data structures and algorithms
2016 09 26
lecture 7
overview

- binary search
- trees
- binary search trees
- material
overview

- binary search

- trees

- binary search trees

- material
look-up table

ordered array, that is, with $A[1],\ldots,A[n]$ increasing,
for storing items from a ordered dictionary

operations for searching, adding, deleting

searching by binary search
binary search

search for key $k$ in a sorted array $A[l \ldots r]$

if $l > r$ then return that $k$ is not present

if $l \leq r$ then let $i := \lfloor (l + r)/2 \rfloor$ and compare $m := A[i]$ with $k$

- if $k < m$ then binary search $k$ in $A[l \ldots i - 1]$
- if $k = m$ then return that $k$ is at $i$
- if $k > m$ then binary search $k$ in $A[i + 1 \ldots r]$
binary search: time complexity

recurrence equation:

\[
T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
T\left(\frac{n}{2}\right) + 1 & \text{if } n > 1
\end{cases}
\]

binary search is in $O(\log n)$

for instance searching in array of size 25000 takes 15 steps
alternative

iterative procedure
look-up table

searching is in $O(\log n)$

adding and deleting are in $O(n)$ (why?)

use in case of few updates

otherwise: tree structure
overview

- binary search

- trees

- binary search trees

- material
general trees can be implemented using a linked structure

in the remainder of 7, we restrict attention to binary trees
binary tree: linked implementation

linked data structure with nodes containing

- \( x.key \) from a totally ordered set
- \( x.left \) points to left child of node \( x \)
- \( x.right \) points to right child of node \( x \)
- \( x.p \) points to parent of node \( x \)

in addition, \( T.root \) points to the root of the tree
binary tree: alternative implementation

remember the heap

binary trees can be represented as arrays using the level numbering
tree traversals

how can we visit all nodes in a tree exactly once?

we will mainly focus on binary trees
preorder traversal

visit first the node and next its successors
preorder traversal: pseudo-code

input: node $v$ in a binary tree

Algorithm $\text{preOrder}(v)$:

1. $\text{visit}(v)$
2. if $v.\text{left} \neq \text{nil}$ then
   1. $\text{preOrder}(v.\text{left})$
3. if $v.\text{right} \neq \text{nil}$ then
   1. $\text{preOrder}(v.\text{right})$

$\text{preOrder}$ is in $\mathcal{O}(n)$ with $n$ number of nodes
postorder traversal

visit all successors and next the node itself
postorder traversal: pseudo-code

input: node \( v \) in binary tree

**Algorithm** postOrder(\( v \)):

1. if \( v.left \neq \text{nil} \) then
   postOrder(\( v.left \))
2. if \( v.right \neq \text{nil} \) then
   postOrder(\( v.right \))
3. visit(\( v \))

postOrder is in \( \mathcal{O}(n) \) with \( n \) number of nodes
inorder traversal: pseudo-code

visit left sub-tree, then node itself, then right sub-tree

Algorithm inOrder(v):
  if not v = nil then
    inOrder(v.left)
    print v.key
    inOrder(v.right)
inorder traversal: time complexity

we visit all nodes, so in $\Omega(n)$

recurrence equation with $k$ nodes in the left sub-tree

$$
T(0) = c \\
T(n) = T(k) + T(n - k - 1) + d
$$

we show: $T(n) \leq (c + d) \cdot n + c$

this yields: $T(n)$ in $O(n)$

hence $T(n)$ in $\Theta(n)$
Euler traversal: idea

generic description of traversals
Euler traversal: pseudo-code

instantiate visitLeft, visitBelow, visitRight as desired

**Algorithm** eulerTour(v):

visitLeft(v)

if $v$.left $\neq$ nil then

    eulerTour(v.left)

visitBelow(v)

if $v$.right $\neq$ nil then

    eulerTour(v.right)

visitRight(v)
overview

- binary search
- trees
- binary search trees
- material
binary search tree

binary tree with the

binary search tree property:
for every node $x$ with key $k$ we have:
its left sub-tree contains only keys less than (equal to) $k$
its right sub-tree contains only keys greater than (equal to) $k$
binary search tree: example
inorder traversal

flattens the binary search tree

and yields an ordered sequence

**Algorithm** inOrder(v):

if not v = nil then
    inOrder(v.left)
    print v.key
    inOrder(v.right)
questions

what are the binary search trees with keys 1, 2, 3

give a min-heap that is not a binary search tree

give a binary search tree that is not a min-heap
searching in a binary search tree: example

look for key 4
searching in a binary search tree

input: node $v$ and key $k$

Algorithm treeSearch($v, k$):
   if $v = \text{nil}$ or $k = v.key$ then
      return $v$
   if $k < v.key$ then
      return treeSearch($v.left, k$)
   else
      return treeSearch($v.right, k$)

what is the worst-case running time of treeSearch?
searching in a binary search tree

input: node $v$ and key $k$

**Algorithm** `treeSearch(v, k)`:

1. if $v = \text{nil}$ or $k = v\.key$ then
   return $v$
2. if $k < v\.key$ then
   return `treeSearch(v\.left, k)`
3. else
   return `treeSearch(v\.right, k)`

what is the worst-case running time of `treeSearch`? in $\mathcal{O}(h)$
alternative: iterative version

Algorithm treeSearchIt(v, k):
    while v ≠ nil and k ≠ v.key do
        if k < v.key then
            v := v.left
        else
            v := v.right
    return v
binary search tree: search smallest key as far as possible to the left
binary search tree: search largest key

as far as possible to the right
Algorithm treeMinimum(x):
    while x.left ≠ nil do
        x := x.left
    return x

Algorithm treeMaximum(x):
    while x.right ≠ nil do
        x := x.right
    return x
question

given a node

how to compute the node that is visited next in inorder traversal?
successor: pseudo-code

input: node $x$

**Algorithm** treeSuccessor($x$):

1. if $x.right \neq \text{nil}$ then
   2. return treeMinimum($x.right$)
3. $y := x.p$
4. while $y \neq \text{nil}$ and $x = y.right$ do
   5. $x := y$
   6. $y := y.p$
5. return $y$

what happens if $x$ contains the largest key?

what is the worst-case running time of treeSuccessor?
time complexity: height is crucial

search, minimum, maximum, successor, predecessor

can all be implemented in $O(h)$ with $h$ the height of the BST
adding: example

add node with key 5
Algorithm insert\((T, z)\):

\[
y := \text{nil}\\
x := T.\text{root}\\
\text{while not } x = \text{nil} \text{ do}\\
\quad y = x\\
\quad \text{if } z.\text{key} < x.\text{key} \text{ then}\\
\quad \quad x := x.\text{left}\\
\quad \text{else}\\
\quad \quad x := x.\text{right}\\
\quad z.p := y\\
\quad \text{if } y = \text{nil} \text{ then}\\
\quad \quad T.\text{root} := z\\
\quad \text{else if } z.\text{key} < y.\text{key} \text{ then}\\
\quad \quad y.\text{left} := z\\
\quad \text{else } y.\text{right} := z
\]
example removal easy case
example removal difficult case

![Tree Diagram]

- Before removal:
  - Nodes: 1, 2, 4, 6, 8, 9
  - Node 6 is the target for removal.

- After removal:
  - Nodes: 1, 2, 4, 7, 9
  - Node 6 is replaced by node 7.
removal

remove node $z$ from binary search tree $T$

if $z$ has at most 1 child then transplant

if $z$ has 2 children then take treeMinimum of right subtree transplant that one on the place of $z$
removal: pseudo-code

Algorithm treeDelete(T, z):
    if z.left = nil then
        transplant(T, z, z.right)
    else if z.right = nil then
        transplant(T, z, z.left)
    else
        y := treeMinimum(z.right)
        if y.p ≠ z then
            transplant(T, y, y.right)
            y.right := z.right
            y.right.p := y
            transplant(T, z, y)
        y.left := z.left
        y.left.p := y
Algorithm transplant\((T, u, v)\):

\[
\begin{align*}
    &\text{if } u.p = \text{nil} \text{ then} \\
    &\quad T.root := v \\
    &\text{else if } u = u.p.left \text{ then} \\
    &\quad u.p.left := v \\
    &\text{else} \\
    &\quad u.p.right := v \\
    &\text{if } v \neq \text{nil} \text{ then} \\
    &\quad v.p := u.p
\end{align*}
\]
binary search tree

operations for searching, adding, deleting all in $O(height)$

best case: height is in $O(log \; n)$

worst case: height is in $O(n)$

expected case: height is in $O(log \; n)$ (no proof)
because the height is crucial for the time complexity of the operations
there are many subclasses of balanced binary search tree
compromise between the optimal and arbitrary binary search tree
overview

- binary search
- trees
- binary search trees
- material
material

wiki binary search tree
how many binary trees with $n$ nodes exist?

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

first Catalan numbers:

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452, ...