Today

- Binary search trees (cont)

Assignments

- HW8 due
- HW9 out ...
- Quiz 9 out (after class, due by Tues)
- Exam 2 next week
Searching based on key in a BST (recursive)

- Is the key of the root node equal to the search key?
  - if yes, found a match!
- Is the search key less than the key of the root node?
  - if yes, search left subtree of the root
- Is the search key greater than the key of the root node?
  - if yes, search the right subtree of the root

Inserting into a BST

Basic Idea:

- Find where the node should be (key search)
- And then insert into that location
- Will always insert into a leaf node!

*Note that the order of insertions will determine the “shape” of the tree!*
Exercise:

1. Create a tree via the inserts: $d b g a c e f g h$
   - is this tree full? complete? balanced?

2. Create a tree via the inserts: $a b c d e f g h$
   - Q: what is wrong with this "tree"? ... it is a linked list!

3. Create a tree via the inserts: $g f e d c b a$
   - Q: what is wrong with this "tree"? ... it is a linked list!
Pointer based binary search trees:

Each Node consists of a left and right “next” pointer

```c
struct Node {
    K key;
    V value;
    Node* left;         // left child pointer
    Node* right;        // right child pointer
};

Node* root;          // pointer to the root node
```
Traversing binary trees

To navigate a specific path usually use a loop
- Like navigating a linked list (but many “forks” in the road)
- For example, for insert, find, remove

Sometimes we want to “visit” every node (or more than a path)
- For example, to get keys in order (our “sort” function) or find range of keys
- As we visit nodes, apply operations (like add to list of keys or print)
- We usually move top-to-bottom ("depth first") and left-to-right

To “visit” every node usually use recursion
- Especially since we don’t know path lengths
- Possible with loops, but much easier with recursion
Given a BST $T$, for traversal we have two cases:

(1). $T$ is empty, so nothing to do

(2). $T$ is not empty, so (in some specific order):

- Visit the root of $T$ (e.g., add root’s key to list or print key)
- Traverse $T$’s left subtree $T_l$
- Traverse $T$’s right subtree $T_r$

We’ll look at three different styles of common traversals

- preorder = visits in order of “arrival” ... prefix (or “polish”) notation ($+ 3 4$)
- inorder = visits in “sort” order ... infix notation ($3 + 4$)
- postorder = visits “bottom up” ... postfix (“reverse polish”) notation ($3 4 +$)
In a “preorder” traversal:

1. visit subtree root
2. traverse left subtree
3. traverse right subtree

Q: What order are the nodes visited?

\[d \; b \; a \; c \; g \; e \; f \; h\]

The basic preorder traversal algorithm

```cpp
void preorder(const Node* subtree_root) {
    // check if done
    if (subtree_root == nullptr) {
        return;
    }
    // ... visit the node ...
    // go left
    preorder(subtree_root->left);
    // go right
    preorder(subtree_root->right);
}
```
In an “inorder” traversal:
1. traverse left subtree
2. visit subtree root
3. traverse right subtree

Q: What order are the nodes visited?

\[ d \quad b \quad a \quad c \quad g \quad e \quad h \quad f \]

The basic preorder traversal algorithm

```c
void inorder(const Node* subtree_root) {
    // check if done
    if (subtree_root == nullptr) return;
    // go left
    inorder(subtree_root->left);
    // ... visit the node ...
    // go right
    inorder(subtree_root->right);
}
```
In a “**postorder**” traversal:
1. traverse left subtree
2. traverse right subtree
3. visit subtree root

Q: What order are the nodes visited?

\[a \ c \ b \ f \ e \ h \ g \ d\]

The basic preorder traversal algorithm

```c
void postorder(const Node* subtree_root)
{
    // check if done
    if (subtree_root == nullptr)
        return;
    // go left
    postorder(subtree_root->left);
    // go right
    postorder(subtree_root->right);
    // ... visit the node ...
}
```
How can we modify these to implement our “range” search?

- i.e., finding all keys within a range of keys $k_1$ to $k_2$

Basic idea:

- do an inorder traversal (traverse left, add, traverse right)
- at each tree node:
  1. if $k \leq k_1$, search $k$’s right subtree (out of range)
  2. if $k_1 \leq k \leq k_2$, search $k$’s left and right subtree (in range)
  3. if $k > k_2$, search $k$’s left subtree (out of range)

Q: What order are nodes traversed for range $c$ to $e$?

- enter $d$, enter $b$, enter $c$, add $c$, add $d$, enter $g$, enter $e$, enter $f$, add $e$
Implementing remove for a BST

Some examples:

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](image)

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](image)

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](image)
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:

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

The basic idea ...

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 is a leaf
7   ...
8   // case 2: subtree_root has one child
9   ...
10  // case 3: subtree_root has two children
11   // use iteration to find, replace, delete inorder successor
12   ...
13 } return subtree_root;
```