Lecture 22: Hashing
CSCI 700 - Algorithms I

Andrew Rosenberg
Last Time

- Introductory Hashing
Open Addressing Hashing
Recap

- Hash Tables allow us improved expected lookup and deletion
- Performance is dependent on the load factor $\alpha = nm$
- ...and a good hash function.
Collisions

Chaining is a common and simple approach to avoiding collisions.
Chaining overhead

- Chaining requires additional memory to store each pointer.
- To store an integer, takes one 32 or 64 bits for the integer, and the same again for the pointer to the next one.
- Can we do better?
Open addressing

- Open Addressing stores the data in the hash table directly.
- By eliminating the need for $n$ pointers, the size of the hash table $m$ can be increased to store $\frac{\text{sizeof}(A)}{n}$ more elements.
- With the same amount of memory, the load factor, $\alpha$ is reduced.
Open addressing avoids collisions by storing the collision “chains” in the hash table itself.

This could be done explicitly, but the memory improvements will be lost.

Instead for each key, $k$, construct a **probe sequence**.

The probe sequence describes a permutation of every element in the hash table.

A new element will be inserted in to the earliest unoccupied index in the probe sequence.
Open addressing example

A = (0, 1, 2, 3)
B = (0, 2, 1, 3)
C = (1, 3, 0, 2)
D = (1, 2, 3, 0)
E = (3, 1, 2, 0)
Open addressing example

A = (0, 1, 2, 3)
B = (0, 2, 1, 3)
C = (1, 3, 0, 2)
D = (1, 2, 3, 0)
E = (3, 1, 2, 0)
Open addressing example

A = (0, 1, 2, 3)
B = (0, 2, 1, 3)
C = (1, 3, 0, 2)
D = (1, 2, 3, 0)
E = (3, 1, 2, 0)
Open addressing example

A = (0, 1, 2, 3)
B = (0, 2, 1, 3)
C = (1, 3, 0, 2)
D = (1, 2, 3, 0)
E = (3, 1, 2, 0)
Open addressing insertion

Hash-Insert(T,k)

\[ i = 0 \]

repeat

\[ j = h(k,i) \]

if \[ T[j] == NIL \] then

\[ T[j] = k \]

return \[ j \]

else

\[ i = i + 1 \]

end if

until \[ i == m \]
Open addressing searching

Hash-Search(T,k)

\[
i = 0\\
\text{repeat}\\
\quad j = h(k,i)\\
\quad \text{if } T[j] == k \text{ then}\\
\quad \quad \text{return } j\\
\quad \text{end if}\\
\quad i = i + 1\\
\text{until } T[j] == \text{NIL or } i == m
\]
With chaining, deletion was easy. Just delete the element in the linked list.

With open addressing, to delete, replace the deleted element with a special tag `DELETED`.

However, then search will need to continue looking along a probe sequence passed `DELETED` entries.

In this case, the search time is independent on $\alpha$. A hash table with few elements can take longer to find an element than a densely packed table, if many elements have been deleted.

If deletion is common, chaining is preferred.
A probe chain, \( h(k, i) \), is a **permutation** of the hash table entries 0 to \( m-1 \).

How can we construct a permutation for a new element quickly?

**Linear Probing**

\[
h(k, i) = h(k) + 1 \mod m
\]

Lots of overlap

if \( h(k') = h(k) + 1 \), \( n-1 \) probe chain elements overlap in order.
How can we construct a permutation for a new element quickly?

**Quadratic Probing**

\[ h(k, i) = (h(k) + c_1 i + c_2 i^2) \mod m \]

If there is a collision, \( h(k) = h(k') \), then \( h(k, i) = h(k, i+1) \).
That is, if two entries have the same first key, they have the same \( n \) keys.
How can we construct a permutation for a new element quickly?

**Double Hashing**

\[ h(k, i) = (h_1(k) + ih_2(k)) \mod m \]

Successful if both hash functions are good.
There are $m!$ possible permutations of $m$ entries. How many are possible with:

- Linear Probing?
- Quadratic Probing?
- Double Hashing? Assume $h_1(k)$ is independent of $h_2(k)$. 

Perfect Hashing

- Hashing provides excellent expected case access insertion and deletion, but can have poor worst-case performance.
- If the set of keys is static, we can have excellent worst case performance as well.
- Static keys: DVD filenames, reserved keywords.
Perfect Hashing

- Construct a regular hash table with $n$ entries. – some entries will be empty.
- Identify a top level hash function $h$.
- Rather than chaining, use a second hash table at each cell.
- For non-empty cells with $n_i$ elements, allocate a new hash table with $n + i^2$ elements.
Perfect Hashing Example
Perfect Hashing

- The sparse table makes it easy to define a hash function which generates no collisions for a particular set of keys. Only $\frac{1}{n}$ cells will be occupied.
- Use an $(a_i k + b_i) \mod m_i$ hash function for each cell. If there are collisions, try a different assignment of $a_i$ and $b_i$ until one yields no collisions.
- Most values of $a_i$ and $b_i$ will yield no collisions.
- This guarantees constant time look up.
- The trade off is that insertion and deletion are outrageously slow. Makes this only useful for static keys.
Bye

- Next time
- Multi-threaded Algorithms