Search Algorithms II

15-110 – Monday 10/21

Announcements

- Welcome back from fall break!
- Check3/Hw3 Revision deadline: **tomorrow at noon**
- Final exam scheduled: **Monday, Dec 9, 2024, 01:00pm-04:00pm**
  - Do not schedule travel before this time!
- Last day for voter registration in PA!

Learning Objectives

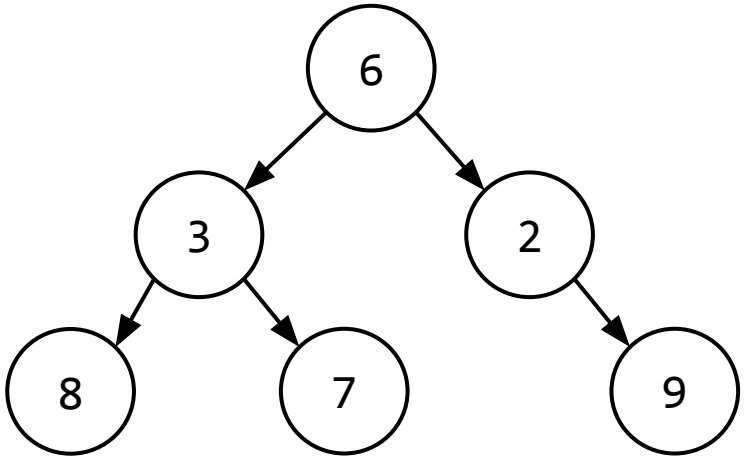
- Identify whether a tree is a **tree**, a **binary tree**, or a **binary search tree (BST)**
- Search for values in trees using **linear search** and in BSTs using **binary search**
- Analyze the **efficiency** of binary search on a **balanced** vs. **unbalanced** BSTs
- Recognize the requirements for building a good **hash function** and a good **hash table** that lead to **constant-time search**

Binary Search Trees

Revisiting search: Like we searched lists **we can also search other data structures**.

Recall we implemented binary and linear search on lists. **We can also do search on trees.**

Is 2 in the tree?



To do **linear search on a tree**, visit every node using recursion until we find the value.

Recursive case: Check if root node is the value we are looking for. Search the left subtree, search the right subtree.

Base case: If the tree is empty, return False.

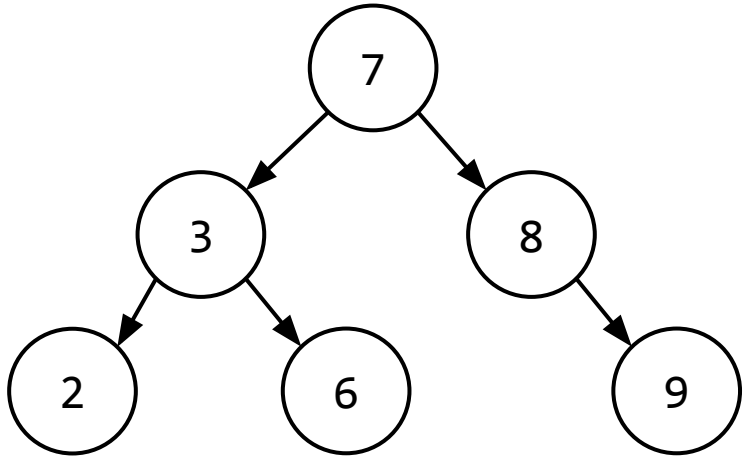
```
def search(t, target):
  if t == None:
    return False
  else:
    if t["contents"] == target:
        return True
    leftSide = search(t["left"], target)
    rightSide = search(t["right"], target)
    return leftSide or rightSide
```

To do binary search on a tree, our tree must be sorted. Binary search trees (BST), are a special binary tree that is sorted.

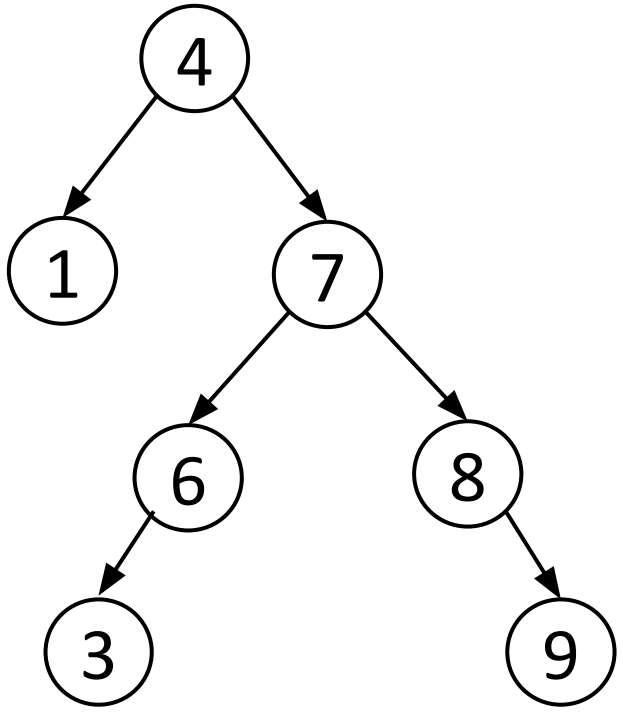
A BST has these constraints:

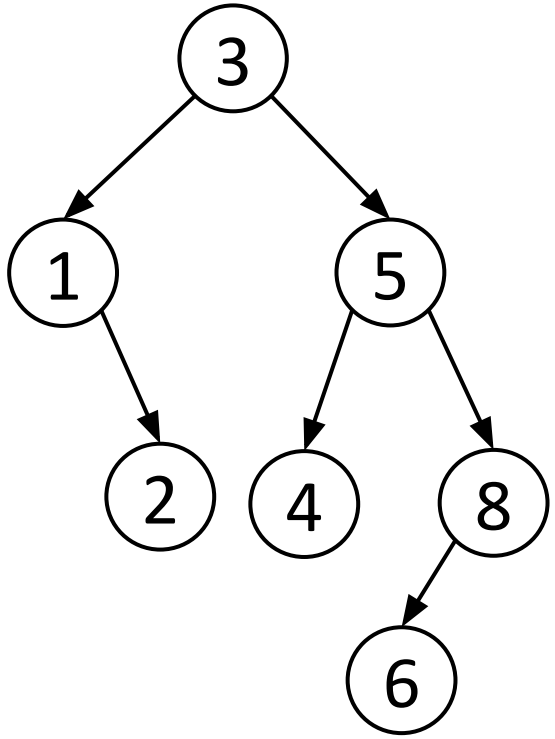
For **every** node **n** with value **v**:

- Each **left child** (and all its children, etc.) must have values strictly **less than v**
- Each **right child** (and all its children, etc.) must have values strictly **greater than v**



Example: Is this a BST?



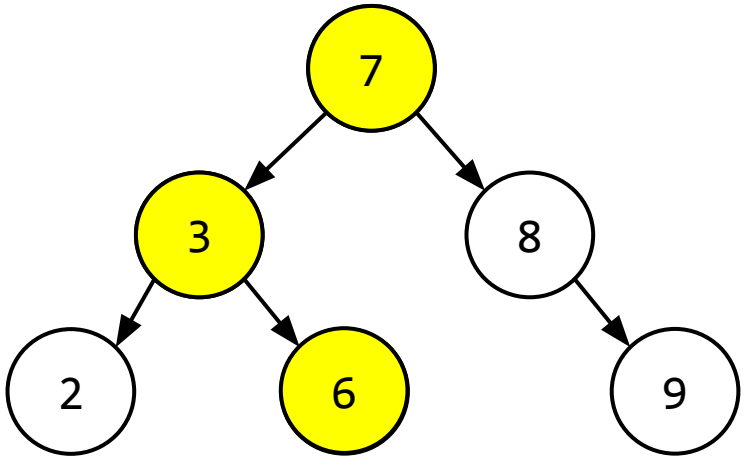


Searching a BST with binary search:

If we are searching for 6:

6 < 7: So we only have to search the left subtree

6 > 3: So we only have to search the right subtree.



We would write **binary search for a BST in Python**:

```
def search(t, target):
    if t == None:
        return False
    else:
        if t["contents"] == target:
            return True
        elif target < t["contents"]:
            return search(t["left"], target)
        else:
            return search(t["right"], target)
```

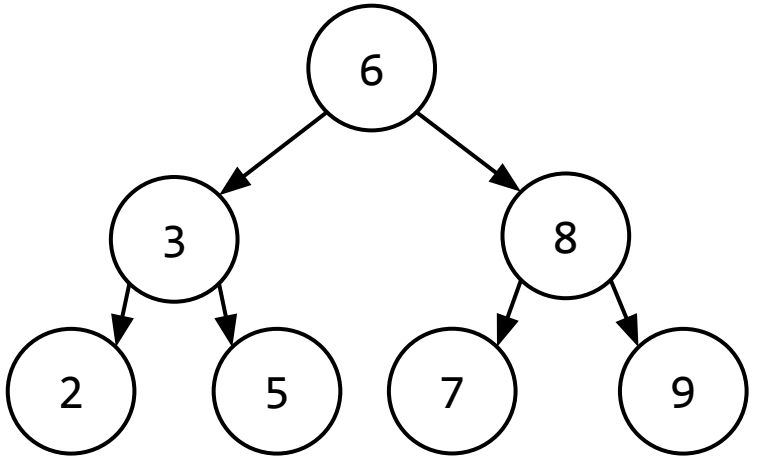
We do just **one** recursive call, either on the left subtree or on the right subtree.

The runtime of binary search on a **balanced tree is O(log n)**.

A tree is **balanced** if for every node in the tree, **the node's left and right subtrees are approximately the same size**.

This results in a tree that minimizes the number of recursive levels.

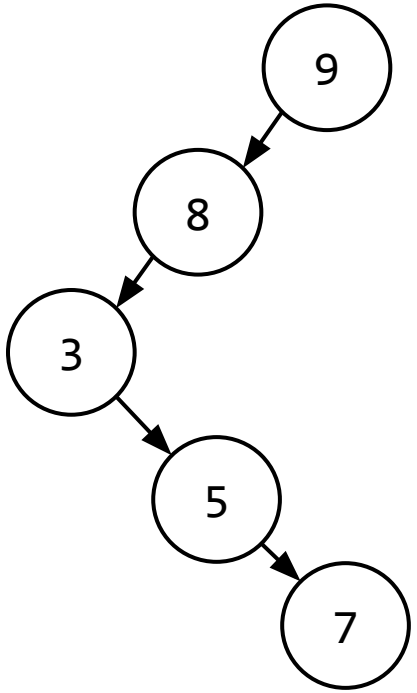
Every time you take a search step in a balanced tree, you cut the number of nodes to be searched in half.



The runtime of binary search on an **unbalanced tree is O(n)**.

A tree is considered unbalanced if at least one node has **significantly different sizes in its left and right children.**

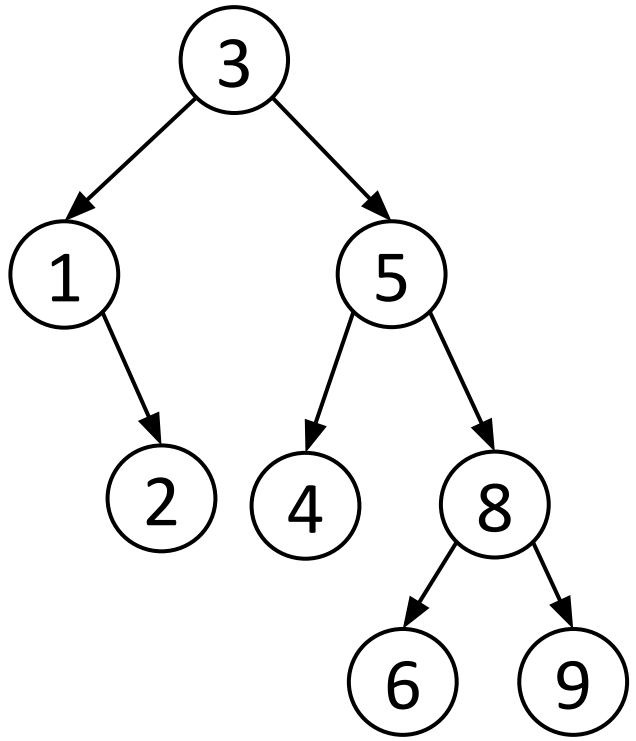
This is a valid BST, but it is still difficult to search! You must visit every single node to determine a number like 6 isn't in the tree. In the worst case, this can still take O(n) time.



BST does have **benefits** over a list.

It is easier to add things to a BST while keeping the data structure sorted.

This is very helpful for systems like hospital priority queues, where patients frequently need to be moved around the queue based on changing circumstances.



Can we **do even better** than binary and linear search?

We did better than linear search by assuming that the list was sorted.

We can often increase the efficiency of an algorithm by **thinking about the problem in a different way**. Try using a different data structure or an entirely different algorithmic approach to solve the problem. Or try putting **new constraints** on the problem to speed the process up.

Goal: Can we add additional constraints to design the **fastest possible** search algorithm?

Optimizing Search w/ Constraints

Search in Real Life – Post Mail Boxes

Consider how you receive mail. Your mail is sent to the post boxes at the lower level of the UC. Do you have to check every box to find your mail?

No - you just **check the box assigned to you**.

This is possible because your mail has an address on the front that includes your mailbox number. Your mail will only be put into a box that has the same number as that address, not other random boxes. **Picking up your mail is a O(1) operation!**

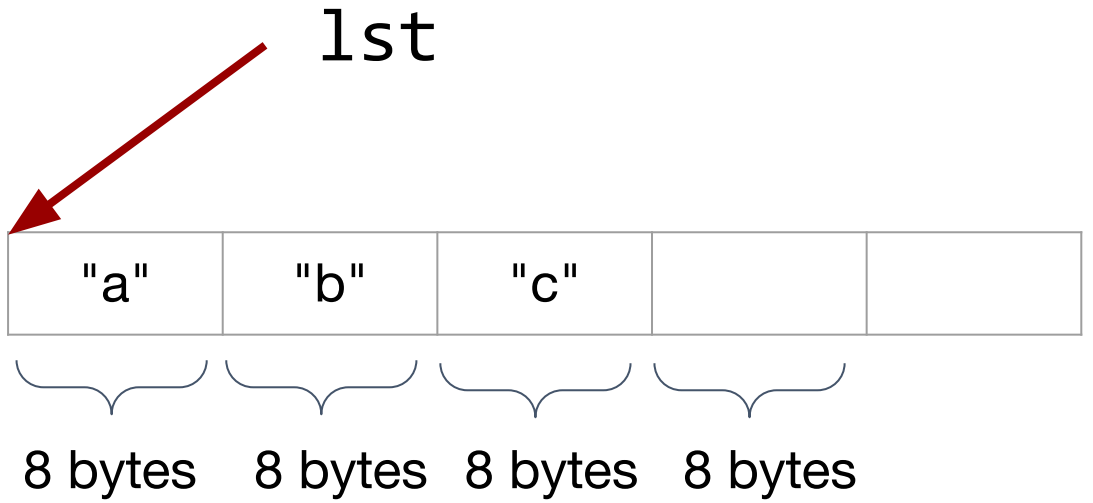
Compare this to picking up a package. Everyone picks up packages at the same window, so you must wait in line. If there are n students, picking up a package is a O(n) operation.

Search in Programming – List Indexes

We can look up an item in a list by its index quickly.

Python stores lists in memory as a series of adjacent parts. Each part holds a single value in the list, and all these parts use the same amount of space.

Can calculate the memory location of the item lst[i] quickly.




```
>> lst[2]
"c"
```

To **implement super-fast search**, we want to combine the ideas of post mail boxes and list index lookup.

We want to determine which index a value should be stored in **based on the value itself**.

If we can **calculate the index based on the value**, we can retrieve the value really quickly without needing to check other indexes.

value	index
"apple"	21
"banana"	145
"grape"	62

```
f("apple")  = 21
f("banana") = 145
f("grape")  = 62
```

A function that maps values to integers is a **hash function**.

Good hash functions have the following:

1. The function should be **deterministic**.

   For every value `x`:

   `hash(x)` must always equal the same value `v`

2. The function should generally produce **different outputs for different inputs**.

   For most values `x` and `y` where `x ≠ y`:

   `hash(x) ≠ hash(y)`

Python has a **built-in hash function**: `hash`.

```
x = "abc"
hash(x) # some giant number
```

By default, **hash only works on immutable objects** including: int, float, bool, str

Question: Why not on mutable objects?

Optimizing Search w/ Hash Tables

A **hash table** uses a hash function to **determine where to store values** in N buckets.

Each bucket has an **index**.

When we place a value in the table, **we put it into a bucket based on its hash value**.

A hash table with 5 buckets:

0	
1	
2	
3	
4	

Example hash table implementation:
With a hash function for integers using **mod %**

Suppose we have a hash table with 5 buckets and our input is always going to be an integer.

We can determine which bucket to put an integer into with mod:

```
# bucket = value % 5
```

A hash table with 5 buckets:

0	
1	
2	
3	
4	

Example hash table implementation:
With a hash function for integers using **mod %**

Suppose we have a hash table with 5 buckets and our input is always going to be an integer.

We can determine which bucket to put an integer into with mod:

```
# bucket = value % 5
```

Insert 2: 2 % 5 = 2

0	
1	
2	2
3	
4	

Example hash table implementation:
With a hash function for integers using **mod %**

Suppose we have a hash table with 5 buckets and our input is always going to be an integer.

We can determine which bucket to put an integer into with mod:

```
# bucket = value % 5
```

Insert 15: 15 % 5 = 0

0	15
1	
2	2
3	
4	

Example hash table implementation:
With a hash function for integers using **mod %**

When two inputs map to the same bucket, this is called a **hash collision**.

Question: How to deal with collisions?

Many possible solutions. One example is to make the bucket contain a list.

Adding 17: 17 % 5 = 2

0	[15]
1	
2	[2, 17]
3	
4	

What about a **hash function for strings**?

Example function:

Compute the len of the string and use that to determine the bucket.

bucket = len(s) % 5

Insert "grape": 5 % 5 = 0

0	"grape"
1	
2	
3	
4	

What about a **hash function for strings**?

Example function:

Compute the len of the string and use that to determine the bucket.

bucket = len(s) % 5

Insert "banana": 6 % 5 = 1

0	"grape"
1	"banana"
2	
3	
4	

Activity: Search a hash table.

Let's say that we want to algorithmically check whether the string "apple" is in a hashtable with 1000 buckets.

You do: Which buckets does the algorithm need to check?

What is the Python code that would return the bucket to check?

0	
1	
2	
3	
..	
1000	

Searching a hash table is **O(1)**.

To search a hash table compute the hash function and mod by the number of buckets.

```
hash(x) % numBuckets
```

Then you **only have to search one bucket, regardless of the size of the hashtable**! Very fast!

0	
1	
2	
3	
..	
1000	

We cannot hash **mutable values**!

Example:
A hash function for a list that takes the first element as the value to hash.

Insertion is ok.

Insert: [2,5,8]

0	
1	
2	[2,5,8]
3	
4	
5	

We cannot hash **mutable values!**

Example:
A hash function for a list that takes the first element as the value to hash.

Insertion is ok.

But lookup is broken. Because we can change the list, we would look for it in the wrong place.

Insert 4 at index 0 of [2,5,8]:
.

0	
1	
2	[4,2,5,8]
3	
4	
5	

Dictionaries are implemented using **hash tables**.

We run the **hash function on the keys** to find the values.

This **explains why** the dictionary keys:

- Must be immutable
- Have to be unique
- Are unordered

So **insertion and lookup (search) in a dictionary is O(1)**!

```
{"A": "ants", "B": "bats"}
```

0	
1	`"B": "bats"`
2	
3	`"A": "ants"`
..	
N	

Learning Objectives

- Identify whether a tree is a **tree**, a **binary tree**, or a **binary search tree (BST)**
- Search for values in trees using **linear search** and in BSTs using **binary search**
- Analyze the **efficiency** of binary search on a **balanced** vs. **unbalanced** BSTs
- Recognize the requirements for building a good **hash function** and a good **hash table** that lead to **constant-time search**