Announcements

- HW1 PAST DUE
- HW2 online: 7 questions, 60 points
- Nat’l Inst visit Thu, ok?

Last time:

- Continued PA1 Walk Through
- Dictionary ADT: Unsorted
- Hashing

Today:

- Finish up hashing
- Sorted Dictionary ADT: Binary search, divide-and-conquer
- Recursive function and recurrence relation
Hash Table and Hashing

Reference items in a table (the hash table) by keys

Use arithmetic operations to transform keys into table addresses (buckets)

Two issues in hashing:

1. construction of the hash function
2. policy for collision resolution
Components of Hash Functions

Hash function \((h())\): transforms the search key into a bucket index

- **hash code**: \(t(key) \rightarrow \text{intmap}\)
  maps the key (which could be a string, etc.) into an integer (intmap)

- **compression map**: \(c(\text{intmap}) \rightarrow \text{bucket}\)
  maps the intmap into an address within the table, i.e.,
  into the range \([0, M - 1]\), for table of size \(M\)

Given a key: \(h(key) \rightarrow c(t(key)) \rightarrow \text{bucket/index}\)
Hash Functions

Common parts of hashing function:

- **truncation** (hash code): remove parts common across keys, or extract bits from keys that are more likely to be unique across keys. Examples:
  - 734 763 **1583**
  - 141.213.8.**193**
  - **cello.eecs.umich.edu**

- **folding** (hash code): partition the key into several parts and combine them in a “convenient” way

- **modulo arithmetic** (compression map)
Example of Folding: Polynomial Hash Code

Polynomial hash code \( t \) takes positional info into account:

\[
t(x[]) = x[0]a^{k-1} + x[1]a^{k-2} + \ldots + x[k-2]a + x[k-1]
\]

If \( a = 33 \), the intmap’s are:

- \( t(“tops”) = \)
- \( t(“pots”) = \)

Good choices of \( a \) for English words: \{33, 37, 39, 41\}

What does it mean for \( a \) to be a good choice?
Why are these particular values good?
Compression Mapping

- division method: $|\text{intmap}| \mod M$, $M$ table size (prime)
- MAD (multiply and divide) method:
  $|a \times \text{intmap} + b| \mod M$, $a$ and $b$ non-negative integers
Hashing: Collision Resolution

With hashing, collision is inevitable

What do you do when two items hash into the same bucket/bin?

Methods of collision resolution:

- separate chaining
- scatter table
  - coalesced chaining
  - open addressing:
    - linear probing
    - quadratic probing
    - double hashing
- extensible hashing: dynamic hashing
Collision Resolution

Separate chaining:
let each bin points to a linked list of elements (see Fig. 8.3)

Coalesced chaining:
- store linked list in unused portion of the table
- if item hashes to an already occupied bin, follow the linked list and add item to the end (coalesced chaining, see Fig. 8.4)
- hash table can only hold as many items as table size

Open addressing: if there’s a collision, apply another hash function repeatedly until there’s no collision. Set of hash functions:
\{h_0, h_1, h_2, \ldots\}
Open Addressing

To probe ::= to look at an entry and compare its key with target

**Linear probing:** \( h_i(key) = (h_0(key) + i) \mod M \)
do a linear search from \( h_0(key) \) until you find an empty slot
(see Fig. 8.6, compare against Fig. 8.4)

Clustering: when contiguous bins are all occupied
Why is clustering undesirable?

Assuming input size \( N \), table size \( 2N \):
What is the best-case cluster distribution?
What is the worst-case cluster distribution?
What’s the average time to find an empty slot in both cases?
Open Addressing (cont)

**Quadratic probing**: \( h_i(key) = (h_0(key) + i^2) \mod M \)
less likely to form clusters, but only works when table is less than half full
because it cannot hit every possible table address

**Double hashing**: \( h_i(key) = (h(key) + ih'(key)) \mod M \)
use 2 distinct hash functions

Removal on scatter tables: complicated, don’t want to move an element
up the table beyond its actual hash location; must rehash the rest of
chain/cluster, or otherwise mark deleted entry as “deleted” (see Fig. 8.5)
Timing Analysis

Differentiate between successful \( S() \) and unsuccessful \( U() \) search

Example: passwd lookup wants success to be quick, failure can be (is desirable to be) slow

What’s the time complexity of search for separate chaining?

Worst case:

- \( S() = \)
- \( U() = \)
Average Case: Load Factor

Load factor: $\lambda = \frac{N}{M}$, where $N =$ number of items, $M =$ hash table size

- empty table, $\lambda = 0$
- half full, $\lambda = 0.5$
- full, $\lambda = 1$
- can $\lambda > 1$?

If hash function is good, length of average chain in separate chaining is $\lambda = \frac{N}{M}$ (number of comparison reduced by a factor of $M$ if using separate chaining)

Given $\lambda$,

- average unsuccessful search time $U(\lambda) = 1 + \lambda$
- average successful search time $S(\lambda) = 1 + \lambda/2$

Sugih Jamin (jamin@eecs.umich.edu)
Average Case Search Times (FYI)

Coalesced chaining:

- \( U(\lambda) = 1 - \frac{1}{4}(e^{2\lambda} - 1 - 2\lambda) \); For \( \lambda = 1 \), \( U(1) = 2.1 \)
- \( S(\lambda) = 1 - \frac{1}{8\lambda}(e^{2\lambda} - 1 - 2\lambda) + \frac{\lambda}{4} \); For \( \lambda = 1 \), \( S(1) = 1.8 \)

Linear probing:

- \( U(\lambda) = \frac{1}{2}(1 + (\frac{1}{1-\lambda})^2) \)
- \( S(\lambda) = \frac{1}{2}(1 + \frac{1}{1-\lambda}) \)

Quadratic probing and double hashing:

- \( U(\lambda) = \frac{1}{1-\lambda} \)
- \( S(\lambda) = \frac{1}{\lambda} \log \frac{1}{1-\lambda} \)
Hashing Search Time

**Separate chaining:** increases gradually

**Scatter table:** increases dramatically as table fills,
when table is full, no more keys may be inserted

**Extensible hashing (dynamic hashing):**
Double table size when it is half full
Old entries must be re-hashed into larger table
This is an expensive, but hopefully infrequent, operation
Doubling cost is *amortized* over individual search/insert

Sugih Jamin (jamin@eecs.umich.edu)
When NOT to use Hashing

Rank search: return the $k^{th}$ largest item

Range search: return values between $h$ and $k$

Sort: return the values in order

Readings: done with Preiss Ch. 8
Dictionary ADT

Table lookup: look up a key in a table

The **key** is associated with some other information (**value**)

Key space is usually more structured/regular than value space, so easier to search

An **entry** is a <key, value> pair

For example: <title, mp3_file>

Usually the ADT stores pointers to entries

Sugih Jamin (jamin@eecs.umich.edu)
Runtime

Unsorted list:

- insert: $O()$
- search: $O()$

Sorted list:

- insert: $O()$
- search: $O()$
Binary Search: Iterative Version

Given a sorted list, with unique keys, perform a *Divide and Conquer* algorithm to find a key

Find 31 in \(a[20 \ 27 \ 29 \ 31 \ 35 \ 38 \ 42 \ 53 \ 59 \ 63 \ 67 \ 78]\)

Write an iterative version of binary search:
ibinsearch(int *a, int n, int key),
a[] sorted, n array size, return index of key

What is the time complexity of the algorithm?
Rule 5: Divide and Conquer: An algorithm is $O(\log n)$ if each constant time $O(1)$ operation (e.g., CMP) can cut the problem size by a fraction (e.g., half)

Corollary: If constant time is required to reduce the problem size by a constant amount, the algorithm is $O(N)$
Recursive Function

What is a recursive function?

Must have termination condition

Example recursive function: $n!$

Iterative version (assume $n \geq 0$):

```c
ifact(int n) { for (int i = n-1; i > 0; i--) n *= i; return (n ? n : 1); }
```

What is its big-Oh complexity?

Recursive version (assume $n \geq 0$):

```c
rfact(int n) { return (n ? n*(rfact(n-1)) : 1); }
```

How to compute its big-Oh?
Time Complexity of \texttt{rfact}(n)

Let $T(n)$ be the operation-count complexity of \texttt{rfact}(n)

Which operation shall we count?

What is the value of $T(n)$?

What is the big-Oh complexity of \texttt{rfact}(n)?

Any constant operation count can be replaced by ’1’

And the space complexity?
Recurrence Relation (Accounting Rule 6)

**Definition:** a recurrence relation is a function of $x$ that is recursively defined in terms of the same function on $x'$, where $x' < x$

Examples:

- $T(n) = 1 + T(n - 1), \ T(0) = 1$
- $T(n) = 2 \times T(n/2) + n, \ T(1) = 1$
- etc.

**Accounting Rule 6:**
Recurrence relations are “natural” descriptions of the timing complexity of recursive algorithms
Binary Search: Recursive Version

Write a recursive version of binary search:

```
rbinsrch(int *a, int left, int right, int key),
```
a[] sorted, left index to first element in array,
right index to last element in array, return index of key

What is the time complexity of the algorithm?

Recall: any constant per-level cost can be represented as ’1’
Divide and Conquer

Divide and conquer doesn’t work with linked list, unfortunately

So why use linked-list at all?

How can we speed up search and use dynamically allocated space?