Computer Science II — CSci 1200
Lecture 19
Data Structure Summary; Hashing, Part 1

Review from Lecture 18

- The erase function and its effect on the tree
- Iterating through the tree using parent pointers to find the in-order successor
- Advanced recursion:
  - Mergesort
  - Nonlinear search.

Today

Reading: Ford&Topp, Chapter 12 through Section 12.5.

- Summary of data structures we have studied thus far
- Stacks and queues, a quick introduction
- Hashing:
  - Idea
  - Hash functions
  - Hash tables
  - Collision resolution

Data Structures: Fundamental Operations

- **Find:** Find the location of a `key_value` in the data structure (container) or, if the `key_value` is not in the data structure, find the location to insert it.
- **Insert:** Given the required location, insert the `key_value` (plus any other data) into the container.
- **Erase:** Given the location of the `key_value` in the container, remove the `key_value` (plus any other data) from the container.

Some container operations in the standard library include a find operation within the insert or erase function.

Data Structures: Analysis

- We will fill in the table below with the costs of operations, making the distinction between average-case and worst-case numbers of operations, as necessary.
- In studying these, remember that binary search trees are the data structure underlying the `std::map` and `std::set` containers, and linked lists underly the `std::list`

Data Structure	Find	Insert	Erase
Array / vector, unsorted			
Array / vector, sorted			
Linked-list, unsorted			
Linked-list, sorted			
Binary search tree			

Stacks and Queues

- One way to obtain computational efficiency is to consider a simplified set of operations.
- **Stacks** allow access, insertion and deletion from only one end called the *top*
  - There is no access to values in the middle of a stack.
  - Stacks may be implemented efficiently in terms of vectors and lists, although vectors are preferrable.
  - All stack operations are $O(1)$
- **Queues** allow insertion at one end, called the *back* and removal from the other end, called the *front*
  - There is no access to values in the middle of a queue.
  - Queues may be implemented efficiently in terms of a list. Using vectors for queues is also possible, but requires more work to get right.
  - All queue operations are $O(1)$
- Stacks and queues are covered in this week's lab.

Hidden Costs

- Linked lists (`std::list`) and binary search trees (including `std::set` and `std::map`) include dynamic memory allocation for each `insert` and for each `erase`
- `std::vector` occasionally re-allocates the underlying array, doubling it in size, and copying all of values stored in the vector.
- Memory management operations tend to be substantially more expensive than "ordinary" operations.
- We will do some simple analysis to show that the vector operations are not as expensive as they initially sound.
- The bottom line is that the hidden costs of memory management make vectors more favorable than the order-notation analyis might otherwise show.

More Advanced Data Structures

- Balanced binary search trees remove the worst-case behavior of binary search trees by "rotating" and "rebalancing" the trees to ensure that
  1. The non-decreasing ordering is maintained.
  2. The worst-case height of the tree is $O(\log N)$, making the primary operations each $O(\log N)$.
  3. Red-black trees are one such balanced tree, and are the basis for `std::set` and `std::map`.
- Hash tables break the $O(\log N)$ barrier at the cost of worst-case $O(N)$ behavior:
  - We will look at techniques for avoiding this worst-case behavior except for artificially-constructed cases.
  - Ordering information is also lost in hashing — in fact that is the point!
  - Putting these two comments together produces the observation that *hashing may be used in place of binary search trees when the ordering of keys does not matter.*
  - Hashing will occupy the rest of today's lecture and Lecture 20.
- Priority-queues are a mixture of trees and queues, where the order in which items are removed depends on a "priority value" assigned to the values as opposed to the order in which the values are inserted.
  - Priority queue operations will have a worst-case time of $O(\log N)$, with an average case closer to $O(1)$.
  - Priority queues are the focus of Lecture 21 and 22.

Hashing

- Given are:
  - Key values, perhaps with associated data values (as in a map).
  - A function f, mapping key values to the integer range $0, \ldots, N-1$.
  - A table (vector or array) of size N.
- For each key value, k, to be stored, compute

$$i = f(k)$$

  and store k and its associated value at location i of the table.
- Simple example:
  - Keys are just integers, `k`, values are strings, `s`.
  - `f(k) = abs(k)%N`
  - Store each pair `<k,s>` at table location `abs(k)%N`.
- Questions:

- What is a good design for `f`, the *hash function*?
- What happens when two keys map to the same table location? This is referred to as a *collision*?

Answering these two questions will be the focus of the rest of our discussion on hashing.

- Applications:
  - Compilers storing variable names ("symbol tables")
  - Routing tables
  - Database indexing
  - File locations in a memory system

  Hashing may be used in place of balanced binary search trees when ordering of the keys is not required.

Hash Functions

- Goals:
  - Fast computation
  - A random, uniform distribution of keys throughout the table, *despite the distribution of keys that are to be stored.*
- Our `f(k) = abs(k)%N` example satisfies the first requirement, but may not satisfy the second.
- An example of a dangerous hash function on strings is to add or multiply the ascii values of the individual keys:

```
unsigned int hash( string const& k, unsigned int N );
{
  unsigned int value = 0;
  for ( unsigned int i=0; i<k.size(); ++i )
    value += k[i];  // conversion to int is automatic
  return k % N;
}
```

  The problem is that different permutations of the same string result in the same hash table location.

- This can be improved through multiplications that involve the position and value of the key:

```
unsigned int hash( string const& k, unsigned int N );
{
  unsigned int value = 0;
  for ( unsigned int i=0; i<k.size(); ++i )
    value = value*8 + k[i];  // conversion to int is automatic
  return k % N;
}
```

- This is better, but can be improved further. The theory of good hash functions is quite involved.

Two Approaches to Collision Resolution

Two classes of approaches to collision resolution are commonly used:

- In *open addressing*, when a table location already stores a key (and its associated value, if any), a different table location is sought in order to store the new value.
- In *separate chaining*, each table location stores a list or vector of the keys (and values) that are hashed to that location.

Collision Resolution: Open Addressing

- Three approaches to handling a collision during an *insert* operation.
  - *Linear probing:* if `i` is the hash location then the following sequence of table locations is tested

    ```
    (i+1)%N, (i+2)%N, (i+3)%N, ...
    ```

    until an empty location is found.
  - *Quadratic probing:* if `i` is the hash location then the following sequence of table locations is tested ("probed"):

    ```
    (i+1)%N, (i+2*2)%N, (i+3*3)%N,  (i+4*4)%N, ...
    ```

    More generally, the j^{th} "probe" of the table is

    $$(i + c_1 j + c_2 j^2) \mod N$$

    where c_1 and c_2 are constants.
  - *secondary hashing*: when a collision occurs a second hash function is applied to compute a new table location. This is repeated until an empty location is found.
- For each of these approaches, the *find* operation follows the same sequence of locations as the *insert* operation. The key value is determined to be absent from the table only when an empty location is found.
- The *erase* function must mark a location as "formerly occupied". If a location is marked empty, instead, *find* may fail. Formerly-occupied locations may (and should) be reused, but only after the *find* operation has been run to completion.
- Problems with open addressing:
  - Slows dramatically when the table is nearly full (e.g. about 80% or higher). This is particularly problematic for linear probing.
  - Fails completely when the table is full.
  - Cost of computing new hash values.
- We will investigate ways to handle the first two problems in Lecture 20.

Collision Resolution: Separate Chaining

- Each table location stores a list of keys (and values) hashed to it.
- Thus, the hashing function really just selects the list to check.
- This works well when the number of items stored in each list is small, e.g. an average of 1.
- Other data structures, such as binary search trees, may be used in place of the list, but these have even greater overhead considering the number of items stored.

Hash Tables: Summary

- Good hash function is crucial
- Two types of collision resolution
- Average case is $O(1)$ for insert, find and erase when