Learning Goals

- Define a binary search tree
- Implement INSERTION and DELETION on a binary search tree
- Implement tree rotations
- Understand the meaning and consequence of balancedness of a BST

Tree: Definition

- A *tree* is a data structure that either is empty or consists a *root* node, with pointers linked to subtrees.

Tree: Definition

- A *tree* is a data structure that either is empty or consists a *root* node, with pointers linked to subtrees.
- If a subtree is not empty, its root is a *child* of the *parent*, the node pointing to it from the larger tree.

Tree: Definition

- A *tree* is a data structure that either is empty or consists a *root* node, with pointers linked to subtrees.
- If a subtree is not empty, its root is a *child* of the *parent*, the node pointing to it from the larger tree.
  - Very often each node in a tree maintains a pointer linked to its parent. For the root, this pointer points to NIL.

Tree: Definition

- A *tree* is a data structure that either is empty or consists a *root* node, with pointers linked to subtrees.
- If a subtree is not empty, its root is a *child* of the *parent*, the node pointing to it from the larger tree.
  - Very often each node in a tree maintains a pointer linked to its parent. For the root, this pointer points to NIL.
- A node whose children are all NIL is called a *leaf*.

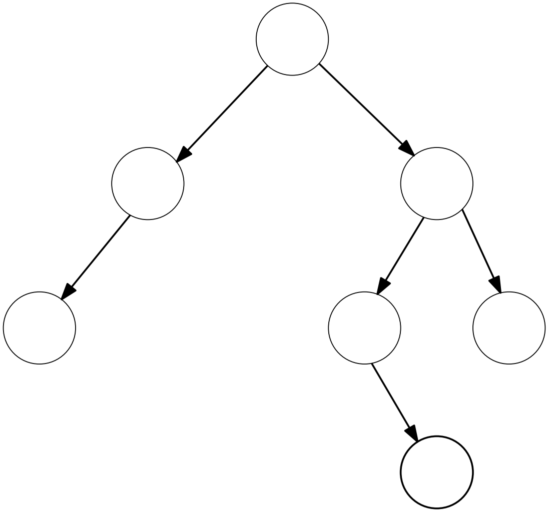
Tree: Definition

- A *tree* is a data structure that either is empty or consists a *root* node, with pointers linked to subtrees.
- If a subtree is not empty, its root is a *child* of the *parent*, the node pointing to it from the larger tree.
  - Very often each node in a tree maintains a pointer linked to its parent. For the root, this pointer points to NIL.
- A node whose children are all NIL is called a *leaf*.
- A tree is *binary* if every node of it has at most 2 children.

Tree: Definition

- A *tree* is a data structure that either is empty or consists a *root* node, with pointers linked to subtrees.
- If a subtree is not empty, its root is a *child* of the *parent*, the node pointing to it from the larger tree.
  - Very often each node in a tree maintains a pointer linked to its parent. For the root, this pointer points to NIL.
- A node whose children are all NIL is called a *leaf*.
- A tree is *binary* if every node of it has at most 2 children.
  - For a binary tree, a node's two children are referred to as its LEFT child and its RIGHT child.

Illustration: A Binary Tree



Binary Search Trees

- As a data structure, a tree stores data in its nodes, so a node has its key value and satellite content.

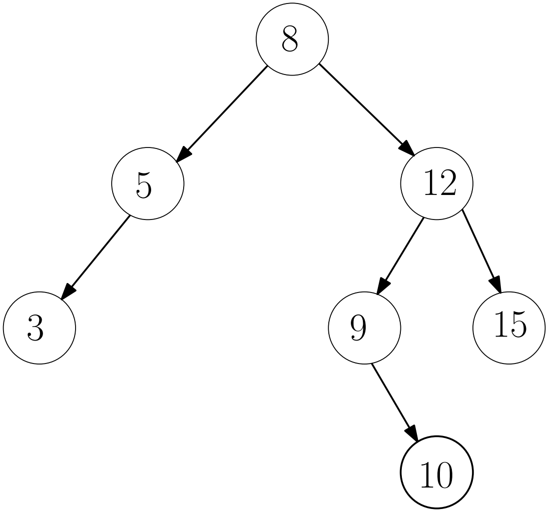
Binary Search Trees

- As a data structure, a tree stores data in its nodes, so a node has its key value and satellite content.
  - Some trees may not store satellite content in some of their nodes, e.g. a B+ tree. In this course we do not consider such trees.

Binary Search Trees

- As a data structure, a tree stores data in its nodes, so a node has its key value and satellite content.
  - Some trees may not store satellite content in some of their nodes, e.g. a B+ tree. In this course we do not consider such trees.
- A binary tree is a *binary search tree* if for every node in it, its key value is larger than (or equal to) all those in its left subtree, and smaller than (or equal to) all those in its right subtree.

A Binary Search Tree



FIND on a BST

- The *height* of a node is the number of nodes in the longest path from it to a leaf. The height of a tree is the height of its root.

FIND on a BST

- The *height* of a node is the number of nodes in the longest path from it to a leaf. The height of a tree is the height of its root.
  - The *depth* of a node is the number of nodes in the longest path from it to the root.

Find on a BST

- The *height* of a node is the number of nodes in the longest path from it to a leaf. The height of a tree is the height of its root.
  - The *depth* of a node is the number of nodes in the longest path from it to the root.
- Let r be the root of a tree and $h(r)$ its height.

FIND on a BST

- The *height* of a node is the number of nodes in the longest path from it to a leaf. The height of a tree is the height of its root.
  - The *depth* of a node is the number of nodes in the longest path from it to the root.
- Let r be the root of a tree and $h(r)$ its height.
- $\textsc{Find}(r, i)$ takes $O(h(r))$ time.

Find on a BST

- The *height* of a node is the number of nodes in the longest path from it to a leaf. The height of a tree is the height of its root.
  - The *depth* of a node is the number of nodes in the longest path from it to the root.
- Let r be the root of a tree and $h(r)$ its height.
- $\textsc{Find}(r, i)$ takes $O(h(r))$ time.
- How good is this?

Find on a BST

- The *height* of a node is the number of nodes in the longest path from it to a leaf. The height of a tree is the height of its root.
  - The *depth* of a node is the number of nodes in the longest path from it to the root.
- Let r be the root of a tree and $h(r)$ its height.
- Find(r, i) takes $O(h(r))$ time.
- How good is this?
  - In the worst case, the tree can be a linked list, so this is linear time.

FIND on a BST

- The *height* of a node is the number of nodes in the longest path from it to a leaf. The height of a tree is the height of its root.
    - The *depth* of a node is the number of nodes in the longest path from it to the root.
- Let r be the root of a tree and $h(r)$ its height.
- FIND(r, i) takes $O(h(r))$ time.
- How good is this?
    - In the worst case, the tree can be a linked list, so this is linear time.
    - In the best case, a tree with n nodes has height $O(\log n)$.

FIND on a BST

- The *height* of a node is the number of nodes in the longest path from it to a leaf. The height of a tree is the height of its root.
  - The *depth* of a node is the number of nodes in the longest path from it to the root.
- Let r be the root of a tree and $h(r)$ its height.
- $\textsc{Find}(r, i)$ takes $O(h(r))$ time.
- How good is this?
  - In the worst case, the tree can be a linked list, so this is linear time.
  - In the best case, a tree with n nodes has height $O(\log n)$.
    - A tree with height h can have $2^h - 1$ nodes.
  - The next few chapters study algorithms that maintain search trees to be closer to the best case and farther from the worst case.

Traversal of a BST

- Given a BST with n nodes, we can output in $O(n)$ time its entries in increasing/decreasing key values.

Traversal of a BST

- Given a BST with n nodes, we can output in $O(n)$ time its entries in increasing/decreasing key values.
- The recursive algorithm is called an *inorder tree walk*: it outputs the root in between the left and the right subtrees.

Traversal of a BST

- Given a BST with n nodes, we can output in $O(n)$ time its entries in increasing/decreasing key values.
- The recursive algorithm is called an *inorder tree walk*: it outputs the root in between the left and the right subtrees.
- Pseudocode:

```
INORDER-TREE-WALK(r);
if x ≠ NIL then
    INORDER-TREE-WALK(left(r));
    print key(x);
    INORDER-TREE-WALK(right(r))
end
```

Min and Max

- Min(r): Given the root r of a BST, return the node with the smallest key value in the tree

Min and Max

- Min(r): Given the root r of a BST, return the node with the smallest key value in the tree
  - Return the leftmost node in the tree

Min and Max

- MIN(r): Given the root r of a BST, return the node with the smallest key value in the tree
  - Return the leftmost node in the tree

```
MIN(r);
if left(r) = NIL then
    return r
else
    return MIN(left(r))
end
```

- MAX(r): Given the root r of a BST, return the node with the largest key value in the tree

Min and Max

- Min(r): Given the root r of a BST, return the node with the smallest key value in the tree
  - Return the leftmost node in the tree

```
Min(r);
if left(r) = Nil then
  | return r
else
  | return Min(left(r))
end
```

- Max(r): Given the root r of a BST, return the node with the largest key value in the tree
- Both operations take $O(h(r))$ time.

Successor and Predecessor

- Successor(x): Given a node x in a BST, we would like to return the node containing the next larger key value

SUCCESSOR and PREDECESSOR

- SUCCESSOR(x): Given a node x in a BST, we would like to return the node containing the next larger key value
  - If x has a right child, the successor should be the minimum value in the right subtree.

Successor and Predecessor

- Successor(x): Given a node x in a BST, we would like to return the node containing the next larger key value
  - If x has a right child, the successor should be the minimum value in the right subtree.
  - Otherwise, we should go up the tree until we find a node that is a left child, and we return the parent.

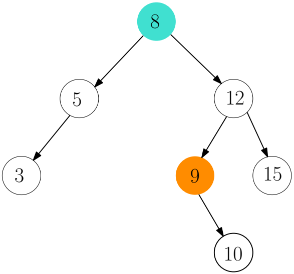
SUCCESSOR and PREDECESSOR

- SUCCESSOR(x): Given a node x in a BST, we would like to return the node containing the next larger key value
  - If x has a right child, the successor should be the minimum value in the right subtree.
  - Otherwise, we should go up the tree until we find a node that is a left child, and we return the parent.

```
SUCCESSOR(x);
if right(x) ≠ NIL then  return MIN(right(x)) ;
while parent(x) ≠ NIL do
    y ← parent(x);
    if x = left(y) then return y;
    x ← y;
end
return NIL
```

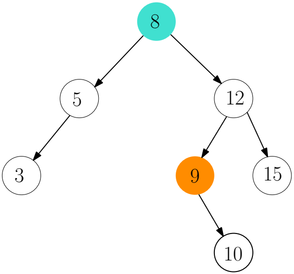
- PREDESSOR(x): Symmetric to SUCCESSOR.

Illustration: Successor

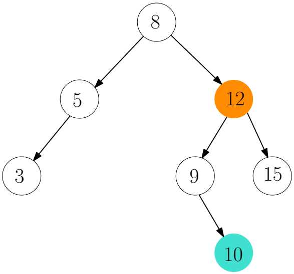


Case 1.

Illustration: Successor



Case 1.



Case 2.

Insertion

- INSERTION(x, r): Given a new node x, insert it into a BST rooted at r.

Insertion

- INSERTION(x, r): Given a new node x, insert it into a BST rooted at r.
- If $key(x) < key(r)$, then if $left(r) \neq$ NIL, call INSERT$(x, left(r))$, else insert x as r's left child.

Insertion

- INSERTION(x, r): Given a new node x, insert it into a BST rooted at r.
- If $key(x) < key(r)$, then if $left(r) \neq$ NIL, call INSERT$(x, left(r))$, else insert x as r's left child.
- Otherwise, $key(x) \geq key(r)$, if $right(r) \neq$ NIL, call INSERT$(x, right(r))$, else insert x as r's right child.

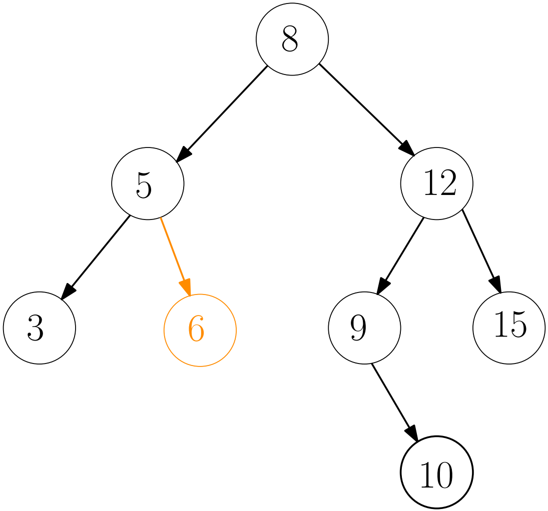
Insertion

- INSERTION(x, r): Given a new node x, insert it into a BST rooted at r.
- If $key(x) < key(r)$, then if $left(r) \neq$ NIL, call INSERT$(x, left(r))$, else insert x as r's left child.
- Otherwise, $key(x) \geq key(r)$, if $right(r) \neq$ NIL, call INSERT$(x, right(r))$, else insert x as r's right child.
- When coding, don't forget to update the pointers and check ill cases.

Insertion

- INSERTION(x, r): Given a new node x, insert it into a BST rooted at r.
- If $key(x) < key(r)$, then if $left(r) \neq$ NIL, call INSERT$(x, left(r))$, else insert x as r's left child.
- Otherwise, $key(x) \geq key(r)$, if $right(r) \neq$ NIL, call INSERT$(x, right(r))$, else insert x as r's right child.
- When coding, don't forget to update the pointers and check ill cases.
- Note that every newly inserted node becomes a leaf.

Illustration: Insertion



Deletion

- DELETE(x): Given a node x in a BST, delete it, while maintaining the tree as a BST

Deletion

- DELETE(x): Given a node x in a BST, delete it, while maintaining the tree as a BST
- If x is a leaf, simply delete it and update the parent's pointer.

Deletion

- DELETE(x): Given a node x in a BST, delete it, while maintaining the tree as a BST
- If x is a leaf, simply delete it and update the parent's pointer.
- If x has only one child, bypass it by updating the parent's and that child's pointer.

Deletion

- Delete(x): Given a node x in a BST, delete it, while maintaining the tree as a BST
- If x is a leaf, simply delete it and update the parent's pointer.
- If x has only one child, bypass it by updating the parent's and that child's pointer.
- What if x has two children?

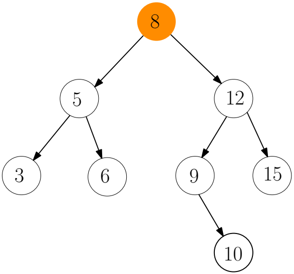
Deletion

- DELETE(x): Given a node x in a BST, delete it, while maintaining the tree as a BST
- If x is a leaf, simply delete it and update the parent's pointer.
- If x has only one child, bypass it by updating the parent's and that child's pointer.
- What if x has two children?
  - Note that now the successor of x must be the minimum valued node y in its right subtree, and y has no left child;

Deletion

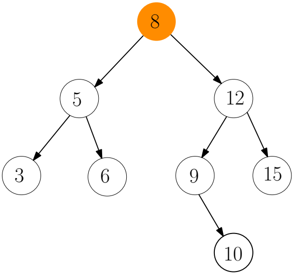
- DELETE(x): Given a node x in a BST, delete it, while maintaining the tree as a BST
- If x is a leaf, simply delete it and update the parent's pointer.
- If x has only one child, bypass it by updating the parent's and that child's pointer.
- What if x has two children?
  - Note that now the successor of x must be the minimum valued node y in its right subtree, and y has no left child;
  - We can replace x by y, and delete y.

Illustration: Deletion

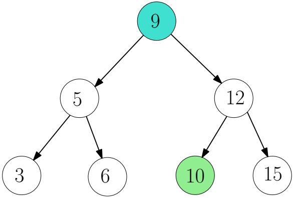


Case 1.

Illustration: Deletion



Case 1.



Case 2.

Tree Rotation

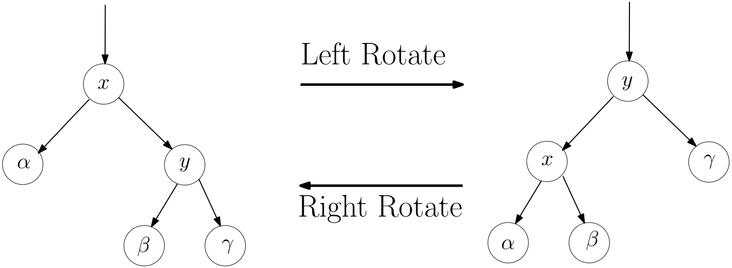
- As we have seen, the same set of key values can be stored in different BSTs, resulting in very different performance.

Tree Rotation

- As we have seen, the same set of key values can be stored in different BSTs, resulting in very different performance.
- An important operation that locally adjusts the structure of a BST while maintaining its search property is *rotation*.

Tree Rotation

- As we have seen, the same set of key values can be stored in different BSTs, resulting in very different performance.
- An important operation that locally adjusts the structure of a BST while maintaining its search property is *rotation*.



BST vs. Hashing

An ideal BST takes time $O(\log n)$ for most basic operations. An ideal Hash table takes only $O(1)$ time. So why BST?

BST vs. Hashing

An ideal BST takes time $O(\log n)$ for most basic operations. An ideal Hash table takes only $O(1)$ time. So why BST?

- BST is more space efficient.
    - A hash table typically has a fraction of unused space.

BST vs. Hashing

An ideal BST takes time $O(\log n)$ for most basic operations. An ideal Hash table takes only $O(1)$ time. So why BST?

- BST is more space efficient.
  - A hash table typically has a fraction of unused space.
- BST is more "persistent".
  - As the data set grows, we may need to resize a hash table. We will see in the next lecture that on average this may not be super expensive, but this may make the performance less smooth, or "persistent".

BST vs. Hashing

An ideal BST takes time $O(\log n)$ for most basic operations. An ideal Hash table takes only $O(1)$ time. So why BST?

- BST is more space efficient.
  - A hash table typically has a fraction of unused space.
- BST is more "persistent".
  - As the data set grows, we may need to resize a hash table. We will see in the next lecture that on average this may not be super expensive, but this may make the performance less smooth, or "persistent".
- BST is more secure against malicious attacks.
  - If an adversary figures out the hash function we use, they can generate data that causes many collisions and slow down the performance.

BST vs. Hashing

An ideal BST takes time $O(\log n)$ for most basic operations. An ideal Hash table takes only $O(1)$ time. So why BST?

- BST is more space efficient.
    - A hash table typically has a fraction of unused space.
- BST is more "persistent".
    - As the data set grows, we may need to resize a hash table. We will see in the next lecture that on average this may not be super expensive, but this may make the performance less smooth, or "persistent".
- BST is more secure against malicious attacks.
    - If an adversary figures out the hash function we use, they can generate data that causes many collisions and slow down the performance.
- BST supports range search and traversal.

BST vs. Hashing

An ideal BST takes time $O(\log n)$ for most basic operations. An ideal Hash table takes only $O(1)$ time. So why BST?

- BST is more space efficient.
  - A hash table typically has a fraction of unused space.
- BST is more "persistent".
  - As the data set grows, we may need to resize a hash table. We will see in the next lecture that on average this may not be super expensive, but this may make the performance less smooth, or "persistent".
- BST is more secure against malicious attacks.
  - If an adversary figures out the hash function we use, they can generate data that causes many collisions and slow down the performance.
- BST supports range search and traversal.
  - Range search example: return all entries whose key values are between 5000 and 10000. BST handles this easily; hashing can be awkward.

BST vs. Hashing

An ideal BST takes time $O(\log n)$ for most basic operations. An ideal Hash table takes only $O(1)$ time. So why BST?

- BST is more space efficient.
    - A hash table typically has a fraction of unused space.
- BST is more "persistent".
    - As the data set grows, we may need to resize a hash table. We will see in the next lecture that on average this may not be super expensive, but this may make the performance less smooth, or "persistent".
- BST is more secure against malicious attacks.
    - If an adversary figures out the hash function we use, they can generate data that causes many collisions and slow down the performance.
- BST supports range search and traversal.
    - Range search example: return all entries whose key values are between 5000 and 10000. BST handles this easily; hashing can be awkward.
    - Traversal example: return all entries in increasing key values. Very natural with BST, but inefficient with hashing.