http://algs4.cs.princeton.edu

3.1 SYMBOL TABLES

- *API*
- *elementary implementations*
- *ordered operations*

Information storage and retrieval

- Computers essentially do four things:
  - Store and retrieve information;
  - Perform operations on information;
  - Send information to devices or networks;
  - Keep time.
- But how *do* they store/retrieve information?
  - At the machine level, they use an *address*;
  - But what about at the application level?
    - We can use an array—which directly translates into an address (by adding the index to the start address); *But*—arrays have to be fixed in size!
    - In practice, there's very little application data that is ideally suited to this sort of addressing.
    - Instead, we typically store/retrieve information by a *key.*
- A data structure based on *key access* is called a ***symbol table.***

http://algs4.cs.princeton.edu

3.1 Symbol Tables

- *API*
- *elementary implementations*
- *ordered operations*

Symbol tables

Key-value pair abstraction.

- Insert a value with an associated, unique key. ← At least, *relatively* unique
- Retrieve a value, by providing the associated key.

Ex. DNS lookup.

- Insert domain name with specified IP address.
- Given domain name, find corresponding IP address.

Domain name (key)	IP address
www.northeastern.edu	104.96.210.247
www.princeton.edu	128.112.128.15
www.yale.edu	130.132.143.21
www.harvard.edu	128.103.060.55
www.redsox.com	209.102.213.46

↑ key ↑ value

Symbol table applications

application	purpose of search	key	value
dictionary	find definition	word	definition
book index	find relevant pages	term	list of page numbers
file share	find song to download	name of song	computer ID
financial account	process transactions	account number	transaction details
web search	find relevant web pages	keyword	list of page names
compiler	find properties of variables	variable name	type and value
routing table	route Internet packets	destination	best route
DNS	find IP address	domain name	IP address
reverse DNS	find domain name	IP address	domain name
genomics	find markers	DNA string	known positions
file system	find file on disk	filename	location on disk

Symbol tables: context

Also known as: maps, dictionaries, associative arrays.

Generalizes arrays. Keys need not be between 0 and $N-1$.

Language support.

- External libraries: C, VisualBasic, Standard ML, bash, ...
- Built-in libraries: Java, C#, C++, Scala, ...
- Built-in to language: Awk, Perl, PHP, Tcl, JavaScript, Python, Ruby, Lua.

every array is an associative array

every object is an associative array

table is the only primitive data structure

```
hasNiceSyntaxForAssociativeArrays["Python"] = true
hasNiceSyntaxForAssociativeArrays["Java"]   = false
```

legal Python code

```
val hasNiceSyntaxForMutableMaps: mutable.Map[String,Boolean] = mutable.Map()
hasNiceSyntaxForMutableMaps("Scala") = true
```

Basic symbol table API

Associative array abstraction. Associate one value with each key.

public class ST<Key, Value>

	ST()	*create an empty symbol table*	Python:
void	put(Key key, Value val)	*put key-value pair into the table*	a[key] = val;
Value	get(Key key)	*value paired with key*	a[key]
boolean	contains(Key key)	*is there a value paired with key?*	
void	delete(Key key)	*remove key (and its value) from table*	
boolean	isEmpty()	*is the table empty?*	
int	size()	*number of key-value pairs in the table*	
Iterable<Key>	keys()	*all the keys in the table*	

In Java, this is called *keySet* and returns a *Set<Key>*

Conventions

- Values are not null. ← Java allows null value
- Method get() returns null if key not present.
- Method put() overwrites old value with new value. ← Java returns old value

Intended consequences.

- Easy to implement contains().

```
public boolean contains(Key key)
{  return get(key) != null;  }
```

- Can implement lazy version of delete().

```
public void delete(Key key)
{  put(key, null);  }
```

Keys and values

Value type: any type (typically defined as a generic).

Key type: any type (typically defined as a generic).

- Use equals() to test equality.
- Use hashCode() to scramble key.
- If keys are Comparable, use compareTo() to allow ordering of keys.

Best practices. Use immutable types for symbol table keys.

- Immutable in Java: Integer, Double, String, java.io.File, …
- Mutable in Java: StringBuilder, java.net.URL, arrays, ...

Hashing

- Arrays are really just too useful and efficient to ignore!
  - We dismissed arrays as a data structure before because we don't normally have an index *within a defined range* [recall that arrays have fixed size which implies that an index is within a defined range].
- Hashing
  - What if we could construct such an index from our key *k* with the following properties?
    - quick: *h(k)* can be evaluated in as short a time as possible;
    - deterministic: *h(k)* is a "pure" function (always yields the same value);
    - defined range: *lo <= h(k) <= hi* (where [*lo*, *hi*] defines the range of valid indices);
    - uniformity: p($h(k_1)$) ~= p(h(k_2)), for any pair of keys k_1, k_2;
  - The following properties are desirable or required for message digests (cryptographic hash functions) but not typically required for symbol table hashing:
    - volatility: if $h(k_1) \neq h(k_2)$ then $k_1 \neq k_2$.
    - unidirectional: It is infeasible to construct *k* from *h(k)*.
    - normalized: some properties of a key may be considered non-significant and therefore take no part in the hash function *h(k)*.

Hashable

- *Hashable* interface:
  - Although not strictly required for ordinary applications of hashing, it is generally true (*volatility*) that if $h(k_1) \mathrel{!=} h(k_2)$ then $k_1 \mathrel{!=} k_2$.
  - Because of the defined range property, it is also possible that $h(k_1) = h(k_2)$ even though $k_1 \mathrel{!=} k_2$. This is known as a *collision*.
  - Therefore, if we find two equal hash values $h(k_1) = h(k_2)$, we will normally want to determine whether $k_1 = k_2$. This requires the ability to test the equality of keys.
  - These two methods *equals*(k_1,k_2) and *hash*(k) should therefore be defined in an interface called *Hashable*.
  - For some reason, Java chose to build this interface into the *Object* class.

“Mapping” in Map-Reduce

- The n^{th} stage of the process looks like this:
  - $Map[K_{n-1},V_{n-1}] \rightarrow Map[K_n,Seq[W_n]] \rightarrow Map[K_n,V_n]$
    “mapper” “reducer(s)”
  - But, typically, the input data to the first stage has no natural key so we use the fact that $Map[K_0,V_0]$ can be transformed directly to $Seq[(K_0,V_0)]$ where $Seq[(\varnothing,V_0)]$ in turn is the equivalent of $Seq[V_0]$.
- Assuming, then, that each (n^{th}) stage of the pipeline works as expected, then overall, we can transform:
  - $Seq[V_0] \rightarrow Map[K_n,V_n]$

Equality test

All Java classes inherit a method equals().

Java requirements. For any references x, y and z:

- Reflexive: x.equals(x) is true.
- Symmetric: x.equals(y) iff y.equals(x).
- Transitive: if x.equals(y) and y.equals(z), then x.equals(z).
- Non-null: x.equals(null) is false.

equivalence relation

do x and y refer to the same object?

Default implementation. (x == y)

Customized implementations. Integer, Double, String, java.io.File, …

User-defined implementations. Some care needed.

Implementing equals for user-defined types

Seems easy.

```
public      class Date implements Comparable<Date>
{
   private final int month;
   private final int day;
   private final int year;
   ...

   public boolean equals(Date that)
   {

      if (this.day   != that.day  ) return false;
      if (this.month != that.month) return false;
      if (this.year  != that.year ) return false;
      return true;
```

check that all significant fields are the same

Implementing equals for user-defined types

Seems easy, but requires some care.

```
public final class Date implements Comparable<Date>
{
   private final int month;
   private final int day;
   private final int year;
   ...

   public boolean equals(Object y)
   {
      if (y == this) return true;

      if (y == null) return false;

      if (y.getClass() != this.getClass())
         return false;

      Date that = (Date) y;
      if (this.day   != that.day  ) return false;
      if (this.month != that.month) return false;
      if (this.year  != that.year ) return false;
      return true;
```

typically unsafe to use equals() with inheritance (would violate symmetry)

must be Object. Why? Experts still debate.

optimize for true object equality

check for null

objects must be in the same class (religion: getClass() vs. instanceof)

cast is guaranteed to succeed

check that all significant fields are the same

Equals design

"Standard" recipe for user-defined types.

- Optimization for reference equality.
- Check against null.
- Check that two objects are of the same type and cast.
- Compare each significant field:
  - if field is a primitive type, use == — but use Double.compare() with double (or otherwise deal with -0.0 and NaN)
  - if field is an object, use equals() — apply rule recursively
  - if field is an array, apply to each entry — can use Arrays.deepEquals(a, b) but not a.equals(b)

Best practices.

e.g., cached Manhattan distance

- No need to use calculated fields that depend on other fields.
- Compare fields mostly likely to differ first.
- Make compareTo() consistent with equals().

x.equals(y) if and only if (x.compareTo(y) == 0)

We’ve done this before…

- INFO6205 repo: *edu.neu.coe.info6205.equable*
    - *Equable.java*
    - *BaseEquable.java*
    - *ComparableEquable.java*
    - *BaseComparableEquable.java*

Equable.java

```
package edu.neu.coe.info6205.equable;
import java.util.Iterator;
public class Equable {
    public Equable(Iterable<?> elements) {
        this.elements = elements;
    }
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Equable equable = (Equable) o;
        Iterator<?> thisIterator = elements.iterator();
        Iterator<?> thatIterator = equable.elements.iterator();
        while (thisIterator.hasNext())
            if (thatIterator.hasNext())
                if (thisIterator.next().equals(thatIterator.next()))
                    continue;
                else
                    return false;
            else
                return false;
        return true;
    }
    @Override
    public int hashCode() {
        int result = 0;
        for (Object element : elements) result = 31 * result + element.hashCode();
        return result;
    }
    protected final Iterable<?> elements;
}
```

ST test client for traces

Build ST by associating value i with i^{th} string from standard input.

```
public static void main(String[] args)
{
   ST<String, Integer> st = new ST<String, Integer>();
   for (int i = 0; !StdIn.isEmpty(); i++)
   {
     String key = StdIn.readString();
     st.put(key, i);
   }
   for (String s : st.keys())
      StdOut.println(s + " " + st.get(s));
}
```

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

output

```
A  8
C  4
E  12
H  5
L  11
M  9
P  10
R  3
S  0
X  7
```

ST test client for analysis

Frequency counter. Read a sequence of strings from standard input and print out one that occurs with highest frequency.

```
% more tinyTale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness
it was the epoch of belief
it was the epoch of incredulity
it was the season of light
it was the season of darkness
it was the spring of hope
it was the winter of despair

% java FrequencyCounter 1 < tinyTale.txt
it 10

% java FrequencyCounter 8 < tale.txt
business 122

% java FrequencyCounter 10 < leipzig1M.txt
government 24763
```

tiny example
(60 words, 20 distinct)

real example
(135,635 words, 10,769 distinct)

real example
(21,191,455 words, 534,580 distinct)

Frequency counter implementation

```
public class FrequencyCounter
{
   public static void main(String[] args)
   {
      int minlen = Integer.parseInt(args[0]);
      ST<String, Integer> st = new ST<String, Integer>();
      while (!StdIn.isEmpty())
      {
         String word = StdIn.readString();
         if (word.length() < minlen) continue;
         if (!st.contains(word)) st.put(word, 1);
         else                    st.put(word, st.get(word) + 1);
      }
      String max = "";
      st.put(max, 0);
      for (String word : st.keys())
         if (st.get(word) > st.get(max))
            max = word;
      StdOut.println(max + " " + st.get(max));
   }
```

create ST

ignore short strings

read string and update frequency

print a string with max freq

http://algs4.cs.princeton.edu

3.1 Symbol Tables

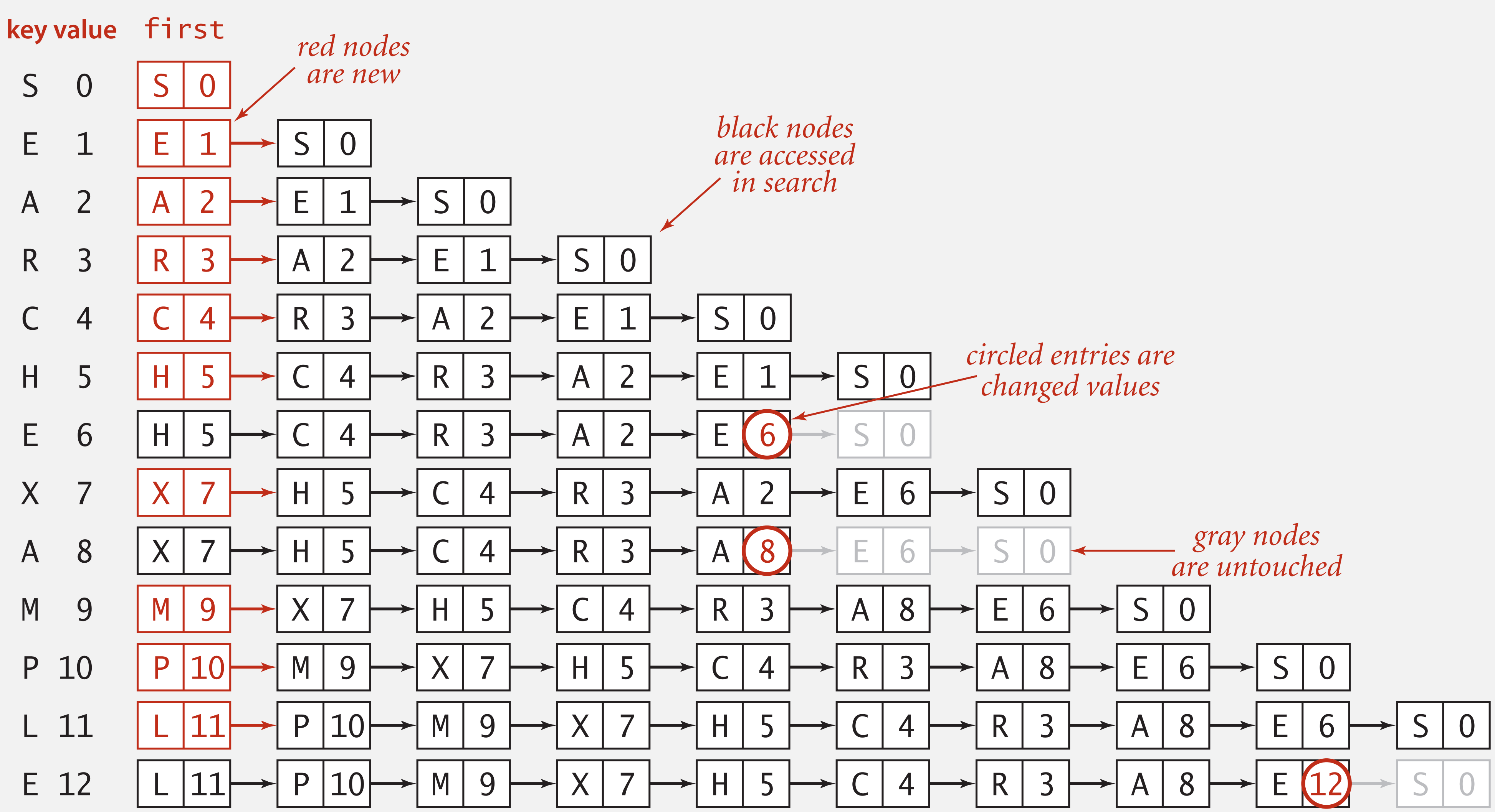
- API
- *elementary implementations*
- *ordered operations*

Sequential search in a linked list

Data structure. Maintain an (unordered) linked list of key-value pairs.

Search. Scan through all keys until find a match.

Insert. Scan through all keys until find a match; if no match add to front.



Trace of linked-list ST implementation for standard indexing client

Elementary ST implementations: summary

ST implementation	guarantee		average case		key interface
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	$N / 2$	N	equals()

Challenge. Efficient implementations of both search and insert.

Binary search in an ordered array

Data structure. Maintain an ordered array of key-value pairs.

Rank helper function. How many keys < k ?

"Rank" here relates to the *Order Statistics* we learned about for *quickselect*. It is not the same as rank in ordinary English.

keys[]

0	1	2	3	4	5	6	7	8	9
A	C	E	H	L	M	P	R	S	X

successful search for P

lo	hi	m										
0	9	4	A	C	E	H	L	M	P	R	S	X
5	9	7	A	C	E	H	L	M	P	R	S	X
5	6	5	A	C	E	H	L	M	P	R	S	X
6	6	6	A	C	E	H	L	M	P	R	S	X

entries in black are `a[lo..hi]`

entry in red is `a[m]`

loop exits with `keys[m] = P`: *return* 6

unsuccessful search for Q

lo	hi	m										
0	9	4	A	C	E	H	L	M	P	R	S	X
5	9	7	A	C	E	H	L	M	P	R	S	X
5	6	5	A	C	E	H	L	M	P	R	S	X
7	6	6	A	C	E	H	L	M	P	R	S	X

loop exits with `lo > hi`: *return* 7

Binary search: Java implementation

```
public Value get(Key key)
{
   if (isEmpty()) return null;
   int i = rank(key);
   if (i < N && keys[i].compareTo(key) == 0) return vals[i];
   else return null;
}
```

```
private int rank(Key key)
{
   int lo = 0, hi = N-1;
   while (lo <= hi)
   {
      int mid = lo + (hi - lo) / 2;
      int cmp = key.compareTo(keys[mid]);
      if      (cmp  < 0) hi = mid - 1;
      else if (cmp  > 0) lo = mid + 1;
      else if (cmp == 0) return mid;
  }
  return lo;
```

Binary search: trace of standard indexing client

Problem. To insert, need to shift all greater keys over.

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

entries in red were inserted

entries in gray did not move

entries in black moved to the right

circled entries are changed values

Elementary ST implementations: summary

ST implementation	guarantee		average case		key interface
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	$N / 2$	N	equals()
binary search (ordered array)	$\log N$	N	$\log N$	$N / 2$	compareTo()

Why?

Challenge. Efficient implementations of both search and insert.

http://algs4.cs.princeton.edu

3.1 Symbol Tables

- *API*
- *elementary implementations*
- *ordered operations*

Examples of ordered symbol table API

	keys	values
min()	09:00:00	Chicago
	09:00:03	Phoenix
	09:00:13	Houston
get(09:00:13)	09:00:59	Chicago
	09:01:10	Houston
floor(09:05:00)	09:03:13	Chicago
	09:10:11	Seattle
select(7)	09:10:25	Seattle
	09:14:25	Phoenix
	09:19:32	Chicago
	09:19:46	Chicago
keys(09:15:00, 09:25:00)	09:21:05	Chicago
	09:22:43	Seattle
	09:22:54	Seattle
	09:25:52	Chicago
ceiling(09:30:00)	09:35:21	Chicago
	09:36:14	Seattle
max()	09:37:44	Phoenix

size(09:15:00, 09:25:00) *is* 5

rank(09:10:25) *is* 7

Ordered symbol table API

public class ST<Key extends Comparable<Key>, Value>

	...	
Key	min()	*smallest key*
Key	max()	*largest key*
Key	floor(Key key)	*largest key less than or equal to key*
Key	ceiling(Key key)	*smallest key greater than or equal to key*
int	rank(Key key)	*number of keys less than key*
Key	select(int k)	*key of rank k*
void	deleteMin()	*delete smallest key*
void	deleteMax()	*delete largest key*
int	size(Key lo, Key hi)	*number of keys between lo and hi*
Iterable<Key>	keys()	*all keys, in sorted order*
Iterable<Key>	keys(Key lo, Key hi)	*keys between lo and hi, in sorted order*

Binary search: ordered symbol table operations summary

	sequential search	binary search
search	N	$\log N$
insert / delete	N	N
min / max	N	1
floor / ceiling	N	$\log N$
rank	N	$\log N$
select	N	1
ordered iteration	$N \log N$	N

See any problem here?

order of growth of the running time for ordered symbol table operations

Efficient symbol table?

- We've been here before. Search (with an ordered ST) is $O(\lg n)$ but building the ST in the first place is $O(n^2)$.
- What sort of techniques can we employ to bring the cost of building an ST down to $O(n \lg n)$?
- ?

Flatland

- *Flatland* by Edwin Abbott Abbott (1884). The narrator is called “A Square” (Abbott^2 ?).