7. Binary Search Trees

Oliver Diessel
odiessel@cse.unsw.edu.au

Last updated: 11:00 8 Sep 2008
Binary Search Trees

Topics to be covered:

- What are trees and BSTs?
- Examples
- BST operations:
  - Searching
  - Insertion
  - Traversal
  - Freeing memory for BST

Slide credits: Alistair Moffat
Lists are a one dimensional recursive structure – each node has one pointer.

If each node has two pointers, a two-dimensional structure can be built.

A **binary tree** is a two-dimensional data structure in which objects are threaded together using **two** pointers in each node.
Example of a binary tree

Path length = 1
Level 1

Level 2

Depth 3 (Level 3)
Defined by path length from root to leaf

Height of tree = 3
Defined by longest path length from root to leaves
Binary Trees

- Lists are a one dimensional recursive structure – each node has one pointer.
- If each node has two pointers, a two-dimensional structure can be built.
- A binary tree is a two-dimensional data structure in which objects are threaded together using two pointers in each node.
- The tree has a root, and a set of leaves. Each node except the root has a parent; and each node has zero, one, or two children.
- If each non-leaf node has two children, the number of leaves in a binary tree of depth $d$ is as large as $2^d$.
- A tree of depth 20 can have over one million leaves.
Binary Search Tree

- If all insertions in the tree follow the rule (at each node) that
  - if the new item has a key *less than* the key at this node, it is recursively inserted into the “left” subtree, and
  - if the new item has a key *greater than* the key at this node, it is recursively inserted into the “right” subtree,

then the tree is a *binary search tree*, and the leaves and internal nodes appear in sorted order from left to right across the tree.

- A *binary search tree* is a binary tree in which the objects are *ordered* from left to right across the tree.
Example of a binary search tree

This BST over the planet names results when they are inserted according to distance from the sun.

The virtue of the BST is the speed with which it can be searched.

Try determining whether Pluto is a planet…
A different insertion order is likely to result in a BST with a different shape.

The first item inserted always remains at the root.

In a BST containing $n$ items there are $n + 1$ NULL pointers.

Each NULL pointer denotes the insertion point for a range of new elements.

For any given new element, there is a single correct insertion point if the tree is to remain a BST.

To search the tree, start at the root. At each level, check the search key against the key in the stored node, and move either left or right, if the search key is less than or greater than respectively.

Either the item will be located, or a NULL pointer will be reached if the search key does not appear in the tree.
Binary Search Trees

- The cost of searching a BST is proportional to the length of the access path.
  - If the tree is large, and reasonably balanced, only a small fraction of all items are on the access path.
- If \( n \) items are inserted into the tree in random order, then the average length of each access path is \( 1.4\log_{2} n \) [see Knuth, The Art of Computer Programming, Vol 3 for proof outline]
  - This is much faster than searching a linked list or an array.
- If the items are not random, the access path might be \( n \) items long.
  - Your tree may become a stick!
- A randomly ordered set of objects placed into a BST can be searched extremely quickly.
  - If the insertion order is not random, there is a risk that the searching behavior will become significantly slower.
Assignment 1 using BSTs

Need functions to…

- search a BST for a word
- insert a word into a BST
- print out the BST
- free up the memory used by the BST
Assignment 1 using BSTs

Define BST node structure:

typedef struct node node_t;

struct node {
    char *word;               /* distinct word */
    int freq;                 /* frequency of word */
    node_t *left;             /* left subtree of node */
    node_t *right;            /* right subtree of node */
};
Searching the tree

node_t *search_tree(node_t *tree, char *search_word);
/*
 * Search the nodes rooted at *tree for *search_word
 * return a pointer to the node found
 *
 * Call search_tree recursively if search_word could be
 * in either the left or right subtree
 */

typedef struct node node_t;

struct node {
    char *word;                /* distinct word */
    int freq;                  /* frequency of word */
    node_t *left;              /* left subtree of node */
    node_t *right;             /* right subtree of node */
};
Inserting into the tree

```c
node_t *insert_in_order(node_t *tree, char *new_word);
/*
 * Insert *new_word into tree rooted at *tree and
 * return pointer to updated tree
 *
 * Call insert_in_order recursively if new_word should be
 * inserted either into the left or right subtree
 */

typedef struct node node_t;

struct node {
    char *word;              /* distinct word */
    int freq;                /* frequency of word */
    node_t *left;            /* left subtree of node */
    node_t *right;           /* right subtree of node */
};
```
Inserting into previous example

Notation:
node("Mercury") refers to the node_t structure containing the word "Mercury"

Recall:
typedef struct node node_t;

struct node {
    char *word;
    int freq;
    node_t *left;
    node_t *right;
};

<table>
<thead>
<tr>
<th>Fn</th>
<th>Calls</th>
<th>Arg1</th>
<th>Arg2</th>
<th>Returns to</th>
<th>Return val</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>insert_in_order</td>
<td>NULL</td>
<td>&quot;Mercury&quot;</td>
<td>tree</td>
<td>&amp;node(&quot;Mercury&quot;)</td>
</tr>
</tbody>
</table>
Inserting into previous example

Notation:
node("Mercury") refers to the node_t structure containing the word “Mercury”

Recall:
typedef struct node node_t;

struct node {
    char *word;
    int freq;
    node_t *left;
    node_t *right;
};

<table>
<thead>
<tr>
<th>Fn</th>
<th>Calls</th>
<th>Arg1</th>
<th>Arg2</th>
<th>Returns to</th>
<th>Return val</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>insert_in_order</td>
<td>&amp;node(&quot;Mercury&quot;)</td>
<td>“Venus”</td>
<td>tree</td>
<td>&amp;node(&quot;Mercury&quot;)</td>
</tr>
<tr>
<td>iio</td>
<td>iio</td>
<td>NULL</td>
<td>“Venus”</td>
<td>node(&quot;Mercury&quot;).right</td>
<td>&amp;node(&quot;Venus&quot;)</td>
</tr>
</tbody>
</table>
Inserting into previous example

Notation:
node("Mercury") refers to the node_t structure containing the word “Mercury”

Recall:
typedef struct node node_t;

struct node {
    char *word;
    int freq;
    node_t *left;
    node_t *right;
};

<table>
<thead>
<tr>
<th>Fn</th>
<th>Calls</th>
<th>Arg1</th>
<th>Arg2</th>
<th>Returns to</th>
<th>Return val</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>insert_in_order</td>
<td>&amp;node(&quot;Mercury&quot;)</td>
<td>“Uranus”</td>
<td>tree</td>
<td>&amp;node(&quot;Mercury&quot;)</td>
</tr>
<tr>
<td>iio</td>
<td>iio</td>
<td>&amp;node(&quot;Venus&quot;)</td>
<td>“Uranus”</td>
<td>node(&quot;Mercury&quot;) .right</td>
<td>&amp;node(&quot;Venus&quot;)</td>
</tr>
<tr>
<td>iio</td>
<td>iio</td>
<td>&amp;node(&quot;Saturn&quot;)</td>
<td>“Uranus”</td>
<td>node(&quot;Venus&quot;) .left</td>
<td>&amp;node(&quot;Saturn&quot;)</td>
</tr>
<tr>
<td>iio</td>
<td>iio</td>
<td>NULL</td>
<td>“Uranus”</td>
<td>node(&quot;Saturn&quot;) .right</td>
<td>&amp;node(&quot;Uranus&quot;)</td>
</tr>
</tbody>
</table>
void print_out_tree(node_t *tree);
/
* Print out the contents of each node in the tree
* rooted at *tree
* 
* Perform an in-order traversal of the tree:
* Call print_out_tree recursively to print out the left
* subtree of *tree before tree->data and the right
* subtree of *tree after tree->data
 */

typedef struct node node_t;

struct node {
    char *word;            /* distinct word */
    int freq;                /* frequency of word */
    node_t *left;            /* left subtree of node */
    node_t *right;           /* right subtree of node */
};
Freeing the tree

```
void free_tree(node_t *tree);
/*
 * Free the memory used by the nodes rooted at *tree.
 *
 * Perform a **post-order traversal** of the tree:
 * Call free_tree recursively to free the left and right
 * subtrees of *tree before freeing the node *tree itself
 */
```

typedef struct node node_t;

```
struct node {
    char *word; /* distinct word */
    int freq; /* frequency of word */
    node_t *left; /* left subtree of node */
    node_t *right; /* right subtree of node */
};
```
Other types of trees

- More complex operations can be developed to keep a tree “balanced”
- The degree of bifurcation (maximum number of possible children per node) does not need to be limited to two
Where to from here?

- Read Moffat on Binary Search Trees (Ch 10.3)
- Week 7
  - Function Calling & Stack Frames
  - Files & File Operations
- Week 8
  - Number Representation & Bit Operations
  - Review of Weeks 1 – 8
- Weeks 9 – 12
  - Graph algorithms
  - Recursion
  - Complexity
  - Searching & Sorting