Ordered Structures and Bit Representations

Instructors: Sam McCauley and Dan Barowy

April 13, 2022

Admin

- Masks optional from today in class
- Keeping masks on in lab in the short term
- How was lab 6?
- Data visualization talk today at 7PM in TBL 211

Wrapping Up Ordered Structures

Beginning the `OrderedVector` class

```
public class OrderedVector<E extends Comparable<E>>
                  implements OrderedStructure<E> {

  protected Vector<E> data;

  public OrderedVector() {
    data = new Vector<E>();
  }

  public void add(E value) {
    int pos = locate(value);
    data.add(pos, value);
  }
```

Implementing `locate()`

- Finds an item in an `OrderedVector` using binary search
- We'll be using an iterative version of binary search (not recursive)
- Recall the invariant of binary search:
- If the item we're looking for is in the array, it is located somewhere within `low...high`

Locate

```
protected int locate(E target) {
   Comparable<E> midValue;
   int low = 0; // lowest location
   int high = data.size(); // highest location
   int mid = (low + high)/2; // low <= mid <= high
   while (low < high) {
      midValue = data.get(mid);
      if (midValue.compareTo(target) < 0) {
         low = mid + 1;
      }
      else {
         high = mid;
      }
      mid = (low+high)/2;
   }
   return low;
}
```

Filling in the rest of `OrderedVector`

- Now that we have `locate()` the rest is pretty easy!
- We already used `locate()` to fill in `add()`
- Let's use `locate()` to fill in `contains()` and `remove()`

Final OrderedVector Methods

```
public boolean contains(E value) {
   int pos = locate(value);
   return pos < size() && data.get(pos).equals(value);
}
public Object remove (E value) {
   if (contains(value)) {
      int pos = locate(value);
      return data.remove(pos);
   }
   else {
      return null;
   }
}
```

These can be found in the `structure5` `OrderedVector` class.

OrderedVector Performance

- Locate?
  - $O(\log n)$
- Add?
  - $O(n)$: locate is $O(\log n)$, but shifting items down is $O(n)$. So overall $O(n + \log n) = O(n)$.
- Contains?
  - $O(\log n)$ (just a call to locate and $O(1)$ extra work)
- Remove?
  - Like add: locate, and then remove (shifting items down as necessary); $O(n)$.

OrderedList

- Let’s talk through how to implement an ordered Linked List (say a `SinglyLinkedList`)
- How can we binary search in a singly linked list? What’s the challenge of doing so?
- Idea of binary search: we compare the item we are searching for to the middle element in the range `low...high` (using a call to `get()`)

Locating in a Linked List

- How long does finding `get(mid)` take in a linked list?
- $O(n)$ just to find *one* mid item
- We can show: $O(n)$ time for `locate()` in total
- Takeaway: ordering a linked list does not lead to faster search!
- The `OrderedList` class is still included in `structure5` however

A Note of Care About Ordered Structures

- This issue is common to all the structures we use that keep items in some order based on their contents

- No good way around it

- Problem: we need to assume that *every time* the objects change, their position in the `OrderedVector` is updated

- Let’s look at an example

Sorting Students by Grade

- We can easily change the `Student` class to allow comparison by age
- Then we can store students in an ordered list by age
- Let’s look at an example
- What happens when the age changes?
- Answer: `OrderedVector` doesn’t know the age changes, so doesn’t stay sorted

An Example that *Does* Work

- Let's store a list of associations between the population of a county and the percentage of people who voted third party in the 2020 election
- So we'd like an `OrderedVector<Association<Integer, Double>>`
- Wait a minute—the `OrderedVector` can only store things that implement `Comparable`. But `Association` doesn't implement `Comparable`
- The type of the key—`Integer`—does implement `Comparable`, however
- Enter: the `ComparableAssociation`. (Some of you may have used this in lab 5.)

ComparableAssociation summary

```
public class ComparableAssociation<K extends Comparable<K>,V>
           extends Association<K,V>
           implements Comparable<ComparableAssociation<K,V>>, Map.Entry<K,V>
{
    public int compareTo(ComparableAssociation<K,V> that)
    {
        return this.getKey().compareTo(that.getKey());
    }
}
```

(This is an example of a class that `implements` two different interfaces. We'll talk about `Map.Entry` in 3 or so weeks.)

Finishing Our Example

- We can store an `OrderedVector` of `ComparableAssociation<Integer,Double>`
- But, what happens when we change one of the `ComparableAssociations`?
- In particular, what happens when the *population* of one of the counties changes? (I.e. we change the key?)
- Answer: `ComparableAssociation` does not allow us to change the key!
- Takeaway: if you're storing a class type in an ordered data structure, control access so that the sorted order cannot change
    - If possible

Binary Representation

How are numbers stored in a computer?

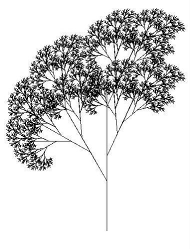
- Using binary!
- Set of `0`s and 1s (32 for `int`, 64 for `long`)
- Let’s see some examples

Bit operations

- Sometimes in computer science it's useful to operate on the bits of a number directly
- << is left shift: shift the bits left (they fall off if run out of room)
- >> is right shift: shift the bits right (they fall off if run out of room)
  - Be careful with negative numbers!!!
  - What are these equivalent to mathematically?
- `&`: take the *bitwise and* of the two numbers
  - Go bit by bit. If both bits are 1, resulting bit is 1. Otherwise it is `0`.
  - Example on board
  - Why would we use this?

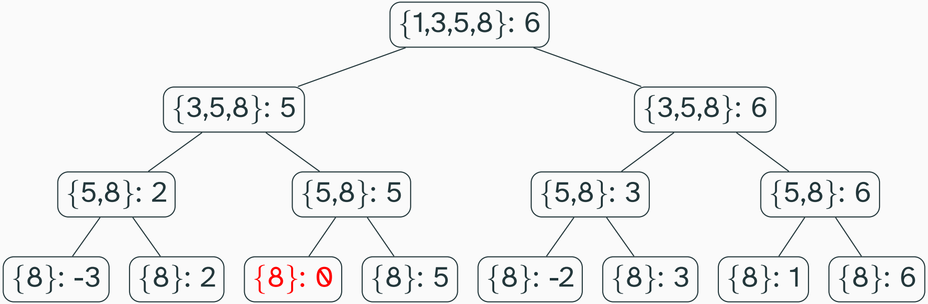
Trees

Trees



- All the ways we've had to store data has been one-dimensional.
    - At the end of the day: every item in our data structure is the *i*th item in the data structure for some *i*
    - All of our access has (indirectly) been through such a one-dimensional mapping
- With trees, we add a second dimension to how we store data
- *Drastic* improvements in what we can store and the performance we can achieve

Trees We've Seen



We can draw the method calls made by a recursive algorithm using a tree! (The above is `canMakeSum()` from lab 3.)

Here: each of the rectangles above (called a *node*) represents a recursive call. We connect each method to the methods it calls.

Trees We've Seen

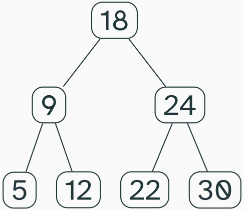
5	9	12	18	22	24	30

Calling back to last lecture: what happens when we do binary search on this array?

Something like: first, we compare our query element to 18. Based on the result, we then compare it to either 9 or 24.

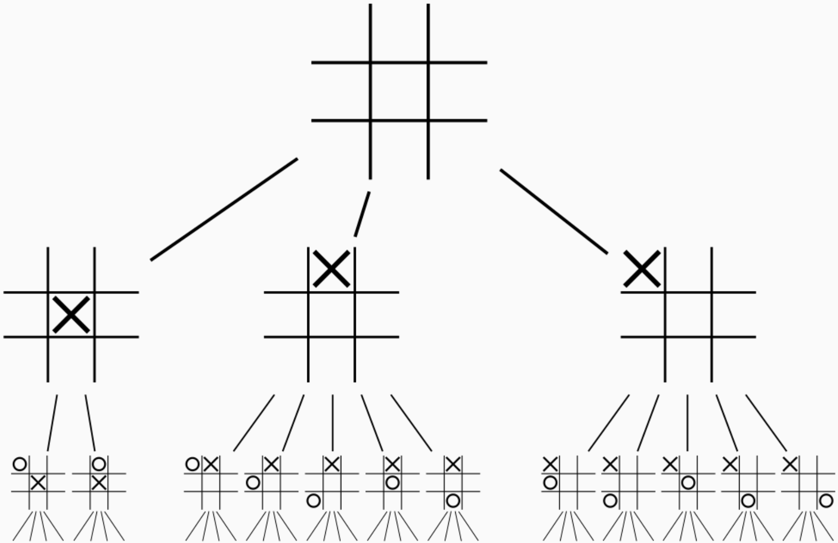
Trees We’ve Seen

5	9	12	18	22	24	30

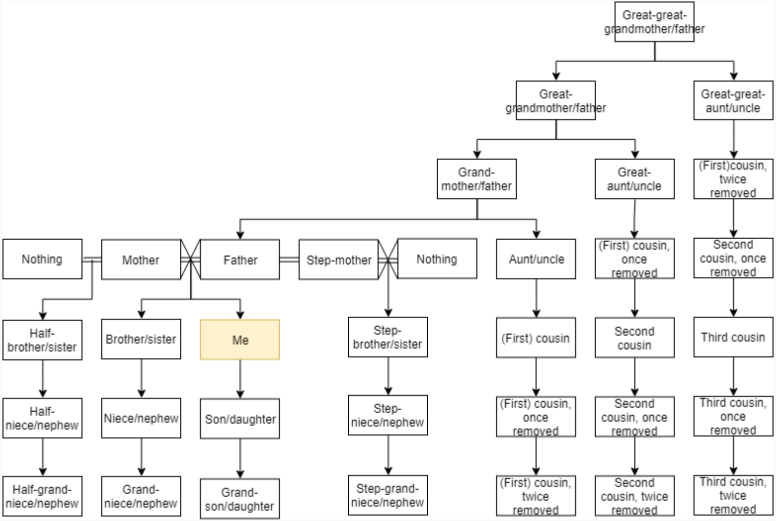


Binary search seems to also have a tree-like structure. We’ll see how to store data in a very similar tree very soon.

Game Tree



Family "Tree"

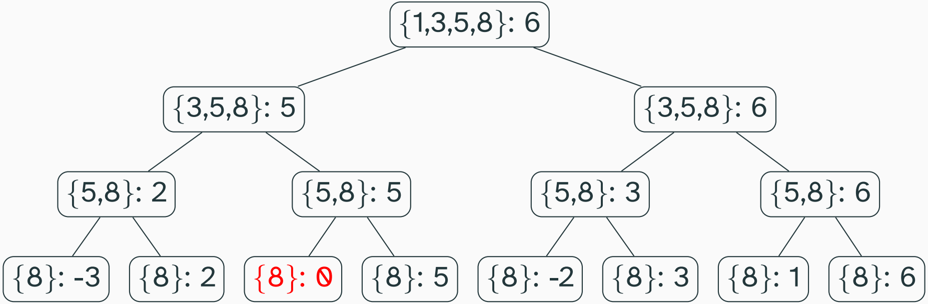


Same basic idea. Though note: not quite a tree by our definition.

Definition of a Tree

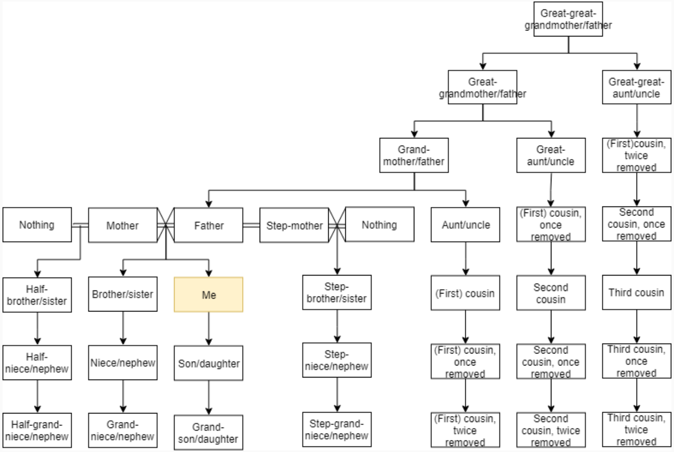
- Tree consists of nodes (the boxes in the images we saw above)
- Nodes are connected by edges (lines in the images we saw above)
- There is one root node that does not have a *parent node*
- Every other node has exactly one *parent node*
- Nodes may have some children.
- A node without a child is called a *leaf*

Labelling nodes



What is the root node in this tree? What are the leaves?

Family “Tree”



Why isn’t this a tree?

- Answer: nodes have multiple parents! (Plus there are a bunch of extra edges in this image.)

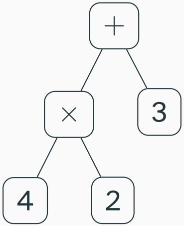
Binary Tree

Binary Tree

- Binary Tree: A tree where each node has at most 2 children
- The degree of a node is the number of children it has. So a binary tree is a tree where all nodes have degree at most 2.
- Let's see an example of a binary tree. Then, we'll discuss the `BinaryTree` class that comes with `structure5`

Expression Tree

$$\boxed{4 \times 2 + 3}$$



We can write arithmetic expressions using a binary tree. (Why is it binary?)

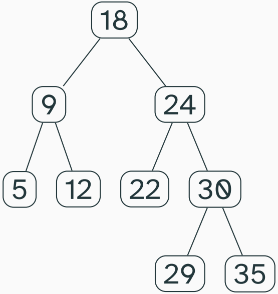
Using a Binary Tree

- *Goal*: store an expression using a binary tree
- Then write some code to evaluate the expression
- Takeaway: practice with binary trees

How to Store a Binary Tree?

- Nodes should probably be objects of some class type.
- Store its children
- In the `SinglyLinkedList`, we had a hidden `Node` class; the `SinglyLinkedList` itself only stored a pointer to the `head`
- `BinaryTree<E>` does not work that way! Just a single recursive class

Visualizing Trees Recursively



Each node in a (binary) tree can be viewed as the root of its own (binary) tree.

BinaryTree plan

- Each node is stored as a `BinaryTree` object
- Stores the value stored at the node
- Stores the parent (of type `BinaryTree`)
  - The root of the tree stores `null` for its parent
- Stores the left and right children (both are of type `BinaryTree`
  - If either doesn't exist, points to an *empty node*
  - Children of an empty node *point to the node itself*
  - There are other ways to implement missing children in a binary tree; this is just one
- Let's take a look at the code for `BinaryTree`
- Now, let's look at how we can evaluate a tree of expressions stored in a `BinaryTree`