CS 234: Data Types and Structures

Naomi Nishimura

Module 5

Date of this version: October 8, 2019

WARNING: Drafts of slides are made available prior to lecture for your convenience. After lecture, slides will be updated to reflect material taught. Check the date on this page to make sure you have the correct, updated version.

WARNING: Slides do not include all class material; if you have missed a lecture, make sure to find out from a classmate what material was presented verbally or on the board.

Case study

Problem: When colour is applied to a part of a web page, what other parts of the page will obtain the same colour?

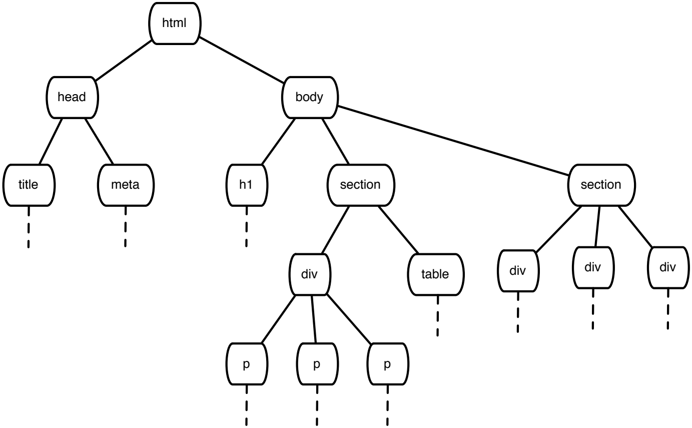
For example, if a section is coloured, the paragraphs and lists will get the same colour.

Recipe for user/plan

1. Determine types of data and operations.
2. For each type, choose/modify/create an ADT.
3. Develop pseudocode algorithm using ADT operations.
4. Calculate cost of algorithm with respect to costs of operations.
5. Using information from provider, choose best option.

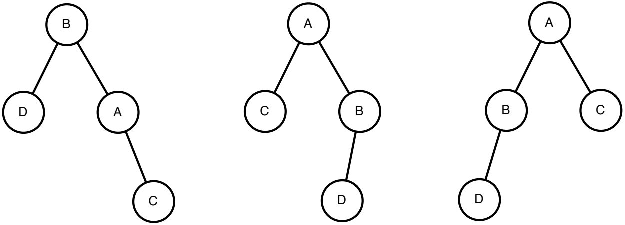
Representing a web page

The Document Object Model represents a web page as a tree.



Tree review

Which of these trees are the same?



Basic definitions

A **tree** is formed of **nodes** connected by **edges**. (This is not the same as a node in a linked list.)

In a **rooted tree**, one node is designated as the **root** of the tree.

In a drawing where the root is at the top, an edge connects a **parent** to a **child**, where the parent is the node closer to the root.

Nodes that share a parent are **siblings**.

A node without children is a **leaf**; a node that is not a leaf is an **internal node**.

A node's parent, its parent's parent, and so on up to the root are its **ancestors**; a node's children, children's children, and so on are its **descendants**.

A node and all its descendants form the **subtree rooted at** that node.

Types of rooted trees

A tree is **unordered** if there is no order specified on the children of a node, and **ordered** otherwise.

A **binary tree** is a tree in which each parent has at most two children and each child is specified as either a **left child** or a **right child**.
In a binary tree, the subtree rooted at the left child is the **left subtree** and the subtree rooted at the right child is the **right subtree**.

Terminology for rooted trees

The **path** between nodes n_0 and n_k is the sequence of nodes $\{n_0, n_1, \ldots, n_k\}$ such that there is an edge between n_i and n_{i+1} for all $0 \leq i < k$; a path is **simple** if each node appears at most once in the sequence. The **length** of a path is the number of edges in the path.

The **depth of a node** n is the length of the path between n and the root; a root is thus at depth 0. All nodes of the same depth are on the same **level**.

The **height of a node** n is the maximum length of any path between n and a leaf in the subtree rooted at n; a leaf thus has height 0. The **height of a tree** is the height of the root of the tree.

Nodes as positions to store data

In earlier ADTs, we accessed data by position, such as:

- top (ADT Stack)
- front (ADT Queue)
- index (ADT Indexed Sequence)
- rank (ADT Ranking)
- row and column (ADT Grid)

We can navigate in a tree by starting at the root, choosing a child, and so on to find a specific node.

How do we refer to a particular node?

Instead of using the path from the root, we'll associate a unique **ID** with each node. The type of data used for the ID may depend on the data structure implementing the ADT (e.g. index in an array or pointer to a node in a linked structure).

Data stored in nodes

Depending on the application, a node of a tree can store various types of data, such as:

- a value
- a weight
- a colour

For now, we will define our ADTs such that each node stores a single value.

Search operations for trees

- Find the value of a node
- Find the root of the tree
- Find the parent of a node
- Find a specific child of a node
- Find all children of a node
- Find the node storing a particular value
- Find all nodes storing a particular value
- Find all nodes in the tree

Issues to consider:

How can multiple nodes be returned?

Use a Group B ADT that allows us to extract all the data items, possibly in some specific order.

Modification operations for trees

- Add a new node
- Delete a node
- Delete a subtree
- Change the value stored in a node
- Swap values stored in two nodes
- Swap subtrees

Issues to consider:

What remains after a node is deleted?

Initially just delete leaves.

Initially start with binary trees.

ADT Binary Tree, without modifications

Preconditions: For all *B* is a binary tree and *Node* is a node in *B*; for *Root B* is not empty.

Name	Returns
Create()	a new empty binary tree
Is_Empty(B)	*True* if empty, else *False*
Root(B)	root of *B*
Value(B, Node)	value stored in *Node*
Parent(B, Node)	parent of *Node* if any, else *False*
Left_Child(B, Node)	left child of *Node* if any, else *False*
Right_Child(B, Node)	right child of *Node* if any, else *False*

ADT Binary Tree, modifications

Preconditions: For all *B* is a binary tree, *Node* is a node in *B*, and *Data* is a data item; for *ADD_LEAF* either *Par* and *Side* are both empty or *Par* is a node in *B* and *Side* is *Left* or *Right*; for *DELETE_LEAF* *Node* is a leaf.

Postconditions: Mutation by *SET_VALUE* (sets value of *Node* to *Data*), *ADD_LEAF* (creates a new node containing *Data* to replace/add the root if *Par* is empty and to replace/add the *Side* subtree of *Par* otherwise), and *DELETE_LEAF* (deletes *Node*).

Name	Returns
SET_VALUE(B, Node, Data)	
ADD_LEAF(B, Par, Side, Data)	new added node storing *Data*
DELETE_LEAF(B, Node)	

Example of use of ADT Binary Tree operations

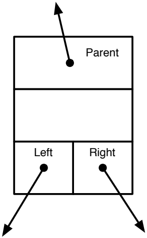
Fruit $\leftarrow$ *CREATE()*
Apple $\leftarrow$ *ADD_LEAF(Fruit, None, None, apple)*
Guava $\leftarrow$ *ADD_LEAF(Fruit, Apple, Left, guava)*
Peach $\leftarrow$ *ADD_LEAF(Fruit, Apple, Right, peach)*
Mango $\leftarrow$ *ADD_LEAF(Fruit, Guava, Right, mango)*
One $\leftarrow$ *ROOT(Fruit)*
Two $\leftarrow$ *PARENT(Fruit, Mango)*
Three $\leftarrow$ *LEFT_CHILD(Fruit, Guava)*
Four $\leftarrow$ *RIGHT_CHILD(Fruit, Guava)*
DELETE_LEAF(Fruit, Peach)
Five $\leftarrow$ *RIGHT_CHILD(Fruit, Apple)*

Linked implementation of ADT Binary Tree

Data structures:

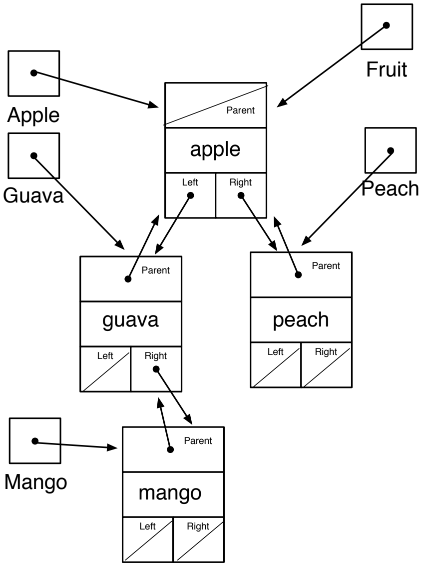
- Variable pointing to root node, if any
- Nodes storing data items and three pointers *Parent* (to parent), *Left* (to left child), and *Right* (to right child)

Worst-case running times of operations are all in $\Theta(1)$.
Cost of searching for a node from the root depends on depth.

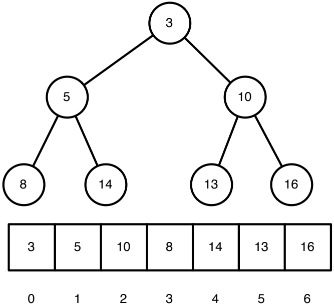


Caution: The word “node” can mean either or both of “node in a tree” and “node in a linked implementation.”

Example illustrated



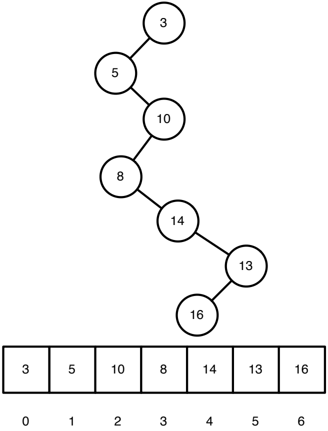
Contiguous implementation of ADT Binary Tree



Observations:

- For node at index p, index of left child is $2p+1$
- For node at index p, index of right child is $2p+2$
- For node at index p, index of parent is $\lfloor (p-1)/2 \rfloor$

Exploring a contiguous implementation



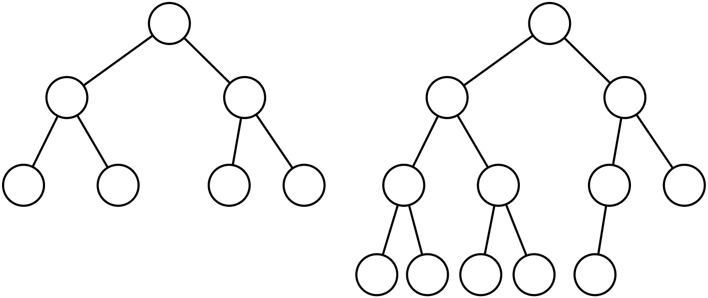
Observations:

- For node at index p, index of left child may not be $2p+1$
- For node at index p, index of right child may not be $2p+2$
- For node at index p, index of parent may not be $\lfloor (p-1)/2 \rfloor$

More terminology for binary trees

In a **perfect** binary tree, each node has zero or two children and all leaves are at the same depth.

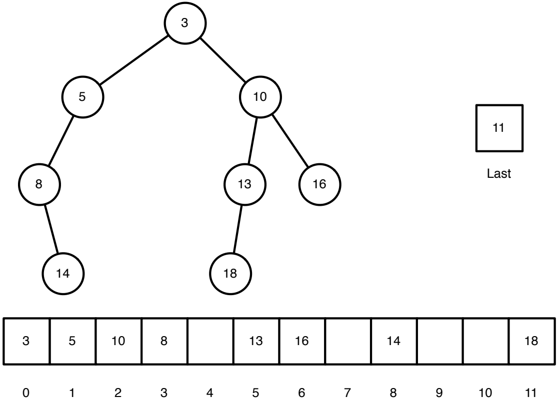
In a **complete** binary tree every level, except possibly the last, is completely filled, and all nodes on the last level are as far to the left as possible.



Contiguous implementation of a ADT Binary Tree

Data structures:

- Array storing values level by level as if all nodes were present
- Variable *Last* with the last index storing an element



Computing a sibling

Options for computing a sibling:

- Write an algorithm using existing ADT operations.
- Augment the ADT by adding a new operation.

Use existing ADT operations:

- Use *Parent* to find parent.
- Use *Left_Child* and *Right_Child* to find children of parent.
- If there is only one, return *False*.
- If there are two, return the one which is not the node itself.

Modifying the implementations for the augmented ADT

Linked implementation:

- Use *Parent* pointer to find parent.
- Use *Left* and *Right* pointers to find children of parent.
- If there is only one, return *False*.
- If there are two, return the one which is not the node itself.
- Cost is $\Theta(1)$.

Contiguous implementation:

- For node at odd index p, index of sibling is $p+1$ (if $p+1$ is at most *Last*).
- For node at even positive index p, index of sibling is $p-1$.
- Cost is $\Theta(1)$.

ADT Ordered Tree

Preconditions: For all *O* is an ordered tree, *Node* is a node in *O*, and *Data* is a data item; for *ONE_CHILD* *Index* is a nonnegative integer at most one less than the number of children of *Node*; for *ADD_LEAF* *Par* is a node in *O* or empty and *Sib* is a child of node *Par* or empty.

Postconditions: Mutation by *ADD_LEAF* (creates a new node containing *Data* to replace/add the root if *Par* is empty, as the first child of *Par* if *Sib* is empty, and otherwise as the next sibling of *Sib*).

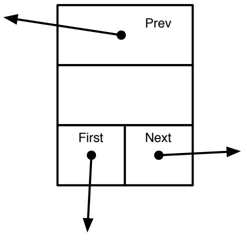
(*CREATE*, *IS_EMPTY*, *ROOT*, *VALUE*, *PARENT*, *SET_VALUE*, and *DELETE_LEAF* like in ADT Binary Tree)

Name	Returns
CHILDREN(O, Node)	all children of *Node* (Group B ADT)
ONE_CHILD(O, Node, Index)	child *Index*
ADD_LEAF(O, Par, Sib, Data)	new added node storing *Data*

Linked implementation of ADT Ordered Tree

Data structures:

- Variable *Root* pointing to root node, if any
- Nodes storing data items and three pointers *Prev* (to parent if first child or previous sibling otherwise), *First* (to first child), and *Next* (to next sibling)



Pseudocode for *Parent(O, Node)*

Use dot notation for fields inside a node in the linked structure.

if *Root(O) == Node*
    return *False*
Found $\leftarrow$ *False*
Current $\leftarrow$ *Node*
while not *Found*
    Previous $\leftarrow$ *Current.Prev*
    if *Current == Previous.First*
        Found $\leftarrow$ *True*
    else
        Current = Previous
return *Previous*

Computing the next sibling

Options for computing the next sibling:

- Write an algorithm using existing ADT operations.
- Augment the ADT by adding a new operation.

Using existing ADT operations:

- Use *Parent* to find parent.
- Use *Children* to find children.
- Scan children to determine next sibling.

Modifying the linked implementation:

- Use *Next* pointer to find next sibling.

Defining and implementing ADT Unordered Tree

ADT definition:

- Similar to ADT Ordered Tree
- Specify only parent, not sibling, when adding a node

Data structures:

- Same data structure as for ADT Ordered Tree
- Adapt algorithms to exploit fact that order of children is not significant

Returning all nodes

Options for returning all nodes:

- Write an algorithm using existing ADT operations.
- Augment the ADT by adding a new operation.

We can find all nodes by determining the root using *Root* and then repeatedly using *Children* (or *Left_Child* and *Right_Child*) to determine all descendants of the root.

Alternatively, we can use the recursive definition of a tree, in which each child of the root can be viewed as the root of a (smaller) tree. The result for the original tree will be determined using the results (obtained recursively) on the smaller trees.

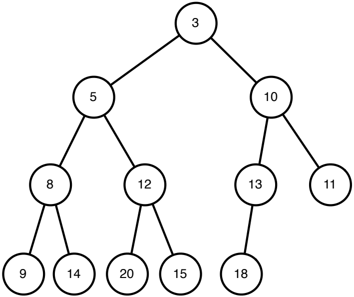
Tree traversals

A **tree traversal** is an ordering of the nodes in the tree.

- In a **level order traversal**, nodes appear in increasing order of depth.
- In a **postorder traversal**, each node appears after its children.
- In a **preorder traversal**, each node appears before its children.
- In an **inorder traversal** (only in a binary tree), for each node all nodes in the left subtree come before the node and all nodes in the right subtree come after the node.

Note: Traversals can be viewed as templates for processing (not just numbering) nodes in a given order.

Traversal example



Algorithms for traversals

Level order:

- Use *Root* to set the current node to the root.
- Use *Children* or *Left_Child* and *Right_Child* to determine children of the current node.
- Add the children to an ADT Queue.
- Repeat the process with the first node in the queue as the current node.

All other traversals:

- Create a recursive algorithm that numbers nodes starting at a given number and produces the last number used.
- For postorder, number the subtrees (in order if a binary or ordered tree), and then give the next number to the root.
- For preorder, give the first number to the root and then number subtrees (in order if a binary or ordered tree).
- For inorder, number the left subtree, give the next number to the root, then number the right subtree.

Modifying an implementation

In a linked implementation, we can **thread** nodes together by adding an extra pointer from a node to the next node in the traversal.

In a contiguous implementation, we obtain a level-order traversal by examining values in order of increasing index.

Module summary

Topics covered:

- Case study: Web page
- Trees
- Decision tree
- Data stored in nodes
- Operations for trees
- ADT Binary Tree
- Linked implementation
- Contiguous implementation
- Perfect and complete trees
- Computing siblings
- ADT Ordered Tree
- Linked implementation
- Computing the next sibling
- ADT Unordered Tree
- Tree traversals