CS 234

Module 5

October 18, 2018

ADTs representing structure

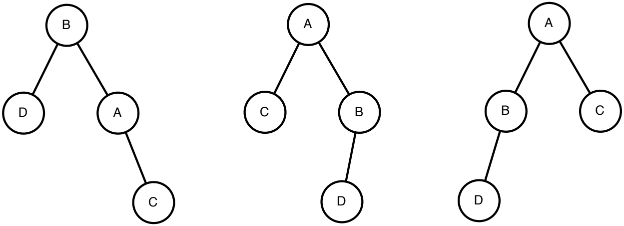
We have seen ADTs where:

- There is no relation among items.
- Items are orderable types of data.
- Order is imposed by operations (e.g. ADT Stack).

Here, we specify more complex structure as we add to the ADT.

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.

Definitions for 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**. The subtree rooted at the left child is the **left subtree** and the subtree rooted at the right child is the **right subtree**.

The **depth of a node** is the number of ancestors of the node; a root is thus at depth 0.

The **height of a node** is determined by considering, in the subtree rooted at the node, the maximum number of edges passed going from a leaf to the node. A leaf has height 0.

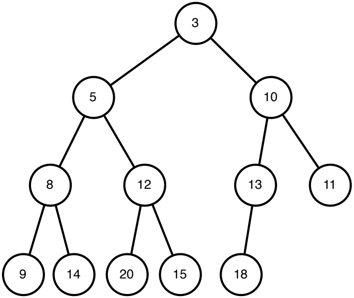
The **height of a tree** is the height of the root of the tree.

Tree traversals

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

- In a **postorder traversal**, each node appears after its children.
- In a **preorder traversal**, each node appears before its children.
- In a **level order traversal**, nodes appear in increasing order of depth.
- 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.

Traversal example

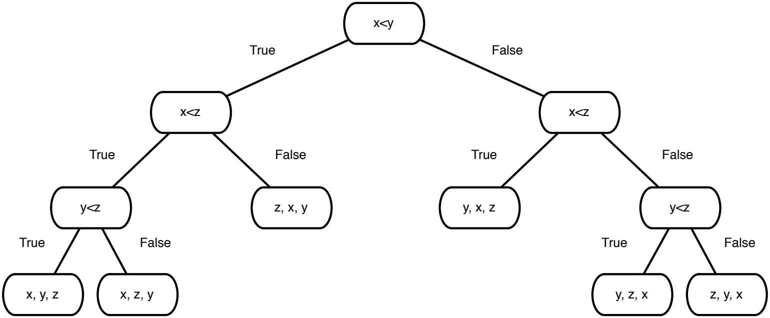


Applications of trees

- family tree (with one parent), language family tree, evolutionary tree
- hierarchy in a business
- organization of a text document (chapters, etc.)
- parse tree
- arithmetic expression

Decision tree

In a **decision tree** representing an algorithm, each node is labeled with a decision being made, the children of a node correspond to the different choices depending on the outcome of the decision, and each leaf corresponds to a possible outcome of the algorithm.



Possible operations for trees

Basic operations:

- Create()
- IsEmpty(T)

Returning a value:

- ValueAtNode(T, n)

Returning more than one value:

- PostOrderValues(T)
- PreOrderValues(T)
- LevelOrderValues(T)

Returning a node:

- FindNode(T, value)
- Root(T)
- Parent(T, n)

Returning more than one node:

- Children(T, n)
- Siblings(T, n)
- Descendants(T, n)
- Ancestors(T, n)
- Leaves(T)
- PostOrderNodes(T)
- PreOrderNodes(T)
- LevelOrderNodes(T)

More operations

Changing the tree:

- ReplaceValue(T, n, newvalue)
- SwapSubtrees(T, n1, n2)
- AddNode(T, p, value)
- DeleteNode(T, n)
- DeleteSubtree(T, n)

Issues to resolve

- Returning multiple values or nodes (Specify ADT; give details in code interface.)
- Specifying the result of deleting a node (Multiple options.)

ADT Binary Tree

Preconditions: For all B is a binary tree, n is a node in B; for Root B is nonempty; for AddNode either p is in B and side is Left or Right or p and side are both None; for DeleteLeaf n is a leaf.

Postconditions: Mutation by AddNode (replace previous subtree or root) and DeleteLeaf (delete leaf).

Name	Returns
Create()	a new empty binary tree
IsEmpty(B)	true if empty, else false
Root(B)	root of B
Parent(B, n)	parent if any, else false
LeftChild(B, n)	left child if any, else false
RightChild(B, n)	right child if any, else false
AddNode(B, p, value, side)	added node
DeleteLeaf(B, n)	

Example of use of ADT Binary Tree operations

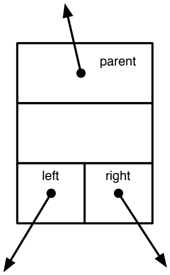
tree $\leftarrow$ Create()
root $\leftarrow$ AddNode(tree, None, apple, None)
leftchild $\leftarrow$ AddNode(tree, root, guava, Left)
rightchild $\leftarrow$ AddNode(tree, root, peach, Right)
grandchild $\leftarrow$ AddNode(tree, leftchild, mango, Right)
print(Root(tree) $==$ root)
print(Parent(tree, grandchild) $==$ leftchild)
print(LeftChild(tree, leftchild) $==$ leftchild)
print(RightChild(tree, leftchild) $==$ grandchild)
DeleteLeaf(tree, rightchild)
print(RightChild(tree, root))

Linked implementation of ADT Binary Tree

Data structure:

- Variable *root* 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.



Note: Node has two meanings here.

Example illustrated

tree $\leftarrow$ Create()
root $\leftarrow$ AddNode(tree, None, apple, None)
leftchild $\leftarrow$ AddNode(tree, root, guava, Left)
rightchild $\leftarrow$ AddNode(tree, root, peach, Right)
grandchild $\leftarrow$ AddNode(tree, leftchild, mango, Right)
print(Root(tree) == root)
print(Parent(tree, grandchild) == leftchild)
print(LeftChild(tree, leftchild) == leftchild)
print(RightChild(tree, leftchild) == grandchild)
DeleteLeaf(tree, rightchild)
print(RightChild(tree, root))

Computing siblings - two views

User view:

- Use ADT operations.
- Use Parent to find parent.
- Use LeftChild and RightChild to find children of parent.
- If there is only one, return node itself (no sibling).
- If there are two, return the one which is not the node itself.

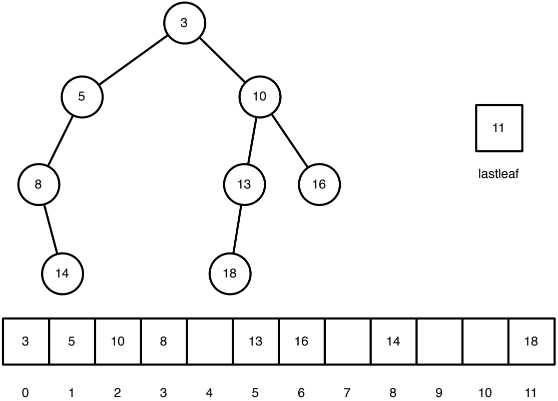
Provider view:

- Create a new ADT operation.
- Use the data structure directly.
- Use *parent* pointer to find parent.
- Use *left* and *right* pointers to find children of parent.
- If there is only one, return node itself (no sibling).
- If there are two, return the one which is not the node itself.

Array implementation of a ADT Binary Tree

Data structure:

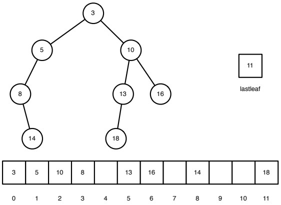
- Array storing values in level-order order as if all nodes were present.
- Variable *lastleaf* with the last index storing an element.



Details of implementation

Parent, children, and siblings can be determined from index in $\Theta(1)$ time:

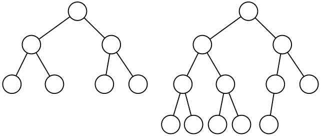
- For node at index p, index of left child is $2p+1$
- For node at index p, index of right child is $2p+2$
- For node at index p, index of parent is $\lfloor (p-1)/2 \rfloor$
- For node at odd index p, index of sibling is $p+1$.
- For node at even positive index p, index of sibling is $p-1$.



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.



ADT Ordered Tree

Preconditions: For all O is an ordered tree, n is a node in O; for Root O is nonempty; for AddNode p is in B and s is its child, p is in B and s is None, or p and s are both None; for DeleteLeaf n is a leaf.

Postconditions: Mutation by AddNode (add node with value as next sibling of s, as first child of p if s is None, or as root if p and s are both None) and DeleteLeaf (delete leaf).

(Create(), IsEmpty(O), Root(O), Parent(O, n), and DeleteLeaf(O, n) like in ADT Binary Tree)

Name	Returns
Children(O, n)	all children (ADT List)
OneChild(O, n, i)	ith child if any, else false
AddNode(O, p, value, s)	added node

Devising a linked implementation

Draw node with pointers to children

- label/item
- pointer to parent
- pointer to first child
- pointer to second child
- more pointers

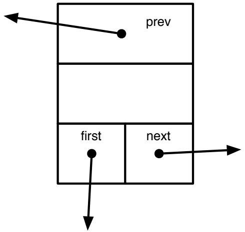
Reject idea except for special case when we know there is a small number of children (maybe more than two but less than unlimited).

Recall what we did in first-year CS when we wanted to store an unbounded number of values together (instead of a structure, we used a list).

Linked implementation of ADT Ordered Tree

Data structure:

- 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), *next* (to next sibling), and *first* (to first child).



Implementing Parent

```
found ← False
current ← x
while not found
    y ← current.prev
    if current == y.first
        found ← True
    else
        current = y
return y
```

Computing next sibling: two views

User view:

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

Provider view:

- Create a new ADT operation.
- Use the data structure directly.
- 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
- Return ADT Set instead of ADT List of children

Data structures:

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

Customizing data structures for trees

Customizing nodes:

- Add extra fields with more information.
- Add extra pointers to connect information.

Variations:

- Store all data in leaves, using internal nodes for search.
- Thread nodes, e.g. as an in-order traversal, by adding an extra pointer from each node to another node in the tree.

Note: Some extra information may be hard to maintain, so it is best when additions and deletions are rare. (Examples: threading, storage of depth or height.)