Lecture 6: Ordered Data Abstractions

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

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?

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 \(<K, V> \in S\), return \(V\)

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: ...
Dictionary Operations

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

Python’s Dictionary Type

We used it in PS2 code:

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

Dictionary List Implementation

```python
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

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

def insert(self, value):
    current = self
    while not current.__next == None:
        current = current.__next
    current.__next = DictionaryNode(value)

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)
```

Dictionary Lookup

```python
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

```python
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

```python
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))
```

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:
      \( N \to N/2 \)

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

```python
# invariant: Records in items are sorted on key by <.
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:])
```

How does this affect the running time?
What is the maximum number of calls to `lookuprange`?

### Ordered Binary Tree

![Ordered Binary Tree Diagram]

Invariant: \( \text{node.left.key} < \text{node.info.key} \)
\( \text{node.right.key} > \text{node.info.key} \)

### Tree Lookup

```python
def lookup(self, k):
    if self.value == k:
        return self
    elif k < self.value:
        return self.left.lookup(k)
    else:
        return self.right.lookup(k)
```

If tree is well-balanced:
\[ N = 2^{h+1} - 1 \]
\[ h \in \Theta(\log N) \]

### Tree Lookup Analysis

Max number of calls: height of tree

### Tree Lookup Iterative

```python
def lookup(self, k):
    current = self
    while not current == None:
        if current.value == k:
            return current
        elif current.left == None:
            return None
        elif current.right == None:
            return None
        else:
            return current.left.lookup(k)
```
Comparison

\[ \begin{align*}
N &= \text{number of nodes in self} \\
h &= \text{height of self}
\end{align*} \]

```
def lookup(self, k):
    if self.value == k:
        return self
    elif k < self.value:
        if self.left == None:
            return None
        else:
            return self.left.lookup(k)
    else:
        if self.right == None:
            return None
        else:
            return self.right.lookup(k)
```

```
def lookup(self, k):
    current = self
    while current != None:
        if current.value == k:
            return current
        elif k < current.value:
            current = current.left
        else:
            current = current.right
    return None
```

<table>
<thead>
<tr>
<th>Code size</th>
<th>12 lines</th>
<th>9 lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max running time</td>
<td>( \Theta(h) )</td>
<td>( \Theta(h) )</td>
</tr>
<tr>
<td>Max space use</td>
<td>( h ) stack frames</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

Worst Running Time

\[ h = N \]

```
running time of lookup \( \in \Theta(N) \)
```

Later in the course, we’ll learn some techniques for keeping trees balanced. Until then, let’s hope we are usually not unlucky (or being attacked\(^1\)).


Charge

- Read Chapter 6
  - You can skip the skip lists section
- PS3 will be posted tomorrow
- Monday:
  - Greedy Algorithms
- Later in the course:
  - More efficient dictionary implementations
  - Python’s provides lookup with running time approximately in \( O(1) \)! (as PS2 #5 asks you to assume)