CS 161: Design and Analysis of Algorithms

Mark Zhandry
Data Structures I: Storing Unordered Data

- Arrays
- Linked Lists
- Stacks
- Queues
- Hash Tables
The Problem

- We have a collection of items we would like to store for later retrieval.
- Items are not comparable (i.e. no notion of $x < y$)
Array Lists

Algorithms are exciting!
Array Lists

• Operations:
  – get(i) returns the value at index i
  – add(i,x) inserts x at index i
    • All values after index i move down the array
  – remove(i) removes the value at index i
    • All values after index i move up the array
Algorithms are exciting!

- Turns out that the time to retrieve values does not depend on length of array (i.e. O(1) time)
Algorithms are exciting!

- add(i,x) requires time proportional to the number of shifts
  - $O(1)$ time at end of array ***
  - $O(n)$ at beginning
remove(1)

- remove(i) requires time proportional to the number of shifts
  - $O(1)$ time at end of array
  - $O(n)$ at beginning or middle

Algorithms exciting!
• If we call add(1, “are”), no room to insert!
Solution: Dynamic Arrays

• Automatically grows when space runs out.
• Must create new array and copy
  – If we grow by 1, we will do many copy operations
  – Instead, we grow by doubling the size
Dynamic Arrays

Index:

0 1 2 3 4 5 6 7 8

very and in fields!

Algorithms exciting useful many!
Amortized Analysis

• Adding to beginning or middle still takes $O(n)$
• Adding to end now sometimes takes $O(n)$
  – However, most of the time $O(1)$.
• Can we make any stronger statements than $O(n)$?
• Amortized Analysis: consider sequence of operations
Amortized Analysis

- What if we add \( n \) values to an initially empty array, always adding to the end? Say \( n = 2^k \)
- Suppose initial size of array is 1, and \( n = 2^k \)
- After adding all \( n \) values, how many copies have we made?
- How much total space has been allocated?
Amortized Analysis

• We double only when the array has size $2^i$ ($i < k$), and the array is full.
  – Number of copies: $2^i$.
  – Amount of space allocated: $2^{i+1}$.

• Total copies:

• Total space:
Amortized Analysis

• To perform \( n = 2^k \) adds to the end of an array takes \( O(n) \) time.

• What about \( n \neq 2^{k'} \)?

• Find the smallest \( n' = 2^k \) such that \( n' \geq n \). Notice that \( n > n'/2 \).

• \( n \) adds take less time than \( n' \) adds, which take \( O(n') = O(2n) = O(n) \) time.
Amortized Analysis

• So any n adds to the end of an array take $O(n)$ time.
• On average, add operations take $O(1)$ time!
Linked Lists

This is a list
Linked Lists

• Operations:
  – get(i) returns the value at index i
  – add(i,x) inserts x at index i
    • All values after index i get higher index
  – remove(i) removes the value at index i
    • All values after index i get lower index
get(4)

• get(i) takes $O(i)$ time
  – $O(1)$ at beginning
  – $O(n)$ at middle or end
add(3,"linked")

This is a list

- add(i,x) takes $O(i)$ time.
  - $O(1)$ at beginning
  - $O(n)$ at middle or end
remove(2)

This is a list

- remove(i) takes $O(i)$ time.
  - $O(1)$ at beginning
  - $O(n)$ at middle or end
Optimization: Doubly Linked List

- add(i, x), get(i), remove(i) take $O(\min(i, n-i))$ time
  - $O(1)$ at beginning or end
  - $O(n)$ in the middle
## Arrays vs. Lists

<table>
<thead>
<tr>
<th></th>
<th>add, remove</th>
<th>get</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Beginning</td>
<td>Middle</td>
</tr>
<tr>
<td><strong>Array</strong></td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Linked List</strong></td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Doubly Linked List</strong></td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
Stacks

- Last in, first out (LIFO) behavior
- Operations:
  - `push(x)`: adds element $x$ to top of stack
  - `peek()`: returns top element of stack w/o removal
  - `pop()`: removes and returns top of stack
Implementing Stacks

• With a linked list:
  – push(x) = add(0, x) \( \text{O}(1) \)
  – peek() = get(0) \( \text{O}(1) \)
  – pop() = {
      temp = get(0)
      remove(0)
      return temp
  } \( \text{O}(1) \)
Implementing Stacks

• With a dynamic array:
  – push(x) = add(n,x) \( O(1) \)
  – peek() = get(n-1) \( O(1) \)
  – pop() = {
    temp = get(n-1)
    remove(n-1)
    return temp
  } \( O(1) \)
Queues

• First in, first out (FIFO) behavior
• Operations:
  – add(x): adds element to end of queue
  – peek(x): returns head of queue w/o removal
  – poll(x): removes and returns head of queue
Implementing Queues

• With a double linked list:
  
  – add(x) = add(0, x) \quad O(1)
  
  – peek() = get(n-1, x) \quad O(1)
  
  – poll() = {
    temp = get(n-1)
    remove(n-1)
    return temp
  } \quad O(1)
Implementing Queues

• With a dynamic array?
  – add(x) = add(n-1,x) \( O(1) \)
  – peek() = get(0) \( O(1) \)
  – poll() = {
      temp = get(0)
      remove(0)
      return temp
  }
Dictionaries

• In list structures, we look up value by its index
• What if we want to look up value by some other key?
• Example:
  – Want to store GPAs of all students, organized by student name.
Dictionaries

• Desired Operations:
  – add(key, value): associates value to key
  – lookup(key): returns the value associated to key
  – remove(key): removes key and corresponding value from dictionary
Implementing Dictionaries

- With a dynamic array:
  - `add(key, value) = add(n, (key, value))` $\mathcal{O}(1)$
  - Duplicates?
  - `lookup(key) = {` $\mathcal{O}(n)$
    For pair `(key’, value’) in list:
    If key’ = key, return value’
  `}`
  - `remove(key) = {` $\mathcal{O}(n)$
    For pair `(key’, value’) in list:
    If key’ = key, remove this pair
  `}`
Problem

• If we want constant time operations, we need an array

• Arrays indexed by integers, we want indexed by keys

• Even if keys are integers, we may want to allow keys much larger than the size of the array
Idea: Hash Tables

• Let $K$ be the space of possible keys
• Let $\{0,\ldots,n-1\}$ be the possible indices of a length-$n$ array
• Choose a function $h: K \rightarrow \{0,\ldots,n-1\}$
  – Called a hash function
Idea: Hash Tables

- Choose a function $h: K \rightarrow \{0, ..., n-1\}$
- $\text{add(key, value)}$: put value into index $h(key)$ $O(1)$
- $\text{lookup(key)}$: get value at $h(key)$ $O(1)$
- $\text{remove(key)}$: delete value at $h(key)$ $O(1)$
Problem

• What if $h(\text{key}) = h(\text{key'})$?
  – We call this a collision

• Solution: Instead of storing value at each index, store a (linked or array) list of values
Chaining

• Each index of the array points to a linked list of (key,value) pairs.

• add(key,value) = {
  – Compute h(key), and let L be the list stored at the index h(key).
  – Search L for a pair (key’,value’) with key=key’
  – If pair found, replace with (key,value)
  – Otherwise add (key,value) to end.
}
Chaining

- Similar operation for lookup and get
- Running time:
  - Must scan entire list, so all operations $O(|L|)$
Picking a good function

- All operations proportional to size of the chains
- To make efficient, need chains to be small (preferably constant size)
Idea 1: One-to-one function

• Ideally want $h(\text{key}_1) \neq h(\text{key}_2)$ for all $\text{key}_1 \neq \text{key}_2$

• Example: $K =$ strings with $m$ characters
  – Let space = 0, a = 1, b = 2, ..., A = 27, B = 28, ...
  – Max value of character: 52
  – If $s=s_1s_2...s_m$, then $h(s) = s_m + 53s_{m-1} + 53^2 s_{m-2} + ... + 53^{m-1} s_1$

• Problem: need $n \geq 53^m$
Idea 2: Many-to-one

• Pick some function $h$ that maps the same number of keys to each index
• Example: $K = m$-bit integers
  – Let $h(x) = x \mod n$
• Problem:
  – What if I happen to store a bunch of values that map to the same index? (ex: $0, n, 2n, \ldots$)
  – Solution: randomness!
Idea 3: Random Functions

• Let $h(key)$ be a random value for each key

• Problem:
  – Need every evaluation of $h(key)$ to return the same value
  – Must remember $h(key)$ for further evaluations
  – Looking up $h(key)$ requires an efficient dictionary!
Idea 4: Universal Hashing

• To minimize collisions, truly random functions are overkill
• Pick from small set of functions that “appear” random
Universal Hashing

- Let $H$ be some subset of the functions from $K$ to $\{0,\ldots, n-1\}$ with the following property:
  - For all $key_1 \neq key_2$

  \[
  \Pr_{h \leftarrow H}[h(key_1) = h(key_2)] \leq \frac{1}{n}
  \]
Example

• Say \( n \) is prime.
• Assume keys are integers smaller than \( n^k \)
• Pick \( k+1 \) random values in \{0,...,n-1\}: \( a_0, ..., a_k \)
• Interpret key as \( k \)-digit number base \( n \):
  \[\text{key} = n^{k-1} b_{k-1} + ... + k b_1 + b_0\]
• \( h(\text{key}) = b_0 a_0 + ... + b_{k-1} a_{k-1} + a_k \mod n \)
Example

• Proof of universality:
  – Let key ≠ key’. They differ in some digit i (i.e. \(b_i \neq b_i’\))
    • Assume w.l.o.g. that \(i=0\)
  – Then \(h(\text{key}) - h(\text{key’}) = (b_0 - b_0')a_0 + ... + (b_{k-1} - b_{k-1}')a_{k-1} \mod n\)
  – If \(h(\text{key}) = h(\text{key’})\), then
    \[
    a_0 = -((b_1 - b_1')a_1 + ... + (b_{k-1} - b_{k-1}')a_{k-1})(b_0 - b_0')^{-1} \mod n
    \]
  – Happens for exactly one choice of \(a_0\), prob = 1/n
Universal Hashing and Chaining

• Theorem: S be any subset of K, key an element of K not in S. Then if h is drawn from a universal family of hash functions,

\[ E_{h \leftarrow H} [\text{number of } s \text{ where } h(\text{key}) = h(s)] \leq \frac{|S|}{n} \]
Universal Hashing and Chaining

\[ E_{h \leftarrow H}[\text{number of } s \text{ where } h(\text{key}) = h(s)] \leq \frac{|S|}{n} \]

- **Proof:** Let \( c_s \) be 0 if \( h(\text{key}) \neq h(s) \), 1 otherwise

\[ E_{h \leftarrow H}[c_s] = \Pr_{h \leftarrow H}[h(\text{key}) = h(s)] \leq 1 / n \]

\[ E_{h \leftarrow H}[\text{number of } s \text{ where } h(\text{key}) = h(s)] = E_{h \leftarrow H} \left[ \sum_{s \in S} c_s \right] = \sum_{s \in S} E_{h \leftarrow H}[c_s] \leq \frac{|S|}{n} \]
Universal Hashing and Chaining

- Let $S$ be the set of values stored in hash table
- Key maps to chain of expected size $1 + |S|/n$
- If we keep $|S|/n$ constant (i.e. $|S|/n \leq 1$), all operations constant time
- What happens if $|S|$ gets to large?
  - Double size of array, choose new hash function, and move over all data to new array.
  - Expensive, but amortized constant time.
Running Times

• add: O(1) expected amortized time
• lookup: O(1) expected time
• remove: O(1) expected time