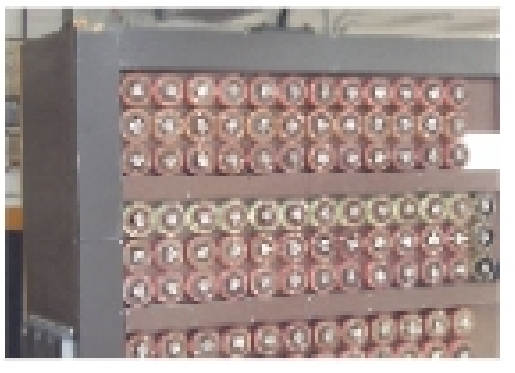
CS216: Program and Data Representation
University of Virginia Computer Science
Spring 2006 David Evans

Lecture 6: Ordered Data Abstractions



http://www.cs.virginia.edu/cs216

Why you should actually **read** course syllabi

Expected Background: Either (CS101, CS201, and CS202) or (CS150 with a B+ or better) or (Instructor Permission). Students entering CS216 are expected to have background in:

- **Programming:** comfortable creating programs that fill more than one screen, and understanding and modifying programs that involve multiple files. Students should be familiar with control structures commonly found in popular languages including decision and looping structures, and be comfortable with procedures and recursive definitions.
- **Mathematics and Logic:** ...

Schedule Update

- PS3 will be posted before midnight tomorrow
  - Review recursive definitions
  - Preparation for Exam 1
  - Read Chapter 6 (skip skip lists, we are skipping Ch 5 for now)
- Exam 1: out Feb 22, due Feb 27
  - Covers PS1-PS3, Lectures 1-8 (next Weds), book Ch 1-4, 6

Unordered Data Abstractions

- Our list and tree abstractions have structure (successor, children, etc.) but no notion that structure is associated with values
- What does this mean about the running time of a lookup operation?

Any operation that looks for an element based on element properties must have running time $\in \Omega(N)$ where N is # of elements

Ordered Data Abstractions

- To do better than $\Omega(N)$ we must be able to know something about where an element can be stored based on its value
  - Can find element without looking at all elements

Dictionary Data Abstraction

- Set of <key, value> pairs
- Operations:
  - MakeEmptyDictionary ()
    - Returns { }
  - Insert (K, V, S)
    - Add <K, V> to S
  - Lookup (K, S)
    - Return value associated with K in S
      - If $\langle K, I \rangle \in S$, return I

Is this enough?

Is this unambiguous?

Dictionary Operations

- MakeEmptyDictionary ()
  - Returns { }
- Insert (K, V, S)
  - If Lookup(K, S) $\neq \Lambda$, $S_{post} = S_{pre} \cup \{<K, V>\}$
  - Otherwise, error
- Lookup (K, S)
  - If $<K, I> \in S$, return I
  - Otherwise return Λ

Python's Dictionary Type

We used it in PS2 code:

```
memo = {}
        memo = MakeEmptyDictionary()
memo[k] = [resU, resV]
        Insert (k, [resU, resV], value, memo)
memo.has_key(k)
        (Lookup (k, memo) = Λ)
res = memo[makeKey (U,V)]
        res = Lookup (makeKey (U, V), memo)
```


Dictionary List Implementation

```
class Record:
    def __init__(self, k, v):
        self.key = k
        self.value = v
    def __str__(self):
        return "<" + str(self.key) + ", " + str(self.value) + ">"

class DictionaryList:
    def __init__(self): self.__node = None

    def lookup (self, key):
        if self.__node == None: return None
        else: return self.__node.lookup (key)

    def insert (self, key, value):
        if self.__node == None:
            self.__node = DictionaryNode (Record (key, value))
        else: self.__node.insert (Record (key, value))
```


Dictionary Node

```
class DictionaryNode:
    def __init__(self,info):
        self.__info = info
        self.__next = None

    # pre: key must not be a key in self
    # post: self_post = {self[0], ...,self[|self| - 1], value}
    # modifies nothing
    def insert (self, value):
        current = self
        while not current.__next == None:
            current = current.__next
        current.__next = DictionaryNode (value)
```


Dictionary Lookup

```
def lookup (self, key):
    if self.__info.key == key:
        return self.__info.value
    else:
        if self.__next == None: return None
        else: return self.__next.lookup (key)
```

What is the asymptotic running time?

$\Theta(N)$ where N is the number of dictionary records

Improving (?) Dictionary

- Order the entries by key
- Stop looking once you get past a key that must be after the lookup key

- Costs: More complex code
  insert is more expensive?
- Benefits: Faster lookup?

Lookup

```
def lookup (self, key):
    if self.__info.key == key:
        return self.__info.value
    elif self.__info.key > key:
        return None
    else:
        if self.__next == None:
            return None
        else:
            res = self.__next.lookup (key)
            return res
```

How does this affect the running time?

Insert

```
class DictionaryOrderedList:
    def insert (self, key, value):
        rec = DictionaryOrderedNode \
                (Record (key, value))
        if self.__node == None: self.__node = rec
        else:
            if key < self.__node._info.key:
                rec._next = self.__node
                self.__node = rec
            else:
                self.__node.insert (Record (key, value))
```


Insert Node Code

```
# pre: key must not be a key in self, key must not be before self's
# post: self_post = {self[0], self[1], ...,self[|self| - 1], value}
# modifies nothing
def insert (self, record):
    current = self
    assert (record.key > current._info.key)
    while not current._next == None:
        if current._next._info.key > record.key:
            break
        current = current._next

    r = DictionaryOrderedNode (record)
    r._next = current._next
    current._next = r
```

How does this affect the running time?

Summary

- Costs:
  - Code size increased by 30%
- Benefits:
  - No growth difference:
    - insert and lookup are still $\Theta(N)$
  - Some absolute difference:
    - Average calls to lookup a non-existent key:

      $N \rightarrow N/2$

More Structure

- Current implementation: each comparison eliminates one element
- Ideal comparison implementation: each comparison eliminates half the elements

If our comparison function has Boolean output, can't do better than eliminating half!

ContinuousTable

```
# invariant: Records in items are sorted on key by <.
def lookup(self, key):
    def lookuprange(items):
        if len(items) == 0: return None
        if len(items) == 1:
            if items[0].key == key: return items[0].value
            else: return None
        middle = len(items) / 2
        if key < items[middle].key:
            return lookuprange (items[:middle])
        else:
            return lookuprange (items[middle:])
    return lookuprange(self.items)
```