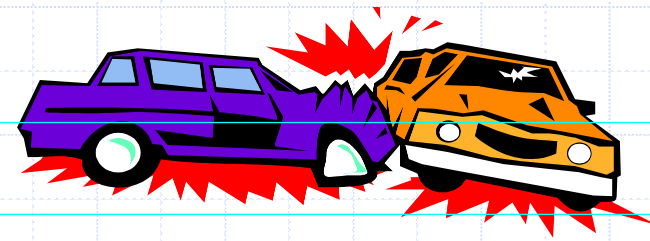
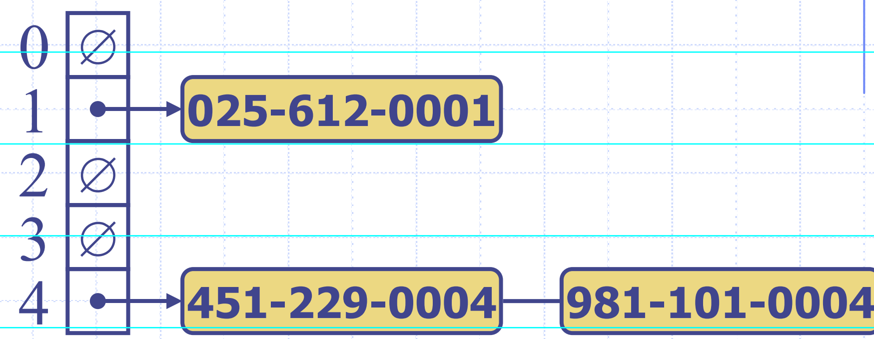
Collision Handling (§ 8.2.5)



- Collisions occur when different elements are mapped to the same cell



- **Separate Chaining**: let each cell in the table point to a linked list of entries that map there
- Separate chaining is simple, but requires (pointers) additional memory outside the table

Map Methods with Separate Chaining used for Collisions

- Delegate operations to a list-based map at each cell:

Algorithm get(k):
Output: The value associated with the key k in the map, or **null** if there is no entry with key equal to k in the map
return $A[h(k)]$.get(k) {delegate the get to the list-based map at $A[h(k)]$}

Invoking one data structure within another

Algorithm put(k,v):
Output: If there is an existing entry in our map with key equal to k, then we return its value (replacing it with v); otherwise, we return **null**
$t = A[h(k)]$.put(k,v) {delegate the put to the list-based map at $A[h(k)]$}
if t = **null then** {k is a new key}
  $n = n + 1$
return t

Algorithm remove(k):
Output: The (removed) value associated with key k in the map, or **null** if there is no entry with key equal to k in the map
$t = A[h(k)]$.remove(k) {delegate the remove to the list-based map at $A[h(k)]$}
if $t \neq$ **null then** {k was found}
  $n = n - 1$
return t

Linear Probing

(Somewhat overcomes memory wasted in pointers stored in linked lists)

- Open addressing: the colliding item is placed in a different cell of the table
- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

1 1 2 1 3 1 5 4

1 1 2 1 1 1 3 1

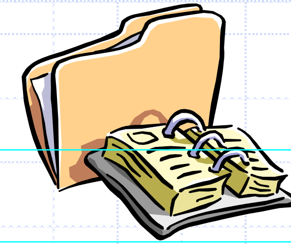
Exercise: Compare total cost of linear probing with linked list for this problem

1. Generally linear probing requires larger value of N in $x \bmod N$ then the linked list implementation

2. Linked list implementation incurs less search & insertion costs than linear probing for same value of N in $x \bmod N$.

3. But linked list stores more data in the form of "next" pointers.

Search with Linear Probing



- Consider a hash table A that uses linear probing
- get(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

Algorithm *get*(k)
    $i \leftarrow h(k)$
    $p \leftarrow 0$
    repeat
        $c \leftarrow A[i]$
        if $c = \varnothing$
            return *null*
        else if $c.key\,() = k$
            return $c.element()$
        else
            $i \leftarrow (i + 1) \bmod N$ since array is being treated as circular
            $p \leftarrow p + 1$
    until $p = N$
    return *null*

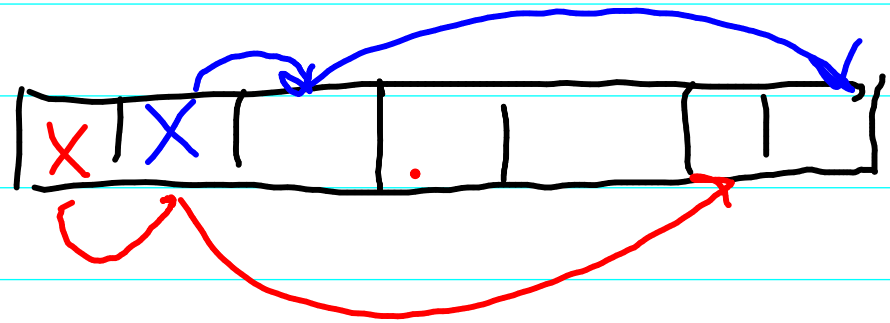
Updates with Linear Probing

empty string
eg. -1
-∞

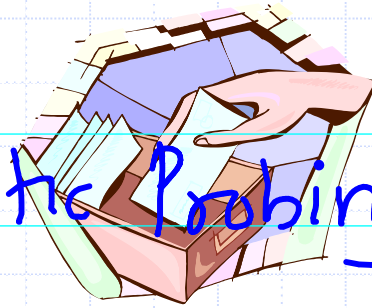
- To handle insertions and deletions, we introduce a special object, called ***AVAILABLE***, which replaces deleted elements
- remove(k)
  - We search for an entry with key k
  - If such an entry (k, o) is found, we replace it with the special item ***AVAILABLE*** and we return element o
  - Else, we return ***null***
- put(k, o)
  - We throw an exception if the table is full
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - A cell i is found that is either empty or stores ***AVAILABLE***, or
    - N cells have been unsuccessfully probed
  - We store entry (k, o) in cell i

will change for double hashing.

Quadratic Probing



Double Hashing : Generalises Linear & Quadratic Probing



Use # of collisions so far & scale it by another hash fn d_2

- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series

$$(i + jd(k)) \bmod N$$

for $j = 0, 1, \ldots, N-1$

- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells
- Common choice of compression function for the secondary hash function:

$$d_2(k) = q - k \bmod q$$

where
  - $q < N$
  - q is a prime
- The possible values for $d_2(k)$ are

$$1, 2, \ldots, q$$

Example of Double ~~Hashing~~ hash fn

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

ignored because # of collisions so far = j = 0

$h(k) + j\,d(k)$

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

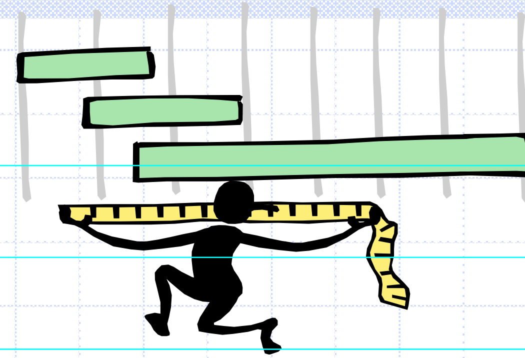
Note: j = # of collisions for the key under consideration

If "22" were not inserted, we would have had only one collision for each of 44 & 31

- Stop searching when $h(k) + j\,d(k) = h(k)$

- Under certain conditions (such as 13 & 7 are coprime), you can be assured that the search actually went through every cell.

Performance of Hashing



eg: For a hash fn that takes each key to same slot

- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the map collide

→ # of elements

The desired size of each linked list

- The load factor $\alpha = n/N$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is → H/W:
$$1 / (1 - \alpha)$$

Uniform hashing achieves load factor α

- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

Java Example

```
/** A hash table with linear probing and the MAD hash function */
public class HashTable implements Map {
  protected static class HashEntry implements Entry {
    Object key, value;
    HashEntry () { /* default constructor */ }
    HashEntry(Object k, Object v) { key = k; value = v; }
    public Object key() { return key; }
    public Object value() { return value; }
    protected Object setValue(Object v) {  // set a new value, returning old
      Object temp = value;
      value = v;
      return temp;  // return old value
    }
  }
  /** Nested class for a default equality tester */
  protected static class DefaultEqualityTester implements EqualityTester {
    DefaultEqualityTester() { /* default constructor */ }
    /** Returns whether the two objects are equal.  */
    public boolean isEqualTo(Object a, Object b) { return a.equals(b); }
  }
  protected static Entry AVAILABLE = new HashEntry(null, null); // empty
        marker
  protected int n = 0;                    // number of entries in the dictionary
  protected int N;                        // capacity of the bucket array
  protected Entry[] A;                                    // bucket array
  protected EqualityTester T;             // the equality tester
  protected int scale, shift;   // the shift and scaling factors
  /** Creates a hash table with initial capacity 1023. */
  public HashTable() {
    N = 1023; // default capacity
    A = new Entry[N];
    T = new DefaultEqualityTester(); // use the default equality tester
    java.util.Random rand = new java.util.Random();
    scale = rand.nextInt(N-1) + 1;
    shift = rand.nextInt(N);
  }
```

```
/** Creates a hash table with the given capacity and equality tester. */
  public HashTable(int bN, EqualityTester tester) {
    N = bN;
    A = new Entry[N];
    T = tester;
    java.util.Random rand = new java.util.Random();
    scale = rand.nextInt(N-1) + 1;
    shift = rand.nextInt(N);
  }
```

Java Example (cont.)

```java
/** Determines whether a key is valid. */
protected void checkKey(Object k) {
  if (k == null) throw new InvalidKeyException("Invalid key: null.");
}
/** Hash function applying MAD method to default hash code. */
public int hashValue(Object key) {
  return Math.abs(key.hashCode()*scale + shift) % N;
}
/** Returns the number of entries in the hash table. */
public int size() { return n; }
/** Returns whether or not the table is empty. */
public boolean isEmpty() { return (n == 0); }
/** Helper search method - returns index of found key or -index-1,
 * where index is the index of an empty or available slot. */
protected int findEntry(Object key) throws InvalidKeyException {
  int avail = 0;
  checkKey(key);
  int i = hashValue(key);
  int j = i;
  do {
    if (A[i] == null)   return -i - 1;  // entry is not found
    if (A[i] == AVAILABLE) {              // bucket is deactivated
        avail = i;                         // remember that this slot is available
        i = (i + 1) % N;                   // keep looking
    }
    else if (T.isEqualTo(key,A[i].key()))  // we have found our entry
        return i;
    else // this slot is occupied--we must keep looking
        i = (i + 1) % N;
  } while (i != j);
  return -avail - 1;  // entry is not found
}
/** Returns the value associated with a key. */
public Object get (Object key) throws InvalidKeyException {
  int i = findEntry(key);  // helper method for finding a key
  if (i < 0) return null;  // there is no value for this key
  return A[i].value();     // return the found value in this case
}
/** Put a key-value pair in the map, replacing previous one if it exists. */
public Object put (Object key, Object value) throws InvalidKeyException {
  if (n >= N/2) rehash(); // rehash to keep the load factor <= 0.5
  int i = findEntry(key); //find the appropriate spot for this entry
  if (i < 0) {       // this key does not already have a value
    A[-i-1] = new HashEntry(key, value); // convert to the proper index
    n++;
    return null;     // there was no previous value
  }
  else                                   // this key has a previous value
    return ((HashEntry) A[i]).setValue(value); // set new value & return old
}
/** Doubles the size of the hash table and rehashes all the entries. */
protected void rehash() {
  N = 2*N;
  Entry[] B = A;
  A = new Entry[N]; // allocate a new version of A twice as big as before
  java.util.Random rand = new java.util.Random();
  scale = rand.nextInt(N-1) + 1;                  // new hash scaling factor
  shift = rand.nextInt(N);                        // new hash shifting factor
  for (int i=0; i&ltB.length; i++)
    if ((B[i] != null) && (B[i] != AVAILABLE)) { // if we have a valid entry
      int j = findEntry(B[i].key());  // find the appropriate spot
      A[-j-1] = B[i];                 // copy into the new array
    }
}
/** Removes the key-value pair with a specified key. */
public Object remove (Object key) throws InvalidKeyException {
  int i = findEntry(key);         // find this key first
  if (i < 0) return null;         // nothing to remove
  Object toReturn = A[i].value();
  A[i] = AVAILABLE;                               // mark this slot as
      deactivated
  n--;
  return toReturn;
}
/** Returns an iterator of keys. */
public java.util.Iterator keys() {
  List keys = new NodeList();
  for (int i=0; i&ltN; i++)
    if ((A[i] != null) && (A[i] != AVAILABLE))
      keys.insertLast(A[i].key());
  return keys.elements();
}
} // ... values() is similar to keys() and is omitted here ...
```

HOMEWORK PROBLEM

(a)What would be a good hash code for a vehicle identification number, that is a string of numbers and letters of the form "9X9XX99X9XX999999," where a "9" represents a digit and an "X" represents a letter?

ANS: Use polynomial hash codes

(b) Now suppose you are given a collection C of n vehicle-speed pairs (veh-id,s), with veh-id denoting the vehicle identification number and s denoting the speed with which the vehicle was detected moving at a particular point of time. Describe an efficient algorithm for computing a histogram of car speeds by making use of some HashMap. What would be the time complexity of your algorithm?

ANS: Given a map { id1:s1, id2:s2....,idn:sn} , decide on some ranges of speeds for the histogram. Let N be number of ranges and s_min and s_max the maximum and mimumum speeds from the map. So the ith range will be

[s_min + (i-1)*N/(s_max-s_min), s_min + i*N/(s_max-s_min)]

Based on this, produce a map { 1:count_1, 2:count_2, ...i:count_i, N:count_N} which contains the count of each ith range. This can be computed by iterating over all the ids just once.

Thus complexity will be O(n).

Extensions to Hash Table:

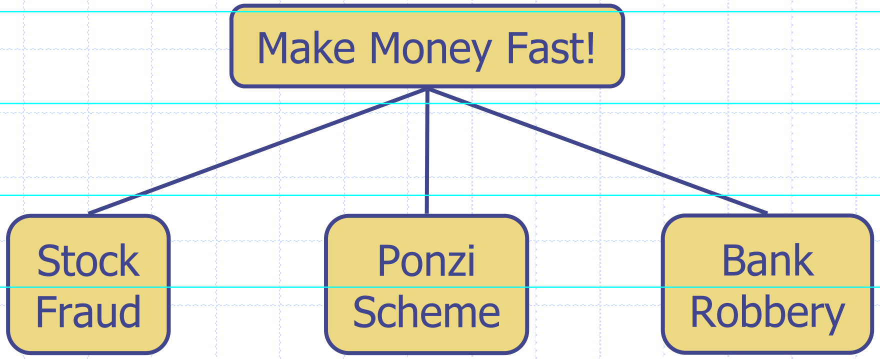
Organising keys s.t

(a) Enumerating keys in "increasing" or "decreasing" order

(b) Want to find smallest or largest key

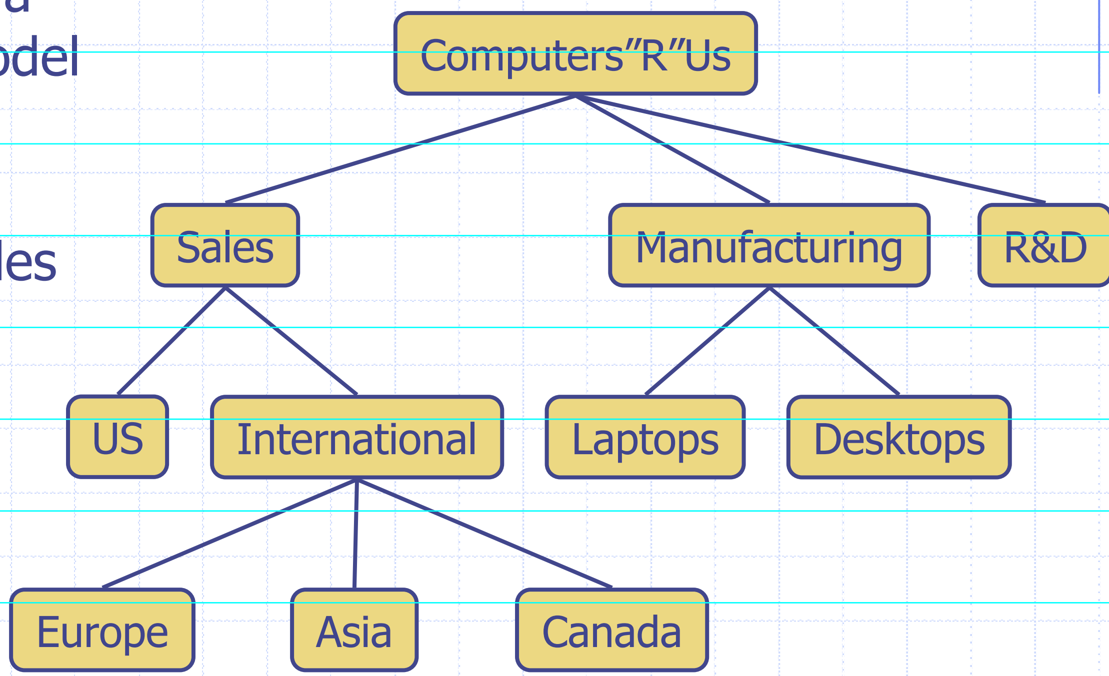
(c) Want to find top "k" keys in terms of their values being the "k" largest

Trees



What is a Tree

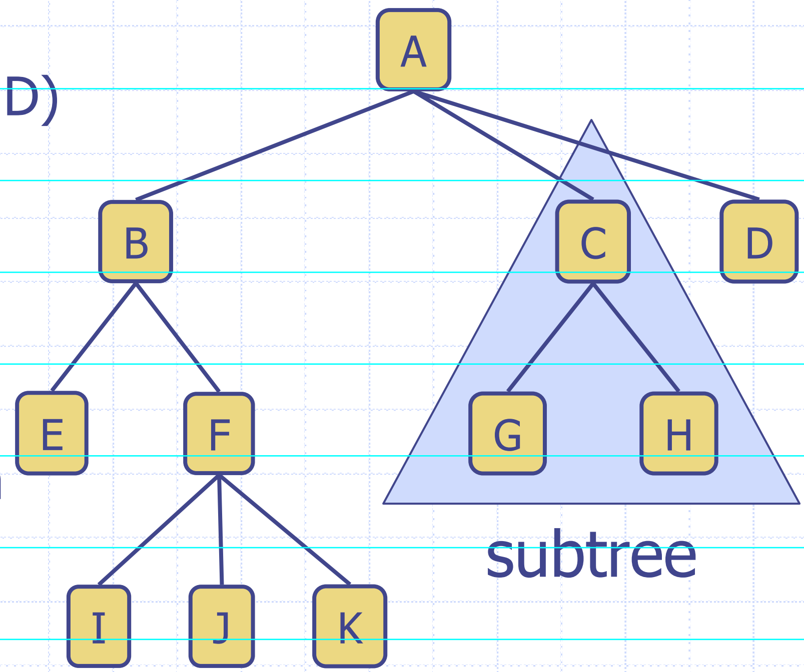
- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



- Arithmatic expression evaluation

Tree Terminology

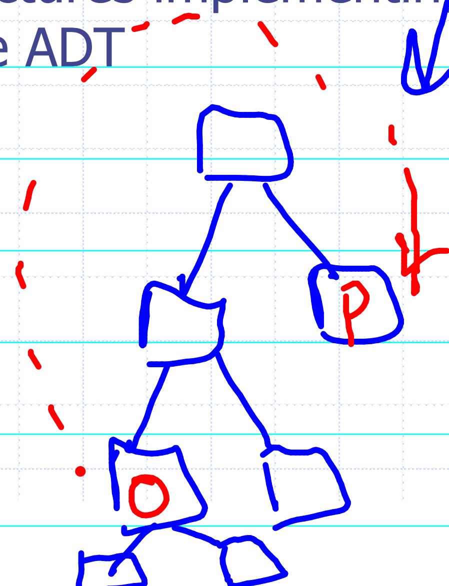
- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of ancestors (upto root)
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild, grand-grandchild, etc.
- Subtree: tree consisting of a node and its descendants



Tree ADT (§ 6.1.2)

- We use positions to abstract nodes
- Generic methods:
  - integer size() = # of nodes
  - boolean isEmpty() = no nodes
  - Iterator elements()
  - Iterator positions()

  somewhat synonymous
- Accessor methods:
  - position root()
  - position parent(p)
  - positionIterator children(p)
- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- Update method:
  - object replace (p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT



Tree for $((a+b)*(c+d))/(p-q)$

sh

Construct trees for each & contrast operations you would do on each

Tree for: