Trees and Binary Search Trees

Chapters 18 and 19

CS 3358
Summer I 2012

Jill Seaman

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

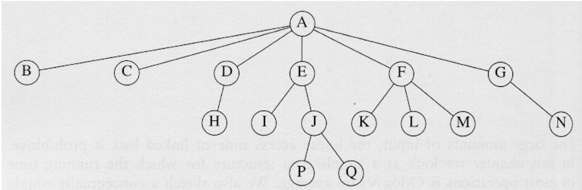


Figure 4.2 A tree

- 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.

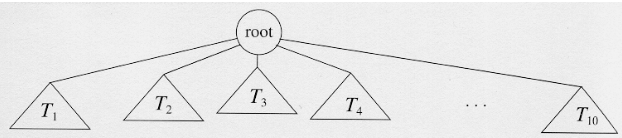


Figure 4.1 Generic tree

Tree terms

- **Path**: sequence of (directed) edges
- **Length of path**: number of edges on the path
- **Depth of a node**: length of path from root to that node.
- **Height of a node:** length of longest path from node to a leaf.
  - height of tree = height of root, depth of deepest leaf
  - leaves have height 0
  - root has depth 0

Example: Unix directory

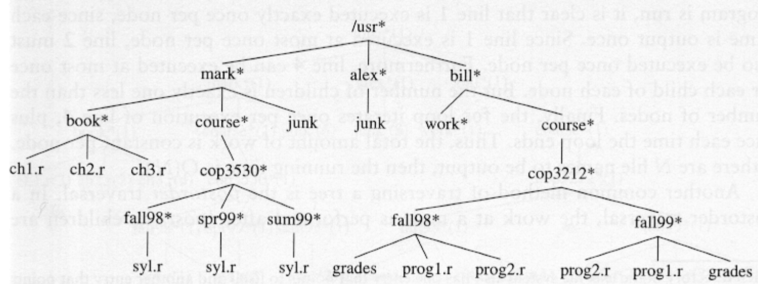


Figure 4.5 UNIX directory

Example: Expression Trees
more generally: syntax trees

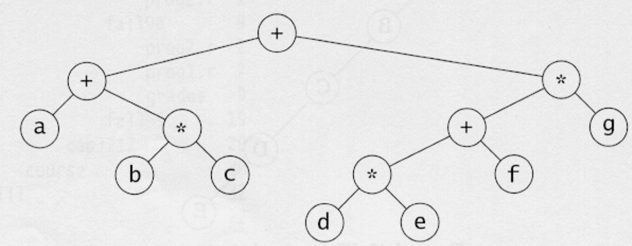


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- 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

This is recursive. What's the base case?

Preorder traversal: Expression Tree

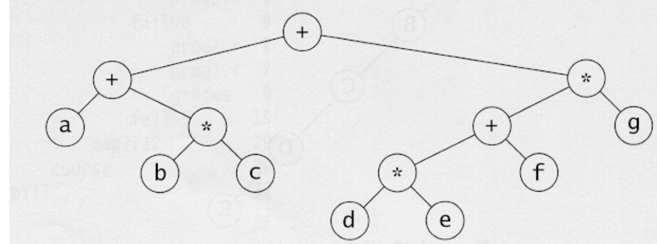


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- print node value, process left tree, then right

  `+ + a * b c * + * d e f g`

- prefix notation

Postorder traversal: Expression Tree

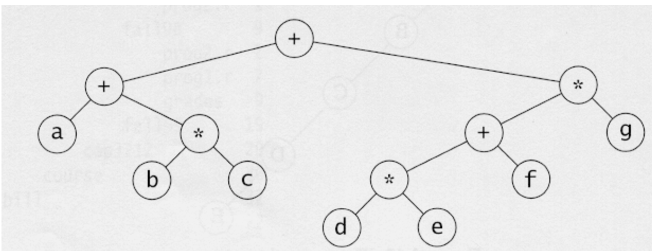


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- process left tree, then right, then node

  `a b c * + d e * f + g *`

- postfix notation

Inorder traversal: Expression Tree

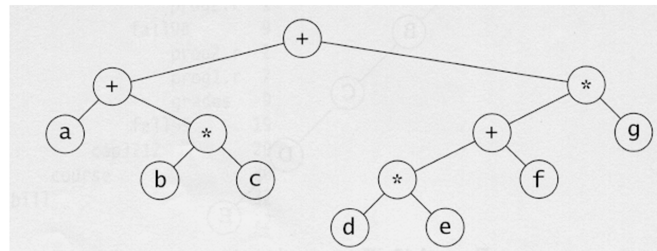


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- 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 * c + d * e + f * g

- infix notation

Example: Unix directory traversal

Preorder

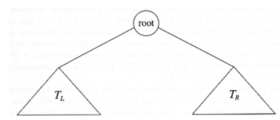
```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall98
                    syl.r
                spr99
                    syl.r
                sum99
                    syl.r
        junk
    alex
        junk
    bill
        work
        course
            cop3212
                fall98
                    grades
                    prog1.r
                    prog2.r
                fall99
                    prog2.r
                    prog1.r
                    grades
```

Postorder

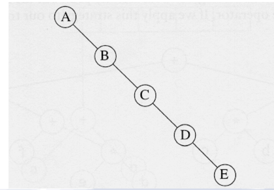
```
                ch1.r            3
                ch2.r            2
                ch3.r            4
            book                10
                        syl.r    1
                    fall98       2
                        syl.r    5
                    spr99        6
                        syl.r    2
                    sum99        3
                cop3530         12
            course              13
            junk                 6
        mark                    30
            junk                 8
        alex                     9
            work                 1
                        grades   3
                        prog1.r  4
                        prog2.r  1
                    fall98       9
                        prog2.r  2
                        prog1.r  7
                        grades   9
                    fall99      19
                cop3212         29
            course              30
        bill                    32
/usr                            72
```


Binary Trees

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



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



n is the number of nodes in the tree.

Binary Trees: implementation

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

```
struct TreeNode {
  Object data;        // the data
  BinaryNode *left;   // left subtree
  BinaryNode *right;  // right subtree
};
```

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

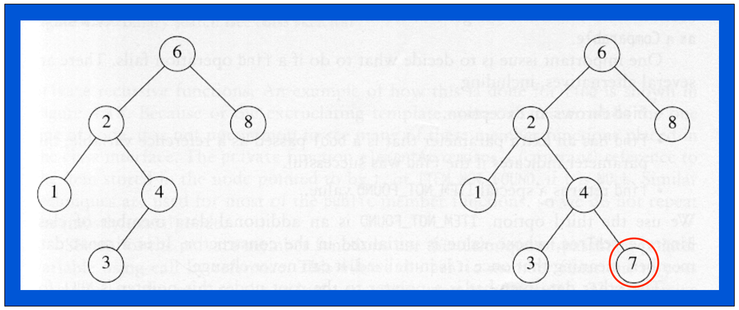
Binary Search Trees

- 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

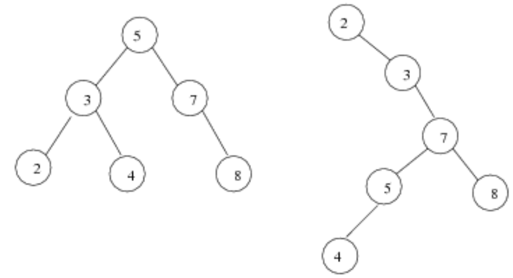
Binary Search Trees



A binary search tree | Not a binary search tree

Binary Search Trees

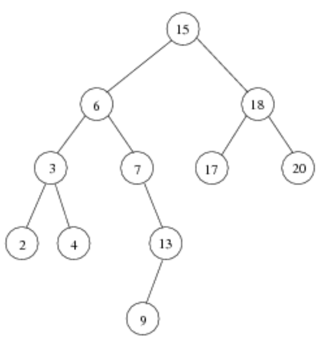
The same set of values may have multiple valid BSTs



- Maximum depth of a node: N
- Average depth of a node: $O(\log_2 N)$

Binary Search Trees

An inorder traversal of a BST shows the values in sorted order



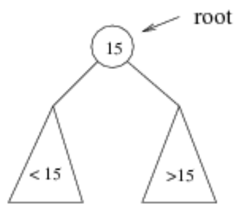
Inorder traversal: 2 3 4 6 7 9 13 15 17 18 20

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)



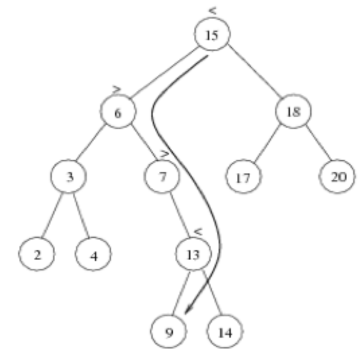
Recursive Algorithm:

- 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.

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



BST: find(x)

- Pseudocode
- Recursive

```
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)

}
```

Base case

BST: findMin()

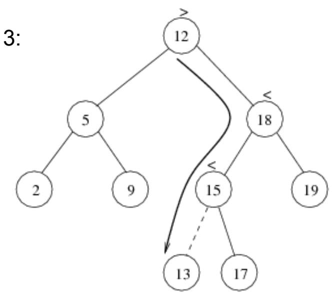
- Smallest element is found by always taking the left branch.
- Pseudocode
- Recursive
- Tree must not be empty

```
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.

Inserting 13:



BST: insert(x)

- Pseudocode
- Recursive

```
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:

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

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

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);
}
```

Public function

Private recursive function

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

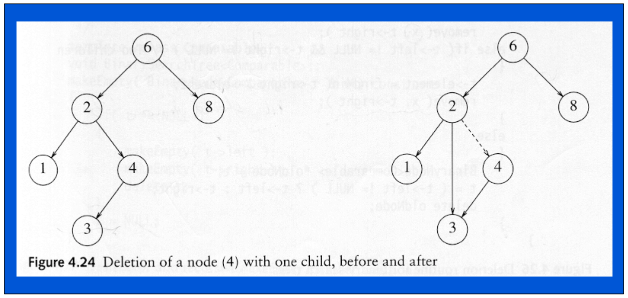


Figure 4.24 Deletion of a node (4) with one child, before and after

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

BST: remove(x)

- 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

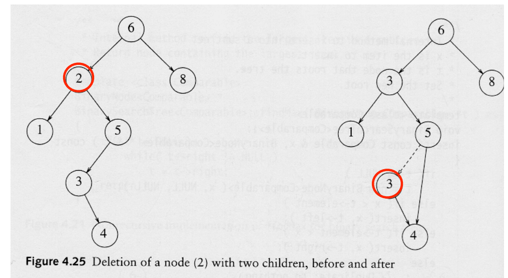


Figure 4.25 Deletion of a node (2) with two children, before and after

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

BST: remove(x)
removeMin

```
template<class ItemType>
void BST_3358 <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

BST: remove(x)
deleteItem

```
template<class ItemType>
void BST_3358 <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;
        }
    }
}
```

Note: t is a pointer passed by reference

Binary Search Trees:
runtime analyses

- Cost of each operation is proportional to the number of nodes accessed
- depth of the node (height of the tree)
- best case: O(log N)	(balanced tree)
- worst case: O(N)	(tree is a list)
- average case: ??
  - Theorem: on average, the depth of a binary search tree node, assuming random insertion sequences, is 1.38 log N