Dictionary
Possible data structures

- Linked list (windowed)
- Block array

- Binary search trees
  - AVL tree
  - Red Black tree
  - B-tree
  - Splay tree

- Faster alternatives
Binary search tree (BST)

- for every internal node $u$, all nodes in $u$’s left subtree precede $u$ in the ordering and all nodes in $u$’s right subtree succeed $u$ in the ordering.
  - In-order traversal will give a sorted list
BST: Searching

If information is stored in a binary search tree, a simple recursive “divide and conquer” algorithm can be used to find elements:

```java
if (t.isEmpty()) terminate unsuccessfully;
else {
    r becomes the element on the root node of t;
    if (e equals r) terminate successfully;
    else if (e < r) repeat search on left subtree;
    else repeat search on right subtree;
}
```
• Insert is always at a leaf
• perform a search for the element as above
• if the element is found, take no further action
• if an empty node is reached, insert a new node containing the element
**BST: Delete**

- delete is straightforward if the element is found on a node with at least one external child
  - just use the standard Bintree delete operation
    - Leaf is easy
    - One child is easy too (replace it with the child)
BST: Delete (cont.)

- Two children
  - i) replace the deleted element with its predecessor — note that the predecessor will always have an empty right child
    - One left; then far right
  - ii) delete the predecessor

You can use the successor concept as well.
Just opposite
Balancing Trees

• Note that the delete procedure described here has a tendency over time to skew the tree to the right — as we have seen this will make it less efficient.

• Alternative: alternate between replacing with predecessor and successor.

• In general, it is beneficial to try to keep the tree as “balanced” or “complete” as possible to maintain search efficiency.

• There are a number of data structures that are designed to keep trees balanced — B-trees, AVL-trees and Red-black trees.
AVL Trees

• Self-balancing trees

• An AVL tree is a binary search tree where
  • for every node, the height of the left and right subtrees differ by at most one.
  • This means the depth of any external node is no more than twice the depth of any other internal node.

Nodes are marked with the height of the right sub-tree minus the height of the left sub-tree.
AVL Tree: Height

- What is the height of an AVL tree of $N$ nodes
- Converse problem: what is the minimum number of nodes are in an AVL tree of $h$
- *(say the right subtree has height 1 more than left subtree for every node)*

$$N_h = 1 + N_{h-1} + N_{h-2}$$

So:

$$N_h > F_h; \text{ golden ratio: } F_h = \frac{\varphi^h}{\sqrt{5}}$$

$$\frac{\varphi^h}{\sqrt{5}} < N$$

$$h \approx \log_\varphi N = 1.44 \log N$$

$$\varphi = \frac{1 + \sqrt{5}}{2} = 1.6180339887...$$
AVL Tree: Height

- \( N_h = 1 + N_{h-1} + N_{h-2} \)
- \( N_h > 1 + 2N_{h-2} \)
- \( N_h > 2N_{h-2} = O(2^{h/2}) \)
- \( h=2 \log N \)
AVL Tree Operations

• Since an AVL tree is a binary search tree, the searching algorithm is exactly the same as for a binary tree.

• However, the insertion and deletion operations must be modified to maintain the balance of the tree.
# AVL Tree: Insert

<table>
<thead>
<tr>
<th>Node</th>
<th>Height of left subtree – height of right subtree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>-1</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
</tr>
</tbody>
</table>

```
 10
   / \
   5   13
  / \   /
 1  6  17
  ```
AVL Tree: Insert

<table>
<thead>
<tr>
<th>Node</th>
<th>Height of left subtree – height of right subtree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>-2</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
</tbody>
</table>
AVL: Insert

- 4 possible situations:
  - Insert into the left subtree of the left child (LL Rotation)
  - Insert into the right subtree of the right child (RR Rotation)
  - Insert into the left subtree of the right child (RL Rotation)
  - Insert into the right subtree of the left child (LR Rotation)
Rotation
LL Rotation
RR Rotation
LR Rotation
RL Rotation
AVL Insert

• If the unbalanced node’s height is:
  • Positive(>=2: left side)
    • If the left child’s height s
      • Positive : LL rotation
      • Negative: LR rotation
  • Negative (<=-2: right side)
    • If the right node’s height is:
      • Positive: RL rotation
      • Negative: RR rotation
AVL Insert: Need to check all
AVL Insert: Summary

• To insert an element, we:
  • 1. Insert the element into an external node (as per usual for a binary search tree).
  • 2. Starting with its parent, check each ancestor of the new node to ensure the left and right subtree heights differ by less than two.
  • 3. If we find a node such that one child has height $k - 2$ and the other has height $k$, then we perform a rotation to restore balance.
AVL Tree: Delete

• Note that both rotations do not increase the height of the sub-tree, so insertion only needs to be done at the lowest unbalanced ancestor.

• To delete an element, we:
  • Delete the element (as per usual for a binary search tree).
  • Starting with its parent, check each ancestor of the new node to make sure it’s balanced.
  • If any node is not balanced, perform the necessary rotation (as above).
  • Continue to check the ancestors of the deleted node up to the root.
AVL Tree: Complexity

• Rotations are constant time operations.
• Insertions and deletions involve searching the tree for the element \( O(h) \), where \( h \) is the height of the tree) and then checking every ancestor of that element \( O(h) \) in the worst case).
• Complexity follows from the claim: the height of an AVL tree is less than \( 2 \log n \) where \( n \) is the number of elements in the tree.
B-Tree

• A B-tree is a tree data structure that allows searching, insertion, and deletion in amortized logarithmic time.
• Each node of a B-tree can contain multiple items and multiple children (these numbers can be varied to tweak performance).

We will consider 2-3 B-trees, where each node can contain up to two items and up to three children.
B-Tree

• If there is just one item in the node, then the B-Tree is organised as a binary search tree: all items in the left sub-tree must be less than the item in the node, and all items in the right sub-tree must be greater.

• If there are two elements in the node, then:
  • all items in the left sub-tree must be less than the smallest item in the node
  • all items in the middle sub-tree must be between the two items in the node
  • all elements in the right sub-tree must be greater than the largest item in the node

• Also, every non-leaf node must have at least two successors and all leaf nodes must be at the same level.
Red-Black Tree

• A red-black tree is another variation of a binary search tree that is slightly more efficient (and complicated) than B-trees and AVL trees.

• A red-black tree is a binary tree where each node is coloured either red or black such that the colouring satisfies:
  • the root property — the root is black
  • the external property — every external node is black
  • the internal property — the children of a red node are black
  • the depth property — every external node has the same number of black ancestors.
Red-Black tree
Red-Black tree: Height

- A subtree rooted at node $v$ has at least $2^{bh(v)} - 1$ internal nodes
  - $bh(v) =$ the number of black nodes (not counting $v$ if it is black) from $v$ to any leaf in the subtree (called the black-height).
Red-Black tree: Insert

• Use the BST insert algorithm to add K to the tree
• colour the node containing K red
• restore red-black tree properties (if necessary)