Binary Search Trees

CMPT 145

Objectives

1. Describe what a binary search tree is, in terms of its basic organizational principle.
2. Describe basic operations of binary search tree, and demonstrate the behaviour in example binary search trees.
3. Explain how tree balance affects the efficiency of various binary search tree operations.

Searching for a data value in a tree

```
def member(tnode, val):
    if tnode == None:
        return False
    elif tnode.data == val:
        return True
    else:
        return member(tnode.left, val) \
            or member(tnode.right, val)
```

Quick Sort

```
def qs(alist):
    if len(alist) == 0:
        return []
    else:
        pivot = alist[0]
        smaller = [x for x in alist if x < pivot]
        equal = [x for x in alist if x == pivot]
        greater = [x for x in alist if x > pivot]
        return qs(smaller) + equal + qs(greater)
```

What is the time complexity of Quick Sort?

Data collections, and search

- The Python operator `in` invokes a method that scans the given sequence, searching for the given item.
- If the list has n elements, `x in alist` is $O(n)$.
- Binary search is $O(\log n)$, provided the list is sorted.
- Is it a good idea to sort a list before searching, so that we can use binary search?
- What if we have a data collection where data is being inserted and deleted frequently?

Binary Search Trees: Motivation

- A Binary Search Tree is a special kind of binary tree.
- It will allow us to keep data organized.
- Fast searches!
- No need to sort the data before searching!

Binary Search Tree Property

If a binary tree satisfies the following property, it is called a Binary Search Tree.

- Let v be any node in the tree.
- We assume no data value appears more than once.

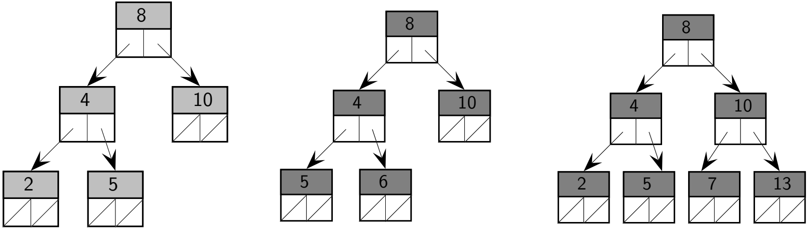
Binary Search Tree Property

1. All the data stored in the left subtree of v is smaller than the data at v.
2. All the data stored in the right subtree of v is larger than the data at v.

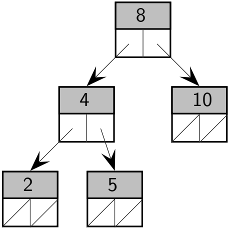
Note: smaller and larger could mean before and after according to a given rule, e.g. alphabetical order.

Binary Search Trees

Are the following trees binary search trees?

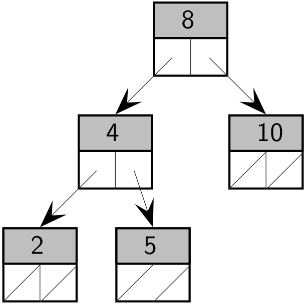


Binary Search Tree traversals



- Pre-order?
- Post-order?
- In-order?

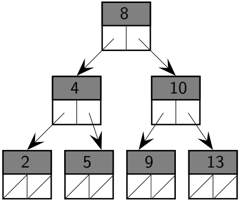
Exercises



Given a tree with the BST property, write a function that:

1. Finds the smallest value
2. Finds the largest value

Searching in a Binary Search Tree



Given a Binary Search Tree, describe how you would find the data value 9.

Algorithm: Member

Given a tree with the BST property and a data value:

- If the tree is empty, return False
- If the root stores the same data value, return True
- If the data value is smaller than the root, look left recursively
- If the data value is larger than the root, look right recursively

Implementation: member

```python
def member_prim(tnode, value):
    '''
    Check if value is stored in the binary search tree.
    Preconditions:
        :param abst: a binary search tree
        :param value: a value
    Postconditions:
        none
    :return: True if value is in the tree
    '''
    if tnode == None:
        return False
    else:
        cval = tnode.data
        if cval == value:
            return True
        elif value < cval:
            return member_prim(tnode.left, value)
        else:
            return member_prim(tnode.right, value)
```

Time complexity: Member

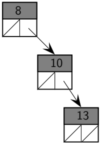
- The recursive case only follows one branch.
- Recursion stops at an empty node, or if the value is found.
- Best-case: if the value is found near the top of the whole tree
- Worst cases:

    1. If the value is not there at all.
    2. If the value is at a leaf node.

- The number of recursive calls depends on the height of the tree.

Time complexity

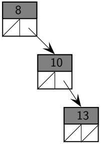
- This height depends on the “shape” of the tree.



- The tree above is known as a *degenerate* tree because each node has zero or one children.
- The height of this tree is the same as the number of nodes.

Time complexity

- The height depends on the “shape” of the tree.

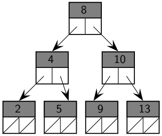


Time complexity - degenerate trees

The worst case time complexity of `member_prim` on *degenerate* search trees is $O(n)$.

Time complexity

- This height depends on the "shape" of the tree.



- This is a *complete* binary tree.
- Given a complete binary tree with n nodes and height h,

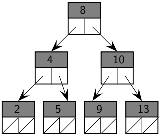
$$n = 2^h - 1$$

- Solving for h:

$$h = \log(n + 1)$$

Time complexity

- This height depends on the “shape” of the tree.



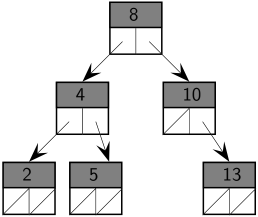
Time complexity - complete binary search trees.

The worst case time complexity of `member_prim` on *complete* binary search trees is $O(\log n)$.

Time complexity: perspective

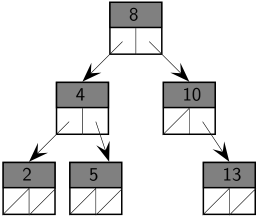
- $O(\log n)$ is hugely better than linear search in a list.
- $O(\log n)$ is the same as binary search on a sorted list.
- If we can find a way to add to and remove values from a tree that's faster than sorting, we win!
  - Sorting is $O(n \log n)$ in general.
  - Sorting + Binary search is $O(n \log n) + O(\log n)$, that is: $O(n \log n)$

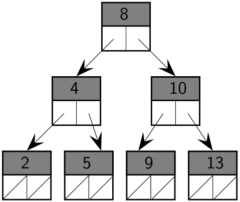
Adding to a tree with the BST property



- Let's say we would like to insert 9 into this tree.
- We have to put this data in the place we would find it, if it were in the tree already!
- As the left child of node 10.
- This is the only place it could go in this tree.

Adding to a tree with the BST property





- Let's say we would like to insert 9 into this tree.
- We have to put this data in the place we would find it, if it were in the tree already!
- As the left child of node 10.
- This is the only place it could go in this tree.

Algorithm: insert

Given a tree with the BST property and a data value:

- If the tree is empty, return a new tree with the value stored
- If the root stores the same data value, return the root
- If the data value is smaller than the root, update the left subtree, recursively
- If the data value is larger than the root, update the right subtree recursively

Implementation: insert

```
def insert_prim(tnode,value):
    '''
    Insert a new value into the binary tree.
    Preconditions:
        :param tnode: a binary search tree
        :param value: a value
    Postconditions:
        If the value is not already in the tree,
        it is added to the tree
    :return: False if the value was already there
    '''
    # next slide
```

Implementation: insert

```
def insert_prim(tnode,value):
    if tnode == None:
        return True, TreeNode(value)
    else:
        cval = tnode.data
        if cval == value:
            return False, tnode
        elif value < cval:
            fl, sub = insert_prim(tnode.left, value)
            if fl:
                tnode.left = sub
            return fl, tnode
        else:
            fl, sub = insert_prim(tnode.right, value)
            if fl:
                tnode.right = sub
            return fl, tnode
```

Time complexity: Insert

- Basically the same algorithm as `member`
- The time complexity analysis is the same.

Time complexity - complete binary search trees.

The worst case time complexity of `insert_prim` on *complete* binary search trees is $O(\log n)$.

Time complexity - degenerate trees

The worst case time complexity of `insert_prim` on *degenerate* search trees is $O(n)$.

Time complexity: perspective

- $O(\log n)$ is hugely better adding a node in a sorted list.
- As long as the tree stays roughly balanced!

Deleting

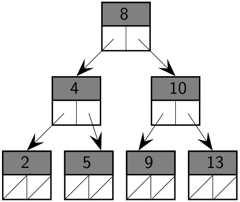
- We want to be able to delete *a single node* of the tree.
- This can be especially tricky if it has children.
- We must also be careful that the resulting tree is a binary search tree.

Deleting

When we delete a node, there are four cases which could occur:

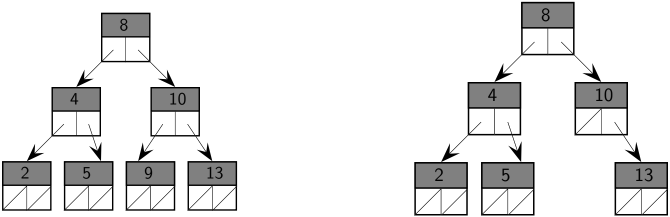
1. The node to be deleted has no children.
2. The node to be deleted has only a right subtree.
3. The node to be deleted has only a left subtree.
4. The node to be deleted has two subtrees.

Case 1: the node to be deleted has no children



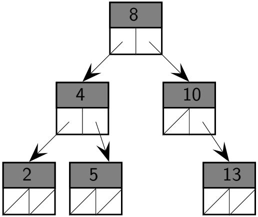
To delete a leaf, e.g., node 9, we can just remove the leaf, and the resulting tree will remain a binary search tree.

Case 1: the node to be deleted has no children



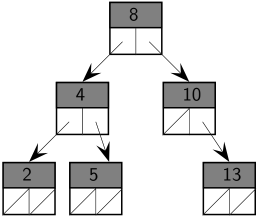
To delete a leaf, e.g., node 9, we can just remove the leaf, and the resulting tree will remain a binary search tree.

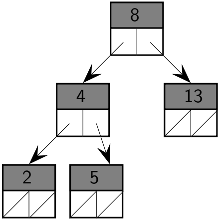
Case 2: the node to be deleted has only a right child



- To delete the node with one child, e.g., node 10, we can connect the entire subtree at node 13.
- The resulting tree will remain a binary search tree.
- Case 3, where the node to be deleted has only a left child is similar.

Case 2: the node to be deleted has only a right child



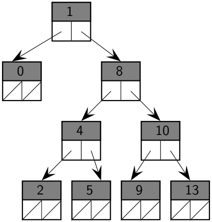


- To delete the node with one child, e.g., node 10, we can connect the entire subtree at node 13.
- The resulting tree will remain a binary search tree.
- Case 3, where the node to be deleted has only a left child is similar.

Case 3: the node to be deleted has only a left child

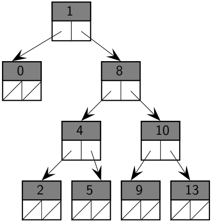
This is similar to Case 2!

Case 4: the node to be deleted has a left and right child



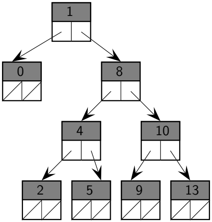
- Deleting the node containing 8 leaves us with 2 subtrees to reconnect.
- There are a bunch of ways to do this.

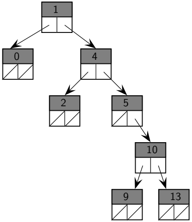
Case 4: the node to be deleted has a left and right child



- We need to maintain the BST property!
- We could connect either subtree to node 8's parent.
- E.g., we could connect node 1 to node 4.
- We need to reconnect the other subtree somewhere.

Case 4: the node to be deleted has a left and right child





- We need to maintain the BST property!
- Everything in node 10's subtree is bigger than anything in node 4's subtree.
- Connect 10 all the way to the right starting from node 4.

Algorithm: delete

Given a tree with the BST property and a data value:

- If the tree is empty, return False
- If the root stores the same data value, reconnect the root's children (see Algorithm reconnect)
- If the data value is smaller than the root, update the left subtree, recursively
- If the data value is larger than the root, update the right subtree recursively

Algorithm: reconnect

Given a tree with the BST property, whose root is to be deleted:

- If the root has no children, return None (the empty tree)
- If the root has a left child, but no right child, return the left child.
- If the root has a right child, but no left child, return the right child.
- If the root has 2 children:
  - Step down the left child, always going right, to find left's largest node.
  - Attach the root's right child as the right subtree of left's largest node
  - Return left.

Time complexity: Delete

- Basically the same algorithm as `member`
- The time complexity analysis is the same.
- The worst case time complexity of `delete` on *complete* binary search trees is $O(\log n)$.
- The worst case time complexity of `delete` on *degenerate* binary search trees is $O(n)$.

Time complexity: reconnect

- Best case: no more than 1 child: $O(1)$
- Worst case: 2 children.
- Number of steps depends on the shape of the left subtree
  - The worst case time complexity of `reconnect` on *complete* binary search trees is $O(\log n)$.
  - The worst case time complexity of `reconnect` on *degenerate* binary search trees is $O(n)$.

Time complexity: perspective

- Sorted list:
  - `member()`: $O(n)$
  - `insert()`: $O(n)$
  - `delete()`: $O(n)$
- Balanced binary search tree:
  - `member()`: $O(\log n)$
  - `insert()`: $O(\log n)$
  - `delete()`: $O(\log n)$
  - As long as the tree stays roughly balanced!
- $O(\log n)$ is hugely better than $O(n)$.

Keeping trees balanced

- None of our algorithms try to keep the tree balanced
- One can easily order insertions and deletions so that the tree stays degenerate.
- On average, binary search trees stay more or less balanced
- There are more complicated algorithms that rearrange the tree at every insertion or deletion to guarantee balance
- This is a second year topic!