CS 250: Foundations of Computer Systems
[Data Structures for Storage Systems]

Dynamic data structures

Feeling smug
About that data structure
Can it dance?
When the data chooses to prance
Can it zig?
When the data zags
Or has it tied itself?
Up in knots

Shrideep Pallickara
Computer Science
Colorado State University

1

Frequently asked questions from the previous class survey

- TCP & UDP & IPv6
  - Are the ports in TCP and UDP operationally related?
  - How many next headers can there be? [8 bits for NextHeader]
- Private IP addresses
  - How many public address does CSU have?
  - Why have a MAC address when I have a private IP address?
  - If addresses are specific to a customer, how can there be multiple private IP addresses?

2

Frequently asked questions from the previous class survey

- DNS
  - How are hostnames and IP addresses related to each other?
  - Relation to IP? Is it any way part of the TCP/IP stack?
  - Overview? How can there be a cache?
  - How does DHCP actually work?
  - Standards body for domain names?
- OSI
  - What is it used for?
  - Difference between the IP stack vs OSI stack?

Topics covered in this lecture

- Some basics
  - Variables and arrays
- Linear scan
- Binary search
- Dynamic data structures

5

Any remotely interesting computer program needs to be able to store and access data from memory

- Individual pieces of data are often stored in **variables**
- Variables are essentially names representing the location (or address) of a piece of data in the computer's memory
- Without variables, a programmer can't track, evaluate, or update the **program's internal state**
  - When you create a variable, the system *allocates* and *assigns* it a location behind the scenes

6

In most programming languages, variables have an associated type

- The variable's *type* denotes exactly what type of data they store
  - E.g.: integers, "floats" for floating-point values, or Booleans for true or false values
- These types tell the program *how much* memory the variable occupies and *how to* use it

Why do we have arrays?

An array is generally used to store multiple related values

- Arrays provide a simple mechanism for storing multiple values in adjacent and indexable bins
- An array is effectively a row of variables—a contiguous block of **equal-sized bins** in the computer's memory

Accessing array elements

- The structure of an array allows you to access any value, also known as an element, within the array by specifying its location, or index
- The bins occupy **adjacent locations** in the computer's memory
  - So we can access individual bins by computing their **offset from the first element** and reading the memory in that location
  - This requires just a single addition and memory lookup regardless of which bin we access
- This structure makes arrays especially convenient for storing items that have an **ordered relationship**

Formally, we reference the value at index i of array A as A[i]

- Most programming languages use **zero-indexed** arrays
- This means the first value of the array resides at index 0, the second at index 1, etc.
- Zero-indexing conveniently allows us to compute an element's location in memory as an **offset** from where the array starts in memory
- The location of the ith item in the array can be computed by:
  - Location(item i) = Location(start of array) + Size of each element × i
  - The location of the element at index zero is the start of the array

Examples of zero and non-zero indexed arrays in programming languages

- Zero-indexed arrays
  - C, C++, Java, Python, C#, Lisp, Javascript, Rust, Go
- Non zero-indexed arrays
  - ALGOL 68, APL, AWK, CFML, COBOL, Fortran, FoxPro, Julia, Lingo, Mathematica, MATLAB, Sass, Smalltalk, Wolfram Language

DATA STRUCTURES FOR STORAGE SYSTEMS

Problem definition

- Given a set of N data points $\mathbf{X} = \{x_1, x_2, \ldots, x_N\}$ and a target value x'
- Find a point $x_i \in \mathbf{X}$ such that $x' = x_i$
  - Or indicate that no such point exists
- In our everyday lives, we would likely describe the task as "Find me this particular thing."

Let's start with a simpler solution: Linear scan

- **Linear scan** searches for a target value by testing each value in our list, one after the other, until:
  - The target is found or
  - We reach the end of our list

Linear Scan: Searching for the value 21

3	11	9	37	7	8	1	21	5	17	31	54

About Linear Scan

- **Thorough but inefficient**, especially for large lists
- Guaranteed to find the item of interest (if the item is in the data)
  - Checks *every possible* item until
    - It finds a match or confirms the item is missing
- Brute force!

Linear Scan: Why is this the case?

- We know nothing about the **structure** of the data
  - There is nothing we can do to streamline the process
- The target value could be in any bin, so we may need to check them all

What's next?

- Let's see how a small amount of structure in the data changes everything

19

20

Binary Search

- An algorithm for efficiently **searching a sorted list**
- Checks the sorted list for a target value by
  - Repeatedly dividing the list in half
  - Determining which of the two halves could contain the target value
    - And discarding the other half

Binary search

- Binary search is an algorithm to find a target value v in a sorted list
- **Only works on sorted data**
- The algorithm can be written to work with data sorted in either increasing or decreasing order
  - We will look at increasing order — lowest to highest

Secret sauce?

- The key to efficient algorithms is using **information or structure** within the data
- In the case of binary search, we use the fact that the array is sorted in increasing order

Binary search: The algorithm

1) Partition the list in half and determine in which half v must reside
2) Discards the half that v is not in

- Repeat the process with only the half that can possibly still contain v
  - Until only one value remains

More formally …

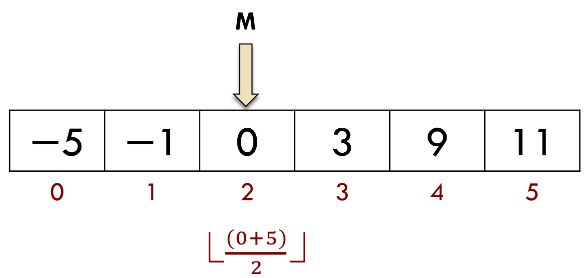
- Consider a sorted array A:
  - A[i] ≤ A[j] for any pair of indexes i and j such that i < j
- While this might not seem like a lot of information
  - It's enough to allow us to **rule out entire sections** of the array
- Similar to how we avoid the ice cream aisle when searching for coffee
  - Once we know an item won't be in a given area, we can rule out that entire set of items in that area without individually checking them

What Binary Search needs

- Binary search tracks the current **search space** with **two bounds**:
  - the **upper bound** IndexHigh marks the highest index of the array that is part of the active search space, and
  - the **lower bound** IndexLow marks the lowest
- Invariant: if the target value is in the array?
  - A[IndexLow] ≤ v ≤ A[IndexHigh]

Binary search: Details [1/2]

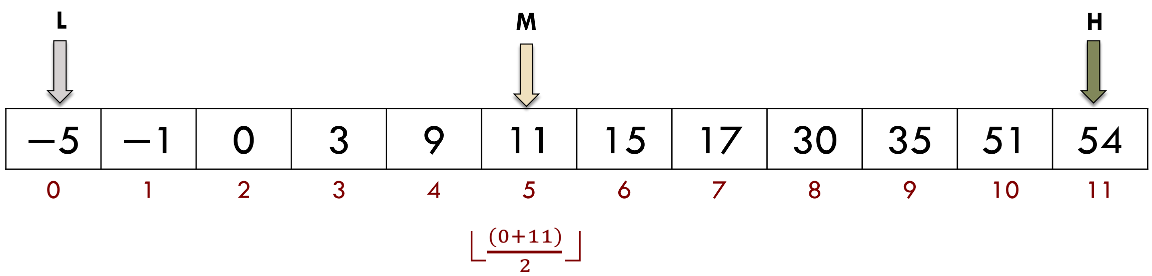
- Start each iteration by choosing the midpoint of the current **search space**:
  - IndexMid = Floor((IndexHigh + IndexLow) / 2)
    - Floor is a mathematical function that rounds a number down to an integer



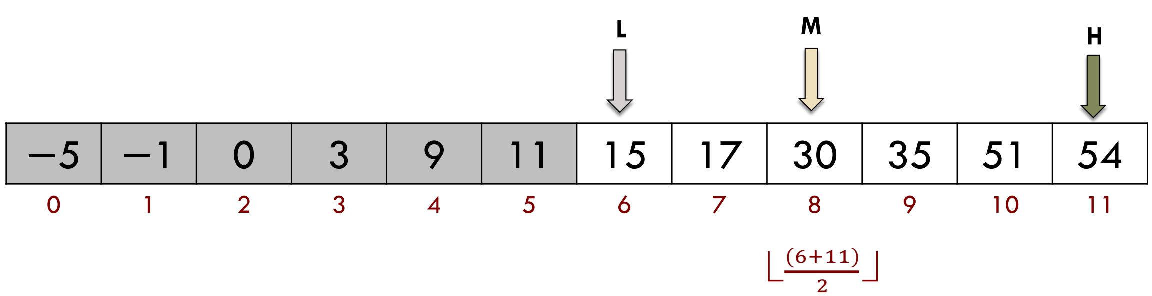
Binary search: Details [1/2]

- Compare value at the middle location, A[IndexMid], with the target value v
  - If A[IndexMid] < v, the target value must lie *after* the middle index
    - Allows us to chop the search space in half by making IndexLow = IndexMid + 1
  - if A[IndexMid] > v, the target value must lie *before* the middle index
    - Allows us to chop the search space in half by making IndexHigh = IndexMid – 1
  - Of course, if A[IndexMid] == v, we immediately conclude the search
    - We've found the target

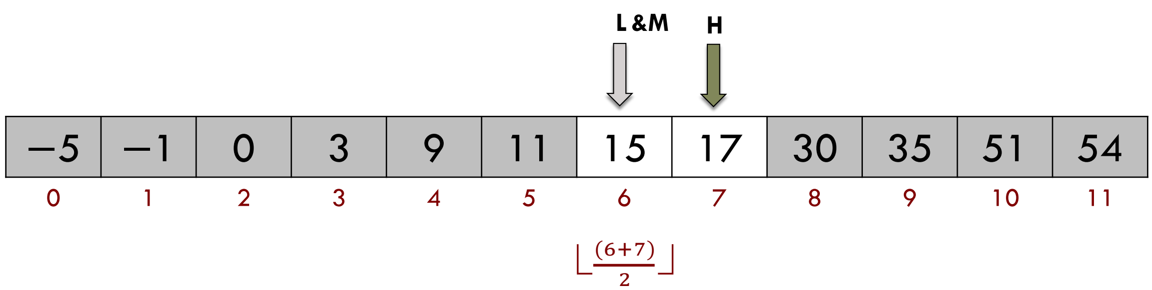
Searching for the value 15



Searching for the value 15



Searching for the value 15



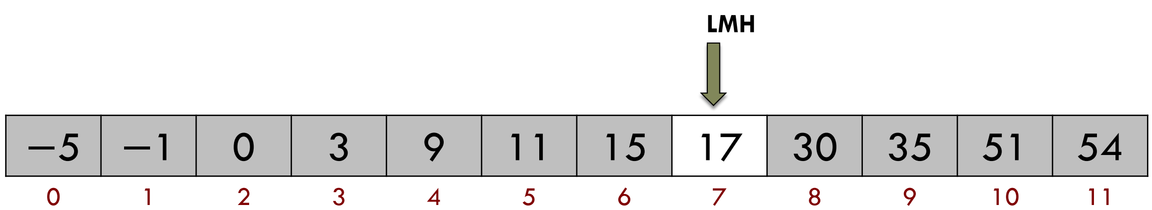
Absent value: Linear scan

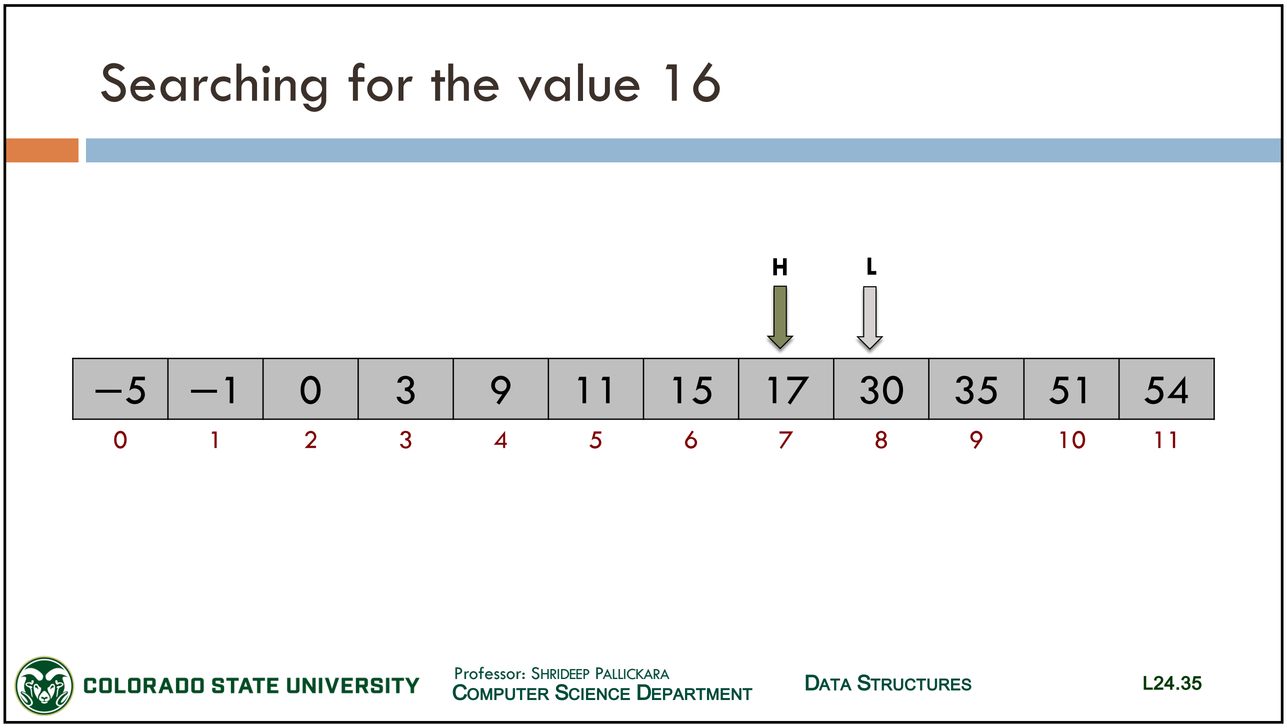
- In the linear scan case, we know that an element is not in the list?
  - As soon as we hit the end of the list

Absent value: Binary search

- We can **assert** our target item does not exist by **testing the bounds**
- As the search progresses?
  - The upper and lower bounds move closer and closer until there are no unexplored values between them
  - Since we are **always moving one of the bounds past the midpoint index**?
    - We can stop the search when IndexHigh < IndexLow
    - At that point, we can guarantee the target value is not in the list

Searching for the value 16




Searching for the value 16
H
L
−5
−1
0
3
9
11
15
17
30
35
51
54
0
1
2
3
4
5
6
7
8
9
10
11
COLORADO STATE UNIVERSITY
Professor: SHRIDEEP PALLICKARA
COMPUTER SCIENCE DEPARTMENT
DATA STRUCTURES
L24.35

35


DYNAMIC DATA STRUCTURES
This place is always such a mess
Sometimes I think I'd like to watch it burn
I'm so alone
Feel just like somebody else
Man, I ain't changed, but I know I ain't the same
One Headlight, Jakob Dylan, Wallflowers

36

Dynamic data structures

- **Alter their structure** as the data changes
- These **structural adaptations** may include
  - Growing the size of the data structure on demand
  - Creating dynamic, mutable linkings between different values, etc.
- Dynamic data structures are at the heart of almost every computer program in the world
  - Underpin some of the most exciting, interesting, and powerful algorithms in computer science

But arrays are so easy to work with … [1/2]

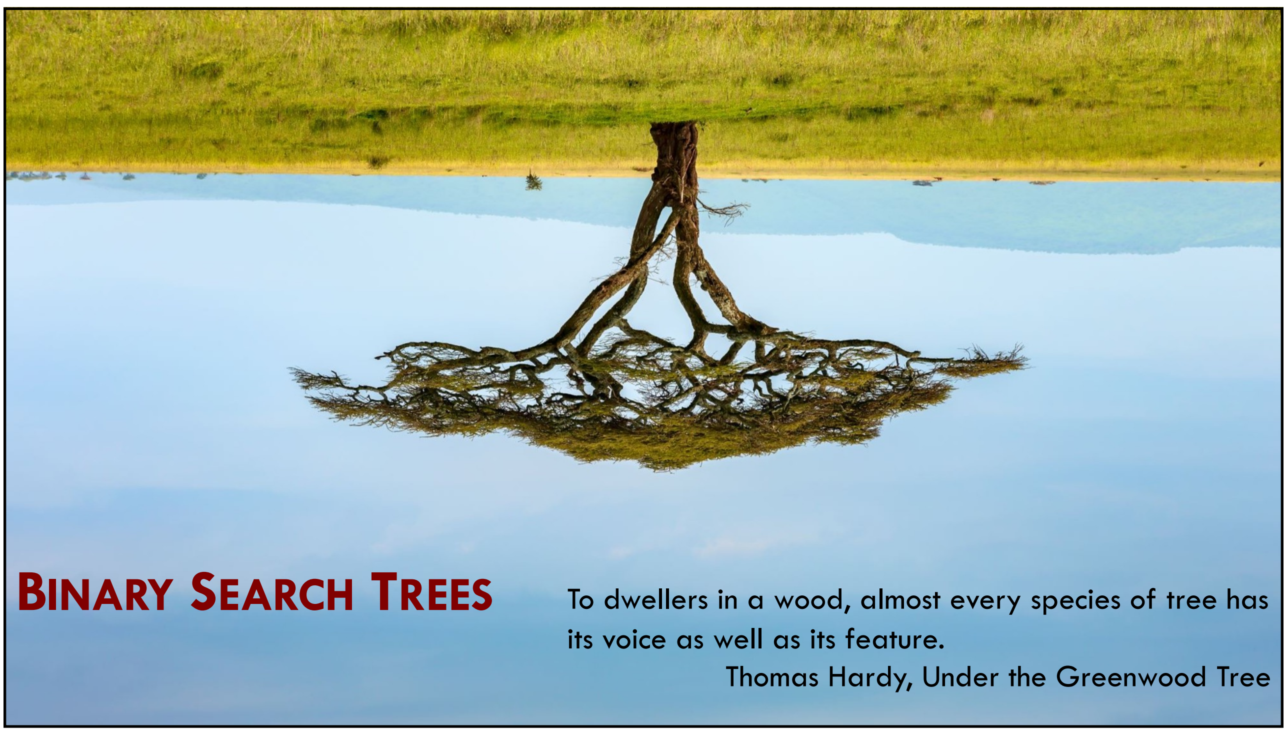
- Arrays are like parking lots
- They give us a place to store information, but don't provide much in the way of adaptation

But arrays are so easy to work with … [1/2]

- Sure, we can sort the values in an array (or cars in our parking lot) and use that structure to make binary search efficient
  - But we're just changing the ordering of the data within the array
  - The **data structure itself is neither changing nor responding to changes** in the data
- If we later change the data in a sorted array, say by modifying the value of an element?
  - We need to re-sort the array

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT DATA STRUCTURES L24.39

39



BINARY SEARCH TREES

To dwellers in a wood, almost every species of tree has its voice as well as its feature.
Thomas Hardy, Under the Greenwood Tree

40

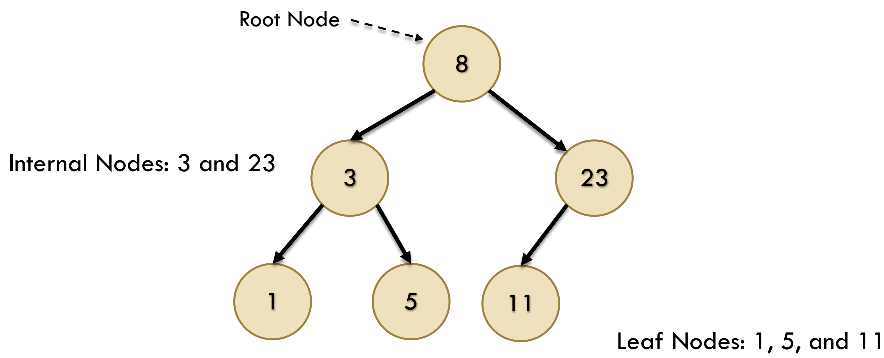
Binary Search Tree (BST)

- Binary search trees use the concepts underpinning the binary search algorithm to create a **dynamic data structure**
  - The key word here is dynamic
- Unlike sorted arrays, binary search trees support the efficient **addition and removal** of elements in addition to searches
  - Making them the perfect blend of the algorithmic efficiency of binary search and the adaptability of dynamic data structures

Tree Structures

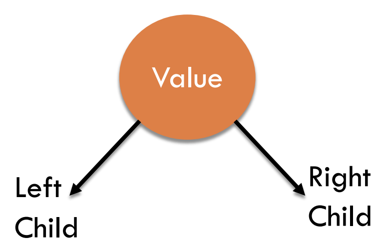
- Trees are **hierarchical** data structures
- Comprise **branching chains** of nodes
- Natural *extension* of linked lists
  - In the case of BSTs, each tree node is permitted two next pointers
    - Point to subsequent nodes in disjoint lists

Example BST



BST Nodes

- A node contains a **value** (of a given type)
  - Plus, **up to two** pointers to lower nodes in the tree
- Nodes with at least one child?
  - Internal nodes
- Nodes without any children?
  - Leaf nodes



Tree nodes may contain other information

- Often include a **pointer back to the node's parent**, for instance
  - Allows the **bottom-up traversals** of the tree
    - In addition to the typical top down
  - Comes in handy when we consider removing nodes

The TreeNode data structure

```
TreeNode {
    Type: value
    TreeNode: left
    TreeNode: right
    TreeNode: parent
}
```

We might also want to store auxiliary data …

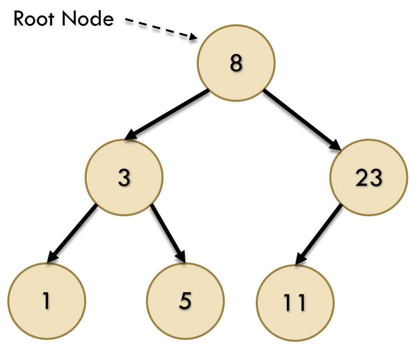
- Storing and searching for individual values are useful
- However, **using these values as keys** for looking up more detailed information greatly extends the power of the data structure
  - For e.g., coffee names as the node's values, allowing us to efficiently look up records for any coffee
    - Our auxiliary data would be a detailed record of that coffee

BST and auxiliary data

- The tree node data structure can either store this auxiliary data
  - Directly
  - Include a pointer to a composite data structure
    - Located somewhere else in memory

Binary search trees

- Start at a single root node at the top of the tree
- **Branch** into multiple paths as they descend
- Allows programs to access the binary search tree through a single pointer
  - The location of its root node



49

Nodes and dispersion in memory

- A search tree's individual nodes can be **scattered** throughout memory
- Each node is only linked to its children and parents through?
  - The power and flexibility of **pointers**

50

The power of the binary search tree stems from how values are organized within the tree

- For any node N
  - The value of *any node* in N's **left** subtree is less than N's value
    - Values in the left node and all nodes below it are less than the value of the current node
  - The value of *any node* in N's **right** subtree is greater than N's value
    - Values in the right node and all nodes below it are greater than the value of the current node
- The rule defines the tree's structure below that node
  - Partitions the subtree into two subsets

** Useful Rule: The number of right-handed people is greater than the number who are left-handed.

The contents of this slide-set are based on the following references

- Matthew Justice. *How Computers Really Work: A Hands-On Guide to the Inner Workings of the Machine*. ISBN-10/ISBN-13 : 1718500661/ 978-1718500662. No Starch Press. 2020. [Chapter 11]
- Jeremy Kubica. *Data Structures the Fun Way: An Amusing Adventure with Coffee-Filled Examples*. No Starch Press. ISBN-10. /13: 1718502605/978-1718502604. [Chapter 2, 3]