CSCI 136:
Data Structures
and
Advanced Programming

Lecture 29

Hashtables and Graphs

Instructor: Kelly Shaw

Williams

Topics

Hash collisions

Graphs

Your to-dos

1. Read **before Mon**: *Bailey*, Ch. 16.4.
2. Lab 9 (partner lab), **due Tuesday 11/29 by 10pm**.
3. No quiz this week!

Pigeonhole principle

Linear probing

Downside: values **cluster** around **collisions**.

collision!

A								
					Dan -11	Dirk 20	Don 6	
	0	1	2	3	4	5	6	7

```
key: “Ed”, value: 7
      index(“Ed”) → ~~5~~ ~~6~~ 7
```

Likelihood of collisions grows as cluster grows.

Our table is **still half empty**! This is **bad**!

Linear probing

$h(\text{key}) + c \times i$

Changing c can mitigate clustering.

E.g., $c = 2$.

collision!

A								
					Dan -11		Dirk 20	
	0	1	2	3	4	5	6	7

```
key: “Dan”, value: -11
      index(“Dan”) → 4

   key: “Dirk”, value: 20
         index(“Dirk”) → ~~4~~ 6
```

Linear probing

Changing c can mitigate clustering.

But it can also **reduce the table’s capacity**.

A								
	Don 6		Doug 22		Dan -11		Dirk 20	
	0	1	2	3	4	5	6	7

Deb 101 **???**

```
key: “Dan”, value: -11
      index(“Dan”) → 4

   key: “Dirk”, value: 20
         index(“Dirk”) → ~~4~~ 6
```

Now we are only using *1/c* buckets!

Linear probing: deletion

Deletion is problematic when using linear probing.

A								
	Don 6		Doug 22		Dan -11		Dirk 20	
	0	1	2	3	4	5	6	7

`delete(“Dan”)`

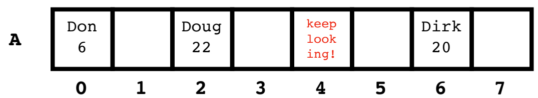
`lookup(“Dirk”)`

We can no longer find `Dirk`.

Linear probing: deletion

Deletion is problematic when using linear probing.

Addressed by leaving a **sentinel** value at deleted location.



`delete("Dan")`

`lookup("Dirk")`

Doesn't reclaim space until all colliding entries deleted.

External chaining

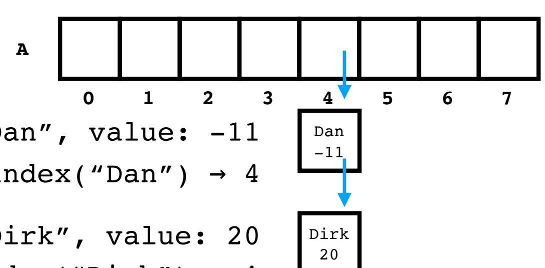
External chaining is a method for resolving collisions in a hash table. Collisions are resolved by storing **more than one value in a bucket**, e.g., using a **list**.

External chaining

Same **bad hash function**:

```
((int) key.charAt(0)) % A.length
```

collision!

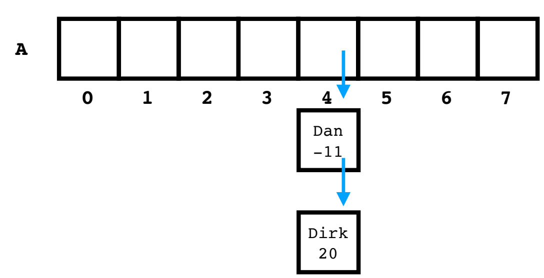


key: "Dan", value: -11
index("Dan") → 4

key: "Dirk", value: 20
index("Dirk") → 4

External chaining: deletion

Deletion is trivial.



Hash Table Expansion

When a hash table **fills up**, we should **expand**, just as with a Vector. But there are some problems…

Hash Table Expansion

Hash tables rely on **the size of the underlying array** to do the indexing. Recall:

```
int index(K key) {
  return abs(h(key) % A.length);
}
```

When a hash table expands, we usually address this by **rehashing** all elements during a copy. **Why is this OK?**

Hash Table Expansion

Another issue: hash table performance **degrades severely** as it **fills up**.

Recall that we can have an **effectively full** hash table even when there is actually space.

$$h(\texttt{key}) + c \times i$$

where $c = 2$

A	0	1	2	3	4	5	6	7
	Don 6		Doug 22		Dan -11		Dirk 20	

Deb 101 **???**

Hash Table Expansion

Therefore, we resize **before** the table is likely to be full.

Let `n` be the **number of elements** stored in a hash table.

Let `m` be the **number of buckets**.

Load factor = `n / m`

When the desired load factor is exceeded, the array is **expanded**.

Hash Table Expansion

There are **two ways** to find a **good load factor**.

1. **Careful analysis of the probability** of attempting to insert more than one element into the same bucket, combined with a preference for acceptable average slowdown.
2. **Empirical measurement**, combined with a preference for acceptable average slowdown.

A load factor between **0.7** and **0.8** is generally thought to be an acceptable default.

Complexity

Method	Successful	Unsuccessful
Linear probes	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
Double hashing	$\frac{1}{\alpha}\ln\frac{1}{(1-\alpha)}$	$\frac{1}{1-\alpha}$
External chaining	$1+\frac{1}{2}\alpha$	$\alpha+e^{-\alpha}$

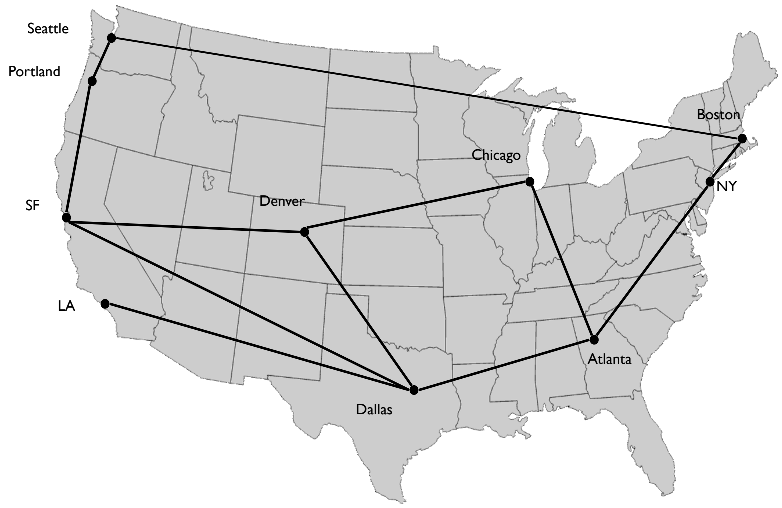
Figure 15.11 Expected theoretical performance of hashing methods, as a function of α, the current load factor. Formulas are for the number of association compares needed to locate the correct value or to demonstrate that the value cannot be found.

Graphs

Tons of Applications

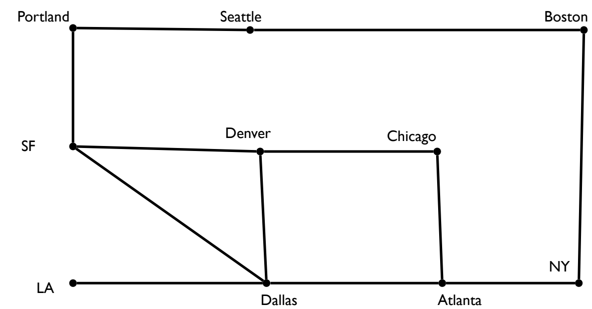
Nodes = subway stops; Edges = track between stops

Tons of Applications



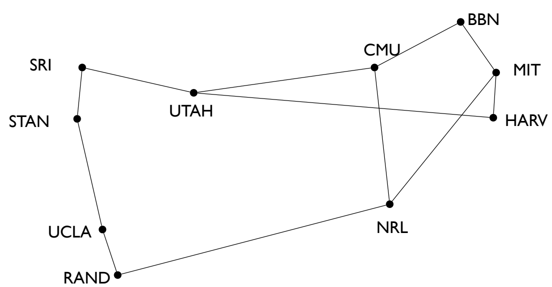
Nodes = cities; Edges = rail lines connecting cities

Tons of Applications



Note: A connection in a graph matters, but not the location of a node.

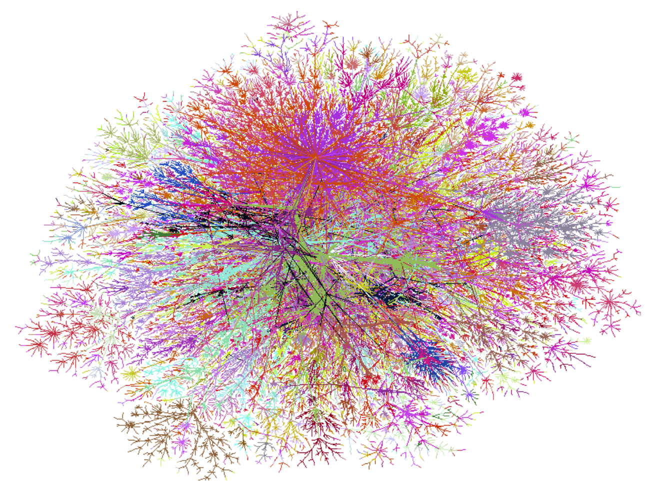
Tons of Applications



Any guesses as to what this is?

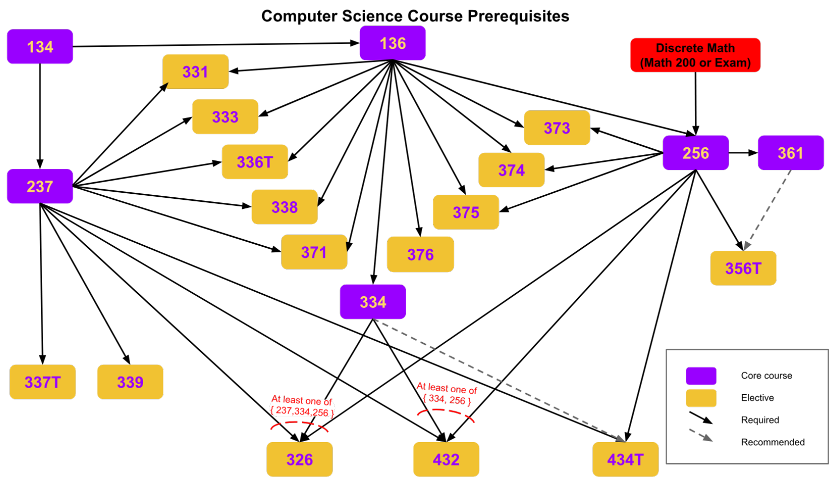
(The Internet, circa 1972.)

Tons of Applications

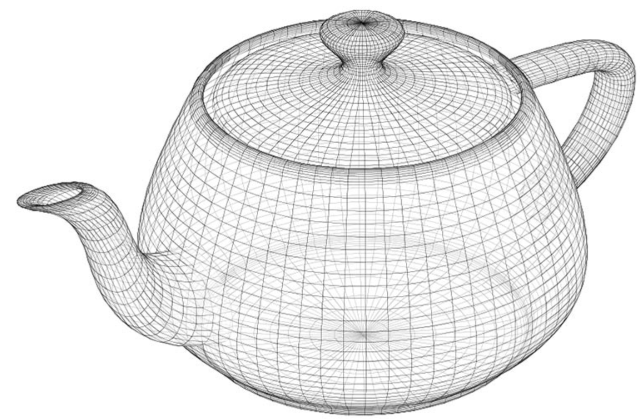


(The Internet, circa 1998.)

Tons of Applications

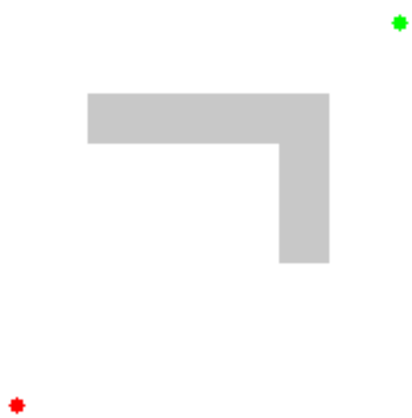


Tons of Applications



A “wireframe” model

Dijkstra’s Algorithm



Undirected graph ADT

An **undirected graph** **G** is an abstract data type that consists of two sets:

- a set **V** of **vertices** (or **nodes**), and
- a set **E** of **undirected edges**.

Undirected graph ADT

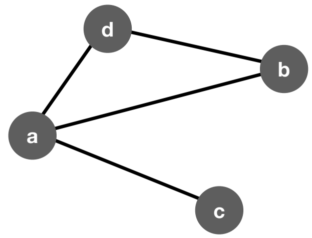
A graph can be used to represent any structure in which pairs of elements are "**related**."

In an undirected graph, **arbitrary data** can be **associated** either with a vertex, an edge, or both.

For example: vertex data = city; edge data = distance.

Undirected graphs are a **good choice** when **a relation is symmetric**. E.g., the distance from Williamstown to Boston is the same as the distance from Boston to Williamstown.

Undirected graph



G =(V, E)

Directed graph ADT

A **directed graph G** is an abstract data type that consists of two sets:

- a set **V** of **vertices** (or **nodes**), and
- a set **E** of **directed edges**.

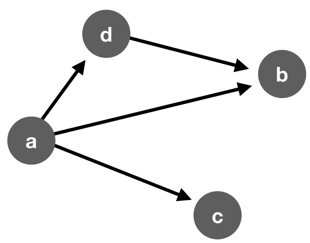
Directed graph ADT

In a directed graph, data can be associated either with a vertex, an edge, or both.

Example: vertex data = people; edge data = "loves".

A **directed graph** is a **good choice** when **relations between vertices are not symmetric**.

Directed graph



G =(V, E)

Walking a graph

A **walk from u to v** in a graph **G = (V, E)** is an alternating sequence of vertices and edges

$$u = v_0,\ e_1,\ v_1,\ e_2,\ v_2,\ \ldots,\ v_{k-1},\ e_k,\ v_k = v$$

such that $e_i = \{v_i, v_{i+1}\}$ for $i = 1, \ldots, k$

- A walk **starts** and **ends** with a **vertex**.
- A walk can travel over any edge and any vertex any number of times.
- If **no edge** appears more than once, the walk is a **path**.
- If **no vertex** appears more than once, the walk is a **simple path**.

Walking in circles

A **closed walk** in a graph **G = (V, E)** is a walk

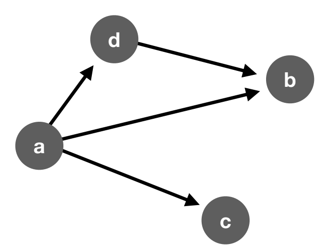
$$v_0,\ e_1,\ v_1,\ e_2,\ v_2,\ \ldots,\ v_{k-1},\ e_k,\ v_k$$

such that $v_0 = v_k$

- A **circuit** is a **path** where $v_0 = v_k$ (no repeated edges)
- A **cycle** is a **simple path** where $v_0 = v_k$ (no repeated vertices except v_0)
- The **length** of a walk is the number of edges in the sequence.

Walking on graphs vs digraphs

In a **directed graph**, a walk can only follow the **direction of the arrows**.



There is **no directed walk** from **b** to **a**.

Useful theorems

(about undirected graphs)

- If there is a **walk** from **u** to **v**, then there is a **walk** from **v** to **u**.
- If there is a **walk** from **u** to **v**, then there is a **path** from **u** to **v** (and from **v** to **u**).
- If there is a **path** from **u** to **v**, then there is a **simple path** from **u** to **v** (and **v** to **u**).
- Every **circuit** through **v** contains a **cycle** through **v**.
- Not every **closed walk** through **v** contains a **cycle** through **v**.

Recap & Next Class

Today:

Graphs

Next class:

Graph operations

Graph representations