1. Introduction

Search

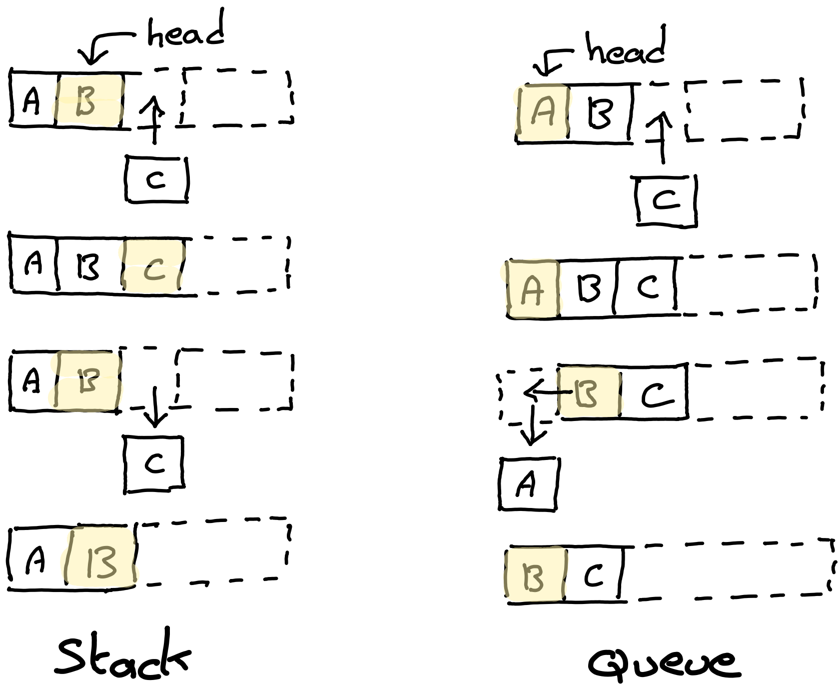
Linear search consists of searching for an element in an array by comparing it to each element in the array in turn. Runtime $O(n)$.

Binary search consists of searching through a sorted array starting with the middle element and subsequently reducing the search space by a factor of 2 until the element is found. Runtime $O(\log_2 n)$.

3. Stacks & Queues

Array implementation

Both stacks and queues can be implemented using arrays, as shown in the examples below.



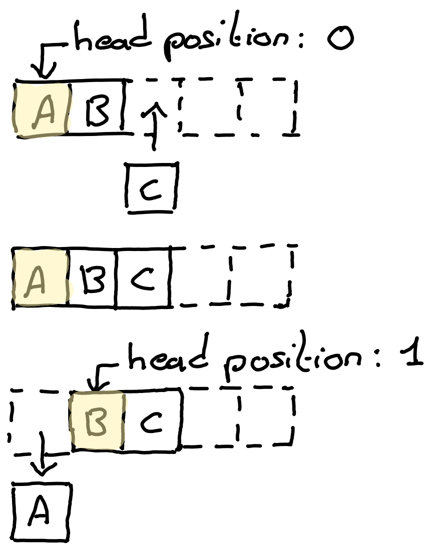
Note that, while the runtime of the push/enqueue operation is $O(1)$ for both, the runtime of the pop/dequeue operation is $O(1)$ for the stack but $O(n)$ for the queue as all remaining elements must be shifted when the head is removed.

Circular queues

The runtime of dequeue operations can be taken down from $O(n)$ to $O(1)$ by using a circular queue.

Instead of shifting all the elements towards the head position on dequeue, circular queues simply shift the head's position.

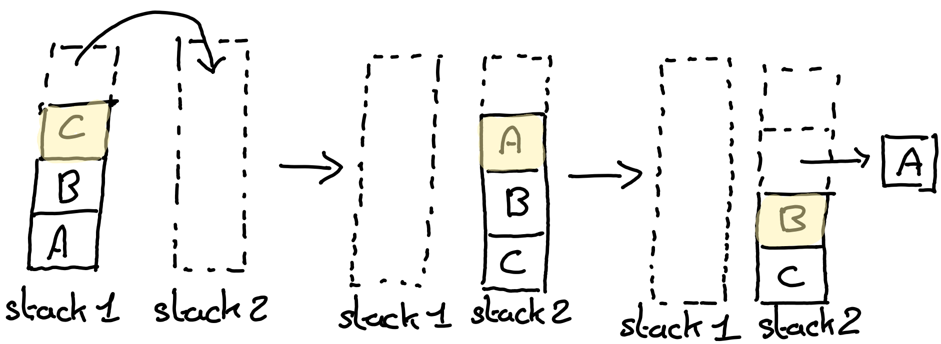
A downside of using circular queues is that their size is fixed in advance and can't be expanded.



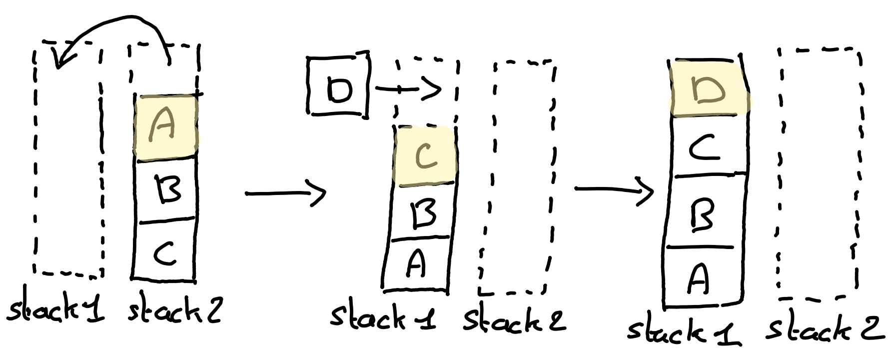
2-stacks implementation of a queue

A queue may be implemented using two stacks:

- stack 1 always contains data in order of insertion, so its head is the last enqueued element
- stack 2 always contains data in order of retrieval, so its head is the first element to be dequeued
- at any given time, at least one stack is empty



dequeue operation

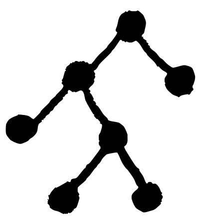


enqueue operation

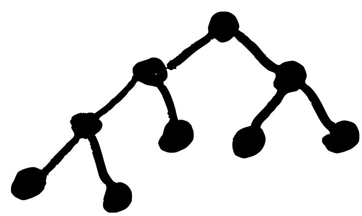
4. Binary trees

A full tree is a binary tree in which each node has exactly zero or two children.

A complete binary tree is one in which every level, except possibly the last, is completely filled and all nodes are as far left as possible.



full tree



complete tree

A tree can be full without being complete and vice-versa (complete without being full).

Traversals

Pre-order traversal:

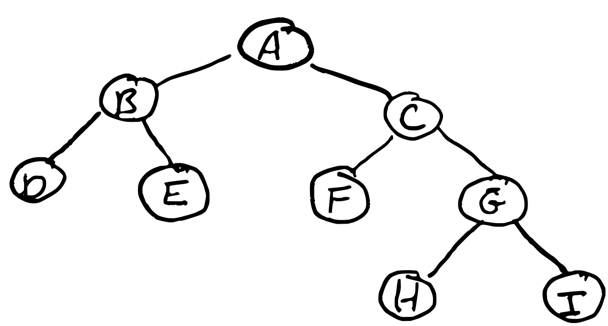
1. process the current node value
2. recursively traverse the left child
3. recursively traverse the right child

In-order traversal:

1. recursively traverse the left child
2. process the current node value
3. recursively traverse the right child

Post-order traversal:

1. recursively traverse the left child
2. recursively traverse the right child
3. process the current node value



Pre-order: ABDECFGHI

In-order: DBEAFCHGI

Post-order: DEBFHIGCA

Search

Depth-first search generally uses a stack in order to keep track of visited nodes.

Breadth-first search (also called level-order traversal) uses a queue.

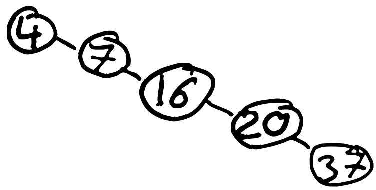
Binary search trees

BSTs are constructed such that, for any node, all children in the left sub-tree contain smaller values while all elements in the right sub-tree contain larger values. This implies BSTs have no duplicate elements.

In-order traversal lets us retrieve all data in a BST in sorted, ascending order.

For optimal performance, a BST should be complete. This is because the time taken to find a node is proportional to the node's depth. If the tree is complete, then its depth will be approximately $\log_2 n$ where n is the number of elements in the tree.

In contrast, a maximally unbalanced tree such as the one depicted below will yield a search runtime of O(n).



Hence, depending on how well balanced a tree is, the search runtime can vary from O(log n) to O(n).

Problem

Which of the following trees will have a better <u>worst-case</u> runtime for searches?

- A balanced BST with depth 20
- An unbalanced BST with depth 18

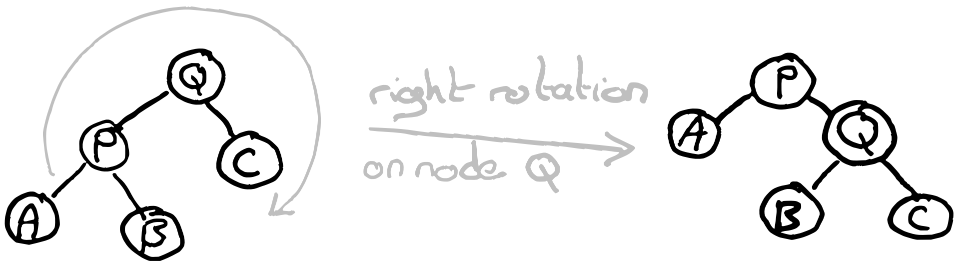
Solution

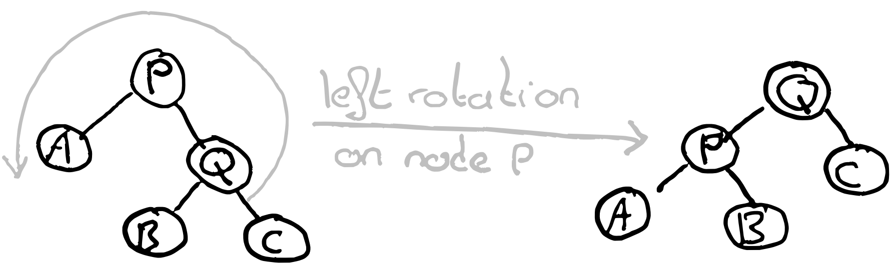
Although the 2nd one is unbalanced, the number of comparisons needed is determined by the depth of the tree (in the <u>worst case</u> it traverses all the way to the bottom of the tree.

Since the 2nd tree has smaller depth than the first, it has a better worst case run time.

Tree rotations

A rotation is an operation on a BST that changes the structure of the tree without affecting the order of the elements (as read by an in-order traversal).
It is used to balance out two branches of different depths.

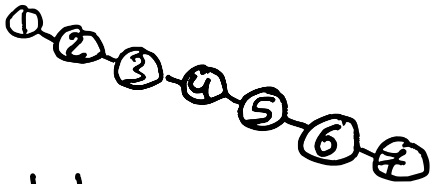




Rotation has no impact on the top node's parents.
As a result, rotations can be used at any level of the tree.

Problem

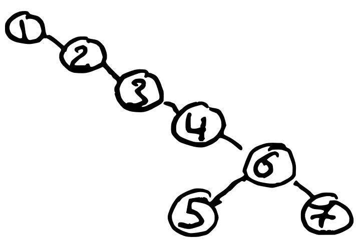
How many rotations are required in order to balance the following maximally unbalanced tree?



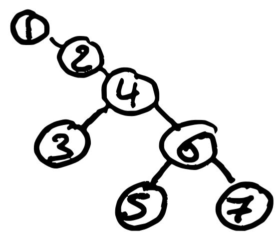
Solution

The following 4 rotations

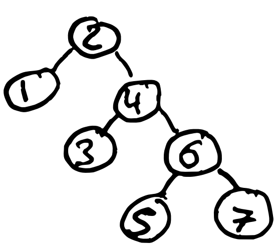
5-6 left



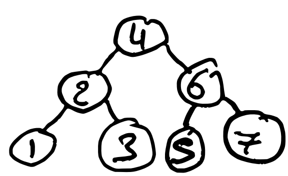
3-4 left



1-2 left



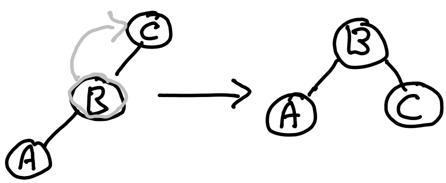
2-4 left



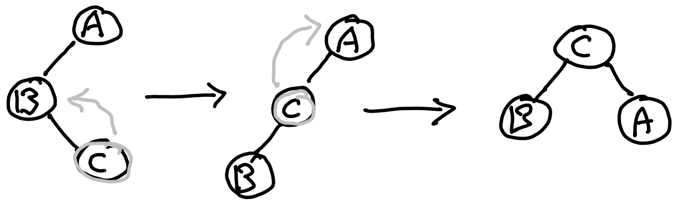
AVL trees

Named after their inventors, Adelson-Velskii and Landis, AVL trees are self-balancing BSTs that automatically apply rotations in the following cases:

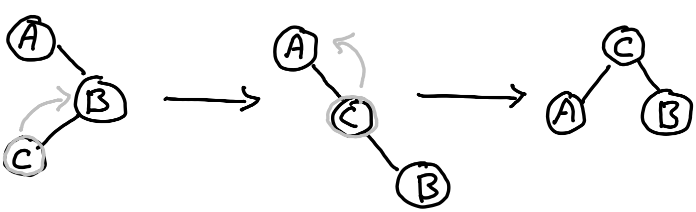
left-left



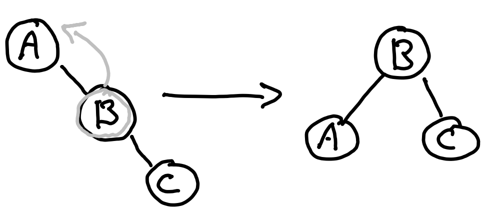
left-right



right-left



right-right



Red black trees

Red black trees are another kind of self-balancing trees. Their lookup is on average slightly slower than the AVL but insertion and deletion are faster.

Each node in the tree is either red or black. Newly inserted nodes are always red. Depending on the color of surrounding nodes, nodes are either rotated or repainted based on a set of constraints.

The guaranteed worst case runtime for search, insertion and deletion is $O(\log n)$, as the tree is guaranteed to have a maximum total height of $2\log(n+1)$.

<u>Algorithm</u>

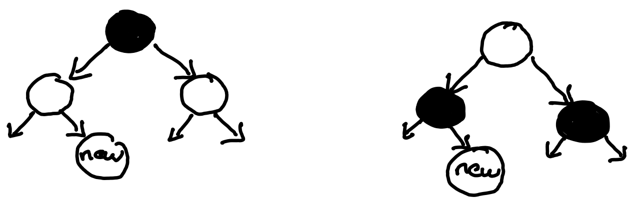
1. Search through the tree to find the correct spot for the new element
2. Put the new node in this spot
3. Paint the new node red
4. Consider the following 4 cases and perform the corresponding action:

Case 1: New node's parent is black

1. Do nothing

Case 2: New node's parent and uncle are red

1. Switch colors as follows

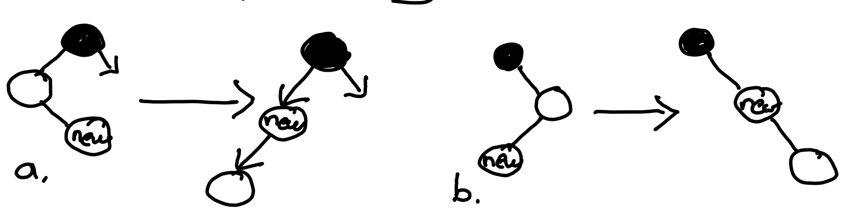


2. If the grandparent now has a red parent, fix by applying the same rule to it.

Case 3: New node's parent is red but uncle is black. The new node's value is between those of its parent and grandparent.

a. If the new node's value is greater than its parent, rotate the parent left

b. If the new node's value is less than its parent, rotate the parent right

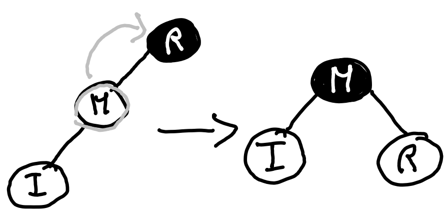


c. Proceed to case 4, swapping the names of new node and its parent.

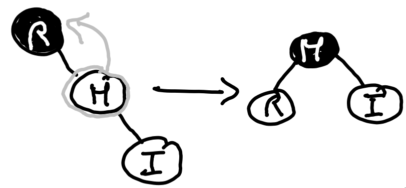
<u>Case 4</u>: New node's parent is red and uncle is black. The parent's value is between those of the new node and its grandparent.

a. If the new node has a value less than the parent, rotate the grandparent right and swap the colors of parent and grandparent.

b. If the new node has a value greater than the parent, rotate the grandparent left and swap their colors.



a.



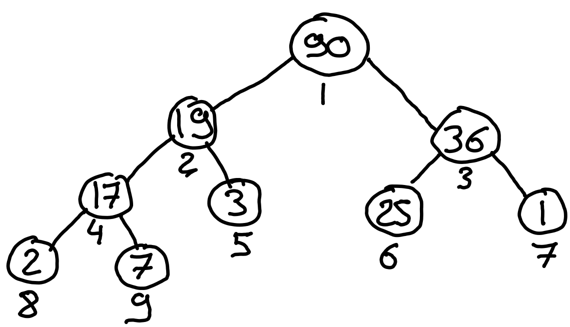
b.

5. Heaps

The binary heap is often considered the best structure for storing priority queues.

Binary heaps are complete binary trees in which the value of each node is always greater than or equal to that of its children.

When sorting the nodes, the children of a node n are numbered 2n and 2n+1.



Insertion is done as follows:

- Insert the new node at the first open position in the heap. Remember the tree must remain complete.
- "Bubble" the number up to the top by repeatedly swapping it with lower priority parents.

Initialisation runtime

Initializing the heap with a single value and repeatedly inserting new nodes using the "bubble up" approach takes $O(n \log_2 n)$ time since we have n nodes and must for each of those traverse down the tree through up to $\log_2 n$ nodes.

It is faster - $O(n)$ - to initialize the heap by placing values in random position before applying the "bubble up" approach, as many nodes will be bubbled up from positions other than the bottom of the tree.

Heapsort