Basics of Storage

Spring 2025

Thinking about storage

In a data structures course, you may have seen data structures for storing and retrieving items within a program

Most questions about how to store data in a database are extensions of the same questions about data structures

Today is about reviewing what we know about data structures for storage and retrieval

What is data?

In general, it is a collection of records

A record has fields and associated values

Example: a book record for a library database

```
ISBN: ...
Author: ...
Title: ...
Publisher: ...
Publication date: ...
```

We'll examine data structures for primitive values before doing records

CRUD operations

CRUD operations are primitive operations for any storage-based data structure

- **Create**: create (insert) records into the data structure
- **Read**: read (find) records in the data structure
- **Update**: update (modify) records in the data structure
- **Delete**: delete records from the data structure

In many data structures, you can get **Update** by doing a **Delete** followed by a **Create**

CRUD operations

Fundamental trade-off of every data structure

How do you make the various CRUD operations efficient?

- there is usually a tension between **find** and **insert**
- to **find** quickly, we need additional structure
- but **insert** needs to maintain that additional structure

We can also often trade off speed and space (faster may require more space)

Big-O notation

Remember: O(-) notation

- Roughly the number of “primitive” steps performed by an operation as a function of the size of the input in the worst case
- Up to a constant, and keeping only the highest order terms

E.g., $O(1)$ = constant time
$O(n)$ = linear in the size *n* of the input
$O(n^2)$ = quadratic in the size *n* of the input
$O(2^n)$ = exponential in the size *n* of the input

Arrays

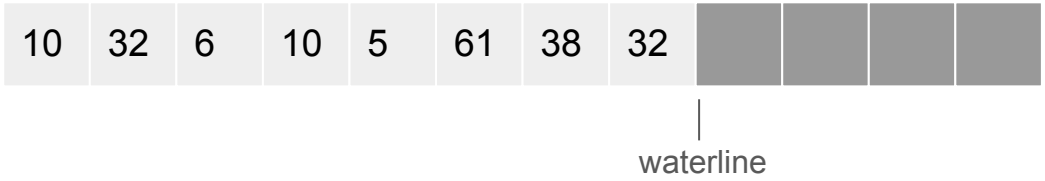
10	32	6	10	5	61	38	32	45	55	66	9

Simplest representation is an array of items

Find:	scan array to find item	O(n)
Insert:	copy array into new array of size n + 1	O(n)
	add item at the end:	O(1)

Note: I'm assuming throughout we can get the length of an array in O(1) time

Arrays with capacity



Preallocate an array of a given capacityq

Find:	scan array to waterline to find item	O(n)
Insert:	if waterline not at capacity, add past waterline	O(1)
	otherwise, copy array into new array with larger capacity	O(n)
	and add item past new waterline	O(1)

Running time trickier to calculate (requires amortized analysis) but ~O(1)

Runtime analysis

	Array	Array with capacity
Find	O(n)	O(n)
Insert	O(n)	O(1)*
Delete	O(n)	O(n)

Sorted arrays

Can we improve the **find** operation?

- keep items sorted — does not improve asymptotic running time
- but combine with binary search…
  -

Binary search

- start in the center of the array
- determine if what you're looking for is left or right of the center item
- recursively do a binary search in the appropriate half of the array

Binary search in sorted arrays

Find 55

1	3	6	10	15	21	28	36	45	55	66	78

Binary search in sorted arrays

Find 55

1	3	6	10	15	21	28	36	45	55	66	78

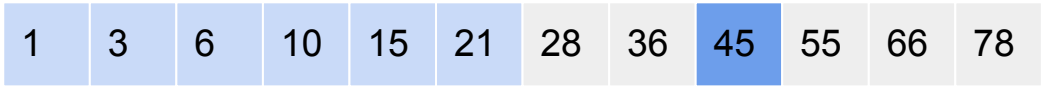
Binary search in sorted arrays

Find 55

1	3	6	10	15	21	28	36	45	55	66	78

Binary search in sorted arrays

Find 55



Binary search in sorted arrays

Find 55

1	3	6	10	15	21	28	36	45	55	66	78

Binary search in sorted arrays

Find 55

Binary search in sorted arrays

Find 55

1	3	6	10	15	21	28	36	45	55	66	78

Binary search in sorted arrays

Find 55

Sorted arrays (with binary search)

Runtime analysis:

Find:	use binary search	O(log n)
Insert:	find where the item would go using binary search	O(log n)
	copy existing array into new array (or use capacity)	O(1) or O(n)
	shift everything past new item position to the right	O(n)

So we improved finding an item at the cost of inserting an item

Runtime analysis

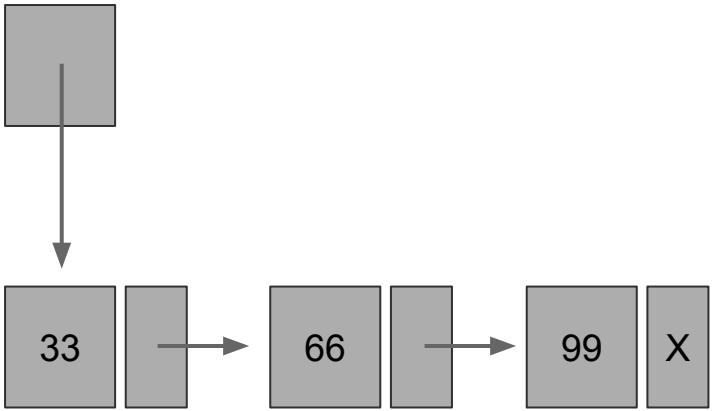
	Array	Array with capacity	Sorted array with capacity
Find	O(n)	O(n)	O(log n)
Insert	O(n)	O(1)*	O(n)
Delete	O(n)	O(n)	O(n)

Linked lists

Insert for sorted arrays is slow because we need to create a spot for the new item between two cells of the array

What if we could just slide in a new cell without shifting the array content?

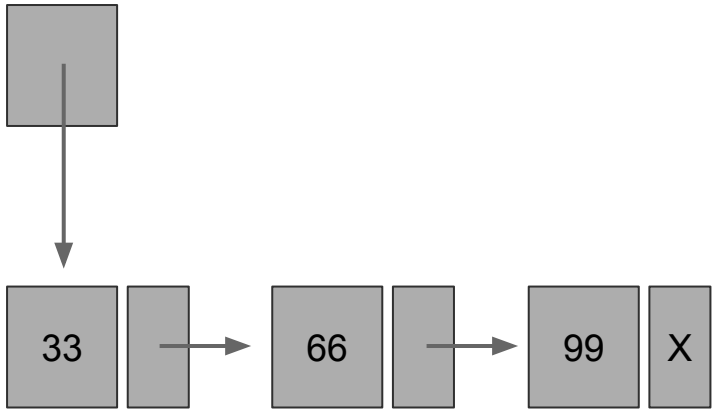
Move to a world where every array cell is separate, and tells you which cell is next in the array



Linked lists

Find: scan from the head of list O(n)

Insert: scan to find where it goes O(n)
surgery to add new node O(1)

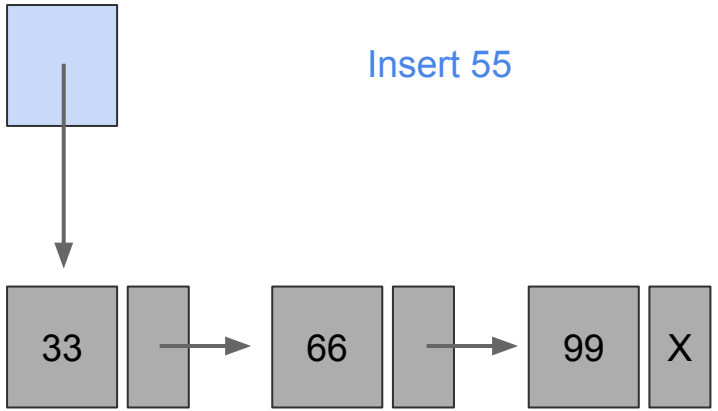


Note: there is no array in this representation — each node of the list is allocated separately in memory

Linked lists

Find: scan from the head of list O(n)

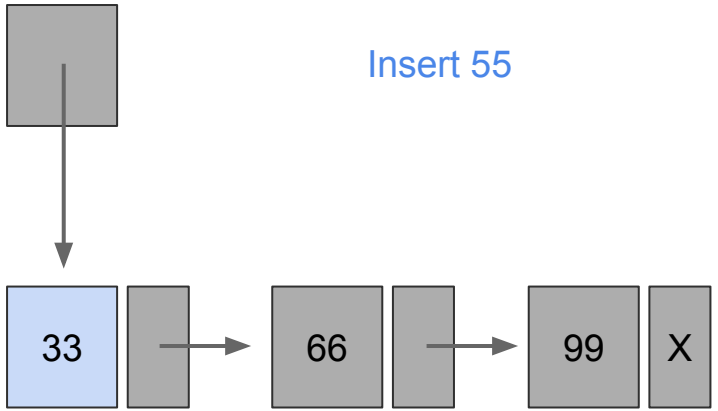
Insert: scan to find where it goes O(n)
surgery to add new node O(1)



Linked lists

Find: scan from the head of list O(n)

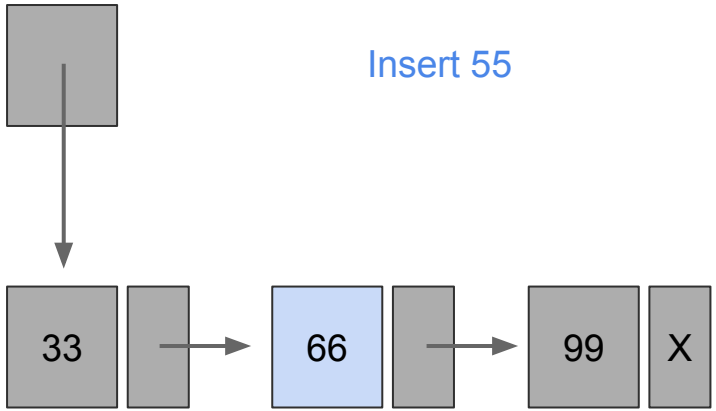
Insert: scan to find where it goes O(n)
surgery to add new node O(1)



Linked lists

Find: scan from the head of list O(n)

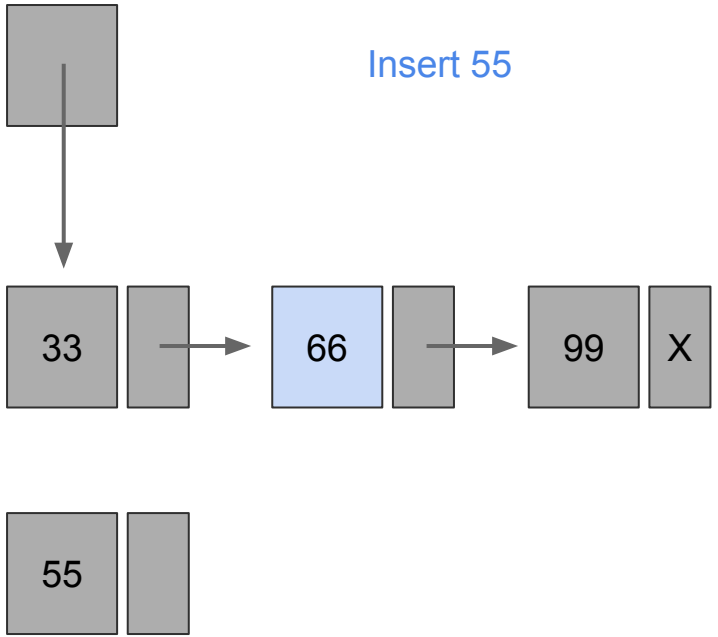
Insert: scan to find where it goes O(n)
surgery to add new node O(1)



Linked lists

Find: scan from the head of list O(n)

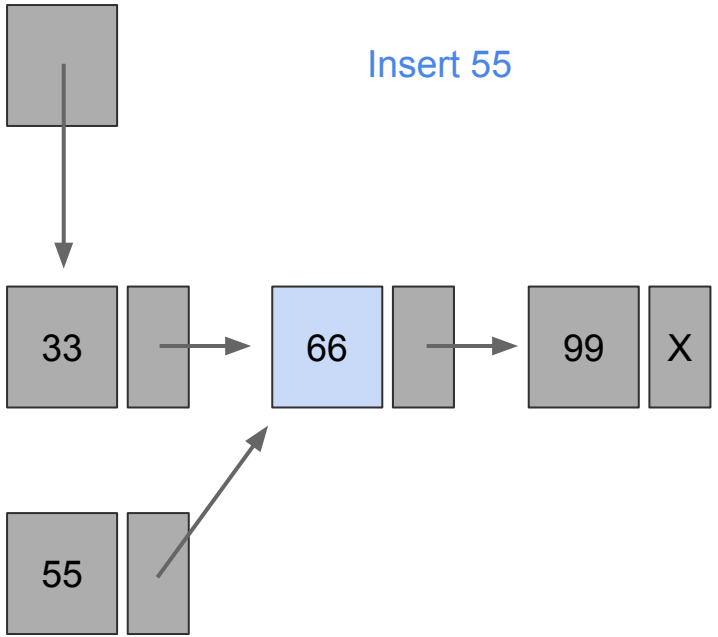
Insert: scan to find where it goes O(n)
surgery to add new node O(1)



Linked lists

Find: scan from the head of list O(n)

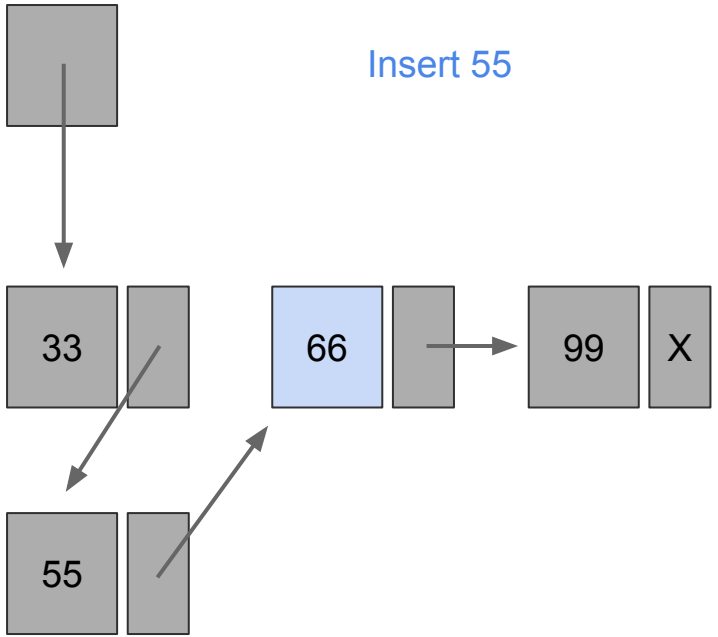
Insert: scan to find where it goes O(n)
surgery to add new node O(1)



Linked lists

Find: scan from the head of list O(n)

Insert: scan to find where it goes O(n)
surgery to add new node O(1)

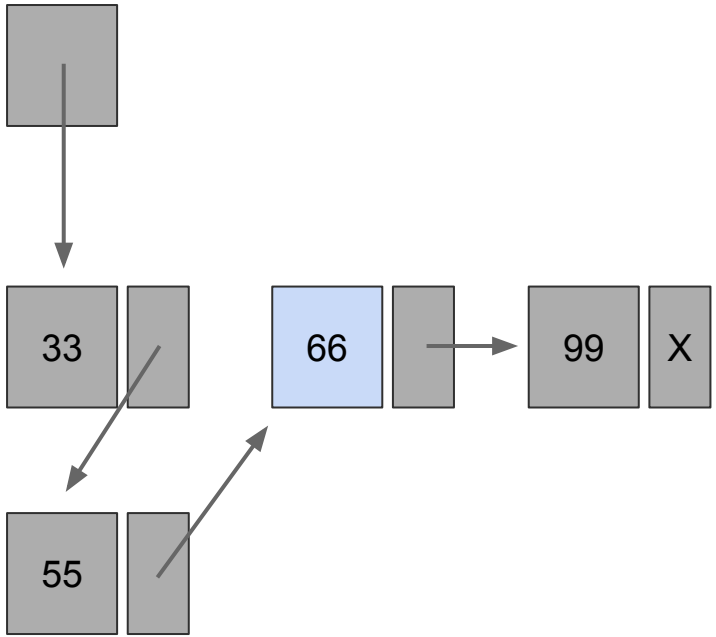


Linked lists

Find: scan from the head of list O(n)

Insert: scan to find where it goes O(n)
surgery to add new node O(1)

We've lost the advantage of binary search!



Runtime analysis

	Linked list	Sorted Linked List
Find	O(n)	O(n)
Insert	O(1)	O(n)
Delete	O(n)	O(n)

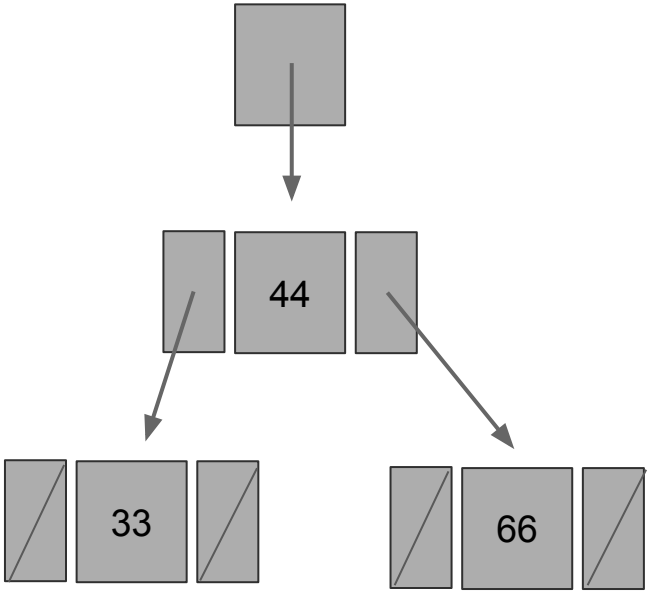
Binary search trees

Take a linked list

Instead of using a pointer to the first item, link to the "center" item

From "center" item, link to the "left side of list" and to the "right side of list"

Apply recursively at all levels

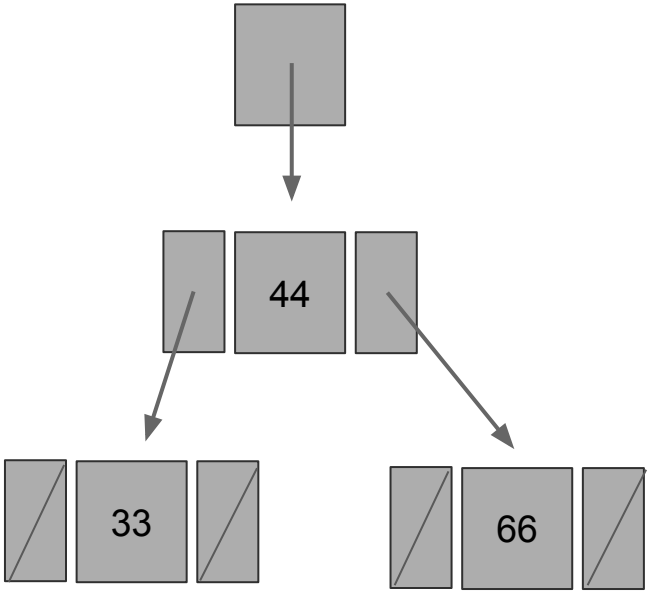


Binary search trees

A binary tree with the binary search property (a binary search tree):

Every node in the left subtree of node N has value < N

Every node in the right subtree of node N has value > N



Find: follow left/right subtrees O(height)

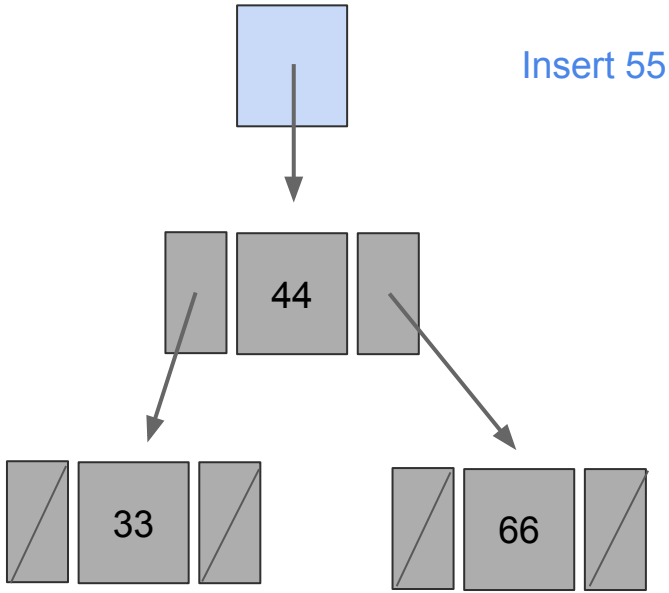
Insert: same, then add node O(height)

Binary search trees

A binary tree with the binary search property (a binary search tree):

Every node in the left subtree of node N has value < N

Every node in the right subtree of node N has value > N



Find: follow left/right subtrees O(height)

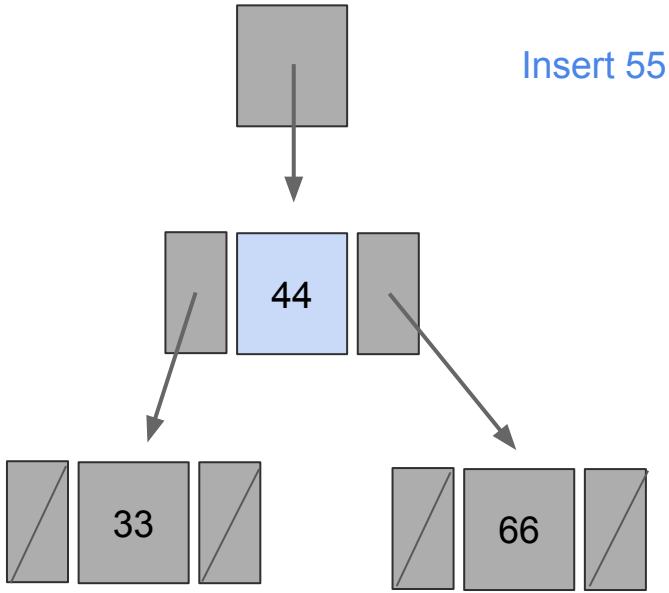
Insert: same, then add node O(height)

Binary search trees

A binary tree with the binary search property (a binary search tree):

Every node in the left subtree of node N has value < N

Every node in the right subtree of node N has value > N



Find: follow left/right subtrees O(height)

Insert: same, then add node O(height)

Binary search trees

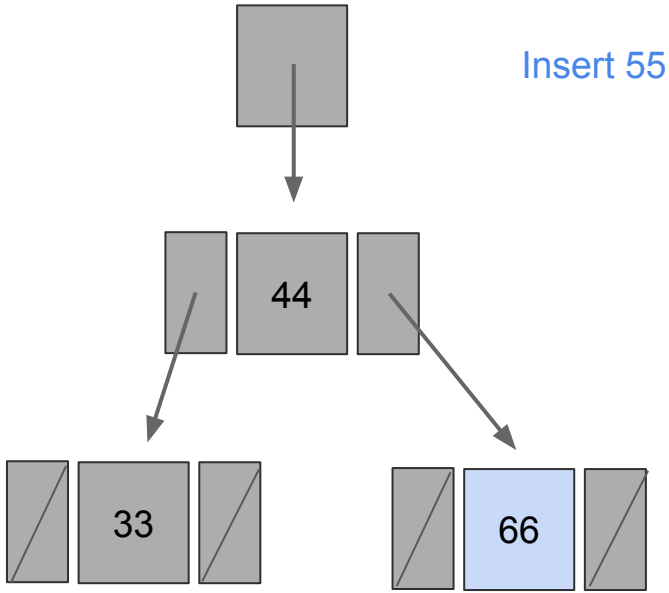
A binary tree with the binary search property (a binary search tree):

Every node in the left subtree of node N has value < N

Every node in the right subtree of node N has value > N

Find: follow left/right subtrees O(height)

Insert: same, then add node O(height)



Binary search trees

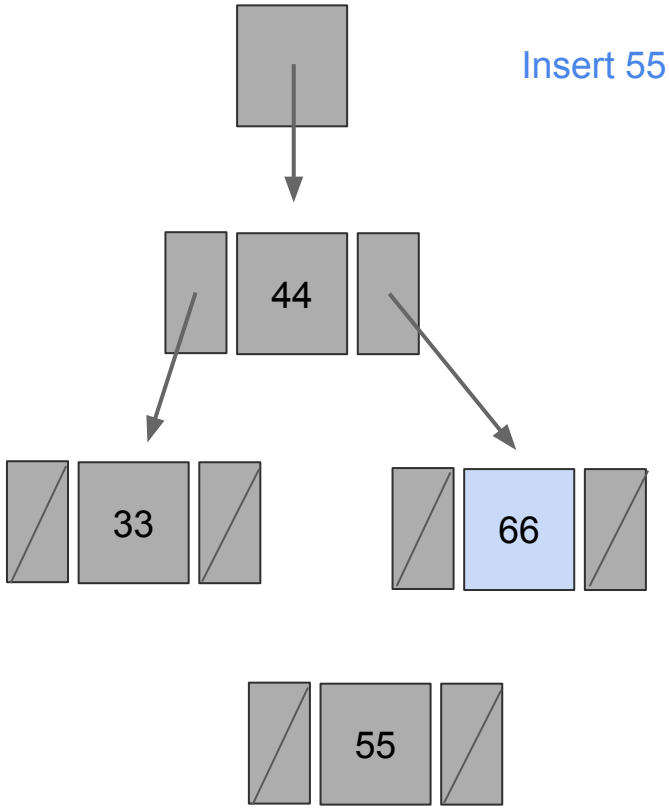
A binary tree with the binary search property (a binary search tree):

Every node in the left subtree of node N has value < N

Every node in the right subtree of node N has value > N

Find: follow left/right subtrees O(height)

Insert: same, then add node O(height)



Binary search trees

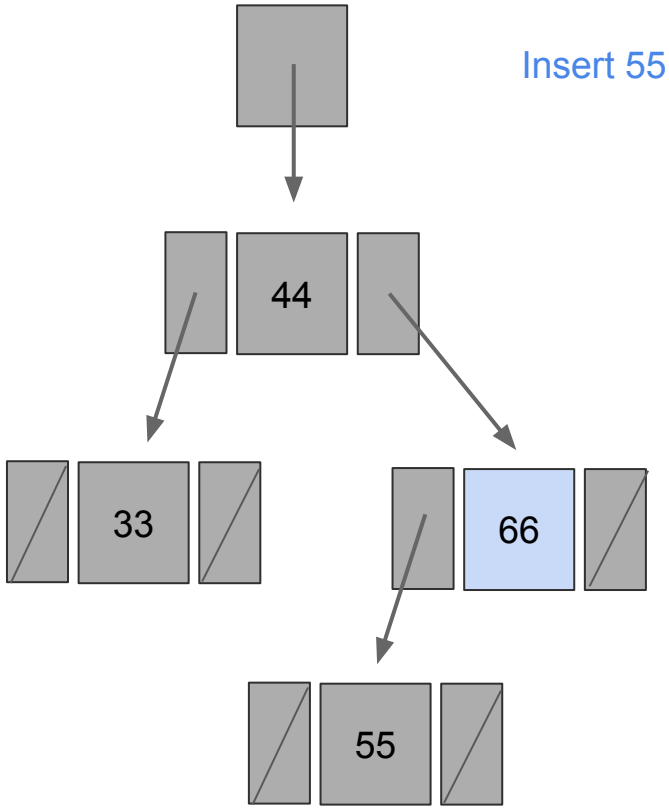
A binary tree with the binary search property (a binary search tree):

Every node in the left subtree of node N has value < N

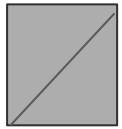
Every node in the right subtree of node N has value > N

Find: follow left/right subtrees O(height)

Insert: same, then add node O(height)



Binary search trees



Finding in a sorted array with binary search is always O(log n)

Not so for a binary search tree

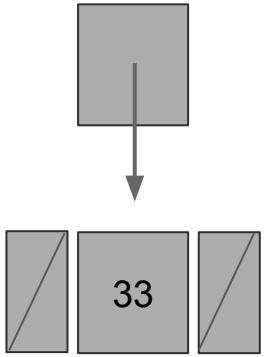
It's easy to get a binary search tree that's *unbalanced* — height is O(n)

Binary search trees

Finding in a sorted array with binary search is always O(log n)

Not so for a binary search tree

It's easy to get a binary search tree that's *unbalanced* — height is O(n)

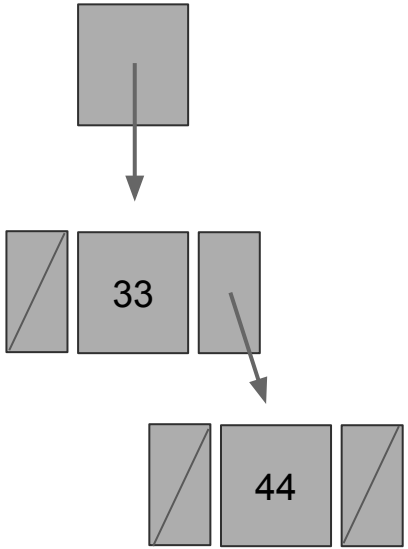


Binary search trees

Finding in a sorted array with binary search is always O(log n)

Not so for a binary search tree

It's easy to get a binary search tree that's *unbalanced* — height is O(n)

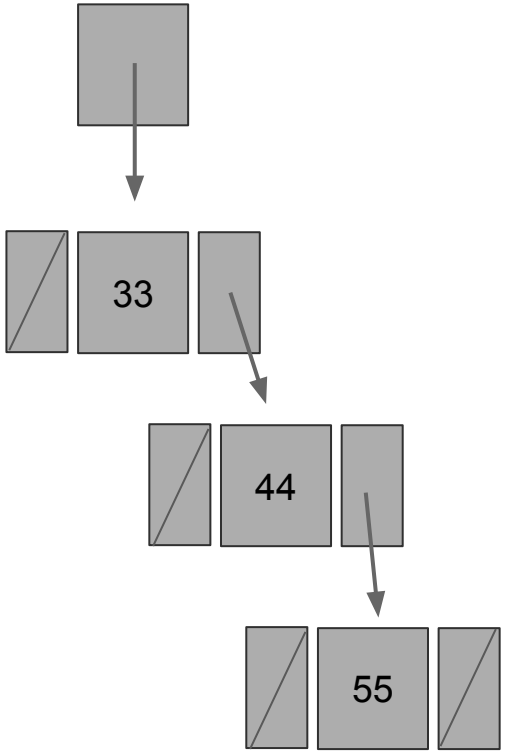


Binary search trees

Finding in a sorted array with binary search is always O(log n)

Not so for a binary search tree

It's easy to get a binary search tree that's *unbalanced* — height is O(n)

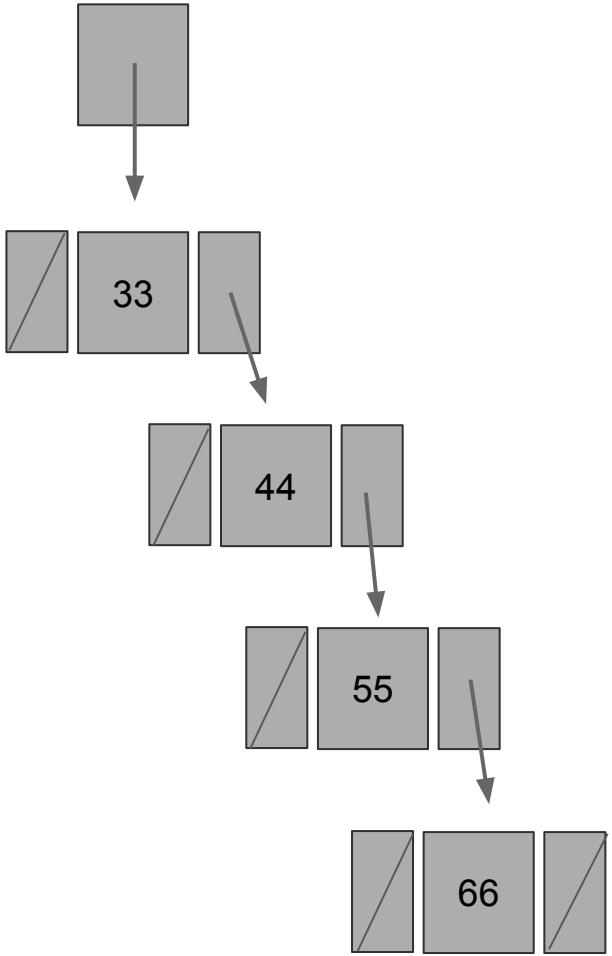


Binary search trees

Finding in a sorted array with binary search is always O(log n)

Not so for a binary search tree

It's easy to get a binary search tree that's *unbalanced* — height is O(n)



Runtime analysis

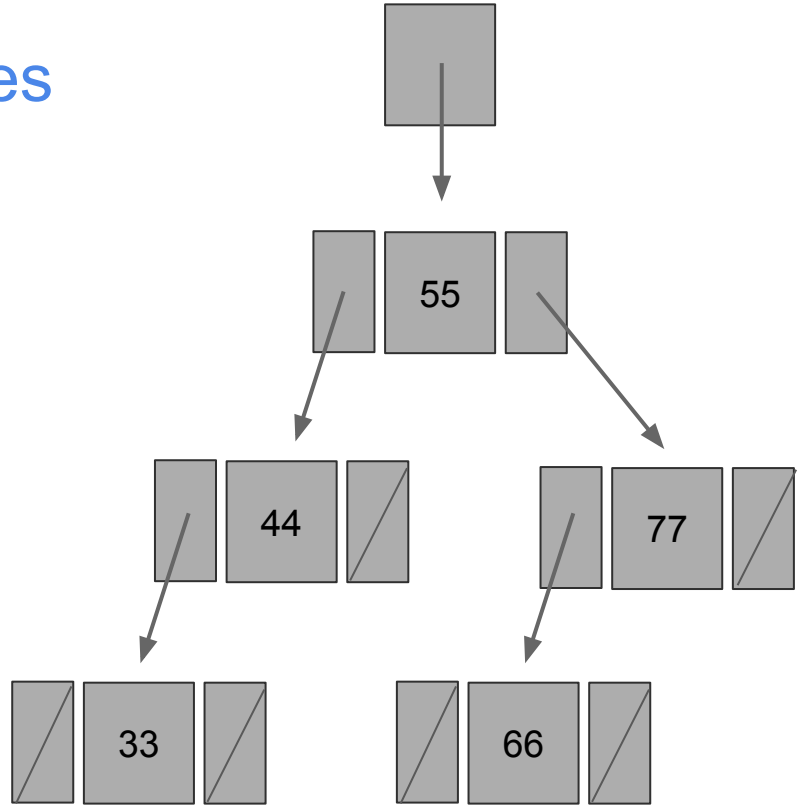
	Binary search tree
Find	O(n)
Insert	O(n)
Delete	O(n)

Balanced binary search trees

Balanced binary search trees have additional properties that ensure that they're always balanced

Height is O(log n)

- AVL trees
- Red-black trees
- 2-3 trees (not binary!)



Runtime analysis

	Binary search tree	Balanced binary search tree
Find	O(n)	O(log n)
Insert	O(n)	O(log n)
Delete	O(n)	O(log n)

Runtime analysis

	Binary search tree	Balanced binary search tree	Array with capacity
Find	O(n)	O(log n)	O(n)
Insert	O(n)	O(log n)	O(1)*
Delete	O(n)	O(log n)	O(n)

In practice

Arrays with capacity are used in Python, Javascript, Java to implement lists

- fast push to the end of list
- slower insert in the middle of a list
- scan to find

Balanced binary search trees (or hash tables) used for structures with fast find

- Dictionaries in Python, objects in Javascript, maps in Java

Relational databases store data on disk using a form of balanced search tree that works well with extremely large files (don't read the whole file to find a record)

Constant-time find and insert operations?

Can we make both **find** and **insert** constant-time?

We'd need to know exactly where in the structure an item goes

- we can't afford to look for it

One solution:

Compute the position of an item in the structure from the item itself

Constant-time find and insert operations?

Intuition:

- suppose items are integers {0, …, N}
- allocate an array of size N + 1, with initial values -1
- store item X at position X in the array
- look for item X at position X in the array

What about when elements are not integers {0, …, N}?

Constant-time find and insert operations?

Intuition:

- suppose items are integers {0, …, N}
- allocate an array of size N + 1, with initial values -1
- store item X at position X in the array
- look for item X at position X in the array

What about when elements are not integers {0, …, N}?

- Use a hash function from integers to {0, …, N} for some N
- example: h(X) = X mod (N + 1)
- If items are not integers, convert them to integers (somehow)

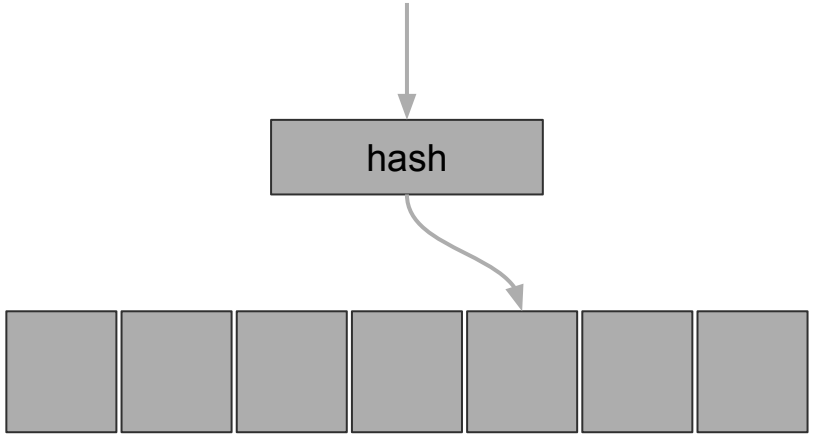
Hash table

A hash table is:

- an array T of size N + 1
- a hash function h from items to {0,, …, N}

Intuition:

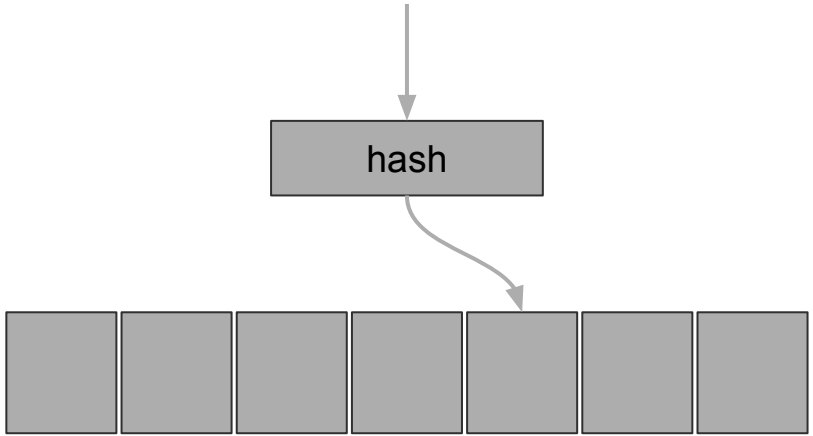
- to insert item X: T[h(X)]
- to find item X: T[h(X)]



Hash table

A hash table is:

- an array T of size N + 1
- a hash function h from items to {0,, …, N}



Intuition:

- to insert item X: T[h(X)]
- to find item X: T[h(X)]

If you have M > N + 1 elements, then there must be a **collision**

some $X_1 \neq X_2$ with $h(X_1) = h(X_2)$ how do we resolve collisions?

Working with records

As I said at the beginning, by data we usually mean collection of records

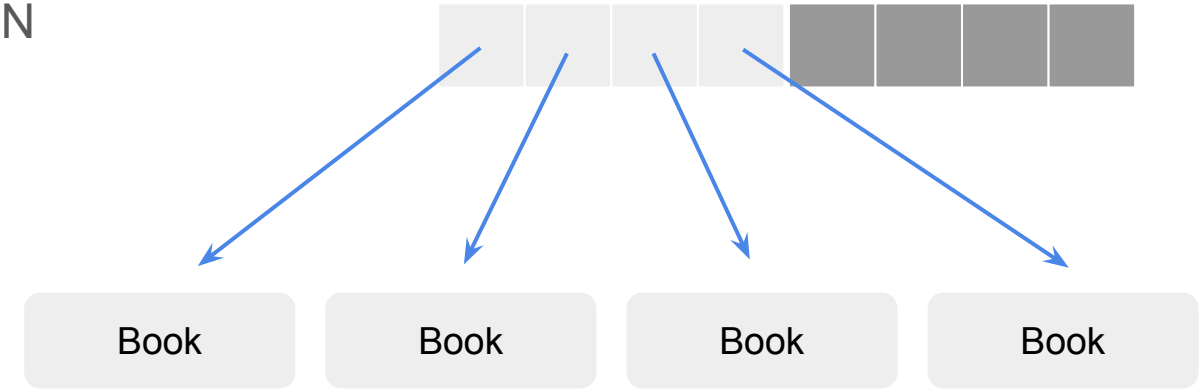
Example: a book record for a library database

```
ISBN: ...
Author: ...
Title: ...
Publisher: ...
Publication date: ...
```

A record needs a key to use as the value to sort by in the data structure

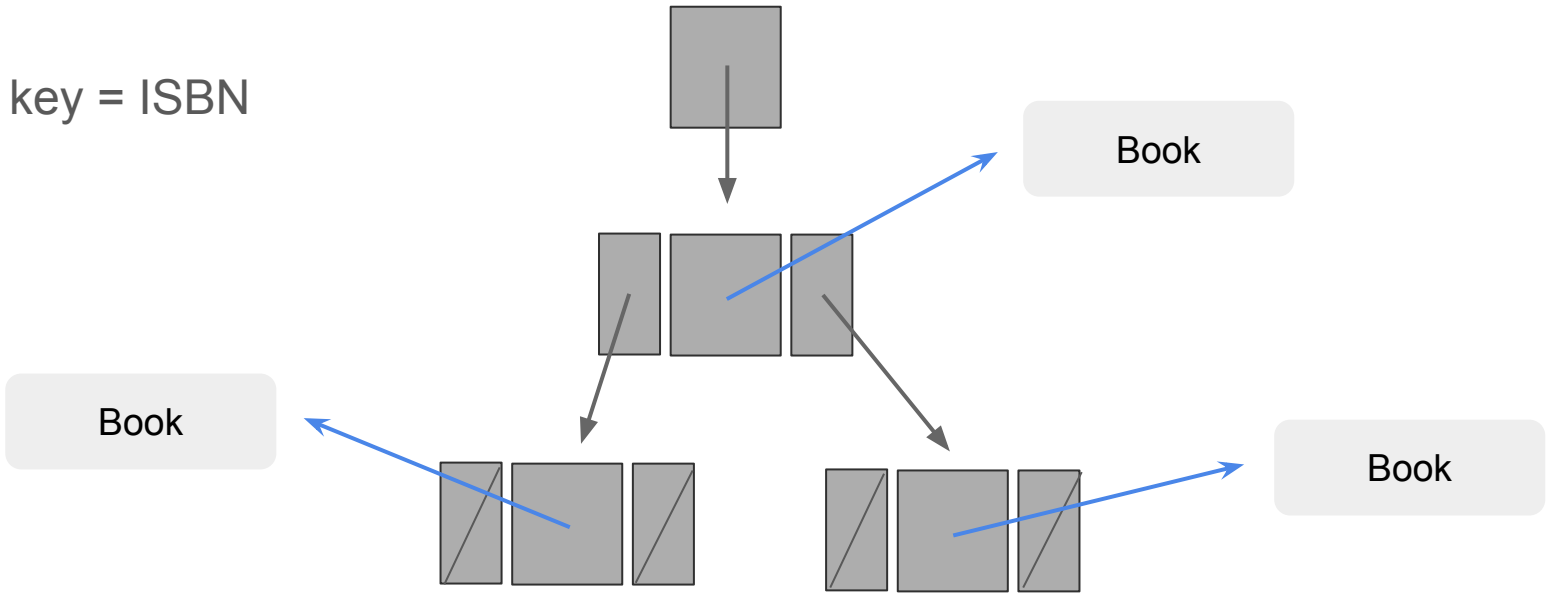
Sorted array with capacity with records

key = ISBN



Binary search tree with records

key = ISBN



Finding across multiple keys

The field we use as a key is what's optimized for the **find** operation

- if we use ISBN as a key, then we can find a book by ISBN quickly

But often we want to be able to find by multiple fields

- what if we also wanted to find a book by title?

We create a new structure keyed by this new field (and we maintain it alongside)

- called an index in the database literature

Sorted array with capacity with records and index