Maps (or Dictionaries)
Goals

- Introduce the Map(or Dictionary) ADT
- Introduce an implementation of the map with a Dynamic Array
• Emphasis on *values* themselves
  – e.g. store names in an AVL tree to quickly lookup club members
  – e.g. store numbers in an AVL tree for a tree sort
• Often, however, we want to associate something else (ie. a value) with the lookup value (ie. a key)
  – e.g. phonebook, dictionary, student roster, etc.
Map ADT (or...dictionary or associative array)

- A Map stores not just values, but *Key-Value pairs*

```java
void put (KT key, VT value)
VT get (KT key)
int containsKey (KT key)
void removeKey (KT key)
```

All comparisons done on the key
All returned values are VT
Can implement with AVLTree, HashTable, DynArr, etc.
void putDynArrayDictionary (struct dyArray *data, KEYTYPE key, VALUETYPE val, comparator compareKey) {
    struct association * ap;
    if (containsKeyDynArrayDictionary(vec, key, compareKey))
        removeKeyDynArrayDictionary (vec, key, compareKey);
    ap = (struct association *) malloc(sizeof(struct association));
    assert(ap != 0);
    ap->key = key;
    ap->value = val;
    addDynArray(vec, ap);
}
int containsMap (DynArr *v, KT key, comparator compare) {
    int i = 0;
    for (i = 0; i < v->size; i++) {
        if (*((struct association *) (v->data[i]))->key, key) == 0) /* found it */
            return 1;
    }
    return 0;
}
• Example Application: Tag Cloud = Concordance + Frequencies

con·cord·ance
kan'kôrdns/
noun
noun: concordance; plural noun: concordances
1. an alphabetical list of the words (esp. the important ones) present in a text, usually with citations of the passages concerned.

• Keys: unique words form the text
• Value: count of each word
Your Turn – Worksheet 36: Dynamic Array Implementation of the Map

- Internally, store **Struct Associations**

- **Put**
  - Ensure that each element in the dictionary has a **unique key**

- **ContainsKey**
  - Loop until you find the ‘key’ and then return true, else false

- **Get**
  - Loop until you find the ‘key’ then return the value

- **RemoveKey**
  - Loop until you find the ‘key’, then remove the entire association
Hash Tables

Concepts
Goals

- Hashing Concepts
• Skip lists and AVL trees reduce the time to perform operations (add, contains, remove) from $O(n)$ to $O(\log n)$

• Can we do better? Can we find a structure that will provide $O(1)$ operations?

• Yes. No. Well, maybe...
Hash Tables

• Hash tables are similar to arrays except...
  – Elements can be indexed by values other than integers **Huh???
  – Multiple values may share an index **What???
Hashing with a Hash Function

Key
- ie. string, url, etc.

Hash function

<table>
<thead>
<tr>
<th>Hash Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Hash to index for storage AND retrieval!
• Spell checker
  – Know all your words before hand
  – Need FAST lookups so you can highlight on the fly
  – Compute an integer index from the string

<table>
<thead>
<tr>
<th>idx</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>hello</td>
</tr>
<tr>
<td>1</td>
<td>pizza</td>
</tr>
<tr>
<td>2</td>
<td>dog</td>
</tr>
<tr>
<td>3</td>
<td>with</td>
</tr>
<tr>
<td>4</td>
<td>front</td>
</tr>
<tr>
<td>5</td>
<td>the</td>
</tr>
<tr>
<td>6</td>
<td>well</td>
</tr>
</tbody>
</table>
• Computing a hash table index is a two-step process:
  1. Transform the value (or key) to an integer (using the hash function)
  2. Map that integer to a valid hash table index (using the mod operator)
Hash Function Goals

• **FAST** (constant time)

• Produce **UNIFORMLY** distributed indices

• **REPEATABLE** (ie. same key always results in same index)
Step 1: Transforming a key to an integer

- **Mapping:** Map (a part of) the key into an integer
  - Example: a letter to its position in the alphabet

- **Folding:** key partitioned into parts which are then combined using efficient operations (such as add, multiply, shift, XOR, etc.)
  - Example: summing the values of each character in a string

<table>
<thead>
<tr>
<th>Key</th>
<th>Mapped chars (position in alphabet)</th>
<th>Folded (+)</th>
</tr>
</thead>
<tbody>
<tr>
<td>eat</td>
<td>5 + 1 + 20</td>
<td>26</td>
</tr>
</tbody>
</table>
Step 1: Transforming a key to an integer

• Shifting: can account for position of characters

Shifted by position in the word (right to left): 0th letter shifted left 0, first letter shifted left 1, etc.

<table>
<thead>
<tr>
<th>Key</th>
<th>Mapped chars (pos in alpha)</th>
<th>Folded (+)</th>
<th>Shifted and Folded</th>
</tr>
</thead>
<tbody>
<tr>
<td>eat</td>
<td>5 + 1 + 20</td>
<td>26</td>
<td>20 + 2 + 20 = 42</td>
</tr>
<tr>
<td>ate</td>
<td>1 + 20 + 5</td>
<td>26</td>
<td>4 + 40 + 5 = 49</td>
</tr>
<tr>
<td>tea</td>
<td>20 + 5 + 1</td>
<td>26</td>
<td>80 + 10 + 1 = 91</td>
</tr>
</tbody>
</table>
Step 2: Mapping to a Valid Index

- Use modulus operator (%) with table size:
  - Example: \( \text{idx} = \text{hash(val)} \mod \text{size} \);

- Use only positive arithmetic or take absolute value

- To get a good distribution of indices, prime numbers make the best table sizes:
  - Example: if you have 1000 elements, a table size of 997 or 1009 is preferable
Hashing: Why do it??

- Assuming
  - Hash function can be computed in constant time
  - computed indices are equally distributed over the table

- Allows for $O(1)$ time bag/map operations!

- But...we have to be able to deal with collisions
A collision occurs when two values hash to the same index.

How do we deal with this????
Hash Tables

Buckets/Chaining
Hash Tables: Resolving Collisions

There are two general approaches to resolving collisions:

1. Open address hashing: if a spot is full, probe for next empty spot

2. Chaining (or buckets): keep a collection at each table entry
Resolving Collisions: **Chaining**

Maintain a collection (typically a Map ADT) at each table entry:

Each collection is called a ‘chain’ (sometimes referred to as a bucket)
Hash Table Implementation: Initialization

```c
struct HashTable {
    struct LinkedList **table; /* Hash table \rightarrow Array of Lists. */
    int capacity;
    int count;
}

void initHashTable(struct HashTable *ht, int cap) {
    int i;

    ht->capacity = cap;
    ht->count   = 0;
    ht->table = malloc(ht->capacity * sizeof(struct LinkedList *));
    assert(ht->table != 0);
    for(i = 0; i < ht->capacity; i++) ht->table[i] = createLinkedList();
}
```
void addHashTable(struct HashTable *ht, TYPE val) {
    /* Compute hash table index. */
    int idx = hash(val) % ht->capacity;

    /* Add to the chain. */
    addList(ht->table[idx], val);
    ht->count++;
}

Hash Table: **Contains & Remove**

- **Contains**: find correct bucket using the hash function, then check to see if element is in the linked list

- **Remove**: if element is in the table (e.g. contains() returns true), remove it from the linked list and decrement the count
Hash Table Size

• How large should your table be?
• How does it affect performance?
Hash Table Size

• Load factor:

- Load factor represents **average number of elements in each bucket**
- For chaining, load factor can be greater than 1

• To maintain good performance: if load factor becomes larger than some fixed limit (say, 8) → double table size

\[ \lambda = \frac{n}{m} \]

\# of elements (count)

Load factor

Capacity of table
• Load factor:
  
  \[ \frac{\lambda}{m} = n \]
  
  The average number of links traversed in successful searches, \( S \), and unsuccessful searches, \( U \), is
  
  \[ S \approx \frac{\lambda}{2} \quad \text{and} \quad U \approx \lambda \]
  
  If load factor becomes larger than some fixed limit (say, 8) \( \rightarrow \) double table size
• Assuming:
  – Time to compute hash function is constant
  – Chaining uses a linked list
  – Worst case analysis ➔ All values hash to same position
  – Best case analysis ➔ Hash function uniformly distributes the values and no collisions

• Contains operation:
  – Worst case for chaining ➔ \( O(n) \)
  – Best case for chaining ➔ \( O(1) \)
Hash Tables With Chaining: Average Case (Unsuccessful)

- Assume hash function distributes elements uniformly (a BIG if)
- And we have **collisions**
- Unsuccessful search:
  - Any key, K, is equally likely to hash to any of the m slots
  - Ave. time to search for a *non-existent* key is average time to search to end of a chain = \( \lambda \)
  - Total Time: \( O(1 + \lambda) \)
    [Time to compute hash function + \( \lambda \)]
Hash Tables With Chaining: **Average Case (Successful)**

- Assume hash function distributes elements uniformly *(a BIG if)*
- And we have **collisions**
- **Successful** search:
  - Any key, K, is equally likely to be any of the n keys stored in the table
  - Assume values are added to end of a chain
  - Expected number of elements examined upon success is 1 more than number elements examined when the sought for element was inserted
  - Therefore, take average over the n items in the table of 1 + expected length of the list to which the i’th element is added:

\[
\frac{1}{n} \sum_{i=1}^{n} \left(1 + \frac{i - 1}{m}\right)
\]

- # elements examined when i’th was inserted
- Ave
- One more
Hash Tables with Chaining: Average Case (successful)

\[
\frac{1}{n} \sum_{i=1}^{n} \left(1 + \frac{i-1}{m}\right) = 1 + \left(\frac{1}{nm}\right) \left(\sum_{i=1}^{n} (i-1)\right)
\]

\[
= 1 + \left(\frac{1}{nm}\right) \left(\frac{n(n-1)}{2}\right)
\]

\[
= 1 + \frac{(n-1)}{2m}
\]

\[
= 1 + \frac{\lambda - 1}{2m}
\]

Time for successful search, **including** time for hashing: \(O(2 + \frac{\lambda}{2} - \frac{1}{2m}) = O(1+\lambda)\)

So, if number of slots is at least proportional to number of elements in the table, then \(n = O(m)\) and consequently, \(\lambda = \frac{n}{m} = O(m)/m = O(1)\)
• So, we want to keep the load factor relatively small

• We monitor (e.g. check) the load factor:
  – Resize table (doubling its size) if load factor is larger than some fixed limit (e.g., 8)

• Only improves things *IF* hash function distributes values uniformly
  – How do we handle a resize?
Design Decisions

• Implement the Map interface to store values with keys (ie. implement a dictionary)
• Rather than store linked lists, build the linked lists directly
  – Link **hashTable;
Concordance + Frequencies

- Use a HashMap in your assignment
- Lookup key [word], return value [frequency count]
  (or, return a ptr so you can update it!)

<table>
<thead>
<tr>
<th>idx</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

help

run 7
help 2
ride 4

3

means that help has occurred 4 times in this selection of text!
Your Turn

• Worksheet 38: Hash Tables using Buckets
Perfect Hash Function

- A perfect hash function is one where there are no collisions
- Minimally Perfect: No collisions AND table size = # of elements
  - Only possible when know all your keys in advance of building the table
### Minimally Perfect Hash Function

Position of 3rd letter (starting at left, index 0), mod 6

<table>
<thead>
<tr>
<th>Name</th>
<th>Position</th>
<th>Calculation</th>
<th>Mod 6</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alfred</td>
<td>f</td>
<td>f = 5 % 6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Alessia</td>
<td>e</td>
<td>e = 4 % 6</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Amina</td>
<td>i</td>
<td>i = 8 % 6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Amy</td>
<td>y</td>
<td>y = 24 % 6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Andy</td>
<td>d</td>
<td>d = 3 % 6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Anne</td>
<td>n</td>
<td>n = 13 % 6</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Position</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Amy</td>
</tr>
<tr>
<td>1</td>
<td>Anne</td>
</tr>
<tr>
<td>2</td>
<td>Amina</td>
</tr>
<tr>
<td>3</td>
<td>Andy</td>
</tr>
<tr>
<td>4</td>
<td>Alessia</td>
</tr>
<tr>
<td>5</td>
<td>Alfred</td>
</tr>
</tbody>
</table>
Hash Tables
Open Address Hashing
Goals

• Open Address Hashing
Hash Tables: Resolving Collisions

There are two general approaches to resolving collisions:

1. Open address hashing: if a spot is full, probe for next empty spot

2. Chaining (or buckets): keep a collection at each table entry
Open Address Hashing

- All values are stored in an array
- Hash value is used to find initial index
- If that position is filled, the next position is examined, then the next, and so on until you find the element OR an empty position is found
- The process of looking for an empty position is termed probing, specifically linear probing when we look to the next element
Open Address Hashing: Example

Eight element table using Amy’s hash function (alphabet position of the 3rd letter of the name -1):

Already added: Amina, Andy, Alessia, Alfred, and Aspen

<table>
<thead>
<tr>
<th>Amina</th>
<th>Andy</th>
<th>Alessia</th>
<th>Alfred</th>
<th>Aspen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-aiqy</td>
<td>1-bjrz</td>
<td>2-cks</td>
<td>3-dlt</td>
<td>4-emu</td>
</tr>
<tr>
<td>5-fnv</td>
<td>6-gow</td>
<td>7-hpx</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** We’ve shown where each letter of the alphabet maps to for simplicity here (given a table size of 8) ...so you don’t have to calculate it!

e.g. Y is the 25th letter (we use 0 index, so the integer value is 24) and 24 mod 8 is 0
Open Address Hashing: Example

Eight element table using Amy’s hash function (alphabet position of the 3\textsuperscript{rd} letter of the name – 1):

Already added: Amina, Andy, Alessia, Alfred, and Aspen

<table>
<thead>
<tr>
<th>Amina</th>
<th>Andy</th>
<th>Alessia</th>
<th>Alfred</th>
<th>Aspen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-aiqy</td>
<td>1-bjrz</td>
<td>2-cks</td>
<td>3-dlt</td>
<td>4-emu</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5-fnv</td>
<td>6-gow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7-hpx</td>
</tr>
</tbody>
</table>

Note: We’ve shown where each letter of the alphabet maps to for simplicity here (given a table size of 8) ...so you don’t have to calculate it!

e.g. Y is the 25\textsuperscript{th} letter (we use 0 index, so the integer value is 24) and 24 mod 8 is 0
Now we need to add: Aimee

Add: Aimee

Placed here

The hashed index position (4) is filled by Alessia: so we probe to find next free location
Suppose **Anne** wants to join:

Add: **Anne**

The hashed index position **(5)** is filled by **Alfred**:

- Probe to find next free location ➔ **what happens when we reach the end of the array**
Open Address Hashing: Adding (cont.)

Suppose Anne wants to join:

Add: Anne

The hashed index position (5) is filled by Alfred:

- Probe to find next free location
- When we get to end of array, wrap around to the beginning
- Eventually, find position at index 1 open
Open Address Hashing: Contains

- Hash to find initial index
- probe forward until
  - value is found, (return 1) OR
  - empty location is found (return 0)

<table>
<thead>
<tr>
<th>Amina</th>
<th>Andy</th>
<th>Alessia</th>
<th>Alfred</th>
<th>Aimee</th>
<th>Aspen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-aiqy</td>
<td>1-bjrz</td>
<td>2-cks</td>
<td>3-dlt</td>
<td>4-emu</td>
<td>5-fnv</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6-gow</td>
<td>7-hpx</td>
</tr>
</tbody>
</table>

- Notice that search time is not uniform
Open Address Hashing: Remove

- Remove is tricky
- What happens if we delete Anne, then search for Alan?

Remove: Anne

Find: Alan

Hashes to

Probing finds null entry → Alan not found
• Simple solution: Don’t allow removal (e.g. words don’t get removed from a spell checker!)
• Alternative solution: replace removed item with “tombstone”
  – Special value that marks deleted entry
  – Can be replaced when adding new entry
  – But doesn’t halt search during contains or remove

Find: Alan

<table>
<thead>
<tr>
<th>Amina</th>
<th><em>TS</em></th>
<th>Alan</th>
<th>Alessia</th>
<th>Alfred</th>
<th>Aimee</th>
<th>Aspen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-aiqy</td>
<td>1-bjrz</td>
<td>2-cks</td>
<td>3-dlt</td>
<td>4-emu</td>
<td>5-fnv</td>
<td>6-gow</td>
</tr>
</tbody>
</table>

Hashes to
Probing skips tombstone → Alan found
Hash Table Size: Load Factor

Load factor:

\[ \lambda = \frac{n}{m} \]

- Represents the portion of the tables that is filled
- For open address hashing, load factor is between 0 and 1 (often somewhere between 0.5 and 0.75)

Want the load factor to remain small in order to avoid collisions – space/speed tradeoff again!
Hash Tables: Algorithmic Complexity

• Assumptions:
  – Time to compute hash function is constant
  – Worst case analysis → All values hash to same position
  – Best case analysis → Hash function uniformly distributes the values and there are no collisions

• Find element operation:
  – Worst case for open addressing → O(\( n \))
  – Best case for open addressing → O(1)
Hash Tables: **Average Case**

- What about average case for successful, S, and unsuccessful searches, U?

\[
S = \frac{1}{\lambda} \ln \frac{1}{1 - \lambda} + \frac{1}{\lambda} \\
U = \frac{1}{1 - \lambda}
\]

<table>
<thead>
<tr>
<th>(\lambda)</th>
<th>(S)</th>
<th>(U)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>(~3.387)</td>
<td>2</td>
</tr>
<tr>
<td>0.9</td>
<td>(~3.7)</td>
<td>10</td>
</tr>
</tbody>
</table>

- If \(\lambda\) is constant, average case is \(O(1)\), but want to keep \(\lambda\) small
Clustering

• Assuming uniform distribution of hash values, what’s the probability that the next value will end up in index 6? in index 2? in index 1?

<table>
<thead>
<tr>
<th>Amina</th>
<th>Andy</th>
<th>Alessia</th>
<th>Alfred</th>
<th>Aspen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-aiqy</td>
<td>1-bjrz</td>
<td>2-cks</td>
<td>3-dlt</td>
<td>4-emu</td>
</tr>
</tbody>
</table>

3/8 1/8 4/8

• As load factor gets larger, the tendency to cluster increases, resulting in longer search times upon collision
Double Hashing

• Rather than use a linear probe (ie. looking at successive locations)...
  – Use a second hash function to determine the probe step
• Helps to reduce clustering
Large Load Factor: What to do?

• Common solution: When load factor becomes too large (say, bigger than 0.75) Reorganize

• Create new table with twice the number of positions

• Copy each element, rehashing using the new table size, placing elements in new table

• Delete the old table
Hashing in Practice

• Need to find good hash function → uniformly distributes keys to all indices

• Open address hashing:
  – Need to tell if a position is empty or not
  – One solution → store only pointers & check for null (== 0)
Tradeoffs

- Chaining takes more memory (ptrs)
- Chaining often preferred when have to deal with deletions
Your Turn

• Complete Worksheet #37: Open Address Hashing