4. Graphs, trees, and hash tables

Graphs and trees, oh my

- We’re going to introduce a little math here … you will revisit these topics in great detail in CS240 and other courses
  - For now, our approach will be only lightweight

- Graph theory is a cornerstone of theoretical computer science!
  - And trees are a kind of graph

Graphs and trees, oh my

A graph

- A graph G consists of a set of vertices (aka nodes) V plus a set of edges E.
  - Let $V = \{v_1, v_2, ..., v_n\}$ and $E = \{e_1, e_2, ..., e_m\}$
  - Then each $e_i$ is actually a pair $(v_i, v_j)$

- A graph can be directed or undirected
  - In a directed graph, $(v_i, v_j)$ is not the same as $(v_j, v_i)$
  - In an undirected graph, they are the same
An example directed graph

\[ G = (V, E) \]
\[ V = \{v_1, v_2, v_3, v_4\} \]
\[ E = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_4, v_3)\} \]

• The actual physical layout of the diagram is unimportant as long as the edges are connecting the correct nodes

• Nodes and edges can have additional attributes
e.g., nodes are cities with name and population,
edges have weights indicating distance and cost of tolls

The travelling salesman problem (TSP)

• Given a list of cities (nodes) and the distances between each pair of cities (weighted edges) ...
  – ... what is the shortest possible route that visits each city exactly once and returns to the origin city?
  – In mathspeak: Find the minimum length Hamiltonian cycle (each node visited exactly once, but finishing again at the start node)

Distance in km
http://algo.inria.fr/broutin/gallery.html
Solving TSP

• **Good/bad news:**
  – You can, in principle, grind out the answer by brute force
    • Pick a starting city, then a second, third, etc.
  – What’s the complexity of this approach, assuming there are N cities?
    • This isn’t feasible solution for large N

• **Realistic good news:**
  – There are polynomial-time heuristics that can solve TSP to within a few percent of optimal with high probability on real-world problems

Solving TSP

• **Mathematical bad/good news:**
  – There is (very strongly believed to be) no exact solution to TSP that is polynomial time
    i.e., (we believe) there is no K such that there is a solution that is guaranteed to be no worse than $O(N^K)$ for an arbitrary graph of N nodes

  • TSP is an example an **NP-hard problem**
    – A formal description what this means will come in a later course 😊
    – Roughly, these are problems that are solvable "in principle", but not practically
      e.g., reliable encryption depends on the existence of problems such as these

Trees

• **A tree is a special kind of graph:**
  – If the tree is non-empty, there is a single special node called the **root**
  – Each node may have zero or more **children**
  – The root node has no **parent**; every other node has exactly one parent
    • The parent relationship is the only kind of edge in a tree
  – Observation: there can be no cycles in the underlying undirected graph

• **Facts about trees**
  – Typically, we direct edges from the parent to the child node
    • But sometimes we draw them in the opposite direction
  – A tree must be fully connected (no orphan nodes or clusters)
  – Observation: For any node, there is a unique path to it from the root
The male side of my family tree (going back only two generations).

The female side of my daughter/wife’s family tree (going back only two generations).

Trees

- The "top" node is the root; it's the only node with no parent
  - When we program, we typically use a special ptr to point to the root node, like the first element of a list
  - So, nullptr == root means the tree has no elements

- A node with no children is called a leaf

- Non-leaves are called internal nodes

Trees

- The height of a node is the length of the longest path from it down to a leaf
  - The height of the tree is the height of the root
  - "David" has height 2; the height of the full tree is 3
  - The height of a leaf is always zero
  - [Look down]

- The depth of a node is the length of the path from the root to it
  - The root has depth zero, its children have depth 1, etc.
  - [Look up]
**Binary trees**

- In a *binary tree*, each node has at most two children
  - We usually call them the *left* and *right* child
  - In C++, *left* and *right* would typically have type `Node`
- The *binary search tree* is one kind of binary tree
- The *heap* is also a kind of binary tree, but it is not a BST as the ordering property is different

**BSTs**

- A *Binary Search Tree* (BST) is a special kind of tree:
  - Each node has a "key" value (that can be compared by "<") plus possibly other info too
  - We assume no duplicate keys for now
  - Each node can have up to two children (*left* and *right*)
  - The *BST property* holds for every node in the tree:
    - The keys of all nodes in the left subtree (if any) are < my key
    - The keys of all nodes in the right subtree (if any) are > my key
  - The following is **NOT** the BST property:
    - At each node, *left->key < key < right->key*
    - This is logically weaker than the BST property; all BSTs satisfy this property but so do some binary trees that are not BSTs

• In a generalized n-ary tree:
  - A node can have any number of children
  - The "child-of" relationship can model just about any kind of (strict hierarchical) relationship
This is a reasonably, but not perfectly, balanced BST

For any given dataset, there are many possible BSTs!
- This is a less balanced BST with the same data

A sorted linearly linked list is (effectively) also a BST!

This is a Fox News Tree

It is considered a "degenerate" BST

BST operations

- We're going to examine lookup, insert, print, and delete on BSTs
  - We'll assume that the keys are strings
  - Also, we'll assume that there's some other interesting data called otherStuff that's a string, tho we won't look at it
    - Realistically, lookup should take a key and return otherStuff, but we'll just return a boolean
`#include <iostream>
#include <string>
#include <cassert>
using namespace std;

struct BST_Node {
  string key;
  // we'll ignore "otherStuff"
  string otherStuff;
  BST_Node* left;
  BST_Node* right;
};

typedef BST_Node* BST;

void BST_init (BST& root) {
  root = nullptr;
}

bool BST_isEmpty(const BST& root) {
  return nullptr == root;
}

We'll also define:
• BST_has/BST_lookup,
• BST_insert,
• BST_print, and (finally)
• BST_delete

BSTs all the way down

• The BST property holds at every node in the whole tree:
  – The keys of all nodes in the left subtree (if any) are < my key
  – The keys of all nodes in the right subtree (if any) are > my key

• This means that if you pick an arbitrary node, the subtree with that node as root is also a BST!
  – This fact is important, as it means we can use recursion to implement most of our routines
  – Also, when we change a BST, we need worry only that the changes preserve the BST property; unchanged subtrees should still be OK.

BST lookup

• In a sorted linked list, we start at the first element

• In a BST operation, we start at the root
  – But if we don't find what we're looking for there?
  – Where do we go next?
**has vs. lookup**

- In real applications, **lookup** is more useful than **has** in most cases
  - Usually, want to return rest of info associated with that key, not just check if it's there
  - To do this, we'd probably return an object but that's going to be done in CS247

```cpp
bool BST_has (const BST& root, string key) {
    if (BST_isEmpty (root)) { // nullptr == ROOT
        return false;
    } else if (key == root->key) {
        return true;
    } else if (key < root->key) {
        return BST_has (root->left, key);
    } else {
        return BST_has (root->right, key);
    }
}
```

```cpp
// An approximation of what lookup might look like
string BST_lookup (const BST& root, string key) {
    if (BST_isEmpty (root)) { // nullptr == ROOT
        return ""; // or "ERROR", or ... umm ...
    } else if (key == root->key) {
        return root->otherStuff;
    } else if (key < root->key) {
        return BST_lookup (root->left, key);
    } else {
        return BST_lookup (root->right, key);
    }
}
```
Insertion into a BST

- You remember lookup, where you stopped either when you found the key you were looking for, or found a `nullptr` where it would have been?
  - Insertion is similar, except that you insert a new node where the `nullptr` was, and make the parent point to the new node
    - This is only one of many correct insertion algorithms for BSTs; there are also clever approaches that try to keep the tree reasonably balanced by rejuggling a bit on insert/delete
  - To make discussion simpler, let’s assume no duplicate keys are ever added
    - However, this code still works fine 😊

```
// This is only one of many possible implementations of BST_insert
void BST_insert (BST& root, string key) {
  if (BST_isEmpty (root)) {
    root = new BST_Node;
    root->key = key;
    root->left = nullptr;
    root->right = nullptr;
  } else if (key < root->key) {
    BST_insert (root->left, key);
  } else {
    BST_insert (root->right, key);
  }
}
```

Generic BST operations

- We usually use recursion!

- First, check if tree is empty
  - It’s a basis case for the recursion (tho maybe not the only one)

- If not empty, consider the current root node
  - Might want to "do something" at current root node
  - May want to traverse left subtree
    - i.e., recursive call to BST_op (root->left, …)
  - May want to traverse right subtree
    - i.e., recursive call to BST_op (root->right, …)

- Let's insert "elephant" into the BST
  - We find out the spot where is would be, if it were actually in the tree …
    - As the right child of dingo
  - You remember we stopped when the current ptr is `nullptr` …
    - Well, if we pass the ptr by reference, then we can simple create a new node "in place"
      - i.e., get the parent to point to the new node
BST structure

- As mentioned before, for a given data set, there are many possible BSTs.
- We've given the simplest, least disruptive correct algorithm for BST\_insert, but there are other correct algorithms also:
  - Usually, they attempt to keep the tree reasonably balanced, which is a good thing.

Q: If you took a sorted list and inserted the keys in that order, what would the resulting BST look like?
A: See left.

- "Interesting", but why should I care?
  - Well, hang on and I'll show you.

BST print

- Start at root; we want to print in alphabetical order, thus:
  - ... everything in left subtree needs to be printed before the root
  - ... and everything in right subtree needs to be printed after the root
  - So print the root in between!

- Now note that this is true for EVERY node in the tree, since each node can be considered to be the root of a sub-BST!
  - ... which leads to the following rather elegant recursive algorithm.
Print and inorder traversal

- BST_print's implementation uses a well-known algorithm for iterating through the nodes of a binary tree (not nec. a BST)
  - It's called an *inorder traversal*
  - It's the only tree traversal we'll look at in CS138, but you will see the idea again in CS240.
  - [Others include pre-order and post-order traversals]

Print and inorder traversal

- While you can trace through the execution carefully using a stack model as we’ve done before, there’s a cool trick for tracing inorder tree traversals:
  - When you encounter *leaf* node (first + only time), perform the op
    - In this case, print the key value
  - The *second* time you encounter an *internal* node, perform the operation
Iterative vs. recursive

- Some problems can be solved both iteratively and recursively. Which is better?

- Recursive solutions are often more mathematically elegant, as is the case with the BST operations we are looking at
  - Often, when presented with a nice recursive solution, it looks "obvious" ... but it might have been pretty hard to design it from scratch, especially if you are not used to thinking recursively.
  - The (after-the-fact) simplicity of a recursive solution might make it easier to get right, as there is less "accidental complexity"

Iterative vs. recursive

- Iterative solutions have a kind of "comfortable" cognitive appeal to many
  - All of the work is done in front of you (probably using a loop, maybe a stack too), it doesn't seem as "magical"
  - But often you have to do more "work" to achieve your goals, and thus there may be more details that can go wrong

- Note that neither approach is intrinsically more efficient than the other
  - In the old days, iterative solutions were sometimes preferred, as the overhead in making lots of recursive procedure calls was a real performance issue
  - With modern compilers and hardware, this is not a significant problem

Iterative vs. recursive

- Recursive solutions work really well for many data structures
  - But apart from that and "parsing", and despite the emphasis that academics like to put on recursion (hey, many of them are basically mathematicians at heart), professional software developers probably don't use recursion as often as you might think

- Let's see an iterative solution to BST_print
  - This is for fun only, OK? Here, recursion rules!

```cpp
// adapted from wikipedia code ... but don't use it
void iterativeBSTPrint (const BST& root) {
    stack<BST> nodeStack; // BST is Node*
    Node *cur = root;
    while (true) {
        if (nullptr != cur) {
            nodeStack.push(cur);
            cur = cur->left;
        } else if (nodeStack.size() == 0) {
            return;
        } else {
            cur = nodeStack.peek();
            nodeStack.pop();
            cout << "    " << cur->key << "\n" << endl;
            cur = cur->right;
        }
    }
}
```
// Compare the previous iterative solution to
// this clean, simple, elegant recursive one.

void BST_print (const BST& root) {
    if (! BST_isEmpty (root)) {
        BST_print (root->left);
        cout << "    " << root->key << "\" << endl;
        BST_print (root->right);
    }
}

Iterative vs. recursive

- The recursive approach is a big winner here, and with recursive data structures like BSTs in general
  - Recursion can always be fudged iteratively using a stack ... which is what the run-time does for you automatically

- Sometimes iterative solutions are simpler and more efficient
  e.g., generating $n^{th}$ Fibonacci number, naïve recursive solution is exponential, iterative version is linear in $n$

Complexity of BST operations

- Assume there are $N$ nodes in the tree ...

- Print is ... what?
  - $O(N)$ since all nodes are visited and printer once.

- What about lookup / insert?
  - Let’s examine the best case, which is a perfectly balanced tree
Complexity of BST operations

- So for a fully packed and balanced tree with N nodes and of height k:
  - Half the nodes are on the bottom level
  - Half the nodes are above (but half of those are at level N-1)

- The worst case for lookup (# of comparisons) is:
  - k for a key that's present
  - k+1 for a key that's not present
    - And k is roughly $\log_2 N$ if the tree is balanced

What's the worst case? (i.e., think unbalanced)
- Ans: A degenerate tree that's effectively just a linearly linked list: $O(N)$

<table>
<thead>
<tr>
<th>N</th>
<th>N/2</th>
<th>ceil ($\log_2 N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>500</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>5,000</td>
<td>14</td>
</tr>
<tr>
<td>100,000</td>
<td>50,000</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>500,000</td>
<td>20</td>
</tr>
</tbody>
</table>

BST recap
- BST property
- lookup, insert, print
- Balance!
Deleting from a BST

- This requires a little thought
  - Must preserve the BST property after deletion
  - Can't just delete a node ... what about the children?
  - Might want to juggle the tree to rebalance it ... in CS240
- Say, here's a simple, non-invasive idea that's cheap to do:
  - Add a boolean "zombie" flag to each node
    - Set to false initially on insertion, set to true on "deletion"
  - Zombie nodes are not printed and are not compared to keys during insert / lookup except for navigation
  - Advantage: simple, doesn't work so initially
  - Disadvantage: Works OK in short term, or if only few deletes ... but performance degrades if there is a lot of zombies
  - Let's not do this.

Deleting from a BST

- Assume we found the node we want to delete, call it the target node, and have access to its parent's link to it
  - Again, to make arguments simpler, we assume all keys are unique

- The hard part is going to be setting the target's parent to point to something "reasonable", and making sure the new tree really is a BST

- Let's consider three cases, based on how many children the target has
Case 1: Target node has no children
Easy, just delete the target node, set parent ptr to nullptr

Case 2: Target node has only one child
Easy, set target’s parent to point to target’s child, then delete target node

Case 1: Target node has no children
Easy, just delete the target node, set parent ptr to nullptr

Case 2: Target node has only one child
Easy, set target’s parent to point to target’s child, then delete target node
Case 2: Target node has only one child

Easy, set target’s parent to point to target’s child, then delete target node

Why does this work?

Suppose WLOG the target node is the right child of its parent
– We know that the subtree rooted at the target’s right child is a BST; thus, the BST property holds at “tiger”. But does the BST property still hold when we make "llama"’s right child be “tiger”?
– We know that target’s key is > key of parent
– And all of target node’s descendants (left and right) have key > parent’s key (BST property)
– The only change we made to the BST is making the parent point to the target’s child, and we know it belongs down the right subtree of the parent
– So the simple “promotion” approach works fine.

Case 3: Target node has two children

Needs some thought

– Let’s assume we’d like to have minimal disruption on the structure of the tree …
– Can we find another node that would be substitutable in place for the target node?
  • [We’ll call it the replacement node]

– Yes, there are guaranteed to be exactly two possible replacement nodes
  • Biggest key down the left subtree, or
  • Smallest key down the right subtree
– So pick one, it doesn’t matter which
  • WLOG we’ll pick the largest key in the left subtree
Case 3: Target node has two children

– Can we swap the replacement node into the target's place?
– What if the replacement node has two children?

Ans: We are guaranteed that it won't!

• All left descendants of the target are < target key
• If largest key in left subtree has a right child, then there is something even bigger, contradiction.
• So we know that the replacement node will have zero or one child

So let's try this:

1. Connect replacement's parent to replacement's child (if it exists) to temporarily disconnect the replacement node from the subtree
2. Then insert the replacement node in place of the target node:
   • Set replacement node's left / right ptrs to values of target's left / right ptrs, respectively
   • Set target's parent to point to replacement node
3. Delete target node

Deleting a node with two children

• A variant approach (arguably, conceptually simpler):
  1. Find the replacement node (e.g., max value in left subtree)
  2. Copy the data values of the replacement node (but not the ptrs) into the target node
     • If this is a BIG object, this might be expensive, but we won't worry for now
  3. Call BST_delete (target->left, target->key)
     • Note that target->key is the key of replacement node, not the original target
     • We are guaranteed that this will hit one of the first two cases and the replacement node will be deleted
Complexity of delete

- With this implementation, you first find the target node, then dive down one of its subtrees to find the largest (smallest) element in the left (right) subtree

- The deepest you will go altogether (i.e., max # of comparisons) is
  - Best case:
  - Worst case:

BSTs and balancing

- We've shown just one approach (the simple, naïve one) to inserting and deleting into a BST
  - We've used a policy of minimal disruption to the existing structure to do inserts / deletes
  - This approach works correctly!
  - ... however, over time BSTs can become very unbalanced, causing performance to degrade
  - Recall that for any data set, there are quite a few possible BSTs, some of which are more balanced than others

- Possible approaches:
  - Ignore unbalanced trees
    - Probably a bad idea ...
  - Ignore unbalanced trees usually; periodically do a mass rebalancing
    - A bit like Garbage Collection and vector space reallocations, but probably not worth it
  - On each insertion, do a little work to make sure that the tree is "reasonably" well balanced (CS240)
    - Examples: red-black trees, AVL trees
(Review) Binary search trees

- A binary search tree (BST) is a binary tree where the following property holds for every node:
  - The keys of all nodes in the left subtree (if any) are < my key
  - The keys of all nodes in the right subtree (if any) are > my key

- A BST is a sorted data container
  - BST_print produces (lexicographically) sorted data
  - And the other operations traverse the data elements lexicographically

- Insertion, lookup, and deletion are all $O(h)$, where $h$ is the tree height
  - If the tree is reasonably well balanced, then $h \approx \log N$
  - This is the most efficient ADT for storing sorted data that we have seen so far!
  - Our insert/delete operations are correct, but do not try to self-balance
  - IRL we use self-balancing BSTs to store sorted collections of data
    e.g., red-black trees (CS240), which have trickier insert/delete functions

<table>
<thead>
<tr>
<th>N</th>
<th>N/2</th>
<th>ceil (log₂ N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>500</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>5,000</td>
<td>14</td>
</tr>
<tr>
<td>100,000</td>
<td>50,000</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>500,000</td>
<td>20</td>
</tr>
</tbody>
</table>
Aside: Reversing a linked list

• Here's a nice problem:
  – Write a function that takes a plain old linked list (POLL) and reverses the order of the elements

```c
struct Node {
    string val;
    Node* next;
};
typedef Node* List;
void reverse (List& first) {...}
```

[This is a great question for interviews, Google was known to use it.]

```c
// Note reference parameter!!
void reverse (List& first) {
    Node* prev = nullptr;
    while (nullptr != first) {
        // Remember first->next before resetting first
        Node* temp = first->next;
        first->next = prev;
        prev = first;
        first = temp;
    }
    // Now reset first at the end!
    first = prev;
}
```

// Let's check if this works for border cases N=0, N=1

Strategy

• We start at the beginning of the list, and we'll iterate using the first ptr
  – first points to the first element of the rest of the list that we haven't processed yet

• In turn, we make each list element point to the previous element instead of the next one

• We'll maintain a pointer prev to the most recently processed element
  – At any moment, starting at prev, we could travel backwards starting at the most recently processed element back to the original first element

• Initially: first points to original first element, prev is nullptr
  – At the end, we'll need to reset first to prev

Other approaches

Similar but slightly simpler problem: Suppose you were asked to just read in string values and print them in reverse order ...

1. You could just read the strings into a vector, using push_back to add elements (and let the vector class add storage as needed)
   – Then you could start at either end of the vector and iterate in the desired direction

2. You could use recursion with no explicit data structure, as you did in a previous assignment
   – Effectively you are using the system's run-time stack as an implicit data structure, which is cool but subtle.
Mid-term info W2017

- Structure of the mid-term
  - T/F, short answer, … maybe
  - What is the output from this program?
  - Write a function that does XXX
  - What is wrong with this program?
  - …

- Words of advice:
  - Don’t panic!
  - Have a glance thru the whole thing
  - Don’t get stuck on one question

Abstract data type (ADT) vs. data structure

- An abstract data type (ADT) should be understandable by looking only at its API
  i.e., function signatures + pre/post conditions of provided operations
  e.g., stack, queue, deque, priority queue, sequence, dictionary (sorted or not)

- A data structure (DS) is more concrete
  - It connotes some idea of underlying implementation / physical structure that may not be discernible from the API alone
  - Data structures can be used to implement ADTs, tho DSs may be somewhat abstract themselves
  e.g., vector, singly-/doubly-linked list, various trees, hash table

- The line between the two ideas is blurry!
  - C++ STL provides a vector but not a sequence, a stack (implemented as a deque, in turn implemented as a dynamic array of pointers), …
The sequence ADT

- A sequence is a container of elements, indexed by a set of contiguous non-negative integers
  - Roughly the same idea as vector
    - Tho, vectors usually have the expectation of being implemented using a dynamic array (fast random access to elements, but costly insertions).
  - Usually, the first index is zero

- Common operations:
  - insert (element, index)
  - append (element) (plus maybe append(sequence))
  - at (index)
  - remove (index)

Implementing a sequence

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>append</th>
<th>at</th>
<th>remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++-style vector</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linked list</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Using an array implies a fixed maximum size
- insert/remove is really two ops: at plus add/remove element
  - Array/vector is O(1) for at, but causes O(N) copies on insert/remove to shift the rest of the elements to the left/right
  - Linked list at is O(N), but adding/removing element requires only changing of a few pointers (which is constant time)
- ACT == Amortized constant time: O(1) usually, but sometimes O(N)

The sequence ADT

- **insert** inserts an element into the sequence at the specified position, moving all following elements "one slot to the right" conceptually, increasing the active size of the container by one

- **append** adds a single element onto the end of the sequence, increasing the size by one (like vector>::push_back)
  - We may also provide a second definition of append that merges one sequence onto the end of another; this is reasonable if we used a linked list

- **at** returns the element at the specified index

- **remove** removes an element from the sequence at the specified position, moving all following elements "one slot to the left", decreasing the size by one

The C++ vector (and list and deque)

- Here, we are distinguishing between the mathematical sequence ADT, and the handy library classes that C++ provides: vector, list, deque

- Roughly, the C++ vector implements the sequence ADT
  - This vector adds a lot more functionality too
  - The C++ standard requires that it be implemented as an array for fast accessing of elements, and allows amortized constant time on append (called push_back) as a reasonably trade-off to achieve this

- C++ libraries list and deque also implement the sequence ADT:
  - list uses a vanilla doubly-linked list
  - deque allows add/removes at the front and back of the list; its implementation is kinda complicated, as we'll see
The C++ vector
(and list and deque)

- vector, deque, list support similar functionality ... but
  they also have some differences
  e.g., vector and deque support an at() method, but list does not

- Also, vector, list, deque use iterators for insert/
  erase (aka remove) instead of integer indexes

- They differ in performance on some operations; pick the one
  that best suits your needs for your problem

- More later ...

The dictionary ADT

- A dictionary is a collection of ordered pairs of the form:
  (key, value)
  - The idea is a simple lookup table:
    - Adding (key, value) to dict means that dict[key] should return value
      afterwards
    - The value can be a string, a set of values, a pointer to an object, ...
    - The dictionary may or may not be sorted by the key values
    - Sometimes called a map, or associative array

- The dictionary ADT supports the following operations:
  - add (key, value)
  - overwrite (key, value)
  - lookup (key)
  - remove (key)
The dictionary ADT

- **remove** removes the ordered pair \((key, dict[key])\) from the dictionary
  - So \(dict[key]\) is undefined afterwards

- Dictionaries may support a **print** operation
  - However, dictionaries are not assumed to be sorted by default!
  - Thus, the print may not be in any special order, but it will print all of the elements exactly once

- A related idea: A dictionary may support an **iterator** that can be used by a client to walk through all of the elements and perform an operation on each, such as "print the value"
  - The order of visiting the element may not be clearly defined; you can't assume it's going to be sorted

Implementing a dictionary

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>overwrite</th>
<th>lookup</th>
<th>remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted vector*</td>
<td>(O(N))</td>
<td>(O(\log N))</td>
<td>(O(\log N))</td>
<td>(O(N))</td>
</tr>
<tr>
<td>Unsorted vector</td>
<td>(O(1))</td>
<td>(O(N))</td>
<td>(O(N))</td>
<td>(O(N))</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>(O(N))</td>
<td>(O(N))</td>
<td>(O(N))</td>
<td>(O(N))</td>
</tr>
<tr>
<td>Unsorted linked list</td>
<td>(O(1))</td>
<td>(O(N))</td>
<td>(O(N))</td>
<td>(O(N))</td>
</tr>
<tr>
<td>Vanilla BST**</td>
<td>(O(\log N))</td>
<td>(O(\log N))</td>
<td>(O(\log N))</td>
<td>(O(\log N))</td>
</tr>
</tbody>
</table>

- Let's assume that operations do not check pre-conditions
  - (You could add a **lookup** if you want to do the checking)

The C++ map

- C++ provides a **map** data type that implements the idea of a dictionary
  - **map** is an ordered (sorted) container, as are the related structures **multimap**, **set**, and **multiset**
  - i.e., if you iterate thru a map, you will get the output in sorted order

C++ provides a **map** data type that implements the idea of a dictionary

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>overwrite</th>
<th>lookup</th>
<th>remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted vector*</td>
<td>(O(N))</td>
<td>(O(\log N))</td>
<td>(O(\log N))</td>
<td>(O(N))</td>
</tr>
<tr>
<td>Unsorted vector</td>
<td>(O(1))</td>
<td>(O(N))</td>
<td>(O(N))</td>
<td>(O(N))</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>(O(N))</td>
<td>(O(N))</td>
<td>(O(N))</td>
<td>(O(N))</td>
</tr>
<tr>
<td>Unsorted linked list</td>
<td>(O(1))</td>
<td>(O(N))</td>
<td>(O(N))</td>
<td>(O(N))</td>
</tr>
<tr>
<td>Vanilla BST**</td>
<td>(O(\log N))</td>
<td>(O(\log N))</td>
<td>(O(\log N))</td>
<td>(O(\log N))</td>
</tr>
</tbody>
</table>

- Let's assume that operations do not check pre-conditions
  - (You could add a **lookup** if you want to do the checking)

* A sorted vector can use binary search for **lookup**, but **add/remove** requires shifting about half of the list over one element.
* Assume the BST is reasonably balanced
The C++ map

• Typical declaration:

```cpp
map<T1, T2> m;
```

– T1 is the **key field** type; it **must support** `operator<`, which must in turn be a **strict weak ordering**
  i.e., anti-reflexive, anti-symmetric, transitive
  • It’s common to use strings or numbers as key; can use **ptrs**
  • Can use a user-defined class but you must ensure there is a “reasonable” `operator<` defined or provide an ordering **functor**

– T2 is the **value field**; can be anything

```cpp
// This creates a "word bag" for a text file.
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main (int argc, char* argv[]) {
    // record the number of times each word appears.
    map<string, int> m;
    string token;
    while (cin >> token) { // This is where the magic happens
        m[token]++;
    }
    // After below line is executed, m["the"] is part of the
    // map even if it didn't appear in the input stream
    cout << ""the" occurred " << m["the"] << " times\n";
    // So let's erase it if so ...
    if (0 == m["the"]) {
        m.erase("the");
    }
    // map supports bi-directional iterators; so this will
    // print pairs in "alphabetical" order of key
    for (map<string, int>::const_iterator i=m.begin();
    i!=m.end(); i++) {
        // For maps, "first" gets you the key ...
        // ... and "second" gets you the value
        cout << (*i).first << " " << (*i).second << endl;
    }
}
```

The C++ map

• The C++ map maintains the elements as sorted, and is usually implemented using a **red-black tree**
  – A red-black tree is a kind of BST that does some self-balancing on insert/delete
  – They guarantee worst case of $O(\log N)$ for insert/delete/lookup

• The C++11 standard added **unsorted** versions of `multimap` and `multiset` called `unordered_map`, `unordered_multimap`, ...
  – They are implemented using ...

[i.e., run this on a bunch of articles from a corpus (e.g., NYTimes). Do some stemming and stop-word elimination, then compare cosine similarity of the different articles. Those that have high values are probably on similar topics]

Try running this program on an arbitrary text file. You could use the output to do NLP analysis if you go looking for some libraries.
The hash table data structure

- So far, we've seen how to organize large data in various ways
  - The most efficient approach we've seen so far is the BST, which is $O(\log_2 N)$ for insert / delete / lookup when reasonably balanced
- We're now going to examine a data container that is $O(1)$ for insert / delete / lookup, but at the expense of being unsorted
  - Sounds crazy and magical, eh?

Hashing in a nutshell

- A **hash table** is simply a vector of $k$ slots (buckets/bins)
  - The slot type might be a struct instance or a pointer to a struct e.g., vector<studentRecord> hashTable(k);
- A **hash function** take a key value (e.g., student number) and calculates a valid bucket index
  
  $hash : key \rightarrow [0..k-1]$

  - We do not care about **locality**
  i.e., if $key1$ is "close to" $key2$, we don't care if $hash(key1)$ is close to $hash(key2)$; in fact, spread is generally a **good** thing

A simple example

- Suppose I have a set of about 400 student records that I want to be able to access quickly via student #
  - Student records have name, student number, addresses, lists of grades etc.
  - We assume that student numbers are unique and that the last three digits are reasonably randomly distributed
  - We'll also assume we have 1000 slots or "buckets" in which to store the records, labelled from 0..999

[1000 is not a large number, but it makes this solution easier to explain]
A simple example

• So here's the simple algorithm:
  – You tell me the last three digits of the student number, and I will put
    the record in that bucket
    e.g., if the digits are 837, then I will put the record in bucket 837
  – That's it!

• What's the problem with this approach?
  – A collision occurs when different input elements map to the same
    bucket
  – But if we have 400 records, what is the chance of a collision, do you
    think? Well 400 is close to 366 and …

Collisions are common!

• If there are N people in a room, how likely is it that at least two share
  the same birthday (ignoring the year)? Let's assume perfectly random
  distribution?
  – 100% probability with N=367
  – 99.9% with N=70
  – 99% probability is reached with N=57
  – 50% probability with N=23

• If 2500 keys are hashed into a million buckets, even with a perfectly
  uniform random distribution, according to the "birthday problem" ...
  – ... there is a 95% chance of at least two of the keys being hashed to the
    same slot

What to do?

• Right, collisions. So what could we do?
  – Closed hashing: Use some strategy to find another bucket in the table
  – Open hashing: Each bucket is actually a ptr to short linked list / BST of
    records

• Here's a simple idea:
  – For insert, if the desired bucket is full, just go to the next one and
    the next one until you find an empty slot
  – This is called closed hashing using linear probing

Closed hashing with linear probing
(aka open addressing)

• On lookup, look where the hash function tells you to go, then start
  probing until
  – you find what you're looking for, or
  – you find an empty slot (and thus you know it's not there), or
  – you've circled around the table (if the table is full, & you didn't find it)

• Notes:
  – If N << K this can work well, at the cost of some wasted space
  – There are other "probing" approaches for finding the next place to
    look e.g., use a secondary (& tertiary, if needed) hash function
Closed hashing with linear probing
(aka open addressing)

- Problems:
  - It’s bounded in the number of possible elements we can store (unless we grab a bigger vector and copy over every so often, which we could do)
  - As N (# records) approaches K (# buckets), insert takes longer and longer

- Also, suppose we need to handle deletes?
  - This wrecks the lookup assumption! Basically, you should never truly delete items, you just mark them as zombies and keep them in place
  - Future inserts can/should overwrite zombies, but lookups treat them as present but uninteresting
  - An amortized approach: Every so often, build a new hash table with the only the non-deleted (non-zombies) values

```cpp
// An implementation of closed hashing with linear probing
enum Status {EMPTY, ACTIVE, ZOMBIE};
const int NOT_FOUND = -1;
struct Node {
  string name;
  int snum;
  Status status;
};
typedef vector<Node> HashTable;

// numActiveElements should really be part of // the table variable
int numActiveElements;

void initHT (HashTable& table, int K)  {
  table.resize(K);
  numActiveElements = 0;
  for (int i=0; i<K; i++) {
    table[i].status = EMPTY;
  }
}

// While this is a legal hash function, its "spread" // is likely to be pretty awful
// We assume key >= 0 and numBuckets > 0
int myhash (int key, int numBuckets) {
  return key % numBuckets;
}

void insertHT (HashTable& table, string name, int snum) {
  cerr << "Insert: " << name << " " << snum << " endl;  
  const int numBuckets = table.size();
  assert (numActiveElements < numBuckets);  
  int slot = myhash (snum, numBuckets);
  // Stop if you find a zombie slot or a never-used slot
  while (ZOMBIE != table[slot].status  
    && EMPTY != table[slot].status) {
    slot = (slot + 1) % numBuckets;
  }
  table[slot].snum = snum;
  table[slot].name = name;
  table[slot].status = ACTIVE;
  numActiveElements++;
}
// Return NOT_FOUND (i.e., -1) if not found
int lookupIndexHT (const HashTable & table, int snum) {
    cerr << "Looking up index: " << snum << endl;
    const int numBuckets = table.size();
    int slot = myhash (snum, numBuckets);
    for (int i = 0; i<numBuckets; i++) {
        if (EMPTY == table[slot].status) {
            // If we find an EMPTY slot, the key's not there
            return NOT_FOUND;
        } else if (ACTIVE == table[slot].status && snum == table[slot].snum) {
            return slot;  // Found it!
        }
        // else it's a ZOMBIE or (ACTIVE and not a match),
        // so keep going
        slot = (slot + 1) % numBuckets;
    }
    // We made it around the horn but didn't find the element
    return NOT_FOUND;  // == -1
}

// Simple! lookupIndexHT does the real work here!
void removeHT (HashTable & table, int snum) {
    cerr << "Remove: " << snum << endl;
    const int index = lookupIndexHT (table, snum);
    assert (NOT_FOUND != index);
    table[index].status = ZOMBIE;
    numActiveElements--;
}

// Print to stdout, not stderr; OUTPUT IS NOT SORTED!!
void printHT (const HashTable & table) {
    cout << "Printing table" << endl;
    for (int i=0; i<(int)table.size(); i++) {
        if (ACTIVE == table[i].status) {
            cout << "    " << i << "    ";
            cout << table[i].snum << "    ";
            cout << table[i].name << endl;
        }
    }
}

bool hasHT (const HashTable& table, int snum) {
    return NOT_FOUND != lookupIndexHT(table, snum);
}

string lookupHT (const HashTable& table, int snum) {
    cerr << "Lookup: " << snum << endl;
    const int index = lookupIndexHT (table, snum);
    if (index == NOT_FOUND) {
        return ""; // lame
    } else {
        return table[index].name;
    }
}

Some (not very random) data:

Insert (986137, "Zainab")
Insert (983145, "Estelle")
Insert (984003, "Pourya")
Insert (984137, "Lijie")
Delete (984137)
Lookup (987137)
Insert (982137, "Tranh")
The cost of closed hashing

- "Linear probing" means keep adding one to the index and check again
  - There are other probing techniques for finding the next place to try if you have a collision in closed hashing; see CS240.

- Typically, with closed hashing, you allocate space for all of the table entries in the table itself at the beginning
  - This can be wasteful if the table is mostly empty much of the time
  - But you don’t have to allocate new Nodes with each insert, which may be preferred to improve performance during active use
    i.e., you pay the allocation cost once, up front at the beginning
    BUT if you do run out of space, then you typically grab a new vector (with 2X the # of elements) and insert each of the old elements into the new table
  - Also, accessing an element’s value does not require a pointer dereference, which is very, very slightly faster

A puzzler

- Eastbound and westbound buses both run every ten minutes

- I go out at a random time each morning, and I take the first bus I see, but I end up going eastbound 90% of the time.
  - How is this possible?

- Because:
  - The eastbound buses come at :09, :19, :29, :39, :49, :59
  - The westbound buses come at :10, :20, :30, :40, :50, :00

The "constant time" promise

- As the table gets fuller and fuller, you will end up with more and more probing being done
  - If the table is 95% full, you have only a 5% chance of hitting an available bucket on the first try
  - If a table is almost full of ACTIVE elements, insert approaches O(N) ⊗
    - Also, this "clustering" affects which empty buckets will be chosen
    - That is, the EMPTY buckets are not randomly distributed
  - If a table has only a few EMPTY (unused) slots, lookup (and thus also remove) approaches O(N) ⊗
  - The text books say that the load factor (% of elements that are ACTIVE) should be less than 80% to be reasonably efficient

Linear probing isn't very effective as the table fills up

- Consider this card shuffling algorithm
  - I start with 52 cards in one pile
  - for i:1..52
    - I randomly pick a number from 1..52
    - If that card is still in the old deck, I put it on top of the new pile
    - If that card is already gone, I just look at the next slot, and so on until I find a card to remove
Another strategy: Open hashing with chaining

- We'll borrow the LOL (list-of-lists) idea from our priority queue implementation
  - Each bucket holds a ptr (to a list) rather than an element
  - This is called open hashing with chaining

// An implementation of open hashing with chaining

```cpp
struct Node {
    string name;
    int snum;   // student number, i.e., "otherStuff"
    Node* next;
};

typedef vector<Node*> HashTable;

void initHT (HashTable& table, int K) {
    table.resize(K);
}

// This is likely a terrible hash function,
// but it does "work"
int myhash (int key, int numBuckets) {
    return key % numBuckets;
}

void nukeHT (HashTable & table) {
    for (int i=0; i<(int)table.size(); i++){
        Node* p = table[i];
        while (nullptr != p) {
            Node* temp = p;
            p = p->next;
            delete temp;
        }
        table.resize(0);
    }
}
### Some analysis

- **We assume K buckets and N records**
  - Best case: no collisions, O(1) lookup
  - Worst case: All inputs map to same bucket, O(N) lookup

- **If we have a "perfect" hash function and N<=K then there will be no collisions**
  - More likely, there will be some collisions
  - If N>K we are guaranteed to have some collisions, and we hope the results are not too lumpy
  - Hidden structure in the input data will lead to lumpy collision sets
    e.g., all student numbers in W11 begin with 203 or 204, so using first three digits would be disastrous!
Some analysis

- I ran it over the student numbers for W11 (130 records) using last two digits as hash function
  - Max # overflow: 4 (= max number of comparisons)
  - # empty buckets: 28/100
  - Not bad but not great
  - This was a brain-dead approach, we can probably do better

Hashing in general

- A hash table consists of
  - a hash table (vector-ish),
  - a hash function, plus
  - a strategy for dealing with collisions

- Hash function maps some data set to a key range
  i.e., convert data to a number, then scale into [0..K-1]
  - In our case, student number to an int in [0..999]
  - We could map the student names instead, and we don't actually need uniqueness in the domain!
    - But of course two students with the same name will map to the same bucket

Important characteristics of a hash function

1. Deterministic, based on key value [absolute requirement!]
   - Must get the same answer for the same input!
     - Else lookup later will fail
   - So we can't just assign a random number tag to each incoming element and spray data at the buckets based on that.
   - Instead we want to take some intrinsic property of the input data that feels random and use that to select the bucket.
2. Good, even-ish spread of results over buckets ("uniformity")
3. Cheap to compute
4. Supports a variable range
   i.e., easy to adapt if # of buckets changes
More notes on hashing

• Hashing does not require that elements have unique keys
  - But each distinct key value will map to the same place, obviously ...
  - ... so if you have a lot of elements with duplicate keys, your will have a lot of collisions
    e.g., using first three chars of UWid as key would "work" but not well

• Hashing doesn't work well if you want to find all elements in a range of values
  e.g., print the elements in alphabetical order
  e.g., return all entries where "Clark" <= key <= "Gilmour"
  - ... because slightly different keys will usually map to wildly different bucket indexes with a good hash function

Some more examples

• Human names and natural language words are very lumpy in their distribution
  - Need to spread them out. English: lots of S's but not so many X's

• One commonly suggested hash for character strings:
  - Sum the ASCII values of the characters, then mod by K
    - This is an absolutely awful hash function; it has terrible performance on English language words (including names)
    - It was suggested in the original edition of The C Programming Language by Kernighan and Ritchie (aka K&R), hence its fame, alas.

// Sum the ASCII values (from K&R)
// Note: This is a terrible hash function!
int myhash2 (string key, int numBuckets) {
    int sum = 0;
    for (int i=0; i<(int) key.size(); i++) {
        sum += (int) key[i];
    }
    return sum % numBuckets;
}

Some data for fun

• /usr/share/dict/words on OSX, over 210K English words, once proper nouns are removed

• Using "sum the ASCII values" as a hash function
  - 1K buckets: 0% empty, avg=210, med=177, max=619
  - 10K buckets: 80% empty, avg=21, med=0, max=610
    - Shortest word: "a", ASCII sum 97
    - Longest word was 24 chars, max ASCII value 122, 24*122=2888
    - No hash sum was less than 97 or more than 2621!
  - 100K buckets: 98% empty, avg = 2, med = 0, max = 610
    - No more buckets used! The decrease in average isn't really informative
Some data for fun

- Using a simple hash I found on the web in 5 minutes:
  - 1K buckets: 0% empty, avg=210, med=211, max=256
  - 10K buckets: 0% empty, avg=21, med=21, max=40
  - 100K buckets: 12% empty, avg=2, med=2, max = 11
  - 200K buckets: 34% empty, avg=1, med=1, max = 8
  - 1M buckets: 81% empty, avg=0, med=0, max = 5

- Moral: Need to make sure you have a good spread over all of the range.
  - Modular arithmetic is your friend.
  - It's good to go around the loop a few times.

Improving the odds

Two basic things you can do:

1. Devise a better hash function
   - Need to study the data, run experiments, read some math books, ...
   - Good hash functions can greatly improve performance

2. Increase the number of buckets
   - But you waste space if you have lots and lots
   - Some hash approaches are "amortized constant time", allowing the table to double in size every so often

Hashing: Error correction

- Hashing often used for error correction:
  - Suppose we're downloading a text file over a noisy line.
  - Compute the string hash of the file at the source and destination, then compare hashes
    - If they match, highly likely the transfer was OK
    - If they don't, you know something is wrong
  e.g., MD5 hash values are 32 bit integers, in wide use tho not for security any more

Hashing: Code cloning

- Copy/paste code is pretty common in large industrial software systems
  - But checking each piece of code against each other is very expensive

- Better idea:
  - Take hash of each function (or each line, or each token, maybe after some "normalization")
  - When two functions have same hash value, do detailed comparison
  - Much cheaper to compare two numbers than two sequences of code
  - If there is duplicated code, notify manager / developer that they may be a refactoring opportunity here (i.e., duplication could be cleaned up)
Hashing: Plagiarism detection

1. Normalize the source code: transform all identifiers into <ID>, all constant values (strings, numbers, etc) into <VAL>, etc.
   - So changing variable names doesn't help you

2. Hash each line of code; a procedure is now modelled as a set of hashes

3. Compare each function of each student against every other one by just checking if the sets have overlaps of hash values
   - Permuting statements / adding useless code doesn't help you
   - Sets themselves can be implemented using hash tables, so this is quite efficient

4. In the case where there is significant overlap of hash values, do a detailed analysis of the suspected procedures using a more expensive comparison method

Hashing: More to come

- We'll revisit hashing a few more times yet
  - We'll re-implement our open hash table using an object-oriented design
  - We'll discuss how to measure the "goodness" of a hash function
  - We'll briefly talk about how hashing can be used in (for example) cryptography and provenance analysis (this won't be on the exam)

4. Graphs, trees, and hash tables

CS138 Winter 2017

Prof. Mike Godfrey
University of Waterloo