Hash Tree Features

- Very small hash tree structure. [16 Bytes]
- Dynamic memory allocation and free.
- Both 32-bit and 64-bit indexes are possible
- Generate hash keys uniformly based on the index.
- Hash trees are balanced by hash keys, and have no rotation costs.
- Standard deviation of hash key is 4 or less.
- Algorithm O(n) is depth(d) x nodes(c)
- Finding walks is (d x c), min:4, avg:12, max:20
- First hash table has smallest, largest index, algorithm O(1).
- The codes implementing of the algorithm is simple.
- Adjust hash tree depth according to system memory and index nr.
- Hash list nodes use include/linux/list.h, hlist as their base.

```
struct hash_tree {
    struct hlist_head head;
    struct hash_tree *next;
} __aligned(16);
```

```
#define HTREE_HASH_KEY(idx, d, bits)    ( sizeof(idx) <= 4 ?    \
    (((u32)idx + d) * htgr32[d]) >> (32 - bits) :         \
    (((u64)idx + d) * htgr64[d]) >> (64 - bits) )
```

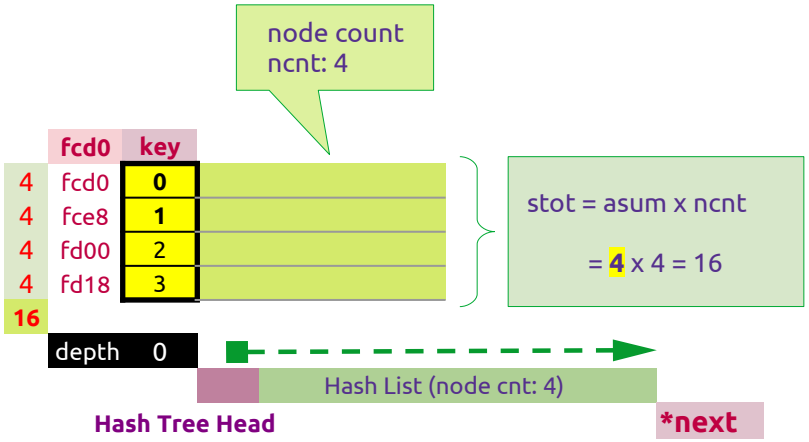
JaeJoon Jung <rgbi3307@gmail.com>

```
hlist_entry_safe(htree[hts->most].head.first,
                 struct htree_data, hnode);
```

```
u8 _ht_hts_get_sbit(u64 maxnr)
```

htree-nodes

bits: 2
table size(anum): 1 << 2 = 4
alloc count(acnt): 1
sum anum(asum): anum x acnt
node count(ncnt): **4**
data count(dcnt): **16**
stot = asum x ncnt = **4** x 4 = 16
eff = dcnt / stot = **16** / 16 = **1.0**



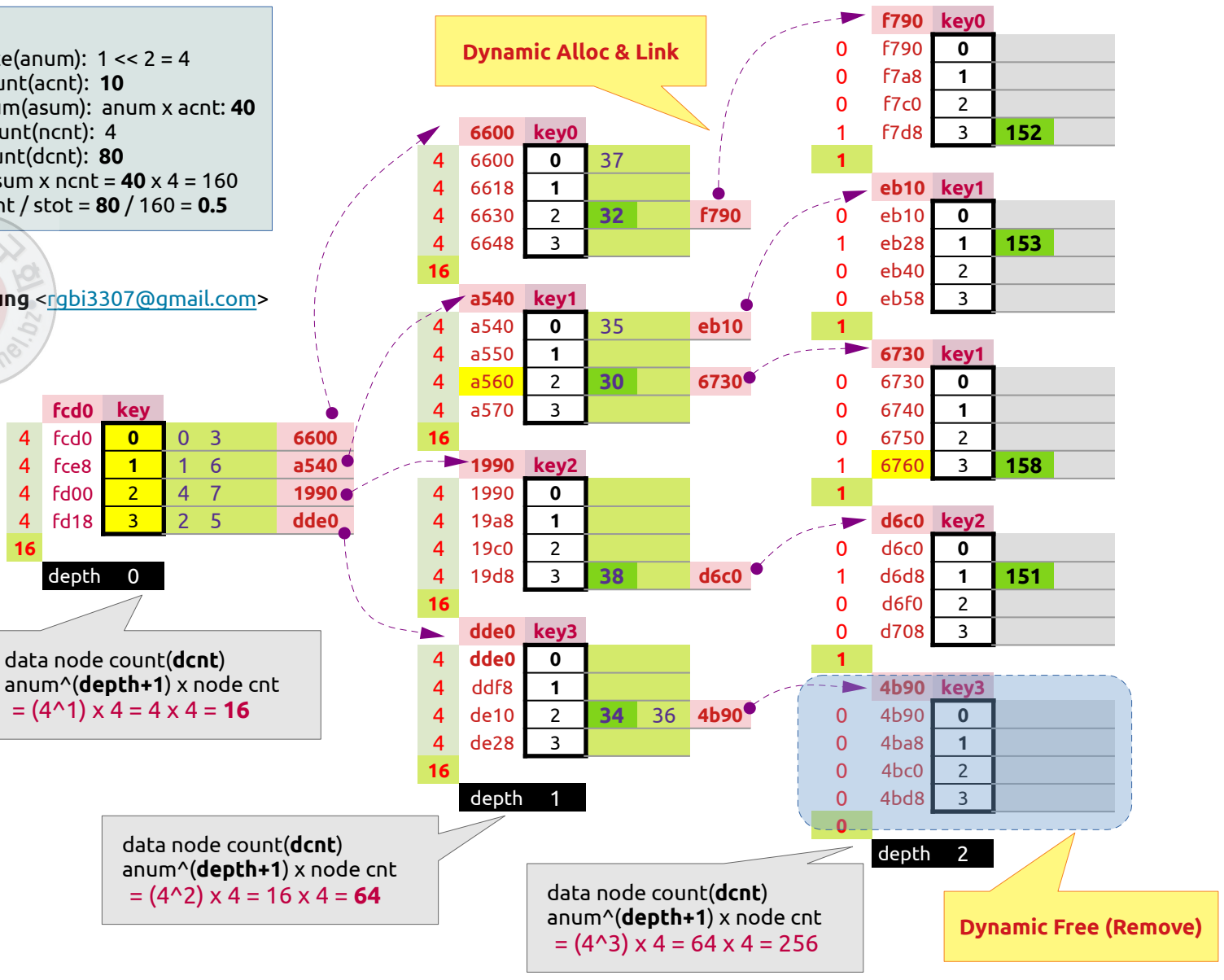

```
struct hash_tree {
    struct hlist_head head;
    struct hash_tree *next;
} __aligned(16);
```

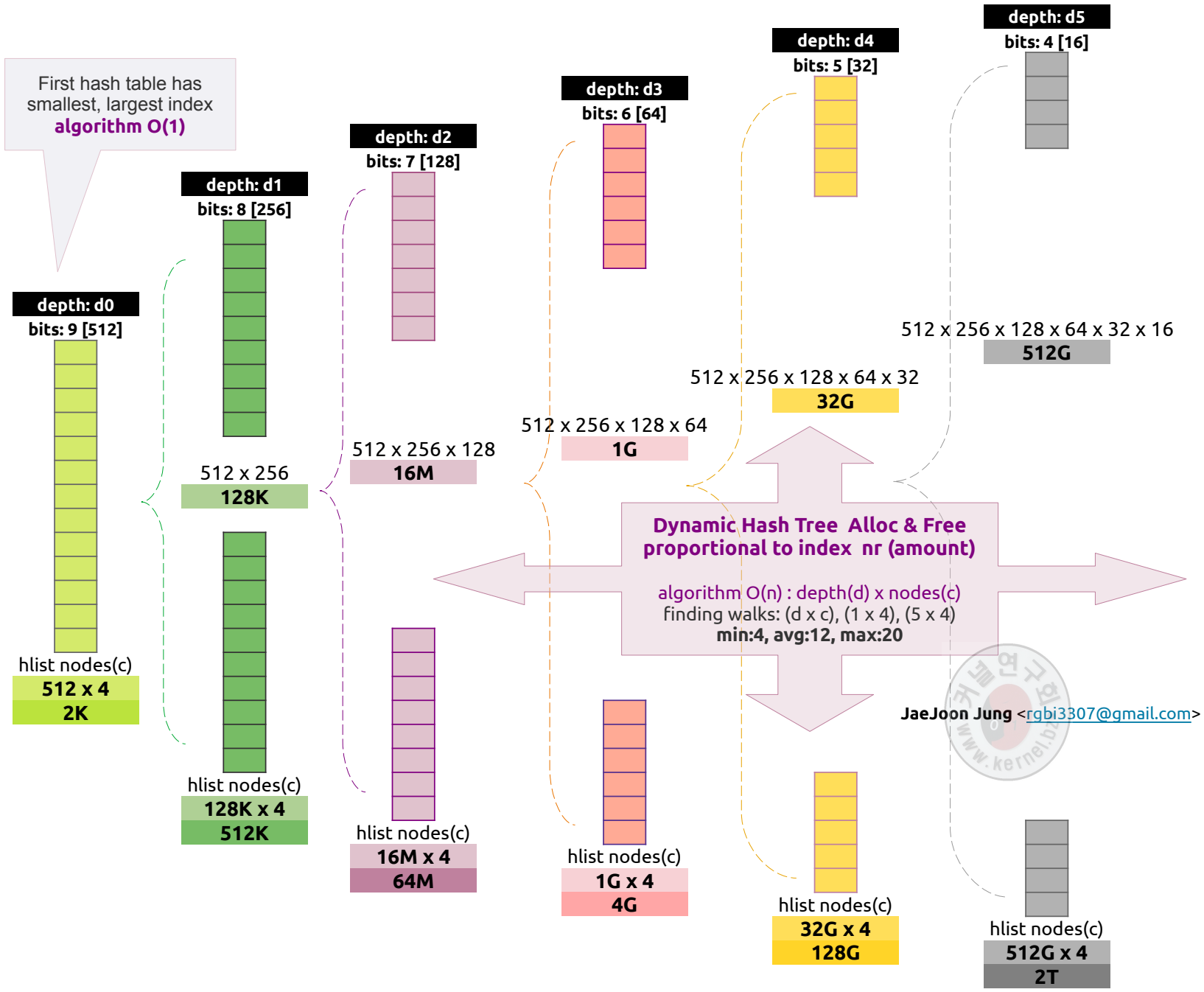
JaeJoon Jung <rgbi3307@gmail.com>

htree-next-links

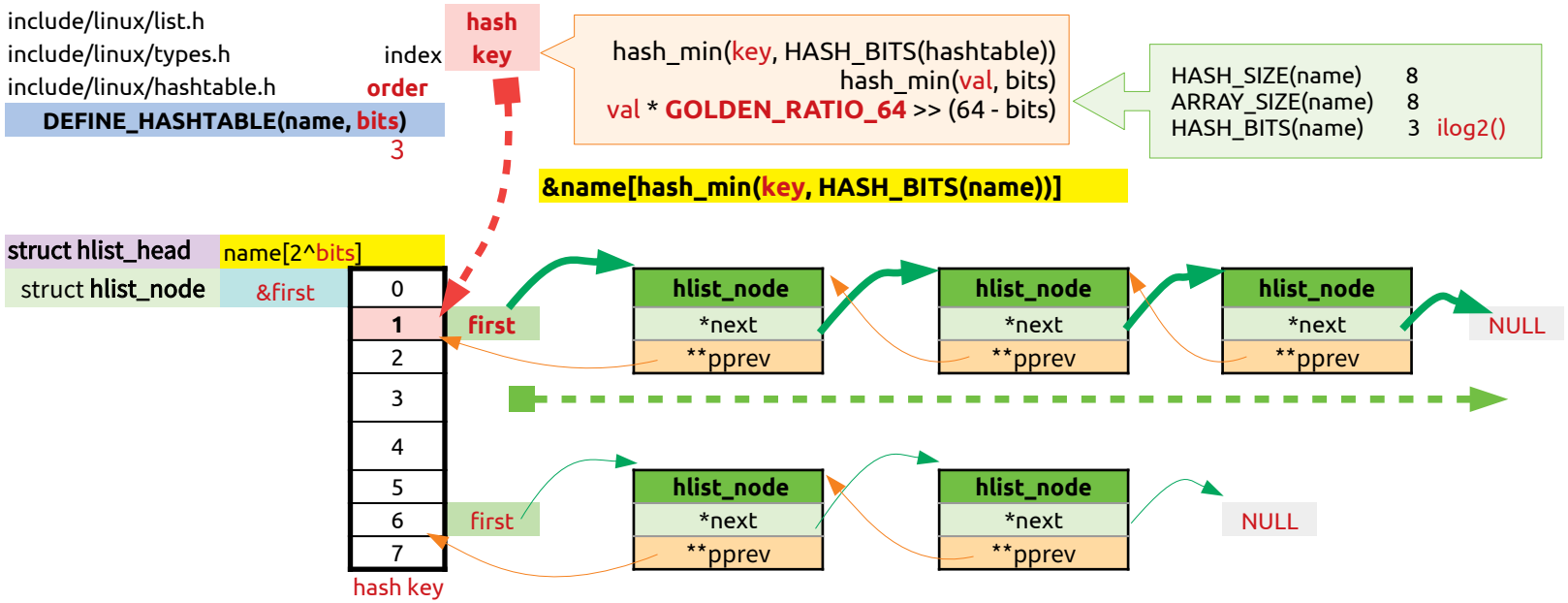
bits: 2
table size(anum): 1 << 2 = 4
alloc count(acnt): **10**
sum anum(asum): anum x acnt: **40**
node count(ncnt): 4
data count(dcnt): **80**
stot = asum x ncnt = **40** x 4 = 160
eff = dcnt / stot = **80** / 160 = **0.5**

JaeJoon Jung <rgbi3307@gmail.com>




htree-depth
First hash table has smallest, largest index
algorithm O(1)
depth: d0
bits: 9 [512]
hlist nodes(c)
512 x 4
2K
depth: d1
bits: 8 [256]
512 x 256
128K
hlist nodes(c)
128K x 4
512K
depth: d2
bits: 7 [128]
512 x 256 x 128
16M
hlist nodes(c)
16M x 4
64M
depth: d3
bits: 6 [64]
512 x 256 x 128 x 64
1G
hlist nodes(c)
1G x 4
4G
depth: d4
bits: 5 [32]
512 x 256 x 128 x 64 x 32
32G
hlist nodes(c)
32G x 4
128G
depth: d5
bits: 4 [16]
512 x 256 x 128 x 64 x 32 x 16
512G
hlist nodes(c)
512G x 4
2T
Dynamic Hash Tree Alloc & Free
proportional to index nr (amount)
algorithm O(n) : depth(d) x nodes(c)
finding walks: (d x c), (1 x 4), (5 x 4)
min:4, avg:12, max:20
JaeJoon Jung <rgbi3307@gmail.com>

hash-lists



JaeJoon Jung <rgbi3307@gmail.com>

hashtable

include/linux/hashtable.h

DEFINE_HASHTABLE(name, bits)

3

[2 ^ bits], bits = HASH_BITS(name)

struct hlist_head	**name**[1 << bits]		
struct **hlist_node**	&first	0	NULL
		1	NULL
		2	NULL
		3	NULL
		4	NULL
		5	NULL
		6	NULL
		7	NULL

ARRAY_SIZE(name)

JaeJoon Jung <rgbi3307@gmail.com>

Hash Tree API flow (include/linux/htree.h, lib/htree.c, lib/htree-test.c)

```
*hts = ht_hts_alloc()                                   /* alloc hts */

ht_hts_clear_init(hts, maxnr, idx_type, sort_type)      /* max index nr, type(32/64bits), sort(ASC, DES) */

*htree = ht_table_alloc(hts)                            /* alloc first(depth:0) htree */
```

```
run_loop() {

        *udata = _data_alloc(index)                     /* alloc udata */


        ht_insert(hts, htree, udata->hdata, ..)
        ht_erase(hts, htree, index)                     /* working data with index */
        hdata = ht_find(hts, htree, index)

        hdata = ht_most_index(hts, htree)               /* smallest, largest index */
        ht_statis(hts, htree, ...)                      /* statistic */
}
```

JaeJoon Jung <rgbi3307@gmail.com>


```
htree_erase_all(hts, htree)                             /* remove all udata */

ht_destroy(hts, htree)                                  /* remove all htree */

kfree(hts)                                              /* remove hts */
```