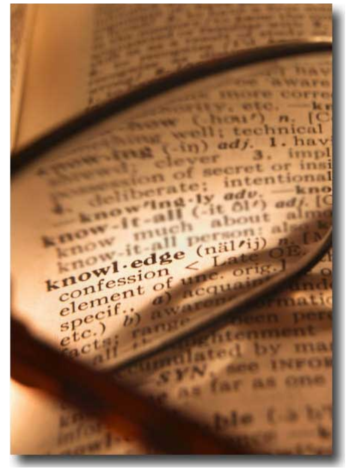
eecs 281 DATA STRUCTURES AND ALGORITHMS

Lecture 5: Dictionary ADT and Hashing
Recurrence Relations

Dictionary ADT



How do you use a dictionary?

Used where you need to do some sort of table lookup:

- search for a key in a table
- the key is usually associated with some data/value of interest

Also known as associative array

Why search for key instead of just searching for the data?

Dictionary ADT

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

Dictionary entry is a <const key_type, data_type> pair

- for example, <title, mp4_file>
- <"Avatar", avatar.mp4>

Normally associate a given key with only a single value or a pointer to data

Dictionary is optimized to quickly add <key, data> pairs, retrieve data by key

Types of Dictionary

Whether items are grouped by some category such as by subject, by popularity, chronologically, etc.

- unordered
- ordered

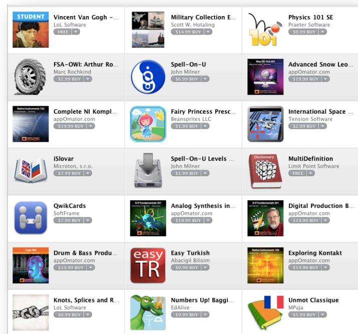
Whether items are listed by a collating sequence of the key, e.g., numerical, alphabetical

- unsorted
- sorted

Adding entries into an ordered (sorted) list must retain the ordered (sorted) property of the list

The AppStore

What type of dictionary do we see at the AppStore?



Unsorted Dictionary Runtimes

Implementation	Search	Insert
Arrays	$O(?)$	$O(?)$
Linked Lists	$O(?)$	$O(?)$
Hashing (amortized)	$O(1)$	$O(1)$

Hashing

Access table items by their keys in relatively constant time regardless of their locations

Main idea: use arithmetic operations (hash function) to transform keys into table locations

- the same key is always hashed to the same location
- such that insert and search are both directed to the same location in $O(1)$ time

Hash table: an array of buckets, where each bucket contains items assigned by a hash function

Hashing Example

In a text editor, to speed up search, we build a hash table and hash each word into the table

Let hash table size (M) $= 16$

Let hash function ($h()$) $=$ (sum all characters) mod 16

- by "sum all characters" we mean sum the ASCII (or UTF-8) representation of the character
- for example, $h(\text{"He"}) = (72+101)\%16 = 13$

Let sample text be the following $N=13$ words:

"He was well educated and from his portrait a shrewd observer might divine"

Hashing Example

(sum all characters) mod 16

He → 13
was → 11
well → 4
educated → 15
and → 3
from → 4
his → 4
portrait → 5
a → 1
shrewd → 13
observer → 8
might → 9
divine → 15

N = 13, M = 16

0	
1	a
2	
3	and
4	well from his
5	portrait
6	
7	
8	observer
9	might
10	
11	was
12	
13	He shrewd
14	
15	educated devine

Collision and Collision Resolution

Collision occurs when the hash function maps two or more items—all having different search keys—into the same bucket

What to do when there is a collision?

Collision-resolution scheme:

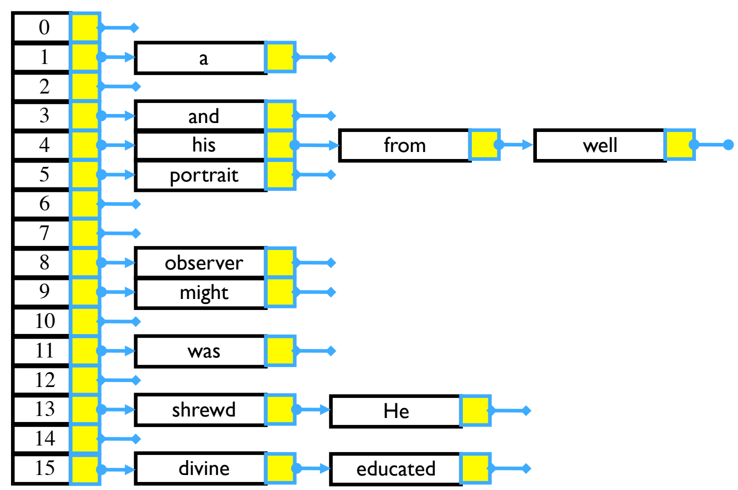
- assigns distinct locations in the hash table to items involved in a collision

Separate Chaining

A collision resolution scheme that lets each bucket points to a linked list of elements

- insertion:
  - compute $k = h(\text{key})$
  - prepend to k^{th} bucket in $O(1)$ time (but may need to check for duplicates)
- search:
  - compute $k = h(\text{key})$
  - search in k^{th} container (e.g., check every element)

Separate Chaining



Performance Analysis

Worst case

- all N elements in one bucket
- searching through a bucket : $O(N)$ time

Average-case analysis:

- table size M, has N keys to store
- average bucket size is N/M
- $L = N/M$ is called the load factor
- average runtime of search: $O(h()) + O(1) + O(L)$
- unsuccessful search: $1+L$ comparisons
- successful search: $1+L/2$ comparisons on average
- for good performance, want small load factor

Why differentiate between successful and unsuccessful search?

How to Improve Runtime?

Can set $M > N$ so that $L < 1$

- then average search time is $O(1)$

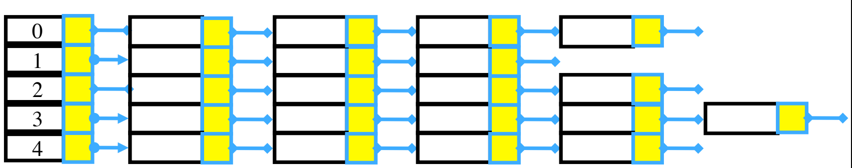
Space-time trade off:

- very large table/array
  - few collisions
  - for the movie title example, can have millions of entries
- small table/array
  - many collisions, may need time to resolve

Table Resizing

If table size is fixed:

- search performance deteriorates with growth (when load factor becomes high)



When load factor becomes too high, resize by doubling the size of hash table

- each entry must be re-hashed, not just moved, into the new hash table
- expensive worst-case; OK if amortized

Table Resizing: Amortized Analysis

Hash table of size $2M$

Assume $O(1)$ operation to insert up to M–1 items: $O(M)$

For the M-th item, create a new hash table of size $4M$: $O(1)$

Rehash all M–1 items to the new table: $O(M)$

Insert new item: $O(1)$

Total cost to insert M items: $O(M + 1 + M + 1) = O(M)$

So, average cost to insert M items is $O(1)$

⇒ Hash table doubling cost is amortized over individual inserts

- though the periodic high cost may not be acceptable to some applications that require smooth running time

Other Ways to Resolve Collisions

Aside from separate chaining, other methods have been proposed for collision resolution

Two main motivations:

1. scatter table: re-use empty spaces in the hash table to hold colliding items: coalesced chaining and open addressing
2. dynamic hashing: grow the hash table incrementally so as not to take the performance hit of rehashing everything when resizing: extendible hashing and linear hashing
   - more complicated hash function to allow for addressing of incrementally grown hash table (not covered)

Coalesced Chaining

Keep the linked list used in separate chaining, but store it in the unused portions of the hash table

Hash table can only hold as many items as table size

If an item hashes to an already occupied bin follow the linked list and add item to the end

If an item is deleted from the linked list, the rest of the list must be "moved up"

- but be careful that an item is not moved up past its original hash bucket

Coalesced Chaining

(sum all characters) mod 16

He → 13
was → 11
well → 4
educated → 15
and → 3
from → 4
his → 4
portrait → 5
a → 1
shrewd → 13
observer → 8
might → 9
divine → 15

$N = 13, M = 16$

"portrait" must never be moved higher than 5

0	divine
1	a
2	
3	and
4	well
5	from
6	his
7	portrait
8	observer
9	might
10	
11	was
12	
13	He
14	shrewd
15	educated

Item Removal

Removal on scatter tables is complicated:

- must not move an element up the table beyond its actual hash location
- must rehash the rest of chain
- example: int%9

	01	~~11~~	02	21	05	31		

then 11 is deleted, 05 cannot be moved up beyond position 5

	01	02	21	31	05			

- or otherwise mark deleted entry as "deleted" (but not empty)

	01	del	02	21	05	31		

Open Addressing

Idea: if there's a collision, apply another hash function from a pre-determined set of hash functions $\{h_0, h_1, h_2, \ldots\}$ repeatedly until there's no collision

To probe: to compare the key of an entry with search key

Linear probing:
$h_i(\text{key}) = (h_0(\text{key})+i) \bmod M$
do a linear search from $h_0(\text{key})$ until you find an empty slot

0	devine
1	a
2	
3	and
4	well
5	from
6	his
7	portrait
8	observer
9	might
10	
11	was
12	
13	He
14	shrewd
15	educated

Open Addressing

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 each case?

Open Addressing

Quadratic probing: $h_i(\text{key}) = (h_0(\text{key})+i^2) \bmod M$
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: $h(\text{key}) = (h_1(\text{key})+ih_2(\text{key})) \bmod M$
uses 2 distinct hash functions

Hash Functions

From Webster dictionary:
Main Entry: hash
Etymology: French *hacher*, from Old French *hachier*, from *hache* battle-ax, of Germanic origin;
akin to Old High German *hAppa* sickle;
akin to Greek *koptein* to cut – more at CAPON
1a : to chop (as meat and potatoes) into small pieces

Hash function ($h()$) maps search keys to buckets, in two steps:
- maps the key into a hash code: $t(\text{key}) \rightarrow$ hashcode
- compression map, maps the hashcode into an address within the table: $c(\text{hashcode}) \rightarrow$ bucket, i.e., maps into the range $[0, M-1]$, for table of size M

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

Hash Functions

Criteria for a good hash function:

- must compute a hash for every key
- must compute the same hash for the same key
- easy and quick to compute
  recall: average runtime of search: $O(h()) + O(1) + O(L)$
- involves the entire search key
- scatters "similar" keys that differ slightly
- minimize collision
  - distribute keys evenly in hash table

Good hash function $=$ avoiding worst case

- we cannot guarantee this
- but can improve statistics
  by ensuring that buckets are used equally

Hash Functions

Common parts of hash functions:

- truncation (hash code): exaggerate parts of key that are more likely to be unique across keys (but hash function must still involve entire key!), e.g.,
  - 734 763 1583
  - 141.213.8.193
  - girard.eecs.umich.edu
- folding (hash code)
- modulo arithmetic (compression map)
  - a cheap way to reduce collision:
    make hash table size a prime number
  - compression map: $c(t(\text{key})) = t(\text{key}) \ \% \ \text{prime\_size}$
  - consequence: avoid "regular" collisions, e.g.,
    $140 \ \% \ 100 = 240 \ \% \ 100 = 1040 \ \% \ 100 = 40$

Hash Strings: Attempt 1

How to hash keys that are not integers?

- string: use the ASCII (or UTF-8) encoding of each char
- float: treat it as a string of bits
- images, viral code snippets, malicious Web site URLs:
  in general, treat the representation as a
  bit-string and sum up or extract parts of it

Let's look at our string hashing function again, this time with a prime number hash table size:

$h() =$ (sum all characters) mod 17

How would the following strings hash?
"stop", "tops", "pots", "spot"

Hash Strings: Attempt 2

Polynomial hash code 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 hashcodes are:

- $t(\text{"listen"}) = \text{'l'}*33^5 + \text{'i'}*33^4 + \text{'s'}*33^3 + \text{'t'}*33^2 + \text{'e'}*33 + \text{'n'}$
- $t(\text{"silent"}) = \text{'s'}*33^5 + \text{'i'}*33^4 + \text{'l'}*33^3 + \text{'e'}*33^2 + \text{'n'}*33 + \text{'t'}$

This is operation is known as folding: partition the key into several parts and combine them in a "convenient" way

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?

Birthday Paradox

What is the smallest number of people in a room for a better-than-even odds (probability ≥ 0.5) that two persons share the same birthday?

Assumptions:

- 366 days to a year
- birthdays are independent (no twins)
- birthdays are equally likely (actually more likely 9 months after a holiday)

Birthday Paradox

Probability that each person in the room has a birthday different from all the other persons in the room:

Probability for the 1^{st} person: 1

Probability for the 2^{nd} person: $\frac{366-1}{366}$

Probability for the 3^{rd} person: $\frac{366-2}{366}$

$\dots$

Probability for the j-th person: $\frac{366-(j-1)}{366} = \frac{367-j}{366}$

Birthday Paradox

Assuming independence, the probability that all k people in the room have different birthdays is:

$$p_k = 1 \cdot \frac{365}{366} \cdot \frac{364}{366} \cdot \ldots \cdot \frac{367-k}{366}$$

The probability that not (all k people in the room have different birthdays), i.e., at least 2 out of the k persons have the same birthday is: $\varepsilon = 1 - p_k$

Birthday Paradox

$\varepsilon = 1 - p_k$

By brute force calculations, we find that: for $k = 22$, $\varepsilon \approx 0.475$, for $k = 23$, $\varepsilon \approx 0.506$

So you only need 23 people in a room for 2 persons to share the same birthday!

More generally, $k \approx \sqrt{2M \log \frac{1}{1-\varepsilon}}$

for $\varepsilon = 0.5$, $k \approx 1.17\sqrt{M}$

For the birthday paradox, $M = 366$

Hashing Collision

How many items (k) does it take to hash two items into the same bucket for a table of size M, with probability ≥ 0.5?

Assuming:

- items are independent
- all possible items are equally likely (clearly not true for English words, for example)

For:

$M = 7, k = 4$
$M = 9, k = 4$
$M = 11, k = 4$
$M = 2^{40}, k = 1\ 226\ 834$
$M = 2^n$, it takes on the order of $\sqrt{M}$ or $2^{n/2}$

Study Questions

1. What is the difference between sequential and associative containers ?
2. What is a hash function ?
3. What makes a good hash function ?
4. What is a hash table ?
5. Why do hash functions for strings use polynomial code?
6. What is the worst-case complexity of hash-table ops?
7. What is load factor and how does it affect complexity of hash-table ops?
8. Why does one use average-case and amortized complexity to evaluate hash-table ops ?

Sorted Dictionary

What kind of operations can we not do with an unsorted dictionary?

Sort: return the values in order

- example: return search results by item's popularity

Rank search: return the k-th largest item

- example: return the next building to be completed in a strategy game

Range search: return values between h and k

- example: return all restaurants within 100 m of user

Sorted Dictionary Runtimes

Implementation	Search	Insert
Arrays	$O(?)$	$O(?)$
Linked Lists	$O(?)$	$O(?)$

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:

```
int ibinsearch(int *a, int n, int key)
```

`a[]` assumed sorted
`n` is array size
return index of `key`

What is the time complexity of the algorithm?

Iterative Binary Search: Analysis

One comparison for every halving of search interval

Continue halving until there's only 1 element left:

$$((\ldots(((n/2)/2)/2)\ldots)/2) = n/2^k$$

It takes k halvings to get to 1 element:

$$n/2^k = 1;\ n = 2^k;\ \log n = \log 2^k;\ k = \log n$$

Time complexity: $O(\log n)$

Accounting Rule 5

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 $O(N)$

Compute $n!$ Iterative Version

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

```
int ifact(int n)
```

What is its time complexity?

Recursion

An alternative to iteration, which uses loops

Recursion is an extremely powerful problem-solving technique

- breaks large problem instance into smaller instances of the identical problem
- a “natural way” (but not the only way!) to think about and implement a divide and conquer strategy

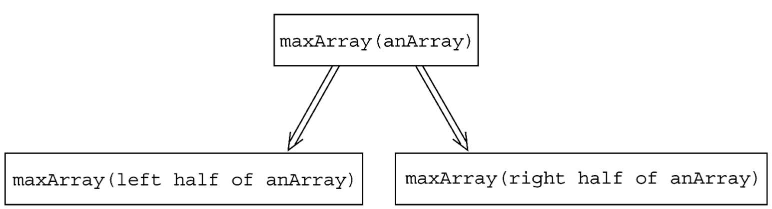
Recursive Function

What are the characteristics of a recursive function?

- a function that calls itself
- each time with a smaller instance of the problem
- must have a termination condition
  - the solution to at least one smaller problem instance, the base case, is known
  - eventually, one of the smaller problem instances must be the base case; reaching the base case enables the recursive calls to stop

Recursively Searching an Array: Finding the Largest Item in an Array

```
if (anArray has only one item) // base case
  maxArray(anArray) is the item in anArray
else if (anArray has more than one item
  maxArray(anArray) is the maximum of
       maxArray(left half of anArray) and
       maxArray(right half of anArray)
```



Recursive Solutions

Four considerations in constructing recursive solutions:

1. How can you define the problem in terms of a smaller problem of the same type?
2. How does each recursive call reduce the size of the problem instance?
3. What instance of the problem can serve as the base case?
4. As the problem size diminishes, will you reach this base case?

Compute $n!$ Recursive Version

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

```
int rfact(int n)
```

How to compute its time complexity?

Let $T(n)$ be the operation-count complexity of `rfact(n)`

Which operation shall we count?

Complexity of Recursive $n!$

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

What is the time complexity of `rfact(n)`?

- any constant operation count can be replaced by '1'

What is the space complexity of `rfact(n)`?

Accounting Rule 6

Definition: a recurrence relation is a mathematical formula that generates the terms in a sequence from previous terms

Examples:

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

Accounting Rule 6:

Recurrence relations are "natural" descriptions of the timing complexity of recursive algorithms

Recursive Binary Search: Code

```
int /* index of key */
rbinsearch(int *a, int f, int n, int key)
/* a[] sorted, f = 1st elt in a, n = sizeof(a) */
{
 int mid;

 if (!n) return NOTFOUND;
 if (n == 1) return (a[f] == key ? f : NOTFOUND);
 mid = f+n/2;
 if (key < a[mid]) return(rbinsearch(a, f, n/2, key));
 else return(rbinsearch(a, mid, n-n/2, key));
}
```

What is the time complexity of the algorithm?

- recall: any constant per-level cost can be represented as '1'

Recursive Binary Search: Analysis

Recurrence relation: $T(n) = 1 + T(n/2), T(1) = 1$

$T(n) = 1 + T(n/2)$

When do we stop the recurrence?

What is k in the end?

Time complexity: $O(?)$