**Searching**

An extremely common application in computing

- given a (large) collection of *items* and a *key* value
- find the item(s) in the collection containing that key
  - item = (key, val₁, val₂, …) *(i.e. a structured data type)*
  - key = value used to distinguish items *(e.g. student ID)*

Applications: Google, databases, …

---

**Tree Data Structures**

**Trees**

Trees are connected graphs

- consisting of nodes and edges *(called *links*)*, with no cycles *(no "up-links")*
- each node contains a *data value* *(or key+data)*
- each node has *links* to ≤ *k* other child nodes *(*k*=2 below)*

---

**Trees**

Trees are used in many contexts, e.g.

- representing hierarchical data structures *(e.g. expressions)*
- efficient searching *(e.g. sets, symbol tables, …)*

---

**Searching**

Maintaining the order in sorted arrays and files is a costly operation.

*Search trees* are as efficient to search but more efficient to maintain.

Example: the following tree corresponds to the sorted array `{2, 5, 10, 12, 14, 17, 20, 24, 29, 30, 31, 32}`:
Trees

Trees can be used as a data structure, but also for illustration.

E.g. showing evaluation of a prefix arithmetic expression

**Binary trees** ($k=2$ children per node) can be defined recursively, as follows:

A **binary tree** is either
- empty (contains no nodes)
- consists of a node, with two subtrees
  - node contains a value
  - left and right subtrees are binary trees

*Other special kinds of tree*

- **$m$-ary tree**: each internal node has exactly $m$ children
- **Ordered tree**: all left values < root, all right values > root
- **Balanced tree**: has minimal height for a given number of nodes
- **Degenerate tree**: has maximal height for a given number of nodes

**Search Trees**

**Binary Search Trees**

*Binary search trees* (or **BSTs**) have the characteristic properties

- each node is the root of 0, 1 or 2 subtrees
- all values in any left subtree are less than root
- all values in any right subtree are greater than root
- these properties applies over all nodes in the tree

(perfectly) **balanced trees** have the properties

- #nodes in left subtree = #nodes in right subtree
- this property applies over all nodes in the tree

**Operations on BSTs:**

- `insert(Tree, Item)` … add new item to tree via key
- `delete(Tree, Key)` … remove item with specified key from tree
- `search(Tree, Key)` … find item containing key in tree
- plus, “bookkeeping” … new(), free(), show(), …

**Notes:**

- nodes contain **Items**; we just show **Item.key**
- keys are unique (not technically necessary)
Examples of binary search trees:

![Balanced Tree vs. Non-balanced Tree](image)

Shape of tree is determined by order of insertion.

---

**Binary Search Trees**

- **Level** of node = path length from root to node
- **Height** (or: depth) of tree = max path length from root to leaf

---

**Height-balanced tree**: ∀ nodes: height(left subtree) = height(right subtree)

Time complexity of tree algorithms is typically $O(\text{height})$

---

**Exercise #1: Insertion into BSTs**

For each of the sequences below

- start from an initially empty binary search tree
- show tree resulting from inserting values in order given

(a) 4 2 6 5 1 7 3
(b) 6 5 2 3 4 7 1
(c) 1 2 3 4 5 6 7

Assume new values are always inserted as new leaf nodes

---

**Representing BSTs**

Binary trees are typically represented by node structures

- containing a value, and pointers to child nodes

Most tree algorithms move down the tree.
If upward movement needed, add a pointer to parent.

---

**Typical data structures for trees**

// a Tree is represented by a pointer to its root node
typedef struct Node *Tree;

// a Node contains its data, plus left and right subtrees
typedef struct Node {
    int  data;
    Tree left, right;
} Node;

// some macros that we will use frequently
#define data(tree)  ((tree)->data)
#define left(tree)  ((tree)->left)
#define right(tree) ((tree)->right)

We ignore items data in Node is just a key
Abstract data vs concrete data ...

Tree Algorithms

Searching in BSTs

Most tree algorithms are best described recursively:

TreeSearch(tree, item):

<table>
<thead>
<tr>
<th>Input</th>
<th>tree, item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>true if item found in tree, false otherwise</td>
</tr>
</tbody>
</table>

| if | tree is empty then |
| return | false |
| else if | item < data(tree) then |
| return | TreeSearch(left(tree), item) |
| else if | item > data(tree) then |
| return | TreeSearch(right(tree), item) |
| else | // found |
| return | true |
| end if |

Insertion into BSTs

Insert an item into appropriate subtree:

insertAtLeaf(tree, item):

<table>
<thead>
<tr>
<th>Input</th>
<th>tree, item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>tree with item inserted</td>
</tr>
</tbody>
</table>

| if | tree is empty then |
| return | new node containing item |
| else if | item < data(tree) then |
| return | insertAtLeaf(left(tree), item) |
| else if | item > data(tree) then |
| return | insertAtLeaf(right(tree), item) |
| else | // avoid duplicates |

Tree Traversal

Iteration (traversal) on ...

- Lists ... visit each value, from first to last
- Graphs ... visit each vertex, order determined by DFS/BFS/...

For binary Trees, several well-defined visiting orders exist:

- 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
- level-order ... visit root, then all its children, then all their children

Exercise #2: Tree Traversal

Show NLR, LNR, LRN traversals for the following tree:

Consider "visiting" an expression tree like:

NLR: + * 1 3 - * 5 7 9  (prefix-order: useful for building tree)
LNR: 1 * 3 + 5 * 7 - 9  (infix-order: "natural" order)
LRN: 1 3 * 5 7 * 9 - +  (postfix-order: useful for evaluation)
Level: + * - 1 3 9 5 7  (level-order: useful for printing tree)
Exercise #3: Non-recursive traversals

Write a non-recursive preorder traversal algorithm.
Assume that you have a stack ADT available.

```plaintext
showBSTreePreorder(t):
| Input tree t |
| push t onto new stack S |
| while stack is not empty do |
| | t=pop(S) |
| | print data(t) |
| | if left(t) is not empty then |
| | | push left(t) onto S |
| | | end if |
| | if right(t) is not empty then |
| | | push right(t) onto S |
| | | end if |
| end while
```

Joining Two Trees

An auxiliary tree operation ...

Tree operations so far have involved just one tree.

An operation on two trees: \( t = \text{joinTrees}(t_1, t_2) \)

- Pre-conditions:
  - takes two BSTs; returns a single BST
  - \( \max(key(t_1)) < \min(key(t_2)) \)
- Post-conditions:
  - result is a BST (i.e. fully ordered)
  - containing all items from \( t_1 \) and \( t_2 \)

Joining Two Trees

Implementation of tree-join:

```plaintext
joinTrees(t_1,t_2):
| Input trees t_1,t_2 |
| Output t_1 and t_2 joined together |
| if t_1 is empty then return t_1 |
| else if t_2 is empty then return t_2 |
| else |
| curr=t_2, parent=NULL |
| while left(curr) is not empty do // find min element in t_2 |
| | parent=curr |
| | curr=left(curr) |
| end while |
| if parent=NULL then |
| | left(parent)=right(curr) // unlink min element from parent |
| | right(curr)=t_2 |
| end if |
| left(curr)=t_1 |
| return curr // curr is new root |
| end if
```
Exercise #4: Joining Two Trees

Join the trees

Deletion from BSTs

Insertion into a binary search tree is easy.
Deletion from a binary search tree is harder.

Four cases to consider …

- empty tree … new tree is also empty
- zero subtrees … unlink node from parent
- one subtree … replace by child
- two subtrees … replace by successor, join two subtrees

Deletion from BSTs

Case 2: item to be deleted is a leaf (zero subtrees)

Just delete the item

Case 3: item to be deleted has one subtree

Replace the item by its only subtree

Case 4: item to be deleted has two subtrees

Just delete the item
Version 1: right child becomes new root, attach left subtree to min element of right subtree

Case 4: item to be deleted has two subtrees

Version 2: join left and right subtree

Pseudocode (version 2):

```plaintext
TreeDelete(t, item):
    Input   tree t, item
    Output  t with item deleted
    if t is not empty then
        // nothing to do if tree is empty
        if item < data(t) then
            left(t) = TreeDelete(left(t), item)
        else if item > data(t) then
            right(t) = TreeDelete(right(t), item)
        else
            // node 't' must be deleted
            if left(t) and right(t) are empty then
                new = empty tree  // 0 children
            else if left(t) is empty then
                new = right(t)    // 1 child
            else if right(t) is empty then
                new = left(t)     // 1 child
            else
                new = joinTrees(left(t), right(t))  // 2 children
            end if
            free memory allocated for t
            t = new
        end if
    end if
    return t
```

Balanced BSTs

Balanced Binary Search Trees

Goal: build binary search trees which have

- minimum height $\Rightarrow$ minimum worst case search cost

Perfectly balanced tree with $N$ nodes has

- $\text{abs}($nodes(LeftSubtree) - nodes(RightSubtree)) $< 2$, for every node
- height of $\log_2 N$ $\Rightarrow$ worst case search $O(\log N)$
Three strategies to improving worst case search in BSTs:

- **randomise** — reduce chance of worst-case scenario occurring
- **amortise** — do more work at insertion to make search faster
- **optimise** — implement all operations with performance bounds

**Operations for Rebalancing**

To assist with rebalancing, we consider new operations:

**Left rotation**
- move right child to root; rearrange links to retain order

**Right rotation**
- move left child to root; rearrange links to retain order

**Insertion at root**
- each new item is added as the new root node

**Tree Rotation**

In tree below: $t_1 < n_2 < t_2 < n_1 < t_3$

Algorithm for right rotation:

\[
\text{rotateRight}(n_1):
\begin{align*}
\text{Input} & \quad \text{tree } n_1 \\
\text{Output} & \quad n_1 \text{ rotated to the right}
\end{align*}
\]

\[
\begin{array}{l}
\text{if } n_1 \text{ is empty } \vee \text{ left}(n_1) \text{ is empty then} \\
\quad \text{return } n_1 \\
\text{end if} \\
\quad n_2 = \text{left}(n_1) \\
\quad \text{left}(n_1) = \text{right}(n_2) \\
\quad \text{right}(n_2) = n_1 \\
\quad \text{return } n_2
\end{array}
\]

**Exercise #5: Tree Rotation**

Consider the tree $t$:

Show the result of $\text{rotateRight}(t)$

... Tree Rotation

Method for rotating tree $T$ right:

- $N_1$ is current root; $N_2$ is root of $N_1$'s left subtree
- $N_1$ gets new left subtree, which is $N_2$'s right subtree
- $N_1$ becomes root of $N_2$'s new right subtree
- $N_2$ becomes new root

Left rotation: swap left/right in the above.

Cost of tree rotation: $O(1)$
Exercise #6: Tree Rotation

Write the algorithm for left rotation

rotateLeft(n_2):
\[\text{Input } \text{tree } n_2\]
\[\text{Output } n_2 \text{ rotated to the left}\]
\[\text{if } n_2 \text{ is empty } \lor \text{right}(n_2) \text{ is empty } \text{ then}\]
\[\text{return } n_2\]
\[\text{end if}\]
\[n_1=\text{right}(n_2)\]
\[\text{right}(n_2)=\text{left}(n_1)\]
\[\text{left}(n_1)=n_2\]
\[\text{return } n_1\]

Insertion at Root

Previous description of BSTs inserted at leaves.

Different approach: insert new item at root.

Potential disadvantages:

- large-scale rearrangement of tree for each insert

Potential advantages:

- recently-inserted items are close to root
- low cost if recent items more likely to be searched

Exercise #7: Insertion at Root

Consider the tree t:

Show the result of insertAtRoot(t,24)

Analysis of insertion-at-root:
Rebalancing Trees

An approach to balanced trees:

- Insert into leaves as for simple BST
- Periodically, rebalance the tree

Question: how frequently/when/how to rebalance?

NewTreeInsert(tree, item):

<table>
<thead>
<tr>
<th>Input</th>
<th>tree, item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>tree with item randomly inserted</td>
</tr>
</tbody>
</table>

```
t = insertAtLeaf(tree, item)
if \#nodes(t) mod k = 0 then
t = rebalance(t)
end if
return t
```

E.g. rebalance after every 20 insertions ⇒ choose k=20

Note: To do this efficiently we would need to change tree data structure and basic operations:

```
typedef struct Node {
  int data;
  int nnodes; // #nodes in my tree
  Tree left, right; // subtrees
} Node;
```

How to rebalance a BST? Move median item to root.

Implementation of rebalance:

```
rebalance(t):
<table>
<thead>
<tr>
<th>Input</th>
<th>tree t with n nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>t rebalanced</td>
</tr>
</tbody>
</table>

if n ≥ 3 then
  t = partition(t, \[n/2\]) // put node with median key at root
  left(t) = rebalance(left(t)) // then rebalance each subtree
  right(t) = rebalance(right(t))
end if
return t
```

For tree with \(N\) nodes, indices are 0..\(N-1\)

Partition: moves \(i\)th node to root
**Implementation of partition operation:**

\[ \text{partition}(\text{tree}, i) : \]

**Input** tree with \( n \) nodes, index \( i \)

**Output** tree with \( i \)th item moved to the root

\[
\begin{align*}
\text{m} &= \text{size}(\text{left}(\text{tree})) \\
\text{if } i &< m \text{ then} \\
\text{left}(\text{tree}) &= \text{partition}(\text{left}(\text{tree}), i) \\
\text{tree} &= \text{rotateRight}(\text{tree}) \\
\text{else if } i &> m \text{ then} \\
\text{right}(\text{tree}) &= \text{partition}(\text{right}(\text{tree}), i-m-1) \\
\text{tree} &= \text{rotateLeft}(\text{tree}) \\
\text{end if} \\
\text{return } \text{tree}
\end{align*}
\]

Note: size(tree) = \( n \), size(left(tree)) = \( m \), size(right(tree)) = \( n-m-1 \) (why -1?)

---

**Exercise #8: Partition**

Consider the tree \( t \):

![Tree Diagram]

Show the result of \( \text{partition}(t, 3) \)

![Result Diagram]

---

**Application of BSTs: Sets**

Trees provide efficient search.

Sets require efficient search

- to find where to insert/delete
- to test for set membership

Logical to implement a set ADT via BSTree

---

**Application of BSTs: Sets**

Assuming we have Tree implementation

- which precludes duplicate key values
- which implements

then Set implementation is

- SetInsert(Set, Item) = TreeInsert(Tree, Item)
- SetDelete(Set, Item) = TreeDelete(Tree, Item.Key)
- SetMember(Set, Item) = TreeSearch(Tree, Item.Key)

---

**Rebalancing Trees**

Analysis of rebalancing: visits every node \( \Rightarrow O(N) \)

Cost means not feasible to rebalance after each insertion.

When to rebalance? … Some possibilities:

- after every \( k \) insertions
- whenever "imbalance" exceeds threshold

Either way, we tolerate worse search performance for periods of time.

Does it solve the problem? … Not completely \( \Rightarrow \) Solution: real balanced trees (next week)
Concrete representation:

```c
#include <BSTree.h>

typedef struct SetRep {
   int   nelems;
   Tree  root;
} SetRep;

Set newSet() {
   Set S = malloc(sizeof(SetRep));
   assert(S != NULL);
   S->nelems = 0;
   S->root = newTree();
   return S;
}
```

Summary

- Binary search tree (BST) data structure
- BST insertion and deletion
- Other tree operations
  - tree rotation
  - tree partition
  - joining trees

Suggested reading:
- Sedgewick, Ch.12.5-12.6, 12.8-12.9

Produced: 3 Oct 2017