trees and binary search trees

chapters 18 and 19

CS 3358
Spring 2015

Jill Seaman

sections 18.1-4, 19.1-3

Dynamic data structures

- Linked Lists
  - dynamic structure, grows and shrinks with data
  - most operations are linear time (O(N)).
- Can we make a simple data structure that can do better?
- Trees
  - dynamic structure, grows and shrinks with data
  - most operations are logarithmic time (O(log N)).

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 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

Tree: example

edges are directed down (source is higher)
D is the parent of H. Q is a child of J.
**Leaf**: a node with no children (like H and P)
**Sibling**: nodes with same parent (like K,L,M)
**Tree: recursive definition**

- **Tree:**
  - is empty or
  - consists of a root node and zero or more nonempty subtrees, with an edge from the root to each subtree.

**Example: Expression Trees**

- more generally: syntax trees
  - leaves are operands
  - internal nodes are operators
  - can represent entire program as a tree

**Tree traversal**

- Tree traversal: operation that converts the values in a tree into a list
  - Often the list is output
- Pre-order traversal
  - Print the data from the root node
  - Do a pre-order traversal on first subtree
  - Do a pre-order traversal on second subtree
  - Do a preorder traversal on last subtree

**Preorder traversal: Expression Tree**

- print node value, process left tree, then right
- prefix notation (for arithmetic expressions)
Postorder traversal:
Expression Tree

- process left tree, then right, then node
  \[ a \cdot b + c + (d \cdot e + f) \cdot g \]
- postfix notation (for arithmetic expressions)

Inorder traversal:
Expression Tree

- if each node has 0 to 2 children, you can do inorder traversal
- process left tree, print node value, then process right tree
  \[ a + b \cdot c + d \cdot e + f \cdot g \]
- infix notation (for arithmetic expressions)

Binary Trees

- **Binary Tree**: a tree in which no node can have more than two children.

  - height: shortest: \( \log_2(n) \) tallest: \( n \)

  \[
  \text{struct TreeNode} \\
  \hspace{1cm} \{ \\
  \hspace{2cm} \text{Object data; } // \text{ the data} \\
  \hspace{2cm} \text{TreeNode } *\text{left; } // \text{ left subtree} \\
  \hspace{2cm} \text{TreeNode } *\text{right; } // \text{ right subtree} \\
  \hspace{1cm} \} \\
  \]

  - Like a linked list, but two “next” pointers.
  - This structure can be used to represent any binary tree.
Binary Search Trees

- A special kind of binary tree
- A data structure 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

The same set of values may have multiple valid BSTs

- Maximum depth of a node: N
- Average depth of a node: $O(\log_2 N)$
Binary Search Trees: operations

- insert(x)
- remove(x) (or delete)
- isEmpty() (returns bool)
- makeEmpty()
- find(x) (returns bool)
- findMin() (returns ItemType)
- findMax() (returns ItemType)

BST: find(x)

Example: search for 9
- compare 9 to 15, go left
- compare 9 to 6, go right
- compare 9 to 7 go right
- compare 9 to 13 go left
- compare 9 to 9: found

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.

```
bool find (ItemType 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)
}
```
**BST: findMin()**

- Smallest element is found by always taking the left branch.
- Pseudocode
- Recursive
- Tree must not be empty

```cpp
ItemType findMin (TreeNode t) {
    assert (!isEmpty(t))
    if (isEmpty(left(t)))
        return value(t)
    return findMin (left(t))
}
```

**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.

```
BST: insert(x)
```

```cpp
bool insert (ItemType 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

```
Linked List example:
```

```cpp
void List<T>::append (T x) {
    append(x, head);
}
```

```cpp
void List<T>::append (T 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 not found, remove node carefully.
  - Must remain a binary search tree (smallers on left, biggers 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 child

![](image)

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

- Case 3: Node has 2 children
  - Replace it with the minimum value in the right subtree
  - Remove minimum in right:
    - will be a leaf (case 1), or have only a right subtree (case 2)
      --cannot have left subtree, or it’s not the minimum

remove(2): replace it with the minimum of its right subtree (3) and delete that node.

![Diagram](image)

Figure 4.25: Deletion of a node (2) with two children, before and after

removeMin

```cpp
template<class ItemType>
void BST<ItemType>::removeMin(TreeNode*& t)
{
    assert (t);   //t must not be empty
    if (t->left) {
        removeMin(t->left);
    }
    else {
        TreeNode *temp = t;
        t = t->right;   //it’s ok if this is null
        delete temp;
    }
}
```

Note: t is a pointer passed by reference
template<class ItemType>
void BST<ItemType>::deleteItem(TreeNode*& t, const ItemType& newItem) {
    if (t == NULL) return;          // not found
    else if (newItem < t->data)      // search left
        deleteItem(t->left, newItem);
    else if (newItem > t->data)      // search right
        deleteItem(t->right, newItem);
    else { // newItem == t->data: remove t
        if (t->left && t->right) {   // two children
            t->data = findMin(t->right);
            removeMin(t->right);
        } else {                     // one or zero children: skip over t
            TreeNode *temp = t;
            if (t->left) t = t->left;
            else t = t->right;        //ok if this is null
            delete temp;
        }
    }
}