CS161 - Hashtables

David Kauchak

- B-Tree demo - http://people.ksp.sk/~kuko/bak/index.html
- What data structures have we seen so far:
  - arrays - get and set particular indices in constant time
  - linked list - insert and delete in constant time
  - stack - LIFO
  - queue - FIFO
  - heap - max/min in logarithmic time
  - BST - search in logarithmic time
  - B-Trees - search on disk in logarithmic disk accesses
- Hashtables: constant time insertion and search (and deletion in some cases), i.e. $\textsc{Search}(S, x)$, $\textsc{Insert}(S, x)$ and $\textsc{Delete}(S, x)$ are $\Theta(1)$
- Applications
  - Is $x \in S$?
  - I use them all the time
  - compilers
  - databases
  - search engines?
  - any time you'd like to store and retrieve non-squential data
  - sparse data: save memory over an array
- Key/Data pair - For anything we're trying to store in a hashtable we need a key that is a number. This raises two related issues:

1. Are we using the entire data as the key or just some representative data? For example, a bank may have full information about a client (name, address, phone number, birth data, social), but only use one piece of data to index the user, like the name or social security number.
2. The key into a hash table needs to be a number. If the data we're storing is numerical, then we're fine. Other times, we need to do a transformation, e.g. strings.

- Why do we need hashtables? Why not just use arrays?

  Picture of space of universe of keys and array

  It is very common that the possible set of values, denoted U, is very large. This can be challenging because

  1. If we want to directly index the items, the size of the array may be prohibitively large, e.g. strings.
  2. Even if we could generate an array large enought, the data may be very sparse, so directly indexing the items would be very wasteful.

  Let n be the number of keys and m be the size of the array, we defined the load factor as $\alpha = n/m$

  If α is small, then the amount of unused, wasted space is large

  Similarly, if the range of possible keys is large, then m must be large

- Hash function - A hash function is a function that maps the universe of keys U to the slots in the hash table, i.e.

  $h : U \to 0, 1, ..., m - 1$

- Collisions

  A collision occurs when $h(x) = h(y)$, but $x \neq y$

  We can try to minimize this by having a "good" hash function, but because $|U| > m$ collisions are inevitable

- Collision resolution by chaining
  - The hashtable consists of an array of linked lists
  - If a collision occurs, add the element into the linked list at that location. Specifically, if two elements x and y are inserted where $h(x) = h(y)$, then the hashtable entry $T(h(x))$ contains a linked list with both x and y.
  - picture

    ChainedHashInsert(T, x)
    insert x at the head of list $T[h(x)]$

    ChainedHashSearch(T, x)
    search for x in list $T[h(x)]$

    ChainedHashDelete(T, x)
    delete x from the list $T[h(key[x])]$

  - Runtime?
    - Insertion - $\Theta(1)$
    - Deletion - $\Theta(1)$
    - Search - ?, depends on how many elements are in the linked list
      - Worst case scenario? All items inserted into the table hash to the same position, e.g. $h(k) = 1$. - $\Theta(n)$
      - Average case
        We haven't yet talked about hash functions yet, but we will assume the best case scenario, that is the hash function h uniformly distributes the set of keys over the range m. This is called *simple uniform hashing*.

        Under simple uniform hashing, what is the average length of a linked list entry in the hashtable? n entries over m slots, $n/m = \alpha$

        Another way to think of this is to consider n tosses of a fair, m sided die. On average, how many times would we see a particular value. For example, say we roll a 10

sided die 100 times, how many times would we expect to see each value? $100/10 = \mathbf{10}$

There are two cases:

1. The element is **not** in the - In this case, we must search the chain of items at $T(h(x))$, which is on average α items long. $O(1 + \alpha)$. *Why do we includ the 1?*

2. The item **is** in the table - On average, we'll find the entry by searching half of the chain items at $T(h(x))$. $O(1 + \alpha/2)$

In both cases, the average case running time for search is still $O(1 + \alpha)$, which is constant.

- Hash functions
  - what makes a good hash function?
    - Approximates the assumption of simple uniform hashing, i.e
    - Deterministic - $h(x)$ should always return the same value.
    - Low cost - All of the constant time assertions for hashtable assume that the hash function is already calculated. If the hash function is computational expensive (i.e. more expensive than $\log n$), then something like BST would be better.
  - The challenge: we generally don't know the distribution of keys
  - One of the most common problems for a hash function is that data items often tend to be clustered, e.g. similar strings, numbers, SSNs, runtimes. A good hash function should be able to spread data items near across the hashtable.
    - Division method - $h(k) = k \bmod m$

m	k	$h(k)$
11	25	3
11	1	1
11	17	6
13	7	7
13	133	3
13	25	12

Don't use a power of 2. If $m = 2^p$ then $h(k)$ is just the p lowest-order bits of k

m	k	bin(k)	$h(k)$
8	25	11001	1
8	1	00001	1
8	17	10001	1

A common rule of thumb is to use a prime number not too close to a power of 2.

* Multiplication method
  Multiply the key by a constant $0 < A < 1$ and extract the fractional part of kA, then scale this by m to get the index.

  $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ (Note, $(kA - \lfloor kA \rfloor)$ simply extracts the fractional portion of kA).

  Common choice is m as a power of 2 and $A = (\sqrt{(}5) - 1)/2 = 0.6180339887...$

  See the book for some heuristics

m	k	A	Ak	$h(k)$
8	15	0.618	9.27	$\lfloor 0.27 * 8 \rfloor = 2$
8	23	0.618	14.214	$\lfloor 0.214 * 8 \rfloor = 1$
8	35	0.618	21.63	$\lfloor 0.63 * 8 \rfloor = 5$
8	100	0.618	61.8	$\lfloor 0.8 * 8 \rfloor = 6$

– Other hash functions
  These are just a very basic look at hash functions. Many more exist for a variety of purposes (e.g. http://en.wikipedia.org/wiki/List_of_hash_functions)
  * Others...
  * Cyclic redudancy checks (i.e. disks, cds, dvds, etc)
  * Checksums (i.e. networking, file transfers)
  * Cryptographic (i.e. MD5, SHA, etc)

- Open addressing
  – Hashtable is only the array of elements (no pointers)
  – When a collision occurs, instead of creating a chain, we compute another slot in the hashtable to examine.

- A hash function doesn't just provide one value, but defines a probe sequence, i.e. the order in which we should try different locations in the table for inserting and searching for a key
- The hash function takes an additional parameter i, which is the number of collisions that have occured and defines the probe sequence. This sequence **must** be a permutation of every hashtable position, i.e.

$$\{h(k,0), h(k,1), h(k,2), ..., h(k,m-1)\}$$

  is a permutation of $\{0, 1, 2, ..., m-1\}$
- Insertion - Follow the probe sequence (by incrementing i in $h(k,i)$) until we find an open slot

```
HASH-INSERT(T, k)
1  i ← 0
2  j ← h(k, i)
3  while i < m − 1 and T[j] ≠ null
4           i ← i + 1
5           j ← h(k, i)
6  if T[j] = null
7           return j
8  else
9           error "hash is full"
```

- Searching - Follow the probe sequence until we either find the key we're looking for or we find an open slot and the key is, therefore, not present

```
HASH-SEARCH(T, k)
1  i ← 0
2  j ← h(k, i)
3  while i < m − 1 and T[j] ≠ null and T[j] ≠ k
4           i ← i + 1
5           j ← h(k, i)
6  if T[j] = k
7           return j
8  else
9           return null
```

– Deletion - If we simply delete a node, then we may "break" the probe sequence. 2 options: ***** Example *****

1. mark node as "deleted" instead of *null* and keep searching if we see a "deleted" node. Increases search times.
2. use chaining. If there will be a lot of deletions, this is generally the best decision.

– Probing schemes - ***** Examples of problems?? *****

∗ Linear probing - If a collision occurs, simply go to the next slot

$$h(k, i) = (h(k) + i) \bmod m$$

for example, $m = 7$ and $h(k) = 4$, then

$h(k, 0) = 4$
$h(k, 1) = 5$
$h(k, 2) = 6$
$h(k, 3) = 0$
$h(k, 4) = 1$
...

Problem: primary clustering - Long runs of occupied slots tend to build up and these tend to grow in value, since probability of any collusion in this cluster increases.

∗ Quadratic probing

$$h(k, i) = (h(k) = c_1 i + c_2 i^2) \bmod m$$

Problem: if $h(x) = h(y)$, then $h(x, i) = h(y, i)$ for all values of i

∗ Double hashing - The probe sequence is determined by a second hash function

$$h(k, i) = (h_1(k) + i(h_2(k)) \bmod m$$

Challenge: h_2 must visit all possible positions in the hashtable

– runtime of Hash-Search and Hash-Insert

We'll assume an ideal hash function where any probe sequence is equally likely

Let X be the number of probes made (either by a call to insert or an unsuccessful call to search). We would like to calcuate on average, how many probes are required before an empty entry is found, i.e. $E[X]$.

$$E[X] = \sum_{i=1}^{m} p(X = i)$$

Let A_i be the event that the ith probe is occupied, then

$$p(X = i) \leq p(X \geq i) = A_1 \cap A_2 \cap A_3 \cap ... \cap A_{i-1}$$

i.e. p(X=i) is bounded by the probability that the probes 1 through $i - 1$ were occupied

$$\begin{aligned} p(X \geq i) &= A_1 \cap A_2 \cap A_3 \cap ... \cap A_{i-1} \\ &= p(A_1)p(A_2|A_1)p(A_3|A_1, A_2)...p(A_i|A_1, ..., A_i - 1) \\ &= \frac{n}{m}\frac{n-1}{m-1}\frac{n-2}{m-2}...\frac{n-i+2}{m-i+2} \\ &\leq (\frac{n}{m}^{i-1}) \\ &= \alpha^{i-1} \end{aligned}$$

and

$$\begin{aligned} E[X] &= \sum_{i=1}^{m} p(X = i) \\ &\leq \sum_{i=1}^{m} p(X \geq i) \\ &\leq \sum_{i=1}^{m} \alpha^{i-1} \\ &\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\ &= \sum_{i=0}^{\infty} \alpha^{i} \\ &= \frac{1}{1-\alpha} \end{aligned}$$

Intuitively, this makes sense. Consider the series above:

$$1 + \alpha + \alpha^2 + \alpha^3 + ...$$

we have to make the first probe, with probability α we have to make a second probe, with probability α^2 we have to make the third, etc.

α	Average number of searches
0.1	$1/(1 - .1) = 1.11$
0.25	$1/(1 - .25) = 1.33$
0.5	$1/(1 - .5) = 2$
0.75	$1/(1 - .75) = 4$
0.9	$1/(1 - .9) = 10$
0.95	$1/(1 - .95) = 20$
0.99	$1/(1 - .99) = 100$

As before, if the search succeeds, we can expect on average to search about half as many entries as when it fails.

- Selecting hashtable size with open addressing

  Ideally, a good rule of thumb is to shoot for $\alpha \leq .5$, i.e no more than about half of the hashtable full.

  At some point, with open addressing, the hashtable can fill up. What happens then?

  * Create a new table and copy the new values over. Depending on the hash function and what data we keep, we may have to recalculate the hash function for every entry. This can result in one insertion being very expensive when the copy occurs.
  * Amortized cost - Some approaches slowly grow the size of the hashtable, by growing the hashtable when a certain size ratio is hit and then slowly copying the values over as new ones are inserted. The logic is more complicated, but can assure a relatively low computational cost on *every* operation.

These notes are adapted from material found in chapter 11 of [1].

References

[1] Thomas H. Cormen, Charles E. Leiserson Ronald L. Rivest and Clifford Stein. 2007. Introduction to Algorithms, 2nd ed. MIT Press.