Cosc 201
Algorithms and Data Structures
Lecture 10 (26/3/2025)
Maps, sets, trees and ordered sets

Brendan McCane
brendan.mccane@otago.ac.nz
And
Michael Albert

Sets

- A data type that we have been carefully skirting around discussing is the *set*.
- A set is just a collection of items, with no repetition allowed.
- The operations we want to support are:
  - *Add*(x): adds an element if it's not already present,
  - *Remove*(x): removes an element if it's present, and
  - *Contains*(x): determines if the set contains an element.
  - Some method of iterating (e.g., for-each loop) over the elements.
- Java defines the Set interface (along with some implementations of it).
- Two important data structures for sets are *hash maps* and *binary search trees*.
- Hash maps are useful when the set is *unordered* and binary search trees are useful when the set is *ordered*.
- We'll look at both in the coming lectures.

Maps

- In computer science, a *map* consists of a set of *keys*, each of which has an associated *value*
- So sets are a prerequisite!
- The fundamental map operations are usually considered to be:
  - *put*(k, v): add the mapping from k to v either by adding k if it's not already present, or by changing the associated value,
  - *get*(k): return the value associated to k if k is present, and
  - *remove*(k): remove the key k.
- The Java Map interface specifies a lot more! So many in fact that they include an AbstractMap class that

  . . . provides a skeletal implementation of the Map interface, to minimize the effort required to implement this interface.

Basic set

- The simplest implementation of a set that I can think of would use an array (or list) as storage.
- To add an element, iterate over the array to see if it's present and if not add it at the end
- To remove an element, iterate over the array to see if it's present and if it's found, overwrite it with the last element.
- To check if an element is present . . . (we used that above).
- The time-complexity of all three operations is $O(n)$ where n is the current size of the set.
- We'd like to do much better.

Ordered set

- In an ordered set, there is some underlying “natural” order on its elements (e.g., dictionary order for `String` objects)
- In that context we’d also like to be able to *traverse* the set in order from a given element.
- E.g., “list the next 100 words in the dictionary after *cat*”
- If the set is **static** (i.e., unchanging) this is easy:
  - Store the elements in a sorted array
  - Use binary search to find elements
  - Traversal is just incrementing a counter
- So what are the drawbacks of using an array for the ordered set data type?

Ordered set (array version)

- Storage for n elements is in a sorted array of size at least n.
- Search is $O(\log n)$ using binary search.
- Traversal is $O(\log n)$ to initialise (it amounts to a search) and then constant time per item after that (just a counter increment).
- The problem is the dynamic operations:
  - *Add*(x): We need to insert x if it's not already present, so we start with a search which is fine, but then insertion is $O(n)$.
  - *Remove*(x): Similarly, we need to find x if present, and then move everything beyond it back over top, so also $O(n)$.
- If we anticipate relatively low use of the dynamic operations we might be happy with this.
- Another approach might be to maintain a *main* array and subsidiary *add* and *remove* arrays and only periodically do the updates to the main array – but this complicates things considerably!

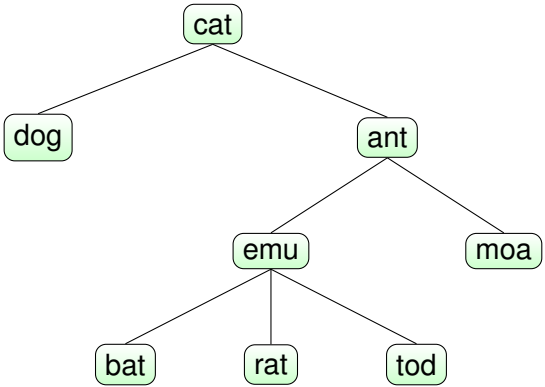
Among the trees

- A variety of data structures called *trees* are fundamental in computer science.
- We've already seen one in our "chains of representatives" implementations of union-find and in our implementation of heaps.
- The type that's important for an ordered set data type is called the *binary search tree*.
- But first some generalities.

Trees in general

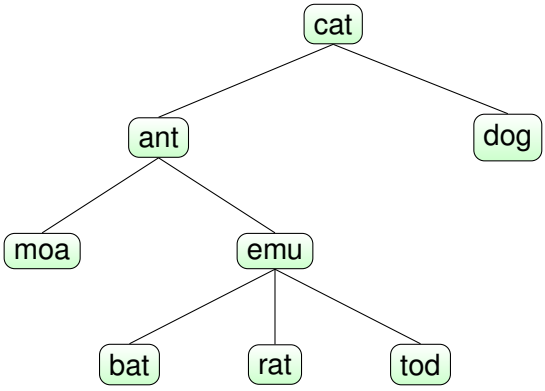
- A tree consists of *nodes*.
- One node is distinguished and called the *root*.
- Each node, except the root, has a unique *parent*.
- Any chain that moves from a node to its parent, its grandparent, etc. eventually winds up at the root.
- The *children* of a node are all the nodes of which it is the parent.
- Nodes may (and usually do) have additional data associated to them, but this is not relevant to the structure of the tree.

1K words (and a few more)



- *cat* is the root (in CS, trees are drawn with the root at the top.)
- The parent of *dog* is *cat*, of *rat* is *emu*.
- Some nodes have two children, one has three (*emu*) and some have none.
- Nodes with no children are called *leaves*.

Is this tree different?



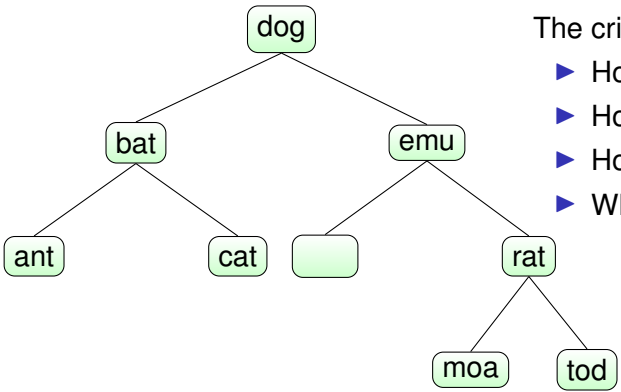
- Sometimes!
- We need to specify whether the order of the children of a node matters.
- Trees in which the order matters (fairly much the normal situation in CS) are sometimes called *plane trees*.
- You just need to check on a case by case basis.
- Sometimes (e.g., in the *binary search trees* we're about to see), there are even fixed slots for the children.

Binary search trees

In a *binary search tree*:

- The node data contains (or is) 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 key values at a node's left child *and all its descendants* must be less than the key of the node.
- The key value at a node's right child *and all its descendants* must be greater than the key of the node.
- We do not allow duplicate keys – if you wish to allow duplicate keys add "or equal to" in *one* case above.

Another 1K words



The critical questions:

- How do we search?
- How do we add?
- How do we traverse?
- What do the operations cost?

We'll come back to these questions in a few lectures.