**Today**

- Binary Search Trees (cont)

**Assignments**

- HW 7 out
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
```
A “subtree” is a tree rooted at a descendent of the root node

- a subtree includes all descendents of the subtree “root” $n$
- we often say a subtree is rooted at $n$

An “empty” tree has no nodes
A “path” is a sequence of nodes $n_i n_j \ldots n_k$

- the next node after $n$ in the sequence is a child of $n$
- a root–to–leaf path starts at the root and traverses children ending at a leaf
- there can be many paths within a tree

The “length” of a path is the number of path nodes

- the length of path $d \rightarrow c$ is 3, $f \rightarrow g$ is 2, and $a \rightarrow a$ is 1
- note: length often defined as number of edges (but less convenient for us)

The “height” of a tree is the length of the longest root-to-leaf path

- the height of the above tree is 3
- the height of the subtree rooted at $b$ is 2
- the height of the subtree rooted at $g$ is 1
- the height of an empty tree is 0
- note: subtract 1 in other definition of length (e.g., 1-node tree has height 0)
Binary Trees

In a binary tree, each node has \textbf{at most} 2 children

Thus, $T$ is a binary tree if

$\bullet$ $T$ has no nodes (is empty); or

$\bullet$ $T$ has the form:

$$
\begin{array}{c}
  r \\
  \downarrow \\
  T_l & & T_r
\end{array}
$$

$\bullet$ where $r$ is the root node of $T$ and $T_l$ and $T_r$ are binary trees

$\bullet$ $T_l$ is the left subtree of $r$ and $T_r$ is the right subtree of $r$

Note that this definition is \textit{recursive}!

$\bullet$ this will be helpful as we compute over trees

We can recursively define the notion of a binary tree’s height ...

$\bullet$ the height of a binary tree $T$ is:

$- 0$ if $T$ is empty

$- 1 + \max$ of the height of the left and right subtrees of $T$'s root
Binary Search Trees

A binary search tree (BST) adds additional constraints to a binary tree

- specifically, nodes are stored in sort order (by key)

The order constraint: For each node $n$ in a BST:

- if node $n_l$ is in the left subtree of $n$ then $\text{key}(n_l) \leq \text{key}(n)$
- if node $n_r$ is in the right subtree of $n$ then $\text{key}(n_r) > \text{key}(n)$

Note: Can think of a BST as a way to combine linked lists with binary search
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

For HW-7: Implement contains() using iteration ...

```cpp
Node* ptr = root;
while (ptr) {
    // ... check and update ptr appropriately ...
}
// if we reach this point, we didn't find the key
```

For HW-7: Implement height() using recursion ...

```cpp
// height helper, where st_root stands for "subtree root"
int height(const Node* st_root) const;
```

Recursion vs. Iteration in BSTs

- traversing a single path only needs iteration (generally faster)
- traversing multiple paths better suited for recursion
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!*

Q: What happens if keys inserted in order?
Q: What happens if keys inserted in reverse order?
Q: What order creates a more “balanced” tree?

For HW-7: Use iteration to implement `insert()`
Traversing binary trees

When navigating a BST, we recursively “visit” every node

- 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

Given a BST $T$, for traversal we have two cases:

1. $T$ is empty, so nothing to do ... base case

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

```c
void preorder(const Node* st_root)
{
    // check if done
    if (!st_root)
        return;
    // ... visit the node ...
    // go left
    preorder(st_root->left);
    // go right
    preorder(st_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?

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

The basic preorder traversal algorithm

```c
void inorder(const Node* st_root) {
    // check if done
    if (!st_root)
        return;
    // go left
    inorder(st_root->left);
    // ... visit the node ...
    // go right
    inorder(st_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

```cpp
void postorder(const Node* st_root) {
    // check if done
    if (!st_root)
        return;
    // go left
    postorder(st_root->left);
    // go right
    postorder(st_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 $k_1 < k < k_2$, search $k$’s left and right subtree (in range)
  3. if $k_2 \leq k$, 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 $e$, add $e$
HW-7: Recursive helpers

(1) To compute the tree height:

```cpp
int height(const Node* st_root) const;
```

(2) To delete tree nodes and reset count:

```cpp
void make_empty(Node* st_root);
```

(3) To copy RHS tree nodes (for use in copy assignment):

```cpp
Node* copy(const Node* rhs_st_root) const;
```

For Example: `root = copy(rhs.root);`

(4) To get a sequence of sorted keys (inorder traversal):

```cpp
void sorted_keys(const Node* st_root, ArraySeq<K>& keys) const;
```

(5) To find matching keys (see previous)

```cpp
void find_keys(const K& k1, const K& k2, const Node* st_root,
               ArraySeq<K>& keys) const;
```

(6) And to erase (which we'll talk about more next time)

```cpp
Node* erase(const K& key, Node* st_root);
```