

Algorithms - 2 Data Structures

Algorithms - 2 Data Structures

Data Structure

Basic Abstract Data Types (ADTs)

stack

queue

Common data structures

1. Heap

2. Hash

dynamic Set operations:

Hashing and Hash Tables

Universal Hashing:

Perfect Hashing

Consistent Hashing

Bloom-filer

Rabin-Karp Algorithm for pattern Searching

3. Tree

Red-Black Tree

AVL

Augmented Data structure

Dynamic order statistics:

Interval tree

B trees

Applications

Data Structure

DS is a way to store and organize data in order to facilitate access and modifications

Basic Abstract Data Types (ADTs)

A mathematical model of the data methods used to modify and access the data, we don't care the implementation

Including: list, stack, queue, set, dictionary

stack

| PUSH(S,x), POP(S)

queue

ENQUEUE(Q,x), DEQUEUE(Q)

Q.head, Q.tail, Q.length

Priority queue

data with a key

INSERT(P,x), EXTRACT-MAX(P), MAX(P), INCREASE-KEY(P,x,k)

Applications: OS(manage jobs), Graph Algorithms(Dijkstra), arrange events, compression(Huffman encoding), heapsort...

How should we implement the priority queue:

binary heap,...

Common data structures

1. Heap

Binary Heap

structure

array structure, visualized as nearly complete tree(only the left of last layer may be not full)

root = A[1], PARENT(i) = $\frac{i}{2}$, LEFT(i) = $2i$, RIGHT(i) = $2i+1$

attributes: A.length

Theorem: A nearly complete binary tree of n nodes has height $\theta(\log n)$

prove:

a complete binary tree with height h has $\sum_{i=0}^h 2^i = \frac{1-2^{h+1}}{1-2} = 2^{h+1} - 1$ nodes
so for a nearly complete tree $1 + \sum_{i=0}^{h-1} 2^i = 2^h \leq n \leq 2^{h+1} - 1$

Some other properties about the tree:

<https://cs.stackexchange.com/questions/841/proving-a-binary-heap-has-lceil-n-2-rceil-leaves>

Attribute

min binary heap PARENT(A[i]) ≤ A[i]

Operations - e.g. Min-heap

```
HEAP-MINIMUM(A)
    return A[1]

HEAP-EXTRACT-MIN(A)
    if A.heap-size < 1
        error 'heap underflow'
    min = A[1]
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size - 1
    MIN-HEAPIFY(A, 1)
    return min
```

```

MIN-HEAP-INSERT(A, key)
    A.heap-size = A.heap-size + 1
    A[A.heap-size] = inf
    HEAP-DECREASER-KEY(A, A.heap-size, key)

HEAP-DECREASE-KEY(A,i,key)
    if key > A[i]
        error 'new key is larger'
    A[i] = key
    while i > 1 and A[PARENT(i)] > A[i]
        exchange A[i] with A[PARENT(i)]
        i = PARENT(i)

BUID-MIN-HEAP(A)
    A.heap-size = A.length
    for i = floor(A.length/2) downto 1
        MAX-HEAPIFY(A,i)

MIN_HEAPIFY(A,i)
    l = LEFT(i)
    r = RIGHT(i)
    if l <= A.heap-size and A[l] < A[i]
        smallest = l
    else smallest = i
    if r <= A.heap-size and A[r] < A[smallest]
        smallest = r
    if smallest != i:
        exchange A[i] and A[smallest]
        MIN-HEAPIFY(A, smallest)

HEAPSORT(A) #O(nlogn)
    BUILD-MAX-HEAP(A)
    for i = A.length downto 2
        exchange A[1] with A[i]
        A.heap-size == A.heap-size - 1
        MAX-HEAPIFY(A, 1)

```

Time complexity of build a heap: $O(n)$

Amortized analysis: determine worst-case of a sequence of data structure operations

layer	height	depth	nodes	time = relate node*c
1	3	0	$1 = 2^0$	$c \cdot 3 \cdot 2^0$
2	2	1	$2 = 2^1$	$c \cdot 2 \cdot 2^1$
3	1	2	$4 = 2^2$	$c \cdot 1 \cdot 2^0$
4	0	3	$\leq 8 = 2^3$	

h is the height of the tree

$$\begin{aligned}
 T(n) &\leq c \sum_{i=1}^h 2^{h-i} \cdot i \leq c \cdot s^h \sum_{i=1}^h s^{h-i} \cdot i \leq c \cdot 2^{\log n} \sum_{i=1}^h 2^{-i} \cdot i \leq cn \sum_{i=1}^{\inf} 2^{-i} \cdot i \\
 &= cn \sum_{i=1}^{\inf} \left(\frac{1}{2}\right)^i \cdot i = \frac{cn}{2} \sum_{i=1}^{\inf} i \cdot \left(\frac{1}{2}\right)^{i-1} = \frac{cn}{2} \frac{1}{(1-1/2)^2} = O(n)
 \end{aligned}$$

notice that:

$$\begin{aligned}
 \frac{\partial \sum_{i=0}^{\inf} x^i}{\partial x} &= \sum_{i=0}^{\inf} i \cdot x^{i-1} \\
 \frac{\partial \frac{1}{1-x}}{\partial x} &= \frac{1}{(1-x)^2} \\
 \sum_{i=0}^{\inf} x^i &= \frac{1}{1-x}
 \end{aligned}$$

2. Hash

dynamic Set operations:

key(x) = k(satellite data)
SEARCH(S,k)
MINIMUM(S)/MAXIMUM(S)
SUCCESSOR(S,x)
PREDECESSOR(S,x)

dictionary operation

SEARCH(S,k) - returns a pointer x, where x.key = k or nil if not found
INSERT(S,x)
DELETE(S,x)

dictionary data structure

universe of keys U
a data structure that stores a subset $S \subseteq U$
operations:
SEARCH(S,k): given $k \in U$
INSERT(S,x): given x, a pointer to $k \notin S$
DELETE(S,x): given x, a pointer to $k \in S$

Hashing and Hash Tables

a hash table is an array T of size m
a hash function creates an index in the array from an $k \in U, h : U \rightarrow \{0, \dots, m-1\}$ (hashes to slot h(k) in T)

Collision solution - chaining:

Operations:

CHAINED-HASH-INSERT(S,x): insert x at the head of list T[x.key]
CHAINED-HASH-DELETE(S,x): delete x from the list T[x.key]

Assumption:

Simple uniform hashing: any key is equally likely to hash to any of the m slots
for all $i \neq j, \Pr[h(k_i) = h(k_j)] = \frac{1}{m}$

Analysis for unsuccessful search

Let S be the items already in the table where $x \notin S, |S| = n$
$$C_{x,y} = \begin{cases} 1 & h(x) = h(y) \\ 0 & \text{otherwise} \end{cases}$$
$$E[C_{x,y}] = 1 \cdot P[h(x) = h(y)] + 0 \cdot P[h(x) \neq h(y)] = \frac{1}{m}$$
$$C_x = \sum_{y \in S, y \neq x} C_{x,y} = \# \text{ of items that x has a collision with}$$
$$E[C_x] = \sum_{y \neq x} E[C_{x,y}] = \frac{n}{m} = \# \text{ items in the table divided by the size of the table}$$

So, if $n = O(m)$ then $\alpha = n/m = O(m)/m = O(1)$, for insertion, deletion, find

Universal Hashing:

choosing a hash function

using universal Hashing - randomness

Solve:

1. Uniform random

2. fundamental weakness

for any hash function, we can find a set of keys that are hashed to the same spot

Universal Class of Hash Functions:

$H = h_1, h_2, \dots, h_w$ is a universal hash function family if for all $k, k' \in U$, $\Pr_{h \in H}[h(k) = h(k')] \leq \frac{1}{m}$, m is the table size

How to construct a universal Hash functions family:

$H_{pm} = \{h_{a,b}(x) = ((ax + b) \bmod p)\} \mid 1 \leq a \leq p-1, 0 \leq b \leq p-1, \text{ and } p \geq \text{all keys}$

Proof:

Let $k, l \in U$ are two keys, WLOG $k > l, k \neq l$.

1. If $h_{ab}(k) = h_{ab}(l)$ it is because they collided after mod p or mod m - Property of prime numbers
 p is a prime larger than any key, and $p > m$

2. If $h_{ab}(k) = h_{ab}(l)$ it is because they collided after mod m - Modulo arithmetic

let $r = (ak+b) \bmod p$, $s = (al+b) \bmod p$

since $a < p$, $k, l < p \Rightarrow a$ doesn't divide p , $(k-l)$ doesn't divide p

the collision cannot be mod p since o/w $0 \neq s - r \equiv a(k - l) \bmod p$

3. There is a 1-1 correspondence between the (r,s) pairs and the (a,b) pairs

$a = ((r-s)((k-l)^{-1} \bmod p)) \bmod p$

$b = (r-ak) \bmod p$

There are $p(p-1)$ possible pairs (r,s) that $r \neq s$, thus there is a 1-1 correspondence between pairs (a,b) and pairs (r,s) . Thus if a, b is chosen randomly the pair (r,s) is equally likely to be any pair of distinct values modulo p

4. If we randomly choose an (r,s) pair, the probability that they collided mod m is $\leq 1/m$. Thus if I randomly choose an (a, b) pair the probability that $h_{a,b}(k) = h_{a,b}(l)$ is at most $\frac{1}{m}$.

of choice for s such that $r \neq s$ and $s \equiv r$ is at most $\lceil \frac{p}{m} \rceil - 1$

(just list all the possible s that collide with r : $r' = r \bmod m$

$r', r'+m, r'+2m, r'+3m, r'+4m, \dots, r' + (\lfloor \frac{p}{m} \rfloor - 1)m$) are all less than p , $r' + (\lfloor \frac{p}{m} \rfloor - 1)m$ may be less than p

Theorem

If for all $k, k' \in U$, $\Pr_{h \in H}[h(k) = h(k')] \leq \frac{1}{m}$, then for any $S \subseteq U$, for any $x \in U - S$ the expected number of collisions between x and the other elements in S is at most n/m , when $|S|=n$

Proof:

Corollary 1: If for all $k, k' \in U$, $\Pr_{h \in H}[h(k) = h(k')] \leq \frac{1}{m}$, then for any $S \subseteq U$, for any $x \in U - S$ the expected number of comparisons for a successful search is $O(1+n/m)$ and for an unsuccessful search is $O(n/m)$ where $|S|=n$.

Corollary 2: Using universal hashing and collision resolution by chaining in an initially empty table with m slots, it takes expected time $\theta(n)$ to handle any sequence of n INSERT, SEARCH, and DELETE operations containing $O(m)$ INSERT operations where m is the table size

Perfect Hashing

Static

We can do better when keys are static:

Given a fixed set of n keys we can construct a static hash table of size $m = \theta(n)$ such that search takes $\theta(1)$ time in the worst case

Theorem:

Hashing n keys into $m = n^2$ slots using $h \in_R H$ then $E(\# \text{ collisions}) < 1/2$

Proof:

probability that two keys collide is $1/m = 1/n^2$, and # pairs of key = $\binom{n}{2}$

$$E(\# \text{ collisions}) \leq \binom{n}{2} \cdot \frac{1}{n^2} = \frac{n(n-1)}{2} \cdot \frac{1}{n^2} < \frac{1}{2}$$

Corollary:

Probability of no collisions $> 1/2$

Proof: X be a R.V. holding the number of collisions, according to Markov inequality:

$$Pr[X \geq 1] \leq \frac{E[X]}{1} < \frac{1}{2}$$

(FYI: Markov's inequality can refer to Math.md)

To reduce the space: two levels

Theorem:

Let $m=n$ be the size of the hash table at level 1

Let n_j^2 be the size of the hash table at index j

Let h be randomly chosen from a universal set H

Then $Pr[\sum_{j=0}^{m-1} (n_j)^2 \geq 4n] < 1/2$

Proof:

the number of pairs that collide at the first layer:

$$\sum_{x \in S} \sum_{y \in S} C_{x,y} = \sum_{j=0}^{m-1} \sum_{x \in S_j} \sum_{y \in S_j} C_{x,y} = \sum_{j=0}^{m-1} (n_j)^2$$

$$E[\sum_{j=0}^{m-1} n_j^2] = E[\sum_{x \in S} \sum_{y \in S} C_{xy}] = n + \sum_{x \in S} \sum_{y \in S-x} E[C_{xy}] \leq n + n(n-1)/m < 2n$$

Markov's inequality: $Pr[\sum_{i=1}^{m-1} (n_i)^2 \geq 4n] < 2n/4n = 1/2$

dynamic

TBC

Consistent Hashing

TBC

Bloom-filer

is a space-efficient probabilistic data structure for test membership

property:

False positive - increase with n

never generate false negative result

can't delete element

can always adding element with the prize that FP increaser

Operations:

insert(x), lookup(x)

Probability of FP:

m: size of bit array,
k: number of hash functions
n: number of expected elements to be inserted in the filter
$$P = (1 - (1 - \frac{1}{m})^{kn})^k$$

Rabin-Karp Algorithm for pattern Searching

<https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/>

uses to compare two string (should be $O(mn)$ if directly check)

text = "abcccsasdwacbab" patten = "abc"

n = len(text), m = len(pattern)
average and best $O(m+n)$, worst $O(mn)$

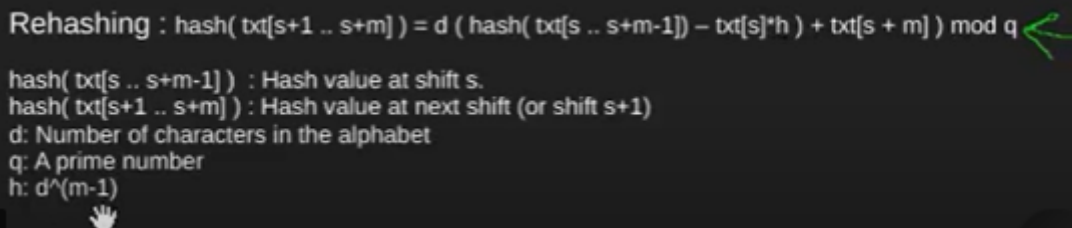
hash(pattern) = x

using sliding window to search in text, every time can compute hash value in $O(1)$: eg Hash("abc"), hash("bcc"), hash("ccc")

Hash function:

hash at the next shift must be efficiently computable $O(1)$ from the current hash value and next character in text

$$\text{hash}(\text{str}) = (\text{str}[0]*d^4 + \text{str}[1]*d^3 + \text{str}[2]*d^2 + \text{str}[3]*d^1 + \text{str}[4])\%q$$



Rehashing : $\text{hash}(\text{txt}[s+1 \dots s+m]) = d (\text{hash}(\text{txt}[s \dots s+m-1]) - \text{txt}[s]*h) + \text{txt}[s+m] \pmod q$

hash(txt[s .. s+m-1]) : Hash value at shift s.
hash(txt[s+1 .. s+m]) : Hash value at next shift (or shift s+1)
d: Number of characters in the alphabet
q: A prime number
h: $d^{(m-1)}$

3. Tree

we may need more operations than dictionary

- maintain order information(for fast retrieval) , support insert/ delete/ search

each node: .key, .satellite_data, .p, .left, .right

Binary Search Tree

the left tree are all smaller, and the right trees are all greater

Self Balancing Tree

Balanced: height is $O(\log n)$

Insertion:

local transformation preserve global property: 1. tree is ordered, 2. perfectly balanced

SEARCH(T,k) $O(\log n)$

INSERT(T,k) $O(\log n)$

DELETE(T,k) $O(\log n)$

2-3 Tree

at most 3 children, all external nodes have same depth(perfectly balanced)

2-node: 1 key 2 links

3-node: 2 keys 3 links

$$\lfloor \log_3 n \rfloor \leq h \leq \lfloor \log_2 n \rfloor$$

Search

Insert

always insert into an existing node

always maintain depth condition

split node if a 3-node:

create a new node, v', push middle key up

2-4 Tree

at most 4 children, all external nodes have same depth(perfectly balanced)

2-node: 1 key 2 links

3-node: 2 keys 3 links

4-node: 3 keys 4 links

$$\lfloor \log_3 n \rfloor \leq h \leq \lfloor \log_2 n \rfloor$$

always insert into an existing node

always maintain depth condition

split node if a 4-node:

create a new node, v', push middle key up

Note: different insert order may cause different structure

<https://stackoverflow.com/questions/38258701/when-would-a-2-3-4-tree-not-have-the-same-structure>

relax the idea that the tree has to be perfectly balanced - RB Tree

Red-Black Tree

is a binary search tree that obeys **5 properties**:

1. Every node is colored red or black
2. The root is black
3. Every leaf(T.nil) is black and doesn't contain any data
4. Both children of a red node are black
5. For every node in the tree, all paths from that node down to a leaf(nil) have the same number of black nodes along the path

Creating a Red-Black BST from a 2-3-4 tree

4-node: one black -> two red

3-node: black -> red have two ways

2-node: black

Creating a 2-3-4 tree from Red-Black BST

merge every red node into its black parent

Height of a node: # edges in a longest path to a leaf

Black height of a node x: bh(x) # black nodes encountered on a path to a leaf, not including the node itself.

Claim: A node with height h has black-height at least h/2

Proof: same as $h \leq 2 * bh$

Claim: The subtree rooted at a node, x, contains at least $2^{bh(x)} - 1$ internal nodes (non nil)

Basis: if x has height 0, it is a leaf(nil node) and $bh(x) = 0$, the subtree rooted at x has 0 internal nodes $2^0 - 1 = 0$

Inductive hypothesis: if x has height h and $b = bh(x)$, then its children have height at most h-1. If the child is red it has black-height $b = bh(x)$, otherwise it has black-height $b-1 = bh(x) - 1$. By the inductive hypothesis each child has at least $2^{bh(x)-1} - 1$ internal nodes. Thus the subtree rooted at x contains at least $2(2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes.

Lemma: A red-black tree with n internal nodes has height at most $2\lg(n+1)$

Proof: Let h be the height, and $b = bh(x)$ be the black height of the root node = x. By the previous two claims $n \geq 2^b - 1 \geq 2^{h/2} - 1$. Thus $n + 1 \geq 2^b \geq 2^{h/2} \implies \lg(n + 1) \geq h/2 \implies h \leq 2\lg(n + 1)$

Corollary: On a red-black tree with n nodes, we can implement dynamic-set operations SEARCH, MINIMUM, MAXIMUM, PREDECCESOR.... In $(\lg n)$

Modify the Tree:

Insert it as a red node, and then handle violation

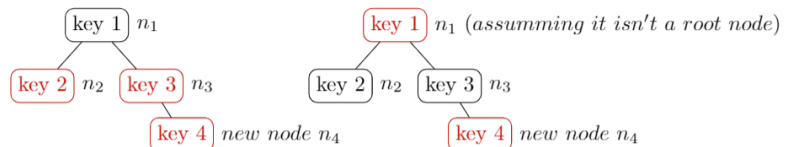
There are three possible violations:

Case1: sibling of parent is red

case 1: double red problem

red-black tree:

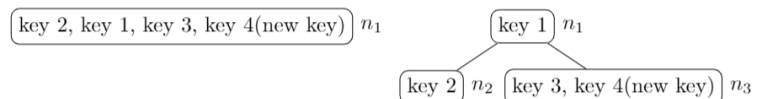
situation: both parent n_2 and sibling of parent n_1 are red nodes



fix: change both parent n_3 and parent's sibling n_2 into black, and change parent's parent n_1 into red, finally set root to be black

2-3-4 tree:

situation: the node to be inserted has already three keys



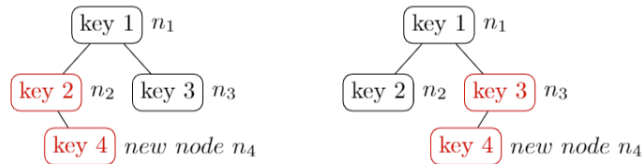
fix: since the origin node has 3 keys, it split into two children, choose the middle one key_1 to be parent.

Case2:

case 2: one black and one red problem

red-black tree:

situation: uncle node is black, the new node and new nodes' parent are in the different side



fix: change rotate new node and new node's parent, to make sure they are in the same side, then it would be case 2.

2-3-4 tree:

situation: the node to be inserted into has 2 keys

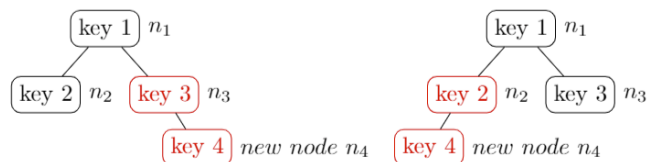


fix: the node to be inserted only two keys, so we can directly insert new key into corresponding node.

Case3:

red-black tree:

situation: sibling of parent is black, the new node and new nodes' parent are in the same side

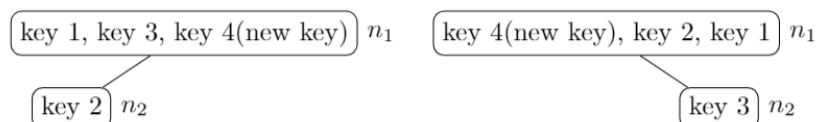


fix: rotate the new node' parent and new node's parent's parent, make new node and new node's parent step into higher level, let new node's parent's parent to be the another children of new node's parent, and than change new node's parent into black, and change new node's parent's parent into red.



2-3-4 tree:

situation: the node to be inserted into has 2 keys



fix: since the origin node has only two keys, just insert new key into the node is fine.

```
#      x                y
#  t1      y      =>      x      t3
#      t2  t3          t1  t2
LEFT-ROTATE(T,x) # no color change
  y = x.right
  # move the t2 from y to x.right
  x.right = y.left
  if y.left != T.nil:
    y.left.p = x
  y.p = x.p
```

```

# attack y to x's original parent
if x.p == T.nil
    T.root = y
elif x == x.p.left
    x.p.left = y
else
    x.p.right = y
# attack x to y
y.left = x
x.p = y

RB-INSERT(T,z)
y = T.nil
x = T.root
# find where to insert
while x!= T.nil
    y = x
    if z.key < x.key
        x = x.left
    else: x = x.right
#create node and attach it
z.p = y
if y == T.nil: # tree T was empty
    T.root = z
else if z.key < y.key
    y.left = z
else y.right = z
# solve color violation
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T,z)

RB-INSERT-FIXUP(T,z)
while z.p.color == RED
    if z.p == z.p.p.right: # parent is a left child
        y = z.p.p.left # y is uncle
        if y.color = RED # case 1: just change the color
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else # case 2 or case 3
            if z == z.p.left # z is a left child case 2
                z = z.p
                RIGHT-ROTATE(T,z)
            # handle case 3
            z.p.color = BLACK
            z.p.p.color = RED
            LEFT-ROTATE(T,z.p.p)
    else (same as then clause with right and left exchanged)
T.root = Black

```

Proof we maintain the Red-Black tree properties:

Loop invariant:

At the start of each iteration of the while loop, node z is red.

There is at most one red-black tree violation either:

a. Z is a red root

b. z and z.p are both red

Initialization:

before we add the new red node z, so if there is violation after insert z, it must be between z and z.parent

Termination: loop terminates when z.p is black. Thus, there is no red-red violation. And finally we set root to be black. => so there will be no violation.

Maintenance:

If we enter the loop, then z and its parent are both red, and no other violation occurs in the red-black tree.

WLOG we only consider the 3 cases, where z.p is the right child of z.p.p let $y = z.p.p.left$ // y is z's uncle. z.p.p is black, since z and z.p are red and there is only one red-red violation

Case1: y is red and z.p.p is black, by making y and z.p black and z.p.p red. Thus black height property is maintained, but we might have created a red-red violation between z.p.p and its parent. Assign $z = z.p.p$

Case2: y is black and z is a left child. Set $z = z.p$ right rotate around z. Now z is now a right child. Both z and z.p are red. Only one case 3 red-red violation.

Case3: y is black z is a right child, make z.p black and z.p.p red left rotate on z.p.p

insertion never performs more than two rotations

<http://staff.ustc.edu.cn/~csl/graduate/algorithms/book6/chap14.htm>

two red nodes in a row, we are done. The body of the while loop is not executed another time, since z.p.p is now black.

What is the running time of RB-INSERT? Since the height of a red-black tree on n nodes is $O(\lg n)$, the call to TREE-INSERT takes $O(\lg n)$ time. The while loop only repeats if case 1 is executed, and then the pointer x moves up the tree. The total number of times the while loop can be executed is therefore $O(\lg n)$. Thus, RB-INSERT takes a total of $O(\lg n)$ time. Interestingly, it never performs more than two rotations, since the while loop terminates if case 2 or case 3 is executed.

Thoughts about duplicate keys:

<https://www.codenong.com/300935/>

<https://stackoverflow.com/questions/36084032/how-to-deal-with-duplicates-in-red-black-trees>

Notice that given n items, we can build heap in $O(n)$ but for red-black tree is $O(n \log n)$:

[https://www.researchgate.net/post/Lower-bound-of-building-a-red-black-tree#:~:text=From%20the%20textbook%2C%20it%20is,takes%20O\(nlogn\)%20time.](https://www.researchgate.net/post/Lower-bound-of-building-a-red-black-tree#:~:text=From%20the%20textbook%2C%20it%20is,takes%20O(nlogn)%20time.)

Just think that when building heap, most nodes have lower height, and for those lower nodes, their operations affect less nodes. (because we combine small heaps - bottom up, $O(1)$ for the deepest layer). For trees, for those lower nodes, their operations affect more nodes. (because we insert node into the tree - top down, $O(\log n)$ for the deepest layer)

AVL

<https://www.cs.auckland.ac.nz/software/AlgAnim/AVL.html>

TBC

Augmented Data structure

e.g. Selection problem

how to find the k th largest item in an array

how to find the rank of an item

Designing augmented ds:

Determining what additional information can be added for new operation

Verifying the additional information can be maintained for the basic modifying operations(insert, delete, rotate,...) on the underlying data structure efficiently

Developing new operation

Dynamic order statistics:

for each node x , add one attribute: $x.size = \#$ of nodes in subtree rooted at x

then: $x.size = x.left.size + x.right.size + 1$

OS-SELECT(T, i) # return a pointer to the node containing the i^{th} smallest key

OS-RANK(T, x) # given a pointer to node x , return the rank of x in the linear in order walk of T

```
x.size = x.left.size + x.right.size + 1

OS-SELECT(x, i)
    r = x.left.size + 1
    if i == r
        return x
    elif i < r
        return OS-SELECT(x.left, i)
    else
        return OS-SELECT(x.right, i - r)

OS-RANK(T, x)
    r = x.left.size + 1
    y = x
    while y != T.root
        if y == y.p.right
            r = r + y.p.left.size + 1
```

Interval tree

Operations:

INTERVAL-INSERT(T, x)

INTERVAL-DELETE(T, x)

INTERVAL-SEARCH(T, i) : return pointer to x in T such that $x.int$ overlaps interval i , any overlapping node in T is OK.

```
INTERVAL-SEARCH(T, i)
    x = T.root
    while x != T.nil and i does not overlap x.int:
        if x.left != T.nil and x.left.max >= i.low
            x = x.left
        else x = x.right
    return x
```

Why we can search like this way:

if $x.left.max \geq i.low \Rightarrow$ if no intervals in the left tree overlaps then no interval in the right subtree overlaps interval i

B trees

application: External memory model

DAM: disk access model, when data is very large RAM isn't a good estimate

a-b Trees:

The keys in each node is in $(a-1, b-1)$ and $2 \leq a \leq (b+1)/2$

max # nodes	max # keys	min # nodes	min # keys
1	b-1	1	1
b	(b-1)b	2	(a-1)2
b^2	$(b-1)b^2$	2a	$(a-1)2a$
...
b^h	$(b-1)b^h$	$2a^{h-1}$	$(a-1)2a^{h-1}$

all leaves are on the same level

insert new key into a leaf

Attribute of a node:

$x.n \in [t-1, 2t-1]$

$x.key_1 \leq x.key_2 \leq x.key_3 \leq x.key_4 \leq \dots \leq x.key_n$

$x.leaf \in \{\text{True}, \text{False}\}$

internal node contains x.n+1 pointers to its children: $x.c_1, x.c_2, x.c_3, \dots, x.c_{n+1}$

Height of B-tree

if $n \geq 1$, then any n-key B-tree T of height h and minimum degree $t \geq 2$ has

$$h \leq \log_t \frac{n+1}{2}$$

proof:

root has at least one key => at least 2 children

other nodes contain at least t-1 keys => contain at least t children

so:

$$n \geq 1 + (t-1) \sum_{i=1}^h 2^{t^i-1} = 1 + 2(t-1) \frac{t^h-1}{t-1} = 2t^h - 1$$

B-Tree operations:

B-Tree-Search()

B-Tree-Insert()

B-Tree-Create(): create an empty root node

root is always in main memory, any node passed as a parameter has already had a disk-read

B-Tree-Search(x,k) # CPU time $O(t \log_t n)$, Disk Access $O(\log_t n)$

```

i = 1
while i <= x.n and k > x.key_i
    i = i + 1
if i <= x.n and k == x.key_i
    return (x,i)
elif x.leaf
    return NIL
else Dist-Read(x,ci)

```

B-Tree-Create(T)

```

x = Allocate-Node()
x.leaf = True
x.n = 0
Disk-Write(x)
T.root(x)

```

B-Tree-Split-Child(x,i) # CPU $O(t)$, Disk Access $O(1)$

Applications

red-black trees: <https://www.quora.com/What-are-some-real-world-applications-of-Red-Black-trees-today>
<https://www.cnblogs.com/yufeng218/p/12465694.html> mysql 引擎的实现
<https://zhuanlan.zhihu.com/p/87124501> b+ 面试题
<https://stackoverflow.com/questions/36084032/how-to-deal-with-duplicates-in-red-black-trees>