CE2001/ CZ2001: Algorithms

Searching

Dr. Loke Yuan Ren

Learning Objectives

At the end of this lecture, students should be able to:

- Explain the basic principle of searching
- Apply the concept of sequential search algorithm using working examples
- Analyse the time complexity of sequential search algorithm

A Generic Search Routine

```
while (there is more possible data to examine){
    examine one datum;
    if (this datum == search key)
            return succeed;
     }
return fail;
```


A Generic Search Routine

```
while (there is more possible data to examine){
    examine one datum;
    if (this datum == search key)
            return succeed;
     }
return fail;
```

A Generic Search Routine

```
while (there is more possible data to examine){
    examine one datum;
    if (this datum == search key)
            return succeed;
    }
return fail;
```


A Generic Search Routine

```
while (there is more possible data to examine){
    examine one datum;
    if(this data == search key)
            return succeed;
    }
return fail;
```


Sequential Search (Unordered Array)

Sequential Search (Unordered Array)

```
int seqSearch (int [ ] Data, int n, int key)
{
   int index;

   for (index = 0; index < n; index++)
                if (key == Data[index])
                        return index; // succeed
   return -1; // fail
}
```

Sequential Search (Unordered Array)

```
int seqSearch (int [ ] Data, int n, int key)
{
   int index;

   for (index = 0; index < n; index++)
                if (key == Data[index])
                        return index; // succeed
   return -1; // fail
}
```


Sequential Search (Unordered Array)

```
int seqSearch (int [ ] Data, int n, int key)
{
   int index;

   for (index = 0; index < n; index++)
                if (key == Data[index])
                        return index; // succeed
   return -1; // fail
}
```


Sequential Search (Unordered Array)

```
int seqSearch (int [ ] Data, int n, int key)
{
   int index;

   for (index = 0; index < n; index++)
                if (key == Data[index])
                        return index; // succeed
   return -1; // fail
}
```


Sequential Search (Unordered Array)

```
int seqSearch (int [ ] Data, int n, int key)
{
   int index;

   for (index = 0; index < n; index++)
                if (key == Data[index])
                        return index; // succeed
   return -1; // fail
}
```

Successful Search

Target Given	14
Location Wanted	3

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
	4	21	36	14	62	91	8	22	7	81	77	10

Successful Search

Target Given	14
Location Wanted	3

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
Index 0	4	21	36	14	62	91	8	22	7	81	77	10

4 ≠14

Successful Search

Target Given	14
Location Wanted	3

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
Index 0	4	21	36	14	62	91	8	22	7	81	77	10
Index 1	4	21	36	14	62	91	8	22	7	81	77	10

4 ≠14

21 ≠14

Successful Search

Target Given	14
Location Wanted	3

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
Index 0	4	21	36	14	62	91	8	22	7	81	77	10
Index 1	4	21	36	14	62	91	8	22	7	81	77	10
Index 2	4	21	36	14	62	91	8	22	7	81	77	10

4 ≠14

21 ≠14

36 ≠14

Successful Search

Target Given	14
Location Wanted	3

Unsuccessful Search

Target Given	72

Unsuccessful Search

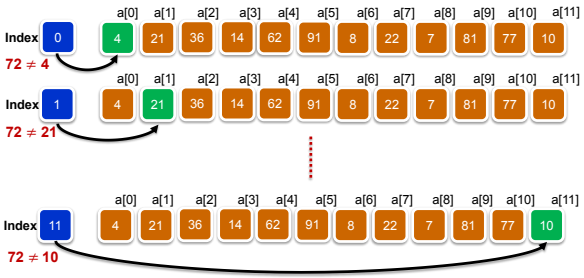
Target Given	72

Unsuccessful Search

Target Given	72

Unsuccessful Search

Target Given: 72



Note: Not all test points are shown in this search.

Complexity of Sequential Search

Complexity of Sequential Search

- **Best-case complexity:** 1 comparison against key
- **Worst-case complexity:** n comparisons against key
- **Average-case complexity:**
  - Consider the situation that the key is found in array
  - e_i represents the event that the key appears in i^{th} position of array, so its probability $P(e_i) = 1/n$
  - $T(e_i)$ is the no. of comparisons done
  - In this case, we have $0 \le i < n$, $T(e_i) = i + 1$
  - If key is in array, Avg. complexity $A_s(n)$:

$$= \sum_{i=0}^{n-1} P(e_i)T(e_i) = \sum_{i=0}^{n-1} \left(\frac{1}{n}\right)(i+1) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}$$

$= \frac{1}{n}\frac{n(n+1)}{2}$

Complexity of Sequential Search

- On the other hand, if key is not in array, number of comparisons $A_f(n)$ is n.
- Combining the above:
  - Pr(succ) $A_s(n)$ + Pr(fail) $A_f(n)$ = q * (n+1)/2 + (1 - q) * n
  - If there is a 50-50 chance that key is not in the array, (i.e. q =1/2),
  - Then average no. of key comparisons is 3n/4 + 1/4 (about 3/4 of entries examined)
- Both worst and average complexity are $\Theta(n)$.

$= \frac{1}{2}\frac{(n+1)}{2} + (1-\frac{1}{2})n$

Recap

- A generic search routine
- Sequential search
  - Examine data item sequentially
  - Average and worst time complexities are both $\theta(n)$

Binary Search

Learning Objectives

At the end of this lecture, students should be able to:

- Explain the Binary Search Algorithm
- Apply the binary search algorithm using working examples
- Analyse worst-case time complexity
- Analyse average-case time complexity

Binary Search

- Applicable when the array is ordered.
- This search method uses the information of the order of the elements and tries to do less work.
- This is another example of the divide and conquer approach.
- Here a problem is divided into 2 smaller sub-problems, one of which does not even have to be solved.

Binary Search


```
int binarySearch (int [ ] E, int first, int last, int k)  // k: Item to search
{
    if (last < first)
       return -1;
    else {
       int mid = (first + last) / 2;
       if (k == E[mid]) return mid;
       else if (k < E[mid])
                return binarySearch (E, first, mid − 1, k);
       else
                return binarySearch (E, mid + 1, last, k);
}
```


Binary Search


```
int binarySearch (int [ ] E, int first, int last, int k)
{
    if (last < first)
       return -1;
    else {
       int mid = (first + last) / 2;
       if (k == E[mid]) return mid;
       else if (k < E[mid])
                return binarySearch (E, first, mid − 1, k);
       else
                return binarySearch (E, mid + 1, last, k);
}
```


Binary Search


```
int binarySearch (int [ ] E, int first, int last, int k)
{
    if (last < first)
       return -1;
    else {
       int mid = (first + last) / 2;
       if (k == E[mid]) return mid;
       else if (k < E[mid])
                return binarySearch (E, first, mid − 1, k);
       else
                return binarySearch (E, mid + 1, last, k);
}
```


Binary Search


```
int binarySearch (int [ ] E, int first, int last, int k)
{
    if (last < first)
       return -1;
    else {
       int mid = (first + last) / 2;
       if (k == E[mid]) return mid;
       else if (k < E[mid])
                return binarySearch (E, first, mid − 1, k);
       else
                return binarySearch (E, mid + 1, last, k);
}
```

Binary Search


```
int binarySearch (int [ ] E, int first, int last, int k)
{
    if (last < first)
       return -1;
    else {
       int mid = (first + last) / 2;
       if (k == E[mid]) return mid;
       else if (k < E[mid])
                return binarySearch (E, first, mid − 1, k);
       else
                return binarySearch (E, mid + 1, last, k);
}
```


Binary Search


```
int binarySearch (int [ ] E, int first, int last, int k)
{
    if (last < first)
       return -1;
    else {
       int mid = (first + last) / 2;
       if (k == E[mid]) return mid;
       else if (k < E[mid])
                return binarySearch (E, first, mid − 1, k);
       else
                return binarySearch (E, mid + 1, last, k);
}
```


Binary Search


```
int binarySearch (int [ ] E, int first, int last, int k)
{
    if (last < first)
       return -1;
    else {
       int mid = (first + last) / 2;
       if (k == E[mid]) return mid;
       else if (k < E[mid])
                return binarySearch (E, first, mid − 1, k);
       else
                return binarySearch (E, mid + 1, last, k);
}
```


Binary Search


```
int binarySearch (int [ ] E, int first, int last, int k)
{
    if (last < first)
       return -1;
    else {
       int mid = (first + last) / 2;
       if (k == E[mid]) return mid;
       else if (k < E[mid])
                return binarySearch (E, first, mid − 1, k);
       else
                return binarySearch (E, mid + 1, last, k);
}
```

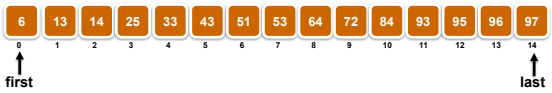
Non-recursive version possible

NANYANG TECHNOLOGICAL UNIVERSITY

Binary Search (Example 1)

Binary Search (Example 1)

Binary search for target **33**

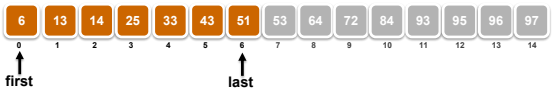


Binary Search (Example 1)

Binary search for target **33**

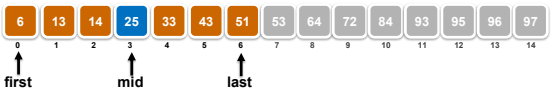
Binary Search (Example 1)

Binary search for target **33**



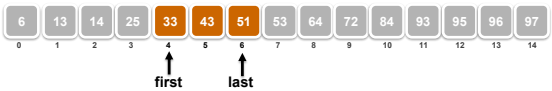
Binary Search (Example 1)

Binary search for target **33**



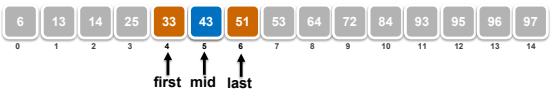
Binary Search (Example 1)

Binary search for target **33**



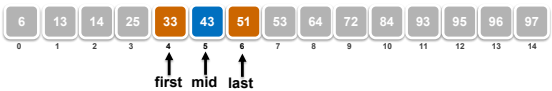
Binary Search (Example 1)

Binary search for target **33**



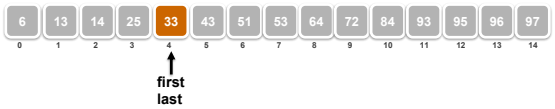
Binary Search (Example 1)

Binary search for target **33**



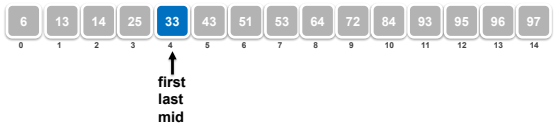
Binary Search (Example 1)

Binary search for target **33**



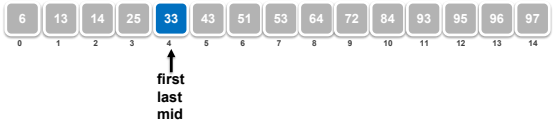
Binary Search (Example 1)

Binary search for target **33**



Binary Search (Example 1)

Binary search for target **33**

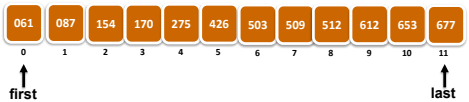


Successfully found 33!

Binary Search (Example 3)

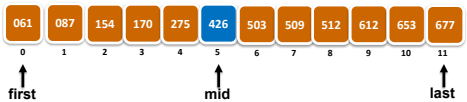
Example: Searching for 400

Binary search for target **400**



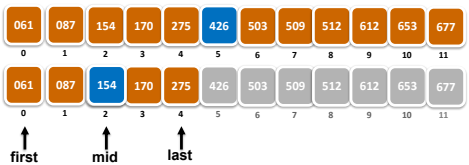
Example: Searching for 400

Binary search for target **400**



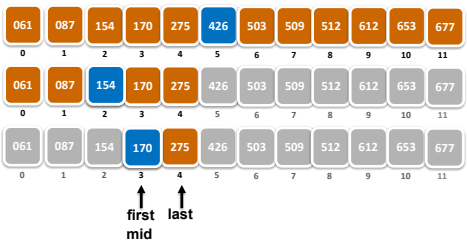
Example: Searching for 400

Binary search for target **400**



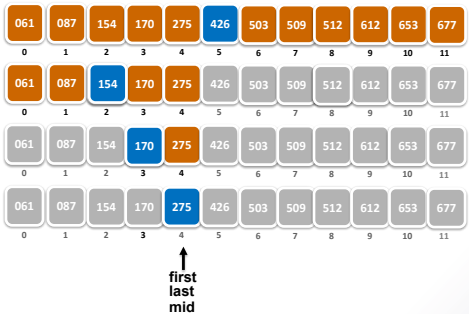
Example: Searching for 400

Binary search for target **400**



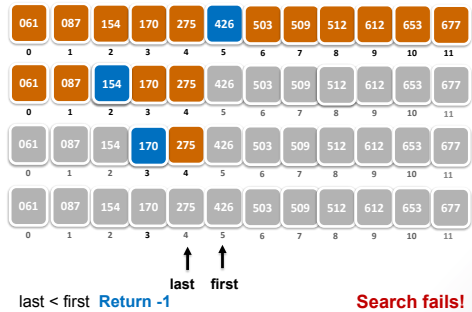
Example: Searching for 400

Binary search for target **400**



Example: Searching for 400

Binary search for target **400**



last < first **Return -1**

Search fails!

Best case:
When search key is right at the middle.

Worst-Case Time Complexity

Worst-Case Time Complexity

```
int binarySearch (int[ ] E, int first, int last, int k)        T(n)
{
   if (last < first) return −1;
   else {
        int mid = (first + last) / 2;                          Constant c
        if (k == E[mid]) return mid;
        else if (k < E[mid])
                return binarySearch(E, first, mid − 1, k);     T(n/2)
        else return binarySearch(E, mid + 1, last, k);         T(n/2)
   }
}
```

So, T(n) = c + T(n/2)

Worst Case is failure case T(0) = 0

T(n) ← time complexity

Actual $T(\frac{n-1}{2})$ But can be approx $T(\frac{n}{2})$

every recursive call ⇒ constant time, c + A recursive call.

Worst-Case Time Complexity

Solve recurrence equation T(n) = T(n/2) + c

T(1) = T(0) + c
T(0) = 0

T(n) = T(n/2) + c
= T(n/4) + 2c
= T(n/8) + 3c
= …
= T(1) + dc
= T(0) + (d+1) c

n → n/2 → n/2² → n/2³ → … → n/2^d

n/2^d ≥ 1 ⇒ d ≤ log₂ n
= ⌊log₂ n⌋

d – max times to divide n by 2 before going below 1

So, worst-case running time T(n) = ⌊log₂ n⌋ +1 = ⌈log₂ (n+1)⌉

Can be proven

n must be an integer.

Prove ⌊log₂ n⌋+1 = ⌈log₂ (n+1)⌉

- Let *k* be some integer number that k= ⌊log₂ n⌋ ,then

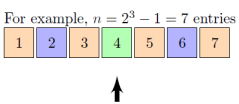
$$k \le \log_2 n < k+1$$
$$2^k \le n < 2^{k+1}$$
$$2^k < n+1 \le 2^{k+1}$$
$$k < \log_2(n+1) \le k+1$$

Eg. k=4,
$2^4 \le n < 2^5$
$2^4 < n+1 \le 2^5$
n is integer

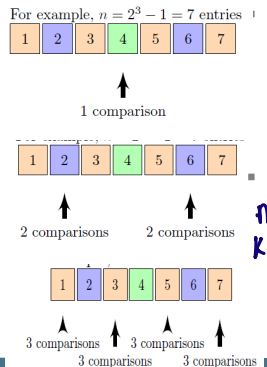
$$\lceil \log_2(n+1) \rceil = k+1$$
$$\lceil \log_2(n+1) \rceil = \lfloor \log_2 n \rfloor + 1$$

Average Number of Comparisons

- Law of Expectations:
- $A_q(n) = q\, A_s(n) + (1 - q)\, A_f(n)$,
- $A_s(n)$: # of comparisons for successful search
- $A_f(n)$: # of comparisons for unsuccessful search
- For simplicity, let $n=2^k-1$,
  - $A_f(n)$ is the worst case discussed in slide 63: $\lceil \log_2 (n+1) \rceil$
  - $A_s(n)$?



Average Number of Comparisons



We can observe that:
- 1 position requires 1 comparison
- 2 positions requires 2 comparisons
- 4 positions requires 3 comparisons
- … …
- 2^{t-1} positions requires t comparisons

- $n=2^k-1$, we have

$$A_s(n) = \frac{1}{n}\sum_{t=1}^{k} t\,2^{t-1}$$
$$= \frac{(k-1)2^k + 1}{n}$$
$$= \frac{[\log_2(n+1)-1](n+1)+1}{n}$$
$$= \log_2(n+1) - 1 + \frac{\log_2 n}{n}$$

Average Number of Comparisons

- The average complexity is

$$\begin{aligned} A_q(n) &= qA_s(n) + (1-q)A_f(n) \\ &= q[\log_2(n+1) - 1 + \frac{\log_2 n}{n}] + (1-q)(\log_2(n+1)) \\ &= \log_2(n+1) - q + q\frac{\log_2 n}{n} \\ &= \Theta(log_2(n)) \end{aligned}$$

- Binary search does approximately $\log_2(n+1)$ comparisons on average for n entries.
  - q is probability which is always ≤ 1
  - $\frac{\log_2 n}{n}$ is very small especially when n >> 1

Recap

- **Binary Search**
  - Divide and conquer strategy
  - Examine data item in the middle and search recursively on single half
  - Average and worst time complexities are both Θ(log(n))

CE2001/ CZ2001: Algorithms

Hashing

Dr. Loke Yuan Ren

Learning Objectives

At the end of this lecture, students should be able to:

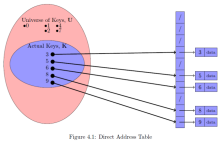
- Explain the concept of Hashing
- Explain the requirement of Hash Functions
- Use different hashing functions

Direct-Address Table

- Idea: Suppose that the set of actual keys is $K \subseteq \{0, 1, ..., m-1\}$ and keys are distinct.

  Set up an array T [0 .. m –1]:

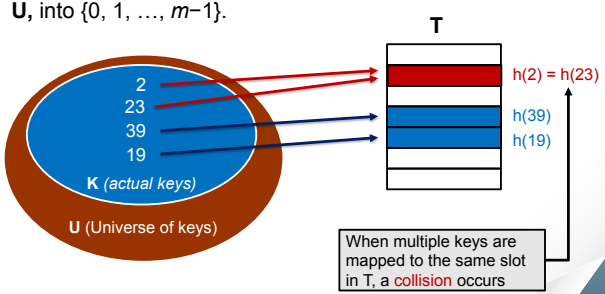
$$T[x.k] = \begin{cases} x, & \text{if } k \in K \wedge x.k = k \\ NIL, & \text{otherwise.} \end{cases}$$



- Then, operations take O(1) time
- **Issue:** The range of keys can be large (m >> |K|), e.g. 64-bit numbers (range m ≈ 18.45 × 10^{18})

Direct-Address Table (Solution)

Use a hash function **h** to map the all keys of the universe, **U,** into {0, 1, …, *m*−1}.



Hashing (Recap)

Motivation: To be able to assign a unique array index to every possible key that could occur in an application.

- Key space may be too *large* for an array on the computer while only a *small* fraction of the key values will appear.
- The purpose of hashing is to translate an extremely large key space into a reasonably small set of integers.
- A hash function f: key space → hash codes

Hash Table

- A hash table H is an array on indexes 0 .. m-1, each entry in the array is called a **hash slot**.

Example: a hash table of 200 entries.

A possible hash function:

f(k) = k mod 200 Note: Not a good hash function.

When multiple keys are mapped to the same hash code, a collision occurs.

Two main issues in designing a hash table:

(1) What hash function to use?

(2) How to handle collisions?

Hash Functions

- A hash function **MUST** return a value within the hash table range.
- It should achieve an even distribution of the keys that actually occur across the range of indices
- It should be easy and quick to compute

Tips
- If we know nothing about the incoming key distribution, evenly distribute the key range over the hash table slots while avoiding obvious opportunities for clustering.
- If we have knowledge of the incoming distribution, use a distribution-dependant hash function.

Hash Function (Example 1)

i. **The Division Method:**

int h(int x) { return(x % 16); }

- This function is entirely dependent on lower 4 bits of key.
- Avoid to use power of 10 for decimal numbers as keys
- The "best" table size is often a prime number not too close to exact powers of 2 for "real" data.

ii. **The Folding Method** (All bits contribute to the result):

- Partition the key into several parts and combine the parts in a convenient way (e.g. addition or multiplication).
- Example (Mid-square method):

  Square the key value, take the middle r bits (from the result) for a hash table of 2^r slots.

Hash Function (Example 2)

```
int h(char x[10])
{       int i, sum;
        for (sum=0, i=0; i < 10; i++)
                sum += (int) x[i];
        return(sum % M);
}
```

Only good if the sum is large compared to M

iii. **Multiplicative Congruential method** (pseudo-number generator):

Step 1: Choose the hash table size, h.
Step 2: Choose the multiplier, **a**
a = 8⌊h/23⌋ + 5
Step 3: Define the hash function, **f**
f(k) = (a * k) mod h

Recap

- **Concept of Hashing**
  - Direct-Address Table
  - Hash Functions

NANYANG TECHNOLOGICAL UNIVERSITY

CE2001/ CZ2001: Algorithms

Closed Address Hashing

Dr. Loke Yuan Ren

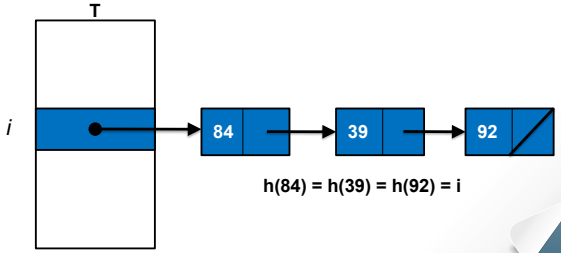
Learning Objectives

At the end of this lecture, students should be able to:

- Explain and apply closed address hashing method for collision handling
- Analyze worst-case and average-case complexity

Collision Handling 1: Closed Address Hashing

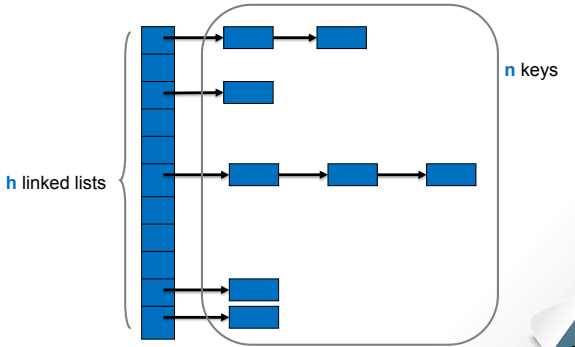
- **Maintains the original hashed address**
- **Records hashed to the same slot are linked into a list**
- **Also called Chained Hashing**



Closed Address Hashing

- Initially, all entries in the hash table are empty lists.
- All elements with hash address i will be inserted into the linked list H[i].
- If there are n records to store in the hash table, then n/h is the *load factor* of the hash table.
- In closed address hashing, there will be *n/h* number of elements in each linked list on average.
- During searching, the searched element with hash address i is compared with elements in linked list H[i] sequentially

Example of Hash Table with Chained Hashing



Analysis of chained hashing

- The worst case behaviour of hashing happens when all elements are hashed to the same slot. In this case,
  - The linked list contains all n elements
  - An unsuccessful search will do n key comparisons
  - A successful search, assuming the probability of searching for each item is 1/n, will take

$$\frac{1}{n}\sum_{i=1}^{n} i = \frac{(n+1)}{2} = \Theta(n)$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Note: Checking if link list is NULL is not counted as a key comparison

Average Case Analysis (Unsuccessful Search)

If we assume that any given item is equally likely to hash into any of the **h** slots, an unsuccessful search on average does **n/h** key comparisons.

Proof:

1. An unsuccessful search means searching to the end of the list.
2. The number of comparisons is equal to the length of the list.
3. The average length of all lists is the load factor, n/h.
4. Thus the expected number of comparisons in an unsuccessful search is n/h.

Average Case Analysis (Successful Search)

Assume that any given item is equally likely to hash into any of the h slots, a successful search on average does Θ(1 + n/h) comparisons.

Proof (Optional)**:**

1. We assume that the items are inserted at the end of the list in each slot.
2. The expected no. of comparisons in a successful search is 1 more than the no. of comparisons done when the sought after item was inserted into the hash table.
3. When the i^{th} item is inserted into the hash table, the average length of all lists is (i-1)/h. So when the i^{th} item is sought for, the no. of comparisons is $\left(1+\frac{i-1}{h}\right)$.

Average Case Analysis (Successful Search)

So the average number of comparisons over n items is:

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{h}\right)$$

$$=\frac{1}{n}\sum_{i=1}^{n}(1)+\frac{1}{nh}\sum_{i=1}^{n}(i-1)$$

$$=1+\frac{1}{nh}\sum_{i=0}^{n-1}i$$

$$=1+\frac{n-1}{2h}$$

Arimetic Series

$$\sum_{i=0}^{n-1} i=\sum_{i=1}^{n-1} i=\frac{n(n-1)}{2}$$

Therefore, a successful search on average does $\Theta(1 + n/h)$ comparisons (end of proof).

If n is proportional to h, i.e. n = O(h),then n/h = O(h)/h = O(1).

Thus, each successful search with chained hashing takes constant time averagely.

but not Theide. answer is $\frac{n-1}{h}$

Recap

- Closed address hashing
  - Handle collision through linked lists
  - Load factor = n/h
  - Worst case takes n comparisons
  - Average case
    - Unsuccessful search n/h comparisons
    - Successful search O(1)

CE2001/ CZ2001: Algorithms

Open Address Hashing

Dr. Loke Yuan Ren

Learning Objectives

At the end of this lecture, students should be able to:

- Describe how open address hashing works
- Apply open address hashing
- Explain and apply two types of rehashing:
  - Linear probing
  - Double hashing

Open Address Hashing

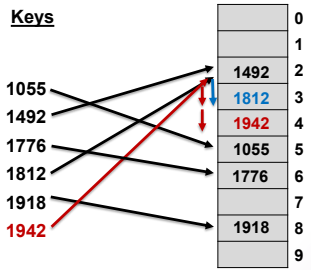
- To store all elements in the hash table
- The load factor n/h is never greater than 1
- When collision occurs, rehash/ probe the alternative empty slot
- Linear Probing: h(k,i) = (h'(k)+i) mod m
  1. Hash function, h(k) = k mod m
  2. If h(k) is non-empty slot
     - Rehash: h'(*h(k)*,i) = (*h(k)* + i) mod m, i=1, …, m-1
     - REPEAT till empty slot is found

k: key
m: hash table size

Open Address Hashing (Example)

- Consider the linear probing policy for storing the following keys: 1055, 1492, 1776, 1812, 1918, 1942. The hash function:

h(k) = k mod 10



Searching A Key with Rehashing

```
compute the hash code, code = h(k);
loc = code;  ans = blank structure;
while (H[loc] is not empty) {
   if (H[loc].key == k) {
      ans = H[loc];
      break; }
   else {
      loc = rehash(loc);
      if (loc == code)  break; }
}
ans =H[loc]
```

Key found!

Compute another location

Note: Checking if H[loc] is empty is not counted as a key comparison

Searching A Key with Rehashing

```
compute the hash code, code = h(k);
loc = code;  ans = blank structure;
while (H[loc] is not empty) {
   if (H[loc].key == k) {
      ans = H[loc];
      break; }
   else {
      loc = rehash(loc);
      if (loc == code)  break; }
}
```

Key found!

Compute another location

Note: Checking if H[loc] is empty is not counted as a key comparison

Behaviour of Rehashing

- **Three outcomes of searching:**
  1. key found at h(k) – success!
  2. position empty – fail!
  3. probe table (downwards subject to mod. table size) until key found or empty slot met or whole table searched.
- **Deletion under open addressing**

  A slot becomes empty when an element is deleted: should be marked as 'obsolete' or 'tombstone' instead, so that searching will not stop there.

Limitations of Rehashing

- **Issue with linear probing:** Searching is expensive when the load factor approaches 1.
  - **Primary clustering:** Long runs of occupied slots

Double Hashing

- **Double hashing:** a random rehashing method
  - Hash function, h(k) = k mod m
  - d = hashIncr(k);
  - rehash(j, d) = (h(k) + i*d) mod m , i=0,1,2…m-1

    where **hashIncr( )** is another hash function.
- The hash table size, m, should be a prime number.

Linear Probing Vs Double Hashing (Example)

To store the following keys:

1051, 1492, 1776, 1812, 1918, 1561, 523, 1340

- **Linear probing**

  h(k) = k mod 11 and rehash(j) = (j+i) mod 11, i=1, 2…10

- **Double hashing**

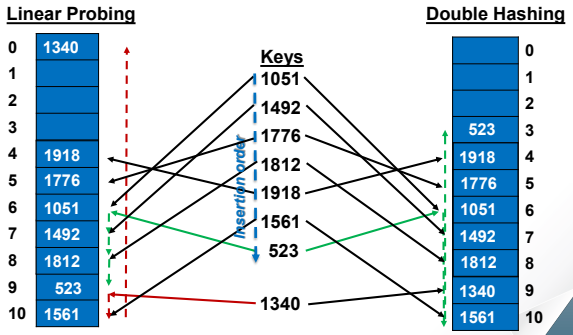
  h (k) = k mod 11, d = hashIncr(k) = k mod 8 + 1;
  and rehash(j, d) = (j+i*d) mod 11 , i=1,2…10

Linear Probing Vs Double Hashing (Example)

To store the following keys:
1051, 1492, 1776, 1812, 1918, 1561, 523, 1340

- **Linear probing**
  h(k) = k mod 11 and rehash(j) = (j+i) mod 11, i=1, 2…10
- **Double hashing**
  h (k) = k mod 11, d = hashIncr(k) = k mod 8 + 1;
  and rehash(j, d) = (j+i*d) mod 11 , i=1,2…10

Linear Probing Vs Double Hashing (Example)



Double Hashing

- d = hashIncr(k) = k mod 8 + 1
  - Why do I +1?
- The size of hash table in double hashing must be prime number?

Double hashing
h (k) = k mod 11, d = hashIncr(k) = k mod 8 + 1;
*and rehash(j, d) = (j+i*d) mod 11 , i=1,2…10*

Recap

- Collision Handling
  - Open Address Hashing – not fixed address (hash function)
  - Closed Address Hashing – fixed address