Kernel data structures

Kernel data structures

The Linux Kernel is a known source of best practices

- It is self-contained, as it can't rely on libc
- It's a complex project that would collapse from oversmartness
- It has a lot of data to handle, and problems repeat overall

Rob Pike's rules of programming (shortened):

- Rule 3. Fancy algorithms are slow when n is small, and n is usually small.
- Rule 4. Use simple algorithms as well as simple data structures.
- Rule 5. Data structures, not algorithms, are central to programming.

Obviously, the kernel uses linked lists in several places

Obviously, the kernel hosts a number of sorted data

Lists are the most basic data structures

Everybody is implementing lists all the time

- This is a trivial example, that reverses stdin lines to stdout

```
struct str_item { char str[16]; struct str_item *next; };
#define list_insert(h, new) \
     ({(new)->next = (h); (h) = (new);})
#define list_extract(h) \
     ({struct int_item *res = (h); if (h) (h)=(h)->next; res;})

        while (fgets(line, 16, stdin)) {
                item = malloc(sizeof(*item));
                memcpy(item->str, line, sizeof(item->str));
                list_insert(head, item);
        }
        while ( (item = list_extract(head)) ) {
                printf("%s", item->str); free(item);
       }
```

Such code is being rewritten over and over

The list must be reimplemented for each and every data item

```
struct str_item {
    char str[16];
    struct str_item *next;
};
while (fgets(line, 16, stdin))
  {
  item = malloc(sizeof(*item));
  memcpy(item->str, line, 16);
  list_insert(head, item);
  }
```

```
struct int_item {
    int value;
    struct int_item *next;
};
while (fgets(line, 16, stdin))
  {
  item = malloc(sizeof(*item));
  item->value = atoi(line);
  list_insert(head, item);
  }
```

And double-linked lists are not so easy to write and rewrite

- Simple lists allow some operations, but are quite limited
- You really need something more to do stuff different from reversing

A possible solution: the generic list

```
struct generic_list {
        struct generic_list *next;
        void *payload;
};
#define list_insert(h, new) \
        ({(new)->next = (h); (h) = (new);})
#define list_extract(h) \
        ({struct generic_list *res = (h); if (h) (h) = (h)->next; res;})
```

The implementation above leads to the following code

```
    while (fgets(line, 16, stdin)) {
            item = malloc(sizeof(*item));
            item->payload = malloc(sizeof(line));
            memcpy(item->payload, line, sizeof(line));
            list_insert(head, item);
    }
```

We are used to separate the payload from the real work

- This approach allows to refine list management over time
- There is minimal effort in porting to a different payload
- The slightly extra work (alloc/free) is not expected to be a problem

Actual measures show the result is very bad

Modern systems feature a lot of tricks to be faster

- The average case is greatly improved
- But the worst case is greatly worsened

The problem in this case is most likely cache memory

- Data access within the same cache line is almost free
  but access to a different cache line is awfully expensive
- Here, the two allocations will often fall on different cache lines
- Every access to data requires two RAM accesses (with two cache miss)

The "generic" code shown is 20% slower than simple lists

- The measure is on the whole program, including I/O

Make it as simple as possible, not simpler

The real "simple" solution is going back to single allocations

- The individual allocation ensures better data locality
- Which in turn means less cache miss events, and more performance

But we need a basically different approach...

The approach taken in the kernel is reversing the structure

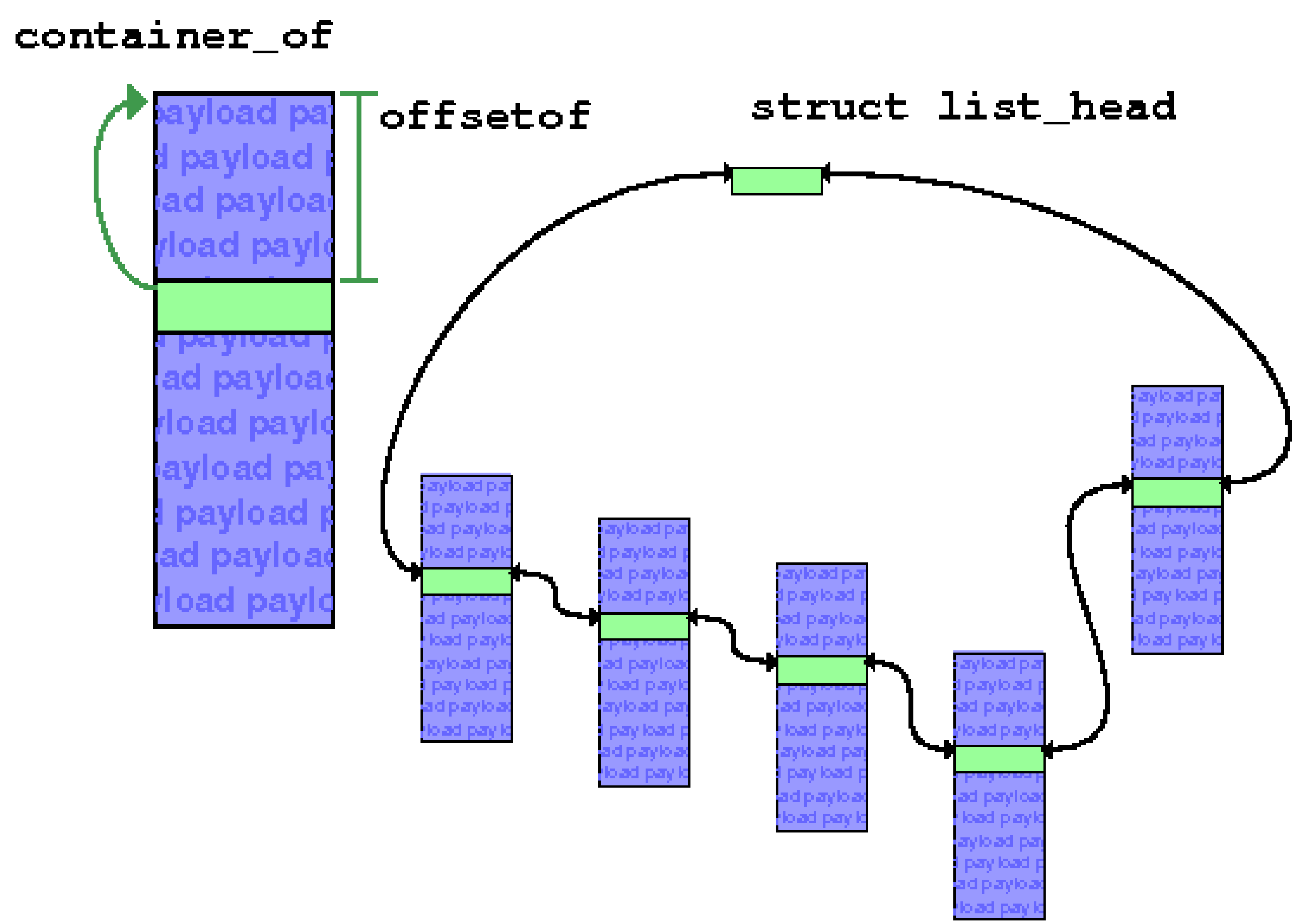
- Instead of including the payload in the list structure
  the list structure is included in the payload itself

```
#include <linux/list.h>
struct list_head {
        struct list_head *next, *prev;
};
```

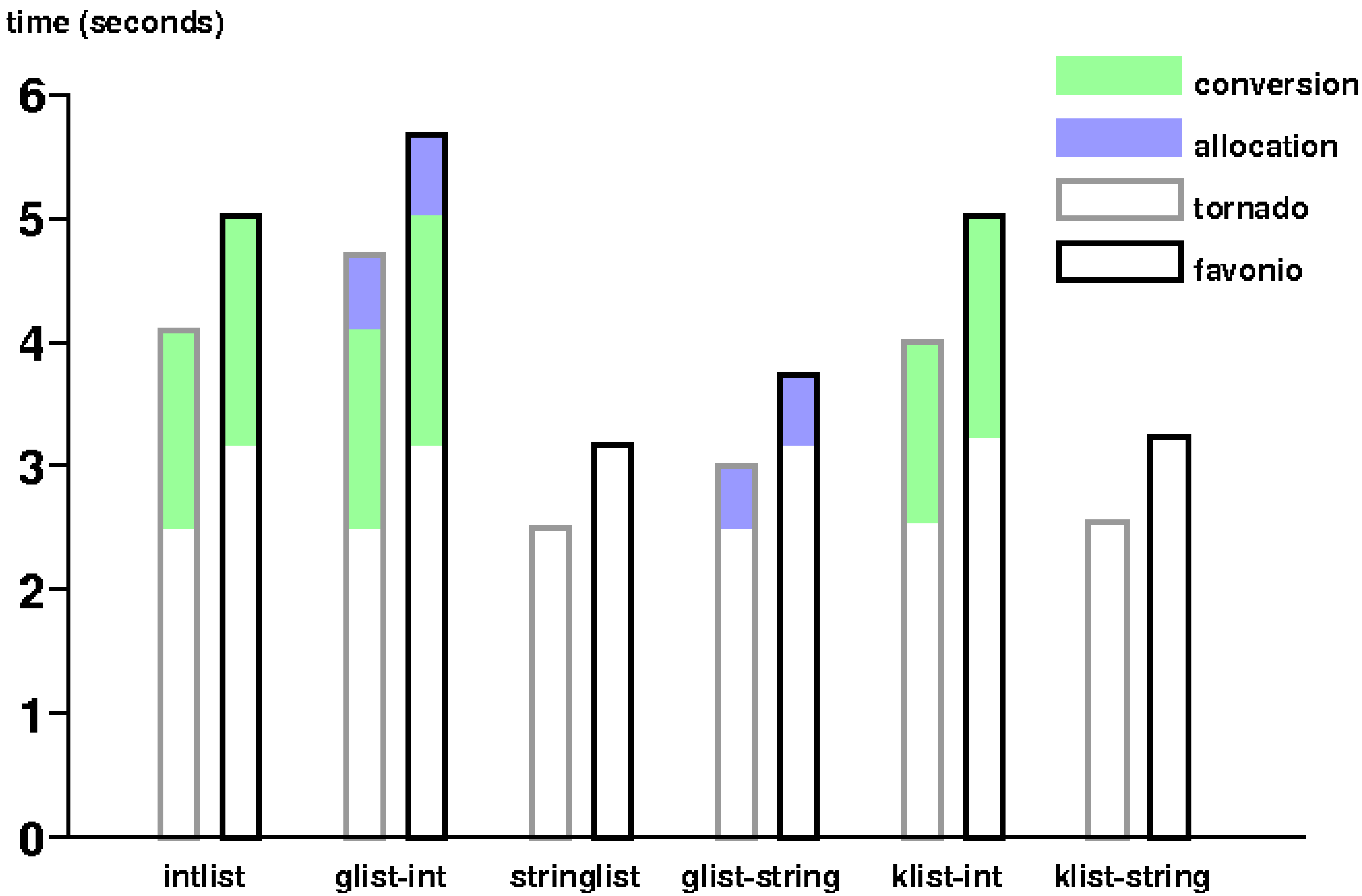
```
static inline void list_add(struct list_head *new, struct list_head *head);
static inline void list_add_tail(struct list_head *new, struct list_head *head);
static inline void list_del(struct list_head *entry);
/* .... */
```

The basic tool under this is "container_of"

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
#define container_of(ptr, type, member) ({                        \
        const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
        (type *)( (char *)__mptr - offsetof(type,member) );})
```



Klist performance is the same as trivial lists



- A double-linked list is as efficient as the single-linked one
- A data structure can be in several lists at the same time
- Cache locality is warranted
- The code offers many more goodies, like "list_for_each(list)"

The sorting problem: trivial trees

Like trivial lists, trivial trees are often rewritten

```
int tt_insert(struct node *tree, char *s)
{
    struct node *new, **nextp;
    if (strcmp(tree->s, s) > 0) {
        if (tree->left)
            return tt_insert(tree->left, s);
        else
            nextp = &tree->left;
    } else {
        if (tree->right)
            return tt_insert(tree->right, s);
        else
            nextp = &tree->right;
    }
    new = calloc(1, sizeof(*new));
    if (!new)
        return -1;
    strcpy(new->s, s);
    *nextp = new;
    return 0;
}
```

```
struct node {
    char s[SLEN];
    struct node *left;
    struct node *right;
};
```

Trivial trees, actually, are not up to the task

Unlikely trivial lists, a trivial tree is rarely acceptable

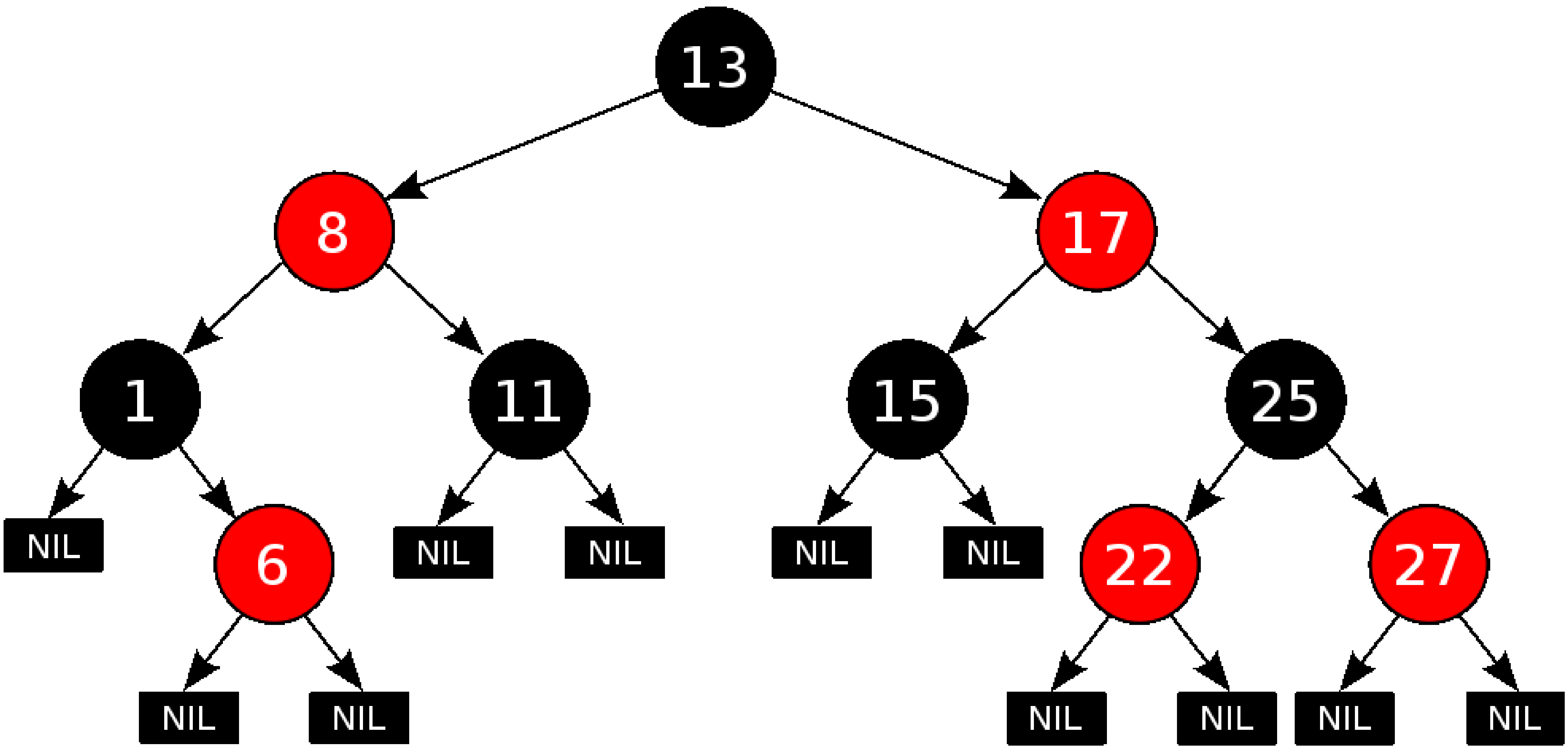
- It suffers horribly from being unbalanced
- Search time degenerates from O(log n) to O(n)
- Thus, it is not suitable for real use, not even in small environments

Balanced trees are not sth you can master in half an hour

- To make things worse, there are several implementations

RB trees, almost balanced (Bayer, 1972)

- **Nodes are either red or black**
- **Children of red nodes must be black**
- **All root-to-leaf paths have the same number of black nodes**



RB trees in the kernel (Arcangeli, 1999)

Kernel rbtrees go inside the payload, like lists

```
static void insert_line(struct line_rb *item,
          struct rb_root *root)
{
    struct rb_node **p = &root->rb_node;
    struct rb_node *parent = NULL;
    struct line_rb *lrb;

    while (*p)
    {
        parent = *p;
        lrb = rb_entry(parent, struct line_rb, rb);

        if (strcmp(item->line, lrb->line) < 0)
            p = &(*p)->rb_left;
        else
            p = &(*p)->rb_right;
    }

    rb_link_node(&item->rb, parent, p);
    rb_insert_color(&item->rb, root);
}
```

```
struct line_rb {
    char line[SLEN];
    struct rb_node rb;
};
```

The code is split between .h and .c files

Rebalancing and a few more operations are library functions

- There is a little more work involved in porting to user space
- The implementation is very efficient, both in size and speed
- The suggested traversal of the tree is iterative, not recursive

```
rudo$ nm --size-sort linux_rbtree.o
00000021 T rb_first
00000021 T rb_last
00000042 T rb_next
00000042 T rb_prev
0000005d T rb_replace_node
0000006d t __rb_rotate_left
0000006d t __rb_rotate_right
000000eb T rb_insert_color
000002c9 T rb_erase
```

A sorting program built with linux-rbtree is

- Almost as fast as /usr/bin/sort
- Comparable with trivial-tree on random data
- Slower than qsort, but data is always sorted during operation

Reusing Linux rbtree outside Linux

You may re-use this rbtree implementation in practice

- For example, to implement malloc

A malloc implementation with rbtree is small

- It is also reasonably fast and scales well

Surprisingly, a first-fit implementation is faster if the number of blocks is not exceedingly large

And we are back the Pike's programming rules:

- Rule 1. You can't know where a program is spending its time.
- Rule 2. Always measure before you optimize.
- Rule 3. Fancy algorithms are slow when n is small, and n is usually small.

Other goodies and final considerations

The two structures shown are just examples

- In the headers and in lib/ there are more structures
  - hash tables (several flavours)
  - other kinds of trees
  - other flavours of lists
  - checksumming algorithms
  - decompression of the various kinds
  - ....

The implementation is very high quality, in all cases

The code is self-contained and very easy to reuse

It's very good code for teaching programming as well