Trees

Week 12

Gaddis: 20
Weiss: 21.1-3

CS 5301
Spring 2014

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 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

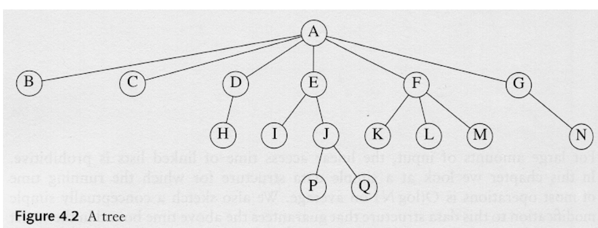


Figure 4.2 A tree

- 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 (a subtree is a Tree).

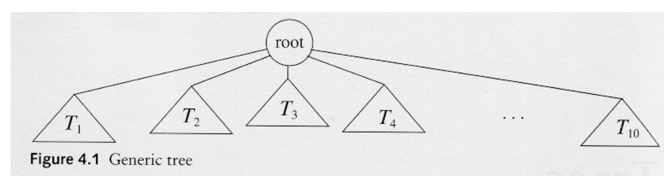


Figure 4.1 Generic tree

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.

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

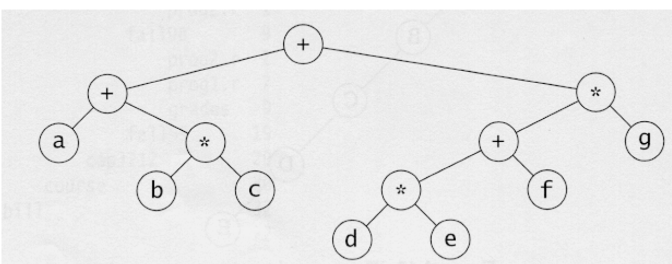


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- print node value, process left tree, then right

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

- prefix notation (for arithmetic expressions)

Postorder traversal:
Expression Tree

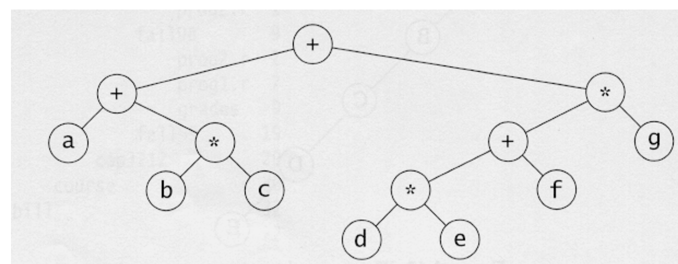


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- process left tree, then right, then node

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

- postfix notation (for arithmetic expressions)

Inorder traversal:
Expression Tree

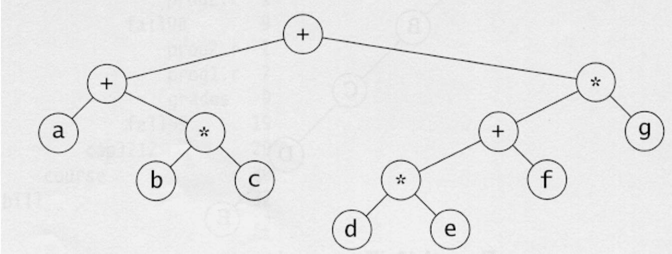


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

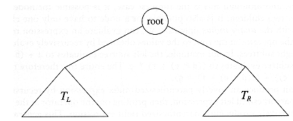
- IF each node has 0 to 2 children, you can do inorder traversal
- process left tree, print node value, then process right tree

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

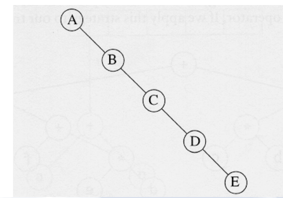
- infix notation (for arithmetic expressions)

Binary Trees

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



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



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 {
  <type> data;        // the data
  TreeNode *left;   // left subtree
  TreeNode *right; // right subtree
};
```

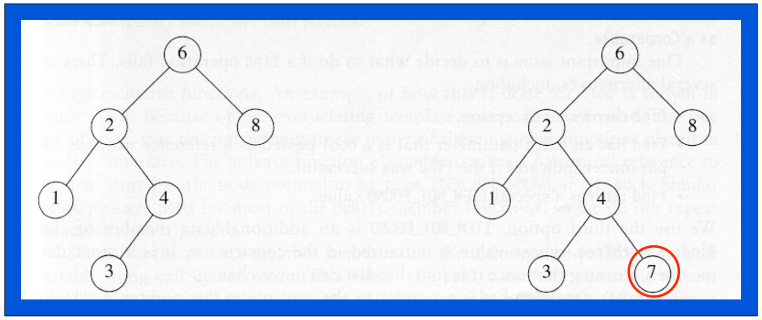
- Like a linked list, but two “next” pointers.
- This structure can be used to represent any binary tree.

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

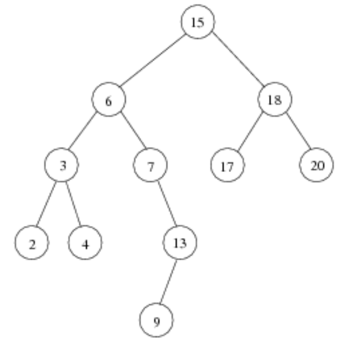


A binary search tree

Not a binary search tree

Binary Search Trees

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



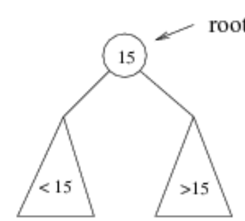
Inorder traversal: 2 3 4 6 7 9 13 15 17 18 20

Binary Search Trees: operations

- insert(x)
- remove(x) (or delete)
- isEmpty() (returns bool)

- find(x) (returns bool)
- findMin() (returns <type>)
- findMax() (returns <type>)

BST: find(x)



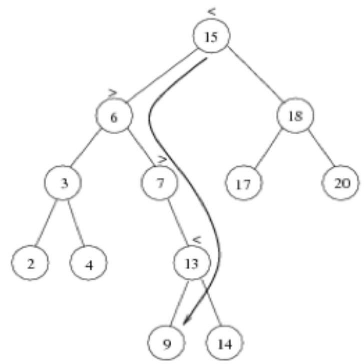
Recursive Algorithm:

- 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 (<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: findMin()

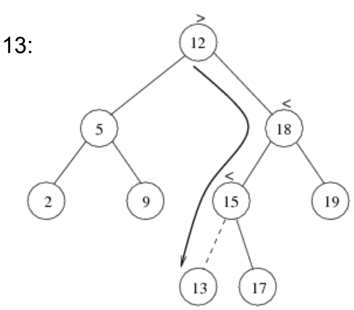
- Smallest element is found by always taking the left branch.
- Pseudocode
- Recursive
- Tree must not be empty

```
<type> 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 (<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

```
void List::append (double x) {        Public function
   append(x, head);
}

void List::append (double x, Node *& p) {
                                          Private recursive function
   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

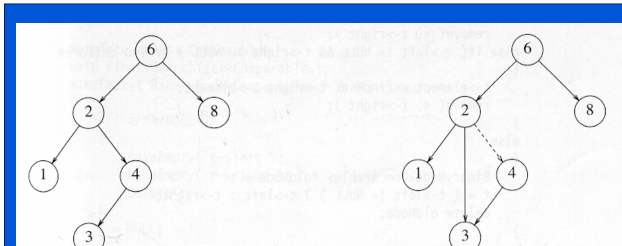


Figure 4.24 Deletion of a node (4) with one child, before and after

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

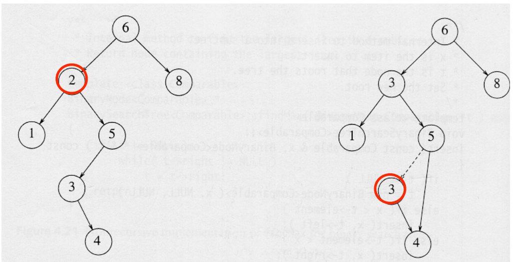


Figure 4.25 Deletion of a node (2) with two children, before and after

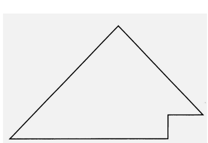
remove(2): replace it with the minimum of its right subtree (3) and delete that node.

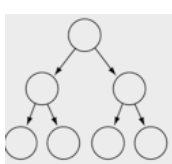
Binary heap data structure

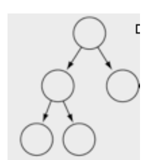
- A binary heap is a special kind of binary tree
  - has a restricted structure (must be complete)
  - has an ordering property (parent value is smaller than child values)
  - NOT a Binary Search Tree!
- Used in the following applications
  - Priority queue implementation: supports enqueue and deleteMin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

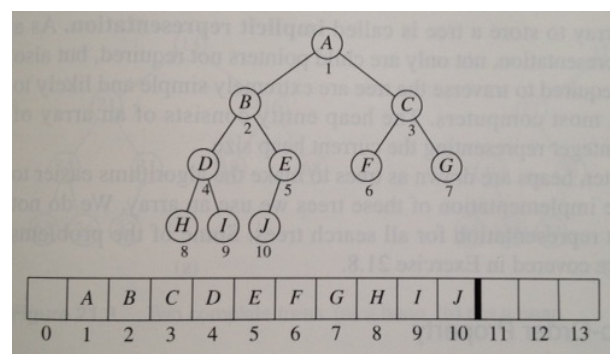






Complete Binary Trees

- 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 <= size, else i is leaf)
    - right child at position `2i+1` (if 2i+1 <= size, else i is leaf)
    - parent is in position `floor(i/2)` (or use integer division)

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.

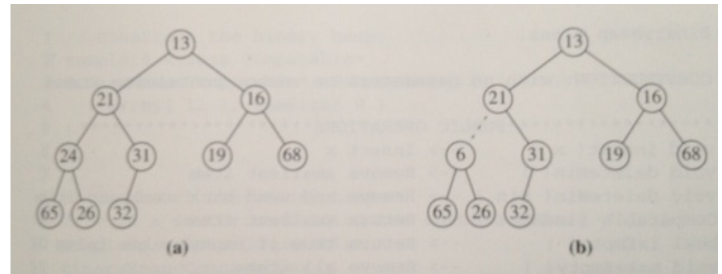


Figure 21.3 Two complete trees: (a) a heap; (b) not a heap.

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)

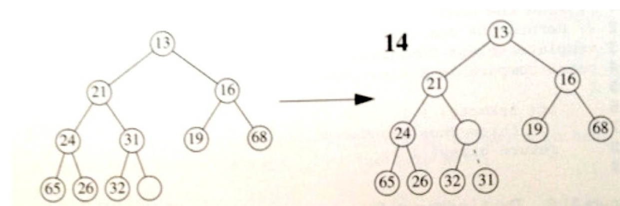


Figure 21.7 Attempt to insert 14, creating the hole and bubbling the hole up.

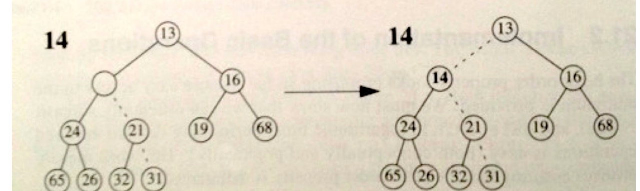


Figure 21.8 The remaining two steps required to insert 14 in the original heap shown in Figure 21.7.

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 both children are greater than the parent: done
  - otherwise, swap the smaller of the two children with the parent, repeat
- Called "percolate down" or "reheap down"
- preserves ordering property
- O(log n)

Heap: deleteMin()

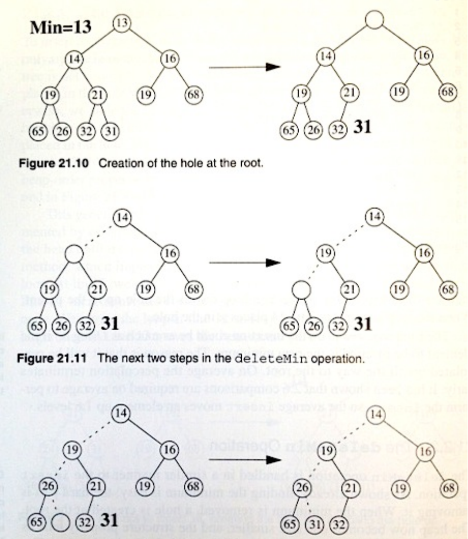


Figure 21.10 Creation of the hole at the root.

Figure 21.11 The next two steps in the deleteMin operation.

Figure 21.12 The Last two steps in the deleteMin operation.