COMP2521 24T3
Hash Tables

Sushmita Ruj
cs2521@cse.unsw.edu.au

hash tables
hashing
collision resolution

Motivation

Keys and Values

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 courses to number of enrolments
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

unordered array

[0]	[1]	[2]	[3]	[4]	[5]
jas green	andrew red	sasha purple	jake yellow	kevin blue	hayden red

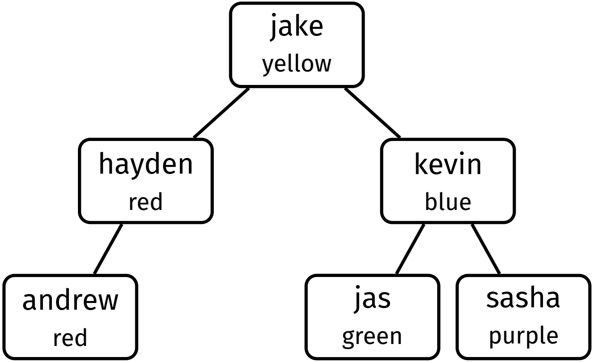
Performance?
Insert: $O(n)$
Lookup: $O(n)$
Delete: $O(n)$

ordered array

[0]	[1]	[2]	[3]	[4]	[5]
andrew red	hayden red	jake yellow	jas green	kevin blue	sasha purple

Performance?
Insert: $O(n)$
Lookup: $O(\log n)$
Delete: $O(n)$

balanced binary search tree



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

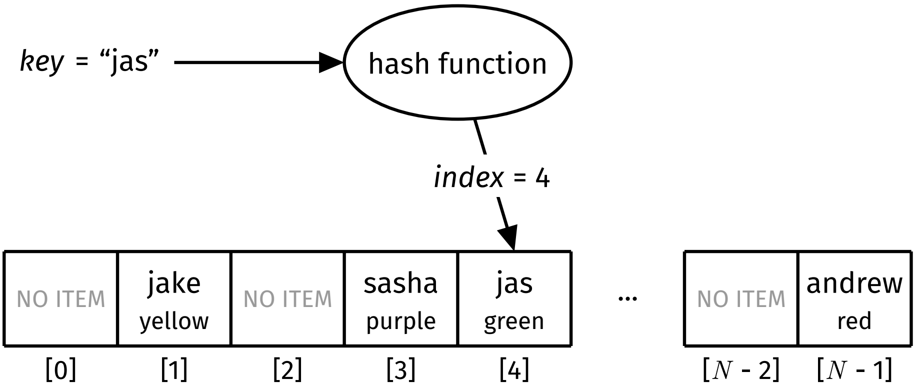
Hash Tables

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


key = “jas”
hash function
index = 4
NO ITEM
jake
yellow
NO ITEM
sasha
purple
jas
green
...
NO ITEM
andrew
red
[0]
[1]
[2]
[3]
[4]
[N - 2]
[N - 1]

Hash Tables

Interface

```
/** 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);
```

Hash Tables

Example Usage

```
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

Hash Functions

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

Hash Functions

Basic mechanism of hash functions:

```
int hash(Key key, int N) {
    int val = convert key to 32-bit int
    return val % N;
}
```

Simple hash function for `int`s:

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

Simple hash function for strings:

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

Hash Functions

Example

More robust hash function for strings:

```
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)...

```
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) {
        a += (k[0] + (k[1] << 8) + (k[2] << 16) + (k[3] << 24));
        b += (k[4] + (k[5] << 8) + (k[6] << 16) + (k[7] << 24));
        c += (k[8] + (k[9] << 8) + (k[10] << 16) + (k[11] << 24));
        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 ^= (b >> 13); \
  a -= b; a -= c; a ^= (c >> 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); \
}
```

Hashing

Example

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

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

Hashing
Example

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

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

Hashing
Example

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]

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

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

Hashing

Example

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

$$h(8) = 8$$

[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

$$h(8) = 8$$

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

Hashing

Example

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

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

Hashing
Example

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

$$h(15) = 4$$

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

Hashing

Example

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

$$h(15) = 4$$

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

index 4 already contains an item $\Rightarrow$ collision!

Often, the range of possible key values is *much* larger than the range of indices ($[0, N-1]$), so collisions are inevitable.

A hash collision occurs when for two keys x and y, $x \neq y$, but $h(x) = h(y)$.

A hash table must have a method for resolving collisions.

Collision Resolution

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

Collision Resolution

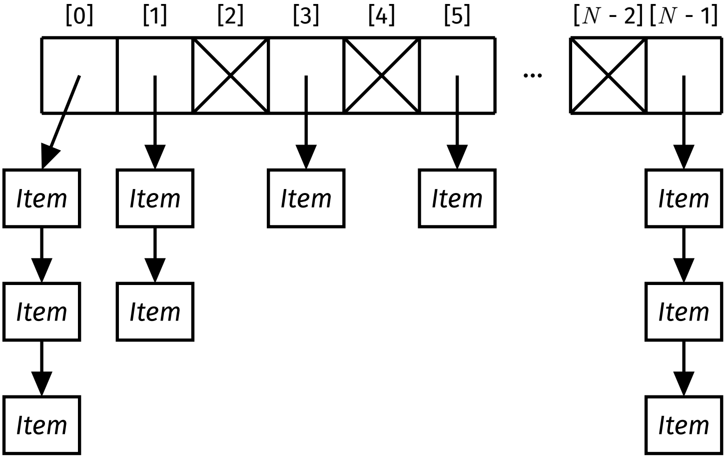
Important statistic: load factor (α)

- Ratio of items to slots; $\alpha = M/N$
- Useful when analysing collision resolution methods

Separate Chaining

Resolve collisions by having multiple items per array slot.

Each array slot contains a linked list of items that are hashed to that index.



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 4 16 42 8 15

[0]	[1]	[2]	[3]	[4]	[5]	[6]

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 4 16 42 8 15

[0]	[1]	[2]	[3]	[4]	[5]	[6]

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 4 16 42 8 15

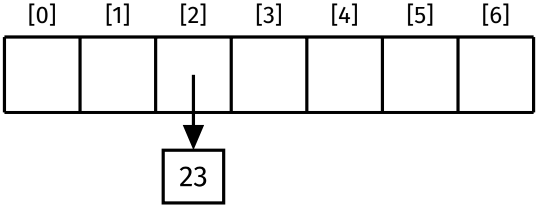
$$h(23) = 23 \ \% \ 7 = 2$$

[0]	[1]	[2]	[3]	[4]	[5]	[6]

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

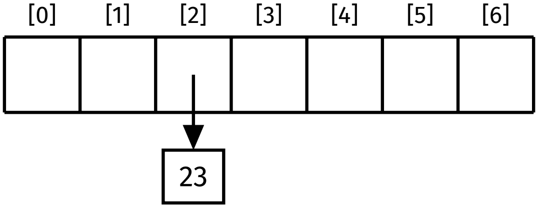
23 4 16 42 8 15

$$h(23) = 23 \,\%\, 7 = 2$$



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

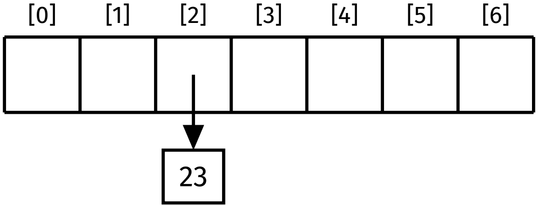
23 4 16 42 8 15



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

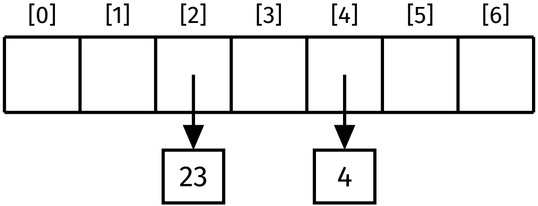
$$h(4) = 4 \% 7 = 4$$



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

$$h(4) = 4 \% 7 = 4$$

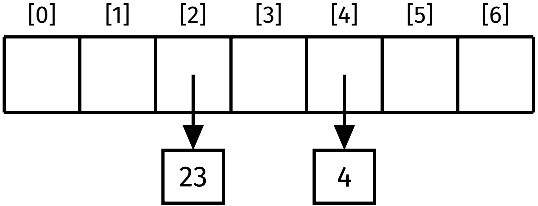


Separate Chaining

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 4 16 42 8 15



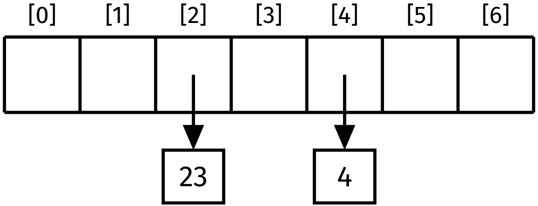
Separate Chaining

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 4 16 42 8 15

$$h(16) = 16 \,\%\, 7 = 2$$

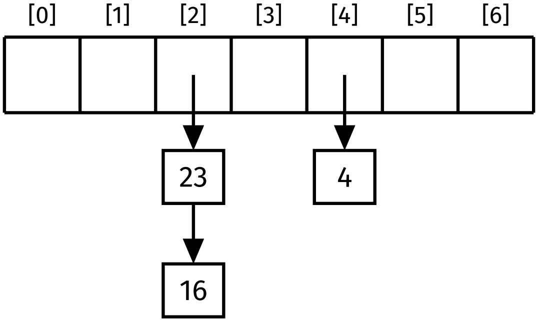


Separate Chaining
Example

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

23 4 16 42 8 15

$$h(16) = 16 \% 7 = 2$$

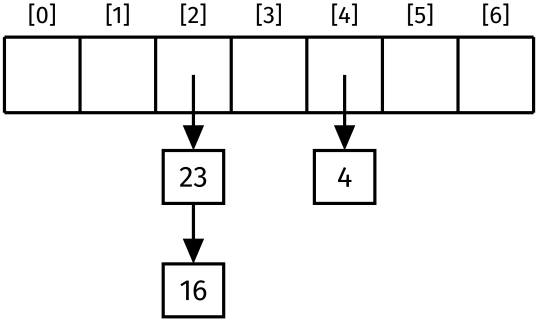


Separate Chaining

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 4 16 42 8 15



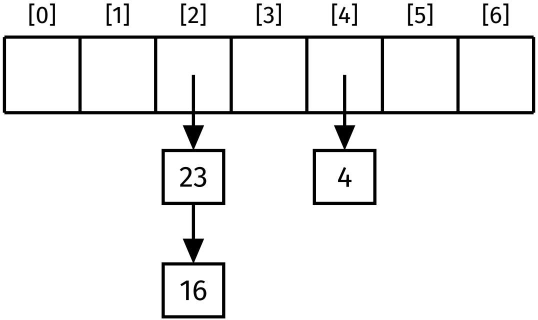
Separate Chaining

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 4 16 42 8 15

$$h(42) = 42 \% 7 = 0$$

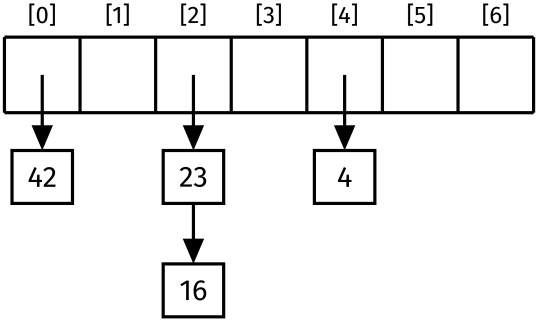


Separate Chaining
Example

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

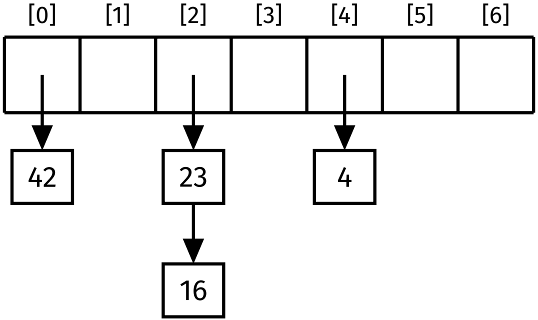
23 4 16 42 8 15

$$h(42) = 42 \,\%\, 7 = 0$$



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

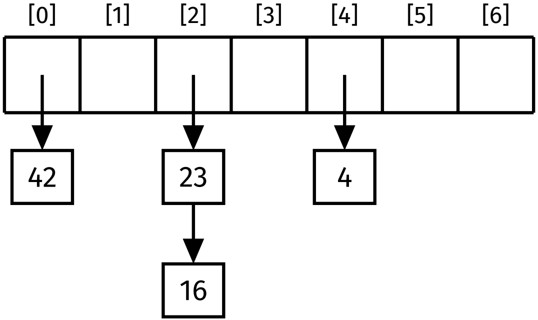
23 4 16 42 8 15



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

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



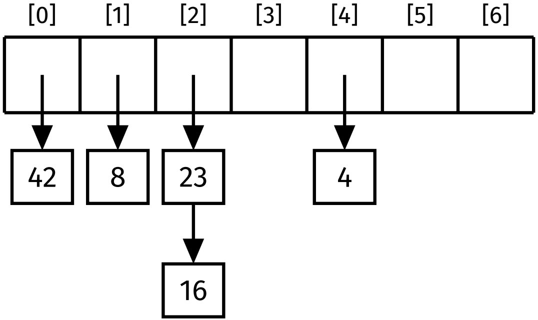
Separate Chaining

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 4 16 42 8 15

$$h(8) = 8 \,\%\, 7 = 1$$

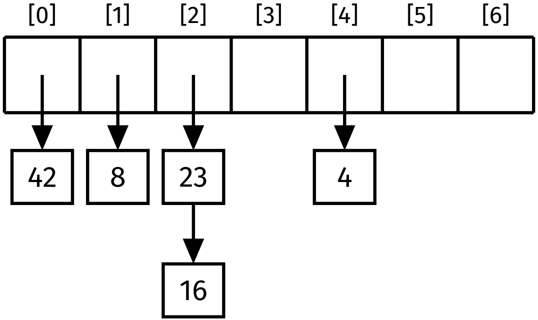


Separate Chaining

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 4 16 42 8 15



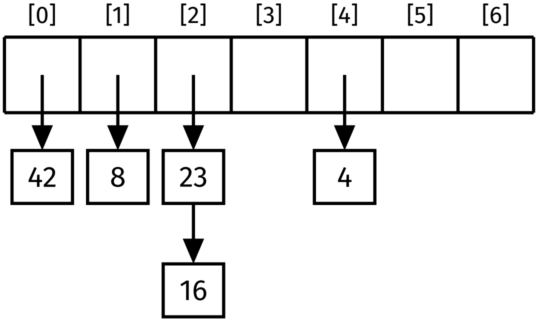
Separate Chaining

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 4 16 42 8 15

$$h(15) = 15 \,\%\, 7 = 1$$

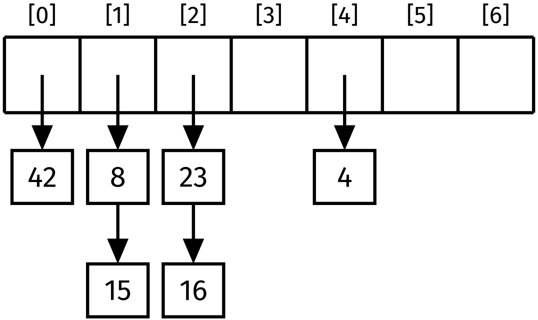


Separate Chaining
Example

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

23 4 16 42 8 15

$$h(15) = 15 \,\%\, 7 = 1$$



Assuming integer keys and values:

```
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;
}
```

Separate Chaining

Implementation

```
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;
}
```

Separate Chaining

Implementation

```
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;
}
```

Separate Chaining

Implementation


```
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:

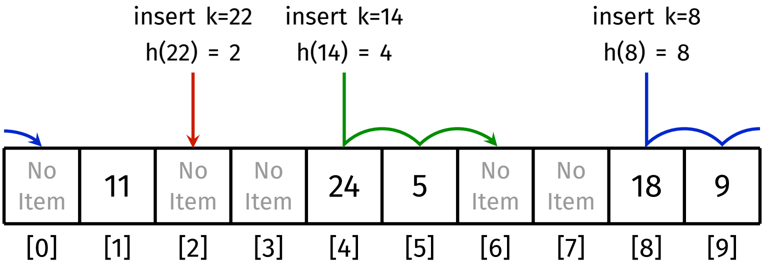
- If good hash and $\alpha \leq 1$, cost is $O(1)$
- If good hash and $\alpha > 1$, cost is $O(M/N)$
  - To avoid degrading perfomance, hash table should be resized when $\alpha \approx 1$

Linear Probing

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



Assuming integer keys and values:

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

struct slot {
    int key;
    int value;
    bool empty;
};
```

Linear Probing

Constructor


```
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;
    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

Linear Probing

Lookup - Implementation

```
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:

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

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	24	5	14	4	18	No Item

deleting 24 (incorrectly)

No Item	11	No Item	No Item	No Item	5	14	4	18	No Item

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

Linear Probing

Backshift Deletion - Example

Using the backshift method, delete 24 from this hash table:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	24	5	14	4	18	No Item

Linear Probing

Backshift Deletion - Example

Step 1: Remove 24

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	No Item	5	14	4	18	No Item

Step 2: Re-insert 5

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	No Item	5	14	4	18	No Item

Step 3: Re-insert 14

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	14	5	No Item	4	18	No Item

Linear Probing

Backshift Deletion - Example

Step 4: Re-insert 4

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	14	5	4	No Item	18	No Item

Step 5: Re-insert 18

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	14	5	4	No Item	18	No Item

This will be a task in the week 9 lab exercise!

Linear Probing

Tombstone Deletion - Example

Using the tombstone method, delete 14 from this hash table:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	24	5	14	4	18	No Item

Linear Probing

Tombstone Deletion - Example

After deleting 14:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	24	5	DEL	4	18	No Item

Search for 4:

h(4) = 4

No Item	11	No Item	No Item	24	5	DEL	4	18	No Item

Linear Probing

Tombstone Deletion - Example

Insert 15:

h(15) = 5

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	24	5	DEL	4	18	No Item

Result:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	24	5	15	4	18	No Item

Linear Probing

Deletion - Remarks

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 ($h(k) = k \ \% \ 15$):

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]

Insert 1, 2, 3, 17, 18

	1	2	3	17	18									

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

	1	2	3	17	18		7	22	9	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):

load factor (α)	0.50	0.67	0.75	0.90
search hit	1.5	2.0	3.0	5.5
search miss	2.5	5.0	8.5	55.5

Double Hashing

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 $h_2(k) = k \% 3 + 1$

[0]	[1]	[2]	[3]	[4]	[5]	[6]
No Item	15	No Item	10	4	No Item	No Item

Double Hashing

Example: Insert 22

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

$h(22) = 22 \ \% \ 7 = 1 \Rightarrow$ collision!

No Item	15	No Item	10	4	No Item	No Item
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Double Hashing

Example: Insert 22

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

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

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

No Item	15	No Item	10	4	No Item	No Item
[0]	[1]	[2]	[3]	[4]	[5]	[6]

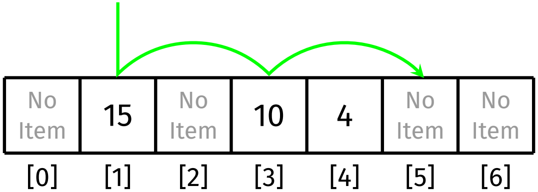
Double Hashing

Example: Insert 22

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

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

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



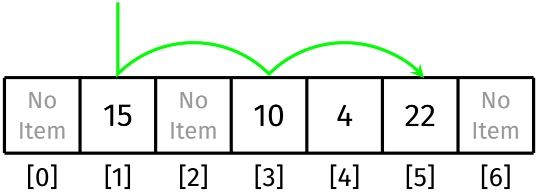
Double Hashing

Example: Insert 22

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

$h(22) = 22 \ \% \ 7 = 1 \Rightarrow$ collision!

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



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

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

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

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

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

$$h(5) = 5 \% 11 = 5$$

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

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

$$h(5) = 5 \,\%\, 11 = 5$$

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

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

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

$$h(20) = 20 \ \% \ 11 = 9$$

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

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

$$h(20) = 20 \,\%\, 11 = 9$$

					5				20	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

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

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
					5				20	

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

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

					5				20	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

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

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

$$h_2(16) = 16 \% 5 + 1 = 2$$

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
					5				20	

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

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

$$h_2(16) = 16 \;\%\; 5 + 1 = 2$$

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
					5		16		20	

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

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
					5		16		20	

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

$$h(1) = 1 \% 11 = 1$$

					5		16		20	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

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

$$h(1) = 1 \ \% \ 11 = 1$$

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	1				5		16		20	

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

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	1				5		16		20	

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!

	1				5		16		20	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

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

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

$$h_2(42) = 42 \ \% \ 5 + 1 = 3$$

	1				5		16		20	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

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

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

$$h_2(42) = 42 \% 5 + 1 = 3$$

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	1			42	5		16		20	

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

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	1			42	5		16		20	

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

$h(15) = 15 \% 11 = 4 \Rightarrow$ collision!

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	1			42	5		16		20	

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

$$h(15) = 15 \;\%\; 11 = 4 \Rightarrow \text{collision!}$$

$$h_2(15) = 15 \;\%\; 5 + 1 = 1$$

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	1			42	5		16		20	

Double Hashing

Example

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

5 20 16 1 42 15

$$h(15) = 15 \mathbin{\%} 11 = 4 \Rightarrow$$ collision!

$$h_2(15) = 15 \mathbin{\%} 5 + 1 = 1$$

	1			42	5	15	16		20	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

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

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	1			42	5	15	16		20	

Assuming integer keys and values:

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

struct slot {
    int key;
    int value;
    bool empty;
};
```

Double Hashing

Constructor


```
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;
}
```

Double Hashing

Insert - Implementation


```c
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

```
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;
}
```

Double Hashing
Deletion

How to delete an item?

Backshift method is harder to implement
due to large increments

Tombstone method (lazy deletion) still works

Double Hashing

Lookup - Analysis

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):

load factor (α)	0.50	0.67	0.75	0.90
search hit	1.4	1.6	1.8	2.6
search miss	1.5	2.0	3.0	5.5

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

- Initial size of hash table?
- How to resize a hash table?
- How to avoid two calls when performing lookup?

Design Issues

What should the initial size of the hash table be?

- If hash table is small initially, and many items are inserted, hash table will be resized many times
- Idea: Provide another function for creating hash table that allows users to specify initial size

```
HashTable HashTableNewWithSize(int N) {
    HashTable ht = malloc(sizeof(*ht));
    ht->slots = malloc(N * sizeof(*(ht->slots)));
    ...
    return ht;
}
```

Design Issues

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

Design Issues

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

```
int HashTableGetOrDefault(HashTable ht, int key, int defaultValue);
```

Feedback

https://forms.office.com/r/zEqxUXvmLR