Terminology

- **Abstract Data Type (ADT)**
  - Mathematical description of a “thing” with set of operations on that “thing”; doesn’t specify the details of how it’s done
  - Ex, Stack: You push stuff and you pop stuff
    - Could use an array, could use a linked list

- **Algorithm**
  - A high level, language-independent description of a step-by-step process
    - Ex: Binary search

- **Data structure**
  - A specific family of algorithms & data for implementing an ADT
    - Ex: Linked list stack

- **Implementation of a data structure**
  - A specific implementation in a specific language
**Big Oh’s Family**

- **Big Oh: Upper bound:** \( O( f(n) ) \) is the set of all functions asymptotically less than or equal to \( f(n) \)
  
  - \( g(n) \) is in \( O( f(n) ) \) if there exist constants \( c \) and \( n_0 \) such that
    \[
    g(n) \leq c f(n) \text{ for all } n \geq n_0
    \]

- **Big Omega: Lower bound:** \( \Omega( f(n) ) \) is the set of all functions asymptotically greater than or equal to \( f(n) \)
  
  - \( g(n) \) is in \( \Omega( f(n) ) \) if there exist constants \( c \) and \( n_0 \) such that
    \[
    g(n) \geq c f(n) \text{ for all } n \geq n_0
    \]

- **Big Theta: Tight bound:** \( \theta( f(n) ) \) is the set of all functions asymptotically equal to \( f(n) \)
  
  - Intersection of \( O( f(n) ) \) and \( \Omega( f(n) ) \) (use different \( c \) values)
Common recurrence relations

\[
\begin{align*}
T(n) &= O(1) + T(n-1) & \text{linear} \\
T(n) &= O(1) + 2T(n/2) & \text{linear} \\
T(n) &= O(1) + T(n/2) & \text{logarithmic} \\
T(n) &= O(1) + 2T(n-1) & \text{exponential} \\
T(n) &= O(n) + T(n-1) & \text{quadratic} \\
T(n) &= O(n) + T(n/2) & \text{linear} \\
T(n) &= O(n) + 2T(n/2) & O(n \log n)
\end{align*}
\]

- Solving to a closed form (summary):
  - Ex: \( T(n) = 2 + T(n-1), \ T(1) = 5 \)
  - Expand: \( T(n) = 2 + T(n-1) = 2 + 2 + T(n-2) = \ldots = 2 + 2 + 2 + \ldots + 2 + 5 \)
  - Determine \# of times recurrence was applied to get to the base case; call it \( k \)
  - \( T(n) = 2(k-1) + 5 = 2k + 3 \)
  - Determine \( k \) in terms of \( n \); here \( k = n \); plug into equation
  - \( T(n) = 2n + 3 \)
Binary Heap: Priority Queue DS

- Structure property: A complete binary tree
- Heap ordering property: For every (non-root) node the parent node’s value is less than the node’s value
- Array representation; index starting at 1
- Poor performance for general ‘find’

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Run-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>Place in next available spot; percUp</td>
<td>$O(\log n)$ worst; $O(1)$ expected</td>
</tr>
<tr>
<td>DeleteMin</td>
<td>Remember root value; place last node in root; percDown</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>BuildHeap</td>
<td>Treat array as heap; percDown elements index $\leq$ size/2</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Binary Search Tree: Dictionary ADT

- Structure property: Binary tree; values in left subtree < this node’s value; values in right subtree > this node’s value
- Height $O(\log n)$ if balanced; $O(n)$ if not
- No guarantees on balance

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Run-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find</td>
<td>Check my value against node’s: go left or right</td>
<td>$O(n)$ worst</td>
</tr>
<tr>
<td>Insert</td>
<td>Traverse like in find; create new node there</td>
<td>$O(n)$ worst</td>
</tr>
<tr>
<td>Delete</td>
<td>Traverse like in find; 3 cases: has no children, 1 child or 2 children</td>
<td>$O(n)$ worst</td>
</tr>
</tbody>
</table>
AVL Tree: Dictionary ADT

- **Structure property**: BST
- **Balance property**: $|\text{left.height} - \text{right.height}| \leq 1$
- **Balance guaranteed**: $O(\log n)$ height
- **Perform $O(1)$ rotations to fix balance**: at most one required per insert
- **4 rotation cases**: depend on
  - At what node the imbalance is detected
  - At which of 4 subtrees the insertion was performed, relative to the detecting node

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Run-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find</td>
<td>BST find</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Insert</td>
<td>BST insert, then recurse back up, check for imbalance &amp; perform necessary rotations</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
# B-Tree: Dictionary ADT

- **2 constants:** M & L
  - Internal nodes (except root) have between \([M/2]\) and \(M\) children (inclusive)
  - Leaf nodes have between \([L/2]\) and \(L\) data items (inclusive)
  - Root has between 2 & M children (inclusive); or between 0 & L data items if a leaf
  - Base M & L on disk block size
- **All leaves on same level; all data at leaves**
- **If in child branch, value is \(\geq\) prev key in parent, < next key**
- **Goal:** Shallow tree to reduce disk accesses
- **Height:** \(O(\log_M n)\)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Run-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find</td>
<td>Binary Search to find which child to take on each node; Binary Search in leaf to find data item</td>
<td>(O(\log_2 M \log_M n))</td>
</tr>
<tr>
<td>Insert</td>
<td>Find leaf; insert in sorted order; if overflow, split leaf; if parent overflows, split parent; may need to recursively split all the way to root</td>
<td>(O(L + M \log_M n)) worst (split root) (O(L + \log_2 M \log_M n)) expected</td>
</tr>
<tr>
<td>Delete</td>
<td>Find leaf; remove value, shift others as appropriate; if underflow, adopt and/or merge; may need to merge all the way to the root</td>
<td>(O(L + M \log_M n)) worst (replace root) (O(L + \log_2 M \log_M n)) expected</td>
</tr>
</tbody>
</table>
Hash tables (in general): Dictionary ADT (pretty much)

- Store everything in an array
- To do this, provide a mapping from key to index
  - Ex: “Jean Valjean” → 24601
- Keyspace >> table size; need to deal with ‘collisions’; we consider 2 varieties:
  - Separate Chaining: Linked list at each index
  - Open Addressing: Store all in table; give series of indices
- Keep table size prime
- Define load factor: \( \lambda = \frac{N}{\text{TableSize}} \)
- Rehashing: O(n)
- Great performance (usually)
- Can’t efficiently do findMin, in-order traversal, etc.

Hash table

![hash function: index = h(key)]

TableSize – 1

Key space (e.g., integers, strings)
Hash tables: Separate Chaining

- Each array cell is a ‘bucket’ that can store several items (say, using a linked sort); each (conceptually) boundless
  - $\lambda$: average # items per bucket
  - $\lambda$ can be greater than 1

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Run-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find</td>
<td>Go to list at index, step through until we find correct item or reach the end</td>
<td>$O(\lambda)$ expected, $O(n)$ worst</td>
</tr>
<tr>
<td>Insert</td>
<td>Go to list at index, insert at start</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Delete</td>
<td>Go to list at index, find and delete</td>
<td>$O(\lambda)$ expected, $O(n)$ worst</td>
</tr>
</tbody>
</table>
Hash tables: Open Addressing

- Keep all items directly in table
- ‘Probe’ indices according to $(h(key) + f(i)) \mod \text{TableSize}$
  where $i$ is the # of the attempt (starting at $i=0$)
- Linear probing: $f(i)=i$
  - Will always find a spot if one is available
  - Problem of primary clustering
- Quadratic probing: $f(i)=i^2$
  - Will find space if $\lambda<1/2$ & TableSize is prime
  - Problem of secondary clustering
- Double Hashing: $f(i)=i^*g(key)$
  - Avoids clustering problems (if $g$ is well chosen)
  - $g(key)$ must never evaluate to 0
  - $\lambda=1$ means table is full; no inserts possible

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Run-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find</td>
<td>Probe until found (success) or empty space hit (fail)</td>
<td>$O(1), O(n)$; specific estimates in slides</td>
</tr>
<tr>
<td>Insert</td>
<td>Probe until found – replace value, or until empty - place at that index</td>
<td>$O(1), O(n)$; specific estimates in slides</td>
</tr>
<tr>
<td>Delete</td>
<td>Use lazy deletion</td>
<td>Same as find</td>
</tr>
</tbody>
</table>

Insert: 38, 19, 8, 109, 10
Using Linear Probing