More on hash tables.
Hash tables

• In the previous lecture we discussed how hash tables enable $O(1)$-time add/find/remove operations in the average case.

• The trade-off necessary to achieve $O(1)$ time was the extra space needed to store a large, sparse array.

• Hash tables consist of a large array, plus a hash function to distribute the user’s data “evenly” across the array.

  • The input to the hash function is the key, and its output is an index into the hash table’s array.

  • Simple example:

    ```c
    int hashFunction (int key) {
        return key % M;  // M is size of _array
    }
    ```
Keys and hash codes

• So far we have assumed that the key is always an integer, e.g., studentID.

• But what if wanted the student’s fullName (i.e., a string) to be the key?

• Java gives us additional flexibility in how keys are converted into array indices.

• Instead of hashing the key directly, we instead hash the key’s hash code.

• A hash code is a way of describing any object o using just a primitive int.
Hash code examples

• Suppose our key is:

  • A single character $c$:
    
    • We could convert $c$ into its ASCII value, which is an integer (from 0-127).

  • A String $s$ of characters:
    
    • We could convert each $c$ in $s$ to its ASCII value, and then add them together.

• An image $im$:

  • We could add together the pixel values across all three (R,G,B) channels.

Note: these are just hypothetical examples, not necessarily how Java actually implements hash codes!
Keys and hash codes

• The hash code serves as an “intermediary value” between the object’s key and its assigned array index in a hash table.

• Instead of just

  _array[hashFunction(key)]

  we instead write:

  _array[hashFunction(key.hashCode())];

• The object.hashCode() method converts any Java object into an integer.
hashCode()  

• In Java, all objects support the `hashCode()` method, defined in class `Object`.  

• By default, `hashCode()` simply returns the object’s location (address) in memory.  

• A subclass `A` can override the default implementation when a customized implementation would improve performance, i.e., result in fewer collisions, or when `A` overrides the `equals(o)` method (more later).
hashCode()

- In Java, the `hashCode()` method must uphold two properties:
  1. **Deterministic** -- multiple subsequent calls to `hashCode()` on the same object `o` must return the same value.
  2. Otherwise, `hashFunction(key.hashCode())` would map into a different array index -- and the hash table wouldn’t be able to find `o`.

```java
_array[hashFunction(o.key.hashCode())] = o; // Add
...
return _array[hashFunction(o.key.hashCode())]; // Find
```
hashCode()

2. **Consistent across equal instances** -- if `o1.equals(o2)`, then `o1.hashCode()` *must* equal `o2.hashCode()`:

```java
final String s1 = "hello";
final String s2 = new String("hello"); // Distinct copy
int hashCode1 = s1.hashCode();
int hashCode2 = s2.hashCode(); // Must equal hashCode1
```

- This means that if class A overrides the `equals()` method, then it must also override `hashCode()`.

- Calling `hashCode()` is sometimes faster than calling `equals(o)`; hence, `hashCode()` offers a “fast check” that objects `o1` and `o2` might be equal:
  - if `o1.hashCode() != o2.hashCode()`, then `o1 cannot equal o2`. 
hashCode()

• In addition, it is desirable for hashCode() to have:

3. Wide distribution across instances -- hashCode() should return different values for different instances of the same class as much as possible.

• If A.hashCode() returned the same hash value for every instance o, then all objects of type A would map into the same array index. hashCode() is always the same.

    _array[hashFunction(key1.hashCode())] = o1;
    _array[hashFunction(key2.hashCode())] = o2;  // Collision
    _array[hashFunction(key3.hashCode())] = o3;  // Collision
    _array[hashFunction(key4.hashCode())] = o4;  // Collision

• This would yield terrible \(O(n)\) hash performance!
The `String` class overrides the `equals()` method so that two distinct `String` objects `s1` and `s2` whose character sequences are identical are defined to be `equal`, e.g.:

```java
String s1 = "test1";
String s2 = new String("test1");  // distinct copy

boolean isSameAddress = (s1 == s2);  // false
boolean isEqual = s1.equals(s2);  // true
```
hashCode() and equals():

Example 1

- Since s1 and s2 are equal, their hash codes must be equal as well (according to hashCode() contract):

```java
String s1 = "test1";
String s2 = new String("test1");  // distinct copy

int hashCode1 = s1.hashCode();  // 110251487
int hashCode2 = s2.hashCode();  // 110251487
boolean isSameHashCode = (hashCode1 == hashCode2);  // true
```
**hashCode() and equals()**: Example 1

- The `String.hashCode()` method is implemented in the following way:
  - If the length $n$ of $s$ is 0, then $s.hashCode()$ is 0.
  - Otherwise, $s.hashCode()$ is:
    - $s[0] \times 31^{n-1} + s[1] \times 31^{n-2} + \ldots + s[n-1]$

- This formula ensures that strings with equal contents have the same hash code.

- It also tends to “spread” the hash codes of various strings evenly over the entire range of integers (-$2^{31}$ to $+2^{31}-1$).
Hash table ADTs

• So far we’ve focused more on how a hash table is implemented *internally* and less how a user would *use* it.

• There are two different *interfaces* that a hash table ADT might offer.

• The interface varies depending on whether:
  1. Key is a field *inside* the whole record.
  2. Key is *separate* and stored *outside* the record.
Key inside the record

• In some previous examples we’ve conceptualized the key as a field within the whole object, e.g.:

```java
class Student {
    int _studentID;
    String _firstName, _lastName;
    boolean _ownsTeddyBear;
}
```

• This implementation of keys then lends itself to the following hash table interface:

```java
interface HashTable<T extends HasKey> {
    void add (T o);
    T get (T o);
}
```

where the hypothetical `HasKey` interface guarantees that `T` offers a method called `Object getKey()`.
Key inside the record

• The `add(o)` and `get(o)` methods might then be implemented as:

```java
void add (T o) {
    final Object key = o.getKey();
    _array[hashFunction(key.hashCode())] = o;
}

T get (T o) {
    final Object key = o.getKey();
    return _array[hashFunction(key.hashCode())];
}
```

Here we're assuming that each `T` offers some method `getKey()` which returns the object's key -- e.g., the `_studentID` field in Integer form.
Since every Java object offers a `hashCode()` method, we can get rid of defining the key at all:

```java
void add (T o) {
    _array[hashFunction(o.hashCode())] = o;
}

T get (T o) {
    return _array[hashFunction(o.hashCode())];
}
```

Now we just compute the hash code of `o` directly.
Key inside the record

• We can then simplify the interface of the hash table:

```java
interface HashTable<T> {
    void add (T o);
    T get (T o);
}
```

• This is the interface used in P5.

• Notice how the `add(o)` and `get(o)` methods are identical as for lists, BSTs, etc.

No longer necessary for `T` to implement some `HasKey` interface.
Key inside the record

- The user can then use the hash table as follows:

```java
class Student {
    int _studentID;
    ...
    int hashCode () {
        return _studentID;
    }
}

final hashTable<Student> students =
    new HashTable<Student>();

students.add(new Student(
    12345, "Jacky", "O’Nassis", true
));
students.add(new Student(
    9231, "Bette", "Midler", false
));

...
final Student bette = students.get(new Student(9231));
```

She has a teddy bear.

She does not.
More commonly, however, hash tables separate the key from the value.

A typical hash table interface might be:

```java
interface HashTable<K, V> {
    void put (K key, V value);
    V get (K key);
}
```

Here, we are defining two different type parameters K (for keys) and V (for values).
The user would then use the hash table in the following way:

```java
class Student {
    String _firstName, _lastName;
    boolean _hasTeddyBear;
}

final HashTable<Integer, Student> hashTable =
    new HashTable<Integer, Student>() {
        @Override
        public Student get(Integer key) {
            return super.get(key);
        }
    };

final Student jacky = hashTable.get(12345);
```
Dictionaries

- Separating keys from values is especially useful when we use a hash table as a dictionary.

- A dictionary is a data structure for storing a set of associations between keys and values.

- Each key can be associated with at most one value.
Dictionaries

• Examples:

• We can create a dictionary of English words to their meanings:

```
HashTable<String,String> englishDictionary =
    new HashTable<String,String>();
englishDictionary.put(
    "eggplant",
    "The somewhat large egg-shaped fruit of a tropical Old World plant, eaten as a vegetable."
);
```

```
String meaning = englishDictionary.get("eggplant");
```
Caches.
Caches

• Having concluded our discussion of hash tables, we can now show a useful example of combining two data structures to build a third: in this case, a cache.

• Consider a situation in which a program needs to retrieve data from a container that is slow.

• The slow speed might arise due to a long distance over which the data must travel, or to the slow data rate at which a device can deliver information.
Caches

• Examples:
  
  • A web browser downloads a webpage from an external server. Server is far away.

  • A spreadsheet program loads a file from disk. Disk is slow.

  • The CPU must read the value of a variable stored in main memory (instead of on-chip storage). RAM is slow.

• In each case, the program fetches data from secondary storage and loads it into primary storage.

• Primary storage is faster and “closer” to the user than secondary storage.

• What is “slow” in one context may be “fast” in another.
Caches

• Examples:

  • A web browser downloads a webpage from an external server.
    • **Primary storage:** computer memory (RAM) and/or disk.
    • **Secondary storage:** web server.

  • A spreadsheet program loads a file from disk.
    • **Primary storage:** computer memory (RAM).
    • **Secondary storage:** disk.

  • The CPU must read the value of a variable stored in *main memory* (instead of on-chip storage).
    • **Primary storage:** CPU registers.
    • **Secondary storage:** computer memory (RAM).
Caches

• Now, suppose that the same data X tends to be fetched from secondary storage repeatedly.

• In this case, we can save time by introducing an intermediary data container -- a cache -- that “remembers” the data fetched from secondary storage.

• A cache is a data structure that offers high-speed access to a small amount of data that must otherwise be written to/read from a slower, secondary storage container.
Caches: small and fast

• Caches are inherently fast and small:
  
  • Fast because they reside in primary storage, not secondary storage.
  
  • If they were slow, we’d forget the cache and just access secondary storage directly.
  
  • Small because they are typically more expensive than secondary storage.
  
  • If they were cheap, we’d just store everything in the cache and forget secondary storage.
Caches in action

• A user’s request to fetch data $X$ from secondary storage is “intercepted” by the cache:

• If the cache already contains $X$, then the cache returns $X$ to the user immediately.

• Fetching $X$ from secondary storage is unnecessary.

• Otherwise (cache does not contain $X$), the cache forwards the user’s request to secondary storage.

• Both read and write caches exist; here, we deal only with read caches.
Caches

User -> Cache

Fetch X.

Secondary storage
Caches

User

Cache

Secondary storage

Time

Fetch X.

Is X in cache?

No.
Caches

User

Cache

Secondary storage

Time

Fetch X.

Is X in cache?
No.

Fetch X.

Respond to request.

Monday, August 29, 2011
Caches

User → Cache

Fetch X.

Is X in cache?
No. → Fetch X.

Store X in cache.

Deliver X.

Cache → Secondary storage

Respond to request.

Monday, August 29, 2011
Caches

User

Cache

Secondary storage

Fetch X.

Is X in cache?

No.

Fetch X.

Store X in cache.

Deliver X.

Deliver X.

Respond to request.

Time
Caches

User

Cache

Secondary storage

Time

Fetch X.

Is X in cache?
No.

Fetch X.

Store X in cache.

Deliver X.

Respond to request.

Deliver X.

Fetch X.

Fetch X.
Caches

User

Cache

Is X in cache?
Yes.

Fetch X.

Store X in cache.

Deliver X.

Fetch X.

Is X in cache?
No.

Fetch X.

Deliver X.

Respond to request.

Secondary storage
Caches

User

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch X.</td>
</tr>
<tr>
<td>Deliver X.</td>
</tr>
</tbody>
</table>

Cache

Is X in cache?
No. Fetch X.
Store X in cache.
Deliver X.

Is X in cache?
Yes. Deliver X.

Secondary storage

Respond to request.

Monday, August 29, 2011
Caches: definitions

• If the user requests item $X$ from the cache, and $X$ is contained in the cache, then we have a **cache hit**.

• Otherwise, if $X$ is *not* in the cache, then we have a **cache miss**.

• $X$ must then be fetched from secondary storage.

• The size of the cache is always *finite*.

• For every cache miss: if the cache is *full*, the cache must decide which element to “forget”, i.e., **evict**.

• The choice of which data to evict can affect the cache **miss rate** (fraction of cache accesses that miss) and thereby the performance of the computer system.
Eviction policies

- The algorithm that decides which object to evict is called an **eviction policy**.

- The choice of eviction policy can make a large impact on system performance.

- An *optimal* eviction policy determines which element in the cache will not be used again for the longest period of time, and then evicts it.

- This minimizes the expected cache miss rate.

- Unfortunately, this optimal policy is rarely achievable because it’s difficult to predict which items will be needed in the future.
Least-recently-used caches

• One of the most commonly implemented eviction policies is *least-recently-used* (LRU).

• Whenever we must evict an element from the cache, we pick the least-recently-used element.

• *Justification:* It seems reasonable that an item that has not been used in a long time will continue not to be requested for a while longer.

• Empirically, LRU has shown to perform “similarly” to the *optimal* eviction policy in many practical applications.
LRU in action

- How would an LRU cache handle the following sequence of requests?
- A B A C A B B C
LRU in action

• How would an LRU cache handle the following sequence of requests?

  • A B A C A B B C
LRU in action

- How would an LRU cache handle the following sequence of requests?
- A B A C A B B C
LRU in action

• How would an LRU cache handle the following sequence of requests?

• A B A C A B B C
LRU in action

• How would an LRU cache handle the following sequence of requests?

  • A B A C A B B C

  A B was LRU.
LRU in action

• How would an LRU cache handle the following sequence of requests?

• A B A C A B B C

Cache contents

Time

A
A B
A B
A C
A C
LRU in action

• How would an LRU cache handle the following sequence of requests?

• A B A C A B B C

Cache contents

Time

A
A B
A B
A C
A C
A B
C was LRU.
How would an LRU cache handle the following sequence of requests?

- A B A C A B B C
LRU in action

• How would an LRU cache handle the following sequence of requests?

• A B A C A B B C

There were 5 cache misses out of 8 accesses; hence, cache miss rate is 0.625.
LRU Cache

• We wish to construct a Cache ADT that uses the LRU eviction policy.

• The cache will mediate access to some other, arbitrary secondary storage container.

• The user will request data by calling `Cache.get(key)` and expect the associated `value` to be returned.

• If `key` is not stored in the cache, then the cache should forward the request to the secondary storage.
LRU Cache interface

• Before designing a Java interface for the LRU cache, let’s first conceptualize how the user might access the secondary storage without the cache.

• Suppose the secondary storage has the following interface:

```java
interface Storage<K,V> {
    // Fetches and returns the data specified by key
    V get (K key);
}
```

• Here, the key might be the URL of a web page we’re fetching, and the value might be the web page itself.
Now, let’s define a Java interface for an LRU cache:

```java
// Least-recently-used (LRU) cache.
// The get(key) method should take O(1) time
// for an n-element cache.
//
// Implementing classes should offer a
// constructor with one parameter of type
// Storage that specifies the cache’s
// secondary storage.
interface LRUCache<K,V> {
    V get (K key);
}
```
LRU Cache implementation

• The LRUCache interface imposes the constraint that \texttt{get(key)} must operate in \(O(1)\) time for an \(n\)-element cache.

• Each call to \texttt{get(key)} must potentially:

1. Determine whether the desired object (specified by \texttt{key}) is stored in the cache in \(O(1)\) time.

2. If \texttt{key} is in cache, then:
   (a) Make \texttt{key} the MRU item in \(O(1)\) time.
   (b) Return the \texttt{key}'s associated \texttt{value} in \(O(1)\) time.
3. Else (key is not in cache):
   
   (a) Call \( \text{value} = \_\text{secondaryStorage}.\text{get}(\text{key}) \).
   
   • This is no problem because it is still \( O(1) \) regardless of the size of the cache \( n \).
   
   (b) Find the least-recently-used (LRU) item in \( O(1) \) time.
   
   (c) Replace the LRU item with \((\text{key}, \text{value})\), which is now the most-recently-used (MRU) item in the cache, in \( O(1) \) time.
LRU Cache implementation

- Hence, an implementation of LRUCache might look something like:

```java
class LRUCacheImpl<K,V> implements LRUCache<K,V>{
    final Storage<K,V> _secondaryStorage;
    ...

    LRUCacheImpl (Storage<K,V> secondaryStorage) {
        _secondaryStorage = secondaryStorage;
    }

    V get (K key) {
        // If key in cache
        //    Fetch value from cache
        // Else
        //    value = _secondaryStorage.get(key);
        // ...
        // Return value;
    }
}
```

But what will be the “underlying storage” for the cache entries themselves?
Our “underlying storage” will consist of 2 components:

1. A queue of nodes to hold the relative order in which data are accessed.

- For $n$-element cache, max length of queue is $n$.
- LRU at the front, MRU at the back of the queue.
- Each node will contain both a key (e.g., URL) and corresponding value (e.g., webpage).

\[ W \text{ is LRU item.} \]

\[ Z \text{ is MRU item.} \]
LRU Cache implementation

- All the important cache data is stored in the queue.

- Whenever data X is requested, we move its Node to the back of the queue because it’s now the MRU item.

- Whenever data V (not in the cache) is requested, we fetch it from secondary storage, and then store it in the cache.

- We must evict the LRU item to make room.

\[\text{\texttt{W}}\] is LRU item.

\[\text{\texttt{Z}}\] is MRU item.
LRU Cache implementation

- All the important cache data is stored in the queue.
- Whenever data $X$ is requested, we move its Node to the back of the queue because it’s now the MRU item.
- Whenever data $V$ (not in the cache) is requested, we fetch it from secondary storage, and then store it in the cache.
- We must evict the LRU item to make room.

\[ \text{\_front} \quad \text{\_back} \quad n = 4 \]

$W$ is LRU item.

$Z$ is MRU item.
LRU Cache implementation

- All the important cache data is stored in the queue.
- Whenever data X is requested, we move its Node to the back of the queue because it’s now the MRU item.
- Whenever data V (not in the cache) is requested, we fetch it from secondary storage, and then store it in the cache.
- We must evict the LRU item to make room.

\[ \begin{align*}
\text{Node} & \quad \text{Node} & \quad \text{Node} & \quad \text{Node} \\
\_key: W & \quad \_key: X & \quad \_key: Y & \quad \_key: Z \\
\_value: \ldots & \quad \_value: \ldots & \quad \_value: \ldots & \quad \_value: \ldots \\
\_front & \quad & \quad & \quad \_back \\
\end{align*} \]

\( \_key: V \quad \_value: \ldots \)

\( n = 4 \)
LRU Cache implementation

- All the important cache data is stored in the queue.
- Whenever data X is requested, we move its Node to the back of the queue because it’s now the MRU item.
- Whenever data V (not in the cache) is requested, we fetch it from secondary storage, and then store it in the cache.
- We must evict the LRU item to make room.

W was LRU item and was evicted. V is now MRU item.
Reality check

• Suppose the cache stores $n = 3$ elements, and suppose the user requests the following webpages in the following order:

  cnn.com
google.com
gmail.com
yahoo.com
npr.org
wikipedia.org
cnn.com
gmail.com
npr.org
cnn.com
imdb.com

• Show the queue at each step.
LRU Cache implementation

- Unfortunately, a queue by itself will not suffice to implement the `LRUCache` interface.

- When we want to update a `Node`'s position in the queue to MRU, we have to find the node ($O(n)$).

- However, we can use an additional `HashTable<K, Node>` to “jump” to the desired `Node` in $O(1)$ time.
LRU Cache implementation

- Every key stored in the queue will also have an entry in a hash table.

The hash table affords $O(1)$ access to any cache item, given its key.

The queue affords $O(1)$ access to the LRU item (_front) in the cache.

<table>
<thead>
<tr>
<th>Key</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
LRU Cache implementation

Whenever the user calls `cache.get(X)`, item X becomes the MRU item.

Using the hash table, X’s node in the queue can be found in $O(1)$ time.

Its node is then moved to the back of the queue in $O(1)$ time.
LRU Cache implementation

Whenever the user calls `cache.get(X)`, item X becomes the MRU item.

Using the hash table, X’s Node in the queue can be found in $O(1)$ time.

Its Node is then moved to the back of the queue in $O(1)$ time.
LRU Cache implementation

If the user calls `cache.get(A)` and triggers an eviction, then the LRU node is removed from the queue and the hash table.

```
<table>
<thead>
<tr>
<th>Key</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

Node

- `_key: W`
- `_value: ...`

- `_front`

- `_key: Y`
- `_value: ...`

- `_key: Z`
- `_value: ...`

- `_key: X`
- `_value: ...`

- `_back`

\[ n = 4 \]
LKRU Cache implementation

• In summary:

• An LRU cache is an example of combining data structures to harness their individual strengths.

• To implement an LRU cache with $O(1)$ time for $\forall \text{get}(K \text{key})$, we need fast access both to the LRU item, and to an arbitrary item specified by $\text{key}$.

• A queue gives us $O(1)$ access to the LRU item (front of queue).

• A hash table gives us $O(1)$ access to an arbitrary Node in the queue.
Graphs.
The last fundamental data structure we will cover in this course is a *graph*.

Mathematically, a **graph** consists of a set $N$ of **nodes** (aka **vertices**) connected by a set $E$ of **edges**.
In computer science, graphs are useful for describing *relationships* (edges) among *things* (nodes).

E.g., each node might represent a *Facebook* user, and each edge might represent whether two Facebook users are *friends*. 
Graphs

- E.g., each node might represent a computer server, and each edge represents whether two nodes are linked by Ethernet.
Graphs

- Like trees, graphs consist of nodes and edges.
- Unlike trees, graphs can contain cycles.
- Graphs can be either undirected (as below)...

![Diagram of an undirected graph]

Nodes: 1, 2, 3, 4, 5, 6
Edges: (1, 2), (2, 3), (3, 4), (4, 5), (5, 6)
Graphs

• ...or **directed** (as below).

• *Directed graphs* are useful for describing *asymmetric* relationships, e.g., “I know who Rick Santorum is, but he doesn’t know who I am.”
In the graph below, \( N = \{ 1, 2, 3, 4, 5, 6 \} \).

An edge in a directed graph from node \( m \) to node \( n \) can be described as an ordered pair \( (m, n) \).

In the graph below, \( E = \{ (2, 3), (3, 1), (1, 2), (4, 1), (5, 6) \} \).
Graphs

- If a graph is undirected, then for every edge \((m, n) \in E\), we also have \((n, m) \in E\).

- For the graph below, \(E = \{ (2, 3), (3, 2), (1, 3), (3, 1), (1, 2), (2, 1), (1, 4), (4, 1), (5, 6), (6, 5) \} \).
Graphs

- Whenever \((m, n) \in E\), we say that node \(m\) is **adjacent** (or **connected**) to node \(n\).
Graphs

- In some graphs, edges have **weights** associated with them to represent distance, cost, etc.

- In this case, an edge can be represented as an ordered triplet \((m, n, w_{mn})\) where \(w_{mn}\) is the weight from \(m\) to \(n\).
Graphs

• An example of a weighted graph is an airline map that shows cities connected by flights, and the weight of each edge is the distance (km) between those cities.
Representing graphs

• To use graphs as a data structure, we must devise a way of representing a graph in memory.

• Let $N$ be the set of nodes and $E$ be the set of edges.

• The number of nodes is $|N|$, and the number of edges is $|E|$.

• To represent the set of nodes in memory, we can use an $|N|$-element array, where each node is assigned a unique index.

• This is both time- and space-efficient.
Representing graphs

• To represent the set of edges, we can use two alternative representations:
  • An **adjacency matrix** $A$ for the whole graph.
  • An **adjacency list** for every node $m \in N$. 
Adjacency matrices

• An **adjacency matrix** $A$ is an $|N| \times |N|$ matrix, where $|N|$ is the number of nodes in the graph.

• For an *unweighted* graph, the $(mn)$th entry of $A$ contains a 1 or a 0 depending on whether edge $(m, n) \in E$.

• For a *weighted* graph, the $(mn)$th entry of $A$ contains the *weight* of edge $(m, n) \in E$.

• If $(m, n) \not\in E$, then we can store either 0, infinity, or null (depending on what’s most useful).
Adjacency matrices

Example for *directed* graph:

\[
\begin{pmatrix}
1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0
\end{pmatrix}
\]
Adjacency matrices

Example for *undirected* graph:

In an *undirected* graph, the adjacency matrix $A$ equals its own transpose (i.e., $A = A^T$).

![Diagram of an undirected graph with vertices 1, 2, 3, 4, 5 and edges connecting 1-2, 2-3, 3-4, 4-5]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Adjacency matrices

- Adjacency matrices offer fast access to the presence/absence of any edge in the graph.
- However, for graphs in which edges are sparse, they are space-inefficient ($O(|N|^2)$).
- A space-saving (but slower) alternative is adjacency lists...
Adjacency lists

- With adjacency lists, every node maintains a list of other nodes to which it is connected.

Node 1: { 2 }
Node 2: { 3 }
Node 3: { 1 }
Node 4: { 1 }
Node 5: { 4, 1 }
Adjacency lists

- Adjacency lists require only $O(|E|)$ space to store all the edges.
- However, they require $O(|E|)$ time to find a particular edge.

Node 1: \{ 2 \}
Node 2: \{ 3 \}
Node 3: \{ 1 \}
Node 4: \{ 1 \}
Node 5: \{ 4, 1 \}
Graphs in computer science

• Graphs find many uses in computer science in almost every sub-discipline:
  • Computability/complexity theory.
  • Networking.
  • Machine learning.
  • Social networks.
  • Compilers
  • ...

Monday, August 29, 2011
Graphs in computer science

• Here, we will give a very superficial (but hopefully better than no) treatment of graphs.

• One of the fundamental algorithms associated with graphs is finding the shortest path between any two nodes $m, n$.

• This has applications in many real-world problems, such as...
Kevin Bacon and Erdős numbers

• ...

Monday, August 29, 2011