

CS 561, Lecture 2 : Randomization in Data Structures

Jared Saia
University of New Mexico

Outline

- Hash Tables
- Bloom Filters
- Skip Lists

Dictionary ADT

A dictionary ADT implements the following operations

- *Insert(x)*: puts the item x into the dictionary
- *Delete(x)*: deletes the item x from the dictionary
- *IsIn(x)*: returns true iff the item x is in the dictionary

Dictionary ADT

- Frequently, we think of the items being stored in the dictionary as *keys*
- The keys typically have *records* associated with them which are carried around with the key but not used by the ADT implementation
- Thus we can implement functions like:
 - *Insert(k,r)*: puts the item (k,r) into the dictionary if the key k is not already there, otherwise returns an error
 - *Delete(k)*: deletes the item with key k from the dictionary
 - *Lookup(k)*: returns the item (k,r) if k is in the dictionary, otherwise returns null

Implementing Dictionaries

- The simplest way to implement a dictionary ADT is with a linked list
- Let l be a linked list data structure, assume we have the following operations defined for l
 - $\text{head}(l)$: returns a pointer to the head of the list
 - $\text{next}(p)$: given a pointer p into the list, returns a pointer to the next element in the list if such exists, null otherwise
 - $\text{previous}(p)$: given a pointer p into the list, returns a pointer to the previous element in the list if such exists, null otherwise
 - $\text{key}(p)$: given a pointer into the list, returns the key value of that item
 - $\text{record}(p)$: given a pointer into the list, returns the record value of that item

At-Home Exercise

Implement a dictionary with a linked list

- Q1: Write the operation `Lookup(k)` which returns a pointer to the item with key `k` if it is in the dictionary or null otherwise
- Q2: Write the operation `Insert(k,r)`
- Q3: Write the operation `Delete(k)`
- Q4: For a dictionary with n elements, what is the runtime of all of these operations for the linked list data structure?
- Q5: Describe how you would use this dictionary ADT to count the number of occurrences of each word in an online book.

_____ Dictionaries _____

- This linked list implementation of dictionaries is very slow
- Q: Can we do better?
- A: Yes, with hash tables, AVL trees, etc

Hash Tables

Hash Tables implement the Dictionary ADT, namely:

- Insert(x) - $O(1)$ expected time, $\Theta(n)$ worst case
- Lookup(x) - $O(1)$ expected time, $\Theta(n)$ worst case
- Delete(x) - $O(1)$ expected time, $\Theta(n)$ worst case

Direct Addressing

- Suppose universe of keys is $U = \{0, 1, \dots, m - 1\}$, where m is not too large
- *Assume no two elements have the same key*
- We use an array $T[0..m - 1]$ to store the keys
- Slot k contains the elem with key k

Direct Address Functions

```
DA-Search(T,k){ return T[k];}
```

```
DA-Insert(T,x){ T[key(x)] = x;}
```

```
DA-Delete(T,x){ T[key(x)] = NIL;}
```

Each of these operations takes $O(1)$ time

Direct Addressing Problem

- If universe U is large, storing the array T may be impractical
- Also much space can be wasted in T if number of objects stored is small
- Q: Can we do better?
- A: Yes we can trade time for space

Hash Tables

- “Key” Idea: An element with key k is stored in slot $h(k)$, where h is a *hash function* mapping U into the set $\{0, \dots, m-1\}$
- Main problem: Two keys can now hash to the same slot
- Q: How do we resolve this problem?
- A1: Try to prevent it by hashing keys to “random” slots and making the table large enough
- A2: Chaining
- A3: Open Addressing

Chained Hash

In chaining, all elements that hash to the same slot are put in a linked list.

```
CH-Insert(T,x){Insert x at the head of list T[h(key(x))];}
```

```
CH-Search(T,k){search for elem with key k in list T[h(k)];}
```

```
CH-Delete(T,x){delete x from the list T[h(key(x))];}
```

Analysis

- CH-Insert and CH-Delete take $O(1)$ time if the list is doubly linked and there are no duplicate keys
- Q: How long does CH-Search take?
- A: It depends. In particular, depends on the *load factor*, $\alpha = n/m$ (i.e. average number of elems in a list)

CH-Search Analysis

- Worst case analysis: everyone hashes to one slot so $\Theta(n)$
- For average case, make the *simple uniform hashing* assumption: any given elem is equally likely to hash into any of the m slots, indep. of the other elems
- Let n_i be a random variable giving the length of the list at the i -th slot
- Then time to do a search for key k is $1 + n_{h(k)}$

CH-Search Analysis

- Q: What is $E(n_{h(k)})$?
- A: We know that $h(k)$ is uniformly distributed among $\{0, \dots, m-1\}$
- Thus, $E(n_{h(k)}) = \sum_{i=0}^{m-1} (1/m)n_i = n/m = \alpha$

Hash Functions

- Want each key to be equally likely to hash to any of the m slots, independently of the other keys
- Key idea is to use the hash function to “break up” any patterns that might exist in the data
- We will always assume a key is a natural number (can e.g. easily convert strings to natural numbers)

Division Method

- $h(k) = k \bmod m$
- Want m to be a *prime number*, which is not too close to a power of 2
- Why? Reduces collisions in the case where there is periodicity in the keys inserted

Hash Tables Wrapup

Hash Tables implement the Dictionary ADT, namely:

- Insert(x) - $O(1)$ expected time, $\Theta(n)$ worst case
- Lookup(x) - $O(1)$ expected time, $\Theta(n)$ worst case
- Delete(x) - $O(1)$ expected time, $\Theta(n)$ worst case

Bloom Filters

- Randomized data structure for representing a set. Implementations:
- Insert(x) :
- IsMember(x) :
- Allow false positives but require very little space
- Used frequently in: Databases, networking problems, p2p networks, packet routing

Bloom Filters

- Have m slots, k hash functions, n elements; assume hash functions are all independent
- Each slot stores 1 bit, initially all bits are 0
- Insert(x) : Set the bit in slots $h_1(x), h_2(x), \dots, h_k(x)$ to 1
- IsMember(x) : Return yes iff the bits in $h_1(x), h_2(x), \dots, h_k(x)$ are all 1

Analysis Sketch

- m slots, k hash functions, n elements; assume hash functions are all independent
- Then $P(\text{fixed slot is still } 0) = (1 - 1/m)^{kn}$
- Useful fact from Taylor expansion of e^{-x} :
 $e^{-x} - x^2/2 \leq 1 - x \leq e^{-x}$ for $x < 1$
- Then if $x \leq 1$

$$e^{-x}(1 - x^2) \leq 1 - x \leq e^{-x}$$

Analysis

- Thus we have the following to good approximation.

$$\begin{aligned} P(\text{fixed slot is still } 0) &= (1 - 1/m)^{kn} \\ &\approx e^{-km/n} \end{aligned}$$

- Let $p = e^{-kn/m}$ and let ρ be the fraction of 0 bits after n elements inserted then

$$P(\text{false positive}) = (1 - \rho)^k \approx (1 - p)^k$$

- Where the first approximation holds because ρ is very close to p (by a Martingale argument beyond the scope of this class)

Analysis

- Want to minimize $(1 - p)^k$, which is equivalent to minimizing $g = k \ln(1 - p)$
- Trick: Note that $g = -(n/m) \ln(p) \ln(1 - p)$
- By symmetry, this is minimized when $p = 1/2$ or equivalently $k = (m/n) \ln 2$
- False positive rate is then $(1/2)^k \approx (.6185)^{m/n}$

Tricks

- Can get the union of two sets by just taking the bitwise-or of the bit-vectors for the corresponding Bloom filters
- Can easily half the size of a bloom filter - assume size is power of 2 then just bitwise-or the first and second halves together
- Can approximate the size of the intersection of two sets - inner product of the bit vectors associated with the Bloom filters is a good approximation to this.

Extensions

- Counting Bloom filters handle deletions: instead of storing bits, store integers in the slots. Insertion increments, deletion decrements.
- Bloomier Filters: Also allow for data to be inserted in the filter - similar functionality to hash tables but less space, and the possibility of false positives.

Skip List

- Enables insertions and searches for ordered keys in $O(\log n)$ expected time
- Very elegant randomized data structure, simple to code but analysis is subtle
- They guarantee that, with high probability, all the major operations take $O(\log n)$ time (e.g. Find-Max, Find i -th element, etc.)

Skip List

- A skip list is basically a collection of doubly-linked lists, L_1, L_2, \dots, L_x , for some integer x
- Each list has a special head and tail node, the keys of these nodes are assumed to be $-\text{MAXNUM}$ and $+\text{MAXNUM}$ respectively
- The keys in each list are in sorted order (non-decreasing)

Skip List

- Every node is stored in the bottom list
- For each node in the bottom list, we flip a coin over and over until we get tails. For each heads, we make a duplicate of the node.
- The duplicates are stacked up in levels and the nodes on each level are strung together in sorted linked lists
- Each node v stores a search key ($\text{key}(v)$), a pointer to its next lower copy ($\text{down}(v)$), and a pointer to the next node in its level ($\text{right}(v)$).

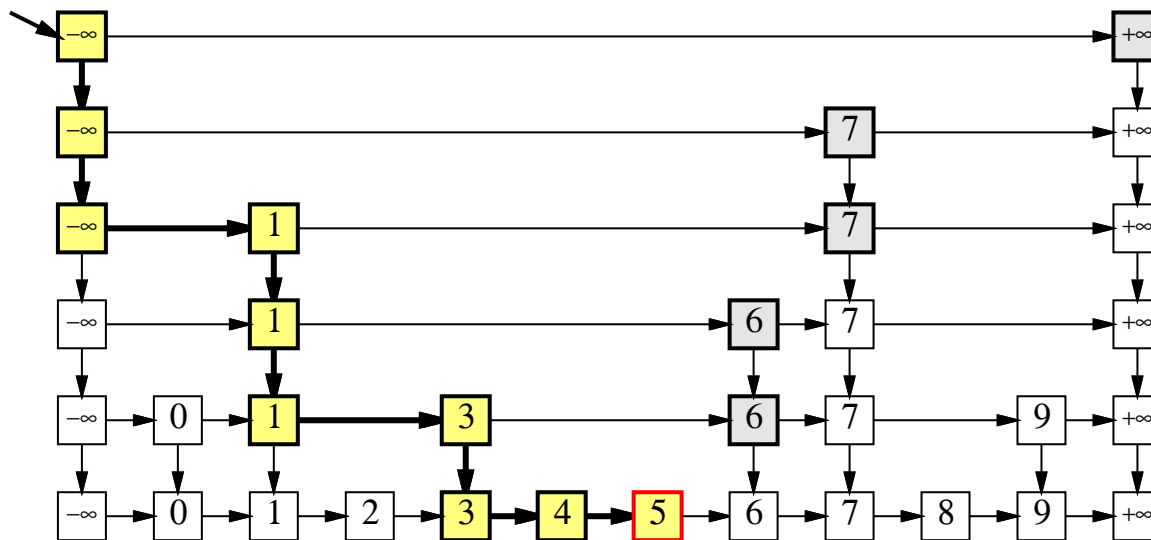
Search

- To do a search for a key, x , we start at the leftmost node L in the highest level
- We then scan through each level as far as we can without passing the target value x and then proceed down to the next level
- The search ends either when we find the key x or fail to find x on the lowest level

Search

```
SkipListFind(x, L){  
    v = L;  
    while (v != NULL) and (Key(v) != x){  
        if (Key(Right(v)) > x)  
            v = Down(v);  
        else  
            v = Right(v);  
    }  
    return v;  
}
```


Search Example



Insert

p is a constant between 0 and 1, typically $p = 1/2$, let `rand()` return a random value between 0 and 1

```
Insert(k){
```

```
  First call Search(k), let pLeft be the leftmost elem  $\leq k$  in L1
```

```
  Insert k in L1, to the right of pLeft
```

```
  i = 2;
```

```
  while (rand() $\leq$  p){
```

```
    insert k in the appropriate place in Li;
```

```
}
```

Deletion

- Deletion is very simple
- First do a search for the key to be deleted
- Then delete that key from all the lists it appears in from the bottom up, making sure to “zip up” the lists after the deletion

Analysis

- Intuitively, each level of the skip list has about half the number of nodes of the previous level, so we expect the total number of levels to be about $O(\log n)$
- Similarly, each time we add another level, we cut the search time in half except for a constant overhead
- So after $O(\log n)$ levels, we would expect a search time of $O(\log n)$
- We will now formalize these two intuitive observations

Height of Skip List

- For some key, i , let X_i be the maximum height of i in the skip list.
- Q: What is the probability that $X_i \geq 2 \log n$?
- A: If $p = 1/2$, we have:

$$\begin{aligned} P(X_i \geq 2 \log n) &= \left(\frac{1}{2}\right)^{2 \log n} \\ &= \frac{1}{(2^{\log n})^2} \\ &= \frac{1}{n^2} \end{aligned}$$

- Thus the probability that a particular key i achieves height $2 \log n$ is $\frac{1}{n^2}$

Height of Skip List

- Q: What is the probability that *any* key achieves height $2 \log n$?
- A: We want

$$P(X_1 \geq 2 \log n \text{ or } X_2 \geq 2 \log n \text{ or } \dots \text{ or } X_n \geq 2 \log n)$$

- By a Union Bound, this probability is no more than

$$P(X_1 \geq k \log n) + P(X_2 \geq k \log n) + \dots + P(X_n \geq k \log n)$$

- Which equals:

$$\sum_{i=1}^n \frac{1}{n^2} = \frac{n}{n^2} = 1/n$$

Height of Skip List

- This probability gets small as n gets large
- In particular, the probability of having a skip list of size exceeding $2 \log n$ is $o(1)$
- If an event occurs with probability $1 - o(1)$, we say that it occurs *with high probability*
- *Key Point:* The height of a skip list is $O(\log n)$ with high probability.

In-Class Exercise Trick

A trick for computing expectations of discrete positive random variables:

- Let X be a discrete r.v., that takes on values from 1 to n

$$E(X) = \sum_{i=1}^n P(X \geq i)$$

Why?

$$\begin{aligned}\sum_{i=1}^n P(X \geq i) &= P(X = 1) + P(X = 2) + P(X = 3) + \dots \\ &+ P(X = 2) + P(X = 3) + P(X = 4) + \dots \\ &+ P(X = 3) + P(X = 4) + P(X = 5) + \dots \\ &+ \dots \\ &= 1 * P(X = 1) + 2 * P(X = 2) + 3 * P(X = 3) + \dots \\ &= E(X)\end{aligned}$$

In-Class Exercise

Q: How much memory do we expect a skip list to use up?

- Let X_i be the number of lists that element i is inserted in.
- Q: What is $P(X_i \geq 1)$, $P(X_i \geq 2)$, $P(X_i \geq 3)$?
- Q: What is $P(X_i \geq k)$ for general k ?
- Q: What is $E(X_i)$?
- Q: Let $X = \sum_{i=1}^n X_i$. What is $E(X)$?

Search Time

- Its easier to analyze the search time if we imagine running the search backwards
- Imagine that we start at the found node v in the bottommost list and we trace the path backwards to the top leftmost sentinel, L
- This will give us the length of the search path from L to v which is the time required to do the search

Backwards Search

```
SLFback(v){
  while (v != L){
    if (Up(v) != NIL)
      v = Up(v);
    else
      v = Left(v);
  }
}
```

Backward Search

- For every node v in the skip list $Up(v)$ exists with probability $1/2$. So for purposes of analysis, SLFBack is the same as the following algorithm:

```
FlipWalk(v){
  while (v != L){
    if (COINFLIP == HEADS)
      v = Up(v);
    else
      v = Left(v);
  }
}
```

Analysis

- For this algorithm, the expected number of heads is exactly the same as the expected number of tails
- Thus the expected run time of the algorithm is twice the expected number of upward jumps
- Since we already know that the number of upward jumps is $O(\log n)$ with high probability, we can conclude that the expected search time is $O(\log n)$