# Linux kernel data structure

## Red-Black Tree & Hash table

### Practical Class 12

Systems and Storage Laboratory
Department of Computer Science and Engineering
Chung-Ang University

# Red-Black Tree

**Linux kernel data structure**

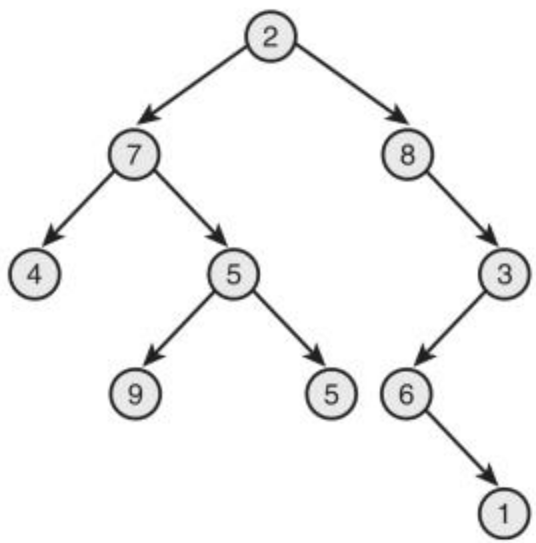
# What are red-black trees?

- **Red-black trees are**
  - A type of self-balancing binary search tree
  - Used for storing sortable key/value data pairs
- **Red-black tree differs from**
  - **Radix trees**
    - Used to efficiently store sparse arrays
    - Thus use long integer indexes to insert/access/delete nodes
  - **Hash tables**
    - Not kept sorted to be easily traversed in order
    - Must be tuned for a specific size and hash function
      - ✓ While rbtrees scale gracefully storing arbitrary keys
- **For more information on the nature and implementation of Red Black Trees, see:**
  - Linux Weekly News article on red-black trees (https://lwn.net/Articles/184495/)
  - Wikipedia entry on red-black trees (https://en.wikipedia.org/wiki/Red-black_tree)

# Tree basics : Binary Tree

## ❖ Properties

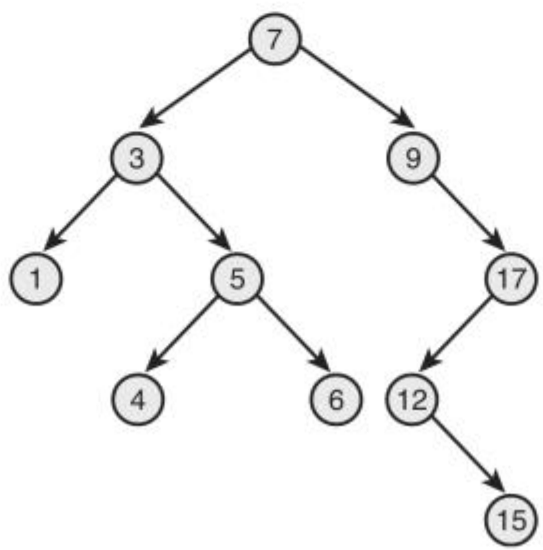
- Nodes have zero, one, or two children
- Root has no parent, other nodes have one



# Tree basics : Binary Search Tree

## Properties

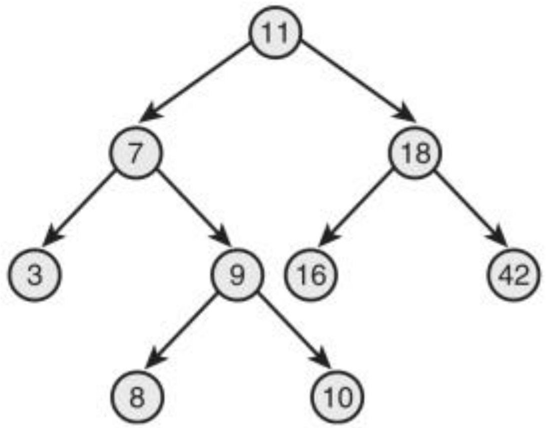
- Left children < parent
- Right children > parent
- Search and ordered traversal are efficient



# Tree basics : Balanced Binary Search Tree

## Properties

- Depth of all leaves differs by at most one
- Puts a boundary on the worst case operations

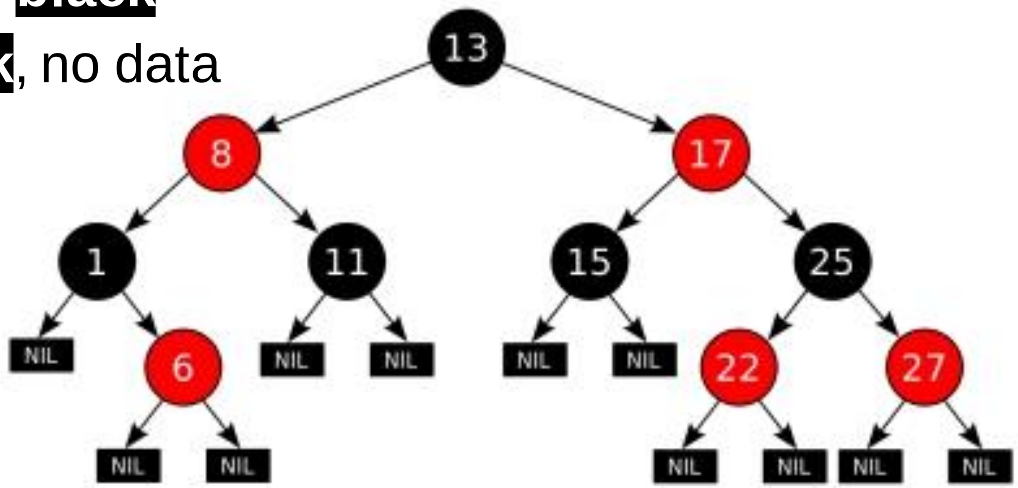


# Tree basics : Red-black Tree

## ❖ Properties

- A type of **self-balancing binary search tree**
- The path from a node to one of its leaves contains the same number of black nodes as the shortest path to any of its other leaves.
- Fast search, insert, delete operations: *O(logN)*

  - **Nodes**: red or black
  - **Leaves**: black, no data



# Red-black Tree data structures

## Rbtree node

- Which will be embedded into your data structure similar to list_head and hlist node

```
struct rb_node {
    unsigned long  __rb_parent_color;
    struct rb_node *rb_right;
    struct rb_node *rb_left;
} __attribute__((aligned(sizeof(long))));
```

## Rbtree root

```
struct rb_root {
    struct rb_node *rb_node;
};
```

- Initializing rbtree root

```
#define RB_ROOT (struct rb_root) { NULL, }
```

# Red-Black Tree APIs

- **Iterating through the elements stored in an rbtree (in sort order)**
  - Find logical next and previous nodes in a tree

```
struct rb_node *rb_next(const struct rb_node *);
struct rb_node *rb_prev(const struct rb_node *);
struct rb_node *rb_first(const struct rb_root *);
struct rb_node *rb_last(const struct rb_root *);
```

- **Insert node**
  - Insert a new node under a parent connected via rb_link

```
void rb_link_node(struct rb_node *node, struct rb_node *parent,
                  struct rb_node **rb_link);
```

# Red-Black Tree APIs

## Replace node

- Replace an existing node in a tree with a new one with the same key
  - Fast replacement of a single node without remove/rebalance/add/rebalance

```
void rb_replace_node(struct rb_node *victim, struct rb_node *new,
                     struct rb_root *root);
```

## Rebalance (Recoloring)

- Re-balance an rbtree after inserting a node if necessary

```
void rb_insert_color(struct rb_node *, struct rb_root *);
```

## Delete a node

- Remove an existing node from a tree

```
void rb_erase(struct rb_node *, struct rb_root *);
```

# Red-Black Tree example

- Create a simple module to test Red-Black Tree

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/rbtree.h>
#include <linux/slab.h> // for kmalloc

#define FALSE 0
#define TRUE 1

int __init hello_module_init(void)
{
    printk("module init\n");

    rbtree_exmaple();

    return 0;
}

void __exit hello_module_cleanup(void)
{
    printk("Bye Module\n");
}

module_init(hello_module_init);
module_exit(hello_module_cleanup);
```

# Red-Black Tree example

## ❖ Example Red-Black Tree test codes

```
struct mytype {
    struct rb_node node;
    int key;
    int value;
};

void rbtree_exmaple(void)
{
    struct rb_root mytree = RB_ROOT;
    int i = 0, succeed;

    /* rb_node create and insert */
    for (; i < 20; i++) {
        struct mytype *new = kmalloc(sizeof(struct mytype), GFP_KERNEL);

        if (!new)
            return NULL;
        new->value = i * 10;
        new->key = i;

        succeed = my_rb_insert(&mytree, new);
    }
    ...
```

*Initialize our rbtree root*

*Create 20 nodes and insert them into rbtree*

# Red-Black Tree example

```
    ...
    /* rb_tree traversal using iterator */
    struct rb_node *node;
    for (node = rb_first(&mytree); node; node = rb_next(node))
        printk("(key, value) = (%d, %d)\n",      \
            rb_entry(node, struct mytype, node)->key,     \
            rb_entry(node, struct mytype, node)->value);


    /* rb_tree find node */
    struct mytype *found_node = my_rb_search(&mytree, 8);

    if(!found_node){
        return NULL;
    }
    printk("find : (key,value)=(%d.%d)\n",
        found_node->key, found_node->value);


    /* rb_tree delete node */
    my_rb_delete(&mytree, 0);
}
```

*Traverse all nodes in the rbtree by using iterator*

*Find index 8 node from rbtree*

*Delete index 0 node from rbtree*

# Red-Black Tree example

- **Insert**
  - Inserting data in the tree involves:
    1. First search for the place to insert the new node,
    2. Then insert the new node
    3. Rebalance ("recoloring") the tree.
  - The search for insertion differs from the previous search by finding the location of the pointer on which to graft the new node.
  - The new node also needs a link to its parent node for rebalancing purposes.

# Red-Black Tree example

## ❖ Insert implementation example

- To use rbtrees you'll have to implement your own insert function

```c
int my_rb_insert(struct rb_root *root, struct mytype *data)
{
    struct rb_node **new = &(root->rb_node), *parent = NULL;

    /* Figure out "where" to put new node */
    while (*new) {
        struct mytype *this = rb_entry(*new, struct mytype, node);

        parent = *new;
        if (this->key > data->key)
            new = &((*new)->rb_left);
        else if (this->key < data->key)
            new = &((*new)->rb_right);
        else  /* this->key == data->key; node already exists */
            return FALSE;
    }

    rb_link_node(&data->node, parent, new); /*relinking*/
    rb_insert_color(&data->node, root);     /*recoloring & rebalancing*/

    return TRUE;
}
```

*Search for the place to insert the new node*

*Failed to insert*

*Insert the new node*

*Rebalance ("recoloring") the tree*

*Insert succeed*

# Red-Black Tree example

## Search

- To use rbtrees you'll have to implement your own search function
- Writing a search function for your tree is fairly straightforward:
  1. Start at the root,
  2. Compare each value,
  3. Follow the left or right branch as necessary.

```
struct mytype *my_rb_search(struct rb_root *root, int key)
{
    struct rb_node *node = root->rb_node;

    while (node) {
        struct mytype *data = rb_entry(node, struct mytype, node);

        if (data->key > key)
            node = node->rb_left;
        else if (data->key < key)
            node = node->rb_right;
        else   /* data->key == key */
            return data;
    }
    return NULL;
}
```

*Iterating the rbtree*

*Succeed, found what we were looking for*

*Failed, nothing was there with the index of "key"*

# Red-Black Tree example

## ❖ Delete node

```
int my_rb_delete(struct rb_root *mytree, int key)
{
    struct mytype *data = my_rb_search(mytree, key);

    if (data) {
        rb_erase(&data->node, mytree);
        kfree(data);
    }
}
```

*Search index "key" from rbtree*

*Remove found node from rbtree*

*Free allocated memory for node*

# Red-Black Tree Usage

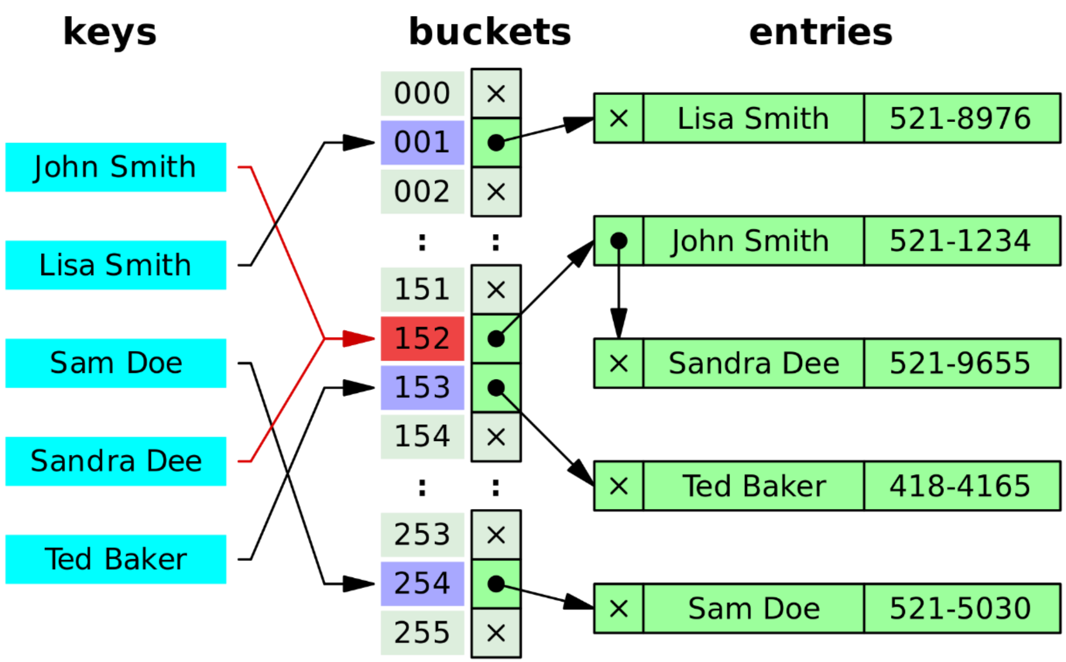
- **Completely Fair Scheduling (CFS)**
  - Default task scheduler in Linux
  - Each task has **vruntime**, which presents how much time a task has run
  - CFS always picks a process with the smallest **vruntime** for fairness
  - Per-task **vruntime** structure is maintained in a rbtree
- **The deadline and CFQ I/O schedulers**
- **The packet CD/DVD driver**
  - Employ rbtrees to track requests
- **The high-resolution timer**
  - Uses an rbtree to organize outstanding timer requests
- **The ext3 filesystem**
  - Tracks directory entries in a red-black tree
- **Virtual memory areas (VMAs)**
- **Epoll file descriptors**
- **Cryptographic keys**
- **Network packets in the "hierarchical token bucket" scheduler**
  - Tracked with red-black trees
- **And More...**

# Hash table

**Linux kernel data structure**

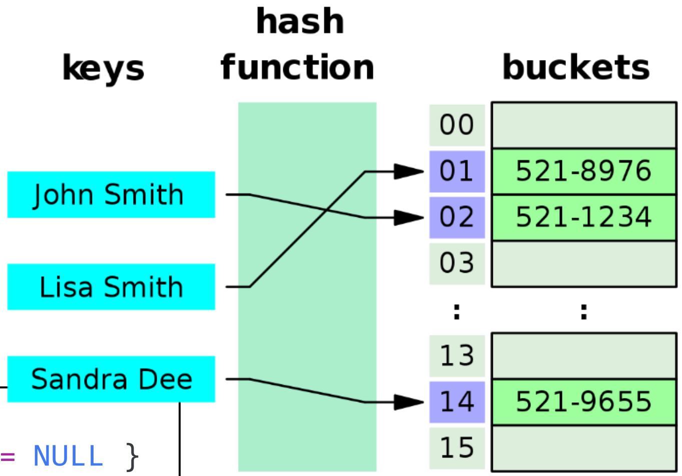
# Hash table

- **A simple fixed-size chained hash table**
  - The size of bucket array is fixed at initialization as a $2^N$
  - Each bucket has a singly linked list or doubly linked list to resolve hash collision.
  - Time complexity: O(1)



# Hash table data structures

## ❖ Hash bucket




```
struct hlist_head {
    struct hlist_node *first;
};
```

- Initializing hash bucket

```
#define HLIST_HEAD_INIT { .first = NULL }
#define HLIST_HEAD(name) struct hlist_head name = {  .first = NULL }
#define INIT_HLIST_HEAD(ptr) ((ptr)->first = NULL)
```

## ❖ Collision list

- Similar to list_head, hlist_node is embedded into a data structure

```
struct hlist_node {
    struct hlist_node *next;
    struct hlist_node **pprev;    /* &prev->next */
};
```

- Initializing collision list

```
static inline void INIT_HLIST_NODE(struct hlist_node *h)
{
    h->next = NULL;
    h->pprev = NULL;
}
```

# Hash table APIs

## ❖ Define a hashtable with 2^bits buckets

```
#define DEFINE_HASHTABLE(name, bits)                            \
    struct hlist_head name[1 << (bits)] =                       \
            { [0 ... ((1 << (bits)) - 1)] = HLIST_HEAD_INIT }
```

## ❖ Initialize a hash table

```
#define hash_init(hashtable) __hash_init(hashtable, HASH_SIZE(hashtable))
```

- **hashtable**: hashtable to be initialized

- **__hash_init:** an internal function for hash_init()

```
static inline void __hash_init(struct hlist_head *ht, unsigned int sz)
{
    unsigned int i;

    for (i = 0; i < sz; i++)
        INIT_HLIST_HEAD(&ht[i]);
}
```

# Hash table APIs

❖ **Insert**

- **hash_add:** add an object to a hashtable

```
#define hash_add(hashtable, node, key)                                \
    hlist_add_head(node, &hashtable[hash_min(key, HASH_BITS(hashtable))])
```

  - **hashtable**: hashtable to add to
  - **node**: the &struct hlist_node of the object to be added
  - **key**: the key of the object to be added

# Hash table APIs

❖ Search **(not safe while deleting entries)**

- **hash_for_each:** iterate over a hashtable

```
#define hash_for_each(name, bkt, obj, member)              \
    for ((bkt) = 0, obj = NULL; obj == NULL && (bkt) < HASH_SIZE(name);\
            (bkt)++)\
        hlist_for_each_entry(obj, &name[bkt], member)
```

  - **name**: hashtable to iterate
  - **bkt**: integer to use as bucket loop cursor
  - **obj**: the type * to use as a loop cursor for each entry
  - **member**: the name of the hlist_node within the struct

- **hash_for_each_possible**: iterate over all possible objects hashing to the same bucket

```
#define hash_for_each_possible(name, obj, member, key)          \
    hlist_for_each_entry(obj, &name[hash_min(key, HASH_BITS(name))], member)
```

  - **name**: hashtable to iterate
  - **obj**: the type * to use as a loop cursor for each entry
  - **member**: the name of the hlist_node within the struct
  - **key**: the key of the objects to iterate over

# Hash table APIs

- **Search (safe for deleting entries)**
  - **hash_for_each_safe:** iterate over a hashtable safe against removal of hash entry

```
#define hash_for_each_safe(name, bkt, tmp, obj, member)         \
    for ((bkt) = 0, obj = NULL; obj == NULL && (bkt) < HASH_SIZE(name);\
            (bkt)++)\
        hlist_for_each_entry_safe(obj, tmp, &name[bkt], member)
```

    - **name**: hashtable to iterate
    - **bkt**: integer to use as bucket loop cursor
    - **tmp**: a &struct used for temporary storage
    - **obj**: the type * to use as a loop cursor for each entry
    - **member**: the name of the hlist_node within the struct
  - **hash_for_each_possible_safe:** iterate over all possible objects hashing to the same bucket safe against removals

```
#define hash_for_each_possible_safe(name, obj, tmp, member, key)    \
    hlist_for_each_entry_safe(obj, tmp,\
        &name[hash_min(key, HASH_BITS(name))], member)
```

    - **name**: hashtable to iterate
    - **obj**: the type * to use as a loop cursor for each entry
    - **tmp**: a &struct used for temporary storage
    - **member**: the name of the hlist_node within the struct
    - **key**: the key of the objects to iterate over

# Hash table APIs

- **Delete**
  - **hash_del:** remove an object from a hashtable

```
static inline void hash_del(struct hlist_node *node)
{
    hlist_del_init(node);
}
```

- **node**: &struct hlist_node of the object to remove

# Hash table example

- Create a simple module to test Hash table

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/hashtable.h>
#include <linux/slab.h> // for kmalloc

#define MY_HASH_BITS 2

int __init hello_module_init(void)
{
    printk("module init\n");

    hashtable_exmaple();

    return 0;
}

void __exit hello_module_cleanup(void)
{
    printk("Bye Module\n");
}

module_init(hello_module_init);
module_exit(hello_module_cleanup);
```

# Hash table example

## ❖ Example Hash table test codes

```
struct mytype {
    u32 key;
    int value;
    struct hlist_node hnode;
};

void hashtable_exmaple(void)
{
    DEFINE_HASHTABLE(my_hash, MY_HASH_BITS);
    hash_init(my_hash);

    /* (key,value) insert */
    int i;
    for (i = 0; i < 10; i++) {
        struct mytype *new = kmalloc(sizeof(struct mytype), GFP_KERNEL);

        new->value = i * 10;
        new->key = i;
        memset(&new->hnode, 0, sizeof(struct hlist_node));

        hash_add(my_hash, &new->hnode, new->key);
    }
    ...
```

*Initialize our hash table*

*Create 10 nodes and insert them into hash table*

# Hash table example

```
    /* circuit every bucket */
    int bkt;
    struct my_node *cur;
    hash_for_each(my_hash, bkt, cur, hnode) {
        printk("(key, value) = (%d, %d) is in bucket[%d]\n",
            cur->key, cur->value, bkt);
    }

    /* 4th bucket search */
    hlist_for_each_entry(cur, &my_hash[3], hnode) {
        printk("(key,value) = (%d, %d) is in bucket[3]\n",
            cur->key, cur->value);
    }

    /* 4th bucket delete */
    struct hlist_node* tmp;
    hlist_for_each_entry_safe(cur, tmp, &my_hash[3], hnode) {
        hash_del(&cur->hnode);
    }
}
```

*Traverse all nodes in the hash table*

*Find every nodes in 4th bucket's collision list*

*Delete every nodes in 4th bucket's collision list*

# Hash table Usage

- **Transparent Hugepage**
  - Finds physically consecutive 4KB pages
  - Remaps consecutive 4KB pages to a 2MB page (huge page)
  - Saves TLB entries and improves memory access performance by reducing TLB miss
  - Maintains per-process memory structure, ***struct mm_struct***

# Design patterns

**of kernel data structures**

# Design patterns of kernel data structures

- **Embedding its pointer structure**
  - **list_head**, **hlist_node**, **rb_node**
  - The programmer has full control of placement of fields in the structure
    - In case they need to put important fields close together to improve cache utilization
  - A structure can easily be on two or more lists quite independently, simply by having multiple **list_head** fields
  - **container_of**, **list_entry** and **rb_entry** are used to get its embedding data structure

# Design patterns of kernel data structures

- **Tool box rather than a complete solution for generic service**
  - Sometimes it is best not to provide a complete solution for a generic service, but rather to provide a suite of tools that can be used to build custom solutions.
    - For example, none of Linux list, hash table, and rbtree provides a search function.
  - You should build your own using given low-level primitives

# Design patterns of kernel data structures

- **Caller locks**
  - When there is any doubt, choose to have the caller take locks rather than the callee.
  - This puts more control in the hands of the client of a function.