Group assignment 3

Advanced Algorithms and Data Structures

Authors: psl788, wlc376, knx373

Hand-in deadline: December 13, 2022

Hashing exercises

2.1

The probability of sampling, with replacement, two of the same element from an independent, uniform distribution with $|\Omega|$ possible outcomes is $1/|\Omega|$.

In our case, if h is a truly independent hash function from U to $[m]$ with $\big|[m]\big| = m$, then we can view h as an independent distribution, and since keys hash independently, we can consider random variables from that distribution as being i.i.d..

Since we have $\big|[m]\big| = m$, we have that:

$$\Pr_h\big[h(x) = h(y)\big] = \frac{1}{|[m]|} = \frac{1}{m}.$$

Yes, the truly independent hash function $h : U \to [m]$ is universal.

2.2

For a hash function $h : U \to [m]$ to have collision probability 0, we must have $m \geq |U|$.

Consider hashing $|U|$ keys with $m \geq |U|$. Placing each hash in a distinct bucket in $[m]$, then we will have $|U|$ buckets with exactly 1 item in them and $m - |U|$ empty buckets.

On the other hand, consider hashing $|U|$ keys into $m < |U|$. First, we hash m items into distinct buckets in $[m]$. At this point, we have m buckets with exactly 1 item in them and $m - m = 0$ empty buckets, but we still have $|U| - m > 0$ keys left to hash. By the pigeonhole theorem, we will have at least 1 collision.

2.3

Yes, the identity function f is a universal hash function as long as $u \leq m$, since for distinct x and y we have:

$$\Pr_h[f(x) = f(y)] \;=\; \Pr_h[x = y] \;=\; 0 \;\leq\; \frac{1}{m}.$$

2.4

Expected number of elements in $L[h(x)]$ given $x \in S$.

The first three steps of the derivation follow those in section 2.1 of the hashing notes:

$$\begin{aligned}
\mathbb{E}_h\Big[\,|L[h(x)]|\,\Big] &= \mathbb{E}_h\left[\sum_{y\in S}\mathbb{I}(h(y)=h(x))\right] \\
&= \sum_{y\in S}\mathbb{E}_h\Big[\mathbb{I}(h(y)=h(x))\Big] \\
&= \sum_{y\in S}\Pr_h\left[h(y)=h(x)\right] \qquad (1)
\end{aligned}$$

Since we know x to be in S, we can pull x out of the summation:

$$\begin{aligned}
\sum_{y\in S}\Pr_h\left[h(y)=h(x)\right] &= \Pr_h\left[h(x)=h(x)\right] + \sum_{y\in S\setminus\{x\}}\Pr_h\left[h(y)=h(x)\right] \\
&= 1 + \Big|S\setminus\{x\}\Big|\cdot\frac{1}{m} \\
&= 1 + \frac{n-1}{m} \\
&< 1 + \frac{m}{m} \qquad (2) \\
&= 2.
\end{aligned}$$

Equation (2) follows from $(n-1) < n \leq m$.

Assuming h is 2-approximately universal

We start the derivation from Equation (1):

$$\begin{aligned}
\sum_{y \in S} \Pr_h [h(y) = h(x)] &= \sum_{y \in S} \frac{2}{m} && (3) \\
&= n \frac{2}{m} \\
&\leq m \frac{2}{m} && (4) \\
&= 2.
\end{aligned}$$

Equation (3) uses 2-approximate universality of h, and Equation (4) uses $n \leq m$.

2.5

Probability of false positive for a given $x \in U \setminus S$ for universal hash functions $s : U \to [n^3]$ and $h : U \to [n]$:

$$
\begin{aligned}
\Pr_{h,s}[\exists y \in S : h(y) = h(x) \wedge s(y) = s(x)] &\leq \sum_{y \in S} \Pr_{h,s}[h(y) = h(x) \wedge s(y) = s(x)] && (5)\\
&= \sum_{y \in S} \Pr_{h}[h(y) = h(x)] \Pr_{s}[s(y) = s(x)] && (6)\\
&= \sum_{y \in S} \frac{1}{n}\frac{1}{n^3}\\
&= |S|\frac{1}{n}\frac{1}{n^3}\\
&= \frac{|S|}{n^4}.
\end{aligned}
$$

Equation (5) follows from the union bound, and Equation (6) uses independence of h and s.

2.6

This task will look at the hashing function defined by

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod m,$$

where $(a, b) \in [p]^2$.

(a) h may not be universal

If we choose $a = 0$ the hashing function no longer depends on the value of x and will give the the same answer for all $x \in [p]$:

$$h_{a=0,\ b} \;=\; (b \bmod p) \bmod m.$$

The collision probability is therefore 1, and the hashing function is therefore not universal.

(b) h is 2-approximately universal

In the following, we assume $1 < m < u \leq p$, as is assumed in

For uniformly random $(a, b) \in [p]^2$ we have

$$\begin{aligned}
\Pr_{(a,\,b)\,\in\,[p]^2}[h_{a,b}(x) = h_{a,b}(y)] \;&=\; \Pr[a \neq 0] \cdot \Pr_{a\,\in\,[p]_+,\,b\,\in\,[p]}[h_{a,b}(x) = h_{a,b}(y)] \\
&\quad\;+\; \Pr[a = 0] \cdot \Pr_{b\,\in\,[p]}[h_{0,b}(x) = h_{0,b}(y)] \\
&\leq \frac{p-1}{p}\frac{1}{m} + \frac{1}{p} \cdot 1 \qquad (7) \\
&= \frac{p+m-1}{mp} \\
&< \frac{2p}{mp}, \qquad (8) \\
&= \frac{2}{m}.
\end{aligned}$$

Equation (7) follows from universality of $h_{a,b}$ when $a \in [p]_+$, and from $\Pr_{b\,\in\,[p]}[h_{0,b}(x) = h_{0,b}(y)] = 1$, while Equation (8) follows from $m < u \leq p$.

3.1

Hash function $h : [u] \mapsto [m]$ is 3-independent if the probability of every three-wise event is $\frac{1}{m^3}$. Following observation 3.1 from Thorup's notes, it can be expressed in an equivalent manner: 3-independence means that each key is hashed uniformly into $[m]$, and that every three distinct keys are hashed independently.

Thus, k-independence is a symmetrical abstraction obtained by simply exchanging the threes in the above explanation of 3-independence with k's.

We define k-independence mathematically as such:

$$\Pr\left[\bigwedge_{i=1}^{k} h(x_i) = y_i\right] = \frac{1}{m^k}.$$

3.2

The definition of a strongly universal hashing function, h, says that every pair of distinct keys hash independently and for every key $x \in U$ and hash value $q \in [m]$, we have $\Pr[h(x) = q] \leq c/m$. Following the notation of the notes by Mikkel Thorup, $x, y \in [u]$ is a given pair of distinct keys. Using the definition of a strongly universal hashing function, the upper bound for the pairwise event probability is

$$\begin{aligned} \Pr[h(x) = q \wedge h(y) = r] &= \Pr[h(x) = q] \Pr[h(y) = r] \\ &\leq \frac{c^2}{m^2}. \end{aligned}$$

3.3

Since the random hash function $h : U \mapsto [m]$ is c-approximately strongly universal we know that for every key $x \in [u]$ and every hash value $q \in [m]$ the probability of $h(x) = q$ is:

$$\Pr\left[h(x) = q\right] \leq \frac{c}{m}.$$

Strong universality also requires that every pair of distinct keys, $x, y \in [u]$, hash independently. Then if h is c-approximately strongly universal, it is also c-approximately universal since:

$$\begin{aligned}
\Pr[h(x) = h(y)] &= \sum_{q\in[m]} \Pr[h(x) = q \wedge h(y) = q] \\
&= \sum_{q\in[m]} \Pr[h(x) = q] \cdot \Pr[h(y) = q] \\
&\leq \sum_{q\in[m]} \Pr[h(x) = q] \cdot \frac{c}{m} \\
&= \frac{c}{m} \cdot \sum_{q\in[m]} \Pr[h(x) = q] \\
&= \frac{c}{m} \cdot 1 = \frac{c}{m}
\end{aligned}$$

The second equality is a consequence of independence, while the inequality is due to the collision probability since $h(x) = q$. Lastly, since $h(x) = q$ we have that $\sum_{q\in[m]} \Pr[h(x) = q] = 1$. Thus, $\Pr[h(x) = h(y)] \leq \frac{c}{m}$ and h is also c-approximately universal.

3.4

We want to show that not all pairs of keys hash independently. To see this, we will consider all pairs $(x, x + k \cdot 2^w)$ for any $k \in \mathbb{Z}$.

But first, recall the following three basic properties of modular arithmetic:

$$mn \bmod n = 0. \tag{9}$$

$$(m \bmod n) \bmod n = m \bmod n. \tag{10}$$

$$(a + b) \bmod n = \Big[(a \bmod n) + (b \bmod n)\Big] \bmod n. \tag{11}$$

Where m, n, a, b are all integers[1].

Using these rules, we will show that $h_a(x) = h_a(x + k2^w)$ for all x and k:

$$\begin{aligned}
h_a(x) &= \Big\lfloor (ax \bmod 2^w)/2^{w-\ell} \Big\rfloor \\
&= \Big\lfloor ([ax \bmod 2^w] \bmod 2^w)/2^{w-\ell} \Big\rfloor && (12) \\
&= \Big\lfloor \big(\big[(ax \bmod 2^w) + \underbrace{(ak2^w \bmod 2^w)}_{= 0 \text{ by eq. (9)}}\big] \bmod 2^w\big)/2^{w-\ell} \Big\rfloor && (13) \\
&= \Big\lfloor ((ax + ak2^w) \bmod 2^w)/2^{w-\ell} \Big\rfloor && (14) \\
&= \Big\lfloor (a(x + k2^w) \bmod 2^w)/2^{w-\ell} \Big\rfloor \\
&= h_a(x + k2^w).
\end{aligned}$$

Equation (12) follows from eq. (10) (identity of mod); Equation (13) follows from eq. (9) (and, of course, the fact that is it always legal to add 0); Equation (14) follows from eq. (11) (distributivity), and the final two derivations use simply distributivity of multiplication and the definition of $h_a(x)$.

Hence there is a dependency between all pairs $(x, x + k2^w)$ for any integers x and a, and non-negative integers w and ℓ with $w \geq \ell$.

[1] Equation 9 follows directly from the definition of the mod operator, while equations 10 and 11 are identity and distributivity of the mod operator, respectively.

3.5

Given $S_{h,t}(B)$ and $S_{h,t}(C)$, the size of the symmetric difference $(B \setminus C) \cup (C \setminus B)$ can be calculated by re-expressing the symmetric difference in terms of $S_{h,\,t}(B)$ and $S_{h,\,t}(C)$ as such:

$$
\begin{aligned}
\mathbb{E}\left[\left|(B \setminus C) \cup (C \setminus B)\right|\right] &= \mathbb{E}\left[\left|(B \cup C) \setminus (B \cap C)\right|\right] && (15)\\
&= \mathbb{E}\left[|B| + |C| - 2|B \cap C|\right] && (16)\\
&= \mathbb{E}\left[|B|\right] + \mathbb{E}\left[|C|\right] - 2\mathbb{E}\left[|B \cap C|\right] && (17)\\
&= \tfrac{m}{t}\left|S_{h,\,t}(B)\right| + \tfrac{m}{t}\left|S_{h,\,t}(C)\right| - 2\tfrac{m}{t}\left|S_{h,\,t}(B \cap C)\right| && (18)\\
&= \tfrac{m}{t}\Big(\left|S_{h,\,t}(B)\right| + \left|S_{h,\,t}(C)\right| - 2\left|S_{h,\,t}(B) \cap S_{h,\,t}(C)\right|\Big) && (19)
\end{aligned}
$$

where eq. (15) uses that the symmetric difference between B and C is equivalent to the difference between their union and their intersection; eq. (16) expresses the cardinality of this difference as the sum of cardinalities of B and C minus twice the cardinality of their intersection; eq. (17) uses linearity of expectation; eq. (18) uses $\mathbb{E}\left[|A|\right] = \frac{m}{t}\left|S_{h,\,t}(A)\right|$ (as per Thorup's notes, section 3.1); and eq. (19) uses $S_{h,\,t}(X \cap Y) = S_{h,\,t}(X) \cap S_{h,\,t}(Y)$ (ibid.) and distributivity of multiplication.

3.6

We are going to use lemma 3.2 of Thorup's notes to compute the bound, which states that if X is a sum of pairwise independent 0-1 indicator variables $X_a = [h(a) < t]$ with expected mean μ, then:

$$\Pr\left[|X - \mu| \;\geq\; q\sqrt{\mu}\right] \quad \leq \quad \frac{1}{q^2}.$$

In this particular case, we have $\mu = 10^6$, and hence:

$$\begin{aligned} q\sqrt{\mu} \quad &= q\sqrt{10^6} = 10,000 \\ \Leftrightarrow \qquad\qquad q &= 10. \end{aligned}$$

By lemma 3.2, our bound is then:

$$\begin{aligned} \Pr\left[|X - \mu| \;\geq\; 10,000\right] \quad &\leq \quad \frac{1}{10^2} \\ &= \quad \frac{1}{100} = \frac{t}{m} = p. \end{aligned}$$

vEB-tree exercises

CLRS 20.3-1

Membership testing in $O(1)$ time and $\theta(u)$ space

As stated in the hint for exercise 20.3-4, we can modify the vEB structure to include a size u array of bits, where the i'th entry is 1 if key i is a member of V; else it is 0. This membership array uses $\theta(u)$ space Since this membership array uses $\theta(u)$ space, it does not affect asymptotic space usage of vEB, and, in addition, the array can be used to test membership in $O(1)$ time.

Let vEB-Tree-Members(V) be a function which, given a top-level tree V, returns a reference to such a membership array for V. We can then test membership of x by inspecting the element vEB-Tree-Members(V)[x]. The array is zero-initialized upon creation of an empty tree, and the x'th entry is modified upon insertion/deletion of key x.

Supporting duplicate keys

Let A = vEB-Tree-Members(V).

Instead of setting $A[x] = 1$ upon insertion, each time a given key x is inserted into the tree, the corresponding entry in A is incremented by one. The actual operation vEB-Tree-Insert(V, x) is only called if the corresponding index, $A[x]$ is zero, since otherwise the element is already in the tree. The course of action is similar when deleting an element, except that the actual call to vEB-Tree-Delete(V, x) is only made when we are deleting the last occurrence of x.

Below snippet shows pseudocode for the modified operations. Neither operation assumes anything about the number of occurrences of x presently in V.

```
vEB-Tree-Insert'(V, x):
  if vEB-Tree-Members(v)[x] == 0:
    vEB-Tree-Insert(V, x)
  vEB-Tree-Members(v)[x] += 1

vEB-Tree-Delete'(V, x):
  if vEB-Tree-Members(V)[x] <= 0:
    return
  elseif vEB-Tree-Members(v)[x] == 1:
    vEB-Tree-Delete(V, x)
  vEB-Tree-Members(v)[x] -= 1
```

CLRS 20.3-2

In our solution 20.3-1, we used an auxiliary array A to keep count of the number of occurrences of each key in V.

Our solution to supporting satellite data is similar and also uses vEB-Tree-Members(V), but in this case the entries of A are either 0 or 1 since we no longer support duplicate keys.

Assuming any single piece of satellite data uses $O(1)$ space, we can store satellite data in a second, auxiliary size $V.u$ array for an asymptotic space contribution of $\theta(u)$.

This auxiliary satellite data array is accessed using the function vEB-Tree-Satellites(V), which, given a vEB tree V, returns a reference to an array of references to satellite data, where the i'th element is a reference to the satellite data for key i if i is in V, and else NIL.

Below snippet shows pseudocode for extracting satellite data, as well as the modified insert and delete operations:

```
vEB-Get-Satellite(V, x):
  return vEB-Tree-Satellites(V)[x] // returns NIL if x is not in V.

vEB-Tree-Insert'(V, x, data):
  if vEB-Tree-Members(V)[x] == 0:
    vEB-Tree-Insert(V, x)
    vEB-Tree-Members(V)[x] = 1
    vEB-Tree-Satellites(V)[x] = data

vEB-Tree-Delete'(V, x):
  if vEB-Tree-Members(V)[x] == 1:
    vEB-Tree-Delete(V, x)
    vEB-Tree-Members(V)[x] = 0
    vEB-Tree-Satellites(V)[x] = NIL
```

If we enforce the rule that all keys in V must have associated satellite data, then we can omit the membership array and instead use the existence of satellite data to determine membership – however, this does not affect asymptotic space usage.

CLRS 20.3-3

As per CLRS 20.3, a vEB(u) tree consists of a u, *min*, and *max* field, and, if u is strictly greater than 2, it also consists of a *summary* reference to a vEB($\sqrt[\uparrow]{u}$) tree as well as a size $\sqrt[\uparrow]{u}$ array *cluster*, each element of which is a reference to a vEB($\sqrt[\downarrow]{u}$) tree.

Using this, our pseudocode for creating new vEB trees is:

```
vEB-Tree-Create(u):
  if u <= 2:
    V; // allocate memory for u, min, and max fields, and
       // store a reference to this memory in variable V.
  else:
    upper_sqrt_u = 2 ** ceil (log2(u) / 2)
    lower_sqrt_u = 2 ** floor(log2(u) / 2)

    V; // allocate memory for u, min, and max fields, as well as
       // a summary reference and upper_sqrt_u cluster references,
       // and store a reference to this memory in variable V.

    V.summary = vEB-Tree-Create(upper_sqrt_u);
    for i = 0 to upper_sqrt_u - 1:
        V.cluster[i] = vEB-Tree-Create(lower_sqrt_u)

  V.u   = u
  V.min = NIL
  V.max = NIL
  return V
```

CLRS 20.3-4

Inserting existing keys

Since x already exists in V, we will never have $V.min = NIL$, and hence the else-block on line 3 executes, and again, since x is in V, we can neither have $x < V.min$ nor $x > V.max$, so lines 4 and 11 are never reached.

The relevant lines of code are then 5-9. If we have $V.u > 2$, we enter the if-block, but since x is in V the if-statement on line 6 is never executed (since $x \in V$ implies that its associated cluster must be non-empty). Hence the recursive call on line 9 is always made. Eventually recursion reaches one of two possible cases: if recursion reaches a non-base case tree V with $V.min = x$, then a spurious extra copy of x is inserted in a base-case tree of size 1 with x as the minimum, which is unintended behavior.

On the other hand, if recursion reaches a base-case tree V with $V.u \leq 2$, then no further code is executed, and behavior is as expected (and in this base-case tree x exists as either $V.min$ or $V.max$).

Deleting non-existing keys

For our analysis of vEB-Tree-Delete(V, x), we assume that $0 \leq x < V.u$, as is assumed in CLRS.

First, if $V.min = V.max$, ie. V contains exactly one element, then vEB-Tree-Delete(V, x) sets $V.min := V.max := NIL$, effectively deleting the singular item in V regardless of its value, which is unwanted and adverse behavior.

Else, if $V.u = 2$ and $x \in U$, then V must contain either 0 or 1 item since we know that $x \notin V$. However, the case where V contains 1 item has already been handled, and hence this elseif-block will only evaluate true when the tree is empty, which is completely unintended behavior, and in fact this has the adverse effect of inserting a 0 or a 1 into the previously empty V.

Else, the else-block on line 9 will evaluate. First, since $x \notin V$, we cannot have $x = V.min$, so this if-block is ignored. Next, the call to vEB-Tree-Delete($V.cluster$[high(x)], low(x)) on line 13 will always be made, but since $x \notin V$, we cannot have low(x) $\in V.cluster$[high(x)] – in fact, we cannot even be sure that this particular cluster even exists, and if it does not then we have undefined behavior upon indexing it.

If it *does* exist, however, then the indexing is well-defined but the recursive deletion may (and will) provoke the unintended/undefined behavior (as described above) in the sub-trees.

Modifying insert and delete

Our solution to this is practically identical to our pseudocode for exercise 20.3-2, in which we modified insert and delete operations to support satellite data, except here, the lines of code pertaining to satellite data are omitted:

```
vEB-Tree-Insert'(V, x):
  // if x not in V: insert x and mark x present.
  if vEB-Tree-Members(V)[x] == 0:
    vEB-Tree-Insert(V, x)
    vEB-Tree-Members(V)[x] = 1

vEB-Tree-Delete'(V, x):
  // if x in V: delete x and mark x not present.
  if vEB-Tree-Members(V)[x] == 1:
    vEB-Tree-Delete(V, x)
    vEB-Tree-Members(V)[x] = 0
```

CLRS 20.3-5

The running times of the vEB tree is given by the recurrence given in (20.4). Consider the same recurrence with clusters of universe size $u^{1-\frac{1}{k}}$:

$$T(u) \leq T(u^{1-\frac{1}{k}}) + O(1)$$

This recurrence is solved similarly to recurrence (20.4) using the master method. We can rewrite by letting $m = \log u$:

$$T(2^m) \leq T(2^{m\left(1-\frac{1}{k}\right)}) + O(1) = T(2^{\frac{m(k-1)}{k}}) + O(1)$$

Letting $S(m) = T(2^m)$ we get:

$$S(m) \leq S\left(\frac{m(k-1)}{k}\right) + O(1) = S\left(\frac{m}{\frac{k}{k-1}}\right) + O(1)$$

Here we have that $a = 1$, $b = \dfrac{k}{k-1}$, $f(m) = O(1)$ and $m^{\log_b a} = m^{\log_{1/(k-1)}(1)} = O(m^0) = O(1)$. Since $f(m) = O(n^{\log_b a}) = O(1)$, case 2 of the master method applies and we get:

$$S(m) = O(m^{\log_{k/(k-1)}(1)} \log(m)) = O(\log m)$$

We see that the solution $S(m) = O(\log m)$ is the same as for recurrence (20.4), so we have $T(u) = T(2^m) = S(m) = O(\log m) = O(\log \log u)$. As a concluding remark, it is evident that the operations run in the same time even though the trees are constructed to have $u^{1/k}$ clusters.

CLRS 20.3-6

The cost of performing n operations is the time taken to create a vEB tree plus the time spent performing n operations: $u + n \log\log u$. Spreading this cost across n operations, we get that the cost of each operation is $O(\log\log u)$. Isolating n in the expression we get that $n \geq \dfrac{u}{\log\log u}$ is the smallest number of operations n for which the amortized time of each operation in a vEB tree is $O(\log\log u)$.

Summaries

psl788

Hashing

- Introduction to hashing
- Hashing properties: strongly and c-approximately universal hash functions
- Applications: coordinated sampling and hash tables with chaining

van Emde Boas Trees

- Introduction and reasoning behind van Emde Boas trees
- Illustrating structure using a small example
- Analysis of running time

wlc376 – exam presentation dispositions

Hashing

1. intro: what is hashing (mapping values from a larger set to a smaller set), and why is it useful (turning variable-length keys into fixed-length keys; improving complexities of various algorithms which require associative arrays)

2. (c-approximate) universality and (c-approximate) strong universality

3. example application: coordinated sampling (?)

Van Emde Boas trees

1. intro: serves same purpose as binary search trees, but with better time complexity ($O(\log \log |U|)$ for most operations) so long as the universe U of possible keys is bounded, but *considerably* worse space complexity $\theta(|U|)$, which is only efficient for reasonably small $|U|$, eg. $|U| \leq 2^{32}$. can improve on space usage, but this is outside of scope for this presentation.

2. intuition: keep a tree structure over the levels of the tree

3. give small example for eg. $U = 16$ (too large??).

4. proof: not quite a proof, but can show run-time of insert (and how omitting recursive storage of min results in the $O(\log \log |U|)$ complexity.)

knx373

Hashing

- Definition and purpose of a hashing function
- Importance of space to represent h, time to compute $h(x)$ and properties of the random variable
- Universality, including strong- and c-approximately universal
- Analysis of example hash function: Multiply-mod-prime or multiply-shift
- Application: Coordinated sampling

van Emde Boas trees

- Introduction to the vEB structure
- Show small example with for example $u = 4$
- What enables the lg(lg(u)) running time insert, delete and successor? Min, max, summary, do not propagate min
- Show running time for insert operation
- Application: Network routers