CMSC 420 – 0201 – Fall 2019
Lecture 04

Binary Search Trees

Searching

- Set of entries $\{e_1, \ldots, e_n\}$, where each entry is a key-value pair (x_i, v_i)
- Store these so that given any key x, we can efficiently retrieve the associated value v (or report that it is not present)
- Good coding versus our conventions:
  - To simplify code fragments in lecture, we will assume two fixed types, Key and Value, but these would typically be class generics, e.g., class Dictionary<K,V>
  - We will use usual comparison operators for keys (==, <=, >, etc), but these would normally be implemented using a Comparator class (e.g., compare(x,y))

Dictionary

Core Operations

- `void insert(Key x, Value v):`
  - Inserts an entry with the key-value pair (x, v)
  - We assume that keys are unique, and so if this key already exists, an error condition will be signaled (e.g., an exception will be thrown)
- `void delete(Key x):`
  - Delete the entry with x's key from the dictionary
  - If this key does not appear in the dictionary, then an error conditioned is signaled
- `Value find(Key x):`
  - Determine whether there is an entry matching x 's key in the dictionary
  - If so, it returns a reference to associated value. Otherwise, it returns a `null` reference.

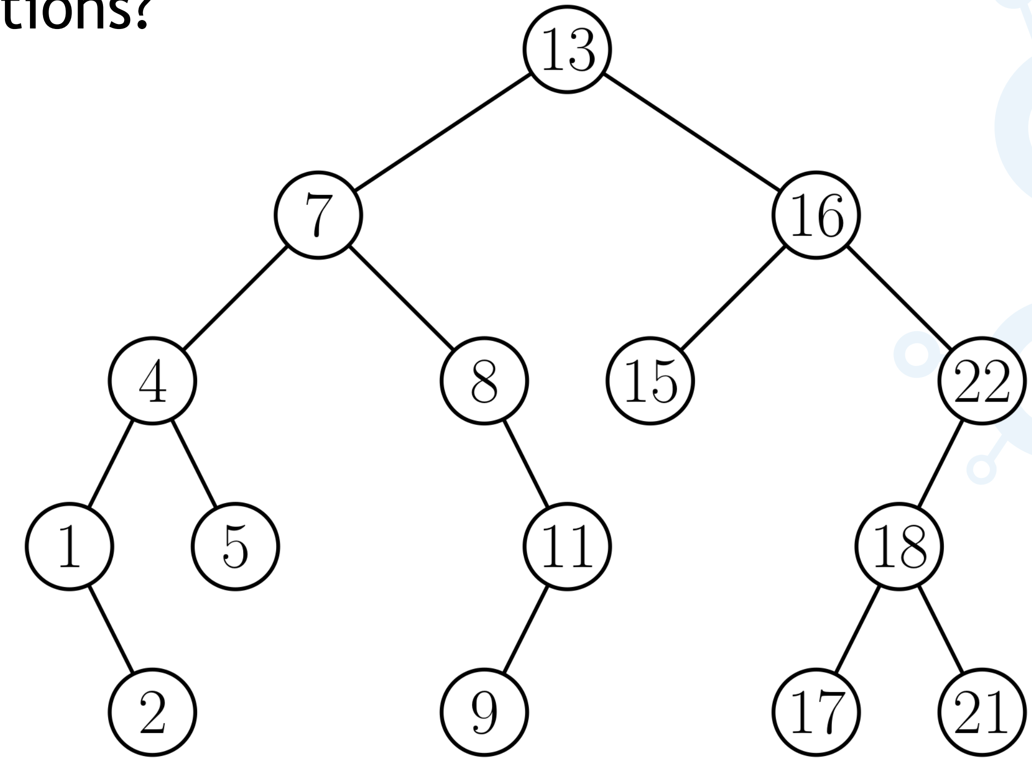
Dictionary

Sequential Allocation

- Allocate entries in an array. Simple, but not very efficient
- `Unsorted array:`
  - Insertion in $O(1)$, but still need $O(n)$ to check for duplicates
  - Find and Delete in $O(n)$
- `Sorted array:`
  - Find in $O(\log n)$ through binary search
  - Insert and Delete in $O(n)$

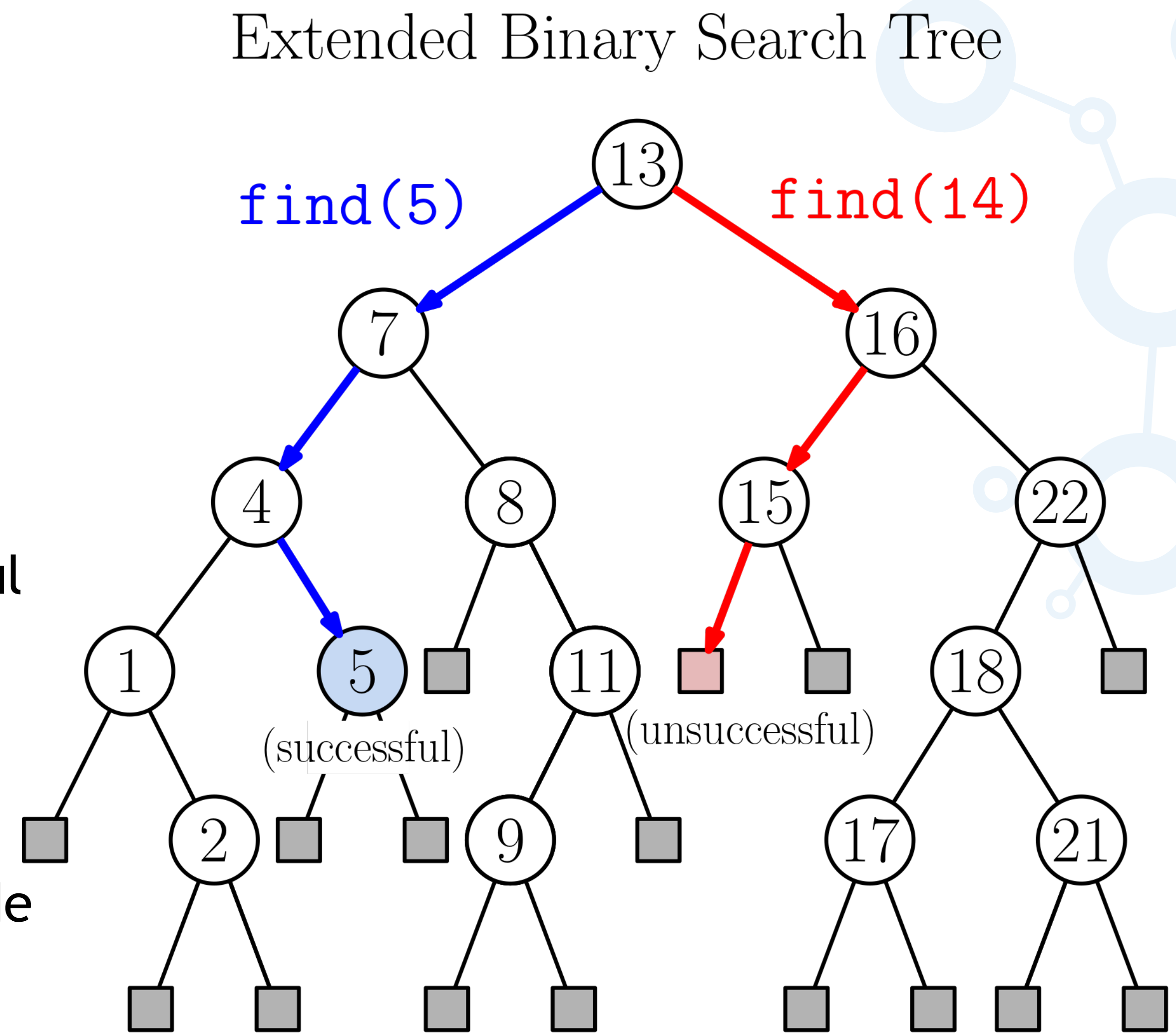
Binary Search Tree

- Can we achieve O(log n) time for all operations?
- Yes! Binary search trees
- Store entries in a binary tree, so that an inorder traversal encounters keys in ascending order



Binary Search Tree - Find

- To find a key x, start at the root
- For each node p:
  - `if (x == p.key)` - Success!
  - `if (x < p.key)` - Search `p.left`
  - `if (x > p.key)` - Search `p.right`
  - `if (p == null)` - Search is unsuccessful
- Can view the tree "as if" it were an extended tree:
  - Successful search ends at internal node
  - Unsuccessful search ends at external node



Binary Search Tree - Find

Recursive formulation

```
Value find(Key x, BinaryNode p) {
       if (p == null) return null;                  // unsuccessful search
       else if (x < p.key)                          // x is smaller?
           return find(x, p.left);                  // ... search left
       else if (x > p.key)                          // x is larger?
           return find(x, p.right);                 // ... search right
       else return p.value;                         // successful search
   }
```

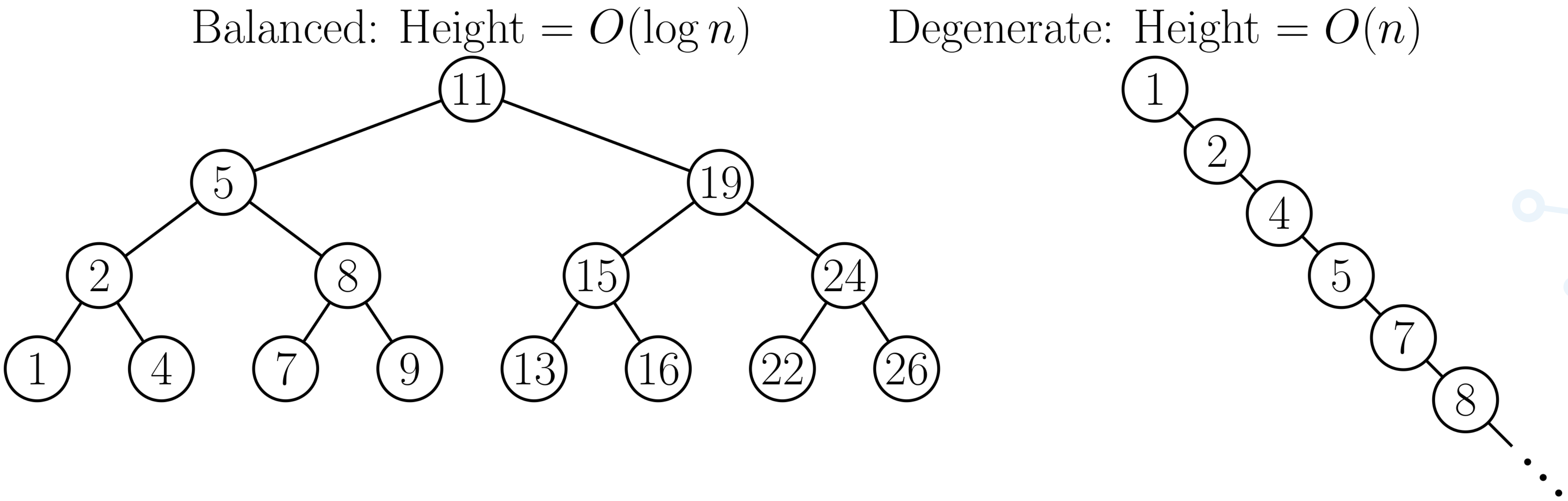
Binary Search Tree - Find

Iterative formulation

```
Value find(Key x) {
      BinaryNode p = root;                   // start at the root
      while (p != null) {                    // until we fall out of tree
          if (x < p.key) p = p.left;         // x is smaller? ...search left
          else if (x > p.key) p = p.right;   // x is larger? ...search right
          else return p.value;               // successful search
      }
      return null;                           // unsuccessful search
  }
```

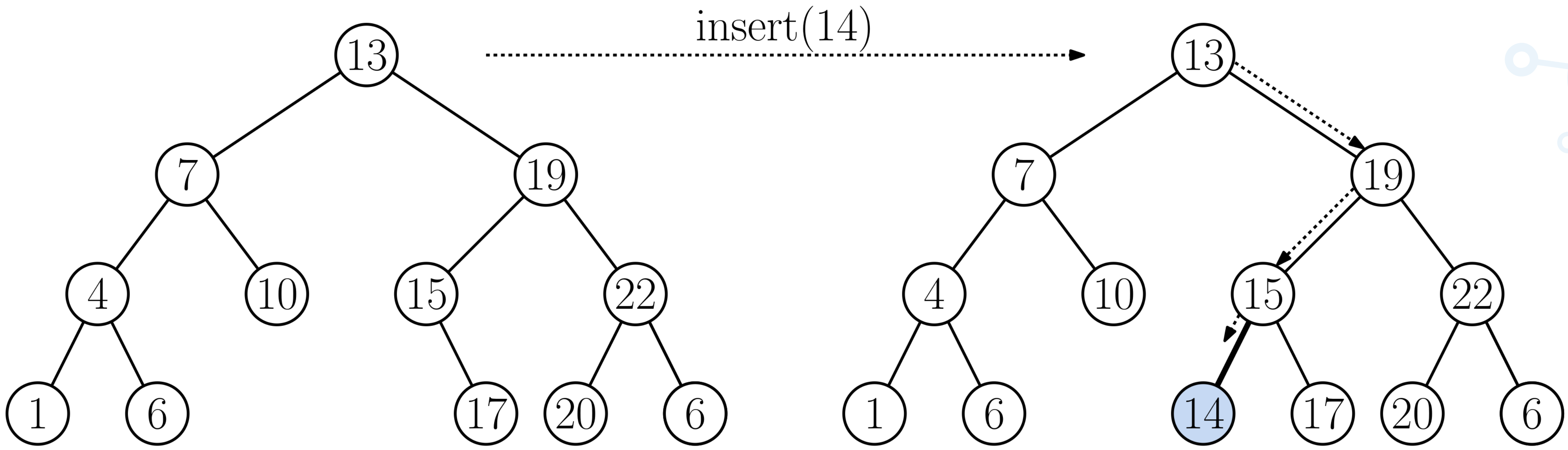
Binary Search Trees

Search time depends on the height of the tree



Binary Search Tree - Insert

- To insert a new key-value pair, first find the key
- If you find it, duplicate-key error
- Otherwise, insert the new node at the spot where you "fall out" of the tree



Binary Search Tree - Insert

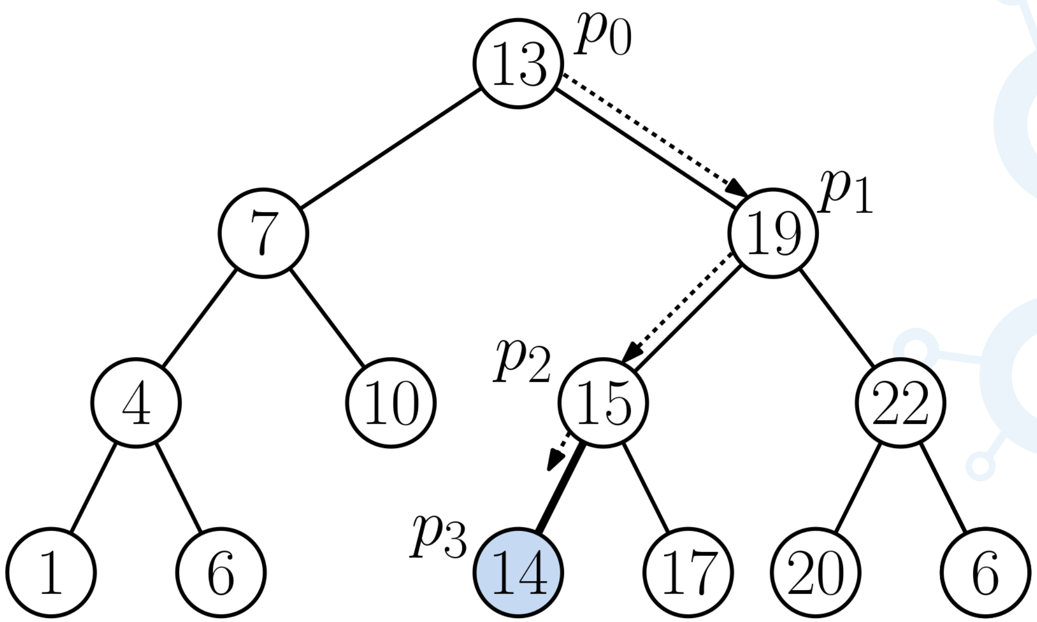
```
BinaryNode insert(Key x, Value v, BinaryNode p) {
       if (p == null)                             // fell out of the tree?
           p = new BinaryNode(x, v, null, null);  // ... create new leaf here
       else if (x < p.key)                        // x is smaller?
           p.left  = insert(x, v, p.left);        // ...insert left
       else if (x > p.key)                        // x is larger?
           p.right = insert(x, v, p.right);       // ...insert right
       else throw DuplicateKeyException;          // x is equal ...duplicate!
       return p                                   // ref to current node
   }
```

Binary Search Tree - Insert

- Beware: This code is tricky!
- In the statement:

```
p.left  = insert(x, v, p.left);
```

- Note that the return value from `insert` is used to modify the parent's child pointer
- A reference to newly created node p_3 is inserted into the left-child link of p_2



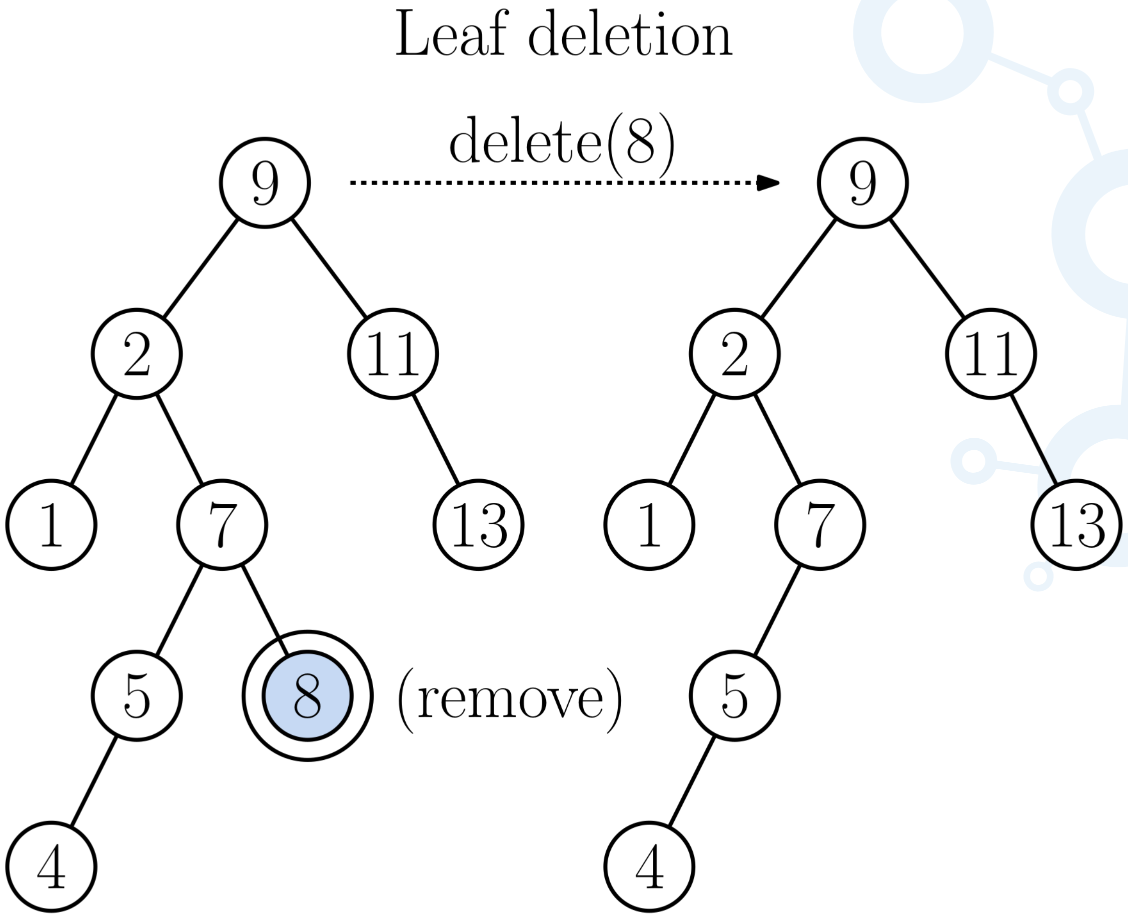
Binary Search Tree - Delete

- First find the node to delete, and then remove it, and fix the links
- Deletion is more complex than insertion
  - Insertion creates a new leaf, but any node may be deleted
- Cases:
  - Deleting a leaf (zero children)
  - Deleting a node with one child
  - Deleting a node with two children

Binary Search Tree - Delete

Leaf deletion (zero children)

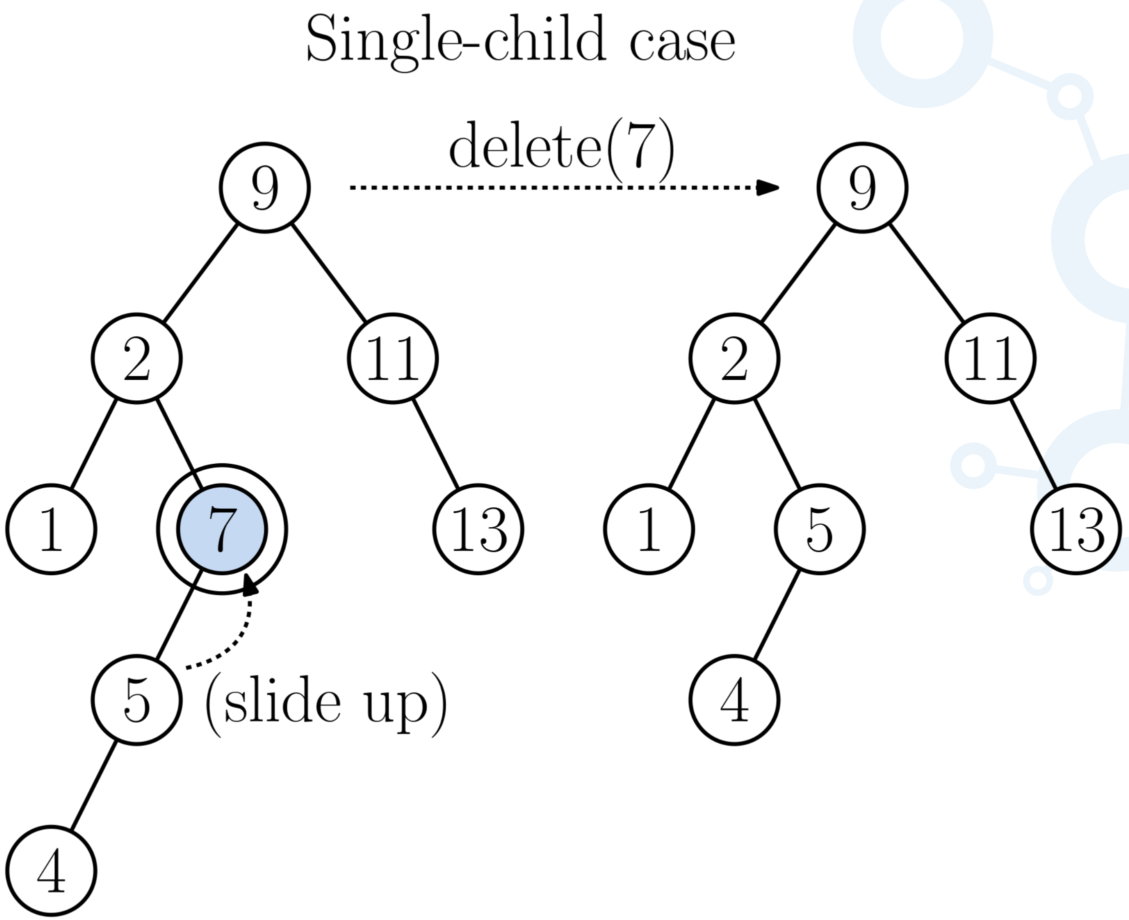
- Simply remove this node (and set parent's child link to null)



Binary Search Tree - Delete

Single-child case

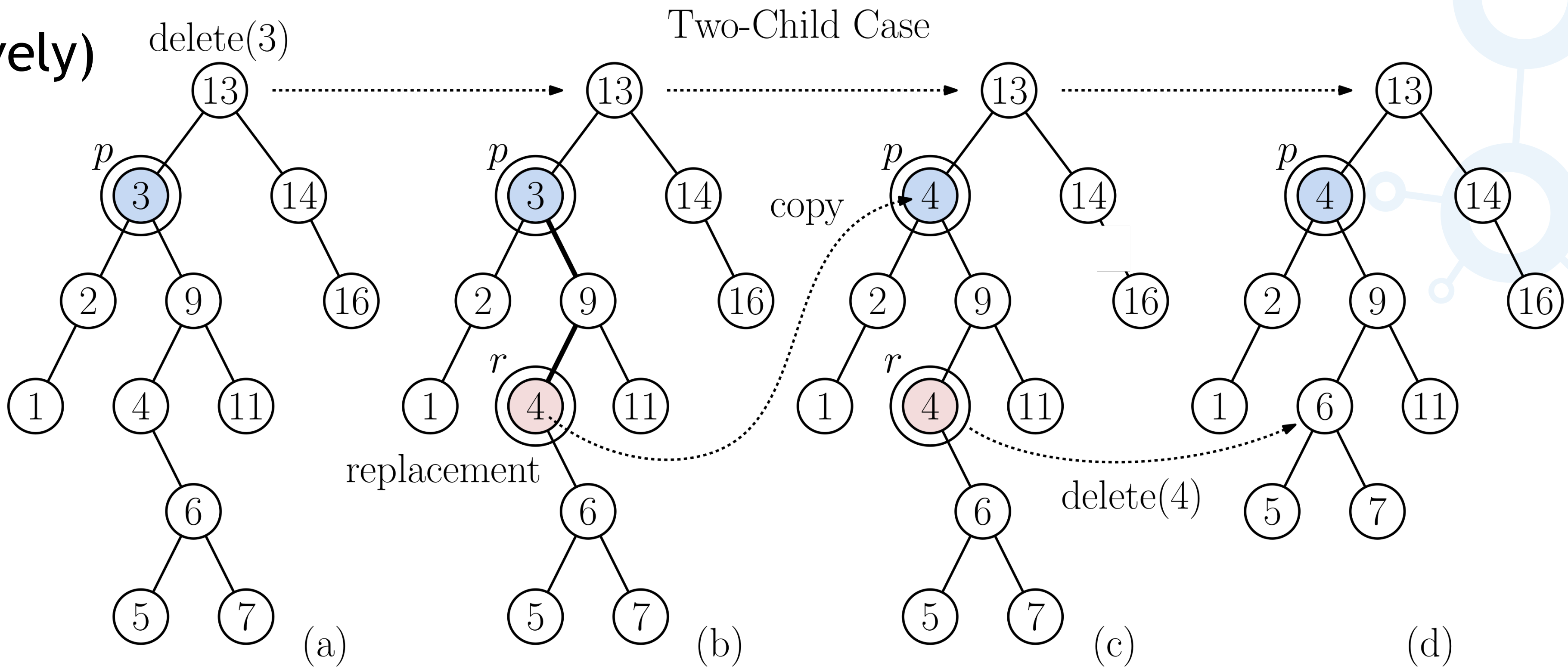
- Link the child in so that replaces the deleted node



Binary Search Tree - Delete

Two-Child Case

- Find a replacement node, r, our inorder successor
- Copy r's contents into the deleted node
- Delete r (recursively)



Binary Search Tree - Deletion

- First, define a helper procedure to find the replacement node
- This is the inorder successor, or equivalently, the leftmost node of the right subtree

```
BinaryNode findReplacement(BinaryNode p) {          // find p's replacement node
    BinaryNode r = p.right;                         // start in p's right subtree
    while (r.left != null) r = r.left;              // go to the leftmost node
    return r;
}
```

Binary Search Tree - Deletion

```
BinaryNode delete(Key x, BinaryNode p) {
       if (p == null)                                  // fell out of tree?
           throw KeyNotFoundException;                 // ...error - no such key
       else {
           if (x < p.data)                             // look in left subtree
               p.left = delete(x, p.left);
           else if (x > p.data)                        // look in right subtree
               p.right = delete(x, p.right);
                                                       // found it!
           else if (p.left == null || p.right == null) { // either child empty?
               if (p.left == null) return p.right;     // return replacement node
               else                return p.left;
           }
           else {                                      // both children present
               r = findReplacement(p);                 // find replacement node
               copy r's contents to p;                 // copy its contents to p
               p.right = delete(r.key, p.right);       // delete the replacement
           }
       }
       return p;
   }
```

Binary Search Tree - Analysis

- All operations take time $O(h)$, where h is the height of the tree
- But what is the height?
  - Worst case: $O(n)$
  - Best case: $O(\log n)$
  - Expected case? If keys are inserted in random order, then the expected depth of any node is $O(\log n)$
- The proof is rather messy (deriving a recurrence and solving it)
- We will show a weaker result, that the expected depth of the leftmost node is $O(\log n)$

Binary Search Tree - Analysis

Depth of the Leftmost Node

Theorem: Given a set of n keys $x_1 < x_2 < \ldots < x_n$, let $D(n)$ denote the expected depth of node x_1 after inserting all these keys in a binary search tree, under the assumption that all $n!$ insertion orders are equally likely. Then $D(n) \leq \ln n$, where $\ln$ denotes the natural logarithm.

Proof:

Overview:

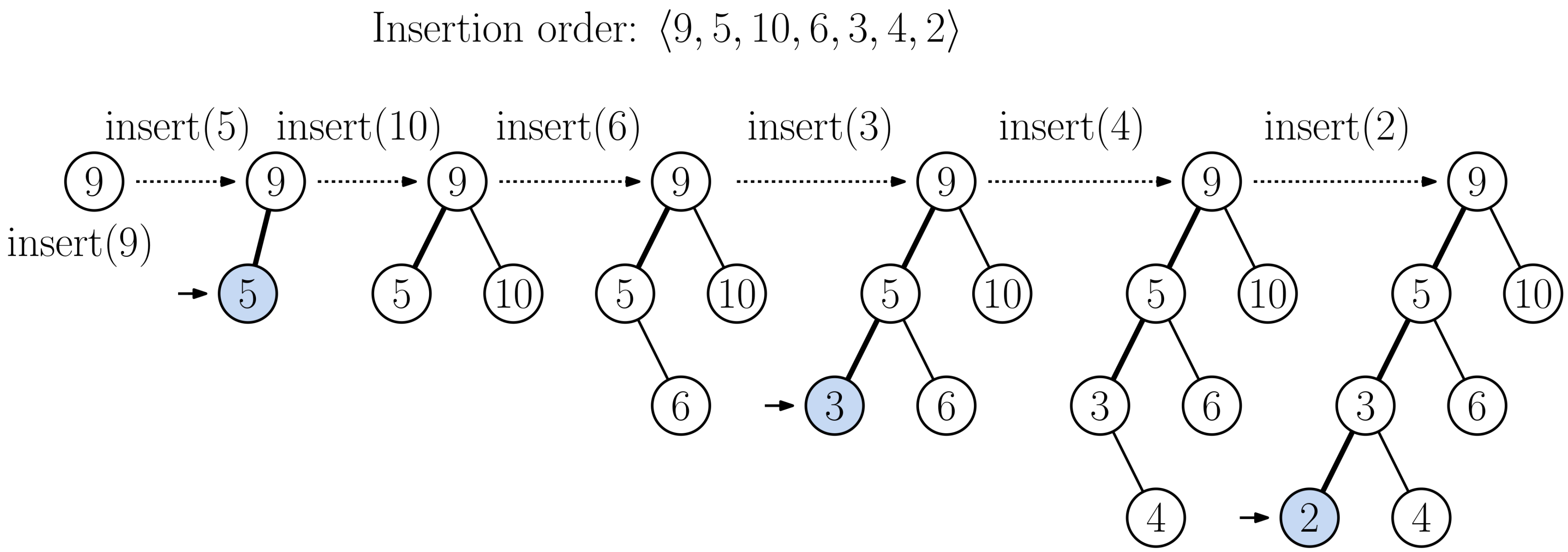
- We'll show that the depth of the leftmost node increases by one whenever the key being inserted is smaller than all the keys that preceded it
- We'll show that the probability of this occurring with the i-th insertion is $\frac{1}{i}$
- This implies that the expected height of the node is bounded by the Harmonic series

Binary Search Tree - Analysis

Depth of the Leftmost Node

Proof:

- Consider any i, $2 \leq i \leq n$. Observe that the depth of the leftmost node increases by one only when the i-th item to be inserted is the minimum among all the keys inserted so far



Binary Search Tree - Analysis

Depth of the Leftmost Node

Proof:

- Consider any i, $2 \leq i \leq n$. Observe that the depth of the leftmost node increases by one only when the i-th item to be inserted is the minimum among all the keys inserted so far
- Let X_i be a random variable that is 1 if the i-th item in the insertion sequence is the smallest so far and 0 otherwise
- Since the order of the first i items is random, $\Pr(X_i = 1) = \frac{1}{i}$ (anyone can be the min)
- Each time this event happens, the depth of the leftmost node increases by 1.
- Thus,

$$D(n) = \sum_{i=2}^{n} \Pr(X_i = 1) = \sum_{i=2}^{n} \frac{1}{\mathrm{i}} \leq \left(\sum_{i=1}^{n} \frac{1}{\mathrm{i}}\right) - 1 = \mathrm{H(n)} - 1,$$

- where $H(n)$ is the famous Harmonic Series. It is well known that $H(n) \leq (\ln n) + 1$.
- So $D(n) \leq \ln n$, as desired

Binary Search Tree - Analysis

Deletions behave differently

- Suppose you have a tree with roughly n nodes in the steady state, where nodes are inserted and deleted randomly
- You might think that the expected height would be $O(\log n)$, but it is not!
- Over time, the height converges to $O(\sqrt{n})$
- Why? Choosing the replacement node as the inorder successor, introduces a systematic bias into the tree's structure
- A more balanced approach would be to randomly switch between the inorder predecessor and inorder successor
- It is conjectured that with balanced deletion, the height of the tree is the same as in the insertion-only case [Culberson & Munro, 1990]

Summary

- Dictionary data structure
- Sequential allocation – Simple but slow
- Binary Search Trees
  - Definition
  - Finding a key
  - Inserting a key-value pair
  - Deleting a key
- Analysis
  - Expected case for insertion
  - The difficulty of deletions