Maps So Far

• hash tables for unsorted data
  • expected $O(1)$ for many operations but...
    • have to implement a good hash function
    • have to handle collisions
    • bad if we need sorted iteration

• sorted table map for sorted data
  • $O(\log n)$ searches and sorted but...
    • linear insertions/deletions

Want a sorted map with sub-linear insertions and deletions
Binary Search Trees

• Make the binary search part of the structure itself!
• Recursive definition
  • all values in left subtree are less than current node
  • all values in right subtree are greater than current node
• in-order traversal gives sorted order
  • (example on board)
• searching depends upon height of tree
  • best case: $O(\log n)$
  • worst case: $O(n)$
Navigating

- Want to avoid full traversals
  - `first()` – position of least key
  - `last()` – position of greatest key
  - `before()` – greatest key less than current
  - `after()` – least key greater than current
- (Examples in notes/textbook)
- Searches start at root, $O(h)$
Insertions/Deletions

• Insertion
  • search until failure, then insert at failure site

• Deletion
  • if leaf
    • just delete
  • else
    • if single subtree
      • delete
      • move subtree up
    • if two subtrees
      • delete
      • move before-hand node up
      • move before-hand subtrees as if it was deleted
Implementation

• `binary_search_tree.py`
  • public vs non-public API
• Most operations $O(h)$, but recall
  $\log(n+1) - 1 \leq h \leq n - 1$
  Murphy’s Law: $h = n - 1$
• How to avoid? Make the tree self-balancing!
  • We’ll look at three different approaches
Balancing

- *rotation*
  - moves a child above a parent
  - $O(1)$ if linked-node tree
- *trinode restructuring*
  - double rotation
- rely on correct double-labeling of nodes
  - $x, y, z$ for generations
  - $a, b, c, T_1, T_2, T_3, T_4$ for in-order order
- (Examples on board)