# Hashing

Abhiram Ranade

March 7, 2019

# Concluding remarks on balanced search trees

Balanced search tree: Data structure to represent sets.
Operations allowed on a set S:

- S.insert(x) : Insert x into S.
- S.find(x) : Determine if x is present in S.
- S.erase(x) : Remove x from set.

All operations can be performed in time $O(\log n)$ where $n$ is the number of elements currently in the set.

$O(\log n)$ time support also for `S.lower_bound(v)` operation which returns iterator to element equal or larger than `v`.

Similarly upper bound operation.

Iterators can be used for many operations including scanning through elements in amortized $O(1)$ time per element.

Available in C++ STL class `set`

# Maps

Map = function, set of pairs (x,y) "(key,value)"

C++ class map

Operations allowed on a map M:

- M.insert(x,y) : insert the pair (x,y) into M.

  Syntax: M[x] = y;
- M.find(x) : decide whether a pair (x,y) exists in M and if yes return y.

  Syntax: (M.count(x) > 0) ? M[x] : ...
- M.erase(x) : delete pair (x,y) if it exists.

Other operations such as upper_bound, lower_bound also available.

Can scan through elements in increasing order of x using iterators.

Implemented using balanced search trees.

- Each node stores both x,y. Nodes are ordered by x.
- Insert, find, erase, upper/lower bound: $O(\log n)$ time.
- Scanning through elements in $O(1)$ amortized time.

# Hashing: a faster way to represent sets and maps

- Insert, find, delete take time "$O(1)$".
  Typically the time will be constant. Explained later.
- Lower and upper bound operations not supported.
  Not needed in many applications.
- Available in C++ classes unordered_set and unordered_map.
  key/set elements need not satisfy any ordering
  Ordering relationship is not used in implementation.

A map (x,y) which supports insert(x,y), find(x), erase(x) is called a dictionary

Balanced search trees implement a dictionary such that each operation takes $O(\log n)$ time.

Hashing can be used to implement a dictionary such that each operation takes $O(1)$ time typically.

# Warmup: Representing sets over a small universe

Suppose we wish to represent sets which are subsets of $\{0, \ldots, 99\}$.

Implementation:

- For each set S maintain an array s[0..99].
- Invariant: s[i]==1 iff $i \in S$.
- S.insert(i) : s[i] = 1.
- S.find[i] : s[i] == 1
- S.erase[i] : s[i] = 0

"Direct address table"

Representing maps (x,y) where x $\in \{0, .., 99\}$

- Use an additional data array to store y part.
- M.insert(x,y) : s[x] = 1; data[x] = y;

What do we do if the key, x, is not from a small universe?
Example: 10 letter names: Table size $= 26^{10} > 2^{45}$
Impractical!

# Hashing: Approximating a direct address table

Suppose keys come from a universe $U = \{0, 1, \ldots, |U| - 1\}$
Want to store $X \subset U$ with $|X| = n$.
Goal: Use $O(n)$ storage, and not $O(|U|)$.

Idea:

- Use a table $T$ of size $m = kn$ for some small $k$.
- Select $h : U \to \{0, 1, \ldots, m - 1\}$. e.g. $h(x) = x \bmod m$
- "If $x \in X$, T[h(x)] = 1". Almost...
- What if $h(x) = h(y)$ for two keys $x \neq y$?
  $T[q]$ = vector of keys $x$ s.t. $h(x) = q$.
- Find(x) : Check if vector $T[h(x)]$ contains $x$.
- Delete(x) : Remove $x$ from vector $T[h(x)]$

10 letter names: U = {10 digit number in radix 26}
$x = x_0 + R(x_1 + R(x_2 + R(\ldots)))$, where $R = 26$
To evaluate $h(x)$ perform each addition/mult mod $m$

Total storage used (table + names) $\approx kn + n \log |U|$.

# Performance of hashing - "Typical case"

- Keys come from universe $U = \{0, \ldots, |U| - 1\}$
- Assume: we know how many keys $n$ will be in table.
- Table has size $m = kn$. Say $k = 2$.
- Key mapping function $h(x) = x \bmod m$
- Some set $X$ of $n$ keys stored in table.

Assumption: We are storing $n$ randomly drawn keys into $X$

- Keys will distribute nicely among table entries.
- E[number of keys in T[i]] $= O(n/m) = O(1)$
- Expected insertion time : $O(n/m) = O(1)$
- Expected find/delete time : $O(n/m) = O(1)$

$k = \frac{n}{m}$ : "load factor" of table.

What if we don't know $n$? We resize, like vectors: Exercise

# Performance of hashing - Worst case

- Keys come from universe $U = \{0, \ldots, |U| - 1\}$
- Table has size $m$.
- Key mapping function $h(x) = x \bmod m$
- Some set $X$ of $n$ keys stored in table.

Worst case:

All $x \in X$ map to same table slot, i.e. $h(x) = \alpha$ for all $x \in X$.

Keys $x, y$ have $h(x) = h(y)$: "collision".

Each insertion will insert into the same list.

We should first check if the set already contains the key

$\Rightarrow$ $n$ Insertions will take time $O(n^2)$

If we wish to do find($y$) where $h(y) = \alpha$, time $= O(n)$.

"Worst case is not likely."

Informal assertion; no probability space..

# Other uses of hash functions

## Minimizing transmission cost:

- Suppose you have file f1, your friend has file f2.
- You wish to know if $f1 = f2$.
- Can this be checked without transmitting entire files, if small probability of error is allowed?

## Solution using agreed upon hash function *h*

- Your friend computes h(f2) and sends to you.
- You check if received h(f2)=h(f1).
- If files are different received value will be different with large probability. Repeat for several hash functions.

  Even single character difference will be detected.
- Transmission cost is small because hash value is small.

# Remarks

- Key requirement: Function $h$ should distribute keys of interest over the table slots. "$h$ should appear to distribute randomly"

  hash = random mess, so hash function $h$, hash table $T$
- Reasonably simple choices work for $h$. $h$ mod $m$ will not work if keys are likely to be separated by $\alpha m$.

  Choose $m$ = random prime!

  $m = 2^k$, $h(x) = x \cdot A$ mod $m$, where A is odd.

  Something like this works for arbitrary sets w.h.p.
- Chaining: Handling collisions by keeping a list.
- Linear probing: If collision at $h(x)$ check if $h(x) + 1$ is empty, if not then $h(x) + 2$ and so on until an empty slot is found.
- Many other ways of resolving collisions also studied.
- Hash functions are typically much better than balanced trees if you only want insert/find/delete, and not lower/upper bound operations.
- C++ classes unordered_set and unordered_map provide iterators which can be used to step through all elements; but no order is guaranteed.

# Remarks continued: Universal Hash functions

Hash functions as defined are very useful, but we cannot say anything rigorous about them.

We cannot justify the assumption that keys to be inserted are drawn at random.

A different idea: Allow arbitrary keys to be inserted, but choose the hash function at random!

- The idea works!
- Over the space of all possible hash functions, we can show that the insertion and find operations will require $O(1)$ expected time.

Idea described next. Without proof.

Only for information (see CLRS), not for the exam.

# Universal class of hash functions

Universe of keys: $U = \{0, \ldots, n-1\}$
Set of table slots: $T = \{0, \ldots, m-1\}$
"hash function" : map from $U$ to $T$

Universal set of hash functions: A set $H$ of hash functions is universal if $\Pr[h(x) = h(y)] \leq \frac{1}{m}$ when $h$ is chosen from $H$ at random, for all $x, y, x \neq y$.

$h(x) = h(y)$: "$x, y$ collide"

Remarks:

- Probability of collision if $x, y$ were stored randomly $= \frac{1}{m}$.
  Choosing hash function at random does the same, for all $x, y$.
- Claim made only about pairs of keys, so "Universal$_2$".
- Example of universal set soon.

Theorem: Let a hash function $h$ be selected at random from a universal set of hash functions. Then the expected time to process a sequence of $r$ requests containing $k$ insertions is $O(r(1 + \frac{k}{m}))$.
So choose table size $m =$ O(set size), i.e. load factor $= O(1)$.

# A universal set of hash functions

$p$ : prime number $> |U|$.

Chosen offline

$a$ : integer from $\{1, \ldots, p-1\} = Z_p^*$

$b$ : integer from $\{0, \ldots, p-1\} = Z_p$

Auxiliary functions $g_{a,b}$ : $g_{a,b}(x) = ax + b \bmod p$

Hash functions: $h_{a,b}$ : $h_{a,b}(x) = g_{a,b}(x) \bmod m$

Theorem: $H = \{h_{a,b} \mid a \in Z_p^*,\ b \in Z_p\}$ is universal.