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 an 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:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(\frac{n}{2}) + 1 & \text{if } n > 1 \end{cases}$$

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

tree

read 10.3

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: pseudo-code

input: node \( v \) in a binary tree

\begin{algorithm}
\textbf{preOrder}(v):
\begin{algorithmic}
\State \text{visit}(v)
\If{\( v.\text{left} \neq \text{nil} \)}
\State \text{preOrder}(v.\text{left})
\EndIf
\If{\( v.\text{right} \neq \text{nil} \)}
\State \text{preOrder}(v.\text{right})
\EndIf
\end{algorithmic}
\end{algorithm}

preOrder is in \( O(n) \) with \( n \) number of nodes
postorder traversal: pseudo-code

input: node \( v \) in binary tree

Algorithm postOrder(\( v \)):

\[
\text{if } v.\text{left} \neq \text{nil} \text{ then} \\
\quad \text{postOrder}(v.\text{left}) \\
\text{if } v.\text{right} \neq \text{nil} \text{ then} \\
\quad \text{postOrder}(v.\text{right}) \\
\quad \text{visit}(v)
\]

postOrder is in \( O(n) \) with \( n \) number of nodes

inorder traversal: pseudo-code

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

Algorithm inOrder(\( v \)):

\[
\text{if not } v = \text{nil then} \\
\quad \text{inOrder}(v.\text{left}) \\
\quad \text{print } v.\text{key} \\
\quad \text{inOrder}(v.\text{right})
\]

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 ≠ nil then
    eulerTour(v.left)
visitBelow(v)
if v.right ≠ nil then
    eulerTour(v.right)
visitRight(v)

binary search tree

binary tree with the

two 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)

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 = 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? in $O(h)$
alternative: iterative version

**Algorithm** treeSearchIt\((v, k)\):

1. while \(v \neq \text{nil} \) and \(k \neq v\text{.key}\) do
2.   if \(k < v\text{.key}\) then
3.     \(v := v\text{.left}\)
4.   else
5.     \(v := v\text{.right}\)
6. return \(v\)

binary search tree: search smallest key

as far as possible to the left

minimum and maximum: pseudo-code

**Algorithm** treeMinimum\((x)\):

1. while \(x\text{.left} \neq \text{nil}\) do
2.   \(x := x\text{.left}\)
3. return \(x\)

**Algorithm** treeMaximum\((x)\):

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

adding: example

add node with key 5

search, minimum, maximum, successor, predecessor

can all be implemented in $O(h)$ with $h$ the height of the BST
Algorithm insert($T, z$):
   $y := \text{nil}$
   $x := T.\text{root}$
   \textbf{while not} $x = \text{nil}$ \textbf{do}
      $y := x$
      \textbf{if} $z.\text{key} < x.\text{key}$ \textbf{then}
         $x := x.\text{left}$
      \textbf{else}
         $x := x.\text{right}$
   $z.p := y$
   \textbf{if} $y = \text{nil}$ \textbf{then}
      $T.\text{root} := z$
   \textbf{else if} $z.\text{key} < y.\text{key}$ \textbf{then}
      $y.\text{left} := z$
   \textbf{else} $y.\text{right} := z$

example removal easy case

example removal difficult case

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

and pseudo-code for transplant

Algorithm transplant(T, u, v):
    if u.p = nil then
        T.root := v
    else if u = u.p.left then
        u.p.left := v
    else
        u.p.right := v
    if v ≠ nil then
        v.p := u.p

binary search tree

operations for searching, adding, deleting all in \(O(\text{height})\)

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)

binary search trees: further improvements

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
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, \ldots