SC1007
Data Structures and Algorithms

Hash Tables

College of Engineering

School of Computer Science and Engineering

Dr. Loke Yuan Ren
Lecturer
yrloke@ntu.edu.sg

Overview

- Exhaustive Algorithm: Sequential Search
- Decrease-and-conquer Algorithm: Binary Search
- Data Structure Approach:
  - Hashing
    - Open Hashing
    - Closed Hashing

Time Complexity of Sequential Search

- Best-case complexity: Θ (1) , 1 comparison against key (the first item is the search key)
- Worst-case complexity: Θ (n) , n comparison against key (Either the last item or no item is the search key)
- Average-case complexity:

Key is in the search array:

- e_i represents the event that the key appears in ith position

  so its probability $P(e_i) = 1/n$
- $T(e_i)$ is the no. of comparisons done
- Average complexity: $A_s(n) = \sum_{i=1}^{n}(\frac{1}{n})(i)$
- $= \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$ = Θ (n)
-

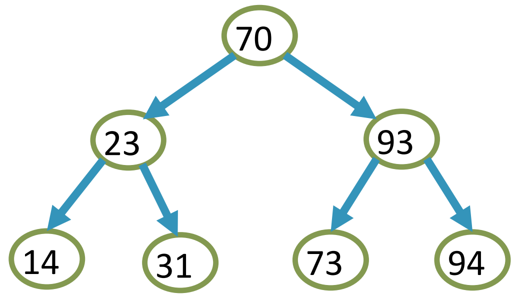
Key is not in the search array:

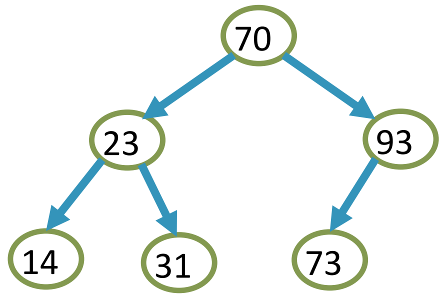
- n comparisons (if it is an linked list, you may need to take an extra comparison
- $A_s(n) = n$ = Θ (n)

Pr(succ) $A_s(n)$ + Pr(fail) $A_f(n)$ $= q\frac{n+1}{2} + (1-q)n$

Both worst and average complexity are Θ(n)

Terminology





- The Height of a tree: The number of **edges** on the longest path from the root to a leaf
- The Depth/Level of a node: The number of edges from the node to the root of its tree.

For a complete binary tree with height *H*, we have:

$$\mathbf{2^H - 1 < n \leq 2^{H+1} - 1}$$

where *n* is an integer and the size of the tree

$$\mathbf{2^H \leq n < 2^{H+1}} \qquad \textbf{(eg. } 7<n \leq 15 \equiv 8 \leq n<16\textbf{)}$$

$$\mathbf{H \leq \log_2 n < H+1}$$

If H is an integer, H+1 must be the next integer.

Minimal Height = $\lfloor \log_2 n \rfloor$

Binary Search – Worst Case Time complexity

- Assumed that it is a complete binary tree

```
BTNode* findBSTNode(BTNode *cur, char c){
    if (cur == NULL) {
       printf("Not Found!\n");
       return cur;
    }

    if(c==cur->item){
       printf("Found!\n");
       return cur;
    }

    if(c<cur->item)
       return findBSTNode(cur->left,c);
    else
       return findBSTNode(cur->right,c);
}
```

T(n)

Constant c

$\boldsymbol{T((n-1)/2)}$

$\boldsymbol{T((n-1)/2)}$

$$T(n) = T\left(\frac{n-1}{2}\right) + c$$

$$= T\left(\frac{\left(\frac{n-1}{2}\right) - 1}{2}\right) + 2c = T\left(\frac{n-1-2}{2^2}\right) + 2c$$

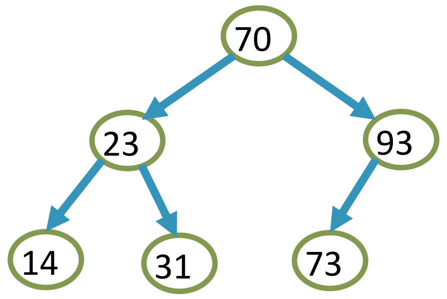
$$= T\left(\frac{\frac{n-1-2}{2^2} - 1}{2}\right) + 3c = T\left(\frac{n-1-2-2^2}{2^3}\right) + 3c$$

$$\ldots$$

$$= T\left(\frac{n-(1+2\ldots+2^{k-2}+2^{k-1})}{2^k}\right) + kc$$

$$= T\left(\frac{n-2^k+1}{2^k}\right) + kc$$

Binary Search – Worst Case Time complexity



```
BTNode* findBSTNode(BTNode *cur, char c){
    if (cur == NULL) {
       printf("can't find!");
       return cur;
    }

    if(c==cur->item){
       printf("Found!\n");
       return cur;
    }

    if(c<cur->item)
       return findBSTNode(cur->left,c);
    else
       return findBSTNode(cur->right,c);
}
```

T(n)

Constant c

$\boldsymbol{T((n-1)/2)}$

$\boldsymbol{T((n-1)/2)}$

- Assumed that it is a complete binary tree

$$= T\left(\frac{n-2^k+1}{2^k}\right) + kc$$

$$0 < \frac{n-2^k+1}{2^k} \leq 1$$

$$0 < \frac{n+1}{2^k} - 1 \leq 1$$

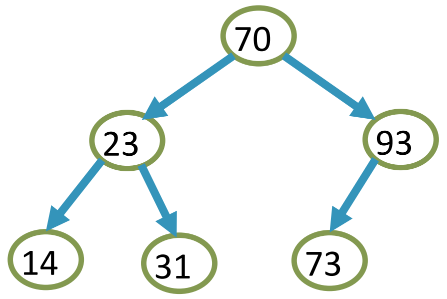
$$1 < \frac{n+1}{2^k} \leq 2$$

$$2^k < n+1 \leq 2^{k+1}$$

$$k < \log_2(n+1) \leq k+1$$

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

Binary Search – Worst Case Time complexity



```
BTNode* findBSTNode(BTNode *cur, char c){
    if (cur == NULL) {
       printf("can't find!");
       return cur;
    }

    if(c==cur->item){
       printf("Found!\n");
       return cur;
    }

    if(c<cur->item)
       return findBSTNode(cur->left,c);
    else
       return findBSTNode(cur->right,c);
}
```

T(n)

Constant c

$\boldsymbol{T((n-1)/2)}$

$\boldsymbol{T((n-1)/2)}$

- Assumed that it is a complete binary tree

$$= T\left(\frac{n-2^k+1}{2^k}\right) + kc$$

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

$$\lfloor \log_2 n \rfloor + 1 = \mathrm{k} + 1$$

$$k = \lfloor \log_2 n \rfloor$$

$$= c + kc$$

$$= (\lfloor \log_2 n \rfloor + 1)c$$

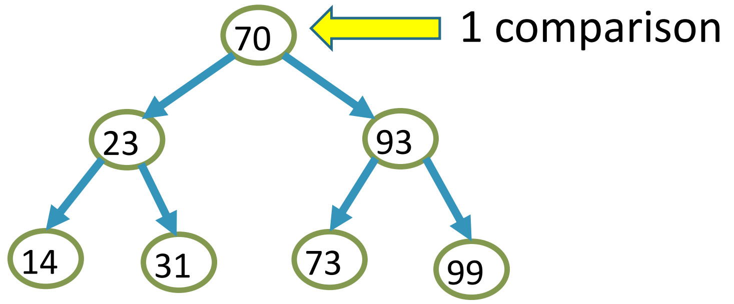
$$= \Theta(\log_2 n)$$

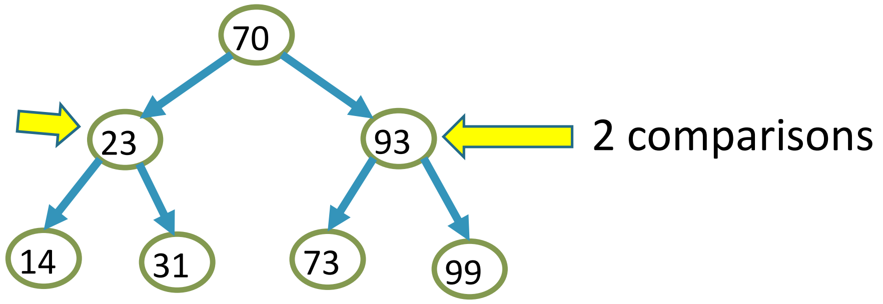
Binary Search – Average Case Time Complexity

- $A_s(n)$: # of comparisons for successful search
- $A_f(n)$: # of comparisons for unsuccessful search (worst case): $\Theta(\log_2 n)$

$$A(n) = qA_s(n) + (1-q)A_f(n)$$

For $A_s(n)$, we assume $n = 2^k - 1$ first





Binary Search – Average Case Time Complexity

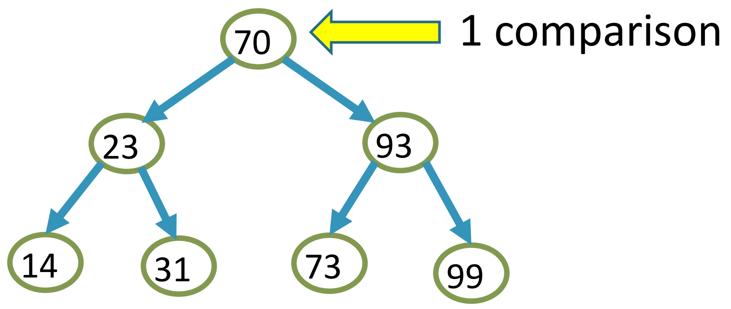
$$A(n) = qA_s(n) + (1-q)A_f(n)$$

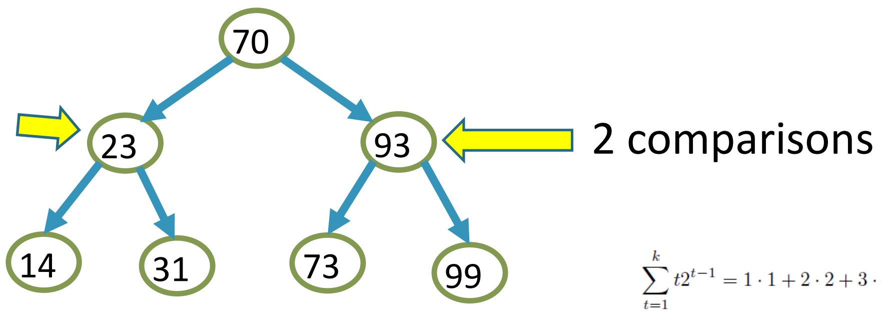
- For $A_s(n)$, we assume $n = 2^k - 1$ first
- We can observe that:
  - 1 position requires 1 comparison
  - 2 positions requires 2 comparisons
  - 4 positions requires 3 comparisons
  - … …
  - 2^{t-1} positions requires t comparisons
- n=2^k-1, we have $A_s(n) = \frac{1}{n}\sum_{t=1}^{k} t2^{t-1}$

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

$$= \frac{[\log_2(n+1) - 1](n+1) + 1}{n}$$

$$= \log_2(n+1) - 1 + \frac{\log_2(n+1)}{n}$$





$$\sum_{t=1}^{k} t2^{t-1} = 1\cdot 1 + 2\cdot 2 + 3\cdot 4 + 4\cdot 8 + \ldots + k\cdot 2^{k-1}$$

$$2\sum_{t=1}^{k} t2^{t-1} = \quad 1\cdot 2 + 2\cdot 4 + 3\cdot 8 + \ldots + (k-1)\cdot 2^{k-1} + k\cdot 2^k$$

$$(2-1)\sum_{t=1}^{k} t2^{t-1} = -1\cdot 1 - 1\cdot 2 - 1\cdot 4 - 1\cdot 8 - \ldots - 1\cdot 2^{k-1} + k\cdot 2^k \quad \triangleright \text{eq. 2 - eq. 1}$$

$$\sum_{t=1}^{k} t2^{t-1} = -2^k + 1 + k\cdot 2^k \quad \triangleright \text{geometric series}$$

$$= 2^k(k-1) + 1$$

Binary Search – Average Case Time Complexity

- The time complexity is

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

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

Hash Table

- A typical space and time trade-off in algorithms
- A **hash table** is a data structure that allows for efficient lookup, insertion, and deletion of key-value pairs.
- To achieve search time in O(1), we can use hash tables but memory usage will be increased significantly.

Some Applications of Hash Tables

- **Caching**: Hash tables are commonly used in caching applications to store frequently accessed data. The hash table can be used to store key-value pairs, where the key is a unique identifier for the data and the value is the data itself.
- **Databases**: Hash tables are often used in databases to provide fast lookups of data. For example, a hash table can be used to store a table of users, with the user ID as the key and the user's information as the value.
- **Counting**: Hash tables can be used to count occurrences of items in a dataset. Each item can be stored as a key in the hash table, with the value representing the count of occurrences. This can be used in applications like word frequency analysis, where the frequency of words in a document is analyzed.
- **Cryptography**: Hash tables are used in cryptography to store password hashes. When a user logs in, their password is hashed and compared to the hashed value stored in the hash table. This allows for secure authentication without storing the actual passwords in the hash table.

Hashing

- **Hashing** is the process of using a hash function to map data of arbitrary size to fixed-size values or keys.
- To design a hash table for data retrieval, we need to consider
  - Hash function
  - Collision and its resolutions
  - Delete a key from a hash table
  - Resizing hash table – Dynamic hash table

Direct-Address Table

- Assume that the keys of elements K drawn from the universe of possible keys U
- No two elements have the same key
- Search time is O(1) but …
    - The array size is enormous
    - |U| >> |K|
    - if the keys are integers between 1 and 1000, the array used to implement the direct-address table must have 1000 elements.

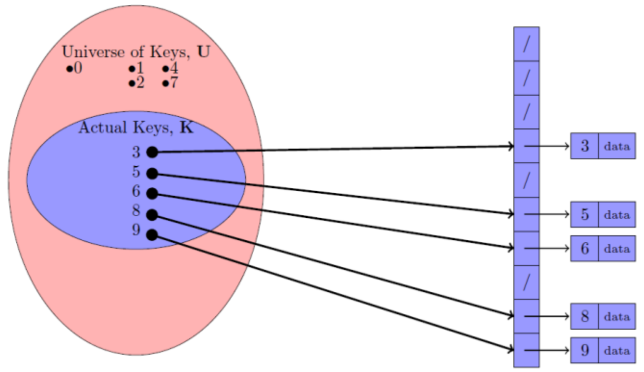


Figure 4.1: Direct Address Table

What is hashing?

- To reduce the key space to a reasonable size
- Each key is mapped to a unique index (**hash value/code/address**)
- Search time remains O(1) on the average

hash function: {all possible keys} → {0, 1, 2, …, h-1}

- The array is called a **hash table**
- Each entry in the hash table is called a **hash slot**
- When multiple keys are mapped to the same hash value, a **collision** occurs
- If there are n records stored in a hash table with h slots, its **load factor is** $\boldsymbol{\alpha = \frac{n}{h}}$

Hash Functions

- Must map all possible value within the range of the hash table uniquely
- Mapping should achieve an even distribution of the keys
- Easy and fast to compute
- Minimize collision

1. Modulo Arithmetic
2. Folding
3. Mid-square
4. Multiplicative Congruential Method
5. Etc.

Hash Functions

1. Modulo Arithmetic: $\mathrm{H}(k) = k \bmod h$

For keys are chosen from

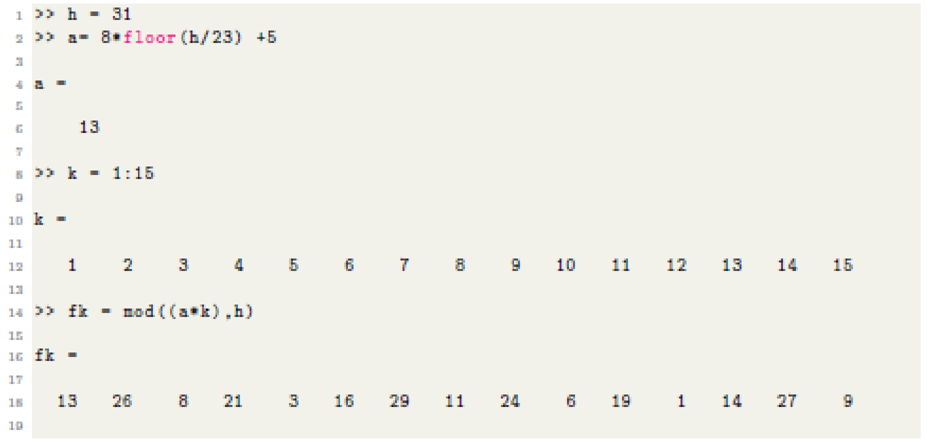
- decimal number → h avoid to use powers of 10
- unknown lower p-bit patterns → h avoid to use powers of 2
- “real” data → h should be a prime number but not too close to any power of 2

2. Folding
- Partition the key into several parts and combine the parts in a convenient way
- Shift folding: Divide the key into a few parts and added up these parts

Hash Functions

3. Mid-square
   - The key is squared and the middle part of the result is used as the hash address
     - Eg. k=3121, $k^2 = 3121^2 = 9740641 \rightarrow H(k) = 406$

4. Multiplicative Congruential Method
   - Pseudo-random number generator
     - $a = 8\left\lfloor\frac{h}{23}\right\rfloor + 5$
     - $H(k) = (a \times k)\ mod\ h$



```
>> h = 31
>> a= 8*floor(h/23) +5

a =

    13

>> k = 1:15

k =

     1     2     3     4     5     6     7     8     9    10    11    12    13    14    15

>> fk = mod((a*k),h)

fk =

    13    26     8    21     3    16    29    11    24     6    19     1    14    27     9

```

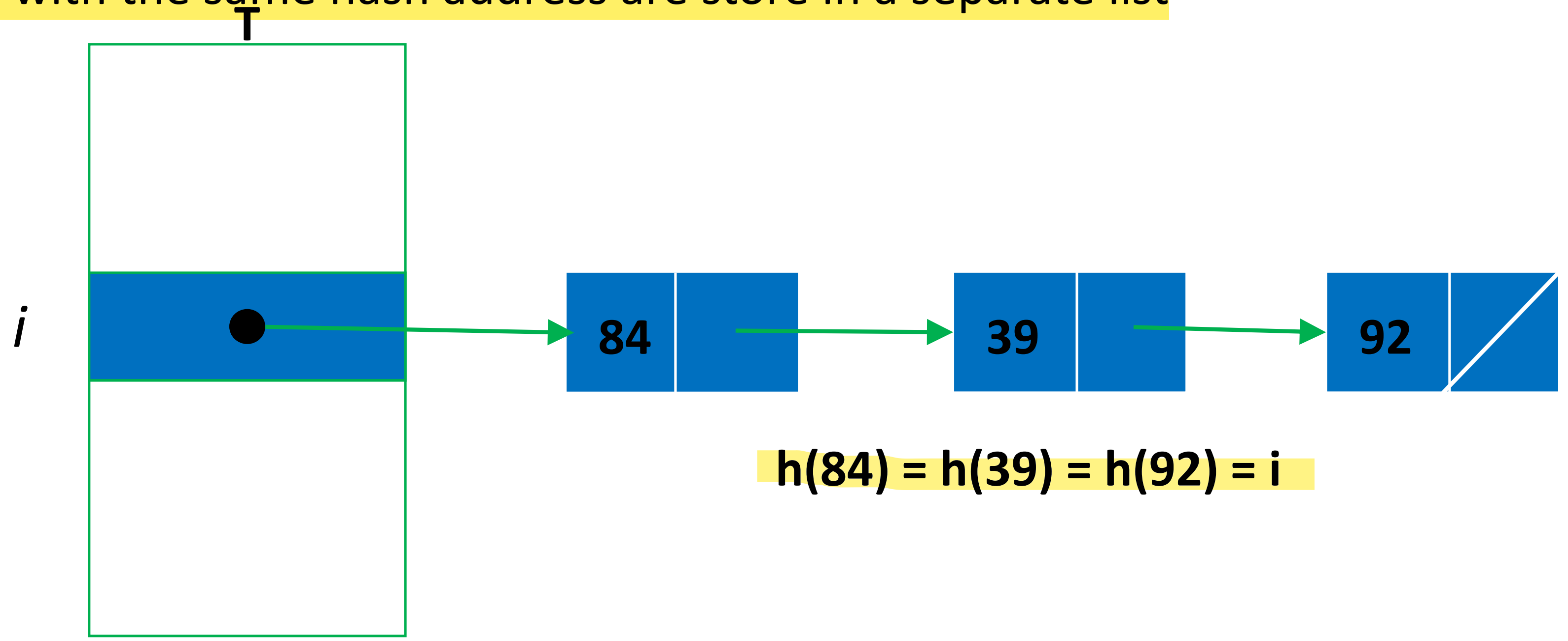
Collision Resolutions

- Closed Addressing Hashing – a.k.a separate chaining

- Open Addressing Hashing
  - Linear Probing
  - Quadratic Probing
  - Double Probing

Closed Addressing: Separate Chaining

- Keys are not stored in the table itself
- All the keys with the same hash address are store in a separate list



- During searching, the searched element with hash address i is compared with elements in linked list H[i] sequentially
- In closed address hashing, there will be α number of elements in each linked list on average.

Closed Addressing: Separate Chaining

Time complexity in the worst-case analysis:

- When all elements are hashed to the same slot
- A linked list contains all n elements
- Its unsuccessful search takes n key comparisons, $\Theta(n)$
- Its successful search, assuming the probability of searching for each item is $\frac{1}{n}$
  $\frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2} = \Theta(n)$
  - It is just like a sequential search

Closed Address Hashing: Separate Chaining

Time complexity in the average-case analysis:

- All elements are equally likely hashed into h slots.
- Its unsuccessful search takes $\frac{n}{h}$ key comparisons, $\Theta(\alpha)$
- Its successful search takes 1 more than the number of comparisons done when the sought after item was inserted into the hash table
  - Before the ith item is inserted into the hash table, the average length of all lists is $\frac{i-1}{h}$
  - When the ith item is sought for, the no. of comparisons is $(1 + \frac{i-1}{h})$
  - The average number of key comparisons over n items
  - If α is constant (n is proportional to h), then $\Theta(1)$
  - Searching takes constant time averagely

success fail

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

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

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

Open Addressing

- Keys are stored in the table itself
- α cannot be greater than 1
- When collision occurs, probe is required for the alternate slot
  - Ideally, the probing approach can visit every possible slot.

1. Linear Probing: probe the next slot

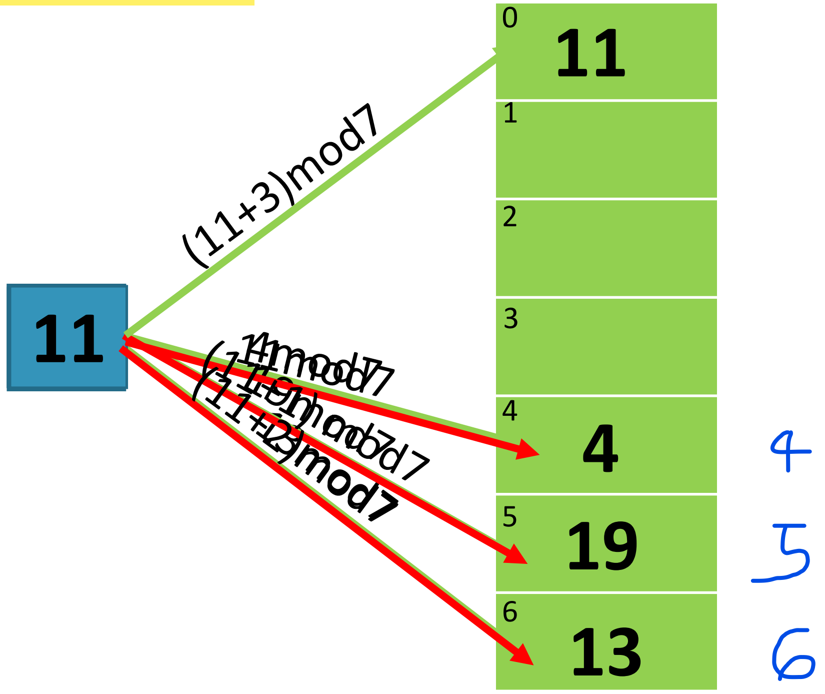
$H(k, i) = (k + i) \bmod h$ where $i \in [0, h - 1]$ −

eg. $H(k, i) = (k + i) \bmod 7$

$k \in \{4, 13, 19, 11\}$

Primary clustering:

- A long runs of occupied slots
- Average search time is increased



Open Addressing

2. Quadratic Probing

$$H(k,i) = (k + c_1 i + c_2 i^2) \bmod h \quad \text{where } c_1 \text{ and } c_2 \text{ are constants, } c_2 \neq 0$$

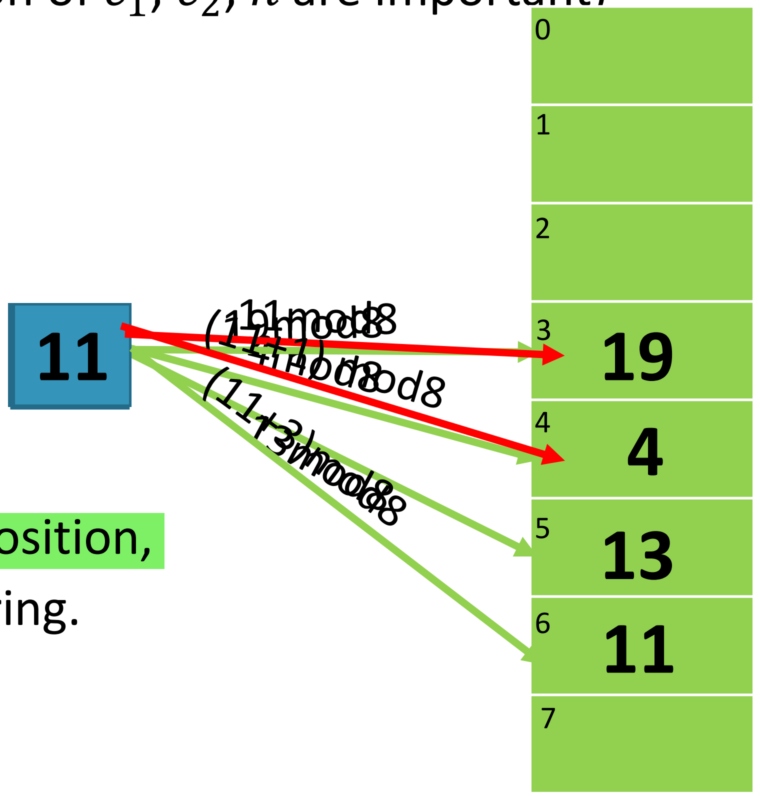
- May not all hash table slots be on the probe sequence (selection of c_1, c_2, h are important)
- For $h = 2^n$, a good choice for the constants are $c_1 = c_2 = \frac{1}{2}$

eg. $H(k,i) = \left(k + \frac{1}{2}i + \frac{1}{2}i^2\right) \bmod 8$

$k \in \{4, 13, 19, 11\}$

4 5 3 | 3
4
3+1+2=6

$(\frac{1}{2}i + \frac{1}{2}i^2)$mod 8
1
3
6
2
7
5
4



- Secondary Clustering: if two keys have the same initial probe position, their probe sequences will be the same. This will form a clustering.
    - Inserting k=3 in the previous example.

Open Addressing

3. Double Hashing: a random probing method

$H(k, i) = (k + iD(k)) \bmod h$ where $i \in [0, h-1]$ and $D(k)$ is another hash function

- The hash table size h should be a prime number

eg. $H(k, i) = (k + iD(k)) \bmod 7$

$D(k) = (k) \bmod 3 + 1$

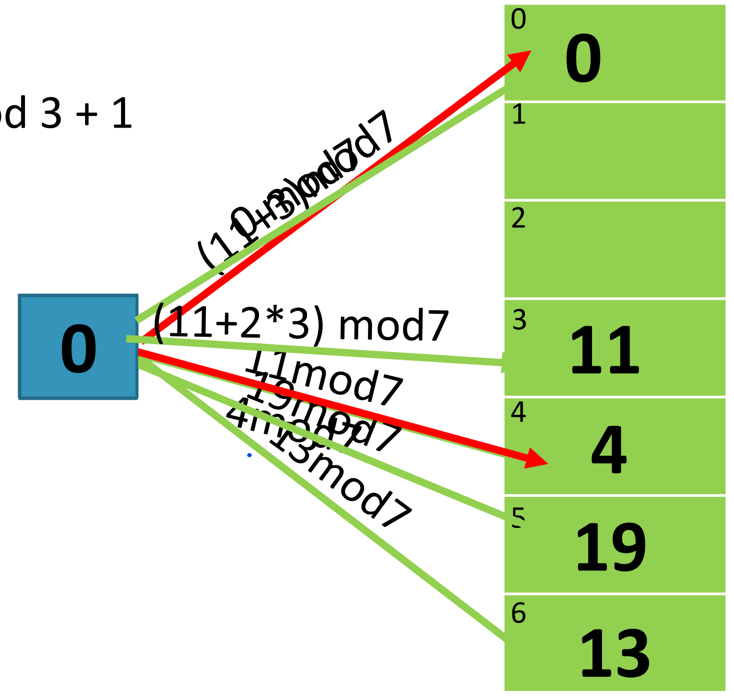
$k \in \{0, 4, 13, 19, 11\}$

0 4 6 5 | 4

$0 = 3(1) + 11 \bmod 7$

$3 = 3(2) + 11 \bmod 7$

D(11) = 11 mod 3 + 1



Time Complexity

Linear Probing

- Successful Search: $\frac{1}{2}(1 + \frac{1}{1-\alpha})$
- Unsuccessful Search: $\frac{1}{2}(1 + \frac{1}{1-\alpha})$

Double Hashing

- Successful Search: $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$
- Unsuccessful Search: $\frac{1}{1-\alpha}$

*Proof can be found in The Art of Computer Programming by Knuth Donald (1973)

Delete A Key Under Open Addressing

- Leave the deleted key in the table
- Make a marker indicating that it is deleted
- Overwrite it when a new key is inserted to the slot
- May need to do a “garbage collection” when a large number of deletions are done
  - To improve the search time

Rehashing: Expanding the Hash Table

- As α increases, the time complexity also increases

Solution:

- Increase the size of hash table (doubled)
- Rehash all keys into new larger hash table

Summary

- Exhaustive Algorithm: Sequential Search: O(n)
- Decrease-and-conquer Algorithm: Binary Search: $O(\log_2 n)$
- Data Structures: Hashing
    - Closed Hashing: Separate Chaining: O(α) on average
    - Open Hashing
        - Linear Probing
        - Quadratic Probing
        - Double Probing
    - Delete keys
    - Rehashing