## Searching

### Hash Tables
- List (unsorted)
- List (sorted)
- Array (unsorted)
- Array (sorted)
- Heap
- Hash table
- Binary tree (unbalanced)
- Binary tree (balanced)

### Binary Search Trees

<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>
Hash Tables
Hash Functions

$h : key \rightarrow 0..m - 1$

1. mediocre is easy, good takes effort
2. want time (at most) linear in key size
3. perfect hashing is possible (and efficient) if keys known
   - linear time to construct, linear space to store
4. minimal perfect hashing is possible!
Hash Functions

\[ h : \text{key} \rightarrow 0..m - 1 \]

1. mediocre is easy, good takes effort
2. want time (at most) linear in key size
3. perfect hashing is possible (and efficient) if keys known
   - linear time to construct, linear space to store
4. minimal perfect hashing is possible!

bad news:

- if \( |\text{keys}| \geq m \), there must be collisions
- if \( |\text{keys}| \geq n \cdot m \), then \( \exists \) set of \( n \) that map to same bin
Hash Functions

Desiderata:

- make collisions unlikely
  - spread keys across all hashes
  - for each key, each hash equally likely
- similar keys get different hashes
  - all bits of key affect the hash
  - every bit of key affects every bit of hash
- no input always gives worst-case behavior
- fast to compute
- low memory requirement
- easy to implement
1. \( \text{hash} \leftarrow 0 \)

2. for each byte of key

3. \( \text{hash} \leftarrow ((\text{hash} \times \text{multiplier}) + \text{byte}) \mod \text{table-size} \)

want \( \text{multiplier} \) to smear bits, not shift them (to avoid interaction with table size)

\( \text{multiplier} = 31 \) or 127
Another Hash

assume we have a table of 256 random integers

1. \( \text{hash} \leftarrow 0 \)
2. for each byte of key
3. rotate the bits in \( \text{hash} \) by 1
4. \( \text{hash} \leftarrow \text{hash xor table[byte]} \)
5. return \( \text{hash mod table-size} \)

each byte affects all bits
rotate makes order matter

universal class of hash functions: for randomly chosen keys, randomly chosen function from class has \( P(\text{collision}) = 1/m \)

good on average case (over inputs) \( \neq \) good average case on any input
Binary Search Trees
node: data, left, right, parent

What’s the invariant?
if no right child, want lowest ancestor ‘on the right’

\[
\text{succ}(x) \\
1. \text{if right child exists} \\
2. \text{return min under right child} \\
3. \text{else} \\
4. \text{return up}(x)
\]

\[
\text{up}(x) \\
5. \ p \leftarrow x’s \text{ parent} \\
4. \text{if } p \text{ doesn’t exist or } x \text{ is } p’s \text{ left child} \\
5. \text{return } p \\
6. \text{else} \\
7. \text{return up}(p)
\]
insert \((n)\)
1. \(n\)'s parent ← find-parent\((n, \text{root}, \text{nil})\)
2. if parent is nil
3. root ← \(n\)
4. else
5. if \(n\) should be before parent
6. parent's left child ← \(n\)
7. else
8. parent's right child ← \(n\)

find-parent\((n, \text{curr}, \text{parent})\)
9. if curr doesn’t exist
10. return parent
11. if \(n\) should be before curr
12. return find-parent\((n, \text{curr's left child}, \text{curr})\)
13. else
14. return find-parent\((n, \text{curr's right child}, \text{curr})\)
3 cases of delete($n$):

1. no kids: pointer from parent $\leftarrow$ nil
2. 1 kid: substitute child for $n$ at parent
3. 2 kids: let successor be $s$.
   
   note $s$ is in $n$’s right subtree and has no left child.
   
   (a) $s$ takes $n$’s place at parent
   (b) $n$’s left subtree becomes $s$’s
   (c) somehow, rest of $n$’s right subtree becomes $s$’s...

will split 3(c) into 2 cases...
4 cases of delete\( (n) \):

1. no kids or no left child: substitute right subtree at parent
2. no right child: substitute left subtree at parent

   now we have the hard 2-kids cases:

3. successor \( s \) is \( n \)'s right child:
   (a) substitute \( s \) for \( n \)
   (b) add \( n \)'s left subtree as \( s \)'s left subtree

4. successor \( s \) is deeper:
   (a) substitute \( s \)'s right subtree for \( s \)
   (b) add \( n \)'s right subtree as \( s \)'s right subtree
   (c) as above, substitute \( s \) for \( n \)
   (d) as above, add \( n \)'s left subtree as \( s \)'s left subtree
Moving Subtrees

put new where old was:

substitute(old, new)
1. if old’s parent is nil
2. root ← new
3. else
4. if old is parent’s left child
5. parent’s left child ← new
6. else, parent’s right child ← new
7. if new ≠ nil
8. new’s parent ← old’s parent
delete($n$)
1. if $n$ has no left child
2. substitute($n$, $n$’s right subtree)  \textit{case 1}
3. else if $n$ has no right child
4. substitute($n$, $n$’s left subtree)  \textit{case 2}
5. else
6. $s \leftarrow$ min in $n$’s right subtree
7. if $n$ is not $s$’s parent  \textit{case 4}
8. substitute($s$, $s$’s right subtree)
9. $s$’s right subtree $\leftarrow$ $n$’s right subtree
10. $s$’s right child’s parent $\leftarrow s$
11. substitute($n$, $s$)  \textit{cases 3 and 4}
12. $s$’s left subtree $\leftarrow$ $n$’s left subtree
13. $s$’s left child’s parent $\leftarrow s$
Random Deletion/Insertion Behavior

Jeff Eppinger: don’t try this at home!
Jeff Eppinger: don’t try this at home! Delete should alternate between successor and predecessor.

ACM’s 1983 George E. Forsythe Award for best undergraduate student paper

Real solution: balanced trees!
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!