csci 210: Data Structures

Trees
Summary

Topics

• general trees, definitions and properties
• interface and implementation
• tree traversal algorithms
  • depth and height
  • pre-order traversal
  • post-order traversal
• binary trees
  • properties
  • interface
  • implementation

• binary search trees
  • definition
  • h-n relationship
  • search, insert, delete
  • performance

READING:

• GT textbook chapter 7 and 10.1
Trees

- So far we have seen linear structures
  - linear: before and after relationship
  - lists, vectors, arrays, stacks, queues, etc

- Non-linear structure: trees
  - probably the most fundamental structure in computing
  - hierarchical structure
  - Terminology: from family trees (genealogy)
Trees

- store elements hierarchically
- the top element: root
- except the root, each element has a parent
- each element has 0 or more children
Trees

Definition
- A tree $T$ is a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following:
  - if $T$ is not empty, $T$ has a special tree called the root that has no parent
  - each node $v$ of $T$ different than the root has a unique parent node $w$; each node with parent $w$ is a child of $w$

Recursive definition
- $T$ is either empty
- or consists of a node $r$ (the root) and a possibly empty set of trees whose roots are the children of $r$

Terminology
- siblings: two nodes that have the same parent are called siblings
- internal nodes
  - nodes that have children
- external nodes or leaves
  - nodes that don’t have children
- ancestors
- descendants
ancestors of u
Trees

descendants of u
Application of trees

- Applications of trees
  - class hierarchy in Java
  - file system
  - storing hierarchies in organizations
Tree ADT

- Whatever the implementation of a tree is, its interface is the following
  - root()
  - size()
  - isEmpty()
  - parent(v)
  - children(v)
  - isInternal(v)
  - isExternal(v)
  - isRoot()
class Tree {
    TreeNode root;

    // tree ADT methods..
}

class TreeNode<Type> {
    Type data;
    int size;
    TreeNode parent;
    TreeNode firstChild;
    TreeNode nextSibling;

    getParent();
    getChild();
    getNextSibling();
}
Depth:
- depth(T, v) is the number of ancestors of v, excluding v itself

Recursive formulation
- if v == root, then depth(v) = 0
- else, depth(v) is 1 + depth(parent(v))

Compute the depth of a node v in tree T: int depth(T, v)

Algorithm:
```java
int depth(T, v) {
    if T.isRoot(v) return 0
    return 1 + depth(T, T.parent(v))
}
```

Analysis:
- O(number of ancestors) = O(depth_v)
- in the worst case the path is a linked-list and v is the leaf
- ==> O(n), where n is the number of nodes in the tree
Height:
- Height of a node v in T is the length of the longest path from v to any leaf.

Recursive formulation:
- If v is leaf, then its height is 0.
- Else height(v) = 1 + maximum height of a child of v.

Definition: The height of a tree is the height of its root.

Compute the height of tree T: int height(T, v).

Height and depth are “symmetrical.”
Proposition: The height of a tree T is the maximum depth of one of its leaves.
Height

- **Algorithm:**

  ```c
  int height(T, v) {
    if T.isExternal(v) return 0;
    int h = 0;
    for each child w of v in T do
      h = max(h, height(T, w))
    return h+1;
  }
  ```

- **Analysis:**
  - **total time:** the sum of times spent at all nodes in all recursive calls
  - **the recursion:**
    - v calls height(w) recursively on all children w of v
    - height() will eventually be called on every descendant of v
    - overall: height() is called on each node precisely once, because each node has one parent
  - **aside from recursion**
    - for each node v: go through all children of v
      - O(1 + c_v) where c_v is the number of children of v
    - over all nodes: O(n) + SUM (c_v)
      - each node is child of only one node, so its processed once as a child
      - SUM(c_v) = n - 1
  - **total:** O(n), where n is the number of nodes in the tree
A traversal is a systematic way to visit all nodes of T.

- **pre-order:** root, children
  - parent comes before children; overall root first
- **post-order:** children, root
  - parent comes after children; overall root last

```c
void preorder(T, v)
    visit v
    for each child w of v in T do
        preorder(w)
```

```c
void postorder(T, v)
    for each child w of v in T do
        postorder(w)
    visit v
```

Analysis: $O(n)$ [same arguments as before]
Examples

- Tree associated with a document

```
Pape
Title     Abstract     Ch1     Ch2     Ch3     Refs
```

- In what order do you read the document?

```
1.1  1.2
3.1  3.2
```
Example

- Tree associated with an arithmetical expression

```
+  
\  
3 *  
\  
- +  
\  
12 5 1 7
```

- Write method that evaluates the expression. In what order do you traverse the tree?
Binary trees
Definition: A binary tree is a tree such that
- every node has at most 2 children
- each node is labeled as being either a left child or a right child

Recursive definition:
- a binary tree is empty;
- or it consists of
  - a node (the root) that stores an element
  - a binary tree, called the left subtree of T
  - a binary tree, called the right subtree of T

Binary tree interface
- left(v)
- right(v)
- hasLeft(v)
- hasRight(v)
- + isInternal(v), isExternal(v), isRoot(v), size(), isEmpty()
In a binary tree

- level 0 has \( \leq 1 \) node
- level 1 has \( \leq 2 \) nodes
- level 2 has \( \leq 4 \) nodes
- ...
- level \( i \) has \( \leq 2^i \) nodes

Proposition: Let \( T \) be a binary tree with \( n \) nodes and height \( h \). Then

- \( h+1 \leq n \leq 2^{h+1} - 1 \)

- \( \log(n+1) - 1 \leq h \leq n-1 \)
Binary tree implementation

- use a linked-list structure; each node points to its left and right children; the tree class stores the root node and the size of the tree

- implement the following functions:
  - left(v)
  - right(v)
  - hasLeft(v)
  - hasRight(v)
  - isInternal(v)
  - isExternal(v)
  - isRoot(v)
  - size()
  - isEmpty()
  - also
    - insertLeft(v,e)
    - insertRight(v,e)
    - remove(e)
    - addRoot(e)
Binary tree operations

- **insertLeft(v, e):**
  - create and return a new node w storing element e, add w as the left child of v
  - an error occurs if v already has a left child

- **insertRight(v, e)**

- **remove(v):**
  - remove node v, replace it with its child, if any, and return the element stored at v
  - an error occurs if v has 2 children

- **addRoot(e):**
  - create and return a new node r storing element e and make r the root of the tree;
  - an error occurs if the tree is not empty

- **attach(v, T1, T2):**
  - attach T1 and T2 respectively as the left and right subtrees of the external node v
  - an error occurs if v is not external
Performance

- all \( O(1) \)
  - left(\( v \))
  - right(\( v \))
  - hasLeft(\( v \))
  - hasRight(\( v \))
  - isInternal(\( v \))
  - isExternal(\( v \))
  - isRoot(\( v \))
  - size()
  - isEmpty()
  - addRoot(\( e \))
  - insertLeft(\( v,e \))
  - insertRight(\( v,e \))
  - remove(\( e \))
**Binary tree traversals**

- Binary tree computations often involve traversals
  - pre-order: root left right
  - post-order: left right root

- Additional traversal for binary trees
  - in-order: left root right
    - visit the nodes from left to right

- Exercise:
  - write methods to implement each traversal on binary trees
Application: Tree drawing

- Come up with a solution to "draw" a binary tree in the following way. Essentially, we need to assign coordinate x and y to each node.
  - node v in the tree
    - x(v) = ?
    - y(v) = ?
Application: Tree drawing

- We can use an in-order traversal and assign coordinate x and y of each node in the following way:
  - x(v) is the number of nodes visited before v in the in-order traversal of v
  - y(v) is the depth of v
Binary tree searching

- write search(v, k)
  - search for element k in the subtree rooted at v
  - return the node that contains k
  - return null if not found

- performance
  - ?
Binary Search Trees (BST)

**Motivation:**
- want a structure that can search fast
- arrays: search fast, updates slow
- linked lists: search slow, updates fast

**Intuition:**
- tree combines the advantages of arrays and linked lists

**Definition:**
- a BST is a binary tree with the following “search” property
  - for any node $v$
    - all nodes in $T_1 \leq k$
    - all node in $T_2 > k$
Example

BST

V

\[ T_1 \]
\[ \leq k \]

\[ T_2 \]
\[ > k \]
Sorting a BST

- Print the elements in the BST in sorted order
Sorting a BST

- Print the elements in the BST in sorted order.
- In-order traversal: left - node - right
- Analysis: $O(n)$

```java
// print the elements in tree of v in order
sort(BSTNode v)
if (v == null) return;
sort(v.left());
print v.getData();
sort(v.right());
```
Searching in a BST
Searching in a BST

//return the node w such that w.getData() == k or null if such a node
//does not exist
BSTNode search (v, k)   {
    if (v == null) return null;
    if (v.getData() == k) return v;
    if (k < v.getData()) return search(v.left(), k);
    else return search(v.right(), k)
}

- Analysis:
  - search traverses (only) a path down from the root
  - does NOT traverse the entire tree
  - \( O(\text{depth of result node}) = O(h) \), where \( h \) is the height of the tree
Inserting in a BST

- insert 25
Inserting in a BST

- insert 25
  - There is only one place where 25 can go

- //create and insert node with key k in the right place
  - void insert (v, k) {
    //this can only happen if inserting in an empty tree
    if (v == null) return new BSTNode(k);
    if (k <= v.getData()) {
      if (v.left() == null) {
        //insert node as left child of v
        u = new BSTNode(k);
        v.setLeft(u);
      } else {
        return insert(v.left(), k);
      }
    } else {//if (v.getData() > k) {
      ...
    }
  }
Inserting in a BST

**Analysis:**

- similar with searching
- traverses a path from the root to the inserted node
- $O($depth of inserted node$)$
- this is $O(h)$, where $h$ is the height of the tree
Deleting in a BST

- delete 87
- delete 21
- delete 90

**case 1: delete a leaf**
- if x is left of its parent, set parent(x).left = null
- else set parent(x).right = null

**case 2: delete a node with one child**
- link parent(x) to the child of x

**case 2: delete a node with 2 children**
- ??
Deleting in a BST

- delete 90

- copy in u 94 and delete 94
  - the left-most child of right(x)
- or
- copy in u 87 and delete 87
  - the right-most child of left(x)

![BST Diagram]

- node has <=1 child
- node has <=1 child
Deleting in a BST

- **Analysis:**
  - traverses a path from the root to the deleted node
  - and sometimes from the deleted node to its left-most child
  - this is $O(h)$, where $h$ is the height of the tree
BST performance

- Because of search property, all operations follow one root-leaf path
  - insert: $O(h)$
  - delete: $O(h)$
  - search: $O(h)$

- We know that in a tree of $n$ nodes
  - $h \geq \lg (n+1) - 1$
  - $h \leq n - 1$

- So in the worst case $h$ is $O(n)$
  - BST insert, search, delete: $O(n)$
  - just like linked lists/arrays
BST performance

- worst-case scenario
  - start with an empty tree
  - insert 1
  - insert 2
  - insert 3
  - insert 4
  - ...
  - insert n

- it is possible to maintain that the height of the tree is $\Theta(\lg n)$ at all times
  - by adding additional constraints
  - perform rotations during insert and delete to maintain these constraints

- Balanced BSTs: $h$ is $\Theta(\lg n)$
  - Red-Black trees
  - AVL trees
  - 2-3-4 trees
  - B-trees

- to find out more.... take csci231 (Algorithms)