CSC2040

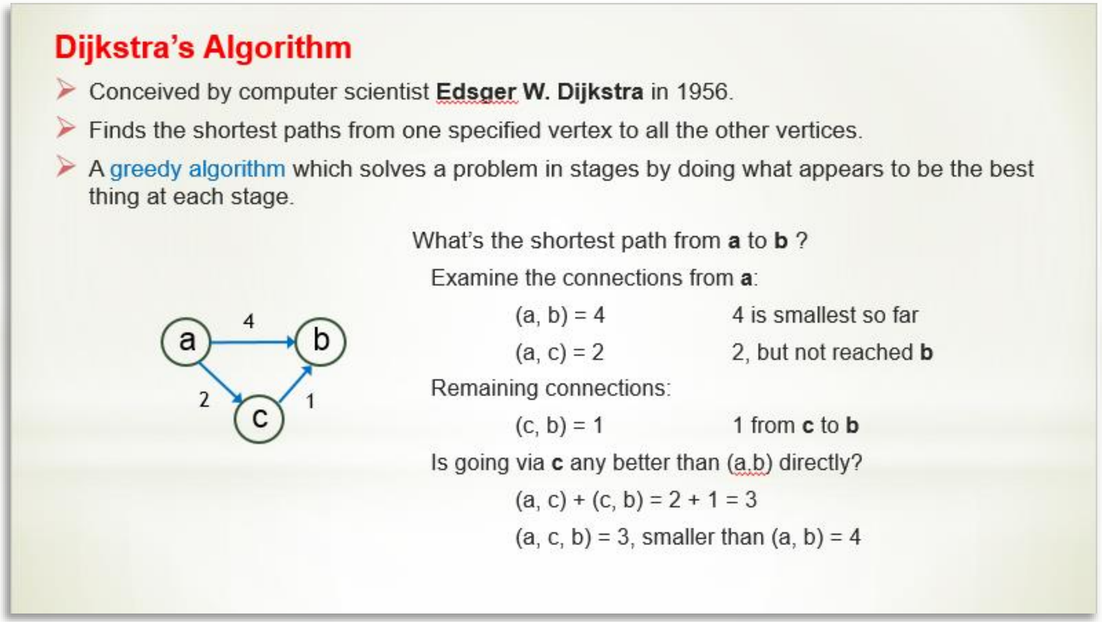
Data Structures, Algorithms and Programming Languages

Hash Tables

Data Structures, Algorithms and Programming Languages

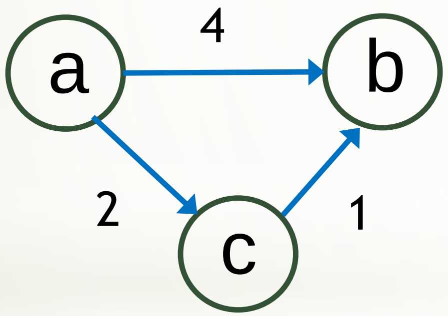
Last Time:

Graphs, Djikstra’s Algorithm (shortest path)



Dijkstra's Algorithm

- Conceived by computer scientist **Edsger W. Dijkstra** in 1956.
- Finds the shortest paths from one specified vertex to all the other vertices.
- A greedy algorithm which solves a problem in stages by doing what appears to be the best thing at each stage.



What's the shortest path from **a** to **b** ?

Examine the connections from **a**:

(a, b) = 4 — 4 is smallest so far

(a, c) = 2 — 2, but not reached **b**

Remaining connections:

(c, b) = 1 — 1 from **c** to **b**

Is going via **c** any better than (a,b) directly?

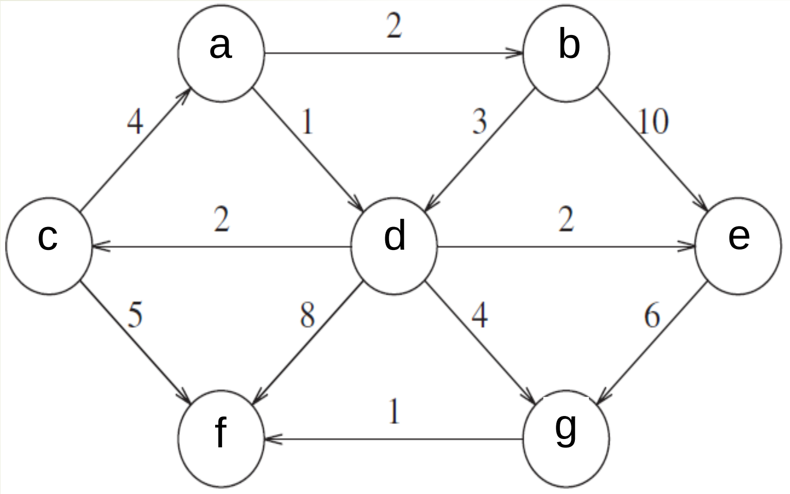
(a, c) + (c, b) = 2 + 1 = 3

(a, c, b) = 3, smaller than (a, b) = 4

Dijkstra's Algorithm

- A more complex example.

As we explore each path, the distance value for f (from a) gets lower (we find shorter paths)

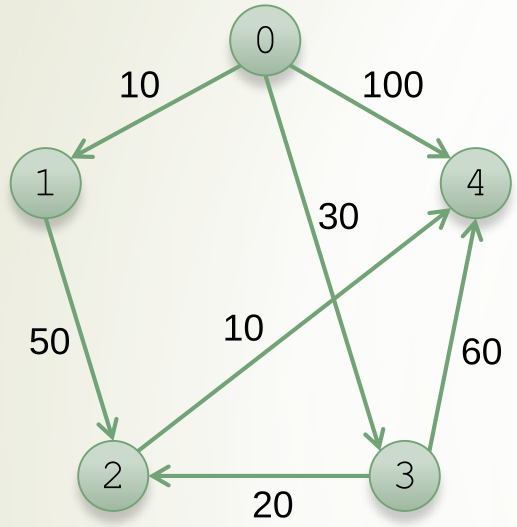


vertex	initially	Unprocessed vertex u with smallest distance and its adjacent vertices v as new destinations					
		d	b	c	e	g	f
		c,e,f,g	d, e	f	g	f	
a (start)	0 (a)						0 (a)
b	2 (a)						2 (a)
c	infinity	3 (d)					3 (d)
d	1 (a)						1 (a)
e	infinity	3 (d)					3 (d)
f	infinity	9 (d)		8 (c)		6 (g)	6 (g)
g	infinity	5 (d)					5 (d)

if dis(a,u) + weight(u,v) < dis(a,v)
update:
  dis(a,v) = dis(a,u) + weight(u,v)
  Parent(v) = u

The last column shows the lowest accumulative weights from `a` to each specific vertex, with the immediate parents in parentheses for backtracking the path (e.g., for retrieving the path from `a` to `f`).

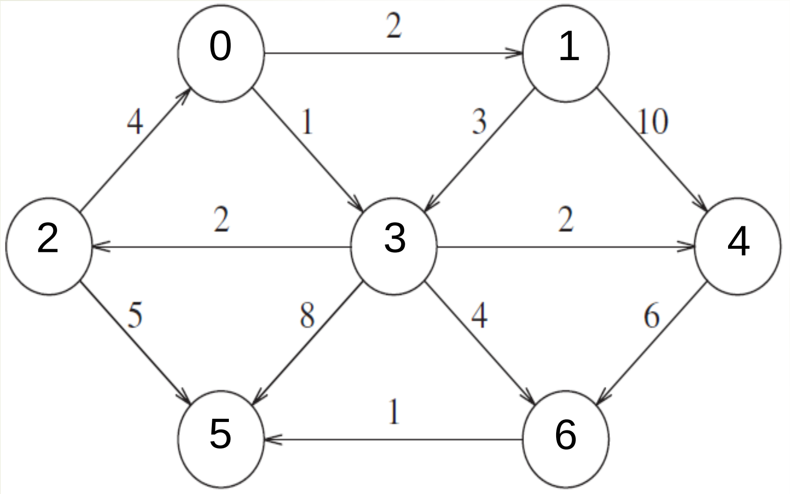
Test Dijkstra's Algorithm



weighted_digraph1.txt

```
5
d
0 1 10
0 3 30
0 4 100
1 2 50
2 4 10
3 2 20
3 4 60
```

```
Provide a graph definition file name: ..\..\weighted_digraph1.txt
Specifiy destination vertex v: 4
The shortest distance from 0 is: 60
The shortest path from 0 is: 0 3 2 4
```



weighted_digraph2.txt

```
7
d
0 1 2
0 3 1
1 3 3
1 4 10
2 0 4
2 5 5
3 2 2
3 4 2
3 5 8
3 6 4
4 6 6
6 5 1
```

```
Provide a graph definition file name: ..\..\weighted_digraph2.txt
Specifiy destination vertex v: 5
The shortest distance from 0 is: 6
The shortest path from 0 is: 0 3 6 5
```

Hash Tables

- **Hash Tables**
  - **What are Hash Tables?**
  - **Hash Function**
  - **Techniques to resolve collisions**

What are hash tables?

- Also known as Associative storage or Content-addressable memory
- Use the content to be stored to decide its storage location
- In a dictionary, we find the definition of a word on a page dictated by the word itself (it's letters and the alphabetical order relative to other words).

- Example:

Store phone records to facilitate search (e.g., using name to find the number), insertion etc.

"Tom 84274" "Jack 21298" "Harry 69496" "Sam 82879" "Pete 24840" ……

- Array – $O(N)$ for search, insertion, deletion, N is the number of records
- Linked list, Stack, Queue – $O(N)$ for search, $O(1)$ for insertion, deletion
- Binary tree – $O(\log_2 N)$ for all three operations

What are hash tables?

- What if we directly use the content – name – as the storage address?

Address		#10005		#10007			#10010	#10011			#10014
Memory		84274		82879			24840	21298			69496
“look-up name”		“Tom”		“Sam”			“Pete”	“Jack”			“David”

- Different names should be represented (and their data stored) at different locations.

- So, given an enquiry - **name**, we should be able to retrieve its associated information – **phone number**, or to insert the information, in constant time **O(1)**.

- How do we reference a physical memory location using actual content?

What are hash tables?

- This memory system is called a **Hash Table** – heavily used in Information Retrieval.

Address		#10005		#10007			#10010	#10011			#10014
Memory		84274		82879			24840	21298			69496
"look-up name"		"Tom"		"Sam"			"Pete"	"Jack"			"David"

- Each table cell contains a **Key-Value(s)** pair (e.g., a name-phone number pair, a keyword-article pair etc.).

- The **Key** is used to address the memory to retrieve or insert the **Value(s)**.

- We find the memory location by applying a **Hash Function** to the **Key**.

Hash function

- How to convert a key to an index of the table (i.e., address of the memory)?

- For integer keys, a simple strategy:

$$index(Key) = Key \% TableSize$$

TableSize is the total number of cells in the hash table.

- The modulo operation forces $0 \leq index(Key) < TableSize$ for any Key value.
- So if TableSize was 10, the index will be between 0 and 9, no matter what value Key is:

Key = 5, index = 5

Key = 10, index = 0

Key = 3354, index = 4

Hash function

- TableSize needs to be chosen carefully.

- Example:

  insert keys 89, 18, 49, 58, 69 into a hash table of TableSize = 10.

  9 = 89 % 10
  8 = 18 % 10
  9 = 49 % 10
  8 = 58 % 10
  9 = 69 % 10

	index	
	0	
	1	
	2	
	3	
	4	
	5	
	6	
	7	
58 →	8	18
69→49 →	9	89

- Two or more keys hashing to the same index is known as a **collision**.

Hash function

- It is a good idea to choose a **prime** TableSize.

- Example:

  insert keys 89, 18, 49, 58, 69 into a hash table of TableSize = 7.

  5 = 89 % 7
  4 = 18 % 7
  0 = 49 % 7
  2 = 58 % 7
  6 = 69 % 7

index	
0	49
1	
2	58
3	
4	18
5	89
6	69

- A prime TableSize may generate more random, evenly distributed hash indices.

Hash function

- Another requirement for the hash function is that it is simple and fast to compute.

- For `char`-string keys, `key`, of length `keyLen`, the following is a good hash function:

$$\text{index}(\text{key}) = \left(\sum_{i=0}^{\text{keyLen}-1} \text{key}[\text{keyLen} - i - 1] \times 37^{i}\right) \% \text{ TableSize}$$

for example:

$$\text{index}(\text{"key"}) = (\text{'y'} + \text{'e'} \times 37 + \text{'k'} \times 37^2) \% 7$$

```
unsigned int hash_index = 0;
for (int i = 0; i < keyLen; i++)
   hash_index = 37 * hash_index + key[i];
hash_index %= TableSize;
```

Used in Java String class

HashTable.h / .cpp

```cpp
#ifndef HASHTABLE_H
#define HASHTABLE_H

// hash table storing class X objects using linear probing
template <class X>
class HashTable {
public:
    // constructor sets the hash table size & load threshold
    HashTable(int table_size, double load_threshold = 0.75);
    // destructor
    ~HashTable() { for(int i = 0; i < Table.size(); i++) if (Table[i]) delete Table[i]; }

    // search for object a in the table
    size_t find(X& a);  // size_t = unsigned int
    // insert new object a in the table, return true if done
    bool insert(X& a);

private:
    // the hash table & number of objects stored
    vector<X*> Table;
    size_t num_x;
    // maximum load threshold
    double LOAD_TH;
};

template <class X>
HashTable<X>::HashTable(int table_size, double load_threshold)
{
    for (int i = 0; i < table_size; i++) Table.push_back(NULL);
    num_x = 0;
    LOAD_TH = load_threshold;
}
```

```cpp
template <class X>
size_t HashTable<X>::find(X& a)
{
    // calculate the hash index
    size_t index = a.hash_index() % Table.size();
    // search, find index of matching key or the 1st empty slot
    while (Table[index] != NULL && Table[index]->get_key() != a.get_key())
        index = (index + 1) % Table.size();
    // retrieve matching value to a if found
    if (Table[index] != NULL) a.set_value(Table[index]->get_value());

    return index;
}

template <class X>
bool HashTable<X>::insert(X& a)
{
    // calculate the load factor of the table
    double load_factor = (double)num_x / (double)Table.size();
    if (load_factor > LOAD_TH) {
        // replace the following return by rehashing - practical work
        return 0;
    }

    // search a in the able
    size_t index = find(a);
    // not found, create a new entry in the table
    if (Table[index] == NULL) {
        Table[index] = new X(a);
        num_x++;
        return 1;
    }

    // object already in table, do nothing
    return 0;
}
#endif
```

* CSC2040

Data Structures, Algorithms and Programming Languages

Code Demo:

`“HashTable.cpp”`

Hash function

- Example: insert "Tom", "Sam", "Pete", "Jack", "David" with TableSize = 5.

$$\text{index}(\text{"Tom"}) = (\text{'m'} + \text{'o'} \times 37 + \text{'T'} \times 37^2) \% 5$$
$$= (109 + 111 \times 37 + 84 \times 37^2) \% 5$$
$$= 119212 \% 5 = 2$$

$$\text{index}(\text{"Sam"}) = (\text{'m'} + \text{'a'} \times 37 + \text{'S'} \times 37^2) \% 5 = 0$$

$$\text{index}(\text{"Pete"}) = (\text{'e'} + \text{'t'} \times 37 + \text{'e'} \times 37^2 + \text{'P'} \times 37^3) \% 5 = 2$$

$$\text{index}(\text{"Jack"}) = (\text{'k'} + \text{'c'} \times 37 + \text{'a'} \times 37^2 + \text{'J'} \times 37^3) \% 5 = 0$$

$$\text{index}(\text{"David"}) = (\text{'d'} + \text{'i'} \times 37 + \text{'v'} \times 37^2 + \text{'a'} \times 37^3 + \text{'D'} \times 37^4) \% 5 = 1$$

	index	
Jack →	0	Sam
	1	David
Pete →	2	Tom
	3	
	4	

- We have collisions and need to resolve them.

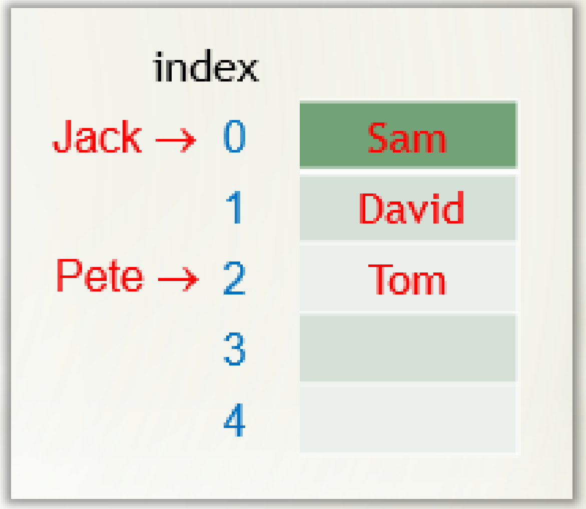
Linear probing

- There are several methods which we can use to find alternative cells in a collision
- **Linear probing** is used to quickly find another cell near to the original

- If a cell is occupied, increment the hash value until an empty cell is found.

- The new hash value is calculated from the old hash value:

  index_new(key) = {index_old(key) + 1} % TableSize



Linear probing

- Example: insert "Tom", "Sam", "Pete", "Jack", "David" with TableSize = 5.

index("Tom") = 119212 % 5 = 2

index("Sam") = 117325 % 5 = 0

index("Pete") = 4194902 % 5 = 2

Resolve: index_new("Pete") = {2 + 1} % 5 = 3 % 5 = 3

index("Jack") = 3884885 % 5 = 0

Resolve: index_new("Jack") = {0 + 1} % 5 = 1

index("David") = 132521816 % 5 = 1

Resolve: index_new("David") = {1 + 1} % 5 = 2 % 5 = 2

Resolve: index_new("David") = {2 + 1} % 5 = 3 % 5 = 3

Resolve: index_new("David") = {3 + 1} % 5 = 4 % 5 = 4

index	
0	Sam
1	Jack
2	Tom
3	Pete
4	David

Reducing collision by rehashing

- More empty cells, less probable to collide
- Table load factor (percentage of occupancy) can be defined as:

  load_factor = (double) num_keys_stored / (double) TableSize

- If load_factor > LOAD_THRESHOLD (prechosen "near limit", e.g., 0.75), consider rehashing by using a larger table
- Rehashing algorithm:
  - Allocate a new hash table about twice as big the original table
  - Reinsert each old element into the new hash table using the new hash indices
  - Delete the old table
  - Start to reference the new table instead of the old table

Rehashing

Example: store "Tom", "Sam", "Pete", "Jack", "David"

index(key)
$= \left(\sum_{i=0}^{keyLen-1} key[keyLen - i - 1] \times 37^{i}\right) \% \ 5$

	index	
Jack →	0	Sam
	1	David
Pete →	2	Tom
	3	
	4	

Old table (size 5)

Re-insert old items using new indices

index(key)
$= \left(\sum_{i=0}^{keyLen-1} key[keyLen - i - 1] \times 37^{i}\right) \% \ 11$

New table (size 11)

0	
1	
2	
3	
4	Jack
5	Tom
6	
7	
8	Pete
9	David
10	Sam

load_factor = 3 / 5 = **0.6**

load_facor = 5 / 11 ≈ **0.45**

Linear Probing to Store Arbitrary Key-Value(s) Map Class Objects

```cpp
#ifndef HASHTABLE_H
#define HASHTABLE_H

// hash table storing class X objects using linear probing
template <class X>
class HashTable {
public:
    // constructor sets the hash table size & load threshold
    HashTable(int table_size, double load_threshold = 0.75);
    // destructor
    ~HashTable() { for(int i = 0; i < Table.size(); i++) if (Table[i]) delete Table[i]; }

    // search for object a in the table
    size_t find(X& a);  // size_t = unsigned int
    // insert new object a in the table, return true if done
    bool insert(X& a);

private:
    // the hash table & number of objects stored
    vector<X*> Table;
    size_t num_x;
    // maximum load threshold
    double LOAD_TH;
};

template <class X>
HashTable<X>::HashTable(int table_size, double load_threshold)
{
    for (int i = 0; i < table_size; i++) Table.push_back(NULL);
    num_x = 0;
    LOAD_TH = load_threshold;
}
```

```cpp
template <class X>
size_t HashTable<X>::find(X& a)
{
    // calculate the hash index
    size_t index = a.hash_index() % Table.size();
    // search, find index of matching key or the 1st empty slot
    while (Table[index] != NULL && Table[index]->get_key() != a.get_key())
        index = (index + 1) % Table.size();
    // retrieve matching value to a if found
    if (Table[index] != NULL) a.set_value(Table[index]->get_value());

    return index;
}

template <class X>
bool HashTable<X>::insert(X& a)
{
    // calculate the load factor of the table
    double load_factor = (double)num_x / (double)Table.size();
    if (load_factor > LOAD_TH) {
        // replace the following return by rehashing - practical work
        return 0;
    }

    // search a in the able
    size_t index = find(a);
    // not found, create a new entry in the table
    if (Table[index] == NULL) {
        Table[index] = new X(a);
        num_x++;
        return 1;
    }

    // object already in table, do nothing
    return 0;
}
#endif
```

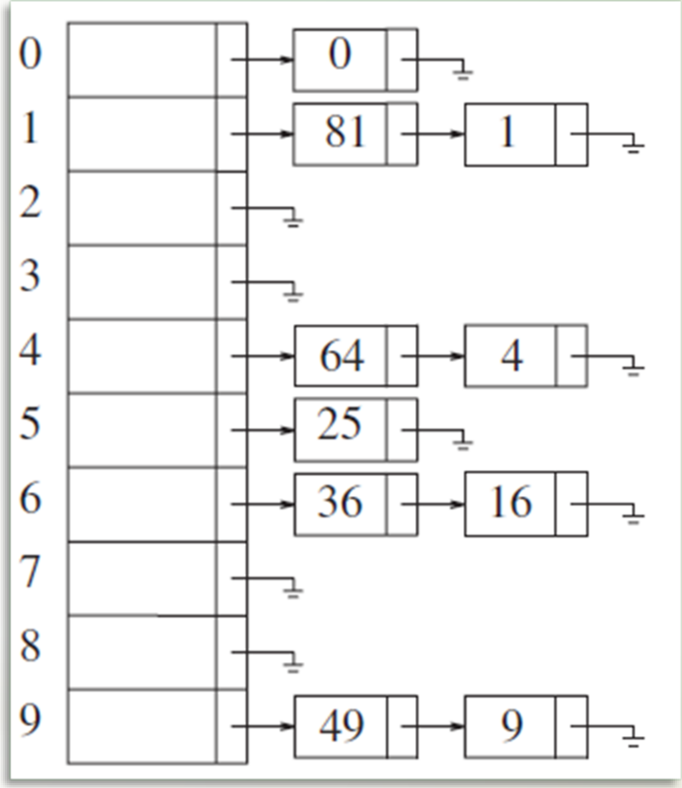
Test by Storing PhoneDir Objects

```cpp
// a class of phone records
class PhoneDir {
public:
    PhoneDir(string name, int number = -1)
        : name(name), number(number) {};

    string get_key() { return name; }
    int get_value() { return number; }
    void set_value(int num) { number = num; }

    size_t hash_index(); // return hash index of key: name

private:
    string name;    // key
    int number;     // value
};

size_t PhoneDir::hash_index()
{
    size_t hash_index = 0;
    for (int i = 0; i < name.size(); i++) {
        char c = name[i];
        hash_index = 37 * hash_index + c;
    }
    return hash_index;
}
```

```cpp
int main()
{
    // store phone records in hash table with size 11
    HashTable<PhoneDir> HTable(11);
    HTable.insert(PhoneDir("Tom", 123456));
    HTable.insert(PhoneDir("John", 346834));
    HTable.insert(PhoneDir("Jack", 347980));
    HTable.insert(PhoneDir("Clare", 328709));
    HTable.insert(PhoneDir("Razel", 335566));

    // serach using name for phone number over the hash table
    char yn = 'y';
    do {
        cout << "Whose number are you looking for? ";
        string name; cin >> name;

        PhoneDir enquiry(name);
        cout << "index = " << HTable.find(enquiry);
        cout << ", name = " << enquiry.get_key();
        cout << ", number = " << enquiry.get_value() << endl;

        cout << "Another (y/n)? "; cin >> yn;
    } while (yn == 'y');

    return 0;
}
```

Separate chaining

- Keep a list of all elements that hash to the same location.
- List is built as collisions occur.
- Each collision item is added to list, rather than rehashing to another location



- Example:

  insert the first 10 perfect squares

  0, 1, 4, 9, 16, 25, 36, 49, 64, 81

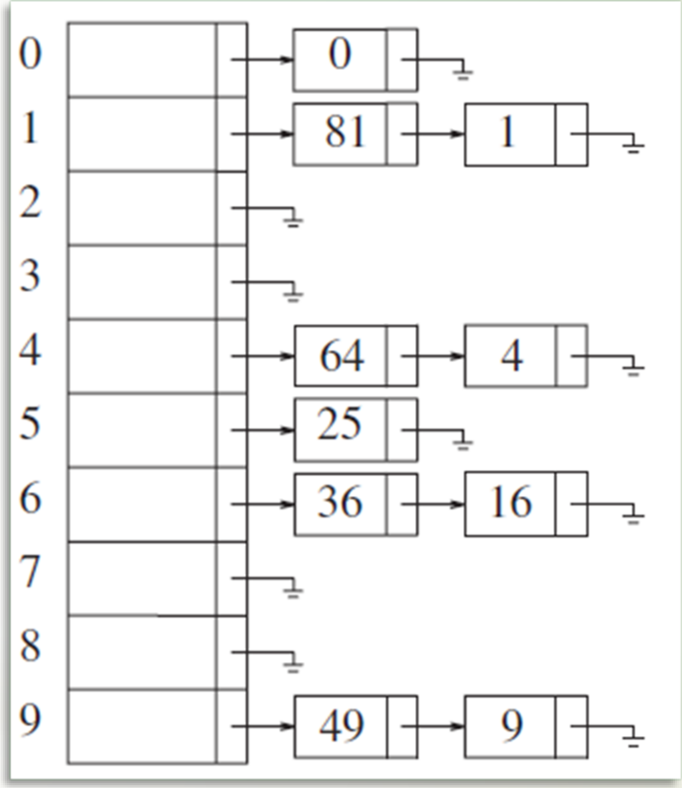
  with hash function:

  index(key) = key % 10 with TableSize = 10

  (not prime, for indication only).

Separate chaining

- Search / Insertion operation:
  - Use hash function to determine which list to traverse
  - Search the appropriate list for the key
  - New key can be inserted either at the front or at end of list



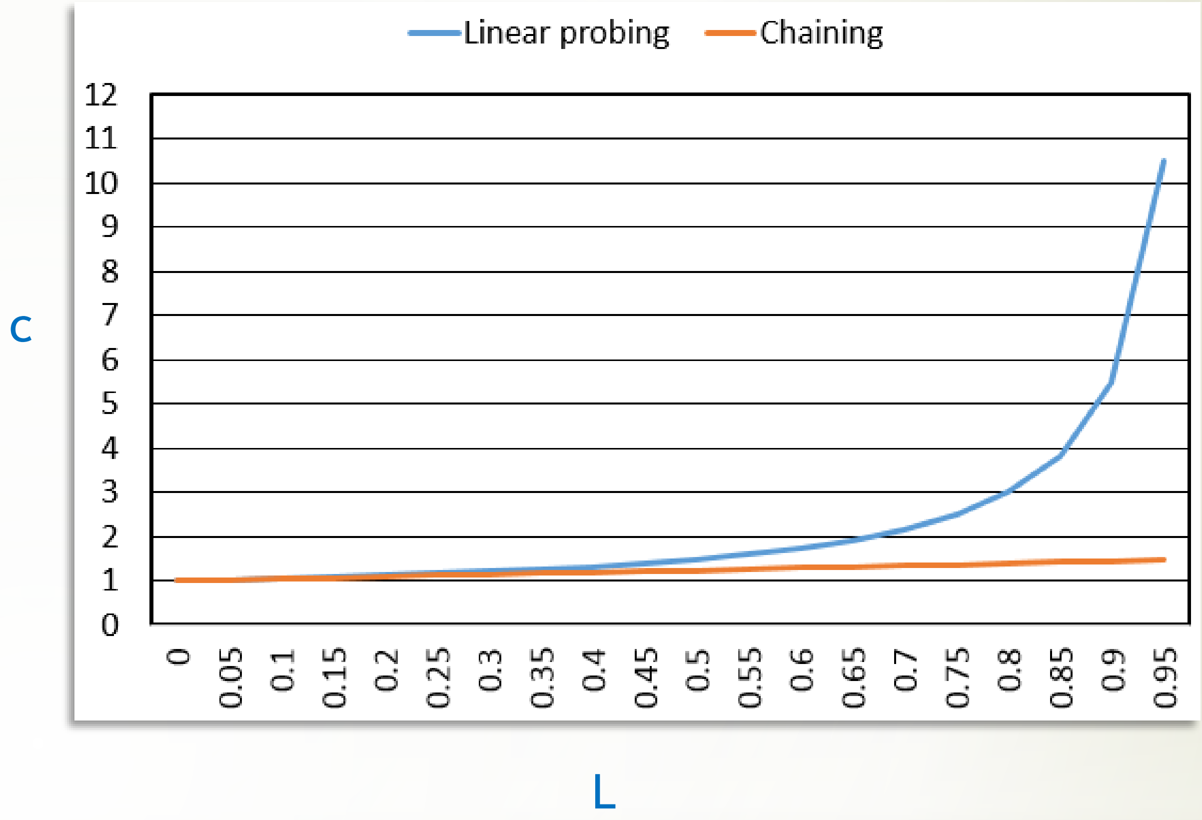
- Advantages:
  - Only items having the same hash indices will be examined
  - Store more items than the number of table indices

Performance

Expected number of comparisons for finding an item (L is the load factor):

Linear probing: $c = \frac{1}{2}\left(1 + \frac{1}{L-1}\right)$

Chaining: $c = 1 + \frac{L}{2}$



- Chaining performs much better than Linear probing when the Load factor approaches 1.0 (ie. the hash table gets full).

Other techniques

- Another expression of linear probing:

  The i'th probe index(key) = {start_index(key) + i} % TableSize

- Quadratic probing:

  The i'th probe index(key) = {start_index(key) + i^2} % TableSize

- Double hashing:

  The i'th probe index(key) = {start_index(key) + i×$hash_2$(key)} % TableSize

  ($hash_2$ is a second hash function applied to key)

CSC2040

Data Structures, Algorithms and Programming Languages

***Next Time*:**

Exam Revision