

iOS Dev Accelerator

Week 8 Day 1

- Container View Controllers
- Trees and BSTs
- Manual Memory Management

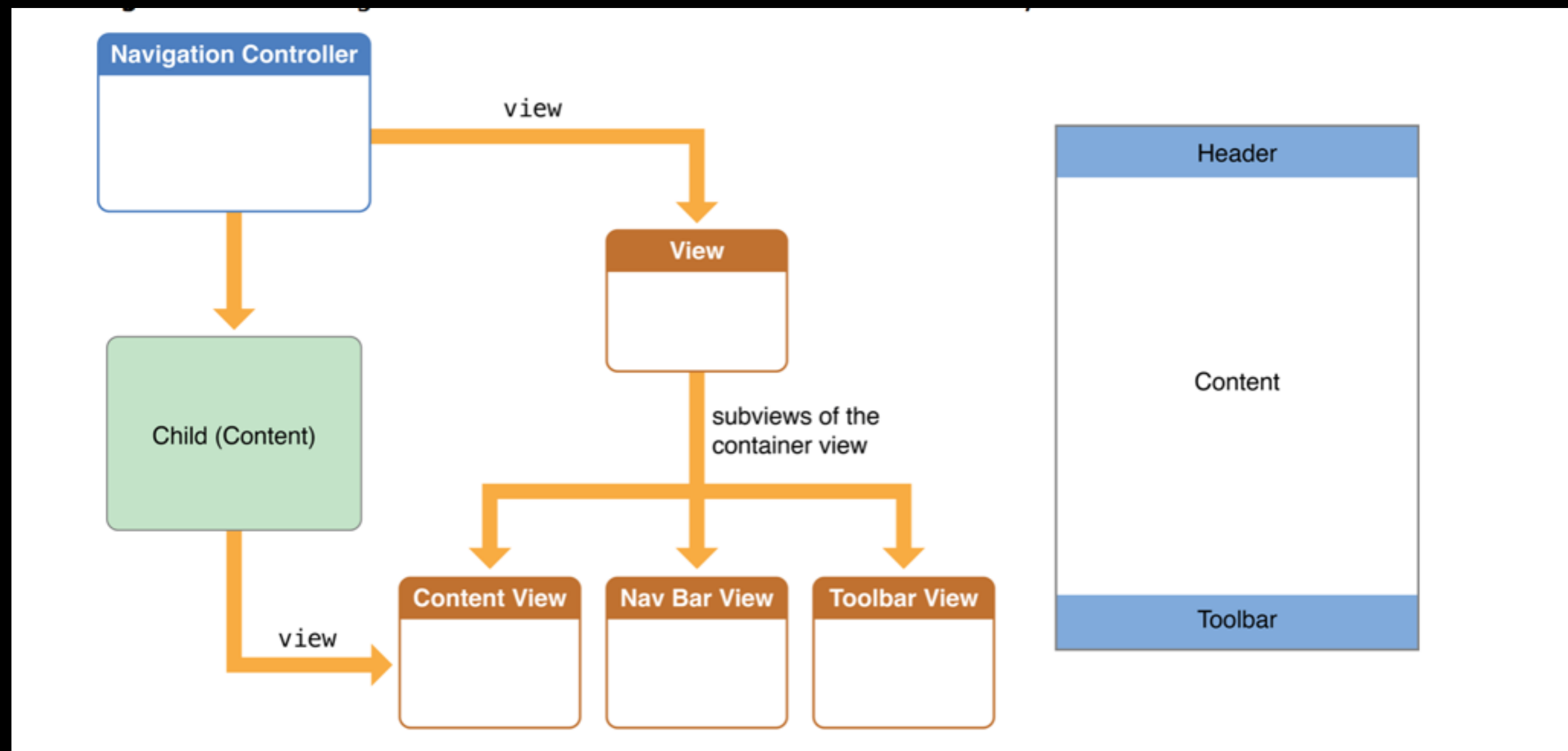
Container View Controllers

Container View Controllers

- All of the view controllers you have created so far are considered 'Content View Controllers'
- Container View Controllers are similar to Content View Controllers, except they manage parent-child relationships between the Container VC, the parent, and its Content VC(s), the children.
- The Container VC is in charge of displaying its children VC's views.

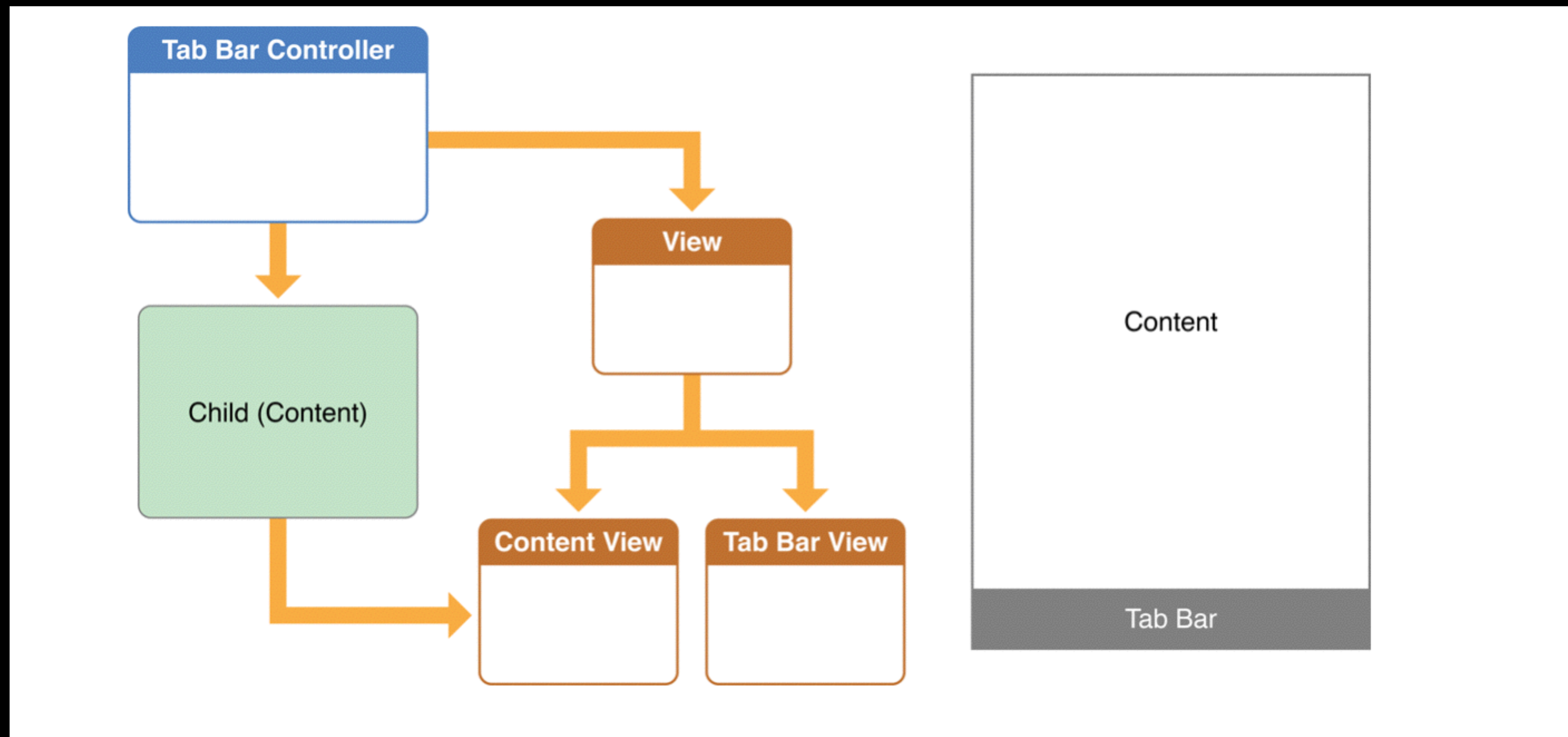
Container View Controllers

- An example of a Container View Controller you have used is the Navigation Controller:



Container View Controllers

- Same Idea with a Tab Bar Controller:



Container View Controllers

- “The goal of implementing a container is to be able to add another view controller’s view as a subview in your container’s view hierarchy”
- When a new VC is added on screen, you can ensure the right events (viewDidLoad, viewWillAppear, etc) are given to all VC’s by associating the new VC as a child of the container VC.

Getting your child on screen

```
func setupChildVC() {  
    //create a new VC  
    var childVC = UIViewController()  
    //tell this VC that we are adding a child VC  
    self.addChildViewController(childVC)  
    //set the child VC's view's frame, in this case it will  
        completely cover the container VC  
    childVC.view.frame = self.view.frame  
    //add the child view to the parent view  
    self.view.addSubview(childVC.view)  
    //notify the child vc he is now on screen  
    childVC.didMoveToParentViewController(self)  
}
```

Taking the child off screen

```
childVC.willMoveToParentViewController(nil)  
childVC.view.removeFromSuperview()  
childVC.removeFromParentViewController()
```


Transitioning your Child VCs

```
//being process of removing old one and adding new one
childVC.willMoveToParentViewController(nil)
self.addChildViewController(nextChildVC)

//set the new child's frame to be off screen to the left, and create an end frame
that is off the screen to the right
nextChildVC.view.frame = CGRectMake(-700, 0, self.view.frame.width, self.view.
frame.height)
var endFrame = CGRectMake(700, 0, childVC.view.frame.width, childVC.view.frame.
height)

//call the transition method on our container view controller
self.transitionFromViewController(childVC, toViewController: nextChildVC,
duration: 0.5, options: UIViewAnimationOptions.TransitionCrossDissolve,
animations: { () -> Void in
//give our new child a frame equal to our containers view, putting him fully
on screen. set our old child's view to the end frame so it slides off to
the right
nextChildVC.view.frame = self.view.frame
childVC.view.frame = endFrame

}) { (success) -> Void in

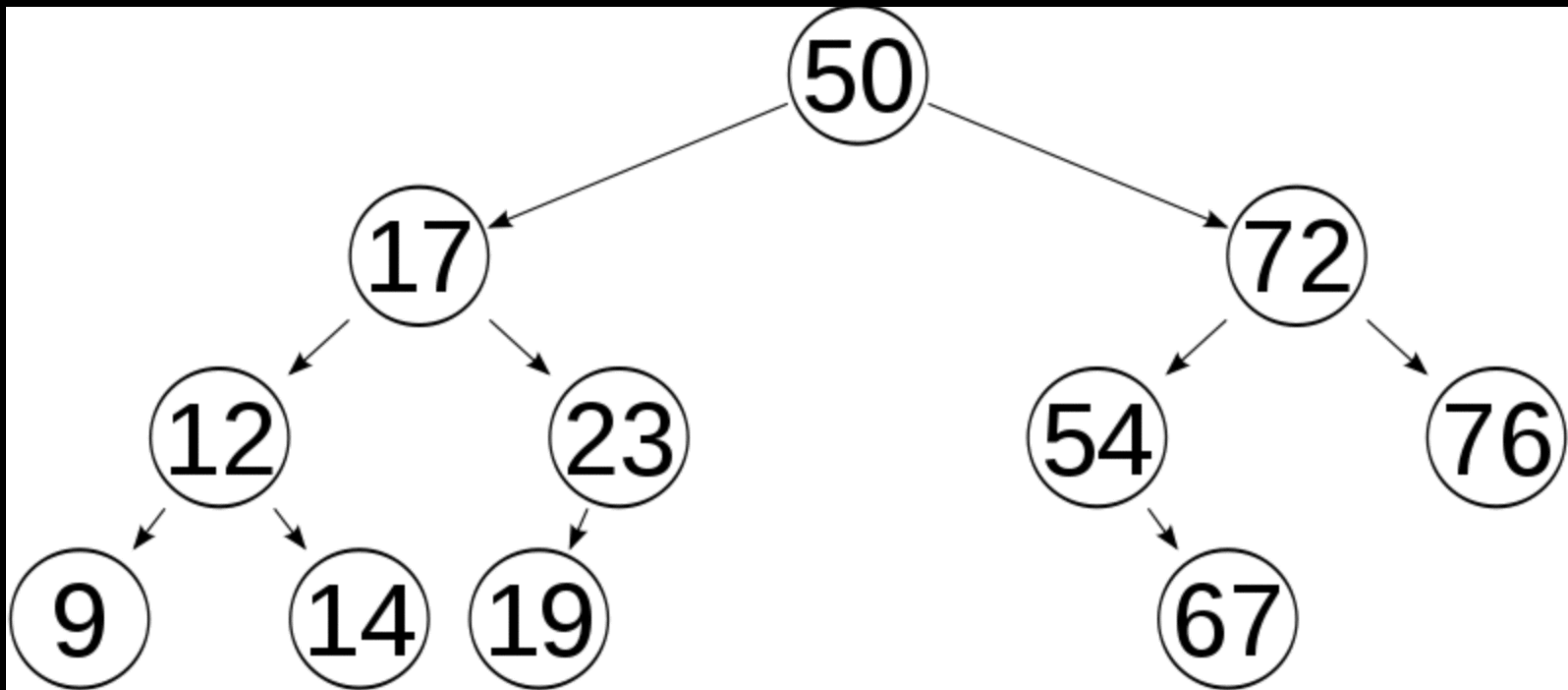
//finish the operation|
childVC.removeFromParentViewController()
nextChildVC.didMoveToParentViewController(self)

}
```

Demo

Binary Search Tree Data Structure

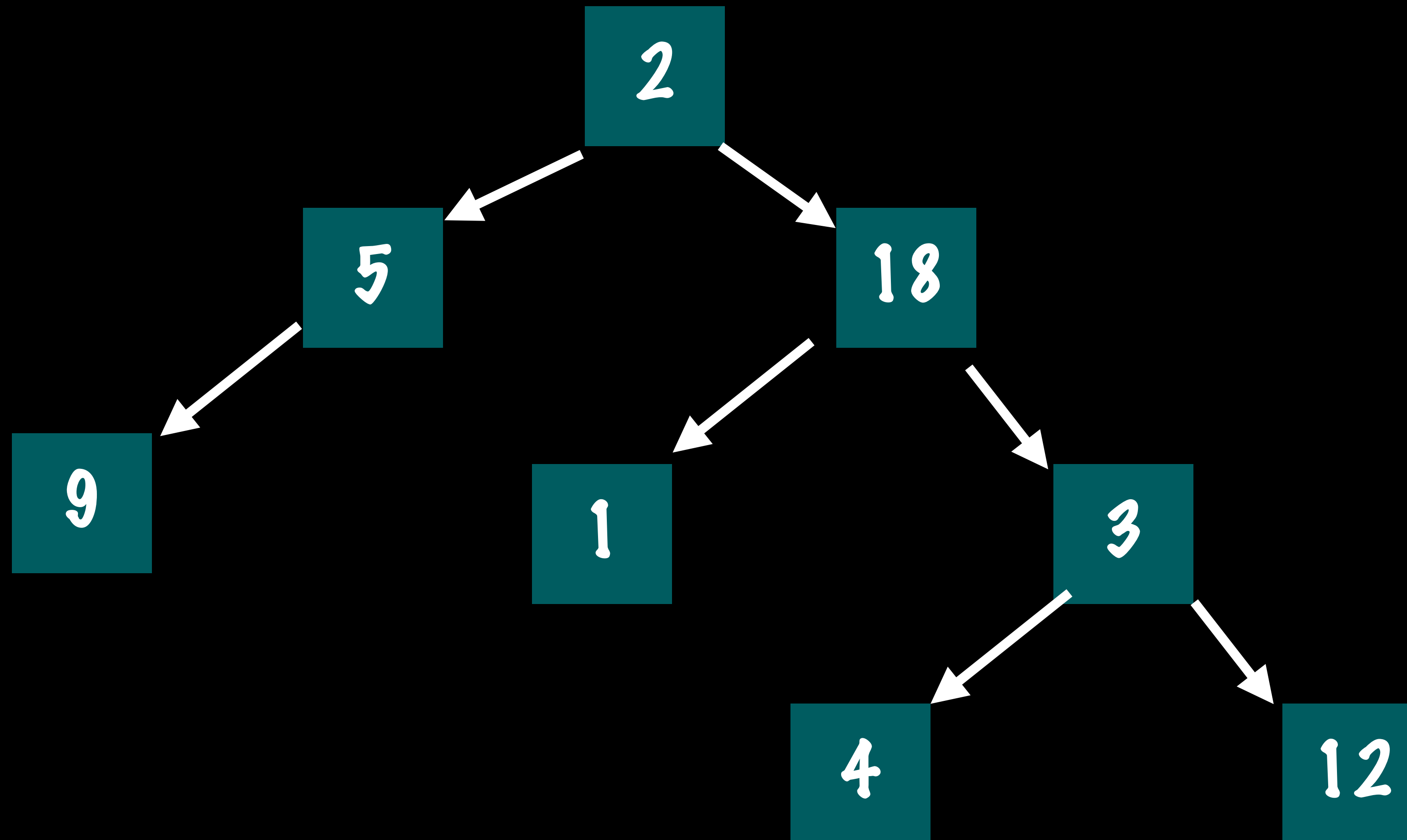
Binary Search Tree



Tree Data Structure

- Hierarchical structure with root node and subtrees and children
- Can be ordered or unordered
- Each element or node in the tree has a value and pointers to zero or more children nodes
- Binary Tree is a specialized type of Tree, where each node has at most **two** children.

Tree Data Structure



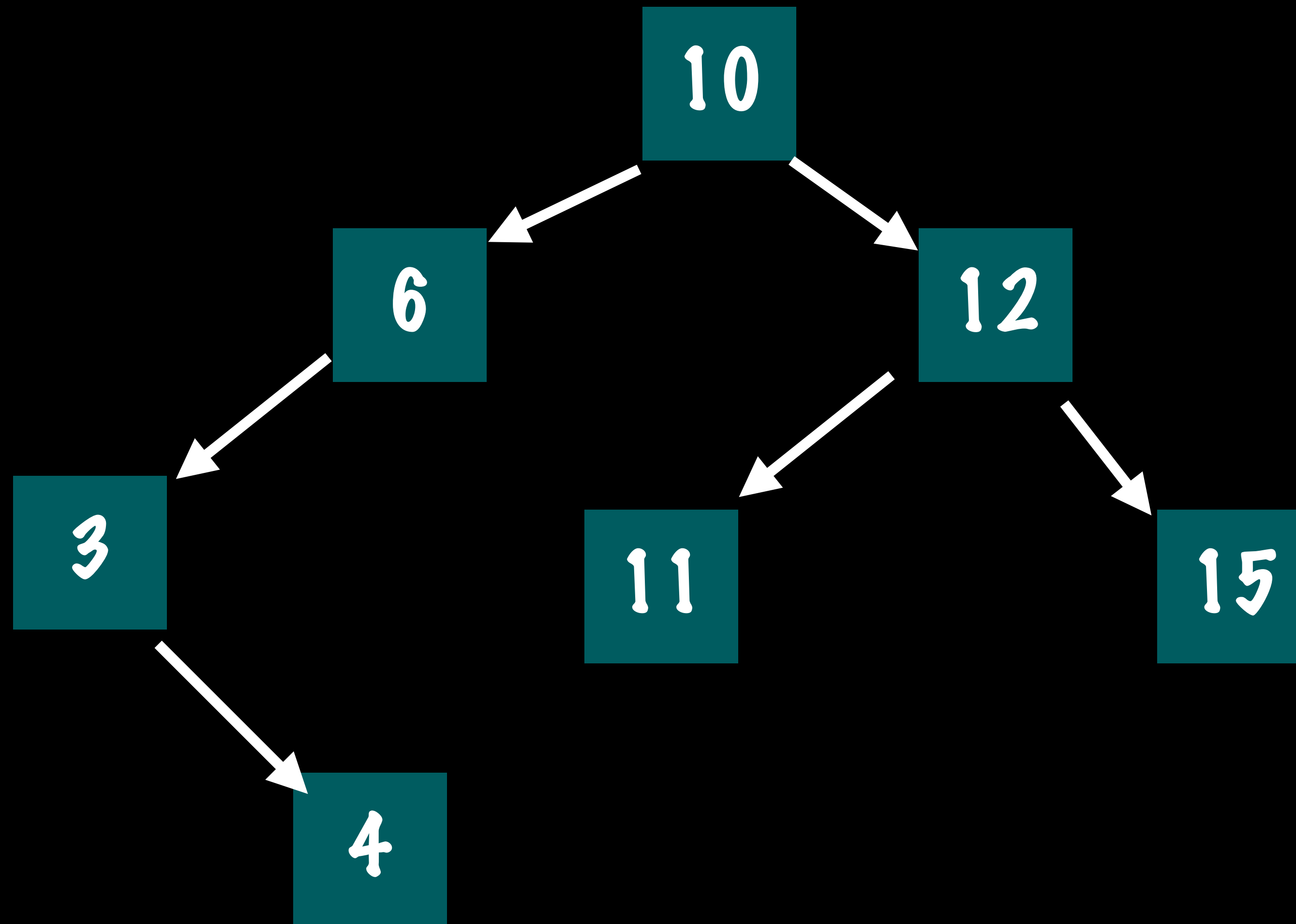
Binary Search Tree (BST)

- An ordered or sorted binary tree
- Each node has a comparable key or value
- The key in any node is larger than the keys in all nodes in that node's left sub-tree
- The key in any node is smaller than the keys in all nodes in that node's right sub-tree

Binary Search Tree Properties

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
- Each node can have up to two successor nodes.
- There must be no duplicate nodes.
- A unique path exists from the root to every other node.

Binary Search Tree



Demo

Memory Management

- “Application memory management is the process of allocating memory during your program’s runtime, using it, and freeing it when you are done with it”
- Objective-C has two methods of application memory management:
 - Manual retain-release, or MRR, allows you to explicitly manage memory by keeping track of objects you own using reference counting.
 - Automatic Reference Counting, or ARC, uses the same reference counting that MRR does, but it inserts the appropriate memory management method calls for you at compile-time. Swift uses ARC as well.
 - The big difference between ARC and traditional garbage collectors is that ARC doesn't break memory cycles for you. ARC provides weak references to accomplish this.

Memory Problems

- There are two general problems that good memory practices will help you avoid:
 - Freeing or overwriting data that is still in use, causing corruption and crashes (dangling pointer)
 - Not freeing data that is no longer in use, which is a memory leak. Leaks needlessly increase the memory footprint of your app, often times to levels that will cause your app to be terminated.

Object Ownership

- Memory management is based on object ownership.
- An object can have many owners.
- As long as an object has one owner, it is kept alive.
- If an object has no owners, it is destroyed by the run time.

Being the owner

- You own any object you create, as long you create it using alloc, new, copy, or mutable copy.
- You can own any object by calling retain on it.
- You normally only call retain in the implementation of your init or an accessor method, or to prevent an object from being invalidated because of some other operation.

```
- (NSString *)fullName {
    NSString *string = [NSString stringWithFormat:@"%s@ %s@",
                        self.firstName, self.lastName];

    return string;
}
```

We don't own string because we didn't use alloc,new,copy

```
{
    Person *aPerson = [[Person alloc] init];
    // ...
    NSString *name = aPerson.fullName;
    // ...
    [aPerson release];
}
```

We own aPerson because we used alloc, so we had to release it when done with it

Autorelease

- You can use autorelease when you need to send a deferred release message.
- Typically when returning an object from a method.
- You need autorelease because regular release would release the object before you could return it from the method:

```
- (NSString *)fullName {  
    NSString *string = [[[NSString alloc] initWithFormat:@"%@" "%@",  
                                                                self.firstName, self.lastName] autorelease];  
  
    return string;  
}
```

Dealloc

- NSObject provides a method called dealloc.
- Dealloc is invoked automatically when an object has no owners and its memory is reclaimed.
- The role of dealloc is to free the objects own memory, and to dispose of any other objects or resources its holding onto, like properties.
- You never call dealloc directly, the system calls it for you.

Dealloc Example

```
@interface Person : NSObject
@property (retain) NSString *firstName;
@property (retain) NSString *lastName;
@property (assign, readonly) NSString *fullName;
@end

@implementation Person
// ...

- (void)dealloc
{
    [_firstName release];
    [_lastName release];
    [super dealloc];
}
@end
```


Accessor methods and MM

- When your class has a property that's an object, you need to make sure that object isn't released while your class is still using it.
- You need to claim ownership for this. And then you need to relinquish ownership when appropriate.
- Harness the power of accessor methods to make this simple and easy.

Accessor methods and MM

```
- (NSNumber *)count {  
    return _count;  
}
```

getter can just return the object

```
- (void)setCount:(NSNumber *)newCount {  
    [newCount retain];  
    [_count release];  
    // Make the new assignment.  
    _count = newCount;  
}
```

setter must release the old object and
retain the new

When not to use Accessor Methods

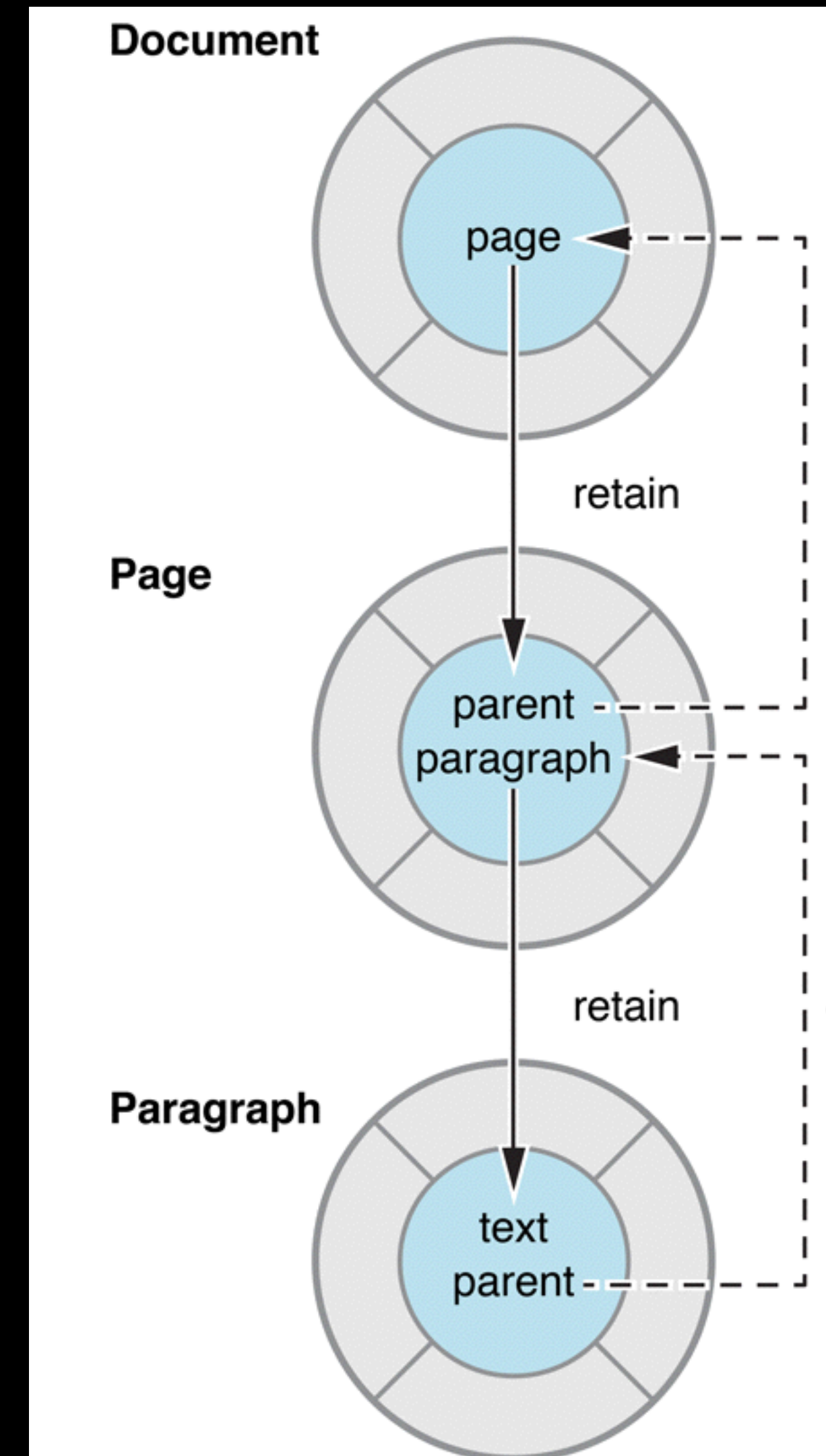
- Don't use the accessor methods in your init methods and dealloc. This eliminates the chance of a bug related to a setter executing or accessing code that hasn't been setup yet.

```
- init {  
    self = [super init];  
    if (self) {  
        _count = [[NSNumber alloc] initWithInteger:0];  
    }  
    return self;  
}
```

```
- (void)dealloc {  
    [_count release];  
    [super dealloc];  
}
```

Retain Cycles

- Retaining an object creates a strong reference to that object.
- An object cannot be dealloc'd until all of its strong references are released.
- A retain cycle happens when two objects have strong references to each other.



Use weak references to avoid retain cycles

- A weak reference is a non-owning relationship where the source object does not retain the object to which it has a reference.
- Cocoa conventions dictate that parent objects should have strong references to their children, and children have weak references to their parents.
- Examples of weak references in Cocoa: data sources, delegates, and notification observers.
- That's why you need to specifically unregister for notifications when your object is about to be released, since the notification center only keeps weak references to objects who sign up for notifications. If you didn't unregister, the notification center may try to deliver a notification to a released object. CRASH

Keeping objects alive while in use

- Objects can be released at any time.
- Like when an object's parent is released, and the parent was the only owner of the child, the child is released too.
- Collections own the objects they contain. When an object is removed from a collection like an array or dictionary, if the collection was the only owner then the object is released.
- So when you really need to keep an object alive while in use, call retain on it, execute your code, and then release it.

Retain count

- Ownership is implemented through something called reference counting.
- Every object has a retain count.
- When you create an object it has a retain count of 1.
- When an object has retain called on it, its retain count is incremented by 1.
- When an object has release called on it, its retain count is reduced by 1.
- When you an object has autorelease called on it, its retain count is reduced by 1 at the end of the current autorelease pool block.
- If an object's retain count is reduced to zero, it is released.
- **You should never need to manually access an object's retain count; The value returned is not accurate.**

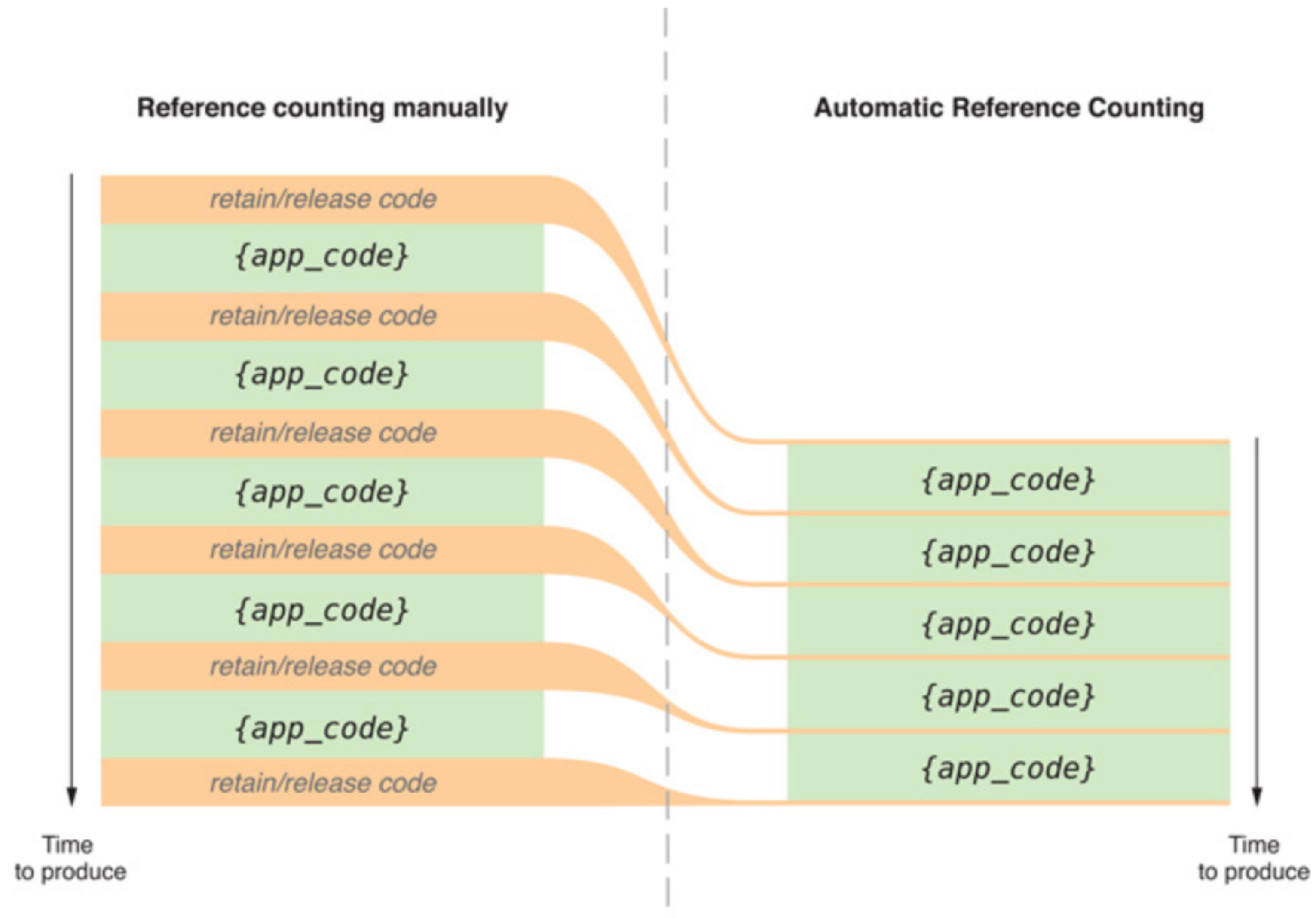
AutoRelease Pool Blocks

- “Autorelease pool blocks provide a mechanism whereby you can relinquish ownership of an object, but avoid the possibility of it being deallocated immediately (such as when you return an object from a method)”
- You usually don't need to create your own.
- 3 common scenarios when you create one:
 - If you are writing a command-line tool.
 - You write a loop that creates many temporary objects.
 - If you spawn a secondary thread.

AutoRelease Pool Blocks

```
for (NSURL *url in urls) {  
  
    @autoreleasepool {  
        NSError *error;  
        NSString *fileContents = [NSString stringWithContentsOfURL:url  
                                encoding:NSUTF8StringEncoding error:&error];  
        /* Process the string, creating and autoreleasing more objects. */  
    }  
}
```

ARC



Automatic Reference Counting (ARC)

- ARC enforces new rules: You cannot invoke `dealloc`, `retain`, `release`, `retainCount`, or `autorelease`.
- ARC introduced weak and strong property attributes. Strong is the default.
- You can explicitly create weak references with `__weak` qualifier while declaring a variable.
- Compiler will insert appropriate memory management calls for you.

ARC

```
// The following declaration is a synonym for: @property(retain) MyClass *myObject;  
@property(strong) MyClass *myObject;  
  
// The following declaration is similar to "@property(assign) MyClass *myObject;"  
// except that if the MyClass instance is deallocated,  
// the property value is set to nil instead of remaining as a dangling pointer.  
@property(weak) MyClass *myObject;
```


Weak self and blocks

- When referring to self inside a block (or closure), best practice is to create a weak reference in self before the block and use that reference instead of self.
- Since blocks and closures capture strong references to all objects accessed in their bodies, if the object represented by self were to get a strong pointer to the block or closure, then we would have a retain cycle.

Blocks and `__weak self`

- Be careful accessing self inside a block, or you could create a retain cycle
- Use `__weak self` and `__strong self` to avoid this
- You may see this referred to as the `@weakify` pattern

<http://aceontech.com/objc/ios/2014/01/10/weakify-a-more-elegant-solution-to-weakself.html>

Blocks and `__weak self`

```
// Create a weak reference to self
__weak typeof(self)weakSelf = self;

[self.context performBlock:^(
    // Create a strong reference to self, based on the previous weak reference.
    // This prevents a direct strong reference so we don't get
    // into a retain cycle to self.
    // Also prevents self from becoming nil half-way through execution.

    __strong typeof(weakSelf)strongSelf = weakSelf;

    // Do something else

    NSError *error;
    [strongSelf.context save:&error];

    // Do something else
)];
```

Demo