ID1020: Hash Tables

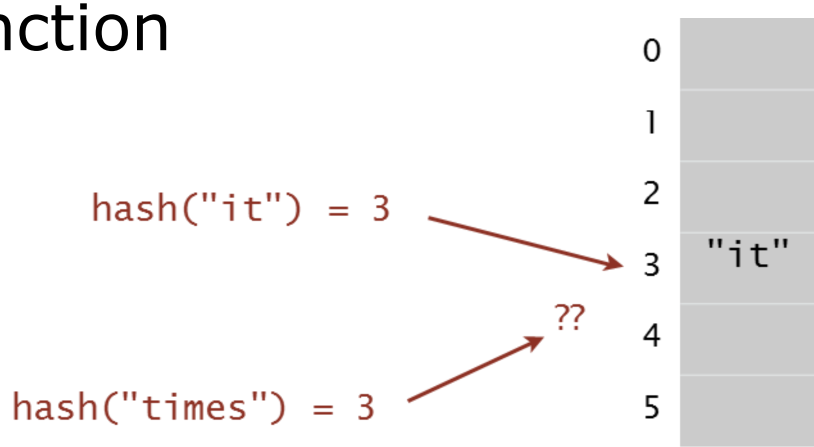
Dr. Per Brand
pbrand@kth.se

kap 3.4

Slides adapted from Algoritms 4th Edition, Sedgewick.

Introduction to Hashing

- Save items in an indexed table
  - Hash function computes array index from key
- Issues
  - Choosing a good hash function
  - Equality test
  - Collision Resolution



Hash function

- The goal
  - Efficiently computable and deterministic!
  - Each table index equally likely for each key
- We require
  - If `x.equals(y)`, then `(x.hashCode() == y.hashCode())`
- We desire
  - If `!x.equals(y)`, then `(x.hashCode()!= y.hashCode())`.
- Problem
  - Keys are almost never uniformly distributed
    - We have clusters
  - How to have a function that maps from a clustered domain to uniformly distributed range?

Examples

- Using very simple hash functions
- Phone number
  - Bad hash function: First three digits
  - Better: Last three digits
  - Why?

- Personnummer
  - Which tree digits are best?
  - Think about age distribution
  - Think about birth distribution over the year

Using Java hashcode

- All Java classes inherit a method *hashCode()*
  - Return 32-bit integer
  - To use in array of size M take modulo M

- Customized implementations
  - Integer, Double, String, File, URL, Date, etc.
-
- Default implementation
  - Memory address
  - Generally poor choice

- User-defined types
  - User needs to choose appropriate hash function

Java library functions

```
public final class Integer
{
   private final int value;
   ...

   public int hashCode()
   {  return value;   }
}
```

```
public final class Boolean
{
   private final boolean value;
   ...

   public int hashCode()
   {
      if (value) return 1231;
      else       return 1237;
   }
}
```

```
public final class Double
{
   private final double value;
   ...

   public int hashCode()
   {
      long bits = doubleToLongBits(value);
      return (int) (bits ^ (bits >>> 32));
   }
}
```

convert to IEEE 64-bit representation;
xor most significant 32-bits
with least significant 32-bits

Java library - strings

Java library implementation

```
public final class String
{
   private final char[] s;
   ...

   public int hashCode()
   {
      int hash = 0;
      for (int i = 0; i < length(); i++)
         hash = s[i] + (31 * hash);
      return hash;
   }
}
```

i^{th} character of s

char	Unicode
…	…
'a'	97
'b'	98
'c'	99
…	…

- Horner's method to hash string of length L: L multiplies/adds.
- Equivalent to $h = s[0] \cdot 31^{L-1} + \ldots + s[L-3] \cdot 31^2 + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0$.

- Note: the use of 31
- Note: fairly expensive

String optimization

- Cache the value

```
public final class String
{
   private int hash = 0;
   private final char[] s;
   ...

   public int hashCode()
   {
      int h = hash;
      if (h != 0) return h;
      for (int i = 0; i < length(); i++)
         h = s[i] + (31 * h);
      hash = h;
      return h;
   }
}
```

cache of hash code

return cached value

store cache of hash code

- Design pattern for user-defined types

Standard recipe

- Combine each field using *31*New+Old*
  - If primitive use *Java library*
  - If reference type recurse
  - If array apply to each entry

- In practice
  - Recipe works reasonably well
  - However no guarantee

- Important
  - Use the whole key !

Modular hashing

- Hash code: a 32-bit integer between -2^{31} and $2^{31}-1$
- Hash function: An index between 0 and M-1
  - Where M is the size of the array
  - M typically a power of 2 OR a prime Why?
  - Basic idea: use modulo

```
private int hash(Key key)
{   return key.hashCode() % M;   }
```

bug

```
private int hash(Key key)
{   return Math.abs(key.hashCode()) % M;   }
```

1-in-a-billion bug

hashCode() of "polygenelubricants" is -2^{31}

```
private int hash(Key key)
{   return (key.hashCode() & 0x7fffffff) % M;   }
```

correct

Uniform hashing assumption

- Assumption: Hash functions uniformly and independently distribute the keys
- Ideal that is not met but generally close enough

- If true then M distinct entries in an array of size M most loaded slot has $\Theta(\log M / \log \log M)$ entries
  - E.g., hash value frequencies for words in Tale of Two Cities (M = 97)

Hash Table API

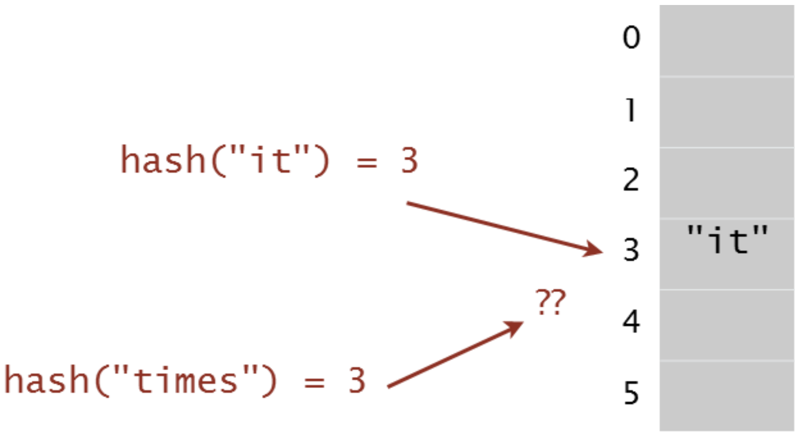
- Compared to BST hash tables have a lean API
- Basically just get and put

- In particular, it lacks
  - Ordered iterations
  - Rank
  - Select
  - interval size

Collisions

- Collision:
  - 2 distinct keys hashing to same index
  - Cannot in practice be avoided
    - Quadratic memory !!
    - Birthday problem

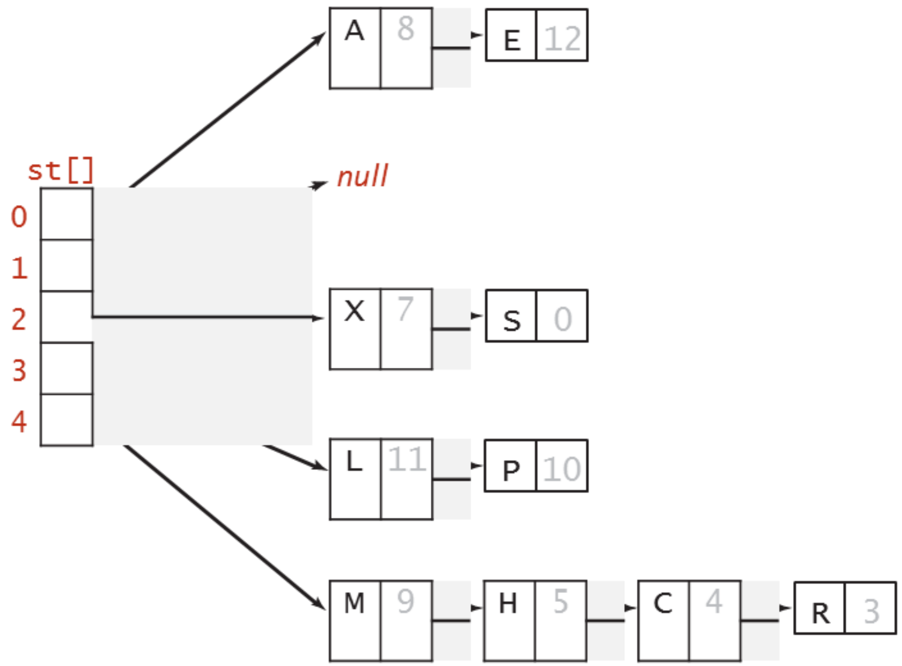
- Challenge
  - Deal with efficiently



Method 1

- Separate chaining hash tables
  - Use array M<N of linked lists
  - Hash: map to index between 0 and M-1
  - Insert: put in front
  - Search: search a (hopefully) short chain

key	hash	value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9
P	3	10
L	3	11
E	0	12



Separate chaining in Java

```
public class SeparateChainingHashST<Key, Value>
{
   private int M = 97;                // number of chains
   private Node[] st = new Node[M];   // array of chains

   private static class Node
   {
      private Object key;
      private Object val;
      private Node next;
      ...
   }

   private int hash(Key key)
   {  return (key.hashCode() & 0x7fffffff) % M;   }

   public Value get(Key key) {
      int i = hash(key);
      for (Node x = st[i]; x != null; x = x.next)
         if (key.equals(x.key)) return (Value) x.val;
      return null;
   }

}
```

array doubling and halving code omitted

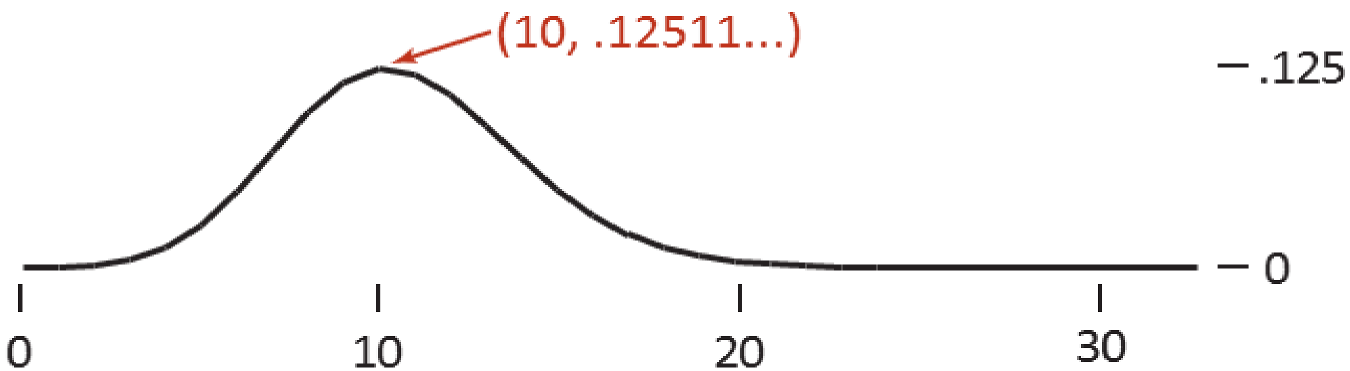
no generic array creation
(declare key and value of type Object)

Separate chaining in Java (2)

```
public class SeparateChainingHashST<Key, Value>
{
   private int M = 97;                      // number of chains
   private Node[] st = new Node[M];         // array of chains

   private static class Node
   {
      private Object key;
      private Object val;
      private Node next;
      ...
   }

   private int hash(Key key)
   {  return (key.hashCode() & 0x7fffffff) % M;    }

   public void put(Key key, Value val) {
      int i = hash(key);
      for (Node x = st[i]; x != null; x = x.next)
         if (key.equals(x.key)) { x.val = val; return; }
      st[i] = new Node(key, val, st[i]);
   }

}
```

Analysis of separate chaining

- Proposition
  - With uniform hashing
  - Number of keys in a list is < cN/M where c is a small constant : probablity very high.
  - Binomial distribution



Binomial distribution ($N = 10^4$, $M = 10^3$, a = 10)

Consequence

- Cost of get/put is proportional to N/M

- Classic time/space tradeoff
- When M is very large cost approaches 1
  - No collisions
- When M is very small cost approaches N

- In practice make M as large as you can afford.

Method 2 for dealing with collisions

- Linear probing
  - Method 2 for dealing with collisions
  - Hash as before to find preferred slot (index)
  - Put: if slot not free use next free slot(wraparound)
  - Get: if slot occupied but no match try next until either
    - Match
    - Free slot (search miss)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

M = 16

- What happens when N=M-1, N=M, N=M+1 ?

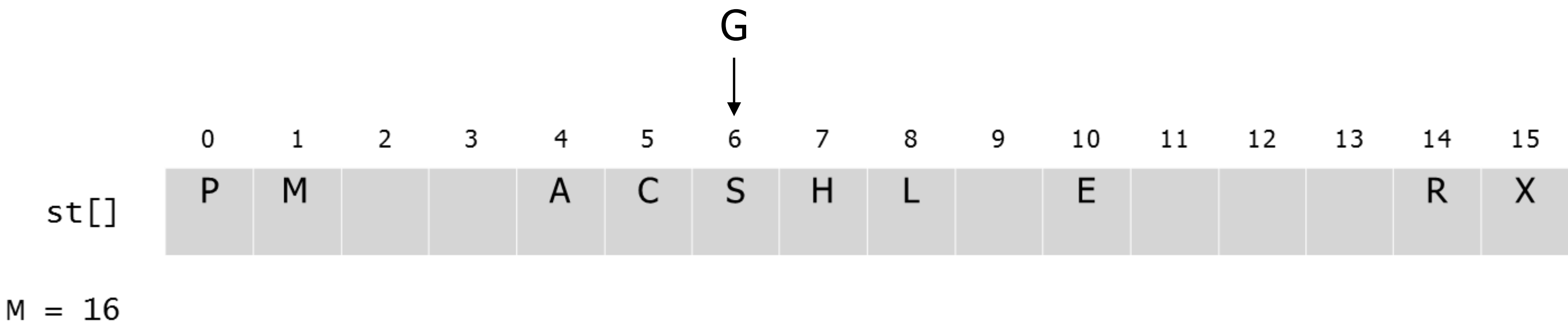
Linear Probing Implementation

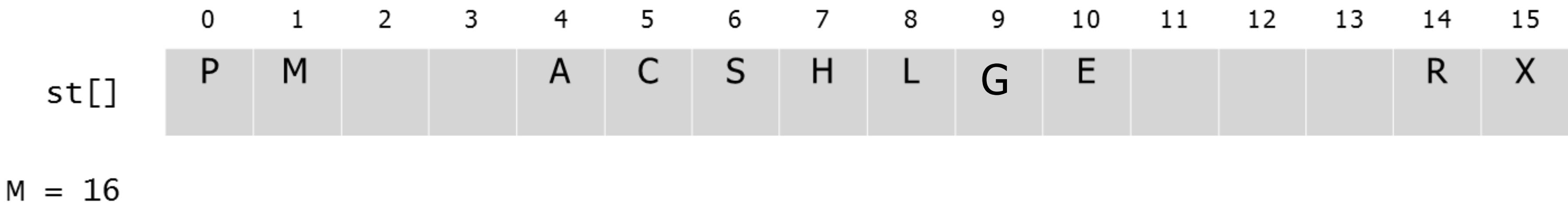
```
public class LinearProbingHashST<Key, Value>
{
   private int M = 30001;
   private Value[] vals = (Value[]) new Object[M];
   private Key[]   keys = (Key[])   new Object[M];

   private int hash(Key key) {  /* as before */  }

   public void put(Key key, Value val)
   {
      int i;
      for (i = hash(key); keys[i] != null; i = (i+1) % M)
         if (keys[i].equals(key))
             break;
      keys[i] = key;
      vals[i] = val;
   }

   public Value get(Key key)
   {
      for (int i = hash(key); keys[i] != null; i = (i+1) % M)
         if (key.equals(keys[i]))
             return vals[i];
      return null;
   }
}
```

array doubling and halving code omitted

Potential problem: clustering

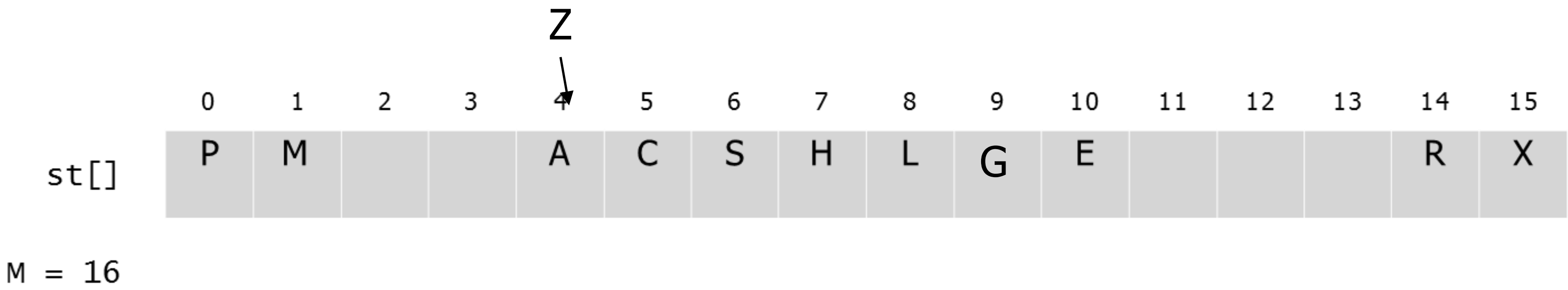
- Cluster: contiguous block of items
- Observation:
  - Adding new time likely to hit in the middle of big clusters





Potential problem: clustering (2)

- Observation:
  - Clusters tend to grow



Load factor

- N items in a hash table of size M
  - load = N/M
- As expected performance depends on load
  - When load is ½
    - Average number of probes for hit is 3/2
    - For search miss is 5/2
  - Full analysis in the book

- Consider case of load ½
  - What is best case ?
  - What is worst case ?

Load 1/2

- Best case distribution
  - Odd indices used, even indices empty
- Worst case distribution
  - One big cluster occupying half of the array
  - Note: in both cases average length of cluster ½
- Best case
  - Search miss 1+1/2
- Worst case
  - Search miss ~N/4
- Typically array resizing is used to keep hash table with 1/8 < load < ½

Hash Table warning (1)

- Temptation to simplify hash function
  - E.g., for long strings only use each jth character
- In Java 1.1
  - String hashcode skipped characters for long strings

- Danger
  - Can make for many collisions

```
http://www.cs.princeton.edu/introcs/13loop/Hello.java
http://www.cs.princeton.edu/introcs/13loop/Hello.class
http://www.cs.princeton.edu/introcs/13loop/Hello.html
http://www.cs.princeton.edu/introcs/12type/index.html
```

Hash table warning (2)

- Denial-of-service attacks
  - Malicious adversary knows your hash function
    - E.g., by reading Java documentation
  - Malicious adversary can create items that you hash
  - Attack
    - Creates many items that have the same hashcode
    - Result: collision
- Real world examples
  - Perl 5.8.0 - carefully chosen strings
  - Linux 2.4.20 kernel – save files with carefully chosen names

Example

- Attacker's point of view
  - Assuming java string library hashcode used
  - Goal find a family of strings with same hash code
  - Not so difficult

key	hashCode()
"Aa"	2112
"BB"	2112

key	hashCode()
"AaAaAaAa"	-540425984
"AaAaAaBB"	-540425984
"AaAaBBAa"	-540425984
"AaAaBBBB"	-540425984
"AaBBAaAa"	-540425984
"AaBBAaBB"	-540425984
"AaBBBBAa"	-540425984
"AaBBBBBB"	-540425984

key	hashCode()
"BBAaAaAa"	-540425984
"BBAaAaBB"	-540425984
"BBAaBBAa"	-540425984
"BBAaBBBB"	-540425984
"BBBBAaAa"	-540425984
"BBBBAaBB"	-540425984
"BBBBBBAa"	-540425984
"BBBBBBBB"	-540425984

2^N strings of length 2N that hash to same value!

Symbol table summary

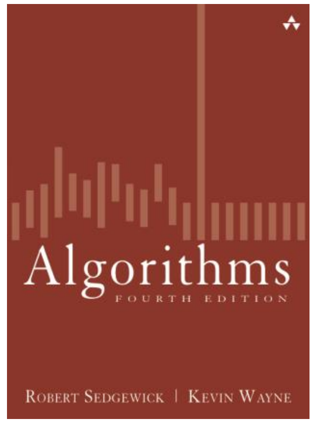
implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	`compareTo()`
separate chaining	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	`equals()` `hashCode()`
linear probing	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	`equals()` `hashCode()`

* under uniform hashing assumption

ID1020: Applications

Dr. Per Brand
perbrand@sics.se

kap 3.5 +



Slides adapted from Algoritms 4th Edition, Sedgewick.

Sets

- 1st example
  - Filtering using sets
  - Need less than before
    - No values only keys

- Filtering
  - Find all words {in / not in} a given set

Set API

`public class SET<Key extends Comparable<Key>>`		
	`SET()`	*create an empty set*
`void`	`add(Key key)`	*add the key to the set*
`boolean`	`contains(Key key)`	*is the key in the set?*
`void`	`remove(Key key)`	*remove the key from the set*
`int`	`size()`	*return the number of keys in the set*
`Iterator<Key>`	`iterator()`	*iterator through keys in the set*

How to implement ?

Filtering examples

application	purpose	key	in list
spell checker	identify misspelled words	word	dictionary words
browser	mark visited pages	URL	visited pages
parental controls	block sites	URL	bad sites
chess	detect draw	board	positions
spam filter	eliminate spam	IP address	spam addresses
credit cards	check for stolen cards	number	stolen cards

Example 2

- More or less straightforward usage of symbol table
- Dictionary lookup

Dictionary lookup

- Comma separated value file (CSV)

```
% more classlist.csv
13,Berl,Ethan Michael,P01,eberl
12,Cao,Phillips Minghua,P01,pcao
11,Chehoud,Christel,P01,cchehoud
10,Douglas,Malia Morioka,P01,malia
12,Haddock,Sara Lynn,P01,shaddock
12,Hantman,Nicole Samantha,P01,nhantman
11,Hesterberg,Adam Classen,P01,ahesterb
13,Hwang,Roland Lee,P01,rhwang
13,Hyde,Gregory Thomas,P01,ghyde
13,Kim,Hyunmoon,P01,hktwo
12,Korac,Damjan,P01,dkorac
11,MacDonald,Graham David,P01,gmacdona
10,Michal,Brian Thomas,P01,bmichal
12,Nam,Seung Hyeon,P01,seungnam
11,Nastasescu,Maria Monica,P01,mnastase
11,Pan,Di,P01,dpan
12,Partridge,Brenton Alan,P01,bpartrid
13,Rilee,Alexander,P01,arilee
13,Roopakalu,Ajay,P01,aroopaka
11,Sheng,Ben C,P01,bsheng
12,Webb,Natalie Sue,P01,nwebb
⋮
```

login is key

first name is value

```
% java LookupCSV classlist.csv 4 1
eberl
Ethan
nwebb
Natalie
```

login is key

section is value

```
% java LookupCSV classlist.csv 4 3
dpan
P01
```

Implementation

```
public class LookupCSV
{
   public static void main(String[] args)
   {
      In in = new In(args[0]);
      int keyField = Integer.parseInt(args[1]);
      int valField = Integer.parseInt(args[2]);

      ST<String, String> st = new ST<String, String>();
      while (!in.isEmpty())
      {
         String line = in.readLine();
         String[] tokens = line.split(",");
         String key = tokens[keyField];
         String val = tokens[valField];
         st.put(key, val);
      }

      while (!StdIn.isEmpty())
      {
         String s = StdIn.readString();
         if (!st.contains(s)) StdOut.println("Not found");
         else                 StdOut.println(st.get(s));
      }
   }
}
```

process input file

build symbol table

process lookups
with standard I/O

Example 3: File indexing

- Given a list of files create an index
  - To efficiently find all files containing giving query string

- Solution symbol table
  - Key: string
  - Value: set of files containing string

File indexing in action

```
% ls *.txt
aesop.txt magna.txt moby.txt
sawyer.txt tale.txt

% java FileIndex *.txt

freedom
magna.txt moby.txt tale.txt

whale
moby.txt

lamb
sawyer.txt aesop.txt
```

```
% ls *.java
BlackList.java Concordance.java
DeDup.java FileIndex.java ST.java
SET.java WhiteList.java

% java FileIndex *.java

import
FileIndex.java SET.java ST.java

Comparator
null
```

Implementation

```
import java.io.File;
public class FileIndex
{
   public static void main(String[] args)
   {
      ST<String, SET<File>> st = new ST<String, SET<File>>();

      for (String filename : args) {
         File file = new File(filename);
         In in = new In(file);
         while (!in.isEmpty())
         {
            String key = in.readString();
            if (!st.contains(key))
               st.put(key, new SET<File>());
            SET<File> set = st.get(key);
            set.add(file);
         }
      }

      while (!StdIn.isEmpty())
      {
         String query = StdIn.readString();
         StdOut.println(st.get(query));
      }
   }
}
```

symbol table

list of file names from command line

for each word in file, add file to corresponding set

process queries

Example 4: Sparse vectors/matrices

- Standard implementation

$$\begin{array}{cccc} \texttt{a[][]} & \texttt{x[]} & & \texttt{b[]} \\ \begin{bmatrix} 0 & .90 & 0 & 0 & 0 \\ 0 & 0 & .36 & .36 & .18 \\ 0 & 0 & 0 & .90 & 0 \\ .90 & 0 & 0 & 0 & 0 \\ .47 & 0 & .47 & 0 & 0 \end{bmatrix} & \begin{bmatrix} .05 \\ .04 \\ .36 \\ .37 \\ .19 \end{bmatrix} & = & \begin{bmatrix} .036 \\ .297 \\ .333 \\ .045 \\ .1927 \end{bmatrix} \end{array}$$

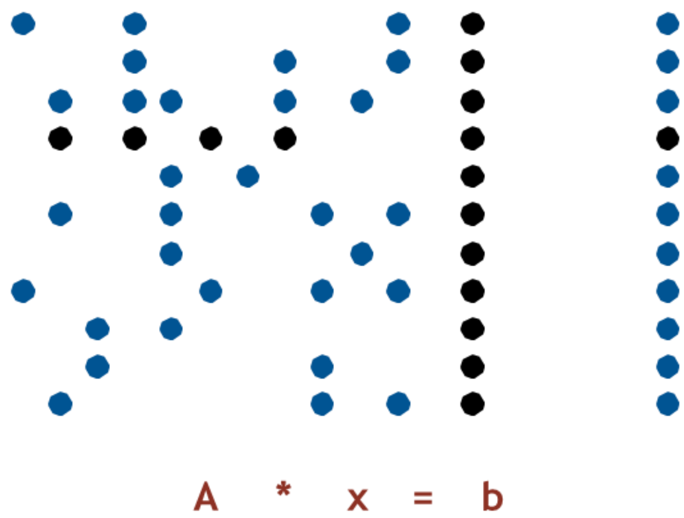
```
...
double[][] a = new double[N][N];
double[] x = new double[N];
double[] b = new double[N];
...
// initialize a[][] and x[]
...
for (int i = 0; i < N; i++)
{
   sum = 0.0;
   for (int j = 0; j < N; j++)
      sum += a[i][j]*x[j];
   b[i] = sum;
}
```

nested loops
(N^2 running time)

Problem

- Time and memory

Assumptions. Matrix dimension is 10,000; average nonzeros per row ~ 10.



Vector representations

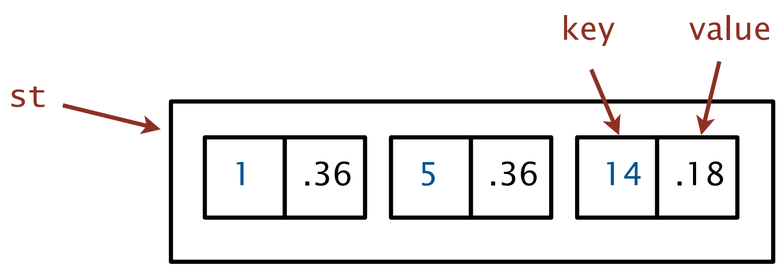
1d array (standard) representation.

- Constant time access to elements.
- Space proportional to N.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	.36	0	0	0	.36	0	0	0	0	0	0	0	0	.18	0	0	0	0	0

Symbol table representation.

- Key =index ,value =entry
- Efficient iterator.
- Space proportional to number of nonzeros.



Sparse vector implementation

```
public class SparseVector
{
   private HashST<Integer, Double> v;

   public SparseVector()
   {  v = new HashST<Integer, Double>();   }

   public void put(int i, double x)
   {  v.put(i, x);   }

   public double get(int i)
   {
      if (!v.contains(i)) return 0.0;
      else return v.get(i);
   }

   public Iterable<Integer> indices()
   {  return v.keys();   }

   public double dot(double[] that)
   {
       double sum = 0.0;
       for (int i : indices())
           sum += that[i]*this.get(i);
       return sum;
   }
}
```

HashST because order not important

empty ST represents all 0s vector

a[i] = value

return a[i]

dot product is constant time for sparse vectors

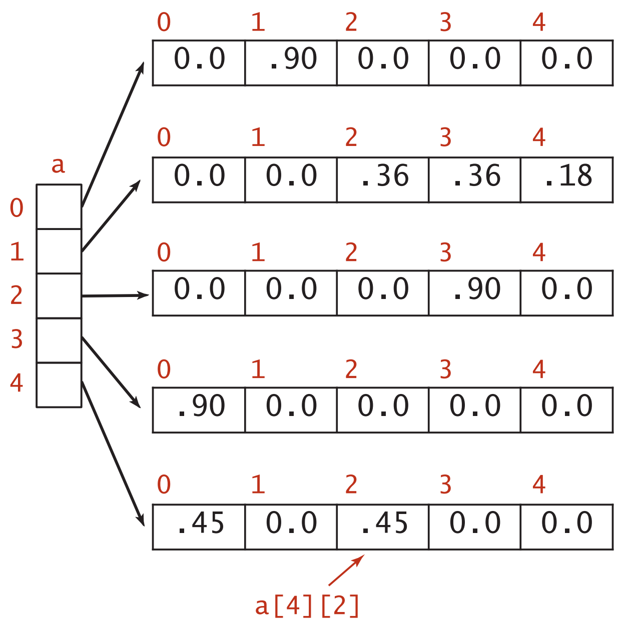
Matrix representations

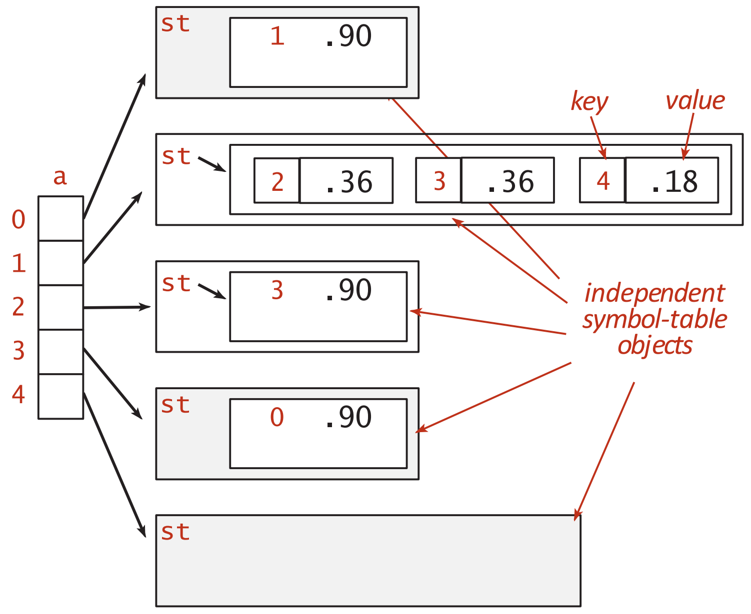
2D array (standard) matrix representation: Each row of matrix is an array.

- Constant time access to elements.
- Space proportional to N^2.

Sparse matrix representation: Each row of matrix is a sparse vector.

- Efficient access to elements.
- Space proportional to number of nonzeros (plus N).





What to choose?

- Choose red-black BSTs or derivative if
  - Worst-case complexity is important OR
  - Need BST wide range of operations
    - Rank, select, sort, iterate
- Otherwise use hash tables

- Worst case complexity especially important
  - Mission-critical applications
    - E.g. Nuclear power plants, airplanes
  - Partially open and subject to DoS attacks

Problem Example

- From last exam
- Using what we have covered

Task

You have two large text files, one with Shakespeare's collected works and en one with Milton's collected works. Assume that both contain N words. You should write a program to determine the following:

a) Number of words that Shakespeare uses at least once but that Milton doesn't use at all.
b) The M most frequently used words that Shakespeare uses but Milton doesn't.

Determine time complexity (you may refine your analysis with other faktors)

N: number of words in the text documents

U: number of distinct words in Shakespeare

Assume that the text does not contain proper Names or other non-words (i.e. the text has been pre-filtered)

You can only use the datastructures & algorithms that were used in the course.

For maximum points your solution must be the most efficient within a constant factor

Symbol table summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	`compareTo()`
separate chaining	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	`equals()` `hashCode()`
linear probing	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	`equals()` `hashCode()`

* under uniform hashing assumption

Pseudo-code

```
BST shakeTree = new BST<String,Integer>();   %% (t.ex. röd-svart BST

for all words w in Shakespeare {
        Integer v=shakeTree.get(w);
        if v==NULL then shakeTree.put(w,1)
        else shakeTree.put(w,v+1);}   %% antar att put skriver över


for all words w in Milton {
        Integer v=shakeTree.get(w);
        if v!=NULL then shakeTree.delete (w);
```

Svar A: = shakeTree.size()

Pseudocode (2)

```
Queue q=shakeTree.keys();

MaxPQ<KeyValuePair>pq = new
MaxPQ<KeyValuePair>(shakeTree.size())

w = q.dequeue();
while (w != NULL) {
        pq.insert(new KeyValuePair(shakeTree.get(w),w);
        w=q.dequeue();}

SVAR_B   for(i=0,i++,i<M) printPair(pq.max());
```