CS161 - Search Trees

David Kauchak

- Binary Search - Given a sorted list of values $A$, find a particular value. Similar to looking something up in a dictionary or phone book: $O(\log n)$

- Binary search tree (BST) - A binary search tree is a binary tree where a parent's value is greater than all children to the left and less than or equal to all children to the right. Specifically, given a node $x$ in a BST:

$$\text{Left}(x) < x \leq \text{Right}(x)$$

As with other tree structures, can be implemented with pointers or with an array

Look at example(s)

- Given the definition, what else can we say?
  * All elements to the left of a node are less than the node
  * All elements to the right of a node are greater than or equal to the node
  * The smallest element is the left-most node
  * The largest element is the right-most node

- Why not the setup below?:

$$\text{Left}(x) \leq x \leq \text{Right}(x)$$

- Which of the set operations is this data structure good/bad for?
  * SEARCH$(S, k)$ - good
  * INSERT$(S, k)$ - average
* **DELETE**(*S*, *x*) - average
* **MINIMUM**(*S*) - good
* **MAXIMUM**(*S*) - good

Enumerating the elements in order:

**INORDER_TREE_WALK**(x)

1. if *x* ≠ null
   2. **INORDER_TREE_WALK**(Left(*x*))
   3. print *x*
   4. **INORDER_TREE_WALK**(Right(*x*))

* Is it correct?
  Definition of BST: Left(*x*) < *x* ≤ Right(*x*) and proof by induction.
* Runtime?
  Given a node with *k* nodes in the left subtree and *n* − *k* − 1 nodes in the right subtree, the recurrence is:

\[
T(n) = T(k) + T(n - k - 1) + c
\]

we can solve this, or, answer the following two questions:

1. How much work is done for each call to **INORDER_TREE_WALK**?
2. How many calls are made to **INORDER_TREE_WALK**?

* What needs to be changed to traverse in reverse order?

* Pre-order and post-order traversals?

Searching for a particular value:

**BST_SEARCH**(x, *k*)

1. if *x* = null or *k* = *x*
2. return *x*
3. elseif *k* < *x*
4. return **BST_SEARCH**(Left(*x*), *k*)
5. else
6. return **BST_SEARCH**(Right(*x*), *k*)
IterativeBSTSearch\((x, k)\)
1. while \(x \neq \text{null}\) and \(k \neq x\)
2. \hspace{1em} if \(k < x\)
3. \hspace{2em} \(x \leftarrow \text{Left}(x)\)
4. \hspace{1em} else
5. \hspace{2em} \(x \leftarrow \text{Right}(x)\)
6. return \(x\)

1. Is it correct?
2. Runtime? What is the worst case? The node we’re looking for is a leaf and it is the deepest leaf - \(O(h)\)

- Finding the min/max

BSTMin\((x)\)
1. if \(\text{Left}(x) = \text{null}\)
2. \hspace{1em} return \(x\)
3. else
4. \hspace{1em} return BSTMin(\text{Left}(x))

IterativeBSTMin\((x)\)
1. while \(\text{Left}(x) \neq \text{null}\)
2. \hspace{1em} \(x \leftarrow \text{Left}(x)\)
3. return \(x\)

* Is it correct?
  LEFT\((x) < x \leq \text{Right}(x)\), therefore the smallest element is the leftmost element.
* Runtime? We always visit a leave of the tree. Worst case, this leave is the lowest leave - \(O(h)\)
* What needs to be changed to find the max?

- Successor and predecessor
  
  * A simple look:
    
    - Predecessor is the right-most node of the left sub-tree, i.e. the largest node of all of the elements that are less than a node.
    - Successor is the left-most node of the right sub-tree, i.e. the smallest node of all of the elements that are larger than a node.
* What if a node does not have a left or right subtree?

Let’s examine successor. If a node \( x \) doesn’t have a right sub-tree, then either the element is the largest element and doesn’t have a successor or it’s successor, call it \( y \), is the element in the tree to which \( x \) is the predecessor. So, we want to find the node \( y \) such that \( x \) is the right-most node of the left sub-tree of \( y \). Another way of saying it, we want to find the lowest ancestor of \( x \) whose left child is also an ancestor of \( x \).

**Successor(\( x \))**

1. if Right(\( x \)) \( \neq \) null
2. return BSTMin(Right(\( x \)))
3. else
4. \( y \leftarrow \) Parent(\( x \))
5. while \( y \neq \) null and \( x \) = Right(\( y \))
6. \( x \leftarrow y \)
7. \( y \leftarrow \) Parent(\( y \))
8. return \( y \)

- Is it correct?
- Runtime? Worst case, we have to traverse the tree from one of the leaves to the root. \( O(h) \)

- Insertion into a BST
BSTInsert($T, x$)
1. if $\text{Root}(T) = \text{null}$
2. \hspace{1em} $\text{Root}(T) \leftarrow x$
3. else
4. \hspace{1em} $y \leftarrow \text{Root}(T)$
5. while $y \neq \text{null}$
6. \hspace{2em} $\text{prev} \leftarrow y$
7. if $x < y$
8. \hspace{3em} $y \leftarrow \text{Left}(y)$
9. else
10. \hspace{3em} $y \leftarrow \text{Right}(y)$
11. $\text{Parent}(x) \leftarrow \text{prev}$
12. if $x < \text{prev}$
13. \hspace{2em} $\text{Left}(\text{prev}) \leftarrow x$
14. else
15. \hspace{2em} $\text{Right}(\text{prev}) \leftarrow x$

* Is it correct? Assuming no duplicates in the tree, finds the appropriate parent and inserts the value. Lines 6-8 make sure that the BST property is maintained.

What happens if there is a duplicate?
* Runtime? $O(h)$

- Deleting a node: 3 cases
  1. If $x$ has no children, remove $x$
  2. If $x$ has only one child, splice out $x$
  3. If $x$ has two children, replace $x$ with its successor in the list.
     Will it always have a successor?
     * Is it correct?
     * Runtime? $O(h)$ for the call to find the successor.

- Examples

- Most of the algorithms run in time bounded by the height of the tree.
  * What is the worst case height? When does this happen?
  * What is the best case height?
• Randomized BST version - The expected height of a randomly built binary search tree is \( O(\log n) \), i.e. a tree where the values inserted are randomly selected.

• Balanced trees - If we can make sure that the trees are balanced, then all of the operations bounded by the height run in time \( O(\log n) \).

Red-Black trees, AVL trees, ...

• B-Trees

  − A B-Tree is a balanced \( n \)-ary tree with the following properties:
    * Each node \( x \) contains between \( t - 1 \) and \( 2t - 1 \) keys (denoted \( n(x) \)) stored in increasing order, denoted \( K_x \):
      \[ K_x = K_x[1] \leq K_x[2] \leq \ldots \leq K_x[n(x)] \]
    * Each internal node also contains \( n(x) + 1 \) children (i.e. between \( t \) and \( 2t \) children), denoted \( C_x = C_x[1], C_x[2], \ldots, C_x[n(x)+1] \)
    * The keys of a parent delimit the values that a child’s keys can take. Specifically
      \[ K_{C_x[1]} \leq K_x[1] \leq K_{C_x[2]} \leq K_x[2] \leq \ldots \leq K_x[n(x)] \leq K_{C_x[n(x)+1]} \]

      For example, if the a node has \( K_x[i] = 15 \) and \( K_x[i+1] = 25 \)
      then child \( i + 1 \) must have keys between 15 and 25.

    * All leaves have the same depth

  − Example B-Tree

  − Why B-Trees vs. Red-Black vs ...?

    * Memory is limited or there is huge amount of data to be stored
    * In the extreme, only one node is kept in memory and the rest on disk
    * Size of the nodes is determined by a page size in memory
    * We will count both run-time as well as the number of disk accesses
    * Because \( t \) is generally large, the height of a B-tree is generally quite small, e.g. if \( t = 1001 \) then a B-Tree of height 2 can over one billion values.
- Height of a B-Tree
  For a tree of height \( h \), what is the smallest number of keys a B-Tree can have?

  \( h = 0 \), 1 node
  \( h = 1 \), 2 nodes
  \( h = 2 \), \( 2t \) nodes
  \( h = 3 \), \( 2t^2 \) nodes

  and each node must contain at least \( t - 1 \) keys

  \[
  n \geq 1 + (t - 1) \sum_{i=1}^{h} 2^i - 1
  = 1 + 2(t - 1) \left( \frac{t^h - 1}{t - 1} \right)
  = 2t^h - 1
  \]

  so, \( t^h \leq \frac{n + 1}{2} \) and \( h \leq \log_t \frac{n + 1}{2} \)

B-TreeSearch\((x, k)\)
  1. \( i \leftarrow 1 \)
  2. while \( i \leq n(x) \) and \( k > K_x[i] \)
     \( i \leftarrow i + 1 \)
  3. if \( i \leq n(x) \) and \( k = K_x[i] \)
     return \((x, i)\)
  4. else
     DiskRead\((C_x[i])\)
     return B-TreeSearch\((C_x[i], k)\)

* Is it correct?
* Runtime?
  \( O(h) = O(\log_t n) \) calls to B-TreeSearch

  \( O(\log_t n) \) disk accesses

  Each call to B-TreeSearch takes at most \( O(t) \) time, so runtime is \( O(t \log_t n) \)
Why don’t we use binary search to find the correct location?

Inserting a node into a B-Tree
Starting at the root, follow the appropriate path down to a leaf node by finding the child such that \( key_i[x] < val \leq key_{i+1}[x] \). At each node:

- If the node is full (contains \( 2t - 1 \) keys), split the keys about the medial value into two nodes and add this median value to the parent node
- If the node is a leaf node, insert it into its correct spot

Walk though example in book

Is it correct?
- Does the item end up in the correct place?
- Are the tree properties maintained?

Running time?
Without any splitting, similar to B-TREE SEARCH with one additional disk write.

What happens when a node is split?
- 3 disk write operations, one for the parent node and 2 for the split nodes
- Runtime is \( O(t) \) to split a node since we’re just iterating through the elements a few times

What’s the maximum number of nodes that can be split? \( O(h) \)
In both of these situations, \( O(h) = O(\log_t n) \) disk accesses and runtime of \( O(th) = O(t \log_t n) \)

Deleting a node from a B-Tree
\( O(\log_t n) \) disk accesses \( O(t \log_t n) \) runtime

These notes are adapted from material found in chapters 12, 18 of [1].

References