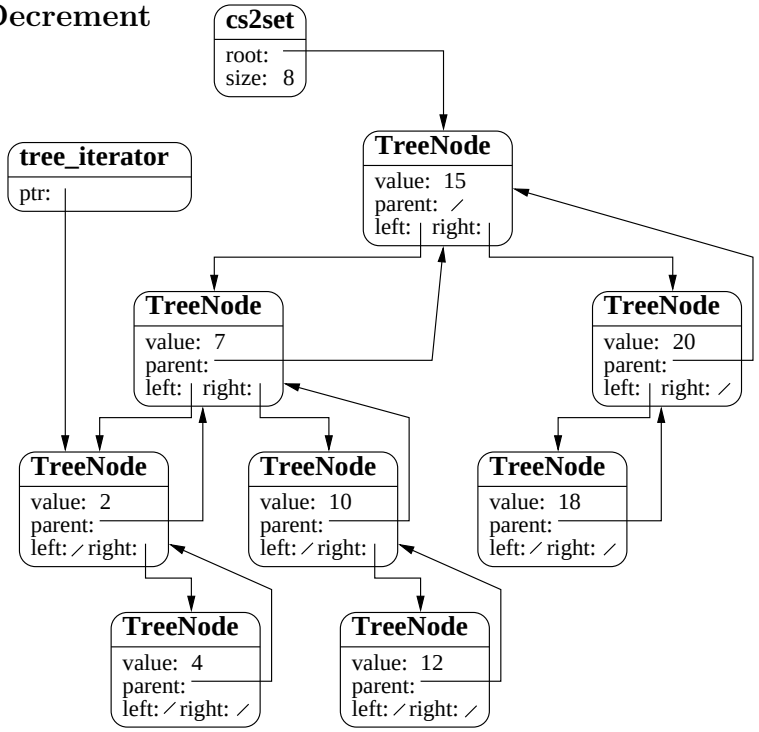
CSCI-1200 Computer Science II — Fall 2006
Lecture 22 — Hash Tables

Review from Lectures 20 & 21

- Binary Trees & Binary Search Trees
- cs2set class implemented using a Binary Search Tree
- BST operations: find, insert, destroy, erase (remove element), printing, begin & end (iterators), tree height

22.1 Tree Iterators – Increment & Decrement



- The increment operator should change the iterator's pointer to point to the next TreeNode in an in-order traversal — the "in-order successor" — while the decrement operator should change the iterator's pointer to point to the "in-order predecessor".
- Unlike the situation with lists and vectors, these predecessors and successors are not necessarily "nearby" (either in physical memory or by following a link) in the tree.
- There are two common solution approaches:
  - Each iterator maintains a stack of pointers representing the path down the tree to the current node.
  - Each node stores a parent pointer (see diagram). Only the root node has a null parent pointer.
- If we choose the parent pointer method, we'll need to rewrite the `insert` and `erase` member functions to correctly adjust parent pointers.
- Although iterator increment looks expensive in the worst case for a single application of `operator++`, it is fairly easy to show that iterating through a tree storing n nodes requires $O(n)$ operations overall.

22.2 Exercise

- Implement an algorithm for finding the in-order successor of a node.

```
TreeNode* FindSuccessor(TreeNode *current) {




}
```

22.3 Binary Search Tree Performance

- The efficiency of the main find, insert, & erase algorithms depends on the height of the tree.
- The best-case and average-case heights of a binary search tree storing n nodes are both $O(\log n)$. The worst-case, which often can happen in practice, is $O(n)$.

  Exercise: Draw an example tree and specify the arguments to find, insert, & erase that would exhibit this worst-case behavior.

- Algorithms that automatically re-balance a tree data structure in order to avoid the worst-case behavior will be covered in Data Structures and Algorithms. Examples include red-black trees or AVL trees.
- Standard Library (STL) maps & sets are implemented using a tree data structure. That's why all the find, insert, and erase operations run in $O(\log n)$ operations *and* iterating through the data structure produces the elements in sorted order.

22.4 Today's Lecture: Hash Tables

- Hash Tables, Hash Functions, and Collision Resolution
- Hash Table Performance
- Binary Search Trees vs. Hash Tables

22.5 Definition: What's a Hash Table?

- A table implementation with *constant time access*.
  - Like a map, we can store key-value pair associations in the hash table.
    But it's even faster to do find, insert, and erase with a hash table!
    However, hash tables *don't* store the data in sorted order.
- A hash table is implemented with a array at the top level.
- Each key is mapped to a slot in the array by a *hash function*.

22.6 Definition: What's a Hash Function?

- A simple function of one argument (the key) which returns an index (a bucket or slot in the array).
- Ideally the function will "uniformly" distribute the keys throughout the range of legal index values (0 → k-1).
- **What's a collision?**
  When the hash function maps multiple (different) keys to the same index.
- **How do we deal with collisions?**
  One way to resolve this is by storing a linked list of values at each slot in the array.

22.7 Example: Caller ID

- We are given a phonebook with 50,000 name/number pairings. Each number is a 10 digit number. We need to create a data structure to lookup the name matching a particular phone number. Ideally, name lookup should be O(1) time expected, and the caller ID system should use O(n) memory (n = 50,000).
- We'll review how we solved this problem in Lab 5 with an STL `vector` then an STL `map`. Finally, we'll implement the system with a hash table.
- Note: In the toy implementations that follow we use small datasets, but we should evaluate the system scaled up to handle the large dataset.

22.8 Caller ID with an STL Vector

```
void add(vector<string> &phonebook, int number, string name) {
  phonebook[number] = name;
}

void identify(const vector<string> &phonebook, int number) {
  if (phonebook[number] == "UNASSIGNED")
    cout << "unknown caller!" << endl;
  else
    cout << phonebook[number] << " is calling!" << endl;
}

int main() {
  // create the phonebook, initially all numbers are unassigned
  vector<string> phonebook(10000, "UNASSIGNED");

  // add several names to the phonebook
  add(phonebook, 1111, "fred");
  add(phonebook, 2222, "sally");
  add(phonebook, 3333, "george");

  // test the phonebook
  identify(phonebook, 2222);
  identify(phonebook, 4444);
}
```

Exercise: What's the memory usage for the vector-based Caller ID system? What's the expected running time for find, insert, and erase?

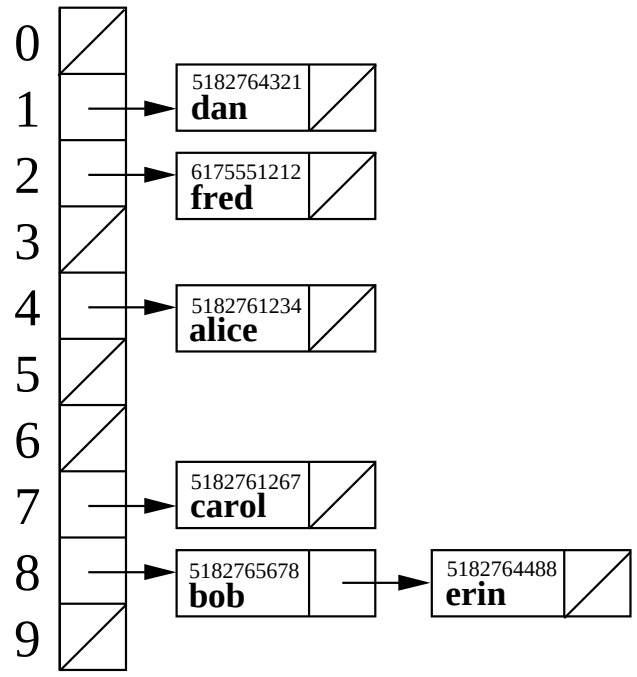
22.9 Caller ID with an STL Map

```
void add(map<int,string> &phonebook, int number, string name) {
  phonebook[number] = name;
}

void identify(const map<int,string> &phonebook, int number) {
  map<int,string>::const_iterator tmp = phonebook.find(number);
  if (tmp == phonebook.end())
    cout << "unknown caller!" << endl;
  else
    cout << tmp->second << " is calling!" << endl;
}

int main() {
  // create the phonebook, initially all numbers are unassigned
  map<int,string> phonebook;

  // add several names to the phonebook
  add(phonebook,1111,"fred");
  add(phonebook,2222,"sally");
  add(phonebook,3333,"george");

  // test the phonebook
  identify(phonebook,2222);
  identify(phonebook,4444);
}
```

Exercise: What's the memory usage for the map-based Caller ID system? What's the expected running time for find, insert, and erase?

22.10 Now let's implement Caller ID with a Hash Table

```
#define PHONEBOOK_SIZE 10

class Node {
public:
  int number;
  string name;
  Node* next;
};

// corresponds a phone number to a slot in the array
int hash_function(int number) {

}

// add a number, name pair to the phonebook
void add(Node* phonebook[PHONEBOOK_SIZE],
         int number, string name) {









}

// given a phone number, determine who is calling
void identify(Node* phonebook[PHONEBOOK_SIZE], int number) {









}

int main() {
  // create the phonebook, initially all numbers are unassigned
  Node* phonebook[PHONEBOOK_SIZE];
  for (int i = 0; i < PHONEBOOK_SIZE; i++)
    phonebook[i] = NULL;

  // add several names to the phonebook
  add(phonebook, 1111, "fred");
  add(phonebook, 2222, "sally");
  add(phonebook, 3333, "george");

  // test the phonebook
  identify(phonebook, 2222);
  identify(phonebook, 4444);
}
```



22.11 Exercise: Choosing a Hash Function

- What's a good hash function for this application?
- What's a bad hash function for this application?

22.12 Exercise: Hash Table Performance

- What's the memory usage for the hash-table-based Caller ID system?
- What's the expected running time for find, insert, and erase?