SEARCHING AND TREES

- COMP1927 Computing 17x1
- Sedgewick Chapters 5, 12
SEARCHING (CONT)

Searching is a very important/frequent operation. Several approaches have been developed:

- $O(n)$ ... linear scan (search technique of last resort)

- $O(\log n)$ ... binary search, search trees (trees also have other uses)

- $O(1)$ ... hash tables (only $O(1)$ under optimal conditions)
SEARCHING (CONT)

Linear structures: arrays, linked lists
Arrays = random access.
Lists = sequential access.

<table>
<thead>
<tr>
<th></th>
<th>Array</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted</td>
<td>$O(n)$ (linear scan)</td>
<td>$O(n)$ (linear scan)</td>
</tr>
<tr>
<td>Sorted</td>
<td>$O(\log n)$ (binary search)</td>
<td>$O(n)$ (linear scan)</td>
</tr>
</tbody>
</table>
SEARCHING

- Storing and searching sorted data:
  - Dilemma: Inserting into a sorted sequence
    - Finding the insertion point on an array – $O(\log n)$ but then we have to move everything along to create room for the new item
    - Finding insertion point on a linked list $O(n)$ but then we can add the item in constant time.
  - Can we get the best of both worlds?
**TREE TERMINOLOGY**

Trees are branched data structures

- consisting of *nodes (vertices)* and *links (edges)*, with no cycles
- each node contains a *data* value (or key + data)
- each node has links to $\leq k$ other nodes  ($k=2$ below)
TREES

- Trees can be viewed as a set of nested structures: each node has $k$ possibly empty subtrees.
Uses of Trees

- Trees are used in many contexts, e.g. representing hierarchical data structures (e.g. expressions)
- Efficient searching (e.g. sets, symbol tables, ...)

![Diagram of a Search Tree](image1)

![Diagram of an Expression Tree](image2)
TREE TERMINOLGY

- **Level** of a node in a tree (or depth) is one higher than the level of its parent
  - Depth of the root is 0
- We call the length of the longest path from the root to a node the **height** of a tree
SPECIAL PROPERTIES OF SOME TREES

- **M-ary tree**: each internal node has exactly M children

- **Ordered tree**: order of the children at every node is specified through constraints on the data/keys in the nodes

- **Balanced tree**: a tree with properties that
  - #nodes in left subtree = #nodes in right subtree
  - this property applies over all nodes in the tree
BINARY TREES

For much of this course, we focus on binary trees

A binary tree (simplest type of M-ary tree) is defined recursively, as being either:

- empty (contains no nodes)
- consisting of a node, with two sub-trees
  - each node contains a value
  - the left and right sub-trees are binary trees
**Binary Trees: Properties**

- A binary tree with $n$ nodes has a height of
  - at most
    - $n-1$ (if degenerate) (an unbalanced tree, where for each parent node, there is only one child node)
  - at least
    - $\lfloor \log_2(n) \rfloor$ (if balanced)
BINARY SEARCH TREE (BST)

- A BST is an **ordered binary tree** that has:
  - all values in left sub-tree being less than root
  - all values in right sub-tree are greater than root
  - this property applies over all nodes in the tree
  - each node is the root of 0, 1 or 2 sub-trees

Three binary search trees
**Binary Trees**

Shape of tree is determined by the order of insertion

![Balanced Tree](image)

![Non-balanced Tree](image)
**Binary Search Trees**

Depth of tree = max path length from root to leaf

Depth of tree with $n$ nodes:  \( \min = \lfloor \log_2 n \rfloor, \quad \max = n - 1 \)

Height balanced tree:  \( \forall \) nodes, \( \text{depth(left subtree)} \cong \text{depth(right subtree)} \)

Time complexity of tree algorithms is typically \( O(\text{depth}) \)
**Exercise: Insertion into BSTs**

- For each of the sequences below start from an initially empty binary search tree
  - show the tree resulting from inserting the values in the order given
  - What is the height of each tree?
- (a) 4 2 6 5 1 7 3
- (b) 5 3 6 2 4 7 1
- (c) 1 2 3 4 5 6 7
TREES: TRAVERSAL

- For trees, several well-defined visiting orders exist:
  - Depth first traversals
    - preorder (NLR) ... visit root, then left subtree, then right subtree
    - inorder (LNR) ... visit left subtree, then root, then right subtree
    - postorder (LRN) ... visit left subtree, then right subtree, then root
  - Breadth-first traversal or level-order ... visit root, then all its children, then all their children
EXAMPLE OF TRAVERSALS ON A BINARY TREE

- Pre-Order: 4 2 1 3 8 6 9
- In-Order: 1 2 3 4 6 8 9
- Post-Order: 1 3 2 6 9 8 4
- Level-Order: 4 2 8 1 3 6 9
**Representing BSTs**

A binary search tree is a generalization of a linked list:

- nodes are a structure with two links to nodes
- empty trees are NULL links

```c
typedef struct treenode *Treelink;

struct treenode {
    int data;
    Treelink left, right;
};
```
REPRESENTING BSTs

Abstract data vs concrete data
Binary Search Trees

Operations on BSTs:

- traverse(TreeLink, (*visit)) ... traverse tree
- insert(TreeLink, Item) ... add new item to tree via key
- delete(TreeLink, Item) ... remove item with specified key from tree
- search(TreeLink, Item) ... find item containing key in tree
- height(TreeLink) ... compute depth of tree
- nodes(TreeLink) ... count #nodes in tree
- plus, "bookeeping" ... new(), dispose(), show(), empty(), ...

Notes: keys are unique (not technically necessary)
**Binary Search Trees**

**Traversal (with parameterised visit option)**

```c
void traverse(TreeLink t, void (*visit)(Item)) {
    if (t == null) return;
    (*visit)(t);  // NLR traversal
    traverse(t->left, visit);
    // put "visit data" here for LNR
    traverse(t->right, visit);
    // put "visit data" here for LRN
}
```
SEARCHING IN BSTs

- Recursive version

// Returns non-zero if item is found, // zero otherwise
int search(TreeLink n, Item i){
    int result;
    if(n == NULL){
        result = 0;
    }else if(i < n->data){
        result = search(n->left,i);
    }else if(i > n->data)
        result = search(n->right,i);
    }else{ // you found the item
        result = 1;
    }
    return result;
}

* Exercise: Try writing an iterative version
INSERTION INTO A BST

- Cases for inserting value V into tree T:
  - T is empty, make new node with V as root of new tree
  - root node contains V, tree unchanged (no dupes)
  - V < value in root, insert into left subtree (recursive)
  - V > value in root, insert into right subtree (recursive)

- Non-recursive insertion of V into tree T:
  - search to location where V belongs, keeping parent
  - make new node and attach to parent
  - whether to attach L or R depends on last move
Insertion into a BST

//Returns the root of the tree
//Inserts duplicates on the left hand side of tree
Treelink insertRec (Treelink tree, TreeItem item) {
    if(tree == NULL) {
        Treelink newNode = createNode(item);
        return newNode; //now the root of the tree
    } else {
        if(item <= tree->item) {
            tree->left = insertRec(tree->left, item); //
        } else {
            tree->right = insertRec(tree->right, item);
        }
    }
    return tree;
}
Deletion from BSTs

- Insertion into a binary search tree is easy:
  - find location in tree where node to be added
  - create node and link to parent

- Deletion from a binary search tree is harder:
  - find the node to be deleted and its parent
  - unlink node from parent and delete
  - replace node in tree by ... ???
DELETION FROM BSTS...

- Easy option ... don't delete; just mark node as deleted
  - future searches simply ignore marked nodes
- If we want to delete, three cases to consider ...
  - zero subtrees ... unlink node from parent
  - one subtree ... replace node by child
  - two subtrees ... two children; one link in parent
DELETION FROM BSTS

- Case 1: value to be deleted is a leaf (zero subtrees)

delete k ...
DELETION FROM BSTS

- Case 1: value to be deleted is a leaf (zero subtrees)
  
  deleted k ...
DELETION FROM BSTS

- Case 2: value to be deleted has one subtree

delete p ...
DELETION FROM BSTS

- Case 2: value to be deleted has one subtree
DELETION FROM BSTS

- Case 3a: value to be deleted has two subtrees
- Replace deleted node by its immediate successor
  - The smallest (leftmost) node in the right subtree

delete m ...
DELETION FROM BSTS

- Case: value to be deleted has two subtrees
**Binary Search Tree Properties**

- **Cost for searching/deleting:**
  - Worst case: key is not in BST – search the height of the tree
    - Balanced trees – $O(\log_2 n)$
    - Degenerate trees – $O(n)$

- **Cost for insertion:**
  - Always traverse the height of the tree
    - Balanced trees – $O(\log_2 n)$
    - Degenerate trees – $O(n)$
New Binary Search Tree

// Item, Key, Node, Link, Tree types as before
#define key(it) ((it).key)
// operations on keys
#define cmp(k1,k2) ((k1) - (k2))
#define lt(k1,k2) (cmp(k1,k2) < 0)
#define eq(k1,k2) (cmp(k1,k2) == 0)
#define gt(k1,k2) (cmp(k1,k2) > 0)
// standard tree operations
Tree newTree();
Tree insert(Tree, Item);
Tree delete(Tree, Key);
int find(Tree, Key);
I NSERTION U SING new T REE ADT

// more standard tree operations
void dropTree(Tree);
void showTree(Tree);
int depth(Tree);
int nnodes(Tree); // aka size()

// functions internal to ADT
Link rotateR( Link );
Link rotateL( Link );
Tree insertAtRoot( Tree, Item );
Tree insertRandom( Tree, Item );
Insert At Root - Rotate Operations
**Insert At Root - Rotate Operations**

```c
Link rotateR(Link n1) {
    if (n1 == NULL) return NULL;
    Link n2 = n1->left;
    if (n2 == NULL) return n1;
    n1->left = n2->right;
    n2->right = n1;
    return n2;
}
```

Left rotation is similar with n1/n2 and left/right switched.
INSERTION AT ROOT

- Previous description of BSTs inserted at leaves.
- Different approach: insert new value at root.
- Method for inserting at root (recursive):
  - base case:
    - tree is empty; make new node and make it root
  - recursive case:
    - insert new node as root of L/R subtree
    - lift new node to root by R/L rotation
**Insertion at Root**

1. Initial tree:
   - Root: 6
   - Left child: 2
     - Left child: 1
     - Right child: 3
   - Right child: 8
     - Left child: 7
     - Right child: 9

2. Insertion of 4:
   - New node: 4
   - Link 4 to 2 as the left child.

3. Rotating left from 2:
   - New tree configuration.

4. Rotating left from 2 again:
   - Final tree configuration.
Tree insertAtRoot(Tree t, Item it) {
    if (t == NULL) return newNode(it);
    int diff = cmp(key(it), key(t->value));
    if (diff == 0) t->value = it;
    else if (diff < 0) {
        t->left = insertAtRoot(t->left, it);
        t = rotateR(t);
    }
    else if (diff > 0) {
        t->right = insertAtRoot(t->right, it);
        t = rotateL(t);
    }
    return t;
}