Search Trees

Computer Science S-111
Harvard University
David G. Sullivan, Ph.D.

Binary Search Trees

- Search-tree property: for each node $k$:
  - all nodes in $k$’s left subtree are $< k$
  - all nodes in $k$’s right subtree are $\geq k$
- Our earlier binary-tree example is a search tree:

  ![Binary Search Tree Diagram]

- With a search tree, an inorder traversal visits the nodes in order!
  - in order of increasing key values
Searching for an Item in a Binary Search Tree

• Algorithm for searching for an item with a key \( k \):
  if \( k == \) the root node’s key, you’re done
  else if \( k < \) the root node’s key, search the left subtree
  else search the right subtree

• Example: search for 7

Implementing Binary-Tree Search

```java
public class LinkedTree {
    // Nodes have keys that are ints
    private Node root;

    public LLList search(int key) {    // "wrapper method"
        Node n = searchTree(root, key);  // get Node for key
        if (n == null) {
            return null;     // no such key
        } else {
            return n.data;   // return list of values for key
        }
    }

    private static Node searchTree(Node root, int key) {
        // two base cases (order matters!)
        if ( ) {
            // two recursive cases
        } else if (    ) {
        } else if (    ) {
        } else {
        }
    }
}
```
Inserting an Item in a Binary Search Tree

- We need a method with this header
  ```java
  public void insert(int key, Object data)
  ```
  that we can use to add a new (key, data) pair to the tree.

- Example 1: a search tree containing student records
  - key = the student's ID number (an integer)
  - data = a string with the rest of the student record
  - we want to be able to write client code that looks like this:
  ```java
  LinkedTree students = new LinkedTree();
  students.insert(23, "Jill Jones,sophomore,comp sci");
  students.insert(45, "Al Zhang,junior,english");
  ```

- Example 2: a search tree containing scrabble words
  - key = a scrabble score (an integer)
  - data = a word with that scrabble score
  ```java
  LinkedTree tree = new LinkedTree();
  tree.insert(4, "lost");
  ```

- We want to insert an item whose key is \( k \).
- We traverse the tree as if we were searching for \( k \).
- If we find a node with key \( k \), we add the data item to the list of items for that node.
  ```java
  example: tree.insert(4, "sail")
  ```
- If we don't find it, the last node we encounter will be the parent \( P \) of the new node (see example at right).
  - if \( k < P \)'s key, make the new node \( P \)'s left child
  - else make the node \( P \)'s right child
- Special case: if the tree is empty, make the new node the root of the tree.
- **Important:** The resulting tree is still a search tree!
Implementing Binary-Tree Insertion

- We'll implement part of the `insert()` method together.
- We'll use iteration rather than recursion.
- Our method will use two references/pointers:
  - `trav`: performs the traversal down to the point of insertion
  - `parent`: stays one behind `trav`
    - like the `trail` reference that we sometimes use when traversing a linked list

```java
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        } // what should go here?
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
        // what should go here?
    }
    Node newNode = new Node(key, data);
    if (root == null) {   // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```
Deleting Items from a Binary Search Tree

- Three cases for deleting a node $x$
- **Case 1:** $x$ has no children.
  Remove $x$ from the tree by setting its parent’s reference to null.

  ex: delete 4

- **Case 2:** $x$ has one child.
  Take the parent’s reference to $x$ and make it refer to $x$’s child.

  ex: delete 12

- **Case 3:** $x$ has two children
  - we can’t give both children to the parent. why?

  • instead, we leave $x$’s node where it is, and we replace its contents with those from another node
  • the replacement must maintain the search-tree inequalities

  ex:
  delete 12

  two options: which ones?
Deleting Items from a Binary Search Tree (cont.)

- **Case 3:** \( x \) has two children (continued):
  - replace \( x \)'s contents with those from the smallest node in \( x \)'s right subtree—call it \( y \)
  - we then delete \( y \)
    - it will either be a leaf node or will have one right child. why?
  - thus, we can delete it using case 1 or 2

\[
\text{ex: delete 12}
\]

![Diagram of deleting 12 from a binary search tree]

<table>
<thead>
<tr>
<th>12</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>20</td>
</tr>
</tbody>
</table>

- **Which Nodes Could We Use To Replace 9?**

![Diagram of a binary search tree with a node labeled 9]
Implementing Deletion

```java
define 26:
public LLList delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }
    // Delete the node (if any) and return the removed items.
    if (trav == null) {  // no such key
        return null;
    } else {
        LLList removedData = trav.data;
        deleteNode(trav, parent);  // call helper method
        return removedData;
    }
}
```

Implementing Case 3

```java
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        // Find a replacement - and
        // the replacement's parent.
        Node replaceParent = toDelete;
        // Get the smallest item
        // in the right subtree.
        Node replace = toDelete.right;
        // what should go here?
    } else {
        ...
    }
    // Replace toDelete's key and data
    // with those of the replacement item.
    toDelete.key = replace.key;
    toDelete.data = replace.data;
    // Recursively delete the replacement
    // item's old node. It has at most one
    // child, so we don't have to
    // worry about infinite recursion.
    deleteNode(replace, replaceParent);
    } else {
        ...
    }
```
Implementing Cases 1 and 2

```java
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        ...
    } else {
        Node toDeleteChild;
        if (toDelete.left != null)
            toDeleteChild = toDelete.left;
        else
            toDeleteChild = toDelete.right;
        // Note: in case 1, toDeleteChild
        // will have a value of null.
        if (toDelete == root)
            root = toDeleteChild;
        else if (toDelete.key < parent.key)
            parent.left = toDeleteChild;
        else
            parent.right = toDeleteChild;
    }
}
```

Recall: Path, Depth, Level, and Height

- There is exactly one path (one sequence of edges) connecting each node to the root.
- *depth* of a node = # of edges on the path from it to the root
- Nodes with the same depth form a *level* of the tree.
- The *height* of a tree is the maximum depth of its nodes.
  - example: the tree above has a height of 2
Efficiency of a Binary Search Tree

• For a tree containing \( n \) items, what is the efficiency of any of the traversal algorithms?
  • you process all \( n \) of the nodes
  • you perform \( O(1) \) operations on each of them

• Insertion and deletion both have the same time complexity as searching.
  • insertion is a search followed by \( O(1) \) operations
  • deletion involves either:
    • a search followed by \( O(1) \) operations (cases 1 and 2)
    • a search partway down the tree for the item, followed by a search further down for its replacement, followed by \( O(1) \) operations (case 3)

Efficiency of a Binary Search Tree (cont.)

• Time complexity of searching:
  • best case:

  • worst case:
    • you have to go all the way down to level \( h \) before finding the key or realizing it isn't there
    • along the path to level \( h \), you process \( h + 1 \) nodes
  • average case:

• What is the height of a tree containing \( n \) items?
Balanced Trees

- A tree is balanced if, for each of its nodes, the node’s subtrees have the same height or have heights that differ by 1.
  - example:
    - 26: both subtrees have a height of 1
    - 12: left subtree has height 0
      right subtree is empty (height = -1)
    - 32: both subtrees have a height of 0
    - all leaf nodes: both subtrees are empty
  - For a balanced tree with \( n \) nodes, height = \( O(\log n) \)
    - each time that you follow an edge down the longest path, you cut the problem size roughly in half!
  - Therefore, for a balanced binary search tree, the worst case for search / insert / delete is \( O(h) = O(\log n) \)
    - the "best" worst-case time complexity

What If the Tree Isn't Balanced?

- Extreme case: the tree is equivalent to a linked list
  - height = \( n - 1 \)
  - Therefore, for a unbalanced binary search tree, the worst case for search / insert / delete is \( O(h) = O(n) \)
    - the "worst" worst-case time complexity
- We’ll look next at search-tree variants that take special measures to ensure balance.
2-3 Trees

- A 2-3 tree is a balanced tree in which:
  - all nodes have equal-height subtrees (perfect balance)
  - each node is either
    - a 2-node, which contains one data item and 0 or 2 children
    - a 3-node, which contains two data items and 0 or 3 children
  - the keys form a search tree

- Example:

```
2-node:
  k
  <k  >k

3-node:
  k1  k2
  <k1 <k2 >k2
```

Search in 2-3 Trees

- Algorithm for searching for an item with a key k:
  if k == one of the root node’s keys, you’re done
  else if k < the root node’s first key
    search the left subtree
  else if the root is a 3-node and k < its second key
    search the middle subtree
  else
    search the right subtree

- Example: search for 87

```
28 61
  10 40
  3 14 20 34 51 68 80 87 93 97
```

```
28 61
  10 40
  3 14 20 34 51 68 80 87 93 97
```
Insertion in 2-3 Trees

- Algorithm for inserting an item with a key $k$:
  - search for $k$, but don't stop until you hit a leaf node
  - let L be the leaf node at the end of the search
  - if L is a 2-node
    - add $k$ to L, making it a 3-node
  - else if L is a 3-node
    - split L into two 2-nodes containing the items with the smallest and largest of: $k$, L’s 1st key, L’s 2nd key
    - the middle item is “sent up” and inserted in L’s parent

example: add 52

Example 1: Insert 8

- Search for 8:

- Add 8 to the leaf node, making it a 3-node:
Example 2: Insert 17

- Search for 17:

- Split the leaf node, and send up the middle of 14, 17, 20 and insert it the leaf node's parent:

Example 3: Insert 92

- In which node will we initially try to insert it?
Example 3: Insert 92

- Search for 92:

```
   28 61
   10
   3 14 20 34 51 68 80 87 93 97
```

- Split the leaf node, and send up the middle of 92, 93, 97 and insert it the leaf node’s parent:

```
   28 61
   40
   34 51 68 80 87 93 97
```

- In this case, the leaf node’s parent is also a 3-node, so we need to split is as well…
Example 3 (cont.)

- We split the [77 90] node and we send up the middle of 77, 90, 93:
  - We try to insert it in the root node, but the root is also full!

![Diagram showing tree splitting](image)

- Then we split the root, which increases the tree’s height by 1, but the tree is still balanced.
- This is only case in which the tree’s height increases.

Efficiency of 2-3 Trees

- A 2-3 tree containing n items has a height \( \leq \log_2 n \).
- Thus, search and insertion are both \( O(\log n) \).
  - a search visits at most \( \log_2 n \) nodes
  - an insertion begins with a search; in the worst case, it goes all the way back up to the root performing splits, so it visits at most \( 2\log_2 n \) nodes
- Deletion is tricky – you may need to coalesce nodes! However, it also has a time complexity of \( O(\log n) \).
- Thus, we can use 2-3 trees for a \( O(\log n) \)-time data dictionary.
External Storage

- The balanced trees that we've covered don't work well if you want to store the data dictionary externally – i.e., on disk.

- Key facts about disks:
  - data is transferred to and from disk in units called blocks, which are typically 4 or 8 KB in size
  - disk accesses are slow!
    - reading a block takes ~10 milliseconds (10^{-3} sec)
    - vs. reading from memory, which takes ~10 nanoseconds
    - in 10 ms, a modern CPU can perform millions of operations!

B-Trees

- A B-tree of order $m$ is a tree in which each node has:
  - at most $2m$ entries (and, for internal nodes, $2m + 1$ children)
  - at least $m$ entries (and, for internal nodes, $m + 1$ children)
  - exception: the root node may have as few as 1 entry
  - a 2-3 tree is essentially a B-tree of order 1

- To minimize the number of disk accesses, we make $m$ as large as possible.
  - each disk read brings in more items
  - the tree will be shorter (each level has more nodes), and thus searching for an item requires fewer disk reads

- A large value of $m$ doesn't make sense for a memory-only tree, because it leads to many key comparisons per node.

- These comparisons are less expensive than accessing the disk, so large values of $m$ make sense for on-disk trees.
Example: a B-Tree of Order 2

- \( m = 2 \): at most \( 2m = 4 \) items per node (and at most 5 children)
  - at least \( m = 2 \) items per node (and at least 3 children)
    - (except the root, which could have 1 item)
- The above tree holds the same keys this 2-3 tree:

Search in B-Trees

- Similar to search in a 2-3 tree.
- Example: search for 87
Insertion in B-Trees

• Similar to insertion in a 2-3 tree:
  search for the key until you reach a leaf node
  if a leaf node has fewer than $2m$ items, add the item
to the leaf node
  else split the node, dividing up the $2m + 1$ items:
    the smallest $m$ items remain in the original node
    the largest $m$ items go in a new node
    send the middle entry up and insert it (and a pointer to
    the new node) in the parent

• Example of an insertion without a split: insert 13

Splits in B-Trees

• Insert 5 into the result of the previous insertion:

  The middle item (the 10) was sent up to the root.
  It has no room, so it is split as well, and a new root is formed:

• Splitting the root increases the tree’s height by 1, but the tree
  is still balanced.  This is only way that the tree’s height increases.
  When an internal node is split, its $2m + 2$ pointers are split evenly
  between the original node and the new node.
Analysis of B-Trees

- All internal nodes have at least $m$ children (actually, at least $m+1$).
- Thus, a B-tree with $n$ items has a height $\leq \log_m n$, and search and insertion are both $O(\log_m n)$.
- As with 2-3 trees, deletion is tricky, but it’s still logarithmic.

Search Trees: Conclusions

- Binary search trees can be $O(\log n)$, but they can degenerate to $O(n)$ running time if they are out of balance.
- 2-3 trees and B-trees are balanced search trees that guarantee $O(\log n)$ performance.
- When data is stored on disk, the most important performance consideration is reducing the number of disk accesses.
- B-trees offer improved performance for on-disk data dictionaries.