Object Oriented Patterns
A Historical Perspective

Dr. Charles R. Severance

www.cc4e.com

code.cc4e.com (sample code)

online.dr-chuck.com

Outline – Object Oriented

- Review from previous courses
- A historical perspective
- Using C to build an OO support
- Building a Python str() class in C
- Building a Python list() class in C
- Building a Python dict() class in C
- Reference: 6.5.1

OO Review – online.dr-chuck.com

- OO Python
  - Python for Everybody – Chapter 14
  - Django for Everybody
- OO JavaScript
  - Django for Everybody
  - Web Applications for Everybody

- The goal of this content is to explore how one would implement OO patterns using C

Definitions (Review)

- Class – a Template – cookie cutter
  - Attribute – Some data item that is to be contained in each instance of the class
  - Method – Some code (i.e. like a function) that operates within the context of an instance of the class
- Object – A particular instance of a class "stamped out" from the class when a new instance of the class is requested

Image CC-By 2.0

constructor

attributes

methods

attributes

```
import math

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def dump(self):
        print('Object point@%x x=%f y=%f' % (id(self),self.x,self.y))

    def origin(self):
        return math.sqrt(self.x*self.x+self.y*self.y)

pt = Point(4.0, 5.0)
Point.dump(pt)
print('Origin',pt.origin())
del(pt)
```

constructed

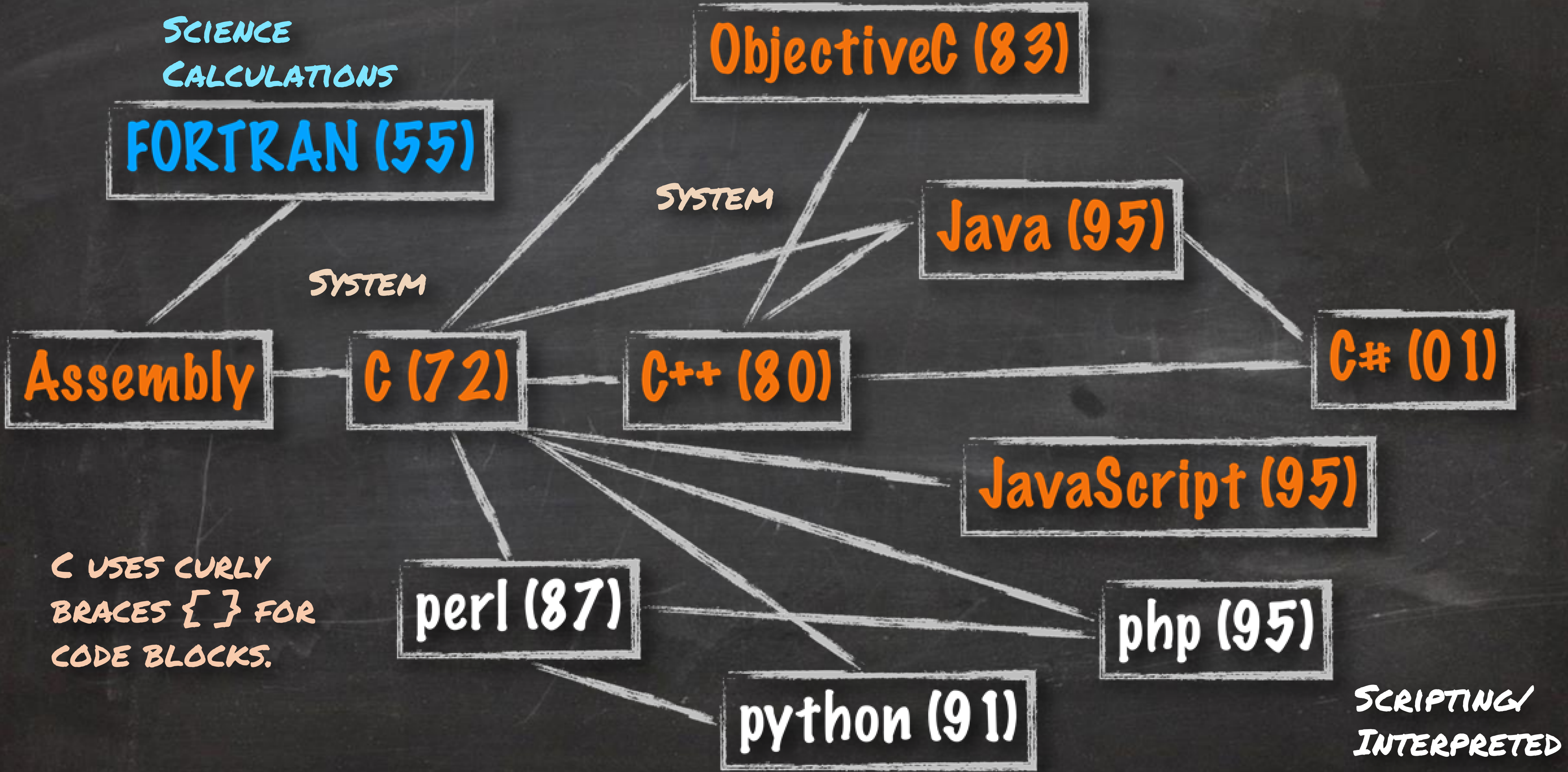
destructed

```
Object point@102ad1f10 x=4.000000 y=5.000000
Origin 6.4031242374328485
```

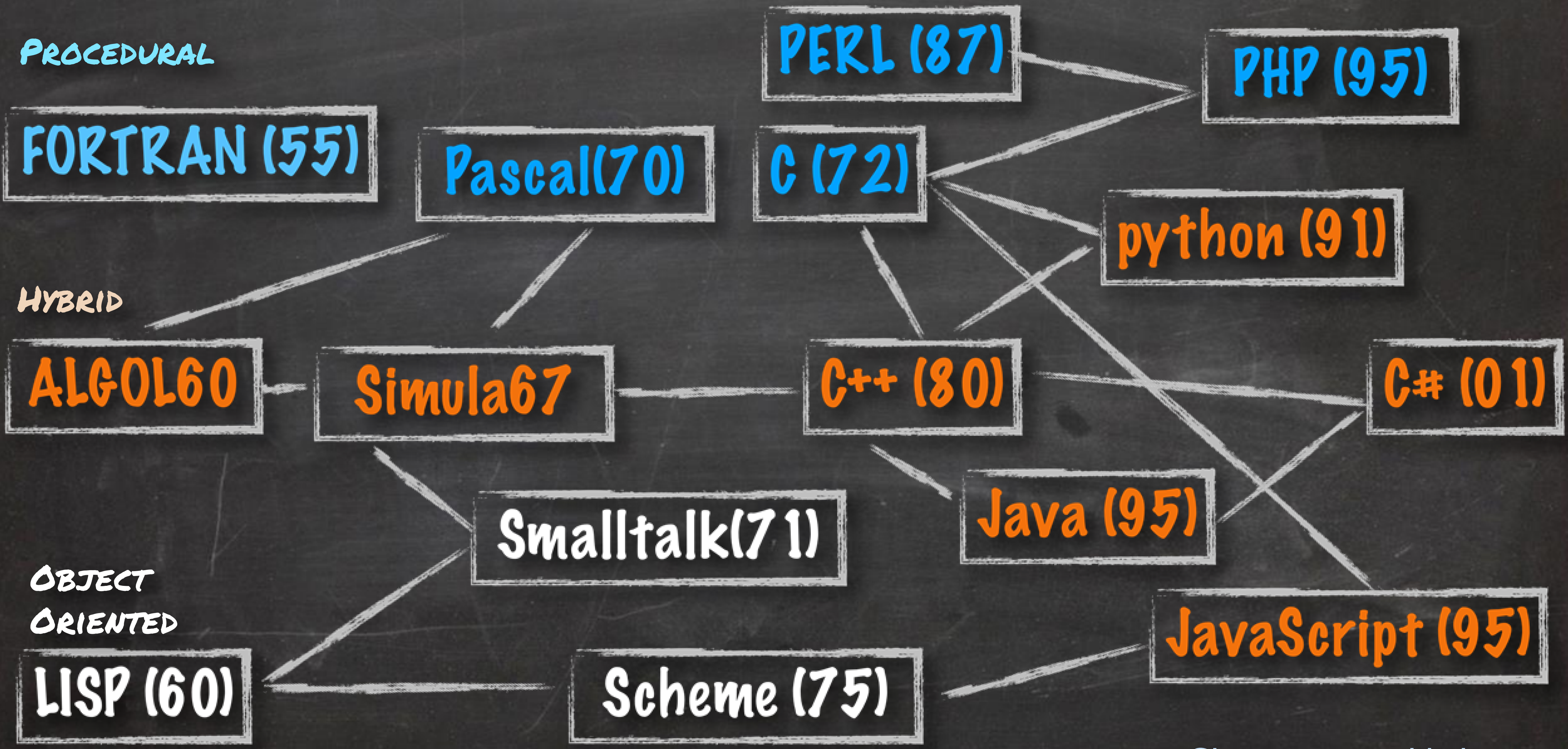
kr_03_01.py

Object Orientation

Across Time and Programming Languages



Wikipedia: History of programming languages



Classes were added to PHP in 2000

Wikipedia: Object-oriented programming

OO Implementations Over Time

- Python (1991)
- C++ (1980)
- Java (1995)
- JavaScript (1995)
- PHP (2000)
- C# (2001)

P.S. The more languages you know the better – Dr. Chuck

```
import math

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def dump(self):
        print('Object point@%x x=%f y=%f' %
           (id(self),self.x,self.y))

    def origin(self):
        return math.sqrt(self.x*self.x+self.y*self.y)

pt = Point(4.0, 5.0)
Point.dump(pt)
print('Origin',pt.origin())
del(pt)
```

```
Object point@102ad1f10 x=4.000000 y=5.000000
Origin 6.4031242374328485
```

Python uses the concept of "self" to refer to the variables stored in the current instance. The constructor is by convention "__init__".

```
#include <iostream>
#include <math.h>

class Point {
  public:
    double x, y;

    Point(double xc, double yc)  {
        x = xc;
        y = yc;
    };

    void dump() {
        printf("Object point x=%f y=%f\n", x, y);
    }

    double origin() {
        return sqrt(x*x+y*y);
    }
};

int main() {
    Point pt(4.0, 5.0);
    pt.dump();
    printf("Origin: %f\n", pt.origin());
}
```

```
Object point x=4.000000 y=5.000000
Origin: 6.40312
```

C++ was initially implemented as a pre-processing pass that did textual transformations and then passed the code into the C compiler. There is no concept of "self" or "this".

1980 **cc_03_01.cpp**

```
import java.lang.Math;

public class Point {
   double x,y;

   Point(double x, double y) {
       this.x = x;
       this.y = y;
   }

   void dump() {
       System.out.printf("%s x=%f y=%f\n",
           this, this.x, this.y);
   }

   double origin() {
        return Math.sqrt(this.x*this.x+this.y*this.y);
   }
}
------------------------------------------------------
public class cc_03_01 {

    public static void main(String[] args)
    {
        Point pt = new Point(4.0, 5.0);
        pt.dump();
        System.out.printf("Origin %f\n",pt.origin());
    }
}
```

```
Point@7cef4e59 x=4.000000 y=5.000000
Origin 6.403124
```

The keyword "this" is a reserved word in the Java language. It is used to refer to the current instance of the class. The constructor is an un-typed function with the same name as the class. When you print "this" out, it resolves to the name of the class and an indication of which instance we are currently in. The toString() method produces the string.

Point.java
cc_03_01.java

```
function Point(x, y) {
    this.x = x;
    this.y = y;
    this.party = function () {
        this.x = this.x + 1;
        console.log("So far "+this.x);
    }

    this.dump = function () {
        console.log("Object point x=%f y=%f",
              this.x, this.y);
    }

    this.origin = function () {
        return Math.sqrt(this.x*this.x+this.y*this.y);
    }
}

pt = new Point(4.0, 5.0);

pt.dump();
console.log("Origin %f",pt.origin());
```

```
Object point x=4 y=5
Origin 6.4031242374328485
```

JavaScript takes the "this" and "new" keywords from Java but not much else. There is no "class" keyword. Formatted The way methods are defined take inspiration from functional languages like Scheme and use the "first-class function" pattern. Output tracks C very closely. The class definition itself overloads the "function" keyword.

TypeScript is an "evolved" version of JavaScript and adds "class" and "this" keywords allowing for a more traditional Java-like syntax for declaring objects.

1995 **cc_03_01.js**

```
class Point {
    private $x, $y;

    public function __construct($x, $y) {
        $this->x = $x;
        $this->y = $y;
    }

    public function dump() {
        printf("Object point x=%f y=%f\n",
               $this->x, $this->y);
    }

    public function origin() {
        return sqrt($this->x*$this->x+
                    $this->y*$this->y);
    }
}

$pt = new Point(4.0, 5.0);
$pt->dump();
printf("Origin %f\n",$pt->origin());
```

```
Object point x=4.000000 y=5.000000
Origin 6.403124
```

PHP introduced Object Orientation in PHP 4 in 2000 so it could borrow patterns like "this", "class", and "new" from languages like Java. Because PHP started with Perl 😟 , variables start with '$'. Also, PHP uses the '.' operator for concatenation for the "object lookup" operator, PHP had to use "->" which is inspired by the syntax C uses with a pointer to a structure.

2000 **cc_03_01.php**

```
using System;

public class Point
{
   public double x,y;

   public Point(double cx, double cy) {
       x = cx;
       y = cy;
   }

   public void dump() {
       Console.WriteLine("Point object x={0} y={1}", x, y);
   }

   public double origin() {
        return Math.Sqrt(x*x+y*y);
   }
}

class TestPoint{
    public static void Main(string[] args)
     {
         Point pt = new Point(4.0, 5.0);
         pt.dump();
         Console.WriteLine("Origin {0}", pt.origin());
     }
}
```

```
Point object x=4 y=5
Origin 6.40312423743285
```

Microsoft's C# was relatively recent and so it could look at the other approaches and pick what they saw as the best parts. C# took most of its inspiration from Java.

2001

cc_03_01.cs

Using C to Build OO Support

How was Python's OO layered on top of C?

```
import math

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def dump(self):
        print('Object point@%x x=%f y=%f' %
            (id(self),self.x,self.y))

    def origin(self):
        return
math.sqrt(self.x*self.x+self.y*self.y)

pt = Point(4.0, 5.0)
Point.dump(pt)
print('Origin',pt.origin())
del(pt)
```

```
Object point@102ad1f10 x=4.000000 y=5.000000
Origin 6.4031242374328485
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

struct Point {
    double x;
    double y;

    void (*del)(const struct Point* self);
    void (*dump)(const struct Point* self);
    double (*origin)(const struct Point* self);
};

void point_dump(const struct Point* self)
{
    printf("Object point@%p x=%f y=%f\n",
            self, self->x, self->y);
}

void point_del(const struct Point* self) {
  free((void *)self);
}

double point_origin(const struct Point* self) {
    return sqrt(self->x*self->x + self->y*self->y);
}
```

```c
struct Point * point_new(double x, double y) {
    struct Point *p = malloc(sizeof(*p));
    p->x = x;
    p->y = y;
    p->dump = &point_dump;
    p->origin = &point_origin;
    p->del = &point_del;
    return p;
}

int main(void)
{
    struct Point * pt = point_new(4.0,5.0);
    pt->dump(pt);
    printf("Origin %f\n", pt->origin(pt));
    pt->del(pt);
}
```

```
Object point@0x600003e94030 x=4.000000 y=5.000000
Origin 6.403124
```

1975

cc_03_01.c

Building a Python str() Class

```python
x = str()
x = x + 'H'
print(x)
x = x + 'ello world'
print(x)
x = 'A completely new string'
print("String = ", x)
print("Length = ", len(x))
```

```
H
Hello world
String =  A completely new string
Length =  23
```

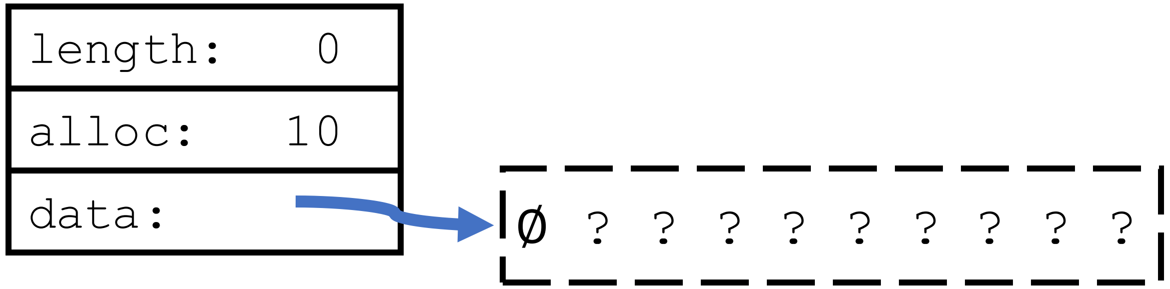
```c
int main(void)
{
    printf("Testing pystr class\n");
    struct pystr * x = pystr_new();
    pystr_dump(x);

    pystr_append(x, 'H');
    pystr_dump(x);

    pystr_appends(x, "ello world");
    pystr_dump(x);

    pystr_assign(x, "A completely new string");
    printf("String = %s\n", pystr_str(x));
    printf("Length = %d\n", pystr_len(x));
    pystr_del(x);
}
```

```
Testing pystr class
Object length=0 alloc=10 data=
Object length=1 alloc=10 data=H
Object length=11 alloc=20 data=Hello world
String = A completely new string
Length = 23
```

cc_03_02.py

String Structure and Constructor

```
struct pystr
{
    int length;
    int alloc; /* The length of *data */
    char *data;
};

/* x = str() */
struct pystr * pystr_new() {
    struct pystr *p = malloc(sizeof(*p));
    p->length = 0;
    p->alloc = 10;
    p->data = malloc(10);
    p->data[0] = '\0';
    return p;
}

...

int main(void)
{
   struct pystr * x = pystr_new();
   ...
}
```



cc_03_02.c

Some Methods

```
struct pystr
{
    int length;
    char *data;
    int alloc; /* The length of *data */
};

/* Constructor - x = str() */
struct pystr * pystr_new() {
    struct pystr *p = malloc(sizeof(*p));
    p->length = 0;
    p->alloc = 10;
    p->data = malloc(10);
    p->data[0] = '\0';
    return p;
}

/* Destructor - del(x) */
void pystr_del(const struct pystr* self) {
    free((void *)self->data); /* free string first */
    free((void *)self);
}

void pystr_dump(const struct pystr* self)
{
    printf("Pystr length=%d alloc=%d data=%s\n",
            self->length, self->alloc, self->data);
}
```

```
int pystr_len(const struct pystr* self)
{
    return self->length;
}

char *pystr_str(const struct pystr* self)
{
    return self->data;
}
```

cc_03_02.c

Data Methods

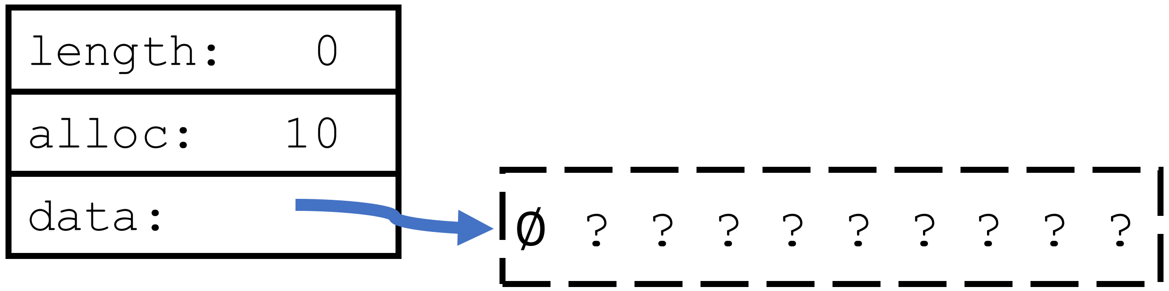
- We have three methods to put data into our structure
- You can build pystr_append() and then use it to make the code in pystr_appends() and pystr_assign() very simple.
- Objects like to reuse their own methods because not repeating yourself code is more reliable

```
/* x = x + 'h'; */

void pystr_append(struct pystr* self, char ch) {
    /* Need about 10 lines of code here ☺ */
}

/* x = x + "hello"; */

void pystr_appends(struct pystr* self, char *str) {
    /* Need one line of code here ☺ */
}

/* x = "hello"; */

void pystr_assign(struct pystr* self, char *str) {
    /* Need three lines of code here ☺ */
}

int main() {
    struct pystr * x = pystr_new();
    pystr_append(x, 'H');
    pystr_dump(x);
    pystr_appends(x, "ello world");
    pystr_dump(x);
    pystr_assign(x, "A completely new string");
}
```

pystr_append()

- Recall that the constructor allocates a 10 character array and sets alloc to 10
- Length of the actual string is zero because there is no data
- We can append 9 characters with the data array that is initially allocated

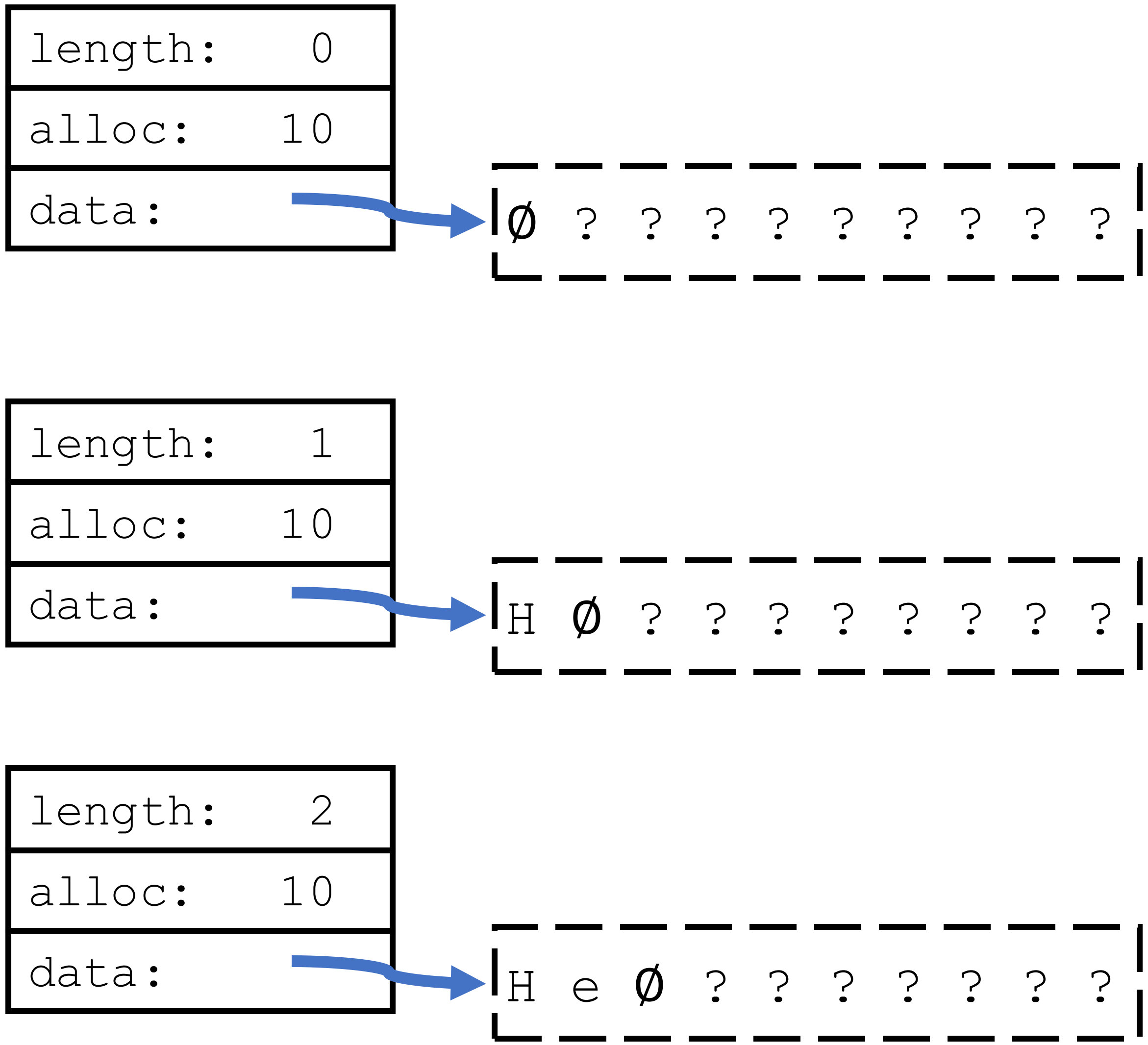
```
struct pystr
{
    int length;
    char *data;
    int alloc; /* the length of *data */
};

struct pystr * pystr_new() {
    struct pystr *p = malloc(sizeof(*p));
    p->length = 0;
    p->alloc = 10;
    p->data = malloc(10);
    p->data[0] = '\0';
    return p;
}
```



cc_03_02.c

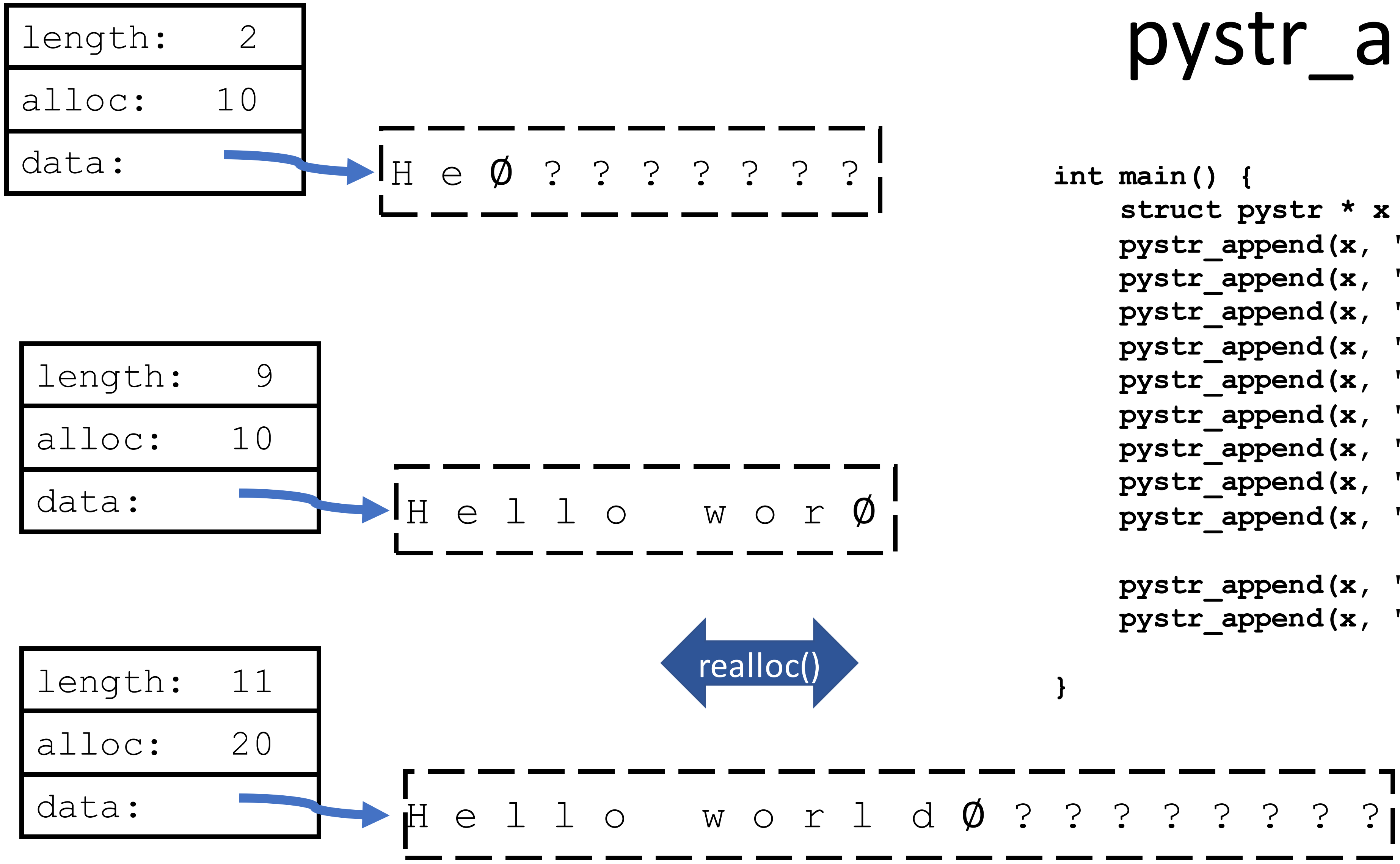
pystr_append()

```
struct pystr
{
    int length;
    char *data;
    int alloc; /* the length of *data */
};

int main() {
    struct pystr * x = pystr_new();
    pystr_append(x, 'H');
    pystr_append(x, 'e');
}
```



cc_03_02.c

pystr_append()



```
int main() {
    struct pystr * x = pystr_new();
    pystr_append(x, 'H');
    pystr_append(x, 'e');
    pystr_append(x, 'l');
    pystr_append(x, 'l');
    pystr_append(x, 'o');
    pystr_append(x, ' ');
    pystr_append(x, 'w');
    pystr_append(x, 'o');
    pystr_append(x, 'r');

    pystr_append(x, 'l');
    pystr_append(x, 'd');

}
```

realloc()

- We can extend the size of a dynamically allocated area by calling realloc() with the current pointer to the area and the needed size
- We may get a different address from realloc()
- The data will be copied and the old data will be freed

```
struct pystr * pystr_new() {
    struct pystr *p = malloc(sizeof(*p));
    p->length = 0;
    p->alloc = 10;
    p->data = malloc(10);
    p->data[0] = '\0';
    return p;
}

void pystr_append(struct pystr* self, char ch) {
    /* If we don't have space for 1 character plus
       termination, allocate 10 more */

    if ( self->length >= (self->alloc - 2) ) {
        self->alloc = self->alloc + 10;
        self->data = (char *)
            realloc(self->data, self->alloc);
    }
    ...
}
```

Tutorial: C dynamic memory allocation

Using our "class"

```
x = str()
x = x + 'H'
print(x)
x = x + 'ello world'
print(x)
x = 'A completely new string'
print("String = ", x)
print("Length = ", len(x))
```

```
H
Hello world
String =  A completely new string
Length =  23
```

cc_03_02.py

```
int main(void)
{
    struct pystr * x = pystr_new();
    pystr_dump(x);

    pystr_append(x, 'H');
    pystr_dump(x);

    pystr_appends(x, "ello world");
    pystr_dump(x);

    pystr_assign(x, "A completely new string");
    printf("String = %s\n", pystr_str(x));
    printf("Length = %d\n", pystr_len(x));
    pystr_del(x);
}
```

```
Testing pystr class
Object length=0 alloc=10 data=
Object length=1 alloc=10 data=H
Object length=11 alloc=20 data=Hello world
String = A completely new string
Length = 23
```

cc_03_02.c

Building a Python list() Class

```
lst = list();
lst.append("Hello world");
print(lst)
lst.append("Catch phrase");
print(lst)
lst.append("Brian");
print(lst)
print("Length =", len(lst));

print("Brian?", lst.index("Brian"));

if "Bob" in lst:
    print("Bob?", lst.index("Bob"));
else:
    print("Bob? 404");
```

```
int main(void)
{
    struct pylist * lst = pylist_new();
    pylist_append(lst, "Hello world");
    pylist_print(lst);
    pylist_append(lst, "Catch phrase");
    pylist_print(lst);
    pylist_append(lst, "Brian");
    pylist_print(lst);
    printf("Length = %d\n", pylist_len(lst));
    printf("Brian? %d\n", pylist_index(lst, "Brian"));
    printf("Bob? %d\n", pylist_index(lst, "Bob"));
    pylist_del(lst);
}
```

```
['Hello world']
['Hello world', 'Catch phrase']
['Hello world', 'Catch phrase', 'Brian']
Length = 3
Brian? 2
Bob? 404
```

```
['Hello world']
['Hello world', 'Catch phrase']
['Hello world', 'Catch phrase', 'Brian']
Length = 3
Brian? 2
Bob? -1
```

cc_03_03.c

cc_03_03.py

Basic stuff

```c
struct lnode {
    char *text;
    struct lnode *next;
};

struct pylist {
  struct lnode *head;
  struct lnode *tail;
  int count;
};

/* Constructor - lst = list() */
struct pylist * pylist_new() {
    struct pylist *p = malloc(sizeof(*p));
    p->head = NULL;
    p->tail = NULL;
    p->count = 0;
    return p;
}
```

```c
/* Destructor - del(lst) */
void pylist_del(struct pylist* self) {
    struct lnode *cur, *next;
    cur = self->head;
    while(cur) {
        free(cur->text);
        next = cur->next;
        free(cur);
        cur = next;
    }
    free((void *)self);
}
```

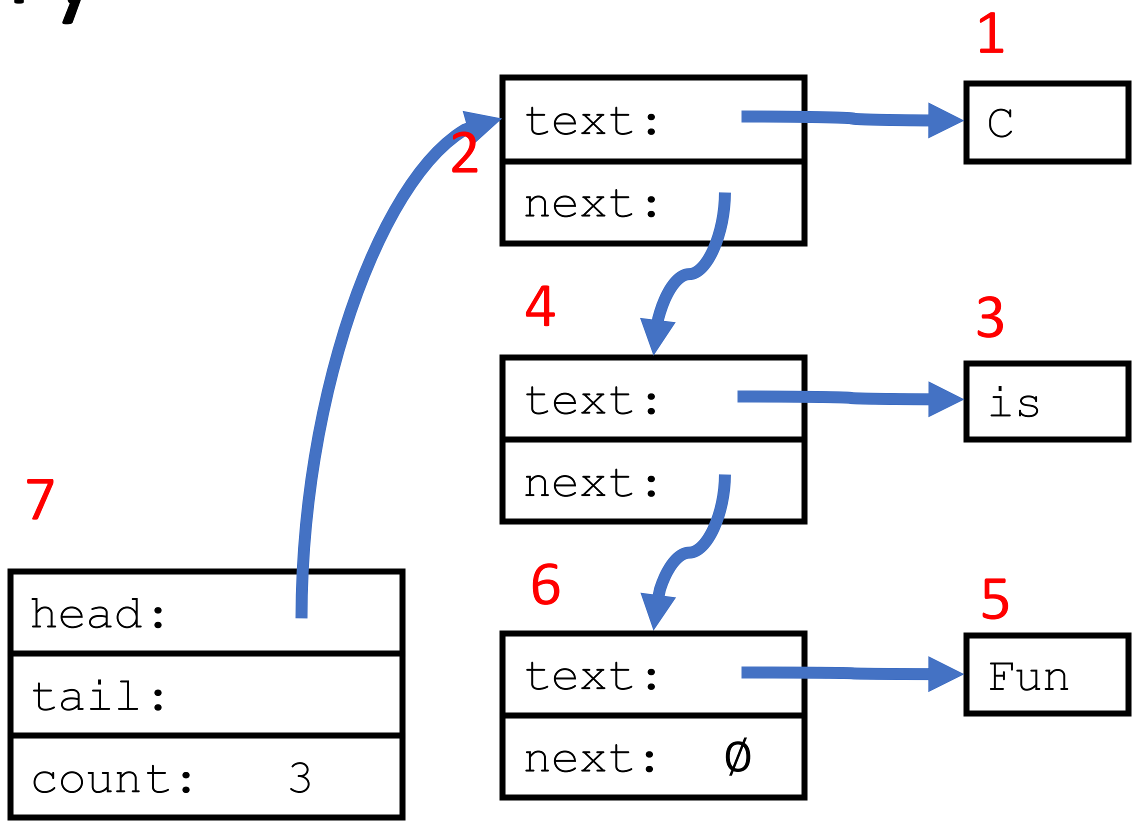
Important: Free structures from the "leaves" inwards – free the actual structure *last*

cc_03_03.c

Freeing Dynamic Memory

```
/* Destructor - del(lst) */
void pylist_del(struct pylist* self) {
    struct lnode *cur, *next;
    cur = self->head;
    while(cur) {
        free(cur->text); 1, 3, 5
        next = cur->next;
        free(cur); 2, 4, 6
        cur = next;
    }
    free((void *)self); 7
}
```

Even the order of next->cur->next and the free(cur) matters because once free() is called, you should assume the data is gone or zeroed out



cc_03_03.c

Printing a list

```
/* print(lst) */
void pylist_print(struct pylist* self)
{
    /* About 10 lines of code
       The output should match Python's
       list output

       ['Hello world', 'Catch phrase']

       Use printf cleverly, *not* string
       concatenation since this is C, not Python.
    */
}
```

```
['Hello world']
['Hello world', 'Catch phrase']
['Hello world', 'Catch phrase', 'Brian']
Length = 3
Brian? 2
Bob? -1
```

Remember that unlike Python's print() which always includes a newline, C's printf() only adds a newline (\n) if you include it in the format string. So it is easy to print "long lines" with no line breaks with for loops in C.

cc_03_03.c

More methods for you to build

```
/* len(lst) */
int pylist_len(const struct pylist* self)
{
    /* One line of code */
}

/* lst.append("Hello world") */
void pylist_append(struct pylist* self, char *str) {
    /* Review: Chapter 6 lectures and assignments */
}

/* lst.index("Hello world") - if not found -1 */
int pylist_index(struct pylist* self, char *str)
{
    /* Seven or so lines of code */
}
```

cc_03_03.c

Using our class

- This is almost line for line identical to the Python version of this code
- The output is also pretty much identical

```
int main(void)
{
    struct pylist * lst = pylist_new();
    pylist_append(lst, "Hello world");
    pylist_print(lst);
    pylist_append(lst, "Catch phrase");
    pylist_print(lst);
    pylist_append(lst, "Brian");
    pylist_print(lst);
    printf("Length = %d\n", pylist_len(lst));
    printf("Brian? %d\n", pylist_index(lst, "Brian"));
    printf("Bob? %d\n", pylist_index(lst, "Bob"));
    pylist_del(lst);
}
```

```
['Hello world']
['Hello world', 'Catch phrase']
['Hello world', 'Catch phrase', 'Brian']
Length = 3
Brian? 2
Bob? -1
```

cc_03_03.py

cc_03_03.c

Building a Python dict() Class

Python Dictionary

```
dct = dict();
dct["z"] = "Catch phrase"
print(dct);
dct["z"] = "W"
print(dct);
dct["y"] = "B"
dct["c"] = "C"
dct["a"] = "D"
print(dct);
print("Length =", len(dct));
print("z=", dct.get("z", 404))
print("x=", dct.get("x", 404))
print("\nDump")
for key in dct:
    print(key+"="+dct[key])
```

```
{'z': 'Catch phrase'}
{'z': 'W'}
{'z': 'W', 'y': 'B', 'c': 'C', 'a': 'D'}
Length = 4
z= W
x= 404

Dump
z=W
y=B
c=C
a=D
```

cc_03_04.py

C Dictionary

```c
int main(void)
{
    struct pydict * dct = pydict_new();
    pydict_put(dct, "z", "Catch phrase");
    pydict_print(dct);
    pydict_put(dct, "z", "W");
    pydict_print(dct);
    pydict_put(dct, "y", "B");
    pydict_put(dct, "c", "C");
    pydict_put(dct, "a", "D");
    pydict_print(dct);
    printf("Length =%d\n",pydict_len(dct));

    printf("z=%s\n", pydict_get(dct, "z"));
    printf("x=%s\n", pydict_get(dct, "x"));

    printf("\nDump\n");
    for(struct dnode * cur = dct->head; cur != NULL ; cur = cur->next ) {
        printf("%s=%s\n", cur->key, cur->value);
    }

    pydict_del(dct);
}
```

```
{'z': 'Catch phrase'}
{'z': 'W'}
{'z': 'W', 'y': 'B', 'c': 'C', 'a': 'D'}
Length =4
z=W
x=(null)

Dump
z=W
y=B
c=C
a=D
```

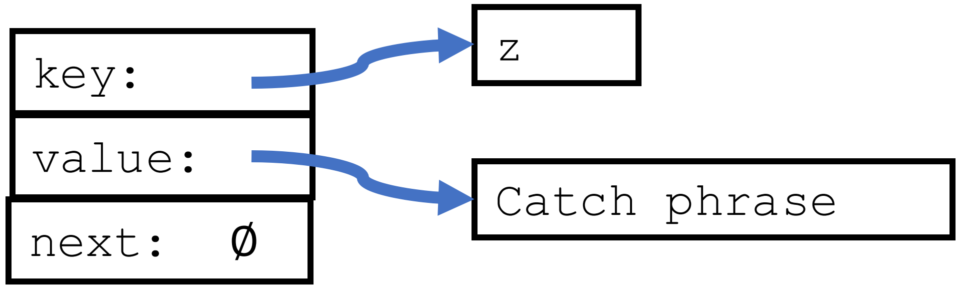
cc_03_04.c

Basic stuff

```
struct dnode {
    char *key;
    char *value;
    struct dnode *next;
};

struct pydict {
  struct dnode *head;
  struct dnode *tail;
  int count;
};

/* Constructor - dct = dict() */
struct pydict * pydict_new() {
    struct pydict *p = malloc(sizeof(*p));
    p->head = NULL;
    p->tail = NULL;
    p->count = 0;
    return p;
}
```

cc_03_04.c

```
/* Destructor - del(dct) */
void pydict_del(struct pydict* self) {
    struct dnode *cur, *next;
    cur = self->head;
    while(cur) {
        free(cur->key);
        free(cur->value);
        next = cur->next;
        free(cur);
        cur = next;
    }
    free((void *)self);
}
```



```
int main(void)
{
    struct pydict * dct = pydict_new();
    pydict_put(dct, "z", "Catch phrase");
}
```

Methods for you to build

```
/* len(dct) */
int pydict_len(const struct pydict* self)
{
    /* One line of code */
}

/* print(lst) */
/* {'z': 'W', 'y': 'B', 'c': 'C', 'a': 'D'} */
void pydict_print(struct pydict* self)
{
    /* Some code */
}
```

cc_03_04.c

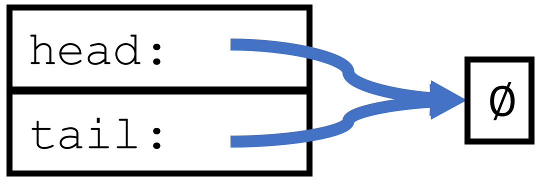
A Reusable Method

```
struct dnode* pydict_find(struct pydict* self, char *key)
{
     /* Six lines of code */
}

/* x.get(key) - Returns NULL if not found */
char* pydict_get(struct pydict* self, char *key)
{
    struct dnode *entry = pydict_find(self, key);
    /* Two lines of code */
}

/* x[key] = value; Insert or replace the value associated with a key */
struct pydict* pydict_put(struct pydict* self, char *key, char *value) {
    struct dnode *old = pydict_find(self, key);
    if ( old != NULL ) {
        /* Some code */
    } else {
       /* Some code */
    }
}
```

cc_03_04.c

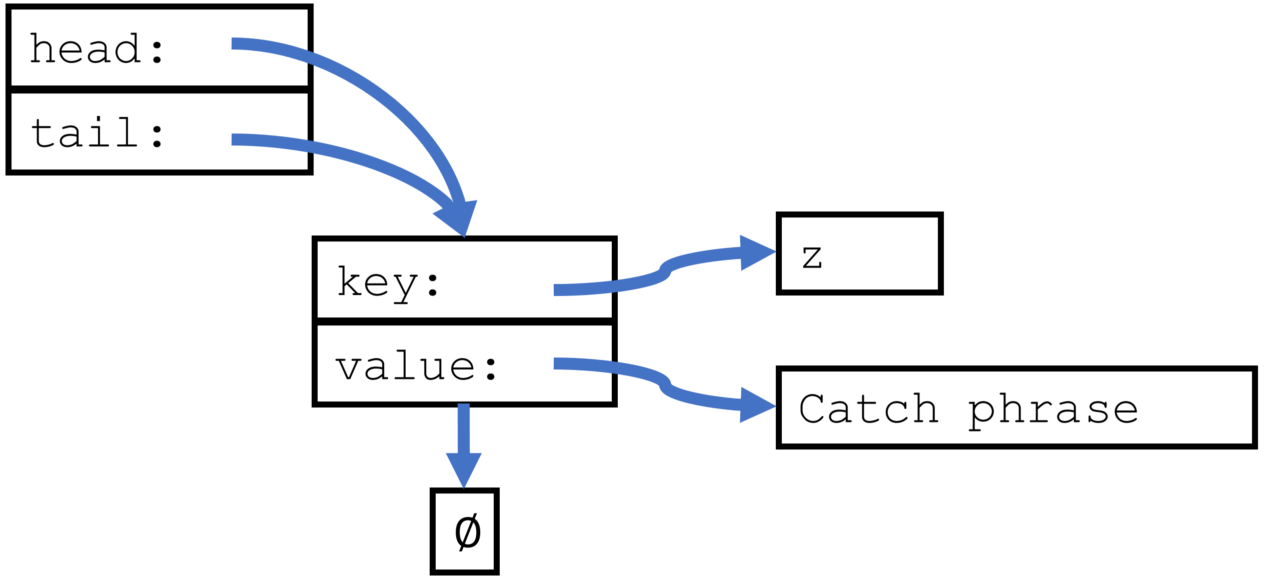
Building a Dictionary

```
struct dnode {
    char *key;
    char *value;
    struct dnode *next;
};

struct pydict {
  struct dnode *head;
  struct dnode *tail;
  int count;
};

int main(void)
{
    struct pydict * dct = pydict_new();
    pydict_put(dct, "z", "Catch phrase");
    pydict_put(dct, "z", "W");
    pydict_put(dct, "y", "B");
    pydict_put(dct, "c", "C");
    pydict_put(dct, "a", "D");
    ...
}
```



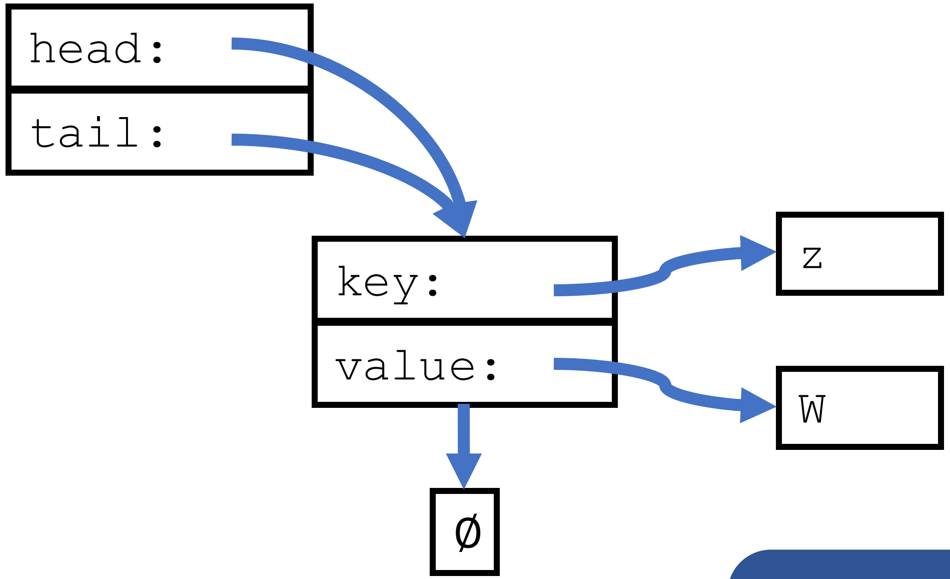
cc_03_04.c

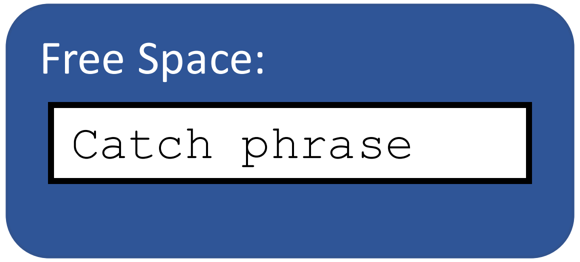
```
struct dnode {
    char *key;
    char *value;
    struct dnode *next;
};

struct pydict {
  struct dnode *head;
  struct dnode *tail;
  int count;
};

int main(void)
{
    struct pydict * dct = pydict_new();
    pydict_put(dct, "z", "Catch phrase");
    pydict_put(dct, "z", "W");
    pydict_put(dct, "y", "B");
    pydict_put(dct, "c", "C");
    pydict_put(dct, "a", "D");
    ...
}
```



cc_03_04.c

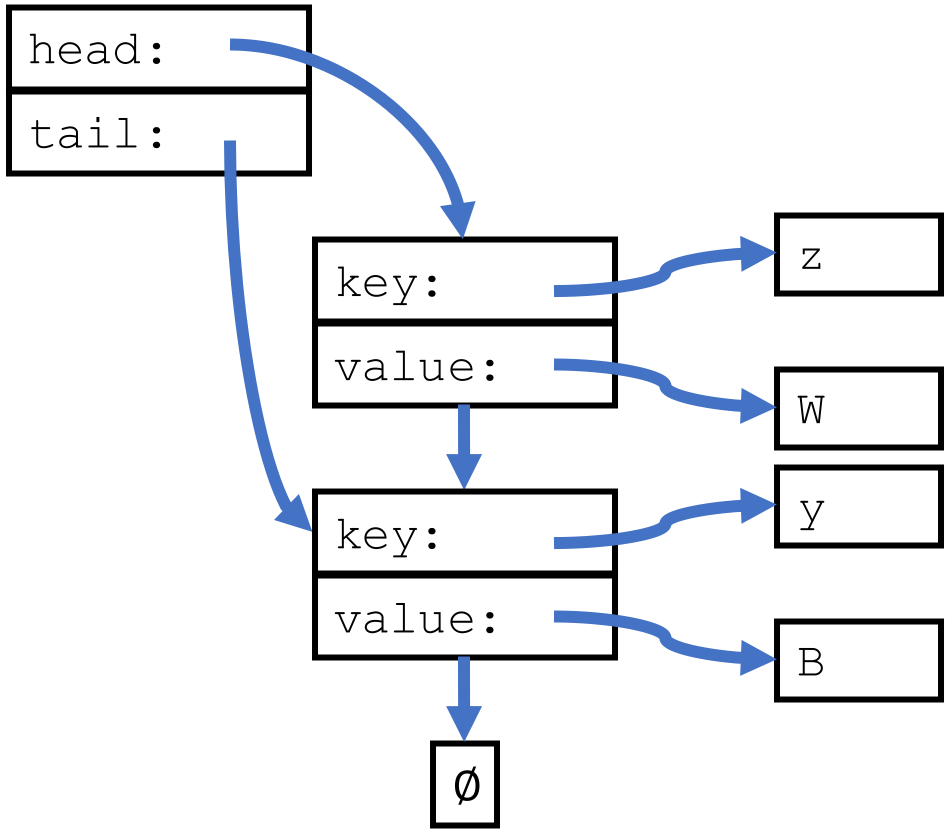
```
struct dnode {
    char *key;
    char *value;
    struct dnode *next;
};

struct pydict {
  struct dnode *head;
  struct dnode *tail;
  int count;
};

int main(void)
{
    struct pydict * dct = pydict_new();
    pydict_put(dct, "z", "Catch phrase");
    pydict_put(dct, "z", "W");
    pydict_put(dct, "y", "B");
    pydict_put(dct, "c", "C");
    pydict_put(dct, "a", "D");
    ...
}
```





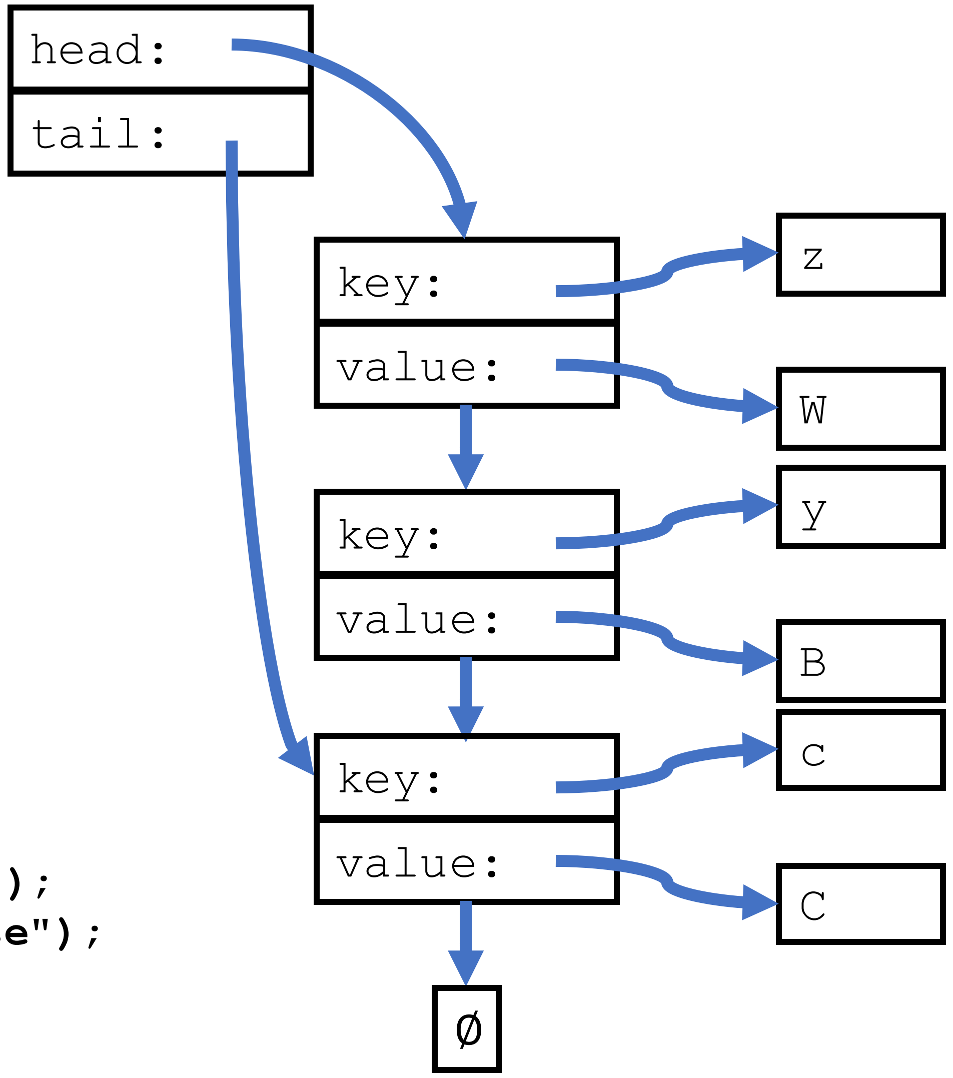
Make sure to free the data in value that was the "Catch phrase" string before allocating the "W" string and setting value to point to the newly allocated string.

```
struct dnode {
    char *key;
    char *value;
    struct dnode *next;
};

struct pydict {
  struct dnode *head;
  struct dnode *tail;
  int count;
};

int main(void)
{
    struct pydict * dct = pydict_new();
    pydict_put(dct, "z", "Catch phrase");
    pydict_put(dct, "z", "W");
    pydict_put(dct, "y", "B");
    pydict_put(dct, "c", "C");
    pydict_put(dct, "a", "D");
    ...
}
```



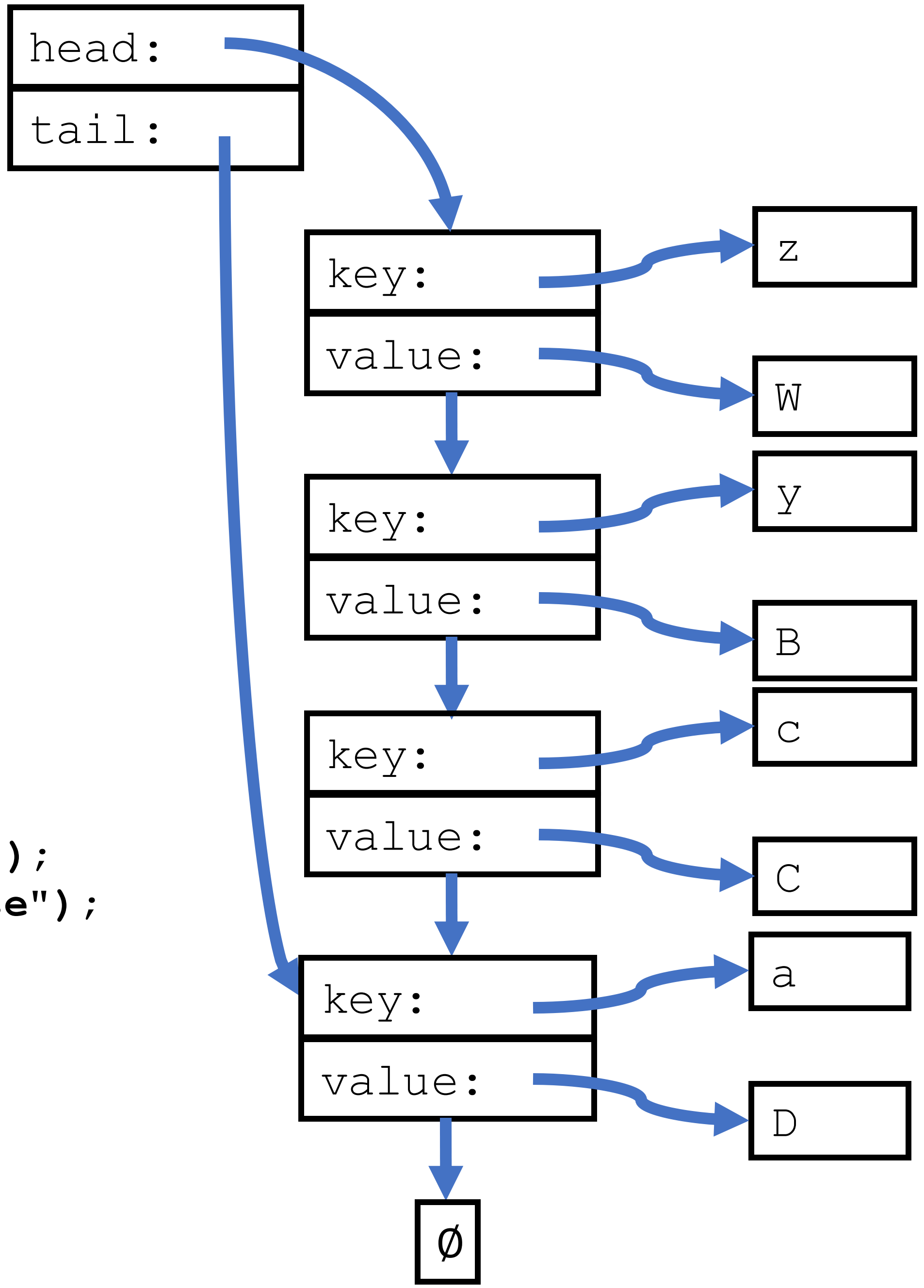
cc_03_04.c

```
struct dnode {
    char *key;
    char *value;
    struct dnode *next;
};

struct pydict {
  struct dnode *head;
  struct dnode *tail;
  int count;
};

int main(void)
{
    struct pydict * dct = pydict_new();
    pydict_put(dct, "z", "Catch phrase");
    pydict_put(dct, "z", "W");
    pydict_put(dct, "y", "B");
    pydict_put(dct, "c", "C");
    pydict_put(dct, "a", "D");
    ...
}
```



cc_03_04.c

```
struct dnode {
    char *key;
    char *value;
    struct dnode *next;
};

struct pydict {
  struct dnode *head;
  struct dnode *tail;
  int count;
};

int main(void)
{
    struct pydict * dct = pydict_new();
    pydict_put(dct, "z", "Catch phrase");
    pydict_put(dct, "z", "W");
    pydict_put(dct, "y", "B");
    pydict_put(dct, "c", "C");
    pydict_put(dct, "a", "D");
    ...
}
```



cc_03_04.c

Python OrderedDict() versus dict()

- Before Python 3.7 the dict() class did not remember the order of items as they were inserted
    - The Pre-3.7 Python used internal implementation used hashing to give fast performance for key lookup and insert
- Since we have implemented dictionary functionality on top of a linked list, our code behaves more like an OrderedDict()
- Our insert is quick but our key lookup using get() is slow as it might need to scan the entire list
- We will address this performance issue in an upcoming lecture with an awesome data structure that combines linked lists *and* hashing ☺

Summary

- OO Review from previous courses
- OO – A historical perspective
- Using C to build an OO support
- Building a Python str() class in C
- Building a Python list() class in C
- Building a Python dict() class in C (OrderedDict actually)

Acknowledgements / Contributions

Initial Development: Charles Severance, University of Michigan School of Information

Insert new Contributors and Translators here including names and dates

Continue new Contributors and Translators here

Building a Better Object

Dr. Charles R. Severance

www.cc4e.com

code.cc4e.com (sample code)

online.dr-chuck.com

Problems with our "pydict" class

- The previous dict implementation was just a linked list with a key – we need to build multiple implementations for different performance requirements
- We need to include our methods in the struct instead of using prefix-style function naming conventions
- We need a better abstraction for looping that does not require "peeking" at the class-internal attributes

Object Oriented Principles

- Encapsulation – Bundling code and data together
  - Wikipedia: Encapsulation (computer programming)
- Abstraction – Separate interface from implementation
  - Wikipedia: Abstraction (computer science)
- Inheritance – Creating new classes by extending existing classes (DRY)
  - Wikipedia: Inheritance (object-oriented programming)
- Polymorphism - Later ☺
  - Wikipedia: Polymorphism (computer science)

Encapsulation by Naming Convention

- So far to keep things simple we associate a method for a class with a simple naming convention
- It seems simple enough but is a bad idea in practice.
  - Python strings are objects that follow the principle of encapsulation
  - PHP strings are a type plus an associated / prefixed library

- Before I show you the next slide, I need to emphasize that I love many many things about PHP – but there are some annoyances ☺

Encapsulation by Naming Convention (is bad)

```
# Python
x = "A string with old in it"

print(x.find('with'))

y = x.replace('old', 'new')

print(y)

print(len(y))
```

```
# PHP
$x = "A string with old in it";

print(strpos($x,'with')."\n");

$y = str_replace(???, ???, ???);

print($y."\n");

print(???????($y)."\n");
```

Encapsulation by Naming Convention (is bad)

```
# Python
x = "A string with old in it"

print(x.find('with'))

y = x.replace('old', 'new')

print(y)

print(len(y))
```

```
# PHP
$x = "A string with old in it";

print(strpos($x,'with')."\n");

$y = str_replace('old', 'new',
$x);

print($y."\n");

print(strlen($y)."\n");
```

Putting Methods in the Structure

```
int main(void)
{
    struct pydict * dct = pydict_new();

    pydict_put(dct, "z", "Catch phrase");
    pydict_put(dct, "z", "W");

    printf("Length =%d\n",pydict_len(dct));

    printf("z=%s\n", pydict_get(dct, "z"));

    pydict_del(dct);
}
```

```
int main(void)
{
    struct pydict * dct = pydict_new();

    dct->put(dct, "z", "Catch phrase");
    dct->put(dct, "z", "W");

    printf("Length =%d\n",dct->len(dct));

    printf("z=%s\n", dct->get(dct, "z"));

    dct->del(dct);
}
```

Because we are emulating OO patterns in C, moving new() into the structure is possible but adds a layer of complexity at compile *and* run time. Similarly we need to have 'dct' twice in method calls so methods have access to the instance in the methods. Languages like C++ and Python solve this by adjusting their compiler.

Access Control / "Leaky" Abstractions

- When the class is designed such that the calling code needs to look at **data attributes** *inside* the class – we call this a "leaky abstraction"
- The implementation details like the choice of an internal variable name within the class are "leaking" out into the calling code.
- When calling code depends on this internal implementation names and approaches, it means that we cannot change the code in the class without breaking calling code
- We need to define a "contract" between the class and its calling code that we agree won't change. We call this contract an "interface".

```c
int main(void)
{
    struct pydict * dct = pydict_new();
    pydict_put(dct, "z", "Catch phrase");
    pydict_print(dct);
    pydict_put(dct, "z", "W");
    pydict_print(dct);
    pydict_put(dct, "y", "B");
    pydict_put(dct, "c", "C");
    pydict_put(dct, "a", "D");
    pydict_print(dct);
    printf("Length =%d\n",pydict_len(dct));

    printf("z=%s\n", pydict_get(dct, "z"));
    printf("x=%s\n", pydict_get(dct, "x"));

    printf("\nDump\n");

    for(struct dnode * cur = dct->head; cur != NULL ; cur = cur->next ) {
        printf("%s=%s\n", cur->key, cur->value);
    }

    pydict_del(dct);
}
```

```c
struct dnode {
    char *key;
    char *value;
    struct dnode *next;
};

struct pydict {
  struct dnode *head;
  struct dnode *tail;
  int count;
};
```

cc_03_04.c

Controlling Access to Object Attributes

- As we build a class, we decide which elements are part of our contract with our calling code
- In real OO languages, in the class definition, we mark attributes and methods with our intended access level
    - Accessible by the calling code – "public"
    - Accessible only within the class – "private"
    - Accessible within the class and other *internal* classes – "protected"

```
int main(void)
{
    struct pydict * dct = pydict_new();
    pydict_put(dct, "z", "Catch phrase");
    pydict_print(dct);
    pydict_put(dct, "z", "W");
    pydict_print(dct);
    pydict_put(dct, "y", "B");
    pydict_put(dct, "c", "C");
    pydict_put(dct, "a", "D");
    pydict_print(dct);
    printf("Length =%d\n",pydict_len(dct));

    printf("z=%s\n", pydict_get(dct, "z"));
    printf("x=%s\n", pydict_get(dct, "x"));

    printf("\nDump\n");

    for(struct dnode * cur = dct->head; cur != NULL ; cur = cur->next ) {
        printf("%s=%s\n", cur->key, cur->value);
    }

    pydict_del(dct);
}
```

```
struct dnode {
      /* protected */ char *key;
      /* protected */ char *value;
      /* protected */ struct dnode *next;
};

struct pydict {
   /* private */ struct dnode *head;
   /* private */ struct dnode *tail;
   /* private */ int count;

   /* public */ pydict_len();
   /* public */ pydict_len();
   /* public */ pydict_get();
   /* public */ pydict_del();

};
```

Abstraction Boundary

Early C++ access control syntax was implemented by a pre-processor.

cc_03_04.c

Access Control in Java

```
public class Point {
   private double x,y;

   Point(double x, double y) {
       this.x = x;
       this.y = y;
   }

   public void dump() {
       System.out.printf("%s x=%f y=%f\n", this, this.x, this.y);
   }
}
```

Access Control in C++

```
class Point {
  private:
    double x, y;

  public:
    Point(double xc, double yc)  {
        x = xc;
        y = yc;
    };

    void dump() {
        printf("Object point x=%f y=%f\n", x, y);
    }
};
```

Access Control in Python

```
class Point:
    __x = 0.0
    __y = 0.0

    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def dump(self):
        print('Object point@%x x=%f y=%f' % (id(self),self.__x,self.__y))
```

What in a Map?

For dictionary like structures

Modern Maps – Key / Value Pairs

- "Map" is a common term we use to describe abstract key/value collections
  - C++ map
  - Python dictionary
  - Java Map
  - PHP Arrays
- Iterator Pattern as an abstraction for looping across multiple implementations

```
d = dict()

print("Testing dict class\n");
d["z"] = 8
d["z"] = 1
d["y"] = 9
d["b"] = 3
d["a"] = 4
print(d);

print("z=%d" % (d.get("z", 42), ));
print("x=%d" %  (d.get("x", 42), ));

items = iter(d.items())
print(type(items));

entry = next(items, False)
while (entry) :
    print(entry)
    entry = next(items, False)
```

```
Testing dict class

{'z': 1, 'y': 9, 'b': 3, 'a': 4}
z=1
x=42
<class 'dict_itemiterator'>
('z', 1)
('y', 9)
('b', 3)
('a', 4)
```

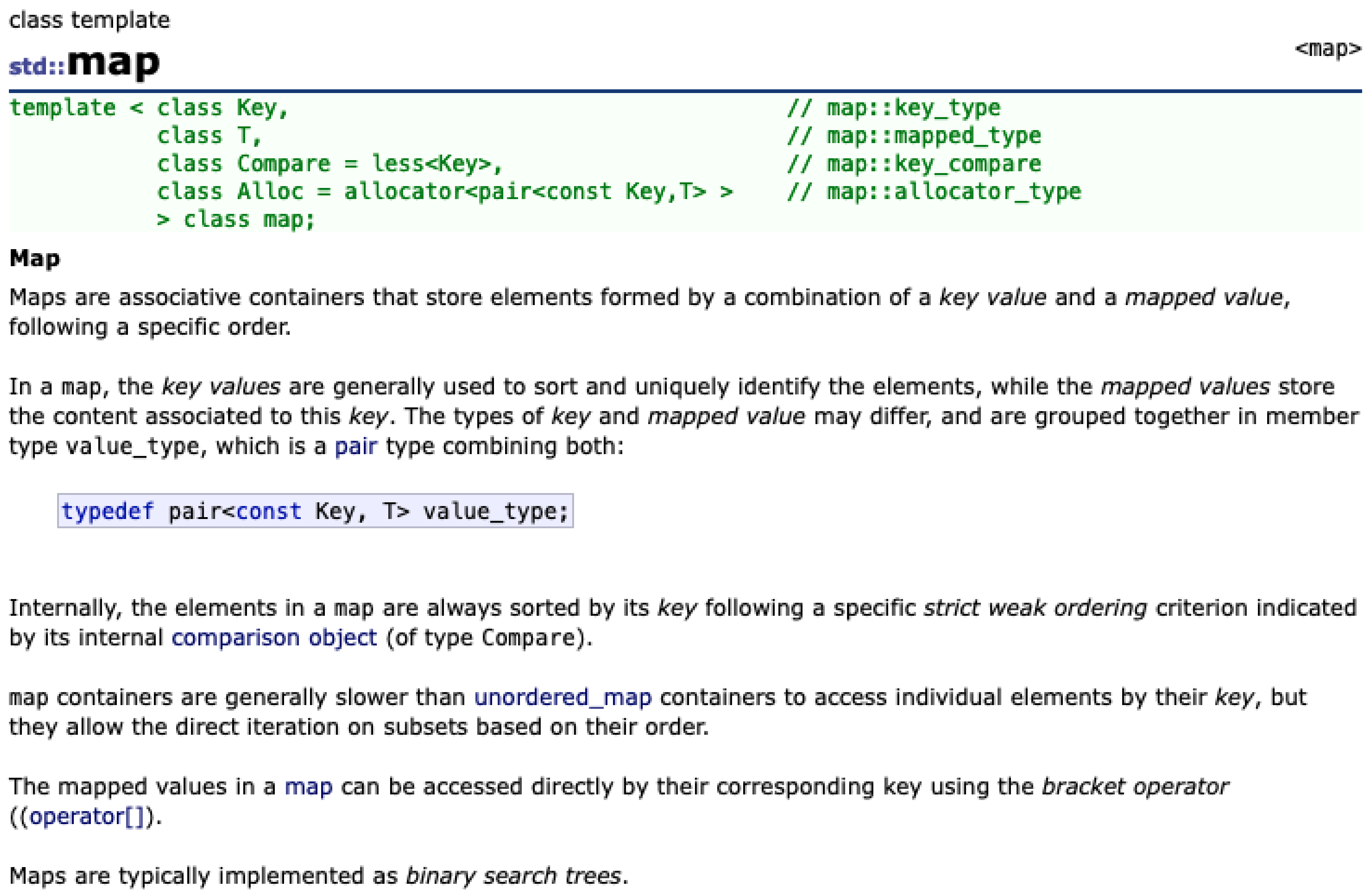
cc_04_03.py

```
$a = array();

print("Testing php arrays\n");
$a["z"] = 8;
$a["z"] = 1;
$a["y"] = 9;
$a["b"] = 3;
$a["a"] = 4;
print_r($a);

printf("z=%d\n", $a["z"] ?? 42);
printf("x=%d\n", $a["x"] ?? 42);

printf("\nIterate\n");
foreach( $a as $k => $v ) {
    printf(" %s=%d\n", $k, $v);
}
```

```
Testing dict class
Array
(
    [z] => 1
    [y] => 9
    [b] => 3
    [a] => 4
)
z=1
x=42

Iterate
 z=1
 y=9
 b=3
 a=4
```

cc_04_03.php

C++



class template

<map>

std::map

```
template < class Key,                                     // map::key_type
           class T,                                       // map::mapped_type
           class Compare = less<Key>,                     // map::key_compare
           class Alloc = allocator<pair<const Key,T> >    // map::allocator_type
           > class map;
```

Map

Maps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order.

In a map, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values* store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type value_type, which is a pair type combining both:

```
typedef pair<const Key, T> value_type;
```

Internally, the elements in a map are always sorted by its *key* following a specific *strict weak ordering* criterion indicated by its internal comparison object (of type Compare).

map containers are generally slower than unordered_map containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

The mapped values in a map can be accessed directly by their corresponding key using the *bracket operator* ((operator[]).

Maps are typically implemented as *binary search trees*.

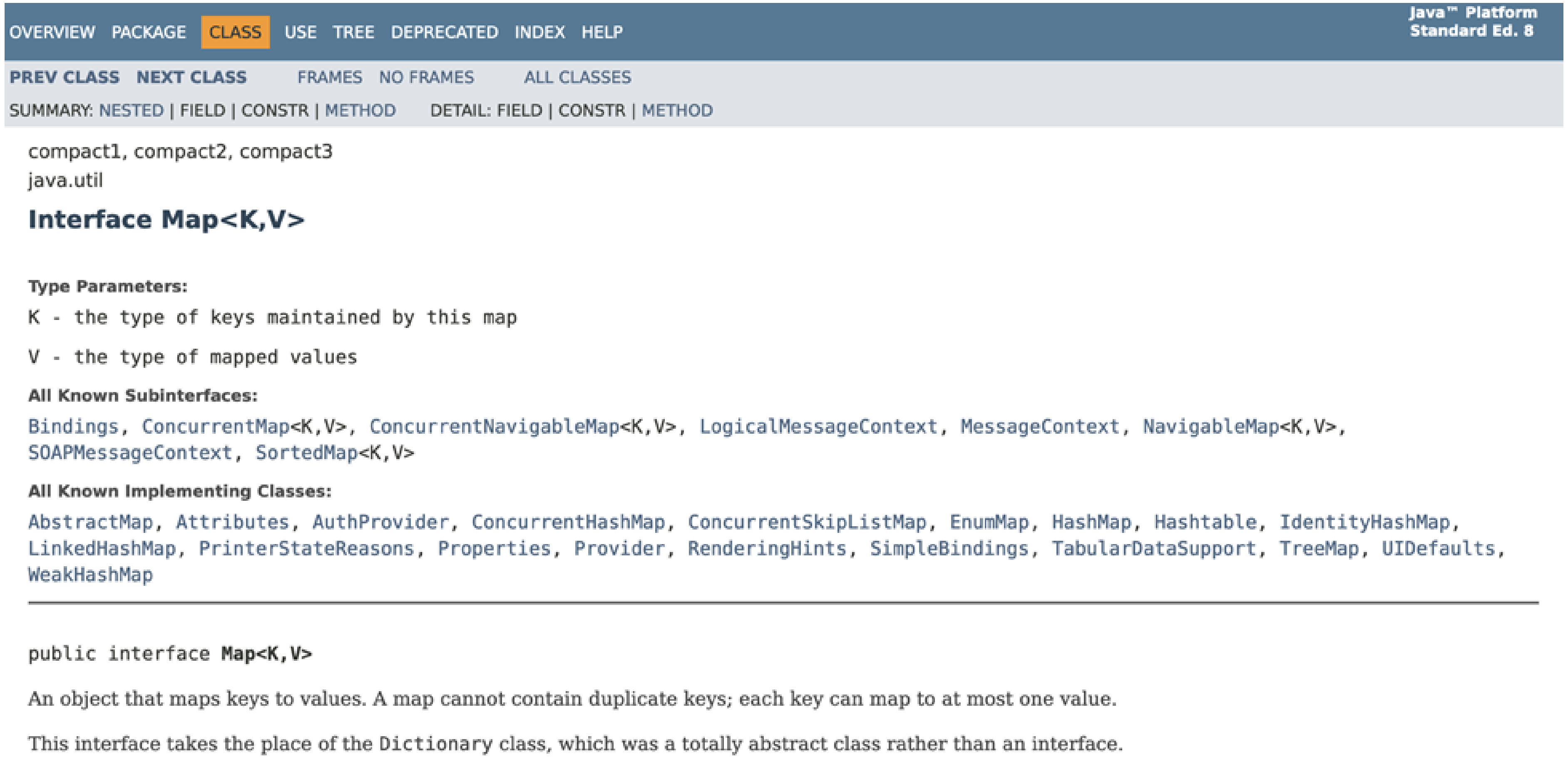
Reference: map

```
#include <iostream>
#include <map>
using namespace std;

int main() {
    map<string, int> mp;

    printf("Testing map class\n");
    mp["z"] = 8;
    mp["z"] = 1;
    mp["y"] = 2;
    mp["b"] = 3;
    mp["a"] = 4;

    printf("z=%d\n", (mp.count("z") ? mp["z"] : 42));
    printf("x=%d\n", (mp.count("x") ? mp["x"] : 42));

    printf("\nIterate\n");
    for (auto cur = mp.begin(); cur != mp.end(); ++cur) {
        printf(" %s=%d\n", cur->first.c_str(), cur->second);
    }
}
```

```
Testing map class
z=1
x=42

Iterate
 a=4
 b=3
 y=2
 z=1
```

cc_04_03.cpp

Java Map<String, Integer>


OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
Java™ Platform Standard Ed. 8
PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3
java.util

Interface Map<K,V>

Type Parameters:

`K - the type of keys maintained by this map`

`V - the type of mapped values`

All Known Subinterfaces:

`Bindings, ConcurrentMap<K,V>, ConcurrentNavigableMap<K,V>, LogicalMessageContext, MessageContext, NavigableMap<K,V>, SOAPMessageContext, SortedMap<K,V>`

All Known Implementing Classes:

`AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap`

```
public interface Map<K,V>
```

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the `Dictionary` class, which was a totally abstract class rather than an interface.

Reference: Java map

```
import java.util.Map;
import java.util.TreeMap;

public class cc_04_03 {

    public static void main(String[] args)
    {
        Map<String, Integer> map = new TreeMap<String, Integer> ();

        System.out.printf("Testing Map class\n");
        map.put("z", 8);
        map.put("z", 1);
        map.put("y", 2);
        map.put("b", 3);
        map.put("a", 4);
        System.out.println(map);

        System.out.println("z="+map.getOrDefault("z", 42));
        System.out.println("x="+map.getOrDefault("x", 42));

        System.out.printf("\nIterate\n");
        for (Map.Entry<String,Integer> entry : map.entrySet()) {
                System.out.println("Key = " + entry.getKey() + ", Value = " + entry.getValue());
        }

    }
}
```

```
Testing Map class
{a=4, b=3, y=2, z=1}
z=1
x=42

Iterate
Key = a, Value = 4
Key = b, Value = 3
Key = y, Value = 2
Key = z, Value = 1
```

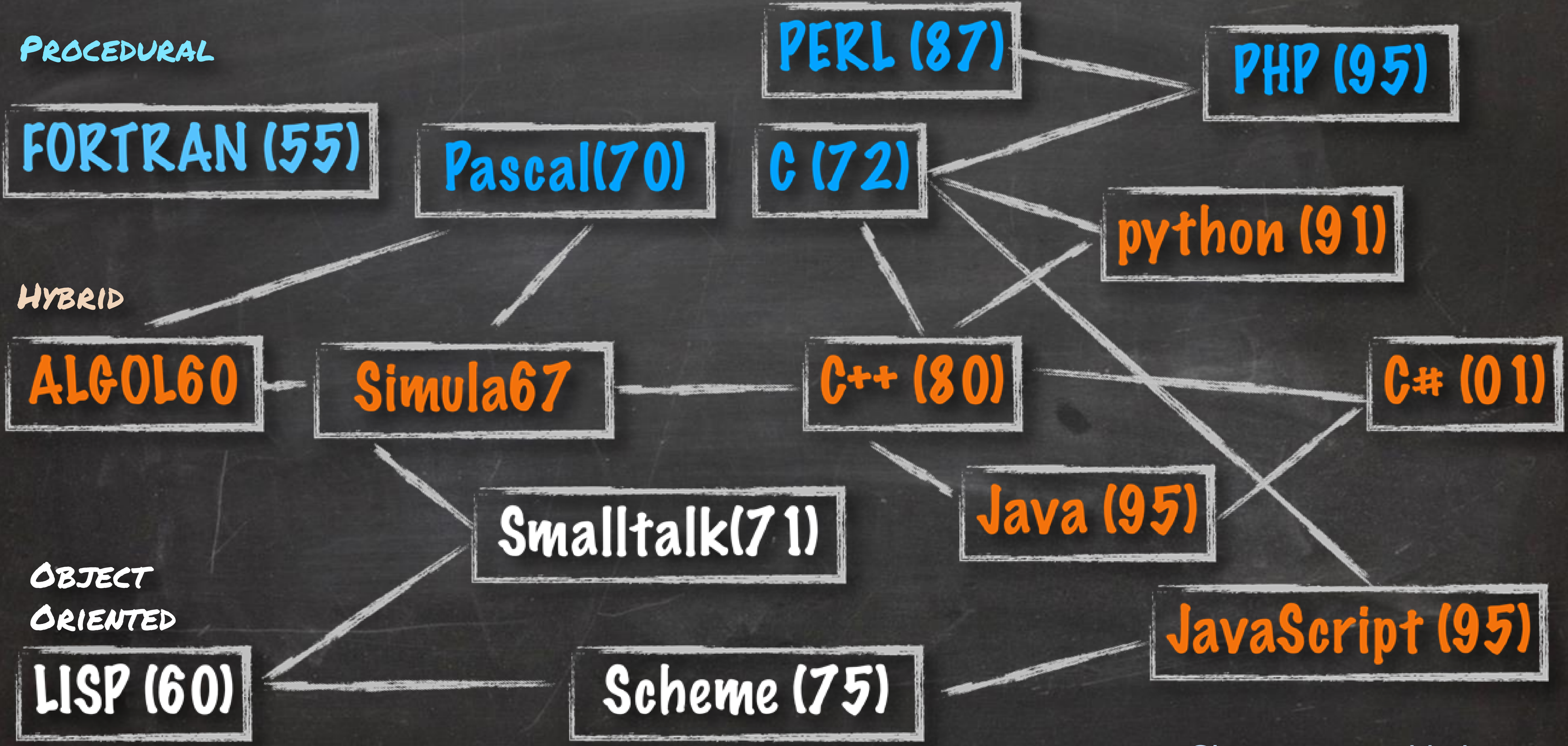
cc_04_03.java

```
d = dict()

print("Testing dict class\n");
d["z"] = 8
d["z"] = 1
d["y"] = 9
d["b"] = 3
d["a"] = 4
print(d);

print("z=%d" % (d.get("z", 42), ));
print("x=%d" %  (d.get("x", 42), ));

items = iter(d.items())
print(type(items));

print("Iterate");
entry = next(items, False)
while (entry) :
    print(entry)
    entry = next(items, False)
```

```
Testing dict class

{'z': 1, 'y': 9, 'b': 3, 'a': 4}
z=1
x=42
<class 'dict_itemiterator'>
Iterate
('z', 1)
('y', 9)
('b', 3)
('a', 4)
```

cc_04_03.py

A Minute of C++



Classes were added to PHP in 2000

Wikipedia: Object-oriented programming

Three Syntaxes

```
// C++ 1980 – Pre-process + compile
map<string, int> mp;

mp["z"] = 8;
```

```
# Python 1991 - Interpreted
d = dict()

d["z"] = 8
```

```
// Java 1995 – Compiled
Map<String, Integer> map =
  new TreeMap<String, Integer> ();

map.put("z", 8);
```

Java chose not to extend the language syntax to use [] and instead used get() and put() methods. C++ has a "pretty unique" approach that makes the succinct / natural syntax possible. The C++ syntax influenced the Python and most other syntaxes except for Java.

The C++ OO approach is somewhat unique

- C++ allows a class developer to create classes that can be used as if they new native types like integer and floating point
- Operator Overloading

```
#include <iostream>

class TenInt {
  private:
    int values[10];
  public:
    int & operator [](const int & index) {
        printf("- Returning reference to %d\n", index);
        return values[index];
    }
};

int main() {
    TenInt ten;
    ten[1] = 40;
    printf("printf ten[1] contains %d\n", ten[1]);

    ten[5] = ten[1] + 2;
    printf("Done assigning ten[5]\n");
    printf("printf ten[5] contains %d\n", ten[5]);
}
```

cc_04_04.cpp

```
#include <iostream>

class TenInt {
 private:
    int values[10];
 public:
    int & operator [](const int & index) {
        printf("-- Returning reference to %d\n", index);
        return values[index];
    }
};

int main() {
    TenInt ten;
    ten[1] = 40;
    printf("printf ten[1] contains %d\n", ten[1]);

    ten[5] = ten[1] + 2;
    printf("Done assigning ten[5]\n");
    printf("printf ten[5] contains %d\n", ten[5]);
}
```

```
-- Returning reference to 1
-- Returning reference to 1
printf ten[1] contains 40
-- Returning reference to 1
-- Returning reference to 5
Done assigning ten[5]
-- Returning reference to 5
printf ten[5] contains 42
```

Java did not want to support call by reference, and even more did not support returning a reference from within a function. C++ figured that a good programmer would use these features wisely.

cc_04_04.cpp

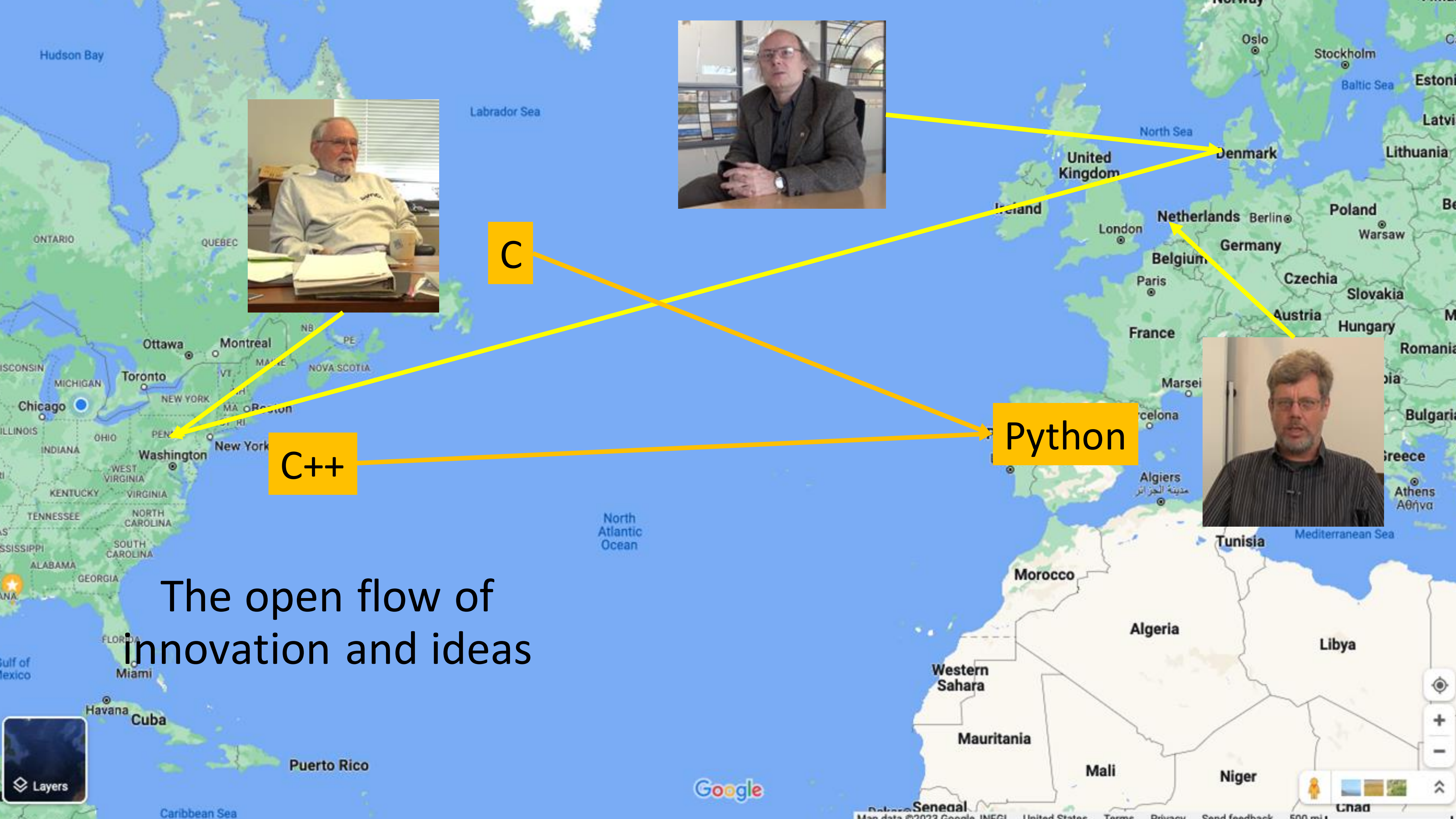
The C++ Influence in Python goes very deep

- The Python approach to providing this syntax to us was to extend the language to transform the bracket syntax into function calls to internal class methods

```
x = dict()
x[1] = 40
print('print x[1]', x.__getitem__(1))

# x[5] = x[1] + 2
x.__setitem__(5, x.__getitem__(1) + 2)
print(x)
```

```
print x[1] 40
{1: 40, 5: 42}
```

cc_04_04.py



The open flow of innovation and ideas

Python has operator overloading!

```
#include <iostream>

class TenInt {
  private:
    int values[10];
  public:
    int & operator [](const int & index) {
        printf("-- Returning reference to %d\n", index);
        return values[index];
    }
};

int main() {
    TenInt ten;
    ten[1] = 40;
    printf("printf ten[1] contains %d\n", ten[1]);

    ten[5] = ten[1] + 2;
    printf("Done assigning ten[5]\n");
    printf("printf ten[5] contains %d\n", ten[5]);
}
```

cc_04_04.cpp

```
class TenInt:
    __values = dict()

    def __setitem__(self, index, value) :
        self.__values[index] = value

    def __getitem__(self, index) :
        return self.__values[index]

ten = TenInt()
ten[1] = 40;
print("print ten[1] contains", ten[1]);

ten[5] = ten[1] + 2;
print("Done assigning ten[5]");
print("print ten[5] contains", ten[5]);
```

```
print ten[1] contains 40
Done assigning ten[5]
print ten[5] contains 42
```

cc_04_05.py

Implementing Encapsulation

Moving methods into our C-based Map class (refactor your existing code)

```
int main(void)
{
    struct MapEntry *cur;
    struct Map * map = Map_new();

    printf("Testing Map class\n");
    map->put(map, "z", 8);
    map->put(map, "z", 1);
    map->put(map, "y", 2);
    map->put(map, "b", 3);
    map->put(map, "a", 4);
    map->dump(map);

    printf("z=%d\n", map->get(map, "z", 42));
    printf("x=%d\n", map->get(map, "x", 42));

    /* No iterator (for now) */

    map->del(map);
}
```

```
Testing Map class
Object Map count=4
  z=1
  y=2
  b=3
  a=4
z=1
x=42
```

MapEntry Structure

- This is the structure that will make up the nodes in the list.
- The key is a character string – the actual data will be saved in a newly allocated space.
- The value is an int and will be allocated right in the node.

```
struct MapEntry {
    char *key;  /* public */
    int value;  /* public */
    struct MapEntry *__prev;
    struct MapEntry *__next;
};
```

Map structure

- This contains the attributes *and* methods
- We will use pointers to functions to make it so we access the methods from the Map object.
- Encapsulation is the bundling of attributes and methods together

```
struct Map {
   /* Private attributes */
   struct MapEntry *__head;
   struct MapEntry *__tail;
   int __count;

   /* Public methods */
   void (*put)(struct Map* self,
               char *key, int value);
   int (*get)(struct Map* self,
              char *key, int def);
   int (*size)(struct Map* self);
   void (*dump)(struct Map* self);
   void (*del)(struct Map* self);
};
```

Constructor

- Allocate the Map object instance and fill it with defaults.
- In a C++ class, the name of Map_put() function is obscured and only accessible via the put() method

```
struct Map * Map_new() {
    struct Map *p = malloc(sizeof(*p));

    p->__head = NULL;
    p->__tail = NULL;
    p->__count = 0;

    p->put = &__Map_put;
    p->get = &__Map_get;
    p->size = &__Map_size;
    p->dump = &__Map_dump;
    p->del = &__Map_del;
    return p;
}
```

Map_dump

- Build a simple debug tool right away ☺
- We are *inside* the class so accessing private values is expected

```
/**
 * map->dump - In effect a toString() except we print
 * the contents of the Map to stdout
 *
 * self - The pointer to the instance of this class.
 */
void __Map_dump(struct Map* self)
{
    struct MapEntry *cur;
    printf("Object Map count=%d\n", self->count);
    for(cur = self->__head; cur != NULL ; cur = cur->__next ) {
         printf("  %s=%d\n", cur->key, cur->value);
    }
}
```

Destructor

- Free the allocated key strings, then the MapEntry structure
- Note that we take cur->next before we free the node, assuming that cur data might be gone.
- At the very end we free the Map structure

```
/**
 * Destructor for the Map Class
 *
 * Loops through and frees all the keys and
 * entries in the map.  The values are integers
 * so there is no need to free them.
 */
void __Map_del(struct Map* self) {
    struct MapEntry *cur, *next;
    cur = self->__head;
    while(cur) {
        free(cur->key);
        /* value is just part of the struct */
        next = cur->__next;
        free(cur);
        cur = next;
    }
    free((void *)self);
}
```

Map_get

- Returns the value stored at the key or a default value
- Pretty simple when you can use __Map_find()

```
/**
 * map->get - Locate and return the value for the
 * corresponding key or a default value
 *
 * self - The pointer to the instance of this class.
 * key - A character pointer to the key value
 * def - A default value to return if the key is
 *       not in the Map
 *
 * Returns an integer
 *
 * Sample call:
 *
 * int ret = map->get(map, "z", 42);
 *
 * This method takes inspiration from the Python code:
 *
 *   value = map.get("key", 42)
 */
int __Map_get(struct Map* self, char *key, int def)
{
    struct MapEntry *retval = __Map_find(self, key);
    if ( retval == NULL ) return def;
    return retval->value;
}
```

Map_put

- This is not used by main() – it is just for in-class use – that is the definition of "private" in OO-speak
- Uses __Map_find() to check if the key is already in the map

```
/**
 * map->put - Add or update an entry in the Map
 *
 * self - The pointer to the instance of this class.
 * key - A character pointer to the key value
 * value - The value to be stored with the associated key
 *
 * If the key is not in the Map, an entry is added.  If there
 * is already an entry in the Map for the key, the value
 * is updated.
 *
 * Sample call:
 *
 *    map->put(map, "x", 42);
 *
 * This method takes inspiration from the Python code:
 *
 *   map["key"] = value
 */
void __Map_put(struct Map* self, char *key, int value) {
    ...
}
```

Iterators

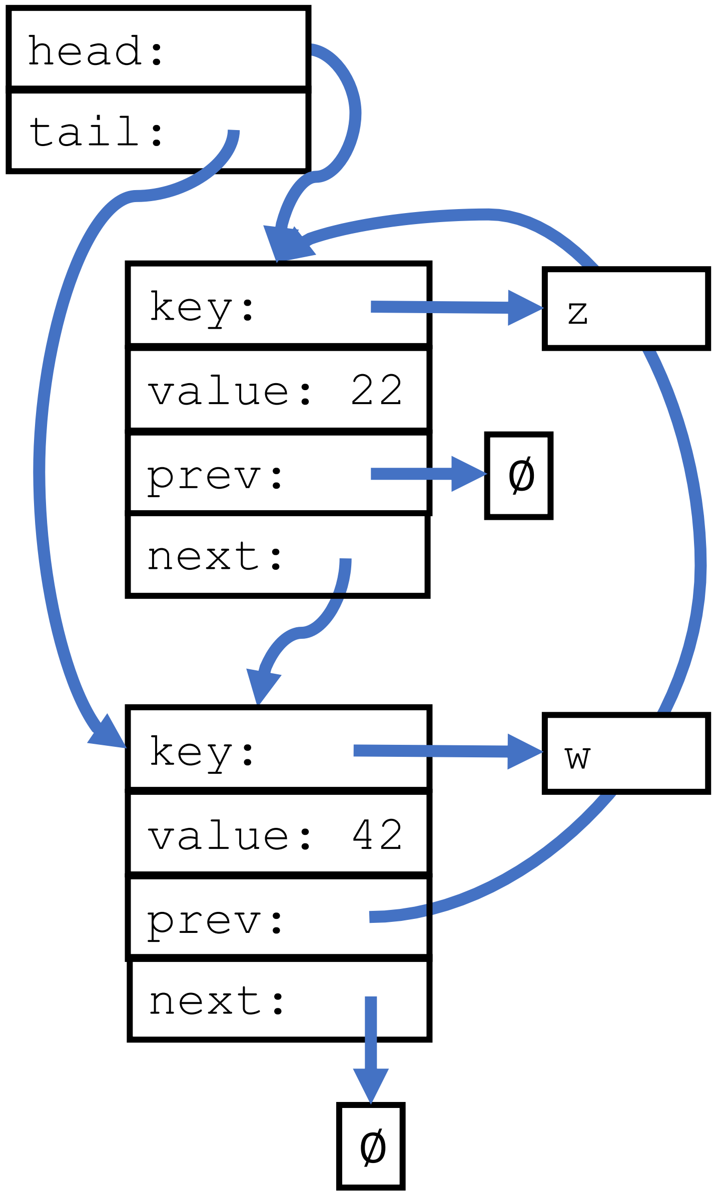
Looping while respecting the abstraction boundary

```
int main(void)
{
    struct Map * map = Map_new();
    struct MapEntry *cur;
    struct MapIter *iter;

    printf("Map test\n");
    map->put(map, "z", 8);
    map->put(map, "z", 1);
    map->put(map, "y", 2);
    map->put(map, "b", 3);
    map->put(map, "a", 4);
    map->dump(map);

    printf("z=%d\n", map->get(map, "z", 42));
    printf("x=%d\n", map->get(map, "x", 42));

    printf("\nIterate\n");
    iter = map->iter(map);
    while(1) {
        cur = iter->next(iter);
        if ( cur == NULL ) break;
        printf("%s=%d\n", cur->key, cur->value);
    }
    iter->del(iter);

    map->del(map);
}
```

```
Testing Map class
Object Map count=4
  z=1
  y=2
  b=3
  a=4
z=1
x=42

Iterate
 y=2
 b=3
 a=4
```

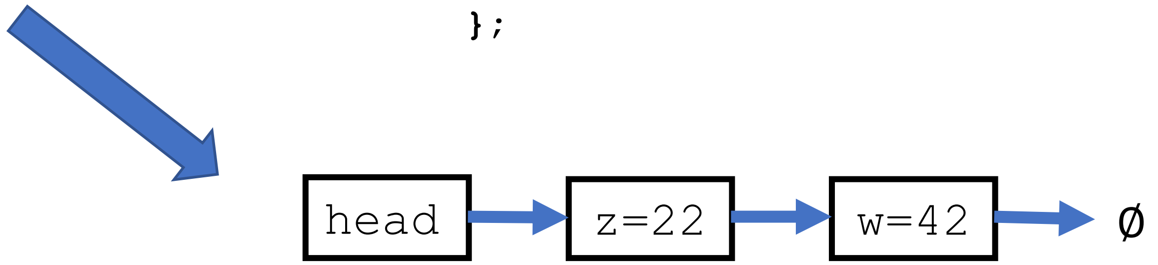
Recall that key and value are public attributes in MapEntry.

cc_04_06.c



Simplifying our Pictures

```
struct MapEntry {
    char *key;   /* public */
    int value;   /* public */
    struct MapEntry *__prev;
    struct MapEntry *__next;
};

struct Map {
   struct MapEntry *__head;
   struct MapEntry *__tail;
   ...
};
```



Separation of Concerns

- Following the practice or "abstraction", object builders make attributes critical to the functioning of the object "private"
- We do not want our calling code to access **head** or **count** – and even worse the object will break badly if the main program starts changing with these values without full understanding of the implementation

```
struct Map {
   /* Attributes */
   struct MapEntry *__head;
   struct MapEntry *__tail;
   int __count;

   /* Methods */
   ...
};


/* This loop is OK in an object
   method - but not in main() */

for (current = map->__head;
     current != NULL;
     current = current->__next )
{

}
```

[Wikipedia: Separation of concerns](#)

About Iterators

- To allow code outside the the object to traverse a list, we use iterators
- Once we have an iterator, each time we call "next" we get the next item in the list until the end

```
x = {'a': 1, 'b': 2, 'c': 3}
print('x is', x)

y = list(x)
print('y is', y)

it = iter(x)
print('it is', it)

while True :
    item = next(it, False)
    if item is False : break
    print('item is', item)
```

```
x is {'a': 1, 'b': 2, 'c': 3}
y is ['a', 'b', 'c']
it is <dict_keyiterator object at 0x100410a40>
item is a
item is b
item is c
```

Wikipedia: Iterator

cc_05_06.py

Using an iterator

- You create the iterator
- Then in a loop, you call next() to get each successive entry in the map, until you exhaust the entries
- We could make key and value private in MapEntry – but we just keep them public to reduce code size on slides

```c
printf("\nIterate\n");
iter = map->iter(map);
while(1) {
    cur = iter->next(iter);
    if ( cur == NULL ) break;
    printf(" %s=%d\n", cur->key, cur->value);
}
iter->del(iter);
```

cc_04_03.c

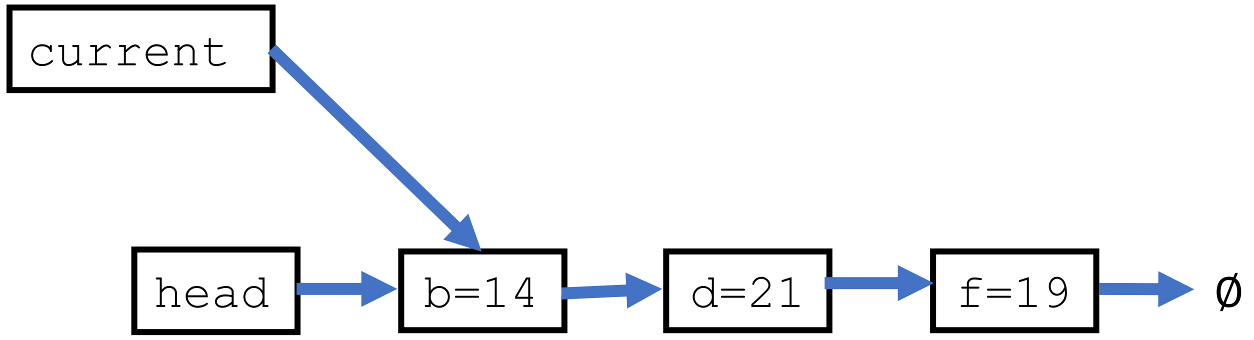
```python
x = {'a': 1, 'b': 2, 'c': 3}

it = iter(x)
while True :
    item = next(it, False)
    if item is False : break
    print('item is', item)
```

cc_04_05.py

Struct MapIter

- The MapIter is a "related class" to it can access "protected" values in MapEntry without violating the abstraction.

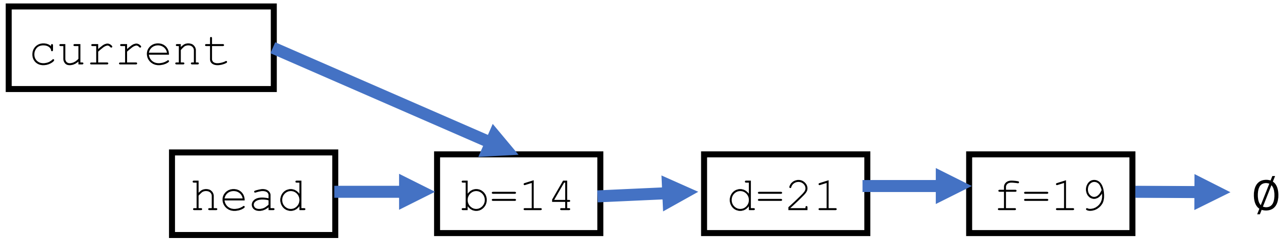
```
/*
 * A MapIter contains the current item and whether
 * this is a forward or reverse iterator.
 */
struct MapIter {
    struct MapEntry *__current;
    struct MapEntry* (*next)(struct MapIter* self);
    void (*del)(struct MapIter* self);
};
```



Map_iter()

- We allocate a MapIter and set it up with initial
- If self->head is NULL, the list is empty.
- We can access protected data in the Map class because the MapIter implementation is part of the Map implementation

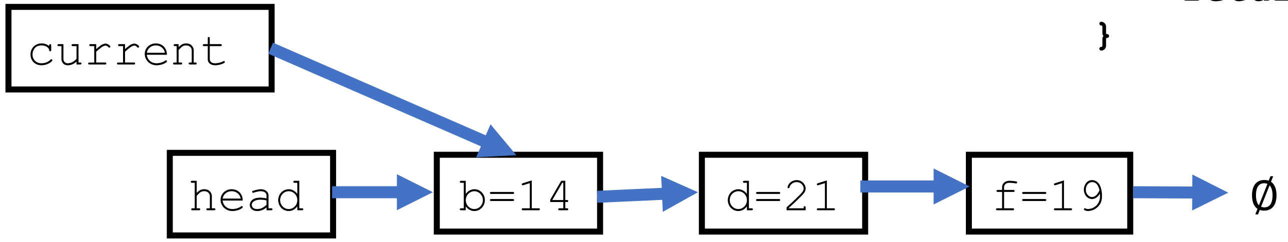
```
/**
 * __Map_iter - Create an iterator from the head the
 * self - The pointer to the instance of this class.
 *
 * returns NULL when there are no entries in the Map
 *
 * This is inspired by the following Python code
 * that creates an iterator from a dictionary:
 *
 *     x = {'a': 1, 'b': 2, 'c': 3}
 *     it = iter(x)
 */
struct MapIter* __Map_iter(struct Map* self)
{
    struct MapIter *iter = malloc(sizeof(*iter));
    iter->__current = self->__head;
    iter->next = &__MapIter_next;
    iter->del = &__MapIter_del;
    return iter;
}
```



MapIter_next()

- Note we must return the current MapEntry before we advance current so we see the first MapEntry

```
/**
 * __MapIter_next - Advance the iterator forwards
 * or backwards and return the next item
 *
 * self - The pointer to the instance of this class.
 *
 * returns NULL when there are no more entries
 *
 * This is inspired by the following Python code:
 *
 *   item = next(iterator, False)
 */
struct MapEntry* __MapIter_next(struct MapIter* self)
{
    struct MapEntry * retval = self->__current;

    if ( retval == NULL) return NULL;
    self->__current = self->__current->__next;
    return retval;
}
```

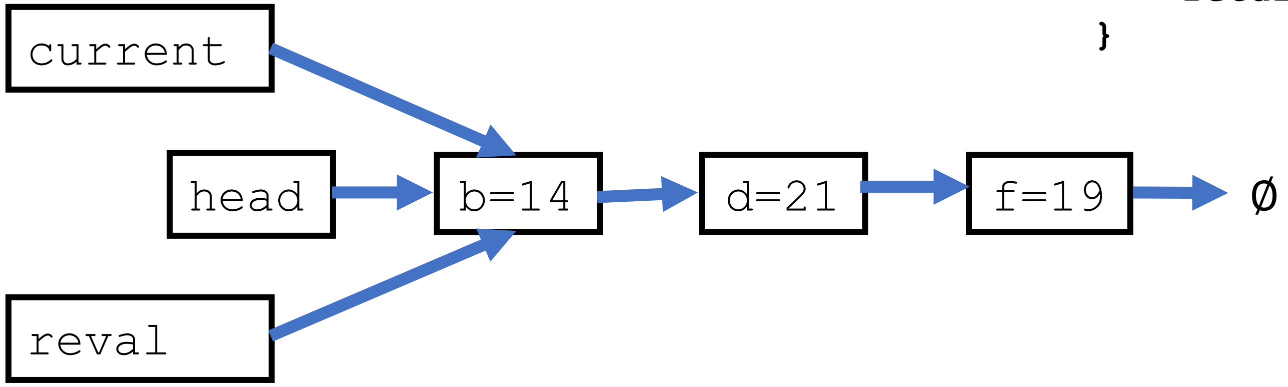


Constructed, before the first call to next()

MapIter_next()

- Note we must return the current MapEntry before we advance current so we see the first MapEntry

```
/**
 * __MapIter_next - Advance the iterator forwards
 * or backwards and return the next item
 *
 * self - The pointer to the instance of this class.
 *
 * returns NULL when there are no more entries
 *
 * This is inspired by the following Python code:
 *
 *   item = next(iterator, False)
 */
struct MapEntry* __MapIter_next(struct MapIter* self)
{
    struct MapEntry * retval = self->__current;

    if ( retval == NULL) return NULL;
    self->__current = self->__current->__next;
    return retval;
}
```

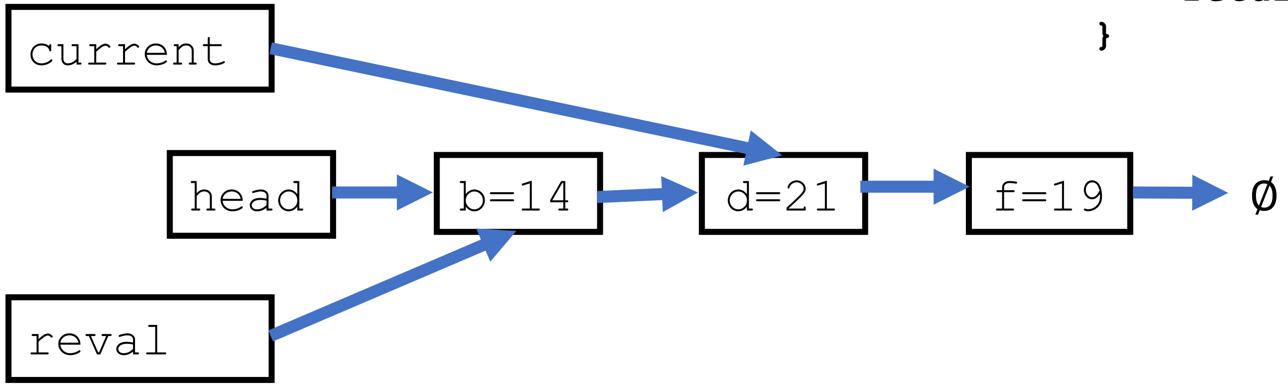


Next is called, grab current, then advance

MapIter_next()

- Note we must return the current MapEntry before we advance current so we see the first MapEntry

```
/**
 * __MapIter_next - Advance the iterator forwards
 * or backwards and return the next item
 *
 * self - The pointer to the instance of this class.
 *
 * returns NULL when there are no more entries
 *
 * This is inspired by the following Python code:
 *
 *   item = next(iterator, False)
 */
struct MapEntry* __MapIter_next(struct MapIter* self)
{
    struct MapEntry * retval = self->__current;

    if ( retval == NULL) return NULL;
    self->__current = self->__current->__next;
    return retval;
}
```

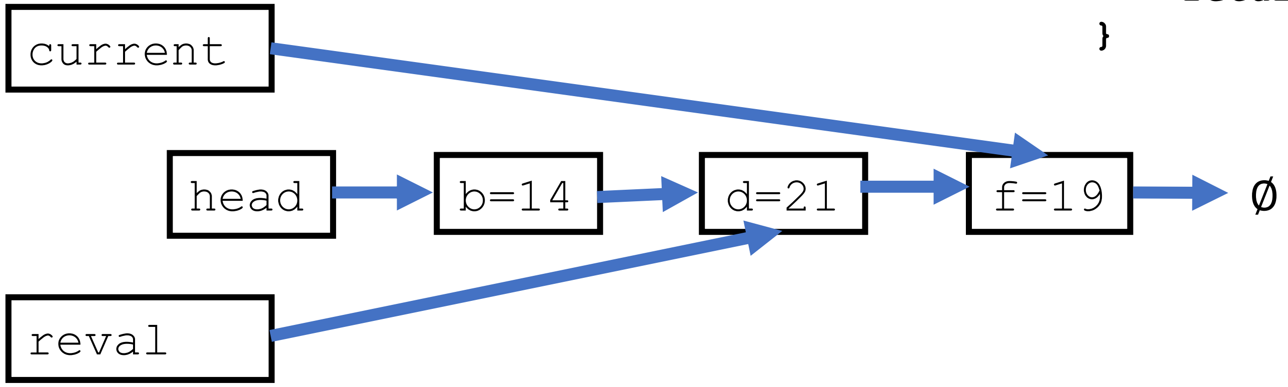


Next returns "b=14" and current is advanced

MapIter_next()

- Note we must return the current MapEntry before we advance current so we see the first MapEntry

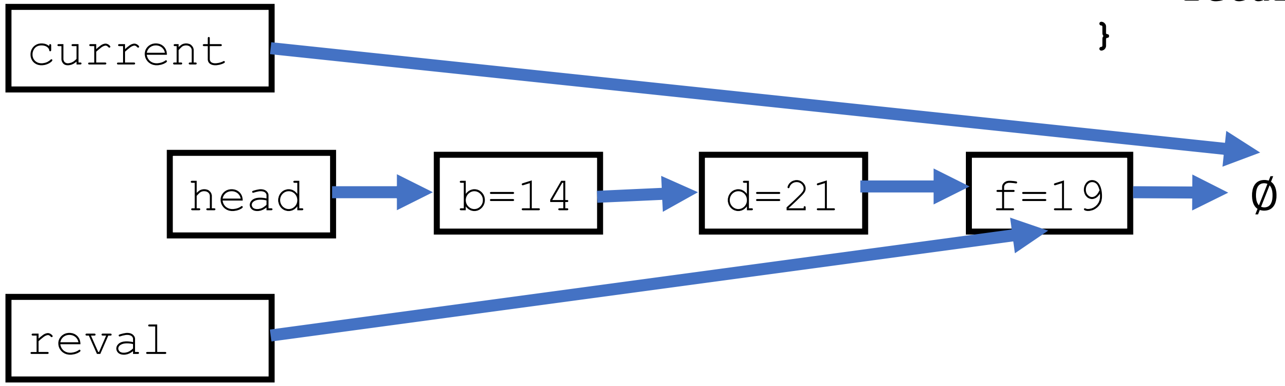
```
/**
 * __MapIter_next - Advance the iterator forwards
 * or backwards and return the next item
 *
 * self - The pointer to the instance of this class.
 *
 * returns NULL when there are no more entries
 *
 * This is inspired by the following Python code:
 *
 *   item = next(iterator, False)
 */
struct MapEntry* __MapIter_next(struct MapIter* self)
{
    struct MapEntry * retval = self->__current;

    if ( retval == NULL) return NULL;
    self->__current = self->__current->__next;
    return retval;
}
```



MapIter_next()

- Note we must return the current MapEntry before we advance current so we see the first MapEntry

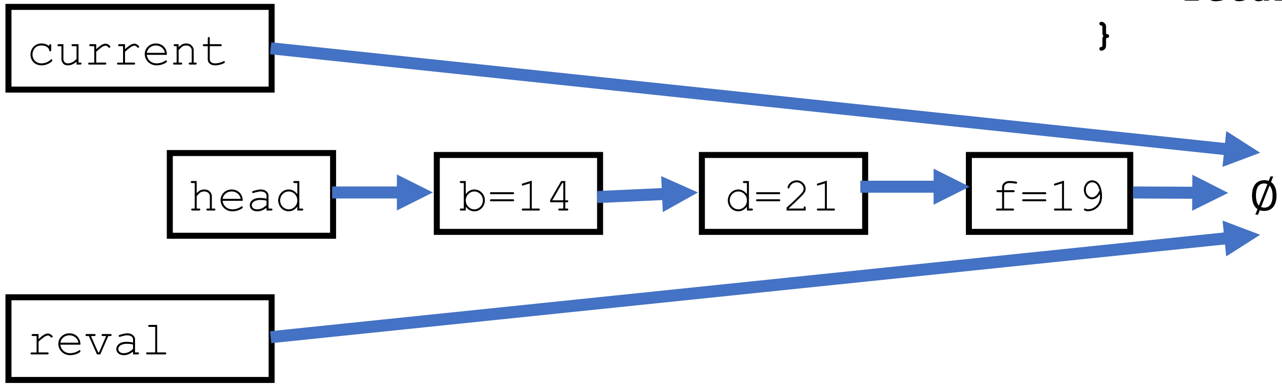
```
/**
 * __MapIter_next - Advance the iterator forwards
 * or backwards and return the next item
 *
 * self - The pointer to the instance of this class.
 *
 * returns NULL when there are no more entries
 *
 * This is inspired by the following Python code:
 *
 *   item = next(iterator, False)
 */
struct MapEntry* __MapIter_next(struct MapIter* self)
{
    struct MapEntry * retval = self->__current;

    if ( retval == NULL) return NULL;
    self->__current = self->__current->__next;
    return retval;
}
```



MapIter_next()

- Note we must return the current MapEntry before we advance current so we see the first MapEntry

```
/**
 * __MapIter_next - Advance the iterator forwards
 * or backwards and return the next item
 *
 * self - The pointer to the instance of this class.
 *
 * returns NULL when there are no more entries
 *
 * This is inspired by the following Python code:
 *
 *   item = next(iterator, False)
 */
struct MapEntry* __MapIter_next(struct MapIter* self)
{
    struct MapEntry * retval = self->__current;

    if ( retval == NULL) return NULL;
    self->__current = self->__current->__next;
    return retval;
}
```



Using an iterator

- You create the iterator
- Then in a loop, you call next() to get each successive entry in the map, until you exhaust the entries
- We could make key and value private in MapEntry – but we just keep them public to reduce code size on slides

```
printf("\nIterate\n");
iter = map->iter(map);
while(1) {
    cur = iter->next(iter);
    if ( cur == NULL ) break;
    printf(" %s=%d\n", cur->key, cur->value);
}
iter->del(iter);
```

cc_04_06.c

```
x = {'a': 1, 'b': 2, 'c': 3}

it = iter(x)
while True :
    item = next(it, False)
    if item is False : break
    print('item is', item)
```

cc_04_05.py

Summary

- We have improved the design of our class interface
  - Abstraction
  - Encapsulation
- We do this to allow multiple Map implementations of varying complexity and improved performance characteristics
  - Tree Structure
  - Hash Structure

Acknowledgements / Contributions

Initial Development: Charles Severance, University of Michigan School of Information

Insert new Contributors and Translators here including names and dates

Continue new Contributors and Translators here

Tree Maps and Hash Maps

Dr. Charles R. Severance

www.cc4e.com

code.cc4e.com (sample code)

online.dr-chuck.com

“Stopping by Woods on a Snowy Evening”

Whose woods these are I think I know.
His house is in the village though;
He will not see me stopping here
To watch his woods fill up with snow.

My little horse must think it queer
To stop without a farmhouse near
Between the woods and frozen lake
The darkest evening of the year.

He gives his harness bells a shake
To ask if there is some mistake.
The only other sound’s the sweep
Of easy wind and downy flake.

The woods are lovely, dark and deep,
But I have promises to keep,
And miles to go before I sleep,
And miles to go before I sleep.

```
name = input('Enter file:')
handle = open(name)

counts = dict()
for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word,0) + 1

bigcount = None
bigword = None
for word,count in counts.items():
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)
```

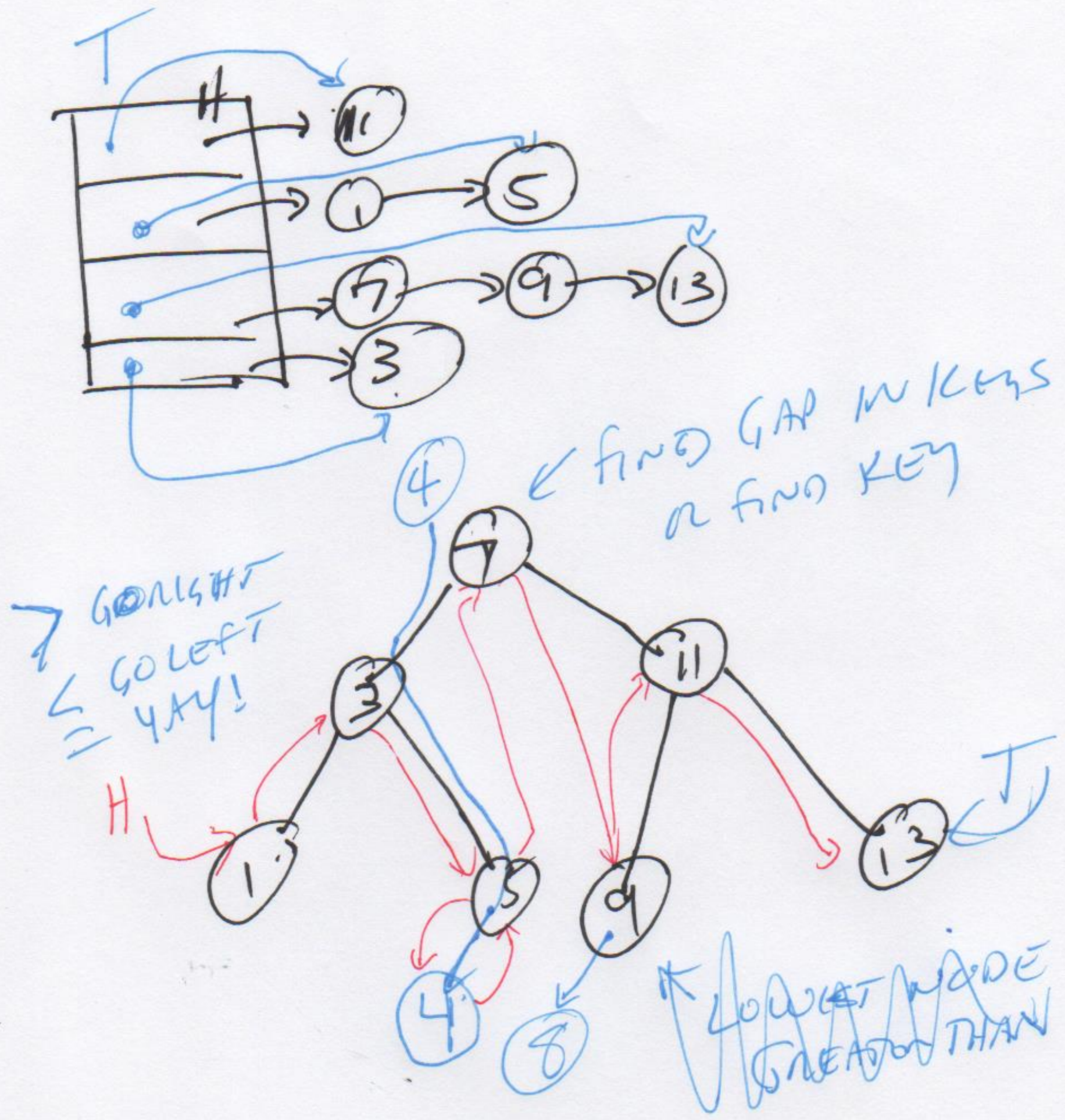
```
python words.py
Enter file: words.txt
to 16
```

```
python words.py
Enter file: clown.txt
the 7
```

Key/Value Implementation Alternatives

- We will build an unordered java.util.HashMap / Python 2 dict()
  - Chapter 6.6
- We will build a sorted java.util.TreeMap – plus an iterator
  - Chapter 6.5 (Simultaneous Linked List + Tree)
  - It would be named java.util.LinkedTreeMap if Java had such a thing ☺

Python	**C++ (capabilities of any map)**	**Java**
Python 2 dict - unordered	Associative	java.util.HashMap
Python 3.1 - OrderedDict	Ordered	java.util.TreeMap (*)
	Unique Keys	java.util.LinkedHashMap
	Map (key / simple value)	
		* No iterator


T
H
← FIND GAP IN KEYS
OR FIND KEY
> GO RIGHT
≤ GO LEFT
YAY!
H
T
LOWEST NODE
GREATER THAN

Hash Map

The answer to the most common programming interview question!

HashMap

- Weird order (like Python 2 dict and Java HashMap)
- Fast insert and lookup (like Python 2 dict and Java HashMap)
- Iterable (like Python 2 dict and Java HashMap)
- Builds on Linked Lists

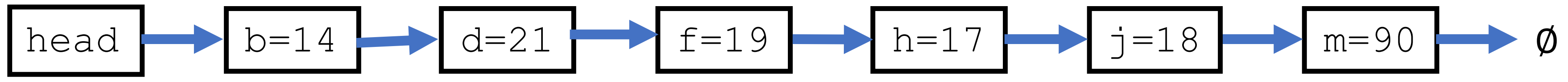
- Surprisingly easy is you really get Linked Lists
- Chapter 6.5.1 and 6.6 in K&R (6.5.2 is harder than 6.6)
- Most popular programming interview question ever!

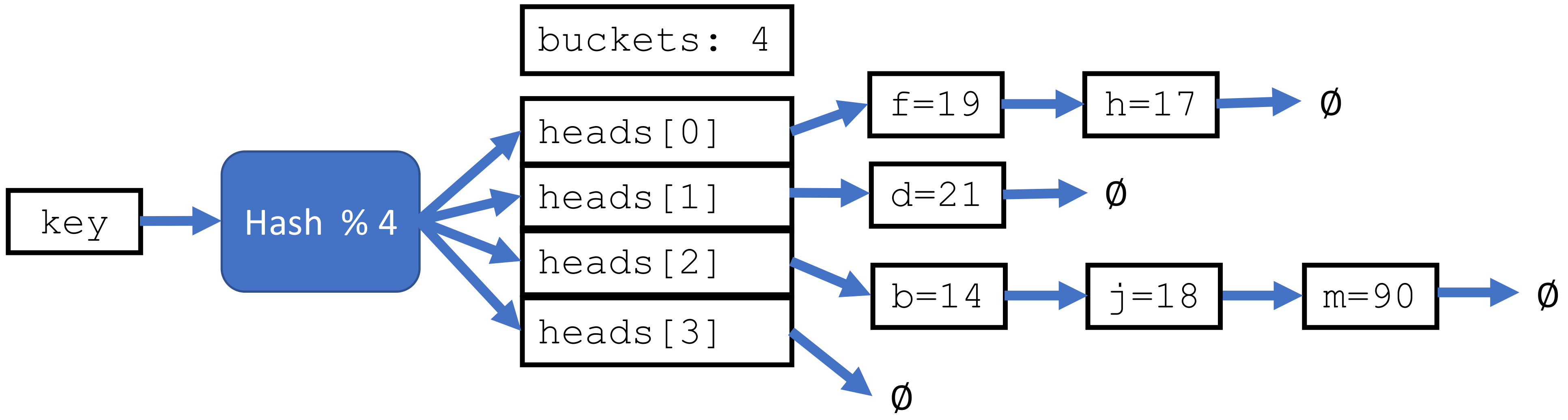
Changing From ListMap to HashMap (1 of 2)

```
struct MapEntry {
    char *key;  /* public */
    int value;  /* public */
    struct MapEntry *__prev;
    struct MapEntry *__next;
};

struct ListMap {
   struct MapEntry *__head;
   struct MapEntry *__tail;
   ...
};
```

```
struct HashMapEntry {
    char *key;  /* public */
    int value;  /* public */
    struct HashMapEntry *__prev;
    struct HashMapEntry *__next;
};

struct HashMap {
    int __buckets;
    struct HashMapEntry *__heads[8];
    struct HashMapEntry *__tails[8];
   ...
};
```

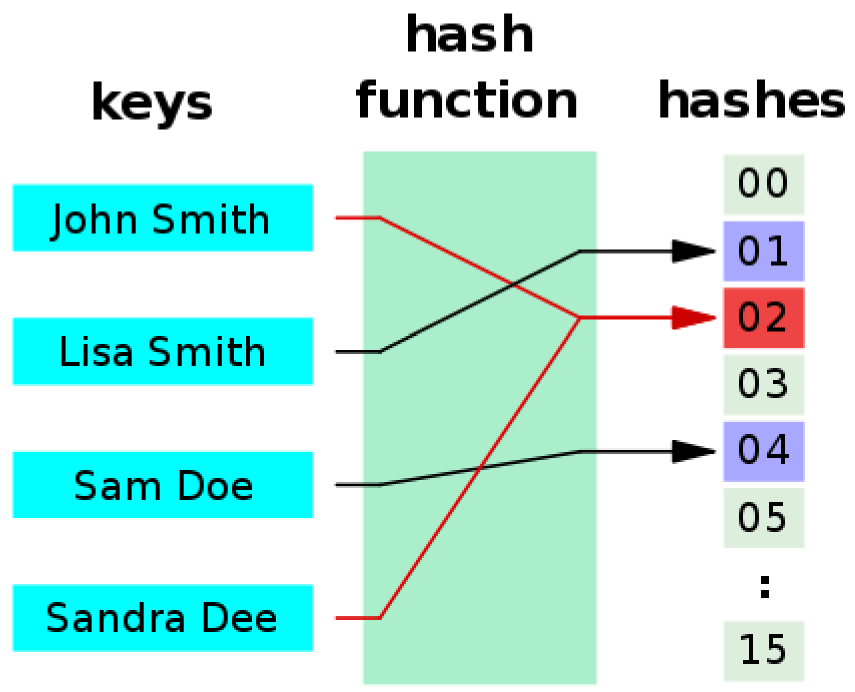
Changing From ListMap to HashMap (2 of 2)





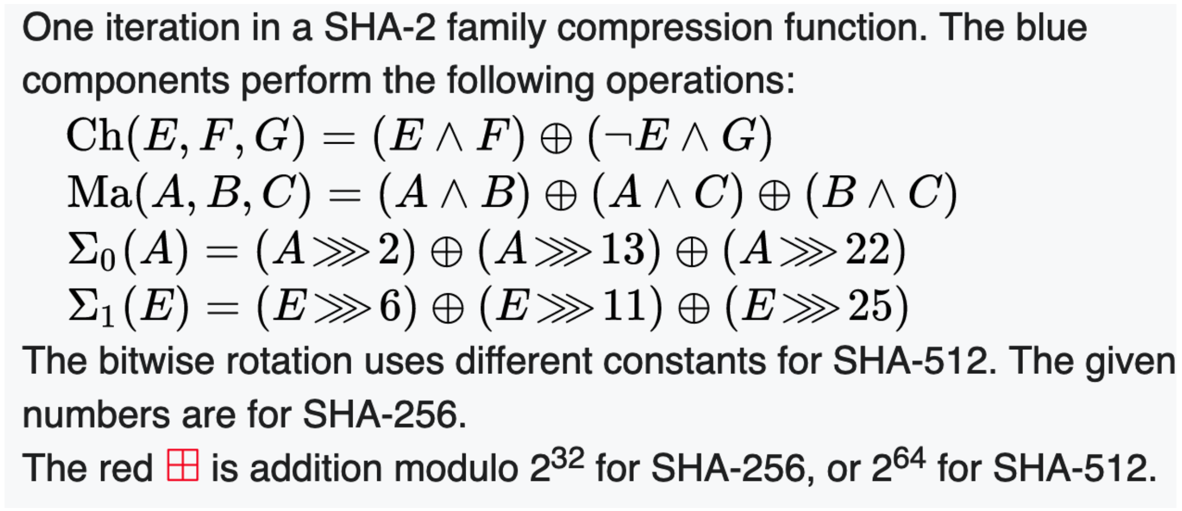
Hashes

A hash function is any algorithm or subroutine that maps large data sets to smaller data sets, called keys. For example, a single integer can serve as an index to an array (cf. associative array). The values returned by a hash function are called hash values, hash codes, hash sums, checksums, or simply hashes.
Hash functions are mostly used to accelerate table lookup or data comparison tasks such as finding items in a database...



[Wikipedia: Hash function]

SHA-256 Compression Function

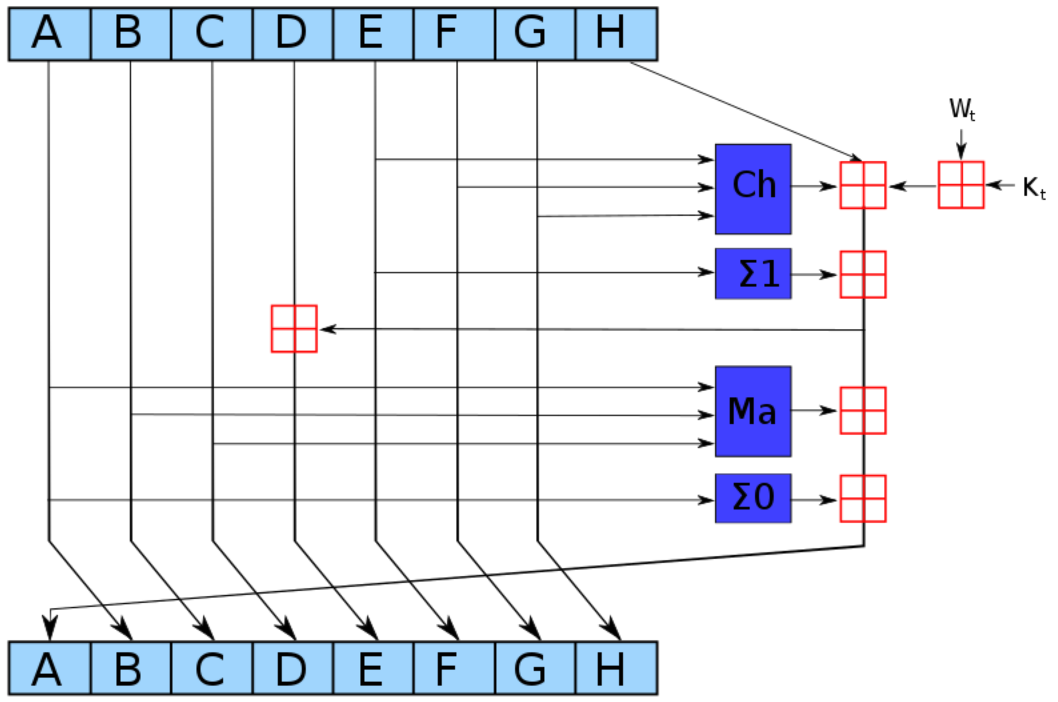


One iteration in a SHA-2 family compression function. The blue components perform the following operations:

$$\mathrm{Ch}(E,F,G) = (E \land F) \oplus (\neg E \land G)$$
$$\mathrm{Ma}(A,B,C) = (A \land B) \oplus (A \land C) \oplus (B \land C)$$
$$\Sigma_0(A) = (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22)$$
$$\Sigma_1(E) = (E \ggg 6) \oplus (E \ggg 11) \oplus (E \ggg 25)$$

The bitwise rotation uses different constants for SHA-512. The given numbers are for SHA-256.

The red ⊞ is addition modulo 2^{32} for SHA-256, or 2^{64} for SHA-512.



Wikipedia: SHA-2

```
int getBucket(char *str, int buckets)
{
    unsigned int hash = 123456;
    printf("\nHashing %s\n", str);
    if ( str == NULL ) return 0;
    for( ; *str ; str++) {
        hash = ( hash << 3 ) ^ *str;
        printf("%c 0x%08x %d\n", *str,
               hash, hash % buckets);
    }
    return hash % buckets;
}

int main() {
    int h;

    h = getBucket("Hi", 8);
    h = getBucket("Hello", 8);
    h = getBucket("World", 8);
}
```

```
Hashing Hi
H 0x000f1248 0
i 0x00789229 1

Hashing Hello
H 0x000f1248 0
e 0x00789225 5
l 0x03c49144 4
l 0x1e248a4c 4
o 0xf124520f 7

Hashing World
W 0x000f1257 7
o 0x007892d7 7
r 0x03c496ca 2
l 0x1e24b63c 4
d 0xf125b184 4
```

cc_05_01.c

Lets Build our HashMap

First, make a copy of our List Map and change a (very) few things

```
struct HashMap * HashMap_new() {
    struct HashMap *p = malloc(sizeof(*p));

    p->__buckets = 8;
    for(int i=0; i < p->__buckets; i++ ) {
        p->__heads[i] = NULL;
        p->__tails[i] = NULL;
    }

    p->__count = 0;

    p->put = &__HashMap_put;
    p->get = &__HashMap_get;
    p->size = &__HashMap_size;
    p->dump = &__HashMap_dump;
    p->iter = &__HashMap_iter;
    p->del = &__HashMap_del;
    return p;
}
```

```
struct HashMap {
    int __buckets;
    struct HashMapEntry *__heads[8];
    struct HashMapEntry *__tails[8];
    int __count;
   ...
};
```

```
struct HashMapEntry* __HashMap_find(struct HashMap* self, char *key, int bucket)
{
    struct HashMapEntry *cur;
    if ( self == NULL || key == NULL ) return NULL;
    for(cur = self->__heads[bucket]; cur != NULL ; cur = cur->__next ) {
        if(strcmp(key, cur->key) == 0 ) return cur;
    }
    return NULL;
}


int __HashMap_get(struct HashMap* self, char *key, int def)
{
    int bucket = getBucket(key, self->__buckets);
    struct HashMapEntry *retval = __HashMap_find(self, key, bucket);
    if ( retval == NULL ) return def;
    return retval->value;
}
```

```c
void __Map_put(struct Map* self, char *key, int value) {

    struct MapEntry *old, *new;
    char *new_key;


    old = __Map_find(self, key);
    if ( old != NULL ) {
        old->value = value;
        return;
    }

    /* Not found - time to insert */
    new = malloc(sizeof(*new));
    new->__next = NULL;

    if ( self->__head == NULL ) self->__head = new;
    if ( self->__tail != NULL ) self->__tail->__next = new;
    new->__prev = self->__tail;
    self->__tail = new;

    ...
}
```

```
void __HashMap_put(struct HashMap* self, char *key, int value) {
    int bucket;
    struct HashMapEntry *old, *new;
    char *new_key, *new_value;

    bucket = getBucket(key, self->__buckets);
    old = __HashMap_find(self, key, bucket);
    if ( old != NULL ) {
        old->value = value;
        return;
    }

    /* Not found - time to insert */
    new = malloc(sizeof(*new));
    new->__next = NULL;

    if ( self->__heads[bucket] == NULL ) self->__heads[bucket] = new;
    if ( self->__tails[bucket] != NULL ) self->__tails[bucket]->__next = new;
    new->__prev = self->__tails[bucket];
    self->__tails[bucket] = new;

    ...
}
```

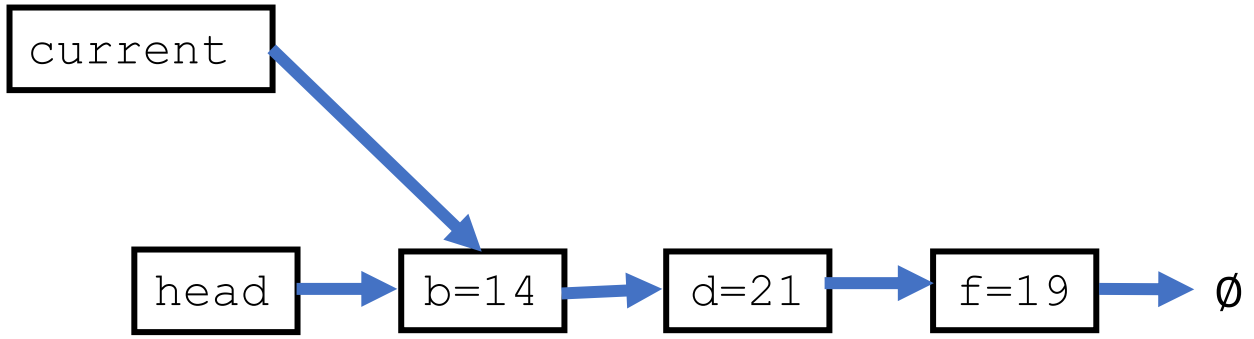
```
void __HashMap_dump(struct HashMap* self)
{
    int i;
    struct HashMapEntry *cur;

    printf("Object HashHashMap@%p count=%d buckets=%d\n",
                              self, self->__count, self->__buckets);
    for(i = 0; i < self->__buckets; i++ ) {
        for(cur = self->__heads[i]; cur != NULL ; cur = cur->__next ) {
            printf(" %s=%d [%d]\n", cur->key, cur->value, i);
        }
    }
}
```

```
Object HashHashMap@0x6000035ac000 count=4
buckets=8
 y=2 [1]
 a=4 [1]
 z=1 [2]
 b=3 [2]
```

Struct MapIter (review)

- The MapIter is a "related class" to it can access "protected" values in MapEntry without violating the abstraction.

```
/*
 * A MapIter contains the current item and whether
 * this is a forward or reverse iterator.
 */
struct MapIter {
   struct MapEntry *__current;
   struct MapEntry* (*next)(struct MapIter* self);
   void (*del)(struct MapIter* self);
};
```



Using an iterator

- You create the iterator
- Then in a loop, you call next() to get each successive entry in the map, until you exhaust the entries
- We could make key and value private in MapEntry – but we just keep them public to reduce code size on slides

```c
printf("\nIterate\n");
iter = map->iter(map);
while(1) {
    cur = iter->next(iter);
    if ( cur == NULL ) break;
    printf(" %s=%d\n", cur->key, cur->value);
}
iter->del(iter);
```

cc_04_03.c

```python
x = {'a': 1, 'b': 2, 'c': 3}

it = iter(x)
while True :
    item = next(it, False)
    if item is False : break
    print('item is', item)
```

cc_04_05.py

Hash Map Iterator Requirements

- Must move through all buckets
- Must skip empty buckets

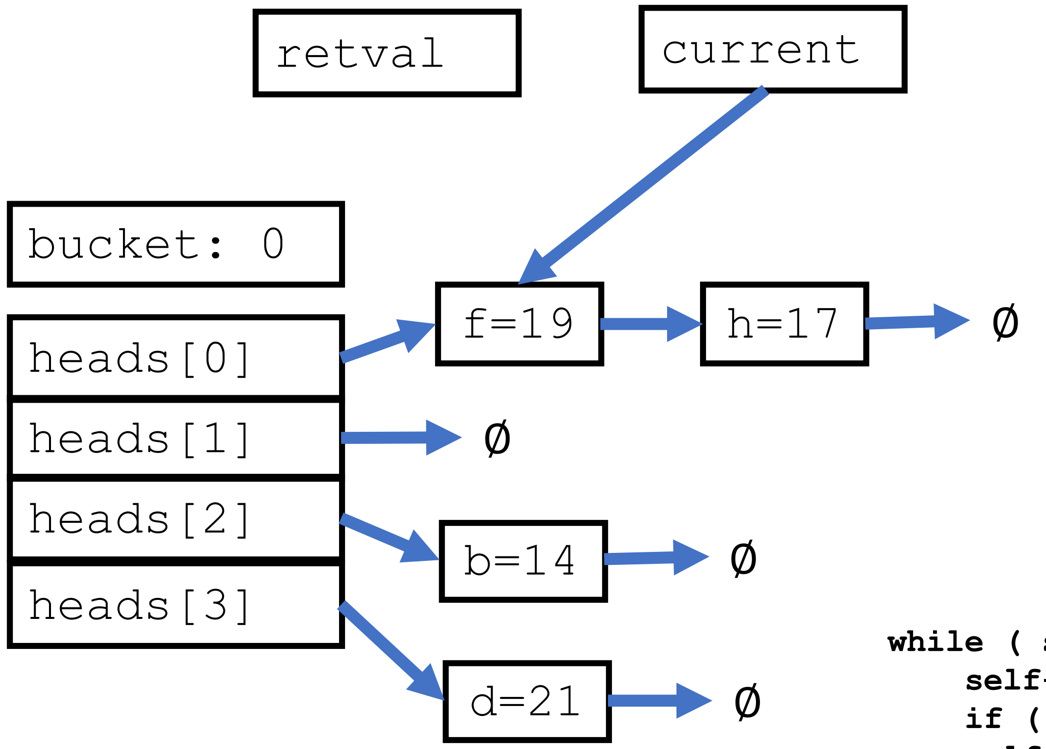
```
struct HashMapIter {
    int __bucket;
    struct HashMap *__map;
    struct HashMapEntry *__current;

    struct HashMapEntry* (*next)(struct HashMapIter* self);
    void (*del)(struct HashMapIter* self);
};
```

```
/**
 * map->iter - Create an iterator from the head of the HashMap
 *
 * self - The pointer to the instance of this class.
 *
 * returns NULL when there are no entries in the HashMap
 *
 * This is inspired by the following Python code
 * that creates an iterator from a dictionary:
 *
 *     x = {'a': 1, 'b': 2, 'c': 3}
 *     it = iter(x)
 */
struct HashMapIter* __HashMap_iter(struct HashMap* map)
{
    struct HashMapIter *iter = malloc(sizeof(*iter));
    iter->__map = map;
    iter->__bucket = 0;
    iter->__current = map->__heads[iter->__bucket];
    iter->next = &__HashMapIter_next;
    iter->del = &__HashMapIter_del;
    return iter;
}
```

HashMapIter next

```
/**
 * HashMapIter_next - Advance the iterator forwards
 * This is inspired by the following Python code:
 *
 *   item = next(iterator, None)
 */
struct HashMapEntry* __HashMapIter_next(struct HashMapIter* self)
{
    struct HashMapEntry* retval;

    /* If we are at the end of a chain and there are still more buckets
     * scan for a bucket that is not NULL */
    while ( self->__current == NULL) {
        self->__bucket++;
        if ( self->__bucket >= self->__map->__buckets ) return NULL;
        self->__current = self->__map->__heads[self->__bucket];
    }

    retval = self->__current;
    if ( self->__current != NULL ) self->__current = self->__current->__next;
    return retval;
}
```

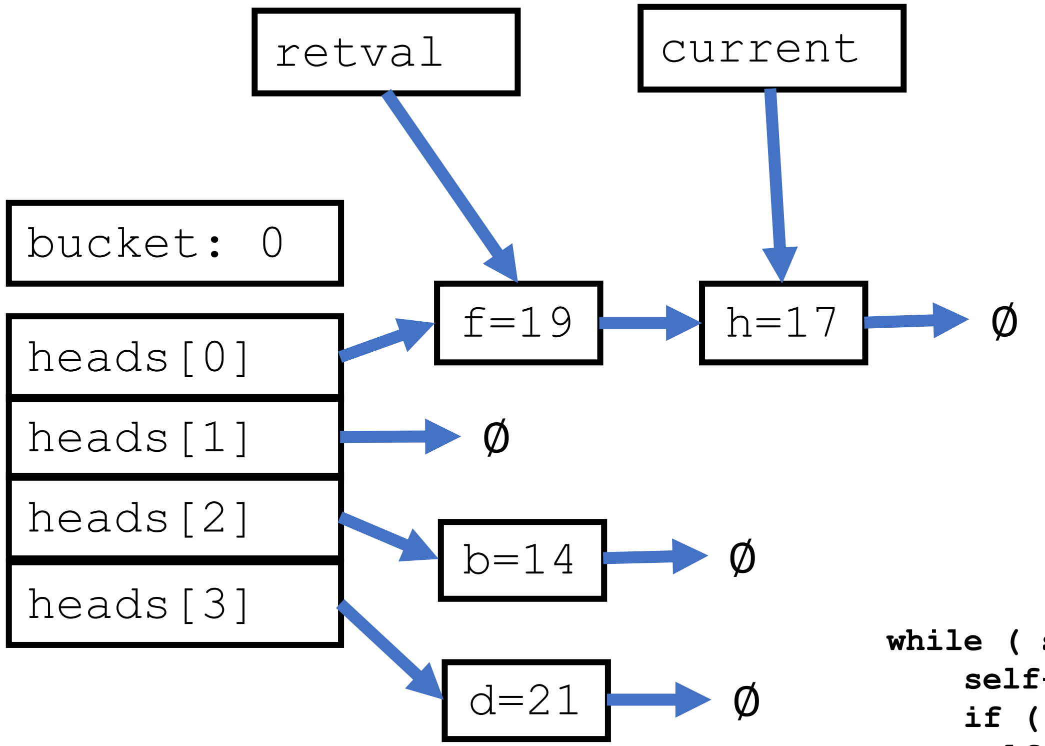
A HashMap Iterator In Action (1 of 10)



**Just constructed,
in the first
call to next()**

```
while ( self->__current == NULL) {
    self->__bucket++;
    if ( self->__bucket >= self->__map->__buckets ) return NULL;
    self->__current = self->__map->__heads[self->__bucket];
}

retval = self->__current;
if ( self->__current != NULL ) self->__current = self->__current->__next;
return retval;
```

A HashMap Iterator In Action (2 of 10)

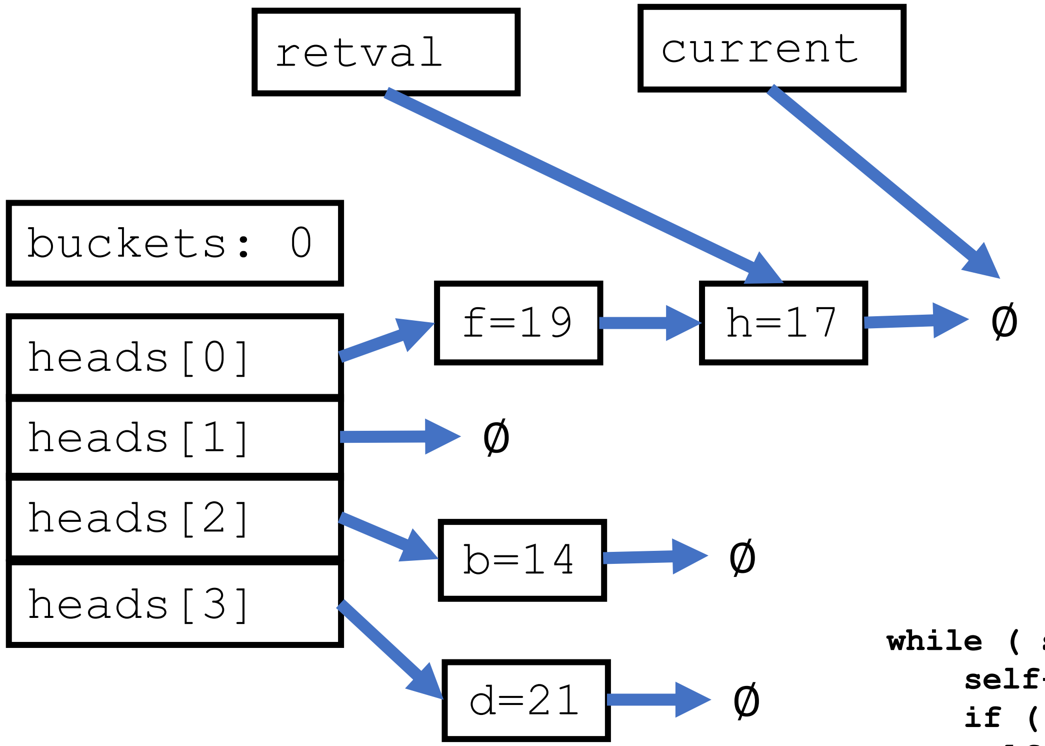


Returned values:

f=19

```
while ( self->__current == NULL) {
    self->__bucket++;
    if ( self->__bucket >= self->__map->__buckets ) return NULL;
    self->__current = self->__map->__heads[self->__bucket];
}

retval = self->__current;
if ( self->__current != NULL ) self->__current = self->__current->__next;
return retval;
```

A HashMap Iterator In Action (3 of 10)

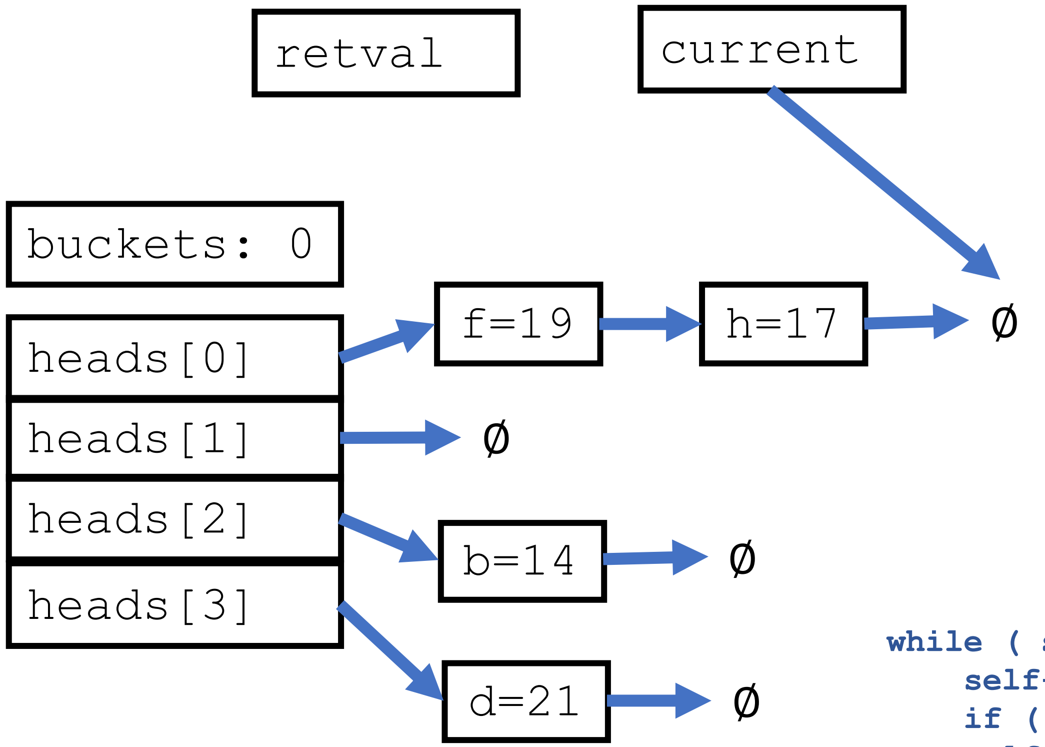


Second call
Returned values:

f=19
h=17

```
while ( self->__current == NULL) {
    self->__bucket++;
    if ( self->__bucket >= self->__map->__buckets ) return NULL;
    self->__current = self->__map->__heads[self->__bucket];
}

retval = self->__current;
if ( self->__current != NULL ) self->__current = self->__current->__next;
return retval;
```

A HashMap Iterator In Action (4 of 10)

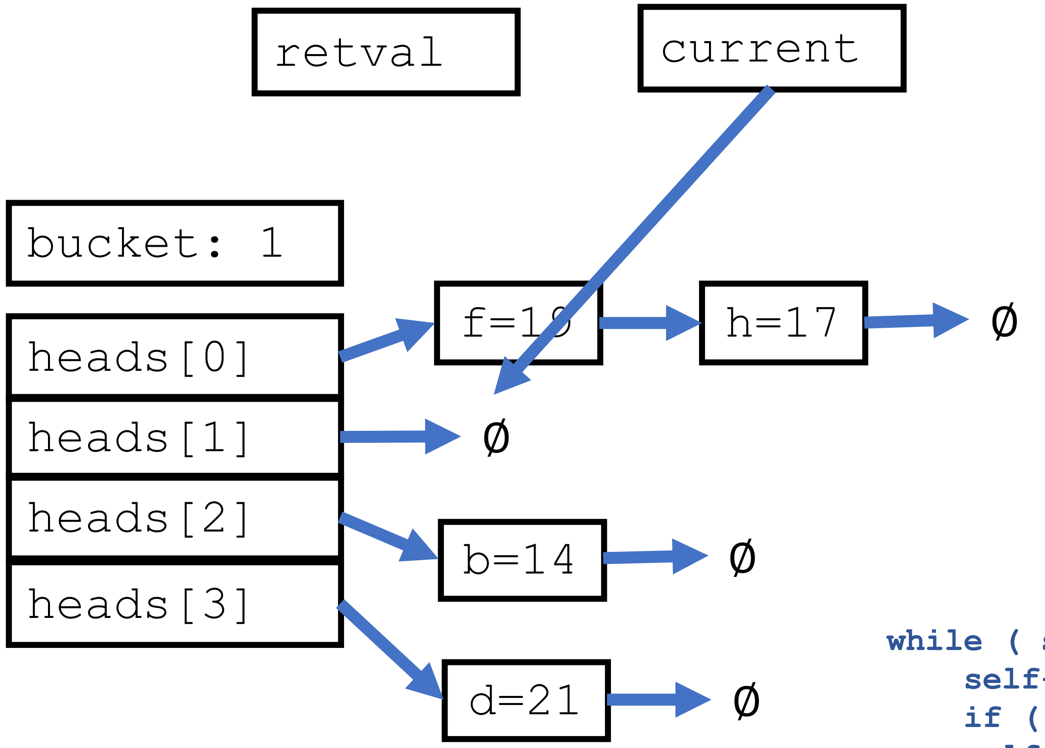


Third call

f=19
h=17
???

```
while ( self->__current == NULL) {
    self->__bucket++;
    if ( self->__bucket >= self->__map->__buckets ) return NULL;
    self->__current = self->__map->__heads[self->__bucket];
}

retval = self->__current;
if ( self->__current != NULL ) self->__current = self->__current->__next;
return retval;
```

A HashMap Iterator In Action (5 of 10)

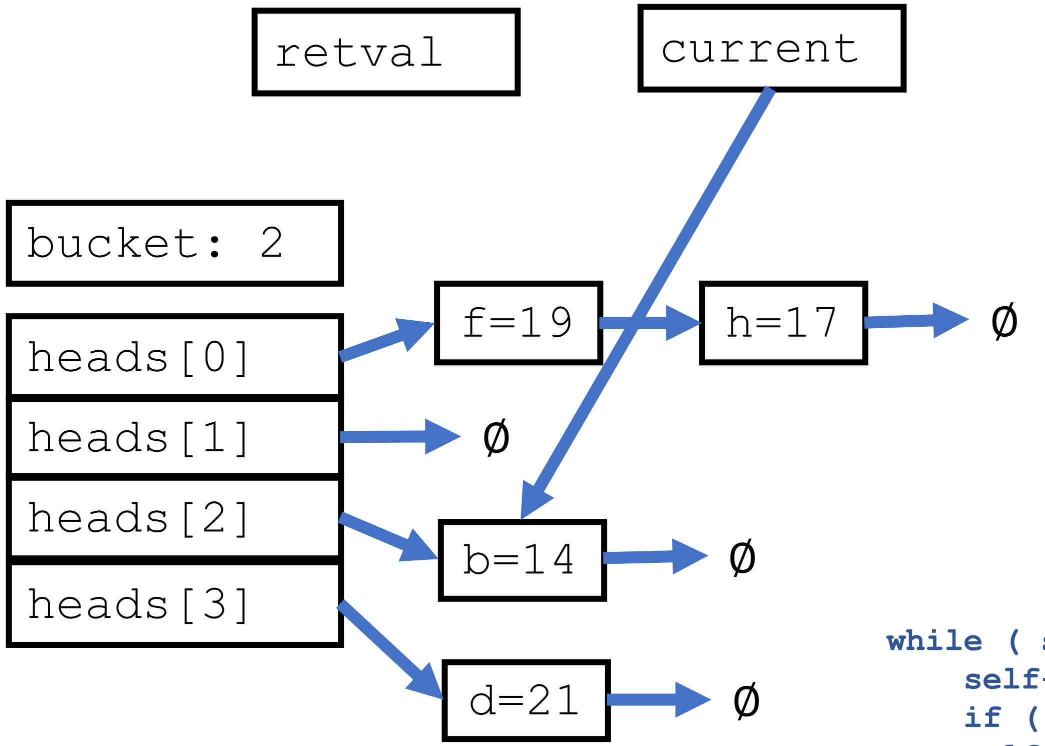


Third call, in while loop

f=19
h=17
???

```
while ( self->__current == NULL) {
    self->__bucket++;
    if ( self->__bucket >= self->__map->__buckets ) return NULL;
    self->__current = self->__map->__heads[self->__bucket];
}

retval = self->__current;
if ( self->__current != NULL ) self->__current = self->__current->__next;
return retval;
```

A HashMap Iterator In Action (6 of 10)

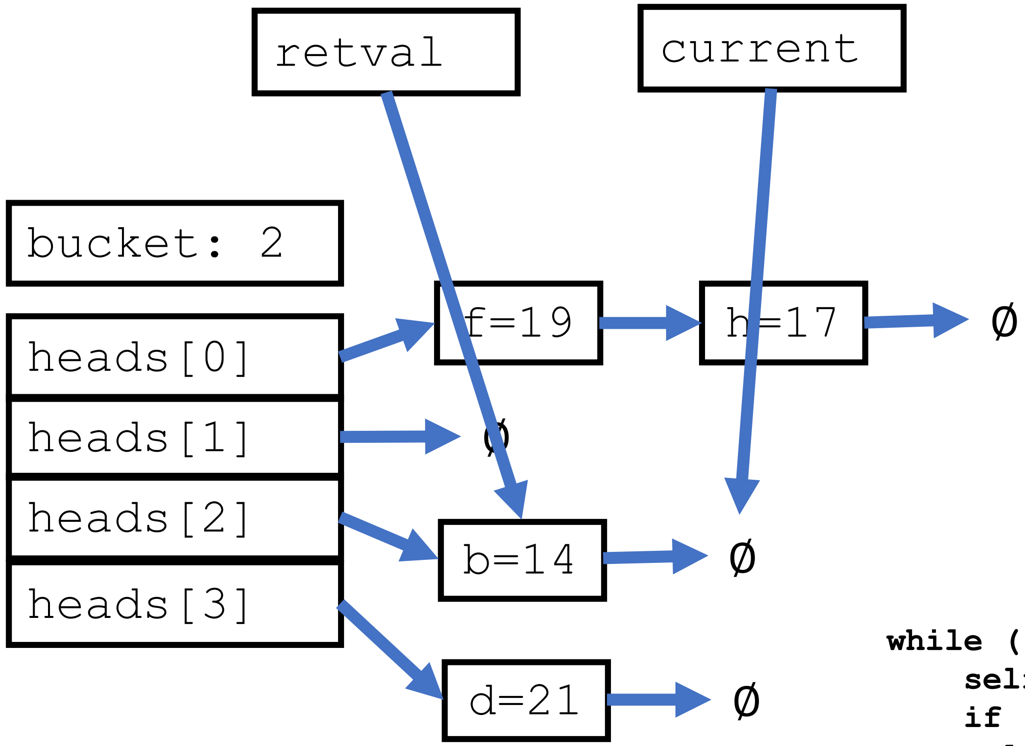


Third call, out of the while loop

f=19
h=17
???

```
while ( self->__current == NULL) {
    self->__bucket++;
    if ( self->__bucket >= self->__map->__buckets ) return NULL;
    self->__current = self->__map->__heads[self->__bucket];
}

retval = self->__current;
if ( self->__current != NULL ) self->__current = self->__current->__next;
return retval;
```

A HashMap Iterator In Action (7 of 10)

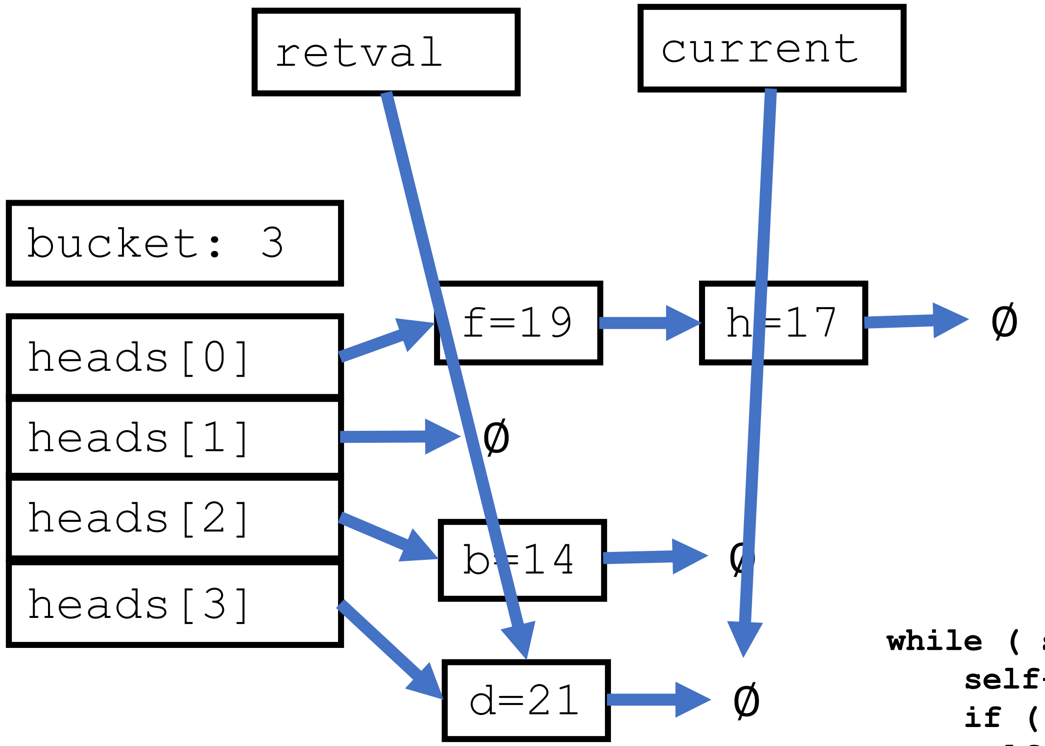


Third call,
Returned value:

f=19
h=17
b=14

```
while ( self->__current == NULL) {
    self->__bucket++;
    if ( self->__bucket >= self->__map->__buckets ) return NULL;
    self->__current = self->__map->__heads[self->__bucket];
}

retval = self->__current;
if ( self->__current != NULL ) self->__current = self->__current->__next;
return retval;
```

A HashMap Iterator In Action (8 of 10)

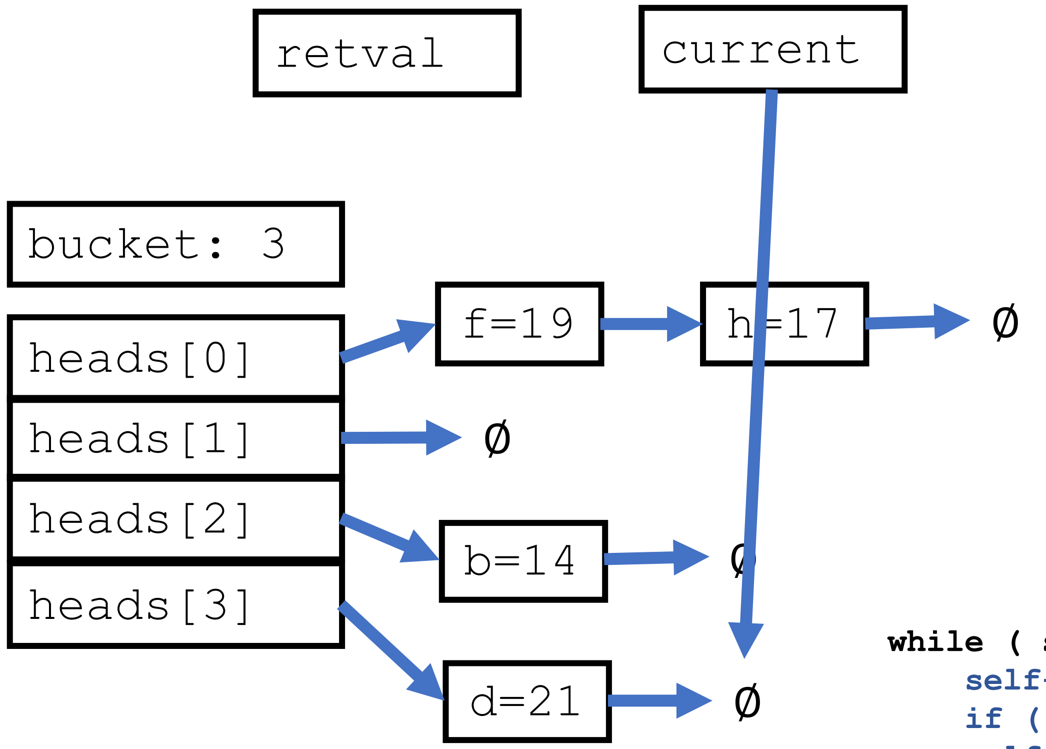


Fourth call,
Returned values:

f=19
h=17
b=14
d=21

```
while ( self->__current == NULL) {
    self->__bucket++;
    if ( self->__bucket >= self->__map->__buckets ) return NULL;
    self->__current = self->__map->__heads[self->__bucket];
}

retval = self->__current;
if ( self->__current != NULL ) self->__current = self->__current->__next;
return retval;
```

A HashMap Iterator In Action (9 of 10)

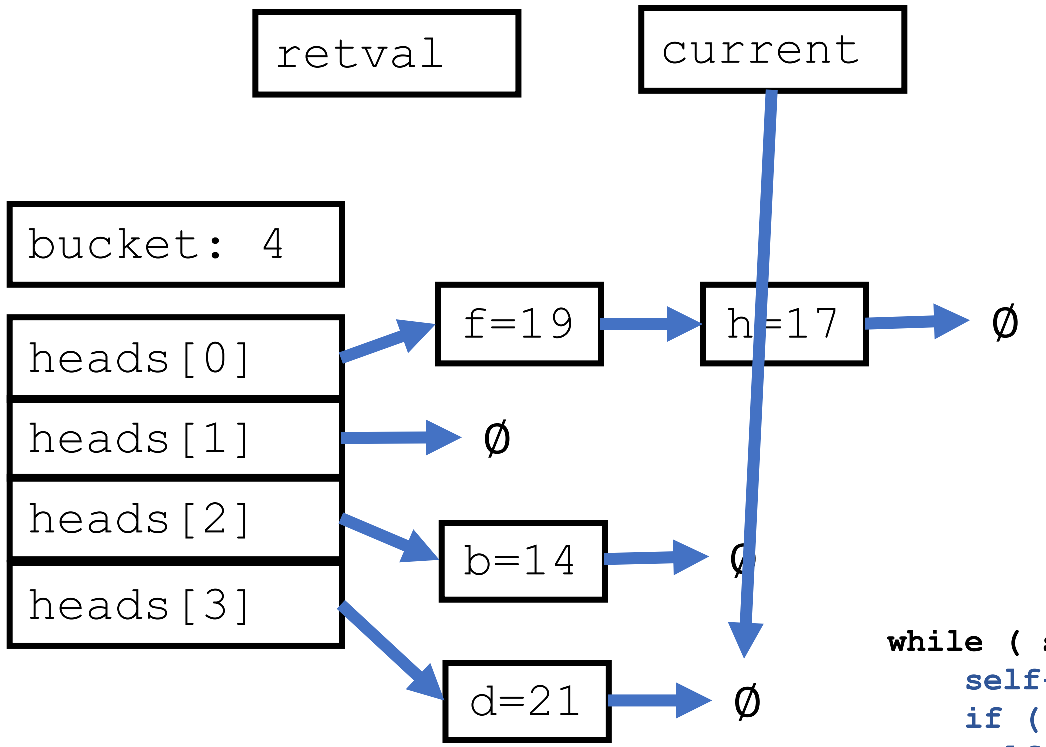


Fifth call,
Current is null

f=19
h=17
b=14
d=21
???

```
while ( self->__current == NULL) {
    self->__bucket++;
    if ( self->__bucket >= self->__map->__buckets ) return NULL;
    self->__current = self->__map->__heads[self->__bucket];
}

retval = self->__current;
if ( self->__current != NULL ) self->__current = self->__current->__next;
return retval;
```

A HashMap Iterator In Action (10 of 10)



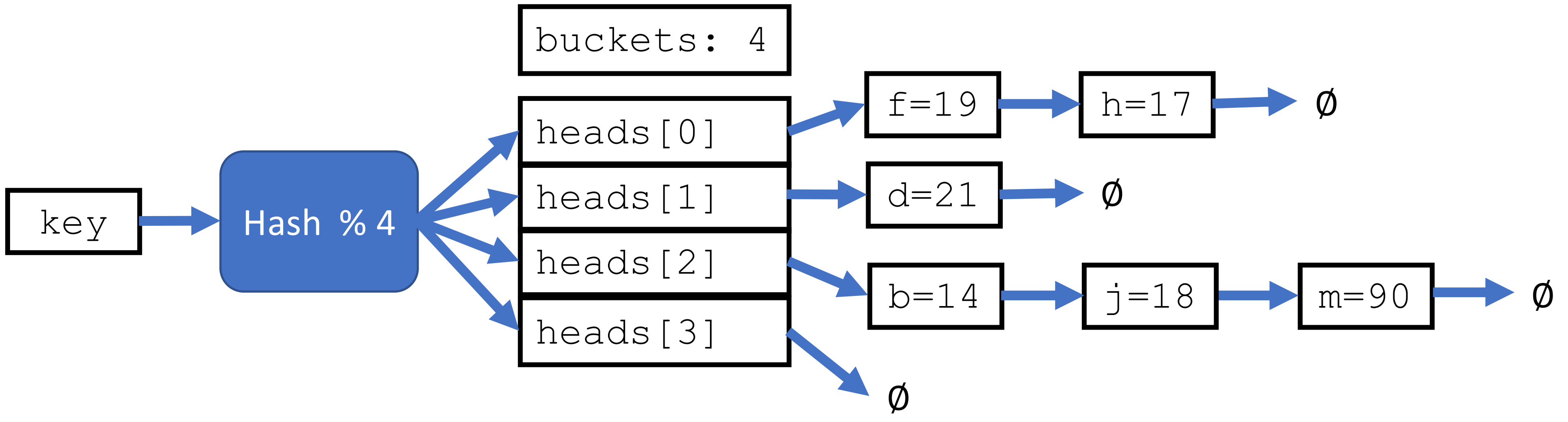
Fifth call
returns NULL

f=19
h=17
b=14
d=21
NULL

```
while ( self->__current == NULL) {
    self->__bucket++;
    if ( self->__bucket >= self->__map->__buckets ) return NULL;
    self->__current = self->__map->__heads[self->__bucket];
}

retval = self->__current;
if ( self->__current != NULL ) self->__current = self->__current->__next;
return retval;
```

We have more to do in our HashMap

- At some point if the linked lists from each bucket get too long performance suffers.
- When the map reaches a certain load factor, the bucket size is increased (often doubled) and the entries are rehashed and spread across more buckets reducing the average chain length



HashMap Iterator

- It is surprisingly simple
- It is very similar to the list iterator
- This simple HashMap iterators presents results in a pseudo-random order
- We are at a point where we have built the the two foundational types of Python 2.0

- Next we will explore a linked list that maintains sorted order and can be iterated in key order

LinkedTreeMap

A modern, flexible key/value store

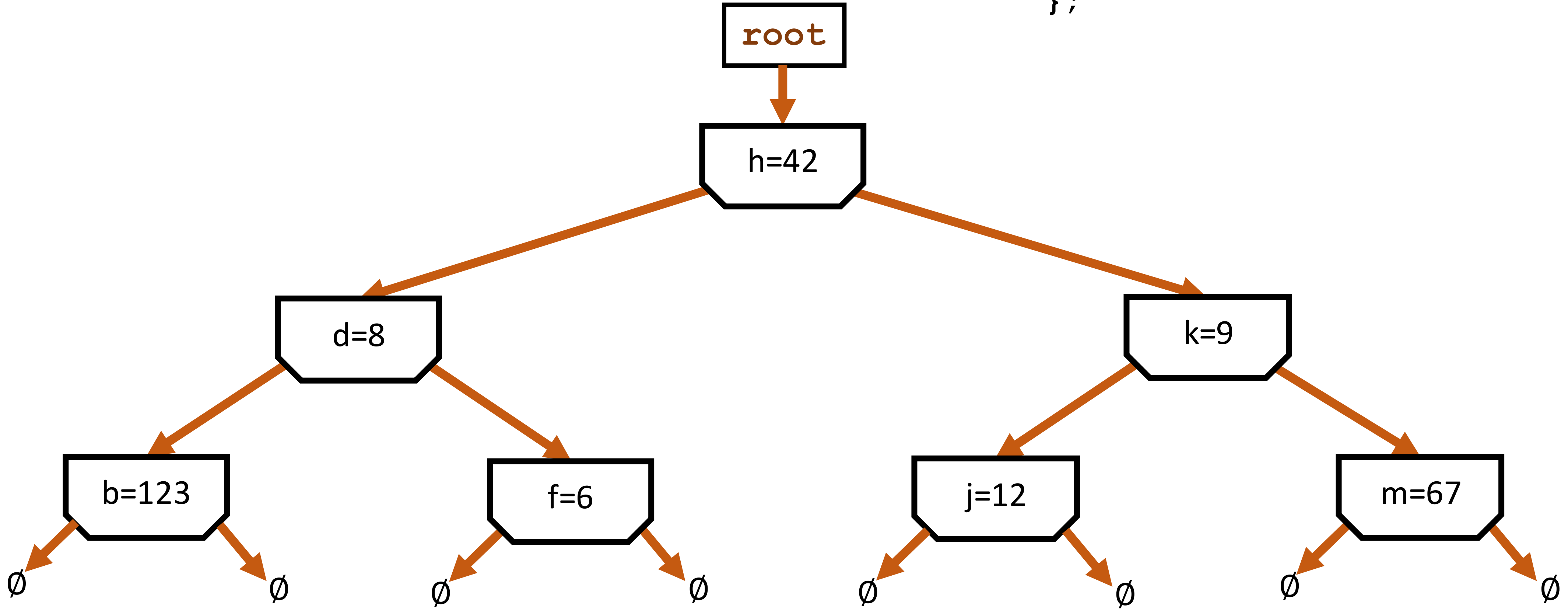
A LinkedTreeMap

- Stays ordered (like Python OrderedDict)
- Stays sorted (like Java TreeMap)
- Can be iterated (like C++ map, and OrderedDict, but not TreeMap)
- Fast lookup using ***secondary*** binary tree index (6.5.2)

- Simultaneously a sorted linked list (6.5.1) and binary tree (6.5.2)

```
struct TreeMap {
    struct TreeMapEntry *__root;
    int __count;
    …
}
```

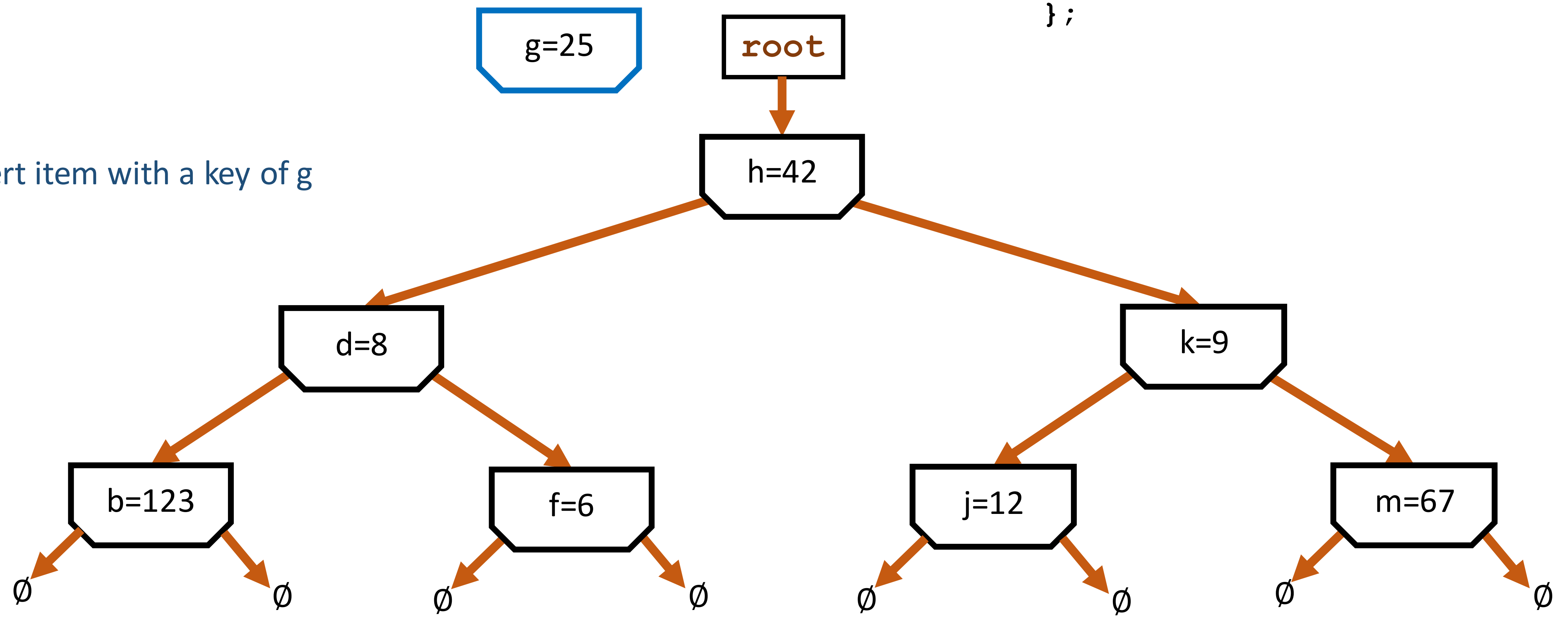
```
struct TreeMapEntry {
    char *key;   /* public */
    int value;   /* public */
    struct TreeMapEntry *__left;
    struct TreeMapEntry *__right;
};
```



```
struct TreeMap {
    struct TreeMapEntry *__root;
    int __count;
    …
}
```

```
struct TreeMapEntry {
    char *key;   /* public */
    int value;   /* public */
    struct TreeMapEntry *__left;
    struct TreeMapEntry *__right;
};
```

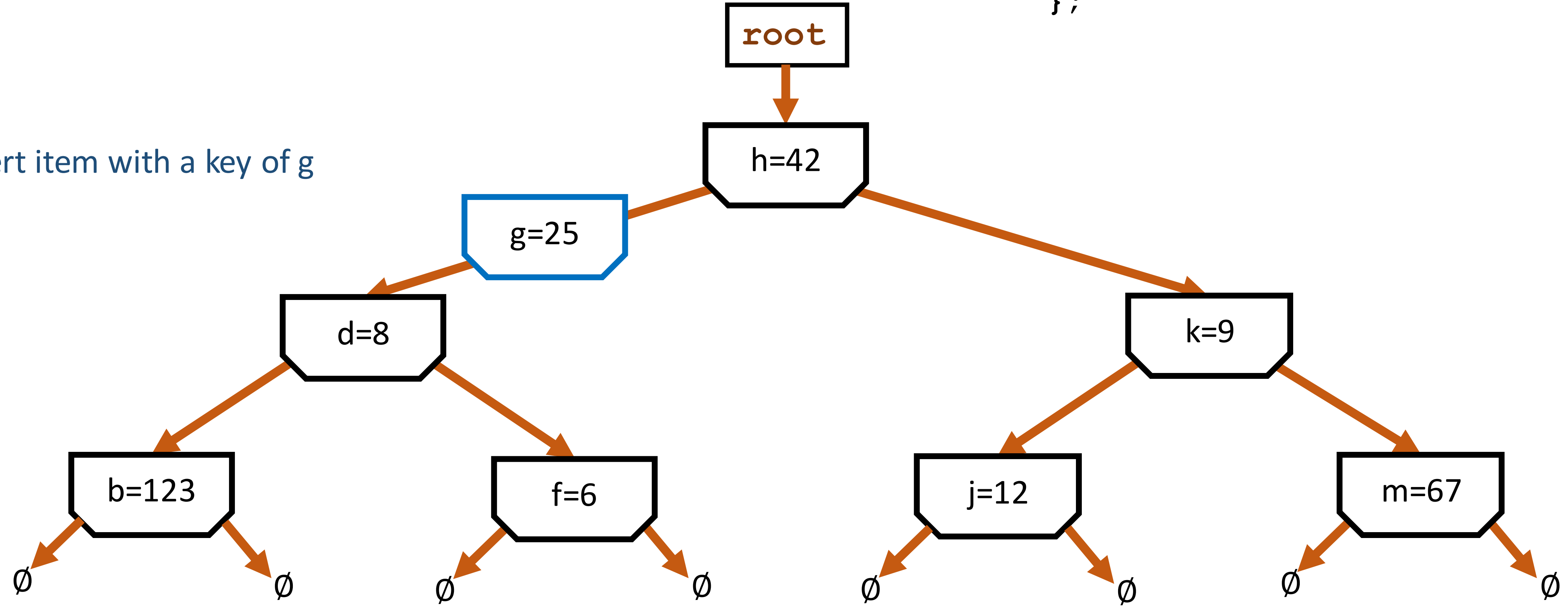
Insert item with a key of g



```
struct TreeMap {
   struct TreeMapEntry *__root;
   int __count;
   …
}
```

```
struct TreeMapEntry {
    char *key;   /* public */
    int value;   /* public */
    struct TreeMapEntry *__left;
    struct TreeMapEntry *__right;
};
```

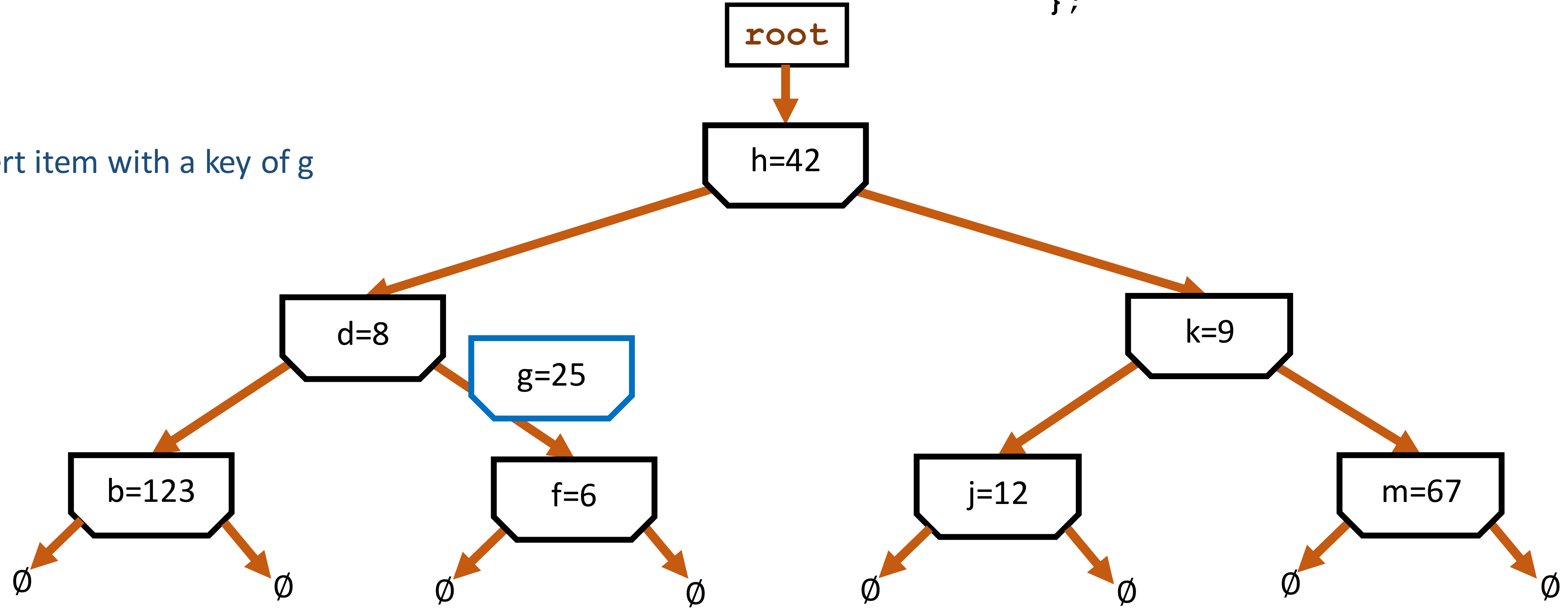
Insert item with a key of g



```
struct TreeMap {
    struct TreeMapEntry *__root;
    int __count;
    …
}
```

```
struct TreeMapEntry {
     char *key;   /* public */
     int value;   /* public */
     struct TreeMapEntry *__left;
     struct TreeMapEntry *__right;
};
```

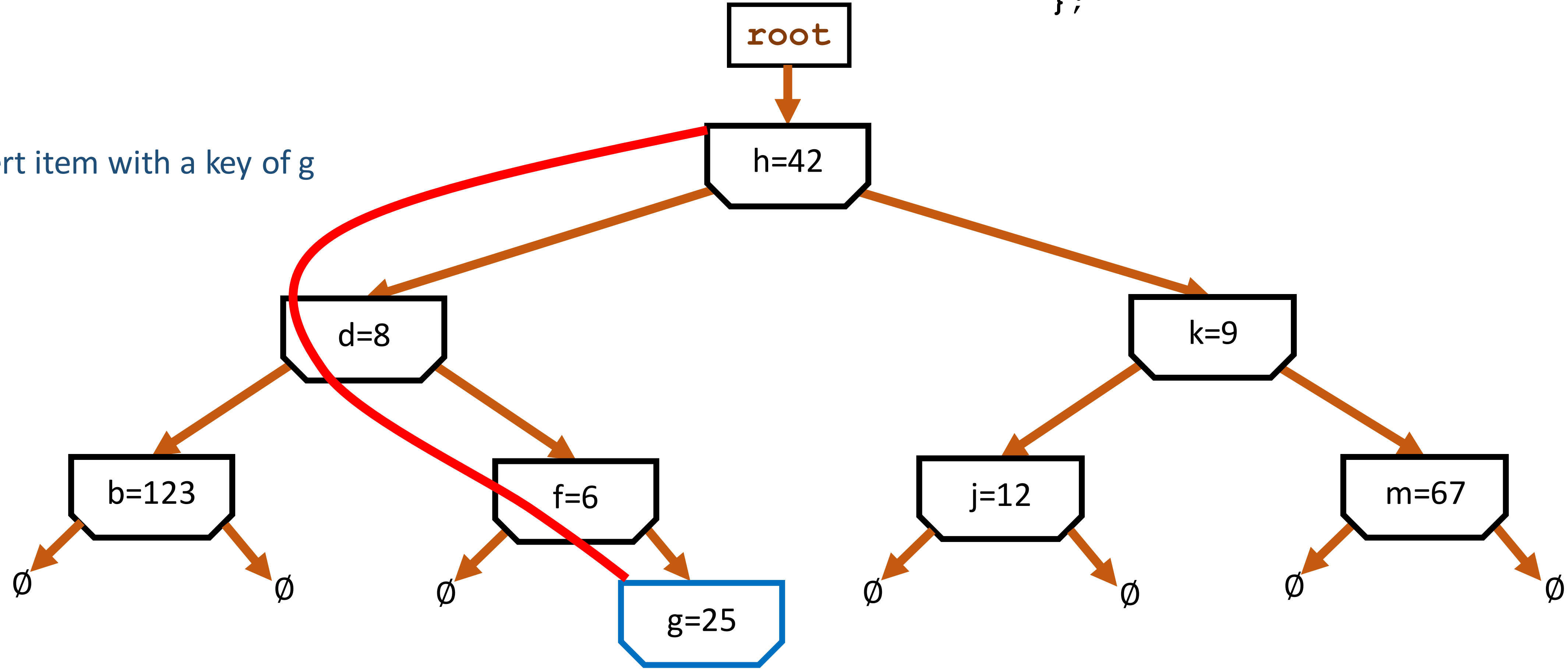
Insert item with a key of g



```
struct TreeMap {
    struct TreeMapEntry *__root;
    int __count;
    …
}
```

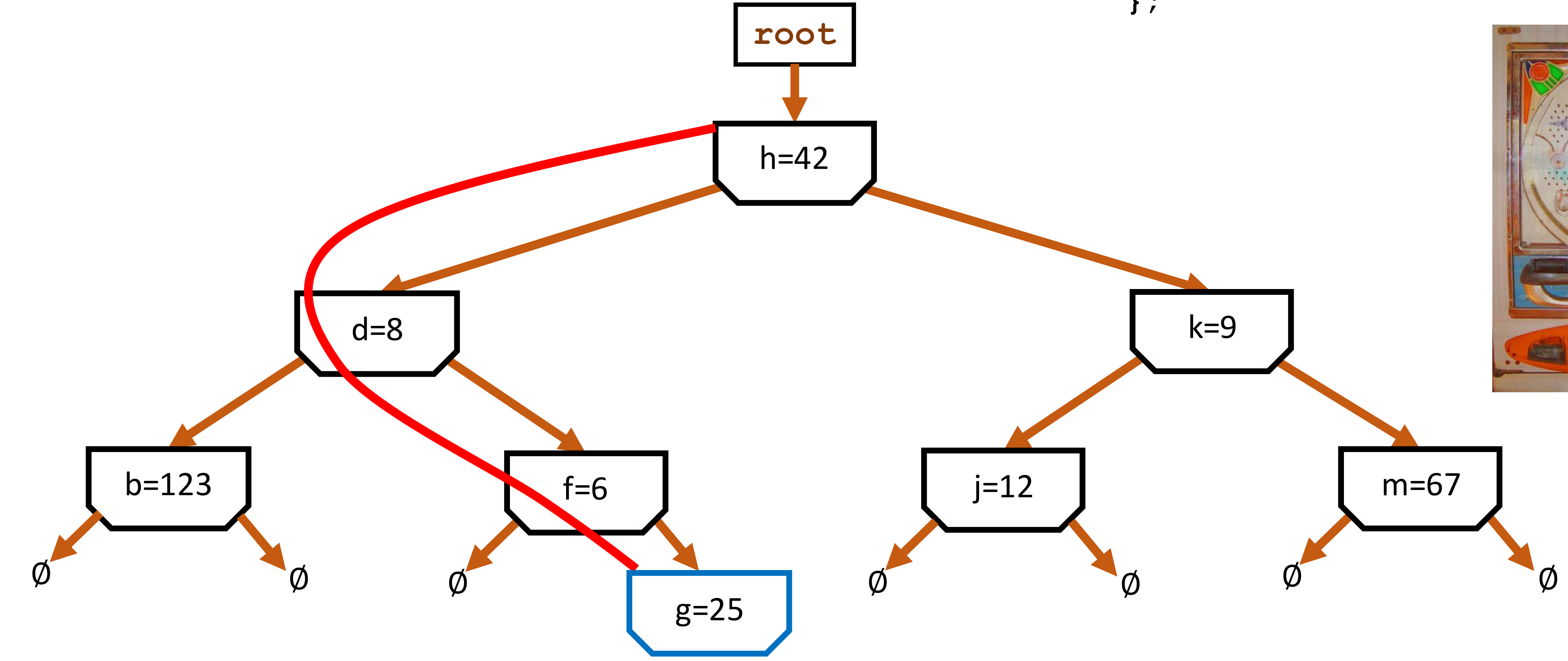
```
struct TreeMapEntry {
     char *key;   /* public */
     int value;   /* public */
     struct TreeMapEntry *__left;
     struct TreeMapEntry *__right;
};
```

Insert item with a key of g

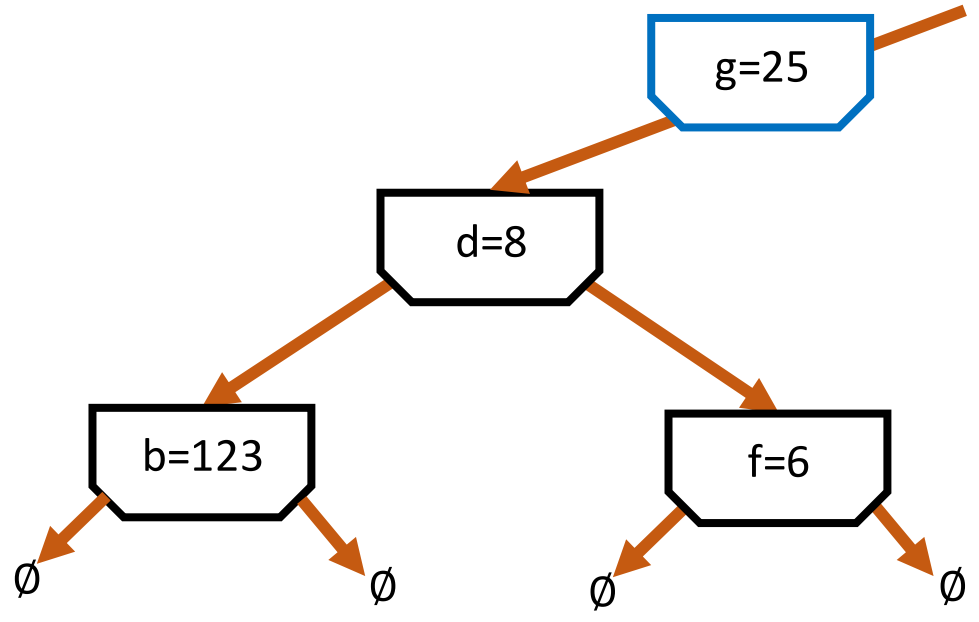


```
struct TreeMap {
    struct TreeMapEntry *__root;
    int __count;
    …
}
```

```
struct TreeMapEntry {
    char *key;   /* public */
    int value;   /* public */
    struct TreeMapEntry *__left;
    struct TreeMapEntry *__right;
};
```



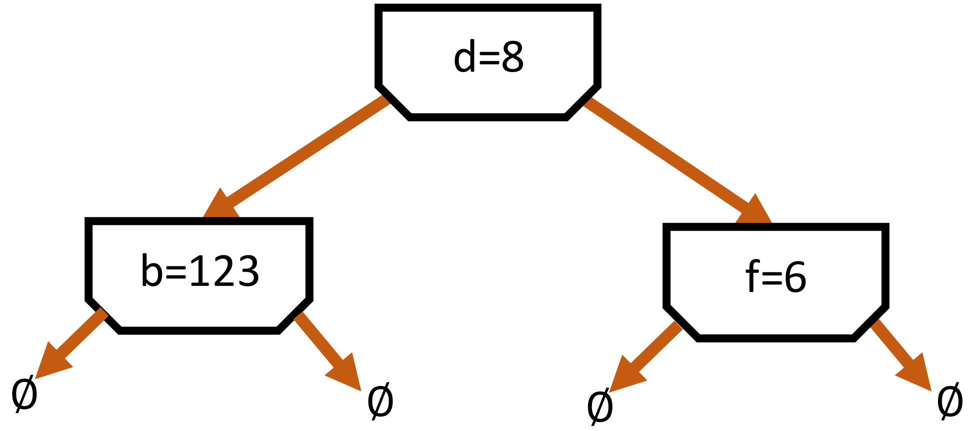
```
void __TreeMap_put(struct TreeMap* self, char *key, int value) {

    struct TreeMapEntry *cur, *left, *right;
    int cmp;

    cur = self->__root;
    while(cur != NULL ) {
        cmp = strcmp(key, cur->key);
        if(cmp == 0 ) {
            cur->value = value;
            return;
        }
        if( cmp < 0 ) {
            cur = cur->__left;
        } else {
            cur = cur->__right;
        }
    }

    ...
}
```



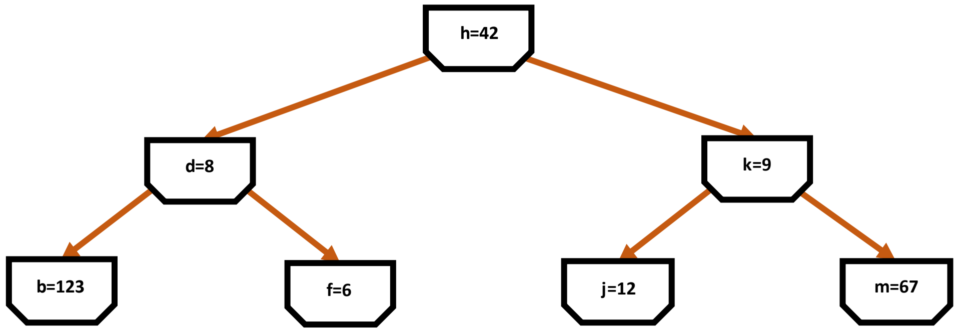
As long as the tree is correctly maintained you will always find either a match, or a place to correctly insert the new record. The tree is not guaranteed to be balanced - that depends on the insert order.

```
int main(void)
{
    struct TreeMap * map = TreeMap_new();

    map->put(map, "h", 22);
    map->put(map, "h", 42);  // Replace
    map->put(map, "d", 8);
    map->put(map, "b", 123);
    map->put(map, "f", 6);
    map->dump(map);

    map->put(map, "k", 9);
    map->put(map, "m", 67);
    map->put(map, "j", 12);
    map->dump(map);
}
```

```
Object TreeMap@0x6000003dcd20 count=4
h=42
| d=8
| | b=123
| | f=6

Object TreeMap@0x6000003dcd20 count=7
h=42
| d=8
| | b=123
| | f=6
| k=9
| | j=12
| | m=67
```

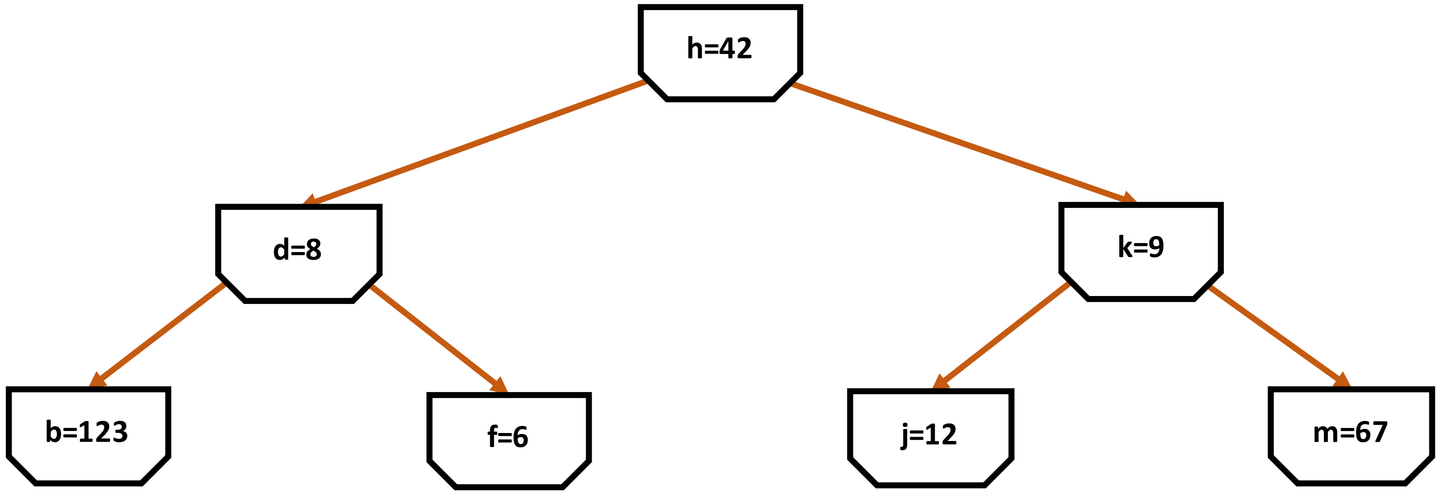


```c
void __TreeMap_dump_tree(struct TreeMapEntry *cur, int depth)
{
    if ( cur == NULL ) return;
    for(int i=0;i<depth;i++) printf("| ");
    printf("%s=%d\n", cur->key, cur->value);
    if ( cur->__left != NULL ) {
        __TreeMap_dump_tree(cur->__left, depth+1);
    }
    if ( cur->__right != NULL ) {
        __TreeMap_dump_tree(cur->__right, depth+1);
    }
}

...

    // Recursively print the tree view
    __TreeMap_dump_tree(self->__root, 0);
```



```
Object TreeMap@0x6000003dcd20 count=4
h=42
| d=8
| | b=123
| | f=6

Object TreeMap@0x6000003dcd20 count=7
h=42
| d=8
| | b=123
| | f=6
| k=9
| | j=12
| | m=67
```

```
int __TreeMap_get(struct TreeMap* self, char *key, int def)
{
    int cmp;
    struct TreeMapEntry *cur;

    if ( key == NULL || self->__root == NULL ) return def;

    cur = self->__root;
    while(cur != NULL ) {
        cmp = strcmp(key, cur->key);
        if(cmp == 0 ) return cur->value;
        else if(cmp < 0 ) cur = cur->__left;
        else cur = cur->__right;

    }
    return def;
}
```

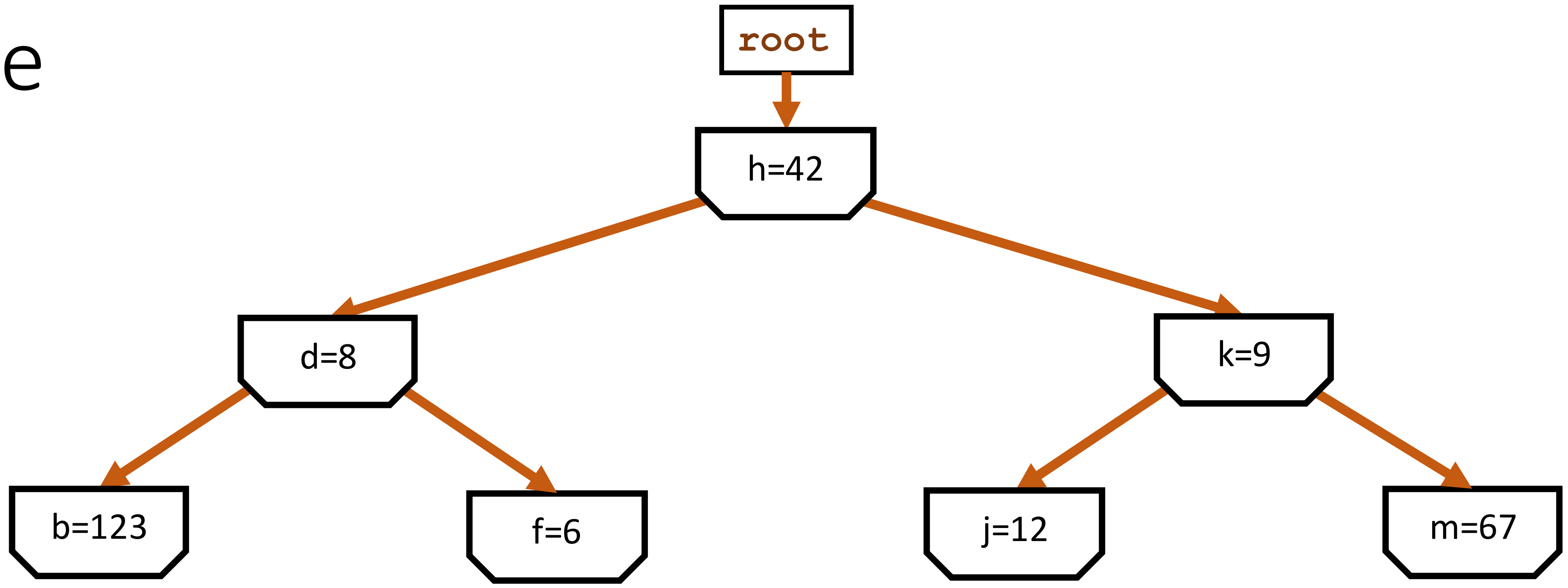


Cannot easily build an Iterator for a pure Tree

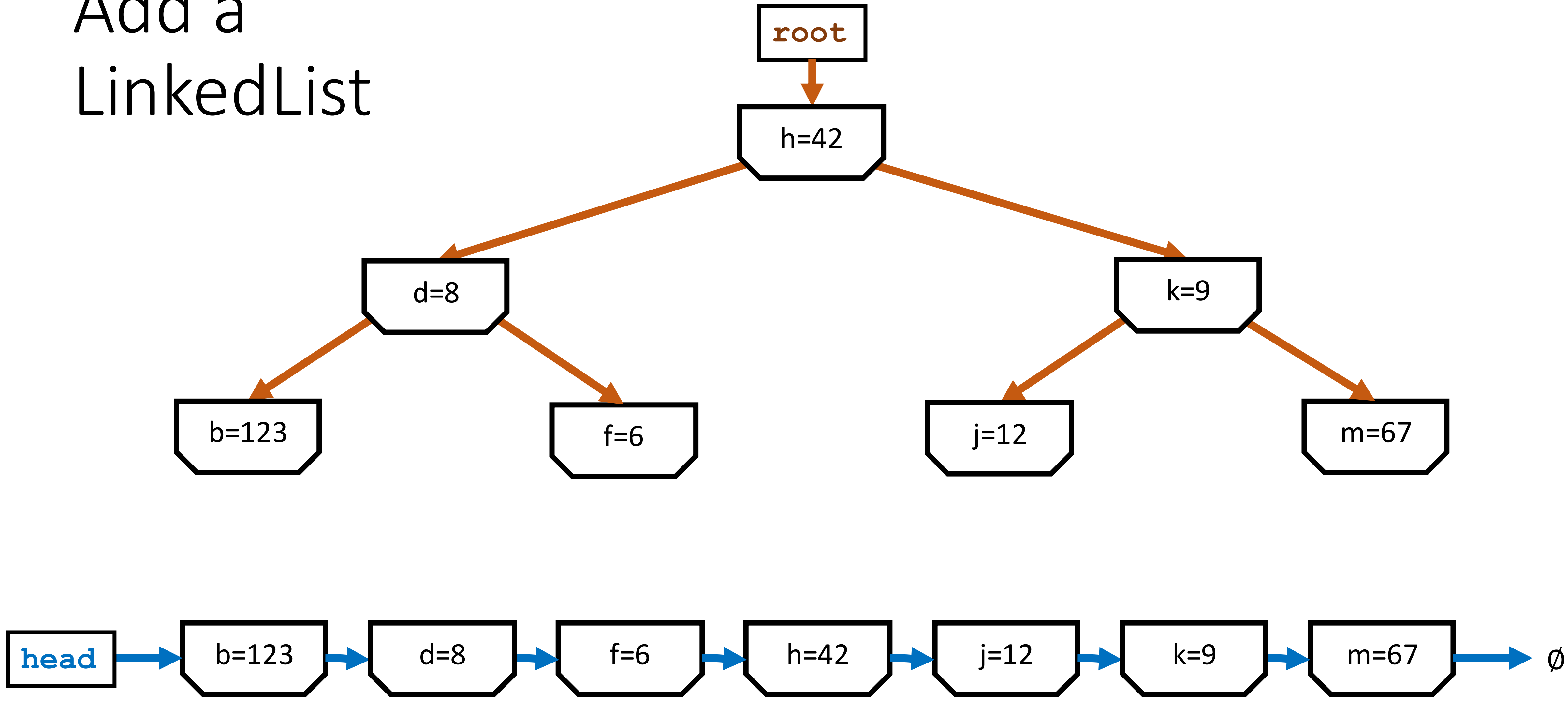
- ListMap can support an ordered iterator
- HashMap can support an unordered iterator
- A TreeMap cannot support an iterator without building a stack within the iterator

```
printf("\nIterate\n");
iter = map->iter(map);
while(1) {
    cur = iter->next(iter);
    if ( cur == NULL ) break;
    printf(" %s=%d\n", cur->key, cur->value);
}
iter->del(iter);
```

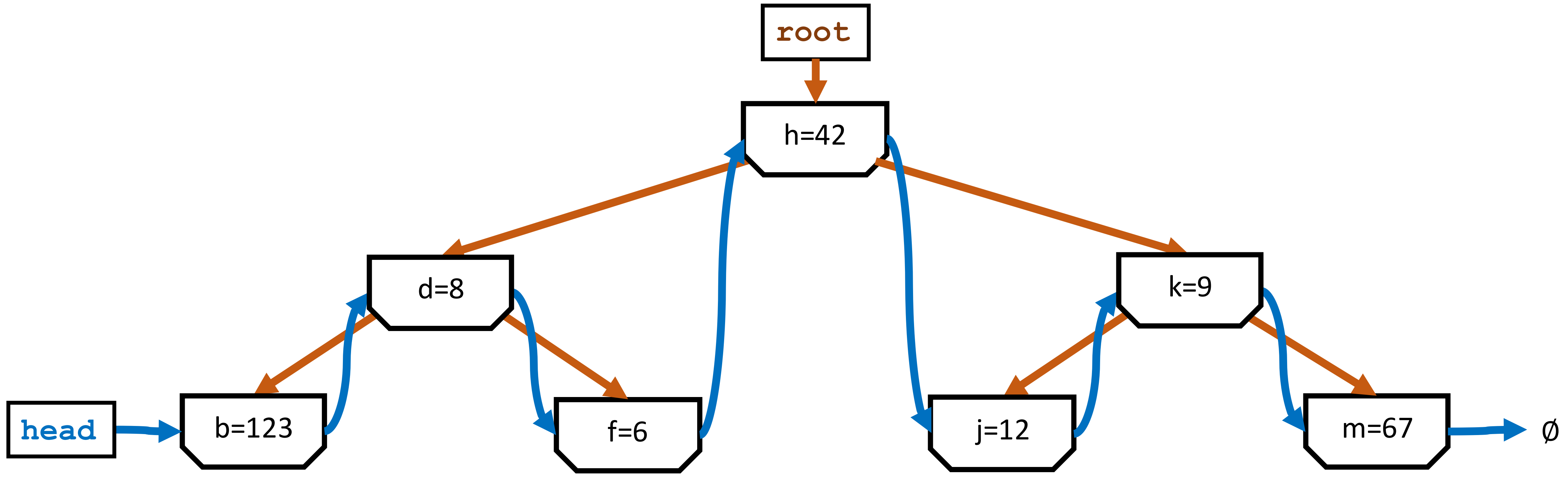
Tree



Add a LinkedList

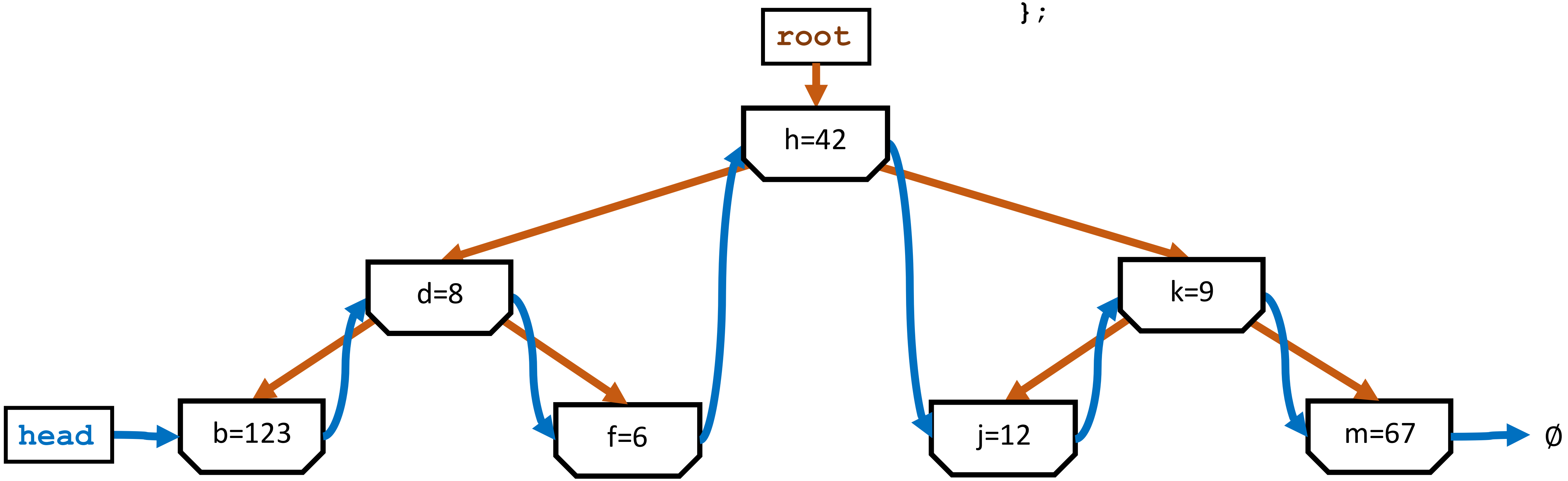


Linked TreeMap



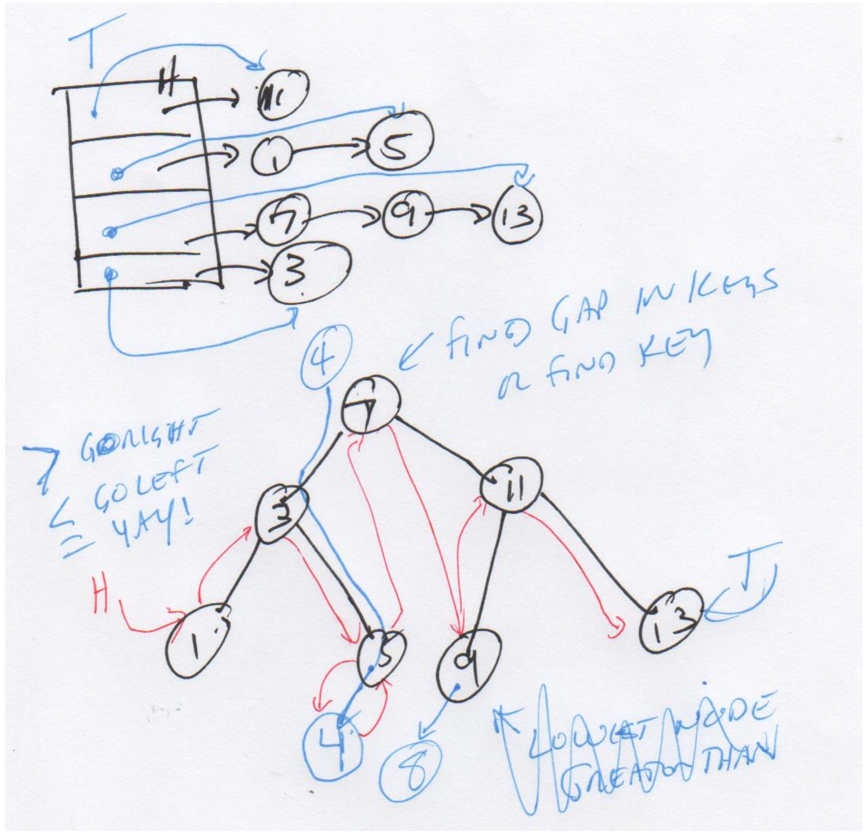
```
struct TreeMap {
    struct TreeMapEntry *__head;

    struct TreeMapEntry *__root;
    …
}
```

```
struct TreeMapEntry {
    char *key;
    int value;
    struct TreeMapEntry *__next;

    struct TreeMapEntry *__left;
    struct TreeMapEntry *__right;
};
```



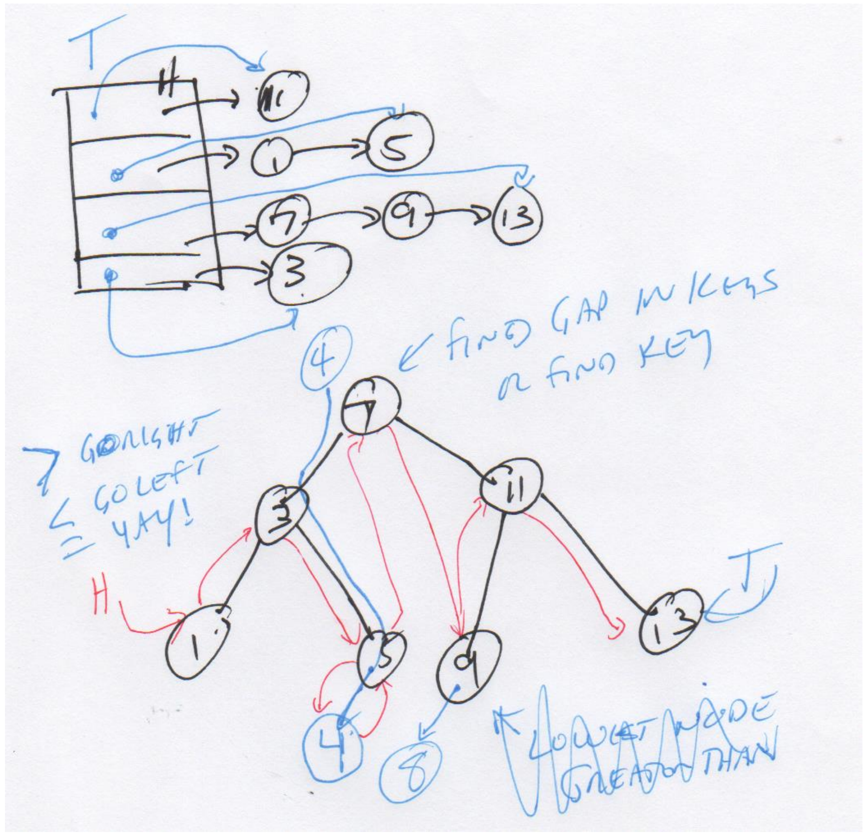
Making put() work

Updating two data structures at the same time



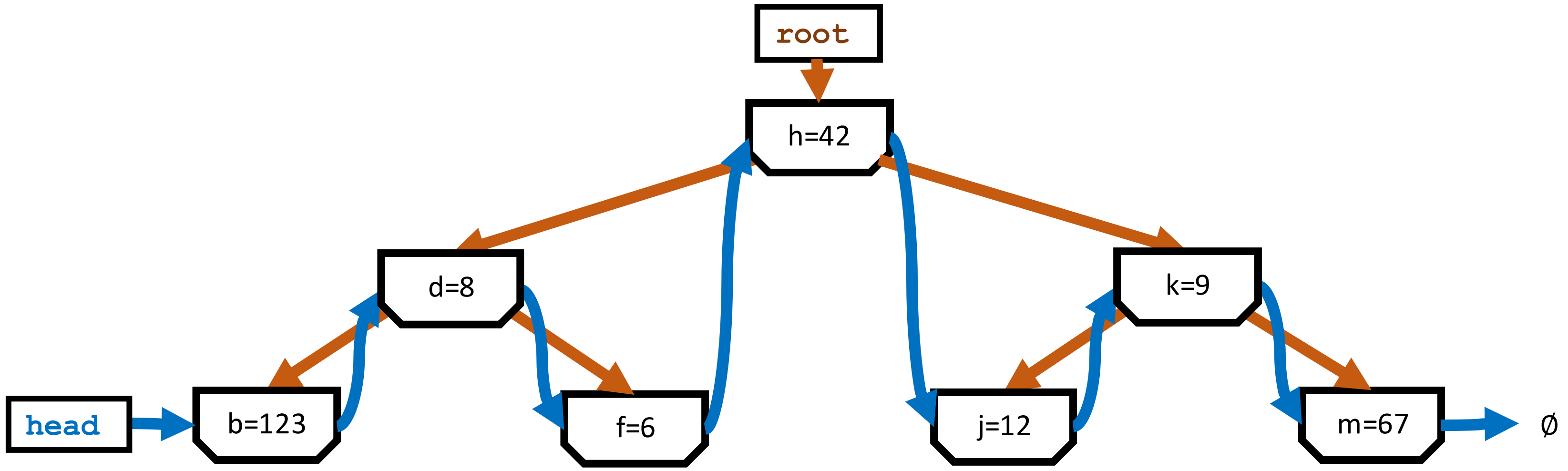
Before we start

- It likely impossible to write put() using pieces from Google searches or a long conversation with an AI bot
- You need to understand it all before you write it
- Once you understand it – the code should look clean and simple to you
- My put() implementation was broken until it was perfect
- I threw it away completely several times



Performance of put()

- A binary tree search is O(log N) while a sorted list search is O(N)
  - Million entry list search average: 500,000
  - Million entry tree search average: ($\log_2$ 1000000) = 20
- We use the Tree to find where to insert or update a key / value pair



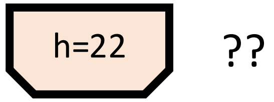
Use cases for put()

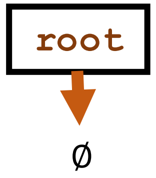
- Inserting into an empty (easy)
- Inserting into a right gap
- Inserting into a left gap
- Inserting at the beginning
- Inserting at the end
- Replacing an entry (easy)

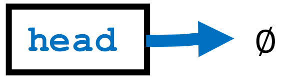
```
struct TreeMapEntry {
    char *key;
    int value;
    struct TreeMapEntry *__next;

    struct TreeMapEntry *__left;
    struct TreeMapEntry *__right;
};

struct TreeMap {
   struct TreeMapEntry *__head;

   struct TreeMapEntry *__root;
   …
}


struct TreeMap * TreeMap_new() {
    struct TreeMap *p = malloc(sizeof(*p));
    p->__head = NULL;
    p->__root = NULL;
   return p;
}
```

Empty list (1 of 2)

```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    // Scan to see if the entry is in the list
    // If the entry is found - update it

    // Entry not found - create and fill
    new = malloc(sizeof(*new));
         ...

    // If list is empty - just connect it
    if ( self->__head == NULL ) {
        self->__head = new;
        self->__root = new;
        return;
    }
    ...
}

...
     map->put(map, "h", 22);
```

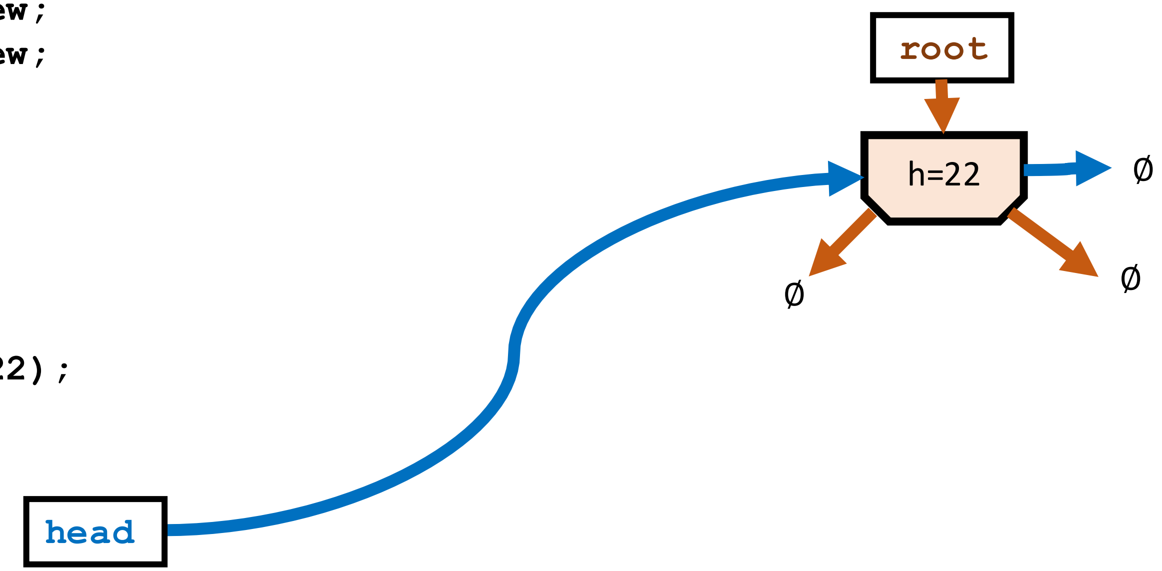






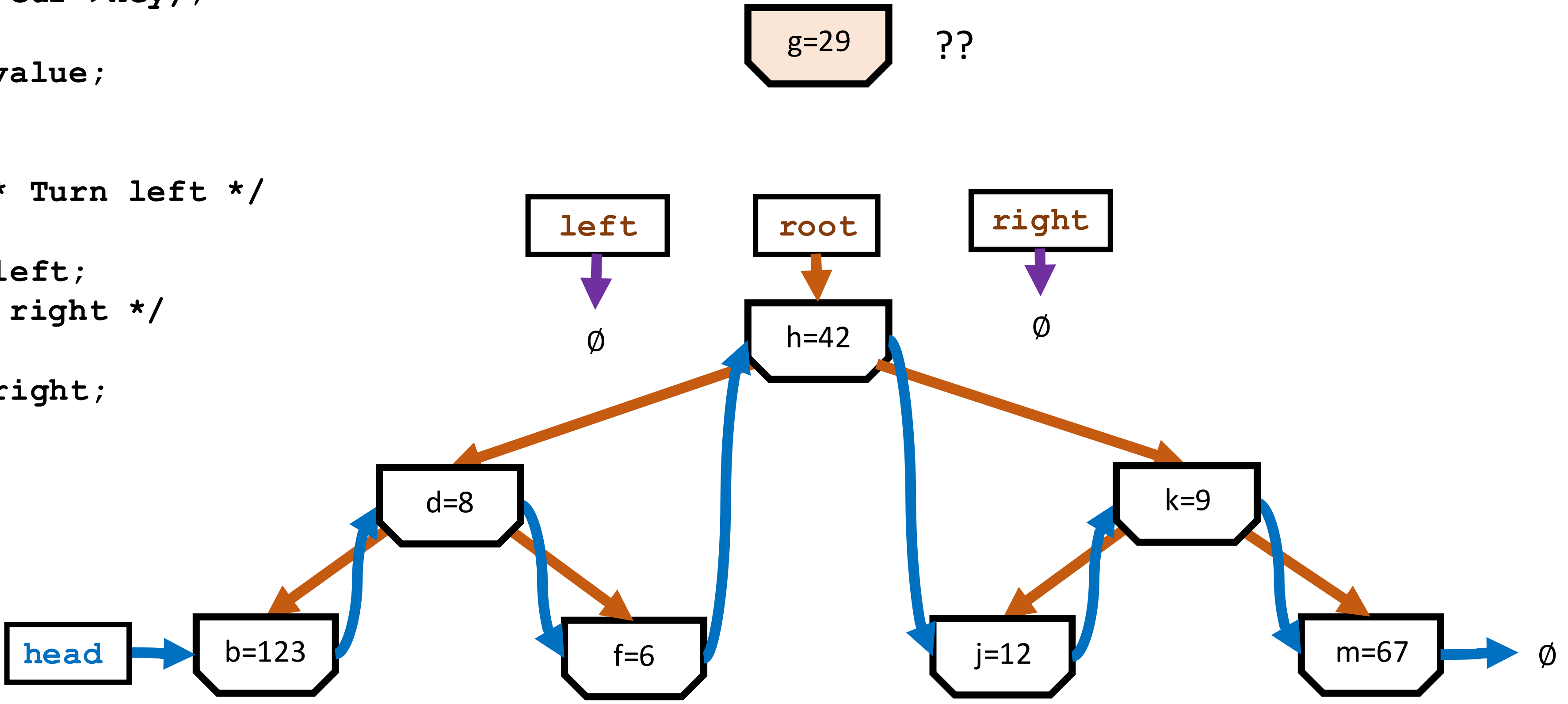
Empty list (2 of 2)

```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    // Scan to see if the entry is in the list
    // If the entry is found - update it

    // Entry not found - create and fill
    new = malloc(sizeof(*new));
         ...

    // If list is empty - just connect it
    if ( self->__head == NULL ) {
        self->__head = new;
        self->__root = new;
        return;
    }
    ...
}

...
     map->put(map, "h", 22);
```



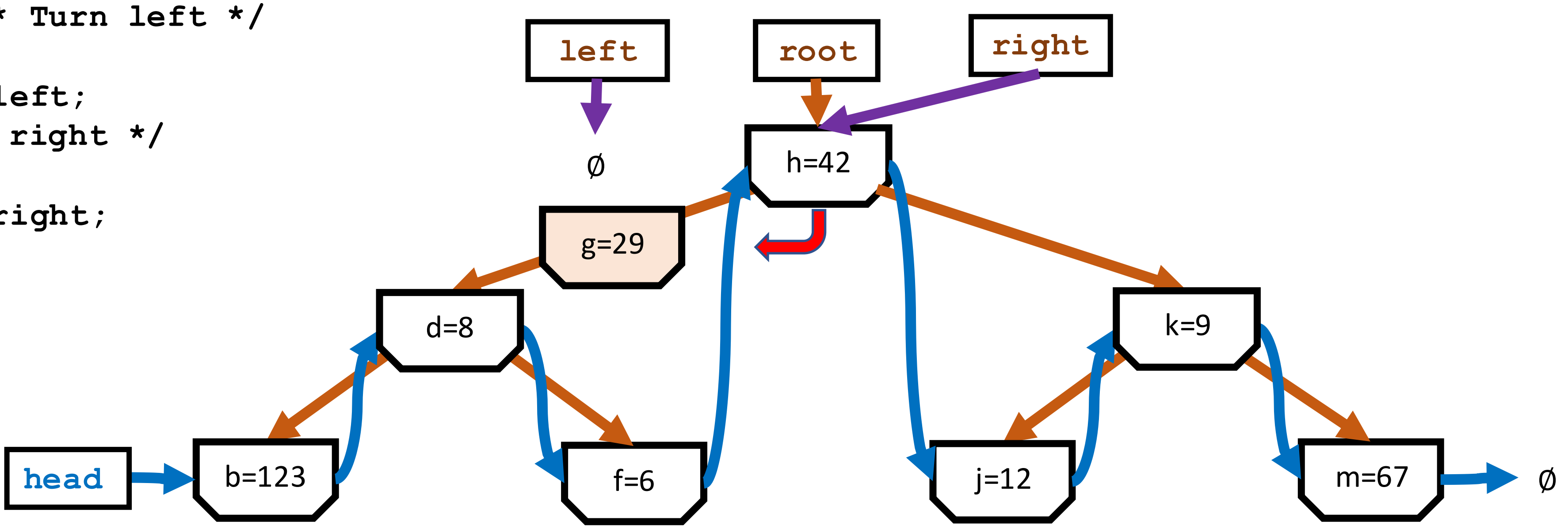
Finding an Item or Gap (1 of 3)

```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    cur = self->__root;
    right = NULL;  /* Our nearest right neighbor */
    left = NULL;   /* Out nearest left neighbor */
    while(cur != NULL ) {
        cmp = strcmp(key, cur->key);
        if(cmp == 0 ) {
            cur->value = value;
            return;
        }
        if( cmp < 0 ) { /* Turn left */
            right = cur;
            cur = cur->__left;
        } else {  /* Turn right */
            left = cur;
            cur = cur->__right;
        }
    }

    ...
}
```



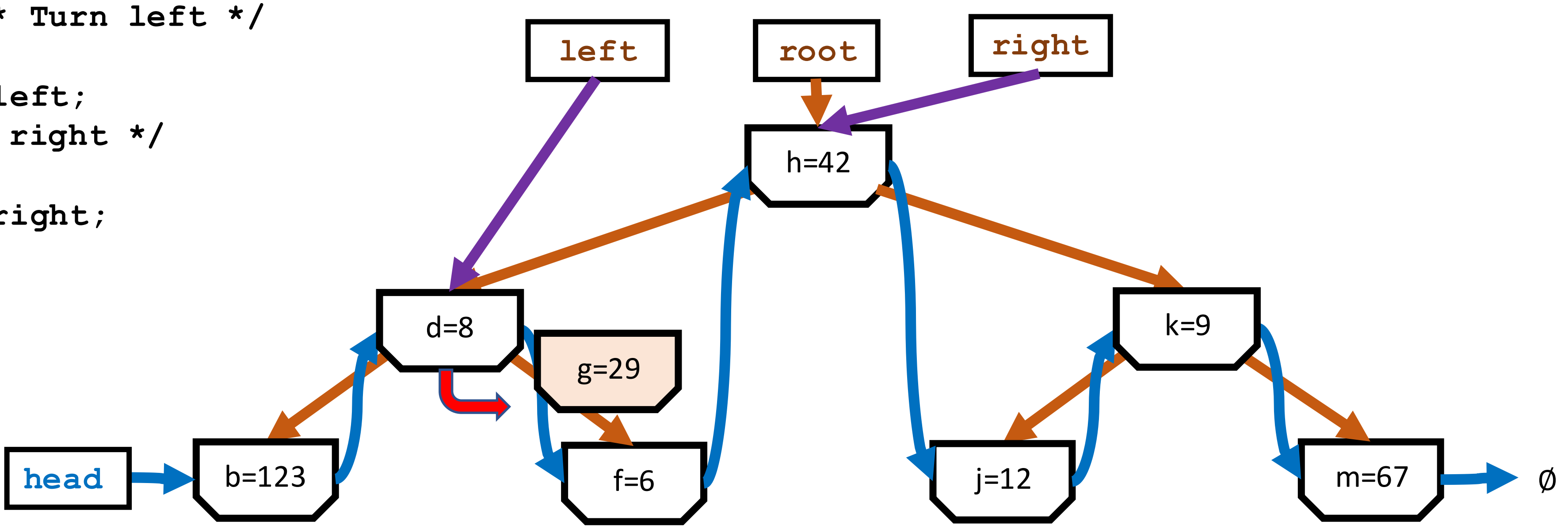
Finding an Item or Gap (2 of 3)

```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    cur = self->__root;
    right = NULL;  /* Our nearest right neighbor */
    left = NULL;   /* Out nearest left neighbor */
    while(cur != NULL ) {
        cmp = strcmp(key, cur->key);
        if(cmp == 0 ) {
            cur->value = value;
            return;
        }
        if( cmp < 0 ) { /* Turn left */
            right = cur;
            cur = cur->__left;
        } else {  /* Turn right */
            left = cur;
            cur = cur->__right;
        }
    }

    ...
}
```



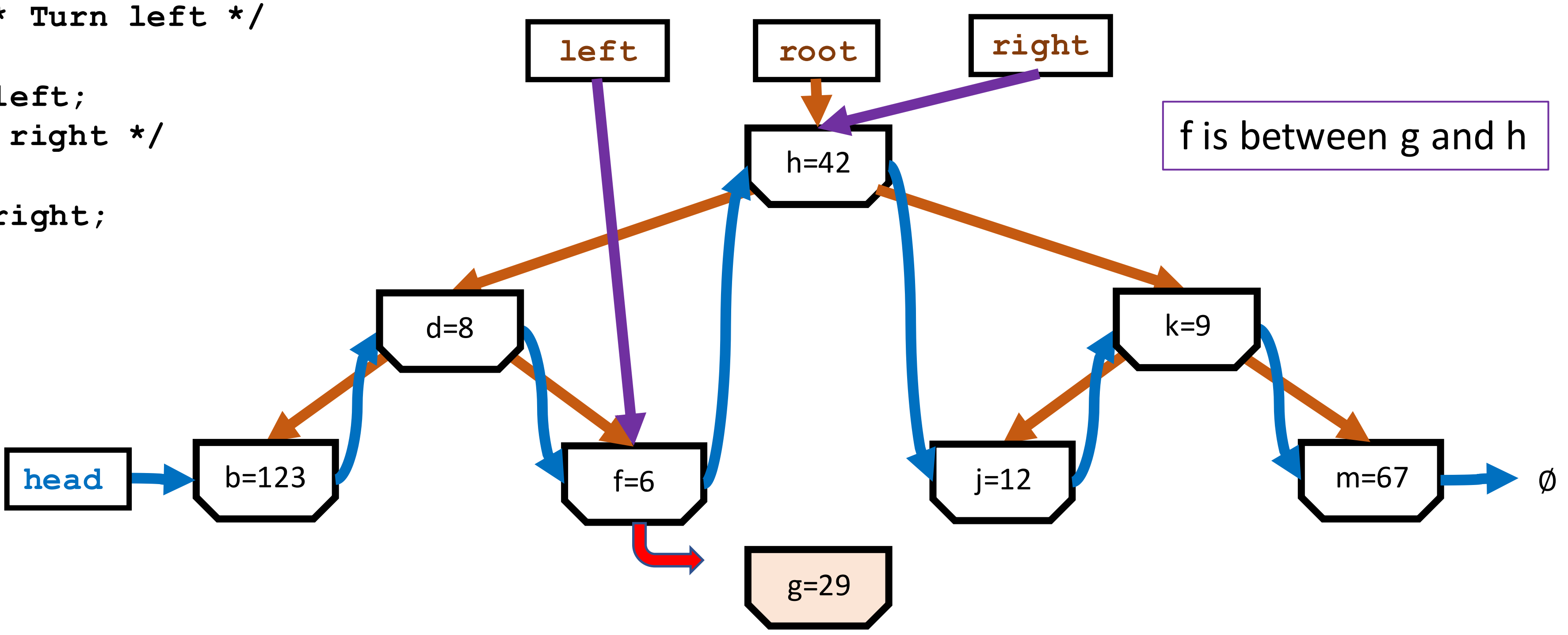
Finding an Item or Gap (3 of 3)

```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    cur = self->__root;
    right = NULL;  /* Our nearest right neighbor */
    left = NULL;   /* Out nearest left neighbor */
    while(cur != NULL ) {
        cmp = strcmp(key, cur->key);
        if(cmp == 0 ) {
            cur->value = value;
            return;
        }
        if( cmp < 0 ) { /* Turn left */
            right = cur;
            cur = cur->__left;
        } else {  /* Turn right */
            left = cur;
            cur = cur->__right;
        }
    }

    ...
}
```



```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    cur = self->__root;
    right = NULL;  /* Our nearest right neighbor */
    left = NULL;   /* Out nearest left neighbor */
    while(cur != NULL ) {
        cmp = strcmp(key, cur->key);
        if(cmp == 0 ) {
            cur->value = value;
            return;
        }
        if( cmp < 0 ) { /* Turn left */
            right = cur;
            cur = cur->__left;
        } else {  /* Turn right */
            left = cur;
            cur = cur->__right;
        }
    }

    ...
}
```

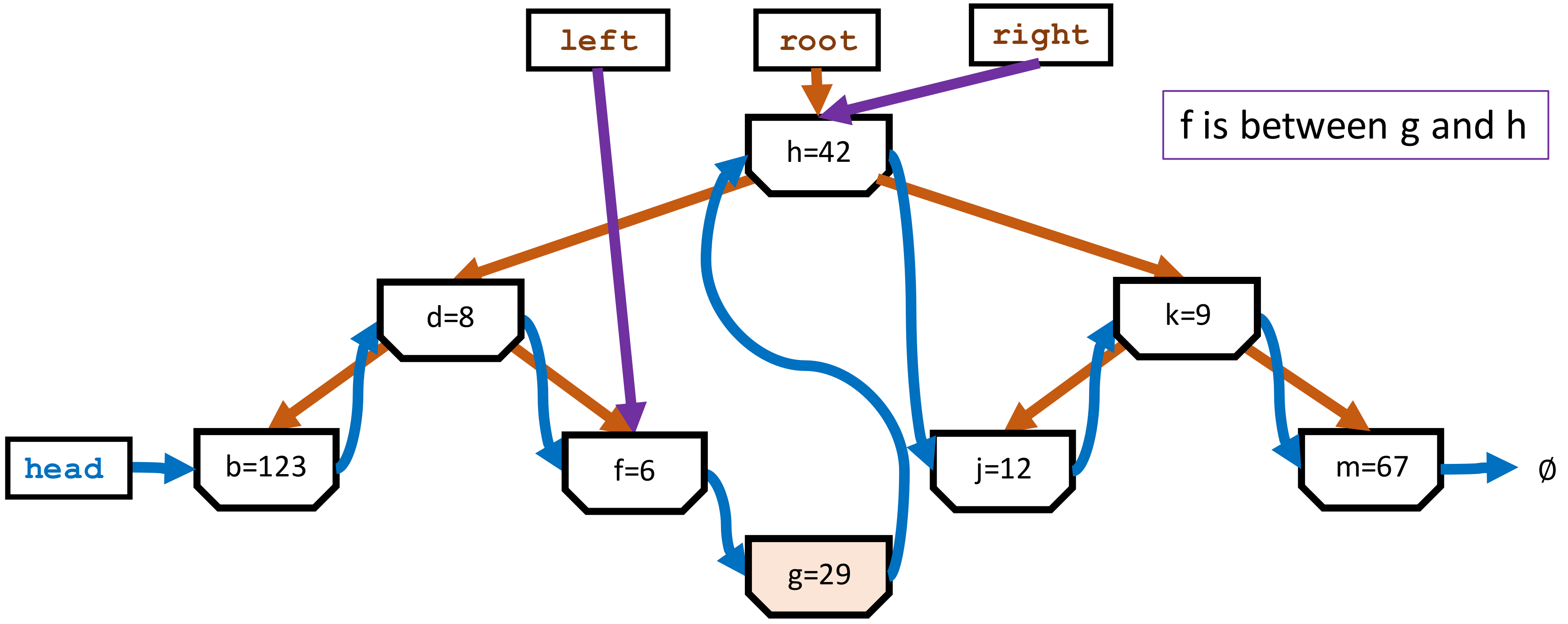
Found a gap to the right of a node



f is between g and h

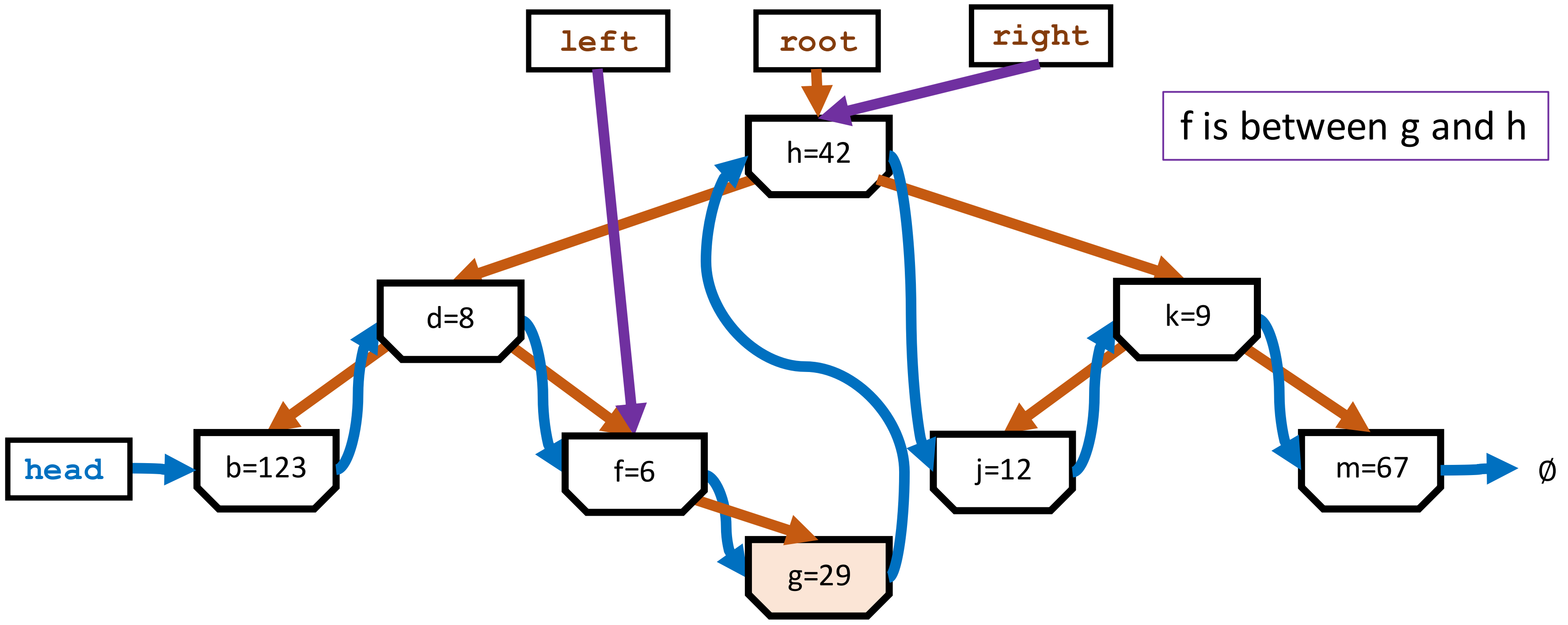
Insert into the Linked List

```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    ...
    // Insert into the sorted linked list
    if ( left != NULL ) {
        new->__next = right;
        left->__next = new;
    } else {
        new->__next = self->__head;
        self->__head = new;
    }
    ...
}
```



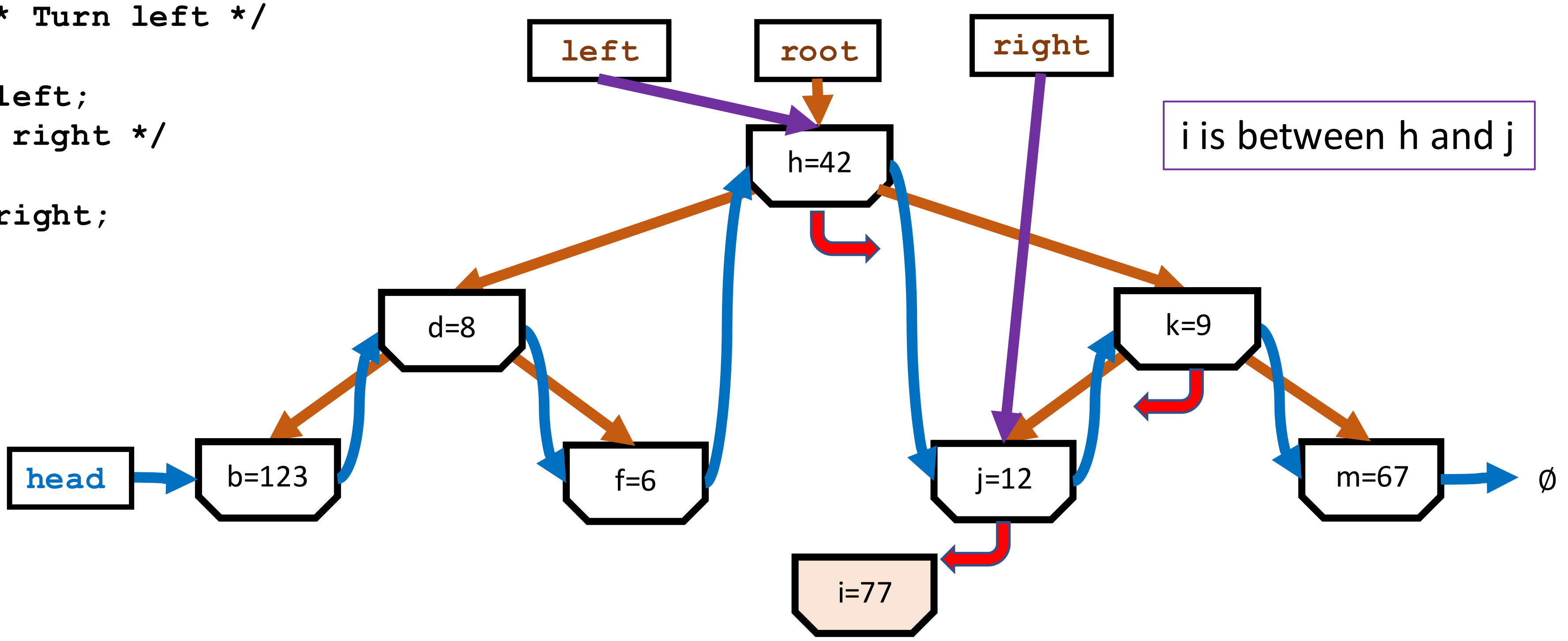
Insert into the Tree (right)

```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    ...
    // Insert into the tree
    if ( right != NULL && right->__left == NULL ) {
        right->__left = new;
    } else if ( left != NULL && left->__right == NULL ) {
        left->__right = new;
    } else {
        printf("FAIL\n");
    }    ...
}
```



```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    cur = self->__root;
    right = NULL;  /* Our nearest right neighbor */
    left = NULL;   /* Out nearest left neighbor */
    while(cur != NULL ) {
        cmp = strcmp(key, cur->key);
        if(cmp == 0 ) {
            cur->value = value;
            return;
        }
        if( cmp < 0 ) { /* Turn left */
            right = cur;
            cur = cur->__left;
        } else {  /* Turn right */
            left = cur;
            cur = cur->__right;
        }
    }

    ...
}
```

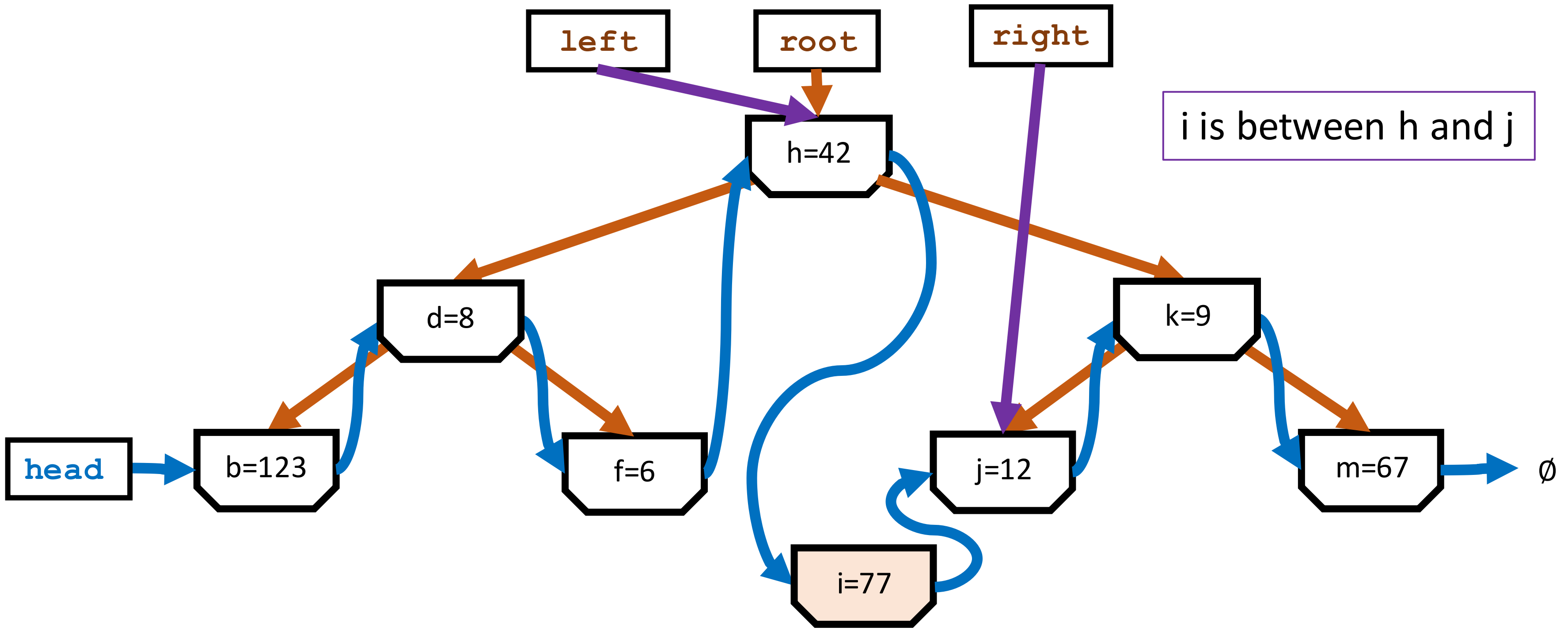
Finding an Item or Left Gap



i is between h and j

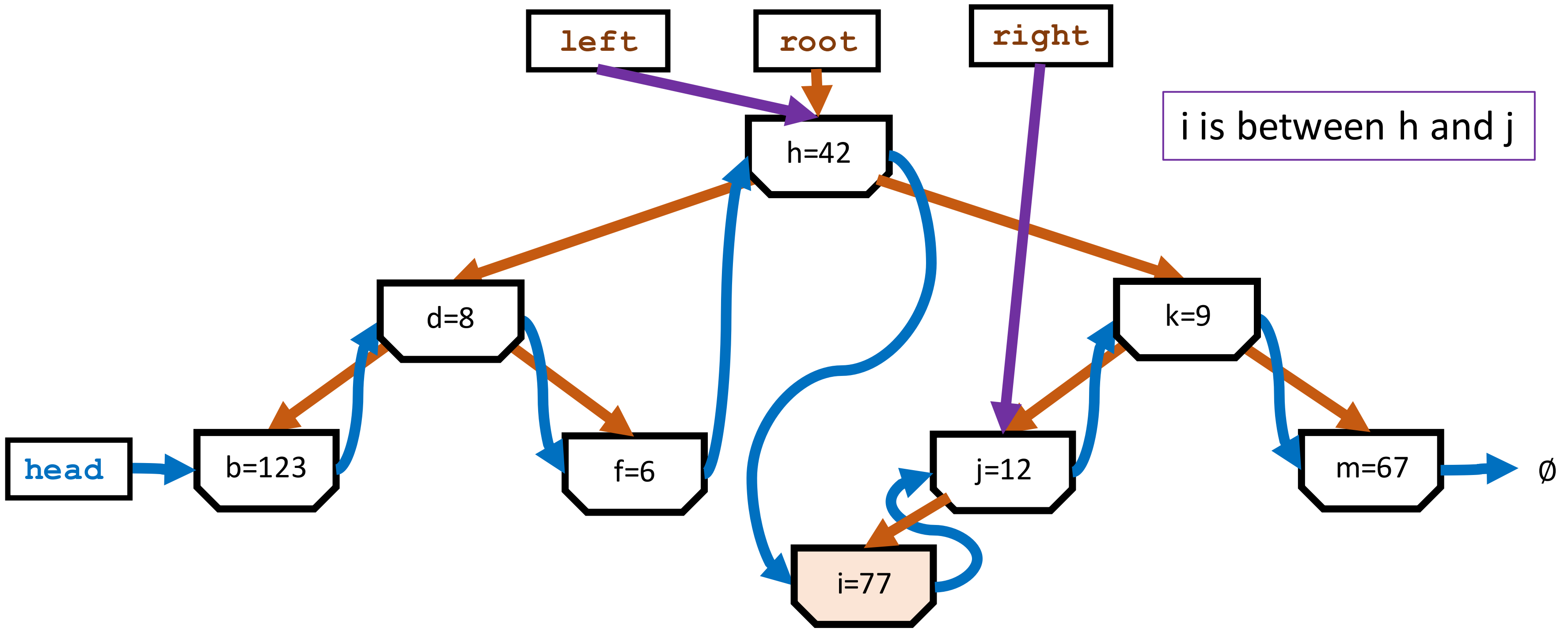
```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    ...
    // Insert into the sorted linked list
    if ( left != NULL ) {
        new->__next = right;
        left->__next = new;
    } else {
        new->__next = self->__head;
        self->__head = new;
    }
    ...
}
```

Insert into the Linked List



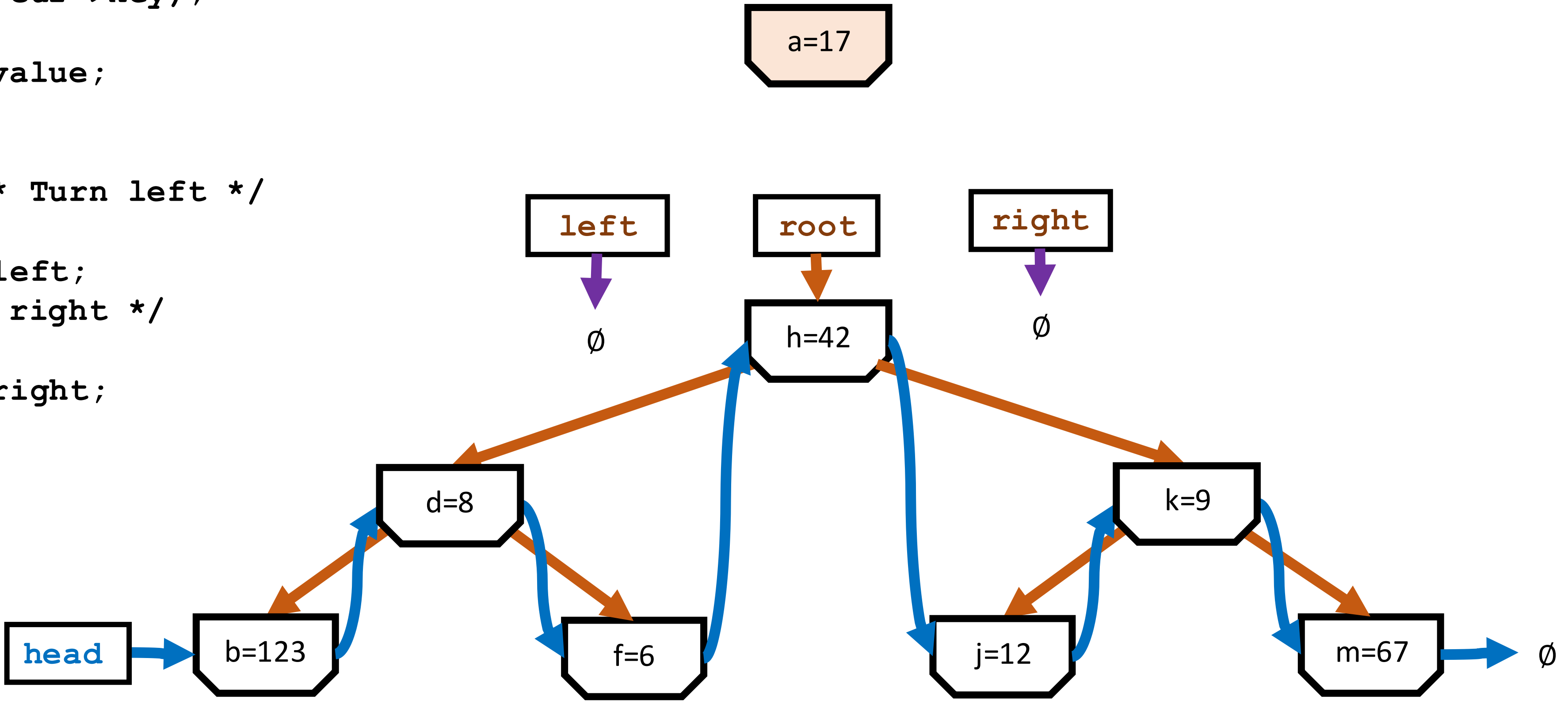
Insert into the Tree (left gap)

```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    ...
    // Insert into the tree
    if ( right != NULL && right->__left == NULL ) {
        right->__left = new;
    } else if ( left != NULL && left->__right == NULL ) {
        left->__right = new;
    } else {
        printf("FAIL\n");
    }    ...
}
```



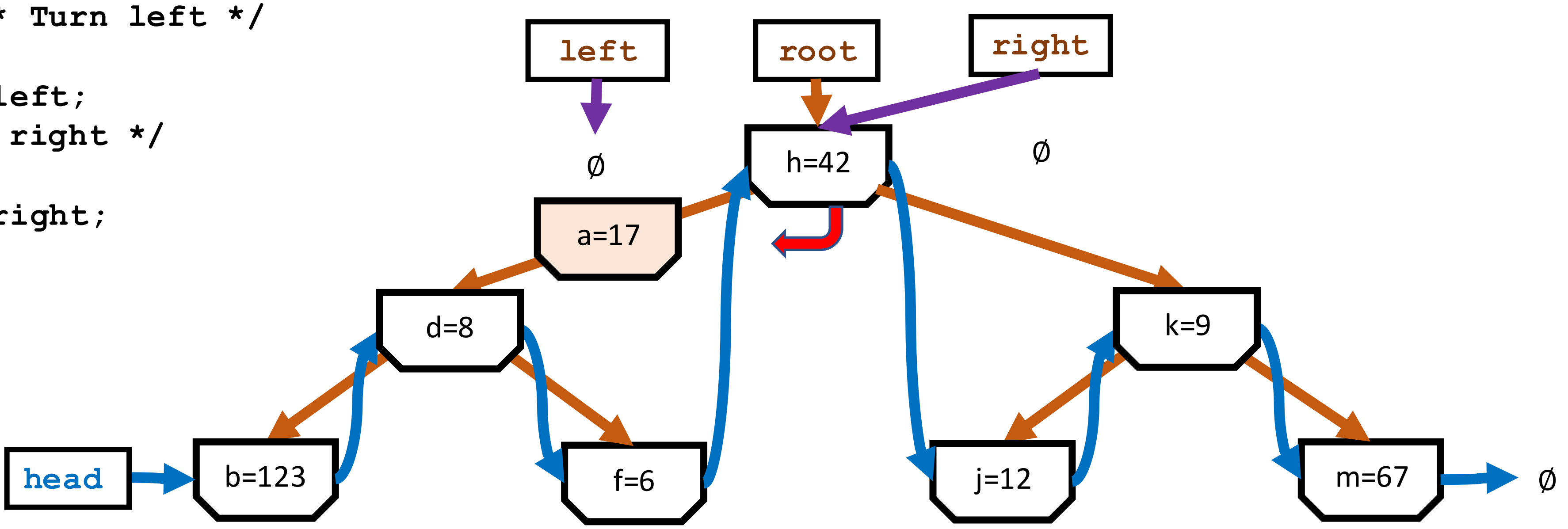
```
void __TreeMap_put(struct TreeMap* self,
    char *key, int value) {

     cur = self->__root;
     right = NULL;  /* Our nearest right neighbor */
     left = NULL;   /* Out nearest left neighbor */
     while(cur != NULL ) {
          cmp = strcmp(key, cur->key);
          if(cmp == 0 ) {
               cur->value = value;
               return;
          }
          if( cmp < 0 ) { /* Turn left */
               right = cur;
               cur = cur->__left;
          } else {  /* Turn right */
               left = cur;
               cur = cur->__right;
          }
     }

     ...
}
```

At the start...



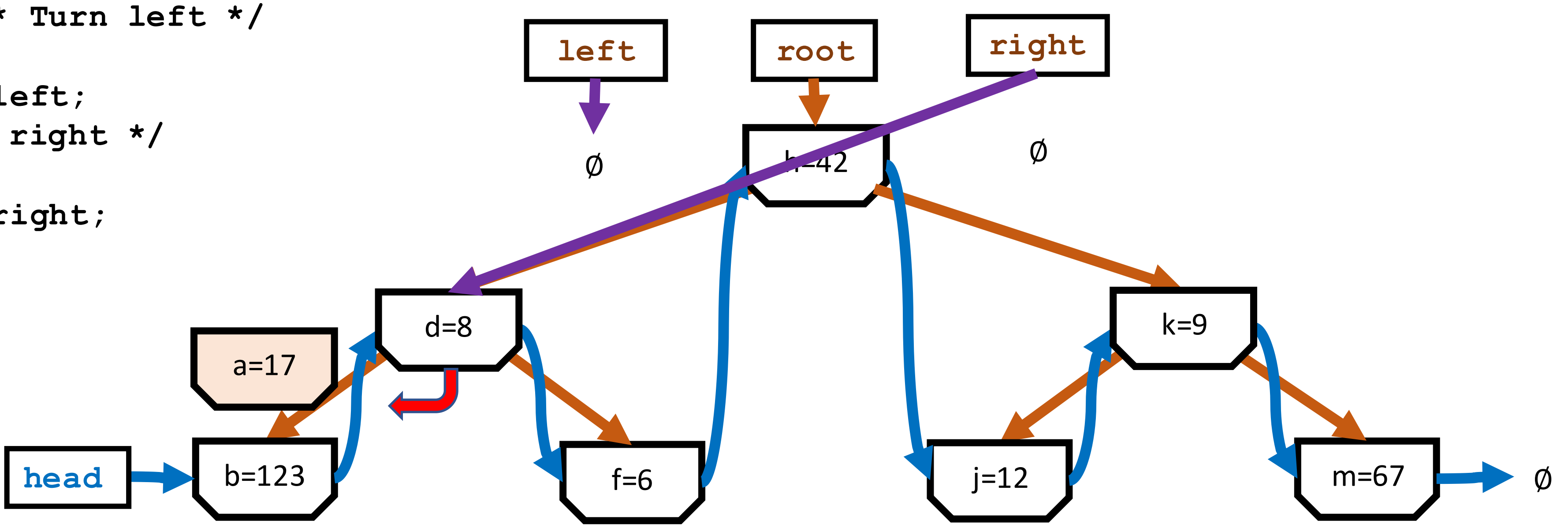
To the left...

```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    cur = self->__root;
    right = NULL;  /* Our nearest right neighbor */
    left = NULL;   /* Out nearest left neighbor */
    while(cur != NULL ) {
        cmp = strcmp(key, cur->key);
        if(cmp == 0 ) {
            cur->value = value;
            return;
        }
        if( cmp < 0 ) { /* Turn left */
            right = cur;
            cur = cur->__left;
        } else {  /* Turn right */
            left = cur;
            cur = cur->__right;
        }
    }

    ...
}
```



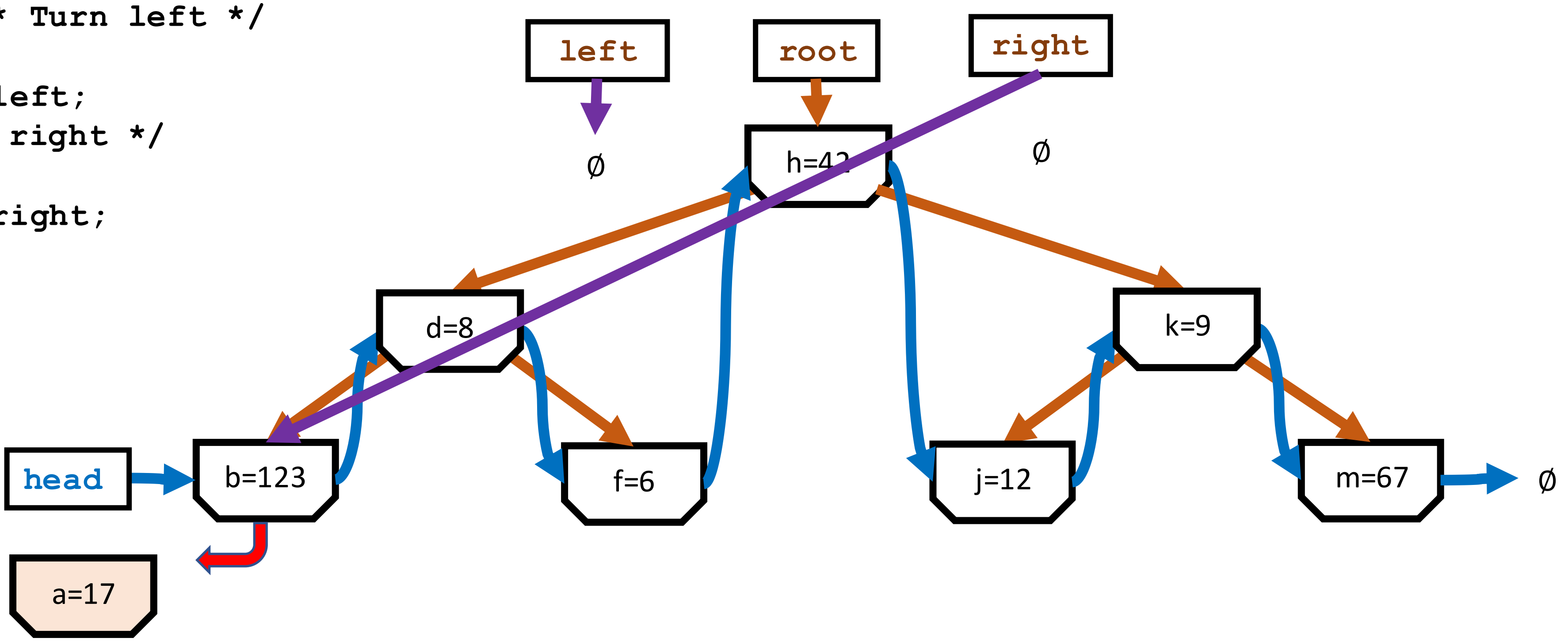
To the left... to the left...

```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    cur = self->__root;
    right = NULL;  /* Our nearest right neighbor */
    left = NULL;   /* Out nearest left neighbor */
    while(cur != NULL ) {
        cmp = strcmp(key, cur->key);
        if(cmp == 0 ) {
            cur->value = value;
            return;
        }
        if( cmp < 0 ) { /* Turn left */
            right = cur;
            cur = cur->__left;
        } else {  /* Turn right */
            left = cur;
            cur = cur->__right;
        }
    }

    ...
}
```



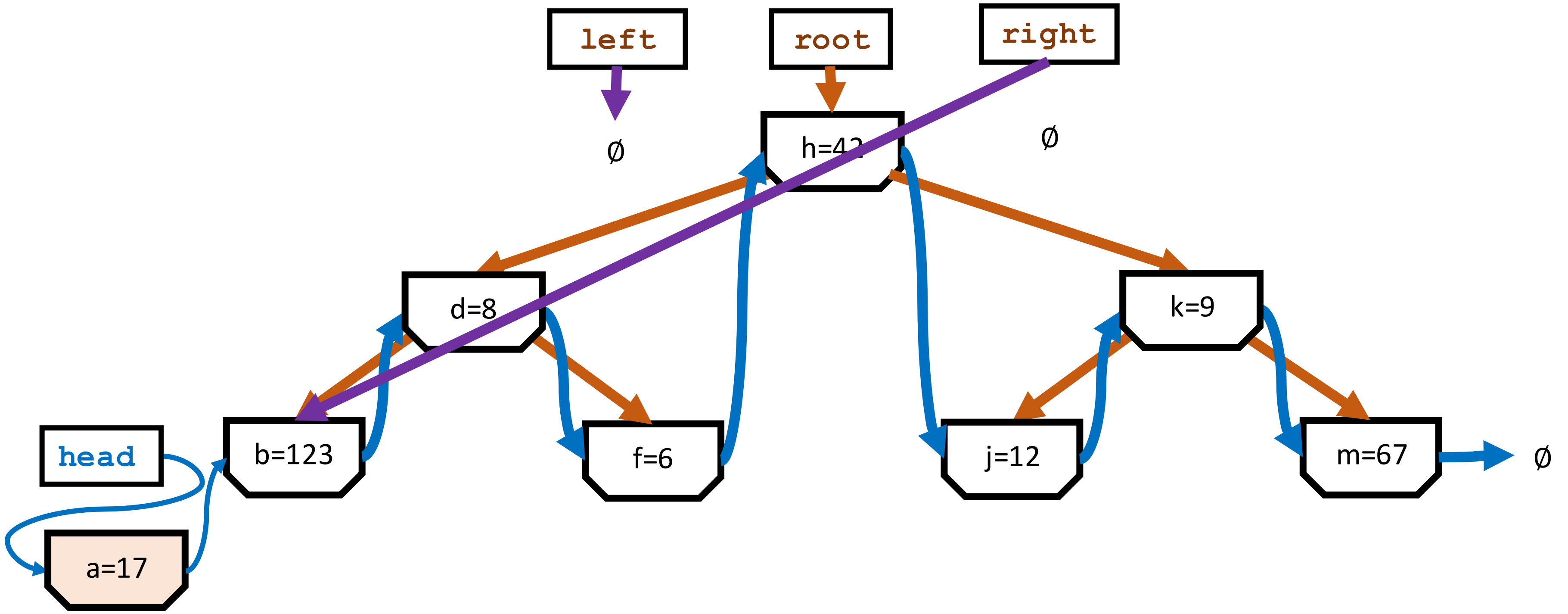
To the left...

```
void __TreeMap_put(struct TreeMap* self,
    char *key, int value) {

     cur = self->__root;
     right = NULL;  /* Our nearest right neighbor */
     left = NULL;   /* Out nearest left neighbor */
     while(cur != NULL ) {
         cmp = strcmp(key, cur->key);
         if(cmp == 0 ) {
             cur->value = value;
             return;
         }
         if( cmp < 0 ) { /* Turn left */
             right = cur;
             cur = cur->__left;
         } else {  /* Turn right */
             left = cur;
             cur = cur->__right;
         }
     }

     ...
}
```



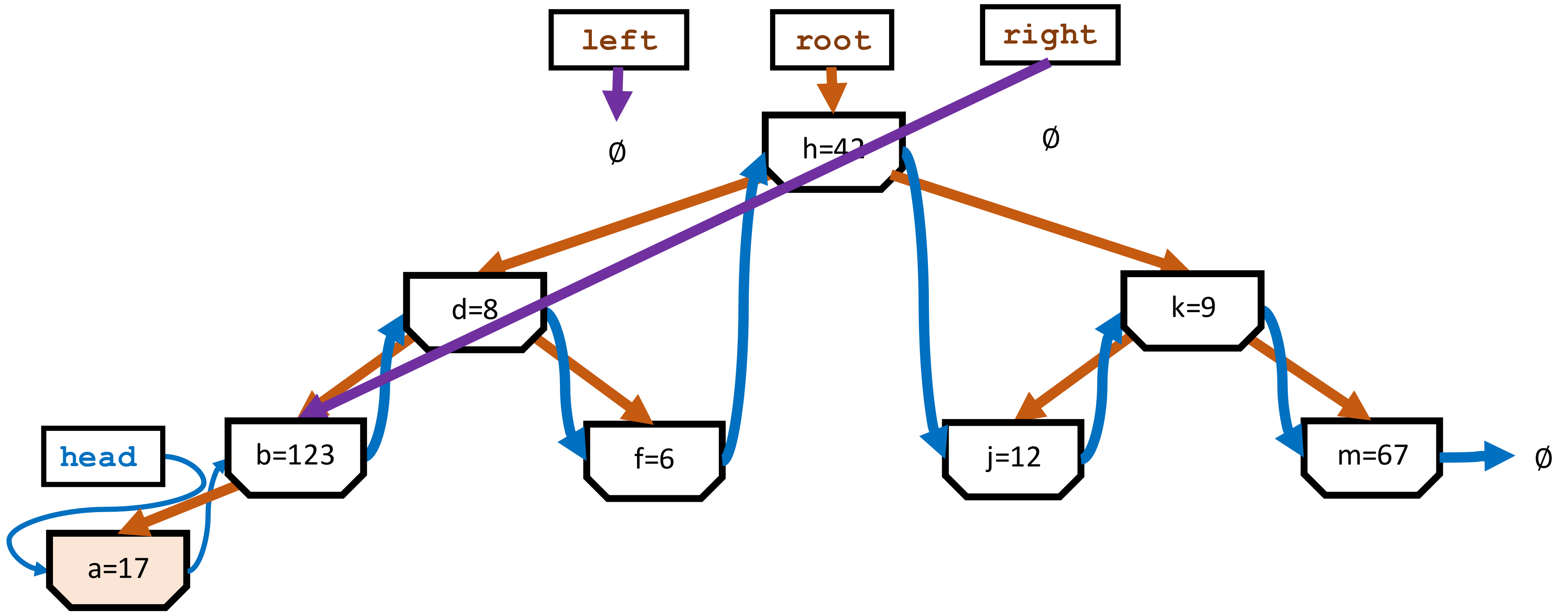
Insert into the linked list

```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    ...
    // Insert into the sorted linked list
    if ( left != NULL ) {
        new->__next = right;
        left->__next = new;
    } else {
        new->__next = self->__head;
        self->__head = new;
    }
    ...
}
```



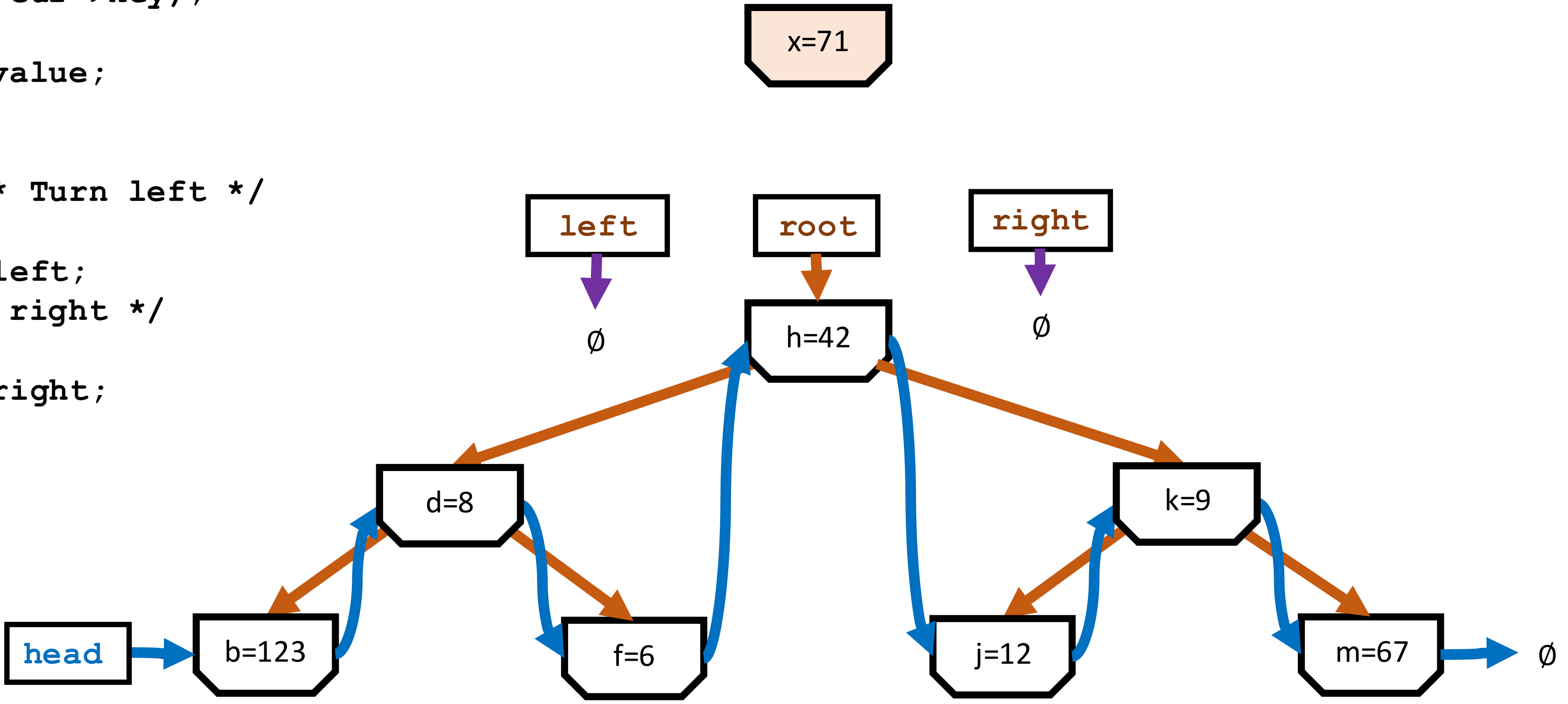
Insert into the tree

```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    ...
    // Insert into the tree
    if ( right != NULL && right->__left == NULL ) {
        right->__left = new;
    } else if ( left != NULL && left->__right == NULL ) {
        left->__right = new;
    } else {
        printf("FAIL\n");
    }
    ...
}
```



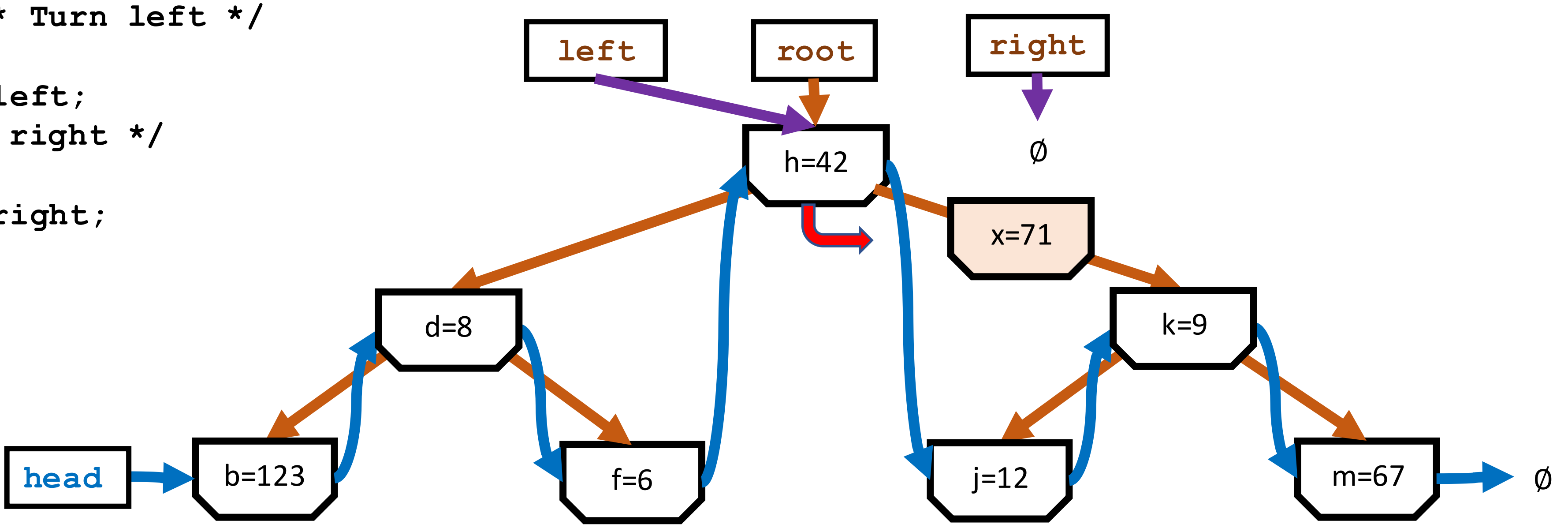
At the end…

```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    cur = self->__root;
    right = NULL;  /* Our nearest right neighbor */
    left = NULL;   /* Out nearest left neighbor */
    while(cur != NULL ) {
        cmp = strcmp(key, cur->key);
        if(cmp == 0 ) {
            cur->value = value;
            return;
        }
        if( cmp < 0 ) { /* Turn left */
            right = cur;
            cur = cur->__left;
        } else {  /* Turn right */
            left = cur;
            cur = cur->__right;
        }
    }

    ...
}
```



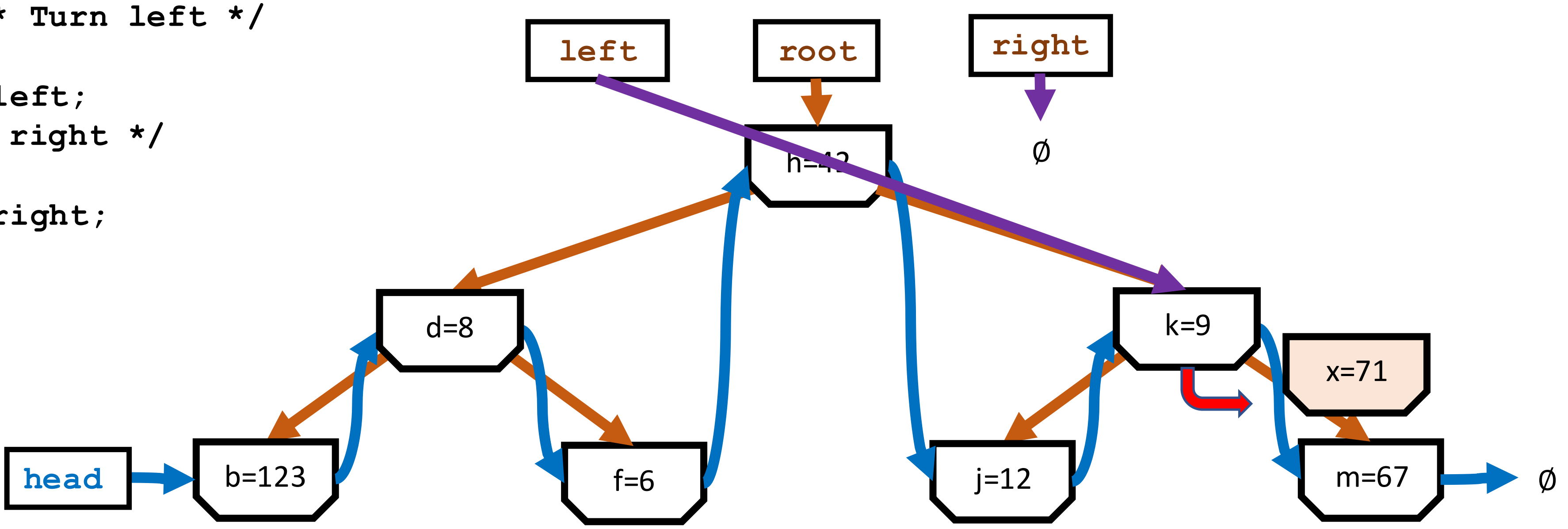
```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    cur = self->__root;
    right = NULL;  /* Our nearest right neighbor */
    left = NULL;   /* Out nearest left neighbor */
    while(cur != NULL ) {
        cmp = strcmp(key, cur->key);
        if(cmp == 0 ) {
            cur->value = value;
            return;
        }
        if( cmp < 0 ) { /* Turn left */
            right = cur;
            cur = cur->__left;
        } else {  /* Turn right */
            left = cur;
            cur = cur->__right;
        }
    }

    ...
}
```

To the right..



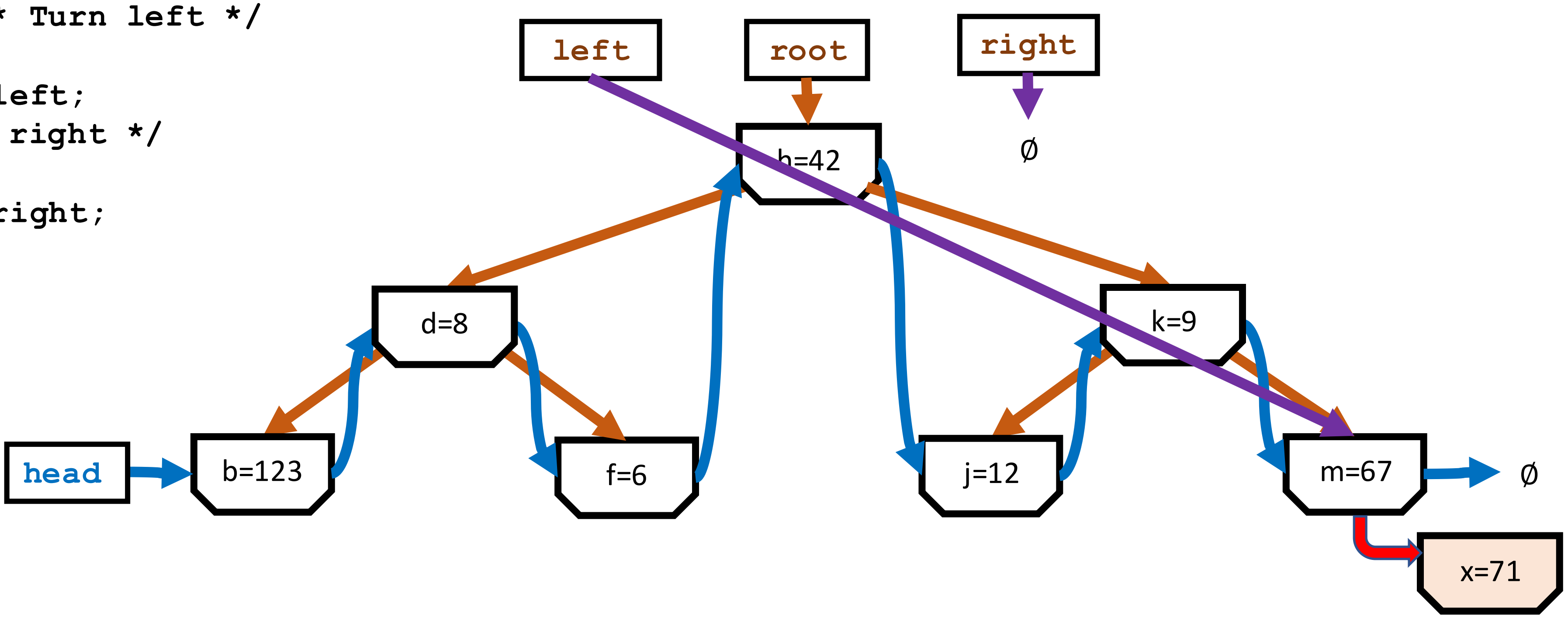
```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    cur = self->__root;
    right = NULL;  /* Our nearest right neighbor */
    left = NULL;   /* Out nearest left neighbor */
    while(cur != NULL ) {
        cmp = strcmp(key, cur->key);
        if(cmp == 0 ) {
            cur->value = value;
            return;
        }
        if( cmp < 0 ) { /* Turn left */
            right = cur;
            cur = cur->__left;
        } else {  /* Turn right */
            left = cur;
            cur = cur->__right;
        }
    }

    ...
}
```

To the right..
To the right..



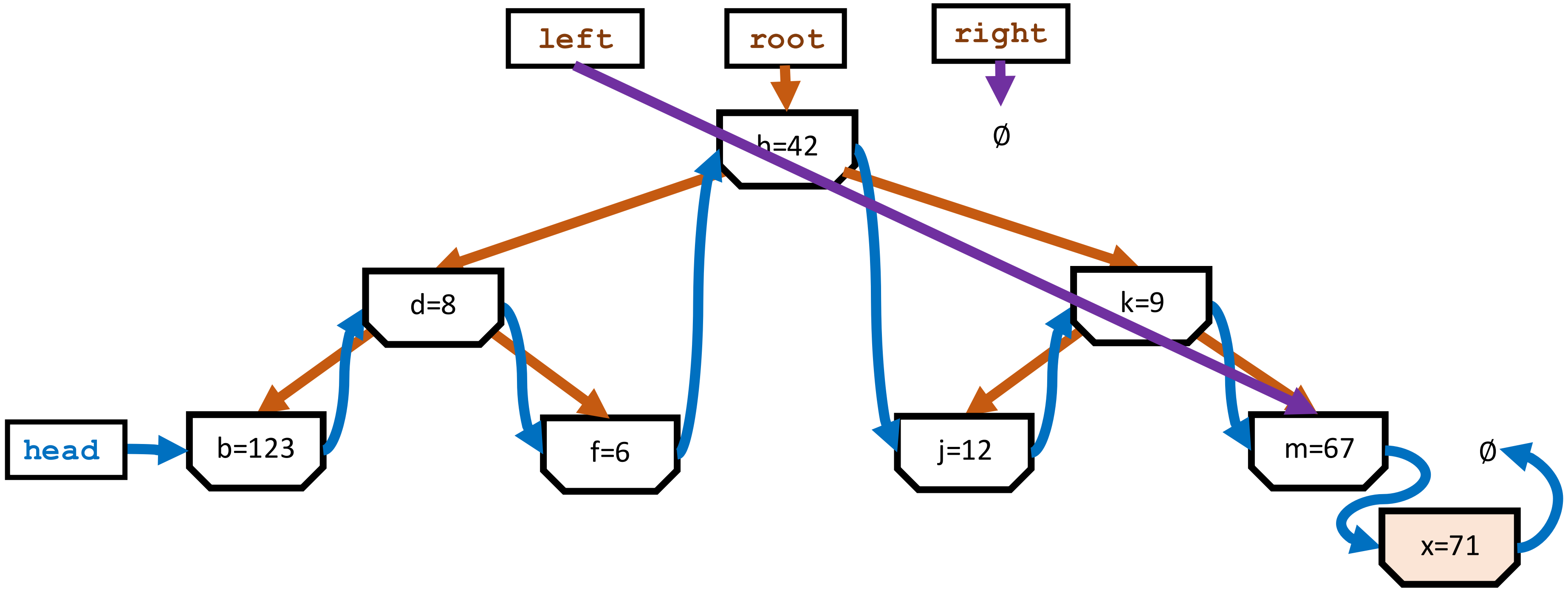
```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    cur = self->__root;
    right = NULL;  /* Our nearest right neighbor */
    left = NULL;   /* Out nearest left neighbor */
    while(cur != NULL ) {
        cmp = strcmp(key, cur->key);
        if(cmp == 0 ) {
            cur->value = value;
            return;
        }
        if( cmp < 0 ) { /* Turn left */
            right = cur;
            cur = cur->__left;
        } else {  /* Turn right */
            left = cur;
            cur = cur->__right;
        }
    }

    ...
}
```

To the right..
To the right..



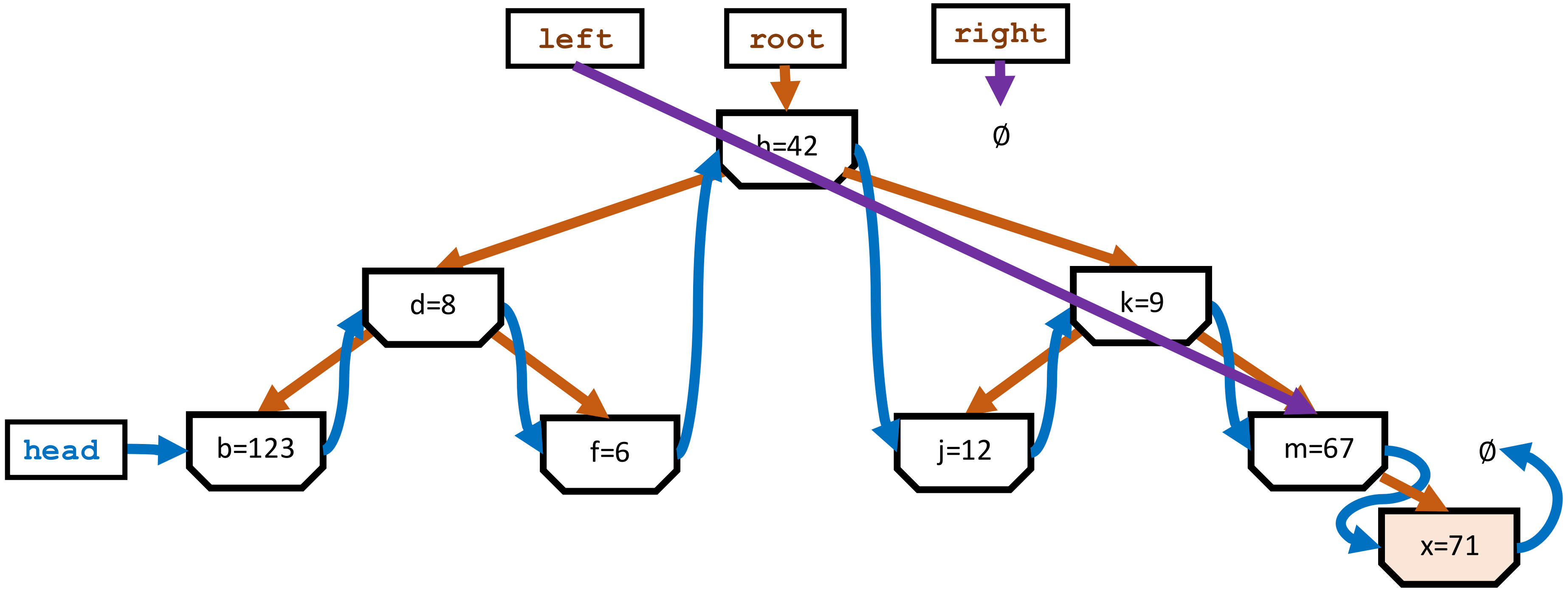
Insert into the Linked List

```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    ...
    // Insert into the sorted linked list
    if ( left != NULL ) {
        new->__next = right;
        left->__next = new;
    } else {
        new->__next = self->__head;
        self->__head = new;
    }
    ...
}
```



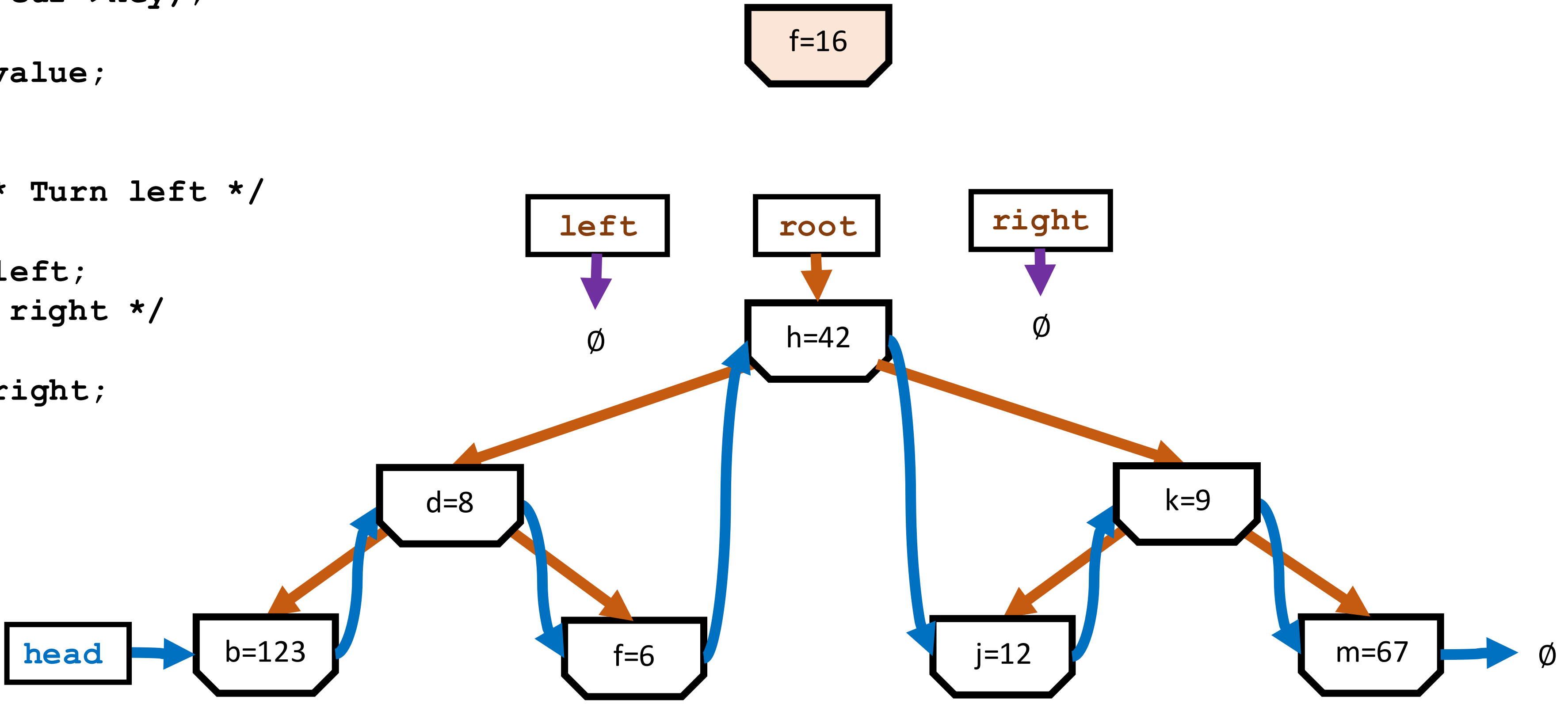
Insert into the Tree

```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    ...
    // Insert into the tree
    if ( right != NULL && right->__left == NULL ) {
        right->__left = new;
    } else if ( left != NULL && left->__right == NULL ) {
        left->__right = new;
    } else {
        printf("FAIL\n");
    }
    ...
}
```



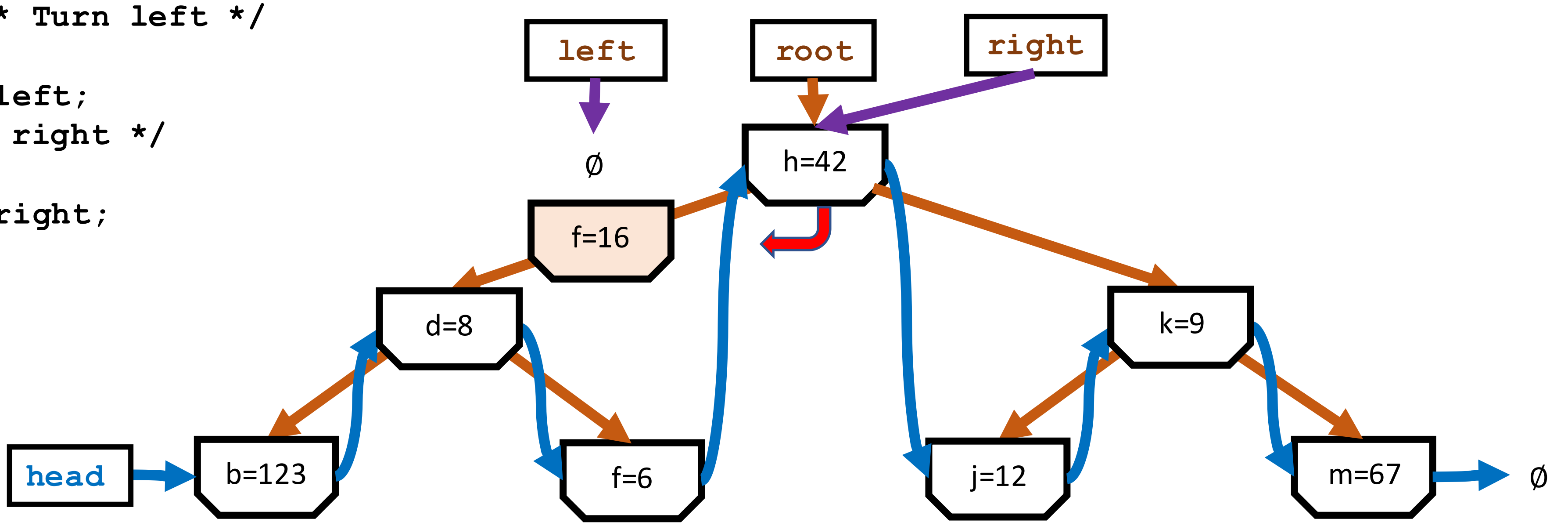
Replace... (1 of 2)

```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    cur = self->__root;
    right = NULL;  /* Our nearest right neighbor */
    left = NULL;   /* Out nearest left neighbor */
    while(cur != NULL ) {
        cmp = strcmp(key, cur->key);
        if(cmp == 0 ) {
            cur->value = value;
            return;
        }
        if( cmp < 0 ) { /* Turn left */
            right = cur;
            cur = cur->__left;
        } else {  /* Turn right */
            left = cur;
            cur = cur->__right;
        }
    }

    ...
}
```



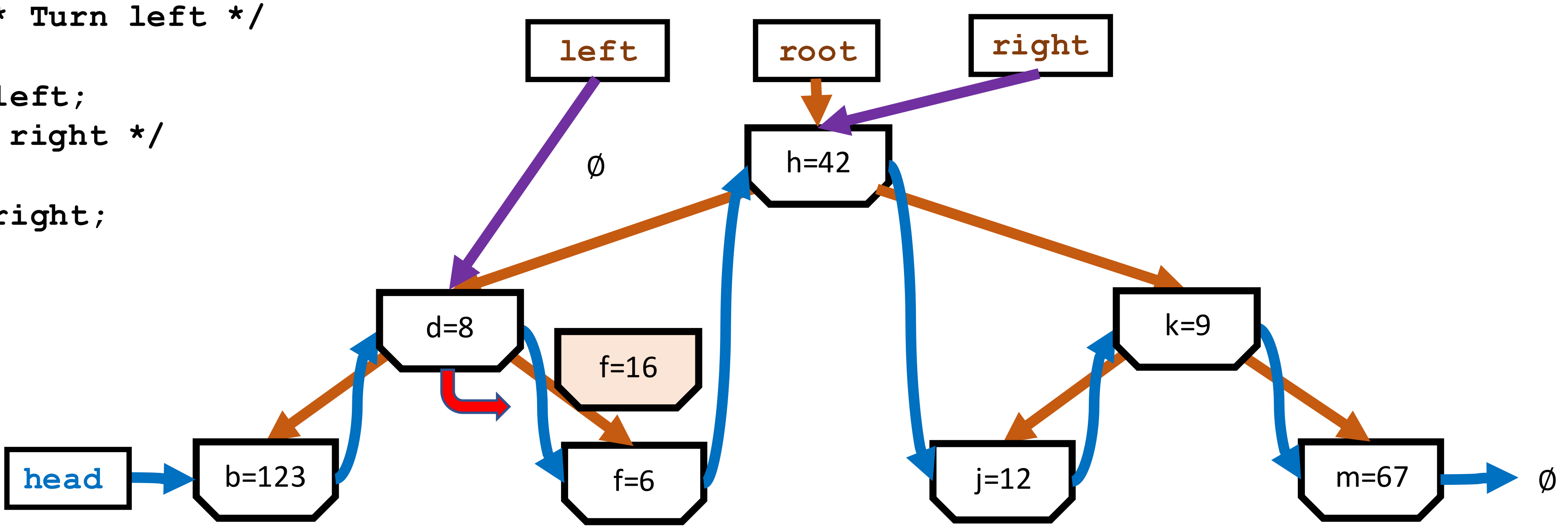
Replace... (2 of 3)

```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    cur = self->__root;
    right = NULL;  /* Our nearest right neighbor */
    left = NULL;   /* Out nearest left neighbor */
    while(cur != NULL ) {
        cmp = strcmp(key, cur->key);
        if(cmp == 0 ) {
            cur->value = value;
            return;
        }
        if( cmp < 0 ) { /* Turn left */
            right = cur;
            cur = cur->__left;
        } else {  /* Turn right */
            left = cur;
            cur = cur->__right;
        }
    }

    ...
}
```



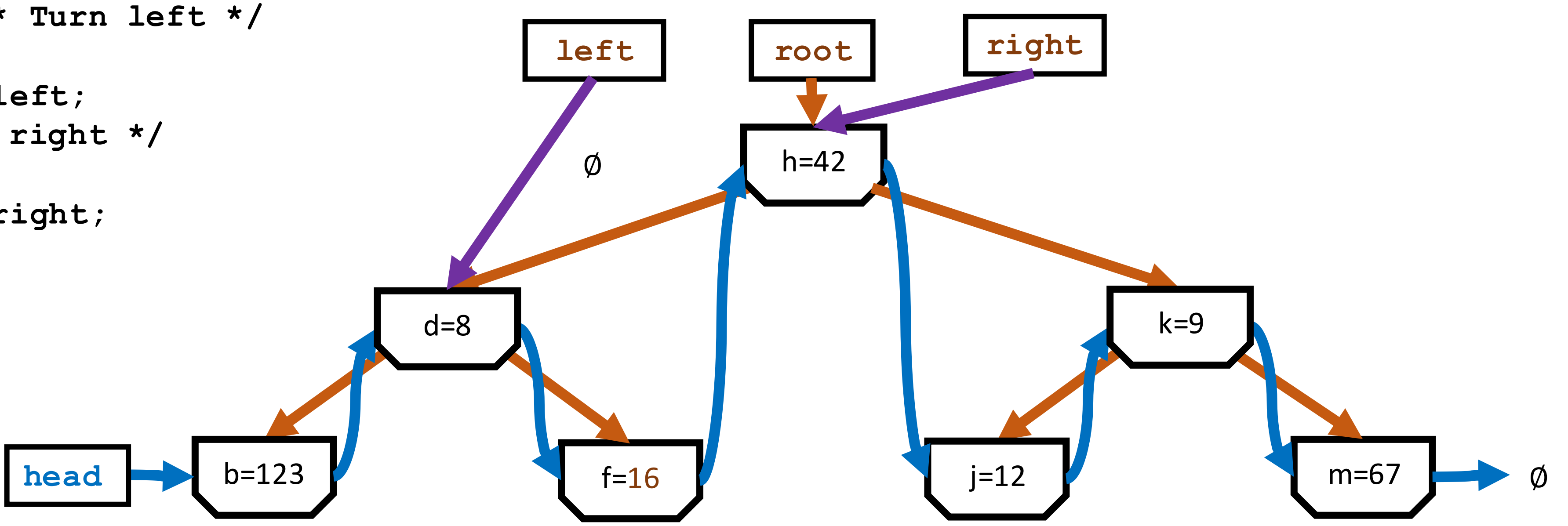
```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    cur = self->__root;
    right = NULL;  /* Our nearest right neighbor */
    left = NULL;   /* Out nearest left neighbor */
    while(cur != NULL ) {
        cmp = strcmp(key, cur->key);
        if(cmp == 0 ) {
            cur->value = value;
            return;
        }
        if( cmp < 0 ) { /* Turn left */
            right = cur;
            cur = cur->__left;
        } else {  /* Turn right */
            left = cur;
            cur = cur->__right;
        }
    }

    ...
}
```

We have a match!!



Update Value

```
void __TreeMap_put(struct TreeMap* self,
   char *key, int value) {

    cur = self->__root;
    right = NULL;  /* Our nearest right neighbor */
    left = NULL;   /* Out nearest left neighbor */
    while(cur != NULL ) {
        cmp = strcmp(key, cur->key);
        if(cmp == 0 ) {
            cur->value = value;
            return;
        }
        if( cmp < 0 ) { /* Turn left */
            right = cur;
            cur = cur->__left;
        } else {  /* Turn right */
            left = cur;
            cur = cur->__right;
        }
    }

    ...
}
```



Test cases for put()

- Inserting into an empty (easy)
- Inserting into a right gap
- Inserting into a left gap
- Inserting at the beginning
- Inserting at the end
- Replacing an entry (easy)

Why Program?

Chapter 1

Python for Everybody
www.py4e.com

```
name = input('Enter file:')
handle = open(name)

counts = dict()
for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word,0) + 1

bigcount = None
bigword = None
for word,count in counts.items():
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)
```

```
python words.py
Enter file: words.txt
to 16
```

```
python words.py
Enter file: clown.txt
the 7
```

```
#include <ctype.h>

#include "map_tree.c"

/**
 * The main program to test and exercise the
 * TreeMap classes.
 */
int main(void)
{
    struct TreeMap * map = TreeMap_new();
    struct TreeMapEntry *cur;
    struct TreeMapIter *iter;
    char name[100];  // Yes, this is dangerous
    char word[100];  // Yes, this is dangerous
    int i,j;
    int count, maxvalue;
    char *maxkey;
```

```
python words.py
Enter file: words.txt
to 16
```

```
gcc words.py
a.out
Enter file: words.txt
to 16
```

```
printf("Enter file name: ");
scanf("%s", name);

FILE *fp = fopen(name, "r");
// Loop over each word in the file
while (fscanf(fp, "%s", word) != EOF) {
    for (i=0, j=0; word[i] != '\0'; i++) {
        if ( ! isalpha(word[i]) ) continue;
        word[j++] = tolower(word[i]);
    }
    word[j] = '\0';
    count = map->get(map, word, 0);
    map->put(map, word, count+1);
}
fclose(fp);

map->dump(map);
```

```
python words.py
Enter file: words.txt
to 16
```

```
gcc words.py
a.out
Enter file: words.txt
to 16
```

```
    maxkey = NULL;
    maxvalue = -1;
    iter = map->iter(map);
    while(1) {
        cur = iter->next(iter);
        if ( cur == NULL ) break;
        if ( maxkey == NULL || cur->value > maxvalue ) {
            maxkey = cur->key;
            maxvalue = cur->value;
        }
    }
    iter->del(iter);
    printf("\n%s %d\n", maxkey, maxvalue);

    map->del(map);
}
```

```
python words.py
Enter file: words.txt
to 16
```

```
gcc words.py
a.out
Enter file: words.txt
to 16
```

“Stopping by Woods on a Snowy Evening”

Whose woods these are I think I know.
His house is in the village though;
He will not see me stopping here
To watch his woods fill up with snow.

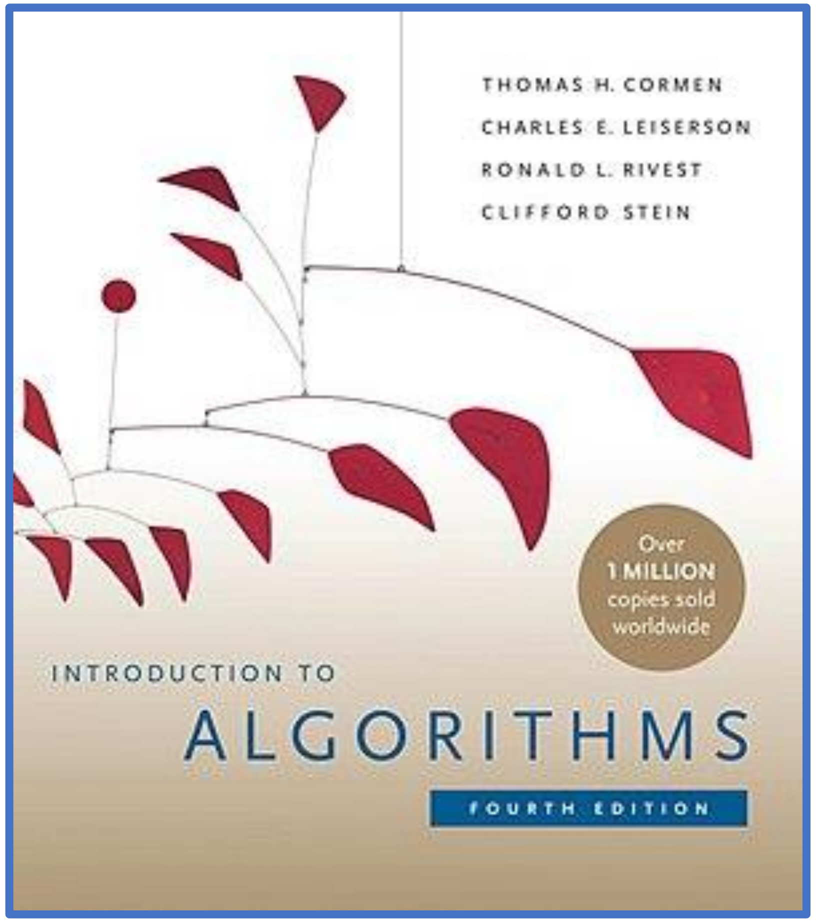
My little horse must think it queer
To stop without a farmhouse near
Between the woods and frozen lake
The darkest evening of the year.

He gives his harness bells a shake
To ask if there is some mistake.
The only other sound’s the sweep
Of easy wind and downy flake.

The woods are lovely, dark and deep,
But I have promises to keep,
And miles to go before I sleep,
And miles to go before I sleep.

The end is really the beginning…

- Data structures are cool
- Like a recipe from a cookbook
- You combine foundational ingredients to produce something amazing
- Think of this course as learning how to cook your first omelette
- Your journey can continue with many great "cook" books



Acknowledgements / Contributions

Initial Development: Charles Severance, University of Michigan School of Information

Insert new Contributors and Translators here including names and dates

Continue new Contributors and Translators here