COMP2521 24T1
Hash Tables

Kevin Luxa

cs2521@cse.unsw.edu.au

associative arrays
hash tables
hashing
collision resolution
A commonly desired abstraction in computer science and in the real world is the ability to map one kind of data to another, in other words, map keys to values.

Examples:
- Map words to definitions
- Map student numbers to names
- Map people to favourite colors
An **associative array** is an abstract data type that stores key-value pairs, where keys are unique.

It supports the following operations:

- **insert**
  - insert a key-value pair

- **lookup**
  - given a key, return its associated value

- **delete**
  - given a key, delete its key-value pair

Note:

Associative arrays are also called **maps**, symbol tables, or **dictionaries**.
How to implement an associative array?

unordered array

ordered array

balanced binary search tree
### Motivation

**Assiociative Arrays**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td>[5]</td>
</tr>
<tr>
<td>jas green</td>
<td>andrew red</td>
<td>sasha purple</td>
<td>jake yellow</td>
<td>kevin blue</td>
<td>hayden red</td>
</tr>
</tbody>
</table>

**Performance?**
- Insert: $O(n)$
- Lookup: $O(n)$
- Delete: $O(n)$
Motivation

Associative Arrays

ordered array

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td>[5]</td>
</tr>
<tr>
<td>andrew red</td>
<td>hayden red</td>
<td>jake yellow</td>
<td>jas green</td>
<td>kevin blue</td>
<td>sasha purple</td>
</tr>
</tbody>
</table>

Performance?

Insert: \(O(n)\)

Lookup: \(O(\log n)\)

Delete: \(O(n)\)
Motivation

Associative Arrays

balanced binary search tree

- jake: yellow
- hayden: red
- kevin: blue
- andrew: red
- jas: green
- sasha: purple

Performance?
- Insert: $O(\log n)$
- Lookup: $O(\log n)$
- Delete: $O(\log n)$
How to implement an associative array?

unordered array

ordered array

balanced binary search tree

hash table
A hash table is a data structure that implements an associative array.

It uses an array to store key-value pairs, and a hash function that, given a key, computes an index into the array where the associated value can be found.

A good hash table implementation has an average performance of $O(1)$ for insertion, lookup and deletion!
Hash Tables

Motivation

Hashing

Collision Resolution

Design Issues

**key** = “jas”  ➔  hash function

*index* = 4

<table>
<thead>
<tr>
<th>NO ITEM</th>
<th>jake yellow</th>
<th>NO ITEM</th>
<th>sasha purple</th>
<th>jas green</th>
<th>...</th>
<th>NO ITEM</th>
<th>andrew red</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td></td>
<td>[N - 2]</td>
<td>[N - 1]</td>
</tr>
</tbody>
</table>
/** Creates a new hash table */
HashTable HashTableNew(void);

/** Frees all memory allocated to the hash table */
void HashTableFree(HashTable ht);

/** Inserts a key-value pair into the hash table
   If the key already exists, replaces the value */
void HashTableInsert(HashTable ht, Key key, Value value);

/** Returns true if the hash table contains the given key,
   and false otherwise */
bool HashTableContains(HashTable ht, Key key);

/** Returns the value associated with the given key
   Assumes that the key exists */
Value HashTableGet(HashTable ht, Key key);

/** Deletes the key-value pair associated with the given key */
void HashTableDelete(HashTable ht, Key key);

/** Returns the number of key-value pairs in the hash table */
int HashTableSize(HashTable ht);
HashTable ht = HashTableNew();

HashTableInsert(ht, "jas", "green");
HashTableInsert(ht, "andrew", "red");
HashTableInsert(ht, "sasha", "purple");
HashTableInsert(ht, "jake", "yellow");

printf("jas' fav colour is %s\n", HashTableGet(ht, "jas")); // green

HashTableInsert(ht, "jas", "orange");
printf("jas' fav colour is %s\n", HashTableGet(ht, "jas")); // orange

HashTableDelete(ht, "jas");
if (!HashTableContains(ht, "jas")) {
    printf("jas has no fav colour\n");
}

HashTableFree(ht);
Hashing is the process of mapping data of arbitrary size to fixed-size values using a hash function.

**Applications:**
- Hash tables
- Password storage and verification
- Verifying integrity of messages and files
- Database indexing
  - ...many others
A hash function:

• Maps a key to an index in the range $[0, N - 1]$
  • where $N$ is the size of the array
• Must be cheap to compute
• Is deterministic
  • Given the same key, will always return the same index
• Ideally, maps keys uniformly over the range of indices
Basic mechanism of hash functions:

```c
int hash(Key key, int N) {
    int val = convert key to 32-bit int
    return val % N;
}
```
Simple hash function for ints:

```c
int hash(int key, int N) {
    return key % N;
}
```

Simple hash function for strings:

```c
int hash(char *key, int N) {
    int sum = 0;
    for (int i = 0; key[i] != '\'0'; i++) {
        sum += key[i];
    }
    return sum % N;
}
```
More robust hash function for strings:

```c
int hash(char *key, int N) {
    int h = 0, a = 31415, b = 21783;
    for (char *c = key; *c != '\0'; c++) {
        a = a * b % (N - 1);
        h = (a * h + *c) % N;
    }
    return h;
}
```
A real hash function (from PostgreSQL DBMS)...

```c
int hash_any(unsigned char *k, register int keylen, int N) {
    register uint32 a, b, c, len;

    // set up internal state
    len = keylen;
    a = b = 0x9e3779b9;
    c = 3923095;

    // handle most of the key, in 12-char chunks
    while (len >= 12) {
        mix(a, b, c);
        k += 12; len -= 12;
    }

    // collect any data from remaining bytes into a,b,c
    mix(a, b, c);
    return c % N;
}
```
...where mix is defined as:

```c
#define mix(a, b, c) \
{
    a -= b; a -= c; a ^= (c >> 13); \ 
    b -= c; b -= a; b ^= (a << 8); \ 
    c -= a; c -= b; c ^= (a >> 13); \ 
    a -= b; a -= c; a ^= (b >> 12); \ 
    b -= c; b -= a; b ^= (a << 16); \ 
    c -= a; c -= b; c ^= (b >> 5); \ 
    a -= b; a -= c; a ^= (c >> 3); \ 
    b -= c; b -= a; b ^= (a << 10); \ 
    c -= a; c -= b; c ^= (b >> 15); \ 
}
```
Given a hash table with 11 slots and the hash function $h(k) = k \mod 11$, insert the following keys:

4 8 15 16 23 42
Given a hash table with 11 slots and the hash function $h(k) = k \% 11$, insert the following keys:

$4 \ 8 \ 15 \ 16 \ 23 \ 42$
Given a hash table with 11 slots and the hash function $h(k) = k \% 11$, insert the following keys:

4  8  15  16  23  42

$h(4) = 4$
Given a hash table with 11 slots and the hash function $h(k) = k \% 11$, insert the following keys:

$$4 \ 8 \ 15 \ 16 \ 23 \ 42$$

$h(4) = 4$

```
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]

[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]

[ ] [ ] [ ] [ ] [4] [ ] [ ] [ ] [ ] [ ]
```
Given a hash table with 11 slots and the hash function $h(k) = k \% 11$, insert the following keys:

4 8 15 16 23 42

<table>
<thead>
<tr>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
<th>[10]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Given a hash table with 11 slots and the hash function $h(k) = k \% 11$, insert the following keys:

4  8  15  16  23  42

$h(8) = 8$
Given a hash table with 11 slots and the hash function $h(k) = k \% 11$, insert the following keys:

$$4 \quad 8 \quad 15 \quad 16 \quad 23 \quad 42$$

$h(8) = 8$
Given a hash table with 11 slots and the hash function $h(k) = k \% 11$, insert the following keys:

4 8 15 16 23 42
Given a hash table with 11 slots and the hash function $h(k) = k \% 11$, insert the following keys:

$$4 \ 8 \ 15 \ 16 \ 23 \ 42$$

$h(15) = 4$

index 4 already contains an item ⇒ collision!
Given a hash table with 11 slots and the hash function $h(k) = k \% 11$, insert the following keys:

$$4 \quad 8 \quad 15 \quad 16 \quad 23 \quad 42$$

$h(15) = 4$

Index 4 already contains an item ⇒ collision!
Often, the range of possible key values is \textit{much} larger than the range of indices \([0, N - 1]\), so collisions are inevitable.

A \textbf{hash collision} occurs when for two keys \(x\) and \(y\),
\[x \neq y, \text{ but } h(x) = h(y).\]

A hash table must have a method for resolving collisions.
Collision resolution methods:

- **Separate chaining**
  - Each array slot contains a list of the items hashed to that index
  - Allows multiple items in one slot

- **Linear probing**
  - Check rest of array slots consecutively until an empty slot is found

- **Double hashing**
  - Instead of checking slots consecutively, use an increment which is determined by a secondary hash
Important statistic: **load factor** ($\alpha$)

- Ratio of items to slots; $\alpha = M/N$
- Useful when analysing collision resolution methods
Resolve collisions by having multiple items per array slot.

Each array slot contains a linked list of items that are hashed to that index.
Given a hash table with 7 slots that uses separate chaining and the hash function \( h(k) = k \% 7 \), insert the following keys:

\[
23 \ 4 \ 16 \ 42 \ 8 \ 15
\]
Given a hash table with 7 slots that uses separate chaining and the hash function $h(k) = k \mod 7$, insert the following keys:

23  4  16  42  8  15
Given a hash table with 7 slots that uses separate chaining and the hash function \( h(k) = k \mod 7 \), insert the following keys:

\[
\begin{array}{c}
23 & 4 & 16 & 42 & 8 & 15 \\
\end{array}
\]

\[
h(23) = 23 \mod 7 = 2
\]
Given a hash table with 7 slots that uses separate chaining and the hash function \( h(k) = k \% 7 \), insert the following keys:

\[
23 \ 4 \ 16 \ 42 \ 8 \ 15
\]

\[h(23) = 23 \% 7 = 2\]
Given a hash table with 7 slots that uses separate chaining and the hash function $h(k) = k \% 7$, insert the following keys:

23 4 16 42 8 15
Given a hash table with 7 slots that uses separate chaining and the hash function $h(k) = k \% 7$, insert the following keys:

$$23\ 4\ 16\ 42\ 8\ 15$$

$h(4) = 4 \% 7 = 4$
Given a hash table with 7 slots that uses separate chaining and the hash function \( h(k) = k \% 7 \), insert the following keys:

\[
23 \ 4 \ 16 \ 42 \ 8 \ 15
\]

\[
h(4) = 4 \% 7 = 4
\]
Given a hash table with 7 slots that uses separate chaining and the hash function \( h(k) = k \mod 7 \), insert the following keys:

23 4 16 42 8 15
Given a hash table with 7 slots that uses separate chaining and the hash function $h(k) = k \% 7$, insert the following keys:

23 4 16 42 8 15

$h(16) = 16 \% 7 = 2$
Given a hash table with 7 slots that uses separate chaining and the hash function $h(k) = k \mod 7$, insert the following keys:

\begin{align*}
23 & \quad 4 & \quad 16 & \quad 42 & \quad 8 & \quad 15
\end{align*}

$h(16) = 16 \mod 7 = 2$
Given a hash table with 7 slots that uses separate chaining and the hash function $h(k) = k \mod 7$, insert the following keys:

23  4  16  42  8  15

![Hash table diagram]

Where:

- $h(23) = 23 \mod 7 = 2$
- $h(4) = 4 \mod 7 = 4$
- $h(16) = 16 \mod 7 = 2$
- $h(42) = 42 \mod 7 = 0$
- $h(8) = 8 \mod 7 = 1$
- $h(15) = 15 \mod 7 = 1$
Given a hash table with 7 slots that uses separate chaining and the hash function $h(k) = k \mod 7$, insert the following keys:

23 4 16 42 8 15

$h(42) = 42 \mod 7 = 0$
Given a hash table with 7 slots that uses separate chaining and the hash function \( h(k) = k \% 7 \), insert the following keys:

\[
23 \ 4 \ 16 \ 42 \ 8 \ 15
\]

\( h(42) = 42 \% 7 = 0 \)
Given a hash table with 7 slots that uses separate chaining and the hash function \( h(k) = k \mod 7 \), insert the following keys:

\[
\begin{align*}
23 & \quad 4 \quad 16 \quad 42 \quad 8 \quad 15
\end{align*}
\]
Given a hash table with 7 slots that uses separate chaining and the hash function $h(k) = k \mod 7$, insert the following keys:

$$23 \ 4 \ 16 \ 42 \ 8 \ 15$$

$h(8) = 8 \mod 7 = 1$
Given a hash table with 7 slots that uses separate chaining and the hash function $h(k) = k \mod 7$, insert the following keys:

$$23 \ 4 \ 16 \ 42 \ 8 \ 15$$

$$h(8) = 8 \mod 7 = 1$$

![Diagram of hash table with separate chaining and inserted keys 42, 8, 23, 4, 16.]}
Given a hash table with 7 slots that uses separate chaining and the hash function $h(k) = k \mod 7$, insert the following keys:

23  4  16  42  8  15

Diagram:

```
[0] [1] [2] [3] [4] [5] [6]
   └───┴───┴───┴───┴───┘
      42  8  23  4
          └───┘
              16
```
Given a hash table with 7 slots that uses separate chaining and the hash function $h(k) = k \mod 7$, insert the following keys:

23 4 16 42 8 15

$h(15) = 15 \mod 7 = 1$
Separate Chaining
Example

Given a hash table with 7 slots that uses separate chaining and the hash function \( h(k) = k \% 7 \), insert the following keys:

\[
23 \quad 4 \quad 16 \quad 42 \quad 8 \quad 15
\]

\( h(15) = 15 \% 7 = 1 \)
Assuming integer keys and values:

```c
struct hashTable {
    struct node **slots; // array of lists
    int numSlots;
    int numItems;
};

struct node {
    int key;
    int value;
    struct node *next;
};
```
Separate Chaining
Implementation

HashTable HashTableNew(void) {
    HashTable ht = malloc(sizeof(*ht));
    ht->slots = calloc(INITIAL_NUM_SLOTS, sizeof(struct node *));
    ht->numSlots = INITIAL_NUM_SLOTS;
    ht->numItems = 0;
    return ht;
}
Separate Chaining
Implementation

void HashTableInsert(HashTable ht, int key, int value) {
    if (/* load factor exceeds threshold */) {
        // resize hash table
    }

    int i = hash(key, ht->numSlots);
    ht->slots[i] = doInsert(ht, ht->slots[i], key, value);
}

struct node *doInsert(HashTable ht, struct node *list, int key, int value) {
    if (list == NULL) {
        ht->numItems++;
        return newNode(key, value);
    } else if (list->key == key) {
        list->value = value; // replace value
    } else {
        list->next = doInsert(ht, list->next, key, value);
    }
    return list;
}
bool HashTableContains(HashTable ht, int key) {
    int i = hash(key, ht->numSlots);
    struct node *curr = ht->slots[i];
    while (curr != NULL) {
        if (curr->key == key) {
            return true;
        }
        curr = curr->next;
    }
    return false;
}
Motivation

Hash Tables

Hashing

Collision Resolution

Separate Chaining

Example

Implementation

Analysis

Linear probing

Double hashing

Design Issues

Separate Chaining

Implementation

```c
int HashTableGet(HashTable ht, int key) {
    int i = hash(key, ht->numSlots);
    struct node *curr = ht->slots[i];
    while (curr != NULL) {
        if (curr->key == key) {
            return curr->value;
        }
        curr = curr->next;
    }
    error;
}
```
void HashTableDelete(HashTable ht, int key) {
    int i = hash(key, ht->numSlots);
    ht->slots[i] = doDelete(ht, ht->slots[i], key);
}

struct node *doDelete(HashTable ht, struct node *list, int key) {
    if (list == NULL) {
        return NULL;
    } else if (list->key == key) {
        struct node *newHead = list->next;
        free(list);
        ht->numItems--;
        return newHead;
    } else {
        list->next = doDelete(ht, list->next, key);
        return list;
    }
}
Cost analysis:

- \( N \) array slots, \( M \) items
- Average list length \( L = M/N \)
- Best case: Items evenly distributed, so maximum list length is \( \lceil M/N \rceil \)
  - Cost of insert/lookup/delete: \( O(M/N) \)
- Worst case: One list of length \( M \)
  - Cost of insert/lookup/delete: \( O(M) \)

Average costs:

- If good hash and \( \alpha \leq 1 \), cost is \( O(1) \)
- If good hash and \( \alpha > 1 \), cost is \( O(M/N) \)
  - To avoid degrading performance, hash table should be resized when \( \alpha \approx 1 \)
Resolve collisions by finding a new slot for the item
- Each array slot stores a single item (unlike separate chaining)
- On a hash collision, try next slot, then next, until an empty slot is found
- Insert item into empty slot

Example: $h(k) = k \% 10$

insert $k=22$

$ h(22) = 2 $  

insert $k=14$

$ h(14) = 4 $  

insert $k=8$

$ h(8) = 8 $
Assuming integer keys and values:

```c
struct hashTable {
    struct slot *slots;
    int numSlots;
    int numItems;
};

struct slot {
    int key;
    int value;
    bool empty;
};
```
Hash Table HashTableNew(void) {
    HashTable ht = malloc(sizeof(*ht));
    ht->slots = malloc(INITIAL_CAPACITY * sizeof(struct slot));
    for (int i = 0; i < ht->numSlots; i++) {
        ht->slots[i].empty = true;
    }
    ht->numSlots = INITIAL_CAPACITY;
    ht->numItems = 0;
    return ht;
}
Process for insertion:

1. If load factor exceeds threshold, resize
   - Whether to do this or not is a design decision
2. Hash given key to get an index
3. Starting from this index, find first slot that either:
   - Contains the given key, or
   - Is empty
4. If the slot is empty, store the key and value, otherwise just replace the value

This will be a task in the week 9 lab exercise!
Process for lookup:

1. Hash given key to get an index
2. Starting from this index, find first slot that either:
   - Contains the given key, or
   - Is empty
3. If the slot contains the given key, return the value, otherwise error
   - This is a design decision
int HashTableGet(HashTable ht, int key) {
    int i = hash(key, ht->numSlots);
    for (int j = 0; j < ht->numSlots; j++) {
        if (ht->slots[i].empty) break;
        if (ht->slots[i].key == key) {
            return ht->slots[i].value;
        }
        i = (i + 1) % ht->numSlots;
    }
    error;
}
How to delete an item?

We can’t simply remove the item and be done, as this can break the probe paths for other items, for example:

\[ h(k) = k \% 10 \]

<table>
<thead>
<tr>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Item</td>
<td>11</td>
<td>No Item</td>
<td>No Item</td>
<td>24</td>
<td>5</td>
<td>14</td>
<td>4</td>
<td>18</td>
<td>No Item</td>
</tr>
</tbody>
</table>

Deleting 24 (incorrectly)

<table>
<thead>
<tr>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Item</td>
<td>11</td>
<td>No Item</td>
<td>No Item</td>
<td>No Item</td>
<td>5</td>
<td>14</td>
<td>4</td>
<td>18</td>
<td>No Item</td>
</tr>
</tbody>
</table>

Probe path for 14 and 4 is broken!
Two primary methods for deletion:

1. **Backshift**
   - Remove and re-insert all items between the deleted item and the next empty slot

2. **Tombstone**
   - Replace the deleted item with a “deleted” marker (AKA a tombstone) that:
     - Is treated as empty during insertion
     - Is treated as occupied during lookup
Using the backshift method, delete 24 from this hash table:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Item</td>
<td>11</td>
<td>No Item</td>
<td>No Item</td>
<td>24</td>
<td>5</td>
<td>14</td>
<td>4</td>
<td>18</td>
<td>No Item</td>
</tr>
</tbody>
</table>
# Linear Probing

**Backshift Deletion - Example**

## Step 1: Remove 24

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Item</td>
<td>11</td>
<td>No Item</td>
<td>No Item</td>
<td>No Item</td>
<td>5</td>
<td>14</td>
<td>4</td>
<td>18</td>
<td>No Item</td>
</tr>
</tbody>
</table>

## Step 2: Re-insert 5

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Item</td>
<td>11</td>
<td>No Item</td>
<td>No Item</td>
<td>No Item</td>
<td>5</td>
<td>14</td>
<td>4</td>
<td>18</td>
<td>No Item</td>
</tr>
</tbody>
</table>

## Step 3: Re-insert 14

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Item</td>
<td>11</td>
<td>No Item</td>
<td>No Item</td>
<td>No Item</td>
<td>14</td>
<td>5</td>
<td>No Item</td>
<td>4</td>
<td>18</td>
</tr>
</tbody>
</table>
Linear Probing

Backshift Deletion - Example

Step 4: Re-insert 4

<table>
<thead>
<tr>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Item</td>
<td>11</td>
<td>No Item</td>
<td>No Item</td>
<td>14</td>
<td>5</td>
<td>4</td>
<td>No Item</td>
<td>18</td>
<td>No Item</td>
</tr>
</tbody>
</table>

Step 5: Re-insert 18

<table>
<thead>
<tr>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Item</td>
<td>11</td>
<td>No Item</td>
<td>No Item</td>
<td>14</td>
<td>5</td>
<td>4</td>
<td>No Item</td>
<td>18</td>
<td>No Item</td>
</tr>
</tbody>
</table>

This will be a task in the week 9 lab exercise!
Using the tombstone method, delete 14 from this hash table:

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td>[5]</td>
<td>[6]</td>
<td>[7]</td>
<td>[8]</td>
</tr>
<tr>
<td>No Item</td>
<td>11</td>
<td>No Item</td>
<td>No Item</td>
<td>24</td>
<td>5</td>
<td>14</td>
<td>4</td>
<td>18</td>
</tr>
</tbody>
</table>
After deleting 14:

<table>
<thead>
<tr>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Item</td>
<td>11</td>
<td>No Item</td>
<td>No Item</td>
<td>24</td>
<td>5</td>
<td>DEL</td>
<td>4</td>
<td>18</td>
<td>No Item</td>
</tr>
</tbody>
</table>

Search for 4:

\[ h(4) = 4 \]
Insert 15:

\[ h(15) = 5 \]

Result:
Backshift method:
- Moves items closer to their hash index
  - Thus reducing the length of their probe path
- Deletion becomes more expensive

Tombstone method:
- Fast
- But does not reduce probe path length
- Large number of deletions will cause tombstones to build up
Problem with linear probing: clustering

- Items tend to cluster together into long runs
  - i.e., long contiguous regions that don’t contain empty slots
- Long runs are a problem:
  - Insertions must travel to the end of a run
  - Lookups of non-existent keys must travel to the end of a run

Causes of clustering:

- The longer a run becomes, the more likely it is to accrue additional items
- Two long runs can be connected together into an even longer run due to the insertion of an item between them
Example \( h(k) = k \mod 15 \):

Insert 1, 2, 3, 17, 18

Insert 7, 9, 22, 24, 37, 11

What happens if we insert/search for 8? How about if we insert 6?
Analysis of lookup:

- Hash function is $O(1)$
- Subsequent cost depends on probe path length
  - Affected by load factor $\alpha = M/N$
  - Analysed by Donald Knuth in 1963
  - Average cost for successful search $= \frac{1}{2} \left(1 + \frac{1}{1-\alpha} \right)$
  - Average cost for unsuccessful search $= \frac{1}{2} \left(1 + \frac{1}{(1-\alpha)^2} \right)$

Example costs (assuming large hash table):

<table>
<thead>
<tr>
<th>load factor ($\alpha$)</th>
<th>0.50</th>
<th>0.67</th>
<th>0.75</th>
<th>0.90</th>
</tr>
</thead>
<tbody>
<tr>
<td>search hit</td>
<td>1.5</td>
<td>2.0</td>
<td>3.0</td>
<td>5.5</td>
</tr>
<tr>
<td>search miss</td>
<td>2.5</td>
<td>5.0</td>
<td>8.5</td>
<td>55.5</td>
</tr>
</tbody>
</table>
Double hashing improves on linear probing:

- By using an increment which...
  - is based on a secondary hash of the key
  - ensures that all slots will be visited
    (by using an increment which is relatively prime to \( N \))
- Tends to reduce clustering \( \Rightarrow \) shorter probe paths

To generate relatively prime number:

- Set table size to prime, e.g., \( N = 127 \)
- Ensure secondary hash function returns number in range \([1, N - 1]\)
Double Hashing

Example: Insert 22

Suppose \( h(k) = k \% 7 \) and \( h_2(k) = k \% 3 + 1 \)

<table>
<thead>
<tr>
<th>No Item</th>
<th>15</th>
<th>No Item</th>
<th>10</th>
<th>4</th>
<th>No Item</th>
<th>No Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td></td>
<td>[1]</td>
<td></td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
</tr>
<tr>
<td></td>
<td>No Item</td>
<td></td>
<td>No Item</td>
<td></td>
<td></td>
<td>No Item</td>
</tr>
</tbody>
</table>
Example: Insert 22

Suppose \( h(k) = k \mod 7 \) and \( h_2(k) = k \mod 3 + 1 \)

\[
h(22) = 22 \mod 7 = 1 \Rightarrow \text{collision!}
\]
Example: Insert 22

Suppose $h(k) = k \mod 7$ and $h_2(k) = k \mod 3 + 1$

$$h(22) = 22 \mod 7 = 1 \Rightarrow \text{collision!}$$

$$h_2(22) = 22 \mod 3 + 1 = 2$$

<table>
<thead>
<tr>
<th>No Item</th>
<th>15</th>
<th>No Item</th>
<th>10</th>
<th>4</th>
<th>No Item</th>
<th>No Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td></td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td>[5]</td>
</tr>
</tbody>
</table>
Example: Insert 22

Suppose \( h(k) = k \% 7 \) and \( h_2(k) = k \% 3 + 1 \)

\[
h(22) = 22 \% 7 = 1 \Rightarrow \text{collision!}
\]

\[
h_2(22) = 22 \% 3 + 1 = 2
\]
Example: Insert 22

Suppose \( h(k) = k \mod 7 \) and \( h_2(k) = k \mod 3 + 1 \)

\[
\begin{align*}
    h(22) &= 22 \mod 7 = 1 \Rightarrow \text{collision!} \\
    h_2(22) &= 22 \mod 3 + 1 = 2
\end{align*}
\]
Given a hash table with 11 slots that uses double hashing, with primary hash function $h(k) = k \mod 11$ and secondary hash function $h_2(k) = k \mod 5 + 1$, insert the following keys:

5 20 16 1 42 15
Given a hash table with 11 slots that uses double hashing, with primary hash function $h(k) = k \% 11$ and secondary hash function $h_2(k) = k \% 5 + 1$, insert the following keys:

5 20 16 1 42 15
Given a hash table with 11 slots that uses double hashing, with primary hash function $h(k) = k \% 11$ and secondary hash function $h_2(k) = k \% 5 + 1$, insert the following keys:

5 20 16 1 42 15

$h(5) = 5 \% 11 = 5$
Given a hash table with 11 slots that uses double hashing, with primary hash function $h(k) = k \% 11$ and secondary hash function $h_2(k) = k \% 5 + 1$, insert the following keys:

\[5 \ 20 \ 16 \ 1 \ 42 \ 15\]

$h(5) = 5 \% 11 = 5$
Given a hash table with 11 slots that uses double hashing, with primary hash function \( h(k) = k \% 11 \) and secondary hash function \( h_2(k) = k \% 5 + 1 \), insert the following keys:

5  20  16  1  42  15
Given a hash table with 11 slots that uses double hashing, with primary hash function \( h(k) = k \% 11 \) and secondary hash function \( h_2(k) = k \% 5 + 1 \), insert the following keys:

\[
5 \quad 20 \quad 16 \quad 1 \quad 42 \quad 15
\]

\[
h(20) = 20 \% 11 = 9
\]
Given a hash table with 11 slots that uses double hashing, with primary hash function \( h(k) = k \% 11 \) and secondary hash function \( h_2(k) = k \% 5 + 1 \), insert the following keys:

\[
5 \quad 20 \quad 16 \quad 1 \quad 42 \quad 15
\]

\[
h(20) = 20 \% 11 = 9
\]
Given a hash table with 11 slots that uses double hashing, with primary hash function \( h(k) = k \mod 11 \)
and secondary hash function \( h_2(k) = k \mod 5 + 1 \),
insert the following keys:

\[
5 \ 20 \ 16 \ 1 \ 42 \ 15
\]
Given a hash table with 11 slots that uses double hashing, with primary hash function \( h(k) = k \% 11 \) and secondary hash function \( h_2(k) = k \% 5 + 1 \), insert the following keys:

\[
5 \quad 20 \quad 16 \quad 1 \quad 42 \quad 15
\]

\( h(16) = 16 \% 11 = 5 \Rightarrow \text{collision!} \)
Given a hash table with 11 slots that uses double hashing, with primary hash function \( h(k) = k \% 11 \) and secondary hash function \( h_2(k) = k \% 5 + 1 \), insert the following keys:

\[
5 \ 20 \ 16 \ 1 \ 42 \ 15
\]

\( h(16) = 16 \% 11 = 5 \Rightarrow \text{collision!} \)

\( h_2(16) = 16 \% 5 + 1 = 2 \)
Given a hash table with 11 slots that uses double hashing, with primary hash function \( h(k) = k \% 11 \) and secondary hash function \( h_2(k) = k \% 5 + 1 \), insert the following keys:

\[
\begin{align*}
5 & \quad 20 & \quad 16 & \quad 1 & \quad 42 & \quad 15 \\
\end{align*}
\]

\( h(16) = 16 \% 11 = 5 \Rightarrow \text{collision!} \)

\( h_2(16) = 16 \% 5 + 1 = 2 \)
Given a hash table with 11 slots that uses double hashing, with primary hash function $h(k) = k \% 11$ and secondary hash function $h_2(k) = k \% 5 + 1$, insert the following keys:

$$5 \quad 20 \quad 16 \quad 1 \quad 42 \quad 15$$

<table>
<thead>
<tr>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
<th>[10]</th>
</tr>
</thead>
</table>
Given a hash table with 11 slots that uses double hashing, with primary hash function \( h(k) = k \% 11 \) and secondary hash function \( h_2(k) = k \% 5 + 1 \), insert the following keys:

\[
5 \quad 20 \quad 16 \quad 1 \quad 42 \quad 15
\]

\[
h(1) = 1 \% 11 = 1
\]
Given a hash table with 11 slots that uses double hashing, with primary hash function $h(k) = k \mod 11$ and secondary hash function $h_2(k) = k \mod 5 + 1$, insert the following keys:

$$5 \ 20 \ 16 \ 1 \ 42 \ 15$$

$h(1) = 1 \mod 11 = 1$
Double Hashing

Example

Given a hash table with 11 slots that uses double hashing, with primary hash function $h(k) = k \% 11$ and secondary hash function $h_2(k) = k \% 5 + 1$, insert the following keys:

5 20 16 1 42 15
Given a hash table with 11 slots that uses double hashing, with primary hash function $h(k) = k \% 11$ and secondary hash function $h_2(k) = k \% 5 + 1$, insert the following keys:

$5 \ 20 \ 16 \ 1 \ 42 \ 15$

$h(42) = 42 \% 11 = 9 \Rightarrow \text{collision!}$
Given a hash table with 11 slots that uses double hashing, with primary hash function $h(k) = k \% 11$ and secondary hash function $h_2(k) = k \% 5 + 1$, insert the following keys:

$$5 \ 20 \ 16 \ 1 \ 42 \ 15$$

$h(42) = 42 \% 11 = 9 \Rightarrow$ collision!

$h_2(42) = 42 \% 5 + 1 = 3$
Given a hash table with 11 slots that uses double hashing, with primary hash function $h(k) = k \% 11$ and secondary hash function $h_2(k) = k \% 5 + 1$, insert the following keys:

$5 \ 20 \ 16 \ 1 \ 42 \ 15$

$h(42) = 42 \% 11 = 9 \Rightarrow$ collision!

$h_2(42) = 42 \% 5 + 1 = 3$
Given a hash table with 11 slots that uses double hashing, with primary hash function \( h(k) = k \% 11 \) and secondary hash function \( h_2(k) = k \% 5 + 1 \), insert the following keys:

\[
5 \quad 20 \quad 16 \quad 1 \quad 42 \quad 15
\]
Given a hash table with 11 slots that uses double hashing, with primary hash function \( h(k) = k \mod 11 \) and secondary hash function \( h_2(k) = k \mod 5 + 1 \), insert the following keys:

\[
5 \quad 20 \quad 16 \quad 1 \quad 42 \quad 15
\]

\( h(15) = 15 \mod 11 = 4 \Rightarrow \text{collision!} \)
Given a hash table with 11 slots that uses double hashing, with primary hash function \( h(k) = k \ % \ 11 \) and secondary hash function \( h_2(k) = k \ % \ 5 + 1 \), insert the following keys:

\[
5 \ 20 \ 16 \ 1 \ 42 \ 15
\]

\[
h(15) = 15 \ % \ 11 = 4 \Rightarrow \text{collision!}
\]

\[
h_2(15) = 15 \ % \ 5 + 1 = 1
\]
Given a hash table with 11 slots that uses double hashing, with primary hash function \( h(k) = k \% 11 \) and secondary hash function \( h_2(k) = k \% 5 + 1 \), insert the following keys:

\[
5 \ 20 \ 16 \ 1 \ 42 \ 15
\]

\( h(5) = 5 \% 11 = 5 \)
\( h(20) = 20 \% 11 = 9 \)
\( h(16) = 16 \% 11 = 5 \) ⇒ collision!
\( h_2(16) = 16 \% 5 + 1 = 2 \)
\( h(1) = 1 \% 11 = 1 \)
\( h(42) = 42 \% 11 = 9 \) ⇒ collision!
\( h_2(42) = 42 \% 5 + 1 = 3 \)
\( h(15) = 15 \% 11 = 4 \) ⇒ collision!
\( h_2(15) = 15 \% 5 + 1 = 1 \)
Given a hash table with 11 slots that uses double hashing, with primary hash function $h(k) = k \% 11$ and secondary hash function $h_2(k) = k \% 5 + 1$, insert the following keys:

5 20 16 1 42 15
Assuming integer keys and values:

```c
struct hashTable {
    struct slot *slots;
    int numSlots;
    int numItems;
    int hash2Mod;
};

struct slot {
    int key;
    int value;
    bool empty;
};
```
HashTable HashTableNew(void) {
    HashTable ht = malloc(sizeof(*ht));
    ht->slots = malloc(INITIAL_CAPACITY * sizeof(struct slot));
    for (int i = 0; i < ht->numSlots; i++) {
        ht->slots[i].empty = true;
    }

    ht->numSlots = INITIAL_CAPACITY;
    ht->numItems = 0;
    ht->hash2Mod = findSuitableMod(INITIAL_CAPACITY);
    return ht;
}
void HashTableInsert(HashTable ht, int key, int value) {
    if (/* load factor exceeds threshold */) {
        // resize
    }

    int i = hash(key, ht->numSlots);
    int inc = hash2(key, ht->hash2Mod);

    for (int j = 0; j < ht->numSlots; j++) {
        if (ht->slots[i].empty) {
            ht->slots[i].key = key;
            ht->slots[i].value = value;
            ht->slots[i].empty = false;
            ht->numItems++;
            return;
        }
        if (ht->slots[i].key == key) {
            ht->slots[i].value = value;
            return;
        }
        i = (i + inc) % ht->numSlots;
    }
}

Double Hashing
Lookup - Implementation

```c
int HashTableGet(HashTable ht, int key) {
    int i = hash(key, ht->numSlots);
    int inc = hash2(key, ht->hash2Mod);

    for (int j = 0; j < ht->numSlots; j++) {
        if (ht->slots[i].empty) break;
        if (ht->slots[i].key == key) {
            return ht->slots[i].value;
        }
        i = (i + inc) % ht->numSlots;
    }

    error;
}
```
How to delete an item?

Backshift method is harder to implement due to large increments

Tombstone method (lazy deletion) still works
Analysis of lookup:

- Hash function is $O(1)$
- Subsequent cost depends on probe path length
  - Affected by load factor $\alpha = M/N$
  - Average cost for successful search $= \frac{1}{\alpha} \ln \left( \frac{1}{1-\alpha} \right)$
  - Average cost for unsuccessful search $= \frac{1}{1-\alpha}$

Example costs (assuming large hash table):

<table>
<thead>
<tr>
<th>load factor ($\alpha$)</th>
<th>0.50</th>
<th>0.67</th>
<th>0.75</th>
<th>0.90</th>
</tr>
</thead>
<tbody>
<tr>
<td>search hit</td>
<td>1.4</td>
<td>1.6</td>
<td>1.8</td>
<td>2.6</td>
</tr>
<tr>
<td>search miss</td>
<td>1.5</td>
<td>2.0</td>
<td>3.0</td>
<td>5.5</td>
</tr>
</tbody>
</table>

Can be significantly better than linear probing
- Especially if table is heavily loaded
Collision resolution approaches:

- Separate chaining: Easy to implement, allows $\alpha > 1$
- Linear probing: Fast if $\alpha \ll 1$, complex deletion
- Double hashing: Avoids clustering issues with linear probing

All approaches can be used to achieve $O(1)$ performance on average, assuming

- good hash function
- table is appropriately resized if load factor exceeds threshold
Design Issues

- How to resize a hash table?
- How to avoid two calls when performing lookup?
How do we resize a hash table?

• Hash function depends on the number of slots
  • Items may not belong at the same index after resizing
• So all items must be re-inserted
• How much to resize by?
  • Good strategy is to roughly double the number of slots every resizing
How to avoid two calls when performing lookup?

- HashTableGet assumes the given key exists, and generates an error if it doesn’t.
- So to look up an item which we don’t know exists, we must perform two calls:
  - One call to HashTableContains to check for existence of key
  - One call to HashTableGet to get the value
- Idea: Provide another function that allows user to specify a default value to return if key does not exist

```c
int HashTableGetOrDefault(HashTable ht, int key, int defaultValue);
```
https://forms.office.com/r/5c0fb4tvMb