binary search trees: towards AVL trees

binary search in ordered array (look-up table)
binary search in binary search tree (BST)
because the height is crucial for the time complexity of the operations
there are many subclasses of balanced binary search trees
compromise between the optimal and arbitrary binary search tree
examples of balanced BSTs: AVL trees, red-black trees, splay trees

recap definitions

binary tree:
every node has at most 2 successors (empty tree is also a binary tree)

depth of a node $x$:
length (number of edges) of a path from the root to $x$

height of a node $x$:
length of a maximal path from $x$ to a leaf

height of a tree:
height of its root
AVL-tree: definition

binary search tree with in addition balance-property:

for every node
the absolute value of the difference between
the height of the left sub-tree
and the height of the right sub-tree
is at most 1
(take height $-1$ for the empty tree)

AVL-tree: example

the key is in the node, the height of the sub-tree above the node

AVL-tree: height

height of an AVL-tree with $n$ nodes is in $O(\log n)$
(no proof)

operations on AVL-trees

search: exactly as for binary search trees
worst-case time complexity is in $O(h)$ so in $O(\log n)$

add: as for binary search trees, and then restore balance

remove: as for binary search trees, and then restore balance
the four cases of unbalanced nodes after insertion

example: left-left

adding yields left-left unbalanced node:

rebalance using single rotation of nodes with labels 3 and 2:

left-left general

left-left unbalanced node

rebalance using single rotation:
(compare \((ab)c = a(bc)\))

left-left: general intuition of correctness

insertion in A-subtree

\(AB\)-tree balanced before and after insertion

hence if after insertion \(A\)-tree height \(h + 1\) then \(B\)-tree height \(h\)

\(ABC\)-tree balanced before but not after insertion

hence \(C\)-tree height \(h\)

after single rotation: \(ABC\)-subtree balanced and has height as before
symmetry: right-right

right-right unbalanced node
rebalance using single rotation

example: left-right

adding yields left-right unbalanced node

rebalance using double rotation (cf. \(a(bc)d = (ab)(cd)\):

left-right general
left-right unbalanced node

rebalance using double rotation

symmetry: right-left

right-left unbalanced node
rebalance using double rotation
rules for rebalancing

single rotation: left-left and right-right

double rotation: left-right and right-left

inorder of the nodes remains the same

AVL-trees: insertion

step 1: insert new node as for binary search tree

step 2: identify the first unbalanced node on the path from the new node to the root

step 3: rebalance that node (there are 4 possible cases)

we need at most 1 rebalance (single or double rotation) step (why?)

insertion operation is in $O(\log n)$

insertion: correctness

only nodes on the path from the inserted node to the root may become unbalanced.

take the lowest unbalanced node $x$

a single or double rotation restores the balance of $x$

in addition, $x$ gets the same height as before insertion

hence above $x$ the nodes are balanced as before insertion

so the tree is an AVL tree (again)

question: add node with key 0

\begin{center}
\begin{tikzpicture}
  \node (0) at (0,0) {1}
  \node (3) at (1,1) {3}
  \node (2) at (0.5,2) {2}
  \node (4) at (1.5,2) {4}
  \node (5) at (2,1) {5}
  \node (6) at (2.5,1) {6}
  \node (7) at (3,0) {7}
  \node (8) at (2.5,3) {8}
  \node (10) at (3.5,2) {10}
  \node (9) at (3,1) {9}
  \node (11) at (4,0) {11}
  \node (12) at (4.5,0) {12}

  \draw (0) -- (2);
  \draw (2) -- (3);
  \draw (3) -- (5);
  \draw (4) -- (5);
  \draw (5) -- (6);
  \draw (6) -- (7);
  \draw (8) -- (5);
  \draw (10) -- (8);
  \draw (9) -- (10);
  \draw (11) -- (10);
  \draw (12) -- (11);
\end{tikzpicture}
\end{center}
the six cases of unbalanced nodes after removal

example: left
removal yields left unbalance

rebalance using single rotation

AVL-trees: removal

step 1: remove node as for binary search trees
step 2: walk from the removal point upwards and rebalance the first unbalanced node
how to rotate? take the heaviest child
step 3: go back to step 1 if necessary
removal: remarks

example: remove node with key 9

6 possible cases of unbalanced nodes after update
we do not need more rules for rebalancing
after removal we possibly need more than one rebalance step
removal is in $O(\log n)$

overview for ordered dictionaries

<table>
<thead>
<tr>
<th></th>
<th>search</th>
<th>insert</th>
<th>remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>log file</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>lookup table</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>binary search tree</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>AVL-tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

AVL-trees: inventors

Adelson-Velskii (1922-2014) and Landis (1921-1997) in 1962
overview

- AVL-trees
- amortized complexity
- material

stack

- operation push(e) in $O(1)$
- operation pop in $O(1)$
- worst-case cost per operation is in $O(1)$
- worst-case cost for $n$ operations is in $O(n)$

stack with multipop

- operation push(e) in $O(1)$
- operation pop in $O(1)$
- new operation multipop($k$): remove $k$ elements as far as possible

**Algorithm** multipop($S, k$):

```
while not isEmpty($S$) and $k > 0$ do
  pop()
  $k := k - 1$
```

- time taken by multipop is minimum of $k$ and number of elements on $S$

- we consider a sequence of $n$ operations
- worst-case cost for multipop is in $O(n)$
- worst-case cost for an operation is in $O(n)$
- worst-case cost for $n$ operations is in $O(n^2)$
- this analysis is not tight
stack with multipop: aggregate analysis

observe: a single multipop operation may be expensive
observe: an object that is pushed on the stack can be popped at most once
therefore: a sequence of $n$ operations on an initially empty stack is in $O(n)$
hence: the average cost of an operation is in $O(1)$
we take for the amortized cost the average cost

aggregate analysis

we take a worst-case sequence of $n$ operations
we take the average cost of one operation
amortized cost := average cost
all operations have the same amortized cost
no average case analysis, no probability distribution of possible inputs

accounting method: actual cost and amortized cost

we have the actual cost of an operation
we define what we charge for an operation
amortized cost := what we charge
amortized cost intuitively: actual cost plus credit
different operations may have different amortized costs
goal: we can pay for any sequence of operations using amortized costs

accounting method: consider sequences

we require for all sequences of $n$ operations:

$$\sum_{i=1}^{n} \text{amortized-cost}_i \geq \sum_{i=1}^{n} \text{actual-cost}_i$$

so for any sequence of operations the amortized cost bounds the actual cost
so we require that at any 'time' $n$ the total credit is non-negative:

$$\sum_{i=1}^{n} \text{amortized-cost}_i - \sum_{i=1}^{n} \text{actual-cost}_i \geq 0$$
multipop: accounting method

intuition: with push we pay for the later pop

we have actual costs:

actual cost for push and for pop: 1

actual cost for multipop: \( \min(k, s) \)
(removing \( k \) elements from stack with \( s \) elements)

we define amortized costs:

amortized cost for push: 2

amortized cost for pop and for multipop: 0

total amortized cost of a sequence of \( n \) operations in \( O(n) \)

so actual cost of a sequence of \( n \) operations also in \( O(n) \)

binary counter

an array \( A[0 \ldots k-1] \) of bits

lowest-order bit in \( A[0] \)

higher-order bit in \( A[k - 1] \)

so number stored: \( x = \sum_{i=0}^{k-1} A[i] \cdot 2^i \)

initially \( x = 0 \)

incrementing binary counter

time complexity counter

pseudo-code for the operation increment:

```plaintext
Algorithm increment(A):
    i := 0
    while i < A.length and A[i] = 1 do
        A[i] := 0
        i := i + 1
    if i < A.length then
        A[i] := 1
```

we consider a sequence of \( n \) operations

worst-case cost for increment is in \( O(k) \)

worst-case cost for \( n \) operations is in \( O(nk) \)

this analysis is not tight
counter: aggregate analysis

observe: not all bits flip every time increment is called

$A[0]$ flips every time

$A[1]$ flips every other time, or $\lfloor n/2 \rfloor$ times of the $n$ operations

$A[2]$ flips $\lfloor n/4 \rfloor$ times in a sequence of $n$ operations

$A[i]$ flips $\lfloor n/2^i \rfloor$ times in a sequence of $n$ operations

total number of flips in a sequence of $n$ operations:

$$\sum_{i=0}^{k-1} \lfloor n/2^i \rfloor < n \cdot \sum_{i=0}^\infty \frac{1}{2^i} = 2n$$

hence the average cost of increment is in $O(1)$

we use $\sum_{k=0}^\infty x^k = \frac{1}{1-x}$ for $|x| < 1$

counter: accounting method

running time is proportional to number of bits flipped

we have actual costs:

actual cost for flipping bit from 0 to 1 : 1

actual cost for flipping bit from 1 to 0 : 1

we define amortized costs:

amortized cost for flipping bit from 0 to 1: 2

amortized cost for flipping bit from 1 to 0: 0

overview

- AVL-trees
- amortized complexity

material

- material