Trees

Overview

- ***Trees*** → subject of our first study of a ***nonlinear*** structure
  - All data structures we've seen so far have a ***linear*** structure in which there is a 1st entry, 2nd entry, 3rd entry,…, and last entry
    - *Array*: 1st element, 2nd element, 3rd element,…, and last element
    - *Linked-list*: 1st node, 2nd node, 3rd node,…, and last node
- In a nonlinear structure, the ***components do not form a simple sequence***
  - The linking between components is more complex
- (incidentally) ***Graphs*** → another nonlinear data structure
  - Tree ↔ *hierarchy*
  - Graph ↔ *network*

What's the basic difference?

Trees

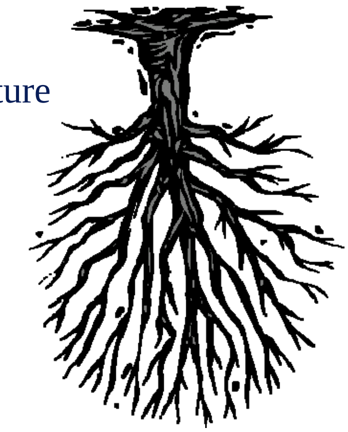
Overview

- A *real* tree starts at the root and branches as it grows
- At the ends of the branches are leaves
- The data structure that we are studying is called a tree because it shares some of these properties

Trees

Overview

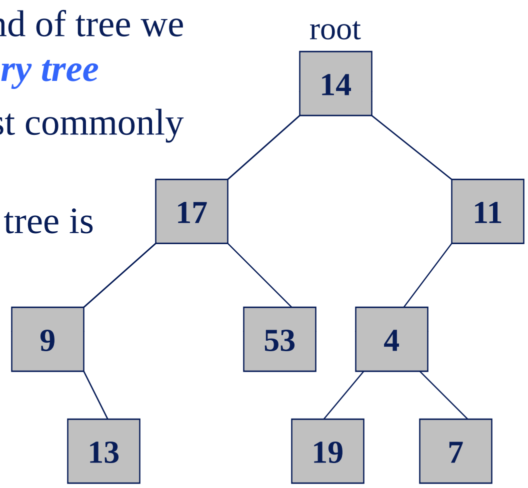
- A computer scientist’s tree is viewed “upside down”
- The root is at the top of the structure
- As you move down the tree you encounter branches and leaves



Trees

Binary Tree as Example

- The first (and only?) kind of tree we will study is called ***binary tree***
- Binary trees are the most commonly used tree data structure
- An example of a binary tree is shown on the right
  - binary tree of integers



Trees

- A tree is made up of ***nodes*** (shown as "boxes")
  - Just like a node of a linked-list, each node of a tree contains some *data* (plus *linking information*)
- The *node at the top of the tree* is called the ***root***
- Each node in a *binary tree* can have *up to two nodes* below it, referred to as the node's ***children***

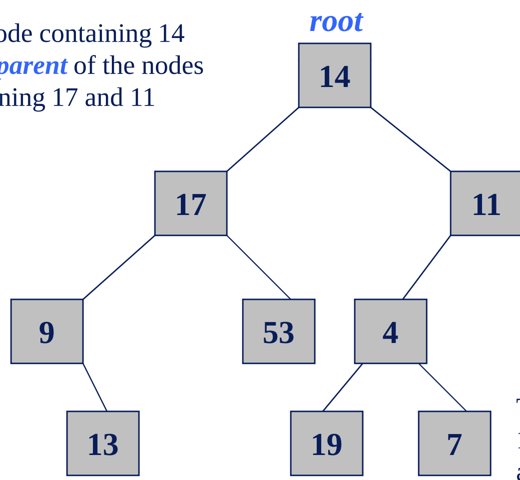
Trees

- In a binary tree, the child linked to a node's left is called the ***left child*** and the node linked to it's right is called the ***right child***
- A node with *no children* is called a ***leaf***
- Except for the root, each node has *exactly one parent* (the root has no parent)
- The ***parent*** of a node is the node linked *above it*

Trees

Terminology applied to our Example

The node containing 14
is the ***parent*** of the nodes
containing 17 and 11



The nodes containing
13, 53, 19, and 7
are ***leaves***

Binary Trees

Definition

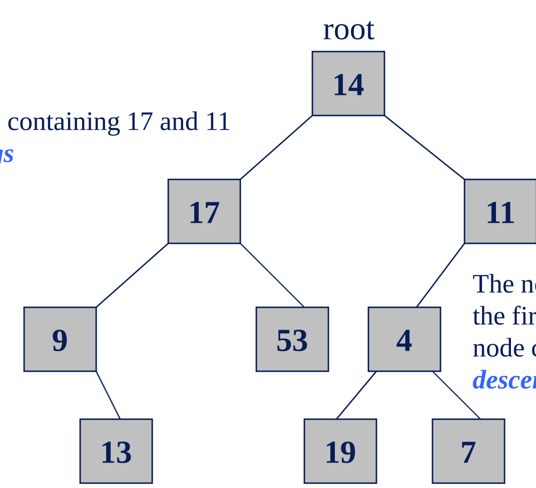
- A ***binary tree*** is a finite set of ***nodes***
- If the set is empty, we have a "trivial" tree with no nodes called the ***empty tree***
- If the set is not empty, then following rules apply:
  - There is one special node, called the ***root***
  - Each node may be associated with ***up to two*** other different nodes, called its ***left child*** and its ***right child***
  - Each node, except the root, has ***exactly one parent***
  - If you start at a node and move to the node's parent, then move again to that node's parent, and keep doing this, eventually you will reach the root → all nodes are reachable through the root

Trees — More Terminology

- Two nodes are ***siblings*** if they have the *same parent*
- A node’s parent is its first ***ancestor***, the parent of the node's parent is its next ancestor…
- A node’s children are its first ***descendants***, the node's children’s children are its next descendants…

Trees — Terminology applied to our Example

The nodes containing 17 and 11 are ***siblings***

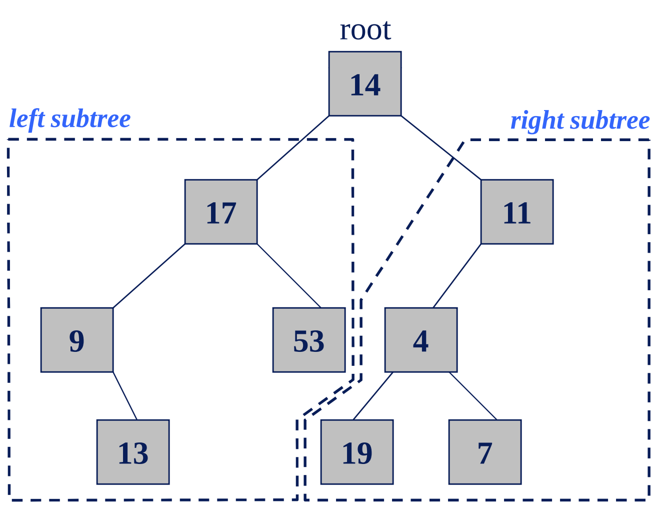


The node containing 11 is the first ***ancestor*** of 4, the node containing 7 is the first ***descendant*** of 4

Trees — More Terminology

- For a node in a binary tree, the nodes beginning with its *left child and below* are its ***left subtree***. The nodes beginning with its *right child and below* are its ***right subtree***
  - You may begin to see now why recursion and trees usually work together → a tree is "recursively" defined in terms of a node and smaller trees

Trees — Terminology applied to our Example



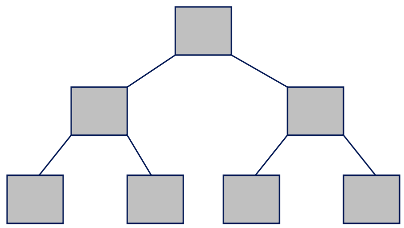
- Depth of a node
  - The ***depth of a node*** is the *number of steps required to move from the node to the root*
    - If we are at a node ***n*** and start moving up and toward the root, each time we moved up a level (to the node's parent), we are said to have moved one step (up the tree).
    - The depth of root is 0
    - (Some authors define depth a little differently → with the depth of root being 1 instead of 0)
    - (Some authors use ***level*** instead of depth when referring to the depth of a node)

- Depth of a tree
  - The ***depth of a tree*** is the *maximum* depth of any of its leaves
  - If a tree has only one node, the root, then its depth is 0
  - The empty tree doesn't have any leaves at all, so its depth is -1 by definition
  - (Some authors use ***height*** instead of depth when referring to the depth of a tree)

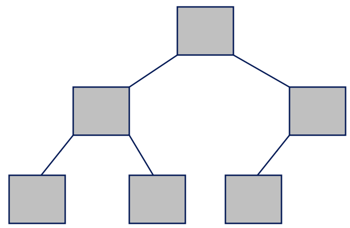
Binary Trees

Full Binary Tree

- In a ***full binary tree***, every leaf has the same depth, and every non-leaf has two children



full

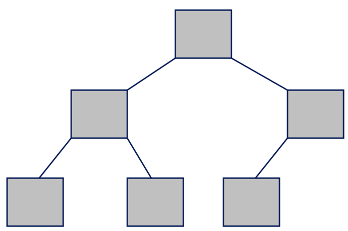


not full

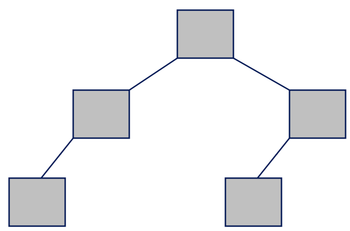
Binary Trees

Complete Binary Tree

- In a ***complete binary tree***, *all levels are full except possibly the last level* and, if the last level is not full, then all nodes of the last level are *as far to the left as possible*



complete



not complete

General Trees

- A binary tree gets its name from the fact that each node may have *at most two children*
- In a ***general tree***, a *node can have any number of children*
- Binary trees have special applications in computer science, but other types of trees are useful as well

General Trees Definition

- A tree (in general) is a finite set of nodes
- If the set is empty, the tree is the empty tree
- If the set is not empty, then following rules apply:
  - There is one special node called the root
  - Each node may be associated with ***zero or more*** different nodes, called its children
  - Except for the root, each node has exactly one parent
  - If you start at any node and repeatedly move toward the parent, you will eventually reach the root

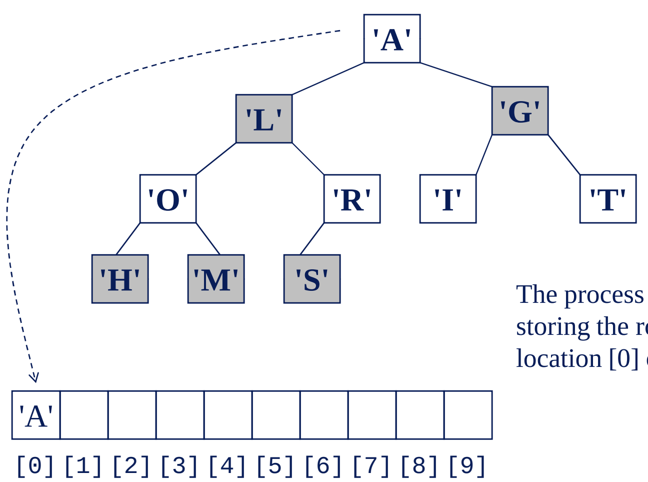
Representing Trees

Array Representation of Complete Binary Trees

- (**Recap**) In a ***complete binary tree***, all leaves have the same depth and are as far to the left as possible
- Complete binary trees have a simple representation using arrays
  - With compile-time (fixed-size) arrays, the size of the data structure does not get larger or smaller during execution
  - Dynamic arrays allow the data structure to grow or shrink

Representing Trees

Array Representation of Complete Binary Trees

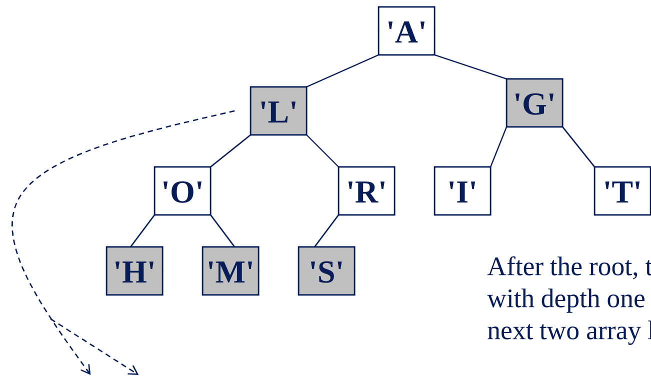


The process begins by storing the root's data at location [0] of the array

Representing Trees

Binary Tree

Array Representation of Complete Binary Trees



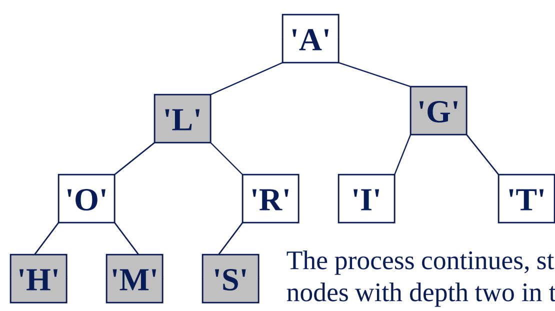
After the root, the two nodes with depth one are stored in the next two array locations as shown

'A'	'L'	'G'							
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Representing Trees

Binary Tree

Array Representation of Complete Binary Trees



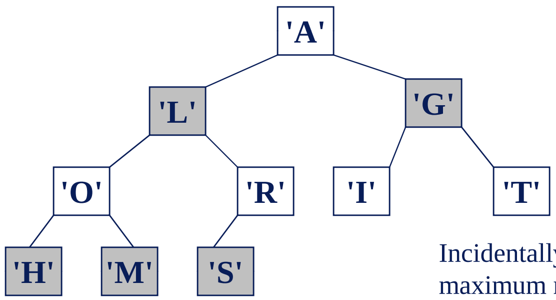
The process continues, storing the next four nodes with depth two in the next four array locations, followed by the storing of the last three nodes with depth three as shown

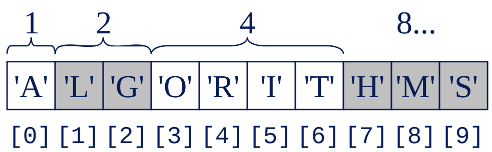
'A'	'L'	'G'	'O'	'R'	'I'	'T'	'H'	'M'	'S'
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Representing Trees

Binary Tree

Array Representation of Complete Binary Trees





Incidentally, notice that the maximum number of nodes at each depth of a binary tree is 2^d, where d is the depth in question

Representing Trees

Binary Tree

Array Representation of Complete Binary Trees
Reason Why it is Convenient

- The data from the ***root*** always appears in the **[0]** element of the array
- If the data for a ***non-root*** node appears in element **[i]** of the array...
  - ...then the data for its ***parent*** is always at location **[(i - 1) / 2]**
    - using ***integer division***
  - ...the data for its ***left child*** is always at location **[2i + 1]**
  - ...and the data for its ***right child*** is always at location**[2i + 2]**

Representing Trees

Binary Tree

Array Representation of Binary Trees

- A dynamic array may be used as an alternative to the fixed-size array implementation
  - A third member variable is needed to keep track of the size of the dynamic array
- ***Non-complete binary trees*** can be implemented using arrays also, but a mechanism to determine which children exist will be needed

Representing Trees

Binary Tree

Pointer-Based (Node-Based) Representation

- We can represent a binary tree by its individual nodes in the same fashion as we did for a linked list
  - Each node is an instance of a **`class`**
    - A class **`template`** would provide desirable flexibility
  - Each node contains pointers that link the node to other nodes (its children)
    - If a child pointer is the ***null address*** then that child (subtree) doesn't exist
    - It is possible to include other pointers (*e.g.*, a pointer to the node's parent) but they are not needed for recursive tree traversal functions (will see)
  - Nodes are dynamically created (memory allocated) and destroyed (memory released) as needed → much like linked list
  - The entire tree is represented by a pointer to the root node (or ***root pointer***) → like the *head pointer* for a linked list

Representing Trees

Class for Binary Tree Nodes (Interface)

```
template <class Item>
class binary_tree_node
{
public:
   typedef Item value_type;
   binary_tree_node(const Item& init_data = Item(),
                    binary_tree_node *init_left = 0,
                    binary_tree_node *init_right = 0);

   ... // other public member functions (see next slide)

private:
   Item data_field;
   binary_tree_node *left_field;
   binary_tree_node *right_field;
};
```


Representing Trees

Class for Binary Tree Nodes (Interface)

```
template <class Item>
class binary_tree_node
{
public:
   ... // typedef and constructor (see previous slide)
   Item& data();
   binary_tree_node*& left();
   binary_tree_node*& right();
   void set_data(const Item& new_data);
   void set_left(binary_tree_node* new_left);
   void set_right(binary_tree_node* new_right);
   const Item& data() const;
   const binary_tree_node* left() const;
   const binary_tree_node* right() const;
   bool is_leaf() const;

... // private section (see previous slide)
};
```

Representing Trees

Class for Binary Tree Nodes (Implementation)

```
template <class Item>
binary_tree_node<Item>
::binary_tree_node(const Item& init_data,
                   binary_tree_node<Item>* init_left,
                   binary_tree_node<Item>* init_right)
: data_field(init_data), left_field(init_left),
  right_field(init_right)
{ }

template <class Item>
binary tree node<Item>*& binary tree node<Item>::left()
{ return left_field; }

template <class Item>
binary_tree_node<Item>*& binary_tree_node<Item>::right()
{ return right_field; }
```


Representing Trees

Class for Binary Tree Nodes (Implementation)

```
template <class Item>
void binary_tree_node<Item>::
set_data(const Item& new_data)
{ data_field = new_data; }

template <class Item>
void binary_tree_node<Item>::
set_left(binary_tree_node<Item>* new_left)
{ left_field = new_left; }

template <class Item>
void binary_tree_node<Item>::
set_right(binary_tree_node<Item>* new_right)
{ right_field = new_right; }

template <class Item>
Item& binary_tree_node<Item>::data()
{ return data_field; }
```

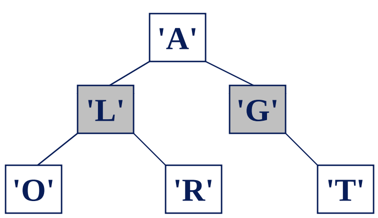
Representing Trees

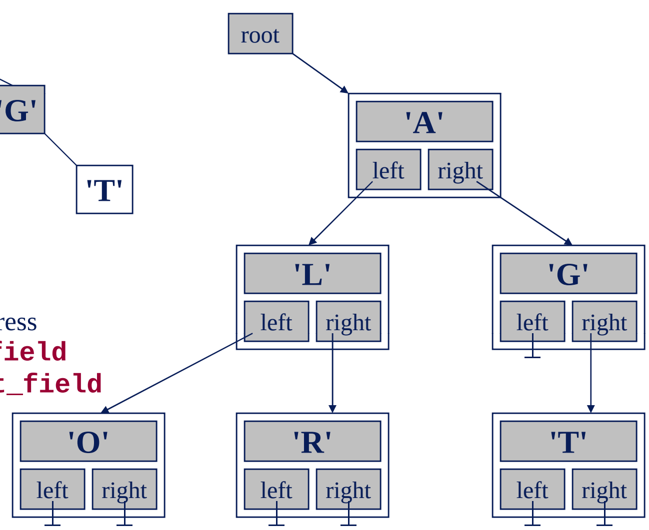
Class for Binary Tree Nodes (Implementation)

```
template <class Item>
const Item& binary_tree_node<Item>::data() const
{ return data_field; }

template <class Item>
const binary_tree_node<Item>*
binary_tree_node<Item>::left() const
{ return left_field; }

template <class Item>
const binary_tree_node<Item>*
binary_tree_node<Item>::right() const
{ return right_field; }

template <class Item>
bool binary_tree_node<Item>::is_leaf() const
{ return (left_field == 0) && (right_field == 0); }
```


Representing Trees

Node-Based Binary Tree





⊥ indicates the null address
left is short for `left_field`
right is short for `right_field`

Node-Based Binary Trees

Binary Tree Toolkit

- Collection of functions for creating and manipulating binary trees
  - Can be used to develop ADTs that use binary trees to store data
  - Much like the linked list toolkit that we have used to implement ADTs that use linked lists to store data
- Two such functions are described in Section 10.3 of the textbook (page 466):
  - **`tree_clear`** → frees up all the nodes of a tree
  - **`tree_copy`** → copies a binary tree
- (More such functions are described in Section 10.4 of the textbook)

Node-Based Binary Trees

Binary Tree Toolkit

Function to Free Up All Nodes of a Tree

```
template <class Item>
void tree_clear(binary_tree_node<Item>*& root_ptr)
{
   if (root_ptr != 0)
   {
      tree_clear( root_ptr->left() );
      tree_clear( root_ptr->right() );
      delete root_ptr;
      root_ptr = 0;
   }
}
```

Node-Based Binary Trees

Function to Copy a Tree

```
template <class Item>
binary_tree_node<Item>*
tree_copy(const binary_tree_node<Item>* root_ptr)
{
   if (root_ptr == 0)
     return 0;
  else
  {
     binary_tree_node<Item> *l_ptr = 0, *r_ptr = 0;
    l_ptr = tree_copy( root_ptr->left() );
    r_ptr = tree_copy( root_ptr->right() );
    return new binary_tree_node<Item>(root_ptr->data(),
                                        l_ptr, r_ptr);
  }
}
```


Textbook Readings

- Chapter 10
  - Section 10.1
  - Section 10.2
  - Section 10.3