http://www.cs.unh.edu/~ruml/cs758

2 handouts: slides, asst 5
### Searching

<table>
<thead>
<tr>
<th>Structure</th>
<th>Find</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>List (unsorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>List (sorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array (unsorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array (sorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash table</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary tree (unbalanced)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary tree (balanced)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What about long keys?

Can we detect miss without examining entire key?
trie: test digits of key, branching on values

- some nodes do not hold values
- fixed order
- depth = length
- canonical representation

retrieval

CLRS: ‘trie’ = ‘radix tree’
Wikipedia: ‘trie’ ≠ ‘radix tree’
Sedgewick: ‘trie’ ≠ ‘digital search tree’

duplicate keys?

what's their weakness?
Wikipedia ‘radix tree’ = ‘radix trie’ = ‘patricia trie’: compressed trie, every internal node has at least two leaves beneath.

Sedgewick: ‘digital search tree’: value at every node, just like binary trees except test bits instead of full compare.
<table>
<thead>
<tr>
<th>Structure</th>
<th>Find</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>List (unsorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>List (sorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array (unsorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array (sorted)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash table</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary tree (unbalanced)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary tree (balanced)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trie</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Given a list of records, which may contain duplicates, return a list containing each record at most once.
Break

- asst 4
- asst 5
- schedule
Binary Search Trees
Invariant: For each node $n$, all nodes in $n$’s left subtree $\leq n$, all nodes in right subtree $\geq n$.

insert ($n$)
1. $n$’s parent $\leftarrow$ find-parent($n$, root, nil)
2. if parent is nil
3. root $\leftarrow n$
4. else
5. if $n$ should be before parent
6. parent’s left child $\leftarrow n$
7. else
8. parent’s right child $\leftarrow n$
Find-Parent Specification

Need:
1) find-parent returns nil iff empty tree,
2) find-parent returns leaf node \( p \) directly adjacent to \( n \). I.e., either

\[
\text{predecessor}(p) \leq n \leq p \quad \text{or} \quad p \leq n \leq \text{successor}(p)
\]

so that attaching \( n \) to \( p \) preserves BST ordering.
Property 1

find-parent($n$, curr, parent)
9. if curr doesn’t exist
10. return parent
11. if $n$ should be before curr
12. return find-parent($n$, curr’s left child, curr)
13. else
14. return find-parent($n$, curr’s right child, curr)

1) called as find-parent($n$, root, nil), so only way return value is nil is if curr=root=nil (ie, empty tree).

2) Invariant: if we replace curr with $n$ (and ignore curr’s children), $n$ is in proper place in the tree.
Property 2

Invariant: If we replace curr with \( n \) (and ignore curr’s children), \( n \) is in proper place in the tree.

Initialization: curr is root. Replacing root with \( n \) has \( n \) in correct place with respect to the zero remaining nodes.

Maintenance: We were the correct position if curr were replaced by \( n \). To set up next iteration, we move to the correct side of curr. This preserves BST ordering with respect to curr as we move below it and it (and its other child) enters the ‘active tree’.

Termination: We terminate when we move off the correct side of a leaf. The BST invariant holds everywhere if that null pointer were replaced by \( n \) because there are no children to ignore. Thus, we know that:

\[
\text{predecessor}(p) \leq n \leq p \quad \text{or} \quad p \leq n \leq \text{successor}(p)
\]
Algorithms
Beyond craftsmanship lies invention, and it is here that lean, spare, fast programs are born. Almost always these are the result of strategic breakthrough rather than tactical cleverness. Sometimes the strategic breakthrough will be a new algorithm, such the Cooley-Tukey Fast Fourier transform or the substitution of an $n \log n$ sort for an $n^2$ set of comparisons.

Much more often, strategic breakthrough will come from redoing the representation of the data or tables. This is where the heart of a program lies. Show me your flowchart and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won’t usually need your flowchart; it’ll be obvious.

— Fred Brooks, 1974 (lead on IBM System/360, Turing Award 1999)
Smart data structures and dumb code works a lot better than the other way around.

— Guy Steele, 2002 (ACM Fellow, inventor of Scheme, editor of *The Hacker’s Dictionary*)
Types of Algorithms

- divide and conquer
- dynamic programming
- greedy
- backtracking
- (reduction)
Dynamic Programming
Fibonacci Numbers

\[ F_n = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
F_{n-1} + F_{n-2} & \text{for } n \geq 2 
\end{cases} \]

What is the complexity of the naive algorithm? How to make this efficient?
Memoization

- recursive decomposition
- polynomial number of subproblems
- cache results in look-up table

one form of *dynamic programming*
■ What’s still confusing?
■ What question didn’t you get to ask today?
■ What would you like to hear more about?

Please write down your most pressing question about algorithms and put it in the box on your way out.

Thanks!