Binary Trees

Carlos Moreno
cmoreno@uwaterloo.ca
EIT-4103
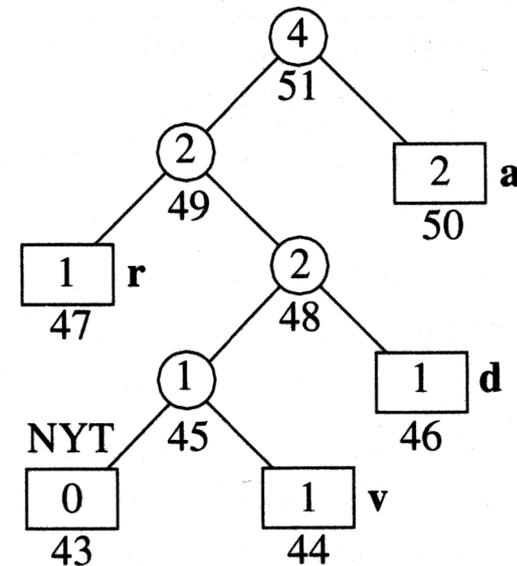
https://ece.uwaterloo.ca/~cmoreno/ece250

Binary Trees

Standard reminder to set phones to silent/vibrate mode, please!

Binary Trees

- ## Today's class:

  - ### We'll look at binary trees — definition and some properties and related concepts.

  - ### Talk about its implementation.

  - ### Look at the notions of perfect and complete binary trees.

    - Implementing it with array storage.

Binary Trees

- The definition of a binary tree is quite straightforward:

    - A tree with the structure constrained such that each node has exactly two children.

Binary Trees

- The definition of a binary tree is quite straightforward:

  - A tree with the structure constrained such that each node has exactly two children.

    – Notice, *exactly* two children — not up to two children!

Binary Trees

- The definition of a binary tree is quite straightforward:

    - A tree with the structure constrained such that each node has exactly two children.

        – Notice, *exactly* two children — not up to two children!

   - Each child is either empty or another binary tree.

Binary Trees

- The definition of a binary tree is quite straightforward:

  - A tree with the structure constrained such that each node has exactly two children.

    – Notice, *exactly* two children — not up to two children!

  - Each child is either empty or another binary tree.

  - Given this constraint, we can label the two children as *left* and *right* nodes or subtrees.
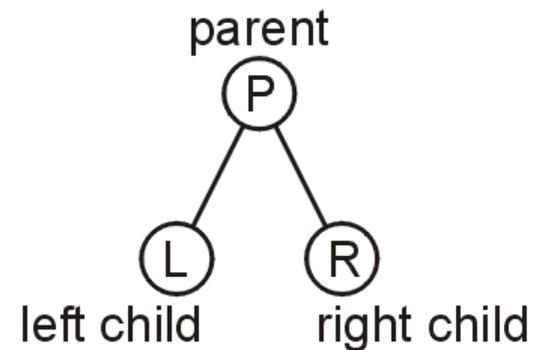
Binary Trees

- Examples of binary trees with five nodes:
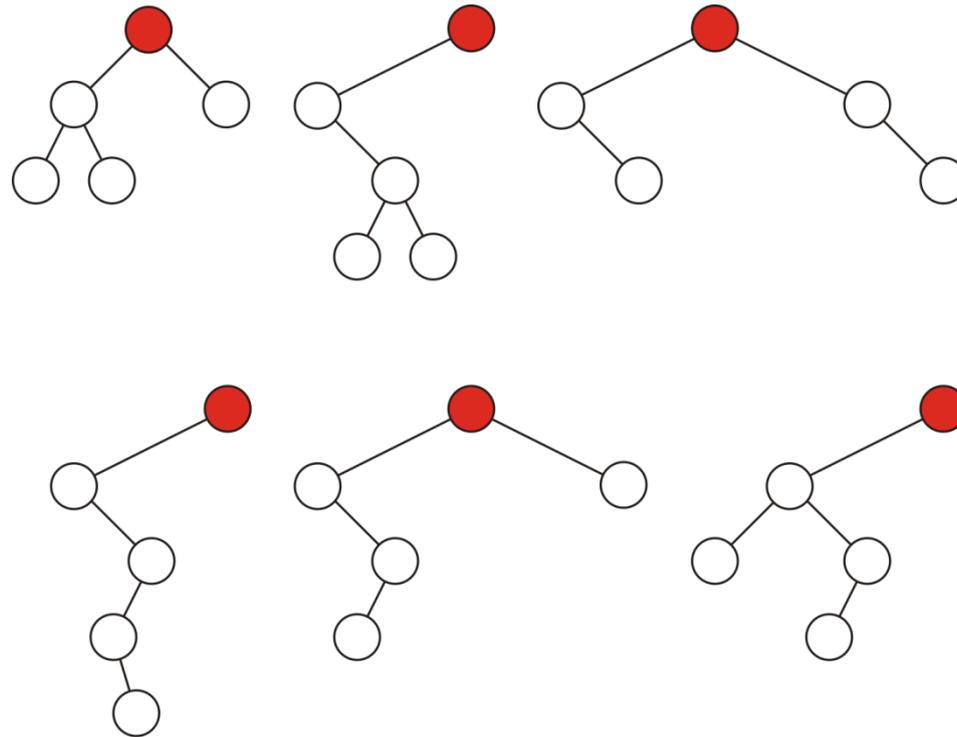
Binary Trees

- Definition: A *full node* is a node where both left and right sub-trees are non-empty trees:

Binary Trees

- Definition: A *full node* is a node where both left and right sub-trees are non-empty trees:
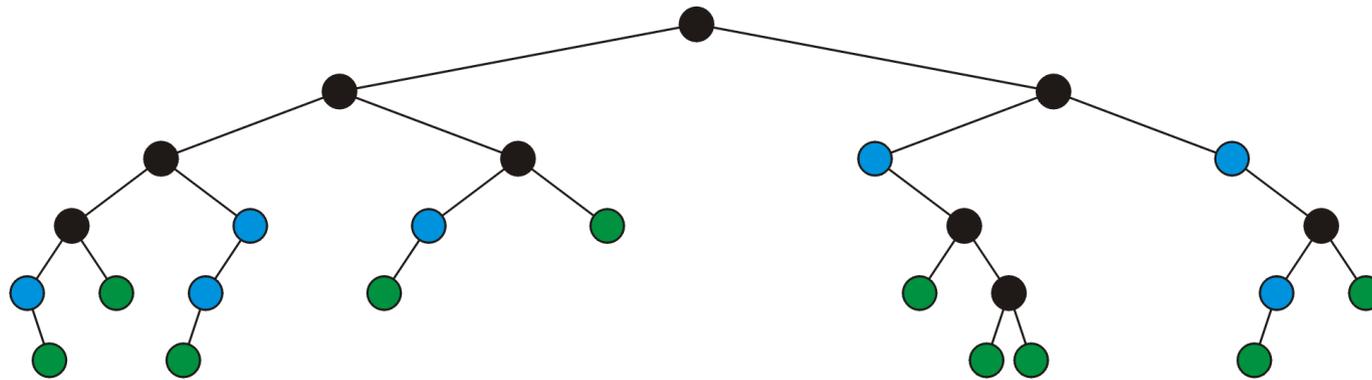
■ Full nodes   ■ Leaf nodes   ■ Neither

Binary Trees

- Definition: An *empty node* or *null sub-tree* is a location where a new leaf node (or a sub-tree) could be inserted.

Binary Trees

- Definition: An *empty node* or *null sub-tree* is a location where a new leaf node (or a sub-tree) could be inserted.

  - Graphically, the missing branches.
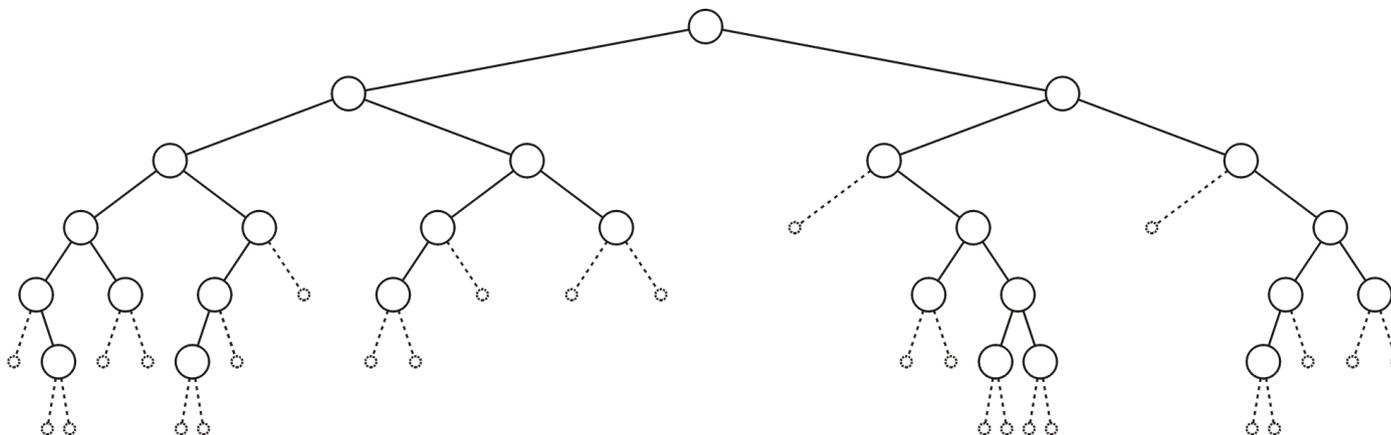
Binary Trees

- Definition: A *full binary tree* is a binary tree where each node is either a full node or a leaf node.

Binary Trees

- Definition: A *full binary tree* is a binary tree where each node is either a full node or a leaf node.

Binary Trees

- Implementing binary trees...

Binary Trees

- **Implementing binary trees...**
  - Clearly, since these are specific (constrained) types of trees, we could use a normal implementation of a tree, and ensure that the constraints are always applied.

Binary Trees

- **Implementing binary trees...**
    - Clearly, since these are specific (constrained) types of trees, we could use a normal implementation of a tree, and ensure that the constraints are always applied.
        - Not a very interesting approach! (*why?*)

Binary Trees

- ## Implementing binary trees...

  - ### A better approach is, of course, having two named pointers (as in, two data members), `left` and `right` (well, or `d_left`, `d_right`, or whatever naming convention for data members).

Binary Trees

- Implementing binary trees...

  - A better approach is, of course, having two named pointers (as in, two data members), `left` and `right` (well, or `d_left`, `d_right`, or whatever naming convention for data members).

    – If a child node is absent (i.e., an empty node or null sub-tree), we represent it with a null pointer in the corresponding child (left or right).

Binary Trees

```
template <typename Type>
class Binary_node
{
    Type d_element;
    Binary_node<Type> * d_parent;
    Binary_node<Type> * d_left;
    Binary_node<Type> * d_right;

public:
    Binary_node (const Type & obj);

    Type retrieve() const;
    Node<Type> * left() const;
    Node<Type> * right() const;

    // etc.
};
```

Binary Trees

- A small caveat ....

Binary Trees

- A small caveat ....

    - The code for traversal (e.g., recursive functions or recursive node's methods) can become a bit more "verbose" than in the case of general trees.

Binary Trees

- A small caveat ....

    - The code for traversal (e.g., recursive functions or recursive node's methods) can become a bit more "verbose" than in the case of general trees.

    - If a pointer to a child node is null, we're not allowed to dereference it.

Binary Trees

- A small caveat ....

  - The code for traversal (e.g., recursive functions or recursive node's methods) can become a bit more "verbose" than in the case of general trees.

  - If a pointer to a child node is null, we're not allowed to dereference it.

    - And since each of the two pointers has its own name, we have to explicitly check them (as in, *individually*!)

Binary Trees

- A small caveat ....

  - The code for traversal (e.g., recursive functions or recursive node's methods) can become a bit more "verbose" than in the case of general trees.

  - If a pointer to a child node is null, we're not allowed to dereference it.

    – And since each of the two pointers has its own name, we have to explicitly check them (as in, *individually*!)

  - For example, here's what a recursive size() method could look like...

Binary Trees

```cpp
template <typename Type>
int Binary_node<Type>::size() const
{
    int count = 1;    // this one
    if (left() != NULL)
    {
        count += left()->size();
    }

    if (right() != NULL)
    {
        count += right()->size();
    }

    return count;
}
```

Binary Trees

- In this particular example, a recursive function (*standalone* function, as opposed to a method) would be simpler with respect to that detail....

Binary Trees

```cpp
template <typename Type>
int size (const Binary_node<Type> * node)
{
    if (node == NULL)
    {
        return 0;
    }

    return 1 + size(node->left()) + size(node->right());
}
```

Binary Trees

```
template <typename Type>
int size (const Binary_node<Type> * node)
{
    if (node == NULL)
    {
        return 0;
    }

    return 1 + size(node->left()) + size(node->right());
}
```

Why, then, if we can get away with it this way we can't do the
same with the method approach?

Binary Trees

- Let's look at *perfect* binary trees...

Binary Trees

- A *perfect binary tree* of height h is a binary tree where:

  - All leaf nodes have the same depth h.
  - All other nodes are full.

Binary Trees

- A *perfect binary tree* of height h is a binary tree where:

  - All leaf nodes have the same depth h.

  - All other nodes are full.

  - Here's an example of a perfect binary tree:

Binary Trees

- Or a nice recursive definition:

  - A binary tree of height 0 is perfect.

  - A binary tree with height $h > 0$ is perfect if both sub-trees are perfect binary trees of height $h-1$.

Binary Trees

- From this definition, we can prove, for example, that a perfect binary tree of height h has $2^{h+1}-1$ nodes.

Binary Trees

- From this definition, we can prove, for example, that a perfect binary tree of height h has $2^{h+1}-1$ nodes.

- We'll proceed by induction on h. So, we have to prove that: (1) The statement is true for $h = 0$; and (2) that if the statement is true for h, that implies that it is also true for $h+1$

Binary Trees

- You do it for now (we'll show and discuss the proof during class)

Binary Trees

- As a direct consequence of this, the height of a perfect binary tree of n nodes is $\Theta(\log n)$:

$$n = 2^{h+1} - 1 \;\Rightarrow\; h = \lg(n+1) - 1 \;=\; \Theta(\log n)$$

Binary Trees

- As a direct consequence of this, the height of a perfect binary tree of n nodes is $\Theta(\log n)$:

$$n = 2^{h+1} - 1 \;\Rightarrow\; h = \lg(n+1) - 1 \;=\; \Theta(\log n)$$

- This is interesting — many operations with trees have a run time that goes with the depth of some path within the tree; if we have a perfect tree (or something *close* to it), we know that those operations run in O(log n).

Binary Trees

- Now, what could be *close* to a perfect binary tree?

Binary Trees

- Now, what could be *close* to a perfect binary tree?

  - One of the limitations with perfect binary trees is that the number of nodes is always $n = 2^k - 1$.

  - It would be nice to have something similar, but defined for all values of n.

Binary Trees

- Definition (informal): A *complete* binary tree is a binary tree that is filled at each depth from left to right (sort of filled in the same order as a breadth-first traversal).

   - That is, we are not allowed insertions at arbitrary positions!

Binary Trees

- Definition (informal): A *complete* binary tree is a binary tree that is filled at each depth from left to right (sort of filled in the same order as a breadth-first traversal).

  - That is, we are not allowed insertions at arbitrary positions!

  - Also, removals are only allowed from the "last" position.

Binary Trees

- Definition – recursive:

  - A binary tree with height 0 is a *complete* binary tree.

  - A complete binary tree of height $h > 0$ is a binary tree where either:

    - The left sub-tree is a complete tree of height $h-1$ and the right sub-tree is a perfect tree of height $h-2$, or
    - The left sub-tree is a perfect tree of height $h-1$ and the right sub-tree is a complete tree of height $h-1$.

Binary Trees

- Graphically:
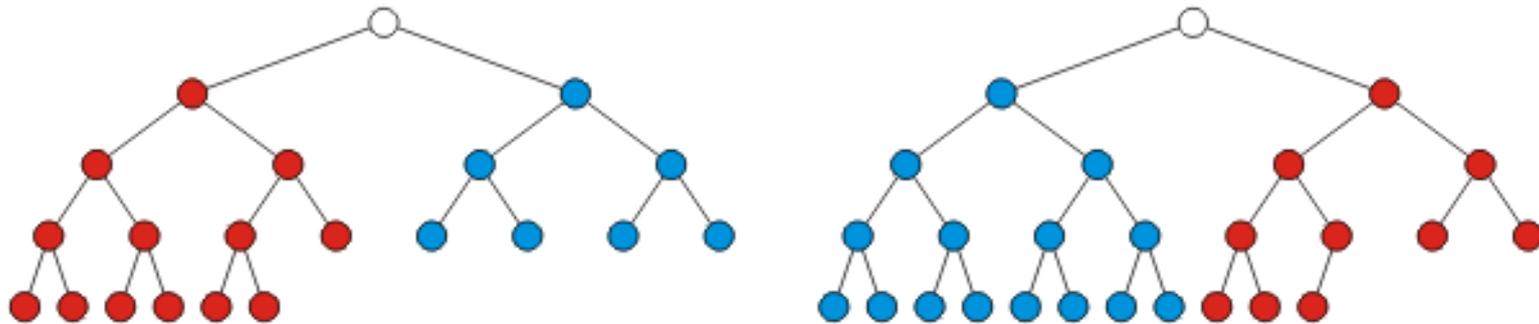
Binary Trees

- The *very* interesting aspect of a complete binary tree is that we can efficiently store it using an array!

Binary Trees

- The *very* interesting aspect of a complete binary tree is that we can efficiently store it using an array!

  - We traverse the tree in breadth-first order, placing the entries into the array
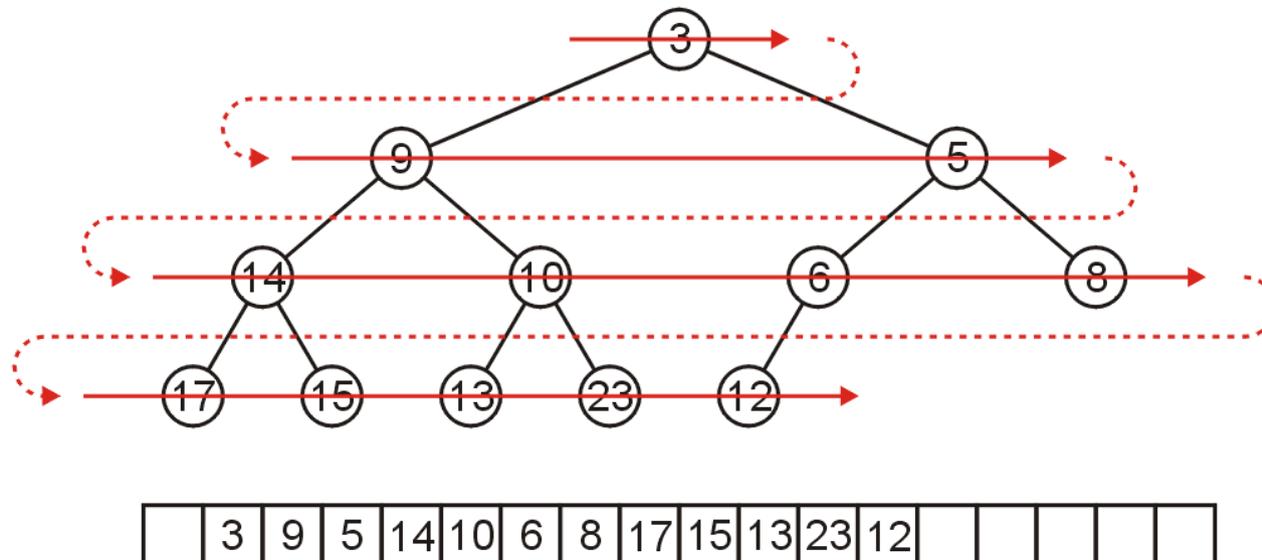
Binary Trees

- We notice that insertions and removals can only be done at the end of the array (not a bad thing — quite the contrary, if we think about it!)

Binary Trees

- Ok, but we could have done this with any other type of trees, right?

Binary Trees

- Ok, but we could have done this with any other type of trees, right?

  - The problem would be: how do we efficiently access the nodes? (e.g., given a node, how do we access its children? Its parent? etc.)

Binary Trees

- Ok, but we could have done this with any other type of trees, right?

  - The problem would be: how do we efficiently access the nodes? (e.g., given a node, how do we access its children? Its parent? etc.)

  - With binary trees, the "fixed" structure of each node having exactly two children yields a nice and simple formula to relate these!

Binary Trees

- Ok, but we could have done this with any other type of trees, right?
    - The problem would be: how do we efficiently access the nodes? (e.g., given a node, how do we access its children? Its parent? etc.)
    - With binary trees, the "fixed" structure of each node having exactly two children yields a nice and simple formula to relate these!
        - At each depth, there are twice as many nodes as in the previous depth!
        - Can you see the rest of where this is going?

Binary Trees

- Back to our "why is this only for complete binary trees" case ...

Binary Trees

- Back to our "why is this only for complete binary trees" case ...

- Again, we could ask: why can't we do this with any binary tree? (we agree that with a general tree, efficient access to children and parent is a problem). But any binary tree does have the structure to facilitate this.

Binary Trees

- Back to our "why is this only for complete binary trees" case ...

- Again, we could ask: why can't we do this with any binary tree? (we agree that with a general tree, efficient access to children and parent is a problem). But any binary tree does have the structure to facilitate this.

  - Can you see where this is going?

Binary Trees

- The worst-case storage requirement for storing an arbitrary binary tree of n nodes is $\Theta(2^n)$

  - Worst-case happens if the elements form a linear arrangement (i.e., every node has only one child)

Summary

- During today's lesson, we discussed:
  - Binary trees
    - Definition
    - Some of its properties and related concepts
    - Discussed some aspects of their implementation
  - Perfect binary trees
  - Complete binary trees
    - Implementing them in terms of an array!