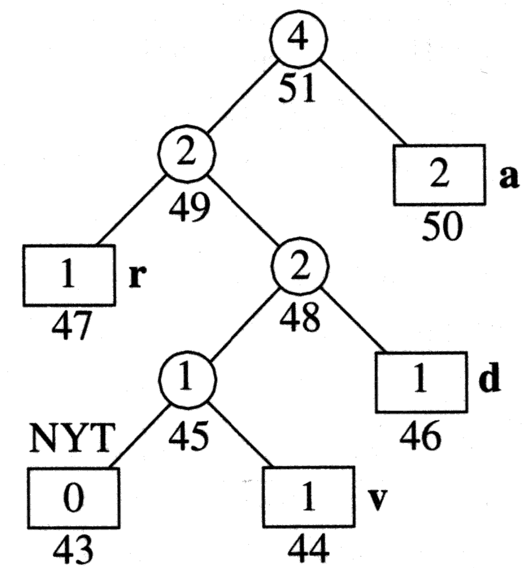
Trees



Carlos Moreno
cmoreno @ uwaterloo.ca
EIT-4103

https://ece.uwaterloo.ca/~cmoreno/ece250

These slides, the course material, and course web site are based on work by Douglas W. Harder

Trees

- Today's class:
  - We'll discuss one possible implementation for trees (the general type of trees)
  - We'll look at tree traversal — strategies to visit every element in the tree following a given sequence
    - Breadth-first traversal
    - Depth-first traversal — two types (for now), depending on whether we deal first with “self” node or with the (strict) descendants.

Trees

- We recall from several lectures ago, some examples of typical operations on hierarchically ordered data (determine precedence between elements, nearest common predecessor, depth).

Trees

- We recall from several lectures ago, some examples of typical operations on hierarchically ordered data (determine precedence between elements, nearest common predecessor, depth).
- To implement those, and others, it seems clear that we need a data structure that follows quite closely our “visual” idea of a tree.

Trees

- For example, if I simply give you an unordered list of IDs for elements, and then an unordered list of immediate predecessor pairs, would you be able to perform those operations?

Trees

- For example, if I simply give you an unordered list of IDs for elements, and then an unordered list of immediate predecessor pairs, would you be able to perform those operations?
- Let's see ...

Trees

- Example:
  - Elements are: T, H, D, G, E, P, A, S
  - Associations are (denoted as pair {parent,child}):
    - {T, E}, {P, D}, {G, H}, {P, S}, {T, A}, {P, T}, {G, P}

Trees

- Example:
  - Elements are: T, H, D, G, E, P, A, S
  - Associations are (denoted as pair {parent,child}):
    - {T, E}, {P, D}, {G, H}, {P, S}, {T, A}, {P, T}, {G, P}
  - Question (more or less easy): What is the root of that tree?

Trees

- Example:
  - Elements are: T, H, D, G, E, P, A, S
  - Associations are (denoted as pair {parent,child}):
    - {T, E}, {P, D}, {G, H}, {P, S}, {T, A}, {P, T}, {G, P}
  - Question (more or less easy): What is the root of that tree?
    - Follow-up question: what was the run time that takes for you to find the answer?

Trees

- Example:
  - Elements are: T, H, D, G, E, P, A, S
  - Associations are (denoted as pair {parent,child}):
    - {T, E}, {P, D}, {G, H}, {P, S}, {T, A}, {P, T}, {G, P}
  - Question (a little tougher): What is nearest common predecessor of E and S?

Trees

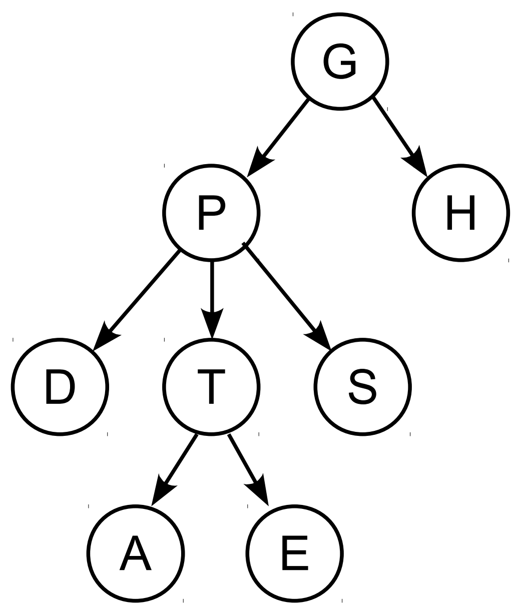
- Example:
  - Elements are: T, H, D, G, E, P, A, S
  - Associations are (denoted as pair {parent,child}):
    - {T, E}, {P, D}, {G, H}, {P, S}, {T, A}, {P, T}, {G, P}
  - Question (a little tougher): What is nearest common predecessor of E and S?
    - So, we won't even try this one !!! (and I bet if some of you got the answer in less than 45 to 60 seconds it is because you drew the corresponding tree !!)

Trees

- Example:
  - Elements are: T, H, D, G, E, P, A, S
  - Associations are (denoted as pair {parent,child}):
    - {T, E}, {P, D}, {G, H}, {P, S}, {T, A}, {P, T}, {G, P}
  - Question (a little tougher): What is nearest common predecessor of E and S?
    - Of course, if I actually show you the corresponding tree, then very different story ...

Trees

- Example:
  - Elements are: T, H, D, G, E, P, A, S
  - Associations are (denoted as pair {parent,child}):
    - {T, E}, {P, D}, {G, H}, {P, S}, {T, A}, {P, T}, {G, P}



Trees

- So, it makes sense that our implementation of a tree would provide data that makes it possible to “follow the associations”
- Exactly as they graphically appear on a diagram, allowing us (our brains) to easily follow the associations.

Trees

- A typical implementation could be similar to what you did (hopefully you did it... right?) in lab 0 for a linked list:

Trees

- A typical implementation could be similar to what you did (hopefully you did it... right?) in lab 0 for a linked list:
  - A class **`Node`** to represent nodes.
  - A class **`Tree`** that will own a collection of **`Node`** objects, and will have, among others, a pointer to the root node.

Trees

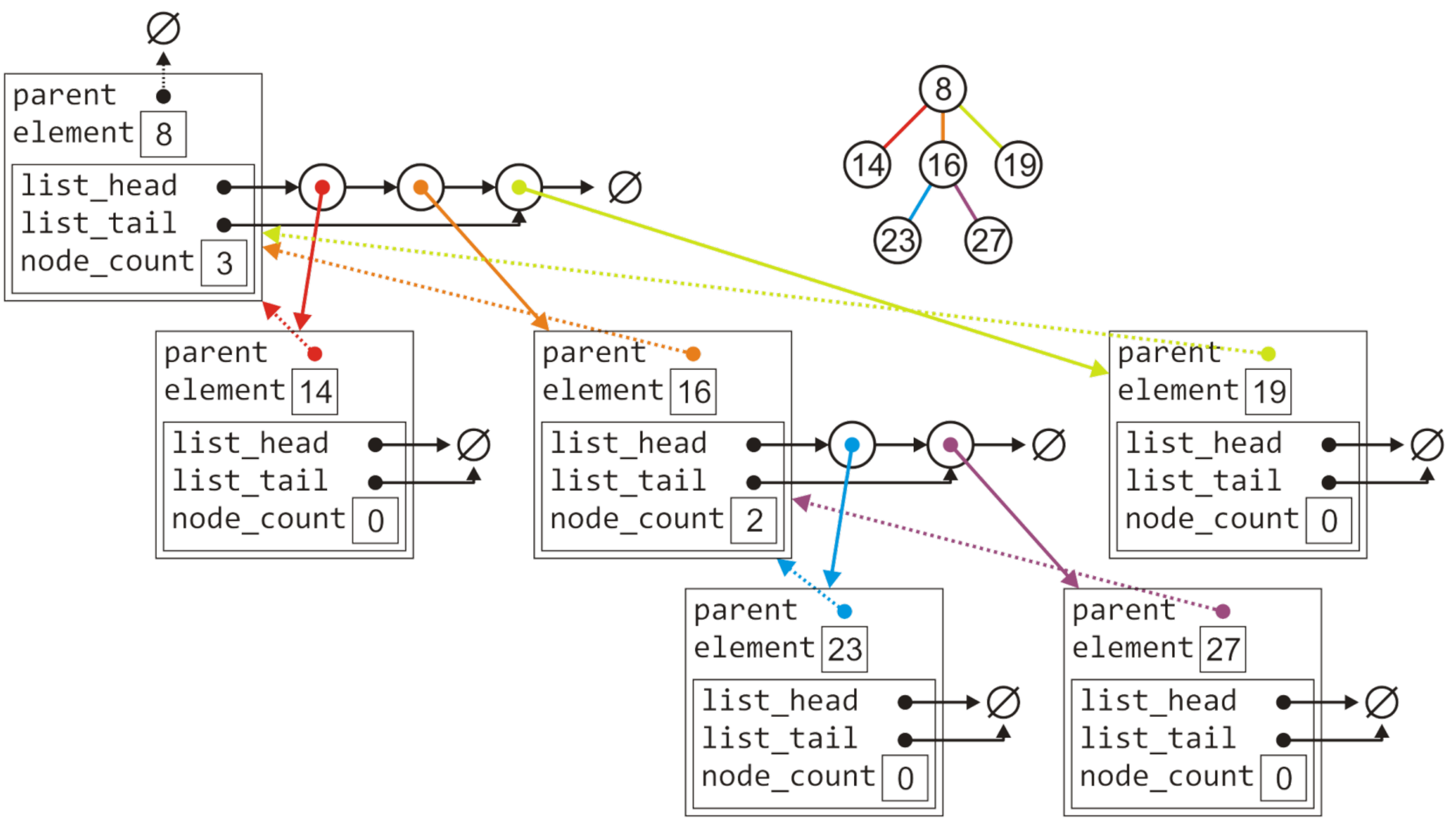
- A typical implementation could be similar to what you did (hopefully you did it... right?) in lab 0 for a linked list:
  - A class `Node` to represent nodes.
  - A class `Tree` that will own a collection of `Node` objects, and will have, among others, a pointer to the root node.
  - As we mentioned last time, by analogy with a linked list, the tree class only needs to know about the root node, and then nodes will point to their children.

Trees

```
template <typename Type>
class Node
{
    Type d_element;
    Node<Type> * d_parent;
    std::list<Node<T> *>  d_children;
      // or std::vector – could store them as an array

public:
    Node (const Type & obj, Node<Type> * parent);

    Type retrieve() const;
    Node<Type> * parent() const;
    int degree() const;
    bool is_root() const;
    bool is_leaf() const;
    Node<Type> * child (int n) const;
    int height() const;

    void insert (const Type & obj);
};
```

Trees

- This tree, with six nodes, would be stored as follows:



Trees

- The implementations are, for the most part, straightforward:

```
template <typename Type>
bool Node<Type>::is_root() const
{
    return parent() == NULL;
}
```

Trees

- The implementations are, for the most part, straightforward:

```
template <typename Type>
int Node<Type>::degree() const
{
    return d_children.size();
}

template <typename Type>
bool Node<Type>::is_leaf() const
{
    return degree() == 0;
}
```

Trees

- The implementations are, for the most part, straightforward:

```
template <typename Type>
int Node<Type>::insert (const Type & obj)
{
   Node<Type> * subtree = new Node<Type>(obj, this);
   d_children.push_back (subtree);
}
```

Trees

- The implementations are, for the most part, straightforward:

```
template <typename Type>
int Node<Type>::insert (const Type & obj)
{
   Node<Type> * subtree = new Node<Type>(obj, this);
   d_children.push_back (subtree);
}
```

- Why `this` as parameter to the constructor?

Trees

- Suppose we wanted to compute the size (number of nodes) in the tree (say that we added a method Node<Type>::size() const:

Trees

- Suppose we wanted to compute the size (number of nodes) in the tree (say that we added a method Node<Type>::size() const:
- Hopefully you recall the nice recursive definition for a tree... So, that would seem to suggest that size() could perfectly be a recursive function! :

Trees

```
template <typename Type>
int Node<Type>::size() const
{
    int count = 1;  // counting this one

    for (list<Node<Type> *>::iterator ch = d_children.begin();
                                      ch != d_children.end();
                                      ++ch)
    {
        count += c->size();
    }

    return count;
}
```

Trees

- Side note: Iterators in the STL (I'm using the STL's linked list class template, std::list) are designed to have a syntax similar to pointers.
- They take advantage of operator overloading, which allows us to define functions and methods that will be invoked when we use a certain operator with an object of that class.

Trees

- So, the loop

```
for (list<Node<Type> *>::iterator ch = d_children.begin();
                                  ch != d_children.end();
                                  ++ch)
```

is nothing more than a "disguised" version of:

```
for (list_iterator ch = d_children.begin();
                   !ch.at_end();
                   ch.advance())
```

Trees

- So, the loop

```
for (list<Node<Type> *>::iterator ch = d_children.begin();
                                  ch != d_children.end();
                                  ++ch)
```

is nothing more than a “disguised” version of:

```
for (list_iterator ch = d_children.begin();
                   !ch.at_end();
                   ch.advance())
```

- The method advance(), for instance, is called operator++() — and it's invoked when using ++

Trees

- The at_end() issue is a little bit trickier — since iterators are like pointers, class list also returns an iterator pointing to one-past-end, so that with the help from overloaded operators == and != (functions `operator==()` and `operator!=()`), we can check if we are already outside the range.

Trees

- Closing the parenthesis ... Say now that we want to obtain the height of the tree.

Trees

- Closing the parenthesis ... Say now that we want to obtain the height of the tree.
- Recursion again ... Right?

Trees

- And since our next topic is tree traversals... How are these recursive functions traversing the tree? (i.e., in what order are they visiting the nodes of the tree?)

Trees

- And since our next topic is tree traversals... How are these recursive functions traversing the tree? (i.e., in what order are they visiting the nodes of the tree?)
- We'll answer this on the board (and no, the answer is not in the next slides, so if you feel like taking notes, by all means do ...)

Trees

- Recursive implementations typically lead to a *depth-first* traversal:
  - We first go as deep as possible below each node before visiting any sibling node.

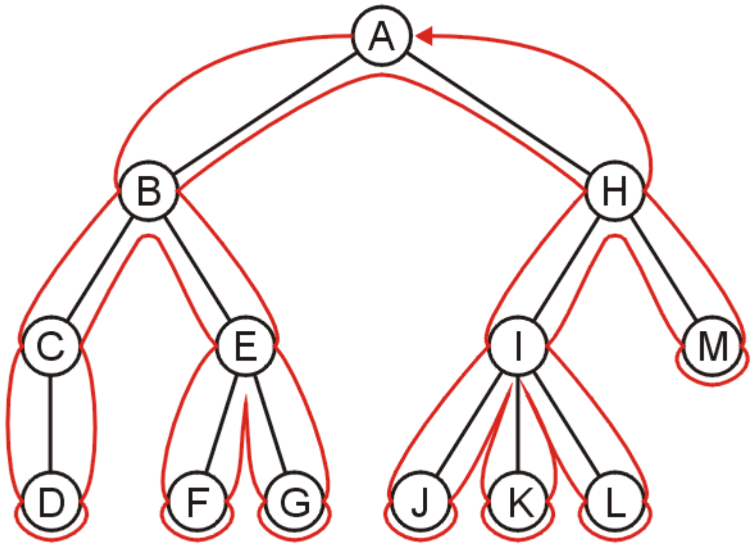
Trees

- Recursive implementations typically lead to a *depth-first* traversal:
  - We first go as deep as possible below each node before visiting any sibling node.

- The other typical traversal is *breadth-first* — all siblings are visited first, before moving to the next depth.

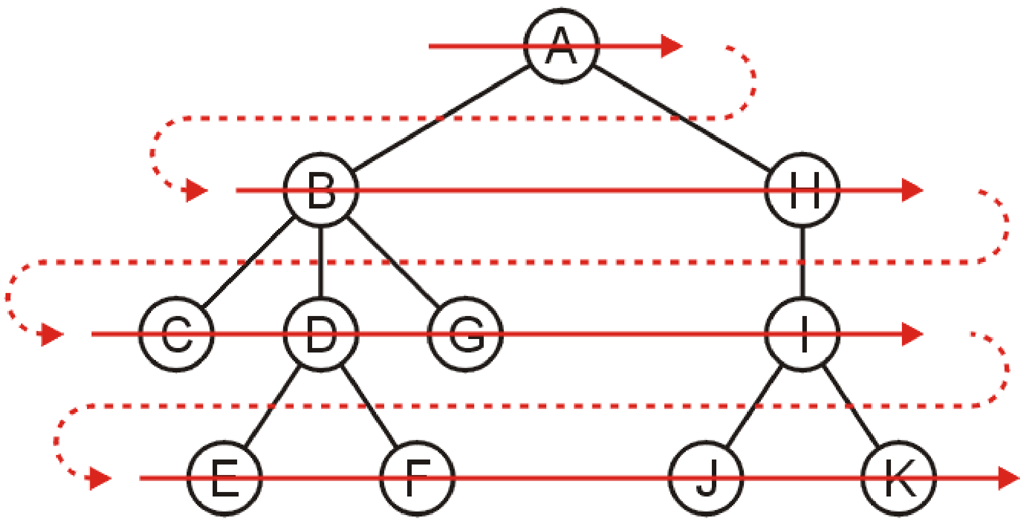
Trees

- Graphically, depth-first traversal goes like this:



Trees

- Breadth-first traversal:



Trees

- Breadth-first traversal can be implemented with a queue (a FIFO data structure)

Trees

- Breadth-first traversal can be implemented with a queue (a FIFO data structure)
- Depth-first requires a stack (LIFO)

Trees

- Breadth-first traversal can be implemented with a queue (a FIFO data structure)
- Depth-first requires a stack (LIFO)
  - Huh?? Didn't we already see some depth-first examples? We didn't use a stack

Trees

- Breadth-first traversal can be implemented with a queue (a FIFO data structure)
- Depth-first requires a stack (LIFO)
  - Huh?? Didn't we already see some depth-first examples? We didn't use a stack — did we??

Trees

- Let's look at breadth-first traversal:
  - We'll use a queue for the nodes (could be a queue holding pointers to the nodes). The procedure is:

Trees

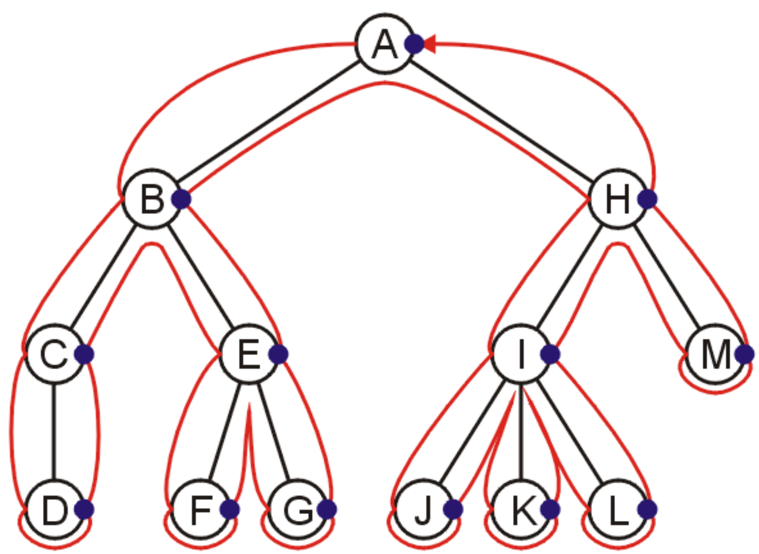
- Let's look at breadth-first traversal:
  - We'll use a queue for the nodes (could be a queue holding pointers to the nodes). The procedure is:
    - Enqueue the root node.
    - While the queue is not empty:
      - Dequeue an element
      - Enqueue all of the children of the just dequeued node

Trees

- Let's look at breadth-first traversal:
  - We'll use a queue for the nodes (could be a queue holding pointers to the nodes). The procedure is:
    - Enqueue the root node.
    - While the queue is not empty:
      - Dequeue an element
      - Enqueue all of the children of the just dequeued node
    - Neat, huh?

Trees

- An important notion with depth-first traversals comes from the observation that each node is visited more than once:
  - When in our way down to visit the children nodes, and when we get back from each of the children.



Trees

- An important notion with depth-first traversals comes from the observation that each node is visited more than once:
  - When in our way down to visit the children nodes, and when we get back from each of the children.
- What if we require some processing for the present node? When would we do it? On our way down, or on our way back?

Trees

- This leads to the distinction between pre-order and post-order depth-first traversals:
  - Pre-order means that we first process the current node, then move to the children.

Trees

- This leads to the distinction between pre-order and post-order depth-first traversals:
  - Pre-order means that we first process the current node, then move to the children.
  - Post-order means that we first process the children, and *then* we process the current one.

Trees

- This leads to the distinction between pre-order and post-order depth-first traversals:
  - Pre-order means that we first process the current node, then move to the children.
  - Post-order means that we first process the children, and *then* we process the current one.
    - Later on (a few lectures from now), we'll see *in-order* traversal — a notion that is only applicable for certain types of trees.

Trees

- So, which one should we use? Or, if both are necessary, when do we use each?

Trees

- What about the HTML rendering example?
- Breadth-first, or depth-first?

Trees

- What about the HTML rendering example?
- Breadth-first, or depth-first?
  - Clearly depth-first: we need to know about how all nested tags (the children/descendants) affect the geometry of the containing tag.

Trees

- What about the HTML rendering example?
- Breadth-first, or depth-first?
  - Clearly depth-first: we need to know about how all nested tags (the children/descendants) affect the geometry of the containing tag.
  - So, pre-order or post-order?

Summary

- During today's class:
  - We continued with topics on Trees
  - Looked at some implementation approaches
  - Investigated traversal strategies:
    - Breadth-first (visit all siblings before descending)
    - Depth-first (go as deep as possible before moving to the next sibling)
      - Pre-order traversal (process current node, then children)
      - Post-order traversal (process all children, then current node)