Trees & Heaps
Week 12
Gaddis: 20
Weiss: 21.1-3
CS 5301
Spring 2016
Jill Seaman

Tree:
- **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
  
  \[ +\ a\ b\ c \ *\ *\ d\ e\ f\ g \]

- **Postorder**: process left tree, then right, then print node value
  
  \[ a\ b\ c\ *\ d\ e\ f\ g\ *\ g \]

- **Inorder**: process left tree, print node value, then process right tree
  
  \[ a\ b\ *\ c\ +\ d\ *\ e\ +\ f\ *\ g \]

Binary Trees: implementation

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

```c
struct TreeNode {
    <type> data;     // 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).

```c
TreeNode *root;
```
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)
  - if the root is NULL
- find(x)    (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)

Recursive 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.

```c
bool find (<type> x, TreeNode t) {
  if (isEmpty(t))
    return false
  if (x < value(t))
    return find (x, left(t))
  if (x > value(t))
    return find (x, right(t))
  return true  // x == value(t)
}
```

Base case
**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:

**Pseudocode**

Recursive

```cpp
void insert (<type> x, TreeNode t) {
    if (isEmpty(t))
        make t’s parent point to new TreeNode(x)
    else if (x < value(t))
        insert (x, left(t))
    else if (x > value(t))
        insert (x, right(t))
    //else x == value(t), do nothing, no duplicates
}
```

**Linked List example:**

- Append x to the end of a singly linked list:
  - Pass the node pointer by reference
  - Recursive

```cpp
void List::append (double x) {
    append(x, head);
}
void List::append (double x, Node *& p) {
    if (p == NULL) {
        p = new Node();
        p->data = x;
        p->next = NULL;
    } else
        append (x, p->next);
}
```

**BST: remove(x)**

- Algorithm is 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, bigger on right).
**BST: remove(x)**

- **Case 1: Node is a leaf**
  - Can be removed without violating BST property
- **Case 2: Node has one child**
  - Make parent pointer bypass the Node and point to that child

**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.

**BST: remove(x)**

- **Case 3: Node has 2 children**
  - Find minimum node in right subtree
    -- 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: 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)

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.

Complete Binary Trees

- 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 \(2i \leq\) size, else i is leaf)
    - right child at position \(2i+1\) (if \(2i+1 \leq\) size, else i is leaf)
    - parent is in position \(\text{floor}(i/2)\) (or use integer division)

Binary 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 is greater than its parent: done
  - else swap with parent, repeat
- Called “percolate up” or “reheap up”
- preserves ordering property
**Heap: insert(x)**

- 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

**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