

Cosc 201  
Algorithms and Data Structures  
Lecture 11 (30/3/2026)  
Operations on binary search trees

Brendan McCane  
[brendan.mccane@otago.ac.nz](mailto:brendan.mccane@otago.ac.nz)  
And  
Michael Albert



## Binary search tree definition

A binary search tree (BST) is a collection of nodes with one distinguished node called the root

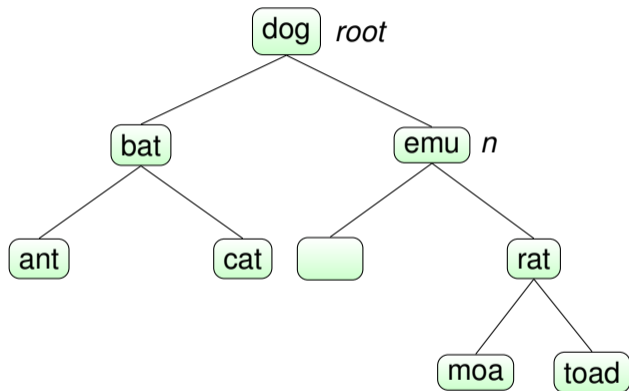
- ▶ A node contains data called a *key* which comes from some ordered type (e.g., `String`).
- ▶ Each node can have at most two children - there are two fixed slots called *left child* and *right child*.
- ▶ The left child of a node and all their descendants are called the left subtree at the node. The right subtree is defined similarly.
- ▶ The key values at all nodes in the left subtree of a node must be less than the key of the node.
- ▶ The key value at all nodes in the right subtree of a node must be greater than the key of the node.
- ▶ Duplicate keys are not permitted.

## Notation for BST pseudo-code

If  $n$  is a node:

- ▶  $n.key$  for its key,
- ▶  $n.l$  and  $n.r$  for its left and right children
- ▶  $n.p$  for its parent
- ▶ **nil** for a “not here” marker, e.g.,  $n.r = \mathbf{nil}$  means “ $n$  has no right child”.

## Example BST



- ▶  $root.p = \mathbf{nil}$
- ▶  $root.r = n$
- ▶  $n.p = root$
- ▶  $n.key = emu$
- ▶  $n.l = \mathbf{nil}$
- ▶  $n.r.key = rat$

## Searching in a BST

We want to search in a BST for a key  $k$  returning **true** if it is found and **false** if it is not.

```
1:  $n \leftarrow \text{root}$ 
2: while  $n \neq \text{nil}$  do
3:   if  $n.\text{key} = k$  then
4:     return true
5:   else if  $n.\text{key} < k$  then
6:      $n \leftarrow n.r$ 
7:   else
8:      $n \leftarrow n.l$ 
9:   end if
10: end while
11: return false
```

## Searching in a BST, recursively

We want to search in a BST for a key  $k$  returning **true** if it is found and **false** if it is not.

The method  $search(k)$  simply calls  $search(k, root)$  where the two-parameter version,  $search(k, n)$  is defined by:

```
1: if  $n = \text{nil}$  then  
2:   return false  
3: else if  $n.key = k$  then  
4:   return true  
5: else if  $n.key < k$  then  
6:   return  $search(k, n.right)$   
7: else  
8:   return  $search(k, n.left)$   
9: end if
```

## Adding to a BST

We want to add a key  $k$  to a BST returning **true** if it is added and **false** if it was already present.

```
1:  $p \leftarrow \text{nil}, c \leftarrow \text{root}$ 
2: while  $c \neq \text{nil}$  do
3:    $p \leftarrow c$ 
4:   if  $p.\text{key} = k$  then
5:     return false
6:   else if  $p.\text{key} < k$  then
7:      $c \leftarrow p.r$ 
8:   else
9:      $c \leftarrow p.l$ 
10:  end if
11: end while
12: make a child node of  $p$  with value  $k$ 
13: return true
```

## Making a child

We want to add a child node with key  $k$  to a parent node  $p$ .

**Assumption** This is **really** what we want to do, i.e.,  $k$  is not the key of  $p$  and the position where this node will be added is currently **nil**.

```
1:  $c \leftarrow \text{new}(k), c.p \leftarrow p$   
2: if  $p.\text{key} < k$  then  
3:    $p.r \leftarrow c$   
4: else  
5:    $p.l \leftarrow c$   
6: end if
```

## Removing from a BST

This one is a little complicated! There are some cases to consider.

- ▶ First find the node,  $c$ , containing the key to be removed and its parent node  $p$ .
- ▶ If  $c$  has no children then just replace it by **nil** as the (appropriate) child of its parent.
- ▶ If  $c$  has only one child,  $d$ , replace it by  $d$  as the (appropriate) child of its parent.
- ▶ But, if  $c$  has two children . . .
- ▶ A new idea – find the *successor* of  $c$ . That's a node with at most one child. Delete it (previous case) and replace  $c$ 's key by the successor's key.

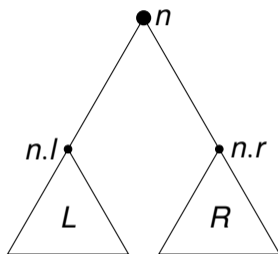
## What's the successor?

The *successor* of a node  $n$  in a BST is the leftmost descendant of its right child.

**Assumption** The node  $n$  has a right child which is not **nil**

```
1:  $c \leftarrow n.r$   
2: while  $c.l \neq \text{nil}$  do  
3:    $c \leftarrow c.l$   
4: end while  
5: return  $c$ .
```

## Traversing a BST



There are three different traversals we commonly consider in a BST. They are all based on the picture to the left.

We need to “visit”  $n$ , visit all the nodes in  $L$  and visit all the nodes in  $R$ . The three different traversals represent three different choices of what order to make those visits in.

### **Preorder**

---

Visit  $n$   
Traverse  $L$   
Traverse  $R$

### **Inorder**

---

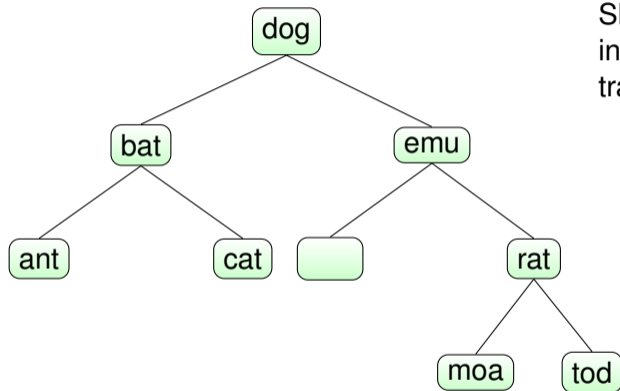
Traverse  $L$   
Visit  $n$   
Traverse  $R$

### **Postorder**

---

Traverse  $L$   
Traverse  $R$   
Visit  $n$

## Traversal example



Showing the keys of the nodes in the order visited by the three traversals:

**Preorder** dog, bat, ant, cat, emu, rat, moa, tod.

**Inorder** ant, bat, cat, dog, emu, moa, rat, tod.

**Postorder** ant, cat, bat, moa, tod, rat, emu, dog.

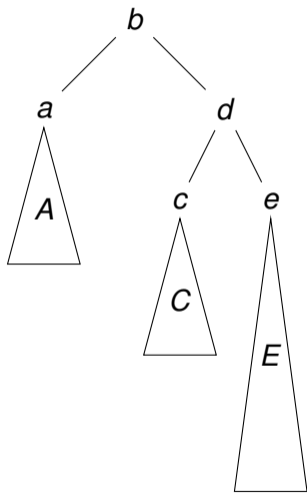
## Long branches are a problem

- ▶ The performance bounds for all of the BST operations are linear in the length of the longest branch.
- ▶ Ideally, we'd like our BST's to look more like heaps (wide and shallow) than long spindly branches.
- ▶ Unlike heaps though, the structure of a BST is out of our control – it depends on the order in which elements are added.
- ▶ In particular if they are added in their natural order we'll get one great big long branch and the BST performance will degrade to that of a linked list.
- ▶ A BST in which there are no “long” branches is called *balanced*.
- ▶ So, are there mechanisms by which we can ensure balance? Can they be implemented with limited additional overhead?

## Is balancing even possible?

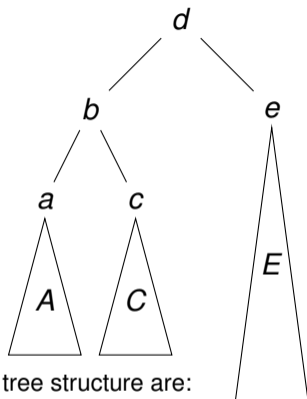
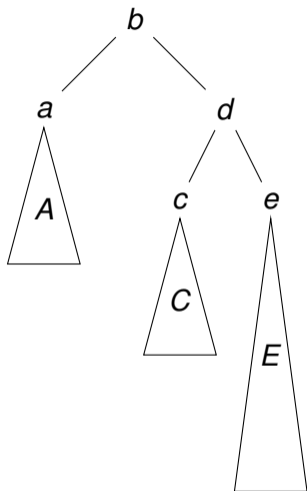
- ▶ In a global way, yes.
  - ▶ Perform an in-order traversal of the whole tree and save the results to a (sorted) array.
  - ▶ Now repeatedly bisect the array - use the middle element as the root
  - ▶ Recursively build the left and right subtrees from the part before the middle and the part after.
  - ▶ But, this hardly meets the “limited additional overhead” criterion!
- ▶ We need some local operation that helps to restore balance.
- ▶ The name of this operation is **rotation**.
- ▶ So, what's that?

## The bad situation (and its fix)



- ▶ Suppose that in this BST, there is a longer chain in  $E$  than elsewhere.
- ▶ The idea is to rotate  $d$  upwards (to become in this view at least the root).
- ▶ Which gives ...
- ▶ And that's an improvement.

## Side by side



The internal changes to the tree structure are:

- ▶ Change *b*'s right child to point to *c*.
- ▶ Change *d*'s left child to point to *b*.
- ▶ Change the link on *b*'s parent (not shown) that pointed to *b* to point to *d* instead.