Today
- Binary search trees (wrap up)

Assignments
- HW9 due Thur
- Exam 2 out (due next Tues before class)

Recall the definitions of “full”, “complete”, and “balanced”

- **Full:** (i) internal nodes have 2 children, (ii) ea. node’s subtrees same height
- **Complete:** full at level \( n - 1 \), level \( n \) filled from left to right
- **Balanced:** ea. node’s left and right subtree heights differ by at most 1
Implementing remove for a BST

Some examples:

![BST Diagram]

What should/could the tree look like after removing:

- Node a?
- Node e?
- Node b?
- Node d?

Removing nodes from a BST can be tricky! ... cases we need to consider:

- The tree is **empty**
- Item to be removed is **not in the tree**
- Node containing item to remove is a **leaf**
- Node containing item to remove has **one child**
- Node containing item to remove has **two children**
Removing a leaf node

- Only have to delete the node, set parent child pointer to `nullptr`

![Diagram of removing a leaf node]

Removing a node with one child

- The parent of the node to be deleted “adopts” the child

![Diagram of removing a node with one child]

Removing a node with two children

- This case is considerably more complicated
- Both children cannot be “adopted”

![Diagram of removing a node with two children]
One Solution: Replace and delete replacement

“inorder” successor = next node in sort order (e.g., c is after b)

“inorder” predecessor = previous node in sort order (e.g., a is before b)

• we’ll replace contents of node to delete with inorder successor’s contents
• then we’ll delete the inorder successor

How do we find the inorder successor?

• go to right child
• then go left as far as possible

Note: The inorder successor will not have a left child

• which means it will have at most one (right) child
• which is an easier case for remove
Implementing remove

- Since we are just following a single path (even down to inorder successor)
- We can just use iteration (looping)

Instead, we'll use a mix of recursion and iteration:

```c
// remove helper
Node* remove(const K& key, Node* subtree_root);
```

The basic idea ...

```c
1 if (subtree_root && key < subtree_root->key)
2    subtree_root->left = remove(key, subtree_root->left);
3 else if (subtree_root && key > subtree_root->key)
4    subtree_root->right = remove(key, subtree_root->right);
5 else if (subtree_root && key == subtree_root->key) {
6    // case 1: subtree_root->left is empty
7    ...
8    // case 2: subtree_root->right is empty
9    ...
10   // case 3: subtree_root has two children
11   // use iteration to find, replace, delete inorder successor
12   ...
13 } return subtree_root;
```
Treesort

Given a list of items to sort, the treesort sorting algorithm ...

- adds each element one-by-one to a binary search tree
- then performs inorder traversal to produce a sorted list

Analysis

Is this a “good” (i.e., efficient) way to sort?

What are the worst cases?
- With unbalanced search trees, list is ordered (either ascending or descending)

What are the best cases?
- List results in a balanced tree after insertions
- Note: balance implies all root-to-leaf paths minimal or “close enough”

What is the time complexity?
- Once tree is created, building output list is $O(n)$ (traverse all nodes once)
- Worst case: $O(n^2)$
  - same as inserting $n$ elements at end of a linked list (without tail pointer)
  - on insertion $i$ have to traverse $i - 1$ nodes: $\sum_{i=1}^{n}(i - 1) = O(n^2)$
- Best case: $O(n \log n)$
  - insertion into a balanced tree is $O(\log n)$ ... since height is $O(\log n)$

So it’s only “efficient” ($O(n \log n)$) if we can guarantee tree is balanced!
More on Full, Complete, and Balanced Trees

Full and complete trees are of **minimum height**

\[ 2^h - 1 \text{ nodes in a full binary tree} \quad \text{... for height } h \]

- if the height is 1, we have \( 2^1 - 1 = 1 \)
- if the height is 2, we have \( 2^2 - 1 = 3 \)
- if the height is 3, we have \( 2^3 - 1 = 7 \)

Note that there are \( 2^{(h-1)} \) nodes at the leaf level

Q: How many nodes in a complete binary tree? \( \text{... } 2^{(h-1)} \leq n \leq 2^h - 1 \)
- if there is only one leaf-level node then \( 2^{(h-1)} - 1 + 1 = 2^{(h-1)} \)
- up to a full binary tree

Q: What is the minimum height of a binary tree

- If the tree is full, since \( n = 2^h - 1 \), we have:
  - \( n + 1 = 2^h \)
  - so: \( \log_2(n + 1) = h \)
- If the tree is complete and \( n = 2^{(h-1)} \) (one leaf-level node)
  - \( \log_2 n = h - 1 \)
  - so: \( (\log_2 n) + 1 = h \)
- Therefore, in a full and complete tree, the height of the tree is \( O(\log n) \)
- Which means all root-to-leaf paths are \( O(\log n) \)