Advanced Topics

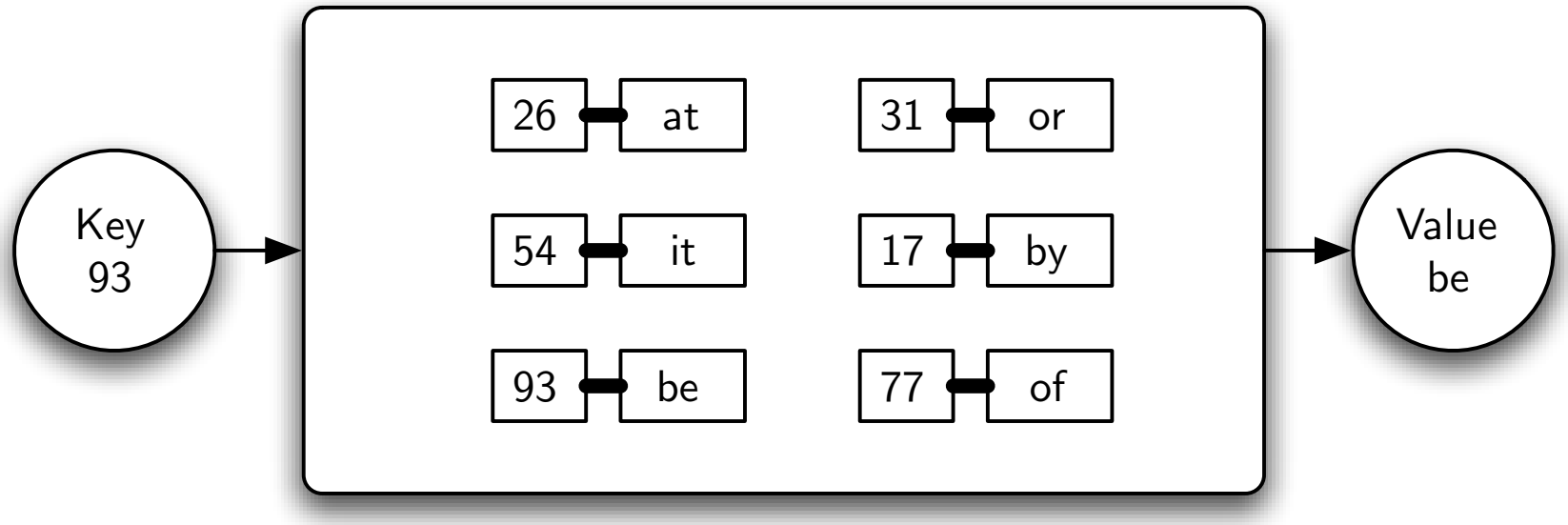
Skip Lists, OctTrees, and Pattern Matching

Brad Miller David Ranum

1/25/06

Outline

An Example Map

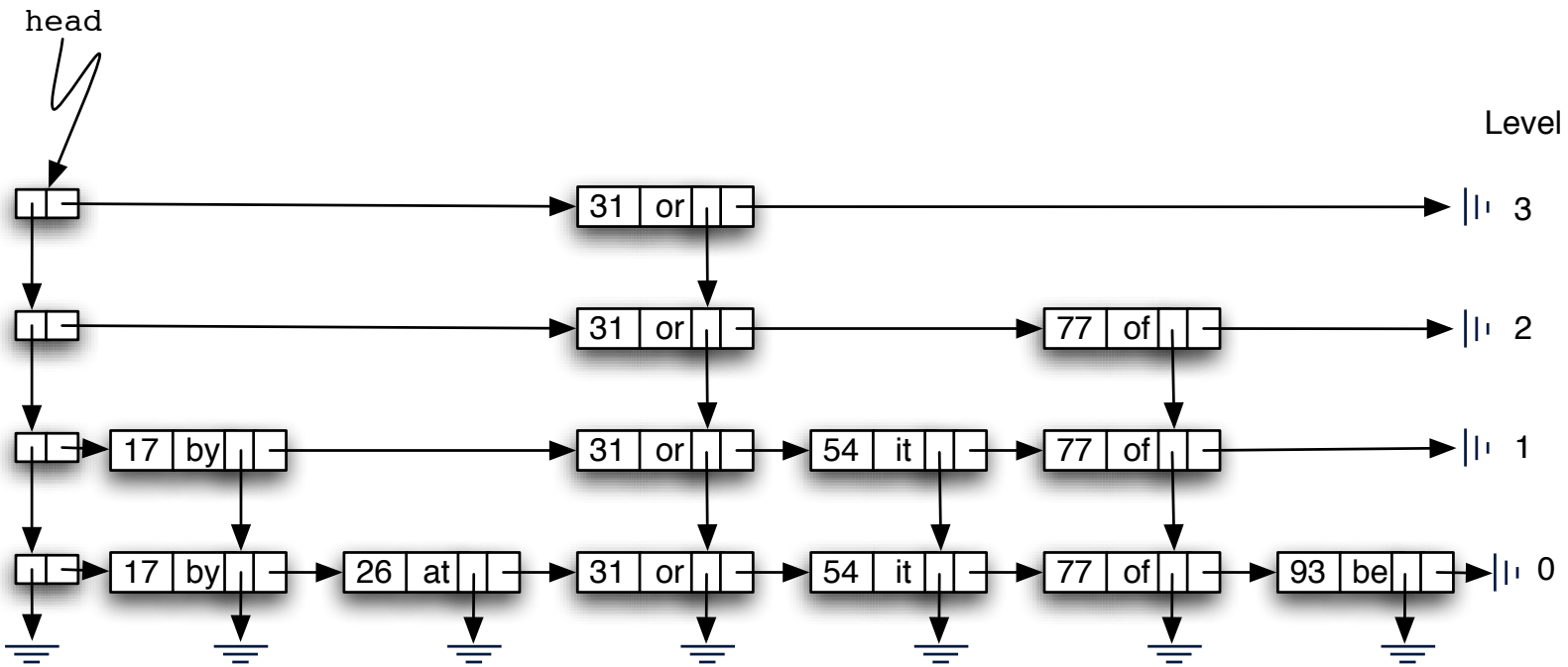


Outline

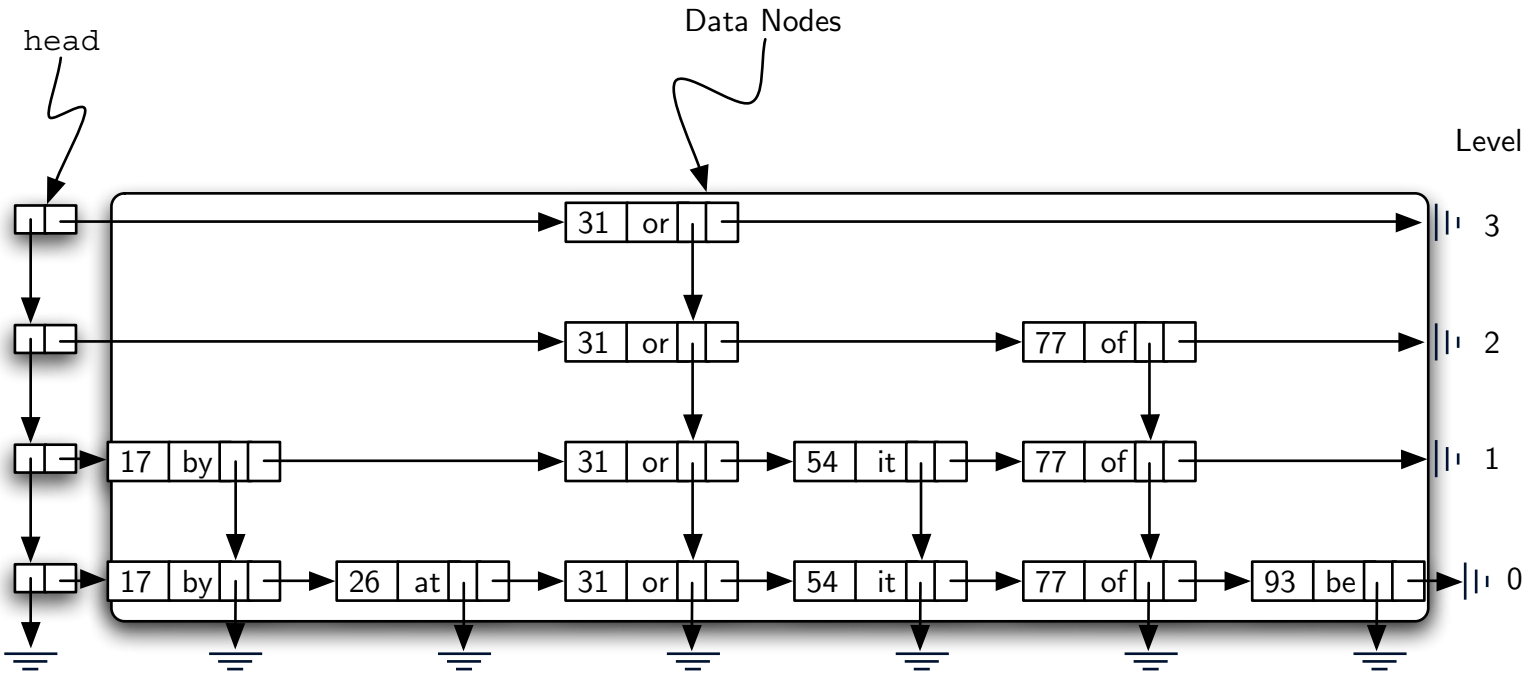
- `Map()` creates a new map that is empty. It needs no parameters and returns an empty map.
- `put(key,value)` adds a new key-value pair to the map. It needs the key and the associated value and returns nothing. Assume the key is not already in the map.
- `get(key)` searches for the key in the map and returns the associated value. It needs the key and returns a value.

Outline

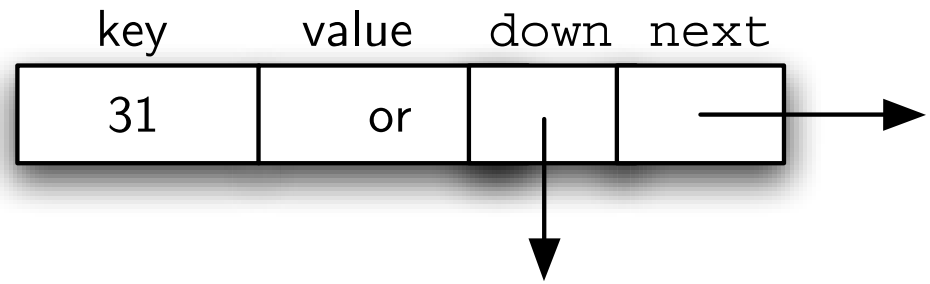
An Example Skip List



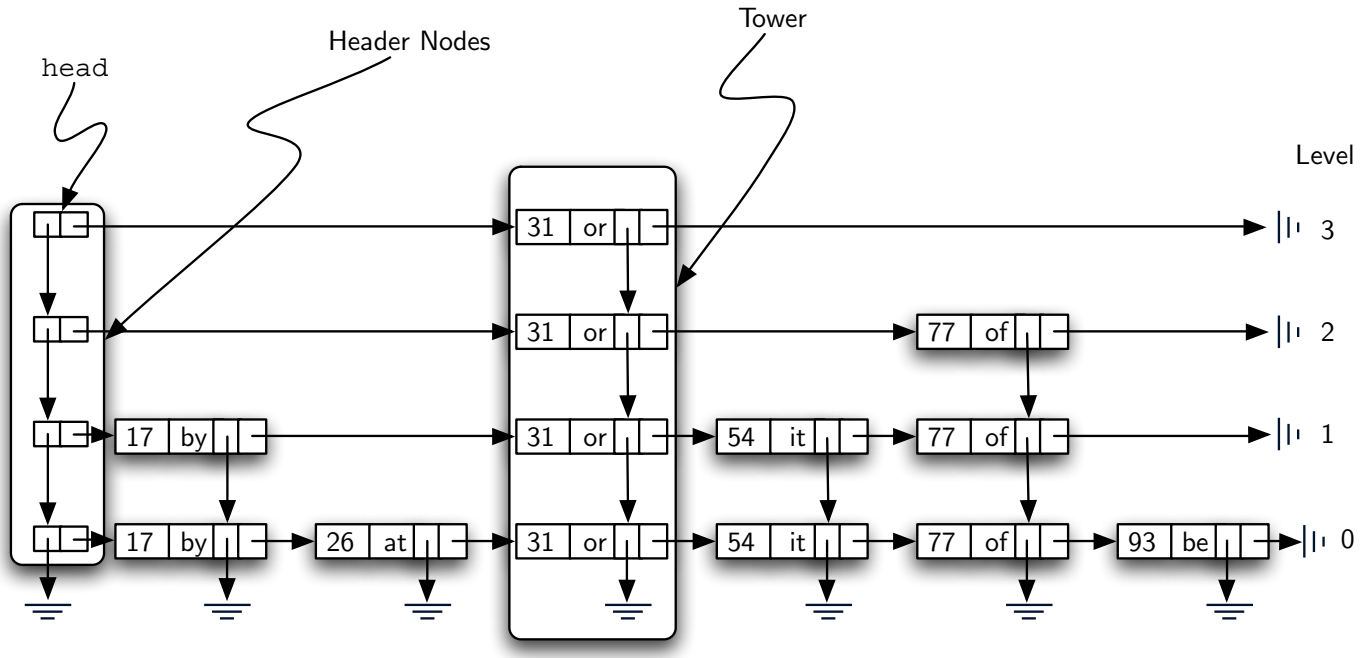
The Body of the Skip List Is Made Up of Data Nodes



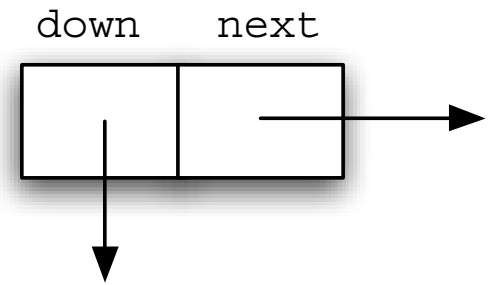
A Single Data Node



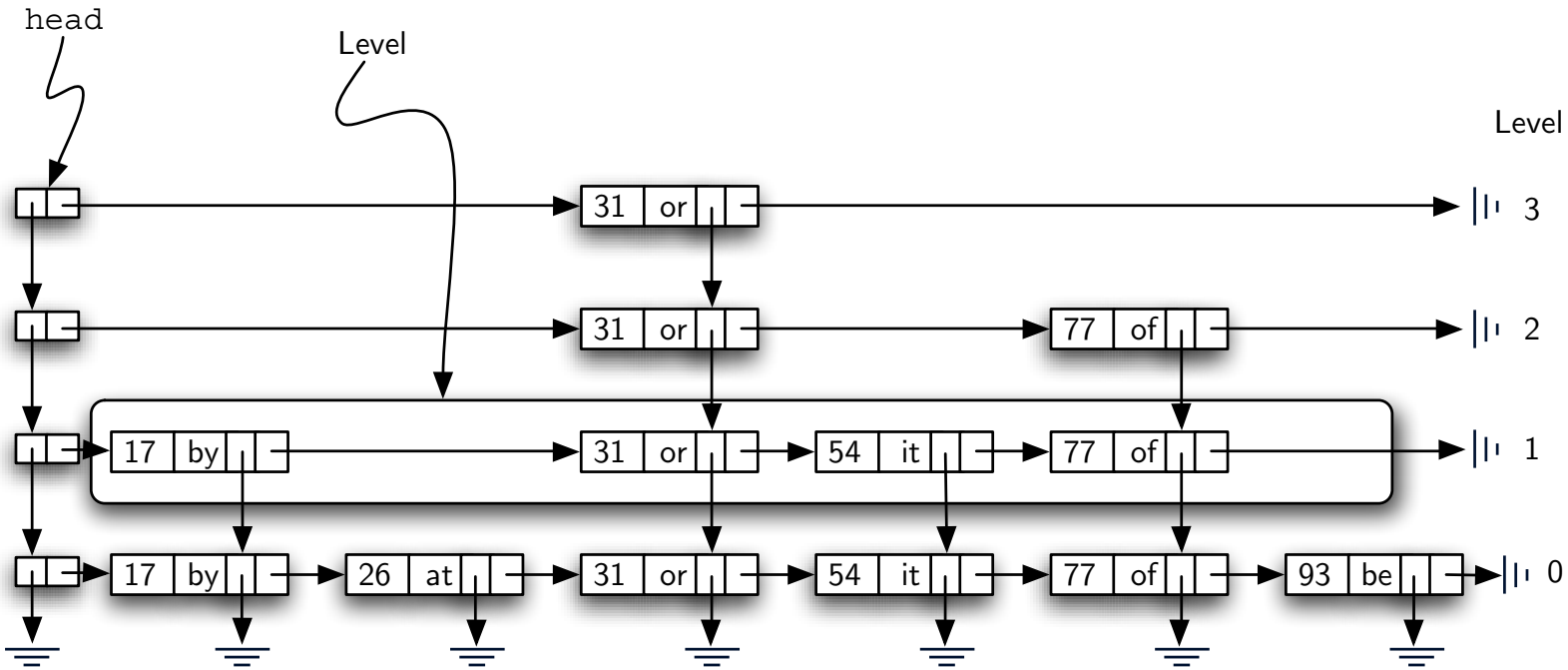
Header Nodes and Towers



Each Header Node Holds Two References



Each Horizontal Group of Data Nodes Is a Level



The `HeaderNode` Class

```
1  class HeaderNode:
2      def __init__(self):
3          self.next = None
4          self.down = None
5
6      def getNext(self):
7          return self.next
8
9      def getDown(self):
10         return self.down
11
12     def setNext(self,newnext):
13         self.next = newnext
14
15     def setDown(self,newdown):
16         self.down = newdown
```

The `DataNode` Class I

```
1  class DataNode:
2      def __init__(self,key,value):
3          self.key = key
4          self.data = value
5          self.next = None
6          self.down = None
7
8      def getKey(self):
9          return self.key
10
11     def getData(self):
12         return self.data
13
14     def getNext(self):
15         return self.next
16
```

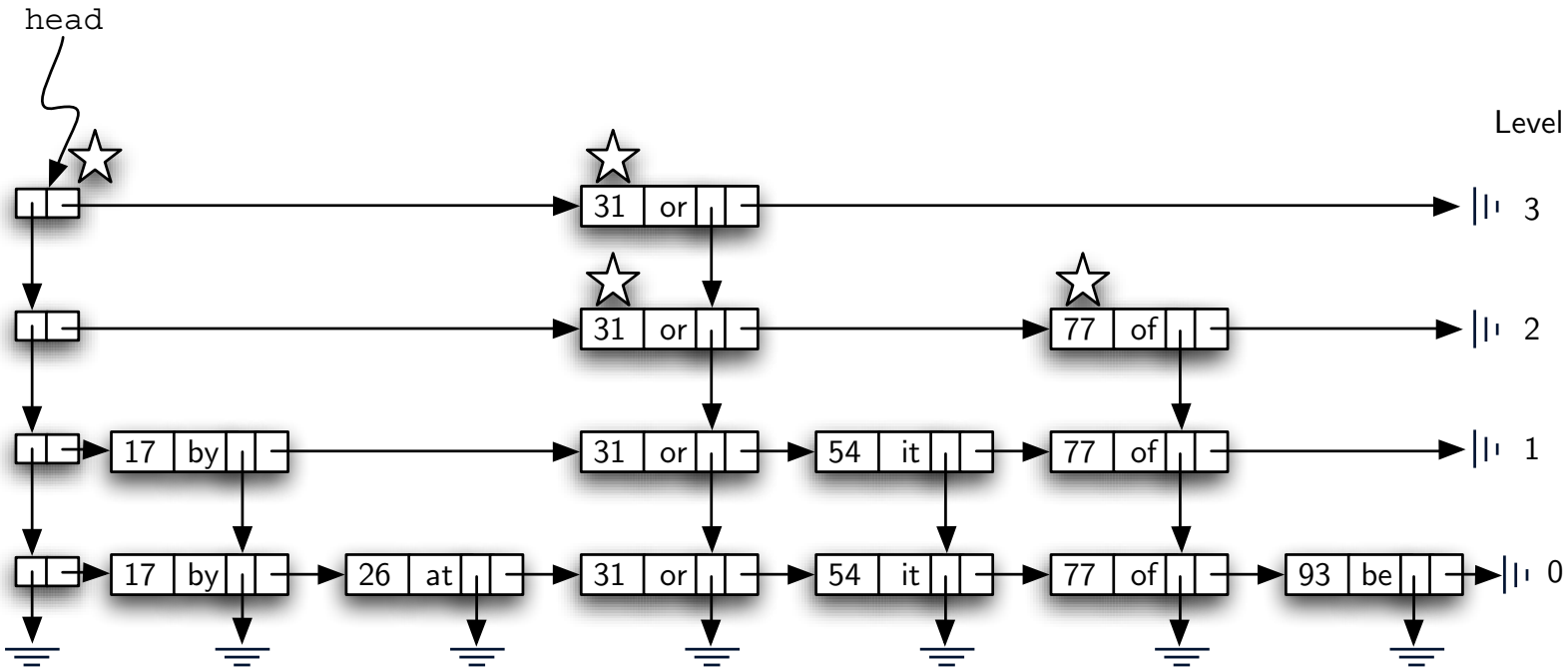
The `DataNode` Class II

```
17      def getDown(self):
18          return self.down
19
20      def setData(self,newdata):
21          self.data = newdata
22
23      def setNext(self,newnext):
24          self.next = newnext
25
26      def setDown(self,newdown):
27          self.down = newdown
```

The `SkipList` Constructor

```
1  class SkipList:
2      def __init__(self):
3          self.head = None
```

Searching for the Key 77



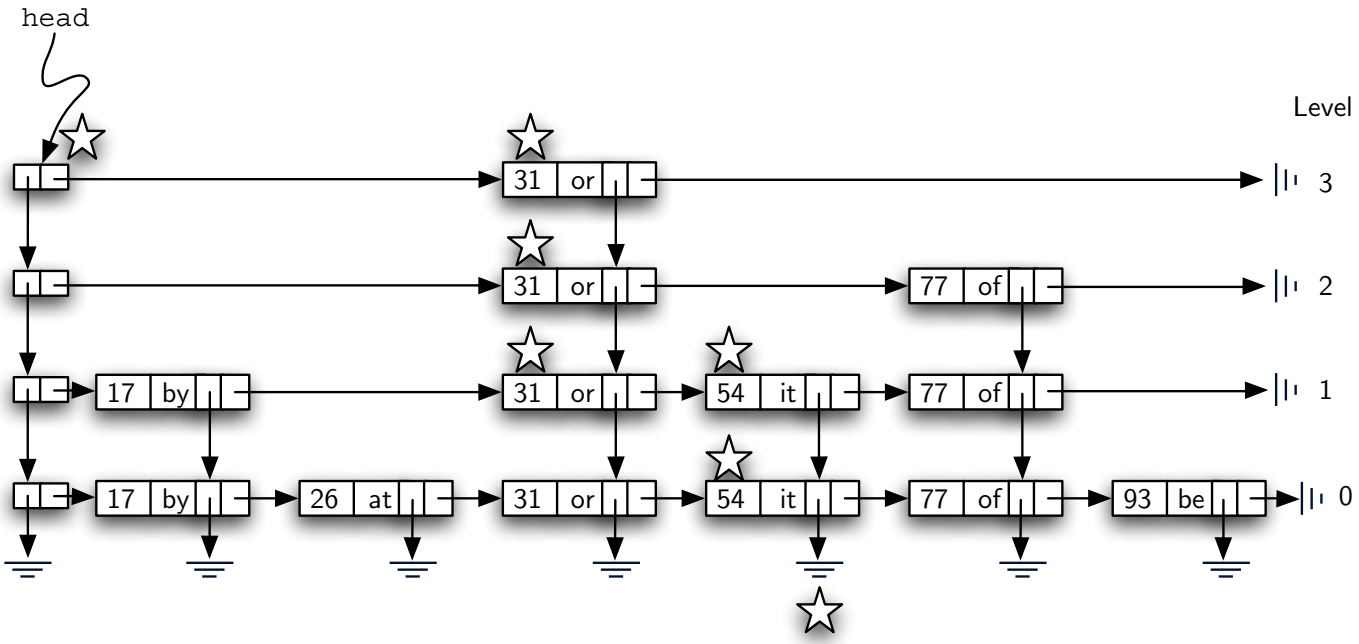
The `search` Method. I

```
1  def search(self,key):
2      current = self.head
3      found = False
4      stop = False
5      while not found and not stop:
6          if current == None:
7              stop = True
8          else:
9              if current.getNext() == None:
10                 current = current.getDown()
11             else:
12                 if current.getNext().getKey() == key:
13                     found = True
14                 else:
15                     if key < current.getNext().getKey():
16                         current = current.getDown()
```

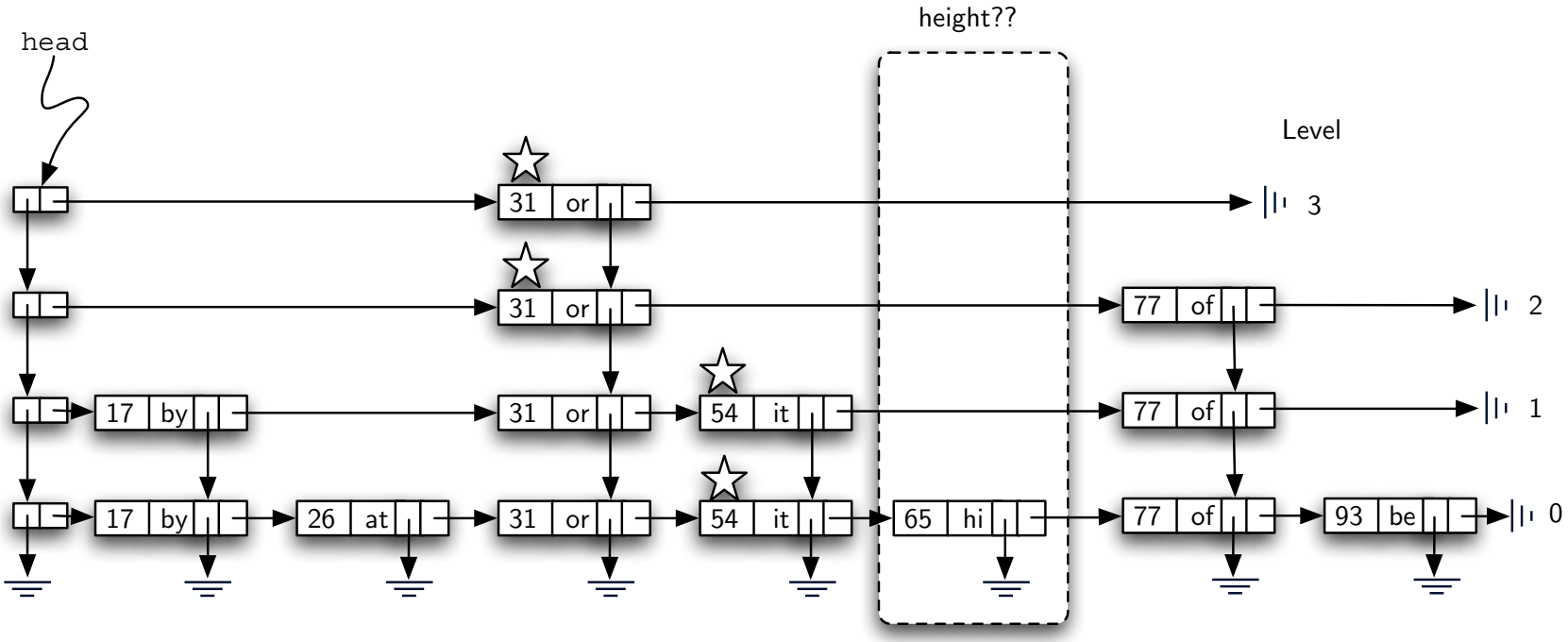
The `search` Method. II

```
17                          else:
18                              current = current.getNext()
19          if found:
20              return current.getNext().getData()
21          else:
22              return None
```

Searching for the Key 65



Adding the Data Node and Tower for 65



A Method to Flip a Coin

```
1  from random import randrange
2  def flip():
3      return randrange(2)
```

The Insert Method: Part 1

```
1  def insert(self,key,data):
2      if self.head == None:
3          self.head = HeaderNode()
4          temp = DataNode(key,data)
5          self.head.setNext(temp)
6          top = temp
7          while flip() == 1:
8              newhead = HeaderNode()
9              temp = DataNode(key,data)
10             temp.setDown(top)
11             newhead.setNext(temp)
12             newhead.setDown(self.head)
13             self.head = newhead
14             top = temp
15     else:
```

The Insert Method: Part 2 I

```
1           towerStack = Stack()
2           current = self.head
3           stop = False
4           while not stop:
5               if current == None:
6                   stop = True
7               else:
8                   if current.getNext() == None:
9                       towerStack.push(current)
10                      current = current.getDown()
11                  else:
12
13
14
15
16
```

The Insert Method: Part 2 II

```
17                      if current.getNext().getKey() > key:
18                          towerStack.push(current)
19                          current = current.getDown()
20                      else:
21                          current = current.getNext()
22
23              lowestLevel = towerStack.pop()
24              temp = DataNode(key,data)
25              temp.setNext(lowestLevel.getNext())
26              lowestLevel.setNext(temp)
27              top = temp
```

The Insert Method: Part 3

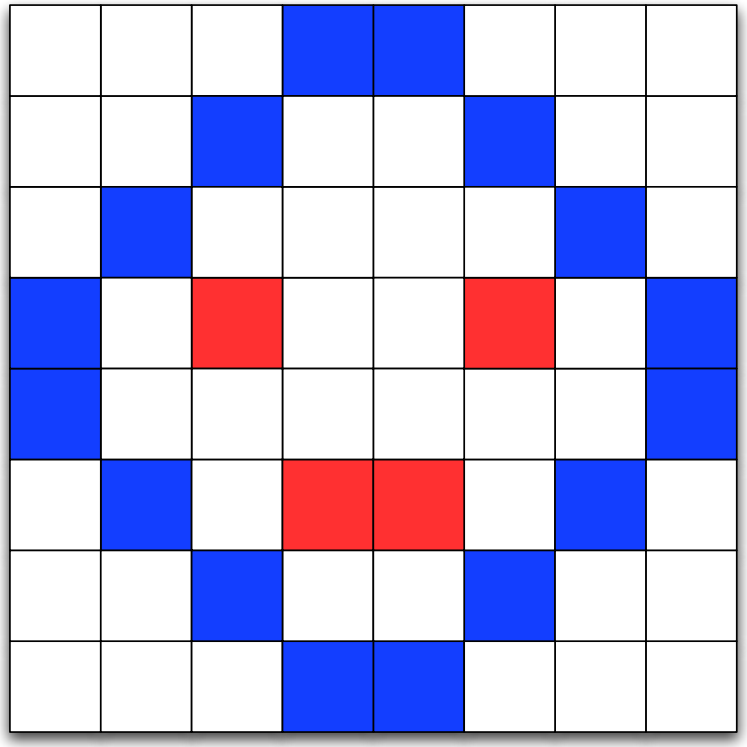
```
1           while flip() == 1:
2               if towerStack.isEmpty():
3                   newhead = HeaderNode()
4                   temp = DataNode(key,data)
5                   temp.setDown(top)
6                   newhead.setNext(temp)
7                   newhead.setDown(self.head)
8                   self.head = newhead
9                   top = temp
10              else:
11                  nextLevel = towerStack.pop()
12                  temp = DataNode(key,data)
13                  temp.setDown(top)
14                  temp.setNext(nextLevel.getNext())
15                  nextLevel.setNext(temp)
16                  top = temp
```

The `Map` Class Implemented Using Skip Lists

```
1  class Map:
2      def __init__(self):
3          self.collection=SkipList()
4
5      def put(self,key,value):
6          self.collection.insert(key,value)
7
8      def get(self,key):
9          return self.collection.search(key)
```

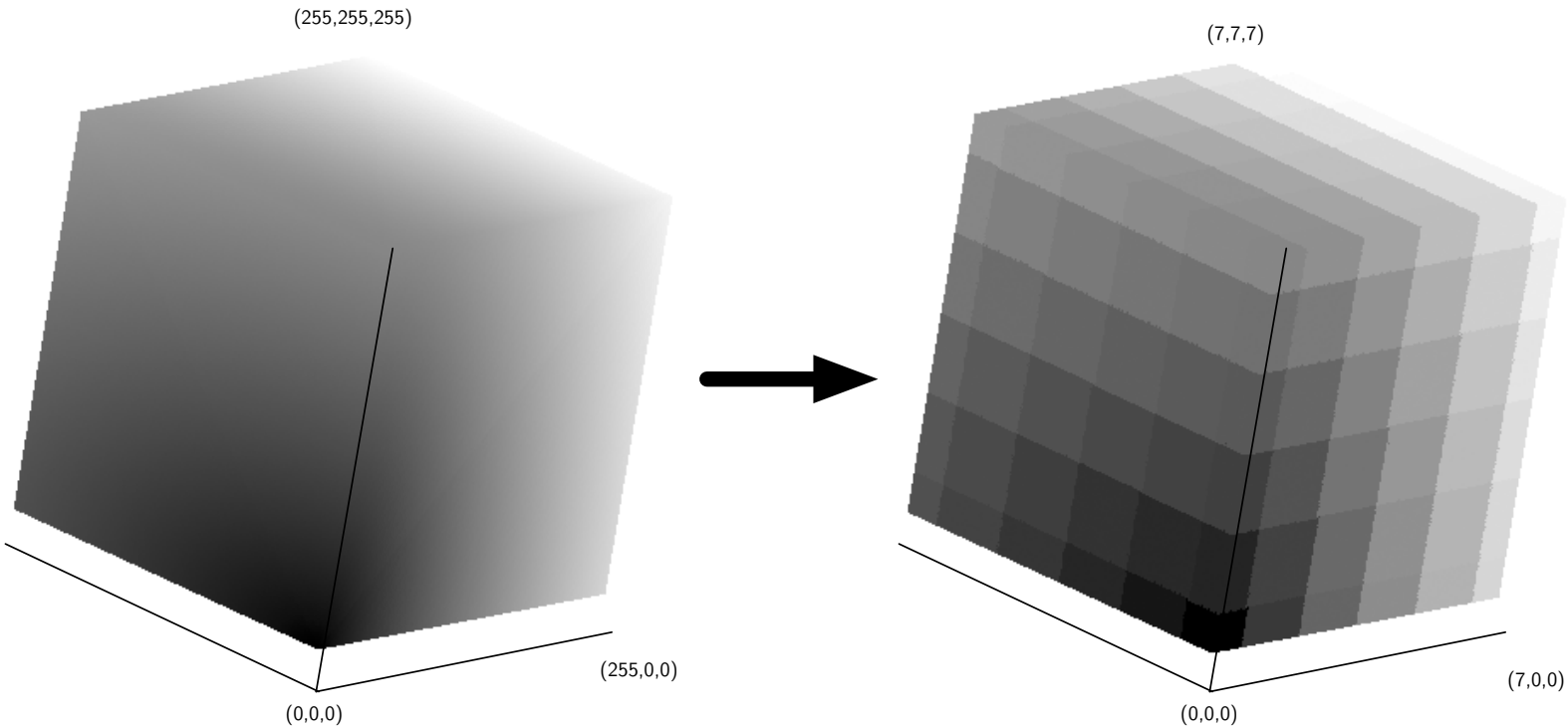
Outline

A Simple Image



Outline

Color Quantization



Simple Quantization Algorithm

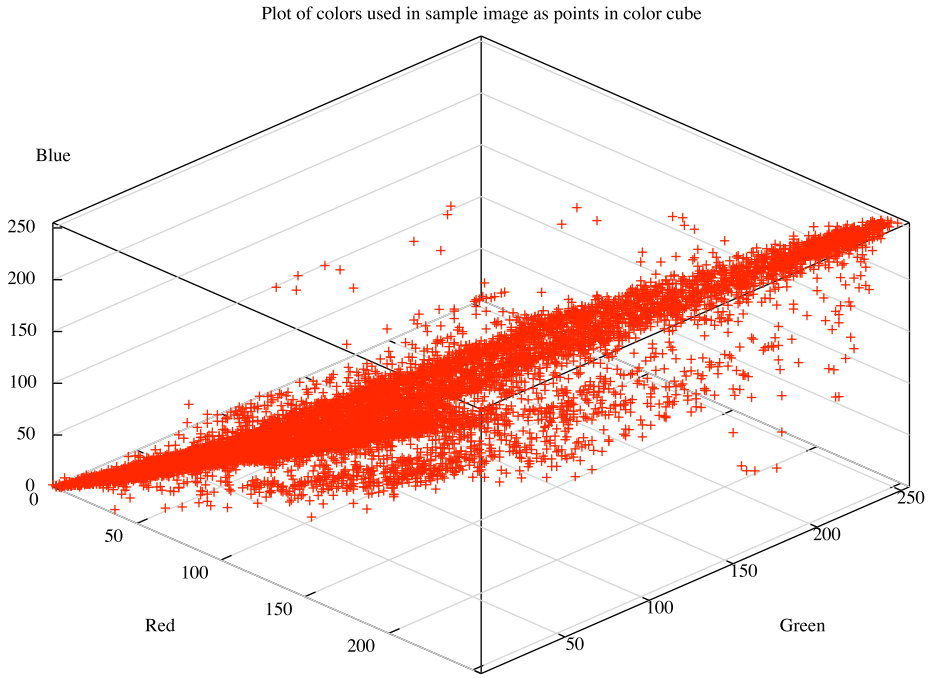
```
1  import sys
2  import os
3  import Image
4  def simpleQuant():
5      im = Image.open('bubbles.jpg')
6      w,h = im.size
7      for row in range(h):
8          for col in range(w):
9              r,g,b = im.getpixel((col,row))
10             r = r / 36 * 36
11             g = g / 42 * 42
12             b = b / 42 * 42
13             im.putpixel((col,row),(r,g,b))
14     im.show()
```

Original Image

Image Quantized with `simpleQuant`

Outline

Plot of Colors Used in Image as Points in Color Cube



- `OctTree()` Create a new empty `OctTree`.
- `insert(r,g,b)` Add a new node to the `OctTree` using the red, green, and blue color values as the key.
- `find(r,g,b)` Find an existing node, or the closest approximation, using the red, green, and blue color values as the search key.
- `reduce(n)` Reduce the size of the `OctTree` so that there are *n* or fewer leaf nodes.

The `OctTree` Class I

```
1       class OctTree:
2       def __init__(self):
3           self.root = None
4           self.maxLevel = 5
5           self.numLeaves = 0
6           self.leafList = []
7
8       def insert(self,r,g,b):
9           if not self.root:
10              self.root = self.otNode(outer=self)
11          self.root.insert(r,g,b,0,self)
12
13      def find(self,r,g,b):
14          if self.root:
15              return self.root.find(r,g,b,0)
16
```

The `OctTree` Class II

```
17      def reduce(self,maxCubes):
18          while len(self.leafList) > maxCubes:
19              smallest = self.findMinCube()
20              smallest.parent.merge()
21              self.leafList.append(smallest.parent)
22              self.numLeaves = self.numLeaves + 1
23
24
25
26
27
28
29
30
31
32
33
```

The `OctTree` Class III

```
34
35      def findMinCube(self):
36          minCount = sys.maxint
37          maxLev = 0
38          minCube = None
39          for i in self.leafList:
40              if i.count <= minCount and i.level >= maxLev:
41                  minCube = i
42                  minCount = i.count
43                  maxLev = i.level
44          return minCube
```

The `otNode` Class and Constructor

```
1   class otNode:
2   def __init__(self,parent=None,level=0,outer=None):
3       self.red = 0
4       self.green = 0
5       self.blue = 0
6       self.count = 0
7       self.parent = parent
8       self.level = level
9       self.oTree = outer
10      self.children = [None]*8
```

otNode insert I

```
1  def insert(self,r,g,b,level,outer):
2      if level < self.oTree.maxLevel:
3          idx = self.computeIndex(r,g,b,level)
4          if self.children[idx] == None:
5              self.children[idx] = outer.otNode(parent=self,
6                                                level=level+1,
7                                                outer=outer)
8          self.children[idx].insert(r,g,b,level+1,outer)
9      else:
10         if self.count == 0:
11             self.oTree.numLeaves = self.oTree.numLeaves + 1
12             self.oTree.leafList.append(self)
13         self.red += r
14         self.green += g
15         self.blue += b
16         self.count = self.count + 1
```

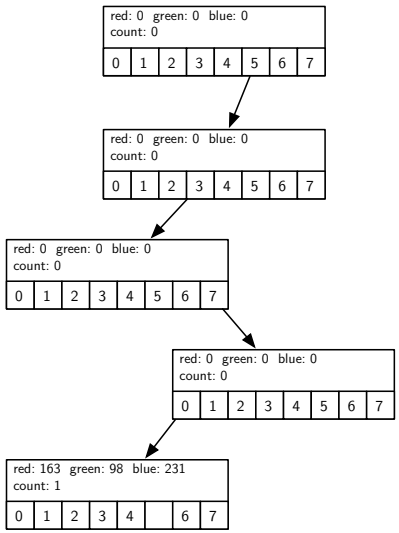
otNode insert II

```
17
18  def computeIndex(self,r,g,b,level):
19      shift = 8 - level
20      rc = r >> shift-2 & 0x4
21      gc = g >> shift-1 & 0x2
22      bc = b >> shift & 0x1
23      return(rc | gc | bc)
```

Computing an Index to Insert a Node in an OctTree

Red	1	0	1	0	0	0	1	1	163
Green	0	1	1	0	0	0	1	0	98
Blue	1	1	1	0	0	1	1	1	231

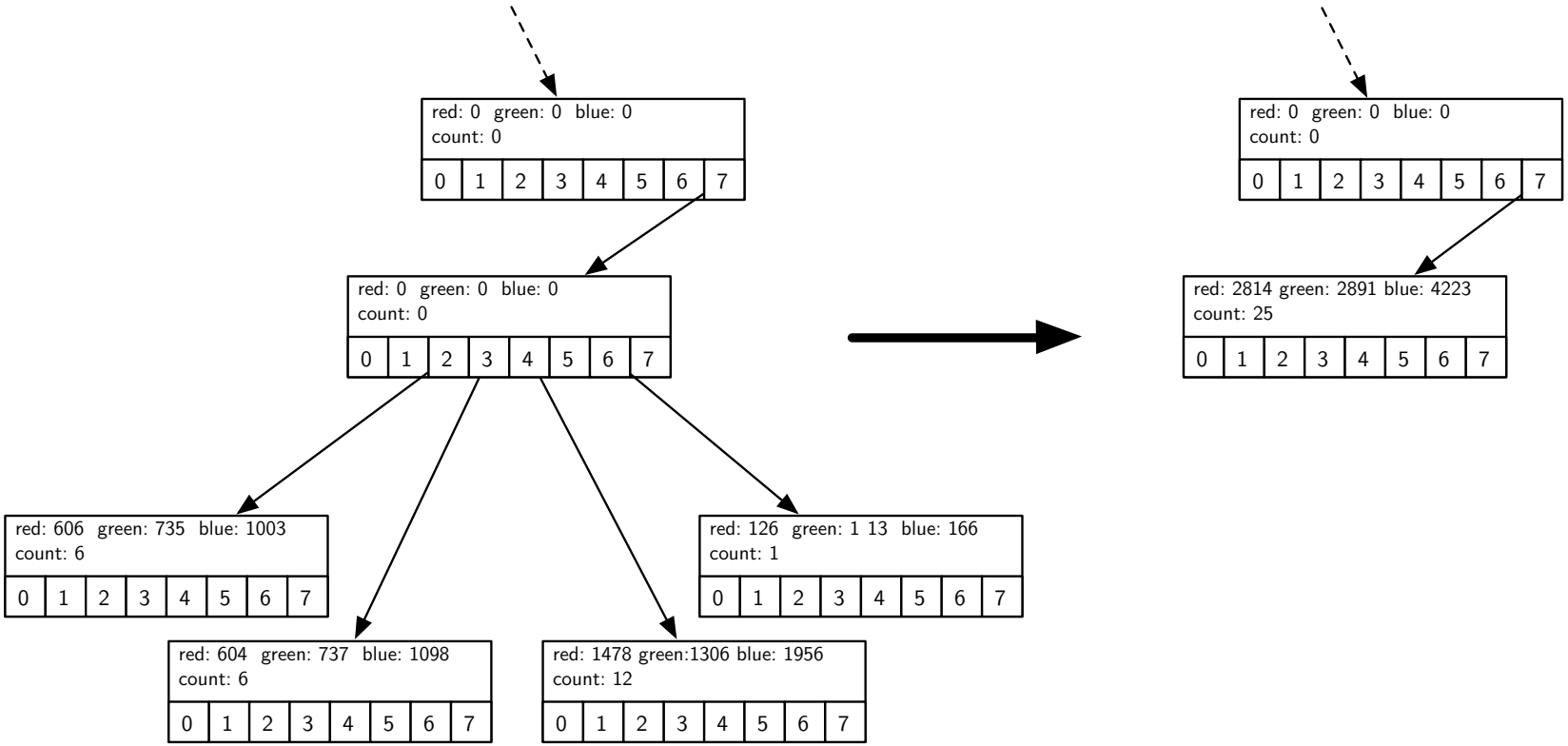
Red	Green	Blue	Level Index
1	0	1	5
0	1	1	3
1	1	1	7
0	0	0	0
0	0	0	0
0	0	1	1
1	1	1	7
1	0	1	5



otNode find

```
1  def find(self,r,g,b,level):
2      if level < self.oTree.maxLevel:
3          idx = self.computeIndex(r,g,b,level)
4          if self.children[idx]:
5              return self.children[idx].find(r,g,b,level+1)
6          elif self.count > 0:
7              return ((self.red/self.count,
8                       self.green/self.count,
9                       self.blue/self.count))
10         else:
11             print "error: No leaf node for this color"
12     else:
13         return ((self.red/self.count,
14                  self.green/self.count,
15                  self.blue/self.count))
```

Merging 4 Leaf Nodes of an `OctTree`



otNode Merge

```
1  def merge(self):
2      for i in self.children:
3          if i:
4              if i.count > 0:
5                  self.oTree.leafList.remove(i)
6                  self.oTree.numLeaves -= 1
7              else:
8                  print "Recursively Merging non-leaf..."
9                  i.merge()
10             self.count += i.count
11             self.red += i.red
12             self.green += i.green
13             self.blue += i.blue
14     for i in range(8):
15         self.children[i] = None
```

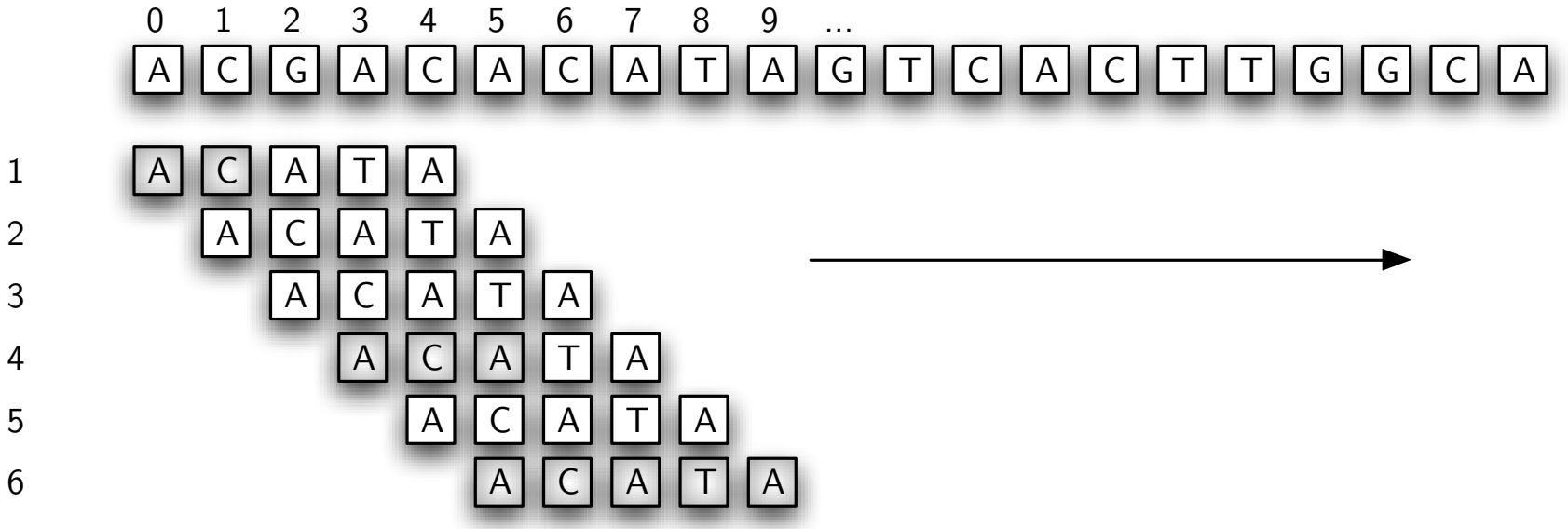
Original Image

Image Quantized with `OctTree`

Outline

Outline

A Simple Pattern-Matching Algorithm



A Simple Pattern Matcher I

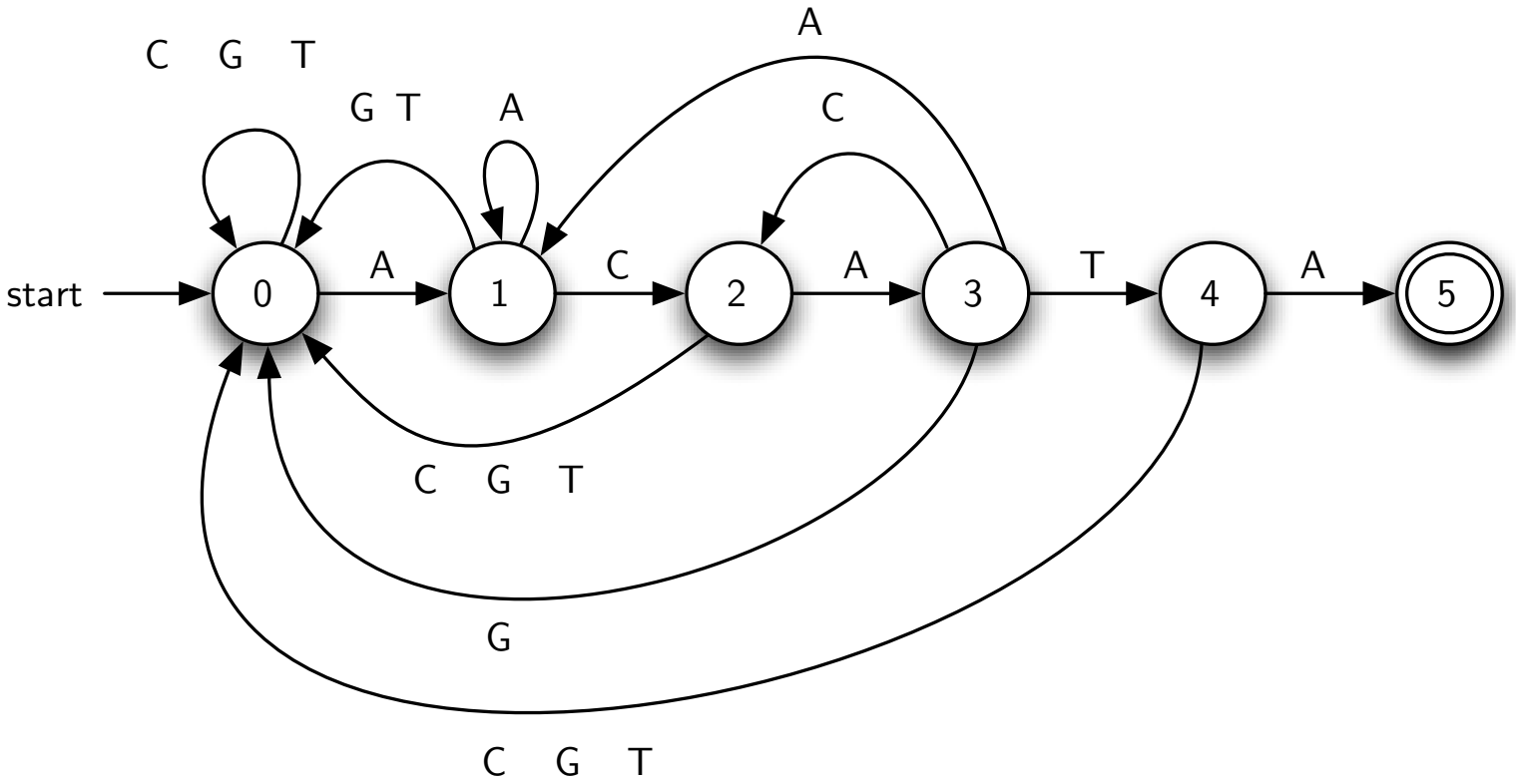
```
1  def simpleMatcher(pattern, text):
2      i=0
3      j=0
4      match = False
5      stop = False
6      while not match and not stop:
7          if text[i] == pattern[j]:
8              i=i+1
9              j=j+1
10         else:
11             i=i+1
12             j=0
13
14         if j == len(pattern):
15             match = True
16         else:
```

A Simple Pattern Matcher II

```
17                if i == len(text):
18                    stop = True
19
20        if match:
21            return i-j
22        else:
23            return -1
```

Outline

A Deterministic Finite Automaton

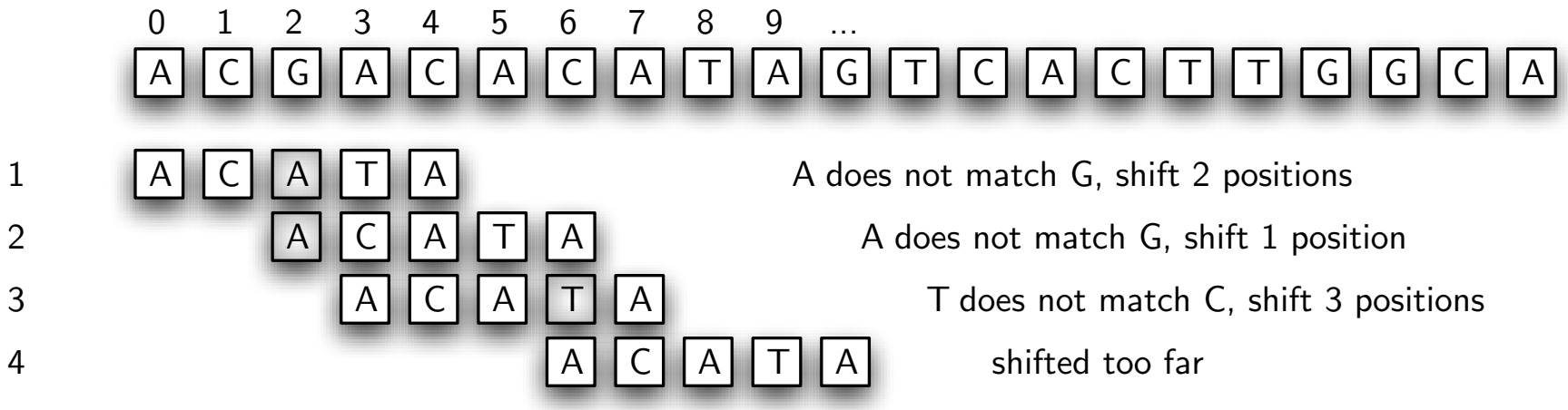


A Trace of the DFA Pattern-Matcher

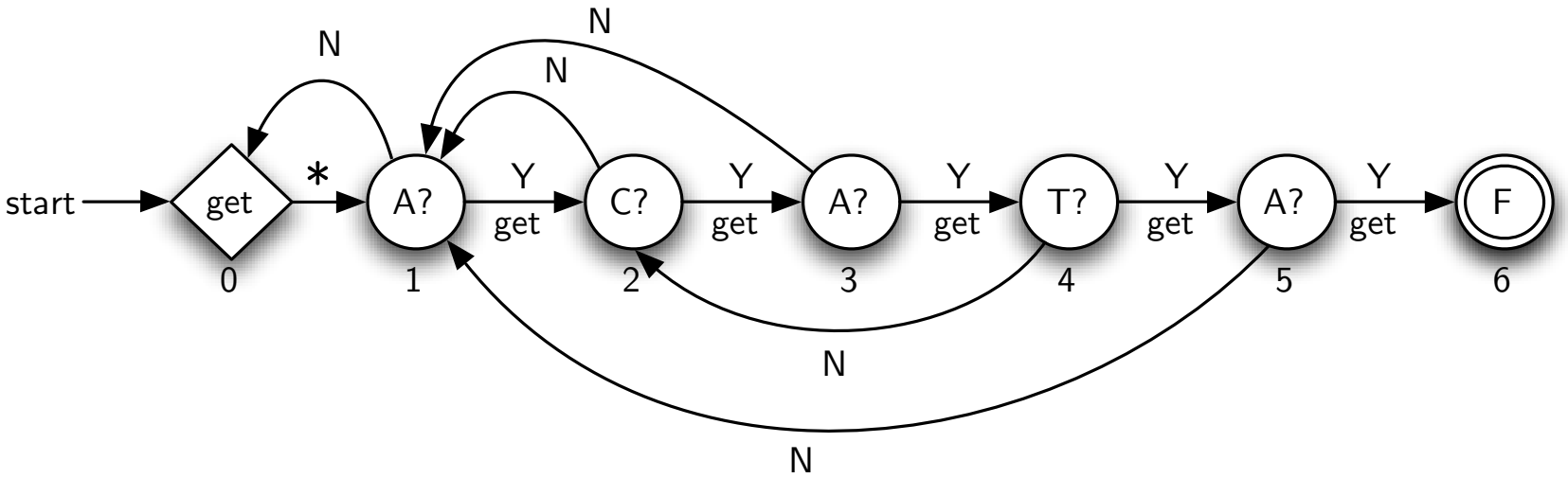
Step	Current State	Current Text Symbol	Next State
1	0	A	1
2	1	C	2
3	2	G	0
4	0	A	1
5	1	C	2
6	2	A	3
7	3	C	2
8	2	A	3
9	3	T	4
10	4	A	5 Þnal

Outline

Simple Pattern-Matcher with Longer Shifts



An Example KMP Graph



A Simple Pattern Matcher

```
1  def mismatchLinks(pattern):
2      augPattern = "0"+pattern
3      links = {}
4      links[1] = 0
5      for k in range(2,len(augPattern)):
6          s = links[k-1]
7          stop = False
8          while s>=1 and not stop:
9              if augPattern[s] == augPattern[k-1]:
10                 stop = True
11             else:
12                 s = links[s]
13         links[k] = s+1
14     return links
```

A Trace of the KMP Pattern-Matcher

Step	Current State	Current Text Symbol	Next State	
1	0	A	1	automatic transition
2	1	A	2	state 1 match, get next
3	2	C	3	state 2 match, get next
4	3	G	1	mismatch
5	1	G	0	mismatch
6	0	A	1	automatic transition
7	1	A	2	
8	2	C	3	
9	3	A	4	
10	4	C	2	mismatch
11	2	C	3	state 2 match
12	3	A	4	
13	4	T	5	
14	5	A	F	success