ID1020: Symbol tables

Dr. Per Brand
pbrand@kth.se

kap 3.1

Slides adapted from Algoritms 4th Edition, Sedgewick.

Symbol tables

- Based on the key-value pair abstraction
- Basic operations
  - Put(Key, Value)
  - Value = Get(Key)

- Example
  - DNS lookup
  - Key is domain name
  - Value is IP address

domain name	IP address
www.cs.princeton.edu	128.112.136.11
www.princeton.edu	128.112.128.15
www.yale.edu	130.132.143.21
www.harvard.edu	128.103.060.55
www.simpsons.com	209.052.165.60

key

value

Examples

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 table API

- Associative array
  - Associate one value with each key

public class ST<Key, Value>

	ST()	*create a symbol table*	
void	put(Key key, Value val)	*put key-value pair into the table (remove* key *from table if value is* null*)*	a[key] = val;
Value	get(Key key)	*value paired with* key *(*null *if* key *is absent)*	a[key]
void	delete(Key key)		
boolean	contains(Key key)	*is there a value paired with* key*?*	
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*	

Our convention

- Values are not null
- Method get(key) returns null if key absent from table
- Method put(key,value) overwrites old value
- Could use this to implement contains and delete
-

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

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

Keys and values

- Values
  - Arbitrary generic type

- Keys – different cases
  - Keys are *comparable* use `compareTo()`
  - Keys are not and use `equals()`
  - Keys are not and use hashing (Not in this lecture)

- Best practice
  - Use immutable types for keys. Why?

Equals

- All Java classes inherit this method
- Reflexive, symmetric, and transitive
- Non-null `x.equals(null)` is `false`

- For user-defined types be careful
- Cannot (in general) use the builtin `equals`
- `Why?`

Example user-defined equals

- First attempt

- Dealing with pointer or reference equality

```
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;
   }
}
```

ch
fie

User-defined equals design

- Optimize for pointer (reference) equality
- Check against null
- Check that object are of same type
- Check all significant fields
  - Fields that are functions of other fields might be ignored
  - For each field
    - If primitive type use ==
    - If an object use `equals` (recursively)
    - If an array use equals on each entry
- Optimization: Check fields most likely to differ first
- Make *compartTo* consistent with *equals*

One application

- Frequency counter
  - Counting the frequency of words in a text document
  - Text documents of different sizes to test different implemenetations.

- Some interesting results when this was first done on famous English authors and playwrights
  - They were ranked
  - Very small differences between number 2 and 3, 3 and 4, and so on.
  - But number 1 beat number 2 by a large factor
  - Who do you think was number one ?

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

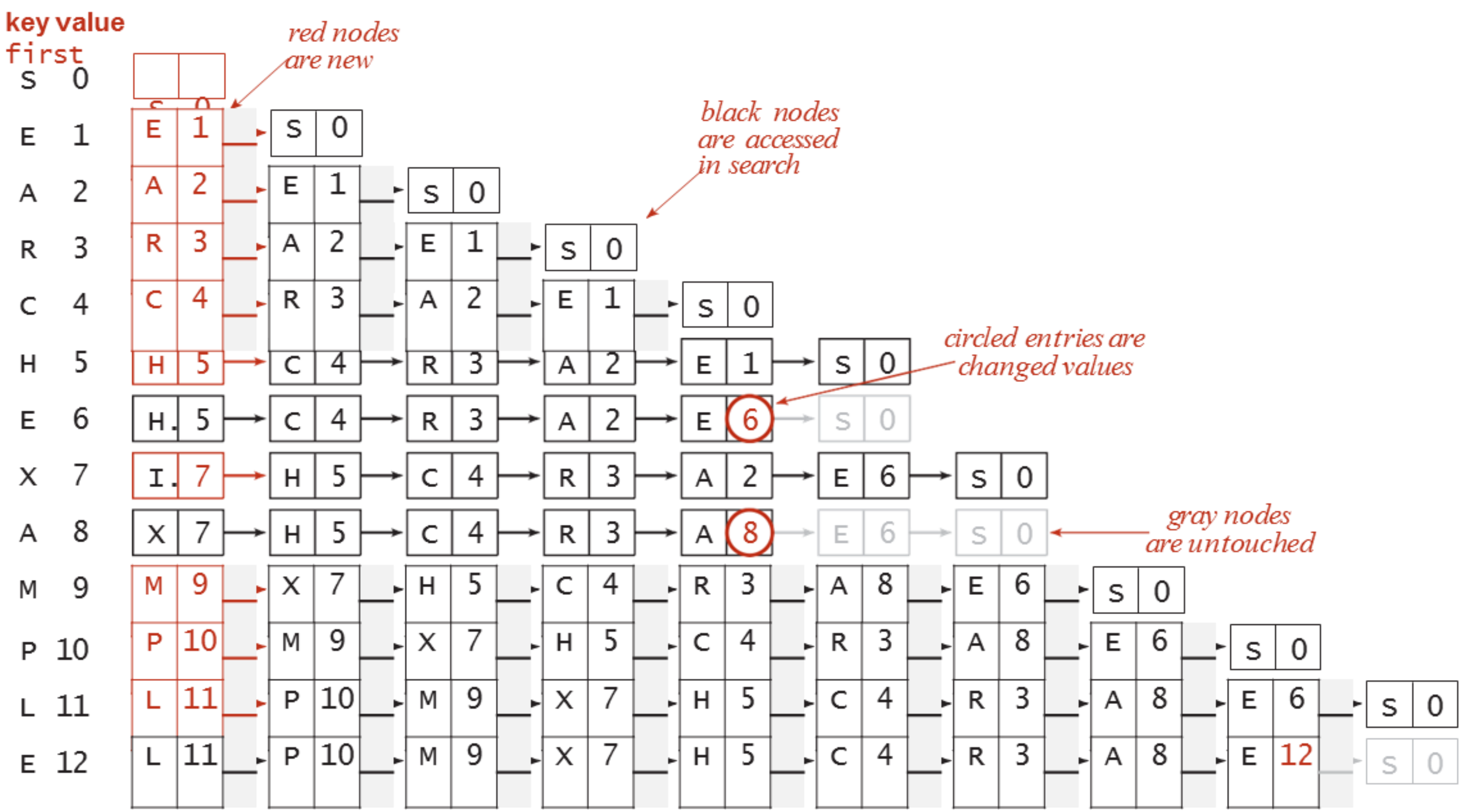
ignore short strings

read string and update frequency

print a string with max freq

Linked list implementation

- Unordered linked list of key-value pairs
- Get - Search through list
- Put – Search through list.
  - If match found overwrite, otherwise add



Complexity of linked list implementation

ST implementation	worst-case cost (after N inserts)		average case (after N random inserts)		ordered iteration?	key interface
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N / 2	N	no	`equals()`

- Note: We use only equals
- Can we do better?

Ordered array

- If we have *compareTo* we could use an ordered array

- Searching (get) can be done by binary search
  - Assume entry (if exists) is between indices lo and hi
  - Check the middle element mid = lo+hi / 2
  - Use compareTo to decide if

    Entry is in upper half, or lower half, or a hit

Rank

- Helper function – how many keys < k

keys[]

successful search for P

lo	hi	m	0	1	2	3	4	5	6	7	8	9
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	1	2	3	4	5	6	7	8	9
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

Search 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)                          number of keys <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;
}
```

Ordered array API

public class	ST<Key extends Comparable<Key>, Value>	
	ST()	*create an ordered symbol table*
void	put(Key key, Value val)	*put key-value pair into the table (remove* key *from table if value is* null*)*
Value	get(Key key)	*value paired with* key *(*null *if* key *is absent)*
void	delete(Key key)	*remove* key *(and its value) from table is there a*
boolean	contains(Key key)	*value paired with* key*?*
boolean	isEmpty()	*is the table empty?*
int	size()	*number of key-value pairs*
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 in* [lo..hi]
Iterable<Key>	keys(Key lo, Key hi)	*keys in* [lo..hi]*, in sorted order all keys in*
Iterable<Key>	keys()	*the table, in sorted order*

Example

	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

Complexity

- Note that ordered array is fine for gets but not for puts

- Next time we will remedy that

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