Lecture 9: Hashing and Skip Lists

(CLRS 11.0–11.3, 11.5, Notes on Skip Lists)

CPS 230, Fall 2001

1 Maintaining ordered set

- Previously we discussed the problem of maintaining an ordered set S under operations
  - Search
  - Insert
  - Delete
  - Successor
  - Predecessor
- We discussed several implementations
  - Array
  - Linked list
  - Skip lists
- We can argue that $\Theta(\log n)$ time is optimal for searching in the decision tree model

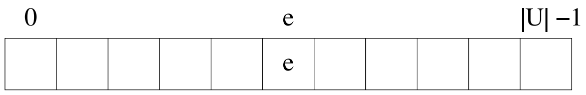
  Recall decision tree model:

  > – Binary tree where each node is labeled $a_i \leq a_j$
  > – Execution corresponds to root-leaf path
  > – Leaf contains result of computation

  - Decision trees correspond to algorithms where we are only allowed to use comparison to gain knowledge about input.
  - Decision tree for search must have n leaves (one for each element)
    $\Longrightarrow$ Tree must have height $\Omega(\log n)$
- In the case of sorting, we saw that we could beat the $\Omega(n \log n)$ decision tree lower bound using *Indirect Addressing* (Radix sort)
  - we can also use indirect addressing idea on ordered set problem.

2 Direct Addressing

- Store element e in cell e of array (we assume elements are integers)



  - Insert/Delete/Search in O(1) time
  - Predecessor/Successor in $O(|U|)$ time ($|U|$ is the size of "universe" U)

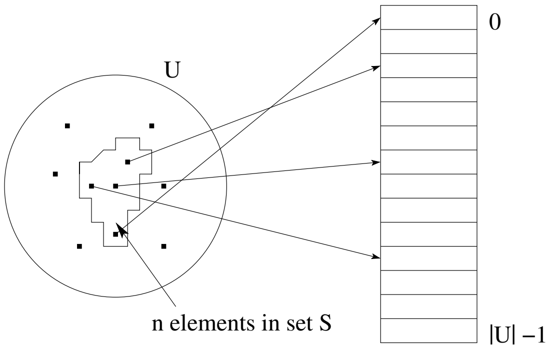
- Note: We could make predecessor/successor efficient by linking neighbor elements, but then *Insert/Delete* becomes $O(|U|)$

- Problem is that $|U|$ can be huge and often $|U| >> n$
  - 32 bit integers $\Longrightarrow |U| = 2^{32}$

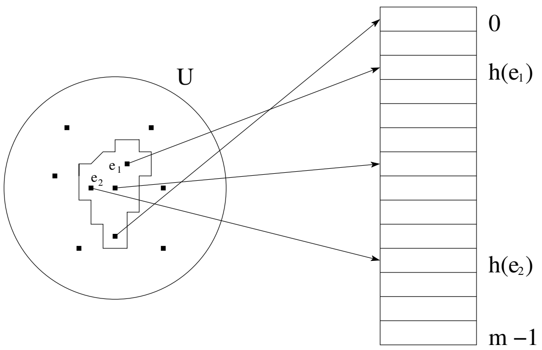
- We can reduce space use using "hashing"

3 Hashing

- To introduce hashing, we look at direct addressing in a slightly different way :

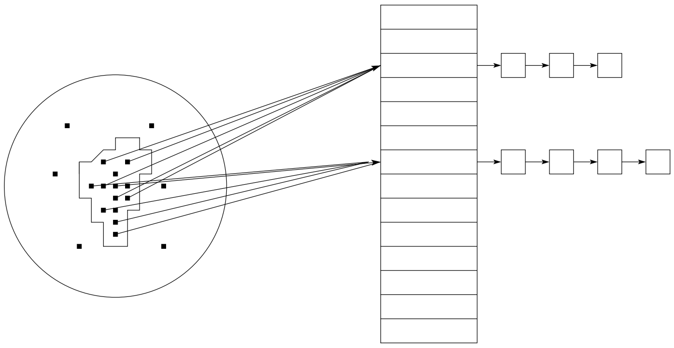


- The main idea is to fix the table size to $m = O(n)$)
  - now element e cannot be stored in cell e
    $\Longrightarrow$ We introduce *hash function* $h(e) : U \rightarrow \{0, 1, \ldots, m-1\}$



We call the array the *hash table*

- Problem is of course that several elements can be stored in same cell ($m < |U|$)
  - We call such an event a *collision*
- We solve this problem using *chaining*
  - Elements mapping to same cell are stored in linked list



  - INSERT/DELETE/SEARCH in O(chain length)
  - PREDECESSOR/SUCCESSOR in $O(m + n)$ since we have to look in all cells and chains

(Note : We assume we can compute $h(e)$ in $O(1)$ time)

- Note: PREDECESSOR/SUCCESSOR bounds are very bad (we will not discuss them further in the following)
  - We call a data structure only supporting INSERT/DELETE/SEARCH a *Dictionary*
  - In a dictionary, order does not really matter
  - Lots of applications of dictionaries, e.g.
    * Symbol table in compilers
    * IP addresses to machine-name table
- Performance of hashing depends on how well $h(e)$ spreads the elements in the hash table
  - Let's make the *simple uniform hashing* assumption:

Any given element is equally likely to hash into any of the m cells

  - Let's analyze the expected time (i.e., expected number of probes in the hash table) for successful and unsuccessful searches as a function of the *load factor* $\alpha = n/m$.
  - Let's define the 0-1 indicator random variable (RV) $X_{i,j} = 1$ if there is a collision between elements i and j, and 0 otherwise.
    * To analyze the expected search time in a successful search, let i, for some $1 \le i \le n$, be the search argument. We assume that i is equally likely to be any of the n keys.

* We pay one probe for i and one probe for each of the later $n-i$ keys that has the same hash value that i does. (We never have to spend time on a key inserted earlier than i was.)

$$\begin{aligned}
E(\text{successful search time} \mid \text{search is for key } i) &= E\left(1 + \sum_{i<j\le n} X_{i,j}\right) \\
&= 1 + \sum_{i<j\le n} E(X_{i,j}) \\
&\le 1 + (n-i)\frac{1}{m}
\end{aligned}$$

But we search for each of the n keys with equal probability $1/n$. Therefore, by conditional expectation, we have

$$\begin{aligned}
E(\text{successful search time}) &= \sum_{1\le i\le n} \Pr\{\text{search for key } i\} \\
&\qquad \times E(\text{successful search time} \mid \text{search for key } i) \\
&= \sum_{1\le i\le n} \frac{1}{n}\left(1 + \frac{n-i}{m}\right) \\
&= \frac{1}{n}\sum_{1\le i\le n}\left(1 + \frac{n-i}{m}\right) \\
&= \frac{1}{n}\left(n + \frac{1}{m}\sum_{1\le i\le n}(n-i)\right) \\
&= \frac{1}{n}\left(n + \frac{1}{m}\sum_{0\le k\le n-1} k\right) \\
&= \frac{1}{n}\left(n + \frac{n(n-1)}{2m}\right) \\
&= 1 + \frac{n-1}{2m} \\
&< 1 + \frac{\alpha}{2}
\end{aligned}$$

* To analyze the expected number of search probes in an unsuccessful search, let x, for some x other than the n keys already in the table, be the search argument. We pay one probe for each of the n keys that have the same hash value that x does.
* We define X_j to be the 0-1 indicator RV that is 1 if j hashes to the same slot as x does, and 0 otherwise.

$$\begin{aligned}
E(\text{unsuccessful search time}) &= E\left(\sum_{1\le j\le n} X_j\right) \\
&= \sum_{1\le j\le n} E(X_j) \\
&\le n\frac{1}{m} \\
&= \alpha
\end{aligned}$$

    - * If $m > n$, then INSERT/DELETE/SEARCH run in $O(1)$ expected time!
  - How do we choose a good hashing function?
    - * Often $h(e) = e \bmod m$ is used ($e \bmod m$ is remainder of e divided by m) Example : $m = 12, e = 100 \Longrightarrow h(e) = 4$ since $100 = 8 \cdot 12 + 4$
    - * m is often chosen to be a prime number far away from a power of 2

      If $m = 2^p$ then $h(e) =$ lowest p bits in e which means that the hashing value only depends on some of the bits in e. If data is not random—not all p-bit patterns equally likely—then this might be a very bad choice, we would rather have $h(e)$ depend on all the bits

4 Universal Hashing

- Given hash function h, we can always find sets of elements that make hashing perform badly (n elements that map to same location)
- Like in quicksort, we can make sure our data structure does not perform badly on a particular input (set of inputs) using randomization
  - We choose a hash function randomly (independent of elements) from a carefully defined set of functions

  - no worst-case inputs
  - good average-case behavior
- We want the set of hash functions to be *universal*

> Let H be a finite collection of functions $U \to 0, 1, \ldots, m-1$.
> H is called **universal** *if and only if* for each $x, y \in U$ the number of functions $h \in H$ for which $h(x) = h(y)$ is at most $|H|/m$.

- If we choose h randomly from H, then the probability of collision between x and y is

$$\frac{|H|/m}{|H|} \leq \frac{1}{m}.$$

- Assuming that we have a universal family H of hash functions, let's analyze the expected time (i.e., expected number of probes in the hash table) for successful and unsuccessful searches as a function of the *load factor* $\alpha = n/m$.
- As before, we define the 0-1 indicator RV $X_{i,j} = 1$ if there is a collision between elements i and j, and 0 otherwise.
  - To analyze the expected search time in a successful search, let i, for some $1 \leq i \leq n$, be the search argument. We pay one probe for i and one probe for each of

the later $n-i$ keys that has the same hash value that i does. (We never have to spend time on a key inserted earlier than i was.)

$$\begin{aligned} E(\text{successful search time}) &= E\left(1+\sum_{i<j\leq n} X_{i,j}\right) \\ &= 1+\sum_{i<j\leq n} E(X_{i,j}) \\ &\leq 1+(n-i)\frac{1}{m} \\ &< 1+\alpha \end{aligned}$$

- To analyze the expected number of search probes in an unsuccessful search, let x, for some x other than the n keys already in the table, be the search argument. We pay one probe for each of the n keys that have the same hash value that x does.
- We define X_j to be the 0-1 indicator RV that is 1 if j hashes to the same slot as x does, and 0 otherwise.

$$\begin{aligned} E(\text{unsuccessful search time}) &= E\left(\sum_{1\leq j\leq n} X_j\right) \\ &= \sum_{1\leq j\leq n} E(X_j) \\ &\leq n\frac{1}{m} \\ &= \alpha \end{aligned}$$

- If $m > n$, then Insert/Delete/Search run in $O(1)$ expected time.

- All we need now is a universal family of hash functions. Here's one such family: Let p be a prime large enough so that all our keys are in the range from 0 to $p-1$. For each $0 \leq a < p$ and for each $0 \leq b < m$, let's define the hash function $h_{a,b}$ as follows:

$$h_{a,b}(k) = ((ak+b) \bmod p) \bmod m.$$

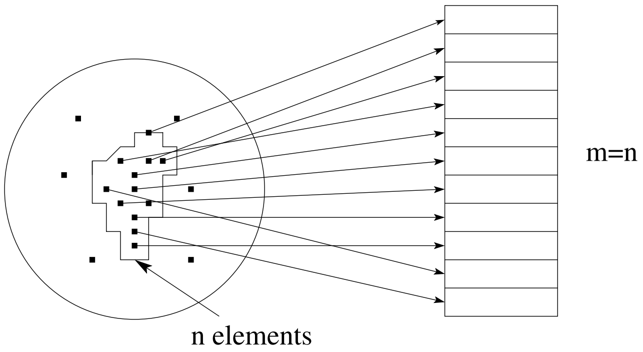
  That is, $h_{a,b}$ is parameterized by two quantities: a and b.

- If the parameters a and b are chosen randomly, the book shows in Theorem 11.5 that the probability of collision of two distinct keys k and ℓ in the range from 0 to $p-1$ is at most $1/m$. See the proof for details; it uses only basic number theory.

5 Dynamic perfect hashing

- Let's suppose we have a universal set of hash functions. (See above.)
- It turns out that we can design a method to do searches in $O(1)$ *worst-case* time.
- We will construct a hash table so that each key can be looked up in constant time in the worst case. The expected time to construct the table and the expected space used by the table will be random variables with mean $O(n)$. (In fact, the probability that these quantities are larger than kn will be exponentially small in k.)

- For simplicity, let's restrict ourselves to a static set of keys. That is, we won't allow insertions and deletions.
- Idea:
  - If the set of n keys is static, we could potentially find a *perfect* hash function h that has *no* collisions for the n keys:



  - We need to be able to store the description of h compactly and to compute h fast.
  - If $m \gg n$, a random hash function is likely to be perfect. In particular, if $m = n^2$, the probability of no collisions in the entire table is at least $1/2$.
  - For keys i and j, let $X_{i,j}$ be the 0-1 indicator RV such that $X_{i,j} = 1$ if there is a collision between elements i and j, and 0 otherwise.

$$
\begin{aligned}
\Pr\{\text{there is a collision}\} &\leq \sum_{1 \leq i < j \leq n} \Pr\{X_{i,j} = 1\} \\
&\leq \sum_{1 \leq i < j \leq n} \frac{1}{m} \\
&= \frac{n(n-1)}{2} \frac{1}{n^2} \\
&< \frac{1}{2}
\end{aligned}
$$

  - But we don't want to use $m = n^2$ slots. So instead we choose $m = n$ and pick a random universal hash function. Then we assign the n keys to slots. There will likely be collisions. For each slot j, for $1 \leq j \leq m = n$, we take the n_j keys that hash to slot j and put them (and no other keys) into a secondary hash table of quadratic size ${n_j}^2$. We pick hash functions at random until we can hash the n_j items with *no collisions at all.*
  - For the hash function we end up with at slot j, we store its parameters (say, a and b if we use the universal family of hash functions defined earlier) in slot j so that whenever we need to query the secondary hash table for slot j, we will know which hash function to use.
  - The expected number of hash functions we have to try for each slot j before we find a perfect hash function is at most 2, since we have at least a $1/2$ chance of finding a perfect hash function for each (random) hash function we try. The expected cost is therefore $O(n_j)$.

- Therefore, the expected cost for all the slots is at most $\sum_{1 \le j \le n} O(n_j) = O(n)$. In fact, we can get a stronger result that the probability that the extra cost is more than kn is exponentially small in k.
- In terms of space, the hash table for slot j uses $O({n_j}^2)$ space, so the total space is $O(\sum_{1 \le j \le n} {n_j}^2)$. What is its expected value?
- Theorem 11.10 shows that the expected total space used is linear:

$$\begin{aligned}
E\left(\sum_{1 \le j \le n} {n_j}^2\right) &= E\left(\sum_{1 \le j \le n}\left(n_j + 2\binom{n_j}{2}\right)\right) \\
&= E\left(\sum_{1 \le j \le n} n_j\right) + 2E\left(\sum_{1 \le j \le n}\binom{n_j}{2}\right) \\
&= E(n) + 2E\left(\sum_{1 \le j \le n} \text{number of collisions in slot } j\right) \\
&= n + 2E(\text{total number of collisions}) \\
&\le n + 2\binom{n}{2}\frac{1}{m} \\
&= n + 2\frac{n-1}{2} \\
&= 2n - 1
\end{aligned}$$

- Corollary 11.12 strengthens Theorem 11.10 to show that the probability of using more than $4n$ space is less than $1/2$.

$\Longrightarrow$ We can start over from scratch until we end up using no more than $4n$ space. With high probability, we'll use $O(n)$ time.

- The constant-time hashing idea can even be made dynamic so that we get $O(1)$ Insert and Delete expected running time.
- Lots of recent results even improve on these ideas.

6 Bucket Sort à la CLRS Section 8.4

- The above analysis is very similar to what is used for the average-case analysis of the Bucket Sort method discussed in the CLRS book in Section 8.4, under the assumption that the keys are drawn from a uniform distribution.
- The algorithm works by partitioning the range of key values into n contiguous buckets. It puts each item into its appropriate bucket (computed by doing a simple division operation). Let n_i be the number of items in bucket i. Then for each bucket i, for $1 \le i \le n$, it sorts the n_i items in bucket i using a simple quadratic-time method like insertion sort. Therefore, the total time to sort is $O(n + \sum_{1 \le i \le n} {n_i}^2)$.
- $\Longrightarrow$ The expected time to sort is $O(n) + E(\sum_{1 \le i \le n} {n_i}^2) = O(n) + \sum_{1 \le i \le n} E({n_i}^2)$. All that remains is to compute $E({n_i}^2)$.

- 0-1 indicator RVs to the rescue! Define $X_j = 1$ if item j hashes to bucket i.

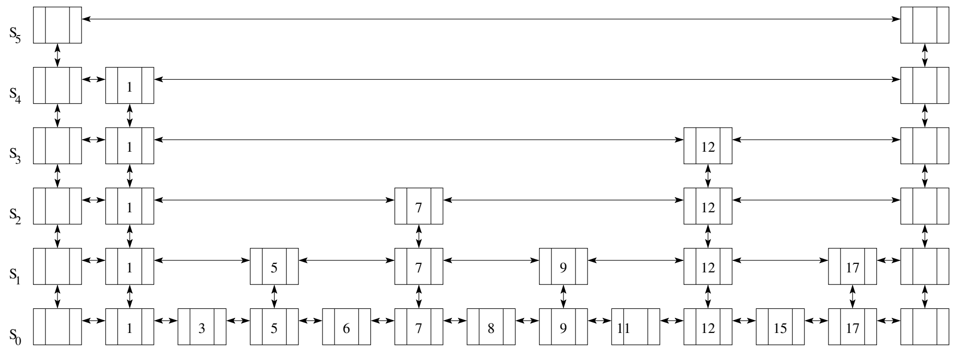
$$
\begin{aligned}
E({n_i}^2) &= E\left(\left(\sum_{1\le j\le n} X_j\right)^2\right) \\
&= E\left(\sum_{1\le j,k\le n} X_j X_k\right) \\
&= \sum_{1\le j,k\le n} E(X_j X_k) \\
&= \sum_{1\le j\le n} E({X_j}^2) + \sum_{\substack{1\le j,k\le n\\ j\ne k}} E(X_j X_k) \\
&= \sum_{1\le j\le n} E(X_j) + \sum_{\substack{1\le j,k\le n\\ j\ne k}} E(X_j X_k) \\
&= \sum_{1\le j\le n} \frac{1}{n} + \sum_{\substack{1\le j,k\le n\\ j\ne k}} \frac{1}{n^2} \\
&= n\frac{1}{n} + n(n-1)\frac{1}{n^2} \\
&= 2 - \frac{1}{n}
\end{aligned}
$$

- Therefore, the expected time to sort is $O(n) + \sum_{1\le i\le n} E({n_i}^2) = O(n)$.

7 Skip Lists

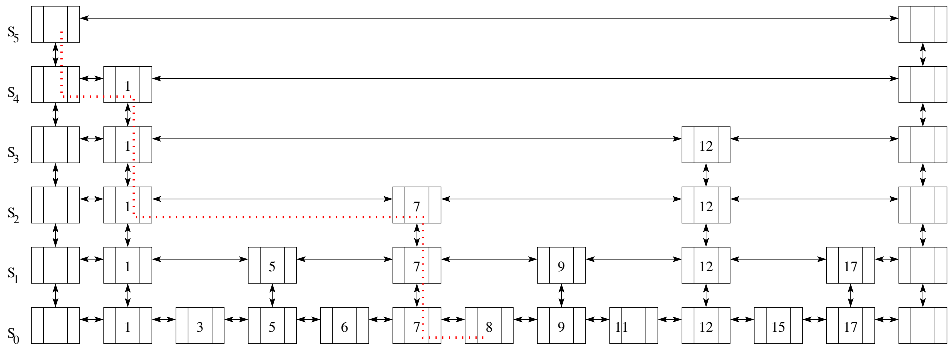
- There are several schemes for keeping search trees reasonably balanced so that we can obtain $O(\log n)$ bounds
  - Often quite complicated.
  - Red-black trees one of the simplest to actually implement.
- When we discussed quicksort we saw how randomization can lead to good expected (average-case) running times.
  - We will now discuss how randomization can be used within the algorithm to obtain a very simple search structure with average-case performance $O(\log n)$ (independent of the data or the operations!)
- Idea in a skip list is best illustrated if we try to build a "search tree" on top of double linked list:
  - Insert elements $-\infty$ and ∞
  - Repeatedly construct double linked list (level S_i) on top of current list (level S_{i-1}) by choosing every second element (and link equal elements together)

    $\Longrightarrow$ Number of levels is $O(\log n)$



– $Search(e)$: Start at topmost left element. Repeatedly drop down one level and search forward until max element $\leq e$ is found.

Example: Search for 8:



$O(\log n)$ time since we move at most one step to the right at each level.

– $Predecessor/Successor$ also in $O(\log n)$ time

– $Insert/Delete$ seems hard to do in better than $O(n)$ time since we might need to rebuild the entire structure after one of the operations.

- Idea in skip list is to let level S_{i+1} consist of a randomly generated subset of elements at level S_i.

  – To decide if an element on level S_i should be on level S_{i+1}, we flip a coin and include the element if it is heads.
  $\Longrightarrow$
  Expected size of S_1 is $n/2$
  Expected size of S_2 is $n/4$
  $\vdots$
  Expected size of S_i is $n/2^i$

- Operations:

  – $Search(e)$ as before.

  – $Delete(e)$: Search to find e and delete all occurrences of e.

  – $Insert(e)$:

    * search to find position of e in S_0
    * Insert e in S_0.
    * Repeatedly flip a coin; insert e and continue to next level if it comes up heads.

- Running time of all the operations is bounded by search running time
  - Down search:
    * Expected time for down search is $E(height)$. Let X_i be the indicator RV that is 1 if there is at least one element on level i. We can therefore express *height* simply as $\sum_{i \geq 0} X_i$.
    * Note that $X_i = \min\{1, |S_i|\}$, where S_i is the set of elements on level i. So X_i can be bounded by either 1 or $|S_i|$.

$$\begin{aligned} height &= \sum_{i \geq 0} X_i \\ &\leq \sum_{i \geq 0} \min\{1, |S_i|\} \\ &\leq \sum_{0 \leq i \leq \log n} 1 + \sum_{i \geq \log n} |S_i| \end{aligned}$$

    For purposes of bounding the sum, it's often useful to break it into two parts and use a different approach for each part. We divided the above sum at level $i = \log n$, since that's when $|S_i| = n/2^i$ starts getting very small. Taking expectations, and by linearity of expectation, we get

$$\begin{aligned} E(height) &\leq E\left(\sum_{0 \leq i \leq \log n} 1\right) + E\left(\sum_{i \geq \log n} |S_i|\right) \\ &= \log n + 1 + \sum_{i \geq \log n} E(|S_i|) \\ &= \log n + 1 + \sum_{i > \log n} n/2^i \\ &= \log n + 1 + \sum_{k > 0} 1/2^k \\ &= \log n + 2. \end{aligned}$$

    * We can easily show (see notes) that $\Pr\{height > c \log n\} \leq 1/n^{c-1}$, which is exponentially small in c.
  - Search forward:
    * If we scan an element on level i, it cannot be on level $i+1$ (because then we would have scanned it there)
      $\Downarrow$
    * Expected number of elements we access while searching forward on level i is at most 1 plus the expected number of successive elements that were not promoted to level $i+1$, which is the number of times we have to flip a coin to get a head.
      $\Downarrow$

    * Expected number of elements we search forward on level i is $1(1/2)+2(1/4)+3(1/8)+\ldots=\sum_{k\geq 1} k/2^k = 2$.

      $\Downarrow$

    * Expected total number of elements we access while searching forward is at most $2 \times E(height) = O(\log n)$.

- Expected space use:
  - Space use is $\sum_{i\geq 0} |S_i|$.
  - Expected space use is therefore $\sum_{i\geq 0} E(|S_i|) = \sum_{i\geq 0} n/2^i = 2n$.
- Note:
  - We only really need forward and down pointers.
  - Running times are expected because of randomization.
  - High-probability bounds can be proven to show that the running time is most likely close to the expected value.