Binary Search Trees

Today continue discussion of Data Structures

Operations: Insert, Delete, Search

Introduce: linked list (§ 10.7 of text CERS)

eample

head → 13[→ 3[→ 8[→ 77]

Data Structures Insert Delete Search Model Elements
Linked List $O(1)$ $O(1)$ $O(n)$ exact match
Sorted Array $O(n)$ $O(n)$ $O(lg n)$ comparison ($<$, $>$, $=$)
Hash Table $O(1)$ $O(1)$ $O(1)$ $O(1)$ hash function

Since we have achieved constant time with hashing, why wouldn't we need any other type of
data structure?

As turns out there are hundreds of data
structures and they are a very rich research
field. Reason is that different types of structures
either suited to different tasks. In this operation
summary, ability search are not the only ones we
may be interested in.

- Simple: minimum (easy to sorted list in sorted array)
- Useful: nearest match (hash)

Examples: Need fast search and fast min (sorted array) $O(1)$ in list,
- imagine running both structures in parallel
- can't built a search in hash ($O(n)$), min in sorted array ($O(1)$)
- problem: to maintain each data structure's integrity, need to insert/delete into each ($O(n)$) for sorted array
- does it help to maintain pointers between corresponding elements of each data structure?
Today we introduce Binary Search Trees (BST)

BST Property

1. All nodes in left subtree of X have key \( \leq \text{key of } X \)
2. All nodes in right subtree of X have key \( \geq \text{key of } X \)

BST Traits

- Binary tree (at most 2 branches per node), left/right rooted
- Each node contains:
  - key \( k \)
  - left node
  - right node
  - parent \( p \)
  - satellite data \( D \)
To illustrate how the algorithm works, let's take a look at how it works:

**Inorder (x)**
- Inorder (left[x]) unless left[x]=nil
- Print key [x]
- Inorder (right[x]) unless right[x]=nil

**Look at our example:**
- Recursively find leftmost element and print it. Then prints parent and right, etc.
- 2 4 7 12 21 23 42

That is, inorder tree walks visits each node in sorted key order. It really just visits each node once, and through recursive calls keeps placeholders throughout the tree.

**Time:** $O(n)$ where n is the number of nodes in the tree.

Now, imagine using the binary search tree as a data structure for information retrieval.

**Search (T, k):**
- $x ← \text{root}(T)$
- While $x ≠ \text{nil}$ and key[x] ≠ k,
  - Do if $k < \text{key}[x]$
    - Then $x ← \text{left}[x]$
  - Else $x ← \text{right}[x]$
- Return $x$

*Story of root node:
- If found, done.
- Otherwise, use BST properties to decide which sub-tree to go down.
- If key is not found, return nil when key does not exist.*

*Like binary search, but are not necessarily of sub-trees, will half each size search. If due is reasonably well balanced, proportionally still fast.*
Analysis of Search

1. First form of comparison model (like binary search)
2. Correctness (invariant)
   - if \( k < \text{key} \), then by BST property, \( k < \text{all keys} \)
     in right subtree (and can safely ignore)
3. Running Time
   - To successfully find node \( x \) \( \rightarrow \Theta(\text{depth}(x)) \)
     where depth = distance from root down to \( x \)
   - Worst case \( \rightarrow \Theta(\text{height}(T)) = \Theta(h) \)
   - Worst case for any tree of \( n \) nodes \( = \Theta(n) \)

What about nearest matches?

- Unsuccessful search: identify location where key would fit
  - i.e., if last move was left
    - next larger element is last node visited
    - next smaller element is last [true or] moved right
      (ditto symmetric statements if last move was right)
Dynamic BST

Insert \((T, k)\)
- Run search until reach \(nil\) (predicate \(k \neq B.T.J = nil\))
- \(B.J.T.J \leftarrow \text{new node with } B.J.T.k = k, B.J.T.l = nil, B.J.T.r = nil, B.J.T.p = \text{new node}\)

(Example)
Running time \(\Theta(k)\)

BST Sort \((A)\)

for \(i \leftarrow 1 \text{ to } n\)
do Insert \((A[i], i)\)
InOrder (root)

Analysis: same comparisons as quicksort, but in different order

\(\Theta(n \log n)\) time in average case (uniformly random permutations)
- avg depth of node = \(\Theta(n \log n)\)
  \(= \text{avg time for insert/search}\)
  - In fact, worst case depth of node \(= \Theta(n \log n)\) in worst case
More BST operations

Minimum (x)

- minimum key in subtree rooted at x

while left[x] ≠ nil
  do x ← left[x]
  return x

Time = O(h)

Maximum is simple change

Successor (x)

- next higher element in Monotony

if right[x] ≠ nil
  then return minimum(right[x])
else y ← parent[x]
  while y ≠ nil and x = right[y]
    do x ← y
    y ← parent[y]
  return y

Time = O(h)

Predecessor is simple change

Delete (x)

- given pointer to a node

1. If x has no children (leaf of tree) then remove x
2. If x has exactly one child, then splice it out
3. If x has ≥ children, then swap x with successor(x)

Time = O(h)
Problem: Worst case BST height is $\Theta(n)$  
   $\Rightarrow$ degenerates to linked list

Solution: Force BST (or related tree) to stay "balanced"  
   $\Rightarrow$ have $\Theta(\log n)$ height

Will get all ops in $\Theta(\log n)$ time

BST ops

- Search
- Insert
- Delete
- Minimum
- Maximum
- Successor
- Predecessor

$\Theta(h)$

started discussion with $\Theta(n)$ data
   problems. Now, with balanced tree, am
   all $\Theta(\log n)$