Lectures 22-23
Binary Search Trees

Adam Smith
Heaps: Review

Heap-leap-jeep-creep(A):
1. A.heap-size = 0
2. For i=1 to len(A)
   • tmp = A[i]
   • Heap-Insert(A,tmp)
3. For i=len(A) down to 1
   • tmp = Heap-Extract-Max(A)
   • A[i]=tmp

What does this do?
What is the running time of this operation?
Trees

- Rooted Tree: collection of nodes and edges
  - Edges go down from root (from parents to children)
  - No two paths to the same node
  - Sometimes, children have “names” (e.g. left, right, middle, etc)
Binary Search Trees

- Binary tree: every node has 0, 1 or 2 children
- BST property:
  - If $y$ is in left subtree of $x$, then $\text{key}[y] \leq \text{key}[x]$
  - same for right
Binary Search Trees

- Binary tree: every node has 0, 1 or 2 children
- BST property:
  - If $y$ is in left subtree of $x$, then $key[y] \leq key[x]$
  - same for right

S. Raskhodnikova and A. Smith. Based on slides by C. Leiserson and E. Demaine.
Implementation

• Keep pointers to parent and both children

• Each NODE has
  - left: pointer to NODE
  - right: pointer to NODE
  - p: pointer to NODE
  - key: real number (or other type that supports comparisons)
Drawing BST’s

- To help you visualize relationships:
  - Drop vertical lines to keep nodes in right order
- For example: picture shows the only legal location to insert a node with key 4
Height of a BST can be...

- ... as little as \( \log(n) \)
  - full balanced tree

- ... as much as \( n-1 \)
  - unbalanced tree
Searching a BST

```plaintext
TREE-SEARCH(x, k)
if x == NIL or k == key[x]
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

Initial call is TREE-SEARCH(T.root, k).

- Running time: $\Theta(\text{height})$
Insertion

- Find location of insertion, keep track of parent during search
- Running time: $\Theta(\text{height})$

```
TREE-INSERT($T, z$)
  $y = \text{NIL}$
  $x = T.\text{root}$
  while $x \neq \text{NIL}$
    $y = x$
    if $z.key < x.key$
      $x = x.left$
    else $x = x.right$
  $z.p = y$
  if $y == \text{NIL}$
    $T.\text{root} = z$  // tree $T$ was empty
  elseif $z.key < y.key$
    $y.left = z$
  else $y.right = z$
```
Tree Min and Max

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x

TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```

• Running time: $\Theta(\text{height})$
Tree Successor

TREE-SUCCESSOR(x)

if x.right ≠ NIL
    return TREE-MINIMUM(x.right)

y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y

• Running time: $\Theta(\text{height})$
Traversals (not just for BSTs)

• Traversal: an algorithm that visits every node in the tree to perform some operation, e.g.,
  ➢ Printing the keys
  ➢ Updating some part of the structure

• 3 basic traversals for trees
  ➢ Inorder
  ➢ Preorder
  ➢ Postorder
Inorder traversal

\begin{algorithm}
\textbf{INORDER-Tree-Walk} (x)
\begin{algorithmic}
\IF {x \neq NIL}
\STATE \textbf{INORDER-Tree-Walk} (x.left)
\STATE \textbf{print} \text{key}[x]
\STATE \textbf{INORDER-Tree-Walk} (x.right)
\ENDIF
\end{algorithmic}
\end{algorithm}

- Running time: $\Theta(n)$
Postorder traversal

• Write pseudocode that computes
  ➢ height of a BST
  ➢ number of nodes in a BST
  ➢ average depth
  ➢ ...

• Example: height
  ➢ Find-height(T)
    • if T==NIL return -1
    • else
      – h1 := Find-height(T.left)
      – h2 := Find-height(T.left)
      – return max(h1, h2) + 1
Preorder Traversal

• Recall that insertion order affects the shape of a BST
  ➢ insertion in sorted order: height $n-1$
  ➢ random order: height $O(\log n)$ with high probability (we will prove that later in the course)
• Write pseudocode that prints the elements of a binary search tree in a plausible insertion order
  ➢ that is, an insertion order that would produce this particular shape
• (print nodes in a preorder traversal)
Exercise on insertion order

- Exercise: Write pseudocode that takes a sorted list and produces a “good” insertion order (that would produce a balanced tree)
- (Hint: divide and conquer: always insert the median of a subarray before inserting the other elements)
Deletion

- Cases analyzed in book
- Running time: $\Theta(\text{height})$

```
TRANSLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v != NIL
    v.p = u.p
```

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSLANT(T, z, z.right) // z has no left child
elseif z.right == NIL
    TRANSLANT(T, z, z.left) // z has just a left child
else // z has two children.
    y = TREE-MINIMUM(z.right) // y is z’s successor
    if y.p != z
        // y lies within z’s right subtree but is not the root of this subtree.
        TRANSLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    // Replace z by y.
    TRANSLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```
Binary-search-tree sort

\[ T \leftarrow \emptyset \quad \triangleright \text{Create an empty BST} \]
for \( i = 1 \) to \( n \)
do \text{TREE-INSERT}(T, A[i])
Perform an inorder tree walk of \( T \).

Example:
\( A = [3 \ 1 \ 8 \ 2 \ 6 \ 7 \ 5] \)

Tree-walk time = \( O(n) \),
but how long does it take to build the BST?
Analysis of BST sort

BST sort performs the same comparisons as quicksort, but in a different order!

The expected time to build the tree is asymptotically the same as the running time of quicksort.
Node depth

The depth of a node $= \text{the number of comparisons made during TREE-INSERT}$. Assuming all input permutations are equally likely, we have

\[
\text{Average node depth} \quad = \quad \frac{1}{n} \mathbb{E} \left[ \sum_{i=1}^{n} (\text{# comparisons to insert node } i) \right] \\
= \frac{1}{n} O(n \lg n) \quad \text{(quicksort analysis)} \\
= O(\lg n).
\]