CSCI 262
Data Structures

Hashtables
Binary Trees

Review: Sets and Maps

- Data structures for holding unique keys
- Sets just hold keys
- Maps associate keys with values
- Principal operations:
  - find() - lookup key/value in set/map
  - insert() - put a new key/value into set/map
  - erase() - remove a key/value from set/map

O(1) Table Lookups

- Suppose keys are known to be in range 0-99:
  - What is easiest way to store keys?
  - What is the “big-O” complexity of find()?
- Arguably, all keys in a computer are numbers!
  - However, range may be very large (too large!)
  - Also, have to ensure uniqueness of number conversion for different keys

HASHTABLES

Basic Hashtable Idea

- Create an array of initially fixed size
- Convert key to an integer (called a hash code)
- Take hash code, mod array size
- Store key at resulting index

It’s that easy, except for collisions!

Very Simple Illustration

- Suppose keys are non-negative integers
- Suppose table size is 5
- Use key as hash code
Collision Resolution

Collisions:
- Table size typically << size of universe of keys
- Many keys will hash to same index!
- Collisions are inevitable (see Birthday paradox)

Different schemes for dealing with collisions:
- Chaining
- Open addressing (not covered today)

Chaining

- Basic idea: store linked list at each index
- When finding:
  - If null pointer at index, return NOT FOUND
  - Else, search every node in linked list for item
- When inserting:
  - First do a find() – if item is in linked list, do nothing
  - If not present in list, insert new item at head of list
- When erasing:
  - Find item
  - If found, remove from linked list

Updated Illustration

- Suppose keys are non-negative integers
- Suppose table size is 5
- Use key as hash code

Analysis of Hashing with Chaining

- Best Case:
  - Every entry occupies a unique location
  - Linked lists are all empty or have a single node
  - All operations thus O(1)
- Worst case?
  - N entries occupying same location
  - find() O(N)
  - Also insert/delete O(N) since find() is first step
    - Inserts really average 1 + ... + N = O(N²) over N inserts \(\rightarrow\) O(N)
    - per insert – gets more complicated with deletions

Analysis, con’t.

- Worst case not so great
- However, we will likely use hashtable many times:
  - Q: what is expected (average) cost of find()? 
  - Probabilistic analysis sketch:
    - Assume every hash code equally probable
    - Expected occupancy in any slot is \(\alpha = N / \text{table size}\)
    - Expected cost of find() is \(1 + \alpha/2 = O(1)\)
    - Typically choose table size so \(\alpha \leq 0.75\) or so.

Analysis, con’t.

If “uniform hashing” assumption holds:
- find() is O(1) expected
- insert() is O(1) plus O(1) for insert at head = O(1)
- erase() is O(1) plus O(1) for erase from linked list = O(1)

All operations are expected O(1)!
(Could get unlucky, of course...)
Hash Functions

- First defense against collisions is a good hash function
- For example: hashing strings
  - Could just take first four bytes, cast to int
    - Easy and fast to compute
    - Can’t distinguish “football”, “footrace”, “foot”, ...
  - Could just add up ascii codes
    - Almost as easy and fast to compute
    - Can’t distinguish “saw” from “was”, though

Designing a Good Hash Function

- A good hash function:
  - Fast to compute
  - Uses entire object
  - Separates similar objects widely
  - “Random-like”

Hashing Strings

- Java’s String hash function:
  \[ h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i} \]

Hashing Integers

First convert key to some number \( k \), then:

- Division method:
  - \( h(k) = k \mod \text{table size} \)
  - Avoid e.g., table size = 2^p \rightarrow else h(k) just low order bits of k!
  - Good choice: prime not too close to exact power of 2
  - Note this method dictates size of hashtable

- Multiplication method:
  - Multiply \( k \) by real constant \( A: 0 < A < 1 \)
  - Extract fractional part of \( kA \)
  - \( h(k) = \lfloor (\text{table size})(kA \mod 1) \rfloor \)
  - Advantage: size of table doesn’t matter!
  - Good choices for \( A \): transcendental numbers, \( \sqrt{2} \), etc.

Multiplication Method Illustration

- Suppose keys are non-negative integers
- Suppose table size is 5
- Use \( A = \sqrt{5} - 1 \)
- Insert 1,2,3,4,5

  E.g., insert 3:
  \[ h(3) = \lfloor (5)(3\cdot0.85410 \mod 1) \rfloor = 4 \]

BINARY TREES
Binary Trees

A binary tree is defined recursively:

- A tree is empty or
- A root node with a left child and a right child, each of which is a binary tree.

### Tree Terminology

- **Internal nodes** (nodes with children)
- **Root node**
- **Edge**
- **Node**
- **Internal nodes (nodes with children)**
- **External nodes / leaves**

**Implementing the Binary Tree**

Just follow the recursive definition to get a simple implementation:

```cpp
template <class T>
class binary_tree_node {
    public:
        T data;
        binary_tree_node<T>* left;
        binary_tree_node<T>* right;
}
```

### Binary Tree Traversals

- A traversal of a tree is the act of visiting every node in the tree once.
- There are three traversal orders:
  - In-order
  - Pre-order
  - Post-order

### In-Order Traversal

Visit the left sub-tree, the root, and then the right sub-tree:

```
The numbers give the order of the visited nodes.
```
**Pre-Order, Post-Order**

Pre-order: visit the root first, then the left and right sub-trees

Post-order: visit the left sub-tree, then the right sub-tree, and finally the root

---

**In-Order Printing**

Note naturally recursive description: visit the left sub-tree, then, the root, and finally the right sub-tree.

So we get a naturally recursive implementation:

```cpp
template <class T>
void print_inorder(binary_tree_node<T>* root) {
  if (root != NULL) {
    print_inorder(root->left);
    cout << root->data << " ";
    print_inorder(root->right);
  }
}
```

---

**Other Implementations**

Can you write the pre-order and post-order traversal code?

---

**Binary Search Trees**

Data structure for holding comparable elements
- Efficient searching, insertion, deletion
- Underlying structure for (ordered) sets, maps

Items in left subtree < item at root; root item < items in right subtree; subtrees are Binary Search Trees.

---

**In Order Traversal**

Visit left sub-tree, visit root, visit right sub-tree.

---

**Searching**

Example: search (root, "cherry")

```cpp
template <class T>
binary_tree_node<T>* search(binary_tree_node<T>* root, T val) {
  if (root == NULL) return NULL;
  if (val == root->data) return root;
  if (val < root->data) return search(root->left, val);
  return search(root->right, val);
}
```
Inserting, Removing

- **Insert:**
  - Always at a leaf
  - Insert new element where you expect to find it (e.g., search first)

- **Removing:**
  - Easy if element is leaf or has one child
  - Otherwise, a bit trickier
  - No time to cover these in this lecture 😊

Complexity of Search

- How many recursive steps?
  - A. ≤ height of tree

Height of Trees

So how high is a tree with N nodes?

Best case: \( h = \lceil \log_2(N+1) \rceil \)

Worst case: \( h = N \)

Self Balancing BSTs

- Trees become unbalanced through series of inserts and deletes
- Self-balancing: perform \( O(\log N) \) or fewer operations to rebalance after insert, delete
- Height is kept at \( O(\log N) \)
- Examples of self-balancing BSTs:
  - Red-Black trees
  - AVL trees
  - Splay trees

Summary

- Unordered maps, sets
  - Built on hashtables
  - \( O(1) \) complexity for all operations (in expectation)
- Ordered maps, sets
  - Built on self-balancing binary search trees
  - \( O(\log n) \) complexity for all operations

WRAPPING UP