Midterm Recitation

CS101 Fall 2024

CS101 Course Team

Nov 2024

Outlines

Outlines

Array

- Representation of polynomial coefficients:
- Increase array capacity:

	Copies per Insertion	Unused Memory
Increase by 1	$n-1$	0
Increase by m	n/m	$m-1$
Increase by a factor of 2	1	n
Increase by a factor of $r>1$	$1/(r-1)$	$(r-1)n$

Linked List

- Given value v and header h, how to find a specific node with value v?
- only by traversing the linked list!
- Time complexity $O(n)$

	Front/1st node	kth node	Back/nth node
Find	$O(1)$	$O(n)$	$O(n)$
Insert After	$O(1)$	$O(1)$	$O(1)$
Replace	$O(1)$	$O(1)$	$O(1)$
Next	$O(1)$	$O(1)$	n/a
Previous	n/a	$O(n)$	$O(n)$

- How to implement a linked list? Array, Stack, Queue
- Doubly Linked List: $O(1)$ previous, $O(n)$ extra space.

Linked List

HW1

Which of the following statements about arrays and linked lists are true?

A. A doubly linked list consumes more memory than a (singly) linked list of the same length.
B. Inserting an element into the middle of an array takes constant time.
C. Reversing a singly linked list takes constant time.
D. Given a pointer to some node in a doubly linked list, we are able to gain access to every node of it.
E. If we implement a queue using the circular array, the minimal memory we need is related to the maximal possible numbers of elements in the queue.

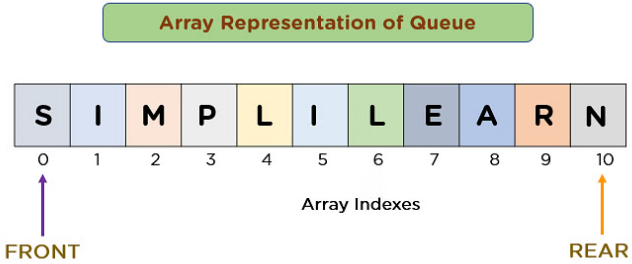
Stack

- LIFO(Last In First Out) Data Structure.
- Basic Operation:
  - **Push**: Adds an element to the top of the stack.
  - **Pop**: Removes the top element from the stack.
  - **Peek**: Returns the top element without removing it.
  - **IsEmpty**: Checks if the stack is empty.
  - **IsFull**: Checks if the stack is full (in case of fixed-size arrays).
- ALL implemented in $O(1)$
- use linked list to implement a stack.
- Check if the given push and pop sequence of the stack is valid or not:

1 3 4 7 5 2 6

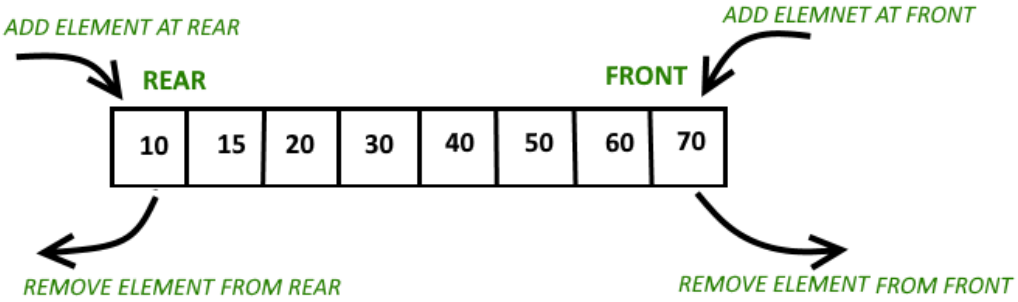
Queue

- FIFO(First In First Out) Data Structure.
- How to use an array to represent a queue?



- uses a linked list to implement a queue.
- Circular Array: Index using a modulo operation.

Deque


ADD ELEMENT AT REAR
REAR
ADD ELEMNET AT FRONT
FRONT
10
15
20
30
40
50
60
70
REMOVE ELEMENT FROM REAR
REMOVE ELEMENT FROM FRONT

Time Complexity

Here is the definition of Landau Symbols without using the limit:

$$
\begin{aligned}
f(n) = \Theta(g(n)) &: \exists c_1, c_2 \in \mathbb{R}^+, \exists n_0, \forall n > n_0, 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n). \\
f(n) = O(g(n)) &: \exists c \in \mathbb{R}^+, \exists n_0, \forall n > n_0, 0 \leq f(n) \leq c \cdot g(n). \\
f(n) = \Omega(g(n)) &: \exists c \in \mathbb{R}^+, \exists n_0, \forall n > n_0, 0 \leq g(n) \leq c \cdot f(n). \\
f(n) = o(g(n)) &: \forall c \in \mathbb{R}^+, \exists n_0, \forall n > n_0, 0 \leq f(n) < c \cdot g(n). \\
f(n) = \omega(g(n)) &: \forall c \in \mathbb{R}^+, \exists n_0, \forall n > n_0, 0 \leq g(n) < c \cdot f(n).
\end{aligned}
$$

Precise Form: $\Theta(), o(), \omega()$

Order:

$$1 < \log n < n < n \log n < n^2 < n^2 \log n < n^3 < 2^n < 3^n < n! < n^n$$

Worst-, Best-, Average-Case

- **Worst Case Analysis**: Upper bound of running time.
  e.g. The worst-case time complexity of the linear search would be $O(n)$.
- **Best Case Analysis**: Lower bound of running time.
  Very rarely used! for linear search, the lower bound should be $\Omega(1)$
- **Average Case Analysis**: take all possible inputs and calculate the computing time for all of the inputs.
  Must know (or predict) the distribution of cases! For linear search, all cases are uniformly distributed.

Example

23 Mid-Term

$f(n) > 0, g(n) > 0, f(n) = \Theta(g(n))$ and a constant $a > 0$, which of the following are **TRUE**?

$A.f(n) + a = \Theta(g(n) + a) \quad B.af(n) = \Theta(ag(n)) \quad C.f(n)^a = \Theta(g(n)^a) \quad D.a^{f(n)} = \Theta(a^{g(n)})$

A. We can prove $\forall n \geq n_0, c_3(g(n) + a) \leq f(n) + a \leq c_4(g(n) + a)$, where $c_4 = 1 + [c_2 > 1](c_2 - 1)$ and c_3 similarly.

$$\frac{f(n)+a}{g(n)+a} \leq \frac{c_2 g(n)+a}{g(n)+a} \leq \begin{cases} \frac{g(n)+a}{g(n)+a} = 1, & c_2 \leq 1 \\ \frac{c_2 g(n)+c_2 a}{g(n)+a} = c_2, & c_2 > 1 \end{cases}$$

B. We can prove $\forall n \geq n_0, c_1 ag(n) \leq af(n) \leq c_2 ag(n)$.

C. We can prove $\forall n \geq n_0, c_1^a g(n)^a \leq f(n)^a \leq c_2^a g(n)^a$.

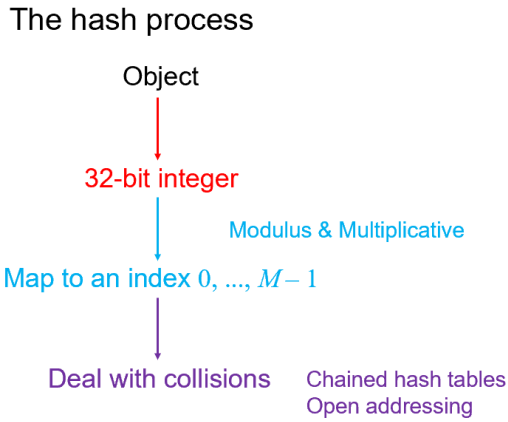
D. Counterexample: $f(n) = n, g(n) = 2n, a = 2$, then $2^n = o(4^n)$.

Outlines

Hash table

- Goal: Hold elements such that all operations has $\Theta(1)$ time complexity. Space complexity is $\Theta(n)$.
- Hash function.
- Map down to $0, 1, \ldots, M-1$.
- Dealing with collisions.

Hash table


The hash process
Object
32-bit integer
Modulus & Multiplicative
Map to an index $0, ..., M-1$
Deal with collisions
Chained hash tables
Open addressing

Hash function

A hash function h should be:

- deterministic: should always return the same value for each object
- should return the same value if two objects are the same

A good hash function should have the following two properties:

- should be fast
- should be 'uniform'

Dealing with collisions

- chained hash tables
- open addressing
  - linear probing
  - quadratic probing

Example for open addressing

23 Mid-term

You are given an open addressing hash table of size $M > 2$ with a uniformly distributed hash function, and we are using linear probing. The probability that both the first and the last slot of the table are filled after the first two insertions is:

Solution: $\frac{3}{M^2}$.

Outlines

Sorting

- All comparison-based sorting algorithms have a lower-bound run-time of $\Omega(n \log n)$.
  - Can be proved by comparison tree.
- An inversion is defined as a pair of entries which are reversed:
  - that is: (a_i, a_j) forms an inversion if $i < j$ but $a_i > a_j$
  - it describes how the list is unsorted
  - a basic fact: swapping two adjacent entries either removes an inversion or introduces an inversion.

Insertion sort

Consider the following observations:

- A list with one element is sorted
- In general, if we have a sorted list of k items, we can insert a new item to create a sorted list of size k + 1

Insertion sort

- An in-place sorting algorithm.
- A stable sorting algorithm.
- Time complexity: $O(n + d)$, where d is the number of inversions.

Bubble sort

Suppose we have an array of data which is unsorted:

- Starting at the front, traverse the array, find the largest item, and move (or bubble) it to the top
- With each subsequent iteration, find the next largest item and bubble it up toward the top of the array
- An in-place sorting algorithm.
- Multiple ways to improve:
  - Flagged bubble sort.
  - Range-limited bubble sort.
  - Alternating bubble sort.
  - They are all worse than insertion sort in practice.

Merge Sort

Core idea: Merge sorted sequences together to sort the whole array.
Recursion relation: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$.
Time complexity: $\Theta(n \log n)$.
Space complexity: $\Theta(n)$. (Not in-place)
Stable: Rely on breaking ties by choosing the front one.

Quick Sort

Core idea: Distinguish those greater than the pivot and less than the pivot.

Recursion relation

Choosing i-th largest as pivot: $T(n) = T(i) + T(n-i) + \Theta(n)$.
Randomized quick-sort: choosing each as a pivot uniformly.
A good approximation of midian: Median-of-three. A deterministic way: Median-of-Median.

Time complexity: $\Theta(n \log n)$ (Average case), $\Theta(n^2)$ (Worst case).
Space complexity: $\Theta(1)$. (In-place).
Not stable: Choosing a non-distinct element as a pivot.

Variant algorithms

- Counting Inversions:
  Trivial idea: Check all (i, j) pairs whether $a_i > a_j$.
  Based on merge-sort: Count "Cross-Inversions" when merging.
- N-th element:
  Trivial idea: Sort then find, takes $\Theta(n \log n)$.
  Based on quick-sort: Consider the number of 2 parts after dividing.

23 Mid-Term

Given an array A of length n, let $f(n)$ be the expected number of comparisons of applying the randomized quick-sort algorithm to sort it and let $g(n)$ be the expected number of inversions in it, then $f(n) = \Omega(g(n))$. (True or False)

Solution: $f(n) = O(n \log n)$ while $g(n) = \Theta(n^2)$

Outlines

Master Theorem

Master Theorem

Given $T(n) = aT(\frac{n}{b}) + f(n)$, $T(1) = 1$.

- $f(n) = o(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a})$
- $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = n^{log_b a} \log n$.
- $f(n) = \Omega(n^{\log_b a + \epsilon})$ with some $\epsilon > 0 \Rightarrow T(n) = \Theta(f(n))$

Example:

1. $T(n) = 2T(\frac{n}{2}) + \Theta(n)$: $a = 2, b = 2, n^{\log_b a} = \Theta(n) \Rightarrow T(n) = \Theta(n \log n)$
2. $T(n) = 7T(\frac{n}{2}) + \Theta(n^2)$: $a = 7, b = 2, n^{\log_b a} = \omega(n^2) \Rightarrow T(n) = \Theta(n^{\log_2 7})$

Recursion Tree & Expansion

Recall

$T(n) = 2T(\frac{n}{2}) + \Theta(n) \Rightarrow \exists$constant c, $T(n) \leq 2T(\frac{n}{2}) + cn$.

$$\begin{aligned} T(n) &\leq 2[2T(\frac{n}{4}) + c\frac{n}{2}] + cn = 4T(\frac{n}{4}) + 2cn \\ &\leq 4[2T(\frac{n}{8}) + c\frac{n}{4}] + 2cn = 8T(\frac{n}{8}) + 3cn. \\ \Rightarrow T(n) &\leq 2^k T(\frac{n}{2^k}) + kcn \end{aligned}$$

Substitute by $k = \log_2 n$, we got $T(n) \leq nT(1) + cn\log_2 n = O(n\log_2 n)$.

Recursion Tree: Visualization Understanding of Expansion.
Time = depth $\times$ (average) time for each layer.

Mathematical Skill

23 Mid-Term

Given two recurrence relation $T(n) = T(0.01n) + T(0.02n) + \Theta(n)$ and $S(n) = S(0.99n) + \Theta(1)$ where $T(0) = S(0) = 0$ and $T(1) = S(1) = 1$, then $T(n) = O(S(n))$. (True or False).

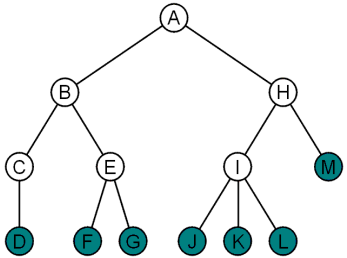
From senior high school: $T(n) = f(T(n-1), \cdots)$
Substitution: $t(m) = T(0.99^{-m})$ i.e. $m = \log_{\frac{1}{0.99}} n$.
Then $t(m) = t(m-1) + \Theta(1) \Rightarrow t(m) = \Theta(m) \Rightarrow T(n) = \Theta(\log n)$.

Other Algorithms using divide and conquer

- Strassen Matrix Multiplication: $O(n^{\log_2 7})$. for $n \times n$ matrix
  (Recent: $n^\omega, \omega < 2.371339$, `https://arxiv.org/abs/2404.16349`)
- Fast Fourier Transform (FFT): $O(n \log n)$ for n-th degree polynomial.

Outlines

Tree



- root node: depth=0
- leaf node, internal node
- degree of a node: Num. of children
- depth of a node, height of a tree
- descendants, ancestors

Tree traversal

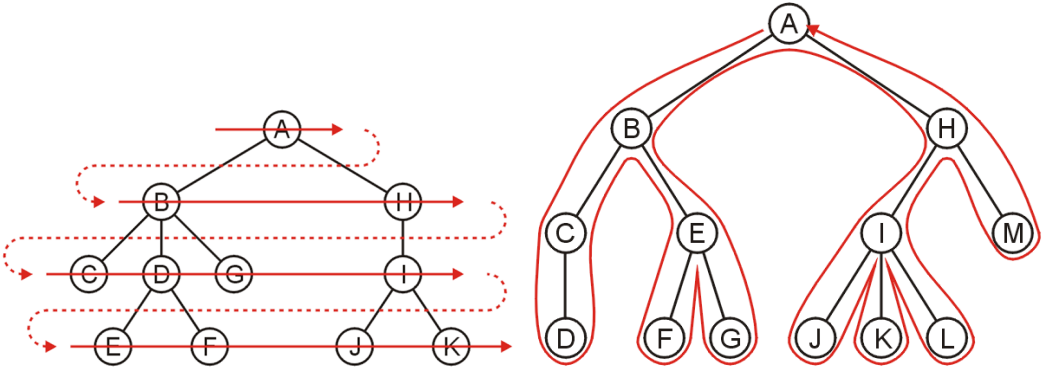
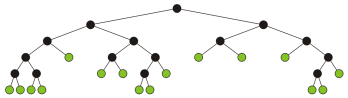
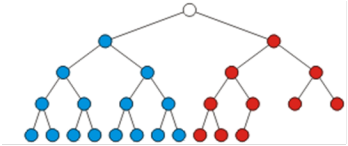
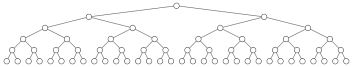


Figure: bfs v.s. dfs

Complexity: $\Theta(n)$ run time and O(n) memory

binary tree

- Binary Tree: at most two children. Array representation: for node i, left son $2i$, right son $2i+1$
- Full Binary Tree: Every node except the leaf nodes has two children.
- Complete Binary Tree: Every level except the last level is completely filled and all the nodes are left justified. (BFS)
- Perfect Binary Tree: Every node except the leaf nodes has two children and every level (the last level too) is completely filled.

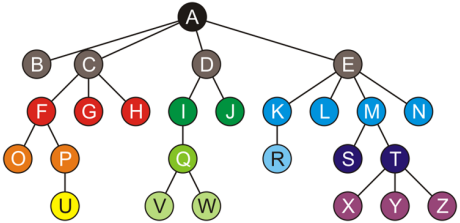




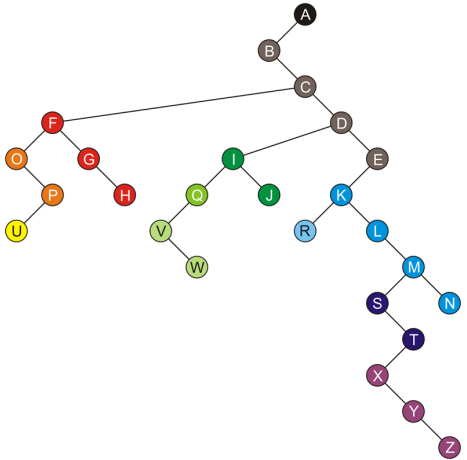
binary tree traversal

- in-order traversal (Left, Root, Right)
- pre-order traversal (Root, Left, Right)
- post-order traversal (Left, Right, Root)
- Breadth-First or Level Order Traversal

Left-child right-sibling

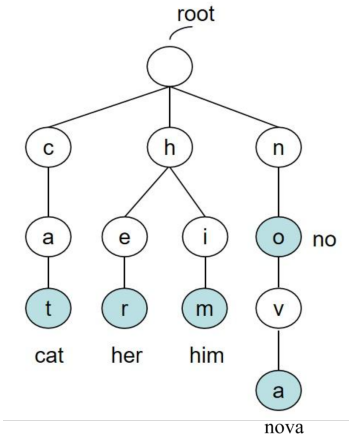
Left: first-child; Right: other children





Trie*

re**trie**val / Prefix Tree



Outlines

Heap

- Min-heap: the value of every node in the subtree is larger than the root.
- max-heap: the value of every node in the subtree is smaller than the root.
- Operation: Top, Pop, Push.
- Pop: Remove objects. rotate every [small] key up.
- Push: add an object as a leaf, then rotate it up.
- **A naive binary heap generally may be incomplete, while a binary heap specifically refers to something that is complete**

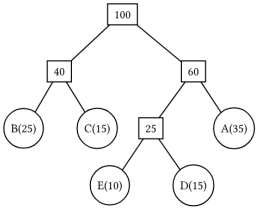
Heap

- Complete trees: the height is limited. so Time complexity will be $O(\log n)$
- Push operation needs to satisfy the rule of CT.
- Pop: copy the last entry in the heap to the root, then rotate it down.
- As it's a complete tree, we can store it using an array.

Huffman tree

motivation: minimize average code length after encoding. (Actually, Huffman coding is the optimal encoding method, which could be proved by greedy.)
Average length:

$$\overline{L} = \sum_{x} p(x)l(x)$$

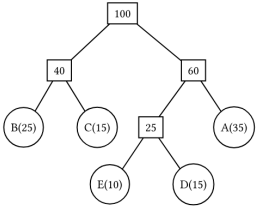


字符	频率	编码
A	35	11
B	25	00
C	15	01
D	15	101
E	10	100

Huffman tree

property:

- Huffman code is not unique (same probability, which side to be 0/1...)
- The character with the longest length is not unique.
- No code is a **prefix** of another node.



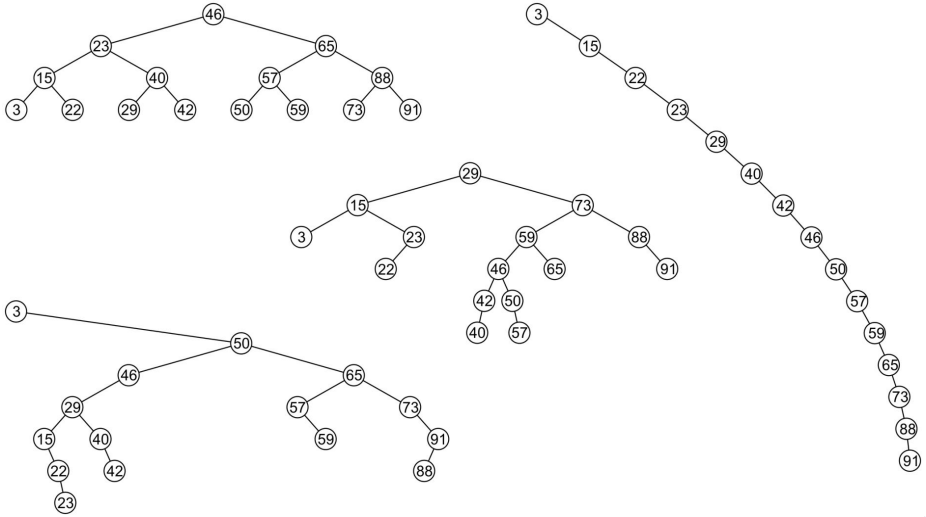
字符	频率	编码
A	35	11
B	25	00
C	15	01
D	15	101
E	10	100

BST

- all objects in the left sub-tree to be less than the object stored in the root node
- all objects in the right sub-tree to be greater than the object in the root object
- the two sub-trees are themselves binary search trees

problem: The height of a BST is $O(n)$, to avoid this problem, we need to add more constraints to the structure, such as AVL-tree.

BST $O(n)$ height



BST Operations

- insert
  Search for the proper position to insert, and then add a new node.
- find
  Similarly with insert, without adding the new node.
- erase
  Search for the proper position, and use the precursor(biggest element among the nodes with smaller value) or successor(smallest element among the nodes with bigger value) of the deleted node to fill in. (In our lecture note, we use the successor for uniform)

BST Complexity

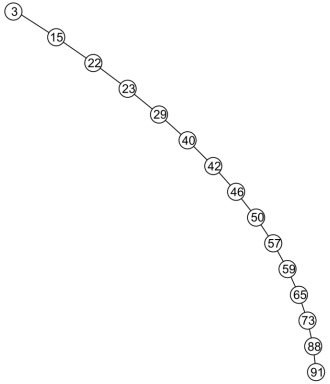
- Find: $O(h)$
- Insert: $O(h)$
- Erase: $O(h)$

If the tree is perfect, these complexities will be $O(\log n)$.
If the tree is closed to a linked list, these complexities will be $O(n)$.

Outlines

Why AVL Tree?

BST has many good properties, but a fatal disadvantage is $h = O(n)$ in the worst case.
To resolve this issue: **AVL tree**, Splay tree, Treap, Red-black tree... to ensure $h = \Theta(\log n)$

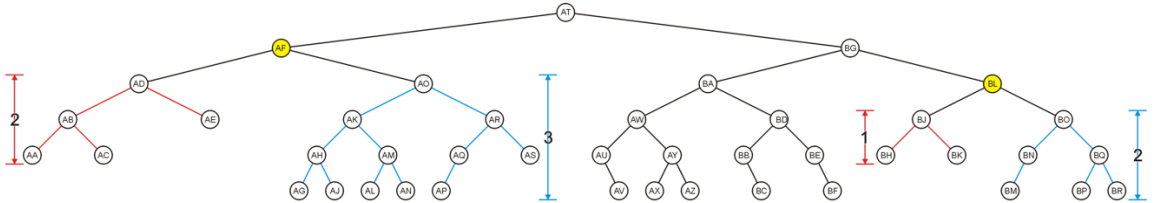


Properties of AVL Tress

Named after **A**delson-**V**elskii and **L**andis.

Balanced:

- The difference in the heights between the left and right sub-trees is at most 1
- Both sub-trees are themselves AVL trees



Height of AVL Tress

An AVL-Tree with height of h, then the number of nodes of the tree:
Upper bound: Perfect binary tree. $n = 2^0 + 2^1 + \ldots + 2^h = 2^{h+1} - 1$
Lower bound:

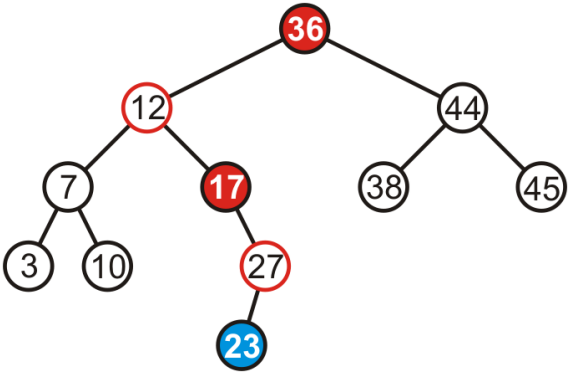
$$F(h) = \begin{cases} 1 & h = 0 \\ 2 & h = 1 \\ F(h-1) + F(h-2) + 1 & h > 1 \end{cases}$$

An empty tree has height –1.
A tree with a single node has a height of 0.
The number of nodes is $\Omega(\alpha^h)$, where $\alpha = \frac{\sqrt{5}+1}{2}$

Operations

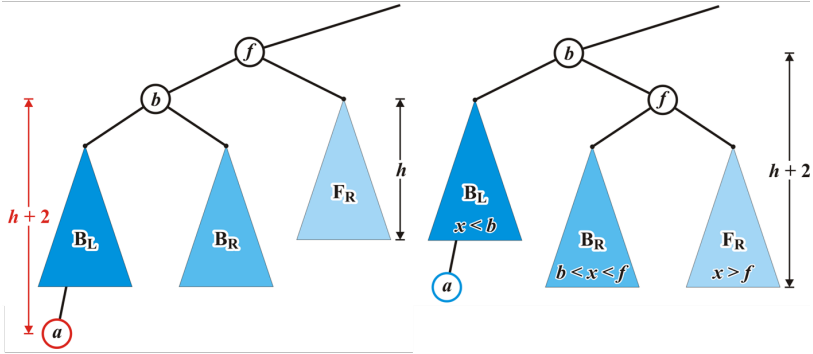
- find: Just do it as BST does.
- insert: Just do it as BST does. However, we may need to correct the imbalance after inserting it.

Insert may cause in-continuous imbalances: e.g. 36 and 17 are imbalanced.



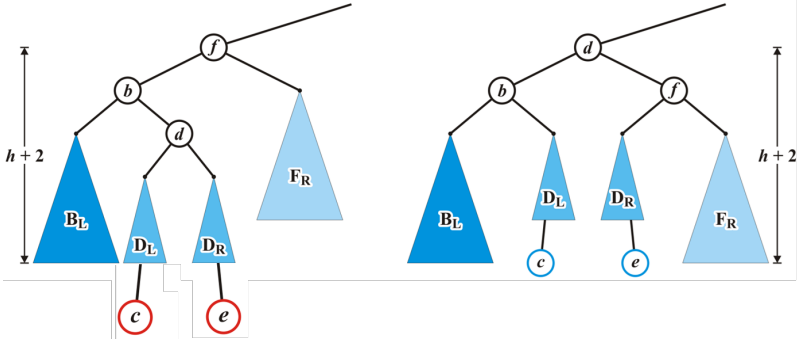
Types of imbalance

- LL(Left-Left)
- RR(Right-Right)



Types of imbalance

- LR(Left-Right)
- RL(Right-Left)



Operations

- erase: Just do it as BST does. However, we may need to correct the imbalance after erasing it.

Erase may cause O(h) imbalances.

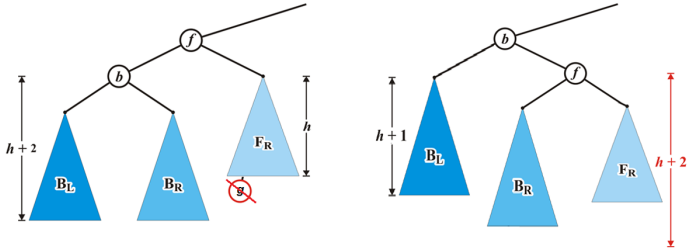
Note: Similar to BST, we can replace the blank with its successor or predecessor after erasing an interior node. (Both are fine, but we mostly use the predecessor by default.)

Types of imbalance

After erasing a node, the two subtrees of b all cause node f imbalanced.
$\Rightarrow$ treat it as LL or RR.

Maintaining Balance: Case 3 (≈Case 1)

This case will only happen after erasing a leaf node, but not after inserting a node. Do the same as in Case 1.



Some Ideas to be Remembereed

1. Both erase and insert can cause more than one balance, while insert **only** needs one rotation to fix them.
2. AVL tree is **not** a complete tree.
3. Pay attention to what the question asks, the time of correction of the insert is not the same as the time of insertion.

Complexity

- find: $\Theta(\log n)$
- insert: $\Theta(\log n)$
- erase: $\Theta(\log n)$
- all kind of corrections: $\Theta(1)$