Binary Search Trees & Trees in General

02-201 / 02-601
Dictionary Abstract Data Type (ADT)

• Most basic and most useful ADT:
  • insert(key, value)
  • delete(key)
  • value = find(key)

• Many languages have it built in like Go’s map:
  - awk: D[“AAPL”] = 130
  - perl: my %D; $D[“AAPL”] = 130;
  - python: D = {}; D[“AAPL”] = 130
  - C++: map<string,string> D = new map<string, string>(); D[“AAPL”] = 130;

• Insert, delete, find each either ≈ log n steps [C++] or expected constant # of steps [perl, python]

• How can such dictionaries are implemented? — There are a number of ways; we’ll see one next.
Trees
Hierarchies

Many ways to represent tree-like information:

- linked hierarchy
- outlines, indentations
- nested, labeled parenthesis
- nested sets

Definition – Rooted Tree

- **nil** is a tree
- If \( T_1, T_2, ..., T_k \) are trees with roots \( r_1, r_2, ..., r_k \) and \( r \) is a node \( \notin \) any \( T_i \), then the structure that consists of the \( T_i \), node \( r \), and edges \((r, r_i)\) is also a tree.
Terminology

- $r$ is the *parent* of its *children* $r_1, r_2, \ldots, r_k$.

- $r_1, r_2, \ldots, r_k$ are *siblings*.

- *root* = distinguished node, usually drawn at top. Has no parent.

- If all children of a node are *nil*, the node is a *leaf*. Otherwise, the node is a *internal node*.

- A *path* in the tree is a sequence of nodes $u_1, u_2, \ldots, u_m$ such that each of the edges $(u, u_{i+1})$ exists.

- A node $u$ is an *ancestor* of $v$ if there is a path from $u$ to $v$.

- A node $u$ is a *descendant* of $v$ if there is a path from $v$ to $u$. 
Height & Depth

- The **height** of node \( u \) is the length of the longest path from \( u \) to a leaf.

- The **depth** of node \( u \) is the length of the path from the root to \( u \).

- Height of the tree = maximum depth of its nodes.

- A **level** is the set of all nodes at the same depth.

```
Depth = 0
    3

Depth = 1
    1   2

Depth = 2
    0   0   0   1

Depth = 3
    0   0   0   0
```

Numbers in nodes give heights
Subtrees, forests, and graphs

- A **subtree** rooted at $u$ is the tree formed from $u$ and all its descendants.

- A **forest** is a (possibly empty) set of trees. The set of subtrees rooted at the children of $r$ form a forest.

- As we’ve defined them, trees are **not** a special case of graphs:
  - Our trees are **oriented** (there is a root which implicitly defines directions on the edges).
  - A **free tree** is a connected graph with no cycles.
Alternative Definition – Rooted Tree

- A tree is a finite set $T$ such that:
  - one element $r \in T$ is designated the root.
  - the remaining nodes are partitioned into $k \geq 0$ disjoint sets $T_1, T_2, \ldots, T_k$, each of which is a tree.

This definition emphasizes the partitioning aspect of trees:

As we move down the we’re dividing the set of elements into more and more parts.

Each part has a distinguished element (that can represent it).
Binary Search Trees
Binary Search Trees (BST)

- **BST Property:** If a node has key \( k \) then keys in the left subtree are \( < k \) and keys in the right subtree are \( > k \).

- For convenience, we disallow duplicate keys.

- Good for implementing the dictionary ADT we’ve already seen: insert, delete, find.
BST Find

Find $k = 6$:
Find $k = 6$:

Is $k < 5$?
BST Find

Find $k = 6$:

Is $k < 5$?  No, go right
Find $k = 6$:

Is $k < 5$?  No, go right

Is $k < 8$?
BST Find

Find $k = 6$:

Is $k < 5$?    No, go right

Is $k < 8$?    Yes, go left
Find $k = 9$: 

```
    5
   / \
  3   8
 / \ / \ 
2  4 6  11
   \
    9
```
BST Find

Find \( k = 9 \):

Is \( k < 5 \)?
BST Find

Find \( k = 9 \):

Is \( k < 5 \)?  No, go right
Find $k = 9$:

Is $k < 5$?  No, go right

Is $k < 8$?
BST Find

Find $k = 9$:

Is $k < 5$?  No, go right

Is $k < 8$?  No, go right
Find $k = 9$:

- Is $k < 5$? \textbf{No, go right}
- Is $k < 8$? \textbf{No, go right}
- Is $k < 11$?
Find $k = 9$:

- Is $k < 5$? No, go right
- Is $k < 8$? No, go right
- Is $k < 11$? Yes, go left
Find $k = 13$:
Find $k = 13$:

Is $k < 5$?
Find $k = 13$:

Is $k < 5$?  No, go right
Find $k = 13$:

Is $k < 5$?  **No, go right**

Is $k < 8$?
Find $k = 13$:

Is $k < 5$?  No, go right

Is $k < 8$?  No, go right
Find $k = 13$:

- Is $k < 5$?  No, go right
- Is $k < 8$?  No, go right
- Is $k < 11$?
Find $k = 13$:

- Is $k < 5$? No, go right
- Is $k < 8$? No, go right
- Is $k < 11$? No, go right

BST Find
insert(T, K):
    q = NULL
    p = T
    while p != nil and p.key != K:
        q = p
        if p.key < K:
            p = p.right
        else if p.key > K:
            p = p.left

    if p != nil: error DUPLICATE

N = new Node(K)
if q.key > K:
    q.left = N
else:
    q.right = N

Same idea as BST Find
**BST Insert**

```python
insert(T, K):
    q = NULL
    p = T
    while p != nil and p.key != K:
        q = p
        if p.key < K:
            p = p.right
        else:
            if p.key > K:
                p = p.left
    if p != nil: error DUPLICATE

N = new Node(K)
if q.key > K:
    q.left = N
else:
    q.right = N
```

*Same idea as BST Find*
BST Insert

insert(T, K):
    q = NULL
    p = T
    while p != nil and p.key != K:
        q = p
        if p.key < K:
            p = p.right
        else if p.key > K:
            p = p.left
    if p != nil: error DUPLICATE

N = new Node(K)
if q.key > K:
    q.left = N
else:
    q.right = N

Same idea as BST Find
insert(T, K):
  q = NULL
  p = T
  while p != nil and p.key != K:
    q = p
    if p.key < K:
      p = p.right
    else:
      if p.key > K:
        p = p.left
  if p != nil: error DUPLICATE

N = new Node(K)
if q.key > K:
  q.left = N
else:
  q.right = N
BST FindMin

```
  5
 /   \
3     8
 /   /   \
2   4   6   11
 /   \
1     
  
  5
 /   \
4     8
 /   /   \
1   3   6   11
```
BST FindMin

Walk left until you can’t go left any more
Walk left until you can’t go left any more
BST Delete

Node is a leaf:

Node has 1 child:

Node has 2 children:
BST Delete

Node is a leaf:

Node has 1 child:

Node has 2 children:
Node is a leaf:

Node has 1 child:

Node has 2 children:
BST Operations Summary

- **Find**: walk left or right according to the key comparison.

- **Insert**: Put the new node where a Find for it would have fallen off the tree.

- **Delete**:
  - If deleting a leaf, just remove it.
  - If deleting a node u with 1 child, move that child up to be a child of u’s parent.
  - If deleting a node u with 2 children: find the smallest key in the subtree rooted at u, delete it, and replace u with that key.
• What’s the worst possible insertion order?

• What’s the best possible insertion order?
Binary Tree Representation
**BST Find Code**

```go
type BSTNode struct {
    key int
    left, right *BSTNode
}
```

A node contains the data (here key) plus pointers to the left and right children.

Recursive implementation:

```go
func BSTFind(root *BSTNode, k int) *BSTNode {
    if root != nil {
        if k == root.key {
            return root
        }
        if k < root.key {
            return BSTFind(root.left, k)
        }
        if k > root.key {
            return BSTFind(root.right, k)
        }
    }
    return nil
}
```

How much memory is used?
BST Find: Non-recursive

We update “root” so that it points to the current node:
Also extended so that this returns both the node and it’s parent

```go
func BSTFind(root *BSTNode, k int) (*BSTNode, *BSTNode) {
    var parent *BSTNode = nil
    for root != nil {
        if k == root.key {
            return parent, root
        }
        parent = root
        if k < root.key {
            root = root.left
        } else if k > root.key {
            root = root.right
        }
    }
    return parent, root
}
```
func BSTInsert(root *BSTNode, k int) (*BSTNode, bool) {
    newNode := CreateBSTNode(k)
    if root == nil {
        return newNode, true
    }

    parent, current := BSTFind(root, k)
    // if key is already in the tree, report error
    if current != nil {
        return root, false
    }

    if newNode.key < parent.key {
        parent.left = newNode
    } else {
        parent.right = newNode
    }

    return root, true
}
func BSTFindMin(root *BSTNode) *BSTNode {
    if root == nil { return nil }
    for root.left != nil {
        root = root.left
    }
    return root
}
```go
func BSTDelete(root *BSTNode, k int) (*BSTNode, bool) {
    if root == nil { return nil, false }
    parent, current := BSTFind(root, k)
    if current == nil { return root, false } // didn’t find

    var pPointer **BSTNode    // !!!
    if parent != nil {
        if current.key < parent.key {
            pPointer = &parent.left
        } else {
            pPointer = &parent.right
        }
    }

    switch {
    case current.left != nil && current.right != nil:
        min := BSTFindMin(current)
        BSTDelete(current, min.key)
        current.key = min.key
    case current.left == nil && current.right == nil:
        *pPointer = nil
    case current.left != nil:
        *pPointer = current.left
    case current.right != nil:
        *pPointer = current.right
    }

    return root, true
}
```

**BST Delete Code**

Find the node to delete and its parent

Coding jujutsu: pPointer is a pointer to the pointer in the parent that we have to change during the delete

The delete cases depend on which children exist in the node we are deleting
Summary

- Binary search trees are a fundamental data structure supporting the “dictionary” (aka map, associative array) operations.

- The requirement that the keys be unique is not crucial: it just adds a few more special cases to the code.

- The running time of all the operations is proportional to the height of the tree.

- Standard BSTs don’t do anything to keep the height small.
More about trees
Binary Tree Traversals

inorder:   HDIBEAJFCG
preorder:  ABEHIECFJG
postorder: HIDEBJFGCA

func traverse(T *Node) {
    if(T != nil) {
        PREORDER(T);
        traverse(T.left);
        INORDER(T);
        traverse(T.right);
        POSTORDER(T);
    }
}

How much space is used?
Basic Properties

- Every node except the root has exactly one parent.

- A tree with \( n \) nodes has \( n-1 \) edges (every node except the root has an edge to its parent).

- There is exactly one path from the root to each node. (Suppose there were 2 paths, then some node along the 2 paths would have 2 parents.)
Binary Trees – Definition

• An **ordered** tree is a tree for which the order of the children of each node is considered important.

  ![Diagram of ordered trees]

  ![Diagram of unordered trees]

  \( r \neq \)

• A **binary tree** is an ordered tree such that each node has \( \leq 2 \) children.

• Call these two children the **left** and **right** children.
Example Binary Trees

The edge cases:

- Only left child
- Only right child
- Single node
- Empty Binary Tree

Small binary tree:
Extended Binary Trees

Every internal node has exactly 2 children.

Every leaf (external node) has exactly 0 children.

Each external node corresponds to one $\Lambda$ in the original tree – let’s us distinguish different instances of $\Lambda$. 

Replace each missing child with external node

Do you need a special flag to tell which nodes are external?
# of External Nodes in Extended Binary Trees

**Thm.** An extended binary tree with \( n \) internal nodes has \( n+1 \) external nodes.

**Proof.** By induction on \( n \).

\( X(n) := \) number of external nodes in binary tree with \( n \) internal nodes.

**Base case:** \( X(0) = 1 = n + 1 \).

**Induction step:** Suppose theorem is true for all \( i < n \). Because \( n \geq 1 \), we have:

\[
X(n) = X(k) + X(n-k-1)
\]
\[
= k+1 + n-k-1 + 1
\]
\[
= n + 1 \quad \Box
\]
**Thm.** An extended binary tree with \( n \) internal nodes has \( n+1 \) external nodes.

**Proof.** Every node has 2 children pointers, for a total of \( 2n \) pointers.

Every node except the root has a parent, for a total of \( n - 1 \) nodes with parents.

These \( n - 1 \) parented nodes are all children, and each takes up 1 child pointer.

\[
\text{(pointers)} - \text{(used child pointers)} = \text{(unused child pointers)}
\]

\[
2n - (n-1) = n + 1
\]

Thus, there are \( n + 1 \) null pointers.

Every null pointer corresponds to one external node by construction. \( \square \)
Full and Complete Binary Trees

- If every node has either 0 or 2 children, a binary tree is called **full**.

- If the lowest $d-1$ levels of a binary tree of height $d$ are filled and level $d$ is partially filled from left to right, the tree is called **complete**.

- If all $d$ levels of a height-$d$ binary tree are filled, the tree is called **perfect**.

Unfortunately, different authors use different tree terminology.
**Thm.** A perfect tree of height \( h \) has \( 2^{h+1} - 1 \) nodes.

**Proof.** By induction on \( h \).

Let \( N(h) \) be number of nodes in a perfect tree of height \( h \).

**Base case:** when \( h = 0 \), tree is a single node. \( N(0) = 1 = 2^{0+1} - 1 \).

**Induction step:** Assume \( N(i) = 2^{i+1} - 1 \) for \( 0 \leq i < h \).

A perfect binary tree of height \( h \) consists of 2 perfect binary trees of height \( h-1 \) plus the root:

\[
N(h) = 2 \times N(h-1) + 1
= 2 \times (2^{h-1+1} - 1) + 1
= 2 \times 2^h - 2 + 1
= 2^{h+1} - 1 \quad \square
\]

\( 2^h \) are leaves
\( 2^h - 1 \) are internal nodes
**Thm.** In a non-empty, full binary tree, the number of internal nodes is always 1 less than the number of leaves.

**Proof.** By induction on $n$.

$L(n) :=$ number of leaves in a non-empty, full tree of $n$ internal nodes.

**Base case:** $L(0) = 1 = n + 1$.

**Induction step:** Assume $L(i) = i + 1$ for $i < n$.

Given $T$ with $n$ internal nodes, remove two sibling leaves.

$T'$ has $n-1$ internal nodes, and by induction hypothesis, $L(n-1) = n$ leaves.

Replace removed leaves to return to tree $T$.

**Thus:** $L(n) = n + 2 - 1 = n + 1$. 

Array Implementation for Complete Binary Trees

left(i): $2i$ if $2i \leq n$ otherwise 0
right(i): $(2i + 1)$ if $2i + 1 \leq n$ otherwise 0
parent(i): $\lfloor i/2 \rfloor$ if $i \geq 2$ otherwise 0
Summary

• Trees are an incredibly common way to organize data:
  • folders on your hard drives
  • URLs: http://www.cs.cmu.edu/~ckingsf/software/sailfish
  • BST, Splay trees, AVL trees, B-trees, Quad-trees, kd-trees, red-black trees, M-trees, … probably thousands of variants that are good for different data and different queries.

• Binary trees in particular are nice because each node partitions the data into 2 subsets and because there are nice relationships between # of nodes and # of leaves, etc.

• Typically, trees are represented using nodes & pointers, though this does not have to be the case.