Trees & Heaps
Week 12
Gaddis: 20
Weiss: 21.1-3
CS 5301
Fall 2018
Jill Seaman

Tree: non-recursive definition

- **Tree**: set of nodes and directed edges
  - **root**: one node is distinguished as the root
  - Every node (except root) has exactly one edge coming into it.
  - Every node can have any number of edges going out of it (zero or more).

- **Parent**: source node of directed edge
- **Child**: terminal node of directed edge
- **Binary Tree**: a tree in which no node can have more than two children.

Tree Traversals: examples

- **Preorder**: print node value, process left tree, then right
  

- **Postorder**: process left tree, then right, then print node value

- **Inorder**: process left tree, print node value, then process right tree

Binary Trees: implementation

- Structure with a data value, and a pointer to the left subtree and another to the right subtree.

  ```
  struct TreeNode {
    int value;       // the data
    TreeNode *left;  // left subtree
    TreeNode *right; // right subtree
  };
  ```

- Like a linked list, but two “next” pointers.
- There is also a special pointer to the root node of the tree (like head for a list).
Binary Search Trees

- A special kind of binary tree, used for efficient searching, insertion, and deletion.

- **Binary Search Tree property:**
  - For every node X in the tree:
    - All the values in the **left** subtree are **smaller** than the value at X.
    - All the values in the **right** subtree are **larger** than the value at X.

- Not all binary trees are binary search trees
- An inorder traversal of a BST shows the values in sorted order

Binary Search Trees: operations

- insert(x)
- remove(x) (or delete)
- isEmpty() (returns bool)
- find(x) (or search, returns bool)
- findMin() (returns <type>)
  - Smallest element is found by always taking the left branch.
- findMax() (returns <type>)
  - Largest element is found by always taking the right branch.

BST: find(x)

Algorithm:
- If we are searching for 15 we are done.
- If we are searching for a key < 15, then we should search in the left subtree.
- If we are searching for a key > 15, then we should search in the right subtree.

```
bool IntBinaryTree::searchNode(int num)
{
    TreeNode *p = root;
    while (p)
    {
        if (p->value == num)
            return true;
        else if (num < p->value)
            p = p->left;
        else
            p = p->right;
    }
    return false;
}
```

- Can also be defined recursively
**BST: insert(x)**

- Algorithm is similar to find(x)
- If x is found, do nothing (no duplicates in tree)
- If x is not found, add a new node with x in place of the last empty subtree that was searched.

Inserting 13:

```
void IntBinaryTree::insert(TreeNode *&nodePtr, int num) {
    if (nodePtr == NULL) {
        // Create a new node and store num in it,
        // making nodePtr point to it
        nodePtr = new TreeNode;
        nodePtr->value = num;
        nodePtr->left = nodePtr->right = NULL;
    } else if (num < nodePtr->value)
        insert(nodePtr->left, num);     // Search the left branch
    else if (num > nodePtr->value)
        insert(nodePtr->right, num);    // Search the right branch
    // else nodePt->value == num, do nothing, no duplicates
}
```

**BST: remove(x)**

- Algorithm starts with finding(x)
- If x is not found, do nothing
- If x is found, remove node carefully.
  - Must remain a binary search tree (smallers on left, biggers on right).
  - The algorithm is described here in the lecture, the code is in the book (and on class website in BinaryTree.zip) in the makeDeletion(TreeNode * &nodePtr) function.

**Case 1: Node has no right child (or no children)**
- Make parent pointer bypass the Node and point to the left child

**Case 2: Node has no left child**
- Make parent pointer bypass the Node and point to the right child

Does not matter if the child is the left or right child of deleted node

**Figure 4.24**: Deletion of a node (4) with one child, before and after
**BST: remove(x)**

- **Case 3: Node has 2 children**
  - Find minimum node in right subtree—this node cannot have left subtree, or it’s not the minimum
  - Move original node’s left subtree to be the left subtree of this node.
  - Make original node’s parent pointer bypass the original node and point to right subtree

**Binary heap data structure**

- A binary heap is a special kind of binary tree
  - has a restricted structure (must be complete)
  - has an ordering property (parent value is smaller than child values)
  - NOT a Binary Search Tree!
- Used in the following applications
  - Priority queue implementation: supports enqueue and deleteMin operations in $O(\log N)$
  - Heap sort: another $O(N \log N)$ sorting algorithm.

**Binary Heap: structure property**

- **Complete binary tree**: a tree that is completely filled
  - every level except the last is completely filled.
  - the bottom level is filled left to right (the leaves are as far left as possible).

**Complete Binary Trees**

- A complete binary tree can be easily stored in an array
  - place the root in position 1 (for convenience)
Complete Binary Trees

Properties

- In the array representation:
  - put root at location 1
  - use an int variable (size) to store number of nodes
  - for a node at position i:
    - left child at position \(2i\) (if \(2i \leq \text{size}\), else i is leaf)
    - right child at position \(2i+1\) (if \(2i+1 \leq \text{size}\), else i is leaf)
    - parent is in position \(\text{floor}(i/2)\) (or use integer division)

- There is a heap implementation on the class website in Heap.zip

Binary Heap:

ordering property

- In a heap, if X is a parent of Y, value(X) is less than or equal to value(Y).
  - the minimum value of the heap is always at the root.

Heap: insert(x)

- First: add a node to tree.
  - must be placed at next available location, size+1, in order to maintain a complete tree.
- Next: maintain the ordering property:
  - if x doesn’t have a parent: done
  - if x is greater than its parent: done
  - else swap with parent, repeat
- Called “percolate up” or “reheap up”
- preserves ordering property
Heap: deleteMin()

- Minimum is at the root, removing it leaves a hole.
  - The last element in the tree must be relocated.
- First: move last element up to the root
- Next: maintain the ordering property, start with root:
  - if no children, do nothing.
  - if one child, swap with parent if it's smaller than the parent.
  - if both children are greater than the parent: done
  - otherwise, swap the smaller of the two children with the parent, and repeat on that child.
- Called "percolate down" or "reheap down"
- preserves ordering property