The Holy Grail!

A Persistent

Hash-Array-Mapped Trie

for C++

@phil_nash
Developer Advocate

The Holy Grail?

A Persistent

Hash-Array-Mapped Trie

for C++

@phil_nash

Developer Advocate

Trie

ReTrieve

ReTrieve

Tries were first described by René de la Briandais in 1959. The term trie was coined two years later by Edward Fredkin, who pronounces it /ˈtriː/ (as "tree"), after the middle syllable of retrieval. However, other authors pronounce it /ˈtraɪ/ (as "try"), in an attempt to distinguish it verbally from “tree”

Let’s talk about sets

set
multiset

unordered_set
unordered_multiset

map
multimap

unordered_map
unordered_multimap

set
multiset

unordered_set
unordered_multiset

map
multimap

unordered_map
unordered_multimap

set

unordered_set

map

unordered_map

set

unordered_set

set

unordered_set

(sorted) vector

set

unordered_set

`set`

red-black tree

`unordered_set`

hash table

`set`

- red-black tree
- strict weak ordering (<)

`unordered_set`

- hash table
- hashable and equatable (`std::hash`, ==)

`set`

- red-black tree
- strict weak ordering (<)
- O(log n) lookup

`unordered_set`

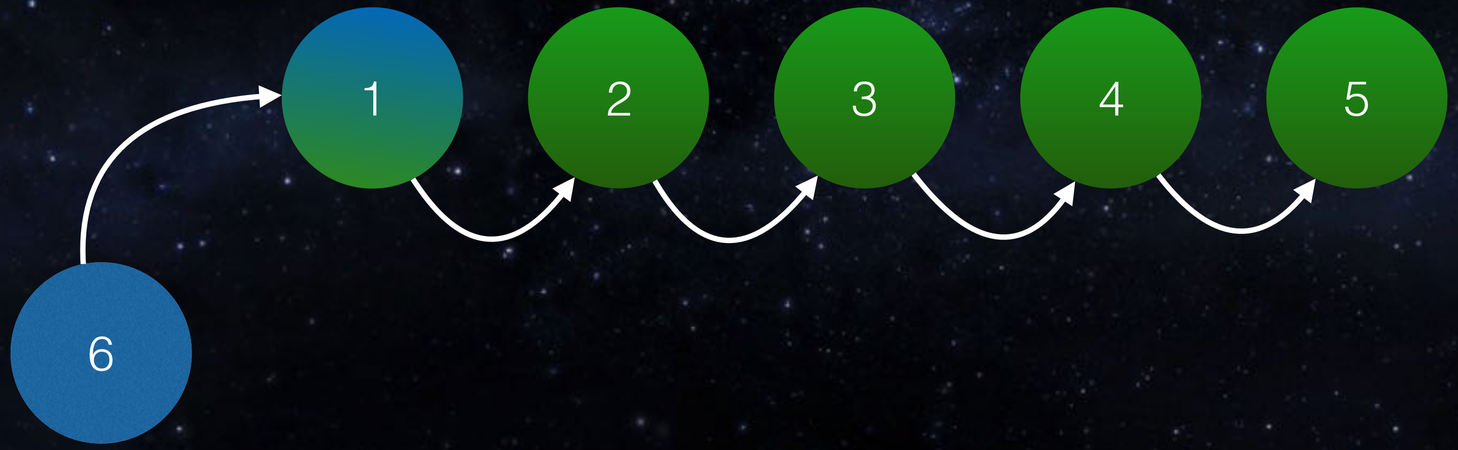
- hash table
- hashable and equatable (`std::hash`, ==)
- O(1) lookup on average -> O(n) worst case

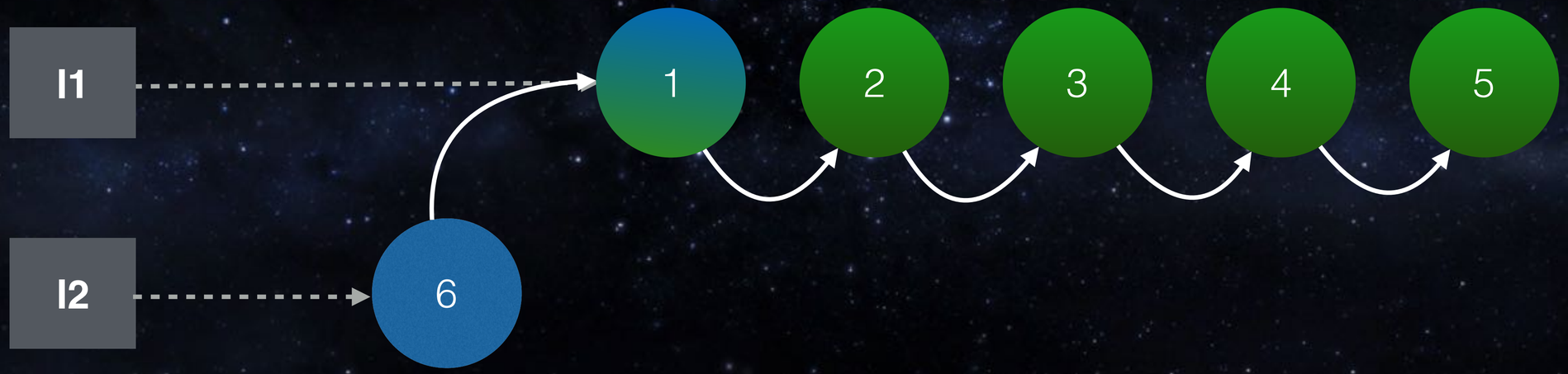
Persistence pays

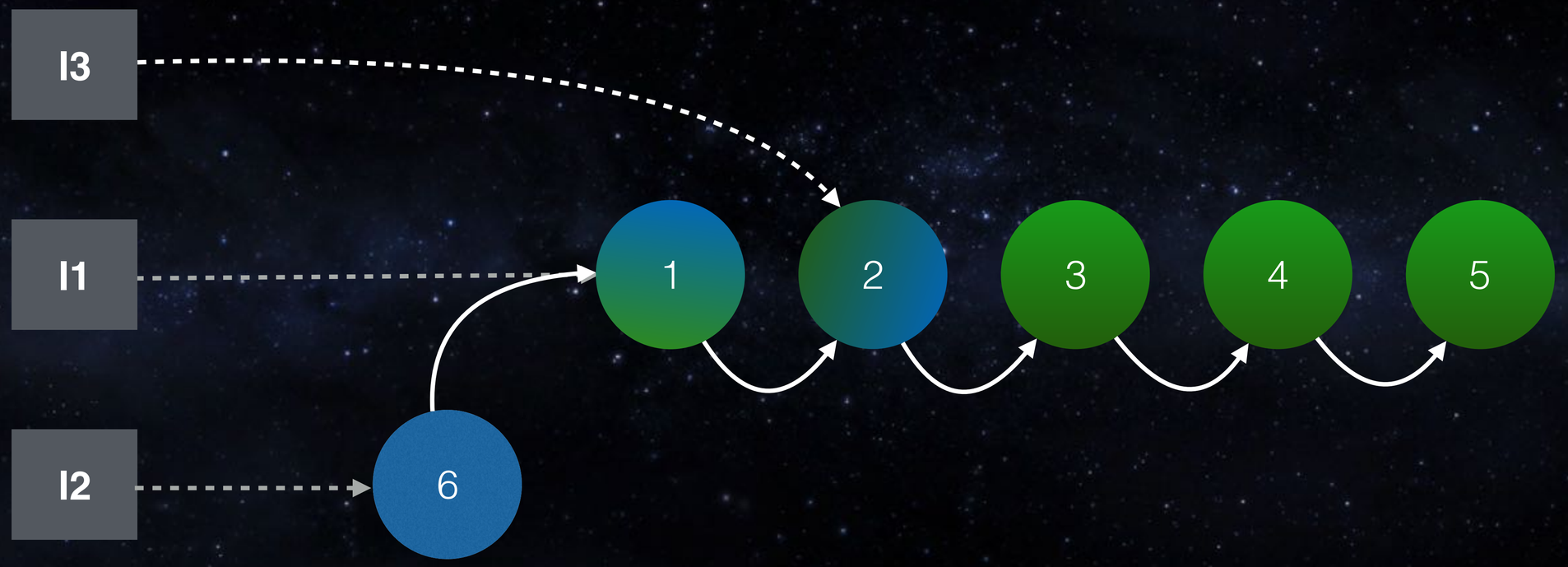
Things I hate:
1 VANDALISM
2 IRONY
3 LISTS.

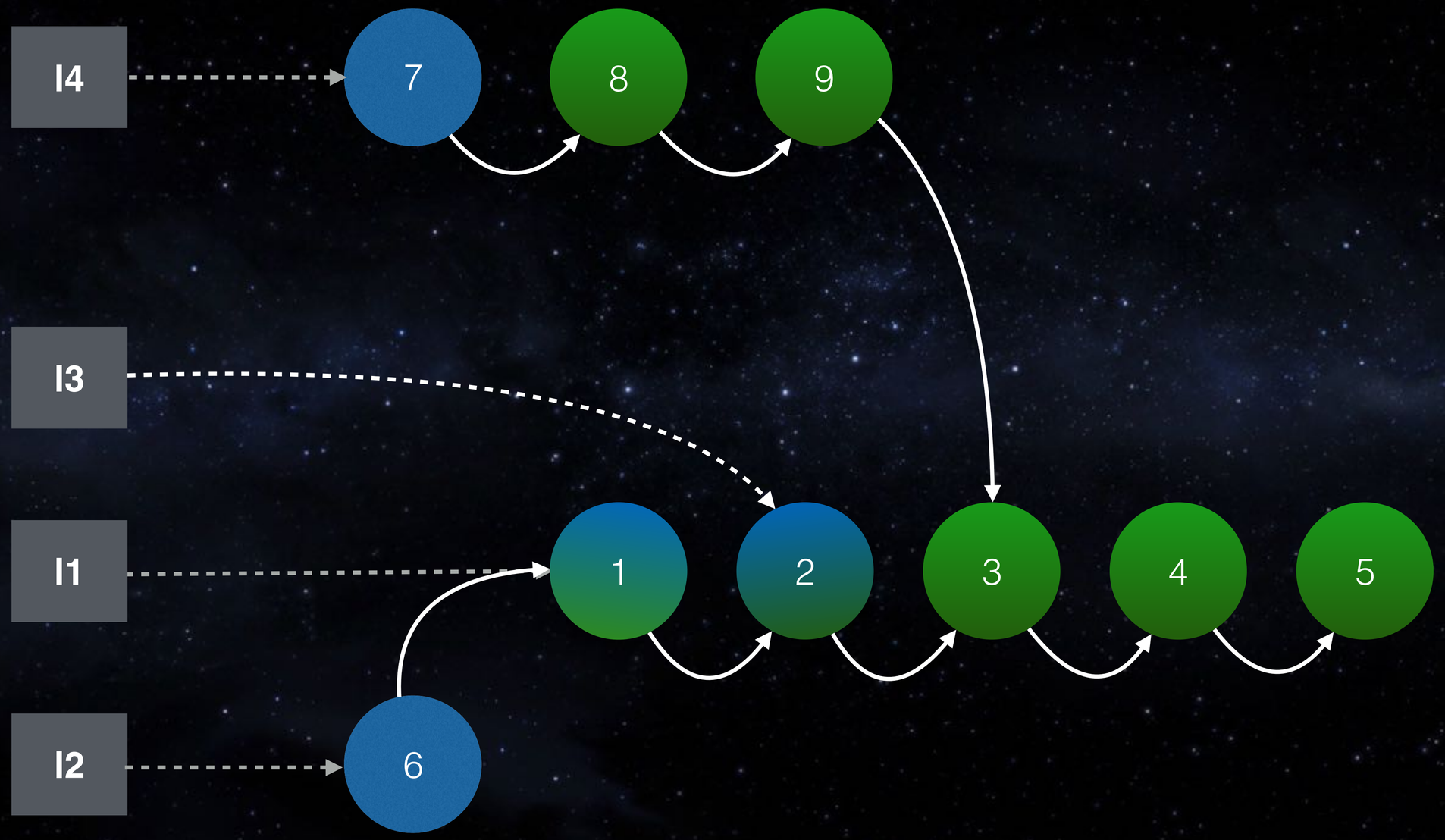
1 2 3 4 5

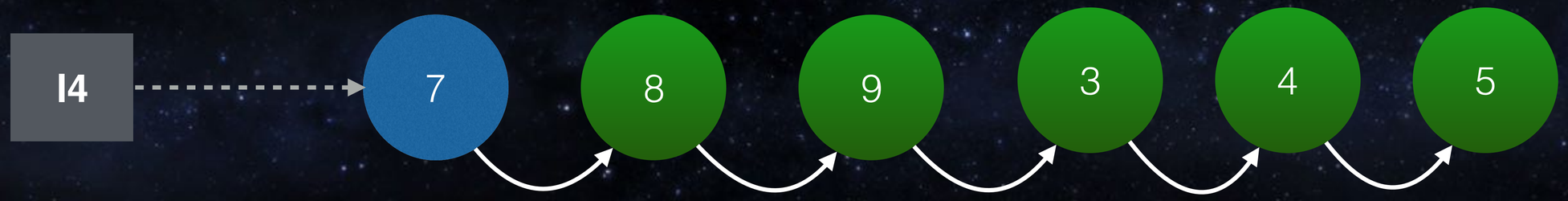
1
2
3
4
5

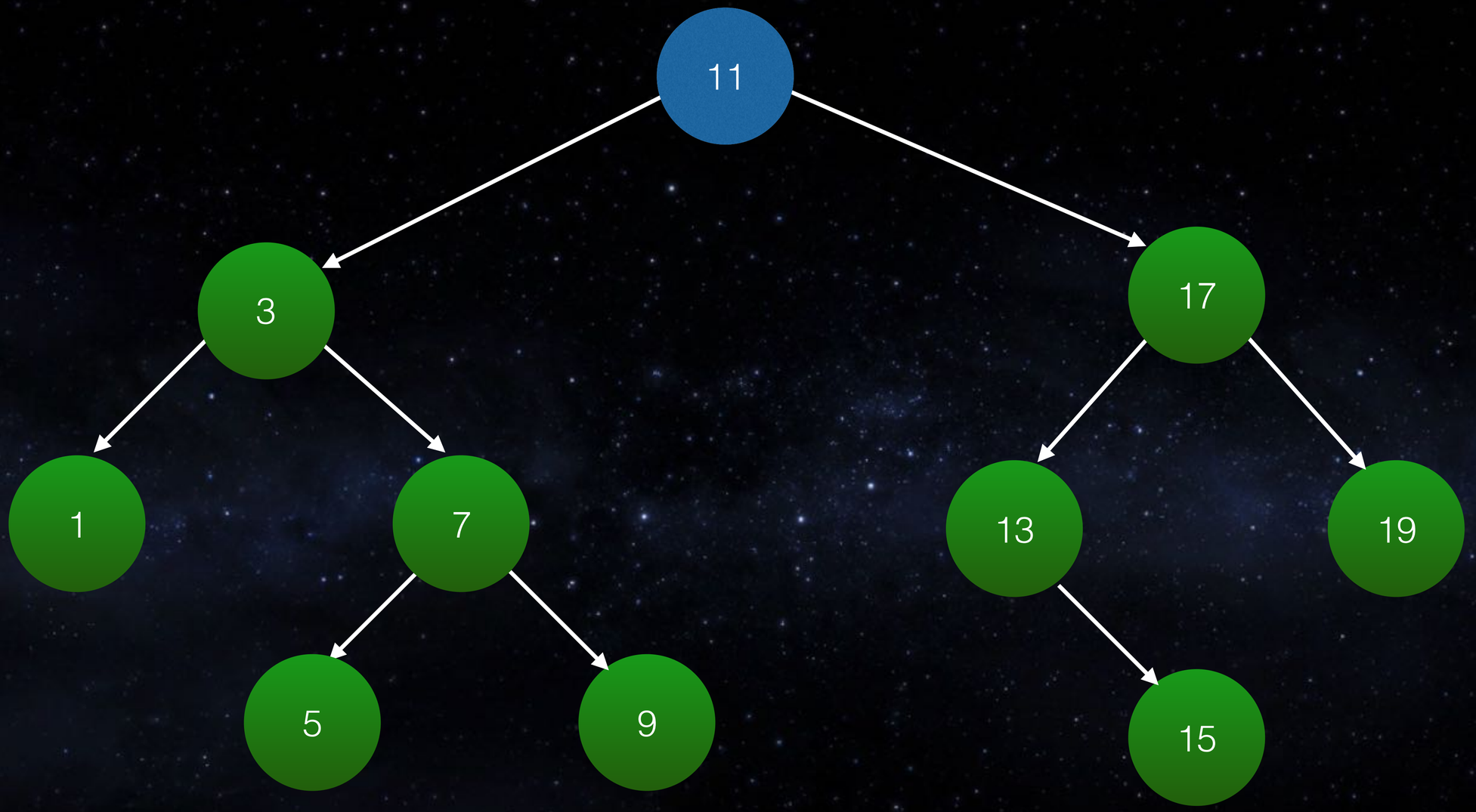

1
2
3
4
5
6

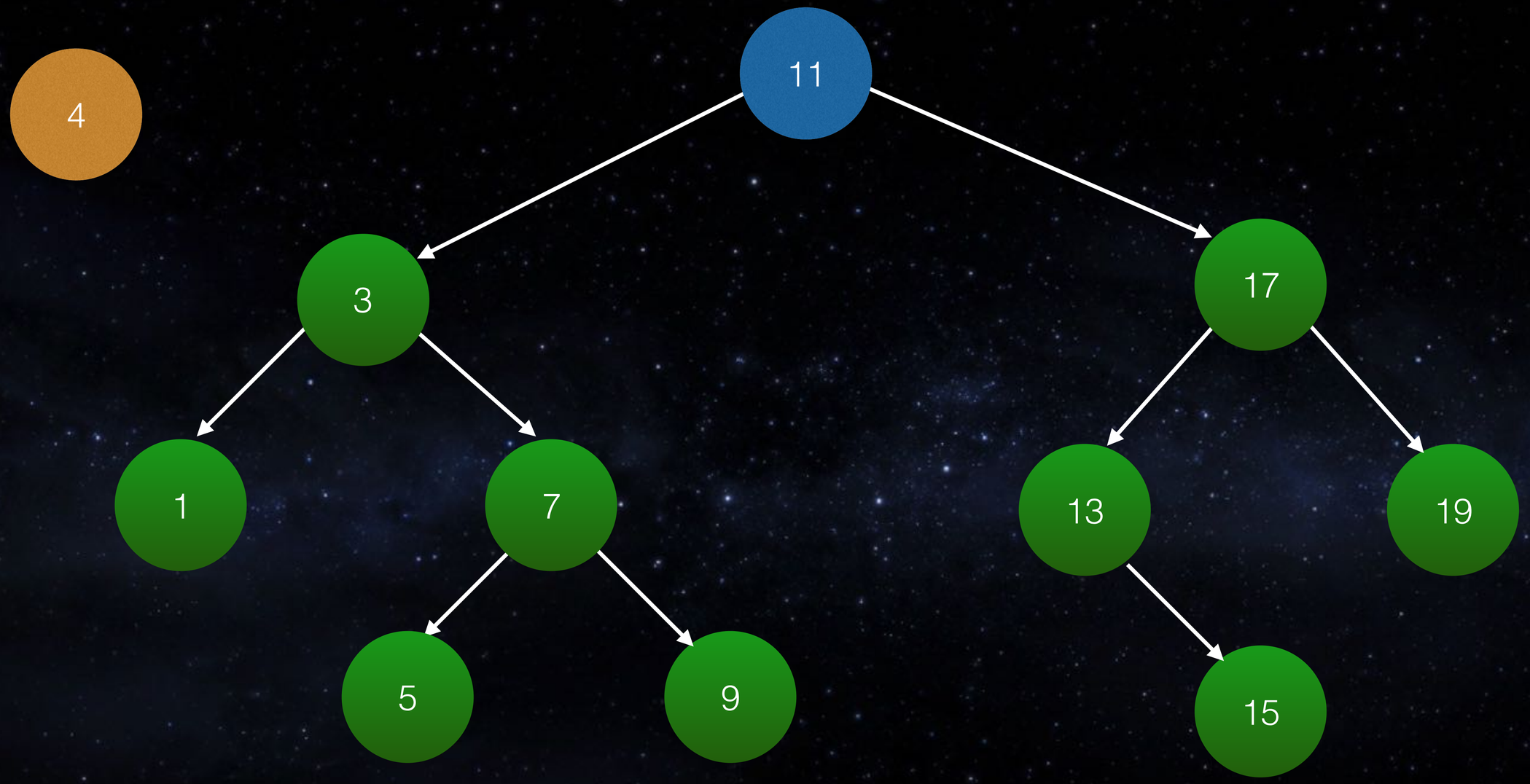

I1
I2
1
2
3
4
5
6

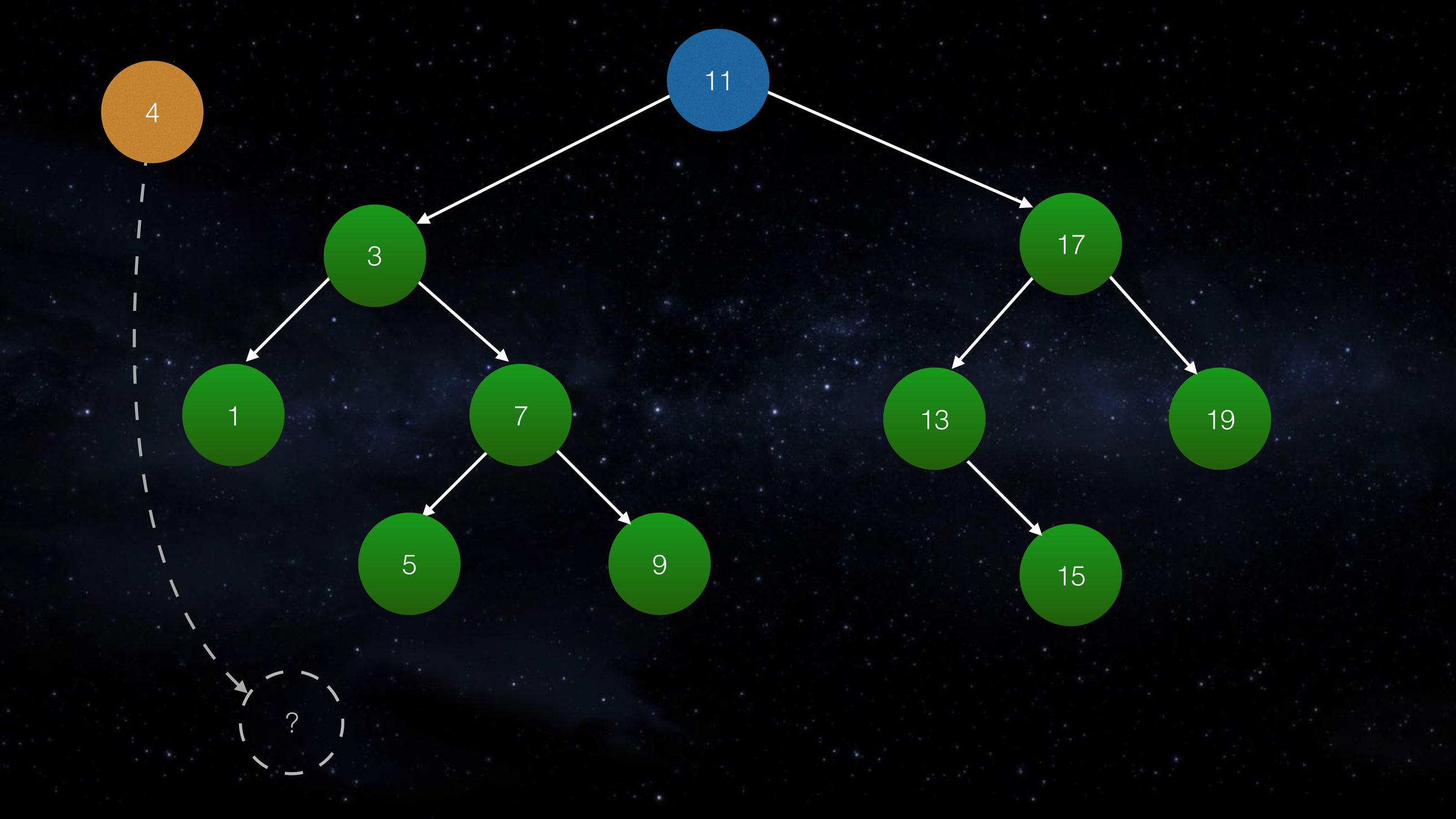

I3
I1
I2
1
2
3
4
5
6

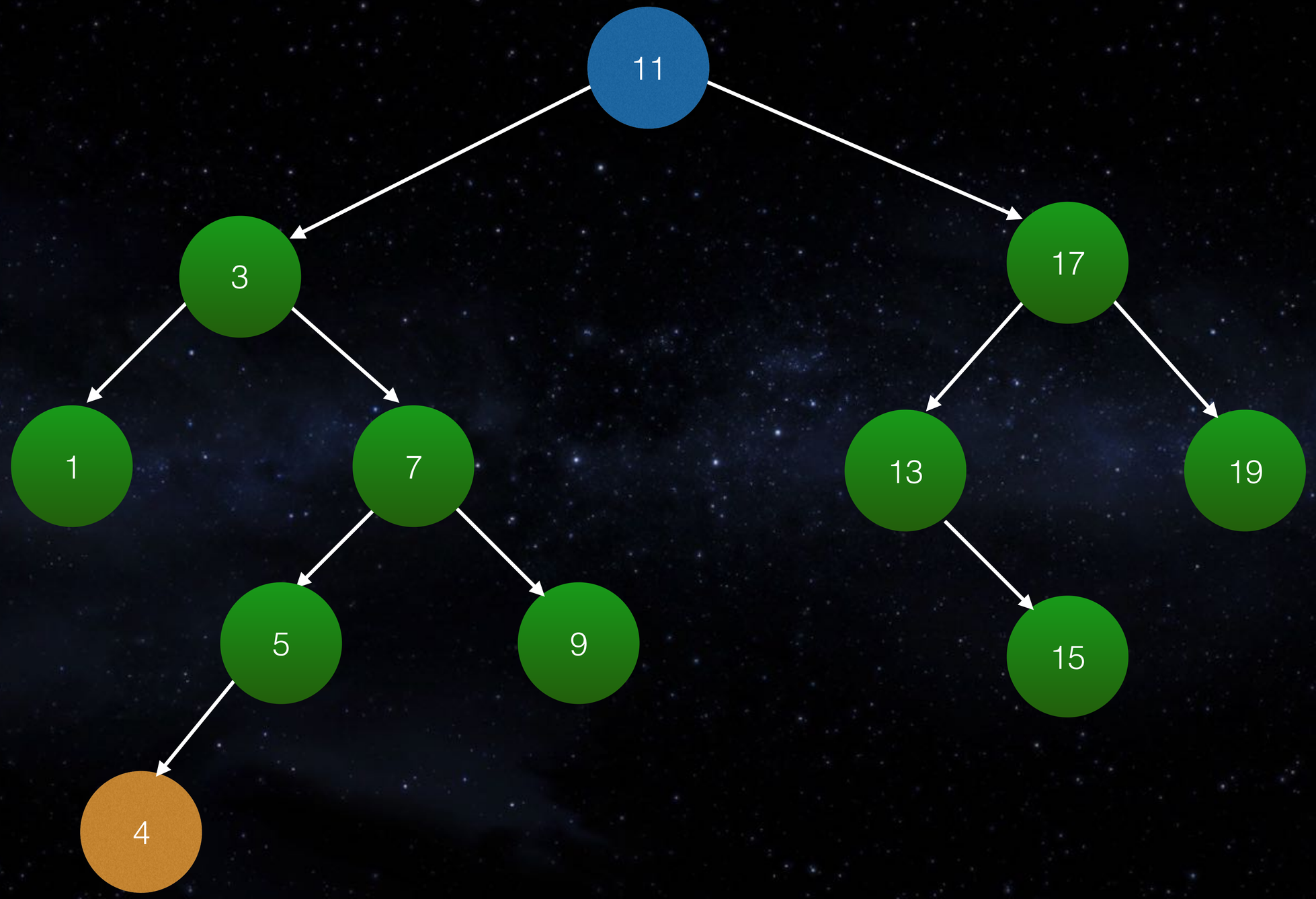

I4
7
8
9
I3
I1
1
2
3
4
5
I2
6

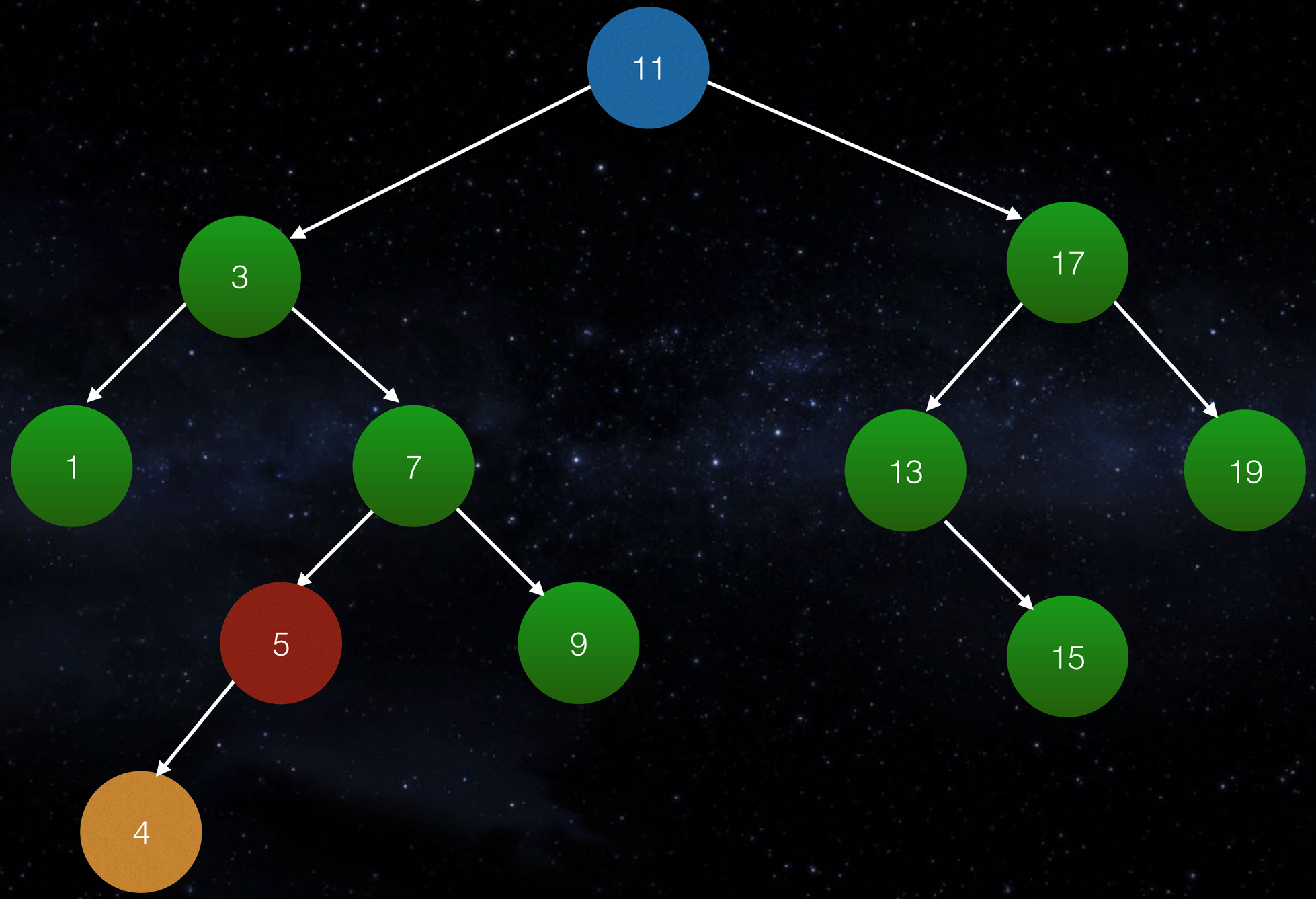

14
7
8
9
3
4
5

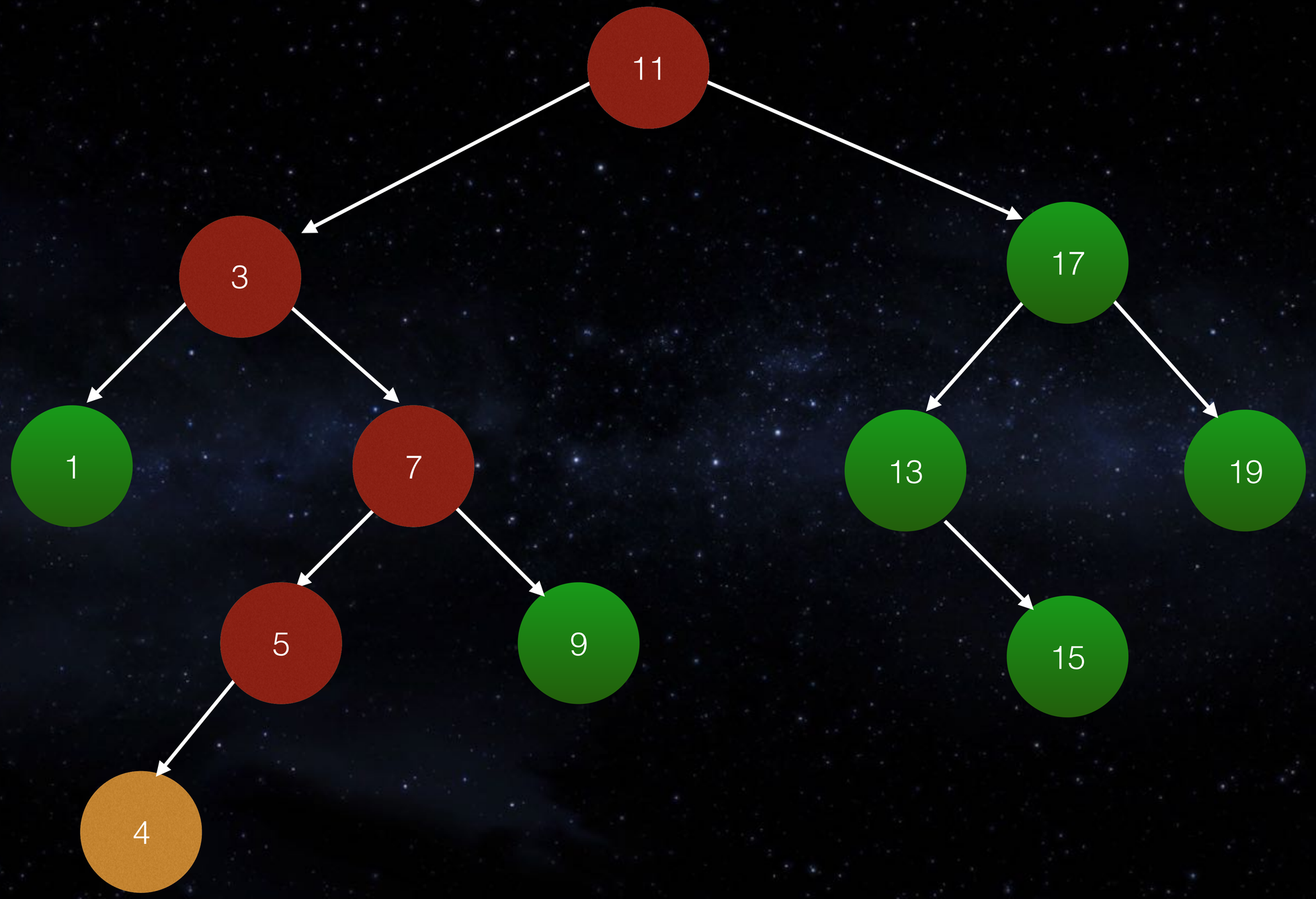

11
3
17
1
7
13
19
5
9
15

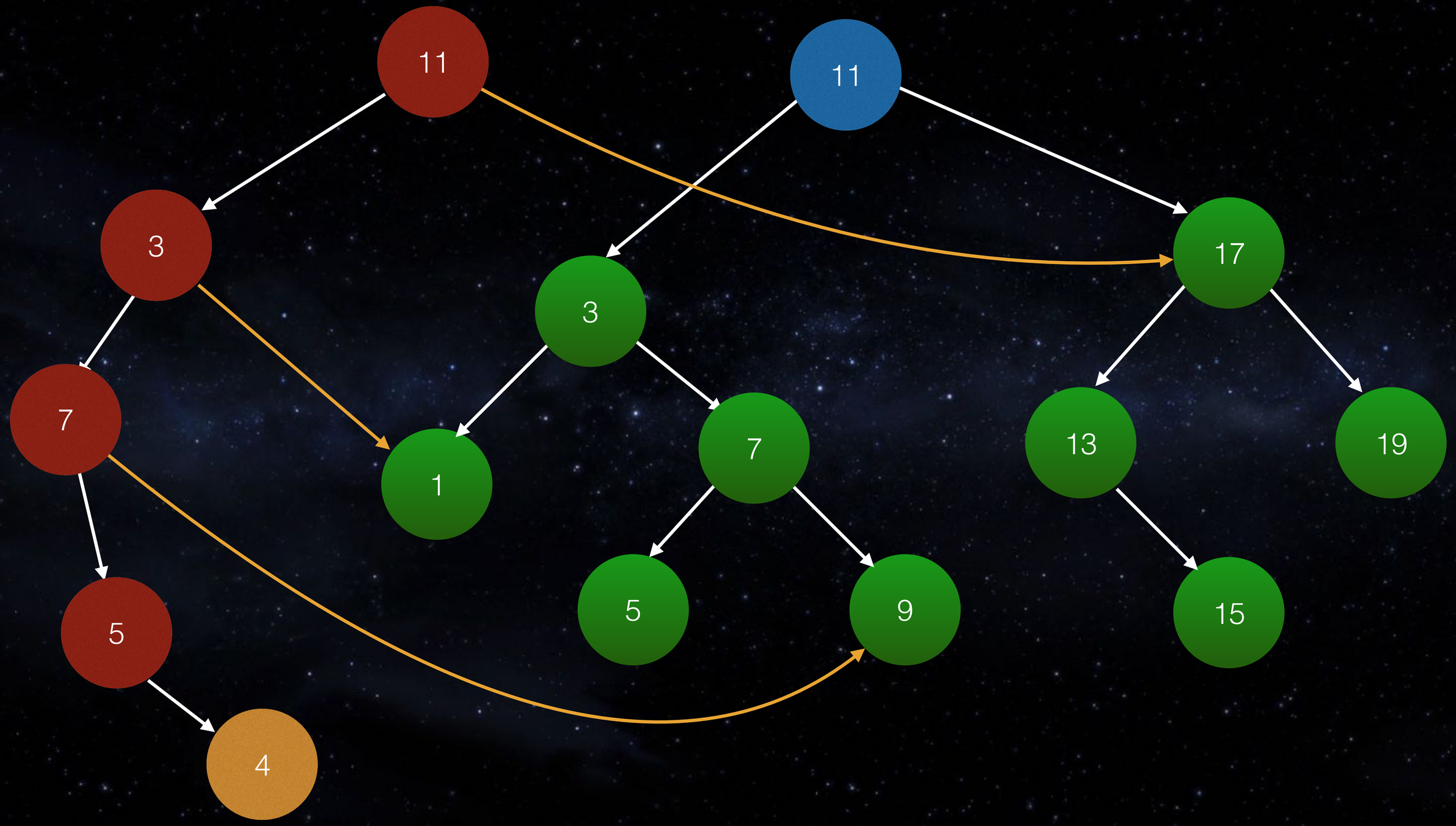

4
11
3
17
1
7
13
19
5
9
15


4
11
3
17
1
7
13
19
5
9
15
?

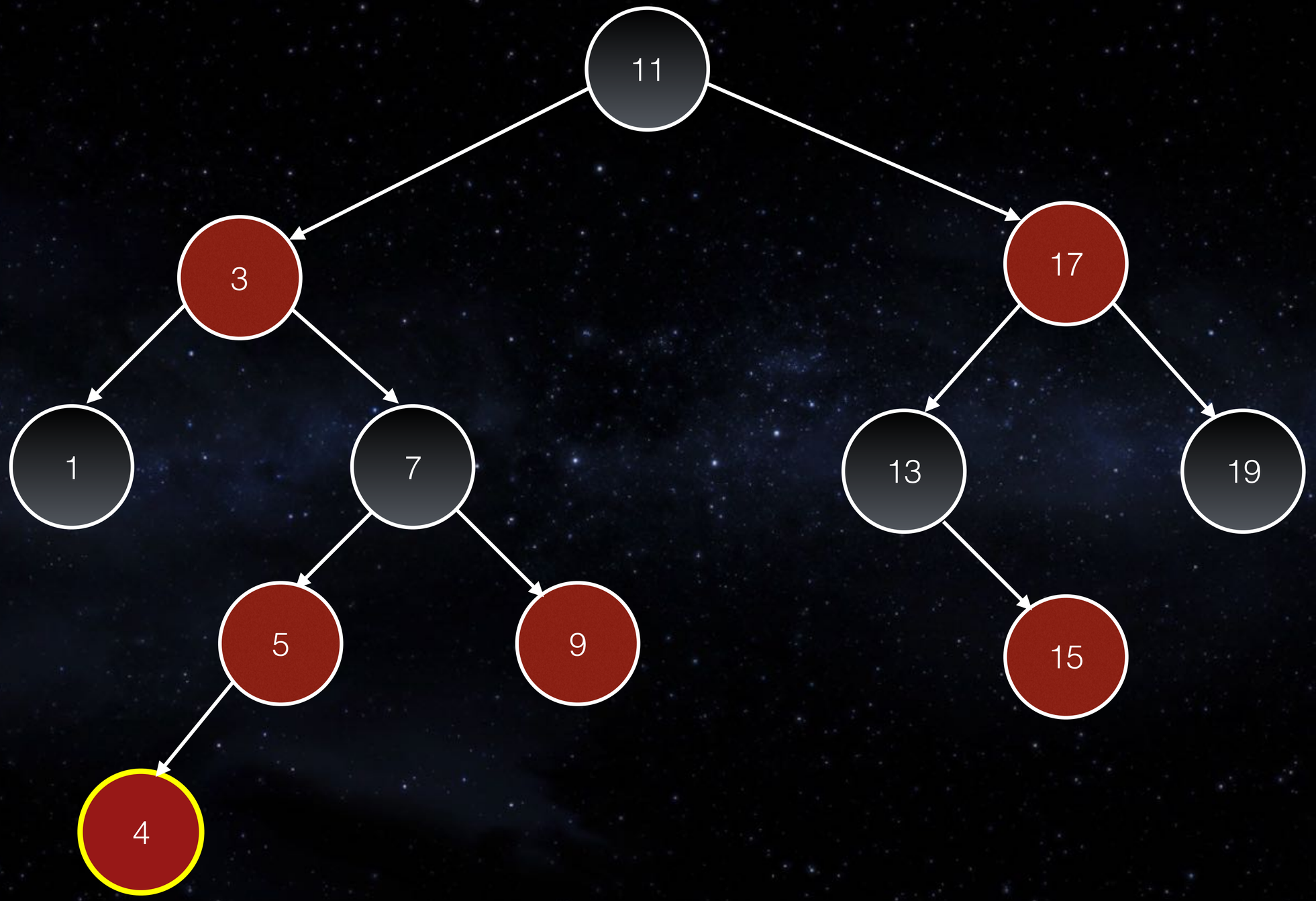

11
3
17
1
7
13
19
5
9
15
4

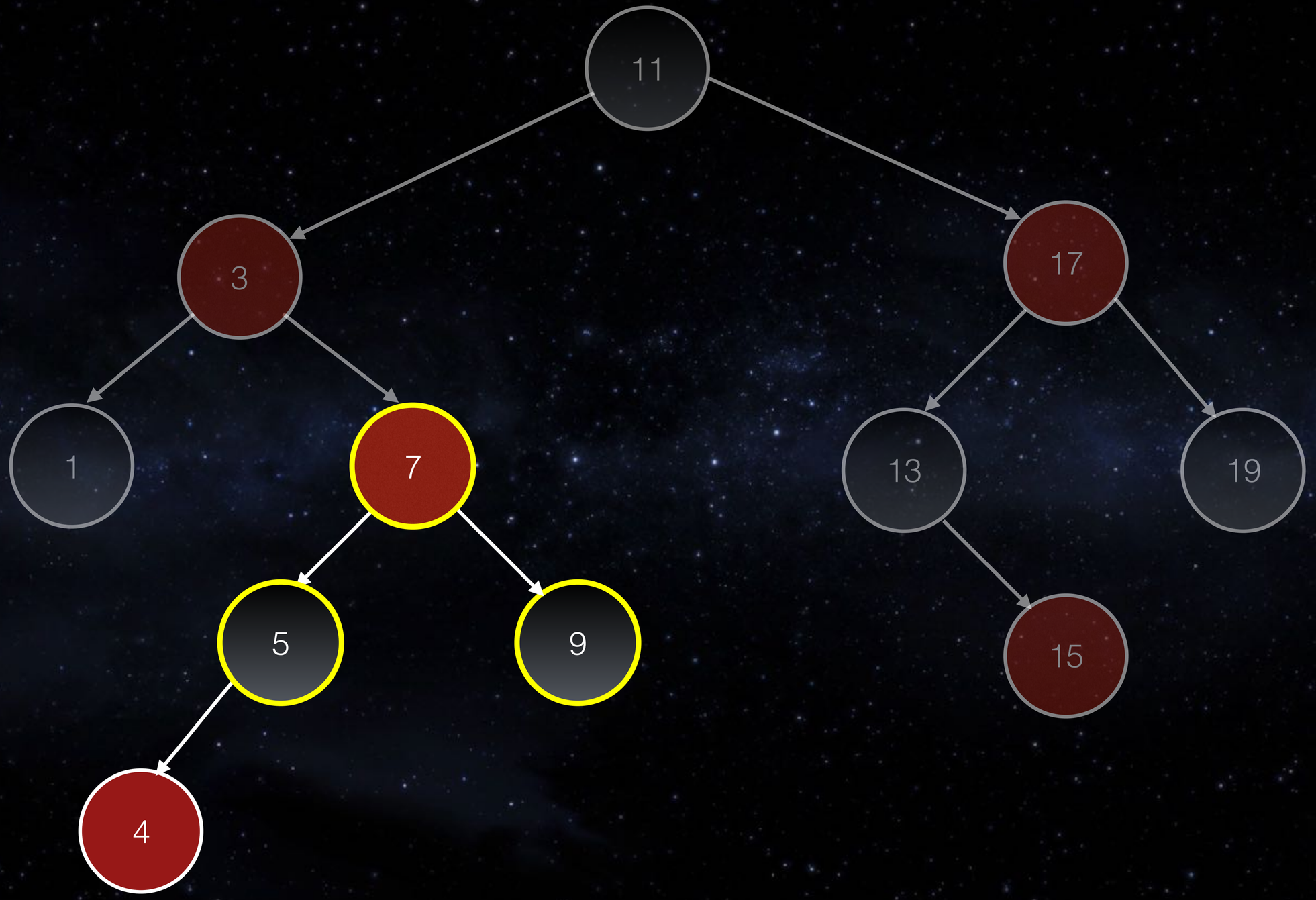

11
3
17
1
7
13
19
5
9
15
4

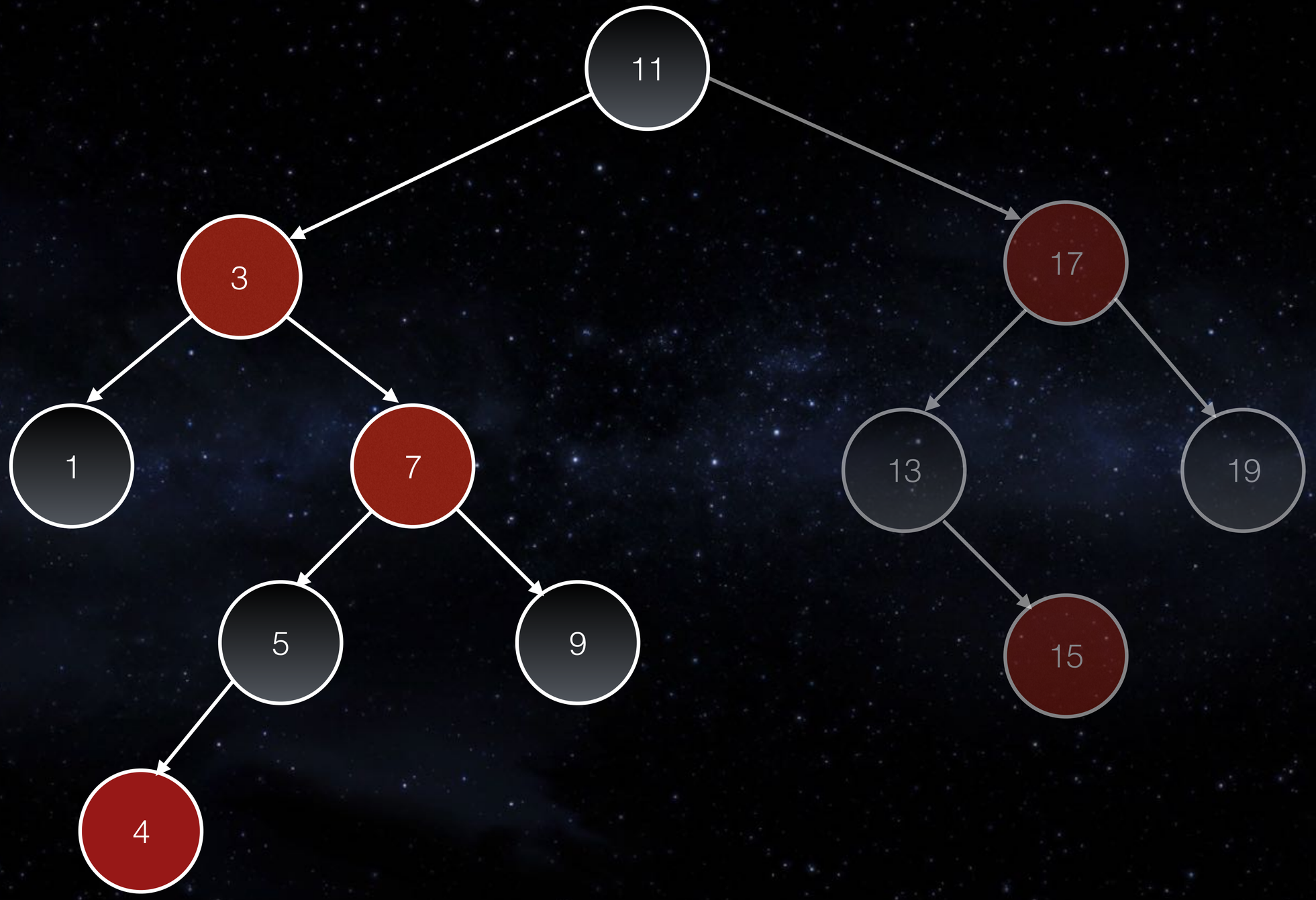

11
3
17
1
7
13
19
5
9
15
4

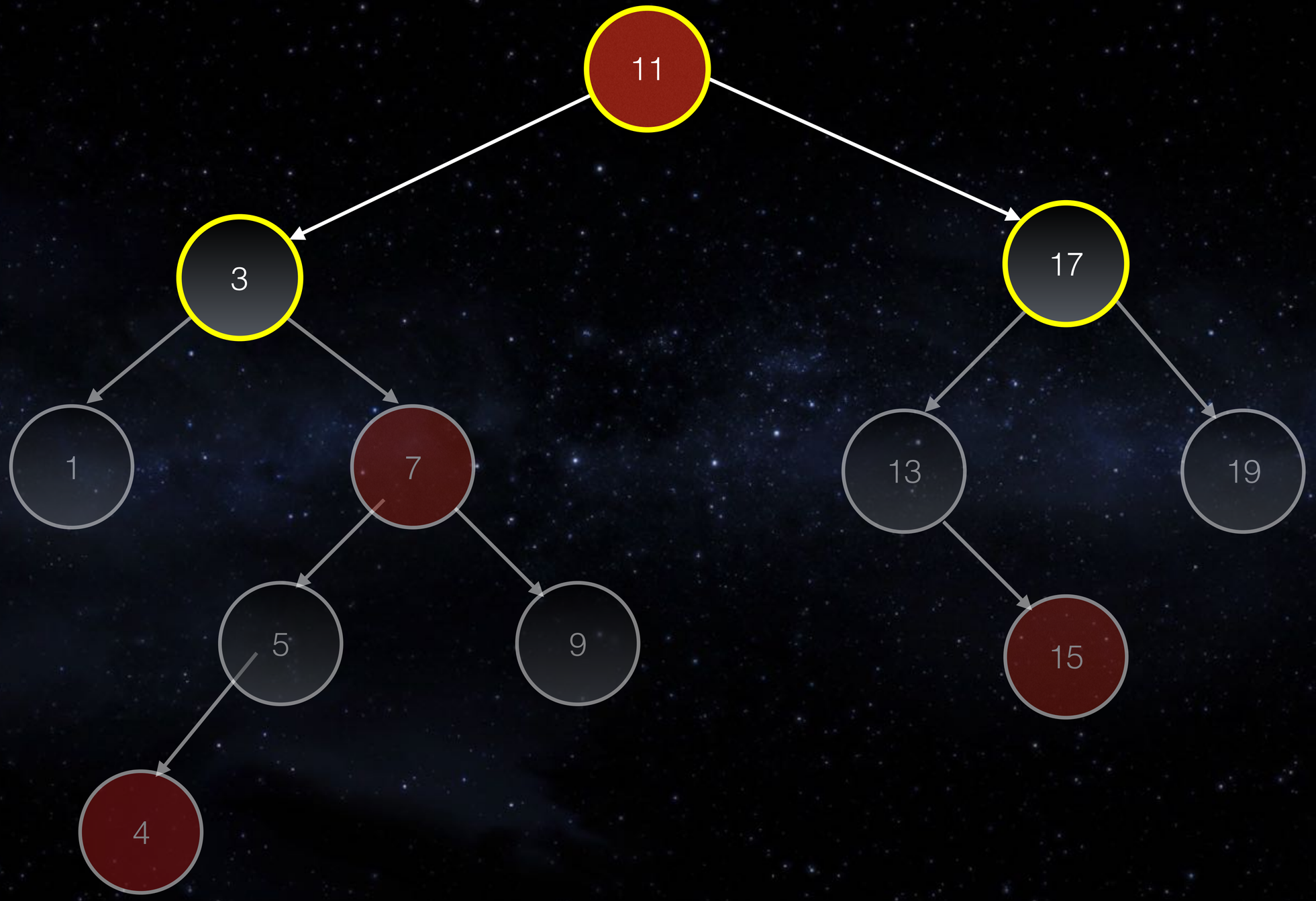

11
11
3
3
17
7
1
7
13
19
5
5
9
15
4

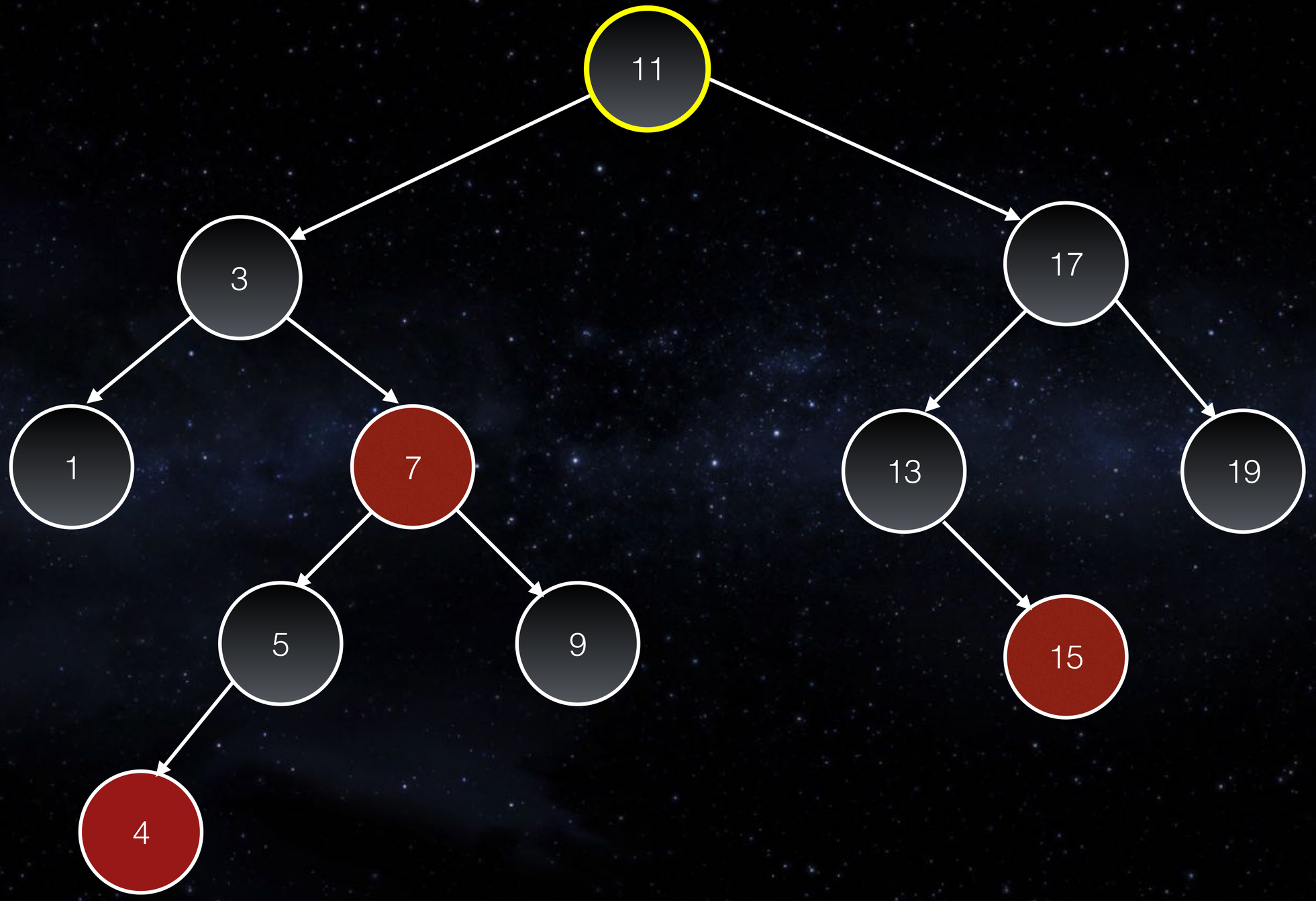
- The root is black
- Red nodes have black children
- New leaves are red
- All paths from root have same number of black nodes

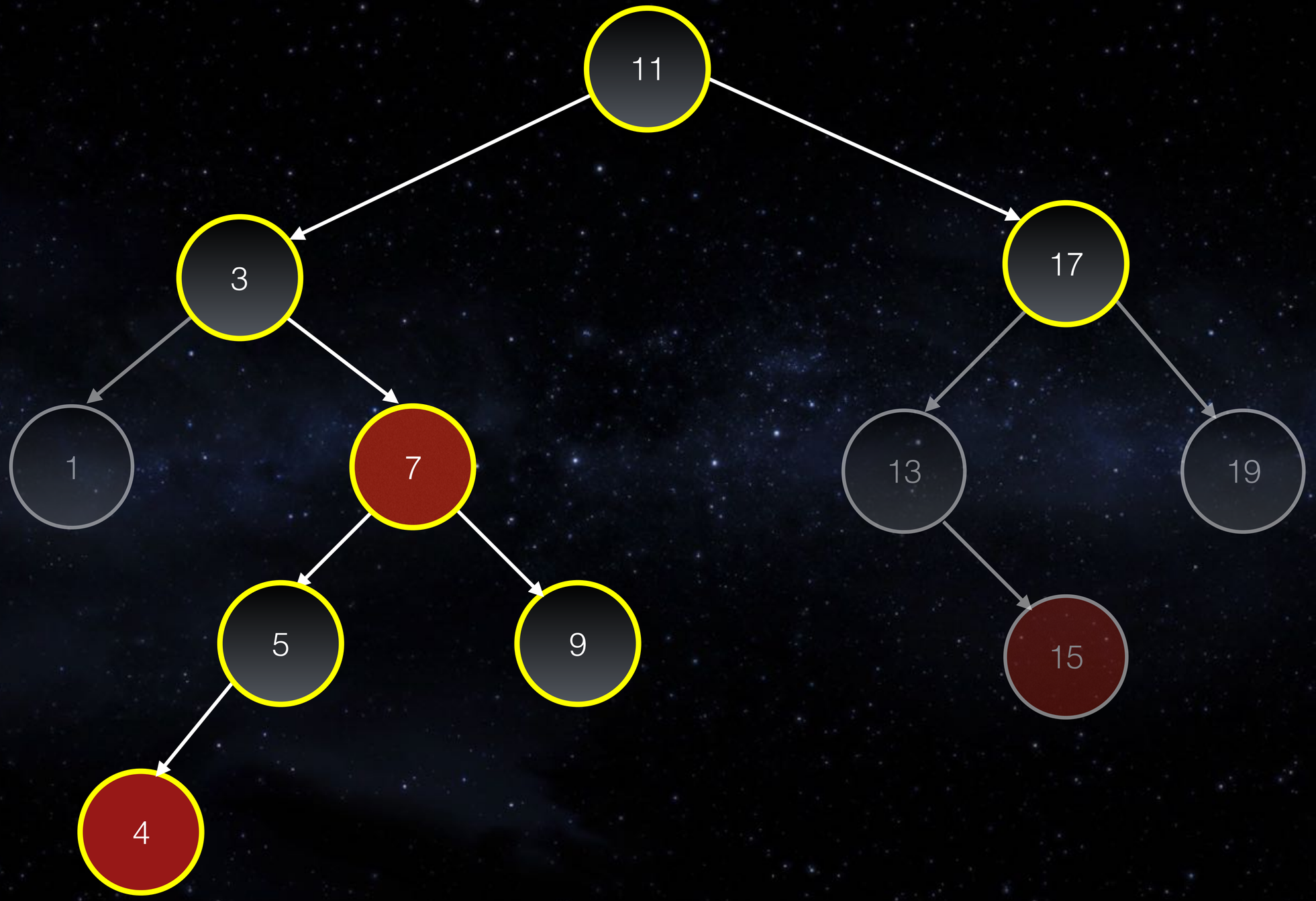

11
3
17
1
7
13
19
5
9
15
4


11
3
17
1
7
13
19
5
9
15
4


11
3
17
1
7
13
19
5
9
15
4


11
3
17
1
7
13
19
5
9
15
4


11
3
17
1
7
13
19
5
9
15
4


11
3
17
1
7
13
19
5
9
15
4

dictionaries, such as the opcodes of an assembler or interpreter).

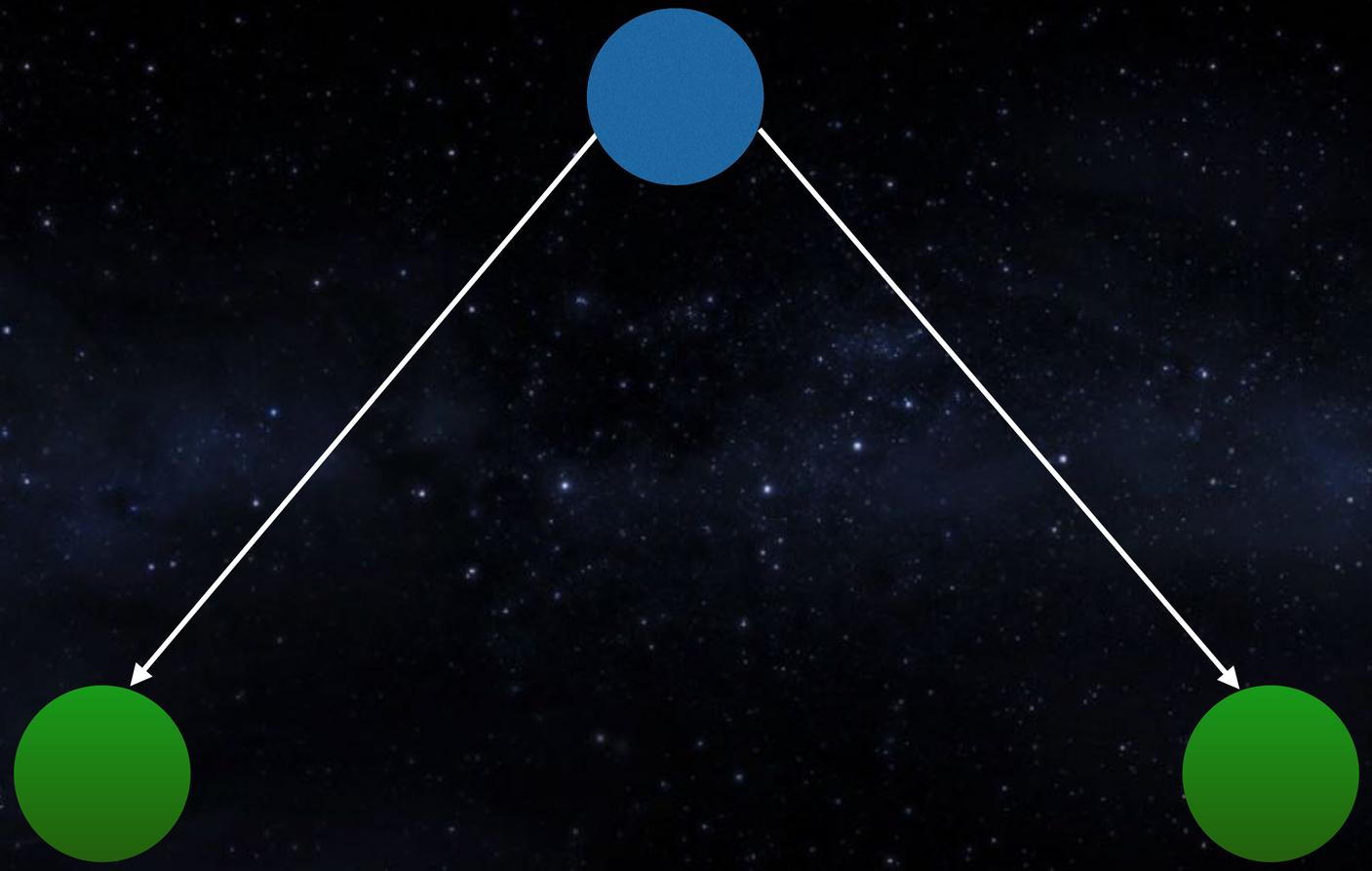
Red–black trees are also particularly valuable in functional programming, where they are one of the most common persistent data structures, used to construct associative arrays and sets which can retain previous versions after mutations. The persistent version of red–black trees requires $O(\log n)$ space for each insertion or deletion, in addition to time.

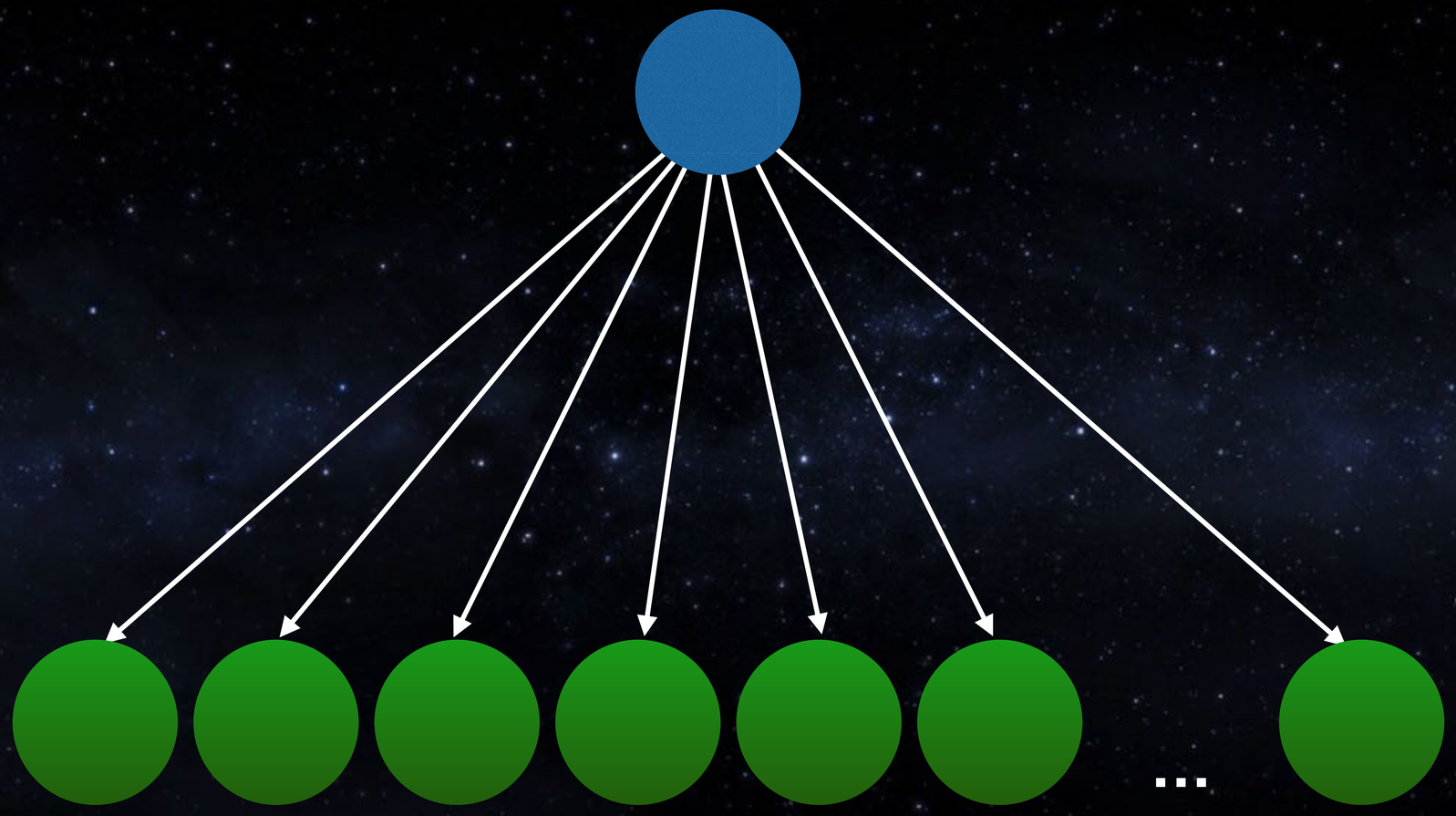
For every 2-4 tree, there are corresponding red–black trees with data elements in the same order. The insertion and deletion operations on 2-4 trees are also equivalent to color-flipping and rotations in red–black trees. This makes 2-4 trees an important tool for understanding the logic behind red–black trees, and this is why many introductory algorithm texts introduce 2-4 trees just before red–black trees, even though 2-4 trees are not often used in practice.

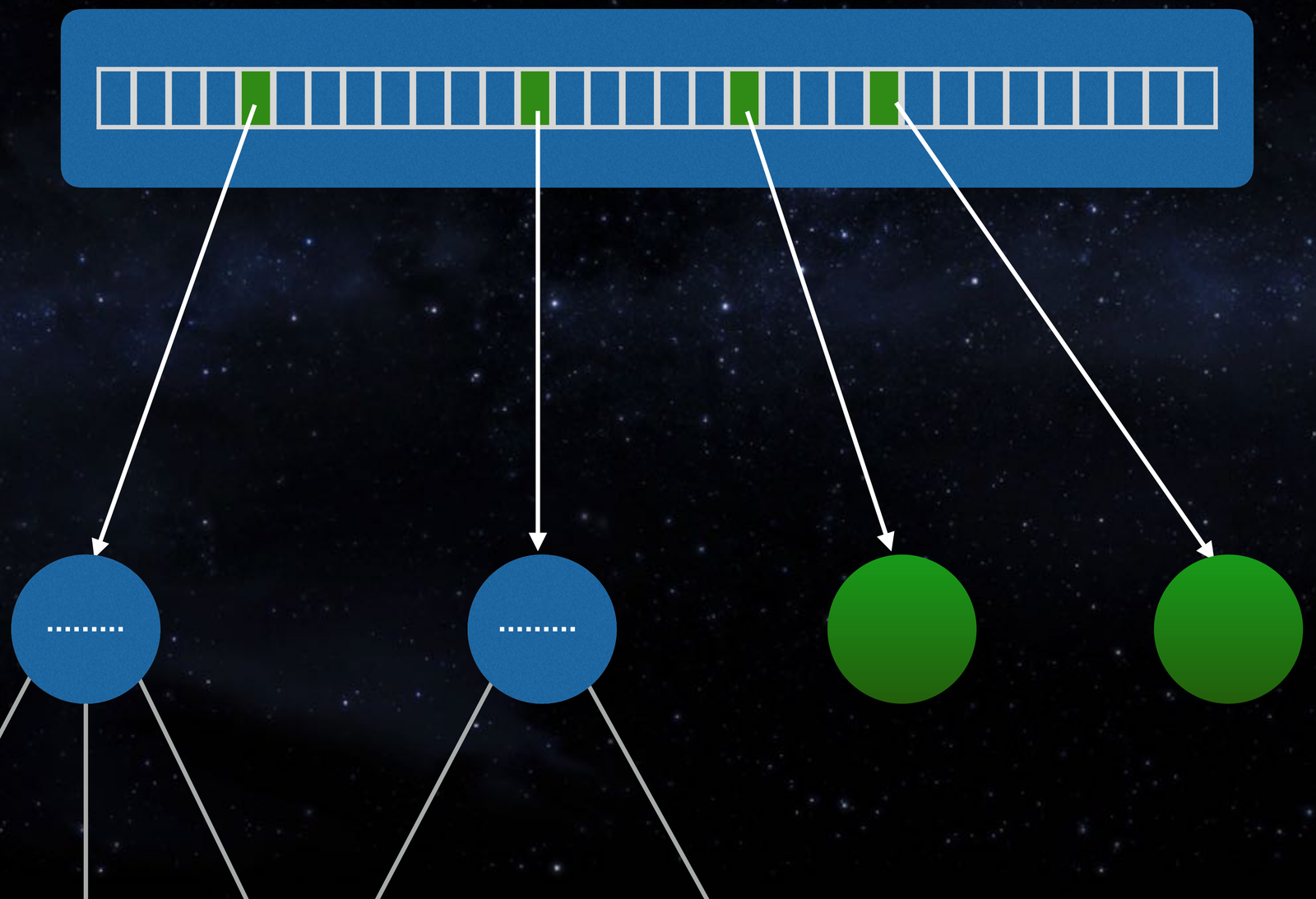
In 2008, Sedgewick introduced a simpler version of the red–black tree called the left-leaning red–black tree[19] by eliminating a previously unspecified degree of freedom in the implementation. The LLRB maintains an additional invariant that all red links must lean left except during inserts and deletes. Red–black trees can be made isometric to either 2-3 trees,[20] or 2-4 trees,[19] for any sequence of operations. The 2-4 tree isometry was described in 1978 by Sedgewick.[*this quote needs a citation*] With 2-4 trees, the isometry is resolved by a "color flip," corresponding to a split, in which the red color of two children nodes leaves the children and moves to the parent node. The tango tree, a type of tree optimized for fast searches, usually[*when?*] uses red–black trees as part of its data structure.

en.wikipedia.org/wiki/Red–black_tree

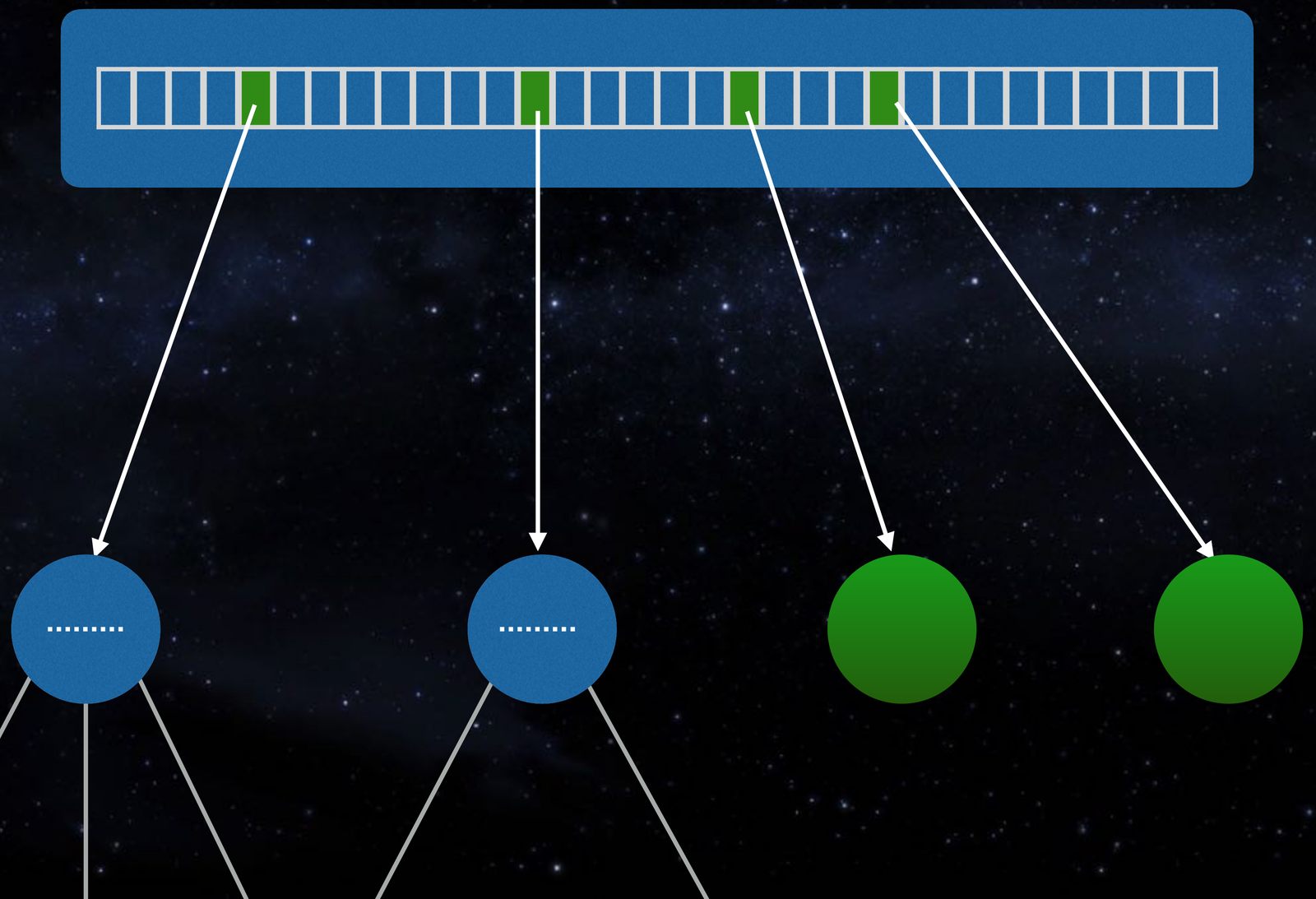
Red–black trees are also particularly valuable in functional programming, where they are one of the most common persistent data structures, used to construct associative arrays and sets which can retain previous versions after mutations. The persistent version of red–black trees requires $O(\log n)$ space for each insertion or deletion, in addition to time.



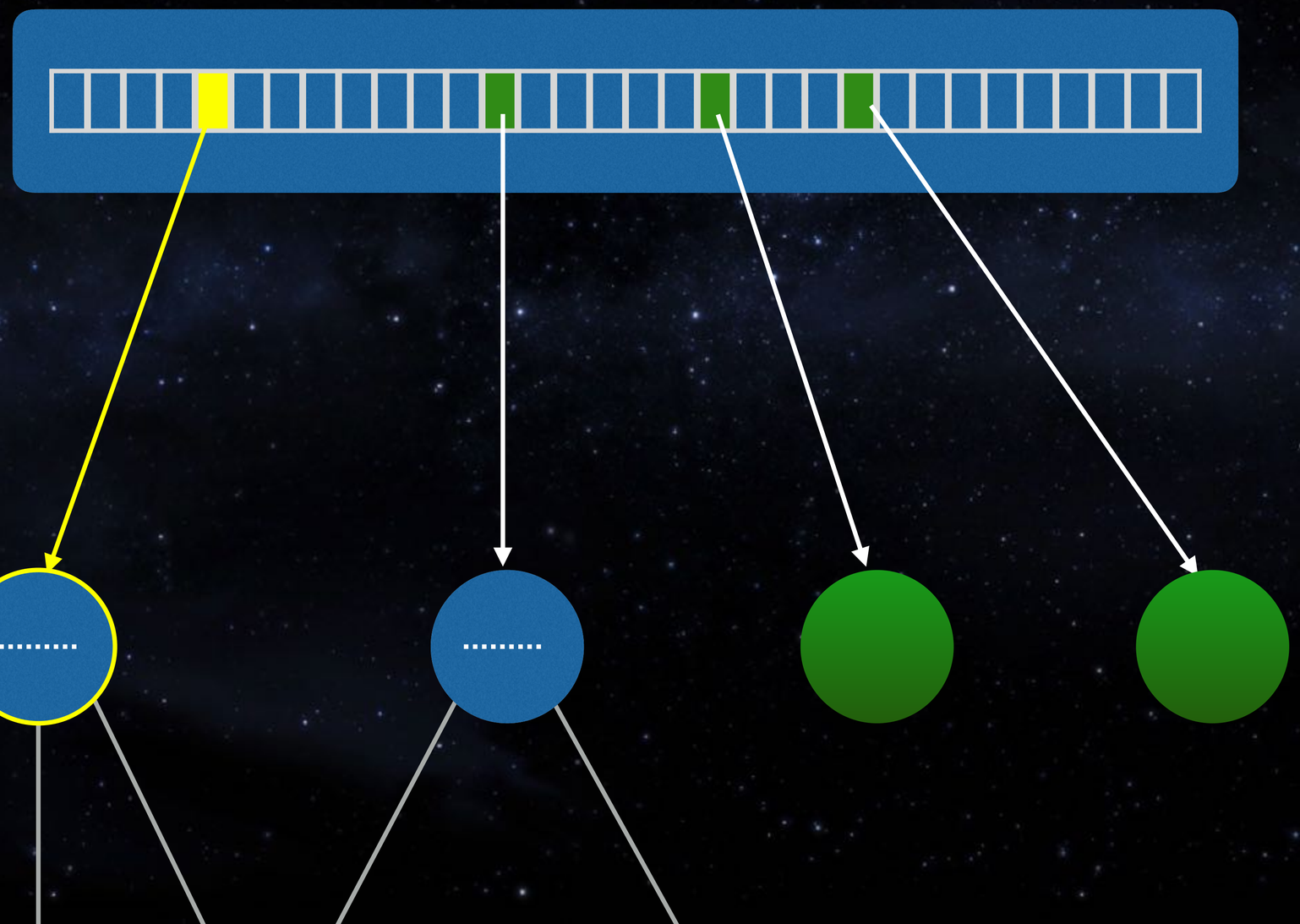




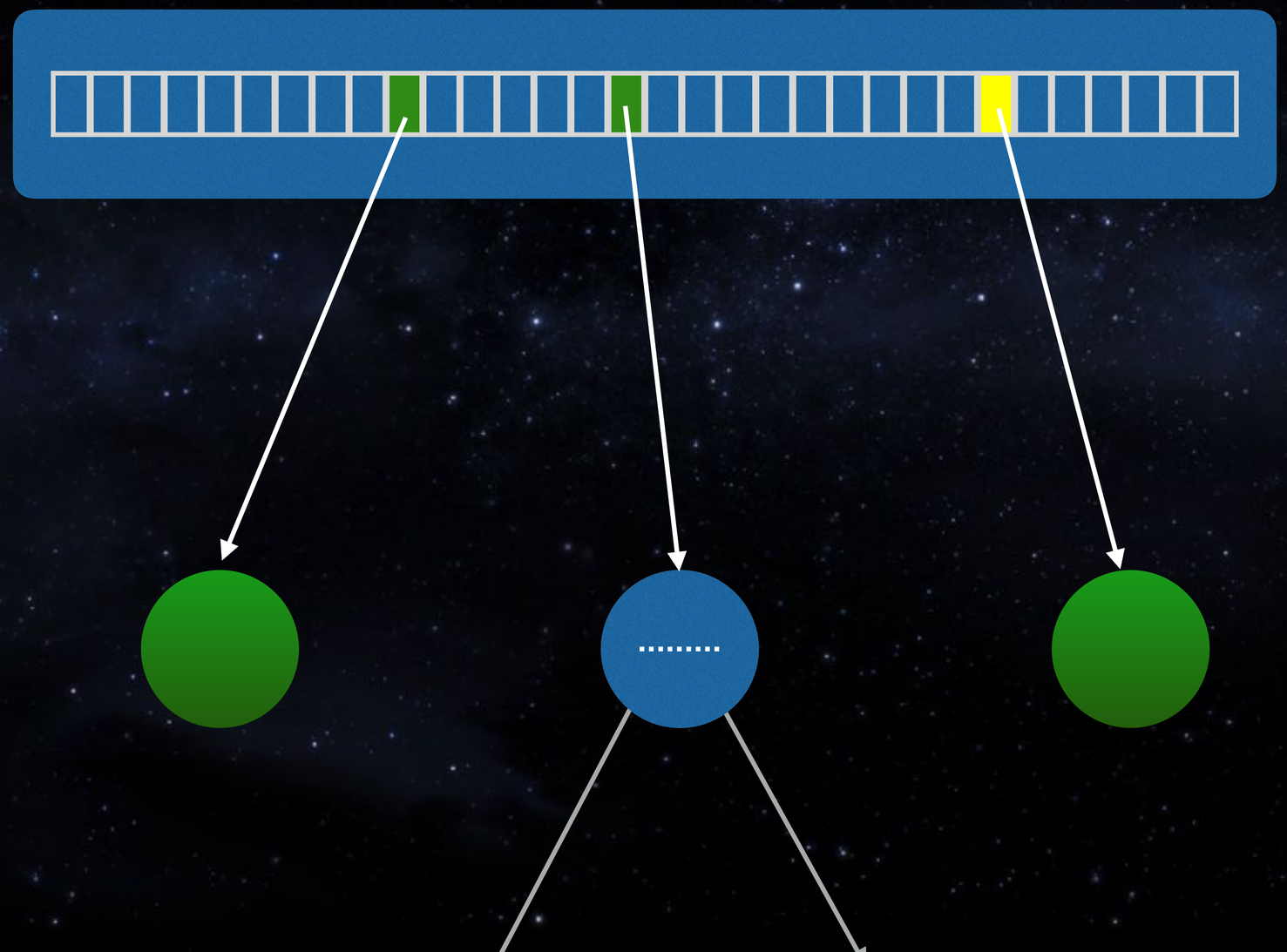
Hash 00110101101001001110100110110 11

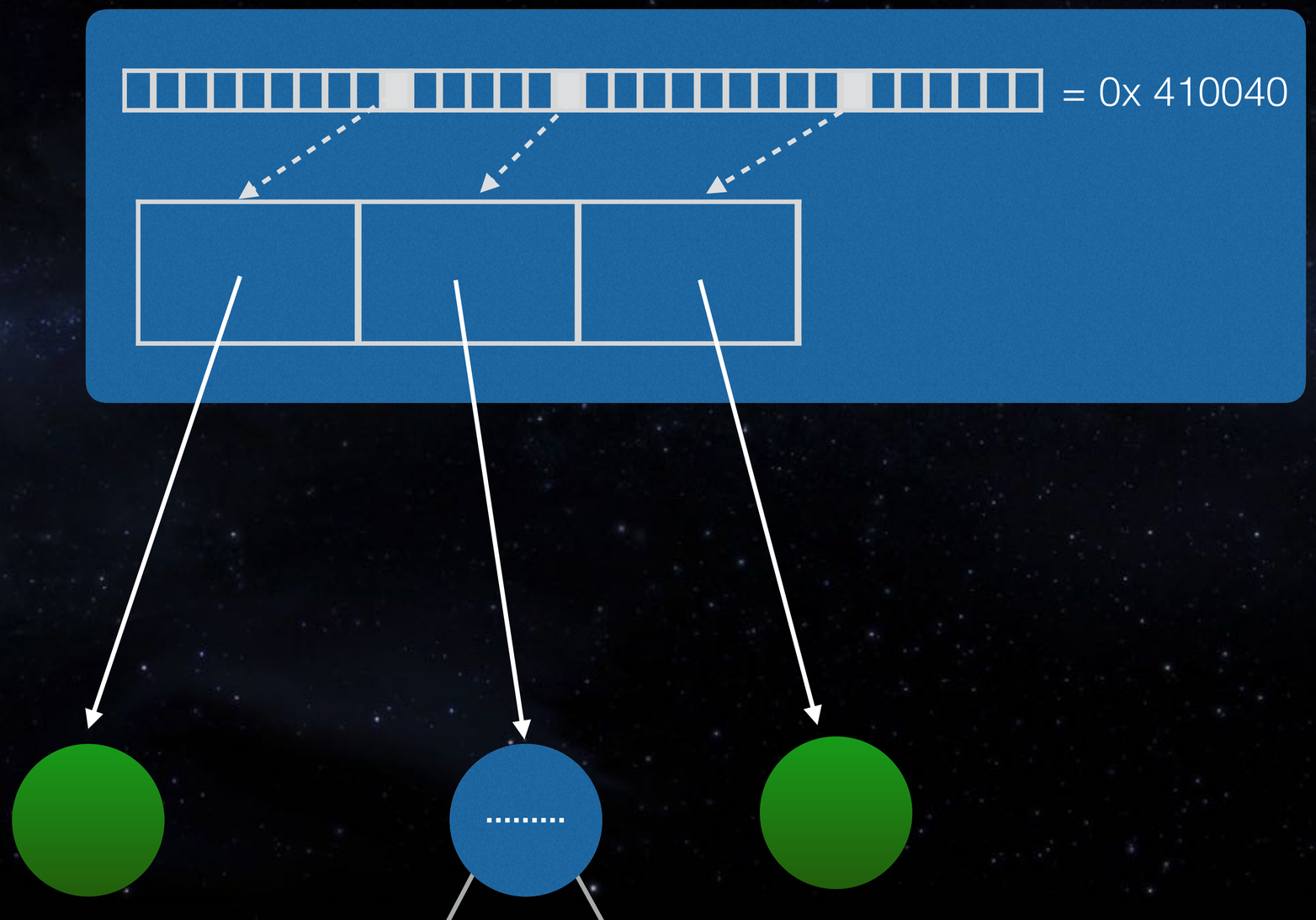


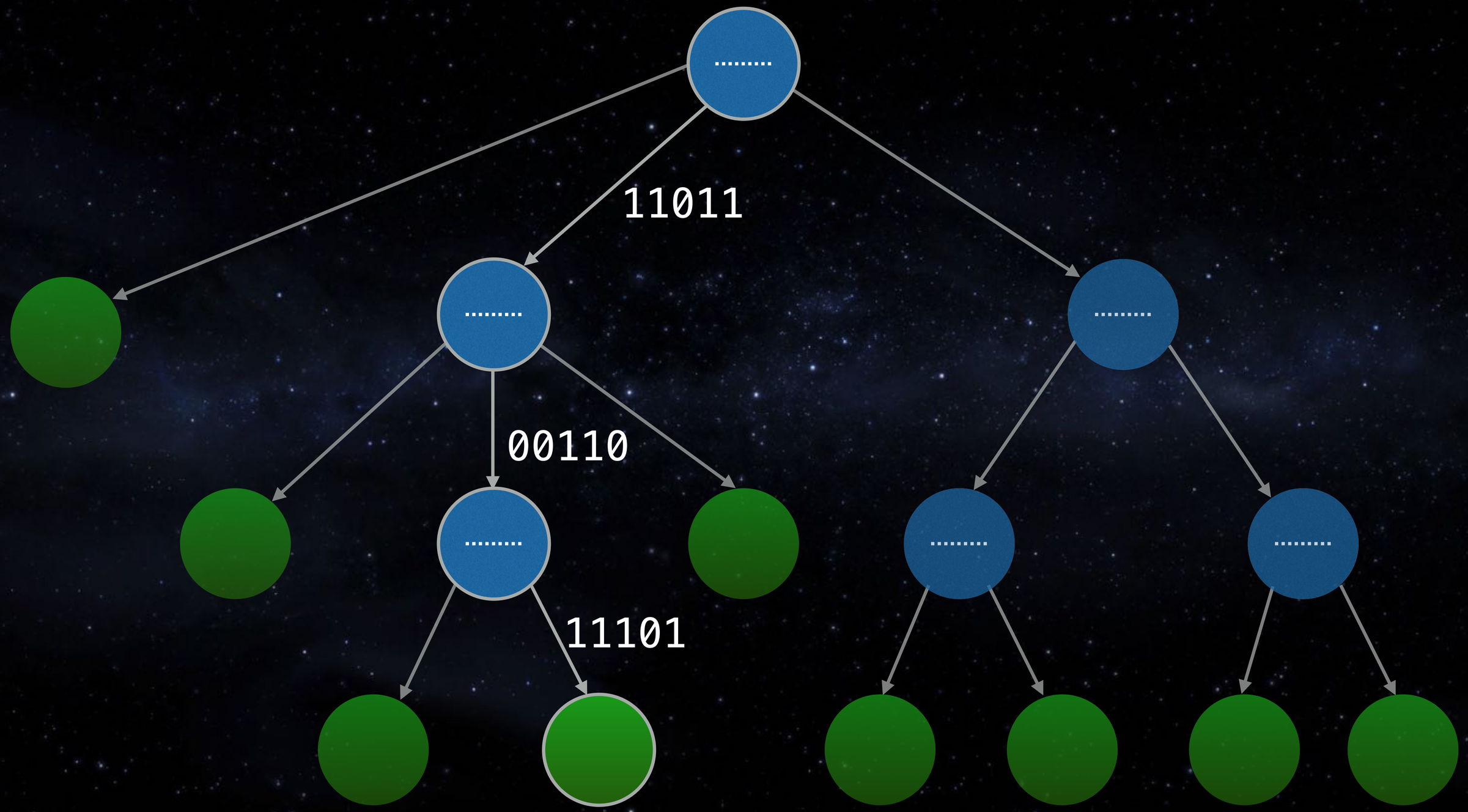
Hash 00110101101001001110100110**11011** **27**

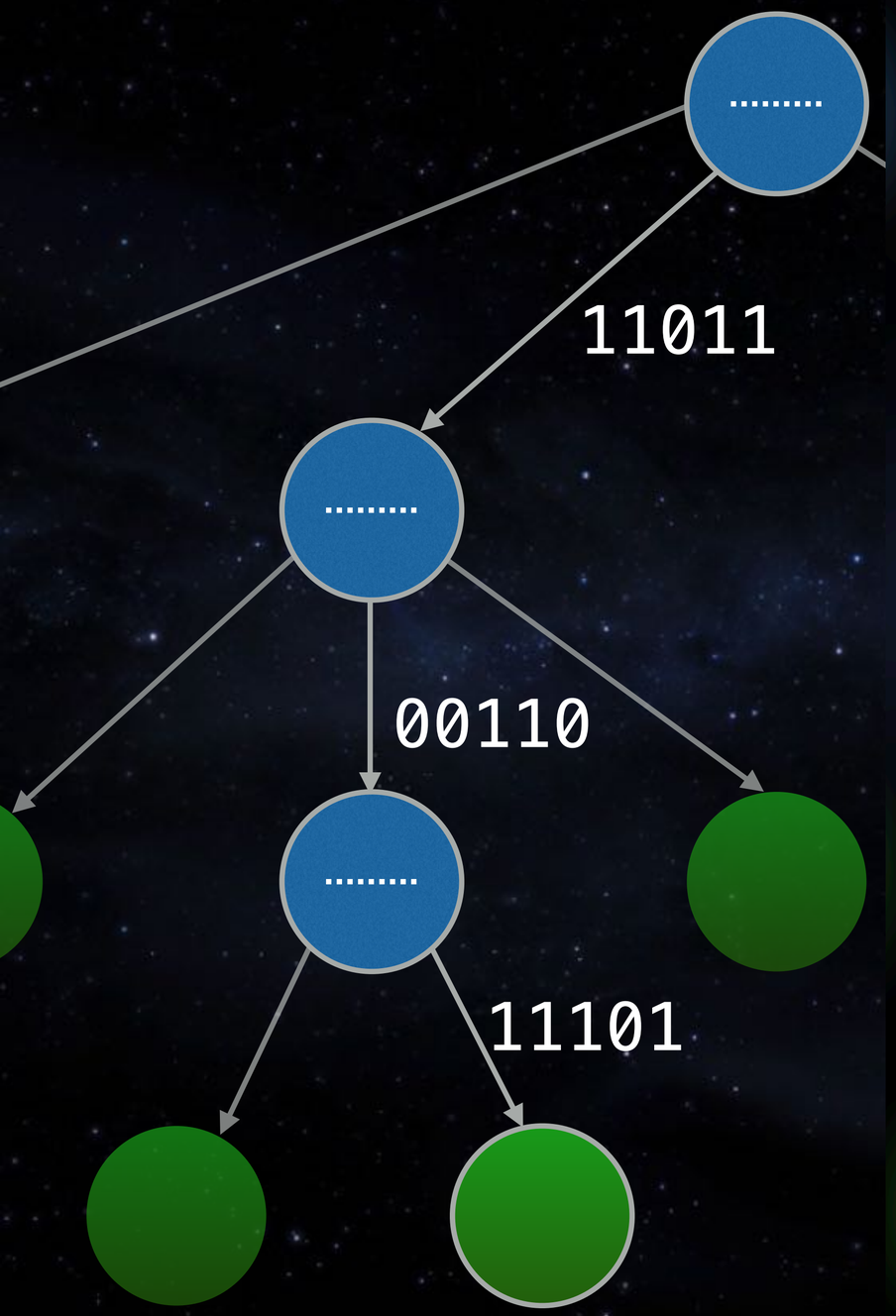


Hash 00110101101001001110100110110 11 6




= 0x 410040
.....


.....
11011
.....
.....
00110
.....
.....
.....
11101



max depth = **6/7** (32 bit)

av. depth for 15m values = **5**

leaves hold arrays of values

more space efficient than hash tables

$O(\log_{32} n)$ complexity

no rebalancing!

But there exists another general purpose data structure that combines many of the characteristics of trees and hash tables into something that in many important ways is superior to both, and with minimal downside (they are close but not quite as fast as pure hash tables). Hash Array Mapped Tries are more memory efficient than hash tables and, as a bonus, are trivially made persistent - with big implications for concurrency, functional programming, and other applications that benefit from being able to treat them immutably (as well as share large amounts of common state in memory at once).

This talk will describe how this data structure works from the ground up and look at a reference implementation I am writing with the intention of proposing as a boost library - and possibly later for standardisation. We'll also look at how it can be used in practice, and some of the performance characteristics.

Speakers

Phil Nash

Developer Advocate, JetBrains

I'm the author of the test framework, Catch, and also have feet in the Swift and F# worlds. | As Developer Advocate at JetBrains I'm involved with CLion, AppCode and ReSharper C++.

But there exists another general purpose data structure that combines many of the characteristics of trees and hash tables into something that in many important ways is superior to both, and with minimal downside (they are close but not quite as fast as pure hash tables). Hash Array Mapped Tries are more memory efficient than hash tables and, as a bonus, are trivially made persistent - with big implications for concurrency, functional programming, and other applications that benefit from being able to treat them immutably (as well as share large amounts of common state in memory at once).

This talk will describe how this data structure works from the ground up and look at a reference implementation I am writing with the intention of proposing as a boost library - and possibly later for standardisation. We'll also look at how it can be used in practice, and some of the performance characteristics.

Performance

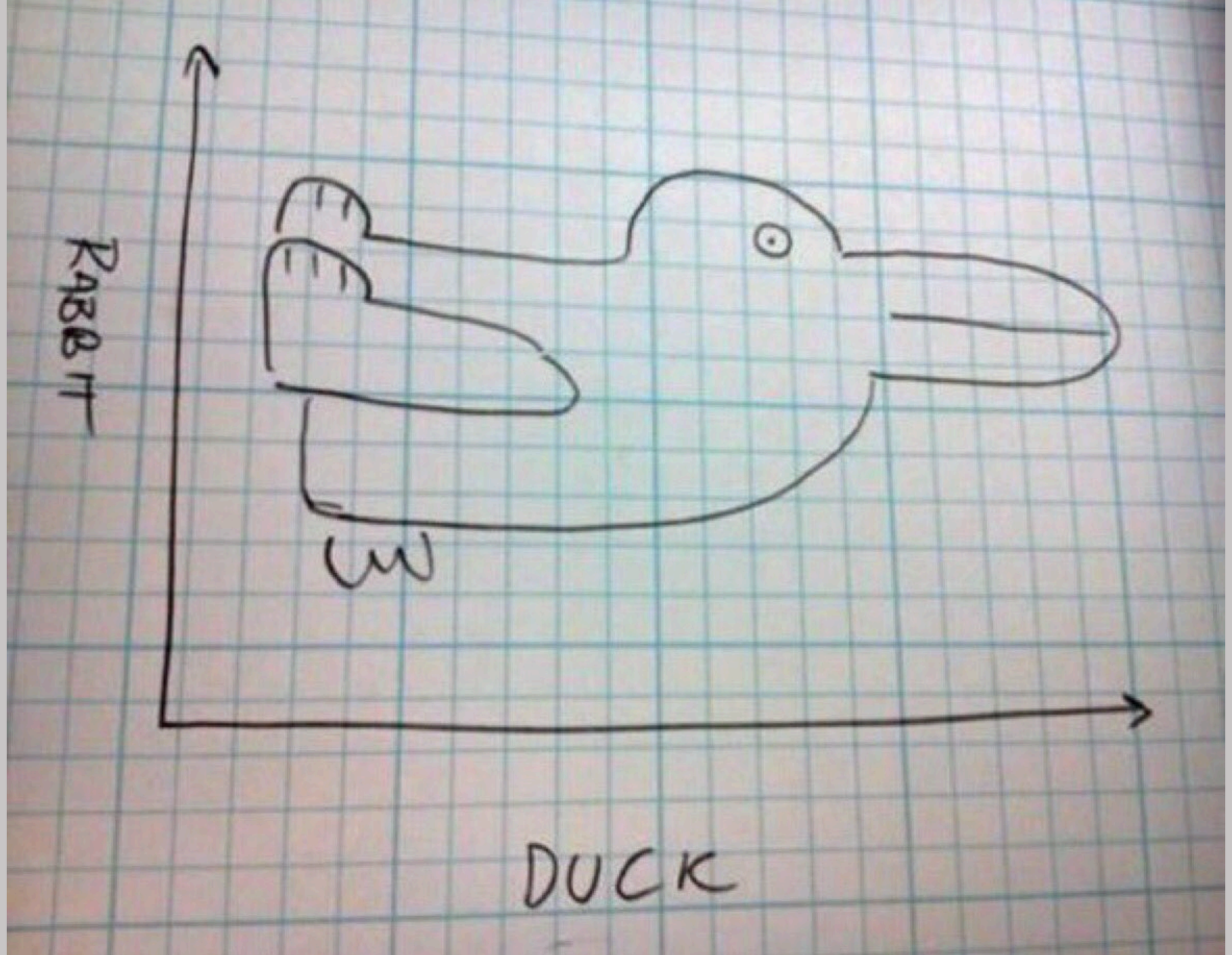
nonius.io

100 samples

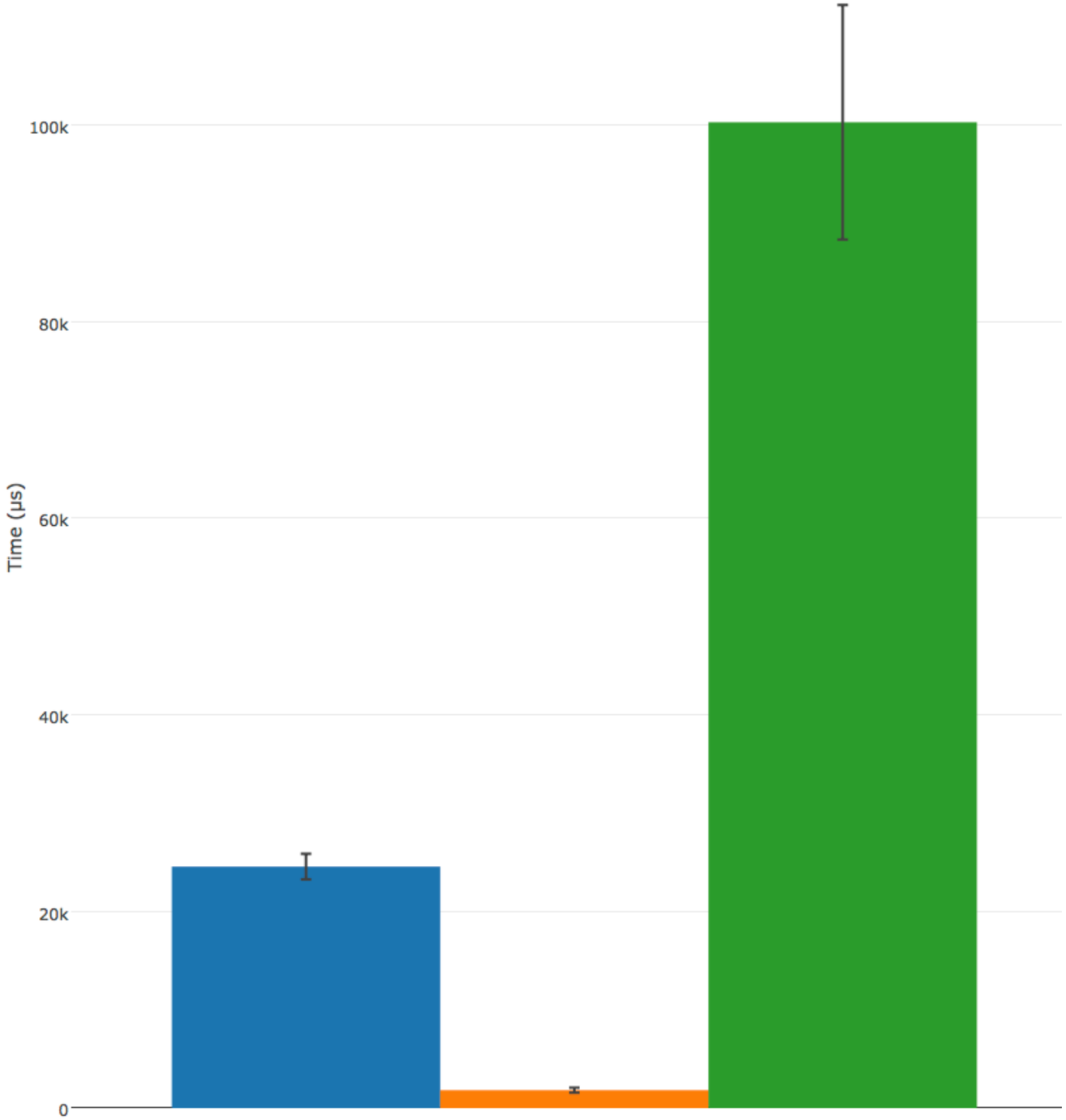
libc++

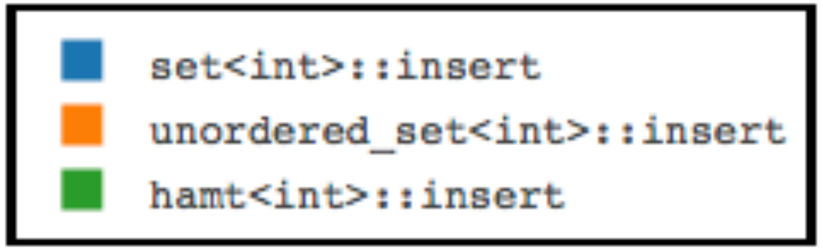
100k ints

hamt, set, unordered_set

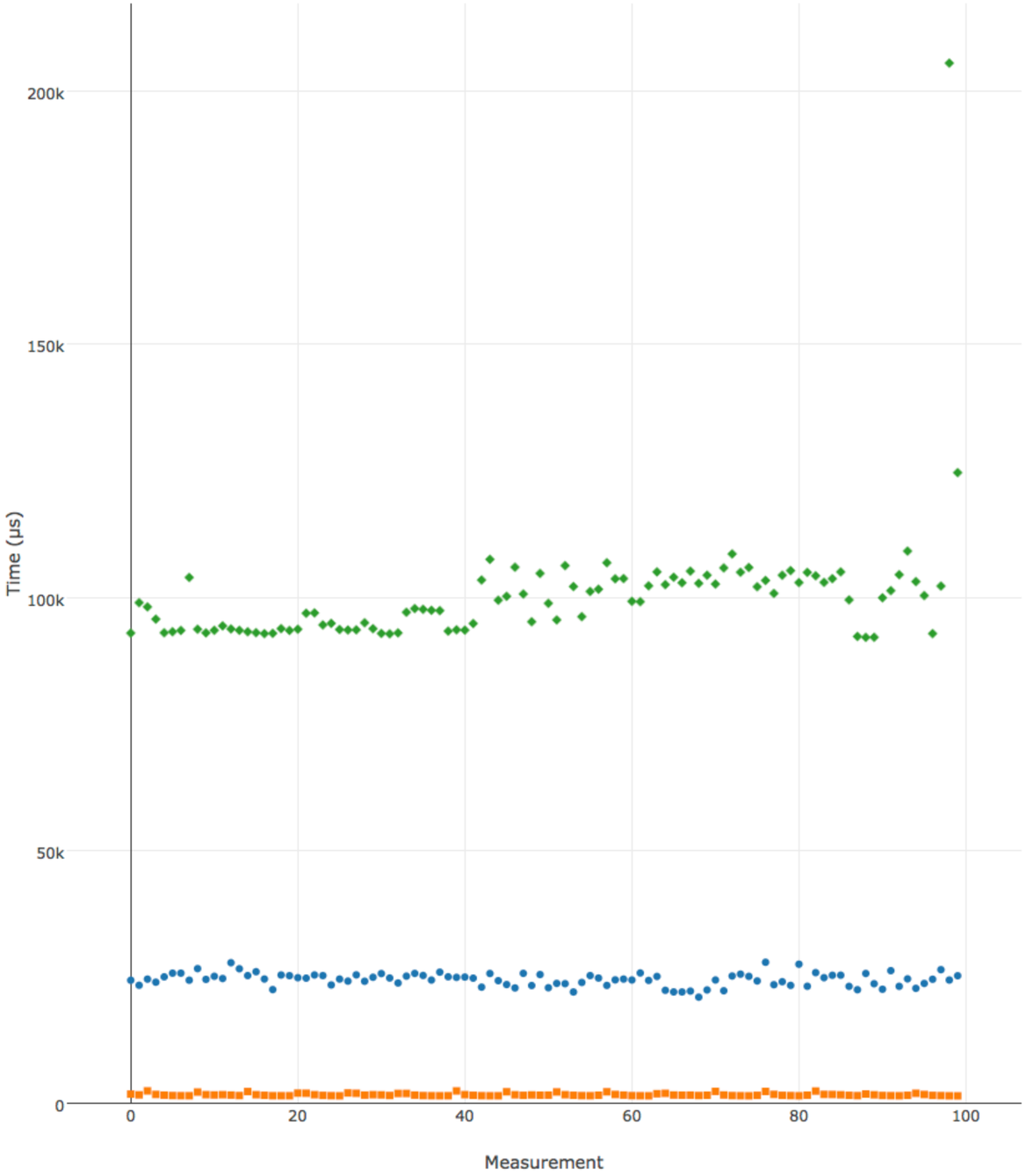

RABBIT
DUCK

insert
100k ints



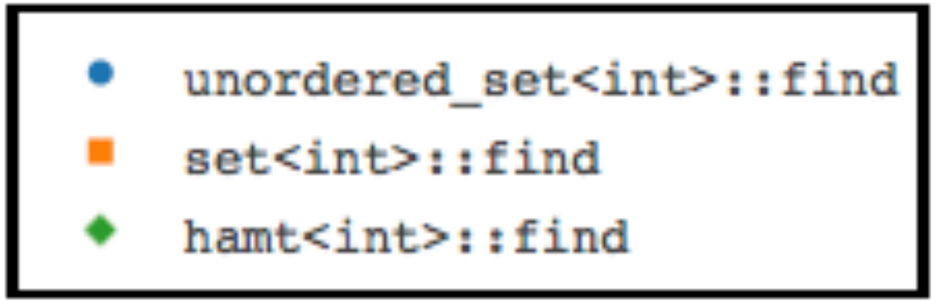


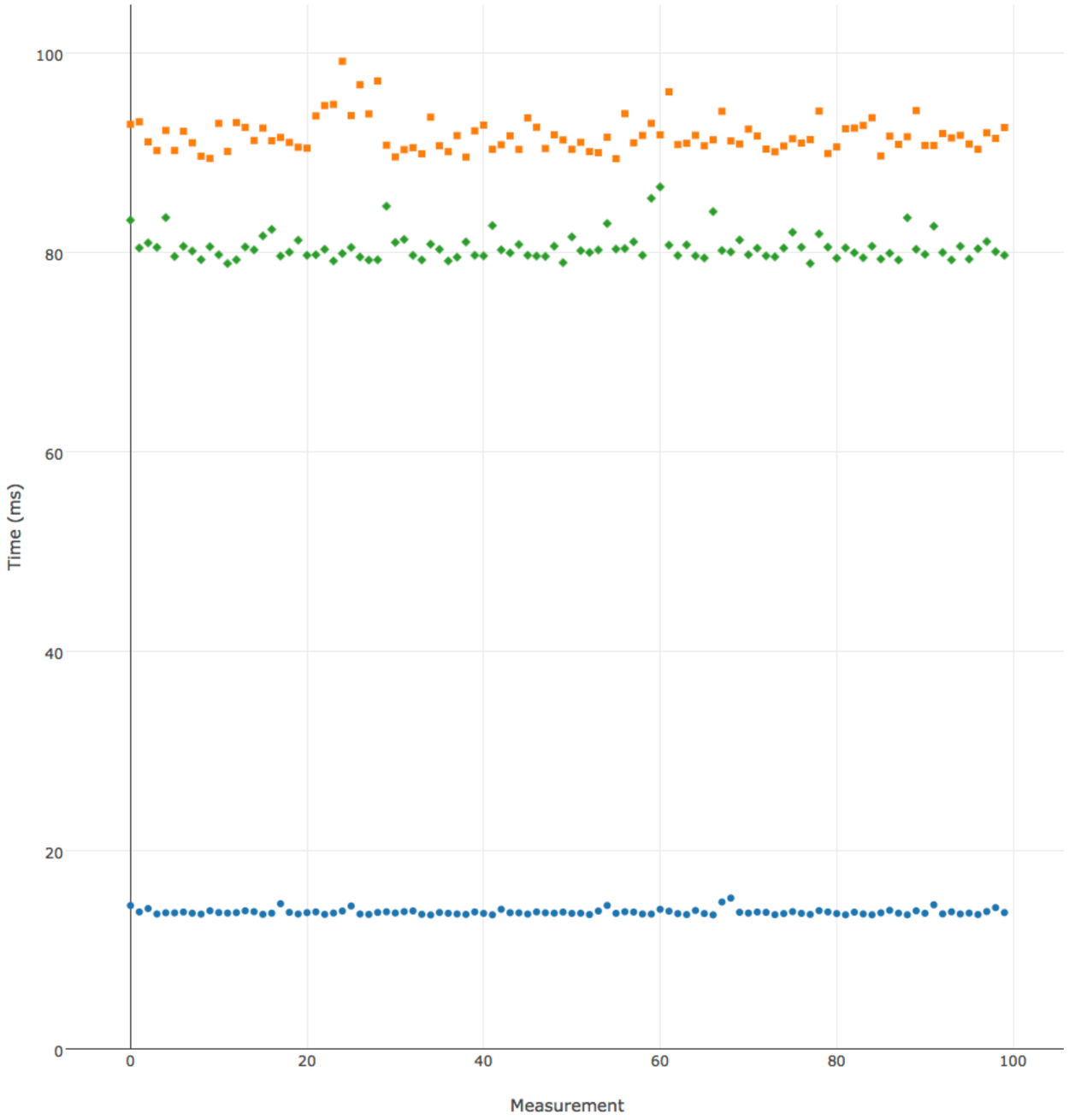
insert
100k ints



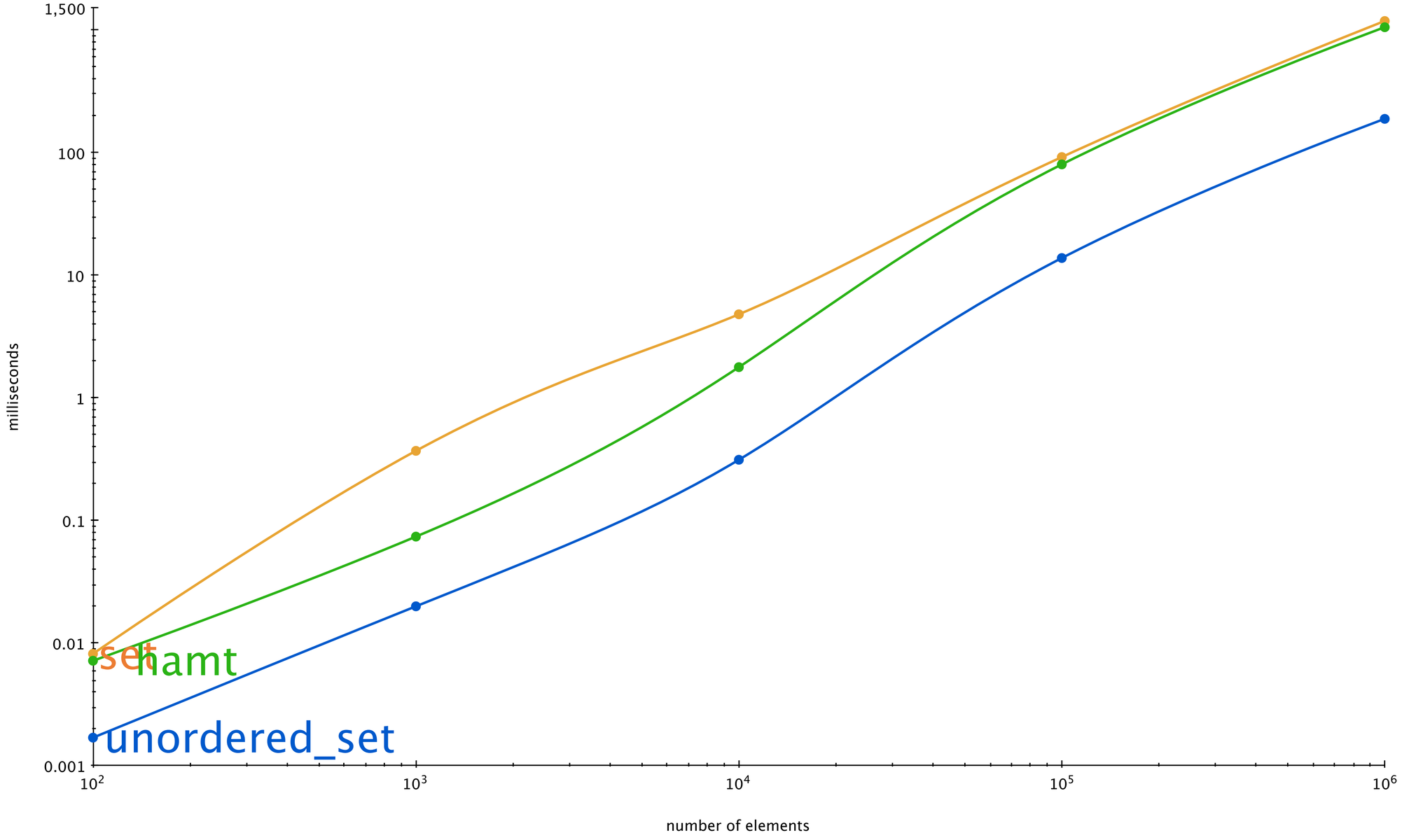
- set<int>::insert
- unordered_set<int>::insert
- hamt<int>::insert

find
100k ints



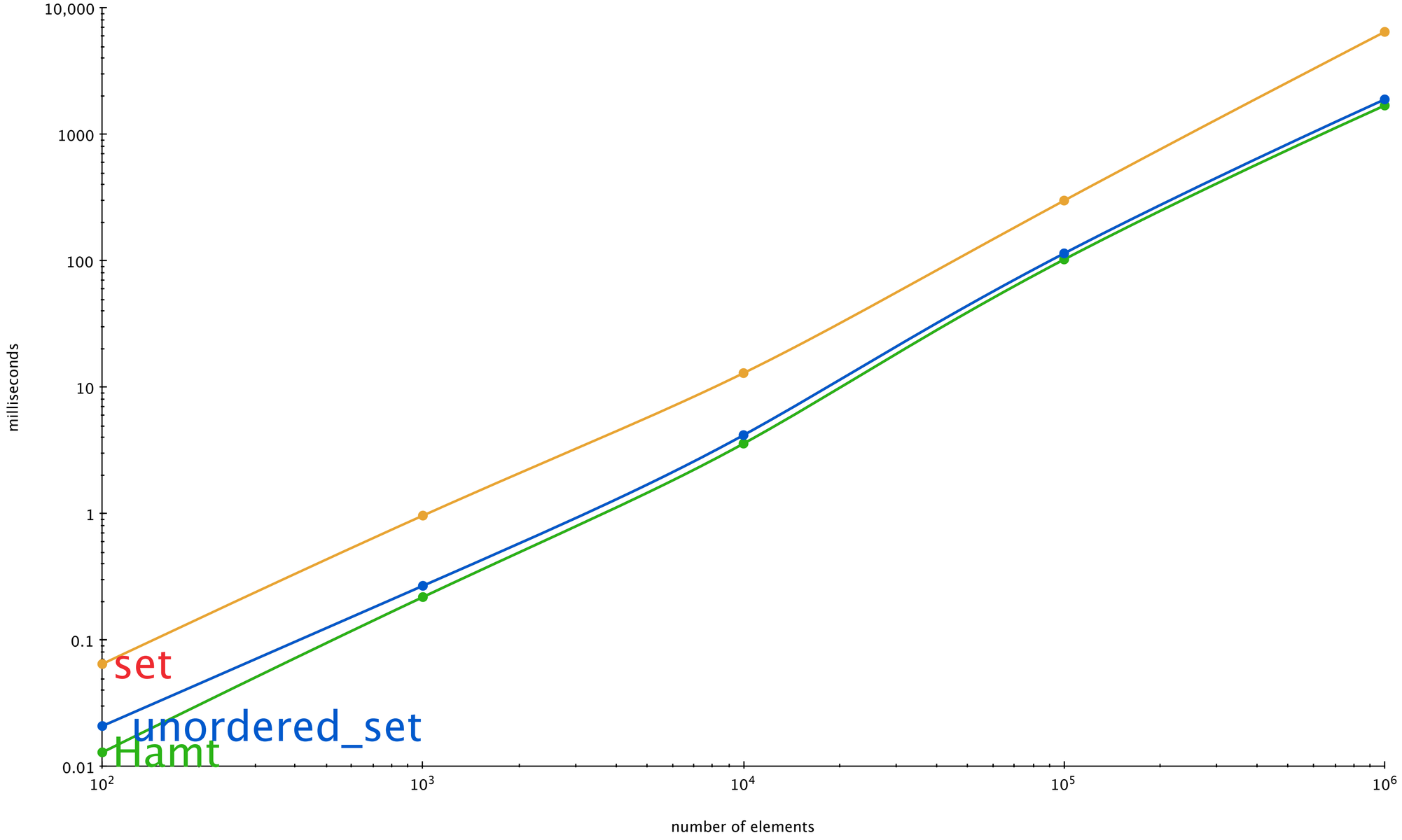


find
100-1m
ints



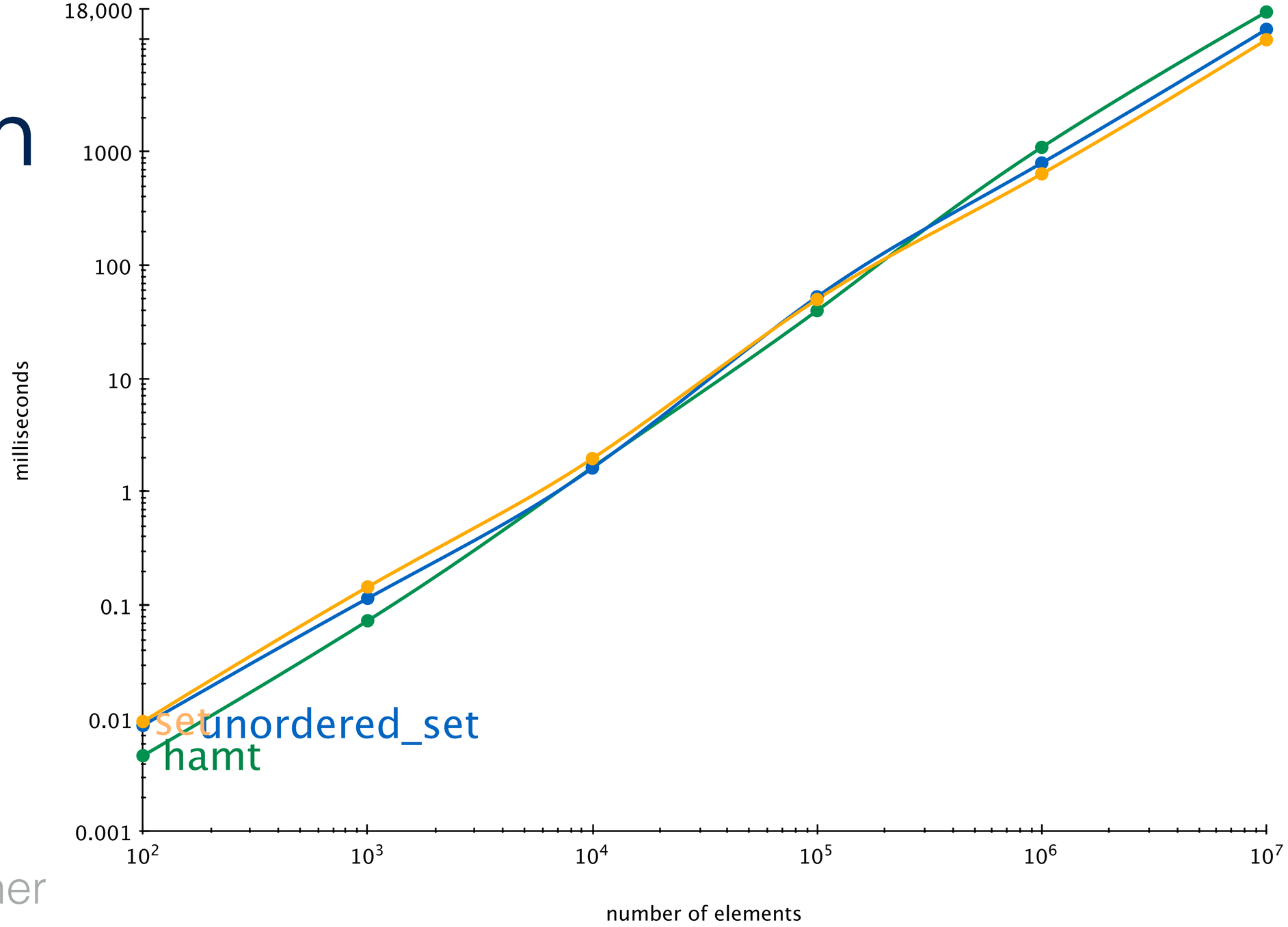
OmniGraphSketcher

find
100-1m strings



OmniGraphSketcher

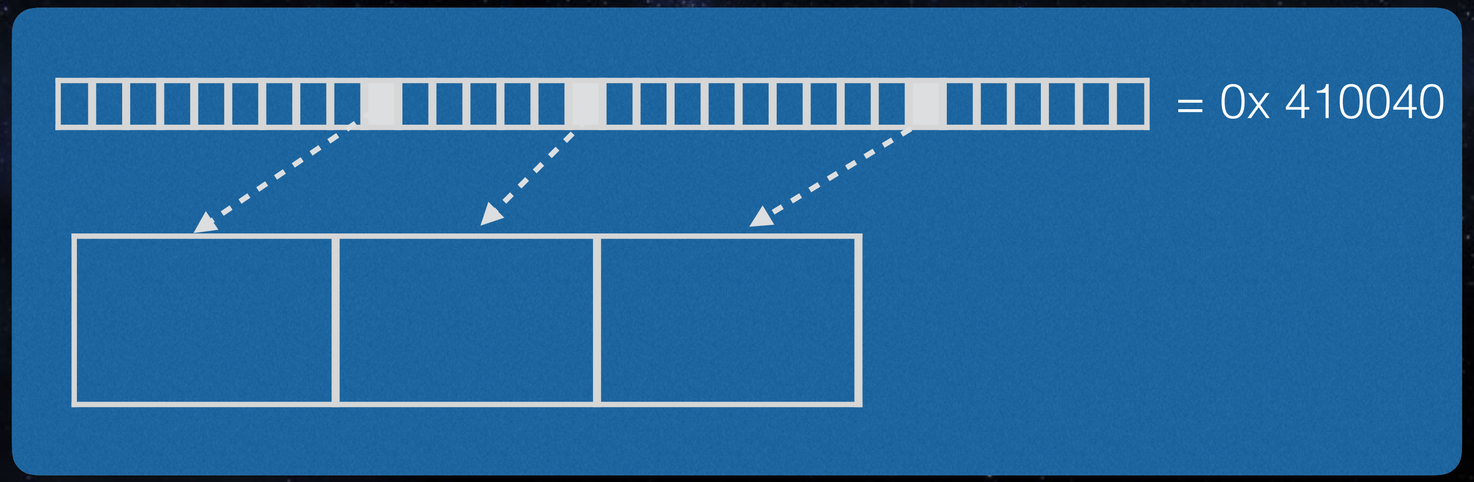
find 100-10m hashes

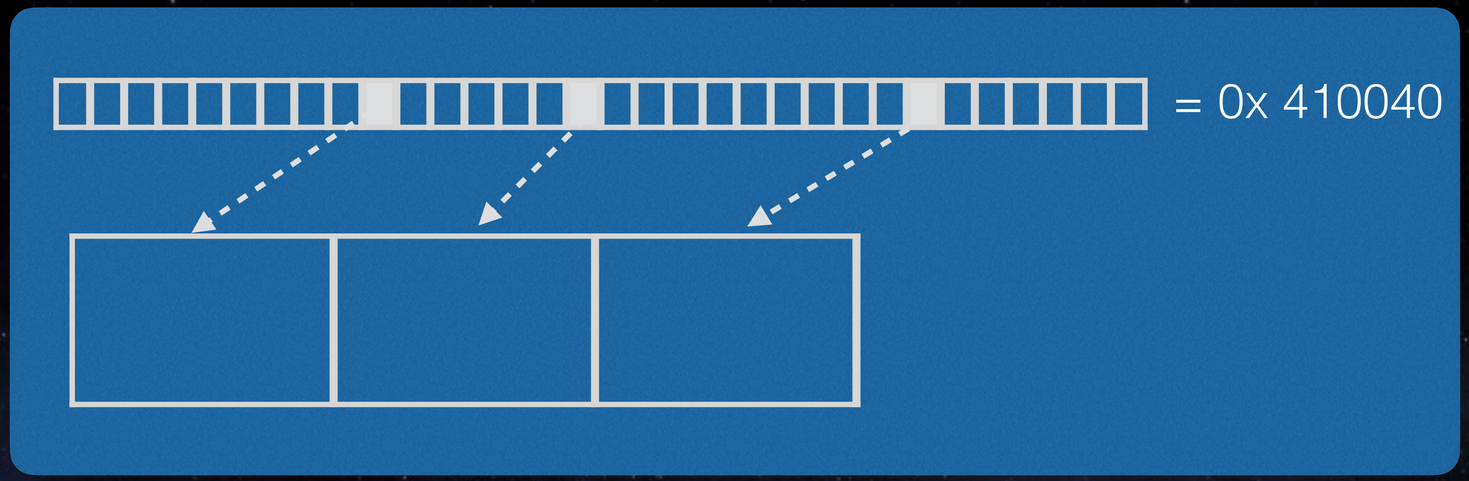


OmniGraphSketcher

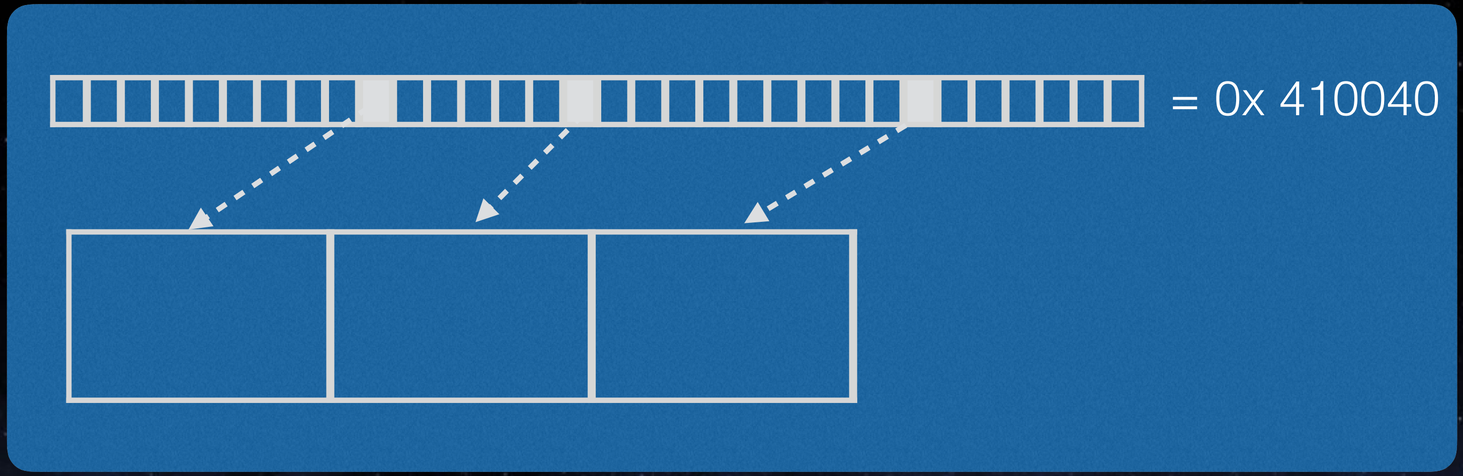
code walkthrough

```
detail::chunked_hash newHash,
std::unique_ptr<leaf_node<T>> &&newLeaf ) -> std::unique_ptr<branch_node<T>> {

    if( existingHash.chunk == newHash.chunk ) {
        auto newChildBranch = extend(existingHash + 1, existingLeaf, newHash + 1, std::move(newLeaf));
        auto newParentBranch = branch_node<T>::create_single(sparse_index(newHash.chunk), newChildBranch.get());
        newChildBranch.release();
        return newParentBranch;
    }
    else {
        auto newBranch = branch_node<T>::create_pair(sparse_index(existingHash.chunk), existingLeaf,
                                                     sparse_index(newHash.chunk), newLeaf.get());
        newLeaf.release();
        addref(existingLeaf);
        return newBranch;
    }
}

template<typename U, typename T>
static auto add_value_at_leaf(path<T> const &path, U &&value) -> branch_node<T> const* {
    auto existingLeaf = path.leaf();
    // If the value already exists we're done
    if( existingLeaf->find( value )) {
        return nullptr;
    }

    detail::chunked_hash existingHash( existingLeaf->hash() );

    // If hash matches then add an extra value to the existing leaf node
    if( existingHash.hash == path.whole_hash() ) {


        auto newLeaf = existingLeaf->with_appended_value(value);
        auto newBranch = path.last_branch()->with_replaced
                (sparse_index(path.hash_chunk()),
                newLeaf.get());

    newLeaf.release();
    auto newRoot = path.rewrite( newBranch.get());
```


= 0x 410040

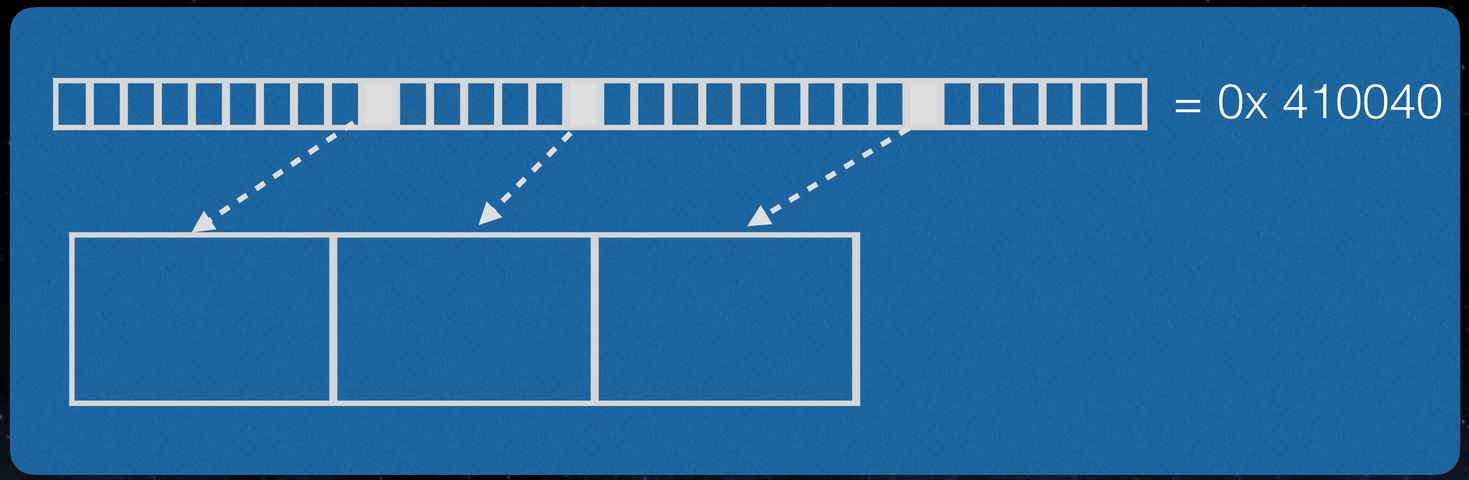



```
inline auto count_set_bits( uint32_t i ) {
    i = i - ((i >> 1) & 0x55555555);
    i = (i & 0x33333333) + ((i >> 2) & 0x33333333);
    return static_cast<uint8_t>( ((i + (i >> 4) & 0xF0F0F0F) * 0x1010101) >> 24 );
}
```




```
// adapted from `http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel`
// - could be substituted for an assembler instruction if available?
// See also: `http://stackoverflow.com/questions/109023/how-to-count-the-number-of-set-bit
```

```
inline auto count_set_bits( uint32_t i ) {
    i = i - ((i >> 1) & 0x55555555);
    i = (i & 0x33333333) + ((i >> 2) & 0x33333333);
    return static_cast<uint8_t>( ((i + (i >> 4) & 0xF0F0F0F) * 0x1010101) >> 24 );
}
```




```
inline auto count_set_bits_popcount( unsigned int i ) {
    return __builtin_popcount( i );
}
```

```
auto toCompact( size_t bitmap ) const {
    auto lowMask = bit_position()-1;
    return compact_index(
            detail::count_set_bits_popcount( static_cast<unsigned int>( bitmap & lowMask ) ) );
}
```

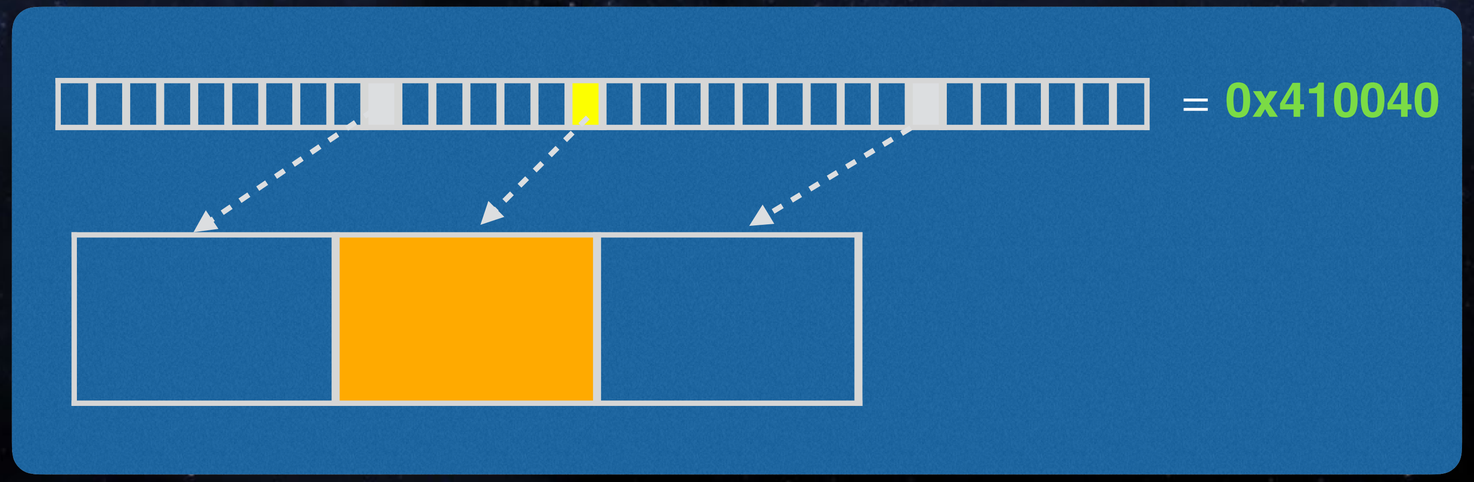
```
// "Strong typedef" for unsigned int that represents an index into a sparse array,
// along with a method for obtaining the compact index, given a bitmap
class sparse_index {
    size_t m_value;
public:
    explicit sparse_index( size_t value ) : m_value( value )  {}

    auto value() const { return m_value; }
    auto bit_position() const { return 1 << m_value; }

    auto toCompact( size_t bitmap ) const {
        auto lowMask = bit_position()-1;
        return compact_index(
                detail::count_set_bits_popcount( static_cast<unsigned int>( bitmap & lowMask ) ) );
    }
};
```

```
// "Strong typedef" for unsigned int that represents an index into a sparse array,
// along with a method for obtaining the compact index, given a bitmap
class sparse_index {
    size_t m_value;
public:
    explicit sparse_index( size_t value ) : m_value( value )  {}

    auto value() const { return m_value; }
    auto bit_position() const { return 1 << m_value; }

    auto toCompact( size_t bitmap ) const {
        auto lowMask = bit_position()-1;
        return compact_index(
                detail::count_set_bits_popcount( static_cast<unsigned int>( bitmap & lowMask ) ) );
    }
};
```

```
// "Strong typedef" for unsigned int that represents a compact index into the physical
// backing array of a sparse array
class compact_index {
    size_t m_value;
public:
    explicit compact_index( size_t value ) : m_value( value ) {}

    auto value() const { return m_value; }
};
```

```
// "Strong typedef" for unsigned int that represents an index into a sparse array,
// along with a method for obtaining the compact index, given a bitmap
class sparse_index {
    size_t m_value;
public:
    explicit sparse_index( size_t value ) : m_value( value )  {}

    auto value() const { return m_value; }
    auto bit_position() const { return 1 << m_value; }

    auto toCompact( size_t bitmap ) const {
        auto lowMask = bit_position()-1;
        return compact_index(
                detail::count_set_bits_popcount( static_cast<unsigned int>( bitmap & lowMask ) ) );
    }
};
```

```
auto bitmap = 0x410040;
auto sparse = sparse_index( 16 );
auto compact = sparse.to_compact( bitmap );
```



```
struct chunked_hash {
    size_t hash;
    size_t shiftedHash;
    size_t chunk;

    explicit chunked_hash( size_t hash )
    :   hash( hash ),
        shiftedHash( hash ),
        chunk( shiftedHash & chunkMask )
    {}

    chunked_hash& operator ++() {
        shiftedHash >>= bitsPerChunk;
        chunk = shiftedHash & chunkMask;
        return *this;
    }
    chunked_hash& operator += ( int chunks ) {
        shiftedHash >>= (bitsPerChunk*chunks);
        chunk = shiftedHash & chunkMask;
        return *this;
    }
    chunked_hash operator + ( int chunks ) {
        chunked_hash newChunkedHash( *this );
        newChunkedHash += chunks;
        return newChunkedHash;
    }
};
```

00110101101001001110100110**11011**

00000001101011010010011101001 10

26

```
enum class node_type { branch, leaf };
```

Non-virtual polymorphism

```
struct node {
    mutable std::atomic<size_t> m_refCount { 1 };
    node_type m_type;

    node() = delete;
    node( node const& ) = delete;
    node( node&& ) = delete;
    node& operator=( node const& ) = delete;
    node& operator=( node&& ) = delete;
```

Internal ref-counting

Non-copyable

```
    explicit node( node_type type ) : m_type( type ) {}
    ~node() = default;
};
```

ref-counting is atomic

```
inline void addref(node const *p) {
    std::atomic_fetch_add_explicit (&p->m_refCount, size_t(1), std::memory_order_relaxed);
}

template<typename NodeT>
inline void release( NodeT const* p ) {
    if( std::atomic_fetch_sub_explicit (&p->m_refCount, size_t(1), std::memory_order_release) == 1 ) {
        std::atomic_thread_fence( std::memory_order_acquire );
        std::default_delete<NodeT>()( const_cast<NodeT *>( p ) );
    }
}
```

```
template<typename T>
class leaf_node : public node {
    friend class std::default_delete<leaf_node>;

    size_t m_size;
    size_t m_hash;
    union { ;
        T m_values[1];
    };

    leaf_node( size_t size, size_t hash )
    :   node( node_type::leaf ),
        m_size( size ),
        m_hash( hash )
    {}

    ~leaf_node() {
        for( size_t i=0; i < m_size; ++i )
            m_values[i].~T();
    }
```

custom global deleter

uninitialised array

explicit destruction of array elements

```
// Calculates the raw storage size for a leaf type that
// contains size elements in the array
static constexpr auto storage_size(size_t size) {
    return sizeof(leaf_node) + sizeof(T*[size-1]);
}

// Creates a new leaf_node type with enough additional storage for
// size items - but does not populate the array
static auto create_unpopulated( size_t size, size_t hash ) {
    assert( size >=1 );
    auto temp = std::make_unique<unsigned char[]>(storage_size(size) );
    auto leaf_ptr = new(temp.get()) leaf_node( size, hash );
    temp.release();
    return std::unique_ptr<leaf_node>( leaf_ptr );
}
```

```
template<typename U>
static auto create( U &&value, size_t hash ) -> std::unique_ptr<leaf_node> {
    auto leaf = create_unpopulated(1, hash);
    new (&leaf->m_values[0]) T( std::forward<U>( value ) );
    return leaf;
}

template<typename U>
auto with_appended_value(U &&newValue) const {
    auto newLeaf = create_unpopulated(m_size + 1, m_hash);
    for( size_t i=0; i < m_size; ++i )
        new (&newLeaf->m_values[i]) T( m_values[i] );
    new (&newLeaf->m_values[m_size]) T( std::forward<U>( newValue ) );
    return newLeaf;
}
```

```
template<typename T>
class branch_node : public node {
    friend class std::default_delete<branch_node>;

    size_t m_size;
    size_t m_bitmap; // set bits indicate the indexed element is a branch or leaf value

    union {
        node const *m_children[1];
    };
```

```
~branch_node() {
    auto len = size();
    for( size_t i = 0; i < len; ++i ) {
        auto node = m_children[i];
        if( node->m_type == node_type::branch )
            release( static_cast<branch_node<T> const*>( node ) );
        else
            release( static_cast<leaf_node<T> const*>( node ) );
    }
}
```

```
// Calculates the raw storage size for a node type that
// contains size elements in the array
static constexpr auto storage_size(size_t size) {
    return sizeof(branch_node) + sizeof(node*)*(size-1);
}

// Creates a new branch_node type with enough additional storage for
// size items - but does not populate the array
static auto create_unpopulated( size_t size, size_t bitmap ) {
    assert( size <= 32 );
    auto temp = std::make_unique<unsigned char[]>(storage_size(size) );

    auto node_ptr = new(temp.get()) branch_node( size, bitmap );
    temp.release();
    return std::unique_ptr<branch_node>( node_ptr );
}
```

```
static auto create_empty() -> std::unique_ptr<branch_node> {
    auto node = create_unpopulated( 1, 0 );
    node->m_size = 0;
    return node;
}

static auto create_single(sparse_index index, node const *child) -> std::unique_ptr<branch_node> {
    auto node = create_unpopulated( 1, static_cast<size_t>( index.bit_position() ) );
    node->m_children[0] = child;
    return node;
}
```

```
static auto create_pair(sparse_index index1, leaf_node<T> const *leaf1, sparse_index index2,
```

```
static auto create_single(sparse_index index, node const *child) -> std::unique_ptr<branch_node> {
    auto node = create_unpopulated( 1, static_cast<size_t>( index.bit_position() ) );
    node->m_children[0] = child;
    return node;
}
```

```
static auto create_pair(sparse_index index1, leaf_node<T> const *leaf1, sparse_index index2,
                        leaf_node<T> const *leaf2) -> std::unique_ptr<branch_node> {
    auto bitmap = static_cast<size_t>( index1.bit_position() | index2.bit_position() );
    auto node = create_unpopulated( 2, bitmap );
    auto children = &node->m_children[0];
    if( index1.value() >  index2.value() ) {
        children[0] = leaf2;
        children[1] = leaf1;
    }
    else {
        children[0] = leaf1;
        children[1] = leaf2;
    }
    return node;
}
```

```
auto with_inserted(sparse_index sparseIndex, node const *child) const -> std::unique_ptr<branch_node> {
```

```
    return node;
}
```

```
auto with_inserted(sparse_index sparseIndex, node const *child) const -> std::unique_ptr<branch_node> {
    auto originalSize = size();
    auto bitmap = m_bitmap | sparseIndex.bit_position();

    // If adding new we need to offset later nodes
    assert( ( m_bitmap & sparseIndex.bit_position() ) == 0 );

    auto node = create_unpopulated( originalSize + 1, bitmap );

    auto splitPoint = sparseIndex.toCompact( m_bitmap ).value();

    for( size_t i = 0; i < splitPoint; ++i ) {
        auto sharedNode = node->m_children[i] = m_children[i];
        addref(sharedNode);
    }

    node->m_children[splitPoint] = child;

    for( size_t i = splitPoint; i < originalSize; ++i ) {
        auto sharedNode = node->m_children[i+1] = m_children[i];
        addref(sharedNode);
    }
    return node;
}
```

```
}
```

```
auto with_replaced(sparse_index sparseIndex, node const *child) const -> std::unique_ptr<branch_node> {
    auto originalSize = size();
    auto bitmap = m_bitmap | sparseIndex.bit_position();

    // If replacing a node we overwrite existing in place
    assert( ( m_bitmap & sparseIndex.bit_position() ) != 0 );

    auto node = create_unpopulated( originalSize, bitmap );

    auto splitPoint = sparseIndex.toCompact( m_bitmap ).value();

    for( size_t i = 0; i < splitPoint; ++i ) {
        auto sharedNode = node->m_children[i] = m_children[i];
        addref(sharedNode);
    }

    node->m_children[splitPoint] = child;

    for( size_t i = splitPoint+1; i < originalSize; ++i ) {
        auto sharedNode = node->m_children[i] = m_children[i];
        addref(sharedNode);
    }
    return node;
}
```

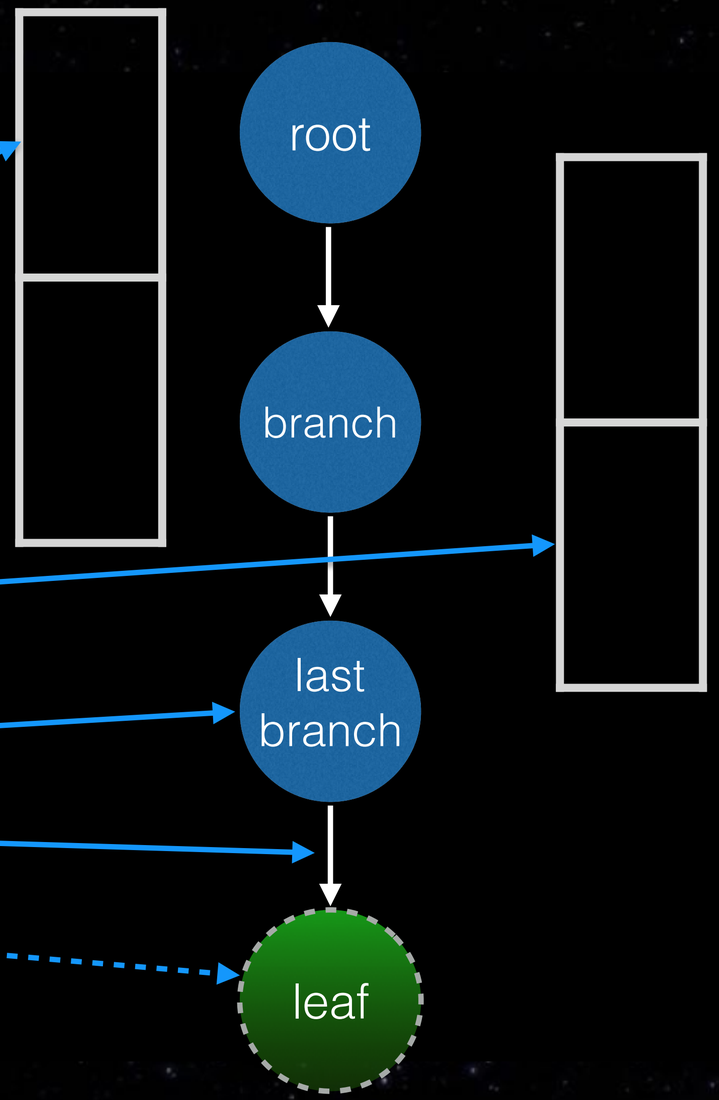
```
}
```

```
auto with_replaced(sparse_index sparseIndex, node const *child) const -> std::unique_ptr<branch_node> {
    auto originalSize = size();
    auto bitmap = m_bitmap | sparseIndex.bit_position();

    // If replacing a node we overwrite existing in place
    assert( ( m_bitmap & sparseIndex.bit_position() ) != 0 );

    auto node = create_unpopulated( originalSize, bitmap );

    auto splitPoint = sparseIndex.toCompact( m_bitmap ).value();

    for( size_t i = 0; i < splitPoint; ++i ) {
        auto sharedNode = node->m_children[i] = m_children[i];
        addref(sharedNode);
    }

    node->m_children[splitPoint] = child;

    for( size_t i = splitPoint+1; i < originalSize; ++i ) {
        auto sharedNode = node->m_children[i] = m_children[i];
        addref(sharedNode);
    }
    return node;
}
```

```
auto get_at(compact_index compactIndex) const {
    return m_children[compactIndex.value()];
}
auto get_at(sparse_index sparseIndex) const -> node const* {
    if( ( m_bitmap & sparseIndex.bit_position() ) == 0 )
        return nullptr;
    return get_at(sparseIndex.toCompact(m_bitmap));
}
```

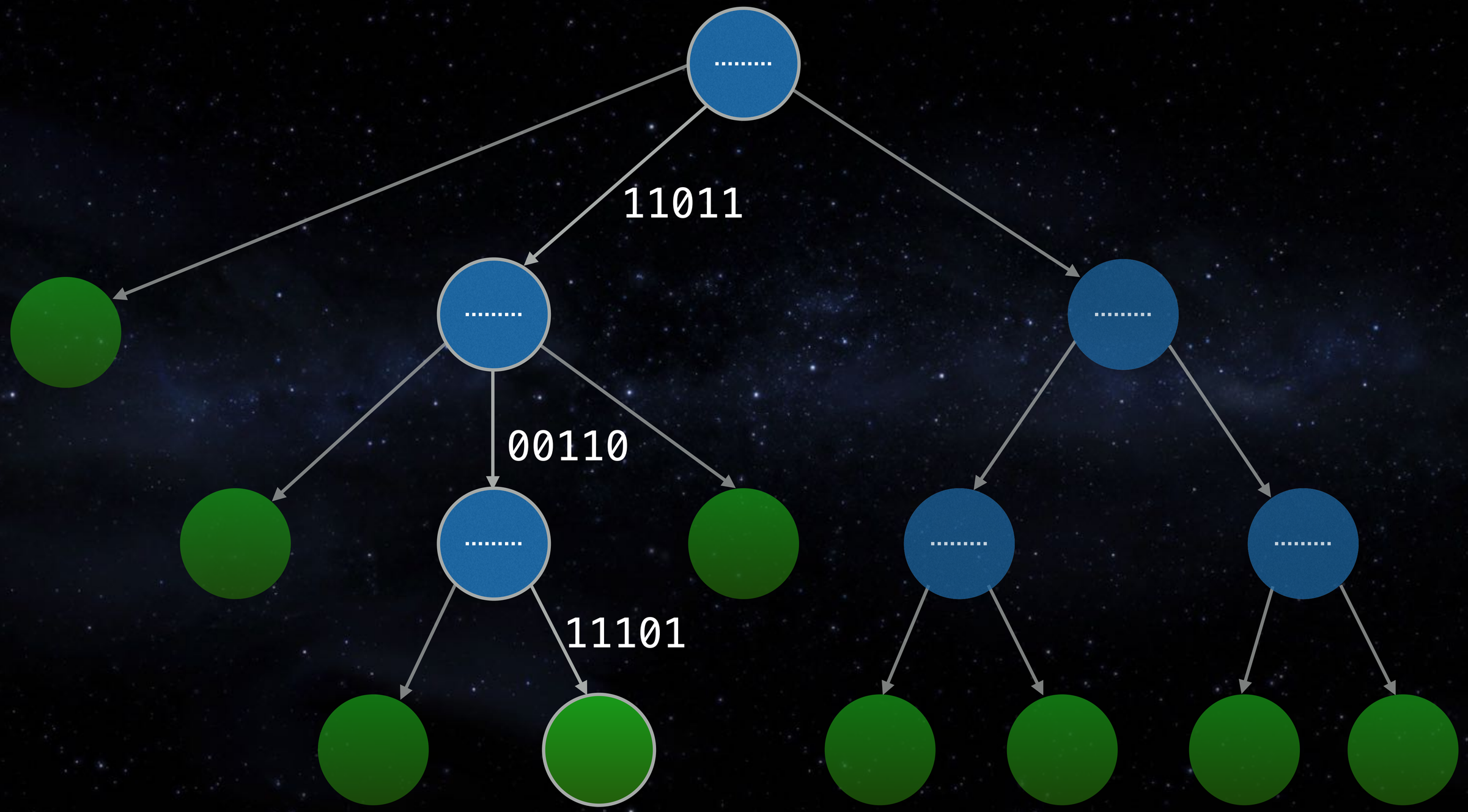
```
// Traces a path of branch_nodes and hash chunks down to either a leaf_node
// that matches the given hash, or the last matching branch_node.
template<typename T>
class path {

    branch_node<T> const *m_branches[detail::maxDepth];
    size_t m_chunks[detail::maxDepth];

    branch_node<T> const *m_lastBranch;
    detail::chunked_hash m_chunkedHash;
    leaf_node<T> const* m_leaf;
    size_t m_size = 0;
```

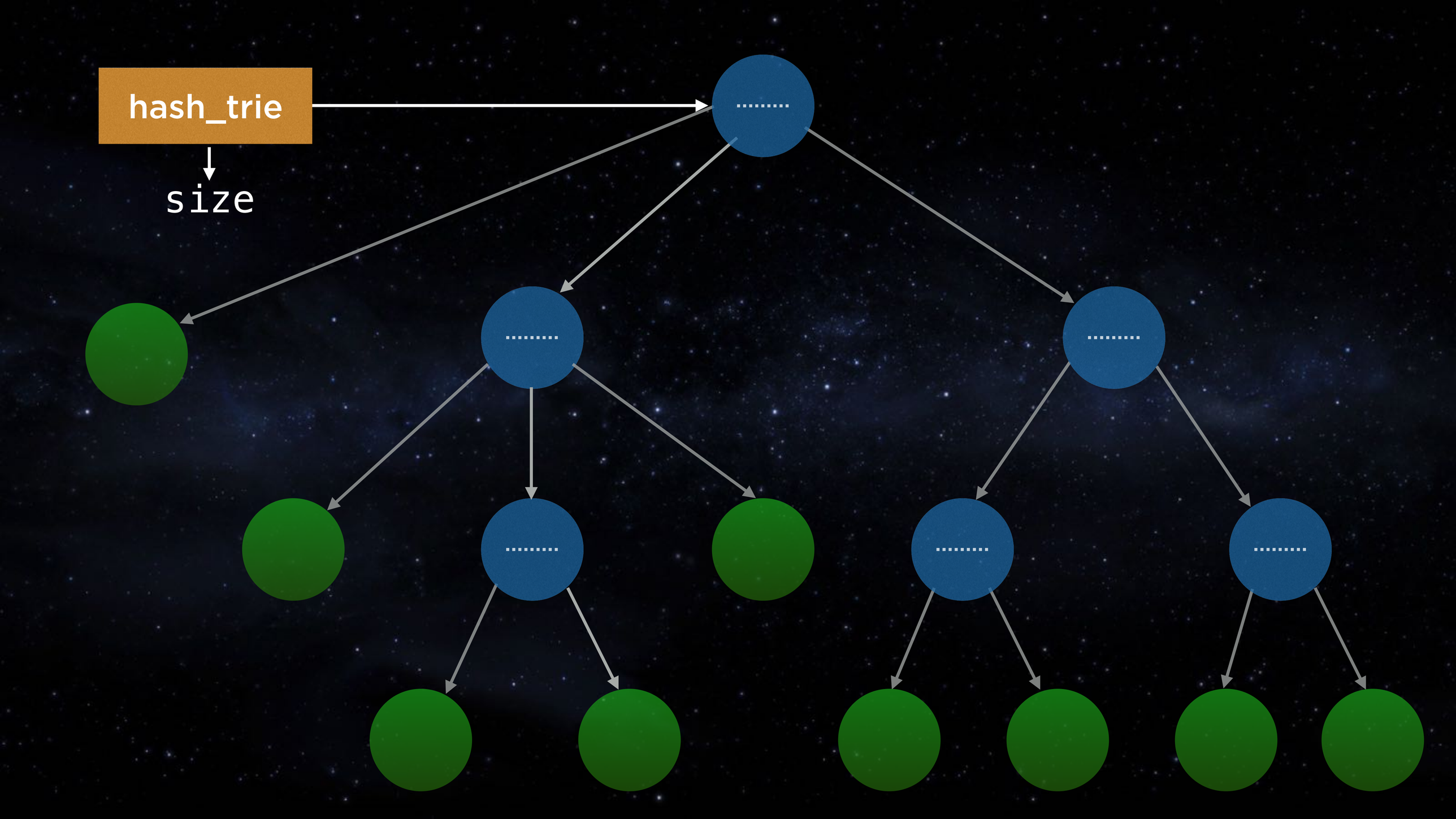
```
/ Traces a path of branch_nodes and hash chunks down
/ that matches the given hash, or the last matching
emplate<typename T>
lass path {

    branch_node<T> const *m_branches[detail::maxDepth
    size_t m_chunks[detail::maxDepth];

    branch_node<T> const *m_lastBranch;
    detail::chunked_hash m_chunkedHash;
    leaf_node<T> const* m_leaf;
    size_t m_size = 0;
```



```
path( T const& value, branch_node<T> const* root )
: m_chunkedHash( std::hash<T>()( value ) ) {
    size_t size = 0;
    auto currentBranch = root;
    auto chunk = m_chunkedHash.chunk;
    auto nextNode = currentBranch->get_at( sparse_index( chunk ) );

    while( nextNode && nextNode->m_type == node_type::branch ) {
        m_branches[size] = currentBranch;
        m_chunks[size] = chunk;

        currentBranch = static_cast<branch_node<T> const*>( nextNode );

        ++size;
        ++m_chunkedHash;

        chunk = m_chunkedHash.chunk;
        nextNode = currentBranch->get_at( sparse_index( chunk ) );
    };
    m_leaf = static_cast<leaf_node<T> const*>( nextNode );
    m_lastBranch = currentBranch;
    m_size = size;
}
```

```
auto rewrite( branch_node<T> const* newBranch ) const -> branch_node<T> const* {
    auto currentBranch = newBranch;

    for( auto i = m_size; i > 0; --i ) {
        sparse_index index( m_chunks[i - 1] );
        auto parent = m_branches[i - 1]->with_replaced( index, currentBranch );
        currentBranch = parent.release();
    }
    return currentBranch;
}
```


.....
11011
.....
.....
00110
.....
.....
.....
11101


hash_trie
size

```
template<typename T>
struct hash_trie_data {
    branch_node<T> const* m_root;
    size_t m_size;
};
```

```
template<typename T>
class hash_trie {

    hash_trie_data<T> m_data;
    friend shared_hash_trie<T>;

    static auto makeEmptyData() -> hash_trie_data<T> {
        return { branch_node<T>::create_empty().release(), 0 };
    }

public:
    hash_trie() : m_data( makeEmptyData() ) {}

    hash_trie( hash_trie_data<T> const& data ) : m_data( data ) {
        addref( m_data.m_root );
    }
    hash_trie( hash_trie<T> const& other ) : hash_trie( other.m_data ) {}
    hash_trie( hash_trie<T>&& other ) : hash_trie() {
        swap( other );
    }

    ~hash_trie() {
        release( m_data.m_root );
    }
```

```
hash_trie& operator = ( hash_trie other ) {
    swap( other );
    return *this;
}
hash_trie& operator = ( hash_trie&& other ) {
    if( !empty() ) {
        hash_trie emptyTemp;
        swap( emptyTemp );
    }
    swap( other );
    return *this;
}

void swap( hash_trie& other ) {
    std::swap( m_data.m_root, other.m_data.m_root );
    std::swap( m_data.m_size, other.m_data.m_size );
}
```

```
template<typename U>
auto insert( U &&value ) {
    if( auto newRoot = inserted( m_data.m_root, std::forward<U>(value) ) ) {
        release( m_data.m_root );
        m_data = { newRoot, size()+1 };
    }
}
```

```
template<typename U, typename T>
auto inserted( branch_node<T> const* root, U &&value) -> branch_node<T> const* {
    static_assert( sizeof( T( value ) ) > 0, "value must be convertible to T" );

    path<T> path( value, root );

    return path.leaf()
           ? add_value_at_leaf( path, std::forward<U>(value) )
           : add_value_at_currently_unset_position
                   ( path, leaf_node<T>::create(std::forward<U>(value), path.whole_hash() ) );
}
```

```
template<typename U, typename T>
auto inserted( branch_node<T> const* root, U &&value) -> branch_node<T> const* {
    static_assert( sizeof( T( value ) ) > 0, "value must be convertible to T" );

    path<T> path( value, root );

    return path.leaf()
           ? add_value_at_leaf( path, std::forward<U>(value) )
           : add_value_at_currently_unset_position
                  ( path, leaf_node<T>::create(std::forward<U>(value), path.whole_hash() ) );
}
```

```
template<typename T>
auto add_value_at_currently_unset_position(path<T> const &path, std::unique_ptr<leaf_node<T>> &&leaf) {
    auto newBranch = path.last_branch()->with_inserted(sparse_index(path.hash_chunk()), leaf.get());
    leaf.release();
    auto newRoot = path.rewrite( newBranch.get() );
    newBranch.release();
    return newRoot;
}
```

```
template<typename U, typename T>
static auto add_value_at_leaf(path<T> const &path, U &&value) -> branch_node<T> const* {
    auto existingLeaf = path.leaf();
    // If the value already exists we're done
    if( existingLeaf->find( value )) {
        return nullptr;
    }
```

```
detail::chunked_hash existingHash( existingLeaf->hash() );
```

```
// If hash matches then add an extra value to the existing leaf node
if( existingHash.hash == path.whole_hash() ) {
```

```
// Different hash, so add branches to the point they diverge
else {
```

```
template<typename T>
auto extend
        (   detail::chunked_hash existingHash,
            leaf_node<T> const *existingLeaf,
            detail::chunked_hash newHash,
            std::unique_ptr<leaf_node<T>> &&newLeaf ) -> std::unique_ptr<branch_node<T>> {
```

```
TEST_CASE( "transaction commit" ) {

    using namespace hamt;

    shared_hash_trie<int> sh;

    auto trans = sh.start_transaction();

    hash_trie<int> h = trans;

    h.insert( 1 );
    h.insert( 2 );
    h.insert( 10 );

    trans.commit( h );

    hash_trie<int> h2 = sh;
    REQUIRE( h2.size() == 3 );
}
```

```
TEST_CASE( "transaction commit" ) {

    using namespace hamt;

    shared_hash_trie<int> sh;

    auto trans = sh.start_transaction();

    hash_trie<int> h = trans;

    h.insert( 1 );
    h.insert( 2 );
    h.insert( 10 );

    trans.commit( h );

    hash_trie<int> h2 = sh;
    REQUIRE( h2.size() == 3 );
}
```

```
TEST_CASE( "transaction task" ) {

    using namespace hamt;

    shared_hash_trie<int> sh;

    sh.update_with([](hash_trie<int> &h) {
        h.insert(1);
        h.insert(2);
        h.insert(10);
    });

    hash_trie<int> h2 = sh;
    REQUIRE( h2.size() == 3 );
}
```

```
template<typename T>
class shared_hash_trie {
    static_assert( std::is_trivially_copyable<hash_trie_data<T>>::value,
                   "hash_trie_data must be trivially copyable to be used atomically" );

    std::atomic<hash_trie_data<T>> m_data;
```

```
template<typename T>
class shared_hash_trie {
    static_assert( std::is_trivially_copyable<hash_trie_data<T>>::value,
                   "hash_trie_data must be trivially copyable to be used atomically" );

    std::atomic<hash_trie_data<T>> m_data;
```

```
shared_hash_trie() : m_data( makeEmptyData() ) {}

shared_hash_trie( hash_trie<T> const& hash_trie ) {
    m_data.store( hash_trie.data(), std::memory_order_relaxed );
    addref( hash_trie.data().m_root );
}

auto data() const -> hash_trie_data<T> {
    return m_data.load( std::memory_order_relaxed );
}
```

```
operator hash_trie<T>() const {
    return data();
}

auto start_transaction() -> hash_trie_transaction<T>;

template<typename L>
void update_with(L const &updateTask);

// "low level" compare-exchange wrapper - use transation
auto reset( hash_trie_data<T>& originalData,
            hash_trie_data<T>& newData ) -> bool {
    if( !m_data.compare_exchange_strong
            ( originalData, newData,
              std::memory_order_release,
              std::memory_order_relaxed ) )
        return false;

    release( originalData.m_root );
    addref( newData.m_root );
    return true;
}
```

```
template<typename T>
class hash_trie_transaction {
    hash_trie_data<T> m_baseData; // For compare-exchange
    shared_hash_trie<T>& m_shared;
```

```
template<typename L>
void update_with(L const &updateTask) {
    while( true ) {
        hash_trie<int> copy( m_baseData );
        updateTask( copy );

        // If we didn't change, don't do anything
        if( copy.data().m_root == m_baseData.m_root )
            break;

        // try to commit, and if successful we're done
        if( commit( copy ) )
            break;

        // if we couldn't commit, rebase and try again
        release( m_baseData.m_root );
        m_baseData = m_shared.data();
        addref( m_baseData.m_root );
    };
}
```

Summary:

can approach performance of hash maps

more space efficient than hash tables

copies are cheap and memory efficient

can hold multiple variations in memory at once

can be made to work nicely with concurrency

Thanks for listening

github.com/philsquared/hash_trie

https://www.youtube.com/watch?v=8hW-LT8qFT0

lampwww.epfl.ch/papers/idealhashtrees.pdf

jahej.com/alt/2011_03_22_ideal-hash-tries-an-implementation-in-c.html

@phil_nash

Presented at C++Now, Copyright © 2017

Also available as a video at:
https://www.youtube.com/watch?v=WT9kmIE3Uis