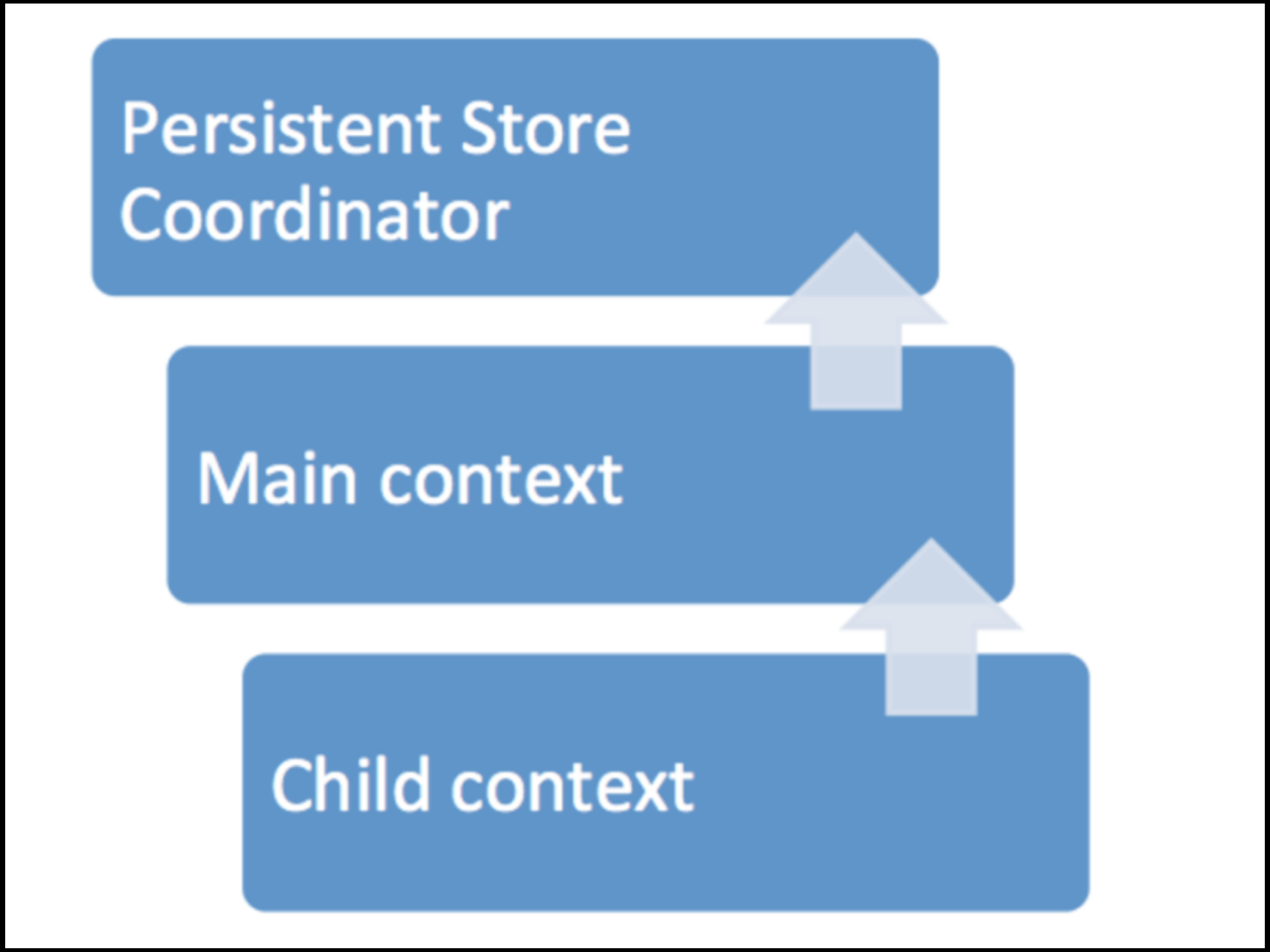
merging in changes from a context

- If you have multiple contexts, it is critical to keep both updated when saves are made on either one
- To do this, you can register for the notification NSManagedObjectContextDidSaveNotification
- You will need to register for this twice. One that listens when the notification is emitted from the your main queue context, and one when emitted from your background queue context.
- The handler for the notification when emitted from the main queue context should call mergeChangesFromContextDidSaveNotification: on the background queue context, and vice versa.

Demo

Child Contexts

- You can create child managed object contexts that point to their ‘parent’
- Every context has a parent:



Child-Parent Contexts

- Calling save on a child automatically applies those changes to its parent context. So it goes up one level (only one level!)
- Things don't actually save until you call save on the context that is a direct child of the persistent store.
- Only use Child contexts for very small operations on your data. This is because calling save on a child context is not nearly as efficient as calling save on a completely separate context (the previous topic)*.
- Basically dont use child contexts, just know what they are.

Sourced from http://floriankugler.com - backstage with nested managed object contexts

Binary Search Tree

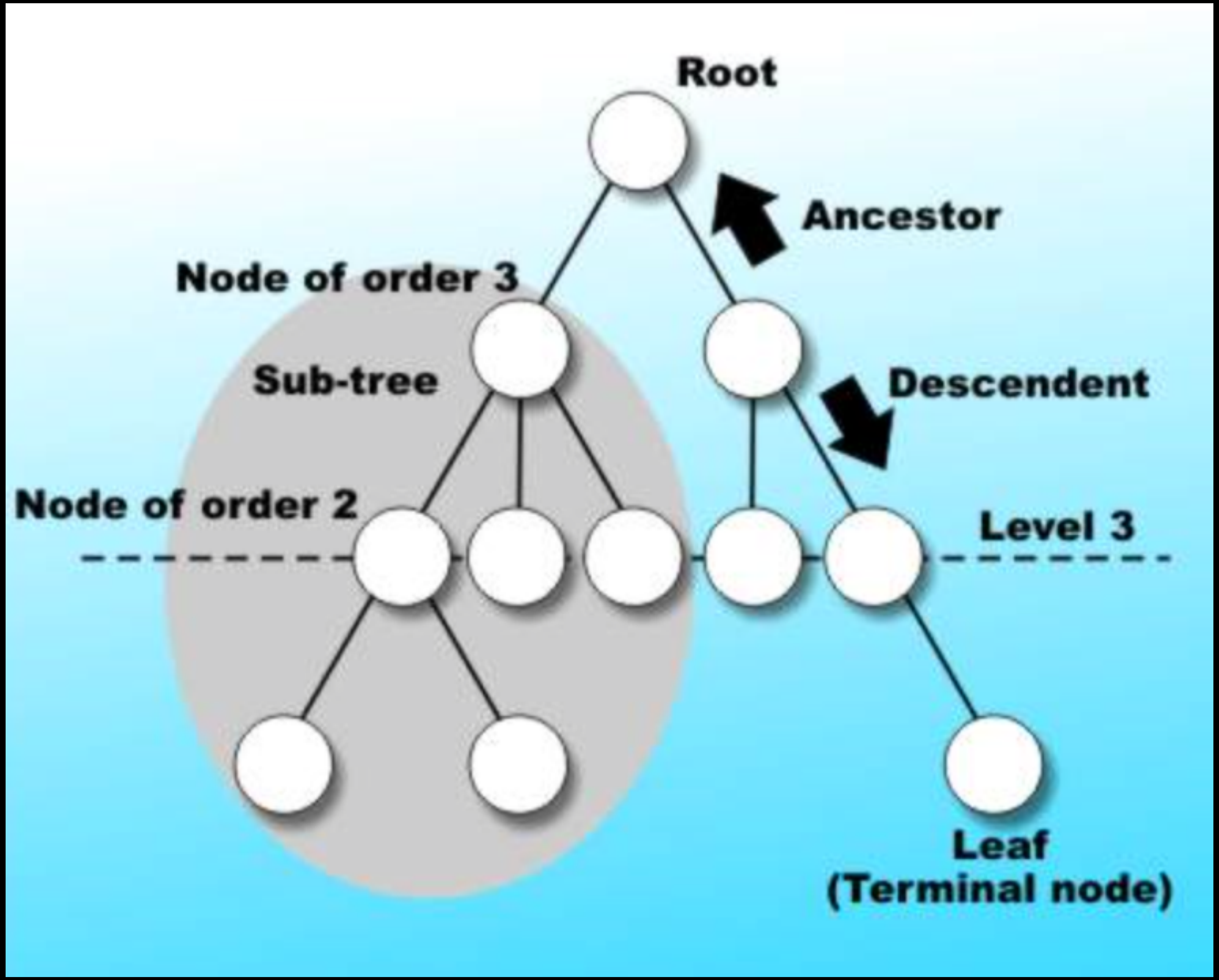
- “A binary search tree is a binary tree where each node has a comparable key and satisfies the restriction that the key in any node is larger than all keys in all nodes in that node’s left subtree, and smaller than the keys in all nodes in that nodes right subtree”

-princeton.edu

Lets dissect that definition a bit

Binary Search Tree

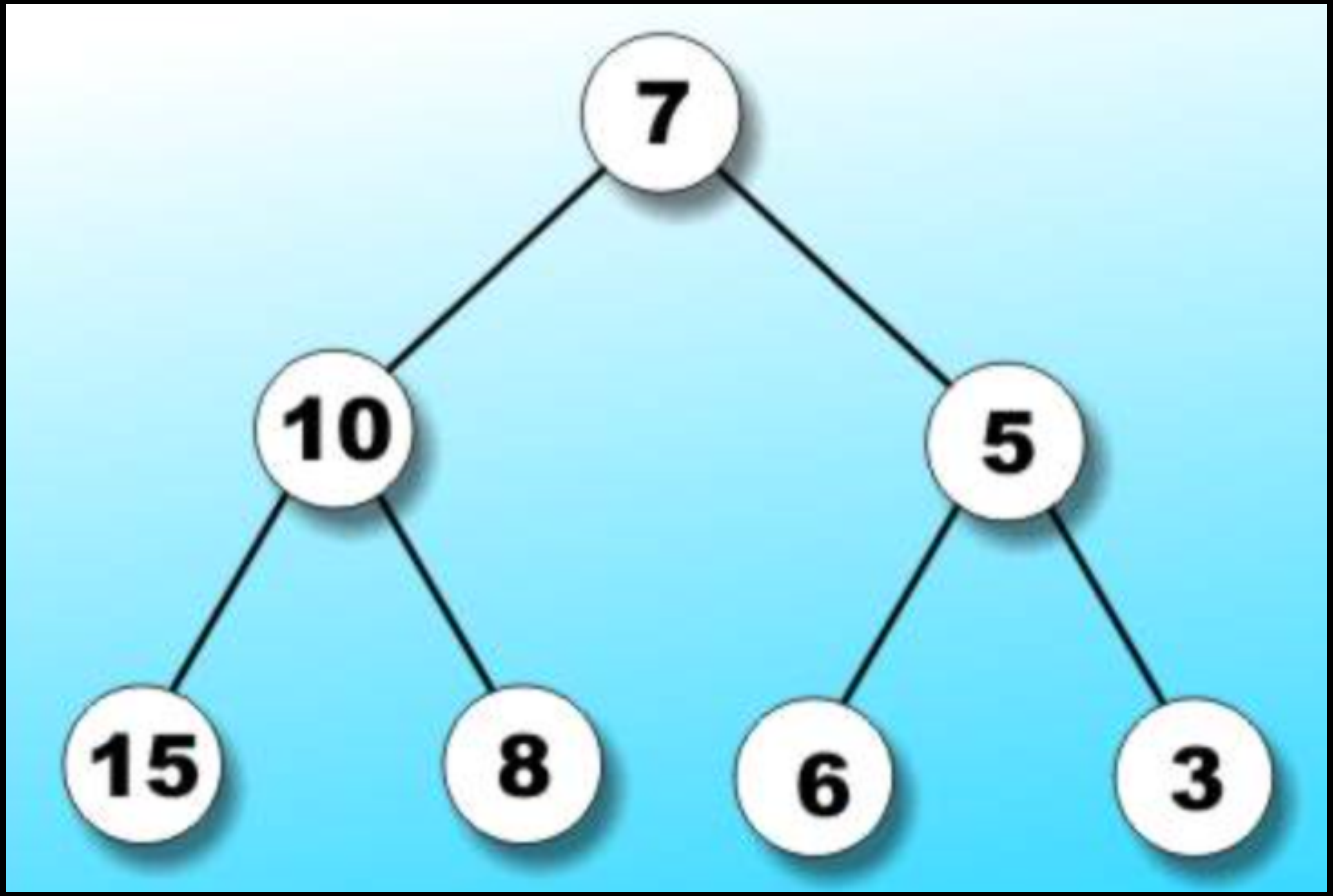
- A binary search tree is a binary **tree**



- A tree is a data structure consisting of nodes organized as a hierarchy

Binary Search Tree

- A binary search tree is a **binary tree**



- A binary tree is a tree in which each node has at most two descendants
- Each node can have zero,one,or two descendants, but no more than two

Tree terminology

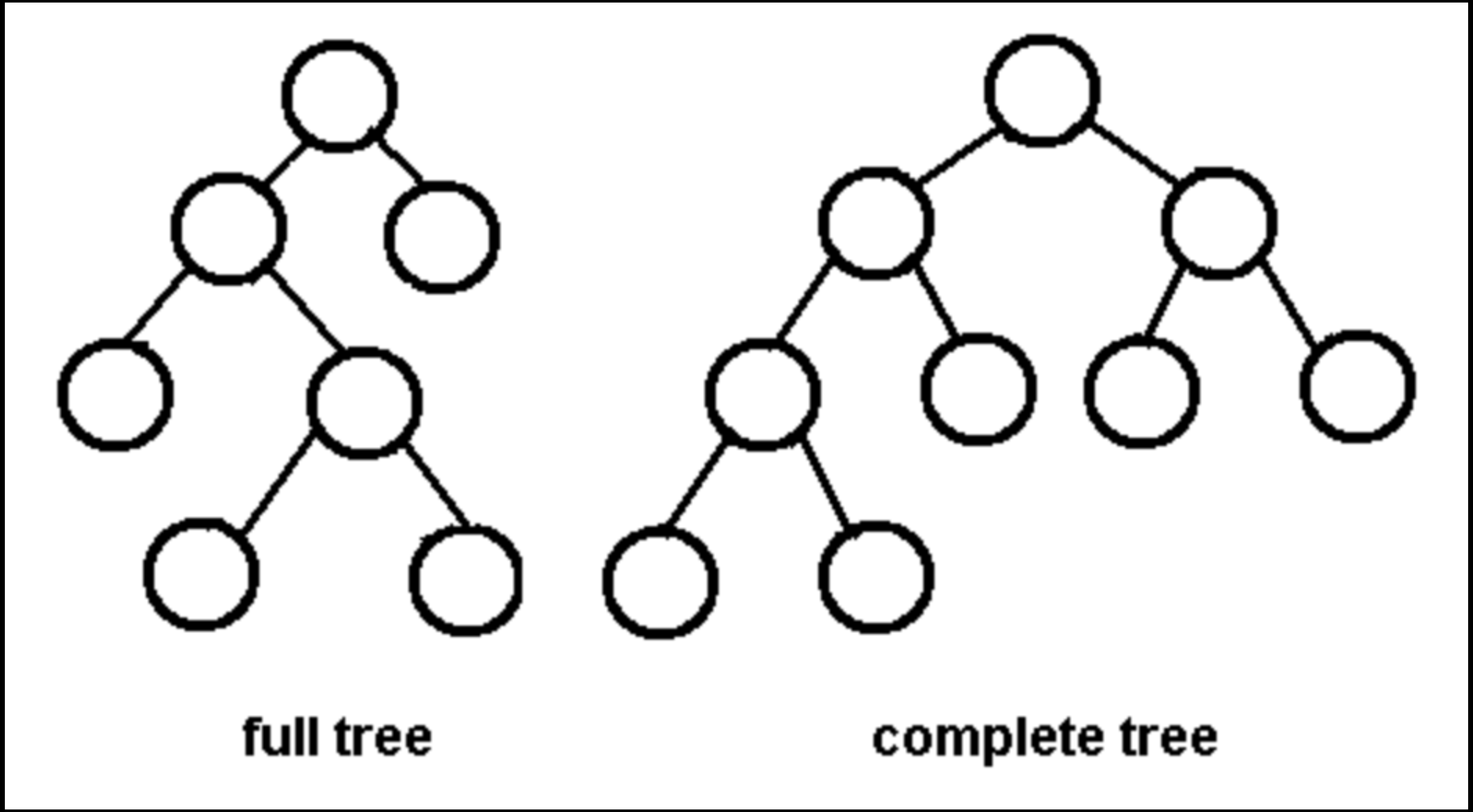
- In a tree data structure, and its different types (like binary), the root node is the topmost node
- In a binary tree, each node contains a left and right reference and a data element
- Every node is connected by a directed edge from exactly one other node. That node is considered the parent.
- The left and right references a node can have are considered that node's children
- Nodes without any children are considered leaves (or sometimes external nodes)
- Nodes with the same parent are considered siblings

Tree terminology cont.

- The depth of a node is the number of edges from the root to the node
- The height of a node is the number of edges from the node to its deepest leaf
- The height of a tree is the height of its root node

Binary Tree types

- A full binary tree is a binary tree in which each node has exactly zero or two children
- A complete binary tree is a binary tree which is completely filled, with the possibly exception of the bottom level, which is filled from left to right



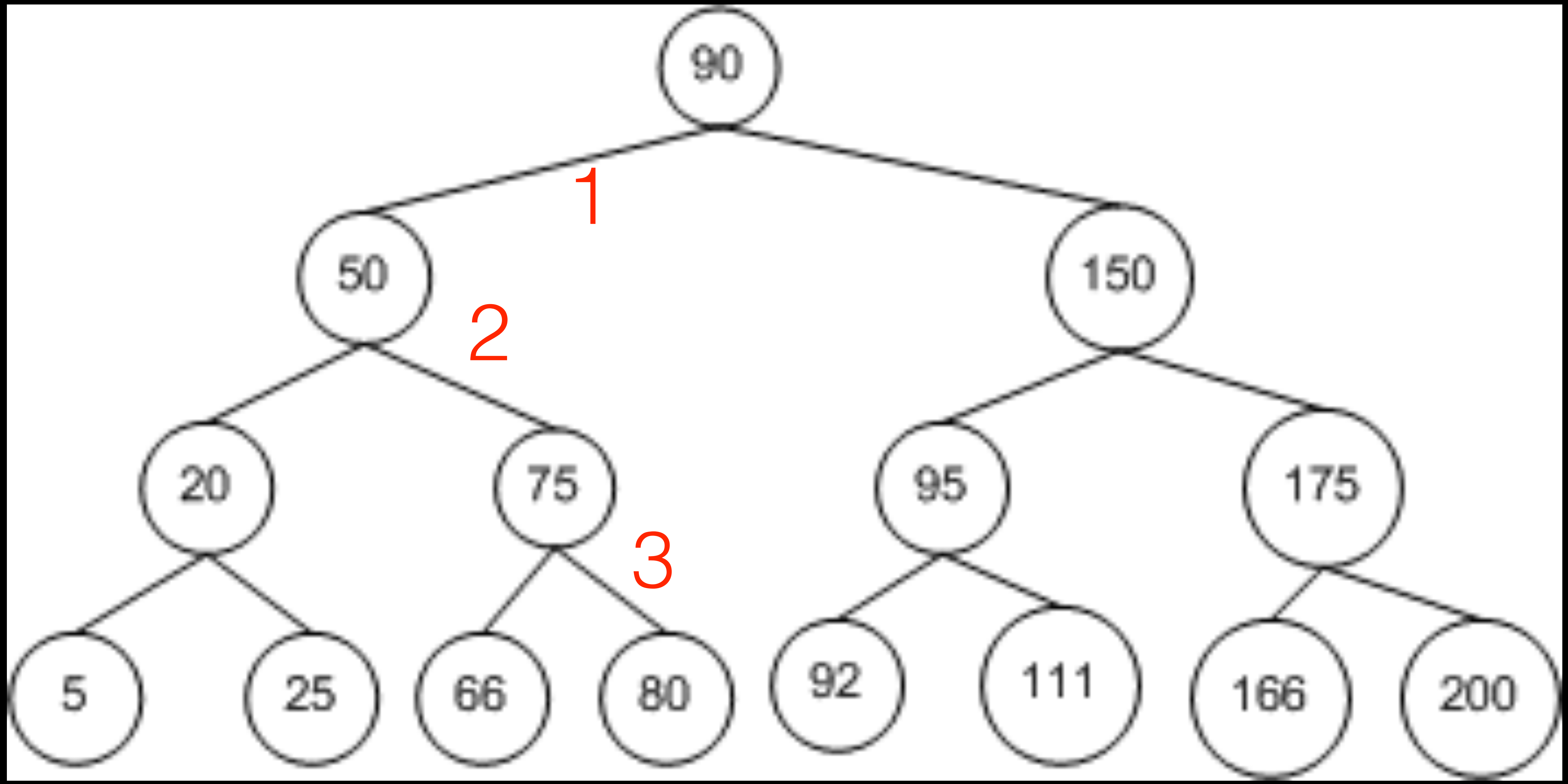
Complete Tree

- A complete tree is a very awesome tree, because it provides the best ratio between number of nodes and height
- The height h of a complete binary tree with N nodes is at most O(log n)
- Thats amazing!

n = 15

log(2) * 15 = 3.9

Which means you only need
to traverse 3 edges to get to the most bottom leaf

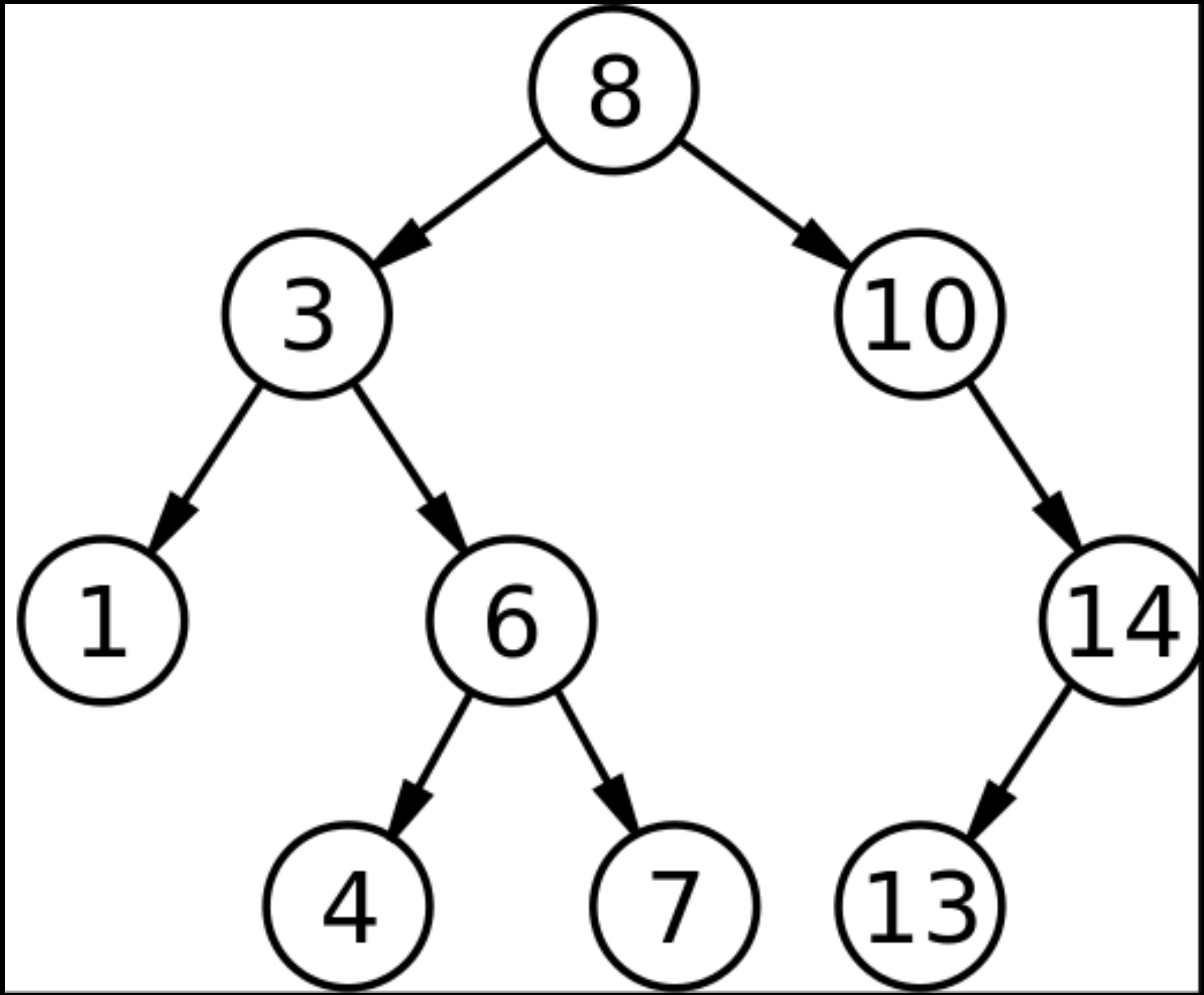


Binary Search Tree

Back to that definition:

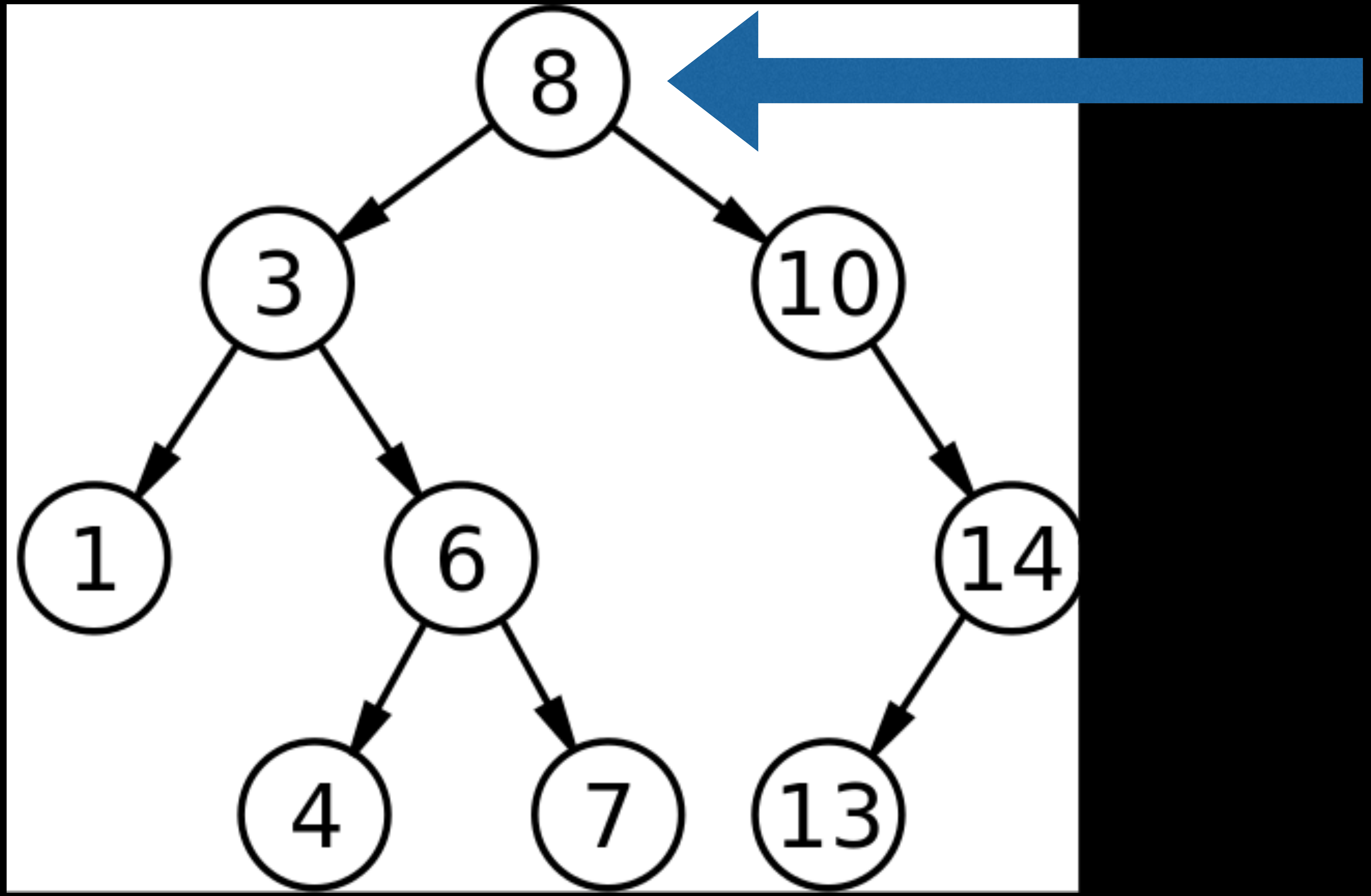
- “A binary search tree is a binary tree where each node has a comparable key and satisfies the restriction that **the key in any node is larger than all keys in all nodes in that node’s left subtree, and smaller than the keys in all nodes in that nodes right subtree**”

Binary Search Tree



Binary Search Tree

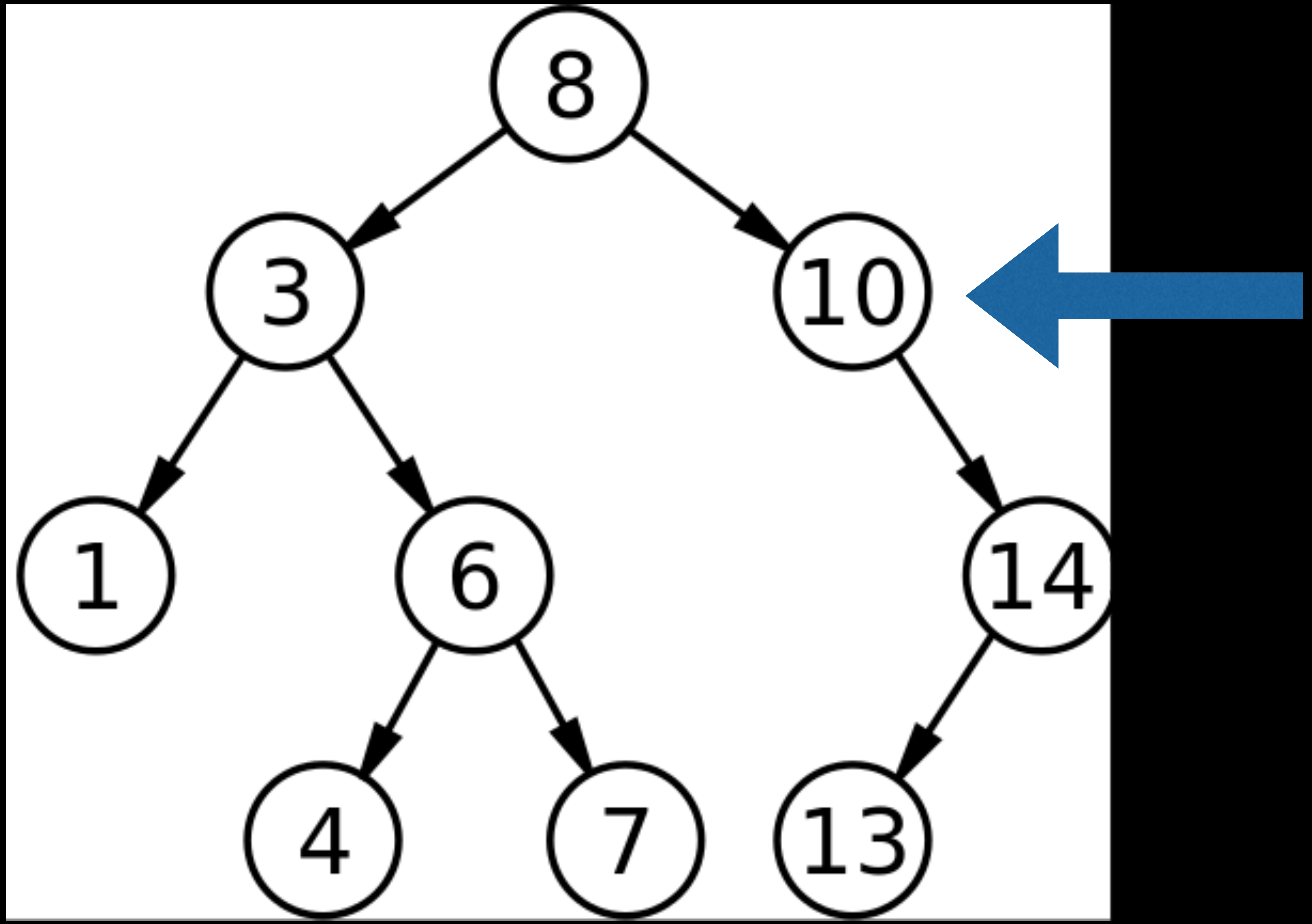
What if wanted to insert 15 into this tree?



We first try to insert it at the root, 15 is larger than 8, and 8 already has a right child, so…

Binary Search Tree

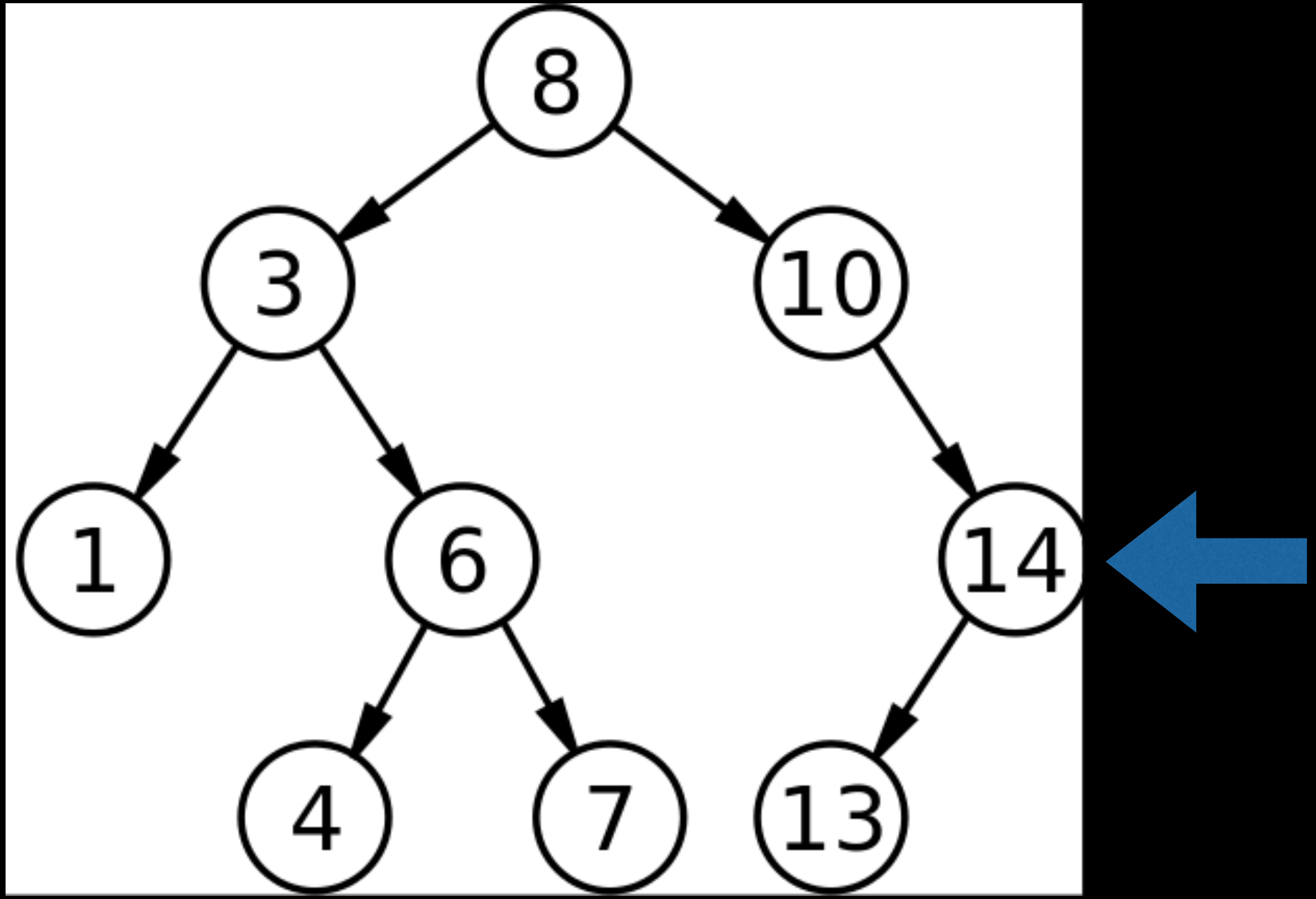
What if wanted to insert 15 into this tree?



We now ask 10, 15 is greater than 10, and 10 node already has a right child, so….

Binary Search Tree

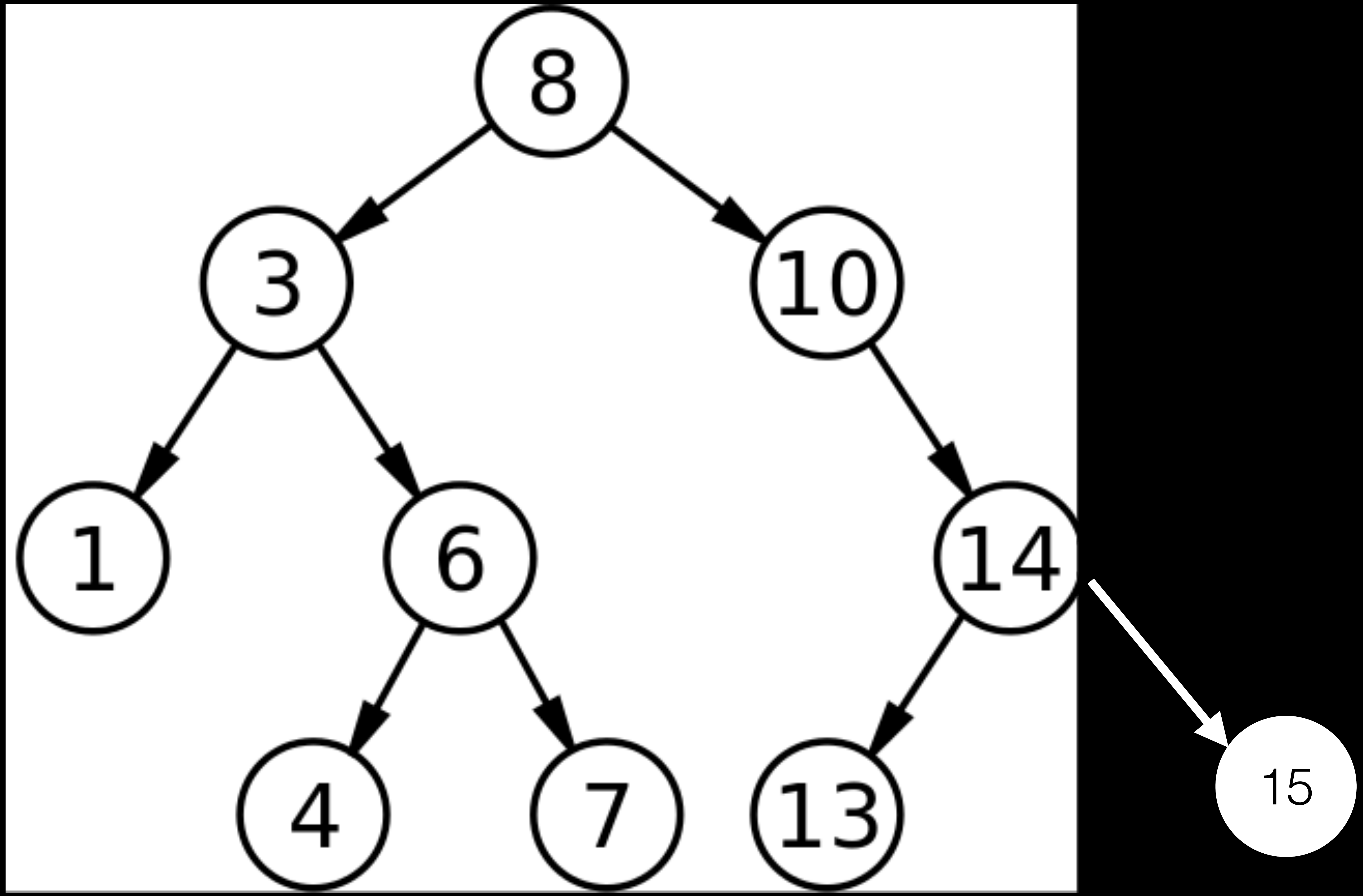
What if wanted to insert 15 into this tree?



We now ask the 14 node. 15 is greater than 14, and 14 has no right child node!

Binary Search Tree

What if wanted to insert 15 into this tree?



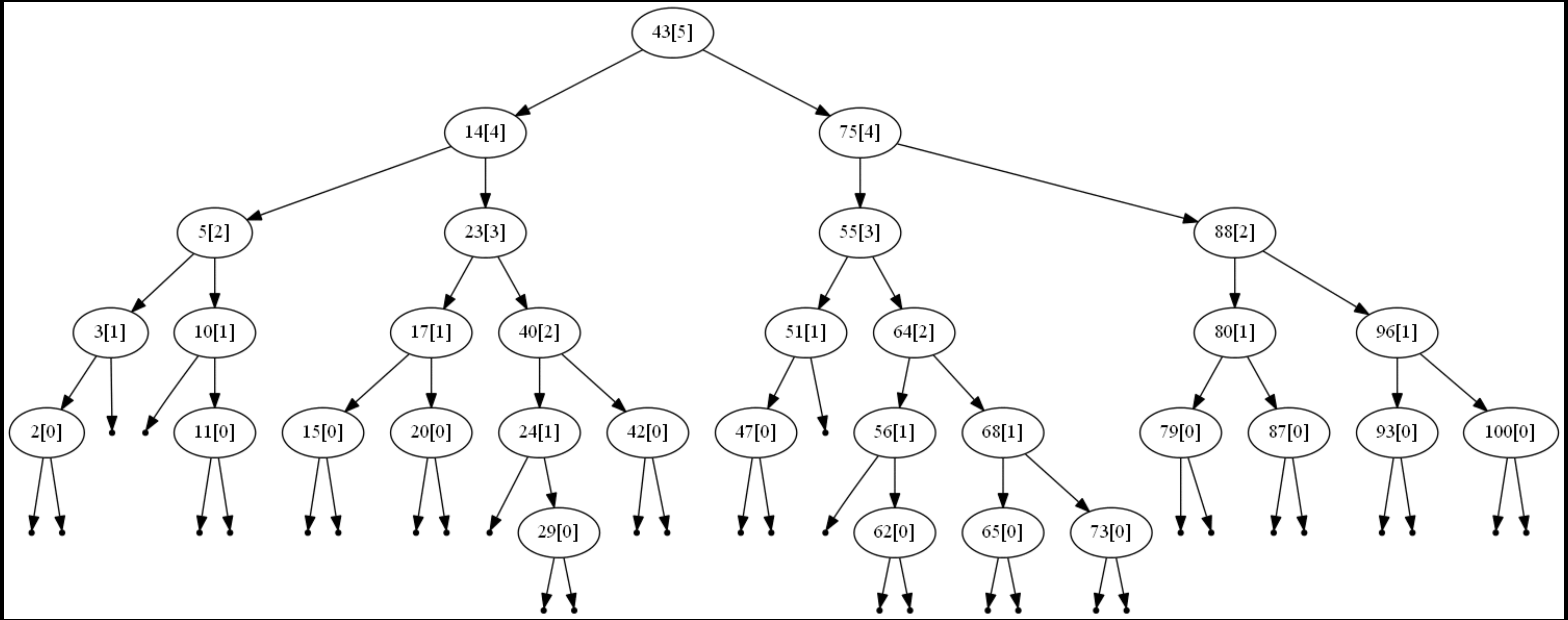
15 now takes it right rightful place in the BST

So why is a binary search tree such a commonly used data structure?

- Because it rocks.
- Specifically, it rocks for searching.
- BST’s have a BigO of O(log n) for search, thats great!
- Lets take a look

Search

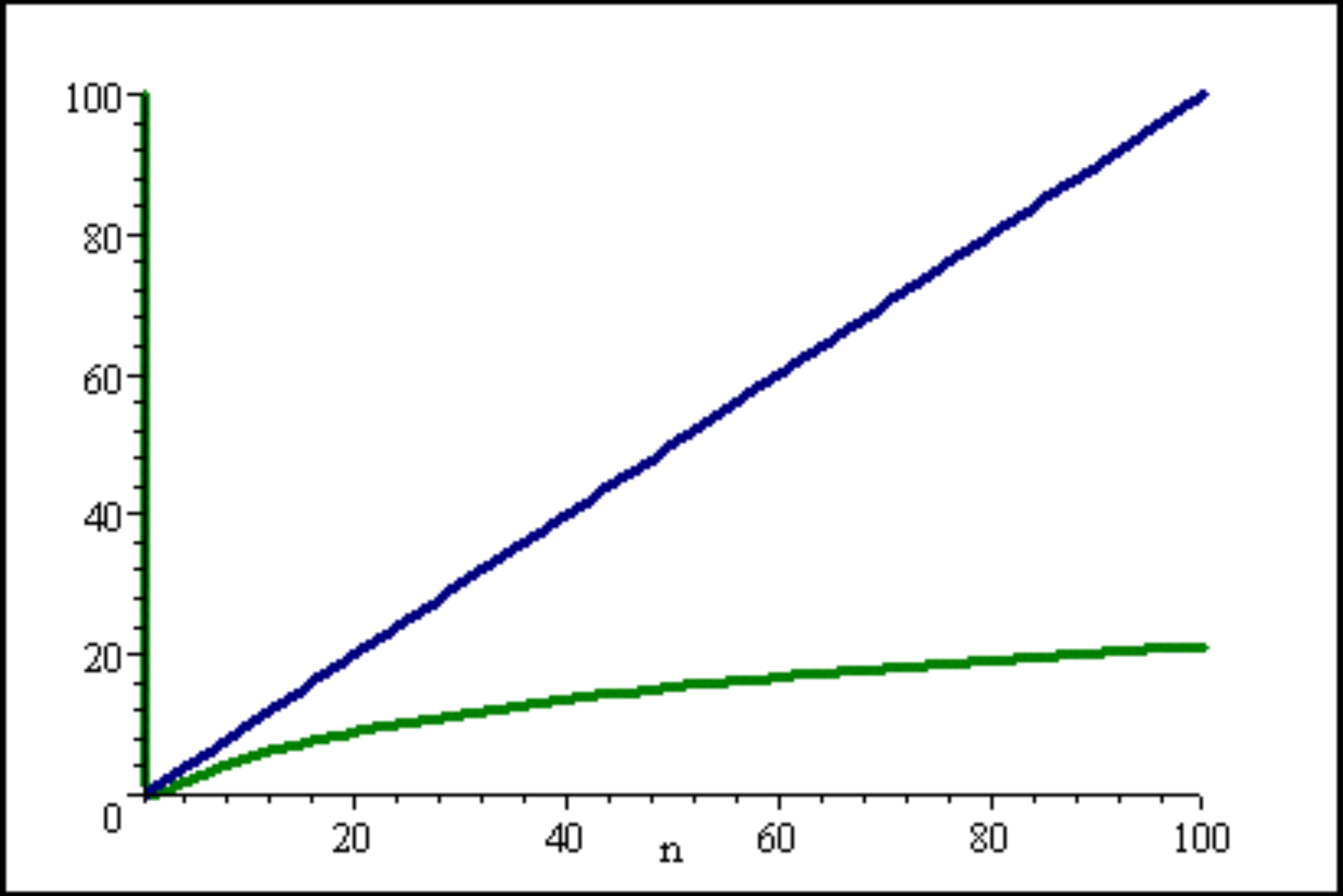
Heres a large looking BST. It has 32 nodes.



log(2) * 32 = 5. At most if would take us 5 traversals, or steps, through this BST to find what we are looking for.

O(log(n)) vs O(n)

Look how slowly log (n) rises compared to the blue (n)



- If your input size n is one million, the big o of n is 1000000, while log(n) is 19.93
- lol

BST and log(n)

- So a Binary search tree with 1 million nodes takes at most 20 steps to find what you are looking for
- Doubling the node count to 2 million, only increases the necessary steps by 1. Thats so good.

Implementing a BST

- When implementing a BST, you can do it a number of different ways.
- I prefer to do it by creating both a binary search tree class, and then a node class to go along with it.

BinarySearchTree Class

- Your Binary search tree class can be pretty simple
- The only property it needs is a node property called root. This is your root node!
- Just like a linked list, as long as you have a strong reference to the root node, the rest of the tree stays alive
- And like a linked list, you will need to come up with ways to traverse the tree.
- The minimum functionality your BST needs is to add a value, find a value, and delete a value

Demo